HEAP-uri

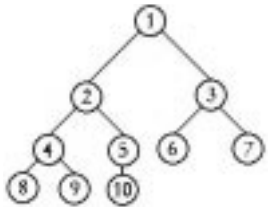
Mihai Scorțaru

În cadrul acestui articol vom prezenta una dintre cele mai interesante structuri de date care poate fi folosită pentru algoritmii de sortare și căutare. Aceasta se implementează relativ ușor și permite crearea de programe care se execută într-un timp mai scurt decât cele care folosesc alte metode pentru organizarea datelor.

Definiția heap-ului

Vom începe prin a defini structura de ***heap***. Denumirea este preluată din limba engleză și, în general, nu este folosită traducerea în limba română. O traducere aproximativă ar putea fi ***grămadă***; se mai folosește și termenul de ***ansamblu***.

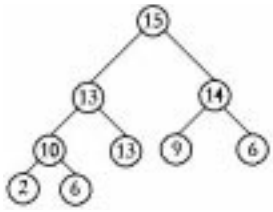
Un *heap* este un vector care poate fi considerat ca fiind un arbore binar după cum se poate vedea și în figura următoare:



Numărul care apare în fiecare nod indică poziția nodului respectiv în vector.

Așadar, rădăcina corespunde primului element al vectorului, fiul său stâng celui de-al doilea element, fiul drept celui de-al treilea și așa mai departe.

Considerăm vectorul `V={15,13,14,10,13,9,6,2,6}`. Arborele echivalent este următorul:



Pentru a obține vectorul corespunzător unui astfel de arbore, vom parcurge nivelurile arborelui începând cu rădăcina și fiecare nivel începând cu nodul cel mai din stânga.

Pentru ca un arbore binar să poată fi considerat un *heap*, toate nivelurile acestuia trebuie să fie complete, cu excepția ultimului care se completează începând cu nodul cel mai din stânga.

Înălțimea unui *heap* este înălțimea arborelui binar corespunzător. Înălțimea unui arbore este definită ca fiind egală cu adâncimea maximă a unui nod din arbore. Adâncimea unui nod este distanța dintre acel nod și rădăcina arborelui.

Un arbore complet de înălțime h are $1 + 2 + 4 + 8 + \ldots + 2^{h-1} = 2^h - 1$ noduri. Datorită faptului că ultimul nivel al unui *heap* poate să nu fie complet, numărul nodurilor unui *heap* este cuprins între 2^h și $2^{h+1}-1$. Reciproc, obținem că un *heap* cu n noduri are înălțimea $[\log_2 n]$.

Studiind organizarea arborelui, observăm că tatăl nodului corespunzător unei poziții $k > 1$ în vector este nodul corespunzător poziției $[k/2]$. Evident, tatăl rădăcinii nu există.

Reciproc, observăm că fiii nodului corespunzător poziției k sunt nodurile corespunzătoare pozițiilor $2 \cdot k$ (fiul stâng) și $2 \cdot k+1$ (fiul drept). În cazul în care valoarea $2 \cdot k$ este egală cu numărul de noduri, atunci fiul drept nu mai există, iar fiul stâng este ultimul nod al *heap*-ului. Dacă această valoare depășește numărul de noduri, atunci nici unul dintre fii nu mai există, nodul corespunzător poziției k fiind o frunză.

Cea mai importantă proprietate a *heap*-ului este aceea că valoarea oricărui nod nu poate fi mai mare decât valoarea nodului tată. Această proprietate face această structură foarte utilă pentru operațiile de căutare. Generalizând, obținem că valoarea corespunzătoare rădăcinii unui subarbore este mai mare sau egală cu oricare dintre valorile corespunzătoare oricărui nod din subarbore. De aici rezultă că rădăcina trebuie să conțină valoarea maximă a elementelor din vector.

Despre relațiile dintre valorile a două noduri, care nu sunt unul descendentul celuilalt, nu se poate face nici o afirmație. Cu alte cuvinte, faptul că un nod se află pe un nivel mai apropiat de rădăcină, nu înseamnă neapărat că el are o valoare mai mare decât nodurile aflate pe niveluri mai depărtate.

Această structură permite efectuarea foarte rapidă a anumitor operații:
- determinarea maximului valorilor dintr-un *heap*: $O(1)$;
- crearea unui *heap* dintr-un vector oarecare: $O(n)$;
- eliminarea unui element: $O(\log n)$;
- inserarea unui element: $O(\log n)$;
- sortarea unui vector: $O(n \cdot \log n)$;
- căutarea unei valori: $O(n)$.

Implementarea

Datorită echivalenței dintre un *heap* și un vector, datele nu vor fi reprezentate în memorie în formă arborescentă, ci în cea vectorială. Pentru implementare vom folosi limbajul C++. Vom considera *heap*-ul ca fiind un vector; tipul corespunzător va putea fi definit astfel:

```
typedef int Heap[MAX],
```

unde `MAX` reprezintă dimensiunea maximă a *heap*-ului. Deoarece dimensiunea *heap*-ului poate fi destul de mare, este recomandabil ca acesta să fie păstrat într-o variabilă globală și să nu fie transmis ca parametru pentru anumite funcții. Vom declara global un vector prin `Heap h;`.

Operațiile cu *heap*-uri trebuie efectuate astfel încât structura de *heap* să fie menținută în permanență, adică arborele să aibă toate nivelurile complete cu excepția ultimului, iar valoarea nici unui nod să nu depășească valoarea nodului părinte. Vom prezenta în continuare fiecare operație amintită.

Determinarea maximului

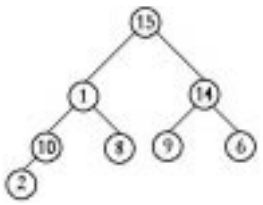
Datorită faptului că rădăcina corespunde primului element din vector și ea conține valoarea maximă, o funcție care determină valoarea maximă a elementelor dintr-un *heap*, va returna valoarea primului element al vectorului. Implementarea este următoarea:

```
int Max(){
  return h[1].
}
```

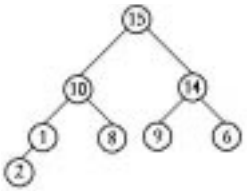
Din motive care vor fi explicate ulterior, poziția `0` nu este folosită.

Crearea unui heap dintr-un vector oarecare

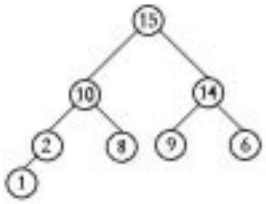
Pentu a prezenta această operație avem nevoie de o alta și anume reconstituirea structurii de *heap* atunci când avem un singur nod care nu respectă proprietatea de *heap* deoarece are o valoare mai mică decât unul dintre fiii săi. O asemenea situație este următoarea:



Nodul cu valoarea `1` nu se află pe poziția corectă deoarece valoarea sa este mai mică decât cel puțin una dintre valorile fiilor săi. În acest caz nodul va trebui coborât în arbore, fiind înlocuit cu unul dintre fiii săi. Dacă alegem nodul cu valoarea `8`, atunci structura de *heap* nu va fi corectă deoarece, după interschimbare, acest nod va avea ca fii un nod cu valoarea `10` și unul cu valoarea `1`, proprietatea de *heap* nefiind respectată. Dacă alegem nodul cu valoarea `10`, acesta va avea ca fii un nod cu valoarea `1` și unul cu valoarea `8`, proprietatea de *heap* fiind respectată în acest caz. Ca urmare, vom interschimba nodul având valoarea `1` cu nodul având valoarea `10`. Ajungem în următoarea situație:



Totuși, propritatea de *heap* nu este încă respectată, deoarece nodul cu valoarea `1` are un fiu cu valoarea `2`, deci cu valoare mai mare. De această dată avem un singur fiu, deci singura opțiune este de a interschimba cele două noduri. Ajungem în următoarea situație în care avem o structură de *heap* corectă:



În concluzie, algoritmul de reconstituire a proprietății de *heap* va coborî în arbore nodul care nu se află pe poziția corectă până în momentul în care nu mai avem fii sau proprietatea de *heap* este respectată. Pentru a coborî nodul în arbore îl vom interschimba cu fiul care are valoarea maximă. Această operație poartă denumirea de ***scufundare***.

Mai trebuie precizat faptul că, datorită relațiilor dintre pozițiile unui nod și cele ale fiilor săi, acestea pot fi determinate folosind doar operații pe biți (care sunt foarte rapide):
- Poziția fiului stâng al nodului de pe poziția k se obține prin deplasarea la stânga cu o poziție a valorii k: `k<<1`.
- Poziția fiului drept al nodului de pe poziția k se obține prin deplasarea la stânga cu o poziție a valorii k și setarea pe `1` a bitului celui mai nesemnificativ: `(k<<1)|1`.
- Poziția tatălui nodului de pe poziția k se obține prin deplasarea la dreapta cu o poziție a valorii k: `k>>1`.

Se observă acum motivul pentru care am ales să nu folosim poziția `0` a vectorului. Poziția fiului stâng al nodului de pe poziția `0` ar fi fost `0<<1=0`. Există artificii care permit și folosirea acestei poziții, dar nu insistăm asupra lor.

Vom implementa o funcție care realizează operația de scufundare. Aceasta va avea ca parametri dimensiunea *heap*-ului și poziția nodului care nu respectă proprietatea de *heap*.

```
void Scufunda(int n,int k){
  int aux,fiu=0;
```

```
  // se alege fiul cu valoarea mai mare
  if ((k<<1)<=n){
    fiu=k<<1;
    if ((k<<1)|1<n && h[(k<<1)|1]>H[k<<1])
      fiu|=1;
    // interschimbare numai daca este nevoie
    if (h[fiu]<=h[k]) fiu=0;
  }
  else fiu=0;
  while (fiu){
    // interschimbare
    aux=h[k];
    h[k]=h[fiu];
    h[fiu=k]=aux;
    // se alege urmatorul fiu
    if (k<<1<=n){
      fiu=k<<1;
      if ((k<<1)|1<n && h[(k<<1)|1]>H[k<<1])
        fiu|=1;
      // interschimbare numai daca este nevoie
      if (h[fiu]<=h[k]) fiu=0;
    }
    else fiu=0;
  }
}
```

Am preferat o variantă nerecursivă datorită vitezei mai mari de execuție. Acest algoritm funcționează numai dacă cei doi subarbori, care au ca rădăcină fiii nodului de pe poziția *k*, au o structură de *heap* corectă.

Vom prezenta acum modalitatea prin care această funcție de scufundare a unui element se folosește pentru a transforma un vector oarecare într-un *heap*.

Este evident faptul că frunzele arborelui au o structură de *heap* corectă. Ca urmare, vom putea apela funcția `Scufunda()` pentru nodurile imediat superioare frunzelor. După aceste apeluri, vom avea structuri de *heap* corecte pentru nodurile imediat superioare frunzelor. Vom trece apoi, pe rând, pe niveluri superioare până când vom ajunge la rădăcină.

În concluzie, va trebui să apelăm funcția `Scufunda()` începând cu nodul de pe poziția [*n*/2] și continuând cu nodurile de pe pozițiile anterioare până când vom ajunge la prima poziție. Vom scrie o funcție care are ca parametru dimensiunea *h* a *heap*-ului.

```
void ConstruiesteHeap(int n){
  for (int i=(n>>1);i;Scufunda(n,i--));
}
```

În cele ce urmează vom determina complexitatea acestui algoritm de construire a unui *heap* dintr-un vector oarecare. La prima vedere, avem *n* noduri, care pot fi scufundate cel mult [$\log_2 n$] niveluri deci, s-ar părea că avem un ordin de complexitate $O(n \cdot \log n)$. În realitate această determinare a complexității este eronată, deoarece nu ține cont de modul în care se fac scufundările. Frunzele reprezintă aproximativ jumătate din noduri și nu sunt scufundate deloc. Aproximativ un sfert dintre noduri se află imediat deasupra frunzelor și acestea sunt scufundate cel mult un nivel. O optime dintre noduri se află cu două niveluri deasupra frunzelor, deci sunt scufundate cel mult două niveluri. Continuând până la rădăcină, obținem un număr cel mult $\sum_{k=0}^{[\log_2 n]} \left(k \cdot \frac{n}{2^{k+1}} \right)$ scufundări. Dacă înălțimea arborelui este *h*, atunci *n* este, aproximativ, 2^h. Înlocuind în formulă obținem $\sum_{k=0}^{h} \left(k \cdot \frac{2^h}{2^{k+1}} \right)$; după efectuarea calculelor se obține ordinul de complexitate $O(2^h)$ pentru algoritmul de construire a unui *heap*.

Deoarece avem $h = [\log_2 n]$ vom obține ordinul de complexitate $O(2^{\log_2 n}) = O(n)$.

Eliminarea unui element

Pentru a efectua această operație avem nevoie, de asemenea, de o alta și anume reconstituirea structurii de *heap* atunci când avem un singur nod care nu respectă proprietatea de *heap*, deoarece are valoarea mai mare decât tatăl său. Este exact fenomenul invers celui prezentat în cazul operației de scufundare.

În acest caz nodul va trebui urcat în arbore. Vom presupune că restul *heap*-ului este corect.

Vom interschimba acest nod cu tatăl său. După interschimbare, nodul ajunge în poziția tatălui și subarborele având rădăcina în acest nod este un *heap* corect deoarece valoarea tatălui era mai mare decât cea a celuilalt fiu. Totuși, s-ar putea ca, după interschimbare, nodul să aibă un tată cu o valoare mai mică. În acest caz nodul va trebui din nou urcat în arbore. Procedeul va continua până când nodul curent devine rădăcina arborelui sau nu mai are un tată cu o valoare mai mică. Această operație poartă numele de *ridicare*.

Vom implementa o funcție care realizează operația de ridicare. Aceasta va avea ca parametri dimensiunea *heap*-ului *h* și poziția nodului care nu respectă proprietatea de *heap*.

```
void Ridica(int n,int k){
  int val=h[k];
  while (k>1 && val>h[k>>1])
    h[k]=h[k>>=1];
  h[k]=val;
}
```

Vom prezenta acum modul în care se folosește această funcție pentru a elimina un element dintr-un *heap*. După eliminarea unui element, în locul său apare un gol în care trebuie să amplasăm un alt element. Deoarece toate nivelurile (cu excepția ultimului) trebuie să fie complete, iar ultimul trebuie să fie ocupat în partea stângă, trebuie să eliminăm ultimul element din *heap*. Vom amplasa valoarea

corespunzătoare acestui element în locul gol și vom verifica dacă structura de *heap* este păstrată. Știm că singurul loc în care structura de *heap* s-ar putea să nu fie respectată este poziția elementului șters care a fost înlocuit cu ultimul element. Dacă valoarea este mai mare decât cea a tatălui, acest nod va trebui ridicat în arbore. Dacă nodul are o valoare mai mică decât cea a unui fiu, atunci nodul trebuie scufundat în arbore. În continuare vom prezenta o funcție care realizează eliminarea unui element dintr-un *heap*.

```
void Elimina(int& n,int k){
  h[k]=h[n--];
  if (k>1 && h[k]>h[k>>1]) Ridica(n,k);
  else Scufunda(n,k);
}
```

Deoarece o scufundare sau o ridicare se poate face cu cel mult h niveluri (h este înălțimea arborelui), iar h este $[\log_2 n]$, ordinul de complexitate al algoritmului de eliminare este $O(\log n)$.

După eliminare, dimensiunea *heap*-ului se reduce cu `1`, deci variabila `n` trebuie decrementată.

Inserarea unui element

Dacă dorim să inserăm un nou element într-un *heap*, operația este mult mai simplă. Trebuie doar să îl amplasăm pe poziția n+1 a vectorului și să îl ridicăm până când ajunge în poziția corectă. O funcție care realizează operația de inserare este prezentată în continuare.

```
void Insereaza(int& n,int val){
  h[++n]=val;
  Ridica(n,n);
}
```

Elementul poate fi ridicat cel mult h niveluri, deci ordinul de complexitate al algoritmului este $O(\log n)$. După eliminare dimensiunea *heap*-ului crește cu `1`, deci variabila `n` trebuie incrementată.

Sortarea unui vector

Algoritmul de sortare care folosește structura de *heap* poartă denumirea de ***heapsort***. Vom începe prin a construi un *heap* pentru vectorul pe care dorim să îl sortăm. După aceea eliminăm maximul și apoi refacem *heap*-ul. După refacere, al doilea element se află pe prima poziție a vectorului; îl eliminăm și refacem din nou *heap*-ul. Continuăm până când nu mai avem elemente în *heap*. Algoritmul de extragere a maximului are complexitatea $O(1)$, iar refacerea *heap*-ului necesită, așa cum vom vedea, o singură scufundare, deci are complexitatea $O(\log n)$. Având în vedere faptul că efectuăm n extrageri, complexitatea algoritmului *heapsort* este $O(n \cdot \log n)$.

Algoritmul poate fi implementat astfel încât să nu necesite spațiu suplimentar de memorie. După o eliminare a maximului, dimensiunea *heap*-ului se va reduce cu `1`, deci ultima poziție va rămâne liberă. Deoarece în vectorul sortat maximul trebuie amplasat exact pe această poziție, la fiecare pas nu trebuie decât să interschimbăm prima și ultima poziție a *heap*-ului și apoi să scufundăm elementul care a devenit rădăcina arborelui.

Algoritmului *heapsort* poate fi implementat prin funcția `HeapSort()` prezentată în continuare:

```
void HeapSort(int n){
  ConstruiesteHeap(n);
  for (int i=n;i>=2;){
    int aux=h[1];
    h[1]=h[i];
    h[i]=aux;
    Scufunda(--i,1);
  }
}
```

Căutarea unei valori

Pentru această operație folosirea *heap*-urilor nu este cea mai indicată soluție, deoarece complexitatea nu poate fi redusă sub $O(n)$. Aceasta se datorează faptului că, deși putem fi siguri că un nod cu valoarea mai mică este descendent al unui nod cu valoarea mai mare, nu știm în care dintre cei doi subarbori să îl căutăm. Totuși, față de căutarea secvențială se poate aduce o îmbunătățire; dacă rădăcina unui subarbore are o valoare mai mică decât valoarea căutată, atunci vom ști că descendenții rădăcinii vor avea valori mai mici, deci nu mai are rost să căutăm în acel subarbore. Astfel, s-ar putea ca porțiuni mari din *heap* să nu mai trebuiască explorate, dar în cazul cel mai defavorabil, este necesară parcurgerea întregului *heap*. Algoritmul poate fi implementat recursiv, destul de simplu, după cum se poate vedea în continuare.

```
int Cauta(int nod,int val){
  if (nod>n || val>h[nod])
    return 0;
  if (val==h[nod])
    return 1;
  return Cauta(nod>>1,val) ||
    Cauta((nod>>1)|1,val);
  }
```

Această funcție trebuie apelată cu valoarea `1` pentru parametrul `nod`.

Bibliografie

1. Thomas H. Cormen, Charles E. Leisserson, Ronald R. Rivest, *Introducere în algoritmi*, Editura *Computer Libris Agora*, Cluj - 2000
2. Cătălin Frâncu, *Psihologia concursurilor de informatică*, Editura *L&S*, București - 1997

Mihai Scorțaru este redactor-șef adjunct al GInfo. Poate fi contactat prin e-mail la adresa skortzy@xnet.ro.