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 Недельчев Є.О*
(шифр, прізвище, ім'я, по батькові)

Перевірив *Сопов О.О.*
(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ 3

2 ЗАВДАННЯ 4

3 ВИКОНАННЯ 7

3.1 Псевдокод алгоритмів 7

3.2 Часова складність пошуку 9

3.3 Програмна реалізація 10

3.3.1 Вихідний код 10

3.3.2 Приклади роботи 13

3.4 Тестування алгоритму 14

3.4.1 Часові характеристики оцінювання 14

ВИСНОВОК 15

КРИТЕРІЇ ОЦІНЮВАННЯ 16

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

Варіант 18

3.1 Псевдокод алгоритмів

```
class BTree:
    function __init__(t):
        root = None
        t = t


    function search(key):
        return _search(root, key)

    function _search(node, key):
        if node is None:
            return None
        i = 0
        while i < node.numKeys and key > node.keys[i]:
            i += 1
        if i < node.numKeys and key == node.keys[i]:
            return node, i
        elif node.isLeaf:
            return None
        else:
            return _search(node.children[i], key)

    function insert(key):
        if root is None:
        root = BTreeNode(t, True)
        root.keys.append(key)
        root.numKeys += 1
        else:
            if root.numKeys == 2 * t - 1:
                newRoot = BTreeNode(t, False)
                newRoot.children.append(root)
                newRoot.splitChild(0)
                if newRoot.keys[0] < key:
                    i = 1
                else:
                    i = 0
                newRoot.children[i].insertNonFull(key)
                root = newRoot
            else:
                root.insertNonFull(key)

    function delete(key):
        if root is None:
```

```
            return
        root.delete(key)
        if root.numKeys == 0:
        root = root.children[0] if root.isLeaf else None

    function update(key, new_key):
        node, index = search(key)
        if node:
            node.keys[index] = new_key
            delete(key)
            insert(new_key)
        else:
            print("Key not found")
```

3.2 Часова складність пошуку

Під час пошуку у B-дереві ми проходимо від кореня до вершини, яка містить або не містить шуканий ключ. У найгіршому випадку ми проходимо $h = 0(\log_t n)$ вершин, у середньому випадку ми також проходимо $h = 0(\log_t n)$ вершин, адже у B-дереві, зрозуміло, більшість вершин є листковими. Для знаходження наступної вершини для пошуку, ми використовуємо однорідний бінарний пошук, часова складність якого $O(n) = O(\log_2 n), n \in [t, 2t] \Rightarrow O(t) = O(\log_2 t)$. Таким чином, часова складність пошуку у B-дереві $O(n) = O(h \cdot \log_2 t) = O(\log_t n \cdot \log_2 t) = O(\ln t \ln n)$.

3.3 Програмна реалізація

3.3.1 Вихідний код

import random

```
class BTreeNode:
    def __init__(self, t, leaf=False):
        self.t = t
        self.leaf = leaf
        self.keys = []
        self.children = []

    def split_child(self, i, child):
        new_child = BTreeNode(t=self.t, leaf=child.leaf)
        self.children.insert(i + 1, new_child)
        new_child.keys = child.keys[self.t:]
        child.keys = child.keys[:self.t]
        if not child.leaf:
            new_child.children = child.children[self.t:]
            child.children = child.children[:self.t]

    def insert_nonfull(self, k):
        i = len(self.keys) - 1
        if self.leaf:
```

```
            self.keys.append(0)
            while i >= 0 and k < self.keys[i]:
                self.keys[i + 1] = self.keys[i]
                i -= 1
            self.keys[i + 1] = k
        else:
            while i >= 0 and k < self.keys[i]:
                i -= 1
            i += 1
            if len(self.children[i].keys) == (2 * self.t - 1):
                self.split_child(i, self.children[i])
                if k > self.keys[i]:
                    i += 1
            self.children[i].insert_nonfull(k)

    def insert(self, k):
        if len(self.keys) == (2 * self.t - 1):
            new_root = BTreeNode(t=self.t)
            new_root.children.append(self)
            new_root.split_child(0, self)
            new_root.insert_nonfull(k)
            return new_root
        else:
            self.insert_nonfull(k)
            return self

    def search(self, k):
        i = 0
        while i < len(self.keys) and k > self.keys[i]:
            i += 1
        if i < len(self.keys) and k == self.keys[i]:
            return self
        elif self.leaf:
            return None
        else:
            return None if i == len(self.children) else
self.children[i].search(k)

    def merge(self, i):
        child = self.children[i]
        sibling = self.children[i + 1]
        child.keys.append(self.keys[i])
        child.keys.extend(sibling.keys)
        if not child.leaf:
            child.children.extend(sibling.children)
        self.keys.pop(i)
        self.children.pop(i + 1)
        return child

    def borrow_from_prev(self, i):
        child = self.children[i]
        sibling = self.children[i - 1]
        child.keys.insert(0, self.keys[i - 1])
        if not child.leaf:
            child.children.insert(0, sibling.children[-1])
            self.keys[i - 1] = sibling.keys.pop(-1)
            if len(sibling.keys) < self.t:
                return sibling
            else:
                return None

    def borrow_from_next(self, i):
        child = self.children[i]
        sibling = self.children[i + 1]
```

```
        child.keys.append(self.keys[i])
        if not child.leaf:
            child.children.append(sibling.children[0])
        self.keys[i] = sibling.keys.pop(0)
        if len(sibling.keys) < self.t:
            return sibling
        else:
            return None

    def delete_at_leaf(self, i):
        self.keys.pop(i)
        if len(self.keys) < self.t:
            return self
        else:
            return None

    def delete_at_internal(self, i):
        if self.children[i].leaf:
            predecessor = self.children[i].keys[-1]
            self.keys[i] = predecessor
            self.children[i] = self.children[i].delete(predecessor)
        else:
            predecessor = self.children[i].get_predecessor()
            self.keys[i] = predecessor
            self.children[i].delete(predecessor)

    def delete(self, k):
        i = 0
        while i < len(self.keys) and k > self.keys[i]:
            i += 1
        if i < len(self.keys) and k == self.keys[i]:
            if self.leaf:
                return self.delete_at_leaf(i)
            else:
                return self.delete_at_internal(i)
        elif self.leaf:
            return None
        else:
            child_to_delete = self.children[i]
            if len(child_to_delete.keys) == self.t - 1:
                left_sibling = None if i == 0 else self.children[i - 1]
                right_sibling = None if i == len(self.children) - 1 else
self.children[i + 1]
                if left_sibling and len(left_sibling.keys) > self.t - 1:
                    child_to_delete.borrow_from_prev(i)
                elif right_sibling and len(right_sibling.keys) > self.t - 1:
                    child_to_delete.borrow_from_next(i)
                else:
                    if left_sibling:
                        self.children[i - 1] = self.merge(i - 1)
                        child_to_delete = self.children[i - 1]
                    else:
                        self.children[i] = self.merge(i)
                        child_to_delete = self.children[i]
            return child_to_delete.delete(k)
        return None


class BTree:
    def __init__(self, t):
        self.root = BTreeNode(t=t, leaf=True)

    def insert(self, k):
        new_root = self.root.insert(k)
```

```
        if new_root:
            self.root = new_root

    def search(self, k):
        return self.root.search(k)

    def delete(self, k):
        self.root.delete(k)
        if len(self.root.keys) == 0 and len(self.root.children) > 0:
            self.root = self.root.children[0]

    def edit(self, old_key, new_key):
        if self.search(old_key) == None:
            print("Key not found in the tree")
            return
        self.delete(old_key)
        self.insert(new_key)
```

3.3.2 Приклади роботи

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

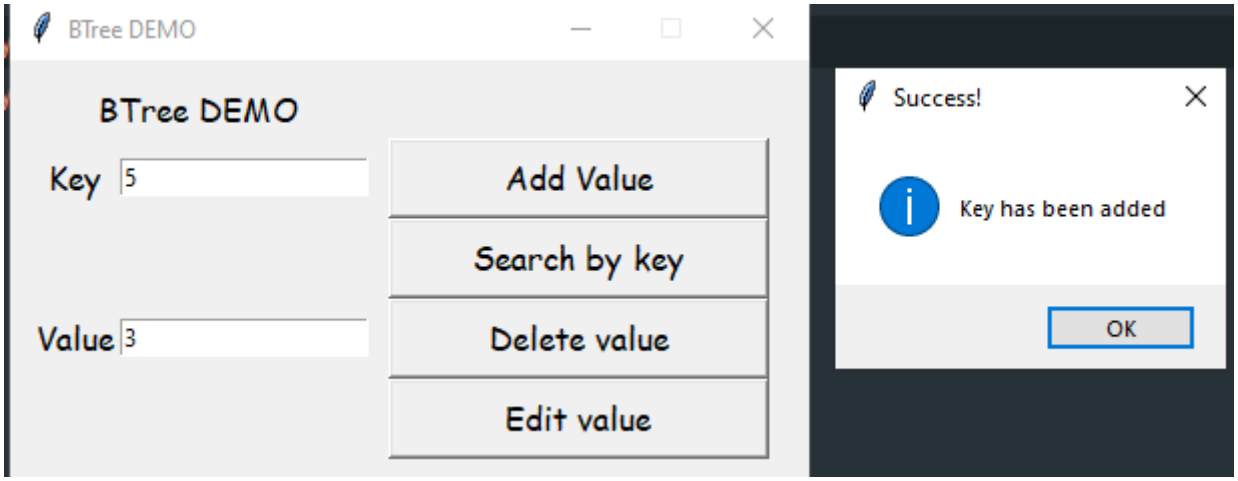

Рисунок 3.1 –Додавання запису

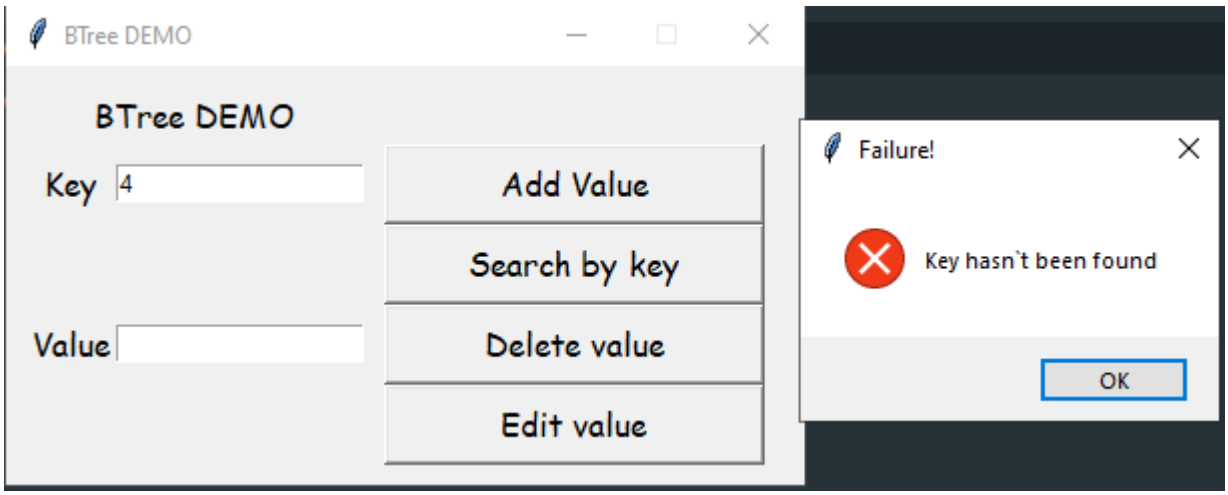

Рисунок 3.2 – Пошук запису

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	15
2	13
3	14
4	15
5	13
6	10
7	13
8	12
9	14
10	13
Середнє	13,2

ВИСНОВОК

В рамках лабораторної роботи було записано алгоритми пошуку, додавання, видалення і редагування запису у B-дереві із t=25 за допомогою псевдокоду. Була записана часова складність пошуку в асимптотичних оцінках, яка склала $O(n) = O(\ln t \ln n)$.

Було виконано програмну реалізацію невеликої СУБД з графічним інетрфейсом користувача з функціями пошуку, додавання, видалення та редагування записів використовучи B-дерево із t=25.

Було заповнено базу випадковими значеннями до 10000 і зафіксовано середнє число порівнянь, яке склало 13,2.

Було зроблено висновок, що B-дерево є доволі ефективним способом зберігання даних, особливо якщо доступ до них здійснюється фізичними блоками, завдяки своїй малій висоті і збалансованості, які підтримуються під час всіх операцій зі зміни даних в дереві.

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.