Міністерство освіти і науки України

КПІ ім. Ігоря Сікорського

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

Лабораторна робота № 1

З дисципліни “Сучасні технології розробки WEB-застосувань на платформі Microsoft.Net ”

студентки ІІ курсу ФІОТ

групи ІК-12

Макарчук Ольги

Перевірив:
Бардін В.

Київ 2023

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

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

Завдання:

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.

Варіант 3:

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

Посилання на код GitHub:

https://github.com/olha-makarchuk/DotNet_lab1-2/tree/master/DotNet_lab1-2

Було створено чотири класи:

1) Бінарне дерево:

```
public class BinaryTree<T>: ICollection<T>
    where T : IComparable<T>
```

2) Вузол бінарного дерева

```
public class Node<T>
    where T : IComparable<T>
```

3) Створення свого списка для потреб реалізації бінарного дерева:

```
public class MyList<T> : IEnumerable<T>
```

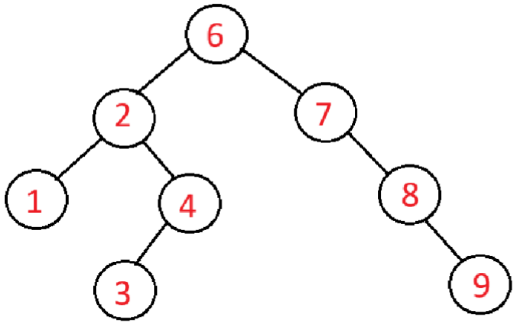
4) Клас аргументів подій:

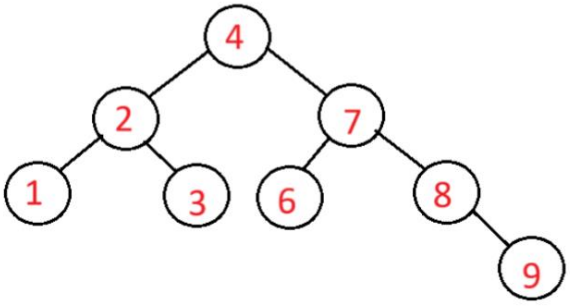
```
public class BinaryTreeEventsArgs<T> : EventArgs
```

Результат роботи програми:

```
Прямий обхід:
6, 2, 1, 4, 3, 7, 8, 9,
Зворотний обхід:
1, 3, 4, 2, 9, 8, 7, 6,
Центрований обхід:
1, 2, 3, 4, 6, 7, 8, 9,

Прямий обхід після балансування:
4, 2, 1, 3, 7, 6, 8, 9,

Видаляємо елемент - 2
Прямий обхід:
4, 3, 1, 7, 6, 8, 9,

Наявність елемента 8 - True
Наявність елемента 5 - False
```

Різниця між початковим деревом та деревом після балансування:





Код програми

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DotNet_lab1_2
{
    public class BinaryTree<T>: ICollection<T>
        where T : IComparable<T>
    {
        public event EventHandler<BinaryTreeEventsArgs<T>> ItemAdded;
        public event EventHandler<BinaryTreeEventsArgs<T>> ItemContained;
        public event EventHandler<BinaryTreeEventsArgs<T>> ItemRemoved;
        public event EventHandler ItemCleaned;

        public Node<T> Root { get; private set; }
        public int Count { get; private set; }

        public bool IsReadOnly => false;

        public void Add(T data)
        {
            if (data == null)
            {
                throw new ArgumentNullException(nameof(data), "Data cannot be
null.");
            }

            if (Root == null)
            {
                Root = new Node<T>(data);
                Count = 1;
                ItemAdded?.Invoke(this, new BinaryTreeEventsArgs<T>(data));
                return;
            }

            Root.Add(data);
            Count++;
        }

        public bool Contains(T data)
        {
            if (data == null)
            {
                throw new ArgumentNullException(nameof(data), "Data cannot be
null.");
            }

            if (Root != null)
            {
                bool found = Root.Contains(data);

                if (found)
                {
                    InvokeItemContained(data);
                }
                return found;
            }

            return false;
        }
        public void Clear()
        {
            Root = null;
            Count = 0;
```

```
            ItemCleaned?.Invoke(this, EventArgs.Empty);

        }

        public void Balance()
        {
            var inorderList = Inorder();
            T[] sortedArray = inorderList.ToArray();
            Root = CreateBalancedBST(sortedArray, 0, sortedArray.Length - 1);
            Count = sortedArray.Length;
        }

        private Node<T> CreateBalancedBST(T[] sortedArray, int start, int end)
        {
            if (start > end)
            {
                return null;
            }

            int mid = (start + end) / 2;
            Node<T> newNode = new Node<T>(sortedArray[mid]);

            newNode.Left = CreateBalancedBST(sortedArray, start, mid - 1);
            newNode.Right = CreateBalancedBST(sortedArray, mid + 1, end);

            return newNode;
        }

        public bool Remove(T data)
        {
            if (data == null)
            {
                throw new ArgumentNullException(nameof(data), "Data cannot be
null.");
            }

            bool success = Contains(data);

            if (success)
            {
                Root.Remove(Root, data);
                Count--;
                InvokeItemRemoved(data);
            }

            return success;
        }

        public MyList<T> Preorder()
        {
            if (Root == null)
            {
                return new MyList<T>();
            }

            return Preorder(Root);

        }

        public MyList<T> Postorder()
        {
            if (Root == null)
            {
                return new MyList<T>();
            }

            return Postorder(Root);

        }
```

```
public MyList<T> Inorder()
{
    if (Root == null)
    {
        return new MyList<T>();
    }

    return Inorder(Root);
}

private MyList<T> Preorder(Node<T> node)
{
    var list = new MyList<T>();
    if (node != null)
    {
        list.Add(node.Data);

        if (node.Left != null)
        {
            list.AddRange(Preorder(node.Left));
        }

        if (node.Right != null)
        {
            list.AddRange(Preorder(node.Right));
        }
    }
    return list;
}

private MyList<T> Postorder(Node<T> node)
{
    var list = new MyList<T>();
    if (node != null)
    {
        if (node.Left != null)
        {
            list.AddRange(Postorder(node.Left));
        }

        if (node.Right != null)
        {
            list.AddRange(Postorder(node.Right));
        }

        list.Add(node.Data);
    }
    return list;
}

private MyList<T> Inorder(Node<T> node)
{
    var list = new MyList<T>();
    if (node != null)
    {
        if (node.Left != null)
        {
            list.AddRange(Inorder(node.Left));
        }

        list.Add(node.Data);

        if (node.Right != null)
        {
            list.AddRange(Inorder(node.Right));
        }
    }
    return list;
}
```

```
        private void InvokeItemAdded(T data) => ItemAdded?.Invoke(this, new
BinaryTreeEventsArgs<T>(data));
        private void InvokeItemContained(T data) => ItemContained?.Invoke(this, new
BinaryTreeEventsArgs<T>(data));
        private void InvokeItemRemoved(T data) => ItemRemoved?.Invoke(this, new
BinaryTreeEventsArgs<T>(data));

        public void CopyTo(T[] array, int arrayIndex)
        {
            if (array == null)
            {
                throw new ArgumentNullException(nameof(array), "The destination
array cannot be null.");
            }

            if (arrayIndex < 0)
            {
                throw new ArgumentOutOfRangeException(nameof(arrayIndex), "Array
index cannot be negative.");
            }

            if (array.Length - arrayIndex < Count)
            {
                throw new ArgumentException("The destination array does not have
enough space.");
            }

            CopyTo(Root, array, ref arrayIndex);
        }

        private void CopyTo(Node<T> node, T[] array, ref int index)
        {
            if (node != null)
            {
                array[index++] = node.Data;
                CopyTo(node.Left, array, ref index);
                CopyTo(node.Right, array, ref index);
            }
        }

        public IEnumerator<T> GetEnumerator()
        {
            return Preorder().GetEnumerator();
        }

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


using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DotNet_lab1_2
{
    public class BinaryTreeEventsArgs<T> : EventArgs
    {
        public T Value { get; }

        public BinaryTreeEventsArgs(T value)
        {
            Value = value;
        }
    }
```

```
}


using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DotNet_lab1_2
{
    public class MyList<T> : IEnumerable<T>
    {
        private T[] items;
        private int count;

        public MyList()
        {
            items = new T[4];
            count = 0;
        }

        public void Add(T item)
        {
            if (item == null)
            {
                throw new ArgumentNullException(nameof(item), "Element cannot be
null");
            }

            if (count == items.Length)
            {
                Array.Resize(ref items, items.Length * 2);
            }
            items[count] = item;
            count++;
        }

        public void AddRange(IEnumerable<T> collection)
        {
            if (collection == null)
            {
                throw new ArgumentNullException(nameof(collection), "Collection
cannot be null");
            }

            foreach (var item in collection)
            {
                Add(item);
            }
        }


        public IEnumerator<T> GetEnumerator()
        {
            return new MyListEnumerator(this);
        }

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

        private class MyListEnumerator : IEnumerator<T>
        {
            private readonly MyList<T> list;
            private int index;

            public MyListEnumerator(MyList<T> list)
```

```
            {
                this.list = list;
                index = -1;
            }

            public T Current => list.items[index];

            object IEnumerator.Current => Current;

            public bool MoveNext()
            {
                index++;
                return index < list.count;
            }

            public void Reset()
            {
                index = -1;
            }

            public void Dispose()
            {
                // Немає необхідності в ресурсах для вивільнення
            }
        }
    }
}


using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DotNet_lab1_2
{
    public class Node<T>
        where T : IComparable<T>
    {
        public T Data { get; set; }
        public Node<T> Left { get; set; }
        public Node<T> Right { get; set; }

        public Node(T data)
        {
            Data = data;
        }

        public void Add(T data)
        {
            var node = new Node<T>(data);

            if (node.CompareTo(Data) == -1)
            {
                if (Left == null)
                {
                    Left = node;
                }
                else
                {
                    Left.Add(data);
                }
            }
            else
            {
                if (Right == null)
                {
                    Right = node;
                }
```

```
            else
            {
                Right.Add(data);
            }
        }
    }

    public bool Contains(T data)
    {
        var node = new Node<T>(data);

        int comparisonResult = node.CompareTo(Data);

        if (comparisonResult == 0)
        {
            return true;
        }
        else if (comparisonResult < 0 && Left != null)
        {
            return Left.Contains(data);
        }
        else if (comparisonResult > 0 && Right != null)
        {
            return Right.Contains(data);
        }

        return false;
    }


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

        int comparisonResult = data.CompareTo(node.Data);

        if (comparisonResult < 0)
        {
            node.Left = Remove(node.Left, data);
        }
        else if (comparisonResult > 0)
        {
            node.Right = Remove(node.Right, data);
        }
        else
        {
            if (node.Left == null)
            {
                return node.Right;
            }
            else if (node.Right == null)
            {
                return node.Left;
            }

            T minValue = FindMinValue(node.Right);
            node.Data = minValue;
            node.Right = Remove(node.Right, minValue);
        }

        return node;
    }

    private T FindMinValue(Node<T> node)
    {
        while (node.Left != null)
        {
```

```
                node = node.Left;
            }

            return node.Data;
        }

        public int CompareTo(T other)
        {
            return Data.CompareTo(other);
        }
    }
}
```

```
using DotNet_lab1_2;

var tree = new BinaryTree<int>();
tree.Add(6);
tree.Add(7);
tree.Add(2);
tree.Add(1);
tree.Add(8);
tree.Add(4);
tree.Add(3);
tree.Add(9);


Console.WriteLine("Прямий обхід:");
foreach (var item in tree.Preorder())
{
    Console.Write(item + ", ");
}

Console.WriteLine("\nЗворотний обхід:");
foreach (var item in tree.Postorder())
{
    Console.Write(item + ", ");
}

Console.WriteLine("\nЦентрований обхід:");

foreach (var item in tree.Inorder())
{
    Console.Write(item + ", ");
}

Console.WriteLine("\n\nПрямий обхід після балансування:");
tree.Balance();
foreach (var item in tree.Preorder())
{
    Console.Write(item + ", ");
}

int remove_element = 2;
Console.WriteLine("\n\nВидаляємо елемент - "+ remove_element+"\nПрямий обхід:");
tree.Remove(remove_element);
foreach (var item in tree.Preorder())
{
    Console.Write(item + ", ");
}

int contains_element1 = 8;
int contains_element2 = 5;
Console.WriteLine("\n\nНаявність елемента " + contains_element1+ " - " +
tree.Contains(contains_element1));
Console.WriteLine("Наявність елемента " + contains_element2 + " - " +
tree.Contains(contains_element2));

Console.Read();
```