Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

Звіт
з лабораторної роботи №1 з дисципліни
«Сучасні технології розробки WEB-застосувань на платформі Microsoft.NET»

«Узагальнені типи (Generic) з підтримкою подій. Колекції»

Варіант 3

Виконав студент ІП-13 Макарчук Лідія Олександрівна
(шифр, прізвище, ім'я, по батькові)

Перевірив Бардін В.
(прізвище, ім'я, по батькові)

Київ 2023

Лабораторна робота №1

Варіант 3

Тема: Узагальнені типи (Generic) з підтримкою подій. Колекції.

Мета: навчитися проектувати та реалізовувати узагальнені типи, а також типи з підтримкою подій.

Постановка задачі

1. Розробити клас власної узагальненої колекції, використовуючи стандартні інтерфейси колекцій із бібліотек System.Collections та System.Collections.Generic. Стандартні колекції при розробці власної не застосовувати. Для колекції передбачити методи внесення даних будь-якого типу, видалення, пошуку та ін. (відповідно до типу колекції).
2. Додати до класу власної узагальненої колекції підтримку подій та обробку виключних ситуацій.
3. Опис класу колекції та всіх необхідних для роботи з колекцією типів зберегти у динамічній бібліотеці.
4. Створити консольний додаток, в якому продемонструвати використання розробленої власної колекції, підписку на події колекції.

3	Бінарне дерево	Додавання вузлів, обходи дерева, перевірка на наявність, пошук(видалення реалізовувати не обов'язково)	Збереження даних за допомогою динамічно зв'язаних вузлів

Виконання завдань

Код

// Node.cs

```
namespace MyBinaryTree;

public class Node<T> where T : IComparable<T>
{
    private T _value;
    public T Value { get => _value; internal set => _value = value; }
```

```
    public Node<T>? Left;

    public Node<T>? Right;

    public Node(T value)
    {
        _value = value;
    }
    internal T InOrderSuccessor()
    {
        Node<T> current = this;
        T value = current.Value;
        while (current.Left != null)
        {
            value = current.Left.Value;
            current = current.Left;
        }
        return value;
    }
}
```

// BinaryTree.cs

```
using MyBinaryTree.EnumeratorFactories;
using MyBinaryTree.Interfaces;
using System.Collections;

namespace MyBinaryTree;

public class BinaryTree<T> : ICollection<T> where T : IComparable<T>
{
    private Node<T>? _root;

    private IEnumeratorFactory<T> _enumeratorFactory;

    private readonly IComparer<T> _comparer;

    private int _count = 0;

    private int _version = 0;

    public Node<T>? Root => _root;

    public int Count => _count;

    public int Version => _version;

    public bool IsReadOnly => false;

    public IEnumeratorFactory<T> EnumeratorFactory { set => _enumeratorFactory = value; }

    public event EventHandler? TreeCleared;

    public event EventHandler<BinaryTreeEventArgs<T>>? ItemAdded;

    public event EventHandler<BinaryTreeEventArgs<T>>? ItemRemoved;

    public BinaryTree() : this(new InorderEnumeratorFactory<T>()) { }

    public BinaryTree(IComparer<T> comparer) : this(new InorderEnumeratorFactory<T>(),
comparer) { }

    public BinaryTree(IEnumeratorFactory<T> enumeratorFactory) : this(enumeratorFactory,
Comparer<T>.Default) { }

    public BinaryTree(IEnumeratorFactory<T> enumeratorFactory, IComparer<T> comparer)
    {
        _enumeratorFactory = enumeratorFactory;
        _comparer = comparer;
```

```
    }

    public void Add(T item)
    {
        if (item is null)
            throw new ArgumentNullException();

        if (_root == null)
        {
            _root = new Node<T>(item);
        }
        else
        {
            Node<T>? previous;
            Node<T>? current = _root;

            do
            {
                previous = current;
                current = _comparer.Compare(item, current.Value) switch
                {
                    < 0 => current.Left,
                    > 0 => current.Right,
                    0 => throw new InvalidOperationException($"Tree already contains item
'{item}'")
                };
            }
            while (current != null);

            if (_comparer.Compare(item, previous.Value) > 0)
                previous.Right = new Node<T>(item);
            else
                previous.Left = new Node<T>(item);
        }
        _count++;
        _version++;
        ItemAdded?.Invoke(this, new BinaryTreeEventArgs<T>(item));
    }

    public bool Contains(T item)
    {
        Node<T>? current = _root;
        while (current != null)
        {
            switch (_comparer.Compare(item, current.Value))
            {
                case < 0: current = current.Left; break;
                case > 0: current = current.Right; break;
                default: return true;
            }
        }
        return false;
    }

    public void Clear()
    {
        _root = null;
        _count = 0;
        _version++;
        TreeCleared?.Invoke(this, EventArgs.Empty);
    }

    public void CopyTo(T[] array, int arrayIndex)
    {
        if (array == null)
            throw new ArgumentNullException(nameof(array));
        if (arrayIndex < 0 || array.Length - arrayIndex < _count)
        {
            throw new ArgumentOutOfRangeException(nameof(array));
```

```
    }

    if (_root == null)
        throw new InvalidOperationException("Tree does not contain any elements");

    foreach (var nodeValue in this)
    {
        array[arrayIndex] = nodeValue;
        arrayIndex++;
    }
}

public bool Remove(T item)
{
    if (item is null)
        return false;

    bool removed = false;
    _root = RemoveRecursion(_root, item);

    if (removed)
    {
        _count--;
        _version++;
        ItemRemoved?.Invoke(this, new BinaryTreeEventArgs<T>(item));
    }

    return removed;


    Node<T>? RemoveRecursion(Node<T>? current, T item)
    {
        if (current == null)
            return current;

        switch (_comparer.Compare(item, current.Value))
        {
            case < 0:
                {
                    current.Left = RemoveRecursion(current.Left, item);
                    break;
                }
            case > 0:
                {
                    current.Right = RemoveRecursion(current.Right, item);
                    break;
                }
            default:
                {
                    removed = true;
                    if (current.Left == null)
                    {
                        return current.Right;
                    }
                    else if (current.Right == null)
                    {
                        return current.Left;
                    }

                    current.Value = current.Right.InOrderSuccessor();
                    current.Right = RemoveRecursion(current.Right, current.Value);
                    break;
                }
        };

        return current;
    }
}
```

```
    public IEnumerator<T> GetEnumerator()
    {
        return _enumeratorFactory.CreateEnumerator(this);
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

// IEnumeratorFactory.cs

```
namespace MyBinaryTree.Interfaces;

public interface IEnumeratorFactory<T> where T : IComparable<T>
{
    IEnumerator<T> CreateEnumerator(BinaryTree<T> tree);
}
```

// PostorderEnumeratorFactory.cs

```
using MyBinaryTree.Interfaces;

namespace MyBinaryTree.EnumeratorFactories;

public class PostorderEnumeratorFactory<T> : IEnumeratorFactory<T> where T :
IComparable<T>
{
    public IEnumerator<T> CreateEnumerator(BinaryTree<T> tree)
    {
        Node<T>? node = tree.Root;

        if (node != null)
        {
            var nodes = new Stack<Node<T>>();
            Node<T>? current, previous = null;

            while (node != null || nodes.Count > 0)
            {
                if (node != null)
                {
                    nodes.Push(node);
                    node = node.Left;
                }
                else
                {
                    current = nodes.Peek();
                    if (current.Right != null && current.Right != previous)
                    {
                        node = current.Right;
                    }
                    else
                    {
                        yield return current.Value;
                        previous = current;
                        nodes.Pop();
                    }
                }
            }
        }
    }
}
```

// PreorderEnumeratorFactory.cs

```
using MyBinaryTree.Interfaces;

namespace MyBinaryTree.EnumeratorFactories;

public class PreorderEnumeratorFactory<T> : IEnumeratorFactory<T> where T :
IComparable<T>
```

```
{
    public IEnumerator<T> CreateEnumerator(BinaryTree<T> tree)
    {
        Node<T>? node = tree.Root;

        if (node != null)
        {
            var nodes = new Stack<Node<T>>();
            Node<T>? current = node;

            while (nodes.Count > 0 || current != null)
            {
                if (current != null)
                {
                    yield return current.Value;

                    if (current.Right != null)
                        nodes.Push(current.Right);

                    current = current.Left;
                }
                else
                {
                    current = nodes.Pop();
                }
            }
        }
    }
}
```

// InorderEnumeratorFactory.cs

```
using MyBinaryTree.Enumerators;
using MyBinaryTree.Interfaces;

namespace MyBinaryTree.EnumeratorFactories;

public class InorderEnumeratorFactory<T> : IEnumeratorFactory<T> where T : IComparable<T>
{
    public IEnumerator<T> CreateEnumerator(BinaryTree<T> tree)
    {
        return new InorderEnumerator<T>(tree);
    }
}
```

// InorderEnumerator.cs

```
using System.Collections;

namespace MyBinaryTree.Enumerators;

public class InorderEnumerator<T> : IEnumerator<T> where T : IComparable<T>
{
    private readonly BinaryTree<T> _tree;
    private readonly int _version;
    private Node<T>? _currentNode;
    private Stack<Node<T>> _nodes;
    private bool _shouldSetCurrentToRight = false;

    public InorderEnumerator(BinaryTree<T> tree)
    {
        _tree = tree;
        _version = tree.Version;
        _currentNode = tree.Root;
        _nodes = new Stack<Node<T>>();
    }
    public T Current => _currentNode!.Value;

    object IEnumerator.Current => _currentNode!.Value;

    public bool MoveNext()
```

```
    {
        if (_shouldSetCurrentToRight)
            _currentNode = _currentNode!.Right;

        while (_nodes.Count > 0 || _currentNode != null)
        {
            if (_currentNode != null)
            {
                _nodes.Push(_currentNode);
                _currentNode = _currentNode.Left;
            }
            else
            {
                _currentNode = _nodes.Pop();
                _shouldSetCurrentToRight = true;
                return true;
            }
        }
        return false;
    }

    public void Reset()
    {
        if (_version != _tree.Version)
        {
            throw new InvalidOperationException();
        }
        _currentNode = _tree.Root;
        _nodes = new Stack<Node<T>>();
        _shouldSetCurrentToRight = false;
    }

    public void Dispose() { }
}
```

// BinaryTreeEventArgs.cs

```
namespace MyBinaryTree;

public class BinaryTreeEventArgs<T> : EventArgs
{
    private readonly T _item;

    public T Item => _item;

    public BinaryTreeEventArgs(T item)
    {
        _item = item;
    }
}
```

// Console client

// Program.cs

```
using MyBinaryTree;
using MyBinaryTree.EnumeratorFactories;
using MyBinaryTree.Interfaces;

namespace ConsoleClient
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Creating int tree...");
            var treeInt = new BinaryTree<int>();

            treeInt.ItemAdded += OnItemAdded<int>;
            treeInt.ItemRemoved += OnItemRemoved<int>;
            treeInt.TreeCleared += OnTreeCleared;
```

```
            treeInt.Add(5);
            treeInt.Add(3);
            treeInt.Add(-2);
            treeInt.Add(8);
            treeInt.Add(1);
            treeInt.Add(4);

            try
            {
                treeInt.Add(-2);
            }
            catch (InvalidOperationException ex)
            {
                Console.WriteLine($"EXCEPTION: {ex.Message}");
            }

            Console.WriteLine("\nTRAVERSALS:");
            Console.WriteLine("Inorder traversal: ");
            OutputTree(treeInt);
            Console.WriteLine("Preorder traversal: ");
            OutputTree(treeInt, new PreorderEnumeratorFactory<int>());
            Console.WriteLine("Postorder traversal: ");
            OutputTree(treeInt, new PostorderEnumeratorFactory<int>());

            const int firstInt = 100;
            const int secondInt = -2;

            Console.WriteLine($"\nTree contains item {firstInt}:
{GetSuccessOrFailureString(treeInt.Contains(firstInt))}");
            Console.WriteLine($"Trying remove item {firstInt}. Success:
{GetSuccessOrFailureString(treeInt.Remove(firstInt))}");
            Console.WriteLine($"\nTree contains item {secondInt}:
{GetSuccessOrFailureString(treeInt.Contains(secondInt))}");
            Console.WriteLine($"Trying remove item {secondInt}. Success:
{GetSuccessOrFailureString(treeInt.Remove(secondInt))}");

            Console.WriteLine($"\nTree contains {treeInt.Count} elements");

            const int arr1Size = 5;
            const int arr2Size = 2;
            int[] arr1 = new int[arr1Size];
            int[] arr2 = new int[arr2Size];

            try
            {
                Console.WriteLine($"Copying tree to array with size {arr1Size}...");
                treeInt.CopyTo( arr1, 0 );
                foreach ( int item in arr1)
                    Console.Write(item + " ");

                Console.WriteLine($"\nCopying tree to array with size {arr2Size}...");
                treeInt.CopyTo(arr2, 0);
                foreach (int item in arr2)
                    Console.Write(item + " ");
            }
            catch (ArgumentOutOfRangeException)
            {
                Console.WriteLine("EXCEPTION: array index is out of range");
            }

            Console.WriteLine("\nClearing tree...");
            treeInt.Clear();

Console.WriteLine("\n=====================================================");

            // String trees
            Console.WriteLine("Creating trees with strings...\n");
```

```
            Console.WriteLine("Case Sensitive tree");
            var treeStringCaseSensitive = new BinaryTree<string>();
            treeStringCaseSensitive.ItemAdded += OnItemAdded<string>;
            treeStringCaseSensitive.Add("Alice");
            treeStringCaseSensitive.Add("Bob");

            Console.WriteLine("Case Insensitive tree");
            var treeStringCaseInsensitive = new
BinaryTree<string>(StringComparer.OrdinalIgnoreCase);
            treeStringCaseInsensitive.ItemAdded += OnItemAdded<string>;
            treeStringCaseInsensitive.Add("Alice");
            treeStringCaseInsensitive.Add("Bob");

            string newStringItem = "alice";
            try
            {
                Console.WriteLine($"Inserting '{newStringItem}' into Case Sensitive
tree...");
                treeStringCaseSensitive.Add(newStringItem);

                Console.WriteLine($"Inserting '{newStringItem}' into Case Insensitive
tree...");
                treeStringCaseInsensitive.Add(newStringItem);
            }
            catch (InvalidOperationException ex)
            {
                Console.WriteLine($"EXCEPTION: {ex.Message}");
            }
        }

        // Methods
        public static void OutputTree<T>(BinaryTree<T> tree, IEnumeratorFactory<T>?
enumeratorFactory = null)
            where T :IComparable<T>
        {
            if (enumeratorFactory != null)
                tree.EnumeratorFactory = enumeratorFactory;

            foreach (var item in tree)
                Console.Write(item + " ");
            Console.WriteLine();
        }

        public static void OnItemAdded<T>(Object? sender, BinaryTreeEventArgs<T> e)
        {
            Console.WriteLine($"{"\u001b[32m"}Added {e.Item}{"\u001b[0m"}");
        }

        public static void OnItemRemoved<T>(Object? sender, BinaryTreeEventArgs<T> e)
        {
            Console.WriteLine($"{"\u001b[31m"}Removed {e.Item}{"\u001b[0m"}");
        }
        public static void OnTreeCleared(Object? sender, EventArgs e)
        {
            Console.WriteLine($"{"\u001b[35m"}Tree cleared event occured!{"\u001b[0m"}");
        }

        public static string GetSuccessOrFailureString(bool success)
        {
            string color = success ? "\u001b[32m" : "\u001b[31m";
            return $"{color}{success}\u001b[0m";
        }
    }
      }
```

Скріншоти результатів виконання:

```
Creating int tree...
Added 5
Added 3
Added -2
Added 8
Added 1
Added 4
EXCEPTION: Tree already contains item '-2'

TRAVERSALS:
Inorder traversal:
-2 1 3 4 5 8
Preorder traversal:
5 3 -2 1 4 8
Postorder traversal:
1 -2 4 3 8 5

Tree contains item 100: False
Trying remove item 100. Success: False

Tree contains item -2: True
Removed -2
Trying remove item -2. Success: True

Tree contains 5 elements
Copying tree to array with size 5...
1 4 3 8 5
Copying tree to array with size 2...
EXCEPTION: array index is out of range

Clearing tree...
Tree cleared event occured!

=========================================================
```

```
=========================================================
Creating trees with strings...

Case Sensitive tree
Added Alice
Added Bob
Case Insensitive tree
Added Alice
Added Bob
Inserting 'alice' into Case Sensitive tree...
Added alice
Inserting 'alice' into Case Insensitive tree...
EXCEPTION: Tree already contains item 'alice'
```

Висновки:

Висновок: під час виконання лабораторної роботи я розробила власну узагальнену колекцію – бінарне дерево. Було додано підтримку подій та обробку виключних ситуацій.