# Curs 4
## Liste. Stive. Cozi

### Liste Liniare (LLin)

<u>LLin – operatii</u>

- insereaza() – exemple
  - L = (a, b, c, d, e, f , g)
  - insereaza(L, 0, x) ⇒ L = (x, a, b, c, d, e, f , g)
    Obs. indexul elementelor a, · · · , g creste cu 1.
  - insereaza(L, 2, x) ⇒ L = (a, b, x, c, d, e, f , g)
  - insereaza(L, 7, x) ⇒ L = (a, b, c, d, e, f , g, x)
  - insereaza(L, 10, x) ⇒ eroare
- elimina()
  - L = (a, b, c, d, e, f , g)
  - elimina(L, 2) ⇒ L = (a, b, d, e, f , g)
    Obs. indexul elementelor a, · · · , g descreste cu 1.
  - elimina(L, 10) ⇒ eroare
- alKlea()
  - L = (a, b, c, d, e, f , g)
  - alKlea(L, 0) ⇒ a
  - alKlea(L, 2) ⇒ c
  - alKlea(L, -2) ⇒ eroare
- elimTotE()
  - L = (a, b, c, a, b, c, a)
  - elimTotE(L, a) ⇒ (b, c, b, c)
  - elimTotE(L, c) ⇒ (a, b, a, b, a)
- parcurge()
  - L = (1, 2, 3, 1, 2, 3)
  - parcurge(L, oriDoi()) ⇒ (2, 4, 6, 2, 4, 6)
  - parcurge(L, incrementeaza()) ⇒ (2, 3, 4, 2, 3, 4)
- poz()
  - L = (a, b, c, a, b, c, d)
  - poz(L, a) ⇒ 0
  - poz(L, c) ⇒ 2
  - poz(L, x) ⇒ -1
- lung()
  - L = (a, b, c, a, b, c, d)
  - lung(L) ⇒ 7

### LLin – implementare cu tablouri

Vectori a[i] in mod obisnuit.

### LLin – implementare cu structuri inlantuite (liste simplu inlan)

-

### Liste Liniare Ordonate (LLinOrd)

- elimina()
  - intrare: L = ($e_0$, . . . , $e_{n-1}$), e ∈ Elt
  - iesire: L = (· · · , $e_{k-1}$, $e_{k+1}$, · · ·), daca e = $e_k$ eroare in caz contrar
- alKlea()
- parcurge()
- poz()

<u>LLinOrd – implementarea cu tablouri</u>

```
function poz(L, e)
begin
      p ← 0; q ← L.ultim
      m ← (p + q)/2
      while (L.tab[m]! = e and p < q) do {
            if (e < L.tab[m]) then
                  q ← m-1
            else
            {
                  p ← m + 1
                  m ← (p + q)/2
                  }
      }
      if (L.tab[m] == e) then
            return m
      else
            return -1
end
```

LLinOrd – complexitatea cautarii

- Implementarea cu tablouri: O(log2 n);
- Implementarea cu liste inlantuite: O(n);

## Stiva

Obiecte:

Liste in care se cunoaste vechimea elementelor introduse: liste LIFO

Stiva – operatii

- stivaVida()
  - intrare: nimic
  - iesire: S = () (lista vida)
- esteVida()
  - intrare: S ∈ Stiva
  - iesire: true daca S este vida – false daca S nu e vida
- push()
  - intrare: S ∈ Stiva, e ∈ Elt
  - iesire: S la care s-a adaugat e ca ultim element introdus (cel cu vechimea cea mai mica).
- pop()
  - intrare: S ∈ Stiva
  - iesire:
    - S din care s-a eliminat ultimul element introdus (cel cu vechimea cea mai mica);
    - eroare daca S este vida.
- top()
  - intrare: S ∈ Stiva
  - iesire: ultimul element introdus in S (cel cu vechimea cea mai mica);

Stiva – implementarea cu liste

| tipul Stiva | | tipul LLin |
|---|---|---|
| $push(S, e)$ | $=$ | $insereaza(S, 0, e)$ |
| $pop(S, e)$ | $=$ | $elimina(S, 0)$ |
| $top(S)$ | $=$ | $alKlea(S, 0)$ |

sau

| tipul Stiva | | tipul LLin |
|---|---|---|
| $push(S, e)$ | $=$ | $insereaza(S, lung(S), e)$ |
| $pop(S, e)$ | $=$ | $elimina(S, lung(S) - 1)$ |
| $top(S)$ | $=$ | $alKlea(S, lung(S) - 1)$ |

Stiva – implementarea cu structuri inlantuite

- push()

```
procedure push(S, e)
begin
        new(q)
        q-> elt ← e
        q-> succ ← S
        S ← q
end
```

- pop()

```
procedure pop(S)
begin
        if S == NULL then
                throw “eroare”
        q ← S
        S ← S-> succ
        delete(q)
end
```

## Coada

Obiecte:

Liste in care se cunoaste vechimea elementelor introduse: liste FIFO

Coada – operatii

- coadaVida()
  - intrare: nimic
  - iesire: C = () (lista vida)
- esteVida()
  - intrare: C ∈ Coada
  - iesire:
    - true daca C este vida
    - false daca C nu e vida
- insereaza()
  - intrare: C ∈ Coada, e ∈ Elt
  - iesire: C la care s-a adaugat e ca ultim element introdus (cel cu vechimea cea mai mica).
- elimina()
  - intrare: C ∈ Coada
  - iesire:
    - C din care s-a eliminat primul element introdus (cel cu vechimea cea mai mare);
    - eroare daca C este vida.
- citeste()
  - intrare: C ∈ Coada
  - iesire:
    - primul element introdus in C (cel cu vechimea cea mai mare);
    - eroare daca C este vida.

Coada – implementarea cu liste

| tipul Coada | | tipul LLin |
|---|---|---|
| $insereaza(C, e)$ | $=$ | $insereaza(C, lung(C), e)$ |
| $elimina(C)$ | $=$ | $elimina(C, 0)$ |
| $citeste(S)$ | $=$ | $alKlea(C, 0)$ |

Coada – implementarea cu tablouri

- insereaza()

```
procedure insereaza(C, e)
begin
    if (C.ultim + 1)%Max == C.prim then
        throw “eroare”
    else
        C.ultim ← (C.ultim + 1)%Max
        C.tab[ultim] ← e
end
```

Coada – implementarea cu structuri inlantuite

- insereaza()

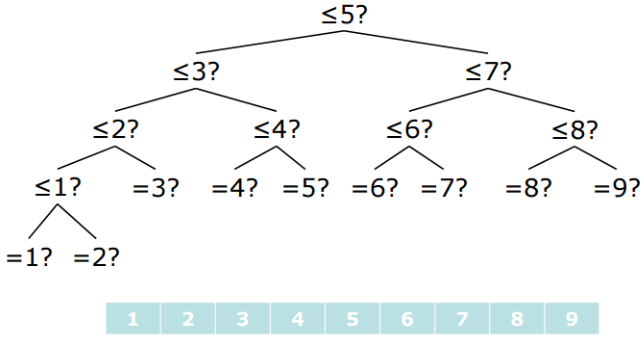
```
procedure insereaza(C, e)
begin
    new(q)
    q- > elt ← e
    q- > succ ← NULL
    if C.ultim == NULL then {
            C.prim ← q
            C.ultim ← q
    }
    else {
            C.ultim- > succ ← q
            C.ultim ← q
    }
end
```

# Curs 5
## Arbori. Arbori binari

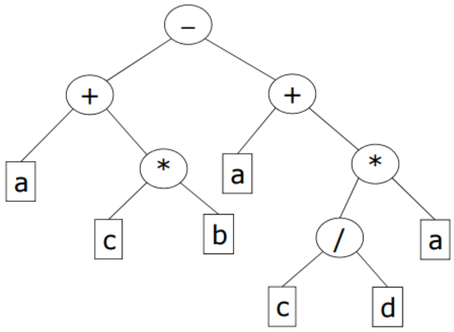
### Arbori (cu radacina)

- Model abstract pentru structuri ierarhice;
- Un arbore este format din noduri legate printr-o relaţie părinte-copil.

Arbori de decizie




Arbori sintactici




(a + c * b) - (a + c / d * a)

Arbori: terminologie

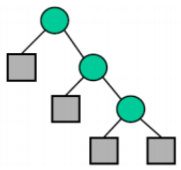
- Radacina: nodul fara parinte
- Nod intern: nod cu fiu >= 1
- Nod extern (frunza): nod fara fii
- Descendentii unui nod: fii, nepoti, etc
- Fratii unui nod: toate celelalte noduri avand acelasi parinte
- Subarbore: arbore format dintr-un nod si descendetii sai

### Arbori binari (ArbBin)

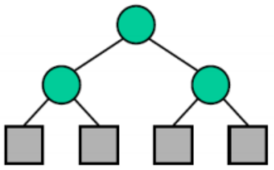
- obiecte : arbori binari.
  - un arbore binar este o colecţie de noduri cu
  - proprietăţile:
    1. orice nod are 0, 1 sau 2 succesori (fii, copii);
    2. orice nod, exceptând unul singur – rădăcina, are un singur nod predecesor (tatăl, părintele);
    3. rădăcina nu are predecesori;
    4. fiii sunt ordonaţi: fiul stâng, fiul drept (daca un nod are un singur fiu, trebuie menţionat care);
    5. nodurile fără fii formează frontiera arborelui.

Arbori binari: proprietati

- Notaţii
  - $n$ numărul de noduri
  - $n_e$ numărul de noduri externe
  - $n_i$ numărul de noduri interne
  - $h$ înălţimea



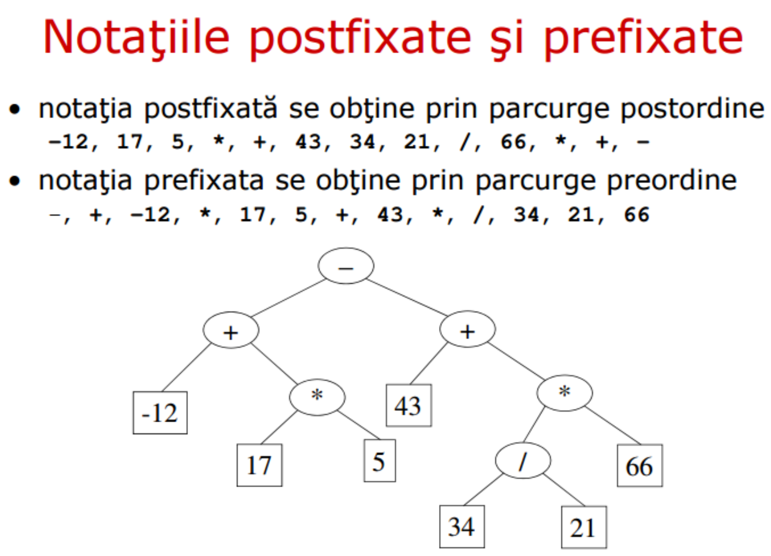
- Arbore propriu: fiecare nod intern are exact 2 fii
- Arbore complet: arbore propriu in care frunzele au aceeasi adancime

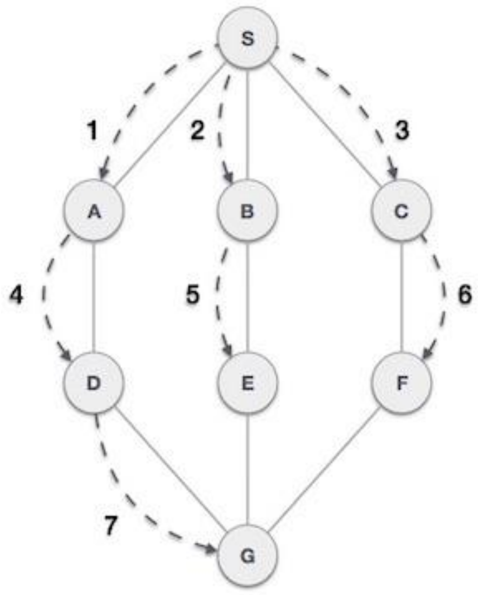


Arbori binari: parcurgeri

- Inordine (SRD)
- Preordine (RSD)
- Postordine (SDR)

Parcurgere BFS




# Curs 6
## Coada cu prioritate. Max-heap

### Coada cu prioritate - tip de date abstract

- Obiecte de tip data: structuri de date in care elementele sunt numite *atomi*; orice atom un *camp-cheie* numit *prioritate;*
- Elementele sunt memorate in functie de prioritate ci nu de pozitia lor.

<u>Coada cu prioritati: operatii</u> (# = cheie)

- Citeste: intrare – o coada cu prioritate C; iesire – atomul din C cu # cmm
- Elimina: intrare – – // – ; iesire – C fara atomul cu # cmm
- Insereaza: intrare – o coada cu prioritate C si un atom **at**; iesire – C cu **at**

### Max-Heap

Sunt arbori binari completi cu proprietatea: pentru orice nod, cheia din acel nod este mai mare sau egala decat cheile din nodurile fii

<u>Inaltimea unui maxHeap</u>
Teorema: Un maxHeap care contine n chei are inaltimea O(log n).

<u>maxHeap – eliminare</u>
Se elimina radacina heap-ului (elementul cu # cmm)

1. Se inlocuieste # radacinii cu # ultimului nod
2. Se sterge ultimul nod
3. Se reface proprietatea de maxHeap

<u>maxHeap – inserarea</u>
Se insereaza noua # intr-un nod nou

1. Se adauga noul nod ca cel mai din stanga pe ultimul nivel
2. Se insereaza noua cheie in acest nod
3. Se reface proprietatea de maxHeap

<u>maxHeap – implementarea cu tablouri</u>

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **12** | **9** | **8** | **7** | **1** | **3** | **4** | **5** | **2** |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Indexarea din 0:
i stanga: 2i + 1
i dreapta: 2i + 2
i parinte: (i-1)/2

<u>maxHeap – inserare</u>

```
procedure insereaza(a, n, cheie)
begin
        n ← n+1
        a[n-1] ← cheie
        j ← n-1
        heap ← false
        while ((j > 0) and not heap)
do
                k ← [(j-1)/2]
                if (a[j] > a[k])
                then swap(a[j], a[k])
                j ← k
                else heap ← true
end
```

```
procedure elimina(a, n)
begin
        a[0] ← a[n-1]
        n ← n-1
        j ← 0
        heap ← false
        while ((2*j+1 < n) and not
heap) do
                k ← 2*j+1
                if ((k < n-1) and (a[k]
        < a[k+1]))
                then k ← k+1
                if (a[j] < a[k])
                then swap(a[j], a[k])
                        j ← k
                else heap ← true
end
```

<u>maxHeap – elimina</u>

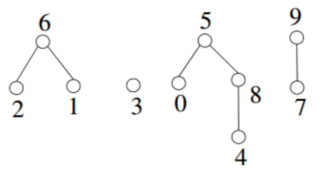
## Colectii de multimi disjuncte

<u>Colectii de multimi disjuncte: tip de date abstract</u>

- Find():
  - intrare – o colectie C, un element i din univers
  - iesire – submultimea din C la care apartine i
- union():
  - intrare – o col C, 2 elemnte (i, j)
  - iesire – C in care componentele lui i si j sunt reunite
- singleton():
  - intrare – o col C, un element i
  - iesire – C la care componenta lui i are pe i ca unic element

<u>Colectii de multimi disjuncte</u> „union-find”

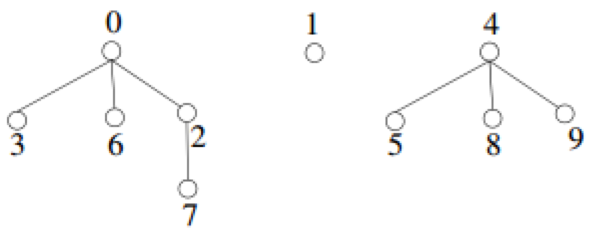
– n=10, {1,2,6},{3}, {0,4,5,8}, {7,9}



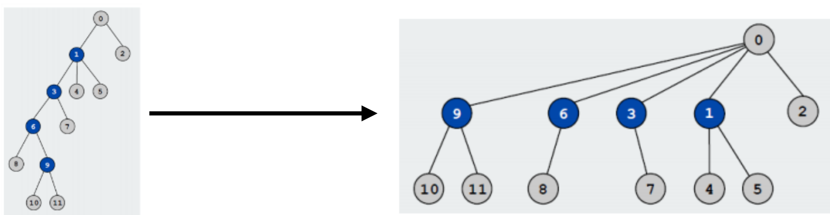

| parinte | 5 | 6 | 6 | -1 | 8 | -1 | -1 | 9 | 5 | -1 |
|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

<u>Structura “union-find” ponderata</u>

! Solutie la problema arborilor dezechilibrati
Mecanism: Memorarea numarului de varfuri din arbore (cu semn negativ)



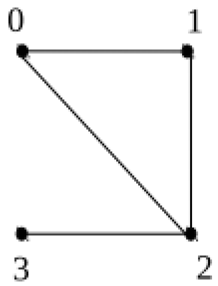

| parinte | -5 | -1 | 0 | 0 | -4 | 4 | 0 | 2 | 4 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |



# Curs 7
## Grafuri

### Graf simplu

G = (V,E)   V – multime de varfuri;   E – multime de muchii;
Obs: u – muchie este o pereche neordonata de varfuri distincte

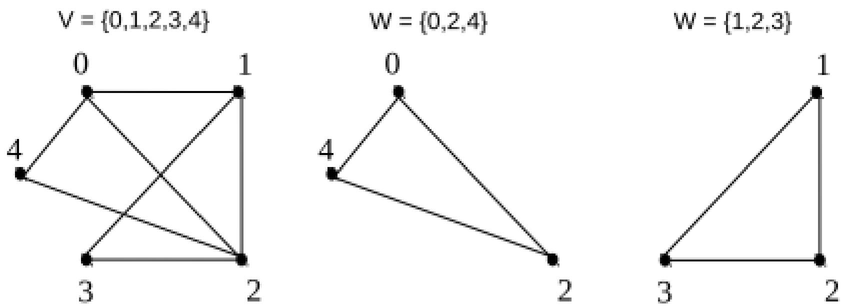



V = {0,1,2,3}
E = { {0,1}, {0,2}, {1,2}, {2,3}}
u = {0,1} = {1,0}

0,1 – extremitatile lui u
u este incidenta in 0 si 1
0 si 1 sunt adicante (vecine)

### Subgraf indus

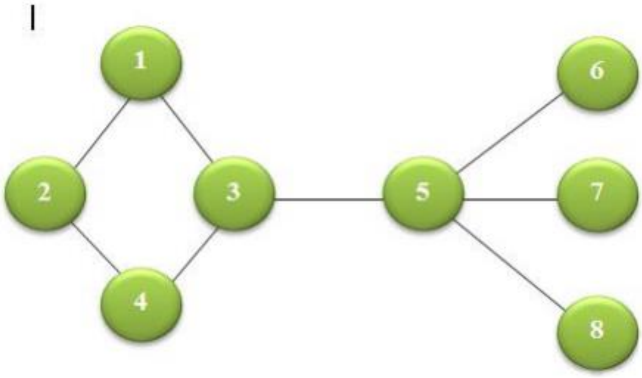
Un subgraf al unui graf G=(V,U) este un graf $H(V_1,U_1)$ astfel încât V1 este inclus in V iar $U_1$ conţine toate muchiile din U care au ambele extremităţi în $V_1$ (un subgraf se obţine eliminând vârfuri din V şi muchiile incidente acestor vârfuri). Vom spune că subgraful H este indus sau generat de mulţimea de vârfuri $V_1$.




### Grafuri – Conexitate

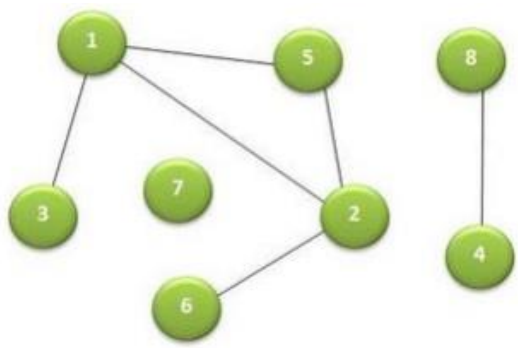
Un graf se numeşte conex dacă pentru oricare două vârfuri x şi y diferite ale sale, există un lanţ care le leagă.

Se numeşte componentă conexă a grafului G=(V,U), un subgraf $C=(V_1,U_1)$ conex al lui G care are proprietatea că nu există nici un lanţ în G care să lege un vârf din mulţimea $V_1$ cu un vârf din mulţimea $V-V_1$.




**GRAF CONEX** - format dintr-o singură componentă conexă

(conţine toate nodurile grafului)




**Graf NECONEX** - alcătuit din trei componente conexe
Componenta conexa 1 conţine nodurile: 1,3,5,2,6
Componenta conexa 2 conţine nodurile: 4, 8
Componenta conexa 3 contine nodurile: 7

### Graf – operatii

- grafVid()
  - intrare: nimic
  - iesire: graful vid (∅,∅)

- esteGrafVid()
  - intrare: G = (V, E)
  - iesire: true daca G = (∅,∅), false altfel

- insereazaMuchie()
  - intrare: G = (V, E), i,j ∈ V
  - iesire: G = (V, E ∪ {i,j})

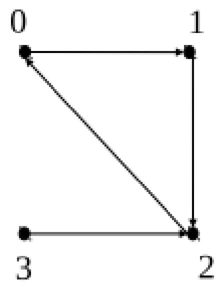
- insereazaVarf()
  - intrare: G = (V, E), V = {0,1,⋯,n-1}
  - iesire: G' = (V', E), V' = {0,1,⋯,n-1, n}

- eliminaMuchie() ...
- eliminaVarf() ...
- listaDeAdiacenta()
  - intrare: G = (V, E), i ∈ V
  - iesire: lista varfurilor adiacente cu i
- listaVarfurilorAccesibile()
  - intrare: G = (V, E), i ∈ V
  - iesire: lista varfurilor accesibile din i

Digraf (graf orientat)

D = (V,A) V – multime de varfuri; A – multime de arce;

Obs: a – arc este o pereche ordonata de varfuri distincte

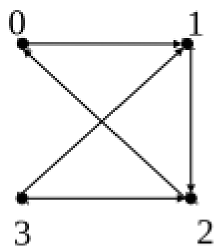



V = {0,1,2,3}
E = { (0,1), (2,0), (1,2), (3,2) }
a = (0,1) != (1,0)

0 – sursa lui a
1 – destinatia lui a

Digraf – Conexitate

- i R j daca si numai daca exista drum de la i la j si drum de la j la i
- R este relatie de echivalenta
- $V_1, \cdot \cdot \cdot, V_p$ clasele de echivalenta
- $G_i = (V_i, A_i)$ subdigraful indus de $V_i$
- $G_1, \cdot \cdot \cdot, G_p$ – componente tare conexe
- digraf tare conex = digraf cu o singura componenta tare conexa




V1 = {0, 1, 2}
A1 = {(0, 1),(1, 2),(2, 0)}
V2 = {3}
A2 = ∅

Digraf – operatii

- digrafVid()
  - intrare: nimic
  - iesire: digraful vid (∅, ∅)
- esteDigrafVid()
  - intrare: D = (V, A),
  - iesire: true daca D = (∅, ∅), false in caz contrar

- insereazaArc()
  - intrare: D = (V, A), i, j ∈ V
  - iesire: D = (V, A ∪ (i, j))
- insereazaVarf()
  - intrare: D = (V, A), V = {0, 1, · · · , n − 1}
  - iesire: D = (V0, A), V0 = {0, 1, · · · , n − 1, n}
- eliminaArc() ...
- eliminaVarf() ...
- listaDeAdiacentaExterioara()
  - intrare: D = (V, A), i ∈ V
  - iesire: lista varfurilor destinatare ale arcelor care pleaca din i
- listaDeAdiacentaInterioara()
  - intrare: D = (V, A), i ∈ V
  - iesire: lista varfurilor sursa ale arcelor care sosesc in i
- listaVarfurilorAccesibile()
  - intrare: D = (V, A), i ∈ V
  - iesire: lista varfurilor accesibile din i

## Implementarea cu matrici de adiacenta

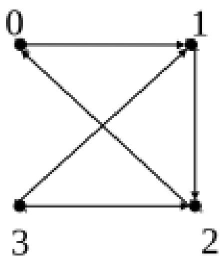
Reprezentarea digrafurilor:

- n numarul de varfuri
- m numarul de arce (optional)
- o matrice (a[i, j] | 1 ≤ i, j ≤ n)
- a[i, j] = if (i, j) ∈ A then 1 else 0

- daca digraful reprezinta un graf, atunci a[i, j] este simetrica
- lista de adiacenta exterioara a lui i ⊆ linia i
- lista de adiacenta interioara a lui i ⊆ coloana i

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 2 | 1 | 0 | 0 | 0 |
| 3 | 0 | 1 | 1 | 0 |

$\longrightarrow$




<u>Operatii</u>

- digrafVid
  n ← 0; m ← 0
- insereazaVarf: O(n)
- insereazaArc: O(1)
- eliminaArc: O(1)

eliminaVarf()

```
Procedure eliminaVirf(a, n, k)
begin
     for i ← 0 to n - 1 do
           for j ← 0 to n - 1 do
           if (i > k) then
                 a[i - 1, j] ← a[i, j]
           if (j > k) then
                 a[i, j - 1] ← a[i, j]
     n ← n - 1
end
```

*timp de executie: O(n2)*

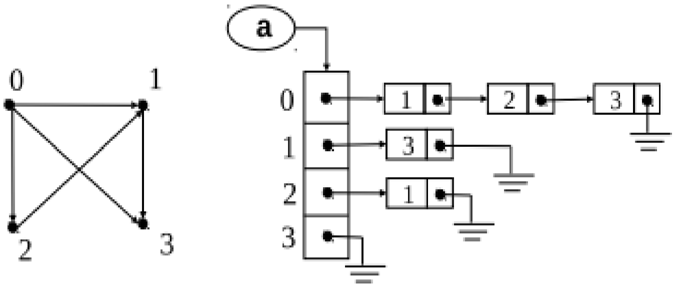
listaVarfurilorAccesibile()

```
Procedure inchReflTranz(a, n, b)
                  // (Warshall, 1962)
begin
     for i ← 0 to n - 1 do
       for j ← 0 to n - 1 do
         b[i, j] ← a[i, j]
         if (i = j) then
           b[i, j] ← 1
     for k ← 0 to n - 1 do
       for i ← 0 to n - 1 do
         if (b[i, k] = 1) then
           for j ← 0 to n - 1 do
             if (b[k, j] = 1) then
                b[i, j] ← 1
end
```

*timp de executie: O(n3)*

## Implementarea cu liste de adiacenta inlantuite

- Reprezentarea digrafurilor cu liste de adiacenta exterioara




  - un tablou a[0..n - 1] de liste inlantuite (pointeri)
  - a[i] este lista de adiacenta exterioara corespunzatoare lui i

Operatii

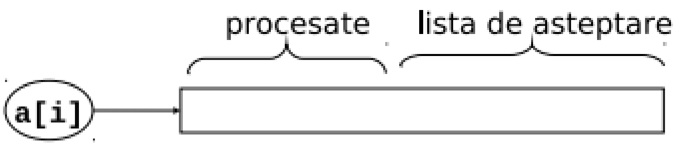
- digrafVid
- insereazaVarf: O(1)
- insereazaArc: O(1)
- eliminaVarf: O(n+m)
- eliminaArc: O(m)

**Procedure** *explorare(a, n, i0, S)*
**begin**
  **for** $i \leftarrow 0$ **to** $n-1$ **do**
    $p[i] \leftarrow a[i]$
  $SB \leftarrow (i0); S \leftarrow (i0)$
  $viziteaza(i0)$
  **while** $(SB \neq \emptyset)$ **do**
    $i \leftarrow citeste(SB)$
    **if** $(p[i] = NULL)$ **then**
      $SB \leftarrow SB - \{i\}$
    **else**
      $j \leftarrow p[i]->varf$
      $p[i] \leftarrow p[i]->succ$
      **if** $(j \notin S)$ **then**
        $SB \leftarrow SB \cup \{j\}$
        $S \leftarrow S \cup \{j\}$
        $viziteaza(j)$
**end**

## Algoritm de parcurgere (DFS, BFS)

Digrafuri: explorare sistematica

- se gestioneaza doua multimi
  - S = mulțimea varfurilor vizitate deja
  - SB ⊆ S submultimea varfurilor pentru care exista sanse sa gasim vecini nevizitati inca
- lista de adiacenta (exterioara) a lui i este divizata in doua:



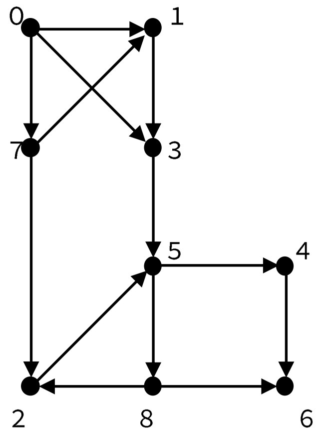

### Explorare sistematica: complexitate

**Teorema**
In ipoteza ca operatiile peste S si SB precum si viziteaza() se realizeaza in O(1), complexitatea timp, in cazul cel mai nefavorabil, a algoritmului explorare este O(n + m).

## Explorarea DFS (Depth First Search)

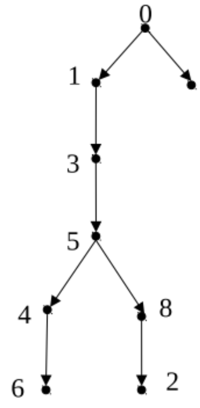
SB este implementata ca stiva

```
    SB ← (i0) ⇔SB ← stivaVida()
                push(SB, i0)
i ← citeste(SB) ⇔i ← top(SB)
  SB ← SB - {i} ⇔pop(SB)
  SB ← SB ∪ {j} ⇔push(SB, j)
```

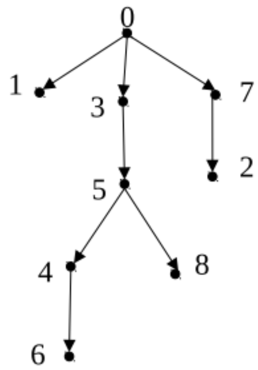



Stiva

| | | |
|---|---|---|
| 6 | 2 | |
| 4 | 8 | |
| 5 | 5 | |
| 3 | 3 | |
| 1 | 1 | 7 |
| 0 | 0 | 0 |




Arborele DFS



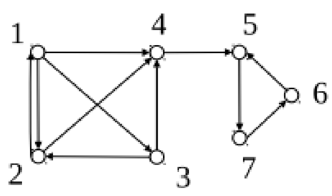

Arborele BFS

## Explorarea BFS (Breadth First Search)

<u>SB este implementata ca o coada</u>

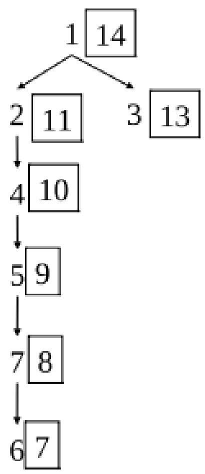
```
     SB ← (i0) ⇔SB ← coadaVida()
                  insereaza(SB, i0)
i ← citeste(SB) ⇔citeste(SB,i)
  SB ← SB – {i} ⇔elimina(SB)
  SB ← SB ∪ {j} ⇔insereaza(SB, j)
```

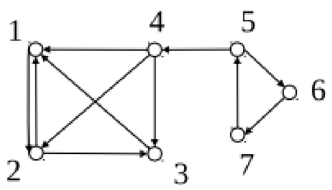
## Determinarea componentelor (tare) conexe



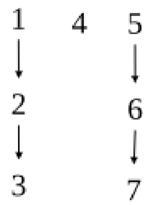

D

$1 \to (2, 3, 4)$
$2 \to (1,\ 4)$
$3 \to (2, 4)$
$4 \to (5)$
$5 \to (7)$
$6 \to (5)$
$7 \to (6)$




$D^T$



# Curs 8
## Sortare

### Continut

- Sortare bazata pe comparatii
  - Interschimbare / bubble sort
  - Insertie / insertion sort
  - Selectie / selection sort
  - Interclasare / merge sort
  - Rapida / quick sort
- Sortare prin numarare
- Sortare prin distribuire

### Sortare prin interschimbare / bubble sort

Principiul de baza: i>i+1 ? interschimba a[i]^=a[i+1] cat timp ∃ i>i+1

Algoritm:

```
Procedure bubbleSort(a, n)
begin
      ultim ← n - 1
      while (ultim > 0) do
            n1 ← ultim-1; ultim ← 0
            for i ← 0 to n1 do
                  if (a[i] > a[i + 1]) then
                        swap(a[i], a[i+1])
                        ultim ← i
end
```

3 7 2 1 4 (n1 = 3)
3 7 2 1 4
3 2 7 1 4
3 2 7 1 4
3 2 1 7 4
3 2 1 7 4
3 2 1 4 7
3 2 1 4 7

3 2 1 4 7 (n1 = 2)
2 3 1 4 7
2 3 1 4 7
2 1 3 4 7
2 1 3 4 7
2 1 3 4 7

2 1 3 4 7 (n1 = 0)
1 2 3 4 7
1 2 3 4 7

Analiza:
Caz nefav: T(n)=O($n^2$) (a[0]>a[1]>...>)
Caz fav: O(n)

### Sortare prin insertie directa

Principiul de baza: presp a[0..i-1] sortat si inseram a[i] ai. a[0..i] e sortat

Algoritm: (cautarea pozitiei lui a[i] secvential)

```
Procedure insertSort(a, n)
begin
      for i ← 1 to n - 1 do
            j ← i-1 // a[0..i - 1] sortat
            temp ← a[i] // caut locul lui temp
            while((j≥0) and (a[j]>temp)) do
                  a[j + 1] ← a[j]
                  j ← j-1
            if (a[j + 1]! = temp) then
                  a[j + 1] ← temp
end
```

Exemplu
3 7 2 1
3 7 2 1
2 3 7 1
1 2 3 7

Analiza:
Caz nefav: T(n)=O($n^2$) (a[0]>a[1]>...>)
Caz fav: O(n)

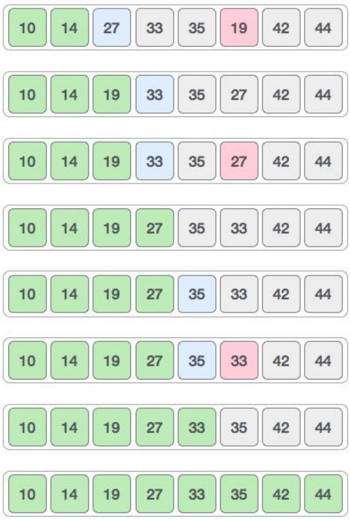
### Sortare prin selectie

Principiul de baza: selecteaza un element si il duce pe pozitia finala and repeat

<u>Selectia poate fi naiva</u> : alegerea elementelor in ordinea in care se afla initial (de la n - 1 la 0 sau de la 0 la n - 1) sau sistematica cu maxHeap.

<u>Algoritm</u>: (naiva) Analiza: Complexitatea timp este mereu $O(n^2)$

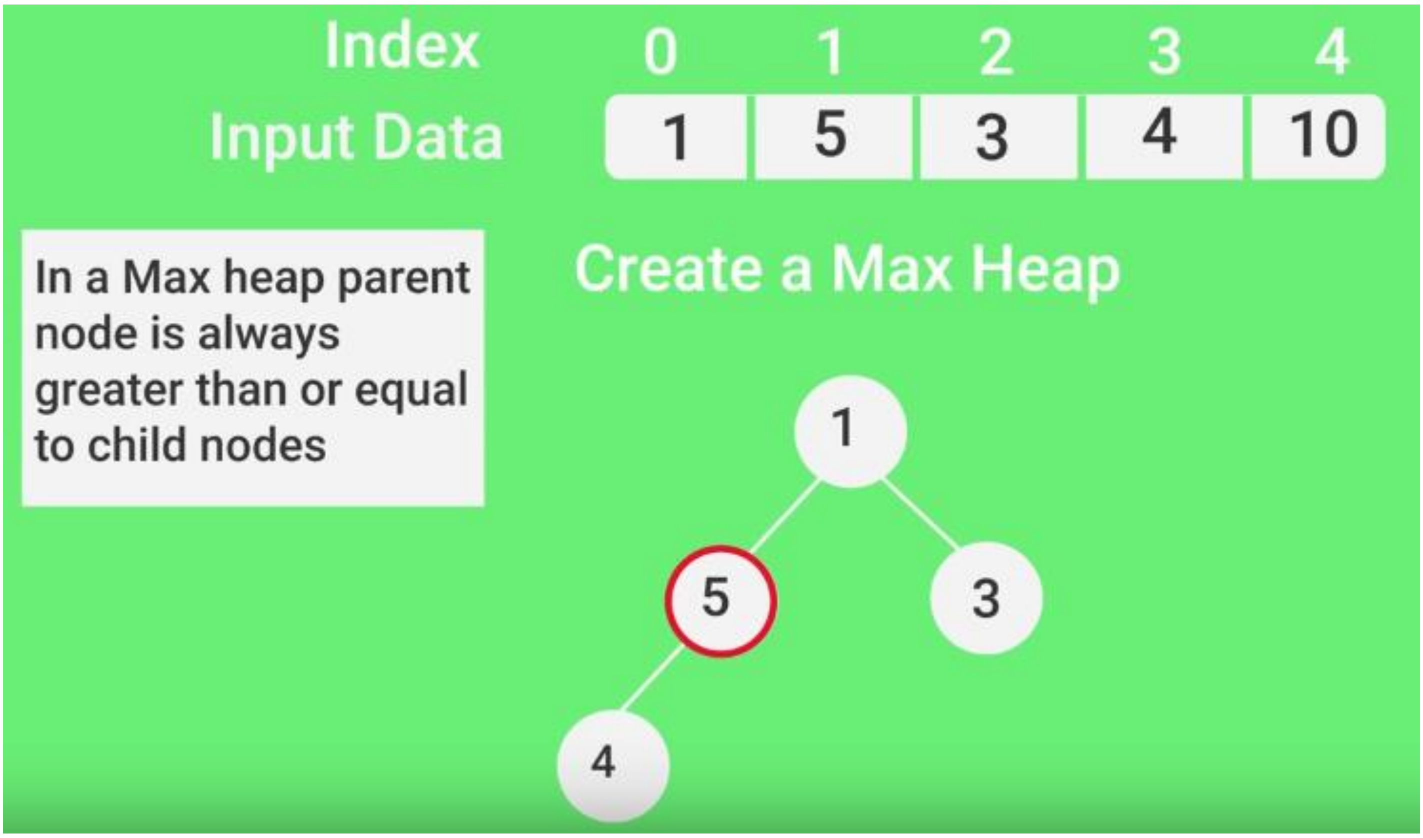
```
Procedure naivSort(a, n)
begin
      for i ← n - 1 to 1 do
      imax ← i
      for j ← i - 1 to 0 do
            if (a[j] > a[imax]) then
                  imax ← j
            if (i! = imax) then
                  swap(a[i], a[imax])
end
```




<u>Algoritm</u>: (heapSort) <u>Analiza</u>: T(n)=O(nlogn)

```
Procedure insereazaAlTlea(a, n, t)
begin
      j ← t
      heap ← false
      while (2*j+1<n and not heap) do
            k ← 2 * j + 1
            if (k<n-1 and a[k]<a[k+1])
                  k ← k + 1
            if (a[j] < a[k]) then
                  swap(a[j], a[k])
                  j ← k
            else
                  heap ← true
end
```

```
Procedure heapSort(a, n)
begin
      // construieste maxheap-ul
      for t ← (n - 1)/2 to 0 do
            insereazaAlTlea(a, n, t)
      // elimina
      r ← n - 1
      while (r > 0) do
            swap(a[0], a[r])
            insereazaAlTlea(a, r, 0)
            r ← r - 1
end
```



## Sortare prin interclasare / merge sort

Principiul de baza: imparti sirul in mijloc pana ajungi la cate 1 element

Algoritm:

```
Procedure DivideEtImpera(P, n, S)
begin
        if (n ≤ n0) then
        determina S prin metode elementare
        else
        imparte P in P1, ..., Pa
        DivideEtImpera(P1, n1, S1)
        ...
        DivideEtImpera(Pa, na, Sa)
        Asambleaza(S1, ..., Sa, S)
end
```

Analiza:

Timp: T(n) = O(nlogn)
Spatiu suplimentar: O(n)

## Sortare prin rapida / quick sort

Principiul de baza: pe baza de pivot

Algoritm:

```
Procedure partitioneaza(a, p, q, k)
begin
        x ← a[p]
        i ← p + 1
        j ← q
        while (i <= j) do
                if (a[i] <= x) then
                i ← i + 1
                if (a[j] >= x) then
                j ← j - 1
                if (i<j) and (a[i]>x) and
(x>a[j])
                then
                        swap(a[i], a[j])
                        i ← i + 1
                        j ← j - 1
        k ← i - 1
        a[p] ← a[k]
        a[k] ← x
end
```

```
Procedure quickSort(a, p, q)
begin
        while (p < q) do
                partitioneaza(a, p, q, k)
                quickSort(a, p, k - 1)
                quickSort(a, k + 1, q)
end
```

Analiza:

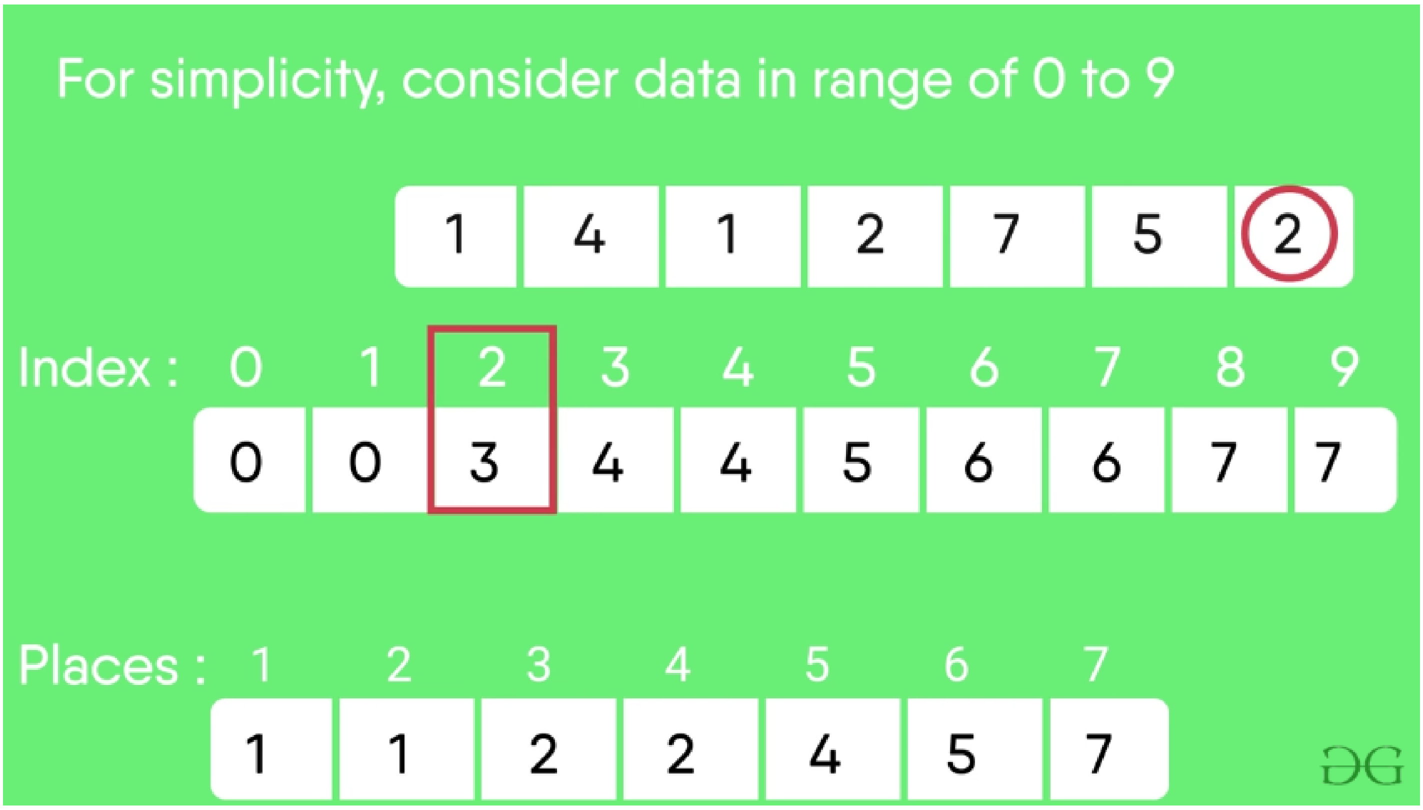
! Alegerea pivotului influenteaza eficienta algoritmului
Cazul cel mai nefav: Pivotul e cea mai mica sau mare valoare T(n) = $O(n^2)$
Caz fav: Un pivot bun, imparte tabloul in 2 subtab de dimensiuni comparabile
Inaltimea arborelui de recursie este O(logn) iar T(n) = (nlogn)

## Sortare prin numarare

Principiul de baza: Se determina pozitia fiecarui element in tabloul sortat numarand cate elemente sunt mai mici decat acesta

```
Procedure countingSort(a, b, n, k)
begin
        for i ← 1 to k do
                c[i] ← 0
        for j ← 0 to n - 1 do
                c[a[j]] ← c[a[j]] + 1
        for i ← 2 to k do
                c[i] ← c[i] + c[i - 1]
        for j ← n - 1 to 0 do
                b[c[a[j]] - 1] ← a[j]
                c[a[j]] ← c[a[j]] - 1
end
```

Analiza:
T(n) = O(k+n) k=lungimea sir




Sortare prin distribuire

Ipoteza: Elementele a[i] sunt distribuite uniform peste intervalul [0,1)
Analiza: T(n) = O(n)

Principiu:
- se divide intervalul [0,1) in n subintervale de marimi egale, numerotate de la 0 la n-1;
- se distribuie elementele a[i] in intervalul corespunzator: bn · a[i]c;
- se sorteaza fiecare pachet folosind o alta metoda;
- se combina cele n pachete intr-o lista sortata.

Algoritm:

```
Procedure bucketSort(a, n)
begin
        for i ← 0 to n - 1 do
                insereaza(B[bn · a[i]c], a[i])
        for i ← 0 to n - 1 do
                sorteaza lista B[i]
        concateneaza in ordine listele B[0], B[1], · · · , B[n - 1]
end
```

## Sortare – complexitate

| **Algoritm** | | **Caz** | |
|---|---|---|---|
| | Favorabil | Mediu | Nefavorabil |
| bubbleSort | n | $n^2$ | $n^2$ |
| insertSort | n | $n^2$ | $n^2$ |
| naivSort | $n^2$ | $n^2$ | $n^2$ |
| heapSort | nlogn | nlogn | nlogn |
| mergeSort | nlogn | nlogn | nlogn |
| quickSort | nlogn | nlogn | $n^2$ |
| countingSort | - | n+k | n+k |
| bucketSort | - | n | - |

## Cand utilizam un anumit algoritm de sortare ?

**Quick sort:** cand nu e nevoie de o metoda stabila si performanta medie e mai importanta decat cea n cazul cel mai nefavorabil; O(n log n) complexitatea timp medie, O(log n) spatiu suplimentar
**Merge sort:** cand este necesara o metoda stabila; complexitate timp O(n log n); dezavantaje: O(n) spatiu suplimentar, constanta mai mare decat cea a QuickSort
**Heap sort:** cand nu e nevoie de o metoda stabila si ne intereseaza mai mult performanta in cazul cel mai nefavorabil decat in cazul mediu; timp O(n log n), spatiu O(1)
**Insert sort:** cand n e mic
**Counting sort:** valori dintr-un interval
**Bucket Sort:** valorile sunt distribuite aproximativ uniform

# Curs 9
# Cautare

## Problema cautarii

- Aspectul static:
  - U multime univers, $S \subseteq U$
  - operatia de cautare:
    - instanta: $a \in U$
    - intrebare: $a \in S$?

- Aspectul dinamic:
  - operatia de inserare
    - intrare: $S$, $x \in U$
    - iesire: $S \cup \{x\}$
  - operatia de stergere
    - intrare: $S$, $x \in U$
    - iesire: $S - \{x\}$

## Cautare in liste liniare – complexitate

| Tip de date | Implementare | Cautare | Inserare | Stergere |
|---|---|---|---|---|
| Lista liniara | Tablouri | O(n) | O(1) | O(n) |
| | Liste inlantuite | O(n) | O(1) | O(1) |
| Lista liniara ordonata | Tablouri | O(logn) | O(n) | O(n) |
| | Liste inlantuite | O(n) | O(n) | O(1) |

## Cautare binara

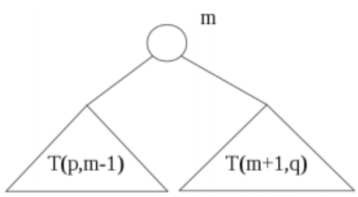
<u>Principiul de baza</u>: Multimea univers este total ordonata (U, <=)
<u>Structura de date utilizata</u>:
- tabloul s[0 ... n-1]
- s[0] < s[1] < ... < s[n-1]

## Cautare binara: aspect static

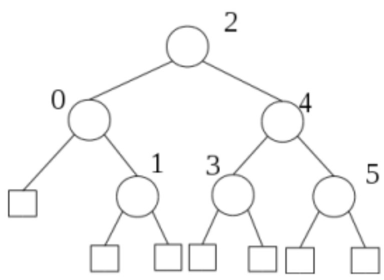
```
Function poz(s[0..n - 1], n, a)
begin
	p ← 0; q ← n - 1
	m ← (p + q)/2
	while (s[m]! = a and p < q) do
		if (a < s[m]) then
			q ← m - 1
		else
			p ← m + 1
		m ← (p + q)/2
	if (s[m] = a) then
		return m
	else
		return -1
end
```

## Arbore binar asociat cautarii binare

T(p,q)




T=T(0,n-1)
n=6



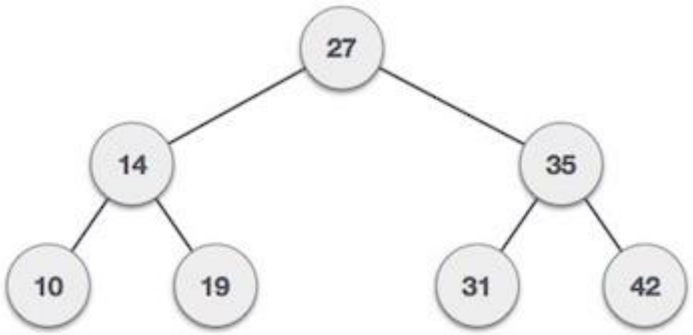
## Arbori binari de cautare

Aspect dinamic
Multimea S sufera operatii de actualizare in timp (inserare/stergere)

Arbore binar de cautare:
In orice nod v este memorata o valoare dintr-o multime total ordonata.
Arborele binar de cautare asociat unei multimi de chei nu este unic.

left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)

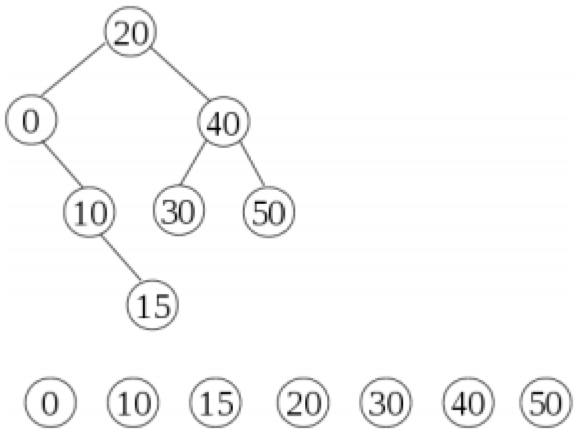
Reprezentare:




Basic Operations:
- the basic operations of a tree -

- **Search** - Searches an element in a tree.
- **Insert** - Inserts an element in a tree.
- **Pre-order Traversal** - Traverses a tree in a pre-order manner.
- **In-order Traversal** - Traverses a tree in an in-order manner.
- **Post-order Traversal** - Traverses a tree in a post-order manner.

Operatii: Sortare
Parcurgere in inordine




```
Function inordine(v, viziteaza)
begin
        if (x == NULL) then
                return
        else
                inordine(v → stg, viziteaza)
                viziteaza(v)
                inordine(v → drp, viziteaza)
end
```

! **Complexitate:** T(n) = O(n)

Operatii: Cautare

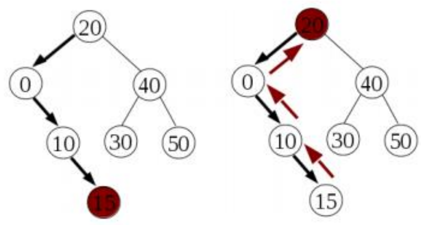
```
Function poz(t, x)
begin
        p ← t
        while (p! = NULL and p → val! = x) do
                if (x < p → val) then
                        p ← p → stg
                else
                        p ← p → drp
        return p
end
```

! **Complexitate:** T(n) = O(h), h – inaltimea

Operatii: Predecesor/Succesor

Modifica operatia de cautare: daca valoarea x nu se gaseste in arbore, atunci returneaza fie cea mai mare valoare < x (predecesor) fie cea mai mica > x (succesor)

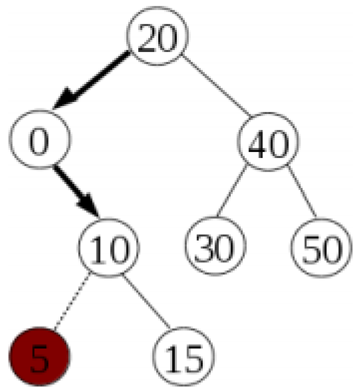
```
Function succesor(t)
begin
        if (t → drp! = NULL) then
                /*min(t → drp)*/
                p ← t → drp
                while (p → stg! = NULL) do
                        p ← p → stg
                return p
        else
                p ← pred[t]
                while (p! = NULL and t == p → drp) do
                        t ← p
                        p ← pred[p]
                return p
end
```




Operatii: Inserare
Similar cu operatia de cautare

**! Complexitate:** T(n) = O(h)

```
Procedure insArbBinCautare(t, x)
begin
        if (t == NULL) then
                new(t);
                t → val ← x;
                t → stg ← NULL; t → drp ← NULL
        else
                p ← t
                while (p! = NULL) do
                        predp ← p
                        if (x < p → val) then
                                p ← p → stg;
                        else if (x > p → val) then
                                p ← p → drp;
                        else p ← NULL;
                if (predp → val! = x) then
                        if (x < predp → val) then
                        /* adauga x ca fiu stinga al lui predp */
                        else
                        /* adauga x ca fiu dreapta al lui predp */ ;
end
```




Operatii: Eliminare
Se cauta x in arborele t; daca exista atunci se disting cazurile:

1. Nodul p ce contine x nu are fii (este frunza):
   - Eliminare simpla
2. Nodul p ce contine x are un fiu:
   - Se face legatura intre parinte si nodul fiu.
3. Nodul p ce contine x are doi fii:
   a. Determina cea mai mare valoare y mai mica decat x (cobori din x la stanga si cobori la dreapta cat se poate)
   b. Interschimba x cu y
   c. Sterge nodul y cazurile 1 si 2

**!Complexitate:** T(n) = O(h)

Algoritm: eliminare (cazul 1 sau 2)

```
Procedure elimCaz1sau2(t, predp, p)
begin
      if (p == t) then
            /* t devine vid sau */
            /* unicul fiu al lui t devine radacina */
      else if (p → stg == NULL) then
            /* inlocuieste in predp pe p cu p → drp */
      else
            /* inlocuieste in predp pe p cu p → stg */
end
```

Algoritm: eliminare (cazul 3)

```
Procedure elimArbBinCautare(t, x)
begin
      if (t! = NULL) then
            p ← t; predp ← NULL
            while (p! = NULL and p → val! = x) do
                  predp ← p
                  if (x < p → val) then p ← p → stg;
                  else p ← p → drp;
            if (p! = NULL) then
                  if (p → stg == NULL or p → drp == NULL) then
                        elimCaz1sau2(t, predp, p)
                  else
                        q ← p → stg; predq ← p
                        while (q → drp! = NULL) do
                              predq ← q; q ← q → drp
                        p → val ← q → val
                        elimCaz1sau2(t, predq, q)
end
```

**!Complexitate timp:** Caz nefav: O(n), n elemente | Caz fav: O(logn)

## Arbori de cautare echilibrati

- Arbori AVL (Adelson-Velsii and Landis, 1962)
- Arbori B/2-3-4 arbori (Bayer and McCreight, 1972)
- Arbori rosu-negru (Bayer, 1972)
- Arbori Splay (Sleator and Tarjan, 1985)
- Treaps (Seidel and Aragon, 1996)

## Arbori AVL

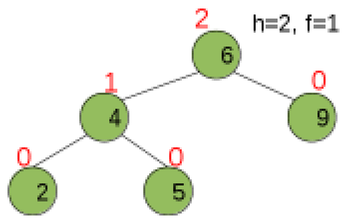
Un arbore binar de cautare ***t*** este un arbore AVL daca pentru orice varf **V** factorul de echilibrare este <= 1

BalanceFactor = height(left-sutree) – height(right-sutree)

Lema: Daca t este un AVL cu n noduri interne atunci h(t) = O(log n)

Teorema: Clasa arborilor AVL este O(log n) stabila

Rotatii: LL → R | RR → L || LR → LR || RL → RL



Rotatie stanga simpla | T(n) = O(1)

```
Procedure rotatieStanga(x)
begin
        y ← x → drp
        x → drp ← y → stg
        y → stg ← x
        return y
end
```

Inserare: algoritm

```
Procedure echilibrare(t, x)
begin
        while (x! = NULL) do
        /* actualizeaza inaltimea h(x) */
        if (h(x → stg)) ≥ 2 + h(x → drp)) then
                if (h(x → stg → stg)) ≥ h(x → stg → drp)) then
                        rotatieDreapta(t, x)
                else
                        rotatieStanga(t, x → stg); rotatieDreapta(t, x)
        else
                if (h(x → drp)) ≥ 2 + h(x → stg)) then
                        if (h(x → drp → drp)) ≥ h(x → drp → stg)) then
                                rotatieStanga(t, x)
                        else
                                rotatieDreapta(t, x → drp); rotatieStanga(t, x)
        x ← pred[x]
end
```

Avantaje si dezavantaje AVL

- Avantaje:
  - Cautarea, inserarea si stergerea se realizeaza cu complexitatea O(log n).
- Dezavantaje:
  - Spatiu suplimentar pentru memorarea inaltimii / factorului de echilibrare.
  - Operatiile de re-echilibrare sunt costisitoare.
- Sunt preferati cand facem mai multe cautari si mai putine inserari si stergeri
- Data Analysis, Data Mining

# Curs 10
## Arbori de cautare echilibrati

### Arbori bicolori / red-black tree

O altă structură de căutare care garantează un timp de execuţie de O(log n) în cazul cel mai defavorabil pentru cele trei operaţii fundamentale pe arbori binari de căutare sunt arborii bicolori, cunoscuţi şi sub denumirea de arbori Roşu-Negru.

### Red-Black definitie:

Un arbore Roşu-şi-Negru este un arbore binar de căutare în care fiecare nod are asociată o culoare (roşu sau negru) conform următoarelor reguli :

1. un nod este colorat cu rosu sau negru;
2. radacina si nodurile frunza (nil – care fac parte din structura) sunt colorate cu negru;
3. daca un nod este rosu, atunci fiii sai sunt negri;
4. drumurile de la un nod la nodurile de pe frontiera au acelasi numar de noduri negre.

!Observatii:

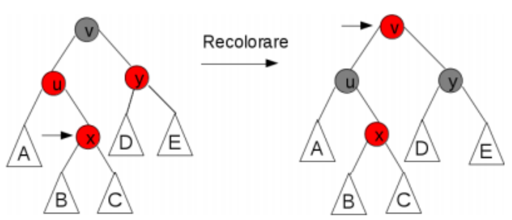
1. Pentru ca toate nodurile care conţin o cheie să fie noduri interne, am adăugat în arbore noduri fictive, marcate , pe care le-am numit externe şi care vor constitui frunzele negre ale arborelui.
2. Din definiţie deducem că orice subarbore a unui arbore Roşu-şi-Negru este arbore Roşu-şi-Negru. Regula 3 poate fi reformulată echivalent astfel: toate drumurile de la un nod la nodurile frunză descendente conţin acelaşi număr de noduri negre.

! Intr-un arbore bicolor cu n noduri operatia de cautare are complexitatea timp O(log n).

### Operatia de inserare:

1. Se cauta pozitia de inserare si se procedeaza ca la arbori binari
2. Se coloreaza noul nod cu rosu
3. Se restaureaza proprietatile de arbore bicolor prin recolorare si rotatii:
   a. Proprietatea 1 – satisfacuta
   b. Proprietatea 2 – satisfacuta (ambii fii ai nodului inserat sunt negrii) (Daca nodul inserat este radacina → recolorare in negru)
   c. Proprietatea 3:
      (1) “unchiul” nodului inserat este rosu:
         - Se recoloreaza “parintele” si “unchiul” in negru si “bunicul” in rosu.
      (2) “unchiul” nodului inserat este negru si nodul inserat este fiul drept al unui fiu stang:
         - Se aplica o rotatie simpla la stanga intre nodul curent si nodul parinte.
      (3) “unchiul” nodului inserat este negru si nodul inserat este fiul stang al unui fiu stang:
         - Se aplica o rotatie simpla la dreapta intre nodul “parinte” si nodul “bunic” + se recoloreaza nodurile “parinte” (in negru) si “bunic” (in rosu).

Caz (1)




Caz (2) si (3)