MODULUL 1 - STRUCTURI DE DATE LINIARE

Datele compuse sau structurile de date sunt colecţii de date în care există anumite relaţii structurale. Fiecare element al unei structuri de date are o anumită poziţie în cadrul structurii şi există un mod specific de acces al acestui element.

Structurile de date sunt utilizate pentru memorarea şi manipularea eficientă cu calculatorul a unor mulţimi dinamice de date. Denumirea de *mulţimi dinamice* provine de la faptul că acestea suferă modificări prin operaţii de inserare şi/sau extragere de elemente.

Reprezentarea elementelor unei mulţimi

În numeroase situaţii un element al unei mulţimi dinamice de date este reprezentat printr-o structură / un obiect, care dispune de mai multe câmpuri, dintre care unele conţin informaţii de interes, iar altele referinţe către alte elemente ale mulţimii de date.

În anumite cazuri se consideră unul dintre câmpuri ca fiind o cheie ce identifică în mod unic elementul.

În cazul cheilor provenind din mulţimi bine ordonate (ex: numere întregi, caractere, şiruri de caractere) pot fi efectuate operaţii de ordonare / sortare sau de determinare a elementului cu cheia minimă respectiv maximă din mulţimea dinamice păstrată în structura de date respectivă.

Operaţiile efectuate asupra unei mulţimi dinamice de date pot fi de două tipuri:

1. Cereri:
   - $CAUTA(S, k)$ - returnează un elementul x din mulţimea structura S, pentru care $x.key = k$, (adică x are cheia k) dacă există un astfel de element în S, sau NULL altfel.
   - $MINIM(S)$ - returnează elementul cu cheia minimă din S.
   - $MAXIM(S)$ - returneazu a elementul cu cheia maximă din S.
   - $SUCCESOR(S, x)$ - returnează elementul din S cu cea mai mică cheie mai mare decât cheia lui x. Dacă x este maximul din S, atunci returnează NULL.

- $PREDECESOR(S, x)$ - returnează elementul din S cu cea mai mare cheie mai mică decât cheia lui x. Dacă x este minimul din S, atunci returnează NULL.

2. Operaţii de modificare a mulţimii:
   - $INSERTIE(S, x)$ - Inserează elementul x în mulţimea / structura S.
   - $STERGERE(S, x)$ - Elimină x din mulţimea / structura S.

O structură de date se numeşte liniară, dacă elementele sale alcătuiesc o secvenţă. O astfel de structură are un prim element şi un ultim element, iar trecerea de la un element la următorul se face în mod secvenţial şi depinde de modul de parcurgere respectiv de disciplina de intrare/ieşire a structurii. Pe baza modului de manipulare a elementelor structurii, respectiv pe baza disciplinei de intrare/ieşire distingem mai multe tipuri de structuri liniare, de exemplu stive, cozi, liste simplu sau dublu înlănţuite.

1 Stiva - *stack*

Definiţie: Stiva este o structură liniară de tip **LIFO** - *Last In First Out*, adică ultimul element introdus va fi primul care se extrage pentru prelucrare. Accesul la elementele stivei se realizează doar prin vârful stivei, unde se află ultimul element introdus.

Reprezentarea în memorie:

- - utilizând un tablou liniar - *array*
- - utilizând o listă înlănţuită

1. Reprezentarea secvenţială - tablou liniar

Se utilizează pentru stiva S

- un vector *data* de dimensiune dim = nr. maxim de elemente ce pot fi introduse.
- o variabilă vf ce reprezentă vârful stivei = poziţia în vector pe care se află ultimul element aparţinând stivei

Observaţii:

- Dacă $vf = -1$ atunci stiva este vidă şi nu pot extrage elemente
- Dacă $vf = dim - 1$ stiva este plină şi nu pot adăuga elemente noi.

Un exemplu de stivă reprezentată secvenţial este prezentat în figura 1. Stiva conţine 7 elemente, iar variabila $vf = 6$.

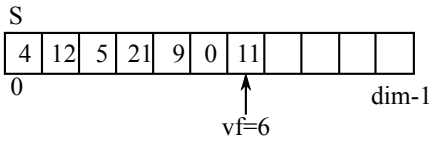


Figure 1: Stivă reprezentată printr-un vector..

Considerând structura STIVA cu atributele *data* - vector de chei, *dim* - dimensiunea maximă a stivei şi vf -poziţia vârfului, putem utiliza următoarele funcţii de adăugare şi eliminare a unui element din stivă S.

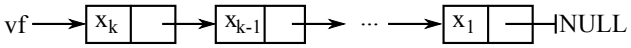
```
PUSH(S, x)
    daca S.vf = S.dim - 1 atunci
         scrie ("Stiva este plina!")
    altfel
         S.vf = S.vf + 1
         S.data[S.vf] = x
    sfarsit daca
RETURN
```

```
POP(S)
   daca S.vf = -1 atunci
        scrie("Stiva este vida")
   altfel
        S.vf = S.vf - 1
   sfarsit daca
RETURN
```

Complexitate: ambele operaţii au complexitatea $O(1)$.

2. Reprezentare dinamică

Pentru fiecare element poate fi utilizată o structură care conţine 2 câmpuri: INFO - pentru informaţie şi NEXT - pointer către elementul următor. Stiva poate fi reprezentată grafic astfel:



Funcţiile de adăugare şi eliminare a elementului din vârful stiveri S sunt:

```
PUSH(S, x)
    aloca memorie pentru y
    y.info = x
    y.next = S.vf
    S.vf = y
```

```
RETURN

 POP(S)
    daca S.vf = NULL atunci
         scrie ("Stiva este vida")
    altfel
         y = S.vf
         S.vf = S.vf.next
         dealoca memoria alocata lui y
    sfarsit daca
RETURN
```

Complexitate: ambele operaţii au complexitatea $O(1)$.

2 Coada -*queue*

Definiţie: Coada este o structură liniară de tip **FIFO** - *First In First Out*, adică primul element introdus va fi primul care se extrage pentru prelucrare. Coada modelează procese care presupun formarea de cozi, de exemplu deservirea clienţilor la un ghişeu. De asemenea se utilizează cozi pentru operaţii precum parcurgerea în lăţime a unui graf sau a unui arbore.

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

Reprezentare

O coadă, ca şi o stivă, poate fi construită în două moduri:

- secvenţial - cu ajutorul unui tablou liniar - **array**
- dinamic - cu ajutorul unei liste înlănţuite

1. Reprezentarea secvenţială

Se utilizează pentru coada C

- un vector *data* de dimensiune dim = nr. maxim de elemente ce pot fi introduse.
- două variabile *prim* şi *ultim* care reprezentă poziţia în vector pe care se află primul, respectiv ultimul element aparţinând cozii. Dacă $ultim < prim$ atunci coada este vidă şi nu pot fi extrase elemente, dacă $ultim = dim - 1$ coada este plină şi nu pot fi adăugate elemente noi.

Un element nou se adaugă mereu după ultimul element şi se creşte variabila *ultim*, iar un element se extrage din coadă prin creşterea variabilei *prim*. De fapt, elementul respectiv nu se şterge efectiv din memorie, dar nu mai este considerat ca făcând parte din coadă. Acest lucru este ilustrat în fig. 2.

La inserţie trebuie verificat dacă este plină coada, iar la ştergere, dacă este goală.

Observaţie: la fiecare extragere şi adăugare, coada migrează înspre dreapta. Astfel se poate ajunge la situaţia în care în vectorul de date corespunzător cozii sunt multe poziţii neocupate, dar totuşi la apelarea functiei PUSH se primeşte mesaj de coadă plină - fig. 2. Acest lucru se evită utilizând cozi circulare!

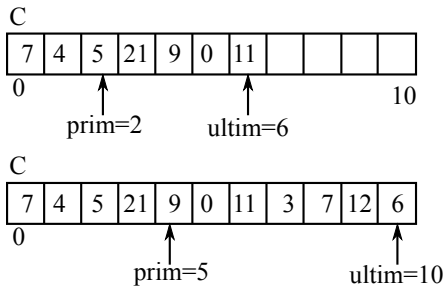


Figure 2: Din coada C s-au extras elementele 5 şi 21 şi s-au adăugat elementele 3, 7, 12, 6. În acest moment primele 4 poziţii sunt considerate neocupate, dar funcţia PUSH returnează mesajul de coadă plină.

Coada circulară

În cazul unei cozi circulare, atunci când variabila $ultim = dim - 1$, la următoarea inserţie *ultim* 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 coadă plină doar atunci când sunt ocupate toate poziţiile alocate în vector pentru elementele cozii.

Condiţiile de coadă vidă/plină

În cazul cozilor circulare condiţiile de coada plină şi 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.3.
- Evident condiţia $ultim = dim - 1$ nu mai generează mesajul de coadă plină

Soluţionarea problemei detecţiei cozii vide sau pline: Această problemă se rezolvă în modul următor. O poziţie din vectorul *data* indicată de elementul *ultim* nu va fi ocupată niciodată. Ea este utilizată doar pentru marcarea sfârşitului cozii. De fapt poziţia indicată de *ultim* reprezintă prima poziţile liberă în vector, după ultimul element din coadă. Condiţiile de coadă vidă / coadă plină sunt în acest caz:

- Dacă $prim = ultim$ atunci coada este vidă
- Dacă $ultim + 1 = prim$ atunci coada este plină. Atentie: cand $ultim = dim - 1$ coada este plină dacă $prim = 0$.

În figura 3 este reprezentat un exemplu de coadă circulară.

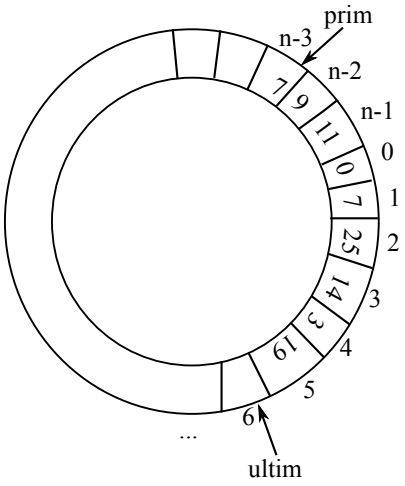
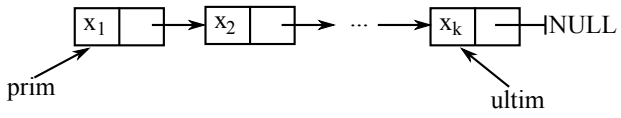


Figure 3: Coadă circulară în care la momentul curent variabila *prim* indică poziţia $n - 3$ şi variabila *ultim* indică poziţia neocupată 6.

2. Reprezentare dinamică

Pentru fiecare element poate fi utilizată o structură care conţine 2 câmpuri: INFO - pentru informaţie şi NEXT - pointer către elementul următor. Coada poate fi reprezentată grafic astfel:



Alocarea memoriei pentru coadă se face dinamic, nu static, aşa 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* şi *ultim*, în care se reţin adresele primului respectiv ultimului element din coadă. Elemente noi se adaugă mereu după ultimul element iar la eliminare se consideră primul element din coadă!

Algoritmi inserţie / eliminare

```
 PUSH(C,x)
      aloca memorie pentru y
      y.info = x
      y.next = NULL
      daca C.prim = NULL atunci //coada vida
            C.prim = C.ultim = y
      altfel
            C.ultim.next = y
            C.ultim = y
      sfarsit daca
RETURN
```

```
 POP(C)
     daca C.prim = NULL atunci
          scrie ("Coada este vida")
     altfel
          y = C.prim
          C.prim = C.prim.next
          sterge y
     sfarsit daca
RETURN
```

3 Liste înlănţuite

3.1 Liste simplu înlănţuite

O listă simplu înlănţuită este o structură de date liniară în care fiecare element conţine un câmp de legătură către elementul următor din listă, câmp pe care îl vom nota prin *next*.

De fapt stivele şi cozile alocate dinamic sunt cazuri particulare de liste simplu înlănţuite cu o anumită disciplină de intrare/ieşire.

Comparaţie cu stive/cozi: Spre deosebire de stive şi cozi, listele în general permit inserarea respectiv ştergerea în orice poziţie a listei. De asemenea suportă operaţia de căutare a unei chei.

Acces: Lista se accesează prin capul listei, reprezentând primul element. Astfel este necesară o variabilă pentru păstrarea adresei capului listei. Notăm în continuare capul listei prin *head*.

Un exemplu de listă simplu înlănţuită este prezentat în figura 4.

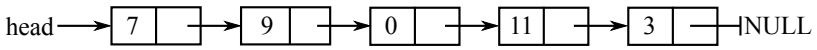


Figure 4: Listă simplu înlănţuită.

Algoritmii pentru implementarea operaţiilor cu liste înlănţuite sunt prezentaţi în continuare.

1. Căutarea unui element

```
CAUTA_LISTA(L,k)
      x = L.head
      cat timp x ≠ NULL si x.info ≠ k
               x = x.next
      sfarsit cat timp
RETURN x
```

Complexitate: În cel mai defavorabil caz, atunci când k nu se găseşte în listă, complexitatea este $O(n)$, unde n = lungimea listei.

2. Inserarea elementului x în listă se realizează la începutul listei

```
INSERTIE_LISTA(L, x)
         x.next = L.head
         L.head = x
RETURN
```

Complexitate: $O(1)$

3. Ştergerea unui element din listă - se presupune ca elementul există în listă şi a fost identificat prin funcţia CAUTA_LISTA.

```
STERGERE_LISTA(L, x)
        daca x = L.head atunci
             L.head = L.head.next
        altfel
             y = L.head
             cat timp y.next ≠ x
                       y = y.next
             sfarsit cat timp
             y.next = x.next
        sfarsit daca
RETURN
```

Complexitate: $O(n)$ - presupune căutarea predecesorului lui x. Acest lucru se evită în cazul listelor dublu înlănţuite.

3.2 Liste dublu înlănţuite

Listele dublu înlănţuite sunt structuri liniare de date în care fiecare element posedă atât o legătură către elementul predecesor *prev* cât şi una către elementul următor *next*. Accesul în listă se realizează prin capul listei către care pointează o variabilă *head*.

Un exemplu de listă dublu înlănţuită este prezentat în figura 5. În cele ce urmează sunt prezentaţi algoritmii pentru operaţiile de căutare, inserţie şi ştergere a unui element din listă.

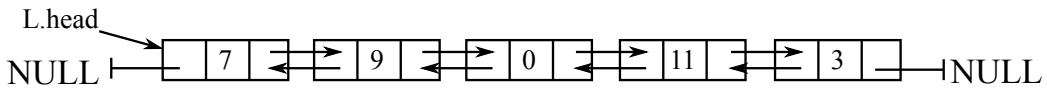


Figure 5: Listă dublu înlănţuită.

1. **Căutarea unui element**: la fel ca pentru liste simplu înlănţuite

```
CAUTA_LISTA_D(L, k)
      x = L.head
      cat timp x ≠ NULL si x.info ≠ k
               x = x.next
      sfarsit cat timp
RETURN x
```

2. **Inserarea** elementului x in listă se realizează la începutul listei

```
INSERTIE_LISTA_D(L, x)
         x.next = L.head
         daca L.head ≠ NULL atunci
```

```
            L.head.prev = x
        sfarsit daca
        x.prev = NULL
        L.head = x
RETURN
```

Complexitate: $O(1)$

3. Ştergerea unui element din listă - se presupune ca elementul există în listă şi a fost identificat prin funcţia de căutare.

```
STERGERE_LISTA_D(L, x)
        daca x.prev ≠ NULL atunci
            x.prev.next = x.next
        altfel
            L.head = x.next
        sfarsit daca
        daca x.next ≠ NULL atunci
            x.next.prev = x.prev
RETURN
```

Complexitate:

- Stergerea efectiva: O(1)
- Dacă elementul x trebuie întâi căutat: O(n)

MODULUL 2 - GRAFURI. ARBORI RĂDĂCINĂ

1 Grafuri - noţiuni de bază

1.1 Definiţii şi notaţii

Definiţie - graf orientat: Se numeşte *graf orientat* o pereche ordonată $G = (N, A)$, în care N reprezintă o mulţime finită de elemente numite noduri sau vârfuri, iar $A = \{a | a = (x, y), x, y \in N\}$ o mulţime de de perechi ordonate de elemente din N care se numesc arce. Numărul de noduri $n = |N|$ se numeşte *ordinul grafului*. Un arc $a = (x, x)$ se numeşte buclă.

Un exemplu de graf orientat este prezentat în figura 1.

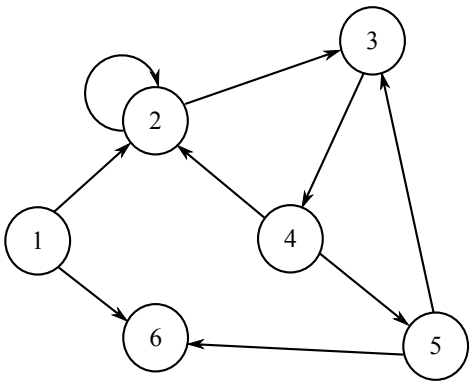


Figure 1: Exemplu de graf orientat

Definiţie - extremităţi: Se consideră un graf orientat $G = (N, A)$ şi $a = (x, y) \in A$ o muchie. Atunci

- x, y se numesc *extremităţi* ale arcului a
- x se numeşte *extremitate iniţială* a arcului şi predecesor al nodului y
- y se numeşte *extremitate finală* a arcului a şi succesor al nodului x
- x şi y se numesc *noduri adiacente*.

Notaţii:

$V^+(x) = \{y|(x,y) \in A\}$ = mulţimea succesorilor lui x

$V^-(x) = \{y|(y,x) \in A\}$ = mulţimea predecesorilor lui y

$V(x) = V^+(x) \cup V^-(x)$ = mulţimea vecinilor lui x

Exemplu: În graful din fig. 1 avem $V^+(4) = \{2,5\}, V^-(4) = \{3\}$

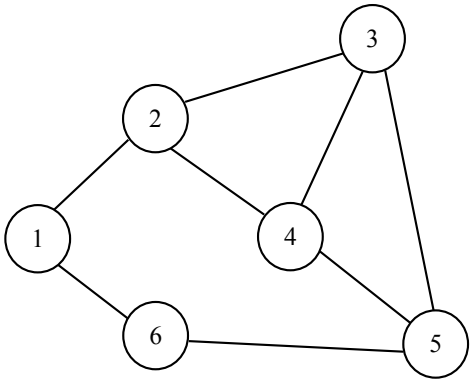


Figure 2: Exemplu de graf neorientat

Definiţie - graf neorientat: Se numeţe *graf neorientat* o pereche ordonată $G = (N, A)$ formată dintr-o mulţime de noduri N şi o mulţime de perechi neordonate $A = \{a|a = [x,y] = [y,x], x,y \in N\}$.

În cazul unei muchii $[x,y]$ dintr-un graf neorientat noţiunile de predecesor şi succesor îşi pierd semnificaţia. Se păstrează semnificaţia noţiunilor de noduri adiacente şi extremităţi. Mulţimea vecinilor lui x în cazul unui graf neorientat este:

$$V(x) = \{y|[x,y] \in A\}.$$

În figura 2 este prezentat un exemplu de graf neorientat.

Definiţie - lanţ: Fie graful orientat $G = (N, A)$. Se numeşte *lanţ* de la nodul x_1 la nodul x_{k+1} o secvenţă $L = (x_1, a_1, x_2, a_2, \ldots, a_k, x_{k+1}), x_i \in N$ pentru $i = \overline{1, k+1}$, $a_i \in A$, pentru $i = \overline{1,k}$ şi $a_i = (x_i, x_{i+1})$ sau $a_i = (x_{i+1}, x_i)$. x_1 şi x_{k+1} se numesc extremităţile lanţului.

- Dacă $a_i = (x_i, x_{i+1})$, atunci a_i se numeşte *arc direct*.
- Dacă $a_i = (x_{i+1}, x_i)$, atunci a_i ne numeşte *arc invers*.
- Dacă $x_1 = x_{k+1}$, atunci L se numeşte *ciclu*.

Putem reprezentat lanţul L în mod simplificat $L = (a_1, a_2, \ldots, a_k)$.

Exemplu: În graful orientat din figura 1 $L = \{(1,2),(2,3),(5,3),(4,5),\ (3,4)\}$ este un lanţ în care arcele $\{(1,2),(2,3)\}$ sunt arce directe, iar $\{(5,3),\ (4,5),(3,4)\}$ sunt arce inverse.

Definiţie - drum Fie $G = (N, A)$ un graf orientat. Se numeşte *drum* în G un lanţ $L = (a_1, a_2, \ldots, a_k)$ în care toate arcele sunt arce directe. Un ciclu care este drum se numeşte *circuit*.

Exemplu: În graful din figura 1 $L_1 = \{(1,2),(2,3),(3,4),(4,2)\}$ este drum, iar $L_2 = \{(2,3),(3,4),(4,2)\}$ este circuit.

Observaţie: noţiunea de drum se defineşte şi în cazul grafurilor neorientate.

Definiţie - graf ponderat: Se numeşte *graf orientat ponderat* sau *reţea orientată* un graf orientat $G = (N, A)$ pentru care se defineşte în plus o funcţie $P : A \to \mathbb{R}$ prin care fiecărei muchii (x, y) i se asociază o pondere/un cost $p(x, y)$.

1.2 Reprezentarea grafurilor

Există două moduri de reprezentare a unui graf orientat $G = (N, A)$ cu $N = \{x_1, x_2, \ldots, x_n\}$ şi A mulţimea muchiilor:

(1) Printr-o matrice de adiacenţă $M = (m_{ij})_{i,j\in 1,n}$ corespunzătoare grafului care se defineşte prin:

$$m_{ij} = \begin{cases} 1, & (x_i, x_j) \in A \\ 0, & \text{altfel} \end{cases}$$

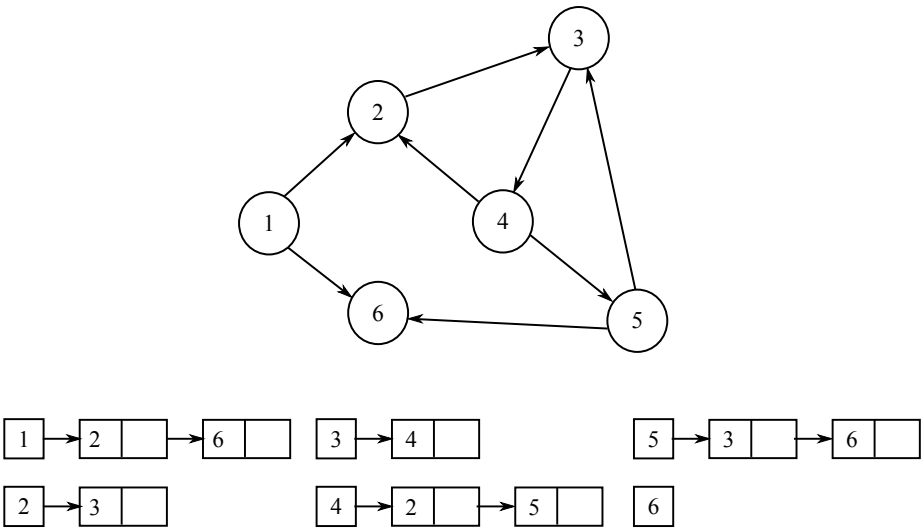
(2) Prin liste de adiacenţă. Pentru fiecare nod $x_i \in N$ se construieşte o listă de adiacenţă (reprezentată printr-o listă înlănţuită) care conţine toate nodurile din $V^+(x_i)$.

Exemplu: Se consideră graful orientat din figură.

(1) Matricea de adiacenţă:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

(2) Liste de adiacenţă:



Observaţii:

- În cazul unui graf neorientat matricea de adiacenţă este simetrică.
- În cazul unui graf cu multe noduri şi relativ puţine muchii, reprezentarea prin liste de adiacenţă este mai compactă şi de obicei preferată.
- În anumiţi algoritmi, în care este utilă determinarea rapidă a existenţei unei muchii între două noduri, se preferă reprezentarea sub formă de matrice de adiacenţă (ex. Alg Floyd-Warshall).

2 Arbori oarecare

2.1 Definiţii şi notaţii

Definiţie - graf conex: Un graf $G = (N, A)$ se numeşte *conex*, dacă pentru oricare $x, y \in N$, $x \neq y$, există un lanţ care are ca extremităţi pe x şi pe y.

Definiţie - arbore: Un graf $G = (N, A)$ se numeşte *arbore*, dacă este conex şi nu conţine cicluri. Un graf $G = (N, A)$ se numeşte *pădure* dacă nu conţine cicluri. Un exemplu de arbore este prezentat în fig. 3.

Definiţie - arborescenţă / arbore rădăcină: Un graf orientat $G = (N, A)$ se numeşte *arborescenţă* sau *arbore rădăcină* dacă este conex, fără cicluri, iar unul dintre vârfuri r a fost selectat drept *rădăcină*. În plus, oricare ar fi un alt nod $x \in N$, există un unic drum de la r la x.

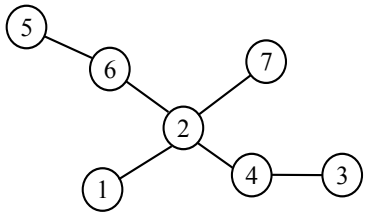


Figure 3: Arbore

Observaţie: De obicei, într-un arbore rădăcină sensul arcelor este dinspre rădăcină către celelalte noduri (*out-tree*). Aşa vor fi consideraţi în continuare arborii în acest curs.

Există şi arbori rădăcină în care orientarea arcelor este inversă, adică de la noduri către rădăcină (*in-tree*). În acest al doilea caz există un drum unic de la orice nod x către rădăcina r.

Proprietăţi:

- Un arbore cu n vârfuri are exact $n-1$ arce.
- Oricare două vârfuri dintr-un arbore sunt conectate prin exact un lanţ.
- Eliminarea orcărui arc are ca rezultat un graf neconex.
- Adăugarea unui nou arc între vârfuri existente produce un ciclu.

Definiţie - ascendent / descendent: Se consideră un arbore de rădăcină r şi x un nod oarecare al arborelui. Atunci:

- Un nod y pentru care (y, x) este o muchie din arbore se numeşte *ascendent direct* sau *părinte* al lui x. Într-un arbore fiecare nod diferit de rădăcină are un singur părinte iar rădăcina nu are nici un părinte.
- Un nod y pentru care (x, y) este o muchie din arbore se numeşte *descendent direct* sau *fiu* al lui x. Un nod oarecare poate avea 0 sau mai mulţi fii.
- Doi descendenţi direcţi ai aceluiaşi nod se numesc *fraţi*.
- Un nod care nu are nici un fiu se numeşte *frunză*.
- Un nod care nu este frunză se numeşte *nod intern* al arborelui.
- Un nod y pentru care există un drum de la x la y se numeşte *descendent* al lui x.

Reprezentarea pe niveluri:

- Pe nivelul 0 se reprezintă nodul rădăcină r.
- Pentru fiecare nod x aflat pe un nivel k, descendenţii săi direcţi se află pe nivelul $k+1$.

În figura 4 este reprezentat un arbore rădăcină pe niveluri.

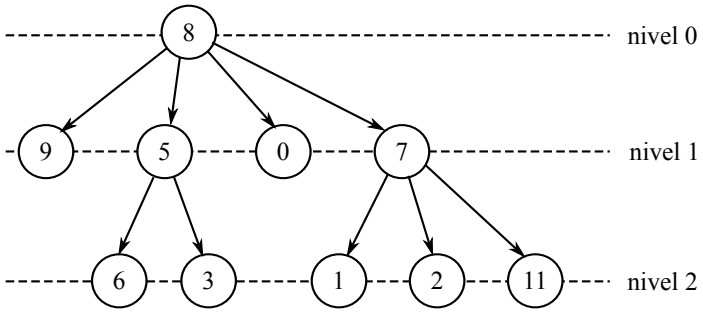


Figure 4: Arbore rădăcină reprezentat pe niveluri.

Definiţie - adâncimea / înălţimea unui arbore: Lungimea drumului de la rădăcina r la un nod x se numeşte adâncimea nodului x. Adâncimea rădăcinii este 0.

Lungimea celui mai lung drum de la x la o frunză se numeşte *înălţime* şi se notează prin $h(x)$.

Definiţie - subarbore: Se numeşte *subarbore de rădăcină* x al unui arbore, arborele format din nodul x împreună cu toţi descendenţii săi.

Arbori n-ari - definiţii

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

În figura 5 este reprezentat un arbore complet.

Definiţie - Arbore binar: Un arbore binar este un arbore în care fiecare nod are cel mult doi descendenţi direcţi. Atunci când fiecare nod are 0 sau 2 fii, arborele se numeşte *arbore binar strict*. În cazul unui arbore binar un nod are un *fiu stâng* şi un *fiu drept*.

2.2 Reprezentarea arborilor

Reprezentare secvenţială - se pretează în cazul arborilor completi.

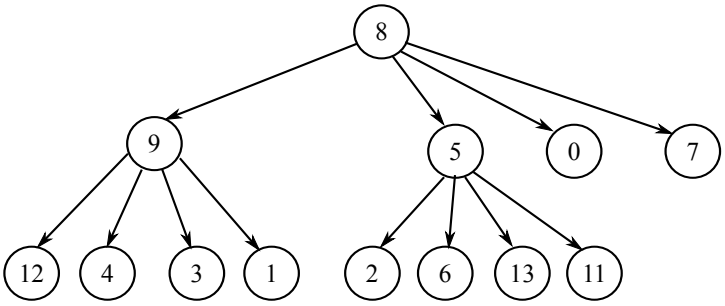


Figure 5: Arbore complet.

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

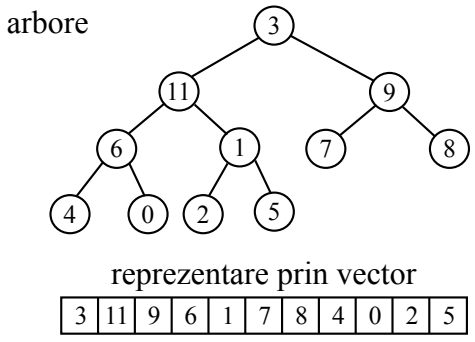


Figure 6: Arbore binar complet memorat într-un vector.

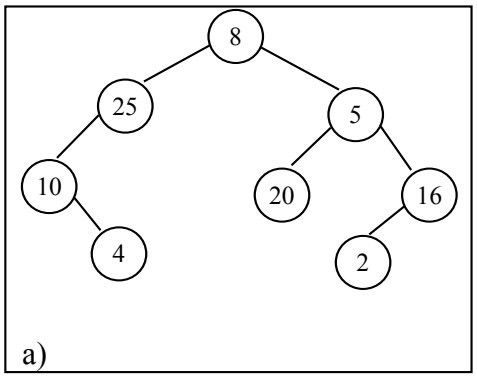
Reprezentare mixtă (pentru arbori binari)

Reprezentarea mixtă utilizează 3 vectori. Primul - INFO - conţine informaţiile din noduri, al doilea - *left* - are pe poziţia i indicele fiului stâng al nodului i - în vectorul INFO -, iar vectorul al treilea - *right* - are pe poziţia i indicele fiului drept al nodului i.

Pentru exemplul din fig. 7 a) reprezentarea mixtă este dată prin vectorii din tabelul din figura 7 b).

Reprezentare cu ajutorul unei structuri de noduri

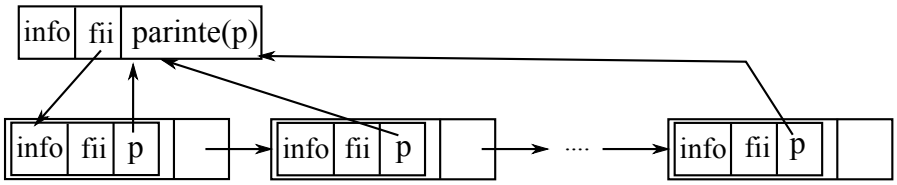
- **Pentru un arbore oarecare** (nu binar): pentru fiecare nod se utilizează o structură în care se reţine informaţia, legătura către o listă / vector de descendenţi direcţi şi (eventual) legătura către nodul părinte.
- Pentru un arbore binar: fiecare nod este reprezentat printr-o structură în care se reţine informaţia, legătura către fiul stâng, legătura către fiul drept şi legătura către părinte.



b)

indice	info	left	right
1	8	2	3
2	25	4	*
3	5	5	6
4	10	*	7
5	20	*	*
6	16	8	*
7	4	*	*
8	2	*	*

Figure 7: a) Arbore binar. b) Tabela de alocare mixtă.



- Pornind de la reprezentarea unui arbore binar, se poate utiliza pentru arbori oarecare o structură asemănătoare, în care se păstrează informaţia, o legătură - fiu - către cel mai din stânga fiu şi o legătură - frate - către primul frate din dreapta al nodului, precum şi o legătură către părinte. Acest lucru este reprezentat grafic în figura 8.

2.3 Parcurgerea arborilor

Există două moduri generale de parcurgere a arborilor oarecare:

- Parcurgerea în lăţime:
  - presupune parcurgerea pe rând a fiecărui nivel de la stânga spre dreapta.
  - se poate realiza practic utilizând o coadă C:
- Parcurgerea în adâncime (preordine):
  - În cazul parcurgerii în adâncime, fiii unui nod sunt vizitaţi tot de la stânga spre dreapta, dar trecerea de la nodul curent la fratele din dreapta se realizează numai dupa vizitarea tuturor descendenţilor nodului curent, deci a întregului subarbore al nodului respectiv.
  - Se poate realiza utilizând o stivă S:

În cazul arborelui din figura 8, parcurgerea în lăţime cu algoritmul de mai sus va avea ca rezultat afişarea cheilor în ordinea: 8, 9, 5, 0, 7, 6, 3, 1, 2, 11. Parcurgerea în preordine cu algoritmul de mai sus va avea ca rezultat afişarea cheilor în ordiea: 8, 9, 5, 6, 3, 0, 7, 1, 2, 11.

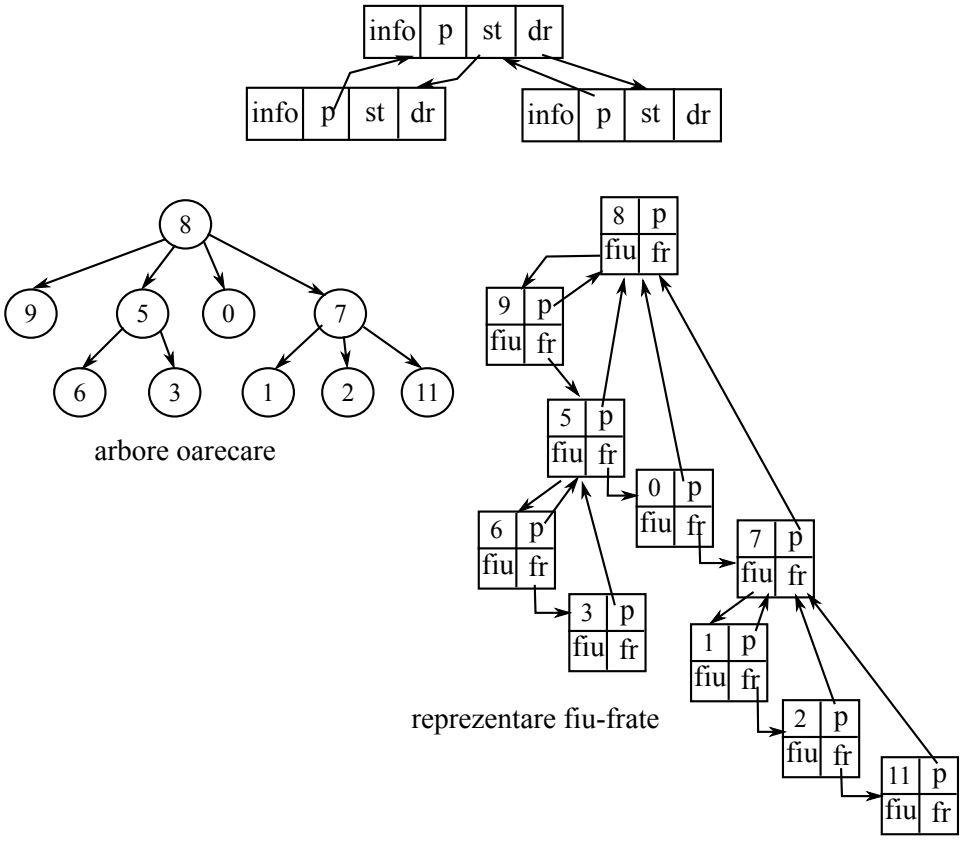


Figure 8: Arbore binar reprezentat printr-o structură de tip fiu-frate.

2.4 Parcurgerea arborilor binari

Parcurgerea unui arbore binar în adâncime presupune parcurgerea fiecărui nod împreună cu subarborele stâng şi subarborele drept. Subarborele stâng se ia în considerare înaintea subarborelui drept.

În funcţie de ordinea în care parcurg rădăcina şi 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 figura 9. Atunci:

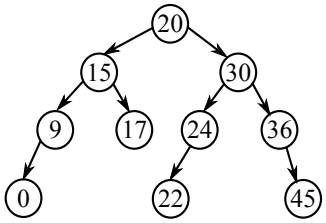


Figure 9: Arbore binar.

- 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.

Exemple de aplicaţii ale arborilor:

- Sortare: Heap-sort
- Codificare: arborele Huffmann
- Compilatoare: arbori sintactici de derivare
- Procesare de imagine / grafică: arbori Quad, arbori PR
- Clasificare: arbori de decizie
- Dicţionare, căutare eficientă: arbori de căutare.

MODULUL 3 - COZI DE PRIORITATE

1 Heap

Definiţie: Un heap binar este un arbore binar complet - fieacare nod intern are exact doi descendenţi, cu excepţia eventual a ultimului nod intern de pe penultimul nivel, iar frunzele se află doar pe ultimele două niveluri, frunzele de pe ultimul nivel sunt ordonate de la stânga spre dreapta - memorat cu ajutorul unui tablou unidimensional (vector) - *array*. În plus există o ordonare a cheilor într-un heap binar, determinată de tipul heap-ului.

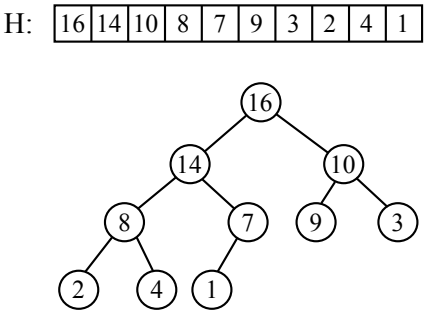


Figure 1: Exemplu de heap-max

1. **Max-heap**: informaţia din fiecare nod este mai mare decât informaţia din oricare descendent al său. Maximul din heap se află în rădăcină (fig. 1).
2. **Min-heap**: informaţia din fiecare nod este mai mică decât informaţia din oricare descendent al său. Minimul din heap se află în rădăcină.

Pentru implementare se poate utiliza structura heap H în care *size* - numărul de chei din Heap. Rădăcina heap-ului se află pe prima poziţie a heapului $H[0]$. Pentru fiecare nod aflat pe poziţia i:

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

Observaţii:

- Înălţimea arborelui care reprezintă heap-ul este $\log_2 n$ unde $n = H.size$ este numărul de noduri din heap.
- Complexitatea operaţiilor de bază este proporţională cu înălţimea arborelui, adică $O(\log_2 n)$.

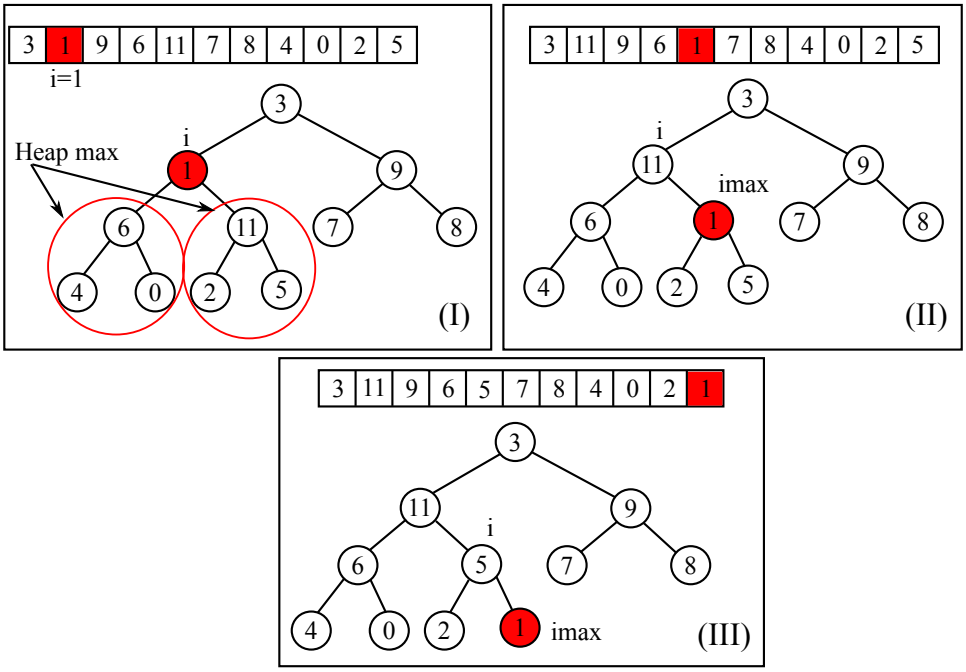


Figure 2: Funcţia MAX_HEAP porneşte în acest exemplu de la nodul i marcat cu roşu în fig. (I). Cei doi subarbori ai săi sunt heap-max. Funcţia are două apeluri recursive, ilustrate în etapele (II) şi (III).

Construcţia unui heap - max

Considerând un vector de elemente, pentru transformarea acestuia într-un heap-max sunt necesare două etape, reprezentate prin funcţiile:

- **MAX_HEAP**(H, i): în care H heap cu $H.size$ elemente şi i indicele din vector a nodului de la se începe. Premiza este aceea că subarborii nodului i sunt heap-max şi doar informaţia din nodul i strică eventual această proprietate. Funcţia **MAX_HEAP** reface proprietatea de heap-max. Această funcţie se întâlneşte în literatură şi sub denumirea *Sift-Down*.
- **CONSTR_HEAP**(H): pe baza funcţiei **MAX_HEAP** se construieşte heap-ul.

```
MAX_HEAP(H,i)
```
$st = 2 * i + 1$
$dr = 2 * i + 2$

```
        imax = i
        daca st < H.size si H[st] > H[i] atunci
              imax = st
        sfarsit daca
        daca dr < H.size si H[dr] > H[imax] atunci
              imax = dr
        sfarsit daca
        daca imax ≠ i atunci
              H[imax] ↔ H[i]
              MAX_HEAP(H,imax)
        sfarsit daca
RETURN
```

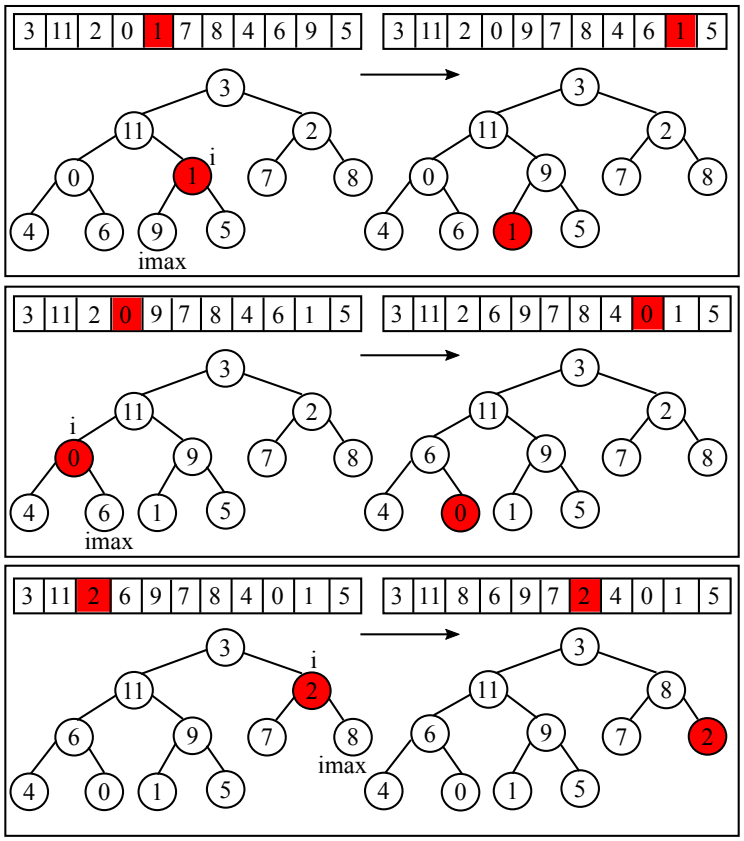


Figure 3: Funcţia CONSTR_HEAP porneşte în acest exemplu de la nodul de pe poziţia a 5-a în heap-ul H, primul nod din dreapta vectorului ce are descendenţi. Apoi se continuă cu nodurile de poziţiile 4 şi 3.

În figurile 3 şi 4 este ilustrată funcţionarea acestui algoritm.

Complexitate: Algoritmul porneşte de la nodul i şi ajunge în cel mai defavorabil caz la o frunză, deci complexitatea depinde de înălţimea arborelui care este $h = log_2 n$. Deci $T(n) = O(log_2 n)$.

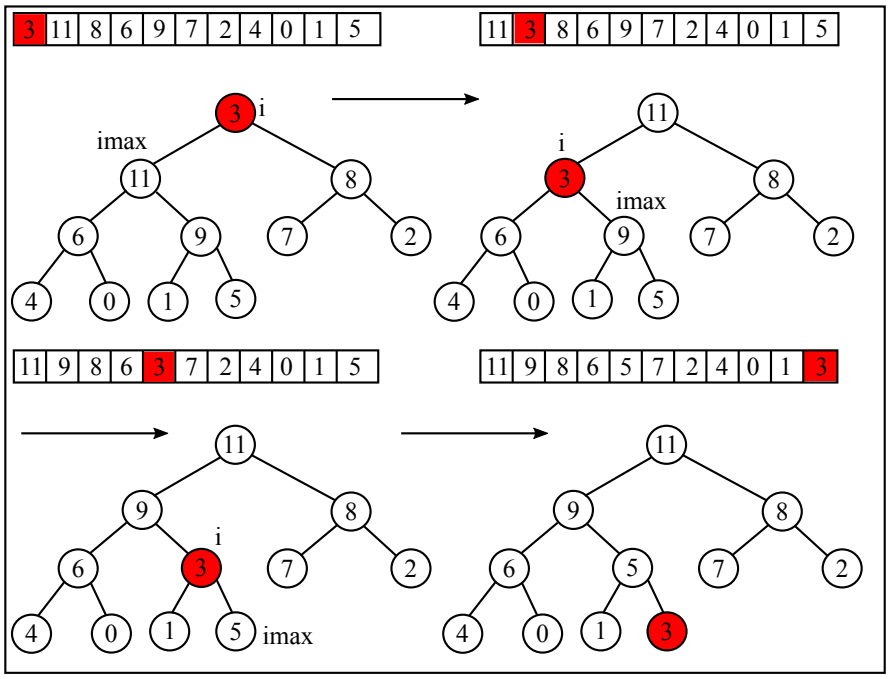


Figure 4: Funcţia MAX_HEAP continuă de la poziţia 2, la care însă nu sunt necesare modificări. Nodul cu cheia 11 satisface proprietatea de max-heap relativ la descendenţii săi, astfel ca se trece la nodul de pe poziţia 1. În final se obţine un heap-max.

Se observă faptul că frunzele sunt heap-max. Atunci e suficient să pornim de la primul nod intern - pornind de la dreapta heap-ului H, adică primul care are cel puţin un descendent. Nodul respectiv se află pe poziţia $H.size/2-1$, iar copiii săi sunt frunze, deci heap-max. Se aplică funcţia de MAX_HEAP pentru nodul $H.size/2-1$, după care se trece la nodul precedent din vector şi aşa mai departe, până la rădăcină.

```
CONSTR_HEAP(H)
        pentru i = H.size/2 − 1, 0, −1
                MAX_HEAP(H,i)
        sfarsit pentru
RETURN
```

În figura 4 este ilustrat procedeul de construcţie a unui heap max.

Complexitate: fiecare apel al funcţiei MAX_HEAP are complexitatea $O(\log_2 n)$.

CONSTR_HEAP efectuează $O(n)$ atfel de apeluri. Deci $T(n)$, unde prin T am notat complexitatea, pentru funcţia CONSTR_HEAP este mărginită superior de $n \log_2 n$. Se demonstrează în literatură faptul că funcţia are complexitate liniară, adică $T(n) = O(n)$, $n = H.size$.

2 Algoritmul de sortare *HeapSort*

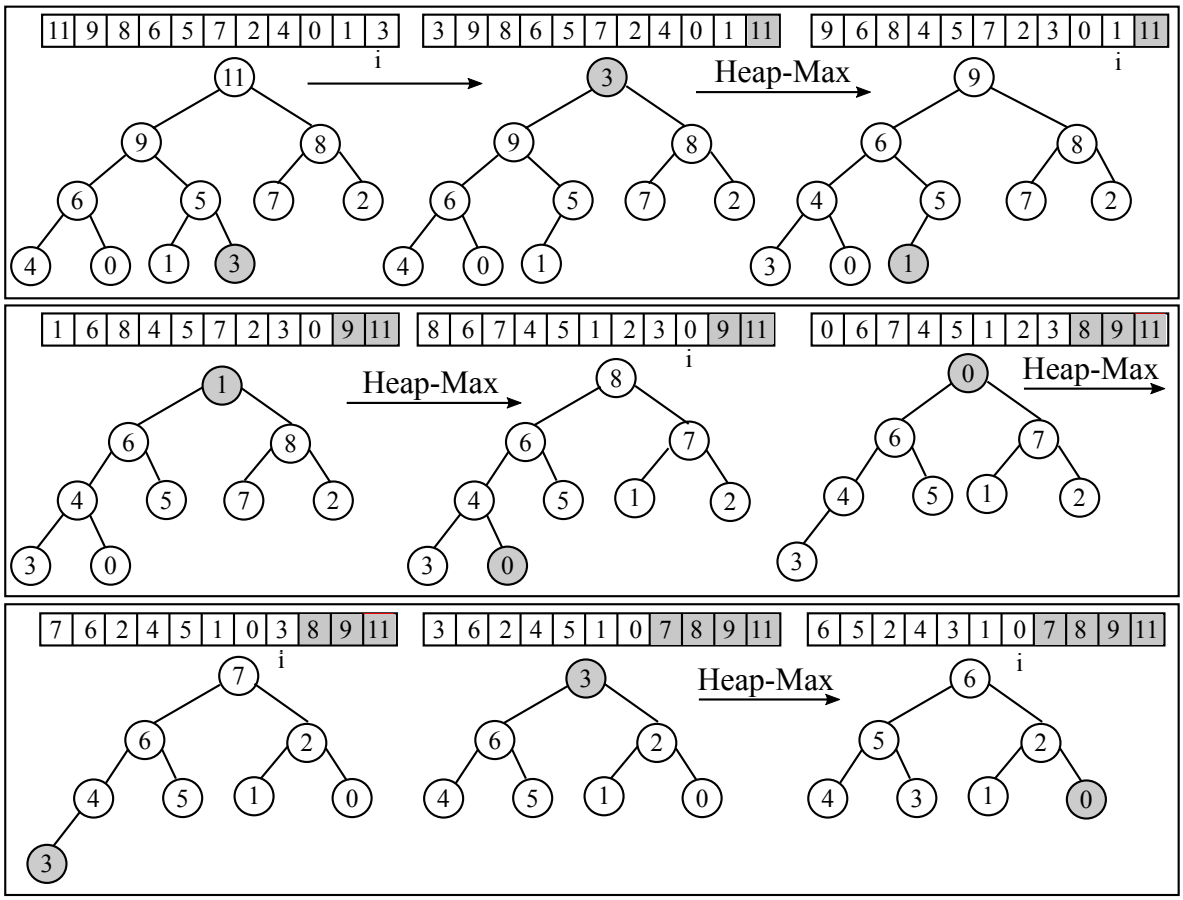


Figure 5: Algoritmul Heapsort.

Se observă faptul că într-un heap-max elementul maxim este plasat în rădăcina arborelui corespunzător, deci pe prima poziţie din vectorul date. În vectorul sortat crescător, elementul maxim trebuie să se afle pe ultima poziţie. Astfel vectorul poate fi sortat prin următorul algoritm: se interschimbă primul element din vectorul heap-ului cu ultimul. Se reduce dimensiunea heap-ului, apoi se reface proprietatea de heap-max aplicând funcţia MAX_HEAP începând din vârful heap-ului. Procedura se reia pentru acest heap redus.

```
HEAPSORT(H)
        CONSTR_HEAP(H)
        n = H.size − 1
        pentru i = n, 1, −1
                H[0] ↔ H[i]
                H.size=H.size-1
                MAX_HEAP(H,0)
        sfarsit pentru
RETURN
```

În figurile 5 şi 6 este ilustrată funcţionarea algoritmului Heapsort.

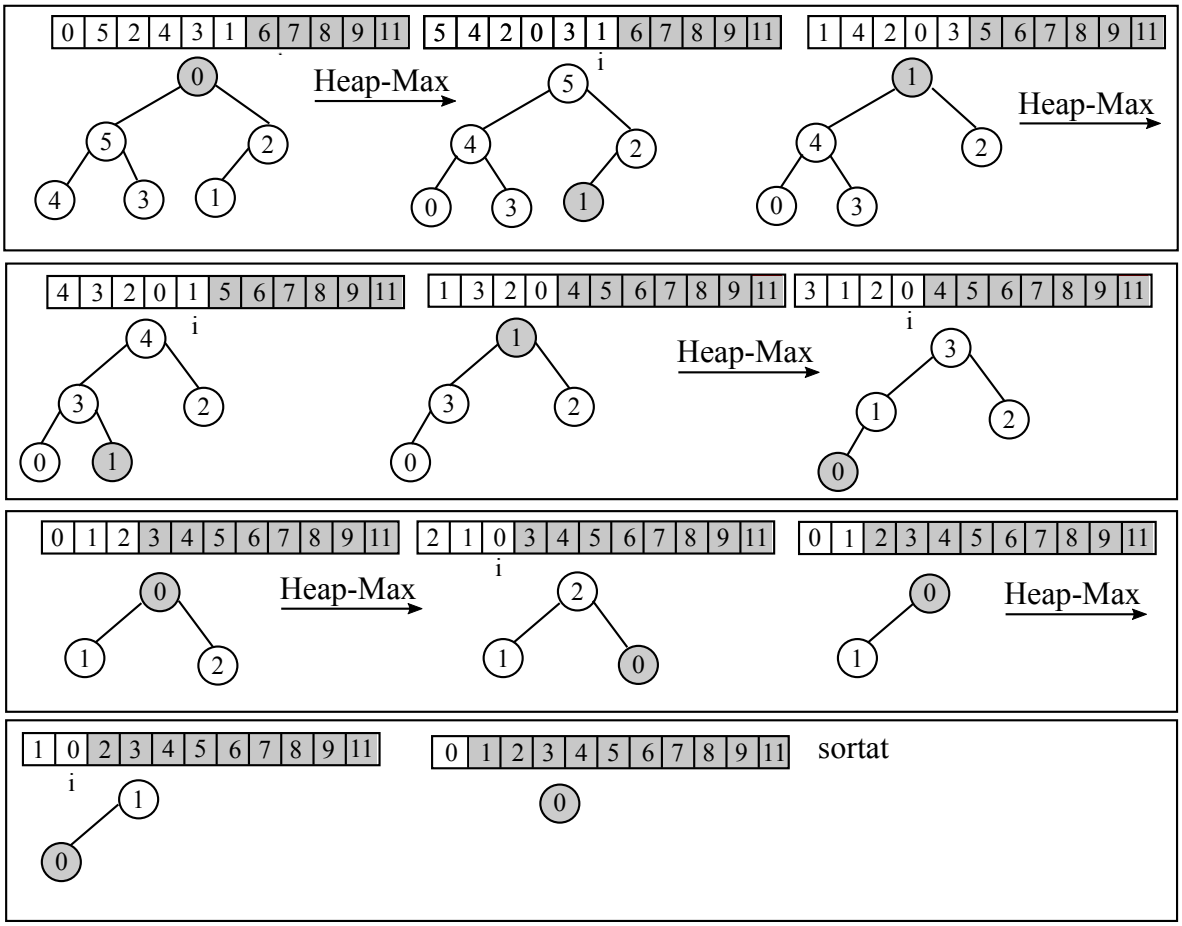


Figure 6: Algoritmul Heapsort.

Complexitate:

- apelul funcţiei CONSTR_HEAP este $O(n)$
- apelul pentru MAX_HEAP este $O(\log_2 n)$
- funcţia MAX_HEAP se apelează de $n-1$ ori

Rezultă $T(n) = O(n \log_2 n)$.

3 Cozi de prioritate

Definiţie: O coadă de prioritate este o structură de date utilizată pentru păstrarea unei mulţimi dinamice de date S în care fiecărui element i se asociază o valoare numită *prioritate*. Pentru gestionarea eficientă a cozilor de prioritate se utilizează heap-uri. Există cozi de max-prioritate - heap-max - şi cozi de min-prioritate - heap-min.

În continuare vom considera cozi cu max-prioritate.

Operaţii în cozi de prioritate (max-heap):

- Inserţia unui element nou
- Determinarea elementului de prioritatea maximă
- Extragerea elementului de prioritate maximă
- Creşterea priorităţii unui element

Utilizare: Cozile de prioritate pot fi utilizate pentru gestionarea proceselor pe un calculator. La fiecare moment se execută procesul de prioritate maximă. Procese noi se pot insera în coadă.
Alt exemplu de aplicaţie este în implementarea unor algoritmi de căutare informată (Dijkstra, A^*).

1. Determinarea elementului cu prioritate maximă: acesta se află în nodul rădăcină al heap-max:

   ```
   CP_MAX(H)
   RETURN H[0]
   ```

   Complexitate: $O(1)$

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

   ```
   CP_EXTRACT_MAX(H)
     daca H.size<1 atunci
          RETURN
     alfel
          max=H[0]
          H[0] = H[H.size-1]
          H.size=H.size-1
          MAX_HEAP(H,0)
     sfarsit daca
   RETURN max
   ```

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

   În figura 7 este prezentat un exemplu pentru extragerea maximului.

3. Creşterea priorităţii elementului de pe poziţia i. Prin această modificare este posibil să se strice proprieatea de heap-max, deoarece s-ar putea ca noua valoare a elementului de pe poziţia i să fie mai mare decât a părintelui său. Pentru aceasta se merge

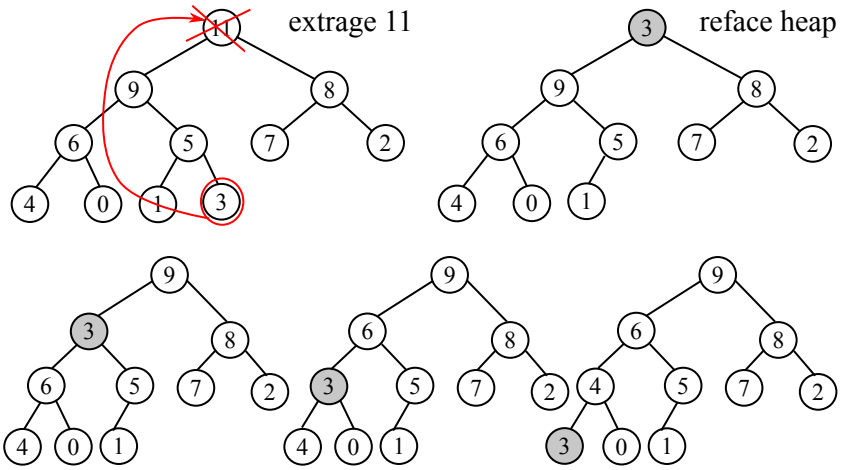


Figure 7: Extragerea maximului.

din părinte în părinte către rădăcină, până se găseşte o poziţie potrivită pentru noua valoare. Algoritmul următor este cunoscut în literatură şi sub denumirea de *Sift-Up*.

```
CP_CRESTE_PRIORITATE(H, i, val)
  daca val < H[i] atunci
       scrie("valoarea este prea mica")
  altfel
       H[i] = val
       p = (i − 1)/2
       cat timp i > 0 si H[p] < val
                H[i] = H[p]
                i=p
                p=(i-1)/2
       sfarsit cat timp
       H[i]=val
  sfarsit daca
RETURN
```

Complexitate: $O(\log_2 n)$ - se porneşte de la o frunză către rădăcină şi deci complexitatea depinde de înălţimea arborelui.

În figura 8 este prezentat un exemplu pentru creşterea priorităţii.

4. Inserţia unui element nou într-un max-heap: se măreşte dimensiunea heap-ului, se plasează noul element pe ultima poziţie cu prioritatea considerată 0 şi apoi se aplică funcţia CP_CRESTE_PRIORITATE pentru acest nou element cu valoarea priorităţii asociate.

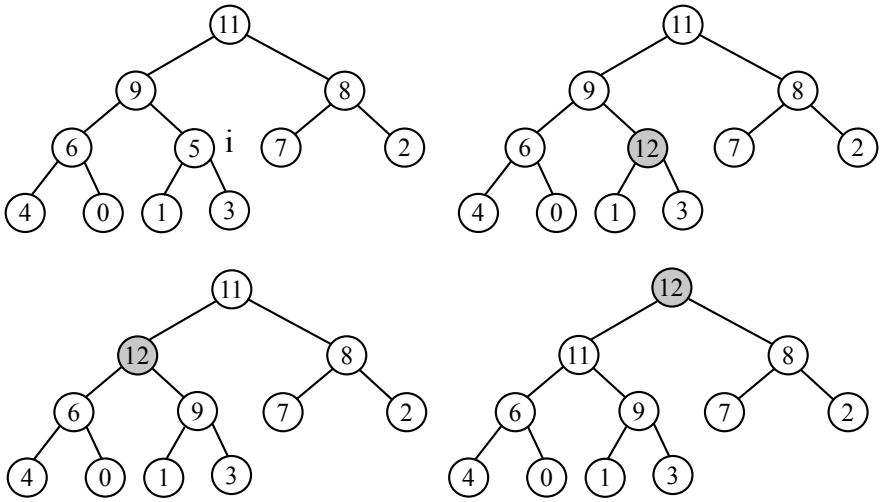


Figure 8: Creşterea priorităţii nodului cu cheia 5 la valoarea 12.

```
CP_INSERT(H,val)
   H[H.size] = 0
   H.size = H.size + 1
   CP_CRESTE_PRIORIT(H,H.size-1,val)
RETURN
```

Complexitate: $O(\log_2 n)$ - se porneşte de la o frunză către rădăcină şi deci complexitatea depinde de înălţimea arborelui.

Observaţii:

- Pe un max-heap se pot implementa cu complexiatea $O(\log_2 n)$ operaţii cu cozi de prioritate.
- Construcţia unui heap-max, care s-a făcu prin apelarea funcţiei MAX_HEAP, se poate realiza şi prin inserţii succesive ale nodurilor în heap.

MODULUL 5 - ARBORI BIANRI DE CĂUTARE

1 Noţiuni de bază

Definiţie: un arbore binar de căutare este un arbore binar cu următoarele proprietăţi.

- 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 relaţia de ordine nu este strictă, dacă nodurile cu chei egale se inserează pe aceeaşi parte a arborelui, inserţia multor noduri cu chei egale are ca urmare obţinerea unui arbore relativ dezechilibrat, având ca urmare o creştere a complexităţii operaţiilor. În continuare se consideră arbori binari de căutare cu chei distincte. Cazul cheilor egale se va discuta separat la sfârşitul cursului.

În figura 1 a) este prezentat un exemplu de arbore binar de căutare.

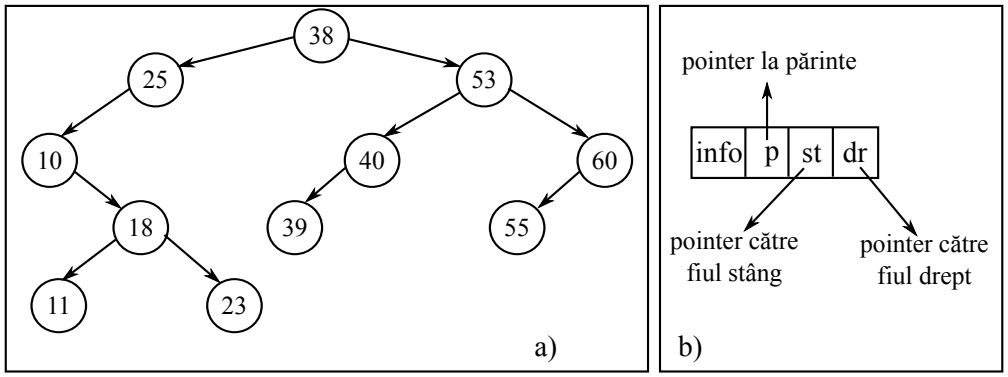


Figure 1: a) Exemplu de arbore binar de căutare; b) Structura unui nod din arbore.

Penru fiecare nod al arborelui se consideră structura din figura 1 b), în care fiecare nod x are câmpurile: $x.info$ = cheia nodului, $x.st$ şi $x.dr$ = fiul stâng şi respectiv fiul drept, $x.p$= părintele nodului x.

2 Operaţii într-un arbore binar de căutare

Operaţiile de bază într-un arbore binar de căutare sunt:

- Căutarea binară a unei chei.
- Determinarea nodului cu cheia minimă/maximă din arbore.
- Căutarea binară a succesorului / predecesorului unui nod
- Inserţie/ştergere a unui nod cu o anumită cheie
- Sortarea cheilor arborelui, prin parcurgerea acestuia în inordine .

Observaţii:

1. Complexitatea operaţiilor într-un arbore binar de căutare este proporţională cu înălţimea arborelui. De fapt, dacă arborele conţine n noduri atunci $O(log_2 n) \leq T(n) \leq O(n)$, $T(n) =$ complexitatea algoritmului utilizat.
2. 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 autobalansează, de exemplu arborii AVL şi arborii roşu-negru. Pentru aceştia se demonstrează faptul că au complexitatea operaţiilor de ordinul $O(\log_2 n)$.

2.1 Căutarea binară

Problemă: Considerând un arbore binar de căutare T cu rădăcina $T.rad$, să se determine nodul cu o cheie dată k.

Soluţie: La fiecare moment dat compar cheia nodului curent x, $x.info$, cu k. Dacă $x.info = k$ atunci se returnează nodul x. Dacă $x.info < k$ atunci se continuă căutarea în subarborele drept al lui x, altfel se continuă căutarea în subarborele stâng al lui x.

Algoritm:

```
AB_CAUT(T,k)
  x=T.rad
  cat timp x ≠ NULL si x.info ≠ k
           daca k < x.info atunci
                 x = x.st
           altfel x = x.dr
           sfarsit daca
  sfarsit cat timp
RETURN x
```

2.2 Minimul dintr-un arbore de căutare

Nodul cu informaţia minimă din subarborele de rădăcină x a unui arbore binar de căutare poate fi găsit pornind de la nodul x şi coborând în descendenţii stângi până la cea mai din stânga frunză. Funcţia AB_MIN returnează nodul cu informaţia minimă.

Algoritm:

```
AB_MIN(x)
  y = x
  cat timp y.st ≠ NULL
           y = y.st
  sfarsit cat timp
RETURN y
```

Observaţie: maximul se determină în mod similar şi anume parcurgând descendenţii drepţi până la cea mai din dreapta frunză.

2.3 Succesorul binar

Succesorul 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 şirul sortat al valorilor din arbore.

- Poate fi determinat prin comparaţii
- Dacă există, este:
  - Cel mai mic element din $x.dr$, dacă $x.dr \neq NULL$
  - Un nod părinte y pentru care x se află în subarborele stâng al lui y, dacă x nu are descendent drept.
- Dacă x este nodul cu cea mai mare cheie, atunci x nu are succesor.

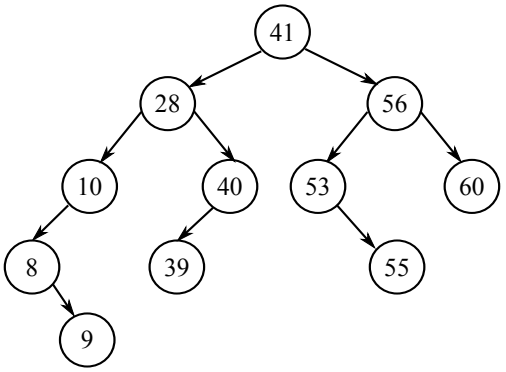


Figure 2: Arbore binar de căutare.

Exemplu: În arborele din figura 2 obţinem:

AB_SUCCESOR(41) = 53
AB_SUCCESOR(39) = 40
AB_SUCCESOR(9) = 10
AB_SUCCESOR(60) - nu există

Algoritm

```
AB_SUCCESOR(x)
   daca x.dr ≠ NULL atunci
        y = AB_MIN(x.dr)
        RETURN y
   sfarsit daca
   y = x.p
   cat timp y ≠ NULL si x = y.dr
            x = y
            y = y.p
   sfarsit cat timp
 RETURN y
```

2.4 Inserarea unui nod

Se consieră arborele binar de căutare T cu rădăcina $T.rad$ şi nodul z, care are câmpurile

$$z.info = k, z.st = NULL, z.dr = NULL.$$

Se doreşte inserarea acestui nod în arborele binar T.
Ideea principală este următoare: pornind de la rădăcină se coboară în arbore, până la un nod, care are cel mult un fiu şi care poate fi părintele nodului z. Pentru a respecta proprietatea de arbore binar de căutare, dacă informaţia nodului curent x este mai mare decât $z.info$, atunci z se va insera în subarborele stâng al lui x, altfel se va insera în subarborele drept. În algoritmul următor x reprezintă nodul curent, care iniţial este $T.rad$ = rădăcina lui T, y reprezintă părintele lui x, iniţial $NULL$.

Algoritm

```
AB_INSERT(T,z)
   y = NULL //pastreaza parintele nodului curent x
   x = T.rad
   cat timp x ≠ NULL
            y = x
            daca z.info < x.info atunci
                 x = x.st
            altfel x = x.dr
            sfarsit daca
```

```
  sfarsit cat timp
  z.p = y
  daca y = NULL atunci T.rad = z //inserarea radacinii
  alfel //verific pe care parte se face insertia
          daca z.info < y.info atunci
                y.st = z
          altfel y.dr = z
          sfarsit daca
  sfarsit daca
RETURN
```

În figura 3 este ilustrat algoritmul de inserţie a unui nod cu cheia 38 într-un arbore binar de căutare.

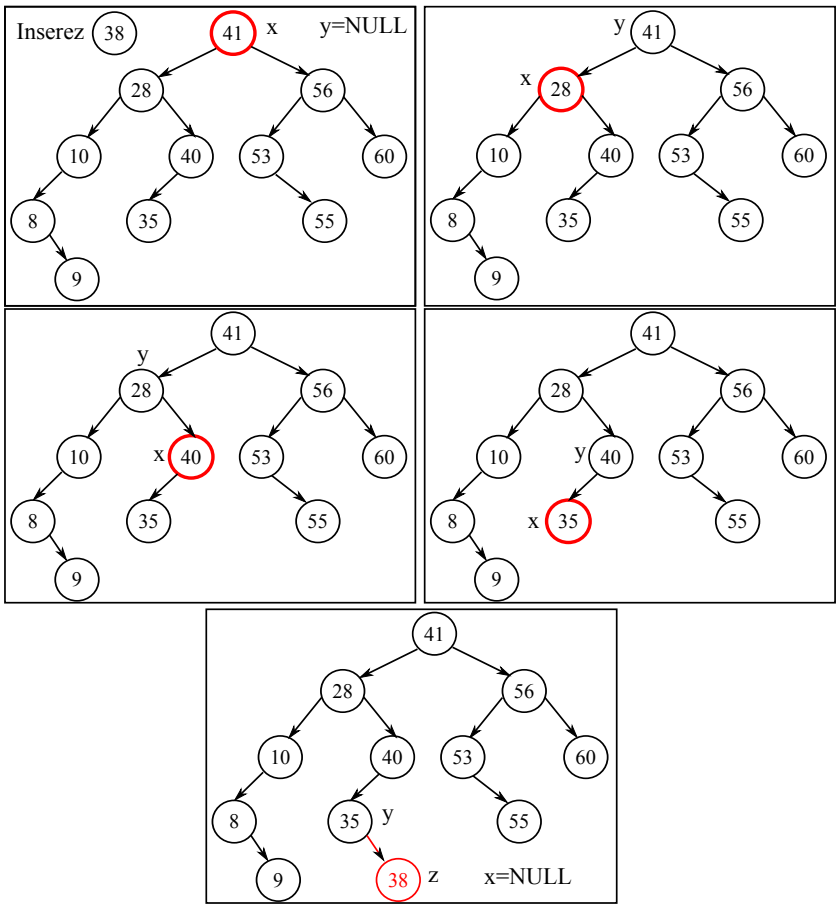


Figure 3: Inserarea nodului cu cheia 28 într-un arbore binar. Nodul curent cu care se compară nodul ce se inserează este marcat cu roşu.

2.5 Ştergerea unui nod

Ştergerea 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 şi atunci este pur şi simplu înlocuit cu NULL
2. z are un singur fiu nenul. Atunci se înlocuieşte z cu acel fiu
3. z are doi fii nenuli. Atunci se determină succesorul y al lui z care se află în subarborele drept al lui z şi evident nu are descendent stâng. Apoi se înlocuieşte nodul z cu nodul y, iar y se înlocuieşte cu fiul său drept.

Observaţie: În cazul în care z are doi descendenţi nenuli, el poate fi înlocuit şi cu predecesorul său.

În funcţia AB_STERGE se consideră T arborele binar, z nodul care trebuie şters şi y nodul cu care se înlocuieşte z. Cele 3 cazuri descrise mai sus vor fi cuprinse în funcţie în următoarele cazuri:

1. z nu are fiu stâng $\Rightarrow$ se înlocuieşte z cu fiul drept - eventual NULL. Acest caz include şi cazul în care z nu are nici un fiu.
2. z nu are fiu drept $\Rightarrow$ se înlocuieşte z cu fiul stâng
3. z are ambii fii nenuli $\Rightarrow$ y=sucesorul lui z care se află în subarborele drept al lui z şi are fiul stâng NULL
   a. y este descendentul drept direct al lui $z \Rightarrow$ se înlocuieşte 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 descendentul drept direct al lui $z \Rightarrow$ se înlocuieşte y cu fiul său drept iar apoi se înlocuieşte nodul z cu nodul y.

În bibliografie (T.H. Cormen - *Introduction to Algorithms*) - este propusă utilizarea unei funcţii ajutătoare TRANSPLANT(T,u,v) care înlocuieşte în arborele T nodul u ca subarbore cu nodul v - de fapt această funcţie realizează doar managementul legaturilor între părintele lui u şi nodul v, legăturile cu fiii se realizează separat în funcţia de ştergere propriu-zisă.

Algoritm

```
AB_TRANSPLANT(T,u,v)
    daca u.p = NULL atunci
         T.rad = v
    altfel
         daca u = u.p.st atunci
              u.p.st = v
         altfel
```

```
            u.p.dr = v
        sfarsit daca
    sfarsit daca
    daca v ≠ NULL atunci
        v.p = u.p
RETURN
```

Modul de funcţionare al funcţiei AB_TRANSPLANT este ilustrat în figura 4.

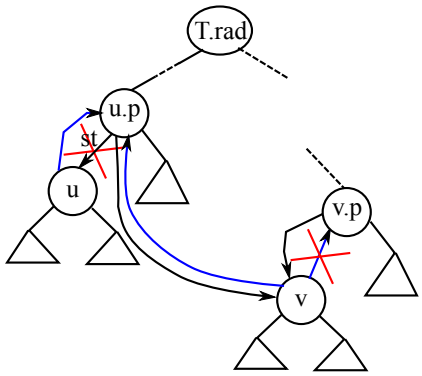


Figure 4: Managementul legăturilor între părintele nodului u şi nodul v.

În continuare este prezentat algoritmul de ştergere a unui nod z dintr-un arbore binar de căutare T.

```
AB_DELETE(T, z)
    daca z.st = NULL atunci
        AB_TRANSPLANT(T, z, z.dr) //înlocuieşte z cu z.dr
    altfel
        daca z.dr = NULL atunci
            AB_TRANSPLANT(T, z, z.st) //înlocuieşte z cu z.st
        altfel
            y = AB_SUCCESOR(z)
            daca y ≠ z.dr atunci
                AB_TRANSPLANT(T, y, y.dr)
                y.dr = z.dr //descendentul drept al lui z
                z.dr.p = y //devine descendent drept al lui y
            sfarsit daca
            AB_TRANSPLANT(T, z, y)
            y.st = z.st //descendentul stang al lui z
            z.st.p = y //devine descendent stang al lui y
        sfarsit daca
    sfarsit daca
RETURN
```

Cazurile luate în considerare de către funcţia AB_DELETE sunt ilustrate în figura 5.

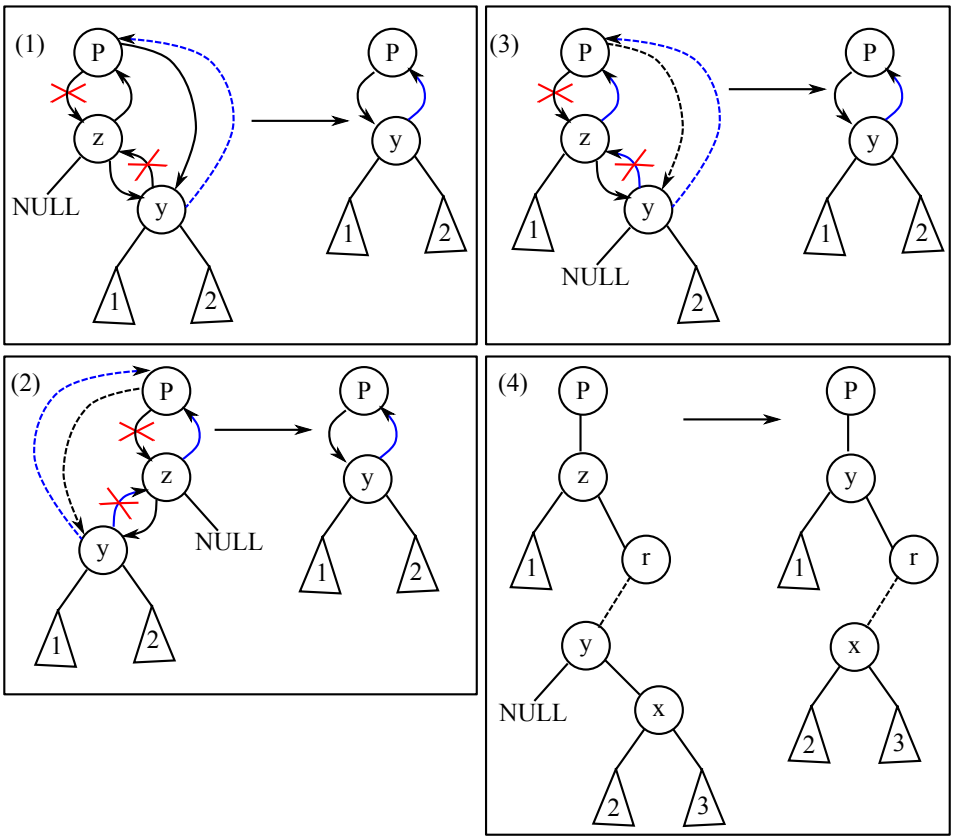


Figure 5: Ştergerea unui nod z care: (1) are fiul stâng nul; (2) are fiul drept nul; (3) are ambii fii nenuli, dar succesorul y este descendent direct al lui z; (4) are ambii fii nenuli, dar succesorul y nu este descendent direct al lui z.

În figura 6 este prezentat un exemplu de ştergere dintr-un arbore binar de căutare..

Complexitate: complexitatea tuturor operaţiilor descrise mai sus, începând de la căutarea binară, au complexitatea $O(h)$, unde $h =$ înălţimea arborelui.

3 Exerciţii rezolvate

1. Care este numărul minim de chei într-un arbore binar (de căutare sau nu) cu înălţimea h?
   Soluţie: Numărul minim de chei se obţine atunci când pe fiecare nivel avem număr minim de noduri, adica unul singur. În acest caz numărul de noduri este $h + 1$.

2. Care este numărul maximde chei într-un arbore binar (de căutare sau nu) cu înălţimea h?
   Soluţie: Numărul maxim de chei se obţine atunci când pe fiecare nivel k avem număr maxm de noduri, adica unul 2^k noduri. În acest caz numărul de noduri este

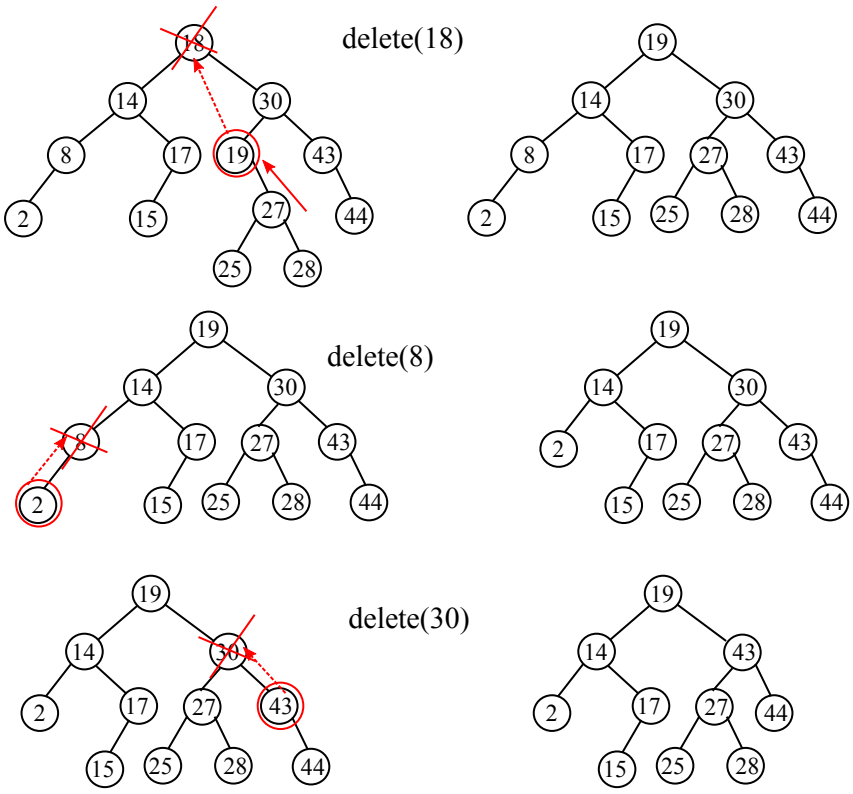


Figure 6: Se şterg pe rând cheile 18, 8, 30

$2^{h+1} - 1$. Pentru detalii se poate vedea exerţitiul 1 de la problemele rezolvate la capitolul de heap-uri, pentru că în acest caz este acelaşi calcul.

3. Care este înălţimea minimă / maximă a unui arbore cu n noduri?
   Soluţie: înălțimea maximă se obţine pentru număr minim de noduri per nivel şi este $n - 1$ şi înălţimea minimă se obţine pentru număr maxim de noduri per nivel şi este $\log_2 n$.

4. Cunoscându-se parcurgerea în preordine a unui arbore binar de căutare, să se refacă arborele. Exemplu: RSD: 23, 17, 10, 15, 19, 35, 26, 24, 30, 37.
   Soluţie:

   var. 1 Observăm faptul că, prin parcurgerea în inordine a unui arbore binar de căutare se obţine şirul cheilor sortat crescător. Deci, cunoscând care sunt cheile arborelui, din parcurgerea în preordine (RSD), putem imediat obţine parcurgerea în inordine şi apoi putem aplica algoritmul discutat la tema3. (vezi documentaţie de pe elearning).

   var. 2 Dacă luăm pur şi simplu cheile în ordinea în care apar în parcurgerea RSD şi le inserăm într-un arbore binar de căutare iniţial vid, obţinem arborele cerut.

   Pentru cazul din exemplu, arborele este prezentat în fig. 7:

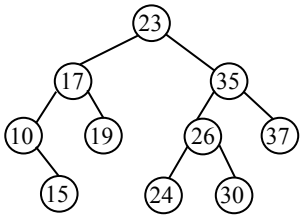


Figure 7: Arborele binar de căutare a cărui parcurgere în preordine este: 23, 17, 10, 15, 19, 35, 26, 24, 30, 37.

MODULUL 5 - II - ARBORI AVL

1 Noţiuni generale

Arborii AVL sunt arbori binari de căutare aproape balansaţi. Denumirea provine de la autorii acestor arbori, doi matematicieni ruşi G.M. Adelson-Velsky şi E.M. Landis. Considerăm pentru fiecare nod x un *factor de balansare* dat prin

$$fb(x) = h(x.dr) - h(x.st)$$

adică, factorul de balansare al unui nod x este reprezentat de către diferenţa dintre înălţimea subarborelui său drept şi înălţimea subarborelui său stâng. În unele documentaţii diferenţa se realizează între înălţimea subarborelui stâng şi cea a subarborelui drept.

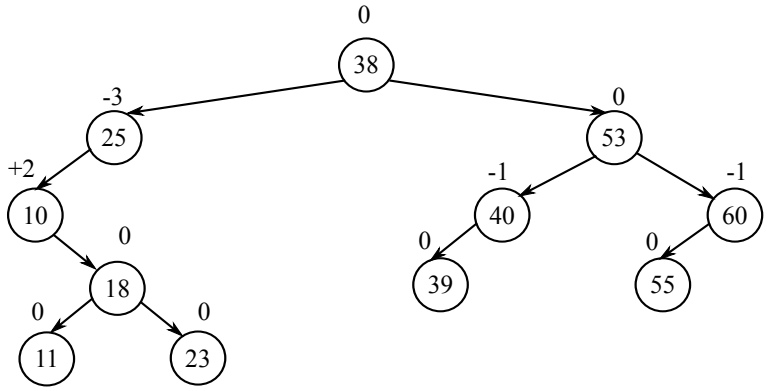


Figure 1: Arbore binar de căutare împreună cu factorii de balansare asociaţi nodurilor. Atenţie: acesta NU este un arbore AVL.

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

În figura 1 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 şi 25 nu sunt balansate.

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

Un exemplu de arbore AVL este prezentat în figura 2.

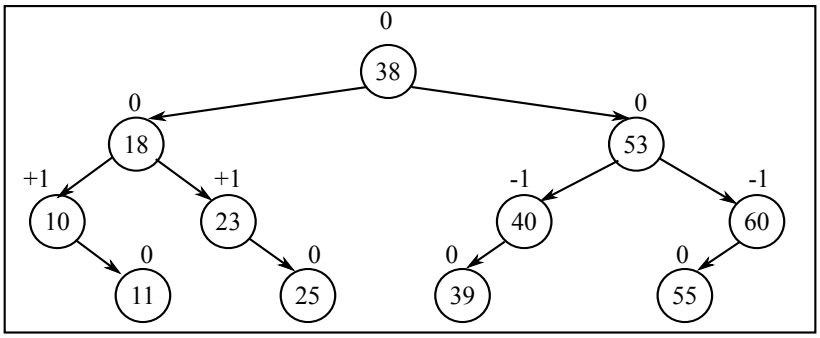


Figure 2: Arbore AVL.

Înălţimea unui arbore AVL

Pentru a determina înălţimea maximă a unui arbore AVL care conţine n noduri se notează cu h înălţimea acestuia şi cu $N(h)$ numărul minim de noduri ale unui arbore AVL de înălţime h. Evident $n \geq N(h)$.

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

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

Cazurile de bază:

$$N(0) = 1$$
$$N(1) = 2$$

Evident: $N(h-1) > N(h-2)$, de unde rezultă $N(h) > 2N(h-2) > 4N(h-4) > \ldots > 2^i N(h-2i)$.
Ajungem la un caz elementar pentru $h-2i = 0$ pentru h par - sau $h-2i = 1$ pentru h impar. De aici rezultă:
$N(h) > 2^{h/2}$ pentru h par sau $N(h) > 2 * 2^{(h-1)/2} = 2^{(h+1)/2}$ pentru h impar.
Cum $N(h)$ = nr. minim de noduri pentru un AVL de înălţime $h \Rightarrow$ numărul n de noduri dintr-un AVL de înălţime h respectă: $n \geq N(h) > 2^{h/2}$
Deci $h < 2log_2(n)$.

Deoarce complexitatea operaţiilor de căutare, inserţie şi ştergere dintr-un arbore binar de căutare este $O(h)$ rezultă că într-un arbore AVL complexitatea acestor operaţii este $O(log_2 n)$, dacă nu se ţine cont de operaţiile de reechilibrare ale arborelui.

Prin operaţii de inserţie/ştergere se poate produce o debalansare a anumitor noduri. Pentru refacerea proprietăţii de arbore AVL utilizează prin operaţii de **rotaţie**.

2 Operaţii într-un arbore AVL

2.1 Rotaţia

Este o operaţie locală care schimbă structura de pointeri într-un arbore binar, dar păstrează proprietăţile acestuia.

Tipuri de rotaţie:

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

Operaţia de rotaţie este ilustrată în figura 3.

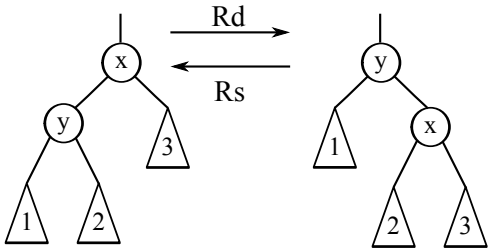


Figure 3: Rotaţii într-un arbore binar de căutare.

Observaţii:

- pentru a putea efectua o rotaţie spre drapta în jurul nodului x, este necesar ca nodul x să aibă un descendent stâng diferit de NULL.
- pentru a putea efectua o rotaţie spre stânga în jurul nodului y, este necesar ca nodul y să aibă un descendent drept diferit de NULL.

Algoritm pentru rotaţia spre stânga în jurul nodului y

```
ROT_ST(T,y)
    x = y.dr
    y.dr = x.st //subarb 2 din imag se mută de la x la y
    daca x.st ≠ NULL atunci
        x.st.p = y//fac legatura subarb.  2 la noul părinte y
```

```
    sfarsit daca
    x.p = y.p
    daca y.p = NULL atunci
         T.rad = x
    altfel
         daca y = y.p.st atunci //fac legatura de la parintele
                                //lui y la x
              y.p.st = x
         altfel y.p.dr = x
         sfarsit daca
    sfarsit daca
    x.st = y //fac legatura de la x la y
    y.p = x //fac legatura de y la x
RETURN
```

Rotaţia înspre dreapta este simetrică.

Complexitate: $O(1)$

2.2 Inserţia

Prin inserarea unui nod nou se poate produce o debalansare în arbore. Acest lucru înseamnă că, cel puţin un nod din arbore va avea după recalcularea factorilor de balansare un factor -2 sau 2.

Rebalansare: Nodul nou inserat are factorul de balansare 0. Recalcularea factorilor de

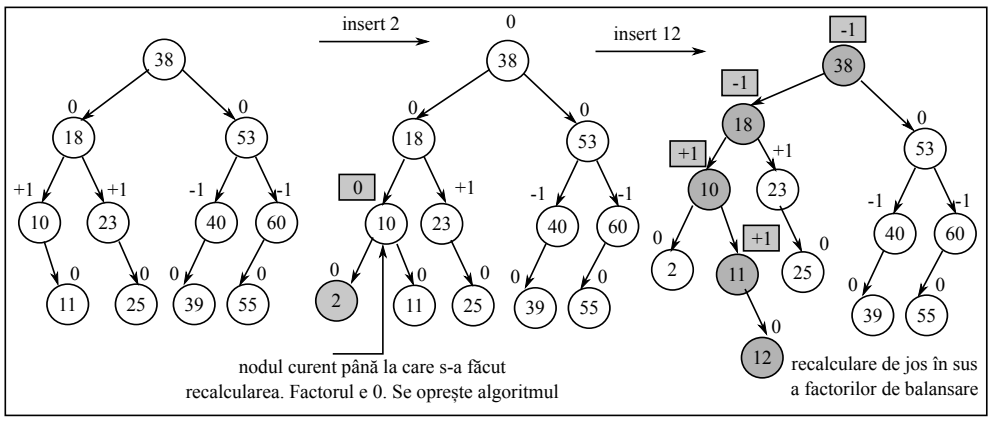


Figure 4: Exemplu de recalculare a factorilor de balansare de la nodul inserat în sus.

balansare începe de la părintele nodului inserat înspre rădăcină.

Notăm cu x nodul curent. Factorul de balansare al lui x se recalculează în modul următor:

- Dacă s-a ajuns la x de la descendentul său stâng, atunci inserţia a fost realizată la stânga lui x, a crescut înălţimea pe stânga lui $x \Rightarrow$ scade factorul de balansare a lui

x cu 1.

- Dacă s-a ajuns la x de la descendentul său drept, atunci inserţia a fost realizată la dreapta lui x, a crescut înălţimea pe dreapta lui $x \Rightarrow$ creşte factorul de balansare a lui x cu 1.

După recalcularea factorului nodului curent x:

- Dacă factorul de balansare nou al lui x este 0, atunci nu a crescut înălţimea subarborelui de rădăcină x, faţă de cât era înainte de inserţie, doar s-a echilibrat. Din acest motiv, nu mai are sens continuarea urcării în arbore. Deci algoritmul se opreşte!
- Dacă factorul de balansare nou al lui x este -1 sau $+1$, atunci s-a produs o creştere a înălţimii subarborelui de rădăcină x pe una dintre ramuri (stângă respectiv dreaptă). Acest lucru produce o modificare a factorului de balansare al părintelui $x.p$, deci se continuă cu procesul de reechilibrare de la $x.p$ ca fiind noul nod curent.
- Dacă factorul de balansare nou al lui x este -2 sau $+2$, atunci s-a produs o debalansare a subarborelui de rădăcină x şi sunt necesare proceduri de rebalansare, care vor fi prezentate mai jos. În urma acestor proceduri, înălţimea subarborelui curent va reveni la cea de dinainte de inserţie (se va vedea în continuare) şi de aceea nu mai este necesară continuarea rebalansării mai sus, deci algoritmul se opreşte.

Un exemplu de astfel de recalculare este prezentat în figura 4.

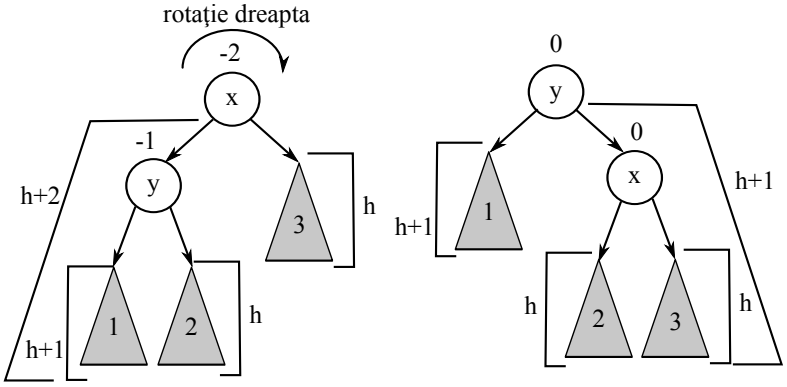


Figure 5: Rebalansarea arbore AVL în cazul 1 a).

Cazurile de rebalansare

Cazul 1

a) $x.fb = -2$. Deoarece debalansarea s-a făcut prin inserţia unui nod într-un arbore AVL valid rezultă că inserţia s-a făcut la stânga lui x. Se verifică factorul de balansare al lui $y = x.st$. Dacă $y.fb = -1$ atunci: se efectuează o rotaţie spre dreapta

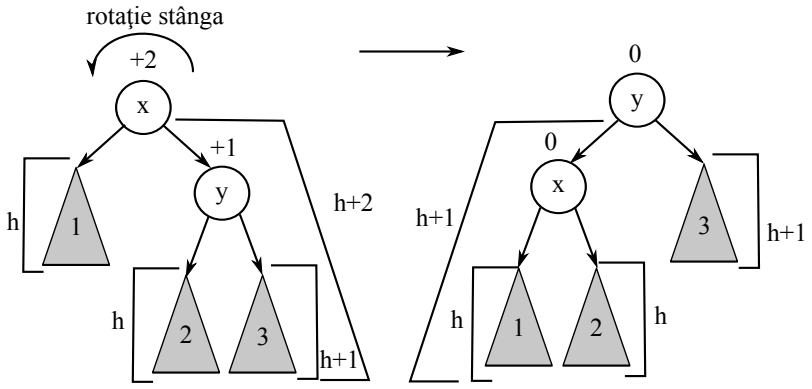


Figure 6: Rebalansarea arbore AVL în cazul 1 b).

în jurul lui x. Rezultatul acestei operaţii este ilustrat în figura 5. Se observă faptul că noii factori de balansare sunt 0 atât pentru x cât şi pentru y. În plus, înainte de inserţie, subarborele stâng al nodului x avea înălţimea $h+1$, iar cel drept h. După rebalansare ambii subarbori au înălţimea $h+1$. Rezultă că înălţimea subarborelui care acum are rădăcina y nu a crescut, faţă de înălţimea înainte de inserţie, 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) $x.fb = 2$. Deoarece debalansarea s-a făcut prin inserţia unui nod într-un arbore AVL valid rezultă că inserţia s-a făcut la dreapta lui x. Se verifică factorul de balansare al lui $y = x.dr$. Dacă $y.fb = 1$ atunci: se efectuează o rotaţie la stânga în jurul lui x. Rezultatul acestei operaţii este ilustrat în figura 6. Observaţii similare cazului 1.a pot fi făcute şi în acest caz.

Cazul 2:

a) $x.fb = -2$. Deoarece debalansarea s-a făcut prin inserţia unui nod într-un arbore AVL valid rezultă că inserţia s-a făcut la stânga lui x. Se verifică factorul de balansare al lui $y = x.st$. Presupunem că $y.fb = 1$. Ce se întâmplă dacă se efectuează o rotaţie spre dreapta în jurul lui x? Rezultatul unei astfel de rotaţii este ilustrat în figura 7.

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

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

Se obţine rezultatul din fig. 8.

b) $x.fb = 2$. Deoarece debalansarea s-a făcut prin inserţia unui nod într-un arbore AVL valid rezultă că inserţia s-a făcut la dreapta lui x. Se verifică factorul de balansare al lui $y = x.dr$. Presupunem că $y.fb = -1$. Dacă efectuăm o rotaţie la stânga în jurul lui x se obţine un efect similar ca la punctul a. Soluţia este deci:

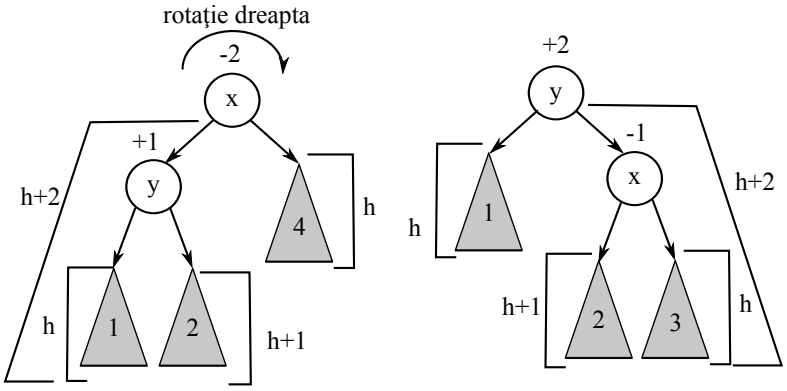


Figure 7: În cazul 2, o rotaţie simplă nu rezolvă problema debalansării arborelui

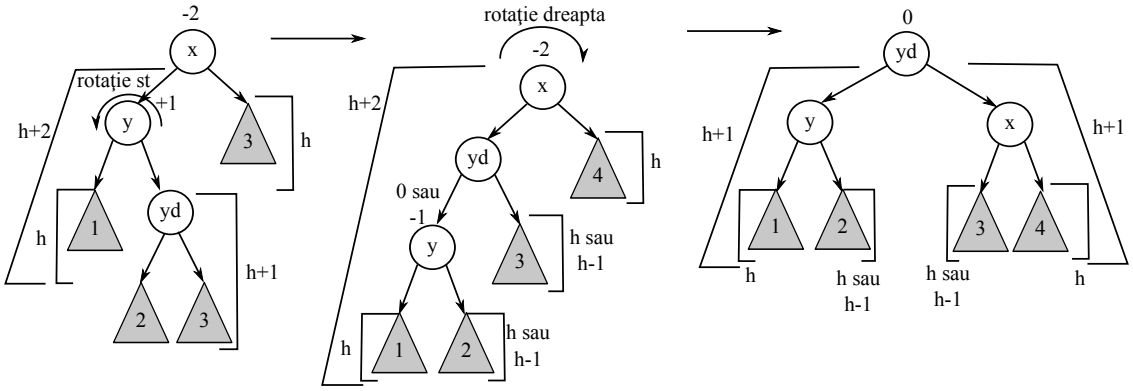


Figure 8: Rebalansarea arborelui în cazul 2.

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

Un exemplu de inserţie într-un arbore AVL este prezentat în figura 9.

2.2.1 Ştergerea unui nod

Ştergerea unui nod se realizează la fel ca pentru arborii binari de căutare obişnuiţi. Dacă notăm cu z nodul care trebuie şters atunci:

- dacă z are cel mult un descendent, se porneşte recalcularea factorilor de balansare de la părintele lui z
- dacă z are doi descendenţi, atunci se determină y, succesorulnodului z, se înlocuieşte nodul y cu descendentul său drept, iar nodul z cu nodul y. În acest caz recalcularea factorilor de balansare începe de la părintele succesorului nodului şters.

Dacă nodul curent este x atunci factorul să de balansare de recalculează astfel:

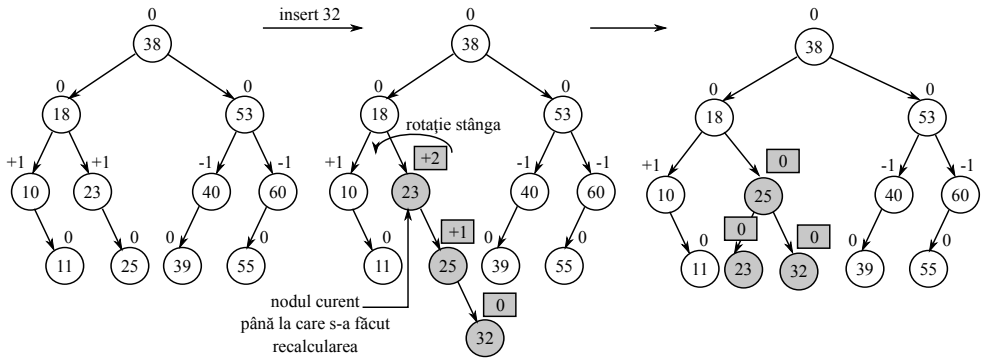


Figure 9: Inserţia nodului cu cheia 32 în arborele AVL din figură.

- Dacă ştergerea a avut loc pe stânga lui x - adică s-a ajuns la x de la fiul său stâng - atunci creşte factorul de balansare a al lui x cu 1
- Dacă ştergerea a avut loc la dreapta lui x, scade factorul de balansare al lui x cu 1.

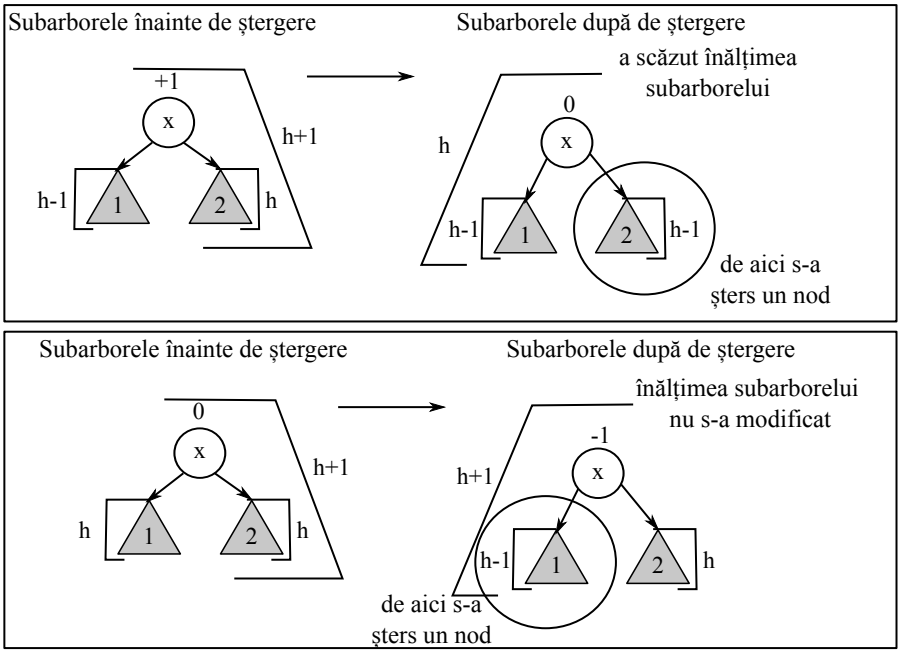


Figure 10: Modificarea înălţimii subarborelui curent după şteregerea unui nod.

Rebalansarea arborelui:

Spre deosebire de inserţie, algoritmul nu se opreşte atunci când factorul de balansare recalculat al nodului curent x este 0, deoarece în acest caz a scăzut înălţimea subarborelui de rădăcină x. Acest lucru, ilustrat în figura 10, produce modificări ale factorului de balansare şi la părintele său şi deci continuă urcarea în arbore.

În schimb, atunci când factorul de balansare al nodului x devine -1 sau $+1$, înseamnă că

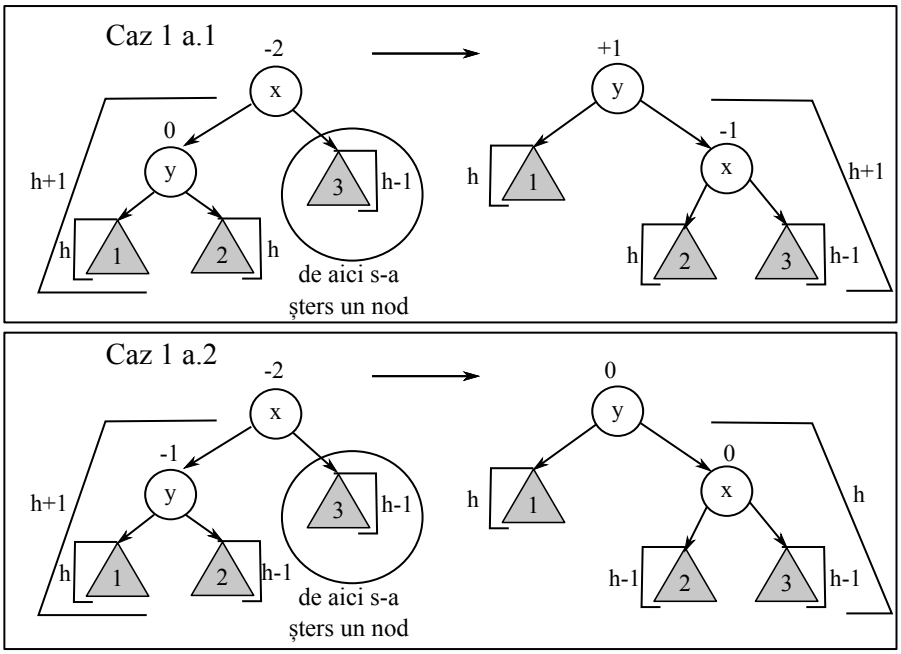


Figure 11: Rebalansarea unui arborele AVL după ştergerea unui nod, cazul 1 a.

doar s-a produs o scădere a înălţimii pe una dintre ramurile lui x, dar nu şi a subarborelui de rădăcină x (fig. 10), deci algoritmul de rebalansare se poate opri.

De asemenea, algoritmul nu se opreşte însă 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 la rădăcină.

Considerând nodul curent x, rebalansarea are următoarele cazuri:

Cazul 1

a. Dacă x are factorul de balansare nou -2 şi factorul lui $y = x.st$ este -1 sau 0, atunci rotaţie la dreapta în jurul lui x. În figura 11 este ilustrat modul de recalculare a factorilor de balansare pentru nodurile x şi y. Se observă şi următorul fapt important. Subarborele de rădăcină x a avut înainte de ştergere înălţimea $h+2$. În cazul în 1.a.1 din fig. 11, în care y a avut factorul de balansare 0 înainte de rebalansare, după rebalansare se observă că nu s-a modificat înălţimea subarborelui, care acum are rădăcina y, şi deci nu mai este necesară continuarea urcării în arbore.

   În schimb, daca y a avut factorul de balansare -1, cazul 1.a.2, se observă din figura 11, că după rebalansare, înălţimea 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 şi factorul de balansare al lui $y = x.dr$ este 0 sau 1 atunci rotaţie la stânga în jururl lui x. În figura 12 este ilustrat modul de recalculare a factorilor de balansare pentru nodurile x şi y. La fel ca în cazul 1.

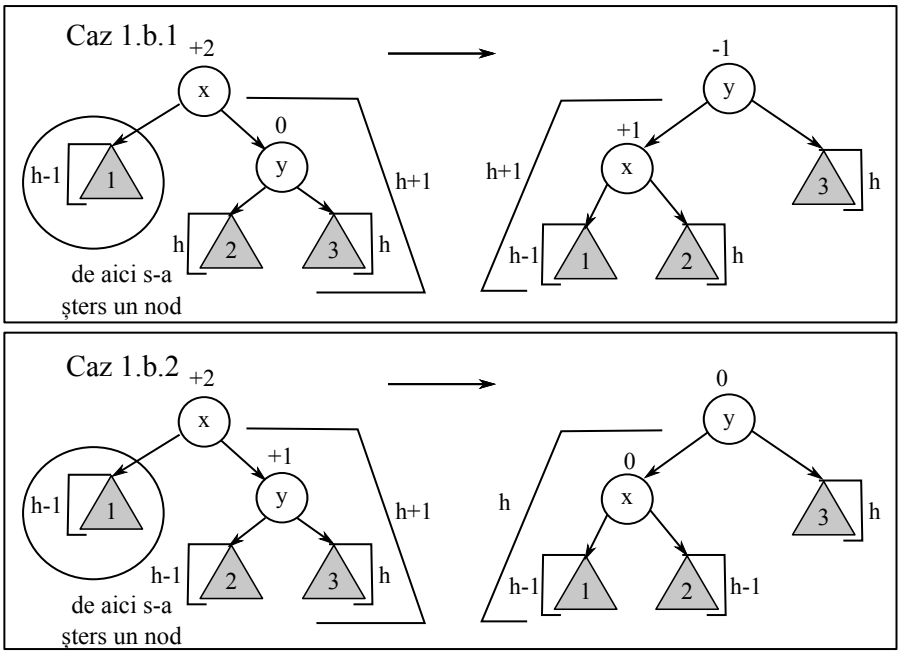


Figure 12: Rebalansarea unui arborele AVL după ştergerea unui nod, cazul 1 b.

a, se observă şi următoarele: Subarborele de rădăcină x a avut înainte de ştergere înălţimea $h+2$. În cazul în 1.b.1, în care y a avut factorul de balansare 0 înainte de rebalansare, după rebalansare se observă că nu s-a modificat înălţimea subarborelui, care acum are rădăcina y, şi deci nu mai este necesară continuarea urcării în arbore.

În schimb, daca y a avut factorul de balansare +1, cazul 1.b.2, se observă din fig. 12, că după rebalansare, înălţimea 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.

Cazul 2

a. Dacă x are factorul de balansare nou -2 şi nodul $y = x.st$ are factorul de balansare 1, atunci pentru rebalansare se efectuează:
   - Întâi rotaţie la stânga în jurul lui y
   - Apoi rotaţie la dreapta în jurul lui x.

b. Dacă x are factorul de balansare nou 2 şi $y = x.dr$ are factorul de balansare -1, atunci pentru rebalansare se efectuează:
   - Întâi rotaţie la dreapta în jurul lui y
   - Apoi rotaţie la stânga în jurul lui x.

În acest caz, după rebalansare scade înălţimea subarborelui curent, comparativ cu înălţimea avută înainte de ştergere. Este deci necesară continuarea algoritmului de la părintele nodului curent x.

MODULUL 5 - III - ARBORI ROŞU-NEGRU

1 Noţiuni generale

Definiţie: Un arbore roşu-negru (ARN) este un arbore binar de căutare în care fiecărui nod i se asociază o culoare - roşu sau negru - şi care are următoarele proprietăţi:

1. Fiecare nod este roşu sau negru - are deci un câmp suplimentar *color*
2. Rădăcina este neagră
3. Fiecare frunză este neagră şi NIL
4. Dacă un nod este roşu, ambii fii sunt negri $\Rightarrow$ părintele unui nod roşu este negru.
5. Pentru fiecare nod x, oricare drum de la nod la o frunză NIL se întâlneşte acelaşi număr de noduri negre (inclusiv frunza NIL şi exclusiv nodul de la care se porneşte). Acest număr se numeşte înălţimea neagră a subarborelui de rădăcină x. Notăm înălţimea neagră cu *bh - black height*.

Un exemplu de arbore roşu-negru este prezentat în figura 1.

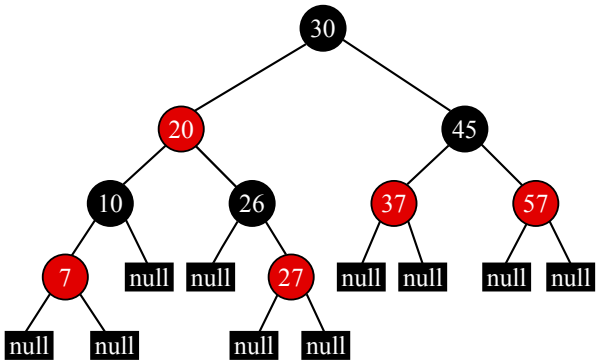


Figure 1: Exemplu de arbore roşu - negru.

Observaţii:

- 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 .

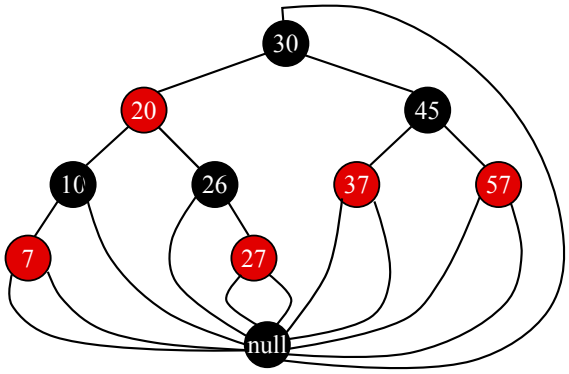


Figure 2: Exemplu de arbore roşu - negru cu santinelă.

- Înălţimea maximă a unui arbore ARN este $2\log_2(n+1)$.
  Se demonstrează prin inducţie că orice subarbore de rădăcină x conţine cel puţin $2^{bh(x)} - 1$ noduri interne. Notând cu h înălţimea arborelui şi cu r rădăcina, din proprietatea 4 se obţine $bh(r) \geq h/2$. 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 descendenţi frunze NIL. Un exemplu de ARN cu santinelă este prezentat în fig. 2.
- Pentru simplitate în continuare vom ignora în desene nodurile NIL.
- Datorită faptului că operaţiile de căutare, maxim, minim, succesor, predecesor depind de înălţimea h a arobrelui, înseamnă că aceste operaţii au complexitatea $O(\log_2 n)$.
- Operaţiile de inserţie şi ştergere sunt ceva mai complicate decât în cazul arborilor binari de căutare simpli, deoarece după inserţie/ştergere trebuie eventual refăcută structura de arbore roşu-negru.

Pentru refacerea proprietăţilor de arbore roşu-negru sunt necesare operaţii de recolorare a nodurilor şi de rotaţie.

1.1 Inserţia într-un arbore roşu-negru

Inserţia propriu-zisă a unui nod într-un arbore roşu-negru se realizează după acelaşi algoritm ca şi inserţia într-un arbore binar de căutare. Practic se porneşte de la rădăcină şi se compară la fiecare pas cheia nodului curent x cu cheia nodului z care se inserează, coborându-se în subarborele stâng dacă $z.info < x.info$ şi în cel drept altfel. Apoi se refac proprietăţile ARN.

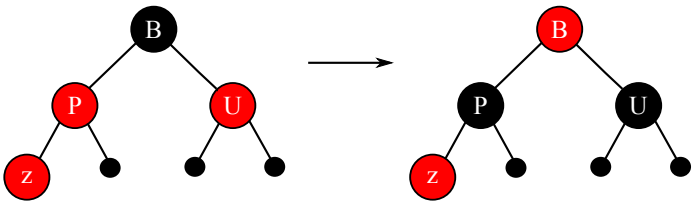


Figure 3: Cazul 1 pentru inserţie.

Observaţii:

- Întotdeauna nodul care se inserează are culoarea roşie
- În arborele T se consideră o santinelă T.Nil

Proprietăţile care pot fi neîndeplinite în cazul inserţiei:

- Proprietăţile 1, 3 şi 5 se păstrează, datorită faptului că se inserează un nod roşu, care are ca descendenţi două frunze NIL, deci se leagă de nodul santinelă T.nil
- Proprietatea 2 poate fi contrazisă dacă nodul inserat este chiar rădăcina sau ca urmare a cazului 1, care va fi discutat în cele ce urmează. Pentru rezolvarea acestei situaţii este suficientă colorarea rădăcinii cu negru.
- Proprietatea 4 poate fi contrazisă, dacă părintele de care s-a legat noul nod are culoarea roşie. În acest caz trebuie refăcută proprietatea. Există în această situaţie 3 cazuri care vor fi discutate în continuare.

Funcţia de refacere a proprietăţilor RN este apelată doar dacă părintele nodului inserat este roşu.

Observaţii:

- Notăm cu z nodul inserat, cu P părintele lui x. Datorită faptului că P are culoarea roşie, deja înainte de inserţie, iar inserţia se produce într-un arbore roşu-negru valid, înseamnă că P nu este rădăcină şi deci există un nod părintel al lui $P \neq T.nil$ pe care îl notăm cu B. Notăm cu U unchiu (fratele părintelui).
- 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 inserţia pe stânga bunicului B.

Cazurile de refacere a proprietăţii 4 în urma inserţiei:

Cazu 1: unchiul lui z este roşu (fig. 3).

Deoarece U, P sunt roşii rezultă că B are culoarea neagră, altfel s-ar contrazice proprietatea 4, $\Rightarrow$ este suficientă recolorarea P, U, B, adică P şi U devin negri iar B roşu.

În urma acestei modificări poate avea loc o contrazicere a proprietăţii 4 pentru B şi

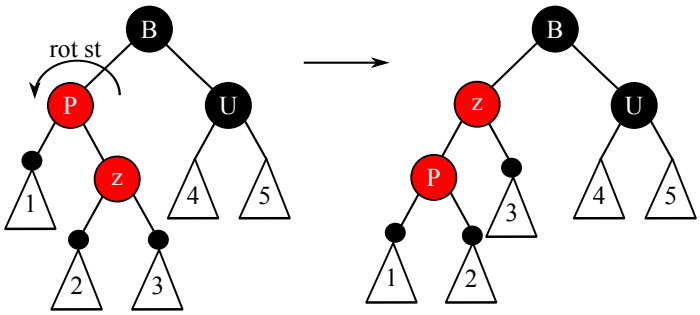


Figure 4: Cazul 2 pentru inserţie.

părintele său, deci se reia procedura de refacere a proprietăţilor roşu-negru, considerând de data aceasta $z = B$. Acelaşi procedeu se aplică şi în cazul în care z se află pe dreapta lui P.

Atunci când unchiul U este negru se disting cazurile 2 şi 3, care diferă prin poziţionarea nodului z relativ la P şi la B.

Cazul 2: unchiul este negru şi

a) z se află pe dreapta lui P, iar P se află la stânga lui B - fig.4

b) z se află pe stânga lui P, iar P se află la dreapta lui B

Soluţionare: în cazul a) - rotaţie la stânga după P, în cazul b) - rotaţie la dreapta după P. Nu se soluţionează complet, ci se trece practic în cazul 3, în care refacerea începe de la nodul care prin rotaţie a coborât, deci de la $z = P$.

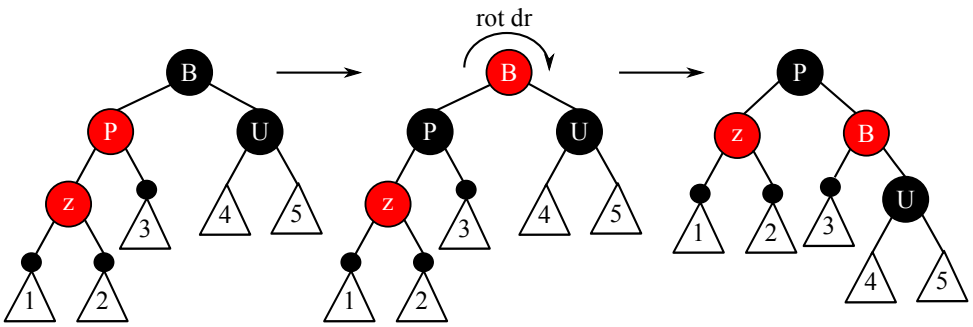


Figure 5: Cazul 3 pentru inserţie.

Cazul 3: unchiul este negru şi

a) z se află pe stânga lui P, iar P se află la stânga lui B - fig.5

b) z se află pe dreapta lui P, iar P se află la dreapta lui B

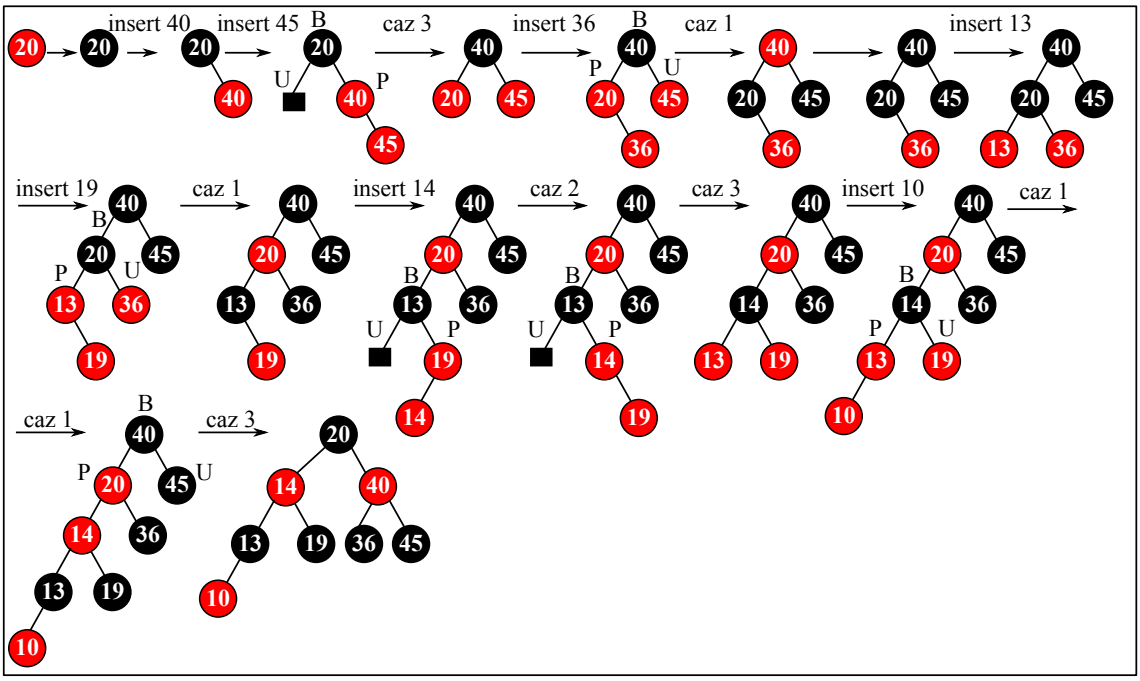


Figure 6: Inserarea cheilor 20, 40, 45, 36, 13, 19, 14, 10 într-un ARN iniţial vid.

Soluţionare::

1. Colorare B cu roşu, colorare P cu negru.
2. pentru situaţia a) Rotaţie la dreapta în jurul lui B, pentru situaţia b) Rotaţie la stânga în jurul lui B.

Un exemplu de construcţie a unui ARN prin inserţie succesivă de chei este prezentat în figura 6.

Algoritm: inserarea nodului z în arborele T

```
ARN_INSERT(T,z)
    y = T.Nil
    x = T.rad
    cat timp x ≠ T.Nil
              y = x
              daca z.info < x.info atunci
                    x = x.st
              altfel x = x.dr
              sfarsit daca
    sfarsit cat timp
    z.p = y
    daca y = T.Nil atunci T.rad = z
    altfel
              daca z.info < y.info atunci
```

```
                    y.st = z
               altfel
                    y.dr = z
               sfarsit daca
          sfarsit daca
          z.st = T.Nil
          z.dr = T.Nil
          z.color = rosu
          ARN_INSERT_REPARA(T,z)
     RETURN
```

Algoritm: refacerea proprietăţilor de arbore roşu-negru

```
     ARN_INSERT_REPARA(T,z)
          cat timp z.p.color = rosu
               daca z.p = z.p.p.st atunci
                    U = z.p.p.dr
_____________________________________________ caz 1
                    daca U.color = rosu atunci
                         z.p.color = negru
                         U.color = negru
                         z.p.p.color = rosu
                         z = z.p.p
                    altfel
_____________________________________________ caz 2
                         daca z = z.p.dr atunci
                              z = z.p
                              ROT_ST(T, z)
                         sfarsit daca
_____________________________________________ caz 3
                         z.p.color = negru
                         z.p.p.color = rosu
                         ROT_DR(T,z.p.p)
                    sfarsit daca
                    altfel
                    //similar dar simetric pentru nodul z
                    //aflat la stânga bunicului
               sfarsit daca
          sfarsit cat timp
          T.rad.color = negru
     RETURN
```

Complexitate: Inserţia are aceeaşi complexitate ca şi în cazul arborilor binari de căutare simpli, deci $O(h)$. Cum înălţimea este cel mult $2\log_2(n+1)$ rezultă o complexitate

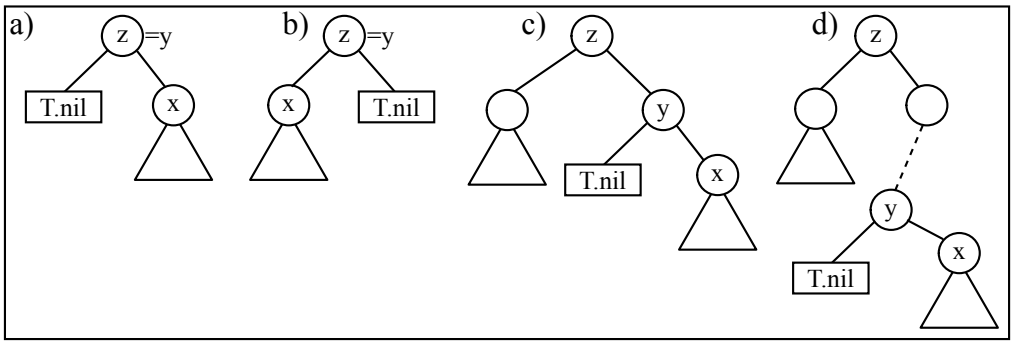


Figure 7: Cazurile pentru ştergerea nodului z. a) z nu are descendent stâng, b) z nu are descendent drept, c) z are 2 descendenţi nenuli şi succesorul lui z este fiu al lui z, d) z are doi descendenţi şi succesorul lui z nu este fiu al lui z.

$O(\log_2 n)$, iar algoritmul de refacere al proprietăţilor de arbore roşu-negru porneşte de jos înspre rădăcină pe o ramură, deci complexitatea este tot $O(\log_2 n)$. Rezultă complexitatea pentru inserţie este $O(\log_2 n)$.

1.2 Ştergerea dintr-un arbore roşu-negru

Operaţia de ştergere dintr-un arbore roşu-negru este mai complicată decât operaţia de inserţie. În prima etapă, operaţia de ştergere are la bază ştergerea dintr-un arbore binar de căutare, cu câteva modificări, după care trebuie efectuată o operaţie de refacere a proprietăţilor de arbore roşu-negru.

Reamintim faptul că, în cazul ştergerii unui nod z dintr-un arbore binar de căutare T se iau în considerare cazurile:

1. $z.st = T.Nil$ - fig.7 a)
2. $z.dr = T.Nil$ - fig.7 b)
3. z are doi fii diferiţi de $T.Nil$ şi $y = sucesor(T, z)$
   a. y descendent direct al lui z - fig.7 c)
   b. y nu este descendent direct al lui z - fig.7 d)

Observaţii:

- Algoritmul de refacere depinde în cazurile 1 şi 2 de culoarea nodului şters z şi de culoarea pe care o are x, iar în cazul 3 de culoarea originală a lui y (după ştergere y preia culoarea lui z) şi de culoarea lui x, unde x este în primele 2 cazuri nodul cu care se înlocuieşte z, iar în al treilea caz este fiul drept al lui y.
- În cazurile 1 şi 2, prin ştergerea nodului z şi înlocuirea cu unul dintre descendenţii direcţi, se poate produce o modificare a culorii în poziţia deţinută anterior de z,

ceea ce poate duce la violarea proprietăţilor de arbore roşu-negru. La fel în cazul 3, la înlocuirea lui y cu x.

- Nodul x este acela care se deplasează în poziţia deţinută anterior de nodul y. Se observă faptul că, x poate fi santinela $T.Nil$

Refacerea proprietăţilor de arbore roşu-negru:

- Dacă nodul z în cazurile 1 şi 2, respectiv y în cazul 3, a avut culoarea iniţială roşu, atunci se păstrează proprietăţile de arbore RN, deoarece
  1. Nu s-a modificat înălţimea neagră pe nici o ramură prin eliminarea unui nod roşu
  2. Nici un nod roşu nu a căpătat un fiu roşu.
  3. Deoarece nici nodul z, nici nodul y în cazul 3, nu au fost rădăcină (culoarea originală a fost roşie) nu s-a modificat nici culoarea rădăcinii, care a rămas neagră
- Rezultă deci că, proprietăţile de arbore RN sunt violate doar dacă a fost eliminat un nod negru. Există mai multe cazuri care trebuie tratate.
- Prin eliminarea unui nod negru din arbore
  1. În primul rând se micşorează înălţimea neagră pe ramura respectivă
  2. Poate apărea vecinătate între două noduri roşii

Cazul 0: culoarea nodului x cu care s-a făcut înlocuirea este roşie. În acest caz singurul lucru care trebuie făcut este recolorarea lui x în negru. Acest lucru rezolvă atât problema (1) cât şi problema (2).

Notaţii: P=părintele lui x, F=fratele lui x, bh = înălţimea neagră originală a subarborelui cu rădăcina P (exclusiv nodul 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.

Cazul 1: F are culoarea roşie - fig. 8

Facem următoarele observaţii:

- Datorită faptului că înainte de ştergere arborele era RN valid, din F roşu şi F descendent direct al lui P rezultă culoarea lui P este neagră.
- Copiii lui F sunt ambii negri (eventual T.nil)

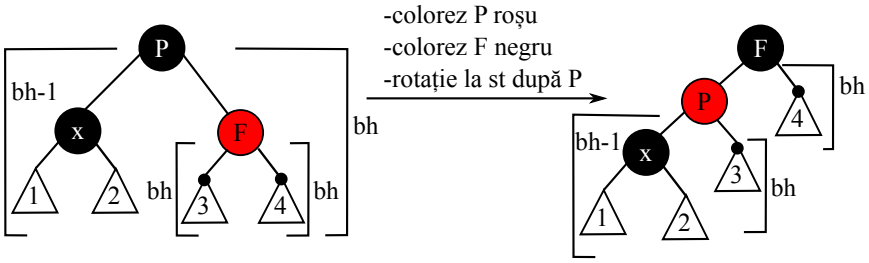


Figure 8: Cazul 1 în ARN

- Prin ştergere s-a micşorat înălţimea neagră pe partea stângă a lui P, adică subarborele drept are înălţimea neagră bh, iar subarborele stâng $bh-1$. Cum F este roşu rezultă că ambii săi subarbori au înălţimea bh şi fiii lui F sunt ambii negri.

Refacerea proprietăţilor de arbore RN:

- Colorare P cu roşu şi colorare F cu negru.
- Rotaţie la stânga în jurul lui P. (Dacă x se află pe dreapta lui P, atunci rotaţia este la dreapta).

În această situaţie F, acum negru, urcă în locul lui P. La dreapta lui F înălţimea neagră a rămas bh. Înălţimea neagră a subarborelui drept al lui P este bh, iar înălţimea neagră subarborelui stâng al lui P este $bh-1$. Deci problema în nodul P a rămas, dar fiul drept al lui P nu mai este roşu ci negru, ceea ce conduce la unul dintre cazurile 2, 3 sau 4.

Deci din cazul 1 se trece la cazul 2, 3 sau 4!

În figura 8 este ilustrată procedura de refacere a proprietăţilor RN în cazul 1.

Cazul 2: F este negru şi ambii fii ai săi sunt negri (eventual T.Nil) - fig. 9.

Observaţii:

- $F \neq T.Nil$, pentru că pe partea stângă a lui P s-a şters un nod negru, iar înainte de ştergere pe ambele părţi ale lui P înălţimea neagră era aceeaşi şi cel puţin 1.
- P poate fi roşu sau negru
- Prin ştergere s-a micşorat înălţimea neagră pe partea stângă a lui P, adică subarborele drept are înălţimea neagră bh, iar subarborele stâng $bh-1$. Cum F este negru rezultă că ambii săi subarbori au înălţimea $bh-1$.

Refacerea proprietăţilor de arbore RN:

- Se colorează F roşu $\Rightarrow$ înălţimea neagră a subarborelui drept al lui P este $bh-1$, deci egală cu cea a subarborelui stâng, dar înălţimea neagră a arborelui P este mai mică decât înainte de ştergere, deci cu o unitate mai mică decât a fratelui său (dacă există). Această problemă se rezolvă astfel:

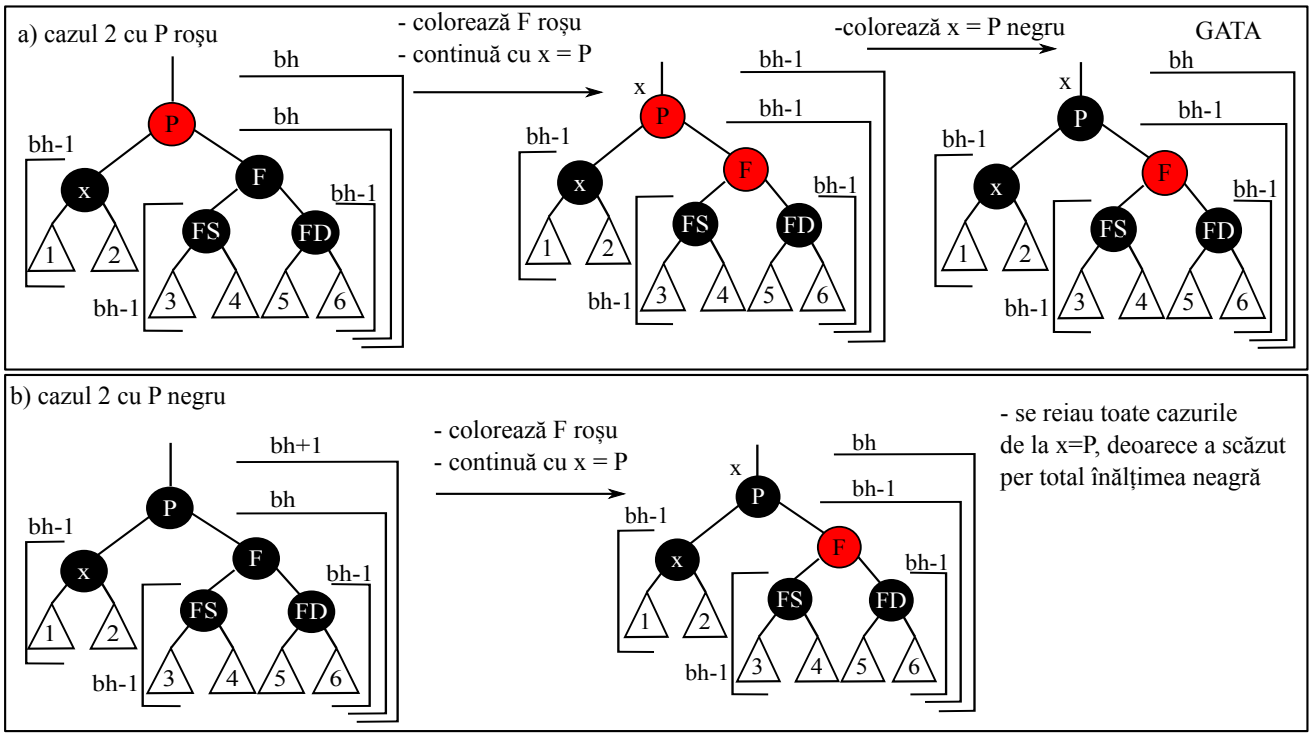


Figure 9: Cazul 2 în ARN

– Dacă P este roşu atunci este suficient să colorăm P cu negru - fig.9 a).

– Altfel problema de refacere a proprietăţilor RN se reia pentru subarborele care are ca rădăcină părintele lui P. Se observă din figura 9 b) că, în acest caz înălţimea neagră a întregului arbore care are rădăcina P a scăzut cu 1 şi trebuie reechilibrat de la acest nod în sus.

Cazul 3: a) F are culoarea neagră, fiul stâng al lui F notat cu FS este roşu, iar cel drept notat cu FD este negru. (fig. 10). Respectiv cazul 3b) simetric, atunci când F este pe stânga lui P şi x pe dreapta, F este negru, FD este roşu şi FS este negru. Cazul 3b) se soluţionează simetric cu 3a) prezentat mai jos.

Observaţii:

- Culoarea lui P poate fi roşie sau neagră.
- F sigur este diferit de $T.Nil$, pentru că pe partea stângă a lui P s-a şters un nod negru, iar înainte de ştergere pe ambele părţi ale lui P înălţimea neagră era aceeaşi şi cel puţin 1.
- Înălţimea neagră a subarborelui stâng al lui P este $bh - 1$, înălţimea neagră a subarborelui cu rădăcina F este bh. Înălţimile negre ale subarborilor FS şi FD sunt $bh - 1$.

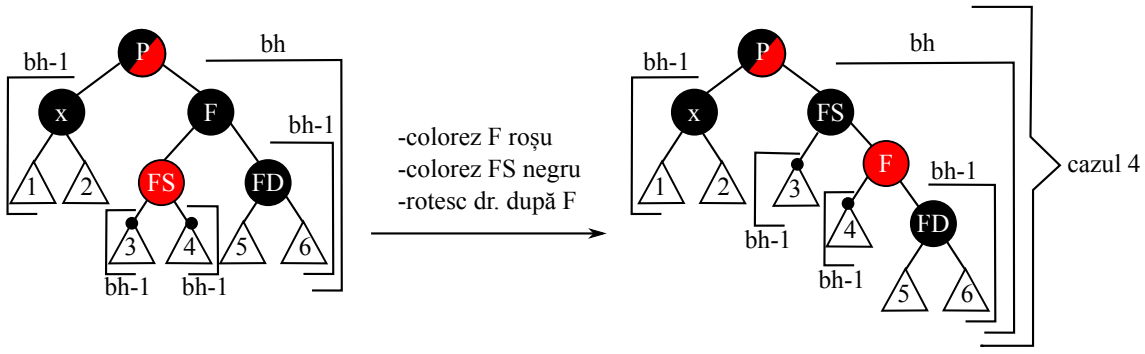


Figure 10: Cazul 3 în ARN. P poate avea culoarea roşie sau culoarea neagră.

Refacerea proprietăţilor de arbore RN: - fig. 10

- Recolorare F şi $FS \Rightarrow F$ devine roşu şi FS devine negru $\Rightarrow$ înălţimea neagră a lui FS devine bh şi înălţimea neagră a lui FD devine $bh - 1$.
- Rotaţie la dreapta în jurul lui $F \Rightarrow FS$ urcă în locul lui F, înălţimea neagră a lui FS devine bh, înălţimea neagră a lui F devine $bh - 1 =$ înălţimea neagră a fiului stâng al lui FS.

Problema la care am ajuns în continuare este refacerea proprietăţilor RN pentru cazul 4.

Cazul 4: F are culoarea neagră iar fiul drept notat cu FD este roşu. Fiul stâng, FS, poate avea oricare dintre cele două culori. (fig. 11). În cazul simetric, atunci când x este pe dreapta lui P, FS este roşu.

Observaţii:

- Culoarea lui P poate fi roşu sau negru.
- F sigur este diferit de $T.Nil$, pentru că pe partea stângă a lui P s-a şters un nod negru, iar înainte de ştergere pe ambele părţi ale lui P înălţimea neagră era aceeaşi şi cel puţin 1.
- Înălţimea neagră a subarborelui stâng al lui P este $bh - 1$, înălţimea neagră a subarborelui cu rădăcina F este bh. Înălţimile negre ale lui FS şi FD sunt $bh - 1$.

Refacerea proprietăţilor de arbore RN: - fig. 11.

- Colorare F cu culoarea lui P
- Colorare P cu negru
- Colorare FD cu negru
- Rotaţie la stânga în jurul lui P (respectiv dreapta în cazul simetric).

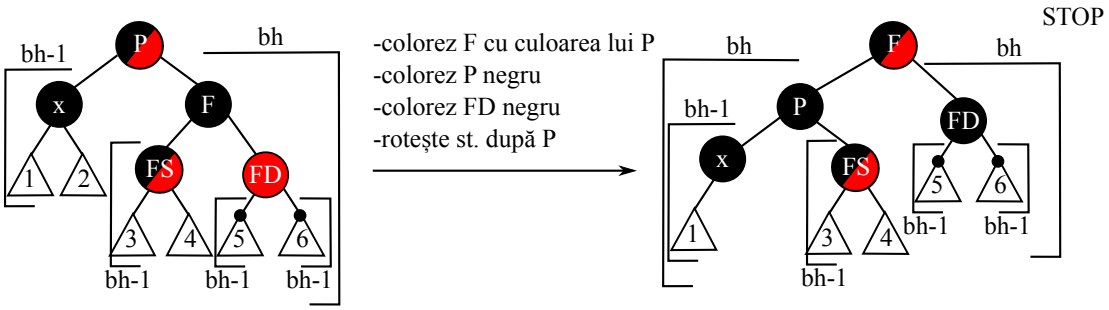


Figure 11: Cazul 4 în ARN

Prin aceste operaţii rezultă: creşterea cu o unitate a înălţimii negre a subarborelui F, adică $bh(F) = bh + 1$. Dar $bh(x) = bh - 1$. Prin rotaţia în jurul lui P, P fiind acum negru, se rebalansează arborele.

Pseudocodul pentru ştergere poate fi găsit în bibliografia recomandată.

MODULUL 6 - ÎMBOGĂŢIREA ARBORILOR ROŞU-NEGRU

1 Noţiuni generale

Îmbogăţirea structurilor de date se realizează în general cu scopul de-a putea efectua şi alte operaţii -cereri - în afara celor uzuale, descrise în capitolele anterioare, sau pentru a creşte eficienţa celor uzuale. Îmbogăţirea unei structuri de date se referă la ataşarea de informaţii suplimentare elementelor structurii - adică adăugarea de câmpuri suplimentare fiecărui nod, astfel încât să permită efectuarea operaţiilor dorite. Aceste câmpuri suplimentare trebuie actualizate odată cu modificarea mulţimii dinamice stocate cu ajutorul structurii de date.

Informaţia suplimentară adăugată nodurilor structurii de date poate fi o valoare, dar poate fi şi o referinţă (pointer)!

Procesul de îmbogăţire a unei structuri de date are 4 etape (Cormen):

1. Alegerea unei structuri de date de bază, pe care urmează să o îmbogăţim: de exempu un arbore roşu-negru.
2. Determinarea informaţiei suplimentare care trebuie păstrată în structura aleasă.
3. Verificarea faptului că, informaţia suplimentară aleasă poate fi actualizată pe parcursul operaţiilor de bază de modificare a structurii, respectiv inserţie/ştergere, fără a creşte complexitatea acestora. Va fi enunţată mai jos o teoremă, care permite în anumite condiţii bine definite, demonstrarea păstrării complexităţii logaritmice în cazul îmbogăţirii unui arbore binar care se autoechilibrează.
4. Dezvoltarea de noi operaţii utile asupra structurii de date îmbogăţite.

Un exemplu clasic pentru îmbogăţirea arborilor roşu-negru sunt arborii pentru statistici de ordine, prezentaţi în continuare.

2 Arbori pentru statistici de ordine

În cadrul cursului de algoritmică, odată cu descrierea algoritmului *Quicksort*, s-a discutat şi despre statistici de ordine. De fapt *statistica de ordin i* a unei mulţimi de n elemente

$a_1, a_2, \ldots, a_n$, este acel element, care s-ar afla în şirul sortat al elementelor pe poziţia a i-a. S-a discutat cum poate fi modificat algoritmul *Quicksort*, astfel încât această problemă să fie rezolvată în timp liniar.

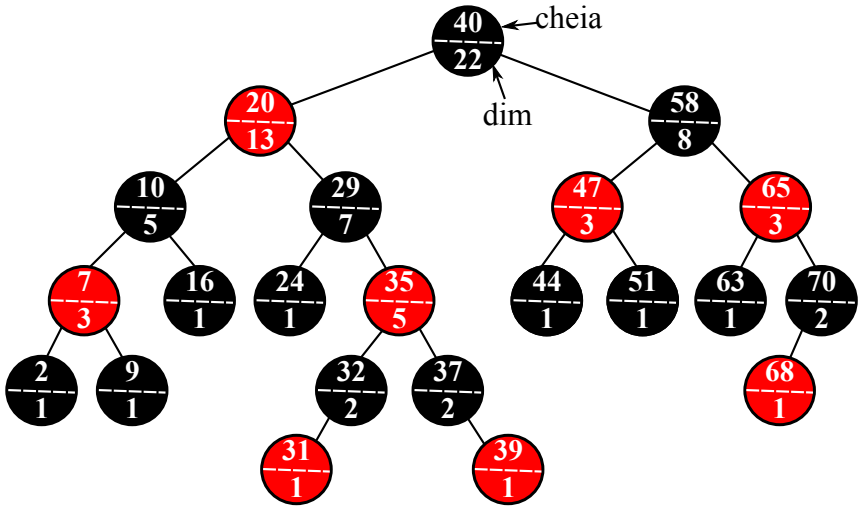


Figure 1: Exemplu de ARN pentru statistici de ordine. Câmpul dimensiune = *dim* al unui nod x = numărul de noduri din subarborele de rădăcină x

Prin îmbogăţirea structurilor de date se poate obţine însă o complexitate $O(\log_2 n)$ pentru determinarea statisticii de ordin i. În plus, pentru fiecare element din şir, poate fi calculat tot cu o complexitate $O(\log_2 n)$ rangul său, adică poziţia pe care s-ar afla în şirul sortat.

Definiţie: Un *arbore pentru statistici de ordine* este un ARN, în care fiecare nod x conţine un câmp suplimentar, $x.dim$, care reprezintă numărul de noduri - fără frunzele nil - ale subarborelui cu rădăcina x. În cazul nodului santinelă $T.nil$, se consideră $T.nil.dim = 0$. Se observă uşor relaţia:

$$x.dim = x.st.dim + x.dr.dim + 1$$

În figura 1 este reprezentat un arbore roşu-negru pentru statistici de ordine.

Pentru un astfel de arbore îmbogăţit cu informaţia *dim* pot fi definite două operaţii noi:

(I) Căutarea elementului de rang i = elementul a cărui cheie s-ar afla pe poziţia i în şirul sortat al cheilor din arbore.

(II) Determinarea rangului unui element = poziţia pe care s-ar afla cheia elementului respectiv în şirul sortat al cheilor din arbore.

Deoarece prin parcurgerea în inordine a unui arbore binar de căutare se obţine şirul sortat al cheilor, rangul unui element este de fapt poziţia sa în urma parcurgerii în inordine a arborelui.

(I) Căutarea elementului de rang i

Observaţii

- Pornind de la rădăcina $T.rad$, în cazul parcurgerii în inordine, întâi se parcurg toate elementele din subarborele drept, care sunt în număr de $T.rad.st.dim$, după care se afişază cheia rădăcinii, iar apoi se parcurg elementele din subarborele drept. Deci rangul lui $T.rad$ este $T.rad.st.dim + 1$.
- Pornind de la un nod x, rangul lui x în cadrul mulţimii formate din elementele aflate în subarborele x este $x.st.dim + 1$.

Putem astfel determina elementul de rang i prin următorul algoritm:

```
ArbStat_SELECT(T,i)
   x = T.rad
   cat timp x ≠ T.nil
        rang = x.st.dim + 1
        daca i = rang atunci
             RETURN x
        sfarsit daca
        daca i<rang atunci
             x=x.st
        altfel
             x=x.dr
             i=i-rang
        sfarsit daca
   sfarsit cat timp
 RETURN x
```

Exemplul: În arborele din figura 1 se caută nodul de rang $r = 16$. Exemplul este ilustrat în figura 2

Notăm cu x nodul curent.

- Iniţial $x = T.rad$, deci $rang = x.st.dim+1 = 13+1=14$. Rădăcina se află pe poziţia 14 în parcurgerea în inordine, deoarece trebuie întâi să se parcurgă toate cele 13 noduri din subarborele stâng.
- $rang < r \Rightarrow$ cobor pe dreapta în arbore prin $x = x.dr$ iar variabila $r = r - rang = 2$, adică se caută elementul de rang 2 în subarborele de răcină $x.dr$, care are cheia 58. Rădăcina se află pe locul 14, iar nodul pe care îl caut se află cu 2 poziţii mai la dreapta de rădăcină, deci pe poziţia 2 în subarborele drept.
- În continuare nodul curent este $x = T.rad.dr$, în această situaţie $rang = x.st.dim + 1 = 4$, deci x se află pe poziţia 4 în subarborele curent, dar nodul căutat se află pe poziţia 2 în acest subarbore.

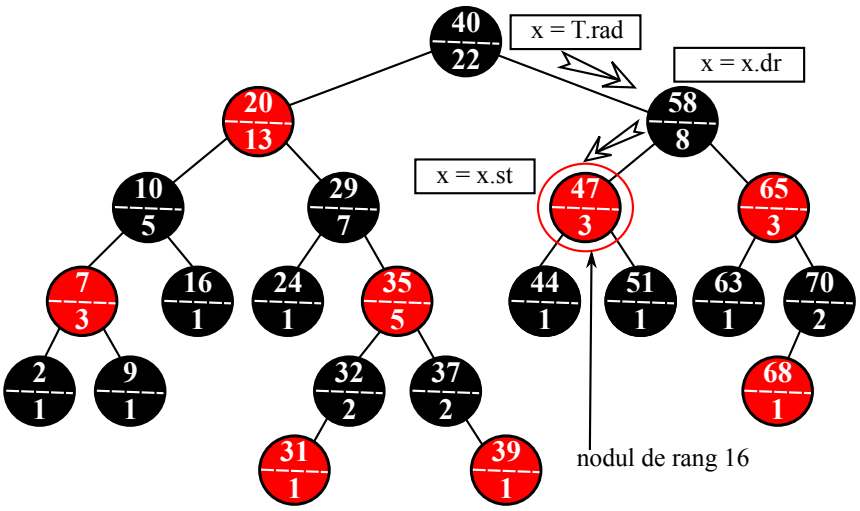


Figure 2: Determinarea elementului cu rangul 16.

- Se intră deci pe prima ramură a instrucţiunii condiţionale şi $x = x.st$ Astfel nodul curent este $T.rad.dr.st$ cu cheia 47 şi variabila $rang = x.st.dim + 1 = 2$
- Nodul curent x cu cheia 47 are rangul $2 = r \Rightarrow$ am ajuns la nodul căutat.

Nodul cu rangul 16 este cel cu cheia 47. Se observă uşor faptul că şirul sortat al cheilor este $\{2, 7, 9, 10, 16, 20, 24, 29, 31, 32, 35, 37, 39, 40, 44, 47, 51, 58, 63, 65, 68, 70\}$, iar al 16-lea element este 47.

Complexitate: algoritmul coboară la fiecare apel iteraţie cu un nivel în arbore. Rezultă faptul că timpul de execuţie este direct proporţional cu înălţimea arborelui, deci este $O(\log_2 n)$.

(II) Determinarea rangului unui element

Observăm următoarele lucruri:

- Dacă la nodul x se ajunge doar prin coborârea în arbore pe descendenţi stângi, atunci rangul lui este chiar $x.st.dim + 1$, adică rangul său în cadrul subarborelui de rădăcină x.
- Dacă x se află la dreapta părintelui său, adică $x.p.dr = x$, atunci în parcurgerea în inordine subarborelui de rădăcină $x.p$, înaintea lui x se vor afla $x.p.st.dim + 1$ elemente.

Putem afla astfel rangul unui nod x în modul următor:

- Pornim de la nodul x către rădăcină, iniţializând rangul cu $x.st.dim + 1$

- Atunci când nodul curent pe drumul spre rădăcină se află pe stânga părintelui său, se lasă rangul nemodificat
- Atunci când nodul curent pe drumul spre rădăcină se află pe dreapta părintelui său, la rang se adaugă $x.p.st.dim + 1$
- Algoritmul se încheie atunci când nodul curent l-a care s-a ajuns este chiar rădăcina

Algoritm

```
ArbStat_RANG(T,x)
    rang = x.st.dim + 1
    cat timp x ≠ T.rad
        y = x.p
        daca x = y.dr atunci
              rang = rang + y.st.dim + 1
        sfarsit daca
        x = y
    sfarsit cat timp
 RETURN rang
```

Exemplul: În arborele din figura 1 se determină rangul nodului x cu cheia 32.

- $x = x.p.st$, deci poziţia lui x în parcurgerea în inordine a arborelui de rădăcină $x.p$ este aceeaşi cu poziţia în parcurgerea în inordine a subarborelui cu rădăcina $x.p$, adică $x.st.dim + 1 = 2$, deci $rang = 2$
- nodul curent devine $x = x.p$ (adică nodul cu cheia 35), $rang = 2$. Noul nod curent cu cheia 35 se află pe dreapta părintelui, deci poziţia lui în parcuregrea în inordine a subarborelui de rădăcină $x.p$ este poziţia lui $x.p$ + rangul în subarborele de rădăcină x, adică $x.p.st.dim + 1 + rang = 4$.
- nodul curent devine din nou $x = x.p$ (adică nodul cu cheia 29). Acest nod se află pe dreapa părintelui, deci $rang = rang + x.p.st.dim + 1 = 4+5+1=10$
- se trece la nodul părinte, care este nodul cu cheia 20, care se află pe stânga rădăcinii
- Rezultatul final este: nodul cu cheia 32 are $rang = 10$, deci se află pe poziţia 10 în parcurgerea în inordine a arborelui.

Complexitate: se parcurge arborele din nodul x către rădăcină, deci complexitatea este $O(\log_2 n)$

Refacerea informaţiei dimensiunii subarborilor după operaţii de modificare

1. **La inserţie:** la fiecare nod care este parcurs pe drumul spre locul de inserţie se incrementează câmpul *dim* cu o unitate. Nodul nou adăugat va avea câmpul $dim = 1$.

2. **La ştergere:** se parcurge arborele de la părintele nodul z care a fost şters (dacă acesta a avut cel mult un descendent) respectiv de la părintele succesorului lui z, către rădăcină şi pentru fiecare nod de pe acest drum, câmpul *dim* se decrementează cu o unitate.

3. **La refacerea proprietăţilor de ARN**, operaţia de rotaţie produce modifiări în numărul de noduri ale unui subarbore în modul următor: considerăm rotaţia la dreapta în jurul nodului x din figura 3.

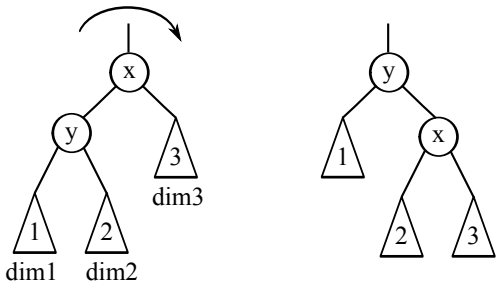


Figure 3:

Se observă din fig.3 că iniţial $x.dim = y.dim + dim3 + 1$. După rotaţie y urcă în locul lui x, dar numărul de noduri din subarborele considerat rămâne nemodificat, deci:

- $y.dim = x.dim$ (valoarea originală a câmpului).
- x coboară în dreapta lui y şi are noua valoare a câmpului dimensiune: $x.dim = dim2 + dim3 + 1 = x.st.dim + x.dr.dim + 1$

Deci la funcţia de rotaţie trebuie adăugate aceste două linii de cod:

$y.dim = x.dim$
$x.dim = x.st.dim + x.dr.dim + 1$

Etapele îmbogăţirii în cazul arborilor pentru statistici de ordine

1. Alegerea unei structuri de date de bază: un ARN
2. Determinarea informaţiei suplimentare: câmpul dimesiune = nr. de noduri di subarborele curent
3. Verificarea actualizării în timp logaritmic: - s-a arătat mai sus
4. Dezvoltarea de noi operaţii utile asupra structurii de date îmbogăţite: funcţiile: Stat_SELECT şi Stat_RANG.

3 Îmbogăţirea arborilor RN - teorema complexităţii

Următoarea teoremă demonstrează faptul că, dacă informaţia suplimentară păstrată în fiecare nod respectă anumite condiţii, atunci complexitatea operaţiilor de inserţie/ştergere într-un arbore RN îşi păstrează complexitatea nemodificată, adică $O(\log_2 n)$.

Teoremă: Considerăm un atribut/câmp suplimentar f prin care se îmbogăţeşte un arbore roşu-negru T cu n noduri. Dacă valoarea câmpului f al oricărui nod x depinde doar de informaţiile din nodurile x, $x.st$, $x.dr$ şi eventual de valorile $x.st.f$ şi $x.dr.f$, atunci valoarea câmpului suplimentar poate fi actualizată la orice operaţie de inserţie/ştergere fără a afecta complexitatea $O(\log_2 n)$ a acestor operaţii. (Cormen)

Demonstraţie: În cazul inserţiei: dacă s-a inserat nodul x ca descendent al nodului $x.p$, valoarea lui $x.f$ se calculază instantaneu, deoarece depinde doar de valorile din x şi de santinela $T.nil$. Modificarea adusă de inserarea lui x se poate reflecta asupra părintelui $x.p$. Modificarea lui $x.p.f$ este de asemenea $O(1)$, iar modificările se refelectă doar asupra părintelui acestuia. Astfel se parcurge arborele de la x către rădăcină şi se actualizează doar câmpul f pentru nodurile aflate pe acest drum. Ceea ce duce la complexitate $O(\log_2 n)$. În cazul operaţiei de refacere a proprietăţilor RN, apar un număr limitat de rotaţii, la care doar câmpurile a două noduri sunt actualizate, iar modificările acestora se propagă doar către părinte şi astfel, iar către rădăcină într-un timp $O(\log_2 n)$. Astfel per total se păstrează complexitatea $O(\log_2 n)$.

Aceeaşi argumentaţie se poate efectua în cazul operaţiei de ştergere.

4 Arbori de intervale

Intervalele pot fi utilizate pentru reprezentarea şi gestionarea evenimentelor/proceselor care se desfăşoară în timp. Un arbore binar de căutare, care are ca informaţie un interval, poate fi utilizat de exemplu pentru căutarea într-o bază de date conţinând evenimente ce se desfăşoară în timp. Pentru a obţine operaţii de inserţie, ştergere şi căutare eficientă se poate utiliza ca structură de bază un arbore roşu-negru.

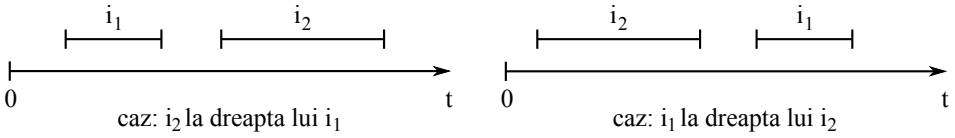
Un interval închis $i = [t_1, t_2]$ este caracterizat prin cele două capete ale sale t_1 şi t_2.

Notăm: $i.low = t_1$ şi $i.high = t_2$.

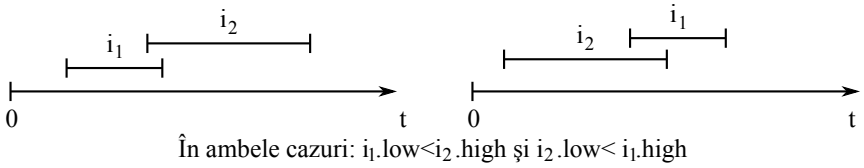
Putem compara două intervale prin verificarea dacă se intersectează sau nu.

Observăm următoarele: considerând două intervale i_1 şi i_2 putem avea următoarele situaţii:

a. Cele două intervale nu se intersectează deloc, atunci sau $i_1.high < i_2.low$ (adică i_2 e la dreapta lui i_1) sau $i_2.high < i_1.low$ (adică i_1 e la dreapta lui i_2).



b. Se intersectează, atunci $i_1.low \leq i_2.high$ şi $i_2.low \leq i_1.high$



Un arbore de intervale se construieşte respectând etapele generale definite mai sus:

1. Alegerea unei structuri de date de bază: un ARN, în care fiecare nod are ca informaţie un interval $x.int$, iar cheia după care se realizează inserţia/ştergerea în arbore este $x.int.low$, adică ordonearea intervalelor se realizează în funcţie de capătul din stânga. Inserţia şi ştergerea se efectuează ca în orice arbore RN. Căutarea se efectuează prin INERVAL_SEARCH(T,i), care caută în T un nod x pentru care $x.int$ se intersectează cu i sau $T.nil$ dacă nu există nici un astfel de interval.

2. Informaţia suplimentară prin care se îmbogăţeşte structura de date: fiecare nod x conţine un câmp $x.max = max(y.int.high$, y nod din subarborele de rădăcină $x)$.

3. Actualizarea informaţiei suplimentare în urma modificării arborelui prin inserţie/ştergere: s-a enunţat înainte o teoremă prin care se demonstra că, dacă informaţia suplimentară dintr-un nod x depinde doar de cheia lui x, de cheile descendenţilor săi şi de informaţiile suplimentare din descendenţii săi, atunci actualizarea informaţiei nu creşte complexitatea operaţiilor de inserţie/ştergere. Dar în cazul unui arbore de intervale:
$$x.max = \max(x.int.high, x.st.max, x.dr.max)$$
deci, condiţiile se respectă.

4. Operaţia nouă care se dezvoltă pe baza informaţiei suplimentare este INTERVAL_SEARCH(T,i), al cărei algoritm este prezentat în continuare.

```
INTERVAL_SEARCH(T,i)
   x = T.rad
   cat timp x ≠ T.nil şi nu se intersectează cu x.int
       dacă x.st ≠ T.nil si i.low ≤ x.st.max atunci
            x = x.st
       altfel x = x.dr
RETURN x
```

Exemplu: Un exemplu de arbore de intervale este prezentat în fig. 4.

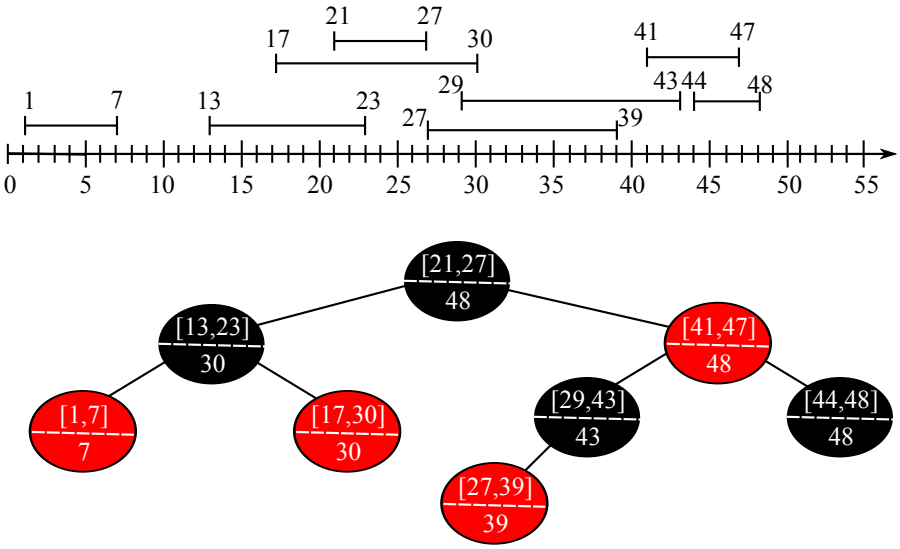


Figure 4: O mulţime de intervale împreună cu arborele RN corespunzător.

.

MODULUL 8 B-ARBORI

B-arborii sunt arbori balansaţi în care toate frunzele au aceeaşi adâncime. Acest tip de arbori reprezintă o generalizare a arborilor binari de căutare. Fiecare nod poate stoca mai multe chei, de la câteva, la mai multe mii. Alături de chei, un nod al arborelui poate conţine şi alte informaţii de interes stocate în structura de date.

O variantă utilizată frecvent a B-arborilor sunt arborii B+, care sunt B-arbori cu proprietatea că informaţiile suplimentare diferite de chei sunt stocate doar în frunze. Nodurile interne conţin doar chei.

1 Elemente de bază

Proprietăţi

1. Fiecare nod x are următoarele atribute (câmpuri):
   (a) $x.n$ numărul de chei stocate în nodul x.
   (b) Vectorul de chei $x.key$
   (c) $x.leaf =$ un câmp boolean care este TRUE, dacă x este frunză şi FALSE altfel.
   (d) Vectorul de legături către fiii lui x: $x.c$
2. Fiecare nod intern x are $x.n + 1$ fii.
3. Valorile cheilor $x.key(i), i = 1, \ldots, x.n$, separă valorile cheilor fiilor lui x după cum urmează: notăm cu $y_i = x.c(i), i = 1, \ldots, x.n + 1$ atunci:

$$y_1.key(1) \leq y_1.key(2) \leq \ldots \leq y_1.key(y_1.n) \leq x.key(1) \leq$$

$$\leq y_2.key(1) \leq y_2.key(2) \leq \ldots \leq y_2.key(y_2.n) \leq x.key(2) \leq$$

$$\ldots$$

$$\leq y_{x.n+1}.key(1) \leq y_{x.n+1}.key(2) \leq \ldots \leq y_{x.n+1}.key(y_{x.n+1}.n)$$

De fapt cheile $x.key(i-1)$ şi $x.key(i)$ definesc intervalul în care se pot afla valorile cheilor din subarborele $x.c(i), i = \overline{2, x.n}$. Cheile din subarborele de rădăcină $x.c(1)$ sunt mai mici decât cheia $x.key(1)$ şi cheile din subarborele de rădăcină $x.c(x.n+1)$ sunt mai mari decât cheia $x.key(x.n)$.

4. Toate frunzele au aceeaşi adâncime = adâncimea/înălţimea h a arborelui.
5. Numărul de chei ale unui nod este limitat inferior şi superior pe baza unei constante t, $t \geq 2$, numită ***gradul minim*** al arborelui, după cum urmează:
   (a) Fiecare nod x cu excepţia rădăcinii, conţine cel puţin $t-1$ chei şi are deci cel puţin t copii. Rădăcina conţine cel puţin o cheie.
   (b) Fiecare nod x conţine cel mult $2t-1$ chei, deci are cel mult $2t$ copii. Un nod care conţine $2t-1$ chei se numeşte ***nod plin***.

Un exemplu de B-arbore cu $t = 2$ în care cheile sunt litere ale alfabetului este prezentat în figura 1. Ordinea considerată este cea lexicografică.

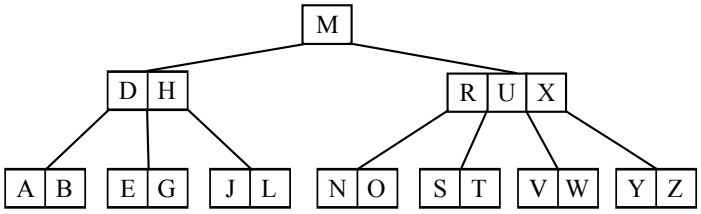


Figure 1: Exemplu de B-arbore

Înălţimea unui B-arbore

Considerăm un B-arbore de grad minim $t \geq 2$ cu n chei. Atunci înălţimea h a arborelui respectă inegalitatea:

$$h \leq \log_t \left(\frac{n+1}{2} \right)$$

Demonstraţie: vom calcula înălţimea unui B-arbore T cu n chei pornind de la un B-arbore T_1 cu aceeaşi înălţime h, dar cu număr minim de chei. Evindent $n \geq$ numărul de chei T_1.
Numărul de chei ale lui T_1, reprezentat în figura 2, se calculează astfel:

Nivel 0 : rădăcina conţine o singură cheie

Nivel 1 : conţine 2 noduri a câte $t-1$ chei

Nivel 2 : conţine $2 * t$ noduri a câte $t-1$ chei

Nivel 3 : conţine $2 * t^2$ noduri a câte $t-1$ chei

.....

Nivel h : conţine $2 * t^{n-1}$ noduri a câte $t-1$ chei

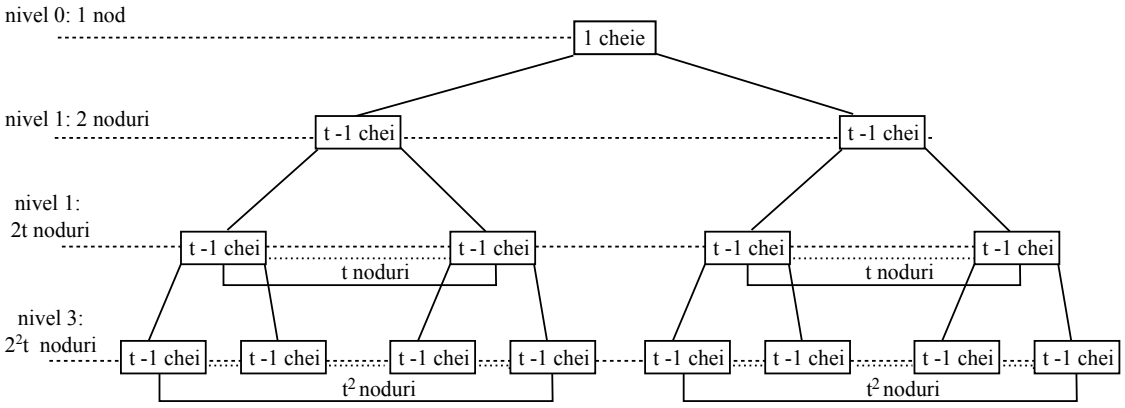


Figure 2: Numărul minim de chei într-un B-arbore de grad minim t şi înălţime h

În total numărul de chei ale arborelui T_1 este:

$$1 + 2(t-1) + 2t(t-1) + 2t^2(t-1) + ... + 2t^{n-1}(t-1) =$$
$$= 1 + 2(t-1)(1 + t + t^2 + ... + t^{n-1}) =$$
$$= 1 + 2(t-1)(t^h - 1)/(t-1) = 2t^h - 1.$$

Deci:

$$n \geq 2t^h - 1 \Rightarrow h \leq \log_t \left(\frac{n+1}{2} \right).$$

Înălţimea unui arbore este cu atât mai mică, cu cât gradul minim t este mai mare. Complexitatea operaţiilor de căutare, inserţie şi ştergere depinde de înălţimea arborelui, deci de t.

Domenii de utilizare

B-arborii pot fi utilizaţi cu succes pentru accesul rapid al datelor de pe hard-disk, prin minimizarea numărului de operaţii 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 înălţimea este mai mică şi accesul la informaţie este mai rapid.

Multe baze de date utilizează B-arbori sau variante ale acestora pentru memorarea datelor.

2 Operaţii

Observaţie: precum s-a specificat anterior, B-arborii sunt utilizaţi în general pentru optimizarea accesului la memorie. Nodul rădăcină este memorat în memoria principală

(în memoria RAM), astfel încât accesul la rădăcină nu necesită operaţii de citire pe disc. Accesul la noduri interne necesită citire pe disc.

Notăm prin: DR, DW operaţiile de citire/scriere pe disc.

2.1 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.

Funcţia BT_CAUT(x, k) este definită recursiv şi are ca parametri rădăcina x a subarborelui în care s-a ajuns cu căutarea şi cheia căutată, k. Funcţia returnează o pereche (x, i), unde x = nodul care conţine cheia k şi i indicele cheii în nodul x, adică $x.key(i) = k$. Dacă în arbore nu se găseşte cheia k, atunci funcţia returnează NULL.

Algoritm:

```
BT_CAUT(x,k)
  i=1
  cat timp i≤ x.n si x.key(i) < k
           i=i+1
  sfarsit cat timp
  daca i≤ x.n si x.key(i) = k atunci
           RETURN(x,i)
  altfel
           daca x.leaf = TRUE atunci
                RETURN NULL
           altfel
                DR(x.c(i)) //citeste de pe disc un nou nod
                RETURN BT_CAUT(x.c(i),k)
           sfarsit daca
  sfarsit daca
RETURN
```

Complexitate: căutarea în fiecare nod este de ordinul $O(t)$, iar căutarea în arbore depinde de înălţimea h a arborelui, care este de ordinul $\log_t n$. Astfel complexitatea totală a algoritmului este $O(t \log_t n)$.

2.2 Crearea unui arbore vid

Crearea unui arbore vid, în care apoi se inserează pe rând chei, presupune întâi alocarea unei zone de memorie pe disc, rezervată rădăcinii. Această procedură se realizează în timp $O(1)$.

```
BT_CREEAZA(T)
  aloca memeorie pentru x
  x.leaf=TRUE
  x.n=0
  DW(x) //scrie pe disc un nou nod
  T.rad=x
RETURN
```

2.3 Inserarea unei chei

Inserarea unei chei într-un B-arbore diferă de inserarea într-un arbore binar prin faptul că nu putem pur şi simplu crea o nouă frunză în B-arbore în care să inserăm cheia respectivă. Un astfel de procedeu ar duce la dezechilibrarea arborelui şi ar contraveni proprietăţii unui B-arbore de a avea toate frunzele pe acelaşi nivel.

Inserţia 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 poziţia din mijloc în sirul cheilor).

Operaţia de divizare a unui nod

Se consideră un nod plin y, deci cu $2t-1$ chei. Presupunem că y este al i-lea descendent al nodului x, care nu este plin. Atunci divizarea se realizează în modul următor:

- $x.n$ creşte cu o unitate: $x.n = x.n + 1$.
- cheile lui x începând de la poziţia i se deplasează cu o poziţie la dreapta în nod.
- se deplasează pointerii către descendenţii $x.c(i+1), \ldots, x.c(x.n)$ cu o poziţie spre dreapta în nodul x.
- cheia $y.key(t)$ urcă în nodul x pe poziţia i
- nodul y se divizează în două noduri noi care conţin câte $t-1$ chei şi anume primul nod conţine cheile $y.key(1), \ldots, y.key(t-1)$ şi al doilea nod conţine cheile $y.key(t+1), \ldots, y.key(2t-1)$
- noile noduri create devin descendenţii $x.c(i)$ şi $x.c(i+1)$.

Această procedură este ilustrată în figura 3.

Observaţii:

1. Dacă x este la rândul său plin, nu putem efectua divizarea. De aceea la inserţie trebuie să ne asigurăm deja pe parcursul căutării frunzei în care se realizează inserţia că pe drum nu întâlnim noduri pline. În caz contrar se realizează o divizare.

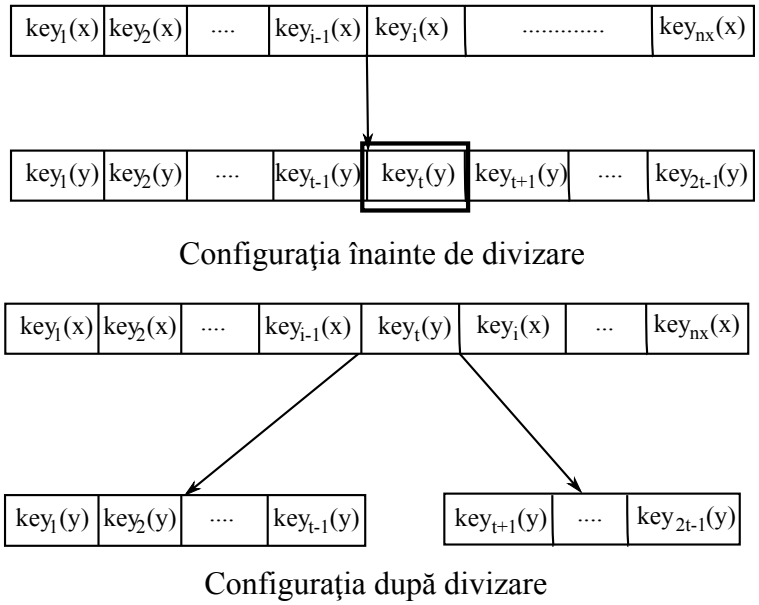


Figure 3: Divizarea unui nod plin

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 trebuie procedat în modul următor:

   Divizarea rădăcinii

   - Se creează un nou nod vid, care va fi noua rădăcină.
   - Se leagă vechea rădăcină ca descendent al noii rădăcini.
   - Se realizează operaţia de divizare.

3. Înălţimea unui B-arbore creşte doar prin divizarea rădăcinii.

Exemplu: Inserarea cheii P în arborele din figura 4 cu $t = 2$. Căutarea începe de la

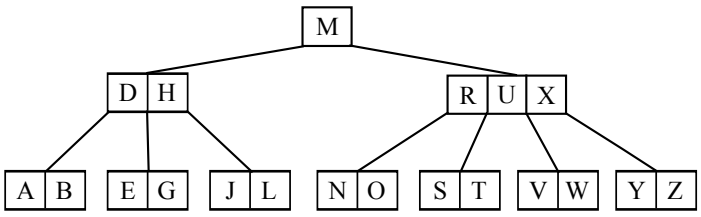


Figure 4: B-arbore în care se va realiza inserţia de chei

nodul rădăcină. $T.rad.n = 1$, iar $T.rad.key(1) < P$, deci trebuie coborât la descendentul $T.rad.c(2)$. Dar Se observă că $T.rad.c(2).n = 2t - 1 = 3$, deci nodul este plin. Înainte de a continua, nodul $T.rad.c(2)$ trebuie divizat. Se obţine arborele din fig 5:

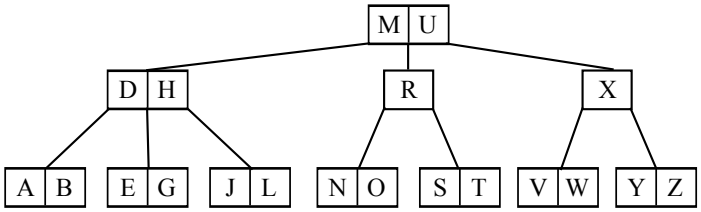


Figure 5: B-arborele după divizarea nodului cu cheile R, U, X

Acum $T.rad.key(1) = M < P$ şi $T.rad.key(2) = U > P$, deci se coboară la descendentul $x_1 = T.rad.c(2)$.

Avem: $x_1.n = 1, x_1.key(1) = R > P$, deci se coboară la nodul $x_2 = x_1.c(1)$, care este frunză. $x_2.n = 2 < 2t-1$, deci nodul nu este plin şi se poate insera P pe poziţia potrivită. Rezultă arborele din fig. 6.

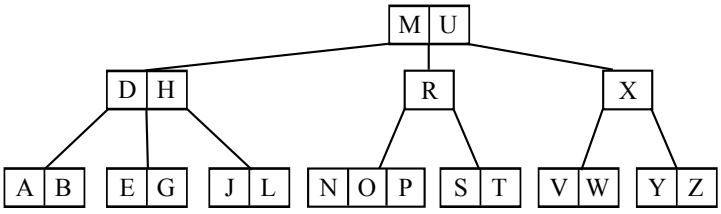


Figure 6: B-arborele rezultat după inserţia cheii P

2.4 Eliminarea unei chei

Eliminarea unei chei dintr-un B-arbore este ceva mai delicată decât inserţia, deoarece uneori trebuie eliminată şi o cheie dintr-un nod intern. Acest lucru ar avea însă ca urmare reducerea numărului de chei din nod şi ca urmare a numărului de descendenţi ai nodului respectiv. Este deci necesară o rearanjare a cheilor şi a descendenţilor în arbore. În plus, la eliminarea unei chei trebuie avut grijă ca numărul de chei dintr-un nod sa nu devină mai mic decât $t-1$. Astfel nu se poate pur şi simplu extrage o cheie dintr-un nod cu $t-1$ chei.

Aceste probleme se rezolvă prin operaţii de rotaţie şi prin fuziuni.

Rotaţia într-un B-arbore

Se consideră nodul x.

- O rotaţie la dreapta în jurul cheii $k = x.key(i)$:
  - mută ultima cheie a fiului $y_i = x.c(i)$ în nodul x în locul cheii k
  - mută cheia k pe prima poziţie în nodul $y_{i+1} = x.c(i+1)$

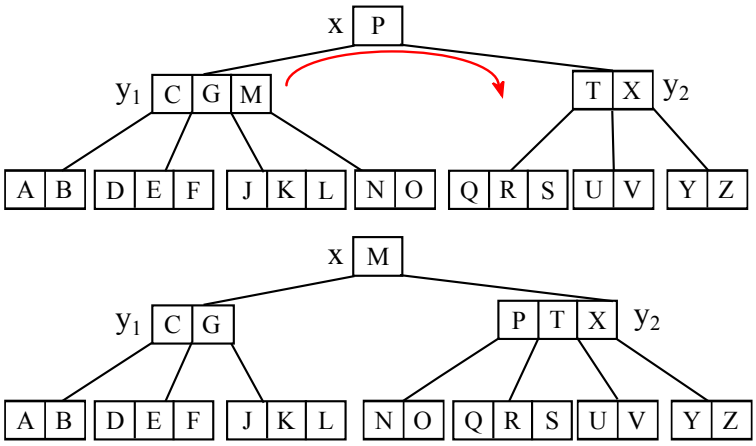


Figure 7: Rotaţie la dreapta în jurul cheii P din rădăcină.

  - mută ultimul fiu al lui y_i ca prim fiu al lui y_{i+1}.

- O rotaţie la stânga în jurul cheii $k = x.key(i)$:
  - mută prima cheie a fiului $y_{i+1} = x.c(i+1)$ în nodul x în locul cheii k
  - mută cheia k pe ultima poziţie din nodul $y_i = x.c(i)$
  - mută primul fiu al lui y_{i+1} ca ultim fiu al lui y_i.

Un exemplu de rotaţie la dreapta într-un B-arbore este prezentat în figura 7.

Fuziunea a două noduri vecine într-un B-arbore

O fuziune nu poate fi realizată decât între doi fraţi vecini, fii ai aceluiaşi nod x, şi care au ambii exact $t-1$ chei. De asemenea părintele x trebuie să aibă cel puţin t chei. Se consideră nodul x, iar descendenţii care fuzionează sunt $y = x.c(i)$ şi $z = x.c(i+1)$. Fuziunea lui y cu z presupune reunirea celor două noduri într-unul singur, iar cheia $x.key(i)$ coboară din nodul x în noul nod rezultat pe poziţia aflată între cheile lui y şi cele ale lui z.

Această operaţie este reprezentată grafic în fig. 8.

Un exemplu de fuziune a descendenţilor rădăcinii din fig. 9 este prezentat în **??**.

Observaţii:

- Înălţimea unui arbore se micşorează atunci când rădăcina are doi fii care fuzionează, ca în exemplul din fig. **??**.
- O fuziune se poate realiza doar dacă părintele nodurilor care fuzionează are cel puţin t chei. Din acest motiv, aşa cum la inserţie pe parcursul căutării nodului în care se inserează, toate nodurile pline întâlnite pe parcurs trebuie divizate, în cazul eliminării unei chei, pe parcursul căutării cheii respective, înainte de a coborî la un

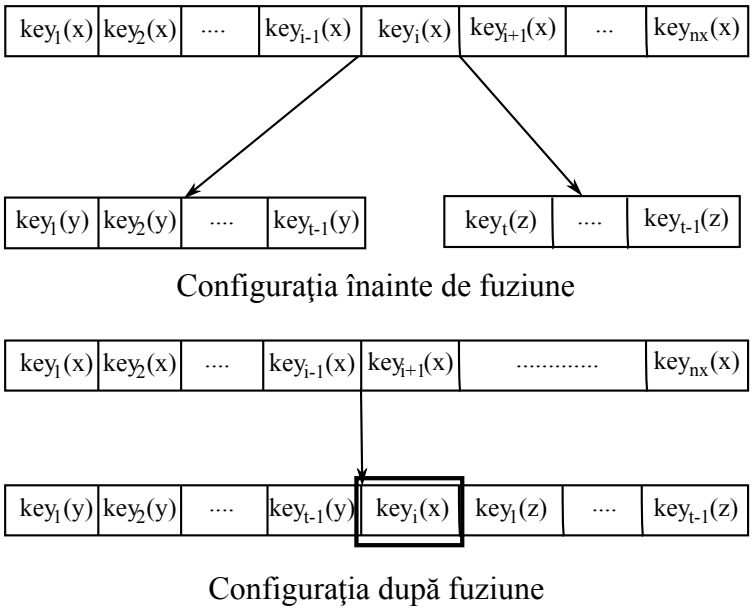


Figure 8: Fuziunea a două noduri vecine.

descendent, trebuie asigurat acestui descendent un minim de t chei.

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 porneşte de la rădăcină şi se coboară în arbore până la întâlnirea nodului ce conţine cheia k.

Notăm cu x nodul curent. Pe parcurs ne asigurăm că orice nod în care se coboară are cel puţin $t-1$ chei.

Cazul I: x este frunză şi k este cheie a lui x. Din faptul că s-a avut grijă ca nodurile în care se coboară să aibă cel puţin t chei, x are cel puţin t chei, deci pur şi simplu se extrage cheia k din nodul x şi se încheie algoritmul.

Cazul II: x nu este frunză şi $k = x.key(i)$. Atunci:

(a) **Dacă** $y = x.c(i)$ (copilul care îl precede pe k) are $y.n \geq t$ atunci se caută k' predecesorul lui k în subarborele de rădăcină y, se şterge recursiv k' din subarborele de rădăcină y şi se înlocuieşte k cu k' în x.

(b) **Altfel dacă** $y = x.c(i+1)$ (copilul care îi urmează lui k) are $y.n \geq t$ atunci se caută k'' succesorul lui k în subarborele de rădăcină y, se şterge recursiv k'' din acest subarbore şi se înlocuieşte k cu k'' în x.

(c) **Altfel** - adică ambii copii aflaţi de o parte şi de alta a lui k au exact $t-1$ chei - atunci: se realizează o fuziune între cei doi copii $x.c(i)$ şi $x.c(i+1)$. Astfel se obţine

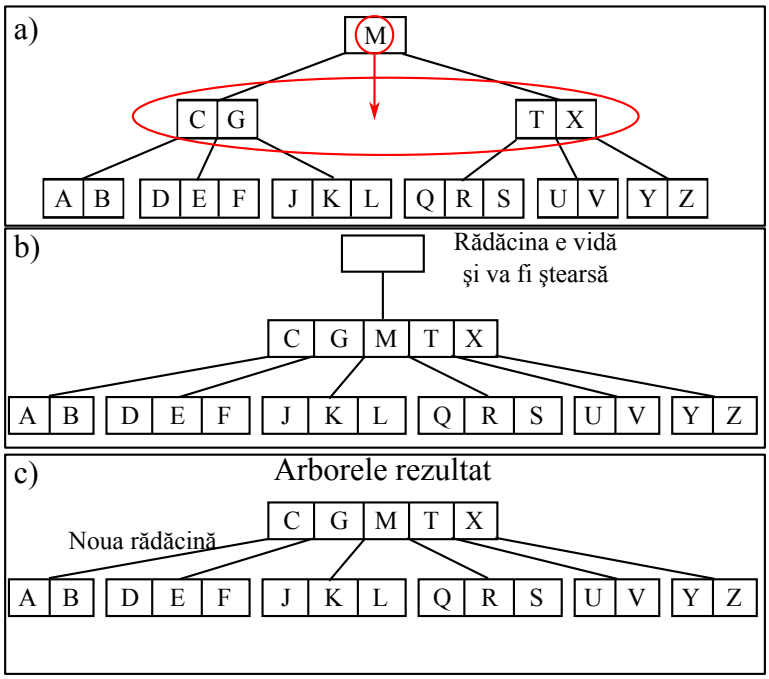


Figure 9: a) Nodurile care urmează să fuzioneze împreună cu cheia M a rădăcinii; b) Arborele după fuzionare; c) Rezultatul după eliminarea rădăcinii vide.

un nou nod, care va conţine cheia k. Aceasta se şterge recursiv din noul nod obţinut prin fuziune.

Cazul III: x nu este fruză şi k nu este cheie a lui x.
Atunci se determină i astfel încât $x.key(i-1) < k < x.key(i)$. Rezultă faptul că trebuie căutată cheia k în subarborele de rădăcină $x.c(i)$. Înainte de a coborî la descendentul $x.c(i)$ al lui x trebuie să ne asigurăm că acest descendent are cel puţin t chei. Dacă $x.c(i)$ are doar $t-1$ chei atunci:

(a) Dacă $x.c(i)$ are un frate vecin cu cel puţin t chei se realizează o rotaţie astfel încât o cheie din acel frate să urce la părintele x şi o cheie din x să coboare la nodul $x.c(i)$. Apoi se şterge recursiv k din $x.c(i)$.

(b) Dacă ambii fraţi vecini au $t-1$ chei trebuie realizată fuziunea lui $x.c(i)$ cu unul dintre aceşti fraţi. Apoi se şterge recursiv k din nodul rezultat prin fuziune.

Exemplu: Considerăm arborele din fig. 10 cu $t = 3$.

Se şterge cheia G: se porneşte de la rădăcină $G < M$ şi 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) $\Rightarrow$ rotaţie la stânga în jurul cheii M din rădăcină. Se obţine arborele din fig. 11 a).

Acum se coboară la nodul x care conţine cheile C, G, M. Se observă că acesta nu e frunză, dar conţine cheia $G = x.key(2)$ şi ambii fii $x.c(2)$ şi $x.c(3)$ au t chei, deci pot aplica cazul

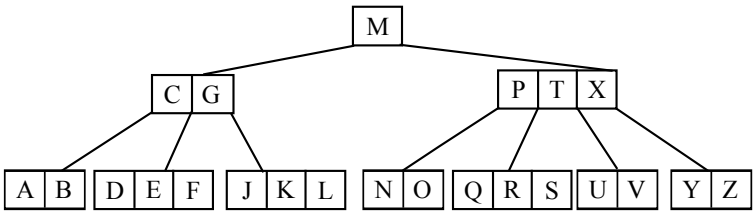


Figure 10: B-arbore cu $t = 3$ în care vor fi efectuate ştergeri de chei

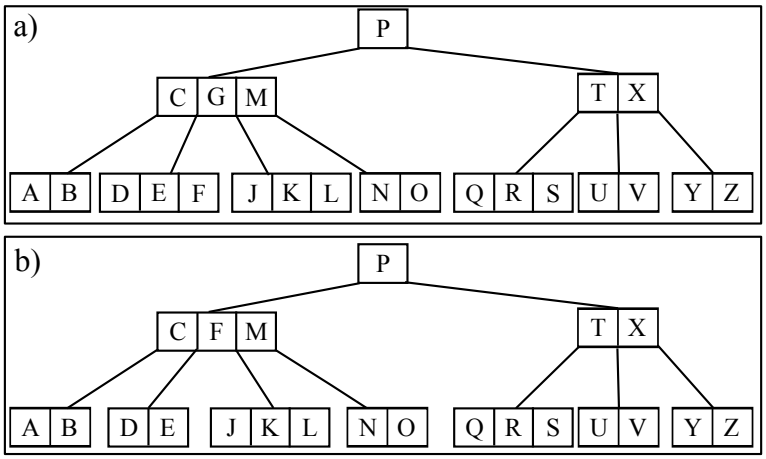


Figure 11: a) B-arborele din fig. 10 după rotaţie; b) Arborele după ştergerea cheii G.

II (a) $\Rightarrow$ caut predecesorul lui G, care este F. Înlocuiesc G cu F şi şterg F din fiul $x.c(2)$. Se obţine arborele din fig. 11 b).

Se şterge cheia B: se porneşte cu $x = T.rad$, se compară $B < x.key(1) \Rightarrow$ se studiază fiul $x.c(1)$. Acesta are t chei $\Rightarrow x = x.c(1)$.

Se observă că în acest moment $B < x.key(1) = C \Rightarrow$ se studiază fiul $x.c(1)$. 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 $x.c(1)$ şi $x.c(2)$. Se obţine arborele din fig. 12.

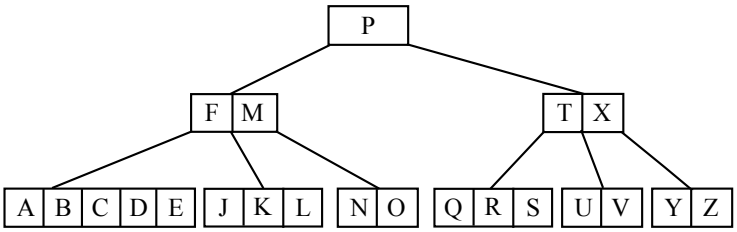


Figure 12: B-arborele după fuziunea nodurilor cu cheile A, B şi respectiv D, E împreună cu cheia C.

Se coboară acum la $x = x.c(1)$ şi se şterge cheia B din nod. Rezultă arborele din fig. 13.

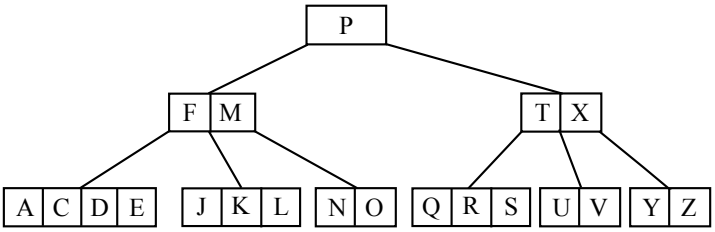


Figure 13: B-arborele după şergerea cheii B.

Se şterge cheia X: se porneşte de la rădăcina $x = T.rad$. $X > x.key(1) \Rightarrow$ se studiază fiul $x.c(2)$. Acesta are doar $t-1$ chei. Unicul frate al său are tot $t-1$ chei $\Rightarrow$ fuziune. Rădăcina iniţială rămâne fără nici o cheie, deci se şterge şi noua rădăcină va fi nodul rezultat în urma fuziunii. Rezultă arborele din figura 14.

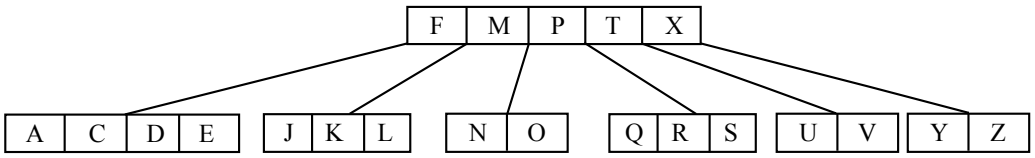


Figure 14: B-arborele după fuziunea fiilor rădăcinii şi ştergerea acesteia.

Cheia X se găseşte în rădăcină şi ambii fii, de o parte şi de alta a lui X au exact $t-1$ chei $\Rightarrow$ fuziune. Rezultă arborele din fig. 15.

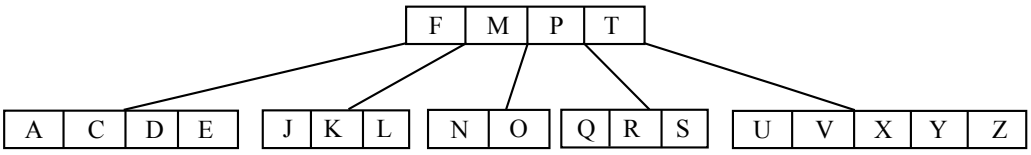


Figure 15: B-arborele după fuziunea nodurilor U, V şi Y, Z.

Acum se poate şterge X din frunza care îl conţine şi se obţine arborele din fig. 16.

Complexitate: Operaţiile într-un B-arbore presupun căutare în interiorul unui nod, care este $O(t)$ precum şi coborâre în arbore, care depinde de înălţimea 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 operaţii la $O(\log_2 t)$.

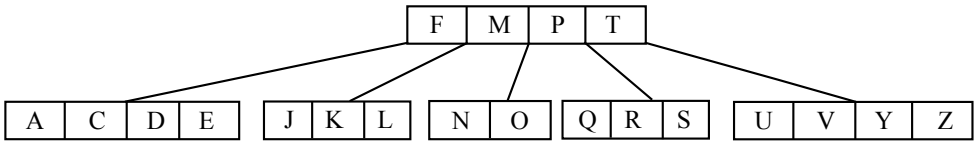


Figure 16: B-arborele după ştergerea cheii X.

MODULUL 9 - TABELE DE REPARTIZARE

Aşa cum s-a menţionat în primul capitol al acestui curs, structurile de date au fost dezvoltate cu scopul memorării şi manipulării ***eficiente*** a unei mulţimi dinamice de date. Acesastă manipulare eficientă depinde desigur semnificativ de scopul urmărit şi de tipurile de operaţii care se doresc a fi efectuate.

Una dintre principalele operaţii pe un set de date este căutarea sau *accesul prin cheie*. Am văzut în capitolele precedente că problema căutării poate fi rezolvată în complexitate logaritmică prin utilizarea arborilor de căutare echilibraţi.

Există însă şi structuri de date care permit accesul prin cheie în timp constant, atunci când nu sunt necesare operaţii de sortare şi păstrare ordonată a setului de date. Aceste structuri sunt *tabelele de repartizare*.

O tabelă de repartizare - *hash table* - este de fapt o generalizare a noţiunii de vector - *array*. Operaţiile de bază sunt inserţia, căutarea şi eventual ştergerea. Tabelele de repartizare pot fi utilizate cu succes pentru implementarea de dicţionare, în care căutarea este cea mai frecventă operaţie. O altă utilizare a tabelelor de repartizare este în cadrul compilatoarelor pentru implementarea tabelei de identificatori sau pentru cuvintele cheie. Vom vedea pe parcurs că, într-o tabelă de repartizate operaţiile au în medie complexitatea $O(1)$. O tabelă de repartizare se memorează într-un tablou liniar.

1 Tabele cu adresare directă

Considerăm $U = \{0, 1, \ldots, m-1\}$ universul cheilor posibile, m relativ mic şi $T[0, \ldots, m-1]$ tabloul în care se memorează elementele din tabelă. Elementul cu cheia k se plasează în T pe poziţia $T[k]$. Dacă în tabelă nu există cheia k, atunci $T[k] = NULL$.

Observaţie: o astfel de tabelă de adresare directă este potrivită, atunci când universul U al cheilor este relativ redus şi numărul de elemente memorate în tabelă este comparabil cu m.

2 Tabele de repartizare - *Hash Tables*

Pentru situaţii în care numărul de elemente stocate este mult mai mic decât universul cheilor se recomandă înlocuirea tabelei cu adresare directă printr-o tabelă de repartizare.

În cazul unei tabele de repartizare $T[0,\ldots,m-1]$ accesul la un element se face prin intermediul unei funcţii de repartizare - *hash function* - $h : U \to \{0,1,\ldots,m-1\}$, în care de obicei $|U|$ este semnificativ mai mare decât m.

Elementul cu cheia k va fi plasat în tabelă pe poziţia $T[h(k)]$. Spunem că, elementul cu cheia k este repartizat pe poziţia $h(k)$ (*hashes to slot* $h(k)$).

Exemplu de funcţie de repartizare:

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

Considerând $m = 11$ şi cheile $\{3,12,15,17\}$ atunci: $h(3) = 3$, $h(12) = 1$, $h(15) = 4$, $h(17) = 6$ (fig. 2).

*	12	*	3	15	*	17	*	*	*	*
0	1	2	3	4	5	6	7	8	9	10

Figure 1: Tabela de repartizare cu funcţia de repartizare $h(k) = k \bmod 11$ în care au fost plasate cheile 3, 12, 15 şi 17. Au fost marcate cu * poziţiile neocupate din tabelă.

Coliziunea

În cazul unei funcţii de repartizare h este posibil ca pentru două chei diferite k_1 şi k_2 să se obţină $h(k_1) = h(k_2)$, adică ambele elemente ar fi repartizate pe aceeaşi poziţie. O astfel de situaţie se numeşte *coliziune*. În exemplul de mai sus $h(24) = h(35) = 2$.

Se pune problema rezolvării coliziunilor. Ideal ar fi, evitarea completă a acestora. Din faptul că se presupune că $|U| > m$, prin tipul de repartizare descris mai sus, acest lucru nu este posibil. Totuşi, prin construirea atentă a funcţiei de repartizare, poate fi redusă semnificativ probabilitatea unei coliziuni. Vor fi discutate pe parcursul acestui curs câteva metode de construcţie a unei funcţii de repartizare, care să îndeplinească acest lucru.

Rezolvarea problemei coliziunilor se poate realiza dacă în loc de a stoca într-o poziţie a tabelei de repartizare un singur element, se păstrează o listă de elemente (*bucket*). Lista de la poziţia $h(k)$ va conţine toate elementele care au această valoarea de repartizare.

Exemplu: Considerăm $h(k) = k \bmod 11$ şi cheile $\{23,34,78,15,37,42,45\}$. $h(23) = 1$, $h(34) = 1$, $h(78) = 1$, $h(15) = 4$, $h(37) = 4$, $h(42) = 9$, $h(45) = 1$

Operaţiile uzuale:

```
TAB_INSERT(T,x)
   Insereaza x in capul listei T[h(x.cheie)]
RETURN
```

```
TAB_CAUT(T,k)
   Cauta elementul cu cheia k in lista T[h(k)]
RETURN

TAB_STERG(T,x)
   sterge x din lista T[h(x.cheie)]
RETURN
```

O tabelă de repartizare cu dimensiunea $m = 11$ care utilizează liste înlănţuite şi în care au fost inserate cheile 33, 35, 55, 46, 12, 2, 7, 10, 11 este reprezentată în figura 2.

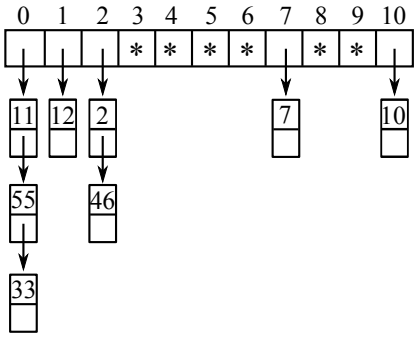


Figure 2: Tabela de repartizare cu funcţia de repartizare $h(k) = k \bmod 11$ şi utilizând liste înlănţuite în care au fost plasate succesiv cheile 33, 35, 55, 46, 12, 2, 7, 10 şi 11. Au fost marcate cu * poziţiile neocupate din tabelă.

Complexitate:

- funcţia de inserţie are evident complexitatea $O(1)$,
- la funcţia de căutare complexitatea depinde de lungimea listelor înlănţuite
- funcţia de ştergere are complexitatea $O(1)$ dacă lista e dublu înlănţuită, altfel depinde de lungimea listei.

Analiza complexităţii

Considerăm tabela de repartizare T cu m poziţii, în care se stochează n elemente, folosind liste înlănţuite pentru rezolvarea coliziunilor. Atunci numărul mediu de elemente stocate într-o listă este $\alpha = n/m$, unde α poate fi mai mic sau mai mare decât 1.

Cea mai defavorabilă situaţie este aceea când toate cele n elemente se repartizează prin h pe aceeaşi poziţie, adică sunt stocate în aceeaşi listă. În această situaţie complexitatea la căutare este $O(n)$. Pentru a evita astfel de situaţii este necesară alegerea unei funcţii de repartizare potrivită. Modul de construcţie a unor funcţii de repartizare care să permită o repartizare cât mai uniformă va fi discutată ulterior.

Repartizarea uniformă presupune ca probabilitatea ca un anumit element să fie de repartizat pe oricare dintre poziţiile din tabelă este aceeaşi.

Considerând o funcţie de repartizare care repartizează cheile uniform în tabela T şi pentru care calculul lui $h(k)$ este $O(1)$, se demonstrează faptul că funcţia de căutare e unei chei este $O(1+\alpha)$ (vezi bibliografia recomandată).

Dacă n este proproţional cu m, atunci $n = am$, deci complexitatea la căutare este $O(1 + am/m) = O(1 + a) = O(1)$.

3 Metode de repartizare

Din discuţia complexităţii rezultă că, o funcţie de repartizare bună permite o repartizare aproape uniformă în tabelă. Pentru acest lucru poziţia obţinută la repartizare ar trebui să fie independentă de orice **pattern** care ar putea fi prezent în setul de date.

În plus, funcţiile de repartizare au ca mulţime a valorilor poziţii în tabela de repartizare, deci elemente din mulţimea numerelor naturale. În cazul în care cheile sunt de exemplu şiruri de caractere - de exemplu **hash tables** utilizate pentru tabela de simboluri din faza de analiză lexicală a unui compilator - este necesară stabilirea unei metode de interpretare a acestora ca numere naturale, trebuie deci determinată o funcţie de la universul cheilor către mulţimea numerelor naturale.

În continuare sunt prezentate câteva metode pentru construcţia funcţiilor de repartizare.

3.1 Metoda diviziunii

Funcţia de repartizare $h : U \to \{0, 1, \ldots, m-1\}$, $h(k) = k \bmod m$

Exemplu: $h(k) = k \bmod 11$.

Observaţie: o alegere bună pentru m este un număr prim nu prea apropiat de o putere a lui 2. În plus acest număr trebuie ales depinzând de numărul de elemente care se estimează că vor fi introduse în tabelă, precum şi de factorul de încărcare dorit.

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

3.2 Metoda multiplicării

În acest caz funcţia de repartizare este de tipul:

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

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

În cazul acestor funcţii modul de alegere a lui m nu influenţează modul de repartizare. Se demonstrează (Knuth) faptul că, o alegere bună pentru A este

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

3.3 Repartizarea universală

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

Acest lucru poate fi evitat, dacă în loc să se folosească o singură funcţie de repartizare, se utilizează o mulţime H de funcţii de repartizare. La fiecare execuţie se selectează în mod aleator din H una dintre funcţiie de repartizare, care va fi apoi utilizată.

Utilizare repartizării universale are ca efect faptul că, pentru acelaşi set de date de intrare, execuţii diferite ale programului produc în general tabele diferite, astfel putându-se evita în orice situaţie cel mai defavorabil caz, eventual prin reluare a repartizării - **rehashing**.

Definiţie: O mulţime finită de funcţii de repartizare H, care repartizează cheile dintr-un univers U într-o tabelă $T[0, \ldots, m-1]$ se numeşte universală, dacă pentru orice două chei k şi l, $k \neq l$, numărul de funcţii din H pentru care $h(k) = h(l)$ este cel mult $|H|/m$.

Acest lucru asigură faptul că, pentru o funcţie aleasă în mod aleator din H, probabilitatea unei coliziuni între k şi l este $1/m$. (Evident, probabilitatea alegerii oricărei funcţii din H este $1/|H|$. Probabilitatea alegerii unei funcţii care repartizează k şi l pe aceeaşi poziţie este($|H|/m) * 1/|H| = 1/m$).

Construcţie a unei clase universale de funcţii de repartizare

Se alege un număr prim p relativ mare, $p >> m$ Pentru fiecare $a \in \{1, \ldots, p-1\}$ şi $b \in \{0, 1, \ldots, p-1\}$ se defineşte funcţia de repartizare:

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

Mulţimea de astfel de funcţii de repartizare este

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

Se poate demonstra că mulţimea astfel definită este universală (vezi bibliografia recomandată).

3.4 Adresare deschisă

În cazul adresării deschise, fiecare poziţie a tabelei de repartizare T conţine un singur element. Pentru rezolvarea coliziunilor la inserţie nu se utilizează liste înlănţuite, ci se testează diferite poziţii, obţinute pe baza funcţiei de repartizare, până când se determină o poziţie liberă, pe care poate fi inserat elementul dorit.

La căutare de asemenea se testează diferite poziţii până când găseşte elementul căutat sau se ajunge la o poziţie neocupată.

Faptul că nu se utilizează liste înlănţuite, ci fiecare poziţie conţine cel mult un element, are ca urmare posibilitatea umplerii tabelei. În schimb se evită utilizarea pointerilor, iar memoria, care altfel ar fi fost utilizată pentru listele înlănţuite, poate fi utilizată pentru o tabelă de dimensiune mai mare.

Modul de testare a poziţiilor în tabelă în cazul adresării deschise nu este liniar ci depinde de cheia inserată şi de numărul de testări efectuate până la momentul curent.

O funcţie de repartizare pentru repartizare deschisă este de forma:

$$h : U \to \{0, 1, \ldots, m-1\} \times \{0, 1, \ldots, m-1\}$$

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

Inserţia unei chei într-o tabelă $T[0, \ldots, m-1]$

```
TAD_INSERT(T,k)
    i = 0
    repeta
        j = h(k, i)
        daca T[j] = NULL atunci
            T[j] = k
```

```
            RETURN j
        sfarsit daca
        i = i + 1
    pana cand i = m
    scrie("tabela este plina")
RETURN -1
```

Algoritmul de căutare testează pentru o anumită cheie k aceeaşi secvenţă de poziţii ca şi algoritmul de inserţie al cheii k.

```
TAD_CAUt(T,k)
    i = 0
    repeta
        j = h(k,i)
        daca T[j] = k atunci
            RETURN j
        i = i + 1
    pana cand i = m sau T[j] = NULL
RETURN -1
```

Observaţie: Operaţia de ştergere este problematică în cadrul adresării deschise. Dacă o cheie k este ştearsă prin marcarea poziţiei cu $NULL$ atunci o cheie p, inserată după k, dar care la inserţie/ştergere presupune testarea poziţiei pe care s-a aflat k, nu va mai fi găsită prin algoritmul de mai sus $\Rightarrow$ este nevoie de un algoritm de complexitate mai mare la căutare.

O soluţie a acestei probleme este marcarea poziţiilor şterse cu un marcaj special de poziţie ştearsă, dar tot presupune complexitate crescută pentru număr mare de poziţii şterse.

Atunci când este nevoie de operaţia de ştergere, se recomandă folosirea de tabele de repartizare cu liste înlănţuite.

În ceea priveşte alegerea unei funcţii de repartizare pentru repartizarea cu adresare deschisă, aceasta poate fi realizată în diferite moduri. Vor fi descrise mai jos trei metode de construcţie a unei astfel de funcţii.

(1) Funcţii de repartizare cu testare liniară

Se consideră o funcţie de repartizare obişnuită: $h_1 : U \to \{0, 1, \ldots, m-1\}$. Se defineşte: $h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$

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

Exemplu: considerăm $m = 11$, $h1(k) = k \bmod 11$ şi

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

Atunci cheile 22, 33, 45, 59, 67, 13, 71 vor fi repartizate după cum urmează:
$h(22,0) = (0+0) \bmod 11 = 0$
$h(33,0) = (0+0) \bmod 11 = 0$, este deja ocupat, deci testarea continuă
$h(33,1) = (0+1) \bmod 11 = 1$
$h(45,0) = (1+0) \bmod 11 = 1$, este deja ocupat, deci testarea continuă
$h(45,1) = (1+1) \bmod 11 = 2$
$h(59,0) = (4+0) \bmod 11 = 4$
$h(67,0) = (1+0) \bmod 11 = 1$, este deja ocupat, deci testarea continuă
$h(67,1) = (1+1) \bmod 11 = 2$, este deja ocupat, deci testarea continuă
$h(67,2) = (1+2) \bmod 11 = 3$,
$h(13,0) = (2+0) \bmod 11 = 2$, este deja ocupat, deci testarea continuă
$h(13,1) = (2+1) \bmod 11 = 3$, este deja ocupat, deci testarea continuă
$h(13,2) = (2+2) \bmod 11 = 4$
$h(71,0) = (5+0) \bmod 11 = 5$

Observaţie: Dezavantajul testării liniare este acela că, pe măsură ce se adaugă elemente, cresc secvenţele de poziţii succesive ocupate - **primary clustering**, ceea ce duce la creşterea complexităţii de inserţie/căutare.

Un mod de îmbunătăţire a acestei situaţii este utilizarea funcţiilor de repartizare cu testare pătratică descrise în continuare.

(2) Funcţiile de repartizare cu testare pătratică

Se consideră o funcţie de repartizare obişnuită: $h_1 : U \to \{0, 1, \ldots, m-1\}$. Se defineşte: $h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$

$$h(k, i) = (h_1(k) + c_1 i + c_2 i^2) \bmod m$$

h_1, funcţie de repartizare obişnuită, c_1 şi c_2 constante întregi pozitive auxiliare.

Prin acest mod de repartizare, se îmbunătăţeşte uşor performanţa. Problema este că, pentru două elemente k_1 şi k_2 distincte, pentru care $h(k_1, 0) = h(k_2, 0)$, oricare ar fi i, $h(k_1, i) = h(k_2, i)$. Acest lucru are ca urmare faptul că, dacă se inserează numeroase elemente care au aceeaşi repartizare iniţială, creşte complexitatea inserţiei. Rezultă o formă oarecum atenuată de *clustering*, numită *secondary clustering* .

Una dintre cele mai bune metode de repartizare deschisă se realizează cu ajutorul **dublei repartizări**.

(3) Funcţii de repartiţie cu dublă repartizare

Se consideră două funcţii de repartizare obişnuite distincte $h_1(k)$ şi $h_2(k)$. Se construieşte funcţia cu dublă repartizare:

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

Observaţie: Se recomandă ca funcţia $h_2(k)$ să producă valori care sunt prime faţă de m. Acest lucru se poate obţine dacă

- m putere a lui 2 şi $h_2(k)$ produce întotdeauna un număr impar
- m prim şi $h_2(k)$ produce numere naturale din $\{0, 1, \ldots, m-1\}$, de exemplu considerăm funcţiile de repartiţie

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

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

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

Exemplu: $h_1(k) = k \bmod 11$, $h_2(k) = 1 + (k \bmod 10)$. Atunci cheile 22, 33, 45, 59, 67, 13, 71 vor fi repartizate după cum urmează:
$h(22, 0) = (22 \bmod 11) \bmod 11 = 0$
$h(33, 0) = (33 \bmod 11) \bmod 11 = 0$, este deja ocupat, deci continuă testarea
$h(33, 1) = ((33 \bmod 11) + (1 + 33 \bmod 10)) \bmod 11 = (0 + 4) \bmod 11 = 4$
$h(45, 0) = (45 \bmod 11) \bmod 11 = 1$
$h(59, 0) = (59 \bmod 11) \bmod 11 = 4$, este deja ocupat, deci continuă testarea
$h(59, 1) = (59 \bmod 11 + 1 + 59 \bmod 10) \bmod 11 = (4 + 10) \bmod 11 = 3$ $h(67, 0) = (67 \bmod 11) \bmod 11 = 1$, este deja ocupat, deci continuă testarea
$h(67, 1) = ((67 \bmod 11) + (1 + 67 \bmod 10)) \bmod 11 = 9$
$h(13, 0) = (13 \bmod 11) \bmod 11 = 2$
$h(71, 0) = (71 \bmod 11) \bmod 11 = 6$

Se observă că sunt necesare mai puţine încercări decât în cazul repartizării cu testare liniară.

Complexitate: considerând factorul de încărcare al tabelei de repartizare $\alpha = n/m < 1$, unde $n =$ numărul de poziţii ocupate şi $m =$ dimensiunea tabelei atunci:

- numărul de testări în cazul unei căutări fără succes este în cel mai defavorabil caz $1/(1-\alpha)$ (presupunând repartizare uniformă) $\Rightarrow$ căutarea fără succes are complexitate $O(1)$ pentru α constant;
- numărul de testări în cazul unei căutări cu succes este în cel mai defavorabil caz $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ (presupunând repartizare uniformă) $\Rightarrow$ căutarea cu succes are complexitate $O(1)$ pentru α constant

Aplicaţii:

- Compilatoare - pentru păstrarea tabelei de identificatori se recomandă utilizarea unei tabele de repartizare, deoarece este necesar acces rapid la informaţie.
- Criptografie - pentru securitatea parolelor se utilizează *hashing* criptografic. Acest tip de repartizare este mai dificil de realizat şi presupune îndeplinirea anumitor criterii de securitate, care depăşesc însă limitele acestui curs.

4 Repartizarea cheilor de tip şir de caractere

O funcţie de repartizare ideală va produce pentru orice două chei inserate poziţii de inserţie diferite. Există situaţii, atunci când sunt cunoscute toate cheile posibile (de exemplu în cazul unui dicţionar) să fie construite funcţii de repatizare ideale în care să se evite orice coliziune. Acest lucru se numeşte *perfect hashing* şi există documentaţie în internet. În cazul general acest lucru este practic imposibil, astfel încât se recomandă construirea de funcţii de repartizare 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 construcţie a acestor funcţii. În plus, o funcţie de repartizare bună trebuie să utilizeze toate părţile unei chei pentru generarea valorii de repartizare. De exemplu, dacă se utilizează *string*-uri pentru chei, nu se vor considera doar primele 5 caractere pentru generarea valorii de repartizare sau dacă se folosesc chei numere întregi cu mai multe cifre, nu se vor utiliza doar anumite cifre ale acestora.

Se spune despre o funcţie de repartizare că obţine o avalanşă - *achieves avalanche* - dacă şi doar prin modificarea unui singur bit al unei chei se obţine o valoare de repartizare complet diferită de cea anterioară. Acest efect ajută la o distribuire uniformă a cheilor în tabelă şi la o reducere a probabilităţii coliziunilor. Conceptul de *avalanche* este derivat de la *hashing*-ul criptografic.

Este extrem de dificil de elaborat o funcţie de repartizare bună. De aceea se recomandă utilizarea unor funcţii deja existente, despre care se ştie că produc rezultate bune. În continuare sunt prezentate câteva funcţii de repartizare cunoscute împreună cu anumite proprietăţi ale lor. Sunt considerate chei de tip *string*, dar pot fi utilizate şi pentru chei de tip întreg, în loc de câte un caracter considerând câte un byte al cheii.