|  |  |
| --- | --- |
|  |  |
| МИНОБРНАУКИ РОССИИ | | |
| Федеральное государственное бюджетное образовательное учреждение  высшего профессионального образования  **«Московский технологический университет»**  **МИРЭА** | | |

Институт Информационных Технологий

Кафедра Корпоративных Информационных Систем

**ОТЧЕТ**

по Лабораторной Работе №2

на тему

«Обобщенное программирование. Generic классы.»

по дисциплине

«ООП»

Выполнил студент группы ИСБО-11-16 Шайхуллин С.В.

Принял Cтарший преподаватель Мирзоян Д.И.

Выполнено «\_\_\_» \_\_\_\_\_\_\_\_\_2017 г.

Зачтено «\_\_\_» \_\_\_\_\_\_\_\_\_ 2017г.

Москва, 2017

**Теоретическая часть**

C выходом версии 2.0 фреймворк .NET стал поддерживать обобщенные типы (generics), а также создание обобщенных методов. Чтобы разобраться в особенности данного явления, сначала посмотрим на проблему, которая могла возникнуть до появления обобщенных типов.

Упаковка (boxing) предполагает преобразование объекта значимого типа (например, типа int) к типу object. При упаковке общеязыковая среда CLR обертывает значение в объект типа System.Object и сохраняет его в управляемой куче (хипе). Распаковка (unboxing), наоборот, предполагает преобразование объекта типа object к значимому типу. Упаковка и распаковка ведут к снижению производительности, так как системе надо осуществить необходимые преобразования. Кроме того, существует другая проблема - проблема безопасности типов. Угловые скобки в описании class Account<T> указывают, что класс является обобщенным, а тип T, заключенный в угловые скобки, будет использоваться этим классом. Необязательно использовать именно букву T, это может быть и любая другая буква или набор символов. Причем сейчас нам неизвестно, что это будет за тип, это может быть любой тип. Поэтому параметр T в угловых скобках в еще называется универсальным параметром, так как вместо него можно подставить любой тип. Кроме обобщенных классов можно также создавать обобщенные методы, которые точно также будут использовать универсальные параметры. Поддержка обобщений (generics) — одно из наиболее впечатляющих нововведений в языке С#. С помощью синтаксиса обобщений можно определить тип, зависящий от другого типа, который не специфицирован в точке определения, а вместо этого определен в точке использования обобщенного типа. Для примера рассмотрим некоторый тип коллекции. С применением типов коллекций обычно создаются такие вещи, как списки, очереди и стеки. Типы коллекций, существующие со времен .NET 1.0, предназначены для хранения любых типов в CLR, поскольку они содержат экземпляры Object и все, что наследуется от Object. Однако при этом вся информация о конкретном типе элементов коллекции отбрасывается, и все умение компилятора отлавливать ошибки типа оказывается непригодным. Ссылку типа, полученную из этой коллекции, можно приводить к любому другому типу, который, по идее, должен иметь элемент, а это может привести к сбою во время выполнения. К тому же первоначальные типы коллекций могут содержать произвольную смесь типов элементов вместо того, чтобы принуждать пользователя в коллекцию вставлять только экземпляры определенного типа. Можно попробовать решить эту проблему, разработав собственные типы, такие как ListOf Integers и ListOf Strings, для каждого типа, который планируется хранить в коллекции. Однако очень скоро обнаружится, что большая часть управляемого кода таких списков будет похожей, или обобщенной, независимо от типа хранимых элементов.

Ключевым понятием здесь является обобщенный. Появляется возможность объявить открытый (или обобщенный) тип и написать общий код лишь однажды. Пользователь этого типа может затем указать, какой тип элементов будет содержать коллекция, прямо в точке, где он решит использовать ее. Вдобавок применение обобщений сулит выигрыш в эффективности.

**Задание**

Создать модуль (библиотеку), предоставляющую шаблоны двоичного и общего дерева с заданным интерфейсом.

1. Добавление элемента в дерево к известному родителю (реализовано выше)
2. Добавление элемента в дочерние к известному элементу
3. Удаление элемента из дерева (и всех его дочерних)
4. Удаление элемента из дочерних известного элемента (и всех его дочерних)
5. Смена родительского элемента
6. Удаление элемента из дерева с переносом его дочерних в родительский элемент
7. Удаление элемента из дочерних известного элемента с переносом дочерних в родительский элемент
8. Обход дерева (реализация интерфейсов IEnumerable и IEnumerator)

**Алгоритм решения задачи**

1. Спроектировать классы общего дерева и его элемента
2. Спроектировать классы бинарного дерева и его элемента
3. Используя элемент TreeView, фреймворка Windows Forms, продемонстрировать их работу

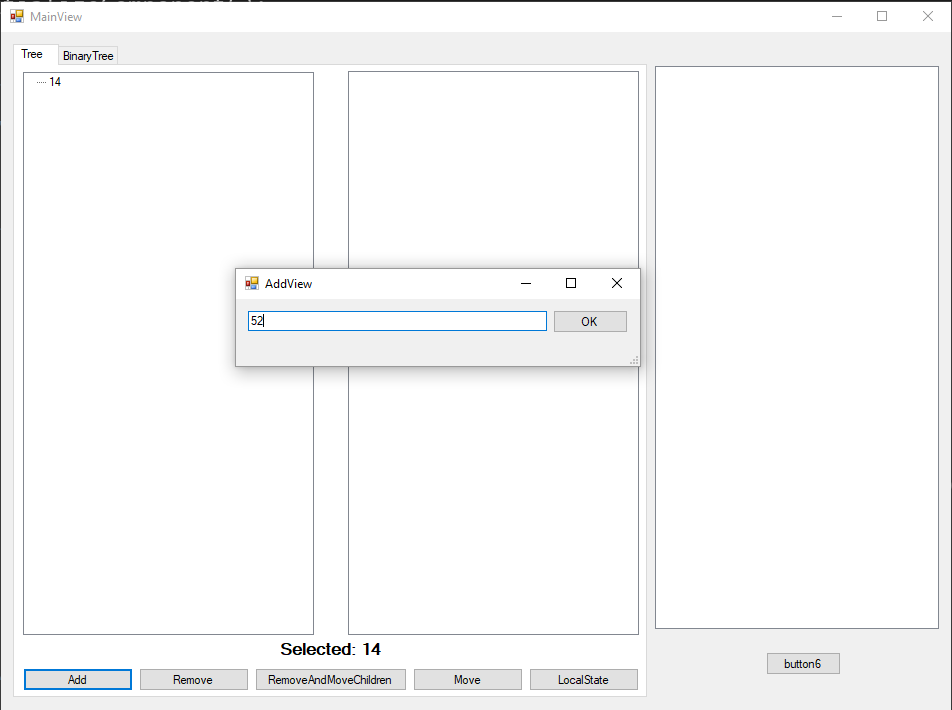
**Тестирование**

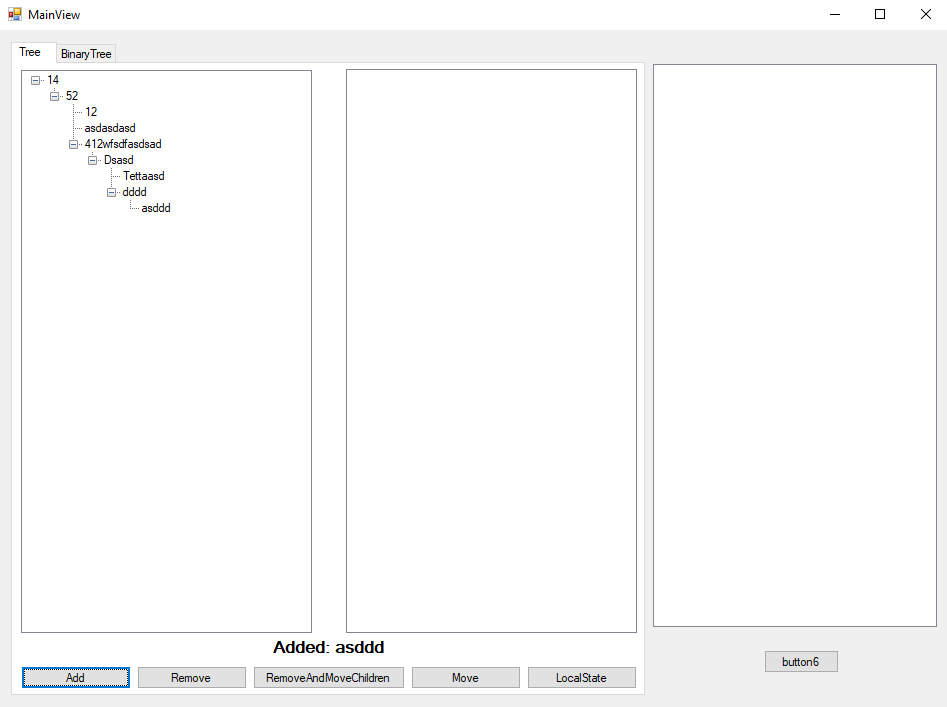
1. Протестировать добавление, удаление, удаление с последующим перемещением дочерних элементов удаленного элемента в элемент-родитель, перемешение для общего дерева
2. Протестировать добавление и удаление элемента для бинарного дерева поиска.
3. Продемонстрировать обход деревьев с помощью перечисления

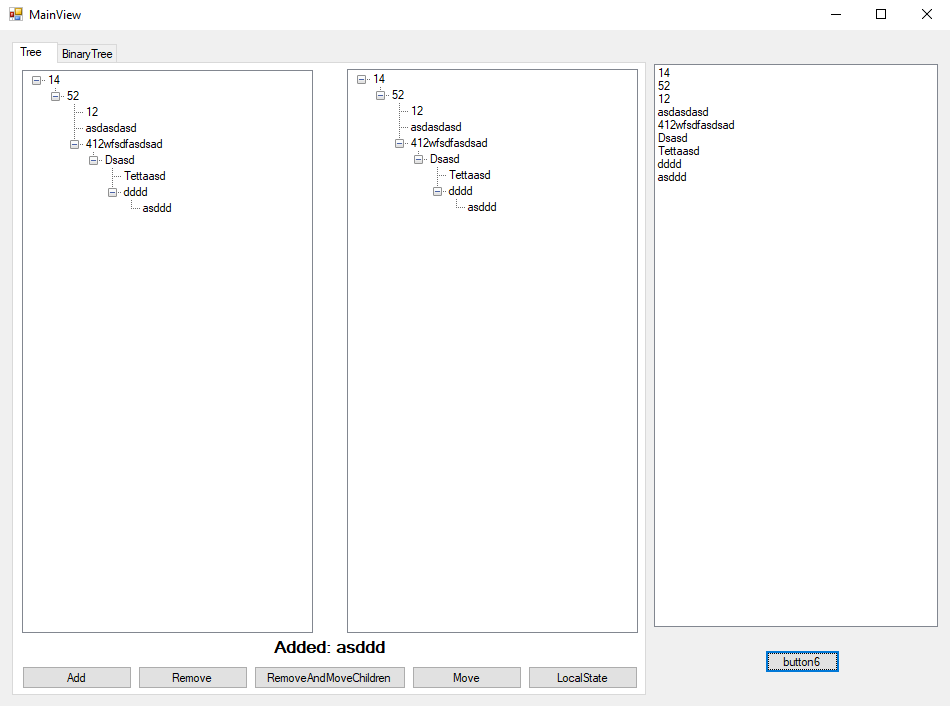
**Заключение**

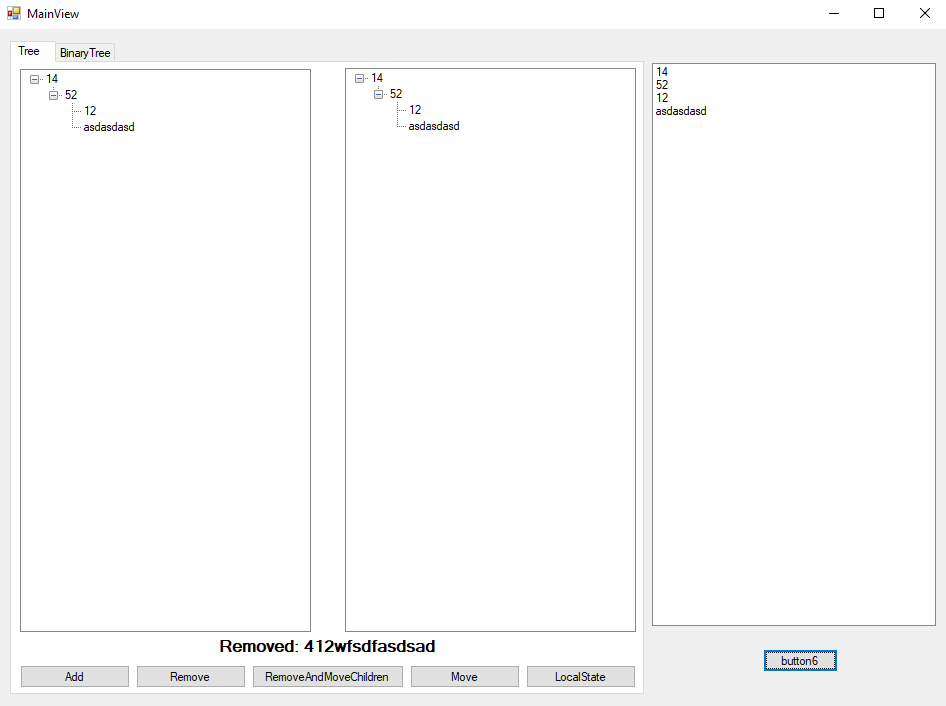
В данной лабораторной работе мы научились применять обобщения для создания строго типизированных структур данных, в данном примере – деревьев.

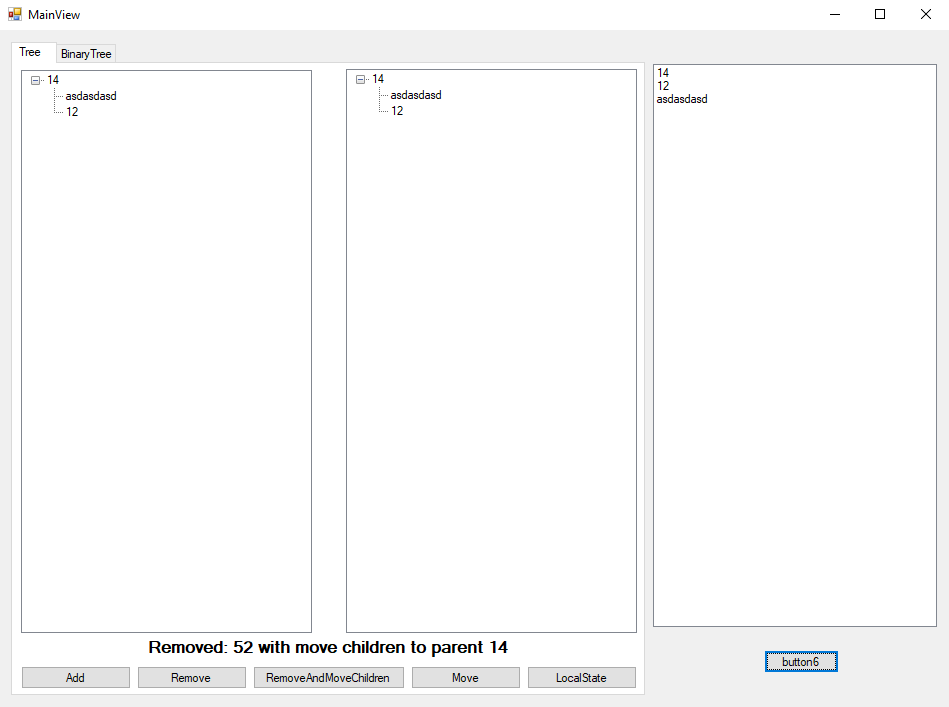
**Пример работы**

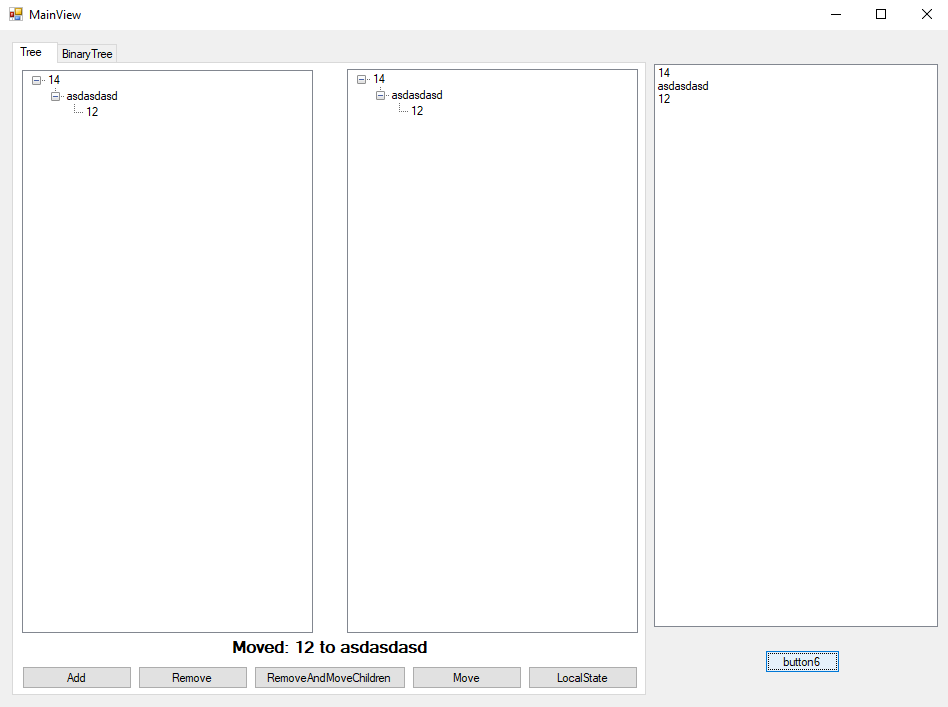


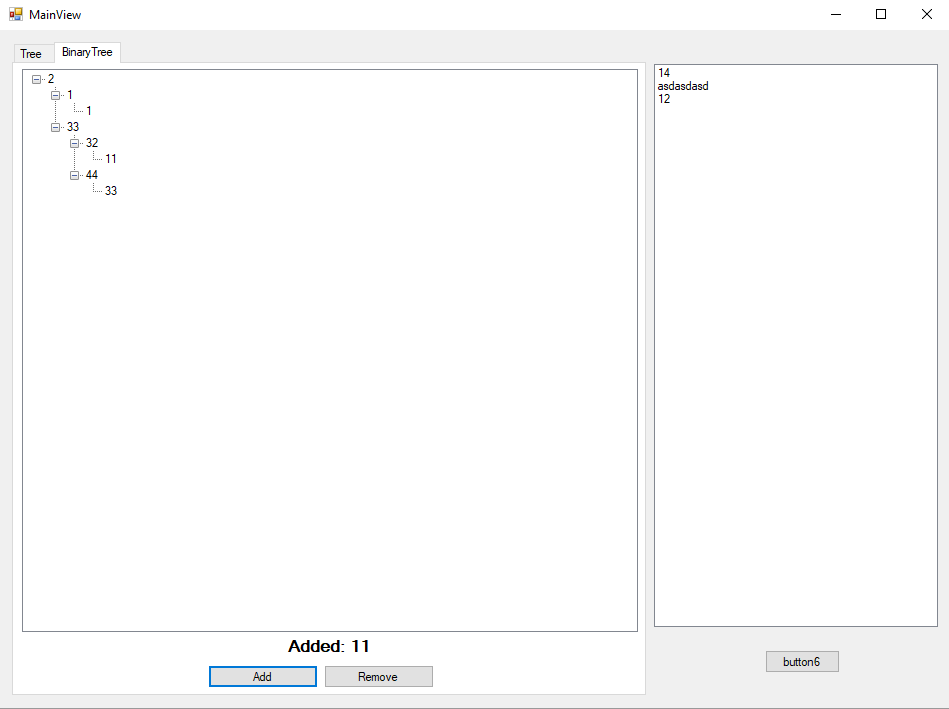


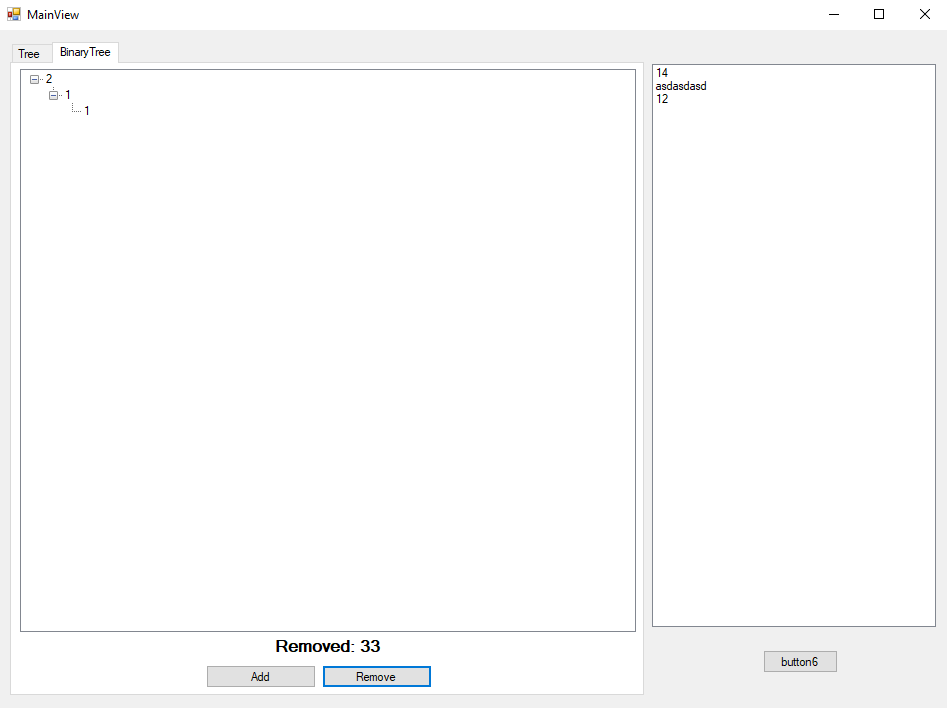












**Исходный код**

using System;

using System.Linq;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace Shaykhullin.Lab2.Views

{

public partial class MainView : Form

{

private TreeNode prevNode;

private Tree<TreeNode> tree;

private BinaryTree<TreeNode> binaryTree;

public MainView()

{

InitializeComponent();

}

private void OnAddClicked(object sender, EventArgs a)

{

var node = leftTree.SelectedNode;

if(node == null)

{

if(tree == null)

{

var selected = new AddView();

selected.button1.Click += (s, e) =>

{

if (string.IsNullOrWhiteSpace(selected.textBox1.Text))

{

selected.label1.Text = "Enter a node name!";

}

else

{

tree = new Tree<TreeNode>(new TreeNode(selected.textBox1.Text));

leftTree.Nodes.Add(tree.Data);

treeStatus.Text = $"Added: {tree.Data.Text}";

selected.Hide();

}

};

selected.ShowDialog();

}

else

{

treeStatus.Text = "Select node!";

}

}

else

{

var selected = new AddView();

selected.button1.Click += (s, e) =>

{

if (string.IsNullOrWhiteSpace(selected.textBox1.Text))

{

selected.label1.Text = "Enter a node name!";

}

else

{

var treeNode = new TreeNode<TreeNode>(new TreeNode(selected.textBox1.Text));

var parent = tree.Single(t => t.Data == node);

treeNode.AddTo(parent);

node.Nodes.Add(treeNode.Data);

node.Expand();

treeStatus.Text = $"Added: {treeNode.Data.Text}";

selected.Hide();

}

};

selected.ShowDialog();

}

}

private void OnRemoveClicked(object sender, EventArgs e)

{

var node = leftTree.SelectedNode;

if(node == null)

{

treeStatus.Text = "Select node!";

return;

}

var treeNode = tree.Single(t => t.Data == node);

if(treeNode is Tree<TreeNode>)

{

tree = null;

}

treeNode.RemoveFromParent();

leftTree.Nodes.Remove(node);

treeStatus.Text = $"Removed: {node.Text}";

leftTree.SelectedNode = null;

}

private void OnRemoveAndMoveChildrenClicked(object sender, EventArgs e)

{

var node = leftTree.SelectedNode;

if (node == null)

{

treeStatus.Text = "Select node!";

return;

}

var treeNode = tree.Single(t => t.Data == node);

if (treeNode is Tree<TreeNode>)

{

tree = null;

}

treeNode.RemoveAndMoveChildrenToParent();

if(node.Parent != null)

{

if(node.Nodes.Count == 0)

{

treeStatus.Text = $"Removed: {node.Text} no children";

}

else

{

for (var i = node.Nodes.Count - 1; i >= 0; i--)

{

var child = node.Nodes[i];

node.Nodes.Remove(child);

node.Parent.Nodes.Add(child);

}

treeStatus.Text = $"Removed: {node.Text} with move children to parent {node.Parent.Text}";

}

}

else

{

treeStatus.Text = $"Removed: {node.Text}";

}

leftTree.Nodes.Remove(node);

leftTree.SelectedNode = null;

}

private void OnMoveClicked(object sender, EventArgs e)

{

var node = leftTree.SelectedNode;

if(node == null)

{

treeStatus.Text = "Select node!";

return;

}

if(prevNode == null)

{

prevNode = node;

treeStatus.Text = $"Selected: {prevNode.Text} to move";

return;

}

if(prevNode == node)

{

treeStatus.Text = "Can't move equal nodes!";

return;

}

var treeNode = tree.Single(t => t.Data == prevNode);

var parent = tree.Single(t => t.Data == node);

tree.Move(treeNode, parent);

prevNode.Parent.Nodes.Remove(prevNode);

node.Nodes.Add(prevNode);

node.Expand();

treeStatus.Text = $"Moved: {prevNode.Text} to {node.Text}";

prevNode = null;

}

private async void OnNodeSelected(object sender, MouseEventArgs e)

{

await Task.Delay(150);

treeStatus.Text = $"Selected: {leftTree.SelectedNode?.Text}";

}

private void OnLocalStateClicked(object sender, EventArgs e)

{

rightTree.Nodes.Clear();

rightTree.Nodes.Add((TreeNode)tree.Data.Clone());

rightTree.ExpandAll();

}

private void OnBinaryAddClicked(object sender, EventArgs a)

{

if (binaryTree == null)

{

var selected = new AddView();

selected.button1.Click += (s, e) =>

{

if (!int.TryParse(selected.textBox1.Text, out var \_))

{

selected.label1.Text = "Node must be a number!";

}

else

{

binaryTree = new BinaryTree<TreeNode>(new TreeNode(selected.textBox1.Text),

(t1, t2) => int.Parse(t1.Text) < int.Parse(t2.Text))

{

Data =

{

Tag = Guid.NewGuid()

}

};

binaryTreeView.Nodes.Add(binaryTree.Data);

binaryTreeStatus.Text = $"Added: {binaryTree.Data.Text}";

selected.Hide();

}

};

selected.ShowDialog();

}

else

{

var selected = new AddView();

selected.button1.Click += (s, e) =>

{

if (!int.TryParse(selected.textBox1.Text, out var \_))

{

selected.label1.Text = "Node must be a number!";

}

else

{

var treeNode = new BinaryTreeNode<TreeNode>(new TreeNode(selected.textBox1.Text))

{

Data =

{

Tag = Guid.NewGuid()

}

};

binaryTree.Add(treeNode);

binaryTreeView.Nodes.Clear();

binaryTreeView.Nodes.Add(binaryTree.Render());

binaryTreeView.ExpandAll();

binaryTreeStatus.Text = $"Added: {treeNode.Data.Text}";

selected.Hide();

}

};

selected.ShowDialog();

}

}

private void OnBinaryRemoveClicked(object sender, EventArgs e)

{

var node = binaryTreeView.SelectedNode;

if(node == null)

{

binaryTreeStatus.Text = "Select node!";

return;

}

var treeNode = binaryTree.First(t => t.Data.Tag == node.Tag);

if(treeNode is BinaryTree<TreeNode>)

{

binaryTree = null;

}

if(binaryTree != null)

{

treeNode.Remove();

binaryTreeView.Nodes.Clear();

binaryTreeView.Nodes.Add(binaryTree.Render());

binaryTreeView.ExpandAll();

}

else

{

binaryTreeView.Nodes.Clear();

}

binaryTreeStatus.Text = $"Removed: {treeNode.Data.Text}";

}

private async void OnBinaryNodeSelected(object sender, MouseEventArgs e)

{

await Task.Delay(150);

binaryTreeStatus.Text = $"Selected: {binaryTreeView.SelectedNode?.Text}";

}

private void OnRenderButtonClicked(object sender, EventArgs e)

{

renderListView.Items.Clear();

renderListView.Items.AddRange(tree.Select(x => x.Data.Text).ToArray());

}

}

}

using System;

namespace Shaykhullin.Lab2

{

public class BinaryTree<TData> : BinaryTreeNode<TData>

{

public Func<TData, TData, bool> Comparer { get; set; }

public BinaryTree(TData data, Func<TData, TData, bool> comparer) : base(data)

{

Comparer = comparer ?? throw new ArgumentNullException(nameof(comparer));

}

public void Add(BinaryTreeNode<TData> node)

{

AddRecursive(this, node);

}

private void AddRecursive(BinaryTreeNode<TData> parent, BinaryTreeNode<TData> node)

{

while(true)

{

if(Comparer(node.Data, parent.Data))

{

if(parent.Left == null)

{

parent.Left = node;

node.Parent = parent;

}

else

{

parent = parent.Left;

continue;

}

}

else

{

if(parent.Right == null)

{

parent.Right = node;

node.Parent = parent;

}

else

{

parent = parent.Right;

continue;

}

}

break;

}

}

public void Remove(BinaryTreeNode<TData> node)

{

node.Remove();

}

}

}

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Windows.Forms;

namespace Shaykhullin.Lab2

{

public static class Extensions

{

public static TreeNode Render(this BinaryTreeNode<TreeNode> tree)

{

return new TreeNode(tree.Data.Text,

new[] { tree.Left, tree.Right }

.Where(x => x != null)

.Select(x => x.Render()).ToArray())

{

Tag = tree.Data.Tag

};

}

}

public class BinaryTreeNode<TData> : IEnumerable<BinaryTreeNode<TData>>

{

public TData Data { get; set; }

public BinaryTreeNode<TData> Parent { get; set; }

public BinaryTreeNode<TData> Left { get; set; }

public BinaryTreeNode<TData> Right { get; set; }

public BinaryTreeNode(TData data)

{

Data = data;

}

public void Remove()

{

if (Parent?.Left == this)

{

Parent.Left = null;

}

else if(Parent?.Right == this)

{

Parent.Right = null;

}

}

public IEnumerator<BinaryTreeNode<TData>> GetEnumerator()

{

foreach (var node in Left ?? Enumerable.Empty<BinaryTreeNode<TData>>())

{

yield return node;

}

yield return this;

foreach (var node in Right ?? Enumerable.Empty<BinaryTreeNode<TData>>())

{

yield return node;

}

}

IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

}

}

namespace Shaykhullin.Lab2

{

public class Tree<TData> : TreeNode<TData>

{

public Tree(TData data) : base(data)

{

}

public void Add(TreeNode<TData> parent, TreeNode<TData> child)

{

child.AddTo(parent);

}

public void Remove(TreeNode<TData> node)

{

node.RemoveFromParent();

}

public void RemoveAndMoveChildrenToParent(TreeNode<TData> node)

{

node.RemoveAndMoveChildrenToParent();

}

public void Move(TreeNode<TData> node, TreeNode<TData> parent)

{

node.MoveTo(parent);

}

public virtual TreeNode<TData> Create(TData data)

{

return new TreeNode<TData>(data);

}

}

}

using System;

using System.Collections;

using System.Collections.Generic;

namespace Shaykhullin.Lab2

{

public class TreeNode<TData> : IEnumerable<TreeNode<TData>>

{

public TData Data { get; set; }

public TreeNode<TData> Parent { get; set; }

public IList<TreeNode<TData>> Children { get; set; }

public TreeNode(TData data)

{

Data = data;

Children = new List<TreeNode<TData>>();

}

public void AddTo(TreeNode<TData> parent)

{

if (Parent != null)

{

throw new InvalidOperationException($"TreeNode<TData> {Data} have already added to parent");

}

Parent = parent;

Parent.Children.Add(this);

}

public void RemoveFromParent()

{

Parent?.Children.Remove(this);

Parent = null;

}

public void RemoveAndMoveChildrenToParent()

{

if (Parent == null)

{

RemoveFromParent();

return;

}

foreach (var child in Children)

{

child.Parent = null;

child.AddTo(Parent);

}

RemoveFromParent();

}

public void MoveTo(TreeNode<TData> parent)

{

if (Parent == null)

{

throw new InvalidOperationException($"TreeNode<TData> {Data} have no parent");

}

RemoveFromParent();

AddTo(parent);

}

public IEnumerator<TreeNode<TData>> GetEnumerator()

{

yield return this;

foreach (var child in Children)

{

foreach (var data in child)

{

yield return data;

}

}

}

IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

}

}