**Міністерство освіти і науки України**

**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"**

**Факультет інформатики та обчислювальної техніки**

**Звіт**

з лабораторної роботи № 6

„ **Бінарні дерева пошуку**”

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-21 Скрипець Ольга Олександрівна*

**Перевірив**

(прізвище, ім'я, по батькові)

Київ 2023

**Зміст**

[1 Мета лабораторної роботи 3](#_Toc136935675)

[2 Завдання 4](#_Toc136935676)

[3 Виконання 5](#_Toc136935677)

[3.1 Аналіз часової складності 5](#_Toc136935678)

[3.2 Програмна реалізація алгоритму 6](#_Toc136935679)

[3.2.1 Вихідний код 6](#_Toc136935680)

[3.2.2 Приклади роботи 9](#_Toc136935681)

[3.3.3. Висновок 10](#_Toc136935682)

# Мета лабораторної роботи

Мета роботи – вивчити основні підходи аналізу обчислювальної складності алгоритмів пошуку оцінити їх ефективність на різних структурах даних.

# Завдання

У даній роботі необхідно виконати два завдання.

1. Перетворити вхідне бінарне дерево у бінарне дерево пошуку

На вхід подається деяке бінарне дерево, із фіксованою структурою (тобто зв’язками між вузлами, їх батьком та нащадками). Необхідно переписати значення вузлів дерева таким чином, щоби:

a) їх нові значення брались тільки з того набору, який присутній у вхідному дереві;

b) зберігалась внутрішня структура дерева (зв’язки між вузол-батько та вузол-нащадки).

2. Пошук сум послідовних вузлів в дереві

Після того, як вхідне дерево перетворене на бінарне дерево пошуку, необхідно розв’язати наступну задачу. Додатково задається деяке число S. В отриманому бінарному дереві пошуку необхідно знайти всі такі монотонні шляхи (які не обов'язково йдуть від кореня, але всі прямують згори вниз), що сума значень вузлів, які належать знайденим шляхам, дорівнює числу S.

# Виконання

## Аналіз часової складності

Алгоритм у даному коді використовує два основних алгоритми - сортування швидким методом (quick sort) і побудову бінарного дерева пошуку (binary search tree).

**Побудова бінарного дерева пошуку (BST):**

* Вставка в BST: О(log n) у середньому та кращому випадку. У випадку збалансованого BST, на кожному рівні вставки зменшується кількість можливих вузлів, до яких можна зробити вставку, вдвічі.
* Найгірший випадок: О(n), коли дерево є вигнутим ланцюжком. У такому випадку, вставка вимагає проходження через всі вузли дерева.

Отже, загальна часова складність даного коду залежить від двох основних алгоритмів:

* Складність сортування швидким методом: O(n log n)
* Складність побудови BST: в середньому O(n log n), в найгіршому випадку O(n^2)

У решті коду, основна операція полягає у проходженні по всіх вузлах дерева і записі шляхів з кореня до листя. Ця операція залежить від кількості вузлів і шляхів у дереві, тому її складність буде O(n \* m), де n - кількість вузлів у дереві, m - кількість шляхів.

Таким чином, загальна часова складність алгоритму у даному коді буде O(n^2) у найгіршому випадку для побудови BST, а також O(n \* m) для проходження по дереву та запису шляхів. Однак, у середньому випадку, коли BST будується збалансованим, складність буде O(n log n) для побудови BST та O(n \* m) для проходження по дереву та запису шляхів.

## Програмна реалізація алгоритму

### Вихідний код

class BST:

def \_\_init\_\_(self, val=None):

self.parent = None

self.left = None

self.right = None

self.val = val

def insert(self, val):

if not self.val:

self.val = val

return

if not self.left:

self.left = BST(val)

self.left.parent = self

return

if self.left.val != 0:

self.left.insert(val)

return

if not self.right:

self.right = BST(val)

self.right.parent = self

return

if self.right.val != 0:

self.right.insert(val)

return

temp = self.parent

while not temp.can\_insert():

temp = temp.parent

if not temp.right:

temp.right = BST(val)

temp.right.parent = self

return

temp.right.insert(val)

def can\_insert(self):

if self.val == 0:

return False

if not self.left or not self.right:

return True

if self.left.val == 0 and self.right.val == 0:

return False

if self.left.can\_insert() or self.right.can\_insert():

return True

return False

def inorder\_tree(self):

global inorder\_array

if self.left and self.left.val != 0:

self.left.inorder\_tree()

inorder\_array.append(self.val)

if self.right and self.right.val != 0:

self.right.inorder\_tree()

def rewrite\_tree(self):

global sorted\_array

if self.val != 0:

if self.left:

self.left.rewrite\_tree()

self.val = sorted\_array.pop(0)

if self.right:

self.right.rewrite\_tree()

def construct\_tree(array):

head = BST()

for element in array:

head.insert(element)

return head

def partition(array, low, high):

pivot = array[high]

i = low - 1

for j in range(low, high):

if array[j] <= pivot:

i = i + 1

(array[i], array[j]) = (array[j], array[i])

(array[i + 1], array[high]) = (array[high], array[i + 1])

return i + 1

def quick\_sort(array, low, high):

if low < high:

pi = partition(array, low, high)

quick\_sort(array, low, pi - 1)

quick\_sort(array, pi + 1, high)

def get\_paths\_util(curr\_node, sum\_so\_far, path):

if not curr\_node:

return

sum\_so\_far += curr\_node.val

path.append(curr\_node.val)

if sum\_so\_far == S:

paths.append(path[:])

if curr\_node.left and curr\_node.left.val != 0:

get\_paths\_util(curr\_node.left, sum\_so\_far, path)

if curr\_node.right and curr\_node.right.val != 0:

get\_paths\_util(curr\_node.right, sum\_so\_far, path)

path.pop(-1)

def get\_paths(node):

if not node:

return

path = []

get\_paths\_util(node, 0, path)

if node.left:

get\_paths(node.left)

if node.right:

get\_paths(node.right)

filename = input("Enter the name of the input file: ")

S = int(input("Enter the value of S: "))

with open(filename) as file:

data = file.read().split(" ")

array = [int(element) for element in data]

head = construct\_tree(array)

inorder\_array = []

head.inorder\_tree()

sorted\_array = inorder\_array[:]

quick\_sort(sorted\_array, 0, len(sorted\_array) - 1)

head.rewrite\_tree()

paths = []

get\_paths(head)

output\_filename = input("Enter the name of the output file: ")

with open(output\_filename, "w") as output\_file:

for line in paths:

output\_file.write(" ".join(str(node) for node in line) + '\n')

### Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

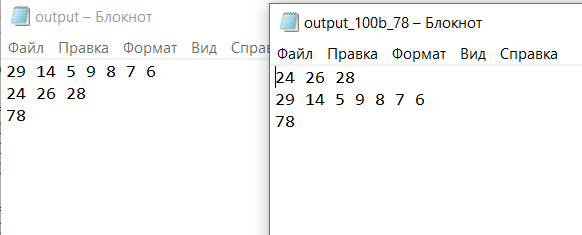


Рисунок 3.1 – Пошук елемента в масиві на 100 елементів

## Висновок

У цій лабораторній роботі було виконано два завдання, що стосуються бінарних дерев пошуку.

У першому завданні ми перетворили вхідне бінарне дерево на бінарне дерево пошуку. Це означає, що значення вузлів були перезаписані з наявного набору значень у вхідному дереві, зберігаючи при цьому внутрішню структуру дерева.

У другому завданні ми розв'язували задачу пошуку монотонних шляхів в перетвореному бінарному дереві пошуку. Нам було задано число S, яке визначало суму значень вузлів, що повинні міститись на знайдених шляхах. Ми знаходили всі шляхи, які відповідали умові, що сума значень на цих шляхах дорівнює числу S.

Загалом, ця лабораторна робота дозволила нам ознайомитись з бінарними деревами пошуку та виконати практичні завдання, пов'язані з перетворенням таких дерев та пошуком сум на послідовних вузлах.