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 Макарчук Лідія Олександрівна* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Сопов Олексій Олександрович* _____
(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 14

3.3 Програмна реалізація 14

3.3.1 Вихідний код 14

3.3.2 Приклади роботи 22

3.4 Тестування алгоритму 26

3.4.1 Часові характеристики оцінювання 26

ВИСНОВОК 27

КРИТЕРІЇ ОЦІНЮВАННЯ 28

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 зщільним 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 16

16	Червоно-чорне дерево

3.1 Псевдокод алгоритмів

Пошук

Search(Tree, key)

1. ПОЧАТОК
2. x = Tree.root
3. ПОКИ x != NIL && x.key != key ПОВТОРИТИ
   - 3.1. ЯКЩО key < x.key
     - 3.1.1. ТО x = x.left
     - 3.1.2. ІНАКШЕ ЯКЩО key < x.key ТО x = x.right
     - 3.1.3. ІНАКШЕ ПОВЕРНУТИ recordExistsError
4. ПОВЕРНУТИ x
5. КІНЕЦЬ

Додавання (insertion)

Insert(Tree, node)

1. ПОЧАТОК
2. y = NIL
3. x = Tree.root
4. ПОКИ x != NULL ПОВТОРИТИ
   - 4.1. y = x
   - 4.2. ЯКЩО node.key < x.key
     - 4.2.1. ТО x = x.left
     - 4.2.2. ІНАКШЕ x = x.right
5. node.parent = y

6. ЯКЩО y == NILL
    6.1. ТО Tree.root = node
    6.2. ІНАКШЕ ЯКЩО node.key < y.key
        6.2.1. ТО y.left = node
        6.2.2. ІНАКШЕ y.right = node
7. node.left = NIL
8. node.right = NIL
9. node.color = RED
10. Insert_Fixup(Tree, node)
11. КІНЕЦЬ

Insert_Fixup(Tree, node)

1. ПОЧАТОК
2. ПОКИ node. parent.color == RED ПОВТОРИТИ
    2.1. ЯКЩО node.parent == node.grandparent.left
        2.1.1. ТО
            2.1.1.1. y = node.grandparent.right
            2.1.1.2. ЯКЩО y.color == RED
                2.1.1.2.1. ТО
                    2.1.1.2.1.1. node.parent.color = BLACK
                    2.1.1.2.1.2. y.color = BLACK
                    2.1.1.2.1.3. node.grandparent.color = RED
                    2.1.1.2.1.4. node = node.grandparent
                2.1.1.2.2. ІНАКШЕ
                2.1.1.2.3. ЯКЩО node = node.parent.right
                    2.1.1.2.3.1. ТО node = node.parent
                    2.1.1.2.3.2. Left_Rotate(Tree, node)
                2.1.1.2.4. node.parent.color = BLACK
                2.1.1.2.5. node.grandparent.color = RED
                2.1.1.2.6. Right_Rotate(Tree, node.grandparent)

2.1.2. ІНАКШЕ

2.1.2.1. y = node.grandparent.left

2.1.2.2. ЯКЩО y.color == RED

2.1.2.2.1. ТО

2.1.2.2.1.1. node.parent.color = BLACK

2.1.2.2.1.2. y.color = BLACK

2.1.2.2.1.3. node.grandparent.color = RED

2.1.2.2.1.4. node = node.grandparent

2.1.2.2.2. ІНАКШЕ

2.1.2.2.3. ЯКЩО node = node.parent.left

2.1.2.2.3.1. ТО node = node.parent

2.1.2.2.3.2. Right_Rotate(Tree, node)

2.1.2.2.4. node.parent.color = BLACK

2.1.2.2.5. node.grandparent.color = RED

2.1.2.2.6. Left_Rotate(Tree, node.grandparent)

3. Tree.root.color = BLACK
4. КІНЕЦЬ

Видалення

Delete(Tree, node)

1. ПОЧАТОК
2. y = node
3. yOriginalColor = y.color
4. ЯКЩО node.left == NIL

4.1. x = node.right

4.2. Transplant(Tree, node, node.right)

5. ІНАКШЕ ЯКЩО node.right == NIL

5.1. x = node.left

5.2. Transplant(Tree, node, node. left)

6. ІНАКШЕ

6.1. y = Minimum(node.right)

6.2. yOriginalColor = y.color

6.3. x = y.right

6.4. ЯКЩО y.parent == node

6.4.1. ТО x.parent = y

6.4.2. ІНАКШЕ

6.4.2.1. Transplant(Tree, y, y.right)

6.4.2.2. y.right = node.right

6.4.2.3. y.right.parent = y

6.5. Transplant(Tree, node, y)

6.6. y.left = node.left

6.7. y.left.parent = y

6.8. y.color = node.color

7. ЯКЩО yOriginalColor == BLACK

7.1. ТО Delete_Fixup(Tree, x)

8. КІНЕЦЬ

Delete_Fixup(Tree, x)

1. ПОЧАТОК
2. ПОКИ x != Tree.root && x.color == BLACK ПОВТОРИТИ

2.1. ЯКЩО x == x.parent.left

2.1.1. ТО

2.1.1.1. w = x.parent.right

2.1.1.2. ЯКЩО w.color == RED

2.1.1.2.1. ТО

2.1.1.2.1.1. w.color = BLACK

2.1.1.2.1.2. x.parent.color = RED

2.1.1.2.1.3. Left_Rotate(Tree, x.parent)

2.1.1.2.1.4. w = x.parent.right

2.1.1.3. ЯКЩО w.left.color == BLACK && w.right.color == BLACK

2.1.1.3.1. ТО

2.1.1.3.1.1. w.color = RED

2.1.1.3.1.2. x = x.parent

2.1.1.3.2. ІНАКШЕ

2.1.1.3.2.1. ЯКЩО w.right.color == BLACK

2.1.1.3.2.1.1. ТО

2.1.1.3.2.1.1.1. w.left.color = BLACK

2.1.1.3.2.1.1.2. w.color = RED

2.1.1.3.2.1.1.3. Right_Rotate(Tree, w)

2.1.1.3.2.1.1.4. w = x.parent.right

2.1.1.3.2.2. w.color = x.parent.color

2.1.1.3.2.3. x.parent.color = BLACK

2.1.1.3.2.4. w.right.color = BLACK

2.1.1.3.2.5. Left_Rotate(Tree, x.parent)

2.1.1.3.2.6. x = Tree.root

2.1.2. ІНАКШЕ

2.1.2.1. w = x.parent.left

2.1.2.2. ЯКЩО w.color == RED

2.1.2.2.1. ТО

2.1.2.2.1.1. w.color = BLACK

2.1.2.2.1.2. x.parent.color = RED

2.1.2.2.1.3. Right_Rotate(Tree, x.parent)

2.1.2.2.1.4. w = x.parent.left

2.1.2.3. ЯКЩО w.right.color == BLACK && w.left.color == BLACK

2.1.2.3.1. ТО

2.1.2.3.1.1. w.color = RED

2.1.2.3.1.2. x = x.parent

2.1.2.3.2. ІНАКШЕ

2.1.2.3.2.1. ЯКЩО w.left.color == BLACK

2.1.2.3.2.1.1. ТО

2.1.2.3.2.1.1.1. w.right.color = BLACK

2.1.2.3.2.1.1.2. w.color = RED

2.1.2.3.2.1.1.3. Left_Rotate(Tree, w)

2.1.2.3.2.1.1.4. w = x.parent.left

2.1.2.3.2.2. w.color = x.parent.color

2.1.2.3.2.3. x.parent.color = BLACK

2.1.2.3.2.4. w.left.color = BLACK

2.1.2.3.2.5. Right_Rotate(Tree, x.parent)

2.1.2.3.2.6. x = Tree.root

3. x.color = BLACK
4. КІНЕЦЬ

Transplant(Tree, u, v)

1. ПОЧАТОК
2. ЯКЩО u.parent == NIL

   2.1. Tree.root = v
3. ІНАКШЕ ЯКЩО u == u.parent.left

   3.1. u.parent.left = v
4. ІНАКШЕ u.parent.right = v
5. v.parent = u.parent
6. КІНЕЦЬ

Редагування

Update(Tree, key, newData)

1. ПОЧАТОК
2. node = Search(Tree, key)
3. ЯКЩО node != NIL ТО node.data = newData

4. КІНЕЦЬ

Допоміжні методи

Left_Rotate(Tree, x)

1. ПОЧАТОК
2. y = x.right
3. x.rigth = y.left
4. ЯКЩО y.left != NIL
    4.1. y.left.parent = x
5. y.parent = x.parent
6. ЯКЩО x.parent == NIL
    6.1. ТО Tree.root = y
    6.2. ІНАКШЕ ЯКЩО x == x.parent.left
        6.2.1. ТО x.parent.left = y
        6.2.2. ІНАКШЕ x.parent.right = y
7. y.left = x
8. x.parent = y
9. КІНЕЦЬ

Right_Rotate(Tree, x)

1. ПОЧАТОК
2. y = x.left
3. x. left = y.right
4. ЯКЩО y. right!= NIL
    4.1. y.right.parent = x
5. y.parent = x.parent
6. ЯКЩО x.parent == NIL
    6.1. ТО Tree.root = y
    6.2. ІНАКШЕ ЯКЩО x == x.parent.right
        6.2.1. ТО x.parent.right = y

6.2.2. ІНАКШЕ x.parent.left = y

7. y.right = x
8. x.parent = y

КІНЕЦЬ

3.2 Часова складність пошуку

Часова складність пошуку для червоно-чорного дерева пов'язана із його висотою. Для визначення висоти скористаємося поняттям чорної висоти червоно-чорного дерева (тобто чорна висота для кореня). Будь-який вузол x може мати щонайменше наступну кількість нащадків (прямих та непрямих у сумі) $2^{bh(x)} - 1$, де $bh(x)$ – чорна висота вузла x. Визначимо $bh(root)$: за визначенням хоча б половина вузлів у будь-якому шляху, не включаючи корінь, повинна бути чорного кольору, тобто $bh(root) \geq h/2$. Тоді $n \geq 2^{h/2} - 1$. Звідси маємо $h \leq 2\log_2(n-1)$. Таким чином при пошуку ми маємо переглянути не більше $h \leq 2\log_2(n-1)$ вузлів. Використовуючи асимптотику запишемо час пошуку вузла у червоно-чорному дереві: O(log(n)).

3.3 Програмна реалізація

3.3.1 Вихідний код

Модуль DBManager.cs

```
using RedBlackTreeAlgo.FileStructure;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using RedBlackTreeAlgo.Exceptions;
using System.Reflection;
using System.IO;
using System.Xml.Linq;
using System.Text.RegularExpressions;

namespace RedBlackTreeAlgo.DatabaseManager
{
   public class DBManager
   {
      private string? currDB;

      private BufferManager buffManager;
      public DBManager(string name)
      {
```

```
            currDB = name;
            byte[] md = ReadMetadata(name + "Meta");
            buffManager = new BufferManager(name, md);
        }


        public void InsertData(string data)
        {
            int key = Convert.ToInt32(data.Split(',')[0]);
            byte[] dataBytes = buffManager.GetDataBytesFromString(data);
            if (dataBytes.Length == Record.dataSpace)
                Insert(key, dataBytes);
            else
                throw new WrongDataFormat("Data has wrong format");
        }
        private void Insert(int key, byte[] data)
        {
            Record y = new Record(null);// y = nill
            Record x = buffManager.getRoot();
            while(!x.IsNill())
            {
                y = x;
                if (key < x.Key)
                    x = buffManager.getLeft(x);
                else if (key > x.Key)
                    x = buffManager.getRight(x);
                else
                    throw new RecordAlreadyExists("Record with key "+ key + " already Exsists");
            }

            Page currPage = buffManager.getCurrPage();
            if (!currPage.isEnoughSpace())
                currPage = buffManager.CreateNewPage();
            Record record = new Record(key, data, currPage.Number, currPage.Position);  //creating record with reference to
written data
            currPage.AddRecord(record); //adding record to page records
            if (!y.IsNill())
            {
                buffManager.setParent(record, y);   //set record parent to y
                if(key < y.Key)
                    buffManager.setLeft(y, record);
                else
                    buffManager.setRight(y, record);
            }
            else
                buffManager.setRoot(record);//root = record

            bool flag =  InsertFixup(record);
            buffManager.CleanPagesAndWriteRoot();
        }
        private bool InsertFixup(Record record)
        {
          while( buffManager.getParent(record).Color == Color.RED)
            {
                if (buffManager.getParent(record) == buffManager.getLeft(buffManager.getGrandparent(record)) )   //if parent
is a left child
                {
                    Record y = buffManager.getRight(buffManager.getGrandparent(record));  //uncle
                    if (y.Color == Color.RED)   //case 1 (uncle is RED). Solution: recolor
                    {
                        buffManager.setColor(buffManager.getParent(record), Color.BLACK);   //set parent color to black
                        buffManager.setColor(y, Color.BLACK);   //set uncle color to black
                        buffManager.setColor(buffManager.getGrandparent(record), Color.RED);    //set grandparent color to red
                        record = buffManager.getGrandparent(record);    //record = record.Grandparent
                    }
                    else
```

```
                {
                    //case 2 (uncle is black, triangle). Solution: transform case 2 into case 3 (rotate)
                    if (record == buffManager.getRight(buffManager.getParent(record) ))
                    {
                        record = buffManager.getParent(record);
                        buffManager.LeftRotate(record);
                    }
                    //case 3 (uncle is black, line). Solution: recolor and rotate
                    buffManager.setColor(buffManager.getParent(record), Color.BLACK);
                    buffManager.setColor(buffManager.getGrandparent(record), Color.RED);
                    buffManager.RightRotate(buffManager.getGrandparent(record));//RightRotate(node.G)
                }
            }
            else //if parent is a right child
            {
                Record y = buffManager.getLeft(buffManager.getGrandparent(record));  //uncle
                if (y.Color == Color.RED)   //case 1 (uncle is RED). Solution: recolor
                {
                    buffManager.setColor(buffManager.getParent(record), Color.BLACK);   //set parent color to black
                    buffManager.setColor(y, Color.BLACK);   //set uncle color to black
                    buffManager.setColor(buffManager.getGrandparent(record), Color.RED);    //set grandparent color to red
                    record = buffManager.getGrandparent(record);    //record = record.Grandparent
                }
                else
                {
                    //case 2 (uncle is black, triangle). Solution: transform case 2 into case 3 (rotate)
                    if (record == buffManager.getLeft(buffManager.getParent(record)) ) //Record.AreEqual(record,
buffManager.getLeft(buffManager.getParent(record)))//
                    {
                        record = buffManager.getParent(record);
                        buffManager.RightRotate(record);
                    }
                    //case 3 (uncle is black, line). Solution: recolor and rotate
                    buffManager.setColor(buffManager.getParent(record), Color.BLACK);
                    buffManager.setColor(buffManager.getGrandparent(record), Color.RED);
                    buffManager.LeftRotate(buffManager.getGrandparent(record));//RightRotate(node.G)
                }
            }
        }
        buffManager.setColor(buffManager.getRoot(), Color.BLACK);   //case 0 (node is root)
        return true;
                }

public string? SearchData(int key, out int comparisonNumber)
    {
        comparisonNumber = 0;
        string? result = null;
        byte[]? dataBytes = null;
        Record record = Search(key, out comparisonNumber);
        if (!record.IsNil())
            dataBytes = record.Data;
        if (dataBytes != null)
        {
            result = buffManager.GetDataStringFromBytes(dataBytes);
        }
        return result;
    }
    private Record Search(int key, out int comparisonNumber)
    {
        comparisonNumber = 0;
        Record x = buffManager.getRoot();
        while (!x.IsNil()  && x.Key != key)
        {
            if (key < x.Key)
                x = buffManager.getLeft(x);
            else
                x = buffManager.getRight(x);
```

```
            comparisonNumber++;
        }
        comparisonNumber++;
        return x;
    }
    public bool Delete(int key)
    {
        bool flag = true; int dummy;
        Record record = Search(key, out dummy);
        if (record.IsNill()) return false; //if there is no such an element
        Record x, y = record;
        Color yOriginalColor = y.Color;
        if (buffManager.getLeft(record).IsNill())// right child or no children
        {
            x = buffManager.getRight(record);
            buffManager.Transplant(record, buffManager.getRight(record));
        }
        else if (buffManager.getRight(record).IsNill())// left child
        {
            x = buffManager.getLeft(record);
            buffManager.Transplant(record, buffManager.getLeft(record));
        }
        else//has both children
        {
            y = buffManager.Minimum(buffManager.getRight(record));
            yOriginalColor = y.Color;
            x = buffManager.getRight(y);
            if (buffManager.getParent(y) == record)// if y is a direct child for record
                buffManager.setParent(x, y);
            else
            {
                buffManager.Transplant(y, buffManager.getRight(y));
                buffManager.setRight(y, buffManager.getRight(record)); //insert successor instead of node
                buffManager.setParent(buffManager.getRight(y), y);
            }
            buffManager.Transplant(record, y);
            buffManager.setLeft(y, buffManager.getLeft(record));
            buffManager.setParent(buffManager.getLeft(y), y);
            buffManager.setColor(y, record.Color);
        }
        if (yOriginalColor == Color.BLACK)
            flag = Delete_Fixup(x);
        record.DeleteRecordData();
        buffManager.getPageWithNumber(record.recordPage).IsDirty = true;
        buffManager.CleanPagesAndWriteRoot();
        return flag;
    }
    private bool Delete_Fixup(Record x)
    {
        Record w; //sibling
        while(x != buffManager.getRoot() && x.Color == Color.BLACK )
        {
            if (x == buffManager.getLeft(buffManager.getParent(x)) )//if left child
            {
                w = buffManager.getRight(buffManager.getParent(x));
                //case 1
                if (w.Color == Color.RED)
                {
                    buffManager.setColor(w, Color.BLACK);
                    buffManager.setColor(buffManager.getParent(x), Color.RED);
                    buffManager.LeftRotate(buffManager.getParent(x));
                    w = buffManager.getRight(buffManager.getParent(x));
                }
                //case 2
                if (!w.IsNill() && buffManager.getLeft(w).Color == Color.BLACK && buffManager.getRight(w).Color ==
Color.BLACK)
```

```
                {
                    buffManager.setColor(w, Color.RED);
                    x = buffManager.getParent(x);
                }
                else
                {
                    if (!w.IsNill() && buffManager.getRight(w).Color == Color.BLACK)
                    {
                        buffManager.setColor(buffManager.getLeft(w), Color.BLACK);
                        buffManager.setColor(w, Color.RED);
                        buffManager.RightRotate(w);
                        w = buffManager.getRight(buffManager.getParent(w));
                    }
                    buffManager.setColor(w, buffManager.getParent(x).Color);
                    buffManager.setColor(buffManager.getParent(x), Color.BLACK);
                    buffManager.setColor(buffManager.getRight(w), Color.BLACK);
                    buffManager.LeftRotate(buffManager.getParent(x));
                    x = buffManager.getRoot();
                }
            }
            else //if right child
            {
                w = buffManager.getLeft(buffManager.getParent(x));
                if (w.Color == Color.RED)
                {
                    buffManager.setColor(w, Color.BLACK);
                    buffManager.setColor(buffManager.getParent(x), Color.RED);
                    buffManager.RightRotate(buffManager.getParent(x));
                    w = buffManager.getLeft(buffManager.getParent(x));
                }
                if (!w.IsNill() && buffManager.getRight(w).Color == Color.BLACK && buffManager.getLeft(w).Color ==
Color.BLACK)
                {
                    buffManager.setColor(w, Color.RED);
                    x = buffManager.getParent(x);
                }
                else
                {
                    if (!w.IsNill() && buffManager.getLeft(w).Color == Color.BLACK)
                    {
                        buffManager.setColor(buffManager.getRight(w), Color.BLACK);
                        buffManager.setColor(w, Color.RED);
                        buffManager.LeftRotate(w);
                        w = buffManager.getLeft(buffManager.getParent(w));
                    }
                    buffManager.setColor(w, buffManager.getParent(x).Color);
                    buffManager.setColor(buffManager.getParent(x), Color.BLACK);
                    buffManager.setColor(buffManager.getLeft(w), Color.BLACK);
                    buffManager.RightRotate(buffManager.getParent(x));
                    x = buffManager.getRoot();
                }
            }
        }
        buffManager.setColor(x, Color.BLACK);
        return true;
                }
public bool UpdateData(string data)
    {
        int key = Convert.ToInt32(data.Split(',')[0]);
        byte[] dataBytes = buffManager.GetDataBytesFromString(data);
        if (dataBytes.Length == Record.dataSpace)
        {
            return Update(key, dataBytes);
        }
        return false;
    }
```

```
        private bool Update(int key, byte[] data)
        {
            int dummy;
            Record record = Search(key, out dummy);
            if (record.IsNill()) return false; //if there is no such an element
            record.Data = data;
            buffManager.getPageWithNumber(record.recordPage).IsDirty = true;
            buffManager.CleanPagesAndWriteRoot();
            return true;
        }
}
```

Модуль BufferManager.cs

```
using RedBlackTreeAlgo.FileStructure;
using System;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Reflection.PortableExecutable;
using System.Text;
using System.Threading.Tasks;
using System.Xml.Linq;

namespace RedBlackTreeAlgo.DatabaseManager
{
    public class BufferManager
    {
        private Dictionary<int, Page> bufferPool;   //page number, page
        private string currDB;
        private List<(int typeSize, char t, string cName)> colmns;


        private static int offsetFromStart = sizeof(int) * 4;   //indicates the pages start
        private static int pageHeaderSize = Page.pageHeaderSize;
        //
        private int currPageNumb;
        // dataSpace;//
        private int rootPage;
        private int rootOffset;

        private bool needToWriteCurrPage; //start of file
        private bool needToWriteRoot;
        private Record _root;

        public BufferManager(string dbName, byte[] metadata)
        {
            currDB = dbName;

            bufferPool = new Dictionary<int, Page>();
            using (var stream = File.Open(currDB, FileMode.Open))
            {
                using(var binaryReader = new BinaryReader(stream))
                {
                    currPageNumb = binaryReader.ReadInt32();//current page
                    Record.dataSpace = binaryReader.ReadInt32();//space of data in each node
                    Record.RecordSize = sizeof(int) * 9 + Record.dataSpace;
                    rootPage = binaryReader.ReadInt32();
                    rootOffset = binaryReader.ReadInt32();
                }
            }
            colmns = Parser.MetadataToData(metadata);//get columns sizes, types and names
            _root = getRecordFromPage(rootPage, rootOffset);
            needToWriteRoot = false;
        }
```

```
        // pages
        //----------------------------------------------------------------------

        //-------------------------------------------------------------------------------
        //records manipulations
        public Record getRoot()
        {
            return _root;
        }
        public void setRoot(Record? record)
        {
            if(record != null && !record.IsNill())
            {
                rootPage = record.recordPage;
                rootOffset = record.recordOffset;
                _root = record;
            }
            else
            {
                rootPage = 0;
                rootOffset = 0;
                if (record != null)
                    _root = record;
                else
                    _root = new Record(null);
            }
            needToWriteRoot = true;
        }

        public void setRecordOnPage(int pageNum, Record record)
        {
            bufferPool[pageNum].setRecord(pageHeaderSize, record);
                    }

    //parent
        public Record getParent(Record record)
        {
            if (record == null)
                return new Record(null);
            if (record.IsNill())
                return record.P;
            return getRecordFromPage(record.ParentPage, record.ParentOffset);
        }
        public void setParent(Record record, Record parent)
        {
            if (!parent.IsNill())
            {
                record.ParentPage = parent.recordPage;
                record.ParentOffset = parent.recordOffset;
                if (record.IsNill())
                    record.P = parent;
            }
            else
            {
                record.ParentPage = 0;
                record.ParentOffset = 0;
                record.P = parent;
            }
            bufferPool[record.recordPage].IsDirty = true;
        }
        public Record? getGrandparent(Record record)
        {
            return getParent(getParent(record));
        }

        //left
```

```
        public Record? getLeft(Record record)
        {
            if (record.IsNill())
                return null;
            if (record.LeftOffset == 0)
                return record.leftNill;
            return getRecordFromPage(record.LeftPage, record.LeftOffset);
        }
        public void setLeft(Record record, Record? successor)
        {
            if (successor != null && !successor.IsNill())
            {
                record.LeftPage = successor.recordPage;
                record.LeftOffset = successor.recordOffset;
            }
            else
            {
                record.LeftPage = 0;
                record.LeftOffset = 0;
                if (successor != null)
                    record.leftNill = successor;
                else
                    record.leftNill = new Record(record);
            }
            bufferPool[record.recordPage].IsDirty = true;
        }
        //right
        public Record? getRight(Record record)
        {
            if (record.IsNill())
                return null;
            if (record.RightOffset == 0)
                return record.rightNill;
            return getRecordFromPage(record.RightPage, record.RightOffset);
        }
        public void setRight(Record? record, Record successor)
        {
            if (successor != null && !successor.IsNill())
            {
                record.RightPage = successor.recordPage;
                record.RightOffset = successor.recordOffset;
            }
            else
            {
                record.RightPage = 0;
                record.RightOffset = 0;
                if (successor != null)
                    record.rightNill = successor;
                else
                    record.rightNill = new Record(record);
            }
            bufferPool[record.recordPage].IsDirty = true;
        }
        //color
        public void setColor(Record? record, Color color)
        {
            if (record != null && record.Color != color)
            {
                record.Color = color;
                bufferPool[record.recordPage].IsDirty = true;
            }
                    }
    public void LeftRotate(Record x)
        {
            Record y = getRight(x);
            setRight(x, getLeft(y)); // x.Right = y.Left
```

```
            if (getLeft(y)!=null)
                setParent(getLeft(y), x);   //y.Left.P = x
            setParent(y, getParent(x)); //y.P = x.P

            if (x.ParentOffset == 0) //if x.P == null
                setRoot(y); //set root to y
            else if (x == getLeft(getParent(x)) ) //x == x.P.Left//(x is left child)
                setLeft(getParent(x), y);//x.P.Left = y
            else //(x is right child)
                setRight(getParent(x), y);//x.P.Right = y
            setLeft(y, x); //y.Left = x
            setParent(x, y);
        }
        public void RightRotate(Record? x)
        {
            Record y = getLeft(x);
            setLeft(x, getRight(y)); // x.Left = y.Right
            if (getRight(y)!=null)
                setParent(getRight(y), x);   //y.Right.P = x
            setParent(y, getParent(x)); //y.P = x.P

            if (x.ParentOffset == 0) //if x.P == null
                setRoot(y); //set root to y
            else if (x == getRight(getParent(x)) ) //x == x.P.Right
                setRight(getParent(x), y);//x.P.Right = y
            else
                setLeft(getParent(x), y);//x.P.Left = y
            setRight(y, x);//y.Right = x
            setParent(x, y);
        }
        public void Transplant(Record u, Record v)
        {
            if (getParent(u).IsNill())
                setRoot(v);//root = v;
            else if(u == getLeft(getParent(u)) )
                setLeft(getParent(u), v);//u.P.Left = v;
            else
                setRight(getParent(u), v);//u.P.Right = v;

            setParent(v, getParent(u));//v.P = u.P;
        }
        public Record Minimum(Record record)
        {
            while (!getLeft(record).IsNill()) //getLeft(record) != null
            {
                record = getLeft(record);
            }
            return record;

        }

    }
}
```

3.3.2 Приклади роботи

На рисунках 3.1, 3.2, 3.3, 3.4, 3.5, 3.6 показані приклади роботи програми для створення бази даних, додавання, пошуку, зміни та видалення запису, а також графічне представлення ключів у створеній базі даних.

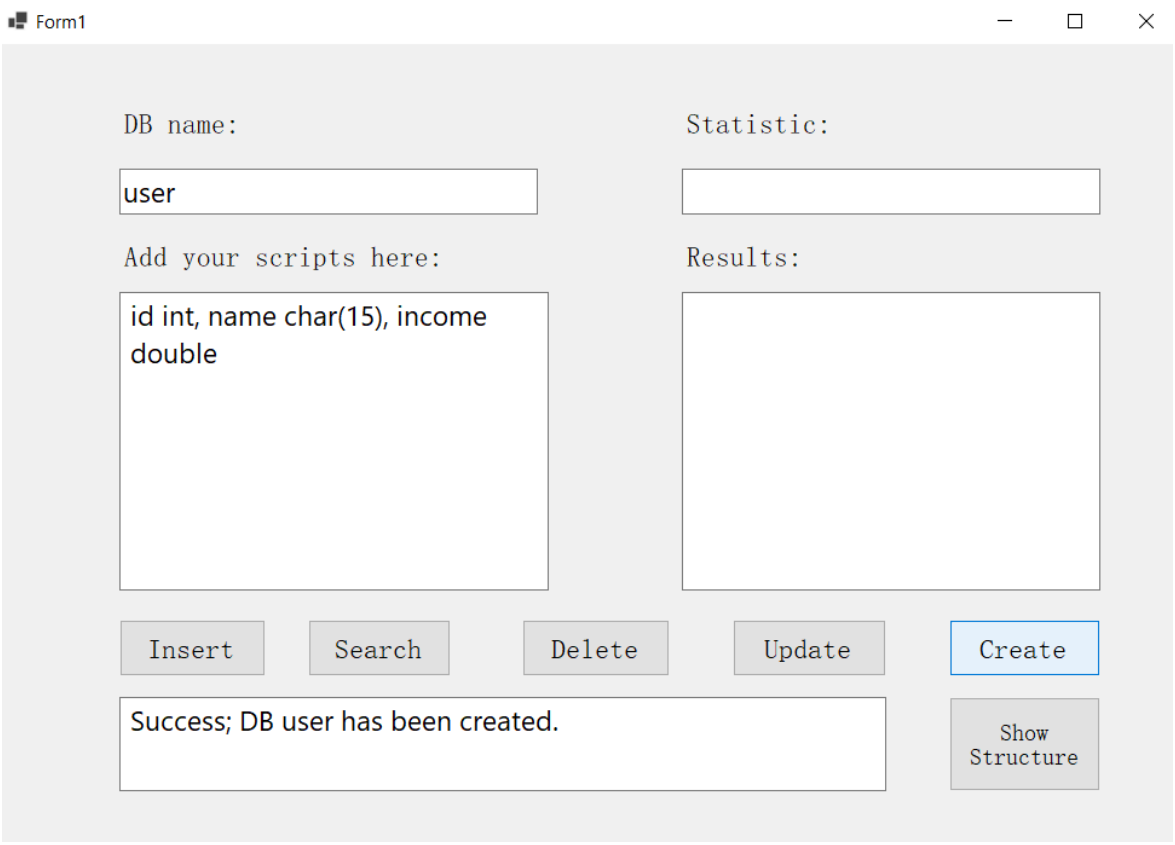

Рисунок 3.1 –Створення бази даних

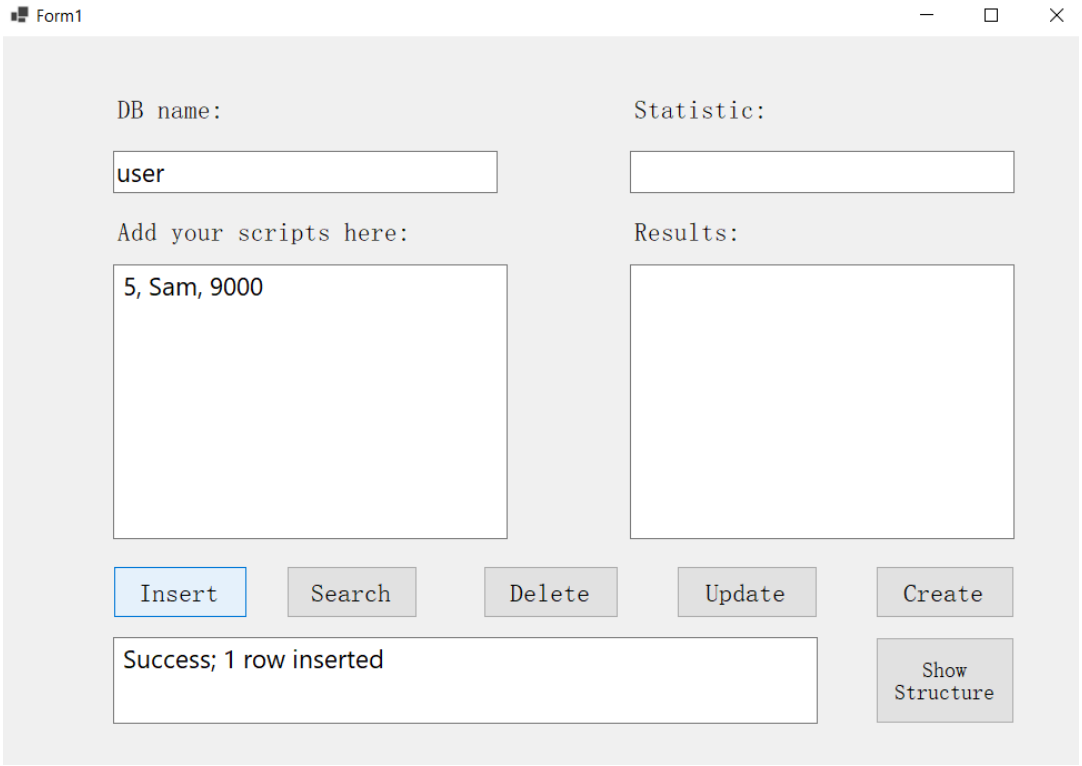

Рисунок 3.2 –Додавання запису

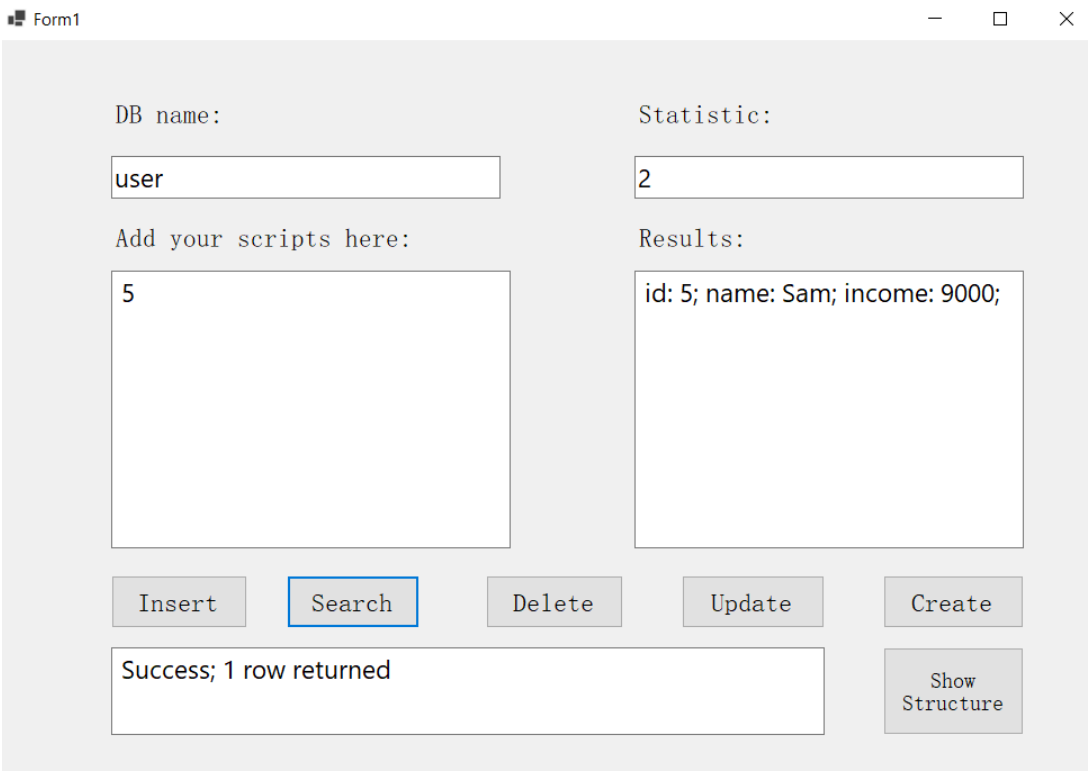

Рисунок 3.3 – Пошук запису

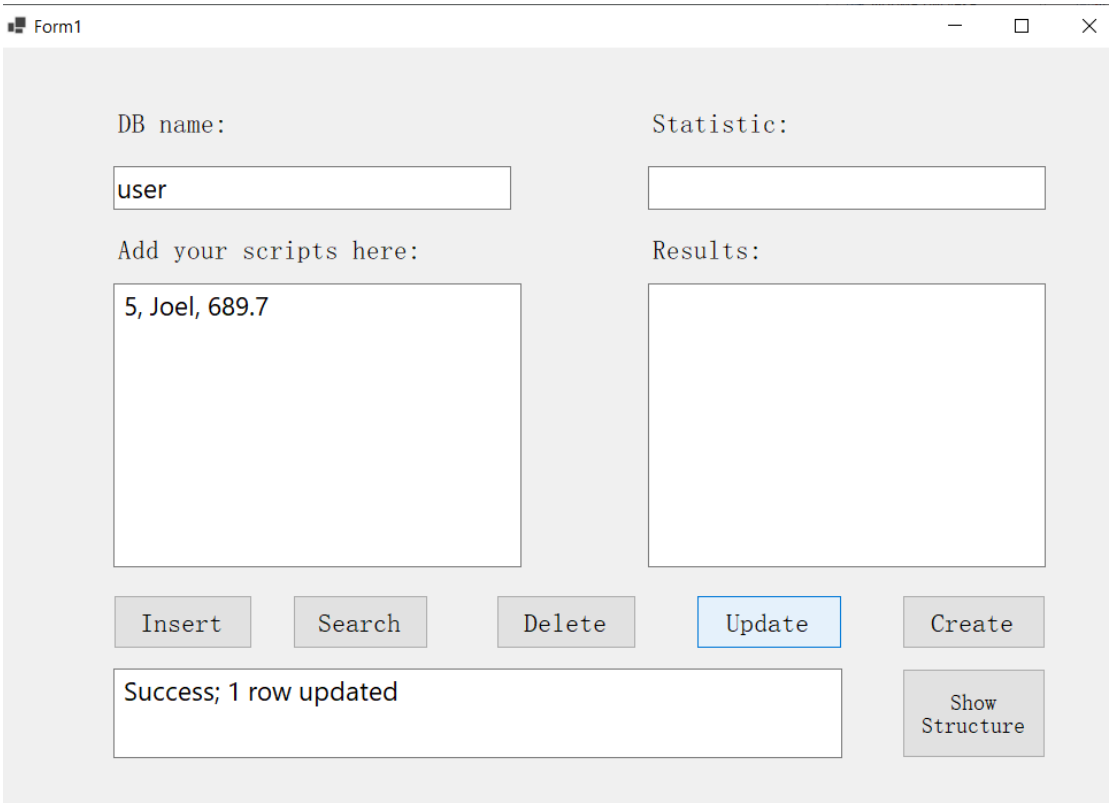

Рисунок 3.4 – Зміна запису

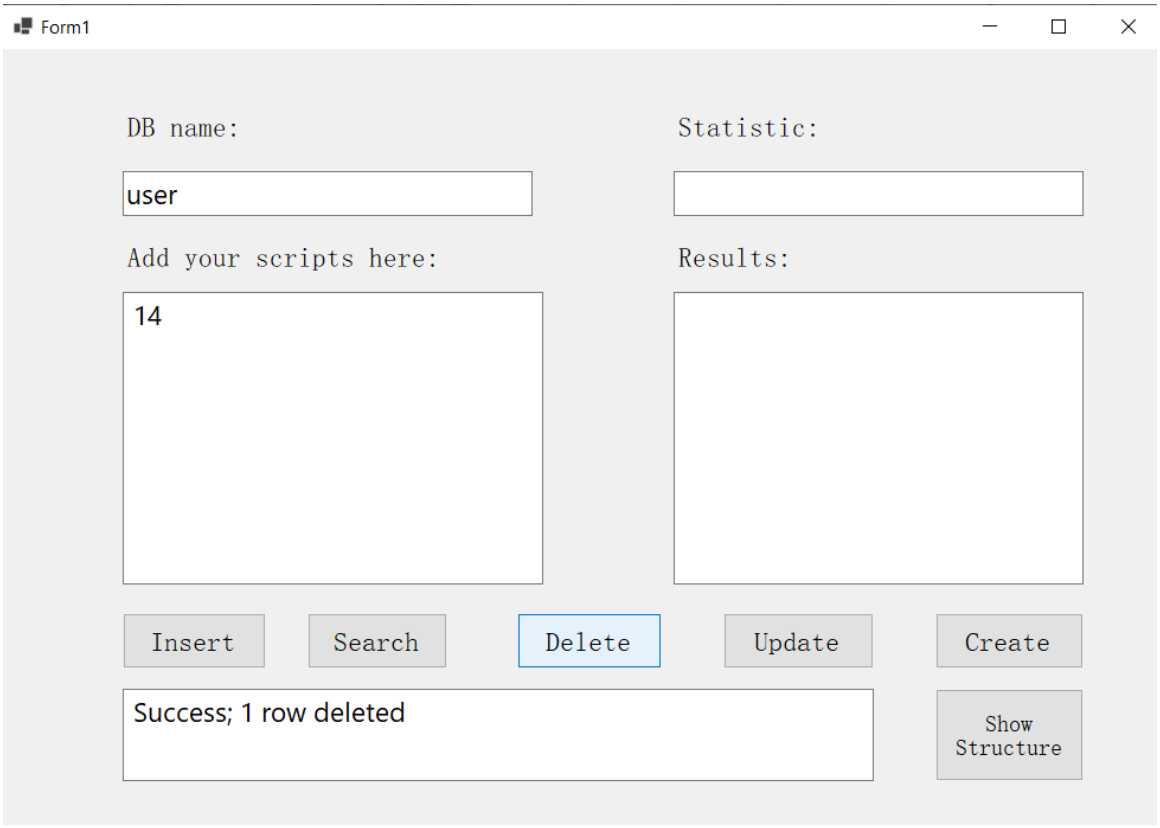

Рисунок 3.5 – Видалення запису

Рисунок 3.6 – Графічне представлення ключів

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	Ключ	Число порівнянь
1	993	14
2	8	13
3	9999	13
4	158	12
5	7921	12
6	63	11
7	3901	14
8	500	12
9	493	13
10	3	11
11	8723	11
12	59	12
13	1000	11
14	2845	13
15	720	13
16(значення, якого немає)	10001	14

ВИСНОВОК

В рамках лабораторної роботи я виконала реалізацію невеликої СУБД на основі червоно-чорного дерева. Були описані функції додавання, пошуку, видалення та зміни даних за ключем за допомогою псевдокоду, а також був написаний програмний код для зазначених функцій. Після цього я протестувала кожну з функцій та переконалась, що вони працюють коректно і після виконання кожної з них червоно-чорне дерево зберігає свої властивості. Далі я створила базу даних з 10000 записами та зафіксувала кількість порівнянь під час пошуку для 15 випадків. У результаті було визначено середнє значення порівнянь, зроблених під час пошуку, яке становить 12.333.

КРИТЕРІЇ ОЦІНЮВАННЯ

За умови здачі лабораторної роботи до 13.11.2022 включно максимальний бал дорівнює – 5. Після13.11.2022 максимальний бал дорівнює – 1.

Критерії оцінювання увідсотках 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.