4

laborator 6

>> Structuri arborescente I

Conţinut

- Arbori
- Arbori binari
- Arbori binari de căutare
- Arbori binari de căutare indexaţi

Referinţe

- **T.H. Cormen, C.E. Leiserson, R.L. Rivest.** *Introducere în algoritmi: cap 5.5, cap.11.4, cap 13*, Editura Computer Libris Agora, 2000 (şi ediţiile ulterioare)
- **R. Ceterchi.** *Materiale de curs*, Anul universitar 2012-2013
- **http://laborator.wikispaces.com/**, *Tema 6*
- **D.E. Knuth.** *Tratat de programare a calculatoarelor - Sortare şi căutare*, Editura Tehnică, 1973 (şi ediţiile ulterioare)

Arbori

Un **arbore** T este un graf neorientat, conex şi aciclic, cu un nod evidenţiat ce se numeşte **rădăcină** şi pe care îl vom nota `root`.

*O mulţime de arbori se mai numeşte **pădure**. De fapt, o pădure este un graf aciclic, care nu este neapărat conex.*

Proprietăţi importante (de caracterizare) ale arborilor

- Dacă eliminăm o muchie oarecare din arbore obţinem un graf care nu mai este conex.
- Dacă adăugăm o muchie oarecare în arbore obţinem un graf care nu mai este aciclic.
- Un arbore este un graf conex în care numărul de muchii $|E|$ este egal cu numărul de vârfuri $|V|$ minus 1.
- Un arbore este un graf aciclic în care numărul de muchii $|E|$ este egal cu numărul de vârfuri $|V|$ minus 1.

Terminologie

Fie x un nod din arborele T.

Orice nod y pe drumul unic dintre `root` şi x se numeşte **strămoş** al lui x. Reciproc, x este un **descendent** al lui y. (Fiecare nod este considerat atât strămoş cât şi descendent al lui însuşi.)

Subarborele cu rădăcina x este arborele alcătuit din nodul x ca rădăcină şi toţi descendenţii acestuia.

Dacă pe drumul de la rădăcină la x ultima muchie este (y,x), atunci y se numeşte **părintele** sau **tatăl** lui x, iar x se numeşte **fiul** sau **copilul** lui y. Două noduri cu acelaşi părinte se numesc **fraţi**.

Singurul nod din arbore fără părinte este rădăcina.

Un nod fără fii se numeşte **nod extern** sau **frunză**. Un nod care nu este nod extern se numeşte **nod intern**.

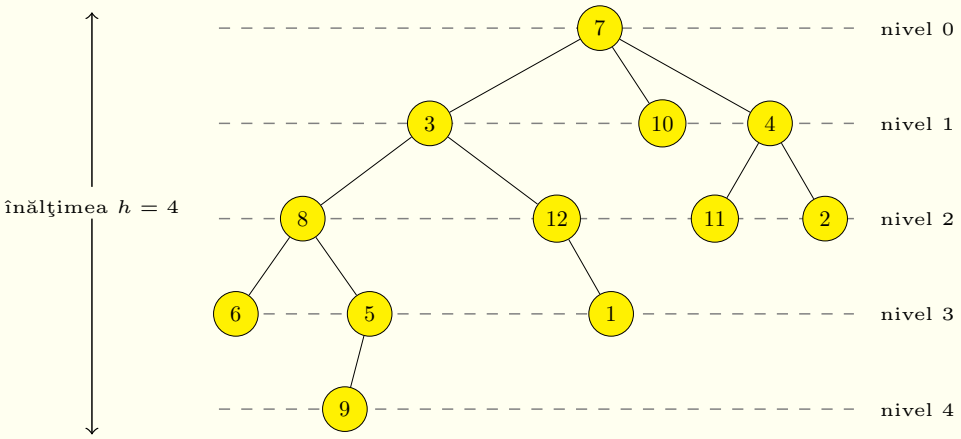
Gradul unui nod este numărul de fii al acestuia.

Înălţimea h a unui arbore este definită ca lungimea maximă a unui drum de la rădăcină la o frunză. Prin lungime se înţelege **numărul de muchii** care sunt parcurse în drumul respectiv. Arborele vid are înălţime 0, arborele care conţine doar un singur nod (rădăcina) are înălţime 0, un arbore care conţine două noduri (rădăcina şi unul din fii ei) are înălţime 1, etc.

*Într-un arbore, oricare două noduri sunt conectate printr-un drum elementar **unic**.*

Putem asocia **niveluri** nodurilor din arbore în modul următor: toate nodurile aflate la **adâncime** $0 \leq i \leq h$ se află la nivelul i, ca în figură. **Adâncimea** unui nod x este lungimea drumului (numărul de muchii) de la `root` la x (adâncimea rădăcinii este 0). Deci înălţimea arborelui este cea mai mare adâncime a unui nod din el.

Se numeşte **arbore ordonat** un arbore cu rădăcină în care fii oricărui nod sunt ordonaţi, mai exact dacă nodul are k fii, atunci există un prim fiu, un al doilea fiu, etc.



Figură 1 Structurarea unui arbore de înălţime h pe $h+1$ niveluri

Arbori binari

Un **arbore binar** T este o structură definită pe o mulţime finită de noduri, care:

- fie nu conţine niciun nod,
- fie conţine:
  - un nod rădăcină,
  - un abore binar numit subarborele stâng şi
  - un arbore binar numit subarbore drept.

*Un arbore care nu conţine niciun nod se numeşte **arbore vid** ($T = \emptyset$) sau **arbore nul** (`T=NIL`).*

Cu alte cuvinte, este vorba despre un arbore cu cel mult doi fii, numiţi fiu stâng, respectiv fiu drept.

Un **arbore binar complet** este un arbore binar în care fiecare nod este fie o frunză (de obicei reprezentată printr-un pătrat), fie are gradul exact 2 (niciunul din fii nu lipseşte). Dintr-un arbore binar se poate obţine un arbore binar complet, înlocuind fiecare fiu absent al unui nod cu o frunză.

Reprezentarea arborilor binari

Vom reprezenta arborii binari folosind structuri de date înlănţuite. Vor fi prezentate două metode de reprezentare. A doua reprezentarea se pretează pentru a fi extinsă pentru arbori cu oricâţi fii.

Prima metodă de reprezentare

Folosim două legături în fiecare nod, `left` şi `right`, pentru a reţine adresa fiului stâng, respectiv a fiului drept. Dacă un fiu lipseşte, adresa sa va fi `NIL` . Fiecare nod conţine şi un câmp de informaţie `info`. În plus, arborele trebuie să deţină o referinţă a rădăcinii sale, care păstrează numele de `root`.

Deşi câmpul de informaţie al nodurilor din arbori este în general numit cheie*, vom păstra denumirea `info` pentru liniaritate.*

Nodul se poate memora în C++ folosind următoarea structură:

```
struct nod {
    int info;
    nod *left;
    nod *right;
};
```

Arborele se poate reţine prin:

```
struct btree {
    nod *root;
    int height;
    int size;
};
```

Observaţi că putem reţine şi câmpuri adiţionale, care ne oferă informaţii suplimentare despre arbore, în tipul structurat ce descrie arborele. În acest caz păstrăm în câmpul `height` înălţimea arborelui spre rădăcina căruia pointează `root`. Câmpul `size` reţine numărul de noduri din arbore, notat şi n.

A doua metodă de reprezentare

A doua metodă de reprezentare, reţine şi legături de la fii la părintele lor. Deci putem cunoaşte părintele unui nod din arbore fără a reţine pointeri de traversare suplimentari. Atunci când operaţiile de bază (inserare, ştergere, căutare, accesare) pe un arbore utilizează des referinţe la părintele unui nod şi, mai mult, când aflarea acestuia nu este la îndemână (nu se poate face simultan), este indicat să facem un compromis la nivel de spaţiu de memorie pentru a obţine un timp de execuţie mai rapid pentru aceste operaţii (sau pentru a accesa mai rapid părintele sau pentru a scurta codul).

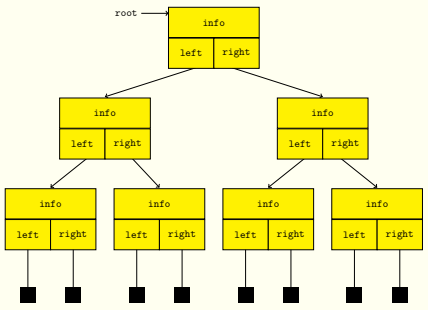
Nodul se poate memora în C++ folosind următoarea structură:

```
struct nod {
    int info;
    nod *left;
    nod *right;
    nod *parent;
};
```

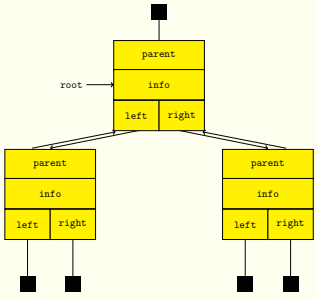
Situaţia este similară introducerii legăturii `previous` la listele dublu înlănţuite, când nu mai era necesar un pointer `anterior` pentru a obţine precedentul unui nod.

Această reprezentare poate fi generalizată pentru arbori cu un număr k de fii în modul următor. Orice nod reţine:

a. *adresa părintelui său,*
b. *adresa primului său fiu (cel mai din stânga) şi*
c. *adresa **fratelui** său (nodul cu acelaşi părinte aflat imediat în dreapta sa).*



(a) Reprezentarea unui arbore binar prin legături fiu stâng şi fiu drept

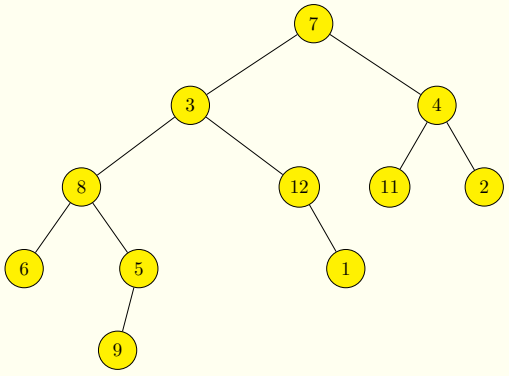


(b) Reprezentarea unui arbore binar prin legături fiu stâng, fiu drept şi părinte

Figură 2 Cele două metode de reprezentare a unui arbore binar

Traversări/parcurgeri ale arborilor

Vor fi prezentate patru modalităţi de parcurgere a unui arbore binar (parcurgerea pe niveluri/în lăţime şi trei parcurgeri în adâncime).



Figură 3 Un exemplu de arbore binar

Traversarea în lăţime/pe niveluri (breadth-first) a unui arbore

Parcurgerea breadth-first (BF) vizitează toate nodurile de pe un nivel de la stânga la dreapta, apoi trece la nivelul următor (primul nivel vizitat este 0).

Pentru arborele din imagine, parcurgerea BF produce secvenţa:

```
[7; 3; 4; 8; 12; 11; 2; 6; 5; 1; 9]
```

Algoritmul utilizează o structură de coadă Q. Funcţia push de inserare în coadă diferă de cea prezentată în laboratorul 4 prin faptul că de această dată inserăm variabile de tipul nodului din arbore şi nu valori întregi. Deci tipul câmpului `info` al unui nod din coada Q trebuie să fie pointer la un nod din arbore.

Deci, dacă un nod din arbore este de tipul `nod`, iar un pointer la un astfel de nod este de tipul `nod *`, atunci un element din coada Q va fi de tipul:

```
struct nodQ {
    nod *info;
    nod *next;
};
```

►► Traversare-BF()

```
1.  if root = NIL then
2.    return
3.  endif
4.  push(Q, root)
5.  while Q ≠ ∅ do
6.    pop(Q, x)
7.    Afişează x->info
8.    if x->left ≠ NIL then
9.      push(Q, x->left)
10.   endif
11.   if x->right ≠ NIL then
12.     push(Q, x->right)
13.   endif
14. endwhile
```

Algoritmul începe prin adăugarea rădăcinii arborelui în coadă (linia 4). La fiecare iteraţie a ciclului `while` , este scos din coadă un nod (cel aflat în coadă de cel mai mult timp) pentru afişare (liniile 6-7). Fiecare fiu nevid al nodului `x` este apoi adăugat în coadă (liniile 8-13). Modul de lucru al unei cozi ne asigură că vom parcurge integral nodurile de pe un nivel înainte de a trece la nodurile de pe nivelul următor.

Traversarea BF poate fi utilizată pentru adăugarea unui nod frunză într-un arbore binar, astfel încât să îl completăm pe niveluri: nu vom insera un nod pe nivelul următor, decât dacă nivelul curent este complet. (Arborele binar din imaginea anterioară nu este echilibrat, nu este complet).

Folosim următoarele observaţii în acest algoritm de inserare.

Orice arbore binar complet este echilibrat.

1. Pentru un arbore binar echilibrat (înălţimea oricăror doi subarbori cu rădăcină pe acelaşi nivel diferă prin cel mult 1), avem:

   $$h = \lfloor \log_2(n) \rfloor,$$

   unde n este numărul de noduri din arbore.
2. Într-un abore binar, orice nivel $0 \le i \le h$ complet conţine 2^i noduri: nivelul 0 ce conţine doar rădăcina are $2^0 = 1$ nod, nivelul 1 ce conţine cei doi fii ai rădăcinii are $2^1 = 2$ noduri, etc.
3. Asociem nodurilor din arbore indexul lor în parcurgerea BF (rădăcinii îi corespunde indexul 1, fiului ei stâng îi corespunde indexul 2, ..., ultimului nod vizitat îi corespunde indexul n.

   Atunci, pentru nodul cu indexul $1 \le i \le n$, avem că:

   a. fiul său stâng, dacă există, are indexul $2 \cdot i$ (număr par), iar
   b. fiul său drept, dacă există, are indexul $2 \cdot i + 1$ (număr impar).

Atunci când inserăm un nod astfel încât să completăm pe niveluri arborele binar, inserăm al $n+1$-lea nod din arbore, căruia îi va corespunde indexul $n+1$ în parcurgerea BF.

Noul nod va fi fiul stâng sau drept al unui nod cu indexul i. Putem determina care fiu va fi şi indexul i astfel:

a. dacă $n+1$ este un număr par, atunci va fi fiul stâng al nodului cu indexul i şi

$$n+1 = 2 \cdot i \Rightarrow i = \frac{n+1}{2}$$

b. dacă $n+1$ este un număr impar, atunci va fi fiul drept al nodului cu indexul işi

$$n+1 = 2 \cdot i + 1 \Rightarrow i = \frac{n+1-1}{2} = \frac{n}{2}$$

Aceasta înseamnă că putem afla indexul părintelui, apoi să parcurgem pe niveluri arborele până la acest index şi să legăm noul nod de nodul cu acest index. Algoritmul următor primeşte ca parametru un nod z şi îl inserează în arbore, conform strategiei de mai sus.

La implementarea în C++ (nu numai) liniile 1-5 pot fi înlocuite cu:
$i \leftarrow \frac{n+1}{2}$.

►► ADAUGĂ-FRUNZĂ-BF(z)

```
1.  if n+1 par then
2.    i ← (n+1)/2
3.  else
4.    i ← n/2
5.  endif
6.  contor ← 0
7.  if root ≠ NIL then
8.    PUSH(Q, root)
9.    while (Q ≠ ∅) and (c ≤ i) do
10.     POP(Q, x)
11.     c ← c + 1
12.     if x->left ≠ NIL then
13.       PUSH(Q, x->left)
14.     endif
15.     if x->right ≠ NIL then
16.       PUSH(Q, x->right)
17.     endif
18.   endwhile
19. endif
20. if contor = 0 then
21.   root ← z
22. else
23.   if n+1 par then
24.     x->left ← z
25.   else
26.     x->right ← z
27.   endif
28. endif
```

Traversarea în adâncime (depth-first) a unui arbore

Există trei posibilităţi pentru traversarea depth-first (DF) a unui arbore binar. Următoarele proceduri *recursive* de parcurgere, afişează cheile din noduri în trei ordini diferite, în funcţie de unde este plasată cheia din rădăcina x a unui arbore (de rădăcină x) faţă de cheile din subarborii stâng şi drept ai acestuia. Toate cele trei proceduri vor fi apelate prima oară pasând ca argument nodul rădăcină (`root`).

Aşa cum vedeţi în imaginea următoare, singurul nod din arbore a cărui legătură spre părinte va fi `NIL` este rădăcina.

Pentru arborele din imaginea anterioară:

- parcurgerea în preordine (RSD) produce secvenţa:
  `[7; 3; 8; 6; 5; 9; 12; 1; 4; 11; 2]`

- parcurgerea în inordine (SRD) produce secvenţa:

  `[6; 8; 9; 5; 3; 12; 1; 7; 11; 4; 2]`
- parcurgerea în postordine (SDR) produce secvenţa:

  `[6; 9; 5; 8; 1; 12; 3; 11; 2; 4; 7]`

Observaţi că în toate trei parcurgerile, cheile din subarborele stâng sunt afişate înaintea cheilor din subarborele drept.

- Parcurgerea în preordine (RSD)

►► TRAVERSARE-PREORDINE(x)

```
1.  if x ≠ NIL then
2.    Afişează x->info
3.    Traversare–Preordine(x->left)
4.    Traversare–Preordine(x->right)
5.  endif
```

- Parcurgerea în inordine (SRD)

►► TRAVERSARE-INORDINE(x)

```
1.  if x ≠ NIL then
2.    Traversare–Preordine(x->left)
3.    Afişează x->info
4.    Traversare–Preordine(x->right)
5.  endif
```

- Parcurgerea în postordine (SDR)

►► TRAVERSARE-POSTORDINE(x)

```
1.  if x ≠ NIL then
2.    Traversare–Preordine(x->left)
3.    Traversare–Preordine(x->right)
4.    Afişează x->info
5.  endif
```

Arbori binari de căutare

Într-un arbore binar de căutare (binary search tree - BST) cheile sunt întotdeauna memorate astfel încât să fie satisfăcută **proprietatea arborelui binar de căutare**:

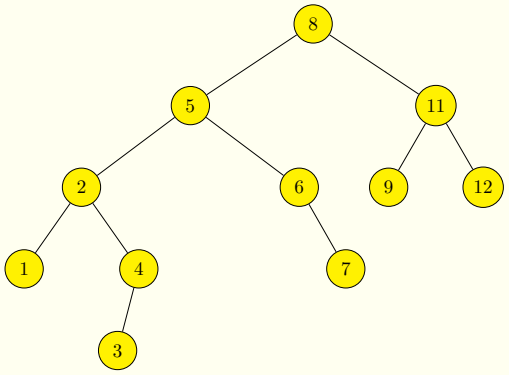
Fie x un nod din arborele binar de căutare. Dacă y este un nod din subarborele stâng al lui x, atunci `x->info` $\geq$ `y->info`. *Dacă y este un nod din subarborele drept al lui x, atunci* `x->info` $\leq$ `y->info`.[3]

Această proprietate se mai numeşte şi **invariantul de ordine**. Motivul pentru care ea constituie un invariant este evident: la orice moment ea trebuie îndeplinită/asigurată în arbore (deci nu „variază"). Identificarea invarianţilor este importantă pentru studiul complexităţii.

Traversarea în inordine a unui arbore binar de căutare afişează în ordine crescătoare toate *cheile* (valorile nodurilor) din arbore. (Deci, putem folosi această tehnică pentru sortare.) Pentru arborele binar de căutare din imagine parcurgerea în inordine (SRD) produce secvenţa:

```
[1; 2; 3; 4; 5; 6; 7; 8; 9; 11; 12]
```

Vom folosi pentru arborii binari de căutare a doua reprezentare prezentată, ce utilizează şi legătura către părintele unui nod.



Figură 4 Structura unui arbore binar de căutare

Căutarea

Vom implementa căutarea recursiv. Algoritmul următor caută în arborele de rădăcină `x` nodul care conţine cheia k. Dacă acesta exista se va returna un pointer către el, altfel se va returna `NIL` . Ca urmare a proprietăţii arborilor de căutare ştim că:

- dacă nodul x conţine o cheie (`x->info`) mai mică decât k, atunci, dacă există în arbore nodul cu cheia k, el se va afla sigur în subarborele drept al lui `x` (liniile 6-7).
- dacă nodul x conţine o cheie (`x->info`) mai mare decât k, atunci, dacă există în arbore nodul cu cheia k, el se va afla sigur în subarborele stâng al lui `x` (liniile 4-5).

În acest laborator lucrăm în ipoteza că nu există chei multiple (k apare cel mult o dată).

Evident, dacă `x->info` este chiar k, atunci căutarea s-a terminat (liniile 1-2). Dacă algoritmul conduce căutarea pe un fiu vid al lui `x`, atunci prin condiţia de la linia 1, la intrarea în procedură, se va returna `NIL` şi algoritmul se termină.

Iniţial, algoritmul se va apela cu argumentul `x=root`.

3 În funcţie de cum definim această proprietate, cu inegalităţi stricte sau nestricte, arborele binar de căutare va fi strict, strict la dreapta, strict la stânga sau nestrict. În acest laborator vom lucra cu arbori binari de căutare stricţi.

▶▶ Caută–Recursiv(x, k)

```
1.  if x = NIL OR k = x->info then
2.    return x
3.  endif
4.  if k < x->info then
5.    return Caută-Recursiv(x->left, k)
6.  else
7.    return Caută-Recursiv(x->right, k)
8.  endif
```

recursive search

Putem transforma algoritmul într-unul iterativ prin introducerea unui ciclu `while` .

▶▶ Caută–Iterativ(x, k)

```
1.  while x ≠ NIL AND k ≠ x->info do
2.    if k < x->info then
3.      x ← x->left
4.    else
5.      x ← x->right
6.    endif
7.  endwhile
8.  return x
```

Determinarea minimului şi a maximului

Observăm că nodul cu cheia minimă este obţinut dacă parcurgem arborele pornind din rădăcină şi folosind doar legătura către fiul stâng. Simetric, maximul se obţine printr-o traversare ce utilizează doar legătura către fiul drept. Aceasta se datorează proprietăţii arborelui binar de căutare.

Procedurile următoare returnează un pointer către elementul cu cheia minimă, respectiv elementul cu cheia maximă din arbore.

►► MINIM(x)

```
1. while x->left ≠ NIL do
2.   x ← x->left
3. endwhile
4. return x
```

►► MAXIM(x)

```
1. while x->right ≠ NIL do
2.   x ← x->right
3. endwhile
4. return x
```

Determinarea succesorului

Pentru operaţia de ştergere vom avea nevoie să înlocuim nodul şters cu un nod rămas din arbore astfel încât proprietatea arborelui binar de căutare să fie păstrată. Mai exact, în locul său va fi plasat acel nod ce conţine cea mai mică cheie ce este mai mare decât cea a nodului şters. Cu alte cuvinte înlocuim un nod x cu nodul y, pentru care avem:

$$y\text{->}info = \min\{z\text{->}info > x\text{->}info\}, \forall\ z \in T.$$

Un asemenea nod y se numeşte **succesorul** lui x.

Pentru găsirea succesorului putem utiliza următorul algoritm.

Similar, predecesorul unui nod x este nodul y cu cea mai mare cheie care este mai mică decât cea a nodului x. Adică:

$$y\text{->}info = \max\{z\text{->}info < x\text{->}info\}, \forall\ z \in T.$$

►► SUCCESOR(x)

```
1. if x->right ≠ NIL then
2.   return MINIM(x->right)
3. endif
4. y ← x->parent
5. while (y ≠ NIL) and (x = y->right) do
6.   x ← y
7.   y ← y->parent
8. endwhile
9. return y
```

successor

Dacă un nod x deţine un fiu drept, `x->right`, atunci succesorul său va fi cheia minimă din subarborele de rădăcină `x->right`. În caz contrar, (fiul drept, `x->right`, este vid), succesorul y va fi strămoşul aflat la cea mai mică adâncime al lui x care are un fiu stâng, `y->left`, ce este de asemenea strămoş al lui x.

Inserarea

Operaţia de inserare, ca şi ştergerea, modifică mulţimea de chei reprezentată prin arbore. Trebuie să ne asigurăm că, după inserare, proprietatea arborelui de căutare a fost conservată.

Algoritmul următor inserează nodul z în arborele T.

►► Inserează(T, z)

```
1.  y ← NIL
2.  x ← T->root
3.  while x ≠ NIL do
4.    y ← x
5.    if z->info < x->info then
6.      x ← x->left
7.    else
8.      x ← x->right
9.    endif
10. endwhile
11. z->parent ← y
12. if y = NIL then
13.   T->root ← z
14. else
15.   if z->info < y->info then
16.     y->left ← z
17.   else
18.     y->right ← z
19.   endif
20. endif
```

Deoarece procedura `Inserează(T,z)` primeşte ca argument un nod, pentru inserarea unei chei, puteţi:

a) să modificaţi procedura astfel încât să primească valoarea de inserat (`val`, de acelaşi tip de date ca `info`) şi să creaţi şi iniţializaţi nodul la intrarea în procedură, sau

b) să scrieţi o procedură separată care creează şi iniţializează nodul nou, după care apelează `Inserează(T,z)` cu acest nod.

insertion

În prima fază trebuie să găsim locul potrivit pentru nodul nou. Liniile 1-10 identifică părintele y al noului nod. Pentru a-l găsi pe y se face o parcurgere în adâncime, ţinând cont de proprietatea arborelui de căutare, până se găseşte un nod terminal x. Părintele acestui nod este referit de y. Parcurgerea efectuată este de fapt căutarea cheii din noul nod, deci vom găsi locul în care aceasta ar trebui să se afle. A doua fază se concentrează pe legarea noului nod de arbore. Noul nod trebuie să conţină o referinţă către părintele său (prima legătură), iar părintele trebuie să conţină o referinţă către noul său fiu (a doua legătură). Linia 11 realizează prima legătură, iar liniile 12-20 plasează noul nod, z, ca fiu al lui y în locul corespunzător (a doua legătură). Linia 13 reprezintă cazul în care T este chiar arborele vid şi atunci noul nod va deveni chiar rădăcina acestuia.

Ştergerea

Pentru ştergerea unui nod `z` dintr-un arbore binar de căutare `T` cu rădăcina `T->root`, putem avea trei situaţii:

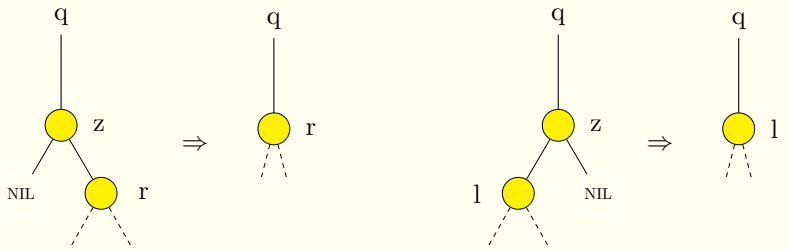
i. dacă `z` este o frunză (nu are niciun fiu), atunci eliminăm nodul `z`:
   - fie prin `(z->parent)->left` ← `NIL` , dacă `(z->parent)->left = z` (`z` era fiu stâng),
   - fie prin `(z->parent)->right` ← `NIL` , dacă `(z->parent)->right = z` (`z` era fiu drept)

ii. dacă `z` are un singur fiu, atunci acel fiu va fi „ridicat” în locul său. Să spunem că `x` reţine fiul nevid al lui `z` (fie `z->left`, fie `z->right`). Atunci eliminarea şi implicit „ridicarea” se realizează:
   - fie prin `(z->parent)->left` ← `x`, dacă `(z->parent)->left = z` (`z` era fiu stâng),
   - fie prin `(z->parent)->right` ← `x`, dacă `(z->parent)->right = z` (`z` era fiu drept)

iii. dacă `z` are ambii fii nevizi, atunci în locul lui `z` va fi plasat `y`, succesorul lui `z`[4]. Pentru aceasta, practic îl ştergem pe `y` din subarborele drept al lui `z` (nu se poate afla decât în subarborele drept) şi tot subarborele drept al lui `z` rămas devine subarborele drept al lui `y`, respectiv subarborele stâng al lui `z` (pe care nu îl modificăm) devine subarborele stâng al lui `y`. Nodul `y` poate fi chiar `z->right`.

4 Într-un arbore binar de căutare succesorul unui nod nu are fiu stâng, iar predecesorul unui nod nu are fiu drept. Altfel, acel fiu ar fi fost chiar el succesorul, respectiv predecesorul.

Practic, putem trata cazul i. împreună cu ii.. Dacă z nu are fiu stâng, atunci z va fi înlocuit de fiul drept. Dacă fiul drept este şi el vid (cazul i.), atunci automat atribuim `NIL` , aşa cum trebuia. Mai jos, figura 5(a) conţine această situaţie, iar figura 5(b) conţine situaţia simetrică, atunci când z nu are fiu drept şi este înlocuit de fiul stâng.

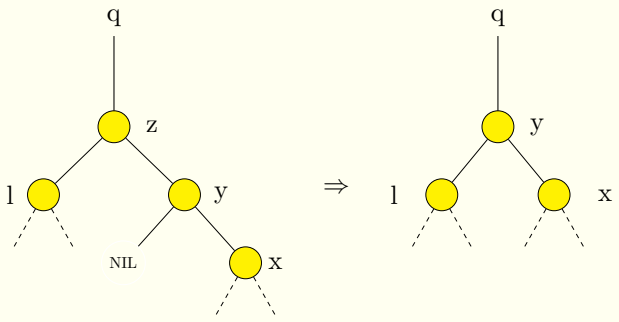


(a) Cazul când *z* nu are fiu stâng **(b)** Cazul când *z* nu are fiu drept

Figură 5 Cazurile i. şi ii. tratate concomitent

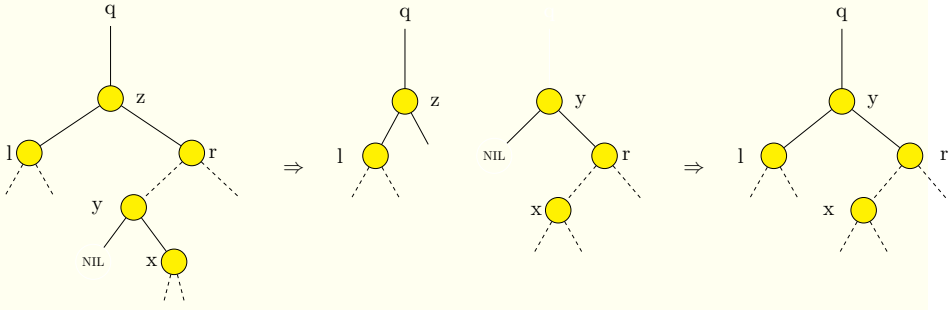
Rămâne tratarea cazului iii. Găsim succesorul y al lui z. Trebuie să îl eliminăm pe y de la locul lui şi să îl aducem în locul lui z:

- dacă `y=z->right`, atunci z este înlocuit de y (situaţia este ilustrată în figura 6),



Figură 6 Cazul când succesorul este chiar fiul drept al nodului *z*

- altfel, îl înlocuim pe y cu fiul său drept şi apoi îl înlocuim pe z cu y (situaţia este ilustrată în figura 7).



Figură 7 Cazul când succesorul nu este fiul drept al nodului *z*

Obţinem astfel 4 cazuri ce se vor regăsi în algoritmul de ştergere.

Utilizăm următoarea procedură, numită `Transplantează(T, u, v)`, pentru înlocuirea subarborelui de rădăcină u cu subarborele de rădăcină v: v devine fiul (stâng sau drept) al părintelui lui u.

►► TRANSPLANTEAZĂ(T, u, v)

```
1.  if u->parent = NIL then
2.    T->root = v
3.  else
4.    if u = (u->parent)->left then
5.      (u->parent)->left  ← v
6.    else
7.      (u->parent)->right  ← v
8.    endif
9.  endif
10. if v ≠ NIL then
11.   v->parent  ←  u->parent
12. endif
```

În continuare aveţi pseudocodul pentru abordarea prezentată.

►► ŞTERGE(T, z)

```
1.  if z->left = NIL then
2.    Transplantează(T, z, z->right)
3.  else
4.    if z->right = NIL then
5.      Transplantează(T, z, z->left)
6.    else
7.      y  ←  Minim(z->right)
8.      if y->parent ≠ z
9.        Transplantează(T, y, y->right)
10.       y->right  ←  z->right
11.       (y->right)->parent  ←  y
12.     endif
13.     Transplantează(T, z, y)
14.     y->left  ←  z->left
15.     (y->left)->parent  ←  y
16.   endif
17. endif
```

Ca şi în cazul inserării, următoarele proceduri primesc ca parametru un pointer către nodul z, ce trebuie şters, nu cheia `z->info`.

De ce este corect să folosim apelul `Minim(z->right)` (linia 7) pentru găsirea succesorului lui y?

Altă procedură pentru ştergere

Alternativ, putem utiliza următoarea procedură de ştergere care tratează cazurile i., ii. şi iii.

Dacă suntem în cazul i. (z nu are fii, atunci părintele lui z va reţine **NIL** în locul legăturii către z. Dacă are un singur fiu (cazul ii.), acesta este preluat ca fiu, pe poziţia corespunzătoare, de părintele lui z. În fine, dacă ambii fii sunt nevizi, este găsit succesorul y care va fi eliminat şi îl va înlocui pe z.

Urmăriţi algoritmul şi depistaţi cum se realizează tratarea cazurilor, deoarece ele sunt organizate diferit.

►► ŞTERGE-2(T, z)

```
1.  if (z->left = NIL) or (z->right = NIL) then
2.    y ← z
3.  else
4.    y ← Succesor(z)
5.  endif
6.  if y->left ≠ NIL then
7.    x ← y->left
8.  else
9.    x ← y->right
10. endif
11. if x ≠ NIL then
12.   x->parent ← y->parent
13. endif
14. if y->parent = NIL then
15.   T->root ← x
16. else
17.   if y = (y->parent)->left then
18.     (y->parent)->left ← x
19.   else
20.     (y->parent)->right ← x
21.   endif
22. endif
23. if y ≠ z then
24.   z->info ← y->info
25. endif
26. return y
```

Arbori binari de căutare indexaţi

Definiţie: Un *arbore binar de căutare indexat* este un arbore binar de căutare, în care fiecare nod are un câmp suplimentar, numit `leftSize`, în care se reţine numărul de descendenţi din subarborele stâng.

Prin urmare nodul corespunzător acestui tip de arbore se poate implementa în C++ folosind următorul tip structurat:

```
struct nod {
    int info;
    nod *left;
    nod *right;
    int leftSize;
};
```

Un exemplu de arbore binar de căutare indexat este în figura 8(a).

Indexul unui element este poziţia sa la traversarea în inordine (SRD) a arborelui cu rădăcina în acel element. Pentru arborele de mai sus, traversarea în inordine este dată de secvenţa următoare:

$$[2,6,7,8,10,15,18,20,25,30,35,40]$$

Indexul fiecărui element este atunci cel din tabelul de mai jos.

nod x	2	6	7	8	10	15	18	20	25	30	35	40
index(x)	0	1	2	3	4	5	6	7	8	9	10	11

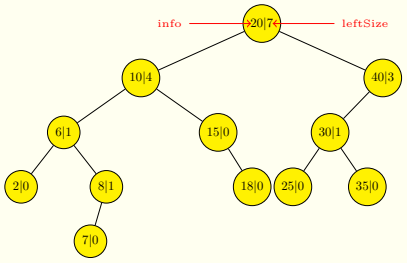
Observăm că `x->leftSize == index(x)`, relativ la elementele din subarborele stâng (ce are rădăcina în nodul `x`). Spre exemplu, pentru nodul 20 (`x->info=20`), care constituie rădăcina întregului arbore (corespunzător parcurgerii în inordine de mai sus), `leftSize` este 7, adică exact `index(20)`. Considerăm acum subarborele cu rădăcina în nodul cu `x->info = 6` (cel ce conţine nodurile 2, 8 şi 7). Parcurgerea sa în inordine produce: [2, 6, 7, 8]. Prin urmare indexul asociat nodului 6 este 1, exact valoarea ce va fi reţinută în `x->leftSize`.

Căutarea unui element cu un anumit index

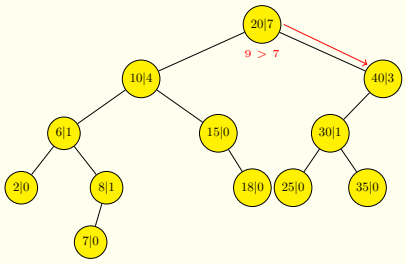
Se doreşte găsirea în arbore a nodului `x` pentru care `index(x)=k`. Ca urmare a observaţiei anterioare, putem deduce indexul unui nod utilizând câmpul `leftSize`.

Pentru găsirea nodului căutat se va aplica recursiv următorul algoritm, pornind de la rădăcină.

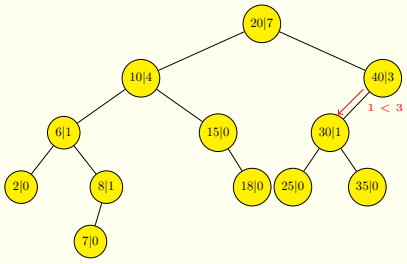
- Dacă `k = x->leftSize`, atunci nodul căutat este `x`.
- Dacă `k < x->leftSize`, atunci nodul căutat se află în subarborele stâng al lui `x` pe.poziţia `k` şi se reia algoritmul pentru `x->left`.
- Dacă `k > x->leftSize`, atunci nodul căutat se află în subarborele drept al lui `x` pe.poziţia `k - x->leftSize - 1` şi se reia algoritmul pentru `x->right`.



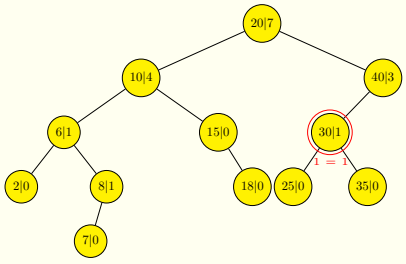
(a) Căutăm nodul `x` cu `index(x)`=9 în arbore. Avem deci `k`=9.



(b) Primul nod verificat va fi 20, rădăcina. Ne aflăm în cazul 3, când `k > x->leftSize`, deoarece `k`=9>7, unde 7 este valoarea câmpului *leftSize* a nodului 20. Prin urmare, continuăm pe subarborele drept al lui 20 şi vrem să găsim nodul de la poziţia `k - x->leftSize - 1 = 9 - 7 - 1 =` 1. Fiul drept al lui 20 este 40.



(c) Nodul 40 are `leftSize`=3, deci este strict mai mare decât `k`=1.Ne aflăm în cazul 2. Continuăm să căutăm nodul de la poziţia `k`=1 în subarborele stâng al lui 40, a cărui rădăcină este 30.



(d) Obţinem cazul 1, când `k=x->leftSize`, deoarece 30 reţine în câmpul `leftSize` valoarea 1, exact valoarea căutată. Algoritmul se încheie.

Figură 8 Exemplu de căutare într-un arbore binar de căutare indexat

Atenţie Algoritmul de mai sus funcţionează în ipoteza că indexarea se face începând cu nodul `x` pentru care `index(x)`=0, aşa cum s-a procedat aici.

PROBLEME

1. **(2p)** Să se implementeze o structură de arbore binar cu cheile numere întregi, inserate pe niveluri. Scrieţi funcţii pentru:
   a. Aăugarea unui nod frunză;
   b. Parcurgerea cheilor conform strategiei RSD (preordine);
   c. Parcurgerea cheilor conform strategiei SRD (inordine);
   d. Parcurgerea cheilor conform strategiei SDR (postordine);
2. **(5p)** Să se implementeze un arbore binar de căutare cu următoarele operaţii:
   a. `insert(root,x)` – inserează cheia `x` în arborele de rădăcină `root`;
   b. `search(root,x)` – returnează 1 dacă elementul `x` se află în arborele de rădăcină `root` şi 0 în caz contrar;
   c. `findMax(root)` – returnează elementul maxim din arborele de rădăcină `root`, fără a-l şterge din arbore;
   d. `delete(root,x)` – şterge nodul cu cheia `x` din arborele de rădăcină `root` (păstrând proprietatea de arbore binar de căutare);
3. **(1p)** Să se utilizeze un arbore binar de căutare pentru sortarea a n numere.
4. **(2p)** Dat un arbore binar de căutare şi doi întregi k_1 şi k_2, să se afişeze toate cheile `x` din arbore care au proprietatea: $k_1 \leq x \leq k_2$.
5. **(3p)** Să se scrie un algoritm pentru afişarea elementului de pe poziţia k (în ordinea crescătoare a elementelor dintr-un şir), folosind un arbore binar de căutare indexat.
6. **(1p)** Să se ordoneze descrescător un şir de cuvinte citite de la tastatură, folosind un arbore binar de căutare.
7. **(10ps)** Zece haiduci au dat peste o comoară de 50 de galbeni. Ei vor sa împartă banii după următorul sistem:

   (a) cel mai bătrân haiduc propune o schemă de distribuire a monedelor;

   (b) haiducii votează dacă sunt de acord cu această schemă; spunem că haiducii sunt de acord cu schema atunci când majoritatea votează în favoarea acesteia. În cazul în care sunt voturi egale pro şi contra, atunci schema este adoptată;

   (c) dacă haiducii sunt de acord cu schema, atunci banii se împart conform propunerii; dacă nu, atunci haiducul care a făcut propunerea este ucis, şi următorul cel mai bătrân haiduc face o nouă propunere.

   Fiecare haiduc îşi bazează deciziile pe următoarele considerente:

   (a) vrea să supravieţuiască;

   (b) vrea să maximizeze suma care îi revine în urma împărţirii;

   (c) nu are încredere în ceilalţi haiduci, aşa că nu sunt posibile aranjamente între ei pentru a împărţi banii.

   Numerotând haiducii cu `H10; H9; ... ; H1` (unde `H10` este cel mai bătrân haiduc, iar `H1` cel mai tânăr), să se spună care este schema de împărţire a monedelor.

■ Termen de predare: **Săptămâna 9 (26–30 noiembrie 2012) inclusiv.**

■ Detalii: **Studenţii pot obţine un maxim de 13 puncte. Problemele 1-2 şi 6 sunt obligatorii. Problemele 3–5 sunt suplimentare. Problema 7 este facultativă, iar termenul de predare pentru ea este săptămâna 8 (19–23 noiembrie). Un singur student poate rezolva problema facultativă.**