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 Васильєв Єгор* _____
(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

```
клас Node:
    конструктор(self):
        self.keys = []
        self.child = []

    @property
    функція leaf(self):
        повернути not self.child

клас BTree:
    конструктор (self, t):
        self.t = t
        self.min_keys = t - 1
        self.max_keys = 2 * t - 1
        self.root = Node()
        self.comps = 0

    функція insert(self, key):
        якщо len(self.root.keys) != self.max_keys:
            self.insert_in_node(self.root, key)
        інакше:
            new_root = Node()
            new_root.child.append(self.root)
            self.split_child(new_root, 0)
            self.root = new_root
            self.insert(key)

    функція insert_in_node(self, node, key):
        i = len(node.keys) - 1
        поки i >= 0 and node.keys[i][0] >= key[0]:
            i -= 1
        якщо node.leaf:
            node.keys.insert(i + 1, key)
        інакше:
            якщо len(node.child[i + 1].keys) == self.max_keys:
                self.split_child(node, i + 1)
                якщо node.keys[i + 1][0] < key[0]:
                    i += 1
            self.insert_in_node(node.child[i + 1], key)
```

```
функція split_child(self, parent, i):
    new_child = Node()
    half_max = self.max_keys // 2
    child = parent.child[i]
    middle = child.keys[half_max]
    new_child.keys = child.keys[half_max + 1:]
    child.keys = child.keys[:half_max]
    якщо not child.leaf:
        new_child.child = child.child[half_max + 1:]
        child.child = child.child[:half_max + 1]
    parent.keys.insert(i, middle)
    parent.child.insert(i + 1, new_child)

функція search(self, key):
    res = self.search_in_node(key, self.root)
    повернути res[-1][1:] якщо res інакше res

функція search_in_node(self, key, node, parent=None):
    keys = list(node.keys)
    n = len(keys)
    якщо not n % 2:
        keys.append((float('inf'), 0))
        n += 1
    rounded_middle = n // 2 + int(n % 2)
    middle = n // 2
    поки middle:
        self.comps += 1
        якщо keys[rounded_middle - 1][0] == key:
            повернути node, parent, rounded_middle - 1,
            node.keys[rounded_middle - 1]
        інкакше якщо keys[rounded_middle - 1][0] < key:
            rounded_middle = rounded_middle + middle // 2 + int(middle % 2)
        інакше:
            rounded_middle = rounded_middle - middle // 2 - int(middle % 2)
        middle = middle // 2
    якщо keys[rounded_middle - 1][0] == key:
        self.comps += 1
        повернути node, parent, rounded_middle - 1,
        node.keys[rounded_middle - 1]
    якщо node.leaf:
        повернути None
    інакше:
```

```
            якщо keys[rounded_middle - 1][0] > key:
                self.comps += 1
                повернути self.search_in_node(key,
                node.child[rounded_middle - 1], node)
            інакше:
                повернути self.search_in_node(key,
                node.child[rounded_middle], node)

    функція edit(self, key):
        r = self.search_in_node(key[0], self.root)
        якщо r:
            node, _, i, _ = r
            node.keys[i] = key
            повернути True
        інакше:
            повернути None

    функція delete(self, k):
        r = self.search_in_node(k, self.root)
        якщо r:
            node, parent, _, _ = r
        інакше:
            повернути False

        i = self.delete_in_node(node, k)

        якщо node.leaf:
            якщо len(node.keys) < self.min_keys:
                i = parent.child.index(node)
                якщо i != 0 and len(parent.child[i - 1].keys) > self.min_keys:
                    node.keys.insert(0, parent.keys.pop(i - 1))
                    parent.keys.insert(i - 1, parent.child[i - 1].keys.pop())
                інакше:
                    якщо i != len(parent.child) - 1 and
                    len(parent.child[i + 1].keys) > self.min_keys:
                        node.keys.append(parent.keys.pop(i))
                        parent.keys.insert(i, parent.child[i + 1].keys.pop(0))
                    інакше якщо i == len(parent.child) - 1:
                        node.keys = parent.child[i - 1].keys +
                        [parent.keys.pop(i - 1)] + node.keys
                        parent.child.pop(i - 1)
                    інакше:
                        node.keys = node.keys + [parent.keys.pop(i)] +
```

```
                    parent.child[i + 1].keys
                    parent.child.pop(i + 1)
        інакше:
            sibling = node.child[i]
            поки not sibling.leaf:
                sibling = sibling.child[-1]
            якщо len(sibling.keys) > self.min_keys:
                node.keys.insert(i, sibling.keys.pop())
            інакше:
                parent = node
                sibling = node.child[i + 1]
                поки not sibling.leaf:
                    parent = sibling
                    sibling = parent.child[0]

                якщо len(sibling.keys) > self.min_keys:
                    node.keys.insert(i, sibling.keys.pop(0))
                інакше:
                    якщо parent == node:
                        node.child[i].keys += node.child[i + 1].keys
                        node.child[i].child += node.child[i + 1].child
                        node.child.pop(i + 1)
                    інакше:
                        node.keys.insert(i, sibling.keys.pop(0))
                        якщо len(parent.child[1].keys) > self.min_keys:
                            sibling.keys.append(parent.keys.pop(0))
                            parent.keys.insert(i, parent.child[1].keys.pop(0))
                        інакше:
                            sibling.keys = sibling.keys + [parent.keys.pop(0)]
                             + parent.child[1].keys
                            parent.child.pop(1)
        повернути True

    @staticmethod
    функція delete_in_node(node, k):
        для i, key в переліченних(node.keys):
            якщо key[0] == k:
                node.keys.pop(i)
                повернути i
```

3.2 Часова складність пошуку

Проходячи від кореня до шуканої вершини, в найгіршому випадку, ми відвідаємо $O(\log_t n)$ вершин. Тепер, маючи на кожному вузлі не більше t елементів, використовуючи бінарний пошук зі складністю $\log_2 t$, отримаємо загальну складність $O(\log_t n * \log_2 t)$). Враховуючи, що $t << n$, маємо кінцеву складність $O(\log_t n) = O(\log n)$

3.3 Програмна реалізація

3.3.1 Вихідний код

Файл GUI.py

```
import customtkinter
from Btree import BTree

class Interface(customtkinter.CTk):
    def __init__(self):
        super().__init__()
        self.tree = BTree(3)

        self.title("B-Tree DBMS")
        self.geometry("720x720")
        self.minsize(360, 360)

        self.grid_columnconfigure(0, weight=1)
        self.grid_rowconfigure(5, weight=1)

        self.upper_frame = customtkinter.CTkFrame(self, corner_radius=0,
        fg_color=("#EBEBEC", "#212325"))
        self.upper_frame.columnconfigure((0, 1, 2, 3), weight=1)
        self.upper_frame.grid(row=0, column=0, sticky="nsew")

        customtkinter.CTkLabel(self.upper_frame, width=60,
        text="Key:").grid(row=0, column=0, padx=10, pady=10)

        self.key = customtkinter.StringVar(value="")
        self.key_field = customtkinter.CTkEntry(self.upper_frame,
        textvariable=self.key, width=80)
        self.key_field.grid(row=0, column=1, padx=10, pady=10)
```

```
        customtkinter.CTkLabel(self.upper_frame, width=80,
        text="Value:").grid(row=0, column=2, padx=10, pady=10)

        self.value = customtkinter.StringVar(value="")
        self.value_field = customtkinter.CTkEntry(self.upper_frame,
        textvariable=self.value, width=120)
        self.value_field.grid(row=0, column=3, padx=10, pady=10)

        self.button_frame = customtkinter.CTkFrame(self,
        corner_radius=0)
        self.button_frame.columnconfigure((0, 1, 2, 3), weight=1)
        self.button_frame.grid(row=1, column=0, sticky="nsew")

        customtkinter.CTkButton(self.button_frame, text="Search",
        command=self.search, fg_color='#ea9148',
        hover_color='#cc873d').grid(row=0, column=0, padx=10, pady=10)
        customtkinter.CTkButton(self.button_frame, text="Insert",
        command=self.insert, fg_color='#ea9148',
        hover_color='#cc873d').grid(row=0, column=1, padx=10, pady=10)
        customtkinter.CTkButton(self.button_frame, text="Edit",
        command=self.edit, fg_color='#ea9148',
        hover_color='#cc873d').grid(row=0, column=2, padx=10, pady=10)
        customtkinter.CTkButton(self.button_frame, text="Delete",
        command=self.delete, fg_color='#ea9148',
        hover_color='#cc873d').grid(row=0, column=3, padx=10, pady=10)
        customtkinter.CTkLabel(self, width=60,
        text="Schema:").grid(row=2, column=0, sticky="W")
        self.schema = customtkinter.CTkTextbox(self, font=("Arial", 14))
        self.schema.grid(row=3, column=0, sticky="nsew", padx=10)

        customtkinter.CTkLabel(self, width=40, text="Log:").grid(row=4,
        column=0, sticky="W")
        self.logbox = customtkinter.CTkTextbox(self, font=("Arial", 14))
        self.logbox.grid(row=5, column=0, sticky="nsew", padx=10,
        pady=(0, 20))

    def search(self):
        try:
            self.log(f"Search:\tKey: {self.key.get()}")
            val = self.tree.search(int(self.key.get()))
            if val:
                self.value.set(val)
                self.log(f"\nSuccessful, value = {val[0]}")
```

```
            else:
                self.value.set("")
                self.log("\nKey is not found")
        except ValueError:
            self.log("\nIncorrect value")

    def insert(self):
        try:
            self.log(f"Insert:\tKey:
            {self.key.get()}\tValue: {self.value.get()}")
            if self.tree.search(int(self.key.get())):
                self.log("\nKey is already in tree")
            elif self.value.get() == "":
                self.log("\nPlease, enter value")
            else:
                self.tree.insert((int(self.key.get()),
                *self.value.get().split(" ")))
                self.update_schema()
                self.log("\nSuccessful")
                self.value.set("")
                self.key.set("")
        except ValueError:
            self.log("\nIncorrect value")

    def edit(self):
        try:
            self.log(f"Edit:\tKey: {self.key.get()}\tNew value:
            {self.value.get()}")
            if self.value.get() == "":
                self.log("\nPlease, enter value")
            elif self.tree.edit((int(self.key.get()),
            *self.value.get().split(" "))):
                self.log("\nSuccessful")
                self.value.set("")
            else:
                self.log("\nKey is not in tree")
        except ValueError:
            self.log("\nIncorrect value")

    def delete(self):
        try:
            self.log(f"Delete\tKey: {self.key.get()}")
            if self.tree.delete(int(self.key.get())):
```

```
                    self.log("\nSuccessful")
                    self.key.set("")
                    self.update_schema()
                else:
                    self.log("\nKey is not in tree")
            except ValueError:
                self.log("\nIncorrect value")

        def log(self, s):
            self.logbox.insert("0.0", s + "\n")

        def update_schema(self):
            self.schema.delete('0.0', "end")
            self.schema.insert("end", self.tree)


    app = Interface()
    app.mainloop()
```

Файл GUI.py

```
import random

class Node:
    def __init__(self):
        self.keys = []
        self.child = []

    @property
    def leaf(self):
        return not self.child


class BTree:
    def __init__(self, t):
        self.t = t
        self.min_keys = t - 1
        self.max_keys = 2 * t - 1
        self.root = Node()
        self.comps = 0

    def insert(self, key):
```

```
    if len(self.root.keys) != self.max_keys:
        self.insert_in_node(self.root, key)
    else:
        new_root = Node()
        new_root.child.append(self.root)
        self.split_child(new_root, 0)
        self.root = new_root
        self.insert(key)

def insert_in_node(self, node, key):
    i = len(node.keys) - 1
    while i >= 0 and node.keys[i][0] >= key[0]:
        i -= 1
    if node.leaf:
        node.keys.insert(i + 1, key)
    else:
        if len(node.child[i + 1].keys) == self.max_keys:
            self.split_child(node, i + 1)
            if node.keys[i + 1][0] < key[0]:
                i += 1
        self.insert_in_node(node.child[i + 1], key)

def split_child(self, parent, i):
    new_child = Node()
    half_max = self.max_keys // 2
    child = parent.child[i]
    middle = child.keys[half_max]
    new_child.keys = child.keys[half_max + 1:]
    child.keys = child.keys[:half_max]
    if not child.leaf:
        new_child.child = child.child[half_max + 1:]
        child.child = child.child[:half_max + 1]
    parent.keys.insert(i, middle)
    parent.child.insert(i + 1, new_child)

def search(self, key):
    res = self.search_in_node(key, self.root)
    return res[-1][1:] if res else res

def search_in_node(self, key, node, parent=None):
    keys = list(node.keys)
    n = len(keys)
    if not n % 2:
```

```
            keys.append((float('inf'), 0))
            n += 1
        rounded_middle = n // 2 + int(n % 2)
        middle = n // 2
        while middle:
            self.comps += 1
            if keys[rounded_middle - 1][0] == key:
                return node, parent, rounded_middle - 1,
                node.keys[rounded_middle - 1]
            elif keys[rounded_middle - 1][0] < key:
                rounded_middle = rounded_middle + middle // 2 + int(middle % 2)
            else:
                rounded_middle = rounded_middle - middle // 2 - int(middle % 2)
            middle = middle // 2
        if keys[rounded_middle - 1][0] == key:
            self.comps += 1
            return node, parent, rounded_middle - 1,
            node.keys[rounded_middle - 1]
        if node.leaf:
            return None
        else:
            if keys[rounded_middle - 1][0] > key:
                self.comps += 1
                return self.search_in_node(key,
                node.child[rounded_middle - 1], node)
            else:
                return self.search_in_node(key,
                node.child[rounded_middle], node)

    def edit(self, key):
        r = self.search_in_node(key[0], self.root)
        if r:
            node, _, i, _ = r
            node.keys[i] = key
            return True
        else:
            return None

    def delete(self, k):
        r = self.search_in_node(k, self.root)
        if r:
            node, parent, _, _ = r
        else:
```

```
        return False

    i = self.delete_in_node(node, k)

    if node.leaf:
        if len(node.keys) < self.min_keys:
            i = parent.child.index(node)
            if i != 0 and len(parent.child[i - 1].keys) > self.min_keys:
                node.keys.insert(0, parent.keys.pop(i - 1))
                parent.keys.insert(i - 1, parent.child[i - 1].keys.pop())
            else:
                if i != len(parent.child) - 1 and
                len(parent.child[i + 1].keys) > self.min_keys:
                    node.keys.append(parent.keys.pop(i))
                    parent.keys.insert(i, parent.child[i + 1].keys.pop(0))
                elif i == len(parent.child) - 1:
                    node.keys = parent.child[i - 1].keys +
                    [parent.keys.pop(i - 1)] + node.keys
                    parent.child.pop(i - 1)
                else:
                    node.keys = node.keys + [parent.keys.pop(i)] +
                    parent.child[i + 1].keys
                    parent.child.pop(i + 1)
    else:
        sibling = node.child[i]
        while not sibling.leaf:
            sibling = sibling.child[-1]
        if len(sibling.keys) > self.min_keys:
            node.keys.insert(i, sibling.keys.pop())
        else:
            parent = node
            sibling = node.child[i + 1]
            while not sibling.leaf:
                parent = sibling
                sibling = parent.child[0]

            if len(sibling.keys) > self.min_keys:
                node.keys.insert(i, sibling.keys.pop(0))
            else:
                if parent == node:
                    node.child[i].keys += node.child[i + 1].keys
                    node.child[i].child += node.child[i + 1].child
                    node.child.pop(i + 1)
```

```
                    else:
                        node.keys.insert(i, sibling.keys.pop(0))
                        if len(parent.child[1].keys) > self.min_keys:
                            sibling.keys.append(parent.keys.pop(0))
                            parent.keys.insert(i, parent.child[1].keys.pop(0))
                        else:
                            sibling.keys = sibling.keys + [parent.keys.pop(0)]
                            + parent.child[1].keys
                            parent.child.pop(1)
        return True

    @staticmethod
    def delete_in_node(node, k):
        for i, key in enumerate(node.keys):
            if key[0] == k:
                node.keys.pop(i)
                return i

    def __repr__(self):
        def show(x, l):
            r = "\t" * l + str([a[0] for a in x.keys])[1:-1] + "\n"
            for child in x.child:
                r += show(child, l + 1)
            return r

        return show(self.root, 0)

    def insert_random_values(self, n):
        values = list(range(n))
        random.shuffle(values)
        for value in values:
            self.insert((value, float('inf')))

    def test(self, n):
        for i in range(1, n + 1):
            self.search(100 * i)
            print(f"Search number: {i}\nComparisons: {self.comps}")
            self.comps = 0

tree = BTree(10)
tree.insert_random_values(10000)
tree.test(15)
```

3.3.2 Приклади роботи

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

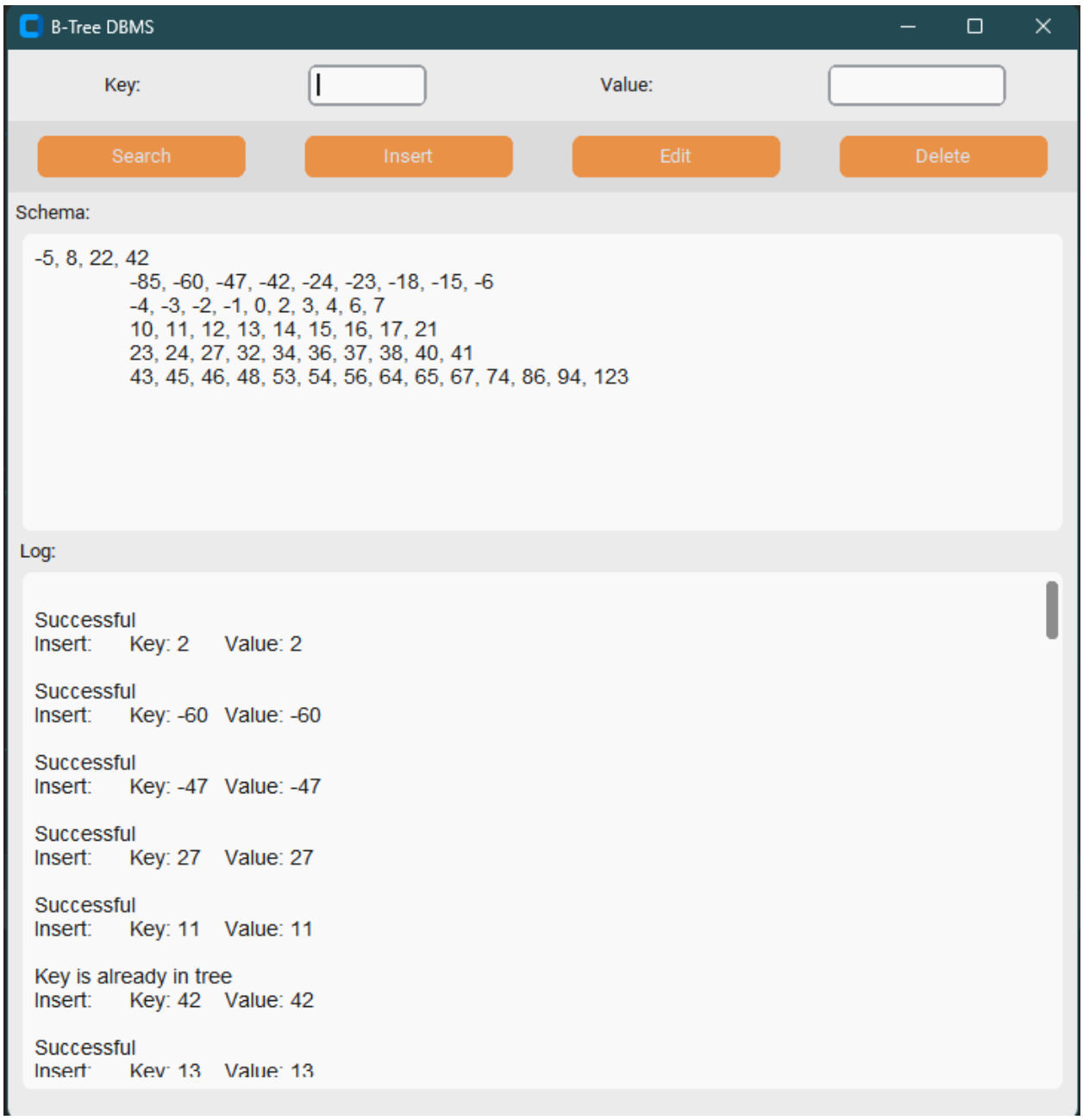

Рисунок 3.1 –Додавання запису

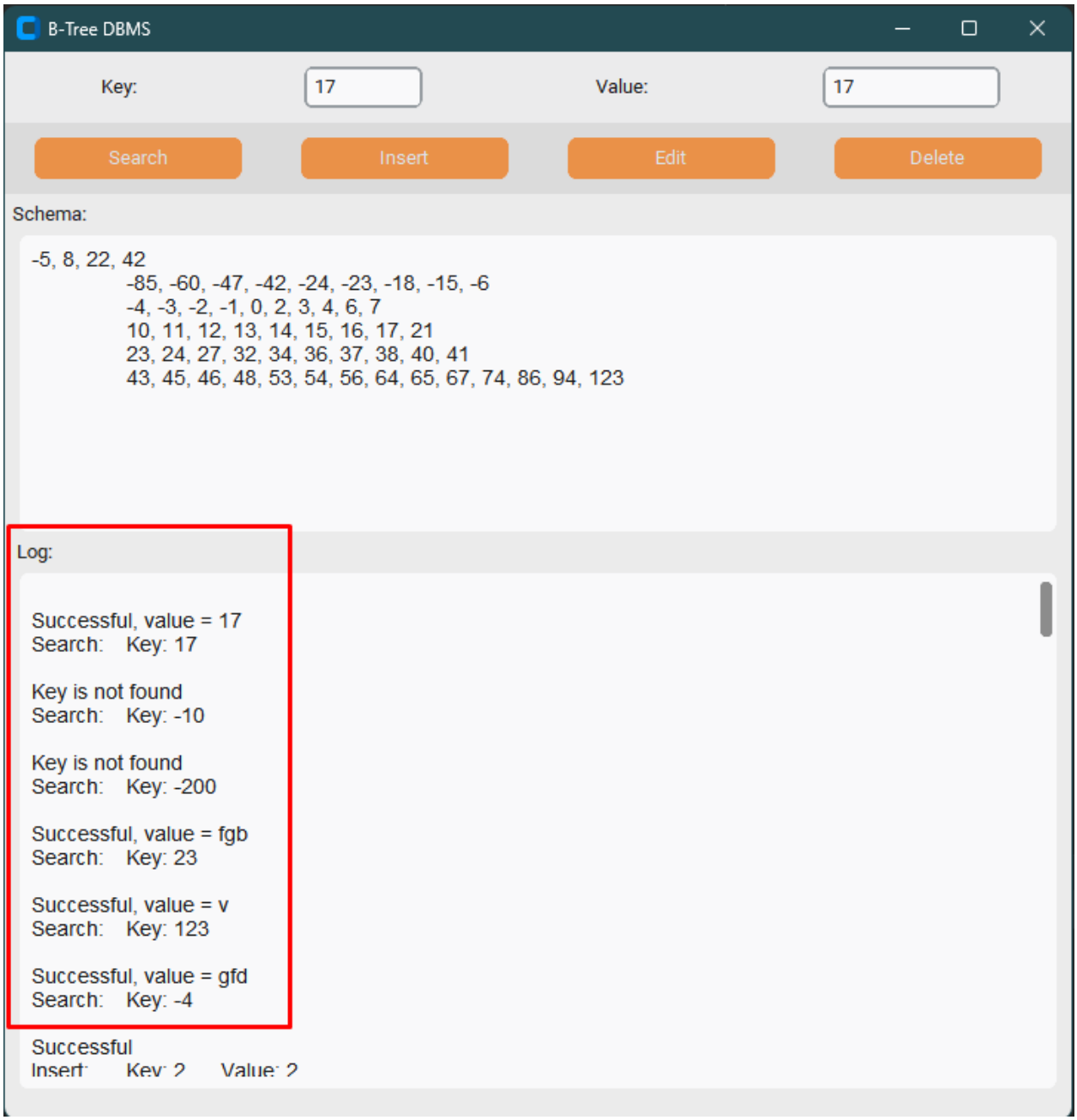

Рисунок 3.2 – Пошук запису

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	12
3	13
4	10
5	13
6	14
7	14
8	9
9	14
10	13
11	13
12	12
13	13
14	12
15	15
Середнє	12,6

ВИСНОВОК

В рамках даної лабораторної роботи було розроблено та виконано програмну реалізацію з графічним інтерфейсом алгоритмів пошуку, редагування та видалення записів у B-дереві з параметром t=10. Було проаналізовано часову складність алгоритму та проведено ряд тестів для визначення середньої кількості порівнянь під час пошуку певного значення, а отримане значення 12,6 свідчить про гарну ефективність досліджуваної структури даних.

B-дерева застосовуються для організації індексів у багатьох сучасних СУБД та для структурування інформації на жорсткому диску (зазвичай метаданих). Диск читає/пише дані великими шматочками за раз (наприклад, по 4кб), і кількість нащадків у вершини підбирається під дане обмеження. Тому виходить дуже "невисоке" дерево, що добре підходить для зберігання на диску. Саме тому B-дерева показуються гарну продуктивність у випадках, коли доступ до даних здійснюється фізичними блоками.

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.