Z41

Algorytm Moore'a znajduje k-ty element w ciągu, w tym przypadku szukany medianę czyli $\frac{n}{2}$.

Algorytm wybiera pivota i rekurencyjnie wywołuje się na odpowiednim fragmencie ciągu, aż liczba elementów będzie mała, wtedy sortuje i bierze k-ty element.

Cała działanie tego algorytmu zależy głównie od efektywności wybierania pivota.

Metoda Fredmana:

W tym podejściu nasz algorytm wybiera pivota aż znajdzie takiego który podzieli ciąg wyjściowy w proporcji conajmniej 1:3.

Zauważmy, że dokładnie połowa liczb ciągu podzieli nam tablicę w ten sposób (wszystko oprócz $\frac{1}{4}n$ min i $\frac{1}{4}n$ max elementów).

Czyli $p = \frac{1}{2}$. Zatem oczekiwana liczba prób do znalezienia poprawnego pivota to 2. ([redacted])

($\bar{E}(x)$ dla rozkładu geometrycznego to $\frac{1}{p}$)

Dla danego pivota i ciągu n liczb wykonujemy $n-1$ porównań, czyli przy uwzględnieniu wartości oczekiwanej $2n-2$.

Ile razem wykonamy porównań dla wyjściowego ciągu?

$$2n + 2 \cdot \tfrac{3}{4}n + 2 \cdot (\tfrac{3}{4})^2 \cdot n + \ldots + \underbrace{C}_{\text{finalny sort}} \leq \frac{2n}{(1-\frac{3}{4})} + C = 8n + C$$

Czyli złożoność $O(n)$

łatwo zauważyć, że tak samo będzie w standardowym podejściu, gdy bierzemy dowolnego pivota.

Z $p=\frac{1}{2}$ pivot będzie dobry, a w przypadku liniowego wyjścia $n-1$ operacji "na darmo". Ale analogicznie jak poprzednio zauważamy, że $E(X)=2$, czyli co drugi pivot będzie ok, więc obliczenia będą takie same.

Zatem $O(n)$.

Z5

Mamy strukturę drzewa binarnego o własnościach:

- kopiec, czyli wartości lewego dziecka $\geq$ ojca
- $h(x.left) \geq h(x.right)$, gdzie h to odległość do najbliższego nulla.

Łatwo zauważyć, że idąc zawsze do prawego dziecka najszybciej dotrzemy do nulla.

Co więcej, zauważmy, że jeśli $h(root) = x$ to wszystkie wierzchołki na głębokości $x-1$, mają dwóch synów.

W przeciwnym wypadku $h(root)$ byłoby mniejsze niż x.

A z tego wynika, że rozmiar takiego drzewa to co najmniej $2^x - 1$, a $h(root) \leq \log(n)$, gdzie n to liczba wierzchołków w drzewie.

Skorzystamy z tego faktu implementując funkcje do naszej kolejki.

Będziemy chcieli napisać 3 funkcje: pop_min()
insert(v)
merge(T_1, T_2)

- pop_min()

Aby usunąć element minimalny, wystarczy usunąć korzeń i zmergować synów:

```
def pop_min():
    res = T.val
    T = merge(T.left, T.right)
    return res
```

- insert

Wystarczy utworzyć drzewo składające się tylko z nowej wartości i zmergować z docelowym:

```
def insert(v):
    new_T = T(v)
    merge(T, new_T)
```

- merge

Będziemy rekurencyjnie łączyć kopiec o większym korzeniu do prawego poddrzewa drzewa o mniejszym korzeniu przy okazji dbając o zachowanie obu własności drzewa.

```
def merge(T1, T2):
    if T1 == NULL:
        return T2
    if T2 == NULL:
        return T1

    if T1.value > T2.value:          // chcemy łączyć T2 do poddrzewa T1
        swap(T1, T2)

    T1.right = merge(T1.right, T2)   // wie null, bo B != null

    if T1.left == NULL:              // przywracamy niemość leftist tree
        swap(T1.left, T1.right)
        T1.h = 1
    elif T1.right.h > T1.left.height:
        swap(T1.right, T1.left)
        T1.h = T1.right.h + 1        // poza ifem

    return T1
```

Złożoność operacji:

- merge - w najgorszym przypadku będziemy na zmianę przechodzić przez prawą ścieżkę T_1 i T_2 czyli $O(\log_2(|T_1|) + \log_2(|T_2|))$

- pop_min() - analogicznie $O(\log_2(|T_1.left|) + \log_2(|T_1.right|))$

- insert - tutaj mergujemy tylko z 1 wierzchołkiem, więc $O(\log_2(n))$