Facultatea de Matematica si Informatica
Universitatea Bucuresti

Structuri de Date si Algoritmi

Curs 11

Dobrovat Anca - Madalina

An universitar 2015 – 2016
Semestrul al II-lea

09/05/2016

Curs 11 - Cuprins

4. Algoritmi de sortare pentru multimi statice (vectori)

Clasa algoritmilor de sortare bazati pe comparatii intre chei.

.....

Sortarea cu ansamble (HeapSort).

Limita inferioara a performantei algoritmilor de sortare bazati pe comparatii intre chei.

Sursa: – R. Ceterchi: "Structuri de date si Algoritmi. Aspecte matematice si aplicatii", Editura Univ. din Bucuresti, 2001

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Reminder - Sortarea prin selectie directa

La fiecare pas i aveam de selectat minimul din multimea A[i..n].

Cautarea minimului se facea secvential, deci numarul de comparatii era tot timpul maxim.

Sortarea prin selectie directa se poate imbunatati daca exista o structura de date care sa permita extragerea minimului / maximului mai rapi, eventual optim.

Structura arborescenta care optimizeaza extagerea minimului / maximului: **arborele binar partial ordonat si complet (ansamblul)**

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Arbori partial ordonati si ansamble

Def Se numeste **arbore partial max-ordonat** un arbore binar cu chei de un tip total ordonat si cu proprietatea ca in orice nod u al sau avem relatiile:

$$\begin{cases} info[u] > info[root(left[u])], \text{ daca } left[u] \text{ este nevid} \\ info[u] > info[root(right[u])], \text{ daca } right[u] \text{ este nevid} \end{cases}$$

Pentru orice nod u: $\begin{cases} info[u] > info[v], \forall v \in left[u] \\ info[u] > info[w], \forall w \in right[u] \end{cases}$

➔ **cheia maxima se va afla in radacina**

Conceptul de arbore partial min-ordonat se defineste analog.

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Arbori partial ordonati si ansamble

Def Arbore binar **complet pe niveluri** este un a.b. cu toate nivelurile pline, eventual cu exceptia ultimului nivel, unde toate nodurile vor aliniate cel mai la stanga.

Acest tip de arbore binar se poate reprezenta ca vector (deci alocare statica si acces in timp O(1) la i si la tata).

Def Se numeste **ansamblu (max-ansamblu)** un arbore binar max-ordonat si complet pe niveluri, reprezentat ca vector.

Conceptul de min – ansamblu se defineste analog.

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Inserarea intr-un ansamblu

Se urmeaza pasii:

1. Se pune nodul de inserat (nod) pe ultimul nivel al arborelui, aliniat cel mai la stanga. (arborele ramane complet).

2. Se repeta (eventual pana la radacina) comparatia intre info[nod] si info[tata[nod]]

    2.1 Daca info[nod] < info[tata[nod]] atunci am gasit locul lui nod in ansamblu (noua cheie nu violeaza conditia de arbore max-ordonat)

    2.2 Daca nu, interschimb nod cu tata[nod] si reluam de la (a).

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Inserarea intr-un ansamblu

```
void InsHeap(int Ans[], int n, int Val)
{  // in ansamblul Ans[1...n] se insereaza Val
   n++;                                          // creste dimensiunea ansamblului cu 1
   p = n;                                          // p = indice pentru nodul curent
   while (p > 1)                                 // cat timp nu am ajuns la radacina
   {
      Tata = p/2;
      if (Val <= Ans[Tata])
      {
         Ans[p] = Val;                                  // se insereaza - cazul (a)
         return;                                                  // am terminat
      }
      else
      {
         Ans[p] = Ans[Tata];                    //  se coboara tatal in locul fiului
         p = Tata;                              //  nodul curent se reactualizeaza
      }
   }
   Ans[1] = Val;                                 //  inserarea in radacina - cazul (b)
}
```

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Construirea unui ansamblu. Asamblarea

Se realizeaza prin inserari repetate:

daca avem cel putin o cheie, atunci ea se va insera in radacina (un arbore cu un nod este ansamblu).

- pt fiecare valoare noua se foloseste algoritmul de inserare **InsHeap**.

```
void Asamblare(int Ans[], int n)
{ int j;
   for (j = 1; j <= n-1; j++)
    InsHeap(A,j,A[j+1]);
 }
```

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Construirea unui ansamblu. Asamblarea

Asamblarea, operatie specifica, este construirea unui ansamblu din inserari repetate de chei care se afla deja in locatiile unui vector:

-A[1] este ansamblu
- la ecare pas iterativ j se apeleaza procedura de inserare in ansamblul A[1..j] a valorii A[j+1], pentru j=1.. n - 1.

```
void Asamblare(int Ans[], int n)
{ int j;
   for (j = 1; j <= n-1; j++)
    InsHeap(A,j,A[j+1]);
 }
```

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Exemplificare pe vector

```
procedure Asamblare (A,n)
   For j← 1,n-1 do
         InsHeap(A,j,A[j+1])
   endFor
endproc
```

```
procedure InsHeap(Ans, n, Val)
   n← n + 1
   p← n
   While p > 1 do
      Tata ← p div 2
      If Val<= Ans[Tata] then
            Ans[p] ← Val
            exit
      Else
            Ans[p] ← Ans[Tata]
            p← Tata
      endIf
   endWhile
   Ans[1]← Val
endproc
```

Vectorul initial A:

7	10	1	3	5	6	4	2	9	8

Ans[1]: 7

InsHeap(Ans, 1, A[2])
n = 2; p = 2;
p > 1 [Tata = 1
10 > Ans[Tata] [Ans[2] = 7
p = 1
Ans[1] = 10

Ans:

10	7

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Expl

```
procedure Asamblare (A,n)
   For j← 1,n-1 do
         InsHeap(A,j,A[j+1])
   endFor
endproc
```

```
procedure InsHeap(Ans, n, Val)
   n← n + 1
   p← n
   While p > 1 do
      Tata ← p div 2
      If Val<= Ans[Tata] then
            Ans[p] ← Val
            exit
      Else
            Ans[p] ← Ans[Tata]
            p← Tata
      endIf
   endWhile
   Ans[1]← Val
endproc
```

Vectorul initial A:

7	10	1	3	5	6	4	2	9	8

Ans[1]: | 7 |

InsHeap(Ans, 1, A[2])

Ans: | 10 | 7 |

InsHeap(Ans, 2, A[3])
n = 3; p = 3;
p > 1 [Tata = 1
1 <= Ans[Tata] [Ans[3] = 1
exit

Ans: | 10 | 7 | 1 |

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Expl

```
procedure Asamblare (A,n)
   For j← 1,n-1 do
         InsHeap(A,j,A[j+1])
   endFor
endproc
```

```
procedure InsHeap(Ans, n, Val)
   n← n + 1
   p← n
   While p > 1 do
      Tata ← p div 2
      If Val<= Ans[Tata] then
            Ans[p] ← Val
            exit
      Else
            Ans[p] ← Ans[Tata]
            p← Tata
      endIf
   endWhile
   Ans[1]← Val
endproc
```

Vectorul initial A:

7	10	1	3	5	6	4	2	9	8

Ans[1]: | 7 |

InsHeap(Ans, 1, A[2])

Ans: | 10 | 7 |

InsHeap(Ans, 2, A[3])

Ans: | 10 | 7 | 1 |

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Expl

Vectorul initial A:

7	10	1	3	5	6	4	2	9	8

```
procedure InsHeap(Ans, n, Val)
    n ← n + 1
    p ← n
    While p > 1 do
        Tata ← p div 2
        If Val<= Ans[Tata] then
              Ans[p] ← Val
              exit
        Else
              Ans[p] ← Ans[Tata]
              p ← Tata
        endIf
    endWhile
    Ans[1] ← Val
endproc
```

InsHeap(Ans, 3, A[4])
n = 4; p = 4;
p > 1 [Tata = 2
3 <= Ans[Tata] [Ans[4] = 3
exit

Ans:

10	7	1	3

InsHeap(Ans, 4, A[5])
n = 5; p = 5;
p > 1 [Tata = 2
5 <= Ans[Tata] [Ans[5] = 5
exit

Ans:

10	7	1	3	5

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Expl

Vectorul initial A:

7	10	1	3	5	6	4	2	9	8

```
procedure InsHeap(Ans, n, Val)
    n ← n + 1
    p ← n
    While p > 1 do
        Tata ← p div 2
        If Val<= Ans[Tata] then
            Ans[p] ← Val
            exit
        Else
            Ans[p] ← Ans[Tata]
            p ← Tata
        endIf
    endWhile
    Ans[1] ← Val
endproc
```

```
InsHeap(Ans, 5, A[6])
n = 6; p = 6;
p > 1   Tata = 3
        6 > Ans[Tata]   Ans[6] = 1
                        p = 3
p > 1   Tata = 1
        6 <= Ans[Tata]  Ans[3] = 6
                        exit
```

Ans:

10	7	6	3	5	1

```
InsHeap(Ans, 6, A[7])
n = 7; p = 7;
p > 1   Tata = 3
        4 <= Ans[Tata]  Ans[7] = 4
                        exit
```

Ans:

10	7	6	3	5	1	4

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Expl

Vectorul initial A:

7	10	1	3	5	6	4	2	9	8

```
procedure InsHeap(Ans, n, Val)
    n ← n + 1
    p ← n
    While p > 1 do
        Tata ← p div 2
        If Val<= Ans[Tata] then
            Ans[p] ← Val
            exit
        Else
            Ans[p] ← Ans[Tata]
            p ← Tata
        endIf
    endWhile
    Ans[1] ← Val
endproc
```

```
InsHeap(Ans, 7, A[8])
n = 8; p = 8;
p > 1   Tata = 4
        2 <= 3 Ans[Tata]   Ans[8] = 2
                           exit
```

Ans:

10	7	6	3	5	1	4	2

```
InsHeap(Ans, 8, A[9])
n = 9; p = 9;
p > 1   Tata = 4
        9 > Ans[Tata]   Ans[9] = 3
                        p=4
p > 1   Tata = 2
        9 > Ans[Tata]   Ans[4] = 7
                        p=2
p > 1   Tata = 1
        9 <= Ans[Tata]  Ans[2] = 9
                        exit
```

Ans:

10	9	6	7	5	1	4	2	3

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Expl

Vectorul initial A:

7	10	1	3	5	6	4	2	9	8

```
procedure InsHeap(Ans, n, Val)
    n ← n + 1
    p ← n
    While p > 1 do
        Tata ← p div 2
        If Val<= Ans[Tata] then
              Ans[p] ← Val
              exit
        Else
              Ans[p] ← Ans[Tata]
              p ← Tata
        endIf
    endWhile
    Ans[1] ← Val
endproc
```

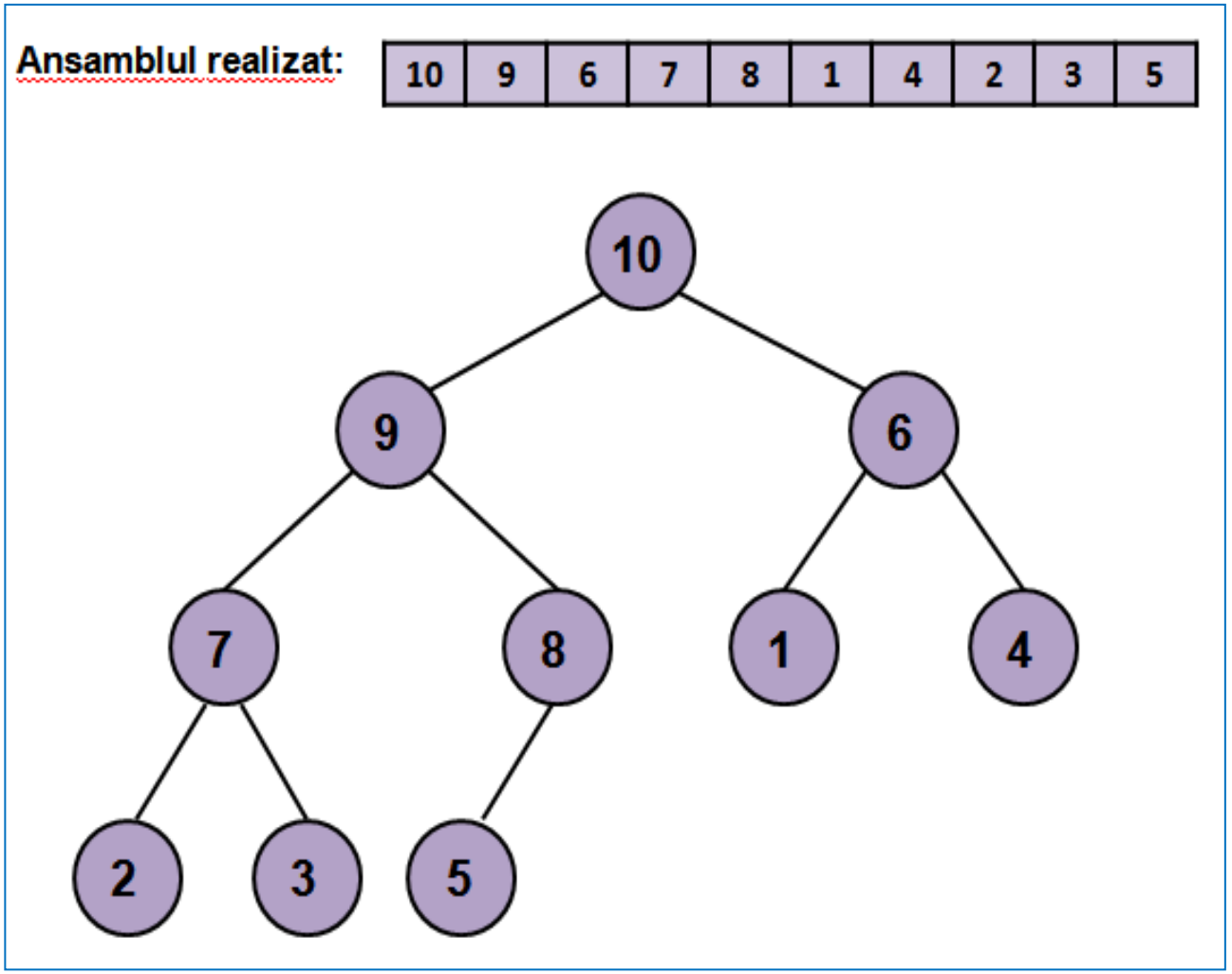
```
InsHeap(Ans, 9, A[10])
n = 10; p = 10;
p > 1   Tata = 5
        8 > Ans[Tata]   Ans[10] = 5
                        p=5
p > 1   Tata = 2
        8 <= Ans[Tata]  Ans[5] = 8
                        exit
```

Ans:

10	9	6	7	8	1	4	2	3	5

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Expl



Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Exemplificare pe arbore partial ordonat

Vectorul initial A:	7	10	1	3	5	6	4	2	9	8

```
procedure Asamblare (A,n)
    For j ← 1,n-1 do
InsHeap(A,j,A[j+1])
    endFor
endproc
```

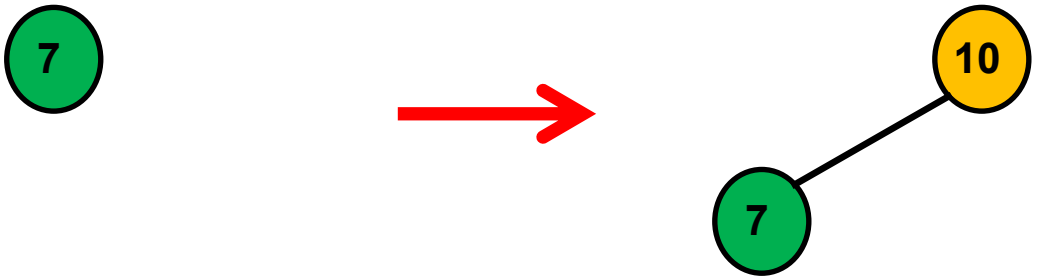
Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Exemplificare pe arbore partial ordonat

Ansamblu la pasul initial: | 7 |

InsHeap(Ans, 1, A[2])

Element de inserat: A[2] = 10



Ansamblu dupa inserare: | 10 | 7 |

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

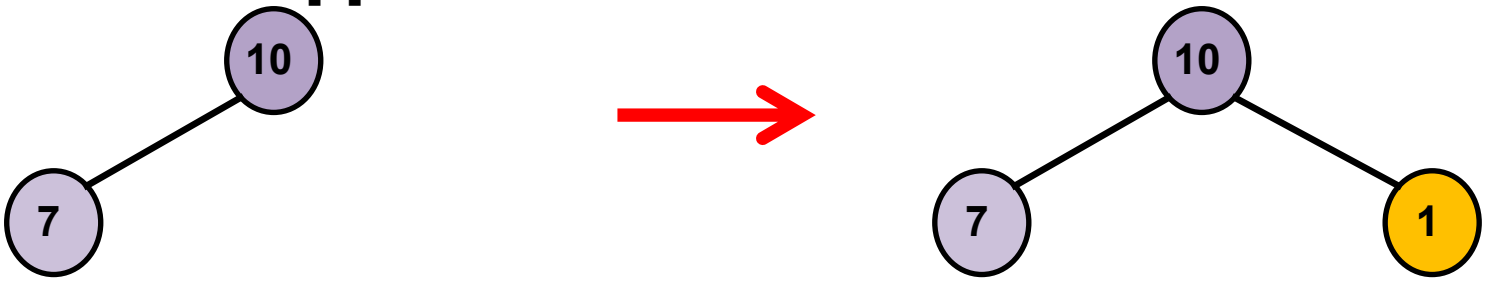
Exemplificare pe arbore partial ordonat

Ansamblu pas anterior:

10	7

InsHeap(Ans, 2, A[3])

Element de inserat: A[3] = 1



Ansamblu dupa inserare:

10	7	1

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

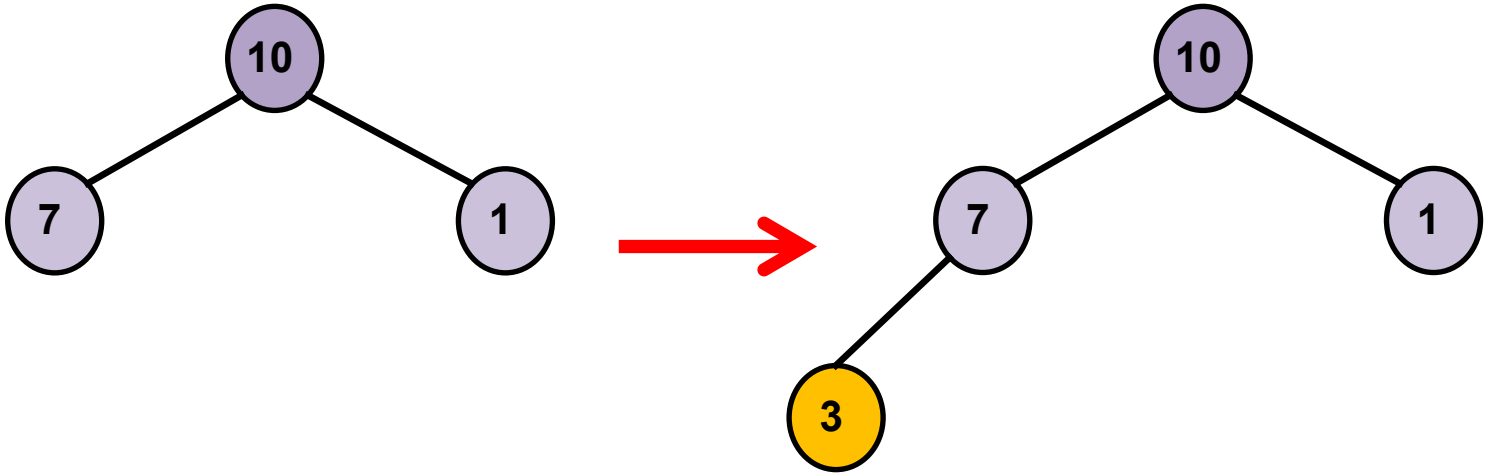
Exemplificare pe arbore partial ordonat

Ansamblu pas anterior:

10	7	1

InsHeap(Ans, 3, A[4])

Element de inserat: A[4] = 3



Ansamblu dupa inserare:

10	7	1	3

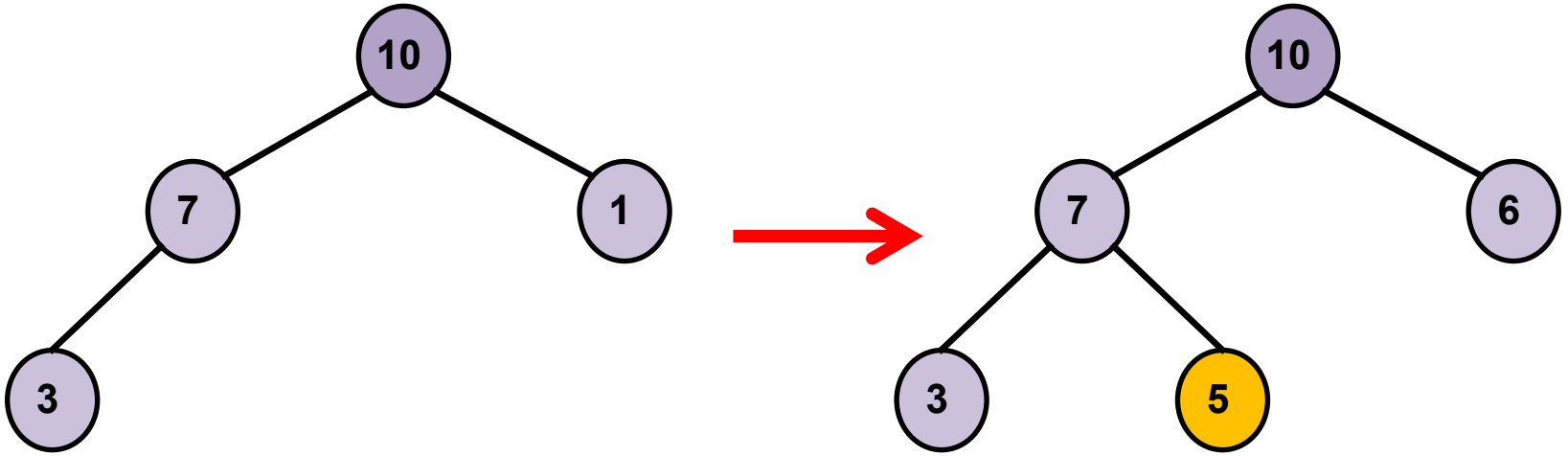
Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Exemplificare pe arbore partial ordonat

Ansamblu pas anterior:

10	7	1	3

Element de inserat: A[5] = 5 InsHeap(Ans, 4, A[5])



Ansamblu dupa inserare:

10	7	1	3	5

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

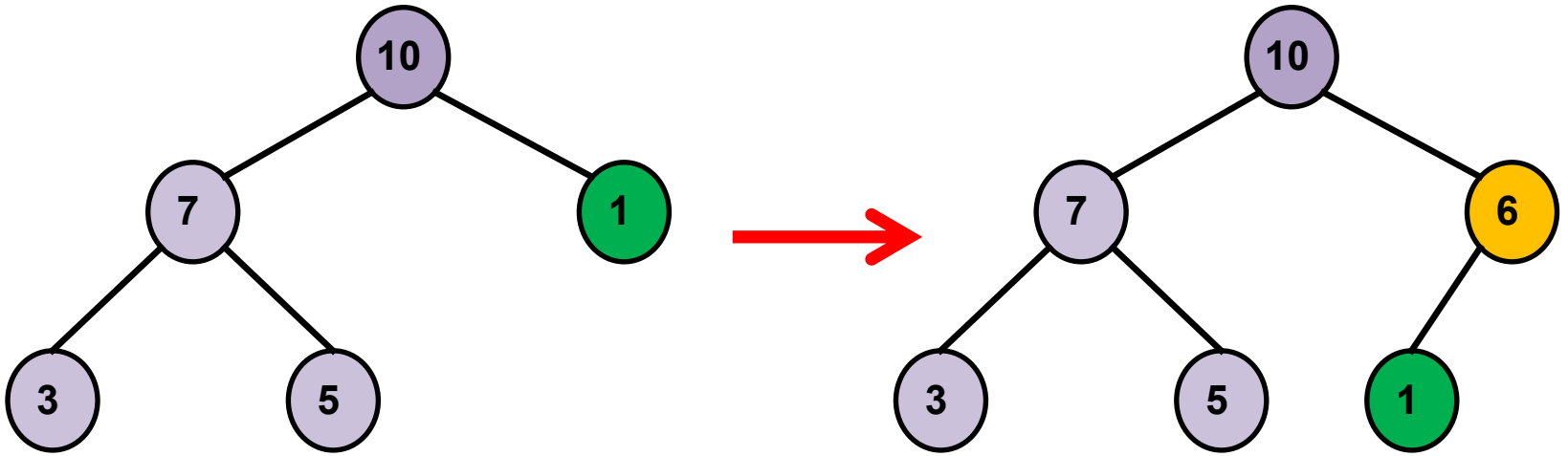
Exemplificare pe arbore partial ordonat

Ansamblu pas anterior:

10	7	1	3	5

Element de inserat: A[6] = 6

InsHeap(Ans, 5, A[6])



Ansamblu dupa inserare:

10	7	6	3	5	1

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

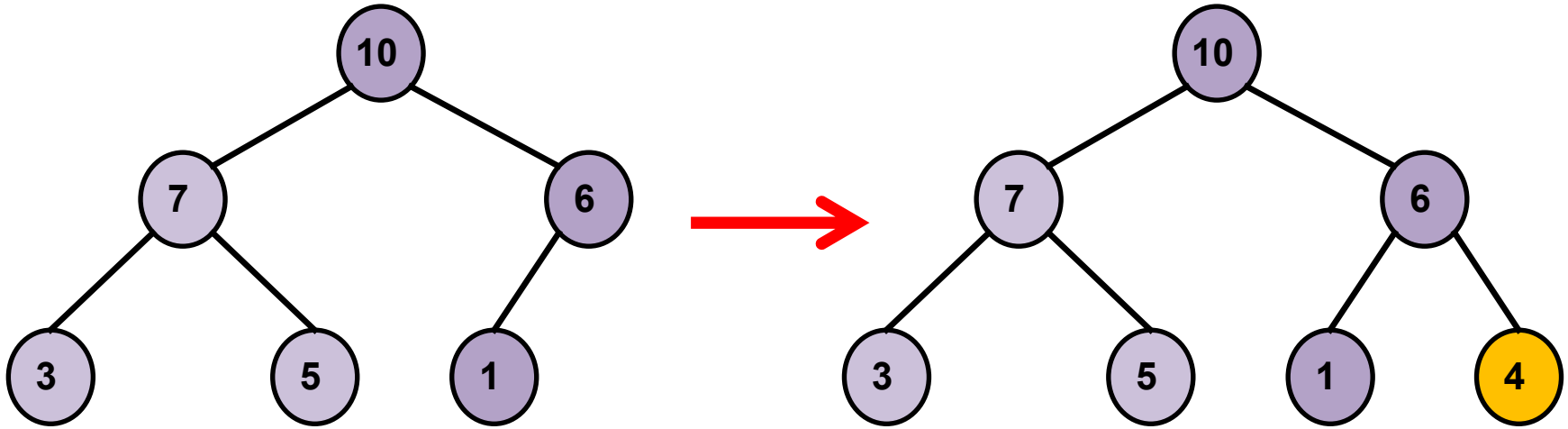
Exemplificare pe arbore partial ordonat

Ansamblu pas anterior:

10	7	6	3	5	1

Element de inserat: A[7] = 4

InsHeap(Ans, 6, A[7])



Ansamblu dupa inserare:

10	7	6	3	5	1	4

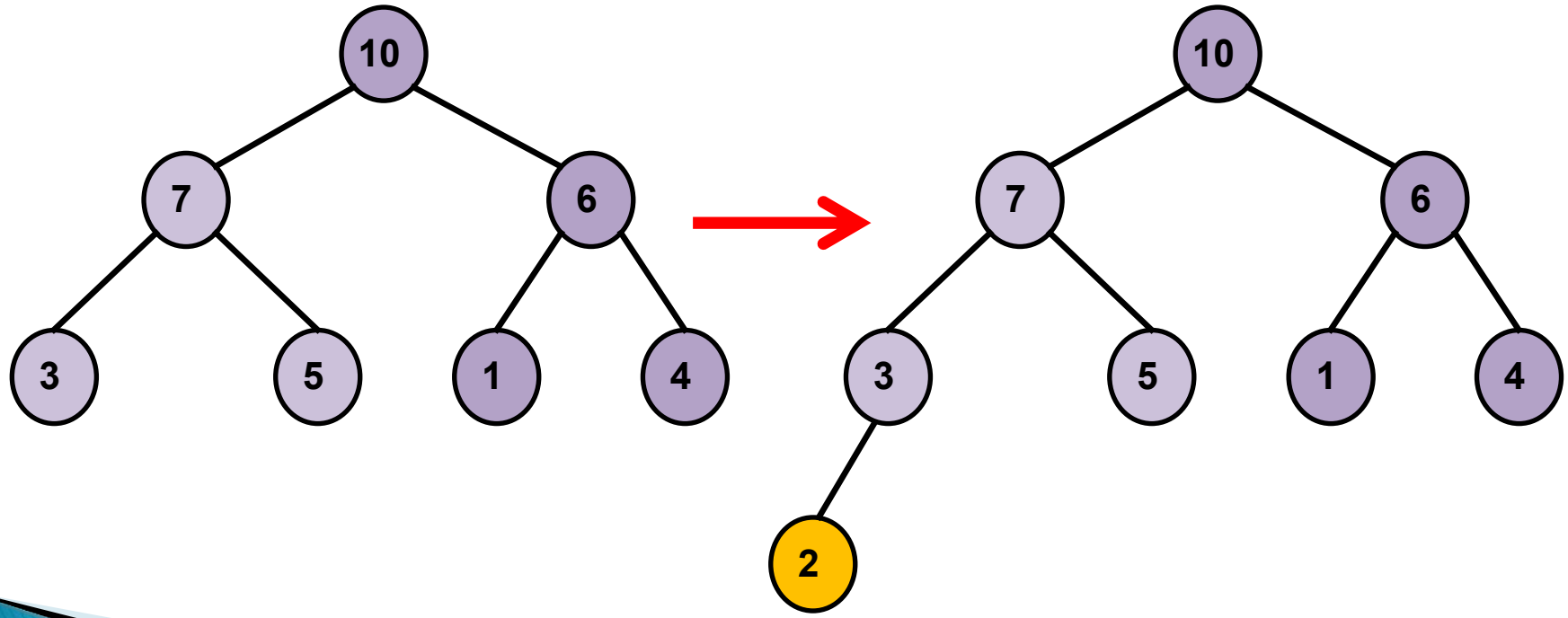
Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Exemplificare pe arbore partial ordonat

Ansamblu pas anterior:

10	7	6	3	5	1	4

Element de inserat: A[8] = 2 InsHeap(Ans, 7, A[8])



Ansamblu dupa inserare:

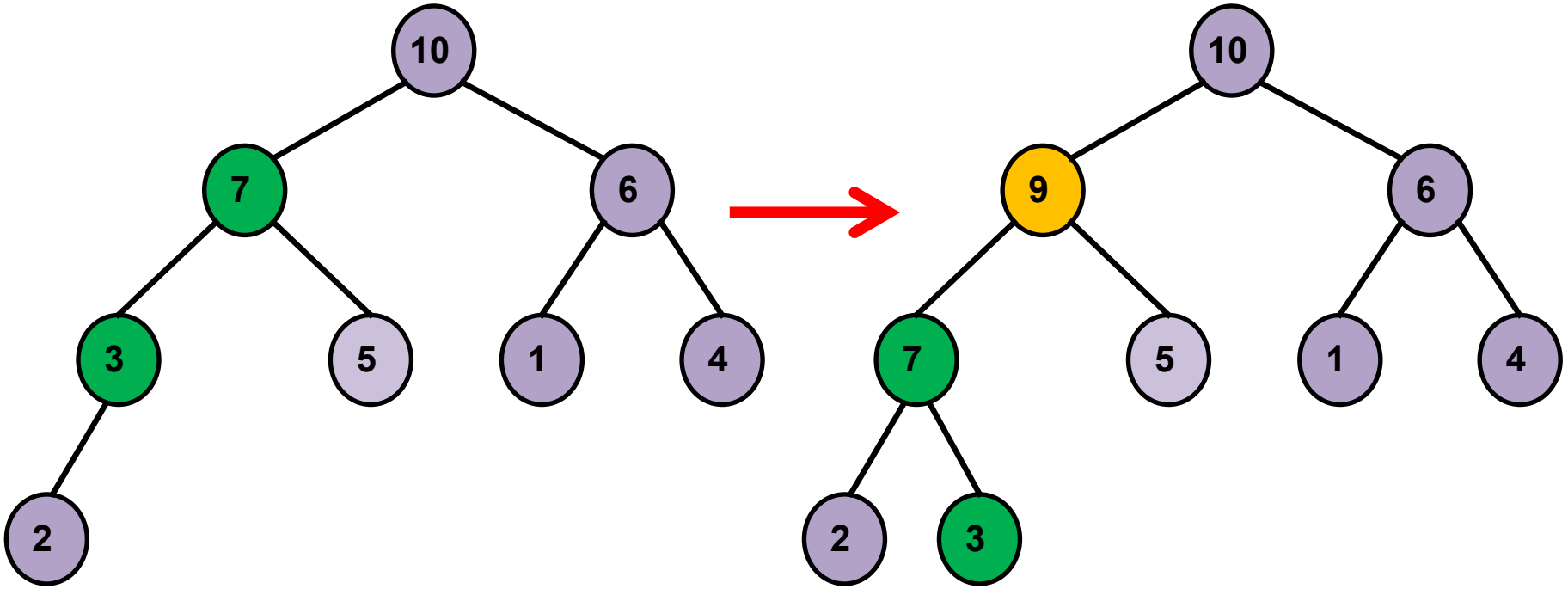
10	7	6	3	5	1	4	2

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Ansamblu pas anterior:

10	7	6	3	5	1	4	2

Element de inserat: A[9] = 9 InsHeap(Ans, 8, A[9])



Ansamblu dupa inserare:

10	9	6	7	5	1	4	2	3

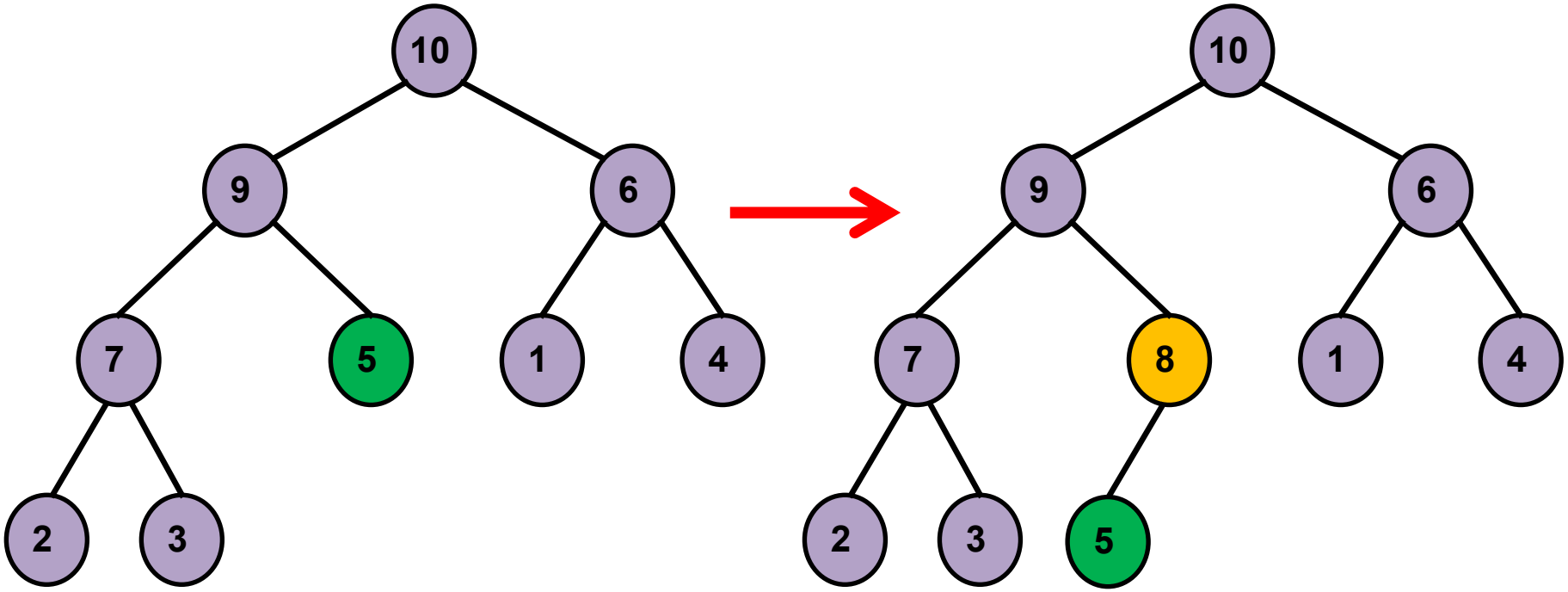
Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Ansamblu pas anterior:

10	9	6	7	5	1	4	2	3

Element de inserat: A[10] = 8

InsHeap(Ans, 9, A[10])



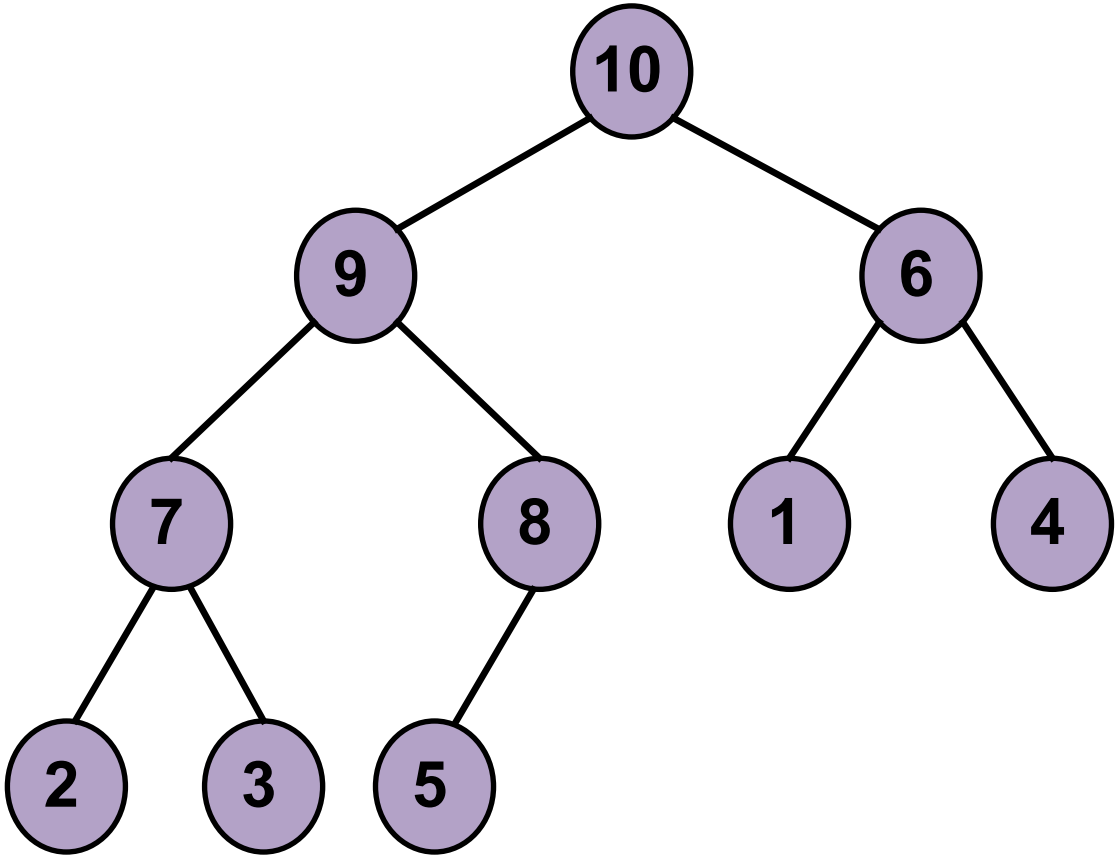
Ansamblu dupa inserare:

10	9	6	7	8	1	4	2	3	5

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Ansamblul realizat:

10	9	6	7	8	1	4	2	3	5



Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Extragerea maximului sau decapitarea unui ansamblu

1. Se extrage valoarea din radacina in vederea procesarii;

2. Se inlocuieste radacina cu ultimul nod (arborele binar ramane complet, dar eventual nu mai e max-ordonat)

3. Coboram noua radacina la locul ei prin comparatii cu cel mai mare dintre fii.

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

```
 void DelHeap (int Ans[], int n, int Val)
{ int Last, p, l, r;
   Val = Ans[1]; // extragerea radacinii
   Last = Ans[n]; // Last e ultimul elem din Ans ce trebuie inserat in Radacina,
                                                  // apoi trebuie sa cautam locul lui
   n--;                                            // scade dimensiunea ansamblului
   p = 1; l = 2; r = 3;          // p indice nod curent, l - fiu stang,r - fiu drept
   while (r <= n) // test de nedepasire a structurii
   {
      if (Last >= Ans[l]) && (Last >= Ans[r]))
      { Ans[p] = Last; // inserarea
         return; } // am terminat
      if (Ans[l] >= Ans[r]) // continuam pe ramura stanga
         { Ans[p] = Ans[l];
            p = l; } // reactualizarea lui p
         else // continuam pe ramura dreapta
         { Ans[p] = Ans[r];
            p = r; // reactualizarea lui p
         }
      l = 2*p; r = l+1; // actualizarea fiilor lui p
   }
   if ((l==n) && (Last < Ans[l]))
   {
      Ans[p] = Ans[l];
      p = l;
   }
```

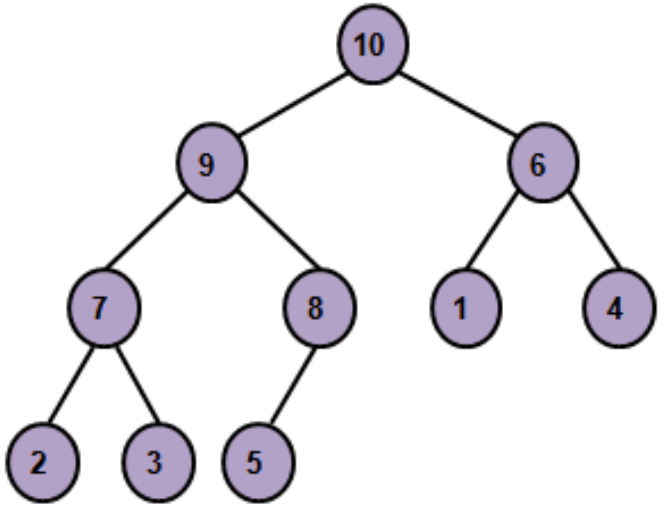
Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Extragerea maximului (Decapitarea unui ansamblu)

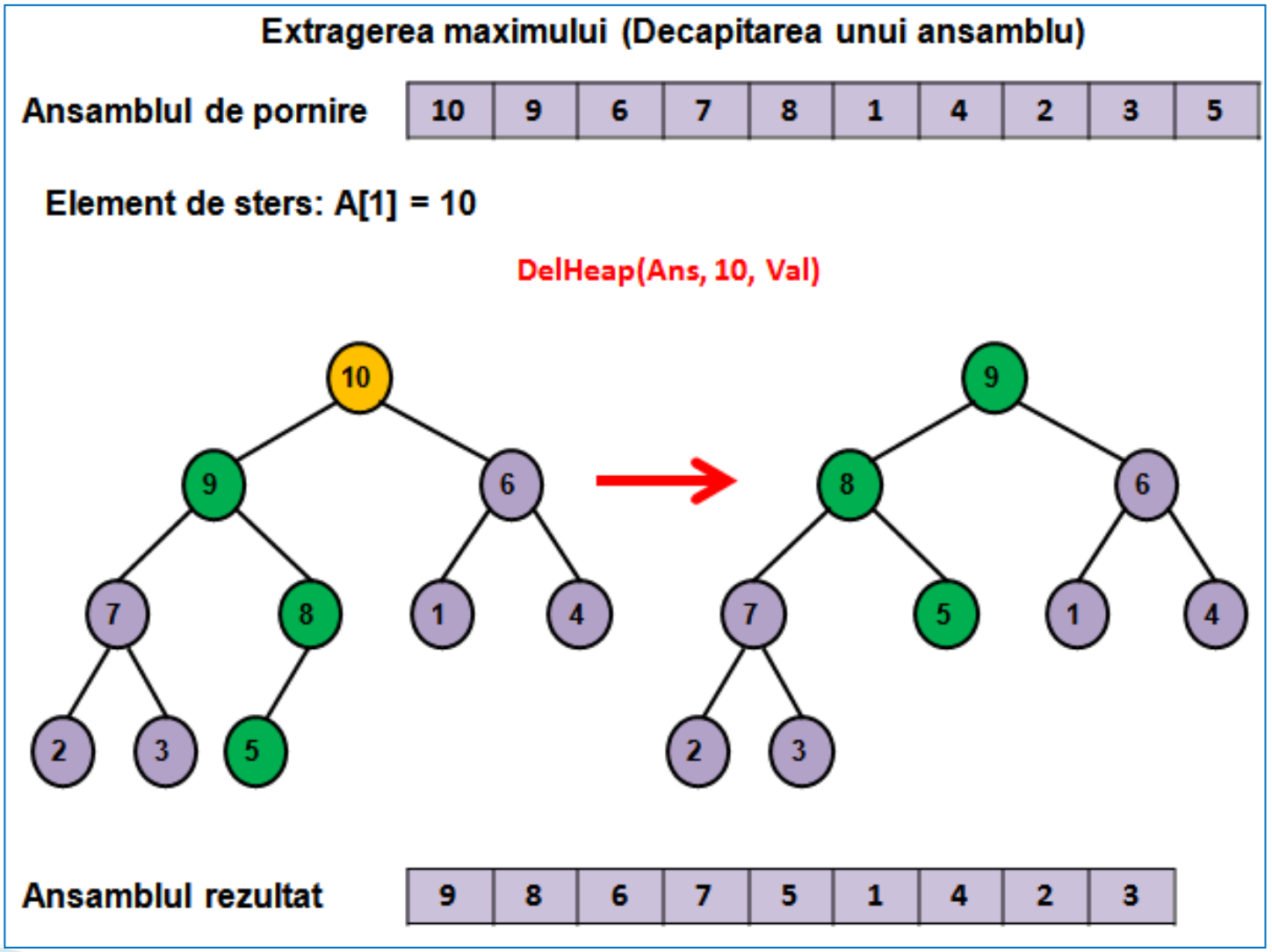
```
procedure DelHeap (Ans, n, Val)
    Val ← Ans[1]
    Last ← Ans[n]
    n ← n-1
    p ← 1; l ← 2; r ← 3
    while (r<=n) do
        if(Last>=Ans[l]) and (Last>=Ans[r]) then
                Ans[p] ← Last
                exit
        endif
        if Ans[l]>=Ans[r] then
                Ans[p] ← Ans[l]
                p ← l
        else
                Ans[p] ← Ans[r]
                p ← r
        endif
        l ← 2*p; r ← l+1;
    endwhile
    if(l==n) and (Last<Ans[l]) then
                Ans[p] ← Ans[l]
                p ← l
    endif
    Ans[p] ← Last
endproc
```

Ansamblul de pornire

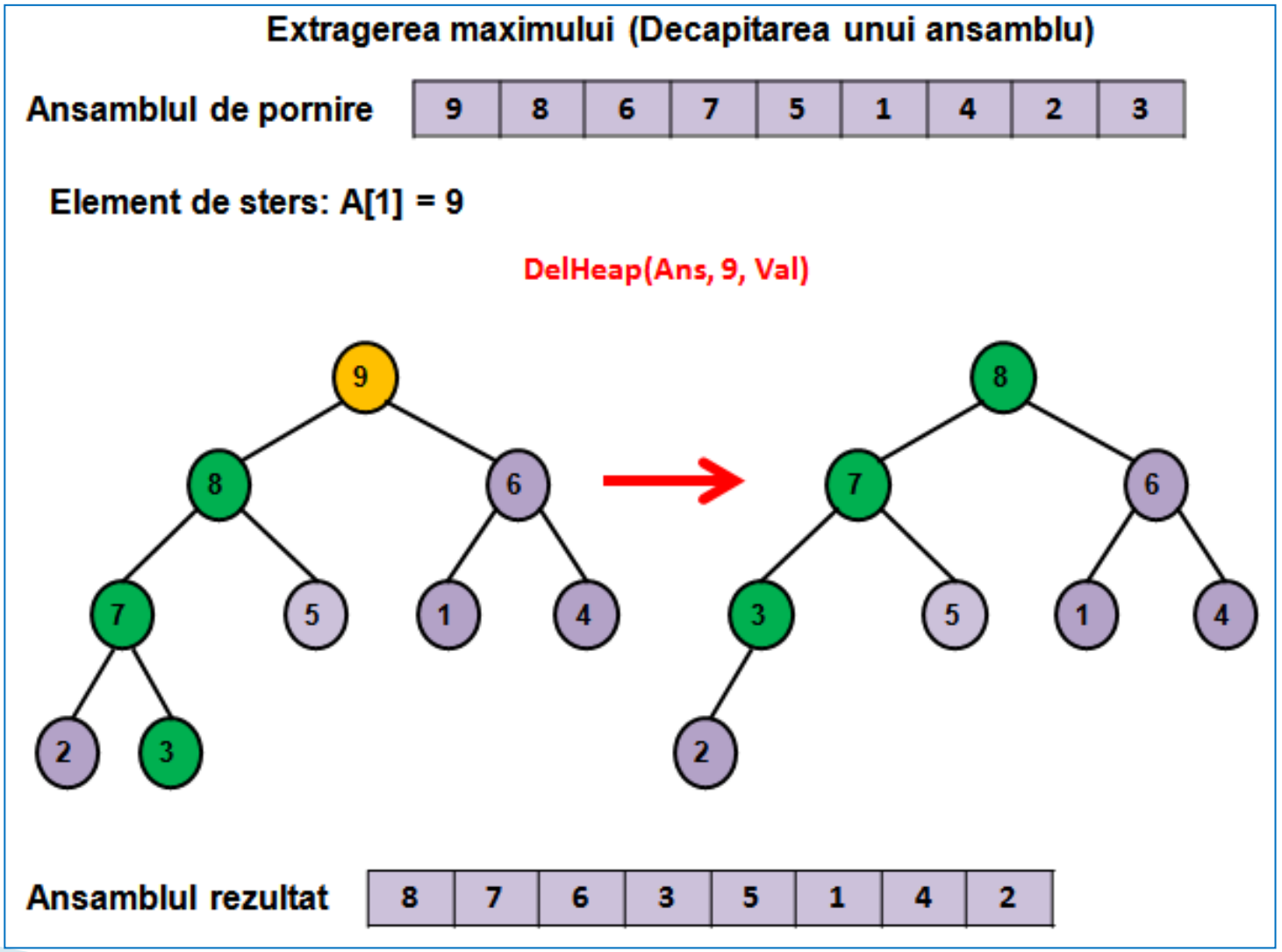
10	9	6	7	8	1	4	2	3	5



Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)



Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)



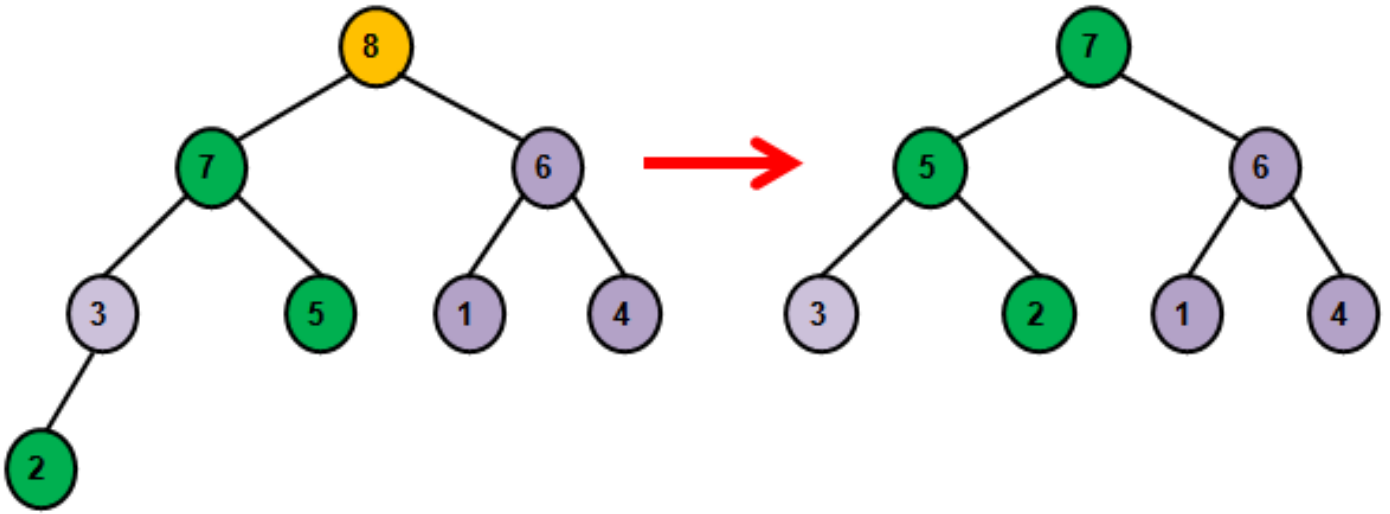
Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Extragerea maximului (Decapitarea unui ansamblu)

Ansamblul de pornire	8	7	6	3	5	1	4	2

Element de sters: A[1] = 8

DelHeap(Ans, 8, Val)



Ansamblul rezultat	7	5	6	3	2	1	4

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

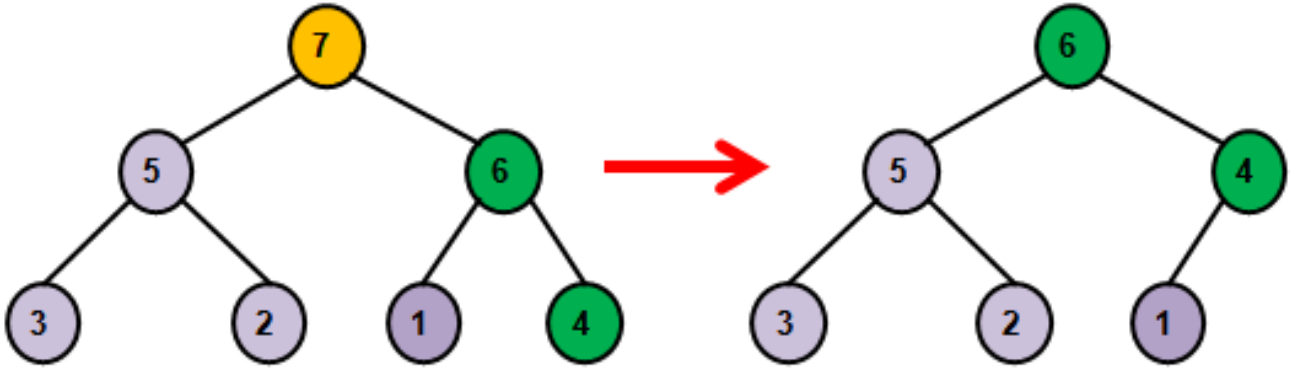
Extragerea maximului (Decapitarea unui ansamblu)

Ansamblul de pornire

7	5	6	3	2	1	4

Element de sters: A[1] = 7

DelHeap(Ans, 7, Val)



Ansamblul rezultat

6	5	4	3	2	1

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

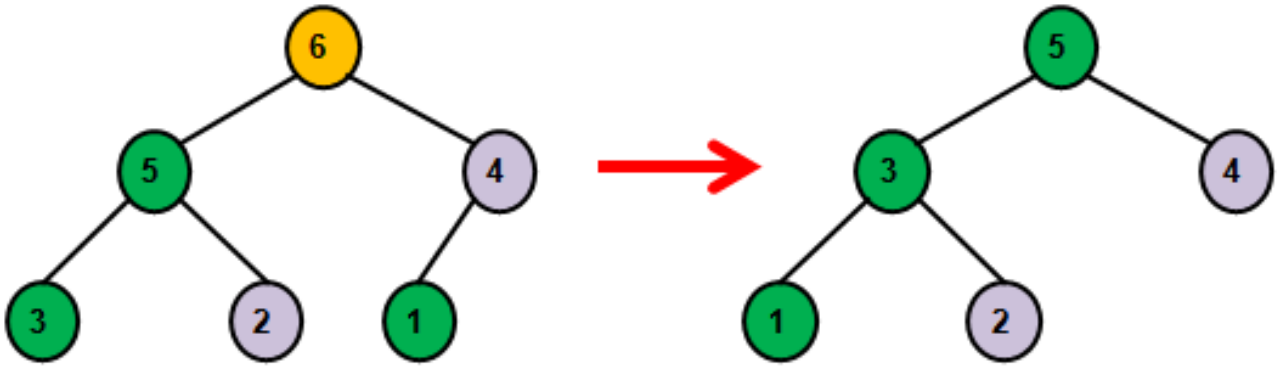
Extragerea maximului (Decapitarea unui ansamblu)

Ansamblul de pornire

6	5	4	3	2	1

Element de sters: A[1] = 6

DelHeap(Ans, 6, Val)



Ansamblul rezultat

5	3	4	1	2

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

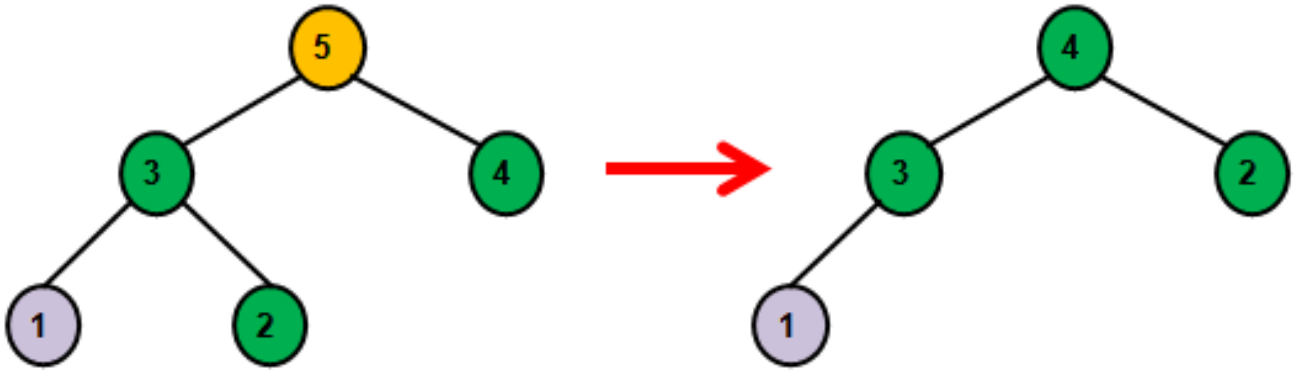
Extragerea maximului (Decapitarea unui ansamblu)

Ansamblul de pornire

5	3	4	1	2

Element de sters: A[1] = 5

DelHeap(Ans, 5, Val)



Ansamblul rezultat

4	3	2	1

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

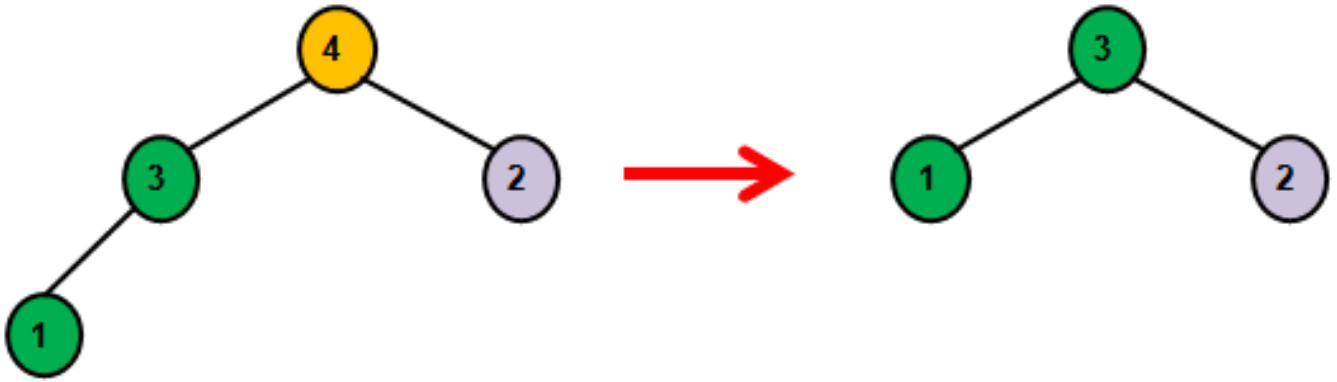
Extragerea maximului (Decapitarea unui ansamblu)

Ansamblul de pornire

4	3	2	1

Element de sters: A[1] = 4

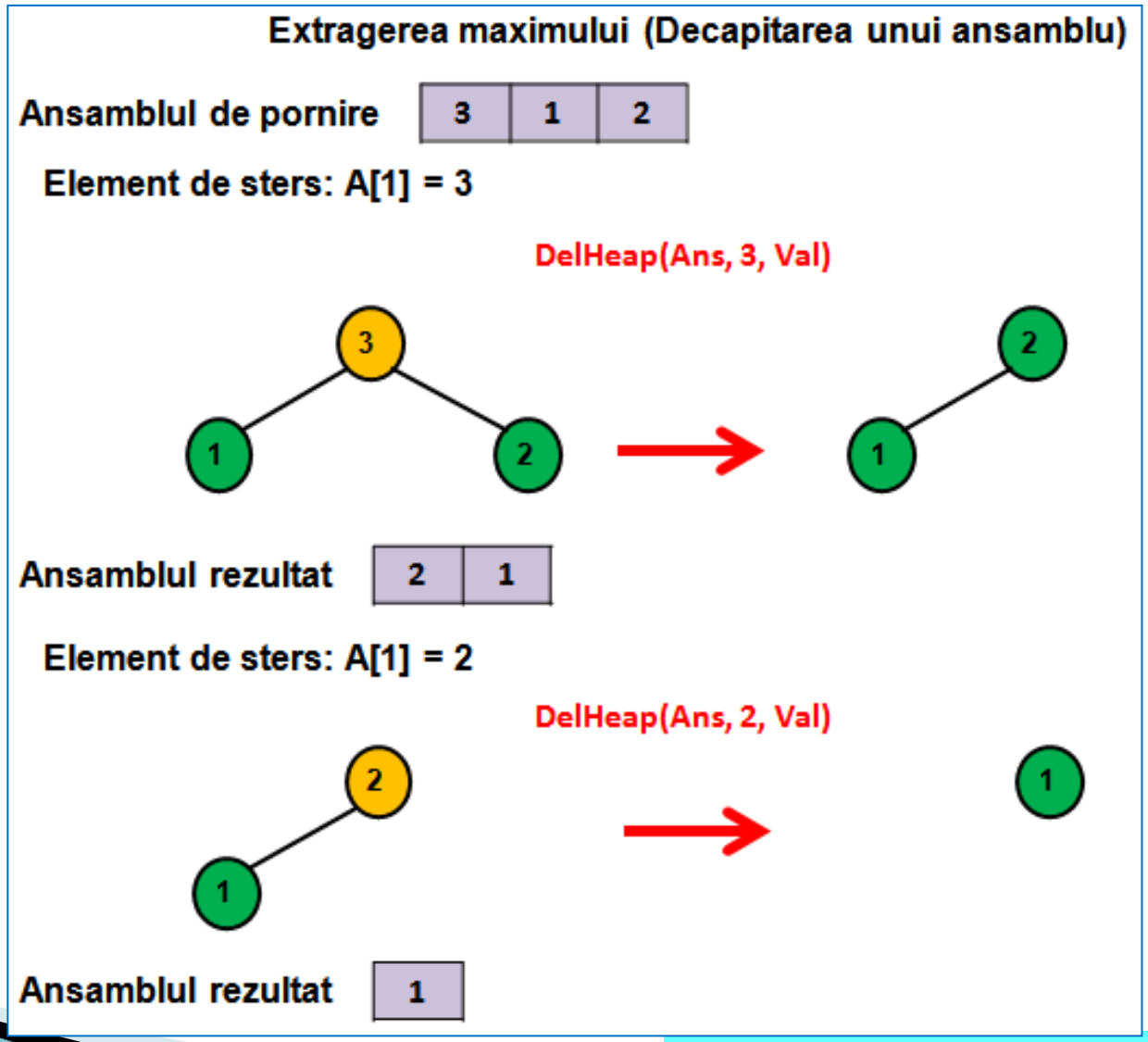
DelHeap(Ans, 4, Val)



Ansamblul rezultat

3	1	2

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)



Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Complexitatea operatiilor intr-un ansamblu

Sortarea cu ansamble

1. (Pas 0) Se asambleaza vectorul A[1..n]. Maximul va pe A[1];

2. (Pas 1) Se decapiteaza ansamblul A[1..n], cu reasamblarea lui A[1..n-1] si se pune maximul pe A[n];

3. (Pas j) Se decapiteaza ansamblul A[1..n-j+1], cu reasamblarea lui A[1..n-j] si se pune maximul pe A[n-j+1].

Dupa n-1 pasi iterativi vectorul A este sortat.

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Complexitatea operatiilor intr-un ansamblu

```
void HeapSort(int Ans[], int n)  // sortam vectorul A[1..n]
{
  Asamblare (A,n);
  while (n > 1)
  {
    DelHeap (A,n,Val);
    A[n+1] = Val;
  }
 }
```

Complexitate: $O(n \lg n)$.

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Varianta recursiva (Cormen)

Heaps (ansamble)

O structura de date de tip heap/ansamblu (binar) este un vector ce poate vazut ca un arbore binar aproape complet (1).

Fiecare cheie din arbore corespunde unui element din vector.

Arborele este complet pe toate nivelele cu exceptia celui mai de jos, unde exista chei incepand din stanga pana la un anumit punct.

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Varianta recursiva (Cormen)

Heaps (ansamble)

Un vector A asociat unui ansamblu este caracterizat de 2 atribute:

-A.length (numarul elementelor din vector);

- A.heap-size (numarul de elemente din "heap") (chiar daca A[1 … A.length] contine numere, doar elementele din A[1 … A.heap-size], unde 0 <= A.heap-size <= A.length), sunt elemente valide in "heap".

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Varianta recursiva (Cormen)

Heaps (ansamble)

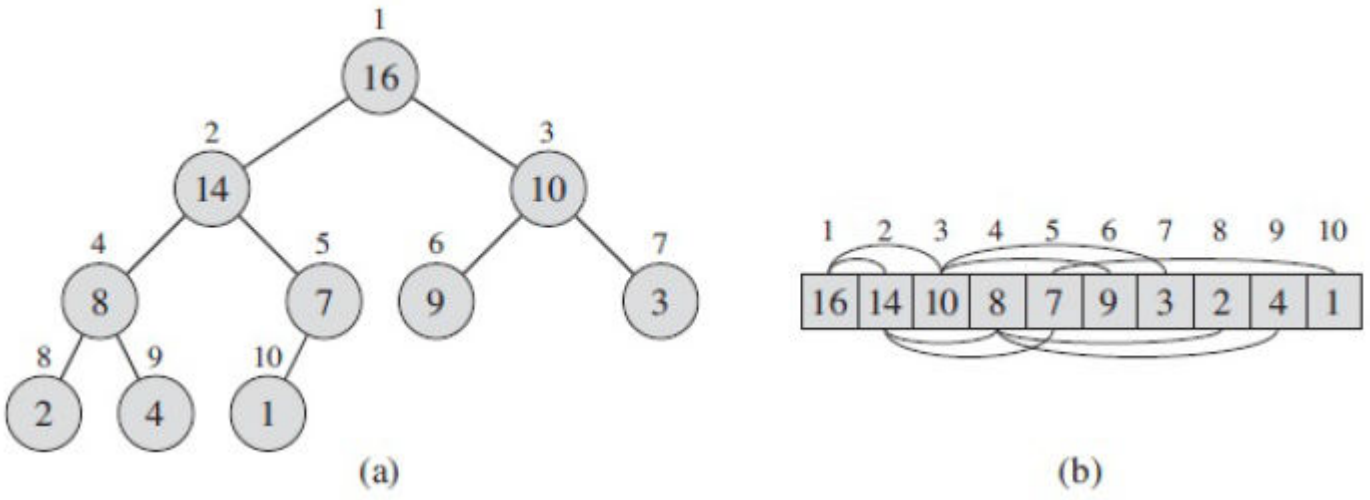
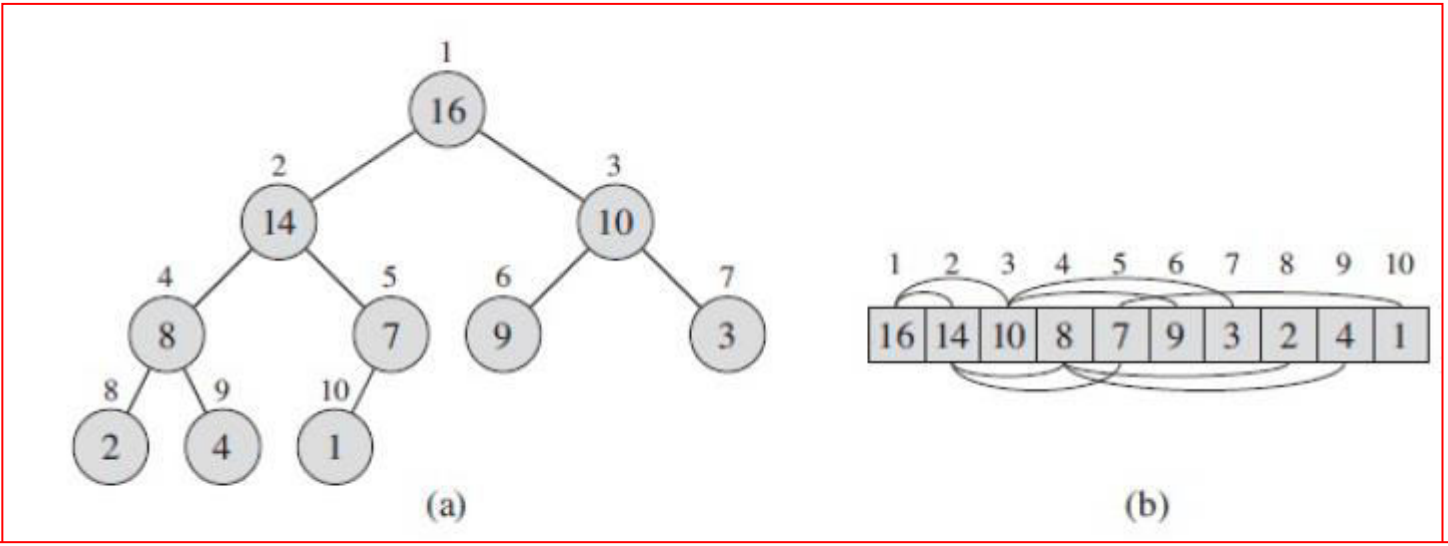


Figure : Un max-ansamblu vazut ca (a) arbore binar si (b) vector. In interiorul cercurilor sunt valorile fiecarui nod. Numarul de deasupra este indicele corespunzator in vector. Deasupra si sub vector sunt trasate liniile aratand relatiile tata-fiu. Parintii sunt intotdeauna la stanga descendentilor. Inaltimea arborelui este 3; nodul cu indicele 4 (avand valoarea 8) are inaltimea 1.

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Varianta recursiva (Cormen)

Heaps (ansamble)



root = A[1];
Pentru orice indice i al unui nod, se poate calcula usor indicii parintelui sau si al descendentilor sai stang si drept:

```
void PARENT(int i) {return i/2;}
void LEFT(int i) {return 2*i;}
void RIGHT(int i) {return 2*i+1;}
```

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Varianta recursiva (Cormen)

Heaps (ansamble)

Sunt 2 tipuri de **arbori binari partiali**:

- **max - heap (max-ordonat)** (oricare nod i != root) A[PARENT(i)] >= A[i])
  Maximul elementelor se afla in root

- **min - heap (min-ordonat)** (oricare nod i != root) A[PARENT(i)] <= A[i])
  Minimul elementelor se afla in root

Pentru algoritmul **HeapSort** va folosita o structura de tip arbore partial max-ordonat.

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Varianta recursiva (Cormen)

Heaps (ansamble)

Se defineste **inaltimea unui nod intr-un heap**, drept numarul de muchii al celei mai lungi cai directe de la nod la o frunza.

Inaltimea unui ansamblu este inaltimea radacinii sale.

Cum un ansamblu cu n elemente se bazeaza pe un arbore binar complet, inaltimea lui este $\Theta(\lg n)$. Se poate demonstra ca operatiile de baza pe un ansamblu ruleaza in timp cel mult proportional cu inaltimea arborelui, deci $\mathcal{O}(\lg n)$.

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Varianta recursiva (Cormen)

Heaps (ansamble)

Proceduri de baza folosite intr-un algoritm de sortare si o structura de date de tip coada cu prioritati:

- **MAX-HEAPIFY**: - timp $\mathcal{O}(\lg n)$ - pastreaza proprietatea de arbore max-ordonat;
- **BUILD-MAX-HEAP**: - timp $\mathcal{O}(n)$ - creaza un arbore max-ordonat pornind de la un vector nesortat;
- **HEAPSORT**: - timp $\mathcal{O}(n \lg n)$ - sorteaza un vector;

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Varianta recursiva (Cormen)

Procedura MAX - HEAPIFY

```
void MAX-HEAPIFY (int A[], int i)
{ int l = LEFT(i);
  int r = RIGHT(i);
  if ((l <= A.heap-size) && (A[l] > A[i]))
    largest = l;
  else
    largest = i;
  if ((r <= A.heap-size) && (A[r] > A[largest]))
    largest = r;
  if (largest != i)
    // interschimba(A[i],A[largest]);
  MAX-HEAPIFY (A, largest)
}
```

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Varianta recursiva (Cormen)

Procedura MAX - HEAPIFY

La fiecare pas se determina maximul dintre A[i], A[LEFT(i)] si A[RIGHT(i)].

Variabila largest stocheaza indicele elementului maxim.

Daca maximul este A[i] atunci subarborele cu radacina in nodul i este deja max-ordonat si procedura se termina.

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Varianta recursiva (Cormen)

Procedura MAX - HEAPIFY

Daca unul din cei doi descendenti contine maximul, A[i] se interschimba cu A[largest], nodul i si descendentii sai satisfacand conditia de arbore partial max-ordonat.

Daca subarborele avand drept radacina nodul indexat de variabila largest nu este max-ordonat, procedura MAX-HEAPIFY se apeleaza recursiv pe acest sub-arbore.

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Procedura MAX - HEAPIFY

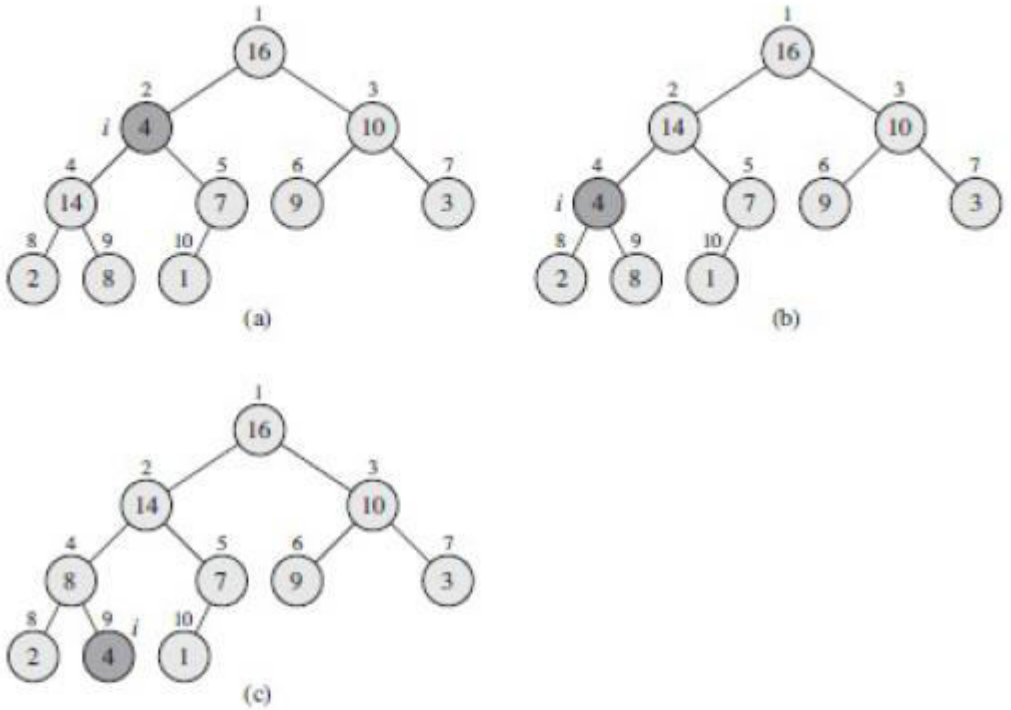


Figure : Apelul *MAX-HEAPIFY(A,2)*, unde A.heap-size = 10.
(a) Configuratia initiala A[2] violeaza conditia de arbore max-ordonat;
(b) Interschimbare(A[2],A[4]): A[2] respecta conditia, dar A[4] nu, deci se apeleaza recursiv *MAX-HEAPIFY(A,4)*; (c) Interschimbare(A[4],A[9]) si apel recursiv *MAX-HEAPIFY (A,9)* - nu se mai fac alte schimbari.

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Varianta recursiva (Cormen)

Analiza costurilor

Timpul de executie a procedurii MAX-HEAPIFY pe un sub-arbore de inaltime **n** cu radacina intr-un nod **i** este dat de:

- $\Theta(1)$: costul stabilirii unei relatii intre $A[i]$, $A[LEFT(i)]$ si $A[RIGHT(i)]$;
- Timpul de rulare a procedurii pe un sub-arbore cu radacina intr-un descendent al nodului i (presupunand ca apelul recursiv are loc).

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Varianta recursiva (Cormen)

Analiza costurilor

Sub-arborii descendenti au inaltimea cel mult $2n/3$ (cazul cel mai defavorabil avand ultimul nivel pe jumatate plin), deci putem da urmatoarea relatie de recurenta:
$T(n) \leq T(2n/3) + \Theta(1)$.

Solutia acestei recurente, obtinuta prin cazul 2 al Teoremei Master, este $T(n) = \mathcal{O}(\lg n)$. Alternativ, se poate spune ca timpul de executie a procedurii MAX-HEAPIFY pe un nod de inaltime h este de ordinul $\mathcal{O}(h)$. Figura 3 prezinta un exemplu de apel al acestei proceduri.

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Varianta recursiva (Cormen)

Construirea unui ansamblu

Se foloseste procedura MAX-HEAPIFY de jos in sus pentru a transforma un vector de lungime n intr-un ansamblu.

Elementele A[[n/2] + 1, n] sunt frunze, deci se incepe cu un ansamblu de lungime 1.

BUILD-MAX-HEAP parcurge nodurile ramase in ordine inversa, de la n/2 la 1, si apeleaza **MAX-HEAPIFY**.

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Varianta recursiva (Cormen)

Construirea unui ansamblu

```
void BUILD-MAX-HEAP (int A[])
  { A.heap-size = A.length;
    for (i = A.length/2; i >= 1; i--)
    MAX-HEAPIFY(A,i)
  }
```

Costul fiecarui apel MAX-HEAPIFY este O(lg n).

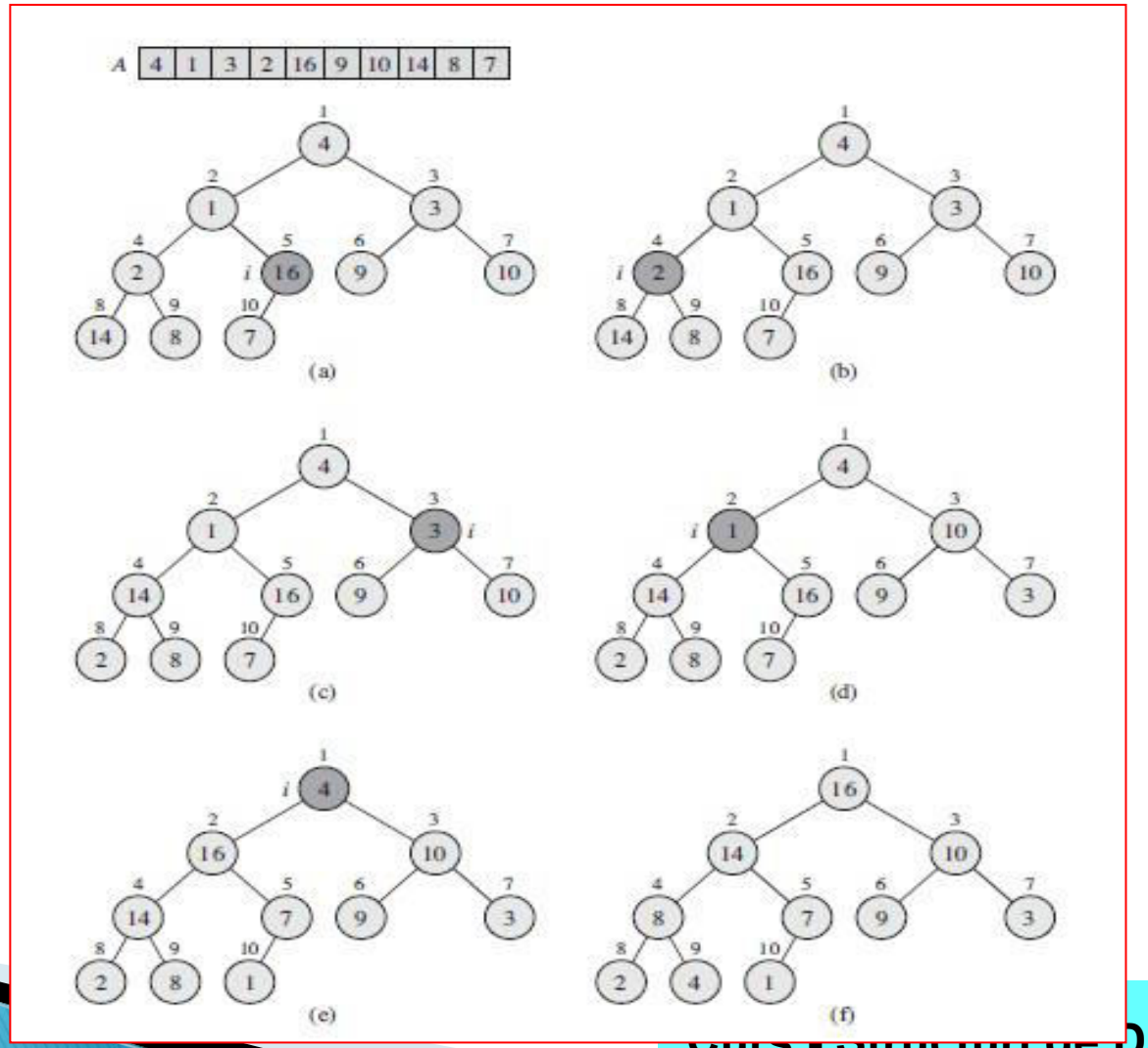
BUILD-MAX-HEAP face O(n) asemenea apeluri.

Concluzia, complexitatea este O(n lg n).

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Varianta recursiva (Cormen)

Construirea unui ansamblu



Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Varianta recursiva (Cormen)

HeapSort

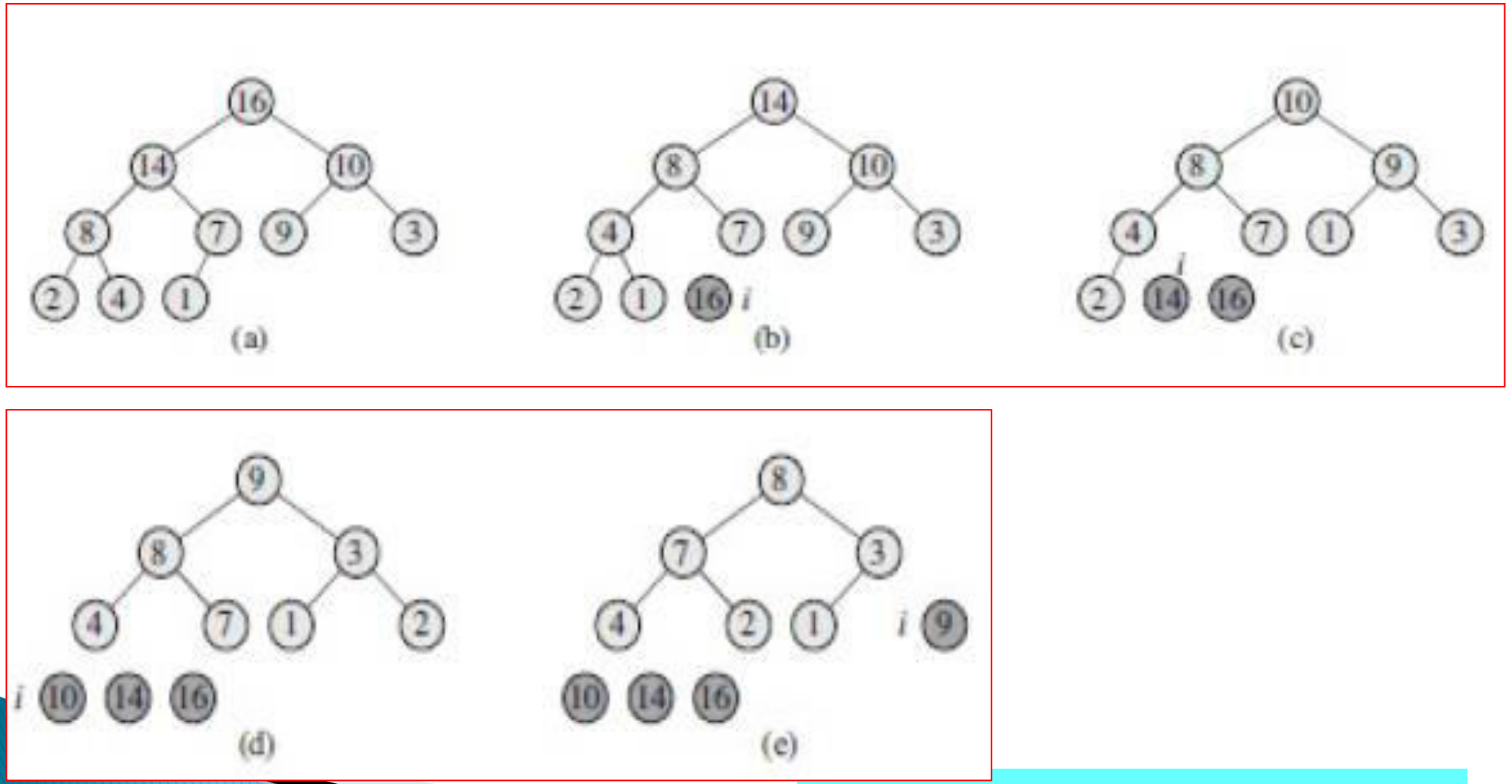
```
    HEAPSORT(A)
1     BUILD-MAX-HEAP(A)
2     for i = A.length downto 2
3         exchange A[1] with A[i]
4         A.heap-size = A.heap-size - 1
5         MAX-HEAPIFY(A,1)
```

```
void HEAPSORT(int A[])
{
  BUILD-MAX-HEAP(A);
  for (i = A.length; i >= 2; i--)
     interschimba(A[1],A[i])
  A.heap-size --;
  MAX-HEAPIFY(A,1)
}
```

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Varianta recursiva (Cormen)

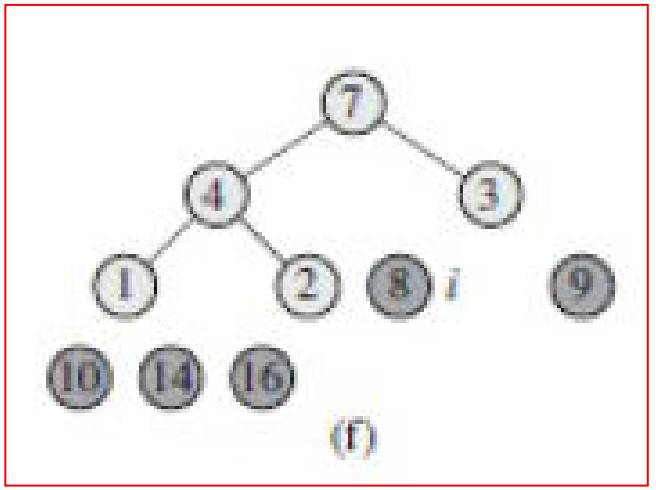
HeapSort

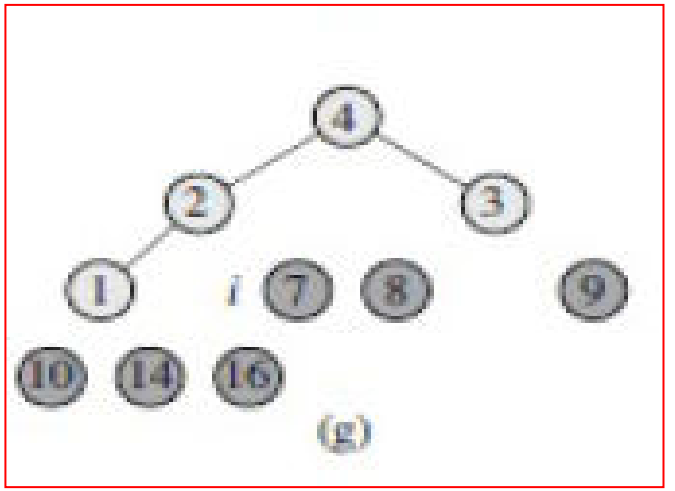


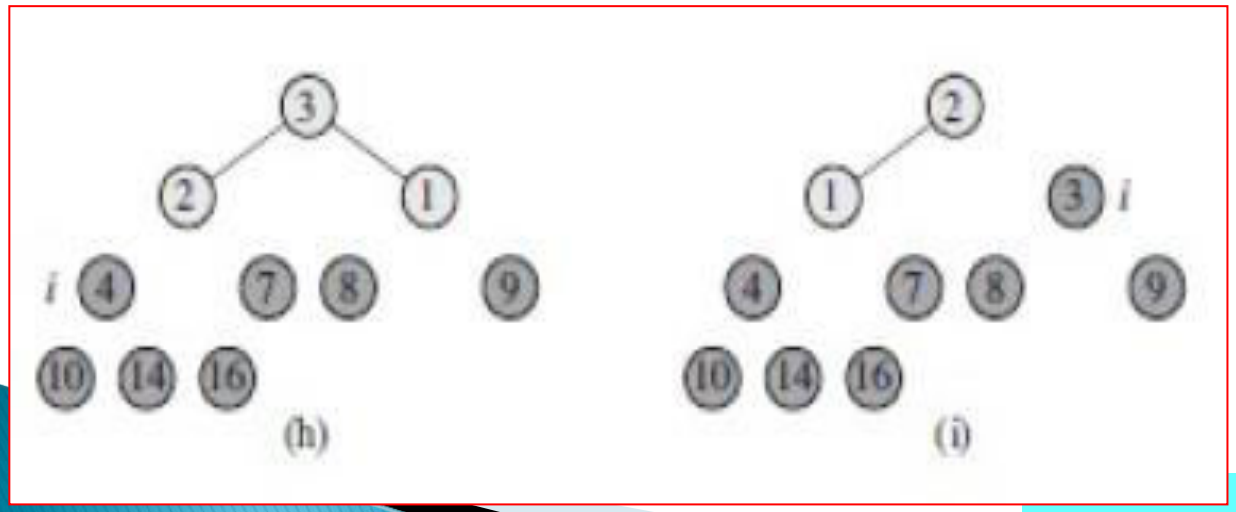
Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Varianta recursiva (Cormen)

HeapSort







Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Varianta recursiva (Cormen)

HeapSort

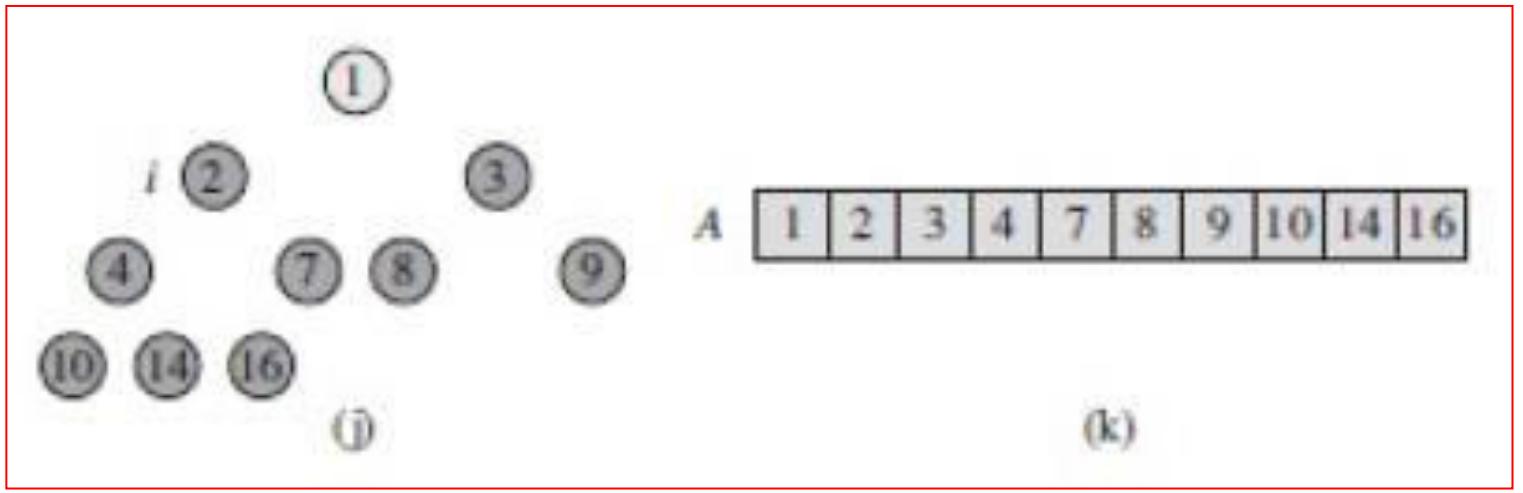


Figure : (a) Ansamblul max-ordonat obtinut prin BUILD-MAX-HEAP; (b)-(j) Ansamblul dupa ecare apel al MAX-HEAPIFY: doar nodurile cu gri deschis raman; (k) vectorul sortat rezultat

Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

Varianta recursiva (Cormen)

HeapSort

Procedura HEAPSORT are complexitatea de $\mathcal{O}(n \lg n)$ deoarece apelul procedurii BUILD-MAXHEAP are $\mathcal{O}(n)$ si fiecare din cele n-1 apeluri ale procedurii MAX-HEAPIFY se face in timp $\mathcal{O}(\lg n)$.

Limita inferioara a performantei algoritmilor de sortare bazati pe comparatii intre chei

Clasa algoritmilor de sortare bazati pe comparatii intre chei:

- sortarea directă prin inserţie
- sortarea directă prin selecţia minimului (sau maximului)
- sortarea directă prin interschimbare
- sortarea rapidă (QuickSort)
- sortarea cu ansamble (HeapSort)

Complexitatea algoritmilor de mai sus se estimează în mod esenţial în funcţie de **numărul de comparaţii** intre elementele multimii de sortat

Limita inferioara a performantei algoritmilor de sortare bazati pe comparatii intre chei

-primii trei algoritmi de mai sus au o complexitate de ordinul lui n^2

-sortarea rapidă şi sortarea cu ansamble au performanţe de ordinul lui $n \log_2 n$.

Aratam in continuare ca **NU** se poate cobori sub acesta limita (in aceasta clasa).

Limita inferioara a performantei algoritmilor de sortare bazati pe comparatii intre chei

Teoremă. Orice algoritm de sortare a n chei, algoritm bazat pe comparaţii între chei, va face cel puţin:

$\lceil \log_2 n! \rceil$ comparaţii în cazul cel mai nefavorabil;

$\lfloor \log_2 n! \rfloor$ comparaţii în cazul mediu.

Demonstraţie. Consideram algoritmii

- datele de intrare sunt vectori de lungime n, $(x_1, x_2, \ldots, x_n)$, cu componentele presupuse distincte.

- bazaţi pe comparaţii de tipul $x_i < x_j$.

Limita inferioara a performantei algoritmilor de sortare bazati pe comparatii intre chei

Unui asemenea algoritm i se asociază un arbore binar de decizie, care va fi arbore binar strict:

(a)o intrucţiune de ieşire va fi o frunză ce conţine vectorul sortat;

(b)o comparaţie $x_i < x_j$ va fi un nod interior, al cărui fiu stâng este arborele asociat instrucţiunilor ce se execută dacă $x_i < x_j$ este adevărată, iar fiul drept este arborele asociat instrucţiunilor ce se execută dacă $x_i < x_j$ este falsă.

Numărul total al instrucţiunilor de ieşire şi deci al frunzelor va fi numărul total de ordini posibile pe vectorul $(x_1, \ldots, x_n)$, deci $n!$.

Limita inferioara a performantei algoritmilor de sortare bazati pe comparatii intre chei

Exemplu: arbore binar de decizie asociat unui algoritm de sortare a unei multimi cu n=3 elemente

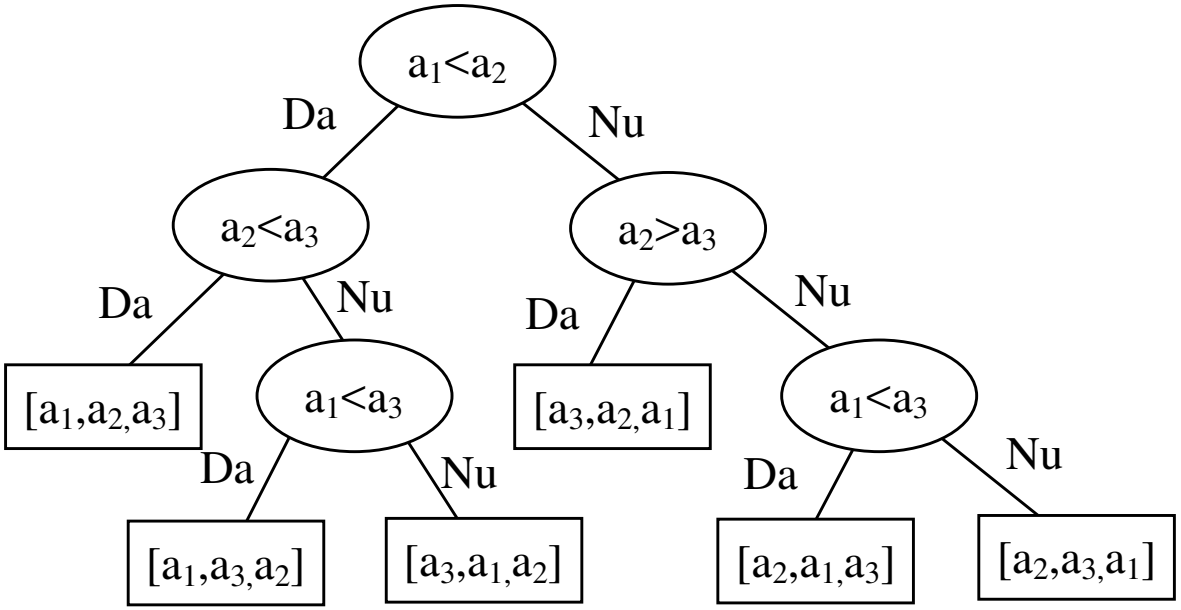


Fig.4.1.6 Arbore binar strict asociat sortării mulţimii $\{a_1, a_2, a_3\}$.

Limita inferioara a performantei algoritmilor de sortare bazati pe comparatii intre chei

Pe un set particular de date de intrare, acţiunea algoritmului este echivalentă cu parcurgerea unui drum de la rădăcină până la frunza ce conţine respectivul set de date ordonat.

Numărul de comparaţii efectuate = numărul de noduri interioare pe acest drum = lungimea acestui drum (măsurată în arce).

Limita inferioara a performantei algoritmilor de sortare bazati pe comparatii intre chei

Numărul de comparaţii pe care îl face algoritmul **în cazul mediu**, revine la a estima numărul mediu de noduri interioare pe toate drumurile, cu alte cuvinte **lungimea externă medie** a arborelui binar strict de decizie asociat: (Propozitia 6) lungimea externă medie a unui asemenea arbore este mai mare decât

$$\lfloor \log_2 n! \rfloor$$

Numărul de comparaţii în cazul **cel mai nefavorabil**, revine la a estima numărul de noduri interioare pe **drumul cel mai lung până la o frunză =** *d* = adâncimea arborelui: (Corolar la Propozitia 3)

$$d >= \lceil \log_2 n! \rceil$$