

KOPIEC

1 Definicja

Definicja 1 Niech T będzie drzewem binarnym o wysokości d , którego wierzchołki zawierają klucze z liniowo uporządkowanego zbioru. Drzewo T nazywamy kopcem *iff* T spełnia następujące warunki:

(1) *Struktura drzewa*

- wszystkie jego liście znajdują się na głębokości d lub $d - 1$;
- wszystkie liście z poziomu $d - 1$ leżą na prawo od wszystkich wierzchołków wewnętrznych z tego poziomu;
- położony najbardziej na prawo wierzchołek wewnętrzny z poziomu $d - 1$ jest jedynym wierzchołkiem wewnętrznym w T , który może mieć jednego syna (co implikuje, że pozostałe wierzchołki wewnętrzne mają po dwóch synów);

(2) *Uporządkowanie*

- klucz w każdym wierzchołku wewnętrznym jest nie mniejszy od kluczy w jego potomkach.

Warunki określające strukturę kopca mogą się wydać nieco skomplikowane. W rzeczywistości mówią one, że dobrą strukturę mają drzewa binarne powstałe przez dopisywanie do początkowo pustego drzewa wierzchołków do kolejnych poziomów drzewa, zapełniając każdy poziom od lewej strony do prawej strony.

2 Implementacja kopców

Kopce w bardzo efektywny sposób mogą być pamiętane w tablicach. Do pamiętania kopca n -elementowego używamy n -elementowej tablicy K :

- korzeń kopca pamiętany jest w $K[1]$,
- lewy syn korzenia pamiętany jest w $K[2]$, prawy syn korzenia - w $K[3]$, itd

Uogólniając: wierzchołki z poziomu k -tego pamiętane są kolejno od lewej do prawej w $K[2^k], K[2^k + 1], \dots, K[2^{k+1} - 1]$.

Fakt 1 Ojciec wierzchołka pamiętanego w $K[i]$ znajduje się w $K[i \text{ div } 2]$ zaś jego dzieci (o ile istnieją) w $K[2i]$ i $K[2i + 1]$.

2.1 Ważniejsze procedury

2.1.1 Procedury przywracające tablicy K własności kopca

Zmiana klucza w wierzchołku kopca może spowodować zaburzenie własności (2). Jeśli nowy klucz jest większy od starego, należy sprawdzić, czy nie jest on większy także od klucza znajdującego się w ojcu. Jeśli tak jest, możemy zamienić miejscami te klucze i sprawdzić czy zaburzenie nie przeniosło się poziom wyżej. Jeśli nowy klucz jest mniejszy od starego, to możemy zamienić go z większym z kluczy znajdujących w jego synach, a następnie sprawdzić czy zaburzenie nie przeniosło się na niższy poziom.

```

procedure zmień-element ( $K[1..n], i, u$ )
   $x \leftarrow K[i]$ 
   $K[i] \leftarrow u$ 
  if  $u < x$  then przesuń-niżej ( $K, i$ )
  else przesuń-wyżej ( $K, i$ )

procedure przesuń-niżej ( $K[1..n], i$ )
   $k \leftarrow i$ 
  repeat
     $j \leftarrow k$ 
    if  $2j \leq n$  and  $K[2j] > K[k]$  then  $k \leftarrow 2j$ 
    if  $2j < n$  and  $K[2j+1] > K[k]$  then  $k \leftarrow 2j+1$ 
     $K[j] \leftrightarrow K[k]$ 
  until  $j = k$ 

procedure przesuń-wyżej ( $K[1..n], i$ )
   $k \leftarrow i$ 
  repeat
     $j \leftarrow k$ 
    if  $j > 1$  and  $K[j \div 2] < K[k]$  then  $k \leftarrow j \div 2$ 
     $K[j] \leftrightarrow K[k]$ 
  until  $j = k$ 

```

2.1.2 Procedura budująca kopiec

Kopiec można budować na wiele sposobów. Można np. zacząć od tablicy jednoelementowej, a następnie dodawać na jej koniec po jednym elemencie, za każdym razem używając procedury *przesuń-wyżej* do przywrócenia własności (2). Ten sposób realizuje poniższa procedura.

```

procedure wolna – budowa – kopca ( $K[1..n]$ )
  for  $i \leftarrow 2$  to  $n$  przesuń-wyżej ( $K, i$ )

```

łatwo sprawdzić, że ta procedura może wymagać czasu $n \log n$, a więc takiego samego jak sortowanie np. metodą *quicksort*, dając jednak znacznie mniej uporządkowaną strukturę.

Inna metoda polega na budowaniu kopca od dołu. Startujemy od kopców 1-elementowych. Następnie używamy tych kopców oraz nowych elementów do utworzenia kopców 3-elementowych: nowy element umieszczamy w korzeniu takiego kopca, a jego synami czynimy korzenie kopców 1-elementowych; następnie używamy procedury *przesuń-niżej* do przywrócenia własności (2). W analogiczny sposób, dla dowolnego k , tworzymy z dwóch kopców $(2^k - 1)$ -elementowych i jednego nowego elementu kopiec $(2^{k+1} - 1)$ -elementowy.

```

procedure buduj – kopiec ( $K[1..n]$ )
  for  $i \leftarrow (n \div 2)$  downto 1 przesuń-niżej ( $K, i$ )

```

Twierdzenie 1 *Procedura buduj – kopiec tworzy kopiec w czasie $O(n)$.*

3 Zastosowania kopców

3.1 HEAPSORT - sortowanie przy użyciu kopca

```
procedure heapsort( $K[1..n]$ )  
  buduj  $\leftarrow$  kopiec( $K$ )  
  for  $i \leftarrow n$  step  $-1$  to  $2$  do  
     $K[1] \leftrightarrow K[i]$   
    przesunij-niżej ( $K[1..i-1], 1$ )  
  return  $K$ 
```

Twierdzenie 2 Algorytm *heapsort* działa w czasie $O(n \log n)$.

3.1.1 Przyspieszenie *heapsortu*

Po usunięciu maksimum na szczycie kopca powstaje dziura, w którą *heapsort* wstawia element z dołu kopca. Element taki jest, z dużym prawdopodobieństwem, mały i zostanie przez procedurę *przesunij-niżej* zsunięty z powrotem nisko. Przesuwając go o jeden poziom w dół *przesunij-niżej* wykonuje dwa porównania. Tak więc z dużym prawdopodobieństwem potrzeba będzie 2-wysokość kopca porównań na przywrócenie własności kopca.

Można postępować nieco oszczędniej. Otóż można najpierw przesunąć dziurę na dół kopca, następnie wstawić w nią ostatni element kopca i używając procedury *przesunij-wyżej* znaleźć dla niego odpowiednie miejsce w kopcu. Oszczędność wynika z tego, że na przesunięcie dziury o jedno miejsce w dół potrzeba tylko jednego porównania oraz z tego, że w średnim przypadku *przesunij-wyżej* będzie przesunąć element o nie więcej niż 2 poziomy w górę.

3.2 Kolejka priorytetowa

Kolejka priorytetowa jest strukturą danych przeznaczoną do pamiętania zbioru S (elementów z jakiegoś uporządkowanego uniwersum) i wykonywania operacji wstawiania elementów do S oraz znajdowania i usuwania największego elementu z S .

Wprost idealnie do implementacji kolejek priorytetowych nadają się kopce.

3.2.1 Procedury realizujące operacje kolejki priorytetowej

```
function find-max( $K[1..n]$ )  
  return  $K[1]$   
  
procedure delete-max( $K[1..n]$ )  
   $K[1] \leftarrow K[n]$   
  przesunij-niżej ( $K[1..n-1], 1$ )  
  
procedure insert-node( $K[1..n], v$ )  
   $K[n+1] \leftarrow v$   
  przesunij-wyżej ( $K[1..n+1], n+1$ )
```

3.3 Podwójna kolejka priorytetowa

Podwójna kolejka priorytetowa umożliwia wykonywanie operacji znajdowania i usuwania zarówno maksymalnego jak i minimalnego elementu.

Prosta implementacja takiej kolejki mogłaby polegać na wykorzystaniu dwóch kopców: jeden z nich byłby uporządkowany malejąco, a drugi - rosnąco. Każdy element kolejki umieszczany byłby w obydwu kopcach. Na użytek operacji *deletemin* i *deletemax* należałoby powiązać ze sobą elementy kopców pamiętające ten sam element kolejki. Zasadniczym mankamentem takiego rozwiązania jest nieoszczędność pamięci. Poniżej przedstawiamy rozwiązanie wolne od tej wady.

Idea rozwiązania:

1. Wykorzystujemy dwa kopce: L i H .
2. W kopcu H pamiętamy $\lceil n/2 \rceil$ elementów, a w kopcu L - $\lfloor n/2 \rfloor$ elementów (n oznacza liczbę elementów kolejki).
3. Kopiec L uporządkowany jest malejąco, a H - rosnąco.
4. Na uporządkowanie z poprzedniego punktu nakładamy dodatkowy warunek. Otóż w kopcach H i L w naturalny sposób zdefiniowane są ścieżki biegnące od korzenia kopca L do korzenia kopca H (patrz Rysunek 1). Chcemy, by na każdej takiej ścieżce klucze były uporządkowane niemalejąco.

Rysunek 1: Dwie przykładowe ścieżki łączące korzenie kopców.

3.3.1 Implementacja operacji *insert*(x)

Jeśli kolejka zawiera parzystą liczbę elementów, x wstawiamy do kopca H . W przeciwnym razie x wstawiamy do kopca L . Wstawienia dokonujemy w zwykły sposób, tj. wstawiając element do pierwszego wolnego liścia. Teraz jednak, zanim użyjemy procedury *przesuń-wyżej*, musimy sprawdzić, w którą stronę należy przesunąć wstawiony element: czy w stronę korzenia kopca H , czy też w stronę korzenia kopca L . Załóżmy, że x został wstawiony do H (przypadek wstawienia do L jest symetryczny). Niech y będzie poprzednikiem x -a ze ścieżki prowadzącej do korzenia L (oczywiście y jest liściem kopca L). Porównujemy x i y . Jeśli $x < y$, przestawiamy te elementy a następnie procedurą *przesuń-wyżej* przesuwamy x w odpowiednie miejsce w kierunku korzenia L . Zauważmy, że przesunięcie y -ka z kopca L do H nie zaburzyło porządku w tym kopcu. Jeśli $x \geq y$, używamy procedury *przesuń-wyżej* do ewentualnego naprawienia porządku w kopcu H .

3.3.2 Implementacja operacji *deletemin*

Usuujemy element z korzenia kopca L . W jego miejsce wstawiamy y - ostatni element kopca L (jeśli przed operacją kopce L i H były równoliczne) albo ostatni element kopca H (jeśli przed operacją L

zawierał o jeden element mniej niż H). Następnie procedurą *przesuń-niżej* przywracamy porządek w kopcu L . Jeśli procedura *przesuń-niżej* przesunęła y na sam dół kopca L , wówczas porównujemy y z odpowiednim (znajdującym się na tej samej pozycji) elementem kopca H . Jeśli y jest od niego większy, zamieniamy te elementy miejscami, a następnie procedurą *przesuń-wyżej* przesuwamy y na odpowiednie miejsce kopca H .

3.3.3 Implementacja operacji *deletemax*

Analogicznie do operacji *deletemin*.

3.3.4 Uwagi końcowe

W literaturze można znaleźć wiele implementacji kolejek podwójnych opartych na strukturze kopców. Implementacja podana przez nas oparta jest na symetrycznych kopcach minimaksowych przedstawionych w [1]. Inne metody można znaleźć np. w [2], [3] i [4].

Literatura

- [1] A. Arvind, C. Pandu Rangan, Symmetric Min-Max heap: A simpler data structure for double-ended priority queue, *Inform. Process. Lett.*, 69(1999), 197-199.
- [2] M.D. Atkinson, J.-R. Sack, T. Strothotte, Min-Max heaps and generalized priority queues, *Comm. ACM*, 29(1986), 996-1000.
- [3] S. Carlsson, The Deap - a double-ended heap to implement double-ended priority queues, *Inform. Process. Lett.*, 26(1987), 33-36.
- [4] S.C. Chang, M.W. Du, Diamond deque: A simple data structure for priority dequeues, *Inform. Process. Lett.*, 46(1993), 231-237.