

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: Григоренко Родіон Ярославович

Перевірив

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

Київ 2022

ЗМІСТ

Варіант 5

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	В-дерево $t=10$, бінарний пошук
8	В-дерево $t=25$, бінарний пошук
9	В-дерево $t=50$, бінарний пошук
10	В-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	В-дерево $t=10$, однорідний бінарний пошук
18	В-дерево $t=25$, однорідний бінарний пошук
19	В-дерево $t=50$, однорідний бінарний пошук
20	В-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	В-дерево t=10, метод Шарра
28	В-дерево t=25, метод Шарра
29	В-дерево t=50, метод Шарра
30	В-дерево t=100, метод Шарра
31	АВЛ-дерево
32	Червоно-чорне дерево
33	В-дерево t=250, бінарний пошук
34	В-дерево t=250, однорідний бінарний пошук
35	В-дерево t=250, метод Шарра

ВИКОНАННЯ

Псевдокод алгоритмів

-Додавання запису

private Node InsertNode(Node current, Node node)

ПОЧАТОК

ЯКЩО node._key < current._key

ЯКЩО current._leftChild == -1

current._leftChild = node._position;

node._parent = current._position;

node._height = -1;

current = IncrementParentsHeight(current);

SetNode(current._position, current);

SetNode(node._position, node);

return GetNode(current._position);

current._leftChild =

InsertNode(GetNode(current._leftChild), node)._position;

```

        current = GetNode(current._position);
        current = BalanceTree(current);
    ІНАКШЕ ЯКЩО (node._key > current._key)
    ЯКЩО current._rightChild == -1
        current._rightChild = node._position;
        node._parent = current._position;
        node._height = -1;
        current = IncrementParentsHeight(current);
        SetNode(current._position, current);
        SetNode(node._position, node);
        return GetNode(current._position);
    current._rightChild = InsertNode(GetNode(current._rightChild),
    node)._position;
        current = GetNode(current._position);
        current = BalanceTree(current);

SetNode(current._position, current);
    return GetNode(current._position);
КІНЕЦЬ

```

-Пошук запису

```

private Node Find(int target, Node current)
ПОЧАТОК
ЯКЩО target < current._key

```

```
ЯКЩО target == current._key
return current;
ІНАКШЕ ЯКЩО current._leftChild == -1
return null;
ІНАКШЕ
return Find(target, GetNode(current._leftChild));
ІНАКШЕ
ЯКЩО target == current._key
return current;
ІНАКШЕ ЯКЩО current._rightChild == -1
return null;
ІНАКШЕ
return Find(target, GetNode(current._rightChild));

КІНЕЦЬ
```

-Видалення запису

```
private Node Delete(Node current, int target)
ПОЧАТОК
    Node parent;
    ЯКЩО current == null
        return null;
    ІНАКШЕ
        ЯКЩО target < current._key
```

```

    long oldChild = current._leftChild;
        current._leftChild =
Delete(GetNode(current._leftChild), target)._position;
    ЯКЩО oldChild == current._leftChild
current = GetNode(current._position);
    ИНАКШЕ
        SetNode(current._position, current);

    ЯКЩО current._leftChild == -1
        DecrementParentsHeight(current, 0);
    ЯКЩО BalanceFactor(current) == -2
    ЯКЩО BalanceFactor(GetNode(current._rightChild)) <=
0
        current = RotateRR(current, true);
            SetNode(current._position, current);
        ИНАКШЕ
            current = RotateRL(current);
                SetNode(current._position, current);

    ИНАКШЕ ЯКЩО target > current._key
        long oldChild = current._rightChild;
            current._rightChild =
Delete(GetNode(current._rightChild), target)._position;
        ЯКЩО oldChild == current._rightChild
        current = GetNode(current._position);
    ИНАКШЕ

```


SetNode(current._position, current);

ЯКЩО current._rightChild == -1

DecrementParentsHeight(current, 0);

ЯКЩО BalanceFactor(current) == 2

ЯКЩО BalanceFactor(GetNode(current._leftChild)) >= 0

current = RotateLL(current, true);

SetNode(current._position, current);

ІНАКШЕ

current = RotateLR(current);

SetNode(current._position, current);

ІНАКШЕ

ЯКЩО current._rightChild != -1

parent = GetNode(current._rightChild);

ПОКИ parent._leftChild != -1

parent = GetNode(parent._leftChild);

current._key = parent._key;

SetNode(current._position, current);

long oldChild = current._rightChild;

current._rightChild =

Delete(GetNode(current._rightChild), parent._key)._position;

ЯКЩО oldChild == current._rightChild

current = GetNode(current._position);

ІНАКШЕ

SetNode(current._position, current);

```

ЯКЩО current._rightChild == -1
DecrementParentsHeight(current, 0);
ЯКЩО BalanceFactor(current) == 2
ЯКЩО BalanceFactor(GetNode(current._leftChild)) >= 0
current = RotateLL(current, true);
                SetNode(current._position, current);

```

ІНАКШЕ

```

current = RotateLR(current);
                SetNode(current._position, current);

```

ІНАКШЕ

```

ЯКЩО current._leftChild != -1
_nodeToDelete = current;
                return GetNode(current._leftChild);

```

ІНАКШЕ

```

_nodeToDelete = current;
                Node n = new Node();
                n._position = -1;
                return n;

```

```

return GetNode(current._position);

```

Часова складність пошуку

Часова складність для даної задачі визначається у кількість звертань до диска. Таким чином використовуючи метод бінарного пошуку в АВЛ дереві можемо дійти висновку, що алгоритм пошуку запису має швидкість $O(\log_2(N))$, де N - це кількість вузлів у дереві, або $O(h)$, де h - висота дерева.

За рахунок того що дерево збалансоване, його висота менше ніж у звичаного бінарного дерева з такою ж кількістю вузлів, що забезпечує швидший пошук.

У звичайному ж незбалансованому бінарному дереві може бути випадок коли всі елементи розташовані в ряд і тоді швидкість пошуку буде оцінювати як $O(n)$, де n - кількість елементів в дереві.

Програмна реалізація

Node.cs

```
using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
using System;
```

```
[Serializable]
```

```
public class Node
```

```
{
```

```
    public long _position = -1;
```

```
    public long _parent = -1;
```

```
public long _leftChild = -1;
public long _rightChild = -1;
public int _key = -1;
public string _content;
public int _height = -1;
```

```
public int _contentSize = 50;
```

```
public Node()
{
```

```
}
```

```
public Node(int key, string content)
```

```
{
```

```
    _key = key;
```

```
    _content = content;
```

```
}
```

```
public byte[] Serialize()
```

```
{
```

```
    byte[] positionBytes = BitConverter.GetBytes(_position);
```

```
    byte[] parentBytes = BitConverter.GetBytes(_parent);
```

```
    byte[] leftChildBytes = BitConverter.GetBytes(_leftChild);
```

```
    byte[] rightChildBytes = BitConverter.GetBytes(_rightChild);
```

```
    byte[] keyBytes = BitConverter.GetBytes(_key);
```

```
byte[] contentBytes = new byte[_contentSize];  
byte[] stringBytes =  
System.Text.Encoding.ASCII.GetBytes(_content);  
  
for (int i = 0; i < contentBytes.Length; i++)  
{  
    if(i < stringBytes.Length)  
        contentBytes[i] = stringBytes[i];  
    else  
        contentBytes[i] = System.Text.Encoding.ASCII.GetBytes("  
")[0];  
}
```

```
byte[] heightBytes = BitConverter.GetBytes(_height);
```

```
int bufferSize = positionBytes.Length + parentBytes.Length +  
leftChildBytes.Length + rightChildBytes.Length + keyBytes.Length +  
contentBytes.Length + heightBytes.Length;
```

```
byte[] buffer = new byte[bufferSize];
```

```
int index = 0;
```

```
positionBytes.CopyTo(buffer, index);
```

```
index += positionBytes.Length;
```

```
parentBytes.CopyTo(buffer, index);
```

```
index += parentBytes.Length;
```

```
leftChildBytes.CopyTo(buffer, index);
```

```
index += leftChildBytes.Length;
```

```
rightChildBytes.CopyTo(buffer, index);
```

```
index += rightChildBytes.Length;
```

```
keyBytes.CopyTo(buffer, index);
```

```
index += keyBytes.Length;
```

```
contentBytes.CopyTo(buffer, index);
```

```
index += contentBytes.Length;
```

```
heightBytes.CopyTo(buffer, index);
```

```
index += heightBytes.Length;
```

```
return buffer;
```

```
}
```

```
public void Deserialize(byte[] buffer)
```

```
{
```

```
    int index = 0;
```

```
    _position = BitConverter.ToInt64(buffer[index..(index +  
sizeof(long))]);
```

```
    index += sizeof(long);
```

```
    _parent = BitConverter.ToInt64(buffer[index..(index +  
sizeof(long))]);
```

```
    index += sizeof(long);
```

```
    _leftChild = BitConverter.ToInt64(buffer[index..(index +  
sizeof(long))]);
```

```
    index += sizeof(long);
```

```
    _rightChild = BitConverter.ToInt64(buffer[index..(index +  
sizeof(long))]);
```

```
    index += sizeof(long);
```

```
    _key = BitConverter.ToInt32(buffer[index..(index +  
sizeof(int))]);
```

```
    index += sizeof(int);
```

```
    _content =  
System.Text.Encoding.ASCII.GetString(buffer[index..(index +  
_contentSize)]);
```

```
    index += _contentSize;
```

```
    _height = BitConverter.ToInt32(buffer[index..(index +  
sizeof(int))]);
```

```
    index += sizeof(int);
```

```
}  
}
```

AVLTree.cs

```
using System.Collections;  
using System.Collections.Generic;  
using System.IO;  
using UnityEngine;  
  
public class AVLTree  
{  
    public long _rootPosition = -1;  
  
    public int _contentSize = 50;  
  
    public int _nodeByteSize = sizeof(long) * 4 + sizeof(int) * 2 + 50;  
  
    private Node _nodeToDelete;  
    public string Add(Node node)  
    {  
        string message;  
  
        using (BinaryWriter binaryWriter = new  
BinaryWriter(File.Open("C:\\Users\\STRIX\\UnityProjects\\PA_LAB  
_3\\Assets\\Files\\Tree.dat", FileMode.Open)))
```



```

{
    node._position = binaryWriter.BaseStream.Length;
}
SetNode(node._position, node);

if (_rootPosition == -1)
{
    _rootPosition = node._position;
    RootChanged();
    message = "Node added as root";
}
else
{
    _rootPosition = InsertNode(GetNode(_rootPosition),
node)._position;
    message = "NodeAdded";
}
return message;
}

```

```

private Node InsertNode(Node current, Node node)

```

```

{
    if(node._key < current._key)
    {
        if(current._leftChild == -1)
        {

```

```

        current._leftChild = node._position;
        node._parent = current._position;
        node._height = -1;
        current = IncrementParentsHeight(current);
        SetNode(current._position, current);
        SetNode(node._position, node);
        return GetNode(current._position);
    }

    current._leftChild = InsertNode(GetNode(current._leftChild),
node)._position;
    current = GetNode(current._position);
    current = BalanceTree(current);
}
else if(node._key > current._key)
{
    if (current._rightChild == -1)
    {
        current._rightChild = node._position;
        node._parent = current._position;
        node._height = -1;
        current = IncrementParentsHeight(current);
        SetNode(current._position, current);
        SetNode(node._position, node);
        return GetNode(current._position);
    }
}

```

```

        current._rightChild =
InsertNode(GetNode(current._rightChild), node)._position;

        current = GetNode(current._position);
        current = BalanceTree(current);

        //Debug.Log(debug._key.ToString() + " " +
debug._height.ToString());
    }
    SetNode(current._position, current);
    return GetNode(current._position);
}

public void Delete(int target)
{
    _rootPosition = Delete(GetNode(_rootPosition),
target)._position;
    DeleteFromMemory(_nodeToDelete);
}

private Node Delete(Node current, int target)
{
    Node parent;
    if (current == null)
    { return null; }
    else
    {
        //left subtree
        if (target < current._key)
        {

```

```

    long oldChild = current._leftChild;
    current._leftChild = Delete(GetNode(current._leftChild),
target)._position;
    if (oldChild == current._leftChild)
        current = GetNode(current._position);
    else
        SetNode(current._position, current);
    if (current._leftChild == -1)
    {
        DecrementParentsHeight(current, 0);
    }
    if (BalanceFactor(current) == -2)//here
    {
        if (BalanceFactor(GetNode(current._rightChild)) <= 0)
        {
            current = RotateRR(current, true);
            SetNode(current._position, current);
        }
        else
        {
            current = RotateRL(current);
            SetNode(current._position, current);
        }
    }
}
//right subtree

```

```

else if (target > current._key)
{
    long oldChild = current._rightChild;
    current._rightChild = Delete(GetNode(current._rightChild),
target)._position;
    if (oldChild == current._rightChild)
        current = GetNode(current._position);
    else
        SetNode(current._position, current);
    if (current._rightChild == -1)
    {
        DecrementParentsHeight(current, 0);
    }
    if (BalanceFactor(current) == 2)
    {
        if (BalanceFactor(GetNode(current._leftChild)) >= 0)
        {
            current = RotateLL(current, true);
            SetNode(current._position, current);
        }
        else
        {
            current = RotateLR(current);
            SetNode(current._position, current);
        }
    }
}

```

```

}
//if target is found
else
{
    if (current._rightChild != -1)
    {
        parent = GetNode(current._rightChild);
        while (parent._leftChild != -1)
        {
            parent = GetNode(parent._leftChild);
        }
        current._key = parent._key;
        SetNode(current._position, current);
        long oldChild = current._rightChild;
        current._rightChild =
Delete(GetNode(current._rightChild), parent._key)._position;
        if (oldChild == current._rightChild)
            current = GetNode(current._position);
        else
            SetNode(current._position, current);
        if (current._rightChild == -1)
        {
            DecrementParentsHeight(current, 0);
        }
        if (BalanceFactor(current) == 2)//rebalancing
        {

```

```

    if (BalanceFactor(GetNode(current._leftChild)) >= 0)
    {
        current = RotateLL(current, true);
        SetNode(current._position, current);
    }
    else
    {
        current = RotateLR(current);
        SetNode(current._position, current);
    }
}

else
{
    if (current._leftChild != -1)
    {
        _nodeToDelete = current;
        return GetNode(current._leftChild);
    }
    else
    {
        _nodeToDelete = current;
        Node n = new Node();
        n._position = -1;
        return n;
    }
}

```

```

        }
    }
}
return GetNode(current._position);
}
public string Find(int key)
{
    Node node = Find(key, GetNode(_rootPosition));
    if (node == null)
    {
        return "Key is not found";
    }
    else
    {
        return node._content;
    }
}

private Node Find(int target, Node current)
{
    if(target < current._key)
    {
        if(target == current._key)
        {
            return current;

```



```

    }
    else if(current._leftChild == -1)
    {
        return null;
    }
    else
    {
        return Find(target, GetNode(current._leftChild));
    }
}
else
{
    if (target == current._key)
    {
        return current;
    }
    else if (current._rightChild == -1)
    {
        return null;
    }
    else
    {
        return Find(target, GetNode(current._rightChild));
    }
}
}

```

```

private Node BalanceTree(Node current)
{
    current = GetNode(current._position);
    int bFactor = BalanceFactor(current);
    if (bFactor > 1)
    {
        if (BalanceFactor(GetNode(current._leftChild)) > 0)
        {
            current = RotateLL(current, true);
        }
        else
        {
            current = RotateLR(current);
        }
    }
    else if(bFactor < -1)
    {
        if(BalanceFactor(GetNode(current._rightChild)) > 0)
        {
            current = RotateRL(current);
        }
        else
        {
            current = RotateRR(current, true);

            //Debug.Log(current._key.ToString() + " " +
current._height.ToString());

```

```

    }
}

SetNode(current._position, current);
return GetNode(current._position);
}

private Node RotateLL(Node parent, bool changeHeight)
{
    Node pivot = GetNode(parent._leftChild);
    parent._leftChild = pivot._rightChild;
    pivot._rightChild = parent._position;
    pivot._parent = parent._parent;
    parent._parent = pivot._position;
    if (pivot._parent == -1)
    {
        _rootPosition = pivot._position;
        RootChanged();
    }
    else
    {
        Node grandparent = GetNode(pivot._parent);
        if (pivot._key > grandparent._key)
            grandparent._rightChild = pivot._position;
        else
        {

```

```

        grandparent._leftChild = pivot._position;
    }
    SetNode(grandparent._position, grandparent);
}
pivot._height++;
if (!changeHeight)
    parent._height--;
SetNode(parent._position, parent);
SetNode(pivot._position, pivot);
if (changeHeight)
{
    parent._height--;
    SetNode(parent._position, parent);
    DecrementParentsHeight(parent, 0);
}
return GetNode(pivot._position);
}

```

```

private Node RotateLR(Node parent)
{
    Node pivot = GetNode(parent._leftChild);
    parent._leftChild = RotateRR(pivot,false)._position;
    return RotateLL(parent, true);
}

```

```

private Node RotateRL(Node parent)

```

```

{
    Node pivot = GetNode(parent._rightChild);
    parent._rightChild = RotateLL(pivot, false)._position;
    return RotateRR(parent, true);
}

```

private Node RotateRR(Node parent, bool changeHeight)

```

{
    Node pivot = GetNode(parent._rightChild);
    parent._rightChild = pivot._leftChild;
    pivot._leftChild = parent._position;
    pivot._parent = parent._parent;
    parent._parent = pivot._position;
    if(pivot._parent == -1)
    {
        _rootPosition = pivot._position;
        RootChanged();
    }
    else
    {
        Node grandparent = GetNode(pivot._parent);
        if(pivot._key > grandparent._key)
            grandparent._rightChild = pivot._position;
        else
        {
            grandparent._leftChild = pivot._position;

```

```

    }
    SetNode(grandparent._position, grandparent);
}
pivot._height++;
SetNode(pivot._position, pivot);
if (!changeHeight)
    parent._height--;
SetNode(parent._position, parent);
SetNode(pivot._position, pivot);
if (changeHeight)
{
    parent._height--;
    SetNode(parent._position, parent);
    DecrementParentsHeight(parent, 0);
}
//Debug.Log(pivot._key.ToString() + " " +
pivot._height.ToString());
return GetNode(pivot._position);
}

```

```

private Node DecrementParentsHeight(Node current, int
lastBalanceFactor)

```

```

{
    Node node = new Node();
    int newLastBalanceFactor = 0;
    if (current._parent != -1)

```

```

{
    node = GetNode(current._parent);
    newLastBalanceFactor = Mathf.Abs(BalanceFactor(node));
}
current._height--;
SetNode(current._position, current);
//if(current._key == 2)
    //Debug.Log(current._key.ToString() + " " +
current._height.ToString());
    if (current._parent == -1)
    {
        if (Mathf.Abs(BalanceFactor(GetNode(current._position))) ==
1 && lastBalanceFactor != 2)
            current._height++;
        SetNode(current._position, current);
        Debug.Log(current._key.ToString() + " " +
current._height.ToString());
        return GetNode(current._position);
    }
    else if (Mathf.Abs(BalanceFactor(GetNode(current._position)))
== 1 && lastBalanceFactor != 2)
    {
        current._height++;
        SetNode(current._position, current);
        Debug.Log(current._key.ToString() + " " +
current._height.ToString());
        return GetNode(current._position);
    }

```

```

    }
    lastBalanceFactor = newLastBalanceFactor;
    //Debug.Log(node._key.ToString() + " " +
node._height.ToString());
    DecrementParentsHeight(node, lastBalanceFactor);
    SetNode(current._position, current);
    Debug.Log(current._key.ToString() + " " +
current._height.ToString());
    return GetNode(current._position);
}

```

```

private Node IncrementParentsHeight(Node current)
{
    current._height++;
    SetNode(current._position, current);
    if (current._parent == -1)
    {
        if (BalanceFactor(GetNode(current._position)) == 0)
            current._height--;
        SetNode(current._position, current);
        //Debug.Log(current._key.ToString() + " " +
current._height.ToString());
        return GetNode(current._position);
    }
}

```



```

else if(BalanceFactor(GetNode(current._parent)) == 0)
{
    SetNode(current._position, current);

    Debug.Log(current._key.ToString() + " " +
current._height.ToString());

    return GetNode(current._position);
}

Node node = new Node();
if (current._parent != -1)
{
    node = GetNode(current._parent);
}

IncrementParentsHeight(node);

SetNode(current._position, current);

//Debug.Log(current._key.ToString() + " " +
current._height.ToString());

return GetNode(current._position);
}

```

```

private int BalanceFactor(Node current)
{
    int l;
    int r;
    if (current._leftChild != -1)
        l = GetNode(current._leftChild)._height;
    else

```

```

        l = -2;
    if (current._rightChild != -1)
        r = GetNode(current._rightChild)._height;
    else
        r = -2;
    return l - r;
}

```

```

public Node GetNode(long position)
{
    Node node = new Node();
    using (BinaryReader binaryReader = new
BinaryReader(File.Open("C:\\Users\\STRIX\\UnityProjects\\PA_LA
B_3\\Assets\\Files\\Tree.dat", FileMode.Open)))
    {
        binaryReader.BaseStream.Seek(position, SeekOrigin.Begin);
        byte[] buffer = binaryReader.ReadBytes(_nodeByteSize);
        node.Deserialize(buffer);
    }
    return node;
}

```

```

public void SetNode(long position, Node node)
{
    using (BinaryWriter binaryWriter = new
BinaryWriter(File.Open("C:\\Users\\STRIX\\UnityProjects\\PA_LAB
_3\\Assets\\Files\\Tree.dat", FileMode.Open)))

```

```

    {
        binaryWriter.BaseStream.Seek(position, SeekOrigin.Begin);
        node._position = position;
        byte[] buffer = node.Serialize();
        binaryWriter.Write(buffer);
    }
}

```

```

private void RootChanged()
{
    using(BinaryWriter binaryWriter = new
BinaryWriter(File.Open("C:\\Users\\STRIX\\UnityProjects\\PA_LAB
_3\\Assets\\Files\\Root.dat", FileMode.Open)))
    {
        binaryWriter.Write(_rootPosition);
    }
}

```

```

public void DeleteFromMemory(Node node)
{
    long fileLength;

    using (BinaryWriter binaryWriter = new
BinaryWriter(File.Open("C:\\Users\\STRIX\\UnityProjects\\PA_LAB
_3\\Assets\\Files\\Tree.dat", FileMode.Open)))
    {
        fileLength = binaryWriter.BaseStream.Length;
    }
}

```

```

Node lastNode = GetNode(fileLength - _nodeByteSize);
if(lastNode._position == _rootPosition)
{
    _rootPosition = node._position;
}
lastNode._position = node._position;
if (lastNode._parent != -1)
{
    Node parent = GetNode(lastNode._parent);
    if (lastNode._key > parent._key)
    {
        parent._rightChild = lastNode._position;
    }
    else
    {
        parent._leftChild = lastNode._position;
    }
    SetNode(parent._position, parent);
}
SetNode(lastNode._position, lastNode);

using (BinaryWriter binaryWriter = new
BinaryWriter(File.Open("C:\\Users\\STRIX\\UnityProjects\\PA_LAB
_3\\Assets\\Files\\Tree.dat", FileMode.Open)))
{
    binaryWriter.BaseStream.SetLength(fileLength -
_nodeByteSize);

```

```
    }  
  }  
}
```

InterfaceManager.cs

```
using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
using System.IO;  
using System;  
using System.Runtime.Serialization.Formatters.Binary;  
using TMPro;  
  
public class InterfaceManager : MonoBehaviour  
{  
    private AVLTree AVLTree = new AVLTree();  
  
    [SerializeField] Transform[] UINodes;  
  
    [SerializeField] private TMP_InputField _inputFindKey;  
    [SerializeField] private TextMeshProUGUI _foundContent;  
  
    [SerializeField] private TMP_InputField _inputAddKey;  
    [SerializeField] private TMP_InputField _inputAddContent;
```

```

[SerializeField] private TMP_InputField _inputDeleteKey;
[SerializeField] private TextMeshProUGUI _addingMessage;

// Start is called before the first frame update
void Start()
{
    using (Stream stream =
File.Open("C:\\Users\\STRIX\\UnityProjects\\PA_LAB_3\\Assets\\Files\\Tree.dat", FileMode.OpenOrCreate))
    {

    }

    using(BinaryReader binaryReader = new
BinaryReader(File.Open("C:\\Users\\STRIX\\UnityProjects\\PA_LAB_3\\Assets\\Files\\Root.dat", FileMode.OpenOrCreate)))
    {
        if(binaryReader.BaseStream.Length != 0)
        {
            AVLTree._rootPosition = binaryReader.ReadInt64();
        }
    }
    ShowTree();
}

// Update is called once per frame
void Update()
{

```

```
}
```

```
public void Add()
{
    int key;
    if(int.TryParse(_inputAddKey.text, out key))
    {

    }
    else
    {
        throw new Exception("Key didn't parse");
    }
    string content = _inputAddContent.text;
    Node node = new Node(key, content);
    string message = AVLTree.Add(node);
    _addingMessage.text = message;
    Node debug = AVLTree.GetNode(AVLTree._rootPosition);
    //Debug.Log(debug._key.ToString() + " " +
debug._height.ToString());
    ShowTree();
}
```

```
public void Delete()
```

```
{
```

```
    Int32.TryParse(_inputDeleteKey.text, out int deleteKey);
    AVLTree.Delete(deleteKey);
    ShowTree();
}
```

```
public void Find()
{
    Int32.TryParse(_inputFindKey.text, out int targetKey);
    string content = AVLTree.Find(targetKey);
    _foundContent.text = content;
}
```

```
public void ShowTree()
{
    Node root = null;
    if (AVLTree._rootPosition != -1)
    {
        root = AVLTree.GetNode(AVLTree._rootPosition);
    }
}
```

```
UINodes[0].GetChild(0).GetComponent<TextMeshProUGUI>().text
= root._key.ToString();
```

```
UINodes[0].GetChild(1).GetComponent<TextMeshProUGUI>().text
= root._height.ToString();
```

```
}
```

```
else
```

```
{
```



```
UINodes[0].GetChild(0).GetComponent<TextMeshProUGUI>().text  
= "NULL";
```

```
UINodes[0].GetChild(1).GetComponent<TextMeshProUGUI>().text  
= "NULL";
```

```
}
```

```
Node first = null, second = null, third = null, fourth = null, fifth =  
null, sixth = null;
```

```
if (root != null)
```

```
{
```

```
    if (root._leftChild != -1)
```

```
    {
```

```
        first = AVLTree.GetNode(root._leftChild);
```

```
    }
```

```
    else
```

```
    {
```

```
UINodes[1].GetChild(0).GetComponent<TextMeshProUGUI>().text  
= "NULL";
```

```
UINodes[1].GetChild(1).GetComponent<TextMeshProUGUI>().text  
= "NULL";
```

```
}
```

```
if (root._rightChild != -1)
```

```
{
```

```
    second = AVLTree.GetNode(root._rightChild);
```

```
}
```

```

        else
        {

UINodes[2].GetChild(0).GetComponent<TextMeshProUGUI>().text
= "NULL";

UINodes[2].GetChild(1).GetComponent<TextMeshProUGUI>().text
= "NULL";

        }
    }
    if(first != null)
    {

UINodes[1].GetChild(0).GetComponent<TextMeshProUGUI>().text
= first._key.ToString();

UINodes[1].GetChild(1).GetComponent<TextMeshProUGUI>().text
= first._height.ToString();
        if (first._leftChild != -1)
        {
            third = AVLTree.GetNode(first._leftChild);

UINodes[3].GetChild(0).GetComponent<TextMeshProUGUI>().text
= third._key.ToString();

UINodes[3].GetChild(1).GetComponent<TextMeshProUGUI>().text
= third._height.ToString();
        }
    }
    else

```

```

    {

    UINodes[3].GetChild(0).GetComponent<TextMeshProUGUI>().text
    = "NULL";

    UINodes[3].GetChild(1).GetComponent<TextMeshProUGUI>().text
    = "NULL";

    }

    if(first._rightChild != -1)
    {

        fourth = AVLTree.GetNode(first._rightChild);

    UINodes[4].GetChild(0).GetComponent<TextMeshProUGUI>().text
    = fourth._key.ToString();

    UINodes[4].GetChild(1).GetComponent<TextMeshProUGUI>().text
    = fourth._height.ToString();

    }

    else
    {

    UINodes[4].GetChild(0).GetComponent<TextMeshProUGUI>().text
    = "NULL";

    UINodes[4].GetChild(1).GetComponent<TextMeshProUGUI>().text
    = "NULL";

    }

    }

    if(second != null)

```

```

{

UINodes[2].GetChild(0).GetComponent<TextMeshProUGUI>().text
= second._key.ToString();

UINodes[2].GetChild(1).GetComponent<TextMeshProUGUI>().text
= second._height.ToString();
    if (second._leftChild != -1)
    {
        fifth = AVLTree.GetNode(second._leftChild);

UINodes[5].GetChild(0).GetComponent<TextMeshProUGUI>().text
= fifth._key.ToString();

UINodes[5].GetChild(1).GetComponent<TextMeshProUGUI>().text
= fifth._height.ToString();
    }
    else
    {

UINodes[5].GetChild(0).GetComponent<TextMeshProUGUI>().text
= "NULL";

UINodes[5].GetChild(1).GetComponent<TextMeshProUGUI>().text
= "NULL";
    }
    if (second._rightChild != -1)
    {
        sixth = AVLTree.GetNode(second._rightChild);

```

```
UINodes[6].GetChild(0).GetComponent<TextMeshProUGUI>().text  
= sixth._key.ToString();
```

```
UINodes[6].GetChild(1).GetComponent<TextMeshProUGUI>().text  
= sixth._height.ToString();
```

```
    }
```

```
    else
```

```
    {
```

```
UINodes[6].GetChild(0).GetComponent<TextMeshProUGUI>().text  
= "NULL";
```

```
UINodes[6].GetChild(1).GetComponent<TextMeshProUGUI>().text  
= "NULL";
```

```
    }
```

```
    }
```

```
}
```

Приклад роботи

На рисунках зображено приклади роботи програми при додаванні, пошуку та видаленні елементів дерева.

В графічному зображенні нод записані: зверху ключ, знизу висота.

Додавання елемента:

To add node enter:
Key:
Content:

Adding message:
NodeAdded

1
-1

2
0

3
-1

NULL
NUL
L

NULL
NUL
L

NULL
NUL
L

NULL
NUL
L

Enter key to find record:

Found content:

Enter key to delete record:

To add node enter:
Key:
Content:

Adding message:
NodeAdded

1
-1

2
1

3
0

4
-1

NULL
NUL
L

NULL
NUL
L

NULL
NUL
L

Enter key to find record:

Found content:

Enter key to delete record:

Пошук елемента:

To add node enter:
Key:
Content:

Adding message:
NodeAdded

4
1

2
0

6
0

1
-1

3
-1

5
-1

7
-1

Enter key to find record:

Found content:

Enter key to delete record:

To add node enter:
Key:
Content:

Adding message:
NodeAdded

4
1

2
0

6
0

1
-1

3
-1

5
-1

7
-1

Enter key to find record:

Found content:
9

Enter key to delete record:

Видалення елемента:

To add node enter:

Key:

Content:

Add

Adding message:

NodeAdded

Enter key to find record:

Find

Found content:

g

Enter key to delete record:

Delete

4
1

2
0

6
0

1
-1

3
-1

5
-1

7
-1

To add node enter:

Key:

Content:

Add

Adding message:

NodeAdded

Enter key to find record:

Find

Found content:

g

Enter key to delete record:

Delete

5
1

2
0

6
0

1
-1

3
-1

NULL
NUL
L

7
-1

Тестування алгоритму

В таблиці наведено кількість пройдених вузлів для 15 спроб пошуку запису по ключу.

Номер спроби пошуку	Кількість пройдених вузлів
1	3
2	3
3	1
4	2
5	3
6	2
7	3
8	4
9	5
10	4
11	4
12	3
13	5
14	4
15	5

ВИСНОВОК

На лабораторній роботі було вивчено основні підходи проектування та обробки складних структур даних, було реалізовано програмне забезпечення, що дає змогу здійснювати пошук, додавання, редагування і видалення даних, в основі якого лежить бінарне АВЛ дерево. У ході роботи було досліджено алгоритм пошуку і ми дійшли

висновку, що швидкість знаходження вузла залежить як $O(\log_2(N))$ від кількості вузлів або як $O(h)$ від висоти дерева. Середня кількість пройдених вузлів при пошуку запису: 3.4 .