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”

Варіант 25

Виконав(ла) *ІП-15, Тонконог В.В.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Ахаладзе І.Е,* _____
(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

Додавання елементу

Add(data)

Початок

Присвоїти Root = Add(Root, data)

Кінець

Add(current, data)

Початок

Якщо current == null **то**

Присвоїти current data

Повернути current

Все Якщо

Інакше Якщо data.key < current.key, **то**

Присвоїти current.Left результат Add(current.Left, data)

Присвоїти current результат BalanceTree(current)

Все Інакше

Інакше Якщо data.key > current.key

Присвоїти current.Right результат Add(current.Right, data)

Присвоїти current результат BalanceTree(current)

Все Інакше

Повернути current

Кінець

ВИДАЛЕННЯ

Delete(target)

Початок

Присвоїти Root = Delete(Root, target)

Кінець

Delete(current, target)

Початок

Оголошуємо parent

Якщо current == null **то**

Повернути null

Все якщо

Інакше

Якщо то target < current.Key

current.Left = Delete(current.Left, target)
Присвоїти current результат BalanceTree(current)

Все Якщо

Інакше Якщо target > current.Key **то**

current.Right = Delete(current. Right, target)
Присвоїти current результат BalanceTree(current)

Все Інакше Якщо

Інакше

Якщо current.Right != null **то**

Присвоїти parent = current.Right

Поки current. Left!= null

Присвоїти parent = parent.Left;

Все поки

current.Key = parent. Key

current.Right = Delete(current.Right,

parent.Row.RowId)

Присвоїти current результат BalanceTree(current

Все Якщо

Інакше

Повернути current. Left

Все Інакше

Все Інакше

Все Інакше

Повернути current

Кінець

ПОШУК

Find(key)

Початок

Повернути Find(key, Root)

Кінець

Find(key, node)

Початок

Якщо node == null

Повернути null

Все Якщо

Якщо key < node.Key **то**

Присвоїти node = Find(key, node.Left)

Все якщо

Якщо key > node.Key **то**

Присвоїти node = Find(key, node.Right)

Все якщо

Повернути node

Кінець

РЕДАГУВАННЯ

Edit(row)

Початок

Присвоїти result = Find(row.RowId)

Присвоїти result.Value = row.Value

Кінець

3.2 Часова складність

Часова складність для даної структури даних є однаковою для операцій пошуку, вставки, видалення, редагування, та складає O(log n) через те, що висота дерева максимум рівна значенню log n + 1, тому що ми маємо завжди збалансоване дерево і це означає що на кожному

наступному рівні кількість вершин подвоюється. Оскільки на кожний ітерації ми заглиблюємось на одиницю глибини, то для проходу до потрібної вершини потрібно здійснити максимум log n + 1 ітерацій.

3.3 Програмна реалізація

3.3.1 Вихідний код

```
public class Row
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int RowId { get; set; }
         [StringLength(30, ErrorMessage = "Value has to be less than 30
characters!")]
        public string Value { get; set; }
    }
    public class Node
    {
        public Row Row { get; set; }
        public Node Left { get; set; }
        public Node Right { get; set; }
        public Node Parent { get; set; }
        public Node() { }
        public Node(Node parent, Row row)
        {
            Parent = parent;
            Row = row;
        }
    }
    public class AVL
    {
        Node Root;
        public AVL(INodeRepository repos)
        {
            if (repos.Nodes != null)
            {
                foreach (Row row in repos.Nodes)
                {
                    Add(new Node { Row = row });
                }
            }
        }
        public void Add(Node data)
        {
            Root = Add(Root, data);
        }
        private Node Add(Node current, Node n)
        {
            if (current == null)
            {
                current = n;
                return current;
            }
            else if (n.Row.RowId < current.Row.RowId)
            {
                current.Left = Add(current.Left, n);
                current = BalanceTree(current);
            }
            else if (n.Row.RowId > current.Row.RowId)
            {
                current.Right = Add(current.Right, n);
```

```
            current = BalanceTree(current);
        }
        return current;
    }
    private Node BalanceTree(Node current)
    {
        int b_factor = HeightDifference(current);
        if (b_factor > 1)
        {
            if (HeightDifference(current.Left) > 0)
            {
                current = SmallRight(current);
            }
            else
            {
                current = BigRight(current);
            }
        }
        else if (b_factor < -1)
        {
            if (HeightDifference(current.Right) > 0)
            {
                current = BigLeft(current);
            }
            else
            {
                current = SmallLeft(current);
            }
        }
        return current;
    }
    public void Delete(int target)
    {//and here
        Root = Delete(Root, target);
    }
    private Node Delete(Node current, int target)
    {
        Node parent;
        if (current == null)
        { return null; }
        else
        {
            //left subtree
            if (target < current.Row.RowId)
            {
                current.Left = Delete(current.Left, target);
                current = BalanceTree(current);
            }
            //right subtree
            else if (target > current.Row.RowId)
            {
                current.Right = Delete(current.Right, target);
                current = BalanceTree(current);
            }
            //if target is found
            else
            {
                if (current.Right != null)
                {
                    //delete its inorder successor
                    parent = current.Right;
                    while (parent.Left != null)
                    {
                        parent = parent.Left;
                    }
```

```
                current.Row.RowId = parent.Row.RowId;
                current.Row.Value = parent.Row.Value;
                current.Right = Delete(current.Right, parent.Row.RowId);
                current = BalanceTree(current);
            }
            else
            {   //if current.left != null
                return current.Left;
            }
        }
    }
    return current;
}
public Node Find(int key, ref int i)
{
    return Find(key, Root, ref i);
}

public Node Find(int key, Node node, ref int i)
{
    i++;
    if (node == null) return null;

    if (key.CompareTo(node.Row.RowId) < 0)
    {
        node = Find(key, node.Left, ref i);
    }
    else if (key.CompareTo(node.Row.RowId) > 0)
    {
        node = Find(key, node.Right, ref i);
    }

    return node;
}

public void Edit( Row row)
{
    int i = 0;
    var result = Find(row.RowId, ref i);
    result.Row.Value = row.Value;
}
public List<Row> ToList()
{
    List<Row> nodes = new List<Row>();
    ToList(Root, nodes);
    return nodes;
}
private void ToList(Node current, List<Row> list)
{
    if (current != null)
    {
        ToList(current.Left, list);
        list.Add(current.Row);
        ToList(current.Right, list);
    }
}
private int HeightDifference(Node current)
{
    int l = Height(current.Left);
    int r = Height(current.Right);
    return l - r;
}
private Node SmallLeft(Node parent)
{
    Node pivot = parent.Right;
```

```
        parent.Right = pivot.Left;
        pivot.Left = parent;
        return pivot;
    }
    private Node SmallRight(Node parent)
    {
        Node pivot = parent.Left;
        parent.Left = pivot.Right;
        pivot.Right = parent;
        return pivot;
    }
    private Node BigRight(Node parent)
    {
        Node pivot = parent.Left;
        parent.Left = SmallLeft(pivot);
        return SmallRight(parent);
    }
    private Node BigLeft(Node parent)
    {
        Node pivot = parent.Right;
        parent.Right = SmallRight(pivot);
        return SmallLeft(parent);
    }

    private int Height(Node node)
    {
        if (node == null)
        {
            return 0;
        }
        return 1 + Math.Max(Height(node.Left), Height(node.Right));
    }
```

3.3.1 Приклади роботи

На рисунках 3.2 і 3.3 показані приклади роботи програми для додавання і пошуку запису (ключ == 10001).

На рисунку 3.1 наведена початкова БД.

9993	9993	Edit
9994	9994	Edit
9995	9995	Edit
9996	9996	Edit
9997	9997	Edit
9998	9998	Edit
9999	9999	Edit
10000	10000	Edit

1 2 3 4 5 6 7 8 9 10

Рисунок 3.1 –Початкова БД

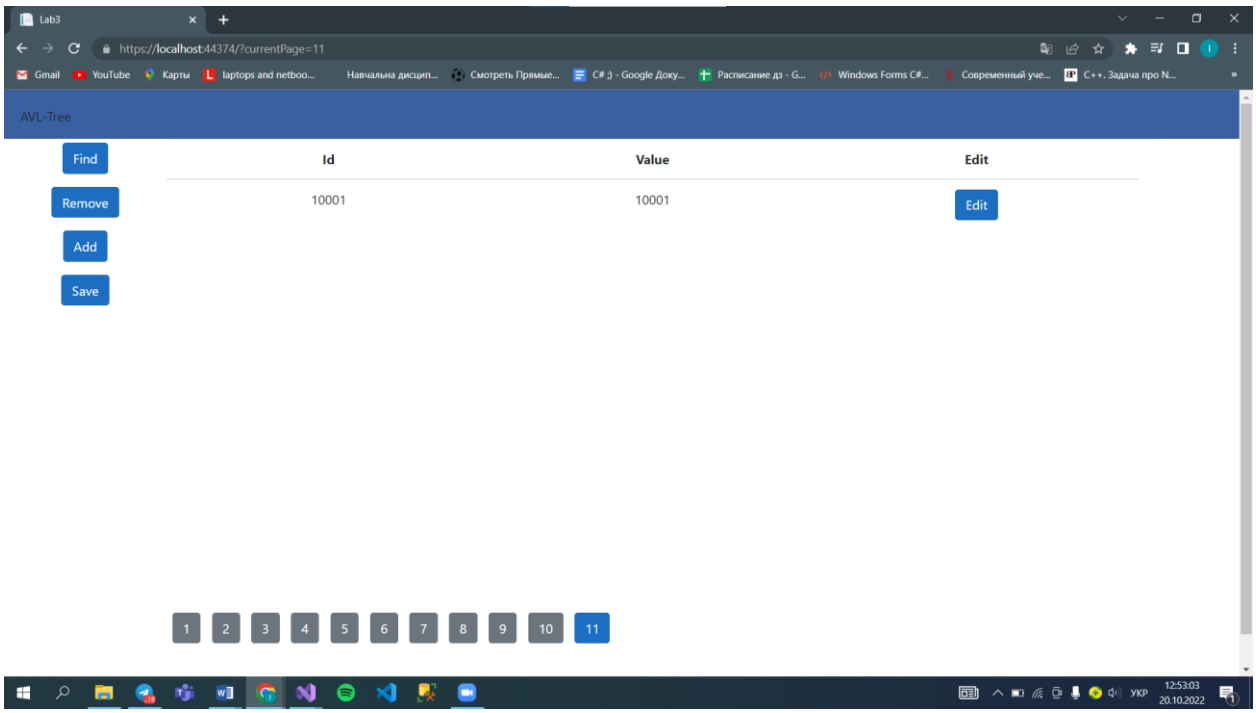

Рисунок 3.2 –Додавання запису

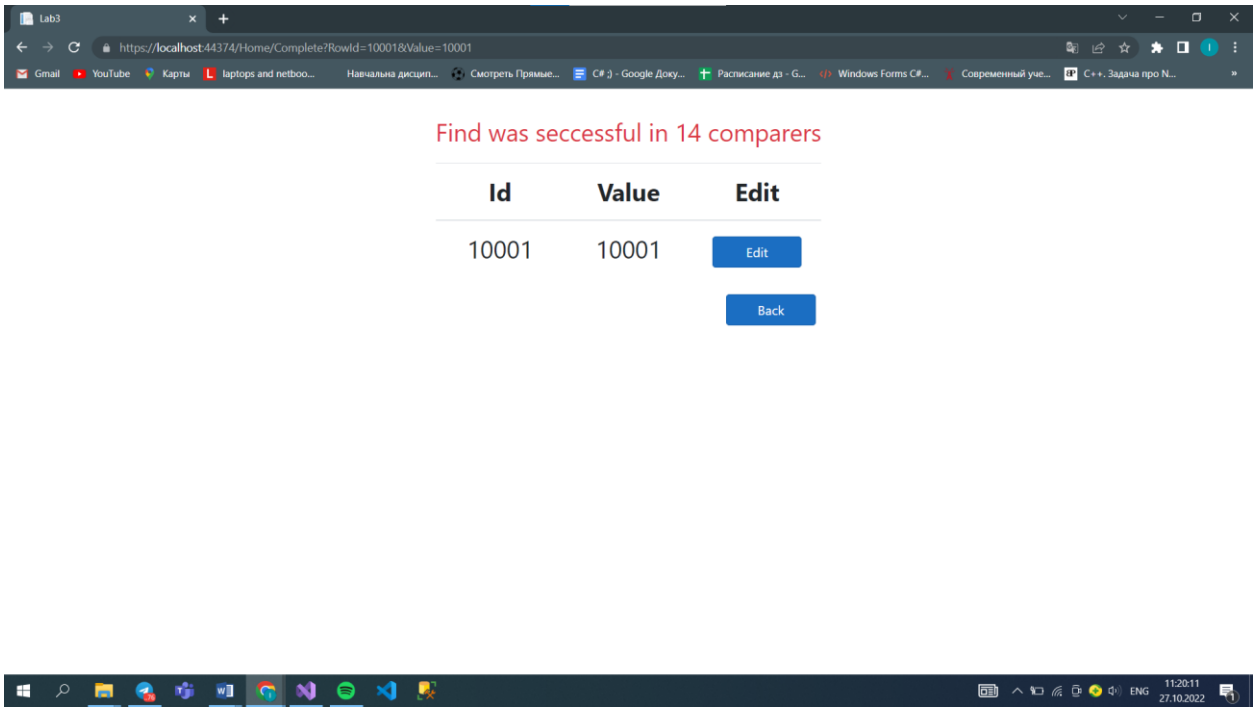

Рисунок 3.3 – Пошук запису

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

В результаті тестування пересвідились у тому, що для пошуку елемента нам необхідно максимум log n + 2 порівнянь, тут ми додаємо один до загальної висоти дерева через те, що ми можемо шукати неіснуючий елемент и для цього можемо спуститись на ще не заповнену глубину. Отже, часова складність додавання/ видалення/ редагування/ пошуку елемента рівна O(log n).

ВИСНОВОК

В рамках лабораторної роботи було створено базу даних на основі АВЛ-дерева. Виконавши дослідження роботи встановили що для пошуку в структурі даних необхідно максимум log n +1 порівнянь. Для вставки елемента, видалення, пошуку та редагування ми використовуємо O(log n) часу адже елемент шукається за O(log n) часу, а всі інші операції є константними в часі.

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.