

Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний
інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт

з лабораторної роботи № 3 з дисципліни
«Проектування алгоритмів»

„Проектування структур даних”

Виконав(ла)

ІП-12 Кушнір Ганна Вікторівна _____

(шифр, прізвище, ім'я, по батькові)

Перевірив

Сопов Олексій Олександрович _____

(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

1	МЕТА ЛАБОРАТОРНОЇ РОБОТИ	3
2	ЗАВДАННЯ	4
3	ВИКОНАННЯ	7
3.1	ПСЕВДОКОД АЛГОРИТМІВ	7
3.1.1	<i>Псевдокод алгоритму пошуку запису</i>	<i>7</i>
3.1.2	<i>Псевдокод алгоритму додавання запису</i>	<i>7</i>
3.1.3	<i>Псевдокод алгоритму видалення запису</i>	<i>8</i>
3.1.4	<i>Псевдокод алгоритму редагування запису</i>	<i>9</i>
3.2	ЧАСОВА СКЛАДНІСТЬ ПОШУКУ	9
3.3	ПРОГРАМНА РЕАЛІЗАЦІЯ	10
3.3.1	<i>Вихідний код</i>	<i>10</i>
3.3.2	<i>Приклади роботи</i>	<i>22</i>
3.4	ТЕСТУВАННЯ АЛГОРИТМУ	23
3.4.1	<i>Часові характеристики оцінювання</i>	<i>23</i>
	ВИСНОВОК	24
	КРИТЕРІЇ ОЦІНЮВАННЯ	25

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мета роботи – вивчити основні підходи проектування та обробки складних структур даних.

2 ЗАВДАННЯ

Відповідно до варіанту (таблиця 2.1), записати алгоритми пошуку, додавання, видалення і редагування запису в структурі даних за допомогою псевдокоду (чи іншого способу по вибору).

Записати часову складність пошуку в структурі в асимптотичних оцінках.

Виконати програмну реалізацію невеликої СУБД з графічним (не консольним) інтерфейсом користувача (дані БД мають зберігатися на ПЗП), з функціями пошуку (алгоритм пошуку у вузлі структури згідно варіанту таблиця 2.1, за необхідності), додавання, видалення та редагування записів (запис складається із ключа і даних, ключі унікальні і цілочисельні, даних може бути декілька полів для одного ключа, але достатньо одного рядка фіксованої довжини). Для зберігання даних використовувати структуру даних згідно варіанту (таблиця 2.1).

Заповнити базу випадковими значеннями до 10000 і зафіксувати середнє (із 10-15 пошуків) число порівнянь для знаходження запису по ключу.

Зробити висновок з лабораторної роботи.

Таблиця 2.1 – Варіанти алгоритмів

№	Структура даних
1	Файли з щільним індексом з перебудовою індексної області, бінарний пошук
2	Файли з щільним індексом з областю переповнення, бінарний пошук
3	Файли з не щільним індексом з перебудовою індексної області, бінарний пошук
4	Файли з не щільним індексом з областю переповнення, бінарний пошук
5	АВЛ-дерево
6	Червоно-чорне дерево
7	В-дерево $t=10$, бінарний пошук
8	В-дерево $t=25$, бінарний пошук

9	В-дерево $t=50$, бінарний пошук
10	В-дерево $t=100$, бінарний пошук
11	Файли з щільним індексом з перебудовою індексної області, однорідний бінарний пошук
12	Файли з щільним індексом з областю переповнення, однорідний бінарний пошук
13	Файли з не щільним індексом з перебудовою індексної області, однорідний бінарний пошук
14	Файли з не щільним індексом з областю переповнення, однорідний бінарний пошук
15	АВЛ-дерево
16	Червоно-чорне дерево
17	В-дерево $t=10$, однорідний бінарний пошук
18	В-дерево $t=25$, однорідний бінарний пошук
19	В-дерево $t=50$, однорідний бінарний пошук
20	В-дерево $t=100$, однорідний бінарний пошук
21	Файли з щільним індексом з перебудовою індексної області, метод Шарра
22	Файли з щільним індексом з областю переповнення, метод Шарра
23	Файли з не щільним індексом з перебудовою індексної області, метод Шарра
24	Файли з не щільним індексом з областю переповнення, метод Шарра
25	АВЛ-дерево
26	Червоно-чорне дерево
27	В-дерево $t=10$, метод Шарра
28	В-дерево $t=25$, метод Шарра
29	В-дерево $t=50$, метод Шарра
30	В-дерево $t=100$, метод Шарра
31	АВЛ-дерево
32	Червоно-чорне дерево

33	В-дерево $t=250$, бінарний пошук
34	В-дерево $t=250$, однорідний бінарний пошук
35	В-дерево $t=250$, метод Шарра

3.1 Псевдокод алгоритмів

3.1.1 Псевдокод алгоритму пошуку запису

Find(key, root)

```

if root is None
    then return None
end if
if key < root.key
    then return Find(key, root.left_child)
end if
if key > root.key
    then return Find(key, root.right_child)
end if
return root

```

3.1.2 Псевдокод алгоритму додавання запису

Insert(key, value, root)

```

if root is None
    then root ← new Node with key and value
    else if key < root.key
        then root.left ← Insert(key, value, root.left)
        if height(root.left) - height(root.right) = 2
            then if key < root.left.key
                then root ← RightRotate(root)
                else root ← LeftRightRotate(root)
            end if
        end if
    else if key > root.key
        then root.right ← Insert(key, value, root.right)
        if height(root.left) - height(root.right) = -2
            then if key > root.right.key
                then root ← LeftRotate(root)
                else root ← RightLeftRotate(root)
            end if
        end if
    end if
end if
root.height ← max(height(root.right), height(root.left)) + 1
return root

```

3.1.3 Псевдокод алгоритму видалення запису

Delete(key, root)

```
if root is None
    then return None
end if
if key < root.key
    then root.left ← Delete(key, root.left)
    else if key > root.key
        then root.right ← Delete(key, root.right)
        else if root does not have children
            then return None
        end if
        if root has 1 child
            then if root has left child
                then return root.left
                else return root.right
            end if
        end if
        successor ← min node of root's right subtree
        swap successor.key and root.key
        swap successor.value and root.value
        root.right ← Delete(successor.key, root.right)
        return root
    end if
end if
root.height ← max(height(root.right), height(root.left)) + 1
return RebalanceNode(root)
```

RebalanceNode(node)

```
if height(node.left) - height(node.right) = 2
    then if height(node.left.left) > height(node.left.right)
        then return RightRotate(node)
        else return LeftRightRotate(node)
    end if
end if
if height(node.left) - height(node.right) = -2
    then if height(node.right.right) > height(node.right.left)
        then return LeftRotate(node)
        else return RightLeftRotate(node)
    end if
end if
return node
```


3.1.4 Псевдокод алгоритму редагування запису

Change(key, new_value, root)

```
if root is None
    then flag ← False
    else if key < root.key
        then root.left, flag ← Change(key, new_value, root.left)
        else if key > root.key
            then root.right, flag ← Change(key, new_value, root.right)
            else root.value ← new_value
                flag ← True
        end if
    end if
end if
return root, flag
```

3.2 Часова складність пошуку

Find(key, root)	Складність
1 if root is None	O(1)
2 then return None	O(1)
end if	
3 if key < root.key	O(1)
4 then return Find(key, root.left_child)	O(log ₂ n)
end if	
5 if key > root.key	O(1)
6 then return Find(key, root.right_child)	O(log ₂ n)
end if	
7 return root	O(1)

Алгоритм пошуку в АВЛ-дереві – рекурсивний. Максимальна кількість його викликів рівна максимальній висоті бінарного дерева, тобто $\log_2 n$, де n – кількість вузлів дерева (ключів бази даних). Оскільки виконання всіх елементарних операцій (порівняння, присвоєння) використовує сталий час ($O(1)$), то загальна часова складність алгоритму у найгіршому випадку складає $O(\log_2 n) = O(\log n)$ – логарифмічна складність.

Мінімальна кількість викликів рекурсивного алгоритму пошуку рівна одиниці (якщо шуканий вузол є коренем дерева). Звідси часова складність алгоритму у найкращому випадку складає $\Omega(1)$ – константа.

У середньому випадку часова складність також є логарифмічною – $\Theta(\log n)$.

3.3 Програмна реалізація

3.3.1 Вихідний код

Файл «AVL_tree.py»:

```
class Node:
    def __init__(self, key: int, value: str, parent = None, left = None, right
= None, height: int = 0):
        self.key = key
        self.value = value
        self.parent = parent
        self.left = left
        self.right = right
        self.height = height

class AVLTree:
    def __init__(self):
        self.root = None

    def find(self, key: int):
        return self._find(key, self.root)

    def _find(self, key: int, node: Node):
        if not node:
            return None
        if key < node.key:
            return self._find(key, node.left)
        if key > node.key:
            return self._find(key, node.right)
        return node

    def height(self, node: Node):
        if not node:
            return -1
        return node.height

    def _right_rotate(self, node: Node):
        temp = node.left
        node.left = temp.right
        temp.right = node
```

```

temp.parent = node.parent
node.parent = temp
if node.left:
    node.left.parent = node
node.height = max(self.height(node.right), self.height(node.left)) + 1
temp.height = max(self.height(temp.left), node.height) + 1
return temp

def _left_rotate(self, node: Node):
    temp = node.right
    node.right = temp.left
    temp.left = node
    temp.parent = node.parent
    node.parent = temp
    if node.right:
        node.right.parent = node
    node.height = max(self.height(node.right), self.height(node.left)) + 1
    temp.height = max(self.height(temp.right), node.height) + 1
    return temp

def _right_left_rotate(self, node: Node):
    node.right = self._right_rotate(node.right)
    return self._left_rotate(node)

def _left_right_rotate(self, node: Node):
    node.left = self._left_rotate(node.left)
    return self._right_rotate(node)

def insert(self, key: int, value):
    self.root, flag = self._insert(key, value, self.root, None)
    return flag

def _insert(self, key: int, value, node: Node, parent: Node):
    if not node:
        node = Node(key, value, parent)
        flag = True
    elif key < node.key:
        node.left, flag = self._insert(key, value, node.left, node)
        if (self.height(node.left) - self.height(node.right)) == 2:
            if key < node.left.key:
                node = self._right_rotate(node)
            else:
                node = self._left_right_rotate(node)
    elif key > node.key:
        node.right, flag = self._insert(key, value, node.right, node)
        if (self.height(node.left) - self.height(node.right)) == -2:
            if key > node.right.key:
                node = self._left_rotate(node)

```

```

        else:
            node = self._right_left_rotate(node)
    else:
        flag = False
        node.height = max(self.height(node.right), self.height(node.left)) + 1
    return node, flag

def _find_min(self, node: Node):
    if node.left:
        return self._find_min(node.left)
    return node

def delete(self, key: int):
    if self.find(key):
        self.root = self._delete(key, self.root)

def _delete(self, key: int, node: Node):
    if not node:
        return None
    if key < node.key:
        node.left = self._delete(key, node.left)
    elif key > node.key:
        node.right = self._delete(key, node.right)
    else:
        if not node.left and not node.right:
            return None
        if not node.left or not node.right:
            if node.left:
                node.left.parent = node.parent
                return node.left
            else:
                node.right.parent = node.parent
                return node.right
        successor = self._find_min(node.right)
        node.key, successor.key = successor.key, node.key
        node.value, successor.value = successor.value, node.value
        node.right = self._delete(successor.key, node.right)
        return node
    node.height = max(self.height(node.left), self.height(node.right)) + 1
    return self._rebalance_node(node)

def _rebalance_node(self, node: Node):
    if (self.height(node.left) - self.height(node.right)) == 2:
        if self.height(node.left.left) > self.height(node.left.right):
            return self._right_rotate(node)
        else:
            return self._left_right_rotate(node)
    if (self.height(node.left) - self.height(node.right)) == -2:

```

```

        if self.height(node.right.right) > self.height(node.right.left):
            return self._left_rotate(node)
        else:
            return self._right_left_rotate(node)
    return node

def change(self, key: int, new_value):
    self.root, flag = self._change(key, new_value, self.root)
    return flag

def _change(self, key: int, new_value, node: Node):
    if not node:
        flag = False
    elif key < node.key:
        node.left, flag = self._change(key, new_value, node.left)
    elif key > node.key:
        node.right, flag = self._change(key, new_value, node.right)
    else:
        node.value = new_value
        flag = True
    return node, flag

def print_tree(self):
    return self._print_tree(self.root, '')

def _print_tree(self, node: Node, out: str, prefix: str = '', root: bool
= True, last: bool = True):
    out += prefix
    if root: out += ''
    elif last: out += '  L'
    else: out += '  |-'
    if node: out += str(node.key) + '\n'
    else: out += '\n'
    if not node or (not node.left and not node.right):
        return out
    if root: prefix += ''
    elif last: prefix += '  '
    else: prefix += '  | '
    if node.right:
        out = self._print_tree(node.left, out, prefix, False, False)
    else:
        out = self._print_tree(node.left, out, prefix, False, True)
    if node.right:
        out = self._print_tree(node.right, out, prefix, False, True)
    return out

```

Файл «main.py»:

```
from tkinter import *
import tkinter.messagebox
from AVL_tree import *
import os.path

filename = 'database.txt'

def main_window():
    root.deiconify()
    btn_find = Button(root, text = 'Find', font = 'Consolas 16', height = 3,
bg = 'lavender blush', command = find_data)
    btn_find.grid(row = 0, sticky = "EW")
    btn_add = Button(root, text = 'Add', font = 'Consolas 16', height = 3, bg
= 'lavender blush', command = add_data)
    btn_add.grid(row = 1, sticky = "EW")
    btn_edit = Button(root, text = 'Edit', font = 'Consolas 16', height = 3,
bg = 'lavender blush', command = edit_data)
    btn_edit.grid(row = 2, sticky = "EW")
    btn_delete = Button(root, text = 'Delete', font = 'Consolas 16', height =
3, bg = 'lavender blush', command = delete_data)
    btn_delete.grid(row = 3, sticky = "EW")
    btn_graphic = Button(root, text = 'Graphic representation\n of keys', font
= 'Consolas 16', height = 3, bg = 'lavender blush', command =
graphic_representation)
    btn_graphic.grid(row = 4, sticky = "EW")
    return

def find_data():
    child_find = Toplevel(root)
    root.withdraw()
    child_find.title('Find Data')
    child_find.geometry('500x300')
    child_find.resizable(0, 0)
    child_find['bg'] = 'lavender'
    child_find.columnconfigure(0, minsize = 500)
    lbl = Label(child_find, text = 'Input Key to Find:', font = 'Cambria 16',
bg = 'lavender')
    lbl.grid(row = 0, pady = 20)
    lbl_key = Label(child_find, text = 'Key', bg = 'lavender')
    lbl_key.grid(row = 1, padx = 5)
    ent_key = Entry(child_find, bg = 'lavender blush')
    ent_key.grid(row = 2, padx = 5, pady = 5)

    def find():
        key_str = ent_key.get()
```

```

        if not key_str.isnumeric():
            tkinter.messagebox.showinfo(title = 'Incorrect Key', message =
'The entered key must be a number.')
        else:
            key = int(key_str)
            node = tree.find(key)
            if not node:
                ent_content["text"] = ''
                tkinter.messagebox.showinfo(title = 'Search Failed', message
= 'The entered key was not found in the database.')
            else:
                ent_content["text"] = node.value
        return

    find_btn = Button(child_find, text = 'Find Data', width = 20, bg =
'lavender blush', command = find)
    find_btn.grid(row = 3, padx = 10, pady = 30)
    lbl_content = Label(child_find, text = 'Content', bg = 'lavender')
    lbl_content.grid(row = 4, padx = 5)
    ent_content = Label(child_find, bg = 'lavender blush', width = 30)
    ent_content.grid(row = 5, padx = 5, pady = 5)

    def delete_child():
        child_find.destroy()
        root.deiconify()

    child_find.protocol("WM_DELETE_WINDOW", delete_child)
    return

def add_data():
    child_add = Toplevel(root)
    root.withdraw()
    child_add.title('Add Data')
    child_add.geometry('500x210')
    child_add.resizable(0, 0)
    child_add['bg'] = 'lavender'
    child_add.columnconfigure([0, 1], minsize = 250)
    lbl = Label(child_add, text = 'Input new Key and Data:', font = 'Cambria
16', bg = 'lavender')
    lbl.grid(row = 0, column = 0, columnspan = 2, pady = 20)
    lbl_key = Label(child_add, text = 'Key', bg = 'lavender')
    lbl_key.grid(row = 1, column = 0, padx = 5)
    lbl_content = Label(child_add, text = 'Content', bg = 'lavender')
    lbl_content.grid(row = 1, column = 1, padx = 5)
    ent_key = Entry(child_add, bg = 'lavender blush')
    ent_key.grid(row = 2, column = 0, padx = 5, pady = 5)
    ent_content = Entry(child_add, bg = 'lavender blush', width = 30)

```

```

ent_content.grid(row = 2, column = 1, padx = 5, pady = 5)
def add():
    key_str = ent_key.get()
    if not key_str.isnumeric():
        tkinter.messagebox.showinfo(title = 'Incorrect Key', message =
'The entered key must be a number.')
    else:
        key = int(key_str)
        value = ent_content.get()
        flag = tree.insert(key, value)
        if not flag:
            tkinter.messagebox.showinfo(title = 'Error Adding', message =
'The entered key already exists in the database.')
            ent_key.delete(0, END)
            ent_content.delete(0, END)
        return

    add_btn = Button(child_add, text = 'Add Data', width = 20, bg = 'lavender
blush', command = add)
    add_btn.grid(row = 3, column = 0, columnspan = 2, padx = 10, pady = 30,
sticky = "E")

def delete_child():
    child_add.destroy()
    root.deiconify()

child_add.protocol("WM_DELETE_WINDOW", delete_child)
return

def edit_data():
    child_edit = Toplevel(root)
    root.withdraw()
    child_edit.title('Edit Data')
    child_edit.geometry('500x210')
    child_edit.resizable(0, 0)
    child_edit['bg'] = 'lavender'
    frm1 = Frame(child_edit, bg = 'lavender')
    frm1.columnconfigure([0, 1], minsize = 250)
    frm2 = Frame(child_edit, bg = 'lavender')
    frm2.columnconfigure([0, 1], minsize = 250)
    frm1.pack()
    lbl1 = Label(frm1, text = 'Input Key to Edit:', font = 'Cambria 16', bg =
'lavender')
    lbl1.grid(row = 0, columnspan = 2, pady = 20)
    lbl_key = Label(frm1, text = 'Key', bg = 'lavender')
    lbl_key.grid(row = 1, columnspan = 2, padx = 5)
    ent_key = Entry(frm1, bg = 'lavender blush')

```



```

ent_key.grid(row = 2, columnspan = 2, padx = 5, pady = 5)

def find():
    key_str = ent_key.get()
    if not key_str.isnumeric():
        tkinter.messagebox.showinfo(title = 'Incorrect Key', message =
'The entered key must be a number.')
    else:
        key = int(key_str)
        node = tree.find(key)
        if not node:
            tkinter.messagebox.showinfo(title = 'Search Failed', message
= 'The entered key was not found in the database.')
        else:
            frm1.pack_forget()
            frm2.pack()
            ent_key_change["text"] = node.key
            ent_content_change.insert(0, node.value)
    return

    find_to_edit_btn = Button(frm1, text = 'Find Data', width = 20, bg =
'lavender blush', command = find)
    find_to_edit_btn.grid(row = 3, columnspan = 2, padx = 10, pady = 30)
    lbl2 = Label(frm2, text = 'Change Key and/or Data:', font = 'Cambria 16',
bg = 'lavender')
    lbl2.grid(row = 0, columnspan = 2, pady = 20)
    lbl_key = Label(frm2, text = 'Key', bg = 'lavender')
    lbl_key.grid(row = 1, column = 0, padx = 5)
    lbl_content = Label(frm2, text = 'Content', bg = 'lavender')
    lbl_content.grid(row = 1, column = 1, padx = 5)
    ent_key_change = Label(frm2, bg = 'lavender blush', width = 20)
    ent_key_change.grid(row = 2, column = 0, padx = 5, pady = 5)
    ent_content_change = Entry(frm2, bg = 'lavender blush', width = 30)
    ent_content_change.grid(row = 2, column = 1, padx = 5, pady = 5)

def change():
    key = int(ent_key.get())
    new_value = ent_content_change.get()
    flag = tree.change(key, new_value)
    if not flag:
        tkinter.messagebox.showinfo(title = 'Search Failed', message =
'The entered key was not found in the database.')
        ent_key.delete(0, END)
        ent_content_change.delete(0, END)
        frm2.pack_forget()
        frm1.pack()
    return

```

```

def cancel():
    ent_key.delete(0, END)
    ent_content_change.delete(0, END)
    frm2.pack_forget()
    frm1.pack()
    return

    edit_btn = Button(frm2, text = 'Edit Data', width = 20, bg = 'lavender
blush', command = change)
    edit_btn.grid(row = 3, column = 0, padx = 10, pady = 30, sticky = "E")
    cancel_btn = Button(frm2, text = 'Cancel', width = 20, bg = 'lavender
blush', command = cancel)
    cancel_btn.grid(row = 3, column = 1, padx = 10, pady = 30, sticky = "W")

def delete_child():
    child_edit.destroy()
    root.deiconify()

child_edit.protocol("WM_DELETE_WINDOW", delete_child)
return

def delete_data():
    child_delete = Toplevel(root)
    root.withdraw()
    child_delete.title('Delete Data')
    child_delete.geometry('500x250')
    child_delete.resizable(0, 0)
    child_delete['bg'] = 'lavender'
    frm1 = Frame(child_delete, bg = 'lavender')
    frm1.columnconfigure([0, 1], minsize = 250)
    frm2 = Frame(child_delete, bg = 'lavender')
    frm2.columnconfigure([0, 1], minsize = 250)
    frm1.pack()
    lbl1 = Label(frm1, text = 'Input Key to Delete:', font = 'Cambria 16', bg
= 'lavender')
    lbl1.grid(row = 0, columnspan = 2, pady = 20)
    lbl_key = Label(frm1, text = 'Key', bg = 'lavender')
    lbl_key.grid(row = 1, columnspan = 2, padx = 5)
    ent_key = Entry(frm1, bg = 'lavender blush')
    ent_key.grid(row = 2, columnspan = 2, padx = 5, pady = 5)

def find():
    key_str = ent_key.get()
    if not key_str.isnumeric():
        tkinter.messagebox.showinfo(title = 'Incorrect Key', message =
'The entered key must be a number.')
    else:

```

```

        key = int(key_str)
        node = tree.find(key)
        if not node:
            tkinter.messagebox.showinfo(title = 'Search Failed', message
= 'The entered key was not found in the database.')
        else:
            frm1.pack_forget()
            frm2.pack()
            ent_key_delete["text"] = node.key
            ent_content_delete["text"] = node.value
    return

    find_to_del_btn = Button(frm1, text = 'Find Data', width = 20, bg =
'lavender blush', command = find)
    find_to_del_btn.grid(row = 3, columnspan = 2, padx = 10, pady = 30)
    lbl2 = Label(frm2, text = 'Your Data:', font = 'Cambria 18', bg =
'lavender')
    lbl2.grid(row = 0, columnspan = 2, pady = 20)
    lbl_key = Label(frm2, text = 'Key', bg = 'lavender')
    lbl_key.grid(row = 1, column = 0, padx = 5)
    lbl_content = Label(frm2, text = 'Content', bg = 'lavender')
    lbl_content.grid(row = 1, column = 1, padx = 5)
    ent_key_delete = Label(frm2, bg = 'lavender blush', width = 20)
    ent_key_delete.grid(row = 2, column = 0, padx = 5, pady = 5)
    ent_content_delete = Label(frm2, bg = 'lavender blush', width = 30)
    ent_content_delete.grid(row = 2, column = 1, padx = 5, pady = 5)

    def dont_delete():
        ent_key.delete(0, END)
        ent_key_delete["text"] = ''
        ent_content_delete["text"] = ''
        frm2.pack_forget()
        frm1.pack()
        return

    def delete():
        key = int(ent_key.get())
        ent_key.delete(0, END)
        ent_key_delete["text"] = ''
        ent_content_delete["text"] = ''
        tree.delete(key)
        frm2.pack_forget()
        frm1.pack()
        return

    lbl3 = Label(frm2, text = 'Are you sure you want to delete?', font =
'Cambria 14', bg = 'lavender')
    lbl3.grid(row = 3, columnspan = 2, pady = 10)

```

```

    yes_btn = Button(frm2, text = 'Yes', width = 10, bg = 'lavender blush',
command = delete)
    yes_btn.grid(row = 4, column = 0, padx = 10, pady = 5, sticky = "E")
    no_btn = Button(frm2, text = 'No', width = 10, bg = 'lavender blush',
command = dont_delete)
    no_btn.grid(row = 4, column = 1, padx = 10, pady = 5, sticky = "W")

    def delete_child():
        child_delete.destroy()
        root.deiconify()

    child_delete.protocol("WM_DELETE_WINDOW", delete_child)
    return

def graphic_representation():
    child_graphic = Toplevel(root)
    root.withdraw()
    child_graphic.title('Graphic Representation of Keys')
    child_graphic.geometry('400x700')
    child_graphic['bg'] = 'lavender'
    out_box = Label(child_graphic, bg = 'lavender')
    text = Text(out_box, font = 'Cambria 12', bg = 'lavender', relief = FLAT,
wrap = NONE)
    scroll_ver = Scrollbar(child_graphic, orient='vertical', command =
text.yview)
    scroll_ver.pack(side = RIGHT, fill = Y)
    scroll_hor = Scrollbar(child_graphic, orient='horizontal', command =
text.xview)
    scroll_hor.pack(side = BOTTOM, fill = X)
    text.config(yscrollcommand = scroll_ver.set, xscrollcommand =
scroll_hor.set)
    out_box.pack(fill = BOTH, expand = 1)
    text.pack(fill = BOTH, expand = 1)
    out = tree.print_tree()
    text.insert(END, '\n' + out)
    text.config(state = 'disabled')

    def delete_child():
        child_graphic.destroy()
        root.deiconify()

    child_graphic.protocol("WM_DELETE_WINDOW", delete_child)
    return

def read_file(file, parent: Node = None):
    key = int(file.readline())

```

```

value = file.readline()[:-1]
height = int(file.readline())
left_child = int(file.readline())
right_child = int(file.readline())
node = Node(key, value, parent, None, None, height)
if left_child: node.left = read_file(file, node)
if right_child: node.right = read_file(file, node)
return node

```

```

def write_file(file, node: Node):
    if not node:
        return
    file.write(str(node.key) + '\n')
    file.write(str(node.value) + '\n')
    file.write(str(node.height) + '\n')
    if node.left: file.write('1\n')
    else:         file.write('0\n')
    if node.right: file.write('1\n')
    else:         file.write('0\n')
    if node.left: write_file(file, node.left)
    if node.right: write_file(file, node.right)
    return

```

```

if __name__ == "__main__":
    root = Tk()
    root.title('Menu')
    root.minsize(width = 400, height = 200)
    root.resizable(0, 0)
    root['bg'] = 'lavender'
    root.columnconfigure(0, minsize = 400)
    main_window()
    tree = AVLTree()
    if os.path.isfile(filename):
        file = open(filename, 'rt')
        tree.root = read_file(file)
        file.close()

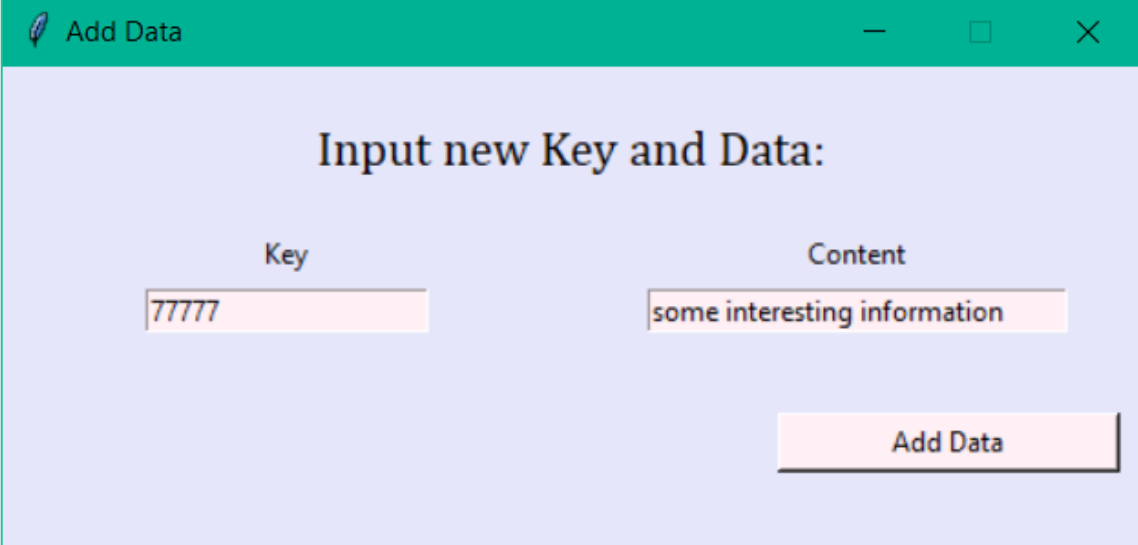
    def save_tree():
        file_write = open(filename, 'wt')
        write_file(file_write, tree.root)
        file_write.close()
        root.destroy()

    root.protocol("WM_DELETE_WINDOW", save_tree)
    root.mainloop()

```

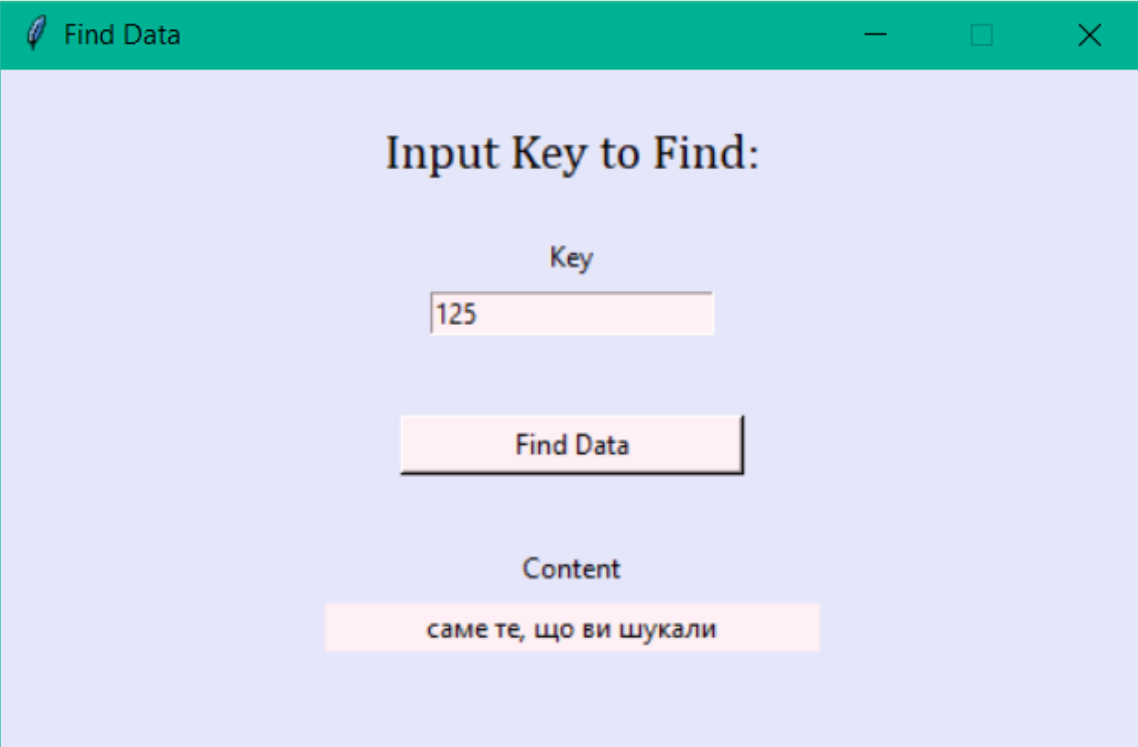
3.3.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми для додавання і пошуку запису.



The screenshot shows a window titled "Add Data" with a teal header bar. The main area has a light blue background and the text "Input new Key and Data:". Below this, there are two input fields. The first is labeled "Key" and contains the text "77777". The second is labeled "Content" and contains the text "some interesting information". At the bottom right, there is a button labeled "Add Data".

Рисунок 3.1 – Додавання запису



The screenshot shows a window titled "Find Data" with a teal header bar. The main area has a light blue background and the text "Input Key to Find:". Below this, there is an input field labeled "Key" containing the text "125". Below the input field is a button labeled "Find Data". At the bottom, there is a label "Content" and a text box containing the text "саме те, що ви шукали".

Рисунок 3.2 – Пошук запису

3.4 Тестування алгоритму

3.4.1 Часові характеристики оцінювання

В таблиці 3.1 наведено кількість порівнянь для 15 спроб пошуку запису по ключу.

Таблиця 3.1 – Число порівнянь при спробі пошуку запису по ключу

Номер спроби пошуку	Число порівнянь
1	13
2	14
3	13
4	12
5	14
6	5
7	11
8	10
9	12
10	10
11	13
12	13
13	11
14	13
15	14

За проведеними експериментами можна зробити висновок, що середнє число порівнянь при спробі пошуку запису по ключу в базі даних, що складається з 10000 записів, приблизно дорівнює 12.

ВИСНОВОК

В рамках даної лабораторної роботи було вивчено основні підходи проектування та обробки складних структур даних.

За допомогою псевдокоду було записано алгоритми пошуку, додавання, видалення і редагування запису в структурі даних «АВЛ-дерево». На основі побудованого алгоритму пошуку було визначено його часову складність в асимптотичних оцінках. Ця складність становила $\Theta(\log n)$ для середнього випадку.

Створені псевдокоди алгоритмів було покладено на мову програмування Python і виконано програмну реалізацію невеликої СУБД з графічним інтерфейсом користувача, дані в якій зберігаються у вигляді «АВЛ-дерева».

Було проведено ряд тестів, за результатами яких було отримано середнє число порівнянь при пошуку запису у структурі даних по ключу. Це число становило 12.

КРИТЕРІЇ ОЦІНЮВАННЯ

За умови здачі лабораторної роботи до 13.11.2022 включно максимальний бал дорівнює – 5. Після 13.11.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

- псевдокод алгоритму – 15%;
- аналіз часової складності – 5%;
- програмна реалізація алгоритму – 65%;
- тестування алгоритму – 10%;
- висновок – 5%.

+1 додатковий бал можна отримати за реалізацію графічного зображення структури ключів.