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 №2 з дисципліни

«Сучасні технології розробки WEB-застосувань на платформі Microsoft .NET»

“Модульне тестування. Ознайомлення з засобами та практиками модульного тестування”

Виконав студент **ІП-14 Щербацький Антон**

Перевірив **Бардін Владислав**

Лабораторна робота 2

Модульне тестування. Ознайомлення з засобами та практиками модульного тестування

Мета лабораторної роботи – навчитися створювати модульні тести для вихідного коду розроблювального програмного забезпечення.

Варіант: 6.

Код програми

```
using System.Collections;

namespace EventDictionaryLib;

public class EventDictionary<TKey, TValue> : IDictionary<TKey, TValue>
{
    private Bucket<TKey, TValue>?[] _buckets;

    public ICollection<TKey> Keys => ExtractItems(kvp => kvp.Key).ToList();
    public ICollection<TValue> Values => ExtractItems(kvp =>
kvp.Value).ToList();

    private int _bucketsCount;
    public int Count { get; private set; }
    public bool IsReadOnly => false;

    private const double _loadFactor = 0.75;
    private const int _resizeFactor = 2;
    private const int _initialCapacity = 10;
    public int Capacity { get; private set; }

    public event Action<KeyValuePair<TKey, TValue>>? OnAdd;
    public event Action<TKey>? OnRemove;
    public event Action<TKey, TValue, TValue>? OnUpdate;

    public EventDictionary()
    {
        Capacity = _initialCapacity;
        _bucketsCount = 0;
        Count = 0;
        _buckets = new Bucket<TKey, TValue>[Capacity];
    }

    public EventDictionary(IDictionary<TKey, TValue> dictionary) : this()
    {
        foreach (var item in dictionary)
        {
            Add(item);
        }
    }

    public IEnumerator<KeyValuePair<TKey, TValue>> GetEnumerator()
    {
        foreach (var bucket in _buckets)
        {
```

```
            if (bucket != null)
            {
                foreach (var item in bucket)
                {
                    yield return item;
                }
            }
        }
    }

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

    public void Add(KeyValuePair<TKey, TValue> item)
    {
        ExceptionHelper.CheckNull(item, "Added item cannot be null");

        if (ContainsKey(item.Key))
        {
            ExceptionHelper.ThrowExistsKey(item.Key);
        }

        var bucketIndex = GetIndex(item.Key);

        if ((double) _bucketsCount / Capacity > _loadFactor)
        {
            Resize();
        }

        CreateBucketIfNull(bucketIndex);

        _buckets[bucketIndex]!.Add(item);
        Count++;

        OnAdd?.Invoke(item);
    }

    public void Clear()
    {
        Capacity = _initialCapacity;
        _buckets = new Bucket<TKey, TValue>[Capacity];
        Count = _bucketsCount = 0;
    }

    public bool Contains(KeyValuePair<TKey, TValue> item)
    {
        ExceptionHelper.CheckNull(item, "Item cannot be null");
        var bucketIndex = GetIndex(item.Key);

        return _buckets[bucketIndex] != null &&
_buckets[bucketIndex]!.Contains(item.Key);
    }

    public void CopyTo(KeyValuePair<TKey, TValue>[] array, int arrayIndex)
    {
        ExceptionHelper.CheckNull(array, "Destination array cannot be null");
        ExceptionHelper.CheckRange(arrayIndex, 0, array.Length, "Index is out of
range");
        ExceptionHelper.CheckRange(Count, 0, array.Length - arrayIndex, "The
destination array has insufficient space");

        var currentIndex = arrayIndex;
        var extractedItems = ExtractItems(item => item);
```

```
        foreach (var kvp in extractedItems)
        {
            array[currentIndex] = kvp;
            currentIndex++;
        }
    }

    public bool Remove(KeyValuePair<TKey, TValue> item)
    {
        ExceptionHelper.CheckNull(item, $"Removed item cannot be null");
        return Remove(item.Key);
    }

    public void Add(TKey key, TValue value)
    {
        ExceptionHelper.CheckNull(key, $"Key to add cannot be null");
        ExceptionHelper.CheckNull(value, $"Added value cannot be null");
        Add(new KeyValuePair<TKey, TValue>(key, value));
    }

    public bool ContainsKey(TKey key)
    {
        ExceptionHelper.CheckNull(key, "Key to search cannot be null");
        var bucketIndex = GetIndex(key);
        return _buckets[bucketIndex] != null &&
_buckets[bucketIndex]!.Contains(key);
    }

    public void Update(TKey key, TValue value)
    {
        ExceptionHelper.CheckNull(key, "Key cannot be null");

        var bucketIndex = GetIndex(key);

        if (_buckets[bucketIndex] == null ||
!_buckets[bucketIndex]!.Contains(key))
        {
            ExceptionHelper.ThrowNotFoundKey(key);
        }

        var oldValue = _buckets[bucketIndex]!.Get(key);
        _buckets[bucketIndex]!.Update(key, value);
        OnUpdate?.Invoke(key, oldValue, value);
    }

    public bool Remove(TKey key)
    {
        ExceptionHelper.CheckNull(key, "Removed key cannot be null");

        var bucketIndex = GetIndex(key);
        var currentBucket = _buckets[bucketIndex];

        if (currentBucket == null || !currentBucket.Contains(key))
        {
            return false;
        }

        currentBucket.Remove(key);

        if (currentBucket.Length == 0)
        {
            _buckets[bucketIndex] = null;
            _bucketsCount--;
        }

        Count--;
```

```
        OnRemove?.Invoke(key);

        return true;
    }

    public bool TryGetValue(TKey key, out TValue value)
    {
        ExceptionHelper.CheckNull(key, $"Searched key cannot be null");
        var bucketIndex = GetIndex(key);

        if (!ContainsKey(key))
        {
            value = default;
            return false;
        }

        value = _buckets[bucketIndex]!.Get(key);
        return true;
    }

    public TValue this[TKey key]
    {
        get
        {
            var index = GetIndex(key);

            if (_buckets[index] == null || !_buckets[index]!.Contains(key))
            {
                throw new KeyNotFoundException($"Key: {key} does not exists");
            }

            return _buckets[index]!.FirstOrDefault(item =>
key!.Equals(item.Key)).Value;
        }
        set
        {
            var bucketIndex = GetIndex(key);

            if(_buckets[bucketIndex] != null &&
_buckets[bucketIndex]!.Contains(key))
            {
                Update(key, value);
            }
            else
            {
                Add(key, value);
            }
        }
    }

    private void CreateBucketIfNull(int bucketIndex)
    {
        _buckets[bucketIndex] ??= new Bucket<TKey, TValue>();
        _bucketsCount++;
    }

    private IEnumerable<T> ExtractItems<T>(Func<KeyValuePair<TKey, TValue>, T>
selector)
    {
        return _buckets
            .Where(bucket => bucket != null)
            .SelectMany(bucket => bucket!
                .Select(selector));
    }

    private int GetIndex(TKey key)
```

```
    {
        ExceptionHelper.CheckNull(key, $"Key cannot be null");

        return Math.Abs(key!.GetHashCode() % Capacity);
    }

    private void Resize()
    {
        var extracted = ExtractItems(item => item);
        Capacity *= _resizeFactor;
        _buckets = new Bucket<TKey, TValue>[Capacity];

        Rehash(extracted);
    }

    private void Rehash(IEnumerable<KeyValuePair<TKey, TValue>> extractedItems)
    {
        foreach (var item in extractedItems)
        {
            var bucketIndex = GetIndex(item.Key);
            CreateBucketIfNull(bucketIndex);
            _buckets[bucketIndex]!.Add(item);
        }
    }
}
using System.Collections;
using System.Runtime.CompilerServices;
using EventDictionaryLib;
namespace Dictionary.Tests;

public class BucketTests
{
    [Fact]
    public void Length_Returns_Zero_If_Empty_Collection()
    {
        // Arrange
        var bucket = new Bucket<string, int>();

        // Act
        int length = bucket.Length;

        // Assert
        Assert.Equal(0, length);
    }

    [Fact]
    public void Add_IncreasesArrayLength_ByOne()
    {
        // Arrange
        var bucket = new Bucket<string, int>();
        int initialLength = bucket.Length;

        // Act
        var item = new KeyValuePair<string, int>("1", 1);
        bucket.Add(item);

        // Assert
        Assert.Equal(initialLength + 1, bucket.Length);
    }

    [Fact]
    public void Bucket_Contains_AddedItem()
    {
        // Arrange
        var bucket = new Bucket<string, int>();
        var item1 = new KeyValuePair<string, int>("1", 1);
```

```
        // Act
        bucket.Add(item1);

        // Assert
        Assert.Equal(item1, bucket._items[^1]);
    }

    [Fact]
    public void Remove_Item_WhenPresent_MustRemoveKey()
    {
        var bucket = new Bucket<string, int>();
        var key = "1";
        var item = new KeyValuePair<string, int>(key, 1);

        bucket.Add(item);
        bucket.Remove(key);

        Assert.False(bucket.Contains(key));
    }

    [Fact]
    public void Remove_Item_Must_Decrease_ItemsCount()
    {
        var bucket = new Bucket<string, int>();
        var item1 = new KeyValuePair<string, int>("1", 1);
        var item2 = new KeyValuePair<string, int>("2", 2);

        bucket.Add(item1);
        bucket.Add(item2);

        bucket.Remove(item1.Key);
        bucket.Remove(item2.Key);

        Assert.Equal(0, bucket.Length);
    }

    [Fact]
    public void Update_Existing_Key_Should_Update_Value()
    {
        var bucket = new Bucket<string, int>();
        var item = new KeyValuePair<string, int>("1", 1);
        var newValue = 2;

        bucket.Add(item);

        bucket.Update("1", newValue);

        Assert.Equal(newValue, bucket.Get(item.Key));
    }

    [Fact]
    public void Update_Existing_Key_Should_Not_Add_New_Item()
    {
        var bucket = new Bucket<string, int>();
        var item = new KeyValuePair<string, int>("1", 1);
        var newValue = 2;
        var initialCount = bucket.Length;

        bucket.Update(item.Key, newValue);

        Assert.Equal(initialCount, bucket.Length);
    }

    [Fact]
    public void Update_NonExisting_Key_Should_Not_Add_New_Item()
```

```
    {
        var bucket = new Bucket<string, int>();
        var item = new KeyValuePair<string, int>("1", 1);

        bucket.Add(item);

        bucket.Update("2", 2);

        Assert.Equal(item.Value, bucket.Get(item.Key));
    }

    [Fact]
    public void If_Key_Exists_Then_Bucket_Must_Contain_Key()
    {
        var bucket = new Bucket<string, int>();
        var item = new KeyValuePair<string, int>("1", 1);

        bucket.Add(item);

        Assert.True(bucket.Contains(item.Key));
    }

    [Fact]
    public void Contains_NonExisting_Key_Must_Return_False()
    {
        var bucket = new Bucket<string, int>();
        var item = new KeyValuePair<string, int>("1", 2);

        bucket.Add(item);

        Assert.False(bucket.Contains("2"));
    }

    [Fact]
    public void Get_Existing_Key_Should_Return_Value()
    {
        var bucket = new Bucket<string, int>();
        var item = new KeyValuePair<string, int>("1", 1);

        bucket.Add(item);

        Assert.Equal(item.Value, bucket.Get(item.Key));
    }

    [Fact]
    public void Get_NonExisting_Key_Must_Return_Default_Value()
    {
        var bucket = new Bucket<string, int>();

        var actual = bucket.Get("non existed");

        Assert.Equal(default, actual);
    }

    [Fact]
    public void GetEnumerator_Must_Return_Expected_Items()
    {
        var items = new List<KeyValuePair<string, int>>()
        {
            new KeyValuePair<string, int>("1", 1),
            new KeyValuePair<string, int>("2", 2),
            new KeyValuePair<string, int>("3", 3)
        };

        var bucket = new Bucket<string, int>()
        {
```

```
            items[0],
            items[1],
            items[2]
        };

        using var enumerator = bucket.GetEnumerator();
        var enumerated = new List<KeyValuePair<string, int>>();

        while (enumerator.MoveNext())
        {
            enumerated.Add(enumerator.Current);
        }

        Assert.Equal(items, enumerated);
    }

    [Fact]
    public void GetEnumerator_Empty_Collection_Must_Return_No_Items()
    {
        var bucket = new Bucket<string, int>();
        using var enumerator = bucket.GetEnumerator();

        Assert.False(enumerator.MoveNext());
    }

    [Fact]
    public void IEnumerable_GetEnumerator_Must_Return_Expected_Items()
    {
        var items = new List<KeyValuePair<string, int>>()
        {
            new KeyValuePair<string, int>("1", 1),
            new KeyValuePair<string, int>("2", 2),
            new KeyValuePair<string, int>("3", 3)
        };

        var bucket = new Bucket<string, int>()
        {
            items[0],
            items[1],
            items[2]
        };

        var enumerator = ((IEnumerable)bucket).GetEnumerator();
        var enumerated = new List<KeyValuePair<string, int>>();

        while (enumerator.MoveNext())
        {
            enumerated.Add((KeyValuePair<string, int>)enumerator.Current);
        }

        Assert.Equal(items, enumerated);
    }

    [Fact]
    public void
IEnumerable_GetEnumerator_Empty_Collection_Must_Return_No_Items()
    {
        var bucket = new Bucket<string, int>();
        var enumerator = ((IEnumerable)bucket).GetEnumerator();

        Assert.False(enumerator.MoveNext());
    }
}
using EventDictionaryLib;

namespace Dictionary.Tests;
```

```
public class ExceptionHelperTests
{
    [Fact]
    public void Throw_ArgumentNullException_If_Item_Null()
    {
        string item = null;

        var exception = Record.Exception(() => ExceptionHelper.CheckNull(item,
string.Empty));

        Assert.IsType<ArgumentNullException>(exception);
    }

    [Fact]
    public void Not_Throw_ArgumentNullException_If_Item_Not_Null()
    {
        string item = "_";

        var exception = Record.Exception(() => ExceptionHelper.CheckNull(item,
string.Empty));

        Assert.Null(exception);
    }

    [Fact]
    public void Throw_KeyNotFoundException_With_Any_Key()
    {
        var key = 1;

        var exception = Record.Exception(() =>
ExceptionHelper.ThrowNotFoundKey(key));

        Assert.IsType<KeyNotFoundException>(exception);
    }

    [Fact]
    public void Throw_ApplicationException_If_Any_Key_Exists()
    {
        var key = 1;

        var exception = Record.Exception(() =>
ExceptionHelper.ThrowExistsKey(key));

        Assert.IsType<ApplicationException>(exception);
    }

    [Fact]
    public void Throw_ArgumentOutOfRangeException_If_Index_OutOfRange()
    {
        var max = 5;
        var min = 0;
        var index = 100;

        var exception = Record.Exception(() => ExceptionHelper.CheckRange(index,
min, max, string.Empty));

        Assert.IsType<ArgumentOutOfRangeException>(exception);
    }

    [Fact]
    public void Not_Throw_OutOfRangeException_If_Index_Is_Not_OutOfRange()
    {
        var max = 5;
        var min = 0;
        var index = 3;
```

```
        var exception = Record.Exception(() => ExceptionHelper.CheckRange(index,
min, max, string.Empty));

        Assert.Null(exception);
    }
}
```

Висновок

На даній лабораторній роботі я навчився використовувати юніт тести для визначення правильності роботи коду.

У результаті виконання лабораторної роботи було написано тести для трьох класів підтримки роботи словника: EventDictionary.Tests, Bucket.Tests, ExceptionHelper.Tests.