**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра вычислительной техники**

**ОТЧЁТ**

**по лабораторной работе №1**

**по дисциплине «Программирование в среде dotNET»**

**Тема: “Реализация базовых алгоритмов средствами языка C#”**

|  |  |  |
| --- | --- | --- |
| Студент гр. 6305 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Силинский А.С. |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Пешехонов К.А. |

Санкт-Петербург

2020

**Оглавление**

[**Цель работы** 3](#_Toc38248990)

[**Задание** 3](#_Toc38248991)

[**Ход работы** 3](#_Toc38248992)

[**Выводы** 5](#_Toc38248993)

[**Приложение** 6](#_Toc38248994)

# **Цель работы**

Целью данной лабораторной работы является реализация 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
3. Реализовать сортировку вставками, слиянием (без использования OrderBy())

# **Ход работы**

В ходе выполнения работы был реализован связный список. Проверим работоспособность этого связного списка:

1. Попробуем добавить в начало списка 2 элемента, а затем добавить еще один в конец:

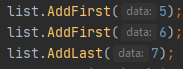


Рис. 1. Входные данные

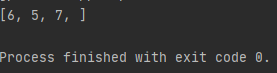


Рис. 2. Выходные данные

1. Попробуем добавить в список произвольные элементы, а затем развернем его:

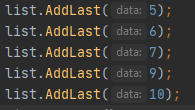


Рис. 3. Входные данные



Рис. 4. Выходные данные

1. Попробуем удалить и добавить элементы с произвольным индексом:

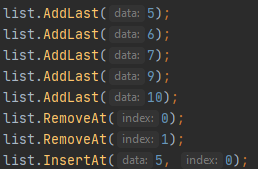


Рис. 5. Входные данные



Рис. 6. Выходные данные

Также было реализовано бинарное дерево, проверим работоспособность этого бинарного дерева:

1. Попробуем добавить в дерево элементы, удалить их и проверить на их наличие в дереве и вывести все элементы этого дерева:

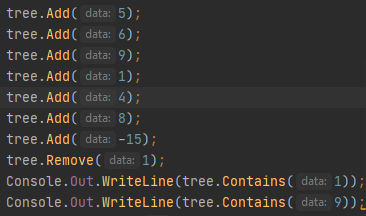


Рис. 7. Входные данные

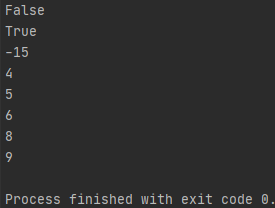


Рис. 8. Выходные данные

Также были реализованы алгоритмы сортировки слиянием и вставками, посмотрим на их работоспособность:

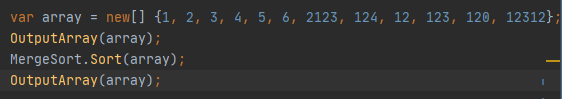


Рис. 9. Входные данные (сортировка слиянием)

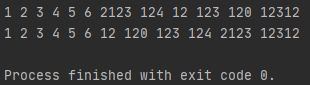


Рис. 10. Выходные данные (сортировка слиянием)

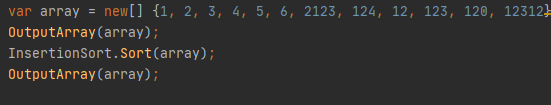


Рис. 11. Входные данные (сортировка вставками)

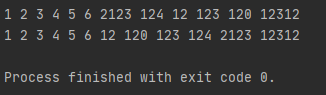


Рис. 12. Выходные данные (сортировка вставками)

Код структур данных и алгоритмов можно найти в приложении к данной лабораторной работе.

# **Выводы**

В ходе выполнения данной лабораторной работы были реализованы следующие структуры и алгоритмы: связный список, бинарное дерево, алгоритм сортировки вставками и слиянием.

# **Приложение**

***LinkedList.cs***

using System;

using System.Collections;

using System.Collections.Generic;

using System.Text;

using DataStructures.structures.enumerators;

using DataStructures.structures.models;

namespace DataStructures.structures

{

public class LinkedList<T> : common.IList<T>

{

private ListNode<T> \_head;

public LinkedList()

{

Count = 0;

\_head = null;

}

public int Count { get; private set; }

public IEnumerator<T> GetEnumerator()

{

return new LinkedListEnumerator<T>(\_head);

}

IEnumerator IEnumerable.GetEnumerator()

{

return GetEnumerator();

}

public T ValueAt(int index)

{

var i = 0;

foreach (var elem in this)

{

if (index == i)

return elem;

i++;

}

throw new IndexOutOfRangeException();

}

public void InsertAt(T data, int index)

{

if (index == 0)

{

\_head = new ListNode<T>(data, \_head);

}

else if (index <= Count)

{

var currentNode = \_head;

for (var i = 0; currentNode != null; i++)

{

if (i == index - 1) currentNode.Next = new ListNode<T>(data, currentNode.Next);

currentNode = currentNode.Next;

}

}

else

{

throw new IndexOutOfRangeException();

}

Count++;

}

public void RemoveAt(int index)

{

if (index == 0)

{

\_head = \_head.Next;

}

else if (index <= Count)

{

var currentNode = \_head;

for (var i = 0; currentNode != null; i++)

{

if (i == index - 1) currentNode.Next = currentNode.Next.Next;

currentNode = currentNode.Next;

}

}

else

{

throw new IndexOutOfRangeException();

}

Count--;

}

public void Reverse()

{

ListNode<T> previousListNode = null;

var currentNode = \_head;

while (currentNode != null)

{

var next = currentNode.Next;

currentNode.Next = previousListNode;

previousListNode = currentNode;

currentNode = next;

}

\_head = previousListNode;

}

public override string ToString()

{

var result = new StringBuilder();

result.Append('[');

foreach (var elem in this) result.Append(elem).Append(", ");

result.Append(']');

return result.ToString();

}

}

}

***TreeSet.cs***

using System;

using System.Collections;

using System.Collections.Generic;

using DataStructures.structures.enumerators;

using DataStructures.structures.models;

namespace DataStructures.structures

{

public class TreeSet<T> : common.ISet<T> where T : IComparable<T>

{

private const int Equal = 0;

private const int LessThan = -1;

private const int GreaterThan = 1;

private TreeNode<T> \_root;

public TreeSet()

{

\_root = null;

}

IEnumerator IEnumerable.GetEnumerator()

{

return GetEnumerator();

}

public IEnumerator<T> GetEnumerator()

{

return new TreeSetEnumerator<T>(\_root);

}

public bool Add(T data)

{

if (Contains(data)) return false;

if (\_root == null)

\_root = new TreeNode<T>(data);

else

AddElem(data, \_root);

return true;

}

public bool Contains(T data)

{

return FindElem(data, \_root);

}

public bool Remove(T data)

{

if (!Contains(data)) return false;

\_root = RemoveElem(data, \_root);

return true;

}

private static TreeNode<T> RemoveElem(T data, TreeNode<T> node)

{

if (node == null) return null;

var compareTo = data.CompareTo(node.Data);

switch (compareTo)

{

case LessThan:

node.Left = RemoveElem(data, node.Left);

break;

case GreaterThan:

node.Right = RemoveElem(data, node.Right);

break;

default:

{

if (node.Right == null) return node.Left;

if (node.Left == null) return node.Right;

var temp = node;

node = MinElem(temp.Right);

node.Right = DeleteMin(temp.Right);

node.Left = temp.Left;

break;

}

}

return node;

}

private static TreeNode<T> DeleteMin(TreeNode<T> node)

{

if (node.Left == null) return node.Right;

node.Left = DeleteMin(node.Left);

return node;

}

private static TreeNode<T> MinElem(TreeNode<T> node)

{

return node.Left == null ? node : MinElem(node.Left);

}

private static bool FindElem(T data, TreeNode<T> node)

{

if (node == null)

return false;

var compareTo = data.CompareTo(node.Data);

return compareTo switch

{

Equal => true,

LessThan => FindElem(data, node.Left),

GreaterThan => FindElem(data, node.Right),

\_ => false

};

}

private static void AddElem(T data, TreeNode<T> node)

{

if (data.CompareTo(node.Data) == LessThan)

{

if (node.Left != null)

AddElem(data, node.Left);

else

node.Left = new TreeNode<T>(data);

}

else

{

if (node.Right != null)

AddElem(data, node.Right);

else

node.Right = new TreeNode<T>(data);

}

}

}

}

***InsertionSort.cs***

using System;

namespace DataStructures.algorithms

{

public static class InsertionSort

{

private static void Swap<T>(ref T e1, ref T e2)

{

var temp = e1;

e1 = e2;

e2 = temp;

}

public static void Sort<T>(T[] array) where T : IComparable<T>

{

for (var i = 1; i < array.Length; i++)

{

var key = array[i];

var j = i;

while (j > 1 && array[j - 1].CompareTo(key) > 0)

{

Swap(ref array[j - 1], ref array[j]);

j--;

}

array[j] = key;

}

}

}

}

***MergeSort.cs***

using System;

namespace DataStructures.algorithms

{

public static class MergeSort

{

private static void Sort<T>(T[] array, int left, int right) where T : IComparable<T>

{

if (left >= right) return;

var middle = (left + right) / 2;

Sort(array, left, middle);

Sort(array, middle + 1, right);

Merge(array, left, middle, right);

}

public static void Sort<T>(T[] array) where T : IComparable<T>

{

Sort(array, 0, array.Length - 1);

}

private static void Merge<T>(T[] array, in int left, int middle, in int right) where T : IComparable<T>

{

var leftArray = new T[middle - left + 1];

var rightArray = new T[right - middle];

Array.Copy(array, left, leftArray, 0, middle - left + 1);

Array.Copy(array, middle + 1, rightArray, 0, right - middle);

var i = 0;

var j = 0;

for (var k = left; k < right + 1; k++)

if (i == leftArray.Length)

{

array[k] = rightArray[j];

j++;

}

else if (j == rightArray.Length)

{

array[k] = leftArray[i];

i++;

}

else if (leftArray[i].CompareTo(rightArray[j]) <= 0)

{

array[k] = leftArray[i];

i++;

}

else

{

array[k] = rightArray[j];

j++;

}

}

}

}