Zadanie 1. (2pkt)

Niech $A = \{a_1, a_2, \dots, a_n\}$ będzie zbiorem kluczy, (takim, że $\forall i = 1, \dots, n-1\ a_i < a_{i+1}$), które chcemy pamiętać w słowniku stałym. Znamy także ciąg $p_1, \dots, p_n$ prawdopodobieństw zapytania o poszczególne klucze. Przyjmujemy, że $p_1 + p_2 + \cdots + p_n = 1$, a więc do słownika nie będą kierowane zapytania o klucze spoza słownika. Chcemy zaimplementować słownik jako drzewo BST. Ułóż algorytm znajdujący takie drzewo, które minimalizuje oczekiwany czas wykonywania operacji na słowniku.

- (P) Algorytm ma działać w czasie $O(n^3)$.
- (Z) Algorytm ma działać w czasie $O(n^2)$.

Wersja P – czas $O(n^3)$

Oczekiwany czas operacji zależy od tego, na jakim poziomie są poszczególne elementy, zatem chcemy sprawić, aby elementy najczęściej odwiedzane (z największym prawdopodobieństwem) były jak najbliżej korzenia.
Zatem celem zadania jest znaleźć drzewo z minimalną wartością:

$$T(A) = \sum_{i=1}^{n} p_i h_i$$

Rozwiązanie z użyciem programowania dynamicznego – znajdujemy koszty najtańszych drzew dla elementów na przedziale <i,j> wiedząc, że $T_{i,i} = p_i$ oraz:

$$T_{i,j} = \min_{i \le k \le j} \left(\left(T_{i,k-1} + \sum_{l=i}^{k-1} p_l \right) + p_k + \left(T_{k+1,j} + \sum_{l=k+1}^{j} p_l \right) \right)$$

Wyjaśnienie: wybieramy k-ty element jako korzeń, mniejsze lądują w lewym poddrzewie (który jest 1 poziom niżej, stąd trzeba dodać podaną sumę), a większe w prawym. Wzór ten skraca się do:

$$T_{i,j} = \sum_{l=i}^{j} p_l + \min_{i \le k \le j} \left(T_{i,k-1} + T_{k+1,j} \right)$$

Oczywiście jeśli k=i lub k=j to wtedy istnieje tylko jedne poddrzewo, a drugie ma wartość 0.
Żeby ułatwić odtwarzanie drzewa będziemy też trzymać tablicę Roots, która będzie przechowywała te optymalne wartości k.

```
// zwraca tablicę indeksów korzeni  optymalnych poddrzew <i,j>
CalcDP(p, n)
T – tablica n x n kosztów optymalnych poddrzew <i,j>
Roots – tablica n x n korzeni drzew T
for i from 1 to n:
        T[i][i] = p[i] // drzewo z tylko 1 elementem
        Roots[i][i] = i // zatem musi on być korzeniem
for size from 1 to n-1: // ilość elementów poddrzewa
        for i from 1 to n: // rząd
                k = FindBestRoot(T, i, i+size)
                Roots[i][i+size] = k
                if (k == i) // wszystkie elementy są w prawym poddrzewie
                        T[i][i+size] = Sum(p, i, i+size) + T[k+1][i+size]
                else if (k == i+size) // wszystkie elementy są w lewym poddrzewie
                        T[i][i+size] = Sum(p, i, i+size) + T[i][k-1]
                else
                        T[i][i+size] = Sum(p, i, i+size) + T[i][k-1] + T[k+1][i+size]
return Roots

// znajduje najlepszego kandydata na korzeń drzewa <i,j>
FindBestRoot(T, i, j)
if (i == j)
        return i
best = i
lowestSum = T[i+1][j]
for k from i+1 to j-1:
        sum = T[i][k-1] + T[k+1][j]
        if (sum < lowestSum)
                lowestSum = sum
                best = k
if (T[i][j-1] < lowestSum) // tylko lewe poddrzewo
        return j
return best

// odtwarza optymalne drzewo na podstawie tablicy dynamicznej
FindTree(a, p, n)
Roots = CalcDP(p, n)
return AddNodes(a, Roots, 1, n)
```

```
// dodaje elementy z a do drzewa optymalnego
AddNodes(a, Roots, i, j)
if (i == j)
        return newNode(a[i])
k = Roots[i][j] // korzeń poddrzewa z elementów a[i,…,j]
Node.value = a[k]
Node.left = AddNodes(a, Roots, i, k-1)
Node.right = AddNodes(a, Roots, k+1, j)
return Node
```

Zadanie 2 (2 pkt)

2. (2pkt) Ułóż algorytm rozwiązujący problem znajdowania najbliżej położonej pary punktów na płaszczyźnie oparty na następującej idei. Niech d będze odległością pomiędzy parą najbliżej położonych punktów spośród punktów $p_1, p_2, \ldots, p_{i-1}$. Sprawdzamy, czy p_i leży w odległości mniejszej niż d, od któregoś z poprzednich punktów. W tym celu dzielimy płaszczyznę na odpowiednio małe kwadraty, tak by w każdym z nich znajdował się nie więcej niż jeden punkt. Te "zajęte" kwadraty pamiętamy w słowniku.

   Twój algorytm powinien działać w oczekiwanym czasie liniowym. Jeśli nie potrafisz zbudować algorytmu opartego na powyższej idei, możesz opracować algorytm oparty na innej (ale spełniający te same wymagania czasowe).

Zadanie 3. (1pkt)

Oblicz jaka jest oczekiwana liczba pustych list po umieszczeniu n kluczy w tablicy haszującej o n elementach.

Wstawiamy n kluczy, każdy do jednej z n list.
Każda z list ma takie same prawdopodobieństwo, że jakaś liczba do niej wpadnie, zatem i-ta lista ma ppb $1 - \frac{1}{n}$, że j-ty element do niej NIE wpadnie.
Dowolna lista będzie pusta jeżeli wszystkie elementy wpadną do innych list, co jest z ppb $\left(1 - \frac{1}{n}\right)^n$. List jest n, stąd oczekiwana liczba pustych list to $n * \left(1 - \frac{1}{n}\right)^n$.

Zadanie 4. (1pkt)

Przeprowadź analizę zamortyzowanego kosztu ciągu operacji insert, deletemin, decrease-key, meld, findmin wykonywanych na kopcach Fibonacciego, w których kaskadowe wykonanie operacji cut wykonywane jest dopiero wtedy, gdy wierzchołek traci trzeciego syna.

Oparte na analizie z Cormena, rozdział 21.2

Kopiec Fibonacciego to taka struktura, w której zapamiętujemy zbiory drzew w porządku kopcowym (min). Dopóki nie wykonujemy operacji deletemin oraz decrease-key to każde drzewo kopców Fibonacciego zachowuje się tak samo jak te kopców dwumianowych, tzn. mamy co najwyżej 1 drzewo każdego stopnia.

1. Analiza Insert

```
Insert(H,x)
x.parent = nil
x.child = nil
x.mark = 0
if (H.min == nil)
        H.min = x
else
        wstaw x na listę korzeni
        if (x < H.min)
                H.min = x
H.n += 1
```

Ze względu na leniwą naturę kopców Fibonacciego Insert(H,x) polega na stworzeniu (w czasie stałym) kopca 1-elementowego zawierającego x, a następnie dołączeniu go do listy korzeni H (w czasie stałym). Zamortyzowany koszt to jest O(1), a maksymalna ilość jednostek konieczna do wykonania insert to 2 – jedna na dołączenie x do listy drzew korzenia H, druga na ewentualne przepięcie wskaźnika minimum.

2. Analiza Deletemin

```
Deletemin(H)
z = H.min
if (z != nil)
        dla każdego x - syna z:
                dodaj x do listy korzeni
                x.parent = nil
        usuń z z listy korzeni
        if (z.right == z)
                H.min = nil
        else
                H.min = z.right
```

```
            Napraw(H)
        H.n -= 1
```

To tutaj naprawiamy strukturę kopca tak, aby nie było zbyt wiele drzew.
W etapie 1 znajdujemy to poddrzewo, które ma wartość H.min w korzeniu. Wszystkie jego T dzieci podłączamy do list korzeni, zwiększając ich ilość do T+O(logn), a następnie usuwamy to poddrzewo. W etapie 2 dokonujemy w procedurze Napraw scalania tych drzew, które są tej samej wysokości tak długo, aż wszystkie drzewa będą różnej wysokości. Wtedy zostanie O(logn) drzew, zatem $\Delta\Phi = -T + O(logn)$
Stąd łączny koszt zamortyzowany to:

$$O(T + logn) + O(1) * \big(-T + O(logn)\big) =$$
$$= O(T) - O(1) * T + O(1) * O(logn) = O(logn)$$

3. Analiza Decrease-key

```
DecreaseKey(H, x, newKey)
if (x.key <= newKey) // nie zwiększamy klucza
        return
x.key = newKey
y = x.parent
if (y != nil and x.key < y.key) // zburzono porządek kopca min
        Cut(H, x, y)
        Cas_Cut(H,y)
if (x.key < H.min) // nowe minimum kopca
        H.min = x

// Usuwanie x od ojca y i przeniesienie go do listy korzeni
Cut(H, x, y)
Usuń x z listy synów y oraz zmniejsz y.degree o 1
Dołącz x do listy korzeni H
x.parent = nil // skoro x jest korzeniem to nie ma rodzica
x.mark = 0 // usunięto 0 dzieci x

// kaskadowanie ucinanie dzieci
Cas_Cut(H, y)
z = y.parent
if (z != nil) // y nie może być korzeniem
```

```
if (y.mark < 3) // usunięto mniej niż 3 dzieci y, nie ucinaj
        y.mark += 1
else
        Cut(H, y, z)
        Cas_Cut(H, z)
```

Łatwo zauważyć, że jeżeli DecreaseKey nie wywoła Cas_Cut to czas wykonania jest O(1). Pokażę jednak, że nawet jeśli czasem mamy wielokrotne wywołanie Cas_Cut to czas zamortyzowany nadal wynosi O(1) i to nawet dla ucinania po stracie 2 syna (a co dopiero dla 3):
Zdefiniujmy funkcję potencjału (co to jest – Cormen 18.3) jako:
$\Phi = n + 2m$, gdzie n to ilość drzew, m to ilość wierzchołków z mark > 0.
Wynika to stąd, że im więcej drzew, tym dłużej trwa Deletemin, a ilość wierzchołków z mark > 0 ma podwójne znaczenie – wpływa na ilość kaskadowych cięć w DecreaseKey oraz na ilość drzew i czas Deletemin.
W momencie, gdy DecreaseKey dokonuje C cięć, odpowiada to:
1. Zwiększa mark 1 wierzchołkowi (+2)
2. Wyzerowuje mark C wierzchołkom (-2C)
3. Dodaje do listy korzeni C wierzchołków (+C)
Wtedy koszt zamortyzowany DecreaseKey to:
$O(C) + O(1) * \Delta\Phi = O(C) + O(1) * (2 - 2C + C) = O(C) + O(1) * (2 - C) =$
$= O(C) + O(1) - O(C) = O(1)$■

4. Analiza Meld

```
Meld(H1, H2)
H = Make-Fib-Heap()
H.min = H1.min
H.roots = Merge(H1.roots, H2.roots)
if (H1.min == nil or (H2.min != nil and H2.min < H1.min))
        H.min = H2.min
H.n = H1.n + H2.n
return H
```

Scalanie w Merge wykonuje się leniwie w O(1), zatem łączny koszt Meld to O(1).
Wynika to stąd, że jedynie przepinamy stałą liczbę wskaźników, max 6:
raz lub dwa H.min, H1.roots.first.prev, H1.roots.last.next, H2.roots.first.prev, H2.roots.last.next.

5. Analiza Findmin

```
FindMin(H)
        return H.min
```

Każdy kopiec Fibonacciego zawiera wskaźnik min na minimalny element, zatem operacja Findmin działa w czasie O(1) z wykorzystaniem 1 jednostki.

Zadanie 5. (1pkt)

Oszacuj oczekiwany czas tworzenia słownika stałego
(metodą podaną na wykładzie).

// todo

Zadanie 6. (1 pkt)

Rozważamy słownik utworzony metodą adresowania otwartego.
Niech n będzie rozmiarem słownika, m - rozmiarem tablicy.
Udowodnij, że przy założeniu, że ciąg $< h(k,0), \dots, h(k,m-1) >$ jest
z równym prawdopodobieństwem dowolną permutacją zbioru $\{0, \dots, m-1\}$,
oczekiwana liczba prób w poszukiwaniu zakończonym sukcesem
jest $\leq \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$, gdzie $\alpha = \frac{n}{m} < 1$.

Wstęp:
Adresowanie otwarte – wszystkie elementy przechowujemy w tablicy, każde jej pole jest pojedynczą wartością a nie listą jak w adresowaniu łańcuchowym. Tutaj z kolizjami radzimy sobie w taki sposób, że mamy funkcję postaci f(x,i), która dla argumentu x z uniwersum oraz $i \in \{0,1, \dots, m-1\}$ wyznacza ciąg pozycji, na których x może się znajdować w tablicy. Argument i pozwala nam określić inną pozycję jeśli początkowa jest zajęta przez inną wartość. Przykład:

$$m = 10, n = 100, f(x,i) = (x+i)\ mod\ 10$$

Chcemy wstawić elementy $\{11,57,31,45,71\}$ w tej kolejności. Zatem:
$f(11,0) = 1$, wolne, zatem 11 będzie na indeksie 1
$f(57,0) = 7$, wolne, zatem 57 będzie na indeksie 7
$f(31,0) = 1$, zajęte przez 11, zatem zwiększamy i do 1
$f(31,1) = 2$, wolne, zatem 31 będzie na indeksie 2
$f(45,0) = 5$, wolne, zatem 45 będzie na indeksie 5
$f(71,0) = 1$, zajęte przez 11, zatem zwiększamy i do 1
$f(71,1) = 2$, zajęte przez 31, zatem zwiększamy i do 2

$f(71,2) = 3$, wolne, zatem 71 będzie na indeksie 3

Część właściwa dowodu:
Jeśli dany element był wstawiony do słownika jako (i+1)-szy, to trzeba wykonać średnio $\frac{m}{m-i}$ porównań aby go odnaleźć. Stąd dla dowolnego elementu średnia ilość porównań aby go odnaleźć wynosi

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} = \frac{1}{\alpha}\left(\sum_{i=1}^{m}\frac{1}{i} - \sum_{i=1}^{m-n}\frac{1}{i}\right) = \frac{1}{\alpha}(H_m - H_{m-n})$$

Z twierdzenia 3.10 z Cormena wiemy, że dla funkcji malejących
(a ciąg harmoniczny taką jest) zachodzi:

$$\int_m^{n+1} f(x)\,dx \le \sum_{k=m}^{n} f(k) \le \int_{m-1}^{n} f(x)\,dx$$

Z czego wynika, że:

$$\sum_{k=1}^{n} f(k) \ge \int_1^{n+1}\frac{1}{x}dx = \ln(n+1) \ge \ln(n)$$

$$\sum_{k=1}^{n} f(k) \le \int_1^{n}\frac{1}{x}dx + 1 = \ln(n) + 1$$

Zatem otrzymujemy:

$$\ln(n) \le \sum_{k=1}^{n} f(k) = H_n \le \ln(n) + 1$$

Wracając do poprzedniego równania:

$$\frac{1}{\alpha}(H_m - H_{m-n}) \le \frac{1}{\alpha}(\ln(m) - \ln(m-n)) = \frac{1}{\alpha} * \ln\left(\frac{m}{m-n}\right) =$$

$$= \frac{1}{\alpha} * \ln\left(\frac{\frac{m}{n}}{\frac{m}{n}-1}\right) = \frac{1}{\alpha} * \ln\left(\frac{\frac{1}{\alpha}}{\frac{1}{\alpha}-1}\right) = \frac{1}{\alpha} * \ln\left(\frac{1}{1-\alpha}\right) \blacksquare$$