

Algorytmy i Struktury Danych, 5. ćwiczenia, rozwiązania z serii 4

2023-11-06

Zadanie 4.1

W tym zadaniu analizujemy algorytm Quick Sort, w którym za element dzielący wybiera się medianę z $(a[l], a[(l+r)/2+1], a[r])$ (zobacz wykład 3). Rozważmy permutację:

$$K_{2n} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & \dots & k-2 & k-1 & k & k+1 & k+2 & k+3 & \dots & 2k-1 & 2k \\ 1 & k+1 & 3 & k+3 & 5 & \dots & 2k-3 & k-1 & 2k-1 & 2 & 4 & 6 & \dots & 2k-2 & 2k \end{pmatrix}$$

Udowodnij, że dla permutacji K_n , gdzie n jest dodatnią liczbą całkowitą podzielną przez 4, Quick Sort działa w czasie $\Omega(n^2)$.

Rozwiązanie: Zauważmy, że w i -tej iteracji rekurencji jako mediana zostanie wybrany element $2i$, stąd podział tablicy w wyniku partition będzie bardzo nie zrównoważony, a czas działania algorytmu będzie wynosił $T(n, i) \leq T(n - i - 1) + n = O(\sum i) = O(n^2)$.

Zadanie 4.2

Niech n będzie liczbą całkowitą większą od 1, a $X = \{(x, y) : x = 0, 1, \dots, n - 1, y = 0, 1, 2, 3, 4\}$ zbiorem punktów na płaszczyźnie. Danych jest n różnych prostych, z których każda przechodzi przez dwa różne punkty ze zbioru X , różniące się zawsze pierwszą współrzędną. Każda prosta zadana jest przez parę punktów $[(x_1, y_1), (x_2, y_2)]$, $x_1 < x_2$. Zaprojektuj algorytm, który w czasie liniowym posortuje wszystkie proste niemalejąco względem ich kątów nachylenia do osi OX .

Rozwiązanie: Zadanie sprowadza się do sortowania ułamków postaci $a_i = \frac{l_i}{m_i}$, gdzie $l_i \in \{0, \dots, 4\}$, $1 \leq m_i < n$. Wszystkie ułamki z $l_i = 0$ oczywiście wypiszemy na początku. Wszystkie pozostałe ułamki możemy znormalizować tak by licznik wynosił dokładnie 12, a mianownik $1 \leq m'_i < 12n$ — stąd możemy je posortować kubełkowo.

Zadanie 4.3

Podaj permutację liczb $1, 2, \dots, 7$, dla której algorytm Heap Sort (w wersji z wykładu) wykona największą liczbę porównań. Uwaga: należy wziąć pod uwagę obie fazy algorytmu - budowę kopca i właściwe sortowanie.

Rozwiązanie: Worst case dla 1, 4, 2, 5, 6, 3, 7 daje 22 porównania. Dla prównania 3, 1, 2, 4, 5, 6, 7 daje 21 porównania.

Zadanie 4.4

Dla dodatniej liczby całkowitej d , $d > 1$, d -kopcem typu MIN, nazywamy zupełne drzewo d -arne z kluczami rozmieszczonymi w porządku kopcowym typu MIN. Zaproponuj wydajną implementację d -kopca w tablicy i dokonaj analizy złożoności obliczeniowych operacji kolejki priorytetowej: Ini, Min, DeleteMin, Insert i DecreaseKey. W jaki sposób dobrać d , żeby dostać jak najszybszą implementację algorytmu Dijkstry, uwzględniającą liczbę krawędzi w grafie.

Rozwiązanie: d -kopiec do drzewo zupełne o stopniu d z porządkiem kopcowym (min w korzeniu). Należy pokazać, że poszczególne operacje wykonuje się w czasie:

- Min — $O(1)$
- DeleteMin — $O(d \cdot \log_d(n))$
- DecreaseKey — $O(\log_d(n))$

Koszt implementacji algorytmu Dijkstry, przy użyciu d -kopców: $O(nd \cdot \log_d(n) + m \cdot \log_d(n))$.

Zanalizować jak należy dobrać d w zależności od m i n (jeśli za d weźmiemy $\max(2, \lceil m/n \rceil)$ to dostajemy $O(\frac{m \log n}{\log m/n})$).

Zadanie 4.5

Rozwiązanie:

Kopiec lewicowy to drzewo binarne, spełniające:

- warunek kopca: $key(x) \geq key(parent(x))$,
- oraz $dist(left(x)) \geq dist(right(x))$, gdzie $dist(x)$ jest odległością do najbliższego potomka o mniej niż 2 synach (przyjmujemy, że $dist(null) = -1$).

Więcej informacji na temat kopców lewicowych można odnaleźć pod adresem: https://en.wikipedia.org/wiki/Leftist_tree.

Przydatne własności:

- Jeśli v jest korzeniem kopca lewicowego, to zawiera on co najmniej $2^{dist(a)}$ węzłów.
- Czyli jeśli kopiec zawiera n węzłów, to $dist(root) = O(\log n)$.
- Dla każdego węzła v spełniony jest warunek $dist(x) = dist(right(x)) + 1$ ($dist(null) = -1$).

Podstawową operacją jest złączanie dwóch kopców:

MERGE(a, b)

- 1: **if** $a = \text{null}$ **then**
- 2: return b ;

```

3: else if  $b = \text{null}$  then
4:   return  $a$ ;
5: end if
6: if  $\text{key}(b) < \text{key}(a)$  then
7:   swap( $a, b$ )
8: end if
9:  $\text{right}(a) = \text{merge}(\text{right}(a), b)$ 
10: if  $\text{dist}(\text{right}(a)) > \text{dist}(\text{left}(a))$  then
11:   swap( $\text{right}(a), \text{left}(a)$ )
12: end if
13: if  $\text{right}(a) = \text{null}$  then
14:    $\text{dist}(a) = 0$ 
15: else
16:    $\text{dist}(a) = 1 + \text{dist}(\text{right}(a))$ 
17: end if
18: return  $a$ 

```

Łatwo pokazać, że złożoność operacji merge wynosi $O(\text{dist}(a) + \text{dist}(b))$, czyli $O(\log n)$.

Operacje *insert* i *extractMin* można zaimplementować używając operacji *merge*.

INSERT(r, x)

```

1:  $p = \text{MAKETREE}(x)$ 
2:  $r = \text{MERGE}(r, p)$ 

```

EXTRACMIN(r)

```

1:  $min = r.value$ 
2:  $r = \text{MERGE}(r.left, r.right)$ 
3: return  $min$ 

```

Operacje IncreaseKey/DecreaseKey możemy zaimplementować w następujący sposób (T – kopiec lewicowy, v – zmieniany węzeł, x – nowa wartość węzła x):

- usuwamy wskazany węzeł v z kopca T (musimy zadbać by wszystkie warunki kopca lewicowego były nadal zachowane)
- tworzymy jednoelementowy kopiec lewicowy T' z wartością x ,
- scalamy T i T' .

Usuwanie węzła z kopca możemy wykonać w następujący sposób:

REMOVENODE(v)

```

1: zastąp  $v$  przez  $\text{MERGE}(v.left, v.right)$ 
2:  $p = v.parent$ 
3: while  $p \neq \text{nil}$  do
4:   if  $\text{dist}(p.right) > \text{dist}(p.left)$  then
5:     SWAP( $p.left, p.right$ )
6:   end if
7:   if  $\text{dist}(p) = \text{dist}(p.right) + 1$  then
8:     break
9:   else
10:     $\text{dist}(p) = \text{dist}(p.right) + 1; p = p.parent$ 
11:   end if
12: end while

```

Analiza pojedynczego kroku pętli while:

- v należy do poddrzewa $p.right$:
 - jeśli wykonano operację *swap*, to kopiec musi mieć co najmniej 2^k węzłów (gdzie k to odległość pomiędzy v i p)
 - wpp. algorytm kończy działanie,
- v należy do poddrzewa $p.right$, jednak zauważmy, że liczba kroków tego typu nie może przekroczyć $O(\log n)$

Zadanie 4.6

Rozwiązanie: Aby otrzymać czas $O(NW + M)$ potrzebujemy kolejki priorytetowej o następujących czasach wykonania poszczególnych operacji:

- EXTRACMIN — $O(W)$
- DECREASEKEY — $O(1)$

Wystarczy zauważyć, że jeśli do jakiegoś wierzchołka istnieje droga, to jej długość jest $\leq NW$. Czyli potrzebujemy tablicy NW elementowej (i -ty element tablicy zawiera listę niedowiedzonych wierzchołków w odległości i od wierzchołka początkowego).