**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)**

(шифр, прізвище, ім'я, по батькові)

*ІП-12 Кушнір Ганна Вікторівна*

**Перевірив**

(прізвище, ім'я, по батькові)

*Сопов Олексій Олександрович*

Київ 2022

Зміст

[1 Мета лабораторної роботи 4](#_Toc119277786)

[2 Завдання 5](#_Toc119277787)

[3 Виконання 8](#_Toc119277788)

[3.1 Псевдокод алгоритмів 8](#_Toc119277789)

[3.1.1 Псевдокод алгоритму пошуку запису 8](#_Toc119277790)

[3.1.2 Псевдокод алгоритму додавання запису 8](#_Toc119277791)

[3.1.3 Псевдокод алгоритму видалення запису 9](#_Toc119277792)

[3.1.4 Псевдокод алгоритму редагування запису 10](#_Toc119277793)

[3.2 Часова складність пошуку 10](#_Toc119277794)

[3.3 Програмна реалізація 11](#_Toc119277795)

[3.3.1 Вихідний код 11](#_Toc119277796)

[3.3.2 Приклади роботи 23](#_Toc119277797)

[3.4 Тестування алгоритму 24](#_Toc119277798)

[3.4.1 Часові характеристики оцінювання 24](#_Toc119277799)

[Висновок 25](#_Toc119277800)

[Критерії оцінювання 26](#_Toc119277801)

# Мета лабораторної роботи

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

# 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 | B-дерево t=10, бінарний пошук |
| 8 | B-дерево t=25, бінарний пошук |
| 9 | B-дерево t=50, бінарний пошук |
| 10 | B-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 | B-дерево t=10, однорідний бінарний пошук |
| 18 | B-дерево t=25, однорідний бінарний пошук |
| 19 | B-дерево t=50, однорідний бінарний пошук |
| 20 | B-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 | B-дерево t=10, метод Шарра |
| 28 | B-дерево t=25, метод Шарра |
| 29 | B-дерево t=50, метод Шарра |
| 30 | B-дерево t=100, метод Шарра |
| 31 | АВЛ-дерево |
| 32 | Червоно-чорне дерево |
| 33 | B-дерево t=250, бінарний пошук |
| 34 | B-дерево t=250, однорідний бінарний пошук |
| 35 | B-дерево t=250, метод Шарра |

# Виконання

## Псевдокод алгоритмів

### 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

### 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

end if

root.height ← max(height(root.right), height(root.left)) + 1

return root

### 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

### 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

## 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(log2n) |
|  | end if |  |
| 5 | if key > root.key | O(1) |
| 6 | then return Find(key, root.right\_child) | O(log2n) |
|  | end if |  |
| 7 | return root | O(1) |

Алгоритм пошуку в АВЛ-дереві – рекурсивний. Максимальна кількість його викликів рівна максимальній висоті бінарного дерева, тобто log2n, 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(log2n) = O(logn) – логарифмічна складність.

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

У середньому випадку часова складність також є логарифмічною – Θ(logn).

## Програмна реалізація

### 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 += ' └─'

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()

### Приклади роботи

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

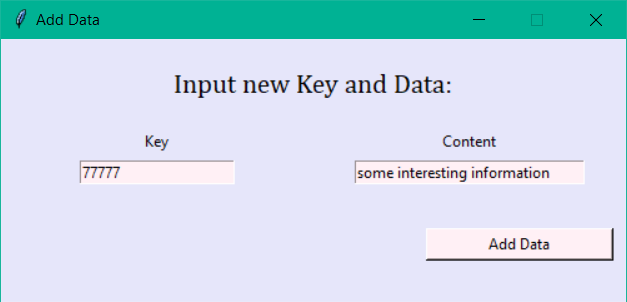


Рисунок 3.1 – Додавання запису

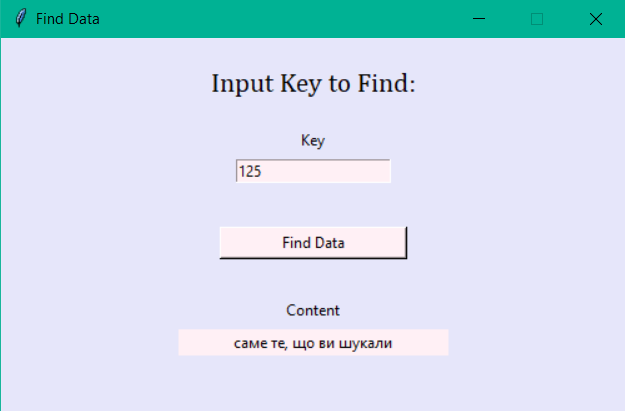


Рисунок 3.2 – Пошук запису

## Тестування алгоритму

### Часові характеристики оцінювання

В таблиці 3.1 наведено кількість порівнянь для 15 спроб 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 |

Таблиця 3.1 – Число порівнянь при спробі пошуку запису по ключу

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 Θ(logn) 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.