Ioana Cristina PLAJER
Mihai Bogdan PETRE
Alexandra BĂICOIANU
Iulian POPA

ALGORITMI FUNDAMENTALI ȘI STRUCTURI DE DATE

Curs practic pentru studenți

Editura Universității Transilvania din Brașov
2026

EDITURA UNIVERSITĂȚII TRANSILVANIA DIN BRAȘOV

Adresa:	Str. Iuliu Maniu nr. 41A
	500091 Brașov
Tel.:	0268 476 050
Fax:	0268 476 051
E-mail:	editura@unitbv.ro

Editură recunoscută CNCSIS, cod 81

ISBN 978-606-19-1857-7 (e-book)

Copyright © Autorii, 2026

Lucrarea a fost avizată de Consiliul Departamentului de Matematică și Informatică, Facultatea de Matematică și Informatică a Universității Transilvania din Brașov.

Cuprins

Introducere **9**

I Fundamente algoritmice și STL 12

1 Biblioteca STL **13**

1.1 Containere simple 14

1.2 Containere secvențiale 18

1.3 Iteratori 24

2 Noțiuni de complexitate **26**

2.1 Aspecte teoretice 26

2.1.1 Analiza complexității. Notații asimptotice 27

2.1.2 Analiza complexității de timp 36

2.1.3 Clasele de complexitate fundamentale 37

2.1.4 Analiza complexității de spațiu 39

2.1.5 Complexitatea relațiilor de recurență 41

2.2 Probleme rezolvate 43

2.3 Probleme propuse 47

3 Pointeri. Alocarea memoriei. Fișiere **51**

3.1 Aspecte teoretice 51

3.1.1 Pointeri în C++ 51

3.1.2 Alocarea statică a memoriei 54

3.1.3 Alocarea dinamică a memoriei 55

3.1.4 Fișiere text 56

3.2 Probleme propuse - Pointeri. Alocare memorie 63

4 Funcții **70**

4.1 Aspecte teoretice 70
4.1.1 Declarația unei funcții 71
4.1.2 Apelarea unei funcții 72
4.1.3 Clasificarea funcțiilor 73
4.1.4 Transmiterea parametrilor 75
4.1.5 (*) Parametrii default 78
4.1.6 (*) Supraîncărcarea funcțiilor 80
4.1.7 (*) Funcții șablon (*template)* 83
4.1.8 (*) Funcții lambda 86
4.1.9 Funcții recursive 91
4.2 Probleme rezolvate 95
4.3 Probleme propuse - Funcții simple 104
4.4 Probleme propuse - Funcții recursive 109

II Structuri de date fundamentale și algoritmi simpli 113

5 Tablouri unidimensionale 114
5.1 Aspecte teoretice 114
5.2 Probleme rezolvate 124
5.3 Probleme propuse - Vectori simpli 133
5.4 Probleme propuse - Vectori cu structuri / obiecte 138

6 Tablouri bidimensionale 143
6.1 Aspecte teoretice 143
6.1.1 Alocarea statică a tablourilor bidimensionale 144
6.1.2 Alocarea dinamică a tablourilor bidimensionale 149
6.1.3 Matrice pătratice 157
6.2 Probleme rezolvate 158
6.3 Probleme propuse - Matrici simple 167
6.4 Probleme propuse - Matrici cu structuri / obiecte 172

7 Structuri de date elementare 176

7.1 Aspecte teoretice 176
7.1.1 Liste înlănțuite 176
7.1.2 Liste în STL 181
7.1.3 Stive și cozi 184
7.1.4 Stive și cozi în STL 185
7.2 Probleme rezolvate 187
7.3 Probleme propuse 209

8 Căutări și sortări simple 215
8.1 Aspecte teoretice 215
8.1.1 Căutări 215
8.1.2 Sortări 225
8.2 Probleme rezolvate 234
8.3 Probleme propuse 243

III Algoritmi eficienți și structuri de prioritizare. Strategii de selecție 245

9 Tabele de dispersie 246
9.1 Aspecte teoretice 246
9.1.1 Funcții de dispersie 246
9.1.2 Hashing pentru chei de tip string - algoritmul FNV-1a 248
9.1.3 Rezolvarea coliziunilor 249
9.1.4 Factor de încărcare și complexitatea operațiilor 252
9.1.5 Tabele de dispersie în STL 253
9.2 Probleme rezolvate 257
9.2.1 Probleme teoretice 257
9.2.2 Probleme aplicative 260
9.3 Probleme propuse 277
9.3.1 Probleme teoretice 277
9.3.2 Probleme aplicative 278

10 Divide et Impera 281

10.1 Aspecte teoretice 281
10.1.1 Etapele metodei 281
10.1.2 Exemplu ilustrativ 282
10.1.3 Complexitatea algoritmilor de tip Divide et Impera 283
10.2 Algoritmi clasici 283
10.2.1 Căutarea binară 284
10.2.2 Quick Sort 285
10.2.3 Merge Sort 290
10.2.4 Algoritmul Karatsuba 293
10.2.5 Șirul lui Fibonacci 299
10.2.6 Turnurile din Hanoi 302
10.2.7 Algoritmul lui Strassen pentru înmulțirea matricilor 304
10.3 Probleme rezolvate 310
10.4 Probleme propuse 317

IV Arbori - Heap-uri 321

11 Arbori. Arbori binari 322
11.1 Aspecte teoretice 322
11.1.1 Reprezentarea în memorie a arborilor 323
11.1.2 Parcurgerea arborilor binari 324
11.2 Probleme rezolvate 325
11.2.1 Probleme teoretice 326
11.2.2 Probleme aplicative 331
11.3 Probleme propuse 347
11.3.1 Probleme teoretice 347
11.3.2 Probleme aplicative 347

12 Heap-uri binare. Cozi de prioritate 353
12.1 Aspecte teoretice 353
12.1.1 Operațiile principale 354

12.1.2 Cozi de prioritate în STL 355
12.1.3 Sortarea prin heap (Heap Sort) 357
12.2 Probleme rezolvate 360
12.2.1 Probleme teoretice 360
12.2.2 Probleme aplicative 364
12.3 Probleme propuse 380
12.3.1 Probleme teoretice 380
12.3.2 Probleme aplicative 381

13 Arbori binari de căutare 385
13.1 Aspecte teoretice 385
13.1.1 Operații principale 387
13.1.2 Arbori AVL 390
13.1.3 Arbori Roșu-Negru 394
13.2 Arbori echilibrați în STL 400
13.3 Probleme rezolvate 406
13.3.1 Probleme teoretice 406
13.3.2 Probleme aplicative 410
13.4 Probleme propuse 421
13.4.1 Probleme teoretice 421
13.4.2 Probleme aplicative 423

V Paradigme algoritmice avansate 428

14 Greedy 429
14.1 Aspecte teoretice 429
14.1.1 Fundamente teoretice detaliate 430
14.1.2 Algoritmul general Greedy explicat 431
14.1.3 Verificarea corectitudinii unui algoritm Greedy 431
14.2 Algoritmi clasici 433
14.2.1 Problema spectacolelor 433

14.2.2 Problema rucsacului 434
14.2.3 Problema planificării proiectelor cu termene limită 437
14.2.4 Concluzii 438
14.3 Probleme rezolvate 439
14.4 Probleme propuse 446

15 Programare Dinamică **451**
15.1 Aspecte teoretice 451
15.1.1 Fundamente teoretice detaliate 451
15.1.2 Tehnica memoizării 452
15.1.3 Abordări în Programarea Dinamică 454
15.2 Algoritmi clasici 457
15.2.1 Problema celui mai lung subșir crescător 457
15.2.2 Problema triunghiului 459
15.2.3 Problema matricei binare maximale 461
15.2.4 Dynamic Time Warping (DTW) 464
15.2.5 Concluzii 474
15.3 Probleme rezolvate 475
15.4 Probleme propuse 486

Bibliografie **490**

Introducere

Algoritmica și structurile de date reprezintă fundamentul esențial pentru formarea solidă a cunoștințelor și abilităților unui programator. În procesul de rezolvare a unei probleme, primul pas constă în identificarea datelor de intrare (informațiile disponibile) și a datelor de ieșire (scopul urmărit). Pe baza acestora, este necesară conceperea unei secvențe de pași logici care, plecând de la intrare, conduc la obținerea rezultatului dorit. Această succesiune formează ceea ce numim un *algoritm*.

Scopul algoritmicii este de a-i învăța pe viitorii programatori cum să identifice corect elementele și pașii unui algoritm, cum să trateze cazurile limită (acele situații rare, dar critice), ce structuri de programare sunt necesare și cum să evalueze eficiența soluțiilor propuse. Pentru o problemă dată, pot exista mai multe soluții posibile, diferite în funcție de complexitate, o măsură a eficienței algoritmului din punctul de vedere al timpului de execuție și al memoriei utilizate. Toate aceste concepte sunt parte integrantă din cursul de anul I destinat specializărilor de informatică intitulat *Algoritmi fundamentali*, și sunt detaliate în materialele didactice aferente.

Pentru a implementa un algoritm, este nevoie de obiecte în care să fie stocate datele de intrare, valorile intermediare și rezultatele finale. Aceste obiecte sunt cunoscute sub denumirea de *structuri de date*. Ele sunt prezentate detaliat în cursul *Structuri de date*, adresat studenților din anul I. Structurile de date permit stocarea și manipularea eficientă a informației, esențială pentru rezolvarea problemelor într-un mod automatizat. Cele mai multe limbaje moderne de programare oferă biblioteci standard care includ implementări predefinite ale acestor structuri, precum și ale unor algoritmi fundamentali (sortări, căutări etc.), reducând astfel timpul necesar dezvoltării aplicațiilor.

Acest îndrumar de laborator este destinat în principal studenților din anul I ai specializărilor de informatică, în special celor de la Facultatea de Matematică și Informatică a Universității „Transilvania” din Brașov, dar poate fi util și altor persoane interesate să își formeze o bază solidă în programare. Având în vedere că limbajele de programare predate în anul I la această facultate sunt C și C++, toate exemplele și problemele prezentate în acest material sunt rezolvate în C++.

În primul capitol este introdusă biblioteca C++ *Standard Template Library (STL)*, din care sunt prezentate o serie de containere (implementări ale unor structuri de date) și algoritmi uzuali.

Începând cu al doilea capitol, fiecare secțiune este organizată astfel: o parte teoretică ce rezumă conceptele de bază, algoritmii și structurile de date relevante; o secțiune de probleme rezolvate, însoțite de explicații și implementări C++; urmate de o serie de probleme propuse pentru studiu individual. Acest format are rolul de a sprijini înțelegerea practică a conceptelor și de a încuraja dezvoltarea abilităților de analiză și implementare algoritmică.

Procesul de rezolvare a problemelor

Rezolvarea unei probleme în programare reprezintă un proces sistematic ce constă din mai multe etape esențiale:

1. Analiza problemei
    - Identificarea datelor de intrare (input)
    - Definirea clară a rezultatelor așteptate (output)
    - Înțelegerea profundă a cerințelor și constrângerilor
2. Proiectarea algoritmului
    - Divizarea problemei în subprobleme (modularizare)
    - Aplicarea principiului KISS (**K**eep **I**t **S**imple, **S**tupid)
    - Crearea de subalgoritmi specializați pentru fiecare acțiune
    - Alegerea structurilor de date adecvate
3. Implementarea
    - Codificarea algoritmului în limbajul C++
    - Respectarea standardelor de codare
    - Documentarea codului

4. Justificarea soluției propuse
    - Justificarea pașilor considerați
    - Testare cu diverse seturi de date:
        - Cazuri normale
        - Cazuri limită
        - Situații de excepție
    - Testare individuală a subprogramelor (unit testing)
    - Demonstrații formale de corectitudine (unde este posibil)
5. Analiza complexității
    - Evaluarea complexității temporale
    - Evaluarea complexității spațiale
    - Optimizarea algoritmului (dacă este necesar)

Notă: Pentru principiul KISS, consultați `https://www.interaction-design.org/literature/article/kiss-keep-it-simple-stupid-a-design-principle`

Acestea sunt etapele pe care le vom urmări de fiecare dată în rezolvarea problemelor.

Partea I

Fundamente algoritmice și STL

Capitolul 1

Biblioteca STL

Biblioteca STL din C++ conține o serie de clase șablon - *template* - care implementează structuri de date uzuale, precum stive, cozi, liste înlănțuite, cozi de prioritate și altele, dar și algoritmi, precum sortare și căutare, funcții și iteratori. Pentru implementarea structurilor de date se utilizează containere, adaptoare de containere și iteratori [1][2].

Șabloanele permit utilizarea structurilor de date indiferent de tipul datelor pe care dorim să le memorăm.

Containerele sunt cele care stochează efectiv informația, de exemplu tablourile - *arrays* - sau listele înlănțuite, iar **adaptoarele de containere** au la bază un container simplu, cu anumite reguli de acces specifice (adaptează un container la cerințele problemei). Astfel de adaptoare sunt de exemplu stivele [3].

Iteratorii permit o iterare uniformă prin elementele structurilor, fără ca utilizatorul să fie nevoit să cunoască modul de acces al elementelor. Iteratorii sunt reprezentați prin pointeri către elemente ale structurii. Cei mai importanți iteratori sunt *begin()* și *end()*, care permit accesul la începutul, respectiv după ultimul element al structurii. Iterarea se face apoi cu ajutorul operatorilor de incrementare $++$ și decrementare $--$[4].

Algoritmii implementați în STL au ca parametri containere din STL și operează pe acestea. În figura 1.1 sunt reprezentate tipurile principale de containere implementate în STL [5][6].

Prezentăm în continuare containerele simple și containerele secvențiale. Apoi, la fiecare capitol care descrie o structură implementată și în STL, va fi prezentata pe scurt clasa corespunzătoare, împreună cu exemple de utilizare.

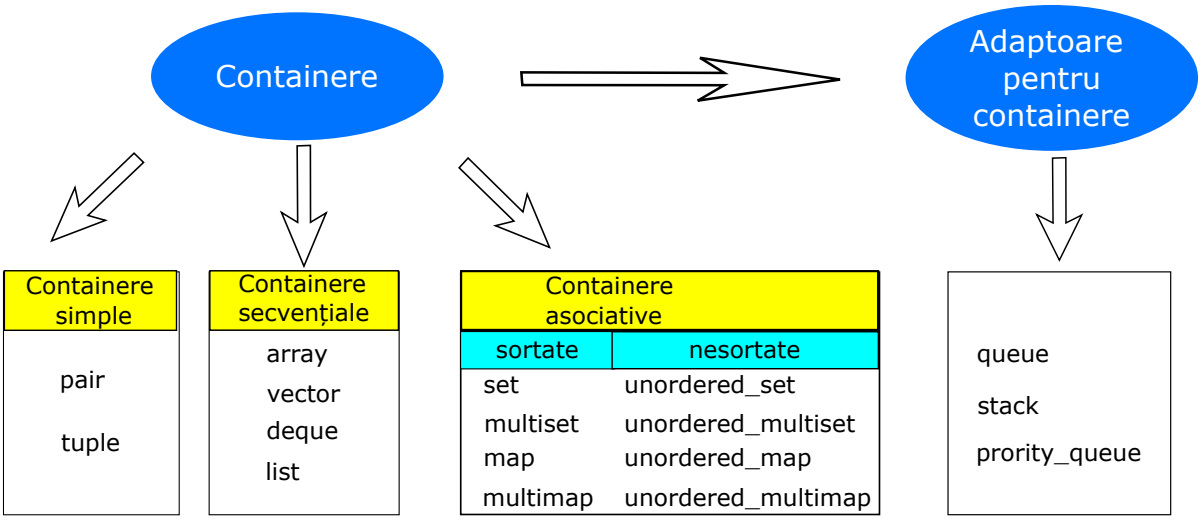


Figura 1.1: Containerele principale din STL

1.1 Containere simple

Containerele simple sunt clase șablon care permit gruparea unui număr finit, relativ mic de variabile de tipuri diferite într-o singură entitate, simplificând astfel managementul acestora în cadrul unei aplicații. În biblioteca STL sunt prezente două astfel de elemente: **pair** și **tuple**.

Tuple

```
template <class T1, class T2, class T3, ..> struct tuple;
```

Această clasă șablon (*template*) definește un container care grupează două sau mai multe valori care pot fi de orice tip într-o singură variabilă, asemenea unei structuri (*struct*). Dacă în cazul structurii fiecare câmp e definit prin tip și un nume, aici variabilele care formează tuplul sunt definite de tip și de locul acestor variabile în cadrul grupării.

Pentru declararea unei variabile de acest tip este necesară specificarea tipurilor de date ce urmează să fie reținute în tuplu. Odată cu declararea variabilei se poate realiza și inițializarea acesteia utilizând constructorul clasei care așteaptă argumentele tuplului sau o altă variabilă de același tip conform secvenței de cod (Exemplul 1.1).

Pentru cazul în care avem deja construită o astfel de variabilă și dorim să îi modificăm valoarea trebuie utilizat operatorul de atribuire, care primește ca al doilea operand un tuplu

existent. Alternativ se poate crea o variabilă temporară care să fie copiată. Pentru a face acest lucru se poate utiliza oricare dintre următoarele abordări:

- declararea explicită unei variabile de acest tip;
- folosirea funcției `std::make_tuple(x, y, z,...)` care creează un obiect de tip `tuple<T1, T2, T3, ...>` pe baza a tuturor valorilor x de tipul $T1$, y de tipul $T2$, z de tipul $T3$, ...;
- folosirea unei liste de inițializare care conține valorile elementelor ce urmează să fie copiate. Această listă este reprezentă cu ajutorul acoladelor.

Exemplu 1.1: Declararea variabilelor de tip tuple

```
#include<iostream>
#include<tuple>

int main()
{
   // declararea obiectului pers1 de tipul tuple,
   // care retine un sir de caractere si doua
   // numere intregi: nume, id si varsta
   std::tuple<std::string, int, int> pers1;

   // declararea variabilei pers2 si initializarea acesteia
    std::tuple<std::string,int,int>pers2("Ion", 12, 23);

   //copierea variabilei pers1 in variabila pers3
   std::tuple<std::string, int, int> pers3(pers1);

   //initializarea variabilei pers1
   pers1 = std::make_tuple("Ana", 13, 20);

    //utilizarea listei de initializare
   std::tuple<float, int, int> val = {4.3, 14, 19};

   return 0;
}
```

Modificarea valorilor păstrate într-un tuplu se poate realiza și cu ajutorul funcției $swap$ care interschimbă valorile a două variabile de același tip.

Accesul la elementele tuplului se poate realiza în diferite moduri, în funcție de context, și anume prin:

- **Funcția** `std::get<element>(tuple)` care returnează o referință către variabile de ordin *element* din tuple. Începând cu C++14, există și varianta funcției *get*, în care *element* este tipul câmpului către care se dorește referința, dar care poate fi folosită doar dacă tuplul dispune de un singur câmp de tipul indicat prin *element*.
- **Funcția** `std::tie(x, y, z,...)` care utilizată împreună cu operatorul = atribuie variabilelor $x, y, z, ...$ valorile reținute într-un tuplu. Este de menționat faptul că aceste variabile trebuie declarate anterior apelului de funcție.
- **Conceptul de** *structured binding declaration* (începând cu C++17) care poate fi privit ca o generalizare a funcției *tie*, acesta cuprinzând procesele de declarare și inițializare a unor variabile ce pot reține valorile sau referința elementelor din tuplu.

Accesul la membrii unui tuplu - Exemplu

```
#include<iostream>
#include<tuple>
#include<string>

int main()
{
    int id, varsta;
    std::string nume;

    std::cout << "Nume=";
    std::getline(std::cin, nume);
    std::cout << "Id="; std::cin >> id;
    std::cout << "Varsta="; std::cin >> varsta;
    std::tuple<std::string, int, int> pers1 = { nume, id, varsta };

    //accesarea campurilor variabilei cu get<indice>
    std::cout << "Nume:" << std::get<0>(pers1) << "\n";
    std::cout << "Id:" << std::get<1>(pers1) << "\n";
    std::cout << "Varsta:" << std::get<2>(pers1) << "\n";

    //copierea valorilor din tuple
    int idCopie, varstaCopie;
    std::string numeCopie;
    std::tie(numeCopie, idCopie, varstaCopie) = pers1;

    //structured binding declaration
    //declararea a 3 varibile ce retin valorile din pers1
    auto [a1, b1, c1] = pers1;
    std::cout << a1 << ", " << b1 << ", "<<c1 << "\n";
```

```
    //copierea referintei valorilor din tuple
    auto& [a2, b2, c2] = pers1;
    std::cout << a2 << ", " << b2 << ", " << c2 << "\n";
    return 0;
}
```

Pair

```
template <class T1, class T2> struct pair;
```

Această clasă *template* definește un container care grupează două valori, care pot fi de tipuri diferite, într-o singură variabilă. Clasa este un caz particular de tuplu și dispune de doi membri publici: *first* și *second*, care permit accesul la cele două valori, precum și de funcții de manipulare similare celor pentru tipului de date *tuple*.

Declararea variabilelor de tip pair - Exemplu

```
#include<iostream>

int main()
{
   //declararea obiectului pers1 de tipul pair,
   //care retine un sir de caractere si un numar intreg
   std::pair<std::string, int> pers1;

   //declararea obiectului pers2 si
   //initializarea acestuia cu valorile "Ion" si 124
   std::pair<std::string, int> pers2("Ion", 124);

   //copierea variabilei pers1 in variabila pers3
   std::pair<std::string, int> pers3(pers1);

   //initializarea cu lista de initializare
   std::pair<std::string, int> pers4 = {"Ana", 110};
   return 0;
}
```

Accesul la membrii variabilelor de tip pair- Exemplu

```
#include<iostream>
#include<string>

int main()
{
    std::pair<std::string, int> pers1;
```

```
    std::pair<std::string, int> pers2("Ana", 225);
    int id;
    std::string nume;

    std::cout << "Nume=";
    std::getline(std::cin, nume);
    std::cout << "Id="; std::cin >> id;

    //crearea unei perechi utilizand valorile citite
    pers1 = std::make_pair(nume, id);

    //alternativa a functiei make_pair,
    //utilizand lista de initializare
    pers1 = { nume, id };
    std::cout << "Prima persoana" << "\n";

    //accesarea campurilor variabilei
    std::cout << "Nume: " << pers1.first << "\n";
    std::cout << "id:" << pers1.second << "\n\n";

    //accesarea campurilor cu get
    std::cout << "A doua persoana" << "\n";
    std::cout << "Nume:" << std::get<0>(pers2) << "\n";
    std::cout << "Id:" << std::get<1>(pers2) << "\n";

    return 0;
}
```

1.2 Containere secvențiale

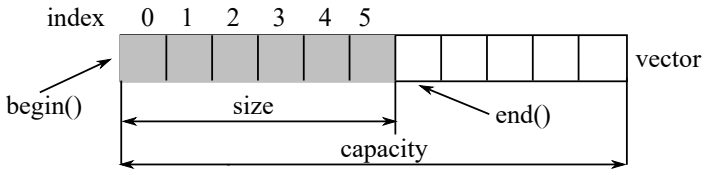
Vector

```
template<class T, class Alloc = allocator<T>>class vector;
```

Clasa `vector` definește un container care stochează o secvență de elemente de tip *T*, tipul trebuie stabilit atunci când se declară un astfel de container. Elementele păstrate în vector se stochează în zone de memorie succesive, astfel încât accesul se poate realiza prin poziție.

Un container de tip vector are dimensiune variabilă și utilizează intern alocare dinamică de memorie. Atunci când se adaugă un element nou la vector, dacă memoria alocată nu este suficientă, se realocă o zonă de memorie nouă, se copiază elementele din zona veche în zona

nouă și se dealocă zona veche de memorie. Aceste acțiuni se realizează automat și sunt gestionate de către *alocator*.



Deoarece operațiile de realocare și copiere sunt costisitoare din punct de vedere al timpului de execuție, acest lucru nu se realizează la fiecare nouă adăugare a unui element, ci alocatorul alocă de obicei ceva mai multă memorie decât este necesară pe moment. Numărul de poziții alocate la fiecare adăugare depinde de obicei de numărul de adăugări efectuate până la momentul respectiv și depinde de implementările din diferitele biblioteci.

Astfel numărul de elemente stocate efectiv în vector poate fi diferit de numărul de poziții alocate, obiectul dispunând de două funcții membre:

- `size()` - returnează numărul de elemente stocate în vector
- `capacity()` - returnează numărul de poziții alocate pentru vector

Alocarea memoriei necesare pentru stocarea elementelor din vector se poate realiza prin apelul consecutiv al funcției care adaugă un element nou după ultimul element din container sau prin utilizarea funcției `reserve(nr:int)` care alocă nr zone de memorie de tipul elementelor din vector.

Capacitatea versus dimensiunea unui vector - Exemplu

```
#include<vector>
int main()
{
    std::vector<int> vector = { 1 };
    vector.reserve(10);
    std::cout << "Capacitatea vectorului:"
        << vector.capacity() << "\n"; // 10 pozitii
    std::cout << "Lungimea vectorului:"
        << vector.size() << "\n";  // lungime 1

    if (vector.empty())
        std::cout << " vectorul este vid\n";
```

```
    else
        std::cout << " vectorul nu este vid\n";
    vector.clear();
    return 0;
}
// capacitatea vectorului este 10, iar lungimea 0
```

Un vectorul care nu conține niciun element se numește *vector vid*, iar această proprietate este verificată prin apelul funcției `empty()`, care întoarce un răspuns boolean.

Dintr-un vector care conține mai multe elemente putem obține un vector vid utilizând funcția `clear()`, care șterge toate elementele din vector, fără a elibera zonele de memorie ocupate de acestea.

Clasa `vector` dispune de o serie de constructori, care permit declararea și inițializarea obiectelor de tip vector.

Exemple de declarare și inițializare a unui vector

```
//declararea unui vector vid de nr intregi
std::vector<int>first;

//vector ce contine elementul 4 de 100 de ori
std::vector<int>second(4, 100);

//vector ce contine elementele {1, 2, 3}
std::vector<int>third = {1, 2, 3};

//vector ce contine elementele lui second
std::vector<int>fourth(second.begin(),second.end());

//vector copie a lui third
std::vector<int>fifth(third);
```

Observație: În exemplul de mai sus, vectorul *fourth* se obține prin ***iterare*** prin vectorul *second*, cu ajutorul iteratorilor `begin()` și `end()`. La finalul secțiuni este prezentată pe scurt noțiunea de iterator împreună cu un exemplu pentru vector.

Adăugarea unui element la sfârșitul unui vector se realizează cu ajutorul funcției `push_back`. Adăugarea unui element cu `push_back` determină creșterea dimensiunii vectorului cu o unitate. Dacă prin aceasta se depășește memoria alocată, adică capacitatea vectorului, atunci se produce realocare de memorie. Argumentul funcției trebuie să fie un obiect de tipul celor stocate în vector.

Ștergerea ultimului element al unui vector se realizează cu ajutorul funcției `pop_back`. Această funcție este apelată fără parametri și determină modificarea lungimii vectorului, dar nu și a capacității acestuia.

```
std::vector<int> vector = {1, 2, 3};
// stergerea elementului de pe ultima pozitie
vector.pop_back();

// inserarea valorii 4 la finalul sirului de valori
vector.push_back(4);
// vectorul contine valorile: {1, 2, 4}
```

Accesul elementelor dintr-un container de tip vector este gestionat cu ajutorul operatorului [], precum și cu ajutorul unor funcții membre:

- **Operatorul []** - returnează valoarea elementului de pe o poziție dată ca parametru. În momentul utilizării acestui operator este necesar să ne asigurăm de faptul că în vector există poziția solicitată prin parametru, deoarece operatorul nu verifică acest aspect, fapt ce cauzează un comportament nedefinit.
- `Funcția at()` - returnează valoarea elementului de pe o poziție dată ca parametru. Această funcție verifică și semnalează printr-o excepție de tipul *out_of_range* dacă poziția se află în afara intervalului de poziții valide.
- `Funcția front()` - returnează referința elementului de pe prima poziție.
- `Funcția back()` - returnează referința elementului de pe ultima poziție

Exemplu:

```
#include<iostream>
#include <vector>
int main()
{
   std::vector<int> vect = {1, 2, 3, 4, 5};

   std::cout << "Primul element: " << vect.front() << "\n";
   std::cout << "Ultimul element: " << vect.back() << "\n";
   std::cout << "Al doilea element: " << vect[1] << "\n";
   std::cout << "Al patrulea element: " << vect.at(3) << "\n";

   return 0;
}
```

Parcurgerea elementelor dintr-un vector se realizează cu ajutorul instrucțiunii repetitive *for* într-unul din următoarele moduri:

- **metoda clasică** - prin intermediul for-ului, variabila *poz* reține la fiecare iterație poziția curentă din vector, iar elementele containerului pot fi accesate cu ajutorul *operatorului [ ]* sau al funcției *at()*.

```
for (int poz = 0; poz < vect.size(); poz++)
    std::cout << vect[poz] << " ";
```

- **range base for** - variabila *element* va reține, pe rând, toate elementele vectorului, acestă metodă nepermițând o parcurgere parțială a containerului. Din punct de vedere sintactic for-ul este format din: tipul elementelor din vector, urmat de numele unei variabile ce va reține pe rând fiecare element din container, apoi operatorul ":" și numele vectorului (containerului) parcurs.

```
for(int element : vect)
    std::cout << element << " ";
```

- **utilizând iteratori** - variabila *it* reține la fiecare iterație un iterator (care va funcționa ca un pointer) către elementul curent din vector. Din punct de vedere sintactic observăm o manieră asemănătoare cu cea a *for-ului clasic*.

```
std::vector<int>::iterator it;
for(it = vect.begin(); it<vect.end(); it++)
    std::cout << *it << " ";
```

Array

```
template < class T, size_t N > class array;
```

Clasa `array` definește un container șablon care stochează o secvență de elemente de tip T, asemenea unui `vector`, principala diferența a celor două containere fiind variabilitatea dimensiunii. Array-ul este un container caracterizat prin dimensiunea fixă de N poziții. Tipul T al elementelor din array, precum și capacitatea acestuia N, trebuie stabilite atunci când se declară un astfel de container.

Pentru manipularea și parcurgerea elementelor dintr-un array se pot utiliza funcții prezentate în cadrul secțiunii *vector*, cu următoarele observații:

- lungimea array-ului este egală cu capacitatea acestuia, iar această valoare este determinată folosind funcția `size();`
- alocarea de memorie se realizează în momentul declarării variabilei, motiv pentru care nu există funcția `reserve();`
- nu putem apela funcții de inserare sau de ștergere a unui element în/din container, dimensiunea array-ului fiind fixă.

Exemplu:

```
#include<iostream>
#include <array>

int main(){
    //declararea unui array de lungime 7,
    //ce contine valori default de tip float
    std::array<float, 7> first;

    //declararea unui array de lungime 10,
    //ce contine valorile intervalului [1,6]
    //pe primele 6 pozitii
    std::array<int, 10> second = {1, 2, 3, 4, 5, 6};

    //declararea unui array copie a lui second
    std::array<int, 10> third(second);

    std::cout << "Lungimea celui de-al doilea array:"
        << second.size() << "\n"; // lungime 10

    if (second.empty())
        std::cout << " vectorul este vid \n";
    else
        std::cout << " vectorul nu este vid\n";

    for (int poz = 0; poz < second.size(); poz++)
        std::cout << second[poz] << " ";
    std::cout << "\n";
    //se va afisa: {1, 2, 3, 4, 5, 6, 0, 0, 0, 0}
    third.at(6) = 7;
    for (int element : third)
        std::cout << element << " ";
    std::cout << "\n";
    //se va afisa: {1, 2, 3, 4, 5, 6, 7, 0, 0, 0}
    return 0;
}
```

1.3 Iteratori

Iteratorii sunt obiecte care *pointează* spre un element aflat într-o colecție de elemente și care permit iterarea prin acea colecție utilizând operatori de incrementare ($++$) sau decrementare ($--$). Utilizarea iteratorilor contribuie la o eficientizare a timpului de execuție, întrucât se lucrează cu pointeri către elementele din container și se evită copierea acestora.

Un alt avantaj oferit de către iteratori este acela că, utilizatorii nu sunt nevoiți să știe, cum trebuie iterat printr-un container, de exemplu listă înlănțuită sau arbore binar (`set`, `map`), acest lucru fiind gestionat de către iterator. O parcurgere, din punct de vedere al codului scris, este absolut similară pentru oricare tip de container.

Containerele din STL conțin suport ce permite parcurgerea acestora utilizând iteratori cu ajutorul funcțiilor `begin()` și `end()`, care returnează un pointer către primul, respectiv către prima poziție de după ultimul element al colecției.

Exemplu:

```
#include<iostream>
#include <vector>

int main()
{
   std::vector<int> vector = {1, 2, 3, 4, 5, 6};

   std::vector<int>::iterator it;
   for (it = vector.begin();it!=vector.end();it++)
      std::cout << *it << " ";

   return 0;
}
```

Observație! Deoarece iteratorul reține un pointer către un element din container este necesar să utilizăm **operatorul *** pentru a obține valoarea elementului.

Pentru a accesa elementul de pe o poziție specifică din cadrul containerului se pot utiliza funcțiile `std::next()` și `std::prev()` sau operatorii de adunare și scădere.

Exemplu:

```
#include<iostream>
#include <vector>

int main(){
   std::vector<int> vector = {1, 2, 3, 4, 5, 6};
```

```
    // iterator catre primul element din vector
    std::vector<int>::iterator begin_it=vector.begin();

    //iterator catre urmatoarea pozitie de dupa
    //ultimul element al vectorului
    std::vector<int>::iterator end_it = vector.end();

    std::cout << *begin_it << " ";//se va afisa val.1
    //std::cout << *end_it << " ";//eroare

    auto next_it = std::next(begin_it, 2);
    std::cout << *next_it << " ";//se va afisa val.3

    //alternativa a functiei std::next()
    std::cout << *(begin_it + 2) << " ";

    auto prev_it = std::prev(end_it, 1);
    std::cout << *prev_it << " ";//se va afisa val.6

    //alternativa a functiei std::prev()
    std::cout << *(end_it - 1) << " ";
    return 0;
}
```

Capitolul 2

Noțiuni de complexitate

2.1 Aspecte teoretice

În dezvoltarea software, un algoritm poate fi implementat în multiple moduri, iar alegerea variantei corecte este esențială pentru performanța aplicației. O metodă intuitivă, dar profund eronată, de a compara doi algoritmi este simpla cronometrare a timpului de execuție. Această abordare este nesigură deoarece rezultatul depinde masiv de factori externi precum viteza procesoarelor, limbajul de programare ales, compilatorul folosit sau chiar și de alte procese rulează pe sistem în acel moment. Cu alte cuvinte, un algoritm care rulează rapid astăzi pe un laptop puternic s-ar putea să se comporte dezastruos pe un server vechi sau, mai important, pe un set de date de zece ori mai mare.

Analiza complexității computationale oferă un răspuns științific la această problemă, înlocuind măsurătorile empirice cu un model teoretic riguros. Ea ne permite să studiem eficiența intrinsecă a unui algoritm abstractizându-l de constrângerile hardware-ului. Principiul de bază este să identificăm operația dominantă (de exemplu, o comparație sau o atribuire) și să determinăm cum crește numărul de execuții ale acesteia odată cu creșterea volumului datelor de intrare, notat convențional cu n. Scopul final este de a anticipa cum se va comporta algoritmul pe seturi de date foarte mari (un scenariu numit *comportament asimptotic*) și de a oferi un cadru obiectiv pentru compararea soluțiilor alternative la aceeași problemă[7].

Astfel, în loc să întrebăm "Cât durează să ruleze?", vom întreba "Cât de mult crește timpul necesar atunci când datele mele devin din ce în ce mai mari?". Această schimbare de perspectivă este fundamentală pentru proiectarea sistemelor scalabile și a aplicațiilor eficiente. Acest capitol introduce instrumentele necesare pentru a răspunde la această întrebare, punând bazele pentru luarea unor decizii informate în materie de performanță algoritmică.

2.1.1 Analiza complexității. Notații asimptotice

Evaluarea performanței unui algoritm prin simpla cronometrare a timpului său de execuție este o metodă intuitivă, dar inerent defectuoasă. Rezultatele obținute prin această abordare sunt extrem de volatile și direct dependente de o multitudine de factori contingenți. În primul rând, ele variază odată cu puterea de calcul a hardware-ului, astfel încât un cod care rulează acceptabil pe un calculator modern ar putea deveni inutilizabil pe o platformă mai veche. În al doilea rând, implementări în limbaje de programare diferite—unul compilat și altul interpretat—vor produce timpi de execuție drastic diferiți pentru același algoritm, făcând orice comparație lipsită de sens. Mai mult, natura și volumul datelor de intrare influențează decisiv măsurătoarea; un algoritm optimizat pentru un anumit tip de structură de date se poate descurca mediocru pe altul, iar teste pe seturi mici de date pot ascunde complet ineficiența care apare la scară mare.

Pentru a depăși aceste limitări, este necesară adoptarea unui model teoretic care să evalueze algoritmul însuși, independent de mediul specific de execuție. Acest model se concentrează pe identificarea și numărarea operațiilor fundamentale, precum comparații, atribuiri sau accesări în memorie, pe care algoritmul le efectuează. Scopul este de a exprima cum crește numărul acestor operații pe măsură ce dimensiunea problemei, notată cu n, crește spre infinit. Această analiză asimptotică ne oferă două capacități cruciale: **predicția** și **comparația**. Prin predicție, putem extrapola cu încredere comportamentul unui algoritm pe seturi de date viitoare, mult mai voluminoase, pe baza unor calcule matematice. Prin comparație, putem contrasta doi algoritmi care adresează aceeași problemă și îi putem ierarhiza în mod obiectiv după eficiența lor intrinsecă, alegându-l pe cel a cărui cerință de resurse computationale crește mai lent.

Notația asimptotică Big O

Analiza eficienței algoritmilor necesită un cadru formal care să abordeze problema esențială a **scalabilității**: cum crește consumul de resurse computationale (timp și memorie) pe măsură ce volumul datelor de intrare, notat convențional prin n, crește la infinit. Notația asimptotică, și în special **notația Big O**, oferă acest cadru, permițând exprimarea unei **margini superioare** a ratei de creștere a unei funcții, abstractizând detaliile hardware-ului și ale implementării specifice.

Formal, spunem că o funcție $T(n)$ aparține clasei $O(g(n))$ dacă și numai dacă există două constante pozitive c și n_0 astfel încât:

$$0 \leq T(n) \leq c * g(n) \text{ pentru orice } n \geq n_0$$

Această inegalitate captează ideea că, pentru valori suficient de mari ale lui n, funcția $g(n)$ **domină** sau **limitează superior** pe $T(n)$ într-un mod proporțional, înmulțită doar cu o constantă finită c.

Interpretarea intuitivă este că Big O descrie **comportamentul în limita** sau **comportamentul asimptotic**.

Termenii de ordin inferior și factorii constanți din $T(n)$ devin neglijabili în comparație cu termenul dominant atunci când n este foarte mare. De exemplu, considerați funcția $T(n) = 5n^3 + 2n^2 + 100n + 500$. Pentru valori mici ale lui n, termenii de ordin inferior ($2n^2$, $100n$, 500) au un impact semnificativ. Cu toate acestea, pe măsură ce n crește, termenul $5n^3$ devine decisiv. Putem demonstra că $T(n) \in O(n^3)$ alegând, de exemplu, $c = 6$ și $n_0 = 10$:

$$5n^3 + 2n^2 + 100n + 500 \leq 5n^3 + 2n^3 + 100n^3 + 500n^3 = 607n^3 \text{ pentru } n \geq 1$$

Dar o alegere mai strânsă este posibilă. Pentru $n \geq 10$, avem $n^3 \geq n^2$, $n^3 \geq n$, și $n^3 \geq 1$ avem:

$$5n^3 + 2n^2 + 100n + 500 \leq 5n^3 + 2n^3 + 10n^3 + 50n^3 = 67n^3$$

Astfel, pentru $n_0 = 10$ și $c = 67$, inegalitatea $T(n) \leq 67 \cdot n^3$ este satisfăcută, ceea ce confirmă faptul că $T(n)$ crește cel la fel de repede ca n^3, abstractizând constantele.

Analiza cazului nefavorabil (Worst-Case Analysis)

În analiza algoritmilor, este deosebit de important să se considere **cazul nefavorabil** (worst-case scenario). Acesta reprezintă configurația datelor de intrare pentru care algoritmul necesită cel mai mare timp de execuție sau cea mai mare cantitate de memorie. Big O, prin definiția sa ca margine superioară, este ideal pentru a caracteriza acest caz, oferind o garanție că performanța algoritmului nu va fi mai proastă decât cea descrisă de $O(g(n))$ pentru orice input posibil de dimensiune n.

Considerăm algoritmul simplu de căutare liniară într-un array nesortat.

Algorithm 1: Functie cauta_secvential(arr, cheie)

```
begin
    for i ← 0, lungime(arr) − 1 do
        if arr[i] == cheie then
            return i // Element gasit
        end
    end
    return -1 // Elementul nu a fost gasit
end
```

Analiză:

1. **Operația dominantă**: Comparația `arr[i] == cheie`. Această operație este executată în fiecare iterație a buclei `for`. Numărul de execuții ale acesteia determină complexitatea de timp.

2. **Cazul favorabil**: Elementul căutat se află pe prima poziție în array. Se face o singură comparație. $T(n) = 1 \in O(1)$.

3. **Cazul mediu**: Elementul este prezent undeva în array, în medie, la jumătate. Numărul mediu de comparații este $\frac{n}{2}$. Deoarece constanta $1/2$ este ignorată, $T(n) \in O(n)$.

4. **Cazul nefavorabil**: Elementul nu se află în array sau se află pe ultima poziție. În acest scenariu, algoritmul trebuie să parcurgă toate cele n elemente și să efectueze n comparații eșuate (sau $n - 1$ eșuate și una reușită). Astfel, $T(n) = n$.

Aplicând definiția Big O pentru cazul nefavorabil ($T(n) = n$), căutăm o funcție $g(n)$ și constante c, n_0 astfel încât $n \leq c \cdot g(n)$ pentru $n \geq n_0$. Alegând $g(n) = n$, $c = 1$, și $n_0 = 1$, inegalitatea $n \leq 1 \cdot n$ este evident adevărată pentru toate $n \geq 1$. Prin urmare, complexitatea de timp în cazul nefavorabil pentru căutarea liniară/secvențială este $O(n)$. Aceasta înseamnă că, în cel mai rău caz posibil, timpul de execuție al algoritmului va crește **liniar** odată cu dimensiunea array-ului. Aceasta este o garanție puternică: indiferent de input, algoritmul nu va face niciodată mai mult de ordinul n operații.

Analiza devine mai interesantă pentru algoritmi cu instrucțiuni repetitive imbricate:

Algorithm 2: Functie proceseaza_matrice(n)

```
begin
    //Bucla 1
    for i ← 0, n − 1 do
        //Bucla 2
        for j ← 0, n − 1 do
            facem_ceva(i, j) //Operație O(1)
        end
    end
end
```

Analiză:

1. **Operația dominantă**: Apelul pentru `facem_ceva(i, j)`, care presupunem că are timp constant $O(1)$. Această operație este executată în interiorul celei mai adânci bucle imbricate.

2. **Numărul de execuții**: Bucla interioară (j) se execută de n ori pentru fiecare iterație a buclei exterioare (i). Bucla exterioară are n iterații.

3. **Total operații**: Numărul total de execuții ale operației dominante este $n \times n = n^2$.

4. **Complexitate**: $T(n) = n^2 \cdot k$, unde k este timpul constant al operației. Conform definiției, $n^2 \cdot k \leq c \cdot n^2$ pentru $n \geq 0$, alegând $c = k$. Prin urmare, complexitatea de timp este $O(n^2)$.

Acest exemplu demonstrează principiul fundamental conform căruia buclele de iterație imbricate, unde fiecare buclă depinde liniar de dimensiunea intrării n, conduc la o creștere polinomială a complexității temporale. Extinzând acest raționament, dacă ar fi fost prezentă o a treia buclă imbricată, similară celorlalte două și dependentă tot de n, numărul total de operații ar fi escaladat la $n \times n \times n = n^3$. În consecință, complexitatea algoritmului ar fi fost cubică, adică $O(n^3)$. Acest principiu de multiplicare a factorilor n se aplică pentru orice număr de bucle imbricate, fiecare contribuind cu un factor de n la gradul polinomului de complexitate.

Este foarte important de înțeles că $T(n) = O(g(n))$ este o afirmație despre forma funcției, nu despre viteza absolută a algoritmului. Un algoritm $O(n)$ poate fi, pentru anumite valori ale lui n, mai lent decât unul $O(n^2)$ datorită constantelor mari ascunse (de exemplu, $T_1(n) = 1000n$ vs. $T_2(n) = n^2$). Cu toate acestea, definiția ne asigură că există un punct n_0 după care algoritmul cu complexitatea asimptotic mai mică ($O(n)$) va fi întotdeauna mai rapid, deoarece n^2 va depăși în cele din urmă $1000n$. Notația Big O ne direcționează astfel către alegerea algoritmilor cu cea mai bună scalabilitate pe termen lung.

Notația asimptotică Big Omega (Ω)

În timp ce notația Big O (O) definește o **limită superioară asimptotică** pentru rata de creștere a unei funcții (de exemplu, timpul de execuție sau spațiul de memorie al unui algoritm), descriind complexitatea în cel mai defavorabil caz și indicând "cât de ineficient" poate deveni algoritmul, notația Big Omega (Ω) descrie **cel mai bun caz posibil** din punct de vedere al performanței, oferind o garanție a performanței minime. Cu alte cuvinte, $\Omega(g(n))$ exprimă faptul că, indiferent de input, algoritmul va avea nevoie de cel puțin o cantitate de resurse proporțională cu $g(n)$ pentru valori suficient de mari ale lui n.

Formal, spunem că o funcție $T(n)$ (timpul de execuție) aparține clasei $\Omega(g(n))$ dacă și numai dacă există constante pozitive c și n_0 astfel încât:

$$0 \leq c * g(n) \leq T(n) \text{ pentru oricare } n \geq n_0$$

Această inegalitate spune că, dincolo de un anumit punct n_0, funcția $T(n)$ este **mărginită inferior** de $g(n)$ înmulțită cu o constantă c. Ea nu oferă o estimare exactă, ci doar o limită inferioară asupra ratei de creștere.

Revenim la algoritmul de căutare liniară într-un array nesortat de mai sus pentru a ilustra diferența față de Big O.

Analiză:

1. **Cazul favorabil**: Elementul căutat se află pe prima poziție. Algoritmul efectuează o singură comparație. $T(n) = 1$.

2. **Definirea limitei inferioare**: Pentru a găsi o margine inferioară, trebuie să găsim o funcție $g(n)$ astfel încât, chiar și în cel mai bun caz, $T(n)$ să fie cel puțin proporțional cu $g(n)$.

3. **Alegerea lui** $g(n)$: Este tentant să spunem că, deoarece în cel mai bun caz facem o singură operație, complexitatea este $\Omega(1)$. Acest lucru este corect. Putem alege $g(n) = 1$, $c = 1$ și $n_0 = 1$. Inegalitatea $1 \cdot 1 \leq T(n)$ este adevărată pentru orice $n \geq 1$, deoarece algoritmul va face întotdeauna cel puțin o comparație (chiar dacă găsește imediat elementul).

Prin urmare, **complexitatea în cel mai bun caz pentru căutarea liniară este** $\Omega(1)$. Aceasta este o afirmație slabă, dar corectă; ea ne spune că nu putem face mai bine de *constant time* pentru acest algoritm în cel mai fericit scenariu.

Analiza devine mai interesantă și mai utilă atunci când căutăm o limită inferioară pentru o întreagă clasă de probleme, nu doar pentru un algoritm specific.

De exemplu, să considerăm problema sortării unui array folosind numai comparații între elemente. Orice algoritm de sortare prin comparație (e.g., Bubble Sort, Merge Sort, Quick Sort) trebuie să rezolve o problemă fundamentală: să găsească ordinea corectă a elementelor prin compararea lor.

Analiză:

1. **Modelul de calcul**: Considerăm că singura operație permisă care oferă informații despre ordinea elementelor este compararea a două elemente (e.g., $a \leq b$).

2. **Numărul de permutări**: Există $n!$ (n factorial) permutări posibile ale unui array de n elemente distincte. Algoritmul trebuie să identifice permutarea corectă (cea sortată) printre toate aceste posibilități.

3. **Modelul arborelui de decizie:** Execuția oricărui algoritm de sortare bazat pe comparații poate fi modelată ca un arbore de decizie binar.

   - Fiecare nod intern al arborelui reprezintă o comparație între două elemente (de exemplu, $A[i]$ vs $A[j]$).

- Fiecare ramură (stânga/dreapta) corespunde rezultatului acelei comparații ($A[i] < A[j]$ sau $A[i] > A[j]$).
- Fiecare frunză a arborelui reprezintă o permutare unică a elementelor de intrare, corespunzătoare unei ordonări complete posibile. Deoarece există n! permutări distincte pentru n elemente, arborele trebuie să aibă cel puțin $n!$ frunze.

4. **Înălțimea minimă a arborelui**: Un arbore binar cu L frunze are o înălțime de cel puțin $\lceil \log_2 L \rceil$. Prin urmare, înălțimea arborelui de decizie (care reprezintă numărul de comparații în cel mai nefavorabil caz) este cel puțin $\lceil \log_2(n!) \rceil$.

5. **Aproximarea lui** $\log(n!)$: Folosind aproximarea lui Stirling ($n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$), putem demonstra că $\log_2(n!) \in \Theta(n \log_2 n)$. Mai precis, $\log_2(n!) \geq c \cdot n \log_2 n$ pentru o constantă $c > 0$ și n suficient de mare.

Această analiză demonstrează un rezultat fundamental: **orice algoritm de sortare prin comparație are o complexitate de timp în cel mai nefavorabil caz de** $\Omega(n \log n)$.

Această afirmație $\Omega(n \log n)$ este extrem de valoroasă deoarece:

- **Stabilește o limită fundamentală**: Ea ne spune că niciun algoritm de sortare prin comparație nu poate fi mai bun decât $O(n \log n)$ în cazul nefavorabil. Dacă am crea unul care pretinde că are complexitatea $O(n)$, am ști imediat că este fals, deoarece ar încălca această limită inferioară.
- **Identifică algoritmii optimi**: Algoritmi precum Merge Sort și Heap Sort au complexitate $O(n \log n)$ în cazul nefavorabil. Aceasta înseamnă că ei ating această limită inferioară. Prin urmare, ei sunt optimali în cadrul clasei de algoritmi de sortare prin comparație; nu se poate face mai bine de atât.
- **Concentrează eforturile de cercetare**: Această limită inferioară ne indică faptul că, pentru a depăși performanța $O(n \log n)$, trebuie să părăsim modelul de sortare prin comparație și să explorăm algoritmi care exploatează proprietăți specifice ale datelor (e.g., Counting Sort, Radix Sort), care pot avea performanță $\Omega(n)$.

Notația asimptotică Big Theta (Θ)

Notația Big Theta (Θ) ocupă un loc special în analiza asimptotică, acela al **preciziei**. În timp ce Big O (O) oferă o margine superioară (o promisiune că nu va fi mai rău) și Big Omega (Ω) oferă o margine inferioară (o asigurare că nu va fi mai bine), Big Theta oferă o **margine strânsă**. Aceasta înseamnă că rata de creștere a funcției este **exact** determinată, abstractizând doar factorii constanți. Este cea mai informativă și de dorită notație atunci când este posibilă.

Formal, spunem că o funcție $T(n)$ aparține clasei $\Theta(g(n))$ dacă și numai dacă există constante pozitive c_1, c_2 și n_0 astfel încât:

$$0 \leq c_1 * g(n) \leq T(n) \leq c_2 * g(n) \text{ pentru oricare } n \geq n_0$$

Această definiție este echivalentă cu afirmația simultană:

$$T(n) \in O(g(n)) \text{ și } T(n) \in \Omega(g(n))$$

Cu alte cuvinte, $T(n)$ este *prinsă* sau *sandvișată* între doi multiplii ai lui $g(n)$ pentru toate valorile suficient de mari ale lui n. Funcția $g(n)$ este atât o margine superioară, cât și una inferioară pentru $T(n)$.

Considerăm un algoritm simplu pentru sumă primelor n numere naturale printr-o buclă:

Algorithm 3: Functie suma_componente(n)

```
begin
    suma ← 0
    for i ← 1, n do
        suma ← suma + i
    end
    return suma
end
```

Analiză:

Numărul de operații (atribuiri și adunări) este proporțional cu n. Avem $T(n) = a \cdot n + b$, unde a și b sunt constante. Putem găsi constantele c_1, c_2 și n_0 pentru $g(n) = n$:

$T(n) = an + b \leq (a + 1)n$ pentru $n \geq b$ (deoarece $b \leq n$).

$T(n) = an + b \geq an$ pentru toate n.

Alegând $c_1 = a$, $c_2 = a + 1$, și $n_0 = \max(1, b)$, inegalitatea $c_1 \cdot n \leq T(n) \leq c_2 \cdot n$ este satisfăcută. Prin urmare, $T(n) \in \Theta(n)$. Aceasta este o descriere precisă a comportamentului algoritmului.

QuickSort este exemplul clasic utilizat pentru a ilustra diferența dintre complexitatea în cazul mediu și cea în cazul nefavorabil, și pentru a arăta cum Big Theta se aplică în mod diferit în aceste scenarii. Ideea de bază a QuickSort este de a alege un element *pivot*, de a partiționa array-ul în elemente mai mici decât pivotul și elemente mai mari decât acesta, și de a apoi sorta recursiv cele două sub-array-uri.

Cazul nefavorabil pentru QuickSort survine atunci când alegerea pivotului este consistent proastă, împărțind array-ul în mod extrem de neechilibrat la fiecare pas. Acest lucru se întâmplă dacă pivotul este întotdeauna cel mai mic sau cel mai mare element.

Exemplu de input nefavorabil: Un șir deja sortat (crescător sau descrescător) dacă pivotul este ales mereu ca primul element.

Analiză: La fiecare apel recursiv, partiționarea produce un sub-array de dimensiune $n-1$ și unul de dimensiune 0. Adâncimea recursivă devine n, iar la fiecare nivel se parcurge un array de dimensiune aproximativ n. Timpul total este proporțional cu $n + (n-1) + (n-2) + ... + 1 = \frac{n(n+1)}{2}$.

Complexitate: $T_{worst}(n) \in \Theta(n^2)$. Aici, Big Theta este folosit corect pentru a descrie **exact** rata de creștere în acest scenariu pesimist.

Cazul mediu presupune că input-urile sunt randomizate sau că pivotul este ales la întâmplare. În medie, partiționarea va împărți array-ul în două părți de dimensiuni relativ echilibrate.

Analiză: Dacă presupunem că partiționarea împarte array-ul în două părți de dimensiuni aproximativ $n/2$, putem modela complexitatea prin ecuația de recurență: $T(n) = 2T(n/2) + \Theta(n)$.

Rezolvare: Conform teoremei Master, soluția acestei recurențe este $T(n) \in \Theta(n \log n)$.

Semnificație: Pentru majoritatea aranjamentelor de date (toate, exceptând cele rare, nefavorabile), QuickSort se comportă exact cu o complexitate $\Theta(n \log n)$.

Acest exemplu arată clar că **cazul mediu și cel nefavorabil pot avea complexități asimptotice radical diferite** ($\Theta(n \log n)$ vs. $\Theta(n^2)$). QuickSort este ales pentru aplicații practice tocmai datorită complexității sale medii excelente, $\Theta(n \log n)$, și faptului că constanta

ascunsă în notația O este foarte mică, făcându-l rapid în practică. Înțelegerea acestui decalaj este crucială:

Pentru proiectare: Ne avertizează să evităm mereu alegerea naivă a pivotului (e.g., primul element) și să folosim tehnici de randomizare sau alegerea medianei din trei pentru a **face cazul nefavorabil extrem de improbabil**.

Pentru analiză: Ne obligă să fim precauți în a nu generaliza. A spune simplu "QuickSort este $O(n^2)$" este corect dar inutil de pesimist. A spune "QuickSort este $\Omega(n \log n)$" este corect dar incomplet. A spune "QuickSort are complexitate medie $\Theta(n \log n)$ și complexitate worst-case $\Theta(n^2)$" oferă cea mai completă și utilă imagine.

Big Theta (Θ) este instrumentul prin care exprimăm **exactitatea** în analiza asimptotică. În timp ce O și Ω sunt utile pentru a oferi garanții și limite, Θ este regula de aur atunci când o putem aplica, deoarece caracterizează complet rata de creștere. Exemplul QuickSort subliniază faptul că un algoritm poate avea mai multe complexități Θ pentru diferite scenarii de input, iar discernământul între acestea este esențial pentru o înțelegere profundă și o alegere informată a algoritmilor.

2.1.2 Analiza complexității de timp

Analiza complexității timp reprezintă nucleul evaluării performanței algoritmilor. Scopul acesteia este de a determina cum crește timpul de execuție al unui algoritm odată cu creșterea dimensiunii datelor de intrare, notată cu n. Această analiză se bazează pe un proces sistematic de abstractizare, concentrându-se pe comportamentul asimptotic și ignorând detaliile care devin nesemnificative pentru valori mari ale lui n.

Analiza complexității timp urmează un proces în trei etape:

1. **Identificarea operației dominante**: Primul pas constă în identificarea operației fundamentale a algoritmului, aceea care se execută de cele mai multe ori și care are cel mai semnificativ impact asupra timpului total de execuție. Aceasta poate fi o comparație (în algoritmi de sortare/căutare), o atribuire, o operație aritmetică sau un acces la memorie. De exemplu, în cazul algoritmilor de sortare prin comparație, operația dominantă este *compararea a două elemente*.

2. **Exprimarea numărului de execuții în funcție de** n: În continuare, se exprimă numărul total de execuții ale operației dominante ca o funcție matematică $T(n)$ a dimensiunii intrării n. Aceasta implică analiza structurilor de control (bucle, recursivitate) și determinarea relațiilor de recurență. De exemplu, pentru o buclă `for` care parcurge un array de la 0 la $n-1$, numărul de execuții ale unui corp de buclă cu complexitate $O(1)$ este n, deci $T(n) = n$.

3. **Simplificarea expresiei asimptotice**: În final, se aplică notația asimptotică pentru a abstractiza comportamentul funcției $T(n)$. Se păstrează doar termenul dominant (termenul care crește cel mai rapid) și se ignoră atât constantele multiplicative, cât și toți termenii de ordin inferior. De exemplu, pentru $T(n) = 3n^2 + 5n + 100$, termenul dominant este $3n^2$. Se ignoră constanta multiplicativă 3 și termenii $5n$ și 100, rezultând o complexitate finală de $O(n^2)$.

2.1.3 Clasele de complexitate fundamentale

Clasele de complexitate oferă o modalitate de a categorisi algoritmii în funcție de rata de creștere a timpului lor de execuție. Prezentăm cele mai importante clase de la cea mai rapidă la cea mai lentă:

$O(1)$ **- Constantă.** Timpul de execuție este constant și nu depinde de dimensiunea datelor de intrare n. Acesta este cazul ideal. **Exemplu**: Accesarea unui element dintr-un array folosind un index, efectuarea unei operații aritmetice simple.

$O(\log n)$ **- Logaritmică.** Timpul de execuție crește logaritmic odată cu creșterea lui n. Aceasta indică o eficiență foarte mare, deoarece valoarea logaritmului crește extrem de încet. Baza logaritmului (2, 10, e) este ignorată în notația asimptotică, deoarece logaritmii cu baze diferite diferă doar printr-o constantă multiplicativă. **Exemplu**: *Căutarea binară* într-un array sortat. La fiecare pas, spațiul de căutare se înjumătățește.

$O(n)$ **- Liniară.** Timpul de execuție crește liniar și proporțional cu dimensiunea n a intrării. Este o complexitate eficientă pentru volume mari de date. **Exemplu**: *Parcurgerea unui array* pentru a găsi un element (căutare liniară), calcularea sumei elementelor unui array.

$O(n \log n)$ **- Linearitmică.** Timpul de execuție crește puțin mai rapid decât liniar. Această complexitate este asociată algoritmilor de sortare eficienți care împart problema în subprobleme, le rezolvă recursiv, și apoi combină soluțiile. **Exemplu**: *Merge Sort*, *Quick Sort* (în cazul mediu), *Heap Sort*.

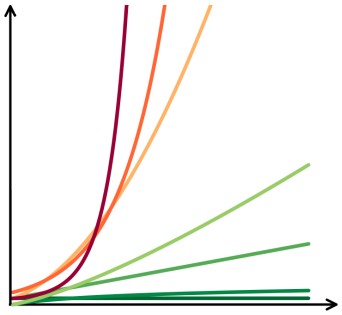
$O(n^2)$ **- Pătratică.** Timpul de execuție crește proporțional cu pătratul dimensiunii intrării. Această complexitate este caracteristică algoritmilor cu două bucle imbricate care procesează toate perechile de elemente. Devine prohibitivă pentru valori mari ale lui n. **Exemplu**: *Sortarea bulelor* (Bubble Sort), *Sortarea prin selecție* (Selection Sort), *Sortarea prin inserție* (Insertion Sort), parcurgerea unei matrice pătratice.

$O(2^n)$ **- Exponențială.** Timpul de execuție crește exponențial odată cu creșterea lui n. Algoritmii cu această complexitate devin rapid inutilizabili chiar și pentru valori mici ale lui n (de exemplu, $n > 40$). Aceștia apar adesea în rezolvarea naivă a problemelor prin forță brută, unde se testează toate submulțimile sau combinațiile posibile. **Exemplu**: Rezolvarea problemei rucsacului folosind forța brută, algoritmul pentru *Turnurile din Hanoi*, calcularea succesiunii Fibonacci naiv (fără memoizare).

$O(n!)$ **- Factorială.** Timpul de execuție crește factorial odată cu creșterea lui n. Aceasta este una dintre cele mai proaste complexități și este asociată cu algoritmi care generează toate permutările sau aranjamentele posibile ale unui set de elemente. Este practic inutilizabilă pentru $n > 12$. **Exemplu**: Generarea *tuturor permutărilor* unui șir de caractere, rezolvarea problemei comis-voiajorului prin căutare exhaustivă.

Funcțiile de complexitate descrise mai sus sunt reprezentate grafic în Figura 2.1.

Înțelegerea acestor clase de complexitate și a metodologiei de analiză este esențială pentru orice programator. Ea oferă un cadru obiectiv pentru a anticipa cum se va comporta un algoritm la scară mare, pentru a alege cel mai eficient algoritm dintre multiplele opțiuni și pentru a evita soluțiile care, deși funcționale pe seturi mici de date, devin practic inutilizabile în producție. Abordarea asimptotică asigură că deciziile luate sunt robuste și orientate către viitor, ținând cont de creșterea inevitabilă a volumului de date.



Timp constant	O(1)
Timp logaritmic	O(log n)
Timp liniar	O(n)
Timp quasiliniar/linearithmic	O(n log n)
Timp patrat	$O(n^2)$
Timp exponential	$O(c^n)$
Timp factorial	O(n!)

Figura 2.1: Funcțiile de complexitate comune în notația O

.

2.1.4 Analiza complexității de spațiu

În timp ce analiza complexității timp se concentrează pe eficiența computațională, analiza complexității de spațiu abordează eficiența în utilizarea memoriei. Ea cuantifică cantitatea totală de memorie de lucru (working memory) de care un algoritm are nevoie pentru a-și finaliza sarcina, **în plus față de spațiul ocupat de datele de intrare în sine**. Această analiză este la fel de crucială ca și cea a timpului, deoarece resursele de memorie sunt finite, iar alocarea excesivă poate duce la scurgerea de memorie (memory leaks) sau la erori de tip "out of memory" în aplicații.

Analiza complexității spațiu urmează un proces similar analizei complexității timp, dar se concentrează pe utilizarea memoriei:

1. **Identificarea surselor de memorie**: Se identifică toate sursele care consumă memorie suplimentară dincolo de storage-ul de bază pentru input. Acestea includ:
    - **Variabilele auxiliare**: Variabilele simple (de tip `int`, `float`, `pointer`) sau temporare folosite în algoritm.
    - **Structurile de date auxiliare**: Array-uri, liste, stive, cozi, hash-uri sau alte containere alocate dinamic în timpul execuției pentru a stoca rezultate intermediare.
    - **Spațiul de stivă (call stack)**: Memoria utilizată pentru a gestiona apelurile de funcții recursive. Fiecare apel adaugă un "stack frame" care conține variabilele locale, parametrii și adresa de returnare.

2. **Cuantificarea spațiului în funcție de** n: Se exprimă memoria totală utilizată de aceste surse ca o funcție $S(n)$ a dimensiunii intrării n. Se consideră de obicei numărul de *celule de memorie* sau numărul de elemente stocate.

3. **Aplicarea notației asimptotice**: Similar complexității timp, se aplică notația Big O pentru a abstractiza rata de creștere a memoriei utilizate. Se păstrează doar termenul dominant și se ignoră constantele.

Exemple practice

$O(1)$ **- Spațiu constant.** Algoritmul utilizează o cantitate fixă și constantă de memorie suplimentară, independent de dimensiunea datelor de intrare n. Acesta este cazul ideal pentru utilizarea memoriei. **Exemplu**: **Sortarea prin selecție** (Selection Sort). Algoritmul operează direct pe array-ul original (in-place), folosind doar un număr mic și fix de variabile pentru a ține minte indecși și valori temporare (de exemplu, o variabilă pentru indexul minimului curent și una pentru schimb temporar). Memoria suplimentară totală este, prin urmare, constantă.

$O(n)$ **- Spațiu liniar.** Algoritmul alocă o structură de date auxiliară a cărei dimensiune este direct proporțională cu dimensiunea intrării n. **Exemplu**: **Merge Sort**. Chiar dacă este un algoritm foarte eficient din punct de vedere al timpului ($O(n \log n)$), el necesită alocarea unui array auxiliar temporar pentru a interclasa sub-array-urile. Dimensiunea acestui array auxiliar este n, deci complexitatea spațială este $O(n)$. Este un exemplu clar al unui compromis (trade-off) între timp și memorie.

$O(n)$ **- Spațiu pentru stivă recursivă.** Algoritmii recursivi pot consuma spațiu liniar datorită stivei de apeluri, chiar dacă nu alocă explicit structuri de date mari. **Exemplu**: O funcție recursivă care calculează suma elementelor unui array. Dacă nu este optimizată (tail recursion), ea va face n apeluri recursive înainte de a începe să se întoarcă, ocupând $O(n)$ spațiu pe stivă.

$O(n^2)$ **- Spațiu pătratic.** Algoritmul creează o structură de date auxiliară bidimensională a cărei dimensiune depinde de n. **Exemplu**: Crearea unei matrice de adiacență pentru un graf cu n noduri. Matricea va avea $n \times n$ elemente, necesitând $O(n^2)$ spațiu.

Observații și compromisuri (Trade-offs)

- **Algoritmi in-place**: Un algoritm care operează **in-place** este unul care transformă datele de intrare fără a necesita o cantitate semnificativă de memorie auxiliară suplimentară, având de obicei complexitatea spațială $O(1)$. Sortările prin selecție și inserție sunt exemple de algoritmi in-place. Sortarea prin interschimbare (Bubble Sort) este, de asemenea, in-place.
- **Compromis timp-spațiu (Time-Space Trade-off)**: Este o situație comună în proiectarea algoritmilor. De multe ori, poți reduce timpul de execuție folosind mai multă memorie (de exemplu, utilizând tabele hash pentru a reduce timpul de căutare de la $O(n)$ la $O(1)$) sau poți reduce memoria folosită în schimbul unui timp de execuție mai mare. Alegerea depinde de constrângerile specifice aplicației.
- **Spațiul de intrare**: În mod convențional, spațiul ocupat de datele de intrare în sine **nu** este inclus în calculul complexității spațiale auxiliare. Totuși, uneori se face distincție între **spațiul total** (inclusiv input) și **spațiul auxiliar** (doar memoria suplimentară). Contextul definește care metrică este mai relevantă.

În concluzie, analiza complexității de spațiu completează imaginea eficienței unui algoritm. Un algoritm rapid poate fi inutilizabil dacă este prea lacom cu memoria, iar un algoritm economic cu memoria poate fi preferat în medii cu resurse limitate, chiar dacă este mai lent. O analiză cuprinzătoare ia în considerare ambele dimensiuni pentru a alege soluția optimă pentru o problemă dată.

2.1.5 Complexitatea relațiilor de recurență

Relațiile de recurență apar frecvent în analiza algoritmilor recursivi. Ele exprimă timpul de execuție $T(n)$ al unui algoritm pentru o problemă de dimensiune n în funcție de timpii pentru probleme mai mici. Aceste relații de recurență se rezolvă cu Teorema Master [3], pe care o enunțăm în continuare.

Teorema 2.1.1 (Teorema Master) *Fie $a \geq 1$ și $b > 1$ constante, fie $f(n)$ o funcție, iar $T(n)$ o funcție definită pe numerele întregi nenegative prin relația de recurență*

$$T(n) = a\,T(n/b) + f(n),$$

unde interpretăm n/b ca fiind fie $\lfloor n/b \rfloor$, fie $\lceil n/b \rceil$.

Atunci, $T(n)$ poate fi mărginită asimptotic după cum urmează:

(1) Dacă $f(n) = O\big(n^{\log_b a - \varepsilon}\big)$ pentru o constantă $\varepsilon > 0$, atunci

$$T(n) = \Theta\big(n^{\log_b a}\big)\,.$$

(2) Dacă $f(n) = \Theta\big(n^{\log_b a}\big)$, atunci

$$T(n) = \Theta\big(n^{\log_b a} \log n\big)\,.$$

(3) Dacă $f(n) = \Omega\big(n^{\log_b a + \varepsilon}\big)$ pentru o constantă $\varepsilon > 0$ și dacă f satisface condiția de regularitate

$$a\,f(n/b) \leq c\,f(n)$$

pentru o constantă $c < 1$ și pentru toate n suficient de mari, atunci

$$T(n) = \Theta(f(n)).$$

Exemple

1. **Sortare prin interclasare (MergeSort):** Se consideră că sortarea se face prin împărțirea recursivă a listei în două jumătăți și combinarea lor.

$$T(n) = 2T(n/2) + \Theta(n)$$

Aici $a = 2$, $b = 2$, $f(n) = \Theta(n)$ și $\log_2 2 = 1$. Cazul 2 se aplică, deci:

$$T(n) = \Theta(n \log n)$$

2. **Exponențiere rapidă:** Se consideră că exponențierea se realizează prin împărțirea exponentului la jumătate la fiecare pas.

$$T(n) = T(n/2) + \Theta(1)$$

Aici $a = 1$, $b = 2$, $f(n) = \Theta(1)$ și $\log_2 1 = 0$. Cazul 2 se aplică cu $k = 0$, deci:

$$T(n) = \Theta(\log n)$$

3. **Algoritm de împărțire în trei:** Se consideră că problema se împarte recursiv în trei subprobleme egale, cu cost liniar pentru combinarea rezultatelor.

$$T(n) = 3T(n/3) + \Theta(n)$$

Aici $a = 3$, $b = 3$, $f(n) = \Theta(n)$ și $\log_3 3 = 1$. Cazul 2 se aplică, deci:

$$T(n) = \Theta(n \log n)$$

2.2 Probleme rezolvate

În secțiunea care urmează propunem o serie de probleme cu complexități.

1. Care dintre următoarele afirmații este adevărată pentru orice n suficient de mare?

   A) $n^3 + 5n^2 \log n \in O(n^3)$

   B) $3^{n+1} \in O(3^n)$

   C) $n! \in O(2^n)$

   D) $2n^2 + 3n \in \Theta(n^2)$

   Rezolvare:

   A) **Adevărat**. n^3 este termenul dominant. $5n^2 \log n$ crește mai încet decât n^3 pentru n mare, deci există constante c și n_0 astfel încât $n^3 + 5n^2 \log n \leq c \cdot n^3$ pentru $n \geq n_0$.

B) **Adevărat**. $3^{n+1} = 3 \cdot 3^n$. Alegând $c = 3$ și $n_0 = 1$, avem $3^{n+1} \leq 3 \cdot 3^n$ pentru toți $n \geq 1$.

C) **Fals**. Funcția $n!$ crește mai repede decât orice funcție exponențială c^n pentru orice constantă c. Prin urmare, $n!$ nu poate fi mărginit superior de 2^n pentru n suficient de mare.

D) **Adevărat**. $2n^2 + 3n \leq 3n^2$ pentru $n \geq 3$ (deoarece $3n \leq n^2$ pentru $n \geq 3$), deci $2n^2 + 3n \in O(n^2)$. De asemenea, $2n^2 + 3n \geq 2n^2$ pentru $n \geq 1$, deci $2n^2 + 3n \in \Omega(n^2)$. În concluzie, $2n^2 + 3n \in \Theta(n^2)$.

2. Aranjați următoarele funcții în ordinea crescătoare a ratei de creștere:

$$f_1(n) = 10n,\ f_2(n) = n \log n,\ f_3(n) = n^{1.5},\ f_4(n) = n^2,\ f_5(n) = 2^n$$

Rezolvare: Ordinea corectă este: $f_1(n)$, $f_2(n)$, $f_3(n)$, $f_4(n)$, $f_5(n)$.

- $10n \in O(n)$ (liniară)
- $n \log n \in O(n \log n)$ (linearitmică)
- $n^{1.5} \in O(n^{1.5})$ (sub-pătratică, dar mai rapidă decât liniară și linearitmică)
- $n^2 \in O(n^2)$ (pătratică)
- $2^n \in O(2^n)$ (exponențială)

Pentru n suficient de mare, avem $10n < n \log n < n^{1.5} < n^2 < 2^n$.

3. Analizați complexitatea de timp pentru următoarea secvență de cod în cazul cel mai defavorabil, cel mai favorabil și mediu.

```
int cautarePozitie(int[] vector, int cheie) {
    for (int i = 0; i < vector.length; i++) {
        if (vector[i] == cheie) {
            return i;
        }
    }
    return -1;
}
```

Rezolvare:

- **Operația dominantă**: Comparația `vector[i] == cheie`.

- **Cazul cel mai favorabil**: Elementul `cheie` se află pe prima poziție în vector (`index = 0`). Se face o singură comparație. $T(n) = 1 \in O(1)$.
- **Cazul cel mai defavorabil**: Elementul `cheie` **nu** se află în array sau se află pe ultima poziție. Se fac n comparații. $T(n) = n \in O(n)$.
- **Cazul mediu**: Presupunând că elementul este prezent în array și poziția sa este uniform distribuită, numărul mediu de comparații este $\frac{n}{2}$. Deoarece constanta $1/2$ este ignorată, $T(n) \in O(n)$.

Concluzie: Complexitatea este $O(1)$ (best-case) și $O(n)$ (worst-case și average-case).

4. Determinați complexitatea timp și spațiu pentru următoarea funcție recursivă care calculează suma elementelor unui array.

```
int suma(int[] vector, int dimensiune) {
    if (dimensiune <= 0) {
        return 0;
    }
    return (suma(vector, dimensiune-1) +
       vector[dimensiune-1]);
}
```

Rezolvare (Timp):

- **Modelare recurență**: Funcția se apelează recursiv de n ori, până când n devine 0. La fiecare apel, se efectuează un număr constant de operații (adunare, accesare element, operații de apel funcție).
- **Ecuația de recurență**: $T(n) = T(n-1) + O(1)$
- **Rezolvare recurență**: Expandând recurența: $T(n) = T(n-1)+c = T(n-2)+2c = \ldots = T(0) + n \cdot c$. Deoarece $T(0)$ este constant, $T(n) \in O(n)$.

Rezolvare (Spațiu):

- **Sursa memorie**: Memoria este utilizată de stiva de apeluri recursive. Fiecare apel al funcției adaugă un "stack frame" care conține parametri, variabile locale și adresa de returnare. Spațiul pentru un frame este constant ($O(1)$).
- **Adâncimea recursiei**: Există $n + 1$ apeluri recursive (de la n descrescător către 0).
- **Complexitate**: $S(n) = (n + 1) \cdot O(1) \in O(n)$.

Concluzie: Complexitatea de timp este $O(n)$, iar complexitatea de spațiu este $O(n)$.

5. Ce complexitate are următorul algoritm al căutării binare [3] ilustrat în continuare?

```
int cautareBinara(int[] vector, int cheie) {
    int stanga = 0;
    int dreapta = vector.length - 1;
    while (stanga <= dreapta) {
        int mijloc = stanga + (dreapta - stanga) / 2;
        if (vector[mijloc] == cheie) {
            return mijloc;
        } else if (vector[mijloc] < cheie) {
            stanga = mijloc + 1;
        } else {
            dreapta = mijloc - 1;
        }
    }
    return -1;
}
```

Rezolvare:

- **Operația dominantă**: Comparația `vector[mijloc] == cheie` (o operație $O(1)$ pe iterație).
- **Reducerea spațiului de căutare**: La fiecare iterație, spațiul de căutare (de dimensiune $dreapta - stanga + 1$) se înjumătățește.
- **Număr maxim de iterații**: Dimensiunea inițială este n. După k iterații, dimensiunea spațiului este cel mult $n/2^k$. Căutarea se oprește când acest spațiu devine mai mic de 1: $n/2^k < 1 \Rightarrow 2^k > n \Rightarrow k > \log_2 n$.
- **Complexitate**: Numărul de iterații este de ordinul $O(\log n)$. Deoarece fiecare iterație efectuează un număr constant de operații, complexitatea totală este $O(\log n)$.
- **Cazuri**: Complexitatea este $O(\log n)$ atât pentru cazul favorabil (elementul la prima înjumătățire), cât și pentru cel defavorabil (elementul nu există sau este la una dintre capete), deoarece procesul de înjumătățire continuă până când spațiul este epuizat.

Concluzie: Complexitatea timp este $O(\log n)$. Complexitatea spațiu este $O(1)$ (algoritm iterativ, folosește doar un număr fix de variabile).

2.3 Probleme propuse

Pentru următoarele probleme analizați complexitatea de timp și de spațiu.

*

1. ```
   count = 0;
   for(int i = n; i > 0; i--) {
       count += i * i;
   }
   ```

2. ```
   product = 1;
   for(int i = 0; i < 100; i += 5) {
       product *= i * 2;
   }
   ```

3. ```
   for(float x = 0.0f; x < 10.0f; x += 0.5f) {
       y = x * x + 2 * x + 1;
       std::cout << "(" << x << "," << y << ")\n";
   }
   ```

4. ```
   sum = 0, product = 1;
   for(int i = 0; i < n; i++) {
       sum += i;
       }
   for(int j = 0; j < m; j++) {
       product *= j;
       }
   ```

**

1. ```
   for(int i = 0; i < n; i++) {
       counter = 0;
       for(int j = 0; j < m; j++) {
           counter += j;
       }
       std::cout << counter << "\n";
   }
   ```

2. ```
   for(i=1;i<=n;i++)
       for(j=1;j<=3*i;j++)
           for(k=1;k<=3*n;k*=2)
               std::cout<<"*";
   ```

3.
```
count = 0;
for(int a = 0, b = 10; a < n && b > 0; a++, b--) {
    count++;
}
```

4.
```
int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

***

1.
```
for(int i = 1; i < m; i++)
    for(int j = 1; i < p; i++) {
        c[i][j]=0;
        for(int k = 1; i < n; i++)
            c[i][j]+=a[i][k]*b[k][j];
    }
```

2.
```
i = 1;
while(x[i]!=v && i<n)
    i++;
if(x[i]==v) r = true;
else r = false;
return r;
```

3.
```
count = 0;
for(int i=0;i<n;i++)
    for(int j=i;j>0;j--)
        count++;
```

4.
```
s=0, x=1;
for(int i=1;i<=n;i++) {
    s+=i; i=2*i-1;
    if(x<s) x++;
}
```

****

1.
```
count = 0;
while(count < n) {
    count += (count % 3 == 0) ? 1 : 2;
}
```

2.
```
i=0, j=0;
for(;i<n;++i)
    while(j<n && arr[i]<arr[j]) j++;
```

3.
```
count=0;
for(lv=n;lv>0;lv/=2) {
    inner_lv=0;
    while(inner_lv<lv) { inner_lv++; count++; }
}
```

4.
```
for(i=0;i<n;i*=3)
    if(x<y) for(j=i;j<n*n;j*=2)
        std::cout << j << " ";
    else for(k=n;k>i;k--)
        std::cout << k << " ";
```

*****

1.
```
for(int i = 1; i * i <= n; i++) {
    if(i % 2 == 0) {
        for(int j = i; j <= n; j += i * i) {
            int complex_var = (i + j) * (i - j);
            if(complex_var > 0) {
                for(int k = 0; k < complex_var % 10; k++) {
                    int nested_temp = k * k;
                    if(nested_temp % 3 == 0) {
                        std::cout << "*";
                    }
                }
            }
        }
    }
    else {
        for(int j = n; j >= 1; j /= 2) {
            std::cout << "%";
        }
    }
}
```

2.
```
for(int i = 1; i < n; i *= 2) {
    for(int j = i; j > 0; j /= 3) {
        if(j % 2 == 0) {
            for(int k = j; k < n; k += k) {
                int counter = 0;
```

```
                while(counter < k * k) {
                    if(counter % i == j % i) {
                        counter += (i + j) / 2;
                    }
                    else {
                        counter += i % (j + 1);
                    }
                }
            }
        }
    }
}
```

3.
```
void recursiveNested(int depth, int max_depth, int branch) {
    if(depth >= max_depth) return;

    for(int i = 0; i < branch * depth; i++) {
        if(i % (depth + 1) == 0) {
            for(int j = i; j > 0; j = (j + 1) / 2) {
                recursiveNested(depth + 1, max_depth, branch
                   + j % 2);
            }
        } else {
            for(int k = 1; k <= i * i; k *= 2) {
                int temp = k;
                while(temp > 0) {
                    temp /= (depth + 2);
                }
            }
        }
    }
}
```

Capitolul 3

Pointeri. Alocarea memoriei. Fișiere

3.1 Aspecte teoretice

3.1.1 Pointeri în C++

Memoria RAM este alcătuită dintr-o succesiune de octeți, fiecare octet fiind caracterizat de poziția acestuia în memorie (asemenea elementelor dintr-un vector, doar că aceste valori sunt reprezentate folosind baza 16) și poartă denumirea de adresă. Fiecare variabilă ocupă în memorie un anumit număr de octeți consecutivi (în funcție de tipul variabilei), iar adresa acesteia este dată de valoarea adresei primul octet în care este salvată valoarea variabilei. Cunoscând adresa unei variabile se poate accesa cu ușurință valoarea acesteia.

Pointerii sunt variabile care stochează adresa de memorie a unei alte variabile, sau altfel spus sunt cele care care indică (*points at*) către variabila a cărei adresă este stocată în interiorul variabilei de tip pointer. Un exemplu este ilustrat în fig.3.1.

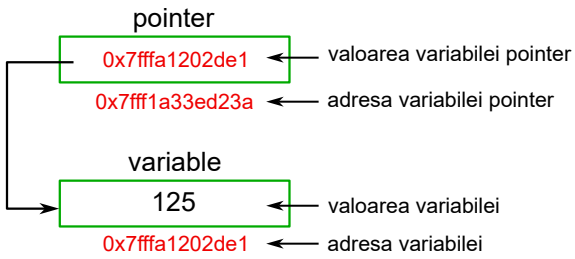


Figura 3.1: Pointer. Reprezentare adaptată după „C++ Pointers”, GeeksforGeeks, disponibil la: https://www.geeksforgeeks.org/cpp/cpp-pointers/ (accesat la 20 octombrie 2026)

.

Exemplu

```
#include <iostream>
int main(){
    int a = 7;
    int *adr = &a;
```

```
    std::cout << "La adresa " << adr;
    std::cout<<" avem valoarea" << a << '\n';
    return 0;
}
```

Pentru variabilele de tip pointer se pot utiliza doi operatori unari esențiali: **&** (operatorul de referențiere) și ***** (operatorul de indirectare). Primul dintre aceștia, operatorul **&**, este folosit pentru a crea o referință către o variabilă existentă. În momentul în care se declară o variabilă de tip referință, aceasta trebuie neapărat inițializată, deoarece o referință reprezintă, în esență, un alt nume pentru aceeași zonă de memorie. Cu alte cuvinte, două nume diferite pot indica aceeași valoare stocată în memorie, iar modificarea printr-unul dintre ele se reflectă automat și asupra celuilalt.

Exemplu:

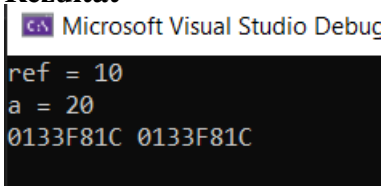
```
#include <iostream>

int main()
{
    int a = 7;
    // ref este o referinta la a.
    int& ref = a;

    a = 10;
    std::cout << "ref = " << ref << '\n';

    ref = 20;
    std::cout << "a = " << a << '\n';
    return 0;
}
```

Rezultat



Acest exemplu arată cum ref devine un alias pentru a și nu reprezintă o variabilă nouă. Când modificăm valoarea lui a, se modifică implicit și valoarea accesată prin ref, iar dacă atribuim o valoare nouă prin ref, aceasta se reflectă imediat în a. De asemenea, operatorul **&** permite obținerea adresei unei variabile, adică a locației de memorie unde aceasta este stocată.

Cel de-al doilea operator, * (operatorul de indirectare), oferă accesul la valoarea aflată în zona de memorie indicată de un pointer. Dacă operatorul **&** furnizează adresa unei variabile, operatorul * permite accesarea conținutului de la acea adresă. În exemplul următor, funcția `swap` primește adresele a două variabile și le interschimbă valorile direct, folosind dereferențierea pointerilor.

```
#include <iostream>
void swap(int *a, int *b){
    int aux = *a;
    *a = *b;
    *b = aux;
}
int main(){
    int x = 10, y = 12;
    swap(&x, &y);
    std::cout<<x<<" "<<y<<'\n';
    return 0;
}
```

Această abordare evidențiază utilitatea pointerilor în transmiterea eficientă a datelor către funcții, fără a crea copii inutile ale variabilelor. Accesul direct la adrese permite manipularea conținutului memoriei într-un mod flexibil și rapid.

Avantajele pointerilor

Pointerii aduc o serie de beneficii importante în programare. În primul rând, ei permit o gestionare precisă a memoriei, oferind posibilitatea de alocare și eliberare manuală a spațiului utilizat. În al doilea rând, utilizarea pointerilor crește eficiența aplicațiilor, deoarece valorile pot fi modificate direct, fără a fi nevoie de copii suplimentare. În sfârșit, pointerii reprezintă fundamentul pentru implementarea structurilor dinamice complexe, precum listele înlănțuite, arborii binari sau grafurile, făcând posibilă construirea de structuri de date flexibile și scalabile.

Exemplu explicat

```
// Declarare variabila si pointer
int num = 10;
int* ptr = &num;

std::cout << "Valoarea lui num: " << num << std::endl;
std::cout << "Adresa lui num: " << &num << std::endl;
std::cout << "Adresa stocata in ptr: " << ptr << std::endl;
```

```
std::cout << "Valoarea indicata de ptr: " << *ptr << std::endl;
```

În acest exemplu, variabila num este declarată de tip `int`, iar pointerul ptr este inițializat cu adresa sa, folosind operatorul **&**. Apoi, operatorul * este folosit pentru a accesa valoarea aflată la adresa stocată în ptr. Astfel, ptr nu reține valoarea în sine, ci locația din memorie unde aceasta este stocată. Prin intermediul pointerilor, putem citi sau modifica direct conținutul memoriei, ceea ce reprezintă un instrument puternic în programarea de nivel scăzut, dar și o responsabilitate importantă pentru evitarea erorilor legate de accesul necorespunzător la memorie. Prin intermediul pointerului putem accesa sau modifica direct conținutul memoriei.

Câteva resurse bibliografice suplimentare sunt:

1. https://www.geeksforgeeks.org/references-in-c/

2. https://www.geeksforgeeks.org/pointers-vs-references-cpp/

Alocarea memoriei. Noțiuni generale

Orice variabilă declarată într-un program ocupă o anumită zonă de memorie destinată stocării valorii sale. Procesul prin care sistemul de operare rezervă acest spațiu se numește **alocare a memoriei**, iar momentul în care acel spațiu este eliberat se numește **dealocare**. În limbajul C++, aceste operații pot avea loc în momente diferite, în funcție de modul de gestionare ales de programator. Există două modalități principale de alocare: statică și dinamică, fiecare cu particularitățile și avantajele sale.

3.1.2 Alocarea statică a memoriei

Alocarea statică presupune rezervarea spațiului de memorie în momentul compilării programului. Cu alte cuvinte, dimensiunea și locațiile exacte ale variabilelor sunt determinate înainte de rularea efectivă a codului. Aceasta înseamnă că memoria este gestionată automat de compilator, fără intervenția programatorului.

Această metodă are avantajul simplității: variabilele sunt create și distruse automat la intrarea și ieșirea din zona lor de vizibilitate. În plus, accesul la date este foarte rapid, deoarece adresele sunt cunoscute și fixe pe durata execuției. Totuși, o limitare importantă este lipsa de

flexibilitate. Dimensiunea memoriei alocate nu poate fi modificată în timpul rulării programului.

Exemplu de alocare statică

```
#include <iostream>

int main() {
    int x = 5; // variabila alocata static
    std::cout << "Valoare: " << x << std::endl;
    return 0;
}
```

În exemplul de mai sus, variabila x este creată automat la începutul execuției funcției `main()` și eliberată odată cu terminarea acesteia. Programatorul nu trebuie să gestioneze manual memoria, iar compilatorul se ocupă complet de rezervarea și eliberarea spațiului necesar.

Alocarea statică se remarcă prin eficiență și simplitate, fiind ideală pentru programele în care dimensiunea datelor este cunoscută dinainte. Totuși, limitarea sa constă în faptul că nu se poate adapta la situațiile în care este nevoie de o cantitate variabilă de memorie sau de structuri de date create în funcție de inputul utilizatorului.

3.1.3 Alocarea dinamică a memoriei

Alocarea dinamică intervine atunci când spațiul de memorie este rezervat în timpul rulării programului. Aceasta permite o flexibilitate mult mai mare, deoarece dimensiunea memoriei poate fi determinată în funcție de datele introduse de utilizator sau de evoluția logicii programului. În C++, acest tip de alocare se realizează cu ajutorul operatorilor `new` și `delete`.

Atunci când se folosește operatorul `new`, sistemul alocă un spațiu de memorie suficient pentru tipul de date specificat și returnează un pointer către acea zonă. La final, pentru a evita pierderile de memorie (*memory leaks*), spațiul trebuie eliberat explicit prin operatorul `delete`.

Exemplu de alocare dinamică

```
#include <iostream>

int main() {
    int* p = new int; // alocare dinamica
    *p = 25;
    std::cout << "Valoarea stocata: " << *p << std::endl;
    delete p; // eliberare memorie
```

```
    return 0;
}
```

În acest exemplu, variabila p este un pointer către un spațiu de memorie alocat în mod dinamic. Valoarea 25 este stocată în acea locație, iar la final programatorul este responsabil să elibereze memoria folosind `delete`. Această practică este esențială pentru a preveni acumularea de memorie neutilizată.

Alocarea dinamică se dovedește indispensabilă în aplicațiile care necesită structurarea datelor într-un mod flexibil: tabele, liste, arbori sau alte structuri complexe care își modifică dimensiunea pe durata execuției. Totuși, această metodă presupune o atenție sporită din partea programatorului, deoarece orice neglijență poate duce la erori severe sau scurgeri de memorie.

În concluzie, diferența fundamentală dintre alocarea statică și cea dinamică rezidă în momentul în care memoria este rezervată și în gradul de control pe care programatorul îl are asupra acesteia. Alocarea statică este simplă și eficientă, potrivită pentru situații bine definite, în timp ce alocarea dinamică oferă libertate și adaptabilitate, dar cere o gestionare atentă. Pointerii constituie elementul central în înțelegerea și utilizarea corectă a acestor concepte, fiind instrumentul prin care programatorul poate accesa și manipula direct memoria, transformând modul de lucru cu datele într-un proces flexibil și precis.

3.1.4 Fișiere text

Fișierele text reprezintă una dintre cele mai uzuale modalități de a salva și încărca informații persistente într-o aplicație C++. Acestea permit stocarea datelor într-un format lizibil, ceea ce face posibilă atât interpretarea de către program, cât și examinarea manuală de către utilizator. În C++, lucrul cu fișiere este gestionat prin biblioteca standard `<fstream>`, care oferă trei clase principale: `ifstream` pentru citirea datelor, `ofstream` pentru scrierea lor și `fstream` pentru operații bidirecționale.

Deschiderea fișierelor

Înainte de a putea citi sau scrie într-un fișier, acesta trebuie deschis. În C++, acest lucru se poate realiza fie direct prin constructorul obiectului de tip flux, fie ulterior, folosind metoda `open()`. De exemplu, o instanță de `ifstream` deschide automat fișierul specificat, iar dacă dorim o deschidere explicită, putem apela funcția `open()` după declararea fluxului.

```
#include <fstream>
#include <iostream>

int main() {
    std::ifstream fin("input.txt");
    if (!fin.is_open()) {
        std::cout << "Eroare la deschiderea fisierului!" <<
           std::endl;
        return 1;
    }

    string cuvant;
    while (fin >> cuvant) {
        std::cout << cuvant << std::endl;
    }

    fin.close();
    return 0;
}
```

În exemplul de mai sus, fișierul `input.txt` este deschis pentru citire. Programul verifică dacă deschiderea a reușit și apoi parcurge conținutul cu ajutorul operatorului », afișând fiecare cuvânt pe o linie separată. La final, fișierul este închis manual prin apelul `close()`.

Scrierea în fișiere

Pentru scriere, se folosește clasa `ofstream`, iar operațiile sunt similare cu cele realizate prin `std::cout`. Datele sunt trimise în fluxul de ieșire și, implicit, scrise în fișierul asociat.

```
#include <fstream>
#include <iostream>

int main() {
    std::ofstream fout("output.txt");
    if (!fout) {
        std::cout << "Nu s-a putut crea fisierul." << std::endl;
        return 1;
    }
    fout << "Acesta este un test." << std::endl;
    fout << "Linia a doua." << std::endl;
    fout.close();
    return 0;
}
```

Programul creează fișierul `output.txt` și scrie două linii de text. Este important de observat că, în mod implicit, un fișier deschis cu `ofstream` este trunchiat, adică orice conținut anterior este șters.

Moduri de deschidere a fișierelor

Fișierele pot fi deschise în diverse moduri, controlate prin așa-numitele „flag-uri” (valori din `std::ios`). Acestea specifică scopul și comportamentul fluxului — citire, scriere, adăugare la sfârșit sau deschidere în mod binar. De exemplu:

- `std::ios::in` – deschidere pentru citire
- `std::ios::out` – deschidere pentru scriere
- `std::ios::ate` – deschidere la sfârșitul fișierului
- `std::ios::app` – scriere la finalul fișierului (append)
- `std::ios::trunc` – trunchiere (șterge conținutul existent)
- `std::ios::binary` – mod binar

```
std::ofstream fout("fisier.txt", std::ios::app); // scriere la
   sfarsit fara stergerea continutului anterior
```

Testarea stării fișierelor

Pentru a verifica dacă un fișier este accesibil și dacă operațiile au reușit, pot fi utilizate metode precum `is_open()`, `eof()`, `fail()` sau `good()`. Aceste funcții oferă un control mai bun asupra fluxului de date și ajută la gestionarea erorilor.

- `is_open()` – verifică dacă fișierul este deschis;
- `eof()` – verifică dacă s-a atins sfârșitul fișierului;
- `fail()` – eroare la operații;
- `good()` – totul funcționează corect.

Citirea rând cu rând și caracter cu caracter

În afară de citirea bazată pe operatorul », limbajul C++ oferă funcții dedicate pentru preluarea datelor linie cu linie sau caracter cu caracter.

Citirea linie cu linie se realizează cu `std::getline()`, care preia o întreagă linie de text până la caracterul de sfârșit de linie:

```
std::string linie;
while (std::getline(fin, linie)) {
    std::cout << linie << std::endl;
}
```

Dacă dorim o citire caracter cu caracter, putem utiliza metoda `get()`, ideală pentru analizarea precisă a conținutului:

```
char c;
while (fin.get(c)) {
    std::cout << c;
}
```

Fișiere bidirecționale cu `fstream`

Clasa `fstream` oferă atât funcționalitatea de citire, cât și cea de scriere în același fișier, fiind potrivită atunci când avem nevoie de acces bidirecțional. Prin combinarea flag-urilor `std::ios::in` și `std::ios::out`, putem deschide același fișier pentru ambele scopuri:

```
std::fstream f("fisier.txt", std::ios::in | std::ios::out);
f << "Scriem text.\n";
f.seekg(0); // revenim la inceput
std::string linie;
std::getline(f, linie);
std::cout << linie << std::endl;
f.close();
```

În acest exemplu, după scriere, poziția de citire este mutată la începutul fișierului prin funcția `seekg()`, iar conținutul este apoi citit și afișat.

Poziționarea în fișiere

C++ permite controlul precis al poziției de citire sau scriere prin intermediul funcțiilor `seekg()`, `seekp()`, `tellg()` și `tellp()`. Acestea sunt utile atunci când dorim să accesăm anumite secțiuni ale fișierului, să calculăm dimensiunea acestuia sau să rescriem doar o parte din conținut.

- `seekg()` – mută cursorul pentru citire;
- `seekp()` – mută cursorul pentru scriere;
- `tellg()` – returnează poziția actuală pentru citire;

- `tellp()` – returnează poziția actuală pentru scriere.

Exemplu:

```
std::ifstream fin("text.txt");
fin.seekg(0, std::ios::end);
int size = fin.tellg();
std::cout << "Dimensiunea fisierului: " << size << " bytes." <<
   std::endl;
```

Codul determină dimensiunea fișierului în octeți, mutând cursorul la sfârșit și preluând poziția curentă a acestuia.

Bune practici

În lucrul cu fișiere este recomandat să se respecte câteva reguli esențiale: fișierele trebuie închise întotdeauna după utilizare, iar validarea deschiderii lor trebuie realizată de fiecare dată. Pentru citirea liniilor complete se recomandă `getline()`, iar pentru scriere repetată fără pierderea conținutului existent se folosește modul `ios::app`. De asemenea, trebuie evitată citirea necontrolată, fără verificarea stării fluxului, pentru a preveni erori de interpretare a datelor.

Prin utilizarea corectă a acestor tehnici, programatorul poate manipula eficient fișiere text în C++, asigurând o gestionare sigură și coerentă a datelor externe.

(*) Citirea dintr-un fișier foarte mare

Să comparăm două metode de citire a conținutului unui fișier text foarte mare (miliarde de caractere) într-un `std::string`:

- Citirea caracter cu caracter (`file.get(c)`)
- Citirea eficientă cu buffer și prealocare

Varianta 1: Citirea caracter cu caracter

```
#include <fstream>
#include <iostream>
#include <string>

std::string citireCaracterCuCaracter(const std::string&
   numeFisier) {
    std::ifstream fisier(numeFisier);
    if (!fisier) {
        std::cerr << "Nu s-a putut deschide fisierul.\n";
```

```
        return "";
    }

    std::string rezultat;
    char caracter;
    while (fisier.get(caracter)) {
        rezultat += caracter; // la fiecare adaugare de caracter
           se realoca tot sirul
    }

    return rezultat;
}
```

- Complexitate teoretică: **O(n^2)** în cazuri nefavorabile (realocări frecvente de memorie).
- Fiecare concatenare `result += c` poate duce la o realocare și copiere completă a șirului.
- Ineficient pentru fișiere mari.

Varianta 2: Citirea rapidă cu prealocare

```
#include <fstream>
#include <iostream>
#include <string>

std::string citireTextDinFisier(const std::string& numeFisier) {
    std::ifstream fisier(numeFisier, std::ios::ate); // deschidere
       la final
    if (!fisier) {
        std::cerr << "Nu s-a putut deschide fisierul.\n";
        return "";
    }

    std::streamsize dimensiune = fisier.tellg(); // dimensiunea
       fisierului
    std::cout << "Dimensiunea fisierului este " << dimensiune <<
       "\n";
    fisier.seekg(0, std::ios::beg); // revenim la inceput

    std::string data(dimensiune, '\0'); // prealocam bufferul
    fisier.read(&data[0], dimensiune);    // citim direct in buffer

    return data;
}
```

- Complexitate: **O(n)** (citire într-un buffer prealocat)

- Eficientă, deoarece:
  - Evită realocările frecvente ale stringului.
  - Accesează discul secvențial.
  - Utilizează un singur apel `read()`.
- Ideală pentru fișiere de dimensiuni medii și mari.

Comparare directă

Caracteristică	**Caracter cu caracter**	**Citire eficientă**
Timp execuție	Ridicat ($O(n^2)$)	Redus (O(n))
Memorie	Crește incremental	Prealocare optimă
Apeluri la sistem	Multe (per caracter)	Unul (buffer întreg)
Complexitate	Ineficientă	Eficientă
Ușor de implementat	Da	Da, dar necesită `seekg`

Deși citirea caracter cu caracter este intuitivă și simplă, ea devine rapid ineficientă pentru fișiere mari. Metoda bazată pe `seekg` și `read` este semnificativ mai performantă, atât ca timp de execuție, cât și ca utilizare a resurselor. În aplicații reale sau în proiecte unde performanța contează, se recomandă folosirea metodei optimizate.

Concluzie

Gestionarea fișierelor este esențială în C++ pentru aplicații care implică stocarea și prelucrarea datelor. Biblioteca `<fstream>` oferă unelte flexibile pentru a lucra eficient cu fișiere text. Înțelegerea fluxurilor de intrare și ieșire este importantă pentru dezvoltarea aplicațiilor robuste și scalabile.

3.2 Probleme propuse - Pointeri. Alocare memorie

Specificați ce se va afișa în consolă în urma execuției următoarelor secvențe de cod.

*

1. ```
   #include <iostream>
   int main()
   {
       int a = 3;
       int* p = &a;
       std::cout << a << std::endl;
       std::cout << p << std::endl;
       std::cout << *p << std::endl;
       std::cout << &p << std::endl;
       std::cout << &*p << std::endl;
       std::cout << *&p << std::endl;
       return 0;
   }
   ```

2. ```
   #include <iostream>
   int main()
   {
       int a = 3;
       int* p = &a;
       int* q = p;
       *q = 4;
       std::cout << a << std::endl;
       std::cout << p << std::endl;
       std::cout << q << std::endl;
       std::cout << *p << std::endl;
       std::cout << *q << std::endl;
       return 0;
   }
   ```

3. ```
   #include <iostream>
   int main()
   {
       int x = 5;
       int y = 10;
       int* px = &x;
       int* py = &y;

       *px = *py;
   ```
   ```

```
    px = py;

    std::cout << "x = " << x << std::endl;
    std::cout << "y = " << y << std::endl;
    std::cout << "*px = " << *px << std::endl;
    std::cout << "*py = " << *py << std::endl;
    std::cout << "px = " << px << std::endl;
    std::cout << "py = " << py << std::endl;

    return 0;
}
```

**

1. ```
   #include <iostream>

   void f1(int a)  { a = 2;  }
   void f2(int& a) { a = 2;  }
   void f3(int* a) { *a = 2; }

   int main()
   {
       int a = 3, b = 3, c = 3;
       f1(a);
       f2(b);
       f3(&c);

       std::cout << a << std::endl << b << std::endl << c <<
          std::endl;

       return 0;
   }
   ```

2. ```
   #include <iostream>

   void increment1(int a)   { a += 1; }
   void increment2(int& a)  { a += 1; }
   void increment3(int* a)  { (*a) += 1; }

   int main()
   {
       int x = 5, y = 5, z = 5;
       increment1(x);
       increment2(y);
       increment3(&z);
   ```

```
    std::cout << x << std::endl;
    std::cout << y << std::endl;
    std::cout << z << std::endl;

    return 0;
}
```

3.
```
#include <iostream>

void swap1(int a, int b){
    int temp = a;
    a = b;
    b = temp;
}
void swap2(int& a, int& b){
    int temp = a;
    a = b;
    b = temp;
}
void swap3(int* a, int* b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main()
{
    int m = 1, n = 2;
    swap1(m, n);
    std::cout << "Dupa swap1: " << m << " " << n << std::endl;
    swap2(m, n);
    std::cout << "Dupa swap2: " << m << " " << n << std::endl;
    swap3(&m, &n);
    std::cout << "Dupa swap3: " << m << " " << n << std::endl;
    return 0;
}
```

***

1.
```
#include <iostream>

int main() {
    int arr[] = {10, 20, 30, 40};
    int *p = arr;
    std::cout << *p++ << std::endl;
```

```
    std::cout << *++p << std::endl;
    std::cout << (*p)++ << std::endl;
    std::cout << ++(*p) << std::endl;
    return 0;
}
```

2.
```
#include <iostream>

void modify(int* p) {
    *p += 10;
    p++;
    *p *= 2;
}

int main() {
    int arr[] = {1, 2, 3, 4};
    int* ptr = arr;

    modify(ptr);

    for(int i = 0; i < 4; i++)
        std::cout << arr[i] << " ";
    return 0;
}
```

3.
```
#include <iostream>

void update(int* p, int n) {
    for(int i = 0; i < n; i++) {
        *(p + i) += i;
    }
    *(p + 1) *= 3;
}

int main() {
    int arr[] = {2, 4, 6, 8};
    int* ptr = arr;
    update(ptr, 4);
    for(int i = 0; i < 4; i++)
        std::cout << arr[i] << " ";
    return 0;
}
```

****

1. ```
   #include <iostream>
   int main() {
       int a = 5;
       int *p = &a;
       int **q = &p;
       **q = 10;
       *p = 20;
       int *r = new int(30);
       *q = r;
       std::cout << a << std::endl;
       std::cout << *p << std::endl;
       std::cout << **q << std::endl;
       delete r;
       return 0;
   }
   ```

2. ```
   #include <iostream>
   int main() {
       int x = 7;
       int* p = &x;
       int** q = &p;

       **q += 3;
       int* r = new int(15);
       *q = r;
       **q += 5;

       std::cout << x << std::endl;
       std::cout << *p << std::endl;
       std::cout << **q << std::endl;
       delete r;
       return 0;
   }
   ```

3. ```
   #include <iostream>
   void setValue(int*** ptr) {
       int* temp = new int(50);
       **ptr = *temp;
       delete temp;
   }
   int main() {
       int a = 10;
       int* p = &a;
   ```
   ```

```
    int** q = &p;

    setValue(&q);

    std::cout << a << std::endl;
    std::cout << *p << std::endl;
    std::cout << **q << std::endl;

    return 0;
}
```

*****

1. ```
   #include <iostream>
   void f1(int *x) { (*x)++; }
   void f2(int *x) { *x = ++(*x) * (*x)++; }
   void f3(int *x) { *x = (*x) * (*x); }
   void apply(int *arr, int size) {
       int *x, *y, *z;
       for (int i = 0; i < size; i++) {
           x = y = z = &arr[i];
           f1(x);  std::cout << *x << " ";
           f2(y);  std::cout << *y << " ";
           f3(z);  std::cout << *z << std::endl;
       }
   }

   int main() {
       int arr[] = {1, 2, 3, 4};
       apply(arr, 4);
       return 0;
   }
   ```

2. ```
   #include <iostream>
   void g1(int* x) { *x += 2; }
   void g2(int* x) { *x = (*x)++ + ++(*x); }
   void g3(int* x) { *x = (*x) * 2 - 1; }
   void transform(int* arr, int size) {
       for(int i = 0; i < size; i++) {
           int *a = &arr[i];
           int *b = &arr[i];
           int *c = &arr[i];
           g1(a); std::cout << *a << " ";
           g2(b); std::cout << *b << " ";
           g3(c); std::cout << *c << std::endl;
   ```

```
    }
}
int main() {
    int arr[] = {1, 2, 3, 4};
    transform(arr, 4);
    return 0;
}
```

3.
```
#include <iostream>
void h1(int** x) { **x += 1; }
void h2(int** x) { **x = ++(**x) * (**x)++; }
void h3(int** x) { **x = (**x) * 3 - 2; }
void process(int* arr, int size) {
    for(int i = 0; i < size; i++) {
        int* p = &arr[i];
        int** pp1 = &p;
        int** pp2 = &p;
        int** pp3 = &p;
        h1(pp1); std::cout << **pp1 << " ";
        h2(pp2); std::cout << **pp2 << " ";
        h3(pp3); std::cout << **pp3 << std::endl;
    }
}
int main() {
    int arr[] = {1, 2, 3, 4};
    process(arr, 4);
    return 0;
}
```

Capitolul 4

Funcții

4.1 Aspecte teoretice

Funcțiile sunt blocuri de cod reutilizabile în C++ și sunt fundamentale în dezvoltarea de programe eficiente și organizate. Ele reprezintă unul dintre cele mai importante concepte în programare, servind ca piloni ai structurării aplicațiilor.

Unul dintre cele mai valoroase avantaje ale funcțiilor este reutilizarea codului. Acest principiu permite programatorilor să scrie o logică specifică o singură dată și să o utilizeze de multiple ori în diferite părți ale programului, economisind timp și reducând substanțial riscul de erori. Prin îmbunătățirea citirii și înțelegerii codului, funcțiile transformă un program monolit într-o colecție de sarcini logice și bine definite, fiecare cu un scop clar, ceea ce face codul mult mai accesibil atât pentru autor, cât și pentru alți dezvoltatori.

Dezvoltarea software-ului beneficiază enorm de modularitatea oferită de funcții. Această abordare permite împărțirea unui program complex și mare în module mai mici, ușor de gestionat, testat și întreținut. Strâns legat de modularitate este conceptul de abstracție, unde funcțiile pot ascunde detalii de implementare complexe în spatele unei interfețe simple. Acest lucru permite utilizatorului (altui programator) să se concentreze pe ce face funcția, nu pe cum o face, separând eficient preocupările.

O problemă comună în programare, rezolvarea sarcinilor repetitive, este adresată direct de utilizarea funcțiilor. Ele elimină necesitatea de a rescrie același cod de multiple ori, centralizează logica și, în cazul necesității unei modificări, aceasta se face într-un singur loc, reducând efortul și potențialele inconsistențe. De asemenea, funcțiile oferă oportunități unice de optimizare. Deoarece fiecare funcție este o unitate distinctă, ea poate fi analizată și optimizată individual pentru a crește performanța generală a programului, fie prin îmbunătățiri algoritmice, fie prin tehnici avansate de compilare.

Flexibilitatea funcțiilor este un alt avantaj major; ele pot fi apelate și orchestrate din orice parte a programului, oferind un control precis asupra fluxului de execuție și al logicii de business. Această flexibilitate se extinde și la conceptul de recursivitate, unde o funcție are capacitatea de a se auto-apela. Această tehnică este indispensabilă pentru rezolvarea elegantă a unor clase specifice de probleme complexe (de exemplu, parcurgerea arborilor sau calculul factorialului) care sunt definibile în termeni de cazuri mai simple ale lor înșile.

În cele din urmă, funcțiile constituie fundamentul pe care se construiesc bibliotecile și extensibilitatea codului. Grupate în biblioteci, ele oferă funcționalități predefinite, promovând reutilizarea la scară largă între proiecte și facilitând colaborarea eficientă în cadrul echipelor. Aceasta permite dezvoltatorilor să se bazeze pe cod verificat și robust, accelerând semnificativ procesul de dezvoltare.

4.1.1 Declarația unei funcții

În limbajul C++, o funcție este definită printr-o declarație care trebuie să conțină mai multe elemente esențiale. Acestea sunt tipul de date al valorii returnate, numele funcției și lista de parametri.

Tipul de date al valorii returnate reprezintă tipul de valoare pe care funcția o va întoarce după execuție. Acesta poate fi un tip de bază precum `int` (pentru numere întregi), `double` (pentru numere reale) sau `char` (pentru caractere). De asemenea, poate fi un tip de date mai complex, precum o structură (`struct`), o clasă (`class`), sau un tip definit de utilizator. Există și cazul special al tipului `void`, care indică faptul că funcția nu returnează nicio valoare.

Numele funcției este un identificator care se recomandă a fi ales astfel, încât să reflecte în mod clar acțiunea pe care o realizează. Este o bună practică ca numele să fie un verb sau să înceapă cu un verb, cum ar fi `calculeazaSuma`, `setDate` sau `getValoare`. Există două convenții populare de scriere: **camelCase** (unde fiecare cuvânt, exceptând primul, începe cu literă mare, de ex. `verificaParola`) și **snake_case** (unde cuvintele sunt separate prin underscore și scrise cu litere mici, de ex. `verifica_parola`). Alegerea unei convenții și respectarea ei consecventă în întregul proiect este esențială pentru lizibilitate. Este important să se evite nume generice sau ambigue precum `functie1()` sau `procesare()`. Pentru funcțiile care returnează valori booleane (adevărat/fals), se recomandă utilizarea unor prefixe descriptive precum `is`, `has` sau `can` (de ex. `isValid`, `hasPermission`, `canExecute`), ceea ce face codul extrem de intuitiv.

Lista de parametri definește datele de intrare pe care funcția le primește pentru a-și îndeplini sarcina. Denumirea parametrilor trebuie să fie la fel de sugestivă ca și numele funcției, evitând termeni vagi precum x sau a. Parametrii ar trebui să urmeze aceeași convenție de denumire aleasă pentru funcții. Rolul unui parametru poate fi sugerat și prin prefixe: parametrii care servesc ca ieșire (output) pot folosi prefixul `out` (ex. `outResult`), iar cei care servesc atât pentru intrare cât și pentru ieșire (input-output) pot folosi prefixul `io` (ex. `ioBuffer`). Limbajul C++ permite, de asemenea, definirea de **valori implicite** pentru parametri, ceea ce îi face opționali la apel. De exemplu, un parametru definit ca `bool verbose = false` poate fi omis la apelul funcției, iar în acest caz, valoarea sa va fi `false`. Acest lucru permite crearea de interfețe flexibile și expresive.

```
// Exemple bune (nume descriptive, clare)

double calculeazaArie(double latime, double inaltime){...}

bool esteVocala(char caracter){...}

int suma(int primaValoare, int aDouaValoare){...}
```

```
// Exemple nerecomandate (nume ambigue, lipsite de claritate)

double f(double a, double b){...}

bool g(char c){...}
```

4.1.2 Apelarea unei funcții

Apelarea unei funcții implică folosirea numelui acesteia urmat de paranteze rotunde, între care se specifică argumentele corespunzătoare, dacă există. Acest proces reprezintă punctul în care controlul executării programului este transferat către codul funcției. Argumentele furnizate trebuie să corespundă exact tipului, numărului și ordinii parametrilor formali declarați în antetul funcției. După executarea instrucțiunilor din corpul funcției, controlul revine în punctul de apel, iar dacă funcția are un tip de returnare diferit de `void`, un rezultat este produs și poate fi utilizat în expresii sau asignat unei variabile.

```
int rezultat = suma(3, 5); // Apelul functiei 'suma' cu argumente
```

În exemplul de mai sus, funcția `suma` este invocată cu argumentele concrete 3 și 5. Valoarea returnată de funcție, care în acest caz presupunem că este rezultatul adunării celor două

numere, este capturată și stocată în variabila `rezultat`. Acest mod de utilizare este tipic atunci când rezultatul calculat este necesar pentru logica ulterioară a programului.

Este important de menționat că rezultatul unei funcții care returnează o valoare nu trebuie neapărat stocat. Apelul poate fi efectuat și fără a captura rezultatul, caz în care acesta este ignorat. Acest lucru este posibil din punct de vedere sintactic, dar se practică în mod obișnuit doar atunci când funcția are efecte secundare (de exemplu, afișează ceva pe ecran, modifică starea unui obiect) și valoarea returnată este secundară sau inutilă în context.

```
suma(3, 5); // Rezultatul functiei 'suma' este ignorat
```

Utilizările funcțiilor sunt extrem de diverse și stau la baza programării structurate. Ele pot efectua operații matematice simple sau complexe, pot gestiona fluxul logic al programului prin decizii și bucle abstractizate, sau pot manipula date și structuri complexe precum vectori, liste sau obiecte. Prin apelarea unei funcții, programatorul poate executa o secvență complexă de instrucțiuni printr-o singură comandă, sporind astfel claritatea, modularitatea și eficiența codului.

4.1.3 Clasificarea funcțiilor

Clasificarea funcțiilor în C++ poate fi realizată pe baza modului în care funcțiile sunt utilizate și ce rol îndeplinesc într-un program. Această clasificare ajută la o mai bună organizare a codului și la înțelegerea modului în care diferitele părți ale programului interacționează.

Funcții operand

Funcțiile operand sunt funcții care efectuează operații și calcule specifice și returnează întotdeauna un rezultat. Scopul principal al acestor funcții este de a procesa date de intrare și de a produce o valoare de ieșire care poate fi utilizată în continuare în program. Ele sunt analoge funcțiilor matematice, unde pentru un set de intrări se obține o ieșire specifică.

Un exemplu clasic este o funcție care calculează suma a două numere. Această funcție primește doi parametri de intrare (două numere) și returnează suma lor. Alt exemplu ar putea fi o funcție care determină valoarea maximă dintre două sau mai multe numere, sau o funcție care calculează rădăcina pătrată a unui număr.

Din punct de vedere al declarației, funcțiile operand au întotdeauna un tip de date de returnare specificat explicit în antetul funcției (precum `int`, `double`, `float`, `bool`, sau orice alt tip

definit). Acest tip indică natura rezultatului pe care funcția îl va produce. De asemenea, ele pot avea parametri formali care definesc datele de intrare necesare efectuării calculului. Parametrii pot fi de orice tip și pot fi transmiși prin diferite mecanisme (prin valoare, prin referință, sau prin pointer).

```
#include <iostream>

int suma(int a, int b) {
    return a + b;
}
int main() {
    int rezultat = suma(5, 3);  // rezultatul poate fi stocat
    std::cout << suma(2, 4);    // rezultatul poate fi afisat
    return 0;
}
```

Funcții procedurale

Funcțiile procedurale sunt proiectate să efectueze o anumită acțiune sau o secvență de operații fără a returna un rezultat direct. Scopul lor principal este de a modifica starea programului prin efecte laterale, cum ar fi actualizarea variabilelor globale, scrierea într-un fișier, afișarea unui mesaj pe ecran sau modificarea structurilor de date. Deoarece nu returnează nicio valoare, ele nu pot fi utilizate în expresii sau asignate unei variabile, ceea ce le diferențiază fundamental de funcțiile operand.

Un exemplu tipic este o funcție care afișează un mesaj (cu orice conținut) pe ecran sau o funcție care sortează elementele unui vector transmis ca parametru. Aceste funcții își îndeplinesc sarcina prin acțiunea pe care o execută, iar rezultatul lor este efectul respectiv, nu o valoare returnată.

```
#include <iostream>

void afiseazaMesaj() {
    std::cout << "Acesta este un mesaj!";
}

int main() {
    afiseazaMesaj();            // corect
    // int x = afiseazaMesaj();// functia nu returneaza valori
    std::cout << afiseazaMesaj();// nu poate fi folosita in
       expresii
    return 0;
}
```

Din punct de vedere sintactic, funcțiile procedurale sunt declarate cu tipul de returnare `void`, care indică în mod explicit absența unei valori de retur. Acest lucru informează atât compilatorul, cât și programatorul că scopul funcției este executarea unei acțiuni, nu calcularea unui rezultat. Deși nu returnează valori, ele pot primi parametri care să le direcționeze comportamentul sau să le ofere datele necesare pentru a-și îndeplini sarcina.

4.1.4 Transmiterea parametrilor

În C++, parametrii pot fi transmiși unei funcții prin două metode principale: prin valoare și prin referință. Alegerea între aceste mecanisme are impact semnificativ atât asupra performanței, cât și asupra modului în care funcția interacționează cu datele externe, afectând posibilitatea de a modifica variabilele originale.

Transmiterea parametrilor prin valoare

Această metodă implică crearea unei copii locale a valorii parametrului în momentul apelului funcției. Orice modificare aplicată asupra acestui parametru în interiorul funcției rămâne locală și nu are niciun efect asupra variabilei originale din *scope*-ul apelant. Acest comportament oferă o izolare completă, asigurând că valorile originale sunt protejate de schimbări accidentale. Totuși, pentru structuri mari de date, copierea poate fi costisitoare din punct de vedere al memoriei și al timpului de execuție.

```
void afiseazaDublu(int numar) {
    std::cout << "Dublul: " << numar * 2 << std::endl;
    numar = 100;  // Modificare doar a copiei locale
}
void incrementeaza(int valoare) {
    valoare++; // Modificare doar a copiei locale
    std::cout << "Valoare in functie: " << valoare << std::endl;
}
int main() {
    int x = 5;
    afiseazaDublu(x);
    std::cout << "x dupa afiseazaDublu: " << x << std::endl; // x
       ramane 5
    incrementeaza(x);
    std::cout << "x dupa incrementeaza: " << x << std::endl; // x
       ramane 5
    return 0;
}
```

În exemplul de mai sus, atât în `afiseazaDublu` cât și în `incrementeaza`, parametrii sunt copii ale variabilei x. Modificările aduse parametrilor (atribuirea lui 100 și incrementarea) se aplică doar acestor copii locale, lăsând variabila originală x din funcția `main` complet neschimbată. Acest lucru demonstrează izolarea oferită de transmiterea prin valoare.

Transmiterea parametrilor prin referință

Această metodă permite funcției să lucreze direct cu variabila originală, fără a crea o copie. Ea se realizează prin utilizarea operatorului & în declarația parametrului. Modificările aduse parametrului în interiorul funcției se aplică direct variabilei originale. Pentru a preveni modificări accidentale ale datelor care nu trebuie schimbate (parametri de intrare), se poate folosi o **referință constantă** (`const&`), care oferă eficiența transmiterii prin referință (evitând copierea) combinată cu siguranța transmiterii prin valoare (imutabilitate).

```
void dubleaza(int &numar) {
    numar *= 2; // Modifica direct variabila originala
    std::cout << "Valoarea in functie: " << numar << std::endl;
}

void afiseazaNumar(const int& numar) { // referinta constanta
    std::cout << "Numar: " << numar << std::endl;
    // numar += 1; // Eroare de compilare - nu se poate modifica o
       referinta constanta
}

int main() {
    int x = 5;
    int numar = 42;

    std::cout << "x inainte de dubleaza: " << x << std::endl; // 5
    dubleaza(x);
    std::cout << "x dupa dubleaza: " << x << std::endl; // 10

    afiseazaNumar(numar); // Parametrul este transmis eficient,
       fara copiere
    std::cout << "Numarul ramane: " << numar << std::endl; // 42

    return 0;
}
```

În exemplul de mai sus, funcția `dubleaza` primește parametrul prin referință. Prin urmare, operația de înmulțire cu 2 se aplică direct variabilei originale x, modificându-i valoa-

rea. Funcția `afiseazaNumar` primește parametrul prin referință constantă. Acest lucru permite transmiterea eficientă a valorii (fără costul suplimentar al copierii), dar orice încercare de a modifica parametrul în interiorul funcției va rezulta într-o eroare de compilare, protejând astfel integritatea datelor originale.

Principalele avantaje ale transmiterii prin referință sunt:

- capacitatea de a modifica variabilele originale;
- evitarea costului de copiere pentru obiecte mari (precum vectori, structuri sau obiecte de clasă);
- posibilitatea de a returna mai multe valori indirect prin parametri, ca în următorul scenariu.

Să considerăm o funcție care primește două numere și returnează simultan suma și produsul lor.

```
void sumaSiProdus(int a, int b, int &suma, int &produs) {
    suma = a + b;
    produs = a * b;
}
```

Se poate și folosi `std::pair` sau tipul `struct` pentru a returna mai multe valori într-un mod mai elegant:

```
// folosind std::pair
pair<int, int> sumaSiProdusPair(int a, int b) {
    return make_pair(a + b, a * b);
}

// folosind struct
struct Rezultat {
    int suma;
    int produs;
};

Rezultat sumaSiProdusStruct(int a, int b) {
    return {a + b, a * b};
}
```

Spre deosebire de pointeri, o referință nu este o entitate independentă, ci un alias al obiectului cu care a fost inițializată. Legătura aceasta se stabilește o singură dată, la declarare, și nu poate fi ulterior schimbată pentru a desemna un alt obiect.

4.1.5 (*) Parametrii default

Parametrii cu valori implicite (parametrii default) reprezintă o facilitate puternică a C++-ului, care permite atribuirea unor valori predefinite anumitor parametri ai unei funcții. Astfel, atunci când apelantul nu specifică explicit un argument, valoarea implicită este folosită automat, rezultând o interfață de apelare mai flexibilă și mai expresivă.

Declararea parametrilor default

Declararea parametrilor cu valori implicite se face în prototipul funcției (de obicei într-un fișier header) prin atribuirea unei valori folosind operatorul =. Un aspect important este că parametrii care au valori implicite trebuie să ocupe ultimele poziții în lista de parametri. Aceasta înseamnă că, odată ce un parametru are o valoare implicită, toți parametrii care îi urmează trebuie să aibă și ei valori implicite.

```
// Exemplu declaratie corecta
void afiseaza(char* mesaj = "Buna ziua!", int repetari = 1) {
    for(int i = 0; i < repetari; i++) {
        std::cout << mesaj << std::endl;
    }
}
```

Moduri de apelare

Flexibilitatea parametrilor impliciți constă în posibilitatea de a apela funcția cu un număr variabil de argumente, de la zero până la toți parametrii.

Apelarea cu valori implicite are loc atunci când se omit argumentele pentru parametrii care au valori predefinite:

```
afiseaza();          // Afiseaza "Buna ziua!" o data
```

Apelarea cu valori personalizate permite suprascrierea valorilor implicite pentru unul sau mai mulți parametri:

```
afiseaza("Salut");  // Afiseaza "Salut" o data (se foloseste
   implicit 1 pentru repetari)

afiseaza("Hey", 3); // Afiseaza "Hey" de 3 ori (se suprascrie si
   mesajul si numarul de repetari)
```

Ordinea parametrilor este esențială. Parametrii default trebuie să fie grupați la sfârșitul listei de parametri. O declarare incorectă va genera eroare de compilare.

```
// CORECT: parametrii cu valori default sunt ultimii
void functie(int a, int b = 0, int c = 1);

// INCORECT: parametrul 'b' nu are valoare default, dar este urmat
   de unul care are ('c')
void functie(int a = 0, int b, int c = 1);
```

În ceea ce privește redefinirea în headere, valoarea implicită se specifică o singură dată, de obicei în declararea funcției (în fișierul header). Definiția funcției (în fișierul .cpp) nu trebuie să repete valorile implicite, ci doar să listeze numele parametrilor.

Avantaje și limitări

Utilizarea parametrilor default prezintă o serie de avantaje importante în dezvoltarea programelor. Un prim beneficiu constă în obținerea unui cod mai concis și mai expresiv, deoarece funcțiile pot fi apelate folosind un număr mai mic de argumente atunci când valorile implicite sunt suficiente. Astfel, programatorul poate evita transmiterea repetată a unor valori care nu se modifică frecvent, păstrând în același timp claritatea și ușurința de înțelegere a codului.

Un alt avantaj este reprezentat de flexibilitatea oferită în utilizarea funcțiilor. Aceeași funcție poate fi apelată în mai multe moduri, în funcție de necesitățile aplicației. Prin folosirea parametrilor default, o funcție poate avea un comportament general, dar poate permite și personalizarea anumitor valori atunci când utilizatorul dorește acest lucru.

De asemenea, parametrii default reduc necesitatea utilizării funcțiilor supraîncărcate (*overload*). În multe situații, în loc să fie definite mai multe variante ale aceleiași funcții cu liste diferite de parametri, poate fi folosită o singură funcție care conține valori implicite pentru argumentele opționale. Acest lucru conduce la un cod mai compact și mai ușor de întreținut.

Totuși, utilizarea parametrilor default prezintă și anumite limitări. Una dintre acestea este posibilitatea apariției ambiguităților atunci când sunt combinați cu funcții supraîncărcate. În cazul în care compilatorul nu poate determina în mod clar ce variantă a unei funcții trebuie apelată, poate apărea o eroare de compilare.

O altă limitare este alegerea unor valori implicite nepotrivite. O valoare default trebuie aleasă cu atenție, deoarece o configurare greșită poate conduce la rezultate neașteptate sau la apariția unor erori dificil de identificat în timpul execuției programului.

În plus, folosirea unui număr mare de parametri default poate afecta lizibilitatea codului. Dacă o funcție are foarte multe argumente opționale, apelurile acesteia pot deveni mai greu de urmărit, iar înțelegerea comportamentului funcției poate necesita analiza valorilor implicite definite în declarație.

Prin urmare, parametrii default reprezintă un mecanism util pentru simplificarea și flexibilizarea codului, însă trebuie utilizați echilibrat, astfel încât să nu afecteze claritatea și mentenabilitatea aplicației.

```
// Exemplu practic: calcul pret cu TVA optional
double calculeazaPret(double pret, double tva = 0.19) {
    return pret * (1 + tva);
}
int main() {
    std::cout << calculeazaPret(100) << std::endl; // 119 -
       foloseste TVA default de 19%
    std::cout << calculeazaPret(100, 0) << std::endl; // 100 -
       fara TVA
    return 0;
}
```

Acest exemplu ilustrează perfect utilitatea parametrilor default: funcția `calculeazaPret` poate fi folosită atât pentru calculul prețului cu TVA standard (cazul cel mai frecvent, fără a specifica al doilea argument), cât și pentru produse scutite de TVA (prin specificarea explicită a valorii zero).

4.1.6 (*) Supraîncărcarea funcțiilor

Supraîncărcarea funcțiilor (*function overloading*) este un mecanism puternic în C++ care permite definirea mai multor funcții cu același nume, dar cu semnături diferite. Această capacitate este extrem de utilă pentru a crea interfețe intuitive pentru operații similare care trebuie să funcționeze cu tipuri diferite de date sau cu un număr variabil de parametri, sporind astfel claritatea și expresivitatea codului.

Reguli ale supraîncărcării

Principala regulă a supraîncărcării este că semnătura funcției trebuie să fie unică. Semnătura unei funcții este definită de: numele funcției, numărul de parametri, tipul parametrilor, ordinea parametrilor.

Deși tipul valorii returnate este o componentă esențială a declarației complete a unei funcții (a *contractului* său), este important de înțeles că, în majoritatea limbajelor de programare, compilatorul nu îl utilizează ca parte a semnăturii pentru rezoluția supraîncărcării. Astfel, atunci când sunt definite două funcții cu același nume și aceeași listă de parametri, dar cu tipuri de returnare diferite, va rezulta o eroare de compilare. Aceasta se întâmplă deoarece compilatorul nu poate distinge între ele doar pe baza argumentelor furnizate la apel, creând o situație de ambiguitate.

Pentru a fi supraîncărcate în mod corect, funcțiile trebuie să difere prin:

- **Numărul de parametri** (de exemplu, o funcție cu 2 parametri vs. una cu 3)
- **Tipul parametrilor** (de exemplu, `int` vs. `double`)
- **Ordinea parametrilor** (de exemplu, `int, double` vs. `double, int`)

Exemple:

```
#include <iostream>
using namespace std;

// 1. Adunare pentru int
int aduna(int a, int b) {
    std::cout << "Versiunea int: ";
    return a + b;
}

// 2. Adunare pentru double
double aduna(double a, double b) {
    std::cout << "Versiunea double: ";
    return a + b;
}

// 3. Adunare pentru 3 parametri int
int aduna(int a, int b, int c) {
    std::cout << "Versiunea cu 3 int: ";
    return a + b + c;
}

int main() {
    std::cout << aduna(5, 3) << std::endl;      // Apel 1:
       Versiunea int
    std::cout << aduna(2.5, 3.7) << std::endl; // Apel 2:
       Versiunea double
```

```
    std::cout << aduna(1, 2, 3) << std::endl;  // Apel 3:
       Versiunea cu 3 int
    return 0;
}
```

Explicații:

- **Apel 1**: Folosește versiunea pentru `int` deoarece ambele argumente sunt literale întregi.
- **Apel 2**: Folosește versiunea pentru `double` deoarece ambele argumente sunt literale cu virgulă mobilă.
- **Apel 3**: Folosește versiunea cu 3 parametri datorită numărului de argumente.
- **Conversia implicită**: În cazul unui apel precum `aduna(2, 3.5)`, compilatorul va aplica conversii standard (în acest caz, va converti `2` la `2.0` de tip `double`) și va apela versiunea funcției pentru `double`. Dacă există mai multe posibilități de conversie la fel de bune, compilatorul nu va putea alege și va semnala o eroare de apel ambiguu.

Cele mai bune practici

Utilizarea corectă a supraîncărcării funcțiilor presupune respectarea unor principii care contribuie la menținerea unui cod clar, ușor de înțeles și de întreținut.

O primă recomandare este utilizarea supraîncărcării doar pentru operații care sunt conceptual similare și care au același scop fundamental. Funcțiile care au același nume trebuie să efectueze operații apropiate ca semnificație, diferența dintre ele fiind reprezentată doar de tipul sau numărul parametrilor primiți. Supraîncărcarea nu trebuie folosită pentru a asocia același nume unor funcții care realizează operații complet diferite, deoarece acest lucru poate reduce lizibilitatea programului și poate crea confuzie pentru utilizatori.

De asemenea, trebuie evitate supraîncărcările care pot conduce la ambiguități. Dacă un apel poate fi interpretat în mod similar de către mai multe variante supraîncărcate, apare o ambiguitate, iar compilatorul nu poate determina care variantă trebuie executată. În astfel de situații, proiectarea funcțiilor trebuie regândită pentru a elimina posibilitatea apariției unor conflicte.

Atunci când există riscul unor conversii automate nedorite, este recomandată utilizarea conversiilor explicite (*casting*). Prin specificarea clară a tipului dorit, programatorul poate indica

exact varianta funcției care trebuie utilizată și poate evita apariția apelurilor ambigue generate de conversiile implicite realizate de compilator.

O altă practică importantă este documentarea tuturor variantelor supraîncărcate ale unei funcții. Deoarece mai multe funcții folosesc același nume, este necesară precizarea rolului fiecărei variante, a parametrilor acceptați și a diferențelor de comportament dintre acestea. Comentariile și documentația corespunzătoare facilitează utilizarea corectă a funcțiilor și simplifică procesul de mentenanță a codului.

Prin urmare, supraîncărcarea funcțiilor trebuie utilizată ca un mecanism de organizare și simplificare a codului, nu ca o metodă de a ascunde diferențe majore de comportament între funcții. O proiectare atentă contribuie la obținerea unui program mai intuitiv și mai sigur.

```
// Exemplu suplimentar: functii de afisare supraincarcate

void afiseaza(int x) {
    std::cout << "Int: " << x << std::endl;
}

void afiseaza(double x) {
    std::cout << "Double: " << x << std::endl;
}

void afiseaza(const char* x) {
    std::cout << "String: " << x << std::endl;
}
```

Acest exemplu ilustrează o utilizare comună și eficientă a supraîncărcării: oferirea unui nume consistent (`afiseaza`) pentru o operație (afișarea) care trebuie să suporte o varietate de tipuri de date. Acest lucru face interfața mult mai ușor de reținut și de utilizat.

4.1.7 (*) Funcții șablon (*template*)

Funcțiile șablon (*template*) reprezintă un mecanism fundamental pentru programarea generică în C++. Ele permit scrierea de cod care operează pe orice tip de date, fără a fi necesară reimplementarea logicii pentru fiecare tip în parte. Acest lucru conduce la o reducere semnificativă a duplicării codului și la creșterea flexibilității și a capacității de reutilizare.

Definirea funcțiilor template

Declararea unei funcții template începe cu cuvântul cheie `template` urmat de lista de parametri template între paranteze unghiulare. Fiecare parametru template este precedat de cuvântul cheie `typename` (sau `class`, care este sinonim în acest context) și reprezintă un tip generic care va fi concretizat la momentul compilării.

```
template <typename T>
T maxim(T a, T b) {
    return (a > b) ? a : b;
}
```

Explicații:

- `template <typename T>` declară un parametru de tip generic numit `T`.
- Funcția `maxim` este definită să opereze pe tipul generic `T`. Ea va compara și returna maximul dintre cele două argumente.
- Tipul concret pentru `T` va fi dedus de compilator în momentul apelului funcției, pe baza tipului argumentelor transmise.

Apelarea funcțiilor template

Apelul unei funcții template poate fi făcut în două moduri: fie prin **deducere de tip**, unde compilatorul determină automat tipul `T` pe baza argumentelor, fie prin **specificare explicită** a tipului, unde programatorul forțează utilizarea unui anumit tip.

```
int main() {
    // Apel cu deducere de tip pentru int
    int a = 5, b = 10;
    std::cout << maxim(a, b) << std::endl; // 10

    // Apel cu specificare explicita a tipului
    std::cout << maxim<int>(a, b) << std::endl; // 10

    // Apel cu deducere de tip pentru double
    double x = 3.14, y = 2.71;
    std::cout << maxim(x, y) << std::endl; // 3.14

    // Apel cu specificare explicita a tipului
    std::cout << maxim<double>(x, y) << std::endl; // 3.14
    return 0;
}
```

Explicații:

- Compilatorul deduce automat tipul `T` din tipul argumentelor funcției. Pentru `maxim(a, b)`, tipul `T` devine `int`; pentru `maxim(x, y)`, devine `double`.
- În momentul compilării, se generează versiuni specializate (instanțe) ale funcției template pentru fiecare tip concret folosit. Acest proces se numește **instantiere**.

Multiple tipuri template

O funcție template poate avea mai mulți parametri de tip, permițând operarea pe tipuri diferite în același apel.

```
template <typename T1, typename T2>
void afiseazaSuma(T1 a, T2 b)
{
    std::cout << "Suma: " << a + b << std::endl;
}
```

Această funcție acceptă două tipuri diferite și afișează suma lor.

```
int main()
{
    afiseazaSuma(5, 3.2);   // Tipurile: T1 = int, T2 = double
    afiseazaSuma(2.5, 7);   // Tipurile: T1 = double, T2 = int
    return 0;
}
```

Avantaje

Utilizarea funcțiilor template aduce numeroase beneficii în dezvoltarea aplicațiilor, în special prin posibilitatea de reutilizare a codului și prin creșterea flexibilității programelor.

Un prim avantaj important este eliminarea duplicării codului. Prin intermediul unei funcții template, programatorul poate defini o singură dată algoritmul sau logica necesară, iar aceeași implementare poate fi utilizată pentru mai multe tipuri de date. Astfel, nu mai este necesară scrierea mai multor variante ale aceleiași funcții pentru fiecare tip în parte, ceea ce reduce dimensiunea codului și facilitează întreținerea acestuia.

Un alt avantaj este flexibilitatea ridicată în utilizare. Funcțiile template pot lucra atât cu tipuri de date predefinite, precum tipuri numerice sau șiruri de caractere, cât și cu tipuri definite de utilizator, cum ar fi clase sau structuri. Singura condiție este ca tipurile respective să ofere

operațiile necesare utilizate în implementarea funcției. Prin urmare, același algoritm poate fi aplicat într-o varietate mare de situații fără modificări ale codului sursă.

De asemenea, funcțiile template oferă performanță ridicată deoarece mecanismul de generare a codului are loc în timpul compilării. Pentru fiecare tip concret utilizat este creată o versiune specializată a funcției, astfel încât execuția programului nu implică costuri suplimentare specifice unor mecanisme generice bazate pe interpretare sau tipuri dinamice. Astfel, funcțiile template combină avantajele reutilizării codului cu eficiența specifică programării statice.

Limitări

În ciuda avantajelor oferite, funcțiile template prezintă și anumite limitări care trebuie luate în considerare în procesul de dezvoltare.

O primă limitare este reprezentată de complexitatea mesajelor de eroare generate de compilator. Erorile apărute în corpul unei funcții template sunt detectate de obicei doar atunci când funcția este instanțiată pentru un anumit tip de date. În cazul utilizării unor tipuri incompatibile, mesajele de eroare pot deveni foarte lungi și dificil de interpretat, deoarece compilatorul afișează întregul proces de generare al versiunii concrete a funcției.

O altă limitare este reprezentată de necesitatea ca tipurile utilizate să suporte operațiile solicitate de funcția template. De exemplu, dacă o funcție utilizează operatorul de comparație > pentru determinarea valorii maxime dintre două elemente, atunci orice tip pentru care funcția este instanțiată trebuie să definească acest operator. În lipsa operației necesare, compilarea va eșua.

Prin urmare, funcțiile template reprezintă un instrument puternic pentru crearea unui cod general și reutilizabil, însă necesită o proiectare atentă pentru a asigura compatibilitatea tipurilor utilizate și pentru a evita dificultățile de depanare.

4.1.8 (*) Funcții lambda

Funcțiile lambda, cunoscute și sub denumirea de expresii lambda sau *closure*-uri, permit definirea rapidă a unor funcții anonime direct în locul în care sunt necesare. Acestea sunt deosebit de utile pentru a furniza logici personalizate, pe scurt, algoritmilor din biblioteca standard și pentru a încapsula bucăți mici de cod.

Sintaxa generală a unei expresii lambda este:

```
[captura](parametri) -> tip_returnat {
    // corpul functiei lambda
}
```

Componentele unei funcții lambda

- **[captură]**: Specifică cum sunt capturate variabilele din domeniul de vizibilitate (scope-ul) înconjurător. Poate fi:
  - `[&]` - Capturează toate variabilele prin referință.
  - `[=]` - Capturează toate variabilele prin valoare.
  - `[a, &b]` - Capturează variabila `a` prin valoare și variabila `b` prin referință.
  - `[&, x]` - Capturează toate variabilele prin referință, cu excepția lui `x` care este capturat prin valoare.
  - `[=, &y]` - Capturează toate variabilele prin valoare, cu excepția lui `y` care este capturat prin referință.
- **(parametri)**: Lista de parametri formali, similară ca a unei funcții obișnuite. Poate fi goală `()`.
- →**tip_returnat**: Specificarea explicită a tipului de returnare este opțională. Compilatorul îl poate deduce în multe cazuri. Este necesară doar dacă corpul lambda este complex și compilatorul nu poate deduce tipul, sau dacă doriți să forțați un anumit tip.
- **corp**: Blocul de cod care definește acțiunea funcției lambda.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    int factor = 2;
    std::vector<int> numere = {1, 2, 3, 4, 5};

    // Exemplu 1: Lambda simpla fara captura
    auto afiseaza = [](int x) { std::cout << x << " "; };
    std::cout << "Vector initial: ";
    std::for_each(numere.begin(), numere.end(), afiseaza);

    // Exemplu 2: Lambda cu captura prin valoare
    auto inmulteste = [factor](int x) { return x * factor; };
```

```
    std::transform(numere.begin(), numere.end(), numere.begin(),
       inmulteste);

    std::cout << "\nDupa inmultire: ";
    std::for_each(numere.begin(), numere.end(), afiseaza);

    // Exemplu 3: Lambda inline cu parametri, folosita pentru
       sortare
    std::sort(numere.begin(), numere.end(), [](int a, int b) {
        return a > b; // criteriu de sortare descrescatoare
    });

    std::cout << "\nDupa sortare: ";
    std::for_each(numere.begin(), numere.end(), afiseaza);

    // Exemplu 4: Lambda cu tip returnat explicit
    auto aduna = [](double a, double b) -> double {
        return a + b;
    };

    std::cout << "\nRezultat adunare: " << aduna(3.5, 2.1) <<
       std::endl;

    return 0;
}
```

Explicații pentru exemplele de mai sus:

Exemplul 1: Lambda `afiseaza` este definită și stocată într-o variabilă `auto`. Ea nu capturează nimic și primește un parametru `int` pe care îl afișează.

Exemplul 2: Lambda `inmulteste` capturează variabila locală `factor` **prin valoare** (`[factor]`) și o folosește pentru a înmulți fiecare element al vectorului.

Exemplul 3: O expresie lambda **inline** este definită direct ca argument pentru funcția `std::sort`. Ea definește un comparator pentru sortare descrescătoare.

Exemplul 4: Funcţia lambda `aduna` specifică explicit tipul de returnare (`-> double`), deși în acest caz simplu compilatorul l-ar fi putut deduce.

Avantaje

Utilizarea funcțiilor lambda oferă numeroase avantaje în dezvoltarea aplicațiilor moderne, în special prin simplificarea codului și prin posibilitatea de a defini comportamente locale acolo unde sunt necesare.

Un prim avantaj important este reprezentat de obținerea unui cod mai compact și mai expresiv. Funcțiile lambda permit definirea unei logici simple direct în locul unde aceasta este utilizată, fără necesitatea declarării unei funcții separate. Acest lucru reduce numărul de funcții auxiliare și face codul mai ușor de urmărit, deoarece implementarea operației apare exact în contextul în care este folosită.

Un alt beneficiu este posibilitatea de accesare a variabilelor locale prin mecanismul de captură. O funcție lambda poate utiliza variabile definite în zona înconjurătoare fără ca acestea să fie transmise explicit ca parametri. Acest mecanism este util în situațiile în care o operație depinde de anumite valori locale, simplificând transmiterea informațiilor necesare.

Funcțiile lambda sunt deosebit de utile în lucrul cu algoritmii din biblioteca standard STL. Acestea reprezintă o soluție eficientă pentru definirea rapidă a predicatelor, comparatorilor sau a altor operații personalizate utilizate de algoritmi precum `std::sort`, `std::find_if`, `std::for_each` sau `std::transform`. În acest mod, comportamentul algoritmilor poate fi modificat fără a fi necesară crearea unor funcții auxiliare separate.

De asemenea, o expresie lambda este implementată intern sub forma unui obiect funcțional (functor). Aceasta înseamnă că poate fi atribuită unei variabile folosind `auto` sau poate fi stocată în obiecte de tip `std::function`. Astfel, funcțiile lambda pot fi tratate similar altor obiecte din program, oferind flexibilitate suplimentară în organizarea codului.

Dezavantaje

În ciuda avantajelor oferite, funcțiile lambda prezintă și anumite limitări care trebuie luate în considerare atunci când sunt utilizate.

O primă limitare apare în cazul logicilor complexe. Deși funcțiile lambda sunt potrivite pentru operații scurte și locale, o expresie lambda foarte lungă sau care conține numeroase ramificații poate reduce lizibilitatea codului. În astfel de situații, definirea unei funcții clasice cu un nume sugestiv poate fi o soluție mai clară, deoarece permite separarea responsabilităților și documentarea mai ușoară a comportamentului.

O altă dificultate este legată de procesul de depanare (debugging). Deoarece funcțiile lambda sunt funcții anonime, identificarea acestora în timpul depanării poate fi mai dificilă decât în cazul funcțiilor denumite. Urmărirea stivei de apeluri poate deveni mai puțin intuitivă, iar stabilirea punctelor de oprire poate necesita uneori pași suplimentari.

De asemenea, funcțiile lambda au o reutilizabilitate limitată. În general, acestea sunt create pentru o utilizare locală și punctuală. Dacă aceeași logică trebuie folosită în mai multe zone ale programului, definirea unei funcții separate este de obicei o alegere mai potrivită, deoarece favorizează reutilizarea și evită duplicarea codului.

O altă posibilă problemă apare în cazul capturării variabilelor prin referință. În special în C++, utilizarea capturii prin simbolul `&` poate conduce la situații dificile, precum existența unor referințe invalide (dangling references), atunci când variabila capturată își pierde durata de viață înainte ca funcția lambda să fie executată. Din acest motiv, mecanismul de captură trebuie utilizat cu atenție pentru a evita comportamente neașteptate.

Utilizări comune

Funcțiile lambda sunt utilizate frecvent în numeroase situații datorită flexibilității și simplității lor.

Una dintre cele mai întâlnite utilizări este personalizarea algoritmilor din biblioteca STL. Funcțiile lambda permit transmiterea rapidă a unor operații proprii către algoritmi precum `std::sort`, pentru definirea ordinii de sortare, `std::find_if`, pentru căutări bazate pe condiții personalizate, sau `std::transform`, pentru aplicarea unor transformări asupra elementelor unei colecții.

Un alt domeniu de utilizare este reprezentat de definirea callback-urilor rapide. Funcțiile lambda pot fi folosite pentru a specifica acțiuni care trebuie executate ulterior, de exemplu ca răspuns la anumite evenimente sau în cadrul operațiilor asincrone.

De asemenea, acestea sunt potrivite pentru operații temporare și simple. În locul creării unor funcții auxiliare care sunt utilizate o singură dată, o funcție lambda permite menținerea logicii direct în zona unde este necesară, reducând astfel cantitatea de cod suplimentar.

În plus, funcțiile lambda reprezintă o alternativă mai sigură și mai expresivă la utilizarea macro-urilor pentru încapsularea unor operații simple. Spre deosebire de macro-uri, acestea res-

pectă regulile de tipizare ale limbajului și sunt verificate de compilator, reducând riscul apariției unor erori dificil de identificat.

4.1.9 Funcții recursive

Funcțiile recursive sunt funcții care se autoapelează direct sau indirect pentru a rezolva o problemă prin descompunerea acesteia în subprobleme mai mici și de același tip. Această tehnică este deosebit de utilă pentru problemele care admit o definiție recursivă naturală, cum ar fi parcurgerea structurilor de date arborescente sau calculul unor șiruri matematice.

Structura funcțiilor recursive

O funcție recursivă bine proiectată trebuie să conțină întotdeauna două componente fundamentale:

1. **Condiția de oprire (cazul de bază)**: O condiție care verifică dacă problema a fost redusă la un caz suficient de simplu încât să poată fi rezolvat direct, fără a mai fi nevoie de un autoapell. Această condiție este esențială pentru a preveni autoapelurile infinite și, în consecință, depășirea stivei (stack overflow).

2. **Pasul recursiv (cazul recursiv)**: Parte a logicii funcției în care problema curentă este descompusă într-una sau mai multe subprobleme mai mici, asemănătoare problemei originale. Funcția se autoapelează pentru a rezolva această subproblemă, iar rezultatul final este obținut prin combinarea soluțiilor subproblemelor.

```
// Exemplu: Suma primelor n numere naturale
int suma(int n) {
    // 1. Conditia de oprire
    if(n <= 0) {
        return 0;
    }
    // 2. Pasul recursiv: problema (suma pana la n) este
       descompusa in
    // (suma pana la n-1) + n
    return suma(n - 1) + n;
}
```

În exemplul anterior, funcția `suma` se oprește atunci când n devine mai mic sau egal cu 0.

Componente esențiale

Orice funcție recursivă corect implementată trebuie să conțină câteva componente fundamentale care asigură rezolvarea problemei și terminarea execuției.

Prima componentă este **condiția de oprire** (cazul de bază). Aceasta reprezintă situația în care problema este suficient de simplă pentru a putea fi rezolvată direct, fără alte apeluri recursive. Existența unei condiții de oprire este esențială, deoarece împiedică apariția unui număr infinit de apeluri recursive și garantează încheierea execuției funcției.

A doua componentă este **pasul recursiv**. În această etapă, funcția își apelează propria implementare, însă pentru o versiune mai simplă sau mai mică a problemei inițiale. De regulă, dimensiunea problemei este redusă la fiecare apel, de exemplu prin trecerea de la valoarea n la $n - 1$. Astfel, fiecare apel recursiv apropie execuția de cazul de bază.

Ultima componentă este **combinarea rezultatelor**. După ce apelurile recursive se încheie și încep să returneze rezultate, funcția utilizează aceste valori pentru a construi soluția problemei inițiale. De exemplu, în calculul sumei primelor n numere naturale, rezultatul obținut pentru $n - 1$ este combinat cu valoarea curentă n pentru a determina răspunsul final.

Stiva de apeluri recursive

Execuția funcțiilor recursive este gestionată de stiva de apeluri (*call stack*), o zonă de memorie utilizată de sistem pentru administrarea apelurilor de funcții.

La fiecare apel recursiv este creat un nou *cadru de stivă* (*stack frame*), care conține informațiile necesare execuției apelului respectiv. Aceste informații includ parametrii funcției, variabilele locale și adresa la care programul trebuie să revină după terminarea apelului.

Pe măsură ce funcția continuă să se autoapeleze, noi cadre sunt adăugate în stivă, ceea ce conduce la un consum progresiv al memoriei disponibile. Acest proces continuă până când este atinsă condiția de oprire, moment în care nu mai sunt generate apeluri recursive suplimentare.

După atingerea cazului de bază, apelurile recursive încep să se încheie în ordinea inversă în care au fost create, respectând principiul *Last-In-First-Out (LIFO)*. Fiecare apel primește rezultatul apelului următor și îl utilizează pentru a calcula propria valoare, până când se ajunge la primul apel efectuat.

Principalul dezavantaj al utilizării recursivității este riscul apariției unei erori de tip *stack overflow*. Aceasta apare atunci când numărul apelurilor recursive devine prea mare, iar spațiul

de memorie alocat stivei este epuizat înainte de atingerea condiției de oprire. Din acest motiv, funcțiile recursive trebuie proiectate astfel încât fiecare apel să apropie execuția de cazul de bază și să limiteze adâncimea recursivității.

Utilizări practice

- **Calcul factorial**:

```
int factorial(int n) {
    // Conditie de oprire: factorialul lui 0 sau 1 este 1
    return (n <= 1) ? 1 : n * factorial(n-1);
}
```

- **Șirul Fibonacci** (versiune ineficientă fără memoizare, demonstrativă):

```
int fibonacci(int n) {
    if(n <= 1) return n; // Caz de baza: F(0)=0, F(1)=1
    return fibonacci(n-1) + fibonacci(n-2); // Pas recursiv
}
```

Atenție: Această implementare simplă are complexitate exponențială ($O(2^n)$) din cauza recalculării masive a acelorași valori.

- **Parcurgerea arborilor binari (post-order)**:

```
void parcurgeArbore(Nod* nod) {
    if(nod == nullptr) return; // Conditie de oprire: nod nul
    parcurgeArbore(nod->stanga); // Parcurge subarborele stang
    parcurgeArbore(nod->dreapta); // Parcurge subarborele
       drept
    // Proceseaza nodul curent aici
}
```

Optimizări importante

Recursivitate în coadă (Tail Recursion)

Acest tip de recursivitate este un caz special în care apelul recursiv este ultima operație executată în funcție, înainte de returnare. Compilatorul poate optimiza astfel de funcții (*Tail Call Optimization - TCO*) prin rescrierea lor sub formă iterativă, evitând creșterea stivei pentru fiecare apel.

Exemplu de factorial optimizat

```
int factorialTail(int n, int acumulator = 1) {
    if(n <= 1) return acumulator; // Conditie de oprire
    // Apelul recursiv este ultima operatie, cu tot calculul in
       acumulator
    return factorialTail(n-1, n * acumulator);
}
// Apel initial: factorialTail(5, 1) -> factorialTail(4, 5) ->
   factorialTail(3, 20) -> ... -> intoarce 120
```

- **Avantaj**: Transformă recursivitatea în iterație la nivel de cod mașină, economisind memorie stivă.
- **Limitare**: Nu toate compilatoarele sau setările de optimizare efectuează automat această conversie.

Memoizare

Memoizarea este o tehnică de optimizare utilizată în special împreună cu algoritmii recursivi, care constă în **stocarea rezultatelor intermediare** ale subproblemelor deja rezolvate. Aceste rezultate sunt memorate într-o structură de date, precum un *vector*, un *map* sau un *unordered_map*, astfel încât, atunci când aceeași subproblemă apare din nou, rezultatul să poată fi reutilizat fără a mai fi recalculat.

Această tehnică este deosebit de eficientă în cazul problemelor care prezintă subprobleme suprapuse (*overlapping subproblems*), adică aceleași calcule sunt efectuate în mod repetat pe parcursul execuției algoritmului. Prin memorarea rezultatelor, fiecare subproblemă este rezolvată o singură dată, iar apelurile ulterioare utilizează valoarea deja calculată.

Un exemplu clasic este calculul termenului din șirul lui Fibonacci. Implementarea recursivă simplă are o complexitate temporală exponențială, de ordinul $O(2^n)$, deoarece aceleași valori sunt calculate de foarte multe ori. Prin aplicarea memoizării, fiecare termen este determinat o singură dată, iar complexitatea algoritmului se reduce la $O(n)$, menținând în același timp claritatea și simplitatea implementării recursive.

Principalul dezavantaj al memoizării este consumul suplimentar de memorie necesar pentru stocarea rezultatelor intermediare. Cu cât numărul de stări distincte este mai mare, cu atât memoria ocupată de structura utilizată pentru memorare va crește.

De asemenea, memoizarea nu reprezintă întotdeauna cea mai potrivită soluție. În cazul problemelor care au un spațiu foarte mare al stărilor sau pentru care fiecare subproblemă este calculată o singură dată, costul suplimentar al memorării poate depăși beneficiile obținute prin evitarea recalculărilor.

Reguli de utilizare

Atunci când se utilizează recursivitatea și memoizarea, este recomandată respectarea unor reguli care contribuie la obținerea unor implementări corecte și eficiente.

În primul rând, funcția recursivă trebuie să conțină întotdeauna o condiție de oprire clară și corect definită. Aceasta garantează terminarea execuției și previne apariția recursivității infinite.

În situațiile în care se estimează că adâncimea recursivității poate deveni foarte mare, este recomandată analizarea unei variante iterative a algoritmului. O soluție iterativă elimină riscul apariției unei erori de tip *stack overflow* și poate utiliza memoria mai eficient.

Memoizarea trebuie utilizată atunci când analiza problemei evidențiază existența unor subprobleme suprapuse. În astfel de cazuri, memorarea rezultatelor intermediare poate conduce la reduceri semnificative ale timpului de execuție, transformând uneori algoritmi exponențiali în algoritmi cu complexitate liniară sau polinomială.

Nu în ultimul rând, este recomandată testarea funcțiilor recursive folosind valori de intrare cât mai mari. Acest lucru permite verificarea consumului de memorie și identificarea eventualelor probleme legate de depășirea limitei stivei de execuție, contribuind astfel la obținerea unor aplicații mai robuste și mai fiabile.

4.2 Probleme rezolvate

În secțiunea curentă propunem o serie de probleme cu numere.

1. Să se determine reprezentarea în baza b (cu $2 \leq b \leq 16$) a unui număr natural n.

Rezolvare:

Date de intrare: Se dau două numere naturale:

- n - numărul ce trebuie convertit (cu $n \geq 0$)
- b - baza în care se face conversia (cu $2 \leq b \leq 16$)

Date de ieșire: Șirul de caractere reprezentând numărul n în baza b.

Algoritm de rezolvare: Pentru a converti un număr natural n din baza 10 într-o bază b (unde $2 \leq b \leq 16$), aplicăm următoarea metodă:

(a) Împărțim succesiv numărul n la baza b.

(b) La fiecare pas, reținem restul împărțirii. Aceste resturi, citite de la ultimul la primul, formează reprezentarea numărului în baza b.

(c) Dacă baza este mai mare decât 10, pentru resturi mai mari de 9 folosim caracterele: A (pentru 10), B (pentru 11), . . ., F (pentru 15).

Algoritm scris în C++:

```
#include <iostream>
#include <string>

std::string conversieBaza(int numar, int baza) {
    if (numar == 0)
        return "0";
    if (baza < 2 || baza > 16)
        return "0";
    string cifre = "0123456789ABCDEF";
    string rezultat = "";
    while (numar > 0) {
        int rest = numar % baza;
        rezultat = cifre[rest] + rezultat;
        numar /= baza;
    }
    return rezultat;
}
int main() {
    int numar, baza;
    std::cin >> numar >> baza;
    std::cout << conversieBaza(numar, baza) << std::endl;
    return 0;
}
```

Justificarea soluției propuse: Algoritmul urmează metoda standard de conversie a unui număr natural din baza 10 într-o altă bază b prin împărțiri succesive la b și colectarea resturilor. Această metodă este demonstrată matematic și este folosită în mod curent în informatică.

- Pentru $n = 25$ și $b = 2$:
  - $25 \div 2 = 12$ rest 1
  - $12 \div 2 = 6$ rest 0
  - $6 \div 2 = 3$ rest 0
  - $3 \div 2 = 1$ rest 1
  - $1 \div 2 = 0$ rest 1
  - Reprezentare binară: 11001

Analiza complexității:

- *Complexitate temporală:* $O(\log_b n)$, deoarece la fiecare pas împărțim n la b, iar numărul de pași este proporțional cu numărul de cifre în baza b.
- *Complexitate spațială:* $O(\log_b n)$, pentru că stocăm cifrele rezultatului într-un șir de lungime proporțională cu numărul de cifre.

2. Să se afișeze primele n numere pare din șirul Fibonacci.

Rezolvare:
Date de intrare: Se dă un număr natural n ($n \geq 1$) care reprezintă câte numere pare din șirul lui Fibonacci trebuie afișate.
Date de ieșire: Se vor afișa primele n numere pare din șirul lui Fibonacci, separate prin spațiu. Vom considera 1 ca fiind primul număr din șir, nu 0.

Algoritm de rezolvare: Șirul lui Fibonacci este definit astfel:

$$F_0 = 1, \quad F_1 = 1, \quad F_k = F_{k-1} + F_{k-2} \text{ pentru } k \geq 2$$

Pentru a rezolva problema, generăm termenii Fibonacci în ordine, fără să-i stocăm într-un vector, ci doar păstrăm ultimii doi termeni calculați. De fiecare dată când un termen este par, îl afișăm și creștem un contor. Ne oprim când am afișat n numere pare.

Algoritm scris în C++:

```cpp
#include <iostream>

void afiseazaPareFibonacci(int numar) {
    int primulNumar = 0, alDoileaNumar = 1, contorizare = 0;

    while (contorizare < numar) {
        int numarNou = primulNumar + alDoileaNumar;
        if (numarNou % 2 == 0) {
            std::cout << numarNou << " ";
            contorizare++;
        }
        primulNumar = alDoileaNumar;
        alDoileaNumar = numarNou;
    }

    std::cout << std::endl;
}

int main() {
    int numar;
    std::cin >> numar;
    afiseazaPareFibonacci(numar);
    return 0;
}
```

Justificarea soluției propuse: Algoritmul generează corect termenii șirului Fibonacci în ordine. Datorită condiției `numarNou % 2 == 0`, doar termenii pari sunt afișați. Contorul `contorizare` garantează că exact n astfel de termeni sunt afișați.

$$\text{Șir Fibonacci: } 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

$$\text{Termeni pari: } 0, 2, 8, 34, 144, \ldots$$

Programul va afișa: $2, 8, 34, \ldots$ (fără 0, deoarece considerăm că primul termen din șir este 1).

Analiza complexității:

- *Complexitate temporală:* $O(k)$, unde k este numărul de termeni generați până la găsirea celor n termeni pari. Deoarece termenii pari apar aproximativ la fiecare 3 termeni, complexitatea este $O(n)$ în practică.
- *Complexitate spațială:* $O(1)$, deoarece se folosesc doar variabile simple pentru stocarea termenilor Fibonacci și a contorului.

3. Se introduc de la tastatură numere naturale până se introduce valoarea -1. Numerele nu se reține într-un vector. Să se verifice dacă numerele, în ordinea citirii lor, sunt strict crescătoare până la un anumit punct maxim (vârf), iar apoi sunt strict descrescătoare.

Rezolvare:
Date de intrare: Se introduc de la tastatură numere naturale, unul câte unul, până când se introduce valoarea -1 care marchează sfârșitul secvenței.
Date de ieșire: Se va afișa:

- "DA" dacă numerele citite formează o secvență strict crescătoare până la un punct maxim (vârf), după care urmează o secvență strict descrescătoare.
- "NU" în caz contrar.

Algoritm de rezolvare: Algoritmul parcurge secvența element cu element, fără să le rețină pe toate.

(a) Citim primul număr, îl considerăm ca fiind începutul secvenței.

(b) Iterăm citind numere până la `-1`:

- În starea *creștere*, verificăm dacă următorul număr este strict mai mare decât precedentul. Dacă da, continuăm creșterea.
- Dacă următorul număr este mai mic decât precedentul, atunci am găsit vârful, trecem în starea *scădere*.

- Dacă următorul număr este egal cu precedentul sau dacă secvența începe cu descreștere (primul pas în starea scădere), șirul nu respectă condiția și afișăm "NU".
- În starea *scădere*, fiecare număr trebuie să fie strict mai mic decât precedentul.

(c) Dacă la final condițiile sunt respectate (a fost măcar o creștere și o scădere validă), afișăm "DA", altfel "NU".

Algoritm scris în C++:

```
#include <iostream>

bool verificaSecventa() {
    int precedent, curent;
    std::cin >> precedent;

    if (precedent == -1) return false; // secventa vida
    enum Stare { CRESTERE, SCADERE }; // se poate considera
       si un bool
    Stare stare = CRESTERE;
    bool varf_gasit = false;

    while (true) {
        std::cin >> curent;
        if (curent == -1) break;
        if (stare == CRESTERE) {
            if (curent > precedent) {
                // continua cresterea
            } else if (curent < precedent) {
                // trecem la scadere
                stare = SCADERE;
                varf_gasit = true;
            } else {
                // egalitate nu e permisa
                return false;
            }
        } else { // stare == SCADERE
            if (curent < precedent) {
                // continua scaderea
            } else {
                // nu este strict descrescatoare
                return false;
            }
        }
```

```
            precedent = curent;
        }
        // pentru a fi corect, trebuie sa fi existat cel putin o
           crestere si o scadere
        return varf_gasit;
}

int main() {
        if (verificaSecventa())
            std::cout << "DA\n";
        else
            std::cout << "NU\n";
        return 0;
}
```

Justificarea soluției propuse:

- Algoritmul citește elementele secvenței unul câte unul, verificând strict creșterea și apoi scăderea.
- Nu se permit egalități și nu nu acceptă scădere înainte de a exista o creștere.
- Dacă nu apare niciodată o scădere (adică vârful nu este găsit), secvența nu este validă.
- Exemplu valid: 1 3 5 4 2 -1 (crește până la 5, apoi scade)
- Exemplu invalid: 2 2 1 -1 (egalitate nu e permisă)
- Exemplu invalid: 5 4 3 -1 (nu există creștere înainte de scădere)

Analiza complexității:

- *Complexitate temporală:* $O(m)$, unde m este numărul de elemente citite (fără -1).
- *Complexitate spațială:* $O(1)$, deoarece se păstrează doar câteva variabile pentru starea curentă și elementele precedente.

4. Se citește un număr natural n de minim 5 cifre. Să se afișeze suma dintre cele mai mari 3 cifre din număr și cele mai mici 3 cifre din număr.

Rezolvare:

Date de intrare: Se citește un număr natural n, cu cel puțin 5 cifre.

Date de ieșire: Se afișează suma dintre suma celor mai mari 3 cifre din n și suma celor mai mici 3 cifre din n.

Algoritm de rezolvare: Pentru a determina cele mai mari și cele mai mici 3 cifre, fără a folosi sortare și fără vectori, se procedează astfel:

- Inițializăm trei variabile pentru cele mai mari cifre cu valori foarte mici (ex: -1) și trei pentru cele mai mici cifre cu valori mari (ex: 10).
- Parcurgem cifrele numărului una câte una (prin extragerea restului împărțirii la 10).
- La fiecare cifră citită, actualizăm cele 3 maxime și cele 3 minime folosind funcții separate care reține cele mai mari respectiv cele mai mici trei valori.
- La final, calculăm suma celor 3 maxime și suma celor 3 minime și le adunăm.

Algoritm scris în C++

```
#include <iostream>
using namespace std;

// Functie care actualizeaza cele mai mari 3 valori
void actualizeazaMaxime(int val, int &max1, int &max2, int
   &max3) {
    if (val > max1) {
        max3 = max2;
        max2 = max1;
        max1 = val;
    } else if (val > max2) {
        max3 = max2;
        max2 = val;
    } else if (val > max3) {
        max3 = val;
    }
}

// Functie care actualizeaza cele mai mici 3 valori
void actualizeazaMinime(int val, int &min1, int &min2, int
   &min3) {
    if (val < min1) {
        min3 = min2;
        min2 = min1;
        min1 = val;
    }
    else if (val < min2) {
```

```
        min3 = min2;
        min2 = val;
    }
    else if (val < min3) {
        min3 = val;
    }
}

// Functie pentru suma a trei numere
int sumaTrei(int a, int b, int c) {
    return a + b + c;
}

// Functie principala care proceseaza numarul si calculeaza
   suma ceruta
int calculeazaSuma(int n) {
    int max1 = -1, max2 = -1, max3 = -1;
    int min1 = 10, min2 = 10, min3 = 10;
    int contor = 0;
    while (n > 0) {
        int cifra = n % 10;
        actualizeazaMaxime(cifra, max1, max2, max3);
        actualizeazaMinime(cifra, min1, min2, min3);
        n /= 10;
        contor++;
    }
    if(contor < 5)
        return -1; // nu exista minim 5 cifre
    int sumaMaxime = sumaTrei(max1, max2, max3);
    int sumaMinime = sumaTrei(min1, min2, min3);
    return sumaMaxime + sumaMinime;
}

int main() {
    int n;
    std::cin >> n;
    std::cout << calculeazaSuma(n) << "\n";
    return 0;
}
```

Justificarea soluției propuse:

- Funcțiile `actualizeazaMaxime` și `actualizeazaMinime` păstrează corect cele mai mari respectiv cele mai mici 3 cifre pe măsură ce se parcurge numărul.
- Funcția `calculeazaSuma` combină aceste valori corect.

- Exemplu: pentru $n = 5816243$, cifrele sunt {5,8,1,6,2,4,3}
  - Cele mai mari 3 cifre: 8, 6, 5 (sumă = 19)
  - Cele mai mici 3 cifre: 1, 2, 3 (sumă = 6)
  - Rezultatul afișat: 25

Analiza complexității:

- *Complexitate temporală:* $O(d)$, unde d este numărul de cifre din n (parcurgere o singură dată).
- *Complexitate spațială:* $O(1)$, deoarece nu se folosesc vectori și se stochează doar un număr fix de variabile.

4.3 Probleme propuse - Funcții simple

*

1. Se citește un număr natural n. Afișați suma cifrelor lui n.
   Exemplu: Pentru $n = 12345$ se va afișa 15, deoarece 1+2+3+4+5=15.

2. Se citește un număr natural n. Afișați produsul cifrelor de pe poziții impare ale lui n. Prima poziție din număr este considerată 0.
   Exemplu: Pentru $n = 123457$ se va afișa 42, deoarece 2*4*7=42.

3. Se citește un număr natural n. Să se verifice dacă numărul n este prim.
   Exemplu: $n = 211$ este prim, iar $n = 213$ nu este prim.

4. Se citește un număr natural n. Să se verifice dacă numărul n este palindrom. Un număr este palindrom dacă are aceeași valoare atât dacă e citit de la stânga la dreapta, cât și de la dreapta la stânga.
   Exemplu: $n = 12321$ este palindrom, iar $n = 12345$ nu este palindrom.

5. Se citește un număr natural n. Afișași numărul natural m egal cu valoarea răsturnatului lui n.
   Exemplu: Pentru $n = 12345$ se va afișa $m = 54321$

6. Se citesc două numere naturale a și b. Să se calculeze cel mai mare divizor comun (CMMDC) al lor.
   Exemplu: Pentru $a = 144$ și $b = 81$ se obține $CMMDC = 9$.

7. Se citesc două numere naturale a și b. Să se determine câte cifre aflate pe aceeași poziție / index din ambele numere sunt identice.
   Exemplu: Pentru $a = 16547$ și $b = 12345$ cifrele de pe poziția 0 și 3 sunt identice (1 și 4), deci 2 cifre.

**

1. Se citește un număr natural n. Afișați toate perechile de numere de la 1 la n care au proprietatea că au aceeași suma a cifrelor.
   Exemplu: Pentru $n = 15$ se vor afișa perechile: (1 10), (2 11), (3 12), (4 13), (5 14), (6 15).

2. Se citesc două numere naturale a și b, cu $a \leq b$. Aflați câte numere palindrom sunt în intervalul $[a, b]$.
   Exemplu: Pentru $a = 100$ și $b = 200$ se vor afișa: 101, 111, 121, 131, 141, 151, 161, 171, 181, 191.

3. Se citește de la tastatură un număr natural n. Apoi se citesc n numere naturale. Afișați câte dintre numerele citite au răsturnatul egal cu primul număr citit.
   Exemplu: Pentru $n = 7$ și numerele 231 78 132 30 132 8 132 se va afișa: 3.

4. Se citește un număr natural n și un sir de n perechi de numere. Determinați și afișați perechea pentru care valoarea CMMDC-ului dintre cele doua valori este maximă.
   Exemplu: Pentru $n = 3$ și perechile (12, 10), (45, 35), (23, 74) se va afișa perechea: (45, 35) pentru ca (12, 10) are CMMDC 2, (45, 35) are CMMDC 5 și (23, 74) are CMMDC 1.

5. Se citesc un număr natural n și o cifră c. Eliminați toate aparițiile cifrei c din numărul n și afișați numărul astfel obținut.
   Exemplu: Pentru $n = 2345324$ și $c = 2$ rezultă numărul 34534.

6. Se citește un număr natural n. Dublați toate cifrele din numărul n și afișați numărul astfel obținut.
   Exemplu: Pentru $n = 1234$ rezultă numărul 11223344.

7. Se citește un număr natural n. Determinați dacă este un pătrat perfect sau un cub perfect.
   Exemplu: n = 64 = 8^2 este pătrat perfect, iar n = 125 = 5^3 este cub perfect.

***

1. Se citește un număr natural k din intervalul $[1, 9]$. Afișați toate numerele n formate din exact k cifre care au proprietatea ca $n - 1$ și $n + 1$ sunt numere prime.
   Exemplu: Pentru $k = 2$ se vor afișa numerele: 12, 18, 30, 42, 60 și 72.

2. Se citește un număr natural k. Găsiți toate numerele cu k cifre unde suma cifrelor este un număr prim și produsul cifrelor este un pătrat perfect.
   Exemplu: Pentru $k = 2$ numerele cerute sunt: 14, 41, 49, 94.

3. Se citește un număr natural n. Să se afișeze cel mai mic număr care se poate forma cu cifrele lui n, fiecare cifră repetându-se de câte ori se repetă și în n.
   Exemplu: Pentru $n = 1204192$ se va afișa: 112249 (se consideră că ar fi un 0 în față, dar e nesemnificativ).

4. Se citește un număr natural n. Să se verifice dacă cifrele lui n sunt în progresie aritmetică.
   Exemplu: Pentru $n = 2596$ cifrele nu sunt în progresie aritmetică, iar pentru $n = 2468$ cifrele sunt în progresie aritmetica cu rația 2.

5. Se citește un număr natural n care are un număr par de cifre. Să se determine și să se afișeze numărul obținut din n prin interschimbarea cifrelor din fiecare pereche de cifre alăturate, perechile fiind formate începând de la ultima cifră: cifra unităților cu cea a zecilor, cea a sutelor cu cea a miilor etc. Numărul rezultat se afișează fără zerouri nesemnificative la
   Exemplu: Pentru $n = 314620$ rezultă 136402.

6. Se citește un număr natural n. Afișați cel mai mare număr care se poate obține din n prin eliminarea unei singure cifre.

Exemplu: Pentru $n = 45329$ se obține 5329, pentru că este ce cel mai mare dintre numerele 4532, 4539, 4529, 4329 și 5329.

7. Se citește un număr natural n. Eliminați toate aparițiile cifrei maxime, dublați toate aparițiile cifrei minime din numărul n și afișați numărul astfel obținut.
   Exemplu: Pentru $n = 6234622$ rezultă numărul 223462222.

****

1. Se consideră două numere naturale. Verificați dacă unul dintre numere se poate obține prin rotirea cifrelor celuilalt (permutări circulare).
   Exemplu: 123 și 312 sunt permutări circulare.

2. Verificați dacă un număr natural n este un număr Armstrong (suma cifrelor ridicate la puterea numărului de cifre este egală cu numărul inițial).
   Exemplu: $n = 153$ este Armstrong, deoarece are 3 cifre și dacă ridicăm fiecare cifră la puterea 3 și le adunăm obținem numărul inițial n: $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$.

3. Găsiți cel mai lung subșir de cifre consecutive într-un număr și lungimea acestuia.
   Exemplu: În 78123419, subșirul 1234 are toate cifrele consecutive și are lungimea 4.

4. Verificați dacă un număr are toți factorii primi distincți (diferiți / unici).
   Exemplu: 30 are factorii 2, 3, 5 care sunt unici, dar 12 are factorii 2, 2, 3 care nu sunt unici.

5. Verificați dacă trei numere naturale pot reprezenta un triunghi și tipul acestuia în caz afirmativ.
   Exemplu: $a = 6$, $b = 8$, $c = 10$ reprezintă un triunghi dreptunghic, iar $a = 1$, $b = 1$, $c = 3$ nu poate fi triunghi.

6. Se citește un număr natural n. Eliminați toate aparițiile grupurilor de cifre identice din numărul n și afișați numărul astfel obținut.
   Exemplu: Pentru $n = 1223444122$ rezultă numărul 131.

7. Determinați reprezentarea în bază $2 \leq b \leq 9$ a unui număr natural n.
   Exemplu: Numărul $n = 678$ este scris în baza $b = 4$ ca fiind 22212.

*****

1. Un număr este super-prim dacă este prim și, în plus, toate prefixele sale (adică toate numerele formate din primele cifre ale sale, de la stânga la dreapta) sunt prime. Se citește un număr natural n. Să se verifice dacă n este super-prim.
   Exemplu: Numărul $n = 233$ este super-prim, deoarece atât el, cât și toate prefixele sale sunt prime: 2, 23, 233.
   Exemplu: Numărul $n = 2347$ nu este super-prim, deoarece deși 2347 este prim, prefixul 234 nu este prim.

2. Definiția 1: Cifra de control a unui număr este obținută prin adunarea repetată a cifrelor sale până când se obține o singură cifră.
   Exemplu: Pentru numărul 999: 9+9+9=27, 2+7=9, deci avem cifra de control 9.
   Definiția 2: Un număr este prodo-divizibil dacă se împarte exact la produsul cifrelor sale (fără rest).
   Exemplu: Pentru numărul 135: $1 * 3 * 5 = 15 \rightarrow 135/15 = 9$ (cu restul 0), deci este prodo-divizibil.
   Se citește un număr natural n.

   (a) Să se calculeze cifra de control a lui $n!$ (factorialul lui n).
   Exemplu: Pentru $n = 9$, avem $9! = 1 * 2 * 3 * \ldots * 9 = 362880$, deci avem $3 + 6 + 2 + 8 + 8 + 0 = 27, 2 + 7 = 9$

   (b) Se citește o listă de n numere naturale. Să se afișeze cifra de control doar pentru numerele prodo-divizibile din listă.
   Exemplu: Pentru $n = 4$ și lista $[135, 101, 22, 18]$, doar 135 este prodo-divizibil și se afișează un singur 9 pentru cifra de control a acestuia.

3. Un număr este mega-cub dacă este cub perfect și suma cifrelor este rădăcina cubă a lui.
   Exemplu: 512 este mega-cub deoarece suma cifrelor: 5 + 1 + 2 = 8 și 8^3 = 512.
   Se citesc două numere naturale a și b, cu $a \leq b$. Să se afișeze toate numerele mega-cub din intervalul $[a, b]$, care au proprietatea suplimentară că au cel puțin o permutare ciclică (diferită de numărul inițial) care este cub perfect.

Exemplu: Mega-cubul 512 are două permutări ciclice: 125 și 251, unde 125 este cub perfect: 5^3 = 125.

4. Un număr natural n se numește rotund dacă reprezentarea sa binară conține un număr egal de 0-uri și 1-uri. Se citesc două numere naturale a și b, cu $a \leq b$. Să se afișeze toate numerele din intervalul $[a, b]$ care simultan au forma $2^k + 1, 1 \leq k$, sunt rotunde și au reprezentarea binară un număr palindrom.
   Exemplu: $n = 9$ are forma $2^3 + 1$ și reprezentarea binară 1001 care este un număr palindrom și rotund (cu 2 de 0 și 2 de 1), deci respectă simultan toate cele trei proprietăți.

5. Un număr este autobiografic dacă fiecare cifră de pe poziția i descrie numărul de apariții ale lui i în număr. Se citește un număr natural n. Să se verifice dacă n este autobiografic.
   Exemplu: Numărul 1210 este autobiografic pentru că:
   Pe poziția 0 este cifra 1 $\rightarrow$ în număr apare o singură dată cifra 0.
   Pe poziția 1 este cifra 2 $\rightarrow$ în număr apar două cifre 1.
   Pe poziția 2 este cifra 1 $\rightarrow$ în număr apare o singură dată cifra 2.
   Pe poziția 3 este cifra 0 $\rightarrow$ în număr nu apare cifra 3.

4.4 Probleme propuse - Funcții recursive

*

1. Să se scrie o funcție recursivă cu un parametru n care să returneze câte cifre impare are n, n număr natural și nenul.
   Exemplu: Pentru $n = 13334$ se va afișa 4.

2. Să se calculeze recursiv factorialul unui număr n natural și pozitiv.
   Exemplu: Pentru $n = 5$ se va afișa 120.

3. Să se calculeze CMMDC-ul a două numere naturale folosind o funcție recursivă.
   Exemplu: Pentru $a = 12$ și $b = 44$ se obține 4.

4. Să se calculeze recursiv suma primelor n pătrate perfecte.
   Exemplu: Pentru $n = 5$ se obține $1 + 4 + 9 + 16 + 25 = 55$.

5. Să se calculeze recursiv produsul cifrelor pare dintr-un număr natural n.
   Exemplu: Pentru $n = 14452$ se obține $4 * 4 * 2 = 32$.

**

1. Să se scrie o funcție recursivă care calculează și returnează diferența dintre suma elementelor de pe poziții pare și cea a celor aflate pe poziții impare dintr-un vector.
   Exemplu: Pentru vectorul $[1, 4, 7, 8, 2, 0, 6]$ se obține $(1 + 7 + 2 + 6) - (2 + 8 + 0) = 16 - 10 = 6$.

2. Să se scrie o funcție recursivă care primește ca parametru un număr natural și returnează numărul obținut din n prin eliminarea cifrelor impare.
   Exemplu: Pentru $n = 813678$ se obține 868.

3. Să se scrie o funcție recursivă care primește un parametru n număr natural și returnează numărul obținut din n prin scăderea cu 1 a cifrelor impare și mărirea cu 1 a celor pare.
   Exemplu: Pentru $n = 2324$ se obține 3235.

4. Să se scrie o funcție recursivă care primește ca parametru un sir de caractere format din cel mult 100 de caractere litere mici și elimina toate vocalele din sir.
   Exemplu: Pentru s = "exemplu de sir" se obține "xmpl d sr".

5. Să se scrie o funcție recursivă care primește trei parametri: un număr natural n și două cifre $c1$ și $c2$. Funcția returnează numărul obținut din n prin înlocuirea tuturor aparițiilor cifrei $c1$ cu cifra $c2$. Rezultatul este un număr natural, deci nu are zerouri nesemnificative la început.
   Exemple: Pentru $n = 11321$, $c1 = 1$, $c2 = 7$ rezultă 77327. Pentru $n = 121$, $c1 = 1$, $c2 = 0$ rezultă 20.

***

1. Să se scrie o funcție recursivă care primește ca parametru un număr natural n și afișează cifrele sale în ordinea din elemente, separate de ":". După ultima cifra nu mai apare niciun separator!
   Exemplu: Pentru $n = 6432$ se va afișa $6 : 4 : 3 : 2$.

2. Se citește un număr natural n. Verificați dacă pătratul lui n se termină cu n, prin recursivitate.
   Exemplu: Pătratul lui $n = 25$ este 625, care se termină în 25, adică n.

3. Definim $f(1) = 1, f(2) = 2, f(n) = n * f(n-1) - f(n-2)$. Scrieți o funcție recursivă care calculează $f(n)$ și analizează creșterea acestei funcții.

4. Se citește un număr natural n. Să se afișeze folosind funcții recursive următorul desen ca în exemplul:
   1
   1 2
   1 2 3
   1 2 3 4
   .....
   1 2 3 4 ... n

5. Se citește un număr natural n. Să se afișeze folosind funcții recursive următorul desen ca în exemplul pentru $n = 5$:
   *
   ***
   *****
   ***
   *

****

Se citește un număr natural n. Următoarele cerințe folosesc proprietatea de prim și divizori.

1. Găsiți recursiv toți factorii primi ai lui n și aflați P = valoarea produsului lor.
   Exemplu: $n = 175$ are factorii primi 5 și 7, care au $P = 35$.

2. Calculați câte cifre prime are n. Apoi, determinați Z = numărul de cifre prime izolate (nu se află în grupuri cu alte cifre prime ca și vecini).
   Exemplu: $n = 675342$ are 4 cifre prime: 7, 5, 3 și 2. Dintre acestea, doar 2 este izolat.

3. Găsiți recursiv toate combinațiile de factori primi distincți care dau n prin produs.
   Exemplu: $n = 21$ se obține din $3 * 7$ sau $7 * 3$.

4. Calculați $S = d_1 - d_2 + d_3 - d_4 + \ldots$ pentru divizorii pozitivi ai lui n, recursiv.
   Exemplu: Pentru $n = 2025$ avem lista de divizori 1, 3, 5, 9, 15, 25, 27, 45, 75, 81, 135, 225, 405, 675 și 2025, deci calculăm $1 - 3 + 5 - 9 + \ldots$ și obținem $S = 1625$.

5. Calculați recursiv CMMDC-ul valorilor P, Z și S pentru n (calculate conform cerințelor anterioare).

Partea II

Structuri de date fundamentale și algoritmi simpli

Capitolul 5

Tablouri unidimensionale

5.1 Aspecte teoretice

Tablourile unidimensionale constituie una dintre cele mai simple și mai importante structuri de date din limbajul C++. Ele oferă o modalitate eficientă de a stoca un set fix de valori de același tip, aranjate secvențial în memorie, permițând accesul rapid la fiecare element prin intermediul unui index numeric. Datorită acestei organizări, tablourile sunt fundamentale pentru reprezentarea colecțiilor de date omogene și pentru implementarea unor algoritmi de prelucrare sistematică.

Ce este un tablou unidimensional?

Un **tablou unidimensional** este o colecție liniară de elemente de același tip, accesate printr-un index numeric. Fiecare element are o poziție unică în tablou, numerotată de la 0 la $n - 1$, unde n este dimensiunea tabloului.

Exemplu conceptual:

```
int valori[5];
```

Această instrucțiune definește un tablou de 5 elemente de tip `int`, fiecare având o poziție:

```
[0]   [1]   [2]   [3]   [4]
```

Caracteristici ale tablourilor unidimensionale

Tablourile unidimensionale prezintă o serie de caracteristici care le fac deosebit de utile:

- **Tip omogen:** toate elementele trebuie să fie de același tip, ceea ce asigură consistență și predictibilitate în prelucrarea datelor.
- **Indexare zero-based:** primul element are indicele 0, iar ultimul indicele $n - 1$, unde n este numărul total de elemente.

- **Acces direct:** orice element poate fi accesat imediat prin intermediul operatorului `[]`, fără a parcurge întreaga structură.
- **Stocare contiguă în memorie:** elementele sunt așezate unul după altul, ceea ce permite un acces eficient și optimizări din partea compilatorului.

Utilitate și aplicabilitate

Tablourile sunt folosite în numeroase situații practice, datorită simplității și eficienței lor. Ele pot fi utilizate pentru:

- Stocarea și procesarea unor serii de valori numerice, cum ar fi notele studenților, vârstele participanților sau temperaturile măsurate într-o zi.
- Implementarea structurilor de date mai complexe, cum ar fi listele, cozile sau stivele.
- Manipularea datelor repetitive în algoritmi care implică iterații, sortări sau căutări.

Astfel, tablourile oferă un echilibru între simplitatea conceptuală și performanța practică, fiind adesea punctul de plecare pentru învățarea structurilor de date dinamice și a pointerilor.

Tipuri de alocare pentru tablouri

În funcție de modul de gestionare a memoriei, tablourile pot fi create în două moduri principale:

1. **Alocare statică** - dimensiunea tabloului este stabilită la compilare și rămâne neschimbată pe durata execuției.
2. **Alocare dinamică** - dimensiunea este determinată în timpul rulării, permițând o flexibilitate mai mare în funcție de necesitățile programului.

Alocarea statică este potrivită pentru situațiile în care numărul de elemente este cunoscut dinainte, în timp ce alocarea dinamică este utilă atunci când dimensiunea tabloului depinde de datele introduse de utilizator sau de rezultatele unor calcule. În ambele cazuri, accesul la elemente se face în același mod, folosind operatorul `[]`.

În paginile următoare vom explora în detaliu ambele variante, cu accent pe operațiile fundamentale care pot fi efectuate asupra unui tablou: citire, afișare, inserare și ștergere de elemente.

Alocarea statică a tablourilor unidimensionale

Alocarea statică a tablourilor unidimensionale presupune rezervarea unui spațiu de memorie fix în timpul compilării. Dimensiunea tabloului este specificată explicit și nu poate fi modificată în timpul execuției programului. Pentru mai multe detalii despre alocarea statică se recomandă consultarea secțiunii de alocare statică a memoriei.

Sintaxă generală

```
tip_de_data nume_tablou[dimensiune];
```

- `tip_de_data` - tipul elementelor (ex: `int`, `double`, `char`).
- `nume_tablou` - identificatorul tabloului.
- `dimensiune` - valoare constantă cunoscută la compilare.

Exemplu simplu

```
int numere[5] = {2, 5, 6, 1, 0};
```

Funcție de citire a valorilor dintr-un tablou static

```
void citireTablou(int tablou[], int dimensiune) {
    for (int i = 0; i < dimensiune; i++) {
        std::cout << "Elementul " << i << ": ";
        std::cin >> tablou[i];
    }
}
```

Funcție de afișare a unui tablou static

```
void afisareTablou(const int tablou[], int dimensiune) {
    std::cout << "Elementele tabloului sunt: ";
    for (int i = 0; i < dimensiune; i++) {
        std::cout << tablou[i] << " ";
    }
    std::cout << std::endl;
}
```

Funcție de inserare a unui element într-un tablou static

Inserarea într-un tablou static presupune deplasarea elementelor către dreapta pentru a face loc elementului nou.

Atenție: Într-un tablou static, trebuie să ne asigurăm că nu depășim capacitatea maximă.

```
bool inserareElement(int tablou[], int &dim, int capacitate, int
   valoare, int pozitie) {
```

```
    if (dim >= capacitate || pozitie < 0 || pozitie > dim)
        return false;

    for (int i = dim; i > pozitie; --i)
        tablou[i] = tablou[i - 1];

    tablou[pozitie] = valoare;
    dim++;
    return true;
}
```

Apeluri posibile:

- Inserare la început: `inserareElement(tablou, dim, 100, 10, 0);`
- Inserare la mijloc: `inserareElement(tablou, dim, 100, 20, dim/2);`
- Inserare la final: `inserareElement(tablou, dim, 100, 30, dim);`

Funcție de ștergere a unui element dintr-un tablou static

Ștergerea presupune mutarea elementelor către stânga de la poziția de șters.

```
bool stergereElement(int tablou[], int &dim, int pozitie) {
    if (dim <= 0 || pozitie < 0 || pozitie >= dim)
        return false;

    for (int i = pozitie; i < dim - 1; ++i)
        tablou[i] = tablou[i + 1];

    dim--;
    return true;
}
```

Apeluri posibile:

- Ștergere de la început: `stergereElement(tablou, dim, 0);`
- Ștergere de la mijloc: `stergereElement(tablou, dim, dim/2);`
- Ștergere de la final: `stergereElement(tablou, dim, dim-1);`

Exemplu complet cu citire, inserare și ștergere

```
int main() {
    const int CAPACITATE = 100;
    int tablou[CAPACITATE];
    int dim = 0;
```

```
    std::cout << "Cate elemente? ";
    std::cin >> dim;
    citireTablou(tablou, dim);

    afisareTablou(tablou, dim);

    inserareElement(tablou, dim, CAPACITATE, 99, 0); // la inceput
    inserareElement(tablou, dim, CAPACITATE, 77, dim / 2); // la
       mijloc
    inserareElement(tablou, dim, CAPACITATE, 55, dim); // la final

    std::cout << "Dupa inserari:\n";
    afisareTablou(tablou, dim);

    stergereElement(tablou, dim, 0);
    stergereElement(tablou, dim, dim / 2);
    stergereElement(tablou, dim, dim - 1);

    std::cout << "Dupa stergeri:\n";
    afisareTablou(tablou, dim);
    return 0;
}
```

Avantaje ale tablourilor statice

Tablourile statice oferă o serie de beneficii importante în situațiile în care dimensiunea datelor este cunoscută încă din faza de proiectare a programului. În primul rând, acestea sunt extrem de eficiente din punct de vedere al utilizării memoriei. Deoarece spațiul este alocat la momentul compilării, accesul la elemente se face direct, fără costuri suplimentare de alocare dinamică, ceea ce conduce la o viteză crescută de execuție.

De asemenea, performanța generală este îmbunătățită datorită faptului că adresele elementelor sunt fixe și cunoscute încă de la compilare, permițând compilatorului să realizeze optimizări la nivel de cod mașină. Structura statică oferă și un grad ridicat de siguranță, întrucât dimensiunea constantă reduce riscul erorilor de tip depășire a limitelor, atâta timp cât se respectă indexarea corectă.

Pe lângă aceste aspecte, tablourile statice se remarcă prin simplitatea lor: nu necesită operații de gestionare manuală a memoriei, cum ar fi `new` sau `delete`, iar declararea și utilizarea lor sunt intuitive. În general, ele reprezintă alegerea potrivită pentru aplicațiile care manipulează un număr fix și cunoscut de elemente, cum ar fi prelucrarea unui set constant de date.

Dezavantaje ale tablourilor statice

Totuși, limitările tablourilor statice devin evidente atunci când volumul de date este variabil sau necunoscut. Dimensiunea fixă reprezintă principala restricție, deoarece aceasta nu poate fi modificată în timpul execuției. În consecință, dacă se alocă un tablou prea mic, programul poate rămâne fără spațiu de stocare, iar dacă se alocă un tablou prea mare, o parte din memorie rămâne neutilizată.

Această lipsă de flexibilitate face ca tablourile statice să fie mai puțin potrivite pentru aplicații care trebuie să se adapteze la date dinamice, precum fișiere cu lungime variabilă sau structuri care cresc în timp. În plus, utilizarea ineficientă a resurselor devine o problemă în programele complexe, în care memoria trebuie gestionată atent. Un alt dezavantaj major este imposibilitatea redimensionării: odată declarat, un tablou static nu poate fi extins sau micșorat fără recompilarea programului.

Prin urmare, tablourile statice sunt potrivite pentru situațiile stabile și previzibile, dar devin limitative în aplicațiile care cer adaptabilitate și scalabilitate.

Avantaje ale tablourilor dinamice

Tablourile dinamice compensează aceste limitări printr-o flexibilitate considerabil mai mare. Ele permit stabilirea dimensiunii în timpul rulării, în funcție de necesitățile reale ale aplicației. Acest comportament dinamic face posibilă scrierea unor programe generalizate, care pot prelucra cantități variabile de date fără a necesita modificări ale codului sursă.

Din perspectiva utilizării memoriei, tablourile dinamice sunt eficiente, întrucât alocă exact spațiul necesar pentru datele existente, evitând risipa asociată cu structurile statice. Mai mult, prin realocare și copiere, aceste tablouri pot fi extinse sau restrânse în funcție de evoluția datelor procesate, devenind astfel ideale pentru aplicații care lucrează cu intrări imprevizibile.

Un alt avantaj esențial este controlul explicit asupra duratei de viață a memoriei. Programatorul poate decide momentul exact în care spațiul este alocat și eliberat, ceea ce permite optimizarea resurselor în funcție de contextul programului. Datorită acestor caracteristici, tablourile dinamice sunt esențiale în dezvoltarea aplicațiilor moderne, care necesită adaptabilitate și eficiență în gestionarea datelor.

Observație: În cadrul bibliotecii standard C++, există o alternativă modernă la tablourile statice, și anume clasa `std::array`. Aceasta oferă avantajele siguranței și integrării cu ecosistemul STL, păstrând în același timp simplitatea și eficiența tablourilor tradiționale. Detalii suplimentare despre această clasă pot fi consultate în secțiunea dedicată containerelor secvențiale (STL).

Alocarea dinamică a tablourilor unidimensionale (cu pointeri)

Alocarea dinamică a tablourilor unidimensionale presupune rezervarea unui spațiu de memorie în timpul execuției programului, iar dimensiunea tabloului este determinată la rulare și poate fi controlată de programator. Pentru mai multe detalii despre alocarea dinamică se recomandă consultarea secțiunii de alocare dinamică a memoriei.

Sintaxă generală

```
tip_de_data* nume_tablou = new tip_de_data[dimensiune]; // alocare

delete[] nume_tablou;                                    //
   eliberare
```

- `tip_de_data` - tipul elementelor (ex: `int`, `double`, `char`).
- `nume_tablou` - identificatorul tabloului.
- `dimensiune` - valoare variabilă cunoscută la rulare.

Exemplu simplu

```
int n;
std::cout << "Dimensiunea tabloului: ";
std::cin >> n;
int* arr = new int[n]; // alocare

// ... folosirea tabloului ...

delete[] arr; // eliberare
```

Funcție de citire a valorilor dintr-un tablou dinamic

```
void citireTablou(int*& tablou, int dimensiune){
    for (int i = 0; i < dimensiune; i++) {
        std::cout << "Elementul " << i << ": ";
        std::cin >> tablou[i];
    }
}
```

Funcție de afișare a unui tablou dinamic

```
void afisareTablou(const int*& tablou, int dimensiune)
{
    std::cout << "Elementele tabloului: ";
    for (int i = 0; i < dimensiune; i++) {
        std::cout << tablou[i] << " ";
    }
    std::cout << std::endl;
}
```

Funcție de inserare a unui element într-un tablou dinamic

Inserarea într-un tablou dinamic presupune deplasarea elementelor către dreapta pentru a face loc elementului nou.

Atenție:

- Într-un tablou dinamic, trebuie să ne asigurăm că nu depășim capacitatea maximă.
- Se creează un tablou nou, iar cel vechi trebuie eliberat din memorie.

```
int* inserareElement(int*& tablou, int &dim, int pozitie, int
   valoare){
    if (pozitie < 0 || pozitie > dim) return tablou;

    int* nouTablou = new int[dim + 1];

    for (int i = 0; i < pozitie; i++)
        nouTablou[i] = tablou[i];

    nouTablou[pozitie] = valoare;

    for (int i = pozitie; i < dim; i++)
        nouTablou[i + 1] = tablou[i];

    delete[] tablou;
    dim++;

    return nouTablou;
}
```

Apeluri:

- Inserare la început: `arr = inserareElement(arr, dim, 0, 10);`
- Inserare la mijloc: `arr = inserareElement(arr, dim, dim/2, 20);`
- Inserare la final: `arr = inserareElement(arr, dim, dim, 30);`

Funcție de ștergere a unui element dintr-un tablou static

Ștergerea presupune mutarea elementelor către stânga de la poziția de șters.

Atenție: Se creează un tablou nou, iar cel vechi trebuie eliberat din memorie.

```
int* stergereElement(int* tablou, int &dim, int pozitie)
{
    if (dim == 0 || pozitie < 0 || pozitie >= dim) return tablou;

    int* nouTablou = new int[dim - 1];

    for (int i = 0; i < pozitie; i++)
        nouTablou[i] = tablou[i];

    for (int i = pozitie + 1; i < dim; i++)
        nouTablou[i - 1] = tablou[i];

    delete[] tablou;
    dim--;
    return nouTablou;
}
```

Apeluri:

- Ștergere de la început: `arr = stergereElement(arr, dim, 0);`
- Ștergere de la mijloc: `arr = stergereElement(arr, dim, dim/2);`
- Ștergere de la final: `arr = stergereElement(arr, dim, dim-1);`

Exemplu complet cu citire, inserare și ștergere

```
int main() {
    int dim;
    std::cout << "Dimensiunea initiala: ";
    std::cin >> dim;

    int* arr = new int[dim];
    citireTablou(arr, dim);
    afisareTablou(arr, dim);

    arr = inserareElement(arr, dim, 0, 111);        // inceput
    arr = inserareElement(arr, dim, dim/2, 222);    // mijloc
    arr = inserareElement(arr, dim, dim, 333);      // final

    std::cout << "Dupa inserari:\n";
    afisareTablou(arr, dim);

    arr = stergereElement(arr, dim, 0);             // inceput
```

```
    arr = stergereElement(arr, dim, dim/2);          // mijloc
    arr = stergereElement(arr, dim, dim-1);          // final

    std::cout << "Dupa stergeri:\n";
    afisareTablou(arr, dim);

    delete[] arr;
    return 0;
}
```

Dezavantaje ale tablourilor dinamice

Deși oferă o flexibilitate ridicată, tablourile dinamice vin și cu o serie de dezavantaje importante care trebuie avute în vedere atunci când sunt utilizate. Principalul aspect este complexitatea crescută a gestionării memoriei. Programatorul trebuie să aibă o înțelegere solidă a modului în care funcționează pointerii și ciclul de viață al datelor, deoarece alocarea și eliberarea memoriei sunt procese efectuate manual, cu ajutorul operatorilor `new` și `delete[]`.

Această responsabilitate sporită aduce și riscuri, cum ar fi scurgerile de memorie (*memory leaks*) care apar atunci când memoria alocată nu este eliberată corect. În programele complexe, astfel de scurgeri pot duce la creșterea necontrolată a consumului de resurse și, în cele din urmă, la blocarea aplicației. Totodată, operațiile de redimensionare a tablourilor dinamice presupun realocarea spațiului și copierea datelor existente, ceea ce implică un cost suplimentar de performanță, mai ales pentru structurile de mari dimensiuni.

Un alt risc important îl reprezintă erorile de acces invalid. Utilizarea unui pointer neinițializat sau eliberarea multiplă a aceleiași zone de memorie pot genera comportamente imprevizibile, de la blocări ale programului până la coruperea datelor. Aceste probleme sunt adesea dificil de diagnosticat, deoarece nu se manifestă imediat, ci în momente imprevizibile pe parcursul execuției. Din acest motiv, depanarea codului care folosește tablouri dinamice este, în general, mai dificilă decât în cazul tablourilor statice, unde gestionarea memoriei este realizată automat de compilator.

Observație: Pentru a evita aceste dificultăți, limbajul C++ oferă alternative moderne și sigure în cadrul bibliotecii standard. O soluție recomandată este utilizarea clasei `std::vector`, care gestionează automat memoria și oferă funcționalități suplimentare, cum ar fi redimensionarea dinamică și verificarea limitelor. Detaliile referitoare la această clasă pot fi consultate în secțiunea dedicată containerelor secvențiale (STL).

5.2 Probleme rezolvate

În secțiunea care urmează propunem o serie de probleme cu tablouri unidimensionale.

Notă: Aceste probleme vor fi rezolvate atât cu alocare statică de memorie, cât și cu alocare dinamică de memorie, respectiv atât simplu, cât și cu STL.

1. Se citește un număr natural n ($n \leq 100$), apoi un șir de n numere întregi. Să se determine dacă elementele tabloului sunt simetrice față de centru (palindromice), adică $a_0 = a_{n-1}$, $a_1 = a_n$, etc.

   Rezolvare:
   Date de intrare: Un număr natural n $(1 \leq n \leq 100)$, reprezentând numărul de elemente ale tabloului, urmat de n numere întregi, stocate într-un tablou unidimensional.
   Date de ieșire: Se va afișa mesajul "DA" dacă tabloul este simetric față de centru (palindromic) și "NU" în caz contrar.

   Algoritm de rezolvare:

   - Se citește valoarea lui n și apoi se citesc cele n elemente ale tabloului.
   - Se compară elementele din capete spre centru:
     - $a[0]$ cu $a[n-1]$
     - $a[1]$ cu $a[n-2]$
     - ...
     - până când cele două poziții se întâlnesc sau se depășesc.
   - Dacă toate perechile de elemente sunt egale, tabloul este palindromic $\Rightarrow$ se afișează "DA".
   - În caz contrar, dacă se găsește o pereche inegală, se oprește verificarea și se afișează "NU".

Algoritm scris în C++:

Notă: Acestă problemă este rezolvată folosind alocare statică a memoriei folosind sintaxă simplă C++.

```
#include <iostream>

void citireTablou(int tablou[], int& dimensiune)
{
    std::cin >> dimensiune;
    for (int i = 0; i < dimensiune; i++)
        std::cin >> tablou[i];
}

bool verificareSimetrie(int tablou[], int dimensiune)
{
    bool esteSimetric = true;
    for (int i = 0; i < dimensiune / 2; i++) {
        if (tablou[i] != tablou[dimensiune - 1 - i]) {
            esteSimetric = false;
            break;
        }
    }
    return esteSimetric;
}

int main() {
    const int dimensiuneMaxima = 100;
    int dimensiune, tablou[dimensiuneMaxima];

    citireTablou(tablou, dimensiune);

    if (verificareSimetrie(tablou, dimensiune))
        std::cout << "DA";
    else
        std::cout << "NU";

    return 0;
}
```

Justificarea soluției propuse: Algoritmul compară doar elementele corespunzătoare de la începutul și sfârșitul tabloului. Dacă toate coincid, tabloul este palindrom. Oprirea la prima nepotrivire optimizează timpul de execuție. Testat pe cazuri simple (ex: $[1, 2, 3, 2, 1] \rightarrow$ "DA", $[1, 2, 3] \rightarrow$ "NU") confirmă corectitudinea.

Analiza complexității:

- *Complexitate temporală:* $O(\frac{n}{2}) = O(n)$ - în cel mai rău caz se parcurge jumătate din tablou.
- *Complexitate spațială:* $O(n)$ - se utilizează un tablou static de dimensiune maximă 100.

2. Se citesc n numere întregi într-un tablou. Să se afișeze diferența dintre suma elementelor de pe poziții pare și suma elementelor de pe poziții impare.

Rezolvare:
Date de intrare: Un număr natural n $(1 \leq n \leq 100)$, urmat de n numere întregi. Acestea se stochează într-un tablou unidimensional cu alocare statică.
Date de ieșire: Se va afișa diferența dintre suma elementelor aflate pe poziții pare și suma elementelor aflate pe poziții impare. (Pozițiile se consideră începând de la 0.)

Algoritm de rezolvare:

- Se citește valoarea lui n și apoi cele n numere întregi, stocate într-un `std::array<int, 100>`.
- Se inițializează două variabile: `sumaPare = 0`, `sumaImpare = 0`.
- Se parcurge tabloul cu un ciclu `for` de la $i = 0$ la $i < n$:
  - dacă $i \mod 2 = 0$, se adaugă $a[i]$ la `sumaPare`
  - altfel, se adaugă $a[i]$ la `sumaImpare`
- Se calculează diferența: `diferență = sumaPare - sumaImpare`
- Se afișează rezultatul.

Algoritm scris în C++:
Notă: Acestă problemă este rezolvată folosind alocare statică a memoriei folosind `std::array`.

```
#include <iostream>
#include <array>
```

```
void citireTablou(std::array<int, 100>& tablou, int&
    dimensiune) {
     std::cin >> dimensiune;
     for (int i = 0; i < dimensiune; i++)
         std::cin >> tablou[i];
}

int calculareDiferenta(std::array<int, 100> tablou, int
    dimensiune) {
     int sumaPare = 0, sumaImpare = 0;
     for (int i = 0; i < dimensiune; i++) {
         if (i % 2 == 0)
             sumaPare += tablou[i];
         else
             sumaImpare += tablou[i];
     }
     return sumaPare - sumaImpare;
}

int main() {
     int dimensiune;
     std::array<int, 100> tablou;
     int suma_pare = 0, suma_impare = 0;

     citireTablou(tablou, dimensiune);
     std::cout << calculareDiferenta(tablou, dimensiune);
     return 0;
}
```

Justificarea soluției propuse: Pentru un tablou de forma $[5, 3, 7, 2, 9]$, elementele de pe poziții pare sunt $5, 7, 9$ (sumă = 21), cele de pe poziții impare sunt $3, 2$ (sumă = 5). Diferența este $21 - 5 = 16$. Algoritmul furnizează rezultatul corect și se comportă așteptat pe toate cazurile testate.

Analiza complexității:

- *Complexitate temporală:* $O(n)$ - se parcurge o singură dată tabloul.
- *Complexitate spațială:* $O(n)$ - tabloul este stocat într-un `std::array` static de dimensiune fixă.

3. Se dau n numere naturale în intervalul $[0, 100]$. Să se construiască și să se afișeze un tablou de frecvență în care pe poziția i se află numărul de apariții ale valorii i în tabloul

inițial.

Rezolvare:

Date de intrare: Se dă un număr natural n ($1 \leq n \leq 10^5$), urmat de n numere naturale din intervalul $[0, 100]$.

Date de ieșire: Se va afișa un tablou de frecvență `frecv` în care pe poziția i ($0 \leq i \leq 100$) se află numărul de apariții ale valorii i în tabloul inițial.

Algoritm de rezolvare:

- Se citește valoarea lui n și apoi cele n numere.
- Se alocă dinamic două tablouri:
  - `tablou`- tabloul inițial, de dimensiune n
  - `frecventa` - tabloul de frecvență, de dimensiune 101 (pentru valorile $0 \ldots 100$)
- Se inițializează tabloul `frecv` cu zero.
- Se frecventa tabloul `tablou`, iar pentru fiecare valoare `tablou[i]`, se incrementează elementul `frecventa[a[i]]`.
- Se afișează valorile din `frecventa`, inclusiv pozițiile lor.
- Se eliberează memoria alocată dinamic cu `delete[]`.

Algoritm scris în C++:

Notă: Acestă problemă este rezolvată folosind alocare dinamică a memoriei folosind `int*`.

```
#include <iostream>

void citireTablou(int*& tablou, int& dimensiune) {
    std::cin >> dimensiune;

    tablou = new int[dimensiune];    // alocare dinamica a
       tabloului de input
    for (int i = 0; i < dimensiune; i++)
        std::cin >> tablou[i];
}
void populareFrecventa(int* tablou, int*&frecventa, int
   dimensiune, int numarMaximFrecventa) {
```

```
    frecventa = new int[numarMaximFrecventa + 1]; // alocare
       dinamica a tabloului de frecventa

    // initializare tabloul de frecventa cu 0
    for (int i = 0; i <= numarMaximFrecventa; i++) {
        frecventa[i] = 0;
    }
    for (int i = 0; i < dimensiune; i++) {
        frecventa[tablou[i]]++;
    }
}

void afisareFrecvente(int*& frecventa, int
   numarMaximFrecventa) {
    // afisarea tabloului de frecventa
    for (int i = 0; i <= numarMaximFrecventa; i++) {
       if (frecventa[i] > 0) {
            std::cout << "Valoarea " << i << " apare de " <<
               frecventa[i] << " ori.\n";
       }
    }
}

int main() {
    int dimensiune, numarMaximFrecventa = 100;
    int *tablou, *frecventa;

    citireTablou(tablou, dimensiune);
    populareFrecventa(tablou, frecventa, dimensiune,
       numarMaximFrecventa);
    afisareFrecvente(frecventa, numarMaximFrecventa);

    // eliberarea memoriei
    delete[] tablou;
    delete[] frecventa;
    return 0;
}
```

Justificarea soluției propuse. Pentru fiecare valoare $x \in [0, 100]$, algoritmul păstrează în `frecventa[x]` numărul de apariții ale valorii x în tabloul inițial. La parcurgerea tabloului, pentru fiecare element `tablou[i]` se execută operația `frecventa[tablou[i]]++`, astfel încât fiecare apariție a unei valori este numărată exact o singură dată. După parcurgerea întregului tablou, `frecventa[x]` conține exact numărul de apariții ale valorii x. Deoarece toate valorile din tabloul inițial aparțin intervalului $[0, 100]$, accesarea poziției

`frecventa[tablou[i]]` este validă pentru fiecare element.

Analiza complexității:

- *Complexitate temporală:* $O(n)$ - se parcurge tabloul o singură dată.
- *Complexitate spațială:* $O(n + k)$, k constantă - pentru tabloul de intrare și tabloul de frecvență.
- **Observație:** Tabloul de frecvență este de dimensiune constantă ($k = 101$), deci costul său nu crește cu dimensiunea lui n.

4. Se dă un tablou unidimensional cu n elemente întregi. Să se determine poziția pe care se află cel mai lung subșir strict crescător și să se afișeze elementele acestuia.

Rezolvare:
Date de intrare: Se dă un tablou unidimensional cu n elemente întregi, $n \geq 1$.
Date de ieșire: Se va afișa poziția de început (indexul) a celui mai lung subșir strict crescător din tablou, precum și elementele acestuia.

Algoritm de rezolvare:

- Se citește numărul n și apoi cele n elemente într-un vector `tablou`.
- Se parcurge vectorul și se determină cel mai lung subșir strict crescător:
  - Se folosesc 4 variabile:
    - `lungCurent` - lungimea subșirului actual,
    - `startCurent` - indexul de început al subșirului curent,
    - `lungMax` - lungimea maximă găsită,
    - `startMax` - indexul de început al subșirului maxim.
  - Se compară fiecare element cu precedentul. Dacă este mai mare, subșirul continuă. Altfel, se verifică dacă subșirul curent este mai lung decât cel precedent și se actualizează valorile maxime.

- După parcurgerea completă, se afișează subșirul cu lungimea maximă începând de la `startMax`.

Algoritm scris în C++:

Notă: Acestă problemă este rezolvată prin alocare dinamică a memoriei folosind `std::vector`

```
#include <iostream>
#include <vector>

void citireTablou(std::vector<int>& tablou, int& dimensiune) {
    std::cin >> dimensiune;

    tablou.resize(dimensiune);
    for (int i = 0; i < dimensiune; i++)
        std::cin >> tablou[i];
}

void gasireSubsir(std::vector<int> tablou, int dimensiune) {
    int lungCurent = 1, startCurent = 0;
    int lungMax = 1, startMax = 0;

    for (int i = 1; i < dimensiune; i++) {
        if (tablou[i] > tablou[i - 1]) {
            lungCurent++;
        } else {
            if (lungCurent > lungMax) {
                lungMax = lungCurent;
                startMax = startCurent;
            }
            lungCurent = 1;
            startCurent = i;
        }
    }
    // Ultima verificare dupa parcurgere
    if (lungCurent > lungMax) {
        lungMax = lungCurent;
        startMax = startCurent;
    }
    std::cout << "Subsirul strict crescator incepe la indexul
       "
         << startMax << " si are lungimea " << lungMax <<
            ":\n";

    for (int i = startMax; i < startMax + lungMax; i++) {
        std::cout << tablou[i] << " ";
    }
```

```
    std::cout << std::endl;
}

int main() {
    std::vector<int> tablou;
    int dimensiune;

    citireTablou(tablou, dimensiune);
    gasireSubsir(tablou, dimensiune);

    return 0;
}
```

Justificarea soluției propuse. Algoritmul menține în permanență cel mai lung subșir strict crescător care se termină la poziția curentă, folosind variabilele `lungCurent` și `startCurent`. Dacă `tablou[i] > tablou[i-1]`, atunci condiția de creștere strictă este îndeplinită, iar subșirul curent poate fi extins prin incrementarea lui `lungCurent`. În caz contrar, subșirul crescător curent se încheie la poziția $i - 1$, iar lungimea și poziția sa de început sunt comparate cu valorile maxime memorate până în acel moment. Dacă subșirul curent este mai lung, valorile `lungMax` și `startMax` sunt actualizate.
La finalul parcurgerii este necesară o verificare suplimentară, deoarece ultimul subșir strict crescător se poate termina chiar la ultimul element al vectorului și, în acest caz, nu există un element ulterior care să determine intrarea în ramura `else`. Prin urmare, după parcurgerea întregului vector, ultimul subșir este comparat cu cel mai lung subșir memorat. Astfel, la final, `lungMax` reprezintă lungimea celui mai lung subșir strict crescător, iar `startMax` reprezintă poziția sa de început.

Analiza complexității:

- *Complexitate temporală:* $O(n)$ - se parcurge o singură dată vectorul.
- *Complexitate spațială:* $O(n)$ - pentru stocarea vectorului `tablou`.

5.3 Probleme propuse - Vectori simpli

*

1. Se citesc două numere naturale n și k, apoi un șir de n numere întregi. Afișați toate elementele divizibile cu k.
   Exemplu: Pentru $[4, 10, 7, 15, 20, 25, 11]$ și $k = 5$, afișați 10, 15, 20, 25.

2. Se citesc trei numere naturale n, x și y, apoi un șir de n numere întregi. Înlocuiți toate aparițiile lui x cu y în șir, fără a crea un șir nou.
   Exemplu: Pentru $[1, 3, 3, 4]$, $x = 3$, $y = 0 \rightarrow [1, 0, 0, 4]$

3. Se citesc două șiruri de numere întregi de dimensiuni diferite. Creați un nou șir concatenând cele două șiruri date.
   Exemplu: Pentru $[1, 2]$ și $[3, 4]$, rezultatul este [1, 2, 3, 4].

4. Se citește un număr natural n, apoi un șir de n numere întregi. Calculați media aritmetică a elementelor care apar o singură dată în vector.
   Exemplu: Pentru [1, 2, 2, 3], elementele unice sunt 1, 3 și au media aritmetică 2.

5. Se citește un număr natural n, apoi un șir de n. Afișați toate elementele de pe poziții prime.
   Exemplu: Pentru [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] $\rightarrow$ 3, 4, 6, 8, 12 de pe pozițiile 2, 3, 5, 7, 11

**

1. Se citesc un număr natural n și cele n elemente ale unui vector. Să se verifice eficient dacă elementele vectorului sunt deja sortate fie crescător, fie descrescător.
   Exemplu: Vectorul [1, 7, 8, 11, 15] are elementele sortate crescător.

2. Se citesc trei numere naturale n, x și y, apoi un șir de n numere întregi. Rotiți șirul cu x poziții la stânga, apoi cu y poziții la dreapta.
   Exemplu: Pentru [1, 2, 3, 4] și $x = 1$, $y = 2$, rotirea la stânga este [2, 3, 4, 1], iar rotirea finală spre dreapta este [4, 1, 2, 3].

3. Se citesc un număr natural n și cele n elemente ale unui vector. Să se înlocuiască elementele de pe pozițiile pare cu cea mai mică cifra din număr iar cele de pe pozițiile impare cu 0 dacă numărul e prim și cu 1 altfel.
   Exemplu: Pentru [17, 23, 9, 15, 42, 5], pozițiile pare (0, 2, 4) au elementele 17, 9, 42, deci se obțin minimele 1, 9, 2; pozițiile impare (1, 3, 5) au elementele 23, 15, 5, dar conform verificării de valori prime se obțin 0, 1, 0. Vectorul rezultat este [1, 0, 9, 1, 2, 0].

4. Se citesc un număr natural n și cele n elemente ale unui vector. Să se șteargă din vector acele elemente care sunt egale cu suma vecinilor săi.
   Exemplu: Pentru [2, 2, 5, 7, 2, 19, 6, 10, 4], am eliminat valorile 2 (pentru că prima valoare are un singur vecin care este identic), 7 (pentru că 5 + 2 = 7) și 10 (pentru că 6 + 4 = 10). Vectorul rezultat este [2, 5, 2, 19, 6, 4].

5. Se citesc un număr natural n și cele n elemente ale unui vector. Să se insereze în vector între oricare două valori vecine suma și produsul acestora. Determinarea sumei și a produsului a două numere se va realiza în cadrul unui singure funcții.
   Exemplu: Pentru [1, 3, 2, 4] vectorul rezultat este [1, 4, 3, 5, 6, 2, 6, 8, 4].

***

1. Se dă un vector v cu n elemente. Să se elimine din vector elementul cu indicele de poziție p citit de la tastatură), iar p să se insereze la începutul vectorului. Să se afișeze noul vector.
   Exemplu: Pentru [2, 51, 2, 91, 3] și $p = 3$, se elimină 91, iar vectorul rezultat este [3, 2, 51, 2, 3]

2. Se citesc două numere naturale n și m și doi vectori crescători de lungime n, respectiv m. Să se creeze și să se afișeze vectorul ordonat crescător obținut prin reuniunea celor doi vectori.
   Exemplu: Pentru $v1 = [1, 2, 5, 10, 13, 20]$ și $v2 = [1, 6, 7, 9, 15, 25]$, obținem: [1, 1, 2, 5, 6, 7, 9, 10, 13, 15, 20, 25]

3. Se citesc un număr natural n și n medii (numere reale cu doua zecimale cu valori cuprinse intre 1 și 10). Să se afișeze o statistică a mediilor astfel: numărul de medii cuprinse în

intervalul $[1, 2]$, numărul de medii cuprinse în intervalul $(2, 3]$, ..., numărul de medii cuprinse în intervalul $(9, 10]$.

4. Se citesc un număr n și un vector de n elemente numere naturale. Verificați dacă pentru fiecare element prim din v, oglinditul său există în vector și este tot prim.
   Exemplu de vector care respectă cerința: [13, 17, 31, 71, 9]

5. Se citesc un număr natural n și cele n elemente ale unui vector. Înlocuiți fiecare element cu lista de divizori ai săi. Numărați apoi elementele distincte.
   Exemplu: [12, 18, 5] $\rightarrow$ [1, 2, 3, 4, 6, 12, 1, 2, 3, 6, 9, 18, 1, 5] $\rightarrow$ 9 elemente distincte

****

1. Se citesc două numere naturale n și k și cele n elemente ale unui vector. Înlocuiți fiecare element cu produsul celorlalte elemente, apoi aplicați mod k:
   Exemplu: $[2, 3, 4]$, $k = 5 \rightarrow [3 * 4\%5 = 2, 2 * 4\%5 = 3, 2 * 3\%5 = 1] \rightarrow$ [2, 3, 1]

2. Se citește un vector de șiruri de caractere.
   - Determinați cel mai lung prefix comun al tuturor șirurilor din vector.
     Exemplu: Pentru ["flower", "flow", "flight"] avem prefixul comun maxim "fl".
   - Determinați dacă toate șirurile din vector sunt anagrame între ele (au aceleași caractere, dar în ordine diferită).
     Exemplu: Vectorul ["listen", "silent", "enlist"] conține doar anagrame.
   - Grupați șirurile care conțin exact aceleași caractere (nu neapărat în aceeași ordine).
     Exemplu: Vectorul ["abc", "bca", "xyz", "zyx"] are ["abc", "bca"] și ["xyz", "zyx"].
   - Găsiți șirul care conține cele mai multe vocale distincte (a, e, i, o, u).
     Exemplu: Vectorul ["hello", "world", "education"], "education" conține 5 vocale distincte

3. Se citesc un număr natural n și cele n elemente ale unui vector. Găsiți toate perechile de elemente $(v[i], v[j])$ pentru care concatenarea lor formează un pătrat perfect.
   Exemplu: Pentru [1, 44, 25] avem ca și pătrare perfecte 1, 144, 441, 25.

4. Se citesc un număr natural n și cele n elemente ale unui vector. Determinați câte "vârfuri" există (avem o pantă crescătoare până la un punct maxim - vârful, iar apoi avem o pantă descrescătoare). Afișați toate vârfurile care au suma pantelor egale.
   Exemplu: În [1, 2, 4, 3, 0, 7, 2] avem 2 vârfuri 4 și 7, iar 4 are pantele cu sumă egală (1+2=3+0).

5. Un număr natural se reține într-un vector (număr mare), astfel încât fiecare componentă a vectorului conține câte o cifră a numărului. Să se înmulțească numărul cu un număr între 1 și 9 (generat aleatoriu). Documentație random: `https://cplusplus.com/reference/cstdlib/rand/`

*****

1. Găsiți un indice k într-un vector numeric astfel încât suma primelor k elemente să fie egală cu produsul ultimelor $n - k$ elemente, toate elementele până la k sunt prime, respectiv toate elementele după k sunt puteri ale lui 2.
   Exemplu: Pentru [2, 4, 8, 1, 2, 4, 1, 2], dacă luăm $k = 3$, primele trei elemente au suma 14, ultimele $n - k = 5$ elemente au produsul 16, iar acestea sunt puteri ale lui 2.

2. Se citește un vector de șiruri de caractere. Găsiți un subșir care apare în toate șirurile și are frecvența vocalelor în ordine descrescătoare ($a \geq e \geq i \geq o \geq u$).
   Exemplu: În ["bdaeiou", "aeiocc", "aeicuo"] avem șirul "aei"

3. Un număr natural se reține într-un vector (număr mare), astfel încât fiecare componentă a vectorului conține câte o cifră a numărului. Fără a a transforma vectorul în număr convertiți numărul mare în baza 2, apoi verificați dacă reprezentarea binară are mai multe 1-uri decât 0-uri.
   Exemplu: Pentru $v1$ = [4, 7, 8] avem $v2$ = [1, 1, 1, 0, 1, 1, 1, 1, 0]

4. Se citesc un număr natural n și cele n elemente ale unui liste. Scopul este să se scrie:
   - O funcție care determină câte secvențe de elemente consecutive convexe și/sau concave există în această listă.
   - O funcție care identifică eficient minimul pentru secvențele convexe

- O funcție care identifică eficient maximul pentru cele concave
- O funcție care determină numărul de elemente al celei mai lungi secvențe găsite

Exemplu: Pentru lista 4 2 5 7 8 3 1 9 8 1 2 5 0 9 11 1 2 3 avem următoarele secvențe posibile:

4, 2, 5, 7, 8 convexă cu minim local 2

2, 5, 7, 8, 3, 1 concavă cu maxim local 8

8, 3, 1, 9 convexă cu minim local 1

1, 9, 8, 1 concavă cu maxim local 9

9, 8, 1, 2, 5 convexă cu minim local 1

1, 2, 5, 0 concavă cu maxim local 5

5, 0, 9, 11 convexă cu minim local 0

0, 9, 11, 1 concavă cu maxim local 11

11, 1, 2, 3 convexă cu minim local 1

Există 5 secvențe convexe și 4 secvențe concave. Secvența cea mai lungă are 6 elemente

Definiții:

- O secvență **convexă** este una în care există un minim local, cu elementele descrescătoare până la minim și crescătoare după minim.
- O secvență **concavă** este una în care există un maxim local, cu elementele crescătoare până la maxim și descrescătoare după maxim.

5. Se dau două numere naturale care reprezintă o fracție subunitară, iar sarcina este să se descompună această fracție într-o sumă de fracții cu numărătorul 1. Să se salveze într-un vector valorile numitorilor.
   Exemple:
   $\frac{12}{57} = \frac{1}{3} + \frac{1}{19} \rightarrow$ vectorul [3, 19]

   $\frac{11}{17} = \frac{1}{2} + \frac{1}{5} + \frac{1}{34} + \frac{1}{170} \rightarrow$ vectorul [2, 5, 34, 170]

5.4 Probleme propuse - Vectori cu structuri / obiecte

*

1. Se citește un vector de date calendaristice. Pentru fiecare dată știm ziua, luna și anul. Să se afișeze cea mai recentă dată (cea mai mare).

2. Se analizează performanțele școlare ale elevilor unei clase. Fiecare elev este caracterizat prin nume și rezultatele la două materii fundamentale: matematică și limba română. Elevii vor fi citiți dintr-un fișier sau de la tastatură.

   (a) Să se afișeze numele elevilor care au media generală la cele două materii de cel puțin 8.

   (b) Să se calculeze și să se afișeze nota medie la matematică a întregii clase.

   (c) Să se afișeze în ordine alfabetică detaliile elevilor din clasă.

3. Se citesc informații despre o colecție de pisici, fiecare identificată printr-un nume, o vârstă exprimată în luni și o greutate în kilograme.

   (a) Să se afișeze numele pisicilor cu vârsta mai mică de 12 luni (1 an).

   (b) Să se calculeze și să se afișeze greutatea totală a tuturor pisicilor din vector.

   (c) Să se afișeze toate pisicile în ordine descrescătoare din punct de vedere al vârstei.

**

1. Se citește o listă de angajați dintr-o companie, fiecare descris prin următoarele atribute: nume, departament (șir de caractere), salariu (număr real) și ani_experiență (număr întreg).

   (a) Să se calculeze și să se afișeze salariul mediu pe departamente. Pentru angajații cu salariu sub media departamentului lor să se afișeze numele și diferența față de medie.

(b) Să se mărească cu 10% salariul angajaților din departamentul IT cu peste 5 ani experiență.

2. Se citește o listă de produse disponibile într-un magazin, fiecare caracterizat prin nume, preț (număr real, în lei), stoc (bucăți disponibile) și statut de reducere (dacă produsul este la reducere).

   (a) Să se afișeze numele și prețul produselor cu stoc mai mic de 10 bucăți.

   (b) Să se calculeze valoarea totală a stocului pentru toate produsele.

   (c) Să se afișeze în ordine descrescătoare din punct de vedere al prețului, numele produselor aflate la reducere și care au prețul peste 50 de lei.

3. Să se gestioneze o colecție de filme, fiecare descris prin următoarele atribute: titlu (șir de caractere), an (număr întreg, anul lansării), rating (număr real, de la 1.0 la 10.0) și gen (șir de caractere).

   (a) Să se scrie o funcție care verifică dacă există un film cu un anumit titlu, iar în caz afirmativ afișează anul și ratingul lui.

   (b) Să se scrie o funcție care afișează filmul din genul "comedie" cu rating-ul cel mai mare.

   (c) Să se scrie o funcție care sortează crescător toate filmele după rating-ul lor.

***

1. Urmărim mai multe operațiuni bancare, unde fiecare tranzacție este descrisă prin: id (număr întreg, identificator unic), tip (șir de caractere: „depunere” sau „retragere”), suma (număr real, pozitiv) și cont_iban (șir de caractere). Se citește lista de tranzacții.

   Pentru fiecare cont IBAN distinct din listă să se genereze un raport ce include soldul final (calculat ca totalul depunerilor minus totalul retragerilor), numărul total de tranzacții efectuate și tipul ultimei tranzacții înregistrate.

2. Se dă o lista de rețete, pentru fiecare rețeta avem o lista de ingrediente. Pentru fiecare ingredient știm un nume și un gramaj/cantitate.

   (a) Să se găsească ingredientul care apare cel mai frecvent din toată lista de rețete și să se afișeze în ce cantitate totală apare acesta.

   (b) Să se sorteze descrescător toate rețetele după numărul de ingrediente din lista lor.

****

1. Un sistem de management pentru studenții unei universități gestionează informații structurate astfel: fiecare student este caracterizat prin nume, an studiu și o listă de discipline fiecare având un nume, număr credite și nota studentului la disciplina respectivă.

   (a) Identificați toți studenții din anul 3 care au cel puțin o disciplină cu media grupei sub 5.0. Pentru acești studenți, afișați numele lor și detaliile disciplinelor respective.

   (b) Un student primește bursă dacă media la toate disciplinele sale este ≥ 8.5). Valoarea bursei se calculează astfel: 300 lei * numărul total de credite din disciplinele promovate (unde nota ≥ 5.0). Salvați pentru fiecare student valoarea bursei (sau 0 în cazul în care nu primește).

   (c) Afișați în mod descrescător studenții din punct de vedere al valorii burselor.

2. În cadrul sistemului de administrare a unei rețele de metrou, se dorește să se implementeze un sistem de monitorizare a numărului de călători pe diferite rute. Vom reprezenta o linie de metrou printr-un id unic și o lista de stații aflate pe acest traseu. O stație are o denumire unică în această rețea și conține informații despre numărul de călători care frecventează în medie această stație într-o singura zi. Aceeași stație (cu denumire unică) poate apărea pe mai multe linii, dar datele sale sunt identice peste tot în acest caz. Să se scrie o funcție care să afișeze detaliile stațiilor din rețea după aplicarea unei sortări descrescătoare din punct de vedere al numărului de călători care frecventează în medie aceste stații într-o singura zi.

3. În cadrul unei companii de livrări rapide, se dorește implementarea unui sistem de evidență a șoferilor și a mașinilor pe care le conduc. Fiecare șofer este identificat printr-un cod unic,

are un nume, un prenume și un istoric de performanță exprimat printr-un scor de evaluare (număr real). Fiecărui șofer îi este asociată exact o mașină din flotă, iar această relație este stabilită direct între cele două entități. O mașină are un model, un consum mediu și un an al fabricației.

(a) Să se sorteze șoferii după scorul de evaluare al lor. În cazul în care doi șoferi au același scor, delimitarea se va face crescător în funcție de anul de fabricație al mașinii.

(b) Să se determine modelul de mașină cel mai frecvent întâlnit în flotă. Afișați acest model și detaliile tuturor șoferilor care dețin acest model.

*****

1. Se consideră un vast univers digital, în care un sistem planetar simulează cu precizie ordinea cosmică și ambițiile explorării spațiale. În acest cosmos virtual, fiecare corp ceresc este atent modelat: un **Satelit** este definit prin numele său, masa sa (număr real în kilograme) și anul descoperirii. Acesta gravitează în jurul unei **Planete**, care are la rândul ei un nume, o temperatură medie (număr real, în Kelvin), o listă de sateliți asociați și un număr de coloniști care și-au stabilit acolo așezăminte. Toate aceste planete orbitează într-un **SistemSolar**, o entitate care reunește mai multe planete, este legată de o galaxie (prin nume, șir de caractere) și poartă amprenta istorică a anului în care a fost descoperit.

   (a) Să se sorteze planetele dintr-un sistem solar în ordinea descrescătoare a numărului de coloniști și să se afișeze pentru fiecare: numele, temperatura medie și numărul de sateliți naturali.

   (b) Să se caute și să se afișeze toate planetele care au cel puțin un satelit descoperit înainte de anul 1900. Pentru fiecare astfel de planetă se afișează detaliile celui mai vechi satelit.

   (c) Să se afișeze toate planetele care îndeplinesc simultan următoarele condiții: temperatura medie este sub 250 Kelvin și au cel puțin 100 de coloniști. Sortați rezultatele crescător după temperatura medie și, pentru fiecare planetă, afișați detaliile sale.

2. Într-un sistem digital de administrare urbană, orașele sunt modelate prin structuri ierarhice ce reflectă realitatea urbanismului modern. Fiecare **Oraș** este definit printr-un nume, o listă de zone urbane și anul fondării (număr întreg). O **ZonăUrbană** conține un identificator numeric, o suprafață exprimată în kilometri pătrați, un vector de clădiri și un indice de poluare al mediului (valoare între 0 și 100). La rândul ei, fiecare **Clădire** este identificată printr-un cod unic, un număr de etaje și un indicator dacă dispune de panouri solare sau nu.

   (a) Afișați toate zonele urbane cu un indice de poluare peste 75. Pentru fiecare astfel de zonă, calculați densitatea clădirilor (număr clădiri / suprafață) și identificați procentul clădirilor care nu dispun de panouri solare.

   (b) Sortați descrescător toate clădirile dintr-un oraș în funcție de numărul de etaje. Afișați pentru fiecare clădire: codul ID, zona din care face parte și dacă are panouri solare.

   (c) Identificați toate zonele urbane în care peste 70% dintre clădiri au panouri solare. Sortați și afișați aceste zone crescător după indicele de poluare.

   (d) În fiecare zonă urbană, identificați clădirea cu cel mai mare număr de etaje și afișează codul ei, numărul de etaje și dacă este sau nu echipată cu panouri solare.

Capitolul 6

Tablouri bidimensionale

6.1 Aspecte teoretice

Tablourile bidimensionale, cunoscute și sub numele de matrici, permit stocarea datelor într-o structură organizată pe rânduri și coloane. Din perspectivă tehnică, acestea sunt implementate ca un tablou de tablouri de același tip.

Ce este un tablou bidimensional?

Un **tablou bidimensional** este o colecție de elemente de același tip organizate pe două dimensiuni, accesibile prin doi indici numerici. Fiecare element ocupă o poziție unică în tablou, numerotată de la 0, 0 până la $m-1$, $n-1$, unde m și n reprezintă dimensiunile tabloului.

Exemplu conceptual:

```
int valori[2][3];
```

Această instrucțiune definește o matrice cu 2 rânduri și 3 coloane cu elemente de tip `int`, fiecare având o poziție determinată de doi indici:

```
[0][0]   [0][1]   [0][2]
[1][0]   [1][1]   [1][2]
```

Caracteristici ale tablourilor bidimensionale

Tablourile bidimensionale sunt structuri de date organizate în rânduri și coloane, iar toate elementele au același tip de date, precum `int`, `float` sau `char`. Accesul la elemente se realizează prin doi indici: unul pentru rând și unul pentru coloană. Aceste tablouri ocupă un bloc continuu de memorie, respectând ordinea rândurilor (row-major order). Dimensiunile, respectiv numărul de rânduri și coloane, trebuie cunoscute sau specificate la compilare. Ele facilitează procesarea datelor bidimensionale, cum ar fi matrici, tabele sau imagini, iar inițializate poate fi

realizată direct la declarare.

Utilitate și aplicabilitate

Tablourile bidimensionale sunt utilizate pentru reprezentarea și manipularea matricelor matematice, precum și pentru stocarea și procesarea datelor tabelare, cum ar fi cele din foile de calcul. Ele permit reprezentarea imaginilor digitale, unde pixelii sunt organizați pe linii și coloane, și sprijină implementarea algoritmilor grafici sau a jocurilor, de exemplu pentru hărți 2D sau table de joc. De asemenea, sunt folosite pentru gestionarea relațiilor între entități, precum grafurile sub formă de matrice de adiacență, și în aplicații de procesare a semnalului sau în analize științifice.

6.1.1 Alocarea statică a tablourilor bidimensionale

Alocarea statică a tablourilor bidimensionale reprezintă procesul prin care memoria necesară pentru stocarea elementelor tabloului este rezervată la compilare, având dimensiuni fixe specificate în momentul declarării.

Sintaxă generală

```
tip_de_data nume_tablou[numar_linii][numar_coloane];
```

Parametrii sunt `tip_de_data`, care reprezintă tipul elementelor (ex: `int`, `double`, `char`), `nume_tablou` pentru identificatorul tabloului și `numar_linii` și `numar_coloane`, valori constante cunoscute la compilare.

Exemplu simplu

```
int matrice[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

Funcție de citire a valorilor dintr-un tablou static

```
void citireMatrice(
int matrice[numar_maxim_randuri][numar_maxim_coloane],
int numar_linii,
int numar_coloane) {
```

```
    for (int i = 0; i < numar_linii; i++) {
        for (int j = 0; j < numar_coloane; j++) {
            std::cin >> matrice[i][j];
        }
    }
}
```

Această funcție permite citirea elementelor dintr-un tablou bidimensional, accesând fiecare element prin doi indici, corespunzători rândului și coloanei. Apelul funcției se face astfel: `citireMatrice(matrice, numar_linii, numar_coloane);`.

Funcție de afișare a unui tablou bidimensional alocat static

```
void afisareMatrice(int
   matrice[numar_maxim_randuri][numar_maxim_coloane], int
   numar_linii, int numar_coloane) {

    for (int i = 0; i < numar_linii; i++) {
        for (int j = 0; j < numar_coloane; j++) {
            std::cout << matrice[i][j] << " ";
        }
        std::cout << std::endl;
    }
}
```

Această funcție parcurge matricea rând cu rând și afișează fiecare element. Poate fi apelată astfel: `afisareMatrice(matrice, numar_linii, numar_coloane);`.

Funcție de inserare a unui linii noi într-un tablou bidimensional alocat static

```
void inserareLinie(
    int matrice[numar_maxim_randuri][numar_maxim_coloane],
    int& numar_linii,
    int numar_coloane,
    int pozitie,
    int linie_noua[numar_maxim_coloane]
) {
    // Verificare pozitie valida
    if (pozitie < 0 || pozitie > numar_linii) {
        std::cout << "Pozitie invalida!\n";
        return;
    }

    // Mutam liniile de la pozitia inserarii in jos pentru a face
       loc
    for (int i = numar_linii; i > pozitie; i--) {
```

```
        for (int j = 0; j < numar_coloane; j++) {
            matrice[i][j] = matrice[i - 1][j];
        }
    }

    // Inseram noua linie la pozitia dorita
    for (int j = 0; j < numar_coloane; j++) {
        matrice[pozitie][j] = linie_noua[j];
    }

    // Actualizam numarul de linii
    numar_linii++;
}
```

Apeluri posibile:

- Inserare la început:

  `inserareLinie(matrice, numar_linii, numar_coloane, 0, linie_noua);`

- Inserare la mijloc):

  `inserareLinie(matrice, numar_linii, numar_coloane, numar_linii/2, linie_noua);`

- Inserare la final:

  `inserareLinie(matrice, numar_linii, numar_coloane, numar_linii, linie_noua);`

Funcție de inserare a unui coloane noi într-un tablou bidimensional alocat static

```
void inserareColoana(
    int matrice[numar_maxim_randuri][numar_maxim_coloane],
    int numar_linii,
    int& numar_coloane,
    int pozitie,
    int coloana_noua[numar_maxim_randuri]) {
    // Verificare pozitie valida
    if (pozitie < 0 || pozitie > numar_coloane) {
        std::cout << "Pozitie invalida!\n";
        return;
    }

    // Mutam coloanele de la pozitia inserarii spre dreapta pentru
       a face loc
    for (int i = 0; i < numar_linii; i++) {
        for (int j = numar_coloane; j > pozitie; j--) {
            matrice[i][j] = matrice[i][j - 1];
        }
    }
```

```
    // Inseram noua coloana la pozitia dorita
    for (int i = 0; i < numar_linii; i++) {
        matrice[i][pozitie] = coloana_noua[i];
    }

    // Actualizam numarul de coloane
    numar_coloane++;
}
```

Apeluri posibile:

- Inserare la început:

  `inserareColoana(matrice, numar_linii, numar_coloane, 0, coloana_noua);`

- Inserare la mijloc:

  `inserareColoana(matrice, numar_linii, numar_coloane, numar_coloane/2, coloana_noua);`

- Inserare la final:

  `inserareColoana(matrice, numar_linii, numar_coloane, numar_coloane, coloana_noua);`

Funcție de ștergere a unui linii într-un tablou bidimensional alocat static

```
void stergereLinie(
    int matrice[numar_maxim_randuri][numar_maxim_coloane],
    int& numar_linii,
    int numar_coloane,
    int pozitie) {
    // Verificare pozitie valida
    if (pozitie < 0 || pozitie >= numar_linii) {
        std::cout << "Pozitie invalida!\n";
        return;
    }
    // Mutam liniile de dupa pozitia de sters in sus pentru a
       suprascrie linia
    for (int i = pozitie; i < numar_linii - 1; i++) {
        for (int j = 0; j < numar_coloane; j++) {
            matrice[i][j] = matrice[i + 1][j];
        }
    }
   // Actualizam numarul de linii
    numar_linii--;
}
```

Apeluri posibile:

- Ștergere la început:

  `stergereLinie(matrice, numar_linii, numar_coloane, 0);`

- Ștergere la mijloc:

  `stergereLinie(matrice, numar_linii, numar_coloane, numar_linii/2);`

- Ștergere la final:

  `stergereLinie(matrice, numar_linii, numar_coloane, numar_linii - 1);`

Funcție de ștergere a unui linii într-un tablou bidimensional alocat static

```
void stergereColoana(
    int matrice[numar_maxim_randuri][numar_maxim_coloane],
    int numar_linii,
    int& numar_coloane,
    int pozitie) {
    // Verificare pozitie valida
    if (pozitie < 0 || pozitie >= numar_coloane) {
        std::cout << "Pozitie invalida!\n";
        return;
    }
    // Mutam coloanele de dupa pozitia de sters spre stanga pentru
       a suprascrie coloana
    for (int i = 0; i < numar_linii; i++) {
        for (int j = pozitie; j < numar_coloane - 1; j++) {
            matrice[i][j] = matrice[i][j + 1];
        }
    }

    // Actualizam numarul de coloane
    numar_coloane--;
}
```

Apeluri posibile:

- Ștergere la început:

  `stergereColoana(matrice, numar_linii, numar_coloane, 0);`

- Ștergere la mijloc:

  `stergereColoana(matrice, numar_linii, numar_coloane, numar_coloane/2);`

- Ștergere la final:

  `stergereColoana(matrice, numar_linii, numar_coloane, numar_coloane-1);`

Exemplu complet cu citire, inserare și ștergere

```
int main() {
    const int numar_maxim_randuri = 100;
    const int numar_maxim_coloane = 100;
    int matrice[numar_maxim_randuri][numar_maxim_coloane];
    int numar_linii, numar_coloane;

    citireMatrice(matrice, numar_linii, numar_coloane);
    afisareMatrice(matrice, numar_linii, numar_coloane);

    // Definirea unei linii noi pentru inserare
    int linie_noua[numar_maxim_coloane];
    std::cout << "Introdu valorile pentru linia noua:\n";
    for (int j = 0; j < numar_coloane; j++) {
        std::cin >> linie_noua[j];
    }

    // Inserare linie la pozitia 1 (exemplu)
    inserareLinie(matrice, numar_linii, numar_coloane, 1,
       linie_noua);
    std::cout << "Dupa inserare linie:\n";
    afisareMatrice(matrice, numar_linii, numar_coloane);

    // Stergere linie de la pozitia 0 (exemplu)
    stergereLinie(matrice, numar_linii, numar_coloane, 0);
    std::cout << "Dupa stergere linie:\n";
    afisareMatrice(matrice, numar_linii, numar_coloane);

    return 0;
}
```

Observație! Pentru o alternativă modernă și flexibilă a tablourilor statice în cadrul bibliotecii standard C++, se recomandă consultarea clasei `std::array`, prezentată în secțiunea de containere secvențiale (STL).

```
std::array<std::array<int, numar_maxim_coloane>,
   numar_maxim_randuri> tablou;
```

6.1.2 Alocarea dinamică a tablourilor bidimensionale

Alocarea dinamică a tablourilor bidimensionale reprezintă procesul prin care memoria pentru matrice este rezervată în timpul execuției programului, permițând definirea dimensiuni-

lor tabloului într-un mod flexibil și variabil, în funcție de necesități.

Sintaxă generală

```
tip_de_data** nume_tablou;

// alocare
nume_tablou = new tip_de_data*[numar_linii];
for (int i = 0; i < numar_linii; i++) {
    nume_tablou[i] = new tip_de_data[numar_coloane];
}

// elinerare
for (int i = 0; i < numar_linii; i++) {
    delete[] nume_tablou[i];
}
delete[] nume_tablou;
```

În această sintaxă, `tip_de_data` reprezintă tipul elementelor (ex: `int`, `double`, `char`), `nume_tablou` este un pointer către pointer care reprezintă matricea, iar `numar_linii` și `numar_coloane` sunt dimensiuni cunoscute la timpul execuției. Memoria alocată trebuie eliberată cu `delete[]` după utilizare.

Funcțiile de alocare dinamică

Exista 2 variante de alocare dinamică a matricilor, una în care fiecare linie este alocata separat și una în care se alocă tot blocul de memorie odată

1. Varianta în care fiecare linie este alocata separat

```
int** alocaMatrice(int numar_linii, int numar_coloane) {
    int** matrice = new int*[numar_linii];

    for (int i = 0; i < numar_linii; i++) {
        matrice[i] = new int[numar_coloane];
    }
    return matrice;
}
```

2. Varianta în care se alocă tot blocul de memorie odată

```
int** alocaMatrice(int numar_linii, int numar_coloane) {
    int** matrice = new int* [numar_linii];
    matrice[0] = new int[numar_linii * numar_coloane];
```

```
    for (int i = 1; i < numar_linii; i++) {
        matrice[i] = matrice[0] + i * numar_coloane;
    }
    return matrice;
}
```

Ambele funcții creează o matrice bidimensională cu dimensiuni specificate la runtime și returnează pointerul către matricea alocată. Apelul funcției: `alocaMatrice(numar_linii, numar_coloane);`.

În prima variantă, `new int*[numar_linii]` alocă memoria pentru pointerii către linii, iar apoi fiecare linie este alocată separat prin `new int[numar_coloane]`. Astfel, matricea are mai multe zone de memorie separate, câte una pentru fiecare linie.

În a doua variantă, memoria pentru pointerii către linii este alocată separat, dar toate elementele matricei sunt păstrate într-un singur bloc continuu de memorie. matrice[0] indică începutul acestui bloc, iar ceilalți pointeri sunt calculați pornind de la această adresă.

Prima variantă este mai simplă și mai intuitivă, deoarece fiecare linie este tratată ca un tablou separat. În plus, liniile pot avea chiar și dimensiuni diferite dacă programul ar avea nevoie de acest lucru.

A doua variantă este mai eficientă din punct de vedere al organizării memoriei, deoarece elementele sunt consecutive în memorie. Acest lucru poate fi avantajos pentru accesarea succesivă a elementelor și pentru utilizarea cache-ului procesorului.

În prima variantă există mai multe alocări dinamice, deoarece se face câte un new pentru fiecare linie. În a doua variantă există o alocare pentru vectorul de pointeri și o singură alocare pentru toate elementele matricei.

Funcțiile de eliberare a memoriei unui tablou dinamic

Dealocarea se va face diferit, în funcție de metoda de alocare folosită.

Pentru prima variantă trebuie eliberată fiecare linie separat și apoi vectorul de pointeri:

```
void eliberezaMatrice(int**& matrice, int numar_linii) {
    for (int i = 0; i < numar_linii; i++) {
        delete[] matrice[i];
    }
    delete[] matrice;
    matrice = nullptr;
}
```

Pentru a doua variantă, deoarece toate elementele sunt într-un singur bloc, se eliberează mai întâi blocul de elemente și apoi vectorul de pointeri:

```
void eliberezaMatrice(int**& matrice,) {
    delete[] matrice[0];
    delete[] matrice;
    matrice = nullptr;
}
```

Aceste funcții eliberează memoria ocupată de matricea alocată dinamic. Apelul se face astfel: `eliberezaMatrice(matrice, numar_linii);` - pentru prima variantă, respectiv `eliberezaMatrice(matrice)` pentru a doua variantă.

Linia `matrice = nullptr;` nu este obligatorie dar prezentăm în continuare o diferență față de situația în care nu este scrisă. Ambele variante sunt corecte din punct de vedere al dezalocării memoriei, însă adăugarea instrucțiunii `matrice = nullptr;` aduce un beneficiu semnificativ în ceea ce privește siguranța codului. Fără această atribuire, după ce memoria este eliberată, pointerul matrice rămâne să indice către o zonă de memorie care nu mai aparține programului. Dacă cineva ar încerca să acceseze sau să elibereze din nou memoria pentru această matrice după apelul funcției, comportamentul ar fi complet nedefinit, putând duce la coruperea memoriei, crash-uri sau vulnerabilități de securitate greu de depistat. Prin setarea pointerului la nullptr, eliminăm această referință periculoasă, iar orice încercare ulterioară de a folosi matricea va produce o eroare vizibilă (de obicei un segmentation fault) în loc să corupă datele în mod silențios, facilitând astfel depistarea erorilor. Totuși, este esențial de înțeles că în C++ parametrii sunt pasați prin valoare, ceea ce înseamnă că în varianta originală, modificarea `matrice = nullptr;` din interiorul funcției nu afectează pointerul din funcția apelantă, deci pentru ca această măsură de siguranță să fie cu adevărat eficientă, ar trebui ca funcția să primească pointerul prin referință `(int**& matrice)`, astfel încât setarea la nullptr să se reflecte și în afara funcției.

Funcție de citire a valorilor dintr-un tablou dinamic

```
void citireMatrice(int**& matrice, int nr_linii, int nr_coloane) {
    std::cout << "Introdu valorile matricei:\n";
    for (int i = 0; i < nr_linii; i++)
        for (int j = 0; j < nr_coloane; j++)
            std::cin >> matrice[i][j];
}
```

Funcția citește dimensiunile și elementele matricei, creând o matrice dinamică corespunzătoare. Apelul funcției: `citireMatrice(numar_linii, numar_coloane);`.

Funcție de afișare a unui tablou dinamic

```
void afisareMatrice(int** matrice, int numar_linii, int
   numar_coloane) {
    std::cout << "Matricea este:\n";
    for (int i = 0; i < numar_linii; i++) {
        for (int j = 0; j < numar_coloane; j++) {
            std::cout << matrice[i][j] << " ";
        }
        std::cout << std::endl;
    }
}
```

Această funcție afișează matricea rând cu rând. Apelul se face astfel: `afisareMatrice(matrice, numar_linii, numar_coloane);`.

Funcție de inserare a unui linii noi într-un tablou dinamic

```
int** inserareLinie(int** matrice, int& numar_linii, int
   numar_coloane, int pozitie, int linie_noua[]) {
    if (pozitie < 0 || pozitie > numar_linii) {
        std::cout << "Pozitie invalida!\n";
        return matrice;
    }

    // Alocare matrice noua cu o linie in plus
    int** matrice_noua = alocaMatrice(numar_linii + 1,
       numar_coloane);

    // Copiere elemente si inserare linie noua
    for (int i = 0; i < pozitie; i++) {
        for (int j = 0; j < numar_coloane; j++) {
            matrice_noua[i][j] = matrice[i][j];
        }
    }
    for (int j = 0; j < numar_coloane; j++) {
        matrice_noua[pozitie][j] = linie_noua[j];
    }
    for (int i = pozitie; i < numar_linii; i++) {
        for (int j = 0; j < numar_coloane; j++) {
            matrice_noua[i + 1][j] = matrice[i][j];
        }
```

```
    }
    // Eliberare matrice veche
    elibereazaMatrice(matrice, numar_linii);

    numar_linii++;
    return matrice_noua;
}
```

Funcție de inserare a unui coloane noi într-un tablou dinamic

```
int** inserareColoana(int** matrice, int numar_linii, int&
   numar_coloane, int pozitie,
    int coloana_noua[]) {
    if (pozitie < 0 || pozitie > numar_coloane) {
        std::cout << "Pozitie invalida!\n";
        return matrice;
    }

    // Alocare matrice noua cu o coloana in plus
    int** matrice_noua =  alocaMatrice(numar_linii, numar_coloane
       + 1);

    // Copiere elemente si inserare coloana noua
    for (int i = 0; i < numar_linii; i++) {
        for (int j = 0; j < pozitie; j++) {
            matrice_noua[i][j] = matrice[i][j];
        }
        matrice_noua[i][pozitie] = coloana_noua[i];
        for (int j = pozitie; j < numar_coloane; j++) {
            matrice_noua[i][j + 1] = matrice[i][j];
        }
    }
    // Eliberare matrice veche
    elibereazaMatrice(matrice, numar_linii);

    numar_coloane++;
    return matrice_noua;
}
```

Funcție de ștergere a unui linii într-un tablou dinamic

```
int** stergereLinie(int** matrice, int& numar_linii, int
   numar_coloane, int pozitie) {
    if (pozitie < 0 || pozitie >= numar_linii) {
        std::cout << "Pozitie invalida!\n";
        return matrice;
    }
```

```
    // Alocare matrice noua cu o linie mai putin
    int** matrice_noua = alocaMatrice(numar_linii - 1,
       numar_coloane);

    // Copiere elemente, sarind linia de sters
    for (int i = 0, k = 0; i < numar_linii; i++) {
        if (i == pozitie) continue;
        for (int j = 0; j < numar_coloane; j++) {
            matrice_noua[k][j] = matrice[i][j];
        }
        k++;
    }

    // Eliberare matrice veche
    elibereazaMatrice(matrice, numar_linii);

    numar_linii--;
    return matrice_noua;
}
```

Funcție de ștergere a unui coloane într-un tablou dinamic

```
int** stergereColoana(int** matrice, int numar_linii, int&
   numar_coloane, int pozitie) {
    if (pozitie < 0 || pozitie >= numar_coloane) {
        std::cout << "Pozitie invalida!\n";
        return matrice;
    }

    // Alocare matrice noua cu o coloana mai putin
    int** matrice_noua = alocaMatrice(numar_linii, numar_coloane -
       1);

    // Copiere elemente, sarind coloana de sters
    for (int i = 0; i < numar_linii; i++) {
        for (int j = 0, k = 0; j < numar_coloane; j++) {
            if (j == pozitie) continue;
            matrice_noua[i][k++] = matrice[i][j];
        }
    }
    // Eliberare matrice veche
    elibereazaMatrice(matrice, numar_linii);

    numar_coloane--;
    return matrice_noua;
}
```

Exemplu complet cu citire, inserare și ștergere

```
int main() {
    int numar_linii, numar_coloane;

    // Citire matrice
    int** matrice = alocare_matrice(numar_linii, numar_coloane);

    citireMatrice(matrice, numar_linii, numar_coloane);

    // Inserare linie noua
    int linie_noua[100]; // presupunem max 100 coloane
    for (int i = 0; i < numar_coloane; i++) {
        linie_noua[i] = i + 10;  // exemplu valori
    }
    matrice = inserareLinie(matrice, numar_linii, numar_coloane,
       1, linie_noua);

    // Inserare coloana noua
    int coloana_noua[100]; // presupunem max 100 linii
    for (int i = 0; i < numar_linii; i++) {
        coloana_noua[i] = i + 20;  // exemplu valori
    }
    matrice = inserareColoana(matrice, numar_linii, numar_coloane,
       2, coloana_noua);

    // Stergere linie
    matrice = stergereLinie(matrice, numar_linii, numar_coloane,
       0);

    // Stergere coloana
    matrice = stergereColoana(matrice, numar_linii, numar_coloane,
       numar_coloane - 1);

    // Afisare matrice finala
    afisareMatrice(matrice, numar_linii, numar_coloane);

    // Eliberare memorie
    elibereazaMatrice(matrice, numar_linii);

    return 0;
}
```

Observație! Pentru o alternativă modernă și flexibilă a tablourilor statice în cadrul bibliotecii standard C++, se recomandă consultarea clasei `std::vector`, prezentată în secțiunea de containere secvențiale (STL).

```
std::vector<std::vector<int>> matrice;
```

6.1.3 Matrice pătratice

O matrice pătratică este o matrice bidimensională cu același număr de linii și coloane, adică: numar_linii = numar_coloane = numar_linii=numar_coloane = n.

Sintaxă generală

```
tip_de_data nume_matrice[n][n];
```

Parametrii includ `tip_de_data`, tipul elementelor matricei (ex: `int`, `double`), `nume_matrice`, identificatorul matricei, și `n`, dimensiunea matricei, care reprezintă atât numărul de linii, cât și numărul de coloane.

Diagonala principală a unei matrici pătratice este formată din elementele situate pe pozițiile (i, i), pentru $i = 0, 1, \ldots, n-1$, unde n reprezintă dimensiunea matricei. Aceste elemente au proprietatea că indicele liniei este egal cu indicele coloanei.

Diagonala secundară este alcătuită din elementele situate pe pozițiile $(i, n-1-i)$, pentru $i = 0, 1, \ldots, n-1$. Aici, suma indicilor liniei și coloanei este constantă și egală cu $n-1$.

Zonele Nord, Sud, Est și Vest ale unei matrici pătratice

Pe lângă diagonale, o matrice pătratică de dimensiune $n \times n$ poate fi împărțită conceptual în patru zone distincte, adesea denumite Nord (N), Sud (S), Est (E) și Vest (V), în raport cu diagonala principală și/sau secundară. Aceste zone sunt utile în analiza și implementarea algoritmilor care necesită prelucrarea selectivă a elementelor matricei.

Pentru o matrice A de dimensiune $n \times n$, cu elemente $A_{i,j}$ (unde i este indicele liniei și j este indicele coloanei, ambele de la 0 la $n-1$):

1. **Zona de Nord (N): Elementele de deasupra diagonalei principale** Această zonă include toate elementele $A_{i,j}$ pentru care indicele liniei este strict mai mic decât indicele coloanei.

   - **Relație concludentă:** $i < j$

   Elementele din această zonă se află "deasupra" diagonalei principale.

2. **Zona de Sud (S): Elementele de sub diagonala principală** Această zonă cuprinde toate elementele $A_{i,j}$ pentru care indicele liniei este strict mai mare decât indicele coloanei.

   - **Relație concludentă:** $i > j$

   Elementele din această zonă se află "sub" diagonala principală.

3. **Zona de Est (E): Elementele de la dreapta diagonalei secundare** Această zonă este formată din elementele $A_{i,j}$ pentru care suma indicilor liniei și coloanei este strict mai mare decât $n - 1$.

   - **Relație concludentă:** $i + j > n - 1$

   Elementele din această zonă se află "la dreapta" diagonalei secundare.

4. **Zona de Vest (V): Elementele de la stânga diagonalei secundare** Această zonă include elementele $A_{i,j}$ pentru care suma indicilor liniei și coloanei este strict mai mică decât $n-1$.

   - **Relație concludentă:** $i + j < n - 1$

   Elementele din această zonă se află "la stânga" diagonalei secundare.

Observații importante:

- Diagonala principală și cea secundară împart matricea în aceste zone, iar elementele de pe diagonale nu fac parte din niciuna dintre aceste zone strict definite (N, S, E, V).
- O matrice poate fi împărțită în patru cadrane distincte prin intersecția diagonalei principale și a celei secundare, dar zonele N, S, E, V definite mai sus sunt mai generale și se referă la poziția relativă față de o singură diagonală.
- În funcție de problema specifică, aceste zone pot fi definite și incluzând diagonalele (ex: $i \leq j$ pentru "Zona de Nord, inclusiv diagonala principală").

6.2 Probleme rezolvate

În secțiunea care urmează propunem o serie de probleme cu tablouri unidimensionale. Problemele vor fi rezolvate atât cu alocare statică de memorie, cât și cu alocare dinamică de memorie, respectiv atât simplu, cât și cu STL.

1. Se citesc un număr natural n și o matrice pătratică cu n linii și n coloane. Să se calculeze suma elementelor de deasupra diagonalei principale.

Rezolvare:

Date de intrare:

- *Un număr natural* $n \in \mathbb{N}^*$, reprezentând dimensiunea matricei pătratice.
- *O matrice pătratică* cu n linii și n coloane, conținând numere întregi.

Date de ieșire: *Un număr întreg* — suma elementelor strict deasupra diagonalei principale.

Algoritm de rezolvare:

- Se citește dimensiunea n.
- Se alocă o matrice pătratică statică de dimensiune $n \times n$.
- Se citesc cele $n \times n$ elemente ale matricei.
- Se parcurge matricea și se adună toate elementele de pe pozițiile (i, j) pentru care $j > i$.
- Se afișează suma obținută.

Algoritm scris în C++:

```
#include <iostream>
const int numar_maxim = 100;

void citireMatrice(int matrice[numar_maxim][numar_maxim], int
   n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            std::cin >> matrice[i][j];

}

int calculSuma(int matrice[numar_maxim][numar_maxim], int n) {
    int suma = 0;
    for (int i = 0; i < n; i++)
        for (int j = i + 1; j < n; j++)
            suma += matrice[i][j];
    return suma;
}
```

```
int main() {
    int n;
    std::cout << "Introdu dimensiunea matricei: ";
    std::cin >> n;
    int matrice[numar_maxim][numar_maxim];
    citireMatrice(matrice, n);
    int suma = calculSuma(matrice, n);
    std::cout << "Suma elementelor deasupra diagonalei
       principale este: " << suma << "\n";
    return 0;
}
```

Justificarea soluției propuse: Algoritmul parcurge doar elementele de pe pozițiile (i, j) cu $j > i$, care corespund exact zonei strict deasupra diagonalei principale într-o matrice pătratică. Matricea este parcursă corect, fără a accesa indecși în afara limitelor, iar fiecare element este adăugat o singură dată.

Analiza complexității:

- *Complexitate temporală*: $\mathcal{O}(n^2)$ - în cel mai rău caz (matrice completă), dar zona deasupra diagonalei are $\frac{n*(n-1)}{2}$ elemente, deci mai eficient în practică.
- *Complexitate spațială*: $\mathcal{O}(n^2)$ - pentru stocarea matricei.

2. Se citesc două numere naturale m și n, precum și o matrice cu $m \times n$ elemente. Să se șteargă toate liniile care au toate elementele numere pare.

Rezolvare:
Date de intrare:

- *Două numere naturale* m și n, dimensiunile matricei.
- *O matrice* de dimensiune $m \times n$, conținând elemente întregi.

Date de ieșire: Matricea rezultată după eliminarea tuturor liniilor în care toate elementele sunt pare.

Algoritm de rezolvare:

- Se citesc valorile m și n.

- Se declară o matrice statică de tip `std::array<std::array<int, n>, m>`.
- Se citesc cele $m \times n$ elemente.
- Se parcurg liniile matricei.
- Pentru fiecare linie, se verifică dacă toate elementele sunt pare.
- Dacă o linie are doar numere pare, ea este ștearsă prin deplasarea liniilor de sub ea cu o poziție mai sus.
- Se actualizează numărul de linii.
- Se afișează matricea modificată.

Algoritm scris în C++:

```
#include <iostream>
#include <array>

const int numar_maxim_linii = 100;
const int numar_maxim_coloane = 100;

void citireMatrice(std::array<std::array<int,
   numar_maxim_coloane>, numar_maxim_linii>& matrice, int m,
   int n) {
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            std::cin >> matrice[i][j];
}

void afisareMatrice(const std::array<std::array<int,
   numar_maxim_coloane>, numar_maxim_linii>& matrice, int m,
   int n) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            std::cout << matrice[i][j] << " ";
        }
        std::cout << "\n";
    }
}

bool toatePare(const std::array<int, numar_maxim_coloane>&
   linie, int n) {
    for (int j = 0; j < n; j++) {
        if (linie[j] % 2 != 0)
            return false;
```

```
    }
    return true;
}

void stergeLiniiPare(std::array<std::array<int,
   numar_maxim_coloane>, numar_maxim_linii>& matrice, int& m,
   int n) {
    int noua_dim = 0;
    for (int i = 0; i < m; i++) {
        if (!toatePare(matrice[i], n)) {
            matrice[noua_dim++] = matrice[i];
        }
    }
    m = noua_dim;
}

int main() {
    int m, n;
    std::cout << "Introdu numarul de linii si coloane: ";
    std::cin >> m >> n;

    std::array<std::array<int, numar_maxim_coloane>,
       numar_maxim_linii> matrice;

    citireMatrice(matrice, m, n);
    stergeLiniiPare(matrice, m, n);

    std::cout << "Matricea dupa stergerea liniilor complet
       pare:\n";
    afisareMatrice(matrice, m, n);

    return 0;
}
```

Verificarea corectitudinii: Fiecare linie este verificată element cu element. Dacă toate sunt pare, linia este eliminată prin suprascrierea acesteia cu următoarea, și se ajustează dimensiunea logică a matricei. Nu sunt accesate indecși în afara limitelor.

Justificarea soluției propuse:

- *Complexitate temporală*: $\mathcal{O}(m \cdot n)$ în cel mai rău caz, deoarece fiecare linie este parcursă și pot apărea mutări de linii.
- *Complexitate spațială*: $\mathcal{O}(m \cdot n)$, ocupat de matricea `std::array`.

3. Se citesc două numere naturale m și n, precum și o matrice de $m \times n$ elemente. Să se dubleze toate coloanele care au toate elementele numere pare.

Rezolvare:

Date de intrare:

- *Două numere naturale* m, n — dimensiunile inițiale ale matricei.
- *O matrice de* $m \times n$ elemente întregi.

Date de ieșire: Matricea obținută prin duplicarea coloanelor formate doar din numere pare.

Algoritm de rezolvare:

- Se citesc valorile m și n, dimensiunile matricei.
- Se alocă dinamic o matrice cu m linii și n coloane.
- Se citesc valorile elementelor matricei de la tastatură.
- Se parcurge fiecare coloană a matricei.
- Pentru fiecare coloană:
  - Se verifică dacă toate elementele sunt pare.
  - Dacă da:
    - Se creează o copie a coloanei într-un vector nou.
    - Se inserează coloana nouă imediat după cea originală, folosind funcția `inserareColoana`.
    - Se actualizează numărul total de coloane.
    - Se avansează indexul curent cu 2 (pentru a evita reanalizarea copiei inserate).
  - Dacă nu, se trece la coloana următoare.
- Se afișează matricea obținută după dublarea coloanelor.
- Se eliberează memoria alocată dinamic pentru matrice.

Algoritm scris în C++ (funcțiile noi):

```
bool toateElementelePareColoana(int** matrice, int nr_linii,
    int col) {
    for (int i = 0; i < nr_linii; i++) {
        if (matrice[i][col] % 2 != 0) {
            return false;
        }
    }
    return true;
}

void dubleazaColoanePare(int**& matrice, int nr_linii, int&
    nr_coloane) {
    int col = 0;
    while (col < nr_coloane) {
        if (toateElementelePareColoana(matrice, nr_linii,
            col)) {
            // Construim coloana noua (copie a celei
                existente)
            int* coloana_noua = new int[nr_linii];
            for (int i = 0; i < nr_linii; i++) {
                coloana_noua[i] = matrice[i][col];
            }

            // Inseram copia imediat dupa coloana curenta
            matrice = inserareColoana(matrice, nr_linii,
                nr_coloane, col + 1, coloana_noua);

            delete[] coloana_noua;
            col += 2; // sarim peste coloana si copia sa
        } else {
            col++;
        }
    }
}

int main() {
    int m, n;
    int** matrice = citireMatrice(m, n);
    dubleazaColoanePare(matrice, m, n);
    std::cout << "Matricea dupa dublarea coloanelor complet
        pare:\n";
    afisareMatrice(matrice, m, n);
    eliberează Matrice(matrice, m);
    return 0;
}
```

Justificarea soluției propuse: Printr-un apel la funcția `toateElementelePareColoana` este testată corect fiecare coloană. Dacă pentru o coloană funcția returnează $true$, aceasta este duplicată și copia este inserată imediat după poziția curentă. Indexul este ajustat pentru a evita inserări multiple pe același set de date.

Analiza complexității:

- *Complexitate temporală*: $\mathcal{O}(m \cdot n^2)$ - deoarece pentru fiecare coloană (până la n) putem face o inserare ($\mathcal{O}(m + n)$).
- *Complexitate spațială*: $\mathcal{O}(m \cdot n)$ - matricea este extinsă prin inserări, dar în continuare stocată dinamic.

4. Se citesc trei numere naturale m, n și p, precum și două matrici de $m \times n$, respectiv $n \times p$ elemente. Să se înmulțească cele două matrici și să se afișeze matricea rezultată.

Rezolvare:
Date de intrare:

- Trei numere naturale m, n, p
- Două matrici:
  - A de dimensiune $m \times n$
  - B de dimensiune $n \times p$

Date de ieșire: Matricea produs $C = A \cdot B$, de dimensiune $m \times p$

Algoritm de rezolvare:

- Se citesc valorile m, n și p.
- Se declară matricea A de dimensiune $m \times n$ și matricea B de dimensiune $n \times p$, folosind `std::vector`.
- Se citește conținutul celor două matrici.
- Se creează o matrice rezultată C de dimensiune $m \times p$, inițializată cu 0.

- Se calculează produsul matricilor: pentru fiecare element $C[i][j]$, se parcurge $k = \overline{0, n-1}$ și se adună produsul $A[i][k] \cdot B[k][j]$.
- Se afișează matricea rezultată.

Algoritm scris în C++:

```
#include <iostream>
#include <vector>

using Matrice = std::vector<std::vector<int>>;

Matrice inmultesteMatricele(const Matrice& A, const Matrice&
   B, int m, int n, int p) {
    Matrice C(m, std::vector<int>(p, 0));
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < p; j++) {
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return C;
}

void citireMatrice(Matrice& M, int linii, int coloane) {
    for (int i = 0; i < linii; i++)
        for (int j = 0; j < coloane; j++)
            std::cin >> M[i][j];
}

void afisareMatrice(const Matrice& M) {
    for (const auto& linie : M) {
        for (int x : linie) {
            std::cout << x << " ";
        }
        std::cout << "\n";
    }
}

int main() {
    int m, n, p;
    std::cout << "Introduceti m, n, p: ";
    std::cin >> m >> n >> p;
    Matrice A(m, std::vector<int>(n));
    Matrice B(n, std::vector<int>(p));
    std::cout << "Cititi matricea A:\n";
```

```
    citireMatrice(A, m, n);
    std::cout << "Cititi matricea B:\n";
    citireMatrice(B, n, p);
    Matrice C = inmultesteMatricele(A, B, m, n, p);
    std::cout << "Rezultatul inmultirii:\n";
    afisareMatrice(C);
    return 0;
}
```

Justificarea soluției propuse: Se verifică dacă dimensiunile matricilor respectă condiția de înmulțire: numărul de coloane din A este egal cu numărul de linii din B. Algoritmul de înmulțire respectă formula matematică standard pentru calculul produsului a două matrici.

Analiza complexității:

- *Complexitate temporală*: $\mathcal{O}(m \cdot n \cdot p)$ - fiecare element din matricea rezultat implică o înmulțire și o adunare pentru fiecare valoare k.
- *Complexitate spațială*: $\mathcal{O}(m \cdot n + n \cdot p + m \cdot p)$ - pentru cele 3 matrici utilizate.

6.3 Probleme propuse - Matrici simple

*

1. Se citește o matrice pătratică de dimensiuni $m \times n$ cu elemente numere naturale. Să se calculeze și să se afișeze suma elementelor de pe diagonala principală.
   Exemplu: Matricea $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ are suma diagonală principală: 1 + 5 + 9 = 15
2. Se citește o matrice de dimensiuni $m \times n$ cu elemente numere naturale. Să se afișeze numărul de elemente pare din matrice.
   Exemplu: Matricea $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ are 3 elemente pare: 2, 4, 6
3. Se citește o matrice pătratică de dimensiuni $n \times n$ cu elemente numere naturale. Să se afișeze elementul maxim și elementul minim din întreaga matrice.

Exemplu: Matricea $\begin{bmatrix} 4 & 9 & 2 \\ 7 & 5 & 1 \\ 3 & 6 & 8 \end{bmatrix}$ are elementul maxim 9 și elementul minim 1

**

1. Se citește o matrice pătratica de dimensiuni $n \times n$, având elementele distincte pe fiecare dintre cele două diagonale. Interschimbați elementul maxim de pe diagonala principală cu elementul minim de pe diagonala secundară. Afișați matricea rezultată.
Exemplu: Pentru matricea $\begin{bmatrix} 5 & 8 & 12 \\ 3 & 9 & 4 \\ 7 & 2 & 6 \end{bmatrix}$ devine $\begin{bmatrix} 5 & 8 & 12 \\ 3 & 7 & 4 \\ 9 & 2 & 6 \end{bmatrix}$
2. Se citește o matrice pătratica cu n linii și n coloane, cu elemente numere naturale. Să se determine sumele elementelor celor 4 triunghiuri determinate de diagonala principală și cea secundară.
Exemplu: Matricea $\begin{bmatrix} 5 & 8 & 12 & 3 & 1 \\ 4 & 9 & 6 & 2 & 0 \\ 7 & 1 & 11 & 10 & 2 \\ 15 & 14 & 13 & 16 & 1 \\ 1 & 2 & 22 & 2 & 4 \end{bmatrix}$ are sumele:
   - Deasupra diagonalei principale și deasupra diagonalei secundare: 8 + 12 + 3 + 6 = 29
   - Deasupra diagonalei principale și dedesubtul diagonalei secundare: 0 + 10 + 2 + 1 = 13
   - Dedesubtul diagonalei principale și deasupra diagonalei secundare: 4 + 7 + 1 + 15 = 27
   - Dedesubtul diagonalei principale și Dedesubtul diagonalei secundare: 13 + 2 + 22 + 2 = 39

3. Construiți și afișați o matrice pătratica de ordin n după modelul de mai jos pentru care $n = 5$:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 2 & 3 & 4 \\ 3 & 2 & 1 & 2 & 3 \\ 4 & 3 & 2 & 1 & 2 \\ 5 & 4 & 3 & 2 & 1 \end{bmatrix}$$

4. Se citește o matrice pătratică de dimensiuni $n \times n$ cu elemente numere naturale distincte. Să se înlocuiască toate elementele pare din matrice cu valoarea maximă impară aflată în matrice.

Exemplu: Matricea $\begin{bmatrix} 4 & 9 & 2 \\ 7 & 5 & 1 \\ 3 & 6 & 8 \end{bmatrix}$ are valoare impară maximă 9 și devine $\begin{bmatrix} 9 & 9 & 9 \\ 7 & 5 & 1 \\ 3 & 9 & 9 \end{bmatrix}$

***

1. Se citește o matrice cu m linii și n coloane cu elemente numere naturale. Afișați numărul de elemente distincte de pe linii. Să se șteargă după aceea liniile cu numărul maxim identificat.

Exemplu: Matricea $\begin{bmatrix} 5 & 8 & 12 \\ 3 & 9 & 3 \\ 5 & 5 & 5 \\ 8 & 3 & 8 \end{bmatrix}$ are pe fiecare linie 3, 2, 1, 2 elemente distincte și devine

$\begin{bmatrix} 3 & 9 & 3 \\ 5 & 5 & 5 \\ 8 & 3 & 8 \end{bmatrix}$

2. Se citește o matrice pătratica de dimensiuni $n \times n$, cu elemente numere naturale. Ordonați crescător elementele de pe diagonala principală prin interschimbări de linii și coloane.

Exemplu: Matricea $\begin{bmatrix} 7 & 12 & 5 & 9 \\ 3 & 8 & 6 & 2 \\ 4 & 1 & 9 & 0 \\ 8 & 5 & 3 & 4 \end{bmatrix}$ devine $\begin{bmatrix} 4 & 8 & 5 & 3 \\ 9 & 7 & 12 & 5 \\ 2 & 3 & 8 & 6 \\ 0 & 4 & 1 & 9 \end{bmatrix}$

3. Se citește o matrice pătratică de dimensiuni $n \times n$ cu elemente numere naturale. Să se interschimbe elementele simetrice față de diagonala principală care au aceeași paritate.

Exemplu: Matricea $\begin{bmatrix} 3 & 4 & 1 & 6 \\ 3 & 4 & 2 & 1 \\ 5 & 6 & 5 & 7 \\ 2 & 4 & 3 & 6 \end{bmatrix}$ devine $\begin{bmatrix} 3 & 4 & 5 & 2 \\ 3 & 4 & 6 & 1 \\ 1 & 2 & 5 & 3 \\ 6 & 4 & 7 & 6 \end{bmatrix}$

****

1. Să se rotească o matrice pătratică, cu n linii și n coloane, cu 90 de grade în sensul acelor de ceas.

Exemplu: Matricea $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$ devine $\begin{bmatrix} 9 & 5 & 1 \\ 10 & 6 & 2 \\ 11 & 7 & 3 \\ 12 & 8 & 4 \end{bmatrix}$

2. Se citește o matrice pătratică de ordin n. Parcurgeți și afișați elementele din matrice începând cu elementul din colțul stânga sus, mergând paralel cu diagonala secundară.

Exemplu: Matricea $\begin{bmatrix} 1 & 3 & 4 & 10 \\ 2 & 5 & 9 & 11 \\ 6 & 8 & 12 & 15 \\ 7 & 13 & 14 & 16 \end{bmatrix}$ se parcurge 1, 2, 3 ..., 14, 15, 16

3. Se dă o matrice pătratică de dimensiuni $n \times n$, care trebuie să conțină toate numerele întregi între 0 și $n^2 - 1$.

   (a) Să se verifice dacă matricea dată conține toate numerele între 0 și $n^2 - 1$ exact o dată (fără duplicate și fără lipsuri).

   (b) Dacă matricea este validă, se va localiza valoarea 0 și se vor genera maxim 4 noi matrice, mutate prin schimburi cu vecinii săi (Nord, Sud, Est, Vest).

Exemplu: Matricea $\begin{bmatrix} 1 & 3 & 4 & 10 \\ 2 & 5 & 9 & 11 \\ 6 & 8 & 12 & 15 \\ 7 & 13 & 14 & \boxed{0} \end{bmatrix}$ are $\begin{bmatrix} 1 & 3 & 4 & 10 \\ 2 & 5 & 9 & 11 \\ 6 & 8 & 12 & \boxed{0} \\ 7 & 13 & 14 & 15 \end{bmatrix}$ și $\begin{bmatrix} 1 & 3 & 4 & 10 \\ 2 & 5 & 9 & 11 \\ 6 & 8 & 12 & 15 \\ 7 & 13 & \boxed{0} & 14 \end{bmatrix}$

*****

1. Se citesc un număr natural n pătrat perfect și apoi n numere naturale. Să se creeze o matrice pătratică care să conțină toate cele n numere citite, completate:

   (a) În spirală în sens invers acelor de ceas (trigonometric), începând din colțul stânga sus.

   (b) În spirală în sensul acelor de ceas, începând din centru (doar pentru n impar).

   Exemplul (a): $\begin{bmatrix} 1 & 8 & 7 \\ 2 & 9 & 6 \\ 3 & 4 & 5 \end{bmatrix}$ - **Exemplul** (b): $\begin{bmatrix} 5 & 4 & 3 \\ 6 & 1 & 2 \\ 7 & 8 & 9 \end{bmatrix}$

2. O parcare ocupă o suprafață dreptunghiulară, având laturile orientate pe direcțiile Nord-Sud, respectiv Est-Vest. Parcarea poate fi reprezentată sub forma unui tablou cu mn celule, așezate pe m linii și n coloane, în care o celulă poate reprezenta: 0 pentru o celulă ce face parte dintr-un drum, 1 pentru un loc de parcare ocupat și -1 pentru un loc de parcare neocupat

$$\begin{bmatrix} 0 & 1 & -1 & 0 & 1 & 0 & -1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & -1 & 1 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & -1 & 1 & 0 \\ -1 & 0 & 1 & 0 & 1 & 0 & -1 \end{bmatrix}$$

   (a) Dacă parcarea are două drumuri principale unul care o taie la mijloc pe direcția Nord-Sud, iar celălalt pe direcția Est-Vest, acestea determină în tablou patru zone dreptunghiulare. Să se determine care dintre aceste zone este mai ocupată de autoturisme.

(b) Implementându-se o **funcție**, să se verifice dacă există în parcare o zonă dreptunghiulară (delimitată de drumuri) care are toate locurile neocupate.

3. O suprafață de teren de păduri și pășuni are dimensiunile $m \times n$ și conține: 0 - zonă de pășune și un număr întreg pozitiv - tip de copac. Pădurile sunt zone dreptunghiulare formate din același tip de copac, separate între ele.

Exemplu:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 4 & 0 & 0 & 2 & 2 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 2 & 0 & 0 & 3 & 0 \\ 0 & 3 & 0 & 4 & 4 & 0 & 2 & 2 & 0 & 0 & 3 & 0 \\ 0 & 3 & 0 & 4 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 2 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 2 & 2 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(a) Identificați pășunile care pot uni două păduri de același tip dacă se plantează tipul respectiv.

Exemplu: Pășunea de la poziția (2, 3) poate uni două păduri de tipul 4.

(b) Identificați dimensiunea și tipul pădurii celei mai mari.

(c) Să se afișeze tipurile de copaci sortate descrescător după dimensiunea totală a pădurilor corespunzătoare. În acest exemplu: 2, 1, 4, 3 cu dimensiunile 10, 8, 6, 5

6.4 Probleme propuse - Matrici cu structuri / obiecte

*

1. În cadrul unei florării se folosesc buchete dispuse într-un aranjament floral cu formă bidimensională. Fiecare buchet este o identificat prin $nume$ (șir de caractere), care reprezintă denumirea buchetului și $pret$ (număr real), care reprezintă prețul în lei. Parcurge toate

buchetele din aranjament și mărește cu 10% prețul tuturor buchetelor care au prețul mai mic de 50 lei.

2. Se folosesc tricouri care urmează să fie aranjate într-un depozit de m rânduri și n coloane, unde rândul i reprezintă mărimea (de la 30 la 50) și coloana j reprezintă stocul (de la 0 la 100). Fiecare tricou este o structură identificată prin $marime$ (șir de caractere), care reprezintă mărimea tricoului și $stoc$ (număr întreg), care reprezintă numărul de tricouri disponibile. Se citește o listă de tricouri care se salvează în depozit. Să se identifice toate rândurile și toate coloanele unde nu se află tricouri. Să se afișeze coloana cu cel mai mare stoc total de tricouri din tot depozitul.

**

1. Se consideră o pădure dreptunghiulară reprezentată printr-un aranjament bidimensional, în care fiecare element poate fi fie un copac, fie un loc gol. Fiecare copac este identificat prin $specie$ (șir de caractere), care reprezintă tipul copacului, și $inaltime$ (număr real), care reprezintă înălțimea copacului în metri. Parcurge pădurea și afișează înălțimea tuturor copacilor care nu au niciun vecin direct (sus, jos, stânga, dreapta) ocupat de alt copac, ordonând valorile în mod descrescător.

2. Se consideră o bibliotecă de filme organizată pe mai multe rafturi, fiecare raft având același număr de filme. Fiecare film este identificat prin $titlu$ (șir de caractere), care reprezintă denumirea filmului, și $categorie$ (șir de caractere), care reprezintă genul filmului (ex: „Comedie”, „Acțiune”, „Dramă”). Să se sorteze filmele de pe fiecare raft în ordine alfabetică după titlu și să se verifice dacă există vreo coloană verticală în bibliotecă în care toate filmele au aceeași categorie după sortare.

***

1. Se consideră un puzzle format din piese aranjate într-un dreptunghi, unde fiecare piesă este identificată printr-o culoare reprezentată ca șir de caractere. Puzzle-ul poate conține piese de culori diferite. Să se determine poziția și dimensiunile celui mai mare dreptunghi format exclusiv din piese de aceeași culoare, specificând culoarea respectivă, coordonata colțului din stânga-sus și numărul de linii și coloane ale dreptunghiului găsit.

2. Se consideră o hartă dreptunghiulară a unui oraș. În fiecare poziție se poate afla fie o casă, fie un loc gol fără construcție. O casă este descrisă printr-o structură ce conține $numar_locuitori$ (număr întreg), reprezentând câți oameni locuiesc în acea casă, și $strada$ (șir de caractere), reprezentând strada pe care se află casa.

   (a) Să se determine numărul total de străzi distincte din oraș, luând în considerare doar străzile pe care există cel puțin o casă

   (b) Să se determine numărul de locuitori izolați, adică persoane care trăiesc în case ce nu au nicio casă vecină pe pozițiile adiacente direct (sus, jos, stânga, dreapta)

   (c) Să se determine strada cu cel mai mic număr total de locuitori, obținut prin însumarea locuitorilor tuturor caselor aflate pe aceeași stradă.

****

1. Se consideră un teren minier dreptunghiular reprezentat printr-un aranjament bidimensional. În fiecare poziție se află fie un zăcământ, fie un loc fără zăcământ. Un zăcământ este descris printr-o structură ce conține denumire (șir de caractere), reprezentând numele zăcământului, și cantitate (număr întreg), reprezentând cantitatea disponibilă în acea poziție.

   (a) Să se afișeze denumirile zăcămintelor ordonate descrescător după cantitatea totală, obținută prin însumarea cantităților tuturor aparițiilor aceluiași zăcământ din teren

   (b) Să se parcurgă terenul în spirală, din exterior spre interior, în ordinea dreapta, jos, stânga, sus, repetând pașii, și să se afișeze detaliile zăcămintelor aflate pe pozițiile prime din această ordine de parcurgere.

*****

1. Se consideră un ecran de televizor reprezentat printr-un aranjament bidimensional de pixeli. Fiecare pixel este o structură identificată prin trei valori întregi: $rosu$, $verde$ și $albastru$, fiecare cu valori între 0 și 255, reprezentând intensitățile componentelor de culoare.

(a) Să se verifice dacă toți pixelii ecranului au cel puțin o componentă de culoare lipsă, adică una dintre valorile roșu, verde sau albastru este egală cu 0

(b) Să se rotească ecranul cu 90 de grade spre dreapta și să se afișeze pixelii rezultați după rotație

(c) Considerând intensitatea unui pixel definită ca media aritmetică a valorilor roșu, verde și albastru, să se afișeze poziția pixelului cu intensitatea maximă

(d) Să se caute și să se afișeze poziția unui pixel cu intensitatea X, unde X este o valoare citită de la tastatură.

Capitolul 7

Structuri de date elementare

7.1 Aspecte teoretice

În prima partea a acestui capitol vor fi discutate listele simplu și dublu înlănțuite. Vom reaminti pe scurt principalele probleme teoretice, modul de funcționare a listelor, algoritmii pentru operațiile principale și listele în biblioteca STL. În continuare se vor discuta stivele și cozile, care pot fi privite de fapt ca niște adaptoare de containere, în sensul că, au la bază un container (structură de date) elementar, precum vector sau listă, pe car îl adaptează pentru o funcționalitate specifică[2]. De asemenea vom introduce și containerele STL specifice pentru stive și cozi.

7.1.1 Liste înlănțuite

Listele înlănțuite sunt structuri de date organizate secvențial, în care fiecare element are un element ce îl precede, cu excepția primului element, și unul care îl urmează, cu excepția ultimului element. Spre deosebire de vectori, în care elementele sunt stocate în zone de memorie alăturare, în cazul listelor, deși trecerea de la un element la altul se realizează secvențial, elementele acestora nu sunt neapărat aranjate în memorie secvențial. Dacă la vector trebuie alocată o zonă de memorie compactă pentru stocarea elementelor, în cazul listelor, de fiecare dată când se adaugă un element nou se alocă memorie doar pentru acesta, iar la eliminarea lui din listă, memoria trebuie eliberată.

Accesul la elementele unei liste se face pornind din capul listei, iar trecerea de la elementul curent la următorul se face pe baza adresei de memorie a următorului element, care este stocată în structura asociată elementului curent.

Există două tipuri mari de liste înlănțuite:

(1) Listele simplu înlănțuite, în care fiecare element este o structură de tip $node$ cu două câmpuri: key - pentru informație și $next$ - pointer către următorul element din listă.

(2) Listele dublu înlănțuite, în care fiecare element este o structură de tip $node$ cu trei câmpuri. La câmpurile key și $next$ se adaugă câmpul $prev$ - pointer către elementul precedent din listă.

Operațiile de bază pe o listă înlănțuită sunt: adăugarea unui element, eliminarea unui element, precum și căutarea unui element.

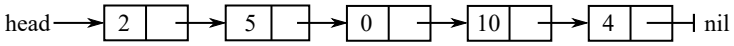
Pentru implementarea unei liste vom considera o structură cu mai multe câmpuri, printre care un câmp $head$ de tip pointer la $node$, care va stoca adresa primului element din listă, un câmp $tail$ tot de tip pointer la $node$, care va stoca adresa ultimului element din listă (acest câmp va fi considerat deocamdată doar în cazul listelor dublu înlănțuite) și un câmp $size$ de tip int, care va memora numărul de elemente din listă. De asemenea în cadrul structurii vor fi implementate și funcțiile corespunzătoare operațiilor pe liste.

Am păstrat denumirea în limba engleză a câmpurilor structurilor, ca să se alinieze cu denumirile din STL.

Există diferite moduri de implementare a funcționalităților unei liste. În acest capitol vom considera implementări similare celor pentru `std::forward_list`, respectiv `std::list` din biblioteca STL.

Liste simplu înlănțuite

Un exemplu de listă simplu înlănțuită este cel din figură:



Structura $node$ corespunzătoare unei astfel de liste, care stochează numere întregi, este:

```
struct node{
    int key;
    node* next = nullptr;
}
```

Desigur, dacă dorim să generalizăm tipul stocat, putem folosi un nod de tip *template*, similar ca pentru funcții, așa cum a fost descris în capitolul 4.

Vom considera pentru implementarea listei simplu înlănțuite structura

```
struct lista_simplu{
    node* head = nullptr; //pointer catre primul element din lista
    int size = 0;
```

```
    //Aici vor urma toate functiile atasate listei
    void push_front(int key){
    //.... codul asociat
    }

    void pop_front(){
    //.... codul asociat
    }
    // .... etc
}
```

În continuare sunt prezentați algoritmii în pseudocod asociați operațiilor de adăugare și de extragere a elementului de la începutul listei, precum și de căutare a unui element într-o listă simplu înlănțuită,.

Algorithm 4: PUSH_FRONT

Input: O listă L, o valoare key care se adaugă la începutul listei.

begin

alocă memorie pentru y

$y.key \leftarrow key$

$y.next \leftarrow L.head$

$L.head \leftarrow y$

$size \leftarrow size + 1$

end

Algorithm 5: POP_FRONT

Input: O listă L

begin

if $L.head \neq nil$ **then**

$y \leftarrow L.head$

$L.head \leftarrow L.head.next$

$size \leftarrow size - 1$

eliberează memoria pentru y

return

else

return *"underflow error"*

end

end

Complexitatea de timp a celor doi algoritmi de adăugare și respectiv ștergere a unui element la începutul listei este constantă.

Algorithm 6: FIND

Input: O listă L, cheia căutată key

begin

$x \leftarrow L.head$

while $x \neq nil$ *AND* $x.key \neq key$ **do**

$x \leftarrow x.next$

end

return x

end

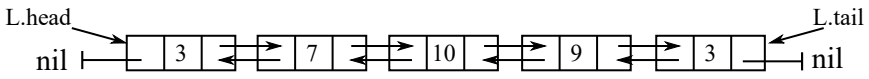
Funcția corespunzătoare algoritmului $FIND$ returnează un pointer către nodul care conține cheia căutată sau pointerul nul.

Complexitatea de timp a algoritmului de căutare a unei chei este aceeași ca pentru căutarea secvențială și este în cazul mediu liniară în n, unde n este numărul de elemente din listă.

Alți algoritmi specifici listelor sunt propuși spre rezolvare la secțiunea de probleme.

Liste dublu înlănțuite

Un exemplu de listă dublu înlănțuită este cel din figură:



Structura $node$ corespunzătoare unei astfel de liste, care stochează numere întregi, este:

```
struct node{
    int key;
    node* prev = nullptr;
    node* next = nullptr;
}
```

Pentru implementarea unei astfel de liste se poate folosi o structură $Lista$, similară cu cea de la listele simplu înlănțuite, în care algoritmii sunt adaptați corespunzător.

```
struct Lista{
    node* head = nullptr; //pointer catre primul element din lista
    node* tail = nullptr; //pointer catre ultimul element din lista
```

```
    int size = 0;

    //Aici vor urma toate functiile atasate listei
    void push_front(int key){
    //.... codul asociat
    }
    void pop_front()
    {
    //.... codul asociat
    }
    // .... etc
}
```

Algoritmul de căutarea unui element într-o listă dublu înlănțuită este identic cu cel de căutare într-o listă simplu înlănțuită. Pentru adăugare și ștergere a unui element la începutul listei, algoritmii adaptați, cu complexitatea de timp constantă sunt:

Algorithm 7: PUSH_FRONT

Input: O listă L, o valoare key care se adaugă la începutul listei.

begin

alocă memorie pentru y

$y.key \leftarrow key$

$y.prev \leftarrow nil$//poate fi omis în general, pentru că în structura $node$ se inițializează acest //câmp automat

$y.next \leftarrow L.head$

if $L.head = nil$ **then**

//Ultimul element va indica și el către noul element introdus

$L.tail \leftarrow y$

else

$L.head.prev \leftarrow y$

end

$L.head \leftarrow y$

$size \leftarrow size + 1$

end

Algorithm 8: POP_FRONT

Input: O listă L

begin

  if $L.head \neq nil$ **then**

    $y \leftarrow L.head$

    $L.head \leftarrow L.head.next$

    $size \leftarrow size - 1$

    eliberează memoria pentru y

    if $L.head = nil$ **then**

      $L.tail \leftarrow nil$

    else

      $L.head.prev = nil$

    end

  else

    return *"underflow error"*

  end

end

7.1.2 Liste în STL

Biblioteca STL oferă suport pentru cele două tipuri de liste:

- lista simplu înlănțuită:

```
template<class T, class Alloc = allocator<T>>class
   forward_list;
```

- lista dublu înlănțuită:

```
template<class T, class Alloc = allocator<T>>class list;
```

Clasele *list* și *forword_list* dispun de o serie de funcții care au aceeași semnătură și același scop, dar cu implementare diferită, în funcție de tipul fiecărei liste. În continuare vor fi prezentate aceste funcționalități pentru listele dublu înlănțuite.

Listele dispun de o serie de constructori, care permit declararea și inițializarea obiectelor utilizând una dintre următoarele maniere (în același mod și pentru `std::forward_list`):

```
//lista vida
```

```
std::list<int>first;

//lista ce contine 5 elemente cu valoarea 20
std::list<int>second(5, 20);

//lista ce contine elementele {1, 2, 3}
std::list<int>third = { 1, 2, 3 };

//lista ce contine elementele lui second
std::list<int>fourth(second.begin(), second.end());

//lista copie a lui third
std::list<int>fifth(third);
```

Exemplu de inserare de elemente și de parcurgere într-o structură de tip `std::list`.

```
#include<iostream>
#include<list>

int main(){
    std::list<int>lista;
    int num = 5, elem;
    for (int index = 0; index < num; index++)
    {
        std::cin >> elem;
        lista.push_front(elem);
        lista.push_back(elem);
    }
    std::list<int>::iterator it = lista.begin();
    for ( ; it != lista.end(); it++)
        std::cout << *it << " ";
    std::cout << std::endl;
    //sau
    for (int elem : lista)
        std::cout << elem << " ";
    std::cout << std::endl;
    return 0;
}
```

În exemplul de mai sus observăm mai multe lucruri:

- Putem parcurge lista folosind un iterator de tip

  ```
  std::list<int>::iterator it;
  ```

  a cărui declarare poate fi simplificată prin:

  ```
  auto it = lista.begin();
  ```

În cazul parcurgerii cu iteratori, variabila it este un obiect care se comportă asemănător unui pointer (are supraîncărcați operatorii $*$ și $++$), iar pentru a accesa elementul din listă către care "pointează" iteratorul, se folosește operatorul $*$.

- Putem parcurge cu operatorul :, precum în cazul instrucțiunii:

```
for (int elem : lista){...}
```

În acest caz variabila $elem$ va conține pe rând copii ale fiecărui element din listă. Pentru eficientizare, s-ar putea renunța la efectuarea unor copii, folosind operatorul de referențiere $\&$.

- Varianta cu iteratori ne permite să parcurgem un număr limitat de elemente, pe când varianta cu : parcurge toate elementele listei.

Alte operații pe containerul `std::list`:

- `insert(it, value)` sau `insert(it, nr, value)`
  - are ca parametrii un iterator, o valoare și, opțional, numărul de copii ale valorii, care vor fi inserate;
  - inserează valoarea (de nr ori) înainte de elementul indicat prin iterator.
- `remove(value)`
  - are ca parametru o valoare;
  - elimină toate aparițiile valorii din listă.
- `erase(it)` sau `erase(it1, it2)`
  - are ca parametri unul sau doi iteratori;
  - elimină din listă elementul indicat de iterator (dacă are un singur parametru), elementele începând de la primul iterator până la cel indicat de al doilea (exclusiv pe cel indicat de al doilea parametru).

Observație: atunci când într-o listă se inserează obiecte (de exemplu un element de tip `std::pair`), mai întâi trebuie creat obiectul, pentru a putea fi adăugat cu `insert`. Alternativ poate fi folosită metoda `emplace`.

Exemplu: de inserare într-o structură

```
std::list<std::pair<int, char>> myList;
```

folosind $insert$

```
    auto it = myList.begin();
    std::pair<int, char> p(10, 'a');
    myList.insert(it2, p);
```

versus $emplace$

```
    myList.emplace(it, 3, 'a');
```

Trebuie menționat și faptul că, pentru `forward_list` și `list` NU este implementată o funcție de căutare de genul FIND, ci căutarea se face în mod explicit de către programator, parcurgând cu iteratori lista sau folosind funcția `std::find` în modul următor:

```
auto it = std::find(lista.begin(), lista.end(), elem);
```

Variabila it în acest exemplu este un iterator care indică elementul cu cheia $elem$ din listă, dacă există, sau $lista.end()$ în caz contrar.

7.1.3 Stive și cozi

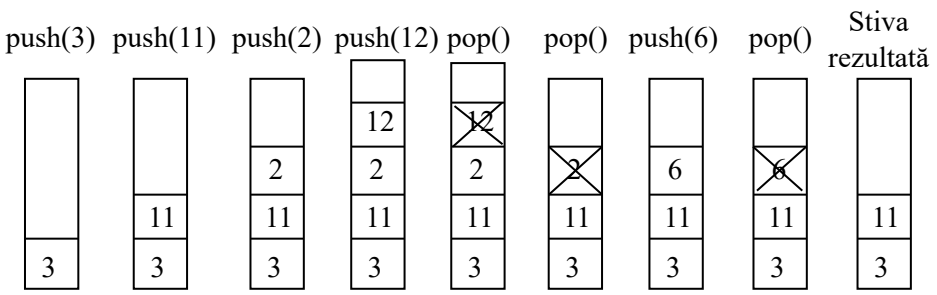
Stivele și cozile, așa cum s-a menționat, sunt adaptoare de containere. Ele au la bază o structură (container), ca de exemplu un vector sau o listă, a cărui funcționare este adaptată proprietăților stivei, respectiv cozii.

Stiva

O stivă este o structură de date secvențială, în care accesul la elemente se face doar la un capăt al acesteia, capăt numit și *vârf / top*. Un element nou care se adaugă la stivă, va fi adăugat în vârf, după ultimul element adăugat anterior, iar un element se extrage doar din vârf. Deci întotdeauna se procesează ultimul element adăugat la stivă, iar elementele sunt accesate în ordinea inversă inserției lor.

O structură de tip stiva trebuie să dispună de o variabilă, care indică elementul din vârf. Pentru adăugarea unui element nou se va folosi o funcție de tip $PUSH$, iar pentru a extrage elementul din vârful stivei, se va folosi o funcție de tip POP. Pentru accesul elementului din vârful stivei există o funcție de TOP.

Exemplu. Considerăm o stivă S, inițial vidă, în care se inserează numere întregi. Dacă asupra stivei se efectuează operațiile PUSH(3), PUSH(11), PUSH(2), PUSH(12), POP(), POP(), PUSH(6), POP(), atunci stiva la fiecare moment va arăta ca în figură:



Containere potrivite pentru stivă sunt vectorii și listele înlănțuite, dar și $deque$, care este un container secvențial, similar unui vector, dar care permite adăugarea și ștergerea la ambele capete în complexitate constantă de timp, precum și accesul prin poziție. Mai multe detalii pot fi găsite în [8].

Coada

O coadă este un container secvențial în care accesul se realizează la ambele capete și anume, la un capăt se realizează inserția și la celălalt ștergerea de elemente. Întotdeauna elementele vor fi accesate în ordinea în care au fost inserate. Cozile modelează, așa cum le spune și numele, procese ce presupun formarea de cozi, de exemplu deservirea unor clienți la un ghișeu.

Spre deosebire de stivă, pentru o coadă sunt necesare două variabile, una care să păstreze accesul la primul element și una care să păstreze accesul la ultimul element din coadă. De asemenea sunt necesare funcții de tip $PUSH$ și POP. În plus, pentru accesul la cele două capete ale cozii sunt necesare două funcții specifice de acces.

Observație: vectorii sunt mai puțin potriviți în mod direct pentru implementarea unei cozi. În materialul de curs este prezentată problema migrării către dreapta a cozii, în urma operațiilor de $PUSH$ și POP. Din acest motiv, în cazul folosirii unui vector drept container de bază, se recomandă să fie tratat în mod *circular* [3], [8].

În schimb pot fi folosite cu succes containere de tip `std::list` sau `std::deque`, cu observația că `std::deque` este în general mai eficient, din motive de management al memoriei. Pentru detalii asupra containerului `std::deque`, recomandăm [8].

7.1.4 Stive și cozi în STL

Pentru deque, stive și cozi există containere șablon (template) specifice în STL.

- **deque**:

```
template < class T, class Alloc = allocator<T> >
   class deque;
```

- **stiva**:

```
template <class T, class Container = deque<T> > class
   stack;
```

- **coada**:

```
template <class T, class Container = deque<T> > class
   queue;
```

Observație: atât pentru `std::deque`, `std::stack` și `std::queue` trebuie indicat la declarare, ce tip vor avea datele stocate în containerele respective. În plus, pentru `std::stack` și `std::queue` trebuie specificat containerul de bază, care în mod predefinit, este setat ca fiind `std::deque`.

Containerul `std::stack` dispune de funcțiile:

- **empty** - verifică dacă stiva este vidă;
- **size** - returnează numărul de elemente din stivă;
- **top** - returnează elementul din vârful stivei;
- **push** - pune un element în vârful stivei;
- **pop** - elimină elementul din vârful stivei.

Containerul `std::queue` dispune de funcțiile:

- **empty** - verifică dacă coada este vidă;
- **size** - returnează numărul de elemente din coadă;
- **front** - returnează elementul de la începutul cozii;
- **back** - returnează elementul de la sfârșitul cozii;
- **push** - pune un element după ultimul element al cozii;
- **pop** - elimină primul elementul din coadă.

7.2 Probleme rezolvate

1. **Implementarea unei liste.** Să se implementeze o listă dublu înlănțuită cu funcționalitățile descrise în continuare.

   Se cere utilizarea unei structuri $node$ care are trei câmpuri: un câmp pentru informație (de tip int) și două câmpuri de tip pointer la $node$ pentru legăturile către elementele precedent și următor.

   Se cere utilizarea unei structuri $List$ care are ca membri două variabile de tip $node*$ reprezentând adresele primului respectiv ultimului element din listă, o variabilă de tip int reprezentând numărul de elemente din listă și funcțiile:

   - $push_front(int\ key)$ - adaugă cheia key în capul listei;
   - $push_back(int\ key)$ - adaugă cheia key la finalul listei;
   - $pop_front()$ - șterge primul element din listă;
   - $pop_back()$ - șterge ultimul element din listă;
   - $find(int\ key)$ - caută o cheie key în listă - returnează pointer la nodul cu cheia key sau $nullptr$;
   - $erase(node * Nod)$ - șterge un element Nod din listă (NU implică căutare). Nodul transmis ca parametru a fost în prealabil găsit cu $find$ sau identificat prin parcurgerea listei;
   - $remove(int\ key)$ - șterge toate aparițiile cheii key (implică parcurgerea listei);
   - $empty()$ - verifică dacă lista e vidă;
   - $clear()$ - golește lista;
   - funcția $print()$ - afișează elementele listei.

   De asemenea $List$ trebuie să dispună de un constructor, care inițializează câmpurile și, preferabil, de un destructor, care să elibereze memoria.

   Testați structura definită în funcția $main$ în modul următor. Declarați o variabilă de tipul $List$, apoi definiți un meniu pe baza unei instrucțiuni $switch$ cu opțiuni pentru fiecare dintre metodele definite, iar apoi testați fiecare funcționalitate.

Rezolvare:

Vor fi prezentate doar implementările pentru funcțiile $push_front$, pop_front, $find$ și $print$. Restul funcțiilor sunt propuse spre rezolvare în secțiunea următoare.

Date de intrare: Se cere definirea unei structuri de tip $List$ cu o serie de funcționalități. Dintre acestea vor fi date ca rezolvare doar $push_front$, pop_front, $find$ și $print$. Apoi se cere testarea funcționalităților în $main$.
Date de ieșire: Structura $List$.

Algoritm de rezolvare: Se vor utiliza algoritmii în pseudocod și modelele de structuri din partea de teorie. Observăm faptul că, datorită plasării funcțiilor în interiorul structurii, nu mai este necesar să transmitem ca parametru către acestea lista, deci algoritmii din pseudocod trebuie adaptați în consecință. Acest lucru este ilustrat în rezolvarea C++.

Algoritm scris în C++:
Structura $List$.

```
#include<iostream>
#include<stdexcept> //pentru tratarea exceptiilor
struct List {
    struct node {
        int key;
        node* prev;
        node* next;
        node(int key = 0, node* prev = nullptr, node* next =
           nullptr) :key(key), prev(prev), next(next) {}
    };
    node* head;
    node* tail;
    int size;

    List() : head(nullptr), tail(nullptr), size(0) {}
    ~List(){
        while (head){
            node* y = head;
            head = head->next;
            delete y;
        }
    }
```

```
    void push_front(int value) {
        node* newNode = new node(value, nullptr, head);
        if (head)
            head->prev = newNode;
        else
            tail = newNode;
        head = newNode;
        size++;
    }

    void pop_front() {
        if (head)
        {
            node* temp = head;
            head = head->next;
            if (!head)
                tail = nullptr;
            else
                head->prev = nullptr;
            delete temp;
            size--;
        }
        else {
            throw std::underflow_error("Nu putem sterge din
               lista vida");
        }
    }

    node* find(int val) {
        node* current = head;
        for (; current != nullptr && current->key != val;
           current = current->next);
        return current;
    }

    void print() {
        for (auto el = head; el != nullptr; el = el->next)
            std::cout << el->key << " ";
    }
};
```

Observații:

- La începutul codului am adăugat:

```
#include<stdexcept>
```

care permite tratarea excepțiilor. În C++, *aruncarea unei excepții* ($throw$) este un mecanism prin care semnalăm că a apărut o eroare în timpul execuției programului. În cazul nostru, eroarea apare atunci când vrem să ștergem un element dintr-o listă care este deja goală, ceea ce se numește o subdepășire (*underflow*).

- Metoda $pop_front()$ are rolul de a șterge primul nod dintr-o listă dublu înlănțuită. Dacă lista nu este goală, ștergerea are loc normal. Însă dacă lista este goală ($head == nullptr$), nu avem ce element să ștergem. În această situație, este corect din punct de vedere logic și sigur pentru program să aruncăm o excepție:

```
throw std::underflow_error("Nu putem sterge din
   lista vida");
```

  adică să informăm restul programului că această operație nu este permisă.

- Aruncarea excepțiilor este utilă deoarece:
  - Previne comportamentul incorect al programului (de exemplu, dereferențierea unui pointer nul).
  - Oferă un mesaj clar despre ce nu a funcționat.
  - Permite tratarea elegantă a erorilor folosind blocuri *try/catch*.

- Datorită faptului, că am definit constructor cu parametrii (impliciți) pentru $node$, se simplifică și funcția de adăugare. Completarea câmpurilor nodului nou se face direct prin transmiterea parametrilor corespunzători.

Funcția $main$:

```
int main(){
    List L;
    int option, val;
    do {
    std::cout << "1. Inserare la incepututl listei \n";
    std::cout << "2. Stergerea elementului de la inceputul
       listei \n";
    std::cout << "3. Cautarea unei valori \n";
    std::cout << "4. Afisarea listei \n";
    std::cout << "opt ="; std::cin >> option;

        switch (option) {
        case 1:{
            std::cout << "Val=";
```

```
                std::cin >> val;
                L.push_front(val);
                break;
            }
        case 2:{
                L.pop_front();
                break;
            }
        case 3:{
                std::cout << "Val=";
                std::cin >> val;
                List::node* nod = L.find(val);
                if (nod != nullptr)   std::cout << "Gasit\n";
                else std::cout << "Nu am gasit\n";
                break;
        }
        case 4:
                L.print();
                  std::cout << "\n";
                  break;
    }
    } while (option < 5);
    return 0;
}
```

Justificarea soluției propuse: Algoritmii sunt corecți având la bază pseudocodul din teorie. Funcțiile de adăugare și ștergere au tratat cazurile limită, respectiv când lista pe care se realizează operațiile este vidă, la inserție prin tratarea variabilei $tail$, iar la ștergere prin aruncarea unei excepții.

Analiza complexității:

- *Complexitate temporală:*
  - Inserția și ștergerea la începutul listei sunt în mod evident $O(1)$
  - Căutarea și afișarea listei sunt $O(n)$, n = numărul de elemente din listă
- *Complexitate spațială:* $O(n)$, pentru că stocăm n elemente în nodurile listei.

2. Se consideră o listă $L1$ care conține elemente de tip `std::tuple<string, string, int>`. Fiecare element reprezintă o persoană cu nume, prenume și vârstă. Rezolvați folosind STL:

a) Citiți n elementele dintr-un fișier $Lista1.txt$ în $L1$ (cu $emplace_back$).

b) Sortați descrescător elementele acestei liste în funcție de vârstă și afișați elementele listei.

c) Căutați în listă persoana cu numele "Popescu" și afișați informațiile despre aceasta.

d) Mai citiți încă o listă $L2$ de persoane din fișierul $Lista2.txt$, iar apoi salvați persoanele comune celor două liste într-o a treia listă $L3$.

Rezolvare:

Date de intrare:

- două variabile $L1$ și $L2$ de tip `std::list<std::tuple<string, string, int»`
- două fișiere $Lista1.txt$ și $Lista2.txt$ cu înregistrări de tip $string$ - nume, $string$ - prenume și int - vârsta.

Date de ieșire:

a) lista $L1$ cu elementele din fișierul "Lista1.txt";

b) lista $L1$ sortată descrescător în funcție de vârstă;

c) iterator către elementul reprezentând persoana cu numele "Popescu", dacă există în $L1$;

d) lista $L2$ cu elementele din fișierul "Lista2.txt" și o a treia lista $L3$ care conține intersecția dintre $L1$ și $L2$.

Algoritm de rezolvare:

a) Se deschide fișierul "Lista1.txt". Cât timp există înregistrări în fișier, se citesc câte 2 variabile de tip $string$ și o variabilă de tip int și se adaugă cu $emplace_back$ la $L1$.

b) Se sortează $L1$ folosind $sort$ din $algorithms$.

c) Se caută persoana "Popescu" folosind $find_if$ din $algorithms$.

d)
- **Varianta naivă**. Se parcurge lista $L1$ și se caută cu $find$ fiecare element din $L1$ în $L2$. Dacă un element din $L1$ există în $L2$, atunci se adaugă la $L3$.
- **Varianta avansată**. Se folosește o tabelă de dispersie. Va fi discutată în capitolul corespunzător.

Algoritm scris în C++: Algoritmul este împărțit pe bucăți pentru o înțelegere mai bună. Începem cu includere de header-e și definirea tipului $persoana$ pentru simplificarea codului.

În plus, pentru sortarea elementelor de tip $tuple$ după una dintre componente cu ajutorul algoritmului $sort$, definim un *comparator*. Acesta va fi dat printr-o funcție, care compară două elemente de tip $persona$.

În cadrul $sort$, funcția de comparare acționează în modul următor: atunci când returnează $true$, se consideră primul argumentul trimis către funcție ca fiind înaintea celui de-al doilea în vectorul sortat, iar când returnează $false$, invers.

```
#include<iostream>
#include<fstream>
#include<list>
#include<tuple>

#include<algorithm>

typedef std::tuple < std::string, std::string, int> persoana;

bool compare(const persoana& p1, const persoana& p2)
{
    return  std::get<2>(p1) > std::get<2>(p2);
}
```

Deci: dacă $compare(p1, p2)$ returnează $true$, atunci $p1$ va fi înaintea lui $p2$ în șirul sortat, altfel ordinea va fi invers.

Citirea unei listei dintr-un fișier se poate face cu o funcție adecvată. Această funcție va fi apelată în $main$.

```
void citireLista(std::list<persoana>& lista, std::string cale)
{
    std::ifstream file(cale);
    while (!file.eof())
    {
        std::string nume, prenume;
        int varsta;
        file >> nume >> prenume >> varsta;
        lista.emplace_back(nume, prenume, varsta);

    }
    file.close();
}
```

Funcția main rezolvă cele 4 puncte din cerință. Pentru punctul c) oferim și o varianta generalizată, de găsire a unei persoane cu un nume citit de la tastatură.

```
int main()
{
    std::list<persoana>L1, L2, L3;
    //[a)] Citirea listei L1
    citireLista(L1, "Lista1.txt");

    if (!L1.empty()){
    //[b)] Sortarea listei L1
        L1.sort(compare); //se pune ca parametru comparatorul

        for (const persoana& p : L1){
            std::cout << std::get<0>(p) << " " <<
               std::get<1>(p) << " cu varsta de " <<
               std::get<2>(p) << " ani\n";
        }

        //[c)] Cautarea persoanei "Popescu"
        //varianta "clasica" cu parcurgerea listei
        auto it2 = L1.begin();

        //Acest for nu face nimic altceva, decat sa avanseze
           in lista
        //pana gaseste numele "Popescu"
        for (; it2 != L1.end() && std::get<0>(*it2) !=
           "Popescu"; it2++);
        if (it2 != L1.end()){
              std::cout << "Am gasit persoana: " <<
                 std::get<0>(*it2) << " " <<
                 std::get<1>(*it2) << " cu varsta " <<
                 std::get<2>(*it2) << "\n";
        }

         //[c)] Varianta generalizata. Cautarea persoanei
         //al carui nume este citit de la tastatura
        std::string nume;
        std::cout << "Ce nume?";
        std::cin >> nume;

        //varianta cu lambda expresii.
        auto it = std::find_if(L1.begin(), L1.end(),
           [nume](persoana e) { return std::get<0>(e) ==
           nume; });

        if (it != L1.end())
```

```
        std::cout << "Am gasit persoana: " <<
           std::get<0>(*it) << " " << std::get<1>(*it) << "
           cu varsta " << std::get<2>(*it) << "\n";
        else std::cout << "Nu exista persoana!\n";
    }

    //[d)]
    citireLista(L2, "Lista2.txt");

    //Determinarea intersectiei cu un for si o instructiune
       find
    for (const persoana& p : L2)
    {
        auto it = std::find(L1.begin(), L1.end(), p);
        if (it != L1.end())
        {
           std::cout << "comun " << std::get<0>(*it) << " "
              << std::get<1>(*it) << " cu varsta " <<
              std::get<2>(*it) << "\n";
           L3.push_back(*it);
        }
    }
    return 0;
}
```

Justificarea soluției propuse: Programul este în mod evident corect. Structurile și algoritmii din STL preiau o mare parte din rezolvare și nu este necesar să explicăm validitatea lor.

Complexitate de timp și spațiu:

a) Este o citire de n elemente, în complexitate liniară de timp și de spațiu.

b) Sortarea este cea din `algorithms` din biblioteca STL, care are complexitatea de timp $O(n \log_2 n)$. Complexitatea spațială este constantă, nu necesită memorie suplimentară.

c) Căutarea în ambele variante are complexitate de timp liniară, pentru că presupune parcurgerea listei. Dar complexitatea de spațiu este constantă (folosim câteva variabile de tip primitiv suplimentare).

d) Complexitatea de timp pentru determinarea intersecției este pătratică, pentru că presupune parcurgerea listei $L1$ și pentru fiecare element căutarea cu $find$ în $L2$.

3. **Structura Stack.** Definiți o structură $template$ de tip $Stack$, care să implementeze funcționalitatea unei stive folosind o listă simplu înlănțuită. În funcția principală *main* se va declara un obiect de tip $Stack < int >$, în care se vor insera pe rând elementele 1, 2, 3, 4, 5. Cât timp stiva nu e vidă, se afișează elementul din vârful stivei, apoi acesta se elimină din stivă.
   Se poate observa faptul că, la problema anterioară de implementare a listei înlănțuite, toată structura a fost inclusă într-un singur fișier, iar elementele structurilor au fost inițializate direct. În plus, lista a fost definită doar pentru tipul int.
   În cazul implementării stivei, vom prezenta o soluție mai elegantă și mai bine organizată din punct de vedere orientat obiect, dar care necesită cunoștințe ceva mai avansate. În plus soluția prezentată va defini stiva de tip $template$, astfel încât să poată stoca orice tip de date se dorește. Vom folosi un fișier de *header* pentru structură și un fișier .cpp pentru implementarea funcțiilor structurii.

   Date de intrare: Valorile 1, 2, 3, 4, 5 care trebuie inserate în stivă, iar apoi extrase și afișate.
   Date de ieșire:

   - structura $Stiva$
   - valorile din stivă afișate în ordine inversă inserării (principiul de stivă).

   Algoritm de rezolvare
   Pentru funcțiile $push$ și pop se folosesc algoritmi similari cu $push_front$ și pop_front de la liste.
   Funcția top doar returnează elementul din vârful stivei.
   Funcția $empty$ testează dacă nodul din vârf este $nullptr$.
   În funcția $main$ se declară o variabilă de tip $Stack < int >$, apoi se inserează elementele 1, 2, 3, 4, 5 folosind funcția $push$. În final, într-o buclă $while$ se va afișa mereu elementul din vârf folosind funcția top și se va șterge cu funcția pop.

   Algoritm scris în C++:
   Fișierul de *header*, **Stack.h** conține definiția structurii:

```
#pragma once
template<class T>
class Stack{
  //dat fiind ca stiva va avea la baza o lista, avem nevoie
     de o
  //structura de tip node
   struct node {
    T key; //cheia este template
    node* next;
        //constructor pentru node
    node(T value, node* next = nullptr):
        key(value),next(next){}
   };
   node* Top;
   int size;
   public:
        //constructor pentru stiva
    Stack():Top(nullptr), size(0) {};
    void push(T element);
    void pop();
    int top();
    bool empty();
};
```

Observații:

- Pentru structura $node$ se observă folosirea unui constructor, care are ca parametri o variabilă de tip T (tipul pe care îl vom defini la declararea unei variabilă de tip $Stack$) și o variabilă de tip $node*$ inițializată cu $nullptr$.
- Pentru structura $Stack$ avem un constructor, care inițializează capul stivei cu $nullptr$ și variabila $size$, care indică numărul de elemente stocate în stivă, cu 0.
- În fișierul **Stack.h** avem doar antetele funcțiilor, care vor fi definite însă în fișierul $Stack.cpp$

Fișierul **Stack.cpp** conține definițiile funcțiilor membre:

```
#include<stdexcept>
#include "Stack.h"
template<class T>
void Stack<T>::push(T element){
    node* newNode = new node(element, Top);
    Top = newNode;
    size++;
}
```

```
template<class T>
void Stack<T>::pop(){
    if(Top){
            node* toDelete = Top;
            Top = Top->next;
            delete toDelete;
            size--;
        }
        else {
            throw std::underflow_error("Nu putem sterge din
               lista vida");
        }
}
template<class T>
int Stack<T>::top(){
    if(Top)
       return Top->key;
    else throw std::underflow_error("Lista este vida");
}
template<class T>
bool Stack<T>::empty(){
    if (!Top) return true;
    return false;
}
```

Funcția principală *main*, în care se declară un obiect *stiva* de tip $stack < int >$, în care se vor insera pe rând elementele 1, 2, 3, 4, 5. Apoi, cât timp stiva nu e vidă, se afișează elementul din vârful stivei, apoi acesta se elimină din stivă.

```
#include <iostream>
#include"Stack.cpp"
int main(){
    Stack<int>stiva;
    int vec[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++)
        stiva.push(vec[i]);
    while (!stiva.empty()){
        std::cout << stiva.top();
        stiva.pop(); }
}
```

Justificarea soluției propuse: Observăm că funcția $push$ respectă algoritmul de $push_front$ pentru liste simplu înlănțuite. Apelul constructorului în prima linie din corpul funcției

realizează alocarea dinamică de memorie pentru noul nod și legarea de capul vechi al listei, iar a doua linie resetează capul listei la noul element.
Funcția pop urmărește întocmai algoritmul din pseudo-cod, iar funcțiile top și $empty$ sunt evident corecte.

Analiza complexității

- *Complexitate temporală*
  - Toate funcțiile din structura $Stack$ au complexitate constantă
  - În funcția $main$ se parcurge vectorul vec în complexitate $O(n)$, unde $n = 5$ este numărul de elemente din vectorul, ale cărui elemente sunt adăugate în stivă.
  - Tot în funcția $main$ elementele din stivă sunt extrase pe rând și afișate, tot în complexitate $O(n)$.
- *Complexitate spațială* Stocarea elementelor în vectorul vec și în stivă are complexitatea $O(n)$.

4. **Problema parantezării corecte.** Se consideră un șir de paranteze rotunde închise și deschise. Să se verifice, dacă parantezarea este corectă. **Exemplu:** $(()())$ este corectă, $())($ nu este corectă. Această problemă poate fi extinsă pentru toate tipurile de paranteze: $[]$, $\{\}$, $()$, cu condițiile suplimentare că fiecare tip de paranteză închisă trebuie să închidă același tip de paranteză deschisă și nicio pereche de paranteze de ordin mai mare nu se poate afla în interiorul uneia de ordin mai mic.

Rezolvare:
Date de intrare: o variabilă de tip $string$ ce conține șirul de paranteze (fără alte caractere).
Date de ieșire: un mesaj care indică dacă parantezarea este corectă sau nu.

Algoritm de rezolvare: Se folosește o stivă $S_paranteze$. Se citește șirul $paranteze$. Se parcurge acest șir caracter cu caracter și pentru fiecare caracter, dacă este $'('$ se pune pe stivă. Altfel, dacă este $')'$, avem două posibilități:

- stiva este vidă - caz în care această paranteză închisă nu corespunde niciunei paranteze deschise ⇒ eroare de parantezare.
- stiva nu este vidă ⇒ se șterge vârful stivei, deoarece s-a închis această paranteză.

Dacă la sfârșitul parcurgerii șirului stiva este vidă, atunci parantezarea este corectă. Altfel există paranteze deschise care nu au fost închise și deci, parantezarea nu este corectă. Funcția care verifică parantezarea corectă, scrisă în C++ folosind structuri din STL este prezentat mai jos.

Algoritm scris în C++:

```
bool parantezare(std::string paranteze){
    std::stack<char>S_paranteze;
    for (char c : paranteze){
        if (c == '(')
              S_paranteze.push(c);
        else{
              if (c==')' && S_paranteze.empty())
                 return false;
              S_paranteze.pop();
           }
    }
    if (S_paranteze.empty())
        return true;
    return false;
}
```

Justificarea soluției propuse: Se observă faptul că, dacă caracterul curent este o paranteză închisă, trebuie ca ea mai întâi să fi fost deschisă, deci trebuie să existe cel puțin o paranteză deschisă în stivă, care va fi ștearsă, pentru a marca faptul că o pereche de paranteze potrivite a fost deja analizată. În caz contrat algoritmul se oprește cu un mesaj de eroare. Dacă șirul de paranteze s-a terminat de parcurs, dar mai există paranteze deschise în stivă, înseamnă că au fost deschise prea multe paranteze și acestea nu au corespondent o paranteză închisă. Deci pentru o parantezare corectă, stiva trebuie să fie vidă la final.

Analiza complexității

- *Complexitate temporală*. Algoritmul presupune parcurgerea iterativă a întregului șir de paranteze, deci este $O(n)$ unde n este lungimea șirului.

- *Complexitate spațială*. Stocarea parantezelor în variabila de tip $strig$, precum și a celor deschise în stivă are complexitatea $O(n)$.

5. **Forma poloneză postfixată.** Evaluați o expresie aritmetică ce conține operanzi numere întregi cu o singură cifră și operatorii $+$, $-$, $/$, $*$, folosind forma poloneză postfixată.

Rezolvare:

Forma poloneză postfixată, denumită în engleză ***Reverse polish notation - RPN*** sau ***Polish postfix notation*** se numește astfel, deoarece a fost dezvoltată de un logician Polonez [9]. Această formă presupune plasarea operatorilor după operanzi, respectând ordinea operanzilor, dar plasând operatorii exact în ordinea în care trebuie efectuați. Astfel nu mai sunt necesare parantezele și evaluarea expresiei se poate face foarte ușor prin parcurgerea formei poloneze și aplicarea operatorilor în ordinea în care apar.

Exemple:

Expresie aritmetică	Forma poloneză postfixată corespunzătoare
$2 + 3$	$2\ 3\ +$
$2 + 3 * 5$	$2\ 3\ 5\ *+$
$(2 + 3) * 5$	$2\ 3 + 5*$
$2 + 3 * (5 - 1 - 1) - 3 * (5 - 2 * (2 - 1))$	$2\ 3\ 5\ 1 - 1 - * + 3\ 5\ 2\ 2\ 1 - * - * -$

Construirea formei poloneze postfixate

Date de intrare: Șirul de caractere care reprezintă expresia aritmetică.
Date de ieșire: Forma poloneză postfixată asociată.

Algoritm de rezolvare: Pentru construirea formei poloneze se utilizează o stivă de caractere în care vor fi plasați operatorii din expresie, împreună cu parantezele deschise. Numim această stivă $stiva_op$. De asemenea se utilizează un vector fp pentru stocarea formei poloneze. Acesta este inițial vid.

Construcția formei poloneze postfixate se realizează în modul următor: se parcurge iterativ șirul de caractere care reprezintă expresia aritmetică. Pentru caracterul $curent$ avem următoarele posibilități:

- $curent$ = cifră (presupunem deocamdată că operanzii au o singură cifră) $\Rightarrow$ se pune automat în vectorul fp
- $curent$ = '(' $\Rightarrow$ se pune automat în stivă.
- $curent$ = ')' $\Rightarrow$ se scot de pe stivă toți operatorii până la prima paranteză deschisă și se pun în vectorul fp. Apoi se scoate și paranteza deschisă. Dacă s-a golit stiva și NU s-a întâlnit paranteză deschisă $\Rightarrow$ eroare de parantezare și nu se poate construi corect forma poloneză.
- $curent$ = operator $\Rightarrow$ trebuie pus pe stivă. DAR înainte se analizează stiva. Cât timp operatorul din vârful stivei e de prioritate cel puțin egală cu cea a lui $curent$, se extrage din stivă și se plasează în fp. Abia apoi se pune $curent$ pe stivă.

După terminarea parcurgerii șirului de intrare, se extrag toți operatorii rămași în stivă și se plasează în forma poloneză. Dacă în stivă au mai rămas paranteze deschise, atunci este eroare de parantezare.
Precedența este stabilită astfel încât: '(' să aibă cea mai mică prioritate, $+$, $-$ au următoarea prioritate și $*$, $/$ au cea mai mare prioritate.

Algorithm 9: Forma_poloneza

Input: expresia aritmetica $expr$

begin

$fp \leftarrow \emptyset$

$stiva_op \leftarrow \emptyset$

for $\forall curent \in expr$ **do**

if *isDigit*$(curent)$ **then**

$fp.push_back(curent)$

else

if $curent =' ('$ **then**

$stiva_op.push(curent)$

else

if $curent =$ ‘)’ **then**

while $stiva_op! = \emptyset$ *AND* $stiva_op.top() ! =' ('$ **do**

$fp.push_back(stiva_op.top())$

$stiva_op.pop()$

end

if $stiva_op! = \emptyset$ **then**

$stiva_op.pop()$ //scoate paranteza

end

else

//daca $curent$ e operator

while $stiva_op! = \emptyset$ *AND*

$prioritate(stiva_op.top()) \geq prioritate(curent)$ **do**

$fp.push_back(stiva_op.top())$

$stiva_op.pop()$

end

$stiva_op.push(curent)$

end

end

end

while $stiva_opk! = \emptyset$ **do**

$fp.push_back(stiva_op.top())$

$stiva_op.pop()$

end

end

Observații:

- Trebuie observat faptul că se consideră că în forma *expr* există doar caractere de tip cifra (numere cu o singură cifră) și caracterele operator $(,)$, $+$, $-$, $*$ și $/$. De asemenea se consideră că nu există spații albe în expresie. Pentru numere cu mai mult de o cifră și alți operatori, precum și alte tipuri de paranteze sau cu spații albe algoritmul trebuie adaptat.
- Prioritatea operațiilor indică ordinea efectuării lor într-o expresie fără paranteze. De exemplu, prioritatea operatorului '$+$' e egală cu cea a lui '$-$' și mai mică decât a operatorului '$*$'. În plus, se recomandă ca paranteza deschisă să primească cea mai mică prioritate, pentru a nu fi scoasă din stivă de alt operator, ci numai de paranteza închisă.
- Pentru evaluare se poate utiliza o stivă de numere. Se parcurge forma poloneză caracter cu caracter. Dacă caracterul curent este o cifră, se introduce în stivă, dacă este un operator, se extrag din stivă două numere din vârf, se aplică operatorul, iar rezultatul se pune din nou în stivă. Numărul rămas la sfârșit în stivă este rezultatul expresiei aritmetice.

Implementarea algoritmului rămâne ca exercițiu de laborator, împreună cu tratarea erorilor și a situației, când se iau în considerare numere cu mai mult de o cifră.

Justificarea soluției propuse:

- Faptul că fiecare cifră este adăugată direct în forma poloneză asigură faptul că operanzii își păstrează ordinea.
- Folosirea unei stive de operatori și faptul că fiecare operator va intra întâi în stivă, după o ordine dată de parantezare și precedență, asigură ordonarea operatorilor în ordinea execuției lor.
- Faptul că un operator mai "slab" scoate operatorii mai "puternici" din stivă asigură efectuarea operațiilor cu precedența mai mare înaintea celor cu precedența mai mică.

- Intrarea parantezei deschise cu cea mai mică prioritate în stivă asigură faptul că doar paranteza închisă va scoate paranteza deschisă din stivă și toți operatorii din interiorul unei perechi de paranteze vor fi executați în consecință.

Exemplu: Se consideră expresia aritmetică: $2 + 3 * (5 - 1 - 1) - 3$
Inițial stiva $stiva_op = \emptyset$ și vectorul $fp = \emptyset$.
În bucla for considerăm $curent$ caracterul curent din expresia aritmetică. Atunci:

- $curent =' 2' \Rightarrow$ se pune în forma poloneză care va deveni $fp = \{'2'\}$
- $curent =' +'$ și $stiva_op = \emptyset \Rightarrow$ pun $'+'$ pe stivă, care devine $stiva_op = \{'+'\}$.
- $curent =' 3' \Rightarrow$ se pune în forma poloneză care va deveni $fp = \{'2',' 3'\}$
- $curent =' *'$. În vârful stivei este $'+'$ și $prioritate('+') < prioritate('*') \Rightarrow$ se pune $'*'$ pe stiva, care devine $stiva_op = \{'+',' *'\}$.
- $curent =' ('\Rightarrow stiva_op.push('(') \Rightarrow stiva_op = \{'+',' *',' ('\}$
- $curent =' 5' \Rightarrow fp.push_back('5') \Rightarrow fp = \{'2',' 3',' 5'\}$
- $curent =' -'$ și $prioritate('(') < prioritate('-') \Rightarrow stiva_op.push('-') \Rightarrow stiva_op = \{'+',' *',' (',' -'\}$
- $curent =' 1' \Rightarrow fp.push_back('1') \Rightarrow fp = \{'2',' 3',' 5',' 1'\}$
- $curent =' -'$ și $prioritate('-') = prioritate('-') \Rightarrow fp.push_back(stiva_op.top())$ și $stiva_op.pop()$ și $stiva_op.push('-') \Rightarrow stiva_op = \{'+',' *',' (',' -'\}$ și $fp = \{'2',' 3',' 5',' 1',' -'\}$
- $curent =' 1' \Rightarrow fp.push_back('1') \Rightarrow fp = \{'2',' 3',' 5',' 1',' -',' 1'\}$
- $curent =')' \Rightarrow$ scoate din stivă toți operatorii până la paranteză deschisă și pune-i în forma poloneză, iar apoi scoate $'('$ din stivă $\Rightarrow stiva_op = \{'+',' *'\}$ și $fp = \{'2',' 3',' 5',' 1',' -',' 1',' -'\}$
- $curent =' -'$ și $prioritate('*') > prioritate('-') \Rightarrow fp.push_back(stiva_op.top())$ și $stiva_op.pop()$. Acum $stiva_op = \{'+'\}$ și $fp = \{'2',' 3',' 5',' 1',' -',' 1',' -',' *'\}$. În continuare $prioritate('-') = prioritate('+')$ (care e în vârful stivei) $\Rightarrow$ $fp.push_back(stiva_op.top())$ și $stiva_op.pop()$. Apoi se face $stiva_op.push('-')$. Acum: $stiva_op = \{'-'\}$ și $fp = \{'2',' 3',' 5',' 1',' -',' 1',' -',' *',' +'\}$

- $curent =' 3' \Rightarrow fp.push_back('3')$. $fp = \{'2','3','5','1','-','1','-','*','+','3'\}$, șirul s-a terminat și în stivă mai avem $'-'$. Acesta se extrage din stivă și se pune în forma poloneză postfixată.

Rezultatul obținut: 2 3 5 1 - 1 - * + 3 -

Analiza complexității:

- *Complexitate temporală:* $O(n)$, unde n este lungimea șirului de elemente din expresia aritmetică, deoarece trebuie parcurs fiecare element și plasat în forma poloneză sau în stiva de operatori. Numărul de elemente din stiva de operatori nu va depăși niciodată n și un element nu se introduce de 2 ori, deci operarea stivei nu modifică complexitatea de timp.
- *Complexitate spațială:* avem n elemente în expresia aritmetică, din care intră cel mult n în forma poloneză, asta dacă nu există paranteze. De asemenea stiva nu va ajunge niciodată să conțină mai mult de n elemente, de obicei mult mai puține. Astfel complexitatea de spațiu este tot $O(n)$.

Algoritm de evaluare a formei poloneze postfixate

Date de intrare: Șirul de caractere care reprezintă forma poloneză postfixată.
Date de ieșire: Rezultatul numeric al evaluării expresiei aritmetice corespunzătoare.

Algoritm de rezolvare:
Observații: În forma poloneză postfixată ordinea operanzilor este aceeași ca în expresia obișnuită, DAR operatorii apar în ordinea efectuări lor. În plus, nu mai sunt necesare paranteze. Astfel evaluarea expresiei se va realiza foarte simplu în modul următor: se parcurge forma poloneză și la întâlnirea unui operator, acesta se aplică celor 2 operanzi care îl preced.

Algorithm 10: Evaluate

Input: forma poloneza postfixată fp

begin
  $stiva_num \leftarrow \emptyset$
  for $\forall el \in fp$ **do**
    if $el \in \{'0', '1', \ldots, '9'\}$ **then**
      $stiva_num.push(int(el))$
    else
      if $stiva_num.size() < 2$ **then**
        scrie("lipsesc operanzi")
        return "eroare"
      else
        $y \leftarrow stiva_num.top()$
        $stiva_num.pop()$
        $x \leftarrow stiva_num.top()$
        $stiva_num.pop()$
      end
      $operator \leftarrow el$
      calculeaza $rez \leftarrow x\ operator\ y$
      $stiva_num.push(rez)$
    end
  end
  if $stiva_num.size() = 1$ **then**
    scrie("rezultatul este ", $stiva_num.top()$)
  else
    scrie("prea multi operanzi")
    return "eroare"
  end
end

Exemplu. Evaluarea forme poloneze postfixate 2 3 5 1 - 1 - * + 3 - se realizează astfel:

Primul operator întâlnit este $-$, care se aplică pentru 5 și 1:

$2\ 3\ \boxed{5\ 1\ -}\ 1\ -\ *\ +\ 3\ 5\ 2\ 2\ 1\ -\ *\ -\ *\ -$.

Se obține: $2\ 3\ 4\ 1\ -\ *\ +\ 3\ 5\ 2\ 2\ 1\ -\ *\ -\ *\ -$.

Acum, primul operator întâlnit este $-$ și se aplică pentru 4 și 1:

$2\ 3\ 4\ 1\ -\ *+\ 3\ 5\ 2\ 2\ 1\ -\ *\ -\ *\ -$

Se obține: $2\ 3\ 3\ *+\ 3\ 5\ 2\ 2\ 1\ -\ *\ -\ *\ -$.

Acum, primul operator întâlnit este $*$ și se aplică pentru 3 și 3:

$2\ 3\ 3\ *\ +\ 3\ 5\ 2\ 2\ 1\ -\ *\ -\ *\ -$

Se obține: $2\ 9\ +\ 3\ 5\ 2\ 2\ 1\ -\ *\ -\ *\ -$.

Acum, primul operator întâlnit este $+$ și se aplică pentru 2 și 9:

$2\ 9\ +\ 3\ 5\ 2\ 2\ 1\ -\ *\ -\ *\ -$

Se obține: $11\ 3\ 5\ 2\ 2\ 1\ -\ *\ -\ *\ -$.

Acum, primul operator întâlnit este $-$ și se aplică pentru 2 și 1:

$11\ 3\ 5\ 2\ 2\ 1\ -\ *\ -\ *-$

Se obține: $11\ 3\ 5\ 2\ 1\ *\ -\ *-$.

Acum, primul operator întâlnit este $*$ și se aplică pentru 2 și 1:

$11\ 3\ 5\ 2\ 1\ *\ -\ *\ -$

Se obține: $11\ 3\ 5\ 2\ -\ *\ -$.

Acum, primul operator întâlnit este $-$ și se aplică pentru 5 și 2:

$11\ 3\ 5\ 2\ -\ *-$

Se obține: $11\ 3\ 3\ *-$.

Acum, primul operator întâlnit este $*$ și se aplică pentru 3 și 3:

$11\ 3\ 3\ *\ -$

Se obține: $11\ 9\ -$.

Singurul operator rămas este $-$ și se aplică pentru 11 și 9:

$11\ 9\ -$

Se obține: 2, care este soluția expresiei aritmetice.

Justificarea soluției propuse: trebuie observat faptul că se consideră că în forma poloneză există doar caractere de tip cifra (numere cu o singură cifră) și caracterele operator $+$, $-$, $*$ și $/$. Pentru numere cu mai mult de o cifră și alți operatori algoritmul trebuie adaptat.

- Adăugarea numerelor pe stivă ne asigură că primii operanzi care se vor lua în considerare sunt aceia, care se află fix înaintea primului operator.

- Extragerea primelor două elemente din vârful stivei și considerarea lor în ordine inversă, atunci când se realizează operația curentă, permite efectuarea operațiilor în ordinea corectă
- Testarea ca stiva să conțină cel puțin 2 elemente înainte de extragere asigură o bună funcționare și atunci când numărul de operatori este prea mare și deci forma poloneză nu este corectă.
- La final, stiva de numere trebuie să conțină o singură valoare care reprezintă rezultatul.

Analiza complexității

- *Complexitate temporală:* $O(n)$, unde n este lungimea formei poloneze postfixate.
- *Complexitate spațială:* $O(n)$, pentru forma poloneză postfixată. Stiva de numere ocupă cel mult $2/3n$ elemente, dar de obicei mult mai puține.

7.3 Probleme propuse

*

1. **Listă înlănțuită.** Considerând prima problemă rezolvată, de implementare a unei liste dublu înlănțuite, adăugații funcțiile:
   - $push_back(key)$ - adaugă cheia key la finalul listei;
   - $pop_back()$ - șterge ultimul element din listă;
   - $remove(int\ key)$ - șterge toate aparițiile cheii key (implică parcurgerea listei). Găsiți un algoritm în materialul de curs[8];
   - $clear()$ - golește lista și eliberează memoria alocată pentru fiecare nod.
2. **Implementare coadă**. Să se implementeze o coadă utilizând liste înlănțuite. Vă trebuie:
   - o structură $node$ cu două câmpuri - un câmp pentru informație (eventual $template$) și un câmp de tip pointer la $node$ pentru legătura la elementul următor.
   - o structură $Coada$ cu

- două câmpuri de tip pointer la nod, pentru primul și ultimul element, inițializate cu $nullptr$;
- un câmp nr_elem de tip int - numărul de elemente din coada;
- funcția $push(elem)$ - pune $elem$ la sfârșitul cozii;
- funcția $pop()$ - elimină elementul de la începutul cozii;
- funcțiile $front()$ și $back()$ returnează primul respectiv ultimul element din coadă;
- funcția $empty()$ - verifică dacă coada este vidă;
- funcția $clear()$ - golește coada;
- funcția $size()$ - returnează numărul de elemente din coadă.

3. **Parantezare corectă.** Se dă un șir format din paranteze deschise și închise de tipurile `(`, `)`, `[`, `]`, `{`, `}`. Scrieți o funcție care primește ca parametru acest șir (`std::string`) și returnează `true` dacă șirul reprezintă o parantezare corectă și `false` în caz contrar.
   O parantezare este corectă dacă fiecare paranteză deschisă este închisă de o paranteză de același tip, iar perechile sunt corect încuibate. În plus:
   - în interiorul unei perechi de paranteze rotunde nu se pot afla alte paranteze;
   - în interiorul unei perechi de paranteze pătrate se pot afla doar perechi de paranteze rotunde;
   - în interiorul unei perechi de acolade se pot afla perechi de paranteze rotunde și pătrate, dar nu și alte perechi de acolade.

   Perechile aflate pe același nivel pot fi succesive (de exemplu, `()[]{}` este corect). Folosiți o stivă (`std::stack`) pentru rezolvarea problemei.
   Exemple: șirurile `[()()]` și `{[()]}` sunt corecte; șirurile `([])`, `()]`, `{{}}` și `[{}]` nu sunt corecte.

**

1. Considerând prima problemă rezolvată, de implementare a unei liste dublu înlănțuite, adăugați funcțiile:

- $erase(node * Nod)$ - șterge un element x de tip $node$ din listă (NU implică căutare). Nodul transmis ca parametru a fost în prealabil găsit cu $find$ sau identificat prin parcurgerea listei;
- $insert(node * Nod, int\ val)$ - inserează un element cu cheia val înainte de nodul indicat de Nod;
- $insert_before(key1, key2)$ - inserează un element cu cheia $key2$ înainte prima apariție a cheii $key1$. Dacă $key1$ nu apare în listă, atunci $key2$ se adaugă la finalul listei.

2. Considerând prima problemă rezolvată, de implementare a unei liste dublu înlănțuite, scrieți o funcție care preia două elemente de tip lista $L1$ și $L2$ și realizează operația de concatenare a elementelor listei $L2$ la sfârșitul listei $L1$, capul listei $L2$ devenind apoi $nullptr$. Rezolvați eficient.

3. Considerând codul pentru implementarea unei liste dublu înlănțuite, scrieți o funcție $palindrom$, care verifică dacă lista are elementele ordonate palindromic.

***

1. Considerând implementarea structurii lista dublu înlănțuită, scrieți un algoritm care are ca parametru o astfel de listă și o sortează, în cea mai bună complexitatea de spațiu și timp posibilă.

2. Folosind o coadă (de preferință în implementare proprie) rezolvați următoarea problemă. La un examen se pot prezenta candidați pe durata a două zile. În fiecare zi timpul alocat pentru examinare este de t ore ($t \leq 6$). La examen se înscriu n candidați. Se citesc din fișier t, n precum și candidații cu numele (de tip **std::string**). Ei vor fi introduși într-o coadă, de unde vor fi extrași pe rând pentru examinare. Pentru fiecare candidat care este la rând, se generează o durată aleatorie cu o valoare între 5 minute și 15 minute. În momentul în care timpul t s-a terminat, deci se încheie prima zi de evaluare, candidații care au rămas în coadă vor fi extrași pe rând și trecuți într-un fișier de ieșire, care va reprezenta lista candidaților pentru ziua a doua de examinare.

3. **Simularea unei stive prin două cozi**. Rezolvați problema parantezării corecte, simulând funcționarea stivei cu ajutorul a două cozi. Puteți folosi `std::queue` din STL. Cerință: definirea unei structuri $stiva$, care în interior să dispună drept containere de două elemente de tip `std::queue` și de funcții de tip $push$ și pop, care apelează însă doar operațiile corespunzătoare ale cozilor din structură. Elementele care se introduc în stivă vor fi stocate în cozile de care dispune stiva. Explicitați într-un comentariu complexitatea operațiilor de adăugare și extragere din această stivă.

4. Implementați evaluarea expresiilor aritmetice folosind forma poloneză postfixată pe baza algoritmilor din secțiunea de probleme rezolvate. Ajustați algoritmii și structurile de date astfel încât să poată fi evaluate și expresii aritmetice cu numere întregi de mai multe cifre și cu posibilitatea să existe și spații în aceste expresii. Aceste spații se vor ignora. De asemenea tratați erorile ce pot apărea în expresia aritmetică.

****

1. **Eliminarea parantezelor redundante**. Se citește dintr-un fișier un șir de caractere reprezentând o expresie aritmetică formată din numere, operatori (+, - /, *), litere (reprezentând variabile), paranteze rotunde, eventual spații albe. Se cere eliminarea parantezelor redundante.

   Exemple:

   (a) Pentru expresia: $((x + y) * 3 + ((z)))$ rezultatul algoritmului ar trebui sa fie: $((x + y) * 3 + z)$.

   (b) Pentru expresia: $(((x + y)) * 3 + ((z - x)))$ rezultatul algoritmului ar trebui sa fie: $((x + y) * 3 + (z - x))$ sau mai bine $(x + y) * 3 + (z - x)$.

   Indicație: puteți folosi ideea de la evaluarea expresiilor aritmetice cu forma poloneză postfixată.

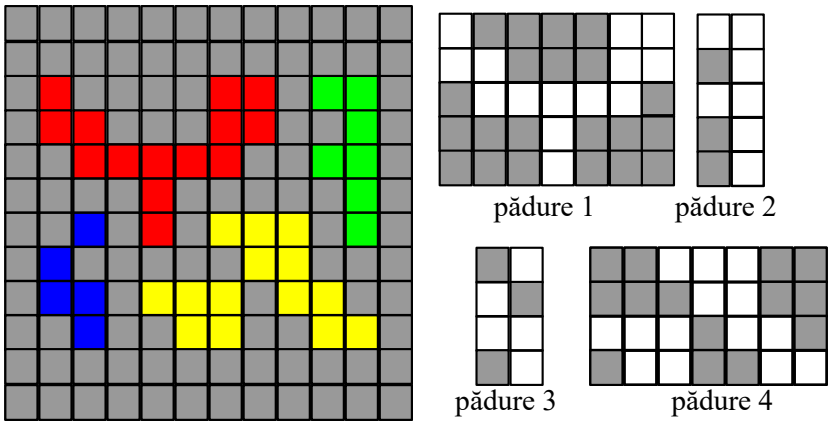
2. **Problema ursilor**. La Brașov pe Tâmpa s-au aventurat doi urși, care reprezintă un pericol pentru populație. Primăria a hotărât relocarea lor în altă zonă împădurită. Pentru relocare se consideră harta unei regiuni potrivite cu acest scop. Harta este reprezentată printr-o

matrice (dimensiune minimă 12×12), în care fiecare poziție reprezintă un hectar de teren. Pozițiile de pădure sunt marcate cu P, celelalte cu $-$. O zonă împădurită compactă este reprezentată de mulțimea tuturor celulelor marcate cu P conectate.

Se va alege pentru relocarea ursilor acea zonă de pădure care:

- are suprafața mai mare decât un prag dat T (citit din fișier) și
- are compactitatea (raportul dintre suprafața sa și suprafața dreptunghiului în care este înscrisă) maximă.

În exemplul din figură se consideră pragul $T = 10$. Zonele colorate reprezintă pădure, iar cele gri sunt regiuni fără pădure. Observăm că dintre cele 4 zone de pădure cea roșie și cea galbenă satisfac ambele cerința ca suprafața să fie mai mare ca T. Compactitatea zonei roșii de pădure este $14/35 = 0.4$, iar a celei galbene este $14/28 = 0.5$. Deci ursii vor fi relocați în zona galbenă. Rezultatul va fi indicat prin perechile de coordonate ale colțului stânga-sus și ale colțului drepta-jos corespunzătoare dreptunghiului în care este înscrisă zona. În exemplul de mai sus $(4, 6)$ și $(10, 9)$ (coordonate ecran!)



Indicație de rezolvare: Se pot folosi o coadă C de coordonate, inițial vidă, și un contor pentru numărul de păduri (care de fapt reprezintă componente conexe). Se începe parcurgerea matricei din colțul stânga sus. La prima coordonată $(coloana, linie)$ pentru care $mat[linie][coloana] =' P'$ se începe detectarea unei prime păduri în modul următor:

- Se incrementează contorul cu 1 (s-a găsit o nouă pădure).

- Se marchează poziția cu valoarea indicată de contor și se plasează perechea $(coloan, linie)$, reprezentând coordonatele ecran, în coada C.
- Se începe o buclă $while(C \neq \emptyset)$ în care, la fiecare iterație:
  - se extrage $curent \leftarrow C.front()$, $C.pop()$ ($curent = (coloana, linie)$);
  - se consideră toți cei 8 vecini vec_i ai coordonatei reprezentată de $curent$, iar acei vecini care reprezintă pădure și încă nu sunt marcați, vor fi marcați cu eticheta pădurii curente și vor fi plasați în coadă ($C.push(vec_i)$).
- Când s-a golit coada, s-a terminat cu pădurea curentă și se reia parcurgerea matricei, iar la următoarea poziție de pădure nemarcată se reia algoritmul de mai sus.

Observație: În bucla $while$, în timp ce se parcurg toate pozițiile corespunzătoare unei păduri, se poate și contoriza numărul de elemente și se pot memora coordonatele minime și maxime ale elementelor pădurii, astfel încât să se poată rezolva și celelalte cerințe.

*****

1. **Deque.** Implementați o structură de tip *deque*. Această structură trebuie să dispună de un container secvențial în care se vor stoca elementele, de funcții de tip *push-back*, *push-front*, *pop-back*, *pop-front*, *size*, *empty*, toate în complexitate constantă. De asemenea trebuie să permită accesul prin poziție la elemente (de exemplu printr-o funcție *at*) și funcții de inserare / ștergere de pe o anumită poziție. Folosiți un vector de `array`-uri (blocuri de memorie de dimensiune fixă). Pentru acest lucru consultați materialul de curs [8].

Capitolul 8

Căutări și sortări simple

8.1 Aspecte teoretice

8.1.1 Căutări

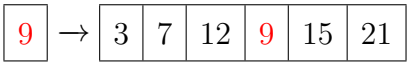
Introducere

Problemele de căutare reprezintă o categorie fundamentală în informatică, având ca scop identificarea existenței sau a poziției unui element într-o colecție de date (inclusiv a unui pointer care indică elementul căutat în structură prin poziție sau adresă). Aceste probleme apar frecvent în aplicații practice, de la prelucrarea informațiilor și organizarea bazelor de date, până la rezolvarea unor situații complexe din viața reală care implică selecția sau regăsirea rapidă a informației.

În mod general, o problemă de căutare presupune existența unei mulțimi de date, numită structură de date, și a unui element sau criteriu care trebuie regăsit în cadrul acesteia. Rezultatul unei căutări poate indica fie poziția elementului dorit, fie faptul că acesta nu se află în structura analizată.

Importanța studiului problemelor de căutare constă în eficiența procesării datelor. Alegerea unei strategii adecvate de căutare influențează direct timpul de execuție și consumul de resurse, mai ales atunci când volumul de date este mare. Din acest motiv, analiza și înțelegerea principiilor generale ale căutării sunt esențiale în proiectarea algoritmilor.

În cadrul acestui subcapitol, vor fi abordate conceptele de bază legate de căutare, modul de formulare a problemelor și criteriile de evaluare a performanței soluțiilor propuse, oferind fundamentul necesar pentru studiul metodelor concrete de căutare.



1. Căutarea liniară

Căutarea liniară este una dintre cele mai simple și intuitive metode de căutare a unui element într-un șir de date. Această metodă presupune parcurgerea elementelor structurii de date, unul câte unul, începând de la prima poziție, până când elementul căutat este găsit sau până când toate elementele au fost verificate.

Căutarea liniară nu impune condiții speciale asupra modului în care sunt organizate datele. Astfel, aceasta poate fi aplicată atât pe șiruri ordonate, cât și pe șiruri neordonate, fiind utilă în special atunci când numărul de elemente este mic sau când structura datelor nu permite alte metode mai eficiente.

Mod de funcționare - Algoritmul compară succesiv elementul căutat cu fiecare element al șirului:

- dacă elementul curent este egal cu valoarea căutată, căutarea se oprește și se returnează poziția;
- dacă se ajunge la finalul șirului fără a găsi valoarea, rezultatul indică faptul că elementul nu există în șir.

Exemplu ilustrativ - Se consideră șirul de elemente:

3	7	12	9	15	21

Dacă se caută valoarea 9, algoritmul va compara succesiv această valoare cu:

$$3 \rightarrow 7 \rightarrow 12 \rightarrow 9$$

În momentul în care valoarea este găsită, căutarea se oprește.

Complexitatea algoritmului

Fie n numărul de elemente din șir.

- **Caz favorabil**: elementul căutat se află pe prima poziție $\Rightarrow$ complexitate $O(1)$;
- **Caz mediu**: elementul se află, în medie, la mijlocul șirului $\Rightarrow$ complexitate $O(n)$;

- **Caz nefavorabil**: elementul se află pe ultima poziție sau nu există în șir $\Rightarrow$ complexitate $O(n)$.

Din punct de vedere al memoriei, algoritmul are complexitate $O(1)$, deoarece utilizează un număr constant de variabile auxiliare.

Implementare în limbajul C++

```
int cautareLiniara(int*& a, int n, int x)
{
    for (int i = 0; i < n; i++)
        if (a[i] == x)
            return i; // pozitia elementului gasit
    return -1; // elementul nu a fost gasit
}
```

Funcția `cautareLiniara` returnează poziția elementului căutat dacă acesta există în șir sau valoarea -1 în caz contrar.

Observații:

1. În funcția `cautareLiniara` există parametrul `int*&a` (pointer la `int`, transmis prin referință) care este un pointer la un întreg (`int*`) transmis prin referință (`&`). Aceasta înseamnă că funcția primește acces direct la pointerul original din funcția apelatoare. Orice modificare a adresei stocate de `a` în interiorul funcției s-ar reflecta asupra pointerului original. Deși în această implementare specifică adresa pointerului nu este modificată, permițând doar accesul la elementele șirului `a[i]`, această formă de transmitere sugerează posibilitatea unei modificări a pointerului în sine.

2. Pentru o funcție de căutare liniară care are ca scop exclusiv citirea datelor și găsirea unui element, fără a modifica șirul sau pointerul la șir, o practică de programare mai sigură și mai clară ar fi utilizarea `const int* a`. Utilizarea `const` indică compilatorului și altor programatori că funcția se angajează să nu modifice conținutul memoriei la care pointează `a`. Aceasta previne erorile accidentale și îmbunătățește lizibilitatea și fiabilitatea codului.

2. Căutarea binară

Căutarea binară este o metodă eficientă de căutare a unui element într-un șir de date, bazată pe principiul *Divide et Impera*. Spre deosebire de căutarea liniară, această metodă necesită ca șirul de date să fie **ordonat** (crescător sau descrescător).

Ideea principală a căutării binare constă în compararea elementului căutat cu elementul aflat la mijlocul șirului. În funcție de rezultat, căutarea continuă doar într-una dintre jumătățile șirului, eliminând astfel o parte semnificativă a datelor la fiecare pas.

Mod de funcționare - Algoritmul de căutare binară urmează pașii:

1. Se stabilește intervalul inițial de căutare, delimitat de prima și ultima poziție.
2. Se determină poziția de mijloc a intervalului.
3. Dacă elementul de pe poziția de mijloc este egal cu valoarea căutată, algoritmul se oprește.
4. Dacă valoarea căutată este mai mică, căutarea continuă în jumătatea stângă.
5. Dacă valoarea căutată este mai mare, căutarea continuă în jumătatea dreaptă.
6. Pașii se repetă până la găsirea elementului sau până la epuizarea intervalului.

Exemplu ilustrativ - Se consideră șirul ordonat:

3	7	9	12	15	21

Pentru a căuta valoarea 9:

- elementul de mijloc este $12 \Rightarrow$ se continuă în jumătatea stângă;
- noul element de mijloc este $9 \Rightarrow$ valoarea a fost găsită.

Complexitatea algoritmului

Fie n numărul de elemente din șir.

- **Caz favorabil**: elementul căutat se află chiar la mijloc $\Rightarrow$ complexitate $O(1)$ (timp constant);

- **Caz mediu**: la fiecare pas intervalul se înjumătățește $\Rightarrow$ complexitate $O(\log_2 n)$;
- **Caz nefavorabil**: elementul nu există în șir $\Rightarrow$ complexitate $O(\log_2 n)$.

Din punct de vedere al memoriei, căutarea binară iterativă are complexitate $O(1)$.

Implementare în limbajul C++

```
int cautareBinaraIterativa(int* a, int n, int x)
{
    int st = 0, dr = n - 1;

    while (st <= dr){
        int mij = (st + dr) / 2;

        if (a[mij] == x)
            return mij;
        else if (x < a[mij])
            dr = mij - 1;
        else
            st = mij + 1;
    }

    return -1;
}

int cautareBinaraRecursiva(int* a, int st, int dr, int x)
{
    if (st > dr)
        return -1;

    int mij = (st + dr) / 2;

    if (a[mij] == x)
        return mij;
    else if (x < a[mij])
        return cautareBinaraRecursiva(a, st, mij - 1, x);
    else
        return cautareBinaraRecursiva(a, mij + 1, dr, x);
}
```

Funcțiile `cautareBinaraIterativa` și `cautareBinaraRecursiva` returnează poziția elementului căutat dacă acesta există în șirul ordonat sau valoarea -1 în caz contrar.

Pentru mai multe detalii despre paradigma *Divide et Impera*, dar și despre algoritmul de căutare binară, vezi secțiunea de Divide et Impera.

3. Căutarea prin interpolare

Căutarea prin interpolare [10] este o metodă de căutare eficientă pentru șiruri de date **ordonate**, care îmbunătățește ideea căutării binare prin estimarea poziției probabile a elementului căutat. Această metodă este deosebit de eficientă atunci când valorile din șir sunt distribuite uniform.

Spre deosebire de căutarea binară, care verifică întotdeauna elementul din mijlocul intervalului, căutarea prin interpolare încearcă să determine o poziție mai apropiată de cea reală a elementului căutat, folosind o formulă de estimare.

Mod de funcționare - Algoritmul urmează pașii:

1. Șirul trebuie să fie ordonat crescător.
2. Se stabilește intervalul inițial de căutare, delimitat de prima și ultima poziție.
3. Se estimează poziția elementului căutat folosind o formulă de interpolare.
4. Dacă elementul estimat este egal cu valoarea căutată, căutarea se oprește.
5. În caz contrar, intervalul de căutare se restrânge corespunzător.
6. Pașii se repetă până la găsirea elementului sau epuizarea intervalului.

Exemplu de formula de interpolare

În căutarea prin interpolare, poziția probabilă a elementului căutat este estimată folosind relația:

$$poz = st + \frac{(dr - st) \cdot (x - a[st])}{a[dr] - a[st]}$$

unde:

- st reprezintă indicele de început al intervalului de căutare;
- dr reprezintă indicele de sfârșit al intervalului de căutare;
- x este valoarea căutată;
- $a[st]$ este valoarea minimă din interval;

- $a[dr]$ este valoarea maximă din interval.

Formula se bazează pe ipoteza că valorile din șir sunt distribuite aproximativ uniform. Astfel, poziția estimată este proporțională cu distanța relativă a valorii căutate față de valorile extreme ale intervalului.

De exemplu, dacă valoarea x este apropiată de $a[st]$, poziția estimată va fi mai apropiată de st, iar dacă valoarea este apropiată de $a[dr]$, poziția estimată va fi mai apropiată de dr.

Exemplu ilustrativ - Se consideră șirul ordonat:

3	7	9	12	15	21

Pentru a căuta valoarea 15, poziția estimată va fi mai apropiată de partea dreaptă a șirului, reducând numărul de comparații necesare.

Complexitatea algoritmului

Fie n numărul de elemente din șir.

- **Caz favorabil**: distribuție uniformă a valorilor $\Rightarrow$ complexitate medie $O(\log_2 \log_2 n)$ (timp log-logaritmic);
- **Caz mediu**: date relativ uniform distribuite $\Rightarrow$ complexitate $O(\log_2 \log_2 n)$;
- **Caz nefavorabil**: distribuție neuniformă $\Rightarrow$ complexitate $O(n)$.

Complexitatea spațială este $O(1)$, deoarece se utilizează un număr constant de variabile auxiliare.

Implementare în limbajul C++

```
int cautareInterpolare(int* a, int n, int x){
    int st = 0, dr = n - 1;
    while (st <= dr && x >= a[st] && x <= a[dr]){
        if (a[st] == a[dr]){
            if (a[st] == x)
                return st;
            else
                return -1;
        }
```

```
        int poz = st + (double)(dr - st) * (x - a[st]) / (a[dr] -
           a[st]);

        if (a[poz] == x)
            return poz;
        else if (x < a[poz])
            dr = poz - 1;
        else
            st = poz + 1;
    }

    return -1;
}
```

Funcția `cautareInterpolare` returnează poziția elementului căutat dacă acesta există în șirul ordonat sau valoarea -1 în caz contrar.

4. Căutarea prin salturi (Jump Search)

Căutarea prin salturi - *jump search* [11] - este o metodă intermediară între căutarea liniară și cea binară, destinată șirurilor ordonate. Algoritmul constă în verificarea elementelor la intervale regulate (salturi) și apoi efectuarea unei căutări liniare într-un bloc restrâns atunci când se detectează că elementul căutat poate fi în acel interval.

Această metodă reduce numărul de comparații față de căutarea liniară, fiind utilă mai ales pentru șiruri mari, unde accesul secvențial complet ar fi costisitor.

Mod de funcționare - Algoritmul urmează pașii:

1. Se alege un pas de salt m, de obicei aproximativ $\sqrt{n}$, unde n este dimensiunea șirului.
2. Se parcurge șirul sărind câte m elemente până când elementul curent depășește valoarea căutată sau se ajunge la final.
3. Se efectuează o căutare liniară în blocul precedent pentru a găsi exact poziția elementului.

Exemplu ilustrativ - Se consideră șirul ordonat:

1	3	5	7	9	12	15	18	20	21

Dacă se caută valoarea 15 și se alege pasul $m = \sqrt{15} = 3$, algoritmul va parcurge elementele:

$$1 \to 7 \to 12 \to 18 > 15$$

Apoi se efectuează o căutare liniară în blocul dintre 12 și 18, găsind elementul 15.

Complexitatea algoritmului

Fie n numărul de elemente din șir și $m \approx \sqrt{n}$ pasul de salt:

- Complexitatea în cel mai bun caz: $O(1)$, dacă elementul se află la prima verificare;
- Complexitatea medie și în cel mai rău caz: $O(\sqrt{n})$, deoarece parcurgerea se face prin salturi și apoi liniar într-un bloc de dimensiune m.

Complexitatea spațială este $O(1)$.

Implementare în limbajul C++

```
int cautareSalturi(int* a, int n, int x)
{
    int m = sqrt(n); // dimensiunea pasului
    int st = 0, dr = 0;

    while (dr < n && a[dr] < x){
        st = dr;
        dr += m;
        if (dr > n - 1)
            dr = n - 1;
    }
    for (int i = st; i <= dr; i++){
        if (a[i] == x)
            return i;
    }
    return -1;
}
```

Funcția `cautareSalturi` returnează poziția elementului căutat dacă acesta există în șirul ordonat sau valoarea -1 în caz contrar.

5. Căutarea exponențială (Exponential Search)

Căutarea exponențială este o metodă eficientă de căutare într-un șir **ordonat**, utilă în special pentru șiruri foarte mari sau când elementul căutat se află aproape de începutul șirului. Algoritmul combină creșterea rapidă a intervalului de căutare cu o căutare binară într-un subinterval restrâns.

Ideea principală este de a găsi rapid un interval care să conțină elementul căutat, apoi de a aplica căutarea binară în acel interval.

Mod de funcționare - Algoritmul urmează pașii:

1. Se verifică primul element; dacă este egal cu elementul căutat, căutarea se încheie.
2. Se crește exponentul i pentru a identifica un interval de tip $[2^{i-1}, 2^i]$ care conține elementul căutat.
3. Se aplică căutarea binară în intervalul determinat.

Exemplu ilustrativ - Se consideră șirul ordonat:

1	3	5	7	9	12	15	21

Dacă se caută valoarea 12:

- primul element este $1 \neq 12$;
- se testează $a[1] = 3 < 12$, $a[2] = 5 < 12$, $a[4] = 9 < 12$, $a[8]$ depășește limita șirului;
- intervalul $[4, 7]$ conține elementul, se aplică căutarea binară în acest interval;
- elementul 12 este găsit la poziția 5.

Complexitatea algoritmului

Fie n numărul de elemente din șir.

- Complexitatea în cel mai bun caz: $O(1)$, dacă elementul se află la început;
- Complexitatea medie și în cel mai rău caz: $O(\log_2 i) + O(\log_2 n)$, unde i este poziția elementului găsit;

- Complexitatea spațială este $O(1)$.

Implementare în limbajul C++

```
int cautareExponentiala(int a[], int n, int x)
{
    if (a[0] == x)
        return 0;

    int i = 1;
    while (i < n && a[i] <= x)
        i *= 2;

    int st = i / 2;
    int dr = (i < n) ? i : n - 1; // i sau ultima pozitie din
       vector daca aceasta este depasita

    return cautareBinara(a, st, dr, x);
}
```

Funcția `cautareExponentiala` returnează poziția elementului căutat dacă acesta există în șirul ordonat sau valoarea -1 în caz contrar.

8.1.2 Sortări

Introducere

Problemele de sortare reprezintă o categorie esențială în informatică și în prelucrarea datelor. Ele presupun aranjarea elementelor unei colecții (șiruri, liste, vectori) într-o anumită ordine, de obicei crescătoare sau descrescătoare, astfel încât operațiile ulterioare asupra datelor să fie mai eficiente și mai ușor de realizat.

Sortarea apare frecvent în aplicații practice, cum ar fi gestionarea bazelor de date, procesarea informațiilor numerice sau alfanumerice, căutarea rapidă a elementelor, generarea de rapoarte și optimizarea unor algoritmi care depind de ordinea datelor.

În mod general, o problemă de sortare presupune:

- existența unui set de date neordonat;
- definirea criteriului de ordine;
- aplicarea unei metode care să rearanjeze datele conform criteriului stabilit.

Studiul sortării este important deoarece eficiența operațiilor care urmează sortării depinde adesea de modul în care datele sunt organizate. Alegerea unei strategii adecvate de sortare poate reduce semnificativ timpul de execuție și consumul de resurse, în special atunci când volumul de date este mare.

În cadrul acestui subcapitol, vor fi abordate conceptele fundamentale legate de sortare, analiza complexității generale și criteriile de performanță ale algoritmilor, fără a se intra încă în metode specifice de sortare.

1. Sortarea prin metoda bulelor (Bubble Sort)

Sortarea prin metoda bulelor [3] este unul dintre cei mai simpli și intuitivi algoritmi de sortare. Ideea principală constă în parcurgerea repetată a șirului de date și compararea elementelor adiacente, interschimbându-le dacă sunt în ordine greșită, astfel încât elementele „mai mari" să „plutească" treptat către capătul șirului.

Deși metoda nu este eficientă pentru șiruri mari, aceasta este utilă pentru înțelegerea principiilor de bază ale sortării și pentru aplicații cu volum redus de date.

Mod de funcționare - Algoritmul parcurge șirul de mai multe ori, repetând următorii pași:

1. Se compară două elemente consecutive.
2. Dacă ele nu sunt în ordinea dorită, se interschimbă.
3. Se continuă procesul până la sfârșitul șirului.
4. Pașii se repetă pentru șirul rămas nesortat, reducând treptat intervalul de analiză.

Exemplu ilustrativ - Se consideră șirul:

5	3	8	4	2

Prima trecere:

$$35842 \quad \Rightarrow \quad 35482 \quad \Rightarrow \quad 35428$$

A doua trecere:

$$3\,4\,5\,2\,8 \quad \Rightarrow \quad 3\,4\,2\,5\,8$$

Și așa mai departe, până când șirul devine:

$$2\,3\,4\,5\,8$$

Complexitatea algoritmului

Fie n numărul de elemente din șir:

- **Caz favorabil**: șirul este deja sortat $\Rightarrow$ complexitate $O(n)$ (dacă se implementează optimizare cu verificare swap);
- **Caz mediu**: elementele sunt amestecate $\Rightarrow$ complexitate $O(n^2)$;
- **Caz nefavorabil**: șirul este sortat descrescător $\Rightarrow$ complexitate $O(n^2)$.

Complexitatea spațială este $O(1)$, deoarece se utilizează doar un număr constant de variabile auxiliare pentru interschimbări.

Implementare în limbajul C++

```
void bubbleSort(int* vector, int dimensiune)
{
    bool interschimbare; // se va verifica daca s-au facut
       schimbari
    for (int i = 0; i < dimensiune - 1; i++){
        interschimbare = false; // presupunem ca este deja sortat
        for (int j = 0; j < dimensiune - i - 1; j++){
            if (vector[j] > vector[j + 1]){
                swap(vector[j], vector[j + 1]);
                interschimbare = true; // presupunerea nu a fost
                   corecta
            }
        }
        if (!interschimbare)
            break; // sirul este deja sortat
    }
}
```

Funcția `bubbleSort` sortează șirul `vector` în ordine crescătoare. Variabila `swapped` permite optimizarea algoritmului, întrerupând procesul dacă nu s-au efectuat interschimbări într-o trecere.

2. Sortarea prin inserție (Insertion Sort)

Sortarea prin inserție [3] este un algoritm intuitiv care construiește treptat un șir sortat prin inserarea fiecărui element din șirul nesortat în poziția corectă din șirul deja sortat.

Această metodă este eficientă pentru șiruri mici sau aproape sortate și are avantajul de a fi simplu de implementat și de înțeles.

Mod de funcționare - Algoritmul parcurge șirul element cu element, efectuând următorii pași:

1. Se consideră primul element ca fiind deja sortat.
2. Pentru fiecare element următor, se compară cu elementele din șirul sortat și se plasează la poziția corespunzătoare.
3. Elementele mai mari sunt deplasate cu o poziție spre dreapta pentru a face loc inserției.

Exemplu ilustrativ - Se consideră șirul:

5	3	8	4	2

Pas cu pas:

- elementul 3 se inserează înaintea lui 5: $[3, 5, 8, 4, 2]$;
- elementul 8 rămâne pe loc: $[3, 5, 8, 4, 2]$;
- elementul 4 se inserează între 3 și 5: $[3, 4, 5, 8, 2]$;
- elementul 2 se inserează la început: $[2, 3, 4, 5, 8]$.

Complexitatea algoritmului

Fie n numărul de elemente din șir:

- **Caz favorabil**: șirul este deja sortat $\Rightarrow$ complexitate $O(n)$;

- **Caz mediu**: elementele sunt amestecate ⇒ complexitate $O(n^2)$;
- **Caz nefavorabil**: șirul este sortat descrescător ⇒ complexitate $O(n^2)$.

Complexitatea spațială este $O(1)$, deoarece nu se utilizează spațiu auxiliar semnificativ.

Implementare în limbajul C++

```
void insertionSort(int* vector, int dimensiune){
    int cheie, j;
    for (int i = 1; i < dimensiune; i++){
        cheie = vector[i];
        j = i - 1;

        // deplaseaza elementele mai mari cu o pozitie spre dreapta
        while (j >= 0 && vector[j] > cheie){
            vector[j + 1] = vector[j];
            j--;
        }
        vector[j + 1] = cheie;
    }
}
```

Funcția `insertionSort` sortează șirul `vector` în ordine crescătoare. Algoritmul construiește treptat șirul sortat prin deplasarea elementelor mai mari și inserarea fiecărui element la poziția corespunzătoare.

3. Sortarea prin selecție (Selection Sort)

Sortarea prin selecție [10] este un algoritm simplu care sortează un șir prin selectarea repetată a celui mai mic (sau celui mai mare) element din șirul nesortat și plasarea acestuia la poziția corespunzătoare în șirul sortat. Algoritmul este ușor de implementat și de înțeles, deși nu este eficient pentru șiruri mari.

Mod de funcționare - Algoritmul parcurge șirul element cu element și efectuează următorii pași:

1. Se caută elementul minim din șirul nesortat.
2. Se interschimbă acest element cu primul element nesortat.

3. Se repetă procesul pentru restul șirului nesortat, reducând treptat intervalul.

Exemplu ilustrativ - Se consideră șirul:

5	3	8	4	2

Pas cu pas:

- elementul minim 2 se mută pe prima poziție: $[2, 3, 8, 4, 5]$;
- elementul minim 3 este deja la poziția corectă: $[2, 3, 8, 4, 5]$;
- elementul minim 4 se mută înaintea lui 8: $[2, 3, 4, 8, 5]$;
- elementul minim 5 se mută înaintea lui 8: $[2, 3, 4, 5, 8]$.

Complexitatea algoritmului

Fie n numărul de elemente din șir:

- **Caz favorabil**: elementele sunt deja sortate $\Rightarrow$ complexitate $O(n^2)$;
- **Caz mediu**: elementele sunt amestecate $\Rightarrow$ complexitate $O(n^2)$;
- **Caz nefavorabil**: șirul este sortat descrescător $\Rightarrow$ complexitate $O(n^2)$.

Complexitatea spațială este $O(1)$, deoarece algoritmul folosește un număr constant de variabile auxiliare pentru interschimbări.

Implementare în limbajul C++

```
void selectionSort(int* vector, int dimensiune)
{
    int indexMinim;
    for (int i = 0; i < dimensiune - 1; i++){
        indexMinim = i;
        for (int j = i + 1; j < dimensiune; j++){
            if (vector[j] < vector[indexMinim])
                indexMinim = j;
        }
        if (indexMinim != i)
            swap(vector[i], vector[indexMinim]);
    }
}
```

Funcția `selectionSort` sortează șirul `vector` în ordine crescătoare prin selectarea repetată a celui mai mic element și mutarea lui la poziția corespunzătoare.

4. Sortarea rapidă (Quick Sort)

Sortarea rapidă este un algoritm de sortare prin comparație extrem de eficient, bazat pe paradigma *Divide et Impera*. Este recunoscut pentru performanța sa rapidă în practică, fiind adesea considerat unul dintre cei mai buni algoritmi de sortare generali. Principiul său fundamental implică selectarea unui element *pivot* și partiționarea șirului în două sub-șiruri: elementele mai mici decât pivotul și elementele mai mari. Acest proces este aplicat recursiv sub-șirurilor, până când întregul șir este sortat. Pentru o analiză detaliată a funcționării și implementării acestui algoritm, inclusiv complexitatea sa, consultați secțiunea dedicată din capitolul de Divide et Impera - QuickSort.

5. Sortarea prin interclasare (Merge Sort)

Sortarea prin interclasare este un alt algoritm de sortare eficient, bazat tot pe strategia *Divide et Impera*. Spre deosebire de Quick Sort, Merge Sort garantează o performanță consistentă, fiind stabil și având o complexitate în timp $O(n \log n)$ în toate cazurile (favorabil, mediu, nefavorabil). Principiul său constă în împărțirea recursivă a șirului în două jumătăți, sortarea fiecărei jumătăți independent, și apoi interclasarea (combinarea) celor două jumătăți sortate într-un singur șir sortat. Pentru o înțelegere aprofundată a mecanismului de funcționare, a implementării și a analizei complexității acestui algoritm, consultați secțiunea dedicată din capitolul de Divide et Impera - MergeSort.

6. Sortarea prin numărare (Counting Sort)

Counting Sort este o sortare potrivită atunci când valorile din vector se află într-un interval relativ mic. Counting Sort nu compară direct elementele între ele, ci folosește un vector auxiliar `count` în care reține de câte ori apare fiecare valoare. Pentru fiecare element `x`, poziția `x - minim` din `count` este incrementată. Scăderea lui `minim` permite algoritmului să funcționeze și atunci când valorile sunt negative. După numărare, parcurgem vectorul `count` și reconstruim vectorul inițial în ordine crescătoare.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

void countingSort(std::vector<int>& v) {
    if (v.empty()) {
        return;
    }

    int minim = *std::min_element(v.begin(), v.end());
    int maxim = *std::max_element(v.begin(), v.end());

    int interval = maxim - minim + 1;
    std::vector<int> count(interval, 0);

    for (int x : v) {
        count[x - minim]++;
    }

    int k = 0;

    for (int i = 0; i < interval; i++) {
        while (count[i] > 0) {
            v[k++] = i + minim;
            count[i]--;
        }
    }
}

int main() {
    std::vector<int> vector = { 4, 2, 2, 8, 3, 3, 1 };

    countingSort(vector);

    for (int element : vector)
        std::cout << element << " ";

    return 0;
}
```

Cadrul în care se folosește Counting Sort este relativ limitat. Este avantajos când elementele sunt numere întregi și diferența dintre valoarea maximă și valoarea minimă nu este foarte mare. De exemplu, pentru valori între 0 și 100, algoritmul este foarte potrivit. Dacă avem însă valori precum 1, 1000000 și 2000000000, vectorul count ar deveni enorm și algoritmul nu mai este practic.

Dacă n este numărul de elemente și k este dimensiunea intervalului de valori, complexitatea temporală este $O(n + k)$, iar complexitatea spațială este $O(k)$. În cazul în care k este mic în raport cu n, Counting Sort poate fi mai rapid decât algoritmi de sortare bazați pe comparații, cum ar fi std::sort, care are complexitate $O(n \log n)$ în cazul general.

Alte metode de sortare

În plus față de sortările descrise anterior, mai există o serie de metode de sortare care sunt utile în anumite situații, în special atunci când datele au anumite proprietăți care pot fi exploatate pentru a obține o complexitate mai bună decât $O(n \log n)$.

Radix Sort [3] sortează elementele pe baza cifrelor lor, de obicei începând cu cifra cea mai puțin semnificativă. La fiecare etapă se folosește o sortare stabilă, de exemplu Counting Sort. Este potrivit pentru numere întregi sau șiruri de caractere cu o lungime controlată. Dacă d este numărul de cifre și k este baza utilizată, complexitatea este $O(d(n + k))$, iar memoria suplimentară este $O(n + k)$. Pentru numere cu un număr limitat de cifre poate fi foarte eficient.

Bucket Sort [3] împarte elementele în mai multe „găleți" în funcție de valoarea lor, sortează separat fiecare găleată și apoi le concatenează. Este eficient atunci când valorile sunt distribuite relativ uniform într-un interval cunoscut. În cazul mediu, complexitatea poate ajunge la $O(n+k)$, însă în cazul nefavorabil poate fi $O(n^2)$, în funcție de metoda folosită pentru sortarea găleților și de distribuția datelor. Memoria suplimentară este de ordinul $O(n + k)$.

Shell Sort [10] este o îmbunătățire a Insertion Sort care compară elemente aflate la anumite distanțe și reduce treptat această distanță până când ajunge la 1. Este util pentru vectori de dimensiuni moderate și nu necesită memorie suplimentară semnificativă. Complexitatea depinde de secvența de pași aleasă; în cazul unei implementări clasice poate ajunge la $O(n^2)$, dar în practică este adesea mult mai rapid decât algoritmii simpli precum **Bubble Sort**, **Insertion Sort** și **Selection Sort**.

Cocktail Shaker Sort [10] este o variantă a **Bubble Sort** în care parcurgerea se face alternativ de la stânga la dreapta și de la dreapta la stânga. Astfel, valorile mari sunt deplasate spre final, iar valorile mici spre început. Este potrivit mai ales pentru scopuri didactice sau pentru vectori mici. Complexitatea în cazul mediu și nefavorabil este $O(n^2)$, iar în cazul favorabil poate fi $O(n)$ dacă vectorul este deja sortat și se detectează acest lucru.

Comb Sort [12] este o îmbunătățire a **Bubble Sort** care compară inițial elemente aflate la distanță și reduce treptat această distanță. Ideea este de a elimina mai rapid elementele foarte mici aflate spre sfârșitul vectorului, problemă care încetinește **Bubble Sort**. Complexitatea în cazul nefavorabil este $O(n^2)$, dar în practică poate fi mai rapid decât **Bubble Sort** și este simplu de implementat.

Tim Sort [13] este o combinație între **Merge Sort** și **Insertion Sort** și este conceput pentru a exploata secvențele care sunt deja parțial sortate. Este folosit în implementări reale ale unor limbaje și biblioteci deoarece oferă performanțe foarte bune pe date reale. Complexitatea este $O(n \log n)$ în cazul nefavorabil, iar pentru date deja sau parțial sortate poate ajunge la $O(n)$. Necesită memorie suplimentară.

Intro Sort [14] combină **Quick Sort**, **Heap Sort** și **Insertion Sort**. Începe de obicei cu **Quick Sort**, dar dacă recursivitatea devine prea mare, trece la Heap Sort pentru a evita cazul nefavorabil al **Quick Sort**. Pentru subvectori mici poate folosi **Insertion Sort**. Are complexitate $O(n \log n)$ în cazul nefavorabil și folosește memorie suplimentară redusă. Este o abordare practică deoarece combină avantajele mai multor algoritmi.

În concluzie, alegerea algoritmului depinde foarte mult de natura datelor. Pentru sortarea generală a unor valori, algoritmi precum **Quick Sort**, **Merge Sort**, **Heap Sort** sau algoritmi hibrizi precum **Intro Sort** și **Tim Sort** sunt alegeri bune. Pentru numere întregi cu un interval restrâns, **Counting Sort** poate fi foarte eficient, iar pentru numere cu un număr controlat de cifre se poate folosi **Radix Sort**. **Bucket Sort** este avantajos atunci când valorile sunt distribuite relativ uniform. **Shell Sort**, **Comb Sort** și **Cocktail Shaker Sort** sunt mai degrabă algoritmi specializați sau didactici, fiind utili pentru a înțelege diferite tehnici de îmbunătățire a sortării.

8.2 Probleme rezolvate

În această secțiune vor fi prezentate o serie de probleme care au la bază sortări. În principiu se recomandă să se folosească metode eficiente - acestea vor fi prezentate în detaliu la capitolul de Divide et Impera. În cazul ăn care pentru rezolvare nu s-a ales deja o metodă specifică de sortare, vom folosi sortarea din STL std::sort.

1. Se consideră 2 vectori. Să se verifice dacă primul vector conține al doilea vector sau nu. Se discută variante pentru complexitate ceruta $O(m \cdot n)$ sau $O(m + n)$ unde m este

numărul de elemente din al doilea vector iar n este dimensiunea pentru primul vector.

Rezolvare:

Date de intrare: Se citesc doi vectori de numere întregi:

- vectorul A cu n elemente;
- vectorul B cu m elemente.

Date de ieșire: Se afișează un mesaj care indică dacă vectorul A conține vectorul B ca subsecvență continuă:

- `DA` – dacă B este conținut în A;
- `NU` – în caz contrar.

Algoritm de rezolvare: Există două variante principale:

(a) Varianta naivă ($O(m \cdot n)$): pentru fiecare poziție posibilă din vectorul A, se verifică element cu element dacă secvența începută în acea poziție este identică cu vectorul B.

(b) Varianta optimizată ($O(m+n)$): se folosește un algoritm de tip KMP (Knuth–Morris–Pratt), adaptat pentru vectori, care construiește un vector de prefixe pentru B și permite parcurgerea eficientă a vectorului A fără reveniri inutile.

Algoritm scris in C++:

```
#include <iostream>
#include <vector>

std::vector<int> prefixFunction(const std::vector<int>& B) {
    int m = B.size();
    std::vector<int> pi(m, 0);
    int k = 0;
    for (int i = 1; i < m; i++) {
        while (k > 0 && B[i] != B[k])
            k = pi[k - 1];
        if (B[i] == B[k])
            k++;
        pi[i] = k;
    }
```

```
    return pi;
}

bool contains(const std::vector<int>& A, const
   std::vector<int>& B) {
    int n = A.size(), m = B.size();
    std::vector<int> pi = prefixFunction(B);
    int k = 0;
    for (int i = 0; i < n; i++) {
        while (k > 0 && A[i] != B[k])
            k = pi[k - 1];
        if (A[i] == B[k])
            k++;
        if (k == m)
            return true;
    }
    return false;
}

int main() {
    int n, m;
    std::cin >> n;
    std::vector<int> A(n);
    for (int i = 0; i < n; i++)
        std::cin >> A[i];

    std::cin >> m;
    std::vector<int> B(m);
    for (int i = 0; i < m; i++)
        std::cin >> B[i];

    if (contains(A, B))
        std::cout << "DA";
    else
        std::cout << "NU";
    return 0;
}
```

Justificarea soluției propuse: Algoritmul verifică toate pozițiile posibile din vectorul A unde ar putea începe vectorul B. În varianta KMP, vectorul de prefixe garantează că, la apariția unei nepotriviri, se revine la cea mai lungă potrivire parțială validă, fără a omite vreo posibilitate. Astfel, dacă există o subsecvență identică cu B în A, aceasta va fi detectată.

Analiza complexității:

- *Complexitate temporală*:
  - Varianta naivă: $O(m \cdot n)$;
  - Varianta optimizată (KMP): $O(m + n)$.
- *Complexitate spațială*:
  - Varianta naivă: $O(1)$ spațiu suplimentar;
  - Varianta KMP: $O(m)$ pentru vectorul de prefixe.

2. Se dă un șir de numere întregi, pozitive și negative. Să se găsească cele două elemente pentru care suma lor este cea mai apropiată de 0.

Rezolvare:
Date de intrare: Se citește un număr întreg n, reprezentând dimensiunea șirului, urmat de n numere întregi (pozitive și/sau negative).
Date de ieșire: Se afișează cele două elemente din șir a căror sumă este cea mai apropiată de 0.

Algoritm de rezolvare: Problema poate fi rezolvată eficient folosind sortarea și tehnica celor doi pointeri/indici:

(a) Se sortează șirul de numere în ordine crescătoare.

(b) Se folosesc doi indici: unul la începutul șirului și unul la sfârșit.

(c) Se calculează suma elementelor indicate de cei doi indici/indici.

(d) Dacă suma este mai apropiată de 0 decât suma minimă curentă, se actualizează soluția.

(e) Dacă suma este negativă, se deplasează indicele din stânga spre dreapta; dacă suma este pozitivă, se deplasează indicele din dreapta spre stânga.

(f) Algoritmul se oprește când cei doi indici se întâlnesc.

Algoritm scris in C++:

```
#include<iostream>
#include<vector>
#include<algorithm>

int main() {
    int n;
    std::cin >> n;
    std::vector<int> v(n);
    for (int i = 0; i < n; i++)
        std::cin >> v[i];

    std::sort(v.begin(), v.end());
    int left = 0, right = n - 1;
    int bestSum = INT_MAX;
    int x = 0, y = 0;

    while (left < right) {
        int sum = v[left] + v[right];
        if (abs(sum) < abs(bestSum)) {
            bestSum = sum;
            x = v[left];
            y = v[right];
        }
        if (sum < 0)
            left++;
        else
            right--;
    }
    std::cout << x << " " << y;
    return 0;
}
```

Justificarea soluției propuse: După sortare, pentru orice pereche de elemente, deplasarea controlată a celor doi pointeri asigură explorarea tuturor combinațiilor relevante. Dacă suma curentă este negativă, creșterea valorii elementului mai mic poate aduce suma mai aproape de 0. Dacă suma este pozitivă, micșorarea valorii elementului mai mare are același efect. Astfel, algoritmul identifică perechea cu suma minimă în valoare absolută.

Analiza complexitatii:

- *Complexitate temporală*:
  - Sortarea șirului: $O(n \log n)$;

  - Parcurgerea cu doi pointeri: $O(n)$;
  - Complexitatea totală: $O(n \log n)$.
- *Complexitate spațială*: Spațiu suplimentar: $O(1)$ (în afara spațiului necesar pentru stocarea vectorului).

3. Se dă un vector de numere întregi pozitive de dimensiune dim. Să se sorteze elementele de tip palindrom din vectorul dat.

Rezolvare:

Date de intrare: Se citește un număr întreg dim, urmat de un vector de dim numere întregi pozitive

Date de ieșire: Se afișează vectorul inițial modificat astfel încât:

- elementele palindrom sunt ordonate crescător între ele;
- elementele care nu sunt palindromuri își păstrează pozițiile relative față de celelalte ne-palindromuri.

Algoritm de rezolvare:

(a) Se definește o funcție care verifică dacă un număr este palindrom.

(b) Se definește o funcție de comparator care compară două elemente doar dacă ambele sunt palindromuri.

(c) Se implementează algoritmul QuickSort, care primește ca parametru funcția de comparator.

(d) În timpul sortării, comparatorul decide ordinea doar pentru elementele palindromuri.

Algoritm scris în C++:

```
#include<iostream>
#include<vector>

bool estePalindrom(int x) {
    int oglindit = 0, copie = x;
    while (copie > 0) {
```

```
        oglindit = oglindit * 10 + copie % 10;
        copie /= 10;
    }
    return oglindit == x;
}

bool comparatorPalindrom(int a, int b) {
    if (estePalindrom(a) && estePalindrom(b))
        return a < b;
    return false;
}

int partition(std::vector<int>& v, int low, int high,
    bool (*cmp)(int, int)) {
    int pivot = v[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (cmp(v[j], pivot)) {
            i++;
            std::swap(v[i], v[j]);
        }
    }
    std::swap(v[i + 1], v[high]);
    return i + 1;
}

void quicksort(std::vector<int>& v, int low, int high,
    bool (*cmp)(int, int)) {
    if (low < high) {
        int pi = partition(v, low, high, cmp);
        quicksort(v, low, pi - 1, cmp);
        quicksort(v, pi + 1, high, cmp);
    }
}

int main() {
    int dim;
    std::cin >> dim;
    std::vector<int> v(dim);
    for (int i = 0; i < dim; i++)
        std::cin >> v[i];
    quicksort(v, 0, v.size() - 1, comparatorPalindrom);
    for (int x : v)
        std::cout << x << " ";
    return 0;
}
```

Justificarea soluției propuse: Funcția `estePalindrom` identifică corect numerele care se citesc identic de la stânga la dreapta și invers. Comparatorul aplică ordonarea doar în cazul în care ambele elemente sunt palindromuri. Algoritmul QuickSort utilizează acest comparator pentru a asigura ordonarea exclusivă a elementelor de tip palindrom, fără a afecta poziționarea relativă a celorlalte valori.

Analiza complexității:

- *Complexitate temporală*:
  - Caz mediu QuickSort: $O(n \log n)$;
  - Caz defavorabil: $O(n^2)$.
- *Complexitate spațială*: Spațiu suplimentar: $O(\log n)$, datorită apelurilor recursive ale algoritmului QuickSort.

4. Se da un vector de numere întregi pozitive de dimensiune dim. Sa se sorteze crescător toate elementele de pe poziții pare din acest sir de numere.

Rezolvare:
Date de intrare: Se citește un număr întreg dim, urmat de un vector de dim numere întregi pozitive.
Date de ieșire: Se afișează vectorul modificat astfel încât:

- elementele aflate pe poziții pare sunt ordonate crescător;
- elementele aflate pe poziții impare își păstrează valorile inițiale și pozițiile.

Algoritm de rezolvare:

(a) Se identifică toate elementele aflate pe poziții pare (considerând indexarea de la 0).

(b) Se extrag aceste elemente într-un vector auxiliar.

(c) Se sortează vectorul auxiliar în ordine crescătoare.

(d) Se reintroduc elementele sortate în vectorul inițial, pe aceleași poziții pare.

Algoritm scris în C++:

```
#include<iostream>
#include<vector>

int main() {
    int dim;
    std::cin >> dim;
    std::vector<int> v(dim);

    for (int i = 0; i < dim; i++)
        std::cin >> v[i];

    std::vector<int> aux;
    for (int i = 0; i < dim; i += 2)
        aux.push_back(v[i]);

    std::sort(aux.begin(), aux.end(), comparatorCrescator);

    int k = 0;
    for (int i = 0; i < dim; i += 2) {
        v[i] = aux[k];
        k++;
    }
    for (int x : v)
        std::cout << x << " ";
    return 0;
}
```

Justificarea soluției propuse: Algoritmul selectează exclusiv elementele de pe poziții pare, le sortează independent și le plasează înapoi pe aceleași poziții. Deoarece elementele de pe poziții impare nu sunt modificate, condiția problemei este respectată, iar ordonarea elementelor pare este corectă și completă.

Analiza complexității:

- *Complexitate temporală*:
  - Extragerea elementelor de pe poziții pare: $O(n)$;
  - Sortarea elementelor pare: $O(\frac{n}{2} \log \frac{n}{2})$;
  - Complexitatea totală: $O(n \log n)$.
- *Complexitate spațială*: Spațiu suplimentar: $O(\frac{n}{2})$, necesar pentru vectorul auxiliar.

8.3 Probleme propuse

*

1. Se dă un vector de numere întregi pozitive de dimensiune dim. Să se sorteze elementele prime din vectorul dat.

2. Se dă un vector de numere întregi pozitive de dimensiune dim. Să se sorteze elementele crescător până la jumătatea șirului, iar de la jumătate la final toate elementele să se sorteze descrescător.

**

1. Sa se testeze dacă un vector de numere întregi dat este sortat crescător. Dacă da, atunci fiind dat un element $elem$, să se insereze elementul la locul lui, adică astfel încât să rămână un șir sortat crescător. Dacă nu este sortat crescător, atunci să se identifice dacă există un eventual element care strică "ordinea", iar dacă exista unul singur, atunci să se elimine acel element.

2. Se dau niște date calendaristice (citite dintr-un fișier text). Să se verifice dacă sunt valide (zi, lună, an) și apoi cele valide să se sorteze crescător.

***

1. Se citește un vector de n numere întregi ordonat descrescător din fișier. Se citește un al doilea vector de m numere întregi din consolă. Folosind căutarea binară să se determine care elemente din al doilea vector apar în primul și pe ce poziții. Rezultatul se va scrie în fișier.

2. Modificați algoritmul de sortare prin inserție astfel încât să folosească căutarea binară pentru a afla poziția unde noua inserare ar trebui să aibă loc.

****

1. Fiind dați doi vectori de dimensiune n și o valoare k, să se reorganizeze elementele din vectori, astfel încât $a[i] + b[i] \geq k$. Dacă cei doi vectori nu se pot reorganiza astfel încât să se poată, atunci se afișează "Nu".

2. Se citește o matrice cu m linii și n coloane. Să se genereze un sir de 100 numere aleatoare și să se afișeze câte dintre aceste elemente sunt pe prima linie a matricei. **Obs:** Se va aplica o operație de sortare asupra liniilor matricei astfel încât prima coloană să fie sortată crescător, ulterior se va apela căutarea binara.

*****

1. Se citește un vector (container) de n numere întregi ordonat descrescător dintr-un fișier. Se citește un al doilea vector de m numere întregi din consolă. Folosind `std::binary_search` să se determine care elemente din al doilea vector apar în primul și să se afișeze pozițiile acestor elemente utilizând `std::lower_bound`. Rezultatul se va scrie în fișier

2. Se dorește sortarea unui vector v de n numere subunitare cu o distribuție uniformă în intervalul $[0, 1)$. **Idee:** se împarte $[0, 1)$ în n subintervale de aceeași lungime (buckets). Pentru fiecare astfel de subinterval se va considera o listă în care se plasează toate elementele din v aflate în subintervalul considerat. (*Bucket-sort* – sortare eficientă pentru numere subunitare uniform distribuite.)

Partea III

Algoritmi eficienți și structuri de prioritizare. Strategii de selecție

Capitolul 9

Tabele de dispersie

9.1 Aspecte teoretice

O problemă importantă este aceea a căutării eficiente într-o mulțime dinamică de date. Într-un vector, căutarea se poate realiza secvențial în complexitate liniară relativ la numărul de elemente n din mulțime, iar dacă elementele sunt sortate, atunci algoritmul căutării binare permite găsirea unui element în complexitate logaritmică, așa cum a fost prezentat în capitolul precedent. Ne referim aici la complexitatea de timp.

Tabelele de dispersie - *hash tables* - propun o abordare nouă, care permite accesul prin cheie, adică prin căutarea unei valori numită cheie, în complexitate de timp constantă în medie. Acest lucru se poate realiza printr-o generalizare a structurii de vector, în care poziția pe care se plasează, respectiv se caută un element este determinată pe baza unei funcții de dispersie (*hash function*).

Considerând o tablă de dispersie stocată într-un tablou unidimensional T de dimensiune m și universul cheilor U, se consideră o funcția:

$$h : U \to [0, m-1]$$

astfel încât o cheie $k \in U$ va fi repartizată în tabelă pe poziția $T(h(k))$.

9.1.1 Funcții de dispersie

În cadrul tabelelor de dispersie, scopul funcțiilor de dispersie este de a asocia fiecărei chei $k \in U$ o poziție în tabelă. În continuare vom discuta cele mai folosite astfel de funcții pentru chei numere naturale, urmând a se prezenta ulterior și cazul cheilor de alt tip.

Dispersie cu metoda diviziunii

Cel mai simplu exemplu de funcție de dispersie pentru numere naturale este funcția *modulo* (restul împărțirii întregi):

$$h : U \to [0, m-1], h(k) = k \text{ modulo } m$$

Vom nota operația *modulo* prin $\%$, ca în C++.

Exemplu: Se consideră o tabelă T de dimensiune $m = 11$ și funcția de dispersie $h(k) = k\%11$. Atunci cheile 3, 15, 7, 21 vor fi plasate în tabelă ca în figura de mai jos, unde pozițiile neocupate s-au marcat prin '*'.

T	*	*	*	3	15	*	*	7	*	*	21
	0	1	2	3	4	5	6	7	8	9	10

În exemplul anterior se observă faptul că, dacă la cheile care au fost deja inserate se dorește și inserarea cheii 26, cu $h(26) = 26\%11 = 4$ apare o problemă, deoarece poziția 4 este deja ocupată cu cheia 15.

Faptul că pentru două chei distincte $k_1 \neq k_2$ obținem $h(k_1) = h(k_2)$ se numește *coliziune*. Există două moduri generale de rezolvare a coliziunilor, care vor fi discutate în secțiunea următoare.

În [3] este discutată problematica reducerii numărului de coliziuni și a repartizării uniforme a cheilor în tabelă. În acest context, se evidențiază faptul că metoda diviziunii este eficientă atunci când dimensiunea m a tabelei este un număr prim.

Dispersie cu metoda multiplicării

De cele mai multe ori este dificil să ne asigurăm că m este număr prim. Din acest motiv, o alternativă mai practică este metoda multiplicării, pentru care formula generală definită în [3] este:

$$h(k) = \lfloor m\,(Ak - \lfloor Ak \rfloor) \rfloor$$

unde $\lfloor x \rfloor$ reprezintă partea întreagă a lui x, iar A este o constantă, $0 < A < 1$. În acest caz modul de alegere a lui m nu influențează modul de repartizare, iar o alegere bună [15] pentru A

este:

$$A = \left(\sqrt{5} - 1\right)/2 \approx 0.618033$$

În practică metoda multiplicativă funcționează cel mai bine atunci când m este o putere a lui 2.

Dispersie cu metoda multiplicare-deplasare

Pentru o dimensiune a tabelei $m = 2^p$, metoda multiplicării poate fi eficientizată din punct de vedere al calculului, folosind operații de multiplicare și deplasare (shiftare) pe biți. Această metodă se numește *multiply-shift* și poate fi implementată prin [8]:

$$h_a(k) = (ak) >> (w - p) \tag{9.1}$$

unde w este numărul de biți folosiți pentru reprezentarea cheilor, de exemplu 32 sau 64. O valoare pentru a este $a = 2654435769$ propusă în [3].

9.1.2 Hashing pentru chei de tip string - algoritmul FNV-1a

În practică, nu toate cheile sunt de tip numeric; unele pot fi șiruri de caractere sau numere reale. Pentru a le repartiza într-o tabelă de dispersie, acestea trebuie mai întâi mapate la un număr natural, printr-o funcție de hash-ing adecvată. De exemplu, un șir de caractere poate fi transformat într-un număr folosind codurile ASCII ale caracterelor. Provocarea constă în combinarea acestor coduri într-un singur număr, astfel încât să evităm depășirea limitelor și coliziunile.

Un algoritm frecvent utilizat pentru această sarcină este FNV [16], a cărui variantă FNV-1a este folosită în implementările bibliotecii standard C++ pentru tipurile $unordered_set$ și $unordered_map$.

În continuare este prezentat algoritmul general pentru FNV-1a, în care constantele FNV_Prime și FNV_Basis depind de numărul de biți pe care este reprezentată valoarea de dispersie rezultată. De exemplu pentru 32 de biți [16]:

$$FNV_Prime = 16777619, FNV_Basis = 2166136261$$

iar pentru 64 de biți:

$$FNV_Prime = 1099511628211, FNV_Basis = 14695981039346656037$$

.

Algorithm 11: FNV-1a-HASH

Input: O cheie key de tip $string$

begin

  $hash_val \leftarrow FNV_Basis$

  for $\forall byte \in key$ **do**

    $hash_val \leftarrow hash_val$ XOR $byte$

    $hash_val \leftarrow hash_val * FNV_prime$

  end

  return $hash_val$

end

Pentru inserarea în tabela de dispersie trebuie considerat $hash_val$ modulo m, unde m este dimensiunea tabelei.

9.1.3 Rezolvarea coliziunilor

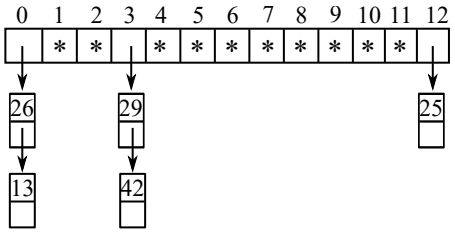
Problema coliziunilor poate fi rezolvată în două moduri principale și anume, prin folosirea de liste înlănțuite sau prin adresare deschisă ce folosește testarea pozițiilor. Ambele metode oferă avantaje și dezavantaje.

Tabele de dispersie cu liste înlănțuite

În cazul acestui tip de tabele de dispersie, fiecare poziție din vectorul ce reprezintă tabela nu va stoca doar un singur element, ci va păstra un pointer către primul element al unei liste înlănțuite, în care se vor memora toate elementele care vor fi repartizate pe poziția respectivă.

Astfel, operația de inserție pe o anumită poziție va fi de fapt o inserție la începutul listei corespunzătoare, iar operațiile de căutare sau de ștergere a unei chei k vor fi de fapt căutare, respectiv ștergere în lista $T[k(k)]$.

Exemplu: Dacă se consideră o tabelă de dispersie T de dimensiune $m = 13$, în care se inserează pe rând cheile 13, 26, 42, 29, 25, rezultatul va arăta ca în figură:



Tabele de dispersie cu adresare deschisă

Tabelele de dispersie cu adresare deschisă nu dispun de liste înlănțuite pentru rezolvarea coliziunilor. Ele stochează pe fiecare poziție un singur element. Dacă la inserarea în tabelă a unei chei, poziția indicată de funcția de dispersie este deja ocupată, se va testa o nouă poziție, acest proces repetându-se până la identificarea unei poziții libere.

O funcție de dispersie pentru tabele cu adresare deschisă este de forma:

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

unde $h(k, t)$ este poziția pentru repartizarea cheii k după t teste.

O condiție necesară pentru o funcție de dispersie într-o astfel de tabelă este aceea, ca pentru o anumită cheie k, pozițiile generate de $h(k, t)$ pentru $t = \overline{0, m-1}$ să reprezinte o permutare a pozițiilor $\{0, 1, \ldots, m-1\}$. În caz contrar, există riscul ca anumite poziții să nu fie testate niciodată.

Observație: Operația de ștergere într-o tabelă cu adresare deschisă este problematică, deoarece întrerupe șirul de testări la căutarea unei chei. Pentru a rezolva această problemă, trebuie marcate pozițiile șterse și ajustați în mod corespunzător algoritmii de inserție și ștergere.

O funcție de dispersie pentru tabele de dispersie cu adresare deschisă se definește pe baza unor funcții de dispersie obișnuite, la care se adaugă componenta de test. Un exemplu este funcția de dublă repartizare $h : U \times \{0, 1, \ldots, m\} \to \{0, 1, \ldots, m\}$, pentru care sunt necesare două funcții de dispersie h_1 și h_2, legate între ele prin formula:

$$h(k, t) = (h_1(k) + t \cdot h_2(k))\%m$$

Satisfacerea condiției ca secvența de poziții testate $\{h(k,0), h(k,1), \ldots, h(k,m-1)\}$ să fie pentru orice cheie k o permutare a pozițiilor $\{0, 1, \ldots, m-1\}$, se poate obține dacă

- $m = 2^p$ și $h_2(k)$ produce mereu un număr impar

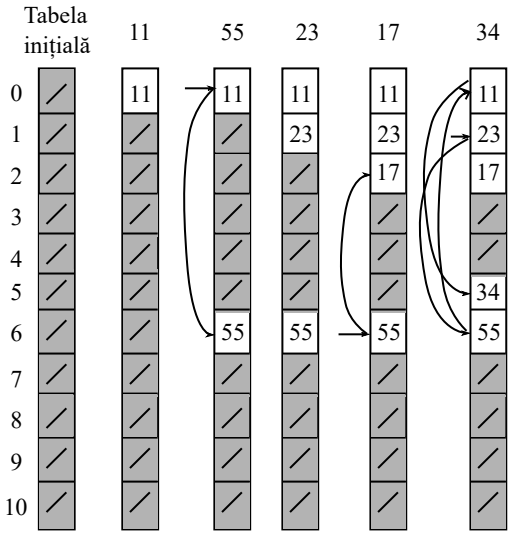
  sau

- m prim și $h_2(k)$ produce numere naturale din $\{0, 1, \ldots, m-1\}$, de exemplu considerăm funcțiile de dispersie

$$h_1(k) = k\%m$$

$$h_2(k) = 1 + (k\%(m-1))$$

Exemplu de inserție: Considerăm o tabelă de dispersie T cu adresare deschisă și dublă repartizare de dimensiune $m = 11$ și funcțiile de dispersie $h_1(k) = k\%m$, $h_2(k) = 1 + k\%(m-1)$. Atunci cheile 11, 55, 23, 17, 34 vor fi inserate astfel:



$h(11,0) = (11\%11 + 0 \cdot (1 + 11\%10))\%11 = 0$

$h(55,0) = (55\%11 + 0 \cdot (1 + 55\%10))\%11 = 0$ - ocupat

$h(55,1) = (55\%11 + 1 \cdot (1 + 55\%10))\%11 = 6$

$h(23,0) = (23\%11 + 0 \cdot (1 + 23\%10))\%11 = 1$

$h(17,0) = (17\%11 + 0 \cdot (1 + 17\%10))\%11 = 6$ - ocupat

$h(17,1) = (17\%11 + 1 \cdot (1 + 17\%10))\%11 = (6 + 8)\%11 = 3$

$h(34,0) = (34\%11 + 0 \cdot (1 + 34\%10))\%11 = 1$ - ocupat

$h(34,1) = (34\%11 + 1 \cdot (1 + 34\%10))\%11 = (1 + 5)\%11 = 6$ - ocupat

$h(34, 2) = (34\%11 + 2 \cdot (1 + 34\%10))\%11 = (1 + 10)\%11 = 0$ - ocupat
$h(34, 3) = (34\%11 + 3 \cdot (1 + 34\%10))\%11 = (1 + 15)\%11 = 5$

9.1.4 Factor de încărcare și complexitatea operațiilor

Se numește factor de încărcare al unei tabele T de dimensiune m, $\alpha = n/m$, unde n este numărul de elemente din tabelă. Se demonstrează în [3] faptul că, în pentru funcții de dispersie bine alese, complexitatea de timp a operațiilor pe tabelele de dispersie, atât în cazul rezolvării coliziunilor cu liste înlănțuite, cât și în cazul adresării deschise, este determinată de α.

- În cazul tabelelor cu liste înlănțuite, α reprezintă numărul mediu de elemente per listă. Dacă m este proporțional cu n atunci se poate arăta că în cazul dispersiei uniforme și independente [3], complexitatea de timp a operațiilor este $O(1)$.
- În cazul tabelelor cu dublă repartizare, complexitatea de timp este constantă dacă factorul de încărcare se menține sub 0.6. Acest lucru înseamnă că, o astfel de tabelă trebuie să conțină un număr semnificativ de poziții neocupate. Dacă se umple prea tare, trebuie realocată memorie și repartizate din nou cheile în tabelă. Această nouă repartizare se numește *rehashing*.

Tabele cu liste versus adresare deschisă

1. **Tabele cu liste înlănțuite**:

   Avantaje:

   - simplu de implementat;
   - nu există pericolul umplerii;
   - nu trebuie testate mai multe poziții.

   Dezavantaje:

   - stocarea în zone diferite de memorie a elementelor unei liste constituie un dezavantaj în cazul memoriei ierarhizate a calculatoarelor moderne;
   - există situații în care este posibil ca multe chei să se repartizeze în aceeași listă, scăzând eficiența.

Exemplu de utilizare: pentru implementarea $unordered_set$, $unordered_map$ în STL

2. **Tabele cu adresare deschisă**:

   Avantaje:

   – păstrează elementele într-un bloc de memorie cu poziții adiacente (avantajul vectorilor);

   – nu necesită pointeri.

   Dezavantaje:

   – folosesc multă memorie care nu va fi ocupată.

   Exemplu de utilizare: pentru implementarea de dicționare în limbajul $Python$.

9.1.5 Tabele de dispersie în STL

În STL tabelele de dispersie sunt implementate prin două clase care presupun unicitatea cheilor și anume:

(1)
```
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_set;
```

(2)
```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<std::pair<const Key, T>>
> class unordered_map;
```

Observăm componentele din template:

Key - indică tipul de date al cheii stocate;

T - în cazul (2) indică tipul valorii asociate cheii;

$Hash$ - indică funcția de dispersie asociată, care este predefinită ca fiind funcția de $hash$ din STL. Aceasta este definită doar în cazul tipurilor primitive(`int`, `float`, `double`), precum și pentru `std::string`. Dacă se dorește stocare unor chei de alt tip este necesară definirea unei funcții de dispersie personalizate. Vom exemplifica acest lucru în secțiunea de probleme rezolvate.

$KeyEqual$ - operatorul care verifică egalitatea a două chei. Dacă tipul de date al cheilor nu dispune de operator de comparare, atunci trebuie definit.

$Allocator$ - se ocupă de gestionarea memoriei.

Diferența majoră între `unordered_set` și `unordered_map` este aceea că, `unordered_set` stochează doar chei, pe când `unordered_map` stochează elemente de tip `std::pair<Key, T>`, adică perechi de elemente în care prima componentă este cheia, iar a doua o valoare satelit. Inserția și căutarea se efectuează pe baza cheii.

În Visual Studio, în versiunea $C++17$, funcția de dispersie este FNV-1a [16], aplicabilă cheilor primitive sau $string$. Coliziunile se rezolvă printr-o listă înlănțuită (vezi [8]), care are inițial $nr_buckets = 8$. Când factorul de încărcare depășește 1, dimensiunea tabelei se dublează.

După calcularea valorii de $hash$ cu FNV-1a, plasarea cheii în tabelă se face astfel:

- se ia $num_buckets - 1$, valoare cu toți biții egali cu 1 (deoarece $num_buckets = 2^l$);
- indicele de inserție este $hash(k)\&(num_buckets - 1)$, unde $\&$ reprezintă operația AND pe biți. Practic, din valoarea $hash$ se păstrează cei mai puțin semnificativi l biți.

Detalii despre implementare pot fi găsite în codul sursă al Visual Studio.

Există de asemenea și două clase în care este permisă prezența cheilor multiple:

(1)
```
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_multiset;
```

(2)
```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
```

```
    class Allocator = std::allocator<std::pair<const Key, T>>
> class unordered_multimap;
```

În continuare vom da exemple de declarare și utilizare a `unordered_map`, clasa `unordered_set` fiind asemănătoare și rămânând să fie testată de către cititori.

Declarare și inițializare

Pentru utilizarea unui `unordered_map` este necesar să se includă header-ul corespunzător cu comanda:

```
#include<unordered_map>
```

Declararea unui obiect de tip `unordered_map` trebuie făcută specificând tipul cheii și tipul valorii, care vor fi stocate în perechile din `unordered_map`. La declarare obiectul poate fi inițializat folosind liste de inițializare sau pe baza altor structuri. Prezentăm câteva exemple în continuare.

Mapă vidă:

```
unordered_map<std::string, int> myMap;
```

Mapă inițializată pe bază de listă de inițializare:

```
unordered_map<std::string, int> myMap = { {"Mirela", 10},
   {"Adrian", 7}, {"Cornel", 8} };
```

Mapă inițializată pe baza unei alte mape (folosind `myMap` definit mai sus). Astfel, se realizează o copie pe baza constructorului de copiere:

```
unordered_map<std::string, int> myMap2(myMap);
```

Mapă inițializată pe baza unui vector de perechi:

```
std::vector<std::pair<std::string, int>> vec={{"Ana", 8},
   {"Mihai", 5}, {"Claudia", 10}};
unordered_map<std::string, int> myMap3(vec.begin(), vec.end());
```

Adăugarea de elemente, căutare și iterare

Pentru a adăuga elemente noi se poate proceda în două moduri:

(1) Folosind funcția `insert`

```
    unordered_map<std::string, int> myMap;
    myMap.insert({"Ana", 10});
    myMap.insert({"Diana", 6});
```

(2) Folosind operatorul de acces []

```
    myMap["Ana"] = 10;
    myMap["Diana"] = 6;
```

Observație: Dacă obiectul myMap conține cheia "Ana" și apelăm încă o dată:

```
myMap.insert({"Ana", 5});
```

instrucțiunea nu va avea niciun efect, deoarece nu este permisă existența de chei duplicat. În schimb, dacă există cheia "Ana", dar folosim instrucțiunea:

```
myMap.["Ana"] = 5;
```

se va accesa perechea cu cheia "Ana" și se va actualiza valoarea asociată, deci acum valoarea asociată cheii "Ana" va fi 5.

Căutarea unei chei se poate realiza folosind `find`. Dacă cheia există, `find` va returna un iterator către perechea cu cheia respectivă, iar în caz contrar va returna iteratorul `end()`.

```
#include <iostream>
#include<unordered_map>
#include<fstream>

int main() {

    std::unordered_map<std::string, int> elevi;
    std::ifstream fisier("Elevi2.txt");

    std::string nume;
    int nota;
    while (fisier >> nume >> nota)
    {
        elevi[nume] = nota;
    }

    std::cout << "Nume cautat: ";
    std::cin >> nume;
    auto it = elevi.find(nume);
    if (it != elevi.end())
        std::cout << "Elevul " << it->first << " are nota " <<
           it->second << "\n";
    else std::cout << "Nu exista acest elev in mapa\n";
    return 0;
}
```

Parcurgerea elementelor dintr-un `unordered_map` se poate realiza pe bază de iteratori sau folosind operatorul :. Exemple mai jos.

```
#include <iostream>
#include<unordered_map>
#include<fstream>

int main() {
    std::unordered_map<std::string, int> elevi;
    std::ifstream fisier("Elevi2.txt");
    std::string nume;
    int nota;

    while (fisier >> nume >> nota)
        elevi[nume] = nota;
    for (auto it = elevi.begin(); it != elevi.end(); it++){
        std::cout << it->first << " are nota " << it->second <<
           "\n";
    }
    for (auto el : elevi){
        std::cout << el.first << " are nota " << el.second<<"\n";
    }
    return 0;
}
```

Atenție: în al doilea caz, al parcurgerii cu operatorul :, variabila el va stoca la fiecare moment o copie a elementului curent din mapă, de tip pereche(cheie, valoare).

9.2 Probleme rezolvate

În această secțiune vom prezenta câteva probleme rezolvate, atât probleme teoretice, cât și probleme aplicative, care presupun utilizarea tabelelor de dispersie pentru o soluționare eficientă.

9.2.1 Probleme teoretice

1. Se consideră o tabelă de dispersie T cu dimensiunea $m = 11$, care rezolvă coliziunile cu liste înlănțuite și funcția de dispersie $h(k) = k\%m$. În această tabelă se consideră cheile 34, 56, 1, 22, 10, 23. Care este numărul de coliziuni între aceste chei, adică numărul de

perechi (k_1, k_2), $k_1 \neq k2$ pentru care $h(k_1) = h(k_2)$?

Rezolvare:

Determinăm pe rând pozițiile de inserție pentru cheile din enunț:

$h(34) = 34\%11 = 1$

$h(56) = 56\%11 = 1$ (coliziune cu 34)

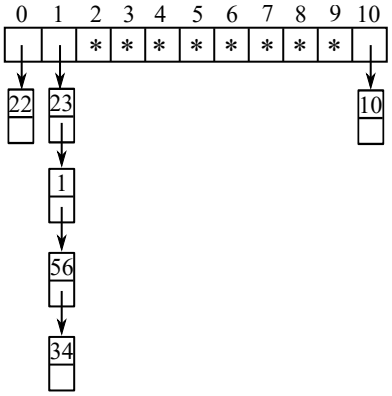
$h(1) = 1\%11 = 1$ (coliziune cu 34 și 1)

$h(22) = 22\%11 = 0$

$h(10) = 10\%11 = 10$

$h(23) = 23\%11 = 1$ (coliziune cu 34 și 1 și 56)

Tabela rezultată este reprezentată în figură:



Coliziunile sunt date de perechile: $(34, 56)$, $(34, 1)$, $(34, 23)$, $(1, 56)$, $(1, 23)$ și $(56, 25)$, deci 6 coliziuni.

2. Fie o tabelă de dispersie cu liste înlănțuite de dimensiune $m = 17$ și cu o funcție de dispersie $h(k) = k\%m$. Să se insereze în listă cheile 36, 19, 17, 85, 37, 87. Care este factorul de încărcare al tabelei?

Rezolvare:

Calculăm pozițiile de inserție pentru cheile: 36, 19, 17, 85, 37, 87.

$h(36) = 36\%17 = 2$

$h(19) = 19\%17 = 2$ (coliziune cu 36)

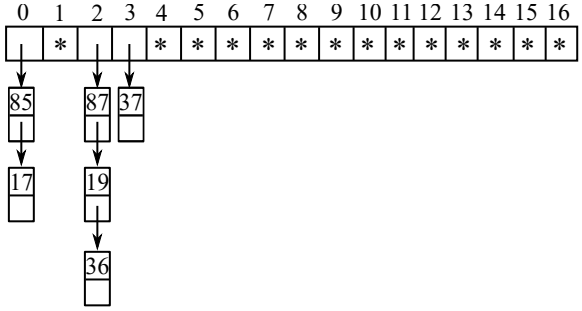
$h(17) = 17\%17 = 0$

$h(85) = 85\%17 = 0$ (coliziune, 85)

$h(37) = 37\%17 = 3$

$h(87) = 87\%17 = 2$ (coliziune cu 36 și 19)

Tabela rezultată este reprezentată în figură:



Factorul de încărcare este dat de raportul dintre numărul de elemente din tabelă și dimensiunea tabelei, în cazu nostru $\alpha = 6/17$.

3. Inserați cheile 25, 30, 36, 41, 52 într-o tabelă de dispersie de dimensiune $m = 11$ utilizând adresarea deschisă cu dublă repartizare, unde $h_1(k) = k$ și $h_2(k) = 1 + (k\%(m-1))$. Care este factorul de încărcare al tabelei.

Rezolvare:

Calculăm pozițiile de inserție pentru cheile din enunț. Formula de calcul pentru adresarea deschisă este dată de:

$$h(k,t) = (h_1(k) + t \cdot h_2(k))\%m$$

unde t reprezintă numărul de teste până la momentul curent. Pentru fiecare cheie k se începe cu $t = 0$. Dacă poziția calculată prin $h(k,0)$ a fost deja ocupată de altă cheie se testează poziția $h(k,1)$, dacă și aceasta este ocupată se continuă testarea, incrementând t, până când se găsește o poziție liberă sau nu mai sunt poziții se testat.

$h(25,0) = (h_1(25) + 0 \cdot h_2(25))\%11 = 25\%11 = 3 \Rightarrow T[3] = 25$

$h(30,0) = (h_1(30) + 0 \cdot h_2(30))\%11 = 30\%11 = 8 \Rightarrow T[8] = 30$

$h(36,0) = (h_1(36) + 0 \cdot h_2(36))\%11 = 36\%11 = 3$ - **ocupat**

$h(36,1) = (h_1(36) + 1 \cdot h_2(36))\%11 = (36 + (1 + 36\%10))\%11 = (36 + 7)\%11 = 10 \Rightarrow$

$T[10] = 36$

$h(41, 0) = (h_1(41) + 0 \cdot h_2(41))\%11 = 41\%11 = 8$ - **ocupat**

$h(41, 1) = (h_1(41) + 1 \cdot h_2(41)\%11 = (41 + (1 + 41\%10))\%11 = (41 + 2)\%11 = 10$ - **ocupat**

$h(41, 2) = (h_1(41) + 2 \cdot h_2(41)\%11 = (41 + 4)\%11 = 45\%11 = 1 \Rightarrow T[1] = 41$

$h(52, 0) = (h_1(52) + 0 \cdot h_2(52))\%11 = 52\%11 = 8$ - **ocupat**

$h(52, 1) = (h_1(52) + 1 \cdot h_2(52))\%11 = (52 + (1 + 52\%10))\%11 = (52 + 3)\%11 = 0 \Rightarrow T[0] = 52$

Tabela rezultată este ilustrată în figura următoare:

T	52	41	*	25	*	*	*	*	30	*	36
	0	1	2	3	4	5	6	7	8	9	10

4. Se consideră o tabelă de dispersie cu dimensiunea $m = 32$ în care coliziunile se rezolvă prin adresare deschisă cu dublă repartizare. Atunci funcțiile de dispersie auxiliare $h_1(k) = k\%m$ și $h_2(k) = 2k\%m$ sunt potrivite pentru o repartizare bună în tabelă. Este adevărată afirmația? Justificați.

   Rezolvare:

   Funcțiile de dispersie auxiliare $h_1(k) = k\%m$ și $h_2(k) = 2k\%m$ **NU sunt potrivite** pentru o repartizare bună în tabelă, deoarece m este un număr par, putere a lui 2, iar $h_2(k) = 2k\%m$ va fi tot un număr par, deci anumite poziții nu vor fi testate niciodată, pentru că funcția h_2 nu produce valori prime față de m.

9.2.2 Probleme aplicative

1. Implementați o structură de tip $hash_table$ în cel mai simplu mod posibil, folosind un vector de liste pentru bucket-uri. Fiecare element va fi te tip `std::pair<std::string, int>` (adică cheia e un `std::string`, valoarea e un `int`). Structura ar trebui să dispună de următoarele:

   - Container-ul în care se stochează bucket-urile și care va fi un vector de liste (folosiți STL). Se pornește cu un număr inițial de 8 liste.
   - Funcțiile corespunzătoare:

- **getBucket** - pentru o valoare naturală returnează indicele bucket-ului corespunzător.
- **size** - returnează numărul de elemente stocate în tabelă.
- **Insert** - inserează o pereche (cheie, valoare). Nu se permit chei duplicate.
- **Find** - găsește un element după cheie
- **Replace(key, value)** - înlocuiește valoarea unei anumite cheie key cu valoarea $value$, dacă aceasta există. Dacă nu există, inserează perechea.
- **Erase** - ștergerea unei valori cu o anumită cheie.
- **Rehashing** - se redimensionează vectorul de liste și se face rehashing.

Rezolvare:

Date de intrare: Nu este vorba de o problemă clasică, ci de implementarea unei structuri, deci nu avem date de intrare / ieșire în mod clasic.
Date de ieșire: -

Algoritm de rezolvare: Se definește structura cu membrii aferenți. Se folosește `std::hash<std::string>` pentru obținerea unei valori număr natural corespunzătoare cheii. Pentru implementarea funcției care alege bucket-ul de inserție, se folosesc algoritmii în pseudocod din curs [8] precum și elementele din biblioteca STL pentru implementarea funcțiilor atașate structurii.

Membrii structurii:

- Un vector de liste care va reprezenta tabela de dispersie, fiecare listă reprezentând un bucket. Dimensiunea tabelei m va fi un număr natural putere a lui 2 și se va începe cu $m = 8$. De fiecare dată când factorul de încărcare depășește valoarea max_load_factor, se va realiza *rehashing*, prin dublarea dimensiunii tabelei și repartizarea elementelor.
- Numărul de bucket-uri.

- Factorul de încărcare.
- Numărul de elemente stocate în tabelă.
- Câteva alte variabile / constante, precum numărul de biți folosit pentru numărul de bucket-uri, folosit pentru shiftarea pe biți în cadrul metodei multiply-shift, explicitate în cod.

Metodele structurii

- **Constructorul:** alocă memorie pentru 8 bucket-uri.
- **Funcția de** `rehashing`**:** dublează numărul de bucket-uri, alocă memorie pentru o nouă tabelă în care va repartiza elementele din vechea tabelă, inițializează vechea tabelă cu tabela nouă.
- **Funcția** `getBucket`**:** calculează valoarea de $hash$ pentru cheia dată și determină pe baza acesteia poziția în tabelă a bucket-ului în care se va insera noul element.
- **Funcția** `Insert`**:** are ca parametru un element de tip pereche (cheie, valoare). Pe baza cheii determină poziția k de inserție apelând funcția `getBucket`. Apoi se caută în lista respectivă cheia. Dacă nu există, se adaugă noul element. Apoi se recalculează factorul de încărcare și, la nevoie, se apelează funcția de `rehashing`.
- **Funcția** `Find`**:** primește ca parametru o cheie, determină bucket-ul în care ar trebui să se afle această cheie, iar apoi caută în lista corespunzătoare elementul cu cheia dată. Funcția returnează un pointer către elementul găsit sau $nullptr$. Modul de implementare cu mai multe liste, nu cu una singură, nu permite returnarea de iteratori, deoarece pentru fiecare listă în parte iteratorul `end()` este diferit. Din acest motiv, o abordare mai eficientă, folosită în STL, este aceea cu o singură listă [8], dar depășește cadrul cerinței.
- **Funcția** `Replace`**:** are ca parametru o pereche de tip (cheie, valoare). Se caută în tabela de dispersie elementul cu cheia respectivă și se înlocuiește valoarea cu valoarea nouă transmisă ca parametru.
- **Funcția** `Erase`**:** primește ca parametru o cheie și elimină din tabelă elementul cu cheia respectivă, dacă există.

Determinarea bucket-ului de inserție: considerând o cheie k, pentru a determina indicele bucket-ului de inserție se folosește metoda *multiply-shift*, dată de formula:

$$h_a(k) = (Ak) >> (w - p)$$

în care A este o constantă care depinde dacă se consideră un sistem pe 32 biți sau unul pe 64 biți. Anume [10]:

- pentru 32 biți $A = 2654435769$
- pentru 64 biți $A = 11400714819323198485$

În plus w este numărul de biți al sistemului considerat, respectiv 32 sau 64, iar p este numărul de biți corespunzător dimensiunii tabelei, care este o putere a lui 2.

Algoritm scris în C++

Definirea structurii cu membrii corespunzători

```
#include <iostream>
#include<vector>
#include<list>
#include<string>
#include<fstream>

struct hash_table {
    static const unsigned int w = 64; // Pentru numere
       intregi pe 64-bits
    static const unsigned int A = 11400714819323198485;
    unsigned int nr_buckets = 8;
    unsigned int bucket_bits = 3; //numar de biti folosit
       pentru numarul //de buckets, care este o putere a lui 2
    unsigned int shift_bits = w - bucket_bits; // Shiftare cu
       (64 - 3 = 61)

    double load_factor = 0;
    double max_load_factor = 1.0;
    unsigned int nr_of_elements = 0;
    //container
    std::vector<std::list<std::pair<std::string, int>>> table;
    //constructor
    hash_table(){
       table.resize(nr_buckets);
    }
```

```
    void Rehashing();
    unsigned int size();
    unsigned int getBucket(std::string key);

    void Insert(std::pair<std::string, int > element);
    std::pair<std::string, int>* Find(std::string key);
    void Replace(std::string key, int value);
    void Erase(std::string key);
};
```

Definim funcția `getBucket`. Această funcție determină poziția de inserție pe baza valorii de hash a cheii, precum și pe baza funcției *multiply-shift*.

```
unsigned int::hash_table getBucket(std::string key){
    auto Hash_value = std::hash<std::string>()(key);
    unsigned int position = (A * Hash_value) >> shift_bits;
    return position;
}
```

Definim funcția de `Rehashing`

```
void::hash_table Rehashing(){
    //dublam numarul de bucket-uri
    nr_buckets *= 2;
    //crestem numarul de biti cu 1
    bucket_bits++;
    //recalculam factorul de shifare
    shift_bits = w - bucket_bits;

    //se face o tabela temporara, in care voi repartiza din
       nou elementele
    std::vector<std::list<std::pair<std::string, int>>>
       temp_table;
    temp_table.resize(nr_buckets);

    for (auto lista : table){
        for (auto el : lista){
            unsigned int position = getBucket(el.first);
            temp_table[position].push_back(el);
        }
    }

    //initializam tabela veche cu cea noua
    table = temp_table;
    load_factor = (double)nr_of_elements / nr_buckets;
}
```

Funcția pentru returnarea dimensiunii tabelei:

```
unsigned int hash_table:: size()
{
    return nr_of_elements;
}
```

Urmează funcția de inserare a unei perechi de tip (cheie, valoare). În cazul nostru cheile sunt de tip `string`, iar valorile de tip `int`. Desigur, dacă definim structura de tip `template`, atunci pot fi alese de către utilizator tipurile de date dorite.

```
void hash_table::Insert(std::pair<std::string, int > element)
{
    unsigned int position = getBucket(element.first);

    //daca se doreste verificarea existentei cheii in tabela
       cu lambda-expresii
    //if (std::find_if(table[position].begin(),
       table[position].end(), [&element](const
       std::pair<std::string, int>& el) {el.first ==
       element.first;})!=table.end())
    //      table[position].push_back(element);

    //cautarea clasica a cheii cu iteratori
    auto it = table[position].begin();
    for (; it != table[position].end() && it->first !=
       element.first; ++it);
    //se adauga elementul in tabela
    if (it == table[position].end())
    {
        table[position].push_back(element);
        ++nr_of_elements;
    }
    //daca factorul de incarcare > factorul maxim facem
       rehashing
    load_factor = (float)nr_of_elements / nr_buckets;
    if (load_factor > max_load_factor)
        Rehashing();
}
```

Funcția `Find` caută un element pe baza cheii sale. Datorită faptului că, implementarea nu este făcută precum cea din STL, folosind o singură listă înlănțuită partajată în bucket-uri [8], ci folosind pentru fiecare bucket o listă separată, nu putem returna un iterator, decât

dacă am defini noi o clasă / structură de tip iterator peste structura noastră proprie. Astfel, funcția `Find` va returna un pointer către elementul căutat, dacă există, sau `nullptr`.

```
//cautarea returneaza un un pointer catre elementul gasit
//altfel returneaza nullptr

std::pair<std::string, int>* hash_table::Find(std::string
   key) {
    unsigned int position = getBucket(key);

    auto it = table[position].begin();
    for (; it != table[position].end() && it->first != key;
       ++it);
    if (it != table[position].end()) {
        return &(*it);  // Return pointer to found element
    }
    else {
        return nullptr;
    }
}
```

În final, prezentăm funcția de înlocuire a valorii unui element cu o cheie dată, precum și funcția de ștergere unui element cu o cheie dată (dacă există).

```
void hash_table:: Replace(std::string key, int value)
    auto it = Find(key);
    if (it)
        it->second = value;
    else Insert({ key, value });
}

void hash_table:: Erase(std::string key){
    unsigned int position = getBucket(key);

    auto it = table[position].begin();
    for (; it != table[position].end() && it->first != key;
       ++it);
    if (it != table[position].end())
        table[position].erase(it);
}
```

Justificarea soluției problemei și analiza complexității

Justificare alegerii constantelor A și w este discutată în bibliografia recomandată [3], [10].

În continuare vom analiza pe rând funcțiile implementate anterior.

Funcția `getBucket`

- *Discutarea implementării.* Determinarea valorii de $hash$ se face folosind STL, deci este sigur corectă.
  Determinarea bucket-ului corespunzător se face aplicând formula *multiply-shift* și corectitudinea este discutată în partea teorie [8].
- *Complexitate temporală* $O(1)$. Toate operațiile sunt în complexitate constantă.
- *Complexitate spațială* $O(1)$

Funcția `Rehashing`

- *Discutarea implementării.* Dat fiind ca numărul de bucket-uri se dublează, este evident faptul ca numărul de biți corespunzători dimensiunii (egală cu numărul de bucket-uri) se incrementează cu 1. Evident, din formula multiply-shift, trebuie actualizat numărul de biți cu care se shiftează.
  Restul funcției reprezintă efectiv alocarea unei tabele noi, în care se copiază pe rând elementele din tabela veche, folosind funcția `getBucket` pentru alegerea bucket-ului potrivit pentru un anumit element. Deoarece folosim `vector` din STL, dealocarea memoriei este gestionată de către alocator și nu trebuie să ne preocupe.
- *Complexitate de temporală.* Parcurgerea fiecărui element din tabela veche este liniară în numărul de elemente ale tabelei (parcurgerea fiecărei liste).
  Identificarea bucket-ului de inserție de realizează în complexitate constantă (hashing, multiply-shift).
  În concluzie complexitatea este $O(n)$, n este numărul de elemente din tabelă.
- *Complexitate spațială.* Pentru stocarea elementelor în tabela inițială sunt folosite m zone de memorie, corespunzătoare vectorului de bucket-uri, precum și n zone de memorie în care sunt stocate elementele propriu-zise.
  Pentru tabela nouă, care se alocă, vor fi $2 * m$ poziții pentru buckert-uri, dar tot n poziții pentru elemente. Datorită faptului că, atunci când factorul de încărcare devine egal cu factorul maxim (setat la 1), se face realocare, avem $n < 2 * m$, deci complexitatea de spațiu este $O(m)$.

Funcția `size`

- *Complexitate temporală* $O(1)$
- *Complexitate spațială* $O(1)$

Funcția `Insert`.

- *Discutarea implementării.* Determinarea poziției bucket-ului în care trebuie inserată perechea (cheie, valoare) se realizează folosind funcția `getBucket` discutată mai sus. Apoi se verifică existența cheii în bucket-ul corespunzător, prin parcurgerea cu iteratori a listei respective. Corectitudinea este evident.

  În cazul, în care nu a fost găsită cheia, se adaugă perechea în listă, folosind $push_back$, funcție proprie $list$ din STL.

  Funcția de `Rehashing` a fost deja discutată.

  Observație. Această funcție ar putea fi implementată și apelând funcția `Find`.
- *Complexitate temporală.* Determinarea indicelui bucket-ului de inserție este $O(1)$. Verificarea existenței cheii în lista corespunzătoare acestui bucket depinde de lungimea listei. Din considerente teoretice, se presupune că în majoritatea cazurilor, aceste liste au foarte puține elemente, deci complexitatea este $O(1)$. Funcția `Rehashing` se apelează doar atunci când factorul devine 1, deci relativ rar, deci în complexitate amortizată complexitatea rămâne constantă.
- *Complexitate spațială.* Considerând strict inserția noului element (nu toată tabela), complexitatea este $O(1)$. Aceleași observații referitor la `Rehashing`,

Funcția `Find`.

- *Discutarea implementării.* Determinarea poziției bucket-ului în care trebuie inserată perechea (cheie, valoare) se realizează folosind funcția `getBucket` discutată mai sus. Apoi se verifică existența cheii în bucket-ul corespunzător, prin parcurgerea cu iteratori a listei respective.
- *Complexitate temporală.* Determinarea indicelui bucket-ului de inserție este $O(1)$. Verificarea existenței cheii în lista corespunzătoare acestui bucket depinde de lungimea listei. Din considerente teoretice, se presupune că în majoritatea cazurilor, aceste liste au foarte puține elemente, deci complexitatea este $O(1)$.
- *Complexitate spațială* $O(1)$

Funcția `Replace` are la bază funcțiile `Find` și `Insert` discutate mai sus.

Funcția `Erase`.

- *Discutarea implementării.* Determinarea poziției bucket-ului în care trebuie inserată perechea (cheie, valoare) se realizează folosind funcția `getBucket` discutată mai sus. Apoi se verifică existența cheii în bucket-ul corespunzător, prin parcurgerea cu iteratori a listei respective. Corectitudinea este evident.
- *Complexitate temporală.* Determinarea indicelui bucket-ului de inserție este $O(1)$. Verificarea existenței cheii în lista corespunzătoare acestui bucket depinde de lungimea listei. Din considerente teoretice, se presupune că în majoritatea cazurilor, aceste liste au foarte puține elemente, deci complexitatea este $O(1)$. Ștergerea perechii (cheie, valoare) se realizează folosind funcția `erase` specifică listelor din STL în complexitate $O(1)$.
- *Complexitate spațială* $O(1)$

2. **Intersecție mulțimi.** Se consideră două mulțimi de nume citite din două fișiere în doi vectori. Determinați intersecția celor două mulțimi folosind $unordered_set$ din STL.

Rezolvare:

Date de intrare: Două fișiere conținând nume de tipul $string$.

Date de ieșire: Un vector de $string$-uri conținând numele comune din cele două fișiere.

Algoritm de rezolvare:

- Se declară trei variabile de tip

```
std::vector<std::string>
```

- Se citesc din fișierul 1 numele și se plasează în primul vector. Se citesc apoi numele din fișierul 2 și se plasează în al doilea vector.
- Se declară o variabilă de tip

```
std::unordered_set<std::string>
```

- Se parcurge primul vector și se introduc elementele sale în `unordered_set`.

- Se parcurge al doilea vector și dacă elementul curent se află în `unordered_set` atunci face parte din intersecție și se adaugă la al treilea vector.

Algoritm scris în C++

```
#include<iostream>
#include<tuple>
#include<unordered_set>
#include<fstream>

int main(){
    std::unordered_set<std::string>nume;
    std::vector<std::string> pers1;
    std::vector<std::string> pers2;
    std::vector<std::string> intersectie;
    std::ifstream fis("pers1.txt");
    std::string persoana;

    while (fis >> persoana){
        pers1.push_back(persoana);
    }
    fis.close();
    fis.open("pers2.txt");

    while (fis >> persoana){
        pers2.push_back(persoana);
    }
    fis.close();

    for (auto& persoana : pers1){
        nume.insert(persoana);
    }
    for (auto& persoana : pers2){
        if (nume.find(persoana) != nume.end()){
            intersectie.push_back(persoana);
        }
    }
    std::cout << "Intersectia este:\n";
    for (auto persoana : intersectie)
        std::cout << persoana << ", ";
    std::cout << "\n";
    return 0;
}
```

Justificarea soluției propuse Se consideră că niciun fișier nu conține dubluri (sunt mulțimi de nume). După ce se inserează în `unordered_set` elementele din prima mulțime, este

evident că, parcurgând a doua mulțime, pentru ca un element din aceasta să se regăsească și în prima mulțime, trebuie să se regăsească de asemenea în set. Astfel în mod evident, în al treilea vector vom avea elementele comune.

Analiza complexității.

- *Complexitatea de timp*. Parcurgerea vectorului $pers1$ și adăugarea tuturor elementelor sale în set se realizează în complexitate $O(m)$, unde m este numărul de elemente din vectorul $pers1$ (adăugarea în set este $O(1)$).
  Parcurgerea vectorului $pers2$ și verificarea existenței elementului curent în set se realizează în complexitate $O(n)$, unde n este numărul de elemente din vectorul $pers2$ (căutarea în set este $O(1)$).
  Rezultă o complexitate de $O(m + n)$.
- *Complexitatea de spațiu*. Vectorii $pers1$ și $pers2$ ocupă $O(m+n)$ spațiu de memorie. Structura $nume$ de tip $unordered_set$ ocupă $O(m+nr_buckets)$ zone de memorie, unde $nr_buckets$ este numărul de bucket-uri din $unordered_set$.

3. **Competiții.** Se consideră un număr de competiții sportive la care s-au înscris concurenți. Pentru fiecare competiție există o listă cu numele, prenumele și vârsta concurenților (un concurent este de tip `std::tuple<std::string, std::string, int>`). Datele se citesc dintr-un fișier, iar concurenții fiecărei competiții pot fi stocați într-un vector. Scrieți o funcție care returnează un vector cu acei concurenți care participă la mai mult de o competiție. Doi concurenți sunt considerați aceeași persoană dacă au același nume, același prenume și aceeași vârstă. Un concurent apare o singură dată în vectorul rezultat, iar fiecare competiție se numără o singură dată pentru acesta. În `main` se afișează, pentru fiecare persoană din acest vector, competițiile la care participă. Folosiți `std::unordered_map`.
   Variantă: rezolvați problema folosind `std::unordered_multimap`.

   Rezolvare:
   Date de intrare: Fișierul de competiții. O listă cu numele competițiilor.
   Date de ieșire: Vectorul de concurenți care participă la mai multe competiții.

Algoritm de rezolvare:

Vom considera fișierul de intrare organizat în modul următor:

- pe prima linie este un număr natural, care indică numărul de competiții
- pentru fiecare competiție sunt indicate întâi pe o linie numele competiției ($string$) și numărul de concurenți, apoi pe liniile următoare este plasat câte un concurent cu: nume ($string$), prenume ($string$) și vârsta (int).

Vom defini în plus tipul $concurent$ ca fiind un tuplu cu trei câmpuri, care vor stoca numele, prenumele și vârsta concurentului, folosind `typedef` sau:

```
using concurent = std::tuple<std::string, std::string, int>;
```

precum și tipul $competiție$, ca o pereche formată dintr-un $string$, ce va stoca numele competiției, și un vector de $concurent$:

```
using competitie = std::pair<std::string,
   std::vector<concurent>>;
```

Etape:

- Se citesc din fișier competițiile într-un vector de competiții.
- Se declară o variabilă care va stoca perechi de tip (concurent, lista de competiții)

  ```
  std::unordered_map<concurent, std::vector<std::string>,
     custom_hash> tabela_concurentilor;
  ```

- Se parcurge vectorul de competiții și, pentru fiecare competiție, se parcurge vectorul său de concurenți, iar pentru fiecare concurent se adaugă în `tabela_concurentior`, în vectorul de $string$-uri corespunzător numelui concurentului, competiția respectivă.
- În final, în `tabela_concurentior` vom avea pentru fiecare concurent lista cu competițiile la care participă și este ușor să identificăm acei concurenți, care participă la mai mult de o competiție.

Observație: În cadrul declarației de mai sus a variabilei `tabela_concurentior` observăm că în afară de tipul cheii și al valorii a mai fost adăugat și numele unei **clase de hashing** $custom_hash$. Trebuie avut în vedere faptul, ca nu există $hash$ predefinit pentru

tuple în STL. Din acest motiv este necesar să declarăm noi un *hash* customizat. Acest lucru trebuie făcut prin definire unei clase sau structuri în care se definește operatorul

```
std::size_t operator()()
```

Varianta 1

Cel mai simplu mod de a defini un astfel de operator ar fi, să se considere fiecare câmp al tuplului și să se combine *hash*-ul calculat de STL pentru aceste câmpuri de exemplu prin operația *XOR* pe biți.

```
struct custom_hash1 {
    std::size_t operator()(const concurent& el) const
    {
        // Combine hashes of x and y using the bitwise XOR
        return std::hash<std::string>()(std::get<0>(el)) ^
            std::hash<std::string>()(std::get<1>(el)) ^
            std::hash<int>()(std::get<2>(el));
    }
};
```

Aceasta nu este o varianta ideală, deoarece poate duce la numeroase coliziuni.

Varianta 2

Putem construi un operator de *hash*-ing folosind un algoritm asemănător cu Bernstein hash (vezi [8]).

```
struct custom_hash2 {
    std::size_t operator()(const concurent& el) const
    {
            //use an ideea similar to Bernstein hash
        std::size_t res = 17;
        res = res * 33 +
           std::hash<std::string>()(std::get<0>(el));
        res = res * 33 +
           std::hash<std::string>()(std::get<1>(el));
        res = res * 33 + std::hash<int>()(std::get<2>(el));
        return res;
   }
};
```

Evident că putem considera și alte funcții de *hash* eficiente, de exemplu FNV-1a, pe care o vom și utiliza în rezolvarea următoare.

Algoritm scris în C++

```
#include<iostream>
#include<tuple>
#include<unordered_map>
#include<fstream>
#include<vector>

using concurent = std::tuple<std::string, std::string, int>;
using competitie = std::pair<std::string,
   std::vector<concurent>>;

struct FNVHasher {
    std::size_t operator()(const concurent& tup) const {
        // FNV-1a constants
        constexpr std::size_t FNV_OFFSET_BASIS =
           14695981039346656037ull; //pentru reprezentarea pe
           64 bits
        constexpr std::size_t FNV_PRIME = 1099511628211ull;
           //pentru reprezentarea pe 64 bits
        std::size_t hash = FNV_OFFSET_BASIS; // pornim cu un
           offset
        // Se calculeaza hash pentru fiecare camp si se
           combina cu FNV-1a
        hash ^= std::hash<std::string>{}(std::get<0>(tup));
        hash *= FNV_PRIME;
        hash ^= std::hash<std::string>{}(std::get<1>(tup));
        hash *= FNV_PRIME;
        hash ^= std::hash<int>{}(std::get<2>(tup));
        hash *= FNV_PRIME;
        return hash;
    }
};

//Functie de afisare a unui concurent
void print(const concurent& pers){
    std::cout << std::get<0>(pers) << " " <<
       std::get<1>(pers) << " cu varsta: " <<
       std::get<2>(pers) << " ani\n";
}

//Functie de afisare a concurentilor care participa la o
   competitie
void print_competitie(const competitie& concurs){
    std::cout << "La " << concurs.first << " s-au inscris:\n";
    for (auto &pers : concurs.second)
        print(pers);
```

```
    std::cout << std::endl;
}

//Functia care determina concurentii care participa la mai
    multe competitii
std::vector<concurent>
    concurenti_cu_mai_multe_competitii(const
    std::unordered_map<concurent, std::vector<std::string>,
    FNVHasher> &tabela_concurentilor){
     std::vector<concurent> rezultat;
     for (auto& pers : tabela_concurentilor){
         if (pers.second.size() > 1){
             rezultat.push_back(pers.first);
         }
     }
     return rezultat;
}
int main(){
     std::ifstream fis("Competitii.txt");

     std::vector<competitie> vector_concursuri;
     std::unordered_map<concurent, std::vector<std::string>,
        custom_hash2> tabela_concurentilor;

     //Citirea din fisier a competitiilor
     int nr_competitii;
     fis >> nr_competitii;

     for (int index = 0; index < nr_competitii; ++index) {
         competitie concurs;
         int nr_concurenti;
         fis >> concurs.first;
         fis >> nr_concurenti;

         //citim fiecare participant
         for (int index2 = 0; index2 < nr_concurenti;
            ++index2){
             std::string nume, prenume;
             int varsta;
             fis >> nume >> prenume >> varsta;
             //adaug participantul la vectorul competitiei
             concurs.second.emplace_back( nume, prenume,
                varsta );
         }
       vector_concursuri.push_back(concurs);
     }
     fis.close();
```

```
    //pentru fiecare concurs, adaug numele comptitiei in
       tabla pentru fiecare concurent care participa
    for (auto& concurs : vector_concursuri)
        for (auto& concurent : concurs.second)
           tabela_concurentilor[concurent].
                push_back(concurs.first);

   std::vector<concurent>cautat =
      concurenti_cu_mai_multe_competitii(tabela_concurentilor);
    std::cout << "Concurentii care participa la mai multe
       competitii:\n";

    for (auto& pers : cautat){
        print(pers);
        std::cout << "participa la competitiile: ";
        for(auto& concurs: tabela_concurentilor[pers])
           std::cout << concurs << ", ";
        std::cout << "\n\n";
    }
}
```

Justificarea soluției propuse. Variabila `vector_concursuri` conține perechi de tip (nume_concurs, vector de concurenți). Practic avem pentru fiecare concurs toate tuplurile care reprezintă concurenții de la acel concurs.

Se parcurge acest vector prin instrucțiunea

```
for (auto& concurs : vector_concursuri)
        for (auto& concurent : concurs.second)
           tabela_concurentilor[concurent].
                push_back(concurs.first);
```

în care pentru fiecare *concurent* se adaugă în `unordered_map`-ul `tabela_concurentilor` pentru fiecare concurent numele competițiilor la care participă.

Apoi este evident, că este suficient să se itereze prin containerul `tabela_concurentilor` și să se determine acei concurenți pentru care vectorul atașat conține mai mult de un element. Acest lucru este făcut prin funcția `concurenti_cu_mai_multe_competitii`.

Analiza complexității

- *Complexitatea de timp.* Parcurgerea vectorului de concursuri pentru a completa tabela concurenților are complexitatea $O(\sum n_k)$, unde n_k este numărul de concurenți înscriși la competiția k.

Funcția `concurenti_cu_mai_multe_competitii` presupune iterarea prin tabela concurenților și are complexitatea $O(n)$, unde n este numărul efectiv de concurenți. Avem $n \leq \sum n_k$, deoarece este posibil ca un concurent să fie înscris la mai multe competiții.

Afișarea concurenților care participă la mai multe competiții presupune parcurgerea vectorului returnat de funcția `concurenti_cu_mai_multe_competitii` și pentru fiecare concurent, afișarea vectorului de competiții asociat din tabela de competiții. Acest lucru este $O(\sum m_j)$, unde m_j este numărul de competiții la care participă concurentul j, dintre acei concurenți aflați în vectorul returnat.

9.3 Probleme propuse

În această secțiune sunt propuse câteva probleme teoretice și o serie de probleme aplicative, multe dintre ele presupunând utilizarea containerelor STL corespunzătoare tabelelor de dispersie.

9.3.1 Probleme teoretice

1. Inserați următoarele chei: 44, 18, 28, 54, 57, 78, 41, 26 într-o tabelă de dispersie de dimensiune $m = 13$ utilizând liste înlănțuite pentru rezolvarea coliziunilor și funcția de dispersie $h(k) = k\%m$.

2. Inserați cheile 25, 30, 36, 41, 52, 84, 55, 11 într-o tabelă de dispersie de dimensiune $m = 11$ cu adresare, deschisă cu dublă repartizare, unde $h_1(k) = k$ și $h_2(k) = 1+(k\%(m-1))$. Cât este factorul de încărcare al tabelei.

3. Se consideră o tabelă de dispersie T cu adresare deschisă ce folosește dubla repartizare și care are dimensiunea m. Care dintre următoarele afirmații sunt adevărate?

   a) Dacă dimensiunea tabelei este $m = 16$, atunci funcțiile auxiliare $h_1(k) = k\%m$ și $h_2(k) = 2k\%m$ sunt foarte potrivite pentru construirea funcției de dispersie.

   b) Factorul de încărcare α nu poate depăși valoarea 1.

   c) Poziția de inserție a unei chei în tabelă depinde doar de valoarea cheii.

d) Cea mai bună eficiență de timp a căutării unui element în tabelă se obține atunci când tabela este plină.

e) Pentru factorul de încărcare $\alpha = 1$, căutarea nereușită are complexitate $O(m)$.

9.3.2 Probleme aplicative

*

1. **Palindrom**. Se citește din fișier un vector de caractere separate prin virgulă. Scrieți o funcție în C++ folosind containerul `unordered_map` din STL care primește ca parametru un astfel de vector și care are ca rezultat un palindrom alcătuit din caracterele vectorului, în cazul în care este posibil.

2. **Permutări.** Se citesc dintr-un fișier două șiruri de caractere s_1 și s_2. Scrieți o funcție C++ care verifică, folosind `std::unordered_map`, dacă s_2 conține exact aceleași caractere ca s_1, în aceleași cantități, indiferent de ordinea lor. Funcția returnează *true* în caz afirmativ și *false* altfel.
   Exemple: `"abca"` și `"caab"` sunt permutări unul altuia, iar `"abc"` și `"abb"` nu sunt.

3. Să se afișeze toate cvadruplurile de numere naturale (a, b, c, d) cu fiecare număr mai mic decât 1000, pentru care $a^2 + b^2 = c^2 + d^2$. Rezolvați în complexitate cât mai bună folosind `unordered_map`. Explicați complexitatea algoritmului folosit.

4. Se consideră 4 vectori de numere A, B, C și D de lungime cel mult 500 fiecare. Să se determine toate cvadruplurile de indici (i, j, k, l) pentru care $A[i]$ + $B[j]$ + $C[k]$ + $D[l]$ = 0. Ce complexitate are algoritmul dvs? Care este cea mai bună complexitate pe care o puteți obține?

**

1. **Duplicate apropiate**. Se citește dintr-un fișier un număr de valori reale și se stochează într-un vector. Să se determine în mod eficient, dacă există două numere egale în vector, aflate la o distanță mai mică sau egală cu o valoare dată $dist$. Folosiți `unordered_set` din STL.

2. **Magazine.** Se consideră nr_mag magazine. Fiecare are un număr de produse. Să se verifice care magazin are cele mai multe produse exclusive (nu apar decât în magazinul respectiv). Citiți dintr-un fișier în câte un vector de `std::string` produsele pentru fiecare magazin. Afișați în final magazinul cu cele mai multe produse exclusive și care sunt aceste produse. Folosiți `unordered_map` din STL.

3. **Fluctuație de prețuri.** Se consideră o listă de produse alimentare împreună cu prețul lor per unitate de măsură. Pentru fiecare produs se cunosc denumirea, prețul și magazinul la care s-a înregistrat prețul respectiv. Pentru fiecare magazin un anumit produs apare o singură dată în listă. Același produs poate apărea în magazine diferite, iar prețul poate varia în funcție magazinul din care provine.
   Scrieți o funcție care preia lista de produse (citite dintr-un fișier într-un vector de produse) și identifică produsul cu cea mai mare fluctuație de preț, unde prin fluctuație de preț se înțelege diferența dintre prețul maxim și cel minim identificate în listă. Folosiți `unordered_map`.

4. **Aruncarea zarurilor** Se consideră 3 zaruri care sunt aruncate de un număr N de ori. Practic pentru fiecare aruncare se generează 3 numere aleatoare în mulțimea $\{1, \ldots, 6\}$. Pentru fiecare triplet de numere (n_1, n_2, n_3), $n_i \in \overline{1,6}$ citit de la tastatură să se indice, de câte ori a fost aruncat tripletul. **Atenție** permutări ale aceluiași triplet nu trebuie considerate separat.

***

1. **Anagrame**. Se consideră un șir de cuvinte citite dintr-un fișier. Scrieți o funcție C++, care să grupeze anagramele. Se consideră anagramă un cuvânt obținut prin rearanjarea literelor altui cuvânt. Folosiți containere STL pentru rezolvare.
   Exemplu:
   Se consideră cuvintele $\{car, rac, cos, amin, arc, soc, polca, lac, cal, pocal, mina, copil, anim\}$. Atunci se vor grupa: $\{car, rac, arc\}$, $\{cos, soc\}$, $\{amin, mina, anim\}$, $\{lac, cal\}$, $\{pocal, polca\}$, $\{copil\}$.

2. **Partidele de șah**. Într-un grup de N prieteni aceștia joaca partide de șah în perechi. Fiecare dintre prieteni este caracterizat prin nume și prenume (nu există doi prieteni cu

aceleași nume și prenume). După doi ani de jocuri, vor să afle care pereche de jucători a jucat cele mai multe partide. Se citesc dintr-un fișier perechile de nume corespunzătoare unei partide. **Indicație**: Folosiți un `unordered_map` care are drept cheie o pereche de jucători și o funcție de hash-ing adecvată.

****

1. **Implementare tabelă de dispersie - cu o singură listă înlănțuită**. Construiți o clasă HashTable (sau HashMap) de tip $template$ potrivită, care să includă operațiile de inserție, ștergere și căutare. Elementele stocate vor fi de tip perechi de (cheie, valoare). Rezolvarea coliziunilor se va realiza folosind bucket-uri stocate într-o singură listă înlănțuită, similar cu implementarea STL, (folosiți `std::list` și indicațiile din curs [8]). Se pornește inițial cu 8 bucket-uri, se folosește pentru identificarea bucket-ului de inserție metoda $multiply - shift$, iar dacă factorul de încărcare depășește maximul permis (setat la 1 inițial, dar poate fi configurat de utilizator) se realizează dublarea dimensiunii tabelei (a numărului de bucket-uri) și repartizarea elementelor în noua tabelă. Implementați de asemenea operatorul [ ] cu aceleași funcționalități ca în STL. În funcția `main` citiți dintr-un fișier n elemente de tip pereche (cheie-valoare) ($n > 20$), repartizați elementele în tabelă, apoi permiteți căutarea, adăugarea, ștergerea de elemente sau parcurgerea structurii folosind un meniu (de tip switch - case). Puteți porni de la codul din problema rezolvată a acestui capitol.

2. **Implementare tabelă de dispersie - adresare deschisă**. Construiți o clasă HashTable (sau HashMap) potrivită folosind dubla repartizare, care să includă operațiile de inserție, căutare și ștergere. Elementele stocate vor fi de tip (cheie, valoare). Folosiți `std::pair` din STL. În cazul dublei repartizări, dacă factorul de încărcare al tabelei depășește 0.7, se cere redimensionarea tabelei (aproximativ dublul dimensiunii inițiale) și redistribuirea elementelor în noua tabelă (*rehashing*). Dacă contaminarea (marcarea de elemente ca șterse) devine prea mare, se face redistribuirea elementelor în tabelă. Implementați de asemenea operatorul [ ] cu aceleași funcționalități ca în STL. În funcția `main` citiți dintr-un fișier n elemente de tip pereche (cheie-valoare) ($n > 20$), repartizați elementele în tabelă, apoi permiteți căutarea, adăugarea sau ștergerea de elemente (meniu). De asemenea permiteți parcurgerea și afișarea perechilor $< cheie, valoare >$ pentru toate elementele din tabelă.

Capitolul 10

Divide et Impera

10.1 Aspecte teoretice

Metoda *Divide et Impera* reprezintă una dintre cele mai elegante și eficiente strategii de proiectare a algoritmilor, având la bază ideea de a aborda problemele complexe printr-o serie de pași simpli și structurali. Principiul fundamental constă în descompunerea recursivă a unei probleme mari într-un set de subprobleme mai mici și similare ca structură, care sunt apoi rezolvate individual, iar soluțiile obținute sunt combinate pentru a forma răspunsul final.

Această abordare constituie temelia multor algoritmi performanți din domenii diverse, precum sortarea, căutarea, procesarea datelor sau analiza numerică. Ea oferă o metodă naturală de structurare logică a problemelor și o modalitate eficientă de optimizare a performanței prin reducerea volumului de calcul.

Pentru a putea aplica metoda *Divide et Impera*, o problemă trebuie să îndeplinească următoarele condiții esențiale:

1. Problema poate fi divizată în două sau mai multe subprobleme independente, a căror rezolvare nu depinde una de cealaltă.

2. Subproblemele sunt similare problemei inițiale, astfel încât același algoritm se poate aplica recursiv.

3. Subproblemele cele mai mici pot fi rezolvate direct, fără alte descompuneri, printr-o metodă simplă și eficientă.

10.1.1 Etapele metodei

Procesul de rezolvare prin *Divide et Impera* se desfășoară în trei etape fundamentale, care se succed logic și se completează reciproc:

1. Descompunerea - În această primă etapă, problema inițială este împărțită în subprobleme mai mici, de același tip. Această divizare continuă recursiv până când se ajunge la o formă elementară a problemei, suficient de simplă pentru a fi rezolvată direct.

2. Rezolvarea subproblemelor - Odată obținute subproblemele simple, fiecare dintre acestea este rezolvată individual. De obicei, această etapă implică apeluri recursive până la atingerea unei condiții de oprire (cazul de bază).

3. Combinarea rezultatelor - În ultima etapă, soluțiile obținute pentru subproblemele simple sunt reunite pentru a forma soluția completă a problemei inițiale. Modul de combinare depinde de natura problemei și de structura datelor implicate.

10.1.2 Exemplu ilustrativ

Pentru a înțelege clar modul de aplicare al metodei, considerăm exemplul calculului unei puteri x^n printr-o abordare recursivă de tip *Divide et Impera*. În loc să înmulțim de n ori baza x, putem observa că:

$$x^n = \begin{cases} (x^{n/2})^2, & \text{dacă } n \text{ este par} \\ x \cdot (x^{n/2})^2, & \text{dacă } n \text{ este impar} \end{cases}$$

Implementarea în C++ a acestei abordări este următoarea:

```
#include <iostream>
using namespace std;

double putere(double x, int n) {
    if (n == 0) return 1;

    double jumatate = putere(x, n / 2);
    if (n % 2 == 0)
        return jumatate * jumatate;
    else
        return x * jumatate * jumatate;
}

int main() {
    double baza = 2.0;
    int exponent = 10;

    double rezultat = putere(baza, exponent);
    cout << baza << " la puterea " << exponent
```

```
        << " este " << rezultat << endl;
    return 0;
}
```

Prin această metodă, numărul de înmulțiri se reduce semnificativ. Fiecare apel recursiv înjumătățește exponentul, astfel încât complexitatea totală devine $O(\log n)$, în contrast cu soluția iterativă clasică de complexitate $O(n)$.

10.1.3 Complexitatea algoritmilor de tip Divide et Impera

Analiza complexității unui algoritm construit pe această metodă se bazează pe ecuația de recurență generală:

$$T(n) = D(n) + S(n) + C(n)$$

unde:

- $D(n)$ reprezintă timpul necesar descompunerii problemei;
- $S(n)$ este timpul pentru rezolvarea subproblemelor;
- $C(n)$ este timpul de combinare a rezultatelor.

În practică, aceste relații sunt analizate folosind Teorema Master, care permite determinarea ordinii de creștere a timpului de execuție în funcție de factorul de împărțire și de complexitatea combinării.

Metoda *Divide et Impera* ilustrează clar importanța gândirii recursive și a unei organizări raționale a procesului de calcul. Ea permite nu doar simplificarea logicii algoritmilor, ci și obținerea unor performanțe superioare, oferind o bază solidă pentru înțelegerea tehnicilor moderne de optimizare și paralelizare a calculelor.

10.2 Algoritmi clasici

Tehnica *Divide et Impera* stă la baza multor algoritmi esențiali din informatică, care au contribuit la dezvoltarea unor metode rapide și eficiente de procesare a datelor. Prin împărțirea unei probleme mari în subprobleme independente și prin combinarea inteligentă a soluțiilor parțiale, aceste algoritmi oferă performanțe superioare celor iterative clasice.

10.2.1 Căutarea binară

Căutarea binară [3] este unul dintre cele mai simple și mai eficiente exemple de algoritm bazat pe principiul *Divide et Impera*. Ea se aplică exclusiv pe tablouri sortate și permite identificarea poziției unui element printr-o serie de comparații succesive care reduc constant spațiul de căutare.

Ideea de bază este următoarea: la fiecare pas se compară elementul căutat cu elementul din mijlocul intervalului curent. Dacă valoarea căutată este mai mică decât cea din mijloc, căutarea continuă în jumătatea stângă; dacă este mai mare, în jumătatea dreaptă. Procesul se repetă recursiv până la găsirea elementului sau până la epuizarea domeniului de căutare.

Pseudocodul algoritmului:

$$\text{CăutareBinara}(v, st, dr, x) : \begin{cases} \text{dacă } st > dr \text{ return } -1, \\ m = \frac{st+dr}{2} \\ \text{dacă } v[m] = x, \text{ return } m \\ \text{dacă } v[m] > x, \text{ return CăutareBinara}(v, st, m-1, x) \\ \text{altfel, return CăutareBinara}(v, m+1, dr, x) \end{cases}$$

Implementare în C++:

```
#include <iostream>

int cautareBinara(int vector[], int stanga, int dreapta, int
   element) {
    if (right >= stanga) {
        int mijloc = stanga + (dreapta - stanga) / 2;
        if (vector[mijloc] == element)
            return mijloc;
        if (vector[mijloc] > element)
            return cautareBinara(vector, stanga, mijloc - 1,
               element);
        return cautareBinara(vector, mijloc + 1, dreapta, element);
    }
    return -1;
}
```

Analiza complexității La fiecare pas, algoritmul înjumătățește dimensiunea spațiului de căutare, reducând problema la o instanță de dimensiune $n/2$. Prin urmare, timpul total de execuție respectă relația de recurență:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

Rezolvând această ecuație de recurență (de exemplu, folosind Teorema Master sau prin substituție), obținem complexitatea $T(n) = O(\log n)$, ceea ce face căutarea binară semnificativ mai rapidă decât căutarea liniară, care are complexitatea $O(n)$.

10.2.2 Quick Sort

Quick Sort [3] este un exemplu emblematic de algoritm care urmează metoda *Divide et Impera*. Algoritmul sortează un vector printr-un proces recursiv de împărțire și stăpânire, demonstrând clar principiul de descompunere a problemei mari în subprobleme independente și apoi combinarea rezultatelor.

Funcționarea sa se bazează pe alegerea unui element numit *pivot*, care este utilizat pentru a partiționa vectorul în doi subvectori: unul care conține elementele mai mici decât pivotul și altul cu elementele mai mari sau egale. Această etapă de partiționare reprezintă pasul de *divide*. Ulterior, fiecare subvector este sortat recursiv, pas care corespunde etapei de *impera*. În Quick Sort, combinarea este implicită, deoarece subvectorii sunt deja poziționați corect după partiționare, astfel încât nu este nevoie de o etapă suplimentară de fuziune.

Implementare în C++:

Varianta Cormen[3] unde pivotul este pe ultima poziție

```
int partitionare(int vector[], int stanga, int dreapta) {
    int pivot = vector[dreapta];
    int i = stanga - 1;

    for (int j = stanga; j <= dreapta - 1; j++) {
        if (vector[j] < pivot) {
            i++;
            std::swap(vector[i], vector[j]);
        }
```

```
    }
    std::swap(vector[i + 1], vector[dreapta]);
    return i + 1;
}

void quickSort(int vector[], int stanga, int dreapta) {
    if (stanga < dreapta) {
        int pivot = partitionare(vector, stanga, dreapta);
        quickSort(vector, stanga, pivot - 1);
        quickSort(vector, pivot + 1, dreapta);
    }
}
```

Aici pivotul este **vector[dreapta]**, **j** merge de la stânga spre dreapta, iar **i** marchează sfârșitul zonei elementelor mai mici sau egale cu pivotul.

Avantajul este că algoritmul este foarte simplu și este exact genul de implementare prezentat în Cormen. Dezavantajul este că alegerea fixă a ultimului element ca pivot poate produce cazuri foarte proaste. De exemplu, pentru un vector deja sortat, partiționarea va fi foarte dezechilibrată.

Varianta Cormen[3] unde pivotul este ales aleatoriu (random)

```
int partitionareRandom(std::vector<int>& v, int st, int dr) {
    int pozitiePivot = st + rand() % (dr - st + 1);

    std::swap(v[pozitiePivot], v[dr]);

    int pivot = v[dr];
    int i = st - 1;

    for (int j = st; j < dr; j++) {
        if (v[j] <= pivot) {
            i++;
            std::swap(v[i], v[j]);
        }
    }
    std::swap(v[i + 1], v[dr]);

    return i + 1;
}

void quickSort(std::vector<int>& v, int st, int dr) {
    if (st < dr) {
        int p = partitionareRandom(v, st, dr);
```

```
        quickSort(v, st, p - 1);
        quickSort(v, p + 1, dr);
    }
}
```

Idea de bază este că alegem aleator o poziție, mutăm elementul respectiv pe ultima poziție și apoi folosim exact aceeași partiționare ca la prima variantă. Ideea importantă este că nu trebuie să modifici algoritmul de partiționare doar pentru că pivotul este ales random. Alegi pivotul random, îl muți în poziția pe care algoritmul o așteaptă și apoi continui normal.

Varianta cu doi indici care vin din capete diferite

```
int partitionare(std::vector<int>& v, int st, int dr) {
    int pivot = v[(st + dr) / 2];

    int i = st - 1;
    int j = dr + 1;

    while (true) {
        do {
            i++;
        } while (v[i] < pivot);

        do {
            j--;
        } while (v[j] > pivot);

        if (i >= j) {
            return j;
        }
        std::swap(v[i], v[j]);
    }
}
void quickSort(std::vector<int>& v, int st, int dr) {
    if (st < dr) {
        int p = partitionare(v, st, dr);
        quickSort(v, st, p);
        quickSort(v, p + 1, dr);
    }
}
```

În această variantă **i** pornește din stânga și crește, iar **j** pornește din dreapta și scade.

Varianta cu doi indici care vin din capete diferite unde pivotul este ales la întâmplare

În această variantă nu avem nevoie să mutăm pivotul la final. Putem pur și simplu să alegem o poziție random și să salvăm valoarea ei, apoi folosim valoarea pivot în cele două căutări.

```
int pozitiePivot = st + rand() % (dr - st + 1);
int pivot = v[pozitiePivot];
```

Restul de cod rămâne perfect identic:

```
#include <vector>
#include <iostream>

int partitionare(std::vector<int>& v, int st, int dr) {
    int pozitiePivot = st + rand() % (dr - st + 1);
    int pivot = v[pozitiePivot];

    int i = st - 1;
    int j = dr + 1;

    while (true) {
        do {
            i++;
        } while (v[i] < pivot);

        do {
            j--;
        } while (v[j] > pivot);

        if (i >= j) {
            return j;
        }
        std::swap(v[i], v[j]);
    }
}

void quickSort(std::vector<int>& v, int st, int dr) {
    if (st < dr) {
        int p = partitionare(v, st, dr);

        quickSort(v, st, p);
        quickSort(v, p + 1, dr);
    }
}

int main()
{
    std::vector<int> vector = { 4, 2, 2, 8, 3, 3, 1 };
```

```
    quickSort(vector, 0, vector.size() - 1);
    for (int element : vector)
        std::cout << element << " ";

    return 0;
}
```

Analiza complexității Complexitatea algoritmului de sortare rapidă depinde în mare măsură de alegerea pivotului și de modul în care acesta partiționează șirul. Fie n numărul de elemente din șirul de sortat.

- **Caz favorabil**: Acest caz apare atunci când pivotul ales împarte șirul în două sub-șiruri de dimensiuni aproape egale la fiecare pas. Ideal, pivotul este întotdeauna elementul median. În această situație, relația de recurență este similară cu cea a Merge Sort:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

  unde $2T\left(\frac{n}{2}\right)$ reprezintă timpul necesar pentru a sorta cele două sub-șiruri de dimensiune $n/2$, iar $O(n)$ este timpul necesar pentru operația de partiționare (parcurgerea șirului o dată).

  Rezolvând această recurență, obținem complexitatea $O(n \log n)$.

- **Caz mediu**: În practică, alegerea pivotului nu este întotdeauna ideală, dar nici cea mai proastă. Dacă pivotul împarte șirul într-un raport aproximativ constant (de exemplu, 1:9, 2:8 etc.), complexitatea medie rămâne logaritmică. Se demonstrează că, în medie, Quick Sort are o performanță excelentă.

  Complexitate: $O(n \log n)$.

- **Caz nefavorabil**: Acest caz apare atunci când alegerea pivotului este cea mai slabă posibilă, adică pivotul este întotdeauna cel mai mic sau cel mai mare element din sub-șirul curent. Aceasta duce la o partiționare inegală, unde un sub-șir are $n - 1$ elemente, iar celălalt are 0 elemente. Relația de recurență devine:

$$T(n) = T(n-1) + T(0) + O(n) \implies T(n) = T(n-1) + O(n)$$

  Rezolvând această recurență, obținem complexitatea: $O(n^2)$ (timp pătratic).

Acest scenariu se întâmplă, de exemplu, dacă șirul este deja sortat (sau invers sortat) și pivotul este ales întotdeauna ca primul sau ultimul element. Pentru a evita cazul nefavorabil, se utilizează diverse strategii de alegere a pivotului, cum ar fi:

- Alegerea unui pivot aleatoriu.
- Alegerea medianei din trei elemente (primul, mijlociul și ultimul).

Aceste tehnici ajută la menținerea performanței Quick Sort aproape de cazul mediu, care este $O(n \log n)$, făcându-l un algoritm foarte eficient în practică.

Quick Sort ilustrează perfect principiile *Divide et Impera*: problema inițială este împărțită, subproblemele sunt rezolvate recursiv, iar rezultatele sunt combinate eficient pentru a obține vectorul sortat.

10.2.3 Merge Sort

Merge Sort [3] este un algoritm de sortare clasic care aplică cu strictețe principiile metodei *Divide et Impera*. Algoritmul urmează toate cele trei etape fundamentale: descompunere, rezolvare și combinare. În primul pas, vectorul este împărțit în doi sub-vectori aproximativ egali (*divide*). Fiecare sub-vector este sortat recursiv (*impera*), iar în final, cei doi sub-vectori sortați sunt interclasați (*combine*) pentru a forma vectorul complet sortat.

Un aspect distinctiv al Merge Sort față de Quick Sort este că etapa de combinare este esențială și consumă o parte semnificativă din timp. Această etapă implică crearea unui vector temporar în care elementele celor doi sub-vectori sortați sunt intercalate în ordine crescătoare, iar apoi valorile sunt copiate înapoi în vectorul inițial.

Implementare în C++:

```
void merge(int vector[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = vector[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = vector[mid + 1 + j];
```

```
    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            vector[k] = L[i];
            i++;
        } else {
            vector[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        vector[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        vector[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int vector[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(vector, left, mid);
        mergeSort(vector, mid + 1, right);

        merge(vector, left, mid, right);
    }
}
```

Analiza complexității Performanța algoritmului Merge Sort este consistentă, având aceeași complexitate în timp în toate cazurile (favorabil, mediu, nefavorabil). Fie n numărul de elemente din șirul de sortat.

Principiul Merge Sort implică trei pași principali:

1. Șirul este împărțit în două subșiruri de dimensiuni aproximativ egale. Această operație durează $O(1)$ (doar calculul poziției de mijloc).

2. Cele două subșiruri sunt sortate recursiv folosind Merge Sort.

3. Cele două subșiruri sortate sunt interclasate (unite) într-un singur șir sortat. Această operație necesită parcurgerea ambelor sub-șiruri o singură dată, având o complexitate de $O(k)$ pentru interclasarea a două șiruri cu un total de k elemente. Pentru un șir de dimensiune n, interclasarea durează $O(n)$.

Pe baza acestor pași, putem stabili relația de recurență pentru timpul de execuție $T(n)$:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

unde:

- $2T\left(\frac{n}{2}\right)$ reprezintă timpul necesar pentru a sorta recursiv cele două sub-șiruri, fiecare de dimensiune $n/2$.
- $O(n)$ reprezintă timpul necesar pentru operația de interclasare (combinare) a celor două sub-șiruri sortate.

Această relație de recurență poate fi rezolvată folosind Teorema Master (cazul 2), sau prin metoda arborelui de recurență:

- La fiecare nivel al recursiei, se efectuează $O(n)$ operații de interclasare.
- Numărul de niveluri ale recursiei (adâncimea arborelui) este $\log_2 n$, deoarece la fiecare pas dimensiunea problemei se înjumătățește până ajunge la un singur element.

Prin urmare, timpul total de execuție este produsul dintre numărul de niveluri și costul pe nivel:

$$T(n) = (\text{număr niveluri}) \times (\text{cost pe nivel}) = \log_2 n \times O(n) = O(n \log n)$$

Astfel, deoarece Merge Sort împarte întotdeauna șirul în bucăți egale și costul operației de interclasare este întotdeauna $O(n)$, complexitatea sa în timp este aceeași, indiferent de ordinea inițială a elementelor din șir.

În plus, Merge Sort necesită un spațiu auxiliar suplimentar pentru a stoca subșirurile interclasate. În implementarea standard, este nevoie de un șir temporar de dimensiune $O(n)$ pentru

operația de interclasare. Acesta este un dezavantaj față de Quick Sort, care poate fi implementat "in-place" (cu spațiu auxiliar $O(\log n)$ pentru stiva de recursie).

Merge Sort este apreciat pentru stabilitatea sa (elementele cu valori egale își păstrează ordinea relativă) și pentru performanța sa garantată de $O(n \log n)$, fiind o alegere excelentă atunci când consistența performanței este esențială.

Merge Sort ilustrează clar principiile *Divide et Impera*: problema inițială este împărțită în subprobleme independente, fiecare subproblemă este rezolvată recursiv, iar rezultatele sunt combinate pentru a obține soluția finală.

10.2.4 Algoritmul Karatsuba

Algoritmul Karatsuba [17] este un exemplu impresionant de aplicație a tehnicii *Divide et Impera* pentru optimizarea operațiilor matematice. În loc să folosească metoda clasică de înmulțire, care are complexitatea $O(n^2)$ pentru numere cu n cifre, Karatsuba reduce numărul de înmulțiri necesare, obținând o complexitate de aproximativ $O(n^{\log_2 3}) \approx O(n^{1.585})$.

Se consideră două numere mari X și Y (numere exprimate în baza 10) descompuse astfel:

$$X = 10^m \cdot a + b, \quad Y = 10^m \cdot c + d$$

în care m reprezintă jumătatea lungimii maxime a numerelor, iar a, b, c, d sunt subnumerele corespunzătoare. În loc să calculăm patru înmulțiri clasice $a \cdot c, a \cdot d, b \cdot c, b \cdot d$, algoritmul Karatsuba folosește doar trei înmulțiri:

$$z_2 = a \cdot c, \quad z_0 = b \cdot d, \quad z_1 = (a + b) \cdot (c + d)$$

Rezultatul final se obține prin combinația:

$$X \cdot Y = 10^{2m} \cdot z_2 + 10^m \cdot (z_1 - z_2 - z_0) + z_0$$

Exemplu numeric: Să considerăm înmulțirea $12 \cdot 34$. Se descompune astfel:

$$12 = 1 \cdot 10 + 2, \quad 34 = 3 \cdot 10 + 4$$

$$\left|\begin{array}{l|l} a=1 & b=2 \\ c=3 & d=4 \end{array}\right.$$

Se calculează:

$$z_2 = a \cdot c = 1 \cdot 3 = 3$$

$$z_0 = b \cdot d = 2 \cdot 4 = 8$$

$$z_1 = (a+b) \cdot (c+d) = 3 \cdot 7 = 21$$

Rezultatul final:

$$12 \cdot 34 = 10^2 \cdot 3 + 10 \cdot (21 - 3 - 8) + 8 = 300 + 100 + 8 = 408$$

Implementare C++:

```
#include <string>
#include <iostream>
#include <algorithm>

long long karatsuba_long_long(long long x, long long y) {
    if (x < 10 || y < 10)
        return x * y;
    int n = std::max(std::to_string(x).length(),
       std::to_string(y).length());
    int m = n / 2;
    long long putere = (long long)pow(10, m);
    long long high1 = x / pow(10, m);
    long long low1 = x % putere;
    long long high2 = y / pow(10, m);
    long long low2 = y % putere;
    long long z0 = karatsuba_long_long(low1, low2);
    long long z2 = karatsuba_long_long(high1, high2);
    long long z1 = karatsuba_long_long(low1 + high1, low2 + high2);
    return z2 * putere * putere + (z1 - z2 - z0) * putere + z0;
}

std::string adunare_string(std::string a, std::string b) {
    std::string rezultat;

    int i = a.size() - 1;
    int j = b.size() - 1;
    int transport = 0;
```

```
    while (i >= 0 || j >= 0 || transport) {
        int suma = transport;

        if (i >= 0)
            suma += a[i--] - '0';

        if (j >= 0)
            suma += b[j--] - '0';

        rezultat += char('0' + suma % 10);
        transport = suma / 10;
    }

    std::reverse(rezultat.begin(), rezultat.end());

    return rezultat;
}

std::string scadere_string(std::string a, std::string b) {
    std::string rezultat;

    int i = a.size() - 1;
    int j = b.size() - 1;
    int imprumut = 0;

    while (i >= 0) {
        int diferenta = (a[i] - '0') - imprumut;

        if (j >= 0)
            diferenta -= b[j--] - '0';

        if (diferenta < 0) {
            diferenta += 10;
            imprumut = 1;
        }
        else {
            imprumut = 0;
        }
        rezultat += char('0' + diferenta);
        i--;
    }

    std::reverse(rezultat.begin(), rezultat.end());

    // eliminam zerourile de la inceput
    int poz = rezultat.find_first_not_of('0');
```

```
    if (poz == std::string::npos)
        return "0";

    return rezultat.substr(poz);
}

std::string karatsuba_string(std::string x, std::string y) {
    // eliminam zerourile de la inceput
    x.erase(0, x.find_first_not_of('0'));
    y.erase(0, y.find_first_not_of('0'));

    if (x.empty()) x = "0";
    if (y.empty()) y = "0";

    // cazul de baza
    if (x.size() <= 2 || y.size() <= 2) {
        int a = stoi(x);
        int b = stoi(y);
        return std::to_string(a * b);
    }

    // facem numerele de aceeasi lungime
    int n = std::max(x.size(), y.size());

    if (n % 2 != 0)
        n++;

    while (x.size() < n)
        x = "0" + x;

    while (y.size() < n)
        y = "0" + y;

    int m = n / 2;

    // impartim numerele
    std::string high1 = x.substr(0, m);
    std::string low1 = x.substr(m);

    std::string high2 = y.substr(0, m);
    std::string low2 = y.substr(m);

    // cele trei produse Karatsuba
    std::string z0 = karatsuba_string(low1, low2);
    std::string z2 = karatsuba_string(high1, high2);
```

```
    std::string suma1 = adunare_string(low1, high1);
    std::string suma2 = adunare_string(low2, high2);

    std::string z1 = karatsuba_string(suma1, suma2);

    // z1 = z2 + partea din mijloc + z0
    std::string mijloc = scadere_string(scadere_string(z1, z2),
       z0);

    // 10^m pentru numere reprezentate ca string
    std::string putere10_m = "1" + std::string(m, '0');

    // 10^(2m)
    std::string putere10_2m = "1" + std::string(2 * m, '0');

    // z2 * 10^(2m)
    std::string rezultat = z2 + std::string(2 * m, '0');

    // partea din mijloc * 10^m
    std::string parteaMijloc = mijloc + std::string(m, '0');

    rezultat = adunare_string(rezultat, parteaMijloc);
    rezultat = adunare_string(rezultat, z0);

    return rezultat;
}

int main()
{
    // pe valori mai mici
    std::cout << karatsuba_long_long(105, 207) << "\n";
    std::cout << karatsuba_string("105", "207") << "\n";

    // pe valori mari
    std::cout << karatsuba_long_long(123456789, 987654321) << "\n";
    //std::cout << karatsuba_long_long(123456789123456789,
       987654321987654321) << "\n"; // prea mare pentru long long
    std::cout << karatsuba_string("123456789123456789",
       "987654321987654321") << "\n";

    return 0;
}
```

Important: funcția **pow** are complexitatea $O(n)$.

Analiza complexității Pe baza descompunerii și a formulelor prezentate anterior pentru înmulțirea numerelor mari X și Y de n cifre, putem stabili relația de recurență pentru timpul de execuție $T(n)$ al algoritmului Karatsuba.

După cum s-a menționat, algoritmul Karatsuba reduce numărul de înmulțiri recursive la trei: $z_2 = a \cdot c$, $z_0 = b \cdot d$, și $z_1 = (a + b) \cdot (c + d)$. Fiecare dintre aceste înmulțiri operează pe numere de aproximativ $n/2$ cifre (sau m, conform notației date). Operațiile de adunare, scădere și shiftare (înmulțire cu puteri ale lui 10, adică 10^{2m} și 10^m) necesare pentru a combina rezultatele parțiale (precum $z_1 - z_2 - z_0$) au o complexitate de $O(n)$, deoarece implică parcurgerea cifrelor numerelor.

Astfel, relația de recurență pentru timpul de execuție $T(n)$ este:

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

unde:

- $3T\left(\frac{n}{2}\right)$ reprezintă timpul necesar pentru cele trei înmulțiri recursive, fiecare pe numere de dimensiune aproximativ $n/2$.
- $O(n)$ reprezintă timpul necesar pentru operațiile de adunare, scădere și shiftare (combinarea rezultatelor parțiale).

Această relație de recurență se încadrează în Cazul 1 al Teoremei Master, cu parametrii $a = 3$, $b = 2$, și $f(n) = O(n)$. Pentru a aplica Teorema Master, comparăm $f(n)$ cu $n^{\log_b a}$: $n^{\log_2 3} \approx n^{1.585}$ Deoarece $f(n) = O(n)$ și $n = O(n^{\log_2 3})$ (mai precis, $n = O(n^{1.585-\epsilon})$ pentru un $\epsilon > 0$), rezultă că complexitatea este dominată de termenul $n^{\log_b a}$.

Prin urmare, rezolvând această recurență, obținem complexitatea în timp a algoritmului Karatsuba:

$$T(n) = O(n^{\log_2 3}) \approx O(n^{1.585})$$

Această complexitate de $O(n^{1.585})$ este asimptotic superioară complexității de $O(n^2)$ a metodei tradiționale de înmulțire. Această îmbunătățire devine semnificativă pentru numere foarte mari (cu multe cifre), unde reducerea numărului de înmulțiri recursive de la patru la trei are un impact major asupra performanței. Deși pentru numere mici, costul operațiilor auxiliare

și al recursivității poate face ca metoda tradițională să fie mai rapidă, pentru n suficient de mare, Karatsuba este algoritmul preferat.

Algoritmul Karatsuba arată cum metoda *Divide et Impera* reduce complexitatea problemelor aparent simple printr-o descompunere inteligentă a subproblemelor și combinarea rezultatelor obținute, economisind operații costisitoare și oferind o performanță superioară algoritmului clasic de înmulțire.

10.2.5 Șirul lui Fibonacci

Șirul lui Fibonacci oferă un exemplu ilustrativ pentru aplicarea tehnicii *Divide et Impera*. Definiția matematică a șirului este:

$$F(n) = F(n-1) + F(n-2), \quad F(0) = 0, \quad F(1) = 1$$

Abordarea iterativă liniară

Algoritmul calculează fiecare termen o singură dată, deci are complexitatea O(n), iar deoarece păstrează doar ultimele două valori, complexitatea spațială este O(1)[3].

```
long long fibonacci(int n) {
    if (n <= 1)
        return n;

    long long a = 0;
    long long b = 1;
    int i = 2;

    while (i <= n) {
        long long c = a + b;
        a = b;
        b = c;
        i++;
    }

    return b;
}
```

Abordarea recursivă naivă

Implementarea recursivă directă reflectă exact definiția matematică, dar suferă de o eficiență redusă, având complexitatea $O(2^n)$ din cauza recalculării repetate a acelorași subprobleme.

```
int fibonacciNaiv(int n) {
    if (n <= 1) return n;
    return fibonacciNaiv(n - 1) + fibonacciNaiv(n - 2);
}
```

Această implementare evidențiază un pericol al metodei *Divide et Impera*: când subproblemele se suprapun și nu sunt independente, eficiența algoritmului scade drastic.

Abordarea cu exponențierea rapidă a matricilor

O strategie mult mai eficientă folosește proprietățile matriciale ale șirului Fibonacci. Se observă că:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{bmatrix}$$

Astfel, $F(n)$ poate fi calculat prin ridicarea la putere a unei matrice 2×2 folosind *Divide et Impera*.

```
// Inmultirea a doua matrici 2x2
vector<vector<long long>> multiplyMatrix(vector<vector<long long>>
   A, vector<vector<long long>> B) {
    vector<vector<long long>> C(2, vector<long long>(2, 0));
    C[0][0] = A[0][0]*B[0][0] + A[0][1]*B[1][0];
    C[0][1] = A[0][0]*B[0][1] + A[0][1]*B[1][1];
    C[1][0] = A[1][0]*B[0][0] + A[1][1]*B[1][0];
    C[1][1] = A[1][0]*B[0][1] + A[1][1]*B[1][1];
    return C;
}

// Ridicare la putere folosind Divide et Impera
vector<vector<long long>> matrixPower(vector<vector<long long>> M,
   int n) {
    if (n == 1) return M;
    if (n % 2 == 0) {
        vector<vector<long long>> half = matrixPower(M, n/2);
        return multiplyMatrix(half, half);
    } else {
        vector<vector<long long>> half = matrixPower(M, (n-1)/2);
        vector<vector<long long>> temp = multiplyMatrix(half,
           half);
```

```
        return multiplyMatrix(temp, M);
    }
}

long long fibonacciEficient(int n) {
    if (n <= 1) return n;
    vector<vector<long long>> base = {{1,1},{1,0}};
    vector<vector<long long>> result = matrixPower(base, n-1);
    return result[0][0];
}
```

Analiza complexității Abordarea cu exponențierea rapidă a matricilor pentru calculul numerelor Fibonacci transformă problema într-una de ridicare la putere a unei matrici, folosind tehnica *Divide et Impera*. Această metodă oferă o îmbunătățire semnificativă a complexității temporale față de calculul recursiv naiv al numerelor Fibonacci și față de calculul iterativ simplu, care necesită $O(n)$ operații.

Calculul lui $F(n)$ se reduce la calculul puterii a $n-1$-a a matricei de bază

$$M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}.$$

Problema devine, așadar, calculul lui M^k, unde $k = n-1$. Această operație se realizează eficient folosind tehnica de exponențiere rapidă, numită și exponențiere binară, care este o aplicație a paradigmei *Divide et Impera*.

Pentru calculul complexității totale vom considera:

- **Complexitatea operației de înmulțire a matricilor** 2×2**:**
  O înmulțire a două matrici 2×2 necesită un număr fix de operații aritmetice elementare. Mai precis, sunt necesare 8 înmulțiri scalare și 4 adunări scalare. Deoarece acest număr este constant și independent de n, complexitatea unei singure operații de înmulțire a matricilor 2×2 este $O(1)$.
- **Complexitatea algoritmului de exponențiere rapidă:**
  Algoritmul de exponențiere rapidă calculează M^k prin reducerea repetată a exponentului la jumătate. Principiul este următorul:
  - Dacă k este par, $M^k = (M^{k/2})^2$.

– Dacă k este impar, $M^k = M \cdot (M^{(k-1)/2})^2$.

La fiecare pas, exponentul k este împărțit aproximativ la 2, iar la fiecare astfel de pas se efectuează un număr constant de înmulțiri de matrici 2×2.

Exponențierea rapidă poate fi implementată atât recursiv, cât și iterativ. În varianta recursivă, reducerea exponentului la jumătate este realizată prin apeluri recursive, urmând principiul *Divide et Impera*. În varianta iterativă, același principiu este aplicat folosind reprezentarea binară a exponentului și actualizând succesiv rezultatul și baza.

Din punct de vedere al complexității temporale, cele două variante sunt echivalente. Numărul de pași necesari pentru a reduce exponentul de la k la 1 este $O(\log k)$, iar la fiecare pas se efectuează un număr constant de înmulțiri de matrici 2×2, fiecare având complexitatea $O(1)$. Prin urmare, complexitatea temporală a exponențierii rapide pentru o matrice 2×2 este $O(\log k)$, atât în varianta recursivă, cât și în cea iterativă.

Diferența dintre cele două variante apare la nivelul complexității spațiale. Varianta recursivă necesită $O(\log k)$ memorie suplimentară pentru stiva apelurilor recursive, în timp ce varianta iterativă necesită $O(1)$ memorie suplimentară, deoarece nu folosește apeluri recursive.

În contextul calculului $F(n)$, exponentul este $n-1$. Prin urmare, complexitatea temporală pentru a calcula M^{n-1} este

$$O(\log(n-1)) = O(\log n).$$

Varianta recursivă are astfel complexitatea $O(\log n)$ din punct de vedere temporal și $O(\log n)$ din punct de vedere spațial, în timp ce varianta iterativă are complexitatea $O(\log n)$ din punct de vedere temporal și $O(1)$ din punct de vedere spațial.

10.2.6 Turnurile din Hanoi

Problema Turnurilor din Hanoi este un exemplu clasic care ilustrează perfect principiul *Divide et Impera*. Scopul este mutarea unui set de discuri de dimensiuni diferite de pe un turn sursă pe un turn destinație, folosind un turn auxiliar, respectând regulile:

- Se poate muta un singur disc odată.

- Un disc mai mare nu poate fi plasat peste un disc mai mic.

Inițial discurile sunt plasate în ordine descrescătoare a dimensiunii lor pe turnul sursă (pornind de la bază).

Abordarea recursivă

Soluția recursivă descompune problema în trei subprobleme mai mici, fiecare rezolvabilă în mod independent:

1. Mută $n-1$ discuri de la turnul sursă la turnul auxiliar, folosind destinația ca auxiliar.
2. Mută discul cel mai mare de la turnul sursă la turnul destinație.
3. Mută $n-1$ discuri de la turnul auxiliar la turnul destinație, folosind sursa ca auxiliar.

```
void hanoi(int n, char sursa, char destinatie, char auxiliar) {
    if (n == 1) {
        cout << "Muta discul 1 de la " << sursa << " la " <<
           destinatie << endl;
        return;
    }

    // Muta n-1 discuri de la sursa la auxiliar
    hanoi(n - 1, sursa, auxiliar, destinatie);

    // Muta discul ramas de la sursa la destinatie
    cout << "Muta discul " << n << " de la " << sursa << " la " <<
       destinatie << endl;

    // Muta n-1 discuri de la auxiliar la destinatie
    hanoi(n - 1, auxiliar, destinatie, sursa);
}
```

Analiza complexității

Numărul total de mutări necesare pentru n discuri se exprimă prin relația de recurență:

$$T(n) = 2T(n-1) + 1$$

care se rezolvă în forma:

$$T(n) = 2^n - 1$$

Astfel, complexitatea este $O(2^n)$, demonstrând creșterea exponențială a pașilor necesari odată cu adăugarea de discuri.

10.2.7 Algoritmul lui Strassen pentru înmulțirea matricilor

Algoritmul lui Strassen [3] reprezintă o aplicație remarcabilă a tehnicii *Divide et Impera* în algebra liniară, oferind o îmbunătățire asimptotică față de algoritmul clasic de înmulțire a matricilor.

Abordarea tradițională vs Strassen

Algoritmul clasic de înmulțire a matricilor are complexitatea $O(n^3)$. La fiecare descompunere recursivă se efectuează 8 înmulțiri și 4 adunări pentru submatrici. Strassen a observat că se pot efectua doar 7 înmulțiri recursive, folosind combinații algebrice ingenioase.

Principiul Divide et Impera

Divide: Matricele A și B de dimensiune $n \times n$ se împart în 4 submatrici de dimensiune $n/2 \times n/2$:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Impera: Se calculează 7 produse mai mici:

$$\begin{aligned} P_1 &= A_{11} \cdot (B_{12} - B_{22}) \\ P_2 &= (A_{11} + A_{12}) \cdot B_{22} \\ P_3 &= (A_{21} + A_{22}) \cdot B_{11} \\ P_4 &= A_{22} \cdot (B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\ P_6 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \\ P_7 &= (A_{11} - A_{21}) \cdot (B_{11} + B_{12}) \end{aligned}$$

Combină: Submatricile rezultatului C se calculează astfel:

$$C_{11} = P_5 + P_4 - P_2 + P_6$$
$$C_{12} = P_1 + P_2$$
$$C_{21} = P_3 + P_4$$
$$C_{22} = P_1 + P_5 - P_3 - P_7$$

Analiza complexității Analiza complexității algoritmului lui Strassen [3] este prezentată în cele ce urmează.

Pentru a înmulți două matrici pătrate de dimensiune $N \times N$, metoda clasică necesită N^3 înmulțiri scalare și $(N-1)N^2$ adunări scalare, rezultând o complexitate temporală de $O(N^3)$.

După cum s-a expus mai sus, algoritmul Strassen împarte recursiv fiecare matrice $N \times N$ în patru submatrici de dimensiune $N/2 \times N/2$. În loc să efectueze cele 8 înmulțiri de submatrici necesare în abordarea *Divide et Impera*, Strassen reușește să calculeze produsul folosind doar 7 înmulțiri de submatrici, în detrimentul unui număr mai mare de adunări și scăderi de submatrici.

Fie $T(N)$ timpul de execuție pentru înmulțirea a două matrici de dimensiune $N \times N$ folosind algoritmul Strassen. Relația de recurență este:

$$T(N) = 7T\left(\frac{N}{2}\right) + O(N^2)$$

unde:

- $7T\left(\frac{N}{2}\right)$ reprezintă timpul necesar pentru cele șapte înmulțiri recursive de submatrici de dimensiune $N/2 \times N/2$.
- $O(N^2)$ reprezintă timpul necesar pentru operațiile de adunare și scădere a submatricilor, precum și pentru combinarea rezultatelor parțiale. Aceste operații implică parcurgerea elementelor matricilor, având o complexitate pătratică.

Această relație de recurență poate fi rezolvată folosind Teorema Master (cazul 1), cu parametrii $a = 7$, $b = 2$, și $f(N) = O(N^2)$. Comparăm $f(N)$ cu $N^{\log_b a}$: $N^{\log_2 7} \approx N^{2.807}$ Deoarece $f(N) = O(N^2)$ și $N^2 = O(N^{\log_2 7})$, complexitatea este dominată de termenul $N^{\log_b a}$.

Astfel, complexitatea temporală a algoritmului Strassen este:

$$T(N) = O(N^{\log_2 7}) \approx O(N^{2.807})$$

Algoritmul Strassen necesită spațiu auxiliar pentru stocarea submatricilor intermediare și pentru stiva de recursie. La fiecare nivel de recursie, sunt create un număr constant de submatrici de dimensiune $N/2 \times N/2$. Adâncimea recursiei este $O(\log N)$. Prin urmare, complexitatea spațială totală este $O(N^2)$, similară cu cea a matricilor de intrare.

Algoritmul lui Strassen pentru matrici 3x3

Deși algoritmul clasic al lui Strassen este optimizat pentru matrici 2×2 și, prin recursivitate, pentru matrici de dimensiune putere a lui 2, metoda poate fi adaptată și pentru matrici 3×3 folosind principiile *Divide et Impera*.

Principiu general

- **Divide:** Matricea 3×3 poate fi împărțită în submatrici 2×2 și elemente individuale sau, alternativ, folosind tehnici de descompunere specifică pentru dimensiuni neregulate.
- **Impera:** Se efectuează înmulțiri recursive pentru submatricile 2×2 și combinații algebrice pentru elementele rămase.
- **Combină:** Rezultatele submatricilor și elementelor individuale se combină pentru a forma matricea finală 3×3.

Exemplu schematic pentru 3×3

Fie matricele:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

Se poate folosi o descompunere în submatrici 2×2 plus elemente rămase:

$$A = \begin{bmatrix} A_{2x2} & a_{13:23} \\ a_{31:32} & a_{33} \end{bmatrix}, \quad B = \begin{bmatrix} B_{2x2} & b_{13:23} \\ b_{31:32} & b_{33} \end{bmatrix}$$

Pași:

1. Înmulțirea submatricilor 2×2 folosind algoritmul Strassen 2×2.
2. Înmulțirea elementelor rămase individuale cu submatrici corespunzătoare.
3. Combinarea tuturor produselor pentru a obține rezultatul final 3×3.

Observații

- Pentru matrici 3×3, avantajul Strassen este mai redus decât pentru matrici mari, deoarece costul combinării și manipulării elementelor individuale devine relativ mare.
- Tehnica *Divide et Impera* rămâne utilă pentru scalabilitate și implementări recursive pentru matrici de dimensiuni arbitrare.
- În practică, pentru dimensiuni mici (cum este 3×3), algoritmul clasic poate fi mai eficient din cauza constantei multiplicative mai mici.

Implementare în C++ pentru matrici de 2x2

```
vector<vector<int>> strassen2x2(vector<vector<int>> A,
                                vector<vector<int>> B) {
    // Elementele matricilor A si B
    int a = A[0][0];
    int b = A[0][1];
    int c = A[1][0];
    int d = A[1][1];

    int e = B[0][0];
    int f = B[0][1];
    int g = B[1][0];
    int h = B[1][1];

    // Cele 7 produse ale algoritmului Strassen
    int M1 = (a + d) * (e + h);
    int M2 = (c + d) * e;
    int M3 = a * (f - h);
```

```
    int M4 = d * (g - e);
    int M5 = (a + b) * h;
    int M6 = (c - a) * (e + f);
    int M7 = (b - d) * (g + h);

    // Matricea rezultat C = A * B
    vector<vector<int>> C(2, vector<int>(2));

    C[0][0] = M1 + M4 - M5 + M7;
    C[0][1] = M3 + M5;
    C[1][0] = M2 + M4;
    C[1][1] = M1 - M2 + M3 + M6;

    return C;
}
```

Implementare în C++ pentru matrici de 3x3

```
vector<vector<int>> strassen3x3(vector<vector<int>> A,
   vector<vector<int>> B) {
    // Descompune A si B in submatrici 2x2 + elemente ramase
    vector<vector<int>> A2x2 = {{A[0][0], A[0][1]}, {A[1][0],
       A[1][1]}};
    vector<vector<int>> B2x2 = {{B[0][0], B[0][1]}, {B[1][0],
       B[1][1]}};

    vector<vector<int>> C2x2 = strassen2x2(A2x2, B2x2);

    // Combina rezultatele cu elementele ramase
    vector<vector<int>> C(3, vector<int>(3,0));
    C[0][0] = C2x2[0][0] + A[0][2]*B[2][0];
    C[0][1] = C2x2[0][1] + A[0][2]*B[2][1];
    C[0][2] = A[0][0]*B[0][2] + A[0][1]*B[1][2] + A[0][2]*B[2][2];
    C[1][0] = C2x2[1][0] + A[1][2]*B[2][0];
    C[1][1] = C2x2[1][1] + A[1][2]*B[2][1];
    C[1][2] = A[1][0]*B[0][2] + A[1][1]*B[1][2] + A[1][2]*B[2][2];
    C[2][0] = A[2][0]*B[0][0] + A[2][1]*B[1][0] + A[2][2]*B[2][0];
    C[2][1] = A[2][0]*B[0][1] + A[2][1]*B[1][1] + A[2][2]*B[2][1];
    C[2][2] = A[2][0]*B[0][2] + A[2][1]*B[1][2] + A[2][2]*B[2][2];

    return C;
}
```

Algoritmul de bordare

Algoritmul Strassen este definit în mod natural pentru matrici a căror dimensiune poate fi împărțită în două părți egale. Pentru o matrice de dimensiune $n \times n$ care nu poate fi împărțită

exact în acest mod, se poate utiliza metoda de *bordare* (padding). Aceasta presupune adăugarea unor linii și coloane suplimentare, astfel încât matricea să ajungă la o dimensiune potrivită pentru aplicarea recursivă a algoritmului Strassen.

De exemplu, pentru o matrice 3×3, aceasta poate fi extinsă la o matrice 4×4, completând linia și coloana adăugate cu zerouri. Aceeași operație se realizează și pentru cea de-a doua matrice. După această extindere, algoritmul Strassen poate împărți matricele în patru submatrici de dimensiune 2×2 și poate continua calculul în mod normal.

După efectuarea înmulțirii matricelor extinse, rezultatul obținut va avea dimensiunea 4×4. Deoarece liniile și coloanele suplimentare au fost completate cu zerouri, rezultatul relevant se află în submatricea corespunzătoare dimensiunii inițiale. Astfel, pentru cazul 3×3, se păstrează doar primele 3 linii și primele 3 coloane ale rezultatului.

Prin urmare, bordarea permite aplicarea algoritmului Strassen și pentru matrici ale căror dimensiuni nu sunt puteri ale lui 2, fără a modifica principiul algoritmului. Zerourile adăugate nu influențează rezultatul final al înmulțirii, iar după terminarea calculului acestea sunt eliminate din matricea rezultat.

Concluzii paradigma Divide et Impera

Pe lângă aspectele menționate anterior, tehnica Divide et Impera oferă câteva beneficii importante:

- **Claritate și modularitate:** Problemele sunt împărțite în subprobleme independente, ceea ce facilitează scrierea și testarea codului.
- **Scalabilitate:** Algoritmii pot fi extinși pentru dimensiuni mari de date prin recursivitate și, în unele cazuri, paralelizare.
- **Reutilizare:** Subproblemele recurente permit reutilizarea funcțiilor și structurilor de date deja implementate.
- **Analiză matematică mai simplă:** Ecuațiile de recurență și Teorema Master permit estimarea precisă a complexității algoritmilor.
- **Optimizare naturală pentru paralelism:** Subproblemele independente pot fi rezolvate simultan pe mai multe fire de execuție sau procesoare.

Tehnica Divide et Impera este una dintre cele mai elegante și versatile metode de proiectare a algoritmilor. Aplicată corect, permite:

- Transformarea problemelor complexe în subprobleme mai simple, cu soluții mai rapide.
- Obținerea unor algoritmi cu complexități asimptotice mult mai bune decât abordările naive.
- Îmbunătățirea clarității și modularității codului, ceea ce facilitează mentenanța și extinderea programelor.
- Exploatarea paralelismului datorită independenței subproblemelor.

Totuși, pentru a beneficia pe deplin de această tehnică, este esențial să se identifice structurile problemelor care permit o descompunere eficientă și să se minimizeze costurile de combinare a soluțiilor. Astfel, Divide et Impera rămâne un principiu central în proiectarea algoritmilor moderni, de la sortări și căutări la operații numerice și procesarea matricelor.

10.3 Probleme rezolvate

În această secțiune vor fi prezentate o serie de probleme rezolvate cu metoda *Divide et Impera*.

1. Se dă un șir de numere întregi, pozitive. Să se găsească dacă acesta este convex sau concav.

   Rezolvare:
   Date de intrare: Se citește un număr întreg n, urmat de un șir de n numere întregi pozitive.
   Date de ieșire: Se afișează:

   - `CONVEX` – dacă șirul este convex;
   - `CONCAV` – dacă șirul este concav;
   - `NICIUNA` – dacă șirul nu respectă niciuna dintre proprietăți.

   Algoritm de rezolvare: Un șir este:

   - **convex** dacă diferențele succesive cresc;

- **concav** dacă diferențele succesive scad.

Se utilizează tehnica **Divide et Impera**:

(a) Șirul este împărțit recursiv în două subșiruri.

(b) Pentru fiecare subșir se verifică local proprietatea de convexitate/concavitate.

(c) La combinare, se verifică și condiția de legătură între cele două subșiruri (diferențele de la graniță).

(d) Rezultatul final este obținut prin combinarea rezultatelor parțiale.

Algoritm scris în C++:

```
#include<iostream>
#include<vector>

struct Rezultat {
    bool convex;
    bool concav;
    int firstDiff;
    int lastDiff;
};

Rezultat verifica(const std::vector<int>& v, int st, int dr)
{
    if (dr - st < 2) {
        return {true, true, 0, 0};
    }

    if (dr - st == 2) {
        int d = v[st + 1] - v[st];
        return {true, true, d, d};
    }

    int mid = (st + dr) / 2;
    Rezultat left = verifica(v, st, mid);
    Rezultat right = verifica(v, mid, dr);

    Rezultat rez;
    rez.firstDiff = left.firstDiff;
    rez.lastDiff = right.lastDiff;

    rez.convex = left.convex && right.convex &&
                 (left.lastDiff <= right.firstDiff);
```

```
    rez.concav = left.concav && right.concav &&
                 (left.lastDiff >= right.firstDiff);

    return rez;
}

int main() {
    int n;
    std::cin >> n;
    std::vector<int> v(n);
    for (int i = 0; i < n; i++)
        std::cin >> v[i];

    Rezultat r = verifica(v, 0, n);

    if (r.convex && !r.concav)
        std::cout << "CONVEX";
    else if (r.concav && !r.convex)
        std::cout << "CONCAV";
    else
        std::cout << "NICIUNA";
    return 0;
}
```

Justificarea soluției propuse: Funcția recursivă verifică proprietățile de convexitate și concavitate pentru fiecare subșir. La fiecare pas de combinare se verifică relația dintre diferențele de la limita subșirurilor. Astfel, dacă șirul global respectă proprietatea, aceasta este menținută pe toate nivelurile de recursivitate.

Analiza complexității:

- *Complexitate temporală*:
  - Relația de recurență: $T(n) = 2T(n/2) + O(1)$;
  - Soluția recurenței: $T(n) = O(n)$.
- *Complexitate spațială*: Spațiu suplimentar: $O(\log n)$, datorat adâncimii recursiei.

2. Se da un vector de numere întregi pozitive de dimensiune dim. Sa se determine simultan minimul și maximul din șir

Rezolvare:

Date de intrare: Se citește un număr întreg `dim`, urmat de un vector de `dim` numere întregi pozitive.

Date de ieșire: Se afișează două valori:

- valoarea minimă din vector;
- valoarea maximă din vector.

Algoritm de rezolvare: Se aplică tehnica **Divide et Impera** pentru a determina simultan minimul și maximul:

(a) Dacă vectorul are un singur element, acesta este atât minimul, cât și maximul.

(b) Vectorul se împarte recursiv în două subvectori de dimensiuni aproximativ egale.

(c) Pentru fiecare subvector se determină recursiv minimul și maximul.

(d) La combinare, minimul global este minimul dintre cele două minime, iar maximul global este maximul dintre cele două maxime.

Algoritm scris în C++:

```
struct MinMax {
    int minim;
    int maxim;
};

MinMax determina(const vector<int>& v, int st, int dr) {
    if (st == dr) {
        return {v[st], v[st]};
    }

    if (dr == st + 1) {
        if (v[st] < v[dr])
            return {v[st], v[dr]};
        else
            return {v[dr], v[st]};
    }

    int mid = (st + dr) / 2;
    MinMax left = determina(v, st, mid);
    MinMax right = determina(v, mid + 1, dr);
```

```
    MinMax rez;
    rez.minim = min(left.minim, right.minim);
    rez.maxim = max(left.maxim, right.maxim);

    return rez;
}

int main() {
    int dim;
    cin >> dim;
    vector<int> v(dim);

    for (int i = 0; i < dim; i++)
        cin >> v[i];

    MinMax rezultat = determina(v, 0, dim - 1);

    cout << rezultat.minim << " " << rezultat.maxim;
    return 0;
}
```

Justificarea soluției propuse: Algoritmul tratează corect cazurile de bază (un singur element sau două elemente). Prin împărțirea vectorului în subvectori și combinarea rezultatelor parțiale, se asigură determinarea corectă a minimului și maximului pentru întregul vector.

Analiza complexității:

- *Complexitate temporală*:
  - Relația de recurență: $T(n) = 2T(n/2) + O(1)$;
  - Complexitatea totală: $O(n)$.
- *Complexitate spațială*: Spațiu suplimentar: $O(\log n)$, determinat de adâncimea recursiei Divide et Impera.

3. Se da un vector de produse identificate prin greutate (număr întreg) și destinație (șir de caractere). Se citește separat o greutate G. Să se sorteze produsele în mod descrescător din punct de vedere al greutății, respectiv crescător după denumire (dacă două produse au aceeași greutate), apoi să se determine eficient produsul cu cea mai apropiata greutate de

G și să se afișeze detaliile acestuia.

Rezolvare:

Date de intrare:

- un număr întreg `dim`;
- `dim` produse, fiecare definit prin greutate și destinație;
- un număr întreg G.

Date de ieșire:

- vectorul de produse sortat conform criteriilor;
- produsul a cărui greutate este cea mai apropiată de G.

Algoritm de rezolvare: Se folosesc două etape bazate pe **Divide et Impera**:

(a) Sortarea produselor utilizând algoritmul Quick Sort, care împarte recursiv vectorul în subsecvențe și folosește un comparator dedicat.

(b) Determinarea produsului cu greutatea cea mai apropiată de G prin căutare binară adaptată, folosind faptul că vectorul este deja sortat după greutate.

Algoritm scris în C++:

```
struct Produs {
    int greutate;
    string destinatie;
};

bool comparatorProdus(const Produs& a, const Produs& b) {
    if (a.greutate != b.greutate)
        return a.greutate > b.greutate; // descrescator
    return a.destinatie < b.destinatie; // crescator
}

int partition(vector<Produs>& v, int low, int high,
              bool (*cmp)(const Produs&, const Produs&)) {
    Produs pivot = v[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
```

```
        if (cmp(v[j], pivot)) {
            i++;
            swap(v[i], v[j]);
        }
    }
    swap(v[i + 1], v[high]);
    return i + 1;
}

void quicksort(vector<Produs>& v, int low, int high,
               bool (*cmp)(const Produs&, const Produs&)) {
    if (low < high) {
        int pi = partition(v, low, high, cmp);
        quicksort(v, low, pi - 1, cmp);
        quicksort(v, pi + 1, high, cmp);
    }
}

int cautaApropiat(const vector<Produs>& v, int G) {
    int st = 0, dr = v.size() - 1;
    int best = 0;

    while (st <= dr) {
        int mid = (st + dr) / 2;
        if (abs(v[mid].greutate - G) <
            abs(v[best].greutate - G))
            best = mid;

        if (v[mid].greutate > G)
            st = mid + 1;
        else
            dr = mid - 1;
    }
    return best;
}

int main() {
    int dim;
    cin >> dim;
    vector<Produs> v(dim);

    for (int i = 0; i < dim; i++)
        cin >> v[i].greutate >> v[i].destinatie;

    int G;
    cin >> G;
```

```
    quicksort(v, 0, v.size() - 1, comparatorProdus);

    int idx = cautaApropiat(v, G);

    for (const auto& p : v)
        cout << p.greutate << " " << p.destinatie << "\n";

    cout << "Produs apropiat de G:\n";
    cout << v[idx].greutate << " " << v[idx].destinatie;

    return 0;
}
```

Justificarea soluției propuse: Comparatorul asigură ordonarea corectă a produselor conform cerinței. Algoritmul Quick Sort împarte vectorul în subprobleme mai mici și le rezolvă recursiv, garantând sortarea completă. Căutarea binară adaptată verifică doar subintervale relevante, identificând produsul cu diferența minimă față de G.

Analiza complexității:

- *Complexitate temporală*:
  - Quick Sort (caz mediu): $O(n \log n)$;
  - Căutarea produsului apropiat: $O(\log n)$;
  - Complexitate totală: $O(n \log n)$.
- *Complexitate spațială*: $O(\log n)$, datorită stivei de apeluri recursive în abordarea Divide et Impera.

10.4 Probleme propuse

*

1. Fie un vector unidimensional de N elemente. Să se determine cu un algoritm *Divide et Impera* (D&I) suma tuturor elementelor sale. O rezolvare posibilă ar fi să se împartă vectorul mereu în jumătăți.
   Exemplu: Pentru $N = 4$ și vectorul [5, 1, 8, 2] suma tuturor elementelor este 16.

2. Fie un vector unidimensional de N elemente. Să se determine cu un algoritm D&I elementul maxim din vector.
   Exemplu: Pentru $N = 4$ și vectorul [5, 1, 8, 2] elementul maxim este 8

3. Fie un vector unidimensional de N elemente și un număr natural X, $X \leq N$. Să se determine cu un algoritm D&I al X-lea cel mai mic element din șir.
   Exemplu: Pentru $N = 6$, $X = 3$ și vectorul [10, 80, 30, 90, 40, 50], rezultatul este 40

**

1. Se dă un vector ale cărui elemente sunt în ordine strict crescătoare până la o anumită poziție, unde se află valoarea maximă, iar după această poziție sunt în ordine strict descrescătoare. Să se determine maximul din vector cu ajutorul unui algoritm de tip D&I.
   Exemplu: Pentru $v = [1, 2, 3, 4, 5, 6, 4, 3, 2]$, elementul maxim este 6.

2. Se dă un șir A de n elemente. Șirul este sortat descrescător și are elemente distincte. Se dorește identificarea următoarei situații: există un index i pentru care $A[i] = i$. Să se elaboreze un algoritm D&I cu complexitate $O(\log n)$.
   Exemplu: Pentru $n = 5$ și șirul descrescător $A = [5, 3, 2, 0, -1]$, rezultatul este $i = 2$ ($A[2] = 2$)

3. Se dă o matrice sortată crescător complet. O matrice sortată crescător complet este o matrice în care elementele sunt în ordine crescătoare atât pe fiecare linie, de la stânga la dreapta, cât și pe fiecare coloană, de sus în jos. Să se determine dacă există elementul k dat în matrice, cu ajutorul unui algoritm de tip D&I.
   Exemplu: Pentru matricea $M = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$ găsim $k = 5$ la poziția $(1, 1)$ - numerotare de la 0

***

1. Fie o matrice de dimensiuni $n \times m$. Să se bordeze matricea recursiv din exterior către interior ca în exemplul următor:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 2 & 2 & 2 & 1 & 0 \\ 0 & 1 & 2 & 3 & 2 & 1 & 0 \\ 0 & 1 & 2 & 2 & 2 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

2. Se consideră un șir de caractere, doar cu valori de 0 și 1. Se știe faptul că șirul conține prima oară toți de 0, apoi toți de 1. Numărați cu ajutorul unui algoritm $D\&I$ câți de 1 sunt în șir.

   Exemplu: Pentru șirul "0001111" rezultatul este 4.

3. Se consideră un șir de caractere, doar cu caractere de 'a' și 'b'. Se știe faptul că șirul conține prima oară toți de 'a', apoi toți de 'b'. Numărați cu ajutorul unui algoritm $D\&I$. câți de 'b' sunt în șir.

   Exemplu: Pentru șirul "aaabbb" rezultatul este 3.

****

1. Considerăm n puncte date prin coordonatele lor, coordonata x și coordonata y. Să se determine distanța dintre cele mai apropiate două puncte. Se consideră distanța Euclidiană.

   Exemplu: Pentru $n = 4$ puncte: $A(1, 2)$, $B(4, 6)$, $C(7, 8)$, $D(9, 1)$, distanța minimă este între $B(4, 6)$ și $C(7, 8)$: $\sqrt{(7-4)^2 + (8-6)^2} = \sqrt{9+4} = \sqrt{13} \approx 3.61$

2. Fie un șir de numere reale de dimensiune n. Se cere să se determine subsecvența de sumă maximă folosind un algoritm D&I.

   Exemplu: Pentru [-2, 1, -3, 4, -1, 2, 1, -5, 4] avem subsecvența [4, -1, 2, 1] cu suma maximă 6

3. O bibliotecă conține mai multe rafturi, pe fiecare raft aflându-se mai multe cărți. Rafturile și cărțile de pe acestea se citesc dintr-un fișier. Se știe că raftul k conține exact k

cărți. Pentru fiecare carte se cunosc anul publicării (număr întreg) și titlul acesteia (șir de caractere).

Să se scrie un program care, folosind tehnica D&I, determină câte rafturi conțin cărțile deja ordonate după:

a) anul publicării cărților;

b) titlul cărților.

Rafturile care nu au deja cărțile ordonate se vor sorta corespunzător criteriului utilizat. Ulterior, să se realizeze căutarea în întreaga bibliotecă a unei cărți cu un an de publicare specific, respectiv a unei cărți cu un titlu specific.

*****

Se consideră 2 polinoame (fiecare polinom dat prin coeficienți și grad).

Să se efectueze înmulțirea celor două polinoame.

Exemplu: Se dau $A(x) = 2x^2 + 3x + 1$ (grad 2) și $B(x) = x - 1$ (grad 1)

Rezultatul este $2x^3 + x^2 - 2x - 1$ (grad 3)

Partea IV

Arbori - Heap-uri

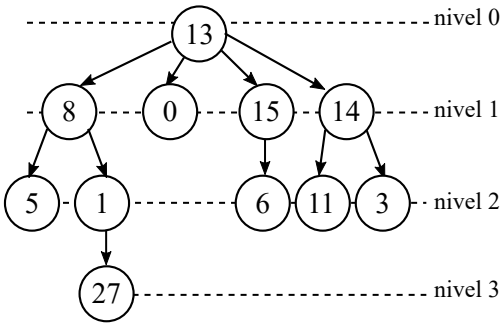
Capitolul 11

Arbori. Arbori binari

11.1 Aspecte teoretice

În informatică arborii reprezintă o structură de date utilizată pentru organizarea ierarhică a informației alcătuită din noduri conectate între ele într-o relație de tip părinte-copil. Nodurile arborelui sunt conectate prin muchii, fără a se forma cicluri, iar fiecare nod poate avea mai mulți copii, dar un singur părinte. Există un singur nod, care nu are părinte, numit rădăcină și care se află cel mai sus în ierarhie. Niciun nod nu împarte vreun fiu cu alt nod. Acele noduri care au cel puțin un fiu se numesc *noduri interne*, iar cele care nu au niciun fiu se numesc *frunze*. Nodurile care au același părinte se numesc *frați*.

În figură este prezentat un exemplu de arbore organizat pe niveluri. Observăm faptul că rădăcina se află pe nivelul 0.



Înălțimea unui nod x, $h(x)$, este lungimea celui mai lung drum de la rădăcină la o frunză, adică numărul de muchii întâlnite pe drumul respectiv. Astfel înălțimea unei frunze este 0. Înălțimea arborelui este egală cu înălțimea rădăcinii, iar în exemplul anterior înălțimea arborelui este $h = 3$.

Adâncimea unui nod este lungimea drumului de la rădăcină la nodul respectiv, deci adâncimea rădăcinii este 0.

Definiții și proprietăți

- Un arbore se numește n-ar, dacă fiecare nod are cel mult n fii. Un arbore binar este un arbore în care fiecare nod are cel mult 2 fii.
- Într-un arbore binar cei doi fii ai unui nod intern se numesc *fiu stâng* și *fiu drept*.
- Numărul maxim de noduri de pe nivelul k al unui al unui arbore n-ar este n^k, deci numărul maxim de noduri de pe nivelul k al unui arbore binar este 2^k (se demonstrează prin inducție după k).
- Un arbore n-ar se numește *complet* dacă toate nivelurile cu excepția eventual a ultimului sunt pline, iar ultimul nivel este completat cu nodurile de la stânga la dreapta.

11.1.1 Reprezentarea în memorie a arborilor

Există diferite moduri de reprezentare a arborilor în memorie. Vom prezenta în continuare două dintre cele mai frecvente moduri, care vor fi utile pentru rezolvarea problemelor propuse.

1. Memorarea arborilor compleți se poate face eficient cu un vector, în care rădăcina se află pe prima poziție din vector (poziția 0), iar cei n fii ai nodului de pe poziția k se află pe pozițiile $n \cdot k + i$, cu $i = \overline{1, n}$.
   În cazul arborilor binari, pentru un nod aflat pe poziția k
   - fiul stâng se află pe poziția $2k + 1$
   - fiul drept se află pe poziția $2k + 2$.
   - părintele se află pe poziția $(k - 1)/2$

   Dacă în vector indicii încep de la 1, se schimbă și aceste relații (vezi [3]).

2. Reprezentarea înlănțuită se realizează printr-o structură de noduri legate între ele prin pointeri.
   - În cazul arborilor n-ari, fiecare nod poate avea câmpurile: key - pentru cheie, $parent$ - pointer către nodul părinte, $children$ - vector de pointeri, care indică fiii nodului.

- În cazul arborilor binari, fiecare nod poate avea câmpurile: key - pentru cheie, $parent$ - pointer către părinte, $left$ - pointer către fiul stâng și $right$ - pointer către fiul drept.

11.1.2 Parcurgerea arborilor binari

Pentru a accesa nodurile unui arbore este necesară parcurgerea într-o anumită ordine, pornind de la rădăcină. Modurile uzuale de parcurgere sunt *parcurgerea în lățime - breadth first* și *parcurgerea în adâncime - depth first*. În cazul parcurgerii în adâncime, în funcție de ordinea în care se vizitează un nod și fiii acestuia avem: *parcurgerea în preordine (RSD)*, *parcurgerea în inordine (SRD)* și *parcurgerea în postordine (SDR)*.

Parcurgerea în lățime a unui arbore binar presupune parcurgerea nodurilor pornind de la rădăcină și parcurgând apoi nodurile de pe fiecare nivel de la stânga la dreapta. Pentru parcurgerea în lățime se poate folosi o coadă de noduri C (pointeri la nod). [8].

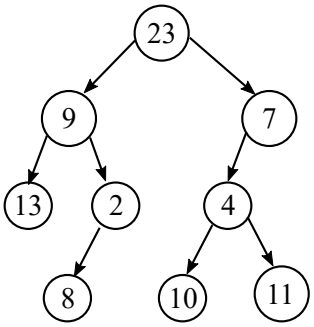
Algorithm 12: Breadth First

```
Input: Arborele binar T
begin
    C ← ∅
    C.push(T.root)
    while C ≠ ∅ do
        curent ← C.top()
        C.pop()
        scrie curent.key
        if curent.left ≠ nil then
            C.push(curent.left)
        end
        if curent.right ≠ nil then
            C.push(curent.right)
        end
    end
end
```

Parcurgerea în adâncime

- **Preordine** (Rădăcină-Stânga-Dreapta - RSD): pentru fiecare nod x se parcurge / prelucrează întâi nodul x, apoi subraborele său stâng și la final subarborele său drept. Modul de parcurgere al subarborilor se face în același mod RSD.
- **Inordine** (Stânga-Rădăcină-Dreapta - SRD): pentru fiecare nod x se parcurge / prelucrează întâi subraborele său stâng, apoi nodul x și la final subarborele său drept. Modul de parcurgere al subarborilor se face în același mod SRD.
- **Postordine** (Stânga-Dreapta-Rădăcină - SDR): pentru fiecare nod x se parcurge / prelucrează întâi subraborele său stâng, apoi subarborele său drept și la final nodul x. Modul de parcurgere al subarborilor se face în același mod SDR.

Exemplu. Se consideră arborele binar:



- **Parcurgerea în lățime**: 23, 9, 7, 13, 2, 4, 8, 10, 11
- **Parcurgerea în adâncime**:
  - **Parcurgerea în preordine (RSD)**: 23, 9, 13, 2, 8, 7, 4, 10, 11
  - **Parcurgerea în inordine (SRD)**: 13, 9, 8, 2, 23, 10, 4, 11, 7
  - **Parcurgerea în postordine (SDR)**: 13, 8, 2, 9, 10, 11, 4, 7, 23

11.2 Probleme rezolvate

În această secțiune vor fi prezentate o serie de probleme teoretice împreună cu modul lor de rezolvare, precum și o serie de probleme aplicative.

11.2.1 Probleme teoretice

1. Desenați structura (fără informație) pentru:

   (a) Toți arborii binari de înălțime 2

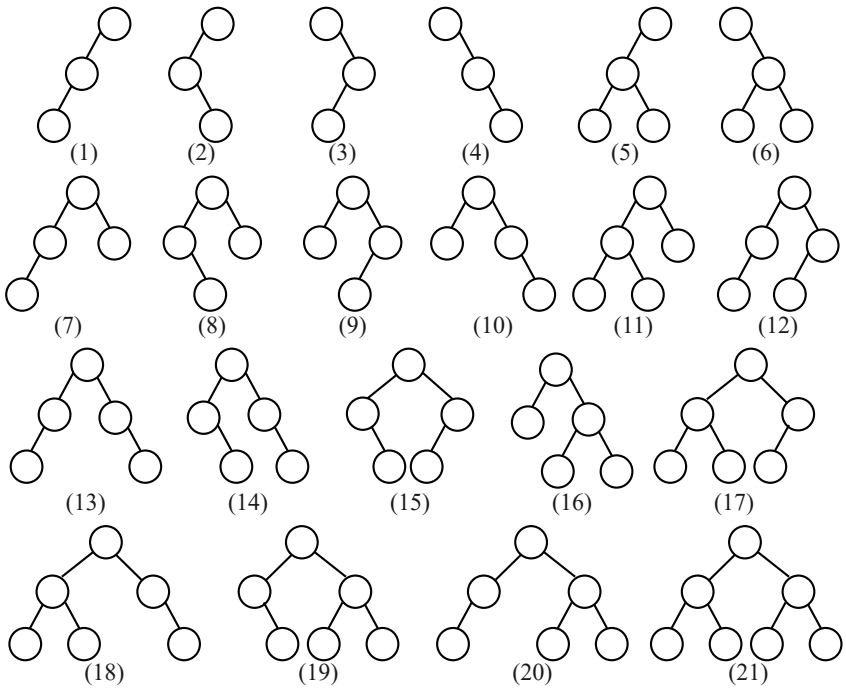
   (b) Un arbore de înălțime 4 care să aibă număr minim / maxim de noduri.

   (c) Un arbore cu 7 noduri de înălțime minimă / maximă.

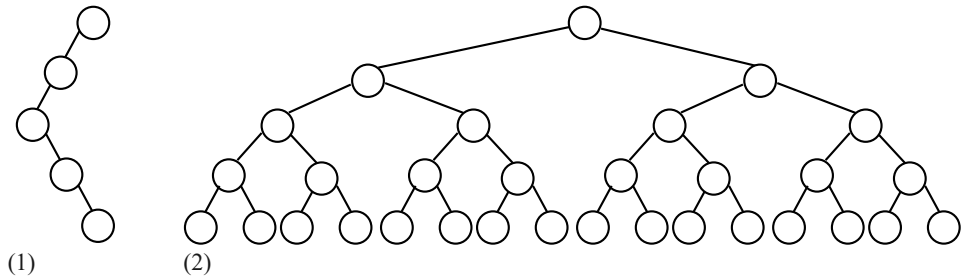
   (d) Toți arborii distincți (ca structură) care au exact 3 noduri.

Rezolvare:

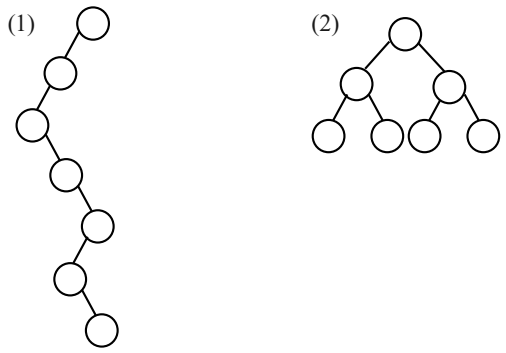
(a) În figurile următoare sunt reprezentate toate configurațiile posibile (fără informație) pentru arbori binari de înălțime 2:



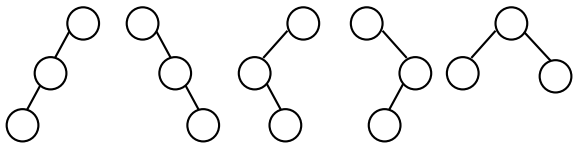
(b) În figura (1) este reprezentat un arbore binar de înălțime $h = 4$ cu număr minim de noduri, $n = 5$, iar în figura (2) un arbore binar de înălțime $h = 4$ cu număr maxim de noduri, $n = 2^{h+1} - 1 = 2^5 - 1 = 31$:



(c) În figura (1) este reprezentat un arbore binar de cu 7 noduri și înălțime maximă $h = n-1 = 6$, iar în figura (2) un arbore binar cu 7 noduri de înălțime minimă $h = \log_2 n = 2$.



(d) În figura următoare sunt reprezentate toate configurațiile de arbore binar cu exact 3 noduri:



2. Desenați arborele binar pentru care:
RSD (preordine): 10, 7, 6, 15, 22, 11, 45, 16, 43, 9, 12
SRD (inordine): 6, 15, 7, 22, 10, 16, 45, 43, 11, 12, 9

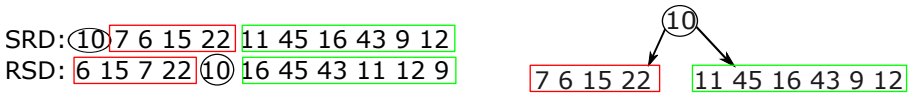
Rezolvare:

Observăm faptul că:

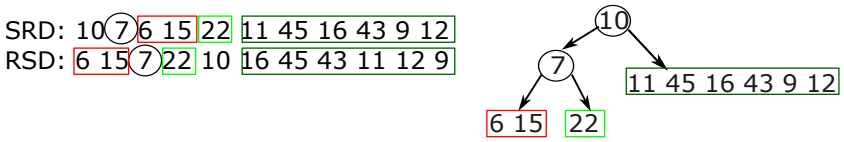
- În parcurgerea în preordine, rădăcina este pe prima poziție, deci 10 este rădăcina.
- În parcurgerea în inordine, rădăcina se află între cheile aflate în subarborele stâng și cele aflate în subarborele drept.

Astfel deducem imediat faptul că arborele căutat are în rădăcină cheia 10. În parcurgerea în inordine, la stânga lui 10 se află toate cheile din subarborele stâng, iar la dreapta cele

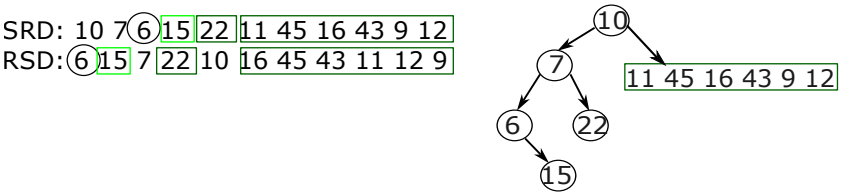
din subarborele drept. În figură, rădăcina a fost marcată printr-un cerc, iar cheile subarborelui stâng sunt marcate de un dreptunghi roșu, iar cele ale subarborelui drept, printr-un dreptunghi verde.



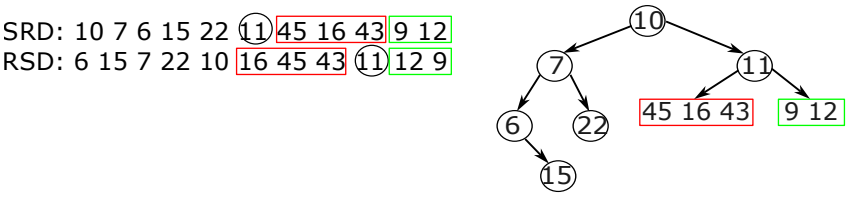
Continuăm în același mod cu subarborele stâng, cel marcat cu roșu, care are în rădăcină cheia 7, cea aflată pe prima poziție în parcurgerea în inordine. Am marcat cu verde închis, subarborele drept al nodului cu cheia 10, care va fi tratat ulterior.

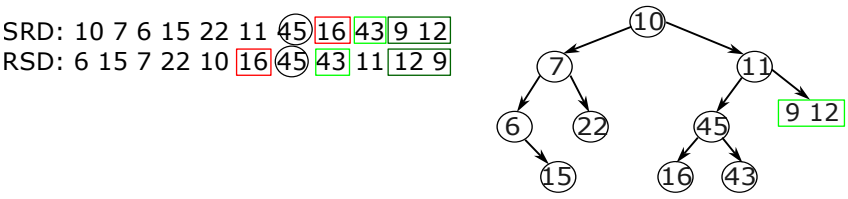


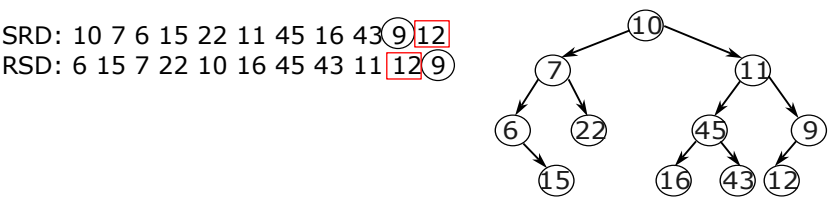
Observăm în figură, că a mai rămas de procesat subarborele stâng al nodului cu cheia 7, care are în rădăcină cheia 6 și obținem:



Acum trecem la procesarea în mod similar subarborele drept al rădăcinii 10.







3. Care este numărul maxim și care este numărul minim de noduri într-un arbore binar de înălțime 5? Justificați.

   Rezolvare:

   Numărul maxim de noduri se obține atunci când fiecare nivel este plin. Deoarece pe nivelul k numărul maxim de noduri este 2^k și deoarece avem nivelul maxim 5, numărul maxim de noduri se obține prin calculul:

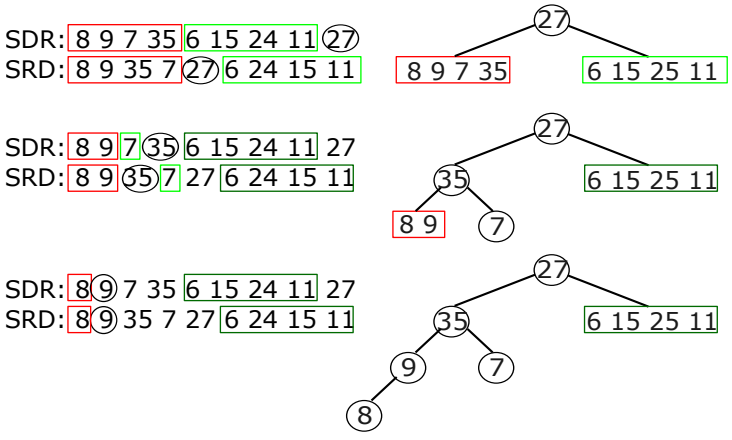
   $$1 + 2 + 2^2 + 2^3 + 2^4 + 2^5 = 2^6 - 1 = 64 - 1 = 63$$

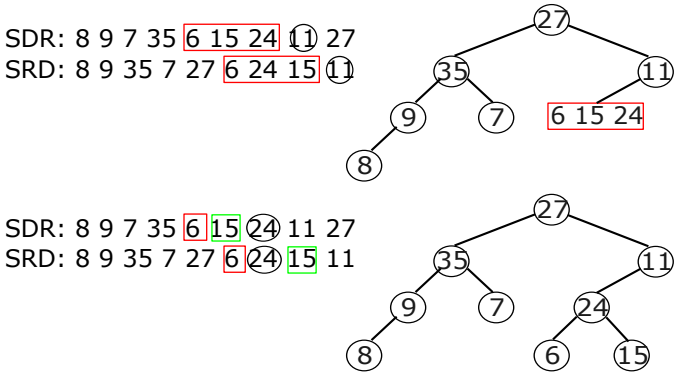
   Numărul minim se obține atunci când pe fiecare nivel avem numărul minim de noduri, adică 1. Deci în cazul nostru numărul minim este 6.

4. Să se refacă un arbore binar dacă se cunosc:
   SDR (postordine): 8, 9, 7, 35, 6, 15, 24, 11, 27
   SRD (inordine): 8, 9, 35, 7, 27, 6, 24, 15, 11

   Rezolvare:

   Atunci când, în loc de parcurgerea în preordine, se cunoaște parcurgerea în postordine (SDR), procedeul este similar ca la exercițiul anterior, doar că trebuie avut în vedere faptul, că rădăcina subarborelui curent se află mereu pe ultima poziție în parcurgerea în postordine a arborelui. Modul de obținere a arborelui binar din cele două parcurgeri este ilustrat în figurile următoare.





5. Scrieți un algoritm în pseudocod, care să realizeze afișarea în ordine de la dreapta la stânga a frunzelor unui arbore binar.

Rezolvare: Propunem mai jos un algoritm recursiv, care se va apela cu rădăcina arborelui transmisă ca argument.

Algorithm 13: Afisaza_frunze

```
Input: Nodul curent nod
begin
    if nod ≠ nil then
        if nod.left = nil AND nod.right = nil then
            print(nod.key)
        end
        Afisaza_frunze(nod.right)
        Afisaza_frunze(nod.left)
    end
end
```

6. Care este înălțimea maximă și care cea minimă a unui arbore binar cu 60 noduri?

Rezolvare:

- Înălțimea maximă se obține atunci când pe fiecare nivel avem număr minim de noduri, adică un nod. În acest caz înălțimea devine $n - 1 = 60 - 1 = 59$.
- Înălțimea minimă se obține atunci când pe fiecare nivel avem număr maxim de noduri și este $h_{min} = \log_2 60 = 5$.

11.2.2 Probleme aplicative

1. Se citește din fișier o expresie aritmetică formată din numere, operatorii de bază (+, -, *, /) și paranteze. Să se construiască un arbore sintactic corespunzător expresiei. Să se afișeze arborele pe niveluri. Se presupune că expresia aritmetică este fără erori.

   Rezolvare:
   Date de intrare: Un $string$ reprezentând expresia aritmetică.
   Date de ieșire: Arborele sintactic corespunzător

   Algoritm de rezolvare:
   Etape:

   - Se construiește forma poloneză postfixată asociată expresiei (vezi [8]).
   - Se consideră o stivă de noduri S.
   - Se parcurge fiecare element al formei poloneze și:
     - dacă este număr, atunci se creează un nou nod, ce are ca informație numărul respectiv și nu are fii. Acest nod se pune în stiva de noduri S.
     - dacă este operator atunci se creează un nou nod, ce are ca informație operatorul, se extrag din stivă două noduri care se vor atașa la dreapta și la stânga nodului nou, iar noul nod va fi pus la loc în stivă.
   - La final în stivă ar trebui să se afle un singur nod, care este rădăcina arborelui.

   Algoritmul în C++
   Includem bibliotecile care ne trebuie:

```
#include<iostream>
#include<fstream>
#include<vector>
#include<stack>
#include<string>
#include<queue>
```

   Pentru forma poloneză postfixată avem nevoie de o funcție ajutătoare, care să determine prioritatea / precedența unui anumit operator. În algoritmul prezentat în [8] s-a definit

pentru '(' precedența 0, pentru '+' și '-' precendența 1, iar pentru '*' și '/' precedența 2.

Observație: în loc de o funcție, am putea folosi un `unordered_map`, care să asocieze fiecărui operator o prioritate.

```
int priority(char c)
{
    switch (c){
    case '(':
        return 0;
    case '+':
    case '-':
        return 1;
    case '/':
    case '*':
        return 2;
    default:
        return -1;
    }
}
```

Folosind algoritmul din [8] construim forma poloneză postfixată prin funcția următoare. Facem însă câteva observații:

- deoarece putem avea numere cu mai multe cifre, vom reține în forma poloneză elemente de tip `std::string`;
- când întâlnim în șirul de caractere o cifră, vom parcurge șirul, atâta timp cât urmează cifre și le vom cumula pe toate într-un `std::string`;
- caracterele de tip *blank* vor fi ignorate;
- nu am tratat în cod posibile erori în alcătuirea expresiei artimetice. Acest lucru rămâne ca exercițiu pentru studenți.

```
std::vector<std::string>forma_poloneza(std::string sir){
    std::vector<std::string>fp;
    std::stack<char> stiva;

    for (int index = 0; index < sir.length(); ++index){
        //colectarea tuturor cifrelor unui numar intr-un
           string
        std::string numar = "";
        while (index < sir.length() &&
           std::isdigit(sir[index])){
            numar += sir[index++];
```

```
        }
        if(numar.length()>0) fp.push_back(numar);
       //daca nu e vorba de o cifra / un numar
        switch (sir[index]) {
            case '(':
                stiva.push('(');
                break;
            case ')':
                while (!stiva.empty() && stiva.top() != '('){
                    std::string op(1, stiva.top());
                    stiva.pop();
                    fp.push_back(op);
                }
                if(!stiva.empty()) stiva.pop();
                break;
            case '+':
            case'-':
            case '/':
            case '*':
                while (!stiva.empty() &&
                   priority(stiva.top())>=
                        priority(sir[index])){
                    std::string op(1, stiva.top());
                    fp.push_back(op);
                    stiva.pop();
                }
                stiva.push(sir[index]);
                break;
            //ignorarea blank-urilor
            case ' ':
            case '\t':
                continue;
                break;
            }
        }
    }
    //golirea stivei
    while (!stiva.empty()){
        std::string op(1, stiva.top());
        fp.push_back(op);
        stiva.pop();
    }
    return fp;
}
```

Pornind de la forma poloneză, parcurgem fiecare element din aceasta și creăm nodurile, pe care le adăugăm în stiva de noduri. Definim întâi structura *node*:

```
struct node {
    std::string key;
    node* left, * right;
    node(std::string val = ""){
        key = val;
        left = right = nullptr;
    }
};
```

Construim arborele sintactic pe baza formei poloneze:

```
node* arboreSintactic(std::vector<std::string> fp)
{
    std::stack<node*> arbore;
    for (auto el:fp){
        //noul nod care va contine elementul curent din forma
           poloneza
        node* newNode = new node(el);
        //verific daca este un numar
        if (isdigit(el[0]))
              arbore.push(newNode);

        /*altfel este operator si trebuie sa extrag din stiva
           primele 2 noduri si sa le leg de noul nod */
        else {
            node* right, * left;
            //stiva trebuie sa contina cel putin 2 elemente
            // (nu avem operatori unari)
            if (arbore.size()>1)
            {
                right = arbore.top();
                arbore.pop();
                left = arbore.top();
                arbore.pop();
            }
            //altfel ceva este gresit in expresia artimetica
            else
                return nullptr;
            newNode->left = left;
            newNode->right = right;

            arbore.push(newNode);
        }
    }
```

```
    //daca in stiva sunt mai mult de 1 element este ceva
       gresit
    if (arbore.size() == 1)
        return arbore.top();
    else return nullptr;
}
```

Ultimul pas este de a parcurge pe niveluri arborele. Pentru aceasta folosim funcția *niveluri* de tip BF (*breadth-first*) construită astfel, încât să afișeze elementele de pe fiecare nivel pe un rând nou.

```
void niveluri(node* root)
{
    if(!root) return;
    //folosim o coada care tine minte perechi (nivel, nod)
    std::queue<std::pair<int,node*>> coada;
    coada.push({0, root});
    int nivelCurent = 0;
    while (!coada.empty())
    {
        node* current = coada.front().second;
        int nivel = coada.front().first;
        coada.pop();
        //aici se verifica trecerea la un nivel nou
        if (nivel != nivelCurent)
        {
            std::cout << "\n";
            nivelCurent = nivel;
        }
        std::cout << current->key << " ";
        if (current->left) coada.push({ nivel + 1,
           current->left });
        if (current->right) coada.push({ nivel + 1,
           current->right });
    }
}
```

Funcția `main` leagă acum toate elementele de mai sus pentru a obține arborele sintactic:

```
int main() {
    //se incearca deschiderea si citirea sirului din fisier
    std::ifstream file("sir.txt");
    if (!file.is_open()) {
        std::cerr << "Eroare la deschiderea fisierului.\n";
        return 1;
    }
    //se presupune ca avem un singur sir pe prima linie
```

```
    std::string sir="";
    if (std::getline(file, sir)) {      // citeste o linie
        std::cout << "Linia: " << sir << '\n';
    }
    else std::cout << "Fisierul este vid sau eroare de
       citire.\n";
    file.close();

    //daca s-a putut citi sirul
    if (sir.length() > 0){
        //constuim forma poloneza postfixata
        std::vector<std::string>fp = forma_poloneza(sir);
        //construim arborele sintactic
        node* root = arboreSintactic(fp);
        niveluri(root);
        std::cout << "\n";
    }
    return 0;
}
```

Justificarea soluției propuse: Construcția formei poloneze postfixate are la bază un algoritm din [8], care a fost explicat acolo pe larg. În forma poloneză postfixată, operanzii apar în aceeași ordine ca în expresia originală, în schimb operatorii apar în ordinea în care trebuie efectuați.

În arborele sintactic, operatorii care trebuie efectuați mai întâi sunt pe nivelurile de jos ale arborelui, iar operatorul aflat în rădăcină este ultimul efectuat. Modul în care se construiește arborele, folosind stiva de noduri, are ca efect exact acest lucru. Primele noduri care se formează sunt cu primii operatori din forma poloneză și se vor afla în subarborii corespunzători operatorilor care apar mai târziu. Ultimul operator din forma poloneză va fi cel plasat în ultimul nod construit, care este cel din rădăcină.

Analiza complexității.

- *Complexitatea de timp*. Construirea formei poloneze se realizează în complexitate $O(m)$, unde m este numărul de elemente din șirul de caractere citit, deoarece fiecare caracter este procesat o singură dată.

  Construirea arborelui sintactic are complexitatea $O(n)$, unde n este numărul de elemente din forma poloneză postfixată (operanzi și operatori). Pentru fiecare element

se creează un nod care de plasează în stivă (deci operații de complexitate constantă). În plus, pentru fiecare nod de tip operator se extrag două noduri din stivă, care se leagă de acesta înainte să fie introdus în stivă (tot operații în complexitate constantă). În mod evident $n \leq m$
Parcurgerea pe niveluri a arborelui este tot în complexitate $O(n)$.
Rezultă o complexitate de $O(m + 2n) = O(m)$.

- *Complexitatea de spațiu* este destul de evident $O(m)$.

2. Se citesc dintr-un fișier n caractere împreună cu codul lor asociat și anume: fiecare caracter va avea asociat un cod formar din cifre binare. Codurile vor avea lungime variabilă și niciun cod nu este prefixul altuia. Să se construiască un arbore binar asociat în modul următor. Fiecare caracter va fi inserat într-o frunză a arborelui coborând în arbore de la rădăcină, pentru un caracter 0 pe stânga și pentru un caracter 1 pe dreapta nodului curent.

Rezolvare:
Date de intrare: Un fișier care conține caractere împreună cu codurile binare asociate (cu condiția că niciun cod nu este prefixul altui cod).
Date de ieșire: Arborele de codificare corespunzător.

Algoritm de rezolvare:
Etape:

- Se citesc perechile $(cod, caracter)$ și se stochează într-un vector de astfel de perechi.
- Se creează rădăcina arborelui, care are drept informație caracterul '-' (sau ce alt caracter dorim, nu este relevant).
- Pentru fiecare pereche $(cod, caracter)$ se pornește în arbore cu un nod $curent$ de la rădăcină. Se parcurge codul element cu element și pentru fiecare element $cod[i]$:
  - Dacă $cod[i] =' 0'$, trebuie coborât pe stânga nodului $curent$. Dacă nu există nod în stânga, acesta trebuie întâi creat.
  - Dacă $cod[i] =' 1'$, trebuie coborât pe dreapta nodului $curent$. Dacă nu există nod în dreapta, acesta trebuie întâi creat.

– Dacă am ajuns la ultimul element din *cod* atunci în nodul *curent* se va plasa caracterul.

Algoritmul în C++:

```
#include<iostream>
#include<fstream>

struct code_tree {
    struct node {
       char key;
       node* left, * right;
       node(char val = '-'){
          key = val;
        left = right = nullptr;
       }
    };
    node* root;

    void construct(std::vector<std::pair<std::string, char>>
       coduri){
        root = new node();
        for (auto el : coduri){
            std::string cod = el.first;
            char caracter = el.second;
            //pornim din radacina cu nodul curent
            node* curent = root;
            for (int i = 0; i < cod.length(); i++){
                 if (cod[i] == '0') {
                    //verific daca am copil stang si daca nu
                       il creez
                    if (!curent->left)
                        curent->left = new node();
                    curent = curent->left;
                 }
                 else {
                    //verific daca am copil drept si daca nu
                       il creez
                     if (!curent->right)
                         curent->right = new node();
                     curent = curent->right;
                 }
        }
        //plasez caracterul curent in frunza
        curent->key = caracter;
       }
    }
```

```
    //aceasta functie permite parcurgerea in preordine
    //si constuirea codului unui caracter pe masura ce se
       coboara in
    //arbore spre caracterul respectiv
    void RSD(node* curent, std::string cod) {
       if (curent){
        //daca sunt in frunza afisez caracterul si codul
        if (!curent->left && !curent->right){
            std::cout << curent->key << ": " << cod << "\n";
        }
        //daca cobor pe stanga adaug la cod un 0
        if (curent->left)
            RSD(curent->left, cod + '0');
        //daca cobor pe dreapta adaug la cod un 1
        if (curent->right)
            RSD(curent->right, cod + '1');
       }
    }
};
int main() {
    std::ifstream file("coduri.txt");
    std::string cod, curent="";
    char caracter;
    std::vector<std::pair<std::string, char>> coduri;
    while (file >> cod >> caracter)
        coduri.emplace_back(cod, caracter);

    code_tree T;

    T.construct(coduri);
    T.RSD(T.root, curent);

    return 0;
}
```

Justificarea soluției propuse: Arborele se construiește prin coborârea exact pe drumul marcat cu codul corespunzător caracterului curent, având grijă la fiecare pas să creeze nodurile încă inexistente.

Analiza complexității.

- *Complexitatea de timp*. Construirea arborelui se realizează prin coborârea în arbore pentru fiecare caracter pe un drum de lungimea codului corespunzător acelui carac-

ter. Complexitatea poate fi deci exprimată prin:

$$O\left(\sum_{k=0}^{n-1} l_k\right)$$

unde n este numărul de caractere iar l_k, $k \in \overline{0, n-1}$ este lungimea codului corespunzător celui de al k-lea caracter.

- *Complexitatea de spațiu* este evident $O(n)$.

3. **Operații pe un arbore binar**. Se citește dintr-un fișier un arbore binar. Modul de citire este la alegere. Variante propuse:
   - Se citesc 2 parcurgeri din care se construiește arborele.
   - Se citesc: 3 vectori - unul cu cheile arborelui, unul cu copii stângi pentru fiecare nod, unul cu copii drepți pentru fiecare nod.
   - Citire pe niveluri (în lățime).

   Să se implementeze următoarele funcții pentru arbore:

   (a) O funcție care calculează înălțimea unui subarbore.

   (b) O funcție care verifică dacă doi arbori sunt identici.

   (c) O funcție care verifică dacă un arbore e complet.

   Rezolvare:
   Vom folosi citirea folosind două parcurgeri: preordine și inordine.

   Date de intrare: Un fișier care conține valorile cheilor obținute prin parcurgerea în preordine pe prima linie și pe cele obținute prin parcurgerea în inordine pe a doua linie. În plus:

   (a) Rădăcina (pointer la nod) subarborelui pentru care se determină înălțimea.

   (b) Cei doi arbori, pentru care se verifică dacă sunt identici.

   (c) Arborele pentru care se verifică dacă este complet.

Date de ieșire:

(a) Înălțimea subarborelui (de tip int).

(b) $true$ sau $false$.

(c) $true$ sau $false$.

Algoritm de rezolvare:

Etape:

- Citim din fișier cele două parcurgeri.
- Construim arborele folosind algoritmul ilustrat în problemele teoretice. Acesta poate fi implementat în manieră *Divide et Impera*.
- Pentru punctul (a) folosim o funcție recursivă, care calculează înălțimea subarborelui curent pe baza înălțimii subarborilor din stânga și din dreapta:

$$h(x) = 1 + max(h(x.left), h(x.right))$$

- Pentru punctul (b) folosim oricare dintre parcurgeri, de exemplu RSD recursiv, și la fiecare moment verificăm dacă nodurile curente în ambii arbori au aceeași cheie.
- Pentru punctul (c) se consideră parcurgerea pe niveluri folosind o coadă. În momentul când se întâlnește primul nod care este frunză, toate celelalte noduri vor fi la rândul lor frunze. Altfel arborele nu este complet. De asemenea, un nod nu poate avea un fiu drept, dar să nu aibă fiu stâng.

Algoritmul în C++

Începem prin definirea structurii arbore, care va conține toate funcțiile cerute:

```
#include<iostream>
#include<fstream>
#include<vector>
#include<queue>

struct tree {
    struct node {
        int key;
        node* left, * right, * parent;
        node(int val = 0){
```

```
            key = val;
            left = right = parent = nullptr;
        }
    };

    node* root=nullptr;
    void makeTree(std::vector<int>rsd, std::vector<int>srd);
    node* construct(const std::vector<int>&rsd, const
       std::vector<int>&srd, int poz_rsd, int stanga_sdr, int
       dreapta_sdr, node*& current);
    void niveluri();
    void RSD(node* current);
    int height(node* current);
    bool isEqual(tree T);
    bool isComplete();
};
```

Definim acum funcția recursivă care construiește arborele pe baza celor două parcurgeri. Parametrii reprezintă: cei doi vectori care conțin parcurgerile, poziția poz_rsd în vectorul rsd la care s-a ajuns, adică indicele cheii care va reprezenta rădăcina subarborelui curent și capetele intervalului reprezentând parcurgerea în inordine (SRD) a subarborelui curent ($stanga_srd$, $dreapta_srd$).

```
tree::node*tree:: construct(const std::vector<int>&rsd, const
   std::vector<int>&srd, int poz_rsd, int stanga_sdr, int
   dreapta_sdr, node*& current)
{
    //verific daca subarborele curent nu e vid
    if (stanga_sdr <= dreapta_sdr)
    {
        //creez noul nod, radacina subarborelui curent pe baza
        //informatiei din parcurgerea rsd
        node* sub_root = new node(rsd[poz_rsd]);
        //identific pozitia aceastei radacini in parcurgerea
           srd
        int index = stanga_sdr;
        for (; index <= dreapta_sdr && srd[index] !=
           rsd[poz_rsd]; index++);
        //determin numarul de elemente din subarborele stang
           curent
        int len = index - stanga_sdr;

        //construiesc subarborii stang si drept curenti in
           mod recursiv
        //folosind impartirea parcurgerii srd, data de pozitia
```

```
        //radacinii curente si a capetelor intervalului
           primite
        //ca parametri
        sub_root->left = construct(rsd, srd, poz_rsd + 1,
           stanga_sdr, index - 1, sub_root);
        sub_root->right = construct(rsd, srd, poz_rsd + len +
           1, index + 1, dreapta_sdr, sub_root);
        return sub_root;
    }
    return nullptr;
}

//functia makeTree apeleaza prima data functia recursiva
   construct
void tree::makeTree(std::vector<int>rsd, std::vector<int>srd)
   {
    root = construct(rsd, srd, 0, 0, rsd.size() - 1, root);
}
```

(a) Funcția care determină înălțimea unui subarbore:

```
int tree::height(tree::node * current)
{
    if (!current) return 0;
    int hs = height(current->left);
    int hd = height(current->right);
    return hs < hd ? 1 + hd : 1 + hs;
}
```

(b) Pentru a verifica dacă doi arbori binari sunt identici vom folosi două funcții cu același nume $isEqual$, dar cu listă diferită de parametri. Prima funcție va primi ca parametru arborele T cu care comparăm arborele curent și va apela a doua funcție, care este recursivă și are ca parametri 2 noduri curente, unul din parcurgerea în preordine a primului arbore, iar cel de-al doilea din parcurgerea în preordine a celui de-al doilea arbore.

```
bool tree::isEqual(tree T) {
    return isEqual(root, T.root);
}

//parcurge ambii arbori in preordine si verifica daca nodurile
//curente au chei diferite sau cel putin unul este null
bool tree::isEqual(tree::node* n1, tree::node* n2)
{
    if (n1 == nullptr && n2 == nullptr) return true;
    if ((n1 && !n2) || (!n1 && n2))
        return false;
```

```
    if (n1->key != n2->key)
        return false;
    return isEqual(n1->left, n2->left) && isEqual(n1->right,
       n2->right);
}
```

(c) Funcția care verifică dacă un arbore binar este complet este de fapt o parcurgere în lățime, în care verificăm ca, după ce scoatem din coadă prima frunză, să nu mai urmeze decât frunze.

```
bool tree::isComplete()
{
    std::queue<node*> myQueue;
    myQueue.push(root);
    //pentru a marca, daca s-a intalnit prima frunza
    bool leaf = false;
    while (!myQueue.empty())
    {
        node* current = myQueue.front();
        myQueue.pop();
        if (current->left)
        {
            if (leaf)
                return false;
            myQueue.push(current->left);
            if (current->right)
                myQueue.push(current->right);
        }
        else //nu am voie sa am fiu drept daca nu am fiu stang
            if (current->right) return false;
            else leaf = true;

    }
}
```

Justificarea soluției propuse: Arborele se construiește prin folosirea algoritmului explicat în cadrul problemelor teoretice rezolvate.

(a) Funcția de determinare a înălțimii se bazează pe formula $h(x) = 1+max(h(x.left), h(x.right))$, care este evidentă.

(b) Funcția care verifică dacă doi arbori sunt egali se bazează pe parcurgerea simultană în preordine a acestora. Apelul funcției se realizează recursiv, coborârea în arbore făcându-se până ce se ajunge la $nullptr$, acest lucru asigură și faptul că nu doar parcurgerile celor doi arbori sunt egale (ceea ce nu ar fi suficient), ci și că aranjarea cheilor în arbore este

aceeași.

(c) Faptul că arborele se parcurge pe niveluri, asigură faptul că, în cazul unui arbore complet și numai atunci, când pe nivelul curent s-a ajuns la o frunză, urmează numai frunze. În plus, datorită faptului că nu se permite ca un nod să aibă fiu drept, dar nu fiu stâng, asigură faptul că pe ultimul nivel frunzele sunt așezate de la stânga la dreapta pe acel nivel, deci respectă proprietățile de arbore complet.

Analiza complexității.

- *Complexitatea de timp*. Construirea arborelui se realizează printr-un algoritm de tip *Divide et Impera* în care la fiecare pas se caută în parcurgerea SRD indicele k al cheii de pe prima poziție din bucățica de parcurgere RSD curentă, iar apoi se apelează aceeași funcție pe bucățile aflate la stânga și la dreapta acestui indice. Complexitatea de timp depinde de numărul de comparații care se efectuează la pasul de căutare al indicelui k, precum și de numărul de divizări.

  Cel mai defavorabil caz se obține când parcurgerea în inordine este reversul parcurgerii în preordine. Atunci arborele arată ca o listă, iar complexitatea este $O(n^2)$, unde n este numărul de chei.

  Cel mai favorabil caz presupune că vectorul SRD este împărțit de fiecare dată exact la jumătate și indicele rădăcinii curente este la jumătatea bucății de vector considerată și se demonstrează în mod similar cu algoritmul Quick-Sort, că în acest caz complexitatea este $O(n \log_2(n))$

  (a) Complexitatea determinării recursive a înălțimii este $O(n)$, unde n este numărul de noduri din subarbore, deoarece fiecare nod este parcurs o dată.

  (b) Complexitatea verificării egalității celor doi arbori este în cel mai favorabil caz $O(1)$, dacă din start rădăcinile sunt diferite și în cel mai defavorabil caz, atunci când arborii sunt egali, este $O(n)$, pentru că se parcurg ambii arbori în întregime, n fiind în acest caz numărul de chei din fiecare arbore.

  Observație: dacă în structura $tree$ păstrăm și o variabilă care reține numărul de noduri din arbore, atunci se poate face o primă verificare a egalității numărului de

noduri al arborilor, ceea ce poate reduce semnificativ timpul de execuție în multe situații particulare, când arborii sunt parțial identici, dar unul are mai multe noduri decât celălalt.

(c) Complexitatea verificării completitudinii este dată ce complexitatea algoritmului BFS (*breadth first search*) care este liniar în numărul de noduri n, adică $O(n)$.

- *Complexitatea de spațiu*. Datorită faptului că am transmis prin referință vectorii ce conțin parcurgerile, nu se vor efectua copii ale acestora la apelurile recursive. Cum în rest este vorba de variabile simple, zona de memorie ocupată pentru construcția arborelui este de ordinul $O(n)$, unde n este numărul de chei.

  (a) Pentru calculul înălțimii unui subarbore, apelul recursiv păstrează pe stivă instanțele precedente al variabilelor pe drumul de coborâre pe ramura curentă. Astfel complexitatea este $O(h)$, unde h este înălțimea subarborelui.

  Cel mai defavorabil caz: atunci când $h = n - 1$, n este numărul de chei din subarbore, complexitatea devine $O(n)$.

  Cel mai favorabil caz: atunci când subarborele este echilibrat și $h = \log_2 n$, complexitatea devine $O(\log_2 n)$.

  (b) Pentru complexitatea de spațiu în cazul verificării egalității dintre arbori, raționamentul este similar ca în cazul determinării înălțimii, doar că, *cel mai favorabil caz* apare atunci când deja rădăcinile celor 2 arbori sunt diferite și atunci complexitatea de spațiu este $O(1)$. În *cel mai defavorabil caz*, apare atunci când arborii sunt identici și dezechilibrați și este $O(n)$, n este numărul de chei în fiecare arbore.

  (c) Complexitatea de spațiu în cazul verificării completitudinii este dată de numărul de noduri păstrate la fiecare moment în coada folosită pentru parcurgerea pe niveluri. Acestea pot fi în cel mai defavorabil caz egal cu numărul maxim de noduri de pe ultimul nivel. În acest caz, când am număr maxim de chei pe ultimul nivel, arborele este echilibrat, iar numărul total de noduri de pe ultimul nivel este $n/2$. Deci complexitatea de spațiu este $O(n)$.

11.3 Probleme propuse

11.3.1 Probleme teoretice

1. Desenați toți arborii binari cu 5 noduri dintre care 3 frunze.

2. Desenați arborele binar pentru care parcurgerile în preordine (RSD) și în inordine (SRD) sunt:
   RSD: 25, 14, 11, 7, 9, 13, 20, 21, 43, 32, 30, 45
   SRD: 7, 9, 11, 13, 14, 20, 21, 25, 30, 32, 43, 45

3. Care este înălțimea maximă și care cea minimă a unui arbore binar cu 25 noduri?

4. Se consideră un arbore binar. Care dintre următoarele afirmații sunt adevărate?
   (a) Arborele se poate reface cunoscând parcurgerea în preordine.
   (b) Un nod x are cheia întotdeauna mai mare decât cheile din subarborele său stâng.
   (c) Dacă înălțimea este 7 atunci numărul de frunze este mai mic sau egal cu 128.
   (d) Dacă înălțimea este 7, atunci numărul maxim de noduri din subarborele stâng al rădăcinii este 64.
   (e) Dacă numărul de noduri stocate este 60 atunci înălțimea minimă este 5.

11.3.2 Probleme aplicative

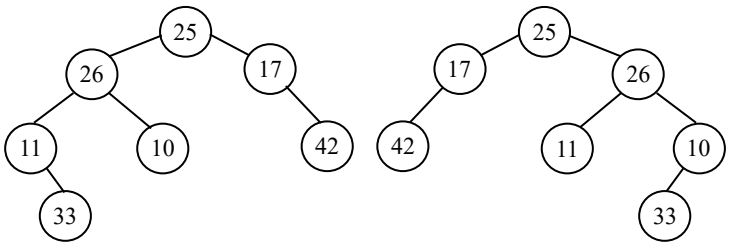
*

1. Scrieți o funcție care să realizeze o copie a unui arbore binar. Folosiți structura dată în secțiunea de probleme rezolvate.

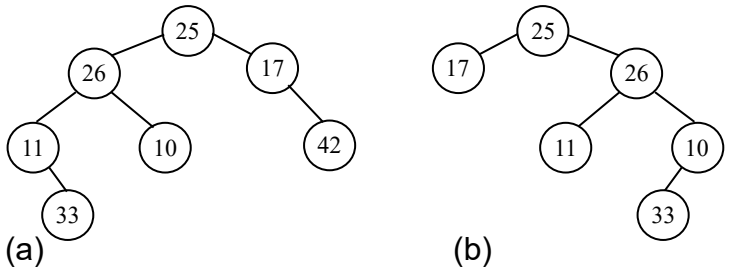
2. Scrieți o funcție care determină adâncimea unui nod. Folosiți structura dată în secțiunea de probleme rezolvate.

3. Scrieți o funcție care să realizeze o copie a unui arbore binar, dar în oglindă (tot ce e pe stânga se mută pe dreapta și invers). Folosiți structura dată în secțiunea de probleme rezolvate.

Exemplu: În figura următoare este prezentat un exemplu de arbore binar împreună cu oglinditul său.

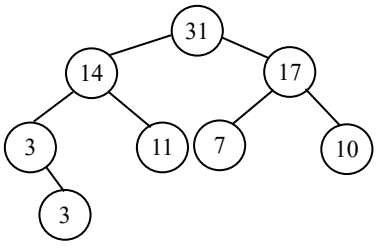


4. Scrieți o funcție care verifică dacă un arbore binar este balansat. Vom considera arborele balansat, dacă diferența absolută dintre înălțimea subarborelui stâng și cea a subarborelui drept este cel mult 1 pentru fiecare nod din arbore.
   Exemplu: În figura următoare (a) este arbore echilibrat conform definiției considerate, iar (b) nu este, deoarece subarborele drept al rădăcinii are înălțimea cu 2 mai mare decât subarborele stâng.



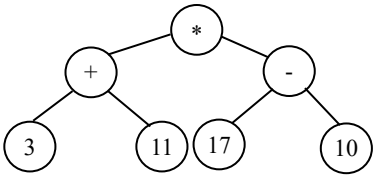
**

1. Un *arbore sumă* este un arbore în care pentru fiecare nod x, cheia sa este suma cheilor fiilor săi. Scrieți o funcție care verifică dacă un arbore binar este un astfel de arbore sumă.
   Exemplu: În figura următoare este prezentat un arbore sumă.



2. Pornind de la problema rezolvată 1, din secțiunea 11.2.2, scrieți o funcție, care evaluează expresia aritmetică stocată în arborele sintactic.

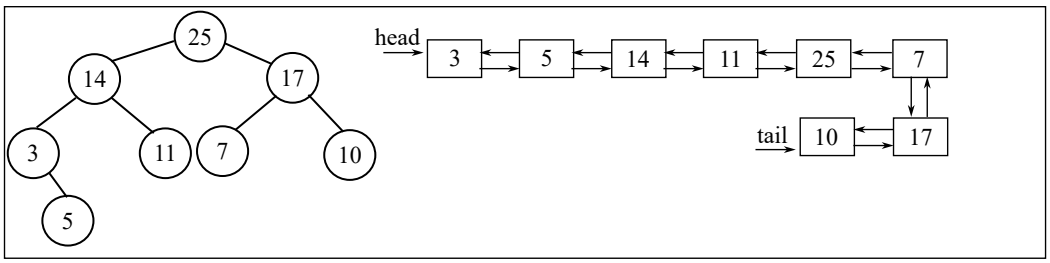
   Exemplu: Pentru arborele sintactic din figura următoare funcția va returna valoarea $(3 + 11) * (17 - 10) = 14 * 7 = 98$.



3. Scrieți o funcție iterativă (nerecursivă) în C++, care să realizeze parcurgerea în preordine a unui arbore binar.

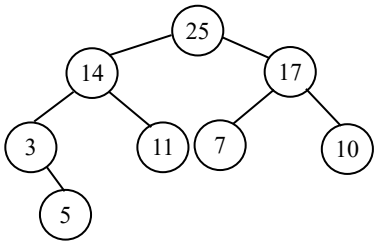
4. Scrieți o funcție care preia ca parametru un arbore binar și produce o listă dublu înlănțuită, care are elementele din arbore în parcurgerea în inordine.

   Exemplu: În figură este prezentat un arbore binar împreună cu lista dublu înlănțuită, care conține cheile arborelui în parcurgerea în inordine.



5. Scrieți o funcție care pentru fiecare frunză a unui arbore binar afișează suma cheilor din nodurile de pe drumul de la rădăcină la acea frunză (inclusiv rădăcina).

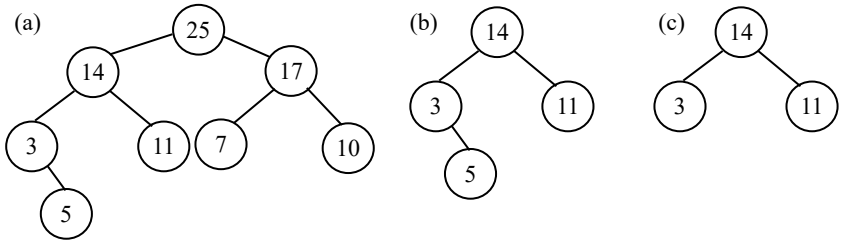
   Exemplu: Pentru arborele din figură se va afișa:



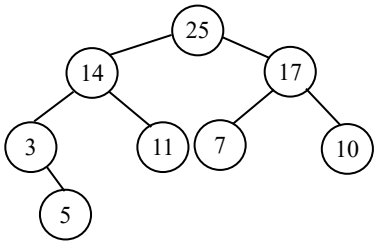
   - pentru frunza 5: 25 + 14 + 3 + 5 = 47

- pentru frunza 11: 25 + 14 + 11 = 50
- pentru frunza 7: 25 + 17 + 7 = 49
- pentru frunza 10: 15 + 17 + 10 = 42

6. Scrieți o funcție care are ca parametri 2 arbori binari T_1 și T_2 și care verifică dacă T_2 poate fi considerat subarbore al lui T_1 (adică există un subarbore al lui T_1 identic cu T_2).
   Exemplu: În figură, (b) este subarbore al lui (a), dar (c) nu este subarbore al lui (a).

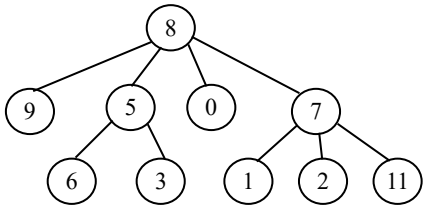


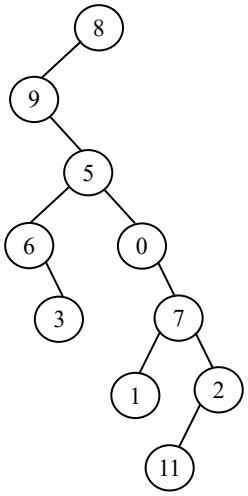
7. Scrieți o funcție care afișează toți *verii* unui anumit nod x, dintr-un arbore binar, unde prin veri înțelegem fiii fratelui părintelui lui x.
   Exemplu: În figură verii nodului 3 sunt 7 și 10.



***

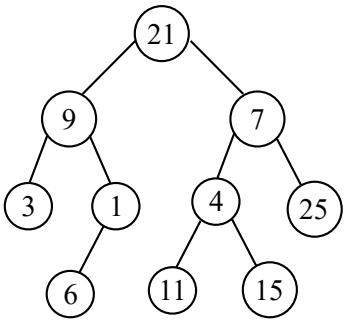
1. Scrieți o funcție care citește din fișier un arbore oarecare (stabiliți modul de organizare al fișierului), iar apoi scrieți o funcție care transformă acest arbore într-unul binar, care este o reprezentare fiu-frate al arborelui original.
   Exemplu: În figură, în dreapta este reprezentarea fiu-frate a arborelui din stânga. În această reprezentare fiul stâng al unui unui nod x este cel mai din stânga fiu al lui x din reprezentarea inițială, iar fiul din dreapta este primul frate din dreapta al lui x din reprezentarea inițială.





2. Scrieți o funcție care preia ca parametri două noduri dintr-un arbore binar și determină cel mai apropiat strămoș comun.

   Exemplu: În arborele din figură, cel mai apropiat strămoș comun al nodurilor 25 și 11 este 7.



3. **Arbore genealogic.** Se consideră arborele genealogic al unei familii (în fiecare nod se păstrează un singur părinte). Fiecare nod are drept fii noduri ce conțin numele copiilor părintelui respectiv. Construiți o structură corespunzătoare, care să conțină:

   - o funcție de citire a arborelui din fișier;
   - o funcție de afișare pe niveluri, astfel încât la afișare nivelurile să fie separate prin 2 rânduri libere și pentru fiecare persoană să se afișeze numele său, numele părintelui și numărul de copii;
   - o funcție, care are ca parametru un nume și afișează numele tatălui și numele copiilor persoanei respective, dacă aceasta se găsește în arbore, altfel semnalează faptul, că nu a fost găsită persoana;

- o funcție care afișează numărul de generații reprezentate în arbore;
- o funcție care afișează persoanele care nu au niciun copil.

Capitolul 12

Heap-uri binare. Cozi de prioritate

12.1 Aspecte teoretice

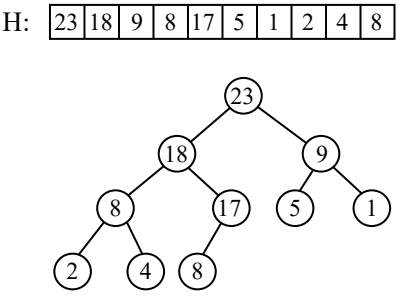
Heap-urile binare sunt arbori binari compleți (vezi definiție de la capitolul de arbori) cu o anumită ordonare a cheilor, dată de tipul de heap și anume:

- **Heap-max**: pentru orice nod x, cheia sa este mai mare sau egală cu orice cheie din subarborele de rădăcină x.
- **Heap-min**: pentru orice nod x, cheia sa este mai mică sau egală cu orice cheie din subarborele de rădăcină x.

Vom discuta în cele ce urmează doar heap-urile max, cele min fiind similare, doar cu modificarea corespunzătoare a inegalităților.

Pentru implementare se folosește la bază un vector în care, pe prima poziție, cea de indice 0, se plasează rădăcina, iar pentru nodul aflat pe poziția i fiul său stâng (dacă există) se află pe poziția $2i + 1$, iar cel drept (dacă există) se află pe poziția $2i + 2$. Părintele nodului de pe poziția i (pentru $i \neq 0$), se află pe poziția $(i - 1)/2$.

Un exemplu de heap-max este:



Observații:

- Dacă indicele primei poziții din vector se consideră 1, atunci relațiile dintre indicele unui nod și indicii fiilor, respectiv indicele părintelui, se schimbă în mod corespunzător [3].

- Vectorul aflat în spatele structurii de heap NU este sortat! Doar aranjat astfel, încât arborele binar asociat să respecte structura de heap.

Heap-urile binare permit implementarea eficientă a cozilor de prioritate. Acestea sunt structuri care permit stocarea și manipularea unor mulțimi dinamice ale căror elemente dispun de o proprietate numită prioritate. Procesarea datelor se face în acest caz în ordinea dată de prioritate.

Cozile de prioritate - *priority queues* - sunt utile de exemplu în gestionarea proceselor unui calculator sau în algoritmi de căutare precum *Dijkstra* sau A^*.

Similar cozilor obișnuite, adăugarea elementelor se realizează la unul dintre capete, iar extragerea la celălalt. Diferența dintre cele două structuri constă în faptul că o coadă de prioritate dispune de un mecanism de rearanjare a elementelor, astfel încât la începutul unei cozi de tip max se află întotdeauna elementul cu prioritatea maximă, iar în cazul cozilor de tip min elementul cu „costul" minim.

12.1.1 Operațiile principale

Operațiile principale pentru un heap sunt:

- Refacerea heap-ului, dacă s-a stricat la vârf. În acest caz, subarborii rădăcinii au structura de heap, dar cheia rădăcinii nu respectă ordonarea cerută. Pentru refacerea heap-ului, cheia din rădăcină trebuie coborâtă în heap, pe poziția potrivită folosind o funcție de tip *sift-down*, numită în [3] *max-heapify*. Algoritmul se găsește atât în [3] cât și în materialul de curs [8].
- Construirea unui heap pe baza unui vector, cu o funcție care apelează succesiv funcția *sift-down* [8].

Complexitatea de timp a algoritmului de construire a heap-ului este liniară în n, iar cea a algoritmului *sift-down* (*max-heapify*) este $O(\log_2 n)$, unde n este numărul de elemente din heap. Acest lucru se demonstrează în [3].

În cazul unei cozi de prioritate, aceasta se construiește prin inserții succesive de elemente. Operațiile de bază sunt:

- accesarea elementului de prioritate maximă, respectiv minimă (pentru cozi de prioritate min);
- extragerea elementului de prioritate maximă, respectiv minimă;
- adăugarea unui nou element.

Complexitate: accesarea elementului maxim, respectiv minim se realizează în complexitate de timp constantă. Celelalte operații au complexitate de timp $O(\log_2 n)$, unde n reprezintă numărul de elemente stocate în coadă.

Algoritmii pot fi găsiți cu explicații extinse în [8].

12.1.2 Cozi de prioritate în STL

În biblioteca STL cozile de prioritate sunt implementate de clasa `priority_queue`. Definiția acesteia este:

```
template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>
> class priority_queue;
```

Observăm următoarele elemente din definiție:

- `class T`: reprezintă tipul de date (*template*) care va fi stocat în coadă și care trebuie indicat la declararea unei variabile de tip `priority_queue`;
- `class Container = std::vector<T>`: indică containerul în care vor fi stocate elementele și care este predefinit de tip `vector`;
- `class Compare = std::less<typename Container::value_type>`: reprezintă comparatorul folosit pentru compararea elementelor, astfel încât să se respecte proprietățile de heap. Observăm faptul că în mod predefinit se folosește comparatorul din STL:

```
        std::less<T>
```

Comparator customizat În cazul obiectelor de tip `priority_queue`, comparatorul utilizat este deosebit de important, deoarece acesta determină modul de comparare a elementelor și, implicit, aranjarea lor conform regulilor unui heap. Așa cum s-a discutat în contextul arborilor binari de căutare, biblioteca STL pune la dispoziție comparatorii predefiniți `std::less`

și `std::greater`, care pot fi aplicați oricărui tip de date pentru care sunt definite operațiile de comparare. Un astfel de comparator suprascrie metoda `operator()`, care primește ca parametri două variabile de același tip și returnează o valoare booleană (`bool`). Astfel, în cazul `std::less`, operatorul returnează valoarea `true` dacă primul parametru este mai mic decât cel de-al doilea și `false` în caz contrar, în timp ce `std::greater` funcționează exact invers.

Atenție! Comparatorul `std::less` în contextul `std::priority_queue` NU definește comportamentul unei cozi de prioritate min ci a unei cozi de prioritate max. DE CE? Pentru că, în cazul `std::priority_queue`, dacă operatorul () din cadrul comparatorului returnează $true$, atunci se va plasa în coada de priorități **al doilea** argument înaintea primului.

Astfel:

- Pentru o coadă de priorități max care stochează numere întregi este suficientă declarația:

```
std::priority_queue<int> myQueue;
```

  pentru că în mod predefinit se va considera comparatorul `std::less`.

- Pentru o coadă de priorități min care stochează numere întregi este necesară declarația:

```
std::priority_queue<int, std::vector<int>,
   std::greater<int>> myQueue;
```

Atunci când vrem să definim o coadă de priorități care stochează elemente de un tip, pentru care nu sunt definiți operatorii de comparație, sau dacă dorim să modificăm modul de comparație al elementelor, putem defini un comparator customizat.

Exemplu: Considerăm elemente de tip `std::pair<int, int>`. În STL compararea pentru două elemente $e1$ și $e2$ de acest tip cu `std::less` se realizează în modul următor: întâi se compară $e1.first$ cu $e2.first$. Dacă $e1.first < e2.first$ comparatorul returnează $true$. Dacă $e1.first = e2.first$, abia atunci se compară $e1.second < e2.second$.

Să presupunem însă că dorim să comparăm modulele (norma L_2) unor vectori reprezentați de elemente de tip `std::pair<int, int>`. Ne trebuie deci un comparator care pentru două elemente de tip:

```
std::pair<int, int> v1, v2;
```

să returneze $true$, dacă

$$\sqrt{(v1.first)^2 + (v1.second)^2} < \sqrt{(v2.first)^2 + (v2.second)^2}$$

și $false$ altfel. Atunci un comparator potrivit pentru un heap-max ce conține astfel de perechi este definit mai jos. Pentru eficiență s-a renunțat la extragerea radicalului.

```
class compare {
public:
    bool operator()(const std::pair<int,int>& v1, const
       std::pair<int,int>& v2) const {
        return (v1.first*v1.first +
           v1.second*v1.second<v1.first*v2.first +
           v2.second*v2.second);
    }
};
```

Observații:

- Cei doi parametri ai operatorului au fost trimiși prin referință, pentru eficientizare și au fost declarați `const` pentru a nu se permite modificarea lor.
- operatorul`( )` este declarat `const` pentru că el nu trebuie să modifice membri ai clasei `compare`.
- S-a renunțat la extragerea radicalului pentru eficientizare.

Operații cu `priority_queue` Metodele specifice `priority_queue` sunt similare ca nume cu ce cele ale `queue`. Astfel avem:

- `top`: returnează elementul de la începutul cozii, adică cel cu prioritatea maximă, respectiv minimă, în funcție de tipul de coadă;
- `push`: adaugă un nou element în coadă, păstrând ordonarea corectă;
- `pop`: elimină elementul de la începutul cozii de prioritate și reordonează coada, astfel încât să se păstreze ordonarea heap-ului.

Mai multe detalii pot fi găsite în documentație online.

12.1.3 Sortarea prin heap (Heap Sort)

Heap Sort este un algoritm de sortare eficient, care transformă șirul într-un heap binar și extrage treptat elementele maxime (sau minime) pentru a construi șirul sortat. Această metodă combină avantajele unui algoritm cu complexitate garantată și utilizarea eficientă a memoriei. Heap-urile sunt abordate în capitolul legat de cozi de prioritate. Detalii pot fi găsite și în [3].

Heap Sort este util pentru șiruri mari și are avantajul de a oferi complexitate $O(n \log n)$ în orice caz, spre deosebire de sortările simple care pot avea $O(n^2)$.

Mod de funcționare - Algoritmul constă în doi pași principali:

1. **Construirea heap-ului**: se transformă șirul într-un heap binar max.
2. **Sortarea**: se extrage repetat elementul rădăcină (maximul) și se plasează la sfârșitul șirului, reducând heap-ul și refăcând proprietatea de heap. Se va obține sortare crescătoare a șirului.

Observație: Algoritmul poate fi adaptat astfel încât să funcționeze și cu un heap-min. De asemenea poate fi adaptat pentru sortare descrescătoare.

Exemplu ilustrativ - Se consideră șirul:

5	3	8	4	2

Pasul 1: se construiește max-heap:

$$8, 4, 5, 3, 2$$

Pasul 2: se extrage maximul și se reface heap-ul pentru restul:

$$[5, 4, 2, 3] \Rightarrow [5, 4, 3, 2]$$

Se repetă procesul până când șirul devine:

$$2, 3, 4, 5, 8$$

Complexitatea algoritmului

Fie n numărul de elemente din șir:

- Complexitatea în orice caz (cel mai bun, mediu sau nefavorabil) este $O(n \log n)$;

- Complexitatea spațială este $O(1)$, deoarece sortarea se face in-place.

Implementare în limbajul C++

```
// functie pentru refacerea heap-ului
void max_heapify(int* vector, int n, int i)
{
    int i_maxim = i;
    int stanga = 2 * i + 1;
    int dreapta = 2 * i + 2;

    if (stanga < n && vector[stanga] > vector[i_maxim])
        i_maxim = stanga;
    if (dreapta < n && vector[dreapta] > vector[i_maxim])
        i_maxim = dreapta;

    if (i_maxim != i){
        swap(vector[i], vector[li_maxim]);
        max_heapify(vector, n, i_maxim);
    }
}

void heapSort(int* vector, int n)
{
    // construire max-heap
    for (int i = n / 2 - 1; i >= 0; i--)
        max_heapify(vector, n, i);

    // extragere elemente din heap
    for (int i = n - 1; i >= 0; i--){
        swap(vector[0], vector[i]);
        max_heapify(vector, i, 0);
    }
}
```

Funcția `heapSort` sortează șirul `a` în ordine crescătoare prin construirea unui max-heap și extragerea succesivă a elementelor maxime. Algoritmul este eficient pentru șiruri mari și oferă complexitate garantată $O(n \log n)$.

În exemplul următor este prezentată implementarea algoritmului Heap Sort folosind containere STL, printre care `prority_queue`.

```
#include<iostream>
#include<queue>
#include<vector>
```

```
void heap_sort(std::vector<int>& v){
    //folosim o coada de prioritati min
    std::priority_queue<int, std::vector<int>, std::greater<int>>
       myQueue(v.begin(), v.end());

    for (int i = 0; i < v.size();i++){
        v[i] = myQueue.top();
        myQueue.pop();
    }
}

int main(){
    std::vector<int> vect;
    int n, x;
    std::cout << "n=";
    std::cin >> n;
    vect.resize(n);
    for (int i = 0; i < n; i++)
        std::cin >> vect[i];

    heap_sort(vect);
    for (int i = 0; i < n; i++)
        std::cout << vect[i] << " ";

    return 0;
}
```

12.2 Probleme rezolvate

12.2.1 Probleme teoretice

1. Care este numărul minim / maxim de noduri într-un heap max de înălțime 5?

   Rezolvare: Numărul minim de noduri într-un heap-max de înălțime h se obține atunci când toate nivelurile până la $h-1$ sunt pline (nu uităm că un heap este un arbore complet), iar pe ultimul nivel avem exact 1 nod. Cum pe un nivel k numărul maxim de noduri este 2^k putem astfel calcula numărul minim de noduri:

$$n_{min} = 1 + 2 + 2^2 + \ldots + 2^{h-1} + 1 = 2^h - 1 + 1 = 2^h$$

În cazul nostru $h = 5$, deci $n_{min} = 2^5 = 32$.

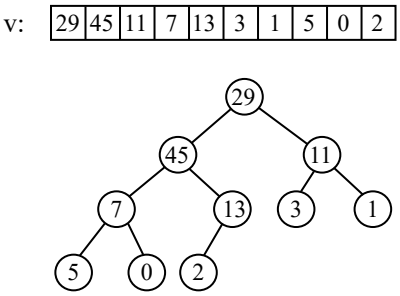
Numărul maxim de noduri se obține atunci când toate nivelurile sunt pline, deci:

$$n_{max} = 1 + 2 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$$

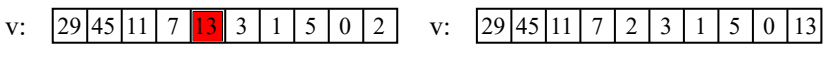
În cazul nostru $h = 5$, deci $n_{max} = 2^6 - 1 = 63$.

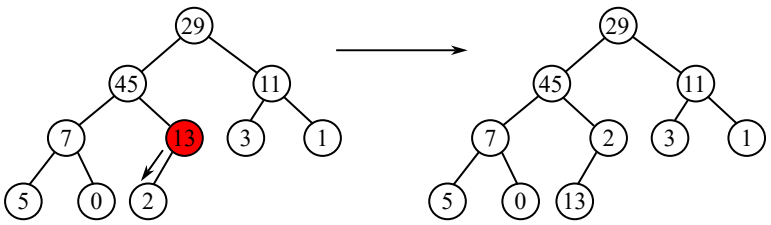
2. Se consideră vectorul $v = \{29, 45, 11, 7, 13, 3, 1, 5, 0, 2\}$. Să se construiască conform algoritmului de construcție a unui heap din [8] heap-ul min corespunzător.

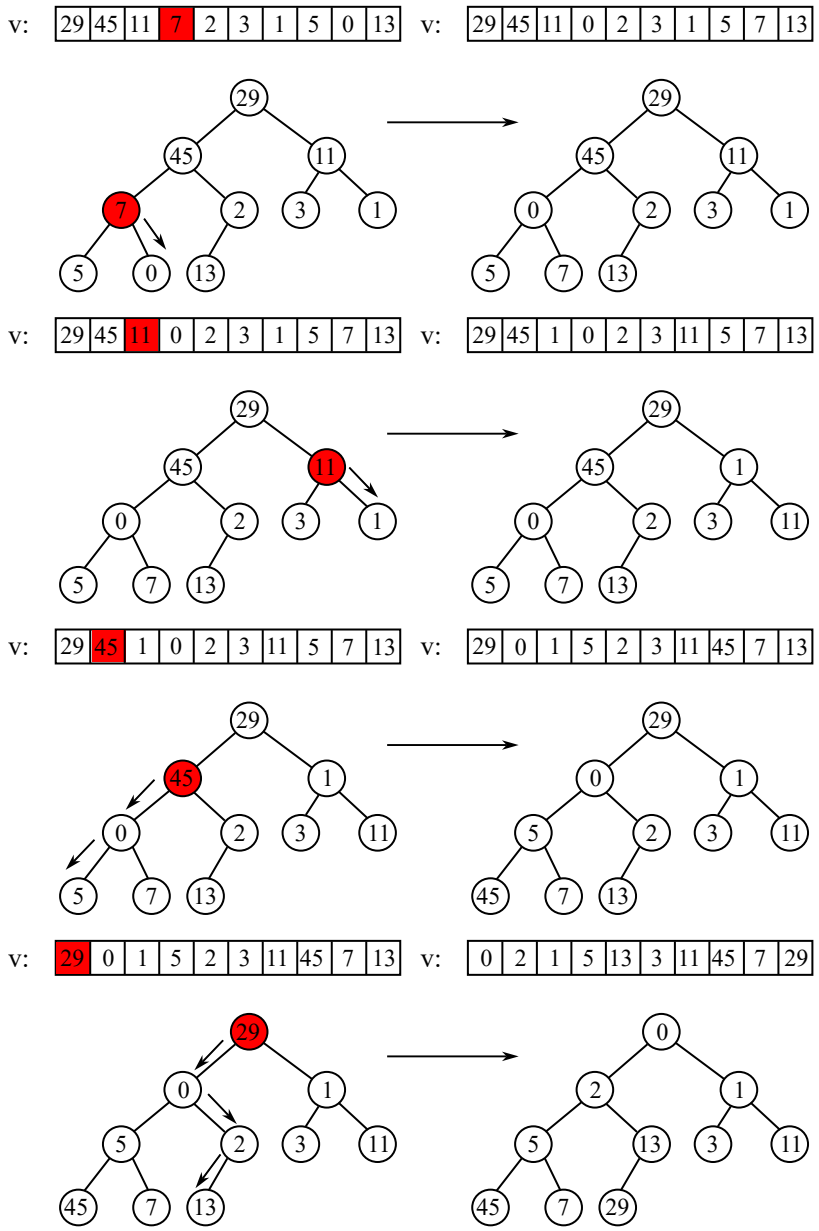
Rezolvare: Arborele binar corespunzător vectorului este:



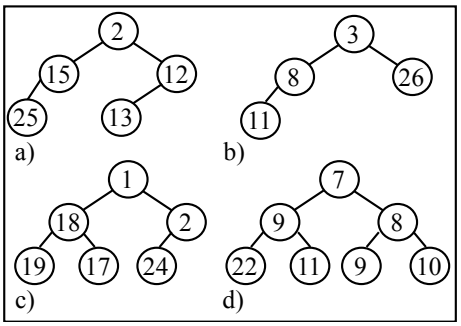
Observăm că nu avem un heap min, deoarece nu se respectă condiția de ordonare. Considerând vectorul de la dreapta la stânga, pornim cu nu nod $curent$ de la primul nod care are fii, acesta fiind nodul cu cheia 13 și refacem proprietatea de heap-min pentru subarborele de rădăcină $curent$, folosind *sift-down*, apoi trecem la următorul nod. Procesul este ilustrat în figurile:







3. Care dintre următorii arbori sunt heap-uri min?

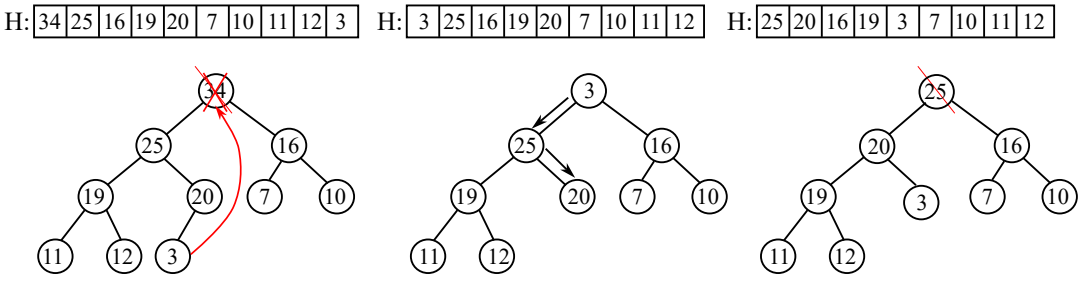


Rezolvare:

- Se observă ușor faptul că (a) nu este arbore complet. Nodul cu cheia 12 nu poate avea fii, dacă nodul cu cheia 15 nu are decât un fiu. Ultimul nivel trebuie completat de la stânga la dreapta.
- Arborele (b) respectă toate proprietățile unui heap-min.
- În arborele (c), care este complet, avem totuși o problemă: fiul drept al nodului 18 are o cheie mai mică (17), ceea ce nu este permis într-un heap-min.
- Arborele (d) respectă toate proprietățile unui heap-min.

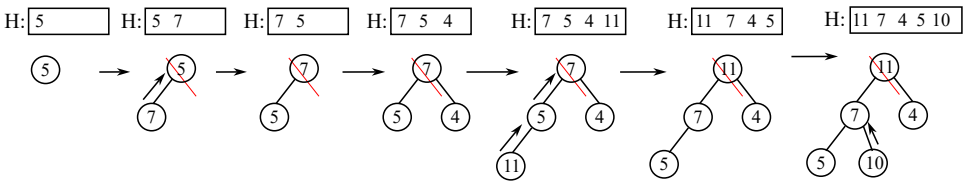
4. Cum arată heap-ul max $\{34, 25, 16, 19, 20, 7, 10, 5, 11, 12, 3\}$ după extragerea maximului?

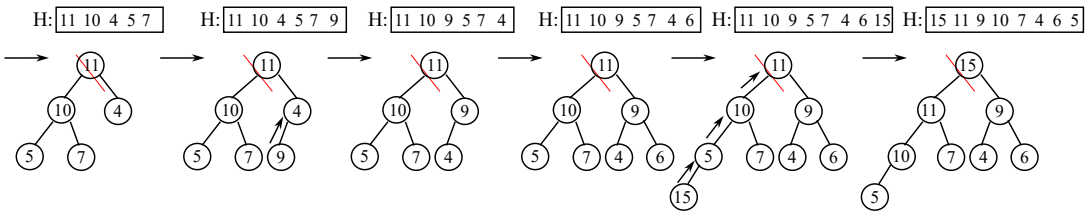
Rezolvare: Extragerea maximului dintr-un heap-max presupune înlocuirea primului element din heap cu elementul aflat pe ultima poziție, reducerea numărului de elemente ale heap-ului cu 1 și apoi refacerea heap-ului, care s-a stricat la vârf, apelând *sift-down*:



5. Inserați pe rând într-o coadă de priorități max inițial vidă cheile 5, 7, 4, 11, 10, 9, 6, 15.

Rezolvare: Soluția este prezentată în figurile de mai jos. La fiecare pas se adaugă la finalul heap-ului cheia curentă, apoi se reface heap-ul folosind *sift-up*.





12.2.2 Probleme aplicative

1. Se consideră că un angajat are de rezolvat o serie de sarcini. Fiecare sarcină are una dintre denumirile

   $$\{comanda_noua, rezolva_corespondenta, livreaza_intern, livreaza_extern, calculeaza_plati\}$$

   o prioritate și o durată estimată. Sarcinile se vor efectua în ordinea determinată de prioritate. Dacă două sarcini au aceeași prioritate, vor fi executate în ordine crescătoare a duratei. Scrieți un program în care inițial se generează în mod aleatoriu 4 sarcini. Acestea se inserează într-o coadă de priorități. Apoi se extrage pe rând câte o sarcină (și se consideră că se execută și se afișează sarcina executată). După fiecare sarcină extrasă se generează o nouă sarcină și se adaugă în coadă. După generarea în total a 10 sarcini, programul se oprește. Rezolvați folosind `std::priority_queue`.

   Rezolvare:

   Date intrare: vectorul de $sarcini$, pentru fiecare sarcină se cunosc denumirea ($string$), prioritatea (int) și durata estimată (int).

   Date ieșire: 10 sarcini rezolvate în ordinea priorității.

   Algoritm de rezolvare

   Etape:

   - Generăm în mod aleatoriu 4 sarcini și le introducem într-o coadă de priorități. Observație: Coada de priorități este de tip max, dacă o prioritate mai mare înseamnă o urgență mai mare și respectiv min, dacă o valoare mai mică a priorității înseamnă o urgentă mai mare. Deci trebuie stabilită la început semnificația valorilor priorităților.

- Se extrage prima sarcină din coadă și se afișează, apoi se generează o nouă sarcină, care se adaugă în coadă.

Observație: este necesar să definim un operator custom, deoarece în coadă se introduc tupluri de tip $< string, int, int >$, reprezentând sarcini, iar ordonarea se va face considerând întâi prioritatea (urgența), iar la priorități egale, durata.

Vom considera $<$ pentru prioritate (punem în vârf elemente cu prioritatea mai mare) și $>$ pentru durată (punem în vârf elemente cu durată mai mică) - vezi explicații legate de modul de funcționare al comparatorilor pentru cozile de prioritate în C++.

Algoritm în C++

```
#include<fstream>
#include<iostream>
#include<tuple>

//pentru generearea de numere aleatoare
#include<cstdlib>
#include<ctime>

//pentru priority-queue includem queue
#include<queue>
#include<vector>

//daca vreau sa am in varf elementele in functie de urgenta:
   0, 1, 2, 3 folosesc >
//daca vreau sa am in varf prioritatea cea mai mare, in
   ordinea: 3, 2, 1, 0 folosesc <

//definim tipul de date sarcina, pentru claritae
using sarcina = std::tuple<std::string, int, int>;

//definim comparatorul - se considera prioritarea cea mai
   mare in varf
class compare {
public:
    bool operator()(const sarcina& s1, const sarcina& s2)
       const {
        if(std::get<1>(s1) == std::get<1>(s2))
            return std::get<2>(s1) > std::get<2>(s2);
        return std::get<1>(s1) < std::get<1>(s2);
    }
};
```

Funcția care generează aleatoriu o nouă sarcină pe baza vectorului cu denumirile sarcinilor și o funcție care afișează o anumită sarcină:

```
std::tuple<std::string, int,
   int>genereaza_sarcina(std::vector<std::string> sarcini){
    std::string nume_sarcina = sarcini[rand() %
       sarcini.size()];
    int durata = rand() % 8 + 1;
    int prioritate = rand() % 3 ;
    return {nume_sarcina, prioritate, durata};
}

void print(sarcina s){
    std::cout << std::get<0>(s) << " cu prioritatea " <<
       std::get<1>(s) << " si durata " << std::get<2>(s) <<
       "ore.\n";
}
```

În funcția *main* vom rezolva problema:

```
int main()
{
    //declaram o coada de prioritati de sarcini, cu
       comparatorul custom
    std::priority_queue<sarcina, std::vector<sarcina>,
       compare> coada_sarcini;

    //definim vecrorul cu numele sarcinilor
    std::vector<std::string> sarcini = { "comanda_noua",
       "rezolva_corespondenta", "livreaza_intern",
       "livreaza_extern", "calculeaza_plati" };

    //initializam seed-ul pentru generatorul de numere
       aleatoare
    srand(time(NULL));

    //generam cele 4 sarcini initiale si le inseram in coada
    for (int i = 0; i < 4; i++){
        auto sarcina_curenta = genereaza_sarcina(sarcini);
        coada_sarcini.push(sarcina_curenta);
    }
    //la fiecare pas urmator extragem o sarcina din coada
    //generam una noua si o introducem in coada
    for (int i = 0; i < 10; i++){
        auto current = coada_sarcini.top();
        coada_sarcini.pop();
        print(current);
        auto sarcina_noua = genereaza_sarcina(sarcini);
```

```
            coada_sarcini.push(sarcina_noua);
        }
        return 0;
}
```

Analiza complexității: Complexitatea inserției și a extragerii maximului într-un heap max este $O(\log_2 n)$, unde n este numărul de elemente din heap. Cum la fiecare pas se extrage un element și se adaugă alt element, heap-ul își păstrează numărul de elemente, iar complexitatea totală este $O(k \log_2 n)$, unde n este numărul de elemente din heap, iar k numărul de adăugări și extrageri din heap.

2. **Codificarea Huffman**. Se citește un text dintr-un fișier. Să se construiască arborele de codificare Huffman corespunzător. Să se afișeze codul corespunzător fiecărui caracter și să se codifice textul. Folosiți `std::priority_queue`.

Codificare cu coduri de lungime variabilă

Algoritmul Huffman [3], [18] este folosit pentru compresia datelor fără pierdere de informație. Algoritmul are la bază ideea, ca în loc să se folosească pentru caractere coduri cu aceeași lungime (de exemplu 8 biți), să se codifice caracterele care sunt utilizate mai frecvent cu coduri mai scurte, iar cele mai rare cu coduri mai lungi. Astfel, pentru texte lungi se obține o reducere a numărului de biți necesari. Pentru a evita confuzia între coduri la decodificare, este necesar să se utilizeze un **cod prefix.** Un cod prefix este un cod în care nici un element al codului nu reprezintă un prefix pentru alt element. Acest lucru permite codificare/decodificare fără ambiguități.

Exemplu: Se consideră alfabetul alcătuit din primele 6 litere $\{a, b, c, d, e, f\}$ și următorul tabel de codificare:

	a	b	c	d	e	f
Frecvența (la 1000 caractere în lb. engl.)	45	13	12	16	9	5
Codificare - lungime fixă (3 biți)	000	001	010	011	100	101
Codificare - lungime variabilă	0	101	100	111	1101	1100

Codificarea 00101100110110100 folosind codurile cu lungime variabilă din rândul al treilea al tabelului se separă în secvențele 0 0 101 100 1101 101 0 0 ce reprezintă textul aabcebaa.

Arborele Huffman

Un mod de construcție a unui astfel de cod prefix de lungime variabilă se realizează cu ajutorului arborelui Huffman. În acest arbore, frunzele reprezintă caracterele de codificat. Muchia de la un nod părinte la fiul stâng se marchează cu caracterul 0, iar cea către fiul drept se marchează cu 1. Codul corespunzător unui caracter dintr-o frunză se obține prin concatenarea caracterelor care se află pre drumul de la rădăcină către frunza respectivă.

Observație: un cod optimal se obține pe baza unui arbore binar în care fiecare nod neterminal are exact doi fii.

Construcția arborelui de codificare Huffman

Se plasează fiecare caracter într-un nod frunză și se plasează într-o coadă de prioritate (min) de noduri. Astfel, în primul element din coadă va fi nodul de prioritate minimă, unde prioritatea este dată de frecvența caracterului respectiv.

Cât timp în coadă se găsesc cel puțin 2 noduri se extrag primele două minime, se leagă ca fii ai unui nou nod părinte, a cărui frecvență (prioritate) va fi calculată ca sumă a priorităților celor doi copii. Acest nod nou se inserează în coada de prioritate.

După obținerea arborelui Huffman se poate construi un *look-up table* în care fiecărui caracter i se asociază codul obținut pe baza arborelui. Acest tabel se folosește apoi pentru codificare.

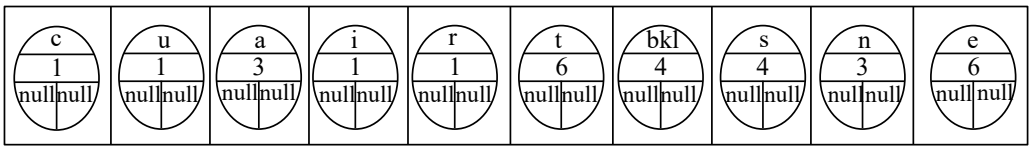
Decodificarea: pentru a putea decodifica corect documentul, este necesar ca atât cel care codifică, cât și cel care decodifică să utilizeze același arbore. Acest lucru se realizează ori prin folosirea de comun acord a aceluiași set de caractere cu aceeași frecvență sau prin transmiterea împreună cu textul codificat și a arborelui de codificare. În literatură sunt propuse diferite moduri de soluționare a acestei probleme.

Exemplu: Considerăm următorul text: "acesta este un test interesant". Avem următorul tablou de frecvențe:

a	c	e	i	n	r	s	t	u	spațiu (blk)
3	1	6	1	3	1	4	6	1	4

Construim un arbore în care fiecare nod are: un câmp frecvență, un câmp caracter și câmpuri de legătură către fiul stâng, fiul drept și către părinte (eventual). Dacă algoritmul de parcurgere al arborelui pentru determinarea codurilor este recursiv, nu este nevoie de legătura la părinte.

Pas 1: Se inserează într-o coadă de priorități (min) nodurile corespunzătoare fiecărui caracter. Rezultatul inserției, folosind un `unordered_map` pentru determinarea frecvenței caracterelor în text, este:



Observație: ordinea de inserție a caracterelor, este dată de ordinea în care sunt stocate în map, în cazul de față: c, u, a, i, r, t, blk (blank), s, n, e.

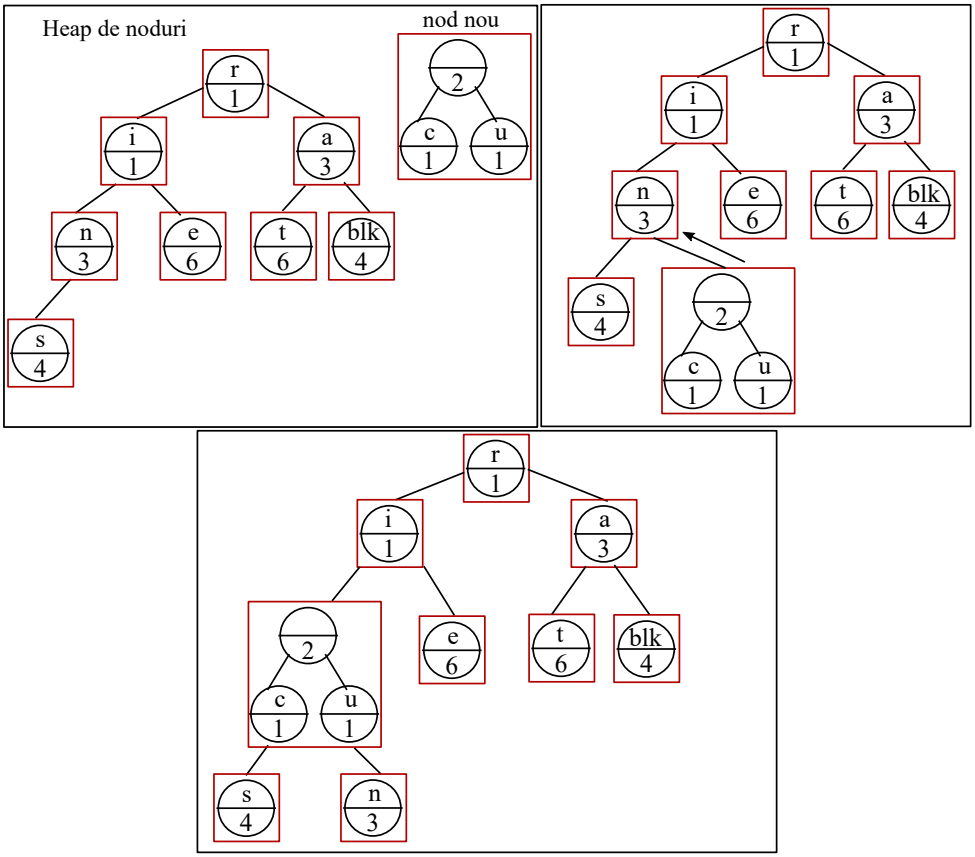
Pas 2: Cât timp în coadă există mai mult decât un nod, se procedează în modul următor:

- se extrag primele noduri (cu frecvența minimă) din coadă;
- se creează un nou nod care are ca frecvență suma frecvențelor celor două noduri;
- se leagă cele două noduri ca fii ai noului nod;
- se inserează apoi în coada de priorități.

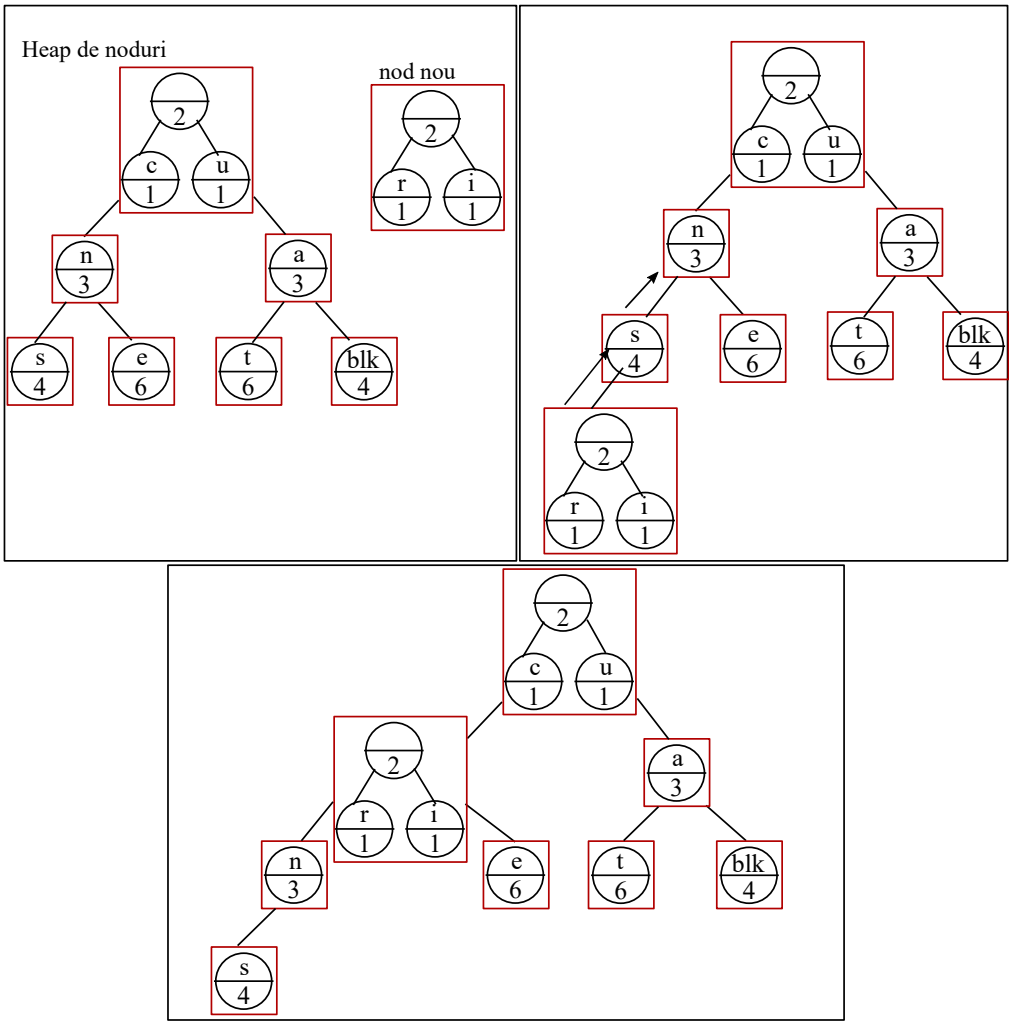
Observăm că, la fiecare extragere a două noduri și inserare a noului nod creat, dimensiunea heap-ului se reduce cu 1.

În următoarele figuri este ilustrat acest algoritm pentru caracterele din exemplu. Elementele din heap sunt încadrate în dreptunghiuri roșii. Reprezentarea este puțin inexactă, datorită faptului că în heap de fapt vom memora pointeri către rădăcinile subarborilor care se construiesc pas cu pas.

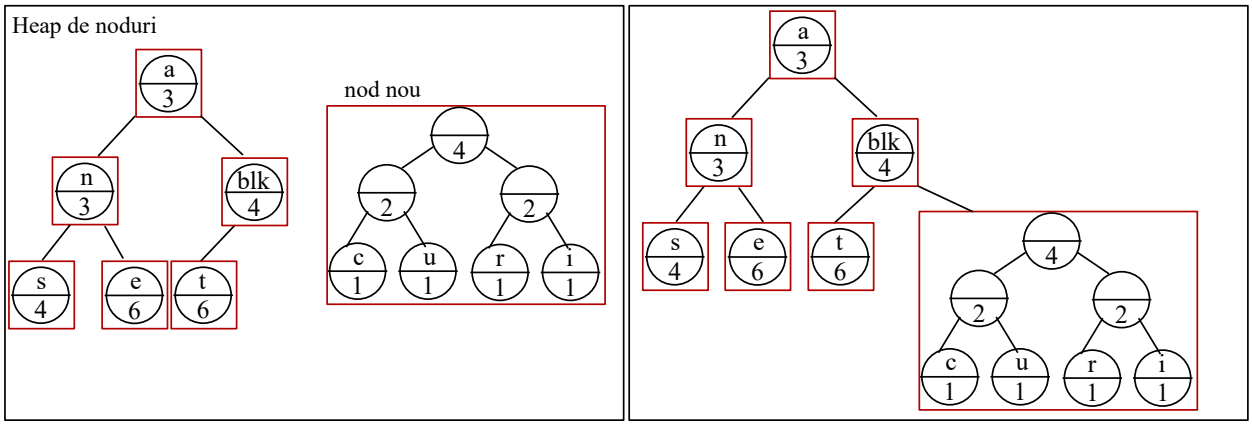
Inițial se extrag din heap nodurile corespunzătoare caracterelor c și u. Se creează un nod nou de care se leagă nodurile extrase. La refacerea heap-ului cu funcția *sift-down*, atunci când frecvența elementului din stânga este aceeași cu cea a elementului din dreapta, se va alege coborârea pe dreapta, datorită inegalității folosite în comparator (vezi codul C++ care urmează).

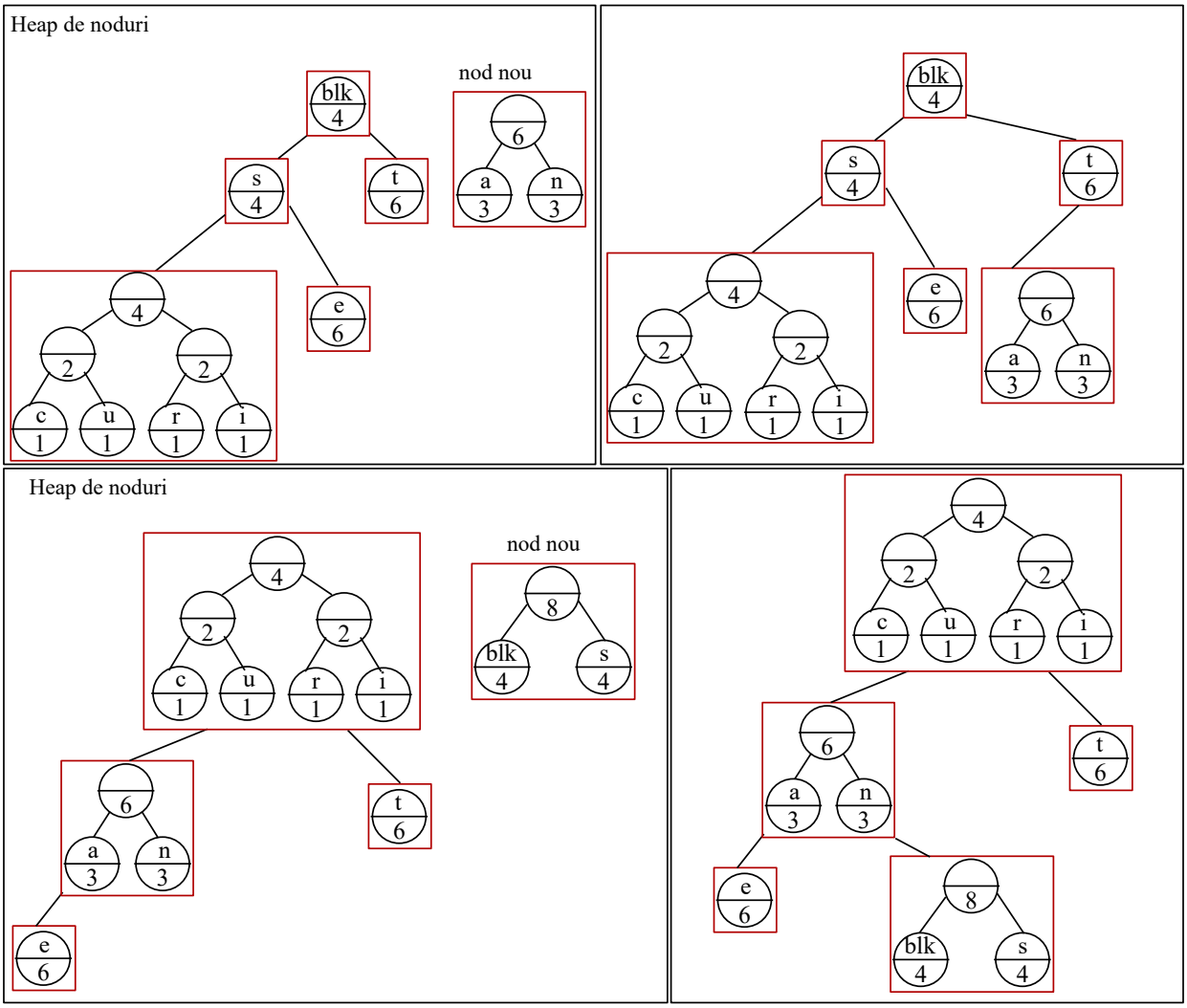


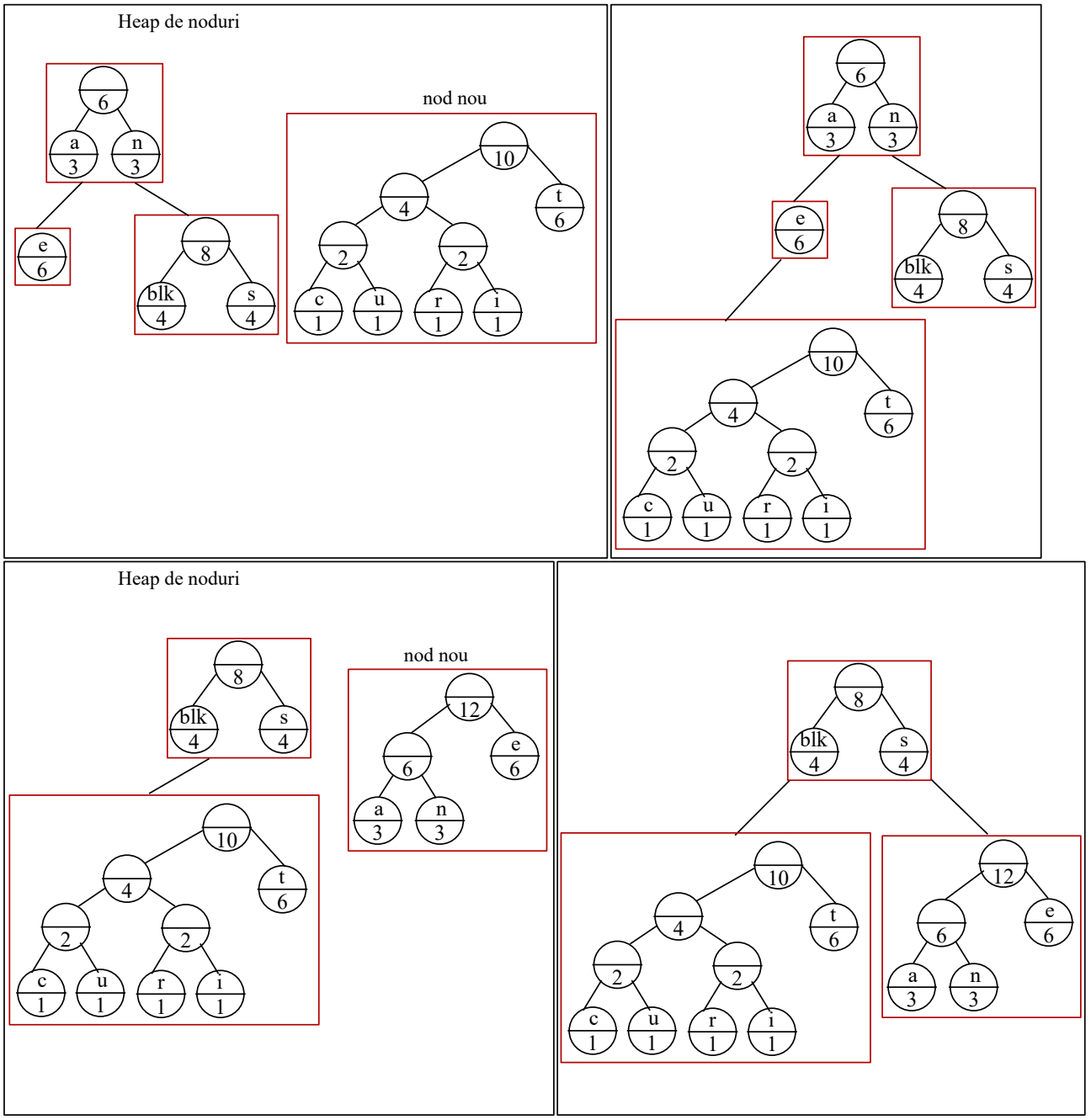
Acum se vor extrage următoarele două noduri: r cu frecvența 1 și i cu frecvența 1 și se vor lega ca fii ai unui nou nod cu frecvența 2. Acest nod va fi reinserat în heap:



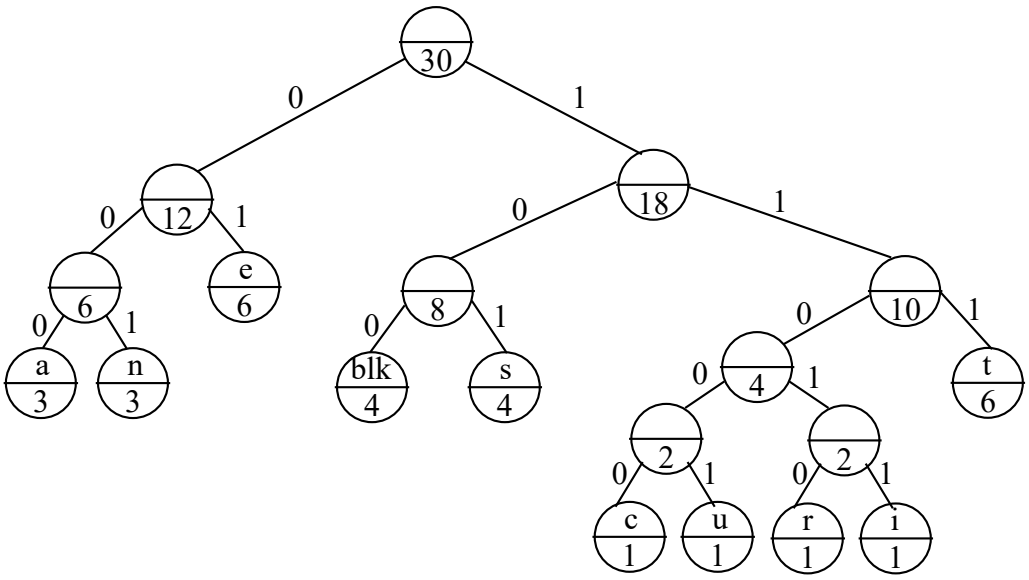
Algoritmul continuă în același mod:




Heap de noduri
nod nou
blk 4
s 4
t 6
6
a 3
n 3
4
2
2
c 1
u 1
r 1
i 1
e 6
Heap de noduri
nod nou
8
blk 4
s 4



Iar în final, după procesarea ultimelor 3 noduri se va ajunge la arborele din figură, pe ramurile cărora s-au plasat și codurile, 0 pe ramurile stângi și 1 pe cele drepte.



Codurile pentru fiecare caracter dintr-o frunză se obțin prin alăturarea caracterelor de 0 și 1, care se întâlnesc pe drumul de la rădăcină la frunza respectivă. Astfel pentru arborele de mai sus caracterele vor avea codurile:

Car	a	c	e	i	n	r	s	t	u	blank
Frecv.	3	1	6	1	3	1	4	6	1	4
Cod	000	11000	01	11011	001	11010	101	111	11001	100

Observație: la aceleași frecvențe, rezultatele pot fi ușor diferite, în funcție de ordinea de inserție a elementelor în coada de priorități, precum și de modul de tratare al elementelor cu aceeași prioritate.

Date de intrare: Un text pe baza căruia se construiește arborele de codificare Huffman.
Date de ieșire: Arborele Huffman și textul codificat.

Algoritmul în C++

Vom considera o structură de tip $node$, pentru nodurile arborelui și o variabilă de tip `unordered_map` pentru stocarea perechilor (caracter, frecvență).

Pentru coada de priorități, în care se vor insera pointeri la $node$ este necesar un comparator custom.

```
#include <iostream>
#include <fstream>
#include<vector>
#include<queue>
#include<string>
#include<unordered_map>

struct node {
    int frecv;
    char character;
    node* left, * right;

    node(int k = 0, char c = '\b') :frecv(k), character(c),
       left(nullptr), right(nullptr) {}
};
//comparatorul necesar pentru coada de prioritati
class Compare
{
public:
    //pentru min priority
    bool operator()(node* n1, node* n2)
    {
        return n1->frecv > n2->frecv;
    }
};
node* constructHuffman(std::unordered_map<char, int> car)
{
    std::priority_queue<node*, std::vector<node*>, Compare>
       coada;
    //initial adaug in coada cate un nod pentru fiecare
       caracter
    for (auto el:car)
    {
        node* nou = new node(el.second, el.first);
        coada.push(nou);
    }

    while (coada.size() > 1)
    {
        //la fiecare iteratie extrag din coada primele doua
           noduri cu cea mai mica frecventa
        node* n1, * n2;
        n1 = coada.top();
        coada.pop();
        n2 = coada.top();
        coada.pop();
```

```
        //creez un nou nod care are drept fii cele doua
           noduri extrase
        node* nou = new node(n1->frecv + n2->frecv);
        nou->left = n1;
        nou->right = n2;

        //adaug noul nod in coada
        coada.push(nou);
    }

    return coada.top();
}
//functia RSC modificata pentru construirea codului
//pe baza arborelui Huffman
//codurile vor fi adaugate intr-un unordered_map
void RSD(node* x, std::string cod, std::unordered_map<char,
   std::string>& coduri)
{
    if (!x)
        return;
    if (x->left)
        RSD(x->left, cod + "0", coduri);
    if (x->right)
        RSD(x->right, cod + "1", coduri);
    if (!x->left && !x->right)
        coduri[x->character] = cod;
}
int main() {
    std::unordered_map<char, int>frecvente;
    std::unordered_map<char, std::string>coduri;

    std::string cod = "";
    std::ifstream fisier("TextCodif.txt");
    //citim din fisier textul linie cu linie
    //si calculam frecventa fiecarui caracter
    if (fisier.is_open()) {
        std::string line;
        while (getline(fisier, line)) {
            for (char c : line)
                frecvente[c]++;
        }

        fisier.close();

        node* Huf_root = constructHuffman(frecvente);
        RSD(Huf_root,cod, coduri);
        for (auto el:coduri)
```

```
        {
            std::cout << el.first << ": " << el.second <<
               "\n";
        }
    }
    return 0;
}
```

Analiza complexității: calcularea frecvenței fiecărui caracter este în complexitate $O(n)$ unde n este numărul de caractere. Dacă se cunoaște această frecvență dinainte, nu este necesară calcularea. Deci nu vom lua în considerare complexitatea acestui pas.

Algoritmul efectiv de construire a arborelui Huffman presupune:

- Construirea cozii de prioritate inițiale, care este liniară în numărul k de caractere distincte existente (din considerente teoretice).
- Executarea ciclului $while$, care la fiecare pas extrage câte 2 noduri din heap și introduce nodul rezultat înapoi. Aceasta înseamnă că, la fiecare iterație, dimensiunea heap-ului scade net cu 1 (se elimină 2 noduri, se adaugă 1). Cum înălțimea heap-ului este $O(\log n)$ pentru un heap cu n noduri, iar n scade progresiv de la k la 1 pe parcursul celor $k-1$ iterații, costul fiecărei operații de heap (extragere sau inserare) scade și el pe măsură ce algoritmul avansează.
  Costul total al buclei este deci:

$$\sum_{j=1}^{k} O(\log_2 j) = O(\log_2 (k!)) = O(k \log_2 k)$$

  Observație: din aproximarea lui Stirling obținem $O(\log_2 (k!)) = O(k \log_2 k)$. Aceasta poate fi găsită în capitolul 3 din [3].

3. **Interclasarea optimală.** Se consideră n vectori de numere întregi, sortați crescător. Scrieți o funcție eficientă care returnează un vector sortat crescător, ce conține elementele tuturor celor n vectori. Folosiți o coadă de prioritate (`std::priority_queue`).

   Rezolvare:

   Date de intrare: Cei n vectori sortați crescător și pentru fiecare vector k numărul său de elemente $len(v_k)$.

Date de ieșire: Un vector ce conține toate elementele celor n vectori de intrare sortate crescător.

Algoritm de rezolvare

Vom considera pentru fiecare vector v_k un indice $index_k$ cu care se va parcurge acest vector. La început toți acești vectori sunt evident 0.

Considerăm un heap-min în care se introduc inițial perechile de tipul $(v_k[0], k)$. Observăm că heap-ul conține elemente de tip pereche, în care primul câmp al perechii reprezintă un element al unui vector, iar al doilea, indicele vectorului de proveniență. Apoi începe o buclă în care:

- se extrage din vârful heap-ului perechea (val, k) și valoarea se va plasa în vectorul sortat;
- se incrementează $index_k$ în vectorul v_k;
- se adaugă în heap perechea $(v_k[index_k], k)$.

Acest proces generalizează algoritmul de interclasare, prin parcurgerea simultană a tuturor vectorilor și plasarea în vectorul rezultat mereu a celei mai mici valori curente.

Algoritmul în C++

```
#include <iostream>
#include <fstream>
#include<vector>
#include<queue>

std::vector<int>InterclasareOpt(std::vector<std::vector<int>>
   vectori)
{
    //definim o coada de prioritati min, deci folosim
       comparatorul std::greater
    std::priority_queue<std::pair<int, int>,
       std::vector<std::pair<int,int>>,
       std::greater<std::pair<int,int>>> coada;

    int nr_vectori = vectori.size();
    //definim un vector in care vom tine mine pentru fiecare
       vector indicele curent
    std::vector<int> indici;
```

```
    std::vector<int>rezult;
    indices.resize(nr_vectori);

    //plasam in coada de prioritati elementul minim din
       fiecare vector (adica cel de pe pozitia 0)
    //impreuna cu indicele elementului, evident 0
    for (int k = 0; k < nr_vectori; k++)
    {
        coada.emplace(vectori[k][0], k);
    }

    while (!coada.empty())
    {
        //in coada avem perechi (val, vect)
        //unde vect reprezinta indicele vectorului
        //din care provine val
        int val = coada.top().first;
        int vect = coada.top().second;
        coada.pop();
        rezult.push_back(val);
        if (indici[vect] < vectori[vect].size()-1)
        {
            indici[vect]++;
            coada.emplace(vectori[vect][indici[vect]], vect);
        }
    }
    return rezult;
}

int main() {
    std::vector<std::vector<int>>vectori = { {1, 3, 5, 11,
       25}, {3, 4, 7, 8}, {0, 4, 5}, {4, 7, 13, 15,19} };
    std::vector<int>interclas = InterclasareOpt(vectori);

    for (int el : interclas)
    {
        std::cout << el << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Analiza complexității: La fiecare iterație, în heap se află cel mult n elemente și anume câte unul din fiecare vector: cel mai mic element încă neprocesat. Operațiile de extragere, respectiv de

adăugare a unui element în heap se efectuează în complexitate $O(\log n)$. Numărul de elemente care sunt introduse, respectiv extrase, este

$$N = \sum_{k=1}^{n} \text{len}(vec_k),$$

unde $\text{len}(vec_k)$ reprezintă lungimea vectorului k. Deci, în total, complexitatea este $O(N \log n)$. Dacă numărul de vectori n este mic în raport cu N (de exemplu constant), factorul $\log n$ poate fi considerat neglijabil, iar algoritmul este practic liniar în N.

12.3 Probleme propuse

12.3.1 Probleme teoretice

1. Care este înălțimea unui heap-max cu 30 chei? Dar cu 46 chei?

2. Inserați pe rând cheile 5, 7, 2, 11, 9, 25, 14, 20, 13 într-o coadă de priorități max inițial vidă. Cum arată heap-ul?

3. Să se reprezinte grafic toate heap-urile min posibile conținând cheile 1,2,3,4. Fiecare cheie apare exact o dată în fiecare heap.

4. Este $v = \{2,10,23,17, 4,11,35,25,15,7\}$ heap-min? Dacă nu, să se construiască un heap-min folosind algoritmul prezentat în [8]. Apoi să se extragă minimul. Cum a arată în final heap-ul?

5. Într-un heap-min al treilea cel mai mic element se va afla întotdeauna pe nivelul 1? Justificați răspunsul.

6. Se consideră un vector V reprezentând un heap-min cu 45 de elemente. Care dintre următoarele afirmații sunt adevărate?

   a) Al treilea cel mai mic element din heap se află sigur printre fiii rădăcinii.

   b) Căutarea elementului maxim se poate realiza în complexitate logaritmică.

c) Ordinea elementelor în heap obținut prin inserții succesive nu depinde de ordinea în care au fost inserate elementele în heap.

d) Dacă pozițiile încep de la 0 în vectorul corespunzător heap-ului, atunci cel mai mic indice, corespunzător unei frunze este 23.

e) Pentru sortarea unui vector în manieră heap-sort se poate folosi și un heap-min.

12.3.2 Probleme aplicative

*

1. Folosind pseodocodul din [3] sau [8] implementați o structură / clasă *coada_de_prioritati*, cu toate metodele aferente.

2. Implementați algoritmul heap-sort pe baza pseudocodului ([8]) și sortați un vector cu acest algoritm.

3. Sortați un vector folosind `priority_queue` din STL.

**

1. Să se scrie o funcție care are ca parametru un vector și care verifică, dacă acesta este heap-max. **Indicație**: puteți folosi problema rezolvată de la capitolul arbori, care verifică dacă un arbore binar este arbore complet.

2. La un centru de distribuție sosesc pachete, care trebuie livrate la diferite adrese. Pentru fiecare pachet se cunosc coordonatele adresei de livrare de tip (x, y) și data maximă până la care trebuie livrate (în format $AAMMZZ$ - de exemplu 250920 reprezintă 20 septembrie 2025). Pachetele vor fi date livratorilor prioritizând pachetele cu cea mai apropiată dată de livrare, iar cele cu aceeași dată vor fi prioritizate după distanța de la centrul de livrare (cele aflate la distanță mai mică se livrează mai întâi). Scrieți un program, care citește dintr-un fișier datele pentru n pachete și afișează care sunt primele k pachete ce vor fi livrate, folosind o structură de tip `priority_queue`.

3. O imprimantă folosită în comun primește diferite comenzi de printare. Fiecare comandă are un număr de pagini, o prioritate (director-1, secretariat-2, practicant-3) și ora trimiterii.

Comenzile sunt executate în ordinea dată de: întâi cea cu mai mică prioritate, la priorități egale, cele cu număr mai mic de pagini și dacă atât prioritatea cât și numărul de pagini sunt egale, se prioritizează cele mai vechi comenzi. Scrieți un program, care generează inițial în mod aleatoriu 5 sarcini de printare, le plasează într-o coadă de priorități, iar apoi la fiecare iterație execută o sarcină și apoi se generează una nouă, care se va adăuga la coada de priorități. Afișați primele 10 sarcini, care se vor executa.

***

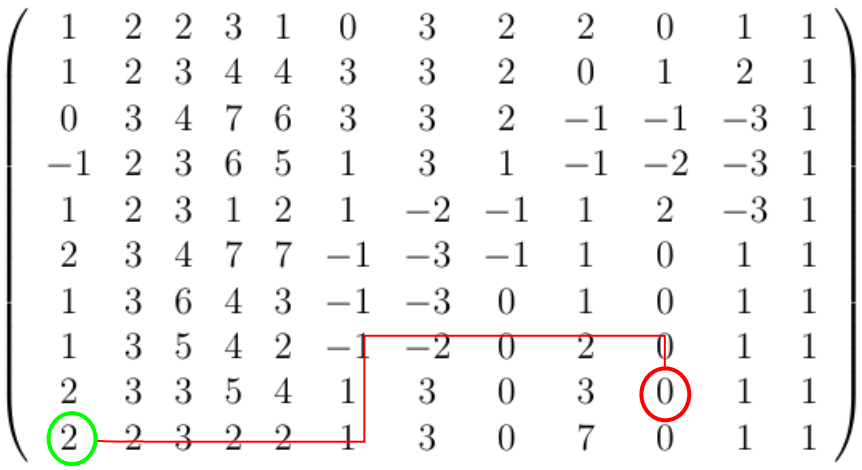
Problema medianului. Se consideră un vector de n numere întregi. Se numește **median** acel element din vector, care s-ar afla pe poziția din mijloc în vectorul sortat. Dacă n este număr par, atunci medianul este media aritmetică a celor 2 numere aflate pe pozițiile din centru.
Exemplu: pentru $v = \{6, 1, 5, 2, 0\}$ medianul este 2. Pentru $v = \{4, 1, 5, 2, 6, 3\}$ medianul este 3.5.
Să se citească pe rând dintr-un fișier n numere și să se afișeze după fiecare citire medianul curent al numerelor citite.
Exemplu: se citesc din fișier elementele 5, 2, 8, 6, 3, 2, 9. Se vor afișa: 5, 3.5, 5, 5.5, 5, 4, 5.
NU se sortează numerele!

****

Problemele din această secțiune presupun folosirea algoritmului A^* utilizând o coadă de priorități. Pentru algoritmul A^* puteți folosi indicațiile din [19].

1. **Problema excursionistului.** Un drumeț trebuie sa ajungă de la localitatea $A(x_0, y_0)$ la localitatea $B(x_1, y_1)$ parcurgând un relief variat. Excursionistul caută drumul cel mai convenabil din punct de vedere al efortului depus. Harta este reprezentată de o grilă dreptunghiulară de numere, reprezentând denivelarea punctului respectiv față de nivelul mării (0). Costul deplasării de la un pătrat al grilei la altul este dat de diferența de nivel + 1. În plus, drumețul nu poate depăși o diferență de nivel mai mare de 2 unități. De asemenea nu poate trece prin locuri mai adânci de -2. Deplasarea se face sus-jos-stânga-dreapta (nu pe diagonală). Reprezentați pe hartă un traseu optim pentru drumeț. Folosiți algoritmul A^*

Exemplu: se consideră harta 10×12 din figură. Drumul este marcat cu roșu de la punctul de start (roșu) la punctul de sosire (verde).



2. **Rezolvare puzzle**: Această problemă presupune rezolvarea unui joc, în care se consideră 8 plăcuțe pătrate numerotate de la 1 la 8, plasate într-o ramă pătrată de dimensiune 3×3. O poziție este liberă. Orice plăcuță vecină cu poziția liberă poate fi glisată pe această poziție. Se cere șirul de configurații (afișat), care duc la aranjarea numerelor în ordine crescătoare, cu poziția liberă în colțul dreapta-jos al cadrului.

   Folosiți algoritmul A^*, precum și structurile de date din STL potrivite (`priority_queue`, `unordered_map`, `vector` etc.)

   Costul unei configurații = numărul de mutări care s-a efectuat din configurația inițială până la aceasta.

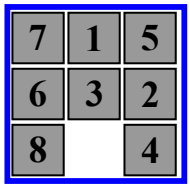
   Euristica 1: - numărul de plăcuțe care încă nu se află pe poziția potrivită.

   Păstrați configurațiile deja generate împreună cu o legătură (definită în mod potrivit) către configurația din care s-a obținut (părintele) într-un `unordered_map`, pentru a putea apoi genera șirul de configurații de la cea inițială la soluție. Pentru exemplul din figură $h_1 = 9$

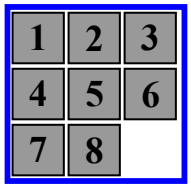
   Euristica 2: suma distanțelor de la fiecare piesă la poziția corectă (*city-block distance*). Pentru exemplul din figură avem $h_2 = 3 + 1 + 2 + 2 + 2 + 2 + 1 + 3 = 18$. Euristica 2 este mai eficientă decât 1, pentru ca se apropie mai bine de costul real.

Indicație de rezolvare: Pentru simplificare, considerați la fiecare pas, că se mută poziția liberă sus, jos, la stânga sau la dreapta.

Exemplu pentru configurația din figură:

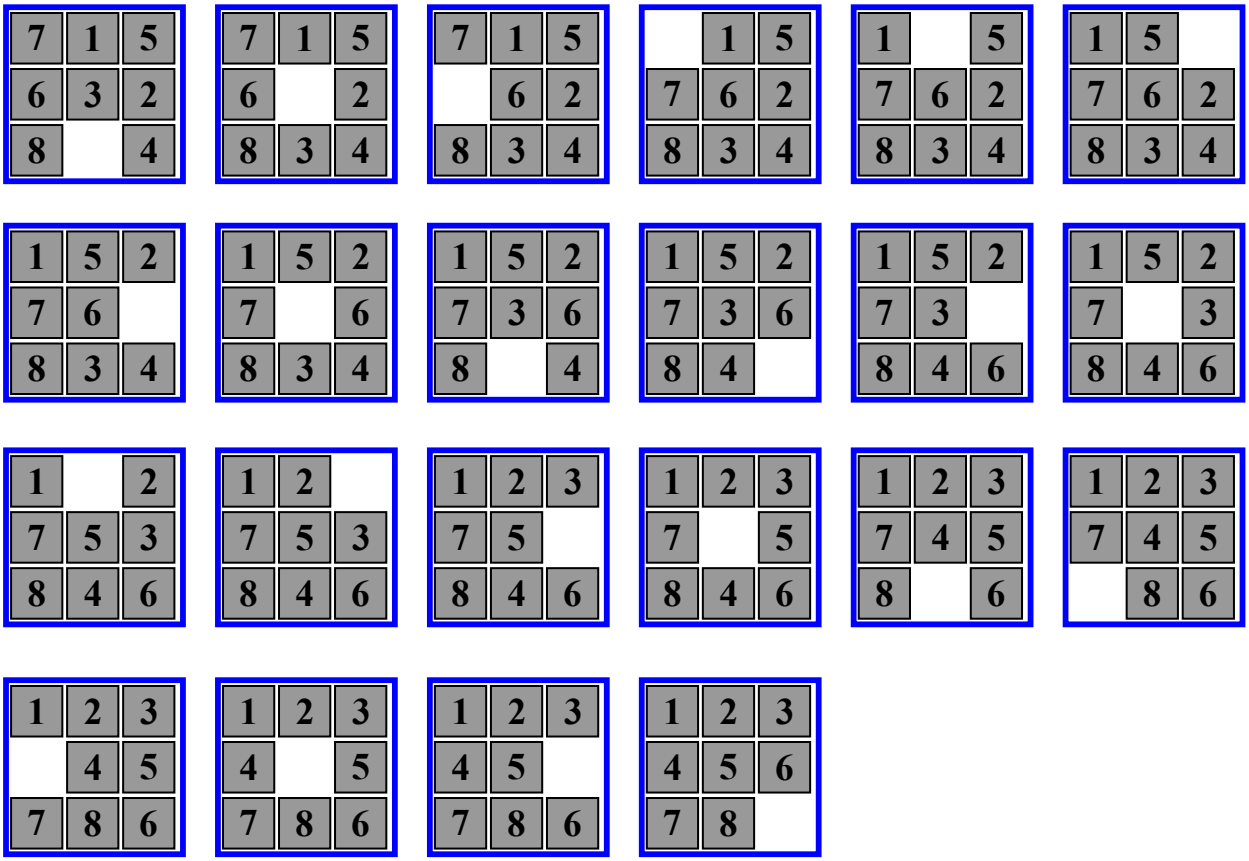


Configurație inițială



Configurație finală

configurațiile intermediare folosind rezolvarea cu euristica 2 și algoritmul A^* sunt:



Capitolul 13

Arbori binari de căutare

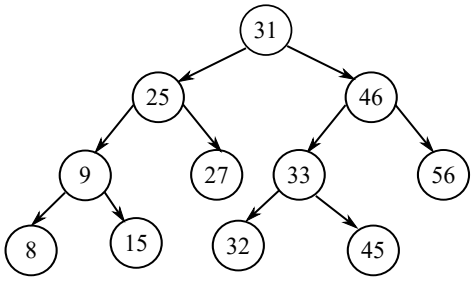
13.1 Aspecte teoretice

Arborii binari de căutare au fost dezvoltați pentru a permite menținerea sortată a unei mulțimi dinamice de date, precum și căutarea în manieră binară într-o astfel de mulțime. Acești arbori stau la baza implementării dicționarelor ordonate în diferite limbaje de programare.

Un arbore binar de căutare este un caz particular de arbore binar, cu proprietatea că pentru fiecare nod x, toate nodurile din subarborele stâng au cheile mai mici decât cheia lui x și toate nodurile din subarborele drept au cheile mai mari decât cheia lui x. Relația de inegalitate este în general strictă, presupunând unicitatea cheilor. Există și arbori binari de căutare în care este permisă existența de chei duplicate, deci în care relația de ordonare nu este strictă, dar aceștia nu vor face obiectul acestei prezentări teoretice, fiind vorba doar despre detalii de tratare a cheilor egale.

Este posibil să avem și arbori cu o ordonare inversă, adică, pentru orice nod x, nodurile din subarborele său stâng să aibă cheile mai mari, iar cele din subarborele drept să aibă cheile mai mici. Acest lucru presupune doar o schimbare de comparator (vezi discuția despre comparatori de la capitolul cu cozi de prioritate).

Un exemplu de arbore binar de căutare cu chei numere naturale este prezentat în figură.



Pentru implementarea unui arbore binar de căutare vom considera noduri cu o structură de 4 câmpuri: key - cheia nodului, $parent$ - pointer către părinte, $left$ și $right$ pointeri către fiii stâng și respectiv drept.

În plus, vom considera o structură $BinTree$, care are un membru $root$ de tip pointer la nod, reprezentând rădăcina arborelui. Toate funcțiile vor fi membre ale acestei structuri, deci nu va fi nevoie să trimitem ca parametru arborele. Astfel, algoritmii prezentați în continuare (sub formă de implementare C++), care se bazează pe algoritmii din [3] și [8], sunt adaptați în consecință (se consideră că funcțiile au acces direct la rădăcina arborelui). În plus, pentru anumite operații va fi prezentată versiunea recursivă.

Pentru noduri și arbore propunem următoarea structură șablon (**template**) în C++, astfel încât să poată fi folosită pentru orice tip de cheie. **Atenție**: comparatorul merge doar pentru tipuri primitive. Structura nu va funcționa bine, dacă tipul de date este unul, pentru care nu sunt definite comparațiile.

```
template<typename T>
struct BinTree {
    struct node {
        T key;
        node* parent;
        node* left;
        node* right;

        node(T val) :key(val), parent(nullptr), left(nullptr),
           right(nullptr) {}

    };

    node* root = nullptr;
    //Aici vor urma functiile pentru opetarii pe arbore
    void Insert(T val); //insereaza cheia val, daca nu exista deja
    node* Find(T val); //cauta nodul cu cheia val
    node* Minimum(node* x); //cauta nodul cu cheia minima
    node* Maximum(node* x);//cauta nodul cu cheia maxima
    node* Successor(node* x); //cauta nodul succesor al lui x
    node* Predecessor(node* x); //cauta nodul predecesor al lui x
    void Erase(node* z); //sterge nodul z
    bool Empty(); //verifica daca arborele e vid
    void Clear(); //goleste arborele prin stergerea tuturor
       nodurilor
    void RSD(node* x); //parcurgerea recursiva RSD de la nodul x
```

```
    void SRD(node* x);//parcurgerea recursiva SRD de la nodul x
    // etc.
};
```

O parte dintre funcțiile din structura $BinTree$ vor fi descrise mai jos, împreună cu implementările lor, iar restul sunt propuse ca exerciții, pe baza pseudocodului din bibliografie.

13.1.1 Operații principale

Operațiile principale pe un arbore binar de căutare sunt:

1. **Căutarea** în manieră binară unei chei key. Se pornește cu un nod x din rădăcină și la fiecare pas se compară key cu cheia nodului curent. Dacă sunt egale, atunci se returnează pointer la nodul curent, altfel dacă $key < x.key$ se va continua căutarea în subarborele stâng al lui x, iar dacă nu, în subarborele drept. Funcția de căutare trebuie să returneze adresa nodului căutat sau $nullptr$ în cazul în care nu este găsită cheia căutată. Algoritmul în pseodocod este prezentat în [8], iar codul corespunzător structurii $BinTree$, definită mai sus, este următorul:

```
template<typename T>
BinTree<T>::node* BinTree<T>::Find(T key)
{
    node* current = root;
    while (current && current->key != key)
    {
        if (key < current->key)
            current = current->left;
        else
            current = current->right;
    }
    return current;
}
```

2. **Inserarea** unei chei. Pentru a insera o cheie val într-un arbore binar de căutare T, se pornește de la rădăcină și se coboară în arbore în aceeași manieră ca în cazul căutării. De fapt se caută poziția de inserție, astfel încât să se păstreze ordinea cheilor. Algoritmul în pseudocod este prezentat în [8]. Mai jos este prezentat codul C++ corespunzător inserției unei chei noi în structura $BinTree$ definită mai sus.

```
template<typename T>
void BinTree<T>::Insert(T val)
{
    node* newNode = new node(val);
    node* current = root;
    node* parent = nullptr;
    while (current && current->key != val)
    {
        parent = current;
        if (val < current->key)
            current = current->left;
        else
            current = current->right;
    }
    if (current)
    {
        std::cout << "Key already present";
        return;
    }
    newNode->parent = parent;
    if (parent == nullptr)
    {
        root = newNode;
    }
    else {
        if (val < parent->key)
            parent->left = newNode;
        else
            parent->right = newNode;
    }
}
```

3. **Ștergerea** unui nod z, care a fost inițial identificat prin căutare se realizează în modul următor.

   - Dacă nodul z nu are fii, atunci se șterge pur și simplu prin legarea părintelui său de pointerul nul.
   - Dacă nodul z are un singur fiu, atunci se înlocuiește cu acesta.
   - Dacă z are doi fii, atunci se va determina succesorul (sau predecesorul) y al lui z, se va înlocui nodul z cu nodul y, iar în locul nodului y, va urca fiul acestuia.
   - Se eliberează memoria alocată pentru z.

Algoritmul în pseudocod pentru ștergere și pentru determinarea succesorului binar al unui nod poate fi găsit în materialul de curs [8].

Complexitatea operațiilor

Se arată în bibliografie faptul că, din punct de vedere al timpului de execuție, operațiile principale într-un arbore binar de căutare au complexitatea $O(h)$, unde h este înălțimea arborelui.

Pentru un arbore cu n noduri, înălțimea maximă se obține atunci când fiecare nod intern are un singur fiu. Astfel arborele ajunge să semene cu o listă, iar $h = n - 1$.

Atunci când arborele este echilibrat $h = \log_2 n$.

Astfel, în cel mai defavorabil caz, complexitatea este liniară în n, iar căutarea în arborele binar de căutare nu este mai eficientă decât căutarea secvențială.

Scopul arborilor binari de căutare este însă acela de a permite o căutare în complexitate logaritmică, similară căutării binare în șirurile sortate. În plus, arborii binari de căutare ar trebui să permită păstrarea sortată a cheilor, tot în complexitate logaritmică.

Pentru a asigura faptul că un arbore binar de căutare are o înălțime proporțională cu $\log_2 n$, unde n este numărul de noduri din arbore, au fost dezvoltați arborii care se auto-echilibrează. Vom discuta în continuare 2 tipuri de astfel de arbori: *arborii AVL* și *arborii roșu-negru (ARN)*. Echilibrarea acestor arbori se realizează folosind operații de rotație.

Rotația

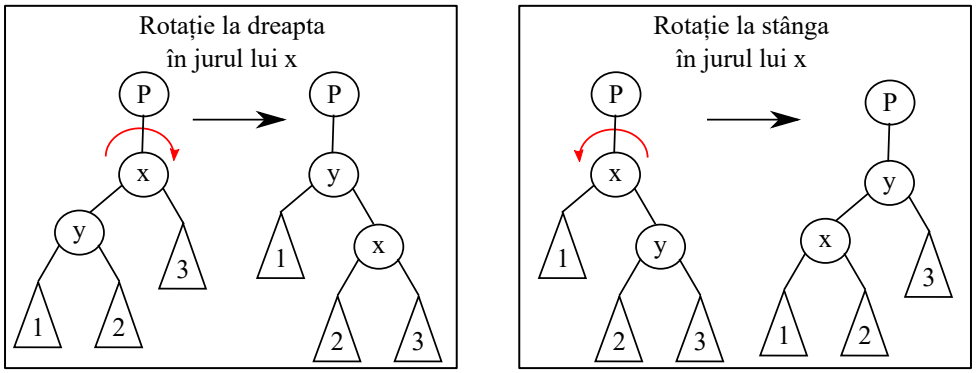


Figura 13.1: Rotația în jurul unui nod. Prin triunghiuri au fost reprezentați subarborii corespunzători nodurilor care se rotesc. Acești subarbori pot fi și vizi, dar pot conține și mai multe noduri.

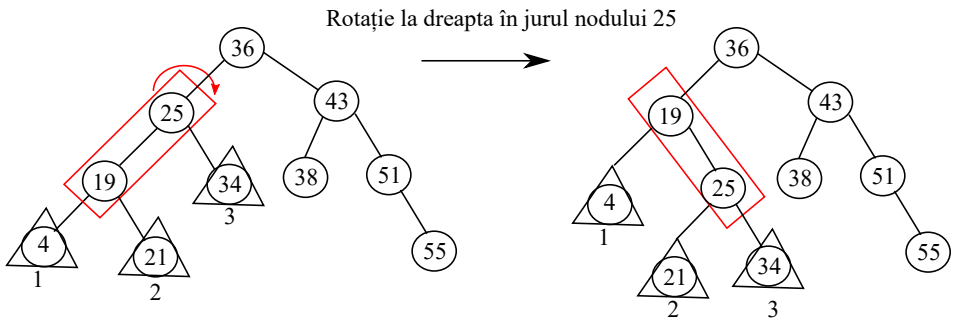


Figura 13.2: Exemplu de rotație la dreapta în jurul nodului 25. Chenarul roșu indică nodurile care își schimbă poziția relativă, iar subarborii implicați în rotație sunt delimitați cu triunghiuri numerotate.

O operație de rotație într-un arbore binar de căutare este o operație de modificare locală a structurii de pointeri, care păstrează însă relația de ordonare a cheilor, caracteristică acestui tip de arbori. În figura 13.1 este prezentată schema generală a rotației în jurul unui nod x. Observăm că subarborele 2 al nodului x se mută ca fiu al nodului y. În figura 13.2 este prezentat un exemplu concret de rotație în jurul unui nod într-un arbore binar de căutare.

13.1.2 Arbori AVL

Arborii AVL sunt arbori binari de căutare echilibrați, propuși de G.M. Adelson-Velsky și E.M. Landis [20]. Fiecărui nod x i se asociază un *factor de balansare*, egal cu diferența dintre înălțimile subarborilor nodului x. Echilibrul este menținut atâta timp cât valoarea absolută a acestui factor nu depășește 1, fapt care asigură o complexitate logaritmică în numărul de noduri din arbore pentru operațiile de bază.

Factorul de balansare al unui nod x poate fi definit ca diferența dintre înălțimea subarborelui drept al lui x și cea a subarborelui stâng:

$$factor(x) = h(x.right) - h(x.left)$$

Un nod se numește **balansat**, dacă factorul său este -1, 0 sau 1. Un arbore binar de căutare se numește arbore AVL, dacă fiecare nod al său este balansat.

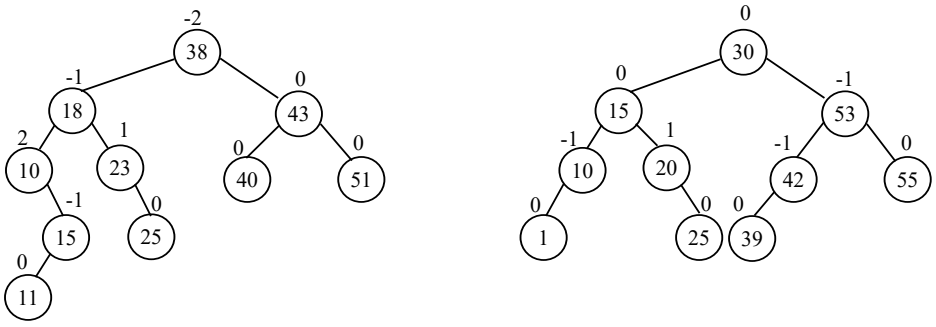


Figura 13.3: Arborele din stânga are unele noduri care nu sunt balansate, deci nu e AVL. Arborele din dreapta este AVL. Deasupra fiecărui nod este indicat factorul de balansare al acestuia.

În figura 13.3 sunt prezentați doi arbori binari de căutare împreună cu factorii de balansare asociați fiecărui nod. Arborele din stânga nu are toate nodurile balansate, deci nu este AVL, iar cel din dreapta este AVL.

Observație: În cazul implementării acestor arbori, în fiecare nod x trebuie reținută o valoare suplimentară. Aceasta ar putea fi factorul de balansare, dar este destul de elaborată recalcularea acestui factor, în urma operațiilor de balansare. De aceea se recomandă stocarea în fiecare nod x a înălțimii subarborelui de rădăcină x. Calculul recursiv al înălțimii, pentru fiecare nod, la operațiile de rebalansare este complet ineficient. De aceea, recalcularea înălțimii după operații de inserție, respectiv ștergere, trebuie realizată odată cu operația de rebalansare, de sus în jos, pe baza înălțimii subarborilor nodului curent.

Operații pe un AVL Principalele operații pe un arbore AVL sunt: căutarea, inserția și ștergerea. Căutarea se realizează în același mod ca într-un arbore binar de căutare obișnuit. Algoritmi pentru această operație pot fi găsiți în bibliografia recomandată.

Inserția unei noi chei într-un arbore AVL se realizează în primă instanță în același mod ca într-un arbore binar de căutare. Doar că, în urma inserției, este posibil ca arborele să nu mai fie balansat. Astfel că, trebuie efectuată operația de rebalansare.

Rebalansarea arborelui AVL se realizează pornind de la nodul nou inserat, care are factorul de balansare 0 (este frunză), urcând în arbore către rădăcină.

La fiecare pas se consideră nodul curent x și se recalculează factorul de balansare al acestuia.

- Dacă factorul nou al lui x devine 0, s-a echilibrat arborele și algoritmul de oprește.
- Dacă factorul nou al lui x devine -1 sau 1, atunci a crescut subarborele stâng, respectiv drept, deci se va modifica și factorul părintelui lui x și se continuă urcarea în arbore.
- Dacă factorul nou al lui x devine -2 atunci înseamnă că a crescut prea mult subarborele stâng al lui x și mă uit la fiul stâng $y = x.left$:
  - **Caz I** dacă $factor(y) = -1$ atunci se efectuează o rotație la dreapta în jurul lui x, se recalculează factorii lui x și y și STOP.
  - **Caz II** dacă $factor(y) = 1$ atunci se efectuează o rotație la stânga în jurul lui y, o rotație la dreapta în jurul lui x, se recalculează factorii nodurilor implicate în rotații și STOP.
- Dacă factorul nou al lui x devine 2 atunci înseamnă că a crescut prea mult subarborele drept al lui x și mă uit la fiul stâng $y = x.right$:
  - **Caz I** dacă $factor(y) = 1$ atunci se efectuează o rotație la stânga în jurul lui x, se recalculează factorii lui x și y și STOP.
  - **Caz II** dacă $factor(y) = -1$ atunci se efectuează o rotație la dreapta în jurul lui y, o rotație la stânga în jurul lui x, se recalculează factorii nodurilor implicate în rotații și STOP.

Ștergerea unui nod z într-un arbore AVL se realizează în primă instanță în același mod ca într-un arbore binar de căutare. Adică, dacă z are 0 sau 1 fiu y, va urca y (dacă z nu are fii atunci urcă $y = nil$ în locul lui z, iar dacă z are 2 fii, se identifică y succesorul lui z, care va urca în locul lui z.

În urma ștergerii, este posibil ca arborele să nu mai fie balansat. Astfel că, trebuie efectuată operația de rebalansare.

Rebalansarea arborelui AVL în cazul ștergerii se realizează pornind de la părintele lui z, dacă z a avut 0 sau 1 fiu, respectiv de la părintele nodului y, acela dinainte ca y să fi urcat în locul lui z, urcând către rădăcină și recalculând la fiecare pas factorul de balansare al nodului curent x.

- Dacă factorul nou al lui x devine -1 sau 1, înseamnă că factorul vechi al lui x a fost 0, s-a scurtat unul dintre subarborii lui x, dar nu s-a modificat per total înălțimea subarborelui de rădăcină x, deci nu mai are sens continuarea urcării și algoritmul de oprește.
- Dacă factorul nou al lui x devine 0, s-a redus înălțimea subarborelui de rădăcină x, deci se va modifica și factorul părintelui lui x și deci se continuă urcarea în arbore.
- Dacă factorul nou al lui x devine -2, atunci înseamnă că a scăzut prea mult subarborele drept al lui x. Ne uităm la fiul stâng $y = x.left$, care este prea înalt:
  - **Caz I** dacă $factor(y) = -1$ atunci se efectuează o rotație la dreapta în jurul lui x, se recalculează factorii lui x și y și se continuă urcând la părintele lui x.
  - **Caz II** dacă $factor(y) = 1$ atunci se efectuează o rotație la stânga în jurul lui y, o rotație la dreapta în jurul lui x, se recalculează factorii nodurilor implicate în rotații și se continuă urcând la părintele lui x.
  - **Caz III** dacă $factor(y) = 0$ atunci se efectuează o rotație la dreapta în jurul lui x, se recalculează factorii nodurilor implicate în rotații și STOP.
- Dacă factorul nou al lui x devine 2 atunci înseamnă că a scăzut prea mult subarborele stâng al lui x și ne uităm la fiul drept $y = x.right$, care este prea înalt:
  - **Caz I** dacă $factor(y) = 1$ atunci se efectuează o rotație la stânga în jurul lui x, se recalculează factorii lui x și y și se continuă urcând la părintele lui x.
  - **Caz II** dacă $factor(y) = -1$ atunci se efectuează o rotație la dreapta în jurul lui y, o rotație la stânga în jurul lui x, se recalculează factorii nodurilor implicate în rotații și se continuă urcând la părintele lui x.
  - **Caz III** dacă $factor(y) = 0$ atunci se efectuează o rotație la stânga în jurul lui x, se recalculează factorii nodurilor implicate în rotații și STOP.

Explicații detaliate pot fi găsite în [8], iar exemple vor fi prezentate în cadrul problemelor teoretice rezolvate.

13.1.3 Arbori Roșu-Negru

Arborii roșu-negru, dezvoltați în 1987 de Leonidas J. Guibas și Robert Sedgewick [21], reprezintă un alt tip de arbori binari de căutare auto-echilibrați. Spre deosebire de arborii AVL, care își mențin echilibrul prin factorul de balansare al nodurilor, arborii roșu-negru (ARN) atribuie fiecărui nod o proprietate denumită *culoare* și respectă o serie de proprietăți specifice ale arborelui. Aceste proprietăți garantează o anumită echilibrare, asigurând astfel că operațiile asupra arborelui se efectuează cu o complexitate logaritmică relativ la numărul de chei stocate.

Proprietăți

Arborii binari de căutare de tip ARN se caracterizează prin următoarele proprietăți, care asigură balansarea relativă a acestora:

- Fiecare nod x are un câmp $x.color$, care poate fi roșu sau negru.
- Rădăcina este întotdeauna neagră.
- Frunzele sunt negre și nil.
- Orice nod roșu are doi fii negri (eventual frunze nil).
- Pe orice drum de la un nod x către orice frunză nil se întâlnesc același număr de noduri negre. Nu se numără x, dar se numără frunza. Acest număr de noduri negre se numește **înălțime neagră** (*black height*) și se va nota în continuare cu $bh(x)$.

În figura 13.4 este prezentat un ARN cu înălțimea neagră $bh = 2$.

Observații:

- În cazul implementării ARN este ineficient din punct de vedere al memoriei, să se creeze atâtea noduri nil, câte frunze există. Din acest motiv se folosește de obicei un nod santinelă unic, de culoare neagră, care nu conține informație.
- Niciun drum de la rădăcină la o frunză, nu poate fi mai lung decât cel mult dublul lungimii altui drum de la rădăcină la altă frunză.
- În continuare, pentru a simplifica desenele, nu vom mai reprezenta frunzele nil, dar nu trebuie uitat de existența acestora.

Complexitatea de timp a operațiilor principale pe un ARN este logaritmică în numărul de chei n stocate în arbore. Acest lucru se demonstrează în bibliografia recomandată [3].

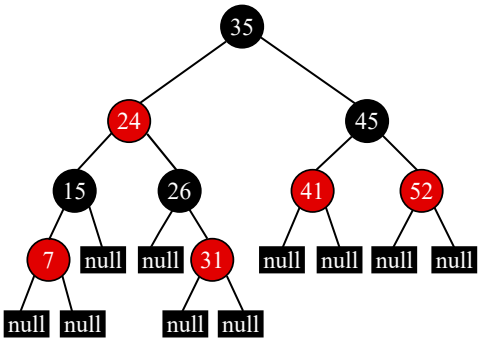


Figura 13.4: Exemplu de arbore roșu-negru.

Operații pe un ARN Principalele operații pe un ARN sunt aceleași ca pe un AVL: căutare, inserție și ștergere a unui element. Căutarea se realizează în același mod ca în cazul arborilor binari de căutare obișnuiți.

Inserția unei chei noi se realizează inițial în același mod ca în cazul arborilor binari de căutare obișnuiți, prin coborârea în arbore până la poziția potrivită de inserție și adăugarea noii chei într-un nod nou, care are **culoarea roșie** și doi fii frunze (negre și nil). Prin inserarea noului nod pot fi stricate proprietățile arborelui și este necesară o procedură de refacere.

Refacerea proprietăților ARN.

Singurele două proprietăți care se pot strica în urma inserției sunt:

(1) Culoarea rădăcinii să fie roșie.

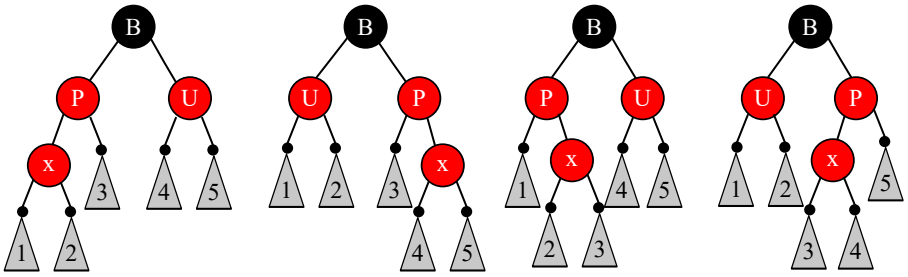
(2) Un nod roșu să aibă ca părinte un alt nod roșu.

Situiația (1) se rezolvă pur și simplu prin recolorarea rădăcinii. Situația (2) presupune urcarea în arbore de la noul nod inserat către rădăcină atâta timp cât părintele P al nodului curent x are culoarea roșie (x este și el roșu) și tratarea a 3 cazuri descrise mai jos.

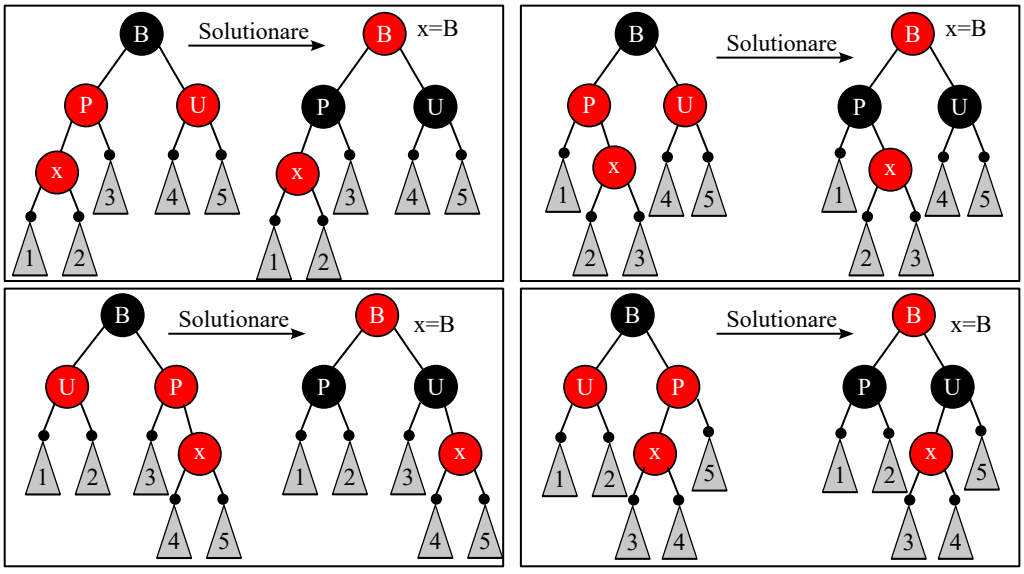
Se notează cu B părintele nodului P, adică bunicul nodului curent x. Dat fiind că P se presupune că are culoarea roșie, sigur nu este rădăcina arborelui, deci B nu este nil și B mai are un fiu în afară de P (eventual nil), notat cu U.

Cazurile de refacere a proprietăților ARN la inserție depind de culoarea lui U.

1. **Cazul 1**: unchiul U al nodului x are culoarea roșie:



Rezolvare: se colorează P și U cu negru și B cu roșu și se reia refacerea de la nodul curent $x = B$. Adică se urcă în arbore, pentru că s-ar putea, ca B să aibă un părinte roșu, ceea ce ar strica proprietățile arborelui în acea poziție.

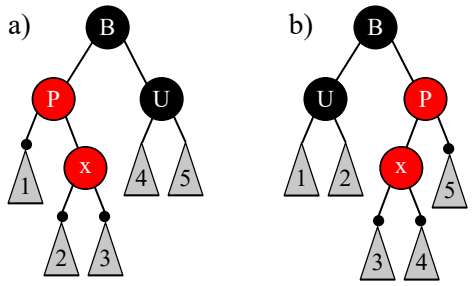


2. **Cazul II**: unchiul U al nodului x are culoarea neagă și

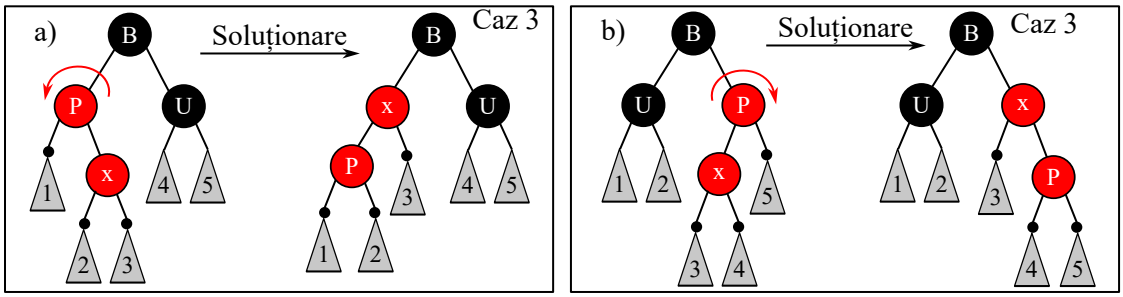
   (a) P e pe stânga lui B, iar x e pe dreapta lui P

SAU

(b) P e pe dreapta lui B, iar x e pe stânga lui P.



Rezolvare: se efectuează o rotație în jurul lui P, la stânga în cazul (a), respectiv la dreapta în cazul (b) și se ajunge în cazul 3.

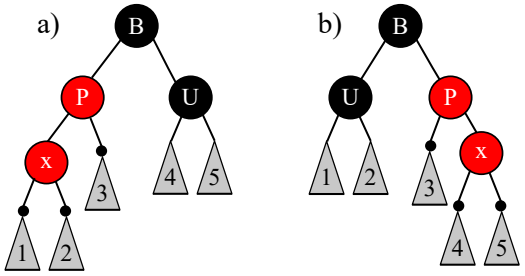


3. **Cazul 3:** unchiul U al nodului x are culoarea neagră și

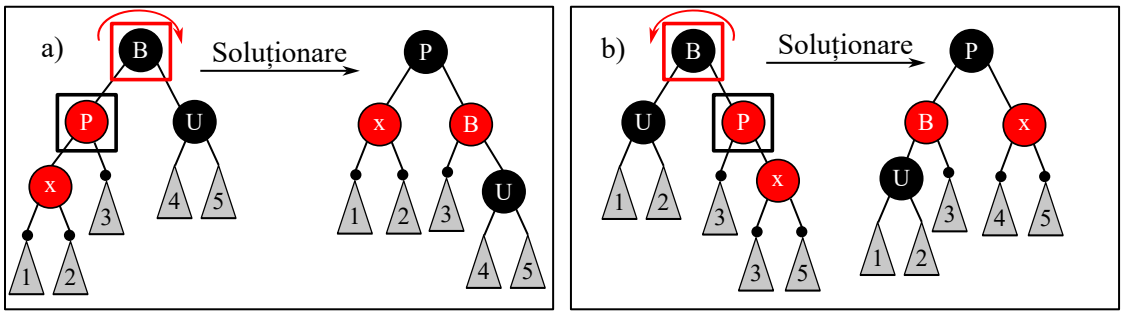
   (a) P e pe stânga lui B, iar x e tot pe stânga lui P

   SAU

   (b) P e pe dreapta lui B, iar x e pe dreapta lui P.



Rezolvare: se colorează P cu negru, se colorează B cu roșu și se efectuează o rotație în jurul lui B, la dreapta în cazul (a), respectiv la stânga în cazul (b) și se oprește algoritmul de refacere.



Ștergerea unui nod z dintr-un ARN se realizează în prima fază în același mod, ca dintr-un arbore binar de căutare. Apoi se aplică un algoritm de refacere a proprietăților ARN, care este descris pe larg în curs [8], precum și în [3]

.

Indicații de implementare Pentru implementarea unui ARN se poate folosi o structură (sau o clasă) de tipul:

```
template<typename T>
struct ARN {
    struct node {
        T key;
        char color;
        node* left, * right, * parent;
        //constructor pentru node
        node(T k, char col, node* l, node*r, node*p) :  key(k),
           color(col), left(l), right(r), parent(p) {
        }
    };

    node* root, * nil;
    ARN() {
        //initializarea nodului santinela
        nil = new node(0, 'n', nil, nil, nil);
        //nodul radacina este initializat cu santinela
        root = nil;
    }
    void insert(T val);
    void insert_repare(node* x);
    node* search(T val);
    void erase(node* z);
    void erase_repare(node* x);
    bool empty();
    void clear();
    node* minimum(node* x); // cauta nodul cu cheia minima
    node* maximum(node* x); // cauta nodul cu cheia maxima
```

```
    node* successor(node* x); // cauta nodul succesor al lui x
    node* predecessor(node* x); // cauta nodul predecesor al lui x
    void RSD(node* x); // parcurgerea recursiva RSD de la nodul x
    void SRD(node* x); // parcurgerea recursiva SRD de la nodul x
    // etc .
};
```

Atenție: în tot codul, în loc de *nullptr* trebuie să se folosească santinela *nil*. Vom da un exemplu pentru implementarea funcției de inserare în arbore a unei chei noi. Funcția `insert_repare` rămâne ca exercițiu de implementare pe baza pseudocodului din literatură [3].

```
template<typename T>
void ARN<T>::insert(T val)
{
    //observam ca se vor initializa fii si parintele noului nod
    //cu santinela
    node* z = new node(val, 'r', nil, nil, nil);
    node* x = root;
    node* y = nil;

    while (x != nil && x->key !=val) //se compara cu santinela nu
       cu nullptr
    {
        y = x;
        if (z->key < x->key)
            x = x->left;
        else
            x = x->right;
    }
    if (x!=nil)
    {
        std::cout << "Key already present";
        return;
    }
    z->parent = y;
    if (y == nil) //se compara cu santinela nu cu nullptr
        root = z;
    else
    {
        if (val < y->key) y->left = z;
        else y->right = z;
    }
    insert_repare(z);
}
```

13.2 Arbori echilibrați în STL

Arborii binari de căutare echilibrați stau la baza implementării claselor STL `set`, `map`, `multiset` și `multimap`. Diferențele între aceste clase sunt similare ca în cazul `unordered_set`, `unordered_map`, `unordered_multiset` și `unordered_multimap`. Adică `set` și `multiset` stochează doar chei, pe când `map` și `multimap` stochează perechi de tip (cheie, valoare). În cazul `set` și `map` cheile sunt unice, pe când în cazul `multiset` și `multimap` sunt permise chei duplicat.

Inserarea și căutarea se realizează în complexitate logaritmică, după cheie. Iterarea se face în inordine, obținându-se astfel elementele sortate după cheie.

Clasele STL sunt:

```
    template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class set;
```

```
   template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T>>
> class map;
```

```
   template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class multiset;
```

```
   template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T>>
> class multimap;
```

Componentele din template sunt:

Key - indică tipul de date al cheii stocate;

T - în cazul `map` și `multimap` indică tipul valorii asociate cheii;

Compare - indică comparatorul cu care se compară două chei. Se observă că în mod predefinit este folosit `std::less<Key>`, care corespunde unui container în care elementele sunt păstrate crescător după cheie. Dacă se va înlocui acest comparator cu `std::greater<Key>` elementele vor fi păstrate descrescător după cheie. Această situație corespunde arborilor binari de căutare, în care elementele din subarborele stâng al unui nod x au cheile mai mari decât cheia lui x, iar cei din subarborele drept au cheile mai mici decât cheia lui x.

Dacă tipul de date al cheii nu dispune de comparator sau se dorește o funcționare diferită a acestuia, atunci comparatorul trebuie definit în mod explicit.

Allocator - se ocupă de gestionarea memoriei.

Comparator

În cazul obiectelor de tip `set`, `map`, `multiset` și `multimap` comparatorul este necesar elementelor pentru a le aranja conform regulilor unui arboe binar de căutare. La capitolul legat de cozi de prioritate, s-au discutat comparatorii STL predefiniți `std::less` și `std::greater`, pecum și `operator()`, definit in cadrul unui comparator.

Dacă dorim să definim un comparator propriu, pentru tipuri de date, pentru care comparația nu este definită sau dacă vrem să modificăm comportamentul față de cel predefinit pentru tipul de dat al cheii, este necesară declararea unei structuri/clase de tip comparator, în care să fie definit operatorul `operator()`.

Exemplu: Presupunem că dorim o structură de tip `set`, care stochează elemente de tip dreptunghi. Considerăm dreptunghiurile definite printr-o structură, care conține 2 elemente de tip `std::pair<float, float>`, reprezentând coordonatele stânga-sus și dreapta-jos ale dreptunghiului. De asemenea structura dispune de o funcție ce calculează aria dreptunghiului. În structura `set` vrem ca ordonarea să se facă după aria dreptunghiului.

Definim structura dreptunghi:

```
struct dreptunghi {
    std::pair<float, float> stanga_sus, dreapta_jos;
    //constructor
    dreptunghi(float x0, float y0, float x1, float y1):
        stanga_sus({x0,y0}),dreapta_jos({x1,y1}){ }
```

```
    //metoda este const, pentru ca nu modifica membrii clasei
    float arie() const
    {
        return (dreapta_jos.first - stanga_sus.first) *
           (dreapta_jos.second - dreapta_jos.first);
    }
};
```

Iar acum definim comparatorul:

```
class compare {
public:
    bool operator ()(const dreptunghi& d1, const dreptunghi& d2)
       const {

        float arie1 = d1.arie();
        float arie2 = d2.arie();
        if (arie1 == arie2)
        {
            return std::make_pair(d1.stanga_sus, d1.dreapta_jos) <
               std::make_pair(d2.stanga_sus, d2.dreapta_jos);
        }
        return arie1 < arie2;
    }
};
```

Observații:

- În primul rând, pentru eficientizare parametrii au fost transmiși prin referință. Se pot face aceleași observații legate de declarațiile de tip `const`, ca în cazul comparatorilor pentru clasa `priority_queue`.
- În al doilea rând, se observă că în comparator nu am luat în considerare doar ariile celor două dreptunghiuri, ci și coordonatele colțurilor din stânga-sus și din dreapta-jos. Dacă am fi comparat doar ariile, atunci pentru două dreptunghiuri diferite (cu coordonate diferite), dar cu aceeași arie, structura ar fi considerat că este vorba despre același dreptunghi, iar la inserare ar fi fost adăugat doar unul dintre aceste dreptunghiuri. Pentru a trata și cazul dreptunghiurilor cu aceeași arie, se definește o ordonare legată de coordonatele celor două colțuri. STL știe să compare elemente de tip `std::pair` în ordine lexicografică.

În funcția `main` vom declara o variabilă de tip `set`, care stochează dreptunghiuri, ordonate după arie, în modul următor:

```
std::set<dreptunghi, compare> dreptunghiuri;
```

În continuare vom exemplifica cu precădere clasa `map`. Lucrurile sunt similare pentru clasa `set` și detalii pot fi găsite în documentația online specifică.

Declarare și inițializare

Pentru utilizarea unui `map` este necesar să se includă header-ul corespunzător cu comanda:

```
#include<map>
```

Declararea unui obiect de tip `map` se face specificând tipul cheii și tipul valorii care vor fi stocate în perechile din `map` și, dacă este nevoie, a comparatorului adecvat. La declarare obiectul poate fi inițializat folosind liste de inițializare sau pe baza altor structuri. Prezentăm câteva exemple în continuare.

Mapă vidă:

```
map<std::pair<std::string, std::pair<int, int>> myMap;
```

Mapă inițializată pe bază de listă de inițializare:

```
map<std::string, int> myMap1 = { {"Mirela", 10}, {"Adrian",
   7}, {"Cornel", 8} };
```

Mapă inițializată pe baza unei alte mape (folosind `myMap` definit mai sus). Astfel, se realizează o copie pe baza constructorului de copiere:

```
map<std::string, int> myMap2(myMap1);
```

Mapă inițializată pe baza unui vector de perechi:

```
std::vector<std::pair<std::string, int>> vec={{"Ana", 8},
   {"Mihai", 5}, {"Claudia", 10}};
map<std::string, int> myMap3(vec.begin(), vec.end());
```

Operații pe `map`

Inserarea. Pentru a adăuga elemente noi se poate proceda în două moduri:

(1) Folosind funcția `insert`

```
unordered_map<std::string, std::pair<int, int>> myMap;
myMap.insert({"Bucuresti", {0,0}});
myMap.insert({"Brasov", {49, 25}});
```

(2) Folosind operatorul de acces [ ]

```
    myMap["Iasi"] = {47, 27};
```

Observație: Dacă obiectul myMap conține cheia "Brasov" și apelăm încă o dată:

```
myMap.insert({"Brasov", {5,5}});
```

instrucțiunea nu va avea niciun efect, deoarece nu este permisă existența de chei duplicat. În schimb, dacă există cheia "Brasov", dar folosim instrucțiunea:

```
myMap.["Brasov"] = {48, 26};
```

se va accesa perechea cu cheia și se va actualiza valoarea asociată.

Căutarea unei chei se poate realiza folosind `find`. Dacă cheia există, `find` va returna un iterator către perechea cu cheia respectivă, iar în caz contrar va returna iteratorul `end()` al obiectului respectiv.

Exemplu:

```
#include <iostream>
#include<fstream>
#include<map>

int main() {
    std::ifstream fisier("Elevi2.txt");
    std::map <std::string, int> elevi;
    std::string nume;
    int nota;

    while (fisier >> nume >> nota)
    {
        elevi[nume] = nota;
    }
    auto it = elevi.find("Maria");
    if (it != elevi.end()) std::cout << "Eleva Maria are nota:" <<
       it->second << "\n";
    else std::cout << "Nu am gasit eleva Maria\n";
    return 0;
}
```

Parcurgerea unui obiect de tip `map` se poate realiza, la fel ca în cazul obiectelor de tip `unordered_map`, cu ajutorul iteratorilor sau al operatorului :.

Ștergerea unui element anume poate fi efectuată pe baza cheii sau folosind un iterator care indică elementul respectiv. De-asemenea, se poate realiza ștergerea tuturor elementelor

începând de la un iterator $it1$ și până la un al doilea iterator $it2$ (exclusiv elementul indicat de $it2$).

Exemplu:

```
#include <iostream>
#include<fstream>
#include<map>

int main() {
    std::ifstream fisier("Elevi3.txt");
    std::map <std::string, int> elevi;
    std::string nume;
    int nota;

    while (fisier >> nume >> nota)
        elevi[nume] = nota;
    elevi.erase("Ion"); //Sterge elementul cu cheia "Ion", daca
       exista
    auto it = elevi.find("Maria");
    if (it != elevi.end())
        elevi.erase(it); //Stergerea elementului indicat de
           iterator
    for (const auto& el : elevi)
        std::cout << el.first << " " << el.second<<"\n";
    auto it2 = elevi.begin();
    //incrementam it2
    for (int i = 0; i < 3 && it2 != elevi.end(); i++, it2++);
        elevi.erase(elevi.begin(), it2); //stergerea primelor 3
           elemente
                                          //din map
   for (const auto& el : elevi)
        std::cout << el.first << " " << el.second << "\n";
    return 0;
}
```

Alternativ, pentru incrementarea iteratorului se poate folosi funcția `advance` din `algorithms`:

```
    advance(it2, 3);
```

Atenție: trebuie avut grijă, ca iteratorul să nu avanseze mai mult decât ultimul element al structurii, altfel se va produce eroare de execuție.

Funcțiile `lower_bound` și `upper_bound`

Datorită faptului că elementele unui `map` sunt păstrate ordonate după cheie, există posibilitatea de a determina în mod eficient, cel mai mic element, mai mare decât o anumită cheie

dată. Acest lucru poate fi efectuat folosind funcțiile `lower_bound` și `upper_bound` în modul următor:

- `lower_bound(k)`: returnează un iterator către primul element din ordinea de iterare a map-ului a cărui cheie nu este ordonată înaintea lui k conform comparatorului. În cazul comparatorului `std::less` asta ar însemna, elementul cu cea mai mică cheie, mai mare sau egală cu k.
- `upper_bound(k)`: returnează un iterator către primul element din ordinea de iterare pentru care k este ordonat strict înaintea sa (indiferent dacă k există sau nu în `map`). În cazul comparatorului `std::less` asta ar însemna, elementul cu cea mai mică cheie, mai mare decât k.

Dacă cheia k nu există în `map`, atunci `lower_bound(k)` și `upper_bound(k)` returnează același iterator.

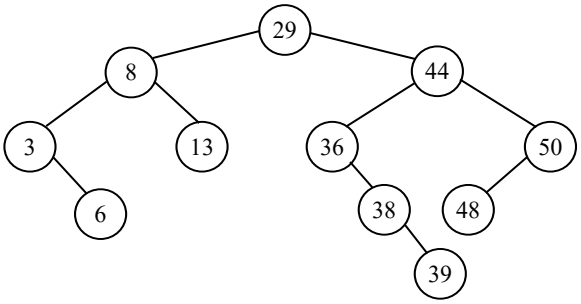
Dacă nu există nicio cheie $\geq k$, atunci ambele funcții returnează iteratorul `end()` al obiectului.

13.3 Probleme rezolvate

13.3.1 Probleme teoretice

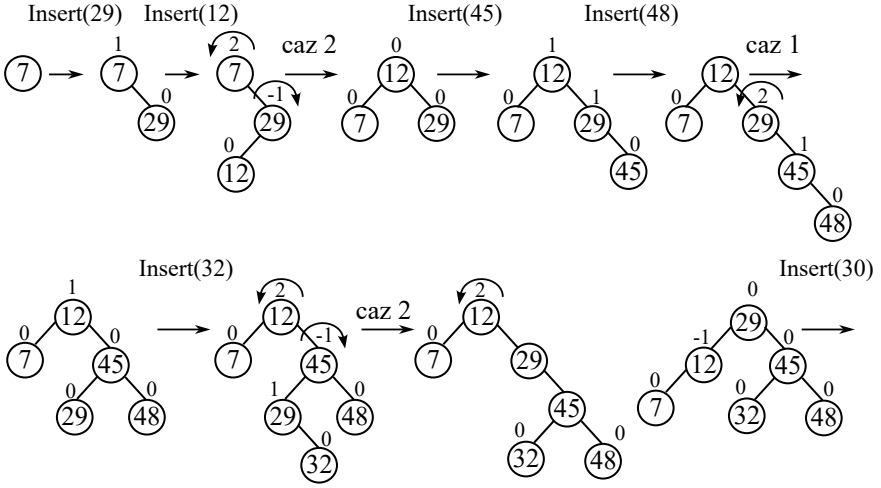
1. Să se deseneze arborele binar de căutare pentru care se cunoaște parcurgerea în preordine: **RSD** = 29, 8, 3, 6, 13, 44, 36, 38, 39, 50, 48.

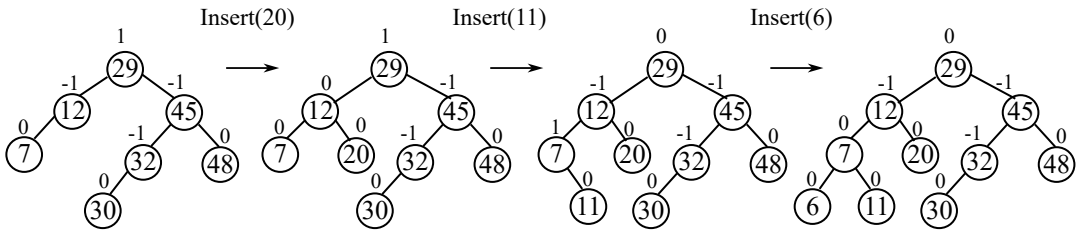
   Rezolvare: Această problemă poate fi rezolvată în 2 moduri. Una dintre soluții se bazează pe faptul că, într-un arbore binar de căutare, parcurgerea în inordine a acestuia obține cheile în ordine sortată. Astfel putem avea cele 2 parcurgeri necesare pentru refacerea arborelui conform algoritmului discutat în capitolul de arbori. Totuși exisă o abordare mult mai simplă și anume, dacă pur și simplu inserăm cheile în ordinea dată de parcurgerea RSD în arbore, se obține arborele căutat, în cazul nostru:



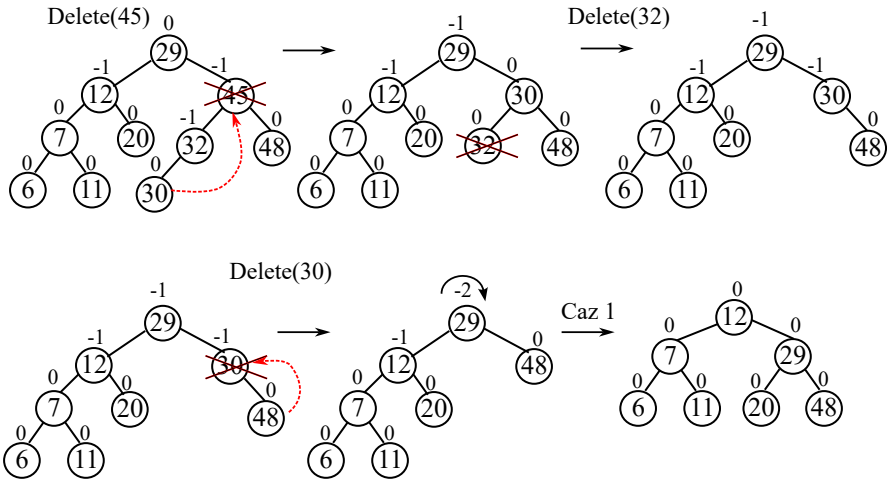
2. Să se insereze într-un AVL inițial vid cheile: 7, 29, 12, 45, 48, 32, 30, 20, 11, 6 și din arborele rezultat să se șteargă pe rând cheile 45, 32, 30. Cum arată arborele rezultat?

Rezolvare: Inserția cheilor este prezentată în figurile următoare:



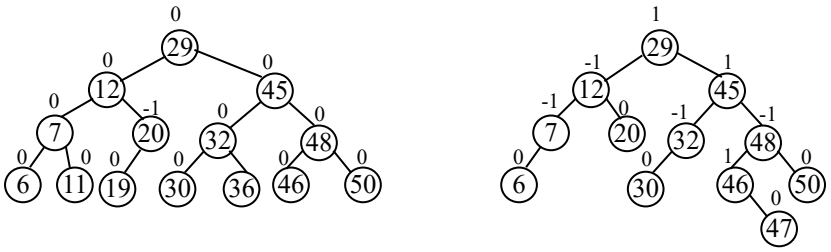


Ștergerea cheilor este prezentată în următoarele figuri. Atunci când nodul ce trebuie șters are doi fii, se va determina succesorul acestuia, cu care apoi se va înlocui.



3. Un AVL de înălțime h are întotdeauna mai puține noduri decât un AVL de înălțime $h+1$? Argumentați răspunsul dat.

Rezolvare Răspunsul este NU și se poate justifica printr-un contra-exemplu. În figura de mai jos sunt ilustrați doi arbori AVL, unul cu înălțimea 3 și unul cu înălțimea 4. Arborele de înălțime 3 (stânga) are 14 noduri, iar cel de înălțime 4 (dreapta) are 12 noduri.



4. Care este numărul minim și care numărul maxim de chei care pot fi stocate într-un ARN cu înălțimea neagră $bh = 4$?

Rezolvare:

(1) Numărul minim de noduri într-un ARN de înălțime neagră bh se obține atunci când arborele conține doar noduri negre. În acest caz fiecare nivel este complet. Dar pe ultimul nivel există doar frunze nil. Astfel nivelurile de la 0 la $bh-1$ sunt pline, iar numărul de chei este

$$n_{min} = 1 + 2 + \ldots + 2^{bh-1} = 2^{bh-1+1} - 1 = 2^{bh} - 1$$

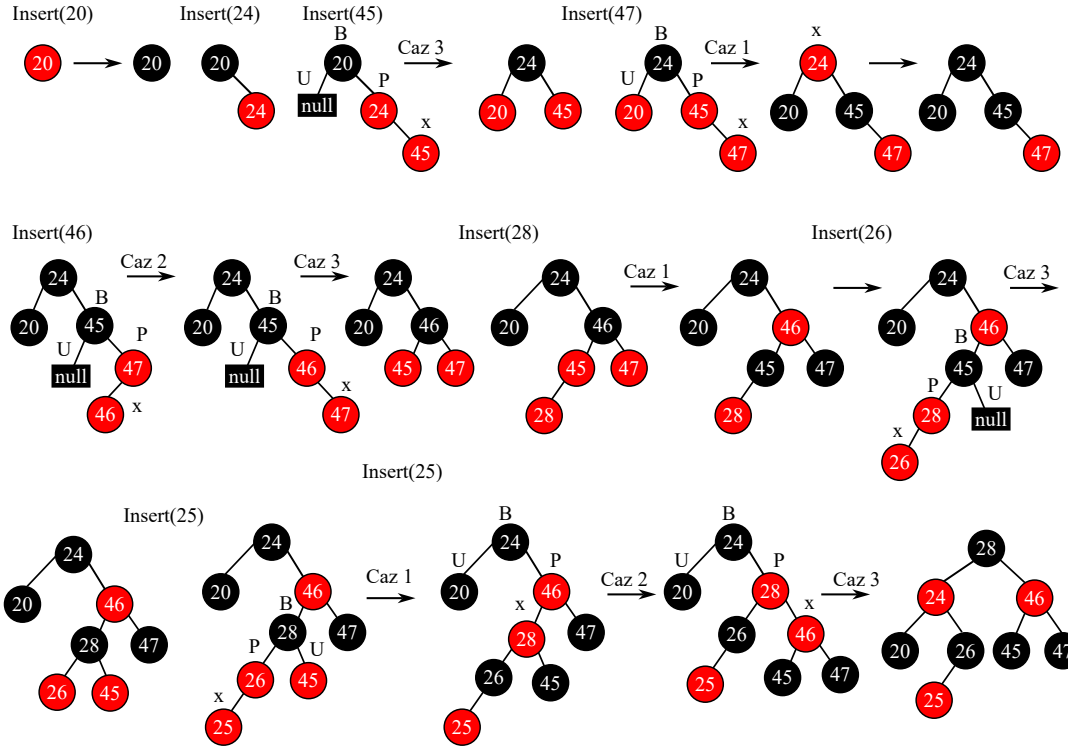
(2) Numărul maxim de chei se obține atunci când avem înălțime maximă, adică $h = 2 \cdot bh$ și fiecare nivel este plin. Să nu uităm că pe ultimul nivel nu avem chei, deci numărul maxim de chei va fi:

$$n_{max} = 1 + 2 + \ldots + 2^{h-1} = 2^{h-1+1} - 1 = 2^h - 1 = 2^{2 \cdot bh} - 1$$

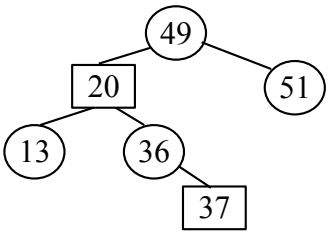
5. Inserați într-un ARN inițial vid cheile: 20, 24, 45, 47, 46, 28, 26, 25.

Rezolvare:

Inserția nodurilor este prezentată în figurile următoare.



6. Se consideră ARN-ul din figură. Care este înălțimea neagră bh a acestuia și care este înălțimea? Să se insereze în acest ARN cheile: 40, 42, 43, 41. Nodurile roșii sunt reprezentate prin dreptunghiuri.

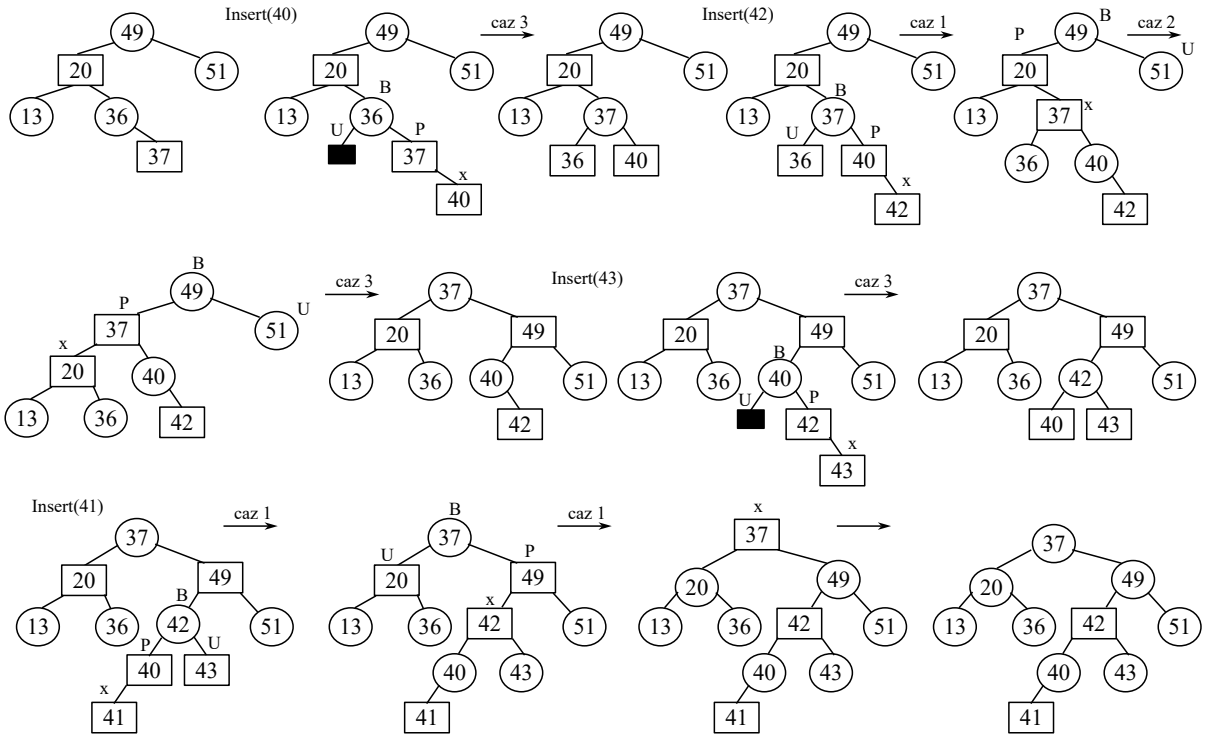


.

Rezolvare:

Înălțimea neagră este $bh = 2$ și înălțimea este $h = 4$ (atenție frunzele negre și nil nu se văd în desen, dar ele sunt acolo!)

Inserția cheilor este ilustrată în figurile următoare:

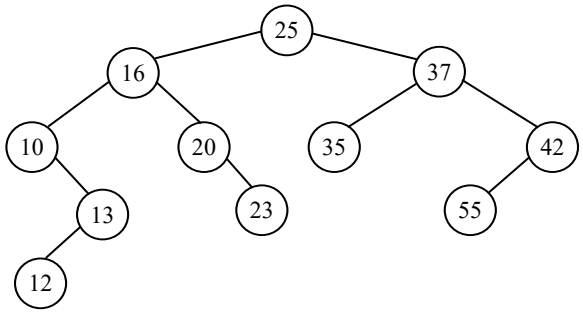


13.3.2 Probleme aplicative

1. Să se scrie o funcție care are ca parametri 2 noduri dintr-un arbore binar de căutare și care returnează primul strămoș comun al celor două noduri (adică acel strămoș, care nu mai are în subarborele determinat de el nici un alt strămoș comun al celor două noduri). Care este

complexitatea algoritmului? **Observație**: dacă nodul x este strămoș al nodului y, atunci primul strămoș comun al celor două este părintele lui x, iar dacă x este rădăcina, atunci nu există un strămoș comun.

Exemplu: Considerăm arborele binar de căutare din figură:



Cel mai apropiat strămoș comun al nodurilor 12 și 23 este nodul 16.

Cel mai apropiat strămoș comun al nodurilor 10 și 16 este nodul 25.

Nodurile 10 și 25 nu au strămoș comun, pentru că 25 este rădăcina și nu are strămoș.

Rezolvare:

Date de intrare: un arbore binar de căutare, pointeri la cele 2 noduri (existente!) $nod1$, respectiv $nod2$. Alternativ se pot da și cheile celor două noduri (presupunând că sunt chei existente în arbore).

Date de ieșire: pointer la nodul care este strămoșul comun.

Algoritm de rezolvare: Se pornește cu un nod $curent$ din rădăcina arborelui și se compară la fiecare moment $curent.key$ cu $nod1.key$ și $nod2.key$. **Observație:** Vom considera $nod1.key < nod2.key$.

- dacă $curent.key < nod1.key$ și $curent.key < nod2.cheie$, atunci este evident că atât $nod1$ cât și $nod2$ se vor afla în subarborele drept al lui $curent$, deci și $curent.right$ este strămoș comun al celor două noduri (dacă nu este chiar unul dintre noduri) și se continuă cu $curent \leftarrow curent.right$.
- dacă $curent.key > nod1.key$ și $curent.key > nod2.key$, atunci atât $nod1$ cât și $nod2$ se vor afla în subarborele stâng al lui $curent$, deci și $curent.left$ este strămoș

comun al celor două noduri (dacă nu este chiar unul dintre noduri) și se continuă cu $curent \leftarrow curent.left$.

- Dacă $curent.key > nod1.key$ și $curent.key < nod2.key$ atunci $nod1$ va fi în subarborele stâng, iar $nod2$ în subarborele drept al lui $curent$. Astfel $curent$ este strămoș comun (și anume cel mai apropiat) al celor 2 noduri.
- dacă $curent.key = nod1.key$ sau $curent.key = nod2.key$, depinde de interpretarea pe care o asociem ideii de prim strămoș comun. Putem considera $curent$ ca fiind strămoșul comun sau putem urca la $curent.parent$. Vom considera a doua variantă.

Algoritm în C++. Vom construi funcția `nod* StramosComun(node* nod1, node*nod2)` ca membră a clasei `BinTree` definită în acest capitol.

```
template<typename T>
BinTree<T>::node* BinTree<T>::StramosComun(node* nod1, node*
   nod2){
    if (!root || !nod1 || !nod2) return nullptr;
    //ne asiguram de faptul ca in nod2 este nodul cu cheia
       mai mare
    if(nod1->key > nod2->key){
        node*temp = nod1;
        nod1 = nod2;
        nod2 = temp;
    }
    node* stramos = root;

    //se va iesi din bucla la gasirea stramosului comun
    //sau la nulltpr, cand nu se mai poate cobora in arbore
    while (true){
        //iesire cu insucces, daca unul dintre noduri nu
           exista
        if (stramos == nullptr) return nullptr;
        //daca unul dintre noduri este parintele celuilat
        //se considera parintele acestuia ca stramos comun
        if (stramos == nod1 || stramos == nod2) return
           stramos->parent;

        //conditia care se indeplineste in cazul gasirii
           stramosului comun
        if (stramos->key < nod1->key && stramos->key >
           nod2->key)
            return stramos;
```

```
        //coborare in arbore
        if (stramos->key < nod1->key && stramos->key <
           nod2->key)
            stramos = stramos->right;
        else
            stramos = stramos->left;
    }

    return stramos;
}
```

Justificarea soluției propuse: Pentru ca un nod $stramos$ să fie strămoș comun pentru două noduri $nod1$ și $nod2$ avem 2 variante:

- $nod1$ și $nod2$ se află în același subarbore al lui $stramos$. Dacă ori $nod1$ ori $nod2$ este fiu al lui $stramos$, atunci $stramos$ este strămoșul comun. Altfel strămoșul comun este în acel subarbore.
- $node1$ și $node2$ se află în subarbori diferiți, atunci $stramos$ este evident strămoșul comun.

Aceste situații sunt tratate în algoritmul de mai sus. În plus sunt tratate și situațiile, în care nodurile nu există în arbore, sau unul dintre noduri este rădăcina și atunci evident nu există strămoș comun.

Analiza complexității: Algoritmul presupune pornirea de la rădăcină și coborârea în arbore către un strămoș comun.

- În cel mai favorabil caz, chiar rădăcina este strămoș comun, deci complexitatea este constantă.
- În cel mai defavorabil caz trebuie să coborâm până la o frunză, deci complexitatea este $O(h)$, unde h este înălțimea arborelui. Dacă arborele este echilibrat, obținem $O(\log_2 n)$, n este numărul de noduri din arbore.

2. Se citesc dintr-un fișier n numere complexe, date prin partea reală și partea imaginară. Acestea se vor stoca într-un container de tip `std::set`, ordonate după modulul lor.

   (a) Să se verifice dacă un număr complex citit de la tastatură se află în set.

(b) Să se afișeze primele k numere complexe cu modulul mai mic decât p. k, p - citite de la tastatură.

Rezolvare:

Date de intrare: n numere complexe, reprezentate printr-o structură/clasă cu două câmpuri: re (partea reală) și im (partea imaginară). În plus:

(a) un număr complex citit de la tastatură;

(b) o variabilă p de tip real și una k de tip întreg.

Date de ieșire:

(a) $true$ sau $false$, dacă un numărul complex citit de la tastatură se află, respectiv nu se află în `std::set`;

(b) primele k numere complexe cu modulul mai mic decât p.

Observații:

Întâi de toate vom defini structura/ respectiv clasa $complex$, cu cele două câmpuri. Clasa va dispune de:

- un constructor
- o funcție de calcul a modulului

Cum folosim un obiect de tip `std::set` care stochează numere complexe și dorim să fie ordonate după modul, este necesară definirea unui comparator *custom*.

Algoritmul în C++:

Vom prezenta două rezolvări, dintre care una folosește o structură **complex**, cu funcțiile definite în interiorul acesteia, iar a doua o clasă **complex**, în care vom supraîncărca și operatorii pentru citirea/scrierea unui număr complex.

Varianta cu struct:

```
#include<iostream>
#include<fstream>
#include<set>
```

```
struct complex {
    int re, im;
    complex(int a = 0, int b = 0) : re(a), im(b) {}
    double modul() const {
        return sqrt(re * re + im * im);
    }
    void print() const {
        std::cout << re << " + ";
        if (im < 0)
            std::cout << "(" << im << ")i";
        else if (im > 0)
            std::cout << im << "i";
    }
    void read(complex& z){
        std::cout<<"re="; std::cin>>z.re;
        std::cout<<"im="; std::cin>>z.im;
    }
};

class compare {
public:
    bool operator()(const complex& z1, const complex& z2)
       const
    {
        if(z1.modul()==z2.modul())
            if(z1.re==z2.re)
                return z1.im<z2.im;
            else
                retrun z1.re<z2.re;
        return z1.modul() < z2.modul();
    }
};

int main()
{
    int k;
    double p;
    std::set<complex, compare> numere_complexe;
    std::ifstream fisier("Complexe.txt");

    int re, im;
    while (fisier >> re >> im)
    {
        numere_complexe.emplace(re, im);
    }
    fisier.close();
    complex z;
```

```
    std::cout << "numarul complex:\n";
    std::cout << "re="; std::cin >> z.re;
    std::cout << "im="; std::cin >> z.im;

    if (numere_complexe.find(z) != numere_complexe.end())
        std::cout << "Am gasit z\n";
    else std::cout << "Nu am gasit z\n";

    std::cout << "Cate numere afisam? k =";
    std::cin >> k;
    std::cout << "Care este modulul maxim? p =";
    std::cin >> p;
    if (k<0 || k>numere_complexe.size())
        std::cout << "k nu este un numar potrivit\n";
    else {
        auto it = numere_complexe.begin();
        int i = 0;
        for (auto it=numere_complexe.begin(); it !=
           numere_complexe.end() && i<k && it->modul()<p;
           i++, it++) {
            it->print();
            std::cout << "\n";
        }
    }
    return 0;
}
```

Varianta folosind o clasă:

```
#include<iostream>
#include<fstream>
#include<set>

class complex {
private:
    int re, im;
public:
    complex(int a = 0, int b = 0) : re(a), im(b) {}
    //supraincarcarea operatorilor pentru citire si scriere
    friend std::ostream& operator<<(std::ostream& os, const
       complex& c);
    friend std::istream& operator>>(std::istream& is,
       complex& c);

    double modul() const;
    int getReal() const { return re; }
    int getImaginary() const { return im; }
};
```

```
//operatorul pentru scriere
std::ostream& operator<<(std::ostream& os, const complex& c) {
    os << c.re;
    if (c.im >= 0)
        os << " + " << c.im << "i";
    else
        os << " - " << (-c.im) << "i";
    return os;
}
//operatorul pentru citire
std::istream& operator>>(std::istream& is, complex& c) {
    std::cout << "re=";
    is >> c.re;
    std::cout << "im=";
    is >> c.im;

    return is;
}
double complex::modul() const {
    return sqrt(re * re + im * im);
}

//comparatorul
class compare {
public:
    bool operator()(const complex& z1, const complex& z2)
       const {
        double m1 = z1.modul();
        double m2 = z2.modul();
        if (m1 == m2)
            if (z1.getReal() == z2.getReal())
                return z1.getImaginary() < z2.getImaginary();
            else return z1.getReal() < z2.getReal();

        return m1 < m2;
    }
};

int main()
{
    std::set<complex, compare> numere_complexe;
    std::ifstream fisier("Complexe.txt");

    int re, im;
    while (fisier >> re >> im)
       numere_complexe.emplace(re, im);
```

```
    fisier.close();
    complex z;
    std::cout << "numarul complex:\n";
    //citim folosind noul operator de citire definit pentru
       clasa complex
    std::cin >> z;

    if (numere_complexe.find(z) != numere_complexe.end())
        std::cout << "Am gasit z\n";
    else std::cout << "Nu am gasit z\n";
    int k;
    double p;
    std::cout << "Cate numere afisam? k =";
    std::cin >> k;
    std::cout << "Care este modulul maxim? p =";
    std::cin >> p;

    if (k<0 || k>numere_complexe.size())
        std::cout << "k nu este un numar potrivit\n";
    else {
        auto it = numere_complexe.begin();
        int i = 0;
        for (auto it = numere_complexe.begin(); it !=
           numere_complexe.end() && i < k && it->modul() <=
           p; i++, it++) {
            //afisare folosind noul operator definit pentru
               clasa complex
            //cum iteratorii sunt pointeri, trebuie sa
               folosim operatorul de dereferentiere
            std::cout << (*it);
            std::cout << "\n";
        }
    }
    return 0;
}
```

Comparatorul custom: Observăm faptul că, deși dorim ca inserția și căutarea să se facă în funcție de modulul numărului complex, în comparator totuși am luat în considerare și câmpurile real și imaginar. Dacă nu am face acest lucru, ci ne-am limita la compararea modulelor, atunci pentru două numere complexe diferite, dar cu același modul, doar unul dintre ele ar fi inserat. Structura ar considera că, având același modul, au aceeași cheie. Nu ar ști, cum să le compare altfel.

Analiza complexității: Operațiile principale într-un `set/map`, respectiv căutarea, inserția și ștergerea au complexitate logaritmică. Astfel:

(a) Căutarea unui număr complex se realizează în $O(\log_2 n)$.

(b) Obținerea iteratorului `begin()` se realizează în $O(\log_2 n)$. Apoi parcurgerea a cel mult k elemente din `set` pornind de la $begin$, are loc prin incrementarea iteratorului. Acest lucru se face prin apelul succesorului binar la fiecare pas, operație realizată tot în $O(\log_2 n)$. Deci per total avem complexitatea $O(k \log_2 n)$.

3. Se citesc dintr-un fișier n produse: nume produs și preț. Acestea se vor stoca într-un `std::map<float, std::vector<std::string»`.

(a) Să se afișeze toate produsele cu prețul p.

(b) Să se afișeze toate produsele care au prețul în intervalul $[a, b]$.

Rezolvare

Date de intrare: fișierul cu produsele în care pe fiecare linie avem denumirea produsului și prețul acestuia.

Date de ieșire:

(a) Produsele, fiecare cu prețul p.

(b) Toate produsele care au prețul în intervalul $[a, b]$.

Algoritm în C++: Se va utiliza un obiect `std::map<float, std::vector<std::string»` care va avea drept cheie prețul, iar pe câmpul valoare un vector cu toate produsele ce au prețul respectiv.

```
#include<iostream>
#include<fstream>
#include<map>
#include<vector>

//upper_bound =  elementul cu cea mai mica cheie mai mare
   decat o cheie data k
//lower_boun = elementul cu cea mai mica cheie mai mare sau
   egala cu o cheie data k
int main()
{
    std::map<float, std::vector<std::string>> produse;
```

```
    std::ifstream file("Produse.txt");
    float pret;
    std::string prod;
    while (file >> prod >> pret)
    {
        produse[pret].push_back(prod);
    }
    file.close()
    std::cout << "pret=";
    std::cin >> pret;
    auto it = produse.find(pret);
    if (it == produse.end()) //daca nu am gasit produse cu
       acest pret
    {
        std::cout << "nu avem produse cu pretul " << pret <<
           "\n";
    }
    else
    {
        std::cout << "Cu pretul " << pret << " avem
           produsele:\n";
        for (auto produs:it->second)
        {
            std::cout << produs << ", ";
        }
        std::cout << "\n";
    }
    int a, b;
    std::cout << "dati intervalul de pret:\n";
    std::cout << "a=";
    std::cin >> a;
    std::cout << "b=";
    std::cin >> b;
    auto it2 = produse.lower_bound(a);
    for (; it2!=produse.end() && it2->first<=b; it2++)
    {
        std::cout << "cu pretul " << it2->first << ":\n";
        for (auto produs : it2->second)
        {
            std::cout << produs << ", ";
        }
        std::cout << "\n";
    }

    return 0;
}
```

Analiza complexității: Operațiile principale într-un `set`/`map`, respectiv căutarea, inserția și ștergerea au complexitate logaritmică. Astfel:

(a) Căutarea unui anumit preț se realizează în $O(\log_2 n)$.

(b) Obținerea unui iterator folosind $lower_bound$ se realizează în $O(\log_2 n)$. Apoi parcurgerea a cel mult $k = b - a + 1$ elemente din `map` pornind de la iteratorul $lower_bound$, are loc prin incrementarea iteratorului. Acest lucru se face prin apelul succesorului binar la fiecare pas, operație realizată tot în $O(\log_2 n)$. Deci per total avem complexitatea $O(k \log_2 n)$.

13.4 Probleme propuse

13.4.1 Probleme teoretice

1. Considerăm un arbore binar de căutare cu 22 de noduri. Care este înălțimea maximă a unui astfel de arbore și care este înălțimea minimă. Precizați în ce caz se obține înălțimea maximă.

2. Desenați arborele binar de căutare pentru care se cunoaște parcurgerea în postordine:
   SDR: 1, 9, 8, 7, 10, 45, 44, 46, 52, 41, 20
   Să se insereze aceste chei, considerate în ordinea dată, într-un AVL.

3. Presupunând că avem într-un arbore binar de căutare numere între 1 și 1000 și se caută nodul cu valoarea 363, care dintre următoarele secvențe NU ar putea fi secvențe de noduri parcurse în timpul căutării? Justificați pentru fiecare secvență răspunsul dat.

   a. 2, 252, 401, 398, 330, 344, 397, 363.

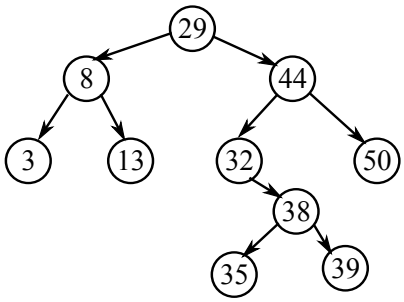
   b. 924, 220, 911, 244, 898, 258, 362, 363.

   c. 925, 202, 911, 240, 912, 245, 363.

   d. 2, 399, 387, 219, 266, 382, 381, 278, 363.

   e. 935, 278, 347, 621, 299, 392, 358, 363.

4. Se consideră arborele binar din figură:



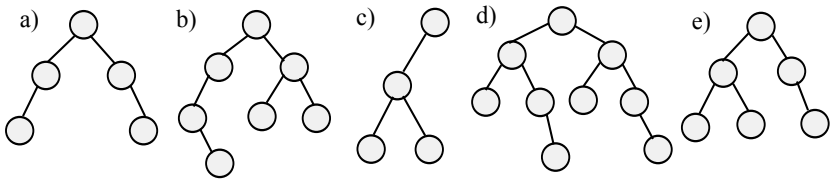
a. Justificați de ce acesta este un arbore binar de căutare.

b. Inserați cheile 12, 7, 9 și ștergeți cheia 29.

c. Dacă se adaugă fiecărui nod factorul de balansare, este arborele obținut la (b) un arbore AVL valid?

5. Dați exemplu de un arbore AVL cu chei numere naturale, care are factorul de balansare al rădăcinii egal cu -1, înălțimea cel puțin 2 și toate cheile mai mici decât 20. În acest arbore inserați cheile 21, 25, 23, 24, 31.

6. Care dintre următoarele configurații poate apărea într-un AVL valid? Justificați răspunsul.



7. Dați exemplu de un ARN cu exact 4 noduri negre interne (deci în afară de frunzele nil) cu cheile mai mari ca 30. În acest arbore inserați cheile 0, 4, 3, 2, 7. Desenați arborele final și precizați care este înălțimea neagră a acestuia.

8. Dați exemplu de un ARN cu înălțimea neagră $bh = 2$, cu exact 3 noduri roșii și cu cheile mai mici ca 15. În acest arbore inserați cheile 20, 24, 23, 19, 18. Desenați arborele final și scrieți parcurgerea în postordine a acestuia.

9. Care este raportul minim și care este raportul maxim între numărul de noduri roșii și cel de noduri negre (interne) dintr-un arbore roșu-negru cu înălțimea neagră $bh = 5$? Justificați răspunsul dat.

13.4.2 Probleme aplicative

*

1. Pornind de la codul prezentat în cadrul secțiunii de teorie și de la pseudocodul din bibliografie [8], scrieți o clasă $BSTree$, pentru un arbore binar de căutare, cu toate funcțiile aferente:
   - INSERT(int key) - inserează un nou nod în arbore cu cheia key. Dacă cheia deja există, nu se va insera.
   - MAXIMUM($node * x$) / MINIMUM($node * x$) - returnează un pointer la nodul cu cheia maximă / minimă din subarborele de rădăcină x.
   - SUCCESSOR($node*x$) / PREDECESSOR($node*x$) - returnează un pointer la nodul care este succesorul / predecesorul nodului x sau $nullptr$.
   - FIND(int key) - returnează un pointer la nodul cu cheia key dacă există sau $nullptr$ altfel.
   - DELETE(int key) - șterge din arbore nodul cu cheia key dacă exist.
   - ERASE($node * x$) - șterge din arbore nodul x (care a fost mai întâi identificat prin FIND).
   - PRINT(int opt) - afișează cheile din arbore în parcurgerea în preordine (dacă $opt = 1$), inordine (dacă $opt = 2$), în postordine (dacă $opt = 3$), pe niveluri (dacă $opt = 4$).
   - CONSTRUCT construiește un arbore binar de căutare pornind de la un vector de chei.
   - EMPTY() - verifică dacă arborele este vid.
   - CLEAR() - șterge toate nodurile din arbore.
2. Scrieți o funcție care pentru un nod dat x dintr-un arbore binar (nu neapărat de căutare), să determine numărul de chei mai mici decât cheia lui x și să afișeze aceste chei în ordine crescătoare. Utilizați structuri suplimentare, dacă este necesar (de exemplu `map`).
3. Scrieți o funcție care să verifice dacă un arbore binar este arbore de căutare.

**

1. Se dau un arbore binar de căutare echilibrat și două valori a și b. Scrieți o funcție care să afișeze în complexitate $O(k+log_2 n)$ toate cheile din arbore aflate în intervalul $[a, b]$, unde k este numărul de astfel de chei și n este numărul total de chei din arbore.

2. Scrieți un program, care să folosească un map pentru a reține un dicționar de cuvinte (de tip $string$). Definiți un comparator custom, care atunci când compară două cuvinte, să nu țină cont, dacă literele din cuvânt sunt litere mici sau majuscule. Citiți dintr-un fișier un text și stocați în dicționar toate cuvintele. Permiteți apoi căutarea unui anumit cuvânt, adăugarea de cuvinte sau afișarea cuvintelor dintr-un anumit interval, dat prin două prefixe. Pentru asta puteți folosi funcțiile `upper_bound`, respectiv `lower_bound` de care dispune `map`.
   Exemplu: Se cere afișarea tuturor cuvintelor în intervalul ["ana", "bac"].

3. Scrieți un program care citește diferite texte (din unul sau mai multe fișiere) și stochează un dicționar, care pentru fiecare cuvânt reține indicele tuturor documentelor în care apare. Permiteți afișarea după o anumită literă de început, sau afișarea în ordine alfabetică.
   Exemplu: Se poate cere afișarea tuturor cuvintelor care încep cu 'm', și pentru acestea indicii documentelor în care apar, sau numărul de documente în care apar (în funcție de dorința utilizatorului).

***

1. Pornind de la codul pentru un arbore binar de căutare simplu, implementați un arbore AVL cu chei numere întregi. Definiți o structură $node$ care dispune de un câmp key pentru informație, un câmp $hieght$ pentru înălțime și câmpuri de tip pointer pentru fiii stâng ($left$) și drept ($right$) și pentru părinte ($parent$). De asemenea structura / clasa $node$ trebuie să dispună de un constructor care inițializează câmpul key cu o valoare transmisă prin parametru, câmpul $height$ cu 1 și câmpurile de tip pointer cu $nullptr$. Definiți în plus structura / clasa AVL care dispune de un membru de tip pointer la $node$, numit $root$ și de funcțiile:

   - INSERT(int key) - inserează un nou nod în arbore cu cheia key. Dacă cheia deja există, nu se va insera.

- INSERT_REPARE(node* x) - reface balansare după inserție (se apelează de către INSERT) pornind de la nodul x.
- MAXIMUM($node * x$) / MINIMUM($node * x$)- returnează un pointer la nodul cu cheia maximă / minimă din subarborele de rădăcină x.
- SUCCESSOR($node*x$) / PREDECESSOR($node*x$) - returnează un pointer la nodul care este succesorul / predecesorul nodului x sau $nullptr$.
- FIND(int key) - returnează un pointer la nodul cu cheia key dacă exist sau $nullptr$ altfel.
- DELETE(int key) - șterge din arbore nodul cu cheia key dacă există.
- ERASE($node * x$) - șterge din arbore nodul x (care a fost mai întâi identificat prin FIND).
- DELETE_REPARE($node*x$) - reface balansarea arborelui după ștergere pornind de la nodul x.
- ROTATE_LEFT(node* x), ROTATE_RIGHT(node *x) - funcțiile de rotație în jurul nodului x. Se recomandă și folosirea unor funcții de rotație dublă, care să creeze direct legăturile după două rotații succesive (vezi cazurile de refacere a balansării), astfel încât să se evite apelul de două ori a unei rotații simple.
- PRINT_TREE(int opt) - afișează cheile din arbore în parcurgerea în preordine (dacă $opt = 1$), inordine (dacă $opt = 2$), în postordine (dacă opt=3), pe niveluri (dacă $opt = 4$). Trebuie afișat și factorul de balansare pentru fiecare nod.
- CLEAR() - șterge toate nodurile din arbore.
- EMPTY() - verifică dacă arborele este vid.
- CONSTRUCT - construiește un arbore AVL pornind de la un vector de chei.

2. Scrieți o funcție care să aibă ca parametri două valori întregi a și b și care să construiască în timp liniar un arbore binar de căutare echilibrat (frunzele să se afle pe cel mult 2 niveluri consecutive) care să conțină valorile întregi din intervalul $[a, b]$. **Indicație de rezolvare**: folosiți un algoritm de tip *Divide et Impera*.

3. Scrieți o funcție nerecursivă care să determine înălțimea unui (sub-)arbore binar, pornind de la nodul rădăcină.

4. Scrieți o funcție care pentru un arbore binar, în care fiecare nod are atașată o culoare (roșu sau negru), verifică dacă acest arbore respectă toate proprietățile unui ARN (nu se verifică frunzele nil, condiție care se presupune că este respectată).

5. Să se implementeze un program care citește dintr-un fișier o serie de obiective turistice și dispune de o funcție $Select$, care pentru un anumit oraș afișează în ordine alfabetică, obiectivele disponibile. Pentru fiecare locație din fișier se cunosc: orașul, denumirea locației, prețul de acces per persoană. Funcția $Select$ peria ca parametru numele orașului și un preț maxim și afișează în ordine crescătoare a prețului (care respectă limita de preț dată în funcție) obiectivele turistice din acel oraș. **Indicație:** folosiți mai multe structuri de date adecvate din biblioteca STL.

6. Scrieți un program care citește dintr-un fișier numele și mediile unor studenți și le memorează în ordinea notelor. Pentru cei cu aceeași notă se realizează ordonarea după nume. Permiteți afișarea ordonată a tuturor studenților cu o anumită medie. Media este de tip număr real, cu ce cel mult 2 zecimale. Se vor considera studenți cu aceeași medie, dacă diferă doar prin a 2 zecimală. De asemenea se cere implementarea unei funcții, care afișează toți studenții cu mediile într-un anumit interval. Folosiți STL.

****

Implementați structura map similar cu cea din STL folosind în spate un arbore Roșu-Negru. Elementele stocate în structură vor fi de tip pereche (cheie, valoare). Structura trebuie să dispună în plus față de funcțiile specifice unui ARN de:

- operatorul = implementat pentru NOD (+ constructor de copiere)
- operatorul [ ] - prin care să poată fi accesat un element cu o anumită cheie /respectiv inserat în cazul în care nu există (de ex: MyMap m; m[3]=7 - inserează elementul cu cheia 3 și valoarea 7.)
- Implementarea unui iterator cu care să se poată itera prin structură.
- Funcțiile EMPTY( ), CLEAR() și SIZE( )

În plus vă asigurați de existența tuturor operatorilor necesari în structură. Structura sa fie template (pe tipul de date și pe comparator).

Partea V

Paradigme algoritmice avansate

Capitolul 14

Greedy

14.1 Aspecte teoretice

Algoritmii Greedy, cunoscuți și sub denumirea de algoritmi lacomi, constituie o paradigmă fundamentală și eficientă în domeniul proiectării algoritmilor de optimizare. Această abordare se bazează pe o strategie euristică, unde, la fiecare etapă a procesului decizional, se selectează opțiunea care pare a fi cea mai avantajoasă la nivel local, fără a anticipa consecințele pe termen lung ale acestei alegeri. Premisa subiacentă este că o serie de decizii optime la nivel local va converge către o soluție optimă la nivel global.

Principiile fundamentale ale acestei metode de programare sunt:

- Alegerea locală optimă - La fiecare pas, algoritmul face cea mai bună alegere posibilă la momentul respectiv, fără a lua în considerare deciziile viitoare sau impactul acestora. Această decizie este ireversibilă.
- Subprobleme optime - O soluție optimă pentru problema originală conține soluții optime pentru subproblemele sale. Această proprietate este esențială pentru validitatea abordării Greedy.

Spre deosebire de alte tehnici algoritmice precum *Divide et Impera*, care descompune problema în subprobleme independente, rezolvându-le recursiv și combinând rezultatele, sau Programarea Dinamică, care rezolvă subprobleme suprapuse, stocând rezultatele intermediare pentru a evita recalcularea, algoritmii Greedy se disting prin caracterul lor nerevizuibil. Odată luată o decizie, aceasta nu mai este modificată ulterior, ceea ce conferă acestor algoritmi o simplitate conceptuală și o eficiență computațională remarcabilă, adesea traducându-se într-o complexitate temporală redusă.

Această secțiune va explora conceptele fundamentale ale tehnicii Greedy, structura generală a algoritmilor lacomi și principiile teoretice care asigură corectitudinea lor.

14.1.1 Fundamente teoretice detaliate

Mulțime de candidați și soluție

În cadrul unui algoritm Greedy, problema este definită printr-o mulțime de candidați din care se va construi soluția. Acești candidați reprezintă elementele de bază pe care algoritmul le poate selecta pentru a construi progresiv soluția finală. O soluție posibilă reprezintă o combinație completă de candidați care satisface toate constrângerile impuse de problemă, iar o soluție fezabilă este o selecție parțială de candidați care poate fi extinsă ulterior pentru a forma o soluție completă. Înțelegerea acestor concepte este esențială pentru a putea defini corect criteriile de selecție și funcția obiectiv.

- **Mulțime de candidați** - toate elementele din care se poate construi soluția.
- **Soluție posibilă** - o selecție completă de candidați care satisface toate constrângerile problemei.
- **Soluție fezabilă** - o selecție parțială de candidați care poate fi extinsă pentru a forma o soluție posibilă.

Criteriu de selecție și funcția obiectiv

Criteriul de selecție determină alegerea „celui mai promițător” candidat la fiecare pas. Practic, acesta este motorul decizional al algoritmului Greedy și trebuie definit astfel încât fiecare alegere locală să conducă către soluția globală optimă. Funcția obiectiv, pe de altă parte, cuantifică calitatea soluției obținute și poate fi orientată spre maximizare sau minimizare. Corelarea corectă dintre criteriul de selecție și funcția obiectiv este ceea ce garantează succesul unui algoritm Greedy.

- **Criteriu de selecție** - o funcție care determină care candidat din mulțimea disponibilă este „cel mai promițător” la pasul curent.
- **Funcție obiectiv** - funcția care trebuie maximizată sau minimizată, cum ar fi suma valorilor, costul total sau timpul necesar.

14.1.2 Algoritmul general Greedy explicat

Structura generală a unui algoritm Greedy presupune parcurgerea mulțimii de candidați într-o ordine determinată de criteriul de selecție, testarea fezabilității fiecărui candidat și includerea lui în soluție doar dacă respectă constrângerile problemei. Alegerea făcută devine permanentă și contribuie direct la construirea soluției finale.

```
template<typename T>
std::vector<T> algoritmGreedy(std::vector<T> candidati) {
    std::vector<T> solutie;

    std::sort(candidati.begin(), candidati.end(),
       criteriuSelectie);

    for (auto candidat : candidati) {
        if (esteFezabil(solutie, candidat)) {
            solutie.push_back(candidat);
        }
        if (esteSolutie(solutie)) {
            break;
        }
    }

    return solutie;
}
```

În acest cod, sortarea candidaților conform criteriului Greedy asigură că elementele cele mai promițătoare sunt evaluate primele. Funcția `esteFezabil` verifică dacă includerea candidatului nu încalcă constrângerile problemei, iar funcția `esteSolutie` verifică dacă s-a atins deja o soluție completă, permițând terminarea buclei înainte de epuizarea tuturor candidaților.

14.1.3 Verificarea corectitudinii unui algoritm Greedy

Validitatea și aplicabilitatea unui algoritm Greedy pentru a obține o soluție optimă globală depind de satisfacerea a două proprietăți fundamentale ale problemei. Demonstrarea corectitudinii unui algoritm Greedy implică, prin urmare, verificarea existenței acestor proprietăți.

1. Proprietatea substructurii optime

   Această proprietate stipulează că o soluție optimă a problemei inițiale trebuie să conțină, la rândul său, soluții optime pentru subproblemele sale. Altfel spus, dacă o soluție optimă

globală este descompusă în componente, fiecare componentă trebuie să reprezinte o soluție optimă pentru subproblema corespunzătoare.

O problemă are proprietatea substructurii optime dacă, pentru o soluție optimă $S = (s_1, s_2, \ldots, s_k)$, subsoluția $S' = (s_2, \ldots, s_k)$ corespunzătoare unei subprobleme constituie o soluție optimă pentru acea subproblemă.

Cel mai adesea, demonstrația se realizează prin *reducere la absurd*. Se presupune că S' nu este o soluție optimă pentru subproblema sa. Aceasta ar implica existența unei alte soluții S'' pentru subproblemă, care este mai bună. Prin înlocuirea lui S' cu S'' în soluția inițială S, s-ar obține o soluție $S_{\text{nou}} = (s_1, S'')$ care ar fi mai bună decât S, contrazicând astfel ipoteza inițială că S era optimă.

2. Proprietatea alegerii Greedy

   Această proprietate afirmă că o soluție optimă globală poate fi atinsă prin efectuarea unei alegeri optime la nivel local, la fiecare pas, fără a fi necesară reconsiderarea deciziilor anterioare. Mai mult, componentele unei soluții optime fie au fost alese direct prin criteriul Greedy, fie pot fi substituite cu elemente selectate conform acestui criteriu fără a compromite optimalitatea soluției.

   O problemă are proprietatea alegerii Greedy dacă soluția optimă a problemei fie este construită direct printr-o strategie Greedy, fie poate fi transformată într-o altă soluție optimă care a fost construită pe baza strategiei Greedy.

   Demonstrația implică adesea a se arăta că, dacă se ia prima decizie conform strategiei Greedy, se ajunge la o subproblemă a cărei soluție optimă, combinată cu acea primă decizie, conduce la o soluție optimă globală. Acest lucru se realizează, de obicei, considerându-se o soluție optimă arbitrară care nu urmează prima alegere Greedy. Se demonstrează apoi că se poate modifica această soluție optimă pentru a include prima alegere Greedy, fără a-i diminua calitatea (fără a o face mai puțin eficientă). Acest proces se poate repeta, deseori prin inducție matematică sau prin utilizarea proprietății substructurii optime, pentru a arăta că întreaga soluție poate fi construită Greedy.

Doar atunci când ambele proprietăți sunt demonstrate pentru o anumită problemă, se poate garanta că un algoritm Greedy va produce o soluție optimă. În absența uneia dintre ele, algoritmul Greedy poate oferi doar o soluție euristică, nu neapărat optimă.

14.2 Algoritmi clasici

14.2.1 Problema spectacolelor

Problema spectacolelor este un exemplu clasic de problemă de optimizare rezolvată prin tehnica Greedy. Se dau n activități, fiecare definită printr-un interval de timp $[s_i, t_i]$, unde s_i este ora de start și t_i este ora de finalizare ($s_i < t_i$). Obiectivul este de a selecta cât mai multe activități care nu se suprapun, adică nu există două activități alese astfel încât intervalele lor să se intersecteze.

Pentru această problemă, strategia Greedy optimă constă în alegerea activităților care se termină cât mai devreme. Această alegere asigură că rămâne cât mai mult timp disponibil pentru activitățile următoare, crescând șansa de a include cât mai multe activități.

Pașii algoritmului sunt următorii:

1. Sortarea activităților în funcție de timpul de finalizare t_i.
2. Inițializarea unei soluții vide și a unui marker care să rețină ora de finalizare a ultimei activități selectate (`ultimFinish`).
3. Parcurgerea activităților sortate. Pentru fiecare activitate, dacă ora sa de start s_i este mai mare sau egală cu `ultimFinish`, activitatea este compatibilă și se adaugă la soluție.
4. Actualizarea `ultimFinish` la ora de finalizare a activității tocmai selectate.

Această metodă asigură selectarea numărului maxim de activități fără suprapuneri, demonstrând proprietatea de substructură optimă și alegerea Greedy.

Implementare în C++

```
struct Spectacol {
    int start, finish;
};

bool comparareFinish(Spectacol a, Spectacol b) {
    return a.finish < b.finish;
}
```

```
std::vector<Spectacol> planificaSpectacole(std::vector<Spectacol>
   spectacole) {
    std::sort(spectacole.begin(), spectacole.end(),
       comparareFinish);

    std::vector<Spectacol> solutie;
    int ultimFinish = 0;

    for (auto spectacol : spectacole) {
        if (spectacol.start >= ultimFinish) {
            solutie.push_back(spectacol);
            ultimFinish = spectacol.finish;
        }
    }
     return solutie;
}
```

Explicația codului

Structura `Spectacol` reține intervalul fiecărei activități. Funcția `comparareFinish` permite sortarea activităților după timpul de finalizare, ceea ce este esențial pentru strategia Greedy. În bucla principală, se verifică compatibilitatea fiecărei activități cu soluția parțială, iar `ultimFinish` se actualizează pentru a reflecta ora de finalizare a ultimei activități selectate.

Complexitatea acestui algoritm este dominată de sortare, adică $O(n \log n)$, iar selecția efectivă a activităților se face în timp $O(n)$. Algoritmul exemplifică clar puterea deciziilor Greedy atunci când problema are proprietatea substructurii optime și alegerea locală duce la soluția globală optimă.

14.2.2 Problema rucsacului

Problema rucsacului este o problemă clasică de optimizare în care se dorește maximizarea valorii totale a obiectelor plasate într-un rucsac cu capacitate limitată W. Fiecare obiect are o greutate w_i și o valoare v_i, iar spre deosebire de versiunea discretă, aici este permis să luăm fracțiuni din obiecte.

Cheia soluției Greedy constă în alegerea obiectelor cu cel mai mare raport valoare/greutate $\frac{v_i}{w_i}$ la fiecare pas. Se consideră că obiectele care oferă mai multă valoare pe unitate de greutate sunt cele mai „profitabile” pentru umplerea rucsacului.

Algoritmul urmează pașii:

1. Calcularea raportului $\frac{v_i}{w_i}$ pentru fiecare obiect.

2. Sortarea obiectelor în ordine descrescătoare a acestui raport.

3. Adăugarea obiectelor în rucsac în această ordine până când capacitatea este complet utilizată. Dacă obiectul curent nu încape complet, se adaugă doar fracțiunea care se potrivește

Această strategie Greedy garantează soluția optimă pentru problema rucsacului fracționar deoarece problema posedă ambele proprietăți esențiale pentru corectitudinea unui algoritm Greedy:

1. Proprietatea alegerii greedy
   Putem demonstra că o alegere Greedy (selectarea obiectului cu cel mai mare raport $\frac{v_i}{w_i}$) conduce întotdeauna la o soluție optimă. Să presupunem, prin reducere la absurd, că există o soluție optimă S_{opt} care nu include, sau nu include în totalitate, obiectul o_1 cu cel mai mare raport valoare/greutate.

   - Dacă S_{opt} nu include deloc o_1, sau include o cantitate mai mică din o_1 decât ar permite capacitatea rucsacului, înseamnă că S_{opt} trebuie să includă o parte dintr-un alt obiect o_j cu un raport $\frac{v_j}{w_j}$ mai mic decât $\frac{v_1}{w_1}$.
   - Putem atunci să înlocuim o cantitate mică de greutate din o_j cu o cantitate egală de greutate din o_1. Deoarece $\frac{v_1}{w_1} > \frac{v_j}{w_j}$, valoarea totală a rucsacului ar crește, contrazicând ipoteza că S_{opt} era soluția optimă. Prin urmare, prima alegere Greedy este întotdeauna parte a unei soluții optime.

2. Proprietatea substructurii optime
   Odată ce am făcut prima alegere Greedy (am adăugat obiectul o_1 în rucsac, fie total, fie parțial), problema se reduce la o subproblemă similară: umplerea capacității rămase a rucsacului cu obiectele rămase (sau cu fracțiunea rămasă din o_1, dacă nu a fost adăugat complet).

   - Dacă soluția optimă pentru problema originală include obiectul o_1 (sau o fracțiune din el), atunci restul soluției optime trebuie să fie o soluție optimă pentru subproblema formată din capacitatea rămasă și obiectele rămase.

- Dacă nu ar fi așa, am putea înlocui soluția suboptimă pentru subproblemă cu una optimă, obținând o soluție mai bună pentru problema originală, ceea ce ar contrazice optimalitatea soluției inițiale.

Astfel, deoarece problema rucsacului fracționar satisface ambele proprietăți (alegerea Greedy și substructura optimă), algoritmul Greedy bazat pe sortarea după raportul valoare/greutate garantează obținerea soluției optime globale.

Implementare în C++

```
struct Obiect {
    double greutate, valoare;
    double raport() const { return valoare / greutate; }
};

bool comparareRaport(Obiect a, Obiect b) {
    return a.raport() > b.raport();
}

double rucsacContinuu(std::vector<Obiect> obiecte, double
   capacitate) {
    std::sort(obiecte.begin(), obiecte.end(), comparareRaport);

    double valoareTotala = 0.0;
    double greutateCurenta = 0.0;

    for (auto obiect : obiecte) {
        if (greutateCurenta + obiect.greutate <= capacitate) {
            valoareTotala += obiect.valoare;
            greutateCurenta += obiect.greutate;
        } else {
            double ramas = capacitate - greutateCurenta;
            valoareTotala += obiect.valoare * (ramas /
               obiect.greutate);
            break;
        }
    }
    return valoareTotala;
}
```

Explicația codului

Structura `Obiect` include atât greutatea, cât și valoarea, precum și o metodă pentru calcularea raportului. Funcția `comparareRaport` permite sortarea obiectelor astfel încât obiectul

cu cel mai mare raport să fie considerat primul. Bucla principală parcurge obiectele sortate, adăugându-le în rucsac cât timp există capacitate disponibilă. În cazul în care un obiect nu încape complet, se adaugă doar fracțiunea corespunzătoare capacității rămase.

Complexitatea algoritmului este $O(n \log n)$, dominată de sortarea obiectelor. Alegerea Greedy la fiecare pas produce soluția globală optimă datorită structurii problemei, demonstrând eficiența paradigmei Greedy în problemele de optimizare continuă.

14.2.3 Problema planificării proiectelor cu termene limită

Această problemă constă în selectarea unui subset de proiecte dintr-un set de n proiecte, fiecare având un profit p_i și un termen limită d_i. Obiectivul este maximizarea profitului total, astfel încât fiecare proiect selectat să fie realizat înainte de termenul său. Problema se întâlnește frecvent în managementul resurselor și al timpului, fiind un exemplu clasic de optimizare combinatorială.

Algoritmul Greedy pentru această problemă se bazează pe ideea că proiectele cu profit mai mare trebuie tratate cu prioritate. Pașii algoritmului sunt:

1. Sortarea proiectelor în ordine descrescătoare a profitului. Această sortare garantează că la fiecare pas considerăm proiectul care aduce cel mai mare beneficiu.

2. Alocarea fiecărui proiect în ultima zi disponibilă din intervalul său $[1, d_i]$, pentru a maximiza posibilitatea de a încadra cât mai multe proiecte ulterior.

3. Dacă nu există niciun slot liber în intervalul respectiv, proiectul este sărit, iar algoritmul trece la următorul proiect din listă.

Această strategie garantează profit maxim deoarece fiecare alegere Greedy (proiectul cu cel mai mare profit disponibil) poate fi plasată într-un slot fără a afecta posibilitatea altor proiecte cu profit mai mic, respectând proprietățile de substructură optimă și alegere Greedy.

Implementare în C++

```
struct Proiect {
    int profit, termen;
};
```

```
bool compararProfit(Proiect a, Proiect b) {
    return a.profit > b.profit;
}

int planificaProiecte(std::vector<Proiect> proiecte) {
    std::sort(proiecte.begin(), proiecte.end(), compararProfit);
    int termenMaxim = 0;
    for (auto proiect : proiecte) {
        termenMaxim = std::max(termenMaxim, proiect.termen);
    }
    std::vector<bool> slotLiber(termenMaxim + 1, true);
    int profitTotal = 0;
    for (auto proiect : proiecte) {
        for (int t = proiect.termen; t > 0; t--) {
            if (slotLiber[t]) {
                profitTotal += proiect.profit;
                slotLiber[t] = false;
                break;
            }
        }
    }
    return profitTotal;
}
```

Explicația codului

Structura `Proiect` reține profitul și termenul limită. Funcția `compararProfit` permite sortarea proiectelor după profit, astfel încât să fie procesate cele mai valoroase primele. Vectorul `slotLiber` urmărește care zile sunt disponibile. Algoritmul parcurge fiecare proiect și încearcă să-l plaseze în ultima zi disponibilă până la termenul său. Dacă nu există nicio zi liberă, proiectul este omis. Astfel, profitul total este maximizat.

Complexitatea algoritmului este $O(n \log n + n \cdot d_{\max})$, dominată de sortarea proiectelor și de căutarea unui slot liber pentru fiecare proiect. Acest exemplu ilustrează cum paradigma Greedy poate transforma o problemă de programare a resurselor într-o soluție eficientă și clară.

14.2.4 Concluzii

Algoritmii Greedy oferă o serie de avantaje clare care îi fac atractivi pentru problemele de optimizare. În primul rând, eficiența lor este remarcabilă: de obicei au complexități temporale polinomiale și sunt capabili să genereze rapid soluții fezabile. În al doilea rând, simplitatea implementării este un punct forte: algoritmii Greedy sunt intuitivi și ușor de programat, ceea ce

reduce riscul de erori și facilitează înțelegerea logicii problemei. În al treilea rând, performanța practică este adesea foarte bună; chiar dacă nu garantează întotdeauna soluția optimă globală, în multe cazuri oferă rezultate apropiate de optim, suficient de bune pentru aplicații reale.

Pe de altă parte, există limitări care trebuie luate în considerare. Alegerea optimului local nu garantează întotdeauna soluția globală, astfel că aplicarea Greedy fără verificarea proprietăților problemei poate duce la soluții suboptime. Lipsa reversibilității este un alt dezavantaj: odată ce o decizie este luată, algoritmul nu revine asupra ei, ceea ce limitează flexibilitatea. În plus, demonstrarea corectitudinii unui algoritm Greedy poate fi dificilă, necesitând adesea argumente riguroase bazate pe proprietatea substructurii optime și pe validitatea alegerii Greedy.

Paradigma Greedy se aplică într-o gamă largă de domenii. În compresia datelor, algoritmul lui Huffman utilizează selecții Greedy pentru codificarea eficientă a simbolurilor. În rețelele de calculatoare, algoritmii de rutare Greedy aleg trasee eficiente pentru pachetele de date. În grafică computațională, triangularea poligoanelor poate fi realizată prin selecții locale optime. În inteligența artificială, algoritmii de căutare Greedy contribuie la explorarea rapidă a spațiului de stări. În planificarea resurselor, problemele precum planificarea activităților sau a proiectelor pot fi abordate eficient cu Greedy, atunci când criteriile de selecție sunt bine definite.

Astfel, tehnica Greedy reprezintă un instrument valoros pentru proiectarea algoritmilor de optimizare. Succesul aplicării sale depinde de identificarea corectă a criteriului de selecție și de verificarea proprietăților fundamentale: substructura optimă și alegerea Greedy. Atunci când aceste condiții sunt satisfăcute, Greedy permite construirea rapidă a unor soluții optime sau aproape optime, cu complexitate temporală excelentă și implementare simplă. Această combinație de eficiență și claritate explică popularitatea paradigmei în numeroase probleme practice de optimizare și planificare.

14.3 Probleme rezolvate

În secțiunea care urmează propunem o serie de probleme de optimizare rezolvate prin metoda Greedy.

1. Se consideră diferite tipuri de bomboane identificate prin gramaj (număr întreg) și preț (număr real). Avem la dispoziție un buget de S lei la dispoziție, citit de la tastatură. Să se

maximizeze numărul de bomboane alese astfel încât acestea să nu depășească bugetul S ales. Să se afișeze numărul bomboanelor alese precum și gramajul și prețul lor total.

Rezolvare:

Date de intrare:

- un număr întreg n – numărul de bomboane disponibile;
- pentru fiecare bomboană: gramaj (număr întreg) și preț (număr real);
- un număr real S – bugetul disponibil.

Date de ieșire:

- numărul total de bomboane alese;
- gramajul total al bomboanelor alese;
- suma totală cheltuită.

Algoritm de rezolvare prin Greedy:

(a) Pentru a maximiza numărul de bomboane, se aleg mai întâi bomboanele cu prețul cel mai mic.

(b) Se sortează bomboanele crescător după preț.

(c) Se parcurge lista sortată și se adaugă bomboane atâta timp cât bugetul permite.

(d) La fiecare pas se actualizează suma cheltuită, gramajul total și numărul de bomboane.

Algoritm scris în C++:

```
#include<iostream>
#include<vector>
#include<algorithm>

struct Bomboana {
    int gramaj;
    double pret;
};

bool comparatorPret(const Bomboana& a, const Bomboana& b) {
    return a.pret < b.pret; }
```

```
int main() {
    int n;
    std::cin >> n;
    std::vector<Bomboana> v(n);

    for (int i = 0; i < n; i++)
        std::cin >> v[i].gramaj >> v[i].pret;

    double S;
    std::cin >> S;
    std::sort(v.begin(), v.end(), comparatorPret);

    int nrBomboane = 0;
    int gramajTotal = 0;
    double suma = 0.0;

    for (int i = 0; i < n; i++) {
        if (suma + v[i].pret <= S) {
            suma += v[i].pret;
            gramajTotal += v[i].gramaj;
            nrBomboane++;
        } else {
            break;
        }
    }

    std::cout << nrBomboane << "\n";
    std::cout << gramajTotal << "\n";
    std::cout << suma;

    return 0;
}
```

Justificarea soluției alese: Algoritmul selectează bomboanele în ordine crescătoare a prețului. Dacă o bomboană mai ieftină nu poate fi inclusă în buget, atunci nicio bomboană mai scumpă nu poate fi inclusă ulterior. Prin urmare, strategia Greedy garantează obținerea unui număr maxim de bomboane fără a depăși bugetul S.

Analiza complexității:

- *Complexitate temporală*:
  - Sortarea bomboanelor: $O(n \log n)$;

  - Parcurgerea listei: $O(n)$;
  - Complexitatea totală: $O(n \log n)$.

- *Complexitate spațială*: Spațiu suplimentar: $O(1)$, în afara vectorului de stocare a datelor.

Demonstrație:

Vom demonstra că strategia Greedy care alege bomboanele în ordinea crescătoare a prețului produce o soluție optimă.
Presupunem că algoritmul Greedy alege bomboanele:

$$G = \{g_1, g_2, \ldots, g_k\},$$

unde acestea sunt ordonate crescător după preț.
Presupunem prin reducere la absurd că există o soluție optimă O care conține un număr mai mare de bomboane decât soluția obținută de algoritmul Greedy.
Fie g_{k+1} prima bomboană pe care algoritmul Greedy nu a putut să o aleagă deoarece adăugarea acesteia ar depăși bugetul disponibil. Deoarece bomboanele sunt sortate crescător după preț, toate bomboanele care urmează în listă au un preț mai mare sau egal cu prețul lui g_{k+1}.
Dacă soluția optimă O ar conține o bomboană care nu a fost aleasă de algoritmul Greedy, putem înlocui acea bomboană cu o bomboană aleasă de algoritmul Greedy care are preț mai mic sau egal. Această înlocuire nu mărește costul total al soluției și nu reduce numărul de bomboane alese.
Prin repetarea acestui proces, putem transforma soluția optimă O într-o soluție care conține exact aceleași bomboane ca algoritmul Greedy, fără a depăși bugetul S.
Rezultă că nu poate exista o soluție care să conțină mai multe bomboane decât soluția obținută de algoritmul Greedy.
Prin urmare, alegerea bomboanelor cu cele mai mici prețuri până la epuizarea bugetului este o strategie optimă și garantează obținerea numărului maxim posibil de bomboane fără depășirea sumei S.

2. O librărie online dorește să implementeze un sistem eficient de management al cărților. Fiecare carte din catalog are: un id numeric unic, un $titlu$ (șir de caractere), un $autor$ (șir de caractere), un *preț* (număr real pozitiv) și un *număr de pagini* (număr întreg pozitiv). Cărțile trebuie afișate alfabetic după numele autorului, iar în cazul în care două cărți au același autor, sortarea se face după preț (crescător). Un client dorește să selecteze cărți din catalog astfel încât suma totală să nu depășească un preț stabilit X. Să se determine și să se afișeze lista cărților selectate, precum și prețul total și numărul total de pagini obținut.

Rezolvare:

Date de intrare:

- un număr întreg n – numărul de cărți;
- pentru fiecare carte: id, $titlu$, $autor$, $pret$, $numar$ de pagini;
- un număr real X – bugetul maxim.

Date de iesire:

- lista cărților selectate optim;
- prețul total al cărților selectate;
- numărul total de pagini obținut.

Algoritm de rezolvare prin metoda Greedy:

(a) Pentru fiecare carte se calculează raportul dintre numărul de pagini și preț:

$$R_i = \frac{pag_i}{pret_i}.$$

(b) Se sortează cărțile descrescător după raportul $pagini/pret$.

(c) Se parcurg cărțile în această ordine și se selectează o carte dacă prețul acesteia nu depășește bugetul disponibil.

(d) Se actualizează permanent suma totală cheltuită și numărul total de pagini.

(e) La afișarea catalogului, cărțile pot fi afișate după autor și, în caz de egalitate, după preț, fără a modifica ordinea utilizată pentru selecția Greedy.

Algoritm scris in C++:

```
#include<iostream>
#include<vector>
#include<algorithm>
#include <string>

struct Carte {
    int id;
    std::string titlu;
    std::string autor;
    double pret;
    int pagini;
};

bool comparatorGreedy(const Carte& a, const Carte& b) {
    double r1 = (double)a.pagini / a.pret;
    double r2 = (double)b.pagini / b.pret;
    return r1 > r2;
}

int main() {
    int n;
    std::cin >> n;
    std::vector<Carte> carti(n);

    for (int i = 0; i < n; i++) {
        std::cin >> carti[i].id;
        std::cin >> carti[i].titlu;
        std::cin >> carti[i].autor;
        std::cin >> carti[i].pret;
        std::cin >> carti[i].pagini;
    }

    double X;
    std::cin >> X;
    std::sort(carti.begin(), carti.end(), comparatorGreedy);

    double suma = 0.0;
    int totalPagini = 0;
    std::vector<Carte> selectate;

    for (const auto& c : carti) {
        if (suma + c.pret <= X) {
```

```
            suma += c.pret;
            totalPagini += c.pagini;
            selectate.push_back(c);
        }
    }

    for (const auto& c : selectate)
        std::cout << c.titlu << " " << c.autor << " "
             << c.pret << " " << c.pagini << "\n";

    std::cout << "Pret total: " << suma << "\n";
    std::cout << "Pagini totale: " << totalPagini;

    return 0;
}
```

Justificarea soluției propuse: Strategia Greedy sortează cărțile descrescător după raportul dintre numărul de pagini și preț. Astfel, sunt analizate mai întâi cărțile care oferă cel mai mare număr de pagini pentru fiecare unitate monetară cheltuită. O carte este selectată dacă prețul acesteia poate fi suportat de bugetul rămas. Prin această strategie se urmărește obținerea unei selecții cât mai eficiente din punctul de vedere al raportului pagini/preț, fără depășirea bugetului X.

Pentru afișarea cărților în ordinea cerută de catalog, după autor și apoi după preț, se poate utiliza o copie a vectorului de cărți sau se poate realiza sortarea într-un moment separat, după finalizarea selecției. Această sortare nu este necesară pentru algoritmul Greedy propriu-zis.

Analiza complexității:

- *Complexitate temporală*:
  - calcularea rapoartelor în timpul sortării: inclusă în operațiile de comparație;
  - Sortarea cărților: $O(n \log n)$;
  - Selecția Greedy: $O(n)$;
  - Complexitate totală: $O(n \log n)$.
- *Complexitate spațială*: Spațiu suplimentar: $O(n)$ pentru lista de cărți selectate.

Demonstrație: Fie o carte C_i cu prețul p_i și numărul de pagini pag_i. Definim eficiența unei cărți prin raportul:

$$R_i = \frac{pag_i}{p_i}.$$

Algoritmul sortează cărțile descrescător după această valoare și analizează mai întâi cărțile care oferă cel mai mare număr de pagini pentru fiecare unitate monetară cheltuită.
Dacă două cărți pot fi comparate în cadrul strategiei Greedy și avem:

$$R_i > R_j,$$

atunci cartea C_i oferă un raport pagini/preț mai bun decât cartea C_j. Prin urmare, strategia Greedy va analiza mai întâi cartea C_i și o va selecta dacă aceasta poate fi achiziționată în limita bugetului disponibil.
În acest mod, algoritmul prioritizează permanent cărțile cu eficiența cea mai mare și construiește soluția prin alegeri locale succesive, fără a depăși bugetul X.
În concluzie, algoritmul utilizează o singură sortare, după raportul $pagini/pret$, iar apoi parcurge cărțile în această ordine pentru a realiza selecția Greedy.

14.4 Probleme propuse

*

1. Aveți la dispoziție un număr nelimitat de bancnote cu valori diferite $\{1, 5, 10, 20, 50, 100, 200, 500\}$.
   Determinați care este numărul minim de bancnote utilizate pentru a obține o suma X.
   Exemplu: Pentru suma $X = 704$, numărul minim de bancnote va fi 2 (20 + 50 = 70).

**

1. Se dă o bucată de hârtie de dimensiune $A \times B$. Hârtia se taie în pătrate de orice dimensiune. Găsiți numărul minim de pătrate în care poate tăia.
   Exemplu: $13 \times 29 = 2 \cdot (13 \times 13) + 4 \cdot (3 \times 3) + 3 \cdot (1 \times 1)$ = 9 bucăți

2. De-a lungul străzii principale din orașul nostru există n plopi, pentru fiecare cunoscându-se înălțimea. Primarul orașului dorește ca plopii să aibă înălțimile în ordine descrescătoare. Pentru aceasta, este posibilă tăierea dintr-un plop a unei bucăți – este o tehnică ecologică, nevătămătoare, în urma căreia plopul nu are de suferit. Plopii nu pot fi înălțați în niciun fel. Determinați numărul minim de plopi din care se va tăia și lungimea totală minimă a bucăților tăiate.
   Exemplu: Pentru $n = 5$ și înălțimile plopilor: $[10, 8, 9, 11, 6]$, rezultatul este: 2 plopi tăiați, lungime totală tăiată = 4 (tăiem din plopul cu înălțimea 9: scădem la 8; tăiem din plopul cu înălțimea 11: scădem la 8; vectorul final este $[10, 8, 8, 8, 6]$)

***

1. Se citește dintr-un fișier dispunerea geografică a N țări. Având la dispoziție 4 culori, se cere să se coloreze harta, astfel încât să nu existe două țări vecine cu aceeași culoare.
   Exemplu de date de intrare fișier: 8 (numărul de țări) și perechile (1 2), (1 4), (1 5), (2 3), (2 4), (3 6), (4 5), (4 6), (4 7) și (7 8), care reprezintă perechi de țări vecine.
   Observație: Perechea (x, y) presupune indirect că există și o pereche (y, x).

2. Pentru cadourile pe care Moş Crăciun urmează să le cumpere copiilor cuminți, Consiliul Polului Nord a alocat suma de s eureni. Știind că în comerțul polar se utilizează $n + 1$ tipuri de bancnote de valori $1, e^1, e^2, e^3, \ldots, e^n$ și faptul că Moșul trebuie să primească un număr minim de bancnote pentru suma aprobată, să se determine numărul de bancnote din fiecare tip utilizat în plata sumei și numărul total de bancnote care i s-au alocat. Să se demonstreze corectitudinea soluției alese pentru rezolvare.

3. RATBV a decis să introducă o cursă rapidă care să străbată orașul, de la un capăt la celălalt, pe o arteră principală. RATBV are N stații propuse pentru alcătuirea traseului însă ele trebuie selectate astfel încât să nu fie nici foarte dese, dar nici foarte rare.
   Criteriile după care RATBV stabilește stațiile în care va opri cursa rapidă sunt:
   - Între două stații alăturate să fie cel puțin X metri.
   - Numărul total de stații de pe traseu (arteră) să fie maxim.

Stațiile sunt numerotate în ordinea în care sunt dispuse de-a lungul traseului, de la 1 la N. Se citesc dintr-un fișier: numărul total de stații propuse N, distanța minimă X dintre două stații consecutive ale traseului și $(N-1)$ numere care reprezintă distanțele dintre fiecare două stații consecutive.

Exemplu de fișier

10

60

100 50 25 25 50 10 10 80 20

Să se scrie un program care afișează numărul maxim de stații care pot fi alese în funcție de criteriile stabilite, precum și care sunt aceste stații.

Începem mereu cu stația 1, dar nu e obligatoriu să și încheiem traseul la stația N. Pe exemplul dat, se va afișa:

5

1 2 4 6 9

4. În curtea unui atelier de reparații auto, sunt N mașini care trebuie sa fie reparate. Deoarece nu sunt suficienți mecanici, în fiecare moment de timp se poate lucra doar la o singură mașina. Cunoscând timpul necesar pentru repararea fiecărei mașini, scrieți un program care calculează numărul maxim de mașini care pot fi reparate într-un interval de timp T. Se citesc următoarele valori dintr-un fișier: numărul de mașini N, timpul total de lucru T, N numere naturale $(t1, t2, \ldots, tn)$ care reprezintă reprezentând timpii necesari pentru repararea fiecărei mașini.

   Exemplu de fișier

   5

   10

   6 2 4 8 2

   Pentru acest exemplu, având la dispoziție $N = 5$ mașini, în timpul $T = 10$ se pot repara maxim 3 mașini.

5. La un birou care se ocupă cu analiza proiectelor de investiții, N investitori au depus până la termenul legal, câte un proiect. Cunoscând timpul necesar pentru analizarea fiecărui proiect, scrieți un program care determină ordinea în care vor fi analizate proiectele, astfel

încât timpul mediu de așteptare pentru investitori să fie minim. Se citesc următoarele valori dintr-un fișier: numărul de proiecte depuse N, N numere naturale $(t1, t2, \ldots, tn)$ care reprezintă timpii necesari pentru analizarea fiecărui proiect.

Exemplu de fișier

5

60 50 30 10 40

Pentru acest exemplu, ordinea analizării proiectelor este 4 3 5 2 1.

****

1. Ionuț vrea să facă o petrecerea și încearcă să se decidă pe cine să cheme. El vrea să aleagă n persoane și a făcut o listă cu perechile de persoane care se cunosc. Vrea să cheme cât mai multe persoane, ținând cont de două restricții:

   (a) fiecare persoana să aibă minim 5 alte persoane pe care le cunoaște;

   (b) fiecare persoana să aibă 5 persoane pe care nu le cunoaște.

   (a) Într-un fișier text avem: lista de n persoane ($nume$ și id) și lista de perechi de persoane care se cunosc (exemplu: (2,4), (2,7), etc). Să se scrie într-un fișier text de ieșire cea mai bună alegere pentru invitații la petrecere, pe baza restricțiilor date mai sus. **Sugestie**: identificați persoanele care nu ar trebui invitate la petrecere.

   (b) Să se explice modelul matematic care stă la baza acestei probleme (cum arată problema de optimizat, cum arată restricțiile pe care trebuie să le îndeplinească soluția).

2. Într-un laborator de analize chimice avem la dispoziție n reactivi. Pentru a evita eventuale accidente, respectiv pentru a evita deprecierea reactivilor, știm că aceștia trebuie să fie stocați în condiții de mediu speciale. Prin urmare pentru fiecare reactiv avem la dispoziție intervalul de temperatură $[tmin, tmax]$ în care trebuie să se încadreze temperatura de stocare a acestuia. Acești reactivi vor fi introduși în frigidere. Fiecare frigider are un termometru cu ajutorul căruia putem stabili temperatura (care se presupune că va fi constantă) care va fi în interiorul acelui frigider (exprimată printr-un număr întreg de grade Celsius).

(a) Să se specifice care este numărul minim de frigidere de care avem nevoie pentru a stoca cei n-reactivi astfel încât diferența $tmax - tmin$ să fie de maxim K grade celsius, K - precizat.

(b) Să se explice modelul matematic care stă la baza acestei probleme (cum arată problema de optimizat, cum arată restricțiile pe care trebuie să le îndeplinească soluția).

3. Se consideră o tablă de șah cu n linii și m coloane. La o poziție dată se află un cal de șah, acesta putându-se deplasa pe tablă în modul specific acestei piese de șah (în formă de L). Să se determine o modalitate de parcurgere a tablei de către cal, astfel încât acesta să nu treacă de două ori prin aceeași poziție.

Capitolul 15

Programare Dinamică

15.1 Aspecte teoretice

Programarea Dinamică (PD) este o metodă avansată de proiectare a algoritmilor, destinată problemelor de optimizare care pot fi descompuse în subprobleme mai mici, dar suprapuse. Spre deosebire de tehnica *Divide et Impera*, unde subproblemele sunt independente, în PD se urmărește reutilizarea rezultatelor deja obținute, evitând astfel recalcularea inutilă. Ideea centrală constă în împărțirea unei probleme complexe într-o serie de decizii secvențiale și în memorarea soluțiilor parțiale pentru a construi progresiv soluția finală.

Această tehnică a fost introdusă de Richard Bellman în anii 1950 și a devenit una dintre cele mai influente paradigme din informatică, fiind aplicată cu succes în numeroase domenii precum optimizarea rețelelor, teoria grafurilor, biologia computațională sau analiza secvențelor.

15.1.1 Fundamente teoretice detaliate

Programarea Dinamică se sprijină pe trei idei fundamentale.

1. Principiul substructurii optime

   Acest principiu stipulează că o soluție optimă a unei probleme poate fi construită din soluțiile optime ale subproblemelor sale. Formal, dacă o soluție optimă pentru o problemă dată implică o secvență de decizii, atunci orice subsecvență a acestei secvențe de decizii trebuie să fie, la rândul său, o soluție optimă pentru subproblema corespunzătoare. Aceasta implică faptul că, pentru a găsi soluția optimă globală, nu este necesar să reevaluăm deciziile anterioare odată ce am determinat soluția optimă pentru o subproblemă.

2. Principiul subproblemelor suprapuse

   O problemă are proprietatea subproblemelor suprapuse dacă, în procesul de descompunere recursivă a problemei inițiale, se întâlnesc în mod repetat aceleași subprobleme. Spre

deosebire de abordarea *Divide et Impera*, unde subproblemele sunt tipic disjuncte, în PD, rezolvarea unei subprobleme poate fi necesară de mai multe ori. Eficiența Programării Dinamice derivă din capacitatea de a stoca soluțiile acestor subprobleme recurente (prin tehnici precum memoizarea sau construirea unei tabele Bottom-Up), evitând astfel recalcularea redundantă și reducând complexitatea computațională de la exponențial la polinomial.

3. Caracterul asemănător al subproblemelor
   Acest principiu, adesea implicit în primele două, subliniază că subproblemele generate prin descompunere sunt de același tip cu problema originală, doar de o dimensiune sau complexitate mai mică. Această uniformitate permite aplicarea repetată a aceleiași logici de rezolvare și facilitează definirea unei relații de recurență care leagă soluția unei probleme de soluțiile subproblemelor sale.

Pentru a putea fi rezolvată prin Programare Dinamică, o problemă trebuie să îndeplinească anumite condiții conceptuale. În primul rând, trebuie să poată fi exprimată ca o succesiune de decizii dependente, unde soluția problemei mari depinde de soluțiile unor probleme mai mici. De asemenea, problema trebuie să respecte principiul optimalității, ceea ce garantează că o combinație de soluții optime locale poate genera o soluție globală optimă.

Un alt aspect esențial este existența subproblemelor suprapuse, care fac posibilă memorarea rezultatelor intermediare. Dacă subproblemele ar fi complet independente, metoda *Divide et Impera* ar fi mai potrivită. În fine, trebuie să existe o modalitate clară de stocare și accesare a acestor soluții parțiale, ceea ce conduce la noțiunea de memoizare.

15.1.2 Tehnica memoizării

Memoizarea este tehnica prin care rezultatele subproblemelor deja calculate sunt salvate pentru a fi reutilizate atunci când aceleași subprobleme apar din nou. Astfel, se evită recalcularea acelorași valori și se poate reduce semnificativ timpul de execuție al unui algoritm recursiv.

Un exemplu clasic este calculul numerelor Fibonacci. Implementarea recursivă naivă a funcției Fibonacci are o complexitate temporală exponențială, deoarece aceleași valori $F(k)$ sunt calculate de mai multe ori. Prin aplicarea memoizării, fiecare valoare $F(k)$ este calculată o singură dată și apoi este păstrată într-un vector pentru a putea fi reutilizată.

În cazul numerelor Fibonacci, deoarece valorile sunt asociate unor indici consecutivi $0, 1, \ldots, n$, un `std::vector` este mai potrivit decât o structură de tip `std::unordered_map`. Accesarea unei poziții din vector are complexitatea $O(1)$ și nu este necesară stocarea explicită a cheilor.

```
#include <iostream>
#include <vector>

// Varianta Top-Down cu memoizare
long long fibonacciMemoizare(int dimensiune, std::vector<long
   long>& vector) {
    if (dimensiune <= 1)
        return dimensiune;

    // Valoarea a fost deja calculata
    if (vector[dimensiune] != -1)
        return vector[dimensiune];

    // Calculam si salvam rezultatul
    vector[dimensiune] = fibonacciMemoizare(dimensiune - 1, vector)
            + fibonacciMemoizare(dimensiune - 2, vector);

    return vector[dimensiune];
}

// Varianta Bottom-Up
long long fibonacciProgramareDinamica(int dimensiune) {
    if (dimensiune <= 1)
        return dimensiune;

    std::vector<long long> dp(dimensiune + 1);

    dp[0] = 0;
    dp[1] = 1;

    for (int i = 2; i <= dimensiune; i++)
        dp[i] = dp[i - 1] + dp[i - 2];
    return dp[n];
}

int main() {
    int dimensiune;
    std::cin >> dimensiune;

    // Vectorul de memoizare este local
    std::vector<long long> vector(dimensiune + 1, -1);
```

```
    long long rezultatMemoizare = fibonacciMemoizare(dimensiune,
        vector);
    long long rezultatDinamica =
        fibonacciProgramareDinamica(dimensiune);

    std::cout << "Fibonacci cu memoizare: "<< rezultatMemoizare <<
        "\n";
    std::cout << "Fibonacci Dinamica: " << rezultatDinamica <<
        "\n";
    return 0;
}
```

În prima versiune se folosește abordarea *Top-Down*. Algoritmul pornește de la problema inițială $F(n)$ și apelează recursiv subproblemele necesare. Înainte de a calcula o valoare $F(k)$, se verifică dacă aceasta a fost deja memorată în vectorul `memo`. Dacă valoarea există, ea este returnată direct; în caz contrar, este calculată, salvată și apoi returnată.

În cea de-a doua versiune se folosește abordarea *Bottom-Up*. În loc să pornească de la $F(n)$ și să descompună problema recursiv, algoritmul începe cu cazurile de bază $F(0)$ și $F(1)$ și construiește succesiv valorile $F(2), F(3), \ldots, F(n)$.

Ambele variante evită recalcularea subproblemelor. Pentru calculul lui $F(n)$, fiecare valoare $F(k)$, pentru $0 \leq k \leq n$, este calculată o singură dată. Prin urmare, complexitatea temporală a ambelor variante este $O(n)$.

Varianta Top-Down utilizează, pe lângă vectorul de memoizare, și stiva apelurilor recursive, astfel încât complexitatea spațială este $O(n)$. Varianta Bottom-Up utilizează un vector de dimensiune $n + 1$, având de asemenea complexitatea spațială $O(n)$.

În cazul particular al șirului Fibonacci, varianta Bottom-Up poate fi optimizată și mai mult, deoarece pentru calcularea lui $F(i)$ sunt necesare doar valorile $F(i-1)$ și $F(i-2)$. Astfel, nu mai este necesară păstrarea întregului vector, iar complexitatea spațială poate fi redusă la $O(1)$.

15.1.3 Abordări în Programarea Dinamică

În aplicarea Programării Dinamice, există două strategii majore de implementare, fiecare cu avantajele și particularitățile sale, dar ambele bazate pe principiile fundamentale ale PD: abordarea Top-Down (cu memoizare) și abordarea Bottom-Up (iterativă).

Abordarea Top-Down (Memoizare)

Această strategie pornește de la problema principală și o descompune recursiv în subprobleme din ce în ce mai mici. Soluțiile subproblemelor sunt calculate doar atunci când sunt necesare și sunt stocate într-o structură de date (de obicei un tabel sau un dicționar) pentru a fi reutilizate ulterior. Această tehnică de stocare se numește memoizare.

Algoritmul începe cu apelul funcției pentru problema de dimensiune maximă. Dacă soluția pentru o anumită subproblemă a fost deja calculată și stocată, aceasta este returnată direct. Altfel, subproblema este rezolvată recursiv, iar rezultatul este stocat înainte de a fi returnat.

Avantaje:

- Intuitivă și ușor de implementat, deoarece urmează direct definiția recursivă a problemei.
- Calculează soluțiile doar pentru subproblemele care sunt efectiv atinse și necesare.

Dezavantaje:

- Poate genera un *overhead* semnificativ datorită apelurilor recursive și gestionării stivei de apeluri.
- Risc de depășire a stivei (*stack overflow*) pentru probleme cu adâncime mare a recursivității.

Abordarea Bottom-Up (Iterativă)

Spre deosebire de abordarea Top-Down, strategia Bottom-Up construiește soluția pornind de la cele mai mici și simple subprobleme, progresând iterativ către problema principală. Soluțiile subproblemelor elementare sunt calculate primele și sunt folosite pentru a deduce soluțiile subproblemelor de dimensiuni mai mari.

Algoritmul inițializează o structură de date (de obicei un tabel unidimensional sau bidimensional) cu soluțiile pentru cazurile de bază. Apoi, utilizează bucle iterative pentru a completa tabelul, calculând soluțiile pentru subproblemele de dimensiuni crescătoare, bazându-se pe soluțiile deja calculate ale subproblemelor mai mici.

Avantaje:

- Elimină overhead-ul apelurilor recursive, fiind adesea mai eficientă din punct de vedere al timpului de execuție și al utilizării memoriei (nu necesită stivă de apeluri).

- Mai puțin susceptibilă la erori de depășire a stivei.
- Oferă o vizualizare clară a dependențelor dintre subprobleme.

Dezavantaje:

- Poate fi mai puțin intuitivă de implementat pentru unii, necesitând o înțelegere clară a ordinii în care trebuie rezolvate subproblemele.
- Calculează soluțiile pentru toate subproblemele posibile, chiar dacă unele dintre ele nu sunt strict necesare pentru soluția finală a problemei inițiale.

Ambele abordări sunt valide și conduc la soluții corecte pentru problemele care satisfac proprietățile Programării Dinamice. Alegerea între Top-Down și Bottom-Up depinde adesea de natura specifică a problemei, de preferințele programatorului și de constrângerile de performanță. În practică, abordarea Bottom-Up este adesea preferată pentru eficiența sa, în timp ce Top-Down poate fi mai rapid de prototipat.

```
// Comparatie intre abordarile Top-Down si Bottom-Up
class DynamicProgramming {
public:
    int topDown(int n) {
        std::vector<int> memo(n + 1, -1);
        return topDownHelper(n, memo);
    }
    int bottomUp(int n) {
        if (n <= 1) return n;
        std::vector<int> dp(n + 1);
        dp[0] = 0; dp[1] = 1;
        for (int i = 2; i <= n; i++)
            dp[i] = dp[i - 1] + dp[i - 2];
        return dp[n];
    }

private:
    int topDownHelper(int n, std::vector<int>& memo) {
        if (n <= 1) return n;
        if (memo[n] != -1) return memo[n];
        memo[n] = topDownHelper(n - 1, memo) + topDownHelper(n -
           2, memo);
        return memo[n];
    }
};
```

Prin combinarea acestor principii - optimalitate, substructură optimă și reutilizarea rezultatelor - programarea dinamică oferă o abordare sistematică și elegantă pentru probleme care altfel ar necesita un efort computațional considerabil.

15.2 Algoritmi clasici

15.2.1 Problema celui mai lung subșir crescător

Una dintre cele mai cunoscute aplicații ale Programării Dinamice este determinarea celui mai lung subșir crescător dintr-un șir dat. Fie un șir de numere întregi $v = (v_1, v_2, \ldots, v_n)$. Se cere identificarea unui subșir $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ cu proprietatea $v_{i_1} < v_{i_2} < \ldots < v_{i_k}$, astfel încât lungimea k să fie maximă.

Această problemă ilustrează perfect structura unei probleme de optimizare care poate fi abordată prin Programare Dinamică. Soluția globală depinde de deciziile anterioare, iar subproblemele se suprapun parțial, ceea ce o face un exemplu ideal de aplicare a acestei tehnici.

Modelul problemei

Pentru fiecare poziție i din șir, se definește o valoare $L[i]$ care reprezintă lungimea celui mai lung subșir crescător care începe la poziția i. Relația de recurență este următoarea:

$$L[i] = 1 + \max(L[j]) \quad \text{pentru toate } j > i \text{ cu } v_i \leq v_j$$

Dacă nu există un astfel de j, atunci $L[i] = 1$, deoarece elementul v_i formează singur un subșir de lungime 1. Astfel, problema inițială se reduce la calcularea tuturor valorilor $L[i]$ și la determinarea maximului acestora.

Descrierea algoritmului

Algoritmul parcurge șirul de la dreapta la stânga și, pentru fiecare element, verifică toate pozițiile următoare pentru a determina cel mai lung subșir crescător posibil. Pe parcurs, se păstrează și un vector de predecesori pentru a reconstrui efectiv subșirul optim.

Complexitatea algoritmului este $O(n^2)$, deoarece pentru fiecare element se compară cu toate elementele aflate după el. Deși există versiuni optimizate $O(n \log n)$, varianta clasică explică excelent principiile Prgramării Dinamice.

```
std::pair<int, std::vector<int>>
   celMaiLungSubsirCrescator(std::vector<int>& sir) {
    int n = sir.size();
    // fiecare element formeaza un subsir de lungime 1
    std::vector<int> lungimi(n, 1);
    std::vector<int> predecesor(n, -1);
    int lungimeMaxima = 1;
    int pozitieMaxima = 0;

    for (int i = n - 1; i >= 0; i--) {
        for (int j = i + 1; j < n; j++) {
            if (sir[i] <= sir[j] && lungimi[j] + 1 > lungimi[i]) {
                lungimi[i] = lungimi[j] + 1;
                predecesor[i] = j;

                if (lungimi[i] > lungimeMaxima) {
                    lungimeMaxima = lungimi[i];
                    pozitieMaxima = i;
                }
            }
        }
    }
    // Reconstruim solutia
    std::vector<int> solutie;
    int current = pozitieMaxima;
    while (current != -1) {
        solutie.push_back(sir[current]);
        current = predecesor[current];
    }
    return {lungimeMaxima, solutie};
}
```

Explicația implementării

În implementarea de mai sus, vectorul $lungimi$ stochează pentru fiecare poziție lungimea celui mai lung subșir crescător care pornește din acea poziție. Vectorul $predecesor$ este utilizat pentru a reconstrui efectiv secvența optimă, urmărind succesiunea de indecși corespunzători.

Algoritmul parcurge elementele dinspre final către început, deoarece valoarea $L[i]$ depinde de valorile $L[j]$ pentru $j > i$. Pentru fiecare pereche (i, j), dacă $v_i \leq v_j$, se verifică dacă extinderea subșirului care începe în j duce la o soluție mai bună pentru i.

După completarea celor doi vectori, se reconstruiește soluția optimă pornind de la elementul corespunzător celei mai mari valori din $lungimi$.

Complexitate și analiză

Complexitatea temporală a algoritmului este $O(n^2)$, deoarece pentru fiecare element se realizează o parcurgere completă a elementelor aflate după el. Complexitatea spațială este $O(n)$, datorită vectorilor auxiliari $lungimi$ și $predecesor$.

Deși pot fi implementate metode mai rapide, de exemplu utilizând algoritmul de căutare binară ($O(n \log n)$), soluția prezentată este un exemplu clasic de problemă rezolvabilă cu Programare Dinamică, evidențiind modul în care memorarea soluțiilor intermediare duce la evitarea recalculărilor și la o formulare clară și sistematică a problemei.

Exemplu de rulare

Pentru șirul $v = [3, 1, 5, 2, 6, 4, 9]$, algoritmul determină subșirul crescător de lungime maximă $[1, 2, 4, 9]$ sau $[1, 5, 6, 9]$, în funcție de ordinea comparării. Lungimea maximă obținută este 4.

15.2.2 Problema triunghiului

Problema triunghiului numeric este un exemplu clasic de aplicare a Programării Dinamice pentru determinarea unei căi optime într-o structură ierarhică. Se consideră un triunghi format din numere întregi, organizate astfel încât fiecare element de pe un rând are două elemente „copii” imediat sub el, în rândul următor.

Se cere găsirea unei căi de la vârful triunghiului până la baza acestuia, astfel încât suma numerelor parcurse să fie maximă. La fiecare pas, deplasarea este permisă doar către unul dintre cele două elemente aflate direct sub poziția curentă.

Modelul problemei

Fie un triunghi reprezentat printr-o matrice a_{ij}, unde i indică rândul și j poziția din rând. Pentru fiecare poziție (i, j), se definește o valoare $S(i, j)$ care reprezintă suma maximă posibilă de la acea poziție până la bază. Relația de recurență care exprimă structura optimă a problemei este:

$$S(i, j) = a_{ij} + \max(S(i + 1, j), S(i + 1, j + 1))$$

Această formulă reflectă principiul optimalității: alegerea locală (maximul dintre cele două ramuri) conduce către soluția globală optimă. Calculul se efectuează pornind de la baza triunghiului și urcând treptat către vârf.

Algoritmul de rezolvare

Pentru implementarea soluției, se poate utiliza un tablou bidimensional dp de aceeași dimensiune ca triunghiul, în care se vor stoca valorile maxime obținute pentru fiecare poziție. Începem de la rândul penultim și actualizăm fiecare element cu suma dintre valoarea proprie și maximul celor două valori aflate dedesubt.

```
int sumaMaximaTriunghi(std::vector<std::vector<int>>& triunghi) {
    int n = triunghi.size();
    std::vector<std::vector<int>> dp = triunghi;

    // Parcurgem de la baza spre varf
    for (int i = n - 2; i >= 0; i--) {
        for (int j = 0; j <= i; j++) {
            dp[i][j] += std::max(dp[i + 1][j], dp[i + 1][j + 1]);
        }
    }

    return dp[0][0];
}
```

După completarea întregii matrici, elementul de la vârful triunghiului ($dp[0][0]$) va conține suma maximă a unei căi posibile.

Optimizare spațială

Deoarece la un moment dat avem nevoie doar de valorile corespunzătoare rândului curent și ale celui de dedesubt, se poate reduce consumul de memorie de la $O(n^2)$ la $O(n)$, utilizând un singur vector auxiliar. Această optimizare păstrează aceeași logică de calcul, dar înlocuiește structura bidimensională cu una liniară.

```
int sumaMaximaTriunghiOptimizat(std::vector<std::vector<int>>&
   triunghi) {
    int n = triunghi.size();
    std::vector<int> dp = triunghi[n - 1];
```

```
    for (int i = n - 2; i >= 0; i--) {
        for (int j = 0; j <= i; j++) {
            dp[j] = triunghi[i][j] + std::max(dp[j], dp[j + 1]);
        }
    }

    return dp[0];
}
```

Analiza complexității

Atât varianta clasică, cât și cea optimizată au o complexitate temporală de $O(n^2)$, unde n reprezintă numărul de rânduri ale triunghiului. Diferența constă în spațiu: soluția standard utilizează $O(n^2)$ memorie, în timp ce varianta optimizată funcționează în $O(n)$.

Exemplu de rulare

Pentru triunghiul următor:

$$\begin{array}{ccccc} 7 & & & & \\ 3 & 8 & & & \\ 8 & 1 & 0 & & \\ 2 & 7 & 4 & 4 & \\ 4 & 5 & 2 & 6 & 5 \end{array}$$

Algoritmul construiește treptat sumele maxime, obținând în final valoarea 30, corespunzătoare drumului $7 \to 3 \to 8 \to 7 \to 5$.

Această problemă evidențiază clar avantajele tehnicii de Programare Dinamică: pornind de la o formulare simplă, dar recursivă, se poate construi o soluție eficientă și elegantă. Programarea Dinamică permite, în acest caz, reducerea drastică a costului computațional față de abordarea brută, care ar presupune explorarea tuturor căilor posibile - o creștere exponențială a combinațiilor. Astfel, problema triunghiului devine un exemplu didactic ideal pentru înțelegerea modului în care se poate construi o soluție optimă prin agregarea sistematică a deciziilor locale.

15.2.3 Problema matricei binare maximale

Problema matricei binare maximale este un exemplu excelent de aplicare a Programării Dinamice în procesarea datelor bidimensionale. Se consideră o matrice binară (formată doar din

valori 0 și 1), și se cere determinarea celei mai mari submatrice pătratice alcătuite exclusiv din elemente de valoare 1.

Această problemă apare frecvent în domenii precum analiza imaginilor binare, procesarea de date spațiale sau detecția regiunilor uniforme într-o hartă digitală.

Modelul problemei

Fie o matrice binară M de dimensiune $m \times n$. Scopul este să determinăm dimensiunea maximă k pentru care există o submatrice pătratică $k \times k$ compusă doar din valori 1. În plus, se poate determina și poziția acestei submatrice în cadrul matricei inițiale.

Pentru fiecare element $M[i][j]$, se definește o valoare auxiliară $dp[i][j]$, reprezentând latura maximă a unui pătrat complet de 1-uri care are colțul din dreapta-jos în poziția (i, j).

Observația fundamentală este că, pentru a extinde un pătrat de 1-uri, toate cele trei celule vecine din stânga, sus și stânga-sus trebuie să permită acest lucru. Astfel, dacă $M[i][j] = 1$, atunci:

$$dp[i][j] = 1 + \min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])$$

Dacă $M[i][j] = 0$, evident $dp[i][j] = 0$. Această relație reflectă clar substructura optimă și dependența bidimensională tipice Programării Dinamice.

Descrierea algoritmului

Implementarea folosește o matrice auxiliară dp de aceeași dimensiune cu cea originală, pentru a reține dimensiunile pătratelor posibile în fiecare punct. Pe măsură ce se parcurge matricea, se actualizează dimensiunea maximă și coordonatele corespunzătoare.

```
std::pair<int, std::pair<int, int>>
   submatricePatraticaMaximala(std::vector<std::vector<int>>&
   matrice) {
    int m = matrice.size();
    int n = matrice[0].size();
    std::vector<std::vector<int>> dp(m, std::vector<int>(n, 0));
    int dimensiuneMaxima = 0;
    std::pair<int, int> pozitieDreaptaJos;

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (matrice[i][j] == 1) {
                if (i == 0 || j == 0) {
```

```
                    dp[i][j] = 1;
                } else {
                    dp[i][j] = std::min({dp[i-1][j], dp[i][j-1],
                        dp[i-1][j-1]}) + 1;
                }

                if (dp[i][j] > dimensiuneMaxima) {
                    dimensiuneMaxima = dp[i][j];
                    pozitieDreaptaJos = {i, j};
                }
            }
        }
    }

    return {dimensiuneMaxima, pozitieDreaptaJos};
}
```

Explicația implementării

Rezultatul întors de funcție este o pereche formată din:

- $dimensiuneMaxima$ - latura pătratului maxim de 1-uri;
- $pozitieDreaptaJos$ - coordonatele colțului din dreapta-jos al acestei submatrice.

Dacă se dorește reconstrucția efectivă a submatricei, aceasta se poate extrage direct din matricea inițială pornind de la aceste coordonate.

Exemplu de aplicare

Pentru matricea:

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

rezultatul algoritmului va fi o submatrice pătratică de dimensiune 3×3, situată cu colțul dreapta-jos în poziția $(1, 2)$. Aceasta corespunde pătratului format din secțiunea:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Complexitate și analiză

Complexitatea temporală a algoritmului este $O(m \cdot n)$, întrucât fiecare element al matricei este procesat o singură dată, iar complexitatea spațială este $O(m \cdot n)$ datorită matricii auxiliare dp. Această complexitate poate fi redusă la $O(n)$ prin folosirea a două rânduri succesive, dacă se dorește o implementare optimizată în spațiu.

Problema matricei binare maximale ilustrează forța Programării Dinamice în gestionarea datelor bidimensionale. Ea oferă o soluție elegantă și eficientă pentru o problemă aparent combinatorială, demonstrând cum memorarea parțială a rezultatelor locale duce la o reducere drastică a timpului de calcul. Astfel de abordări sunt frecvent întâlnite în analiza imaginilor, recunoașterea formelor și optimizările geometrice.

15.2.4 Dynamic Time Warping (DTW)

Algoritmul Dynamic Time Warping (DTW) reprezintă o aplicație avansată a Programării Dinamice, destinată măsurării similarității dintre două secvențe de timp care pot diferi în viteză, fază sau lungime. Spre deosebire de metodele tradiționale de comparare bazate pe distanța euclidiană, DTW permite o potrivire elastică între punctele celor două secvențe, permițând comprimarea sau dilatarea axei temporale.

Această proprietate îl face extrem de util pentru analiza seriilor de date care nu sunt perfect sincronizate, de exemplu, două înregistrări audio ale aceluiași cuvânt pronunțat cu viteze diferite, sau două mișcări fizice executate la ritmuri distincte.

Modelul problemei

Fie două secvențe temporale:

$$X = (x_1, x_2, \ldots, x_n), \quad Y = (y_1, y_2, \ldots, y_m)$$

Scopul este de a alinia aceste secvențe astfel încât distanța totală dintre ele, măsurată de-a lungul unei traiectorii de potrivire (numită *warping path*), să fie minimă.

Matricea distanțelor locale

Primul pas constă în construirea unei matrici a distanțelor locale, notată D, de dimensiune $n \times m$. Fiecare element $D(i, j)$ din această matrice reprezintă costul de a alinia punctul x_i din secvența X cu punctul y_j din secvența Y. Cel mai comun mod de a calcula această distanță locală este:

$$D(i, j) = \|x_i - y_j\|$$

unde $\|\cdot\|$ este o funcție de distanță adecvată pentru tipul de date al elementelor x_i și y_j. Exemple comune includ distanța euclidiană pentru vectori numerici sau distanța Manhattan.

Matricea costurilor cumulate

Pentru a identifica traiectoria de potrivire optimă, se construiește o matrice a costurilor cumulate, notată C, de dimensiune $n \times m$. Fiecare element $C(i, j)$ din această matrice reprezintă costul minim de aliniere a subsecvențelor $X[1 \dots i]$ și $Y[1 \dots j]$. Aceasta este calculată recursiv, folosind principiile Programării Dinamice, conform următoarei relații:

$$C(i, j) = D(i, j) + \min\{C(i-1, j),\ C(i, j-1),\ C(i-1, j-1)\}$$

unde:

- $C(i-1, j)$ reprezintă costul de a extinde alinierea prin adăugarea lui y_j la $X[1 \dots i-1]$. Acest lucru poate fi interpretat ca o "inserție" în X sau o "ștergere" în Y.
- $C(i, j-1)$ reprezintă costul de a extinde alinierea prin adăugarea lui x_i la $Y[1 \dots j-1]$. Acest lucru poate fi interpretat ca o "ștergere" în X sau o "inserție" în Y.
- $C(i-1, j-1)$ reprezintă costul de a extinde alinierea prin potrivirea simultană a lui x_i cu y_j, o "potrivire diagonală".

Condițiile inițiale pentru matricea costurilor cumulate sunt esențiale pentru a asigura o traiectorie de potrivire validă:

$$\begin{aligned} C(0,0) &= 0 \\ C(i,0) &= \infty \quad \text{pentru } i > 0 \\ C(0,j) &= \infty \quad \text{pentru } j > 0 \end{aligned}$$

Această inițializare impune ca traiectoria de potrivire să înceapă de la $(1,1)$ (primul element al ambelor secvențe) și să se termine la (n,m) (ultimul element al ambelor secvențe). Valorile de ∞ (infinit) pentru $C(i,0)$ și $C(0,j)$ previn alinierea cu o secvență goală, cu excepția punctului de start.

Distanța DTW finală

După ce întreaga matrice a costurilor cumulate C este completată prin iterarea de la $(1,1)$ până la (n,m), valoarea finală a distanței DTW minime dintre cele două secvențe X și Y va fi dată de elementul din colțul din dreapta jos al matricei:

$$DTW(X,Y) = C(n,m)$$

Această valoare reprezintă costul minim total al traiectoriei de potrivire optime.

Reconstruirea traiectoriei de potrivire (pas opțional)

Deși $C(n,m)$ oferă distanța minimă, în multe aplicații este la fel de important să se reconstruiască și traiectoria de potrivire efectivă (*warping path*). Aceasta se realizează printr-un proces de urmărire inversă din celula (n,m) până la $(1,1)$. La fiecare pas, se alege celula precedentă (i',j') care a contribuit la valoarea minimă a costului curent $C(i,j)$.

Implementarea algoritmului DTW

```
double dtwDistance(const std::vector<double>& seq1, const
   std::vector<double>& seq2) {

    int n = seq1.size();
    int m = seq2.size();
```

```
    // Matricea distantelor elementare
    std::vector<std::vector<double>> distante(n,
       std::vector<double>(m, 0.0));

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            distante[i][j] = abs(seq1[i] - seq2[j]);
        }
    }

    // Matricea costurilor cumulative
    std::vector<std::vector<double>> cost(n,
       std::vector<double>(m, 0.0));
    cost[0][0] = distante[0][0];

    // Initializarea primei coloane
    for (int i = 1; i < n; i++) {
        cost[i][0] = cost[i-1][0] + distante[i][0];
    }

    // Initializarea primei linii
    for (int j = 1; j < m; j++) {
        cost[0][j] = cost[0][j-1] + distante[0][j];
    }

    // Calculul costurilor cumulative
    for (int i = 1; i < n; i++) {
        for (int j = 1; j < m; j++) {
            cost[i][j] = distante[i][j] + min({
                cost[i-1][j],    // insertie
                cost[i][j-1],    // stergere
                cost[i-1][j-1]   // potrivire
            });
        }
    }

    return cost[n-1][m-1];
}
```

Explicația implementării

Algoritmul construiește pas cu pas o matrice de costuri cumulative, în care fiecare element `cost[i][j]` reprezintă costul minim de potrivire între primele i elemente din prima secvență și primele j elemente din cea de-a doua. Prin adunarea incrementală a celor mai mici

costuri din vecinătățile permise, DTW identifică o cale optimă care aliniază cele două serii de date.

Exemplu ilustrativ

Considerăm secvențele:

$$X = (1, 3, 4, 9), \quad Y = (1, 2, 3, 4, 8)$$

Rezultatul obținut prin DTW va reflecta o potrivire între puncte nealiniate temporal, permițând o distanță totală mai mică decât în cazul unei alinieri rigide.

Complexitate și analiză

Complexitatea temporală a algoritmului este $O(n \cdot m)$, iar cea spațială este de asemenea $O(n \cdot m)$, din cauza matricii bidimensionale de costuri. Totuși, în implementări avansate, spațiul poate fi redus la $O(\min(n, m))$, deoarece fiecare pas depinde doar de linia sau coloana anterioară.

Exemplu numeric

Fie două secvențe numerice cu următoarele valori:

- $referinta = \{0, 0, 1, 0, 0, -1, 4, -3, 0, 0, 2, 0, 0\}$
- $test = \{0, 0, 0, 1, 0, 0, 0, -1, 4, 1, 3, 2, 2, 2, 0, 0, 0\}$

Aceste secvențe prezintă lungimi diferite, ceea ce face ca utilizarea DTW pentru a identifica alinierea optimă între ele să fie o alegere potrivită.

Aplicarea algoritmului DTW presupune următorii pași:

- Pasul 1: Construirea matricei distanțelor

  Se calculează distanța dintre fiecare element din prima secvență, $referinta$, și fiecare element din cea de a doua secvență, $test$, valori ce sunt memorate într-o matrice a distanțelor D.

Fie $referinta[i]$ și $test[j]$ două elemente din prima, respectiv a doua secvență, elementul D(i, j) al matricei distanțelor este definit ca:

$$D(i,j) = (referinta[i] - test[j])^2$$

Rezultatul este o matrice de dimensiune 13×17.

Popularea matricei distanțelor pentru exemplul dat se realizează astfel:

$$\begin{aligned} D(0,0) =& (referinta[0] - test[0])^2 \\ =& (0-0)^2 \\ =& 0^2 \\ =& 0 \end{aligned}$$

3				
2				
1				
0	0			
	0	1	2	3

$$\begin{aligned} D(2,1) =& (referinta[2] - test[1])^2 \\ =& (1-0)^2 \\ =& 1^2 \\ =& 1 \end{aligned}$$

3				
2	1	1		
1	0	0	0	1
0	0	0	0	1
	0	1	2	3

În final matricea distanțelor este:

12	0	0	0	1	0	0	0	1	16	1	9	4	4	4	0	0	0
11	0	0	0	1	0	0	0	1	16	1	9	4	4	4	0	0	0
10	4	4	4	1	4	4	4	9	4	1	1	0	0	0	4	4	4
9	0	0	0	1	0	0	0	1	16	1	9	4	4	4	0	0	0
8	0	0	0	1	0	0	0	1	16	1	9	4	4	4	0	0	0
7	9	9	9	16	9	9	9	4	49	16	36	25	25	25	9	9	9
6	16	16	16	9	16	16	16	25	0	9	1	4	4	4	16	16	16
5	1	1	1	4	1	1	1	0	25	4	16	9	9	9	1	1	1
4	0	0	0	1	0	0	0	1	16	1	9	4	4	4	0	0	0
3	0	0	0	1	0	0	0	1	16	1	9	4	4	4	0	0	0
2	1	1	1	0	1	1	1	4	9	0	4	1	1	1	1	1	1
1	0	0	0	1	0	0	0	1	16	1	9	4	4	4	0	0	0
0	0	0	0	1	0	0	0	1	16	1	9	4	4	4	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Tabelul 15.1: Matricea distanțelor raportat la $referinta$ și $test$

- Pasul 2: Construirea matricei costului cumulat

Pe baza matricei distanțelor, se construiește matricea costului cumulat, care ajută la determinarea aliniamentului optim.

După cum am precizat anterior, elementele acestei matrice, C(i,j), este definit astfel:

- Primul element al matricei: $C(0,0) = D(0,0)$
- Pentru elementele aflate pe coloana de indice $j = 0$:

$$C(i,0) = D(i,0) + C(i-1,0)$$

- Pentru elementele aflate pe linia de indice $i = 0$:

$$C(0,j) = D(0,j) + C(0,j-1)$$

- Pentru toate celelate elemente:

$$C(i,j) = D(i,j) + min \begin{cases} C(i,j-1) & (stergere) \\ C(i-1,j-1) & (potrivire) \\ C(i-1,j) & (insertie) \end{cases}$$

Cu alte cuvinte, costul cumulat reprezintă suma distanței curente și a celui mai mic cost cumulat din vecinătatea celulei (i, j).

Popularea matricei costurilor cumulate pentru exemplul dat se realizează astfel:

$$C(0,0) = D(0,0)$$

3				
2				
1				
0	0			
	0	1	2	3

$$\begin{aligned} C(0,1) =& D(0,1) + C(0,1) \\ =& 0 + 0 \\ =& 0 \end{aligned}$$

3				
2				
1				
0	0	0		
	0	1	2	3

Matricea costurilor cumulate este:

$$C(2,0) = D(2,0) + C(1,0)$$
$$= 1 + 0$$
$$= 1$$

3				
2	1			
1	0	0	0	1
0	0	0	0	1
	0	1	2	3

$$C(2,1) = D(2,1) + min(1,0,0)$$
$$= 1 + 0$$
$$= 1$$

3				
2	1	1		
1	0	0	0	1
0	0	0	0	1
	0	1	2	3

$$C(10,7) = D(10,7) + min(24,20,21)$$
$$= 9 + 20$$
$$= 29$$

24	24	29	
20	20	21	37
20	20	21	37

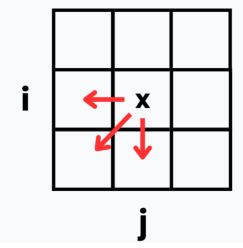
$$C(12,16) = D(12,16) + min(19,19,19)$$
$$= 0 + 19$$
$$= 19$$

27	19	19	19
23	19	19	19
19	23	27	31
37	31	31	31

- Pasul 3: Determinarea alinierii optime

  Alinierea optimă între cele două secvențe este determinată prin parcurgerea matricei costului cumulat de la poziția [12, 16] (ultimul element al matricei) la [0, 0] (primul element al matricei), selectând mereu celulele care contribuie la minimizarea costului cumulat.

  Din poziția $[i, j]$ se urmează direcția cu cel mai mic cost cumulativ din cele 3 posibile:
  - celula din stânga $(C(i, j-1))$,
  - celula de jos $(C(i-1, j))$,
  - celula din diagonală stânga-jos $(C(i-1, j-1))$.



- Pasul 4: Interpretarea rezultatelor

  După cum se poate observa și în figura 15.10, din drumul optim deducem:

  - Traseul roșu arată că elementele secvenței 1 sunt potrivite cu elemente ale secvenței 2 chiar dacă cele două secvențe prezintă lungimi diferite (13 vs 17).

12	31	31	31	30	24	24	24	25	41	21	29	27	27	27	19	19	19
11	31	31	31	29	24	24	24	25	41	20	28	23	23	23	19	19	19
10	31	31	31	28	24	24	24	29	25	19	19	19	19	19	23	27	31
9	27	27	27	28	20	20	20	21	37	18	26	29	33	37	31	31	31
8	27	27	27	28	20	20	20	21	37	17	25	29	33	37	31	31	31
7	27	27	27	27	20	26	26	21	49	16	45	35	39	43	31	40	44
6	18	18	18	11	17	17	17	25	0	9	10	14	18	22	35	35	35
5	2	2	2	5	1	1	1	0	25	17	29	30	32	32	19	19	19
4	1	1	1	2	0	0	0	1	17	13	21	23	23	24	18	18	18
3	1	1	1	1	0	0	0	1	17	12	20	19	20	21	18	18	18
2	1	1	1	0	1	2	2	5	11	11	15	16	17	18	19	20	21
1	0	0	0	1	1	1	1	2	18	19	28	32	36	40	40	40	40
0	0	0	0	1	1	1	1	2	18	19	28	32	36	40	40	40	40
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Tabelul 15.2: Matricea costurilor cumulate raportat la $referinta$ și $test$

- Mișcările orizontale și verticale din traseu arată că anumite valori din a doua secvență sunt „întinse” sau „comasate” pentru a corespunde valorilor din prima secvență.
- Pașii diagonali indică alinierea unu-la-unu între două valori similare din cele două secvențe.
- Costul alinierii celor două grafice este egal cu valoarea elementului de coordonate (12, 16) din matrice costurilor cumulate, adică 19.

Urmând calea minimă, evidențiată prin linia roșie în figura 15.10, putem identifica perechile de puncte care sunt aliniate:

Algoritmul DTW este utilizat într-o gamă variată de domenii:

- recunoașterea vocală (compararea semnalelor audio);
- analiza seriilor temporale financiare (compararea evoluției prețurilor);
- biologie computațională (alinierea secvențelor de expresii genetice);
- robotică și analiză de mișcare (compararea traiectoriilor).

DTW exemplifică eleganța și puterea Programării Dinamice în analiza seriilor temporale complexe. Prin construirea incrementală a unei matrici de costuri și identificarea unei traiectorii minime, algoritmul oferă o metodă robustă pentru compararea secvențelor de lungimi și viteze

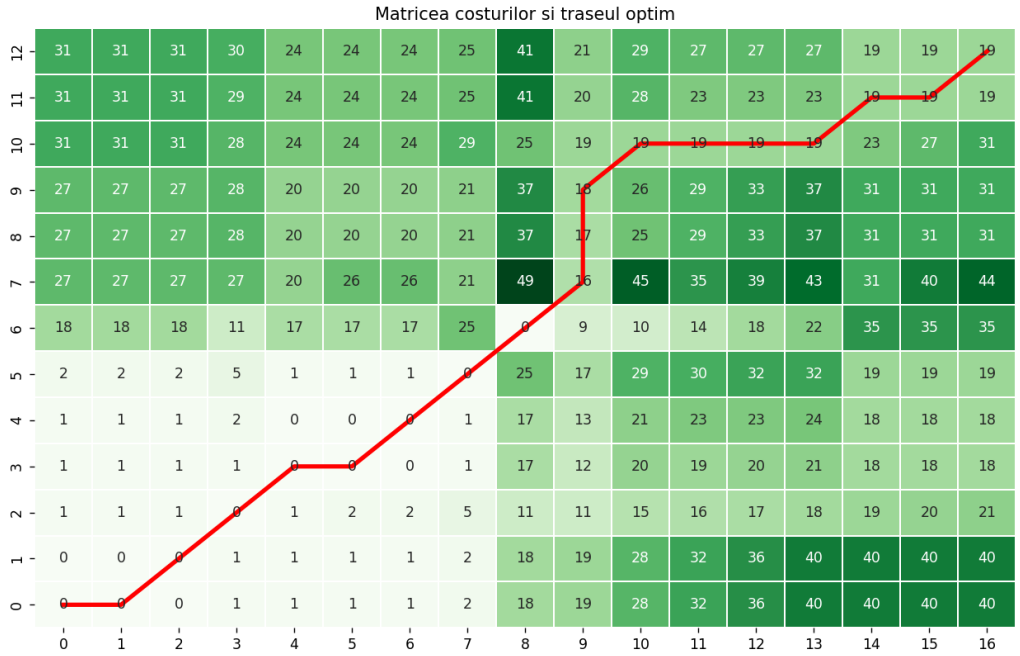


Figura 15.10: Drum optim

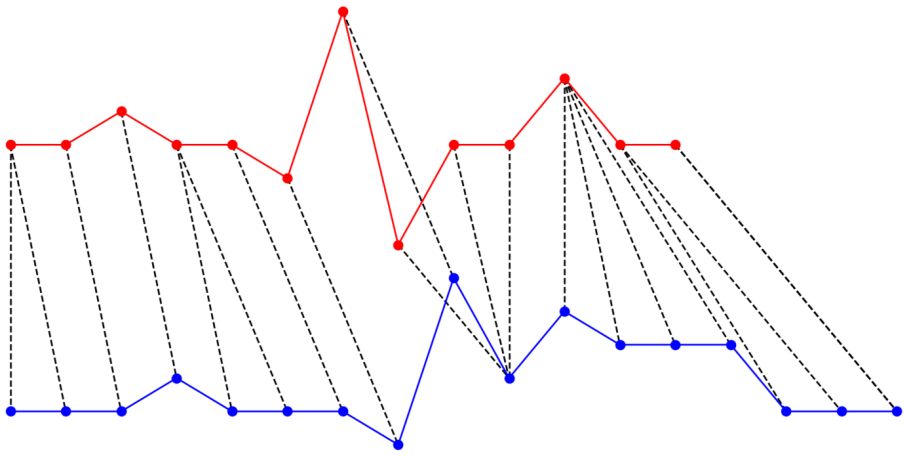


Figura 15.11: Alinierea celor două grafice

diferite. Această abordare a devenit un standard în prelucrarea semnalelor și analiza datelor neliniare.

15.2.5 Concluzii

Programarea Dinamică reprezintă una dintre pietrele de temelie ale informaticii teoretice și aplicate, unind eleganța matematică cu pragmatismul algoritmic. Prin reducerea problemelor complexe la o succesiune de subprobleme gestionabile și prin reutilizarea inteligentă a rezultatelor intermediare, această tehnică oferă un cadru riguros pentru obținerea soluțiilor optime în contexte în care alte metode eșuează.

Importanța Programării Dinamice nu se rezumă la eficiența sa computațională, ci constă în modul în care disciplinează gândirea algoritmică: ea forțează proiectantul să identifice structura internă a problemei, să descopere relațiile recurente și să formalizeze procesul decizional într-o manieră transparentă și logică.

Comparație cu alte tehnici de programare

Programarea Dinamică se deosebește semnificativ de alte paradigme de proiectare a algoritmilor, prin modul său sistematic de abordare a subproblemelor. Comparată cu tehnica Greedy, diferența esențială constă în faptul că PD explorează toate combinațiile posibile ale deciziilor, asigurând întotdeauna obținerea soluției optime globale. În schimb, abordarea Greedy urmărește alegerea unei soluții locale considerate optime la fiecare pas, fără garanția că rezultatul final va fi cel mai bun posibil. Astfel, PD oferă precizie și siguranță în detrimentul timpului de execuție, pe când algoritmii Greedy se disting prin rapiditate, însă pot conduce la soluții suboptime.

În comparație cu paradigma *Divide et Impera*, Programarea Dinamică se concentrează pe problemele care prezintă subprobleme suprapuse. În timp ce *Divide et Impera* presupune divizarea unei probleme în subprobleme independente, rezolvate separat și combinate ulterior, PD evită recalcularea acelorași soluții prin memorarea rezultatelor intermediare. Astfel, Programarea Dinamică se dovedește superioară atunci când există redundanță în subprobleme, oferind un mod eficient de reutilizare a informației deja obținute.

Avantaje și dezavantaje

Principalul avantaj al Programării Dinamice constă în eficiența sa remarcabilă. Prin evitarea recalculării subproblemelor, se obține o reducere semnificativă a complexității temporale, transformând unele probleme considerate anterior imposibile într-un interval de timp rezonabil.

În plus, tehnica garantează obținerea soluției optime, aspect esențial în probleme de optimizare globală. Structura sa sistematică contribuie, de asemenea, la o înțelegere clară a procesului de rezolvare și la o modularitate crescută a algoritmului, făcând analiza și verificarea mai riguroase.

Totuși, aceste beneficii vin însoțite de anumite limitări. Costul de memorie poate fi ridicat, mai ales în cazul problemelor cu un număr mare de dimensiuni sau de stări posibile. Identificarea relațiilor de recurență și a substructurii optimale necesită o analiză atentă și o bună înțelegere a problemei. În plus, pentru situațiile simple, aplicarea PD poate fi excesivă, deoarece introduce un nivel suplimentar de complexitate conceptuală și implementare.

Domenii de aplicare

Programarea Dinamică și-a demonstrat eficiența într-o varietate de domenii științifice și tehnice. În bioinformatică, este utilizată pentru alinierea secvențelor de ADN, unde compararea șirurilor genetice necesită evaluarea tuturor potrivirilor posibile. În procesarea limbajului natural, tehnica este esențială pentru analiza sintactică și semantică, permițând recunoașterea structurilor lingvistice complexe.

În inteligența artificială, PD constituie fundamentul multor algoritmi de planificare și decizie secvențială, cum ar fi algoritmii de învățare prin întărire. În domeniul financiar, contribuie la optimizarea portofoliilor și la gestionarea riscurilor. În rețelele de calculatoare, este utilizată pentru determinarea rutelor optime, iar în robotică, pentru planificarea mișcărilor și optimizarea traseelor. Această diversitate de aplicații evidențiază versatilitatea și robustețea metodei.

15.3 Probleme rezolvate

În secțiunea care urmează propunem o serie de probleme cu Programare Dinamică:

1. Se dă un vector de numere întregi. Să se determine lungimea celei mai lungi subsecvențe strict crescătoare (elementele subsecvenței nu trebuie să fie neapărat consecutive în vector).

   Exemplu: Se citește vectorul $\{10, 22, 9, 33, 21, 50, 41, 60\}$. Cea mai lungă subsecvență strict crescătoare este $\{10, 22, 33, 50, 60\}$, având lungimea 5.

Rezolvare:

Date de intrare:

- un număr întreg n - numărul de elemente din vector;
- un vector v cu n numere întregi.

Date de ieșire:

- lungimea celei mai lungi subsecvențe strict crescătoare.

Algoritm de rezolvare prin Programare Dinamică:

(a) Se definește un vector dp, unde $dp[i]$ reprezintă lungimea celei mai lungi subsecvențe strict crescătoare care se termină pe poziția i.

(b) Inițial, fiecare element formează singur o subsecvență, astfel încât $dp[i] = 1$, pentru orice i.

(c) Pentru fiecare poziție i, se verifică toate pozițiile anterioare $j < i$.

(d) Dacă $v[j] < v[i]$, atunci elementul $v[i]$ poate continua subsecvența care se termină în j. Se actualizează:

$$dp[i] = \max(dp[i], dp[j] + 1).$$

(e) Răspunsul final este valoarea maximă din vectorul dp.

Algoritm scris în C++:

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    int n;
    std::cin >> n;
    std::vector<int> v(n);
    for (int i = 0; i < n; i++)
        std::cin >> v[i];

    std::vector<int> dp(n, 1);
    int maxim = 1;

    for (int i = 1; i < n; i++) {
```

```
        for (int j = 0; j < i; j++) {
            if (v[j] < v[i]) {
                dp[i] = std::max(dp[i], dp[j] + 1);
            }
        }
        maxim = std::max(maxim, dp[i]);
    }

    std::cout << maxim;
    return 0;
}
```

Justificarea soluției alese:

Pentru fiecare poziție i, lungimea celei mai lungi subsecvențe strict crescătoare care se termină în această poziție depinde exclusiv de soluțiile calculate pentru pozițiile anterioare. Astfel, problema prezintă proprietatea de substructură optimă și poate fi rezolvată eficient prin programare dinamică. Fiecare valoare $dp[i]$ este calculată o singură dată folosind rezultate deja determinate, evitând recalculările inutile.

Analiza complexității:

- *Complexitate temporală*:
  - Inițializarea vectorului dp: $O(n)$;
  - Cele două bucle imbricate: $O(n^2)$;
  - Determinarea răspunsului este inclusă în ultima parcurgere.

  Complexitatea totală este: $O(n^2)$.
- *Complexitate spațială*:
  - Vectorul de intrare: $O(n)$;
  - Vectorul dp: $O(n)$.

  Complexitatea spațială suplimentară este: $O(n)$.

Demonstrație:

Vom demonstra prin inducție că după calcularea valorii $dp[i]$, aceasta reprezintă lungimea celei mai lungi subsecvențe strict crescătoare care se termină în poziția i.

Cazul de bază: Pentru $i = 0$, singura subsecvență care se termină în primul element este chiar elementul respectiv, deci:

$$dp[0] = 1.$$

Afirmația este adevărată.

Ipoteza de inducție: Presupunem că pentru toate pozițiile

$$0, 1, \ldots, i-1,$$

valorile dp au fost calculate corect.

Pasul inductiv: Pentru poziția i, algoritmul verifică toate pozițiile anterioare $j < i$. Dacă

$$v[j] < v[i],$$

atunci orice subsecvență optimă care se termină în j poate fi extinsă cu elementul $v[i]$, obținând o subsecvență de lungime

$$dp[j] + 1.$$

Se păstrează valoarea maximă dintre toate aceste posibilități:

$$dp[i] = \max(dp[i], dp[j] + 1).$$

Dacă nu există niciun indice j pentru care

$$v[j] < v[i],$$

atunci subsecvența optimă este formată doar din elementul $v[i]$, iar

$$dp[i] = 1.$$

Prin urmare, $dp[i]$ reprezintă exact lungimea celei mai lungi subsecvențe strict crescătoare care se termină în poziția i.
Cum răspunsul problemei este cea mai mare valoare din vectorul dp, algoritmul determină corect lungimea celei mai lungi subsecvențe strict crescătoare.

2. Se dau două șiruri de caractere. Să se determine lungimea celei mai lungi subsecvențe comune celor două șiruri (caracterele unei subsecvențe nu trebuie să fie neapărat consecutive).
Exemplu: Se citesc șirurile `"ABCDGH"` și `"AEDFHR"`. Cea mai lungă subsecvență comună este `"ADH"`, iar lungimea acesteia este 3.

Rezolvare:
Date de intrare:

- două șiruri de caractere A și B;
- n - lungimea primului șir;
- m - lungimea celui de-al doilea șir.

Date de ieșire:

- lungimea celei mai lungi subsecvențe comune celor două șiruri.

Algoritm de rezolvare prin Programare Dinamică:

(a) Se construiește o matrice dp de dimensiune $(n+1) \times (m+1)$.

(b) Elementul $dp[i][j]$ reprezintă lungimea celei mai lungi subsecvențe comune dintre primele i caractere din șirul A și primele j caractere din șirul B.

(c) Se inițializează prima linie și prima coloană cu valoarea 0, deoarece un șir vid nu are subsecvențe comune cu niciun alt șir.

(d) Pentru fiecare pereche de poziții (i, j):

- dacă $A[i-1] = B[j-1]$, atunci caracterul comun poate fi adăugat la soluția anterioară:

$$dp[i][j] = dp[i-1][j-1] + 1;$$

- în caz contrar:

$$dp[i][j] = \max(dp[i-1][j], dp[i][j-1]).$$

(e) Rezultatul final este valoarea $dp[n][m]$.

Algoritm scris în C++:

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

int main() {
    std::string A, B;
    std::cin >> A >> B;

    int n = A.size();
    int m = B.size();

    std::vector<std::vector<int>> dp(n + 1,
       std::vector<int>(m + 1, 0));

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (A[i - 1] == B[j - 1])
                dp[i][j] = dp[i - 1][j - 1] + 1;
            else
                dp[i][j] = std::max(dp[i - 1][j], dp[i][j -
                   1]);
        }
    }

    std::cout << dp[n][m];

    return 0;
}
```

Justificarea soluției alese:

Problema prezintă proprietatea de substructură optimă. Lungimea celei mai lungi subsecvențe comune pentru două prefixe depinde doar de soluțiile calculate pentru prefixe mai mici. Dacă ultimele caractere coincid, acestea fac parte din subsecvența optimă; în caz contrar, una dintre cele două poziții trebuie eliminată, alegând varianta care produce

subsecvența de lungime maximă. Programarea dinamică evită recalcularea acelorași subprobleme și determină soluția optimă.

Analiza complexității:

- *Complexitate temporală*:
  - Inițializarea matricei: $O(nm)$;
  - Completarea matricei: $O(nm)$.

  Complexitatea totală este: $O(nm)$.
- *Complexitate spațială*:
  - Matricea dp: $O(nm)$.

  Complexitatea spațială este: $O(nm)$.

Demonstrație:
Vom demonstra prin inducție că valoarea $dp[i][j]$ reprezintă lungimea celei mai lungi subsecvențe comune dintre primele i caractere ale șirului A și primele j caractere ale șirului B.

Cazul de bază:
Dacă unul dintre șiruri este vid, atunci nu există nicio subsecvență comună. Prin urmare,

$$dp[0][j] = 0, \qquad dp[i][0] = 0,$$

pentru orice valori ale lui i și j.

Ipoteza de inducție:
Presupunem că pentru toate perechile de indici

$$(i', j'),$$

unde $i' < i$ sau $j' < j$, valorile din matrice au fost calculate corect.

Pasul inductiv:

Considerăm pozițiile i și j.

- Dacă
$$A[i-1] = B[j-1],$$
atunci orice subsecvență comună optimă trebuie să conțină acest caracter la final. Eliminând ultimul caracter din ambele șiruri, rămâne problema pentru prefixele anterioare, astfel încât:
$$dp[i][j] = dp[i-1][j-1] + 1.$$
- Dacă
$$A[i-1] \neq B[j-1],$$
atunci ultimul caracter al cel puțin unuia dintre șiruri nu poate aparține simultan unei subsecvențe comune optime. În consecință, soluția optimă este maximul dintre:
$$dp[i-1][j]$$
și
$$dp[i][j-1].$$

În ambele situații, valoarea calculată pentru $dp[i][j]$ este corectă.

Prin inducție rezultă că toate valorile matricei sunt calculate corect, iar elementul
$$dp[n][m]$$
reprezintă lungimea celei mai lungi subsecvențe comune dintre cele două șiruri.

3. Se dă o matrice cu n linii și m coloane, fiecare element reprezentând un cost pozitiv. Pornind din colțul din stânga sus, se poate merge doar în jos sau spre dreapta. Să se determine costul minim al unui drum până în colțul din dreapta jos.

Exemplu:

$$\begin{bmatrix} 1 & 3 & 1 \\ 1 & 5 & 1 \\ 4 & 2 & 1 \end{bmatrix}$$

Costul minim este 7, corespunzător drumului: $1 \to 3 \to 1 \to 1 \to 1$.

Rezolvare:

Date de intrare:

- două numere întregi n și m – numărul de linii, respectiv numărul de coloane ale matricei;
- o matrice $cost$ de dimensiune $n \times m$, ale cărei elemente reprezintă costurile fiecărei celule.

Date de ieșire:

- costul minim al unui drum de la poziția $(1, 1)$ la poziția (n, m), deplasându-ne doar spre dreapta sau în jos.

Algoritm de rezolvare prin Programare Dinamică:

(a) Se construiește o matrice dp de dimensiune $n \times m$.

(b) Elementul $dp[i][j]$ reprezintă costul minim necesar pentru a ajunge în celula (i, j).

(c) Se inițializează:

- $dp[0][0] = cost[0][0]$;
- prima linie, deoarece deplasarea este posibilă doar spre dreapta;
- prima coloană, deoarece deplasarea este posibilă doar în jos.

(d) Pentru fiecare celulă interioară:

$$dp[i][j] = cost[i][j] + \min(dp[i-1][j], dp[i][j-1]).$$

(e) Rezultatul final este valoarea $dp[n-1][m-1]$.

Algoritm scris în C++:

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    int n, m;
    std::cin >> n >> m;

    std::vector<std::vector<int>> cost(n,
       std::vector<int>(m));
    std::vector<std::vector<int>> dp(n, std::vector<int>(m));

    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            std::cin >> cost[i][j];

    dp[0][0] = cost[0][0];

    for (int j = 1; j < m; j++)
        dp[0][j] = dp[0][j - 1] + cost[0][j];

    for (int i = 1; i < n; i++)
        dp[i][0] = dp[i - 1][0] + cost[i][0];

    for (int i = 1; i < n; i++)
        for (int j = 1; j < m; j++)
            dp[i][j] = cost[i][j] +
                       std::min(dp[i - 1][j], dp[i][j - 1]);
    std::cout << dp[n - 1][m - 1];

    return 0;
}
```

Justificarea soluției alese: Pentru a ajunge într-o celulă (i, j) există doar două posibilități: din celula aflată deasupra $(i-1, j)$ sau din cea aflată la stânga $(i, j-1)$. Costul minim până în poziția curentă depinde exclusiv de costurile minime calculate anterior pentru aceste două celule. Problema are astfel proprietatea de substructură optimă și poate fi rezolvată eficient prin programare dinamică, fiecare subproblemă fiind calculată o singură dată.

Analiza complexității:

- *Complexitate temporală*:
  - Inițializarea primei linii și a primei coloane: $O(n + m)$;
  - Completarea matricei dp: $O(nm)$.

  Complexitatea totală este: $O(nm)$.
- *Complexitate spațială*:
  - Matricea costurilor: $O(nm)$;
  - Matricea dp: $O(nm)$.

  Complexitatea spațială suplimentară este: $O(nm)$.

Demonstrație:
Vom demonstra prin inducție că valoarea $dp[i][j]$ reprezintă costul minim necesar pentru a ajunge în celula (i, j).

Cazul de bază:
Pentru prima celulă,

$$dp[0][0] = cost[0][0],$$

deoarece drumul începe în această poziție.
Pentru prima linie există un singur drum posibil, deplasându-ne numai spre dreapta, astfel:

$$dp[0][j] = dp[0][j-1] + cost[0][j].$$

În mod similar, pentru prima coloană există un singur drum posibil, deplasându-ne numai în jos:

$$dp[i][0] = dp[i-1][0] + cost[i][0].$$

Ipoteza de inducție:
Presupunem că pentru toate celulele situate înaintea poziției (i, j), valorile din matricea dp reprezintă costurile minime corecte.

Pasul inductiv:

Pentru a ajunge în celula (i, j) există exact două posibilități:

- din celula de deasupra $(i-1, j)$;
- din celula din stânga $(i, j-1)$.

Conform ipotezei de inducție, costurile minime până în aceste două celule sunt deja calculate corect. Alegând varianta cu cost minim și adăugând costul celulei curente, obținem:

$$dp[i][j] = cost[i][j] + \min(dp[i-1][j], dp[i][j-1]).$$

Nu poate exista un drum cu cost mai mic, deoarece orice drum către (i, j) trebuie să treacă obligatoriu prin una dintre cele două celule menționate.
Prin inducție rezultă că toate valorile matricei dp sunt calculate corect, iar valoarea

$$dp[n-1][m-1]$$

reprezintă costul minim al unui drum de la colțul din stânga sus la colțul din dreapta jos.

15.4 Probleme propuse

***

1. Suntem vânzători la un magazin de materiale textile. Un client vine să ceară material pentru croitorie de lungime n dar nu îl interesează dacă materialul este tăiat în bucățele mai scurte sau mai lungi. La magazinul nostru avem o listă de prețuri pentru fiecare tăietura posibilă în parte (de exemplu pentru a tăia o bucată de 1 metru cerem 1 leu, pentru o altă bucată de 2 metri cerem 3 lei etc, nu există o legătură proporțională între prețuri și lungimi). Având această listă de prețuri să se determine modul de tăiere în părți mai mici a materialului astfel încât să obținem cel mai mult profit la final.
   Exemplu: Clientul cere o pânză de 4 metri ($n = 4$) iar lista noastră de prețuri pentru diferite mărimi este următoarea: [1, 5, 8, 9, 10, 17, 17, 20]. Cel mai mult profit îl obținem dacă vindem 2 bucăți de câte 2 metri (5 + 5 = 10).

2. Se dă o scară ce are un număr „distanta” de trepte. Se cere să determinăm numărul total de moduri de a urca scara pana în vârf ținând cont că putem acoperi la un moment dat 1, 2 sau 3 trepte.

   Exemplu: Se dă numărul de trepte 4. Rezultatul afișat este: 7 deoarece putem parcurge cele 4 trepte în următoarele 7 moduri: 1+1+1+1, 1+2+1, 2+1+1, 1+1+2, 2+2, 1+3, 3+1.

   $[i0, i1, i2, i3, i4] -> [0, 1, 2, 4]$

   x[3-1] + x[3-2] + x[3-3] = 2+1+0

   1+1+1

   1+2

   2+1

   3

3. Se primește un șir de numere întregi în care fiecare element reprezintă numărul maxim de pași care pot fi înaintați de pe acea poziție. Scrieți o funcție care să returneze numărul minim de salturi pentru a ajunge la sfârșitul vectorului (începând de la primul element). Dacă un element este 0, nu se pot deplasa prin acel element. Dacă sfârșitul nu este accesibil, returnați -1.

   Exemplu: Se citește vectorul format din 1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9. Așadar, de pe prima poziție putem înainta 1 element și anume pe poziția 2. De pe a doua poziție putem înainta până la 3 poziții adică pe pozițiile 3, 4 sau 5. Rezultatul final va fi 3 (numărul minim de salturi necesare să ajungem până la final): $1 -> 3 -> 9 -> 9$.

   $[1, 3, 5, 8, 1, 2, 6, 7, 6, 8, 9]$

   $[0, 1, 2, 2, 2, 3, 3, 3, 3, 3, 3]$

   $[0, m, m, m, m, m, m, m, m, m, m]$

4. Pentru un număr dat $n > 0$, găsiți numărul de moduri diferite în care n poate fi scris ca o sumă a două sau mai multe numere întregi pozitive.

   Exemplu: Se citește $n = 5$. Rezultatul este 6: 1+1+1+1+1, 2+1+1+1, 2+2+1, 3+1+1, 3+2, 4+1.

5. Se dă un vector de ce reprezintă o colecție nelimitată de bancnote cu diferite valori. Se citește o sumă X de la tastatură. Să se determine numărul minim de bancnote din care

poate fi formată suma X citită pe baza vectorului inițial.

Exemplu: Se citește vectorul $\{10, 6, 1\}$ și suma 18. Se va afișa 3.

****

1. Se dă o stradă cu n case construite în linie. Fiecare dintre aceste case are o anumita valoare atribuită obiectelor din ea. Un hoț dorește să fure valoarea maximă posibilă de pe această stradă, însă el nu poate fura din două case adiacente deoarece proprietarul unei case furate le va spune celor doi vecini, din stânga și din dreapta, iar astfel va fi prins. Se cere a determina această valoare maximă posibilă de furat.

   Exemplu: Se dau valorile cumulate a obiectelor din fiecare casa: $\{6, 7, 1, 3, 8, 2, 4\}$. Valoarea maximă de furat fără a fi prins este 19 = 6 + 1 + 8 + 4. Se dau valorile cumulate a obiectelor din fiecare casă: $\{5, 3, 4, 11, 2\}$. Valoarea maxima de furat fără a fi prins este 16 = 5 + 11.

   Astfel, dacă $vect = [i_0, i_1, i_2, i_3, i_4, i_5] = [0, val[i1], max(val[i1], val[i2]), max(Vect[i(3-2)] + val[i3], Vect[i(3-1)])$

2. Fie un text de lungime maximă 100, ce conține doar litere. Textul poate fi codificat, înlocuind aparițiile consecutive ale subșirurilor sale cu subșirul urmat de numărul său de apariții. Să se scrie un program care să codifice optimal un text dat (codul rezultat să fie de lungime minimă).

   Exemplu: Textul T="aaacaaacbbdefdef" poate fi codificat: "a3c1a3c1b2def2" dar și "aaac2b2def2". Evident, cel de al doilea mod de codificare este mai scurt, deci mai convenabil.

3. Se dă un vector de numere întregi. Să se determine dacă este posibil ca vectorul să fie împărțit în doi vectori ce au suma elementelor egală.

   Exemplu: Se citește vectorul $\{3, 1, 1, 2, 2, 1\}$ acesta poate fi împărțit în $\{1, 1, 1, 2\}$ și $\{3, 2\}$ sau în alte moduri deci se afișează "Se poate împărți".

Bibliografie

[1] R. Andonie și I. Gârbacea, *Algoritmi fundamentali. O perspectivă C++*. Cluj-Napoca: Editura Libris, 1995.

[2] J. Farrier, *Data Structures and Algorithms with the C++ STL: A Guide for Modern C++ Practitioners*. Packt Publishing, 2024.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest și C. Stein, *Introduction to algorithms*. MIT press, 2022.

[4] M. Frenţiu, H. F. Pop și G. Şerban, *Programming Fundamentals*. Cluj-Napoca: Cluj University Press, 2006.

[5] M. T. Goodrich și R. Tamassia, *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons, 2002.

[6] S. Russell și P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd. Upper Saddle River, NJ: Prentice Hall, 2003.

[7] S. Arora și B. Barak, *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[8] I. C. Plajer și I. Popa, *Structuri de date. Curs pentru învățământ la distanță*. Editura Universității Transilvania din Brașov, 2025.

[9] E. W. Dijkstra, "ALGOL 60 Translation: An ALGOL 60 Translator for the X1 and Making a Translator for ALGOL 60," Mathematisch Centrum, Amsterdam, The Netherlands, rap. teh. MR 34/61, 1961.

[10] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1998.

[11] B. Shneiderman, "Jump searching: A fast sequential search technique," *Communications of the ACM*, vol. 21, nr. 10, pp. 831–834, 1978.

[12] W. Dobosiewicz, “An Efficient Variation of Bubble Sort,” *Information Processing Letters*, vol. 11, nr. 1, pp. 5–6, 1980.

[13] T. Peters, *Sorting Stability and Timsort*, Python-Dev Mailing List, 2002. adresa: `https://mail.python.org/pipermail/python-dev/2002-July/026839.html`.

[14] D. R. Musser, “Introspective sorting and selection algorithms,” *Software: Practice and Experience*, vol. 27, nr. 8, pp. 983–993, 1997.

[15] D. E. Knuth, *The Art of Computer Programming. Fundamental Algorithms*. Addison Wesley, 1997, vol. 1.

[16] G. Fowler, L. C. Noll, K.-P. Vo, D. Eastlake și T. Hansen, “The FNV Non-Cryptographic Hash Algorithm,” Internet Engineering Task Force (IETF), Internet-Draft draft-eastlake-fnv, 2024. adresa: `https://datatracker.ietf.org/doc/draft-eastlake-fnv/`.

[17] A. Karatsuba, “Multiplication of multidigit numbers on automata,” *Soviet Physics Doklady*, vol. 7, pp. 595–596, 1963.

[18] D. A. Huffman, “A method for the construction of minimum-redundancy codes,” *Proceedings of the IRE*, vol. 40, nr. 9, pp. 1098–1101, 1952.

[19] S. Russell, P. Norvig și A. Intelligence, “A modern approach,” *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, vol. 25, nr. 27, pp. 79–80, 1995.

[20] G. Adelson-Velsky și E. Landis, “An algorithm for the organization of information,” *Soviet Math. Dokl.*, vol. 3, pp. 1259–1263, 1962.

[21] L. J. Guibas și R. Sedgewick, “A dichromatic framework for balanced trees,” în *Proc. 19th Annual Symposium on Foundations of Computer Science (SFCS)*, IEEE, 1978, pp. 8–21.