Національний технічний університет 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»

на тему

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

Виконав:

студент групи ІС-11

Ходос Михайло

Київ – 2023

# Зміст

[1. Постановка задачі 3](file:///C:\Users\User\Desktop\Games%20and%20CO\Homework%20(4%20real!!!!)\Archive\BD\Лабораторна%20робота%20№1%20Ходос%20ІС-11%20(Спроба%204).docx#_Toc114738953)

[2. Лістинг коду 5](file:///C:\Users\User\Desktop\Games%20and%20CO\Homework%20(4%20real!!!!)\Archive\BD\Лабораторна%20робота%20№1%20Ходос%20ІС-11%20(Спроба%204).docx#_Toc114738954)

[3. Результати виконання](file:///C:\Users\User\Desktop\Games%20and%20CO\Homework%20(4%20real!!!!)\Archive\BD\Лабораторна%20робота%20№1%20Ходос%20ІС-11%20(Спроба%204).docx#_Toc114738955) 6

[4. Висновок](file:///C:\Users\User\Desktop\Games%20and%20CO\Homework%20(4%20real!!!!)\Archive\BD\Лабораторна%20робота%20№1%20Ходос%20ІС-11%20(Спроба%204).docx#_Toc114738955) 6

**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.



**Лістинг коду**

Клас MyCollection

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace MyCollection

{

    public class \_SortedList<TKey, TValue> : ICollection<KeyValuePair<TKey, TValue>>

    {

        //SortedList

        public KeyValuePair<TKey, TValue>[] \_items;

        public TKey[] keys;

        public TValue[] values;

        public int size;

        private int \_capacity;

        public event EventHandler AddElement;

        public event EventHandler RemoveElement;

        public event EventHandler ClearArray;

        public int Count => size;

        public bool IsReadOnly => false;

        public object SyncRoot => throw new NotImplementedException();

        public bool IsSynchronized => throw new NotImplementedException();

        public \_SortedList(int capacity)

        {

            if (capacity < 0)

            {

                throw new ArgumentOutOfRangeException(nameof(capacity));

            }

            size = 0;

            \_capacity = capacity;

            if (capacity == 0)

            {

                \_items = Array.Empty<KeyValuePair<TKey, TValue>>();

            }

            else

            {

                \_items = new KeyValuePair<TKey, TValue>[capacity];

                values = new TValue[capacity];

                keys = new TKey[capacity];

            }

        }

        public KeyValuePair<TKey, TValue> this[int index]

        {

            get => \_items[index];

        }

        public void Add(KeyValuePair<TKey, TValue> item)

        {

            if (\_capacity <= size)

            {

                ExpandCapacity();

            }

            int num = 0;

            if (size == 0)

            {

                size++;

                values[0] = item.Value;

                keys[0] = item.Key;

                \_items[0] = item;

                return;

            }

            num = Array.BinarySearch(keys, 0, size, item.Key);

            if (num >= 0)

            {

                return;

            }

            num = -num - 1;

            var TempArray = new KeyValuePair<TKey, TValue>[size + 1];

            var TempKeys = new TKey[size + 1];

            var TempValues = new TValue[size + 1];

            for (int i = 0; i < size + 1; i++)

            {

                if (i == num)

                {

                    TempArray[i] = item;

                    TempKeys[i] = item.Key;

                    TempValues[i] = item.Value;

                }

                else if (i < num)

                {

                    TempArray[i] = \_items[i];

                    TempKeys[i] = keys[i];

                    TempValues[i] = values[i];

                }

                else

                {

                    TempArray[i] = \_items[i - 1];

                    TempKeys[i] = keys[i - 1];

                    TempValues[i] = values[i - 1];

                }

            }

            \_items = TempArray;

            keys = TempKeys;

            values = TempValues;

            size++;

        }

        public void Add(TKey key, TValue value)

        {

            if (\_capacity <= size)

            {

                ExpandCapacity();

            }

            int num;

            if (size == 0)

            {

                size++;

                values[0] = value;

                keys[0] = key;

                \_items[0] = new KeyValuePair<TKey, TValue>(key, value);

                AddElement?.Invoke(this, EventArgs.Empty);

                return;

            }

            num = Array.BinarySearch(keys, 0, size, key);

            if (num >= 0)

            {

                return;

            }

            num = -num - 1;

            var TempArray = new KeyValuePair<TKey, TValue>[size + 1];

            var TempKeys = new TKey[size + 1];

            var TempValues = new TValue[size + 1];

            for (int i = 0; i < size + 1; i++)

            {

                if (i == num)

                {

                    TempArray[i] = new KeyValuePair<TKey, TValue>(key, value);

                    TempKeys[i] = key;

                    TempValues[i] = value;

                }

                else if (i < num)

                {

                    TempArray[i] = \_items[i];

                    TempKeys[i] = keys[i];

                    TempValues[i] = values[i];

                }

                else

                {

                    TempArray[i] = \_items[i - 1];

                    TempKeys[i] = keys[i - 1];

                    TempValues[i] = values[i - 1];

                }

            }

            \_items = TempArray;

            keys = TempKeys;

            values = TempValues;

            size++;

            AddElement?.Invoke(this, EventArgs.Empty);

        }

        public void Clear()

        {

            \_items = Array.Empty<KeyValuePair<TKey, TValue>>();

            values = Array.Empty<TValue>();

            keys = Array.Empty<TKey>();

            \_capacity = 0;

            size = 0;

            ClearArray?.Invoke(this, EventArgs.Empty);

        }

        public bool Contains(KeyValuePair<TKey, TValue> item)

        {

            int num = Array.BinarySearch(keys, 0, size, item.Key);

            if (num >= 0)

            {

                return true;

            }

            return false;

        }

        public bool Contains(TKey key)

        {

            int num = Array.BinarySearch(keys, 0, size, key);

            if (num >= 0)

            {

                return true;

            }

            return false;

        }

        public void CopyTo(KeyValuePair<TKey, TValue>[] array, int arrayIndex)

        {

            if (array == null)

            {

                throw new ArgumentNullException(nameof(array));

            }

            if (arrayIndex < 0)

            {

                throw new ArgumentOutOfRangeException(nameof(arrayIndex));

            }

            if (array.Length - arrayIndex < Count)

            {

                throw new ArgumentException();

            }

            for (int i = 0; i < Count; i++)

            {

                array[i + arrayIndex] = \_items[i];

            }

        }

        public void ExpandCapacity()

        {

            var TempSize = \_capacity \* 2;

            var TempArray = new KeyValuePair<TKey, TValue>[TempSize];

            var TempKeys = new TKey[TempSize];

            var TempValues = new TValue[TempSize];

            CopyTo(TempArray, 0);

            \_capacity = TempSize;

            \_items = TempArray;

            keys = TempKeys;

            values = TempValues;

            for (int i = 0; i < TempSize; i++)

            {

                keys[i] = TempArray[i].Key;

                values[i] = TempArray[i].Value;

            }

        }

        public bool Remove(KeyValuePair<TKey, TValue> item)

        {

            int num = Array.BinarySearch(keys, 0, size, item.Key);

            if (num < 0)

            {

                throw new ArgumentException("Element does not exist");

            }

            var TempArray = new KeyValuePair<TKey, TValue>[size - 1];

            var TempKeys = new TKey[size - 1];

            var TempValues = new TValue[size - 1];

            for (int i = 0; i < \_items.Length; i++)

            {

                if (i != num)

                {

                    TempArray[i] = \_items[i];

                    TempKeys[i] = \_items[i].Key;

                    TempValues[i] = \_items[i].Value;

                }

            }

            \_items = TempArray;

            keys = TempKeys;

            values = TempValues;

            size--;

            RemoveElement?.Invoke(this, EventArgs.Empty);

            return true;

        }

        public bool Remove(TKey key)

        {

            int num = Array.BinarySearch(keys, 0, size, key);

            if (num < 0)

            {

                throw new ArgumentException("Element does not exist");

            }

            Console.WriteLine(num);

            var TempArray = new KeyValuePair<TKey, TValue>[size - 1];

            var TempKeys = new TKey[size - 1];

            var TempValues = new TValue[size - 1];

            for (int i = 0; i < Count; i++)

            {

                if (i != num)

                {

                    if (i < num)

                    {

                        TempArray[i] = \_items[i];

                        TempKeys[i] = \_items[i].Key;

                        TempValues[i] = \_items[i].Value;

                    }

                    else

                    {

                        TempArray[i - 1] = \_items[i];

                        TempKeys[i - 1] = \_items[i].Key;

                        TempValues[i - 1] = \_items[i].Value;

                    }

                }

            }

            \_items = TempArray;

            keys = TempKeys;

            values = TempValues;

            size--;

            RemoveElement?.Invoke(this, EventArgs.Empty);

            return true;

        }

        IEnumerator IEnumerable.GetEnumerator()

        {

            return GetEnumerator();

        }

        public IEnumerator<KeyValuePair<TKey, TValue>> GetEnumerator()

        {

            return new NewEnumenator(\_items);

        }

        private class NewEnumenator : IEnumerator<KeyValuePair<TKey, TValue>>

        {

            private readonly KeyValuePair<TKey, TValue>[] \_items;

            private int index;

            private KeyValuePair<TKey, TValue> current;

            public KeyValuePair<TKey, TValue> Current => current;

            object IEnumerator.Current => current;

            public NewEnumenator (KeyValuePair<TKey, TValue>[] items)

            {

                \_items = items;

                index = 0;

                if (\_items.Any())

                    current = items[index];

            }

            public void Dispose()

            {

                return;

            }

            public bool MoveNext()

            {

                if (index >= \_items.Length)

                {

                    return false;

                }

                current = \_items[index];

                index++;

                return true;

            }

            public void Reset()

            {

                index = 0;

                current = \_items[index];

            }

        }

    }

}

**Результати виконання**

Висновок

На даній лабораторній роботі я спроєктував та реалізував клас узагальненої колекції, використовуючи стандартні інтерфейси System.Collections та System.Collections.Generic.