

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-14 Бабіч Денис

(шифр, прізвище, ім'я, по батькові)

Перевірив

Ахаладзе Ілля Елдарійович

(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

1	Мета лабораторної роботи	3
2	Завдання	4
3	Виконання	5
3.1	Псевдокод алгоритмів	5
3.2	Часова складність пошуку	7
3.3	Програмна реалізація	8
3.3.1	Вихідний код	8
3.3.2	Приклад роботи	17
3.4	Тестування алгоритму	17
3.4.1	Часові характеристики оцінювання	18
	Висновок	19
	Критерії оцінювання	20

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
2	Файли з щільним індексом з областю переповнення, бінарний пошук

3.1 Псевдокод алгоритмів

```

const BLOCK_CAPACITY = 4
const NUMBER_OF_BLOCKS = 4

//Отримання запису з таблиці за ключем
function Get(key, out value) returns boolean
    if (keys.Contains(key)) then
        blockIndex = GetBlockIndex(key)
        reference = blockIndex != -1 ?
            SearchingAlgorithms.GetRecordReference(index[blockIndex], key) :
            SearchingAlgorithms.GetRecordReference(overflow, key)
        if (reference != -1) then
            value = FileWork.GetValue(mainPath, reference)
            return true
        endif
    endif
    value = null
    return false

//Додавання запису до таблиці
function Add(key, value) returns boolean
    if (keys.Add(key)) then
        newRecord = IndexRecord(key, lastReference)
        newMainRecord = MainRecord(0, key, value)
        AppendToFile(mainPath, newMainRecord.ToString())
        count += 1
        lastReference += 1
        blockIndex = GetBlockIndex(key)
        if (blockIndex != -1) then
            index[blockIndex][GetIndexInBlock(key)] = newRecord
            WriteRecords(indexPath, IndexToRecords(index))
            return true
        else
            overflow.Add(newRecord)
            overflow = overflow.OrderBy(record => record.Key).ToList()
            WriteRecords(overflowPath, overflow)
            return true
        endif
    endif
    return false

```

//Редагування запису за ключем

fucntion Edit(key, newValue) **returns boolean**

if (keys.Contains(key)) **then**

 blockIndex = GetBlockIndex(key)

 reference=blockIndex!= -1?

 SearchingAlgorithms.GetRecordReference(index[blockIndex], key) :

 SearchingAlgorithms.GetRecordReference(overflow, key)

if (reference != -1) **then**

 ChangeValue(mainPath, reference, newValue)

return true

endif

endif

return false

//Видалення запису з таблиці за ключем

function Remove(key) **returns boolean**

if (keys.Contains(key)) **then**

 blockIndex = GetBlockIndex(key)

if (blockIndex != -1) **then**

 indexInBlock = GetRecordPosition(index[blockIndex], key)

if (indexInBlock != -1) **then**

 reference = GetRecordReference(index[blockIndex], key)

 index[blockIndex][indexInBlock] = IndexRecord.EmptyRecord

 WriteRecords(indexPath, IndexToRecords(index))

 MarkAsRemoved(mainPath, reference)

endif

else

 index =GetRecordPosition(overflow, key)

if (index != -1) **then**

 reference =GetRecordReference(overflow, key)

 overflow.RemoveAt(index)

 WriteRecords(overflowPath, overflow)

 MarkAsRemoved(mainPath, reference)

endif

endif

 keys.Remove(key)

return true

endif

return false

```
//Отримання індексу блоку за ключем
function GetBlockIndex(key) returns int
    index = (key - 1) / BLOCK_CAPACITY
    if (index < NUMBER_OF_BLOCKS) then
        return index
    return -1

//Отримання індексу запису у блоку
function GetIndexInBlock(key)
    return (key - 1) % BLOCK_CAPACITY
```

3.2 Часова складність пошуку

Бінарний пошук, $O(\log n)$

3.3 Програмна реалізація

3.3.1 Вихідний код

Validation.cs

```
using System;
using System.Windows.Forms;

namespace DataTable
{
    internal static class Validation
    {
        public static bool ValidateKeyInput(RichTextBox richTextBox, out int result)
        {
            result = -1;

            if (richTextBox.Text.Length == 0)
            {
                MessageBox.Show($"Enter the Key.", "Alert", MessageBoxButtons.OK, MessageBoxIcon.Warning);
                return false;
            }

            if (richTextBox.Text.Length > IndexRecord.MAX_KEY_LENGTH)
            {
                MessageBox.Show($"The Key has to be less or equal to {IndexRecord.MAX_KEY_LENGTH} characters.", "Alert", MessageBoxButtons.OK, MessageBoxIcon.Warning);
                return false;
            }

            if (!Int32.TryParse(richTextBox.Text, out result))
            {
                MessageBox.Show($"The field must contain only digits.", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                richTextBox.Text = String.Empty;
            }

            if (result == 0)
            {
                MessageBox.Show($"The key has to be greater than 0.", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                return false;
            }

            return true;
        }

        public static bool ValidateValueInput(RichTextBox richTextBox, int length) => richTextBox.Text.Length <= length;
    }
}
```

DataTable.cs

```
private readonly string mainPath;
private readonly string indexPath;
private readonly string overflowPath;

private const int BLOCK_CAPACITY = 4;
private const int NUMBER_OF_BLOCKS = 4;

private HashSet<int> keys;
private List<IndexRecord>[] index;
private List<IndexRecord> overflow;

private int lastReference;

private int count;
public int Count { get => count; }

public DataTable(string mainPath, string indexPath, string overflowPath)
{
    this.mainPath = mainPath;
    this.indexPath = indexPath;
    this.overflowPath = overflowPath;

    keys = new HashSet<int>();
    lastReference = FileWork.GetNumberOfRecords(mainPath);

    this.InitializeIndex();
    this.InitializeOverflow();
}
```

```

public bool Get(int key, out string value)
{
    if (keys.Contains(key))
    {
        int blockIndex = GetBlockIndex(key);
        int reference = blockIndex != -1 ? SearchingAlgorithms.GetRecordReference(index[blockIndex], key) : SearchingAlgorithms.GetRecordReference(overflow, key);

        if (reference != -1)
        {
            value = FileWork.GetValue(mainPath, reference);
            return true;
        }
    }

    value = null;
    return false;
}

public bool Edit(int key, string newValue)
{
    if (keys.Contains(key))
    {
        int blockIndex = GetBlockIndex(key);
        int reference = blockIndex != -1 ? SearchingAlgorithms.GetRecordReference(index[blockIndex], key) : SearchingAlgorithms.GetRecordReference(overflow, key);

        if (reference != -1)
        {
            FileWork.ChangeValue(mainPath, key, reference, newValue);
            return true;
        }
    }

    return false;
}

public bool Add(int key, string value)
{
    if (keys.Add(key))
    {
        IndexRecord newRecord = new IndexRecord(key, lastReference);
        MainRecord newMainRecord = new MainRecord(0, key, value);
        FileWork.AppendToFile(mainPath, newMainRecord.ToString());
        ++count;
        ++lastReference;

        int blockIndex = GetBlockIndex(key);

        if (blockIndex != -1)
        {
            index[blockIndex][GetIndexInBlock(key)] = newRecord;
            FileWork.WriteRecords(indexPath, IndexToRecords(index));
            return true;
        }
        else
        {
            overflow.Add(newRecord);
            overflow = overflow.OrderBy(record => record.Key).ToList();
            FileWork.WriteRecords(overflowPath, overflow);
            return true;
        }
    }
}

return false;
}

public void Clear()
{
    File.WriteAllText(mainPath, String.Empty);
    File.WriteAllText(indexPath, String.Empty);
    File.WriteAllText(overflowPath, String.Empty);

    lastReference = 0;
    count = 0;
    keys.Clear();
    overflow.Clear();

    foreach(List<IndexRecord> block in index)
    {
        for (int i = 0; i < BLOCK_CAPACITY; ++i)
            block[i] = IndexRecord.EmptyRecord;
    }
}

```

```

public bool Remove(int key)
{
    if (keys.Contains(key))
    {
        int blockIndex = GetBlockIndex(key);
        if (blockIndex != -1)
        {
            int indexInBlock = SearchingAlgorithms.GetRecordPosition(index[blockIndex], key);
            if (indexInBlock != -1)
            {
                int reference = SearchingAlgorithms.GetRecordReference(index[blockIndex], key);
                index[blockIndex][indexInBlock] = IndexRecord.EmptyRecord;
                FileWork.WriteRecords(indexPath, IndexToRecords(index));
                FileWork.MarkRecordAsRemoved(mainPath, reference);
            }
        }
        else
        {
            int index = SearchingAlgorithms.GetRecordPosition(overflow, key);
            if (index != -1)
            {
                int reference = SearchingAlgorithms.GetRecordReference(overflow, key);
                overflow.RemoveAt(index);
                FileWork.WriteRecords(overflowPath, overflow);
                FileWork.MarkRecordAsRemoved(mainPath, reference);
            }
        }
        keys.Remove(key);
        return true;
    }

    return false;
}

```

```

public List<MainRecord> GetMainRecords(string path)
{
    List<MainRecord> records = new List<MainRecord>();

    using (StreamReader sr = new StreamReader(File.OpenRead(path)))
    {
        while (!sr.EndOfStream)
        {
            MainRecord currentRecord = MainRecord.ConvertLineToRecord(sr.ReadLine());

            if (currentRecord.IsDeleted == 0)
                records.Add(currentRecord);
        }
    }

    return records;
}

4 references
public List<IndexRecord> GetIndexRecords(string path)
{
    List<IndexRecord> records = new List<IndexRecord>();

    using (StreamReader sr = new StreamReader(File.OpenRead(path)))
    {
        while (!sr.EndOfStream)
            records.Add(IndexRecord.ConvertLineToRecord(sr.ReadLine()));
    }

    return records;
}

```

```
private int GetBlockIndex(int key)
{
    int index = (key - 1) / BLOCK_CAPACITY;

    if (index < NUMBER_OF_BLOCKS)
        return index;

    return -1;
}

2 references
private int GetIndexInBlock(int key) => (key - 1) % BLOCK_CAPACITY;

2 references
private List<IndexRecord> IndexToRecords(List<IndexRecord>[] index)
{
    List<IndexRecord> records = new List<IndexRecord>();

    foreach (List<IndexRecord> block in index)
    {
        foreach (IndexRecord record in block)
        {
            if (record != null)
                records.Add(record);
        }
    }

    return records;
}
```

```
private void InitializeIndex()
{
    List<IndexRecord> records = this.GetIndexRecords(indexPath);

    index = new List<IndexRecord>[NUMBER_OF_BLOCKS];

    for (int i = 0; i < NUMBER_OF_BLOCKS; ++i)
    {
        index[i] = new List<IndexRecord>(BLOCK_CAPACITY);

        for (int j = 0; j < BLOCK_CAPACITY; ++j)
            index[i].Add(IndexRecord.EmptyRecord);
    }

    int blockIndex;

    foreach (IndexRecord record in records)
    {
        if (record.Key == 0)
            continue;

        blockIndex = GetBlockIndex(record.Key);

        if (blockIndex != -1)
            index[blockIndex][GetIndexInBlock(record.Key)] = record;
    }

    foreach (IndexRecord record in records)
        keys.Add(record.Key);
}

1 reference
private void InitializeOverflow()
{
    overflow = this.GetIndexRecords(overflowPath);

    foreach (IndexRecord record in overflow)
        keys.Add(record.Key);
}
```

FileWork.cs

```
public static string GetValue(string path, int reference)
{
    using (StreamReader sr = new StreamReader(File.OpenRead(path)))
    {
        sr.BaseStream.Position = GetPositionInFile(reference);
        return sr.ReadLine().Split(':')[2].Trim();
    }
}

1 reference
public static void ChangeValue(string path, int key, int reference, string newValue)
{
    using (StreamWriter sw = new StreamWriter(File.OpenWrite(path)))
    {
        sw.BaseStream.Position = GetPositionInFile(reference);
        MainRecord newRecord = new MainRecord(0, key, newValue);
        sw.WriteLine(newRecord.ToString());
    }
}

public static void MarkRecordAsRemoved(string path, int reference)
{
    string line;
    long positionInFile = GetPositionInFile(reference);

    using (StreamReader sr = new StreamReader(File.OpenRead(path)))
    {
        sr.BaseStream.Position = positionInFile;
        line = sr.ReadLine();
    }

    using (StreamWriter sw = new StreamWriter(File.OpenWrite(path)))
    {
        sw.BaseStream.Position = positionInFile;
        sw.WriteLine("1" + line.Substring(1));
    }
}

3 references
private static long GetPositionInFile(int reference) => MainRecord.LENGTH * reference;

1 reference
public static void AppendToFile(string path, string line)
{
    using (StreamWriter sw = File.AppendText(path))
        sw.WriteLine(line);
}

public static void WriteRecords(string path, List<IndexRecord> records)
{
    File.WriteAllText(path, String.Empty);

    using (StreamWriter sw = new StreamWriter(File.OpenWrite(path)))
    {
        foreach (IndexRecord record in records)
            sw.WriteLine(record);
    }
}

1 reference
public static int GetNumberOfRecords(string path)
{
    using (StreamReader sr = new StreamReader(File.OpenRead(path)))
        return sr.ReadToEnd().Split('\n').Length - 1;
}
```

Record.cs

```
public sealed class IndexRecord
{
    public const int MAX_KEY_LENGTH = 6;

    11 references
    public int Key { get; }

    3 references
    public int Reference { get; }

    3 references
    public static IndexRecord EmptyRecord { get => new IndexRecord(0, 0); }

    3 references
    public IndexRecord(int key, int reference)
    {
        this.Key = key;
        this.Reference = reference;
    }

    1 reference
    public static IndexRecord ConvertLineToRecord(string line)
    {
        string[] values = line.Split(':');
        return new IndexRecord(int.Parse(values[0]), int.Parse(values[1]));
    }

    0 references
    public override string ToString() => $"{Key}:{Reference}";
}
```

```
public sealed class MainRecord
{
    public const int MAX_KEY_LENGTH = 6;
    public const int MAX_VALUE_LENGTH = 50;
    public const int LENGTH = 5 + MAX_KEY_LENGTH + MAX_VALUE_LENGTH;

    3 references
    public byte IsDeleted { get; set; }

    2 references
    public int Key { get; }

    2 references
    public string Value { get; set; }

    3 references
    public MainRecord(byte isDeleted, int key, string value)
    {
        this.IsDeleted = isDeleted;
        this.Key = key;
        this.Value = value;
    }

    1 reference
    public static MainRecord ConvertLineToRecord(string line)
    {
        string[] values = line.Split(':');
        return new MainRecord(byte.Parse(values[0]), int.Parse(values[1].Trim()), values[2].Trim());
    }

    - references
    public override string ToString() => $"{IsDeleted}:{Key.ToString().PadLeft(MAX_KEY_LENGTH)}:{Value.PadLeft(MAX_VALUE_LENGTH)}";
}
```

SearchingAlgorithms.cs

```
public static int GetRecordReference(List<IndexRecord> records, int key)
{
    List<IndexRecord> updatedBlock = RemoveEmptyRecords(records);

    int min = 0;
    int max = updatedBlock.Count - 1;

    int mid;
    int guess;

    while (min <= max)
    {
        mid = (min + max) / 2;
        guess = updatedBlock[mid].Key;

        if (guess == key)
            return updatedBlock[mid].Reference;
        else if (guess < key)
            min = mid + 1;
        else
            max = mid - 1;
    }

    return -1;
}

public static int GetRecordPosition(List<IndexRecord> records, int key)
{
    List<IndexRecord> updatedBlock = RemoveEmptyRecords(records);

    int min = 0;
    int max = updatedBlock.Count - 1;

    int mid;
    int guess;

    while (min <= max)
    {
        mid = (min + max) / 2;
        guess = updatedBlock[mid].Key;

        if (guess == key)
            return mid;
        else if (guess < key)
            min = mid + 1;
        else
            max = mid - 1;
    }

    return -1;
}

2 references
private static List<IndexRecord> RemoveEmptyRecords(List<IndexRecord> block)
{
    List<IndexRecord> result = new List<IndexRecord>();

    foreach (IndexRecord record in block)
    {
        if (record.Key != 0)
            result.Add(record);
    }

    return result;
}
```

Form2.cs

```
using System;
using System.Windows.Forms;

namespace DataTable
{
    6 references
    public partial class Form2 : Form
    {
        private Action operation;

        2 references
        public Form2(Action choice)
        {
            operation = choice;
            InitializeComponent();
        }

        1 reference
        private void button1_Click(object sender, EventArgs e)
        {
            if (Validation.ValidateKeyInput(KeyInput, out int key))
            {
                switch (operation)
                {
                    case Action.Getting:
                        if ( MainForm.ExternalInterface.DataTable.Get(key, out string output))
                            MessageBox.Show($"{output}");
                        else
                            MessageBox.Show($"Selected Key does not exist.", "Alert", MessageBoxButtons.OK, MessageBoxIcon.Warning);
                        break;
                    case Action.Removing:
                        if (MainForm.ExternalInterface.DataTable.Remove(key))
                            MainForm.ExternalInterface.UpdateTables();
                        else
                            MessageBox.Show($"Selected Key does not exist.", "Alert", MessageBoxButtons.OK, MessageBoxIcon.Warning);
                        break;
                }
            }
        }
    }
}
```

Form3.cs

```
using System;
using System.Windows.Forms;

namespace DataTable
{
    6 references
    public partial class Form3 : Form
    {
        private Action operation;

        2 references
        public Form3(Action choice)
        {
            operation = choice;
            InitializeComponent();
        }

        1 reference
        private void button1_Click(object sender, EventArgs e)
        {
            if (Validation.ValidateKeyInput(KeyInput, out int key))
            {
                if (Validation.ValidateValueInput(ValueInput, MainRecord.MAX_VALUE_LENGTH))
                {
                    switch (operation)
                    {
                        case Action.Editing:
                            if (MainForm.ExternalInterface.DataTable.Edit(key, ValueInput.Text))
                                MainForm.ExternalInterface.UpdateTables();
                            else
                                MessageBox.Show($"Selected Key does not exist.", "Alert", MessageBoxButtons.OK, MessageBoxIcon.Warning);
                            break;
                        case Action.Adding:
                            if (MainForm.ExternalInterface.DataTable.Add(key, ValueInput.Text))
                                MainForm.ExternalInterface.UpdateTables();
                            else
                                MessageBox.Show($"Selected Key is already exists.", "Alert", MessageBoxButtons.OK, MessageBoxIcon.Warning);
                            break;
                    }
                }
            }
        }
    }
}
```

MainForm.cs

```
using System;
using System.Windows.Forms;

namespace DataTable
{
    12 references
    public enum Action : byte
    {
        Getting = 0,
        Adding,
        Removing,
        Editing,
    }

    11 references
    public partial class MainForm : Form
    {
        private readonly string main = "Files\\main.txt";
        private readonly string index = "Files\\index.txt";
        private readonly string overflow = "Files\\overflow.txt";

        public DataTable DataTable;

        private Form3 adding = new Form3(Action.Adding);
        private Form2 getting = new Form2(Action.Getting);
        private Form3 editing = new Form3(Action.Editing);
        private Form2 removing = new Form2(Action.Removing);

        8 references
        public static MainForm ExternalInterface { get; private set; }

        1 reference
        public MainForm()
        {
            ExternalInterface = this;
            InitializeComponent();
        }
    }
}
```

```
private void Load_Click(object sender, EventArgs e)
{
    DataTable = new DataTable(main, index, overflow);
    UpdateTables();
}

1 reference
private void Add_Click(object sender, EventArgs e) => adding.ShowDialog();

1 reference
private void Get_Click(object sender, EventArgs e) => getting.ShowDialog();

1 reference
private void Edit_Click(object sender, EventArgs e) => editing.ShowDialog();

1 reference
private void Remove_Click(object sender, EventArgs e) => removing.ShowDialog();

1 reference
private void Clear_Click(object sender, EventArgs e)
{
    if (DataTable != null)
    {
        DataTable.Clear();
        UpdateTables();
    }
    else
        MessageBox.Show($"{nameof(DataTable)} {nameof(DataTable)} is empty.", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
}

5 references
public void UpdateTables()
{
    mainRecords.DataSource = DataTable.GetMainRecords(main);
    mainRecords.Columns[0].Visible = false;
    indexRecords.DataSource = DataTable.GetIndexRecords(index);
    overflowRecords.DataSource = DataTable.GetIndexRecords(overflow);
}
```

3.3.2 Приклади роботи

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

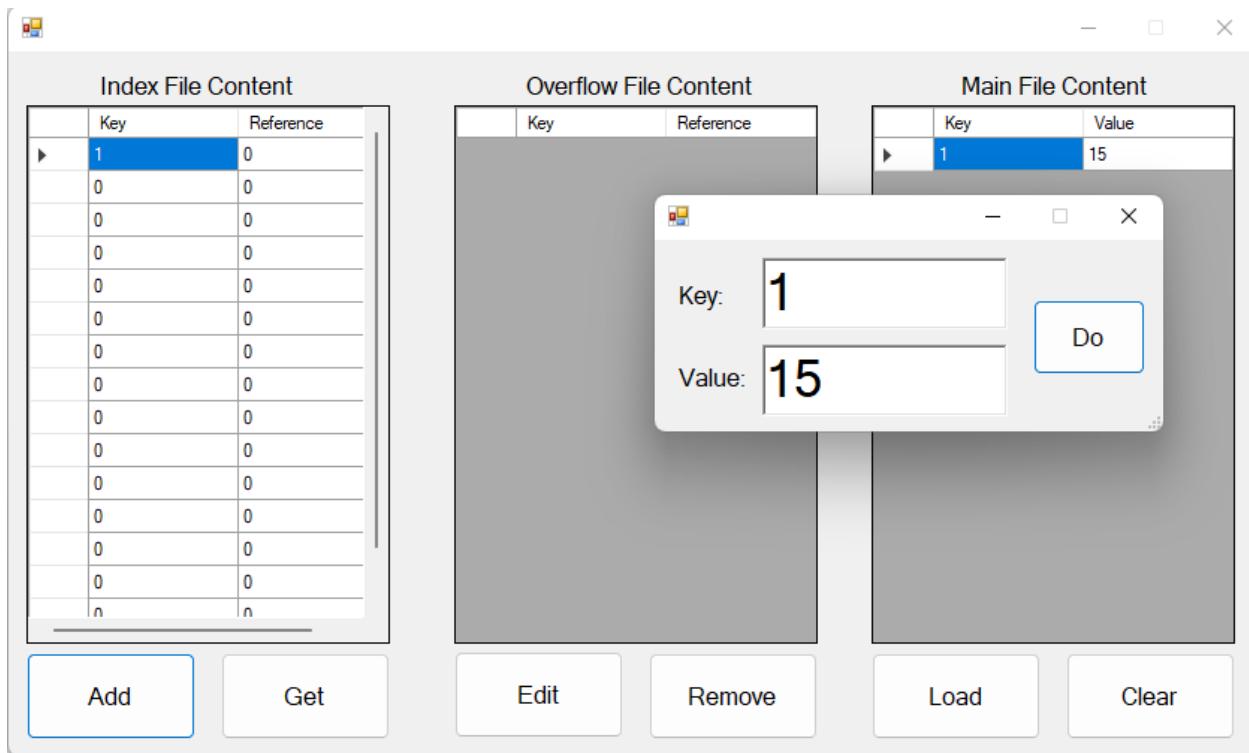


Рисунок 3.1 – Додавання запису

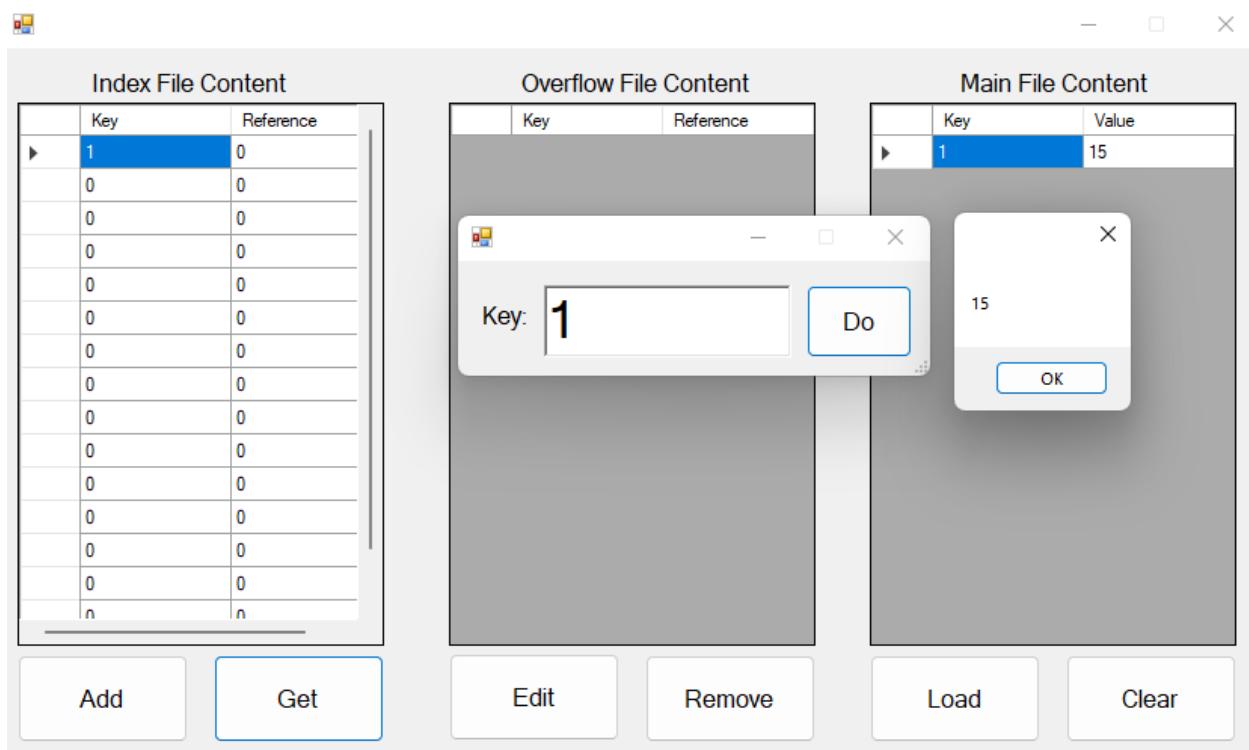


Рисунок 3.2 – Пошук запису

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	2
2	6
3	7
4	15
5	11
6	13
7	4
8	16
9	17
10	12
11	21
12	23
13	20
14	7
15	13

ВИСНОВОК

В рамках лабораторної роботи була розроблена таблична структура даних з щільним індексом та областю переповнення, пошук записів у якій відбувається за допомогою алгоритму бінарного пошуку. Структура даних працює за таким принципом: поки є місце у індексній області і є відповідні ключі, то заповнюється індексна область. У випадку закінчення наявних місць у індексній області, нові записи додаються до області переповнення. Кожному запису у індексній та області переповнення відповідає посилання у основному файлі, у якому зберігаються безпосередньо самі дані. Дано структура даних має такі методи, як додавання до таблиці, отримання даних з таблиці за ключем, видалення даних за ключем, редагування записів і очищення таблиці.

КРИТЕРІЙ 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.