ASD—3

1 Kolejki priorytetowe i algorytm HeapSort

Przez dynamiczny problem sortowania rozumiemy następujące zadanie algorytmiczne. Podać strukturę danych dla elementów dynamicznego, skończonego wielozbioru S, na którym wykonywane są następujące operacje:

1. `construct(q,S)`:: mając dany ciąg $q = [a_1, \ldots, a_n]$ utwórz zbiór $S = \{a_1, \ldots, a_n\}$;
2. `insert(x,S)`:: $S = S \cup \{x\}$;
3. `deletemax(S)`:: usuń z S największy element (dualną operacją jest `deletemin(S)`).

Zakłada się jak poprzednio, że elementy zbioru S brane są z pewnego uniwersum U, które jest liniowo uporządkowane.

Strukturę danych będącą rozwiązaniem dynamicznego problemu sortowania nazywamy kolejką priorytetową.

Do elementarnych implementacji kolejki priorytetowej należą:

1. tablica nieuporządkowana – w której `construct` ma złożoność $O(n)$, `insert` – $O(1)$, `deletemax` – $O(n)$. Jest ona godna polecenia, gdy w wejściowym ciągu operacji jest duża liczba operacji `insert`, a mała `deletemax`.
2. tablica uporządkowana – w której `construct` ma złożoność $O(n \log n)$, `insert` – $O(n)$, `deletemax` – $O(1)$. Jest ona godna polecenia, gdy w wejściowym

ciągu operacji jest duża liczba operacji `deletemax`, a mała `insert` (np. $O(\log n)$).

Podstawową zaletą elementarnych implementacji kolejki priorytetowej jest ich prostota i niski współczynnik proporcjonalności w funkcjach złożoności.

W tym punkcie rozważymy strukturę danych (kopiec), w której operacja `construct` ma złożoność czasową $O(n)$, a `insert` i `deletemax` – $O(\log n)$.

Algorytm HeapSort, czyli algorytm sortowania przez kopcowanie, daje się zapisać za pomocą operacji kolejki priorytetowej.

Ogólny schemat algorytmu sortowania za pomocą kolejki priorytetowej:

Dany jest ciąg $q = [a_1, \ldots, a_n]$. Wykonaj:
(1) `construct(q,S)`
(2) powtórz n razy `deletemax(S)`.

Używając implementacji kopcowej otrzymamy algorytm sortowania HeapSort o pesymistycznej złożoności czasowej $O(n \log n)$.

Przystąpimy teraz do zdefiniowania struktury danych kopca.

Przez *kopiec* rozumiemy drzewo binarne, w którego węzłach znajdują się elementy reprezentowanego zbioru S i spełniony jest warunek kopca:
— dla każdych dwóch węzłów x i y, jeśli x jest następnikiem y, to element w węźle x jest nie większy od elementu w węźle y.

Zauważmy, że:

- element największy zbioru znajduje się w korzeniu drzewa,
- na scieżkach w drzewie, od korzenia do liści, elementy są uporzadkowane w kolejności nierosnącej.

Przez *kopiec zupełny* rozumiemy kopiec będący zupełnym drzewem binarnym tj. takim, w którym wszystkie poziomy wypełnione są całkowicie, z wyjątkiem co najwyżej ostatniego, który jest spójnie wypełniony od strony lewej.

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

$$2^h - 1 < n \leq 2^{h+1} - 1$$

czyli

$$2^h \leq n < 2^{h+1}.$$

Stąd

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

Kopiec zupełny ma regularny kształt, który umożliwia reprezentowanie go w tablicy w "ukryty" sposób:

- Następniki węzła k (o ile istnieją) mają odpowiednio numery $2k$ i $2k+1$.
- Poprzednik węzła k (różnego od korzenia) ma numer $\lfloor k/2 \rfloor$.

Operacja `insert(x,S)` wstawienia elementu x do kopca zupełnego S polega na umieszczeniu x w pierwszym wolnym miejscu ostatniego poziomu, po czym przywróceniu zachodzenia warunku kopca, jeśli wstawiony element jest większy od elementu znajdującego się w poprzedniku. Aby przywrócić zachodzenie warunku kopca idziemy w góre, w stronę korzenia, szukając miejsca, gdzie pasowałby wstawiany element. Operacja `insert` składa się zatem ze wstawienia elementu do liścia, a następnie z zamian dokonywanych na ścieżce do korzenia w celu przywrócenia zachodzenia warunku kopca. Użyjemy pomocniczej procedury `upheap`, której celem jest przywrócenie warunku kopca po zwiększeniu wartości elementu na pozycji k w kopcu–tablicy a reprezentującej S.

```
void upheap(long *a, int k)
{
  int l;
  long v;

  v = a[k];
  l = k/2;
  while (( l != 0) && ( a[l] < v)){

    a[k] = a[l];
    k = l;
    l /= 2;
  }
  a[k] = v;
}

void insert(long x, long *a, int *n)
{/* n -- liczba elementow w tablicy
```

```
    zakladamy, ze jest miejsce w tablicy a */
   (*n)++;
   a[*n] = x;
   upheap(a,*n);
}
```

Ponieważ wysokość kopca zupełnego wynosi $h = \lfloor \log n \rfloor$, więc pesymistyczna złożoność czasowa `insert`, mierzona liczbą porównań, wynosi $W_{\mathrm{insert}}(n) = \lfloor \log n \rfloor + 1$, gdzie n oznacza liczbę elementów w kopcu przed wstawieniem v.

Aby wykonać operację `deletemax(S)` usuwania z kopca elementu największego, usuwamy ten element z korzenia, po czym z ostatniego poziomu bierzemy prawy skrajny liść, usuwamy ten liść z drzewa, natomiast element znajdujący się w tym liściu wstawiamy do korzenia. Po tym warunek kopca może być zaburzony w jednym węźle, tym razem w korzeniu. Opuszczamy element wstawiony do korzenia w dół tak, aby przywrócić zachodzenie warunku kopca.

Użyjemy pomocniczej procedury `downheap`, której celem jest przywrócenie warunku kopca po zmniejszeniu wartości elementu na pozycji k.

```
void downheap(long *a, int k, int n)
{
  int l;
  long v;

  v = a[k]; l = 2*k;
  while (l <= n){
```

```
    if ((l < n) && (a[l] < a[l+1]))
        l = l + 1;
    if (v < a[l]){
        a[k] = a[l];
        k = l;
        l = 2*l;
    }
    else
      l = n + 1;
  }

  a[k] = v;
}
```

Zauważmy, że na każdym poziomie drzewa (z wyjątkiem zerowego) algorytm `downheap` może wykonać dwa porównania.

```
long deletemax(long *a, int *n)
{/* n -- liczba elementow w tablicy, n > 0 */
  long v;

  v = a[1];
  if (*n > 1){
    a[1] = a[*n];
    (*n)--;
    downheap(a,1,*n);
  }
  else *n = 0;

  return v;
}
```

Pesymistyczna złożoność czasowa operacji `deletemax` mierzona liczbą porównań wynosi $W_{\text{deletemax}}(n) = 2\lfloor \log n \rfloor$, gdzie n oznacza liczbę elementów w kopcu po usunięciu elementu największego.

Pozostała jeszcze operacja `construct`. Gdybyśmy realizowali ją za pomocą n operacji `insert`, jej złożoność czasowa byłaby $O(n \log n)$. Daje się ona jednak wykonać w czasie liniowym, jeśli konstrukcję kopca rozpoczniemy od dołu drzewa, konstruując małe podkopce i łącząc je w większe, aż otrzymamy cały kopiec. Zakładając, że podkopce o korzeniach $2i$ i $2i+1$ zostały już skonstruowane, aby połączyć te podkopce i włączyć kolejny element x w węźle i, wystarczy wywołać `downheap(i)` (warunek kopca jest zaburzony tylko dla węzła i).

```
void construct(long *a, int n)
{
  int i;

  for ( i = n/2; i > 0; i-- )
    downheap(a,i,n);
}
```

W końcu uzyskujemy sam algorytm sortowania przez kopcowanie:

```
void HeapSort(long *a, int n)
{
  int v;
  int i;
```

```
  construct(a,n);
  for (i = n; i > 1; )
  {
    v = i;
    a[v] = deletemax(a,&i);
  }
}
```

Wykonanie $n-1$ razy `deletemax` kosztuje $O(n \log n)$. Analizując dokładnie otrzymujemy, że pesymistyczna złożoność algorytmu HeapSort wynosi $W(n) \leq 2n \log n + O(n)$.