Министерство науки и высшего образования Российской Федерации
Муромский институт (филиал)
федерального государственного бюджетного образовательного учреждения высшего образования
**«Владимирский государственный университет
Имени Александра Григорьевича и Николая Григорьевича Столетовых»
(МИВлГУ)**

Факультет ИТ

Кафедра ПИн

ЛАБОРАТОРНАЯ РАБОТА №4

по САОД

Тема Бинарные деревья поиска

Руководитель

Привезенцев Д.Г.
(фамилия, инициалы)

(подпись) (дата)

Студент ПИн-119
(группа)

Лямина И.А.
(фамилия, инициалы)

(подпись) (дата)

Муром 2021

Лабораторная работа №4

Тема: Бинарные деревья поиска.

Задание на лабораторную работу

1. Создание бинарного дерева поиска. Необходимо создать бинарное дерево из *n* элементов (*n* вводится с клавиатуры), элементы дерева задаются случайным образом и добавляются в дерево по следующему принципу:

1) первый элемент добавляем в корень;

2) если элемент меньше текущего и у текущего элемента нет левого потомка, делаем элемент этим потомком. Если левый потомок есть, то переходим к нему и повторяем пункты 2 и 3;

3) если элемент больше текущего и у текущего элемента нет правого потомка, делаем элемент этим потомком. Если правый потомок есть, то переходим к нему и повторяем пункты 2 и 3.

2. Добавление узла в бинарное дерево поиска. Пользователь вводит число, необходимо добавить его в дерево, руководствуясь принципами построения бинарного дерева поиска.

3. Вывод бинарного дерева. Здесь необходимо использовать алгоритм обхода дерева. Принцип обходов дерева был рассмотрен на лекции (см. презентацию к лекции).

4. Поиск образца в бинарном дереве поиска. Необходимо использовать знания о принципе построения бинарного дерева поиска. Фактически алгоритм поиска будет аналогичен алгоритму бинарного поиска для отсортированного в порядке возрастания массива. Другими словами, используется алгоритм построения дерева, только без добавления нового элемента.

Сначала аргумент поиска сравнивается с ключом, находящимся в корне. Если аргумент совпадает с ключом, поиск закончен, если же не совпадает, то в случае, когда аргумент оказывается меньше ключа, поиск продолжается в левом поддереве, а в случае, когда больше ключа, - в правом поддереве. Увеличив уровень на 1, повторяют сравнение, считая текущий узел корнем. Если дальше идти некуда, аргумента поиска в дереве нет.

5. Подсчет количества узлов дерева. Используя любой из рекурсивных обходов, сосчитайте количество узлов дерева.

6. Подсчет числа листьев дерева: аналогично заданию 5.

7. Расчет степени вершины. Для введенной вершины найти ее степень, используя любой из рекурсивных обходов.

8. Расчет уровня вершины: аналогично заданию 7.

9. Расчет высоты дерева. Высотой дерева будет являться мак- симальный из уровней вершин.

10. Поиск образца в бинарном дереве. Принцип построения дерева не должен учитываться, т.е. этот вид поиска должен работать для любого дерева. Для его реализации необходимо обойти все вершины дерева.

11. Удаление узла дерева с поддеревом. Необходимо удалить указанный узел дерева со всеми вершинами его поддерева.

12. Удаление всего дерева. Реализовать с использованием задания 11.

13. Удаление узла с перестройкой дерева. Необходимо удалить указанный узел, при этом все его потомки должны остаться в дереве. На первый взгляд кажется очевидным, что достаточно просто добавить все эти потомки в дерево заново, но такой подход является крайне неоптимальным. Для удаления узла с перестройкой из бинарного дерева поиска наиболее эффективен следующий алгоритм:

- если у удаляемого узла нет потомков, ничего не делаем;
- если у удаляемого узла есть только одна ветвь потомков, первый узел этой ветви становится на место удаляемого;

- если у удаляемого узла имеются обе ветви потомков, то на его место должен встать последний левый потомок правого поддерева (или последний правый потомок левого поддерева).

14. Обход дерева в глубину (не рекурсивный). Для обхода дерева в глубину используется стек. Алгоритм:

- • добавляем в стек корень дерева;
  • пока стек не пуст, выполняем: о удаляем из стека элемент;
  о добавляем в стек потомков этого элемента.

15. Обход дерева в ширину (не рекурсивный). Решается аналогично заданию 14, только вместо стека используется очередь. Обход дерева в ширину позволяет распечатать дерево по уровням, что может пригодиться для организации «красивого» вывода дерева.

Выполнение задания по стеку:

Листинг программы:

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

namespace LB_4
{
   class Program
   {
      static void Main(string[] args)
      {
         var tree = new Tree<int>();
         tree.Add(15);
         tree.Add(10);
         tree.Add(4);
         tree.Add(1);
         tree.Add(7);
         tree.Add(5);
         tree.Add(9);
         tree.Add(13);
         tree.Add(11);
         tree.Add(12);
         tree.Add(14);
         tree.Add(25);
         tree.Add(19);
         tree.Add(17);
         tree.Add(16);
         tree.Add(21);

         Console.Write("Префиксный обход:   ");
         foreach (var item in tree.Preorder())
         {
```

```
            Console.Write(item + ", ");
        }
        Console.Write("\nПостфиксный обход:  ");
        foreach (var item in tree.Postorder())
        {
            Console.Write(item + ", ");
        }

        Console.Write("\nИнфиксный обход:    ");
        foreach (var item in tree.Inorder())
        {
            Console.Write(item + ", ");
        }

        Console.WriteLine("\n");
        Console.WriteLine("Подсчет количества узлов дерева: " + tree.CountNodes);
        Console.WriteLine("Подсчет числа листьев дерева: " + tree.NumberLeaves());
        Console.WriteLine("Расчет степени вершины(25): " + tree.NodeDegree(25));
        Console.WriteLine("Расчет уровня вершины(7): " + tree.NodeLevel(7));
        Console.WriteLine("Расчет высоты дерева: " + tree.TreeHeight());


        Console.Write("\nОбход с помощью стека:  ");
        foreach (var item in tree.StackCollect())
        {
            Console.Write(item + ", ");
        }

        Console.Write("\n\nУдаление узла дерева с поддеревом(19):   ");
        tree.RemoveWithSubtree(19);
        foreach (var item in tree.Preorder())
        {
            Console.Write(item + ", ");
        }

        Console.Write("\nУдаление узла с перестройкой дерева(10):   ");
        tree.Remove(10);
        foreach (var item in tree.Preorder())
        {
            Console.Write(item + ", ");
        }

        Console.ReadLine();
    }
}


class Node<T>:IComparable where T: IComparable
{
    public T Data { get; set; }
    public Node<T> Left { get; set; }
    public Node<T> Right { get; set; }
    public Node<T> Parent { get; set; }

    public Node(T data)
    {
        Data = data;
    }

    public Node(T data, Node<T> parent)
    {
        Data = data;
        Parent = parent;
    }

    public int CompareTo(object obj)
```

```
    {
        if(obj is Node<T> item)
        {
            return Data.CompareTo(item);
        }
        else
        {
            throw new ArgumentException("Не совпадение типов");
        }
    }


    public void Add(T data)
    {
        var node = new Node<T>(data);

        if (node.Data.CompareTo(Data) == -1)
        {
            if (Left == null)
            {
                Left = new Node<T>(data, this);
            }
            else
            {
                Left.Add(data);
            }
        }
        else
        {
            if (Right == null)
            {
                Right = new Node<T>(data, this);
            }
            else
            {
                Right.Add(data);
            }
        }
    }

    public override string ToString()
    {
        return Data.ToString();
    }
}



class Tree<T> where T : IComparable
{
    public Node<T> Root { get; private set; }

    //Подсчет количества узлов дерева
    public int CountNodes { get; private set; }


    public void Add(T data)
    {
        if (Root == null)
        {
            Root = new Node<T>(data);
            CountNodes = 1;
            return;
        }

        Root.Add(data);
```

```
        CountNodes++;
    }


    //префиксный обход - элемент, левое, правое
    public List<T> Preorder()
    {
        if (Root == null)
        {
            return new List<T>();
        }

        return Preorder(Root);
    }
    private List<T> Preorder(Node<T> node)
    {
        var list = new List<T>();

        if (node != null)
        {
            list.Add(node.Data);

            if (node.Left != null)
            {
                list.AddRange(Preorder(node.Left));
            }

            if (node.Right != null)
            {
                list.AddRange(Preorder(node.Right));
            }
        }

        return list;
    }


    //постфиксный обход - левое, правое, элемент
    public List<T> Postorder()
    {
        if (Root == null)
        {
            return new List<T>();
        }

        return Postorder(Root);
    }
    private List<T> Postorder(Node<T> node)
    {
        var list = new List<T>();

        if (node != null)
        {
            if (node.Left != null)
            {
                list.AddRange(Postorder(node.Left));
            }

            if (node.Right != null)
            {
                list.AddRange(Postorder(node.Right));
            }

            list.Add(node.Data);
        }
```

```
        return list;
    }


    //инфиксный обход - левое, элемент, правое
    public List<T> Inorder()
    {
        if (Root == null)
        {
            return new List<T>();
        }

        return Inorder(Root);
    }
    private List<T> Inorder(Node<T> node)
    {
        var list = new List<T>();

        if (node != null)
        {
            if (node.Left != null)
            {
                list.AddRange(Inorder(node.Left));
            }

            list.Add(node.Data);

            if (node.Right != null)
            {
                list.AddRange(Inorder(node.Right));
            }
        }

        return list;
    }


    //поиск
    public Node<T> Search(T data)
    {
        return Search(Root, data);
    }
    private Node<T> Search(Node<T> node, T data)
    {
        if (node == null) return null;

        switch (data.CompareTo(node.Data))
        {
            case 1: return Search(node.Right, data);
            case -1: return Search(node.Left, data);
            case 0: return node;
            default: return null;
        }
    }



    //Подсчет числа листьев дерева
    private int CountLeaves;
    public int NumberLeaves()
    {
        CountLeaves = 0;

        if (Root == null)
        {
            return CountLeaves;
```

```
    }

    NumberLeaves(Root);
    return CountLeaves;
}
private void NumberLeaves(Node<T> node)
{
    if (node != null)
    {
        if (node.Left == null && node.Right == null)
        {
            CountLeaves++;
        }

        if (node.Left != null)
        {
            NumberLeaves(node.Left);
        }

        if (node.Right != null)
        {
            NumberLeaves(node.Right);
        }
    }
}


//Расчет степени вершины - количество потомков
private int Degree;
public int NodeDegree(T data)
{
    Degree = -1;

    //Проверяем, существует ли данный узел
    Node<T> node = Search(data);
    if (node == null)
    {
        return 0;
    }

    NodeDegree(node);
    return Degree;
}
private void NodeDegree(Node<T> node)
{
    if (node != null)
    {
        if (node.Left != null)
        {
            NodeDegree(node.Left);
        }

        if (node.Right != null)
        {
            NodeDegree(node.Right);
        }

        Degree++;
    }
}


//Расчет уровня вершины
private int Level;
public int NodeLevel(T data)
{
```

```
    Level = -1;

    NodeLevel(Root, data);
    return Level;
}
private Node<T> NodeLevel(Node<T> node, T data)
{
    if (node == null) return null;

    Level++;

    switch (data.CompareTo(node.Data))
    {
        case 1: return NodeLevel(node.Right, data);
        case -1: return NodeLevel(node.Left, data);
        case 0: return node;
        default: return null;
    }
}


//Расчет высоты дерева
private int Height;
public int TreeHeight()
{
    Height = 0;

    if (Root == null)
    {
        return Height;
    }

    TreeHeight(Root);
    return Height;
}
private void TreeHeight(Node<T> node)
{
    if (node != null)
    {
        if(NodeLevel(node.Data) > Height)
        {
            Height = NodeLevel(node.Data);
        }

        if (node.Left != null)
        {
            TreeHeight(node.Left);
        }

        if (node.Right != null)
        {
            TreeHeight(node.Right);
        }
    }
}


//поиск для всех деревьев
private Node<T> SearchNode;
public Node<T> SearchForAll(T data)
{
    if (Root == null)
    {
        return null;
    }
```

```
    SearchForAll(Root, data);
    return SearchNode;
}
private void SearchForAll(Node<T> node, T data)
{
    if (node != null)
    {
        if (data.CompareTo(node.Data) == 0)
        {
            SearchNode = node;
        }
        else
        {
            if (node.Left != null)
            {
                SearchForAll(node.Left, data);
            }

            if (node.Right != null)
            {
                SearchForAll(node.Right, data);
            }
        }
    }
}


//Удаление узла дерева с поддеревом
//Удаление всего дерева
public bool RemoveWithSubtree(T data)
{
    //Проверяем, существует ли данный узел
    Node<T> node = Search(data);
    if (node == null)
    {
        return false;
    }

    //Если удаляем корень
    if (node == Root)
    {
        Root = null;

        return true;
    }
    //Если не корень
    else
    {
        if (node == node.Parent.Left)
            node.Parent.Left = null;
        else
        {
            node.Parent.Right = null;
        }
        return true;
    }
}



//Удаление узла с перестройкой дерева
public bool Remove(T data)
{
    //Проверяем, существует ли данный узел
    Node<T> node = Search(data);
    if (node == null)
```

```
{
    return false;
}

//Удаление листьев
if (node.Left == null && node.Right == null)
{
    if (node == node.Parent.Left)
        node.Parent.Left = null;
    else
    {
        node.Parent.Right = null;
    }
    return true;
}

//Если удаляем корень
if (node == Root)
{
    Node<T> curTree;
    if (node.Right != null)
    {
        curTree = node.Right;
    }
    else curTree = node.Left;

    while (curTree.Left != null)
    {
        curTree = curTree.Left;
    }

    T temp = curTree.Data;
    this.Remove(temp);
    node.Data = temp;

    return true;
}

//Удаление узла, имеющего левое поддерево, но не имеющее правого поддерева
if (node.Left != null && node.Right == null)
{
    //Меняем родителя
    node.Left.Parent = node.Parent;
    if (node == node.Parent.Left)
    {
        node.Parent.Left = node.Left;
    }
    else if (node == node.Parent.Right)
    {
        node.Parent.Right = node.Left;
    }
    return true;
}

//Удаление узла, имеющего правое поддерево, но не имеющее левого поддерева
if (node.Left == null && node.Right != null)
{
    //Меняем родителя
    node.Right.Parent = node.Parent;
    if (node == node.Parent.Left)
    {
        node.Parent.Left = node.Right;
    }
    else if (node == node.Parent.Right)
    {
        node.Parent.Right = node.Right;
```

```
            }
            return true;
        }

        //Удаляем узел, имеющий поддеревья с обеих сторон
        if (node.Right != null && node.Left != null)
        {
            var curNode = node.Right;

            while (curNode.Left != null)
            {
                curNode = curNode.Left;
            }

            //Если самый левый элемент является первым потомком
            if (curNode.Parent == node)
            {
                curNode.Left = node.Left;
                node.Left.Parent = curNode;

                curNode.Parent = node.Parent;
                if (node == node.Parent.Left)
                {
                    node.Parent.Left = curNode;
                }
                else if (node == node.Parent.Right)
                {
                    node.Parent.Right = curNode;
                }
                return true;
            }
            //Если самый левый элемент НЕ является первым потомком
            else
            {
                if (curNode.Right != null)
                {
                    curNode.Right.Parent = curNode.Parent;
                }
                curNode.Parent.Left = curNode.Right;

                curNode.Right = node.Right;
                curNode.Left = node.Left;
                node.Left.Parent = curNode;
                node.Right.Parent = curNode;

                curNode.Parent = node.Parent;
                if (node == node.Parent.Left)
                {
                    node.Parent.Left = curNode;
                }
                else if (node == node.Parent.Right)
                {
                    node.Parent.Right = curNode;
                }
                return true;
            }
        }
        return false;
    }



    //Обход дерева в глубину используется стек
    public List<T> StackCollect()
    {
        var nodeStack = new Stack<Node<T>>();
```

```
            var list = new List<Node<T>>();
            var output = new List<T>();
            var node = Root;

            nodeStack.Push(node);
            output.Add(node.Data);
            list.Add(node);

            while (true)
            {
                if (node.Left != null && list.Contains(node.Left) == false)
                {
                    nodeStack.Push(node.Left);
                    output.Add(node.Left.Data);
                    list.Add(node.Left);
                    node = node.Left;
                    continue;
                }
                if (node.Right != null && list.Contains(node.Right) == false)
                {
                    nodeStack.Push(node.Right);
                    output.Add(node.Right.Data);
                    list.Add(node.Right);
                    node = node.Right;
                    continue;
                }
                if (nodeStack.Count == 0)
                {
                    break;
                }
                node = nodeStack.Pop();
            }

            return output;
        }
    }
}
```

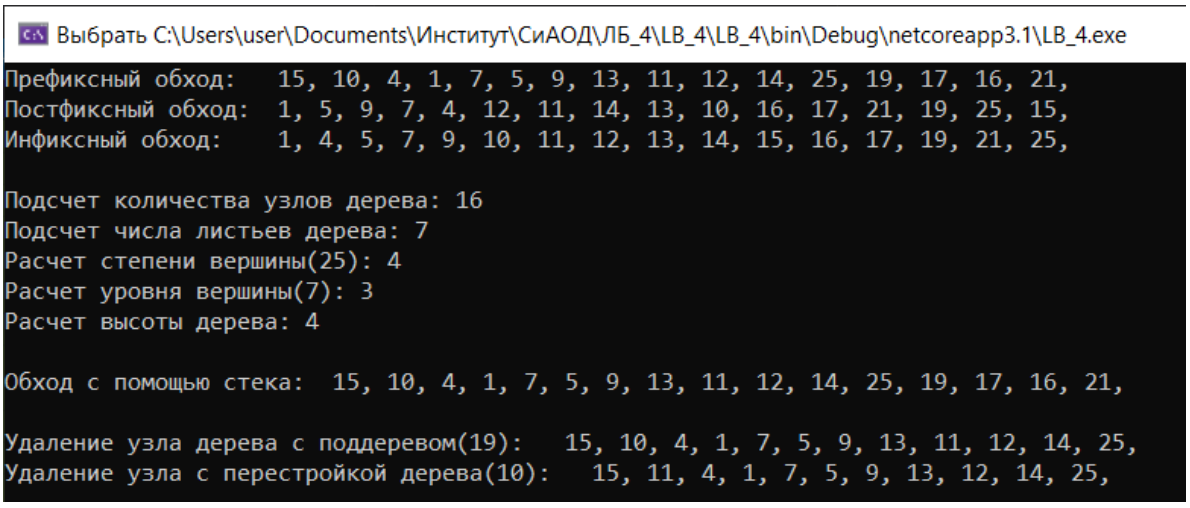

Рисунок 1 – скриншот работы программы

Вывод: в ходе данной лабораторной работы были приобретены навыки по работе с бинарными деревьями поиска.