Wykład – Sortowanie II

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:

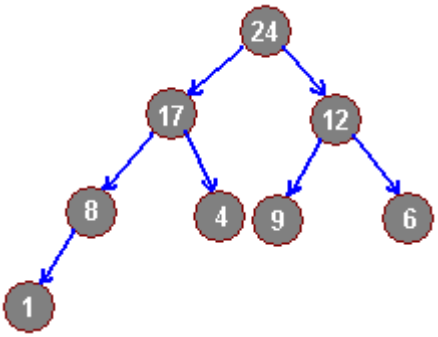
jeśli węzeł x jest następnikiem węzła y,
to key(x) <= key(y)

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 : integer);
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+1;
    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) = 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 pod kopce i łączyć je w większe - aż do powstania całego kopca.

Załóżmy, że kopce o korzeniach 2i i 2i + l 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) - ?

Sortowanie pozycyjne

Binarna reprezentacja danych w komputerze nie jest brana pod uwagę w ogólnych algorytmach sortowania.

Również w językach programowania wysokiego poziomu nie ma operacji na liczbach binarnych.

```
function bits(x, k, j : integer): integer,
begin
    bits := (x div 2^k) mod 2^j
end bits;
```

Funkcja bits wycina j bitów, poczynając od bitu o numerze k z prawej strony

Sortowanie pozycyjnego - metoda liczników częstości.

Załóżmy, że $b = 2^n$.

Dla każdego j = 0, l, ... , b - l liczymy, ile razy j pojawia się w ciągu wejściowym a[1], ... ,a[n].

Na podstawie obliczonych liczników częstości umieszczamy każdy element a, we właściwym miejscu w ciągu wynikowym.

```
procedure countsort,
{a[l. . n]. t[l. . n]. count[0 .. m -l]}
var i, j, p: integer;
begin
    for j := 0 to m-l do count[j] := 0; {inicjowanie}
    for i := 1 to n do count[a[i]] := count[a[i]] + 1;
   {count[j] to liczba wystąpień liczby j}
    for j := 1 to m-l do count[j] := count[j-1] + count[j];
{count[j] to liczba wystąpień elementów <= j}
for i := n downto l do
  begin
      p:= a[i];
      t[count[p]] := p;
      count[p] := count[p] - 1
  end;
for i := 1 to n do a[i] := t[i]
end countsort;
```

Przykład:

q = [1,3,1,2,2,1,3]

1sza iteracja
2ga iteracja count[1] = 3, count[2] = 2, count[3] = 2,
3cia iteracja count[1] = 3, count[2] =5, count[3] = 7,

1szy element zajmie pozycje a[1..count[1]]
2gi element zajmie pozycje a[count[1]+1,..count[2]]
3ci element zajmie pozycje a[count[2]+1,..count[3]]

Wypisywanie elementów do tablicy

- - - - - - 3
- - 1- - - 3
- - 1- 2- 3
- - 122 -3
- 1122 -3
- 112233
1112233

Analiza -algorytmu countsort jest bezpośrednia:

$T(n, m) = A(n, m) = O(n + m)$

$S(n, m) = n + m + 0(1)$

Algorytm jest szybki, gdy $m = O(n)$

Algorytm jest stabilny.

countsort można stosować tylko dla niedużych wartości m (tablica count ma rozmiar m !).

Dla większych wartości m można stosować podobną metodę, ale z dzieleniem liczb do posortowania na części, na których algorytm countsort może działać.

Dzielimy słowa o b bitach na b/e grup e bitowych.

1) sortujemy ciąg liczb, stosując algorytm countsort względem ostatniej (najmniej znaczącej) grupy bitów.

2) sortujemy ciąg liczb względem przedostatniej grupy bitów, itd. …

b/e) sortujemy ciąg liczb względem pierwszej grupy bitów.

Istotna jest stabilność algorytmu countsort.

O pozycji elementu decyduje pierwszych e bitów;
jeśli są one takie same, to decydujące znaczenie mają dopiero następne grupy bitów, które zostały też posortowane.

```
procedure radixsort;
 {m=2^e, a, t[l .. n],count[0 .. m-l]}
var i, j, pass, nofpasses, key: integer;
begin
    nofpasses := b div e;
    if nofpasses * e := b then nofpasses := nofpasses - 1;
    for pass: = 0 to nofpasses do
    begin
        for j :=0 to m-1 do count[j] :=0;
        for i : = l to n do
        begin
            key := bits(a[i], pass*e, e);
            count[key] := count[key] + 1
       end;
       for j:=1 to m-l do count[j]:= coun t[j-1] + count[j];
       for i := n downto 1 do
       begin
           key:= bits(a[i], pass * e, e);
           t[count[key]] := a[i];
          count[key] := count[key]-l
      end;
      for i := 1 to n do a[i] := t[i]
   end
end radixsort;
```

T(n,m) = A(n,m) = O(n+m), gdy b/e = O(1)
S(n,m) = n+m+O(1)

Przykład:

b = 8, e = 2, n = 4

a[l] = 011100 10
a[2] = 110111 01
a[3] = 100000 00
a[4] = 011010 01

pass = 0

a[l] = 1000 00 00
a[2] = 1101 11 01
a[3] = 0110 10 01
a[4] = 0111 00 10

pass = 1

a[l] = 1000 00 00
a[2] = 0111 00 10
a[3] = 0110 10 01
a[4] = 1101 11 01

pass = 2

a[l] = 10 00 00 00
a[2] = 11 01 11 01
a[3] = 01 10 10 01
a[4] = 01 11 00 10

pass = 3

a[l] = 01 10 10 01
a[2] = 01 11 00 10
a[3] = 10 00 00 00
a[4] = 11 01 11 01

Zalety Radixsort:

Liniowa złożność czasowa przy odpowiednich założeniach dotyczących n.

Dla średnich wartości ‘n’ może być szybszy niż quicksort.

Po modyfikacjach może być użyty do sortowania:

- słów w porządku alfabetycznym
- liczb rzeczywistych z pewnego przedziału

Wady:

- Algorytm jest wolny dla małych rozmiarów ‘n’,
- Duże zużycie pamięci dla dużych ‘n’.