# AISD lista 7

Dominik Szczepaniak

June 3, 2024

## 1 Zadanie 1

Czas wykonania operacji na słowniku wyraża się wzorem
$\sum_i p_i \cdot h_i$

h-wysokośc, czas oczekiwany na wykonanie polecenia
p-prawdopodobieństwo odpowaidającej liczbie zapytań
Dążymy do tego, aby wartośc ta była jak najmniejsza. Do rozwiązania tego zadania wykorzystamy programowanie dynamiczne, a by rozpatrzeć każde możliwe drzewo.

Oznaczmy $T_{i,j}$ jako minimalny koszt drzewa zbudowanego z elementów o indeksach z zakresu $< i, j >$.

Przypadki brzegowe: Dla $T_{i,i}$ czas oczekiwania będzie równy $p_i$.

Załóżmy, że wybieramy klucz k jako korzeń drzewa od i do j:

Musimy wziąć więc prefix k-1, sufix k+1 i dodać do tego wszystkie prawdopodobieństwa od i do j. (te od i do k-1 oraz od k+1 do j będą musiały zejść o 1 stopień w dół, więc będziemy musieli dodać do nich 1 stąd suma)

Niech d(s, T) oznacza depth klucza s w drzewie T

$T_{i,j} = \sum_{s\in T} d(s,T)*p(s) = p(S_k)+\sum_{s\in T_L}(d(s,T_L)+1)*p(s)+\sum_{s\in T_R}(d(s,T_R)+1)*p(s)$

$= \sum_{s\in T} p(s) + \sum_{s\in T_L}(d(s,T)) * p(s) + \sum_{s\in T_R}(d(s,T)) * p(s) =$

$= \sum_{s\in T} p(s) + cost(T_L) + cost(T_R)$

Sumy prawdopodoieństw możemy liczyć za pomocą sum prefiksowych.

Tablica n x n będziemy wypełniali wstępująco, zaczynając od przypadków brzegowych. Kolejno dla $T_{i,j}$ gdzie $|i-j| = 1$, potem gry róznica ta jest równa 2,3,4 itd...

Tablica jest kwadratowa, a wyliczenie każdej wartość zajmuje nam czas liniowy, zatem $On^3$

Odzyskanie informacji o tym jak zbudowane jest drzewo jest proste. Wystarczy stworzyć osobną tablicę n-elemetnową i na bieżąco zapisywać w niej wyniki.

# 2 Zadanie 3

Do tablicy n-elementowej wstawiamy n kluczy.

Szansa, że wybrany klucz nie trafi do jakiejś listy wynosi $(n-1)/n$ (bo prawdopodobieństwo, że do niej trafił jest $1/n$ czyli mamy $1 - 1/n$).

Prawdopodobieństwo, ze klucz trafi do listy jest dla kazdej listy takie samo więc oczekiwana ilość pustych list wynosi $E = n * P$ (z liniowości wartości oczekiwanej) gdzie P to prawdopodobieństwo tego, że każda lista jest pusta (prawdopodobieństwo łączne)

List jest n więc

$P = ((n-1)/n)^n$ (wybrany klucz nie trafi do wybranej listy n razy)

Stąd mamy że $E = n * ((n-1)/n)^n$

# 3 Zadanie 4

**Niezminnik kredytow:** Każdy korzeń ma odłożona jednostkę, ponad to każdy wierzchołek, który utracil syna ma tyle odłożonych jednostek co utraconych synów.

**INSERT**

“‘python=

def insert(H, x):

x.p = nil

x.child = nil

x.mark = 0

if H.min == nil:

lista korzeni zawierająca jedynie x

H.min = x

else
wstaw x na listę korzeni
if x.key < H.min.key
H.min = x
H.n = H.n + 1
“‘

Podczas wykonywania tej operacji:
- dołączamy do listy już gotowe drzewo (1 operacja)
- porównujemy z min i ew przepinamy wskaźnik (0 lub 1 operacja)
- przydzielamy jedną jednostkę kredytową korzeniowi w nowym drzewie i jedną już wykorzystaliśmy na dopisanie nowego elementu

Koszt zamortyzowany to $O(1)$.

**FINDMIN**
“‘python=
def findmin(H):
return H.min
“‘

Mamy wskaźnik, który wskazuje na najmniejszy element. Wytarczy więc zobaczyć na co wskazuje.

Nic nie zmieniamy w kopcu, zatem koszt zamortyzowany to $O(1)$.

**DECSREASE KEY**

```
def decrease_key(H, x, k):
    x.key = k
    y = x.p
    if y != nil and x.key < y.key: #jezeli ma rodzica i jest zaburzo
        cut(H, x, y)
        cas_cut(H, y)
    if x.key < H.min.key:
        H.min = x

def cut(H, x, y):
    usun x z listy synow y
    dodaj x do listy korzeni
    x.p = nil
    x.mark = false

def cas_cut(H, y):
```

```
        z = y.p
        if z != nil:
            if y.mark == 2:
                y.mark == 3
            else if u.mark == 1:
                y.mark = 2
            else if y.mark = 0:
                y.mark = 1
            else
                cut(H, y, z)
                cas_cut(H, z)
```

Ucinym i przekładamy poddrzewo jako osobny kopiec. Jemy i jego ojcu przydzielamy po 1 jednostce kredytowej.

Musimy również sprawdzić czy przodek nie ma uciętego trzeciego syna, czyli czy może przypadkiem nie otrzymał właśnie trzeciej jednostki kredytowej. Jeśli tak to wykonujemy odcięcie kaskadowe: przodka wraz z jednostkami kredytowymi przenosimy go jako nowy kopiec w liście, za co płacimy 1 środek, a ojcu oddajemy ostatni 1 środek. Zauważmy, że odcięcie nie ma wpływu na koszt całości.

Zatem koszt zamortyzowany całej operacji to $O(1)$.

**MELD**

```
def meld(H1, H2):
    stworz nowy pusty kopiec H
    H.min = H1.min
    laczymy liste korzeni kopcow H2 i H1
    if H1.min == nil or (H2.min != nil and H2.min.key < H1.min.key):
        H.min = H2.MIN
    H.n = H1.n + H2.n
    return H
```

W przypadku złączenie dwóch kopców przpinamy jedynie wskaźniki.

Zatem koszt zamortyzowany całej operacji to $O(1)$.

**DELETMIN**

```
def deletemin(H):
    z = H.min
    if z != nil:
        dla kazdego syna z:
```

```
        x = aktualnie rozpatrywany syn z
        dodaj x do listy korzeni
        x.p = nil
    usun z z listy korzeni H
    if z == z.right  #jesli jedno elementowa
        H.min = nil
    else
        H.min = z.right
        znalezienie minimum
    H.n = H.n - 1
```

Wykonując operację deletemin musimy zadbać o usunięcie minimum, przepięcia synów minimum, przydzielenia im jednostki kredytowej oraz znalezienie nowego minimum.

Przepięci oraz przydzielenie jednoskti kredytowej zajmie nam $O(lgn)$, ponieważ w najgorszym przypadku mamy $lgn$ dzieci.

Po usunięciu minimum musimy skompaktować wszystkie drzew, a nie wiemy ile ich jest. W każdym z teych drzew jest odłożona jednostka kredytowa, a przez każde drzewo wsytarczy przejśc tylko raz, więc kompaktowanie opłacamy z jednostek kredytowych, a nastepnie pozostaje nam conajwyżej logn drzew w kopcu Fibonacciego.

Połączenie drzew o jednakowym stopniu polega na tym, że to drzewo o mniejszym korzeniu przyłączamy drzewo o większym korzeniu.

Musimy zatem przejść jeszcze po (w najgorszym przpypadku) $lgn$ korzeniach, a by wybrac minimum.

Zatem czas zamortyzowany $O(lgn)$.

# 4 Zadanie 5

Tworzymy słownik stały dostając n kluczy na wejściu.

W takim razie, jeśli chcemy robić słownik stały tak jak na wykładzie to mamy najpierw jedną tablicę, dla której musimy obliczyć hash - w takim razie musimy wylosować funkcje hashującą (w najgorszej opcji 2 razy), a później dla każdego klucza tworzymy kolejne tablice o rozmiarach ${n_j}^2$, co z wykładu wiemy, że sumuje się do O(n) (a dokładnie pod spodem jest około 4n).

Przeanalizujmy w takim razie wszystko od początku - najpierw tworzymy tablicę o rozmiarze n, a potem szukamy hasha i wrzucamy wszystkie elementy, gdzie w najgorszej opcji będziemy musieli to zrobić dwa razy,czyli mamy O(2n) = O(n).

Następnie dla każdej tablicy będziemy musieli stworzyć tablicę o rozmiarze ${n_j}^2$, co z wykładu wiemy, że sumuje się do O(n) dla wszystkich liczb, więc te tworzenia tablic również nie robią nam żadnego problemu.

Następnie dla każdej z podtablic musimy wylosować hasha co najwyżej 2 razy, czyli mamy znowu O(n * 2) = O(n), czyli dalej nas nie boli.

W takim razie każdy krok kosztuje nas O(n), a tych kroków jest 3 - więc mamy O(n).

# 5 Zadanie 6

SKIP ALE ROZW:

współczynnik zapełnienia $\alpha = n/m$

W adresowniu otwartym na pozycję przypada co najwyżej 1 element więc n<=m czyli $\alpha <= 1$

Dla zadanego rozkładu prawdopodobieństwa wystapienia kluczy oraz zachowania funkcji haszujacej na kluczach prwadopodobieństwo wystapienia kazdego ciągu kontrolnego jest takie samo.

Lemat 1. (z wykładu)

`![](https://i.imgur.com/zJn1H75.jpg)`

A.12

`![](https://i.imgur.com/xfe86Al.jpg)`

Dowód:

Wyszukanie elementu k wymaga wykonania tego samego ciągu sprawdzeń jak wstawianie. Z lematu 1 mamy, że jeśli k był i+1-wszym elementem wstawionym do tablicy to oczekiwana liczba sprawdzeń potrzebnych do jego odszukania jest nie większa niż $1/(1 - i/m) = m/(m - i)$. Uśredniając wszystkie n kluczy znajdujace się w tablicy otrzymujemy oczekiwaną liczbę sprawdzeń:

$1/n \sum_{i=0}^{n-1} m/(m - i) =$

$m/n \sum_{i=0}^{n-1} 1/(m - i) =$

$1/\alpha \sum_{k=m-n+1}^{m} 1/k <= (zA.12)$
$1/\alpha \int_{m-n}^{m} (1/x)dx$

$= 1/\alpha * ln(m/(m-n)) =$

$1/\alpha * ln(1/(1-\alpha)$