Київський національний університет імені Тараса Шевченка

Факультет комп`ютерних наук та кібернетики

Кафедра інтелектуальних програмних систем

Алгоритми та складність

Завдання №4

“Оптимальне дерево, динамічне програмування”

Виконав студент 2-го курсу

Групи К-29

Демиденко Андрій Геннадійович

2021

**Предметна область**

Предметна область: Відділ кадрів

Об’єкти: Підрозділи, Співробітники

Примітка: Існує множина підрозділів. Кожен підрозділ включає в себе множину співробітників.

**Завдання**

Реалізувати оптимальне дерево пошуку. У вузлах будуть зберігатись інформація про підрозділи.

**Теорія**

Нехай задана послідовність різних між собою ключів

Така що із цих ключів можна побудувати бінарне дерево пошуку. Для кожного ключа задана ймовірність пошуку цього ключа. Крім цього може виконуватись пошук значень відсутніх в послідовності тому треба передбачити n+1 фіктивних ключів для кожного фіктивного ключа передбачена ймовірність пошуку . Фіктивний ключ представляє всі значення які знаходяться між ключами

Для всіх ймовірностей ключів послідовності і фіктивних ключів справедлива рівність

Оскільки відомі всі ймовірності, то можна знайти математичне сподівання пошуку у бінарному дереві T.

де - глибина вузла в дереві Т.

Отже бінарне дерево пошуку, математичне сподівання пошуку якого є мінімальним - називається оптимальним.

Так як дерево є оптимальним, то і кожне його під дерево є оптимальним.

Визначимо величину як математичне сподівання пошуку в оптимальному дереві пошуку з ключами .

та суму ймовірностей

Таким чином, якщо корінь оптимального під дерева, яке містить ключі то виконується співвідношення:

Останнє рекурсивне співвідношення передбачає, що нам відомо який вузол використовується в якості кореня. На цю роль береться ключ, який приведе до мінімального математичного сподівання пошуку. Враховуючи все це отримуємо остаточну рекурсивну формулу:

**Алгоритм**

1. Обраховування ймовірностей ключів послідовності та фіктивних ключів
2. Рахуємо допоміжну матрицю за якою ми будемо будувати оптимальне дерево. Для цього будуємо дві матриці

Для всіх елементів INT\_MAX:

Всі елементи ми рекурсивно обчислюємо за формулою:

Далі знаходимо елементи матриці

Тобто ми шукаємо всі такі при якому

та мінімальний

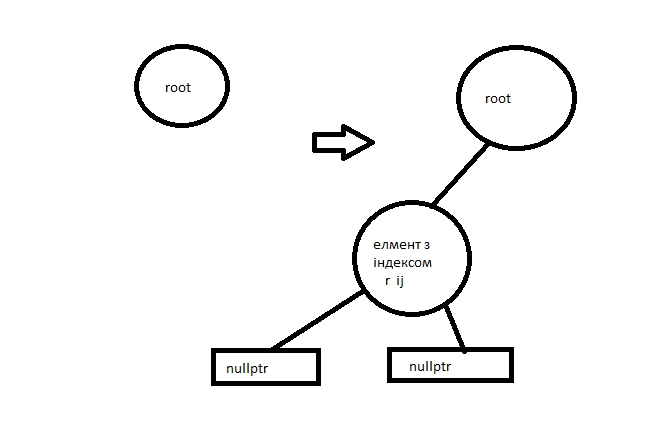
*.*

1. Спочатку шукаємо корінь, для цього шукаємо елемент з індексами з матриці r. Значення цього елементу це номер ключа в послідовності, який стане коренем оптимального дерева. Далі шукаємо дочірні лівий та правий елементи для кореня. Для цього будуємо нові підпослідовності з межами:

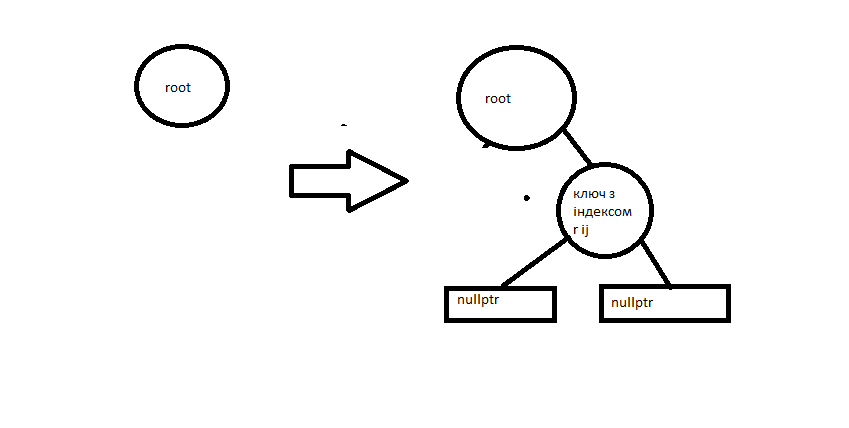
- межі для вибору елементу лівого сина свого батьківського елементу;

- межі для вибору елементу правого сина;

Де елемент матриці на n-му рядку і 1-му стовпці. Тоді якщо то ми зліва приєднуємо ключ, який має індекс у послідовності index= і після цього цьому лівому елементу додаємо нульові під дерева (nullptr).

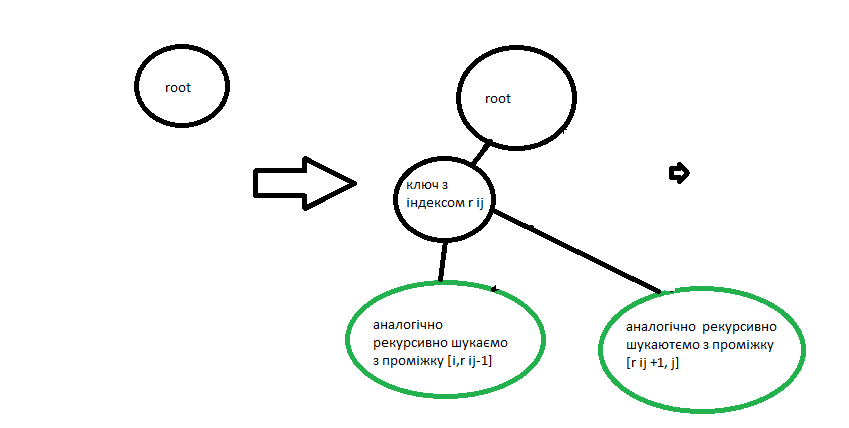


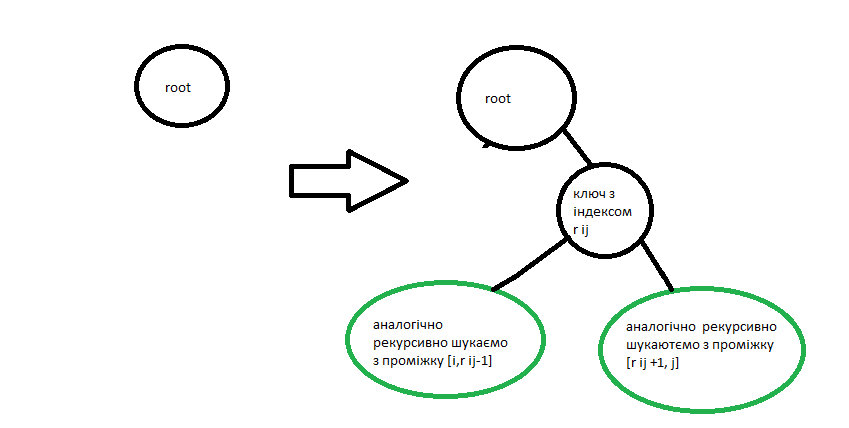
Аналогічно тоді коли



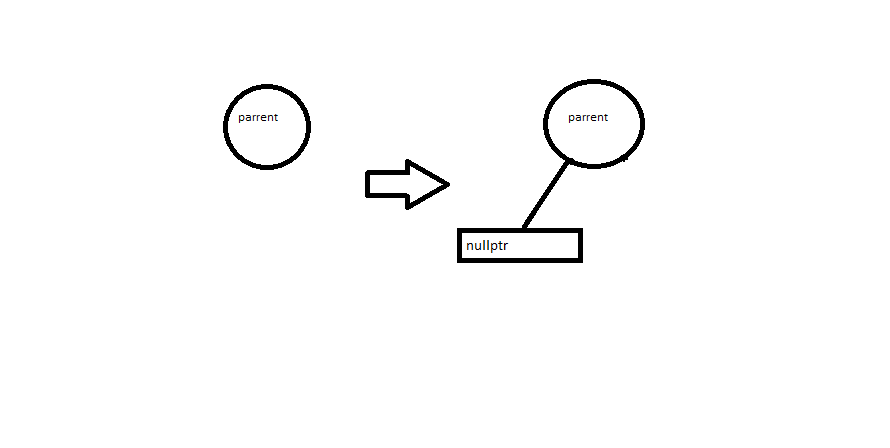
Якщо

Ми беремо елемент з матриці r з індексами і це буде індекс ключа в послідовності, який ми зробимо лівим елементом але на відміну від випадку, коли ми не додаємо цьому лівому елементу нульові елементи, а рекурсивно шукаємо лівий та правий елементи для цього лівого елементу, але вже з проміжків

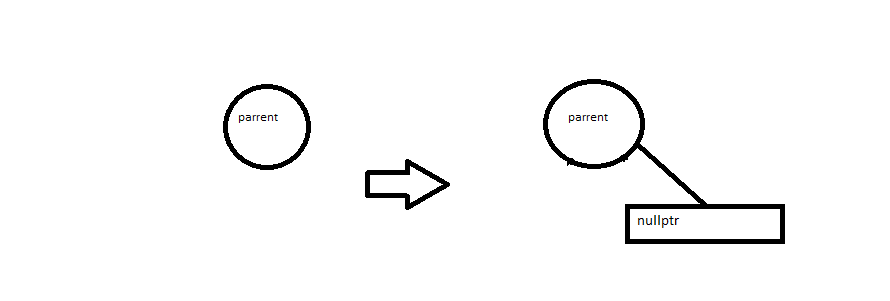


Аналогічно коли 

Якщо то ми просто додажмо нульовий елемент (nullptr) і перестаєм шукать дочірні елементи



Аналогічно коли



Для повного будування оптимального дерева пошуку потрібно виконувати 3 крок до того моменту поки всі і

**Складність**

У середньому складність звернень (пошуку,видалення) до дерева .Унайкращих випадках (це звернення до елемнтів які знаходяться біля корення) складність є асимптотично наближеною до константної складності O(1).

**Мова програмування**

С++

**Модулі програми**

//Структура для опису вузла оптимального бінарного дерева пошуку

template<typename T>

struct BSTNode

//Клас для опису оптимального бінарного дерева

template<typename T>

class OptimalBST

//Клас для опису співобітника

class Worker

//Клас для опису підрозділу

class Department

//Функція яка створює таблицю математичних сподівань

int\*\* optimalBSTRootMatrix(vector<double>& p, vector<double>& q)

//Функція яка створює дерево на основі матриці математичних сподівань

template<typename T>

BSTNode<T>\* CreateOptimalBST

(std::vector<T>& data, int\*\* rootMatrix, int indexI, int indexJ, BSTNode<T>\* parent)

//Функція яка знаходить корінь дерева

template<typename T>

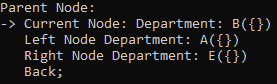
OptimalBST<T> optimalBSTCreateRoot

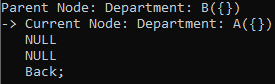
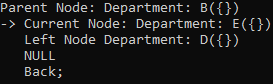
(vector<T>& data, vector<double>& probability, vector<double>& dummy\_probability)

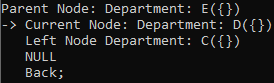
**Інтерфейс користувача**

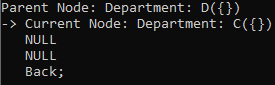
Вхідні дані генеруються програмою і виводяться в консоль

**Приклад виводу програми**

****

** **

****

****

**Тестовий приклад**

Нехай маємо вхідний масив підрозділів

[ {A {}}, {B {}}, {C {}}, {D {}}, {E {}}} ]

1 2 3 4 5

З яких потрібно скласти оптимальне дерево

Масив ймовірностей пошуку даних підрозділів:

p={15,10,5,10,20} (у відсотках)

Тепер визначимо фіктивні ймовірності

Оскільки фіктивна ймовірність це ймовірність звернення до елементу якого немає у масиві підрозділів, то візьмемо фіктивний ключ як елемент масиву підрозділів, назву якого увели не правильно і відповідно це коли увели порожню назву підрозділа.

Всього n+1=6 фіктивних ключів

Масив фіктивних ймовірностей

q={5,10,5,5,5,10}

Будуємо матрицю r

Шукаємо корінь на проміжку (1,5)

Беремо елемент матриці

[ {A {}}, {B {}}, {C {}}, {D {}}, {E {}}} ]

1 2 3 4 5

Отже корінь – (B {})

Тоді лівий дочірній елемент шукаємо проміжку (1,2-1)=(1,1)

B

Шукаємо серед [1,1]

ва

Лівий дочірній елемент :

[ {A {}} , {B {}}, {C {}}, {D {}}, {E {}}} ]

1 2 3 4 5

Дочірні елементи для (A {}) шукаються з проміжків:

Лівий (1,0) - тому просто додаємо nullptr.

Правий (2, 1) - тому теж nullptr.

B

Шукаємо серед [3,5]

ва

A

Правий дочірній елемент з (3,5) :

[ {A {}} , {B {}}, {C {}}, {D {}}, {E {}}} ]

1 2 3 4 5

Для (E {})

Лівий елемент шукаємо (3,5-1)=(3,4)

B

A

E

Шукаємо серед [3,4]

ва

[ {A {}} , {B {}}, {C {}}, {D {}}, {E {}}} ]

1 2 3 4 5

Для вузла (D {})

Лівий дочірній шукається з (3,3)

[ {A {}} , {B {}}, {C {}}, {D {}}, {E {}}} ]

1 2 3 4 5

При продовженні алгоритму всі вузли автоматично стануть nullptr

Остаточне дерево

B

A

E

D

C

**Висновки**

На практиці реалізували оптимальне дерево. До плюсів можна віднести високу швидкість алгоритмів звернення до елементів дерева та і його структуру за якої елементи які до яких майже не звертаються стоять на останніх рівнях дерева. До мінусів можна віднести те що побудова такого дерева вимагає більше даних про елементи ніж у інших бінарних дерев.

**Література**

* Кормен Алгоритми ,побудова і аналіз ст. 331
* <https://site.ada.edu.az/~medv/acm/Docs%20e-olimp/Volume%2016/1522.htm>