Arbori

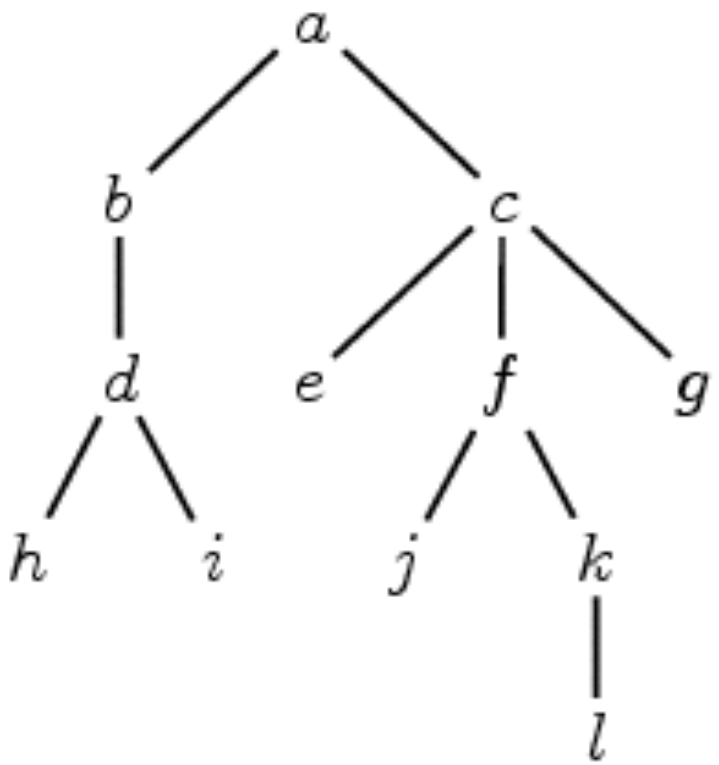
Definitie. Arbori oarecare. Parcurgeri. Arbori cu etichete si arbori pentru expresii. Arbore ADT. Implementari ale arborilor. Arbori binari de cautare

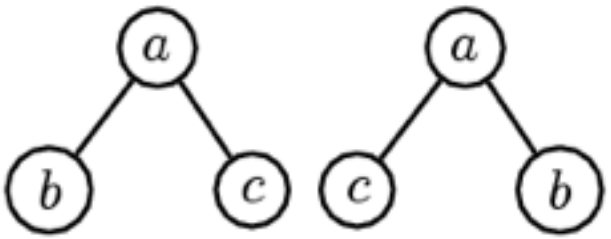
Arbori

- Arbori oarecare: colectie de elemente numite noduri - avand un nod radacina si o relatie de paternitate intre noduri – fapt ce impune o ***structura ierarhica*** a nodurilor.
- Definitie: *structura recursiva*, colectie ierarhica de noduri, fiecare nod:
  - fie este gol
  - fie este o structura care contine o cheie si o colectie de referinte catre noduri *copil,* radacinile $n_1, n_2, \ldots, n_k$ ale sub-arborilor $T_1, T_2, \ldots, T_k$
- Structura de date pentru colectii non-liniare

Terminologia folosita pentru arbori

- stramosi, descendenti, parinti, copii
- *Frunze* (noduri fara copii, noduri terminale)
- *Noduri interne* (noduri cu copii)
  - “radacina” este un nod intern
- Cale (drum)
- Sub-arbori
- Ordinea nodurilor conteaza, frati





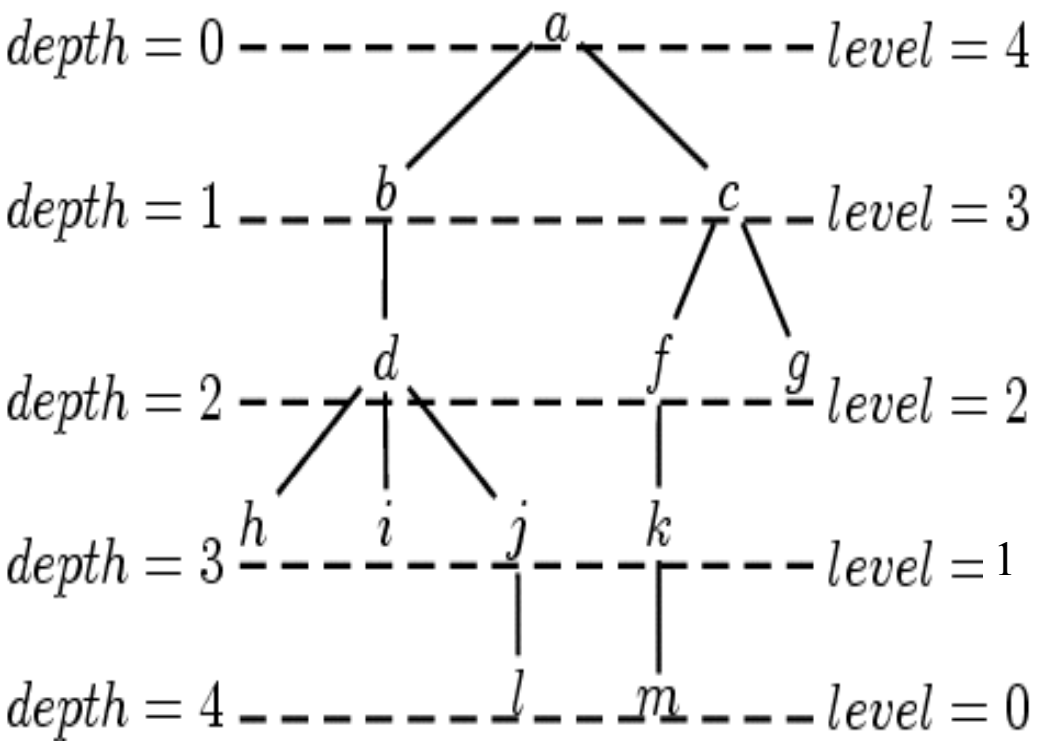
Terminologia pentru arbori

Pentru un arbore oarecare T = (V, E) cu radacina r $\in$ V:

- ***Cale*** (drum): $\langle n_1, n_2, \ldots, n_k \rangle$ astfel incat n_i = parintele lui n_{i+1} pentru $1 \leq i \leq k$. $lungime(cale) = nb\_noduri - 1$
- ***Adancimea unui nod*** (varf) $v \in V$ este $adancime(v)$ = lungimea drumului de la r la v
- ***Inaltimea unui varf*** $v \in V$ este $inaltime(v)$ = lungimea celui mai lung drum de la v la o frunza
- Inaltimea unui arbore *T* este $inaltime(T) = inaltime(r)$
- ***Nivelul*** unui varf $v \in V$ este $nivel(v) = inaltime(T) - adancime(v)$
- ***Diametrul*** unui arbore: lungimea maxima a unei cai intre 2 frunze, in arbore
- ***Sub-arborele*** generat de un varf $v \in V$ este un arbore care consta in nodul radacina v si toti descendentii sai din *T*.

Terminologia pentru arbori - Exemple

- Pentru arborele din imagine:
  - Radacina este *a*.
  - Frunze: *h*, *g*, *i*, *l*, *m*.
  - Stramosii lui *k sunt a*, *c*, *f*.
  - Descendentii lui *d* sunt *h, i, j, l*.
  - Parintele lui *h* este *d*.
  - Copii /fii lui *c* sunt *f*, *g*.
  - Fratii lui *h* sunt *i*, *j*.
  - Inaltime(*T*)=inaltime(*a*) =4



Terminologia pentru arbori oarecare

- Un arbore oarecare este:
  - ***m-ary*** daca fiecare varf intern are cel mult *m* fii.
    - *m* = 2 → **arbore binar**; *m* = 3 → arbore ternar
  - ***m-ary intreg (“full”)*** – fiecare nod intern are exact m fii
  - ***complet m-ary*** – daca este arbore full si toate frunzele sunt la nivelul 0
- Limite:
  - Inaltimea maxima a unui arbore cu `n` varfuri este $n - 1$.
  - Inaltimea maxima a unui arbore binar plin (full) cu `n` varfuri este $(n - 1)/2$
  - Inaltimea minima a unui arbore binary cu `n` varfuri este $\lfloor \log_2 n \rfloor$

Parcurgerile unui arbore

Preordine – se viziteaza radacina, apoi tot in preordine se viziteaza nodurile subarborilor care au ca parinte radacina, incepand cu sub-arborele cel mai din stanga.

`Preordine (`*`n`*`)`

- proceseaza *n*
- pentru fiecare fiu `c` al lui *n*, in ordine de la cel mai din stanga fiu executa `Preordine(`*`c`*`)`

http://algoviz.org/OpenDSA/Books/Everything/html/GenTreeIntro.html

Parcurgerile unui arbore

Inordine – se viziteaza in inordine primul copil, dupa care se proceseaza radacina, dupa care se viziteaza, in inordine, restul copiilor

```
Inordine (n)
```

- `Inordine(fiul cel mai din stanga a lui n)`
- proceseaza n
- pentru fiecare fiu c al lui n, exceptie facand nodul cel mai din stanga, in ordine de la stanga la dreapta se executa `Inordine(c)`

Parcurgerile unui arbore

Postordine– pentru un nod se viziteaza in postordine toti sub-arborii care au ca radacini pe fii nodului dat, apoi se viziteaza nodul. Se incepe parcurgerea de la radacina.

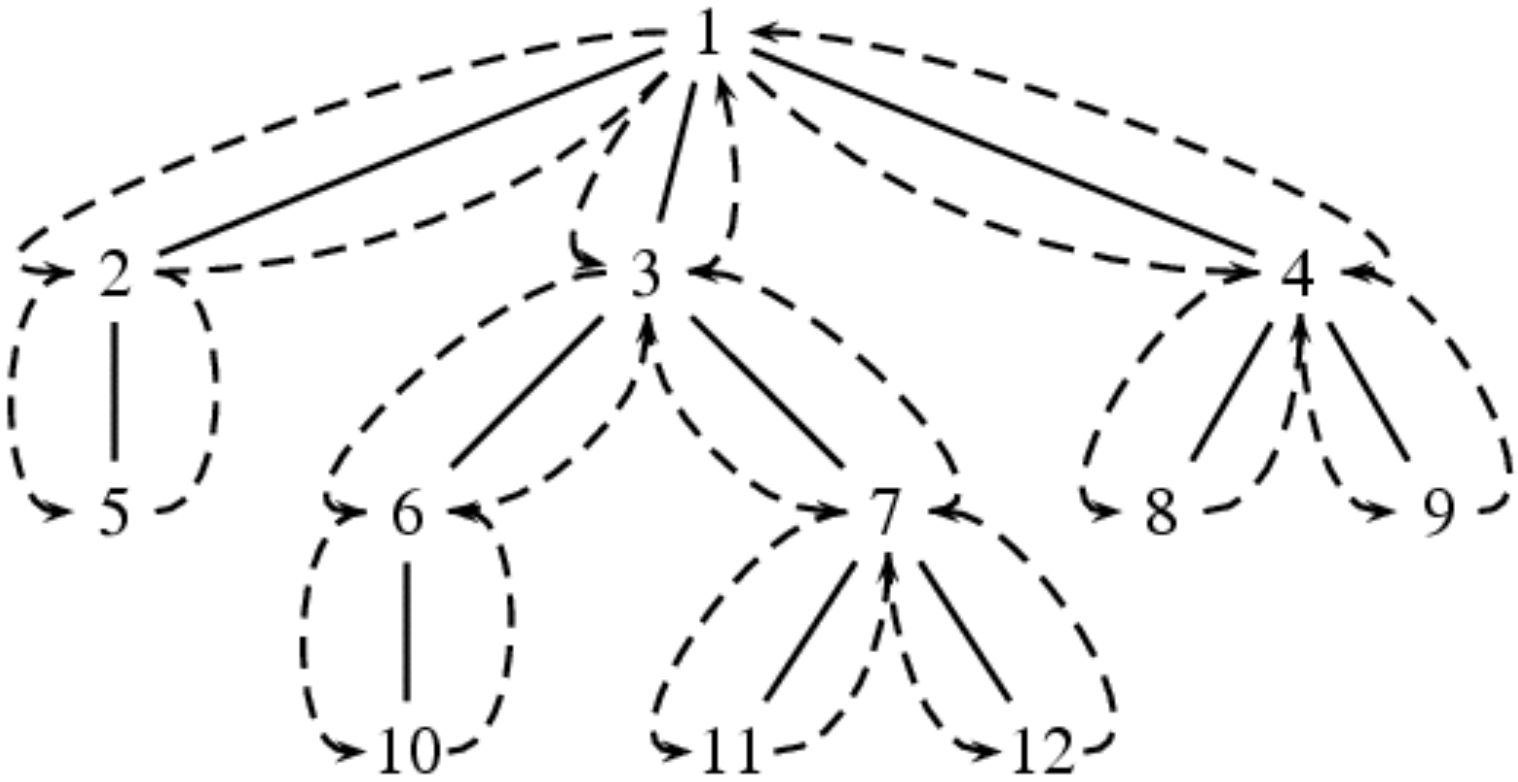
```
Postordine(n)
```

- pentru fiecare fiu `c` al lui `n`, executa `Postordine(c)`
- proceseaza `n`

Informatia despre stramosi

- Parcurgerile in *preordine* si in *postordine* sunt utile pentru a obtine informatia despre stramosi.
- Sa presupunem ca:
  - `post(n)` este pozitia unui nod *n* la o parcurgere in postordinea nodurilor unui arbore si
  - `desc(n)` este numarul de stramosi propri ai lui *n.*
  - Atunci nodurile din subarborele care are ca radacina pe `n` sunt numerotate consecutiv de la `post(n)-desc(n)` pana la `post(n)`
- Pentru a verifica daca un nod *x* este un descendent al unui nod *y*: $post(y)-desc(y) \leq post(x) \leq post(y)$
- **Exercitiu**: Verificati relatia pentru preordine !

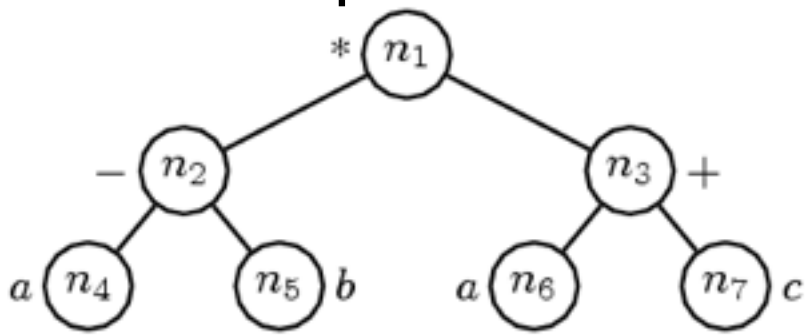
Exemplu de parcurgeri



- preordine: 1, 2, 5, 3, 6, 10, 7, 11, 12, 4, 8, 9.
- postordine: 5, 2, 10, 6, 11, 12, 7, 3, 8, 9, 4, 1.
- inordine: 5, 2, 1, 10, 6, 3, 11, 7, 12, 8, 4, 9.

Arbori etichetati si arbori pentru expresii

- Arborii binari se pot folosi pentru a reprezenta expresii precum:
  - Propozitii compuse
  - Combinatii de multimi
  - Expresii aritmetice

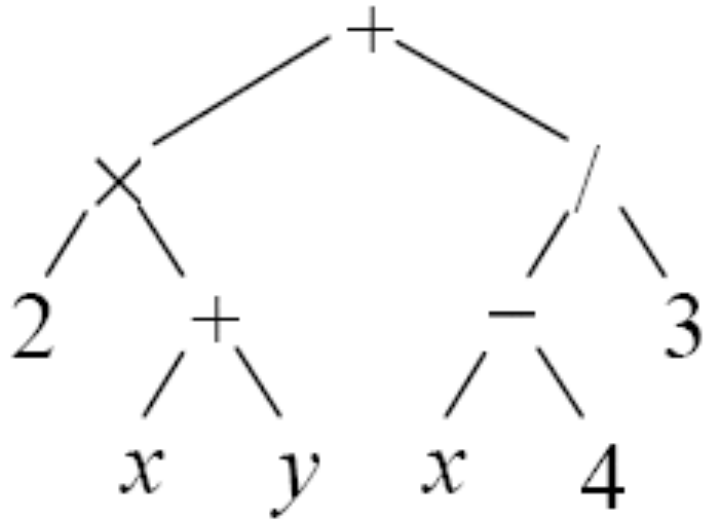


- **Arbore etichetat*:*** fiecare nod are asociata o eticheta sau o valoare
- Arbore pentru expresii aritmetice: nodurile interne reprezinta operatori si frunzele sunt operanzi
  - Operator binar: primul operand este pe frunza stanga iar al doilea operand este pe frunza dreapta
  - Operatori unari: un singul operand pe frunza dreapta

Forme prefix, postfix, infix

- Folosind arborii binari se pot obtine expresii aritmetice in trei reprezentari:
  - Forma infixata:
    - Parcurgere in inordine
    - Se folosesc parantezele pentru a evita ambiguitatile
  - Forma prefixata:
    - Parcurgere in preordine
    - Nu sunt necesare parantezele
  - Forma postfixa:
    - Se foloseste parcurgerea in postordine
    - Nu sunt necesare parantezele
- Expresiile in forma prefixata si postfixa sunt folosite in stiinta calculatoarelor.

Exemplu de arbore pentru expresii:



infix: $(2\times(x+y))+((x-4)/3)$

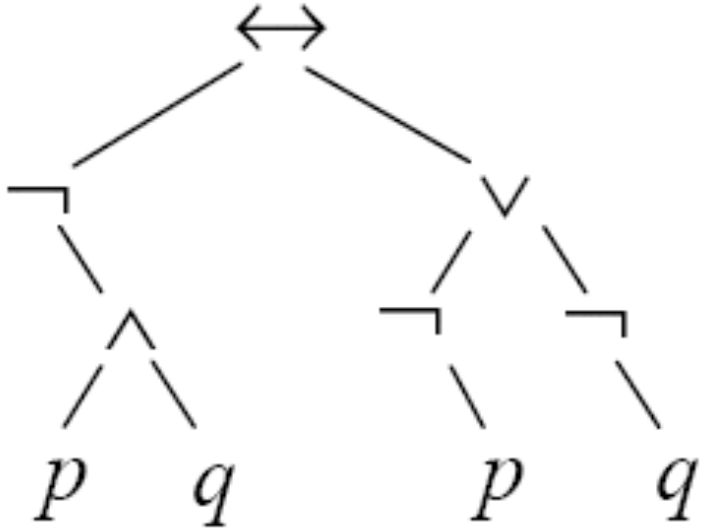
prefix: $+\times 2+x\ y\ /-x\ 4\ 3$

postfix: $2\ x\ y+\times x\ 4-3\ /+$

infix: $(\neg(p\wedge q))\leftrightarrow(\neg p\vee\neg q)$

prefix: $\leftrightarrow\neg\wedge p\ q\vee\neg p\neg q$

postfix: $p\ q\wedge\neg p\neg q\neg\vee\leftrightarrow$



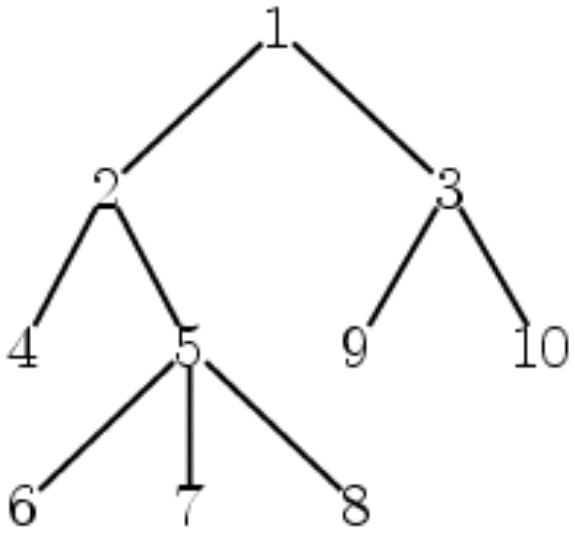
ADT (Abstract Data Type) Tree

- `parent(n, T)`: returneaza parintele nodului n in arborele T. Pentru radacina returneaza un arbore vid (null) notat Λ.
  - Input: nod, arbore; Output: nod sau Λ
- `leftmostChild(n, T)`: returneaza fiul cel mai din stanga al nodului n din arborele T sau Λ pentru o frunza.
  - Input: nod, arbore; Output: nod sau Λ
- `rightSibling(n, T)`: returneaza fratele din dreapta al nodului n in arborele T sau Λ pentru cel mai din dreapta frate.
  - Input: nod, arbore; Output: nod sau Λ
- `label(n, T)`: returneaza eticheta (valoarea asociata) nodului n in arborele T
  - Input: nod, arbore; Output: eticheta
- `root(T)`: returneaza radacina arborelui *T*
  - Input: arbore; Output: nod sau Λ
- *`inord(T), preord(T), postord(T)`*
- etc…

Implementarea arborilor cu vectori

- Exemplu:
  - fiecare nod are referinta catre indexul nodului parinte
  - stocat in vector

Model logic



(a)

Structura fizica

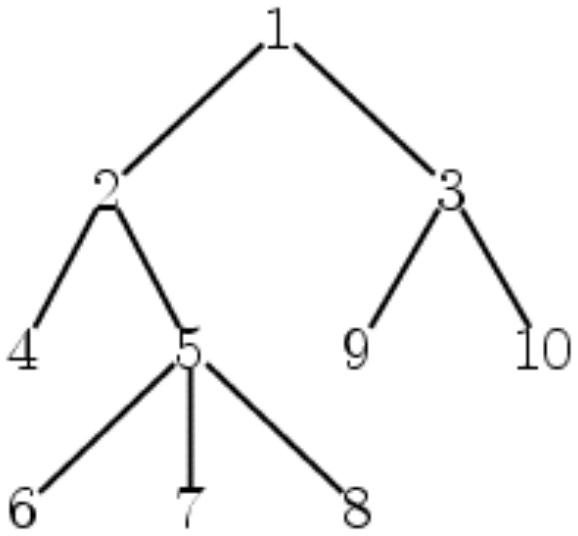
1	2	3	4	5	6	7	8	9	10
0	1	1	2	2	5	5	5	3	3

(b)

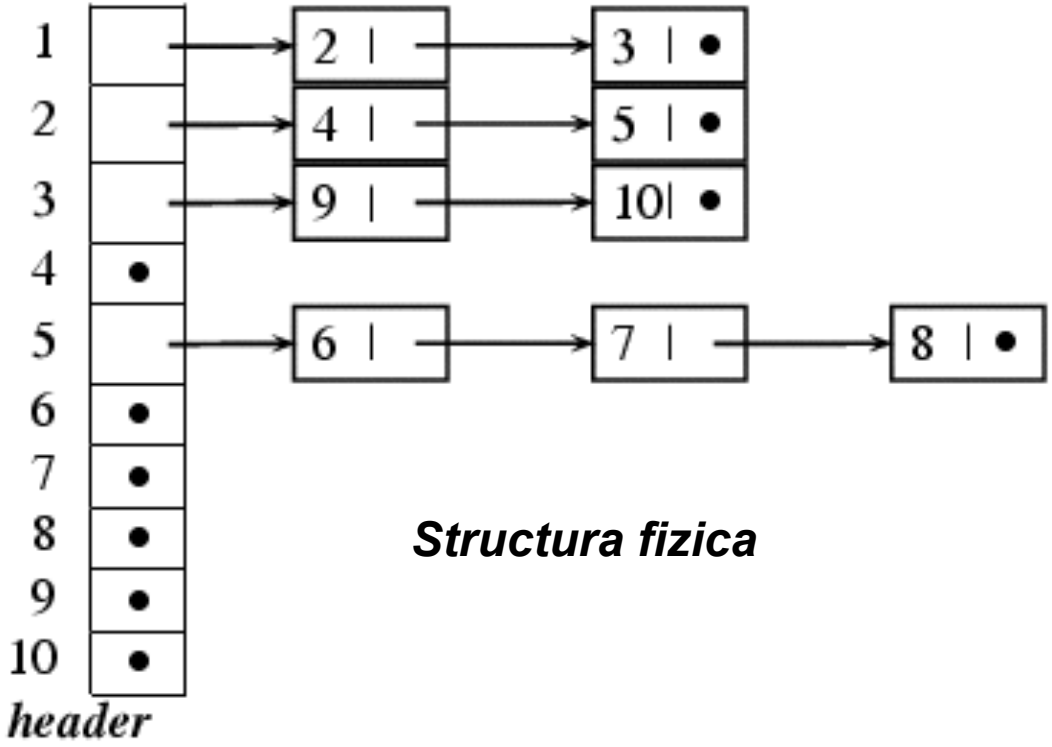
(a) Tree (b) Data structure

Implementarea arborilor. Liste de fii

Model logic



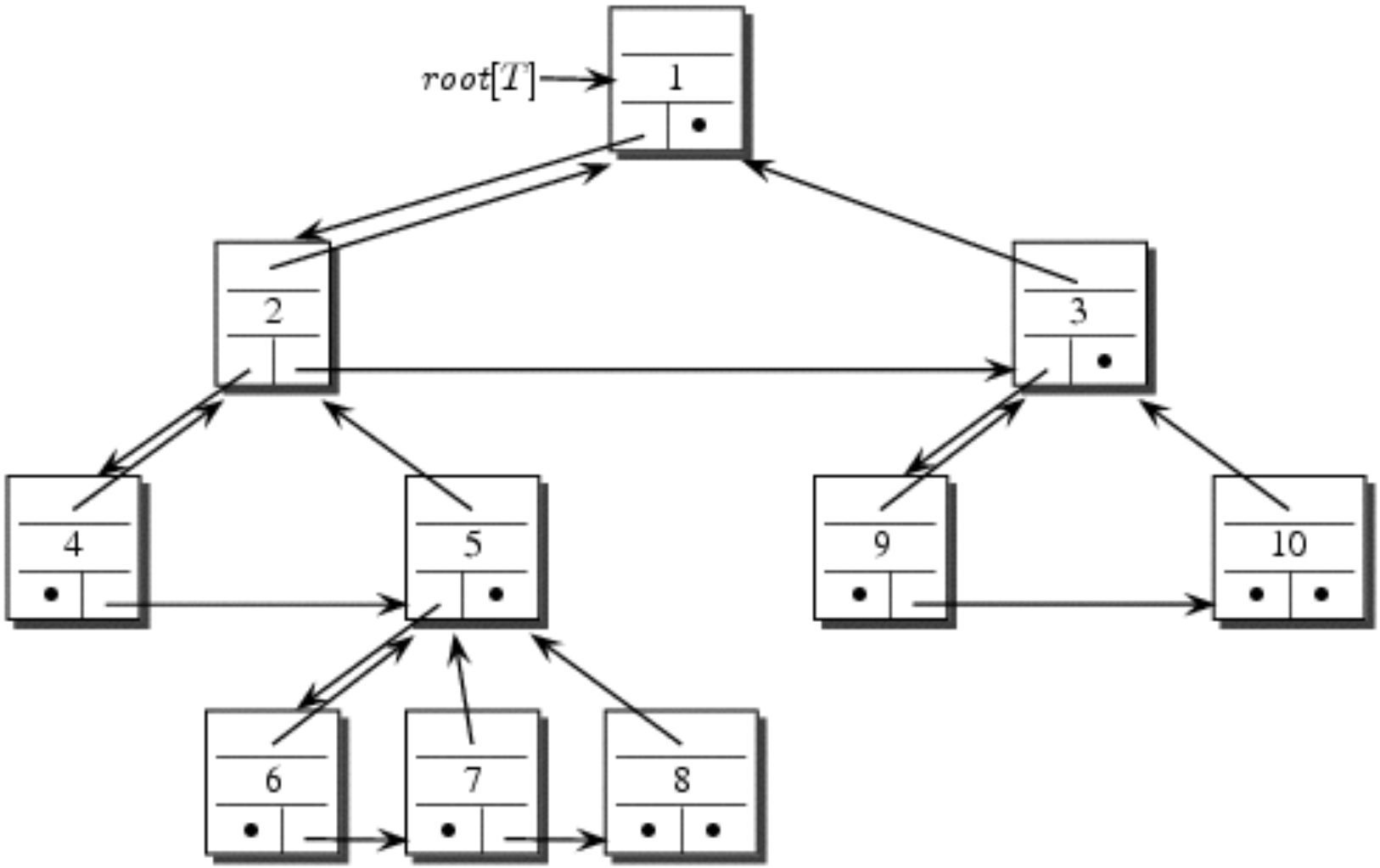
(a) Tree



Structura fizica

(b) Data structure

Reprezentare de arbore binar a arborilor multicai

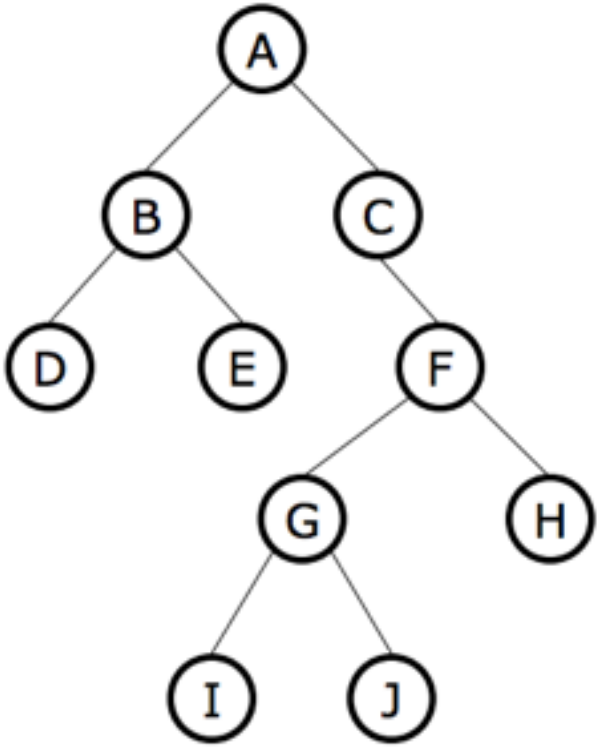


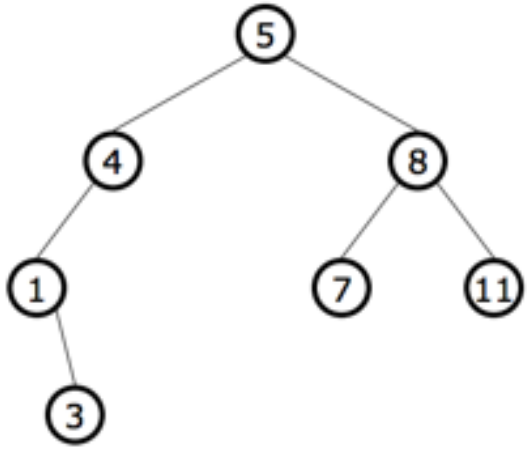
Arbore binar

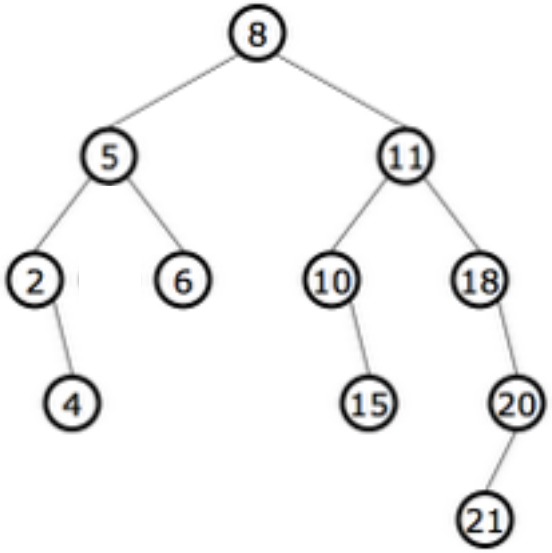
- **Structura recursiva:**
  - *nil* (NULL)
  - nod, denumit radacina, impreuna cu doi arbori binari - subarborele stang (*left*) si subarborele drept (*right*)
- **Reprezentare inlantuita:**
  - campurile cheie, *left (stang)*, *right(drept)*, optional si *p(parinte)*

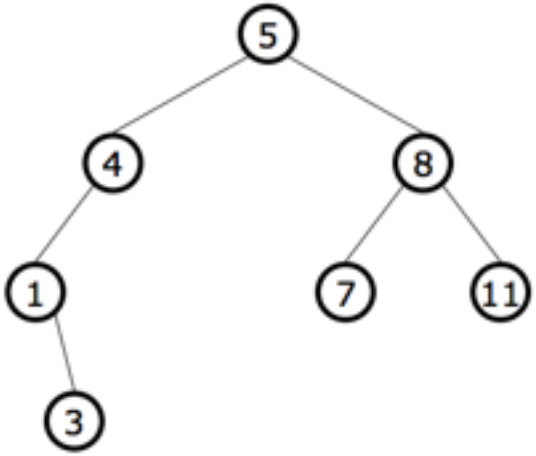
Arbori binari de cautare

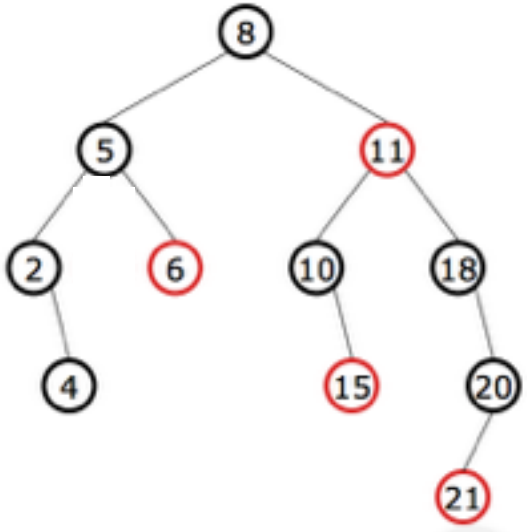
- **Definitie:**
  - arbore binar, chei care pot fi comparate
  - nodurile cu chei mai mici decat valoarea *x* a cheii asociate unui anumit nod se gasesc in subarborele stang al acestuia
  - nodurile ale caror chei au valori mai mari decat *x* se gasesc in subarborele sau drept
- **Operatii:**
  - *search (key, root)*
  - *insert(key, root)*
  - *delete(node, root)*
  - *traverse (inorder, preorder, portorder)*
  - *height(root), diameter(root), successor(node), predecessor(node),* etc…




5
4
8
1
7
11
3

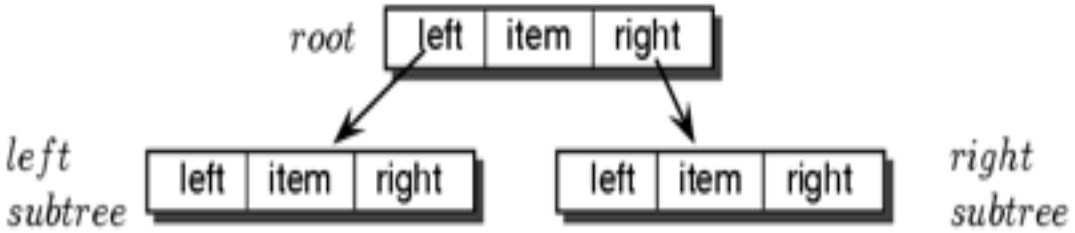

8
5
11
2
6
10
18
4
15
20
21


5
4
8
1
7
11
3


8
5
11
2
6
10
18
4
15
20
21

Implementare arbore binar de cautare

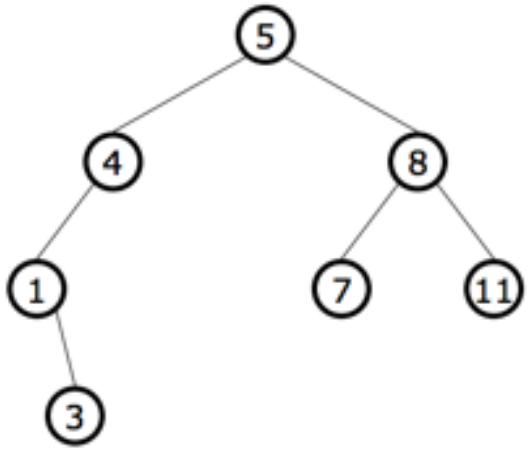
```
typedef struct t_node
{
    void *item;
     struct t_node *left;
    struct t_node *right;
     //struct t_node *parent; //optional
} NodeT;
```



Exemplu pseudocod: preordine

Varianta recursiva

```
preorder(node)
  if (node = null)
    return
  visit(node)
  preorder(node.left)
  preorder(node.right)
```



Varianta iterativa

```
iterativePreorder(node)
s ← empty stack
while (not s.isEmpty() or node ≠ null)
  if (node ≠ null)
    visit(node)
    if (node.right ≠ null)
      s.push(node.right)
    node ← node.left
  else
    node ← s.pop()
```

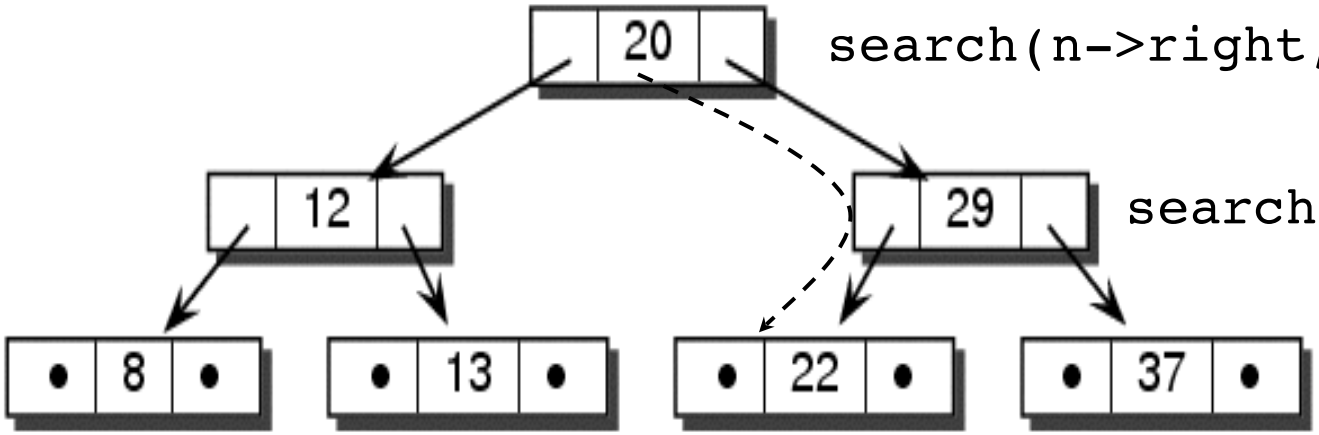
? Care sunt secventele generate de parcurgerile in:
- preordine
- inordine
- postordine

? Care este complexitatea unei parcurgeri?

Functia de cautare

- E.g.
  - key = 22;

```
if ( search( root , &key ) )…
```




```
return n->item;
```

Functia de cautare - varianta recursiva

```
extern int KeyCmp( void *a, void *b );
/* Returns -1, 0, 1 for keys stored with a < b, a == b, a
   > b */

void *search( NodeT *t, void *key ) {
   if ( t == (Node)0 ) return NULL;
   switch( KeyCmp( key, ItemKey(t->item) ) ) {
      case -1 : return FindInTree( t->left, key );
      case 0:   return t->item;
      case +1 : return FindInTree( t->right, key );
      }
   }
```

Less, search left

Greater, search right

Functia de cautare - varianta iterativa

```
void *search( NodeT *t, void *key ) {
  while(t !=(Node)0 && KeyCmp(key,ItemKey(t->item)!=0)
  {
     if(KeyCmp(key,ItemKey(t->item)<0)
          t = t->left;
     else
          t = t->right;
   }
    if(t ==(Node)0)
       return NULL;
    return
       t->item;
```

Performanta functiei de cautare

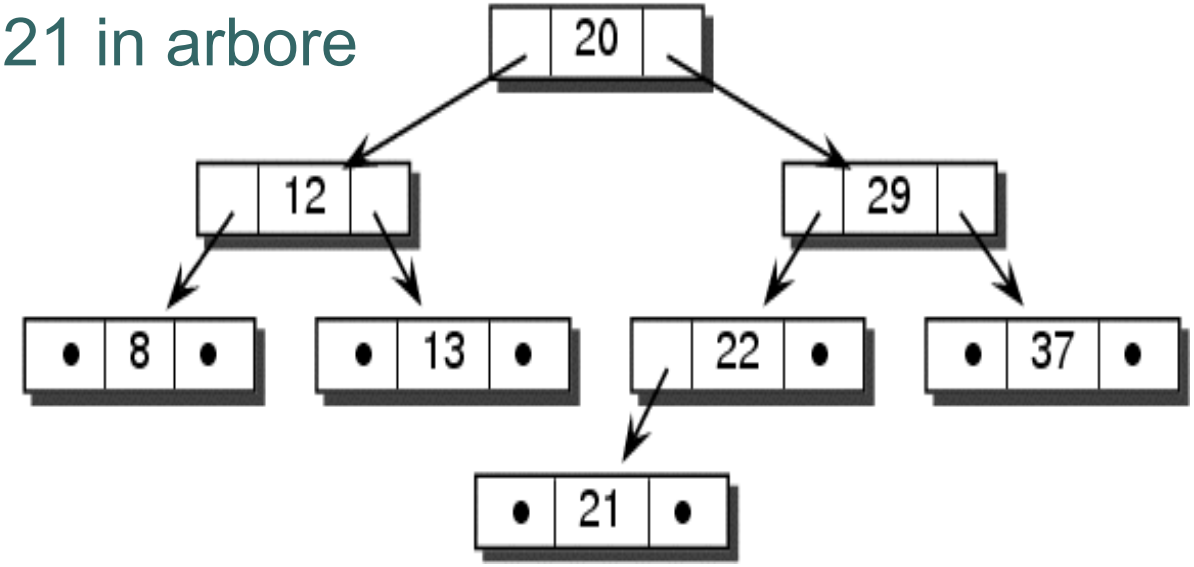
- Inaltime h
  - Noduri parcurse pe un drum de la radacina la o frunza
  - Avem nevoie de cel mult $h+1$ comparatii, deci $O(h)$
- caz mediu:
  - pp. arborele aproximativ echilibrat: $O(\log n)$
- cazul defavorabil?

Alte operatii de cautare

- cautarea nodului minim
- cautarea nodului maxim
- cautarea predecesorului unui nod internn
- cautarea predecesorului unui nod intern
- cautarea predecesorului unei frunze
- cautarea succesorului unei frunze

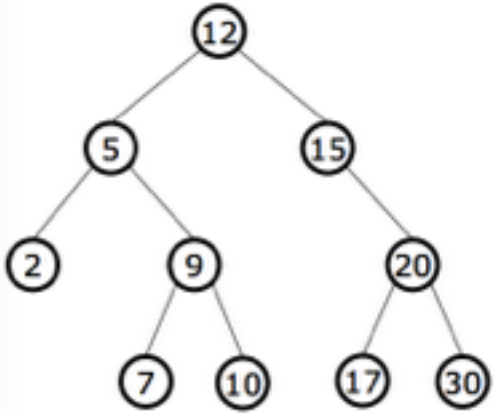
Inserarea unui nod

- **Intotdeauna ca frunza** !!!
- Se insereaza nodul 21 in arbore

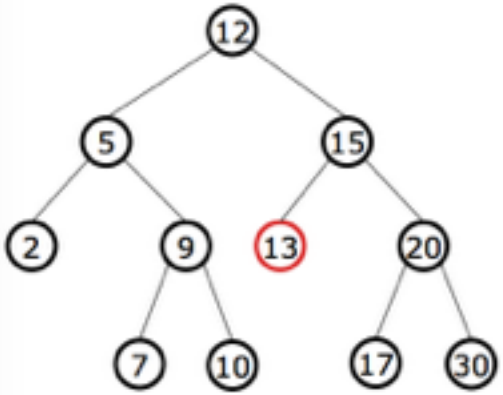


- Avem nevoie de cel mult $h+1$ comparatii
- Crearea unui nod – timp constant
- Inserarea se face in $c_1(h+1)+c_2$

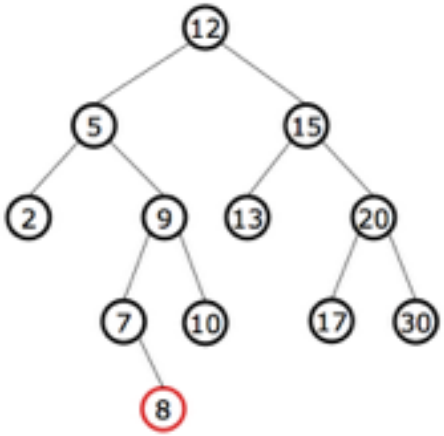
Exemplu Inserare



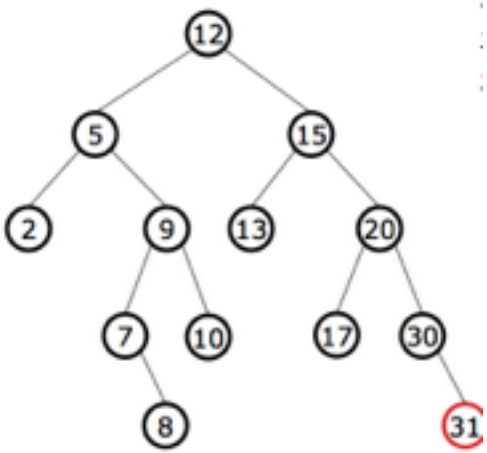
insert(13)



insert(8)



insert(31)



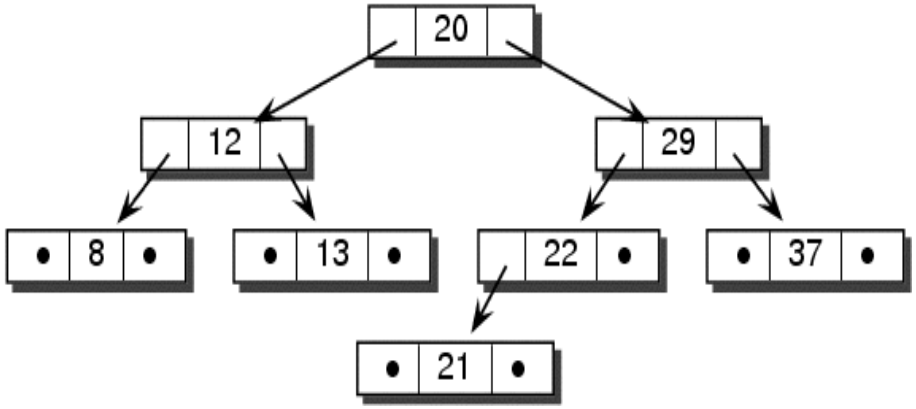
Implementarea functiei de inserare

```
static void insert(NodeT **t, NodeT *new)
{
   NodeT *current = *t;
   /* If it's a null tree, just add it here */
   if (current == NULL)
    {
        *t = new;
        return;
     }
   else
      if(KeyLess(ItemKey(new->item),ItemKey(base->item)))
         AddToTree(&(current->left), new);
      else
         AddToTree(&(current->right), new);
   }
```

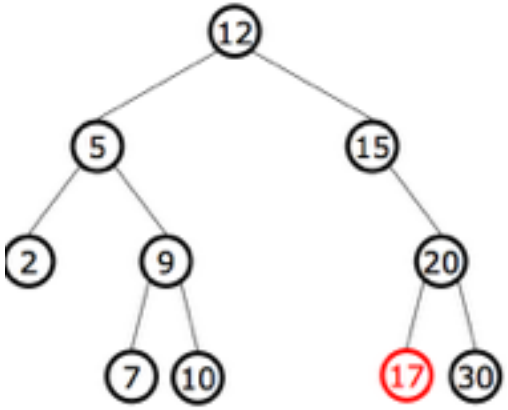
Codul aproape identic cu codul de cautare!
Complexitate?

Stergerea unui nod

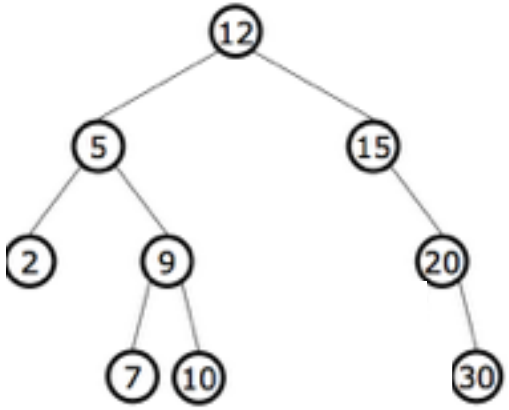
- Mai dificila decat inserarea!
- Idee:
  - se cauta nodul de sters
  - se elimina
  - se reface propr. de ABC
- Cazuri pentru stergere:
  1. Nod terminal (frunza)
  2. Nod cu un singur fiu
  3. Nod cu doi fii
- Exemplu:
  1. Stergeti 8, 13, 21 sau 37
  2. Stergeti 22
  3. Stergeti12, 20 or 29

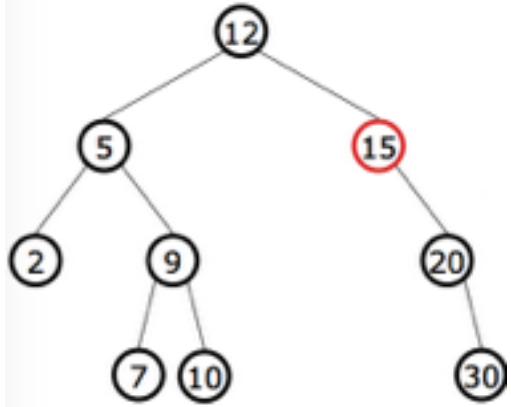


Exemplu stergere: cazuri 1&2

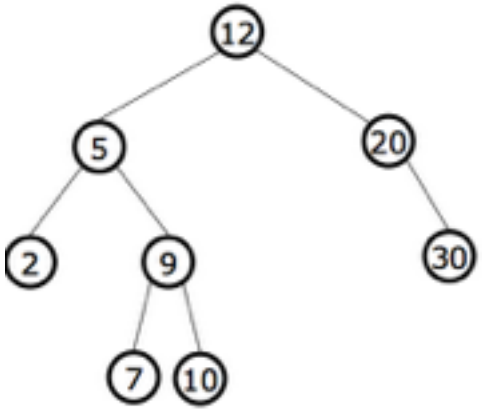


delete(17)

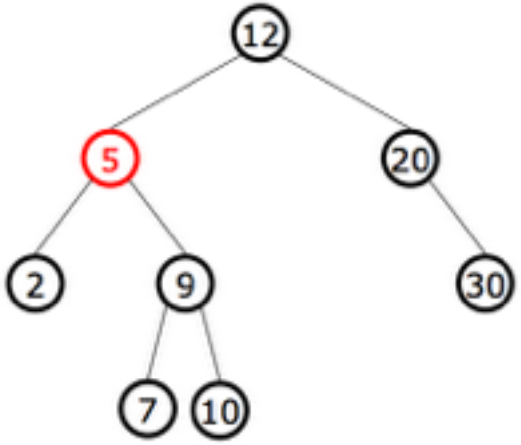




delete(15)

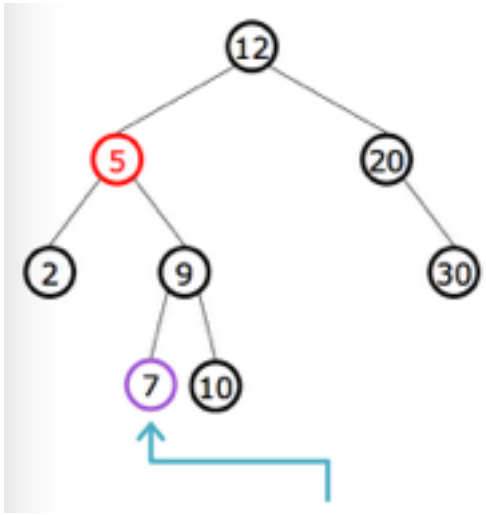


Exemplu stergere: cazul 3

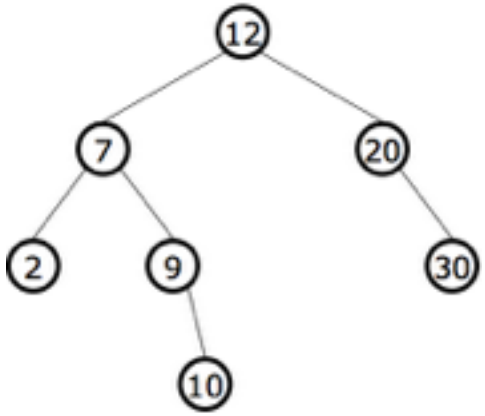


Cum putem pastra proprietatea de ABC?

Inlocuim cu o valoare intre cei 2 copii!
- succesorul: findMin(node->right)
- predecesorul: findMax(node->left)



succesor



Implementarea functiei de stergere

```
BSTDELETE(T, z)
    ▷ Input: z: node; T: tree
    ▷ Output: nothing
1   if z is a leaf                    ▷ (case 0)
2       then remove z
3   if z has one child                ▷ (case 1)
4       then make p[z] point to the child
5   if z has two children             ▷ (case 2)
6       then swap z with its successor
7            perform case 0 or case 1 to delete it
```

Implementarea functiilor de succesor/findMin

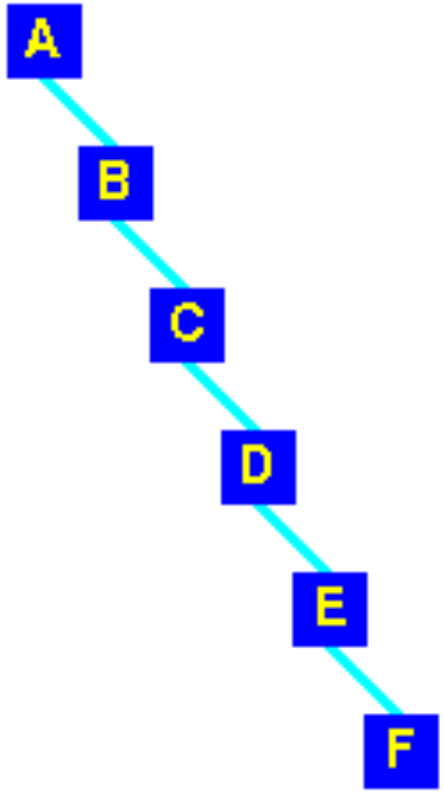
```
BSTMINIMUM(x, k)
    ▷ Input: x: node; k: key to find
    ▷ Output: node or NIL(x, k)
1   while left[x] ≠ NIL
2       do x ← left[x]
3   return x
```

```
BSTSUCCESSOR(x, k)
    ▷ Input: x: node; k: key to find
    ▷ Output: node of minimum key
1   if right[x] ≠ NIL
2      then return BSTMINIMUM(right[x])
3   y ← p[x] ▷ p[x] is parent of node x
4   while y ≠ NIL ∧ x = right[y]
5       do x ← y
6          y ← p[y]
7   return y
```

Performanta operatiilor

- Cautare $c\ \mathrm{h}$
- Inserare $c\ \mathrm{h}$
- Stergere $c\ \mathrm{h}$
- $h = log\ n$? (cazul mediu, da)
- Aparent eficient!
- Sa se construiasca un arbore cu caracterele:

  A B C D E F
- Ne-echilibrat - cazul defavorabil O(n)



Compararea performantei

	Arrays Simple, fast Inflexible	Linked List Simple Flexible	Trees Still Simple Flexible
Add	O(1) O(n) *inc sort*	O(1) *sort -> no adv*	
Delete	O(n)	O(1) - *any* O(n) - *specific*	
Find	O(n) O(logn) *binary search*	O(n) *(no bin search)*	O(log n)

Performanta operatiilor

- Cum putem obtine garantia h ~ log n
  - constructia initiala
    - cheile ordonate (crescator, descrescator) ?
    - mediane?
    - inserari/stergeri ulterioare nu garanteaza mentinerea proprietatii
  - noduri inserate in ordine aleatoare
    - conditie de echilibru, care
      - asigura inaltimea e O(log n)
      - usor de intretinut la inserari/stergeri
- in curand….

Referinte

- AHU, chapter 3
- CLR, chapters 11.3, 11.4
- CLRS, chapter 10.4, 12
- Preiss, chapter: Trees.
- Knuth, vol. 1, 2.3
- Notes