

Ioana Cristina PLAJER

Iulian POPA

STRUCTURI DE DATE



Editura
Universității
Transilvania
din Brașov

2025

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 978-606-19-1820-1 (ebook)

Copyright © Autorul, 2025

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

Cuprins

Introducere	7
Test inițial	10
1 Structuri de date de bază	11
1.1 Unitatea 1 - Liste înlănțuite	13
1.1.1 Introducere	13
1.1.2 Obiective	13
1.1.3 Liste simplu înlănțuite	13
1.1.4 Liste dublu înlănțuite	17
1.1.5 Utilizarea listelor înlănțuite	21
1.1.6 Rezumat	23
1.1.7 Test de autoevaluare	23
1.1.8 Răspunsuri la testul de evaluare a cunoștințelor	24
1.2 Unitatea 2 - Stive și cozi	27
1.2.1 Introducere	27
1.2.2 Obiective	27
1.2.3 Stiva - <i>stack</i>	27
1.2.4 Coadă - <i>queue</i>	30
1.2.5 deque	34
1.2.6 Aplicații ce utilizează stive și cozi	36
1.2.7 Rezumat	36
1.2.8 Test de autoevaluare	37
1.2.9 Răspunsuri la testul de evaluare a cunoștințelor	37
2 Tabele de dispersie	41
2.1 Unitatea de învățare 1 - Tabele de dispersie cu liste înlănțuite	42
2.1.1 Introducere	42
2.1.2 Obiective	42

2.1.3	Tabele cu adresare directă	42
2.1.4	Tabele de dispersie - <i>Hash Tables</i>	44
2.1.5	Problema coliziunilor	45
2.1.6	Operațiile uzuale	46
2.1.7	Analiza complexității	47
2.1.8	Metode de dispersie	49
2.1.9	Metode dinamice	51
2.1.10	Rezumat	53
2.1.11	Test de autoevaluare	53
2.1.12	Răspunsuri la testul de evaluare a cunoștințelor	53
2.2	Unitatea de învățare 2 - Tabele de dispersie cu adresare deschisă	56
2.2.1	Introducere	56
2.2.2	Obiective	56
2.2.3	Testarea pozițiilor	57
2.2.4	<i>Perfect hashing</i>	62
2.2.5	Tratarea cheilor de tip vectorial (șir de caractere)	63
2.2.6	Rezumat	67
2.2.7	Test de autoevaluare	68
2.2.8	Răspunsuri la testul de evaluare a cunoștințelor	68
3	Arbori. Arbori binari	73
3.1	Unitatea de învățare 1 - Arbori	74
3.1.1	Introducere	74
3.1.2	Competențe	74
3.1.3	Definiții și notații	74
3.1.4	Reprezentarea arborilor	77
3.1.5	Parcurgerea arborilor	79
3.1.6	Parcurgerea arborilor binari	80
3.1.7	Rezumat	84
3.1.8	Test de autoevaluare	84
3.1.9	Răspunsuri la testul de evaluare a cunoștințelor	85
4	Heap. Coadă de priorități	88
4.1	Unitatea de învățare 1 - Heap	89
4.1.1	Introducere	89
4.1.2	Obiective	89
4.1.3	Algoritmul de sortare <i>HeapSort</i>	93

4.1.4	Rezumat	95
4.1.5	Test de autoevaluare	95
4.1.6	Răspunsuri la testul de evaluare a cunoștințelor	96
4.2	Unitatea de învățare 2 - Cozi de prioritate	99
4.2.1	Introducere	99
4.2.2	Obiective	99
4.2.3	Operații pe cozi de prioritate	100
4.2.4	Rezumat	103
4.2.5	Test de autoevaluare	103
4.2.6	Răspunsuri la testul de evaluare a cunoștințelor	103
5	Arbori binari de căutare	107
5.1	Unitatea de învățare 1 - Arbori binari de căutare	108
5.1.1	Introducere	108
5.1.2	Obiective	108
5.1.3	Noțiuni de bază	108
5.1.4	Operații într-un arbore binar de căutare	109
5.1.5	Rotăția într-un arbore binar de căutare	118
5.1.6	Rezumat	121
5.1.7	Test de autoevaluare	121
5.1.8	Răspunsuri la testul de evaluare a cunoștințelor	121
5.2	Unitatea de învățare 2 - Arbori AVL	126
5.2.1	Introducere	126
5.2.2	Obiective	126
5.2.3	Noțiuni generale	126
5.2.4	Operații într-un arbore AVL	129
5.2.5	Rezumat	137
5.2.6	Test de autoevaluare	137
5.2.7	Răspunsuri la testul de evaluare a cunoștințelor	138
5.3	Unitatea de învățare 3 - Arbori roșu-negru	142
5.3.1	Introducere	142
5.3.2	Obiective	142
5.3.3	Insertia într-un arbore roșu-negru	143
5.3.4	Ștergerea dintr-un arbore roșu-negru	148
5.3.5	Rezumat	155
5.3.6	Test de autoevaluare	155
5.3.7	Răspunsuri la testul de evaluare a cunoștințelor	156

6	Structuri de date avansate	159
6.1	Unitatea de învățare 1 - B-arbori	160
6.1.1	Introducere	160
6.1.2	Obiective	160
6.1.3	Proprietățile B-arborilor	160
6.1.4	Operații	163
6.1.5	Rezumat	174
6.1.6	Test de autoevaluare	174
6.1.7	Răspunsuri la testul de evaluare a cunoștințelor	175
6.2	Unitatea de învățare 2 - Arbori Quad	180
6.2.1	Introducere	180
6.2.2	Obiective	180
6.2.3	Arbori quad pentru regiuni	180
6.2.4	Arbori quad de tip <i>Point Region</i> (PR)	190
6.2.5	Înălțimea unui arbore quad PR	196
6.2.6	Rezumat	198
6.2.7	Test de autoevaluare	198
6.2.8	Răspunsuri la testul de evaluare a cunoștințelor	199
7	ANEXA	202
7.1	Temă de control - Elemente de programare. Vectori. Matrice	202
7.2	Temă de control - Structuri elementare	203
7.3	Temă de control - Tabele de dispersie	204
7.4	Temă de control - Arbori și heap-uri	204
	Bibliografie	206

Introducere

Cursul Structuri de date a fost elaborat ca suport didactic în special pentru studenții anului I din domeniul informaticii și oferă o descriere detaliată a principalelor structuri necesare pentru dezvoltarea aplicațiilor soft.

Fiecare capitol este dedicat unei anumite categorii de structuri și este structurat în unități de învățare. Aceste unități conține definițiile structurilor discutate, operațiile principale, algoritmi în pseudo-cod, considerații despre complexitate, exemple și exerciții propuse spre rezolvare. Problemele propuse la sfârșitul fiecărei unități sunt rezolvate și au ca scop aprofundarea noțiunilor discutate în partea teoretică.

În cadrul orelor de laborator, vor fi implementate o parte dintre structuri, vor fi utilizate în contextul unor probleme și de asemenea va fi introdusă biblioteca standard C++, corespunzătoare celor discutate teoretic la curs. Această bibliotecă permite dezvoltarea de aplicații eficiente și puternice în versiunile moderne de C++ și ușurează considerabil munca programatorilor



Obiectivele cursului Cursul de Structuri de date are ca obiectiv principal formarea unor fundamente în această zonă, înțelegerea unor principalelor structuri de date folosite în aplicațiile de software, precum și modul de utilizare eficientă a acestora, adaptat la probleme specifice. De asemenea, urmărește familiarizarea studenților cu modul de implementare a acestor structuri, eficiența lor teoretică, dar și în contextul rezolvării de probleme concrete.



Competențe conferite

La sfârșitul acestui curs, studenții vor fi capabili:

- să înțeleagă principalele structuri de date: liste, stive, cozi, tabele de dispersie, cozi de prioritate, arbori binari de căutare;
- să înțeleagă modul de funcționare al unor structuri de date mai avansate, precum B-arbori sau arbori quad;
- să implementeze structurile de bază;
- să poată selecta structurile adecvate rezolvării unor probleme concrete;
- să poată estima complexitatea structurilor folosite în aplicații concrete.



Resurse și mijloace de lucru

Rezolvarea problemelor necesită compilator pentru limbajul folosit în elaborarea programelor: C ++.



Structura cursului

Cursul de procesare de imagine este structurat în 12 unități de învățare. Fiecare unitate de învățare cuprinde: obiective, aspecte teoretice privind tematica unității de învățare respective, exemple, teste de autoevaluare precum și probleme propuse spre discuție și rezolvare / implementare.

De asemenea la finalul cursului, în Anexă, sunt propuse 4 teme de control, dedicate activității practice la laborator. Acesta trebuie rezolvate acasă și vor fi prezentate oral la finalul semestrului.

Soluțiile problemelor din testele de autoevaluare propuse la finalul fiecărei unități, se află imediat după testele corespunzătoare.



Cerințe preliminare

Discipline necesare a fi parcurse și eventual promovate înaintea acestei disciplinei sunt algoritmica, noțiuni de matematică de liceu și cursul de fundamentele programării.



Durata medie de studiu individual

Parcursarea de către studenți a unităților de învățare ale cursului de Structuri de date (excluzând rezolvarea testelor de evaluare) se poate face în timpul indicat pentru fiecare unitate.



Modul de evaluare

Studentii vor fi evaluați pe baza unei note de laborator și a unei note la examenul scris. Fiecare notă reprezintă 50% din nota finală.

Nota de laborator e obținută în modul următor

- o testare de laborator, a cărei notă reprezintă 80% din nota finală de laborator.
- evaluare continuă a activităților din timpul orelor de laborator și a temelor de laborator, această notă reprezintă 20% din nota de laborator.

Pentru promovare este necesară obținerea notei minime 5 atât la laborator cât și la examenul scris.

Test inițial

1. Care este diferența dintre o variabilă și o constantă?
2. Cum se declară și cum se utilizează un tablou în C++?
3. Ce este o funcție și care sunt avantajele utilizării funcțiilor?
4. Ce înseamnă transmiterea parametrilor prin valoare și prin referință?
5. Care este diferența dintre bucla `while` și cea `do-while`?
6. Ce este recursivitatea și puteți da un exemplu simplu de funcție recursivă?
7. Ce este un pointer? Cum se poate aloca dinamic memoria cu ajutorul pointerilor în limbajul C++?
8. Ce este un algoritm?
9. Cum ați sorta un șir de numere în ordine crescătoare folosind o metodă simplă?
10. Care este diferența dintre căutarea liniară și căutarea binară?
11. Ce înseamnă complexitatea temporală a unui algoritm?
12. Puteți da exemple de operații de bază asupra unui tablou (inserare, ștergere, căutare)?
13. Ce este o relație de ordine (de exemplu \leq) și cum se folosește în algoritmi?
14. Ce este o structură (`struct`) și în ce situații este utilă?
15. De ce este importantă analiza complexității algoritmilor înainte de implementare?

Capitolul 1

Structuri de date de bază

Introducere

Structurile de date reprezintă colecții de elemente (date) în care există anumite relații structurale. Fiecare element dintr-o astfel de structură are o anumită poziție în cadrul structurii și există un mod specific de acces al acestui element.

În informatică, structurile de date sunt utilizate pentru memorarea și manipularea eficientă cu calculatorul a unor mulțimi dinamice de date. Denumirea de *mulțimi dinamice* provine de la faptul că acestea pot suferi modificări prin operații de inserare și/sau extragere de elemente.

Reprezentarea elementelor unei mulțimi

În anumite situații, de exemplu în cazul listelor înlănțuite sau a arborilor, un element al unei mulțimi dinamice de date este reprezentat printr-o structură / clasă, care dispune de mai multe câmpuri, dintre care unele conțin informații de interes, iar altele pointeri (legături) către alte elemente ale mulțimii de date (următor, precedent în cazul listelor înlănțuite sau părinte, fii în cazul arborilor).

Unele structuri presupun existența unui câmp *cheie*, care identifică în mod unic fiecare element.

În cazul cheilor provenind din mulțimi bine ordonate, de exemplu numere întregi, caractere, șiruri de caractere, pot fi efectuate operații de ordonare / sortare sau de determinare a elementului cu cheia minimă respectiv maximă din mulțimea dinamică păstrată în structura de date respectivă.

Operațiile efectuate asupra unei mulțimi dinamice de date pot fi de două tipuri:

1. **Cereri** - operații care extrag informații din mulțimea de date, cea mai importantă fiind *căutarea* unui / *accesul* la un element

2. Operații de modificare a mulțimii - inserarea / ștergerea unui element.

Există un număr mare de diferite structuri de date, fiecare având proprietăți specifice și fiind adaptate rezolvării unui anumit tip de probleme. Alegerea structurii de date potrivite în cadrul unui program ar trebui să țină cont de rezultatul dorit și de complexitatea de timp și memorie pe care le presupune utilizarea acelei structuri pentru problema dată.

În cadrul lucrărilor de laborator va fi introdusă biblioteca *C++* **Standard Template Library (STL)**, care are implementate majoritatea structurilor care vor fi discutate și facilitează astfel rezolvarea problemelor folosind limbajul *C++*. Desigur, fiecare limbaj de programare are propriile implementări ale structurilor de date, dar ideile principale din spatele acestora și complexitatea operațiilor sunt cele discutate în partea teoretică.

În acest capitol vor fi introduse cele mai simple structuri de bază, precum listele înlănțuite, stivele și cozile.

Competențe

La sfârșitul acestui modul de învățare studenții:

- Utilizează structurile de date de bază pentru implementarea algoritmilor.
- Implementează o listă simplu înlănțuită împreună cu operațiile de bază pe aceasta.
- Implementează o listă dublu înlănțuită împreună cu operațiile de bază pe aceasta.
- Definesc și utilizează în contexte concrete stivele și cozile.
- Explică complexitatea operațiilor pe structurile de date discutate.

1.1 Unitatea 1 - Liste înlănțuite



1.1.1 Introducere

Listele înlănțuite, alături de tablouri (vectori), fac parte din categoria celor mai simple structuri de date și au elementele aranjate în mod secvențial. Spre deosebire de vectori, care au de asemenea elementele aranjate secvențial atât din punct de vedere conceptual, cât și în memorie, în cazul listelor, deși trecerea de la un element la altul se realizează secvențial, elementele acestora nu sunt aranjate în locații de memorie succesive. Accesul unui element din listă se face pe baza adresei acestuia, stocată de către elementul precedent. Astfel de structuri permit inserarea și extragerea în mod eficient a unor elemente, fără a necesita deplasarea celorlalte elemente, așa cum este cazul de exemplu pentru vectori. În unitatea de față vor fi prezentate listele simplu și dublu înlănțuite, modul lor de funcționare, implementarea, precum și complexitatea operațiilor principale.

O listă înlănțuită este o structură de date în care fiecare element este la rândul său o structură cu mai multe câmpuri, dintre care unul conține informația, iar celelalte reprezintă legături de tip pointer către elementele vecine din listă.



1.1.2 Obiective

La sfârșitul acestei unități de învățare studenții vor înțelege:

- Ce este o listă simplu înlănțuită și cum funcționează.
- Ce este o listă dublu înlănțuită și cum funcționează.



Durata medie de studiu individual

Parcursul de către studenți a acestei unități de învățare se face în 2 ore.

1.1.3 Liste simplu înlănțuite

Fiecare element al unei liste simplu înlănțuite este o structură care dispune de un câmp pentru informație și de un câmp de legătură către următorul element din listă.

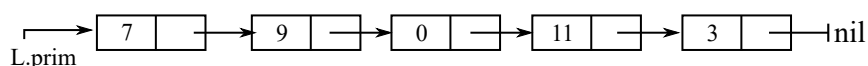
Spre deosebire de un vector / array, elementele unei liste înlănțuite nu sunt neapărat memorate în zone de memorie adiacente și deci nu este necesară alocarea unui bloc de memorie compact pentru elemente sale. În schimb, de fiecare dată când se adaugă un nou element la listă, este necesară alocarea de memorie pentru acest element, iar când se șterge un element din listă, trebuie eliberată memoria corespunzătoare. În $C++$ eliberarea memoriei trebuie făcută

explicit cu comanda **delete**. Alte limbaje de programare dispun de un așa numit *garbage collector*, care se ocupă de eliberarea memoriei, când nu mai este necesară.

Accesul la elementele listei se realizează prin capul listei, reprezentând primul element. Astfel este necesară o variabilă pentru păstrarea adresei primului element al listei.



Exemplu: de listă simplu înlănțuită.



Pe o listă simplu înlănțuită pot fi definite diferite operații. Printre acestea cele mai frecvente sunt: adăugarea / ștergerea unui element la începutul listei, eliminarea elementelor cu o anumită valoare (cheie), căutarea unui element precum și parcurgerea listei. Câțiva dintre algoritmi pentru implementarea operațiilor pe liste simplu înlănțuite sunt prezentați în continuare. Vom considera pentru aceasta o structură de tip listă simplu înlănțuită care are ca atribute un pointer la primul element din listă, denumit *prim* și o valoare întregă *nr_elemente* care indică numărul de elemente prezente în listă. Fiecare element al listei este o structură de tip *NOD*, cu un câmp *cheie* pentru informație și un câmp *urmator*, care indică următorul element din listă. Pentru ultimul element câmpul *urmator* este *nil*.

Adăugarea unui element la începutul unei liste

Algoritm: LISTA_ADAUGA_PRIM

Input: O listă L , o valoare val care se adaugă la începutul listei.

start

 alocă memorie pentru un nod nou

$nou.cheie \leftarrow val$

$nou.urmator \leftarrow L.prim$

$L.prim \leftarrow nou$

$nr_elemente \leftarrow nr_elemente + 1$

stop

Algoritmul *LISTA_ADAUGA_PRIM* permite adăugarea a unei noi valori val la începutul unei liste L . Acest algoritm este ilustrat în figura 1.1.

Complexitate: Algoritmul de adăugare a unui element la începutul listei are complexitate constantă de timp $\Theta(1)$

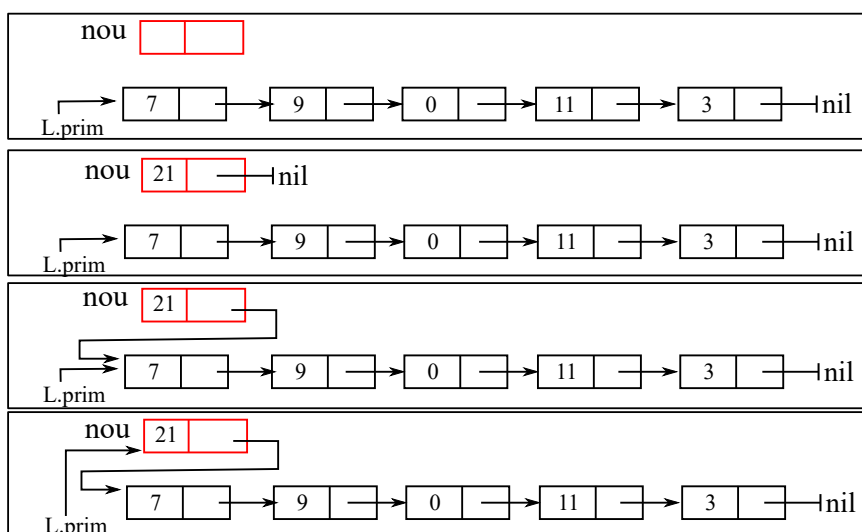


Figura 1.1: Adăugarea valorii 21 în lista simplu înlănțuită din imagine.

Căutarea unui element

Accesul la oricare element din listă se realizează prin parcurgerea acesteia, începând de la capul listei, reprezentat prin elementul *prim*. De asemenea, căutarea unui anumit element se realizează tot prin parcurgerea listei pornind de la primul element, până la identificarea elementului căutat sau până se ajunge la sfârșitul listei, caz în care căutarea se termina prin eșec. Algoritmul de căutare al unei chei (valori) *val* într-o listă simplu înlănțuită *L* este prezentat mai jos.

Algoritm: LISTA_CAUTA

Input: Lista simplu înlănțuită *L*, valoarea căutată *val*

start

curent ← *L.prim*

cat_timp *curent* ≠ *nil* și *curent.cheie* ≠ *val* **executa**

 | *curent* ← *curent.urmator*

sfarsit_cat_timp

 returneaza(*curent*)

stop

Funcția LISTA_CAUTA returnează un pointer către zona de memorie în care este stocat elementul cu valoarea *val*. În cazul în care acest element nu există, algoritmul returnează *nil*.

Complexitate: Căutarea unui element cu o anumită valoare din listă presupune parcurgerea acesteia, până se întâlnește valoarea respectivă. În cel mai defavorabil caz, atunci când elementul nu se află în listă sau se află pe ultima poziție, trebuie parcursă întreaga listă. În medie complexitatea este liniară, $\Theta(n)$, unde n = lungimea listei.

Eliminarea unei valori dintr-o listă

Pentru a șterge o anumită valoare dintr-o listă înlănțuită, mai întâi trebuie găsit nodul ce reține această valoare. Apoi trebuie realizată legătura dintre nodul care precede valoarea căutată și nodul care urmează. Ștergerea dintr-o listă simplu înlănțuită poate fi implementată în diferite moduri. O variantă este aceea de a transmite ca parametru funcției de ștergere o valoare, care se dorește a fi eliminată. În funcție de modul de implementare ar putea fi eliminată prima apariție a valorii în listă sau toate aparițiile (sau ce alte variante se doresc). Algoritmul de mai jos prezintă varianta de eliminare a tuturor aparițiilor unei valori în listă și are complexitate liniară, deoarece presupune parcurgerea întregii liste.

Algoritm: LISTA_ELIMINA

Input: O listă simplu înlănțuită L , valoare val care trebuie eliminată

start

```

cat_timp  $L.prim \neq nil$  si  $L.prim.cheie = val$  executa
  |  $LISTA\_STERGE\_PRIM(L)$ 

```

```

sfarsit_cat_timp

```

```

daca  $L.prim = nil$  atunci
  |  $returneaza()$ 

```

```

sfarsit_daca

```

```

 $curent \leftarrow L.prim$ 

```

```

cat_timp  $curent.urmator \neq nil$  executa

```

```

  | daca  $curent.urmator.cheie = val$  atunci

```

```

    |  $nod \leftarrow curent.urmator$ 

```

```

    |  $curent.urmator \leftarrow curent.urmator.urmator$ 

```

```

    | eliberează memoria alocată pentru  $nod$ 

```

```

    |  $nr\_elemente \leftarrow nr\_elemente - 1$ 

```

```

  | sfarsit_daca

```

```

  | altfel

```

```

    |  $curent \leftarrow curent.urmator$ 

```

```

  | sfarsit_daca

```

```

sfarsit_cat_timp

```

stop

În funcția $LISTA_ELIMINA$, variabila $curent$ reprezintă nodul cu ajutorul căruia se parcurge lista și, în final, nodul dinaintea nodului care va fi șters. Ștergerea din listă presupune de fapt legarea elementului $curent$, de nodul care îi urmează lui $curent.urmator$. În cazul în care valoarea pe care dorim să o ștergem se află chiar la începutul listei, se apelează funcția $LISTA_STERGE_PRIM$, care presupune eliminarea primului element al listei și a cărei imple-

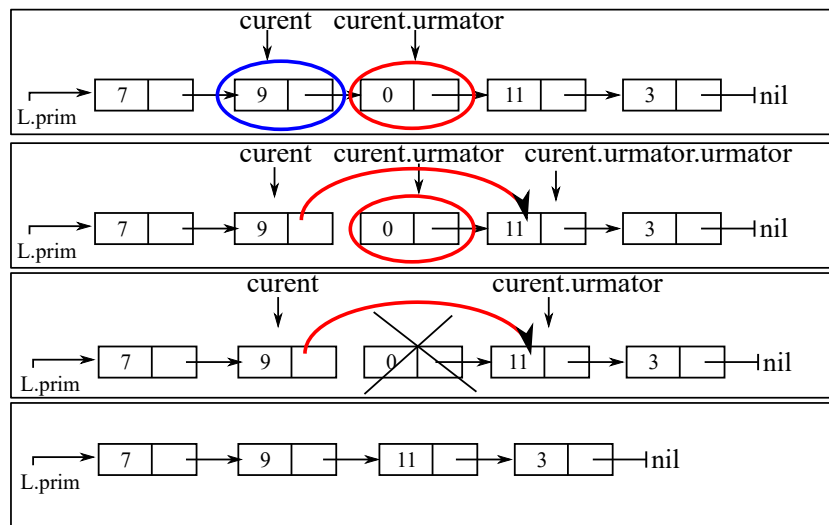


Figura 1.2: Ștergerea valorii 0 dintr-o listă simplu înlănțuită.

mentare rămâne ca exercițiu. Algoritmul este ilustrat în figura 1.2.

Complexitate: Ștergerea unei anumite valori din listă presupune parcurgerea acesteia, până la capăt și atunci când se întâlnește valoarea respectivă, nodul se elimină. Complexitatea operației este liniară, $\Theta(n)$, unde n este lungimea listei.

Desigur, există o serie de alte operații care pot fi efectuate pe o listă simplu înlănțuită, majoritatea presupunând iterarea prin elementele acesteia. Spre exemplu, se pot insera sau șterge elemente pe baza unor constrângeri, se pot efectua sortări, etc.



Implementați o structură de tip LISTA, care are ca membru un pointer la o structură de tip NOD, numit *prim* și un membru de tip *int*, care păstrează numărul de elemente ale listei. În plus, LISTA dispune de funcțiile *LISTA_ADAUGA_PRIM*, care adaugă un element la începutul listei, *LISTA_CAUTA*, care returnează un pointer la nodul ce conține elementul căutat sau *nil*, precum și *LISTA_STERGE_PRIM*, care șterge primul element al unei liste simplu înlănțuite.

De asemenea implementați o funcție care permite parcurgerea listei de la primul element până la ultimul și afișarea valorilor din listă.

1.1.4 Liste dublu înlănțuite

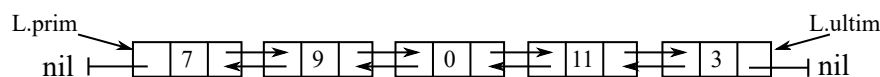
Spre deosebire de elementele unei liste simplu înlănțuite, elementele unei liste dublu înlănțuite posedă, alături de o legătură spre următorul element, și o legătură către elementul precedent.

Accesul la elementele din listă se poate realiza prin capul listei, care indică primul element, dar și prin ultimul element. Vom considera fiecare element din lista dublu înălțuită ca fiind o structură de tip *NOD*, care dispune de câmpurile *cheie* pentru informație, *precedent* pentru legătura la elementul precedent și *urmator* pentru legătura la elementul următor din listă.

Algoritmii prezentați în [4] tratează liste dublu înălțuite, în care accesul se realizează doar la capul listei, printr-o variabilă *prim*, adăugarea în listă făcându-se doar la începutul acesteia. În acest curs însă, vom considera o implementare similară celei din biblioteca STL, în care accesul poate avea loc atât de la primul element, cât și de la ultimul și funcțiile de adăugare și ștergere se vor realiza la ambele capete în complexitate constantă. Structura *LISTA* dispune în acest caz de două câmpuri: *prim* și *ultim*, care reprezintă pointeri către primul, respectiv ultimul element din listă, precum și de un câmp *nr_elemente* care contorizează numărul de elemente prezente în listă.



Exemplu: exemplu de listă dublu înălțuită.



Printre cele mai importante operații asupra unei liste dublu înălțuite sunt: adăugarea / eliminarea unui element, ștergerea unei valori din listă sau iterarea prin listă în ambele direcții. Căutarea unui element presupune parcurgerea listei de la primul element până la găsirea acestuia sau până la sfârșitul listei, în caz de eșec. În continuare vor fi descrise câteva dintre operațiile de bază, împreună cu algoritmi în pseudocod. Alte operații sunt propuse pentru implementare în temele de laborator.

Adăugarea unui element la începutul unei liste

Adăugarea unui noi element la începutul listei presupune alocarea de memorie pentru noul element. Apoi acesta trebuie să fie legat de primul element din listă. Spre deosebire de algoritmul în cazul listei simplu înălțuite, în cazul introducerii într-o listă vidă, trebuie tratat și câmpul *ultim*. Algoritmul în pseudocod este redat în continuare, iar procesul este ilustrat în figura 1.3.

Algoritm: LISTA_ADAUGA_PRIM**Input:** O listă dublu înlănțuită L , a valoare val care se adaugă la început.**start**alocă memorie pentru un nod nou $nou.cheie \leftarrow val$ $nou.urmator \leftarrow L.prim$ $nou.precedent \leftarrow nil$ **daca** $L.prim \neq nil$ **atunci**| $L.prim.precedent \leftarrow nou$ **sfarsit_daca****altfel**| $L.ultim \leftarrow nou$ //ultimul element vor indica spre nou **sfarsit_daca** $L.prim \leftarrow nou$ //primul element vor indica spre nou $nr_elemente \leftarrow nr_elemente + 1$ **stop**

Complexitate: Algoritmul de adăugare a unui element la începutul listei are complexitate constantă de timp și de memorie - $\Theta(1)$.

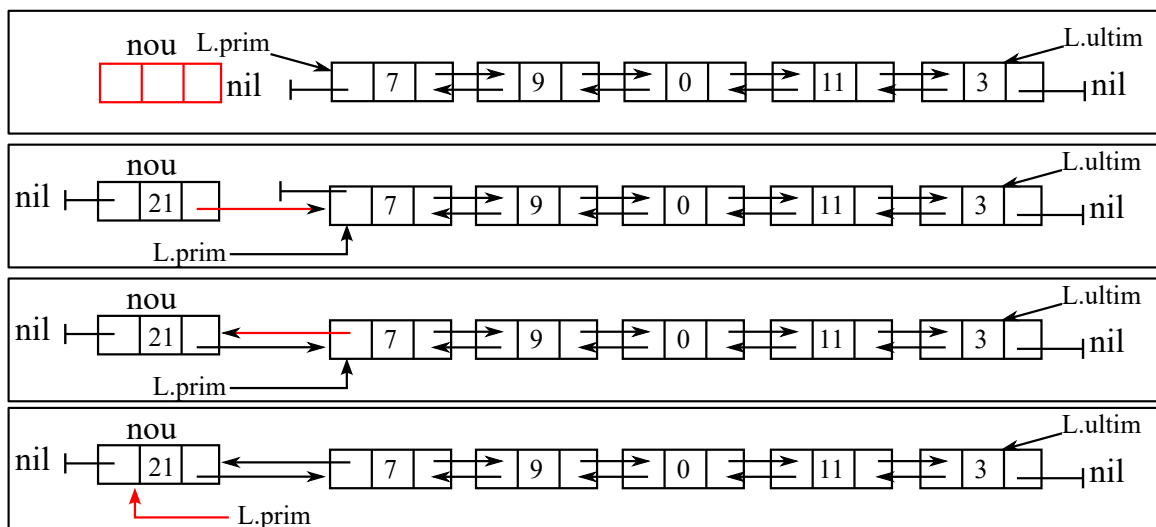


Figura 1.3: Adăugarea valorii 21 într-o listă dublu înlănțuită.

Similar poate fi definit un algoritm pentru inserarea unui nou element la sfârșitul listei folosind accesul la ultimul element indicat de $ultim$.



Implementați o funcție $LISTA_INSEREAZĂ$ care are ca parametru un element de tip nod și o valoare și care inserează valoarea după nodul respectiv. **Variantă:** Se dau ca parametri două valori $val1$ și $val2$ și funcția inserează $val2$ după prima apariție în listă a lui $val1$. Observați diferențele între cele două variante.

Ștergerea dintr-o listă dublu înlănțuită

Ștergerea dintr-o listă dublu înlănțuită poate însemna eliminarea primului sau a ultimului element, eliminarea apariției unei anumite valori (implică o căutare, similar cu ștergerea din lista simplu înlănțuită) sau eliminarea unui nod, care a fost deja identificat, de exemplu după apelul funcției *LISTA_CAUTA*.

Funcția *LISTA_STERGE* prezentată mai jos, realizează ștergerea unui element dintr-o listă înlănțuită reprezentat printr-un pointer la nodul respectiv. Acest lucru se poate face, dacă în prealabil a fost identificat elementul pe baza unei căutări.

Algoritm: LISTA_STERGE

Input: O listă dublu înlănțuită L , pointer la un nod del care trebuie șters.

start

```

daca  $del.precedent \neq nil$  atunci
  |  $del.precedent.urmator \leftarrow del.urmator$ 
sfarsit_daca
altfel
  |  $L.prim \leftarrow del.urmator$ 
sfarsit_daca
daca  $del.urmator \neq nil$  atunci
  |  $del.urmator.precedent \leftarrow del.precedent$ 
sfarsit_daca
altfel
  |  $L.ultim \leftarrow del.precedent$ 
sfarsit_daca
  eliberează memora alocată pentru  $del$ 
   $nr\_elemente \leftarrow nr\_elemente - 1$ 

```

stop

Complexitate: Operația de ștergere efectivă are complexitate constantă. Desigur, identificarea elementului, care trebuie șters, presupune iterarea prin listă și deci complexitate liniară.



Implementați următoarele funcții pentru o listă dublu înlănțuită:

- *LISTA_STERGE_PRIM* care elimină primul element din listă.
- *LISTA_STERGE_ULTIM* care elimină ultimul element din listă.

Funcția *LISTA_ELIMINA*, prezentată în cele ce urmează, are ca parametru o valoare val care se dorește eliminată din lista dublu înlănțuită. Acest lucru presupune parcurgerea listei și la întâlnirea valorii respective apelarea funcției *LISTA_STERGE*, definită mai sus.

Algoritm: LISTA_ELIMINA

Input: O listă dublu înlănțuită L , o valoare val de eliminat.**start** $curent \leftarrow L.prim$ **cat timp** $curent \neq nil$ **executa** $temp \leftarrow curent$ $curent \leftarrow curent.urmator$ **daca** $temp.cheie = val$ **atunci** | $LISTA_STERGE(temp)$ **sfarsit_daca** **sfarsit_cat_timp****stop**

Complexitate: Ștergerea efectivă este $\Theta(1)$, iar parcurgerea listei este $\Theta(n)$, n - numărul de elemente din listă. Astfel complexitatea funcției este $\Theta(n)$.

Căutarea unui element

Algoritmul pentru căutarea unui element într-o listă dublu înlănțuită este identic cu cel de căutare într-o listă simplu înlănțuită, are în mod evident complexitate $\Theta(n)$, unde n reprezintă numărul de elemente ale listei și nu va fi redat încă o dată.

Observație: Modul de implementare al listelor înlănțuite prezentat în acest capitol nu este singurul mod posibil. Denumirile funcțiilor și modul de abordare au fost gândite să se potrivească cu modul de implementare din biblioteca STL, în care, de exemplu, într-o listă simplu înlănțuită avem acces doar la primul element din listă, nu și la ultimul, așa cum avem în cazul celor dublu înlănțuite. Evident că din acest motiv, ștergerea sau adăugarea la sfârșitul unei liste simplu înlănțuite se realizează în complexitate liniară, nu constantă. Dacă dintr-un motiv sau altul se dorește implementarea unei liste simplu înlănțuite, la care aceste operații să se realizeze în complexitate constantă, se poate adăuga un membru suplimentar clasei, care să permită accesul la ultimul element, iar algoritmii se scriu în consecință.

1.1.5 Utilizarea listelor înlănțuite

Dacă ar fi să comparăm listele înlănțuite cu structuri de tip vector, se poate observa ușor că un dezavantaj semnificativ al listelor este acela, de a nu putea accesa elementele prin poziție. Accesul rapid prin poziție este specific vectorilor și este posibil, datorită memorării elementelor unui vector în locații de memorie succesive. În schimb, inserarea sau ștergerea unui element la începutul structurii, sau de pe o poziție arbitrară poate fi mai eficientă în cazul listelor înlănțuite. Deși și aici există controverse, depinzând de natura și de dimensiunea datelor.

Un alt dezavantaj al unei liste înlănțuite este acela că, elementele sale nu sunt stocate în zone de memorie succesive, ci pot fi oriunde în memoria atribuită programului. Astfel nu se cunoaște locația unui element, decât atunci când se accesează, prin traversarea listei. Acest lucru poate constitui un dezavantaj semnificativ de exemplu în paralelizarea execuției de instrucțiuni care accesează date diferite (*out of order execution*) sau transferul de pe CPU pe GPU. În calculatoarele moderne, o serie de operații se execută direct pe CPU, pentru a crește viteza de execuție. În acest caz folosirea listelor înlănțuite poate constitui un dezavantaj semnificativ, mai ales dacă este vorba de un număr redus de elemente.

Dacă aplicația presupune o mulțime de elemente, care se modifică relativ rar sau preponderent prin adăugarea de elemente la sfârșit, dar în care accesul prin poziție este frecvent, evident o structură de tip vector este preferabilă. În schimb, acolo unde modificările prin adăugare / ștergere sunt frecvente, iar accesul prin poziție este de mică importanță și numărul de elemente memorat este mare, se preferă liste înlănțuite [20].

De asemenea, acolo unde sunt frecvente adăugări și extrageri la începutul structurii, vectorii au o performanță slabă. Este de preferat o listă înlănțuită sau un *deque* (urmează să fie prezentat). O comparație interesantă a performanței celor două structuri, vector și listă, este prezentată în [19].

Avantajul unei liste simplu înlănțuite față de o listă dublu înlănțuită este acela că necesită semnificativ mai puțină memorie pentru stocarea legăturilor către elemente vecine. În schimb, spre deosebire de listele dublu înlănțuite, o listă simplu înlănțuită permite iterarea doar într-o singură direcție, de la capul listei spre sfârșit.

Listele înlănțuite pot fi utile de exemplu în implementarea tabelelor de dispersie - *hash tables*, care vor fi discutate în capitolul următor. De asemenea sunt utile, atunci când elementele cu care se lucrează sunt relativ mari, de exemplu pagini de text. O aplicație concretă este acela al navigării înainte și înapoi între imagini (Photoshop) sau al navigării înainte și înapoi între cântecele dintr-un *playlist*.



Să ne reamintim...

- Într-o listă înlănțuită fiecare element este o structură cu mai multe câmpuri, dintre care unul conține informația (cheia), unul conține adresa elementului următor, iar în cazul listelor dublu înlănțuite un câmp conține adresa elementului precedent.
- Accesul la un element într-o listă se face prin parcurgerea listei, începând de la primul element și mergând pe legături către elementele următoare.

- Listele înlănțuite permit adăugarea și ștergerea elementelor la începutul listei în complexitate constantă. Dacă au ca atribut un pointer către ultimul element, adăugarea și ștergerea unui element de la finalul listei se realizează tot în complexitate constantă.
- În cazul listelor înlănțuite, elementele nu sunt stocate în zone de memorie alăturate, ceea ce poate constitui un dezavantaj în cazul calculatoarelor moderne cu o structură ierarhizată a memoriei.

1.1.6 Rezumat



În această secțiune au fost discutate listele simplu și dublu înlănțuite, precum și principalele operații pe liste. Au fost prezentați câțiva algoritmi în pseudocod pentru implementarea unora dintre principalele operații pe liste, împreună cu complexitatea acestora.



1.1.7 Test de autoevaluare

1. Dacă într-o listă dublu înlănțuită L se apelează următoarele funcții în ordinea de mai jos. Care este forma finală a listei?

```

LISTA_ADAUGA_PRIM(L, 2);
LISTA_ADAUGA_PRIM(L, 1);
LISTA_ADAUGA_PRIM(L, 9);
LISTA_ADAUGA_PRIM(L, 2);
LISTA_ELIMINA(L, 2);
LISTA_STERGE_ULTIM(L);
LISTA_ADAUGA_ULTIM(L, 5);

```

2. Considerând o listă simplu înlănțuită circulară L cu 4 noduri: $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$. Ce efect va avea apelul funcției următoare asupra acestei liste? Dar asupra întregului program?
3. Se consideră două liste dublu înlănțuite $L1$ și $L2$. Scrieți o funcție, în pseudocod, care realizează concatenarea elementelor listei $L2$ la sfârșitul listei $L1$. Iar, lista $L2$ va deveni vidă. Care este complexitatea de timp a funcției implementate?
4. Scrieți o funcție $INVERSEAZA_LISTA$ în pseudocod, care inversează ordinea elementelor într-o listă simplu înlănțuită și returnează noul cap al listei. Nu se vor folosi structuri auxiliare.

Algoritm: LISTA_AFISARE**Input:** O listă simplu înlănțuită circulară L **start** $curent \leftarrow L.prim$ **cat_timp** $curent \neq nil$ **executa**| Afîșează ($curent.cheie$)| $curent \leftarrow curent.urmator$ **sfarsit_cat_timp****stop**

5. Care este complexitatea de timp a operației de ștergere a ultimului nod dintr-o listă simplu înlănțuită? Dar dintr-una dublu înlănțuită? Argumentați.

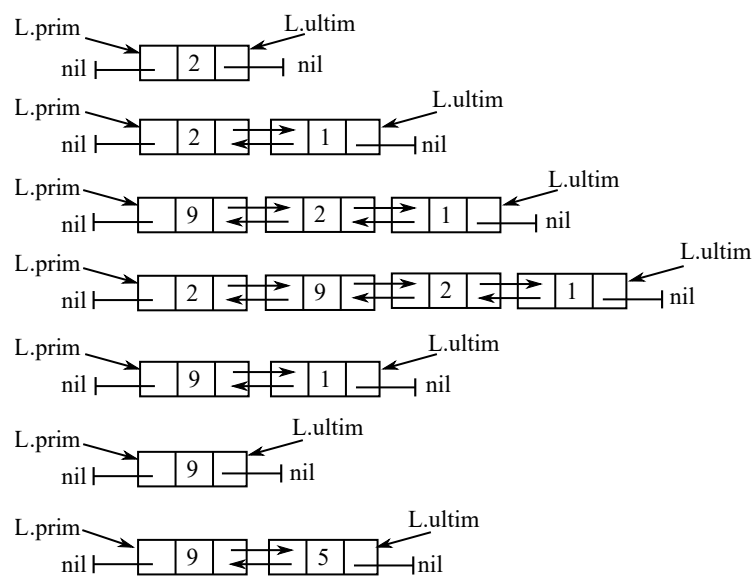
1.1.8 Răspunsuri la testul de evaluare a cunoștințelor

1. Dacă într-o listă dublu înlănțuită L se apelează următoarele funcții în ordinea de mai jos. Care este forma finală a listei?

```

LISTA_ADAUGA_PRIM(L, 2);
LISTA_ADAUGA_PRIM(L, 1);
LISTA_ADAUGA_PRIM(L, 9);
LISTA_ADAUGA_PRIM(L, 2);
LISTA_ELIMINA(L, 2);
LISTA_STERGE_ULTIM(L);
LISTA_ADAUGA_ULTIM(L, 5);

```



Lista va reține două noduri ce conțin valorile 9 și 5.

2. Considerând o listă simplu înlănțuită circulară L cu 4 noduri: $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$. Ce efect va avea apelul funcției următoare asupra acestei liste? Dar asupra întregului program?

Rezolvare: Funcția de afișare a listei circulare definește o variabilă *curent* care este inițializată cu elementul din capătul listei (nodul cu valoarea A), iar apoi aceasta își va schimba valoarea, reținând pe rând fiecare pointer către fiecare nod al listei. Întrucât asupra acestei variabile, ce reține adresa de memorie a elementelor din listă, nu se produc modificări asupra structurii, componenta listei rămâne aceeași.

Pentru că lista este una circulară, fiecare nod aflat în componentă va avea ca element următor un alt nod, ceea ce conduce la o execuție la infinit a instrucțiunii **cat_timp**, variabila *curent* neajungând niciodată la valoarea *nil*, deci programul va intra într-un ciclu infinit. Pentru ca să se oprească, în condiția **cat_timp** ar trebui verificat dacă nodul *curent* devine egal cu nodul *prim*.

3. Se consideră două liste dublu înlănțuite $L1$ și $L2$. Scrieți o funcție, în pseudocod, care realizează concatenarea elementelor listei $L2$ la sfârșitul listei $L1$. Iar, lista $L2$ va deveni vidă. Care este complexitatea de timp a funcției implementate?

Algorithm: CONCATENARE_LISTE

Input: Două liste dublu înlănțuite $L1$ și $L2$

start

daca $L1.prim = nil$ **atunci**

$L1.prim \leftarrow L2.prim$

$L1.ultim \leftarrow L2.ultim$

sfarsit_daca

altfel

daca $L2.prim \neq nil$ **atunci**

$L1.ultim.urmator \leftarrow L2.prim$

$L2.prim.precedent \leftarrow L1.ultim$

$L1.ultim \leftarrow L2.ultim$

sfarsit_daca

sfarsit_daca

$L2.prim \leftarrow nil$

$L2.ultim \leftarrow nil$

stop

Complexitate: $O(1)$

4. Scrieți o funcție *INVERSEAZA_LISTA* în pseod-cod, care inversează ordinea elementelor într-o listă simplu înlănțuită și returnează noul cap al listei. Nu se vor folosi structuri auxiliare.

Algorithm: INVERSEAZA_LISTA

Input: O listă dublu înlănțuită L

start

$prec \leftarrow nil$

$curent \leftarrow prim$

cat timp $curent \neq nil$ **executa**

$urmator \leftarrow curent.urmator$ // reține legătura înainte să o schimbăm

$curent.urmator \leftarrow prec$ // inversăm legătura

 // avansăm în lista

$prec \leftarrow curent$

$curent \leftarrow urmator$

sfarsit_cat_timp

 returneaza($prec$)

stop

Complexitate: $O(n)$

5. Care este complexitatea de timp a operației de ștergere a ultimului nod dintr-o listă simplu înlănțuită? Dar dintr-una dublu înlănțuită? Argumentați.

Rezolvare: Operația de ștergere a unui nod într-o listă simplu înlănțuită se realizează în complexitate $O(1)$, fiind necesară doar refacerea legăturii penultimului nod cu *nil* ca nod următor și ștergerea ultimului nod. Totuși, pentru a obține penultimul nod al listei este necesar să parcurgem întreaga listă, ceea ce presupune un număr de n pași. Astfel, complexitatea întregului algoritm se obține însumând timpii celor două operații și devine $O(n)$.

Operația de ștergere a unui nod dintr-o listă dublu înlănțuită se poate realiza în complexitate $O(1)$, deoarece lista reține o referință către ultimul nod, iar ultimul nod dispune de un pointer către nodul precedent (penultimul nod). Astfel, pentru a elimina acest ultim nod, este suficient să setăm câmpul *urmator* al penultimul nod la *nil* și să eliberăm memoria alocată pentru ultimul nod.

1.2 Unitatea 2 - Stive și cozi



1.2.1 Introducere

Stivele și cozile pot fi considerate niște *adaptoare de containere*, adică niște structuri, care au la bază alte structuri elementare (containere) precum un vector sau o listă înlănțuită, a căror funcționare adaptează containerul conform cerințelor. Astfel, stivele și cozile restricționează pozițiile în care pot fi adăugate sau extrase elemente, determinând astfel ordinea de prelucrare a elementelor în funcție de ordinea de adăugare în structura respectivă. În această unitate vom discuta modul de funcționare al stivelor și cozilor, operațiile principale și cum pot fi implementate folosind drept containere vectorii sau listele.



1.2.2 Obiective

La sfârșitul acestei unități de învățare studenții vor înțelege:

- Ce este o stivă și cum funcționează.
- Ce este o coadă și cum funcționează.
- Cum pot fi implementate aceste structuri.
- Ce este un deque.



Durata medie de studiu individual

Parcursul de către studenți a acestei unități de învățare se face în 2 ore.

1.2.3 Stiva - *stack*

Stiva - *stack* - este o structură cu o disciplină de intrare/ieșire de tip **LIFO** - *Last In First Out*, adică ultimul element introdus va fi primul care se extrage pentru prelucrare. Putem compara stiva cu o mulțime de obiecte puse unul deasupra celuilalt. Primul obiect introdus se află la bază, iar ultimul se află în vârf, deci primul obiect care poate fi luat din stivă este cel din vârf, iar adăugarea unui element nou se va realiza tot prin vârf. Accesul la un element aflat mai jos de vârful stivei nu se poate realiza, decât după ce au fost extrase toate elementele aflate deasupra acestuia.

Stiva poate fi considerată un **adaptor de container**, adică în spate există o altă structură (container), în care se stochează efectiv elementele, iar modul de acces la elemente este cel descris mai sus. Pentru stocarea elementelor din stivă se poate utiliza de exemplu un tablou liniar / vector (reprezentare secvențială), o listă înlănțuită sau o structură de tip *deque*.

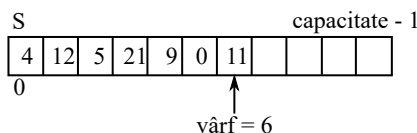
Un obiect de tip stivă trebuie să dispună de funcții pentru adăugarea, respectiv pentru eliminarea unui element în vârful stivei, pentru acces la elementul din vârful stivei și pentru verificarea dacă stiva este vidă. Vom considera cele două moduri de implementare ale stivei, secvențială și înlănțuită și vom prezenta pseudocodul pentru funcțiile *ADAUGA - push* și *EXTRAGE - pop*.

Reprezentarea secvențială

Se va considera o structură *STIVA*, care dispune de un atribut de tip vector / array pentru stocarea elementelor, precum și de un atribut *varf* care reprezintă poziția vârfului în vector. Dacă pentru stocarea elementelor se utilizează containerul *vector* din STL, atunci nu trebuie luată în considerare umplerea stivei, deoarece acest container realocă memorie atunci când este necesar. Dacă în schimb structura folosește un *array*, atunci memoria disponibilă este limitată și trebuie ținut cont de posibilitatea umplerii stivei. În acest caz vom considera un atribut suplimentar *capacitate*, reprezentând numărul maxim de elemente care poate fi stocat în stivă. Se consideră indicii în vector încep de la 0. Astfel dacă $varf = -1$ atunci stiva este vidă și nu pot extrage elemente, iar dacă $varf = capacitate - 1$ stiva este plină și nu pot adăuga elemente.



Exemplu: de stivă care are la bază reprezentarea secvențială. Stiva conține 7 elemente, iar variabila $varf = 6$.



Adăugarea unui element în stivă se realizează prin intermediul unei funcții numită în general *ADAUGA - push*. Aceasta crește indicele *varf* cu o unitate și plasează în vector pe această poziție noul element.

Algoritm: ADAUGA

Input: O stivă *S*, valoarea *val* de adăugat.

start

daca $S.varf = capacitate - 1$ **atunci**

| **returneaza**(eroare de stiva plina) //sau realocare de memorie

sfarsit_daca

$S.varf \leftarrow S.varf + 1$

$S[S.varf] \leftarrow val$

stop

Extragerea din stivă se realizează la vârf, prin intermediul unei funcții *EXTRAGE* - *pop*, care scade indicele *varf*. Dacă stiva este vidă, nu pot fi extrase elemente.

Algoritm: EXTRAGE

Input: O stivă *S*.

start

daca $S.varf = -1$ **atunci**
 | **returneaza**(eroare de stiva vida)

sfarsit_daca

$S.varf \leftarrow S.varf - 1$

stop

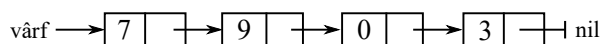
Complexitate: ambele operații sunt practic instantanee - complexitatea este $O(1)$. Doar dacă în cazul umplerii containerului pentru stivă, se realocă o zonă de memorie mai mare și se copiază elementele din zona veche, în zona nouă, această operație este liniară relativ la numărul elementelor din stivă. Operația de realocare trebuie să aibă loc rar, astfel încât complexitatea să rămână în medie constantă.

Reprezentare înlănțuită

În cazul reprezentării înlănțuite se poate utiliza drept container o listă simplu înlănțuită, în care fiecare element este o structură de tip *nod* cu 2 câmpuri: *cheie* - pentru informație și *urmator* - pointer către elementul următor. Desigur, ar putea fi folosită și o listă dublu înlănțuită, dar deoarece pe stivă nu au loc parcurgeri și este suficient accesul la primul element, nu se justifică consumul de memorie presupus de o listă dublu înlănțuită. Putem considera pentru stivă o structură / clasă care să aibă ca membru un element *varf* de tip pointer la *nod* ce reprezintă vârful stivei.



Exemplu: reprezentare a unei stive printr-o listă simplu înlănțuită.



Pentru a adăuga un element nouă într-o stivă ce are la bază o listă înlănțuită, trebuie mai întâi alocată memorie pentru noul element. Apoi adăugarea se face în aceeași manieră descrisă în cazul listelor.

La extragerea unui element din listă, trebuie să eliberăm memoria rezervată pentru acest element.

Funcțiile de adăugare și eliminare a elementului din vârful stivei *S* sunt descrise în continuare.

Algoritm: ADAUGA**Input:** O stivă S , o valoare val de adăugat.**start**| alocă memorie pentru un nod y | $y.cheie \leftarrow val$ | $y.urmator \leftarrow S.varf$ | $S.varf \leftarrow y$ **stop****Algoritm: EXTRAGE****Input:** O stivă S **start**| **daca** $S \neq \emptyset$ **atunci**| | $y \leftarrow S.varf$ | | $S.varf \leftarrow S.varf.urmator$ | | eliberează memoria pentru y | **sfarsit_daca**| **altfel**| | `returneaza(eroare de stiva vida)`| **sfarsit_daca****stop****Complexitate:** ambele operații sunt practic instantanee - complexitatea este $O(1)$.

1.2.4 Coadă - *queue*

Coadă este o structură cu o disciplină de intrare/ieșire de tip **FIFO** - *First In First Out*, adică primul element introdus va fi primul care se extrage pentru prelucrare. Coadă modelează procese care presupun formarea de cozi, de exemplu deservirea clienților la un ghișeu. De asemenea se utilizează cozi pentru parcurgerea în lățime de grafuri, respectiv arbori.

Comparare cu stiva. Spre deosebire de stivă, unde se utilizează o variabilă pentru accesarea vârfului stivei și atât adăugarea cât și extragerea elementelor se realizează prin intermediul acesteia, în cazul unei cozi este necesară memorarea a două elemente: primul - aici se face extragerea și ultimul - aici se face adăugarea.

Similar cu stiva, coada este un adaptor de container, deci are la bază un container în care se păstrează elementele, de exemplu un *array* (vector), o listă înlănțuită sau o structură de tip *deque*.

Un obiect de tip coadă trebuie să dispună de funcții pentru adăugarea unui element la sfârșitul cozii, pentru extragerea primului element al cozii, pentru obținerea primului și ultimului element, precum și pentru verificarea dacă este vidă.

Reprezentarea secvențială

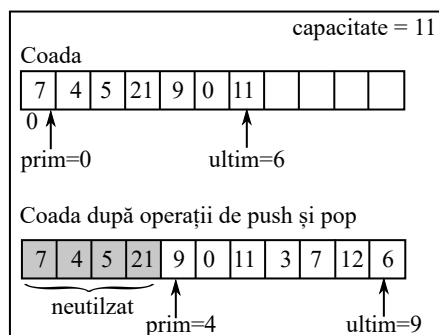


Figura 1.4: Din coada C s-au extras elementele 7, 4, 5 și 21 și s-au adăugat elementele 3, 7, 12, 6. În acest moment primele 4 poziții sunt considerate neocupate, dar funcția *ADAUGA* returnează mesajul de coadă plină.

Se va considera o structură/clasă *COADA* care dispune de un atribut de tip vector / array pentru stocarea elementelor și de două atribute *prim* și *ultim* care indică primul și ultimul element din coadă. Dacă dimensiunea containerului este fixă, atunci coada se poate umple și trebuie considerat și un atribut *capacitate*, care reprezintă numărul maxim de elemente, ce poate fi stocat în coadă.

Observație: adăugarea unui element în coadă se realizează la sfârșitul cozii, ceea ce duce la incrementarea indicelui *ultim*, iar extragerea elementului de la începutul cozii se realizează prin creșterea indicelui *prim*. Astfel, elementul nu se șterge efectiv din memorie, doar ca nu se mai consideră ca aparținând cozii. În figura 1.4 este ilustrat modul în care funcționează o astfel de coadă. Problema care apare este migrarea cozii spre dreapta, odată cu efectuarea de operații succesive de adăugare și ștergere de elemente.

Coadă este considerată plină dacă indicele *ultim* ajunge să fie egal cu ultima poziție din vector și este considerată vidă, dacă indicele *prim* devine mai mare decât *ultim* (prin avansare odată cu apelul funcției *ADAUGA*). Astfel se poate ajunge la situația în care în vectorul de date corespunzător cozii sunt multe poziții considerate neocupate, dar totuși la apelarea funcției *ADAUGA* se primește mesaj de coadă plină - fig. 1.4, deoarece pozițiile considerate libere, aflate înaintea poziției *prim* nu mai sunt accesibile. Această problemă se rezolvă în practică prin utilizarea unei cozi circulare.

Coadă circulară

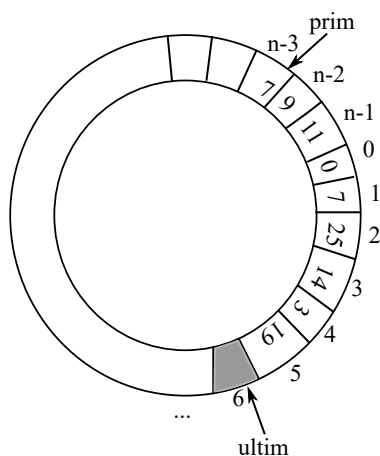


Figura 1.5: Coadă circulară cu n poziții în care la momentul curent variabila $prim$ indică poziția $n - 3$ și variabila $ultim$ indică poziția neocupată 6.

În cazul unei cozi circulare, atunci când variabila $ultim = capacitate - 1$ la următoarea inserție $ultim$ va devine 0, deci se reia circular parcurgerea cozii de la început. Similar se procedează cu variabila $prim$ la extragere. Astfel se va primi mesaj de eroare de coadă plină doar atunci când sunt ocupate toate pozițiile alocate în vector pentru elementele cozii.

Condițiile de coadă vidă/plină

În cazul cozilor circulare condițiile de coada plină și coadă vidă descrise mai sus nu mai sunt valabile:

- Inegalitatea $prim > ultim$ nu înseamnă decât faptul că s-a reluat parcurgerea cozii de la început - vezi fig.1.5.
- Evident condiția $ultim = capacitate - 1$ nu mai generează mesajul de coadă plină

Soluționarea problemei detecției cozii vide sau pline: Această problemă se rezolvă în modul următor. Poziția din vector indicată de elementul $ultim$ nu va fi ocupată, deci în vector va exista mereu o poziție neocupată. Ea este utilizată doar pentru marcarea sfârșitului cozii. Condițiile de coadă vidă / coadă plină sunt în acest caz:

- Dacă $prim = ultim$ atunci coada este vidă
- Dacă $(ultim + 1) \bmod capacitate = prim$ atunci coada este plină

Funcțiile de adăugare și respectiv eliminare a unui element din coadă pot fi descrise prin:

Algoritm: ADAUGA

Input: O coadă C , o valoare val de adăugat.

start

daca $(C.ultim + 1) \bmod C.capacitate = C.prim$ **atunci**

| **returneaza**(eroare de coada plina)

| //Sau realocare de memorie

sfarsit_daca

$C[C.ultim] \leftarrow val$

$C.ultim \leftarrow (C.ultim + 1) \bmod C.capacitate$

stop

Algoritm: EXTRAGE

Input: O coadă C

start

daca $C.prim = C.ultim$ **atunci**

| **returneaza**(eroare de coada vida)

sfarsit_daca

$C.prim \leftarrow C.prim + 1$

daca $C.prim = C.capacitate$ **atunci**

| $C.prim \leftarrow 0$

sfarsit_daca

stop

Complexitatea operațiilor de adăugare și extragere din coadă este constantă, cu excepția realocării în funcția *ADAUGA*, dacă are loc.

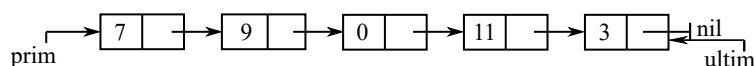
Observație: În practică un *array* este potrivit pentru coadă, doar dacă numărul de elemente pe care dorim să le stocăm, nu este excesiv de mare.

Reprezentare înlănțuită

Pentru fiecare element poate fi utilizată o structură care conține 2 câmpuri: *cheie* - pentru informație și *urmator* - pointer către elementul următor.



Exemplu de coadă reprezentată printr-o listă simplu înlănțuită.



Alocarea memoriei pentru elementele unei astfel de cozi se face dinamic, nu static, așa ca

la vector, adică la introducerea un element nou, trebuie mai întâi alocată memorie, iar la extragerea unui element, trebuie eliberată zona de memorie care era alocată acelui element.

Sunt necesare două variabile *prim* și *ultim*, în care se rețin adresele primului respectiv ultimului element din coadă. Elemente noi se adaugă mereu după ultimul element iar la eliminare se consideră primul element din coadă. În continuare sunt prezentați cei doi algoritmi.

Algoritm: ADAUGA

Input: O coadă C , o valoare val de adăugat.

start

 alocă memorie pentru un nod nou

$nou.cheie \leftarrow val$

$nou.urmator \leftarrow nil$

daca $C.prim = nil$ **atunci**

 | $C.prim \leftarrow nou$

sfarsit_daca

altfel

 | $C.ultim.urmator \leftarrow nou$

sfarsit_daca

$C.ultim \leftarrow nou$

stop

Algoritm: EXTRAGE

Input: O coadă C

start

daca $C.prim = nil$ **atunci**

 | **returneaza**(eroare de coada vida)

sfarsit_daca

$temp \leftarrow C.prim$

$C.prim \leftarrow C.prim.urmator$

 eliberează memoria pentru $temp$

stop

Complexitatea acestor operații este constantă.

1.2.5 deque

O structură interesantă este *deque* sau *double ended queue* [10] și reprezintă o generalizare a ideii de coadă, în sensul că permite atât inserție cât și ștergere la ambele capete în complexitate

constantă. În mod evident o astfel de structură poate fi implementată printr-o listă dublu înălțuită. Acest lucru însă prezintă dezavantajele discutate în unitatea dedicată listelor și se recomandă a fi evitat în contextul calculatoarelor moderne. Implementări mai bune pot fi realizate folosind un *array*, parcurs circular, așa cum a fost descris la secțiunea pentru cozi sau o implementare care are la bază o secvență de vectori, așa cum este realizată în cazul celei din STL. În cel de-al doilea caz, structura este alcătuită din blocuri de dimensiune fixă N , de preferință putere a lui 2 (variante 16 sau 32). La adăugarea unui nou element, blocul de început se completează începând de la ultimul element către primul, iar ultimul bloc începând de la primul către ultimul. Atunci când primul sau ultimul bloc se umple, se alocă un nou bloc de memorie. Adresele acestor blocuri sunt memorate la rândul lor într-un vector. Structura este reprezentată grafic în figura 1.6.

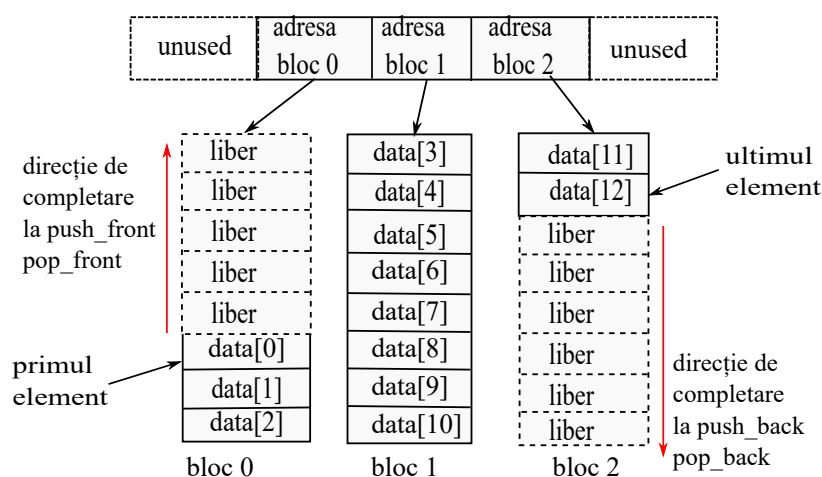


Figura 1.6: Reprezentare grafică a unei implementări pentru *deque* cu un vector de blocuri de dimensiune $N = 8$

Avantajele unei structuri de tip deque implementate astfel sunt:

- accesul rapid prin poziție la elemente (nu ar fi posibil cu o listă înălțuită)
- adăugare / extragere de elemente la ambele capete în complexitate constantă
- la umplerea memoriei alocate nu este necesară realocarea întregului bloc de memorie (ca la *array*), ci doar a unui bloc nou.
- inserarea unui element nou la mijloc se face prin deplasarea a cel mult $n/2$ elemente, unde n este numărul de elemente stocate în structură, spre deosebire de *array*, unde se poate ajunge la deplasarea tuturor elementelor, dacă se dorește inserția la începutul structurii

Astfel un deque poate combina avantajele atât de la *array* cât și de la liste. Este o structură foarte potrivită și pentru implementarea unei cozi.

În [11] este realizată o analiză a performanței acestui container comparativ cu containerul vector.

1.2.6 Aplicații ce utilizează stive și cozi

Stivele se utilizează în diferite probleme, care presupun obținerea unor date în ordine inversă citirii lor. De asemenea recursivitatea are în spate gestionarea memoriei sub formă de stivă. Algoritmii de tip *backtracking* construiesc soluția în mod analog unei stive. Parcurgerile grafurilor și arborilor în adâncime se realizează cu ajutorul stivelor.

Cozile sunt utile în modelarea problemelor ce presupun tratarea datelor de intrare în ordinea citirii lor. De asemenea se utilizează în probleme de parcurgere în lățime a grafurilor și arborilor.



Să ne reamintim...

- Într-o stivă accesul se realizează doar la vârful stivei. Într-o stivă ultimul element introdus va fi primul extras pentru prelucrare.
- Într-o coadă accesul se realizează la ambele capete. Elementele se introduc în coadă la unul dintre capete și se extrag din celălalt.
- Operațiile principale pe stive și cozi se realizează în complexitate constantă.
- Pentru implementare, stivele și cozile au la bază alte containere precum vector, listă sau deque.

1.2.7 Rezumat



În această secțiune au fost discutate stivele, cozile, precum și principalele operații pe acestea. Au fost prezentați câțiva algoritmi în pseudo-cod pentru implementarea operațiilor principale pentru stive și cozi împreună cu complexitatea acestora. Prezenta secțiune a fost urmată de prezentarea structurii deque.



1.2.8 Test de autoevaluare

1. Se consideră o stivă S (inițial vidă) memorată cu ajutorul unui vector. În secvența de mai jos o literă înseamnă o operație de *push* în S a caracterului respectiv, iar un asterisc reprezintă o operație de *pop*. Înainte de fiecare *pop* se afișează pe ecran elementul care urmează să fie extras din stivă. Ce se va afișa pe ecran?

$$ACE * A * STA * * * EST * * E$$

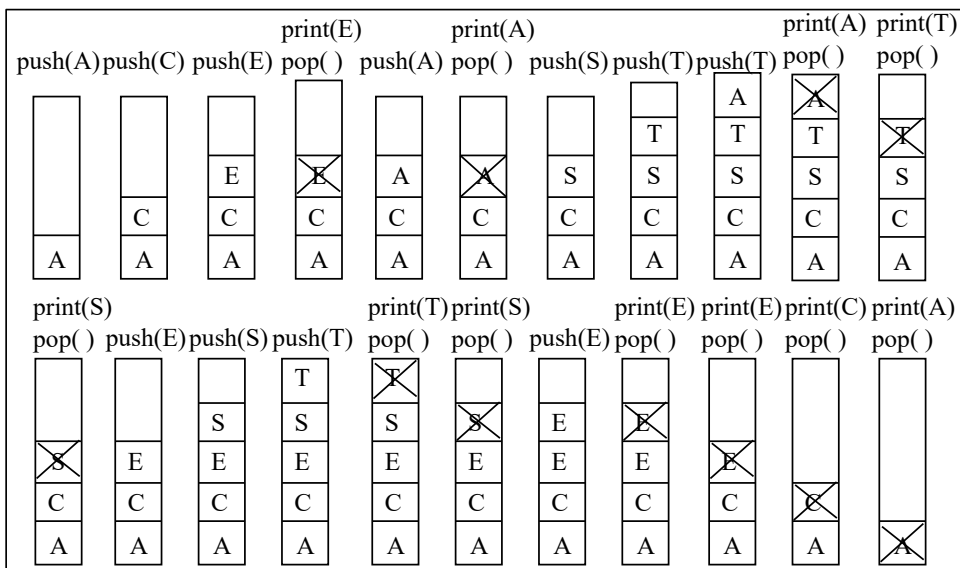
2. Considerăm o stivă S_1 în care se introduc pe rând caracterele A, D, C, B și două stive suplimentare S_2 și S_3 inițial vide. Să se scrie operațiile necesare pentru a trece caracterele din S_1 în S_2 , astfel încât la extragerea pe rând din S_2 și afișarea pe ecran, literele să apară în ordine alfabetică.
3. Se consideră o coadă C . Presupunem că asupra cozii au loc o serie de operații de *push* și *pop*. În cazul operațiilor de *push* sunt adăugate în coadă numere de la 0 la 9 în această ordine. Înainte de fiecare operație de *pop* se afișează elementul care urmează să fie extras. Este posibil ca să fie afișată secvența de valori: 4, 3, 2, 1, 0, 5, 6, 7, 8, 9? (Obs: Nu se efectuează *pop* pe coadă vidă).
4. Scrieți un algoritm în pseudocod care realizează căutarea unui element x într-o coadă folosind o altă coadă suplimentară, folosind doar operațiile specifice cozilor.
5. Scrieți un algoritm în pseudocod care implementează o coadă folosind două stive și operații pe stivă.

1.2.9 Răspunsuri la testul de evaluare a cunoștințelor

1. Se consideră o stivă S (inițial vidă) memorată cu ajutorul unui vector. În secvența de mai jos o literă înseamnă o operație de *push* în S a caracterului respectiv, iar un asterisc reprezintă o operație de *pop*. Înainte de fiecare *pop* se afișează pe ecran elementul care urmează să fie extras din stivă. Ce se va afișa pe ecran?

$$ACE * A * STA * * * EST * * E$$

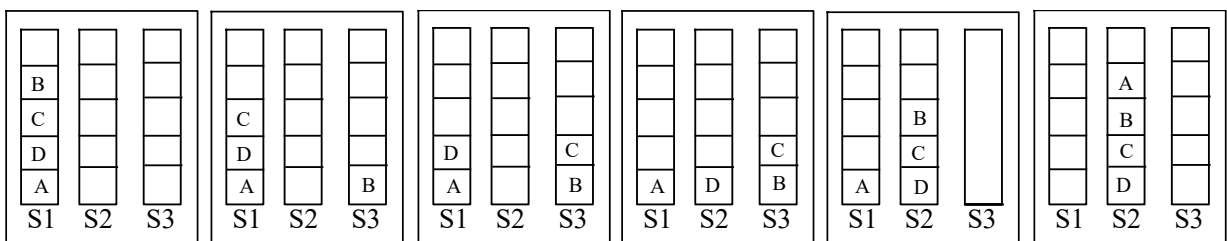
Rezolvare: Ordinea operațiilor pe stiva S indicată de șirul menționat este reprezentat în figura:



Se va afișa deci: *EAATSTSEECA*

- Considerăm o stivă S_1 în care se introduc pe rând caracterele A, D, C, B și două stive suplimentare S_2 și S_3 inițial vide. Să se scrie operațiile necesare pentru a trece caracterele din S_1 în S_2 , astfel încât la extragerea pe rând din S_2 și afișarea pe ecran, literele să apară în ordine alfabetică.

Rezolvare: Ordinea operațiilor pe cele trei stive este reprezentată în figura:



- Se consideră o coadă C . Presupunem că asupra cozii au loc o serie de operații de *push* și *pop*. În cazul operațiilor de *push* sunt adăugate în coadă numere de la 0 la 9 în această ordine. Înainte de fiecare operație de *pop* se afișează elementul care urmează să fie extras. Este posibil ca să fie afișată secvența de valori: 4, 3, 2, 1, 0, 5, 6, 7, 8, 9? (Obs: Nu se efectuează *pop* pe coadă vidă).

Rezolvare: Nu. Această secvență nu poate să apară, deoarece într-o coadă elementele sunt extrase în ordinea în care au fost inserate, iar elementele au fost inserate în ordine crescătoare.

4. Scrieți un algoritm în pseudocod care realizează căutarea unui element x într-o coadă folosind o altă coadă suplimentară, folosind doar operațiile specifice cozilor.

Rezolvare: Se parcurg toate elementele cozii principale C prin scoatere și se introduc într-o coadă auxiliară C_{aux} , păstrând ordinea elementelor. La fiecare pas se verifică dacă elementul curent este egal cu valoarea căutată. Apoi se vor transfera elementele în coada inițială.

Algoritm: CAUTA

Input: Coada C , o valoare val de căutat.

```

start
   $gasit \leftarrow false$ 
   $C_{aux} \leftarrow \emptyset$ 
  cat timp  $C \neq \emptyset$  executa
    daca  $PRIM(C) = val$  atunci
       $gasit \leftarrow true$ 
    sfarsit_daca
       $ADAUGA(C_{aux}, PRIM(C))$ 
       $EXTRAGE(C)$ 
    sfarsit_cat_timp
  cat timp  $C_{aux} \neq \emptyset$  executa
     $ADUGA(C, PRIM(C_{aux}))$ 
     $EXTRAGE(C_{aux})$ 
  sfarsit_cat_timp
  return  $gasit$ 
stop

```

5. Scrieți un algoritm în pseudocod care implementează o coadă folosind două stive și operații pe stivă.

Rezolvare: Pentru simularea comportamentului de coadă, se folosesc două stive S_1 și S_2 , astfel vom lucra cu o variabilă C care reține cele două stive. Operația de inserare presupune adăugarea elementului în stiva S_1 .

Algoritm: ADAUGA

Input: Structura C ce reține cele două stive S_1 și S_2 , o valoare val de adăugat.

```

start
   $ADAUGA(C.S_1, val)$ 
stop

```

Pentru extragerea primului element inserat, cum S_2 este goală, transferăm toate elementele din S_1 în S_2 și extragem din S_2 elementul din vârf. Apoi mutăm la loc elementele din S_2 în S_1 .

Algoritm: EXTRAGE

Input: Structura C ce reține cele două stive S_1 și S_2 .

```

start
  daca  $C.S_1 = \emptyset$  atunci
    | returneaza("Eroare de coada vida")
  sfarsit_daca
  //Muta elementele din  $S_1$  în  $S_2$ 
  cat_timp  $C.S_1 \neq \emptyset$  executa
    |  $ADAUGA(C.S_2, C.S_1.varf)$ 
    |  $EXTRAGE(C.S_1)$ 
  sfarsit_cat_timp
   $EXTRAGE(C.S_2)$ 
  //Muta la loc elementele din  $S_2$  în  $S_1$ 
  cat_timp  $C.S_2 \neq \emptyset$  executa
    |  $ADAUGA(C.S_1, C.S_2.varf)$ 
    |  $EXTRAGE(C.S_2)$ 
  sfarsit_cat_timp
stop

```

Pentru accesarea primului element din coadă algoritmul este similar cu cel de ștergere.

Algoritm: PRIM

Input: Structura C ce reține cele două stive S_1 și S_2 .

```

start
  cat_timp  $C.S_1 \neq \emptyset$  executa
    |  $ADAUGA(C.S_2, C.S_1.varf)$ 
    |  $EXTRAGE(C.S_1)$ 
  sfarsit_cat_timp
   $rez = C.S_2.varf$ 
  cat_timp  $C.S_2 \neq \emptyset$  executa
    |  $ADAUGA(C.S_1, C.S_2.varf)$ 
    |  $EXTRAGE(C.S_2)$ 
  sfarsit_cat_timp
  returneaza( $rez$ )
stop

```

Capitolul 2

Tabele de dispersie

Introducere

Așa cum s-a menționat în primul capitol al acestui curs, structurile de date au fost dezvoltate cu scopul memorării și manipularii *eficiente* a unei mulțimi dinamice de date. Această manipulare eficientă depinde desigur semnificativ de scopul urmărit și de tipurile de operații care se doresc a fi efectuate.

Una dintre principalele operații într-o colecție de date este căutarea sau *accesul prin cheie*. În cazul mulțimilor sortate, căutarea binară permite găsirea unei anumite chei în complexitate logaritmică. Vom vedea în capitolele ce tratează arborii binari de căutare, cum poate fi rezolvată în complexitate logaritmică problema căutării în cazul mulțimilor dinamice de date, deci a mulțimilor care se modifică prin inserții și ștergeri.

Tabelele de dispersie pot fi utilizate cu succes pentru implementarea de dicționare în care căutarea este cea mai frecventă operație. O altă utilizare a tabelelor de dispersie este în cadrul compilatoarelor pentru implementarea tabelii de identificatori.

În capitolul de față sunt prezentate tabele de dispersie - *hash tables*, *hash maps* - structuri de date care permit accesul prin cheie în timp constant. Aceste structuri de date sunt eficiente, atunci când nu este necesară păstrarea ordonată a datelor.

Competențe

La sfârșitul acestui modul de învățare studenții:

- Definesc tabelele de dispersie în general și știu cum se utilizează.
- Selectează o funcție de dispersie adecvată pentru rezolvarea problemei coliziunilor.
- Estimează este complexitatea operațiilor de bază.

2.1 Unitatea de învățare 1 - Tabele de dispersie cu liste înlanțuite



2.1.1 Introducere

O tabelă de dispersie este de fapt o generalizare a noțiunii de vector - *array*. Operațiile de bază sunt inserția, căutarea și eventual ștergerea. Vom vedea pe parcurs că, într-o tabelă de dispersie operațiile au în medie complexitatea $O(1)$. O tabelă de dispersie se memorează într-un *array*. În această unitate vom prezenta întâi tabelele cu adresare directă, utilizabile în contextul restrâns al cheilor de tip număr natural, cu universul cheilor posibile relativ mic. Vom vedea cum se rezolvă problema plasării cheilor într-o tabelă pentru cazul în care universul cheilor posibile este foarte mare, folosind funcții de dispersie și vom discuta problema rezolvării coliziunilor prin liste înlanțuite.



2.1.2 Obiective

La sfârșitul acestei unități de învățare studenții vor înțelege:

- Ce este o tabelă cu adresare directă.
- Ce este o funcție de dispersie și ce este o tabelă de dispersie.
- Ce este o coliziune și cum se poate rezolva.
- Care sunt operațiile de bază și care este complexitatea lor.



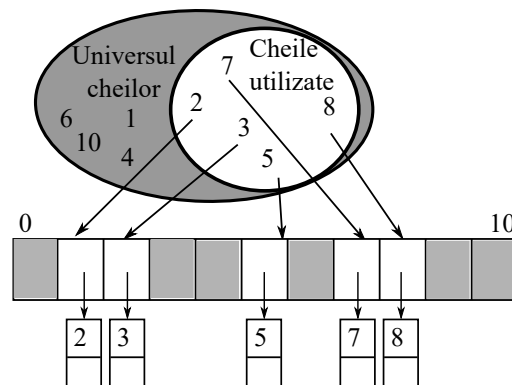
Durata medie de studiu individual

Parcurea de către studenți a acestei unități de învățare se face în 2 oră.

2.1.3 Tabele cu adresare directă

Considerăm $U = \{0, 1, \dots, m - 1\}$ universul cheilor posibile, m relativ mic și tabloul $T[0, \dots, m - 1]$ în care se memorează elementele. Pentru a putea găsi o cheie în timp constant, cea mai simplă metodă de repartizare a cheilor într-o astfel de tabelă este, să plasăm elementul cu cheia k pe poziția k în T . Dacă în tabelă nu există cheia k , atunci $T[k] = nil$. O astfel de tabelă se numește *tabelă cu adresare directă*.

Algoritmii pentru operațiile de căutare, inserție și ștergere sunt extrem de simpli [4].

Algorithm: TA_CAUTA**Input:** tabela T și cheia k **start**| returnează($T[k]$)**stop****Algorithm:** TA_INSEREAZA**Input:** tabela T și elementul x **start**| $T[x.cheie] \leftarrow x$ **stop****Algorithm:** TA_STERGE**Input:** tabela T și elementul x **start**| $T[x.cheie] \leftarrow nil$ **stop****Complexitate:** evident complexitatea este $O(1)$ pentru toate cele trei operații.**Exemplu** de tabelă cu adresare directă.**Observații:**

- Se presupune faptul că fiecare element x are o cheie unică, diferită de cheile tuturor celorlalte elemente din tabelă.
- O tabelă cu adresare directă este potrivită, atunci când universul U al cheilor este relativ redus și numărul de elemente memorate în tabelă este comparabil cu $|U|$.
- De obicei elementele dintr-o tabelă cu adresare directă sau dintr-o tabelă de dispersie au un câmp pentru cheie, după care se face căutarea, și un câmp pentru informația de interes. Uneori cheia corespunde informației și atunci este suficient un singur câmp.

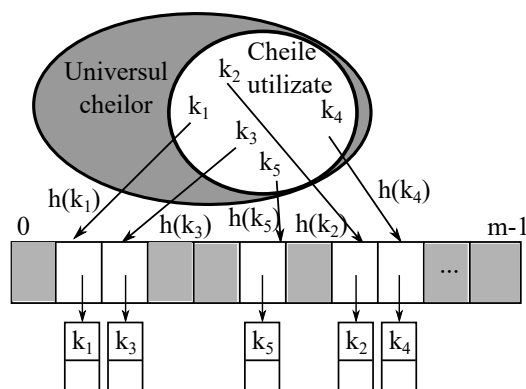


Figura 2.1: Tabelă de dispersie care utilizează funcția de dispersie h

2.1.4 Tabele de dispersie - *Hash Tables*

Pentru situații în care numărul de elemente stocate într-o tabelă este mult mai mic decât universul cheilor, de exemplu atunci când orice număr natural poate fi cheie, tabelele cu adresare directă devin practic inutilizabile, datorită cantității de memorie necesare, care poate fi astfel nelimitată. În această situație se înlocuiește tabela cu adresare directă cu o tabelă de dispersie.

Pentru memorarea unei tabele de dispersie se utilizează de asemenea un *array* T de dimensiune m : $T[0, \dots, m - 1]$. Accesul la un element se face prin intermediul unei funcții de dispersie (repartizare) - *hash function* - $h : U \rightarrow \{0, 1, \dots, m - 1\}$, $U \gg m$.

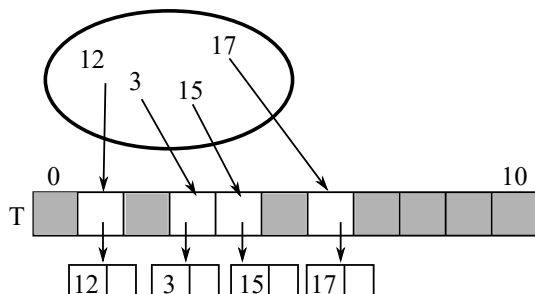
Elementul cu cheia k va fi plasat în tabelă pe poziția $T[h(k)]$. Spunem că, elementul cu cheia k este repartizat pe poziția $h(k)$ (*hashes to slot $h(k)$*). În figura 2.1 este reprezentată grafic o tabelă de dispersie.

O funcție foarte simplă pentru repartizarea numerelor naturale într-o tabelă de dispersie este dată prin funcția *modulo* (mod), care reprezintă restul împărțirii la un număr natural.

$$h : U \rightarrow \{0, 1, \dots, m - 1\}, h(k) = k \bmod m$$



Exemplu Considerând $m = 11$ și cheile $\{3, 12, 15, 17\}$ atunci: $h(3) = 3$, $h(12) = 1$, $h(15) = 4$, $h(17) = 6$



2.1.5 Problema coliziunilor

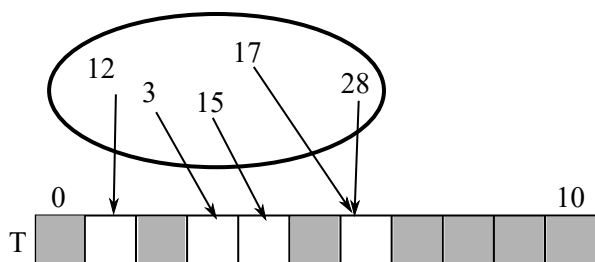


Figura 2.2: Coliziune între cheile 28 și 17.

În cazul unei funcții de dispersie h este posibil ca pentru două chei diferite k_1 și k_2 să se obțină $h(k_1) = h(k_2)$, adică ambele elemente ar fi repartizate pe aceeași poziție. O astfel de situație se numește *coliziune*. În exemplul de mai sus $h(28) = h(17) = 6$ (fig. 2.2).

Se pune astfel problema rezolvării coliziunilor. Ideal ar fi, evitarea completă a acestora. O funcție de dispersie *ideală* ar trebui să producă pentru orice cheie k o valoare aleatorie $h(k)$ (dar mereu aceeași pentru un k dat), aleasă uniform în intervalul $[0, m - 1]$. În plus, modul de repartizare a unei chei k în tabelă nu depinde de modul de repartizare a niciunei alte chei. O astfel de funcție ideală se numește *funcție de dispersie uniformă și independentă* (*uniform independent hash function*).

Din faptul că se presupune că $|U| > m$, prin tipul de repartizare descris mai sus acest lucru nu este posibil în realitate. Totuși, prin construirea atentă a funcției de dispersie, poate fi redusă semnificativ probabilitatea unei coliziuni. Vor fi discutate pe parcursul acestui curs câteva metode de construcție a unei funcții de dispersie potrivite.

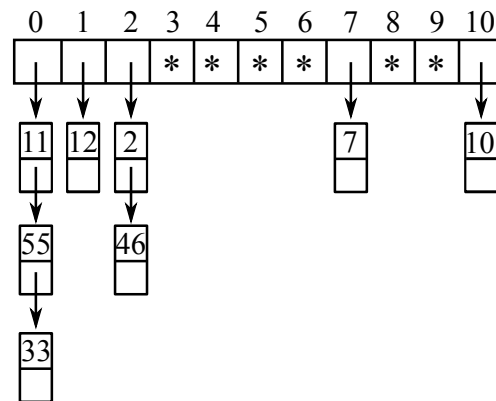
În practică rezolvarea coliziunilor pentru tabele de dispersie poate fi făcută de exemplu cu

ajutorul listelor înlănțuite. Astfel, în loc de a stoca într-o poziție a tabelului de repartizare un singur element, se păstrează o listă de elemente (ideea de *bucket*). Lista de la poziția k va conține toate elementele a căror cheie determină repartizarea pe poziția respectivă.

O altă metodă de rezolvare a coliziunilor este *adresare deschisă*, metodă prezentată în unitatea următoare.



Exemplu: Se consideră o tabelă de dispersie T cu dimensiunea $m = 11$ care utilizează liste înlănțuite și în care se introduc elementele cu cheile 33, 35, 55, 46, 12, 2, 7, 10, 11. Rezultatul folosind inserția la începutul listelor este:



Au fost marcate cu * pozițiile neocupate din tabelă.

2.1.6 Operațiile uzuale

În continuare sunt prezentați algoritmi pentru operațiile de căutare, inserare și ștergere a unui element dintr-o tabelă de dispersie ce utilizează liste înlănțuite pentru rezolvarea coliziunilor. Se consideră că fiecare element x din tabelă este o structură cu două câmpuri $x.cheie$ pentru cheie și $x.info$ pentru informația de interes. În algoritmi se consideră notația $H - Lista$ pentru o tabelă de dispersie cu liste înlănțuite.

Inserția unui element în tabelă de dispersie T se realizează prin adăugarea unui element x cu cheia k la începutul listei $T[k]$. Acest algoritm presupune faptul că, nu vor apărea chei identice în timpul inserției. Dacă dorim să ne asigurăm de unicitatea cheii, acest lucru se poate face prin căutare în lista $T[k]$, în complexitate dată de lungimea acestei liste.

Algorithm: H_LISTE_INSERTEAZA

Input: O tabelă T de dimensiune m , cu funcția de dispersie h , elementul de inserat x .**start**| ADAUGĂ($T[h(x.cheia)], x$) //adaugă x în lista $T[h(x.cheia)]$ **stop**

Algorithm: H_LISTE_CAUTA

Input: O tabelă T de dimensiune m , cu funcția de dispersie h , cheia căutată $cheie$.**Output:** Un pointer la elementul x cu cheia k sau nil .**start**| returneaza($CAUTĂ(T[h(k)], x)$)**stop**

Algorithm: H_LISTE_STERGE

Input: O tabelă T de dimensiune m , cu funcția de dispersie h , elementul de șters x .**start**| ȘTERGE($T[h(x.cheia)], x$) //șterge x din lista $T[h(x.cheia)]$ **stop**

Complexitate:

- funcția de inserție are evident complexitatea $O(1)$;
- la funcția de căutare complexitatea depinde de lungimea listelor înlănțuite;
- funcția de ștergere, care are ca parametru un nod x deja identificat în lista înlănțuită, are complexitatea $O(1)$ dacă lista e dublu înlănțuită, altfel depinde de lungimea listei. Dacă întâi trebuie căutat elementul cu cheia k , apoi șters, se adaugă complexitatea căutării.

2.1.7 Analiza complexității

Considerăm tabela de dispersie T cu m poziții, în care rezolvarea coliziunilor se realizează prin înlănțuire. Presupunem că în T se stochează n elemente. Atunci numărul mediu de elemente stocate într-o listă este $\alpha = n/m$, unde α poate fi mai mic sau mai mare decât 1. α se numește *factor de încărcare* al tablei.

Cea mai defavorabilă situație este aceea când toate cele n elemente se repartizează prin h pe aceeași poziție, adică sunt stocate în aceeași listă. În această situație complexitatea la căutare este $\Theta(n)$. Pentru a evita astfel de situații este necesară alegerea unei funcții de dispersie potrivită. Modul de construcție a unor funcții de dispersie care să permită o repartizare

cât mai uniformă va fi discutată ulterior. Dispersia uniformă presupune ca probabilitatea ca un anumit element să fie de repartizat pe oricare dintre pozițiile din tabelă este aceeași. Vom presupune în continuare că dispersia este uniformă și independentă, unde independentă înseamnă că repartizarea unei chei nu depinde de repartizarea niciunei alte chei.

Notăm cu n_j lungimea listei de pe poziția $T[j]$, $j = \overline{0, m-1}$. Observăm că $n_0 + n_1 + \dots + n_{m-1} = n$, iar valoarea medie sau valoarea **estimată** [4] pentru lungimea oricărei liste n_j este dată prin: $E[n_j] = n/m = \alpha$.

Considerând o funcție de dispersie care repartizează cheile uniform în tabela T și pentru care calculul lui $h(k)$ este $O(1)$, se demonstrează faptul că funcția de căutare a unei chei este $\Theta(1 + \alpha)$ cu ajutorul următoarelor două teoreme.

Teorema 2.1.1 *Într-o tabelă de dispersie care tratează coliziunile prin înlănțuire și în care repartizarea este uniformă și independentă, o căutare fără succes are complexitatea medie de timp $\Theta(1 + \alpha)$. [4]*

Demonstrație: Presupunem că se caută în T cheia k , care nu există. Datorită faptului că s-a presupus o repartizare uniformă cu h , probabilitatea repartizării cheii k pe oricare dintre poziții este aceeași. Timpul necesar pentru a constata că această cheie nu este în T este timpul necesar de parcurgere a listei $T[h(k)]$. În medie acest timp depinde de lungimea medie a listei $h(k)$, deci de $E[n_h(k)] = \alpha$. Dat fiind că se ține cont și timpul necesar pentru calculul valorii $h(k)$ se obține o complexitate de $\Theta(1 + \alpha)$.

Teorema 2.1.2 *Într-o tabelă de dispersie care tratează coliziunile prin înlănțuire și în care repartizarea este uniformă și independentă, o căutare cu succes are complexitatea medie de timp $\Theta(1 + \alpha)$ [4].*

Observație Situația căutării cu succes diferă de cea a căutării fără succes prin faptul că, timpul de căutare al elementului x cu cheia k din tabelă depinde de numărul de elemente aflate în lista $T[h(k)]$ înaintea lui x . Modul de calcul al acestui timp mediu presupune ceva cunoștințe de probabilități și depășește cadrul acestui curs. O descriere detaliată poate fi găsită în bibliografie [4].

Dacă numărul de elemente stocate în tabelă, n , este proporțional cu dimensiunea m a tabelii, adică $n = am$, atunci complexitatea la căutare devine $\Theta(1 + am/m) = \Theta(1 + a) = \Theta(1)$. Cu alte cuvinte, dacă funcția de dispersie este aleasă în mod potrivit, astfel încât să ofere o dispersie uniformă și independentă și dacă dimensiunea tabelii este aleasă proporțional cu numărul estimat de elemente ce vor fi introduse în tabelă, timpul de căutare a unei chei devine în medie constant.

2.1.8 Metode de dispersie

Din discuția complexității rezultă că, o funcție de dispersie bună permite o repartizare aproape uniformă în tabelă. Pentru acest lucru poziția obținută la repartizare ar trebui să fie independentă de orice **pattern** care ar putea fi prezent în setul de date.

În plus, funcțiile de dispersie au ca mulțime a valorilor poziții în tabela de dispersie, deci elemente din mulțimea numerelor naturale. În cazul în care cheile sunt de exemplu șiruri de caractere - situație frecventă de altfel - este necesară stabilirea unei metode de interpretare a acestora ca numere naturale, trebuie deci determinată o funcție de la universul cheilor către mulțimea numerelor naturale. La finalul capitoului vor fi prezentate câteva funcții de dispersie pentru *string*-uri.

Considerarea unei funcții de dispersie fixată pentru orice tip de date de intrare se numește *static hashing* și are aplicabilitate limitată. Alegerea în mod aleatoriu a unei funcții de dispersie la execuția programului se numește *random hashing* și oferă rezultate mai bune.

Metode statice

În continuare sunt prezentate două metode pentru construcția unor funcțiilor de dispersie pentru numere naturale, *metoda diviziunii* și *metoda multiplicării*.

(I) Metoda diviziunii

Funcția de dispersie în acest caz este dată prin:

$$h : U \rightarrow \{0, 1, \dots, m - 1\}, h(k) = k \bmod m$$

Alegerea dimensiunii tablei. O alegere bună pentru dimensiunea tablei m este un număr prim nu prea apropiat de o putere a lui 2.

Justificare. Dacă s-ar alege $m = 2^p$ atunci de fapt împărțirea la m ar reprezenta o *shift*-are a biților cheii, ceea ce ar crea o dependență de *pattern*-uri în structura cheilor, lucru ce trebuie evitat.

În plus acest număr trebuie ales depinzând de numărul de elemente care se estimează că vor fi introduse în tabelă, precum și de factorul de încărcare dorit.

Exemplu: dacă $n = 3000$ și numărul mediu de elemente per poziție este considerat 4 atunci, $\alpha = n/m \Rightarrow m = n/\alpha = 3000/4 = 750$. Se poate considera $m = 751$, care este un număr prim nu prea apropiat de o putere a lui 2.

Această metodă ar putea funcționa satisfăcător, dacă m este număr prim nu prea apropiat de o putere a lui 2, fapt care complică implementarea practică.

(II) Metoda multiplicării

În acest caz funcția de dispersie este de tipul [4]:

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor, 0 < A < 1$$

unde $\lfloor x \rfloor$ reprezintă partea întreagă a lui x .

În cazul acestor funcții modul de alegere a lui m nu influențează modul de repartizare. Se demonstrează [10] faptul că, o alegere bună pentru A este

$$A = (\sqrt{5} - 1) / 2 \approx 0.618033$$

În practică metoda multiplicativă funcționează cel mai bine atunci când m este o putere a lui 2.

Metoda multiplicare-deplasare

O eficientizare din punct de vedere al calculului a metodei multiplicative pentru $m = 2^l$ este metoda multiplicare-deplasare (*the multiply-shift method*) [5], care se folosește de operații de shift-are pe biți.

Problema este următoarea: dat fiind o tabelă T de dimensiune 2^l , pentru orice $0 < k < 2^w$, unde w este numărul de biți necesari pentru stocarea oricărui cuvânt din universul cheilor U (de exemplu 32 sau 64), să se determine poziția de inserție în tabelă pentru k . Această poziție va fi evident în intervalul $[0, 2^l]$, adică pe l biți.

O funcție potrivită pentru aceasta poate fi considerată $h_a(k) : [0, 2^w] \rightarrow [0, 2^l]$ dată prin [5]:

$$h_a(k) = \lfloor ak \bmod 2^w \rfloor / 2^{w-l} \quad (2.1)$$

unde a este un număr natural impar. O valoare pentru a este $a = 2654435769$ propusă în [4]. Formula 2.1 poate fi implementată prin:

$$h_a(k) = (ak \bmod 2^w) \gg (w - l) \quad (2.2)$$

unde \gg reprezintă shiftarea la dreapta cu un număr de biți, în cazul de față $(w - l)$. Operația de modulo ne asigură că rămânem în mulțimea U . Dacă w este numărul de biți alocați de sistem pentru stocarea unui cuvânt, atunci ecuația 2.2 devine:

$$h_a(k) = (ak) \gg (w - l) \quad (2.3)$$

deoarece sistemul trunchiază orice valoare mai mare de 2^w în mod echivalent operației modulo, iar deplasarea (shiftarea) are efectul împărțirii. Astfel numărul de operații se reduce la o înmulțire și o diviziune de numere naturale precum și o shiftare pe biți, ducând la un calcul foarte eficient al funcției de dispersie.



Implementați o tabelă de dispersie care stochează numere naturale și care rezolvă coliziunile prin liste înlănțuite. Folosiți pentru selectarea poziției de inserție metoda multiplicare-deplasare.

2.1.9 Metode dinamice

Dacă repartizarea în tabelă este realizată printr-o funcție dată, există posibilitatea ca, pentru anumite seturi de date, toate cheile să fie repartizate pe aceeași poziție, ajungându-se din nou la complexitatea $\Theta(n)$ la căutare.

Acest lucru poate fi evitat, dacă în loc să se folosească o singură funcție de dispersie, se utilizează o mulțime H de funcții de dispersie. La fiecare execuție se selectează în mod aleatoriu una dintre funcțiile de dispersie din H care va fi apoi utilizată. Această metodă se numește *random hashing*.

Dispersie universală

Utilizarea dispersiei universale are ca efect faptul că, pentru același set de date de intrare, execuții diferite ale programului produc în general tabele diferite, astfel putându-se evita în orice situație cel mai defavorabil caz, eventual prin reluare a dispersiei - **rehashing**.

Definiție: O mulțime finită de funcții de dispersie H , care repartizează cheile dintr-un univers U într-o tabelă $T[0, \dots, m-1]$ se numește universală, dacă pentru orice două chei k și l , $k \neq l$, numărul de funcții din H pentru care $h(k) = h(l)$ este cel mult $|H|/m$ [4].

Acest lucru asigură faptul că, pentru o funcție aleasă în mod aleatoriu din H , probabilitatea unei coliziuni între k și l este cel mult $1/m$. Evident, probabilitatea alegerii oricărei funcții din H este $1/|H|$. Probabilitatea alegerii unei funcții care repartizează k și l pe aceeași poziție este cel mult $(|H|/m) * 1/|H| = 1/m$.

Construcția unor clase universale de funcții de dispersie

(1) **Pe baza metodei diviziunii.** Se alege un număr prim p suficient de mare, astfel încât să fie mai mare decât orice cheie posibilă. Evident $p \gg m$. Pentru fiecare $a \in \{1, \dots, p-1\}$

și $b \in \{0, 1, \dots, p-1\}$ se definește funcția de dispersie [4]:

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m)$$

Mulțimea de astfel de funcții de dispersie este

$$H_{pm} = \{h_{ab} | a \in \{1, \dots, p-1\} \text{ și } b \in \{0, 1, \dots, p-1\}\}$$

cu $|H_{pm}| = p(p-1)$.

Se poate demonstra că mulțimea astfel definită este universală [4].

(2) Pe baza metodei multiplicative (cu shiftare). O familie de funcții de dispersie foarte eficientă poate fi construită pornind de la ecuația 2.2:

$$H = \{h_a | a \text{ numar impar} \}$$

unde

$$h_a(k) = (ak \bmod 2^w) \gg (w-l)$$



Implementați o tabelă de dispersie care folosește dispersia universală. Construiți mulțimea de funcții de dispersie folosind metoda *multiply-shift*.



Să ne reamintim...

- Tabelele de dispersie reprezintă o generalizare a vectorului, în care accesul prin cheie se realizează eficient.
- În tabelele cu adresare directă, elementul cu cheia k se plasează pe poziția k . Accesul se realizează în timp constant, dar ele pot fi folosite doar în context foarte restrâns.
- În tabelele de dispersie determinarea poziției de inserție a unui element se realizează cu ajutorul unei funcții de dispersie.
- Rezolvarea coliziunii între 2 chei se poate realiza prin folosirea de liste înlănțuite sau prin adresare deschisă.
- Factorul de încărcare permite estimarea complexității operațiilor.

2.1.10 Rezumat



În această secțiune au fost discutate tabelele de dispersie cu liste înlănțuite. Au fost prezentate noțiunile de bază, modul de determinare a poziției unei chei într-o tabelă, rezolvarea coliziunilor cu ajutorul listelor înlănțuite și câțiva algoritmi în pseudocod pentru implementarea acestor structuri împreună cu complexitatea acestora, determinată de factorul de încărcare.



2.1.11 Test de autoevaluare

1. Se consideră o tabelă de dispersie de dimensiune $m = 8$ și funcția hash: $h(k) = k \bmod m$. Să se determine structura finală a tabelii după inserarea cheilor: 2, 7, 15, 10, 4, 6, 8 utilizând metoda rezolvării coliziunilor prin liste înlănțuite. Care este factorul de încărcare α al tabelii obținute?
2. Dacă într-o tabelă, lista de indice 2 conține elementele: $12 \rightarrow 22 \rightarrow 32 \rightarrow 42 \rightarrow 52$, care este numărul de comparații necesar pentru a găsi cheia 52?
3. Explicați modul în care complexitatea de timp și spațiu a operațiilor de căutare și inserare depind de factorul de încărcare α al tabelii.
4. Să se scrie un algoritm în pseudocod pentru copierea tuturor elementelor dintr-o tabelă de dispersie într-o tabelă nouă (operația de *rehashing*).

2.1.12 Răspunsuri la testul de evaluare a cunoștințelor

1. Se consideră o tabelă de dispersie de dimensiune $m = 8$ și funcția hash: $h(k) = k \bmod m$. Să se determine structura finală a tabelii după inserarea cheilor: 2, 7, 15, 10, 4, 6, 8 utilizând metoda rezolvării coliziunilor prin liste înlănțuite. Care este factorul de încărcare α al tabelii obținute?

Rezolvare: Se va calcula valoarea funcției de hash $h(k) = k \bmod m$, $m = 8$ pentru fiecare din dintre valorile de intrare:

$$h(2) = 2 \bmod 8 = 2$$

$$h(7) = 7 \bmod 8 = 7$$

$$h(15) = 15 \bmod 8 = 7$$

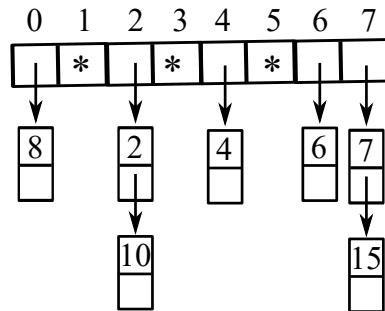
$$h(10) = 10 \bmod 8 = 2$$

$$h(4) = 4 \bmod 8 = 4$$

$$h(6) = 6 \bmod 8 = 6$$

$$h(8) = 8 \bmod 8 = 0$$

Forma finală a tabelii este reprezentată în figura:



Factor de încărcare: $\alpha = \frac{\text{număr chei inserate}}{\text{lungimea tabelii}} = \frac{7}{8} = 0.875$

2. Dacă într-o tabelă, lista de indice 2 conține elementele: $12 \rightarrow 22 \rightarrow 32 \rightarrow 42 \rightarrow 52$, care este numărul de comparații necesar pentru a găsi cheia 52?

Rezolvare: Pentru căutarea cheii 52, aplicăm mai întâi funcția de hash, care indică faptul că aceasta se poate afla pe poziția 2 din cadrul tabelii. Următorul pas constă în căutarea elementului la nivelul listei asociate acelei poziții. Această operație presupune compararea fiecărui element al listei cu valoarea 52, până la parcurgerea întregii liste sau până la găsirea valorii căutate. În cazul de față, pentru a găsi 52, este necesar să parcurgem toate cele 5 noduri: $12 \rightarrow 22 \rightarrow 32 \rightarrow 42 \rightarrow 52$. Rezultă astfel un număr total de 5 comparații.

3. Explicați modul în care complexitatea de timp și spațiu a operațiilor de căutare și inserare depind de factorul de încărcare α al tabelii.

Rezolvare: În cazul tabelii de dispersie care rezolvă coliziunile utilizând liste înlănțuite, timpul mediu al operației de căutare este determinat de formula: $T_{\text{mediu}} \approx O(1 + \alpha)$ (în cazul repartizării uniforme a elementelor). Cum $\alpha = n/m$, n = numărul de elemente stocate în listă și m dimensiunea tabelii, dacă m este proporțional cu n complexitatea de timp devine constantă. Dacă elementele nu sunt repartizate uniform sau α este foarte mare, deci listele devin mai lungi, timpul de căutare crește, putând ajunge la $O(n)$ în cel mai defavorabil caz.

Complexitatea operației de inserție nu este influențată de factorul de încărcare, întrucât inserarea elementelor în listă se face în timp constant $O(1)$.

Complexitatea de spațiu crește proporțional cu numărul elementelor inserate în tabelă.

4. Să se scrie un algoritm în pseudocod pentru copierea tuturor elementelor dintr-o tabelă de dispersie într-o tabelă nouă (operația de *rehashing*).

Rezolvare: Algoritm în pseudocod pentru rehashing (copierea elementelor într-o tabelă nouă de dimensiune dublă). Prin h_2 s-a notat funcția ce calculează poziția de inserție a unei chei în noua tabelă H_2 . Această funcție combină atât calculul valorii de *hash* pentru o cheie care nu e de tip număr natural (dacă este cazul), precum și determinarea poziției în tabelă folosind metoda *multiply-shift* (dat fiind că dimensiunea tablei este putere a lui 2).

Algoritm: L-HASH_REHASHING

Input: Tabela de dispersie H .

start

$dim_noua \leftarrow H.dim * 2$

 alocă memorie pentru noua tabelă H_2 de dimensiune dim_noua

pentru $i \leftarrow 0, H.size - 1$ **executa**

$nod = H[i]$

cat timp $nod \neq nil$ **executa**

$index \leftarrow h_2(nod.cheie)$ //calculează poziția de inserție

 //conform funcției de dispersie pentru noua tabelă

 ADAugĂ($H_2[index]$, $nod.cheie$)

$nod \leftarrow nod.urmator$

sfarsit_cat_timp

sfarsit_for

 eliberează memoria pentru H

$H \leftarrow H_2$

stop

2.2 Unitatea de învățare 2 - Tabele de dispersie cu adresare deschisă



2.2.1 Introducere

În această unitate vom discuta despre tabele de dispersie în care rezolvarea coliziunilor se realizează prin adresare deschisă.

În cazul adresării deschise, fiecare poziție a tabelului de dispersie T conține un singur element. Pentru rezolvarea coliziunilor la inserție nu se utilizează liste înlănțuite, ci se testează diferite poziții, obținute pe baza funcției de dispersie, până când se determină o poziție liberă, pe care poate fi inserat elementul dorit. La căutare se testează de asemenea diferite poziții până când se găsește elementul căutat sau se ajunge la o poziție neocupată.

Faptul că nu se utilizează liste înlănțuite, ci fiecare poziție conține cel mult un element, are ca urmare posibilitatea umplerii tabelului. În schimb se evită pointerii, iar memoria, care altfel ar fi fost necesară pentru listele înlănțuite, poate fi utilizată pentru o tabelă de dimensiune mai mare.

Un alt avantaj al acestui tip de dispersie apare în contextul utilizării memoriei *cache* a calculatoarelor moderne. La capitolul despre liste a fost explicat dezavantajul acestora din punctul de vedere al vitezei de acces la memorie.



2.2.2 Obiective

La sfârșitul acestei unități de învățare studenții vor înțelege:

- Ce este o tabelă cu adresare deschisă și cum se realizează rezolvarea coliziunilor prin testarea pozițiilor.
- Cum se construiesc funcții de dispersie potrivite pentru o dispersie uniformă;
- Care sunt operațiile de bază și care este complexitatea lor.
- Cum se construiesc funcții de dispersie pentru chei de tip vectorial.



Durata medie de studiu individual

Parcursul de către studenți a acestei unități de învățare se face în 2 ore.

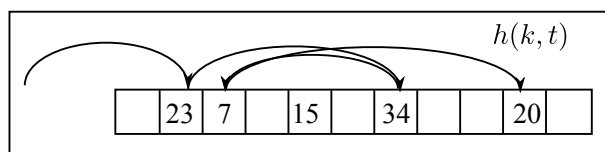


Figura 2.3: Căutarea elementului cu cheia 20. Acesta este găsit după 3 teste.

2.2.3 Testarea pozițiilor

Modul de testare a pozițiilor în tabelă în cazul adresării deschise nu este liniar ci depinde atât de cheia inserată cât și de numărul de testări efectuate până la momentul curent. Forma generală a unei funcții de dispersie în cazul adresării deschise este:

$$h : U \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}$$

$h(k, t)$ = poziția de repartizare a cheii k după t teste.

În mod evident, pentru a permite testarea tuturor pozițiilor posibile și a nu reveni periodic la aceleași poziții, testate anterior, funcția $f(k, t)$ trebuie să producă pentru valorile lui $t \in \{0, \dots, m - 1\}$ o permutare a mulțimii pozițiilor $\{0, \dots, m - 1\}$. Căutarea unei chei într-o astfel de tabelă este ilustrată în figura 2.3.

În continuare sunt prezentați algoritmi generali pentru inserarea și căutare într-o tabelă cu adresare deschisă, construiți pe baza celor din [4].

Algoritm: TAD_INSERTEAZA

Input: O tabelă de dispersie T de dimensiune m , elementul x , unde $x.chieie = k$

start

```

    t ← 0 //nr de poziții testate
    repeta
        pos ← h(k, t) //testează o poziție nouă
        daca T[pos] = nil atunci
            T[pos] ← x
            returneaza(pos)
        sfarsit_daca
        t ← t + 1 //incrementează numărul de teste
    pana_cand t = m;
    returneaza(-1) //tabela este plină

```

stop

Algoritmul de căutare testează pentru o anumită cheie k aceeași secvență de poziții ca algoritmul de inserție al cheii k .

Algorithm: TAD_CAUTA**Input:** O tabelă de dispersie T de dimensiune m , cheia k **start** $t \leftarrow 0$ **repetă** $pos \leftarrow h(k, t)$ **dacă** $T[pos].cheie = k$ **atunci** | returnează(pos) **sfarsit_dacă** $t \leftarrow t + 1$ **pană_cand** $t = m$ sau $T[pos] = nil$; returnează(-1) // eșec**stop**

Operația de ștergere este problematică în cadrul adresării deschise. Dacă o cheie k este ștearsă prin marcarea poziției cu nil atunci o cheie p , inserată după k , dar care la inserție/ștergere presupune testarea poziției pe care s-a aflat k , nu va mai fi găsită prin algoritmul de mai sus.

O soluție a acestei probleme este marcarea pozițiilor șterse cu un marcaj special de poziție ștearsă, acest lucru presupune modificarea algoritmilor de căutare și inserare și în plus duce la o complexitate crescută a operației de căutare pentru un număr mare de poziții șterse. Pe măsură ce crește numărul de poziții marcate ca șterse, crește timpul de căutare și scade performanța. Acest lucru se numește *contaminare*. Singura posibilitate de rezolvare a acestei probleme este prin *rehashing*.

În ceea ce privește alegerea unei funcții de dispersie pentru adresarea deschisă, aceasta poate fi realizată în diferite moduri. Important este, ca pentru oricare cheie k probabilitatea unei secvențe de poziții testate să fie aceeași pentru oricare dintre cele $m!$ secvențe posibile. În acest caz vorbim despre dispersie de permutare uniformă și independentă.

În practică este greu de construit o astfel de funcție de dispersie, însă o variantă care se apropie de acest tip este adresarea deschisă cu dublă dispersie.

Adresarea deschisă cu dublă dispersie

Se consideră două funcții de dispersie auxiliare distincte $h_1(k)$ și $h_2(k)$. Se construiește funcția cu dublă dispersie:

$$h(k, t) = (h_1(k) + th_2(k)) \bmod m$$

Observații:

- poziția inițială testată este $h(k, 0) = h_1(k)$, deci nu depinde de h_2 , oricare ar fi cheia k ;
- poziția $h(k, t)$ este la distanța $h_2(k)$, față de poziția $h(k, t - 1)$, pentru orice $t > 0$;
- pentru ca secvența de poziții $\{h(k, 0), h(k, 1), \dots, h(k, m - 1)\}$ să fie o permutare a pozițiilor $\{0, 1, \dots, m - 1\}$ trebuie ca funcția $h_2(k)$ să producă valori care sunt prime față de m . Dacă există o cheie k , pentru care $h_2(k)$ și m au un divizor comun $1 < d < m$, atunci cel mult după d teste se repetă din nou testarea poziției inițiale. Adică vor rămâne în tabelă poziții netestate!!

Satisfacerea condiției discutate mai sus, se poate obține dacă

- m putere a lui 2 și $h_2(k)$ produce întotdeauna un număr impar
- m prim și $h_2(k)$ produce numere naturale din $\{0, 1, \dots, m - 1\}$, de exemplu considerăm funcțiile de dispersie

$$h_1(k) = k \bmod m$$

$$h_2(k) = 1 + (k \bmod m_1)$$

cu m prim și m_1 astfel încât m_1 ceva mai mic decât m , de exemplu $m_1 = m - 1$.



Exemplu: Considerând funcțiile de dispersie $h_1(k) = k \bmod 11$ și $h_2(k) = 1 + (k \bmod 10)$, cheile 22, 33, 45, 59, 67, 13, 71 vor fi repartizate după cum urmează:

$$h(22, 0) = (22 \bmod 11) \bmod 11 = 0$$

$$h(33, 0) = (33 \bmod 11) \bmod 11 = 0, \text{ este deja ocupat, deci continuă testarea}$$

$$h(33, 1) = ((33 \bmod 11) + (1 + 33 \bmod 10)) \bmod 11 = (0 + 4) \bmod 11 = 4$$

$$h(45, 0) = (45 \bmod 11) \bmod 11 = 1$$

$$h(59, 0) = (59 \bmod 11) \bmod 11 = 4, \text{ este deja ocupat, deci continuă testarea}$$

$$h(59, 1) = (59 \bmod 11 + 1 + 59 \bmod 10) \bmod 11 = (4 + 10) \bmod 11 = 3 \quad h(67, 0) = (67 \bmod 11) \bmod 11 = 1, \text{ este deja ocupat, deci continuă testarea}$$

$$h(67, 1) = ((67 \bmod 11) + (1 + 67 \bmod 10)) \bmod 11 = 9$$

$$h(13, 0) = (13 \bmod 11) \bmod 11 = 2$$

$$h(71, 0) = (71 \bmod 11) \bmod 11 = 6$$

Procesul de inserție a acestor chei, ilustrat grafic în continuare. Se observă că sunt necesare semnificativ mai puține încercări decât în cazul repartizării cu testare liniară.

Tabela inițială	22	33	45	59	67	13
0	22	22	22	22	22	22
1	/	/	45	45	45	45
2	/	/	/	/	/	13
3	/	/	/	59	59	59
4	/	33	33	33	33	33
5	/	/	/	/	/	/
6	/	/	/	/	/	/
7	/	/	/	/	/	/
8	/	/	/	/	/	/
9	/	/	/	/	67	67
10	/	/	/	/	/	/

Funcții de dispersie cu testare liniară

O variantă simplificată a adresării deschise cu dublă repartizare sunt funcțiile de dispersie cu testare liniară. Dacă se consideră o funcție de dispersie auxiliară $h_1 : U \rightarrow \{0, 1, \dots, m-1\}$, și $h_2(k) = 1$ atunci se obține:

$$h : U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}$$

$$h(k, t) = (h_1(k) + t) \bmod m$$



Exemplu: considerăm $m = 11$, $h_1(k) = k \bmod 11$ și

$$h(k, t) = ((k \bmod 11) + t) \bmod 11.$$

Atunci cheile 22, 33, 45, 59, 67, 13 vor fi repartizate după cum urmează:

$$h(22, 0) = (0 + 0) \bmod 11 = 0$$

$$h(33, 0) = (0 + 0) \bmod 11 = 0, \text{ este deja ocupat, deci testarea continuă}$$

$$h(33, 1) = (0 + 1) \bmod 11 = 1$$

$$h(45, 0) = (1 + 0) \bmod 11 = 1, \text{ este deja ocupat, deci testarea continuă}$$

$$h(45, 1) = (1 + 1) \bmod 11 = 2$$

$$h(59, 0) = (4 + 0) \bmod 11 = 4$$

$$h(67, 0) = (1 + 0) \bmod 11 = 1, \text{ este deja ocupat, deci testarea continuă}$$

$$h(67, 1) = (1 + 1) \bmod 11 = 2, \text{ este deja ocupat, deci testarea continuă}$$

$$h(67, 2) = (1 + 2) \bmod 11 = 3,$$

$$h(13, 0) = (2 + 0) \bmod 11 = 2, \text{ este deja ocupat, deci testarea continuă}$$

$$h(13, 1) = (2 + 1) \bmod 11 = 3, \text{ este deja ocupat, deci testarea continuă}$$

$$h(13, 2) = (2 + 2) \bmod 11 = 4 \text{ este deja ocupat, deci testarea continuă}$$

$$h(71, 0) = (2 + 3) \bmod 11 = 5$$

Tabela inițială	22	33	45	59	67	13
0	/	22	22	22	22	22
1	/	/	33	33	33	33
2	/	/	/	45	45	45
3	/	/	/	/	67	67
4	/	/	/	59	59	59
5	/	/	/	/	/	13
6	/	/	/	/	/	/
7	/	/	/	/	/	/
8	/	/	/	/	/	/
9	/	/	/	/	/	/
10	/	/	/	/	/	/

Observație: Dezavantajul testării liniare este acela că, pe măsură ce se adaugă elemente, cresc secvențele de poziții succesive ocupate - **primary clustering**, ceea ce duce la creșterea complexității de inserție/căutare.

Totuși, în cadrul sistemelor moderne cu memorie ierarhizată, acest clustering nu mai constituie un dezavantaj. Acest lucru va fi discutat în urma analizei complexității.

Analiza complexității

Fie o tabelă de dispersie cu factorul de încărcare $\alpha = n/m < 1$, unde n = numărul de poziții ocupate și m = dimensiunea tabelului. De asemenea presupunem că funcția de dispersie asigură repartizarea uniformă. Aceasta presupune că, pentru orice cheie k , toate cele $m!$ permutări ale mulțimii $0, 1, \dots, m - 1$ au aceeași probabilitate de a reprezenta ordinea în care sunt testate pozițiile din tabel. Complexitatea căutării unei chei în acest caz este determinată pe baza următoarelor două teoreme.

Teorema 2.2.1 Pentru o tabelă de dispersie cu adresare deschisă, cu factorul de încărcare $\alpha < 1$, numărul mediu de teste necesare pentru căutarea fără succes este cel mult $1/(1 - \alpha)$, presupunând o distribuție uniformă [4].

Teorema 2.2.2 Pentru o tabelă de dispersie cu adresare deschisă, cu factorul de încărcare $\alpha < 1$, numărul mediu de teste necesare pentru căutarea cu succes este cel mult $\frac{1}{\alpha} \log \frac{1}{1-\alpha}$, presupunând o distribuție uniformă [4].

Ambele teoreme sunt demonstrate cu ajutorul relațiilor probabilistice în [4].

Observație: În practică, dacă factorul de încărcare devine foarte mare, eficiența inserției, care presupune căutarea unei poziții neocupate, deci de fapt căutarea fără succes a cheii elementului ce se inserează, scade semnificativ. Astfel, atunci când α se apropie de 80% se recomandă redimensionarea tabelii și *rehashing*. De exemplu, în limbajul **Python**, care implementează dicționarele folosind adresarea deschisă, se pornește cu o tabelă de dimensiune 8 și atunci când s-a umplut 75% din tabelă, aceasta se redimensionează.

Eficiența în contextul memoriei ierarhizate

S-a discutat mai sus problema pe care o prezintă testarea liniară și anume, apariția în urma coliziunilor a unor grupuri de poziții succesive ocupate. Acest lucru produce în cazul multor coliziuni un timp crescut de căutare, mai ales în situația sistemelor mai vechi cu memorie RAM standard.

Sistemele CPU moderne au însă memoria de lucru ierarhizată, având unul sau mai multe niveluri de memorie *cache* rapidă. Cu cât nivelul de memorie este mai rapid, cu atât se stochează mai puțină informație, dar viteza de calcul este semnificativ mai mare. În afară de aceasta, memoria cache este organizată în blocuri de memorie succesive, încărcate din memoria principală. Astfel, păstrând calculele în interiorul aceluiași bloc de memorie devine extrem de eficient, mult mai eficient decât accesarea unei poziții noi aflate în alt bloc de memorie.

Din aceste considerente, deși la prima vedere testarea liniară pare mai puțin eficientă, ea devine o metodă foarte bună în contextul memoriei ierarhizate.

În plus, în cazul testării liniare există un algoritm simplu de tratare a ștergerilor din tabelă, fără a fi necesară marcarea pozițiilor șterse, care conduce la contaminare. Algoritmul poate fi găsit în [4]

2.2.4 *Perfect hashing*

O funcție de distribuție ideală va produce pentru orice două chei inserate poziții de inserție diferite. Există situații, atunci când sunt cunoscute toate cheile posibile (de exemplu în cazul unui dicționar), să fie construite funcții de dispersie ideale în care să se evite orice coliziune. Acest lucru se numește *perfect hashing*. În [4] este propus un algoritm de dispersie perfect, care are la bază tot ideea de *bucket*-uri similar cu rezolvarea coliziunilor prin înlănțuire, doar că, în loc de liste înlănțuite se utilizează tabele de dispersie secundare.

În cazul general acest lucru este practic imposibil, astfel încât se recomandă construirea de funcții de dispersie care să producă o repartizare uniformă a cheilor în tabelă, rezultând o

probabilitate mică de coliziune. Pentru chei numere întregi au fost prezentate câteva criterii de construcție a acestor funcții. În plus, o funcție de dispersie bună trebuie să utilizeze toate părțile unei chei pentru generarea valorii de dispersie. De exemplu, dacă se utilizează *string*-uri pentru chei, nu se vor considera doar primele 5 caractere pentru generarea valorii de dispersie sau dacă se folosesc chei numere întregi cu mai multe cifre, nu se vor utiliza doar anumite cifre ale acestora.



Implementați o tabelă de dispersie care stochează numere naturale și care rezolvă coliziunile prin adresare deschisă cu dublă dispersie.

2.2.5 Tratarea cheilor de tip vectorial (șir de caractere)

În partea teoretică prezentată până acum s-a considerat doar distribuirea de chei numere naturale. În practică există chei de diferite tipuri, frecvent caractere sau șiruri de caractere. Acestor chei trebuie mai întâi să li se asocieze un număr natural.

În primul rând putem considera pentru fiecare caracter ASCII codul său numeric. Problema este însă, cum combinăm codurile unei secvențe de caractere într-un număr natural, astfel încât să nu depășim valoarea maximă admisă și să evităm coliziunile care se produc, dacă pentru două șiruri de caractere se generează aceeași valoare numerică.

Funcții elementare de transformare a unui *string* într-un număr natural

Cele mai simple metode de a transforma o cheie de tip *string* într-o cheie numerică sunt *hashing*-ul aditiv și XOR-*hash*, descrise mai jos.

Hashing - aditiv - presupune adunarea codurilor numerice corespunzătoare tuturor caracterelor din șir. Acest mod de calcul al unei chei numerice este extrem de prost, deoarece generează coliziuni pentru toate permutările unui șir de caractere. În practică este deci inutilizabil.

XOR-*hash* - presupune aplicarea iterativă a operației XOR asupra caracterelor din șir. Practic se pornește cu valoarea $h_val = 0$, care pentru fiecare caracter $sir[i]$ din șir se modifică prin instrucțiunea

$$h_val \leftarrow h_val \text{ XOR } sir[i]$$

Nici această metodă nu dă rezultate bune în ceea ce privește coliziunile, dar operația XOR poate fi utilizată în cazul unor funcții de repartizare mai sofisticate.

Funcții polinomiale de dispersie

Un tip funcție de dispersie frecvent utilizată pentru a asocia unui *string* un număr natural este un polinom de grad $L - 1$ unde L este lungimea șirului considerat:

$$h(sir) = sir[0] + sir[1] * p + sir[2] * p^2 + \dots + sir[L - 1] * p^{L-1} = \sum_{i=0}^{L-1} sir[i] * p^i$$

În cazul unor șiruri de caractere lungi există riscul de a obține valori extrem de mari, care depășesc maximul admis pentru valorile numerice. Din acest motiv, adesea valoarea obținută se consideră modulo M , unde M este suficient de mare, ca să poată permite obținerea tuturor numerelor naturale din intervalul posibil ($[0, \text{MAX_INT}]$). Formula devine astfel:

$$h(sir) = \left(\sum_{i=0}^{L-1} sir[i] * p^i \right) \bmod M$$

În [12] se folosesc metoda lui Horner și regulile pentru restul împărțirii la un număr natural pentru a calcula iterativ formula, astfel

$$h_n = 0, h_i = (h_{i+1} * p + sir[i]) \bmod M, i = n - 1, \dots, 0$$

Pentru a avea o probabilitate cât mai redusă de coliziuni, este esențială alegerea parametrilor p și M . În [12] este discutat modul de alegere al acestora. În diferite exemple practice se utilizează pentru p un număr prim, de exemplu 31 sau 33 (Bernstein hash), iar pentru M poate fi considerat un număr prim mai mic decât 2^{64} , o valoare întâlnită în literatura online fiind $10^9 + 7$.

Algoritmul FNV

Unul dintre cei mai cunoscuți algoritmi de asociere a unei valori naturale pentru un *string*, algoritmul FNV (Fowler-Noll-Vo) [7], utilizează o metodă similară cu cea a polinoamelor, doar că în loc de adunare între puteri se folosește operația de XOR. Algoritmul general prezentat în continuare utilizează două constante *FNV_Prime* și *FNV_Basis* care depind de numărul de biți pe care este reprezentată valoarea de dispersie rezultată. În [7] este explicat modul de alegere al acestor constante. Pentru valori pe 32 de biți se consideră:

$$FNV_Prime = 16777619, FNV_Basis = 2166136261$$

iar pentru 64 de biți se consideră:

$$FNV_Prime = 1099511628211, FNV_Basis = 14695981039346656037$$

Algorithm: FNV_HASH

Input: O cheie *sir*

start

| $hash \leftarrow FNV_Basis$

| **pentru** \forall byte $b \in sir$ **executa**

| | $hash \leftarrow hash \times FNV_prime$

| | $hash \leftarrow hash \text{ XOR } b$

| **sfarsit_for**

| returneaza($hash$)

stop

Algorithm: FNV-1A_HASH

Input: O cheie *string*

start

| $hash \leftarrow FNV_Basis$

| **pentru** \forall byte $b \in sir$ **executa**

| | $hash \leftarrow hash \text{ XOR } b$

| | $hash \leftarrow hash * FNV_prime$

| **sfarsit_for**

| returneaza($hash$)

stop

Algoritmul FNV-1a este o alternativă a algoritmului FNV, care este utilizată de exemplu de Visual Studio pentru hashing.

Observație Evident, în cazul tuturor acestor funcții care calculează o valoare naturală asociată unui **string**, pentru a insera într-o tabelă de dispersie valoarea obținută, trebuie determinată poziția corespunzătoare printr-una dintre metodele prezentate în unitatea precedentă, de exemplu folosind metoda diviziunii sau metoda multiply-shift.

Tabele de dispersie implementate în STL

Implementare GCC

Modul de implementare al bibliotecilor $C++$ depinde de compilator. În general din sursele online, rezultă că tabelele de dispersie, folosite de `std::unordered_set`, respectiv `std::unordered_map` au în implementare la bază liste înlănțuite, de fapt o singură listă, în care elementele din fiecare *bucket* sunt grupate în "subliste". Ordinea în care aceste "subliste" apar în tabelă nu este

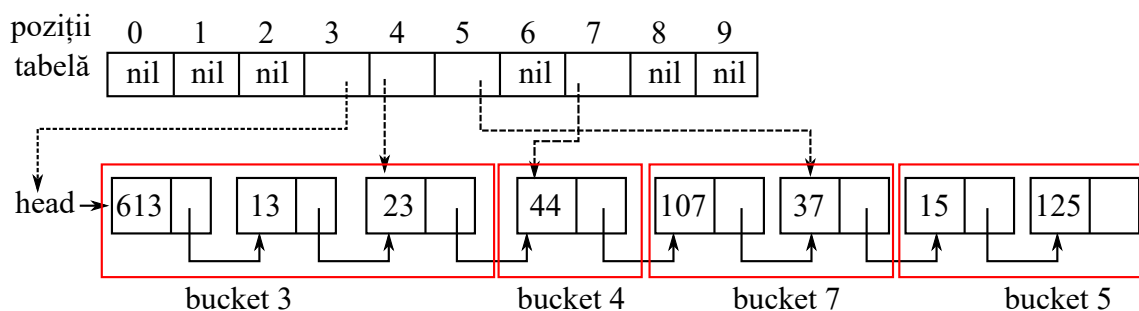


Figura 2.4: Reprezentarea unei table de dispersie de dimensiune 10, în care au fost inserate cheile 23, 44, 37, 107, 125, 15, 613. Dat fiind ca la bază se află un `std::forward_list`, inserția într-un **bucket** se face la începutul acestuia.

relevantă.

Tabela este reprezentată de un *array*, care conține pe fiecare poziție un iterator (pointer) către elementul dinaintea primului element (atunci când se folosește o listă simplu înlănțuită - de ex. GCC) sau către primul element (în cazul unei liste dublu înlănțuite - implementarea Microsoft) din *bucket-ul* corespunzător cheilor repartizate pe poziția respectivă. Implementarea printr-o singură listă permite o iterare rapidă prin elementele tablei de dispersie.

Modul de implementare din GCC este ilustrat în figura 2.4

Inițial, la declarare se pornește cu o tabelă ce dispune de 8 **bucket-uri**. Atunci când factorul de încărcare depășește valoarea maximă admisă, care în mod standard este $\alpha = 1.0$, are loc operația de realocare a memoriei și *reshasing*.

Ordinea în care sunt plasate în listă bucket-urile, precum și ordinea în care se află elementele unui bucket depinde de mai mulți factori, de exemplu de ordinea de inserare a elementelor sau de *reshasing*.

Implementare Visual Studio (C++17)

În Visual Studio (C++17) implementarea folosește o listă dublu înlănțuită, iar algoritmul de hasing este următorul.

Oricărei cheie de tip primitiv (int, char, float, double, string) i se asociază o valoare naturală folosind algoritmul de *hash* FNV-1a [7, 9].

La fel ca în cazul implementării GCC, se consideră inițial $nr_buckets = 8$ și atunci când factorul de încărcare devine mai mare ca 1, se dublează numărul de bucket-uri curente.

După calcularea valorii de *hash* cu FVN-1a selectarea bucket-ului pentru inserție se realizează în modul următor:

- se consider $(num_buckets - 1)$, deoarece $num_buckets = 2^l$ și atunci $(num_buckets - 1)$

are toți biții egali cu 1

- se calculează apoi $hash(k) \& (num_buckets - 1)$ ca indice al bucket-ului în care se va plasa k , unde $\&$ semnifică operația de 'și' pe biți. Cu alte cuvinte, din valoarea $hash$ rezultată după aplicarea FVN-1a, se păstrează doar cei mai semnificativi l biți.

Deși alegerea bucket-ului folosind metoda de mai sus (selectarea ultimilor biți ai valorii de $hash$) nu este foarte bună, din punct de vedere al coliziunilor, comparativ de exemplu cu metoda *multiply-shift*, viteza de calcul este foarte mare. Se consideră că FNV-1a produce o dispersie suficient de bună, ca să atenueze calitatea mai scăzută a modului de alegere a poziției de inserție. Modul de implementare al repartizării în Visual Studio poate fi găsit prin studierea codului sursă.



Să ne reamintim...

- În cazul tabelelor cu adresare deschisă rezolvarea coliziunii între 2 chei se realizează prin testarea pozițiilor.
- Metoda cu dublă dispersie se apropie cel mai mult de o repartizare uniformă în tabelă.
- În contextul memoriei ierarhizate, funcțiile de dispersie cu testare liniară pot constitui un avantaj.
- Dacă factorul de încărcare depășește 0.7, eficiența tabelelor scade semnificativ.

2.2.6 Rezumat

În această secțiune au fost discutate tabelele de dispersie cu adresare deschisă. Au fost prezentate noțiunile de bază, modul de determinare a poziției unei chei într-o tabelă, rezolvarea coliziunilor și câțiva algoritmi în pseudocod pentru implementarea acestor structuri împreună cu complexitatea acestora. Au fost discutate mai multe funcții de dispersie pentru chei de tip vectorial, respectiv string. De asemenea au fost prezentate structurile corespunzătoare din biblioteca STL, respectiv *unordered_set*, *unordered_map*.





2.2.7 Test de autoevaluare

1. Se consideră o tabelă de dispersie cu adresare deschisă, de dimensiune $m = 11$, și funcția de dispersie cu testare liniară $h_1(k) = k \bmod m$. Să se determine structura finală a tabelii după inserarea cheilor: 18, 22, 4, 7, 33, 21, 37.
2. Să se determine structura finală a tabelii dacă valorile 18, 22, 4, 7, 33, 21, 37 sunt inserate într-o tabelă de dispersie de dimensiune $m = 11$, utilizând metoda dublei repartizări cu funcțiile: $h_1(k) = k \bmod 11$, $h_2(k) = 7 - (k \bmod 7)$.
3. Să se explice modul în care factorul de încărcare α influențează timpul mediu și timpul din cel mai rău caz pentru operațiile de căutare.
4. Să se scrie un algoritm în pseudocod pentru operația de *rehashing*, caz în care dimensiunea tabelii se dublează.
5. Să se analizeze impactul unei alegeri nepotrivite a funcției $h_2(k)$ asupra performanței de timp pentru operația de inserție într-o tabelă cu dublă repartizare.
6. Scrieți un algoritm pseudocod care primește ca input o tabelă de dispersie cu dublă repartizare și returnează factorul de încărcare.

2.2.8 Răspunsuri la testul de evaluare a cunoștințelor

1. Se consideră o tabelă de dispersie cu adresare deschisă, de dimensiune $m = 11$, și funcția de dispersie cu testare liniară $h_1(k) = k \bmod m$. Să se determine structura finală a tabelii după inserarea cheilor: 18, 22, 4, 7, 33, 21, 55.

Rezolvare: Valoarea funcției de dispersie se calculează în funcție de valoarea cheii pe care dorim să o inserăm k și a numărului t de teste efectuate până la momentul curent, după o expresie liniară în t , anume

$$h(k, t) = (h_1(k) + t) \bmod m$$

Pentru cheile din enunț se obține:

$$h(18,0) = (h_1(18)+0) \bmod 11 = (18 \bmod 11+0) \bmod 11 = 7 \bmod 11 = 7 \Rightarrow T[7] = 18$$

$$h(22,0) = (22 \bmod 11 + 0) \bmod 11 = 22 \bmod 11 = 0 \Rightarrow T[0] = 22$$

$$h(4,0) = (4 \bmod 11 + 0) \bmod 11 = 4 \bmod 11 = 0 \Rightarrow T[4] = 4$$

$$h(7,0) = (7 \bmod 11 + 0) \bmod 11 = 7 \bmod 11 = 7 \Rightarrow \text{Coliziune (} T[7] \text{ este ocupat)}$$

$$h(7,1) = (7 \bmod 11 + 1) \bmod 11 = (7 + 1) \bmod 11 = 8 \Rightarrow T[8] = 7$$

$$h(33,0) = (33 \bmod 11 + 0) \bmod 11 = 33 \bmod 11 = 0 \Rightarrow \text{Coliziune (} T[0] \text{ este ocupat)}$$

$$h(33,1) = (33 \bmod 11 + 1) \bmod 11 = (0 + 1) \bmod 11 = 1 \Rightarrow T[1] = 33$$

$$h(21,0) = (21 \bmod 11 + 0) \bmod 11 = 21 \bmod 11 = 10 \Rightarrow T[10] = 21$$

$$h(55,0) = (55 \bmod 11 + 0) \bmod 11 = 55 \bmod 11 = 0 \Rightarrow \text{Coliziune (} T[0] \text{ este ocupat)}$$

$$h(55,1) = (55 \bmod 11 + 1) \bmod 11 = (0 + 1) \bmod 11 = 1 \Rightarrow \text{Coliziune (} T[1] \text{ este ocupat)}$$

$$h(55,2) = (55 \bmod 11 + 2) \bmod 11 = (0 + 2) \bmod 11 = 2 \Rightarrow T[2] = 55$$

Reprezentarea vizuală a tabelului este ilustrată în figura:

Tabela inițială		18	22	4	7	33	21	55	
0	/	0	/	0	22	0	22	0	22
1	/	1	/	1	/	1	33	1	33
2	/	2	/	2	/	2	/	2	55
3	/	3	/	3	/	3	/	3	/
4	/	4	/	4	4	4	4	4	4
5	/	5	/	5	/	5	/	5	/
6	/	6	/	6	/	6	/	6	/
7	/	7	18	7	18	7	18	7	18
8	/	8	/	8	/	8	7	8	7
9	/	9	/	9	/	9	/	9	/
10	/	10	/	10	/	10	/	10	21

2. Să se determine structura finală a tabelului dacă valorile 18, 22, 4, 7, 33, 21, 55 sunt inserate într-o tabelă de dispersie de dimensiune $m = 11$, utilizând metoda dublei repartizări cu funcțiile auxiliare: $h_1(k) = k \bmod 11$, $h_2(k) = 7 - (k \bmod 7)$.

Rezolvare: Valoarea funcției de *hash* este definită de formula:

$$h(k, t) = (h_1(k) + t \cdot h_2(k)) \bmod m$$

Pentru cheile din enunț se obține:

$$h(18, 0) = (18 \bmod 11 + 0 \cdot (7 - (18 \bmod 7))) \bmod 11 = (7 + 0 \cdot 3) \bmod 11 = 7 \Rightarrow T[7] = 18$$

$$h(22, 0) = (22 \bmod 11 + 0) \bmod 11 = 0 \bmod 11 = 0 \Rightarrow T[0] = 22$$

$$h(4, 0) = (4 \bmod 11 + 0) \bmod 11 = 4 \bmod 11 = 4 \Rightarrow T[4] = 4$$

$$h(7, 0) = (7 \bmod 11 + 0) \bmod 11 = 7 \bmod 11 = 7 \Rightarrow \text{Coliziune (} T[7] \text{ este ocupat)}$$

$$h(7, 1) = (7 + 1 \cdot (7 - (7 \bmod 7))) \bmod 11 = (7 + 1 \cdot 7) \bmod 11 = 14 \bmod 11 = 3 \Rightarrow T[3] = 7$$

$$h(33, 0) = (33 \bmod 11 + 0) \bmod 11 = 0 \bmod 11 = 0 \Rightarrow \text{Coliziune (} T[0] \text{ este ocupat)}$$

$$h(33, 1) = (0 + 1 \cdot (7 - (33 \bmod 7))) \bmod 11 = (0 + 1 \cdot 2) \bmod 11 = 2 \Rightarrow T[2] = 33$$

$$h(21, 0) = (21 \bmod 11 + 0) \bmod 11 = 10 \Rightarrow T[10] = 21$$

$$h(55, 0) = (55 \bmod 11 + 0) \bmod 11 = 0 \bmod 11 = 0 \Rightarrow \text{Coliziune (} T[0] \text{ este ocupat)}$$

$$h(55, 1) = (0 + 1 \cdot (7 - (55 \bmod 7))) \bmod 11 = 6 \Rightarrow T[6] = 55$$

Reprezentarea vizuală a tabelii este ilustrată în figura:

Tabela		18		22		4		7		33		21		55	
inițială															
0	/	0	/	0	22	0	22	0	22	0	22	0	22	0	22
1	/	1	/	1	/	1	/	1	/	1	/	1	/	1	/
2	/	2	/	2	/	2	/	2	/	2	33	2	33	2	33
3	/	3	/	3	/	3	/	3	7	3	7	3	7	3	7
4	/	4	/	4	/	4	4	4	4	4	4	4	4	4	4
5	/	5	/	5	/	5	/	5	/	5	/	5	/	5	/
6	/	6	/	6	/	6	/	6	/	6	/	6	/	6	55
7	/	7	18	7	18	7	18	7	18	7	18	7	18	7	18
8	/	8	/	8	/	8	/	8	/	8	/	8	/	8	/
9	/	9	/	9	/	9	/	9	/	9	/	9	/	9	/
10	/	10	/	10	/	10	/	10	/	10	/	10	21	10	21

3. Să se explice modul în care factorul de încărcare α influențează timpul mediu și timpul din cel mai rău caz pentru operațiile de căutare.

Rezolvare: Factorul de încărcare este definit prin formula: $\alpha = \frac{n}{m}$, unde n este numărul de elemente stocate, iar m dimensiunea tabelii. Valoarea factorului de încărcare fiind astfel o valoare din intervalul $[0, 1]$. Dacă valoarea $\alpha = 1 \Rightarrow$ tabela este plină (toate pozițiile sunt ocupate), iar dacă $\alpha = 0 \Rightarrow$ tabela este liberă. Timpul mediu este oferit de formula $\approx \frac{1}{1-\alpha}$, astfel că timpul mediu crește direct proporțional cu factorul de încărcare, adică cu gradul de ocupare al tabelii. Când $\alpha \rightarrow 1$, numărul de verificări din cadrul operațiilor de căutare și ștergere crește rapid, iar în cel mai rău caz devine $O(m)$.

4. Să se scrie un algoritm în pseudocod pentru operația de *rehashing*, caz în care dimensiunea tabelii se dublează.

Rezolvare: Operația de *rehashing* presupune dublarea dimensiunii tabelii și reinserarea tuturor elementelor în noua structură, în care acestea vor fi distribuite pe poziții diferite datorită modificării parametrului m din formula funcției de hash. La final, referința către noua tabelă o înlocuiește pe cea a tabelii anterioare, aceasta devenind tabela principală.

Algoritm: REHASHING

Input: O tabelă de dispersie cu dublă repartizare T de dimensiune m

start

$dim_noua \leftarrow m * 2$

$T_{aux} \leftarrow m * 2$

 aloca memorie pentru noua tabelă T_{aux} de dimensiune dim_noua

pentru $i \leftarrow 0, m$ **executa**

daca $T[i] \neq nil$ **atunci**

$t \leftarrow 0$

$poz \leftarrow h(T[i], t)$

cat timp $T_{aux}[poz] = nil$ **executa**

$t \leftarrow t + 1$

$poz \leftarrow h(T[i], t)$

sfarsit_cat_timp

$T_{aux}[poz] \leftarrow T[i]$

sfarsit_daca

sfarsit_for

 eliberează memoria pentru T

$T \leftarrow T_{aux}$

stop

5. Să se analizeze impactul unei alegeri nepotrivite a funcției $h_2(k)$ asupra performanței de timp pentru operația de inserție într-o tabelă cu dublă repartizare.

Rezolvare: Dacă funcția $h_2(k)$ nu este aleasă corect (de exemplu, nu este coprămă cu dimensiunea tabelii sau produce valori foarte mici), atunci pașii de verificare vor fi repetați periodic, fără a acoperi toate pozițiile tabelii. Aceasta poate duce la:

- blocarea procesului de inserție (nu se găsește niciodată o poziție liberă, deși tabela nu este plină)
- creșterea semnificativă a timpului de inserare, până la $O(m)$ în cel mai rău caz

6. Scrieți un algoritm pseudocod care primește ca input o tabelă de dispersie cu dublă repartizare și returnează factorul de încărcare.

Rezolvare: Pentru determinarea factorului de încărcare, este necesar să se numere numărul de valori existente în tabelă și să se calculeze raportul dintre acest număr și dimensiunea totală a tabelii.

Algoritm: CALCUL_FACTOR_INCARCARE

Input: O tabelă de dispersie cu dublă repartizare T de dimensiune m

```
start
   $n \leftarrow 0$ 
  pentru  $i \leftarrow 0, m$  executa
    |   daca  $T[i] \neq nil$  atunci
    |   |  $n \leftarrow n + 1$ 
    |   sfarsit_daca
  sfarsit_for
   $\alpha \leftarrow n/m$ 
  returneaza( $\alpha$ )
stop
```

Capitolul 3

Arbori. Arbori binari

Introducere

Un capitol mare în cadrul structurilor de date îl reprezintă arborii rădăcină. Aceștia au o structură ierarhizată și sunt utili într-o serie de aplicații practice. De-a lungul acestui curs vor fi discutate diferite tipuri de arbori, pornind de la heap-urile binare, arborii binari de căutare și până la structuri mai complexe precum B-arborii sau arborii Quad. În acest modul vor fi introduse definiții și notații generale pentru arbori, precum și modurile de parcurgere a acestora.

Competențe

La sfârșitul acestui modul de învățare studenții vor înțelege:

- Ce este un arbore rădăcină în general și ce este un arbore binar în particular.
- Care sunt modurile de reprezentare și de parcurgere ale unui arbore binar.

3.1 Unitatea de învățare 1 - Arbori



3.1.1 Introducere

În informatică, arborii reprezintă o structură de date esențială, utilizată pentru organizarea ierarhică a informației. Inspirați din natură, arborii informatici reflectă o relație de tip părinte-copil între elemente, permițând modelarea clară și eficientă a datelor care necesită o astfel de structură logică. Deși denumirea sugerează o rădăcină în partea de jos și ramuri în sus, în reprezentarea informatică arborele are rădăcina sus, iar ramificațiile coboară spre nivelurile inferioare, oferind o imagine inversată față de cea biologică.

Un arbore este format din noduri legate între ele prin muchii, iar fiecare nod poate avea mai mulți copii, dar un singur părinte, cu excepția rădăcinii, care nu are niciunul. Această structură permite gestionarea eficientă a datelor, fie că este vorba de fișiere într-un sistem de operare, de elemente într-o interfață grafică, de expresii într-un compilator sau de decizii într-un algoritm de inteligență artificială.

Arborii se regăsesc în numeroase aplicații practice, datorită capacității lor de a facilita căutarea rapidă, inserarea controlată a informației și menținerea ordinii logice.



3.1.2 Competențe

La sfârșitul acestei unități de învățare studenții vor înțelege:

- Ce este un arbore rădăcină în general și ce este un arbore binar în particular.
- Care sunt elementele ce compun un arbore binar.
- Care sunt modurile de reprezentare ale unui arbore binar.
- Care sunt modurile de parcurgere ale unui arbore binar.



Durata medie de studiu individual

Parcurgerea de către studenți a acestei unități de învățare se face în 1 oră.

3.1.3 Definiții și notații

În informatică un arbore este o structură ierarhică de noduri interconectate, dintre care unul este numit rădăcină și se află pe cel mai înalt nivel ierarhic. Într-un arbore nodurile sunt conectate într-o relație de tip părinte - fiu. Fiecare nod, cu excepția rădăcinii, are un părinte. De asemenea, orice nod poate avea 0 sau mai mulți fii. Niciun nod nu împarte vreun fiu cu alt

nod. Din punct de vedere al grafurilor, un arbore este un graf conex fără cicluri.

Un arbore se reprezintă ierarhic pe niveluri:

- Pe nivelul 0 se reprezintă nodul rădăcină r .
- Pentru fiecare nod x aflat pe un nivel k , descendenții săi direcți (fiii) se află pe nivelul $k + 1$.

Un exemplu de arbore este prezentat în fig. 3.1.

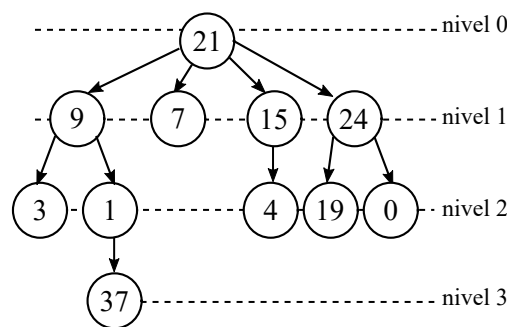


Figura 3.1: Arbore în reprezentarea ierarhică pe niveluri.

Definiții : Se consideră un arbore de rădăcină r și x, y noduri din arbore. Atunci:

- **părinte** al lui y este acel nod x pentru care (x, y) este muchie, iar x se află cu un nivel deasupra lui y ;
- **fiu** al lui x este un nod y pentru care (x, y) este muchie, iar y se află cu un nivel mai jos decât x ;
- **strămoș / ascendent** al lui x este orice nod y la care se poate ajunge de la x urcând de la părinte la părinte;
- **urmaș / descendent** al lui x este orice nod y la care se poate ajunge de la x printr-o succesiune de fii;
- **frați** sunt doi fii ai aceluiași nod;
- **frunză** este un nod fără fii;
- **nod intern** este un nod care nu este frunză.

Definiție - adâncimea / înălțimea unui arbore: Lungimea drumului de la rădăcina r la un nod x se numește adâncimea nodului x . Adâncimea rădăcinii este 0.

Lungimea celui mai lung drum de la x la o frunză se numește *înălțime* și se notează prin $h(x)$.

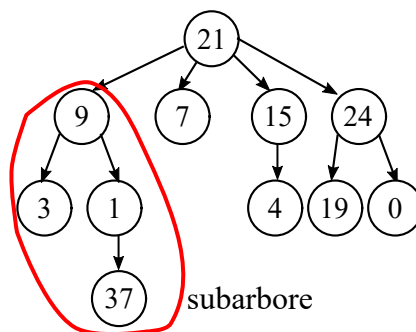


Exemplu: Considerând arborele din figura 3.1 adâncimea nodului 1 este 2 iar înălțimea este 1. Înălțimea unei frunze este 0. **Atenție:** există documentații în care înălțimea unei frunze este considerată egală cu 1.

Definiție - subarbore: Se numește *subarbore de rădăcină x* al unui arbore, arborele format din nodul x împreună cu toți descendenții săi.



Exemplu: în arborele din figură nodurile cuprinse în marcajul roșu formează un subarbore, care are drept rădăcină nodul cu cheia 9.

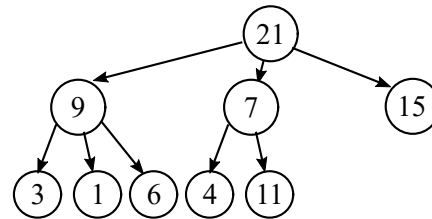


Arbori n -ari - definiții

- Se numește **arbore n -ar** un arbore pentru care fiecare nod are cel mult n fii.
- Un arbore n -ar se numește *arbore plin*, dacă fiecare nod intern are exact n fii.
- Un arbore n -ar se numește *arbore perfect*, dacă dacă fiecare nod intern are exact n fii și toate frunzele au aceeași adâncime.
- Un arbore n -ar se numește *complet*, dacă toate nodurile interne, cu excepția eventual a celor de pe penultimul nivel, au exact n fii, iar nodurile de pe ultimul nivel sunt așezate cel mai la stânga posibil pe nivelul respectiv.



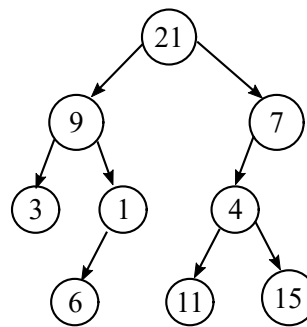
Exemplu: de arbore ternar complet.



Definiție - Arbore binar: Un arbore binar este un arbore în care fiecare nod are cel mult doi descendenți direcți. Atunci când fiecare nod are 0 sau 2 fi, arborele se numește *arbore binar strict*. În cazul unui arbore binar un nod are un *fiu stâng* și un *fiu drept*.



Exemplu: de arbore binar.



3.1.4 Reprezentarea arborilor

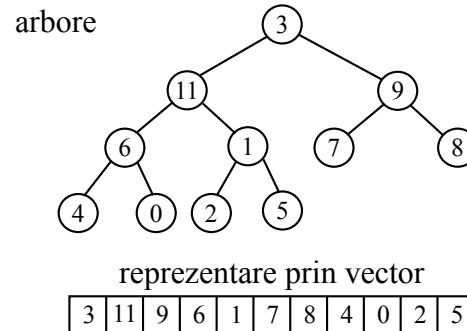
Pentru implementarea arborilor, pot fi utilizate diferite moduri de reprezentare. Cele mai frecvente dintre acestea sunt prezentate în continuare.

Reprezentare secvențială - se pretează în cazul arborilor compleți.

Un arbore n -ar complet de rădăcină r poate fi memorat într-un tablou unidimensional - *array* - în care pe poziția 0 se află rădăcina, pe următoarele n poziții se află fiii rădăcinii și în general pentru un nod oarecare aflat pe poziția i fiul al k -lea, $1 \leq k \leq n$ se află pe poziția $n * i + k$.



Exemplu: de arbore binar complet memorat într-un vector.

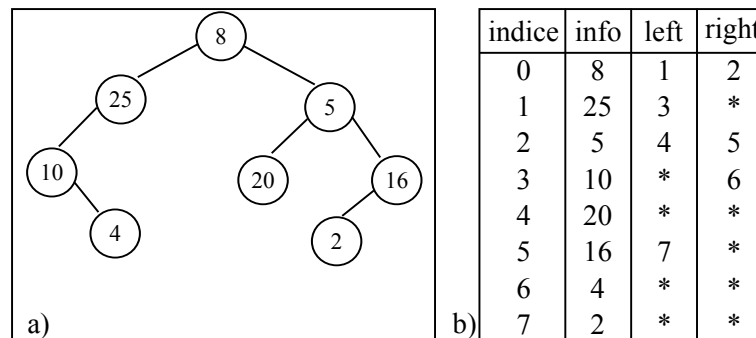


Reprezentare mixtă (pentru arbori binari)

Reprezentarea mixtă poate utiliza 3 vectori. Primul - INFO - conține informațiile din noduri, al doilea - *left* - are pe poziția i indicele fiului stâng al nodului i - în vectorul INFO -, iar vectorul al treilea - *right* - are pe poziția i indicele fiului drept al nodului i . O altă variantă mai comprimată ar fi aceea de utilizare a unui vector de noduri. Fiecare nod ar conține în acest caz 3 câmpuri: unul pentru informație, unul pentru indicele fiului stâng și unul pentru indicele fiului drept. Dacă este necesară și cunoașterea părintelui, atunci trebuie completată structura cu această informație.

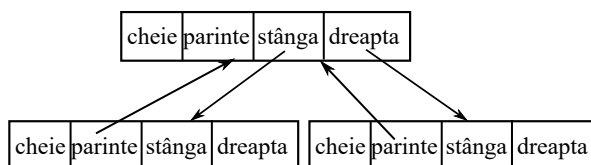


Exemplu: de arbore binar (a) împreună cu reprezentarea mixtă folosind trei vectori (b).



Reprezentare cu ajutorul unei structuri de noduri

- Pentru un arbore oarecare (nu binar): pentru fiecare nod se utilizează o structură în care se reține informația, legătura către o listă / vector de descendenți direcți și (eventual) legătura către nodul părinte.
- Pentru un arbore binar: fiecare nod este reprezentat printr-o structură în care se reține informația, legătura către fiul stâng, legătura către fiul drept și legătura către părinte.



- Pornind de la reprezentarea unui arbore binar, se poate utiliza pentru arbori oarecare o structură asemănătoare, în care se păstrează informația, o legătură - fiu - către cel mai din stânga fiu și o legătură - frate - către primul frate din dreapta al nodului, precum și o legătură către părinte. Un exemplu pentru acest mod de reprezentare este ilustrat grafic în figura 3.2.

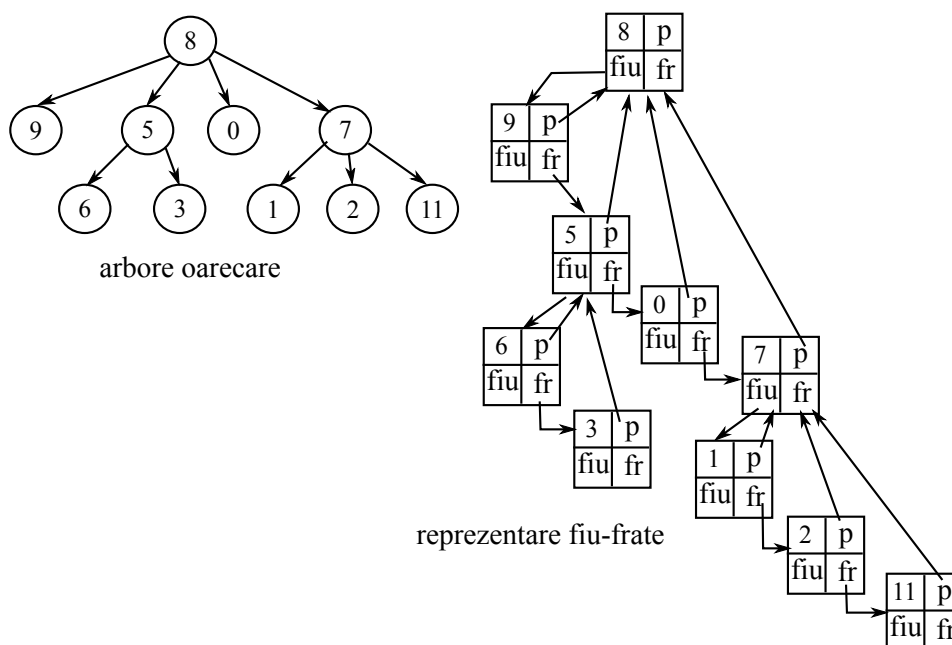


Figura 3.2: Arbore binar reprezentat printr-o structură de tip fiu-frate.

3.1.5 Parcurgerea arborilor

Există două moduri generale de parcurgere a arborilor oarecare:

- **Parcurgerea în lățime:** presupune vizitarea nodurilor arborelui de la stânga la dreapta, pornind de la nivelul 0 și parcurgând pe rând fiecare nivel. Se poate realiza practic utilizând o coadă C .
- **Parcurgerea în adâncime:** presupune vizitarea fiilor unui nod de la stânga spre dreapta, însă trecerea la fratele din dreapta se face abia după ce au fost vizitați toți descendenții

nodului curent, adică întregul său subarbore. Această parcurgere poate fi implementată cu ajutorul unei stive S .



Exemplu: în cazul arborelui din figura 3.2, parcurgerea în lățime va avea ca rezultat afișarea cheilor în ordinea: 8, 9, 5, 0, 7, 6, 3, 1, 2, 11. Parcurgerea în adâncime va avea ca rezultat afișarea cheilor în ordinea: 8, 9, 5, 6, 3, 0, 7, 1, 2, 11.

3.1.6 Parcurgerea arborilor binari

Parcurgerea în lățime a unui arbore binar presupune parcurgerea nodurilor pornind de la rădăcină și parcurgând apoi fiecare nivel de la stânga la dreapta. Acest mod de parcurgere este ilustrată grafic în figura 3.3.

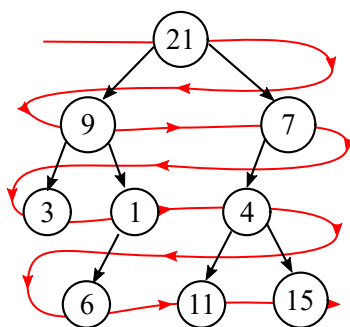


Figura 3.3: Parcurgere pe niveluri.

Algoritmul de parcurgere presupune utilizarea unei cozi C , în care se stochează nodurile, ai căror fii mai trebuie prelucrați.

Algoritm: PARCURGERE-LATIME**Input:** Arborele binar T **start** $C \leftarrow \emptyset$ $ADAUGA(C, T.rad)$ //Pune rădăcina în coada C **cat timp** $C \neq \emptyset$ **executa** $curent \leftarrow PRIM(C)$ $EXTRAGE(C)$ processeaza($curent$)**daca** $curent.stanga \neq nil$ **atunci**| $ADAUGA(C, curent.stanga)$ **sfarsit_daca****daca** $curent.dreapta \neq nil$ **atunci**| $ADAUGA(C, curent.dreapta)$ **sfarsit_daca****sfarsit_cat_timp****stop**

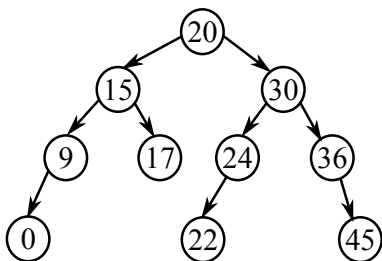
Complexitatea acestui algoritm este $O(n)$, n este numărul de noduri din arbore.

Parcurgerea în adâncime a unui arbore binar presupune parcurgerea fiecărui nod împreună cu subarborele stâng și subarborele drept. Subarborele stâng se ia în considerare înaintea subarborelui drept. În funcție de ordinea în care parcurg rădăcina și subarborii, se disting trei tipuri de parcurgere:

- **Preordine (RSD):** rădăcină, subarbore stâng, subarbore drept
- **Inordine (SRD):** subarbore stâng, rădăcină, subarbore drept
- **Postordine (SDR):** subarbore stâng, subarbore drept, rădăcină



Exemplu: Se consideră arborele binar din figură.



RSD: 20, 15, 9, 0, 17, 30, 24, 22, 36, 45.

SRD: 0, 9, 15, 17, 20, 22, 24, 30, 36, 45.

SDR: 0, 9, 17, 15, 22, 24, 45, 36, 30, 20.

În practică parcurgerea poate fi făcută iterativ utilizând o stivă, sau recursiv. Algoritmii recursivi sunt mai concisi și mai simpli de implementat dar presupun un consum mai mare de memorie.



To Do Dați exemplu de un arbore binar de înălțime $h = 3$ cu 15 noduri. Scrieți apoi cele trei parcurgeri în adâncime corespunzătoare exemplului dat.

Exemplu de algoritm secvențial - parcurgerea în preordine care utilizează o stivă S , în care se depun nodurile pentru care mai trebuie parcurși subarborii dreپți:

Algorithm: PREORDINE

Input: Arborele binar T

start

$S \leftarrow \emptyset$

$curent \leftarrow T.rad$

repetă

cat timp $curent \neq nil$ **execută**

 proceseaza($curent.cheie$)

$ADAUGA(S, curent)$ //pune $curent$ în stiva S

$curent \leftarrow curent.stanga$

sfarsit_cat_timp

daca $S \neq \emptyset$ **atunci**

$curent \leftarrow VARF(S)$

$curent \leftarrow curent.dreapta$

sfarsit_daca

pana_cand $S = \emptyset$;

stop

Algoritmi recursivi

Algorithm: PREORDINE

Input: Un pointer către nodul curent nod

start

daca $nod \neq nil$ **atunci**

 proceseaza(nod)

 PREORDINE($nod.stanga$)

 PREORDINE($nod.dreapta$)

sfarsit_daca

stop

Algoritm: INORDINE

Input: Un pointer către nodul curent *nod***start**

```

daca nod ≠ nil atunci
  | INORDINE(nod.stanga)
  | proceseaza(nod)
  | INORDINE(nod.dreapta)
sfarsit_daca

```

stop

Algoritm: POSTORDINE

Input: Un pointer către nodul curent *nod***start**

```

daca nod ≠ nil atunci
  | POSTORDINE(nod.stanga)
  | POSTORDINE(nod.dreapta)
  | proceseaza(nod)
sfarsit_daca

```

stop

**To Do** Implementați o funcție iterativă *PREORDINE* pe baza algoritmului prezentat anterior.**To Do** Implementați cele trei funcții recursive de parcurgere a un arbore binar în adâncime.

Aplicații practice ale arborilor

Arborii joacă un rol esențial în informatică, având numeroase aplicații practice în organizarea și manipularea datelor. De exemplu, arborii **binari de căutare** sunt utilizați pentru stocarea eficientă a informației, permițând operații rapide de căutare, inserare și ștergere. **Arborii de decizie** sunt folosiți în inteligența artificială și în învățarea automată pentru luarea deciziilor pe baza unor condiții logice. **Heap-urile** sunt folosite pentru implementarea cozilor de prioritate. În sistemele de fișiere, structura de tip arbore este folosită pentru organizarea ierarhică a directoarelor și fișierelor. În codificare și compresie se folosește **arborele Huffmann**, iar în grafică și procesare de imagine își găsesc aplicabilitatea **arbori Quad sau arbori PR**.



Să ne reamintim...

- În informatică un arbore este o structură ierarhică de noduri, conectate între ele, dintre care unul este numit rădăcină și se află pe cel mai înalt nivel ierarhic.
- Într-un arbore nodurile sunt interconectate într-o relație ierarhică de tip părinte - fiu.
- Un arbore n -ar este complet dacă toate nodurile interne, cu excepția eventual a celor de pe penultimul nivel, au exact n fii, iar nodurile de pe ultimul nivel sunt așezate cel mai la stânga posibil pe nivelul respectiv.
- Un arbore binar este un arbore în care fiecare nod are cel mult 2 fii.
- Arborii binari pot fi parcurși în lățime sau în adâncime. Parcurgerile în adâncime pot fi în preordine, inordine sau postordine.

3.1.7 Rezumat



În această secțiune au fost discutați arborii în general, precum și arborii binari în particular, împreună cu elementele constitutive, modurile de implementare și parcurgerile acestora.



3.1.8 Test de autoevaluare

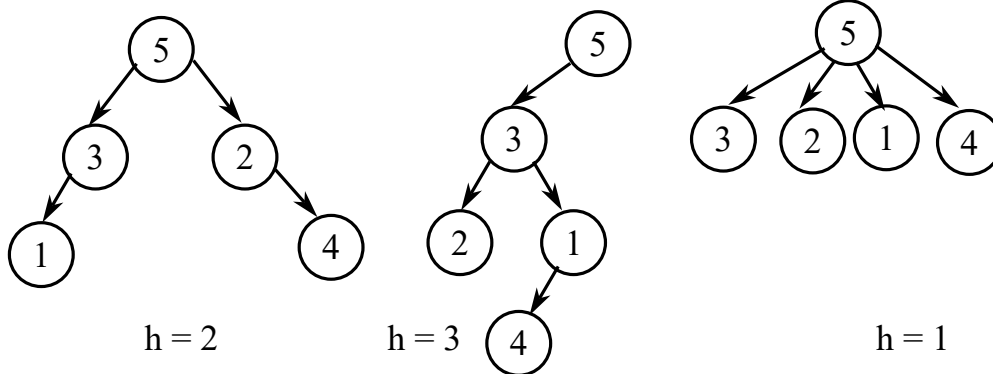
1. Se consideră secvența de noduri 5, 3, 2, 1, 4 obținută prin parcurgerea pe niveluri a unui arbore oarecare. Să se deseneze 3 arbori posibili corespunzători și să se precizeze înălțimea fiecăruia.
2. Care este înălțimea minimă și care este înălțimea maximă a unui arbore binar cu n noduri?
3. Care este numărul minim și numărul maxim de chei ale unui arbore binar cu înălțimea $h = 5$?
4. Cunoscând parcurgerea în preordine 10, 6, 3, 12, 11, 7, 4, 8, 1, 13 și parcurgerea în inordine 3, 6, 11, 12, 7, 10, 8, 4, 13, 1, să se reconstruiască arborele binar.
5. Să se scrie un algoritm în pseudocod pentru determinarea înălțimii unui arbore binar.

6. Să se scrie un algoritm în pseudocod care returnează numărul de frunze dintr-un arbore oarecare, reținut în memorie utilizând repartizarea fiu-frate.

3.1.9 Răspunsuri la testul de evaluare a cunoștințelor

1. Se consideră secvența de noduri 5, 3, 2, 1, 4 obținută prin parcurgerea pe nivele a unui arbore oarecare. Să se deseneze 3 arbori posibili corespunzători și să se precizeze înălțimea fiecăruia.

Rezolvare:



2. Care este înălțimea minimă și care este înălțimea maximă a unui arbore binar cu n noduri?

Rezolvare: Înălțimea maximă se obține pentru număr minim de noduri per nivel și este $n - 1$ și înălțimea minimă se obține pentru număr maxim de noduri per nivel și este $\log_2 n$.

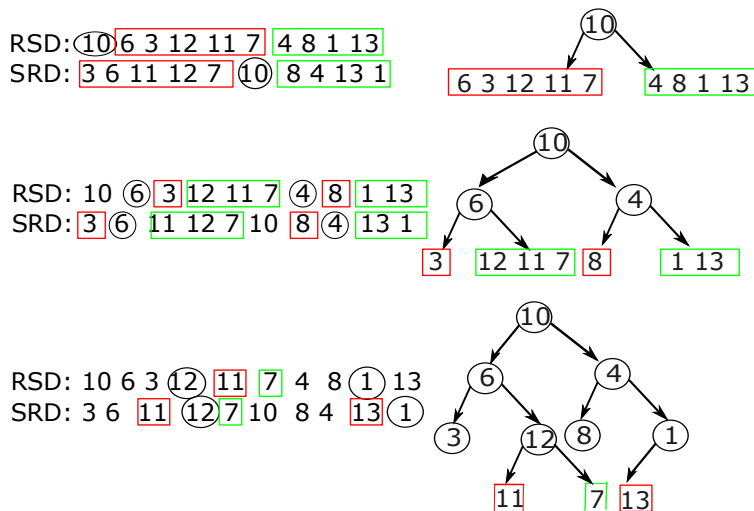
3. Care este numărul minim și numărul maxim de chei ale unui arbore binar cu înălțimea $h = 5$?

Rezolvare: Numărul minim de chei se obține atunci când pe fiecare nivel avem număr minim de noduri, adică unul singur. În acest caz numărul de noduri este $h + 1 \rightarrow 5 + 1 = 6$. Numărul maxim de chei se obține atunci când pe fiecare nivel k avem număr maxim de noduri, adică 2^k noduri. În acest caz numărul de noduri este

$$\begin{aligned}
 &1 + 2 + 2^2 + \dots + 2^h = \\
 &= 2^{h+1} - 1 \rightarrow 2^6 - 1 = 63
 \end{aligned}$$

4. Cunoscând parcurgerea în preordine 10, 6, 3, 12, 11, 7, 4, 8, 1, 13 și parcurgerea în inordine 3, 6, 11, 12, 7, 10, 8, 4, 13, 1, să se reconstruiască arborele binar.

Rezolvare: Rădăcina se află în preordine pe primul loc, deci nodul cu valoare 10 este rădăcina arborelui. Apoi căutăm în inordine nodul rădăcină, iar tot ce se află înaintea acesteia sunt cheile din subarborele stâng, iar ce se află după aceasta sunt nodurile din subarborele drept. Acest procedeu se repetă pentru fiecare subarbore și este reprezentat în figura:



5. Să se scrie un algoritm în pseudocod pentru determinarea înălțimii unui arbore binar.

Algoritm: ÎNĂLȚIME

Input: Rădăcina unui arbore (subarbore) binar reținută în variabila *nod*

Output: Înălțimea arborelui

start

daca *nod* = *nil* **atunci**

 | returneaza 0

sfarsit_daca

h_st ← *Inaltime*(*nod.stanga*)

h_dr ← *Inaltime*(*nod.dreapta*)

 returneaza(1 + max(*h_st*, *h_dr*))

stop

6. Să se scrie un algoritm în pseudocod care returnează numărul de frunze dintr-un arbore oarecare, reținut în memorie utilizând repartizarea fiu-frate.

Algorithm: NUMĂRAREA_FRUNZELOR_DIN_ARBORE

Input: Un arbore A reținut în memorie utilizând reprezentarea fiu-frate**Output:** Numărul de frunze din arbore**start**

```
daca  $A.radacina = nil$  atunci
| returneaza(0)
sfarsit_daca
 $C \leftarrow \emptyset$ 
 $nr\_frunze \leftarrow 0$ 
 $ADAUGA(C, A.radacina)$ 
cat_timp  $C \neq \emptyset$  executa
|  $nod \leftarrow EXTRAGE(C)$ 
|   daca  $nod.fiu = nil$  atunci
|   |  $nr\_frunze \leftarrow nr\_frunze + 1$ 
|   sfarsit_daca
|   daca  $nod.fiu \neq nil$  atunci
|   |  $ADAUGA(C, nod.fiu)$ 
|   sfarsit_daca
|   daca  $nod.frate \neq nil$  atunci
|   |  $ADAUGA(C, nod.frate)$ 
|   sfarsit_daca
sfarsit_cat_timp
returneaza( $nr\_frunze$ )
```

stop

Capitolul 4

Heap. Coadă de priorități

Introducere

Heap-urile reprezintă o clasă specializată de arbori binari, caracterizată printr-o structură completă (toate nivelurile sunt complet populate, cu excepția posibil, a ultimului, care este completat de la stânga la dreapta) și printr-o relație de ordonare între noduri, determinată de tipul heap-ului: max-heap sau min-heap. Unul dintre principalele avantaje ale acestei structuri constă în posibilitatea implementării eficiente utilizând un vector, ceea ce permite accesarea directă a elementelor fără utilizarea explicită a legăturilor între noduri. De asemenea, proprietatea de ordonare a cheilor, în funcție de priorități, face ca heap-urile să fie structuri ideale pentru implementarea eficientă a cozilor de prioritate, permițând operații rapide de inserare și extragere a elementului cu cea mai mare (sau cea mai mică) prioritate.

Competențe

La sfârșitul acestui modul de învățare studenții:

- Știu să definească heap-urile max, respectiv min.
- Știu să implementeze eficient un heap folosind un vector.
- Definesc o coadă de priorități cu operațiile principale.
- Cunosc complexitatea operațiilor de bază.
- Utilizează cozile de prioritate pentru rezolvarea problemelor concrete.

4.1 Unitatea de învățare 1 - Heap



4.1.1 Introducere

În această unitate sunt prezentate heap-urile binar min și max, proprietățile acestora și modul de construcție. De asemenea este prezentat ca exemplu de aplicabilitate algoritmul de sortare heap-sort.



4.1.2 Obiective

La sfârșitul acestei unități de învățare studenții vor înțelege:

- Ce este un heap-max, respectiv un heap-min.
- Cum se implementează heap-urile folosind vectori.
- Cum se construiește un heap pe baza unui vector oarecare.
- Cum funcționează algoritmul heap-sort.



Durata medie de studiu individual

Parcursul de către studenți a acestei unități de învățare se face în 2 ore.

Definiție. Un heap binar este un arbore binar complet - fiecare nod intern are exact doi fii, cu excepția eventual a ultimului nod intern de pe penultimul nivel, iar frunzele se află doar pe ultimele două niveluri, frunzele de pe ultimul nivel sunt așezate de la stânga spre dreapta - memorat cu ajutorul unui tablou unidimensional (vector). În plus, în heap-ul binar există o ordonare a cheilor, determinată de tipul heap-ului.

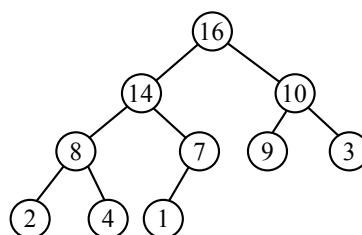
1. **Max-heap:** informația din fiecare nod este mai mare decât informația din oricare descendent al său. Maximul din heap se află în rădăcină.



Exemplu de heap-max.

H:

16	14	10	8	7	9	3	2	4	1
----	----	----	---	---	---	---	---	---	---



2. **Min-heap**: informația din fiecare nod este mai mică decât informația din oricare descendent al său. Minimumul din heap se află în rădăcină.

Pentru implementare se poate utiliza structura heap H , care folosește drept container un vector, în care *dimensiune* este numărul de chei din heap. Rădăcina heap-ului se află pe prima poziție a heap-ului $H[0]$. Pentru fiecare nod aflat pe poziția i :

- Părintele se află pe poziția $(i - 1)/2$.
- Fiul stâng se află pe poziția $2 * i + 1$.
- Fiul drept se află pe poziția $2 * i + 2$.

Observații:

- Atunci când indicii în vector se consideră începând de la 1, se schimbă și relațiile poziționale între părinte și fiu.
- Înălțimea arborelui care reprezintă heap-ul este $\log_2 n$ unde $n = H.dimensiune$ este numărul de noduri din heap.
- Complexitatea operațiilor de bază este proporțională cu înălțimea arborelui, adică $O(\log_2 n)$.

Construcția unui heap - max

Considerând un vector de elemente, pentru transformarea acestuia într-un heap-max sunt necesare două etape, reprezentate prin 2 funcții.

(1) **SIFT-DOWN**(H, i): în care H heap cu $H.dimensiune$ elemente și i poziția în vector a nodului de la care începe algoritmul. Premiza este aceea că subarborii nodului de pe poziția i sunt heap-max și doar informația din nodul i strică eventual această proprietate. Funcția SIFT-DOWN reface proprietatea de heap-max prin coborârea cheii de pe poziția i , pe poziția potrivită în heap.

Algorithm: SIFT-DOWN

Input: Un heap binar max H cu *dimensiune* elemente, indicele i al elementului curent.

start

$stanga \leftarrow 2 * i + 1$ // indicele fiului stâng

$dreapta \leftarrow 2 * i + 2$ // indicele fiului drept

$imax = i$ // indicele valorii maxime

daca $stanga < H.dimensiune$ și $H[stanga] > H[imax]$ **atunci**

 | $imax = stanga$

sfarsit_daca

daca $dreapta < H.dimensiune$ și $H[dreapta] > H[imax]$ **atunci**

 | $imax = dreapta$

sfarsit_daca

daca $imax \neq i$ **atunci**

 | Interschimba($H[i], H[imax]$)

 | SIFT-DOWN($H, imax$)

sfarsit_daca

stop

Complexitate: Algoritmul pornește de la nodul aflat pe poziția i și ajunge în cel mai defavorabil caz la o frunză, deci complexitatea depinde de înălțimea arborelui care este $h = \log_2 n$. Deci $T(n) = O(\log_2 n)$.

Se observă faptul că subarborii reprezentați de frunze sunt heap-max. Atunci e suficient să pornim de la primul nod intern considerat începând de la dreapta heap-ului H , adică primul care are cel puțin un fiu. Nodul respectiv se află pe poziția $H.dimensiune/2 - 1$, iar copiii săi sunt frunze, deci heap-max. Se aplică funcția de SIFT-DOWN pentru nodul $H.dimensiune/2 - 1$, după care se trece la nodul precedent din vector și așa mai departe, până la rădăcină.

(2) **CONSTRUIESTE_HEAP(H):** pe baza funcției SIFT-DOWN.

Algorithm: CONSTRUIESTE_HEAP

Input: Un heap binar max H cu *dimensiune* elemente

start

pentru $i = H.dimensiune/2 - 1, 0, -1$ **executa**

 | SIFT-DOWN(H, i)

sfarsit_for

stop

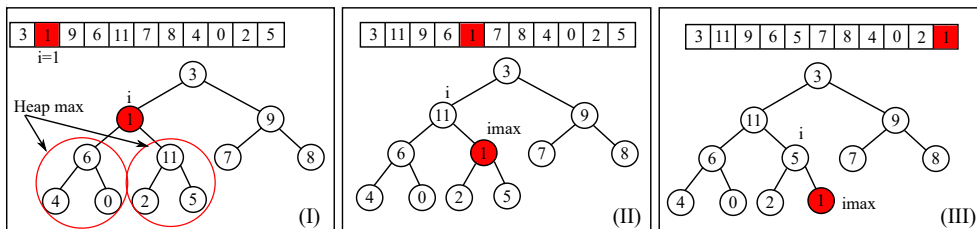
Complexitate: fiecare apel al funcției SIFT-DOWN are complexitatea $O(\log_2 n)$.

CONSTRUIESTE_HEAP efectuează $O(n)$ astfel de apeluri. Deci $T(n)$, unde prin T am no-

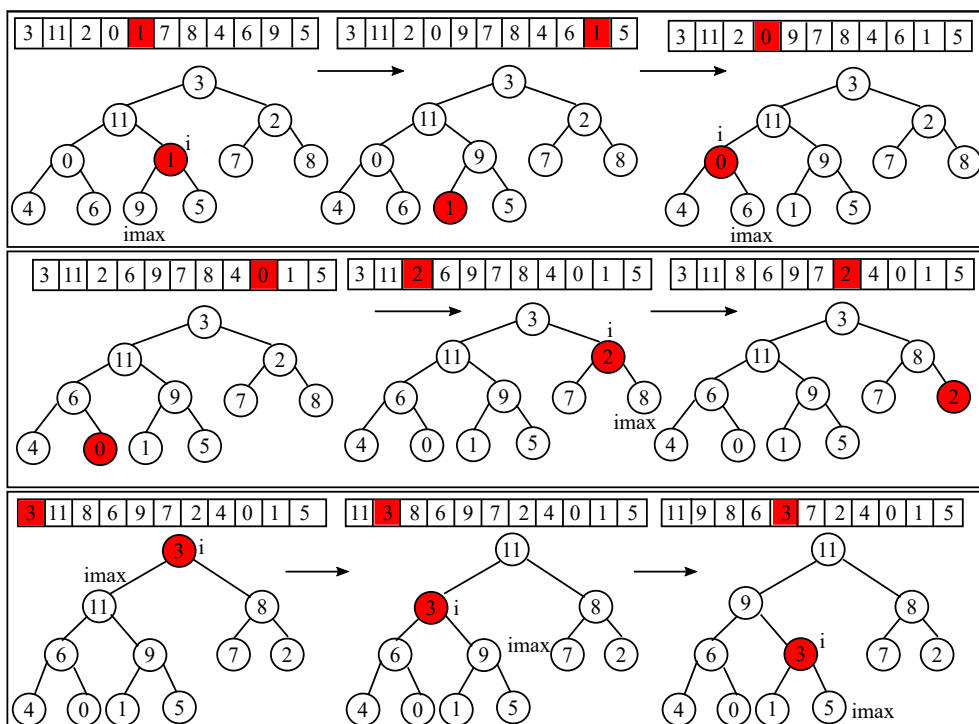
tat complexitatea, pentru funcția CONSTRUIESTE_HEAP, este mărginită superior de $n \log_2 n$. Se demonstrează în literatură [4] faptul că funcția are complexitate liniară, adică $T(n) = O(n)$, $n = H.dimensiune$.

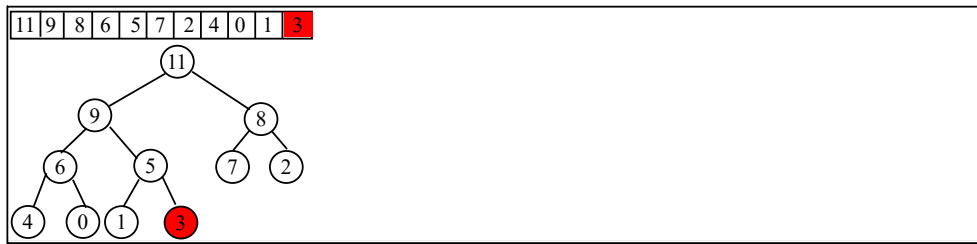


Exemplu al aplicării funcției SIFT-DOWN. Funcția SIFT-DOWN pornește în acest exemplu de la nodul i marcat cu roșu în figura (I). Cei doi subarbori ai săi sunt heap-max. Funcția are două apeluri recursive, ilustrate în etapele (II) și (III).



Exemplu al aplicării funcției CONSTRUIESTE_HEAP. În acest exemplu algoritmul pornește de la nodul de pe poziția a 5-a în heap-ul H .





Un min-heap se construiește similar cu max-heap, doar că se schimbă inegalitățile în funcția SIFT-DOWN.

4.1.3 Algoritmul de sortare *HeapSort*

Algoritmul HeapSort exploatează proprietatea heap-urilor de a menține în vârf elementul cu valoarea maximă, păstrând în același timp relația de ordonare între fiecare părinte și fiii săi.

Algoritm: HEAPSORT

Input: Un vector v cu n elemente, un heap binar max H .

start

 Copiază elementele lui v în heap-ul H

 CONSTRUIESTE_HEAP(H)

pentru $i = n - 1, 1, -1$ **executa**

$v[i] \leftarrow H[0]$

$H[0] \leftarrow H[H.dimensiune - 1]$

$H.dimensiune \leftarrow H.dimensiune - 1$

 SIFT-DOWN($H, 0$)

sfarsit_for

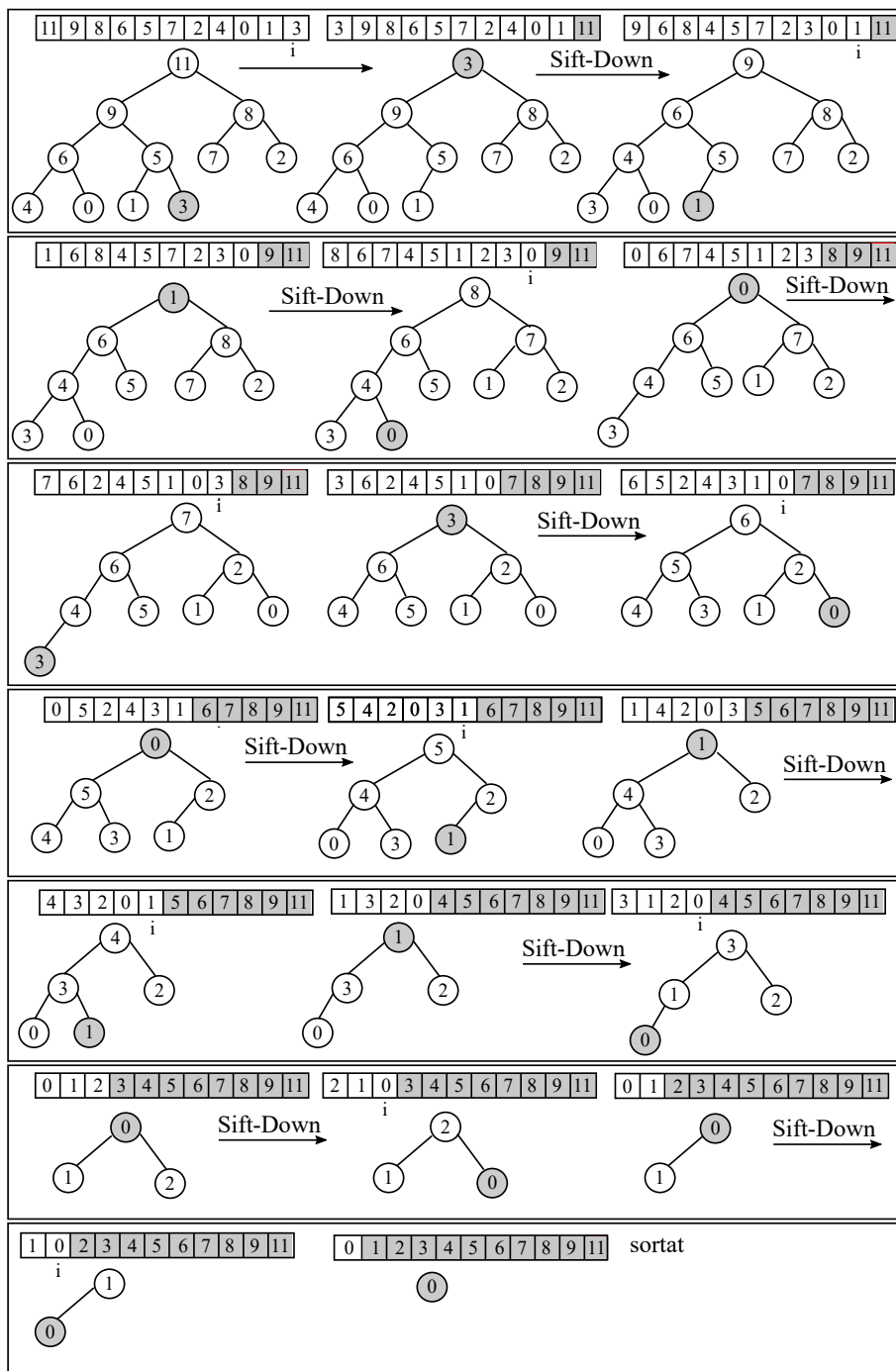
$v[0] \leftarrow H[0]$

stop

Cum heap-ul este memorat într-un vector, iar rădăcina se află pe prima poziție a acestuia, rezultă că valoarea maximă se găsește pe prima poziție a vectorului de date. Într-un vector sortat crescător, elementul maxim trebuie să fie pe ultima poziție. Pornind de la această observație, vectorul poate fi sortat astfel: se interschimbă primul element din heap cu ultimul element al vectorului; se reduce dimensiunea heap-ului și se reface proprietatea de heap-max prin aplicarea funcției SIFT-DOWN, începând din rădăcină. Procedura se repetă până când heap-ul se reduce la un singur element



Exemplu de aplicare al algoritmului HeapSort.



Complexitate: Apelul funcției CONSTRUIESTE_HEAP este $O(n)$, apelul SIFT-DOWN este $O(\log_2 n)$, iar funcția SIFT-DOWN se apelează de $n - 1$ ori. Deci $T(n) = O(n \log_2 n)$.



Implementați algoritmul HeapSort.



Să ne reamintim...

- Heap-urile binare sunt un caz particular de arbori binari compleți cu o ordonare a cheilor dată de tipul de heap (max sau min).
- Avantajul major al structurii de heap este acela, ca poate fi stocat eficient într-un vector.
- Construcția unui heap max sau min pe baza unui vector se poate realiza în complexitate liniară în numărul de elemente din vector.
- Algoritmul HeapSort are la bază construcția unui heap-max (sau min, cu ajustările corespunzătoare) și are complexitate $O(\log_2 n)$.



4.1.4 Rezumat

În această secțiune au fost discutate heap-urile binare max și min, modul de construcție, precum și algoritmul HeapSort.



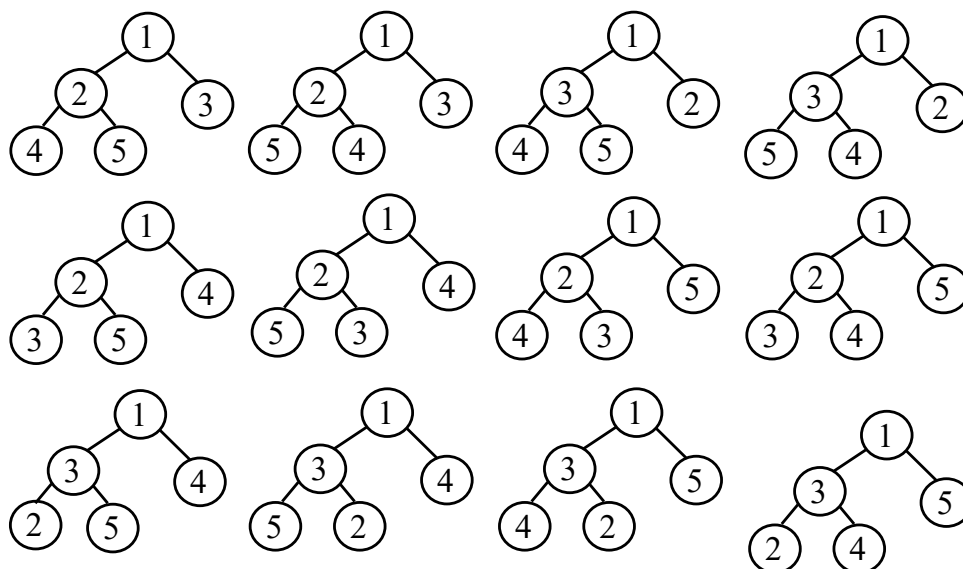
4.1.5 Test de autoevaluare

1. Desenați toți arborii heap-min care se pot forma cu valorile 5, 2, 1, 4, 3.
2. Construiți un max-heap din vectorul: $v = \{10, 6, 5, 1, 20, 82, 9, 4, 17\}$ pe baza algoritmului de construcție din curs. Cum arată vectorul rezultat corespunzător heap-ului min?
3. Care este complexitatea căutării unui element într-un heap?
4. Care este numărul minim și care numărul maxim de elemente ale unui heap de înălțime $h = 6$?
5. Scrieți un algoritm pseudocod pentru modificarea valorii maxime dintr-un max-heap cu o valoare dată ca parametru.
6. Scrieți un algoritm pseudocod pentru construirea unui heap-min prin concatenarea elementelor a două heap-uri min existente.

4.1.6 Răspunsuri la testul de evaluare a cunoștințelor

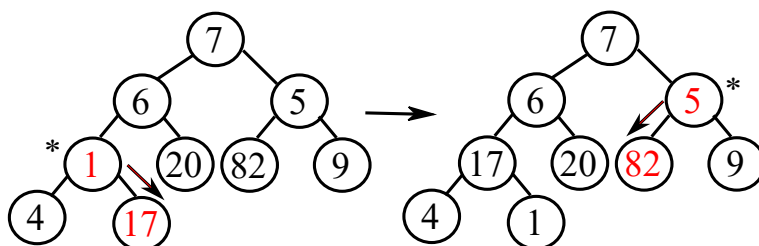
1. Desenați toți arborii heap-min care se pot forma cu valorile 5, 2, 1, 4, 3.

Rezolvare: Se vor analiza toate permutările mulțimii de elemente 1, 2, 3, 4, 5 care, introduse într-un heap-min, respectă proprietatea specifică acestei structuri: informația dintr-un nod părinte este mai mică decât informația din nodurile sale copil. De asemenea, nodurile sunt aranjate sub forma unui arbore binar complet, în conformitate cu definiția unui heap.

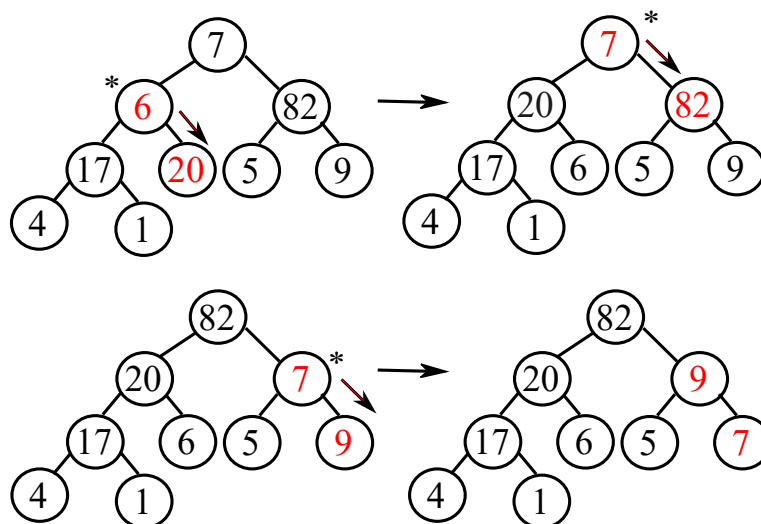


2. Construiți un max-heap din vectorul: $v = \{10, 6, 5, 1, 20, 82, 9, 4, 17\}$ pe baza algoritmului de construcție din curs. Cum arată vectorul rezultat corespunzător heap-ului min?

Rezolvare: Arborele corespunzător vectorului este reprezentat în figura de mai jos, în stânga. Se pornește algoritmul de cel mai din dreapta nod (din vector) care are fii, deci de la nodul de pe poziția $n/2 - 1$, unde n este numărul de elemente. În cazul nostru, acesta este nodul cu valoarea 1, marcat cu * în arbore. Se observă că ambii fii ai acestui nod are cheile mai mari, ce mai mare fiind în dreapta, deci de aplică SIFT-DOWN și cheia 1 coboară în locul cheii 17, rezultând arborele din dreapta.



Se continuă apoi de la nodul cu cheia 5. Se observă că ambii fii ai acestui nod au cheile mai mari, se aplică SIFT-DOWN, iar cheia 5 coboară în locul cheii 28, rezultând arborele din stânga din figura următoare. Același procedeu continuă cu nodurile 6 și 7, ilustrate în figurile următoare.



Vectorul rezultat, corespunzător heap-ului max este:

$$\{82, 20, 9, 17, 6, 5, 7, 4, 1\}$$

3. Care este complexitatea căutării unui element într-un heap?

Rezolvare: Într-un heap, proprietatea fundamentală este că fiecare nod respectă regula max-heap sau min-heap față de copii săi, dar elementele nu sunt sortate (vezi rezultatul exercițiului anterior). Prin urmare, pentru a căuta un element arbitrar, în cel mai rău caz trebuie parcurse toate elementele astfel complexitatea de timp este: $\mathcal{O}(n)$.

4. Care este numărul minim și numărul maxim de elemente ale unui heap de înălțime $h = 6$?

Rezolvare: Un heap de înălțime h :

- Numărul minim de elemente se obține atunci când toate nivelurile sunt pline, cu excepția ultimului nivel pe care se află un singur nod. Astfel avem:

$$\begin{aligned} 1 + 2 + 2^2 + \dots + 2^{h-1} + 1 &= \\ &= (2^{h-1+1} - 1) + 1 = 2^h = 2^6 = 64 \end{aligned}$$

deci numărul de noduri este 64.

- Numărul maxim de elemente se obține atunci când toate nivelurile sunt complet pline, astfel în heap sunt $2^{h+1} - 1 = 2^7 - 1 = 127$ noduri.
5. Scrieți un algoritm pseudo cod pentru modificarea valorii maxime dintr-un max-heap cu o valoare dată ca parametru.

Rezolvare:

Algorithm: MODIFICARE_MAX-HEAP

Input: Un max-heap H , valoare nouă val

start

$H[0] \leftarrow val$
 $SIFT - DOWN(H, 0)$

stop

6. Scrieți un algoritm pseudocod pentru construirea unui heap-min prin concatenarea elementelor a două heap-uri min existente.

Rezolvare:

Algorithm: CONCATENARE_HEAP-MIN

Input: Două heap-uri min H_1 și H_2

Output: Un heap-min H

start

$H \leftarrow H_1 + H_2$ - concatenarea celor 2 vectori
 $n \leftarrow H.dimensiune$
 pentru $i \leftarrow [n/2]$ **to** 1 **executa**
 | $SIFT - DOWN(H, i)$
 sfarsit_for
 returneaza H

stop

4.2 Unitatea de învățare 2 - Cozi de prioritate



4.2.1 Introducere

Una dintre cele mai importante aplicații ale heap-urilor sunt cozile de prioritate. O coadă de prioritate este o structură de date utilizată pentru păstrarea unei mulțimi dinamice de date S în care fiecărui element i se asociază o valoare numită *prioritate*.

În această unitate vom discuta operațiile principale pentru cozile de prioritate, împreună cu algoritmi în pseudocod și complexitatea acestora. De asemenea vom prezenta exemple pentru ilustrarea acestor operații.

În toate exemplele vom considera cozi cu max-prioritate. Pentru cozile cu min-prioritate trebuie doar schimbate inegalitățile.



4.2.2 Obiective

La sfârșitul acestei unități de învățare studenții vor înțelege:

- Ce este o coadă de priorități.
- Care sunt operațiile de bază pe o astfel de coadă.
- Care este complexitatea operațiilor.



Durata medie de studiu individual

Parcursul de către studenți a acestei unități de învățare se face în 2 ore.

Cozile de prioritate pot fi utilizate pentru gestionarea proceselor pe un calculator, astfel încât la fiecare moment să fie executat procesul cu prioritatea maximă. În coadă se pot insera procese noi în orice moment. Un alt exemplu de aplicație îl constituie utilizarea lor în implementarea algoritmilor de căutare informată (Dijkstra, A^*). Termenul de „coadă” provine din analogia cu cozile obișnuite, unde inserarea elementelor se face la un capăt, iar extragerea la celălalt, utilizatorul având acces direct doar la aceste două capete. Diferența față de cozile clasice constă în faptul că, în cozile de prioritate, există un mecanism de rearanjare automată care, după fiecare inserare sau extragere, restabilește proprietatea de heap.

4.2.3 Operații pe cozi de prioritate

1. Determinarea elementului cu prioritate maximă. Acesta se află în nodul rădăcină.

Algoritm: ELEMENT_MAXIM

Input: Un heap max H

start

| return $H[0]$

stop

Complexitate: $O(1)$

2. Extragerea maximului: Se plasează ultimul element din heap pe prima poziție, se scade dimensiunea heap-ului cu o unitate, iar apoi se reface heap-ul prin apelul SIFT-DOWN.

Algoritm: EXTRAGE_ELEMENT_MAXIM

Input: Un heap max H cu *dimensiune* elemente

start

| $H[0] \leftarrow H[H.dimensiune - 1]$

| $H.dimensiune \leftarrow H.dimensiune - 1$

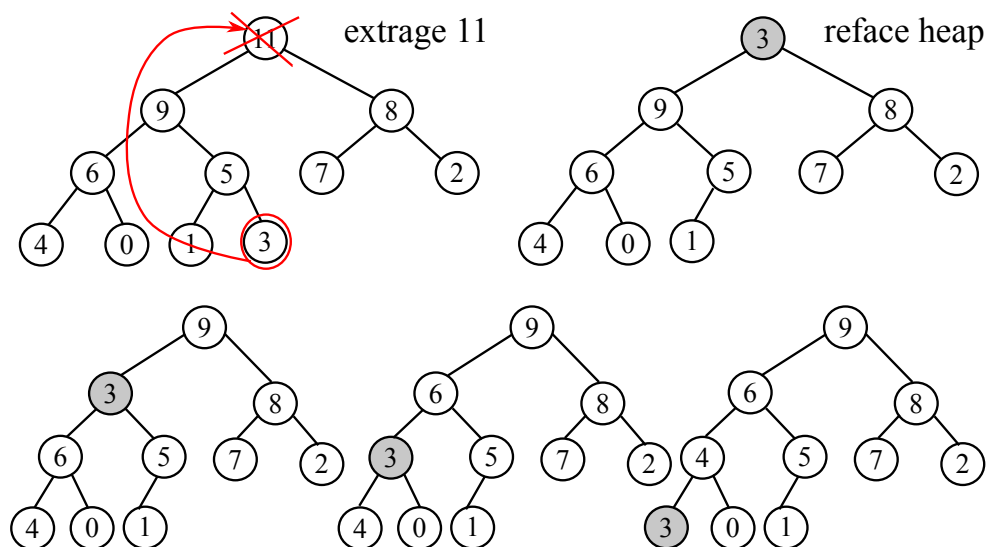
| SIFT-DOWN($H, 0$)

stop

Complexitate: $O(\log_2 n)$ - se aplică o singură dată SIFT-DOWN.



Exemplu pentru extragerea maximului dintr-un heap.



3. Prin creșterea priorității elementului de pe poziția i este posibil să se strice proprietatea de heap-max, deoarece s-ar putea ca noua valoare a elementului de pe poziția i să fie mai mare decât a părintelui său. Pentru refacerea proprietății de heap-max în această situație, se urcă din părinte în părinte către rădăcină, până se găsește o poziție potrivită pentru noua valoare. Algoritmul de urcare a unui element pe poziția potrivită este cunoscut în literatură și sub denumirea de *Sift-Up*.

Algoritm: SIFT-UP

Input: Un heap H , indicele i al elementului care trebuie urcat în heap.

start

$val \leftarrow H[i]$ //valoarea care trebuie urcată

$p \leftarrow (i - 1)/2$ //indicele părintelui

cat timp $i > 0$ și $H[p] < val$ **executa**

$H[i] \leftarrow H[p]$

$i \leftarrow p$

$p \leftarrow (i - 1)/2$

sfarsit_cat_timp

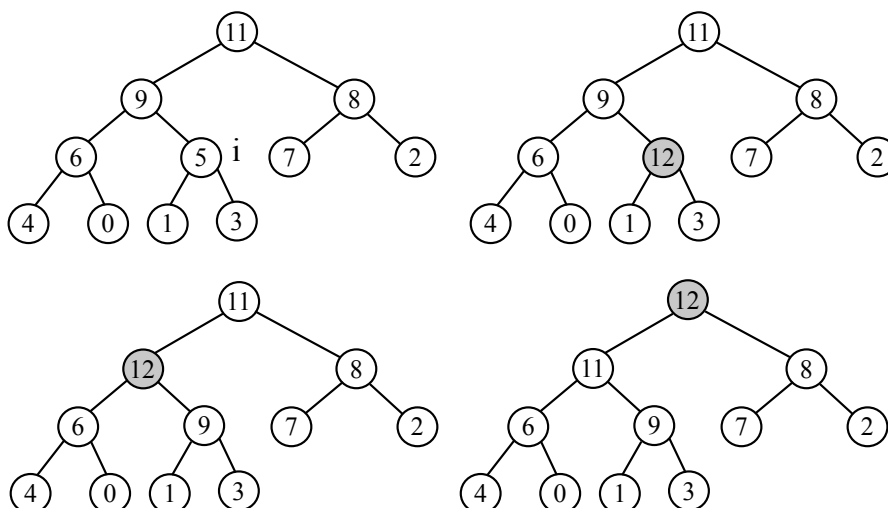
$H[i] \leftarrow val$

stop

Complexitate: $O(\log_2 n)$ - se pornește în cel mai defavorabil caz de la o frunză către rădăcină și deci complexitatea depinde de înălțimea arborelui.



Exemplu: Creșterea priorității nodului cu cheia 5 la valoarea 12.



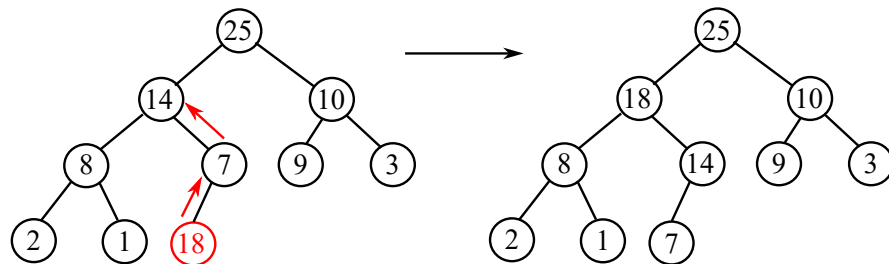
4. Adăugarea unui element nou într-un max-heap: se mărește dimensiunea heap-ului, se plasează noul element pe ultima poziție și apoi se aplică funcția SIFT-UP pentru acest nou element cu valoarea priorității asociate.

Algorithm: ADAUGA**Input:** Valoarea val , de inserat în coada de priorități H **start** $H[H.dimensiune] \leftarrow val$ $H.dimensiune \leftarrow H.dimensiune + 1$ SIFT-UP($H, H.dimensiune - 1$)**stop****Complexitate:** $O(h) = O(\log_2 n)$ - se pornește de la o frunză către rădăcină.**Exemplu:** Inserarea valorii 18 în heap.ADAUGA($H, 18$)H:

25	14	10	8	7	9	3	2	1	18
----	----	----	---	---	---	---	---	---	----

 H:

25	18	10	8	14	9	3	2	1	7
----	----	----	---	----	---	---	---	---	---

**Observații:**

- Pe un max-heap se pot implementa în complexitate $O(\log_2 n)$ operații cu cozi de prioritate.
- Construcția unui heap-max, care s-a făcut prin apelarea funcției SIFT-DOWN, se poate realiza și prin inserții succesive ale nodurilor în heap.



Implementați o coadă de priorități cu toate funcțiile corespunzătoare.

**Să ne reamintim...**

- Cozile de priorități sunt structuri de date ce pot manipula în mod eficient mulțimi dinamice de date, în care fiecărui element i se asociază o prioritate.
- Cozile de prioritate pot fi implementate eficient folosind heap-uri binare, iar operațiile principale sunt: determinarea valorii maxime (respectiv minime), extragerea maximumului (respectiv minimumului) și inserția unui element nou.

4.2.4 Rezumat



În această secțiune au fost discutate cozile de prioritate, cu accentul pe cozi de prioritate max, operațiile de bază pe aceste structuri, algoritmi corespunzători precum și complexitatea acestora.



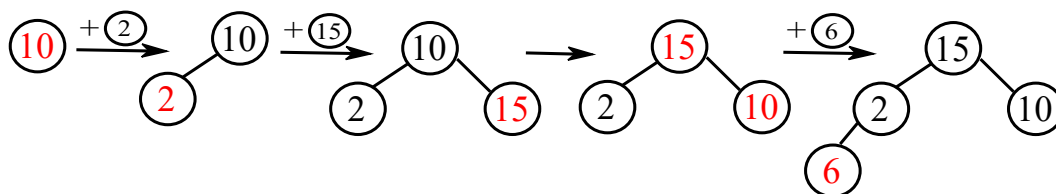
4.2.5 Test de autoevaluare

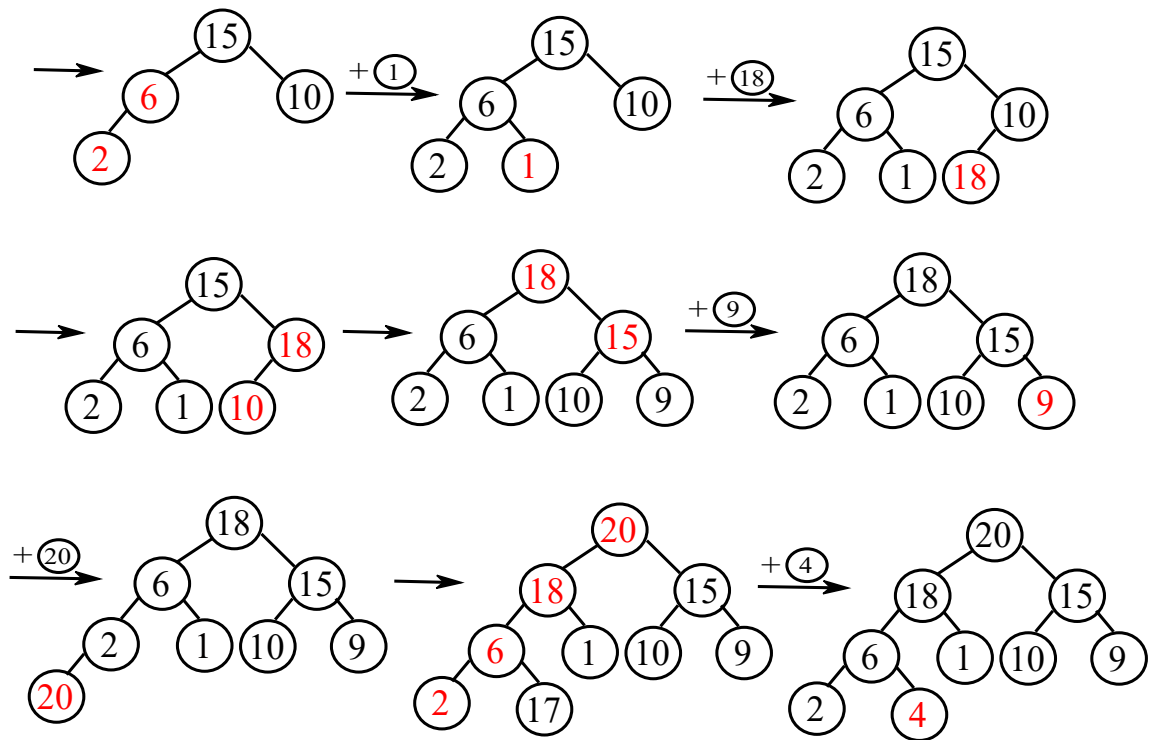
1. Inserți într-o coadă de prioritate max, inițial vidă, valorile: 10, 2, 15, 6, 1, 18, 9, 20, 4.
2. Ștergeți elementul cu prioritatea cea mai mare din coada de la exercițiul anterior și precizați reprezentarea vectorială a heap-ului rezultat.
3. Care este complexitatea determinării celui mai mic element într-o coadă de priorități max și respectiv min?
4. Scrieți un algoritm pseudocod pentru determinarea numărului de elemente cu prioritate mai mare decât un prag dat.
5. Scrieți un algoritm pseudocod pentru ștergerea valorilor mai mici decât un prag dat dintr-o coadă de priorități.

4.2.6 Răspunsuri la testul de evaluare a cunoștințelor

1. Inserți într-o coadă de prioritate max, inițial vidă, valorile: 10, 2, 15, 6, 1, 18, 9, 20, 4.

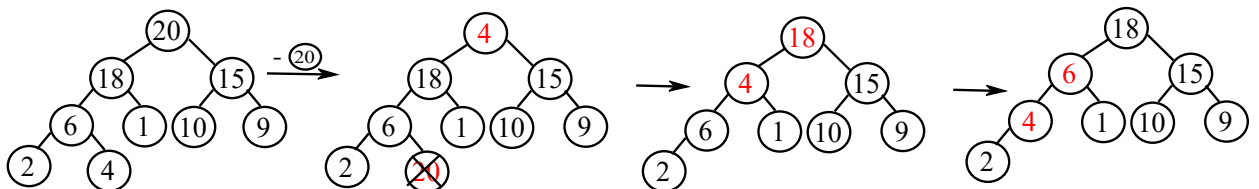
Rezolvare: Fiecare nod nou se inserează pe ultimul nivel al arborelui, în prima poziție liberă, astfel încât să se păstreze proprietatea de arbore binar complet. Ulterior, nodul inserat este comparat cu părintele său și urcă în arbore atât timp cât valoarea părintelui este mai mare decât valoarea nou adăugată.





2. Ștergeți elementul cu prioritatea cea mai mare din coada de la exercițiul anterior și precizați reprezentarea vectorială a heap-ului rezultat.

Rezolvare: Pentru eliminarea elementului cu prioritatea cea mai mare (20, în acest caz), acesta este interschimbabil cu elementul aflat cel mai la dreapta pe ultimul nivel al arborelui (elementul 4). Apoi, elementul 20 este eliminat, iar valoarea mutată în rădăcină va fi coborâtă în arbore pentru a restabili proprietatea de heap-max.



Vectorul rezultat, corespunzător este:

$$\{18, 6, 15, 4, 1, 10, 9, 2\}$$

3. Care este complexitatea determinării celui mai mic element într-o coadă de priorități max și respectiv min?

Rezolvare: Într-o coadă de priorități max, cel mai mic element se poate afla doar într-o frunză, deci în cel mai defavorabil caz trebuie parcurse toate frunzele. Numărul de frunze este aproximativ $n/2$, unde n este numărul total de noduri, deci complexitatea este $O(n)$. Dacă utilizăm o coadă de priorități min, cel mai mic element este mereu în rădăcină, deci complexitatea determinării lui este $O(1)$.

4. Scrieți un algoritm pseudocod pentru determinarea numărului de elemente cu prioritate mai mare decât un prag dat.

Rezolvare: Pentru a determina numărul de valori dintr-o coadă de priorități care sunt mai mari decât un anumit prag, este necesară extragerea elementelor pe rând, până la întâlnirea unei valori mai mici decât pragul specificat. Pentru a nu modifica starea cozii, elementele extrase sunt stocate într-un vector auxiliar, iar la final acestea sunt reinserate în coada de priorități.

Algoritm: NUMARARE

Input: O coadă de priorități C , un prag $prag$

Output: Numărul elementelor cu prioritate $> prag$

start

$V \leftarrow \emptyset$ - vector auxiliar

$nr_elemente \leftarrow 0$

cat timp $C \neq \emptyset$ și $PRIM(C) > prag$ **executa**

$INSEREAZA(V, PRIM(C))$

$EXTRAGE_ELEMENT_MAXIM(C)$

$nr_elemente \leftarrow nr_elemente + 1$

sfarsit_cat_timp

pentru fiecare element din V **executa**

$ADAUGA(C, element)$

sfarsit_for

return $nr_elemente$

stop

$PRIM(C)$ returnează primul element din coadă, care este același cu cel mai mare element.

5. Scrieți un algoritm pseudocod pentru ștergerea valorilor mai mici decât un prag dat dintr-o coadă de priorități max.

Rezolvare: Pentru a elimina valorile mai mici decât un prag specificat, este necesară extragerea elementelor mai mari din coada de priorități (acestea fiind accesate primele într-o coadă de tip max-heap). Elementele extrase care depășesc pragul sunt stocate într-un vector auxiliar, iar la final sunt reinsertate în coada de priorități, care între timp a fost golită. Astfel, în coadă vor rămâne doar elementele care respectă condiția impusă.

Algoritm: ELIMINA_VALORI_MICI

Input: O coadă de priorități C , un prag $prag$

start

$V \leftarrow \emptyset$ - vector auxiliar

cat_timp $C \neq \emptyset$ și $PRIM(C) \geq prag$ **executa**

 | $INSEREAZA(V, PRIM(C))$

 | $EXTRAGE_ELEMENT_MAXIM(C)$

sfarsit_cat_timp

$GOLESTE(C)$

pentru fiecare element din V **executa**

 | $ADAUGA(C, element)$

sfarsit_for

stop

Capitolul 5

Arbori binari de căutare

Introducere

O problemă esențială în proiectarea algoritmilor și a structurilor de date este căutarea eficientă a unei valori într-o mulțime ordonată. În acest scop a fost dezvoltat algoritmul de căutare binară, caracterizat printr-o complexitate logaritmică. Totuși, acest algoritm presupune o mulțime statică, adică o mulțime care nu se modifică și asupra căreia se efectuează exclusiv operații de căutare. În cazul mulțimilor dinamice, unde inserările și/sau eliminările de elemente sunt frecvente, apare suplimentar problema menținerii ordonării. Se cunoaște faptul că cei mai eficienți algoritmi de sortare pe bază de comparații au o complexitate de $O(n \log n)$, unde n reprezintă numărul de elemente, cu excepția unor algoritmi precum Count Sort, a căror aplicabilitate este limitată. Arborii binari de căutare au fost concepuți pentru a permite atât menținerea unei mulțimi ordonate, cât și realizarea căutării într-o manieră similară căutării binare. În acest modul vor fi prezentate arborii binari de căutare, vor fi analizate problemele asociate echilibrării acestora și vor fi discutate două tipuri de arbori auto-echilibrați.

Competențe

La sfârșitul acestui modul de învățare studenții:

- Definesc corect un arbore binar de căutare și cunosc operațiile de bază.
- Definesc ce este un arbore AVL și care este modul în care se auto-echilibrează.
- Definesc arborii roșu-negru și știu cum se produce echilibrarea acestora.
- Explică complexitatea operațiilor de bază în arborii binari de căutare care se auto-echilibrează.

5.1 Unitatea de învățare 1 - Arbori binari de căutare



5.1.1 Introducere

Arborii binari de căutare sunt un caz particular de arbori binari, care au o ordonare a cheilor ce permite căutarea unei chei în manieră binară. În această unitate vor fi descriși acești arbori, împreună cu operațiile principale, pentru care vor fi prezentați și algoritmi în pseudocod.



5.1.2 Obiective

La sfârșitul acestei unități de învățare studenții vor înțelege:

- Ce este un arbore binar de căutare.
- Care este înălțimea unui arbore binar de căutare.
- Care sunt operațiile de bază într-un arbore binar de căutare și care este complexitatea acestora.
- Ce este operația de rotație.



Durata medie de studiu individual

Parcurgerea de către studenți a acestei unități de învățare se face în 2 ore.

5.1.3 Noțiuni de bază

Definiție: un arbore binar de căutare este un arbore binar în care:

- Fiecare nod are o valoare numită cheie
- Pentru fiecare nod este valabil:
 - toate nodurile din subarborele stâng au cheile mai mici decât cheia părintelui;
 - toate nodurile din subarborele drept au cheile mai mari decât cheia părintelui.

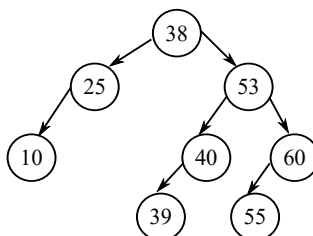
În cazul în care relația de ordine nu este strictă, dacă nodurile cu chei egale se inserează pe aceeași parte a arborelui, inserția multor noduri cu chei egale conduce la obținerea unui arbore relativ dezechilibrat, având ca urmare o creștere a complexității operațiilor. Există diferite metode de tratare a acestei situații, în continuare însă vom considera numai arbori binari de căutare în care cheile sunt unice.

O altă observație este aceea, că pot exista și arbori în care este valabilă inegalitatea inversă, adică pentru fiecare nod x , toate cheile din subarborele stâng sunt mai mari și toate cele din

subarborele drept sunt mai mici decât cheia lui x .



Exemplu de arbore binar de căutare.



Implementarea unui arbore binar de căutare poate fi realizată folosind noduri cu o structură, precum cea din figura 5.1. Astfel, fiecare nod x are câmpurile: $x.cheie$ = cheia nodului, $x.stanga$ și $x.dreapta$ = fiul stâng și respectiv fiul drept, $x.parinte$ = părintele nodului x .

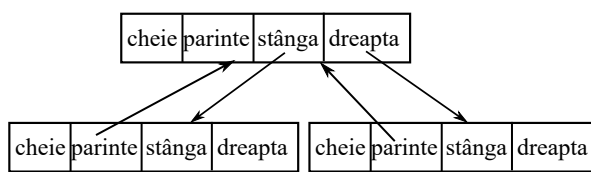


Figura 5.1

5.1.4 Operații într-un arbore binar de căutare

Operațiile de bază într-un arbore binar de căutare sunt: căutarea binară a unei chei, inserția, respectiv ștergerea unor chei și parcurgerea arborelui. Operații ajutătoare sunt determinarea nodului cu cheia minimă/maximă din arbore, precum și căutarea binară a succesorului / predecesorului unui nod.

Observații legate de complexitatea operațiilor

1. Înălțimea maximă h_{max} a unui arbore binar cu n noduri se obține atunci când acesta are pe fiecare nivel un număr minim de noduri, adică 1, și astfel, înălțimea este $n - 1$.
2. Înălțimea minimă h_{min} a unui arbore binar cu n noduri se obține atunci când fiecare nivel k conține numărul maxim de noduri, adică 2^k . În această situație, fiecare nod intern are exact doi fii, iar prin inducție se poate demonstra că $h = \log_2 n$.

3. Complexitatea operațiilor principale într-un arbore binar de căutare este $O(h)$, unde h este înălțimea arborelui, deci este proporțională cu această înălțime. De fapt, dacă arborele conține n noduri atunci $O(\log_2 n) \leq T(n) \leq O(n)$, unde $T(n)$ reprezintă complexitatea de timp a algoritmului. Excepție fac parcurgerile, care sunt liniare în n .
4. În cazul unui arbore binar de căutare oarecare nu poate fi garantată complexitatea căutării binare, adică $O(\log_2 n)$.

Există arbori binari de căutare care se auto-balansează, de exemplu arborii AVL și arborii roșu-negru. Pentru aceștia se demonstrează faptul că au complexitatea operațiilor $O(\log_2 n)$.

În continuare prezentăm operațiile principale, împreună cu algoritmi în pseudocod, având la bază algoritmi din [4].

Căutarea binară

Considerând un arbore binar de căutare T cu rădăcina $T.rad$, pentru a găsi nodul cu o cheie dată val , se pornește cu un nod curent $x = T.rad$. La fiecare iterație se compară cheia nodului x cu val . Dacă $x.cheie = val$ atunci se returnează un pointer la nodul x . Dacă $val < x.cheie$ atunci se continuă căutarea în subarborele stâng al lui x , altfel se continuă căutarea în subarborele drept al lui x .

Algoritm: BIN_CAUTA

Input: Un arbore binar T , cheia căutată val .

Output: Pointer către nodul x cu $x.cheie = val$ sau nil

start

$x \leftarrow T.rad$

cat_timp $x \neq nil$ și $x.cheie \neq val$ **executa**

daca $val < x.cheie$ **atunci**

$x \leftarrow x.stanga$

sfarsit_daca

altfel

$x \leftarrow x.dreapta$

sfarsit_daca

sfarsit_cat_timp

returneaza(x)

stop

Minimul dintr-un arbore de căutare

Nodul cu informația minimă din subarboarele de rădăcină x a unui arbore binar de căutare poate fi găsit pornind de la nodul x și coborând la fiii stânga până la cea mai din stânga frunză. Funcția BIN_MINIM returnează nodul cu informația minimă.

Algoritm: BIN_MINIM

Input: Arborele binar de căutare T și nodul x

Output: Pointer la nodul cu cheia minimă din subarboarele de rădăcină x

start

daca $x \neq nil$ **atunci**

cat timp $x.stanga \neq nil$ **executa**

$x \leftarrow x.stanga$

sfarsit_cat_timp

sfarsit_daca

 returneaza(x)

stop

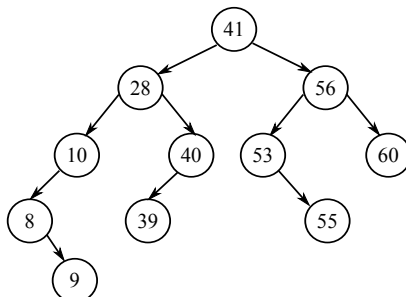
Observație: maximul se determină în mod similar și anume parcurgând fii din dreapta până la cea mai din dreapta frunză.

Succesorul binar

Succesorul binar al unui nod x într-un arbore binar de căutare este acel nod y din arbore, a cărui cheie are valoarea imediat următoare cheii lui x în șirul sortat al cheilor din arbore. Elementul maxim nu are succesor.



Exemplu pentru succesorul binar în arbore binar de căutare din figură.



BIN_SUCCESOR(41) = 53

BIN_SUCCESOR(39) = 40

BIN_SUCCESOR(9) = 10

BIN_SUCCESOR(60) - nu există

Dacă există succesorul binar, el este:

- cel mai mic element din $x.dreapta$, dacă $x.dreapta \neq nil$;
- un nod părinte y pentru care x se află în subarborele stâng al lui y , dacă x nu are descendent drept. În această situație este suficient să se urce în arbore de la x până la primul strămoș cu cheia mai mare decât a lui x .

Algoritm: BIN_SUCCESOR

Input: Arborele binar de căutare T și nodul x

Output: Pointer la nodul y , succesor binar al lui x

start

daca $x = nil$ **atunci**

 | returneaza(x)

sfarsit_daca

daca $x.dreapta \neq nil$ **atunci**

 | $y = \text{BIN_MINIM}(x.dreapta)$

sfarsit_daca

altfel

 | $y \leftarrow x.parinte$

cat timp $y \neq nil$ și $y.cheie < x.cheie$ **executa**

 | $y \leftarrow y.parinte$

sfarsit_cat_timp

sfarsit_daca

 returneaza(y)

stop

Inserarea unui nod

Se consideră arborele binar de căutare T cu rădăcina $T.rad$ și se dorește inserarea unei noi chei val în arbore. Pentru acest lucru trebuie creat un nou nod z , care are câmpurile

$$z.cheie = val, z.stanga = z.dreapta = z.parinte = nil.$$

Ideea principală în cazul inserării este următoare: pornind cu nodul curent $x = T.rad$ se coboară în arbore, până la un nod, care are cel mult un fiu și care poate fi părintele nodului z . Pentru a respecta proprietatea de arbore binar de căutare, dacă informația nodului curent x este mai mare decât val , atunci z se va insera în stânga al lui x , altfel se va insera în dreapta.

În algoritmul următor, bazat pe [4], x reprezintă nodul curent, care pornește din rădăcina lui T , iar y reprezintă părintele lui x , inițial nil . Se verifică de asemenea, să nu existe deja cheia respectivă în arbore.

Algoritm: BIN_INSEREAZA

Input: Arborele binar de căutare T și cheia val

start

$x \leftarrow T.rad$

$y \leftarrow nil$

cat_timp $x \neq nil$ și $x.cheie \neq val$ **executa**

$y \leftarrow x$

daca $val < x.cheie$ **atunci**

$x \leftarrow x.stanga$

sfarsit_daca

altfel

$x \leftarrow x.dreapta$

sfarsit_daca

sfarsit_cat_timp

daca $x \neq nil$ **atunci**

 scrie("Cheia este deja în arbore")

 returneaza()

sfarsit_daca

 alocă memorie pentru un nod z

$z.cheie \leftarrow val$

$z.stanga \leftarrow nil$

$z.dreapta \leftarrow nil$

$z.parinte \leftarrow y$

daca $y = nil$ **atunci**

$T.rad \leftarrow z$

sfarsit_daca

altfel

daca $val < y.cheie$ **atunci**

$y.stanga \leftarrow z$

sfarsit_daca

altfel

$y.dreapta \leftarrow z$

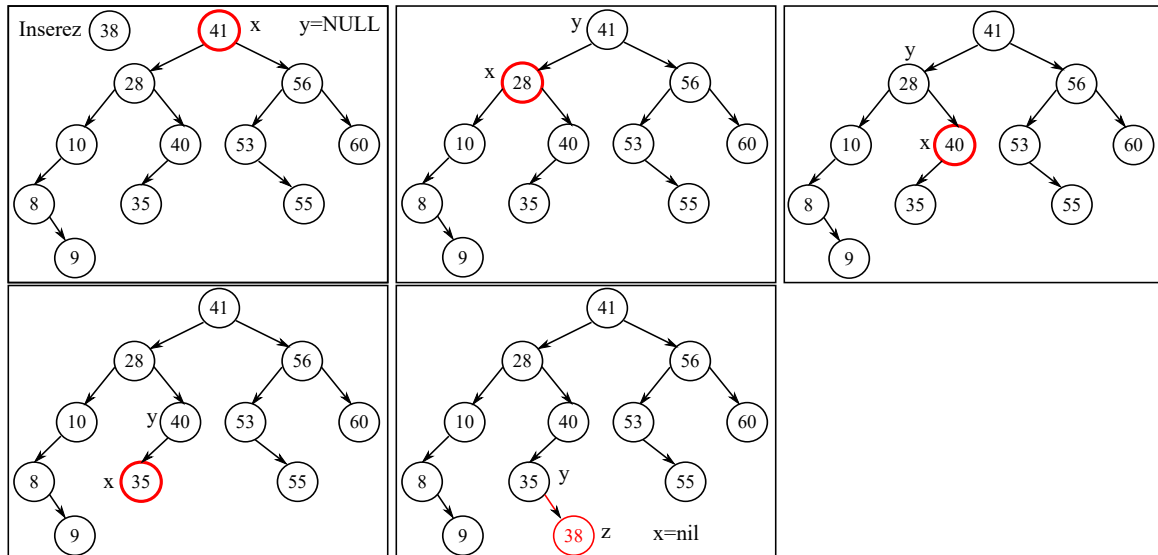
sfarsit_daca

sfarsit_daca

stop



Exemplu pentru ilustrarea algoritmului de inserție a unui nod cu cheia 38 într-un arbore binar de căutare. Nodul curent cu care se compară nodul ce se inserează este marcat cu roșu.



Ștergerea unui nod

Ștergerea unui nod z dintr-un arbore binar T cu rădăcina $T.rad$ este ceva mai elaborată decât inserarea. Sunt luate în considerare următoarele cazuri:

1. z nu are fii și atunci este pur și simplu înlocuit cu nil .
2. z are un singur fiu diferit de nil . Atunci se înlocuiește z cu acel fiu.
3. z are doi fii diferiți de nil . Atunci se determină succesorul y al lui z care se află în subarborele drept al lui z și evident nu are descendent stâng. Apoi se înlocuiește nodul z cu nodul y , iar y se înlocuiește cu fiul său drept.

Observație: În locul succesoriului se poate folosi și predecesorul.

În funcția `BIN_ȘTERGE` se consideră T arborele binar, z nodul care trebuie șters și y nodul cu care se înlocuiește z . Cele 3 cazuri descrise mai sus vor fi cuprinse în funcție în următoarele cazuri:

1. z nu are fiu stâng \Rightarrow se înlocuiește z cu fiul drept - eventual nil . Acest caz include și cazul în care z nu are nici un fiu.
2. z nu are fiu drept \Rightarrow se înlocuiește z cu fiul stâng

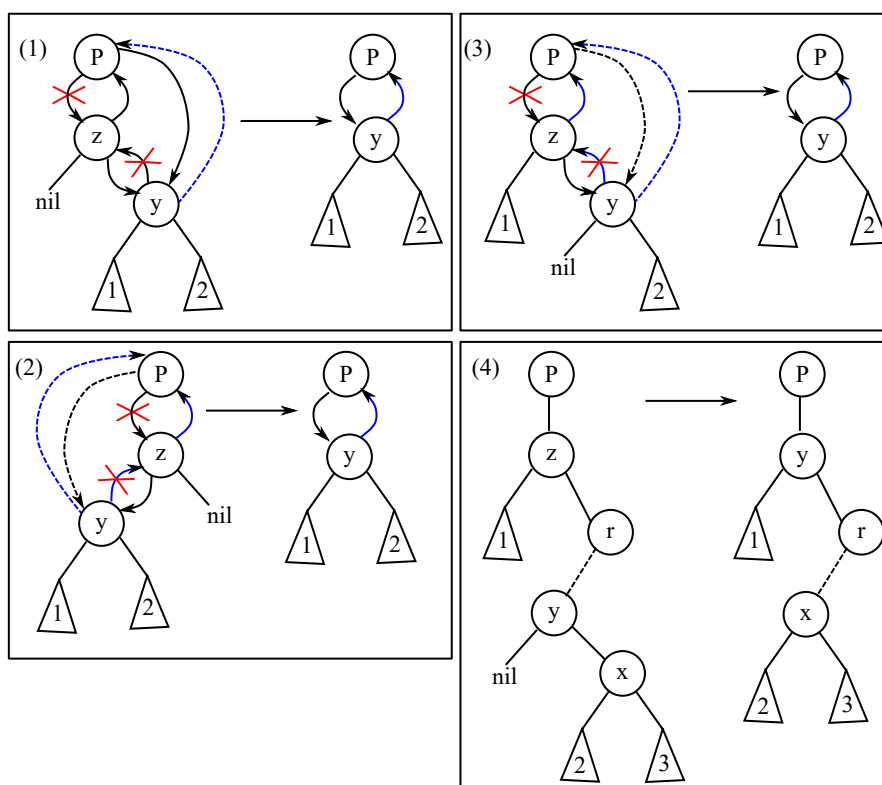


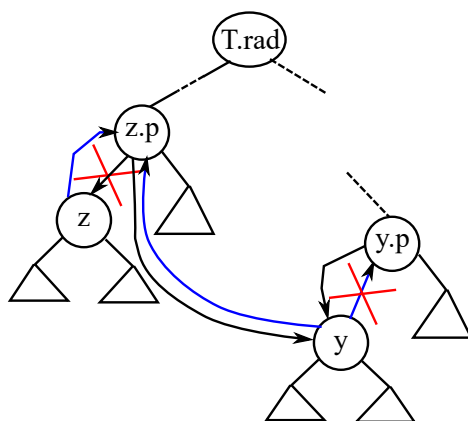
Figura 5.2: Ștergerea unui nod z care: (1) are fiul stâng nil ; (2) are fiul drept nil ; (3) are ambii fii nenuli, dar succesorul y este fiu al lui z ; (4) are ambii fii nenuli, dar succesorul y nu este fiu al lui z .

3. z are ambii fii nenuli $\Rightarrow y = \text{succesorul lui } z$ care se află în subarborele drept al lui z și are fiul stâng nil
 - a. y este fiul drept al lui $z \Rightarrow$ se înlocuiește z cu y (fiul drept al lui y rămâne neschimbat iar fiul stâng al lui z devine fiul stâng al lui y)
 - b. y nu este fiul drept al lui $z \Rightarrow$ se înlocuiește y cu fiul său drept iar apoi se înlocuiește nodul z cu nodul y .

Cazurile considerate de către funcția `BIN_ȘTERGE` sunt ilustrate în figura 5.2. În toate aceste cazuri trebuie înlocuit un nod z (cel care se șterge) cu alt nod y (eventual nil), ceea ce înseamnă că trebuie tratată legătura dintre nodul y și părintele nodului z . Pentru a eficientiza procesul se poate folosi o funcție ajutătoare ([4]) `TRANSPLANT(T, z, y)` care înlocuiește în arborele T nodul z ca fiu al lui $z.parinte$ cu nodul y . De fapt această funcție realizează doar managementul legăturilor între părintele lui z și nodul y , legăturile cu fiii se realizează separat în funcția de ștergere propriu-zisă.

Algorithm: TRANSPLANT**Input:** Arborele binar de căutare T , nodurile z și y **start** **daca** $T.rad = z$ **atunci** | $T.rad \leftarrow y$ **sfarsit_daca** **altfel** **daca** $z.parinte.stanga = z$ **atunci** | $z.parinte.stanga \leftarrow y$ **sfarsit_daca** **altfel** | $z.parinte.dreapta \leftarrow y$ **sfarsit_daca** **sfarsit_daca** **daca** $y \neq nil$ **atunci** | $y.parinte \leftarrow z.parinte$ **sfarsit_daca****stop**

Modul de funcționare al funcției TRANSPLANT este ilustrat în figura 5.3.

Figura 5.3: Managementul legăturilor între părintele nodului z și nodul y .

În continuare este prezentat algoritmul de ștergere a unui nod z dintr-un arbore binar de căutare T [4].

Algoritm: BIN_STERGE**Input:** Arborele binar de căutare T și nodul $z \neq nil$, găsit cu funcția BIN_CAUTA

start

```

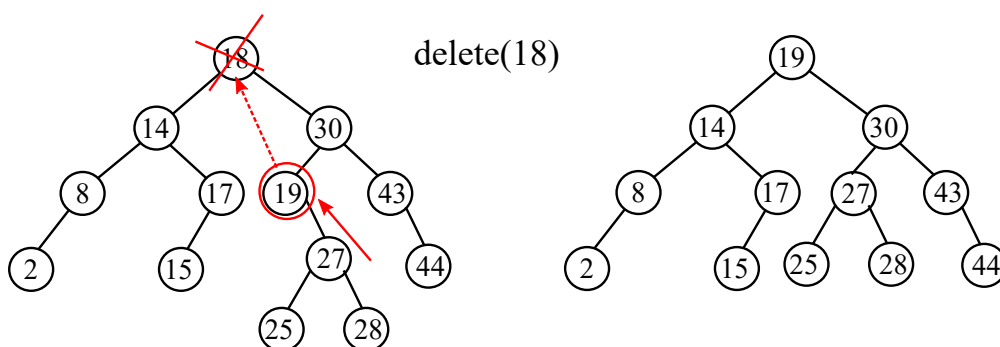
daca  $z.stanga = nil$  atunci
  | TRANSPLANT( $T, z, z.dreapta$ )
sfarsit_daca
altfel
  | daca  $z.dreapta = nil$  atunci
  | | TRANSPLANT( $T, z, z.stanga$ )
  | sfarsit_daca
  | altfel
  | |  $y \leftarrow BIN\_SUCCESOR(z)$ 
  | | daca  $y \neq z.dreapta$  atunci
  | | | TRANSPLANT( $T, y, y.dreapta$ )
  | | |  $y.dreapta \leftarrow z.dreapta$ 
  | | |  $z.dreapta.parinte \leftarrow y$ 
  | | sfarsit_daca
  | | TRANSPLANT( $T, z, y$ )
  | |  $y.stanga \leftarrow z.stanga$ 
  | |  $z.stanga.parinte \leftarrow y$ 
  | | sfarsit_daca
  | sfarsit_daca

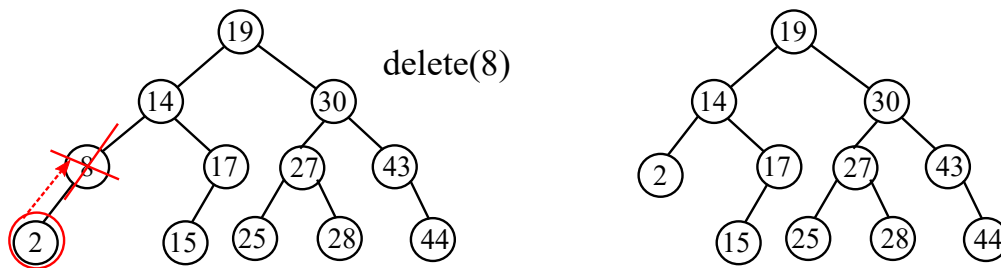
```

stop



Exemplu de ștergere dintr-un arbore binar de căutare. Din arborele din figură se șterg pe rând cheile 18, și 8.





Observație: în unele documentații [18] este prezentată o metodă mai simplă de ștergere a unui nod z cu doi fii nenuli dintr-un arbore binar de căutare. În loc să se înlocuiască nodul z cu succesorul său y și apoi să se șteargă z , se înlocuiește informația lui z cu informația lui y și apoi se șterge y , care are un singur fiu. Problema cu acest mod de ștergere este că, de fapt nu se va șterge pointerul care a fost trimis către funcție, ci un alt pointer. Dacă alte componente al programului păstrează pointeri către y , acest lucru va avea consecințe nefaste, acești pointeri fiind invalidați. În schimb metoda de mai sus, prezentată și în bibliografie [4], garantează faptul că pointerul șters este chiar cel trimis ca parametru către funcția de ștergere

Complexitatea tuturor operațiilor descrise mai sus, începând de la căutarea binară, au complexitatea $O(h)$, unde $h =$ înălțimea arborelui.

5.1.5 Rotația într-un arbore binar de căutare

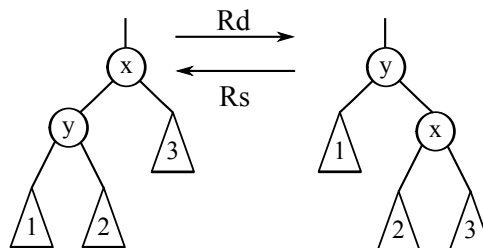


Figura 5.4: Rotații într-un arbore binar de căutare.

Am discutat faptul că într-un arbore binar de căutare nu poate fi garantată o înălțime logaritmică. Pentru n noduri se obține, în cel mai defavorabil caz, înălțimea egală cu $n - 1$. Acest lucru conduce la o complexitate liniară a căutării și astfel, construcția unui arbore binar de căutare își pierde scopul principal.

Există însă tipuri de arbori binari de căutare care se auto-echilibrează și care vor fi discutați în unitățile următoare. Echilibrarea în acești arbori are la bază o operație locală de rotație, care

schimbă structura locală de pointeri în arborele binar de căutare, dar păstrează proprietățile acestuia.

Operația de rotație este ilustrată în figura 5.4, unde notațiile au următoarea semnificație:

- Rd = rotație spre dreapta în jurul nodului x
- Rs = rotație spre stânga în jurul nodului y

În figura 5.5 este ilustrat grafic modul în care se realizează rotația.

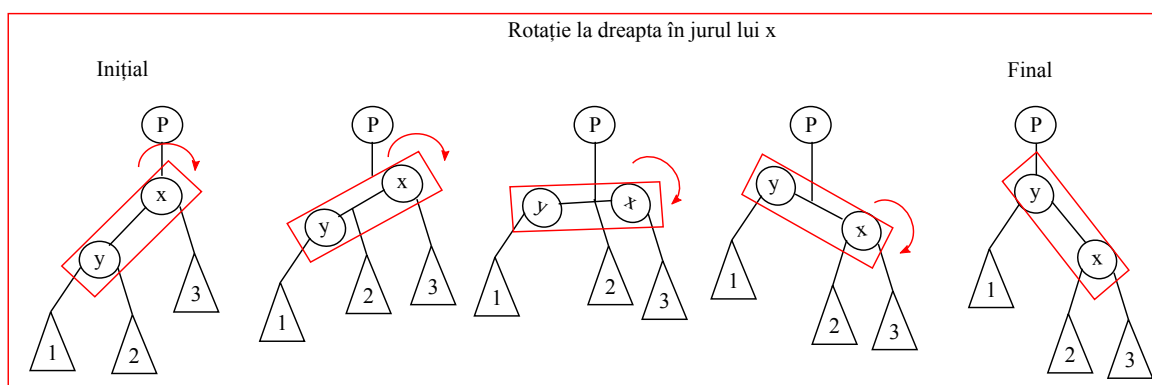


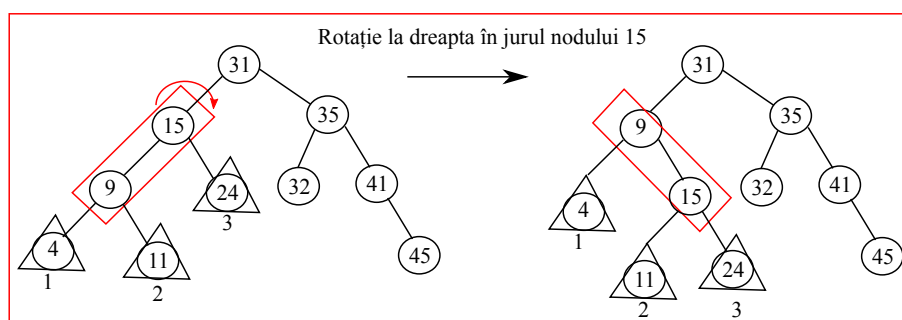
Figura 5.5: Rotație la dreapta în jurul nodului x într-un arbore binar de căutare.

Observații:

- Pentru a se putea efectua o rotație spre dreapta în jurul nodului x , este necesar ca nodul x să aibă fiu stâng diferit de nil .
- Pentru se a putea efectua o rotație spre stânga în jurul nodului y , este necesar ca nodul y să aibă un fiu drept diferit de nil .



Exemplu de rotație la dreapta în jurul nodului cheia 15 într-un arbore binar de căutare.



Algoritmul următor prezintă rotația la dreapta în jurul unui nod x într-un arbore binar T . Pentru rotația la stânga algoritmul este simetric și poate fi găsit în bibliografia recomandată [4].

Algoritm: ROTATIE-DREAPTA

Input: Un arbore binar T , nodul y , în jurul căruia are loc rotația

start

$y \leftarrow x.stanga$

$x.stanga \leftarrow y.dreapta$

daca $y.dreapta \neq nil$ **atunci**

$y.dreapta.parinte \leftarrow x$

sfarsit_daca

$y.parinte \leftarrow x.parinte$

daca $x.parinte = nil$ **atunci**

$T.rad \leftarrow y$

sfarsit_daca

altfel

daca $x.parinte.stanga = x$ **atunci**

$x.parinte.stanga \leftarrow y$

sfarsit_daca

altfel

$x.parinte.dreapta \leftarrow y$

sfarsit_daca

sfarsit_daca

$y.dreapta \leftarrow x$

$x.parinte \leftarrow y$

stop

Complexitate: $O(1)$



Implementați o structură de tip arbore binar de căutare, care dispune de toate funcțiile descrise în curs.



Să ne reamintim...

- Într-un arbore binar de căutare cheile sunt ordonate astfel încât, pentru orice nod x din arbore, toate cheile aflate în subarborele stâng sunt mai mici decât cheia lui x , iar cele din subarborele drept sunt mai mari decât aceasta.
- Un arbore binar de căutare permite căutarea unei chei în manieră binară.

- Parcurgerea în inordine a unui arbore binar de căutare are ca rezultat ordonarea cheilor din arbore.
- Complexitatea operațiilor de bază într-un arbore binar de căutare este determinată de înălțimea arborelui.
- Operația de rotație într-un arbore binar de căutare este o operație de modificare locală a structurii de pointeri din arbore.

5.1.6 Rezumat



În această secțiune au fost discutați arborii în general, precum și arborii binari în particular, împreună cu elementele constitutive, modurile de implementare și parcurgerile acestora. De asemenea a fost introdusă operația de rotație în jurul unui nod, care este utilă pentru echilibrarea unui arbore binar de căutare.



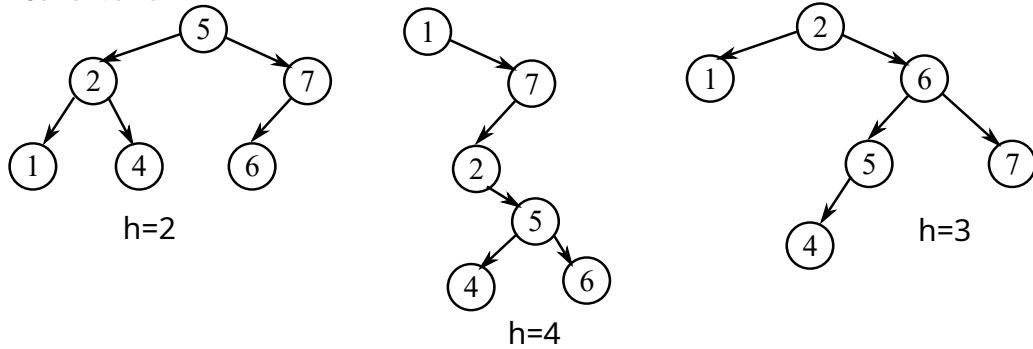
5.1.7 Test de autoevaluare

1. Desenați 3 arbori binari de căutare de înălțimi diferite, care pot fi formați utilizând cheile 5, 1, 7, 4, 6, 2.
2. Insezați într-un arbore binar de căutare, inițial vid, valorile: 15, 10, 20, 8, 14, 11, 12, 4, 9. Efectuați o rotație spre dreapta în jurul rădăcinii și apoi ștergeți nodurile 11 și 10.
3. Care este arborele binar de căutare, a cărui parcurgere în preordine este *RSD*: 23, 17, 10, 15, 19, 35, 26, 24, 30, 37?
4. Scrieți un pseudocod pentru verificarea dacă un arbore este arbore binar de căutare.
5. Scrieți un pseudocod pentru găsirea celui mai mic nod mai mare decât o valoare dată într-un arbore binar de căutare.

5.1.8 Răspunsuri la testul de evaluare a cunoștințelor

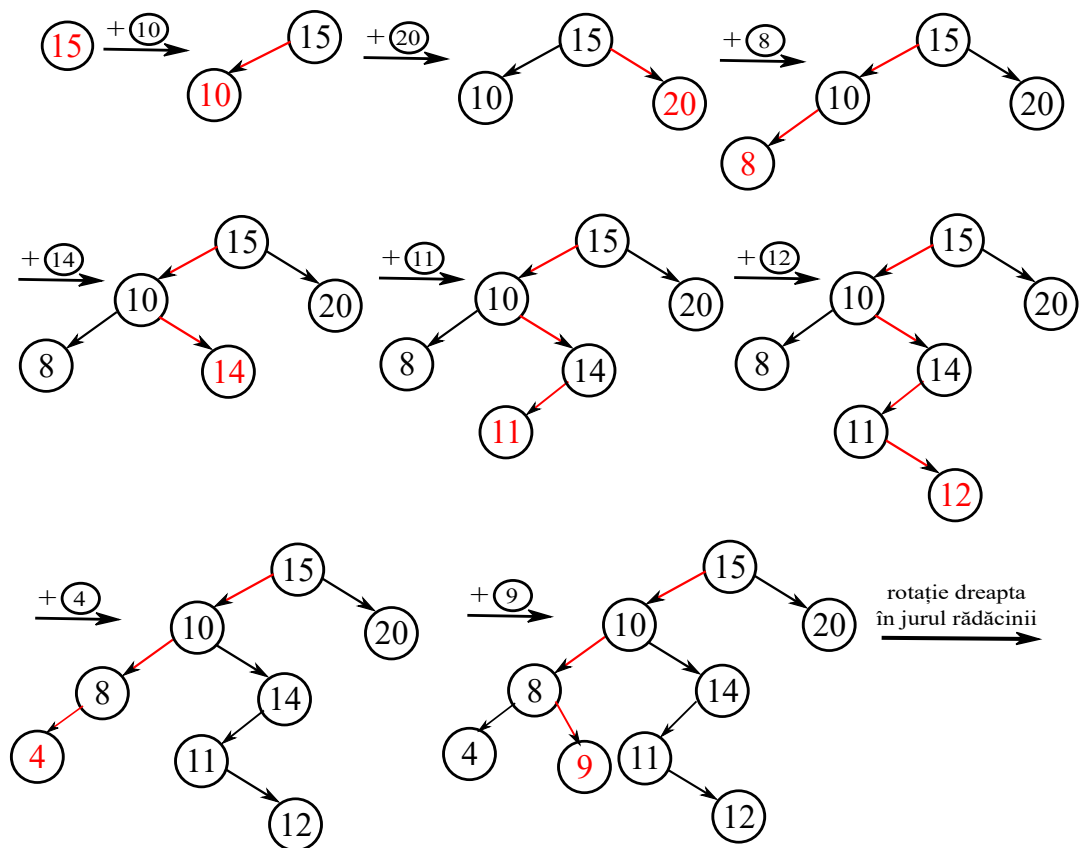
1. Desenați 3 arbori binari de căutare de înălțimi diferite, care pot fi formați utilizând cheile 5, 1, 7, 4, 6, 2.

Rezolvare:

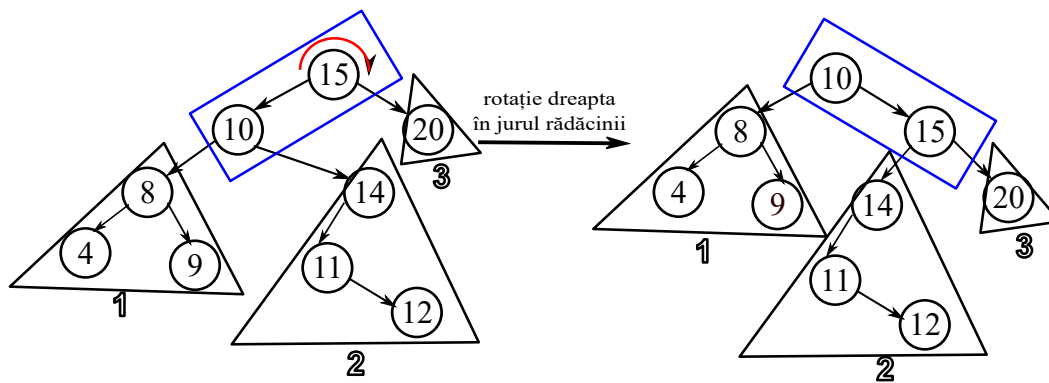


2. Insezați într-un arbore binar de căutare, inițial vid, valorile: 15, 10, 20, 8, 14, 11, 12, 4, 9. Efectuați o rotație spre dreapta în jurul rădăcinii și apoi ștergeți nodurile 11 și 10.

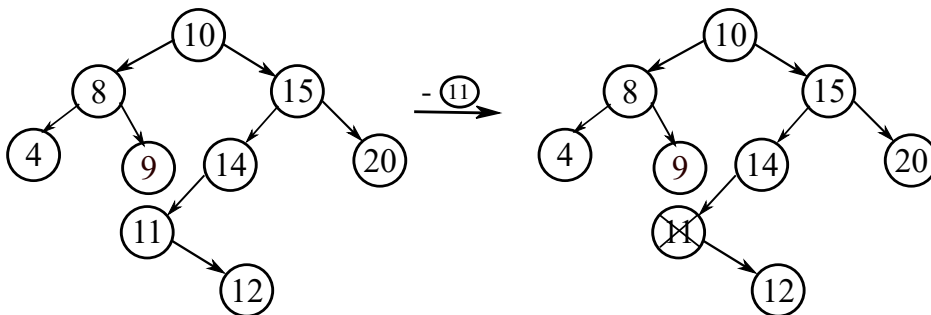
Rezolvare: Insezați întâi cheile. Acest proces este ilustrat în figurile următoare:



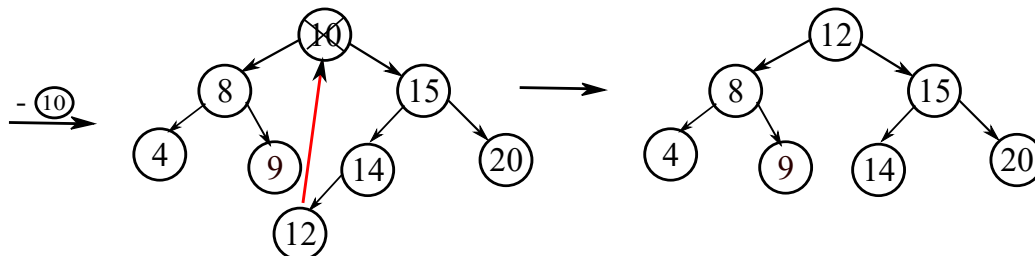
În arborele rezultat efectuăm acum rotația spre dreapta în jurul nodului rădăcină. Observăm că noua rădăcină va fi nodul 10, iar subarboarele marcat cu 2 se mută de la dreapta nodului 10 la stânga nodul 15.



Din arborele rezultat ștergem cheia 11 care are un singur fiu.



Nodul cu cheia 10 este chiar rădăcina și are doi fii nemuli, deci trebuie să urcăm succesorul, adică nodul cu cheia 12, în locul său.



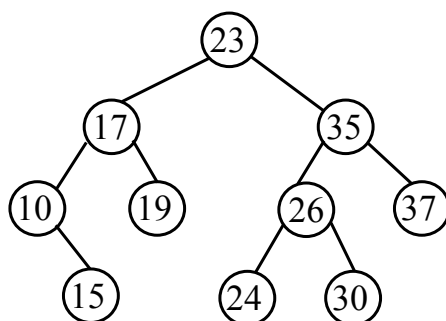
3. Care este arborele binar de căutare, a cărui parcurgere în preordine este $RSD : 23, 17, 10, 15, 19, 35, 26, 24, 30, 37$?

Rezolvare:

Varianta 1: Observăm faptul că, prin parcurgerea în inordine a unui arbore binar de căutare se obține șirul cheilor sortat crescător. Deci, cunoscând care sunt cheile arborelui, din parcurgerea în preordine (RSD), putem imediat obține parcurgerea în inordine. Utilizând cele două parcurgeri se vă reface arborele.

Varianta 2: Dacă luăm pur și simplu cheile în ordinea în care apar în parcurgerea RSD și le inserăm într-un arbore binar de căutare inițial vid, obținem arborele cerut.

Pentru cazul din exemplu, arborele corespunzător este:



4. Scrieți un pseudocod pentru verificarea dacă un arbore este arbore binar de căutare.

Rezolvare: Această problemă se poate rezolva printr-o parcurgere în adâncime. Vom considera cazul recursiv pentru simplitate.

Algoritm: VERIFICA-ARBORE-BINAR-CAUTARE

Input: Nodul *curent* în arborele binar *T*

start

daca *curent* = *nil* **atunci**

| returneaza(*adevarat*)

sfarsit_daca

daca *curent.stanga* ≠ *nil* și *curent.cheie* < *curent.stanga.cheie* **atunci**

| returneaza(*fals*)

sfarsit_daca

daca *curent.dreapta* ≠ *nil* și *curent.cheie* > *curent.dreapta.cheie* **atunci**

| returneaza(*fals*)

sfarsit_daca

ok = VERIFICA_BINAR_CAUT(*curent.stanga*) și

VERIFICA_BINAR_CAUT(*curent.dreapta*)

returneaza(*ok*)

stop

5. Scrieți un pseudocod pentru găsirea celui mai mic nod mai mare decât o valoare dată într-un arbore binar de căutare.

Rezolvare: Această problemă poate fi rezolvată prin adaptarea algoritmului de căutare a unei chei într-un arbore, prin introducerea unei restricții suplimentare care reflectă

condițiile impuse de cerința problemei.

Algorithm: CAUTARE-NOD

Input: Arborele binar T și valoarea val

Output: Nodul cel mai mic, cu valoarea mai mare decât val sau nil

start

```
daca  $rad = nil$  atunci
| returneaza( $nil$ )
sfarsit_daca
 $curent \leftarrow T.rad$ 
 $temp \leftarrow nil$ 
cat_timp  $curent$  executa
|
|   daca  $curent.cheie \leq val$  atunci
|   |    $curent \leftarrow curent.dreapta$ 
|   sfarsit_daca
|   altfel
|   |    $temp \leftarrow curent$ 
|   |    $curent \leftarrow curent.stanga$ 
|   sfarsit_daca
|   returneaza( $temp$ )
sfarsit_cat_timp
```

stop

5.2 Unitatea de învățare 2 - Arbori AVL



5.2.1 Introducere

Arborii AVL sunt arbori binari de căutare aproape balansați. Denumirea provine de la autorii acestor arbori, doi matematicieni ruși G.M. Adelson-Velsky și E.M. Landis [1]. Pentru a realiza balansarea, arborii binari de căutare de tip AVL atașează fiecărui nod o proprietate, numită factor de balansare, reprezentând diferența de înălțime între subarborii nodului. Valoarea absolută a acestei diferențe nu trebuie să depășească o unitate, astfel realizându-se o oarecare balansare, care garantează o complexitate logaritmică a operațiilor de bază.



5.2.2 Obiective

La sfârșitul acestei unități de învățare studenții vor înțelege:

- Ce este factorul de balansare al unui nod.
- Ce este un arbore AVL.
- Care sunt operațiile de bază într-un arbore AVL și cum se realizează balansarea.
- De ce operațiile într-un arbore AVL au complexitate logaritmică.



Durata medie de studiu individual

Parcursarea de către studenți a acestei unități de învățare se face în 3 ore.

5.2.3 Noțiuni generale

Arborii AVL sunt arbori binari de căutare aproape balansați. Această balansare se realizează folosind o proprietate suplimentară a fiecărui nod, numită *factor de balansare*.

Considerăm pentru fiecare nod x un *factor de balansare* dat prin

$$fb(x) = h(x.dreapta) - h(x.stanga)$$

adică, factorul de balansare al unui nod x este reprezentat de către diferența dintre înălțimea subarborelui său drept și înălțimea subarborelui său stâng. În unele documentații diferența se realizează între înălțimea subarborelui stâng și cea a subarborelui drept.

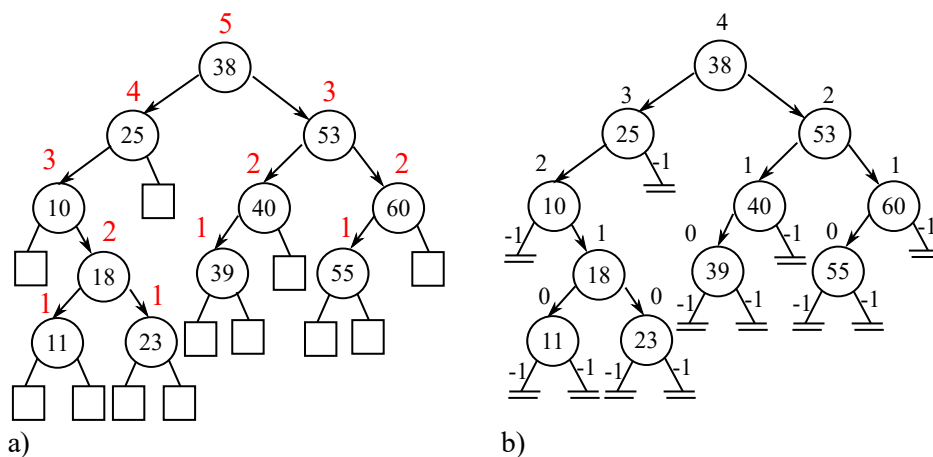


Figura 5.6: a) Înălțimile nodurilor, considerând frunzele legate de "noduri *nil*". b) Înălțimile nodurilor conform definiției uzuale, fără "noduri *nil*".

Înălțimea în arborii AVL

Definiția înălțimii unui nod este lungimea celui mai lung drum de la nod la o frunză, rezultă astfel că înălțimea unei frunze este 0. Pentru calculul corect al factorilor de balansare într-un AVL există două variante de considerare a arborelui:

- se consideră că frunzele, în loc să aibă pe stânga și pe dreapta *nil*, sunt legate de fapt de niște noduri nule. Astfel înălțimea pentru fiecare nod crește cu 1 și deci înălțimea unei frunze va fi 1, iar cea a unui "nod nul" va fi 0 (figura 5.6 a.);
- se consideră în continuare definiția clasică a înălțimii, iar pentru *nil* se consideră "înălțimea" -1 (pentru a calcula corect factorul de balansare al unui nod, care are un singur copil - figura 5.6 b.).

În ambele situații factorul de balansare obținut pentru un anumit nod este același. Precizările de mai sus sunt importante mai ales în contextul implementării unui arbore AVL. În continuarea acestui curs vom considera înălțimea unei frunze ca fiind 1, iar factorul de balansare pentru un nod, care are unul dintre copii *nil* se va calcula considerând copilul *nil* cu înălțimea 0, deci ca în figura 5.6 a).

Definiție - nod balansat: Un nod x se numește balansat, dacă $fb(x) \in \{-1, 0, 1\}$.

În figura 5.7 este prezentat un exemplu de arbore binar de căutare împreună cu factorii de balansare corespunzători fiecărui nod. Se observă faptul că nodurile cu cheile 10 și 25 nu sunt balansate.

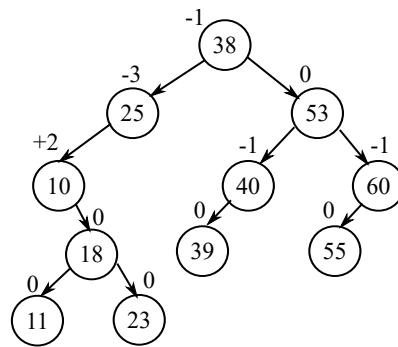
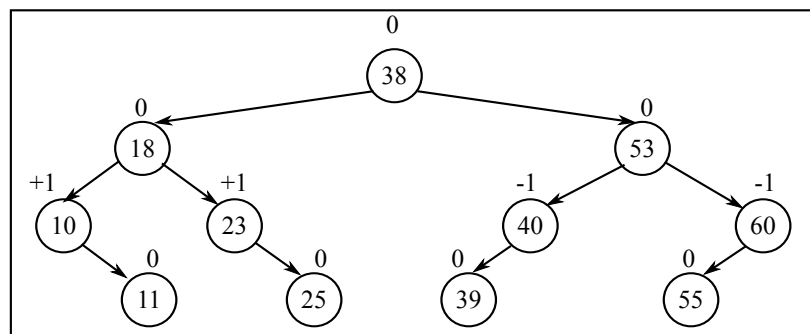


Figura 5.7: Arbore binar de căutare împreună cu factorii de balansare asociați nodurilor. Atenție: acesta NU este un arbore AVL.

Definiție - arbore AVL: Un arbore binar de căutare se numește *arbore AVL*, dacă fiecare nod al său este balansat.



Exemplu de arbore AVL. Deasupra fiecărui nod este indicat factorul de balansare al nodului respectiv.



Complexitatea operațiilor într-un arbore AVL

Complexitatea operațiilor într-un arbore AVL este determinată de înălțimea acestuia. Pentru a determina înălțimea maximă a unui arbore AVL care conține n noduri se notează cu h înălțimea acestuia și cu $N(h)$ numărul minim de noduri ale unui arbore AVL de înălțime h . Evident $n \geq N(h)$.

Factorul de balansare al rădăcinii în cazul unui AVL cu număr minim de noduri, de înălțime h este sigur diferit de 0 (altfel s-ar mai putea șterge noduri din subarborile stâng de ex. fără a modifica înălțimea, ceea ce ar contrazice numărul minim de noduri!). Atunci în mod sigur factorul de balansare al rădăcinii este -1 sau 1. Rezultă astfel că pentru un număr minim de noduri, unul dintre subarbori are înălțimea $h - 1$ iar celălalt are înălțimea $h - 2$.

$$N(h) = N(h - 1) + N(h - 2) + 1.$$

Cazurile de bază (considerând înălțimea unei frunze egală cu 1 și înălțimea pentru *nil* egală cu 0):

$$N(1) = 1$$

$$N(2) = 2$$

Evident: $N(h - 1) > N(h - 2)$, de unde rezultă $N(h) > 2N(h - 2) > 4N(h - 4) > \dots > 2^k N(h - 2k)$.

Ajungem la un caz elementar pentru $h - 2k = 2$ pentru h par - sau $h - 2k = 1$ pentru h impar.

De aici rezultă:

$N(h) > 2 * 2^{h/2}$ pentru h par sau $N(h) > 2 * 2^{(h-1)/2} = 2^{(h+1)/2}$ pentru h impar.

Cum $N(h) = \text{nr. minim de noduri pentru un AVL de înălțime } h \Rightarrow \text{numărul } n \text{ de noduri dintr-un AVL de înălțime } h \text{ respectă: } n \geq N(h) > 2^{h/2}$. Deci $h < 2 \log_2(n)$.

Deoarece complexitatea operațiilor de căutare, inserție și ștergere dintr-un arbore binar de căutare este $O(h)$ rezultă că într-un arbore AVL complexitatea acestor operații este $O(\log_2 n)$, dacă nu se ține cont de operațiile de reechilibrare ale arborelui. Aceste operații însă, au și ele complexitate logaritmică, așa cum se va vedea în continuare.

Prin operații de inserție/ștergere se poate produce o debalansare a anumitor noduri. Pentru refacerea proprietății de arbore AVL, se utilizează operații de **rotație**.

5.2.4 Operații într-un arbore AVL

Principalele operații într-un arbore AVL sunt aceleași ca în orice arbore binar de căutare: căutarea unei chei, inserția și ștergerea unei chei. Căutarea unei chei se realizează la fel ca în cazul unui arbore binar de căutare obișnuit. Operațiile de modificare a arborelui, respectiv inserția și ștergerea unei chei, dispun însă de un mecanism bazat pe rotații, care permite rebalansarea arborelui, astfel încât înălțimea să rămână de ordinul $\log_2 n$, unde n este numărul de noduri din arbore.

Inserția

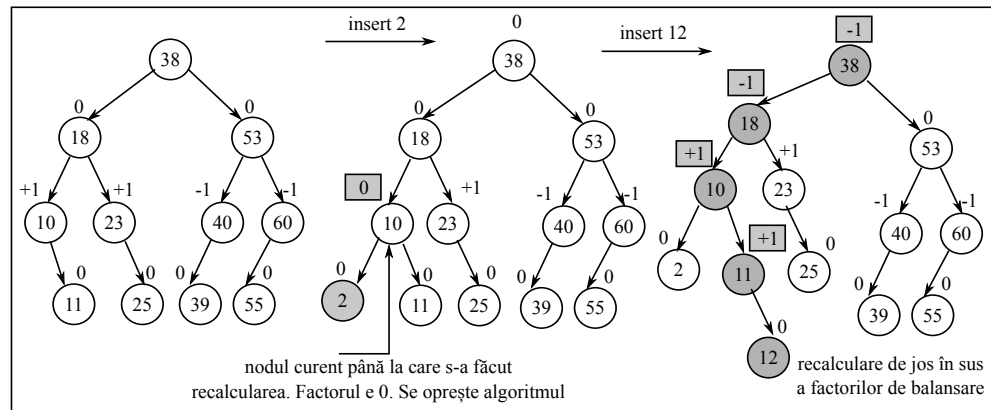
Inserarea unei noi chei într-un arbore AVL se realizează în primă instanță în același mod ca într-un arbore binar de căutare, adică se creează un nou nod x , cu cheia ce se dorește inserată. Acest nod x se inserează apoi la fel ca într-un arbore binar de căutare, devenind o frunză care are factorul de balansare 0. Apoi, pornind de la nodul x și urcând înspre rădăcină, se recalculază factorii de balansare.

În timpul recalculării factorilor de balansare, este posibil ca pe drumul de la x către rădăcină să se producă o debalansare a nodului curent. Acest lucru înseamnă că, după recalcularea fac-

torului de balansare a nodului curent, acest factor devine -2 sau 2 . În acest caz este necesară o operație de rebalansare.



Exemplu de recalculare a factorilor de balansare de la nodul inserat în sus.



Rebalansare: Nodul nou inserat are factorul de balansare 0. Recalcularea factorilor de balansare începe de la părintele nodului inserat înspre rădăcină.

Notăm cu x nodul curent. Se recalculează factorul de balansare al lui x .

- Dacă factorul de balansare nou este 0, atunci nu a crescut înălțimea subarborelui de rădăcină x , față de cât era înainte de inserție, doar s-a echilibrat. Din acest motiv, nu mai are sens continuarea urcării în arbore. Deci algoritmul se oprește!
- Dacă factorul de balansare nou este -1 sau $+1$, atunci s-a produs o creștere a înălțimii subarborelui de rădăcină x pe una dintre ramuri (stângă respectiv dreaptă). Acest lucru produce o modificare a factorului de balansare al părintelui mai sus, deci se continuă cu procesul de la nodul curent $x = x.parinte$.
- Dacă factorul de balansare nou al lui x este -2 sau $+2$, atunci s-a produs o debalansare a subarborelui de rădăcină x și sunt necesare proceduri de rebalansare, care vor fi prezentate mai jos. În urma acestor proceduri, înălțimea subarborelui curent va reveni la cea de dinainte de inserție (se va vedea în continuare) și de aceea nu mai este necesară continuarea rebalansării mai sus, deci algoritmul se oprește.

Cazurile de rebalansare

Cazul 1

- a) $fb(x) = -2$, înseamnă că subarborele stâng este prea înalt. Notăm cu $y = x.stanga$. Dacă $fb(y) = -1$ atunci rebalansarea se realizează printr-o rotație spre dreapta în jurul

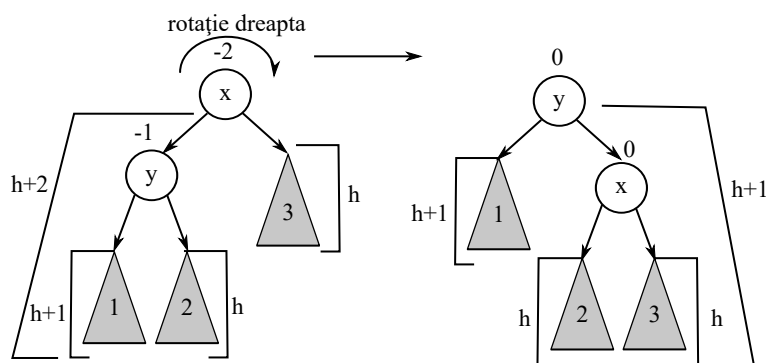


Figura 5.8: Rebalansarea arbore AVL în cazul 1 a).

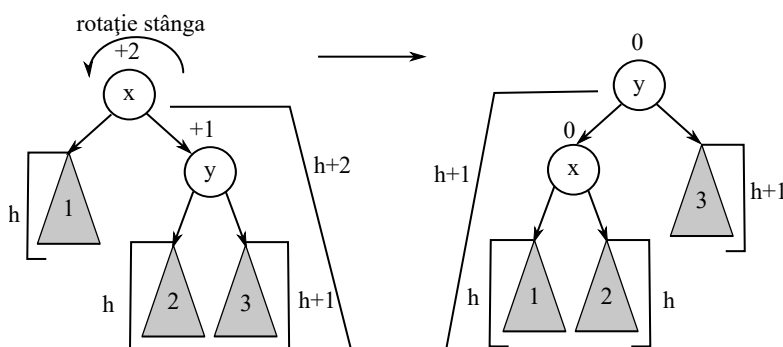


Figura 5.9: Rebalansarea arbore AVL în cazul 1 b).

lui x . Rezultatul acestei operații este ilustrat în figura 5.8. Se observă faptul că noii factori de balansare sunt 0 atât pentru x cât și pentru y . În plus, înainte de inserție, subarborii stâng al nodului x avea înălțimea $h + 1$, iar cel drept h . După rebalansare ambii subarbori au înălțimea $h + 1$. Rezultă că înălțimea subarborului care acum are rădăcina y nu a crescut, față de înălțimea înainte de inserție, când rădăcina era x . Astfel, nu este necesar să continuăm urcarea în arbore, deoarece mai sus nu vor exista modificări în factorul de balansare.

- b) $fb(x) = 2$, înseamnă că subarborii drept este prea înalt. Notăm $y = x.dreapta$. Dacă $fb(y) = 1$ atunci se efectuează o rotație la stânga în jurul lui x . Rezultatul acestei operații este ilustrat în figura 5.9. Observații similare cazului 1.a pot fi făcute și în acest caz.

Cazul 2

- a) $fb(x) = -2$, înseamnă că subarborii stâng este prea înalt. Notăm cu $y = x.stanga$. Presupunem că $fb(y) = 1$. Ce se întâmplă dacă se efectuează o rotație spre dreapta în jurul lui x ? Rezultatul unei astfel de rotații este ilustrat în figura 5.10.

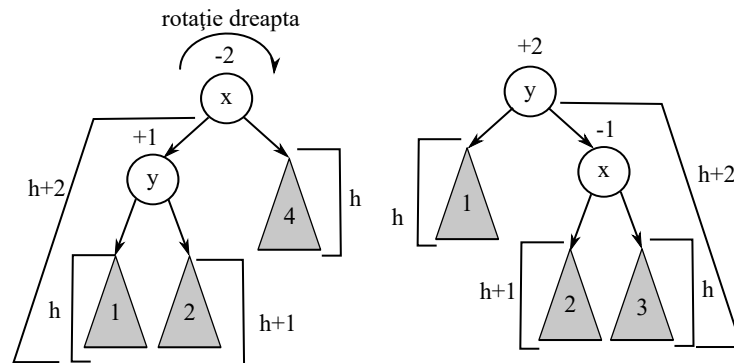


Figura 5.10: În cazul 2, o rotație simplă nu rezolvă problema debalansării arborelui

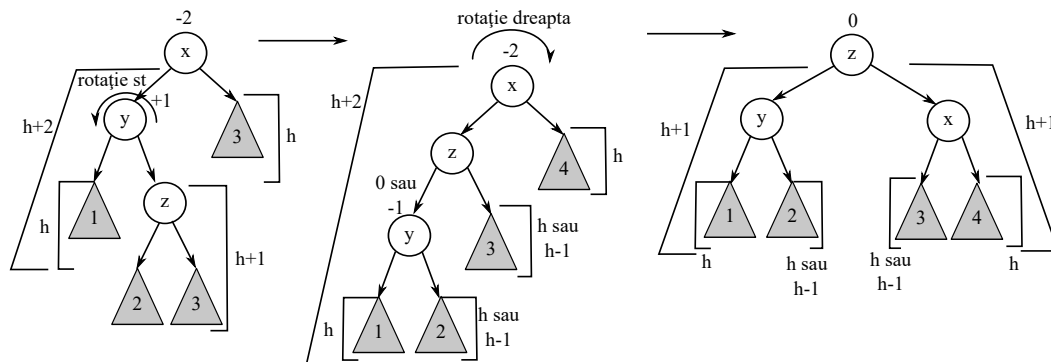


Figura 5.11: Rebalansarea arborelui în cazul 2.

Deci nu se rezolvă debalansarea, ci se mută pe cealaltă parte a arborelui. Soluția este următoarea:

- întâi rotație la stânga în jurul lui y
- apoi rotație la dreapta în jurul lui x .

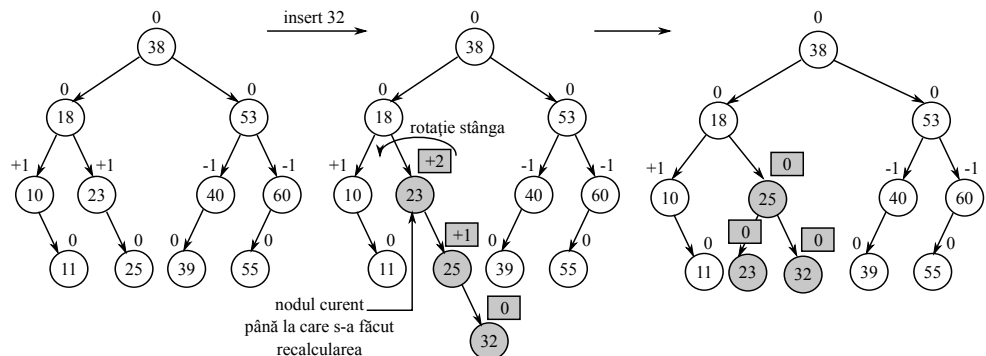
Se obține rezultatul din figura 5.11.

b) $fb(x) = 2$, înseamnă că subarboarele dreapta este prea înalt. Notăm cu $y = x.dreapta$. Presupunem că $fb(y) = -1$. Dacă efectuăm o rotație la stânga în jurul lui x se obține un efect similar ca la punctul a. Soluția este deci:

- întâi rotație la dreapta în jurul lui y
- apoi rotație la stânga în jurul lui x



Exemplu. Inserția nodului cu cheia 32 în arborele AVL din figură.



Inserați cheile 30, 25, 27, 0, 8, 11, 22, 20 într-un arbore AVL inițial vid.

Ștergerea unui nod

Ștergerea unui nod se realizează la fel ca pentru arborii binari de căutare obișnuți. Apoi se procedează la rebalansarea arborelui în modul următor. Dacă notăm cu z nodul care trebuie șters atunci:

- dacă z are cel mult un fiu, se pornește recalcularea factorilor de balansare de la părintele lui z
- dacă z are doi fii, atunci se determină y , succesorul nodului z , se înlocuiește nodul y cu fiul său drept, iar nodul z cu nodul y . În acest caz recalcularea factorilor de balansare începe de la părintele succesorului nodului șters.

Rebalansarea arborelui:

Notăm cu x nodul curent, pentru care se recalculează factorul de balansare. Spre deosebire de inserție, algoritmul nu se oprește atunci când factorul de balansare recalculat al nodului curent x este 0, deoarece în acest caz a scăzut înălțimea subarborelui de rădăcină x . Acest lucru, ilustrat în figura 5.12 sus, produce modificări ale factorului de balansare și la părintele său și deci continuă urcarea în arbore.

În schimb, atunci când factorul de balansare al nodului x devine -1 sau +1, înseamnă că doar s-a produs o scădere a înălțimii pe una dintre ramurile lui x , dar nu și a subarborelui de rădăcină x (figura 5.12 jos), deci algoritmul de rebalansare se poate opri.

De asemenea, algoritmul nu se oprește nici după prima rebalansare a unui nod cu factorul -2 sau +2, ci în cele mai multe cazuri trebuie să continue de la nodul curent la părinte până cel mult la rădăcină.

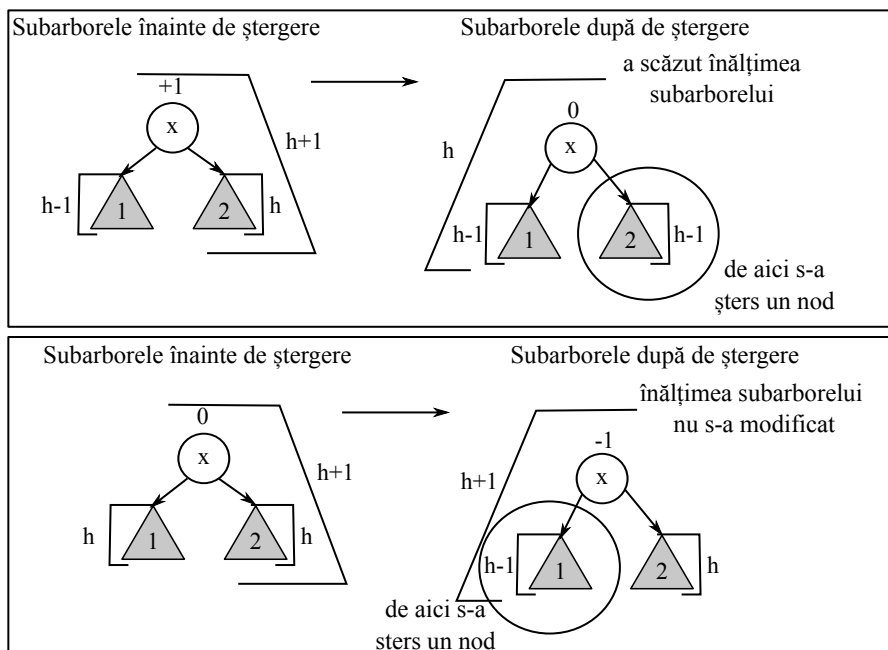


Figura 5.12: Modificarea înălțimii subarborelui curent după ștergerea unui nod.

Considerând nodul curent x , rebalansarea are următoarele cazuri:

Cazul 1

- a. Dacă x are factorul de balansare nou -2 și factorul lui $y = x.stanga$ este -1 , atunci se realizează o rotație la dreapta în jurul lui x . În figura 5.13 este ilustrat modul de recalculare a factorilor de balansare pentru nodurile x și y .

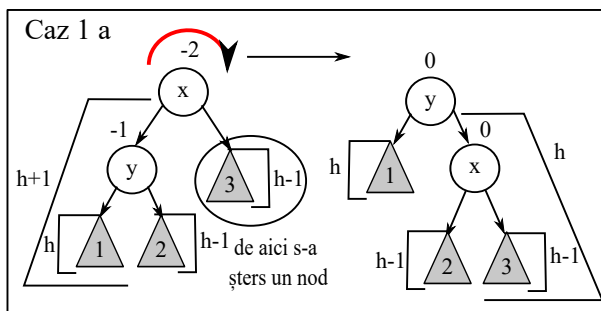


Figura 5.13: Rebalansarea unui arbore AVL după ștergerea unui nod, cazul 1a.

Se observă și următorul fapt important. Subarborele de rădăcină x a avut înainte de ștergere înălțimea $h + 2$. În cazul din figura 5.13, y a avut factorul de balansare -1 . Se observă din figură, că după rebalansare, înălțimea subarborelui care acum are rădăcina y , a scăzut cu 1, ceea ce produce eventuale debalansări mai sus în arbore, deci trebuie continuat la părintele lui x .

- b. Dacă x are factorul de balansare nou 2 și factorul de balansare al lui $y = x.dreapta$ este 1 atunci se efectuează o rotație la stânga în jurul lui x . În figura 5.14 este ilustrat modul de recalculare a factorilor de balansare pentru nodurile x și y .

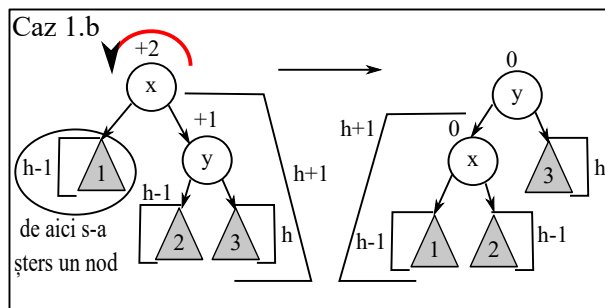


Figura 5.14: Rebalansarea unui arborele AVL după ștergerea unui nod, cazul 1b.

La fel ca în cazul 1a, se observă și următoarele: Subarboarele de rădăcină x a avut înainte de ștergere înălțimea $h + 2$, y a avut factorul de balansare $+1$. Se observă din figura 5.14, că după rebalansare, înălțimea subarboarelui care acum are rădăcina y , a scăzut cu 1, ceea ce produce eventuale debalansări mai sus în arbore, deci trebuie continuat la părintele lui x .

Cazul 2

- a. Dacă x are factorul de balansare nou -2 și nodul $y = x.stanga$ are factorul de balansare 1, atunci pentru rebalansare se efectuează:
- Întâi rotație la stânga în jurul lui y
 - Apoi rotație la dreapta în jurul lui x .
- b. Dacă x are factorul de balansare nou 2 și $y = x.dreapta$ are factorul de balansare -1 , atunci pentru rebalansare se efectuează:
- Întâi rotație la dreapta în jurul lui y
 - Apoi rotație la stânga în jurul lui x .

Justificarea necesității celor două rotații este similară cu cea în cazul inserției. Cele două situații sunt reprezentate în figura 5.15.

Cazul 3

- a. Dacă x are factorul de balansare nou -2 și factorul lui $y = x.stanga$ este 0, atunci refacerea balansării se realizează exact ca la cazul 1a prin rotație la dreapta în jurul lui x . În figura 5.16 este ilustrat modul de recalculare a factorilor de balansare pentru nodurile x și y .

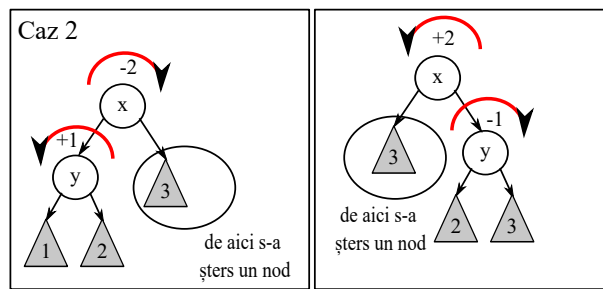


Figura 5.15: Rebalansarea unui arborele AVL după ștergerea unui nod, cazurile 2a și 2b.

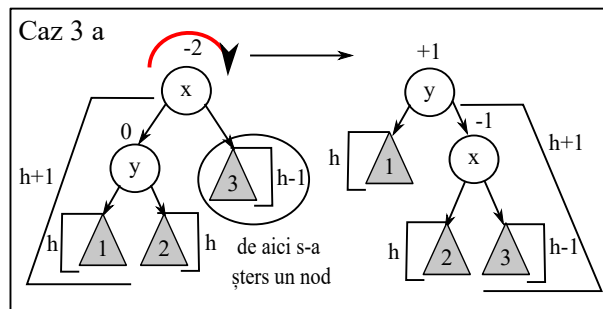


Figura 5.16: Rebalansarea unui arborele AVL după ștergerea unui nod, cazul 3a.

În cazul din figura 5.16, în care y a avut factorul de balansare 0 înainte de rebalansare, după rebalansare se observă că nu s-a modificat înălțimea subarborelui, care acum are rădăcina y , și deci nu mai este necesară continuarea urcării în arbore.

- b. Dacă x are factorul de balansare nou 2 și factorul de balansare al lui $y = x.dreapta$ este 0, atunci rotație la stânga în jurul lui x . În figura 5.17 este ilustrat modul de recalculare a factorilor de balansare pentru nodurile x și y .

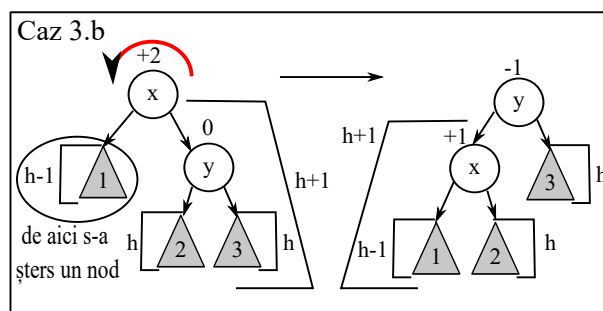


Figura 5.17: Rebalansarea unui arborele AVL după ștergerea unui nod, cazul 3b.

La fel ca în cazul 3a, se observă și următoarele: Subarborele de rădăcină x a avut înainte de ștergere înălțimea $h + 2$. În cazul în 3b, în care y a avut factorul de balansare 0 înainte de rebalansare, după rebalansare se observă că nu s-a modificat înălțimea subarborelui, care acum are rădăcina y , și deci nu mai este necesară continuarea urcării în arbore.

În cazurile 1 și 2, după rebalansare scade înălțimea subarborelui curent, comparativ cu înălțimea avută înainte de ștergere. Este deci necesară continuarea algoritmului de la părintele nodului curent x . În cazul 3 acest lucru nu se întâmplă și deci se poate opri urcarea în arbore. Variante de implementare a arborilor AVL pot fi găsite în bibliografie [3, 18].



Implementați un arbore AVL, în care fiecare nod are o proprietate suplimentară h , care reține înălțimea nodului și care se va folosi pentru calculul factorilor de balansare.



Să ne reamintim...

- Factorul de balansare al unui nod x este dat de diferența dintre înălțimea subarborelui său drept și cea a subarborelui său stâng.
- Un nod se numește balansat, dacă factorul său de balansare este -1, 0 sau 1.
- Un arbore este AVL, dacă este arbore binar de căutare și fiecare nod al său este balansat.
- Înălțimea unui arbore AVL este de ordinul $\log_2 n$, n numărul de noduri din arbore și deci complexitatea operațiilor de bază pe un arbore AVL este logaritmică în numărul de chei stocate..
- Un arbore AVL se rebalansează după operații de inserție și ștergere folosind operații de rotație.

5.2.5 Rezumat



În această secțiune au fost prezentați arborii AVL. A fost definită noțiunea de factor de balansare al unui nod și s-a demonstrat faptul că, arborii AVL au înălțimea de ordinul $\log_2 n$, unde n este numărul de chei stocate în arbore. De asemenea a fost prezentate operațiile de inserție și ștergere ale unui nod, împreună cu modul de rebalansare al arborelui în cazul acestor operații.



5.2.6 Test de autoevaluare

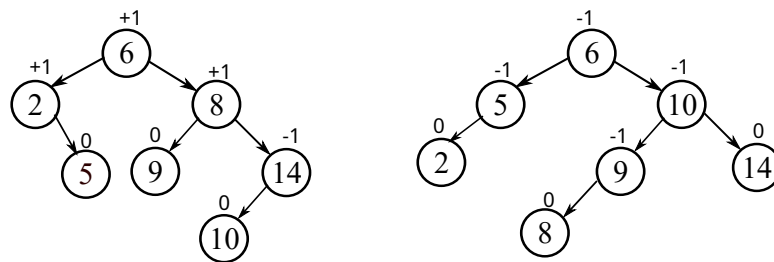
1. Desenați doi arbori AVL care conțin exact cheile 5, 10, 14, 6, 8, 2, 9. Specificați factorul de balansare pentru fiecare nod.

2. Inserați într-un AVL inițial vid valorile: 10, 5, 7, 20, 15, 25, 12, 8, 9, 17. Ștergeți apoi cheile 15, 25 și desenați arborele final.
3. Explicați de ce arborii AVL sunt mai echilibrați decât arborii binari de căutare simpli.
4. Scrieți un algoritm pseudocod pentru calcularea factorului de balansare al unui nod.
5. Scrieți un algoritm pseudocod care verifică dacă un arbore este un AVL corect definit.
6. Care este numărul maxim de rotații necesare pentru o singură operație de inserare?

5.2.7 Răspunsuri la testul de evaluare a cunoștințelor

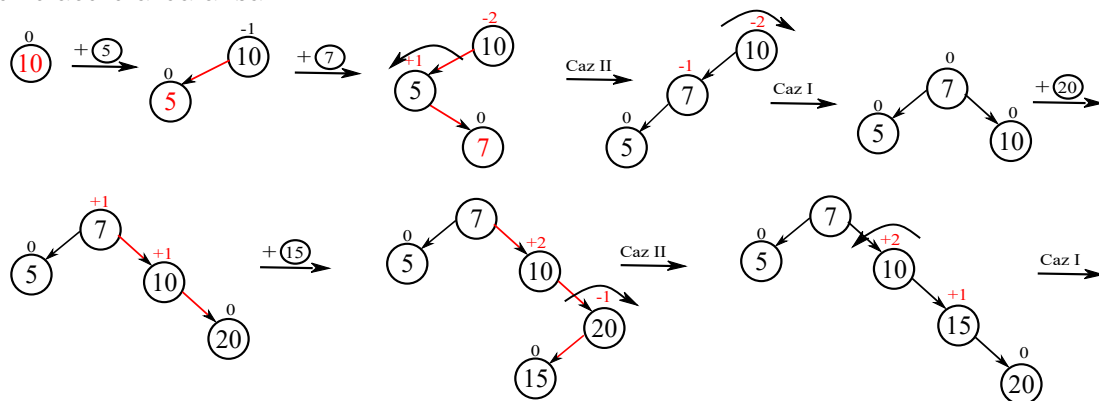
1. Desenați doi arbori AVL care conțin exact cheile 5, 10, 14, 6, 8, 2, 9. Specificați factorul de balansare pentru fiecare nod.

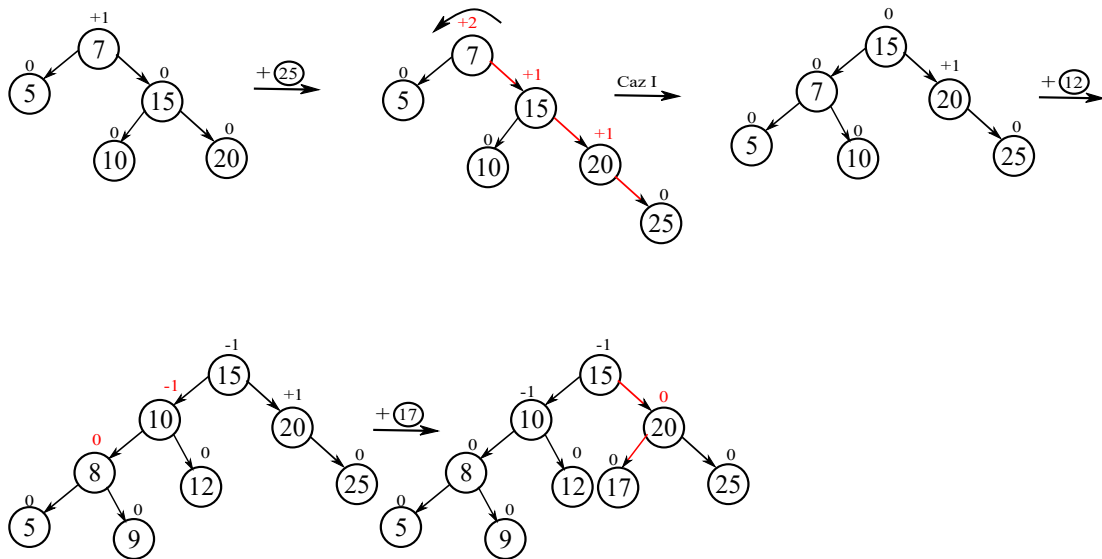
Rezolvare:



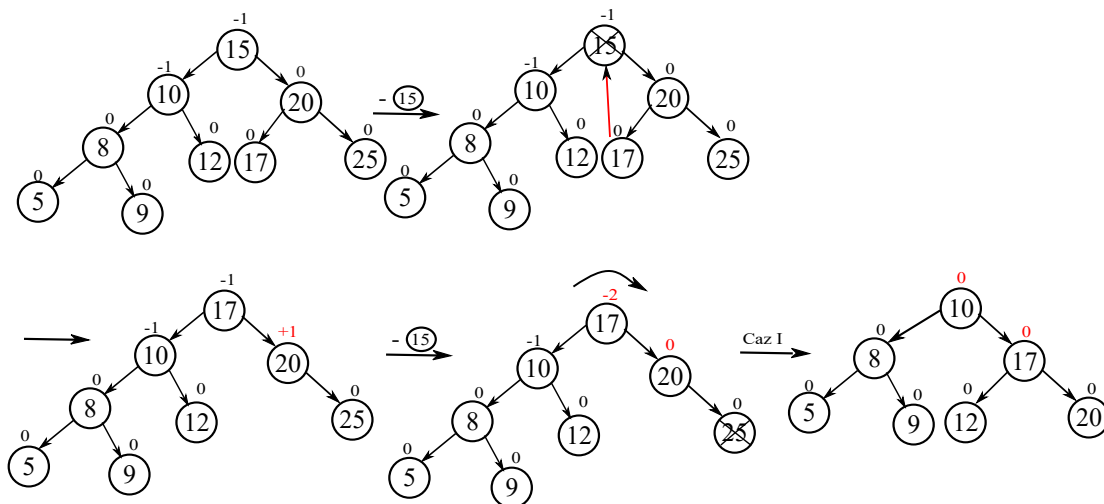
2. Inserați într-un AVL inițial vid valorile: 10, 5, 7, 20, 15, 25, 12, 8, 9, 17. Ștergeți apoi cheile 15, 25 și desenați arborele final.

Rezolvare: Inserția cheilor este prezentată în figurile următoare. Pentru fiecare inserție sunt marcați cu roșu factorii de balansare recalculați. De asemenea sunt indicate cazurile de refacere a balansării.





Arborele AVL obținut în urma inserțiilor este cel în care se realizează ștergerile, ilustrate în figurile următoare:



3. Explicați de ce arborii AVL sunt mai echilibrați decât arborii binari de căutare simpli.

Rezolvare: Într-un arbore binar de căutare simplu, inserarea succesivă a unor valori ordonate poate genera, în cel mai defavorabil caz, un arbore degenerat (asemănător unei liste), cu înălțimea egală cu $n - 1$ pentru n noduri.

Arborii AVL, în schimb, păstrează pentru fiecare nod diferența dintre înălțimile subarborilor stâng și drept (factorul de balansare) în intervalul $-1, 0, 1$. Această condiție asigură o distribuție relativ echilibrată a descendenților în jurul fiecărui nod. Drept urmare, înălțimea unui arbore AVL cu n noduri este de ordin $O(\log n)$, ceea ce garantează că operațiile de căutare, inserare și ștergere se realizează mult mai eficient decât în cazul

arborilor binari simpli.

4. Scrieți un algoritm pseudocod pentru calcularea factorului de balansare al unui nod.

Rezolvare: La prima vedere, problema ar putea fi rezolvată printr-o parcurgere în adâncime, calculând recursiv înălțimea subarborilor stâng și drept (ca în exercițiul 5 din capitolul despre arbori). Totuși, această metodă are complexitate liniară în numărul de noduri al subarborelui, ceea ce afectează complexitatea dorită de $O(\log n)$ pentru inserare și ștergere.

Pentru a păstra complexitatea logaritmică, este necesar ca fiecare nod să rețină fie factorul de balansare, fie, mai simplu, înălțimea nodului curent, care se poate actualiza ușor pe baza înălțimilor fiilor.

Algoritmul în pseudocod pentru această abordare actualizează înălțimea nodului curent și returnează noul factor de balansare.

Notă: La arborii AVL considerăm înălțimea unei frunze egală cu 1. Fiecare nod x are câmpurile: $x.parinte$, $x.stanga$, $x.dreapta$, $x.cheie$ și $x.h$ (înălțimea).

Algoritm: CALUL_FACTOR_BALANSARE

Input: Nodul *curent* pentru care se calculează factorul de balansare

Output: Factorul de balansare al arborelui

start

daca *curent* = nil **atunci**

| returneaza 0

sfarsit_daca

//se determină înălțimea pe stânga și pe dreapta

$h_s \leftarrow 0$

daca *curent.stanga* \neq nil **atunci**

| $h_s \leftarrow x.stanga.h$

sfarsit_daca

$h_d \leftarrow 0$

daca *curent.dreapta* \neq nil **atunci**

| $h_d \leftarrow x.dreapta.h$

sfarsit_daca

$curent.h \leftarrow MAX(h_s, h_d) + 1$

$factor_balansare \leftarrow (h_d - h_s)$

returneaza *factor_balansare*

stop

5. Scrieți un algoritm pseudocod care verifică dacă un arbore este un AVL corect definit.

Rezolvare: Această problemă se poate rezolva printr-o parcurgere în adâncime și vom verifica dacă se respectă proprietățile arborelui binar de căutare, precum și dacă valoarea factorului de balansare este -1, 0 sau 1. Vom considera cazul recursiv pentru simplitate și ne vom folosi de funcția rezolvată la exercițiul anterior.

Algorithm: VERIFICA_AVL

Input: Nodul *curent* în arborele binar *T*

start

daca *curent* = *nil* **atunci**
 | returneaza(*adevarat*)

sfarsit_daca

daca *curent.stanga* ≠ *nil* și *curent.cheie* < *curent.stanga.cheie* **atunci**
 | returneaza(*fals*)

sfarsit_daca

daca *curent.dreapta* ≠ *nil* și *curent.cheie* > *curent.dreapta.cheie* **atunci**
 | returneaza(*fals*)

sfarsit_daca

factor_balansare ← *CALUL_FACTOR_BALANSARE*(*curent*)

daca *factor_balansare* < -1 sau *factor_balansare* > 1 **atunci**
 | returneaza(*fals*)

sfarsit_daca

rezultat = *VERIFICA_BINAR_CAUT*(*curent.stanga*) și
VERIFICA_BINAR_CAUT(*curent.dreapta*)

returneaza(*rezultat*)

stop

6. Care este numărul maxim de rotații necesare pentru o singură operație de inserare?

Rezolvare: O singură inserare într-un arbore AVL poate necesita cel mult **2 rotații** (o rotație dublă: stânga-dreapta sau dreapta-stânga). În cazul cel mai simplu este suficientă o singură rotație, dar în unele situații dezechilibrul se corectează numai prin două rotații succesive. Dat fiind că după prima reechilibrare, algoritmul nu mai necesită urcarea în arbore, nu se va mai ajunge să se efectueze și alte rotații.

5.3 Unitatea de învățare 3 - Arbori roșu-negru



5.3.1 Introducere

Arborii roșu-negru, dezvoltati în 1987 de către Leonidas J. Guibas și Robert Sedgwick [8], sunt un al doilea tip de arbori binari de căutare auto-echilibrați. Spre deosebire de arborii AVL, care folosesc pentru reechilibrare factorul de balansare al nodurilor, arborii roșu-negru atribuie fiecărui nod o proprietate numită *culoare*, împreună cu o serie de alte proprietăți ale arborelui, care asigură o oarecare balansare, astfel încât complexitatea operațiilor în arbore să fie logaritmică în numărul de chei stocate.



5.3.2 Obiective

La sfârșitul acestei unități de învățare studenții vor înțelege:

- Ce este un arbore roșu-negru (ARN).
- Care este înălțimea unui ARN.
- Care sunt operațiile de bază într-un ARN.



Durata medie de studiu individual

Parcurgerea de către studenți a acestei unități de învățare se face în 3 ore.

Definiție: Un arbore roșu-negru (ARN) este un arbore binar de căutare în care fiecărui nod i se asociază o culoare - roșu sau negru - și care are următoarele proprietăți:

1. Fiecare nod este roșu sau negru - are deci un câmp suplimentar *culoare*
2. Rădăcina este neagră
3. Fiecare frunză este neagră și *nil*
4. Dacă un nod este roșu, ambii fii sunt negri \Rightarrow părintele unui nod roșu este negru.
5. Pentru fiecare nod x , pe oricare drum de la x la o frunză *nil* se întâlnește același număr de noduri negre (inclusiv frunza *nil* și exclusiv nodul de la care se pornește). Acest număr se numește înălțimea neagră a subarborelui de rădăcină x , notat cu *bh* - *black height*.

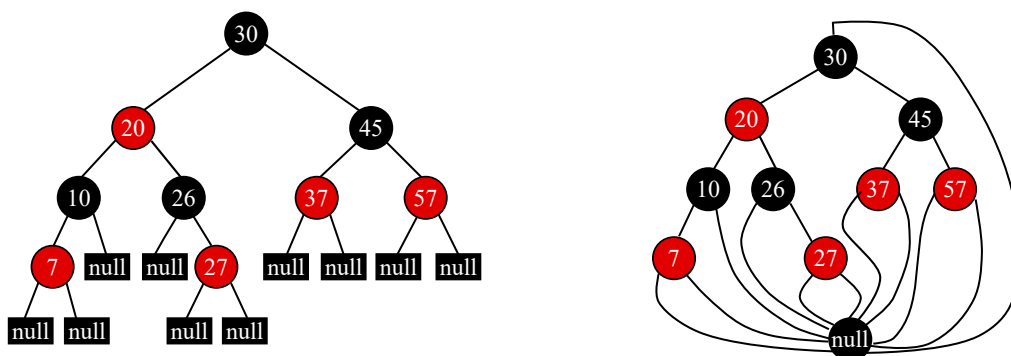
Observații:

- Într-un ARN niciun drum de la rădăcină la o frunză nu poate fi mai lung decât dublul unui alt drum la altă frunză. Acest lucru asigură o balansare relativă a arborelui .

- Înălțimea maximă a unui arbore ARN este $2 \log_2(n + 1)$.
Se demonstrează prin inducție că orice subarbore de rădăcină x conține cel puțin $2^{bh(x)} - 1$ noduri interne. Notând cu h înălțimea arborelui și cu r rădăcina, din proprietatea 4 se obține $bh(r) \geq h/2$. Dar $n \geq 2^{bh(r)} - 1 \geq 2^{h/2} - 1$ de unde rezultă $h \leq 2 \log_2(n + 1)$
- Definirea unui nod *nil* pentru fiecare frunză presupune un consum inutil de memorie. Din acest motiv se poate considera în locul acestor frunze un singur nod santinelă $T.nil$ către care să indice acele noduri interne care au ca fii frunze *nil*. Pentru simplitate în continuare vom ignora în desene nodurile *nil*.
- Datorită faptului că operațiile de căutare, maxim, minim, succesor, predecesor depind de înălțimea h a arborelui, înseamnă că aceste operații au complexitatea $O(\log_2 n)$.



Exemplu de arbore roșu - negru cu înălțimea neagră $bh = 2$ și înălțimea $h = 4$. În dreapta exemplul cu un singur nod santinelă.



Prin operații de inserție/ștergere se pot strica anumite proprietăți RN. Pentru refacerea acestor proprietăți de arbore ARN, sunt necesare operații de recolorare și de **rotație**.

5.3.3 Inserția într-un arbore roșu-negru

Inserția propriu-zisă a unui nod într-un arbore roșu-negru se realizează după același algoritm ca și inserția într-un arbore binar de căutare. Practic se pornește de la rădăcină și se compară la fiecare pas cheia nodului curent x cu cheia nodului z care se inserează, coborându-se în subarboarele stâng dacă $z.cheie < x.cheie$ și în cel drept altfel și se refac proprietățile ARN. Nodul care se inserează are culoarea roșie, iar în structura de tip arbore T se consideră câmpul santinelă $T.nil$.

Proprietățile care pot fi neîndeplinite în cazul inserției:

- Proprietățile 1, 3 și 5 se păstrează, datorită faptului că se inserează un nod roșu, care are ca fii două frunze *nil*, deci se leagă de nodul santinelă $T.nil$
- Proprietatea 2 poate fi încălcată dacă nodul inserat este chiar rădăcina sau ca urmare a cazului 1, care va fi discutat în cele ce urmează. Pentru rezolvarea acestei situații este suficientă colorarea rădăcinii cu negru.
- Proprietatea 4 poate fi încălcată, dacă părintele de care s-a legat noul nod are culoarea roșie. În acest caz trebuie refăcută această proprietate. Există în această situație 3 cazuri care vor fi discutate în continuare.

Notăm cu z nodul inserat și cu P părintele lui z . Datorită faptului că P are deja culoarea roșie înainte de inserție, iar inserția se produce într-un arbore roșu-negru valid, înseamnă că P nu este rădăcină și deci există un nod părinte al lui $P \neq T.nil$ pe care îl notăm cu B (bunic). Notăm cu U unchiul lui z (fratele părintelui).

Nodul P se poate afla la stânga sau la dreapta lui B . Cele două cazuri se tratează în mod similar, prin simetrie. Vom considera în continuare inserția pe stânga bunicului B .

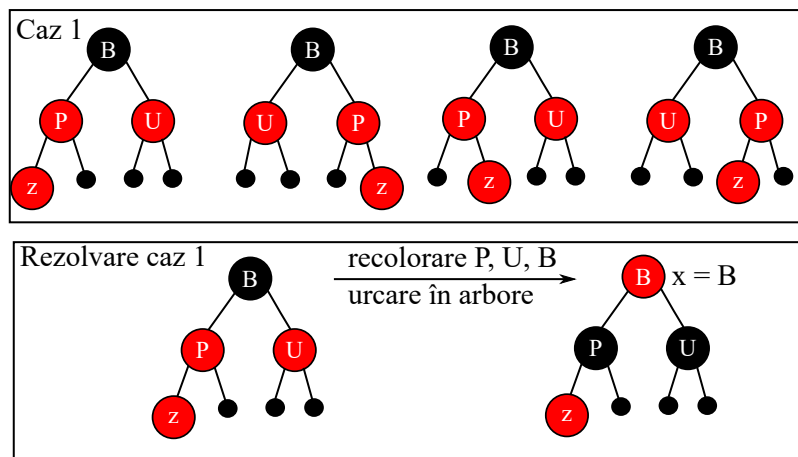


Figura 5.18: Cazul 1 pentru inserție.

Pentru refacerea proprietății 4 după inserarea unui nod nou se iau în considerare trei cazuri.

Cazu 1: unchiul lui z este roșu (figura 5.18).

Deoarece U, P sunt roșii rezultă că B are culoarea neagră, altfel s-ar contrazice proprietatea 4, deci este suficientă recolorarea P, U, B , adică P și U devin negri iar B roșu.

În urma acestei modificări poate avea loc o contrazicere a proprietății 4 pentru B și părintele său, deci se reia procedura de refacere a proprietăților roșu-negru, considerând de data aceasta $z = B$. Același procedeu se aplică și în cazul în care z se află pe dreapta lui P .

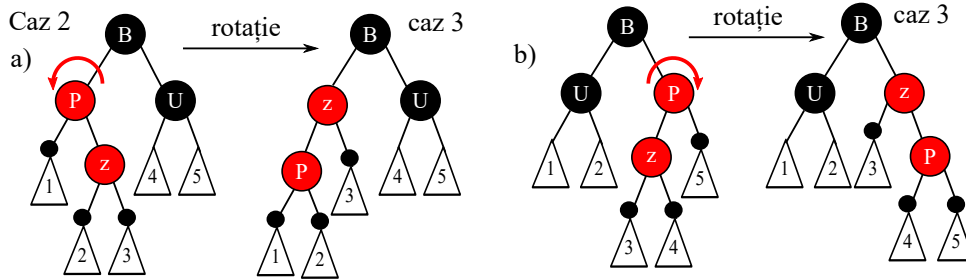


Figura 5.19: Cazul 2 pentru inserție.

Atunci când unchiul U este negru se disting cazurile 2 și 3, care diferă prin poziționarea nodului z relativ la P și la B .

Cazul 2: unchiul este negru și

- a) z se află pe dreapta lui P , iar P se află la stânga lui B - figura 5.19 (a);
- b) z se află pe stânga lui P , iar P se află la dreapta lui B - figura 5.19 (b).

Soluționare: în cazul a) rotație la stânga după P , în cazul b) rotație la dreapta după P . Nu se soluționează complet, ci se trece practic în cazul 3, în care refacerea începe de la nodul care prin rotație a coborât, deci de la $z = P$.

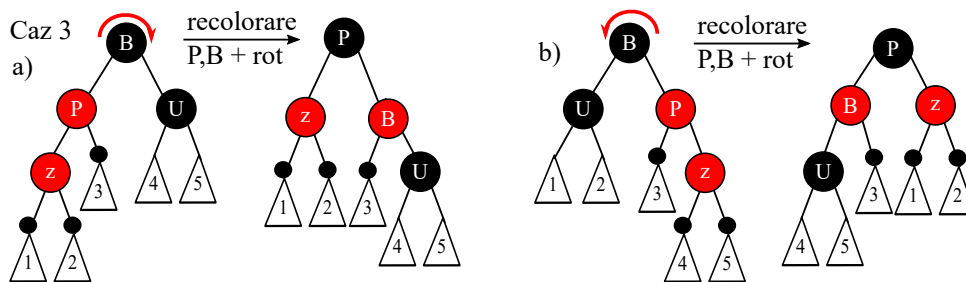


Figura 5.20: Cazul 3 pentru inserție.

Cazul 3: unchiul este negru și

- a) z se află pe stânga lui P , iar P se află la stânga lui B - figura 5.20 (a);
- b) z se află pe dreapta lui P , iar P se află la dreapta lui B - figura 5.20 (b).

Soluționare: se colorează B cu roșu și P cu negru. Apoi pentru situația a) rotație la dreapta în jurul lui B , iar pentru situația b) rotație la stânga în jurul lui B .

Algoritm: ARN_INSEREAZA

Input: Un arbore roșu-negru T , cheia de inserat val **start** $y \leftarrow T.nil$ $x \leftarrow T.rad$ **cat_timp** $x \neq T.nil$ și $x.cheie \neq val$ **executa** $y \leftarrow x$ **daca** $val < x.cheie$ **atunci**| $x \leftarrow x.stanga$ **sfarsit_daca****altfel**| $x \leftarrow x.dreapta$ **sfarsit_daca****sfarsit_cat_timp****daca** $x \neq T.nil$ **atunci**

| scrie("cheia deja exista")

| **return****sfarsit_daca**alocă memorie pentru nodul z $z.cheie \leftarrow val$ $z.stanga \leftarrow T.nil$ $z.dreapta \leftarrow T.nil$ $z.culoare \leftarrow rosu$ $z.parinte \leftarrow y$ **daca** $y = T.nil$ **atunci**| $T.rad \leftarrow z$ **sfarsit_daca****altfel****daca** $val < y.cheie$ **atunci**| $y.stanga \leftarrow z$ **sfarsit_daca****altfel**| $y.dreapta \leftarrow z$ **sfarsit_daca****sfarsit_daca**ARN_INSERT_REPARA(T, z)**stop**

Algoritm: ARN_INSERT_REPARA**Input:** Un arbore roșu-negru T , nodul inserat z **start** $P \leftarrow z.parinte$ //părintele lui z **cat_timp** $P.culoare = rosu$ **executa** $B \leftarrow P.parinte$ //bunicul lui z **daca** $P = B.stanga$ **atunci** $U \leftarrow B.dreapta$

_____ caz 1

daca $U.culoare = rosu$ **atunci** $P.culoare \leftarrow negru$ $U.culoare \leftarrow negru$ $B.culoare \leftarrow rosu$ $z \leftarrow B$ **sfarsit_daca****altfel**

_____ caz 2

daca $z = P.dreapta$ **atunci** $z \leftarrow P$ $ROTATIE - STANGA(T, z)$ **sfarsit_daca**

_____ caz 3

 $P.culoare \leftarrow negru$ $B.culoare \leftarrow rosu$ $ROTATIE - DREAPTA(T, B)$ **sfarsit_daca****sfarsit_daca****altfel** $//similar dar simetric pentru părintele lui $z$$ $//pe partea dreaptă a bunicului lui $z$$ **sfarsit_daca****sfarsit_cat_timp** $T.rad.culoare \leftarrow negru$ **stop**

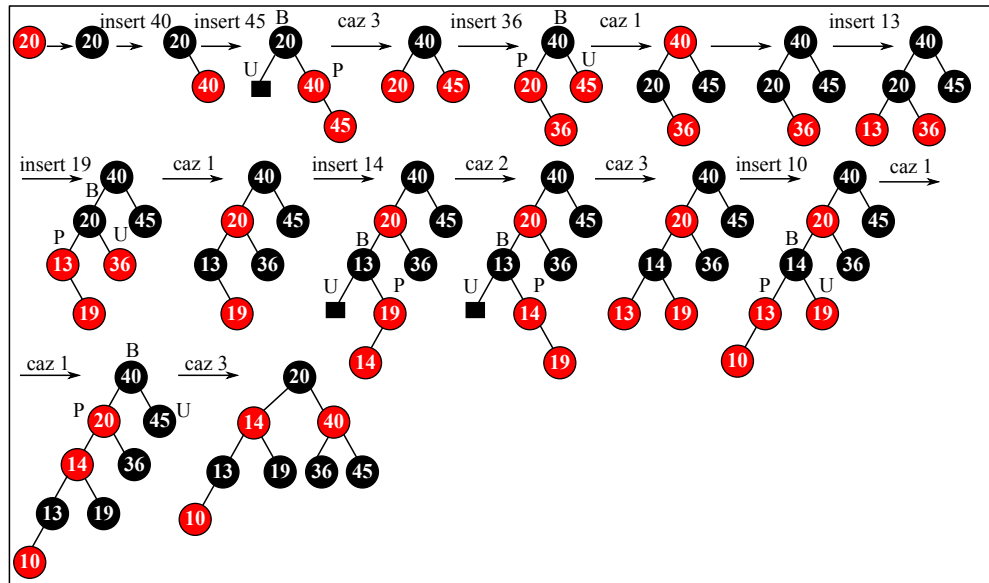
Complexitate: Inserția are aceeași complexitate ca în cazul arborilor binari de căutare simpli, deci $O(h)$. Cum înălțimea este cel mult $2 \log_2(n + 1)$ rezultă o complexitate $O(\log_2 n)$, iar algoritmul de refacere al proprietăților de arbore roșu-negru pornește de jos înspre rădăcină pe o ramură, deci complexitatea este tot $O(\log_2 n)$. Rezultă complexitatea pentru inserție este

$O(\log_2 n)$.

Operația de inserție este descrisă extensiv în bibliografie, iar algoritmi prezentați au la bază algoritmi din [4]. În algoritm se verifică și unicitatea cheii. O cheie deja existentă nu se mai introduce.



Exemplu. Inserarea cheilor 20, 40, 45, 36, 13, 19, 14, 10 într-un ARN inițial vid.



Dați exemplu de un ARN cu înălțimea neagră $bh = 2$, care are exact 3 noduri roșii și toate cheile mai mici decât 15 și inserați în acesta cheile 23, 22, 19, 25.

5.3.4 Ștergerea dintr-un arbore roșu-negru

Operația de ștergere dintr-un arbore roșu-negru este mai complicată decât operația de inserție. În prima etapă, operația de ștergere are la bază ștergerea dintr-un arbore binar de căutare, cu câteva modificări, după care trebuie efectuată o operație de refacere a proprietăților de arbore roșu-negru. Reamintim faptul că, în cazul ștergerii unui nod z dintr-un arbore binar de căutare T se iau în considerare cazurile:

1. $z.stanga = T.nil$ - figura 5.21 a)
2. $z.dreapta = T.nil$ - figura 5.21 b)
3. z are doi fii diferiți de $T.nil$ și $y = BIN_SUCCESOR(z)$
 - a. y fiu al lui z - figura 5.21 c)
 - b. y nu fiu al lui z , ci este undeva mai jos în subarborele drept al lui z - figura 5.21 d)

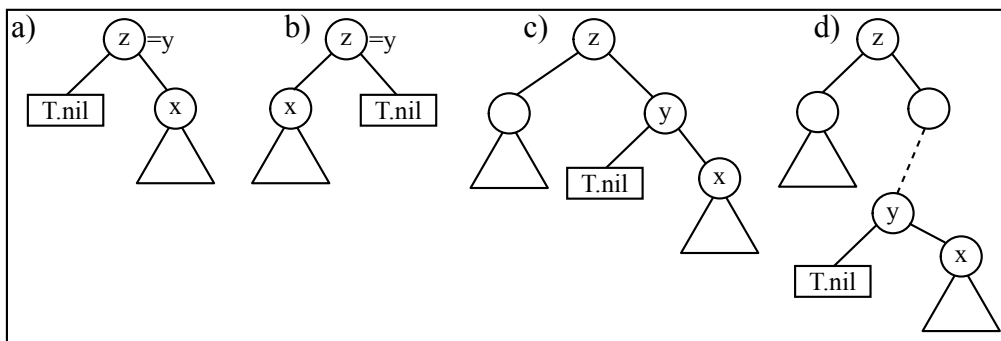


Figura 5.21: Cazurile pentru ștergerea nodului z . a) z are la stânga frunza $T.nil$, b) z are la dreapta frunza $T.nil$, c) z are 2 fii diferiți de $T.nil$ și succesorul lui z este fiu al lui z , d) z are doi fii diferiți de $T.nil$ și succesorul lui z nu este fiu al lui z .

Observații:

- Algoritmul de refacere depinde în cazurile 1 și 2 de culoarea nodului șters z și de culoarea pe care o are x , iar în cazul 3 de culoarea originală a lui y (după ștergere y preia culoarea lui z) și de culoarea lui x , unde x este în primele 2 cazuri nodul cu care se înlocuiește z , iar în al treilea caz este fiul drept al lui y .
- În cazurile 1 și 2, prin ștergerea nodului z și înlocuirea cu unul dintre descendenții direcți, se poate produce o modificare a culorii în poziția deținută anterior de z , ceea ce poate duce la violarea proprietăților de arbore roșu-negru. La fel în cazul 3, la înlocuirea lui y cu x .
- Nodul x este acela care se deplasează în poziția deținută anterior de nodul y . Se observă faptul că, x poate fi santinela $T.nil$

Refacerea proprietăților de arbore roșu-negru:

- Dacă nodul z în cazurile 1 și 2, respectiv y în cazul 3, a avut culoarea inițială roșu, atunci se păstrează proprietățile de arbore RN, deoarece
 1. Nu s-a modificat înălțimea neagră pe nici o ramură prin eliminarea unui nod roșu
 2. Nici un nod roșu nu a căpătat un fiu roșu.
 3. Deoarece nici nodul z , nici nodul y în cazul 3, nu au fost rădăcină (culoarea originală a fost roșie) nu s-a modificat nici culoarea rădăcinii, care a rămas neagră
- Rezultă deci că, proprietățile de arbore RN sunt violate doar dacă a fost eliminat un nod negru. Există mai multe cazuri care trebuie tratate.

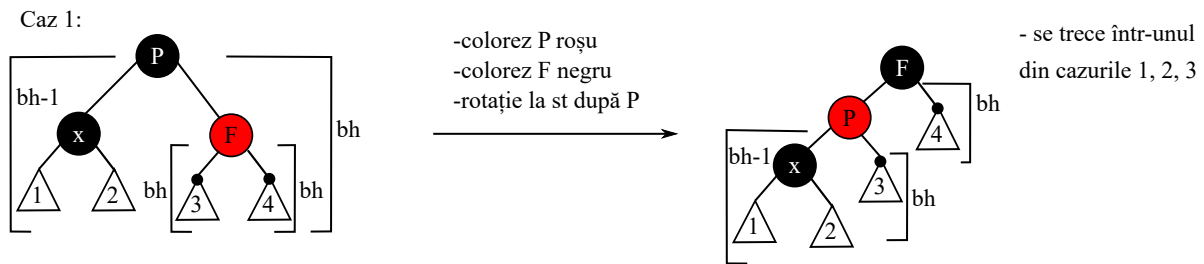


Figura 5.22: Cazul 1 în ARN

- Prin eliminarea unui nod negru din arbore
 1. În primul rând se micșorează înălțimea neagră pe ramura respectivă
 2. Poate apărea vecinătate între două noduri roșii

Cazul 0: culoarea nodului x cu care s-a făcut înlocuirea este roșie. În acest caz singurul lucru care trebuie făcut este recolorarea lui x în negru. Acest lucru rezolvă atât problema (1) cât și problema (2).

Notății: P =părintele lui x , F =fratele lui x , bh = înălțimea neagră originală a subarborelui cu rădăcina P .

În continuare vom considera că nodul x se află la stânga nodului P . Pentru x la dreapta lui P modul de rezolvare este simetric. În toate figurile care ilustrează cazurile de reechilibrare, pentru fiecare nod este indicat numărul de noduri negre aflate pe orice drum către o frunză *nil* din subarboarele respectiv, luând în considerare ramura marcată. Dacă nodul marcat face parte din segmentul considerat, numărarea începe de la el; în caz contrar, numărarea pornește de la fiul său de partea respectivă.

Cazul 1: F are culoarea roșie - figura 5.22.

Facem următoarele observații:

- Datorită faptului că înainte de ștergere arborele era RN valid, din F roșu și F descendent direct al lui P rezultă culoarea lui P este neagră.
- Copiii lui F sunt ambii negri (eventual $T.nil$)
- Prin ștergere s-a micșorat înălțimea neagră pe partea stângă a lui P , adică numărul de noduri negre pe oricare drum de la P spre frunzele din subarboarele drepte este bh , iar spre

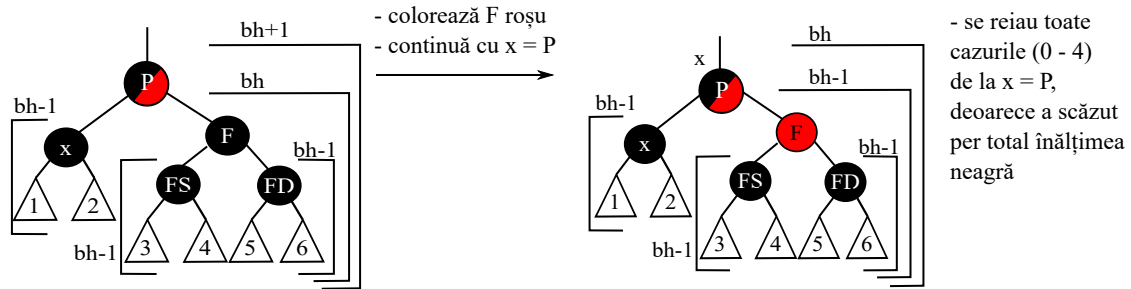


Figura 5.23: Cazul 2 în ARN

frunzele din subarborele stâng este $bh - 1$. Cum F este roșu rezultă că nu contribuie la acest număr de noduri negre. Astfel pe oricare parte a lui F se întâlnesc bh noduri negre în drum spre o frunză *nil* (vezi figura 5.22).

Refacerea proprietăților de arbore RN:

- Colorare P cu roșu și colorare F cu negru.
- Rotație la stânga în jurul lui P . (Dacă x se află pe dreapta lui P , atunci rotația este la dreapta).

În acest caz, F acum negru, urcă în locul lui P . La dreapta lui F înălțimea neagră a rămas bh . Înălțimea neagră pe dreapta lui P este bh , iar înălțimea neagră pe stânga al lui P este $bh - 1$. Deci problema în nodul P a rămas, dar fiul drept al lui P nu mai este roșu ci negru, ceea ce conduce la unul dintre cazurile 2 - 4. Deci din cazul 1 se trece la cazul 2, 3 sau 4! În figura 5.22 este ilustrată procedura de refacere a proprietăților RN în cazul 1.

Cazul 2: F este negru și ambii fii ai săi sunt negri (eventual $T.nil$) - figura 5.23.

Observații:

- $F \neq T.nil$, pentru că pe partea stângă a lui P s-a șters un nod negru, iar înainte de ștergere pe ambele părți ale lui P înălțimea neagră era aceeași și cel puțin 1.
- P poate fi roșu sau negru
- Prin ștergere s-a micșorat înălțimea neagră pe partea stângă a lui P , adică pe partea dreaptă a lui P se întâlnesc bh noduri negre către orice frunză, iar pe partea stângă $bh - 1$. Cum F este negru rezultă că pe pentru ambii săi fii drumurile către o frunză conțin $bh - 1$ noduri negre (atenție, numărând și nodurile FS și FD!).

Refacerea proprietăților de arbore RN:

- Se colorează F roșu \Rightarrow înălțimea neagră a subarborelui drept al lui P este $bh - 1$, deci egală cu cea a subarborelui stâng, dar înălțimea neagră a arborelui P este mai mică decât înainte de ștergere, deci cu o unitate mai mică decât a fratelui său (dacă există). Această problemă se rezolvă reluând cazurile re refacere pornind de la $x = P$ în sus.

Cazul 3: a) F are culoarea neagră, fiul stâng al lui F notat cu FS este roșu, iar cel drept notat cu FD este negru. (figura 5.24). Respectiv cazul 3b) simetric, atunci când F este pe stânga lui P și x pe dreapta, F este negru, FD este roșu și FS este negru. Cazul 3b) se soluționează simetric cu 3a) prezentat mai jos.

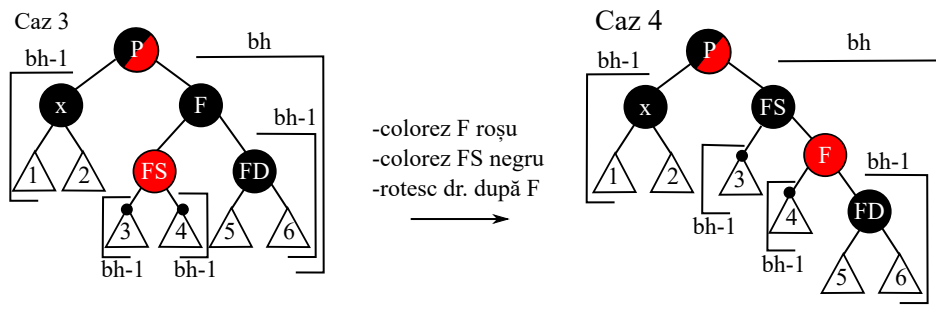


Figura 5.24: Cazul 3 în ARN. P poate avea culoarea roșie sau culoarea neagră.

Observații:

- Culoarea lui P poate fi roșie sau neagră.
- F sigur este diferit de $T.nil$, pentru că pe partea stângă a lui P s-a șters un nod negru, iar înainte de ștergere pe ambele părți ale lui P înălțimea neagră era aceeași și cel puțin 1.
- Numărul de noduri negre ce se întâlnesc pornind din P (exclusiv P) pe drumul spre orice frunză din subarboarele stâng este $bh - 1$, iar spre orice frunză din subarborelui drept este bh . Subarboarele F are înălțimea neagră $bh - 1$ (figura 5.24).

Refacerea proprietăților de arbore RN: - figura 5.24

- Recolorare F și $FS \Rightarrow F$ devine roșu și FS devine negru.
- Rotație la dreapta în jurul lui $F \Rightarrow FS$ urcă în locul lui F .

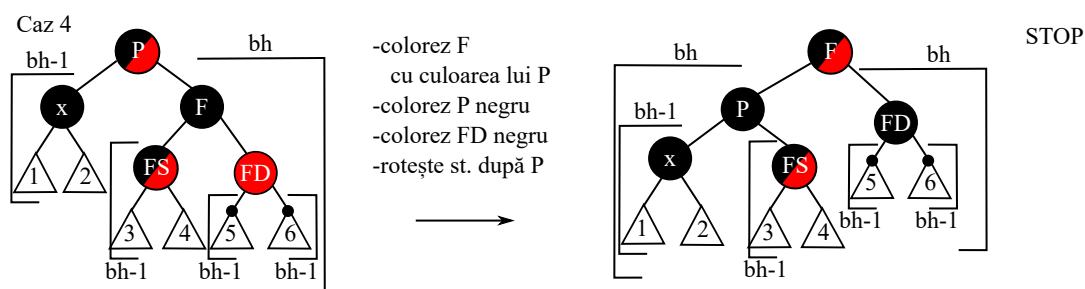


Figura 5.25: Cazul 4 în ARN

Problema la care am ajuns în continuare este refacerea proprietăților RN pentru cazul 4. În cazul în care x este pe dreapta lui P rezolvarea este simetrică.

Cazul 4: F are culoarea neagră iar fiul drept notat cu FD este roșu. Fiul stâng, FS , poate avea oricare dintre cele două culori. (figura 5.25). În cazul simetric, atunci când x este pe dreapta lui P , FS este roșu.

Observații:

- Culoarea lui P poate fi roșu sau negru.
- F sigur este diferit de $T.nil$, pentru că pe partea stângă a lui P s-a șters un nod negru, iar înainte de ștergere pe ambele părți ale lui P înălțimea neagră era aceeași și cel puțin 1.
- Pe orice drum din stânga lui P către o frunză întâlnim $bh - 1$ noduri negre, iar pe dreapta bh . Pe oricare parte a lui F , pe drumurile către o frunză se întâlnesc $bh - 1$ noduri negre (figura 5.25).

Refacerea proprietăților de arbore RN: - figura 5.25.

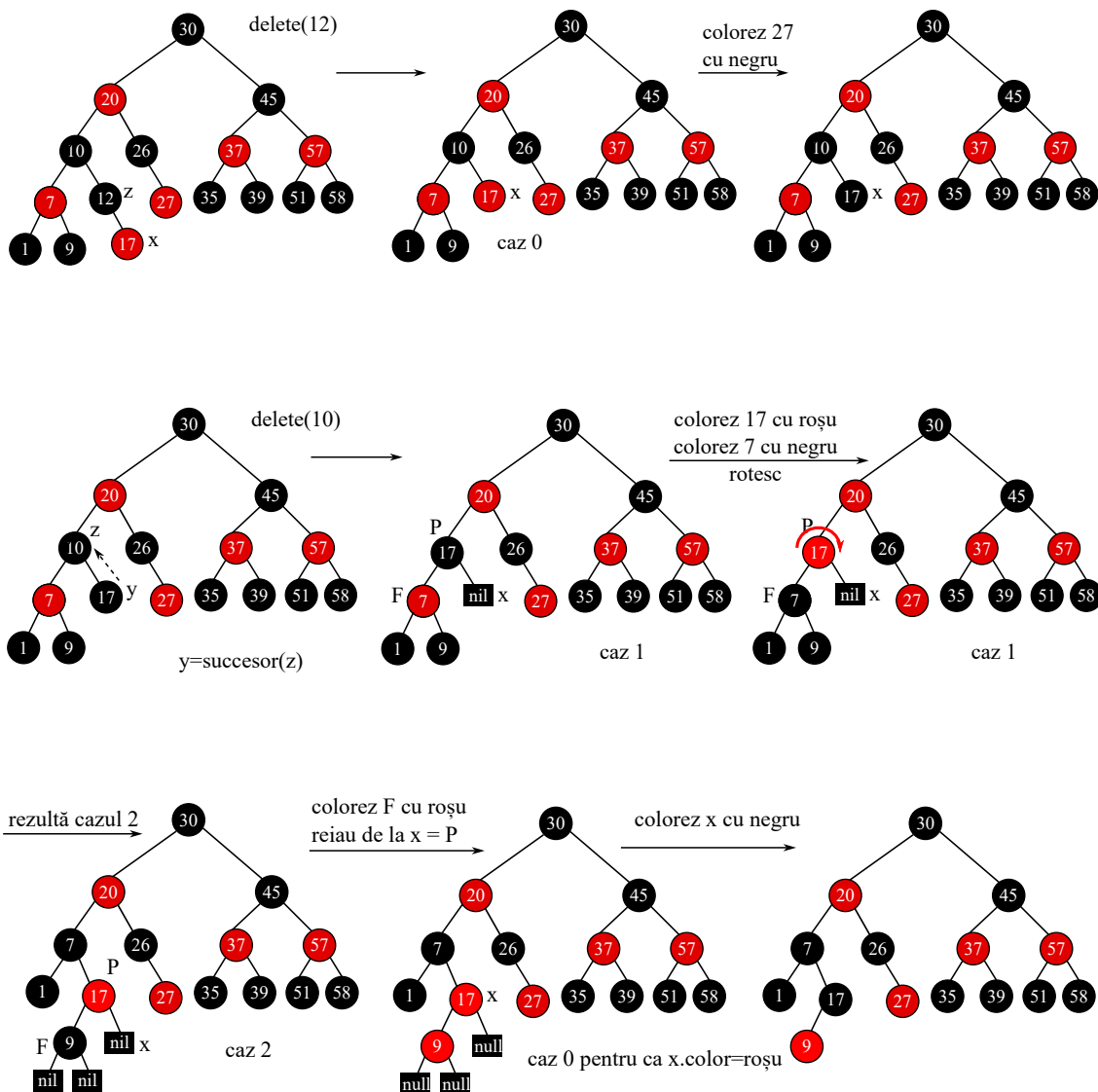
- Colorare F cu culoarea lui P
- Colorare P cu negru
- Colorare FD cu negru
- Rotație la stânga în jurul lui P (respectiv dreapta în cazul simetric).

Se observă în figură reechilibrarea arborelui care duce la oprirea procedurii de refacere a proprietăților ARN.

pseudocodul pentru ștergere poate fi găsit în bibliografia recomandată [4]. Idei de implementare pot fi găsite în [16].



Exemplu. Ștergerea cheilor: 12 și 10 din arborele roșu-negru din figură.



Arborii roșu-negru se pot folosi cu succes pentru implementarea de dicționare în care elementele se păstrează sortate după cheie. Un astfel de exemplu sunt structurile de tip *set* și *map* din STL, în care accesul la elemente se realizează pe baza cheii acestuia.



Să ne reamintim...

- Un arbore roșu-negru este un arbore binar de căutare în care fiecărui nod i se atribuie o proprietate numită culoare.
- Rădăcina unui arbore roșu-negru este întotdeauna neagră, iar frunzele sunt negre și *nil*.
- Orice nod roșu are doi fii negri, deci un nod roșu are un părinte negru.
- Pentru orice nod x , pe orice drum de la x către o frunză se întâlnesc același număr de noduri negre, exclusiv nodul x și inclusiv frunza. Acest număr de numește înălțime neagră.
- În urma inserției sau ștergerii într-un ARN se pot strica proprietățile de mai sus, de aceea sunt necesare operații de rotație și recolorare, care să refacă aceste proprietăți.
- Complexitatea operațiilor de căutare, inserție și ștergere a unei chei se realizează în complexitate logaritmică, în raport cu numărul de chei stocate.

5.3.5 Rezumat



În această secțiune au fost prezentați arborii RN. A fost definită noțiunea de înălțime neagră al unui nod și s-a demonstrat faptul că, un ARN are înălțimea de ordinul $\log_2 n$, unde n este numărul de chei stocate în arbore. De asemenea a fost prezentate operațiile de inserție și ștergere ale unui nod, împreună cu modul de rebalansare al arborelui în cazul acestor operații.



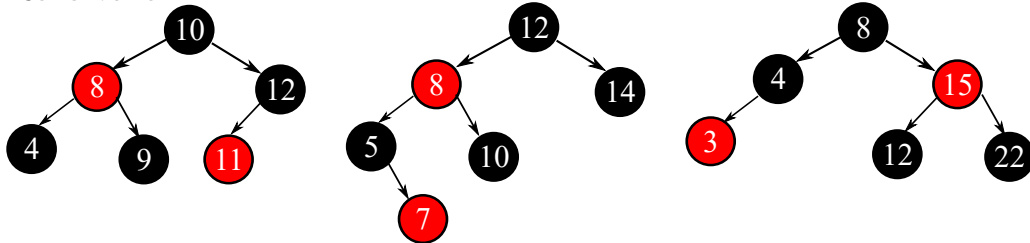
5.3.6 Test de autoevaluare

1. Dați 3 exemple de arbori roșu-negru care conțin exact 2 noduri roșii și 4 noduri negre.
2. Inserați într-un arbore roșu-negru inițial vid cheile: 7, 3, 18, 10, 13, 8, 11, 12.
3. Explicați de ce este important ca orice drum de la rădăcină la oricare frunză să conțină același număr de noduri negre.
4. Scrieți un algoritm pseudocod pentru determinarea înălțimii negre a arborelui.
5. Care este numărul maxim de recolorări necesare pentru o operație de inserare?

5.3.7 Răspunsuri la testul de evaluare a cunoștințelor

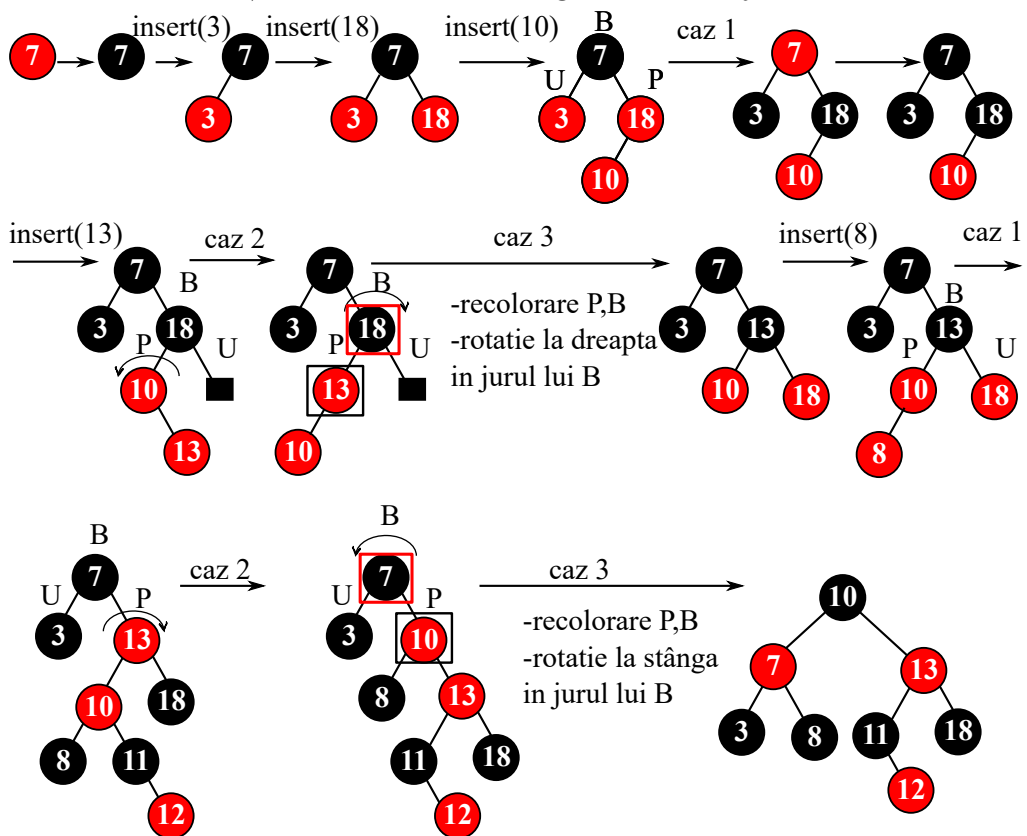
1. Dați 3 exemple de arbori roșu-negru care conțin exact 2 noduri roșii și 4 noduri negre.

Rezolvare:



2. Inserați într-un arbore roșu-negru inițial vid cheile: 7, 3, 18, 10, 13, 8, 11, 12.

Rezolvare: Inserțiile sunt ilustrate în figurile de mai jos:



3. Explicați de ce este important ca orice drum de la rădăcină la oricare frunză să conțină același număr de noduri negre.

Rezolvare: Această condiție, împreună cu condiția de a nu avea două noduri roșii legate printr-un arc, garantează faptul că arborele roșu-negru rămâne echilibrat. De fapt astfel niciun drum de la rădăcină la o frunză nu poate fi mai lung decât cel mult dublul altui drum de la rădăcină la altă frunză. Dacă numărul de noduri negre diferă pe drumurile de la același nod către nodurile frunze, unele frunze ar fi mult mai adânci decât altele, ceea ce ar conduce la o degradare a timpului de căutare. Prin urmare această proprietate asigură că înălțimea arborelui este $O(\log n)$ pe toate ramurile.

4. Scrieți un algoritm pseudocod pentru determinarea înălțimii negre a arborelui.

Rezolvare: Deoarece arborele este un ARN valid, se respectă proprietatea conform căreia toate drumurile de la rădăcină până la o frunză conțin același număr de noduri negre. Prin urmare, pentru a calcula înălțimea neagră este suficient să parcurgem un singur drum. Alegem să coborâm mereu pe fiul stâng până la nodul *nil*, iar la fiecare întâlnire cu un nod negru incrementăm valoarea înălțimii negre.

Algoritm: INALTIME-NEAGRA

Input: Un arbore roșu-negru T

start

$inaltime_neagra \leftarrow 0$

$nod \leftarrow T.radacina$

daca $nod \neq nil$ **atunci**

$nod \leftarrow nod.stanga$

sfarsit_daca

cat timp $nod \neq nil$ **executa**

daca $nod.culoare = negru$ **atunci**

$inaltime_neagra \leftarrow inaltime_neagra + 1$

sfarsit_daca

$nod \leftarrow nod.stanga$

sfarsit_cat_timp

 returneaza($inaltime_neagra + 1$)

stop

5. Care este numărul maxim de recolorări necesare pentru o operație de inserare?

Rezolvare: Analizând cele trei cazuri de refacere a proprietăților roșu-negru, observăm că numărul maxim de recolorări apare în **Cazul 1**, când se schimbă culoarea părintelui, a unchiului și a bunicului pentru nodul inserat z . Deoarece bunicul devine roșu, poate

apărea situația în care acesta are, la rândul său, un părinte roșu. În acest caz, procedura de refacere continuă pe nivelele superioare ale arborelui. Din acest motiv, numărul de recolorări în cel mai defavorabil caz este de ordinul $3 \cdot \log_2 n$.

Capitolul 6

Structuri de date avansate

Introducere

Există numeroase tipuri de arbori, fiecare fiind adaptat la anumite clase de probleme. În acest capitol vom discuta două tipuri de arbori, ce constituie structuri de date semnificativ mai complexe decât cele discutate până acum. B-arborii, abordați în prima unitatea de învățare, reprezintă o generalizare a arborilor binari de căutare și au fost dezvoltati pentru accesarea eficientă a unor cantități mari de date stocate pe disc. Arborii quad, prezentați în a doua unitate de învățare, sunt dedicați stocării și manipulării de date bidimensionale, reprezentând de obicei date spațiale.

Competențe

La sfârșitul acestui modul de învățare studenții:

- Definesc corect noțiunea B-arbore și explică operațiile de bază pe un astfel de arbore.
- Explică ce este un arbore quad pentru regiuni.
- Explică ce este un arbore de tip *Point-Region*

6.1 Unitatea de învățare 1 - B-arbori



6.1.1 Introducere

B-arborii sunt structuri de date echilibrate, caracterizate prin faptul că toate frunzele se află la aceeași adâncime. Ei reprezintă o generalizare a arborilor binari de căutare, permițând stocarea mai multor chei într-un singur nod, de la un număr redus până la câteva mii. Pe lângă chei, fiecare nod poate conține și alte informații utile, în funcție de aplicație.

O variantă des utilizată în practică este arborele B+, derivat din arborele B, care se distinge prin faptul că informațiile suplimentare, altele decât cheile, sunt păstrate exclusiv în nodurile frunză, în timp ce nodurile interne conțin doar chei pentru ghidarea căutării.



6.1.2 Obiective

La sfârșitul acestei unități de învățare studenții vor înțelege:

- Ce este un B-arbore.
- Care este înălțimea unui B-arbore.
- Care sunt operațiile de bază într-un B-arbore și ce complexitate au.



Durata medie de studiu individual

Parcursarea de către studenți a acestei unități de învățare se face în 3 ore.

6.1.3 Proprietățile B-arborilor

B-arborii nu sunt arbori binari și deci, fiecare nod poate avea mai mult de doi fii. Vom vedea și că, spre deosebire de tipurile de arbori discutate în capitolele anterioare, nodurile unui B-arbore conțin de obicei mai multe chei. Pentru a permite o căutare similară căutării în arborii binari de căutare și B-arborii dispun de o ordonare a cheilor, care va fi explicată în cele ce urmează.

Un B-arbore este caracterizat printr-o serie de proprietăți:

1. Fiecare nod x are următoarele atribute (câmpuri):

- (a) $x.n$ numărul de chei stocate în nodul x ;
- (b) vectorul de chei $x.chei$, cu cheile sortate crescător;
- (c) $x.frunza$ = un câmp boolean care este *adevarat*, dacă x este frunză și *fals* altfel;

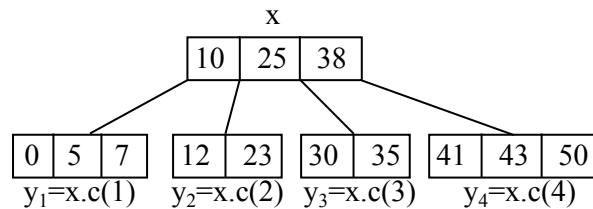
(d) Vectorul de legături către fiii lui x : $x.copii$.

2. Fiecare nod intern x are exact $x.n + 1$ fi.
3. Valorile cheilor $x.chei[i], i = 1, \dots, x.n$, separă valorile cheilor fiilor lui x după cum urmează. Dacă notăm cu $y_i = x.copii[i], i = 1, \dots, x.n + 1$ atunci:

$$\begin{aligned} y_1.chei[1] &\leq y_1.chei[2] \leq \dots \leq y_1.chei[y_1.n] \leq x.chei[1] \leq \\ &\leq y_2.chei[1] \leq y_2.chei[2] \leq \dots \leq y_2.chei[y_2.n] \leq x.chei[2] \leq \\ &\dots \\ &\leq y_{x.n+1}.chei[1] \leq y_{x.n+1}.chei[2] \leq \dots \leq y_{x.n+1}.chei[y_{x.n+1}.n] \end{aligned}$$

De fapt cheile $x.chei[i - 1]$ și $x.chei[i]$ definesc intervalul în care se pot afla valorile cheilor din subarborele $x.copii[i], i = \overline{2, x.n}$. Cheile din subarborele de rădăcină $x.copii[1]$ sunt mai mici decât cheia $x.chei[1]$ și cheile din subarborele de rădăcină $x.copii[x.n + 1]$ sunt mai mari decât cheia $x.chei[x.n]$.

De exemplu în figura de mai jos, în nodul x cheile sunt ordonate. x are 3 chei, deci are 4 copii.



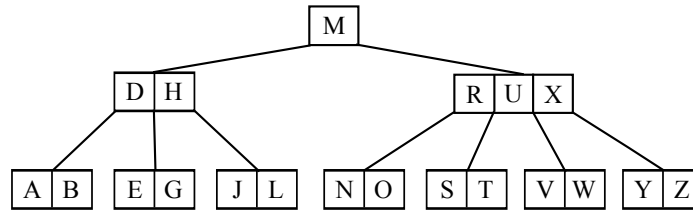
Copilul y_1 are cheile mai mici decât $x.chei[1] = 10$, copilul y_2 are cheile în intervalul $[10, 25]$, determinat de cheile $x.chei[1]$ și $x.chei[2]$, copilul y_3 are cheile în intervalul $[25, 38]$ determinat de cheile $x.chei[2]$ și $x.chei[3]$, iar copilul y_4 are cheile mai mari decât $38 = x.chei[3]$.

4. Toate frunzele au aceeași adâncime = adâncimea/înălțimea h a arborelui.
5. Numărul de chei ale unui nod este limitat inferior și superior pe baza unei constante t , $t \geq 2$, numită **gradul minim** al arborelui, după cum urmează:
 - (a) Fiecare nod x cu excepția rădăcinii, conține cel puțin $t - 1$ chei și are deci cel puțin t copii. Rădăcina conține cel puțin o cheie.

(b) Fiecare nod x conține cel mult $2t - 1$ chei, deci are cel mult $2t$ copii. Un nod care conține $2t - 1$ chei se numește **nod plin**. Această restricție este valabilă și pentru nodul rădăcină.



Exemplu de B-arbore cu $t = 2$ în care cheile sunt litere ale alfabetului. Ordinea considerată este cea lexicografică.



Înălțimea unui B-arbore

Considerăm un B-arbore de grad minim $t \geq 2$ cu n chei. Atunci înălțimea h a arborelui respectă inegalitatea:

$$h \leq \log_t \left(\frac{n + 1}{2} \right)$$

Demonstrație: vom calcula înălțimea unui B-arbore T cu n chei pornind de la un B-arbore T_1 cu aceeași înălțime h , dar cu număr minim de chei. Evident $n \geq$ numărul de chei T_1 .

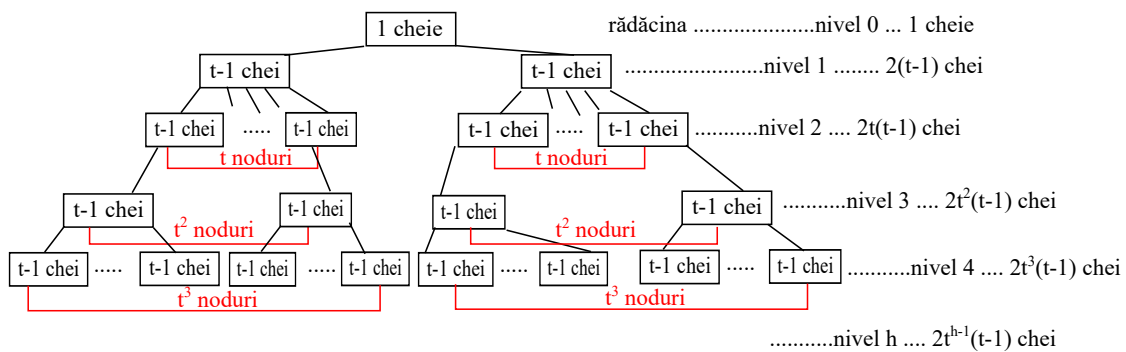


Figura 6.1: Numărul minim de chei într-un B-arbore T_1 de grad minim t și înălțime h

Numărul de chei ale lui T_1 , reprezentat în figura 6.1, se calculează astfel:

- Nivel 0: rădăcina conține o singură cheie
- Nivel 1: conține 2 noduri a câte $t - 1$ chei

- Nivel 2: conține $2 * t$ noduri a câte $t - 1$ chei
- Nivel 3: conține $2 * t^2$ noduri a câte $t - 1$ chei

.....

- Nivel h : conține $2 * t^{h-1}$ noduri a câte $t - 1$ chei

În total numărul de chei ale arborelui T_1 este:

$$\begin{aligned} 1 + 2(t - 1) + 2t(t - 1) + 2t^2(t - 1) + \dots + 2t^{h-1}(t - 1) &= \\ &= 1 + 2(t - 1)(1 + t + t^2 + \dots + t^{h-1}) = \\ &= 1 + 2(t - 1)(t^h - 1)/(t - 1) = 2t^h - 1. \end{aligned}$$

Deci:

$$n \geq 2t^h - 1 \Rightarrow h \leq \log_t \left(\frac{n + 1}{2} \right).$$

Înălțimea unui arbore este cu atât mai mică, cu cât gradul minim t este mai mare. Complexitatea operațiilor de căutare, inserție și ștergere depinde de înălțimea arborelui, deci de t .

Domenii de utilizare

B-arborii pot fi utilizați cu succes pentru accesul rapid al datelor de pe hard-disk, prin minimizarea numărului de operații de citire/scriere. În memoria principală (în memoria RAM), cu acces rapid, se păstrează rădăcina B-arborelui, iar nodurile din subarbore se păstrează în memoria secundară, mai lentă. Accesul la un nod din memoria secundară se realizează în complexitate $O(\log_t n)$, unde baza logaritmului depinde de gradul minim al arborelui, iar numărul de accesări ale discului în acest scop este dat de lungimea drumului de la rădăcină la nodul căutat. Cu cât gradul minim al arborelui este mai mare, cu atât înălțimea este mai mică și accesul la informație este mai rapid.

Multe baze de date utilizează B-arbori sau variante ale acestora pentru memorarea datelor.

6.1.4 Operații

Așa cum s-a menționat anterior, B-arborii sunt utilizați în general pentru optimizarea accesului la memorie. Nodul rădăcină este memorat în memoria principală (în memoria RAM), astfel încât accesul la rădăcină nu necesită operații de citire pe disc. Accesul la noduri interne necesită citire pe disc.

Căutarea unei chei

Căutarea într-un B-arbore este de fapt o generalizare a căutării binare, doar că spre deosebire de aceasta, pentru fiecare nod nu avem doar o decizie între două ramuri, ci avem o decizie între $n + 1$ ramuri, n fiind numărul de chei ale nodului curent.

Algoritm: B-CAUTA

Input: Nodul curent nod , cheia căutată k

start

$i \leftarrow 1$

cat timp $i \leq nod.n$ și $nod.chei[i] < k$ **executa**

$i \leftarrow i + 1$

sfarsit_cat_timp

daca $i \leq nod.n$ și $nod.chei[i] = k$ **atunci**

 returneaza(nod, i)

sfarsit_daca

altfel

daca $nod.frunza = adevarat$ **atunci**

 returneaza(nil)

sfarsit_daca

altfel

 CITESTE-DISC($nod.copii[i]$)

 returneaza($B-CAUTA(nod.copii[i], k)$)

sfarsit_daca

sfarsit_daca

stop

Funcția $B-CAUTA(nod, k)$ [4] este definită recursiv și are ca parametri rădăcina subarborului în care s-a ajuns cu căutarea și cheia căutată. Funcția returnează o pereche (nod, i) , unde nod = nodul care conține cheia k și i indicele cheii în nodul nod , adică $x = nod.chei[i] = k$. Dacă în arbore nu se găsește cheia k , atunci funcția returnează nil .

Complexitate: căutarea în fiecare nod este de ordinul $O(t)$, iar căutarea în arbore depinde de înălțimea h a arborelui, care este de ordinul $\log_t n$. Astfel complexitatea totală a algoritmului este $O(t \log_t n)$.

Datorită faptului că într-un nod cheile se păstrează sortate, se poate realiza și căutare binară. În acest caz, complexitatea devine: $O(\log_2 t \log_t n)$.

Inserarea unei chei

Inserarea unei chei într-un B-arbore diferă de inserarea într-un arbore binar prin faptul că nu putem pur și simplu crea o nouă frunză în B-arbore în care să inserăm cheia respectivă. Un astfel de procedeu ar duce la dezechilibrarea arborelui și ar contraveni proprietății unui B-arbore de a avea toate frunzele pe același nivel.

Insertia unei chei se realizează prin inserarea cheii într-un nod frunză deja existent. Problema apare atunci când frunza în care ar trebui inserat nodul este plină. În acest caz trebuie aplicată o procedură de DIVIZARE a frunzei în jurul cheii mediane (aflate pe poziția din mijloc în șirul cheilor).

Operația de divizare a unui nod

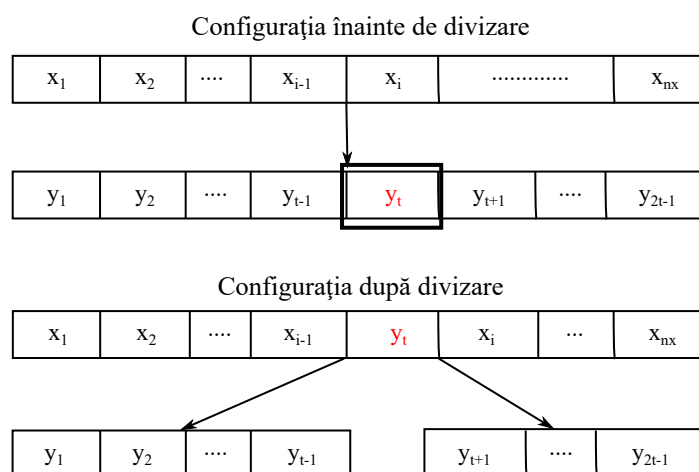


Figura 6.2: Divizarea unui nod plin. S-a notat cu $x_k = x.copii[k]$ și cu $y_k = y.copii[k]$

Se consideră un nod plin y , deci cu $2t - 1$ chei. Presupunem că y este al i -lea fiu al nodului x , care nu este plin. Atunci divizarea se realizează în modul următor:

- $x.n$ crește cu o unitate: $x.n = x.n + 1$.
- cheile lui x începând de la poziția i se deplasează cu o poziție la dreapta în nod.
- se deplasează pointerii către descendenții $x.copii[i + 1], \dots, x.copii[x.n]$ cu o poziție spre dreapta în vectorul $x.copii$.
- cheia $y.chei[t]$ urcă în nodul x , în vectorul $x.chei$ pe poziția i

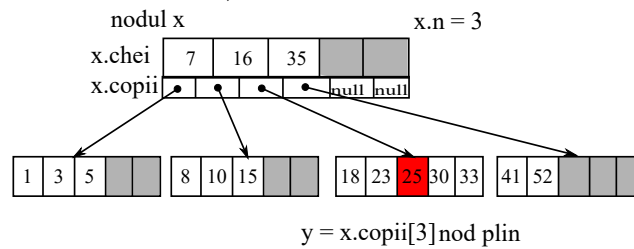
- nodul y se divizează în două noduri noi care conțin câte $t - 1$ chei și anume primul nod conține cheile $y.chei[1], \dots, y.chei[t - 1]$ și al doilea nod conține cheile $y.chei[t + 1], \dots, y.chei[2t - 1]$
- noile noduri create devin copii ai $x.copii[i]$ și $x.copii[i + 1]$.

Această procedură este ilustrată în figura 6.2.

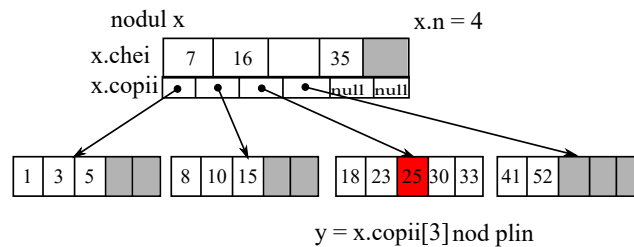


Exemplu pentru divizarea nodului $y = x.copii[i]$ în jurul cheii mediane $y.chei[t]$. În exemplu se avem $t = 3, i = 3$. Cum $t = 3$ rezultă că numărul maxim de chei într-un nod este 5.

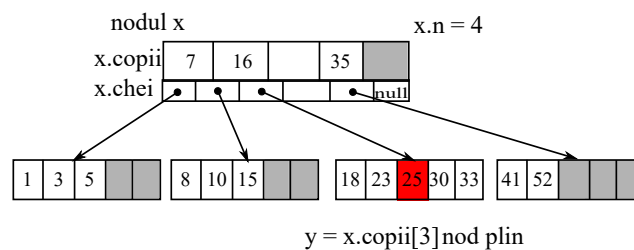
Configurația înainte de divizare



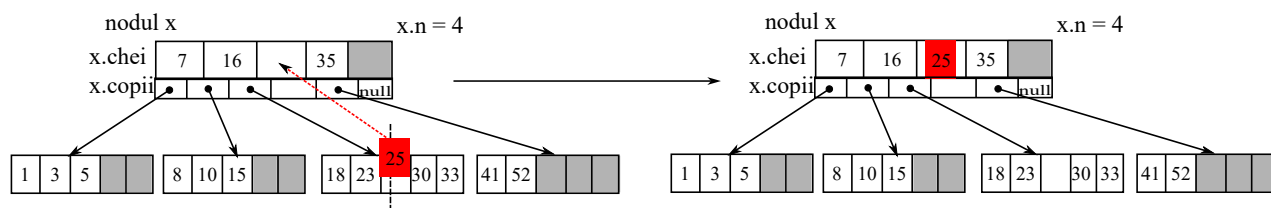
Se incrementează numărul de chei ale nodului x și se deplasează toate cheile începând de la cheia a i -a cu o poziție spre dreapta, în cazul nostru doar a treia.



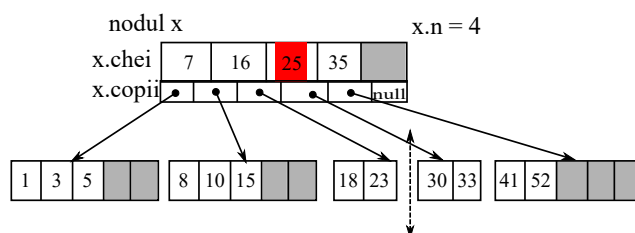
Se deplasează cu o poziție spre dreapta pointerii spre copiii nodului x , începând cu al $i + 1$ -lea, în cazul nostru doar ultimul copil.



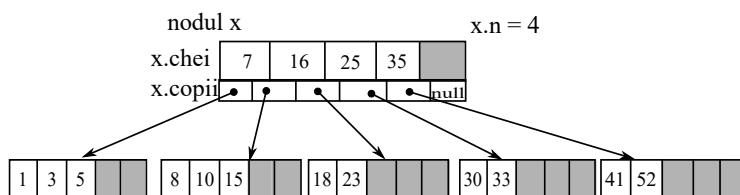
Se plasează cheia mediană din nodul y pe poziția i în nodul x .



Se divizează nodul y în 2 noduri, prin crearea unui nou nod, în care se copiază ultimele $t - 1$ chei ale lui y .



Se obține configurația finală:



Observații:

1. Dacă x este la rândul său plin, nu putem efectua divizarea. De aceea, la inserție trebuie să ne asigurăm deja pe parcursul căutării frunzei în care se realizează inserția că pe drum nu întâlnim noduri pline. În caz contrar, se realizează o divizare.
2. Dacă nodul plin y este rădăcina, atunci nu există un părinte x în care să se insereze cheia mediană din y . Astfel, dacă nodul care trebuie divizat este chiar rădăcina, aceasta trebuie divizată și se creează o nouă rădăcină.

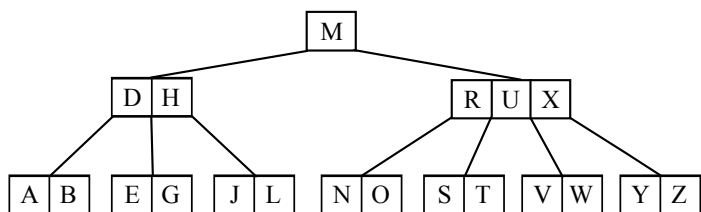
Divizarea rădăcinii

- Se creează un nou nod vid, care va fi noua rădăcină.
- Se leagă vechea rădăcină ca descendent al noii rădăcini.
- Se realizează operația de divizare.

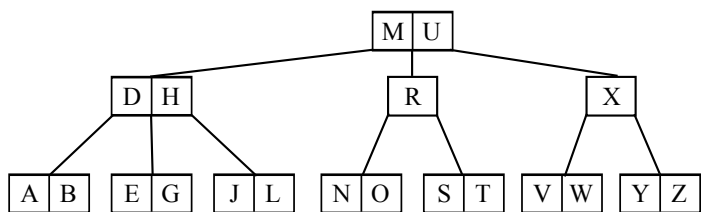
3. Înălțimea unui B-arbore crește doar prin divizarea rădăcinii.



Exemplu pentru inserarea cheii P în B-arborele din figură cu $t = 2$.

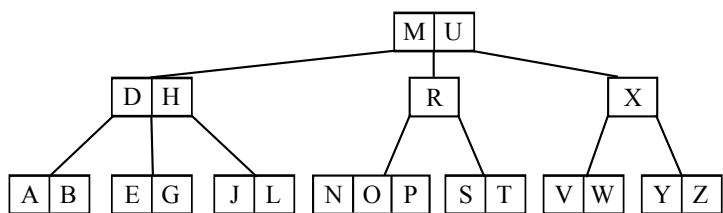


Căutarea începe de la nodul rădăcină. $T.rad.n = 1$, iar $T.rad.chei[1] < P$, deci trebuie coborât la descendentul $T.rad.copii[2]$. Dar se observă că $T.rad.copii[2].n = 2t - 1 = 3$, deci nodul este plin. Înainte de a continua, nodul $T.rad.copii[2]$ trebuie divizat. Se obține arborele:



Acum $T.rad.chei[1] = M < P$ și $T.rad.chei[2] = U > P$, deci se coboară la descendentul $x_1 = T.rad.copii[2]$.

Avem: $x_1.n = 1$, $x_1.chei[1] = R > P$, deci se coboară la nodul $x_2 = x_1.copii[1]$, care este frunză. $x_2.n = 2 < 2t - 1$, deci nodul nu este plin și se poate insera P pe poziția potrivită. Rezultă arborele:



Se consideră arborele din exemplul anterior după ce a fost inserată cheia P . Inserați în acesta în continuare cheile I , K și F .

Eliminarea unei chei

Eliminarea unei chei dintr-un B-arbore este ceva mai delicată decât inserția, deoarece uneori trebuie eliminată și o cheie dintr-un nod intern. Ștergerea pur și simplu a unei chei dintr-un nod intern ar avea însă ca urmare reducerea numărului de chei din nod și deci ar trebui redus

numărului de copii ai nodului respectiv, ceea ce nu se poate. Astfel devine necesară o rearanjare a cheilor și a nodurilor în arbore. În plus, la eliminarea unei chei, trebuie avut grijă ca numărul de chei dintr-un nod să nu devină mai mic decât $t - 1$. Ceea ce înseamnă că nu se poate pur și simplu extrage o cheie dintr-un nod cu $t - 1$ chei. Aceste probleme se rezolvă prin operații de rotație și prin fuziuni.

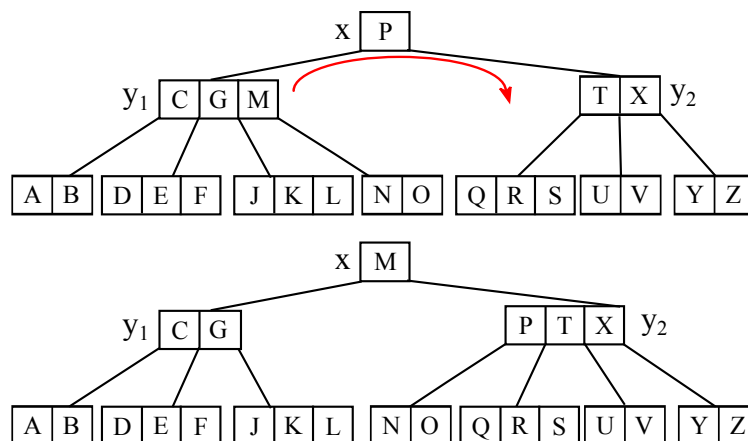
Rotația într-un B-arbore

Se consideră nodul x .

- O rotație la dreapta în jurul cheii $k = x.chei[i]$:
 - mută ultima cheie a fiului $y_i = x.copii[i]$ în nodul x în locul cheii k
 - mută cheia k pe prima poziție în nodul $y_{i+1} = x.copii[i + 1]$
 - mută ultimul fiu al lui y_i ca prim fiu al lui y_{i+1} .
- O rotație la stânga în jurul cheii $k = x.chei[i]$:
 - mută prima cheie a fiului $y_{i+1} = x.copii[i + 1]$ în nodul x în locul cheii k
 - mută cheia k pe ultima poziție din nodul $y_i = x.copii[i]$
 - mută primul fiu al lui y_{i+1} ca ultim fiu al lui y_i .



Exemplu de rotație la dreapta în jurul cheii P din rădăcină



Fuziunea a două noduri vecine într-un B-arbore

O fuziune nu poate fi realizată decât între doi frați vecini, fi ai aceluiași nod x , și care au ambii exact $t - 1$ chei. De asemenea, părintele x trebuie să aibă cel puțin t chei. Se consideră

nodul x , iar descendenții care fuzionează sunt $y = x.copii[i]$ și $z = x.copii[i + 1]$. Fuziunea lui y cu z presupune reunirea celor două noduri într-unul singur, iar cheia $x.chei[i]$ coboară din nodul x în noul nod rezultat pe poziția aflată între cheile lui y și cele ale lui z .

Această operație este reprezentată grafic în figura 6.3.

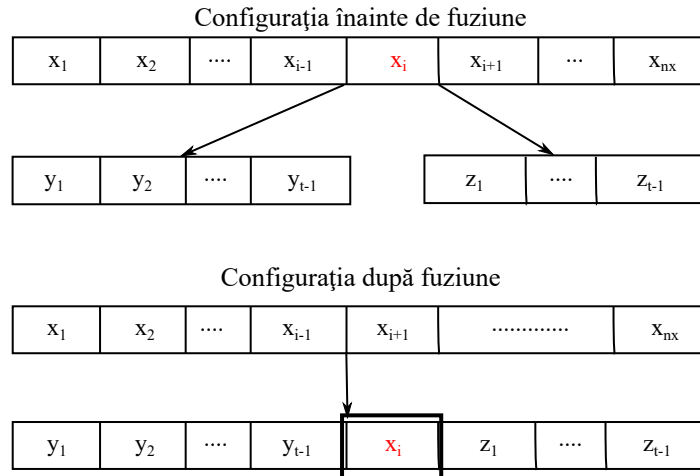


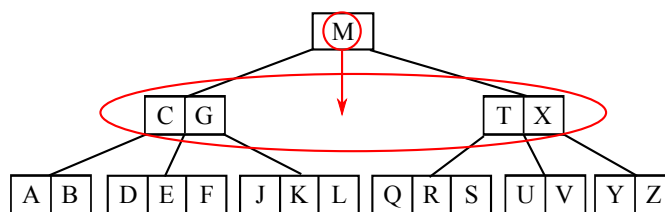
Figura 6.3: Fuziunea a două noduri vecine. S-au notat prin $x_k = x.copii[k]$, $y_k = y.copii[k]$ și $z_k = z.copii[k]$.

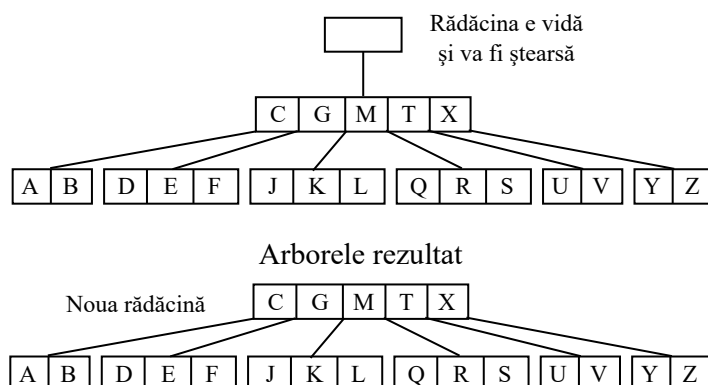
Observații:

- Înălțimea unui arbore se micșorează atunci când rădăcina are doi fii care fuzionează.
- O fuziune se poate realiza doar dacă părintele nodurilor care fuzionează are cel puțin t chei. Din acest motiv, la fel cum, în timpul inserării, pe parcursul căutării nodului în care se va insera cheia, toate nodurile pline întâlnite sunt divizate, în cazul eliminării unei chei, pe parcursul căutării acesteia, înainte de a coborî la un descendent, trebuie să ne asigurăm că acel descendent are cel puțin t chei.



Exemplu de fuziune. a) Nodurile care urmează să fuzioneze împreună cu cheia M a rădăcinii; b) Arborele după fuzionare; c) Rezultatul după eliminarea rădăcinii vide.





Algoritm de eliminare a unei chei dintr-un B-arbore

Eliminarea chei k din B-arborele T cu gradul minim t va fi tratată în mod recursiv. Se pornește de la rădăcină și se coboară în arbore până la întâlnirea nodului ce conține cheia k .

Notăm cu x nodul curent. Pe parcurs ne asigurăm că orice nod în care se coboară are cel puțin t chei.

Cazul I: x este frunză și k este cheie a lui x . Din faptul că s-a avut grijă ca nodurile în care se coboară să aibă cel puțin t chei, x are cel puțin t chei, deci pur și simplu se extrage cheia k din nodul x și se încheie algoritmul.

Cazul II: x nu este frunză și $k = x.chei[i]$. Atunci:

- (a) **Dacă** $y = x.copii[i]$ (copilul care îl precede pe k) are $y.n \geq t$ atunci se caută k' predecesorul lui k în subarborele de rădăcină y , se șterge recursiv k' din subarborele de rădăcină y și se înlocuiește k cu k' în x .
- (b) **Altfel dacă** $y = x.copii[i + 1]$ (copilul care îi urmează lui k) are $y.n \geq t$ atunci se caută k'' succesorul lui k în subarborele de rădăcină y , se șterge recursiv k'' din acest subarbore și se înlocuiește k cu k'' în x .
- (c) **Altfel** - adică ambii copii aflați de o parte și de alta a lui k au exact $t - 1$ chei - atunci: se realizează o fuziune între cei doi copii $x.copii[i]$ și $x.copii[i + 1]$. Astfel se obține un nou nod, care va conține cheia k . Aceasta se șterge recursiv din noul nod obținut prin fuziune.

Cazul III: x nu este frunză și k nu este cheie a lui x .

Atunci se determină i astfel încât $x.chei[i - 1] < k < x.chei[i]$. Rezultă faptul că trebuie căutată

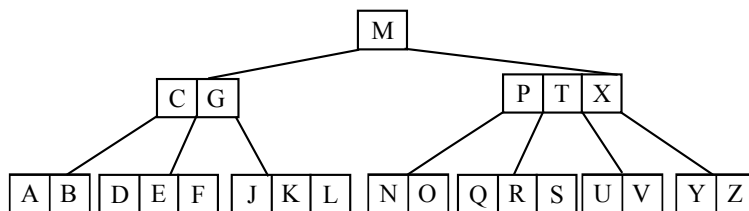
cheia k în subarboarele de rădăcină $x.copii[i]$. Înainte de a coborî la fiul $x.copii[i]$ al lui x trebuie să ne asigurăm că acest fiu are cel puțin t chei. Dacă $x.copii[i]$ are doar $t - 1$ chei atunci:

- (a) Dacă $x.copii[i]$ are un frate vecin cu cel puțin t chei se realizează o rotație astfel încât o cheie din acel frate să urce la părintele x și o cheie din x să coboare la nodul $x.copii[i]$. Apoi se șterge recursiv k din $x.copii[i]$.
- (b) Dacă ambii frați vecini (sau singurul, dacă există doar un frate vecin) au $t - 1$ chei, trebuie realizată fuziunea lui $x.copii[i]$ cu unul dintre acești frați. Apoi se șterge recursiv k din nodul rezultat prin fuziune.

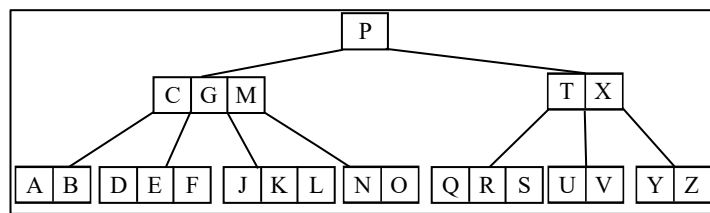
Apoi se șterge recursiv k din $x.copii[i]$.



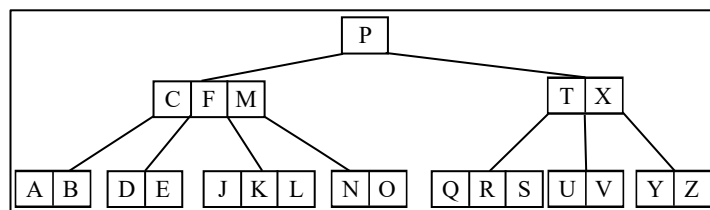
Exemplu Considerăm arborele din figură cu $t = 3$.



Se șterge cheia G : se pornește de la rădăcină $G < M$ și suntem în cazul III. Trebuie coborât la primul copil. Acesta însă are exact $t - 1 = 2$ chei. Fratele drept al acestui nod are t chei, deci suntem în cazul III (a) \Rightarrow rotație la stânga în jurul cheii M din rădăcină. Se obține arborele:

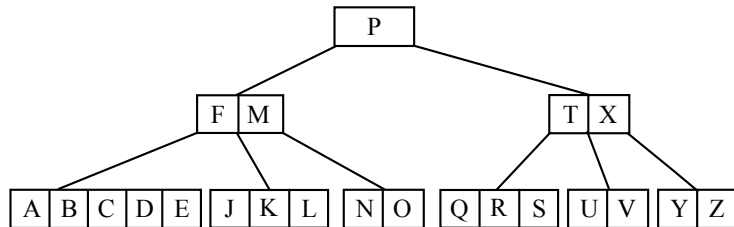


Acum se coboară la nodul x care conține cheile C, G, M . Se observă că acesta nu e frunză, dar conține cheia $G = x.chei[2]$ și ambii fii $x.copii[2]$ și $x.copii[3]$ au t chei, deci pot aplica cazul II (a) \Rightarrow caut predecesorul lui G , care este F . Înlocuiesc G cu F și șterg F din fiul $x.copii[2]$. Se obține arborele:

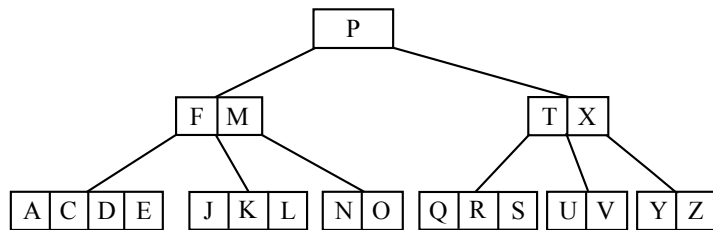


Se șterge cheia B : se pornește cu $x = T.rad$, se compară $B < x.chei[1] \Rightarrow$ se studiază fiul $x.copii[1]$. Acesta are t chei $\Rightarrow x = x.copii[1]$.

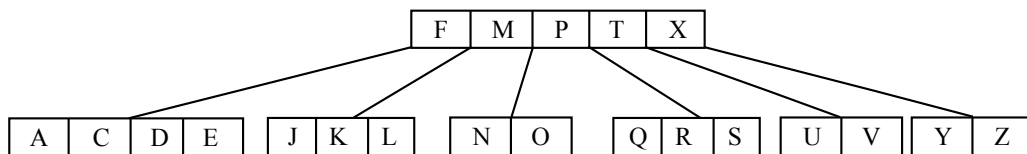
Se observă că în acest moment $B < x.chei[1] = C \Rightarrow$ se studiază fiul $x.copii[1]$. Acest fiu nu are decât $t - 1$ chei, iar singurul său frate are tot $t - 1$ chei \Rightarrow suntem în cazul III (b) \Rightarrow trebuie realizată o fuziune între $x.copii[1]$ și $x.copii[2]$. Se obține arborele:



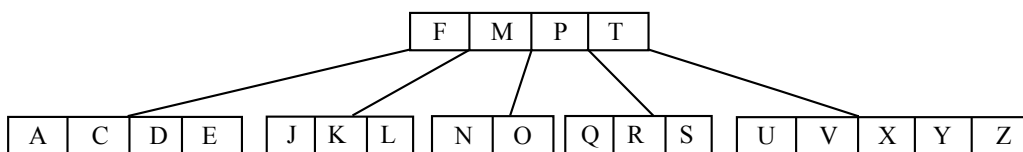
Se coboară acum la $x = x.copii[1]$ care este frunză și se șterge cheia B din nod. Rezultă arborele:



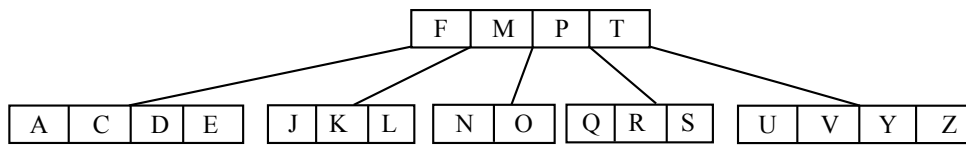
Se șterge cheia X : se pornește de la rădăcina $x = T.rad$. $X > x.chei[1] \Rightarrow$ se studiază fiul $x.copii[2]$. Acesta are doar $t - 1$ chei. Unicul frate al său are tot $t - 1$ chei \Rightarrow fuziune. Rădăcina inițială rămâne fără nici o cheie, deci se șterge și noua rădăcină va fi nodul rezultat în urma fuziunii. Rezultă arborele:



Cheia X se găsește în rădăcină și ambii fii, de o parte și de alta a lui X au exact $t - 1$ chei \Rightarrow fuziune. Se obține:



Acum se poate șterge X din frunza care îl conține și se obține:



Complexitate: Operațiile într-un B-arbore presupun căutare în interiorul unui nod, care este $O(t)$ precum și coborâre în arbore, care depinde de înălțimea h , deci este $O(\log_t n)$. Astfel complexitatea este $O(t \log_t n)$. Pentru t mare, poate fi utilizată căutarea binară într-un nod, astfel scade complexitatea acestei operații la $O(\log_2 t)$.

Algoritmii în pseudocod pentru operațiile într-un B-arbore pot fi găsiți în [4].



Să ne reamintim...

- Un B-arbore este o generalizare a unui arbore binar de căutare.
- Fiecare nod poate să conțină mai multe chei, numărul minim și maxim de chei dintr-un nod depinzând de gradul minim t al arborelui.
- Într-un B-arbore cheile sunt ordonate astfel încât să permită o căutare similară cu cea dintr-un arbore binar de căutare.
- Înălțimea unui B-arbore este de ordinul

$$h \leq \log_t \left(\frac{n+1}{2} \right)$$

unde n este numărul de chei stocate în arbore.

6.1.5 Rezumat



În această secțiune au fost prezentați B-arborii împreună cu proprietățile lor. S-a explicat modul de inserție bazat pe divizarea nodurilor, precum și ștergerea unei chei, care are la bază procesul invers, al fuziunii, și anume divizarea.



6.1.6 Test de autoevaluare

1. Desenați toți B-arborii cu $t = 2$ și $t = 3$ care conțin exact cheile A, B, C, D, E .
2. Inserați într-un B-arbore cu $t = 2$, inițial vid, cheile: 10, 20, 5, 6, 12, 30, 7, 17 (în această

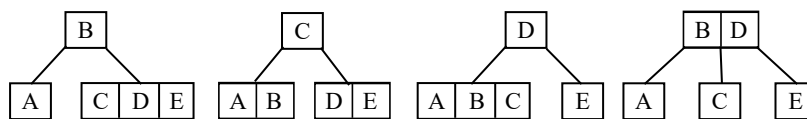
ordine).

3. Explicați de ce căutarea într-un B-arbore este mai eficientă decât într-un arbore binar atunci când datele sunt stocate pe disc.
4. Determinați numărul maxim de chei care pot fi stocate într-un B-arbore, cu gradul minim $t = 3$ de înălțime $h = 4$, care are exact 2 chei în rădăcină.
5. Scrieți un algoritm pseudocod pentru determinarea succesorului unei chei într-un B-arbore.
6. Scrieți un algoritm pseudocod pentru afișarea cheilor unui B-arbore în ordine crescătoare.

6.1.7 Răspunsuri la testul de evaluare a cunoștințelor

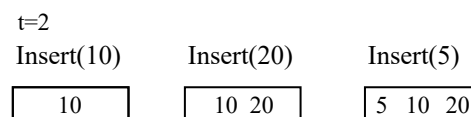
1. Desenați toți B-arborii cu $t = 2$ și $t = 3$ care conțin exact cheile A, B, C, D, E .

Rezolvare: B-arborii cu $t = 2$, care conțin exact cheile cerute sunt cei din figură:

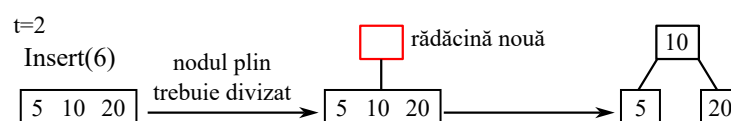


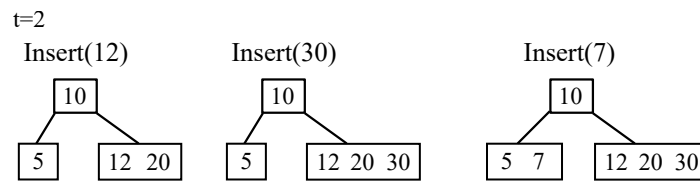
2. Inserați într-un B-arbore cu $t = 2$, inițial vid, cheile: 10, 20, 5, 6, 12, 30, 7, 17 (în această ordine).

Rezolvare: Inițial arborele este vid. Se creează o nouă rădăcină, care este și frunză și în care au loc primele trei chei:

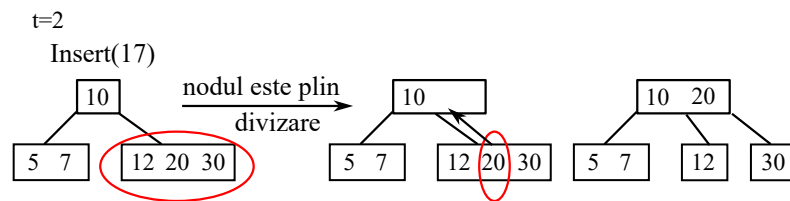


La inserția cheii 6 rădăcina este plină, deci trebuie divizată:





Pentru inserarea cheii 17, trebuie coborât la stânga cheii 10 a rădăcinii, dar nodul este plin, deci trebuie divizat.



S-a obținut arborele final.

- Explicați de ce căutarea într-un B-arbore este mai eficientă decât într-un arbore binar atunci când datele sunt stocate pe disc.

Rezolvare: Într-un arbore binar de căutare, fiecare nod conține o singură cheie și are cel mult doi fii. Astfel, în cel mai bun caz, la fiecare pas domeniul de căutare se reduce la jumătate. Complexitatea este dependentă de înălțimea arborelui, care, pentru un arbore balansat, este $O(\log_2 n)$.

În schimb, un B-arbore permite ca fiecare nod să conțină mai multe chei și să aibă mai mulți fii, ceea ce reduce considerabil numărul de noduri accesate pe parcursul căutării. Adâncimea unui B-arbore este $O(\log_t n)$, unde t reprezintă ordinul arborelui.

- Determinați numărul maxim de chei care pot fi stocate într-un B-arbore, cu gradul minim $t = 3$ de înălțime $h = 4$, care are exact 2 chei în rădăcină.

Rezolvare: Pentru un B-arbore cu gradul minim $t = 3$ avem:

- numărul minim de chei permis într-un nod (în afară de rădăcină): $t - 1 = 2$;
- umărul maxim de chei într-un nod: $2t - 1 = 5$.

Numărul maxim de chei N_{\max} se obține atunci când pe fiecare nivel avem un număr maxim de noduri și fiecare nod este umplut la capacitatea maximă (5 chei). Fiecare nod cu câte 5 chei, care nu este frunză, are 6 copii. Astfel, numărul maxim de copii

de pe nivelul $k + 1$ este $6 \times$ numărul de copii de pe nivelul k . Excepție face numărul de copii de pe nivelul 1, deoarece în rădăcină avem doar două chei, deci pe nivelul 1 avem 3 fi.

Obținem o distribuție a nodurilor și cheilor pe niveluri pentru numărul maxim de chei pentru B-arborele din cerință, ca în tabelul următor.

Nivel	Număr noduri	Chei pe nod	Chei totale pe nivel
0	1	2 (din cerință)	2
1	3	5	15
2	$3 \times 6 = 18$	5	$18 \times 5 = 90$
3	$18 \times 6 = 108$	5	$108 \times 5 = 540$
3	$108 \times 6 = 648$	5	$648 \times 5 = 3240$
Total	778	-	3887

Numărul maxim de chei în B-arborele din cerință este deci $N_{\max} = 3887$

5. Scrieți un algoritm pseudocod pentru determinarea succesorului unei chei într-un B-arbore.

Rezolvare: Succesorul unei chei k este cheia imediat următoare lui k în ordinea sortată a cheilor din B-arbore. Parametrii algoritmului sunt *nod* care conține cheia la poziția i :

- Dacă *nod* este intern, succesorul este minimul din subarborele $nod.copii[i + 1]$.
- Dacă *nod* este o frunză și $i < nod.n$, atunci succesorul este $nod.chei[i + 1]$.
- Dacă *nod* este frunză și $i = nod.n$, succesorul se caută printre strămoșii nodului.

Algorithm: B-SUCCESSOR

Input: Nodul curent nod , indicele i al cheii pentru care se caută succesorul

Output: Perechea (nod, i) , reprezentând nodul ce conține succesorul dorit și indicele acestuia.

start

```

daca  $nod = nil$  atunci
  | returneaza(( $nil, 0$ ))
sfarsit_daca
daca  $nod.frunza = adevarat$  atunci
  | daca  $i < nod.n$  atunci
    | returneaza(( $nod, (i + 1)$ ))
    sfarsit_daca
     $parinte = nod.parinte$ 
    cat timp  $parinte \neq nil$  executa
      | pentru  $k = 1, parinte.n$  executa
        | daca  $parinte.copii[k] = nod$  atunci
          | returneaza(( $parinte, k$ ))
          sfarsit_daca
        sfarsit_for
      |  $nod \leftarrow parinte$ 
      |  $parinte \leftarrow nod.parinte$ 
    sfarsit_cat_timp
  | returneaza(( $nil, 0$ ))
sfarsit_daca
altfel
  | CITESTE-DISC( $nod.copii[i + 1]$ )
  | returneaza(( $B - MINIM(nod), 1$ ))
sfarsit_daca

```

stop

Algoritm: B-MINIM

Input: Nodul curent *nod***Output:** Nodul ce conține minimumul din subarborele de rădăcină *nod***start**| **daca** *nod = nil atunci*| | **returneaza**(*nil*)| **sfarsit_daca**| **cat timp** *nod.frunza = fals executa*| | *nod* ← *nod.copii*[1]| **sfarsit_cat timp**| **returneaza**(*nod*)**stop**

6. Scrieți un algoritm pseudo-cod pentru afișarea cheilor unui B-arbore în ordine crescătoare.

Rezolvare:

Algoritm: B-AFISARE

Input: Nodul curent *nod***start**| **daca** *nod = nil atunci*| | **return**| **sfarsit_daca**| **daca** *nod.frunza = false atunci*| | **pentru** *i = 1, nod.n executa*| | | **scrie**(*nod.chei*[*i*])| | | CITEȘTE-DISC(*nod.copii*[*i*])| | | *B* – PRINT(*nod.copii*[*i*])| | **sfarsit_for**| | CITEȘTE-DISC(*nod.copii*[*nod.n + 1*])| | *B* – PRINT(*nod.copii*[*nod.n + 1*])| **sfarsit_daca**| **altfel**| | **pentru** *i = 1, nod.n executa*| | | **scrie**(*nod.chei*[*i*])| | **sfarsit_for**| **sfarsit_daca****stop**

6.2 Unitatea de învățare 2 - Arbori Quad



6.2.1 Introducere

Arborii quad [6, 14] sunt structuri de date ierarhice care au la bază un principiu de descompunere recursivă a spațiului de tip *divide et impera*. Denumirea provine de la faptul că fiecare nod intern al unui arbore quad are 4 fi.

Arborii quad se diferențiază după:

- Tipul de date pe care îl stochează: puncte, regiuni, curbe, suprafețe, volume.
- Principiul de descompunere: descompunere în părți egale la fiecare nivel (de exemplu: poligoane regulate) sau descompunere pe baza anumitor condiții de intrare.
- Rezoluția descompunerii: poate fi fixată dinainte sau poate depinde de anumite proprietăți ale datelor de intrare.

În această unitate de învățare vor fi descrise două tipuri de astfel de arbori, arborii quad pentru suprafețe și arborii de tip *point region*.



6.2.2 Obiective

La sfârșitul acestei unități de învățare studenții vor înțelege:

- Ce un arbore quad pentru regiuni.
- Cum se construiesc astfel de arbori pentru suprafețe concrete.
- Cum pot fi identificați vecinii unui nod în arbore.
- Ce sunt arborii quad de tip *point region*.
- Ce operații pot fi efectuate pe arbori quad de tip *point region*.



Durata medie de studiu individual

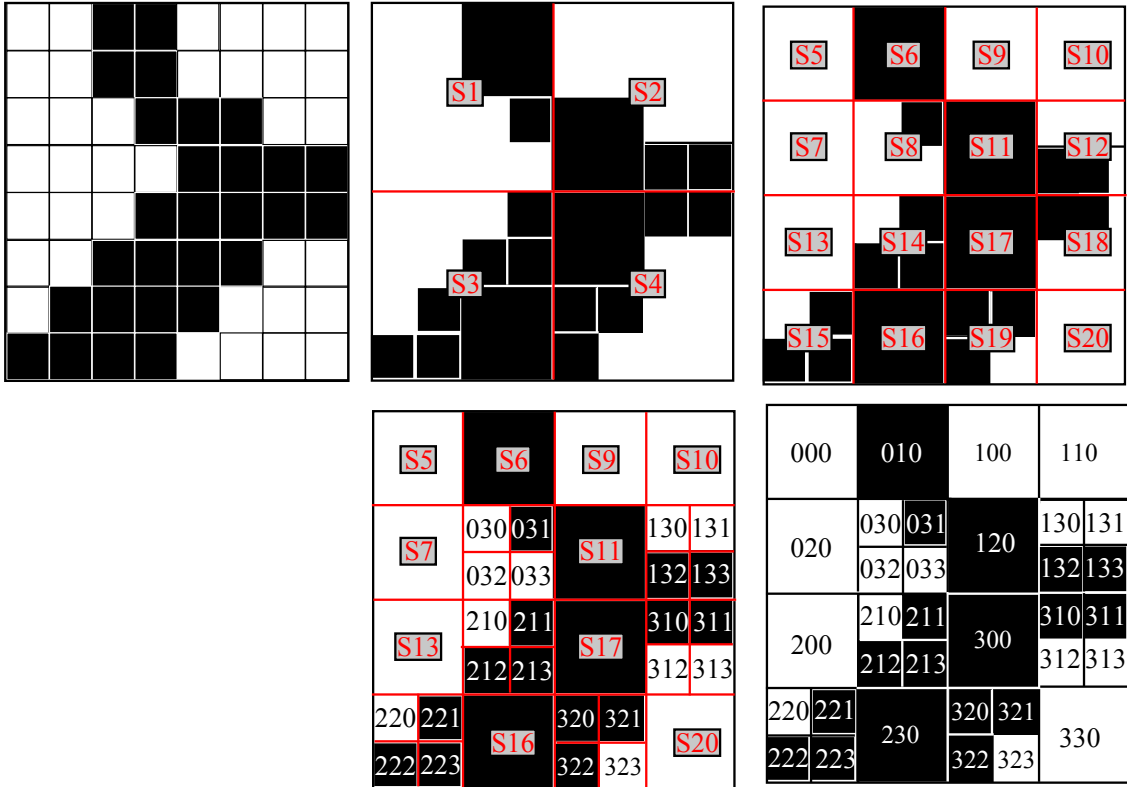
Parcurgerea de către studenți a acestei unități de învățare se face în 3 ore.

6.2.3 Arbori quad pentru regiuni

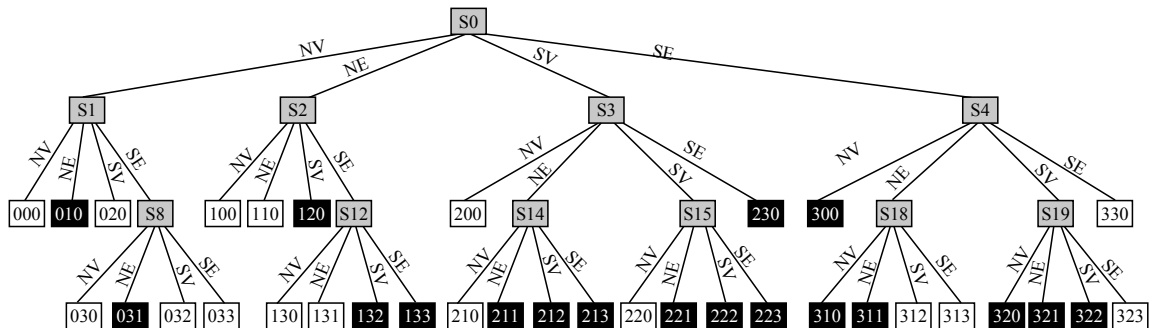
Un arbore quad pentru regiuni se utilizează în general pentru partiționarea spațiului bi- sau tridimensional în regiuni, conform unui anumit criteriu de descompunere.



Exemplu partiționarea unei imagini digitale binare (stânga sus) în regiuni de culoare uniformă, conform algoritmului *Split and Merge*. Rezultatul final este cel din dreapta jos.



Ideea principală a etapei de descompunere - *split* - este aceea de a descompune recursiv suprafața pătrată curentă în patru suprafețe pătrate de dimensiuni egale, până când se ajunge la suprafețe uniforme. Se consideră pentru aceasta imagini de dimensiune $2^n \times 2^n$. Arborele quad corespunzător este:



Codurile asociate frunzelor vor fi explicate la sfârșitul secțiunii.

Observații:

- Structura pătrată pentru partiționarea unei regiuni este cea mai simplă, dar există și alte tipuri de descompuneri: triunghiuri echilaterale, triunghiuri dreptunghice isoscele, hexagoane regulate.
- Arborii quad cresc viteza de execuție a anumitor algoritmi pe imagini, suprafețe, structuri geometrice, dat fiind că majoritatea acestor algoritmi se reduc la o parcurgere, de obicei în preordine, a arborelui quad asociat.
- În diferite situații - de exemplu etapa de unire a regiunilor vecine uniforme - este necesară identificarea vecinilor unui anumit nod.

Adiacență și vecinătate

Două regiuni se numesc vecine, dacă sunt adiacente. Adiacența în spațiu nu înseamnă neapărat o relație simplă în cadrul nodurilor corespunzătoare din arbore. De exemplu, în suprafața binară din exemplul anterior regiunile 211 și 033 sunt adiacente, dar în cadrul arborelui quad corespunzător nodurile respective nu sunt nici în relația de părinte-fiu, nici în relația de frate-frate. Se pune deci problema găsirii în arbore a două noduri, care pe suprafața bidimensională reprezintă regiuni vecine.

Fiecare nod al unui arbore quad corespunde unei regiuni/unui bloc din imagine. Fiecare bloc are 4 laturi și 4 colțuri.

Notății:

- Cele patru laturi ale unui bloc: N (nord), S (sud), E (est), V (vest).
- Cele patru colțuri: NV , NE , SV , SE

Adiacență: spunem că două blocuri P și Q disjuncte sunt adiacente după direcția D ($D = N, S, E, V, NV, NE, SV, SE$), dacă

- P are o parte din latura de pe direcția D comună cu Q .
- Colțul din direcția D al blocului P este adiacent cu colțul opus al blocului Q .



Exemplu pentru adiacență într-un arbore quad. Se consideră figura 6.4: Atunci blocul F este vecin la E cu blocul G , iar blocul 38 este NE adiacent cu blocul H .

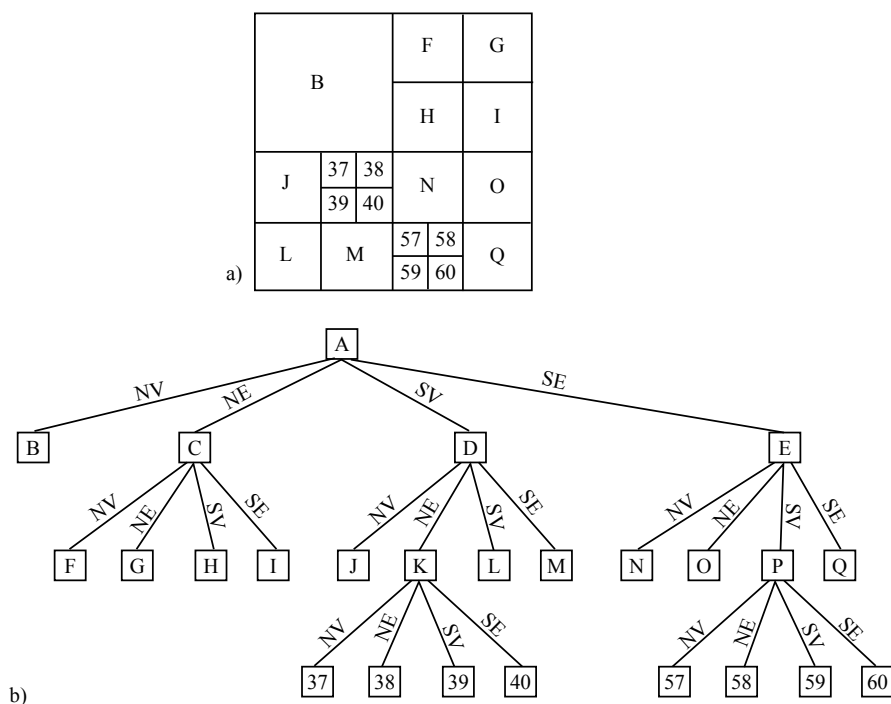


Figura 6.4: a) Suprafață descompusă în zone pătrate uniforme; b) Arborele quad corespunzător

Probleme

- (1) Determinarea vecinilor care sunt adiacenți cu întreaga latură a unui anumit bloc - de exemplu în figura 6.4, blocul I la V cu blocul H , sau blocul 37 la V cu blocul J . În acest caz problema se reduce pentru un anumit bloc P la determinarea celui mai mic bloc cu latura mai mare sau egală cu cea a lui P și care este adiacent cu P după direcția D .
- (2) Determinarea acelor vecini care sunt adiacenți numai cu un segment din latura unui anumit bloc - de exemplu în figura 6.4 blocul 57 la V cu blocul M . În acest caz vecinul căutat trebuie specificat prin informații suplimentare

În continuare prezentăm tipurile de cereri care pot fi efectuate asupra unui quad-tree pentru determinarea de vecini [15, 14, 13].

Notății:

- G = "greater then or equal"
- C = "corner"
- S = "side"
- N = "neighbor"

Cereri

- (1) $GSN(P,D)$ = cel mai mic bloc adiacent cu latura din direcția D a blocului P și care are latura mai mare sau egală cu latura lui P .
- (2) $CSN(P,D,C)$ = cel mai mic bloc care este adiacent cu latura D și colțul C al nodului P .
- (3) $GCN(P,C)$ = cel mai mic bloc adiacent cu P și aflat de partea opusă a colțului C al blocului P și care are latura mai mare sau egală cu latura lui P . **Atenție:** acest bloc trebuie să intersecteze suprafața determinată de colțul opus al colțului C .
- (4) $CCN(P,C)$ = cel mai mic bloc aflat de partea opusă a colțului C al blocului P . și în acest caz este obligatoriu ca blocul astfel determinat să se intersecteze cu suprafața determinată de colțul opus colțului C al lui P .



Exemplu: În figura 6.4:

$GSN(J, N) = B$; $GSN(N, V) = K$ (care se descompune în blocurile 37, 38, 39, 40).

$CCN(58, NV) = N$; $CCN(59, NE) = 58$.

$CSN(J, E, SE) = 39$; $CSN(Q, V, NV) = 58$.

$GCN(J, NE) = B$ NU 37; $GCN(B, SE) = E$; $GCN(58, NV) = N$.

Observații:

- GSN , CSN , GCN , CCN nu definesc relații 1 - la - 1. Pot exista mai multe blocuri care au același vecin: $GSN(H, E) = B$ și $GSN(F, E) = B$.
- Relațiile de vecinătate astfel definite nu sunt neapărat simetrice: $GSN(H, E) = B$, dar $GSN(B, V) = C$.
- Un bloc care nu se află pe marginea imaginii are minim 5 vecini. Acest lucru poate fi dedus din următoarele observații:
 - Un bloc nu poate fi adiacent cu două blocuri de dimensiuni mai mari aflate pe direcții opuse (vezi blocul 37, fig. 6.4).
 - Un bloc poate avea doar doi vecini de latură mai mare, aflați pe direcții care nu sunt opuse (ex: vest și nord). Pe celelalte două laturi și pe un colț mai sunt vecini mai mici sau egali ca dimensiune. (vezi fig. 6.4).
- Un nod are maxim 8 vecini (de dimensiuni mai mari sau egale).

În continuare vor fi considerate doar cererile GSN și GCN .

Determinarea vecinilor adiacenți după o direcție verticală / orizontală

Problematică: determinare unui bloc Q vecin al lui P , $Q = GSN(P, D)$.

Idee generală este aceea de a urca de la nodul P către primul strămoș comun cu Q , iar apoi de a coborî de la acest strămoș comun către Q .

Observații:

- Dacă direcția $D = E$, căutăm vecinul de la E , deci P se află pe ramuri SV sau NV față de strămoșul comun.
- Dacă direcția este $D = V$, căutăm vecinul la V deci P se află pe ramuri SE sau NE față de strămoșul comun.
- Dacă direcția $D = N$, căutăm vecinul de la N , deci P se află pe ramuri SV sau SE față de strămoșul comun.
- Dacă direcția $D = S$, căutăm vecinul de la S , deci P se află pe ramuri NV sau NE față de strămoșul comun

Strămoșul comun se obține când se urcă de la nodul curent către părinte pe o ramură ce nu conține direcția de căutare!



Exemplu: În figura 6.4 considerăm $GSN(N, V) = K$. Se observă că se urcă de la N către E , N fiind descendentul NV al lui E . Apoi de la E spre A , E fiind descendentul SE al lui A . În acest moment subarborele care îl conține pe K se află la **EST** față de nodul curent, care este chiar rădăcina. Deci vecinul de la **VEST** va fi pe una dintre ramurile NV sau SV ale lui A . Coborârea în arbore către vecinul căutat se face pe ramuri simetrice față de direcția D relativ la ramurile pe care s-a făcut urcarea către A .

Atenție: urcarea s-a făcut pe ramurile NV , SE , iar simetria este $D = V$, deci ramurile simetrice vor fi SV (simetricul lui SE față de verticală) și NE (simetricul lui NV față de verticală) și se observă că se ajunge la vecinul căutat K .



Exemple: de determinare a vecinilor cu algoritmul GSN în figura 6.4:

1. $GSN(39, N)$: Stiva = \emptyset , nod = (39) , părinte = (K) , $39 = K.SV$, $D = N$.

Deci ramura de urcare nu conține $N \Rightarrow$ strămoșul comun este chiar $K \Rightarrow$ se oprește bucla cât timp.

Se plasează pe stivă simetricul lui SV față de orizontală $\Rightarrow Stiva = (NV) \Rightarrow$ cobor din K pe ramura NV și ajung la nodul 37.

Din imagine se vede clar că s-a determinat vecinul dorit.

2. $GSN(59,V)$: $Stiva = \emptyset$, $nod = (59)$, $părinte = (P)$, $(59) = (P).SV$, $D = V$

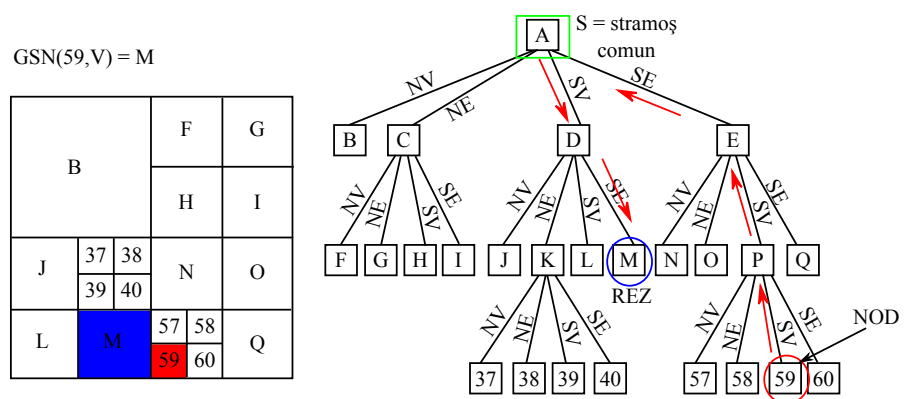
Deci ramura de urcare conține direcția $V \Rightarrow$ plasez pe stivă simetricul lui SV față de verticală $\Rightarrow Stiva = (SE)$

În plus $nod = (P)$, $părinte = (E)$ ($P = (E).SV$), deci ramura de urcare conține $V \Rightarrow Stiva = (SE, SE)$.

În plus $nod = (E)$, $părinte = (A)$ și ($E = (A).SE$). Nu mai conține direcția \Rightarrow ne oprim din ciclare.

În plus se pune pe stivă SV , deci $Stiva = (SV, SE, SE)$.

De la rădăcină se coboară pe ramurile din stivă: $A.SV = D$, $D.SE = M$ dar M este frunză deci ne oprim și vecinul căutat este M . Procesul este ilustrat în figura de mai jos:



Determinarea vecinilor diagonali

Problematică: determinare unui bloc Q vecin al lui P , $Q = GCN(P, D_1D_2)$, $D_1 \in \{N, S\}$, $D_2 \in \{E, V\}$.

Idee generală: caut un nod strămoș al lui P vecin orizontal / vertical cu un nod strămoș al nodului căutat. Acest lucru se realizează după cum urmează, considerând perechile de arce complementare: $(NV, SE)(NE, SV)$:

- se urcă de la nodul curent spre părinte, până când nodul se află pe partea $D_1D_2 \neq D_3D_4$ a părintelui. Acest părinte devine nodul curent pe care îl notăm cu Q .

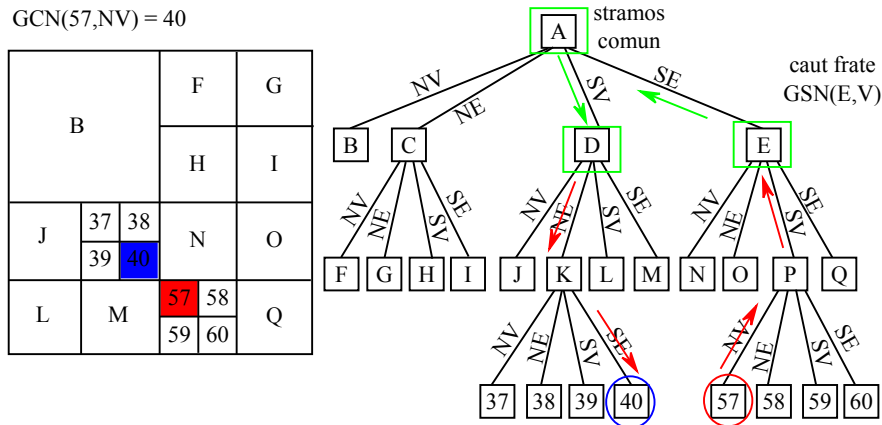
- **dacă** $D_1 \neq D_3$ și $D_2 \neq D_4$ atunci Q este strămoș comun și tot ceea ce trebuie făcut este, să se coboare din Q pe arce complementare cu cele pe care s-a urcat, adică dacă s-a urcat pe arc SV se coboară pe arc NE , dacă s-a urcat pe arc SE se coboară pe arc NV , etc.
- **altfel**
 - **dacă** $D_1 \neq D_3$ atunci caut vecinul R lui Q după direcția D_2 cu algoritmul $R = GSN(Q, D_2)$, R va deveni nodul curent.
 - **altfel** dacă $D_2 \neq D_4$ atunci caut vecinul R lui Q după direcția D_1 cu algoritmul $R = GSN(Q, D_1)$, R va deveni nodul curent.
 - Din nodul R obținut astfel se coboară pe arce complementare cu cele pe care s-a ajuns la Q .



Exemple: de determinare a vecinilor folosind algoritmul GCN pentru arborele din figura 6.4:

1. $GCN(57, NV)$:

- se urcă la părintele P pe ramura NV egală cu direcția din apel, deci se continuă la părintele lui P , E , pe ramură $SV \neq NV$
- Dar $D_1 = N$, $D_2 = V$ și $D_3 = S$, $D_4 = V$, deci $D_1 \neq D_3$ și $D_2 = D_4$ $Q = E$, rezultă căutarea vecinului R al lui Q , după algoritmul $GSN(E, V)$, adică se urcă la părinte, cât timp ramura de urcare conține V . Ramura de urcare de la E la A este SE , care nu îl conține pe V , deci algoritmul se oprește.
- Se mai urcă o dată. Părintele este $A =$ rădăcina. Acum se coboară conform algoritmului GSN pe ramura SV , adică la D , care este vecinul la V al lui E .
- Acum, conform algoritmului GCN se coboară pe o ramură opusă a lui SV , adică NE și se ajunge la K . De aici se coboară pe ramura SE și ajunge la 40.
- Din figura următoare se observă că s-a obținut exact vecinul căutat.



2. GCN(38, NE):

- Se urcă la părintele K pe o ramură NE , egală cu direcția de apel, deci se continuă urcarea la părintele lui K , D , pe o ramură NE , tot egală cu ramura de apel. Se urcă la părintele lui D , care este rădăcina A , pe o ramură $SV \neq NE$, cu $D_1 = N$, $D_2 = E$, $D_3 = S$, $D_4 = V$.
- Deci $D_1 \neq D_3$ și $D_2 \neq D_4$. S-a ajuns la un părinte comun al lui 38 cu vecinul căutat. Se coboară acum pe ramurile NE la C și SV la H . Nu mai există descendenți pentru H , deci algoritmul se oprește, iar H este vecinul căutat (vezi figura anterioară).

Aplicații: algoritmul *Split and Merge* de segmentare pentru imagini digitale.

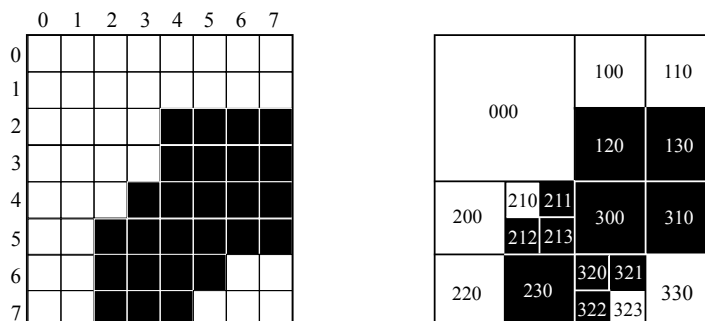
Quadtree liniar

Principala problemă a implementării clasice a unui arbore quad, adică folosind o structură de pointeri, în cazul unei imagini, este numărul mare de noduri interne necesare. De fapt, în aplicații reale, în cazul imaginilor se utilizează o structură de date numită arbori quad liniari - *Linear Quadtrees* - care reprezintă de fapt lista frunzelor în parcurgerea de la stânga la dreapta. Fiecare frunză este o structură de tip nod, care conține un câmp pentru culoare, un câmp pentru nivelul la care se află în arbore și un cod unic în baza 4.

Codul fiecărei frunze este alcătuit prin interclasarea codurilor binare ale coordonatelor y și x ale pixelului din colțul stânga-sus al blocului din imagine reprezentat prin frunză. Codul binar obținut prin interclasarea celor două coduri este transformat într-un cod în baza 4 prin gruparea cifrelor binare două câte două într-o cifră din baza 4. Algoritmul presupune imagini de dimensiuni $2^n \times 2^n$ [2, 17].



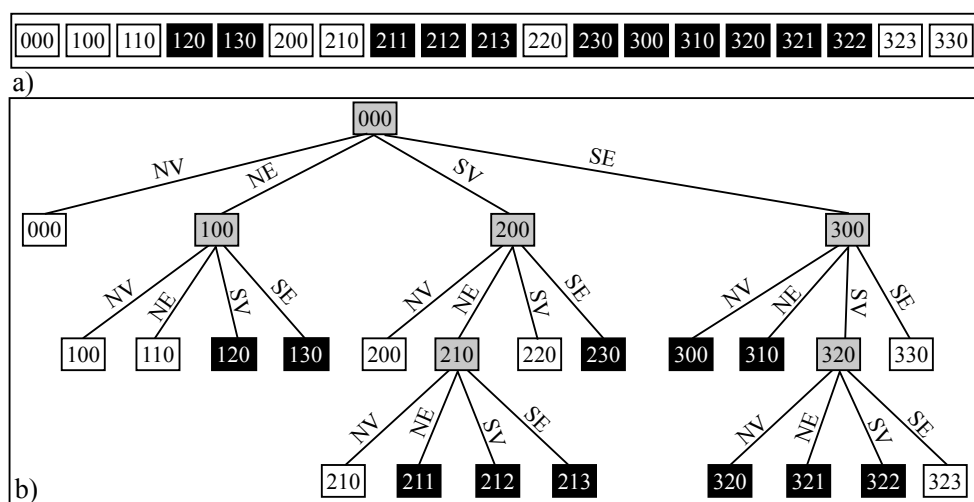
Exemple: în imaginea $2^3 \times 2^3$ din partea dreaptă a figurii:



coordonatele y, x ale colțului stânga sus:

- ale blocului B : sunt 0, 0, în binar pe trei poziții $y = 000$, $x = 000$. după interclasare codul este 000000. Grupând două câte două cifrele binare și transformând în baza 4 obținem codul 000
- ale blocului F : sunt 0, 4, în binar pe trei poziții $y = 000$, $x = 100$. după interclasare codul este 010000. Grupând două câte două cifrele binare și transformând în baza 4 obținem codul 100

Arborele liniar asociat a) și arborele desfășurat b) sunt prezentate în figura următoare. În partea stângă a figurii sunt marcate codurile fiecărei frunze.



Determinarea vecinilor într-un arbore quad liniar

- **calculul codului unui vecin:** este relativ simplă determinarea codului unui vecin de pe același nivel de descompunere, de exemplu, pentru un anumit nod vecinii de pe același

nivel care au același părinte au codul în care diferă doar cifra corespunzătoare nivelului respectiv.

- **căutarea în lista de frunze:** nu este obligatoriu să existe un vecin pe același nivel, de exemplu, blocul cu codul 210 nu are la nord vecin pe același nivel de descompunere, ci doar pe un nivel superior. Rezultă că, după calculul codului corespunzător vecinului, acesta trebuie căutat în lista de frunze.

6.2.4 Arbori quad de tip *Point Region* (PR)

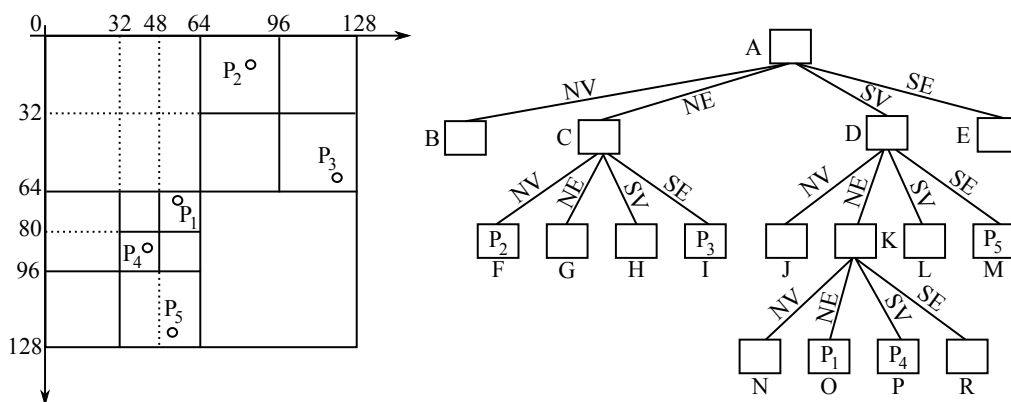


Figura 6.5: a) Suprafața pentru care se realizează împărțirea în regiuni împreună cu 5 puncte. b) Arborele PR corespunzător.

Arborii quad de tip point-region împart o suprafață pătrată în mod similar cu arborii quad pentru suprafețe/imagini, cu deosebirea că împărțirea nu se realizează pe baza proprietăților suprafeței, ci pe baza unui set de puncte plasate pe suprafața dată [15].

Idee generală: se consideră o suprafață bidimensională cu coordonatele (x, y) în domeniul $[0, dim] \times [0, dim]$ și n puncte $P_i = (x_i, y_i)$, $i = \overline{1, n}$ plasate pe această suprafață.

Suprafața se împarte recursiv în patru suprafețe egale, până când fiecare frunză conține cel mult unul dintre punctele P_i . Un exemplu este prezentat în figura 6.5.

Căutarea unui nod

Se dorește căutarea nodului în care s-ar afla punctul P de coordonate $P.x$ și $P.y$, dacă acest punct ar exista pe suprafața dată. Dacă se consideră nodul curent Z la care s-a ajuns în arbore, notăm cu $Z.stanga - sus$ colțul stânga sus al suprafeței reprezentate de Z și cu $Z.dreapta - jos$ colțul dreapta jos al acestei suprafețe. Pe baza coordonatelor acestor colțuri poate fi stabilit cadranul în care ar trebui să se afle punctul P și deci, pe ce ramură trebuie coborât în arbore.

Algorithm: PR-CAUTA**Input:** Un arbore PR T , cheia căutată P .**Output:** Pointer către nodul Z care ar putea conține P sau nil **start**

daca ($P.x < 0$ sau $P.x > T.dreapta_jos.x$ sau $P.y < 0$ sau $P.y > T.dreapta_jos.y$) **atunci**
 | returneaza(nil)

sfarsit_daca**cat_timp** $Z \neq$ frunza **executa**

| $mijloc_x = (Z.dreapta_jos.x + Z.stanga_sus.x)/2$

| $mijloc_y = (Z.dreapta_jos.y + Z.stanga_sus.y)/2$

| **daca** $P.x \leq mijloc_x$ și $P.y \leq mijloc_y$ **atunci**

| | $Z \leftarrow Z.NV$

sfarsit_daca**altfel**

| **daca** $P.x > mijloc_x$ și $P.y > mijloc_y$ **atunci**

| | $Z \leftarrow Z.SE$

sfarsit_daca**altfel**

| | **daca** $P.x \leq mijloc_x$ **atunci**

| | | $Z \leftarrow Z.SV$

sfarsit_daca**altfel**

| | | $Z \leftarrow Z.NE$

sfarsit_daca**sfarsit_daca****sfarsit_daca****sfarsit_cat_timp**returneaza(Z)**stop**

Exemplu: considerăm arborele din fig. 6.5, construit pe baza setului de puncte $\{(55, 67), (80, 15), (119, 58), (41, 85), (53, 117)\}$. Atunci **PR-CAUTA**($T, (50, 105)$):

$Z = A$ nu e frunză \Rightarrow verific în care dintre cele 4 blocuri descendente poate fi găsit $P = (50, 105)$.

$$\left. \begin{array}{l} 0 < P.x = 50 < 64 \\ 64 < P.y = 105 < 128 \end{array} \right\} \Rightarrow Z = Z.SV = D$$

D nu este frunză \Rightarrow verific \hat{c} în care dintre cele 4 blocuri descendente se plasează P .

$$\left. \begin{array}{l} 32 < P.x = 50 < 64 \\ 96 < P.y = 105 < 128 \end{array} \right\} \Rightarrow Z = Z.SE = M$$

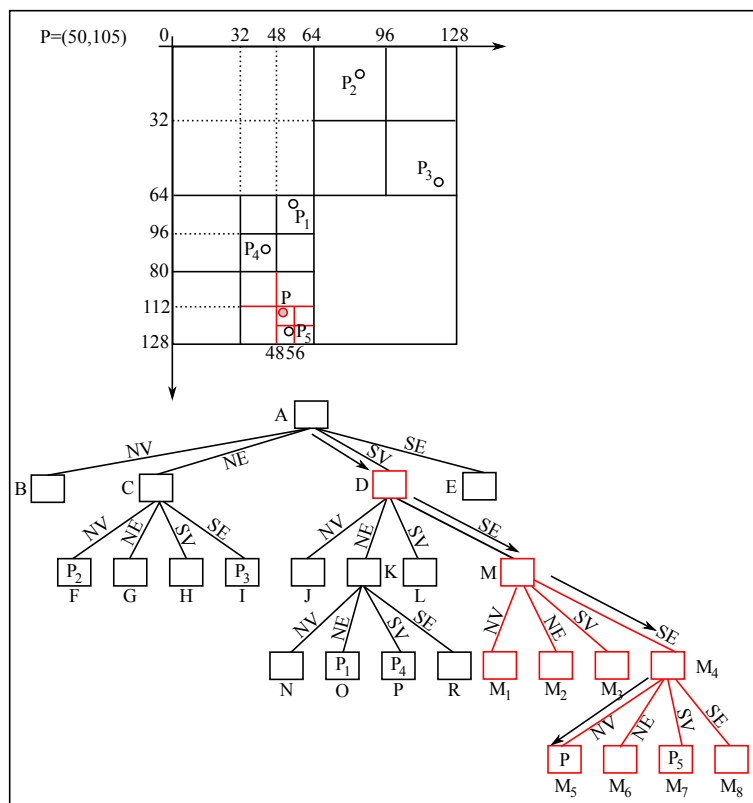
M este frunza corespunzătoare punctului $(50,105)$.

Inserarea într-un arbore PR

Dacă se dorește inserarea punctului P într-un arbore quad PR, atunci, după identificarea frunzei Z care corespunde punctului P , se verifică dacă frunza respectivă conține deja un punct, în exemplul anterior P_5 , dacă nu, atunci se inserează P în câmpul informație. Altfel (cazul exemplului) se sparge blocul corespunzător frunzei Z în 4 blocuri noi, frunza se înlocuiește cu un nod intern, care are 4 fii, corespunzători noilor blocuri obținute, se plasează punctul aflat în frunza Z în acela dintre cei 4 fii noi care este potrivit și se continuă procesul de căutare a unei frunze corespunzătoare lui P , cu eventuale noi spargerii, dacă este necesar.



Exemplu: pentru inserția nodului $P = (50, 105)$ în arborele din figura 6.5 este reprezentat în figura:



Algoritm: PR-INSEREAZA

Input: Un arbore PR T , un *punct* de inserat, dimensiunea dim a hărții.

start

```

daca  $punct.x < 0$  sau  $punct.x > dim$  OR  $punct.y < 0$  sau  $punct.y > dim$  atunci
  | scrie("în afara hărții")
  | return
sfarsit_daca
daca  $T = nil$  atunci
  | alocă memorie pentru nodul  $X$ 
  |  $X.cheie \leftarrow punct$ 
  |  $X.stanga\_sus \leftarrow (0, 0)$ ,  $X.dreapta\_jos \leftarrow (dim, dim)$ 
  |  $X.SV \leftarrow nil$ ,  $X.SE \leftarrow nil$ ,  $X.NV \leftarrow nil$ ,  $X.NE \leftarrow nil$ ,  $X.parinte \leftarrow nil$ 
  |  $T.rad \leftarrow X$ 
  | return
sfarsit_daca
 $X \leftarrow T.rad$ 
repetă
  | //alege frunza potrivită pentru  $punct$ 
  |  $Z \leftarrow PR-CAUTA(X, punct)$ 
  | daca  $Z.cheie = nil$  atunci
  | |  $Z.cheie \leftarrow punct$ 
  | sfarsit_daca
  | altfel
  | | sparge nodul  $Z$  în 4 noduri  $Z_1, Z_2, Z_3$  și  $Z_4$ 
  | | pune  $Z.cheie$  în nodul corespunzător
  | |  $Z.NV \leftarrow Z_1$ ,  $Z.NE \leftarrow Z_2$ ,  $Z.SV \leftarrow Z_3$ ,  $Z.SE \leftarrow Z_4$ 
  | |  $Z.cheie \leftarrow nil$ 
  | |  $Z_1.parinte \leftarrow Z$ ,  $Z_2.parinte \leftarrow Z$ ,  $Z_3.parinte \leftarrow Z$ ,  $Z_4.parinte \leftarrow Z$ 
  | sfarsit_daca
  |  $X \leftarrow Z$ 
pana_cand  $X = frunza$ ;
return

```

stop

Construcția unui arbore quad PR se poate realiza pe baza unui vector de puncte prin inserții succesive într-un arbore inițial vid.

Eliminarea unui punct

Pentru eliminarea unui punct P din arbore, se caută frunza Z corespunzătoare nodului P . Dacă Z nu conține P , atunci P nu există în arbore și deci nu poate fi șters. Altfel se șterge P din frunza Z .

Dacă frații lui Z sunt frunze și în blocul reprezentat prin aceste 4 frunze se află cel mult un punct Q din mulțimea de puncte ale PR-arborelui, atunci cele patru frunze se șterg, iar punctul Q se mută în părintele lui Z , care acum devine frunză.

Această etapă se repetă, până când se ajunge la un nod, ai cărui descendenți nu sunt cu toții frunze sau sunt 4 frunze care conțin împreună cel puțin 2 puncte.

Algoritm: PR-STERGE

Input: Un arbore PR T un punct P de eliminat.

begin

$Z \leftarrow \text{PR-CAUTA}(T, P)$

daca $Z.\text{cheie} = P$ **atunci**

$Z.\text{cheie} \leftarrow \text{nil}$

$Y \leftarrow Z.\text{parinte}$

$ok \leftarrow \text{adevarat}$

cat timp $Y \neq \text{nil}$ și $ok = \text{adevarat}$ **executa**

 //verifică dacă nodul Y are 4 fii (frunze) care împreună conțin cel mult

 //un punct și în caz afirmativ returnează informația aceluia fiu frunză

 //care nu este vid

$Info \leftarrow \text{POATE_FLUNIT}(Y)$

daca $Info = \text{nil}$ **atunci**

 | $ok \leftarrow \text{fals}$

sfarsit_daca

altfel

 | $Y.\text{cheie} \leftarrow Info$

 | $Y.NV \leftarrow \text{nil}, Y.NE \leftarrow \text{nil}, Y.SV \leftarrow \text{nil}, Y.SE \leftarrow \text{nil}$

 | $Z \leftarrow Y$

 | $Y \leftarrow Y.\text{parinte}$

sfarsit_daca

sfarsit_cat_timp

sfarsit_daca

altfel

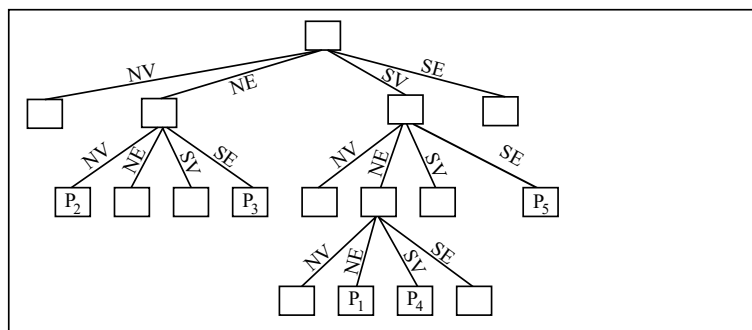
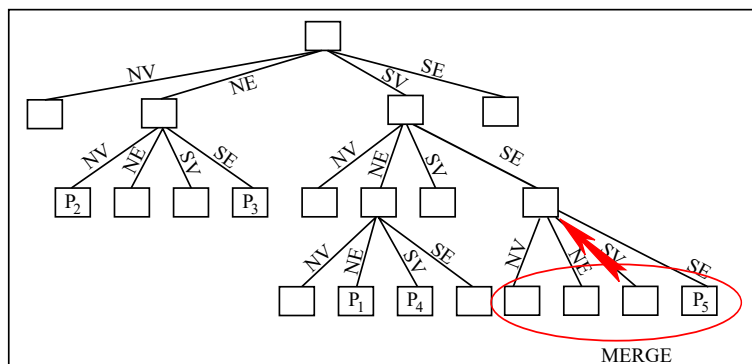
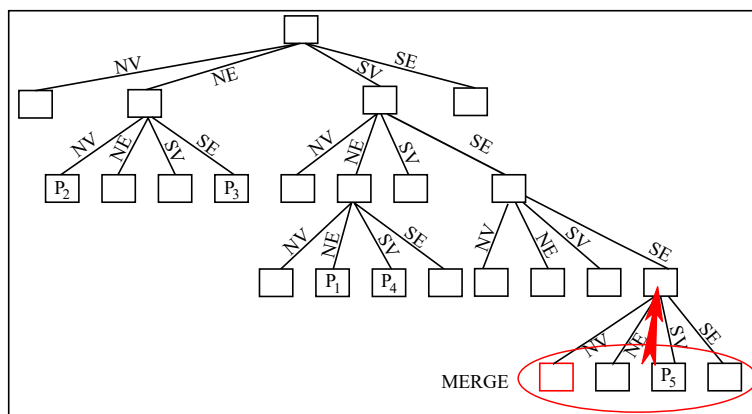
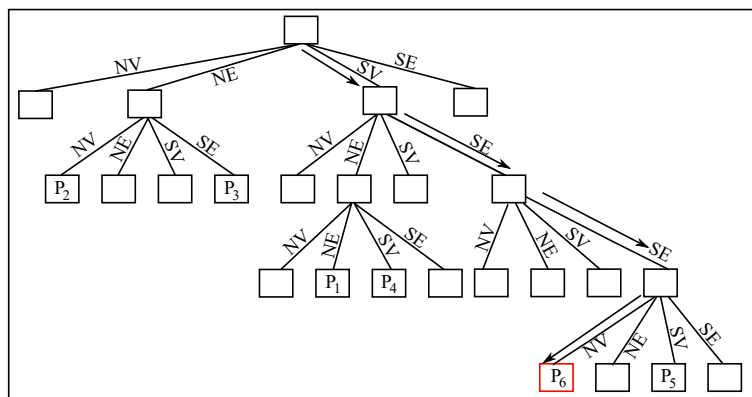
 | scrie("P nu exista")

sfarsit_daca

end



Exemplu: de ștergere a punctului $(50,105)$ din arborele care conține punctele: $\{P_1 = (55, 67), P_2 = (80, 15), P_3 = (119, 58), P_4 = (41, 85), P_5 = (53, 117), P_6 = (50, 105)\}$.



Observații

1. Forma unui PR-QuadTree nu depinde de ordinea în care sunt inserate punctele în arbore.

2. Dacă se inserează două puncte foarte apropiate este posibil să trebuiască efectuate numeroase spargeri ale blocurilor, până se ajunge la separarea celor două puncte, ceea ce conduce la creșterea în adâncime a arborelui și la introducerea unui număr semnificativ de noduri ce nu conțin nici un punct.

6.2.5 Înălțimea unui arbore quad PR

Considerând $d =$ distanța minimă între două puncte din blocul original și n dimensiunea laturii acestui bloc, atunci adâncimea h a arborelui quad PR corespunzător este:

$$\log_4 n \leq h \leq \log_2 \left(\sqrt{2}n/d \right) + 1$$

Demonstrație: evident, dacă toate frunzele sunt la aceeași adâncime, $h = \log_4 n$.

Altfel: cele mai mici blocuri, adică cele mai adânci frunze, se obțin pentru separarea celor mai apropiate două puncte dintr-un cadran (fig. 6.6). Presupunem că aceste două puncte, P_1 și P_2 , se află

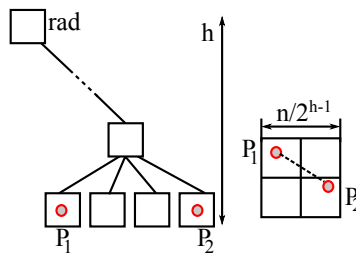


Figura 6.6: Cele mai apropiate două puncte de la cel mai de jos nivel.

la adâncimea h . Părintele din care au provenit se află la adâncimea $h - 1$ și are dimensiunea laturii $L = n/2^{h-1}$. Distanța între P_1 și P_2 este d și

$$d \leq \sqrt{2}n/2^{h-1}$$

De aici rezultă

$$h - 1 \leq \log_2 \left(\sqrt{2}n/d \right) \Rightarrow h \leq \log_2 \left(\sqrt{(2)n/d} \right) + 1$$

Determinarea vecinilor: același algoritm ca și pentru arbori quad.

Exemplu de aplicație

Se consideră un bloc pătrat de dimensiune $2^n \times 2^n$, care reprezintă o hartă, pe care se află amplasate m orașe. Acestea pot fi reprezentate cu ajutorul unui arbore quad PR. Să se determine toate orașele aflate la distanța maximă d față de un oraș dat P .

Idee de rezolvare: Se parcurge în preordine arborele. Pentru fiecare nod curent X , se verifică dacă poate conține orașe la distanța cel mult d de P . Acest lucru poate fi determinat pe baza coordonatelor colțurilor blocului $X.stanga_sus$ și $X.dreapta_jos$:

$$P.x - d \geq X.stanga_sus.x \text{ și } P.y - d \geq X.stanga_sus.y$$

$$\text{și } P.x + d \leq X.dreapta_jos.x \text{ și } P.y + d \leq X.dreapta_jos.y.$$

În caz afirmativ se continuă coborârea pe subarborele de rădăcină X , altfel se oprește căutarea în acel subarbor.

Un exemplu de astfel de căutarea este prezentat în figura 6.7.

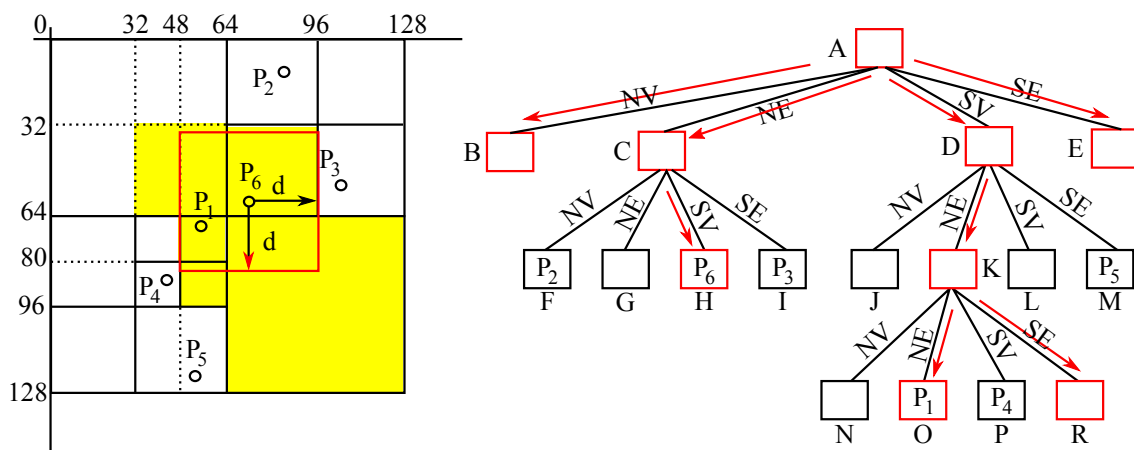


Figura 6.7: Căutarea punctelor aflate la distanța cel mult d față de un punct dat.



Să ne reamintim...

- Un arbore quad este un arbore în care fiecare nod intern are exact 4 fii.
- Arborii quad se folosesc pentru împărțirea spațiului bidimensional.
- Arborii quad pentru regiuni stochează în fiecare nod o suprafață pătrată, iar fiii reprezintă cele 4 sferturi ale acesteia, după partiționare.
- Arborii PR reprezintă partiționarea unei suprafețe în funcție de o mulțime de puncte aflate pe suprafața respectivă.

6.2.6 Rezumat

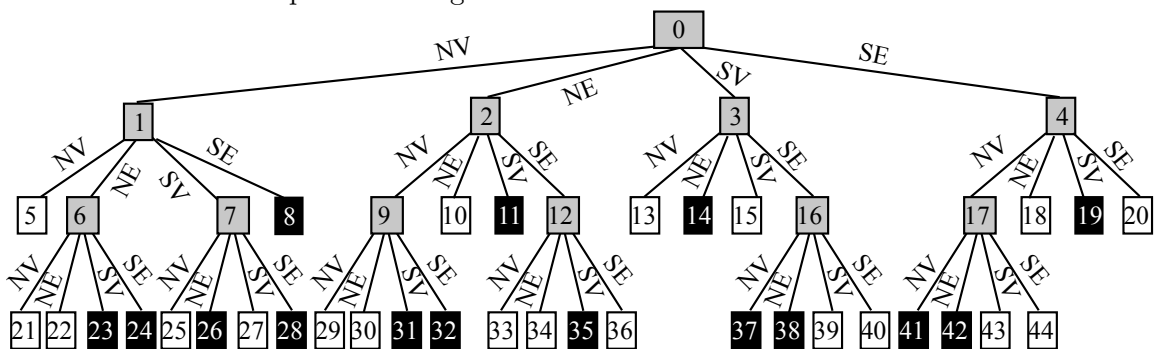


În această unitate de învățare au fost prezentați arborii quad și anume arborii quad pentru regiuni și arborii de tip *point region*. Au fost prezentate modurile de partiționare în cazul fiecărui tip de arbore quad, probleme de vecinătate și operațiile principale.



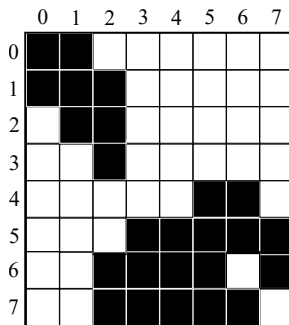
6.2.7 Test de autoevaluare

1. Se consideră arborele quad din imaginea de dimensiune $2^3 \times 2^3$:



- a. Indicați vecinul la V al lui 41. Folosiți algoritmul GSN. Pentru verificare desenați și imaginea corespunzătoare arborelui.
- b. Calculați codul în baza 4 corespunzător frunzei 37.

2. Se dă următoare imagine $2^3 \times 2^3$.

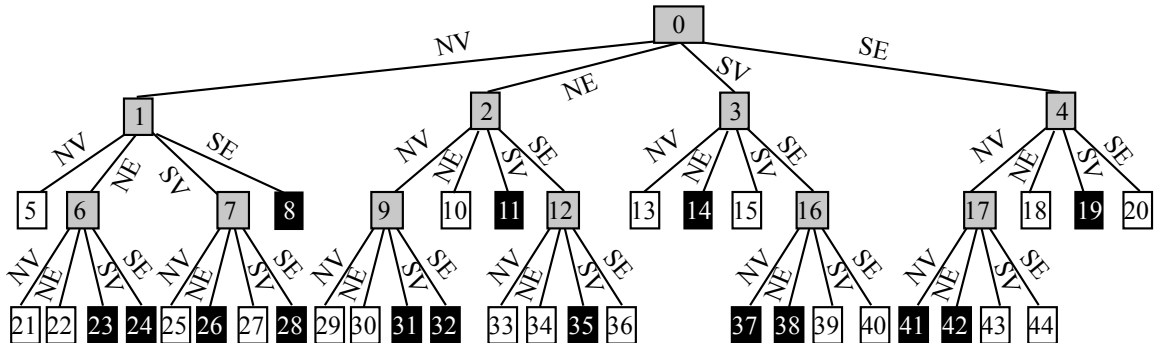


Să se deseneze arborele quad corespunzător obținut prin partiționarea în suprafețe uniforme. Să se calculeze codurile corespunzătoare pentru frunzele negre.

3. Se consideră suprafața de dimensiune 128×128 , care conține mulțimea de puncte $P = \{(20, 45), (45, 60), (100, \dots)\}$. Desenați suprafața cu punctele și construiți arborele PR asociat.

6.2.8 Răspunsuri la testul de evaluare a cunoștințelor

1. Se consideră arborele quad din imaginea de dimensiune $2^3 \times 2^3$:



- Indicați vecinul la V al lui 41. Folosiți algoritmul GSN. Pentru verificare desenați și imaginea corespunzătoare arborelui.
- Calculați codul în baza 4 corespunzător frunzei 37.

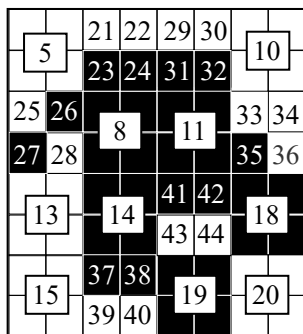
Rezolvare:

a. Se aplică algoritmul $GSN(41, V)$. Se observă că la părintele lui 41 se urcă pe o ramură de NV către nodul 17, deci pe stivă se pune NE. Se continuă urcarea spre părintele 4, tot pe ramură de NV, deci se pune NE pe stivă. În continuare se va urca pe o ramură de SE, deci nu mai apare V și nodul rădăcină 0 este părintele comun, de la care se va face coborârea către vecinul căutat, iar pe stivă se pune SV.

Stiva conține acum: $\{NE, NE, SV\}$, unde baza stivei este la stânga și vârful la dreapta.

Coborârea de la rădăcină pe o ramură SV către nodul 3. De aici se coboară pe o ramură de NE către 14 și ne oprim. Acesta este vecinul la V al nodului 41.

Imaginea asociată arborelui este:

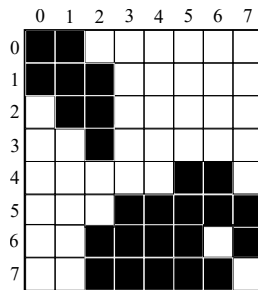


Se observă imediat, că într-adevăr vecinul la V al nodului 41 este nodul 14.

- b. Etichetele frunzelor arborelui quad din figură, nu reprezintă codurile în baza 4 corespunzătoare acestor frunze.

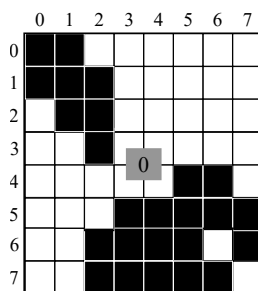
Calculăm codul corespunzător frunzei marcate cu 37. Coordonatele acestei frunze sunt $x = 2, y = 7$. Scrise în baza 2 devin $x = 010, y = 111$. Se intercalează codul lui y , cu cel al lui x și se obține codul 101110. După ce grupăm cifrele câte două și calculăm cifra corespunzătoare în baza 4 pentru fiecare grup, se obține 232.

2. Se dă următoare imagine $2^3 \times 2^3$.

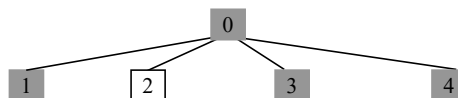
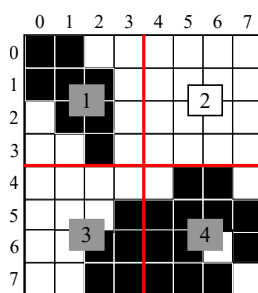


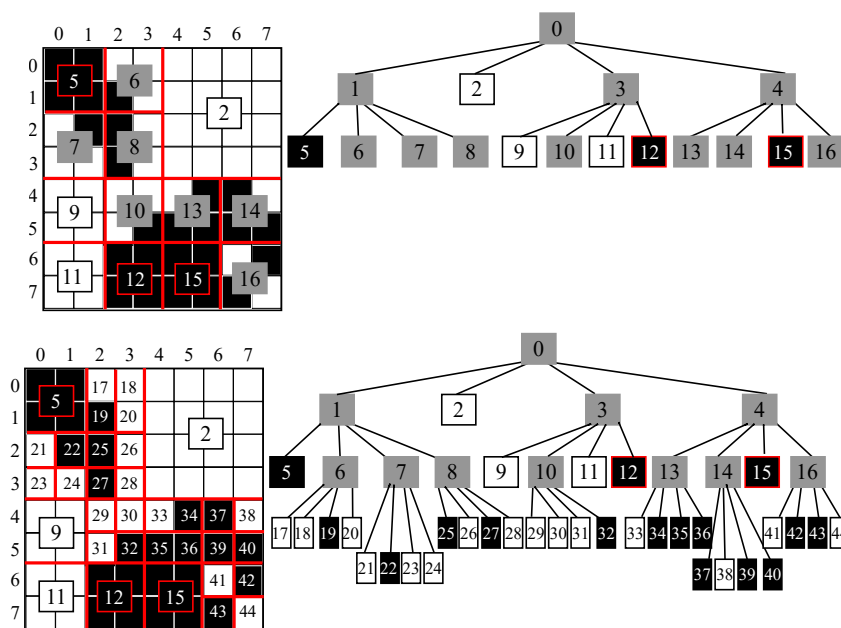
Să se deseneze arborele quad corespunzător obținut prin partiționarea în suprafețe uniforme. Să se calculeze codurile corespunzătoare pentru frunzele negre.

Rezolvare: Prezentăm în figurile de mai jos modul de împărțire a imaginii și construcția arborelui quad asociat. Au fost colorate cu gri acele noduri, care nu sunt frunze.



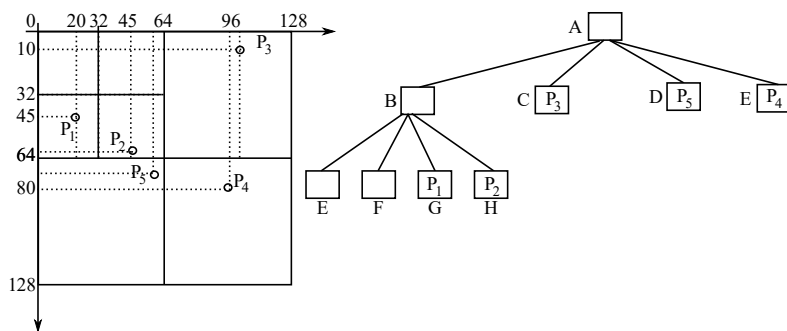
0





3. Se consideră suprafața de dimensiune 128×128 , care conține mulțimea de puncte $P = \{(20, 45), (45, 60), (100, 100)\}$. Desenați suprafața cu punctele și construiți arborele PR asociat.

Rezolvare: Suprafața de dimensiune 28×28 asociată, cu mulțimea de puncte $P = \{(20, 45), (45, 60), (100, 100), (96, 80), (60, 70)\}$, împreună cu arborele PR asociat este reprezentat în figura următoare.



Capitolul 7

ANEXA

Această secțiune conține o serie de teme de control, care presupun aplicarea practică a noțiunilor acumulate în partea teoretică a cursului. Problemele propuse presupun implementarea structurilor discutate precum și utilizarea acestora în contextul rezolvării unor probleme concrete.

Temele de control sunt structurate în funcție de capitolele teoretice. Ele vor fi rezolvate de către studenți acasă, folosind mediul de programare C++ și biblioteca SLT, precum și materialele de laborator, care vor fi disponibile pe platforma de elearning a Universității Transilvania.

Rezolvările vor fi încărcate pe platforma de elearning și vor fi prezentate personal la finalul semestrului pentru a fi notate.



7.1 Temă de control - Elemente de programare. Vectori. Matrice

1. Numim perechea (x, y) pereche ordonată dacă $x < y$.
 - (a) Să se verifice dacă un vector conține doar perechi ordonate (care satisfac condiția de mai sus).
 - (b) Să se verifice dacă oricare x din prima jumătate a vectorul formează o pereche ordonată cu oricare y din cea de-a doua jumătate.
2. Să se verifice și să se afișeze dacă un vector este superior sau inferior. Un vector este superior dacă acesta conține mai multe elemente cu valoarea mai mare decât media aritmetică a întregului vector și inferior altfel.
3. Se consideră un vector de n puncte. Fiecare punct este un element de tipul unei structuri cu două câmpuri, reprezentând coordonatele spațiale (x, y) . Să se afișeze perechile de 4 puncte care pot forma un dreptunghi. Dacă nu există astfel de pereche se va afișa un mesaj corespunzător.

4. Se consideră un teren dreptunghiular reprezentat de o matrice de dimensiuni $height \times width$. Pe acest teren pasc văcuțe și se plimbă găște. Fiecare poziție de pe teren este marcată cu 0 sau 2. Dacă este marcată cu 2, înseamnă că acolo se află două picioare. Văcuțele sunt reprezentate de două perechi picioare aflate pe poziții alăturate (pe verticală sau pe orizontală). Găștele sunt reprezentate de poziții marcate cu 2, înconjurată de 0. Să se afișeze câte văcuțe și câte găște sunt pe teren.
5. Un profesor a studiat structura relațiilor dintre elevii săi. Pentru a reprezenta această structură, profesorul a numerotat elevii de la 1 la n și a construit o matrice pătratică cu n linii astfel: $a[i, j] = 1$ dacă elevul i îl agreează pe elevul j și 0 altfel. Se consideră că fiecare elev se agreează pe sine însuși.
- (a) Afișați pe ecran elevul (elevii dacă sunt mai mulți) care are (au) cei mai mulți prieteni și cați prieteni are (au). Se consideră prieteni doi elevi care se agreează reciproc.
- (b) Există vreun elev care nu are niciun prieten?
- (c) Afișați elevii care nu sunt agreeți de nimeni.

Exemplu: se consideră 6 elevi și matricea de prietenie următoare

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

- (a) Elevul 2 are cei mai mulți prieteni (2) și anume pe elevul 1 și pe elevul 5.
- (b) Elevul 3 nu are niciun prieten. El agreează elevii 4 și 5, dar este agreeat de elevii 1 și 2.
- (c) Elevul care nu este agreeat de nimeni este elevul 6.

Observație: În exemplu numerotarea elevilor a fost făcută de la 1 la 6. În implementarea C++ aveți grijă să folosiți în matrice indici de la 0 și să ajustați numerotările.



7.2 Temă de control - Structuri elementare

1. Simulați funcționarea unei cozi folosind două stive. Scrieți un program C++ în care să creați o structură *stiva*, cu funcții de tip *push*, *pop*, *empty()* și *clear()*, iar cele două cozi vor fi membrii al structurii, iar funcțiile stivei vor avea la bază funcțiile cozilor. Puteți folosi implementări proprii sau elemente de STL, discutate la laborator.

- Implementați o listă dublu înlănțuită, cu funcții de tip *push_front*, *push_back*, *pop_front*, *pop_back*, *find*, *erase*, *empty*, *clear*. *size*.

De asemenea adăugați funcțiile:

- *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.
- *remove(int key)* - șterge toate aparițiile cheii *key* (implică parcurgerea listei)

- Inversare elemente vector.** Să se inverseze elementele unui vector utilizând o stivă. Primul element se va interschimba cu ultimul, al doilea cu penultimul, etc.
- Parantezare corectă:** Se dă un șir de paranteze deschise și închise de tip (,), [,], {, }. Să se verifice dacă șirul este corect, adică fiecare paranteză deschisă este închisă apoi, iar paranteze de ordin mai mare nu sunt incluse în paranteze de ordin mai mic sau de același ordin. Folosiți o stivă (*std::stack <char >*) pentru rezolvare. **Exemplu:** șirul `[]()` este corect, șirul `[]]` nu este corect, șirul `()]` (nu este corect).
- Implementați algoritmi de la secțiunea de probleme rezolvate, de capitolul 1, unitatea 2.



7.3 Temă de control - Tabele de dispersie

- Implementați o structură / clasă de tip tabelă de dispersie care stochează numere naturale și care rezolvă coliziunile prin liste înlănțuite. Folosiți pentru selectarea poziției de inserție metoda multiplicare-deplasare, iar pentru implementare o abordare orientă obiect.
- Palindrom.** Se consideră un vector de caractere (citit din fișier). Să se scrie o funcție care are ca parametru acest șir și care returnează un palindrom format cu toate caracterele șirului, dacă acest lucru este posibil sau un șir vid altfel. Implementați eficient folosind **unordered_map** din STL.
- Permutări.** Se consideră două șiruri de caractere (citite din fișier). Să se scrie o funcție care are ca parametru cele două șiruri și care returnează *true* dacă al doilea este o permutare a primului și *false* altfel. Implementați folosind **unordered_map** din STL.
- Concurenți.** Se consideră un număr de competiții sportive la care s-au înscris concurenți. Pentru fiecare competiție există o listă cu numele și prenumele concurenților (pereche de valori de tip *std::string*). Aceste liste se citesc dintr-un fișier. Să se scrie o funcție care indică persoanele care participă la mai mult de o singură competiție.



7.4 Temă de control - Arbori și heap-uri

1. **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 dreپți pentru fiecare nod.
- Citire pe niveluri (în lățime).
- Printr-un vector de tați.

Să se implementeze următoarele funcții pentru arbore:

- O funcție ce calculează înălțimea unui subarbore.
- Funcții pentru parcurgerea arborelui (RSD, SRD, SDR, niveluri).
- O funcție care afișează frunzele de la stânga la dreapta.
- O funcție care verifică dacă doi arbori sunt identici.
- O funcție care verifică dacă un arbore e complet.
- O funcție care determină adâncimea unui nod.

2. **Sortare** Să se implementeze algoritmul **Heap-Sort**. Să se sorteze un vector de numere folosind apoi algoritmul.

3. **Coadă de priorități.** Implementați o coadă de priorități folosind o structură (clasă) `PRIORITY_QUEUE`, care și care să dispună de toate funcțiile necesare, descrise în partea teoretică.

4. **Interclasarea optimală.** Se consideră n vectori de numere întregi sortați crescător. Să se interclaseseze acești vectori într-unul singur cu număr minim de comparații. Folosiți o coadă de priorități.

Bibliografie

- [1] G. M. Adelson-Velsky și E. M. Landis, “An algorithm for the organization of information”, în *Soviet Math. Dokl.* 3 (1962), pp. 1259–1263.
- [2] Kunio Aizawa și Shojiro Tanaka, “A Constant-Time Algorithm for Finding Neighbors in Quadtrees”, în *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31.7 (2009), pp. 1178–1183, DOI: 10.1109/TPAMI.2008.145.
- [3] Peter Brass, *Advanced Data Structures*, Cambridge University Press, 2008.
- [4] Thomas H Cormen et al., *Introduction to algorithms*, MIT press, 2022.
- [5] Martin Dietzfelbinger et al., “A reliable randomized algorithm for the closest-pair problem”, în *Journal of Algorithms* 25.1 (1997), pp. 19–51.
- [6] R. A. Finkel și J. L. Bentley, “Quad trees: A data structure for retrieval on composite keys”, în *Acta Informatica* 4.1-9 (1974).
- [7] Glenn Fowler et al., “The FNV non-cryptographic hash algorithm”, în *IETF-draft: Fremont, CA, USA* (2011).
- [8] Leonidas J. Guibas și Robert Sedgwick, “A dichromatic framework for balanced trees”, în *19th Annual Symposium on Foundations of Computer Science (SFCS)*, IEEE, 1978, pp. 8–21.
- [9] Catherine Hayes și David Malone, “An Evaluation of FNV Non-Cryptographic Hash Functions”, în *2024 35th Irish Signals and Systems Conference (ISSC)*, IEEE, 2024, pp. 1–8.
- [10] Donald Ervin Knuth, *The Art of Computer Programming. Fundamental Algorithms. Third Edition*, vol. 1, Addison Wesley, 1997.
- [11] Joseph M. Newcomer, *An In-Depth Study of the STL Deque Container*, Accessed: 2025-02-27, 2004, URL: <https://www.codeproject.com/Articles/5425/An-In-Depth-Study-of-the-STL-Deque-Container>.
- [12] Jakub PACHOCKI și Jakub RADOSZEWSKI, “Where to Use and How not to Use Polynomial String Hashing.”, în *Olympiads in Informatics* 7 (2013).

-
- [13] Hanan Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS*, Reading, MA: Addison-Wesley, 1990, ISBN: 0-201-50300-0.
- [14] Hanan Samet, *Foundations of Multidimensional and Metric Data Structures*, Morgan Kaufmann, 2006.
- [15] Hanan Samet, “The quadtree and related hierarchical data structures”, în *ACM Computing Surveys (CSUR)* 16.2 (1984), pp. 187–260, DOI: 10.1145/356924.356930.
- [16] Robert Sedgewick și Kevin Wayne, *Algorithms*, a 4-a ed., Addison-Wesley, 2011.
- [17] C. A. Shaffer și H. Samet, “Optimal quadtree construction algorithms”, în *Computer Vision, Graphics, and Image Processing* 37.3 (1987), pp. 402–419, DOI: 10.1016/S0734-189X(87)80038-6.
- [18] Mark Allen Weiss, *Data Structures and Algorithm Analysis in C++*, a 4-a ed., Pearson, 2013.
- [19] Baptiste Wicht, *C++ Benchmark: std::vector vs std::list*, Accessed: 2025-02-18, 2012, URL: <https://baptiste-wicht.com/posts/2012/11/cpp-benchmark-vector-vs-list.html>.
- [20] Michał Wrona, *Performance of Array vs Linked List on Modern Computers*, Accessed: 2025-02-18, 2021, URL: <https://dzone.com/articles/performance-of-array-vs-linked-list-on-modern-comp>.