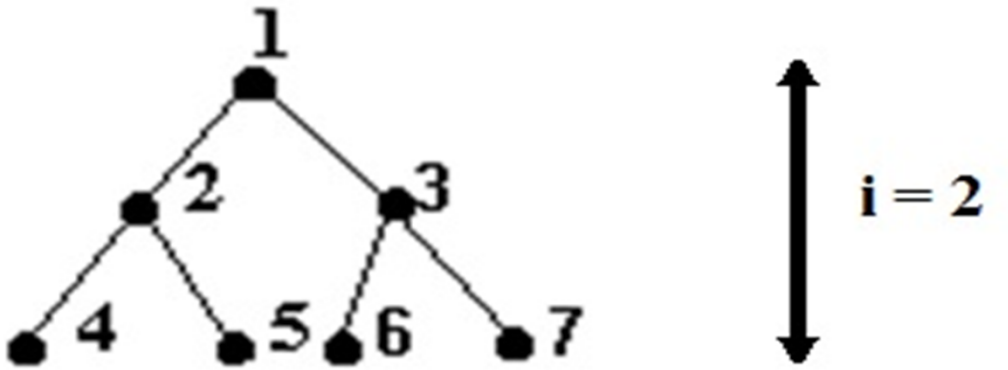
SDA (PC2)
Curs 12
Arborele heap și tehnici de sortare

Iulian Năstac

Arbori binari speciali

Recapitulare

P: Un arbore binar de înălțime **i** are maximum $2^{i+1}-1$ varfuri (sau noduri)



Observație:

Orice nivel de adâncime **k** are maximum 2^k noduri

Arborele binar plin

Recapitulare

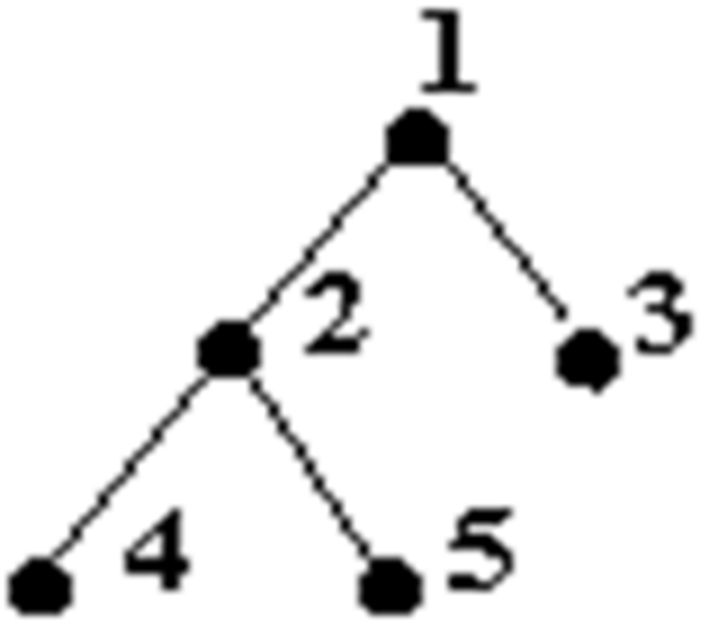
Arborele binar plin este arborele care are numărul maxim de vârfuri ($2^{i+1}-1$) pentru o înălțime **i** dată.

De exemplu, arborele binar plin cu înălţimea 2 se prezintă astfel:

Arborele binar complet

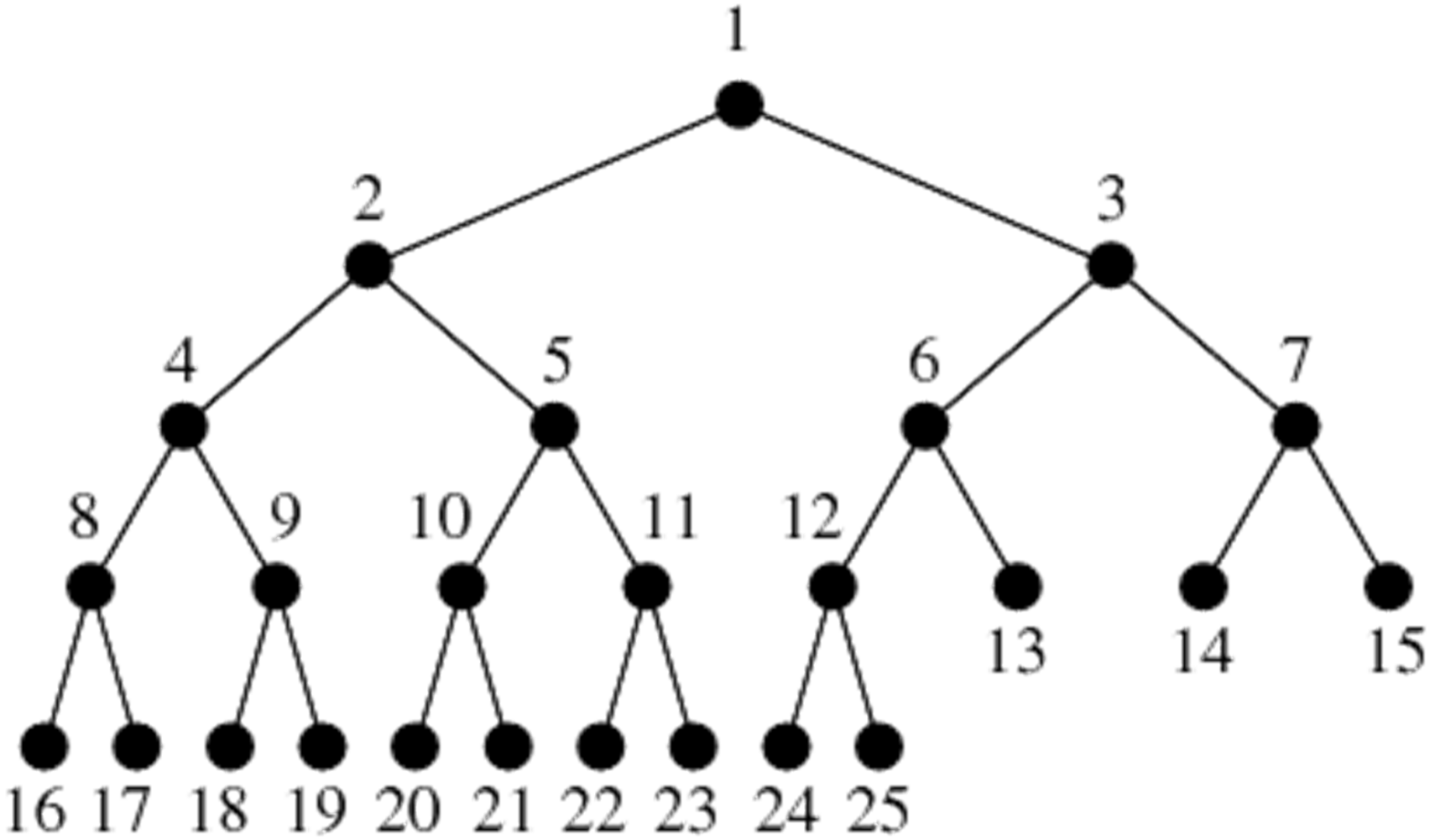
Recapitulare

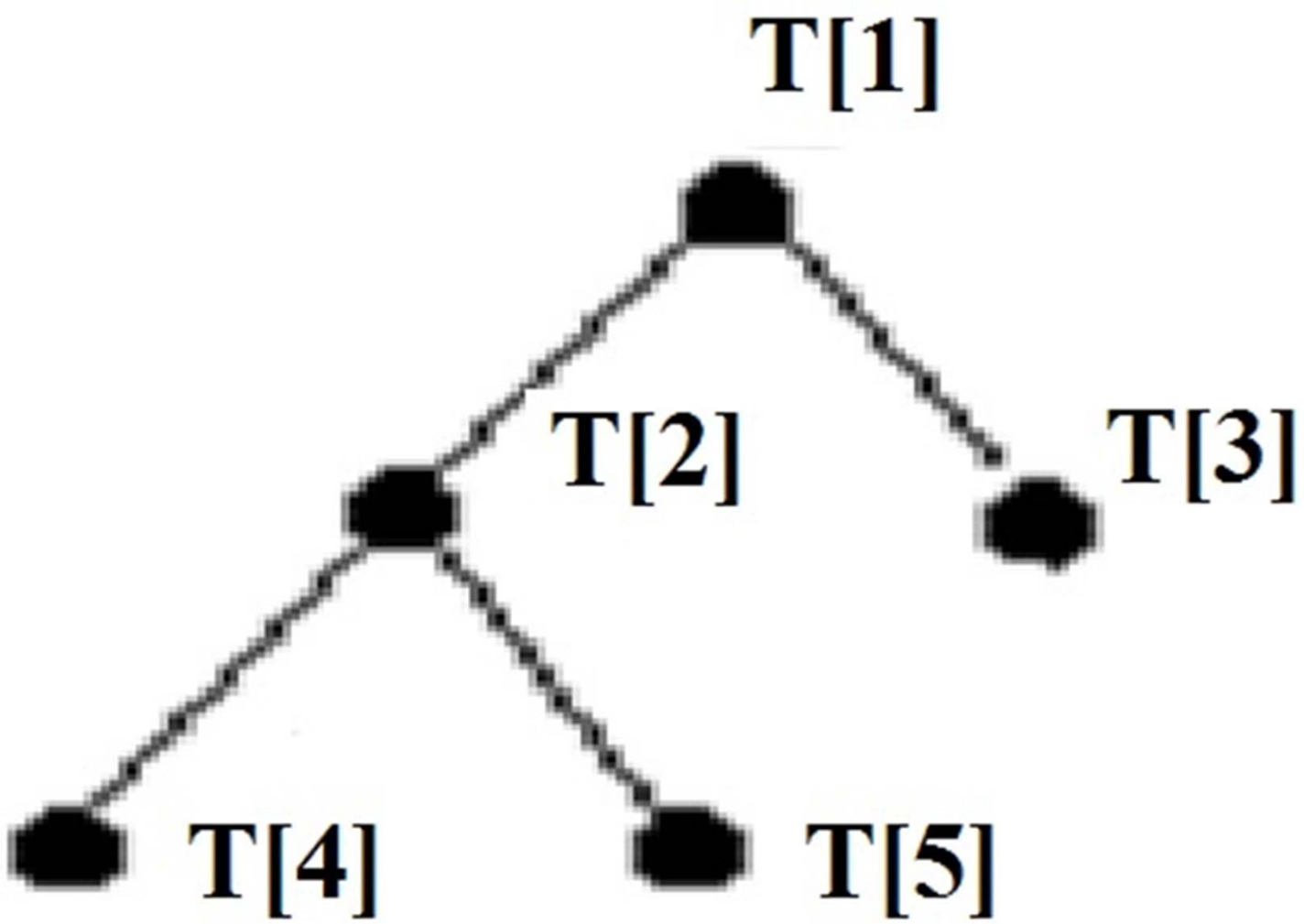
Un arbore binar cu ***n*** vârfuri și de înălțime ***i*** este **complet** dacă se obține din arborele binar plin de înălțime ***i***, prin eliminarea vârfurilor numerotate cu ***n+1***, ***n+2***, ... până la $2^{i+1}-1$.



Observație :

- Un arbore binar complet se poate reprezenta secvențial folosind un tablou **T**, punând vârfurile de adâncime ***k***, de la stânga la dreapta, în pozițiile: $T[2^k]$, $T[2^k+1]$, …, $T[2^{k+1}-1]$, cu excepția nivelului final care poate fi incomplet.


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25



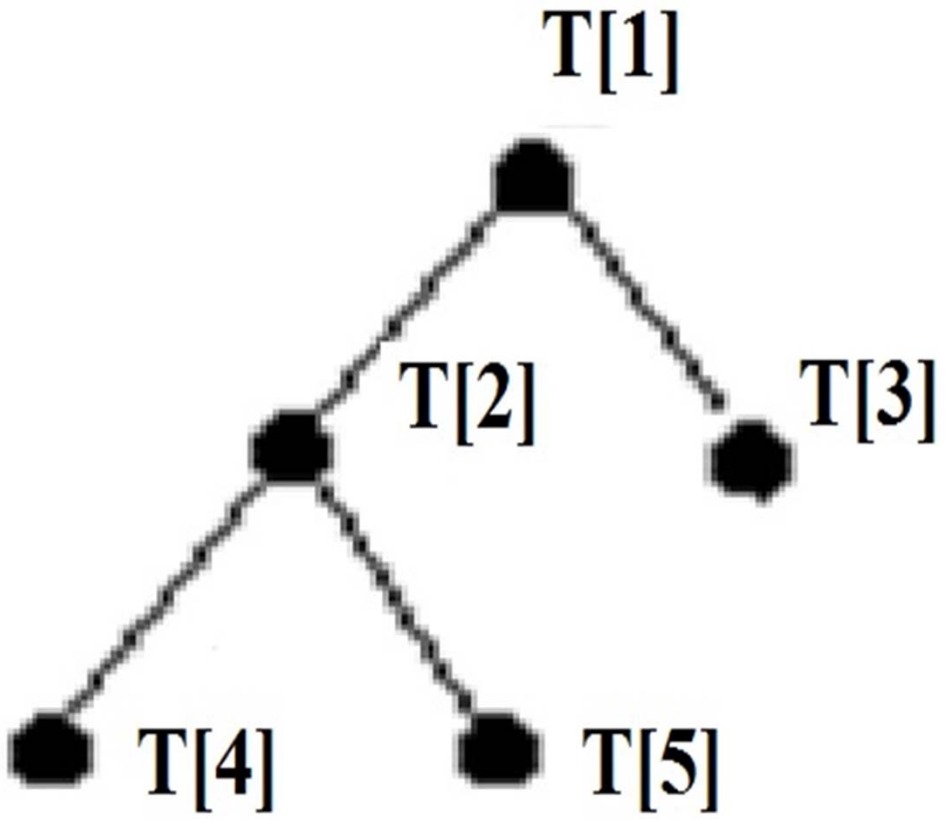
• **Atenție**: acesta este un tablou generic care începe cu T [1] și nu cu T [0].

• Se pot opera modificările necesare atunci când se scrie codul în C.

Observații:

- Tatăl unui vârf reprezentat în **T[i]**, i>1, se află în **T[i div 2]** .

- Fiii unui vârf reprezentat în **T[i]**, se află (dacă există) în **T[2·i]** și **T[2·i + 1]** .



Înălțimea unui arbore binar complet (Recapitulare)

- Am demonstrat în cursul anterior că înălțimea unui arbore binar complet cu n vârfuri este:

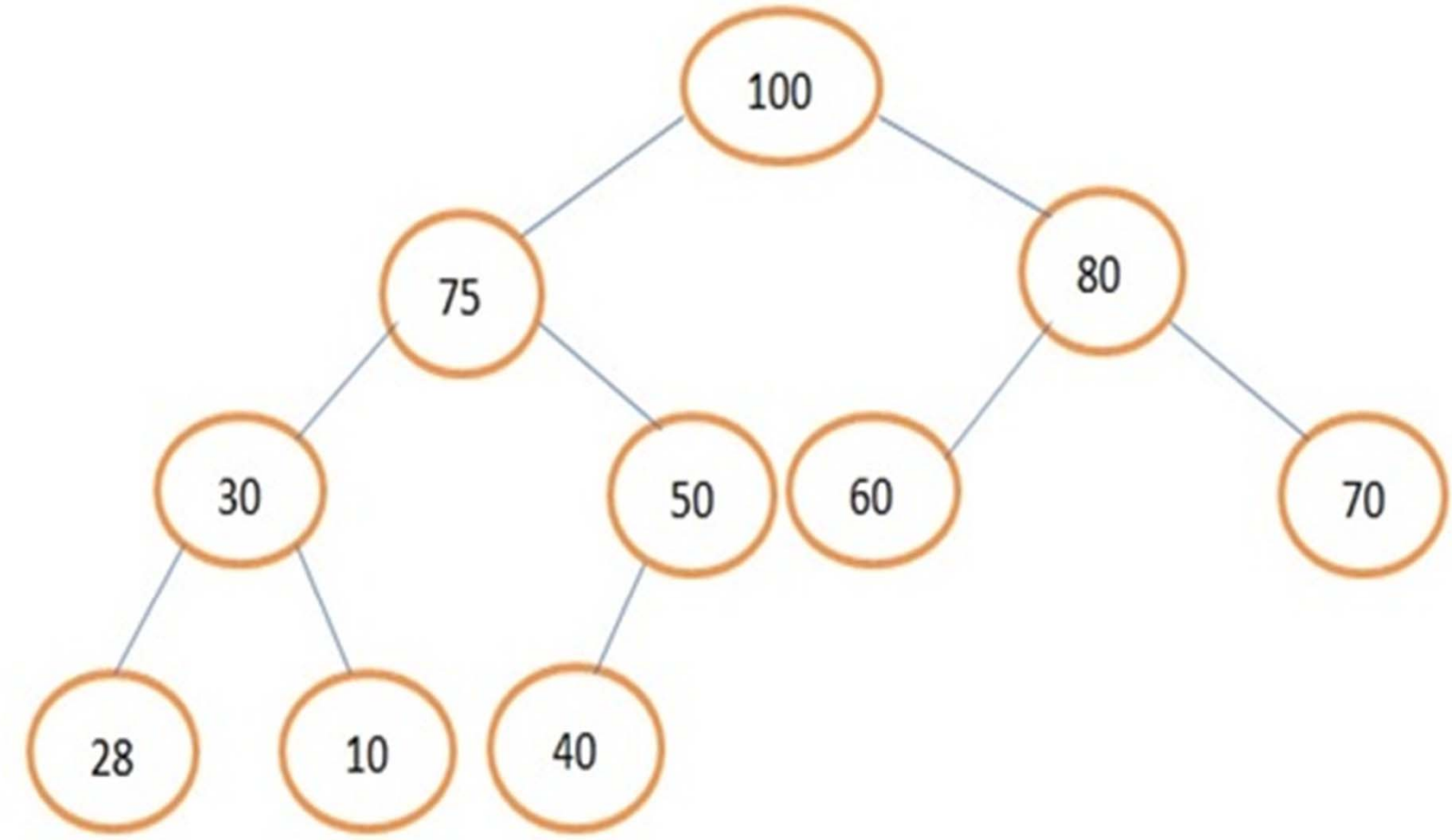
$$i = \lfloor \log_2 n \rfloor$$

Arborele HEAP

Recapitulare

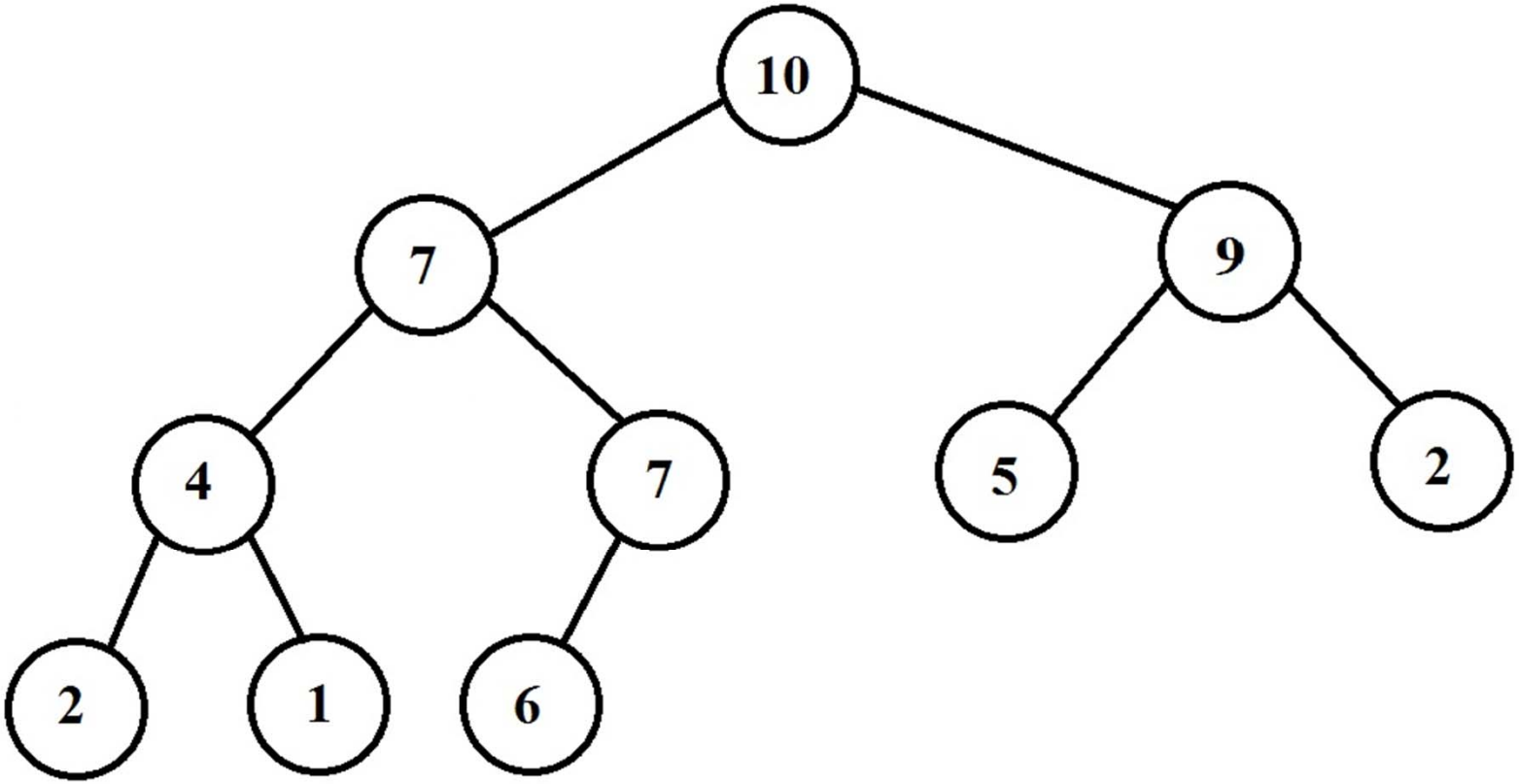
Un heap este un arbore “binar” complet care are proprietarea că valoarea fiecărui vârf este mai mare sau egală cu valoarea fiecărui fiu al său.

Observație: orice heap poate fi reprezentat printr-un vector (tablou unidimensional)



100	75	80	30	50	60	70	28	10	40
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

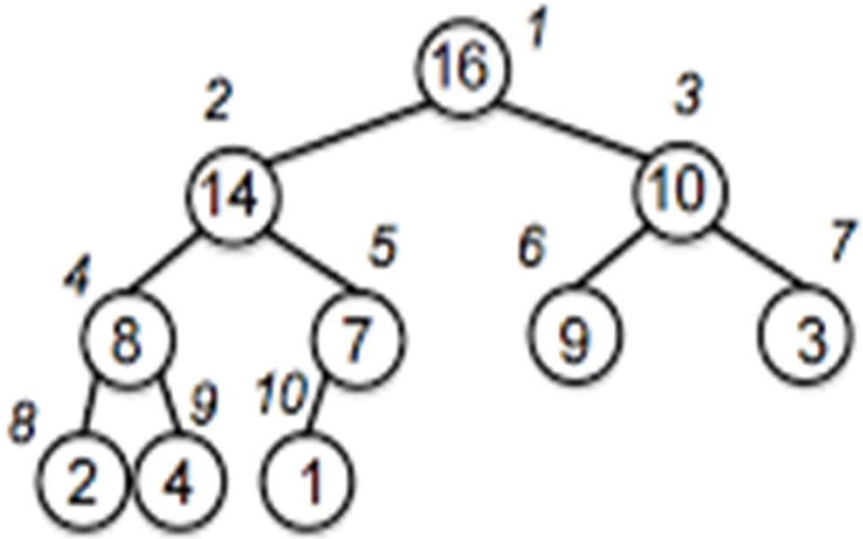
Exemplu 2:



Acest heap poate fi reprezentat prin următorul tablou:

10	7	9	4	7	5	2	2	1	6
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

Exemplul 3:



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Acest heap poate fi reprezentat prin următorul vector:

16	14	10	8	7	9	3	2	4	1
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

Observații:

- Într-un heap se pot opera modificări la nivel de nod (schimbarea valorii nodului curent).
- Astfel valoarea unui nod poate crește sau poate fi micșorată anulând ordinea inițială de heap.
- Ordinea de heap poate fi restabilită simplu prin intermediul a două operațiuni numite de cernere și de filtrare.

Filtrarea în heap

Dacă valoarea unui vârf crește astfel încât depășește valoarea tatălui, este suficient să se schimbe între ele aceste două valori și să se continue procesul în mod ascendent, până când proprietatea de heap va fi restabilită.

Se spune că valoarea modificată a fost **filtrată** (***percolated***) către noua sa poziție.

Cernerea în heap

Dacă valoarea unui vârf scade astfel încât devine mai mică decât valoarea fiului mai mare, este suficient să schimbăm între ele aceste două valori, procesul continuând în mod descendent, până când proprietatea de heap va fi restabilită.

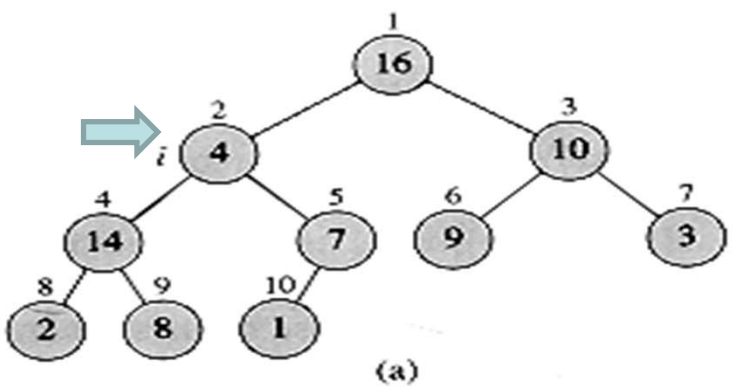
Se spune că valoarea modificată a fost cernută (**sift down**) către noua sa poziție.

Notă:

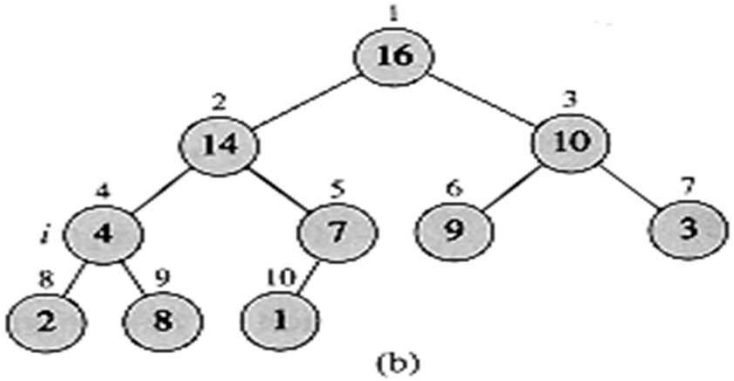
- În continuare, marea majoritate a funcțiilor vor fi scrise în varianta **pseudocod**

Pseudocodul funcției de cernere

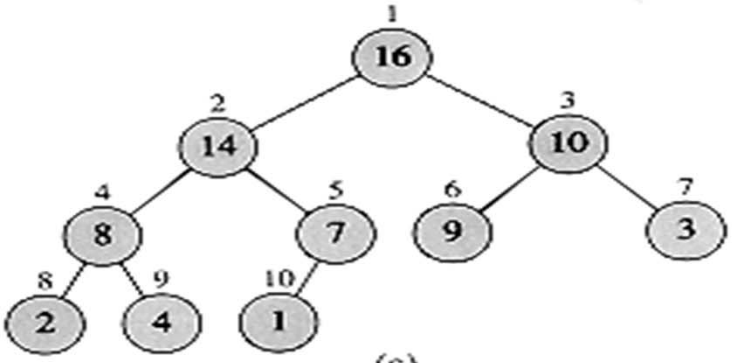
```
void cerne (T[1…n], i)
{ int k, x, j;
  k ← i ;
  do      {
            j ← k ;
            if ((2j ≤ n) ∧ (T[2j] > T[k]))  then k ← 2j;
            if ((2j+1 ≤ n) ∧ (T[2j+1] > T[k]))  then k ← 2j+1;
            x ← T[j];
            T[j] ← T[k];
            T[k] ← x
          } while (j ≠ k)
}
```



(a)

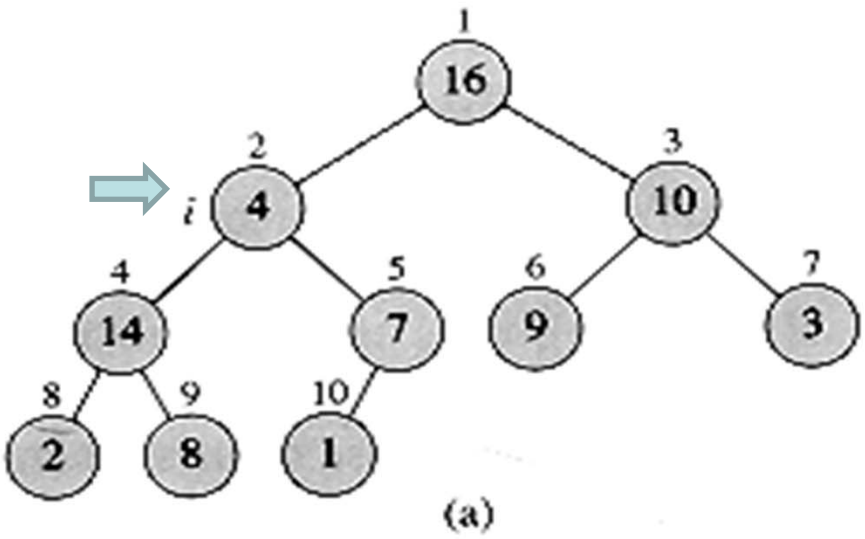


(b)

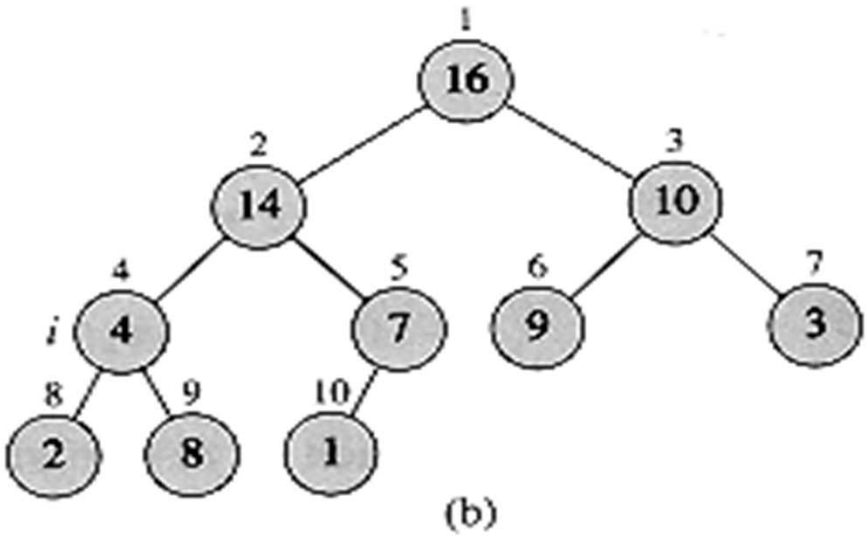


(c)

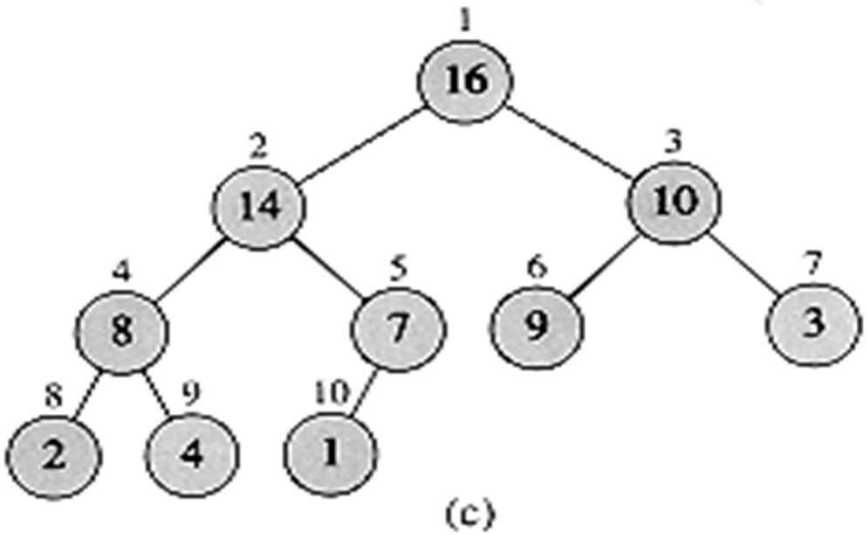
În figura **(a)** este configurația inițială a arborelui heap, cu **A [2]** la nodul **i** = 2 ce încalcă proprietatea heap-ului, deoarece A [2] nu este mai mare decât ambii copii. Proprietatea de heap este restabilită pentru nodul 2 în figura **(b)** prin schimbul A [2] cu A [4] (folosind alte două variabile k, j și un tampon x, unde inițial k = i și j = k), care totuși distruge proprietatea heap pentru nodul 4. Funcția de cernere (sift-down) continuă cu o buclă (în jos către nodurile copii) până când nu mai există modificări la structura de date. Aici, acest lucru este vizibil schimbând A [4] cu A [9], așa cum se arată în figura **(c)**.


1
16
2
i
4
3
10
4
14
5
7
6
9
7
3
8
2
9
8
10
1

(a)


1
16
2
14
3
10
4
i
4
5
7
6
9
7
3
8
2
9
8
10
1

(b)


1
16
2
14
3
10
4
8
5
7
6
9
7
3
8
2
9
4
10
1

(c)

Pseudocodul funcției de filtrare

```
void filtreaza (T[1…n], i)
{ int k, j, x;
  k ← i ;
  do      {
           j ← k ;
           if ((j > 1) ∧ (T[ j div 2] < T[k]))  then k ← j div 2;
           x ← T[ j];
           T[ j] ← T[k];
           T[k] ← x
          } while (j ≠ k)
}
```

Restabilirea proprietății de heap

Considerăm T[1..n] ca fiind un heap. Fie **i**, 1≤i≤n. Lui T[i] i se atribuie valoarea ν și apoi restabilim proprietatea de heap.

```
void restab_heap (T[1..n], i, ν)
{ variab. loc. x;
  x ← T[i];
  T[i] ← ν;
  if  ν < x        then   cerne(T, i);
                   else filtreaza(T,i);
}
```

Heap-ul este un modelul util pentru:

- **Determinarea și extragerea maximului dintr-o mulțime.**
- **Inserarea unui nou vârf.**
- **Modificarea valorii unui vârf** (cu restab_heap).

Operațiile anterioare pot fi folosite pentru a implementa o listă dinamică de priorități:

- Valoarea unui vârf va da prioritatea elementului corespunzător.
- Evenimentul cu probabilitatea cea mai mare se va afla mereu la rădăcina ***heap***-ului.
- Prioritatea unui eveniment poate fi modificată în mod dinamic.

Acestea sunt câteva principii care stau la baza programelor de baze de date.

Exemple de funcții utile:

1) **Funcția pentru găsirea unui maxim**:

```
gaseste_maxim (T[1..n])
{  return T[1];
}
```

2) **Funcția pentru extragerea unui maxim (și ștergerea sa):**

```
extr_max (T[1..n])
{ var loc x;
  x ← T[1];
  T[1] ← T[n];
  cerne (T[1..n-1], 1);
  return x;
}
```

3) **Funcția de inserare a unui element nou în heap**:

```
insert (T[1..n], ν)
{
    T[n+1] ← ν;
    filtreaza (T[1..n+1], n+1);
}
```

Observație: în limbajul C nu este luată în considerație valoarea maximă a numărului de elemente ale unui tablou, folosit ca parametru de funcție, astfel încât, de exemplu, pentru funcția de cernere, filtrare, etc., va trebui să ținem seama de un parametru suplimentar.

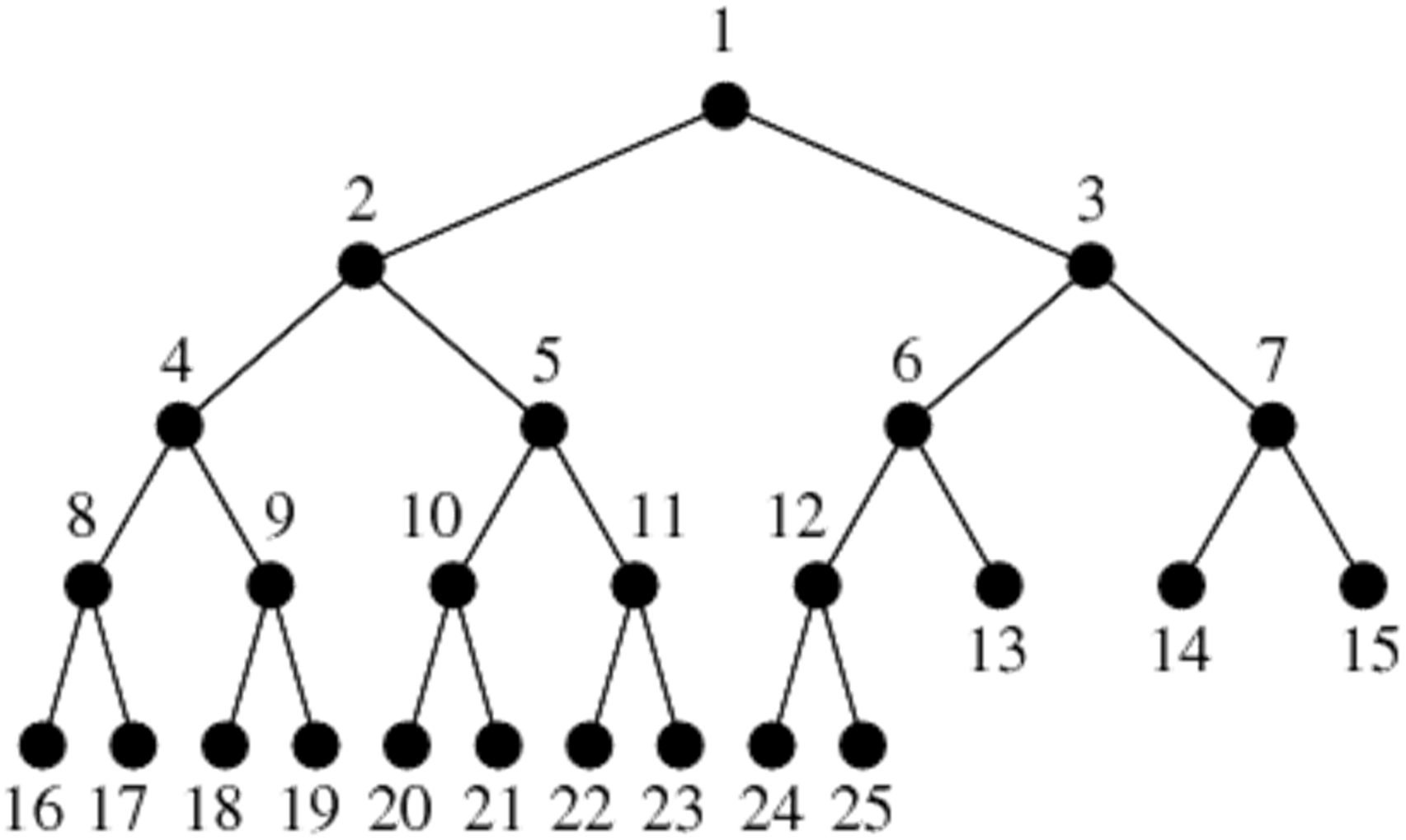
Ex.: **cerne (T[ ], *n*, i)**

unde ***n*** indică indicele ultimului element folosit (!de fapt numerotarea ar trebui să înceapă de la 0).

Cum putem forma un heap pornind de la un vector neordonat T[1..n] ?

- O soluție mai puțin eficientă este aceea de a porni de la un heap virtual vid și adăugăm elementele unul câte unul.

```
slow_make_heap(T[1..n])
{
   for (i=2; i ≤ n; i++)
        filtreaza (T[1..i], i);
}
```


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25

Există un algoritm mai eficient care lucrează liniar (din punct de vedere al ordinului/eficienței):

```
make_heap(T[1..n])
{
        for (i = n div 2; i ≥ 1; i - -)
                cerne (T[ ], i);
}
```

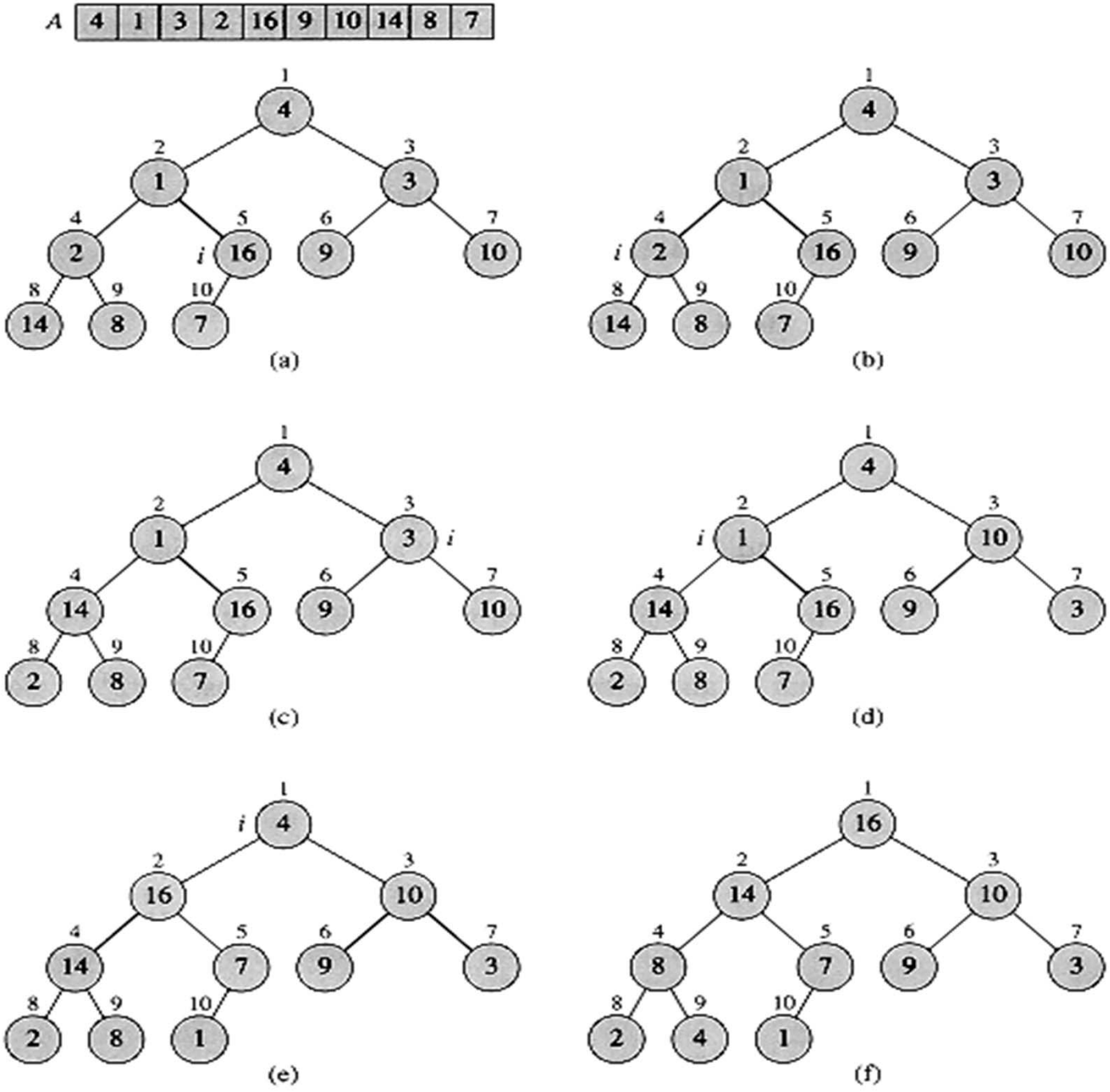
Cum putem construi un heap dintr-un șir (sau vector) arbitrar de numere

- Ca exemplu putem porni de la următorul vector (care nu este un heap):

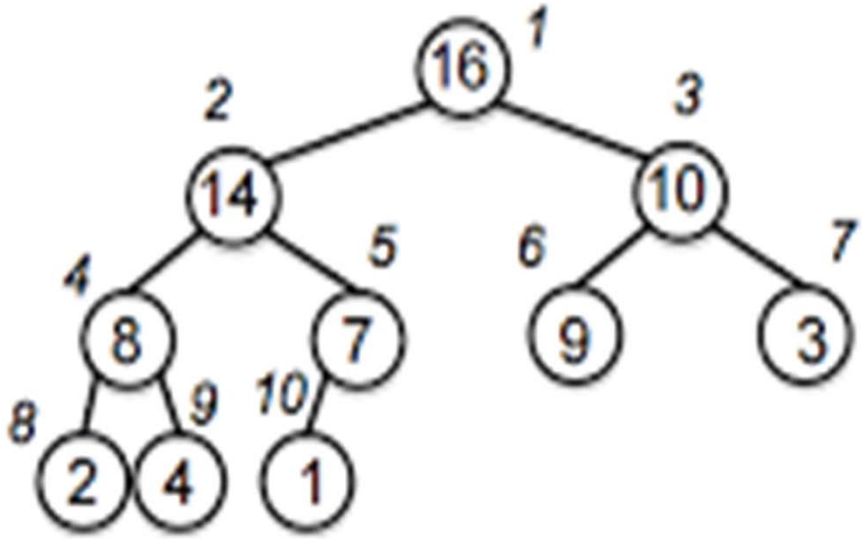
4	1	3	2	16	9	10	14	8	7
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

Printr-o secvență de câțiva pași (folosind make_heap) obținem:

16	14	10	8	7	9	3	2	4	1
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]


A 4 1 3 2 16 9 10 14 8 7
(a)
(b)
(c)
(d)
(e)
(f)

Arborele heap rezultat este:



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Acest heap poate fi reprezentat prin următorul vector:

16	14	10	8	7	9	3	2	4	1
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

Alt exemplu (ca temă):

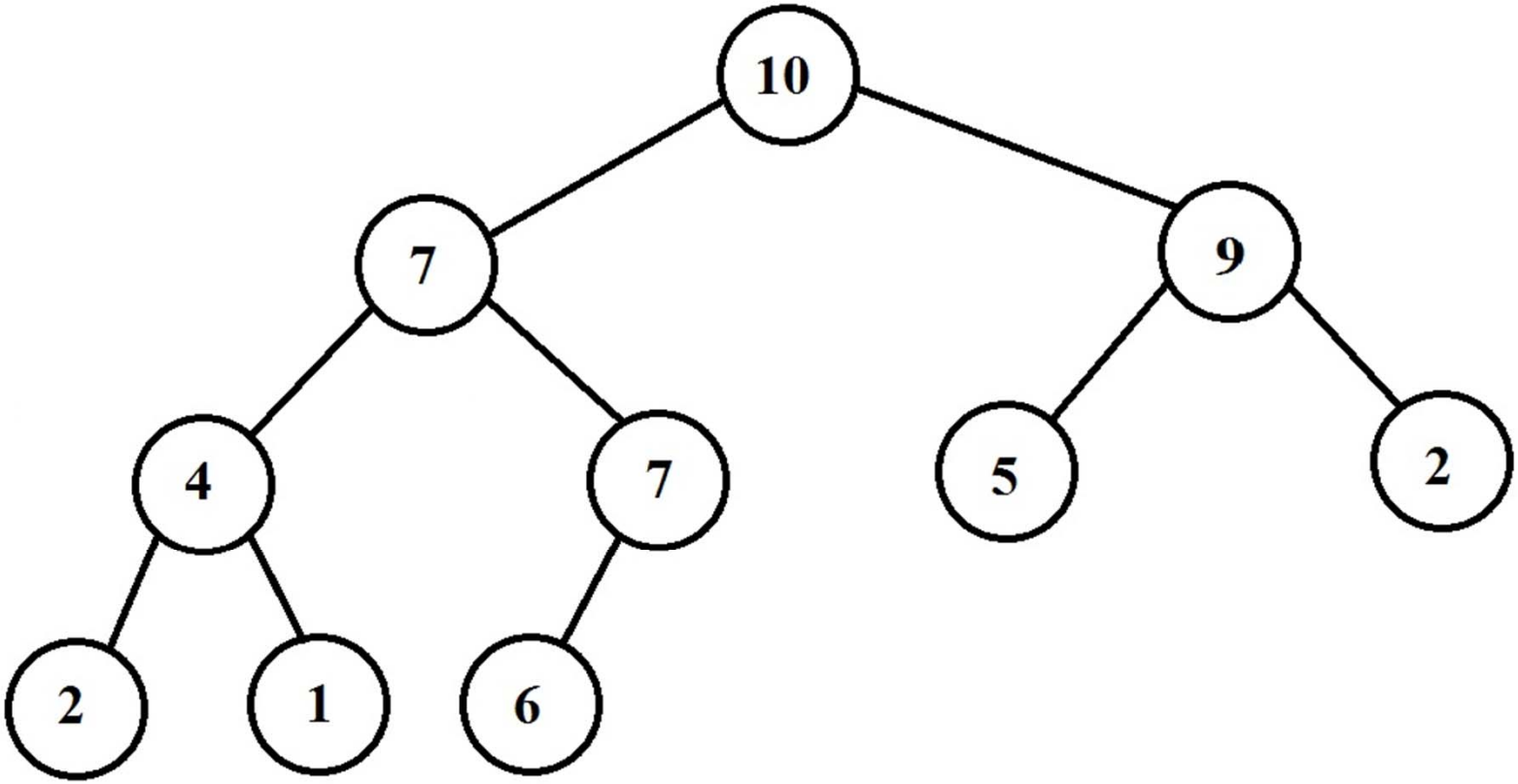
- Pornind de la vectorul următor (care nu este un heap):

1	6	9	2	7	5	2	7	4	10
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

Printr-o succesiune de câțiva pași se ajunge la:

10	7	9	4	7	5	2	2	1	6
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

Rezultatul ar trebui să arate așa:



Acest heap poate fi reprezentat prin următorul vector:

10	7	9	4	7	5	2	2	1	6
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

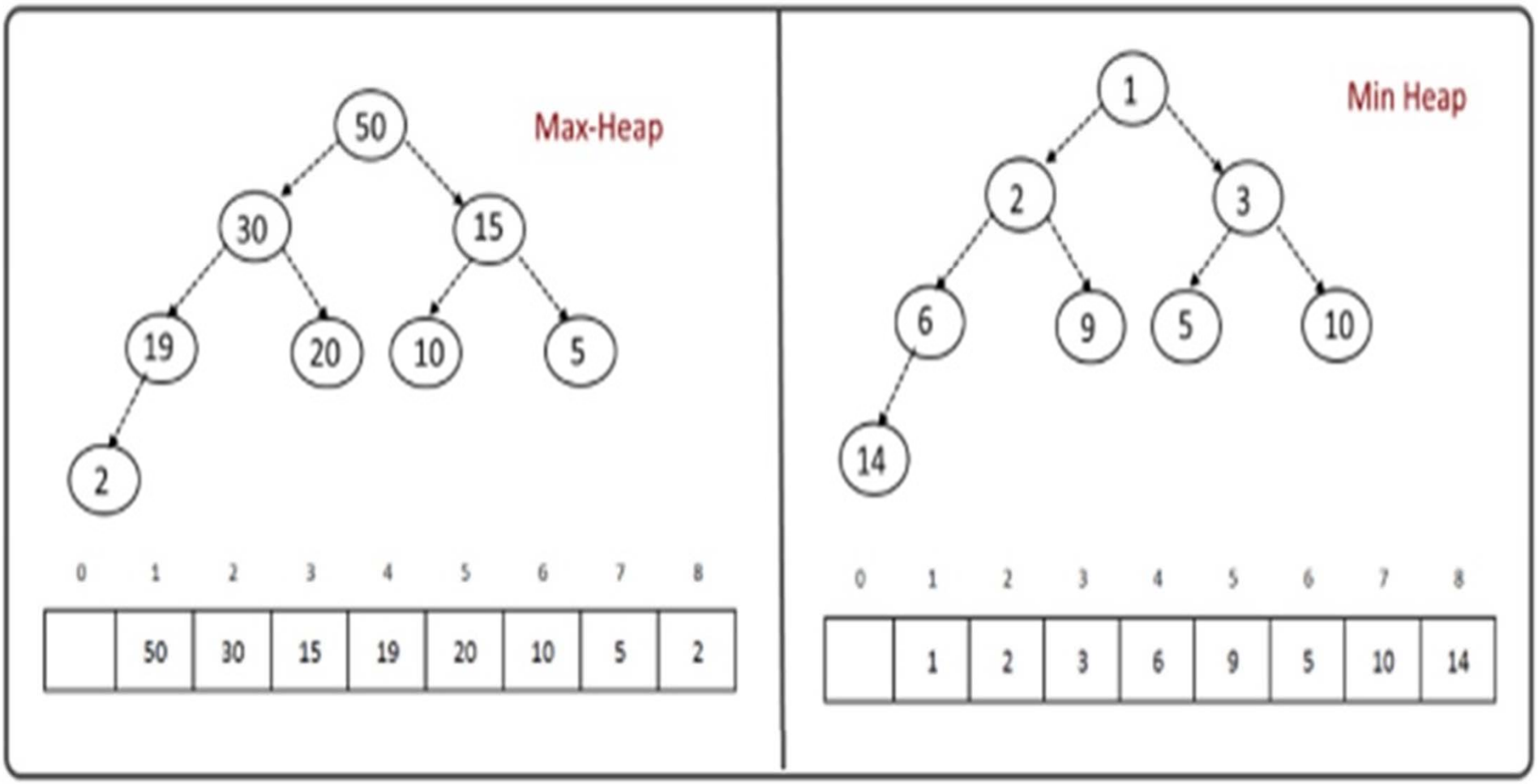
Există variante

- Def.: Un **min-heap** este un heap inversat (valoarea fiecărui vârf este mai mică sau egală cu valoarea fiecărui fiu al său).

Observații:

- Rădăcina min-heap-ului va conține cel mai mic element al vectorului.
- Se modifică în mod corespunzător și celelalte proceduri de manipulare a heap-ului.

Heap-ul clasic (Max Heap) în comparație cu un Min Heap



Atenție!

- Heap-ul este o structură de date foarte atractivă, dar are și limitări.

- Există operații care nu pot fi efectuate eficient într-un heap.

Dezavantajele unui heap:

- Necesitatea ca arborele să fie complet.

- Găsirea unui vârf, cu o anumită valoare dată, este ineficientă (pentru că pot exista mai multe vârfuri cu aceeași valoare plasate pe diferite ramuri și nivele din heap).

• O extensie a conceptului de heap este posibilă pentru arbori compleți și ordonați a căror noduri neterminale au mai mult de doi fii.

• O astfel de structură accelerează procedura de filtrare.

Principala aplicație:
O nouă tehnică de sortare
(**heapsort**)

- Noțiunea de heap a fost introdusă și folosită în anii 60 de Robert W. Floyd și J. W. J. Williams pentru crearea unui algoritm de sortare numit **heapsort**.

```
heapsort (T[1..n])
{
        make_heap(T);
        for( i = n; i ≥ 2; i - -)
                {
                 T[1] ↔ T[i];
                 cerne (T[1..i-1], 1)
                }
}
```

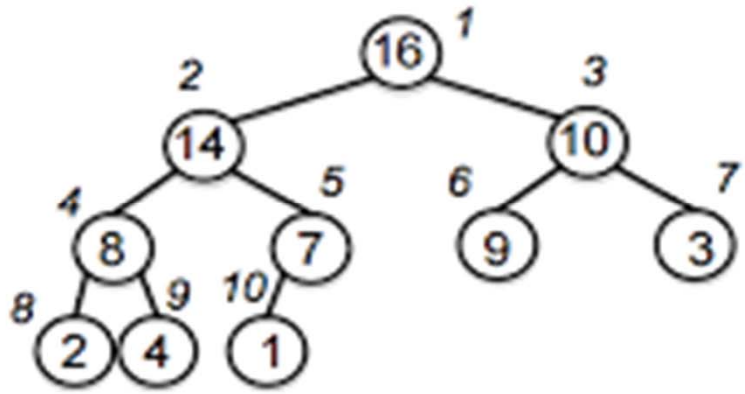
Exemplu

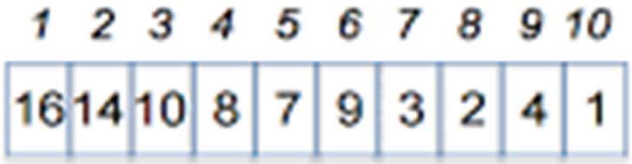
Se dorește obținerea unui arbore heap dintr-un vector nesortat

Pornim de la următorul vector (care nu este un heap):

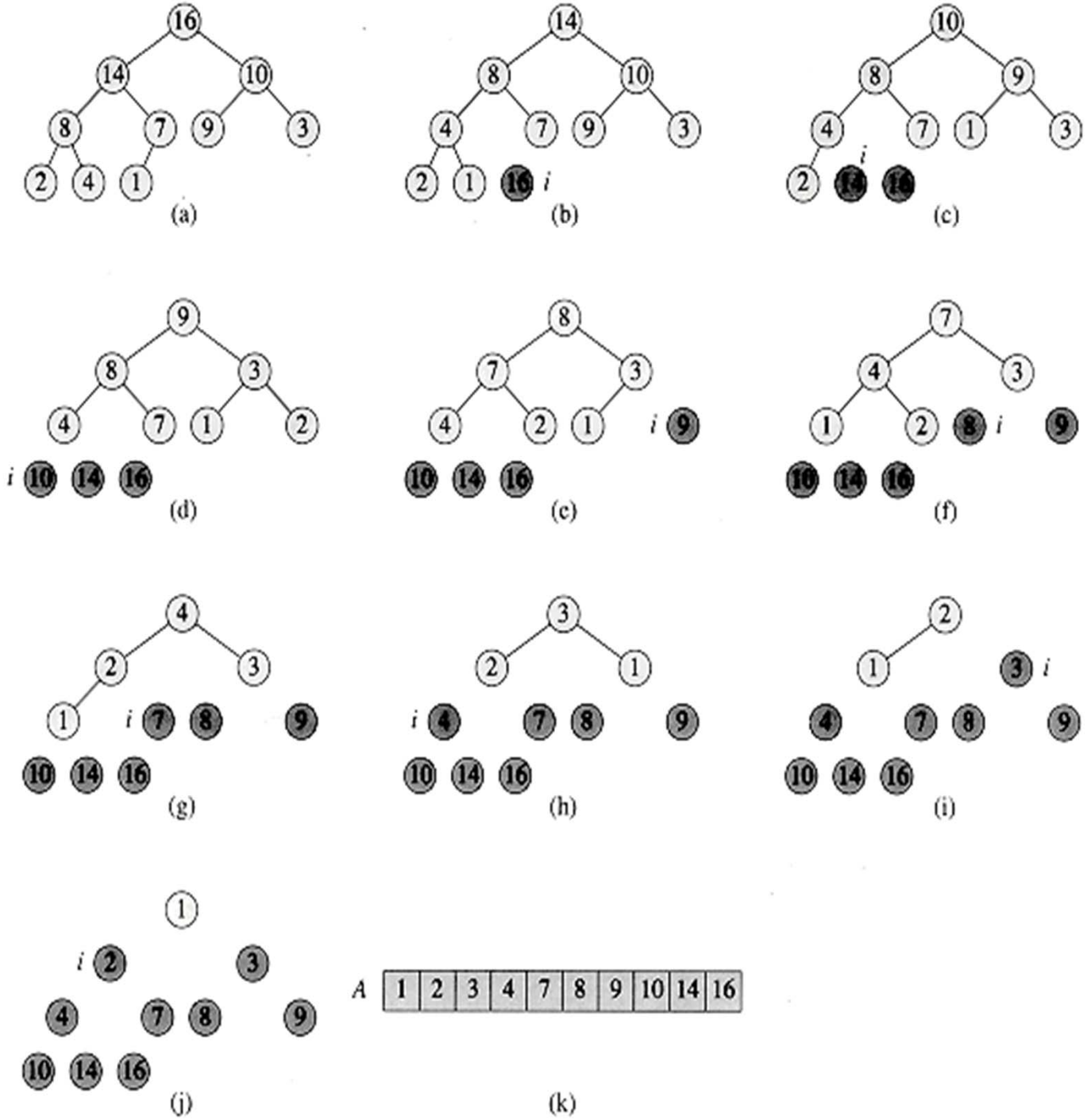
4	1	3	2	16	9	10	14	8	7
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

După apelarea funcției **make_heap** obținem:





16	14	10	8	7	9	3	2	4	1
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

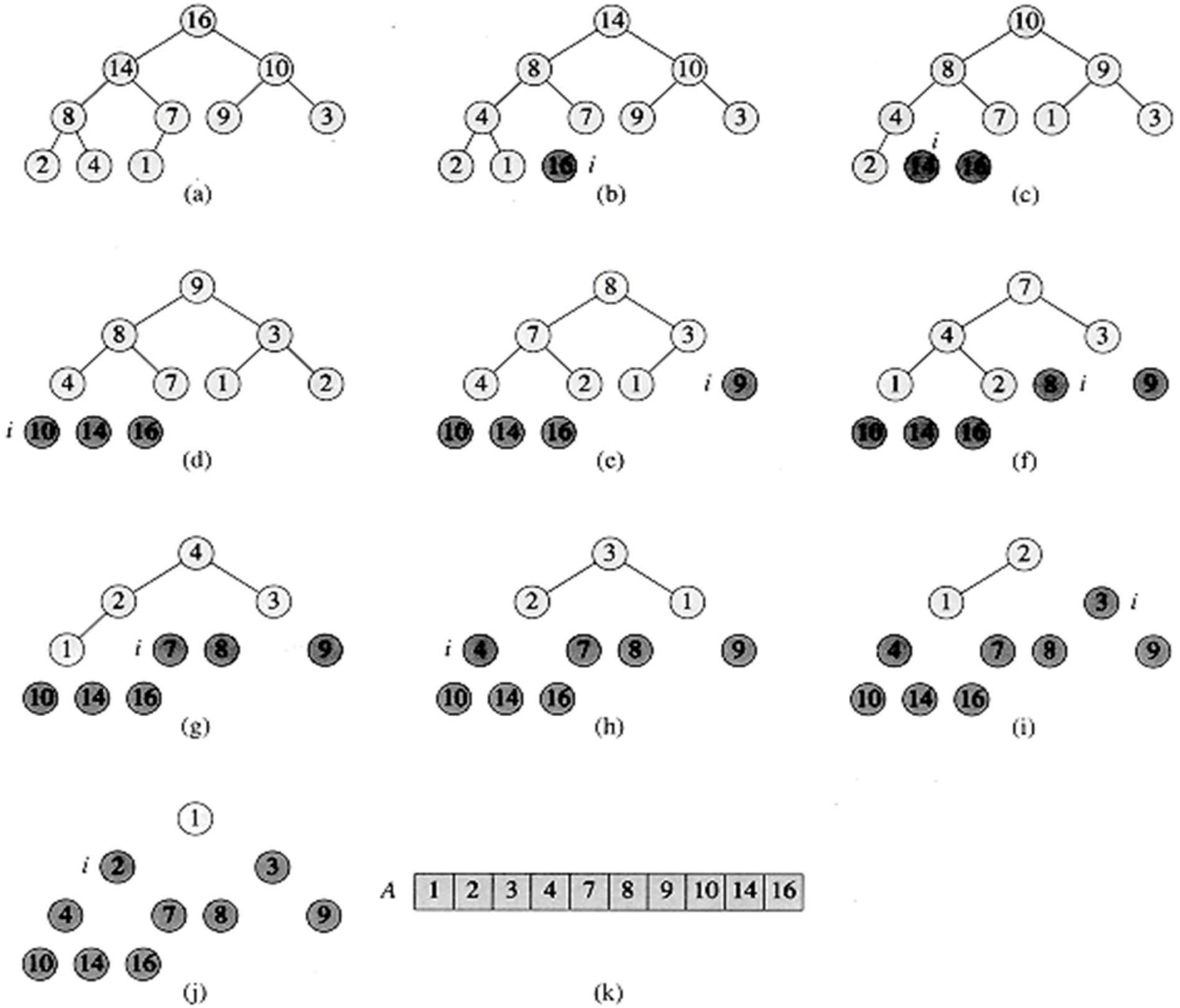


Operațiile din HEAPSORT:

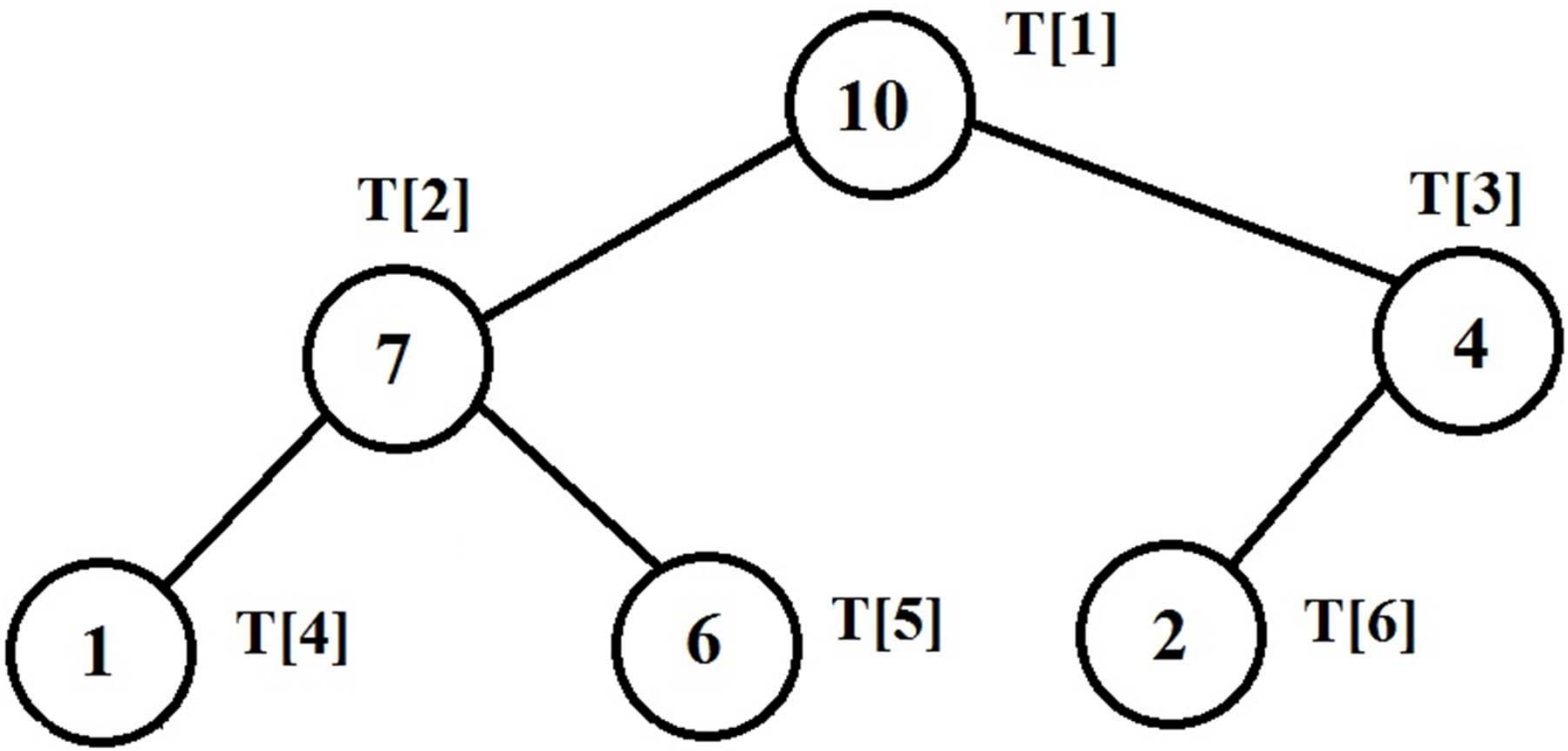
(a) Arborele heap imediat după ce a fost construit de funcția **make_heap**.

(b)-(j) Arborele heap imediat după fiecare apel al funcției **sift_down(T[1…i-1], 1)**. Este arătată valoarea în nodul de indice i după fiecare interschimbare. Numai nodurile legate au mai rămas în heap-ul micșorat.

(k) Rezultatul este vectorul sortat **A**


16 14 10 8 7 9 3 2 4 1
(a)
14 8 10 4 7 9 3 2 1 16 i
(b)
10 8 9 4 7 1 3 i 2 14 16
(c)
9 8 3 4 7 1 2 i 10 14 16
(d)
8 7 3 4 2 1 i 9 10 14 16
(e)
7 4 3 1 2 8 i 9 10 14 16
(f)
4 2 3 1 i 7 8 9 10 14 16
(g)
3 2 1 i 4 7 8 9 10 14 16
(h)
2 1 3 i 4 7 8 9 10 14 16
(i)
1 i 2 3 4 7 8 9 10 14 16
(j)
A 1 2 3 4 7 8 9 10 14 16
(k)

Alt exemplu (ca temă):



Alți algoritmi de sortare

- Inserție
- Selecție
- Merge sort
- Quicksort
- etc.

Metode de sortare

- Prin **metode de sortare** se înţelege o varietate de tehnici în urma cărora sunt **ordonate**, după anumite criterii specificate, diferite *secvenţe de obiecte* (date) care prezintă o trăsătură comună. În acest scop considerăm că datele sunt o *colecţie de elemente* de un anumit tip şi fiecare element conţine o dată sau chiar mai multe, în raport cu care se realizează ordonarea. O astfel de dată se numeşte **cheie**.

Pentru limbajul de programare C, un algoritm de sortare se poate realiza prin una din următoarele metode:

1. Aranjând datele care se sortează în aşa fel încât cheile lor să corespundă ordinii dorite.

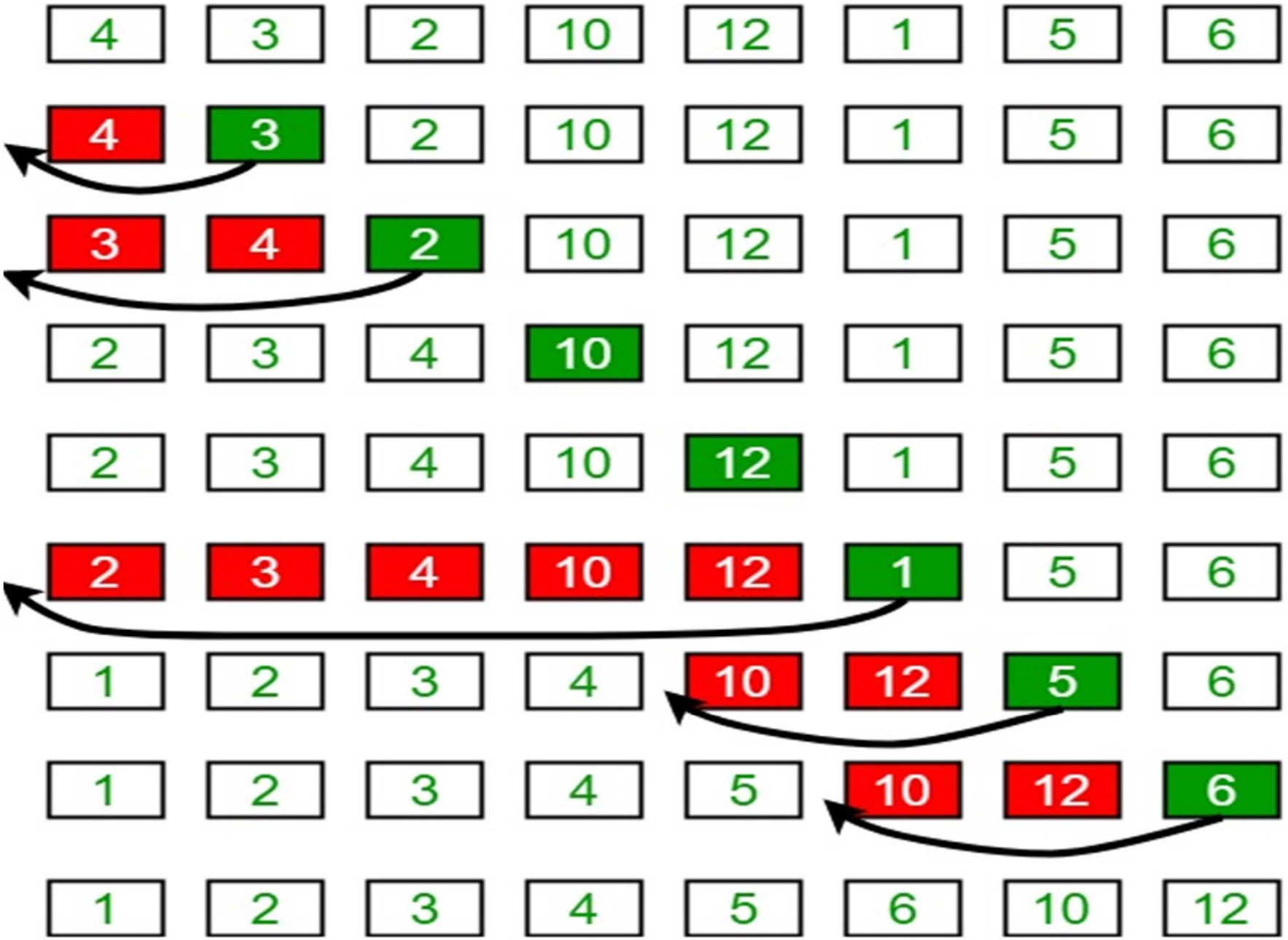
2. Ordonând un tablou de pointeri spre datele care trebuiesc sortate în aşa fel încât considerându-i pe aceştia în ordinea crescătoare a indicilor tabloului, datele spre care pointează să formeze o mulţime ordonată în acord cu ordinea dorită.

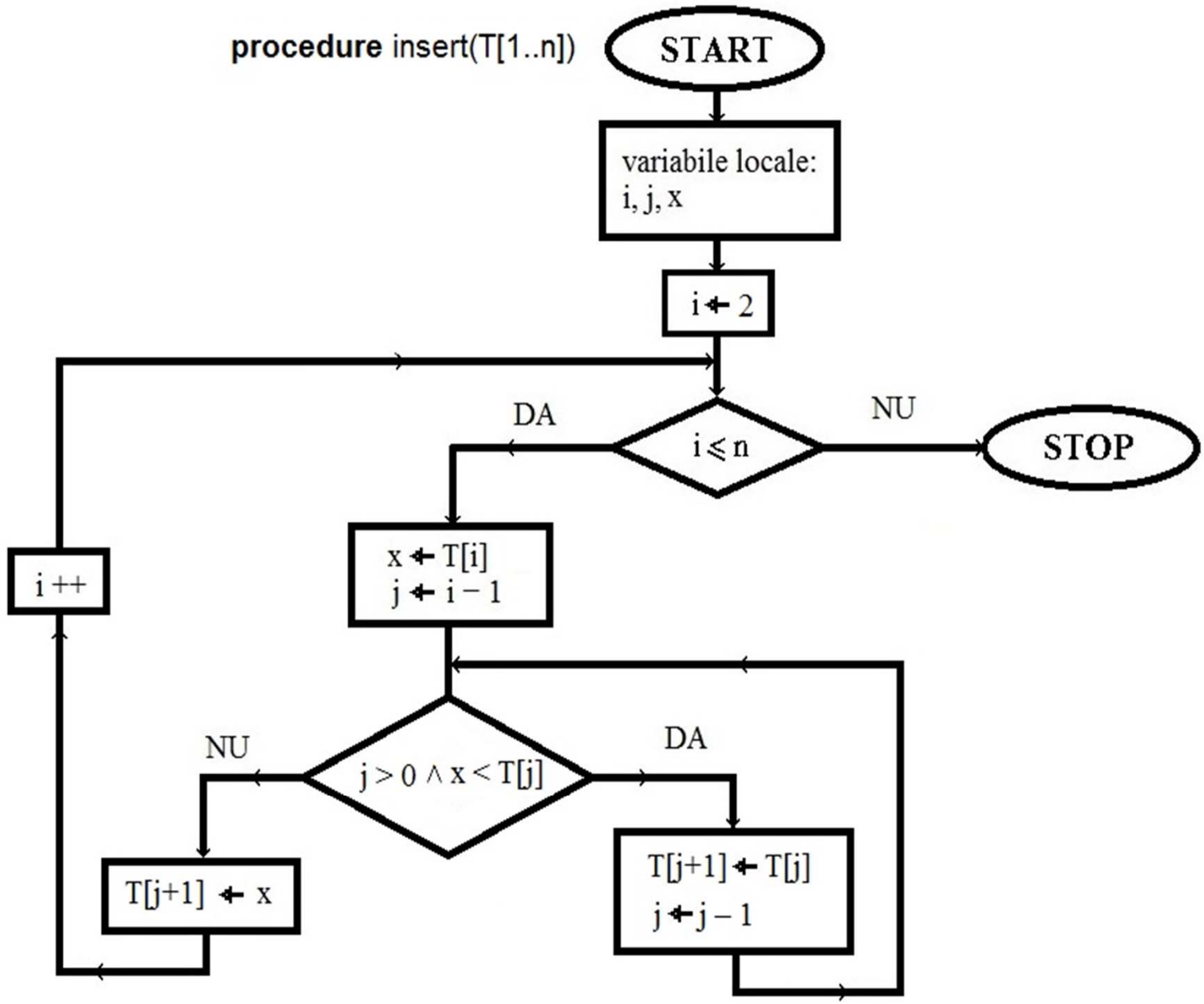
Observație:

- În continuare ne vom restrânge aria de lucru la sortarea tablourilor unidimensionale (vectori) cu date numerice.

Algoritmul de sortare prin inserţie

- Ideea generală a sortării ***prin inserţie*** este să considerăm pe rând fiecare element al şirului şi să îl inserăm în subşirul ordonat, creat anterior din elementele precedente. Operaţia de inserare implică deplasarea spre dreapta a unei secvenţe.


4 3 2 10 12 1 5 6
4 3 2 10 12 1 5 6
3 4 2 10 12 1 5 6
2 3 4 10 12 1 5 6
2 3 4 10 12 1 5 6
2 3 4 10 12 1 5 6
1 2 3 4 10 12 5 6
1 2 3 4 5 10 12 6
1 2 3 4 5 6 10 12


procedure insert(T[1..n])
START
variabile locale:
i, j, x
i ← 2
DA
i ⩽ n
NU
STOP
x ← T[i]
j ← i − 1
i ++
NU
j > 0 ∧ x < T[j]
DA
T[j+1] ← x
T[j+1] ← T[j]
j ← j − 1

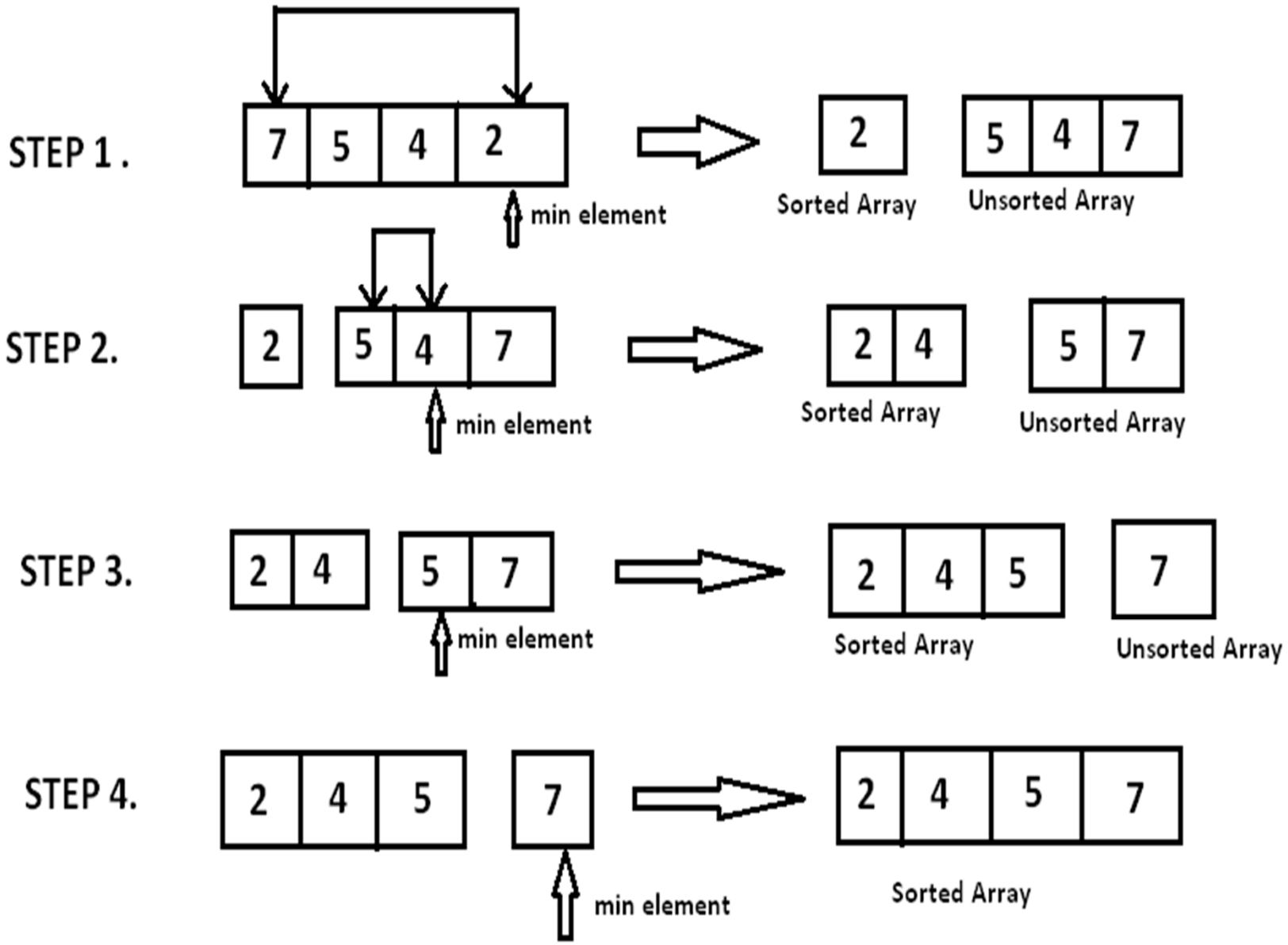
O descriere succintă a algoritmului în limbaj pseudocod este următoarea:

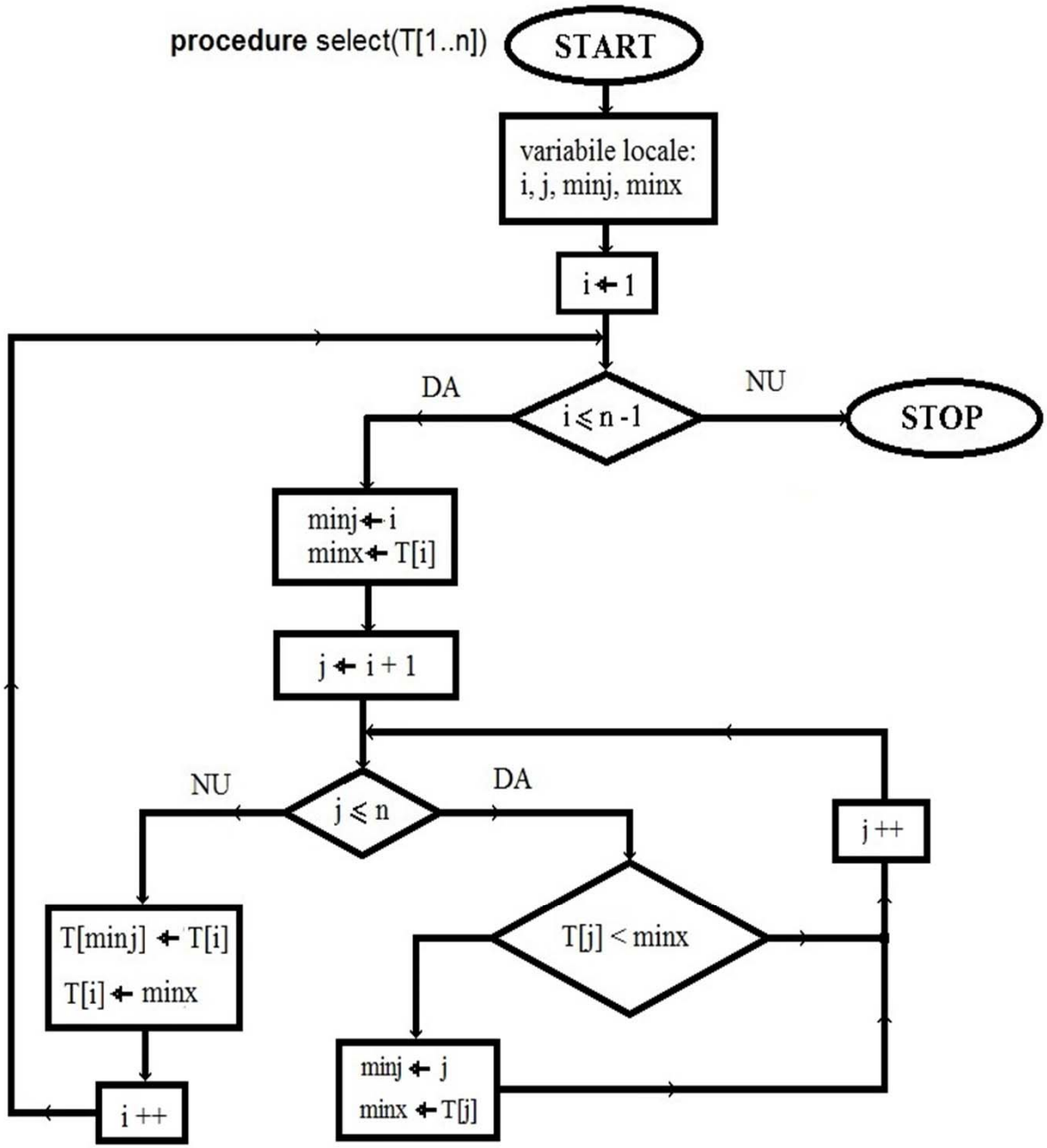
```
procedure insert(T[1..n])
{
        var. loc. i, j, x
        for i ← 2 to n do
        {
                x ← T[i]
                j ← i -1
                while (j>0 and x<T[j]) do
                        {
                         T[j+1] ← T[j]
                          j ← j – 1
                        }
                T[j+1] ← x
        }
}
```

Observaţie: La implementarea algoritmului de mai sus în cod C va trebui avut în vedere faptul că tablourile pornesc de la indicele 0.

Algoritmul de sortare prin selecţie

- În cadrul algoritmului de sortare prin selecţie se plasează la fiecare pas câte un element al vectorului direct pe poziţia sa finală. Şirul sortat creşte treptat pe masură ce partea nesortată, din care se selectează noi elemente, scade corespunzător.


STEP 1 .
7
5
4
2
min element
2
Sorted Array
5
4
7
Unsorted Array
STEP 2.
2
5
4
7
min element
2
4
Sorted Array
5
7
Unsorted Array
STEP 3.
2
4
5
7
min element
2
4
5
Sorted Array
7
Unsorted Array
STEP 4.
2
4
5
7
min element
2
4
5
7
Sorted Array


procedure select(T[1..n])
START
variabile locale:
i, j, minj, minx
i ← 1
DA
i ⩽ n -1
NU
STOP
minj ← i
minx ← T[i]
j ← i + 1
NU
j ⩽ n
DA
j ++
T[j] < minx
T[minj] ← T[i]
T[i] ← minx
minj ← j
minx ← T[j]
i ++

În pseudocod, algoritmului de sortare prin selecţie este descris de următoarea secvenţă:

```
procedure select(T[1..n])
{
   var. loc. i, j, min_j, min_x,
   for i ← 1 to n-1 do
            { min_j ← i
              min_x ← T[i]
              for j ← i+1 to n do
                        if T[j] < min_x then
                                  { min_j ← j
                                    min_x ← T[j]
                                  }
              T[min_j] ← T[i]
              T[i] ← min_x
            }
}
```

Observaţie: La implementarea algoritmului de mai sus în cod C va trebui avut în vedere faptul că tablourile pornesc de la indicele 0.

Algoritmul Bubblesort

- Acest algoritm face parte din cadrul metodelor de interschimbare, în care ideea generală este de a parcurge o serie de înregistrări şi de a le interschimba pe cele care nu verifică ordinea impusă. Cea mai simplă metodă este cea de interschimbare directă, care se mai numeşte **Bubblesort** sau **metoda bulelor**.

- Metoda bulelor are la bază compararea a două chei învecinate (de indici succesivi sau din noduri succesive). Dacă acestea nu sunt în ordinea cerută, atunci se permută elementele respective sau pointerii spre ele. Deşi eficientă, această metodă prezintă un număr mare de permutări.

Algoritmul Shell

- Metoda Shell a fost propusă de către Donald Shell în 1959 în urma căutării unei îmbunătăţiri a metodei bulelor. Astfel, în algoritmul Bubblesort, se compară elementele vecine şi dacă nu sunt în ordinea cerută, atunci ele se permută. De aceea, elementele care nu sunt în ordinea corectă se deplasează cu o singură poziţie. Pentru a îmbunătăţi acest procedeu, ar trebui să realizăm deplasări cu mai multe locuri ale elementelor, la o permutare a lor. Acest lucru se poate realiza dacă în loc de compararea elementelor învecinate se compară elemente aflate la o anumită distanţă între ele. În cazul în care ele nu sunt în ordinea cerută, acestea se vor permuta. Astfel, elementele se deplasează făcând salturi mai mari decât o poziţie.

- Distanţa dintre elementele comparate se numeşte increment. Incrementul se micşorează după o parcurgere a şirului de elemente care se sortează şi se reia parcurgerea de la începutul şirului. De aici rezultă şi denumirea alternativă de ***sortare cu micşorarea incrementului***.

Metoda de sortare Shell poate fi definită astfel:

1) Pornim cu un increment **inc = n/2**, unde prin **n** am notat numărul elementelor care se sortează.
2) Se realizează o parcurgere a şirului de elemente care se sortează.
3) Se înjumătăţeşte incrementul inc = inc/2 .
4) Dacă inc > 0, atunci se reia de la pasul *2)*, altfel algoritmul se opreşte.

Observaţie *Parcurgerea şirului de elemente implică următorii paşi*:

1) i = inc.
2) j = i – inc + 1 .
3) Dacă j > 0 şi elementele de ordine j şi j+inc nu satisfac criteriul de ordonare, atunci elementele respective se permută. Altfel se continuă cu pasul 6.
4) j = j – inc.
5) Se reia de la pasul 3.
6) i = i + 1.
7) Dacă i > n, se termină parcurgerea curentă a şirului. Altfel se reia de la pasul 2.

Observaţie:

- La implementarea algoritmului de mai sus în cod C va trebui avut în vedere faptul că tablourile pornesc de la indicele 0.

- Astfel iniţializarea de la punctul *2)* devine **j = i – inc** .

Backtracking

- **Backtracking** este numele generic al unui grup de algoritmi de descoperire a tuturor soluțiilor unei probleme de calcul.
- Un astfel de algoritm se bazează pe construirea incrementală de soluții posibile (denumite **candidat**), abandonând fiecare candidat parțial imediat ce devine clar că acesta nu are șanse să devină o soluție validă.

Observații:

- Un backtracking la limită este o problemă de căutare într-un arbore binar (**pre**, **in** sau **post** - ordine).

- Exemplul de bază folosit în numeroase publicații este problema reginelor, care cere să se găsească toate modurile în care pot fi așezate pe o tablă de șah opt regine astfel încât să nu se atace. (vezi problema de laborator de pe site http://www.euroqual.pub.ro/wp-content/uploads/sda_lab_06_backtracking.pdf)

Backtracking
(alte observații)

- Backtracking depinde de:

  - procedurile de tip "cutie neagră" construite de utilizator care definesc problema care trebuie rezolvată,

  - de natura candidaților parțiali,

  - de modul în care acestea sunt extinse în candidați complete.

- Este un algoritm metaheurist mai degrabă decât un algoritm specific.

- Totusi spre deosebire de multe alte meta-heuristici, este garantată găsirea tuturor soluțiilor la o problemă finită într-o perioadă limitată de timp.

Analiza eficienței algoritmilor

- Analiza eficienței unui algoritm are ca scop estimarea volumului de *resurse de calcul* necesare pentru execuția algoritmului.

Prin resurse se poate înțelege:

- ***Spațiul de memorie*** necesar pentru stocarea datelor pe care le prelucrează algoritmul.

- ***Timpul*** necesar pentru execuția tuturor prelucrărilor specificate în algoritm.