**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#**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 6306 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Бохан Ю.А. |
|  |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Пешехонов К. А. |

Санкт-Петербург

2020

**Цель работы:**

Реализовать несколько базовых алгоритмов в среде .NET с помощью языка C#.

**Задание:**

1) Реализовать связный список: создание, удаление, добавление про-извольных элементов, реверс списка - без использования стандартных кол-лекций/LINQ (только IEnumerable);

2) Реализовать бинарное дерево: заполнение, поиск, удаление эле-мента - без использования стандартных деревьев;

3) Реализовать сортировку вставками - без .OrderBy().

**Код программы :**

Класс DoubleLinkedList:

using System;  
using System.Collections;  
using System.Collections.Generic;  
  
namespace Laba1  
{  
 public class DoubleLinkedList<T>: IEnumerable<T> where T : IComparable, new()  
 {  
 DoubleNode<T> head; // головной/первый элемент  
 DoubleNode<T> tail; // последний/хвостовой элемент  
 int count; // количество элементов в списке  
   
 // добавление элемента  
 public void Add(T data)  
 {  
 DoubleNode<T> node = new DoubleNode<T>(data);  
   
 if (head == null)  
 head = node;  
 else  
 {  
 tail.Next = node;  
 node.Previous = tail;  
 }  
 tail = node;  
 count++;  
 }  
 public void AddFirst(T data)  
 {  
 DoubleNode<T> node = new DoubleNode<T>(data);  
 DoubleNode<T> temp = head;  
 node.Next = temp;  
 head = node;  
 if (count == 0)  
 tail = head;  
 else  
 temp.Previous = node;  
 count++;  
 }  
 // удаление  
 public bool Remove(T data)  
 {  
 DoubleNode<T> current = head;  
   
 // поиск удаляемого узла  
 while (current != null)  
 {  
 if (current.data.Equals(data))  
 {  
 break;  
 }  
 current = current.Next;  
 }  
 if(current!=null)  
 {  
 // если узел не последний  
 if(current.Next!=null)  
 {  
 current.Next.Previous = current.Previous;  
 }  
 else  
 {  
 // если последний, переустанавливаем tail  
 tail = current.Previous;  
 }  
   
 // если узел не первый  
 if(current.Previous!=null)  
 {  
 current.Previous.Next = current.Next;  
 }  
 else  
 {  
 // если первый, переустанавливаем head  
 head = current.Next;  
 }  
 count--;  
 return true;  
 }  
 return false;  
 }  
   
 public void Output()  
 {  
 if(IsEmpty)   
 Console.Write("Empty");  
 DoubleNode<T> node = head;  
 while(node != null)  
 {  
 Console.Write("{0} ", node.data);  
 node = node.Next;  
 }  
 Console.Write('\n');  
 }  
   
   
 public void Sort()  
 {  
 DoubleNode<T> newBegin = null;  
 DoubleNode<T> newEnd = head; //null  
 while (head != null)  
 {  
 DoubleNode<T> node = head;  
 if (head.CompareTo(newEnd) > 0)  
 newEnd = head;  
 head = head.Next;  
 if (newBegin == null || node.CompareTo(newBegin) < 0)  
 {  
 node.Next = newBegin;  
 newBegin = node;  
 }  
 else  
 {  
 DoubleNode<T> current= newBegin;  
 while (current.Next != null && node.CompareTo(current.Next) >= 0)  
 {  
 current = current.Next;  
 }  
 node.Next = current.Next;  
 current.Next = node;  
 }  
 }  
 head = newBegin;  
 tail = newEnd;  
 }  
   
   
 public int Count { get { return count; } }  
 public bool IsEmpty { get { return count == 0; } }  
   
 public void Clear()  
 {  
 head = null;  
 tail = null;  
 count = 0;  
 }  
   
 public bool Contains(T data)  
 {  
 DoubleNode<T> current = head;  
 while (current != null)  
 {  
 if (current.data.Equals(data))  
 return true;  
 current = current.Next;  
 }  
 return false;  
 }  
   
 IEnumerator IEnumerable.GetEnumerator()  
 {  
 return ((IEnumerable)this).GetEnumerator();  
 }  
   
 IEnumerator<T> IEnumerable<T>.GetEnumerator()  
 {  
 DoubleNode<T> current = head;  
 while (current != null)  
 {  
 yield return current.data;  
 current = current.Next;  
 }  
 }  
   
 public IEnumerable<T> BackEnumerator()  
 {  
 DoubleNode<T> current = tail;  
 while (current != null)  
 {  
 yield return current.data;  
 current = current.Previous;  
 }  
 }  
 }  
}

Класс Tree:

using System;  
  
namespace Laba1  
{  
 public class Tree<T> where T : IComparable  
 {  
 public TreeNode<T> RootNode { get; set; }  
   
 public TreeNode<T> Add(TreeNode<T> node, TreeNode<T> currentNode = null)  
 {  
 if (RootNode == null)  
 {  
 node.ParentNode = null;  
 return RootNode = node;  
 }  
 currentNode ??= RootNode;  
 node.ParentNode = currentNode;  
 int result;  
 result = node.Data.CompareTo(currentNode.Data);  
 if (result == 0)  
 return currentNode;  
 if (result < 0)  
 return currentNode.LeftNode == null ? (currentNode.LeftNode = node) : Add(node, currentNode.LeftNode);  
 return currentNode.RightNode == null ? (currentNode.RightNode = node) : Add(node, currentNode.RightNode);  
 }  
   
 public TreeNode<T> Add(T data)  
 {  
 return Add(new TreeNode<T>(data));  
 }  
   
 public TreeNode<T> FindNode(T data, TreeNode<T> startWithNode = null)  
 {  
 startWithNode ??= RootNode;  
 int result;  
 result = data.CompareTo(startWithNode.Data);  
 if (result == 0)  
 return startWithNode;  
 if (result < 0)  
 return startWithNode.LeftNode == null ? null : FindNode(data, startWithNode.LeftNode);  
 return startWithNode.RightNode == null ? null : FindNode(data, startWithNode.RightNode);  
 }  
   
 public void Remove(TreeNode<T> node)  
 {   
 if (node == null) return;  
 var currentNodeSide = node.NodeSide;   
 //если у узла нет подузлов, можно его удалить  
 if (node.LeftNode == null && node.RightNode == null)  
 {  
 if (currentNodeSide == Side.Left)  
 node.ParentNode.LeftNode = null;  
 else  
 node.ParentNode.RightNode = null;  
 }  
 //если нет левого, то правый ставим на место удаляемого   
 else   
 if (node.LeftNode == null)  
 {  
 if (currentNodeSide == Side.Left)  
 node.ParentNode.LeftNode = node.RightNode;  
 else  
 node.ParentNode.RightNode = node.RightNode;  
 node.RightNode.ParentNode = node.ParentNode;  
 }  
 //если нет правого, то левый ставим на место удаляемого   
 else   
 if (node.RightNode == null)  
 {  
 if (currentNodeSide == Side.Left)  
 node.ParentNode.LeftNode = node.LeftNode;  
 else  
 node.ParentNode.RightNode = node.LeftNode;  
 node.LeftNode.ParentNode = node.ParentNode;  
 }  
 //если оба дочерних присутствуют, то правый становится на место удаляемого,а левый вставляется в правый  
 else  
 {  
 switch (currentNodeSide)  
 {  
 case Side.Left:  
 node.ParentNode.LeftNode = node.RightNode;  
 node.RightNode.ParentNode = node.ParentNode;  
 Add(node.LeftNode, node.RightNode);  
 break;  
 case Side.Right:  
 node.ParentNode.RightNode = node.RightNode;  
 node.RightNode.ParentNode = node.ParentNode;  
 Add(node.LeftNode, node.RightNode);  
 break;  
 default:  
 var bufLeft = node.LeftNode;  
 var bufRightLeft = node.RightNode.LeftNode;  
 var bufRightRight = node.RightNode.RightNode;  
 node.Data = node.RightNode.Data;  
 node.RightNode = bufRightRight;  
 node.LeftNode = bufRightLeft;  
 Add(bufLeft, node);  
 break;  
 }  
 }  
 }  
   
 public void Remove(T data)  
 {  
 var foundNode = FindNode(data);  
 Remove(foundNode);  
 }  
  
 private void PrintTree(TreeNode<T> startNode, string indent = "", Side? side = null)  
 {  
 if (startNode != null)  
 {  
 var nodeSide = side == null ? "+" : side == Side.Left ? "L" : "R";  
 Console.WriteLine($"{indent} [{nodeSide}]- {startNode.Data}");  
 indent += new string(' ', 3);  
 //рекурсивный вызов для левой и правой веток  
 PrintTree(startNode.LeftNode, indent, Side.Left);  
 PrintTree(startNode.RightNode, indent, Side.Right);  
 }  
 }  
   
 public void PrintTree()  
 {  
 PrintTree(RootNode);  
 }  
   
 }  
}

**Вывод**

В процессе выполнения лабораторной работы были реализованы несколько стандартных структур данных средствами языка C# (связный список, бинарное дерево). Были получены навыки работы в среде .NET на языке C#.