**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)

*Піонтківський Віталій ІП-12*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.Н.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc114359761)

[2 Завдання 4](#_Toc114359762)

[3 Виконання 7](#_Toc114359763)

[3.1 Псевдокод алгоритмів 7](#_Toc114359764)

[3.2 Часова складність пошуку 7](#_Toc114359765)

[3.3 Програмна реалізація 7](#_Toc114359766)

[3.3.1 Вихідний код 7](#_Toc114359767)

[3.3.2 Приклади роботи 7](#_Toc114359768)

[3.4 Тестування алгоритму 8](#_Toc114359769)

[3.4.1 Часові характеристики оцінювання 8](#_Toc114359770)

[Висновок 9](#_Toc114359771)

[Критерії оцінювання 10](#_Toc114359772)

# 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** |
| 20 | B-дерево t=100, однорідний бінарний пошук |

# Виконання

## Псевдокод алгоритмів

CLASS BTreeNode:

    FUNCTION init(self, leaf=False):

        self.leaf = leaf

        self.keys = []

        self.child = []

    FUNCTION search(self, k):

        res = self.binary\_search(k)

        IF res:

            RETURN res

        IF self.leaf:

            RETURN None

        i = 0

        WHILE i < len(self.keys) and k > self.keys[i][0]:

            i += 1

        RETURN self.child[i].search(k)

    FUNCTION binary\_search(self, k):

        start = 0

        end = len(self.keys) - 1

        WHILE start <= end:

            mid = (end + start) // 2

            IF self.keys[mid][0] > k:

                end = mid - 1

            ELIF self.keys[mid][0] < k:

                start = mid + 1

            ELSE:

                RETURN self.keys[mid][1]

        RETURN None

    FUNCTION edit(self, k, value):

        i = 0

        WHILE i < len(self.keys) and k > self.keys[i][0]:

            i += 1

        IF i < len(self.keys) and k == self.keys[i][0]:

            self.keys[i][1] = value

        ELSE:

            RETURN self.child[i].edit(k, value)

class BTree:

    FUNCTION init(self, t):

        self.root = BTreeNode(True)

        self.t = t

        self.all\_values = list()

    FUNCTION insert(self, k):

        root = self.root

        IF len(root.keys) == (2 \* self.t) - 1:

            temp = BTreeNode()

            self.root = temp

            temp.child.insert(0, root)

            self.split\_child(temp, 0)

            self.insert\_non\_full(temp, k)

        ELSE:

            self.insert\_non\_full(root, k)

    FUNCTION insert\_non\_full(self, x, k):

        i = len(x.keys) - 1

        IF x.leaf:

            x.keys.append((None, None))

            WHILE i >= 0 and x.keys[i][0] > k[0]:

                x.keys[i + 1] = x.keys[i]

                i -= 1

            x.keys[i + 1] = k

        ELSE:

            WHILE i >= 0 and x.keys[i][0] > k[0]:

                i -= 1

            i += 1

            IF len(x.child[i].keys) == (2 \* self.t) - 1:

                self.split\_child(x, i)

                IF k[0] > x.keys[i][0]:

                    i += 1

            self.insert\_non\_full(x.child[i], k)

    FUNCTION split\_child(self, x, i):

        t = self.t

        y = x.child[i]

        z = BTreeNode(y.leaf)

        x.child.insert(i + 1, z)

        x.keys.insert(i, y.keys[t - 1])

        z.keys = y.keys[t: (2 \* t) - 1]

        y.keys = y.keys[0: t - 1]

        IF not y.leaf:

            z.child = y.child[t: 2 \* t]

            y.child = y.child[0: t]

    FUNCTION print\_tree(self, x, l=0):

        print("Level ", l, " ", len(x.keys), end=":")

        FOR i in x.keys:

            print(i, end=" ")

        print()

        l += 1

        IF len(x.child) > 0:

            FOR i in x.child:

                self.print\_tree(i, l)

    FUNCTION save\_data(self, node):

        FOR i in range(0, len(node.keys)):

            IF not node.leaf:

                self.save\_data(node.child[i])

            self.all\_values.append(node.keys[i])

        IF not node.leaf:

            self.save\_data(node.child[-1])

    FUNCTION search(self, k):

        IF self.root is None:

            RETURN None

        ELSE:

            RETURN self.root.search(k)

    FUNCTION edit(self, k, value):

        self.root.edit(k, value)

    FUNCTION delete(self, x, k):

        t = self.t

        i = 0

        WHILE i < len(x.keys) and k[0] > x.keys[i][0]:

            i += 1

        IF x.leaf:

            IF i < len(x.keys) and x.keys[i][0] == k[0]:

                x.keys.pop(i)

                RETURN

            RETURN

        IF i < len(x.keys) and x.keys[i][0] == k[0]:

            RETURN self.delete\_internal\_node(x, k, i)

        ELIF len(x.child[i].keys) >= t:

            self.delete(x.child[i], k)

        ELSE:

            IF i != 0 and i + 2 < len(x.child):

                IF len(x.child[i - 1].keys) >= t:

                    self.delete\_sibling(x, i, i - 1)

                ELIF len(x.child[i + 1].keys) >= t:

                    self.delete\_sibling(x, i, i + 1)

                ELSE:

                    self.delete\_merge(x, i, i + 1)

            ELIF i == 0:

                IF len(x.child[i + 1].keys) >= t:

                    self.delete\_sibling(x, i, i + 1)

                ELSE:

                    self.delete\_merge(x, i, i + 1)

            ELIF i + 1 == len(x.child):

                IF len(x.child[i - 1].keys) >= t:

                    self.delete\_sibling(x, i, i - 1)

                ELSE:

                    self.delete\_merge(x, i, i - 1)

            self.delete(x.child[i], k)

    FUNCTION delete\_internal\_node(self, x, k, i):

        t = self.t

        IF x.leaf:

            IF x.keys[i][0] == k[0]:

                x.keys.pop(i)

                RETURN

            RETURN

        IF len(x.child[i].keys) >= t:

            x.keys[i] = self.delete\_predecessor(x.child[i])

            RETURN

        ELIF len(x.child[i + 1].keys) >= t:

            x.keys[i] = self.delete\_successor(x.child[i + 1])

            RETURN

        ELSE:

            self.delete\_merge(x, i, i + 1)

            self.delete\_internal\_node(x.child[i], k, self.t - 1)

    FUNCTION delete\_predecessor(self, x):

        IF x.leaf:

            RETURN x.keys.pop(0)

        n = len(x.keys) - 1

        IF len(x.child[n].keys) >= self.t:

            self.delete\_sibling(x, n + 1, n)

        ELSE:

            self.delete\_merge(x, n, n + 1)

        self.delete\_predecessor(x.child[n])

    FUNCTION delete\_successor(self, x):

        IF x.leaf:

            RETURN x.keys.pop(0)

        IF len(x.child[1].keys) >= self.t:

            self.delete\_sibling(x, 0, 1)

        ELSE:

            self.delete\_merge(x, 0, 1)

        self.delete\_successor(x.child[0])

    FUNCTION delete\_merge(self, x, i, j):

        cnode = x.child[i]

        IF j > i:

            rsnode = x.child[j]

            cnode.keys.append(x.keys[i])

            FOR k in range(len(rsnode.keys)):

                cnode.keys.append(rsnode.keys[k])

                IF len(rsnode.child) > 0:

                    cnode.child.append(rsnode.child[k])

            IF len(rsnode.child) > 0:

                cnode.child.append(rsnode.child.pop())

            new = cnode

            x.keys.pop(i)

            x.child.pop(j)

        ELSE:

            lsnode = x.child[j]

            lsnode.keys.append(x.keys[j])

            FOR i in range(len(cnode.keys)):

                lsnode.keys.append(cnode.keys[i])

                IF len(lsnode.child) > 0:

                    lsnode.child.append(cnode.child[i])

            IF len(lsnode.child) > 0:

                lsnode.child.append(cnode.child.pop())

            new = lsnode

            x.keys.pop(j)

            x.child.pop(i)

        IF x == self.root and len(x.keys) == 0:

            self.root = new

    FUNCTION delete\_sibling(self, x, i, j):

        cnode = x.child[i]

        IF i < j:

            rsnode = x.child[j]

            cnode.keys.append(x.keys[i])

            x.keys[i] = rsnode.keys[0]

            IF len(rsnode.child) > 0:

                cnode.child.append(rsnode.child[0])

                rsnode.child.pop(0)

            rsnode.keys.pop(0)

        ELSE:

            lsnode = x.child[j]

            cnode.keys.insert(0, x.keys[i - 1])

            x.keys[i - 1] = lsnode.keys.pop()

            IF len(lsnode.child) > 0:

                cnode.child.insert(0, lsnode.child.pop())

## Часова складність пошуку

Часова складність алгоритму бінарного пошуку становить O(log n). У кожній ітерації довжина області пошуку зменшується вдвічі. Він скорочується в геометричній прогресії.

## Програмна реалізація

### Вихідний код

from View import View

def main():

    app = View()

    app.run()

    app.mainloop()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

class BTreeNode:

    def \_\_init\_\_(self, leaf=False):

        self.leaf = leaf

        self.keys = []

        self.child = []

    def search(self, k):

        res = self.binary\_search(k)

        if res:

            return res

        if self.leaf:

            return None

        i = 0

        while i < len(self.keys) and k > self.keys[i][0]:

            i += 1

        return self.child[i].search(k)

    def binary\_search(self, k):

        start = 0

        end = len(self.keys) - 1

        while start <= end:

            mid = (end + start) // 2

            if self.keys[mid][0] > k:

                end = mid - 1

            elif self.keys[mid][0] < k:

                start = mid + 1

            else:

                return self.keys[mid][1]

        return None

    def edit(self, k, value):

        i = 0

        while i < len(self.keys) and k > self.keys[i][0]:

            i += 1

        if i < len(self.keys) and k == self.keys[i][0]:

            self.keys[i][1] = value

        else:

            return self.child[i].edit(k, value)

class BTree:

    def \_\_init\_\_(self, t):

        self.root = BTreeNode(True)

        self.t = t

        self.all\_values = list()

    def insert(self, k):

        root = self.root

        if len(root.keys) == (2 \* self.t) - 1:

            temp = BTreeNode()

            self.root = temp

            temp.child.insert(0, root)

            self.split\_child(temp, 0)

            self.insert\_non\_full(temp, k)

        else:

            self.insert\_non\_full(root, k)

    def insert\_non\_full(self, x, k):

        i = len(x.keys) - 1

        if x.leaf:

            x.keys.append((None, None))

            while i >= 0 and x.keys[i][0] > k[0]:

                x.keys[i + 1] = x.keys[i]

                i -= 1

            x.keys[i + 1] = k

        else:

            while i >= 0 and x.keys[i][0] > k[0]:

                i -= 1

            i += 1

            if len(x.child[i].keys) == (2 \* self.t) - 1:

                self.split\_child(x, i)

                if k[0] > x.keys[i][0]:

                    i += 1

            self.insert\_non\_full(x.child[i], k)

    def split\_child(self, x, i):

        t = self.t

        y = x.child[i]

        z = BTreeNode(y.leaf)

        x.child.insert(i + 1, z)

        x.keys.insert(i, y.keys[t - 1])

        z.keys = y.keys[t: (2 \* t) - 1]

        y.keys = y.keys[0: t - 1]

        if not y.leaf:

            z.child = y.child[t: 2 \* t]

            y.child = y.child[0: t]

    def print\_tree(self, x, l=0):

        print("Level ", l, " ", len(x.keys), end=":")

        for i in x.keys:

            print(i, end=" ")

        print()

        l += 1

        if len(x.child) > 0:

            for i in x.child:

                self.print\_tree(i, l)

    def save\_data(self, node):

        for i in range(0, len(node.keys)):

            if not node.leaf:

                self.save\_data(node.child[i])

            self.all\_values.append(node.keys[i])

        if not node.leaf:

            self.save\_data(node.child[-1])

    def search(self, k):

        if self.root is None:

            return None

        else:

            return self.root.search(k)

    def edit(self, k, value):

        self.root.edit(k, value)

    def delete(self, x, k):

        t = self.t

        i = 0

        while i < len(x.keys) and k[0] > x.keys[i][0]:

            i += 1

        if x.leaf:

            if i < len(x.keys) and x.keys[i][0] == k[0]:

                x.keys.pop(i)

                return

            return

        if i < len(x.keys) and x.keys[i][0] == k[0]:

            return self.delete\_internal\_node(x, k, i)

        elif len(x.child[i].keys) >= t:

            self.delete(x.child[i], k)

        else:

            if i != 0 and i + 2 < len(x.child):

                if len(x.child[i - 1].keys) >= t:

                    self.delete\_sibling(x, i, i - 1)

                elif len(x.child[i + 1].keys) >= t:

                    self.delete\_sibling(x, i, i + 1)

                else:

                    self.delete\_merge(x, i, i + 1)

            elif i == 0:

                if len(x.child[i + 1].keys) >= t:

                    self.delete\_sibling(x, i, i + 1)

                else:

                    self.delete\_merge(x, i, i + 1)

            elif i + 1 == len(x.child):

                if len(x.child[i - 1].keys) >= t:

                    self.delete\_sibling(x, i, i - 1)

                else:

                    self.delete\_merge(x, i, i - 1)

            self.delete(x.child[i], k)

    def delete\_internal\_node(self, x, k, i):

        t = self.t

        if x.leaf:

            if x.keys[i][0] == k[0]:

                x.keys.pop(i)

                return

            return

        if len(x.child[i].keys) >= t:

            x.keys[i] = self.delete\_predecessor(x.child[i])

            return

        elif len(x.child[i + 1].keys) >= t:

            x.keys[i] = self.delete\_successor(x.child[i + 1])

            return

        else:

            self.delete\_merge(x, i, i + 1)

            self.delete\_internal\_node(x.child[i], k, self.t - 1)

    def delete\_predecessor(self, x):

        if x.leaf:

            return x.keys.pop(0)

        n = len(x.keys) - 1

        if len(x.child[n].keys) >= self.t:

            self.delete\_sibling(x, n + 1, n)

        else:

            self.delete\_merge(x, n, n + 1)

        self.delete\_predecessor(x.child[n])

    def delete\_successor(self, x):

        if x.leaf:

            return x.keys.pop(0)

        if len(x.child[1].keys) >= self.t:

            self.delete\_sibling(x, 0, 1)

        else:

            self.delete\_merge(x, 0, 1)

        self.delete\_successor(x.child[0])

    def delete\_merge(self, x, i, j):

        cnode = x.child[i]

        if j > i:

            rsnode = x.child[j]

            cnode.keys.append(x.keys[i])

            for k in range(len(rsnode.keys)):

                cnode.keys.append(rsnode.keys[k])

                if len(rsnode.child) > 0:

                    cnode.child.append(rsnode.child[k])

            if len(rsnode.child) > 0:

                cnode.child.append(rsnode.child.pop())

            new = cnode

            x.keys.pop(i)

            x.child.pop(j)

        else:

            lsnode = x.child[j]

            lsnode.keys.append(x.keys[j])

            for i in range(len(cnode.keys)):

                lsnode.keys.append(cnode.keys[i])

                if len(lsnode.child) > 0:

                    lsnode.child.append(cnode.child[i])

            if len(lsnode.child) > 0:

                lsnode.child.append(cnode.child.pop())

            new = lsnode

            x.keys.pop(j)

            x.child.pop(i)

        if x == self.root and len(x.keys) == 0:

            self.root = new

    def delete\_sibling(self, x, i, j):

        cnode = x.child[i]

        if i < j:

            rsnode = x.child[j]

            cnode.keys.append(x.keys[i])

            x.keys[i] = rsnode.keys[0]

            if len(rsnode.child) > 0:

                cnode.child.append(rsnode.child[0])

                rsnode.child.pop(0)

            rsnode.keys.pop(0)

        else:

            lsnode = x.child[j]

            cnode.keys.insert(0, x.keys[i - 1])

            x.keys[i - 1] = lsnode.keys.pop()

            if len(lsnode.child) > 0:

                cnode.child.insert(0, lsnode.child.pop())

T = 100

FILE\_NAME = 'data/MOCK\_DATA.txt'

from Model import \*

from tkinter import messagebox

class Control:

    def \_\_init\_\_(self, view):

        self.\_\_view = view

        self.\_\_BTree = BTree(T)

        self.open\_file()

    def open\_file(self):

        try:

            file = open(FILE\_NAME, "rt")

            data = file.read()

            data = data.strip()

            data = data.split('\n')

            for i in range(0, len(data)):

                k = data[i][1:-1].split(', ')

                self.\_\_BTree.insert([int(k[0]), k[1][1:-1]])

            file.close()

            self.\_\_view.set\_label(f"File {FILE\_NAME} was successfully opened")

        except IOError:

            messagebox.showerror("Error", f"File {FILE\_NAME} can not be open")

    def submit(self, action, key, value):

        try:

            key = int(key)

        except ValueError:

            action = 0

            messagebox.showerror('Error', "Key is not right")

        match action:

            case 1:

                self.\_\_search(key)

            case 2:

                self.\_\_add(key, value)

            case 3:

                self.\_\_delete(key)

            case 4:

                self.\_\_edit(key, value)

            case \_:

                pass

    def \_\_search(self, key):

        value = self.\_\_BTree.search(key)

        if value is not None:

            self.\_\_view.set\_key(key)

            self.\_\_view.set\_value(value)

            self.\_\_view.set\_label(f"Key: {key} Value: {value}")

        else:

            self.\_\_view.set\_label("There is no such value")

            self.\_\_view.set\_value("There is no such value")

    def \_\_add(self, key, value):

        value = value.strip()

        if value:

            res = self.\_\_BTree.search(key)

            if not res:

                self.\_\_BTree.insert([key, value])

                self.\_\_view.set\_label("Element was added")

            else:

                messagebox.showwarning("Warning", "The key is exists")

        else:

            messagebox.showwarning("Warning", "Field 'Value' is  empty")

    def \_\_delete(self, key):

        res = self.\_\_BTree.search(key)

        if res:

            self.\_\_BTree.delete(self.\_\_BTree.root, [key, ])

            self.\_\_view.set\_label(f"Element {key} was successfully deleted")

        else:

            messagebox.showwarning("Warning", "There is no such key")

    def \_\_edit(self, key, value):

        value = value.strip()

        if value:

            res = self.\_\_BTree.search(key)

            if res:

                self.\_\_BTree.edit(key, value)

                self.\_\_view.set\_label("Element was successfully changed")

            else:

                messagebox.showwarning("Warning", "There is no such key")

        else:

            messagebox.showwarning("Warning", "Field 'Value' is  empty")

    def save\_file(self):

        self.\_\_BTree.all\_values = list()

        self.\_\_BTree.save\_data(self.\_\_BTree.root)

        with open(FILE\_NAME, "wt") as file:

            file.writelines("%s\n" % i for i in self.\_\_BTree.all\_values)

        self.\_\_view.set\_label("DB is successfully saved")

from tkinter import \*

from Control import \*

class View(Tk):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.\_\_frame = Frame(self)

        self.\_\_bd\_lbl = Label(self.\_\_frame, text="BTree", font=("Times New Roman", 14))

        self.\_\_key\_lbl = Label(self.\_\_frame, text="Key", font=("Times New Roman", 12))

        self.\_\_key\_ent = Entry(self.\_\_frame, font=("Times New Roman", 12))

        self.\_\_value\_lbl = Label(self.\_\_frame, text="Value", font=("Times New Roman", 12))

        self.\_\_value\_ent = Entry(self.\_\_frame, font=("Times New Roman", 12))

        self.\_\_clear\_btn = Button(self.\_\_frame, text="Clear fields", font=("Times New Roman", 12), command=self.\_\_clear)

        self.\_\_choose\_lbl = Label(self.\_\_frame, text="Choose option", font=("Times New Roman", 14))

        self.\_\_action = IntVar()

        self.\_\_action.set(1)

        self.\_\_search\_rbtn = Radiobutton(self.\_\_frame, text="Search by key", variable=self.\_\_action, value=1,

                                         font=("Times New Roman", 12))

        self.\_\_add\_rbtn = Radiobutton(self.\_\_frame, text="Add value", variable=self.\_\_action, value=2,

                                      font=("Times New Roman", 12))

        self.\_\_delete\_rbtn = Radiobutton(self.\_\_frame, text="Delete value", variable=self.\_\_action, value=3,

                                         font=("Times New Roman", 12))

        self.\_\_edit\_rbtn = Radiobutton(self.\_\_frame, text="Change", variable=self.\_\_action, value=4,

                                       font=("Times New Roman", 12))

        self.\_\_submit\_btn = Button(self.\_\_frame, text="Execute", font=("Times New Roman", 12), command=self.\_\_submit)

        self.\_\_result = StringVar()

        self.\_control = Control(self)

        self.\_\_result\_lbl = Label(self.\_\_frame, textvariable=self.\_\_result, font=("Times New Roman", 14))

        self.\_\_save\_btn = Button(self.\_\_frame, text="Save results to file", font=("Times New Roman", 12),

                                 command=self.\_control.save\_file)

    def run(self):

        self.\_\_setup\_window()

        self.\_\_place\_widgets()

    def \_\_setup\_window(self):

        self.title("Lab 3")

        monitor\_height = self.winfo\_screenheight()

        monitor\_width = self.winfo\_screenwidth()

        app\_height = 500

        app\_width = 600

        self.geometry(

            f"{app\_width}x{app\_height}+{int((monitor\_width - app\_width) / 2)}+{int((monitor\_height - app\_height) / 2) - 100}")

        self.resizable(False, False)

        self.\_\_frame.place(relheight=0.96, relwidth=0.96, relx=0.02, rely=0.02)

    def \_\_place\_widgets(self):

        self.\_\_frame.columnconfigure(0, weight=1)

        self.\_\_frame.columnconfigure(1, weight=1)

        self.\_\_bd\_lbl.grid(row=0, column=0, columnspan=2, pady=5)

        self.\_\_key\_lbl.grid(row=1, column=0, sticky="e", pady=5, padx=20)

        self.\_\_key\_ent.grid(row=1, column=1, sticky="wens", pady=5)

        self.\_\_value\_lbl.grid(row=2, column=0, sticky="e", pady=5, padx=20)

        self.\_\_value\_ent.grid(row=2, column=1, sticky="wens", pady=5)

        self.\_\_clear\_btn.grid(row=3, column=0, columnspan=2, pady=5)

        self.\_\_choose\_lbl.grid(row=4, column=0, columnspan=2, pady=5)

        self.\_\_search\_rbtn.grid(row=5, column=0, sticky="w")

        self.\_\_add\_rbtn.grid(row=6, column=0, sticky="w")

        self.\_\_delete\_rbtn.grid(row=7, column=0, sticky="w")

        self.\_\_edit\_rbtn.grid(row=8, column=0, sticky="w")

        self.\_\_submit\_btn.grid(row=5, column=1, rowspan=4, sticky="we")

        self.\_\_result\_lbl.grid(row=9, column=0, columnspan=2, pady=5, sticky="w")

        self.\_\_save\_btn.grid(row=10, column=0, columnspan=2, pady=5, sticky="we")

    def \_\_clear(self):

        self.\_\_key\_ent.delete(0, END)

        self.\_\_value\_ent.delete(0, END)

    def \_\_submit(self):

        self.\_control.submit(self.\_\_action.get(), self.\_\_key\_ent.get(), self.\_\_value\_ent.get())

    def set\_key(self, key):

        self.\_\_key\_ent.delete(0, END)

        self.\_\_key\_ent.insert(0, key)

    def set\_value(self, value):

        self.\_\_value\_ent.delete(0, END)

        self.\_\_value\_ent.insert(0, value)

    def set\_label(self, line):

        self.\_\_result.set(line)

### Приклади роботи

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

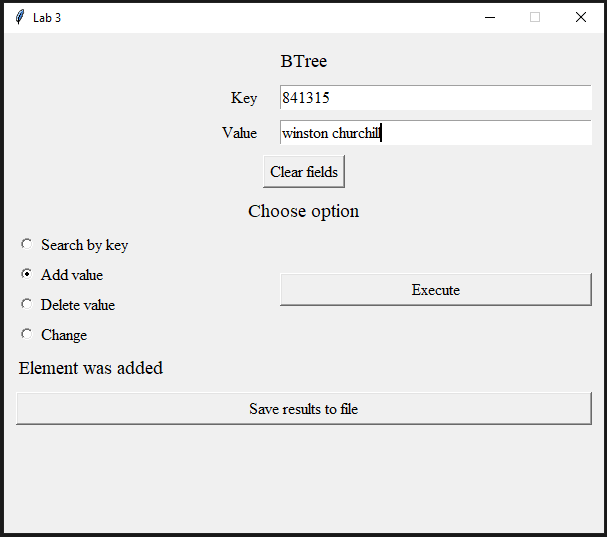


Рисунок 3.1 –Додавання запису

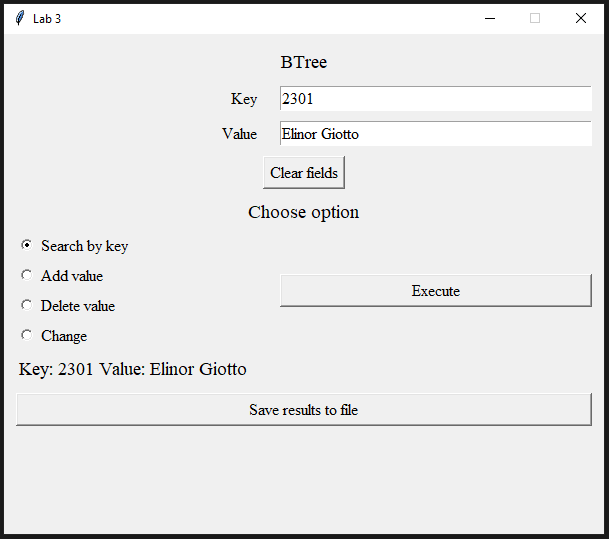


Рисунок 3.2 – Пошук запису

## Тестування алгоритму

### Часові характеристики оцінювання

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

Таблиця 3.1 – Число порівнянь при спробі пошуку запису по ключу

|  |  |
| --- | --- |
| Номер спроби пошуку | Число порівнянь |
| 1 | 24 |
| 2 | 15 |
| 3 | 38 |
| 4 | 22 |
| 5 | 27 |
| 6 | 23 |
| 7 | 8 |
| 8 | 18 |
| 9 | 20 |
| 10 | 27 |
| 11 | 25 |
| 12 | 30 |
| 13 | 9 |
| 14 | 14 |
| 15 | 23 |

Висновок

Під час виконання лабораторної роботи було створено програмного забезпечення для функціонування простої субд з графічним інтерфейсом, в якій дані зберігаються бі деревом. Програмне забезпечення містить функції для додавання, редагування, видалення та пошуку елементів, які складаються з ключа та значення. Програма містить в собі чотири класи: два відповідають за графічне забезпечення програми, один за бі дерево і один створений для запуску програми.

За допомогою однорідного бінарного пошуку та структури бі дерева пошук елементів займає мінімальну кількість часу та ітерацій порівняння елементів і чудово працює.