Arbori binari de căutare

Arbori Binari de Căutare

Un **arbore binar de căutare** este un arbore **binar** care satisface următoarea proprietate:

Pentru un nod x:

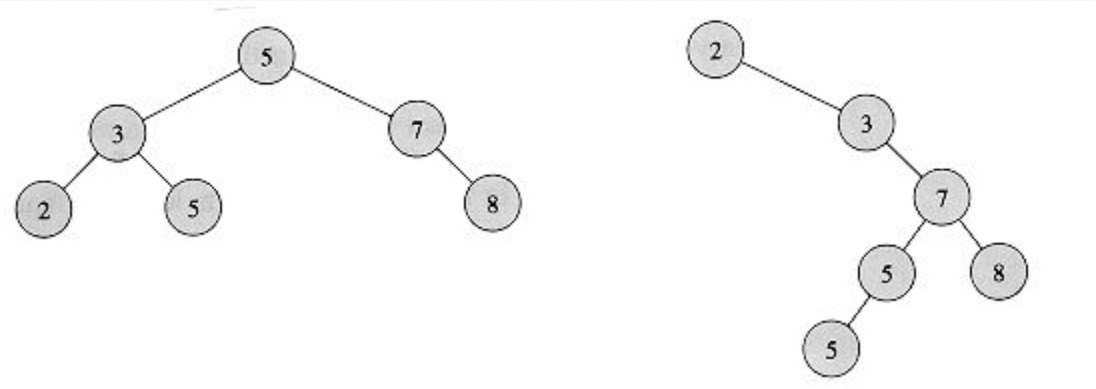
- Dacă **y** este un nod din subarborele *stâng* al lui **x**, atunci `cheie[y]` ≤ `cheie[x]`
- Dacă **y** este un nod din subarborele *drept* al lui **x**, atunci `cheie[x]` ≤ `cheie[y]`

Arbori Binari de Căutare

Un **arbore binar de căutare** este un arbore **binar** care satisface următoarea proprietate:

Pentru un nod x:

- Dacă **y** este un nod din subarborele ***stâng*** al lui **x**, atunci `cheie[y]` ≤ `cheie[x]`
- Dacă **y** este un nod din subarborele ***drept*** al lui **x**, atunci `cheie[x]` ≤ `cheie[y]`



Arbori Binari

Un ***arbore binar strict*** este un arbore binar în care fiecare nod fie nu are nici un fiu, fie are exact doi fii.

Nodurile cu doi copii se vor numi ***noduri interne***, iar cele fără copii se vor numi ***noduri externe*** sau ***frunze***.

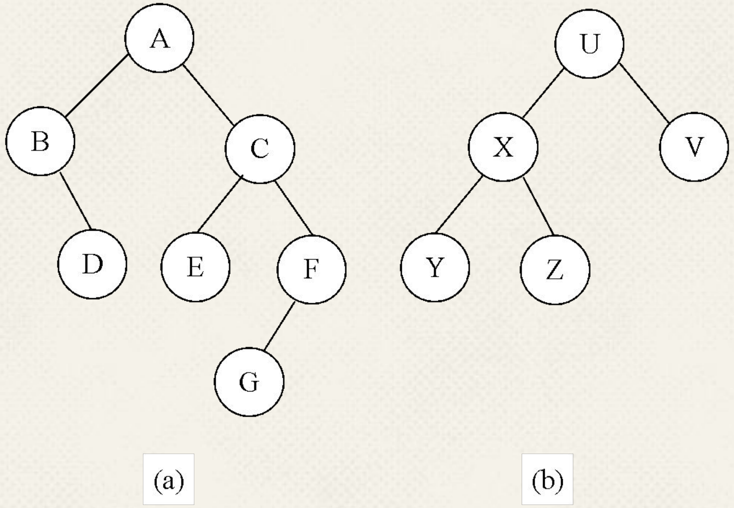
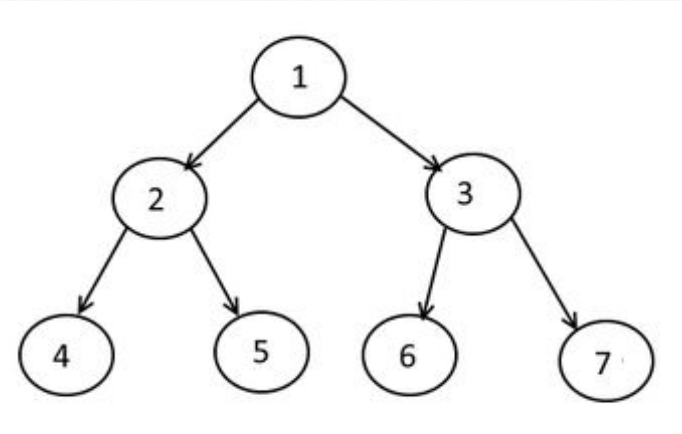


Fig.4.1.1. (a) Un arbore binar nestrict. (b) Arbore binar strict.

Arbori Binari - Parcurgeri

Parcurgeri în arbori binari:

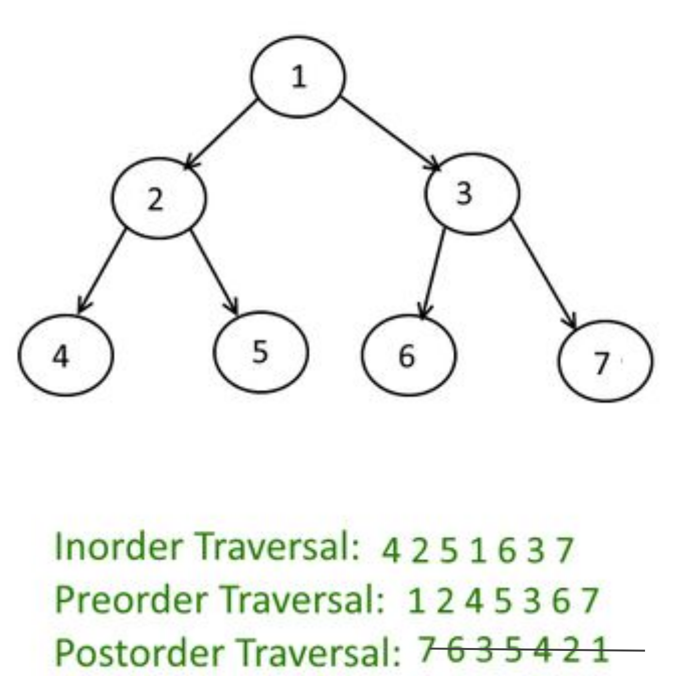


- **Inordine** (SRD, stânga rădăcină dreapta)
- **Preordine** (RSD, rădăcină stânga dreapta)
- **Postordine** (SDR, stânga dreapta rădăcină)

Arbori Binari - Parcurgeri

Parcurgeri în arbori binari:

- **Inordine** (SRD, stânga rădăcină dreapta)
- **Preordine** (RSD, rădăcină stânga dreapta)
- **Postordine** (SDR, stânga dreapta rădăcină)



4 5 2 6 7 3 1

Arbori Binari - Parcurgeri

```
void par_rsd (BTREE t) {
	if (t != NULL) {
		visit(t);
		par_rsd(t->left);
		par_rsd(t->right);
	}
}
```

```
void par_srd (BTREE t) {
	if (t != NULL) {
		par_srd(t->left);
		visit(t);
		par_srd(t->right);
	}
}
```

```
void par_sdr (BTREE t) {
	if (t != NULL) {
		par_sdr(t->left);
		par_sdr(t->right);
		visit(t);
	}
}
```

Link pt vizualizare

TEMĂ: Se dau SRD și RSD. Afișati arborele

Arbori Binari de Căutare

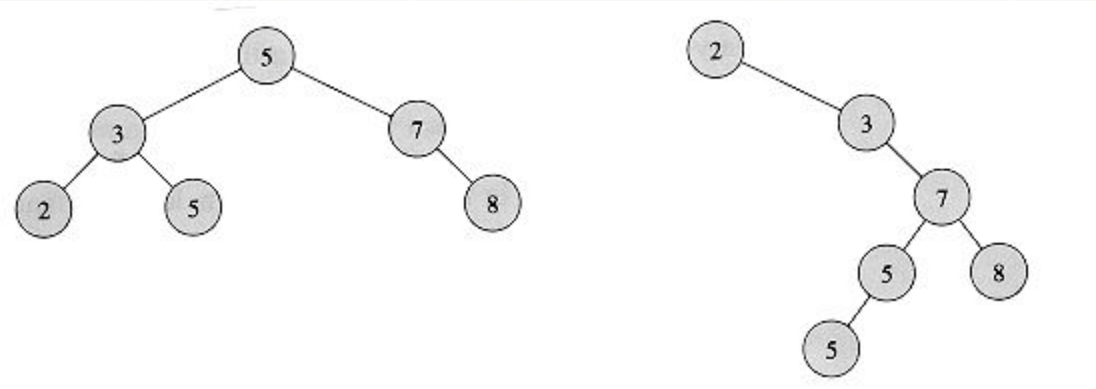
- Înălțimea arborelui ?
  - Minim ?
  - Maxim ?

Arbori Binari de Căutare

- Înălțimea arborelui ?
  - Minim
    - Arbore Binar Complet → Înălțime **log n**
  - Maxim
    - Dacă avem lanț (elementele sunt inserate în ordine crescătoare sau descrescătoare) → Înălțime **n**

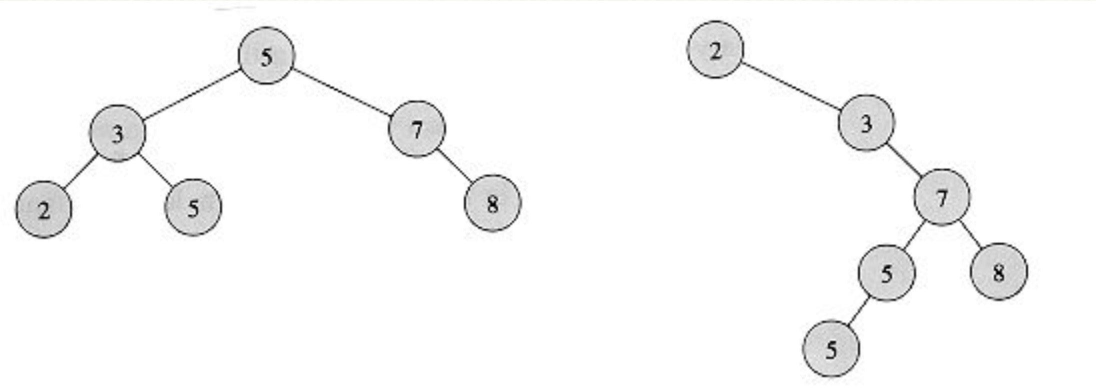
Arbori Binari de Căutare

- Ce parcurgere ne oferă vectorul sortat?
  - Preordine
  - Inordine
  - Postordine



Arbori Binari de Căutare

- Parcurgerea **inordine** ne oferă vectorul sortat
  - Preordine 5 3 2 5 7 8 | 2 3 7 5 5 8
  - Inordine 2 3 5 5 7 8 | 2 3 5 5 7 8
  - Postordine 2 5 3 8 7 5 | 5 5 8 7 3 2
- Restul parcurgerilor sunt diferite pentru cei 2 arbori.

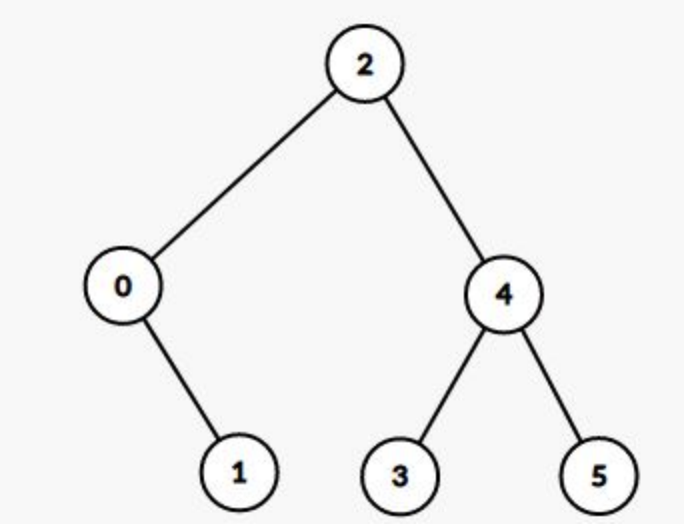


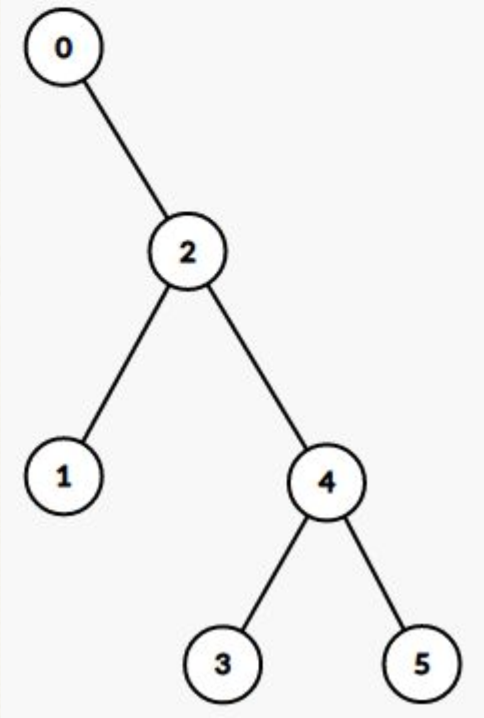
Exercițiu

Desenați arbori binari de înălțime 2, 3, 4, 5 pentru valorile {1, 2, 3, 4, 5}.

Exercițiu

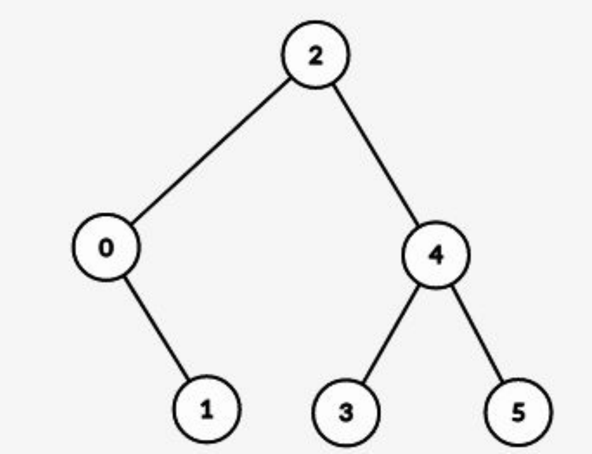
Desenați arbori binari de înălțime 2, 3, 4, 5 pentru valorile {0, 1, 2, 3, 4, 5}.

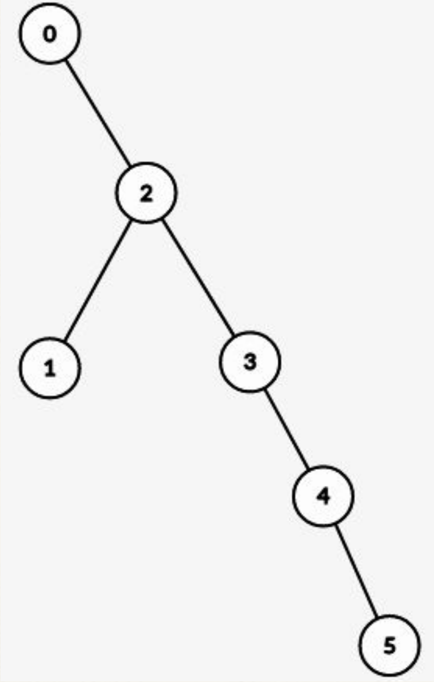


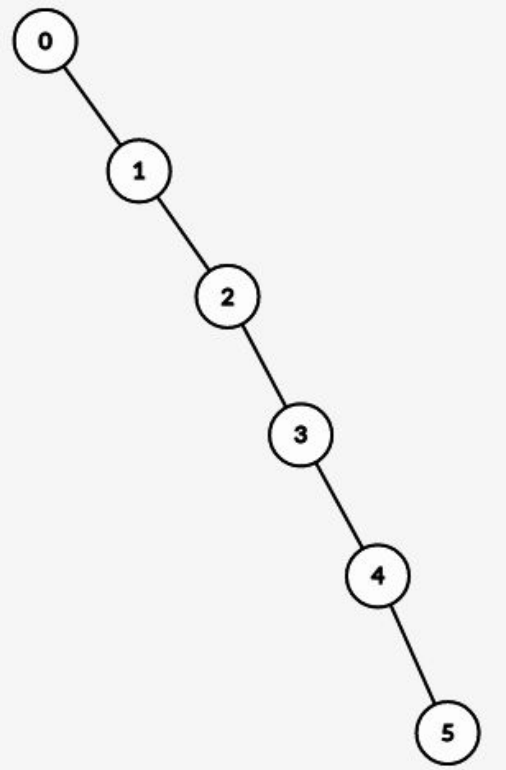


Exercițiu

Desenați arbori binari de înălțime 2, 3, 4, 5 pentru valorile {0, 1, 2, 3, 4, 5}.

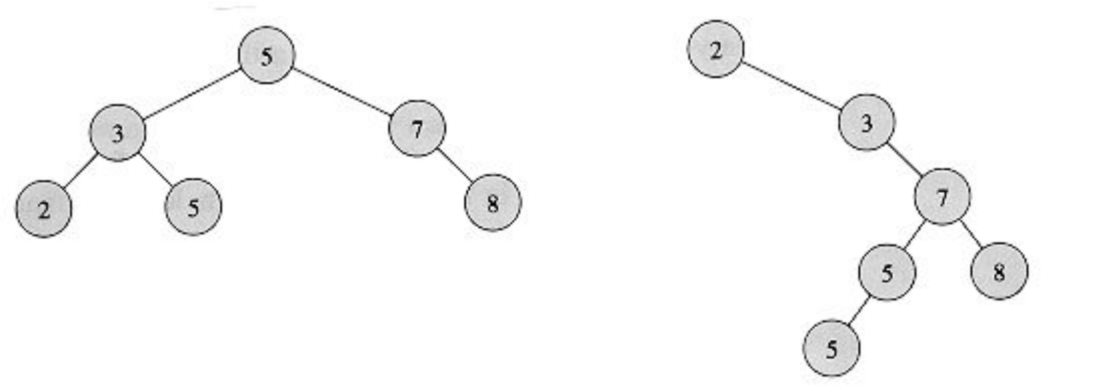






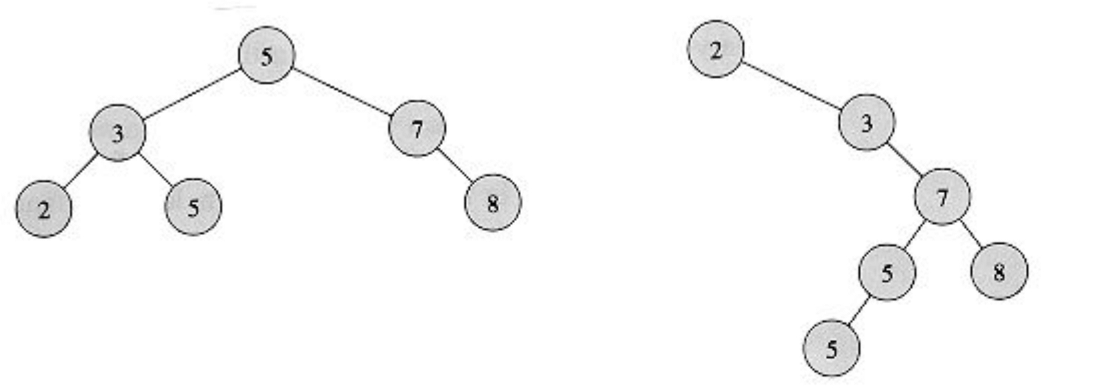
Minim și Maxim

- Unde se află minimul?



Minim și Maxim

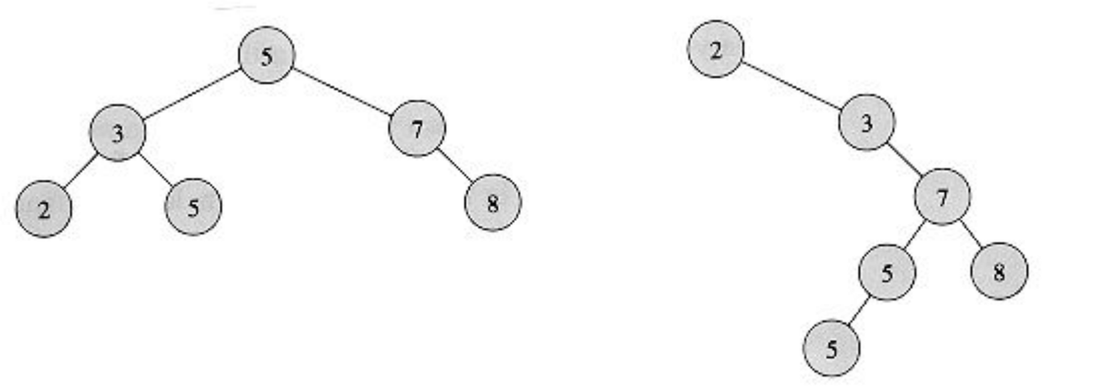
- Unde se află minimul?
  - În cel mai din stânga nod




```
TREE-MINIMUM (x)
while left[x] ≠ NIL
    do x ← left[x]
return x
```

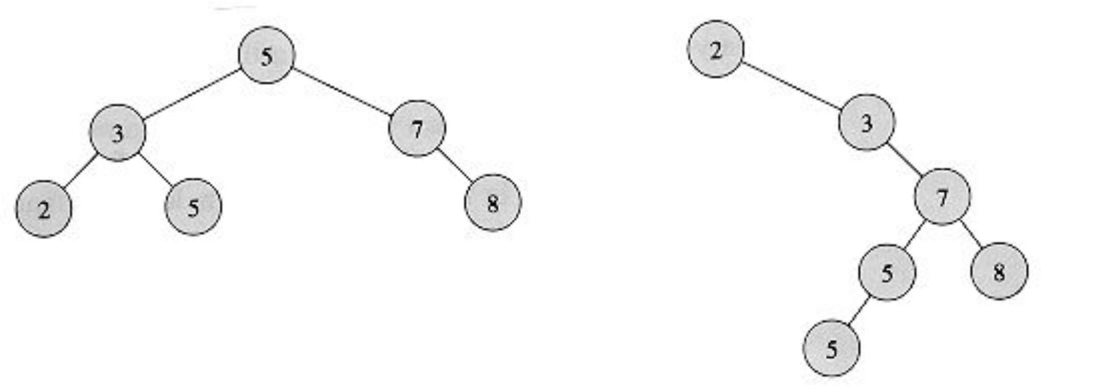
Minim și Maxim

- Unde se află maximul?



Minim și Maxim

- Unde se află maximul?
  - În cel mai din dreapta nod

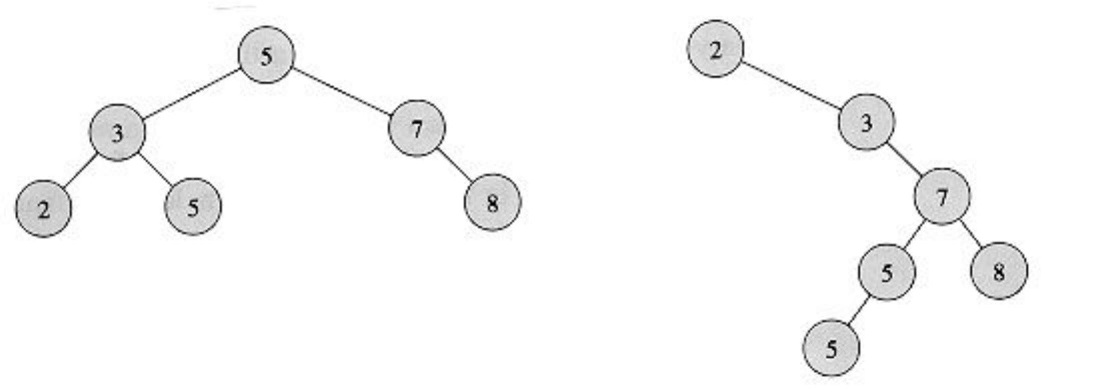



```
TREE-MAXIMUM (x)
while right[x] ≠ NIL
    do x ← right[x]
return x
```

Complexitate?

Minim și Maxim

- Unde se află maximul?
  - În cel mai din dreapta nod




```
TREE-MAXIMUM (x)
while right[x] ≠ NIL
    do x ← right[x]
return x
```

Complexitate? O(h)

Căutare

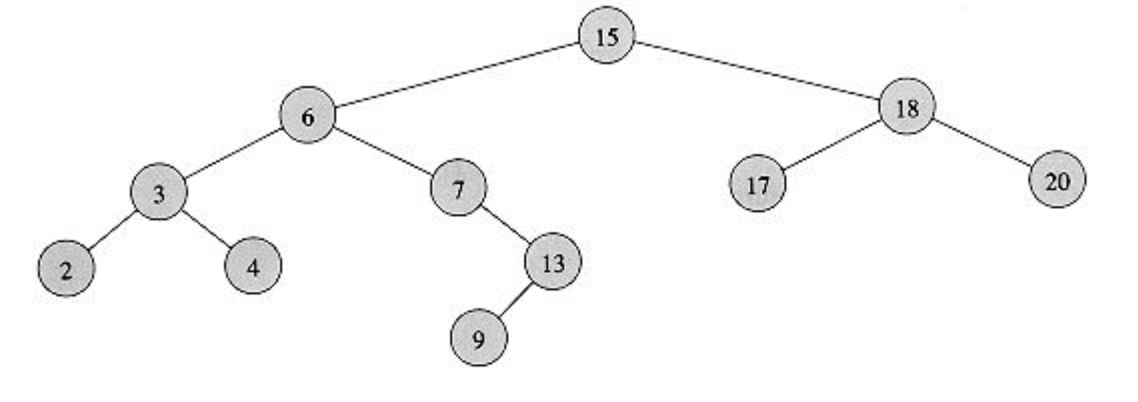
Minimul și maximul se găsesc mai greu decât într-un heap. Avantajul major al arborilor binari de căutare este că permit o căutare “relativ” eficientă.

Cum găsim un element?

Căutare

Minimul și maximul se găsesc mai greu decât într-un heap. Avantajul major al arborilor binari de căutare este că permit o căutare “relativ” eficientă.

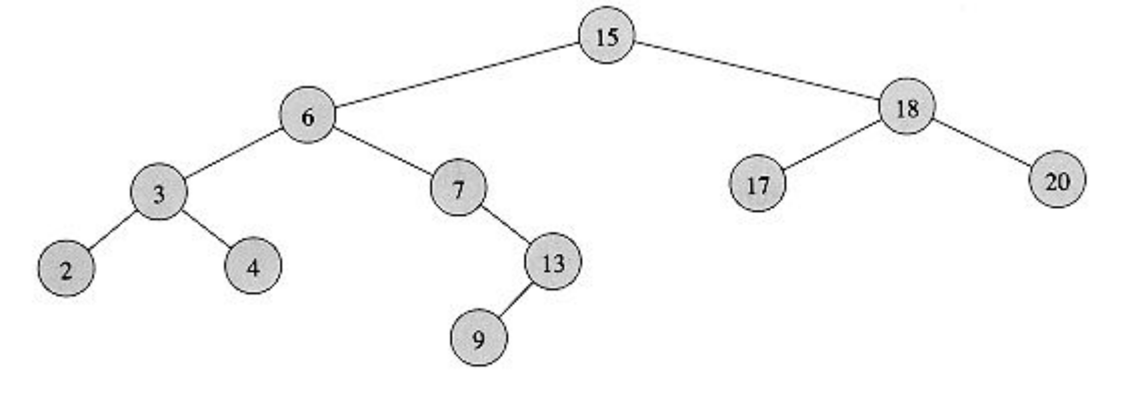
Cum găsim un element?



Căutare

Începem din rădăcină și dacă valoarea din nodul curent este mai mică decât ceea ce căutăm, mergem în stânga, dacă valoarea e mai mare, mergem în dreapta.

Evident, ne oprim dacă am găsit valoarea.

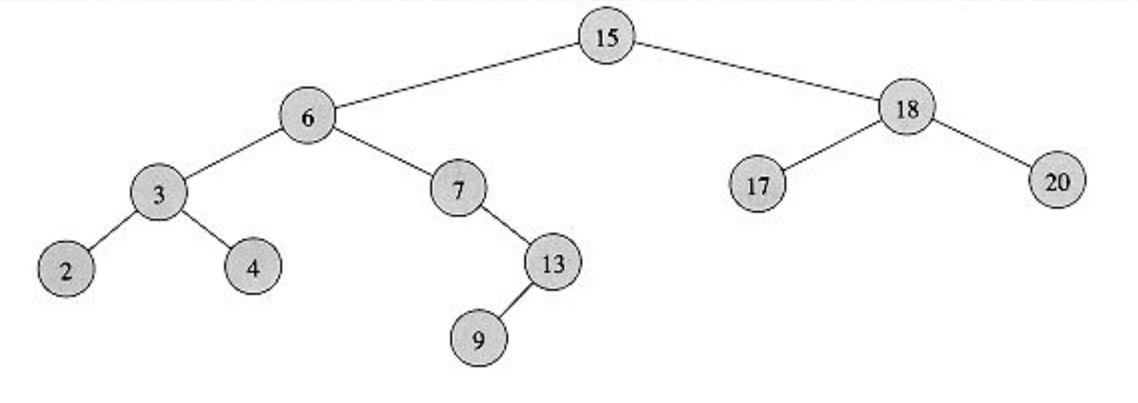


Căutare

```
ITERATIVE-TREE-SEARCH (x,k)
while x ≠ NIL and k ≠ key[x]
    do if k < key[x]
        then x ← left[x]
        else x ← right[x]
return x
```

Complexitate: **O(h)**

```
TREE-SEARCH (x, k)
if x = NIL or k = key[x]
    then return x
 if k < key[x]
    then return TREE-SEARCH (left[x], k)
    else return TREE-SEARCH (right[x], k)
```



Succesor / Predecesor

Până acum, puteam să ținem un dicționar și un heap și să facem aceleași operații.

Succesor: Se dă un nod din arbore.
Care este cea mai **mică** valoare din arbore ⩾ val[x] (valorea nodului)?

Predecesor: Se dă un nod din arbore.
Care este cea mai **mare** valoare din arbore ⩽ val[x] (valorea nodului)?

Cum facem?

Succesor / Predecesor

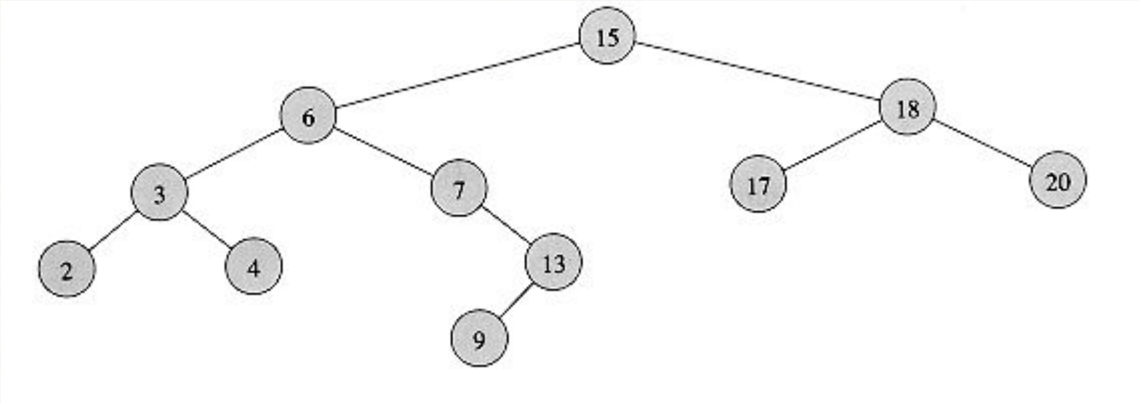
Succesor de 3?

Succesor de 6?

Succesor de 15?

Succesor de 13?

Succesor de 4?



Succesor / Predecesor

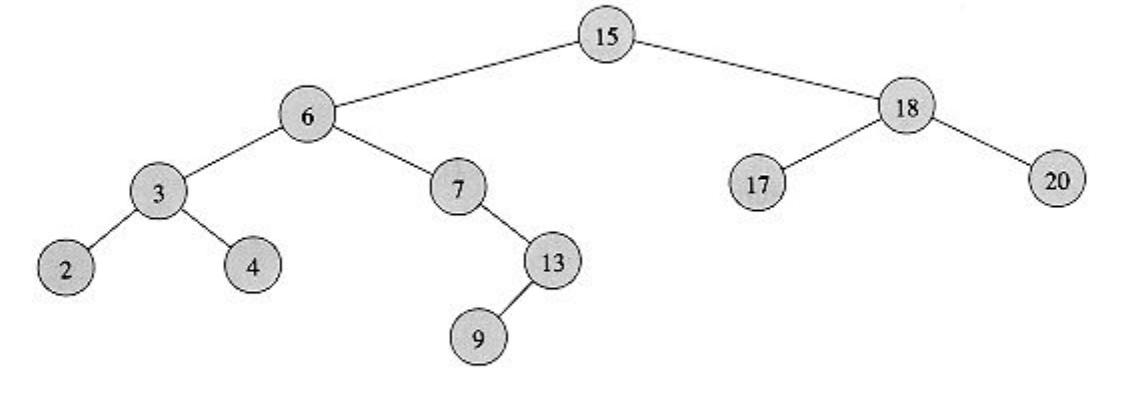
Succesor de 3? → 4

Succesor de 6? → 7

Succesor de 15? → 17

Succesor de 13? → 15

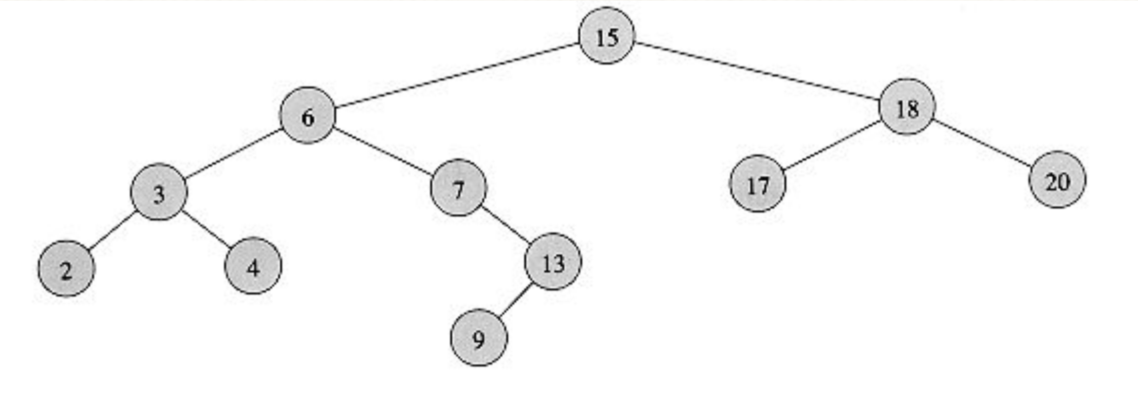
Succesor de 4? → 6



Succesor / Predecesor

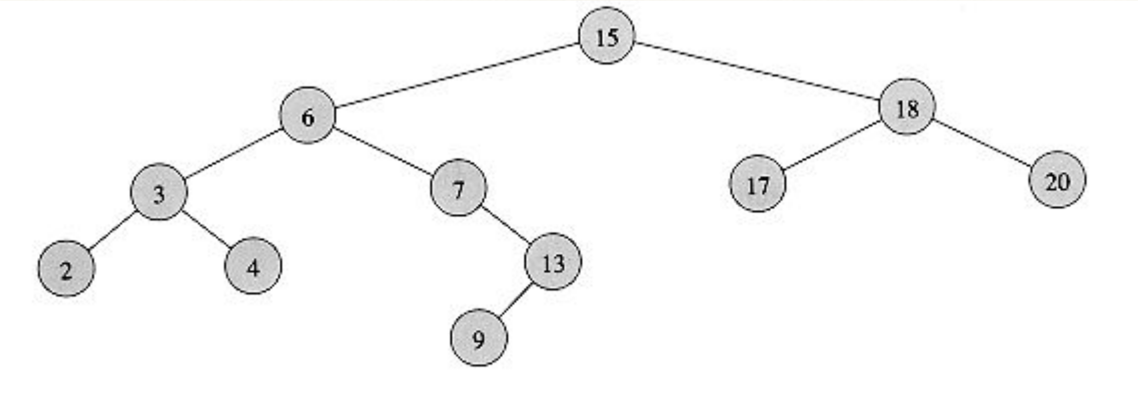
Caz 1) Dacă am fiu drept, atunci cel mai mic element va fi cel mai mic element din subarborele drept. Adică dreapta→stânga→stânga→ … →stânga (vezi 7 sau 15)

Caz 2) Dacă **nu** am fiu drept, atunci va fi primul strămoș al meu în care eu sunt în subarborele stâng al său (vezi 13, 4, 17)



Succesor / Predecesor

```
TREE SUCCESSOR(x)
if right[x] ≠ NIL
    then return TREE-MINIMUM(right[x])
y ← p[x]
while y ≠ NIL and x = right[y]
     do x ← y
        y ← p[y]
return y
```

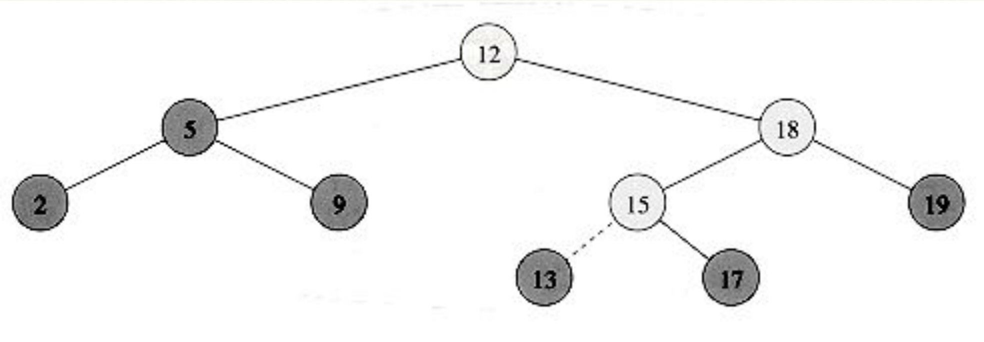


Complexitate: **O(h)**

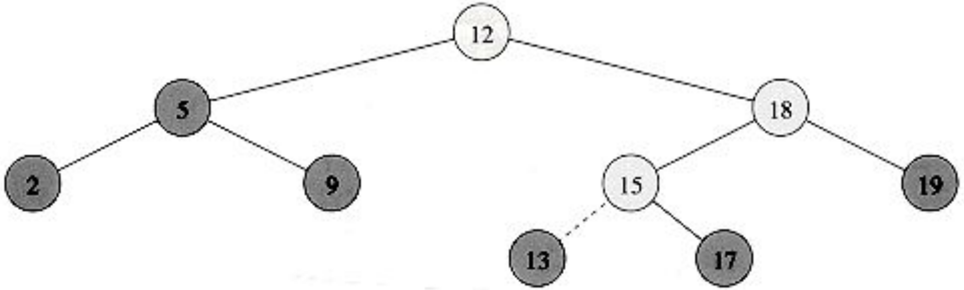
Inserare

Similar cu căutarea.

Inserare 13:



Inserare

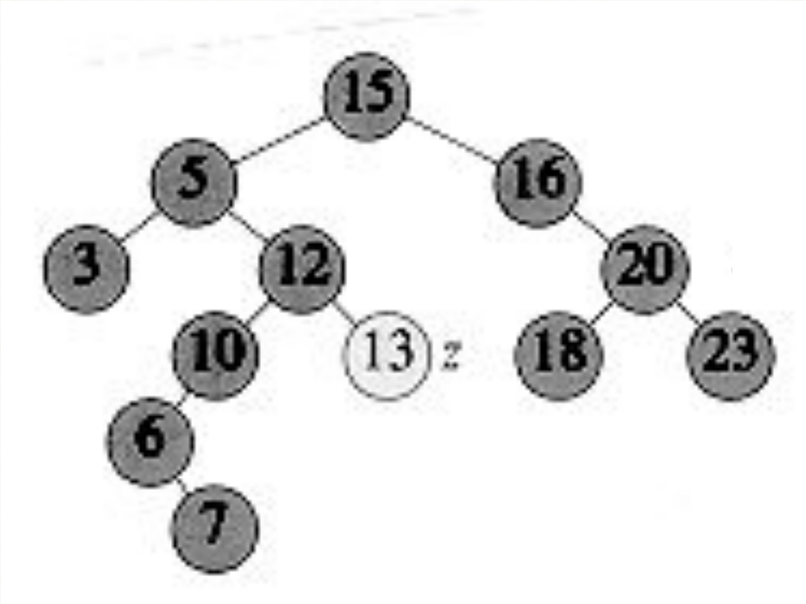


Complexitate: **O(h)**

```
TREE-INSERT(T,z)
y ← NIL
x ← root[T]
while x ≠ NIL
      do y ← x
         if key[z] < key[x]
            then x ← left[x]
            else x ← right[x]
p[z] ← y
if y = NIL
      then root[T] ← z
      else if key[z] < key[y]
              then left[y] ← z
              else right[y] ← z
```

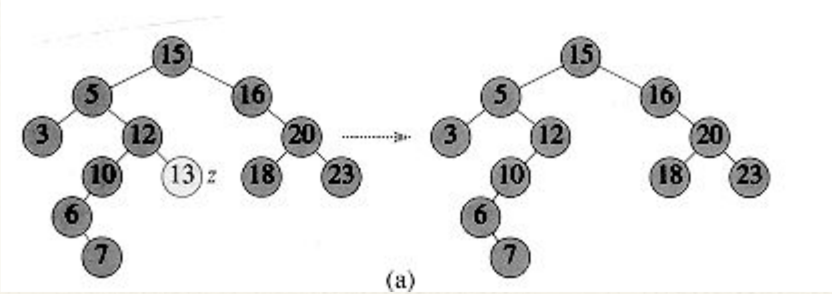
Ștergere

- Cum?
- Cum îl ștergem pe 13?
- Dar pe 7? Pe 16?
- Pe 5?
- Dar pe 15?

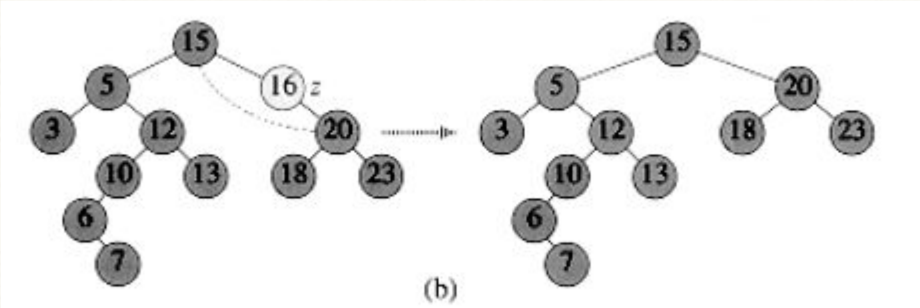


Ștergere

- Cum?
- Cum îl ștergem pe 13?
- Dar pe 7? Pe 16?
- Pe 5?
- Dar pe 15?



(a)



(b)

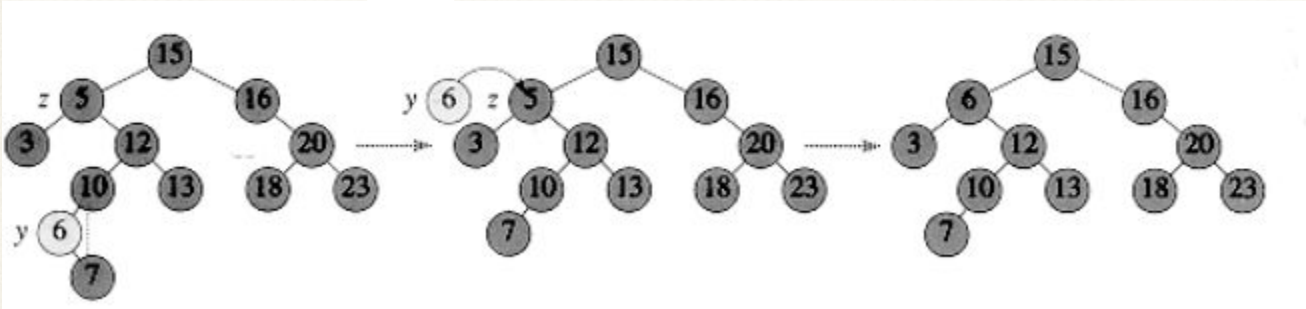
Ștergere

Exercițiu:

- Demonstrați că succesorul unui nod cu 2 fii are maxim un fiu.

Ștergere

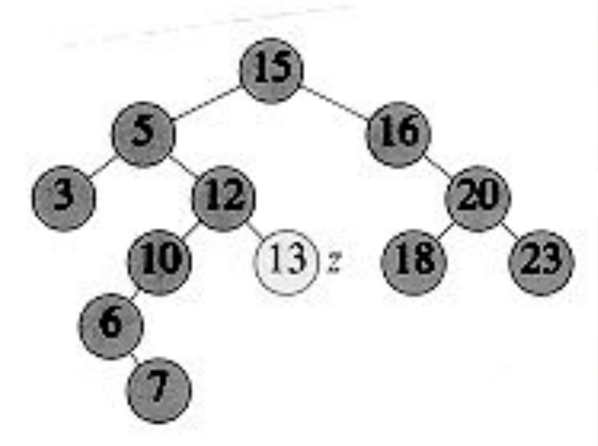
- Cum?
- Cum îl ștergem pe 13?
- Dar pe 7?
- Pe 5?
- Dar pe 15?



Ștergere

Avem 3 cazuri:

1) Dacă nodul **nu are** fii, îl ștergem.
2) Dacă are **un** fiu, îl ștergem și creăm o tată și noul fiu.

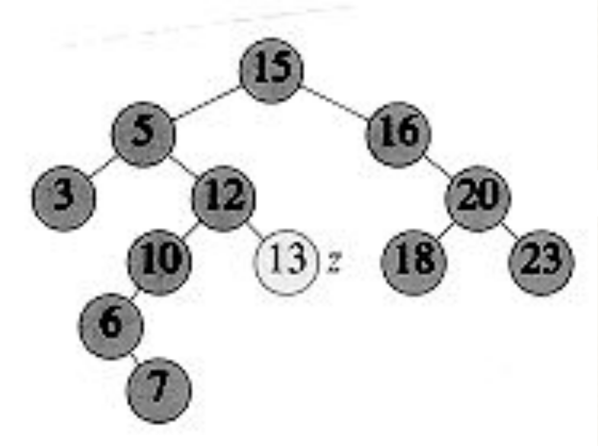



```
TREE-DELETE(T, z)
if left[z] = NIL or right[z] = NIL
    then y ← z
    else y ← TREE-SUCCESSOR(z)
if left[y] ≠ NIL
    then x ← left[y]
    else x ← right[y]
if x ≠ NIL
    then p[x] ← p[y]
if p[y] = NIL
    then root[T] ← x
    else if y = left[p[y]]
            then left[p[y]] ← x
            else right[p[y]] ← x
if y ≠ z
    then key[z] ← key[y]
            ▷ If y has other fields, copy them, too.
return y
```

Ștergere

Avem 3 cazuri:

3) Dacă are **ambii** fii, găsim succesorul său, punem în locul său și înlocuim legătura acestui nod cu singurul fiu (dacă există)




```
TREE-DELETE(T, z)
if left[z] = NIL or right[z] = NIL
    then y ← z
    else y ← TREE-SUCCESSOR(z)
if left[y] ≠ NIL
    then x ← left[y]
    else x ← right[y]
if x ≠ NIL
    then p[x] ← p[y]
if p[y] = NIL
    then root[T] ← x
    else if y = left[p[y]]
            then left[p[y]] ← x
            else right[p[y]] ← x
if y ≠ z
    then key[z] ← key[y]
         ▷ If y has other fields, copy them, too.
return y
```

Complexitate

Operație	Complexitate
Căutare	O(?)
Găsire Minim	O(?)
Inserare	O(?)
Succesor / Predecesor	O(?)
Ștergere	O(?)

Complexitate

Operație	Complexitate
Căutare	O(h)
Găsire Minim	O(h)
Inserare	O(h)
Succesor / Predecesor	O(h)
Ștergere	O(h)

Arbori Binari de Căutare cu Chei Egale

Ce facem dacă avem mai multe chei egale ?

Arbori Binari de Căutare cu Chei Egale

Ce facem dacă avem mai multe chei egale ?

- În caz de egalitate, alegem tot timpul stânga sau dreapta și inserăm în aceeași direcție
- Ținem o listă cu toate elementele egale într-un singur nod (sau un contor care să numere aparițiile, dacă nu avem alte informații)

Arbori Binari Echilibrați

- AVL
- Arbori Roșu-Negri
- Treap-uri
- Splay Trees
- B-arbori

- Skip Lists (nu sunt arbori binari de căutare, dar…)

Search trees (dynamic sets/associative arrays)	2–3 · 2–3–4 · AA · (a,b) · AVL · B · B+ · B* · B^x · (Optimal) Binary search · Dancing · HTree · Interval · Order statistic · (Left-leaning) Red-black · Scapegoat · Splay · T · **Treap** · UB · Weight-balanced

Bibliografie

[Introducere în Algoritmi Cormen Leiserson Rivest]