Лабораторная работа №4

Лабораторная работа №4

Дерево поиска. Частотный словарь.

Составить программу, реализующую частотный словарь текста.

Программа должна производить следующие действия;

1. Выделять в выбранном текстовом файле отдельные слова.

2. Помещать слова в дерево поиска.

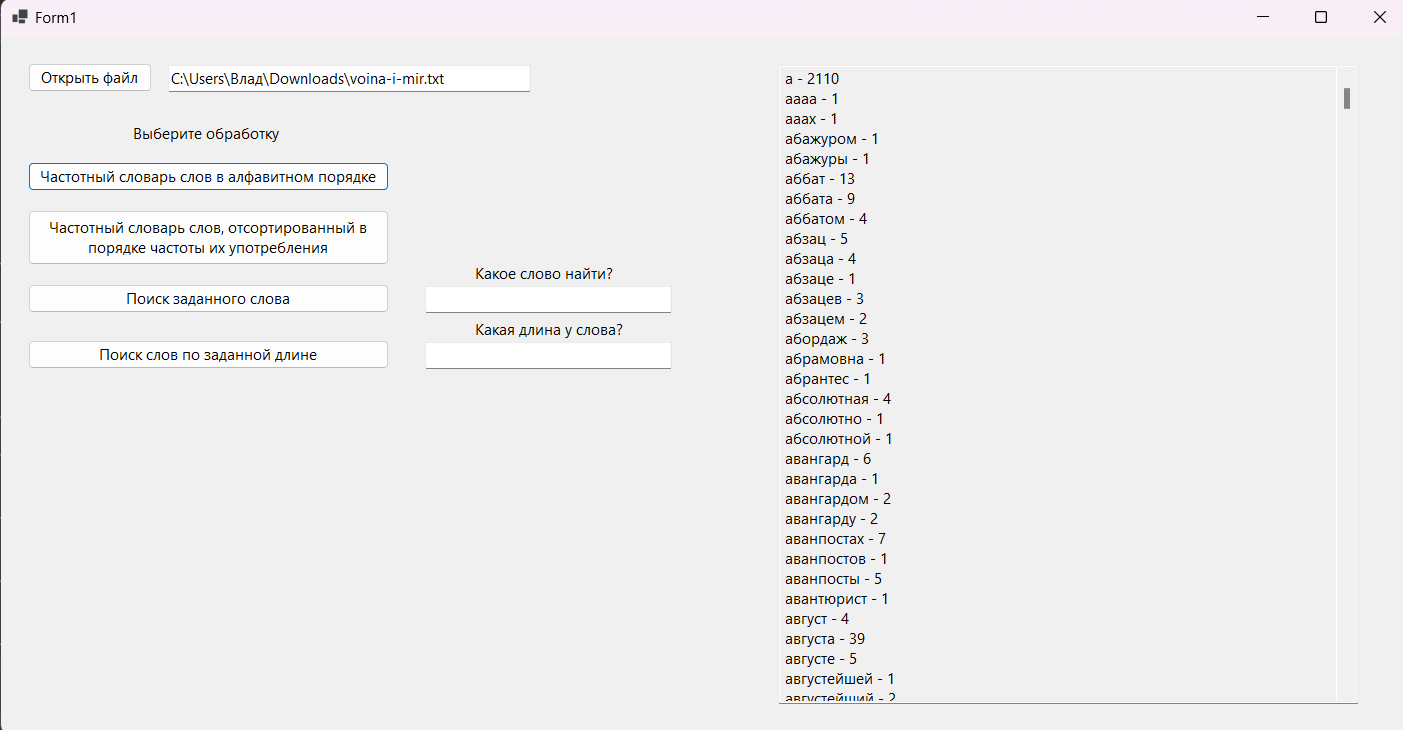
3. Выводить из дерева частотный словарь слов в алфавитном порядке.

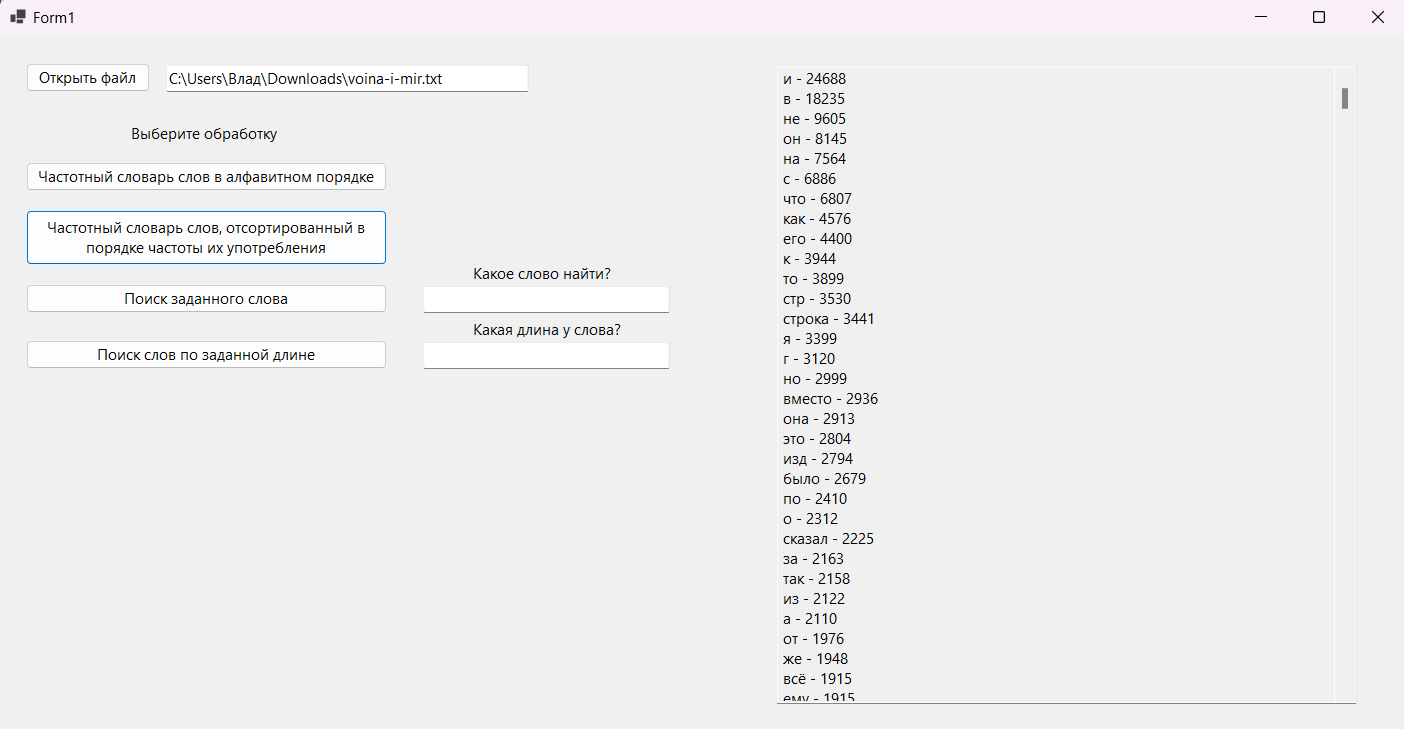
4. Выводить частотный словарь слов, отсортированный в порядке частоты их употребления.

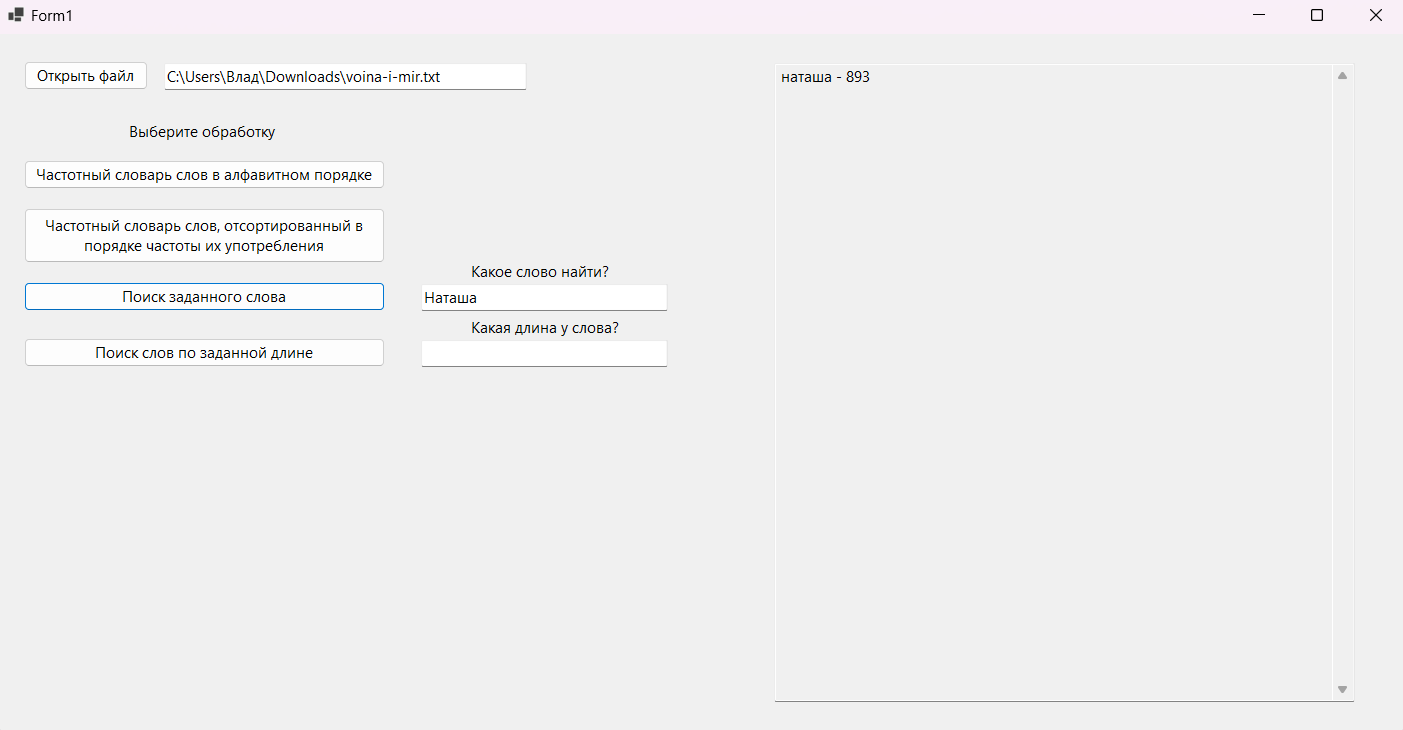
5. Производить поиск заданного слова.

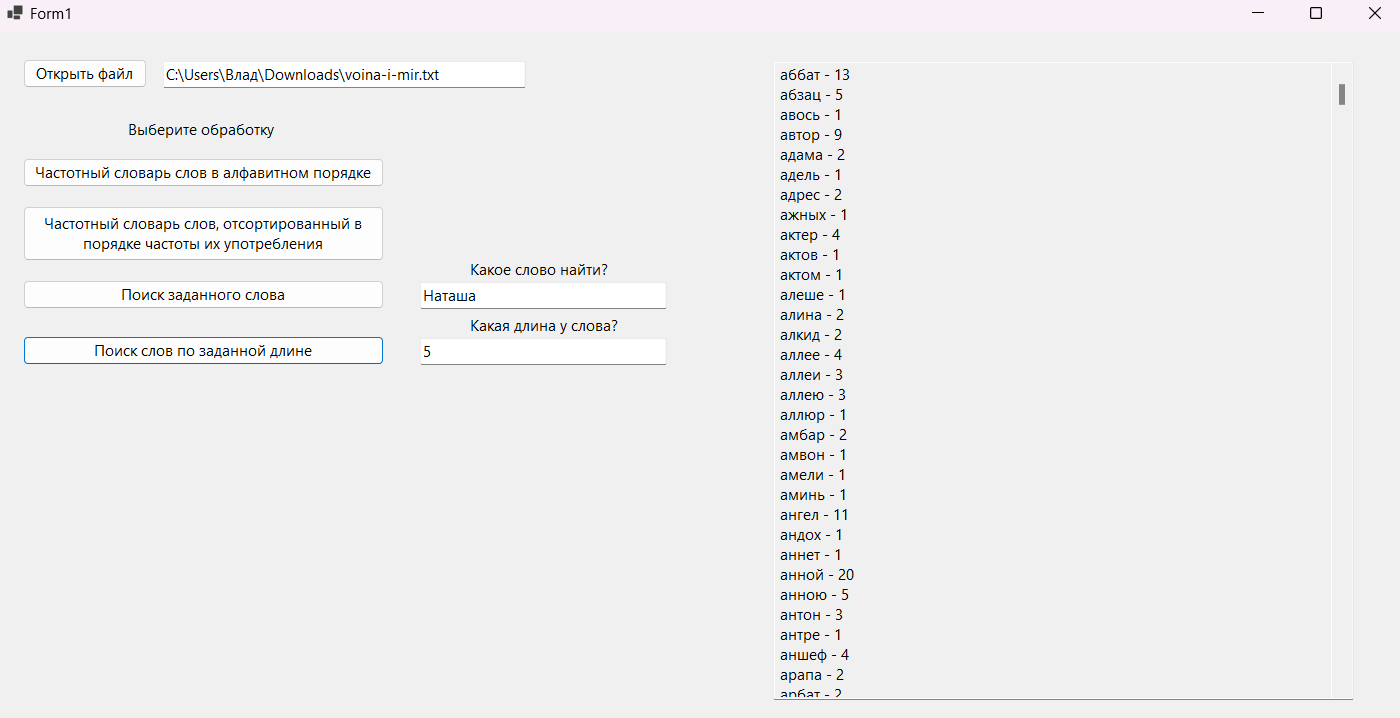
6. Производить фильтрацию, оставляющую в словаре слова определенной длины.

Тесты:



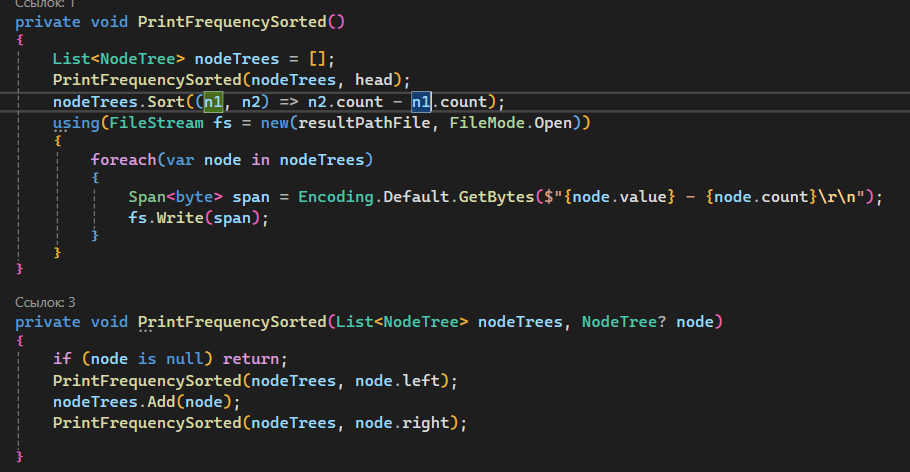






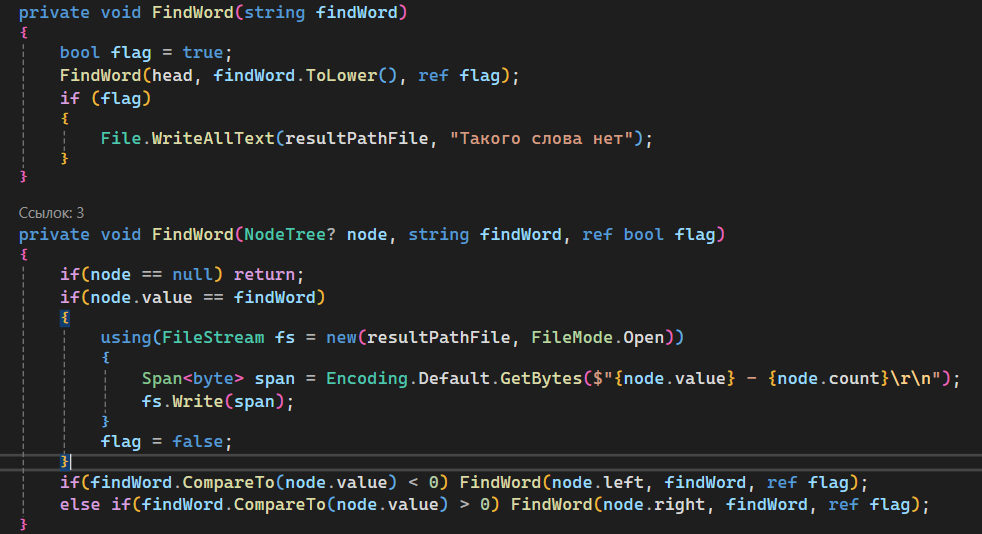
Алгоритм:

Вывод частотного словаря слов, отсортированного в порядке частоты их употребления



Вывод частотного словаря в порядке частоте употребления работает добавления всех узлов дерева в список через рекурсию, а потом сортируем его по полю count в порядке убывания. Далее записываем в все частоты в файл.

Поиск слова:

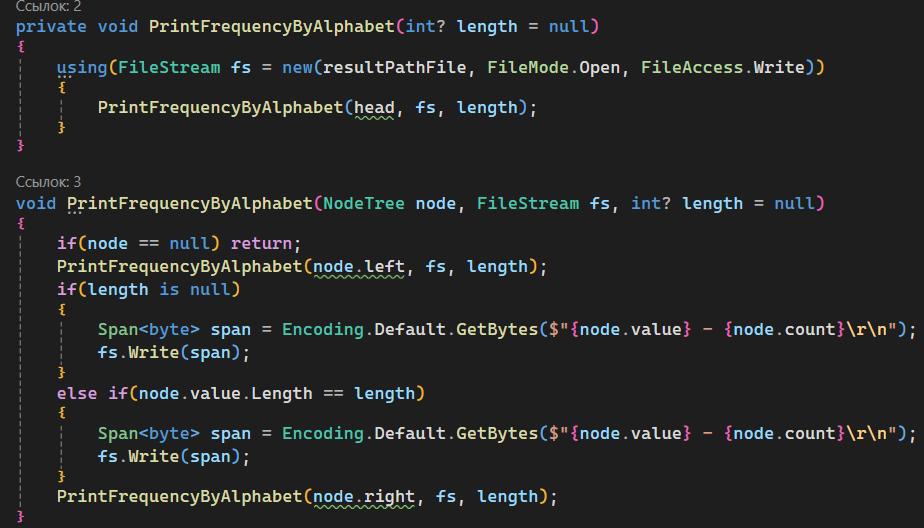


Поиск работает так же через рекурсию. Крайние случаи: узел равен null или значение узла равно слову, которое ищем. В первом случае выходим из тела метода. Во втором случае открываем файл и записываем в него слово и его частоту.

Рекурсивный случай: проверяем в какой стороне нам нужно искать слово. Будем делать это по сравнению слов: значение узла и слово, которое ищем. Если слово, которое ищем меньше значения узла, то идем влево. Если больше, то вправо.

Также в метод передается ссылка на булеву переменную. Она нужна для того, чтобы после выполнения метода проверить нашли ли мы слова. Если нет, то записываем в файл, что данного слова нет в файле.

Вывод по алфавиту и указанной длине



вывод дерева делаем по алфавиту делаем так же через рекурсию.

Крайних случаев три: узел является null, когда мы выводим по алфавиту и записываем файл слово с его частотой, когда мы выводим по длине слова и длина значения узла равна заданной длине.

Сам вывод работает по симметричному выводу(т.е сначала левое поддерево, корень, правое поддерево).

BinaryTree.cs

using System.Text;

using System.Xml.Linq;

namespace Lr2

{

public class BinaryTree

{

public class NodeTree(string data)

{

public string value = data;

public NodeTree? left;

public NodeTree? right;

public int count = 1;

}

public NodeTree? head;

public BinaryTree()

{

File.WriteAllText(resultPathFile, "");

}

private readonly char[] splitter = { ' ', ',', '.', '!', ':', ';', '?', '–', '—', ' ', '―', '°', '\*', '\*', '[', '-', '\n', '\r', '\t', ']', '(', ')', '…', '«', '»', '“', '“', '\'', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '„', '‘', '’', ' ', '\_', '\"', '/', '&', '=' };

public readonly string resultPathFile = @"C:\Users\Влад\Desktop\Университет\Второй курс\Структуры и алгоритмы\result.txt";

public void Add(string data) => head = Add(head, data);

private NodeTree Add(NodeTree? tree, string data)

{

var tmp = tree;

if (tree == null)

{

return new NodeTree(data);

}

var comparer = data.CompareTo(tree.value);

if (comparer < 0) tmp.left = Add(tree.left, data);

else if (comparer > 0) tmp.right = Add(tree.right, data);

else tree.count++;

return tmp!;

}

public void ReadFile(string path, PrintMode printMode, string? findWord = null, int? length = null)

{

using(StreamReader sr = new(path))

{

StringBuilder word = new();

int numberSymbol = sr.Read();

while(sr.Peek() != -1)

{

char symbol = (char)numberSymbol;

if (splitter.Contains(symbol))

{

if(word.ToString() != "")

Add(word.ToString().ToLower());

word.Clear();

numberSymbol = sr.Read();

continue;

}

word.Append(symbol);

numberSymbol = sr.Read();

}

}

switch (printMode)

{

case PrintMode.PrintFrequencyByAlphabet:

PrintFrequencyByAlphabet();

break;

case PrintMode.PrintFrequencySorted:

PrintFrequencySorted();

break;

case PrintMode.FindWord:

FindWord(findWord!);

break;

case PrintMode.PrintFrequencyByLength:

PrintFrequencyByAlphabet(length);

break;

}

}

private void PrintFrequencySorted()

{

List<NodeTree> nodeTrees = [];

PrintFrequencySorted(nodeTrees, head);

nodeTrees.Sort((n1, n2) => n2.count - n1.count);

using(FileStream fs = new(resultPathFile, FileMode.Open))

{

foreach(var node in nodeTrees)

{

Span<byte> span = Encoding.Default.GetBytes($"{node.value} - {node.count}\r\n");

fs.Write(span);

}

}

}

private void PrintFrequencySorted(List<NodeTree> nodeTrees, NodeTree? node)

{

if (node is null) return;

PrintFrequencySorted(nodeTrees, node.left);

nodeTrees.Add(node);

PrintFrequencySorted(nodeTrees, node.right);

}

private void FindWord(string findWord)

{

bool flag = true;

FindWord(head, findWord.ToLower(), ref flag);

if (flag)

{

File.WriteAllText(resultPathFile, "Такого слова нет");

}

}

private void FindWord(NodeTree? node, string findWord, ref bool flag)

{

if(node == null) return;

if(node.value == findWord)

{

using(FileStream fs = new(resultPathFile, FileMode.Open))

{

Span<byte> span = Encoding.Default.GetBytes($"{node.value} - {node.count}\r\n");

fs.Write(span);

}

flag = false;

}

if(findWord.CompareTo(node.value) < 0) FindWord(node.left, findWord, ref flag);

else if(findWord.CompareTo(node.value) > 0) FindWord(node.right, findWord, ref flag);

}

private void PrintFrequencyByAlphabet(int? length = null)

{

using(FileStream fs = new(resultPathFile, FileMode.Open, FileAccess.Write))

{

PrintFrequencyByAlphabet(head, fs, length);

}

}

void PrintFrequencyByAlphabet(NodeTree node, FileStream fs, int? length = null)

{

if(node == null) return;

PrintFrequencyByAlphabet(node.left, fs, length);

if(length is null)

{

Span<byte> span = Encoding.Default.GetBytes($"{node.value} - {node.count}\r\n");

fs.Write(span);

}

else if(node.value.Length == length)

{

Span<byte> span = Encoding.Default.GetBytes($"{node.value} - {node.count}\r\n");

fs.Write(span);

}

PrintFrequencyByAlphabet(node.right, fs, length);

}

/// <summary>Прямой вывод дерева.</summary>

public void PrintTreeS()

{

PrintTreeS(head);

Console.WriteLine("\n");

}

private void PrintTreeS(NodeTree? tree)

{

if (tree == null)

return;

Console.Write($"{tree.value} ");

PrintTreeS(tree.left);

PrintTreeS(tree.right);

}

public void PrintTreeR()

{

PrintTreeR(head);

Console.WriteLine("\n");

}

private void PrintTreeR(NodeTree? tree)

{

if (tree == null)

return;

PrintTreeR(tree.left);

PrintTreeR(tree.right);

Console.Write($"{tree.value} ");

}

public void PrintTreeC()

{

PrintTreeC(head);

Console.WriteLine("\n");

}

private void PrintTreeC(NodeTree? tree)

{

if (tree == null)

return;

PrintTreeC(tree.left);

Console.Write($"{tree.value} ");

PrintTreeC(tree.right);

}

public BinaryTree Copy()

{

var copy = Copy(head);

var result = new BinaryTree();

result.head = copy;

return result;

}

private NodeTree? Copy(NodeTree? node)

{

if (node == null)

return null;

NodeTree result = new(node.value);

result.left = Copy(node.left);

result.right = Copy(node.right);

return result;

}

}

}