**Arbori binari de căutare**

**Definiţie**: numim **arbore binar de căutare** (**ABC**) un arbore binar în care orice nod, cu excepţia nodurilor terminale, se bucură de proprietatea că valoarea cheii este mai mare ca informaţia din nodul fiu stâng şi mai mică decât informaţia din nodul fiu drept.

Într-un **ABC** informaţia din noduri este unică. În figura următoare este prezentat un arbore binar de căutare.

**50**

**60**

**55**

**65**

**70**

**80**

**90**

**30**

**17**

**10**

**15**

**20**

O proprietate foarte importantă a **ABC**-ului este aceea că oferă posibilitatea ordonării crescătoare a valorilor memorate în noduri prin parcurgerea arborelui în **inordine**.

**Crearea unui ABC** se realizează adăugând intr-un arbore iniţial vid, pe rând, câte un nod. Ideea este următoarea:

* dacă arborele este vid, se creează rădăcina
* dacă arborele nu este vid, se compară informaţia noului nod cu informaţia din nodul curent. Dacă este mai mare, se reia procesul pentru subarborele drept; dacă este mai mică se reia procesul pentru subarborele stâng.

Următoarea secvenţă de program realizează adăgarea unui nod într-un **ABC** după metoda descrisă mai sus:

**void Adauga(ABC\* &r,int info)**

**{**

**if(!r)** //arborele este vid, creez radacina

**{**

**r=new ABC; r->inf=info; r->st=r->dr=0;**

**}**

**else**

**if(info<r->inf) Adauga(r->st,info);**

**else if(info>r->inf) Adauga(r->dr,info);**

**else cot<<"\ninformatie duplicat\n";**

**}**

Parametrii funcţiei **Adauga()** reprezintă nodul curent la un moment dat. Deoarece se alocă memorie în heap pentru un nod nou, adresa acestuia este transmisă prin referinţă.

**Căutarea unei informaţii într-un ABC** se bazează pe ideea construcţiei arborelui. Se compară informaţia cu cea din nodul curent. Dacă sunt egale, căutarea se încheie, dacă nu, căutarea continuă în subarborele stâng sau în subarborele drept după cum valoarea căutată este mai mică, respectiv mai mare decât informaţia din nodul curent. În cazul în care se ajunge într-un subarbore vid (adresă nulă), informaţia căutată nu se găseşte în **ABC** şi se afişează un mesaj. Funcţia următoare implementează acestă metodă de căutare.

**void Cauta(ABC \*p,int x)**

**{**

**if(!p)**

**cout<<"\ninformatia %d nu exista in arbore\n",x);**

**else**

**if(x<p->inf) Cauta(p->st,x);**

**else if(x>p->inf) Cauta(p->dr,x);**

**else cout<<"\ninformatia “<<x<<” exista in arbore\n";**

**}**

**Ştergerea unui nod dintr-un ABC** este cea mai complexă operaţie. Mai întâi se caută in **ABC** informaţia ce trebuie ştearsă. În acest scop se foloseşte funcţia **Cauta\_sterge()**. Pe lângă adresa nodului care se şterge (pointerul **ns**), această funcţie furnizează şi adresa nodului părinte al nodului căutat (pointerul **tns**), adresă care va fi necesară pentru refacerea legăturilor. Odată determinat nodul care trebuie şters (mai exact adresa sa), există trei cazuri determinante pentru modul în care se va face ştergerea:

1. nodul nu are descendenţi, caz în care legătura stngă sau dreaptă a nodului tată se va modifica în NULL
2. nodul are un singur descendent, caz în care legătura stângă sau dreaptă a nodului tată se şterge şi nodul tată va prelua legătura către descendentul fiului său
3. nodul are doi descendenţi, caz în care ştergerea se va reduce la cele două cazuri descrise anterior astfel:

* se caută în subarborele stâng al nodului care se şterge nodul cu valoarea minimă şi se reţine şi adresa nodului tată al acestuia. Funcţia care realizează acest lucru este Determina\_minim().

Exemplu: din arborele următor dorim să ştergem nodul cu cheia 50

**minim**

**90**

**80**

**65**

**70**

**60**

**55**

**17**

**10**

**30**

**15**

**20**

**50**

* se interschimbă valoarea din nodul ce trebuie şters (50) cu valoarea din nodul ce reţine minimul determinat anterior (10)
* se şterge nodul care conţine acum valoarea (50). Acest nod este fără descendenţi sau are doar descendent drept. El nu poate avea descendent stâng pentru că reprezintă cea mai mică valoare din subarborele stâng al nodului ce trebuie şters

**10**

**15**

**60**

**20**

**17**

**50**

**30**

**55**

**70**

**65**

**80**

**90**

* se reface structura de ABC, dacă este cazul, prin retrogradarea valorii minime aduse din nodul care s-a şters

**10**

**15**

**60**

**20**

**17**

**30**

**55**

**70**

**65**

**80**

**90**

Programul următor gestionează operaţiile de creare şi prelucrare a unui **ABC**. Se crează un arbore iniţial din **n** valori citite de la tastatură. Prin intermediul unor opţiuni de meniu se poate realiza în mod repetat adăugarea unui nod în **ABC**, căutarea unei valori **x**, ştergerea unei valori **x** şi afişarea nodurilor din arbore în inordine (ordonate crescător după chei).

**#include<iostream.h>**

**typedef struct nod{**

**int inf;**

**nod \*st,\*dr;**

**}ABC;**

**ABC \*tns;**

**void Adauga(ABC\* &r,int info)**

**{**

**if(!r)** //arborele este vid, creez radacina

**{**

**r=new ABC; r->inf=info; r->st=r->dr=0;**

**}**

**else**

**if(info<r->inf) Adauga(r->st,info);**

**else if(info>r->inf) Adauga(r->dr,info);**

**else cout<<"\ninformatie duplicat\n";**

**}**

**void Creare(ABC\* &r)**

**{**

**int info,n,i;**

**cout<<"numar de noduri: "; cin>>n;**

**for(i=1;i<=n;i++)**

**{**

**cout<<"informatia nodului <<i<<”: "; cin>>info;**

**Adauga(r,info);**

**}**

**}**

**void Cauta(ABC \*p,int x)**

**{**

**if(!p)**

**cout<<"\ninformatia “<<” nu exista in arbore\n";**

**else**

**if(x<p->inf) Cauta(p->st,x);**

**else if(x>p->inf) Cauta(p->dr,x);**

**else cout<<"\ninformatia “<<x<<” exista in arbore\n";**

**}**

**ABC\* Determina\_minim(ABC \*p)**

**{**

**while(p->st) { tns=p; p=p->st; }**

**return p;**

**}**

**ABC\* Cauta\_sterge(ABC \*p,int x)**

**{**

**if(!p) return 0;**

**else if(p->inf<x) { tns=p; p=p->dr; return Cauta\_sterge(p,x); }**

**else if(p->inf>x) { tns=p; p=p->st; return Cauta\_sterge(p,x); }**

**else return p;**

**}**

**void Sterge(ABC\* &r,int x)**

**{**

**ABC \*ns,\*minim,\*ns1; int aux;**

**ns=Cauta\_sterge(r,x);**

**if(!ns) cout<<"\ninformatia “<<x<<” exista in arbore\n";**

**else**

**if(ns->dr==ns->st)** //nodul ce trebuie sters nu are descendenti

**{**

**if(ns->inf<tns->inf) tns->st=0;**

**else tns->dr=0;**

**delete ns;**

**}**

**else**

**if(ns->dr==0&&ns->st)** //nodul are fiu stang

**{**

**if(tns->inf<ns->inf) tns->dr=ns->st;**

**else tns->st=ns->st;**

**delete ns;**

**}**

**else**

**if(ns->st==0&&ns->dr)** //nodul are fiu drept

**{**

**if(tns->inf>ns->inf) tns->st=ns->dr;**

**else tns->dr=ns->dr;**

**delete ns;**

**}**

**else** //nodul ce trebuie sters are ambii fii

**{**

**minim=Determina\_minim(ns); ns1=ns;**

**ns->inf=minim->inf;**

**if(minim->dr==minim->st)**

//nodul minim nu are descendenti

**{ tns->st=0; delete minim; }**

**else** //nodul minm are fiu drept

**{ tns->st=minim->dr; delete minim; }**

**while(ns1->st&&ns1->inf<ns1->st->inf)**

**{**

**aux=ns1->inf; ns1->inf=ns1->st->inf;**

**ns1->st->inf=aux; ns1=ns1->st;**

**}**

**}**

**}**

**void SRD(ABC \*r)**

**{**

**if(r)**

**{**

**SRD(r->st); cout<<r->inf; SRD(r->dr);**

**}**

**}**

**void main()**

**{**

**ABC \*rad=0; int x,opt;**

**Creare(rad); //se creaza ABC-ul initial**

**do{**

**cout<<"\n\n\tOptiuni:\n 1-Cauta o valoare in ABC\n 2-Adauga un**

**nod in ABC\n";**

**cout<<" 3-Sterge un nod din ABC\n 4-Lista nodurilor in inordine**

**\n 5-Iesire\n";**

**cout<<"optiunea: "; cin>>opt;**

**switch(opt)**

**{**

**case 1: cout<<"valoarea: "); cin>>x;**

**Cauta(rad,x); break;**

**case 2: cout<<"valoarea: "); cin>>x;**

**Adauga(rad,x); break;**

**case 3: cout<<"valoarea: "); cin>>x;**

**Sterge(rad,x); break;**

**case 4: SRD(rad);**

**}**

**}while(opt!=5);**

**}**

**9. Sortarea cu ansamble**

Sortarea cu ansamble face parte din clasa strategiilor de sortare prin selecţie. Acestea au la bază selectarea pe rând a elementelor şi plasarea acestora pe locul lor final. Sortarea cu ansamble se bazează pe o structură de date de tip arbore binar, reprezentat printr-un tablou unidimensional.

Fie **A** un vector cu **n** elemente, **n∈N\***. Vom asocia mai întâi vectorului **A**, în mod unic un arbore binar.

**Definiţie**: Fiind dat **n∈N\***, se numeşte arbore binar ataşat lui **n** arborele binar

**Hn=({1,2,…,n},E)** cu **E={([i/2],i), ∀i∈{1,2,…n}}**

Pentru **n=10**, **H10** este reprezentat mai jos:

**1**

**2**

**3**

**6**

**7**

**5**

**4**

**10**

**9**

**8**

**Observaţii**:

1. dacă **2i≤n**, vârful **i** are descendent stâng, şi anume vârful **2i**; dacă **2i+1≤n**, vârful **i** are descendent drept, şi anume vârful **2i+1**
2. orice vârf **i** (**2≤i≤n**) are ca vârf tată nodul **[i/2]**
3. **Hn** are **p=[log2n]+1** nivele. Pe fiecare nivel **k** (**k≤n**) sunt **2k** vârfuri (etichetate cu **2k,2k+1,….2k+1-1**), cu posibila excepţie a ultimului nivel care poate fi incomplet (acesta va conţine **2p**,**2p+1**,…..,**min(2p+1-1,n)**).

**Definiţie**: Fie **A** un vector cu **n** elemente, **n∈N\***. Se numeşte arbore binar ataşat lui **A**, arborele binar **Hn(A)**, obţinut din **Hn** prin etichetarea vârfurilor **i** cu **A[i]**, **∀i∈{1,2,…,n}**.

**Exemplu**: Vectorului **A=(10,20,15,8,7,12)** i se ataşează arborele binar **H6(A)** din figura de mai jos.

**10**

**20**

**15**

**12**

**7**

**8**

**A[1]**

**A[3]**

**A[6]**

**A[5]**

**A[4]**

**A[2]**

**Definiţie**: Vectorul **A** cu **n** elemente selectate dintr-o mulţime total ordonată (cu relaţia **≤**), formează un ***ansamblu*** (***grămadă binară*** sau ***heap***) dacă **∀i∈{2,....,n} , A[i]≤A[i div 2]**.

Din perspectiva arborelui binar ataşat vectorului **A**, definiţia anterioară se traduce astfel: vectorul **A** cu **n** elemente formează un heap, dacă pentru orice nod al arborelui binar ataşat, valoarea asociată acestui nod este mai mare sau egală cu valorile asociate descendenţilor săi (dacă aceştia există).

Vectorul **A** din figura de mai sus nu este un ansamblu, deoarece **A[1]<A[2]** . În schimb vectorul **B=(20,10,15,8,7,12)** este un ansamblu şi are arborele binar **H6(B)** prezentat în figura de mai jos.

**20**

**10**

**15**

**12**

**7**

**8**

**B[1]**

**B[3]**

**B[6]**

**B[5]**

**B[4]**

**B[2]**

Deducem că într-un ansamblu, valoarea lui **A[1]** este mai mare sau egală decât toate elementele vectorului şi ca urmare, în arborele binar asociat ansamblului, rădăcina conţine valoarea maximă. Aceeaşi proprietate se deduce pentru valoarea **A[i]** şi subarborele a cărui rădăcină este etichetată cu **A[i]**.

Ansamblele pot fi utilizate pentru a realiza sortarea unui şir **A** de **n** valori numerice, metodă cunoscută sub numele de **sortare cu ansamble** (**heap-sort**). Dacă am reuşit să transformăm şirul **A** într-un heap, cel mai mare element al său este în nodul rădăcină; îl putem lua de aici şi plasa pe locul său definitiv în ordinea crescătoare dorită, aducând în locul său elementul de pe ultimul loc. Reconstituim apoi un heap cu **n-1** elemente salvând ce se mai poate salva din heap-ul anterior. Se repetă acest lucru până când rămâne un singur element care, el singur, formează un heap. Paşii mari ai algoritmului sunt:

* **creare heap**
* **for(i=n;i>=2;i--)**

**{ \* schimba a[1] cu a[i]**

**\* ajustarea heap-ului cu i-1 elemente**

}

Trebuie să rezolvăm două probleme: crearea heap-ului iniţial şi ajustarea lui în fazele următoare. Pentru **a** **construi** heap-ul plecăm de la **A[1]** care poate fi privit ca un heap şi adăugăm pe rând câte un element (**A[2], A[3],......,A[n]**), la fiecare adăugare structura trebuiind să rămână un heap cu un element în plus. Tehinica de adăugare presupune un şir de verificări, elementul adăugat **i** fiind descendentul nodului **[i/2]**. Dacă faţă de părintele său, nodul **i** are valoarea cel mult egală, adăgarea este terminată. Dacă nu, apare necesar un şir de schimbări de valori pe verticală, urmate de noi verificări, între descendenţi şi părinţi, până ce condiţia este satisfăcută sau s-a făcut o ultimă schimbare între nodul 1 şi un descendent al său.

**Exemplu**: Fie **n=6** şi **A=(3,5,7,6,4,8)**. Etapele construirii heap-ului sunt:

* **A[1]=3** formează un heap de lungime 1
* Se adaugă **A[2]=5** şi **A[1] ↔A[2]**
* Se adaugă **A[3]=7** şi **A[1] ↔A[3]**
* Se adaugă **A[4]=6** şi **A[4] ↔A[2]**
* Se adaugă **A[5]=4**
* Se adaugă **A[6]=8** şi **A[6] ↔A[3]** , **A[3] ↔A[1]**

Se obţine vectorul **A=(8,6,7,3,4,5)**.

În pseudocod crearea heap-ului se face astfel:

**pentru j=2,n** // adauga in heap a[j]

**copil🡨j; parinte🡨[copil/2]**

**cat\_timp(parinte≥1)**

**daca a[copil]>a[parinte]**

**atunci a[copil] ↔a[parinte]**

**copil🡨parinte**

**parinte🡨[copil/2]**

**altfel parinte🡨0**

**sfarsit\_cat\_timp**

**sfarsit\_pentru**

Corectarea heap-ului este necesară în faza a doua, când elementele se pot plasa pe locul lor definitiv în ordine crescătoare şi când, utilizând o deplasare de sus în jos, un element care a fost adus în nodul rădăcină şi nu este cel puţin egal cu oricare dintre descendenţii săi, trebuie deplasat la locul său în arbore. Dacă am ajuns cu ajustarea la nodul **i** (iniţial **i=1**), verificăm dacă **i** are descendenţi; dacă are doi fii, îl determinăm pe acela care conţine cea mai mare valoare. Dacă relaţia dintre nodul **i** şi acest fiu este satisfăcută, ajustarea este terminată. Dacă nu, se face schimbarea pe verticală între valorile asociate celor două noduri şi se continuă analog, considerând drept nod părinte descendentul determinat anterior.

**Exemplu**: Fie heap-ul **A=(8,6,7,3,4,5)** construit anterior. Ajustarea se face în următorii paşi:

* **A[1] ↔A[6]; A[1] ↔A[3] 🡪A=(7,6,5,3,4,8)**
* **A[1] ↔A[5]; A[1] ↔A[2] 🡪A=(6,4,5,3,7,8)**
* **A[1] ↔A[4]; A[1] ↔A[3] 🡪A=(5,4,3,6,7,8)**
* **A[1] ↔A[3]; A[1] ↔A[2] 🡪A=(4,3,5,6,7,8)**
* **A[1] ↔A[2] 🡪A=(3,4,5,6,7,8)**

În pseudocod ajustarea heap-ului se face astfel:

**parinte🡨1; copil🡨2**

**cat\_timp (copil≤i-1)**

**daca (copil+1≤i-1) and (A[copil+1]>A[copil])**

**atunci copil🡨copil+1**

**sfarsit\_daca**

**daca A[parinte]<A[copil]**

**atunci A[copil] ↔A[parinte]**

**parinte🡨copil**

**copil🡨2\*parinte**

**altfel copil🡨i**

**sfarsit\_daca**

**sfarsit\_cat\_timp**

Algoritmul heapsort are complexitatea **Φ(n\*log2n)**. Programul corespunzător pentru implementarea acestei metode de sortare este :

**#include<stdio.h>**

**#define N 50**

**int a[N],n;**

**void schimba(int& x,int& y)**

**{**

**int aux=x; x=y; y=aux;**

**}**

**void creare\_heap()**

**{**

**int parinte,copil,j;**

**for(j=2;j<=n;j++)**

**{**

**copil=j; parinte=copil/2;**

**while(parinte>=1)**

**if(a[copil]>a[parinte])**

**{**

**schimba(a[copil],a[parinte]);**

**copil=parinte; parinte=copil/2;**

**}**

**else**

**parinte=0;**

**}**

**}**

**void ajustare\_heap(int i)**

**{**

**int parinte=1,copil=2;**

**while(copil<=i-1)**

**{**

**if(copil+1<=i-1 && a[copil+1]>a[copil]) copil++;**

**if(a[parinte]<a[copil])**

**{**

**schimba(a[parinte],a[copil]);**

**parinte=copil; copil=2\*parinte;**

**}**

**else**

**copil=i;**

**}**

**}**

**void main()**

**{**

**int i;**

**cout<<"n="; cin>>n;**

**for(i=1;i<=n;i++)**

**{**

**cout<<"a[%d]=",i);**

**cin>>a[i];**

**}**

**cout<<"\n\tVectorul initial este:\n";**

**for(i=1;i<=n;i++) cout<<a[i]<<” ”;**

**cout<<"\n\n";**

**creare\_heap();**

**for(i=n;i>=2;i--)**

**{**

**schimba(a[1],a[i]);**

**ajustare\_heap(i);**

**}**

**cout<<"\n\tVectorul sortat este:\n";**

**for(i=1;i<=n;i++) cout<<a[i]<<” ”;**

**cout<<"\n";**

**}**

O varianta recursiva pentru algoritmul **heapsort** este:

**#include<stdio.h>**

**int a[50],n;**

**void Combinare(int vf,int n)**

**{**

**int baza=2\*vf,valoare=a[vf],gata=0;**

**while(baza<=n && !gata)**

**{**

**if(baza<n && a[baza]<a[baza+1]) baza++;**

**if(valoare<a[baza])**

**{**

**a[vf]=a[baza]; vf=baza; baza\*=2;**

**}**

**else gata=1;**

**}**

**a[vf]=valoare;**

**}**

**void Heap()**

**{**

**int i;**

**for(i=n/2;i>=1;i--) Combinare(i,n);**

**}**

**void Citeste()**

**{**

**int i;**

**cout<<"n="; cin>>n;**

**for(i=1;i<=n;i++)**

**{**

**cout<<"a[“<<i<<”]="; cin>>a[i];**

**}**

**}**

**void HeapSort()**

**{**

**int i,x;**

**Heap();**

**for(i=n;i>=2;i--)**

**{**

**x=a[1]; a[1]=a[i]; a[i]=x;**

**Combinare(1,i-1);**

**}**

**}**

**void main()**

**{**

**int i;**

**Citeste(); HeapSort();**

**cout<<"\nVectorul sortat: ";**

**for(i=1;i<=n;i++) cout<<a[i]<<” ”;**

**cout<<"\n");**

**}**

Funcţia **Combinare()** formează un ansamblu (**heap**) crescător dintr-un vârf şi alte două **heap**-uri, iar funcţia **Heap()** organizează vectorul ca **heap** crescător (sau **MinHeap**).