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-13 Лисенко Анастасія* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Головченко М.Н.* _____
(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

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	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

3 ВИКОНАННЯ

3.1 Псевдокод алгоритмів

FUNCTION insert_node(self, root, key, data):

```
IF not root
   THEN
   return TreeNode(key, data)
ELSE IF key < root.key
   THEN
   root.left<-self.insert_node(root.left, key, data)
ELSE IF key > root.key
    THEN
   root.right<-self.insert_node(root.right, key, data)
ENDIF
root.height<-1 + max(self.getHeight(root.left),
                     self.getHeight(root.right))

// Update the balance factor and balance the tree
balanceFactor<-self.getBalance(root)
IF balanceFactor > 1
   THEN
   IF key < root.left.key
      THEN
      return self.rightRotate(root)
   ELSE
      root.left<-self.leftRotate(root.left)
      return self.rightRotate(root)
   ENDIF
ENDIF
```

```
    IF balanceFactor < -1
        THEN
        IF key > root.right.key
            THEN
            return self.leftRotate(root)
        ELSE
            root.right<-self.rightRotate(root.right)
            return self.leftRotate(root)
        ENDIF
    ENDIF
    return root
END FUNCTION
```

FUNCTION delete_node(self, root, key):

```
    // Find the node to be deleted and remove it
    IF not root
        THEN
        return root
    ELSE IF key < root.key
        THEN
        root.left<-self.delete_node(root.left, key)
    ELSE IF key > root.key
        THEN
        root.right<-self.delete_node(root.right, key)
    ELSE
        IF root.left is None
            THEN
            temp<-root.right
            root<-None
            return temp
```

```
    ELSE IF root.right is None
        temp<-root.left
        root<-None
        return temp
    ENDIF
    temp<-self.getMinValueNode(root.right)
    root.key<-temp.key
    root.data<-temp.data
    root.right<-self.delete_node(root.right,
                                 temp.key)
IF root is None
    THEN
    return root
ENDIF
// Update the balance factor of nodes
root.height<-1 + max(self.getHeight(root.left),
                     self.getHeight(root.right))
balanceFactor<-self.getBalance(root)
// Balance the tree
IF balanceFactor > 1
    THEN
    IF self.getBalance(root.left) >= 0
        THEN
        return self.rightRotate(root)
    ELSE
        root.left<-self.leftRotate(root.left)
        return self.rightRotate(root)
    ENDIF
ENDIF
```

```
    IF balanceFactor < -1
       THEN
       IF self.getBalance(root.right) <= 0
          THEN
          return self.leftRotate(root)
       ELSE
          root.right<-self.rightRotate(root.right)
          return self.leftRotate(root)
       ENDIF
    ENDIF
    return root
ENDFUNCTION
FUNCTION  search(self, root, val):
    IF root is None
       THEN
       return False
    ENDIF
    ELSE IF root.key = val
       THEN
       return root.data
    ELSE IF root.key < val
       THEN
       return self.search(root.right, val)
    ELSE IF root.key > val
       THEN
       return self.search(root.left, val)
    END IF
    return False
ENDFUNCTION
```

FUNCTION edit(self, root, val, data):

IF root.key = val

THEN

root.data<-data

ENDIF

elif root.key < val: //You will need to change this to CASE OF

self.edit(root.right, val, data)

ELSE

self.edit(root.left, val, data)

ENDFUNCTION

3.2 Часова складність пошуку

Операції обертання (обертання вліво і вправо) займають постійний час, оскільки там змінюється лише кілька покажчиків. Оновлення висоти та отримання коефіцієнта балансу(balance factor) також займає постійний час. Таким чином, часова складність вставки вершини в АВЛ дерево залишається такою ж, як і вставки в Бінарне дерево, яка дорівнює O(h), де h є висотою дерева. Оскільки дерево АВЛ є збалансованим, висота дорівнює O(log(n)). Отже, часова складність вставки AVL дорівнює O(log(n)).

3.3 Програмна реалізація

3.3.1 Вихідний код

AVL.py

```
import collections
import sys

# Create a tree node
class TreeNode(object):
    def __init__(self, key, data):
        self.key = key
        self.data = data
        self.left = None
        self.right = None
        self.height = 1


class AVLTree(object):
```

```
    # Function to insert a node
    def insert_node(self, root, key, data):

        # Find the correct location and insert the node
        if not root:

            return TreeNode(key, data)
        elif key < root.key:
            root.left = self.insert_node(root.left, key, data)
        elif key > root.key:
            root.right = self.insert_node(root.right, key, data)

        root.height = 1 + max(self.getHeight(root.left),
                              self.getHeight(root.right))

        # Update the balance factor and balance the tree
        balanceFactor = self.getBalance(root)
        if balanceFactor > 1:
            if key < root.left.key:
                return self.rightRotate(root)
            else:
                root.left = self.leftRotate(root.left)
                return self.rightRotate(root)

        if balanceFactor < -1:
            if key > root.right.key:
                return self.leftRotate(root)
            else:
                root.right = self.rightRotate(root.right)
                return self.leftRotate(root)

        return root

    # Function to delete a node
    def delete_node(self, root, key):

        # Find the node to be deleted and remove it
        if not root:
            return root
        elif key < root.key:
            root.left = self.delete_node(root.left, key)
        elif key > root.key:
            root.right = self.delete_node(root.right, key)
        else:
            if root.left is None:
                temp = root.right
                root = None
                return temp
            elif root.right is None:
                temp = root.left
                root = None
                return temp
            temp = self.getMinValueNode(root.right)
            root.key = temp.key
            root.data = temp.data
            root.right = self.delete_node(root.right,
                                          temp.key)
        if root is None:
            return root

        # Update the balance factor of nodes
        root.height = 1 + max(self.getHeight(root.left),
                              self.getHeight(root.right))
```

```
        balanceFactor = self.getBalance(root)

        # Balance the tree
        if balanceFactor > 1:
            if self.getBalance(root.left) >= 0:
                return self.rightRotate(root)
            else:
                root.left = self.leftRotate(root.left)
                return self.rightRotate(root)
        if balanceFactor < -1:
            if self.getBalance(root.right) <= 0:
                return self.leftRotate(root)
            else:
                root.right = self.rightRotate(root.right)
                return self.leftRotate(root)
        return root

    # Function to perform left rotation
    def leftRotate(self, z):
        y = z.right
        T2 = y.left
        y.left = z
        z.right = T2
        z.height = 1 + max(self.getHeight(z.left),
                           self.getHeight(z.right))
        y.height = 1 + max(self.getHeight(y.left),
                           self.getHeight(y.right))
        return y

    # Function to perform right rotation
    def rightRotate(self, z):
        y = z.left
        T3 = y.right
        y.right = z
        z.left = T3
        z.height = 1 + max(self.getHeight(z.left),
                           self.getHeight(z.right))
        y.height = 1 + max(self.getHeight(y.left),
                           self.getHeight(y.right))
        return y

    # Get the height of the node
    def getHeight(self, root):
        if not root:
            return 0
        return root.height

    # Get balance factore of the node
    def getBalance(self, root):
        if not root:
            return 0
        return self.getHeight(root.left) - self.getHeight(root.right)

    def getMinValueNode(self, root):
        if root is None or root.left is None:
            return root
        return self.getMinValueNode(root.left)

    def search(self, root, val):
        if root is None:
            return False
        elif root.key == val:
            return root.data
```

```
        elif root.key < val:
            return self.search(root.right, val)
        elif root.key > val:
            return self.search(root.left, val)
        return False

    def isInTheTree(self, root, val):
        if root is None:
            return False
        elif root.key == val:
            return True
        elif root.key < val:
            return self.isInTheTree(root.right, val)
        elif root.key > val:
            return self.isInTheTree(root.left, val)
        return False

    def edit(self, root, val, data):
        # if root is None:
        if root.key == val:
            root.data = data
        elif root.key < val:
            self.edit(root.right, val, data)
        else:
            self.edit(root.left, val, data)

    def preOrder(self, root):
        if not root:
            return
        print("{0} ".format(root.key), end="")
        self.preOrder(root.left)
        self.preOrder(root.right)

    def serialize(self, root):
        components = []
        def incorporate(root, components):
            line = str(root.key) + ";" + str(root.data) + '\n'
            components.append(line)
            if root.left:
                components.append('L')
                incorporate(root.left, components)
            if root.right:
                components.append('R')
                incorporate(root.right, components)
            components.append('U')
            components.append('\n')
            return ''.join(components)

        incorporate(root, components)
        components.pop()
        return ''.join(components)

    def writeToFile(self, root, filename):
        serialized = self.serialize(root)
        myfile = open(filename, 'w')
        myfile.write(serialized)
        myfile.close()

    def inOrder(self, root):
        if not root:
            return
        self.inOrder(root.left)
        print("{0} ".format(root.key), end="")
        self.inOrder(root.right)
```

```
    # Print the tree
    def printHelper(self, currPtr, indent, last):
        if currPtr != None:
            print(indent, end="")
            if last:
                print("|----", end="")
                indent += "     "
            else:
                print("|----", end="")
                indent += "|    "
            print(currPtr.key, currPtr.data)
            self.printHelper(currPtr.left, indent, False)
            self.printHelper(currPtr.right, indent, True)

    def display(self, root):
        lines_prnt = ''
        lines, *_ = self._display_aux(root)
        for line in lines:
            lines_prnt += line +"\n"
        return lines_prnt

    def _display_aux(self, root):
        """Returns list of strings, width, height, and horizontal coordinate
of the root."""
        # No child.
        if root is None:
            return "", 0, 0, 0
        if root.right is None and root.left is None:
            line = '%s' % root.key
            width = len(line)
            height = 1
            middle = width // 2
            return [line], width, height, middle

        # Only left child.
        if root.right is None:
            lines, n, p, x = self._display_aux(root.left)
            s = '%s' % root.key
            u = len(s)
            first_line = (x + 1) * ' ' + (n - x - 1) * '_' + s
            second_line = x * ' ' + '/' + (n - x - 1 + u) * ' '
            shifted_lines = [line + u * ' ' for line in lines]
            return [first_line, second_line] + shifted_lines, n + u, p + 2, n
+ u // 2

        # Only right child.
        if root.left is None:
            lines, n, p, x = self._display_aux(root.right)
            s = '%s' % root.key
            u = len(s)
            first_line = s + x * '_' + (n - x) * ' '
            second_line = (u + x) * ' ' + '\\' + (n - x - 1) * ' '
            shifted_lines = [u * ' ' + line for line in lines]
            return [first_line, second_line] + shifted_lines, n + u, p + 2, u
// 2

        # Two children.
        left, n, p, x = self._display_aux(root.left)
        right, m, q, y = self._display_aux(root.right)
        s = '%s' % root.key
        u = len(s)
```

```
        first_line = (x + 1) * ' ' + (n - x - 1) * '_' + s + y * '_' + (m -
y) * ' '
        second_line = x * ' ' + '/' + (n - x - 1 + u + y) * ' ' + '\\' + (m -
y - 1) * ' '
        if p < q:
            left += [n * ' '] * (q - p)
        elif q < p:
            right += [m * ' '] * (p - q)
        zipped_lines = zip(left, right)
        lines = [first_line, second_line] + [a + u * ' ' + b for a, b in
zipped_lines]
        return lines, n + m + u, max(p, q) + 2, n + u // 2

    def levelOrderTraversal(self, root):
        ans = []

        # Return Null if the tree is empty
        if root is None:
            return ans

        # Initialize queue
        queue = collections.deque()
        queue.append(root)

        # Iterate over the queue until it's empty
        while queue:
            # Check the length of queue
            currSize = len(queue)
            currList = []

            while currSize > 0:
                # Dequeue element
                currNode = queue.popleft()
                currList.append(currNode.key)
                currSize -= 1

                # Check for left child
                if currNode.left is not None:
                    queue.append(currNode.left)
                # Check for right child
                if currNode.right is not None:
                    queue.append(currNode.right)

            # Append the currList to answer after each iteration
            ans.append(currList)

        # Return answer list
        return ans


def deserialize(filename):
    chars = ''
    nodes = []
    next_child = None
    f = open(filename, "r")
    for line in f:
        for char in line:
            if char not in ('L', 'R', 'U'):
                chars += char
                chars = chars.strip('\n')
            else:
                if not nodes:
                    if chars.find(";") != -1:
                        nodes.append(TreeNode(int(chars.split(";")[0]),
```

```
chars.split(";")[1]))
                elif next_child == 'left':
                    if chars.find(";") != -1:
                        nodes[-1].left = (TreeNode(int(chars.split(";")[0]),
chars.split(";")[1]))
                    nodes.append(nodes[-1].left)
                elif next_child == 'right':
                    if chars.find(";") != -1:
                        nodes[-1].right = (TreeNode(int(chars.split(";")[0]),
chars.split(";")[1]))
                    nodes.append(nodes[-1].right)
                elif next_child == 'up':
                    nodes.pop()
                if char == 'L':
                    next_child = 'left'
                elif char == 'R':
                    next_child = 'right'
                elif char == 'U':
                    next_child = 'up'
                chars = ''
    return nodes[0]
```

gui.py

```
from tkinter import *
from AVL import AVLTree, deserialize
import os

root = Tk()
root.title("AVL Tree Visualisation")
root.geometry("1080x720")
filename = "tree_serialization.txt"


def sel():
    if var.get() == 1:
        button.config(text="Insert", font=("Arial", 15), width=17, height=2)
        e1.delete(0, "end")
        e2.config(state=NORMAL)
        e2.delete(0, "end")
        message.config(state=NORMAL)
        message.delete("1.0", "end")
        message.config(state=DISABLED)

    elif var.get() == 2:
        button.config(text="Delete", font=("Arial", 15), width=17, height=2)
        e1.delete(0, "end")
        e2.config(state=NORMAL)
        e2.delete(0, "end")
        e2.config(state=DISABLED)
        message.config(state=NORMAL)
        message.delete("1.0", "end")
        message.config(state=DISABLED)

    elif var.get() == 3:
        button.config(text="Search", font=("Arial", 15), width=17, height=2)
        e1.delete(0, "end")
        e2.config(state=NORMAL)
        e2.delete(0, "end")
        e2.config(state=DISABLED)
```

```
        message.config(state=NORMAL)
        message.delete("1.0", "end")
        message.config(state=DISABLED)

    elif var.get() == 4:
        button.config(text="Edit", font=("Arial", 15), width=17, height=2)
        e1.delete(0, "end")
        e2.config(state=NORMAL)
        e2.delete(0, "end")
        message.config(state=NORMAL)
        message.delete("1.0", "end")
        message.config(state=DISABLED)


myTree = AVLTree()
treeRoot = None


def button_clicked():
    global myTree
    global treeRoot
    message.config(state=NORMAL)
    message.delete("1.0", "end")
    if var.get() == 1:
        if len(e1.get()) > 0:
            if e1.get().isdigit():
                if not myTree.isInTheTree(treeRoot, int(e1.get())):
                    if len(e2.get()) > 0:
                        key = int(e1.get())
                        data = e2.get()
                        treeRoot = myTree.insert_node(treeRoot, key, data)
                        myTree.writeToFile(treeRoot, filename)
                        message.delete("1.0", "end")
                        message.insert(INSERT, "node with key " + str(key) +
" was inserted")
                        message.config(state=DISABLED)
                    else:
                        message.delete("1.0", "end")
                        message.insert(INSERT, "enter the data")
                        message.config(state=DISABLED)
                else:
                    message.delete("1.0", "end")
                    message.insert(INSERT, "enter an unique key")
                    message.config(state=DISABLED)
            elif not e1.get().isdigit():
                message.delete("1.0", "end")
                message.insert(INSERT, "enter the digit for the key")
                message.config(state=DISABLED)
        else:
            message.delete("1.0", "end")
            message.insert(INSERT, "enter the key")
            message.config(state=DISABLED)

    elif var.get() == 2:
        if len(e1.get()) > 0:
            if e1.get().isdigit():
                key = int(e1.get())
                if myTree.isInTheTree(treeRoot, key):
                    treeRoot = myTree.delete_node(treeRoot, key)
                    myTree.writeToFile(treeRoot, filename)
                    message.delete("1.0", "end")
                    message.insert(INSERT, "node with key " + str(key) + "
was deleted")
                    message.config(state=DISABLED)
```

```
                else:
                    message.delete("1.0", "end")
                    message.insert(INSERT, "enter a key that exists in the
tree")
                    message.config(state=DISABLED)
            elif not e1.get().isdigit():
                message.delete("1.0", "end")
                message.insert(INSERT, "enter the digit for the key")
                message.config(state=DISABLED)
        else:
            message.delete("1.0", "end")
            message.insert(INSERT, "enter the key")
            message.config(state=DISABLED)

    elif var.get() == 3:
        if len(e1.get()) > 0:
            if e1.get().isdigit():
                key = int(e1.get())
                if myTree.isInTheTree(treeRoot, key):
                    data = myTree.search(treeRoot, key)
                    message.delete("1.0", "end")
                    message.insert(INSERT, "node with key " + str(key) + "
was found.\nit has data "+ data)
                    message.config(state=DISABLED)
                    e2.config(state=NORMAL)
                    e2.insert(0, data)
                    e2.config(state=DISABLED)
                else:
                    message.delete("1.0", "end")
                    message.insert(INSERT, "enter a key that exists in the
tree")
                    message.config(state=DISABLED)
            elif not e1.get().isdigit():
                message.delete("1.0", "end")
                message.insert(INSERT, "enter the digit for the key")
                message.config(state=DISABLED)
        else:
            message.delete("1.0", "end")
            message.insert(INSERT, "enter the key")
            message.config(state=DISABLED)

    elif var.get() == 4:
        if len(e1.get()) > 0:
            if e1.get().isdigit() and len(e2.get()) > 0:
                key = int(e1.get())
                data = e2.get()
                if myTree.isInTheTree(treeRoot, key):
                    myTree.edit(treeRoot, key, data)
                    myTree.writeToFile(treeRoot, filename)
                    message.delete("1.0", "end")
                    message.insert(INSERT, "node with key " + str(key) + "
was edited.\nit has new data "+ data)
                    message.config(state=DISABLED)
                else:
                    message.delete("1.0", "end")
                    message.insert(INSERT, "enter a key that exists in the
tree")
                    message.config(state=DISABLED)
            elif not e1.get().isdigit():
                message.delete("1.0", "end")
                message.insert(INSERT, "enter the digit for the key")
                message.config(state=DISABLED)
        else:
            message.delete("1.0", "end")
```

```
            message.insert(INSERT, "enter the key")
            message.config(state=DISABLED)
    tree.config(state=NORMAL)
    tree.delete("1.0", "end")
    tree.insert(INSERT, myTree.display(treeRoot))
    tree.config(state=DISABLED)

def deleteTree():
    global treeRoot
    file = open(filename, "w")
    file.truncate()
    file.close()
    tree.config(state=NORMAL)
    tree.delete("1.0", "end")
    tree.config(state=DISABLED)
    treeRoot = None

var = IntVar()

R1 = Radiobutton(root, text="Insert", variable=var, value=1, command=sel,
font=("Arial", 20), padx=10)

R2 = Radiobutton(root, text="Delete", variable=var, value=2, command=sel,
font=("Arial", 20), padx=10)

R3 = Radiobutton(root, text="Search", variable=var, value=3, command=sel,
font=("Arial", 20), padx=10)

R4 = Radiobutton(root, text="Edit", variable=var, value=4, command=sel,
font=("Arial", 20), padx=10)

R1.grid(row=0, column=0)
R2.grid(row=1, column=0)
R3.grid(row=2, column=0)
R4.grid(row=3, column=0)

label = Label(root, text="Enter key", font=("Arial", 15))
label.grid(row=0, column=1)

e1 = Entry(root, width=30, bd=4, font=("Arial", 15))
e1.grid(row=1, column=1)

label1 = Label(root, text="Enter data", font=("Arial", 15))
label1.grid(row=2, column=1)

e2 = Entry(root, width=30, bd=4, font=("Arial", 15))
e2.grid(row=3, column=1)

button = Button(root, text='', command=button_clicked, width=25, height=3)
button.grid(row=0, column=2, rowspan=2)

buttonDeleteTree = Button(root, text='Delete database', command=deleteTree,
font=("Arial", 15), width=17, height=2)
buttonDeleteTree.grid(row=4, column=2, padx=10)

message = Text(root, width=28, height=3, font=("Arial", 10))
message.grid(row=2, column=2, rowspan=2)

tree = Text(root, font=("Arial", 13))
tree.grid(row=4, column=1, rowspan=2)
if os.stat(filename).st_size != 0:
    treeRoot = deserialize(filename)
    tree.config(state=NORMAL)
    tree.delete("1.0", "end")
```

```
    tree.insert(INSERT, myTree.display(treeRoot))
    tree.config(state=DISABLED)


def main():
    root.mainloop()


if __name__ == "__main__":
    main()
```

3.3.2 Приклади роботи

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

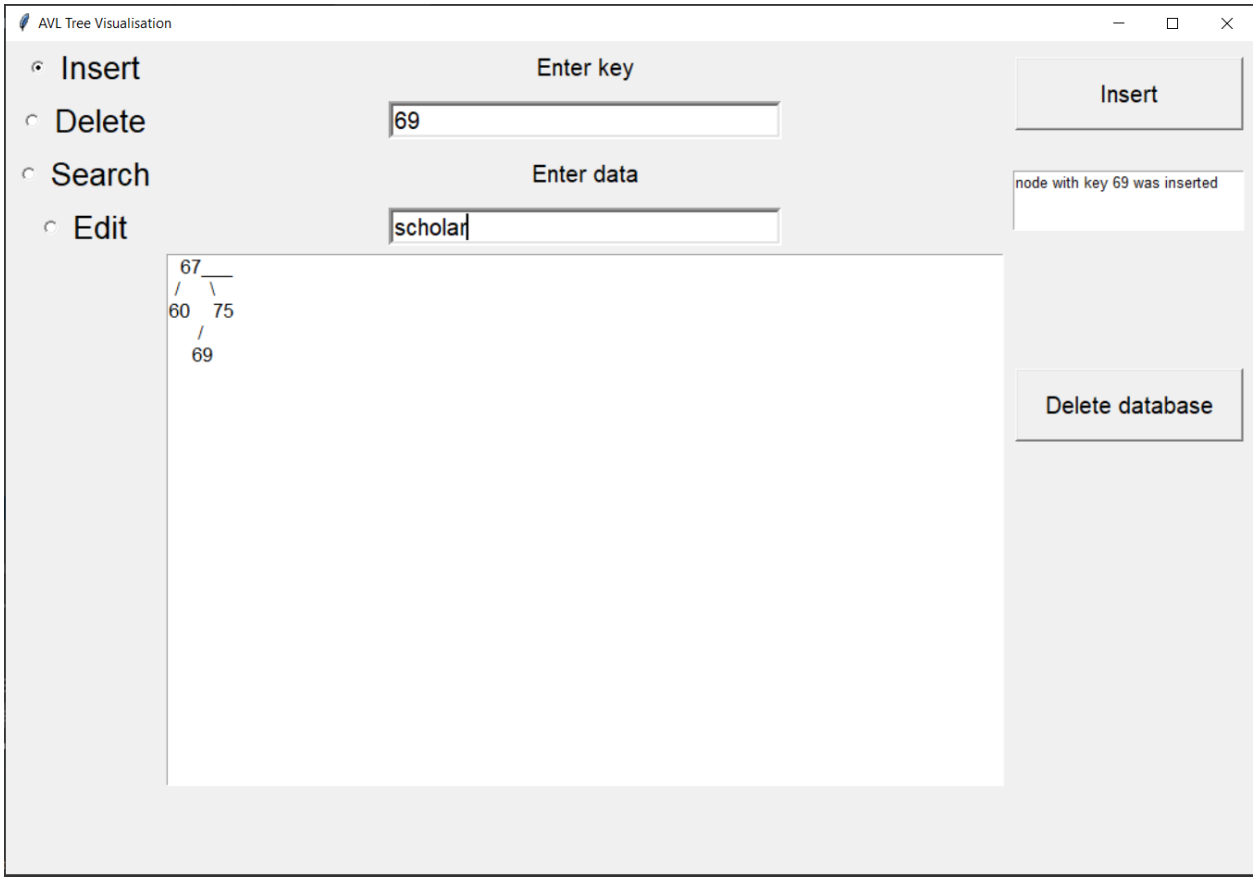

Рисунок 3.1 –Додавання запису

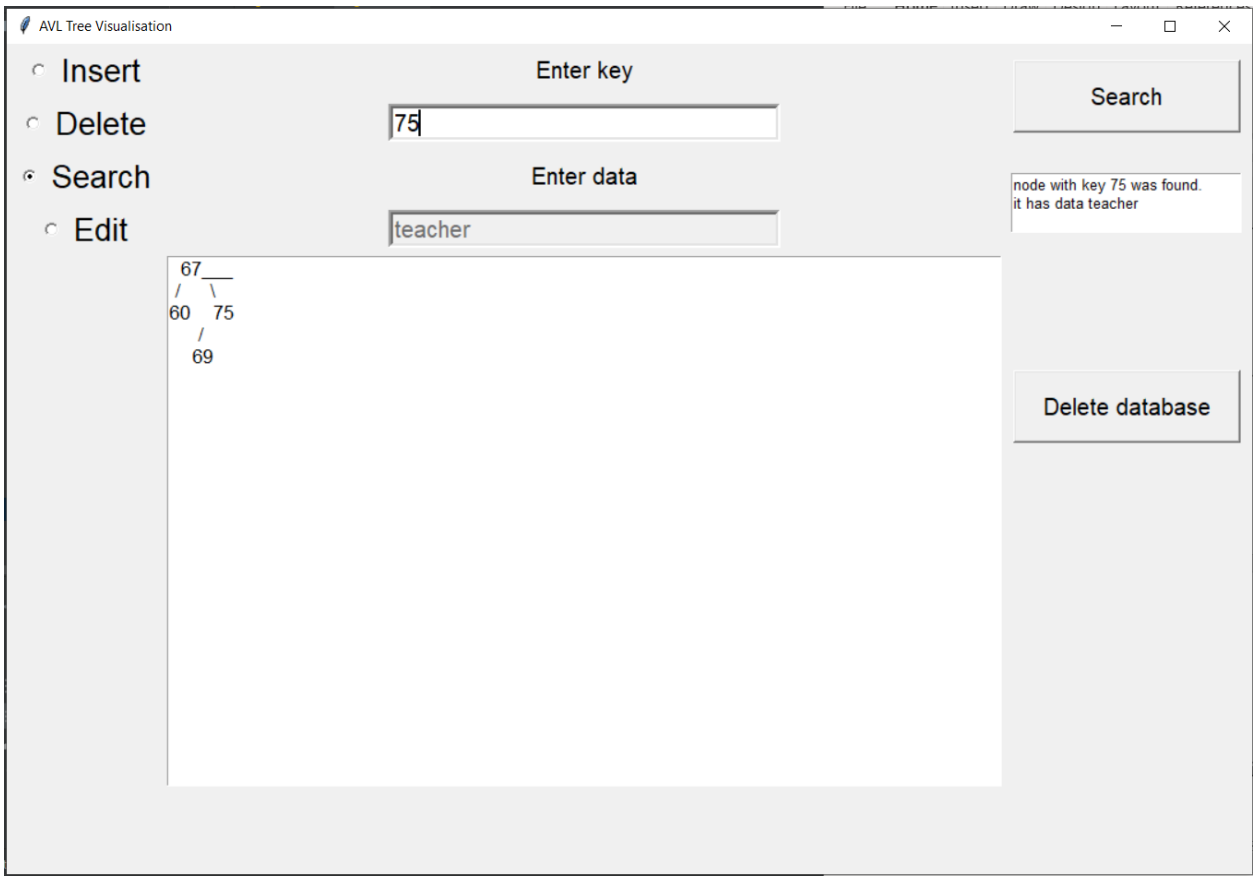

Рисунок 3.2 – Пошук запису

Тестування алгоритму

3.3.3 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	3
2	2
3	4
4	1
5	5
6	2
7	5
8	3
9	4
10	5
11	3
12	4
13	5
14	3
15	4

ВИСНОВОК

В рамках лабораторної роботи я виконала програмну реалізацію невеликої СУБД з графічним інтерфейсом користувача, з функціями пошуку, додавання, видалення та редагування записів. Для зберігання даних використовувати структуру даних згідно варіанту(АВЛ дерево). Протестувала всі написані функції. Переконалася, що вони працюють коректно.

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.