Київський національний університет імені Тараса Шевченка

Факультет комп’ютерних наук та кібернетики

Кафедра інтелектуальних інформацій систем

Алгоритми та складніcть

Лабораторна робота №2

Завдання №2 – Оптимальні дерева бінарного

пошуку (динамічне програмування)

Тип даних – комплексні числа

Виконав студент 2-го курсу

Групи ІПС-21

Ольховатий Ігор

2023

**Зміст**

[Теоретичні відомості 3](#_Toc130039300)

[Алгоритм 3](#_Toc130039301)

[Складність 5](#_Toc130039302)

[Мова програмування 5](#_Toc130039303)

[Модулі програми 5](#_Toc130039304)

[Інтерфейс користувача 7](#_Toc130039305)

[Демонстрація роботи 8](#_Toc130039306)

[Висновок 10](#_Toc130039307)

[Література 11](#_Toc130039308)

# **Теоретичні відомості**

# **Бінарне дерево пошуку** (BST), яке також називають упорядкованим або відсортованим бінарним деревом, є кореневою структурою даних бінарного дерева з ключем кожного внутрішнього вузла, більшим за всі ключі в лівому піддереві відповідного вузла та меншим за ті, що знаходяться в його правому піддереві. Часова складність операцій над бінарним деревом пошуку прямо пропорційна висоті дерева.

**Оптимальне двійкове дерево пошуку** (Optimal BST), яке іноді називають збалансованим бінарним деревом, є бінарним деревом пошуку, яке забезпечує найменший можливий час пошуку (або очікуваний час пошуку) для заданої послідовності доступу (або ймовірності доступу).

# **Алгоритм**

# Алгоритм Кнута для оптимальних дерев

# У своїй роботі 1970 року «Оптимальні бінарні дерева пошуку» Дональд Кнут пропонує метод пошуку оптимальне бінарне дерево пошуку із заданим набором значень і ймовірністю пошуку кожного значення і пошук значення не в дереві між кожними послідовними ключами. Цей метод спирається на динамічне програмування, щоб отримати час побудови . Хоча це повільно, цей метод є гарантовано сформує дерево з найменшою очікуваною глибиною пошуку.

1. Створюється таблиця, яка обчислює ймовірність кожного повного піддерева. Повне піддерево — це таке, яке містить усі значення між двома ключами всередині дерева.
2. Використовуючи цю таблицю, для кожного повного піддерева, додаємо загальну ймовірність піддерева до мінімального набору з двох піддерев, який існує для всіх можливих коренів. У цьому процесі також записується вибраний корінь.
3. Значення для цілого дерева — це очікувана глибина пошуку, а збережені корені можна використовувати для формування дерева.

# **Складність**

Складність в найгіршому випадку

Складність в середньому випадку

Загалом метод має часову складність , де *n* – кількість елементів у дереві, а за базову операцію вибрано порівняння двох елементів. Кнут показує прискорений метод, що призводить має часову складність . Це прискорення відбувається під час вибору кореня для кожного піддерева. Замість того, щоб дивитися на кожен можливий корінь, розглядаються лише корені між або рівними корен. з піддерева, де відсутні найменший і найбільший вузли. Це генерує телескопічний ефект, який дозволяє цій частині алгоритму потім в середньому мати часову складність *O(n)* для *n* піддерев*.* Саме наведене Кнутом рішення (з використанням підходів динамічного програмування) і було реалізовано в рамках даної лабораторної роботи.

# **Мова програмування**

Python

# **Модулі програми**

Використані програмні пакети (стандартні бібліотеки програм мовою Python):

import random - використання випадкових чисел

import math - використання математичних операцій

Функції, класи та їх методи:

class Complex:

Представляє комплексне число з дійсною та уявною складовою.

def \_\_init\_\_(self, real: int = 0, imag: int = 0, rand: bool = False) -> None:

Ініціалізує новий об’єкт типу Complex.

def \_\_hash\_\_(self) -> int:

Повертає хеш-значення для цього комплексного об’єкта.

def \_\_str\_\_(self) -> str:

*Повертає рядкове представлення цього комплексного об’єкта.*

def \_\_repr\_\_(self) -> str:

Повертає рядкове представлення цього об’єкта Complex, яке можна використовувати для відтворення об’єкта.

def \_\_add\_\_(self, other: Union[int, float, 'Complex']) -> 'Complex':

Додає заданий об’єкт до цього комплексного об’єкта та повертає новий комплексний об’єкт, що представляє суму.

def \_\_sub\_\_(self, other: Union[int, float, 'Complex']) -> 'Complex':

Віднімає заданий об’єкт із цього комплексного об’єкта та повертає новий складний об’єкт, що представляє різниця.

def \_\_mul\_\_(self, other: Union[int, float, 'Complex']) -> 'Complex':

Помножує заданий об’єкт на цей комплексний об’єкт і повертає новий комплексний об’єкт, що представляє добуток.

def \_\_truediv\_\_(self, other: Union[int, float, 'Complex']) -> 'Complex':

Ділить цей комплексний об’єкт на даний об’єкт і повертає новий комплексний об’єкт, що представляє приватне.

def \_\_abs\_\_(self) -> float:

Повертає величину комплексного числа.

def \_\_eq\_\_(self, other) -> bool:

Визначає, чи рівні два комплексні числа.

def \_\_lt\_\_(self, other) -> bool:

Визначає, чи комплексне число менше іншого комплексного числа.

def \_\_le\_\_(self, other) -> bool:

Визначає, чи комплексне число менше або дорівнює іншому комплексному числу.

def \_\_gt\_\_(self, other) -> bool:

Визначає, чи комплексне число більше за інше комплексне число.

def \_\_ge\_\_(self, other) -> bool:

Визначає, чи комплексне число більше або дорівнює іншому комплексному числу.

class BST:  
 def \_\_init\_\_(self, value: Optional[any] = None) -> None:

Створює новий вузол дерева з указаним значенням.

def insert(self, value: any) -> None:

Вставляє вказане значення в бінарне дерево пошуку.

def find(self, value: any) -> any:

Знаходить вузол, що містить указане значення, у бінарному дереві пошуку.

def find\_with\_depth(self, value: any, depth: int) -> tuple[bool, int]:

Знаходить вузол, що містить вказане значення, у бінарному дереві пошуку, і повертає кортеж логічного значення, що вказує, чи було знайдено значення, і глибину, на якій було знайдено значення.

def inorder\_traversal(self, func: Callable[[any], None]) -> None:

Переглядає бінарне дерево пошуку в порядку та виконує вказану функцію

на кожному вузлі.

def preorder\_traversal(self, func: Callable[[any], None]) -> None:

Проходить двійкове дерево пошуку в попередньому порядку та виконує вказану функцію на кожному вузлі.

def \_\_str\_\_(self, depth: int = 0) -> str:

Повертає рядкове представлення бінарного дерева пошуку в обертаному вигляді.

def display(self):

Відображає двійкову структуру дерева пошуку в горизонтальному порядку.

def \_display\_aux(self):

Допоміжна функція, яка повертає список рядків, ширину, висоту та горизонтальну координату кореня.

def find\_optimal\_tree\_ordering(beta\_list: list[float],  
 alpha\_list: list[float],  
 beta\_len: int) -> tuple[list[list[Optional[float]]], list[list[Optional[int]]]]:

Знаходить оптимальне впорядкування дерева за допомогою списку ваг і альфа.

def construct\_tree(root\_table: list) -> list:

Створює бінарне дерево за допомогою кореневої таблиці та повертає список його елементів у попередньому обході.

def construct\_tree\_inline(root\_table: list[list[int]], key\_list: list[any]) -> BST:

Будує бінарне дерево пошуку з кореневої таблиці та списку ключів.

def print\_table(table: list[list[int]]):

Роздруковує 2D таблицю.

def print\_value(x: any):

Роздруковує значення

def generate\_probs(data: list[any]) -> tuple[list[float], list[float], list[any]]:

Функція generate\_probs приймає список даних будь-якого типу та повертає кортеж із трьох списків. Функція спочатку підраховує кількість входжень кожного елемента даних у вхідному списку за допомогою методу Counter(). Потім він обчислює альфа- та бета-значення, необхідні для побудови оптимального бінарного дерева пошуку.

def test(data=None):

Тестування коректності роботи алгоритму та демонстрація результатів

# **Інтерфейс користувача**

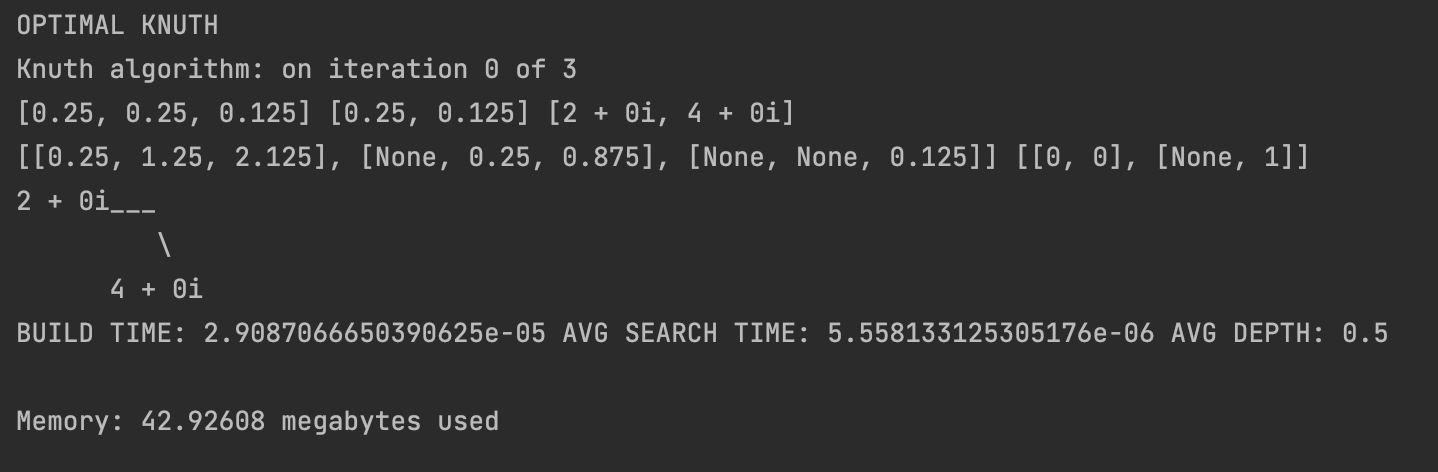
Інтерфейсом користувача для взаємодії з програмою є консоль.

# **Демонстрація роботи**

Вхідні дані:

test\_data = [Complex(1, 0), Complex(1, 0), Complex(2, 0), Complex(2, 0),  
 Complex(3, 0), Complex(3, 0), Complex(4, 0), Complex(5, 0)]

test(test\_data)

Результат роботи:

Опишемо, як саме працює програма. Ми передаємо конструктору класу Complex пари чисел, що позначають дійсну та уявну частину числа, а далі список таких комплексних чисел використовується в якості вхідних даних.

Далі наш набір даних спочатку підраховує кількість входжень кожного елемента даних у вхідному списку, потім він обчислює альфа та бета значення, необхідні для побудови оптимального бінарного дерева пошуку. Альфа представляють ймовірності пошуку елементів у кореневих вузлах оптимального бінарного дерева пошуку, тоді як бета представляють ймовірності пошуку елементів у некореневих вузлах. Іншими словами, альфа призначаються елементам, чия очікувана частота пошуку вища, ніж їх частота як дочірнього вузла іншого вузла, тоді як бета призначаються елементам, чия очікувана частота пошуку вища як дочірній для іншого вузла.

Беручи до уваги ймовірності пошуку кожного ключа, ми можемо побудувати дерево, яке мінімізує середній час пошуку ключа. Альфа та бета значення використовуються для обчислення очікуваного часу пошуку для кожного вузла, а оптимальне дерево будується шляхом мінімізації суми очікуваного часу пошуку для всіх вузлів у дереві.

Для цього вхідний набір даних сортується у порядку зростання, а далі значення імовірності з парним номером потрапляє до списку альфа значень, а значення з непарними номерами попадають до бета значень. У нашому випадку відсортований набір даних виглядає як [(1 + 0i, 2), (2 + 0i, 2), (3 + 0i, 2), (4 + 0i, 1), (5 + 0i, 1)], список альфа імовірностей це [0.25, 0.25, 0.125], список бета імовірностей це [0.25, 0.125], а значення бета [2 + 0i, 4 + 0i]. Далі запускається алгоритм пошуку оптимального впорядкування дерева. Цей алгоритм знаходить оптимальне впорядкування дерева за допомогою списку вагових коефіцієнтів і альфа. Він повертає два двохвимірні масиви. Перший масив містить оптимальні упорядкування дерев, а другий масив містить індекси коренів кожного піддерева в оптимальних упорядкуваннях дерев. Алгоритм працює шляхом ініціалізації трьох двовимірних масивів: exp\_table, weight\_table і root\_table. Потім він заповнює масиви exp\_table і weight\_table за допомогою динамічного програмування. Масив exp\_table містить очікувані значення піддерев. Масив weight\_table містить суму всіх ваг і альфа-факторів від кореня до поточного вузла. Масив root\_table містить індекси коренів кожного піддерева в оптимальному порядку дерев. Алгоритм повторює beta\_len + 1 крок, заповнюючи масиви exp\_table і weight\_table для кожного кроку. Він використовує вкладений цикл для обчислення оптимального значення для кожного піддерева. На кожному кроці він перевіряє, чи лівий індекс є одиничним від правого. Якщо так, то корінь лише один. В іншому випадку він скидається за допомогою прискорення, а правий індекс збільшується на 1, оскільки він включний. Потім він виконує ітерацію по кожному кореню між лівим і правим індексами та обчислює очікуване значення поточного піддерева. Якщо це очікуване значення менше поточного мінімального очікуваного значення, воно оновлює масиви exp\_table і root\_table. У нашому випадку exp\_table = [[0.25, 1.25, 2.125], [None, 0.25, 0.875], [None, None, 0.125]], а root\_table = [[0, 0], [None, 1]]. Маючи значення кореневої таблиці (root\_table) та значення у кореневих вузлах і будується дерево. У кореневому вузлі повинно бути значення бета з індексом 0, тобто 4 + 0i, а його правий син значення з індексом 1, тобто 4 + 0i. Очевидно, наш алгоритм знайшов оптимальну побудову дерева коректно.

# **Висновок**

# Даний алгоритм будує дерева з найкращою очікуваною глибиною пошуку, але це не обов'язково співвідноситься зі збалансованим деревом, це не гарантує дерево логарифмічної висоти в середньому, однак, це гарантує щонайбільше логарифмічний пошук, оскільки його очікувана глибина менша або дорівнює будь-якому іншому формуванню дерева. Оскільки це статичний метод, вставка за межі оригіналу побудова та видалення не допускаються, що досить великий мінус подібної структури даних.

# **Література**

1. <https://en.wikipedia.org/wiki/Optimal_binary_search_tree>
2. <https://en.wikipedia.org/wiki/Binary_search_tree>