**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)**

(шифр, прізвище, ім'я, по батькові)

*ІП-13 Жмайло Дмитро О.*

**Перевірив**

(прізвище, ім'я, по батькові)

*Сопов О. О.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc114359761)

[2 Завдання 4](#_Toc114359762)

[3 Виконання 7](#_Toc114359763)

[3.1 Псевдокод алгоритмів 7](#_Toc114359764)

[3.2 Часова складність пошуку 7](#_Toc114359765)

[3.3 Програмна реалізація 8](#_Toc114359766)

[3.3.1 Вихідний код 8](#_Toc114359767)

[3.3.2 Приклади роботи 35](#_Toc114359768)

[3.4 Тестування алгоритму 37](#_Toc114359769)

[3.4.1 Часові характеристики оцінювання 37](#_Toc114359770)

[Висновок 37](#_Toc114359771)

[Критерії оцінювання 39](#_Toc114359772)

# Мета лабораторної роботи

Мета роботи – вивчити основні підходи проектування та обробки складних структур даних.

# 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 |

# Виконання

## Псевдокод алгоритмів (бінарного пошуку в дереві)

## 

## Часова складність пошуку

Пошук у B-tree складається з **2** основних процедур:

* пошук вузла, який має складність **𝑂(log 𝑛)**
* пошук елемента всередині вузла бінарним алгоритмом, який має складність **𝑂(log 𝑡)**

*де n – це кількість вузлів у дереві, та t – кількість елементів у вузлі*

## Загальна складність двох функцій дорівнює **𝑂(𝑡 ∗ log 𝑛).**

## Програмна реалізація

### Вихідний код

Program.cs

namespace lab3PA

{

internal static class Program

{

/// <summary>

/// The main entry point for the application.

/// </summary>

[STAThread]

static void Main()

{

ApplicationConfiguration.Initialize();

Application.Run(new Form1());

}

}

}

Form1.cs

namespace lab3PA

{

public partial class Form1 : Form

{

static string path = "db.txt";

const int DEGREE = 50;

BTree? tree = FileHelper.ReadFromFile(path, DEGREE);

public Form1()

{

InitializeComponent();

}

private void Form1\_Load(object sender, EventArgs e)

{

}

private void button1\_Click(object sender, EventArgs e)

{

string message = "";

if (radioButton1.Checked)

{

if (string.IsNullOrEmpty(textBox1.Text))

{

message = "Помилка! Поля введення значення ключа повинно бути заповненим";

}

else

{

if(int.TryParse(textBox1.Text, out int key))

{

int compCount = 0;

NodeData? result = tree.Search(key, ref compCount);

if (result != null)

{

message = $"Ключ {key} знайдено за {compCount} порівнянь! Йому відповідає значення: '{result.Value}'";

}

else

{

message = $"Ключ {key} не знайдено!";

}

}

else

{

message = $"Помилка! {key} - некоректний ключ. Пошук неможливий";

}

}

}

else if(radioButton2.Checked)

{

if (string.IsNullOrEmpty(textBox1.Text) || string.IsNullOrEmpty(textBox2.Text))

{

message = "Помилка! Поля введення повинні бути заповненими. Додавання запису неможливе";

}

else

{

if (int.TryParse(textBox1.Text, out int key))

{

if (!FileHelper.IsThereRepeatingKeys(tree, key))

{

tree.Insert(key, textBox2.Text);

message = "Запис успішно додано до БД!";

}

else

{

message = $"Помилка! {key} ключ уже є в БД. Додавання запису неможливе";

}

}

else

{

message = $"Помилка! {key} - некоректний ключ. Додавання запису неможливе";

}

}

}

else if (radioButton3.Checked)

{

if (string.IsNullOrEmpty(textBox1.Text))

{

message = "Помилка! Поля введення значення ключа повинно бути заповненим. Видалення неможливе";

}

else

{

if (int.TryParse(textBox1.Text, out int key))

{

bool success = true;

tree.Delete(key, ref success);

if (success)

{

message = $"Ключ {key} видалено успішно";

}

else

{

message = $"Ключ {key} не знайдено! Видалення неможливе";

}

}

else

{

message = $"Помилка! {key} - некоректний ключ. Видалення неможливе";

}

}

}

else

{

if (string.IsNullOrEmpty(textBox1.Text) || string.IsNullOrEmpty(textBox2.Text))

{

message = "Помилка! Поля введення повинні бути заповненими. Редагування неможливе";

}

else

{

if (int.TryParse(textBox1.Text, out int key))

{

bool success = false;

tree.Edit(key, textBox2.Text, ref success);

if (success)

{

message = $"Ключ {key} змінено успішно! Нове значення - '{textBox2.Text}'";

}

else

{

message = $"Ключ {key} не знайдено! Редагування неможливе";

}

}

}

}

textBox3.AppendText(message + Environment.NewLine);

}

private void button3\_Click(object sender, EventArgs e)

{

FileHelper.SaveFile(path, tree, out string message);

textBox3.AppendText(message + Environment.NewLine);

}

private void button2\_Click(object sender, EventArgs e)

{

File.Delete(path);

string message = "Файл успішно очищено!";

textBox3.AppendText(message + Environment.NewLine);

}

private void button4\_Click(object sender, EventArgs e)

{

tree.SaveData(tree.root);

textBox3.AppendText(Environment.NewLine + "-=Дерево має вигляд=-" + Environment.NewLine);

for (int i = 0; i < tree.fullTreeList.Count; i++)

{

textBox3.AppendText(tree.fullTreeList[i].ToString() + Environment.NewLine);

}

}

}

}

BTree.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace lab3PA

{

internal class BTree

{

public BTNode? root { get; set; }

private int T;

public List<NodeData> fullTreeList { get; set; }

public BTree(int degree)

{

root = null;

T = degree;

fullTreeList = new List<NodeData>();

}

public NodeData? Search(int key, ref int compCount)

{

return root?.Search(key, ref compCount);

}

public BTNode? PrimitiveSearch(int key)

{

return root?.PrimitiveSearch(key);

}

public void Insert(int k, string Val)

{

if (root == null)

{

root = new BTNode(T);

root.records[0] = new NodeData(k, Val);

root.RecordsAmount++;

return;

}

if (root.IsFull)

{

BTNode sub = new BTNode(T, false);

sub.Children[0] = root;

sub.SplitChildNode(0, root);

int i = 0;

if (sub.records[0].Key < k)

{

i++;

}

sub.Children[i].Insert(k, Val);

root = sub;

}

else

{

root.Insert(k, Val);

}

}

public void Delete(int key, ref bool success)

{

if (root.Equals(null))

{

success = false;

return;

}

root.Delete(key);

if (root.RecordsAmount == 0)

{

if (root.IsLeaf)

{

root = null;

}

else

{

root = root.Children[0];

}

}

}

public void Edit(int key, string newVal, ref bool success)

{

BTNode? find = PrimitiveSearch(key);

if (find == null)

{

success = false;

return;

}

int idx = 0;

while (key > find.records[idx].Key)

{

idx++;

}

find.records[idx].Value = newVal;

success = true;

}

public void SaveData(BTNode node)

{

fullTreeList.Clear();

if (node is not null)

{

for (int i = 0; i < node.records.Length; i++)

{

if (!node.IsLeaf)

SaveData(node.Children[i]);

if (node.records is not null && !node.records[i].IsNull)

fullTreeList.Add(node.records[i]);

}

if (!node.IsLeaf)

SaveData(node.Children[^1]);

}

}

}

}

BTNode.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace lab3PA

{

internal class BTNode

{

public NodeData[] records { get; set; } = Array.Empty<NodeData>();

public int T { get; set; }

public BTNode[] Children { get; set; }

public int RecordsAmount { get; set; }

public bool IsLeaf { get; set; }

internal BTNode(int t, bool isleaf = true)

{

T = t;

records = new NodeData[2 \* T - 1];

for (int i = 0; i < 2 \* T - 1; i++)

{

records[i] = new NodeData();

}

Children = new BTNode[2 \* T];

RecordsAmount = 0;

IsLeaf = isleaf;

}

public bool IsFull => RecordsAmount == records.Length;

public BTNode? PrimitiveSearch(int key)

{

int index = FindIndex(key);

if (records[index].Key == key)

{

return this;

}

if (IsLeaf)

{

return null;

}

return Children[index].PrimitiveSearch(key);

}

public NodeData? Search(int key, ref int compCount)

{

NodeData? result = BinarySearch(key, ref compCount);

if (result is not null)

{

return result;

}

if (IsLeaf)

{

return null;

}

int i = 0;

while (i < records.Length && key > records[i].Key)

{

i++;

}

return Children[i].Search(key, ref compCount);

}

public NodeData? BinarySearch(int key, ref int comparisonsCount)

{

int start = 0;

int end = records.Length - 1;

int middle;

while (start <= end)

{

middle = (end + start) / 2;

comparisonsCount++;

if (records[middle].Key == key)

{

return records[middle];

}

comparisonsCount++;

if (records[middle].Key > key)

{

end = middle - 1;

}

else if(records[middle].Key < key)

{

start = middle + 1;

}

comparisonsCount++;

}

return null;

}

public void Insert(int key, string value)

{

int i = RecordsAmount - 1;

if (IsLeaf)

{

while (i >= 0 && key < records[i].Key)

{

records[i + 1] = records[i];

i--;

}

records[i + 1] = new NodeData(key, value);

RecordsAmount++;

return;

}

while (i >= 0 && key < records[i].Key)

{

i--;

}

if (Children[i + 1].IsFull)

{

SplitChildNode(i + 1, Children[i + 1]);

if (records[i + 1].Key < key)

{

i++;

}

}

Children[i + 1].Insert(key, value);

}

public void SplitChildNode(int index, BTNode child)

{

BTNode newChild = new BTNode(T, child.IsLeaf);

newChild.RecordsAmount = T - 1;

for (int i = 0; i < T - 1; i++)

{

newChild.records[i] = child.records[i + T];

}

if (!IsLeaf)

{

for (int i = 0; i < T; i++)

{

newChild.Children[i] = child.Children[i + T];

}

}

child.RecordsAmount = T - 1;

for (int i = RecordsAmount; i >= index + 1; i--)

{

Children[i + 1] = Children[i];

}

Children[index + 1] = newChild;

for (int i = RecordsAmount - 1; i >= index; i--)

{

records[i + 1] = records[i];

}

records[index] = child.records[T - 1];

RecordsAmount++;

}

private int FindIndex(int key)

{

int index = 0;

while (index < RecordsAmount && key > records[index].Key)

{

index++;

}

return index;

}

internal void Delete(int key)

{

int index = FindIndex(key);

if (index < RecordsAmount && records[index].Key == key)

{

if (IsLeaf)

{

DeleteLeaf(index);

}

else

{

if (Children[index].RecordsAmount >= T)

{

DeletePrev(index);

}

else if (Children[index + 1].RecordsAmount >= T)

{

DeleteNext(index);

}

else

{

Merge(index);

Children[index].Delete(key);

}

}

}

else

{

if (IsLeaf) return;

bool isInLastSubtree = index == RecordsAmount;

if (Children[index].RecordsAmount < T)

{

TakeOneKey(index);

}

if (isInLastSubtree && index > RecordsAmount)

{

Children[index - 1].Delete(key);

}

else

{

Children[index].Delete(key);

}

}

}

private void DeleteLeaf(int index)

{

for (int i = index + 1; i < RecordsAmount - 1; i++)

{

records[i - 1] = records[i];

}

RecordsAmount--;

}

private void DeletePrev(int index)

{

BTNode ptr = Children[index];

while (!ptr.IsLeaf)

{

ptr = ptr.Children[ptr.RecordsAmount];

}

NodeData prev = ptr.records[ptr.RecordsAmount - 1];

ptr.Delete(prev.Key);

records[index] = prev;

}

private void DeleteNext(int index)

{

BTNode ptr = Children[index + 1];

while (!ptr.IsLeaf)

{

ptr = ptr.Children[0];

}

NodeData next = ptr.records[0];

ptr.Delete(next.Key);

records[index] = next;

}

private void Merge(int index)

{

BTNode left = Children[index];

BTNode right = Children[index + 1];

left.records[RecordsAmount] = records[index];

left.RecordsAmount++;

for (int i = 0; i < right.RecordsAmount; i++)

{

left.records[i + T] = right.records[i];

}

if (!left.IsLeaf)

{

for (int i = 0; i <= right.RecordsAmount; i++)

{

left.Children[i + T] = right.Children[i];

}

}

for (int i = index + 1; i < RecordsAmount - 1; i++)

{

records[i - 1] = records[i];

Children[i] = Children[i + 1];

}

RecordsAmount--;

left.RecordsAmount += right.RecordsAmount;

}

private void TakeOneKey(int index)

{

if (!Children[index - 1].Equals(null) && Children[index - 1].RecordsAmount >= T)

{

TakeFromLeft(index);

}

else if (!Children[index + 1].Equals(null) && Children[index + 1].RecordsAmount >= T)

{

TakeFromRight(index);

}

else

{

if (index != RecordsAmount)

{

Merge(index);

}

else

{

Merge(index - 1);

}

}

}

private void TakeFromLeft(int index)

{

for (int i = Children[index].RecordsAmount - 1; i > 0; i--)

{

Children[index].records[i] = Children[index].records[i - 1];

}

Children[index].records[0] = Children[index - 1].records[RecordsAmount - 1];

if (!Children[index].IsLeaf)

{

for (int i = Children[index].RecordsAmount - 1; i > 0; i--)

{

Children[index].Children[i + 1] = Children[index].Children[i];

}

Children[index].Children[0] = Children[index - 1].Children[RecordsAmount];

}

Children[index].RecordsAmount++;

Children[index - 1].RecordsAmount--;

}

private void TakeFromRight(int index)

{

Children[index].records[RecordsAmount] = Children[index + 1].records[0];

for (int i = 0; i < Children[index + 1].RecordsAmount - 1; i++)

{

Children[index + 1].records[i] = Children[index + 1].records[i + 1];

}

if (!Children[index].IsLeaf)

{

Children[index].Children[RecordsAmount + 1] = Children[index + 1].Children[0];

for (int i = 0; i < Children[index + 1].RecordsAmount; i++)

{

Children[index + 1].Children[i] = Children[index + 1].Children[i + 1];

}

}

Children[index].RecordsAmount++;

Children[index + 1].RecordsAmount--;

}

}

}

FileHelper.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace lab3PA

{

internal static class FileHelper

{

public static BTree? ReadFromFile(string path, int degree)

{

string line = "";

BTree tree = new BTree(degree);

if (File.Exists(path))

{

using (StreamReader reader = new StreamReader(path))

{

while (!string.IsNullOrEmpty(line = reader.ReadLine()))

{

string[] data = line.Split('|');

if (data.Length == 2)

{

if (int.TryParse(data[0], out int key))

{

tree.Insert(key, data[1]);

}

}

else

{

return tree;

}

}

}

return tree;

}

else

return tree;

}

public static void SaveFile(string path, BTree tree, out string message)

{

tree.SaveData(tree.root);

using (StreamWriter writer = new StreamWriter(path, false))

{

for (int i = 0; i < tree.fullTreeList.Count; i++)

{

writer.WriteLine(tree.fullTreeList[i].ToString());

}

}

message = "Файл записано успішно!";

}

public static bool IsThereRepeatingKeys(BTree tree, int key)

{

if (tree is null)

return false;

int count = 0;

NodeData? result = tree.Search(key, ref count);

if (result is null)

{

return false;

}

else

return true;

}

}

}

NodeData.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace lab3PA

{

internal class NodeData

{

public int Key { get; set; }

public string? Value { get; set; }

public bool IsNull => Key == Int32.MaxValue;

public NodeData()

{

Key = Int32.MaxValue;

Value = null;

}

public NodeData(int k, string v)

{

Key = k;

Value = v;

}

public override string ToString()

{

return $"{Key}|{Value}";

}

}

}

### Приклади роботи

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

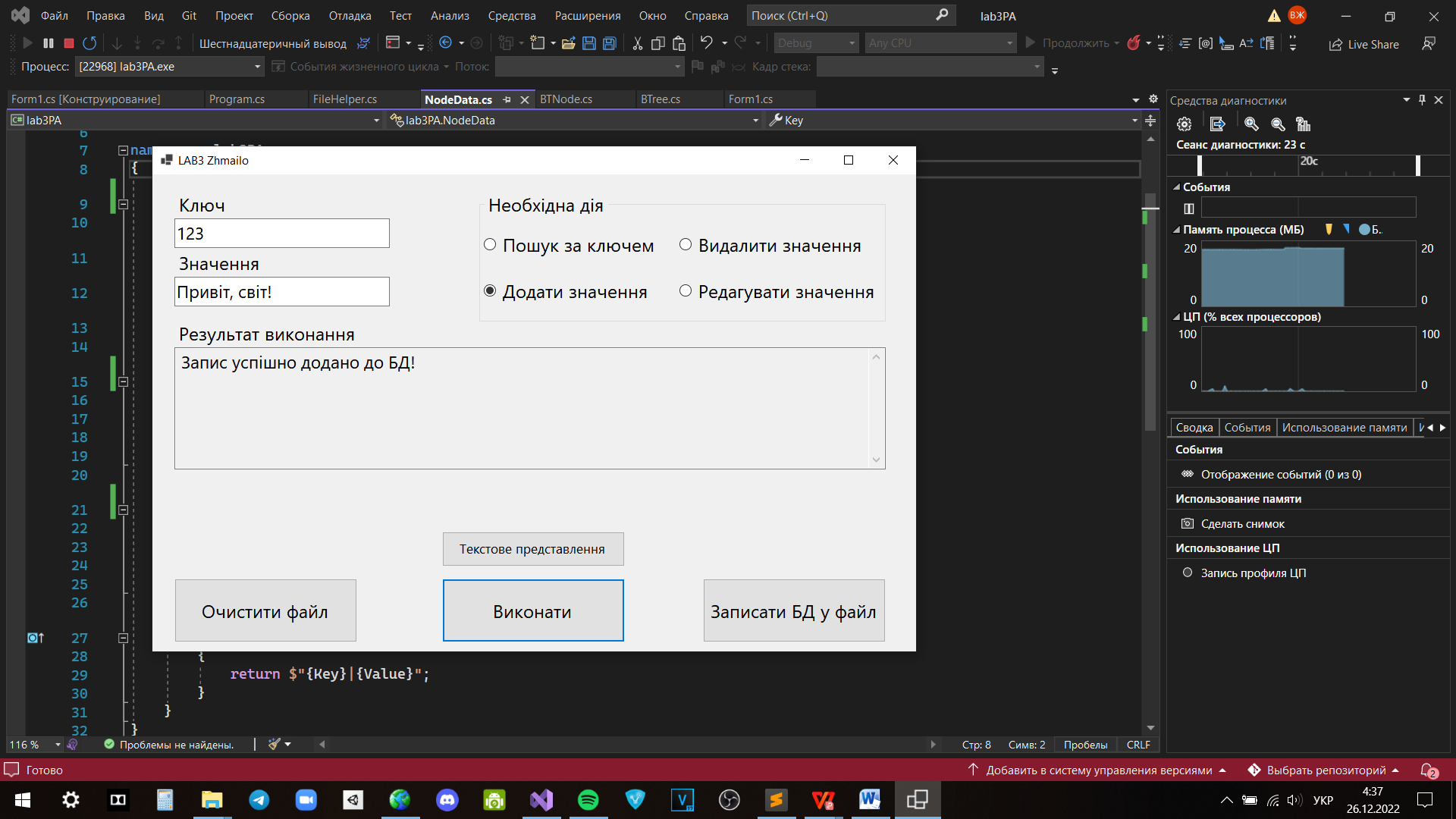


Рисунок 3.1 –Додавання запису

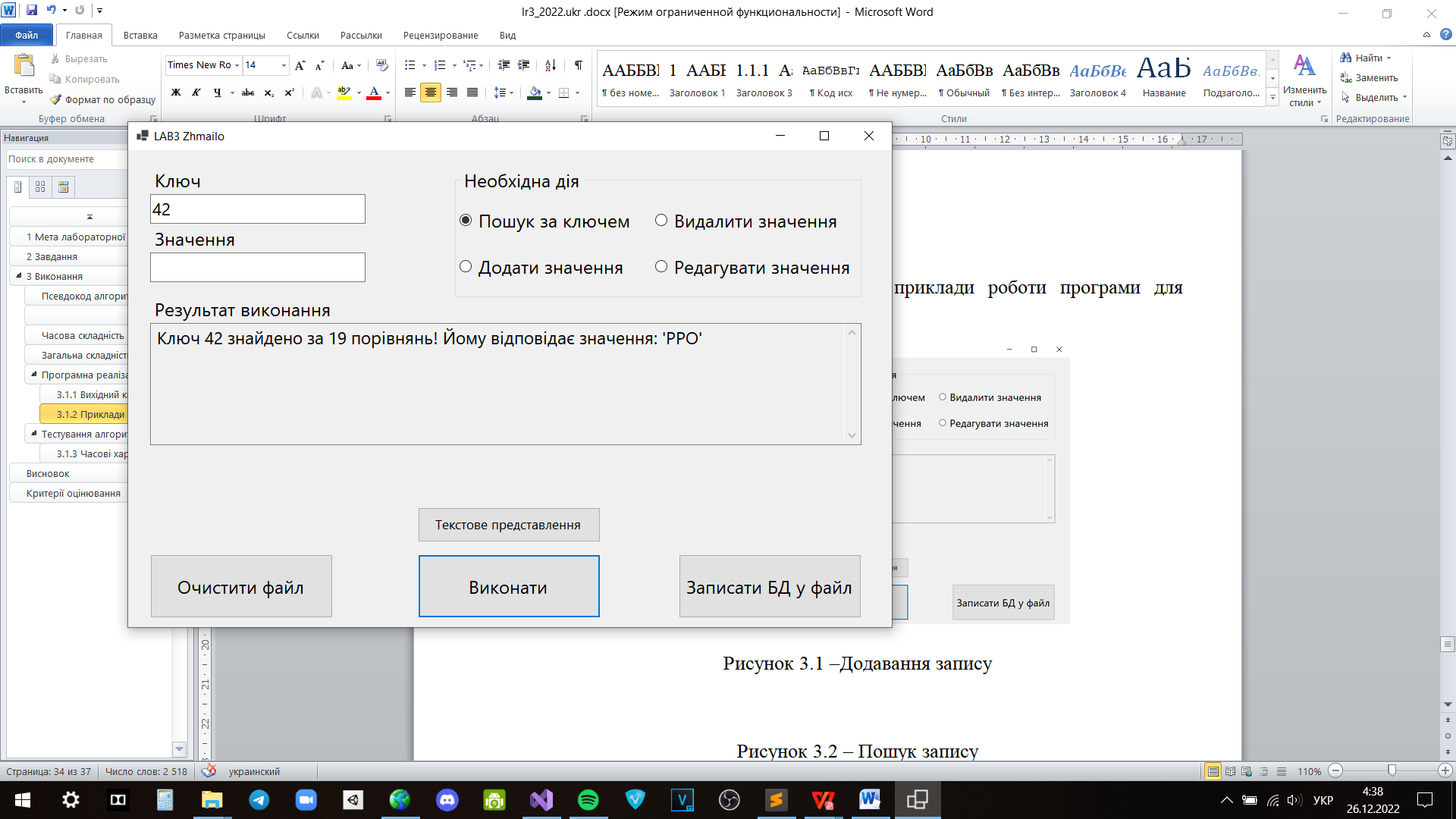
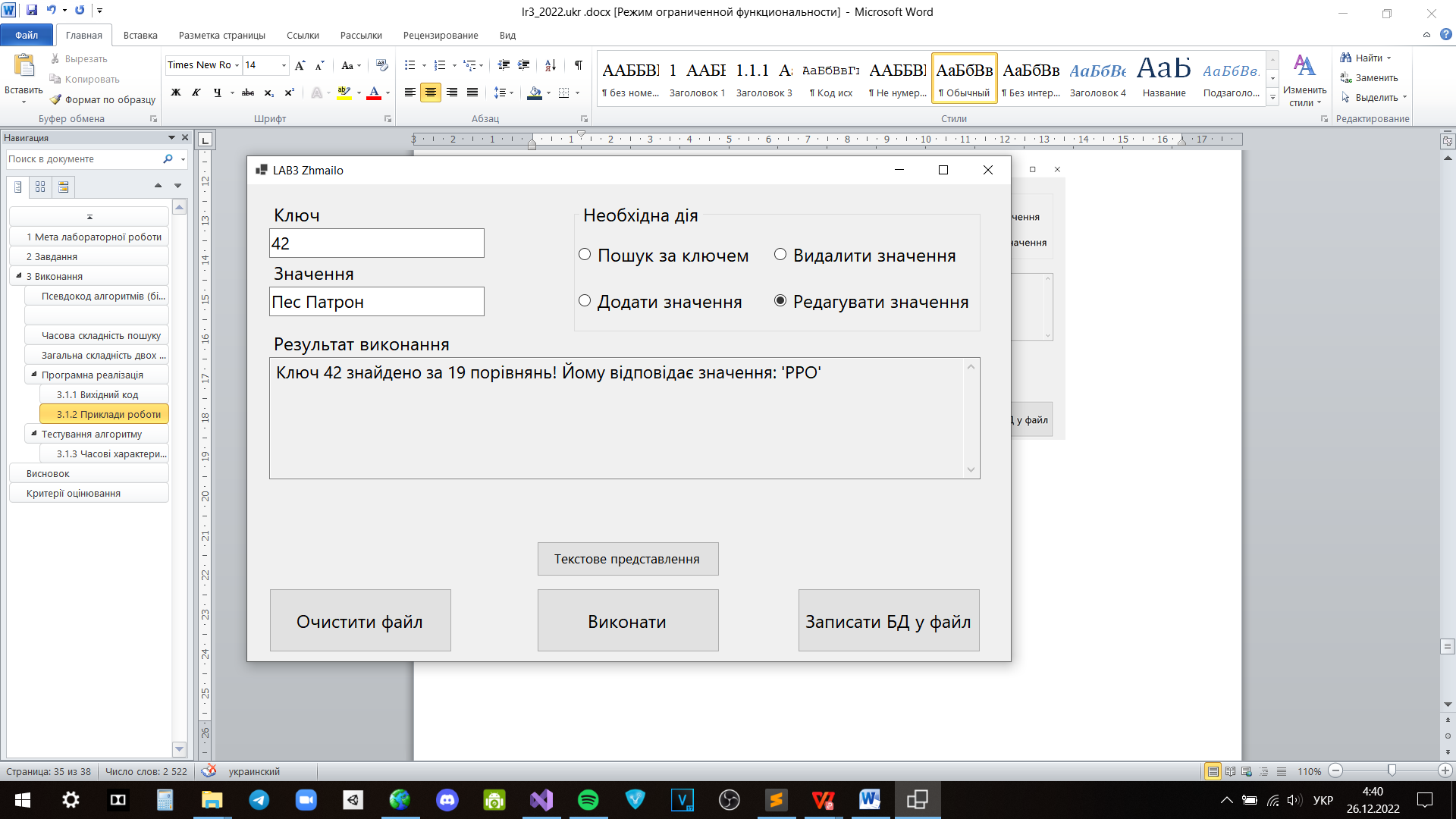


Рисунок 3.2 – Пошук запису



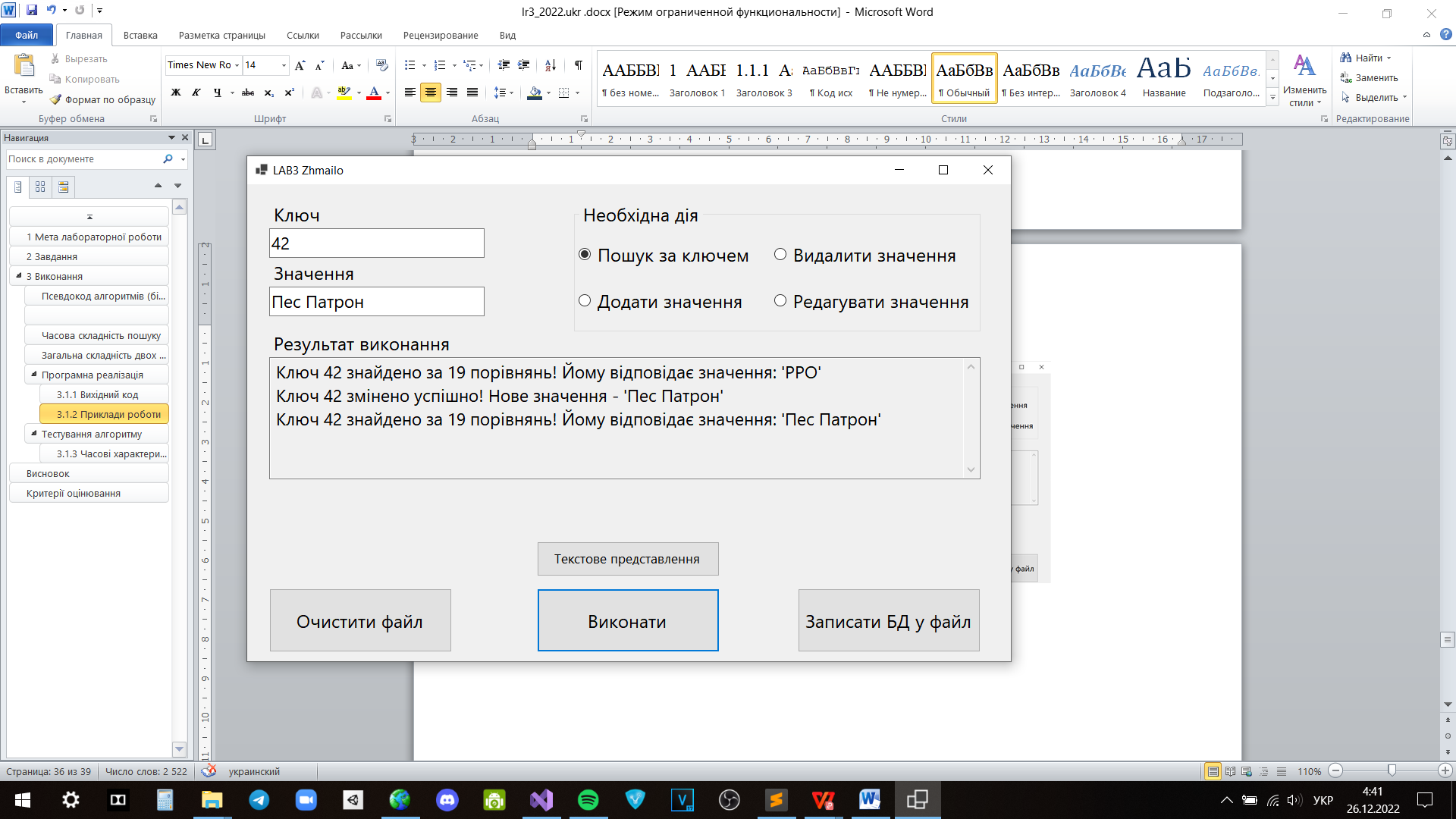


Рисунок 3.3 – Редагування запису

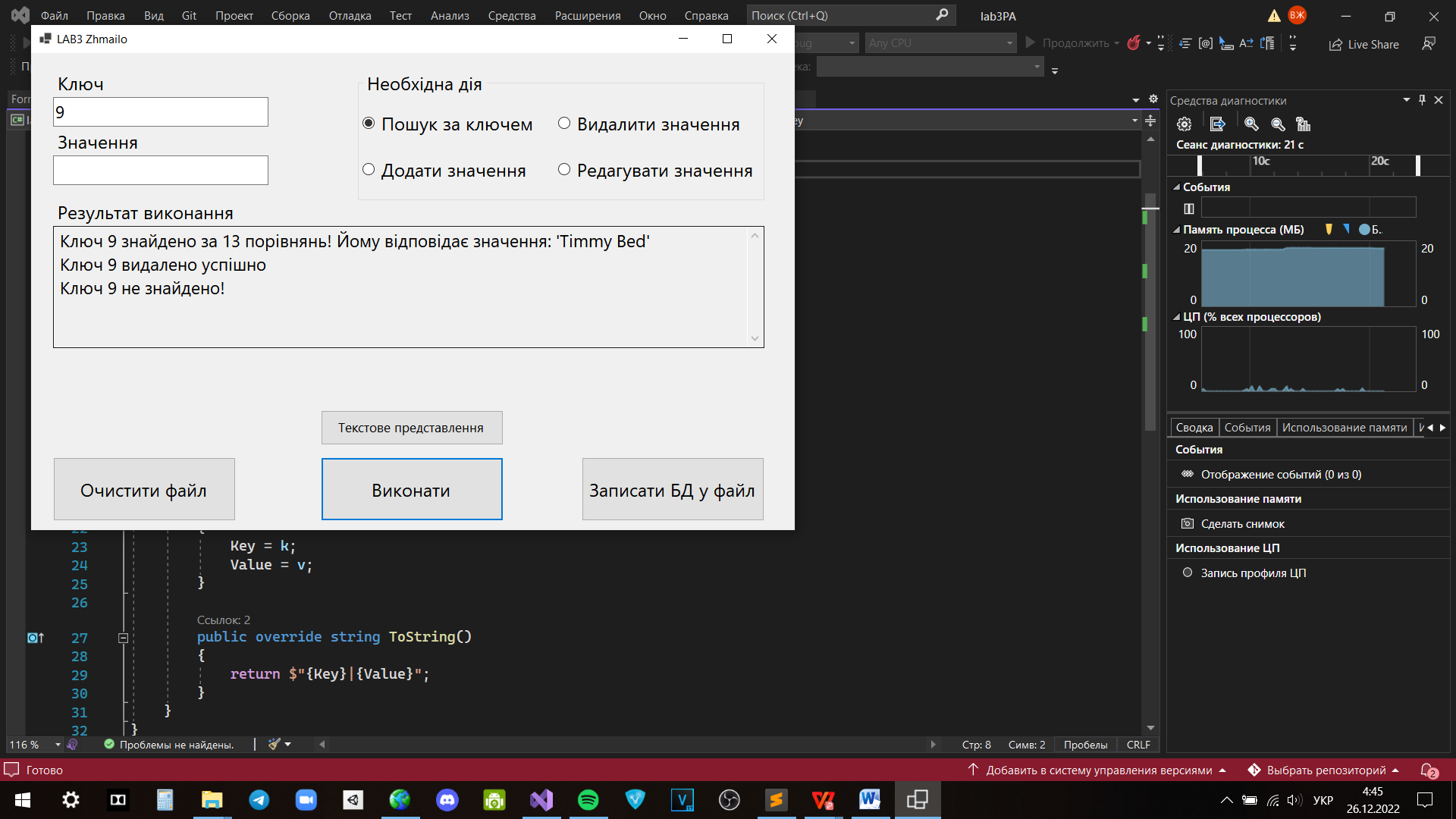


Рисунок 3.4 – Видалення запису

## Тестування алгоритму

### 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 | 16 |
| 2 | 16 |
| 3 | 21 |
| 4 | 13 |
| 5 | 19 |
| 6 | 16 |
| 7 | 19 |
| 8 | 16 |
| 9 | 19 |
| 10 | 13 |
| 11 | 10 |
| 12 | 22 |
| 13 | 16 |
| 14 | 13 |
| 15 | 19 |

Отже ми бачимо, що середня кількість порівнянь для алгоритму бінарного пошуку це ≈ 17 порівнянь.

Висновок

В рамках даної лабораторної роботи я дослідив як організовують збереження даних в програмах та розглянув такі структури даних, як червоно-чорні дерева, авл-дерева та B-дерева. Реалізував власну спрощену версію БД на основі B-дерева за допомогою мови програмування C#.

Розроблена програма має графічний інтерфейс користувача,

який дозволяє свторювати БД, шукати, додавати, редагувати та видаляти

значення звідти. Ця програма може використовуватися для збереження інформації типу ключ-значення на жорсткому диску комп'ютера.

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.