

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра ВТ

ОТЧЕТ
по лабораторной работе №1
по дисциплине «Программирование в среде .NET»
Тема: РЕАЛИЗАЦИЯ БАЗОВЫХ АЛГОРИТМОВ СРЕДСТВАМИ
ЯЗЫКА C#

Студент гр. 6306

Милакин А. Д.

Преподаватель

Пешехонов К. А.

Санкт-Петербург
2020
Цель работы

Исследовать вопрос реализации базовых алгоритмов средствами языка C#.

Задание

1) Реализовать связный список: создание, удаление, добавление произвольных элементов, реверс списка - без использования стандартных коллекций/LINQ (только IEnumerable).

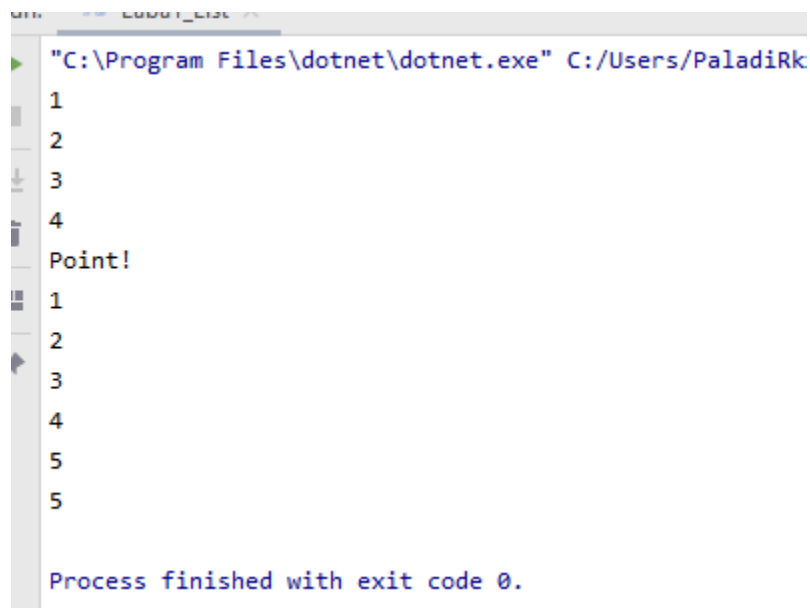
2) Реализовать бинарное дерево: заполнение, поиск, удаление элемента - без использования стандартных деревьев.

Практическая часть

Обращаю внимание на то, что связный список и бинарное дерево выполнены с использованием дженериков (обобщений), в результате чего данные не обязательно должны быть целыми числами.

Связный список

В качестве демонстрации работы связного списка сначала создадим список из пяти элементов, забираем последний элемент и вставим его дважды в конце списка.



```
"C:\Program Files\dotnet\dotnet.exe" C:/Users/PaladiRk
1
2
3
4
Point!
1
2
3
4
5
5
Process finished with exit code 0.
```

Рисунок 1. Связный список.

Бинарное дерево

В рамках демонстрации работы бинарного дерева сначала создадим его, вставим в него 6 элементов, выведем по убыванию, так же по возрастанию. Найдём элемент со значением 2. Удалим элемент с датой 3 и выведем дерево.

```
Print begin tree:  
6 5 2 1 0  
Print begin tree:  
0 1 2 5 6  
Search node with data 2 : 2  
Print begin tree after delete elem with data 3:  
0 1 2 5 6  
  
Process finished with exit code 0.
```

Рисунок 2. Бинарное дерево.

Код программ

Связный список

Node.cs

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

namespace Lab1_List
{
    class Node<T>
    {
        private readonly T _data;
        public Node<T> Next;

        public Node(T data)
        {
            _data = data;
            this.Next = null;
        }

        public T GetData()
        {
            return _data;
        }

        public override bool Equals(object val)
        {
            if (val == null) return this.Equals(null);
            if (!(val is Node<T> m))
                return false;
            return Equals(m.GetData(), this.GetData());
        }

        private bool Equals(Node<T> other)
        {
            return EqualityComparer<T>.Default.Equals(_data, other._data);
        }

        public override int GetHashCode()
        {
            return GetHashCode.Combine(_data);
        }
    }
}
```

List.cs

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

namespace Lab1_List
{
    class List<T> : IEnumerable<T>
    {
        private readonly Node<T> _root;
        private Node<T> _end;
        private int _countNode;
        private IEnumerable<T> _enumerableImplementation;

        public List(Node<T> root)
        {
            _root = root;
            _end = root;
            _countNode = 1;
        }

        public int GetCountNode()
        {
            return _countNode;
        }

        public Node<T> GetRoot()
        {
            return _root;
        }

        IEnumerator<T> IEnumerable<T>.GetEnumerator()
        {
            return _enumerableImplementation.GetEnumerator();
        }

        public IEnumerator GetEnumerator()
        {
            var val = _root;
            while (val != null)
            {
                yield return val;
                val = val.Next;
            }
        }

        public T Pop()
        {
            var node = _end;

            if (ReferenceEquals(_end, _root))
            {
                _end = _root;
                _root.Next = null;
                return node.GetData();
            }
        }
    }
}
```

```

Node<T> prev = _root;
while (prev.Next.Next != null)
{
    prev = prev.Next;
}

prev.Next = null;
--_countNode;
_end = prev;
return node.GetData();
}

public void Push(Node<T> node)
{
    if (node != null)
    {
        _end.Next = node;
        node.Next = null;
        _end = node;
        ++_countNode;
    }
}

public void Add(T item)
{
    if (item == null)
    {
        return;
    }
    var node = new Node<T>(item);
    _end.Next = node;
    node.Next = null;
    _end = node;
    ++_countNode;
}

public bool Contains(T item)
{
    if (item == null)
        return false;
    return this.Cast<Node<T>>().Any(node => Equals(node.GetData(), item));
}

public bool Contains(Node<T> node)
{
    if (node == null)
        return false;
    return this.Cast<Node<T>>().Contains(node);
}
}
}

```

Program.cs

```
using System;

namespace Laba1_List
{
    internal static class Program
    {
        public static void Main(string[] args)
        {
            var list = new List<int>(new Node<int>(1)) { 2, 3, 4, 5 };
            var lastNode = list.Pop();
            foreach (Node<int> n in list)
            {
                Console.WriteLine(n.GetData());
            }
            Console.WriteLine("Point!");
            list.Push(new Node<int>(lastNode));
            list.Push(new Node<int>(lastNode));
            foreach (Node<int> n in list)
            {
                Console.WriteLine(n.GetData());
            }
        }
    }
}
```

Бинарное дерево

Node.cs

```
using System;

namespace Laba1_Tree
{
    class Node<T> where T : IComparable
    {
        public T Data { get; set; }
        public Node<T> Left { get; set; }
        public Node<T> Right { get; set; }
        public Node(T data)
        {
            Data = data;
        }
        public Node(T data, Node<T> left, Node<T> right)
        {
            Data = data;
            Left = left;
            Right = right;
        }
    }
}
```

BinaryTree.cs

```
using System;
using System.Collections.ObjectModel;

namespace Laba1_Tree
{
    class BinaryTree<T>: Collection<T>, IEquatable<T> where T : IComparable
    {
        public Node<T> Root { get; private set; }

        public BinaryTree()
        {
            Root = null;
        }
        public BinaryTree(Node<T> root)
        {
            Root = root;
        }
        public override int GetHashCode()
        {
            return (Root != null ? Root.GetHashCode() : 0);
        }

        public new void Add(T elem)
```



```

{
    Node<T> previous;
    previous = null;
    Node<T> current = Root;

    while (current != null)
    {
        previous = current;
        if (current.Data.CompareTo(elem) > 0)
        {
            current = current.Left;
        }
        else if (current.Data.CompareTo(elem) < 0)
        {
            current = current.Right;
        }
        else return;
    }

    Node<T> node = new Node<T>(elem);
    if (previous == null)
    {
        Root = node;
    }
    else
    {
        if (previous.Data.CompareTo(elem) > 0)
        {
            previous.Left = node;
        }
        else
            previous.Right = node;
    }
}

public void Delete(T data)
{
    Delete(Root, data);
}

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

```

```

    if (node.Data.CompareTo(data) > 0)
    {
        node.Left = Delete(node.Left, data);
    }
    else if (node.Data.CompareTo(data) < 0)
    {
        node.Right = Delete(node.Right, data);
    }
    else
    {
        if (node.Left == null)
        {
            return node.Right;
        }
        else if (node.Right == null)
        {
            return node.Left;
        }

        T tempData = node.Right.Data;
        while (node.Right.Left != null)
        {
            tempData = node.Right.Left.Data;
            node.Right = node.Right.Left;
        }

        node.Data = tempData;
        node.Right = Delete(node.Right, node.Data);
    }

    return node;
}

public Node<T> Search(T data)
{
    return Search(Root, data);
}

private Node<T> Search(Node<T> node, T data)
{
    if (node == null || node.Data.CompareTo(data) == 0)
    {
        return node;
    }

```

```

        if (node.Data.CompareTo(data) > 0)
        {
            return Search(node.Left, data);
        }
        return Search(node.Right, data);
    }

    public void PrintLowToHigh()
    {
        PrintLowToHigh(Root);
        Console.WriteLine();
    }

    private void PrintLowToHigh(Node<T> node)
    {
        if (node != null)
        {
            PrintLowToHigh(node.Left);
            Console.Write(node.Data);
            Console.Write(" ");
            PrintLowToHigh(node.Right);
        }
    }

    public void PrintHighToLow()
    {
        PrintHighToLow(Root);
        Console.WriteLine();
    }

    private void PrintHighToLow(Node<T> node)
    {
        if (node != null)
        {
            PrintHighToLow(node.Right);
            Console.Write(node.Data);
            Console.Write(" ");
            PrintHighToLow(node.Left);
        }
    }

    protected bool Equals(BinaryTree<T> other)
    {
        return Equals(Root, other.Root);
    }

```

```

    }

    public bool Equals(T other)
    {
        return Search(Root, other) != null;
    }

    public override bool Equals(object obj)
    {
        if (ReferenceEquals(null, obj)) return false;
        if (ReferenceEquals(this, obj)) return true;
        if (obj.GetType() != this.GetType()) return false;
        return Equals((BinaryTree<T>) obj);
    }
}
}

```

Program.cs

```

using System;

namespace Laba1_Tree
{
    class Program
    {
        static void Main(string[] args)
        {
            var numbers = new BinaryTree<int>();
            numbers.Add(0);
            numbers.Add(1);
            numbers.Add(2);
            numbers.Add(5);
            numbers.Add(6);

            Console.WriteLine("Print begin tree:");
            numbers.PrintHighToLow();
            Console.WriteLine("Print begin tree:");
            numbers.PrintLowToHigh();

            Node<int> node = numbers.Search(2);

            Console.WriteLine("Search node with data 2 : {0:D}", node.Data);
            numbers.Delete(3);
            Console.WriteLine("Print begin tree after delete elem with data 3:");
            numbers.PrintLowToHigh();
        }
    }
}

```

Выводы по работе

В процессе выполнения лабораторной работы мы исследовали вопрос реализации базовых алгоритмов средствами языка C#. Изучив основы незнакомой до этого среды разработки Rider, а также некоторые особенности среды .Net.