Laboratorul de Structuri de Date si Algoritmi – Lucrarea nr. 12

Arbori heap. HeapSort

1. **Reprezentarea implicita a arborilor binari**
2. **Arbori heap**
3. **Algoritmul HeapSort**

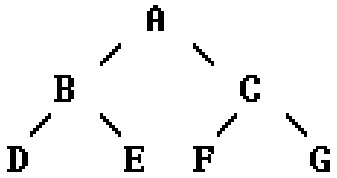
1. Reprezentarea implicita a arborilor binari

In reprezentarea implicita, structura arborelui nu este specificata prin informatii de inlantuire (pointeri), ci prin pozitia in care nodurile apar intr-un vector. Astfel, radacina arborelui va fi memorata in elementul **V[1]**, decendentii acesteia in **V[2]** si **V[3]**, decendentii nodului memorat in **V[2]**, vor fi memorati in **V[4]** si **V[5]** iar cei ai nodului memorat in **V[3]** in **V[6]** si **V[7]**, si asa mai departe.

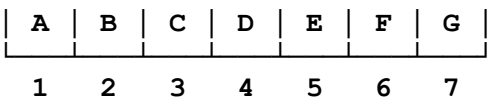
Pentru un nod care este memorat in pozitia **i** fiul sting va fi memorat in pozitia **2*i** iar cel drept in pozitia **2*i+1**:

Pozitie nod	Pozitie tata	Pozitie fiu sting	Pozitie fiu drept
i	[i/2]	2*i	2*i+1

De exemplu:



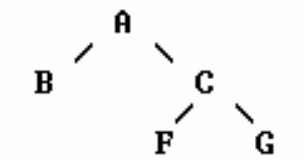
Va fi reprezentat implicit:



Atunci cind arborele nu este complet nodurile lipsa vor fi inlocuite cu o valoare speciala care indica lipsa nodului.

Laborator de Structuri de Date si Algoritmi – Lucrarea nr. 12

De exemplu **:**

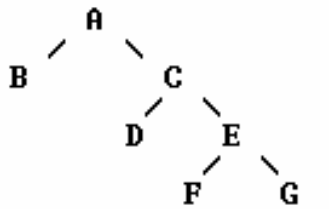


Va fi reprezentat implicit:

A	B	C	-	-	F	G
1	2	3	4	5	6	7

Pentru arbori dezechilibrati reprezentarea implicita face risipa de spatiu de memorare:

De exemplu:



Va fi reprezentat implicit:

A	B	C	-	-	D	E	-	-	-	-	-	F	G
1	2	3	4	5	6	7	9	10	11	12	13	14	15

Dimensiunea vectorului necesar pentru a memora un anumit arbore este mai mica sau egala cu:

$$2^{ad} - 1$$

unde ***ad*** este adincimea arborelui.

Observatie:

In C/C++ indiciele primului element din vector este **0**. Modul in care am definit reprezentarea implicita a arborilor binari nu permite reprezentarea radacinii arborelui in **V[0]**, deoarece `0*2 = 0`. Puteti alege una din urmatoarele solutii:

- elementul de vector **V[0**] sa ramina neintrebuintat;
- sa adaptam relatiile:

Pozitie nod	Pozitie tata	Pozitie fiu sting	Pozitie fiu drept
i	[(i+1)/2]-1	2*(i+1)-1	2*(i+1)

Pentru claritate vom alege prima solutie.

Pentru exemplificarea modului de utilizare a reprezentarii implicite a arborilor binari, in continuare este prezentata parcurgerea unui astfel de arbore in inordine. Presupunem urmatoarele declaratii de date globale:

```
char A[DIMMAX+1];
     //Vectorul in care este plasata reprezentarea implicita
int n;
     //Indicele ultimului nod care apare memorat in vector.
     //Daca structura arborelui nu prezinta "goluri" n este
     //numarul de noduri din arbore
```

Parcurgerea in inordine:

```
void inordine(int i)
{
     if (i<=n && (A[i]!='-'){
           inordine(i*2);
           prelucrare(A[i]);
           inordine(i*2+1);
           }
}
```

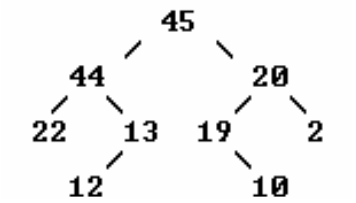
Aceasta procedura va fi apelata din programul principal cu linia:

```
inordine(1);  //1 este pozitia in vector a radacinii
```

2. Arbori heap

Proprietatea care defineste structura unui arbore heap este urmatoarea: Valoarea cheii memorate in radacina este mai mare decit toate valorile cheilor continute in subarborii descendentii.
Aceasta proprietate trebuie sa fie indeplinita pentru toti subarborii, de pe orice nivel in arborele heap.

Exemplu:



Laborator de Structuri de Date si Algoritmi – Lucrarea nr. 12

Aplicatiile pricipale ale arborilor heap sint:
- implementarea tipului de date abstract "**coada cu prioritate**".
- agoritmul de sortare **HeapSort**.

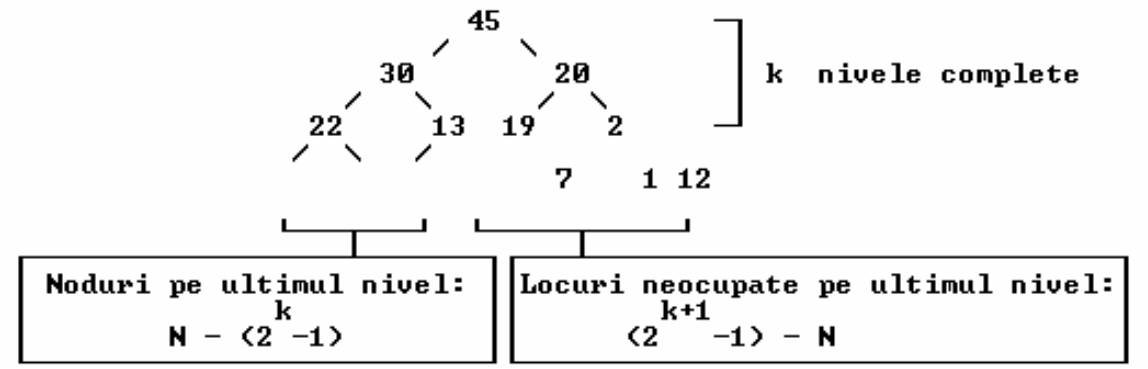
O ***coada cu prioritate*** este o structura de date de tip "container" pentru care se definesc urmatoarele functii de acces:

`INSERT(C, X)` - pune in coada **C** atomul **X**;

`REMOVE(C)` - scoate din coada **C** si returneaza atomul cu valoarea cea mai mare a cheii.

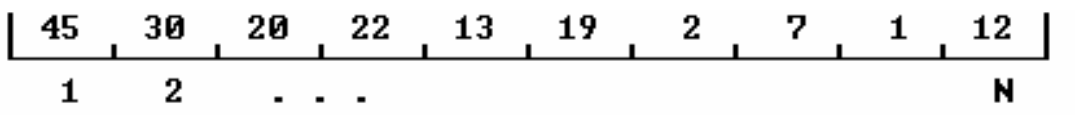
Arborii heap pot fi reprezentati explicit (reprezentarea standard a arborilor binari) sau implicit. *Reprezentarea implicita* a arborilor heap este cea mai folosita, si ea va face obiectul celor prezentate in continuare.

Vom realiza in cotinuare operatiile `INSERT` si `REMOVE` pentru arbori heap in reprezentare implicita. Arborii heap la care ne referim sint formati din **k** nivele complete iar pe nivelul **k+1** nodurile sint grupate in partea stinga:



Rezulta in reprezentarea implicita un vector fara "goluri" de dimensiune **N**:

Vectorul:

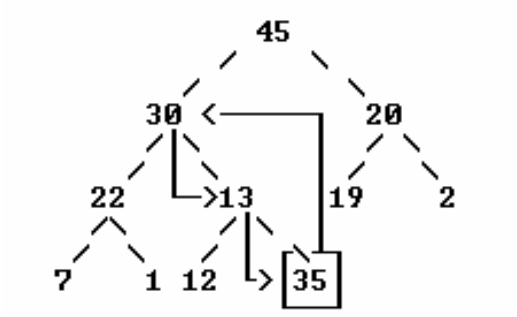


Laborator de Structuri de Date si Algoritmi – Lucrarea nr. 12

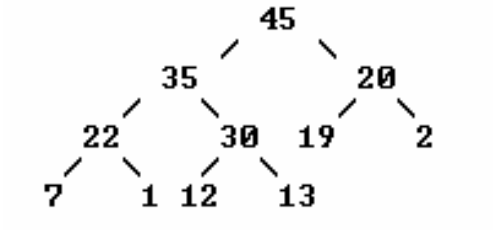
Operatia INSERT

Se adauga **X** la sfirsitul vectorului si apoi se promoveaza spre radacina pina ajunge in pozitia in care structura de heap a arborelui in reprezentare implicita este respectata.

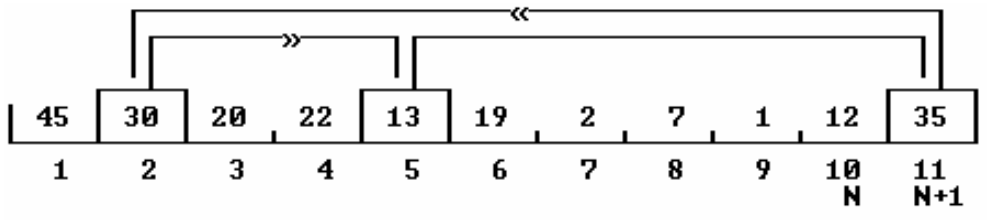
De exemplu, se insereaza valoarea **35** in heapul de mai sus:



Pentru a pastra strutura arborelui vloarea **35** trebuie promovata pina pe nivelul **2**. Valorile de pe portiunea din ramura parcursa vor fi retrogradate cu un nivel. Rezulta arborele:



Modificarile sint aplicate de fapt vectorului cu reprezentarea implicita:



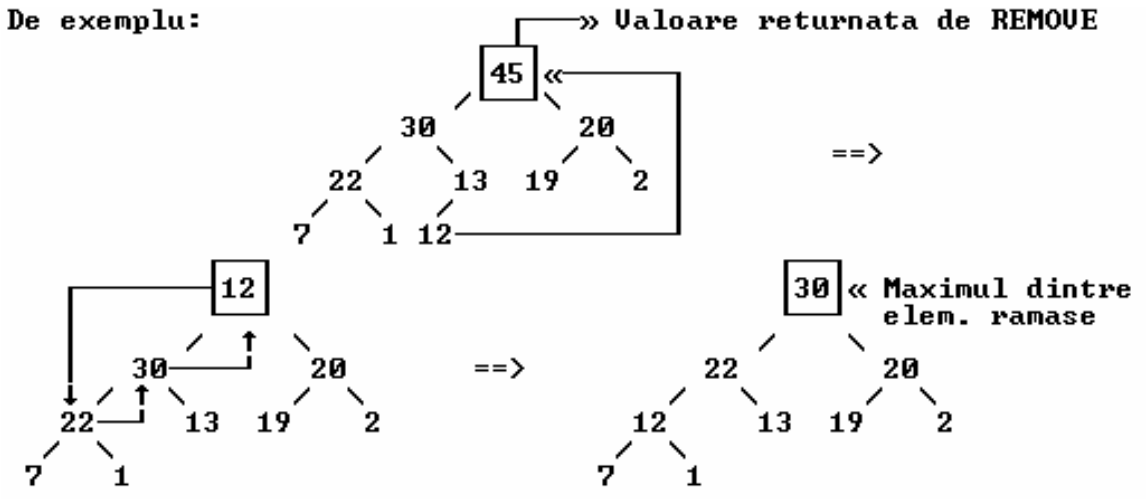
In algoritmul urmator promovarea valorii inserate se face "din aproape in aproape", aceasta urcind in arbore din nivel in nivel.

Laborator de Structuri de Date si Algoritmi – Lucrarea nr. 12

```
INSERT(A,N,X)// N trebuie sa fie pasat prin referinta
{
 A[N+1] = X  //adauga valoarea de inserat la sfirsitul vectorului
 N = N+1
 fiu = N
 parinte = N / 2           // impartire intreaga
 WHILE parinte>=1 DO       // parcurge ramura spre radacina
       |IF A[parinte]<A[fiu] THEN
       |   |SCHIMBA(A[parinte], A[fiu]); //retrogradeaza A[parinte]
       |   |fiu := parinte;             |
       |   |parinte := parinte div 2;   |  // urca spre radacina
       |ELSE parinte := 0;               // pentru parasirea buclei
}
```

Operatia REMOVE

Operatia **REMOVE** sterge din heap valoarea cea mai mare, valoare care se afla intotdeauna in radacina arborelui heap, deci in prima pozitie a heap-ului. Pentru a reface structura de heap se aduce in prima pozitie ultimul element din vector si apoi se retrogradeaza pina se reface structura de heap. La retrogradarea cu un nivel valoarea din radacina este schimbata cu descendentul care are valoarea maxima.



Iata algoritmul:

```
REMOVE(A,N)              // N trebuie sa fie pasat prin referinta
{
 IF N=0 THEN eroare
 ELSE |ret_val = A[1];    //Valoarea de returnat
      |A[1] = A[N];       //Aduce ultimul element in radacina
      |N = N-1;           //Micsoreaza heapul
      |parinte = 1;
      |fiu = 2;
      |WHILE fiu<=N DO    //parcurge o ramura in sens descendent
      |     |IF (fiu+1<=N) and (A[fiu]<A[fiu+1]) THEN
      |     |     fiu = fiu+1;     // este ales fiu cel mai mare
      |     |IF A[fiu]>A[parinte] THEN   // daca parinte<max(fii)
      |     |     |SCHIMBA(A[parinte],A[fiu]); // retrogradeaza
      |     |     |parinte := fiu;     |
      |     |     |fiu := fiu*2;       |        // coboara
      |     |ELSE fiu=N+1;  // pentru parasirea buclei
}
```

3. Heapsort

Heapsort este un algoritm de sortare care nu necesita memorie suplimentara (sorteaza vectorul pe loc), care functioneaza in doua etape:

[A] transfomarea vectorului de sortat intr-un arbore heap;

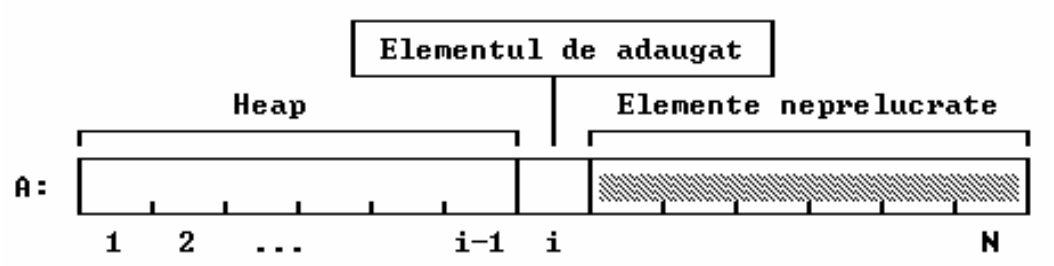
[B] reducerea treptata a dimensiunii heap-ului prin scoaterea valorii din radacina. Valorile rezultate in ordine descrescatoare sint plasate in vector pe spatiul eliberat, de la dreapta la stinga.

[A] Construirea heap-ului se poate face urmind doua strategii

[A1] Construirea heap-ului de sus in jos:

Vectorul va fi parcurs de la stinga la dreapta, corespunzator unei deplasari in arbore de sus in jos.

Pentru adaugarea elementului din pozitia **i** in vector se presupune ca elementele **A[1..i-1]** formeaza un heap.



Laborator de Structuri de Date si Algoritmi – Lucrarea nr. 12

Adaugarea elementului **A[i]** este realizata prin intermediul operatiei **INSERT**:

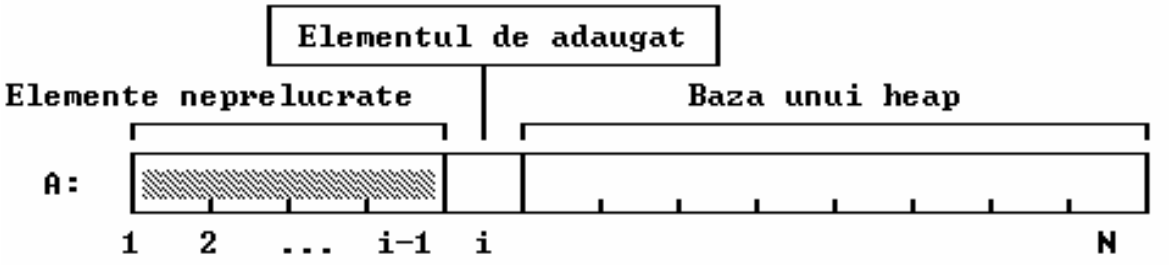
```
BUILD_HEAP(A,N)
{
   FOR i = 2 TO N DO
      INSERT(A,i-1,A[i])
}
```

Construirea heap-ului prin aceasta metoda este un algoritm de complexitate **O(n·log(n))** (Vezi demonstratia in curs).

[A1] Construirea heap-ului de jos in sus:

Vectorul va fi parcurs de la dreapta la stinga, corespunzator unei deplasari in arbore de jos in sus. Pentru adaugarea elementului din pozitia **i** in vector se presupune ca elementele **A[i+1..N]** respecta conditia care defineste un heap, constituind "baza" (elementele de pe nivelele inferioare ale) unui heap, deci:

$$\begin{cases} A[j] \geq A[j*2] & \text{si} \\ A[j] \geq A[j*2+1], & \text{pentru } i+1 \leq j \leq [N/2] \end{cases}$$



Elementul adaugat, **A[i]**, va fi retrogradat la fel cu modul de retrogradare folosit in procedura **REMOVE**.

```
BUILD_HEAP(A,N)
{
   FOR i = [N/2] TO 1 STEP -1 DO
      RETRO(A,N,i)
}

RETRO(A,N,i)
{
   parinte := i;
   fiu := 2*i;
   WHILE fiu<=N DO  // parcurge o ramura in sens descendent
        |IF (fiu+1<=N) and (A[fiu]<A[fiu+1]) THEN
        |     fiu := fiu+1;     // este ales fiu cel mai mare
```

```
        |IF A[fiu]>A[parinte] THEN // daca parinte<max(fii)
        |     |SCHIMBA(A[parinte],A[fiu]);  // retrogradeaza
        |     |parinte := fiu;     |
        |     |fiu := fiu*2;       | // coboara
        |ELSE fiu:=N+1;   // pentru parasirea buclei
}
```

Construirea heap-ului prin aceasta metoda este un algoritm de complexitate **O(n*log n)** (Vezi demonstratia in curs).

TEMA

1. Implementati operatiile de ***inserare*** si ***stergere*** dintr-un heap. Implementati urmatorul algoritm de sortare: Valorile de sortat sint inserate intr-un heap, apoi sint sterse pina la golirea heap-ului rezultind ordinea descrescatoare.

2. Implementati algorimul de sortare Heapsort.