Sortowanie
Kolejki priorytetowe i algorytm Heapsort

Dynamiczny problem sortowania:

podać strukturę danych dla elementów dynamicznego skończonego multi-zbioru S, względem którego są wykonywane następujące operacje:

(a) construct(q, S): utworzenie multi-zbioru
   S = {a_1,..a_n} dla danej listy q = [a_1,..,a_n];

(b) insert(x,S): S := S u {x};

(c) deletmax(S) - usunięcie z S największego elementu (dualną operacją jest deletemin(S): usunięcie z S najmniejszego elementu);

(*) zakładamy że elementy zbioru S pochodzą z pewnego uniwersum U, które jest liniowo uporządkowane.

Strukturę danych będącą rozwiązaniem dynamicznego problemu sortowania nazywamy **kolejką priorytetową**.

Elementarne implementacje **kolejki priorytetowej**:

1) **lista nieuporządkowana**, w której:
construct ma złożoność O(n),
insert - 0(1),
deletemax - O(n);
polecana wtedy, kiedy w ciągu wejściowym operacji jest dużo operacji insert, a mało deletemax;

2) **lista uporządkowana**, w której:
construct ma złożoność O(nlog n),
insert - O(n),
deletemax - 0(1);
polecana wtedy, kiedy w ciągu wejściowym operacji jest dużo operacji deletemax, a mało insert (na przykład O(log n)).

3) **kopiec**, dla którego
operacja construct ma złożoność czasową O(n),
insert i deletemax - O(logn).

Kopiec - drzewo binarne, w węzłach którego znajdują się elementy reprezentowanego multi-zbioru S i jest spełniony tzw. warunek kopca:

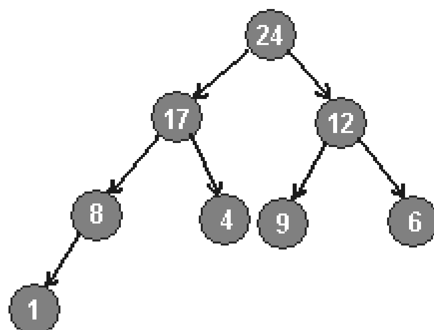
jeśli węzeł x jest następnikiem węzła y,
to key(x) <= key(y)

Elementy są wpisane do węzłów kopca zgodnie z porządkiem **kopcowym**, a drzewo ma uporządkowanie **kopcowe**.

Obserwacje:

(a) w korzeniu kopca znajduje się największy element (bądź jeden z największych elementów, jeśli jest ich kilka);

(b) na ścieżkach w kopcu, od korzenia do liścia, elementy są uporządkowane w porządku nierosnącym.



Algorytm **heapsort** - algorytm sortowania przez kopcowanie, można zapisać za pomocą operacji kolejki priorytetowej.

Ogólny schemat algorytmu sortowania za pomocą kolejki priorytetowej:

dana jest lista q = [a_1, ... ,a_n];

wykonaj:

(a) construct(q, S);

(b) powtórz n-1 razy deletemax(S).

Kopiec zupełny –

kopiec będący zupełnym drzewem binarnym, tzn. takim, w którym:

- wszystkie poziomy są wypełnione całkowicie,
- z wyjątkiem co najwyżej ostatniego - spójnie wypełnionego od strony lewej.

Zależność między wysokością **h** a liczbą węzłów **n** w kopcu zupełnym jest następująca:

$$2^h - 1 < n <= 2^{h+1} - 1$$

Czyli

$$2^h <= n < 2^{h+1}$$

Zatem:

$$h = \lfloor \log n \rfloor$$

Rozważmy numerację węzłów kopca zupełnego poziomami (od strony lewej do prawej).

Z definicji kopca zupełnego jest ona spójna.

Obliczmy numer poprzednika i następnika węzła o numerze k.

Przykład:

W kopcu, w którym każdy węzeł wewnętrzny ma dwa następniki: j = 2k.

Fakty:

- Następniki węzła k, gdy istnieją, mają numery2k i 2k+1.
- Poprzednik węzła k (różnego od korzenia) ma numer |_ k/2_|.

Operacja insert(x,S) - wstawienie elementu x do kopca zupełnego S, polega na:

umieszczeniu x w pierwszym wolnym miejscu ostatniego poziomu (lub następnego poziomu, gdy ostatni poziom jest całkowicie wypełniony) i

przywróceniu zachodzenia warunku kopca, jeśli wstawiony element jest większy niż element znajdujący się w poprzedniku.

Aby przywrócić zachodzenie warunku kopca, idziemy w górę w stronę korzenia, szukając miejsca, gdzie pasowałby wstawiany element.

Insert:
wstawienie elementu do liścia i dokonywanie później na ścieżce do korzenia zamian w celu przywrócenia zachodzenia warunku kopca - procedura pomocnicza upheap.

```
procedure upheap(k : in teger);
var l, v : integer;
begin
    v:= a[k]; a[0] := +∞;
    l :=k div 2;
    {warunek kopca jest zaburzony co najwyżej
     tylko dla v}
    while a[l] < v do
    begin {węzeł l jest poprzednikiem węzła k}
        a[k] := a[l];
        k := l; l := l div 2
    end;
    a[k] := v
end upheap;

procedure insert (v : integer);
begin
    n:=n+l;
    a[n] := v;
    upheap(n)
end insert;
```

T(n) = O(logn)

Operacja **deletemax(S)** - usunięcie z kopca największego elementu,

- usuwamy ten element z korzenia,
- usuwamy prawy skrajny liść z ostatniego poziomu,
- element znajdujący się w tym liściu wstawiamy do korzenia.
- Po tym wstawieniu warunek kopca może być zaburzony w jednym węźle, tym razem w korzeniu, wtedy:
- należy tak opuścić w dół element wstawiony do korzenia, żeby przywrócić zachodzenie warunku kopca.

```
procedure downheap(k : integer);
label L;
var i, i, v: integer;
begin
    v:= a[k];
    while k <= n div 2 do
    begin
        j := 2 * k; {j jest następnikiem k}
        if j < n then if a[j] < a[j + l] then j : = j + l ;
        if v >= a[j] then goto L;
        a[k] : = a[j];
        k := j
   end;
L: a[k]:= v
end downheap;
```

```
function deletemax : integer;
begin
    deletemax := a[1];
    a[1] := a[n];
    n := n - l;
    downheap(1)
end deletemax;
```

$T(n) = 2\lfloor \log n \rfloor = O(\log n)$

|

Operacja **construct**.

Realizowana za pomocą n operacji insert, to jej złożoność czasowa byłaby O(nlog n).

Można wykonać ją w czasie liniowym.

Trzeba konstrukcję kopca rozpocząć od dołu drzewa, tworzyć małe podkopce i łączyć je w większe - aż do powstania całego kopca.

Załóżmy, że kopce o korzeniach 2i i 2i + 1 zostały już skonstruowane.

Aby połączyć je i wstawić kolejny element x w węźle i, wystarczy wywołać downheap(i) (warunek kopca jest zaburzony tylko dla węzła i).

```
procedure construct;
{elementy listy q= [a_1, ... ,a_n] znajdują
  się w tablicy a[1 .. n]}
var i : integer;
begin
    for i : = n div 2 downto 1 do downheap (i)
end construct;
```

T(n) = O(n)

```
procedure heapsort;
{a[l .. n] - lista do posortowania}
var m,i : integer;
begin
   m := n;
   construct;
   for i : = m downto 2 do
      ali] := deletemax
   n := m
end heapsort
```

T(n) = 2nlog n + O(n)
A(n) - ?
S(n) = O(1)

Jeśli kopiec reprezentujemy w tablicy, to każdą operację z wyjątkiem Union można wykonać w czasie O(log n).

DRZEWA TURNIEJOWE – kolejny wykład