Structuri de date arborescente

Heap-uri

Situații practice impun în mod frecvent alegerea dintr-o mulțime dinamică de valori a uneia sau unora care îndeplinesc anumite condiții. Spre exemplu, o firmă are în vedere spre onorare, în primul rând comenzile cele mai rentabile. Este deci necesar, ca o structură de date dinamică adecvată, să ofere cu „efort" minim de prelucrare, informația cerută de un anumit criteriu de optim. Constatăm că este vorba despre *selectarea* unor informații dintr-un volum de date, organizate după un criteriu stabilit. O astfel de facilitate de prelucrare este oferită de o categorie de *arbori binari*, cunoscuți în literatura de specialitate sub numele de *heap*-uri.

Min-heapuri şi max-heapuri

Definiție
Un *max-heap* este un arbore binar complet în care valoarea memorată în orice nod al său este mai mare sau egală decât valorile memorate în nodurile fii ai acestuia.

Corespunzător se poate defini *min-heap*-ul, ca un arbore binar complet în care valoarea memorată în orice nod este mai mică sau egală decât valorile memorate în nodurile fii ai acestuia.

În cele ce urmează, un *max-heap* va fi numit *heap*, urmând ca atunci când este vorba despre un *min-heap*, aceasta să se precizeze în mod explicit.

Un *heap* fiind un arbore binar complet, reprezentarea cea mai adecvată este reprezentarea secvențială. Dacă *heap*-ul are `n` noduri, numerotate de la 1 la `n`, va fi suficient să reținem informațiile asociate nodurilor într-un vector cu `n` elemente, relațiile dintre noduri fiind implicite. Mai exact, dacă `x` este un nod oarecare din *heap*, fiul stâng al lui `x`, fiul drept, respectiv nodul părinte al lui `x` pot fi determinați cu formulele:

$$st(x)=\begin{cases}2x, & \text{dacă } 2x \le n\\ \text{nu există}, & \text{dacă } 2x > n\end{cases} \qquad dr(x)=\begin{cases}2x+1, & \text{dacă } 2x+1 \le n\\ \text{nu există}, & \text{dacă } 2x+1 > n\end{cases}$$

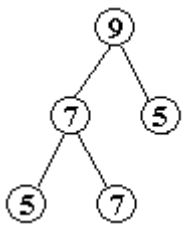
$$tata(x)=\begin{cases}[x/2], & \text{dacă } x > 1\\ \text{nu există}, & \text{dacă } x = 1\end{cases}$$

Utilizând reprezentarea secvențială, putem reformula definiția unui *heap* astfel:

Definiție
Tabloul `H[1..n]` formează un *max-heap* dacă `H[i]≤H[i/2]`, `∀i∈{2, 3, ..., n}`.

Tabloul `H[1..n]` formează un *min-heap* dacă `H[i]≥H[i/2]`, `∀i∈{2, 3, ..., n}`.

De exemplu, arborele binar complet din figură este un *max-heap*.



şi va fi reprezentat implicit:

1	2	3	4	5
9	7	5	5	7

Crearea unui heap prin inserări succesive

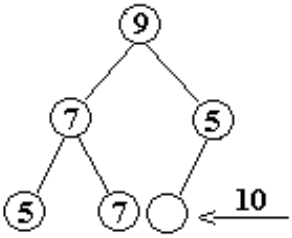
O primă soluţie este de a insera succesiv elementele în *heap*, plecând iniţial de la *heap*-ul format dintr-un singur element:

```
void CreareHeap()
//organizeaza ca heap vectorul global H, inserând succesiv elemente
{for (int i=2; i≤=n; i++) InsertHeap(i-1, H[i]); }
```

Inserarea unui nod într-un heap

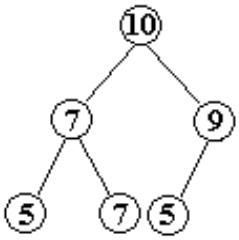
Fie `H[1..n]` un *heap* cu `n` elemente şi `x` valoarea ce trebuie inserată. Valoarea `x` va fi memorată în *heap* pe poziţia `n+1`, apoi se restaurează eventual *heap*-ul, promovând" valoarea `x` spre rădăcină, până când proprietatea de *heap* este restabilită.

De exemplu, să inserăm valoarea `10` în *heap*-ul din figură:



Valoarea inserată va ocupa poziţia `6`. Pentru a conserva proprietatea de *heap*, valoarea `10` trebuie promovată până în rădăcină, valorile de pe drumul parcurs spre rădăcină fiind retrogradate succesiv, cu un nivel.

Rezultă arborele:



Modificările sunt aplicate de fapt vectorului ce constituie reprezentarea implicită:

1	2	3	4	5	6
10	7	9	5	7	5

```
void InsertHeap (int n, TipCheie x)
//inserează valoarea x în heap-ul H de dimensiune n
{
 int fiu=++n; //fiu indica pozitia pe care trebuie plasata valoarea x
 int tata=n/2;
 while (tata && H[tata]<x) // valoarea x trebuie "promovata"
        {H[fiu]=H[tata];
         fiu=tata; tata=fiu/2;}
H[fiu]=x;}
```

Observaţie

Analizând *complexitatea* procedurii de inserare deducem că, în cazul cel mai defavorabil, valoarea nou inserată urcă până în rădăcina arborelui, deci necesită `O(h)` operaţii elementare, unde `h` este înălţimea *heap*-ului. Un *heap* fiind un arbore binar complet, înălţimea sa `h` va fi [$\log_2 n$], deci inserarea unui nod într-un *heap* cu `n` elemente este de `O(log n)`.

Să estimăm complexitatea în cazul cel mai defavorabil a procedurii de construcţie a *heap*-ului prin inserări succesive. Dacă toate cele 2^i valori de pe nivelul `i` urcă până în rădăcină, obţinem:

$$\sum_{i=1}^{[\log n]} i \cdot 2^i < \log n \sum_{i=1}^{[\log n]} 2^i = O(n \log n)$$

Crearea unui heap prin combinări succesive

O strategie mai eficientă de construcţie a unui heap se bazează pe ideea de echilibrare. Apelul funcţiei `InsertHeap(n,x)` poate fi interpretat ca o combinare de două *heap*-uri: un *heap* cu `n` elemente şi un *heap* format numai din elementul `x`. Putem construi *heap*-ul cu rădăcina `H[i]` combinând la fiecare pas `i` două *heap*-uri de dimensiuni apropiate: *heap*-ul cu rădăcina `2i` cu heap-ul cu rădăcina `2i+1` şi cu elementul `H[i]`.

```
void CreareHeap()
//organizeaza ca heap vectorul global H
{for (int i=n/2; i; i--) CombHeap(i,n);}
```

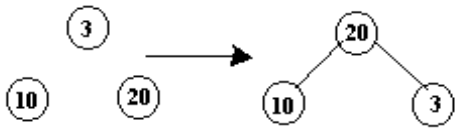
Funcţia `CombHeap(i,n)` combină elementul `H[i]` cu *heap*-ul cu rădăcina `2i` şi cu *heap*-ul cu rădăcina `2i+1`.

```
void CombHeap (int i, int n)
{
 TipCheie v = H[i];
 int tata=i;           //tata indica pozitia pe care va fi plasata valoarea v
 int fiu=2*i;                                                    //fiul stang
 while (fiu<=n)
    {if (fiu<n)                                                 //exista fiu drept
       if (H[fiu]<H[fiu+1]) fiu++; //fiu indica fiul cu valoarea cea mai mare
     if (v<H[fiu])
        {H[tata]=H[fiu]; // valoarea celui mai mare dintre fii urca in arbore
         tata=fiu;                                          //v coboara în arbore
         fiu=fiu*2;}
         else
         fiu=n+1;                    //am determinat pozitia corecta pentru v
    }
H[tata]= v;}
```

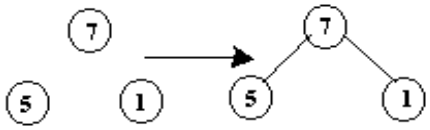
De exemplu, organizarea ca *heap* a vectorului `H={0,7,3,5,1,10,20}` se face în `3` paşi.

La pasul `i=3` se apelează `CombHeap(3,7)`:

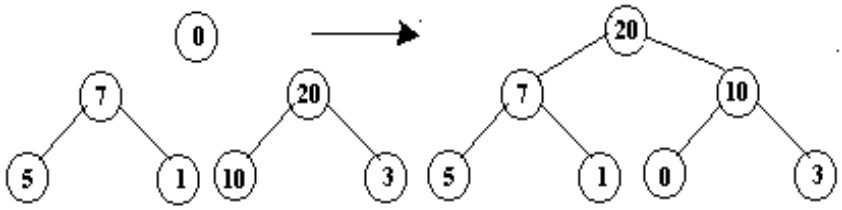
Deci `H={0,7,20,5,1,10,3}`.



La pasul `i=2` se apelează `CombHeap(2,7)`:



La pasul `i=1`, se apelează `CombHeap(1,7)`:



Deci *heap*-ul în reprezentare implicită este `H={20,7,10,5,1,0,3}`.

Complexitatea algoritmului

Notăm $h=[\log_2 n]$ înălţimea *heap*-ului. Pentru fiecare nod `x` de pe nivelul `i`, $i \in \{0, 1, ..., h-1\}$, pentru a construi *heap*-ul cu rădăcina `x` se fac cel mult `h-i` coborâri a valorii `x` în arbore. Deci, în cazul cel mai defavorabil, numărul de operaţii elementare efectuate de algoritm pentru a construi un *heap* cu `n` elemente este:

$$T_n \le \sum_{i=0}^{h-1} 2^i (h-i) = \sum_{j=1}^{h} j \cdot 2^{h-j} = \sum_{j=1}^{h} 2^h \cdot \frac{j}{2^j} \le n \sum_{j=1}^{h} \frac{j}{2^j} < 2n = O(n)$$

Algoritmul de construcție a unui *heap* a devenit astfel liniar.

Heapsort

O primă aplicație este sortarea unui vector cu ajutorul *heap*-urilor, metodă cunoscută sub denumirea de *heapsort*.

Primul pas este de a organiza vectorul ca un *heap*. Deoarece, conform proprietății de *heap*, elementul maxim din vector este plasat în rădăcina *heap*-ului, deci pe poziția `1`, el poate fi plasat pe poziția sa corectă, interschimbându-l cu elementul de pe poziția `n`. Noul element din rădăcina *heap*-ului nu respectă proprietatea de *heap*, dar subarborii rădăcinii rămân *heap*-uri. Prin urmare trebuie restaurat *heap*-ul, apelând funcția `CombHeap(1,n-1)`. Elementul de pe poziția `n` fiind deja pe poziția sa corectă nu mai este inclus în *heap*. Procedeul se repetă până când toate elementele vectorului sunt plasate pe pozițiile lor corecte. Funcția `HeapSort()` ordonează crescător vectorul global `H` cu `n` elemente:

```
void HeapSort()
{ TipCheie aux;
  CreareHeap();
  for (int i=n; i>1; i--)           //plaseaza corect un element pe pozitia i
      { aux=H[1]; H[1]=H[i]; H[i]=aux;  //interschimba elementul H[i] cu H[1]
        CombHeap(1, i-1); }                          //restaureaza heap-ul
}
```

Complexitatea algoritmului

Complexitatea algoritmului *heapsort* este *de* `O(n log n)`, crearea *heap*-ului fiind liniară, iar fiecare dintre cele `n-1` apeluri ale funcției `CombHeap()` fiind de `O(log n)`. Deci algoritmul de sortare cu ajutorul *heap*-urilor este optimal în cazul cel mai defavorabil.

Cozi cu prioritate

Algoritmul *heapsort* este un algoritm optimal în cazul cel mai defavorabil, dar de obicei în practică este utilizat algoritmul de sortare rapidă (*quicksort*), chiar dacă acesta este optimal doar în medie. Cea mai frecvent utilizată aplicație a unui *heap* este implementarea cozilor cu prioritate.

Definiție

O *coadă cu prioritate* este o structură de date abstractă formată din elemente care au asociată o valoare numită cheie sau prioritate şi care suportă următoarele operații:

- `Insert(Q, x)`: inserează elementul `x` în coada cu prioritate `Q`;
- `ExtractMax(Q)`: extrage din coada cu prioritate `Q` elementul de valoare maximă.

Observație

În mod analog se poate defini o coadă cu min-prioritate, pentru care interesează operația de extragere din coadă a elementului de prioritate minimă.

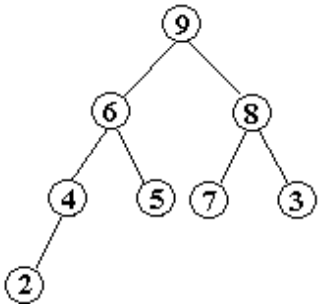
Există multe aplicații ale cozilor cu prioritate. De exemplu, planificarea execuției programelor pe un calculator: coada cu prioritate reține programele ce trebuie executate în funcție de priorităţile lor relative. Când se încheie sau se întrerupe execuția unui program, programul cu cea mai mare prioritate din coadă este selectat şi lansat în execuție (operația `ExtractMax`). Adăugarea unui nou program în coadă se realizează cu operația `Insert`.

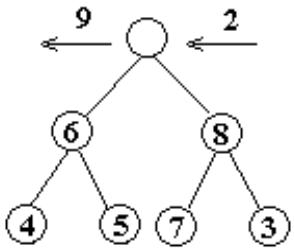
Există diverse modalități de a implementa o coadă cu prioritate: ca un vector, ca o listă înlănțuită, ordonată sau nu etc. Fiecare din aceste variante optimizează una din cele două operații, cealaltă realizându-se în timp liniar.

O structură de date simplă ce permite implementarea ambelor operații în timp logaritmic este *heap*-ul. Am prezentat şi analizat deja procedura `InsertHeap`, care realizează operația `Insert`. Extragerea elementului de prioritate maximă presupune ştergerea din *heap* a valorii din rădăcină. Pentru aceasta se

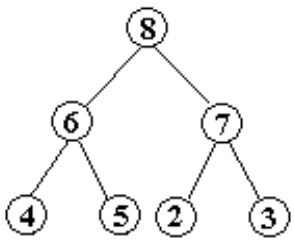
plasează în rădăcină (prima poziție din vector) ultimul element din *heap*. Evident, dimensiunea *heap*-ului scade, iar *heap*-ul trebuie restaurat. Cum subarborii rădăcinii şi-au păstrat structura de *heap*, restaurarea se face combinând valoarea din rădăcină cu *heap*-urile fii.

De exemplu, din *heap*-ul din figură se extrage nodul cu valoarea `9`, locul lui fiind ocupat de nodul cu valoarea 2.





Valoarea `8` este maximul dintre valorilor fiilor rădăcinii. Ca urmare, `8` îl înlocuieşte pe `2`, apoi `2` este înlocuit de `7`. Rezultă *heap*-ul următor:



În reprezentare implicită, *heap*-ul

1	2	3	4	5	6	7	8
9	6	8	4	5	7	3	2

a devenit

1	2	3	4	5	6	7
8	6	7	4	5	2	3

Funcția `ExtractMax(n,H)` extrage elementul de prioritate maximă din *heap*-ul `H` de dimensiune `n`:

```
TipCheie ExtractMax(int & n, Heap H)
{TipCheie v=H[1];                                  //retin valoarea din radacina
 H[1]=H[n--];
 CombHeap(1, n);
 return v;}                                                  //restaurez heap-ul
```

Complexitatea operației de extragere a elementului de cheie maximă se reduce la complexitatea algoritmului de combinare a două *heap*-uri, care este de `O(log n)`.

Min-max-heapuri

Pentru rezolvarea anumitor probleme este necesară uneori determinarea sau extragerea atât a valorilor minime cât şi a valorilor maxime.

Definiție

O *coadă cu dublă prioritate* este o structură de date abstractă formată din elemente care au asociată o valoare numită cheie sau prioritate şi care suportă următoarele operații:

- `Insert(Q,x)`: inserează elementul `x` în coada cu prioritate `Q`;
- `ExtractMax(Q)`: extrage din coada `Q` elementul de valoare maximă;
- `ExtractMin(Q)`: extrage din coada `Q` elementul de valoare minimă.

Această structură de date abstractă poate fi implementată eficient utilizând o structură de date numită *min-max-heap*.

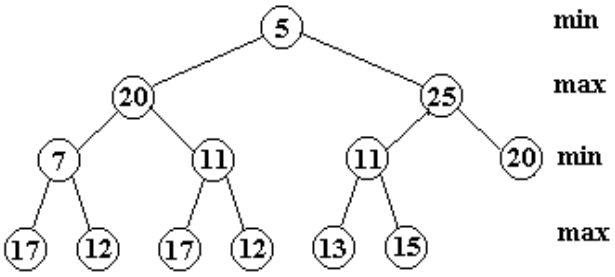
Definiție

Un *min-max-heap* este un arbore binar complet în care:

– nivelurile sunt, alternativ, *niveluri minime*, respectiv *niveluri maxime*, rădăcina aflându-se pe un nivel minim;
– dacă x este un nod situat pe un nivel minim, valoarea nodului x, este mai mică decât toate valorile din nodurile subarborelui de rădăcină x, nodul x numindu-se *nod minim*;
– dacă x este un nod situat pe un nivel maxim, valoarea sa este mai mare decât toate valorile din nodurile subarborelui de rădăcină x, nodul x numindu-se *nod maxim*.

Se constată că nivelurile cu număr de ordine par sunt niveluri minime, iar cele cu număr de ordine impar, niveluri maxime.

Un exemplu de *min-max-heap* este prezentat în figură:



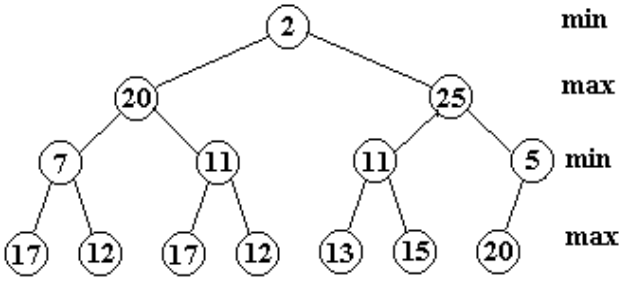
Min-max heap-urile fiind arbori binari compleți, reprezentarea cea mai eficientă este reprezentarea secvențială. Pentru *min-max heap*-ul din figură reprezentarea va fi:

1	2	3	4	5	6	7	8	9	10	11	12	13
5	20	25	7	11	11	20	17	12	17	12	13	15

Inserarea unui nod într-un min-max-heap

Noul nod se memorează pe prima poziție neocupată din vectorul ce constituie reprezentarea secvențială a *min-max heap*-ului şi apoi se restaurează *heap*-ul, *„promovând”* valoarea nou inserată spre rădăcină până ajunge în poziția în care structura de *min-max-heap* se reface. Evident, la inserare, dimensiunea *min-max heap*-ului creşte cu o unitate.

De exemplu, să inserăm nodul cu valoarea 2 în *min-max-heap*-ul din figură. Noul nod va ocupa poziția 14 în vector, tatăl său fiind nodul de pe poziția 7, având valoarea 20. Nodul tată este un nod de pe un nivel minim, deci valoarea sa trebuie să fie mai mică decât valorile nodurilor din subarbore. Din acest motiv nodul cu valoarea 2 „promovează” pe nivelul superior, ajungând în acest fel pe un nivel minim, iar valoarea nodului tată coboară în arbore. Nodul nou inserat se află pe poziția 7 pe un nivel minim şi continuăm compararea valorii sale cu valorile nodurilor situate pe niveluri minime, „promovând-ul” cât timp este necesar, pentru a conserva structura *min-max heap*-ului. Se obține arborele

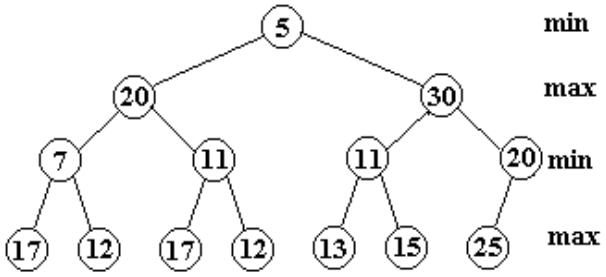


obținut din figură:

Reprezentarea implicită este:

1	2	3	4	5	6	7	8	9	10	11	12	13	13
2	20	25	7	11	11	5	17	12	17	12	13	15	20

Dacă în arborele inițial am insera un nod cu valoarea `30`, cum `30>20` (valoarea tatălui său, situat pe un nivel minim), trebuie să verificăm proprietatea de *min-max heap* relativ la nodurile maxime de pe drumul spre rădăcină. Deci comparăm valoarea nou inserată cu valoarea nodului de pe poziția `3`. Cum `25<30`, valoarea `30` „promovează" în arbore. Continuăm comparațiile cu toate nodurile maxime de pe drumul spre rădăcină, „promovând" noua valoare, până când structura *min-max heap*-ului este restaurată. Rezultă arborele:



Reprezentarea implicită este:

1	2	3	4	5	6	7	8	9	10	11	12	13	14
5	20	30	7	11	11	20	17	12	17	12	13	15	25

Funcția `Insert(n, H, x)` inserează valoarea `x` în *min-max heap*-ul `H`, de dimensiune `n`.

```
void Insert(int& n, MInMaxHeap H, TipCheie x)
{int niv, fiu, tata;
 H[++n]=x;
 if (n>1)
    {fiu=n;                                        //pozitia curenta a valorii x
     niv=[log2n];                          //nivelul pe care este inserat noul nod
     if (niv%2==0)                          //nodul este inserat pe un nivel minim
        {tata=n/2;                              //nodul tata situat pe nivel maxim
         if(x>H[tata])//compar cu nodurile maxime de pe  drumul spre radacina
            {while  (tata && x>H[tata])
                   { H[fiu]=H[tata];
                     fiu=tata;  tata=fiu/4; }
             H[fiu]=x; }
             else      //compar cu nodurile minime de pe drumul spre radacina
            {tata=fiu/4;
             while  (tata && x<H[tata])
                    {H[fiu]=H[tata];
                    fiu=tata; tata=fiu/4;   }
             H[fiu]=x;}
         else                              //nodul este inserat pe un nivel maxim
         {tata=n/2;                            //nodul tata situat pe nivel minim
          if(x<H[tata])//compar cu nodurile minime de pe drumul spre radacina
             {while  (tata && x<H[tata])
                    {H[fiu]=H[tata];
                     fiu=tata; tata=fiu/4;   }
              H[fiu]=x; }
              else     //compar cu nodurile maxime de pe drumul spre radacina
              {tata=fiu/4;
               while  (tata && x>H[tata])
                      {H[fiu]=H[tata];
                       fiu=tata;  tata=fiu/4;   }
               H[fiu]=x; }
         }
      }
}
```

Observaţie

Complexitatea procedurii de inserare a unei valori într-un *min-max heap* este de `O(log n)`, valoarea noului nod fiind „promovată” în cel mai defavorabil caz până în rădăcina arborelui, un *min-max heap* fiind un arbore binar complet, înălţimea sa este de `O(log n)`.

Extragerea nodului minim dintr-un min-max-heap

Extragerea nodului minim revine la extragerea rădăcinii. Valoarea rădăcinii este înlocuită cu valoarea nodului aflat pe ultima poziţie în arbore, apoi aceasta este „retrogradată”, până la restabilirea structurii de *min-max-heap*.

Rădăcina se află întotdeauna pe un nivel minim, deci pentru a restaura *min-max heap*-ul, comparăm valoarea plasată în rădăcină cu valoarea celui mai mic dintre nepoţi, retrogradând succesiv această valoare până la restabilirea structurii de *min-max heap*. Există situaţia limită când nivelul pe care se află valoarea este ultimul nivel minim, caz în care valoarea trebuie comparată cu valorile nodurilor (dacă acestea există) de pe nivelul maxim următor şi eventual schimbate între ele.

De exemplu, să considerăm *min-max-heapul* din exemplul precedent. Valoarea rădăcinii va fi temporar înlocuită cu valoarea `15`. Valoarea `15` va fi schimbată cu `7`, valoarea celui mai mic dintre nepoţii săi. Valoarea `15` este acum plasată pe ultimul nivel minim, trebuie comparată cu valorile nodurilor de pe nivelul maxim următor. Astfel va fi schimbată cu valoarea `12`. Se obţine arborele din

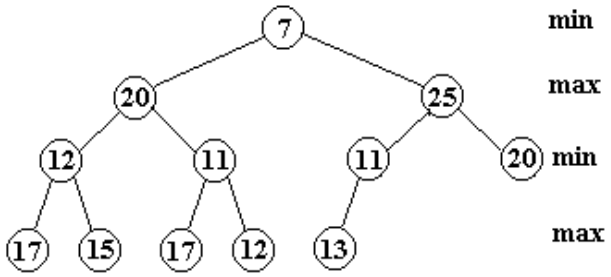


figura următoare:

Reprezentarea secvenţială a *min-max heap-ului* fiind:

1	2	3	4	5	6	7	8	9	10	11	12
7	20	25	12	11	11	20	17	15	17	12	13

Funcţia `ExtractMin(H,n)` extrage elementul minim din *min-max heap*-ul `H`, de dimensiune `n`:

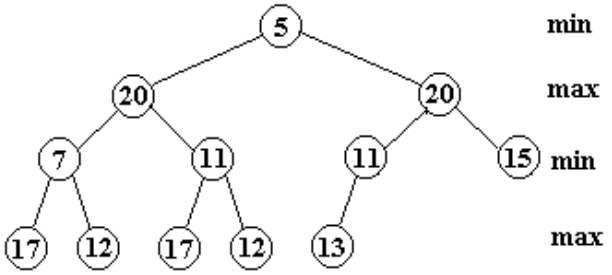
```
TipCheie ExtractMin(MinMaxHeap H, int& n)
{int poz=1,x;
 int gata=0;     //gata va fi 1 când terminam comparatiile pe niveluri minime
 TipCheie v=H[n--], rez=H[1];
 while (!gata && 4*poz <= n)
       {x= MinNepot(poz); //x este pozitia celui mai mic nepot al nodului poz
        if  (v<H[x])  gata =1;
            else
            {H[poz]=H[x]; poz=x; }
       }
H[poz]=v;
//compar cu nodurile de pe nivelul maxim urmator
x=MinFiu(poz);              //x este pozitia celui mai mic fiu al nodului poz
if (x <= n)                                          //exista cel putin un fiu
   if (v>H[x])
       {H[poz]=H[x]; H[x]=v;}
return rez; }
```

Extragerea elementului maxim

Elementul de valoare maximă este fiul rădăcinii cu valoarea cea mai mare. Deci, pentru a extrage maximul dintr-un *min-max heap*, se înlocuieşte cel mai mare dintre fiii rădăcinii cu valoarea ultimului nod, „retrogradând" succesiv această valoare până când structura de *min-max heap* este restabilită.

În situația limită când nu există nivel maxim, va fi extrasă valoarea nodului rădăcină.

De exemplu, să extragem elementul maxim din min-max-heapul din exemplul precedent. Acesta este nodul cu valoarea `25`. Valoarea ultimului nod va fi temporar plasată în nodul `3`. Nodul extras se află întotdeauna pe un nivel maxim. Vom compara succesiv noua valoare cu valoarea celui mai mare dintre nepoți, eventual retrogradând-o, până la restabilirea structurii de *min-max heap*, sau până când nu mai există nepoți. În cazul în care unul din fiii situați pe nivelul minim următor ar avea o valoare mai mare, le interschimbăm. Deci interschimbăm valoarea `15` cu valoarea `20`. Obținem arborele:



Reprezentarea secvențială a acestui arbore este:

1	2	3	4	5	6	7	8	9	10	11	12
5	20	20	7	11	11	15	17	12	17	12	13

Funcția `ExtractMax(H,n)` extrage valoarea maximă din *min-max heap*-ul `H`, de dimensiune `n`:

```
TipCheie ExtractMax(MinMaxHeap H, int& n)
{int poz=2, x, gata=0;
 if (n==1)                                          //exista un singur nod
     {n=0; return H[1];}
 TipCheie rez=H[2], v;
 if (n>2 && H[3]>H[2])
   {rez=H[3]; poz=3;}
  v=H[n--];                                    // restauram min-max heap-ul
  while (!gata && 4*poz <=n)
        {x=MaxNepot(poz);//x este pozitia celui mai mare nepot al nodului poz
         if  (v>H[x]) gata=1;
             else
             {H[poz]=H[x]; poz=x;}
        }
H[poz]=v;
//compar cu nodurile de pe nivelul minim  urmator
x=MaxFiu(poz);             //x este pozitia celui mai mare fiu al nodului poz
if (x <= n)                                         //exista cel putin un fiu
   if (v<H[x])
      {H[poz]=H[x]; H[x]=v;}
return rez;}
```

Observație

Complexitatea operațiilor `ExtractMin()` şi `ExtractMax()` este *logaritmică*, fiind urmată, în cazul cel mai defavorabil, de o eventuală actualizare a valorilor nodurilor de pe drumul de la nodul extras până la ultimul nivel.

Heap-uri binomiale

În secțiunile precedente am extins noțiunea de coadă cu prioritate, definind o nouă structura de date abstractă, coada cu dublă prioritate, care să permită extragerea eficientă atât a elementului minim, cât şi a elementului maxim.

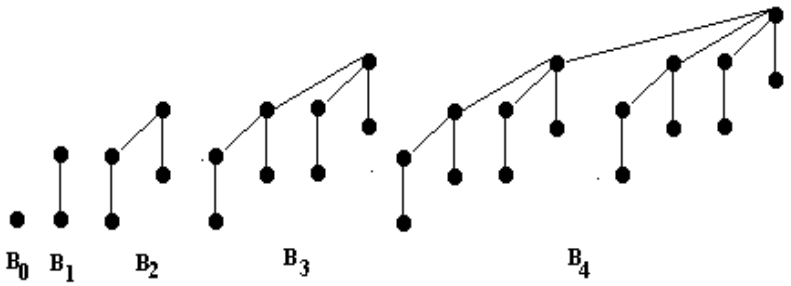
O altă operație utilă pe cozi cu prioritate ar fi operația `Union(`Q_1`,`Q_2`)` care să unifice cozile cu prioritate Q_1 şi Q_2. Se va crea o nouă coadă cu prioritate, care să conțină toate elementele din Q_1 şi Q_2, cozile Q_1 şi Q_2 fiind distruse.

Această operație nu poate fi implementată în mod eficient cu ajutorul *heap*-urilor, singura soluție acceptabilă fiind de a concatena *heap*-urile şi apoi de a aplica pentru restaurare procedura `CombHeap()`, care necesită un timp liniar.

O soluție în acest sens o constituie *heap*-urile binomiale.

Definiție
Un *arbore binomial* B_n este
- format dintr-un singur nod, dacă $n=0$;
- format din doi arbori binomiali B_{n-1}, astfel încât rădăcina unuia este cel mai din stânga fiu al rădăcinii celuilalt, dacă $n>0$.

De exemplu,



Lemă (Proprietăți ale arborilor binomiali)
Arborele binomial B_n are următoarele proprietăți:
- conține 2^n noduri;
- are înălțimea n;
- pe nivelul i, $\forall i \in \{0, 1, ..., n\}$, există C_n^i noduri;
- rădăcina are gradul n, mai mare decât al oricărui alt nod din arbore; mai mult, dacă numerotăm fiii rădăcinii de la stânga la dreapta cu $n-1$, $n-2$, ..., 0, fiul i este rădăcina subarborelui B_i.

Demonstrație
Vom face demonstrația prin inducție după n.

Pentru $n=0$, toate cele patru proprietăți sunt verificate în mod trivial. Presupunem proprietățile adevărate pentru $n-1$ şi demonstrăm proprietățile pentru n.

B_n constă din două copii ale arborelui binomial B_{n-1}, deci are $2^{n-1}+2^{n-1}=2^n$ noduri.

Din modul de construcție a arborelui B_n, deducem că înălțimea lui B_n este cu o unitate mai mare decât înălțimea arborelui B_{n-1}, deci înălțimea este $n-1+1=n$.

Fie $N(n,i)$ numărul de noduri de pe nivelul i din B_n. Cum B_n este format din doi arbori B_{n-1}, pe nivelul i, $\forall i \in \{2, 3, ..., n-1\}$, se găsesc nodurile de pe nivelul i dintr-un arbore B_{n-1} şi nodurile de pe nivelul $i-1$ din celălalt B_{n-1}.

$$N(n,i) = N(n-1,i) + N(n-1,i-1) = C_{n-1}^{i} + C_{n-1}^{i-1} = C_n^i$$

Pe nivelul 0 se găseşte un singur nod (rădăcina) $=C_n^0$, iar nivelul n este constituit din nivelul n−1 al unui arbore B_{n-1}, deci din $1=C_n^n$ noduri.

Singurul nod din B_n care are grad mai mare decât nodurile din B_{n-1} este rădăcina, care are un fiu mai mult decât rădăcina arborelui B_{n-1}. Cum din ipoteza inductivă rădăcina arborelui B_{n-1} are gradul n−1, deducem că gradul rădăcinii arborelui B_n este n. Tot din ipoteza inductivă, fiii rădăcinii arborelui B_{n-1} sunt, de la stânga la dreapta, respectiv, rădăcinile arborilor B_{n-2}, B_{n-3}, ..., B_0. Când B_{n-1} este plasat drept cel mai din stânga fiu al rădăcinii arborelui B_{n-1}, subarborii rădăcinii arborelui B_n sunt, de la stânga la dreapta, B_{n-1}, B_{n-2} ,..., B_0.

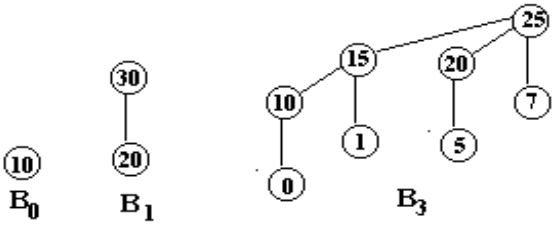
Observaţie
Denumirea arbore binomial provine de la faptul că numărul de noduri de pe nivelul i este C_n^i, valori denumite coeficienţi binomiali.

Definiţie
Un *heap binomial* (*B-heap*) este o pădure formată din arbori binomiali ce satisfac următoarele proprietăţi:
– fiecare arbore binomial este un *heap*;
– în *B-heap* nu există doi arbori binomiali pentru care rădăcinile să aibă acelaşi grad.

Observaţie
Dacă fiecare arbore binomial este un *max-heap*, *B-heap*-ul se va numi *max-heap* binomial şi în acest caz rădăcina fiecărui arbore binomial conţine elementul cu valoarea maximă. Dacă fiecare arbore binomial este un *min-heap*, *B-heap*-ul se va numi *min-heap* binomial şi în acest caz rădăcina fiecărui arbore binomial conţine elementul cu valoarea minimă. În continuare, prin *heap* vom înţelege un *max-heap*, cazul *min-heap* poate fi tratat în mod similar.

A doua condiţie din definiţia *B-heap*-ului asigură faptul că un *B-heap* cu n noduri poate conţine cel mult $[\log_2 n]+1$ arbori binomiali, deoarece reprezentarea binară a lui n are $[\log_2 n]+1$ biţi.

Conform proprietăţii 1. din lemă, arborele binomial B_i apare în *B-heap*-ul cu n noduri dacă şi numai dacă bit-ul b_i din descompunerea binară a lui n este 1. Deci, un *B-heap* cu n noduri conţine cel mult $[\log_2 n]+1$ arbori binomiali. De exemplu, în figură este reprezentat un *B-heap* cu n=11 noduri.

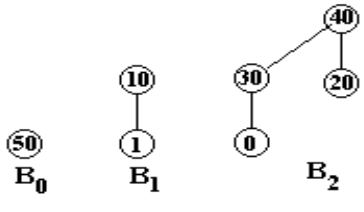


Reprezentarea heap-urilor binomiale

Vom transforma fiecare arbore binomial într-un arbore binar, utilizând reprezentarea fiu-frate, în plus reţinând pentru fiecare nod din arbore şi un pointer către părintele său.

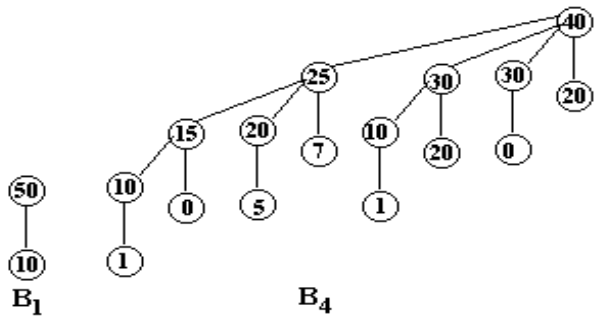
Vom reprezenta *B-heap*-ul reţinând poziţional într-un vector H de dimensiune $[\log_2 n]+1$ pointeri spre rădăcinile arborilor binomiali din *heap*-ul binomial, sau NULL dacă arborele binomial corespunzător poziţiei lipseşte din *B-heap*.

Unificarea heap-urilor binomiale



Pentru a unifica două *B-heap*-uri unificăm succesiv toţi arborii binomiali de acelaşi grad, până când pădurea obţinută conţine numai arbori binomiali având rădăcini de grade diferite. Spre exemplu să unificăm *heap*-ul din figura precedentă cu următorul *heap*:

Obţinem *B-heap*-ul:



Funcţia `Union(H1,H2)` unifică *B-heap*-urile `H1` şi `H2` şi întoarce *B-heap*-ul rezultat.

```
BHeap Union (BHeap H1, BHeap H2)
{
BHeap H;
ArboreBinomial T=NULL;                                  //arbore de "transport"
for (int i=1; i<=Lg; i++)
     if (H1[i] && H2[i])       //exista doi arbori binomiali de acelasi grad
         if (T)                // exista si un arbore binomial de "transport"
             {H[i]=T;
              T=UnionArbBin(H1[i], H2[i]);}
             else               //nu exista arbore de "transport"
             {H[i]=NULL;
              T=UnionArbBin(H1[i], H2[i]);}
        else                    //exista un singur arbore de grad i
        if (T)                  // exista un arbore binomial de "transport"
           {H[i]=NULL;
            if (H1[i]) T=UnionArbBin(T, H1[i]);
                else  T=UnionArbBin(T, H2[i]);}
           else                 //nu exista arbore de "transport"
           if (H1[i])H[i]=H1[i]
               else  H[i]=H2[i];
return H;}
```

Observaţie

`Lg` este o constantă globală care indică numărul maxim de arbori binomiali dintr-un *heap*. Am presupus că prin operaţia de unificare a două *B-heap*-uri numărul maxim de arbori dintr-un *B-heap* nu este depăşit. Testând valoarea variabilei `T` la sfârşitul funcţiei `Union()` putem da eventual un mesaj de eroare, în cazul în care în urma unificării `T≠NULL`, deci s-a obţinut prin transport un arbore mai mult decât numărul maxim de arbori din *B-heap*.

În funcţia `Union()` am apelat funcţia `UnionArbBin()`, care realizează unificarea a doi arbori binomiali de acelaşi grad:

```
ArboreBinomial UnionArbBin (ArboreBinomial T1, ArboreBinomial T2)
{if (T1->inf < T2->inf) //T1 va fi cel mai din stanga fiu al radacinii lui T2
     {T1->p=T2;      //radacina arborelui T2 va fi parintele radacinii lui T1
      T1->frate=T2->fiu;
               //primul fiu al radacinii lui T2 devine primul frate al lui T1
      T2->fiu=T1;                              //T1 devine primul fiu al lui T2
      return T2;}
                        //T2 va fi cel mai din stanga fiu al radacinii lui T1
T2->p=T1;             //radacina arborelui T1 va fi parintele radacinii lui T2
T2->frate=T1->fiu;
               //primul fiu al radacinii lui T1 devine primul frate al lui T2
T1->fiu=T2;                                      //T2 devine primul fiu al lui T1
```

```
return T1;}
```

Inserarea unei valori într-un heap binomial

Funcția `InsertBHeap(H,x)` inserează valoarea `x` în heapul binomial `H`. În acest scop se construieşte un *heap* binomial format numai din valoarea `x`, care apoi este unificat cu *heap*-ul `H`.

```
void InsertBHeap (BHeap H, TipCheie x)
{BHeap H1;
 ArboreBinomial T;
                        //creez arborele binomial format numai din valoarea x
 T=new Nod; T->inf=x;
 T->p=T->fiu=T->frate=NULL;
            // construiesc B-heap-ul H1, format numai din arborele binomial T
 H1[0]=T;
 for (int i=1; i<=Lg; i++) H1[i]=NULL;
 H=Union(H, H1);}
```

Extragerea nodului de valoare maximă dintr-un heap binomial

Funcția `ExtractMax(H)` extrage nodul de valoare maximă din *B-heap*-ul `H`. Pentru aceasta determinăm arborele binomial `T` care conține în rădăcină cea mai mare valoare şi îl eliminăm din *B-heap*-ul `H`. Extragem valoarea maximă din rădăcina arborelui binomial `T`, obținând astfel o pădure de arbori binomiali (subarborii rădăcinii lui `T`), deci un nou *B-heap* `H1`, pe care îl unificăm cu *B-heap*-ul `H`.

```
TipCheie ExtractMax (BHeap H)
{BHeap H1;
 TipCheie vmax= -Infinit;     //cea mai mica valoare posibila pentru TipCheie
 int pmax;            //determin arborele binomial ce contine valoarea maxima
 for (int i=0; i<=Lg; i++)
    if (H[i] && H[i]->inf > vmax)
       {vmax=H[i]->inf; pmax=i;}
ArboreBinomial T=H[pmax]; //arborele binomial ce contine valoarea maxima vmax
H[pmax]=NULL;                                          //extrag T din B-heap
            //construiesc B-heap-ul H1, format din subarborii radacinii lui T
for (i=0; i<=Lg; i++) H1[i]=NULL;
ArboreBinomial q=T->fiu;
for (i=pmax-1; i>=0; i--)             //consider subarborii in ordine inversa
    {H1[i]=q; q=q->frate;}
H=Union(H, H1);                          //unific B-heap-ul H cu B-heap-ul H1
return vmax;}
```

Complexitatea operațiilor pe heap-uri binomiale

Numărul maxim de arbori într-un *B-heap* cu `n` noduri este $[\log_2 n]+1$. Unificarea a doi arbori binomiali se realizează în timp constant, deci unificarea a două *B-heap*-uri se realizează în timp logaritmic.

Observație

Pentru claritate, în funcția de unificare a două *heap*-uri binomiale am parcurs complet vectorii ce reprezintă *B-heap*-urile, ceea ce face ca algoritmul să fie logaritmic în funcție de numărul maxim de noduri dintr-un *B-heap*. O modificare facilă, pe aceeaşi idee, poate conduce la un algoritm logaritmic în funcție de `max(n1,n2)`, numărul efectiv de noduri din *B-heap*-urile care se unifică.

Operațiile `Insert()` şi `ExtractMax()`, bazându-se pe operația de `Union()`, au de asemenea *complexitate logaritmică*.