**Київський національний університет імені Тараса Шевченка**

**Факультет комп’ютерних наук та кібернетики**

**Кафедра інтелектуальних програмних систем**

**Алгоритми та складність**

**Завдання № 4**

**Варіант № 3**

**Звіт**

**Виконав:**

студент групи К-29

Грищенко Юрій Анатолійович

**Київ-2020**

**Завдання**

Реалізуйте оптимальне бінарне дерево пошуку (динамічне програмування).

Предметна область: відділ кадрів (варіант 3). Об’єкти: відділення фірми, працівники. Маємо множину відділень, у кожному відділенні зберігається множина працівників.

**Теорія**

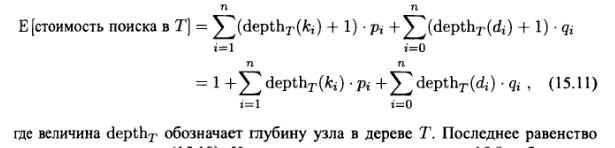
Припустимо, що у нас є відсортовона множина працівників, що містить багато елементів, і нам необхідно досить часто проводити по ній пошук (за іменем). Також відомо, що деякі працівники “цінніші” за інших, і їх імена потрібно шукати набагато частіше за інших.

Це можна реалізувати за допомогою бінарного дерева пошуку.

В завданні №2 ми розгядали збалансовані дерева. Такі структури даних дають змогу швидко проводити операції над ними (в більшості випадків, за час O(log n)), але ми припускали, що всі елементи рівноцінні. В цьому ж випадку, якщо ми знаємо, що деякі елементи використовуються частіше за інших, то є сенс зберігати їх ближче до кореня, а менш цінних працівників зберігати глибше.

Нехай у нас є множина з N елементів (k0, k1, … kn-1), і для кожного з них відома вірогідність пошуку (p0, p1, … pn-1). Оскільки є вірогідність того, що шукатиметься елемент, якого немає в нашій множині, введемо також фіктивні елементи (d0, d1, … dn) з вірогідностями q0, q1, … qn. qo  позначає вірогідність попадання елемента, меншого за k0, qn - більшого за kn-1, інші qi – між ai-1 та ai/

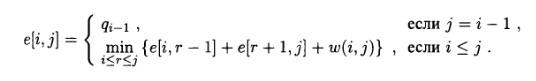
Тоді можна ввести поняття **ціни дерева**:

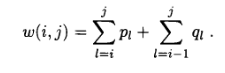
 Нам необхідно побудувати дерево так, щоб E було мінімальним.

Відомо, що в цій задачі є **оптимальна підструктура**: щоб дерево пошуку було оптимальним, необхідно, щоб його піддерева також були оптимальними. Можна вивести рекурсивну формулу.

Введемо позначення e[i,j] як ціну дерева, що містить елементи ki-1, …, kj-1 і відповідні фіктивні елементи di-1, … dj. Припустимо, що e[i,i-1] відповідає дереву, що містить лише di-1.

Тоді маємо:

 де

 (виведення докладно розписано в [1, с. 433]).

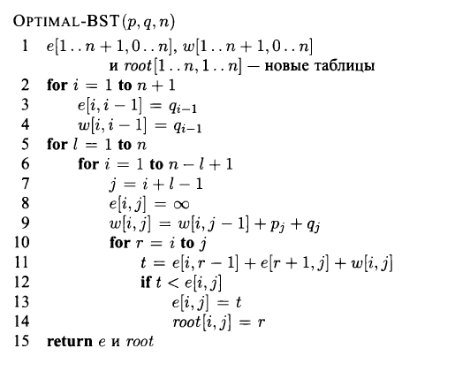
**Алгоритм**

Щоб знайти значення e для піддерев розміром N, необхідно розрахувати значення для піддерев розміром N-1, і т.д. Отже, замість того, щоб користуватися звичайною рекурсією (яка в цьому випадку займала би O(n^4) часу), скористаємося **принципами динамічного програмування**.

Знайдемо значення e для піддерев розміром 1 і збережемо їх. Далі, ці значення використаємо для обробки піддерев розміром 2, і т.д. поки не знайдемо значення для всього дерева розміром N. Це дозволяє уникнути повторних розрахунків одних і тих самих значень, як це буває при використанні рекурсії.

Також оптимізуємо алгоритм за рахунок зберігання w(i, j). Значення w використовується на кожному кроці алгоритму, і кожного разу воно рахується за час O(n). Підрахувавши його один раз і зберігши його, зможемо покращити час виконання всього алгоритму від O(n^4) до O(n^3).

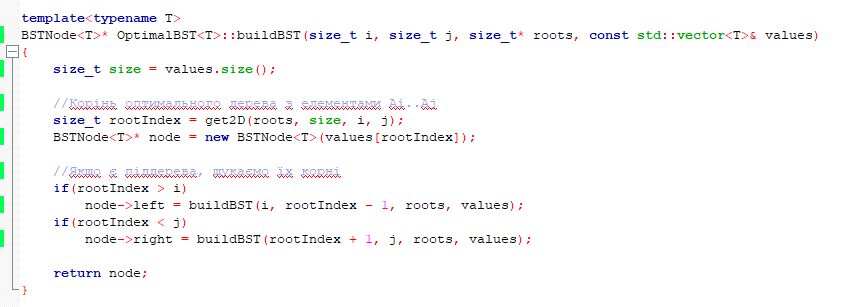
Маємо алгоритм (1), який підрахує ціну оптимального бінарного дерева пошуку і дасть нам можливість його збудувати:



Бачимо, що знадобилося лише 3 вкладених циклів, кожен з яких працює за час O(n). Отже, загальна складність алгоритму — O(n^3).

В результаті отримуємо таблицю цін e і таблицю root. В таблиці е значення e[1, n] дасть нам ціну всього нашого оптимального дерева. При цьому root зберігає індекси коренів піддерев: root[i, j] містить індекс піддерева з елементами ki, …, kj.

Отже, побудувати за допомогою цієї таблиці дерево можна за допомогою простого рекурсивного алгоритму (2):

 Кожен виклик цієї функції будує піддерево, в корені якого один з елементів ki, …, kj, отже загалом вона викликається n разів.

**Отже, побудувати оптимальне бінарне дерево пошуку за допомогою динамічного програмування можна за час O(n^3), з використанням пам’яті O(n^2) для таблиць.**

**Основні модулі програми**

**Мова програмування: C++14.**

У своїй програмі я розбив модулі на дві категорії: ті, що стосуються саме предметної області (HumanResources.\*) і ті, що стосуються оптимального бін. дерева пошуку (OptimalBST.hpp)

OptimalBST<T> реалізовано як template-клас, тобто він може зберігати обєкти будь-якого класу. Єдина умова: для класу Т мають бути реалізовані оператори порівняння (>. ≥, <. ≤), за допомогою яких будуть сортуватися елементи.

Основна операція над деревом — **створення дерева, маючи вектори зі значеннями k, p, q.** Для цього маємо конструктор OptimalBST<T>::OptimalBST(const std::vector<T>& values, const std::vector<float>& freqs, const std::vector<float>& auxFreqs)

де freqs – масив вірогідностей p, auxFreqs – масив вірогідностей для фіктивних елемнтів q.

Конструктор виконує алгоритм (1), а потім викликає функцію

BSTNode<T>\* OptimalBST<T>::buildBST(size\_t i, size\_t j, size\_t\* roots, const std::vector<T>& values)

яка виконує алгоритм (2) і будує власне бінарне дерево.

Також маємо функцію пошуку елемента у BSTNode:

bool OptimalBST<T>::contains(const T& value)

**Інтерфейс користувача, тестові приклади**

У програмі є інтерактивний режим, який дозволяє користувачеві ввести імена працівників values, і вірогідності freqs, auxFreqs. Програма виводить структуру оптимального дерева.

Enter employee name (or "stop"): **Yurii**

Enter employee name (or "stop"): **Carl**

Enter employee name (or "stop"): **Alice**

Enter employee name (or "stop"): **Bob**

Enter employee name (or "stop"): **Zzz**

Enter employee name (or "stop"): **stop**

Enter cost of employees before Alice: **100**

Enter cost for Alice: **50**

Enter cost for employees between Alice and Bob: **3**

Enter cost for Bob: **10**

Enter cost for employees between Bob and Carl: **4**

Enter cost for Carl: **20**

Enter cost for employees between Carl and Yurii: **15**

Enter cost for Yurii: **10000**

Enter cost for employees between Yurii and Zzz: **90**

Enter cost for Zzz: **100**

Enter cost of employees after Zzz: **150**

Creating tree...

Optimal cost: 11346

(aux. values) - cost: 100

Alice - cost: 50

(aux. values) - cost: 3

Bob - cost: 10

(aux. values) - cost: 4

Carl - cost: 20

(aux. values) - cost: 15

Yurii - cost: 10000

(aux. values) - cost: 90

Zzz - cost: 100

(aux. values) - cost: 150

== Tree ==

{ Name: Yurii }

Left:

{ Name: Alice }

Right:

{ Name: Carl }

Left:

{ Name: Bob }

Right:

{ Name: Zzz }

Available commands: create, print, find-by-name, help, exit

> **create**

Deleting tree...

Enter employee name (or "stop"): **Alice**

Enter employee name (or "stop"): **Bob**

Enter employee name (or "stop"): **Carl**

Enter employee name (or "stop"): **stop**

Enter cost of employees before Alice: **0**

Enter cost for Alice: **0.5**

Enter cost for employees between Alice and Bob: **0**

Enter cost for Bob: **0.1**

Enter cost for employees between Bob and Carl: **0**

Enter cost for Carl: **0.4**

Enter cost of employees after Carl: **0**

Creating tree...

Optimal cost: 1.6

(aux. values) - cost: 0

Alice - cost: 0.5

(aux. values) - cost: 0

Bob - cost: 0.1

(aux. values) - cost: 0

Carl - cost: 0.4

(aux. values) - cost: 0

== Tree ==

{ Name: Alice }

Right:

{ Name: Carl }

Left:

{ Name: Bob }

> **exit**

**Висновки**

Ми дослідили основи динамічного програмування і застосували їх для побудови оптимального бінарного дерева пошуку. Ця структура дозволяє виконувати пошук у відсортованій множині даних з оптимальною швидкістю, якщо ми знаємо вірогідність пошуку певних елементів.

Бачимо, що за рахунок дин. програмування можна позбутися зайвих обчислень даних, які можуть з’явитися при використанні рекурсії.

**Список використаних джерел**

1. Т.Кормен, Ч.Лейзерсон, Р.Ривест, К.Штайн. - Алгоритмы. Построение и анализ. - 2013, с.431-438