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. Deci este suficient să reținem într-un vector informațiile asociate nodurilor, relațiile dintre noduri fiind descrise de formulele:

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

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

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

unde x este un nod oarecare în arbore, iar n numărul de noduri din arbore.

Utilizând reprezentarea secvențială, putem reformula definiția unui heap astfel:

Definiție

Tabloul H[1..n] formează un *max-heap* dacă

$H[i] \le H[i \text{ div } 2], \forall i \in \{2,3,...,n\}$.

Tabloul H[1..n] formează un *min-heap* dacă

$H[i] \ge H[i \text{ div } 2], \forall i \in \{2,3,...,n\}$.

De exemplu, arborele binar complet din figura 1 este un max-heap.

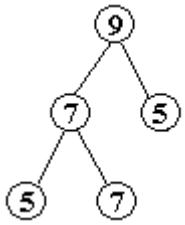


Fig. 1.

şi va fi reprezentat implicit :

1	2	3	4	5
9	7	5	5	7

4.1.1.Crearea unui heap[*]

I. O primă soluție este de a insera succesiv elementele în heap, plecând inițial de la heap-ul format dintr-un singur element:

```
procedure CreareHeap;
//organizează vectorul global H, de dimensiune n, ca un heap inserând
//succesiv elemente;
var i: integer;
begin
  for i := 2 to n do InsertHeap(i-1, H[i]);
end;
```

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 poziția n+1, apoi se

[*] Programul HeapSort de la sfârşitul capitolului implementează operațiile de creare a unui max-heap, inserarea unei valori, extragerea elementului maxim şi sortarea cu ajutorul heap-urilor.

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 figura 1:

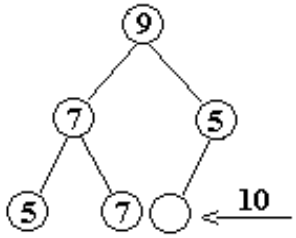


Fig. 2.

Valoarea inserată va ocupa poziția 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 :

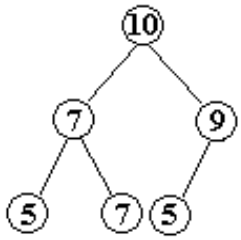


Fig. 3.

Modificările sunt aplicate de fapt vectorului ce constituie reprezentarea implicită :

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

```
procedure InsertHeap (n: integer; x: TipCheie);
//inserează valoarea x în heap-ul H de dimensiune n
var fiu, tata: integer;
begin
inc(n); fiu := n;//fiu indică poziția pe care trebuie plasată valoarea x
tata := n div 2;
while (tata > 0) and (H[tata] < x) do
        begin  // valoarea x trebuie "promovată"
          H[fiu] := H[tata];
          fiu := tata; tata := fiu div 2;
        end;
H[fiu] := x
end;
```

Observaţie

Vectorul H a fost considerat variabilă globală.

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:

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

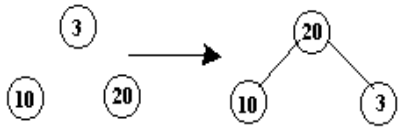
II. O strategie mai eficientă de construcţie a unui heap se bazează pe ideea de echilibrare. Apelul procedurii *InsertHeap(n, x)* poate fi interpretat ca o combinare a două heap-uri: un heap cu n elemente şi un heap format numai din elementul x. Putem construi heap-ul cu rădăcina H[i] combinând la fiecare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

```
procedure CreareHeap;
//organizează vectorul global H, de dimensiune n, ca un heap
var i:integer;
begin
for i:=n div 2 downto 1 do CombHeap(i,n);
end;

procedure CombHeap (i, n: integer);
//combină elementul H[i] cu heap-ul cu rădăcina 2i şi heap-ul cu
//rădăcina 2i+1
var fiu, tata: integer; v: TipCheie;
begin
v := H[i]; tata := i; //tata indică poziţia pe care va fi plasată valoarea v
fiu := 2*i; //fiul stâng
while fiu <= n do
   begin
      if fiu < n then //există fiu drept
          if H[fiu] < H[fiu+1] then fiu := fiu+1;
          //fiu indică fiul cu  valoarea cea mai mare
```

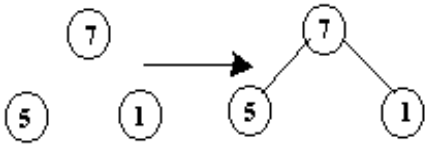
```
      if v < H[fiu] then
          begin
             H[tata] := H[fiu];
              // valoarea celui mai mare dintre fii urcă în arbore
             tata := fiu; //v coboară în arbore
              fiu := fiu*2;
          end
          else
          fiu := n+1; //am determinat poziția corectă pentru v
    end;
H[tata] := v;
end;
```

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):

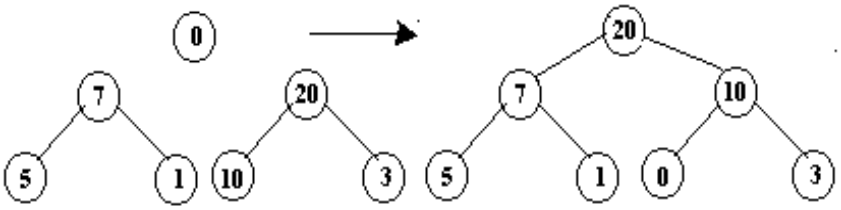


Deci H={0,7,20,5,1,10,3}.

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] înălțimea heap-ului. Pentru fiecare nod x de pe nivelul i, i∈{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 operații 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.

4.1.2. 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. Deci trebuie restaurat heap-ul, apelând 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.

```
procedure HeapSort;
//ordonează crescător vectorul global H, de dimensiune n, utilizând
//heap-uri
var i: integer;
begin
CreareHeap;
for i := n downto 2 do //plasează corect un element  pe poziția i
        begin
          H[1] ⇔ H[i]; // schimbă elementul H[i] cu H[1]
          CombHeap(1, i-1); //restaurează heap-ul
        end
end;
```

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

4.1.3. 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 prioritățile 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 figura 4 se extrage nodul cu valoarea 9,

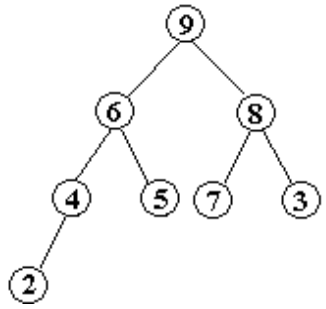


Fig. 4.

locul lui fiind ocupat de nodul cu valoarea 2.

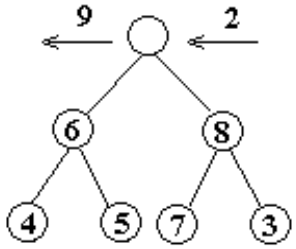


Fig. 5.

8 este maximul dintre succesorii săi. El îl înlocuieşte pe 2. Apoi 2 este înlocuit de 7. Rezultă heap-ul următor :

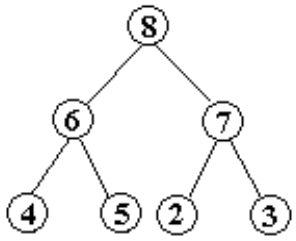


Fig. 6.

În reprezentare implicită, heap-ul

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

devine

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

function ExtractMax(var n: integer; var H: Heap): TipCheie;
//extrage elementul de prioritate maximă din heap-ul H de dimensiune n

```
begin
ExtractMax := H[1]; //extrage valoarea din rădăcină
H[1] := H[n];
dec(n);
CombHeap(1, n); //restaurează heap-ul
end;
```

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)*.

Resurse:
Software educațional Heap, de pe arhiva educațională .campion

Aplicație:
Dijkstra optimizat cu heap-uri.