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»

«Проєктування і реалізація колекції даних»

Виконав студент **ІП-14 Щербацький Антон**

Перевірив **Бардін В.**

Лабораторна робота 1

Узагальнені типи (Generic) з підтримкою подій. Колекції

Мета лабораторної роботи – навчитися проектувати та реалізовувати узагальнені типи, а також типи з підтримкою подій.

Варіант: 6.

Завдання:

6	Словник	Див. Dictionary<TKey, TValue>	Збереження даних за допомогою динамічно зв'язаного списку або вектору

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.

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 { get;  set; }
    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; 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 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");

        var bucketIndex = GetIndex(item.Key);

        if (ContainsKey(item.Key))
        {
            ExceptionHelper.ThrowExistsKey(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]
            .Any(el => el.Key.Equals(item.Key)
                       && el.Value.Equals(item.Value));
    }

    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)
    {
        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(_buckets, extracted);
    }

    private void Rehash(Bucket<TKey, TValue>[] buckets,
IEnumerable<KeyValuePair<TKey, TValue>> extractedItems)
    {
        foreach (var item in extractedItems)
        {
            var bucketIndex = GetIndex(item.Key);
            CreateBucketIfNull(bucketIndex);
            _buckets[bucketIndex].Add(item);
        }
```

```
    }
}
using System.Collections;

namespace EventDictionaryLib;

internal class Bucket<TKey, TValue> : IEnumerable<KeyValuePair<TKey, TValue>>
{
   private KeyValuePair<TKey, TValue>[] _items;
   public int Length => _items.Length;

   public Bucket()
   {
      _items = Array.Empty<KeyValuePair<TKey, TValue>>();
   }

   public void Add(KeyValuePair<TKey, TValue> item)
   {
      Array.Resize(ref _items, _items.Length + 1);
      _items[^1] = item;
   }

   public void Remove(TKey key)
   {
      var removedCount = 0;

      for (int i = 0; i < _items.Length; i++)
      {
         if (_items[i].Key.Equals(key))
         {
            removedCount++;
         }
         else if (removedCount > 0)
         {
            _items[i - removedCount] = _items[i];
         }
      }

      if (removedCount > 0)
      {
         Array.Resize(ref _items, _items.Length - removedCount);
      }
   }

   public void Update(TKey key, TValue value)
   {
      for (var i = 0; i < _items.Length; i++)
      {
         if (_items[i].Key.Equals(key))
         {
            _items[i] = new KeyValuePair<TKey, TValue>(key, value);
            return;
         }
      }
   }

   public bool Contains(TKey key)
   {
      return _items.Any(item => item.Key.Equals(key));
   }

   public TValue Get(TKey key)
   {
      return _items.FirstOrDefault(item => item.Key.Equals(key)).Value;
   }
```

```
    public IEnumerator<KeyValuePair<TKey, TValue>> GetEnumerator()
    {
        return ((IEnumerable<KeyValuePair<TKey, TValue>>)_items).GetEnumerator();
    }

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

public static class ExceptionHelper
{
    public static void CheckNull<T>(T? item, string message)
    {
        if (item == null)
        {
            throw new ArgumentNullException(message);
        }
    }

    public static void CheckRange(int index, int min, int max, string message)
    {
        if (index < min || index > max)
        {
            throw new ArgumentOutOfRangeException(message);
        }
    }

    public static void ThrowNotFoundKey<T>(T key)
    {
        throw new KeyNotFoundException($"Key: {key} does not exists");
    }

    public static void ThrowExistsKey<T>(T key)
    {
        throw new ApplicationException($"Key: {key} is already exists");
    }
}
namespace MyDictionary;

public static class ExtendedConsole
{
    public static void WriteLine(string line, ConsoleColor backgroundColor =
ConsoleColor.Black,
        ConsoleColor foregroundColor = ConsoleColor.White)
    {
        Console.BackgroundColor = backgroundColor;
        Console.ForegroundColor = foregroundColor;

        Console.WriteLine(line);

        Console.ResetColor();
    }
}
using System.Diagnostics;
using EventDictionaryLib;

namespace MyDictionary;

public class Program
{
    static void Main()
```

```
    {
        var randomizer = new Random();
        var dict = new EventDictionary<string, int>();
        var itemsCount = 25;

        dict.OnAdd += item =>
            ExtendedConsole.WriteLine($"Added item: {item.Value} with key:
{item.Key}. Current elements count: {dict.Count}",
                foregroundColor: ConsoleColor.Green);

        dict.OnRemove += key =>
            ExtendedConsole.WriteLine($"Item with key: {key} removed. Current
elements count: {dict.Count}",
                foregroundColor: ConsoleColor.Red);

        dict.OnUpdate += (key, oldValue, newValue) =>
            ExtendedConsole.WriteLine($"Element with key: {key}, old value:
{oldValue} updated to {newValue}",
                foregroundColor: ConsoleColor.Yellow);

        Console.WriteLine(new string('-', 30) + "ADD TEST" + new string('-', 30));
        TestAdd(dict, itemsCount);

        Console.WriteLine("\n" + new string('-', 30) + "REMOVE TEST" + new
string('-', 30));
        TestRemove(itemsCount, dict);

        Console.WriteLine("\n" + new string('-', 30) + "UPDATE TEST" + new
string('-', 30));
        TestUpdate(itemsCount, dict);

        Console.WriteLine("\n" + new string('-', 30) + "GET TEST" + new string('-
', 30));
        TestGet(itemsCount, dict);

        Console.WriteLine("\n" + new string('-', 30) + "Dictionary" + new
string('-', 30));
        Print(dict);

        Console.ReadKey();
    }

    private static void Print<TKey, TValue>(IDictionary<TKey, TValue> dictionary)
    {
        foreach (var item in dictionary)
        {
            Console.WriteLine($"Item key: {item.Key}, value: {item.Value}");
        }
    }

    private static void TestGet(int itemsCount, EventDictionary<string, int> dict)
    {
        var randomizer = new Random();

        for (int i = 0; i < 10; i++)
        {
            var randomKey = randomizer.Next(itemsCount + 10).ToString();
            int? value = null;

            try
            {
                value = dict[randomKey];
            }
            catch (KeyNotFoundException e)
            {
```

```
                ExtendedConsole.WriteLine(e.Message, backgroundColor:
ConsoleColor.Red);
            }
            finally
            {
                if (value != null)
                {
                    ExtendedConsole.WriteLine($"Get value: {value} with key:
{randomKey}",
                        foregroundColor: ConsoleColor.Green);
                }
            }
        }
    }

    private static void TestUpdate(int itemsCount, EventDictionary<string, int>
dict)
    {
        var randomizer = new Random();

        for (int i = 0; i < 10; i++)
        {
            var randomKey = randomizer.Next(itemsCount + 10).ToString();

            try
            {
                dict.Update(randomKey, 999);
            }
            catch (KeyNotFoundException e)
            {
                ExtendedConsole.WriteLine(e.Message, backgroundColor:
ConsoleColor.Red);
            }
        }
    }

    private static void TestRemove(int itemsCount, EventDictionary<string, int>
dict)
    {
        var randomizer = new Random();

        for (int i = 0; i < 10; i++)
        {
            var randomKey = randomizer.Next(itemsCount + 10).ToString();
            var removed = dict.Remove(randomKey);

            if (!removed)
            {
                ExtendedConsole.WriteLine($"Dictionary does not contain key:
{randomKey}",
                    backgroundColor: ConsoleColor.Red);
            }
        }
    }

    private static void TestAdd(EventDictionary<string, int> dict, int itemsCount)
    {
        try
        {
            Fill(dict, itemsCount);

            dict.Add("2", 222);
        }
        catch (ApplicationException e)
        {
```

```
            ExtendedConsole.WriteLine(e.Message, backgroundColor:
ConsoleColor.Red);
        }
    }

    static void Fill(EventDictionary<string, int> dictionary, int itemsCount)
    {
        for (int i = 0; i < itemsCount; i++)
        {
            var item = i + 1;
            dictionary[(i + 1).ToString()] = item;
        }
    }
}
```

Виконання програми

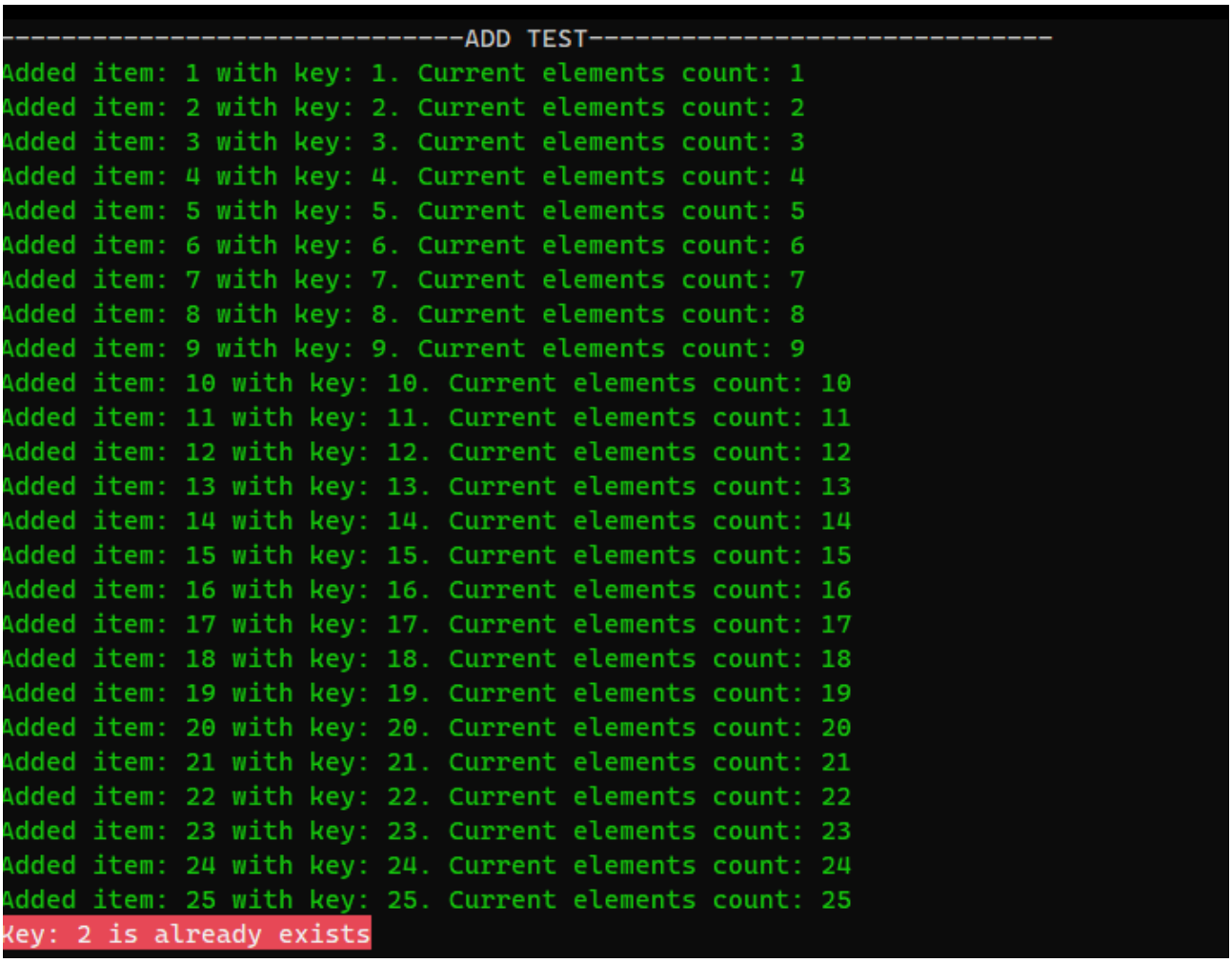

Рис 1 – Початкове додавання елементів у словник

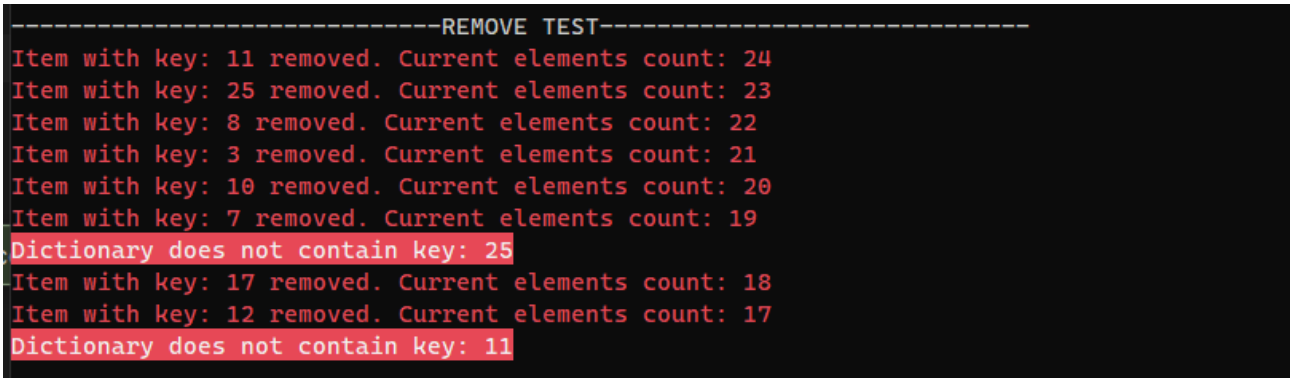

Рис 2 – Видалення елементів за випадково згенерованими ключами

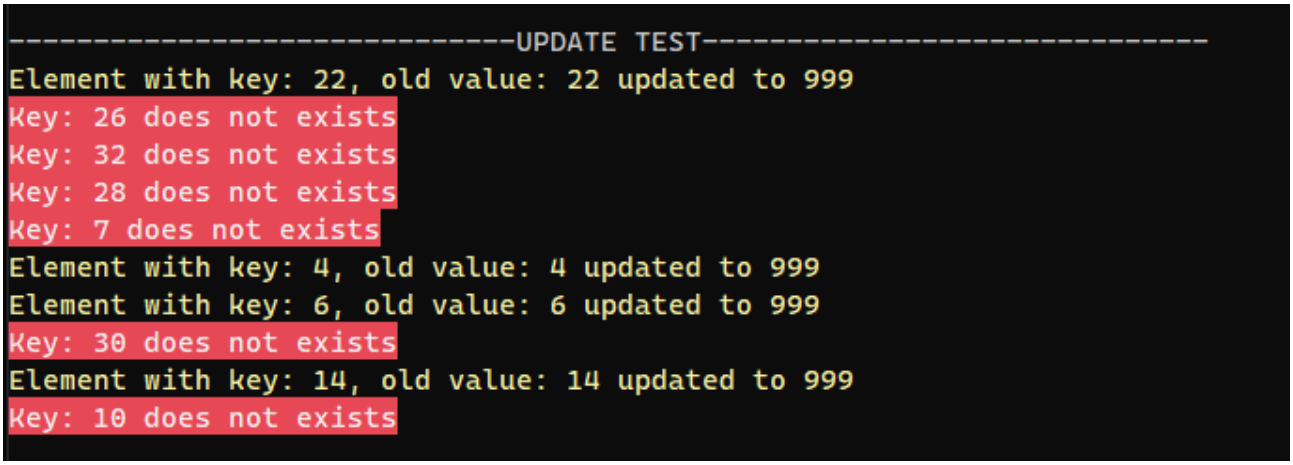

Рис 3 – Оновлення елементів за випадково згенерованими ключами

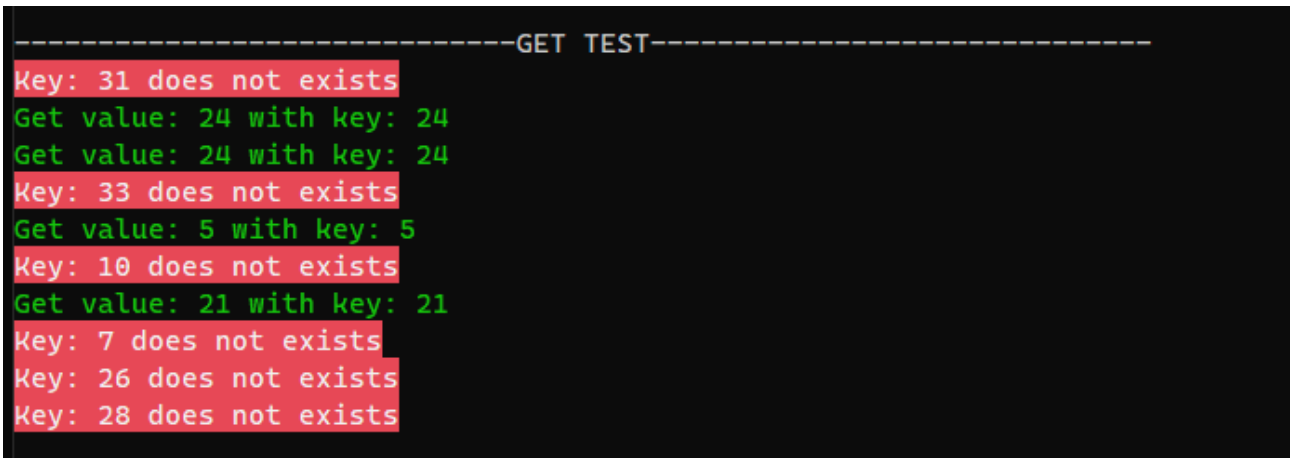

Рис 4 – Отримання елементів за випадково згенерованими ключами

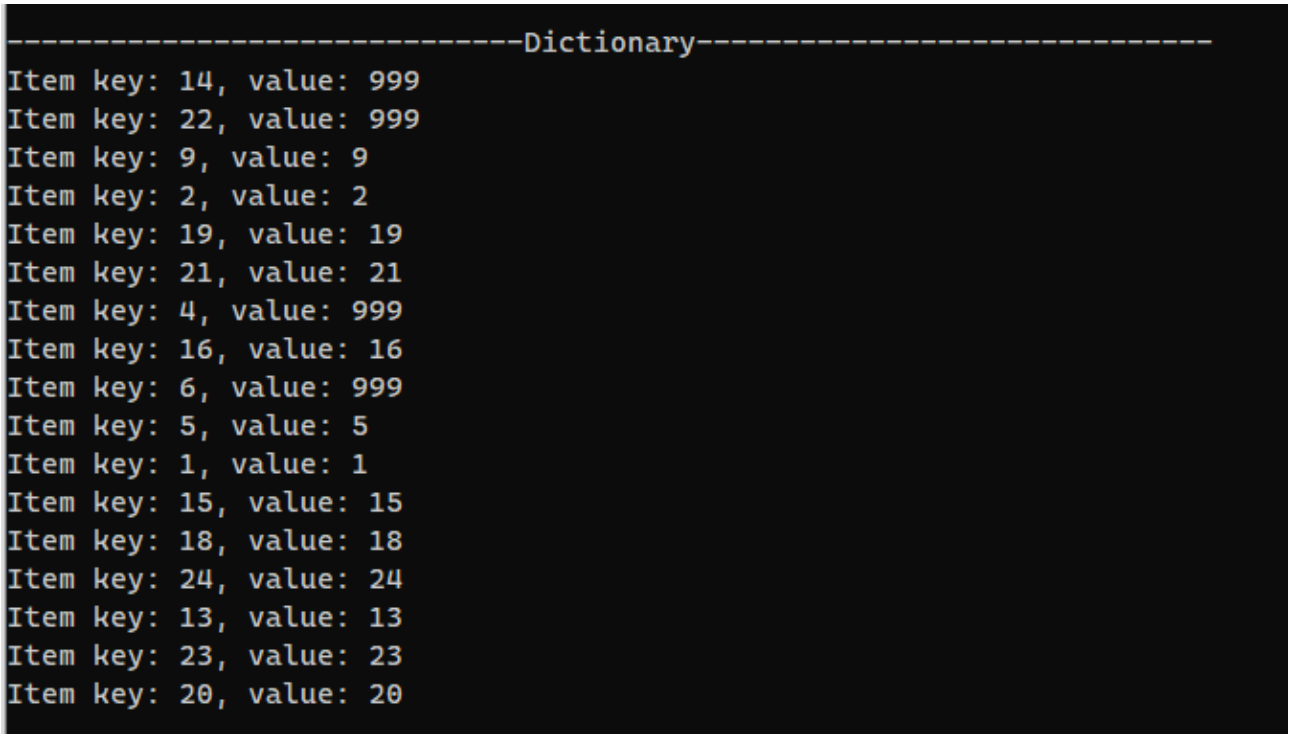

Рис 5 – Словник після внесення всіх змін

Висновок

На даній лабораторній роботі я навчився використовувати інтерфейси колекцій та спроектував власну колекцію – словник.

Колекція підтримує підписки на події: при додаванні елементу, при видаленні елементу, при оновленні елементу.

У результаті виконання роботи було розроблено консольний застосунок для виконання команд над словником.