В.В. Подбельский

Использованы иллюстрации пособия Daniel Solis, Illustrated C#

Иллюстрации к курсу лекций по дисциплине «Программирование на C#»

Модуль 3. Лекция 2b

Обзор Коллекций в .NET

Благодарность Д.Д. Сагалову за помощь в переработке презентаций

Упаковка и Распаковка

Упаковка (boxing) – операция приведения типа значения к ссылочному типу (object или интерфейсу, реализуемому данным типом).

Упаковка приводит к оборачиванию значения внутри экземпляра Object и, как следствие, его сохранению на куче.
Упаковка не может приводить к ошибкам этапа выполнения, поэтому она всегда **является неявным преобразованием типов**.

Распаковка – действие, обратное упаковке, позволяющее скопировать упакованное значение в переменную типа значения.

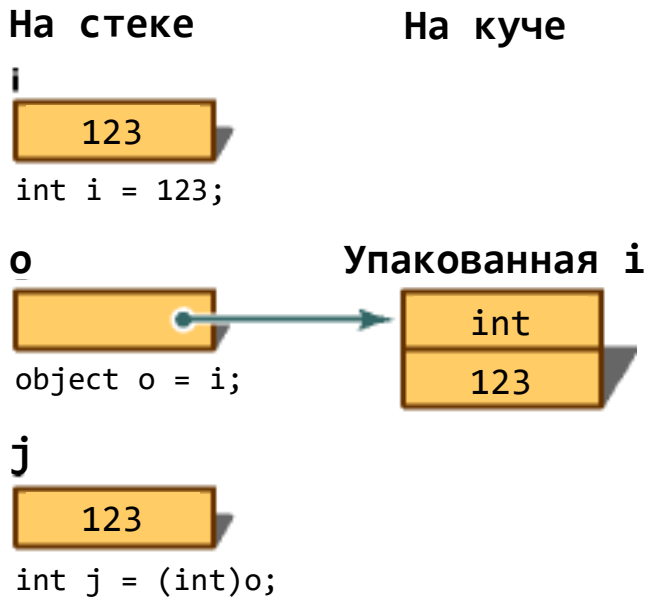
Упаковка не гарантирует успешность преобразования, в случае его недопустимости возникает **InvalidCastException**.

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/boxing-and-unboxing

Особенности Распаковки

Распаковка осуществляется путём применения 3 шагов:

1) Проверка инвариантности (точного совпадения) типов;
2) Проверка на null, при попытке распаковать null **NullReferenceException** возникает всегда;
3) Копирование значения из кучи в указанную переменную при условии успешного прохождения двух прошлых шагов.



Важно: тип, к которому приводится упакованный объект на шаге 1) должен строго совпадать с изначальным типом упакованного значения, иначе возникнет **InvalidCastException**. Иными словами, приведения типов при распаковке не учитываются.

Недостатки Использования Упаковки и Распаковки

Упаковка и распаковка – **вычислительно дорогие операции**, что связано с:

- Необходимостью перемещения сущностей между стеком и кучей, приводящей к копированию данных и динамическому выделению памяти в процессе;
- Дополнительными проверками на соответствие типов.

Согласно документации Microsoft:

- Упаковка может быть до 20 раз медленнее обычного присваивания для ссылочных типов;
- Распаковка может быть вплоть до 4 раз медленнее присваивания.

Ссылка на документацию по производительности:
https://docs.microsoft.com/en-us/dotnet/framework/performance/performance-tips

Примеры: Упаковка и Распаковка

```
int int1 = 4;
// Значение int1 упаковывается как int:
object boxingContainer1 = int1;
// Значение int1 упаковывается как byte:
object boxingContainer2 = (byte)int1;

int correctUnboxedInt = (int)boxingContainer1 + 228;
byte correctUnboxedByte = (byte) boxingContainer2;

// Исключение – попытка распаковать int как short:
short incorrectUnboxing1 = (short)boxingContainer1;
// Исключение – попытка распаковать int как byte:
int incorrectUnboxing2 = 5 + (byte)boxingContainer1;
// Исключение – попытка распаковать byte как int:
int incorrectUnboxing3 = (int)boxingContainer2;
```

Абстрактные Типы Данных. Коллекции

Абстрактный тип данных (АТД) – модель типов данных. АТД определяется семантикой (поведением) *с точки зрения пользователя* и задаёт набор возможных операций и поведение этих операций.

Как правило, большое число языков программирования предоставляют реализацию различных АТД как часть стандартной библиотеки. В .Net реализации АТД выделены в отдельные пространства имён BCL.

Другим довольно распространённым понятиям являются **коллекции** – типы данных, определяющий логику хранения внутри себя набора значений одного или нескольких типов по определённым правилам.
Фактически, коллекции являются реализациями АТД в языках программирования.

Основные Пространства Имён Коллекций C#

Пространство Имён	Краткое описание
System.Collections	Необобщённые классы и интерфейсы коллекций.
System.Collections.Generic	Обобщённые классы и интерфейсы коллекций.
System.Collections.Immutable	Классы и интерфейсы неизменяемых коллекций.
System.Collections.Concurrent	Потокобезопасные (за счёт синхронизации) коллекции.
System.Collections.ObjectModel	Коллекции, предназначаемые для использования в качестве возвращаемых значений методов/свойств. Часть коллекций данного пространства имён определяет события на добавление/изменение коллекции.
System.Collections.Specialized	Строго типизированные необобщённые типы коллекций.

Коллекции из System.Collections

До появления обобщений (C# 2) все коллекции хранили элементы по ссылкам типа Object.

Так, все коллекции пространства имён **System.Collections** являются не типизированными, что ведёт к проблемам:

- Отсутствие проверки типов позволяет добавлять объекты произвольного типа (не соответствующие ожиданиям);
- При хранении типов значений возникают заметные накладные расходы за счёт операций упаковки-распаковки;
- При извлечении элементов из необобщённых коллекций возникает постоянная необходимость в приведении типов от Object к целевым.

Поэтому рекомендуется использовать альтернативы из пространства имён **System.Collections.Generic**.

Необобщённые Коллекции – System.Collections

Класс	Краткое описание
ArrayList	Динамический массив с элементами типа Object.
Queue	Очередь.
Stack	Стек.
Hashtable	Хэш-таблица с парами ключ-значение.
SortedList	Коллекция упорядоченных пар ключ-значение.

Замена на Обобщённые Коллекции

Ниже представлен список обобщённых коллекций, соответствующих необобщённым:

Коллекция	Замена	Коллекция	Замена
ArrayList	List<T>	DictionaryEntry	KeyValuePair<TKey, TValue>
CaseInsensitiveComparer	StringComparer. OrdinalIgnoreCase	Hashtable	Dictionary<TKey, TValue>
CaseInsensitiveHashCodeProvider	StringComparer. OrdinalIgnoreCase	Queue	Queue<T>
CollectionBase	Collection<T>	ReadOnlyCollectionBase	ReadOnlyCollection<T>
Comparer	Comparer<T>	SortedList	SortedList<TKey, TValue>

Динамический Массив

Проблема: обычные массивы позволяют хранить фиксированное количество элементов, что бывает неудобным, когда реальный размер массива часто меняется динамически.

Динамический массив представляет собой структуру данных, содержащую внутри массив, автоматически расширяющий свою ёмкость по мере необходимости.
На заметку: расширение реализуется с использованием Array.Resize() для хранимого внутри массива.

Сложность операций:

- Вставка в конец/удаление из конца: **O(1)** или **O(N)** (*в случае изменения размера массива*)
- Доступ по индексу: **O(1)**
- Поиск элемента по значению: **O(N)**
- Вставка в произвольное место/удаление из произвольного места: **O(N)**
- Поиск в отсортированном: **O(log N)** (*с помощью бинарного поиска*)

Исходный код реализации в BCL: ArrayList, List<T>

Конструкторы Класса ArrayList

Конструктор	Краткое описание
ArrayList()	Создаёт пустой экземпляр класса с ёмкостью по умолчанию.
ArrayList(ICollection)	Создаёт экземпляр класса, копируя в него элементы данной коллекции. Ёмкость равна количеству скопированных элементов.
ArrayList(int)	Создаёт пустой экземпляр класса с заданной начальной ёмкостью.

ArrayList. Пример 1

```
using System;
using System.Collections;
class Program
{
    static void Main()
    {
        ArrayList mix = new ArrayList();
        Console.WriteLine($"mix.Capacity = {mix.Capacity}");
        mix.Add(13);
        Console.WriteLine($"mix.Capacity = {mix.Capacity}");
        mix.Add("thirteen");
        Console.WriteLine($"mix.Count = {mix.Count}");
        Console.WriteLine($"mix.Capacity = {mix.Capacity}");
        foreach (object obj in mix)
        {
            Console.Write(obj + "\t");
        }
    }
}
```

Вывод:

mix.Capacity = 0
mix.Capacity = 4
mix.Count = 2
mix.Capacity = 4
13 thirteen

ArrayList. Пример 2

```
using System;
using System.Collections;

ArrayList roll = new ArrayList();
roll.AddRange(new object[] { 0, "zero", 1, "one", 2, "two" });
for (int i = 0; i < roll.Count; i++)
{
    if (roll[i] is int)
        roll[i] = (int)roll[i] * 2;
}
foreach (object ob in roll) Console.Write(ob + "  ");
Console.WriteLine();

for (int i = 0; i < roll.Count; i++)
{
    if (roll[i] is string)
        roll.Remove(roll[i]);
}
roll.Reverse();

foreach (object obj in roll) Console.Write(obj + "  ");
```

Вывод:

0 zero 2 one 4 two

4 2 0

Класс List<T>

List<T> – обобщённая реализация динамического массива.

Для получения информации о размере/ёмкости определены свойства:

- Count – свойство только для чтения, возвращает реальное количество элементов внутри;
- Capacity – возвращает или задаёт общее число элементов, которые может вместить внутренний массив. Если значение задаётся, то размер массива сразу же будет изменён;

Метод TrimExcess() – задаёт ёмкость, равную фактическому числу элементов, если это число меньше порогового значения (90% - т. е. если неиспользуемая ёмкость < 10%, то размер не изменится).

Основные Методы List<T>. Часть 1

Метод	Краткое описание
Add	Добавляет элемент в конец списка.
AddRange	Добавляет диапазон элементов переданной коллекции в конец списка.
BinarySearch	Выполняет бинарный поиск в списке или его диапазоне. Для выполнения операции список должен быть *отсортирован*.
Clear	Удаляет все элементы списка (сложность – O(N), где N = Count)
Contains	Возвращает true, если элемент найден в списке и false в противном случае.
GetRange	Создаёт *поверхностную* копию диапазона элементов указанного списка.
IndexOf	Возвращает индекс *первого* вхождения элемента в списке или -1, если он не найден.
LastIndexOf	Возвращает индекс *последнего* вхождения элемента в списке или -1, если он не найден.

Основные Методы List<T>. Часть 2

Метод	Краткое описание
Insert	Выполняет вставку элемента на заданную позицию, сдвигая другие элементы.
InsertRange	Выполняет вставку диапазона элементов другой коллекции на заданную позицию, сдвигая другие элементы данного списка.
Remove	Удаляет первое вхождение указанного элемента из списка.
RemoveAt	Удаляет элемент по заданному индексу.
RemoveRange	Удаляет указанный диапазон элементов.
Reverse	Меняет порядок элементов всего списка или его диапазона на обратный.
Sort	Выполняет сортировку всего списка или его диапазона.
ToArray	Копирует элементы списка в новый массив и возвращает его.

Пример Фрагмента Реализации List<T>

```
public class MyList<T>
{
    T[] data = new T[4];
    public int Count { get; private set; }
    public int Capacity { get; private set; }

    public T this[int i]
    {
        get => data[i];
        set => data[i] = value;
    }
    public void Add(T value)
    {
        if (Count >= data.Length)
        {
            Capacity = data.Length * 2;
            System.Array.Resize(ref data, Capacity);
        }
        data[Count++] = value;
    }
}
```

Пример: Добавление Элементов и Расширение List<T>

```
using System;
using System.Collections.Generic;

List<int> list = new List<int>();
Console.WriteLine($"Initial value of Capacity: {list.Capacity}");
for (int i = 0, cap = list.Capacity; i < 50; ++i)
{
    list.Add(i);
    if (cap != list.Capacity)
    {
        cap = list.Capacity;
        Console.WriteLine($"i = {i} => new Capacity = {list.Capacity}");
    }
}
```

Вывод:

Initial value of Capacity: 0
i = 0 => new Capacity = 4
i = 4 => new Capacity = 8
i = 8 => new Capacity = 16
i = 16 => new Capacity = 32
i = 32 => new Capacity = 64

Пример 2: List<T>. Сортировка с Делегатом

```
using System;
using System.Collections.Generic;

List<byte> byteRoll = new List<byte>();
byteRoll.Add(12);
byteRoll.Add(255);
byteRoll.Add(33);
Console.WriteLine("byteRoll.Capacity = " + byteRoll.Capacity);
foreach (byte byteValue in byteRoll)
{
    Console.Write(byteValue + "\t");
}
Console.WriteLine();
byteRoll.Sort((lhs, rhs) => -lhs.CompareTo(rhs));
foreach (byte reverseSortedByte in byteRoll)
{
    Console.Write(reverseSortedByte + "\t");
}
```

Аналогично классу Array, List<T> определяет методы, принимающие делегаты.

Вывод:

byteRoll.Capacity = 4
12 255 33
255 33 12

Двусвязный Список

Проблема: при использовании массивов вставка в произвольное место/удаление из произвольного места приводит к сдвигу всех остальных элементов, что может быть нежелательным.

Двусвязный список представляет собой структуру данных из элементов-узлов, каждый из которых ссылается на последующий и предыдущий узлы. Таким образом, вставка и удаление не требуют перемещения всех последующих/предыдущих элементов в памяти.

Сложность операций:

- Вставка в конец/удаление из конца: **O(1)**
  Доступ по индексу: **не предоставляется**, допускается хранение ссылок на узлы
- Поиск элемента по значению: **O(N)**

Подумайте, почему может быть эффективнее хранить элементы непрерывно в динамическом массиве?

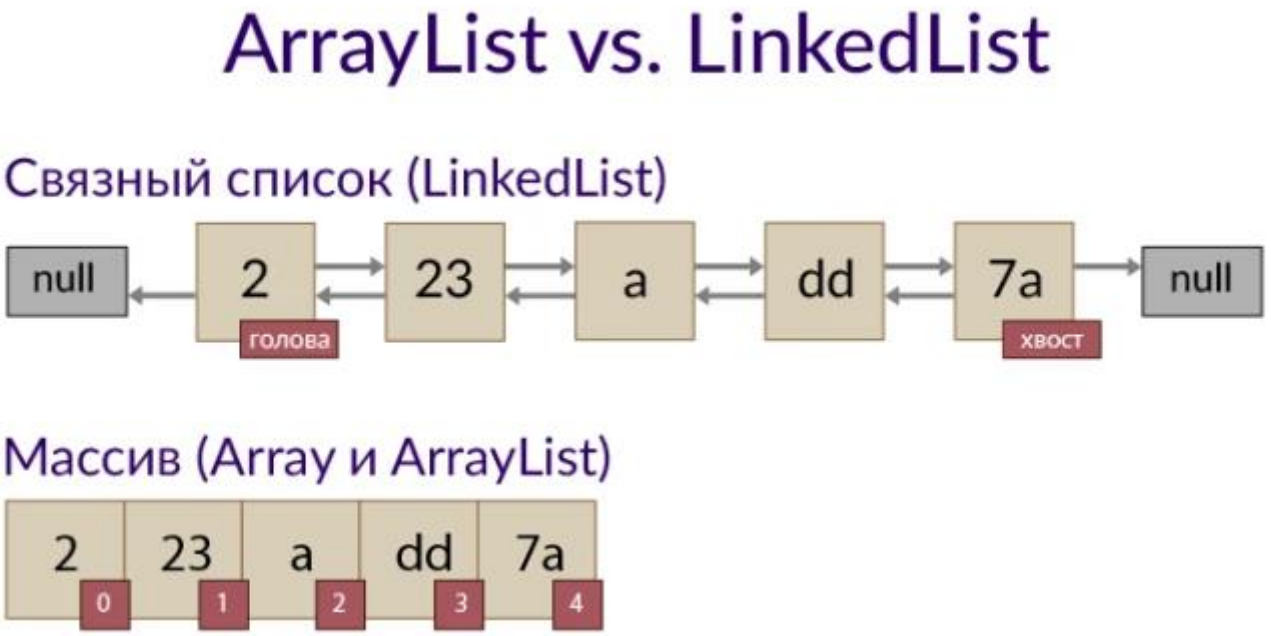
- Вставка в произвольное место/удаление из произвольного места: **O(1)**

Исходный код реализации в BCL: LinkedList<T>

Класс LinkedList<T>

LinkedList<T> – обобщённая реализация двусвязного списка. Узлы представлены в виде объектов типа **LinkedListNode<T>**, связанных друг с другом ссылками.

На заметку: каждый узел закрепляется за своим связным списком. Узел, который принадлежит одному списку, нельзя добавить в другой – возникнет исключение.



Пример Работы с LinkedList<T>

```
using System;
using System.Collections.Generic;

LinkedList<string> tune = new LinkedList<string>();
tune.AddFirst("do");                        // do
tune.AddLast("so");                         // do-so
tune.AddAfter(tune.First, "re");            // do-re-so
tune.AddAfter(tune.First.Next, "mi");       // do-re-mi-so
tune.AddBefore(tune.Last, "fa");            // do-re-mi-fa-so

tune.RemoveFirst();                         // re-mi-fa-so
tune.RemoveLast();                          // re-mi-fa
LinkedListNode<string> miNode = tune.Find("mi");
tune.Remove(miNode);                        // re-fa
tune.AddFirst(miNode);                      // mi-re-fa
foreach (string note in tune)
{
    Console.WriteLine(note);
}
```

Вывод:

mi
re
fa

Очередь

Проблема: в определённых случаях необходимо смоделировать ситуацию, когда *первыми удаляются* те элементы, которые были добавлены *раньше* всех остальных.

Очередь – специальная структура данных, работающая по схеме “First In – First Out” (FIFO). В начале из очереди удаляются элементы, добавленные первыми.
Последний элемент очереди называют **хвостом** (tail), а первый – **головой** (head).

На заметку: очередь может реализовываться на основе других структур данных, можно встретить её упоминание как *адаптера* для другой коллекции.

Операция *добавления* применяется к хвосту очереди, операции *удаления* и *чтения* – к голове.

Исходный код реализации в BCL: Queue, Queue<T>

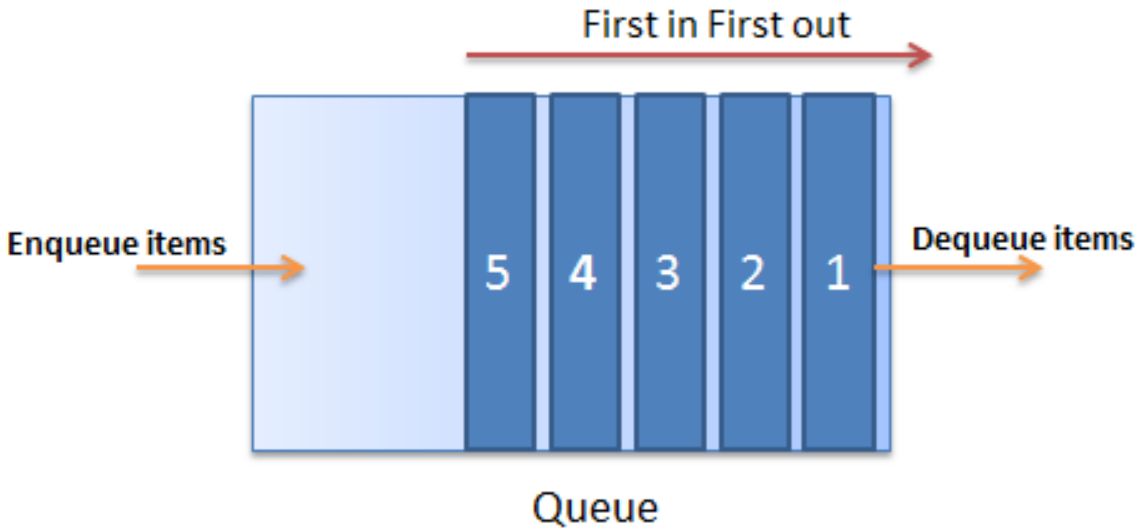
Класс Queue<T>

Queue<T> – обобщённая реализация очереди.

Для добавления/удаления элементов используются методы:

- Enqueue(T) – добавляет элемент в конец очереди;
- Dequeue() – удаляет объект из начала очереди и возвращает его.

Для просмотра элемента в голове очереди без удаления используется метод Peek().



Пример: Очередь Сообщений Сервера

```
using System;
using System.Collections.Generic;

Queue<string> serverMsgQueue = new Queue<string>();
for (int i = 0; i < 5; i++)
{
    serverMsgQueue.Enqueue($"message {i}");
    Console.WriteLine($"Server: added message {i}");
}
Console.WriteLine();

while (serverMsgQueue.Count > 0)
{
    Console.WriteLine($"{serverMsgQueue.Dequeue()} received");
}
```

Вывод:

```
Server: added message 0
Server: added message 1
Server: added message 2
Server: added message 3
Server: added message 4

message 0 received
message 1 received
message 2 received
message 3 received
message 4 received
```

Стек

Проблема: в определённых случаях необходимо смоделировать ситуацию, когда *первыми удаляются* те элементы, которые были добавлены *последними*.

Стек – специальная структура данных, работающая по схеме “Last In – First Out” (LIFO). Первыми из стека удаляются элементы, добавленные последними. Последний добавленный элемент стека называют **вершиной**.

На заметку: стек может реализовываться на основе других структур данных, можно встретить его упоминание как *адаптера* для другой коллекции.

Операции *добавления*, *удаления* и *чтения* применяются к вершине стека.

Исходный код реализации в BCL: Stack, Stack<T>

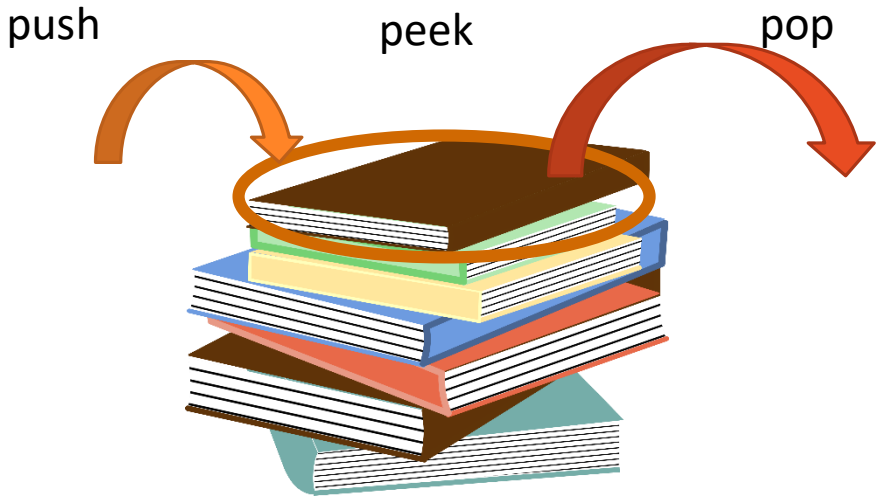
Класс Stack<T>

Stack<T> – обобщённая реализация стека.

Для добавления/удаления элементов используются методы:

- Push(T) – добавляет элемент на вершину стека;
- Pop() – удаляет объект с вершины стека и возвращает его.

Для просмотра элемента на вершине стека без удаления используется метод Peek().



Пример Фрагмента Реализации Stack<T>

```
using System;
using System.Collections.Generic;
class MyStack<T>
{
    public int Count { get => _stack.Count; }
    // Вершина стека - последний элемент списка.
    private List<T> _stack = new List<T>(4);

    public void Push(T value) => _stack.Add(value);
    public T Pop()
    {
        if (Count == 0)
            throw new InvalidOperationException("The stack is empty.");

        int topElementIndex = _stack.Count - 1;
        T removedTopElement = _stack[topElementIndex];
        _stack.RemoveAt(topElementIndex);

        return removedTopElement;
    }
}
```

Пример Использования Класса Stack<T>

```
using System;
using System.Collections.Generic;

Stack<int> demoStack = new Stack<int>();
Console.WriteLine($"Element count: {demoStack.Count}\n");

demoStack.Push(5);
demoStack.Push(10);
demoStack.Push(15);
demoStack.Push(20);
Console.WriteLine($"Element count: {demoStack.Count}");
Console.WriteLine($"Top element: {demoStack.Peek()}\n");

demoStack.Pop();
Console.WriteLine($"Element count: {demoStack.Count}");
Console.WriteLine($"Top element: {demoStack.Peek()}");

Console.Write("\n-> ");
while (demoStack.Count != 0) {
    Console.Write(demoStack.Pop() + " -> ");
}
```

Вывод:

Element count: 0

Element count: 4
Top element: 20

Element count: 3
Top element: 15

-> 15 -> 10 -> 5 ->

Хэш-Таблица

Проблема: возникают ситуации, когда необходимо хранить пары ключ-значение с быстрым поиском как в массиве *без необходимости сортировки элементов*.

Хэш-таблица представляет собой структуру данных, содержащую внутри массив *результатов применения хэш-функции* к каждому из ключей, сопоставленный со множеством значений по каждому из ключей

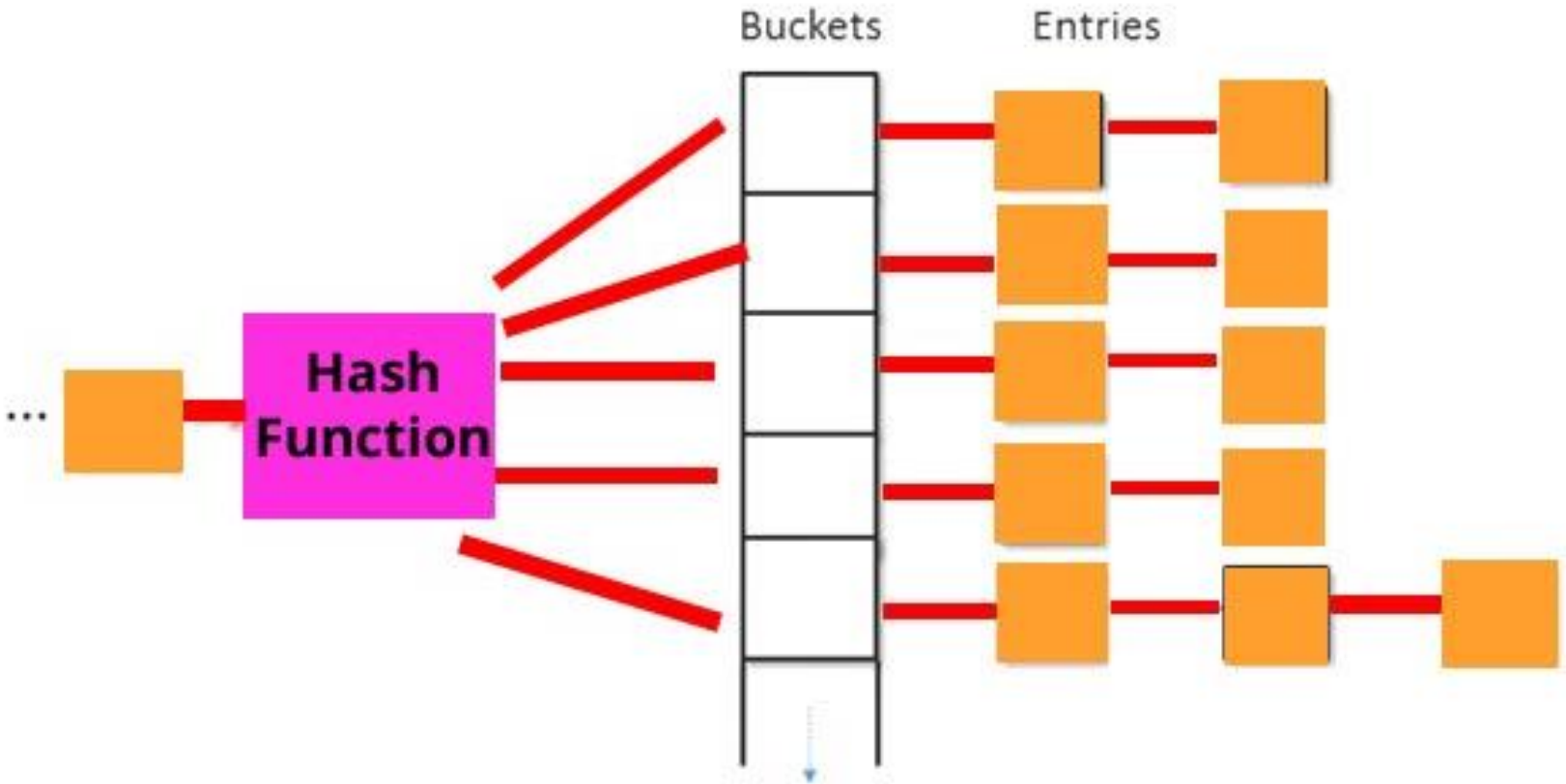
**почему для значений требуется массив, о коллизиях рассказывается в курсе алгоритмов.*

Сложность операций:

- Вставка: **O(1)** или **O(N)** (*в случае перестройки хэш-таблицы*)
- Поиск элемента по значению: **O(1)** или **O(N)** в случае коллизий;
- Удаление: **O(1)** или **O(N)** в случае коллизий;

Исходный код реализации в BCL: Dictionary<TKey, TValue>, HashSet<T>

Визуализация Хэш-Таблицы



Класс HashSet<T>

HashSet<T> – обобщённая реализация хэш-таблицы.

Для добавления/удаления элементов используются методы:

- Add(T) – добавляет элемент с указанным ключом, если его нет, и возвращает true, в противном случае возвращает false;
- Remove(T) – удаляет элемент при его наличии и возвращает true. Возвращает false, если элемент не найден.

HashSet<T> поддерживает некоторые операции над множествами:

- Объединение – UnionWith(IEnumerable<T>);
- Пересечение – IntersectWith(IEnumerable<T>);
- Вычитание – ExceptWith(IEnumerable<T>);

и некоторые другие… (см. документацию).

Пример: Список Заблокированных Абонентов

```
using System;
using System.Collections.Generic;

HashSet<int> blockList = new HashSet<int>();
// Блокировка номеров:
blockList.Add(7_777_999);
blockList.Add(900);
blockList.Add(555_35_35);

// Новый входящий звонок:
int incomingCallNumber = int.Parse(Console.ReadLine());
if (blockList.Contains(incomingCallNumber))
{
    Console.WriteLine($"Unable to proceed a call from {incomingCallNumber}.\n" +
                      $"You have been blocked.");
    return;
}
Console.WriteLine($"Proceeding your call from {incomingCallNumber}...");
```

Ввод:
372

Вывод:
Proceeding your call from 372...

Ввод:
900

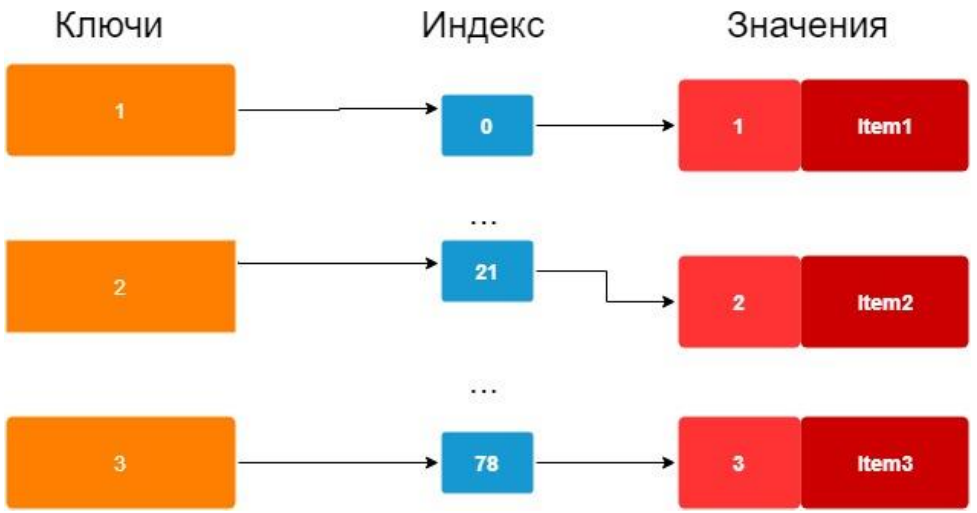
Вывод:
Unable to proceed a call from 900.
You have been blocked.

Класс Dictionary<TKey, TValue>

Dictionary<TKey, TValue> – обобщённая реализация хэш-таблицы с явно заданным типом ключа.

Для добавления/удаления элементов используются методы:

- Add(TKey, TValue) – пытается добавить элемент с указанным ключом, если его нет, при наличии ключа возникает **ArgumentException**;
- Remove(TKey) – удаляет элемент по ключу при его наличии и возвращает true. Возвращает false, если ключ не найден.



Особенность Добавления в Dictionary<TKey, TValue>

Метод Add(TKey, TValue) выдает исключение если:

1. Ключ null (значение может быть null, если TValue – ссылочный тип);
2. Элемент с таким ключом уже существует.

Обращение по несуществующему ключу выдает исключение.

Чтобы избежать исключений, связанных с наличием ключа, рекомендуется использовать методы:

- Метод ContainsKey(TKey) – возвращает true при наличии ключа и false при его отсутствии;
- Метод TryAdd(TKey, TValue) – возвращает false в случае наличия ключа и true в случае успешной вставки. Помните, что null в качестве ключа недопустим, как и в случае с Add.

Пример 1: Телефонная Книга

```
using System;
using System.Collections.Generic;

Dictionary<string, int> phones = new Dictionary<string, int>();
phones.Add("mom", 5555);
phones.Add("dad", 8888);
phones.Add("brother", 9999);
Console.WriteLine($"Brother's number = {phones["brother"]}");
```

Вывод:
Brother's number = 9999

Пример 2: Подсчёт Слов в Книге

```
using System;
using System.Collections.Generic;
using System.IO;

Dictionary<string, int> words = new Dictionary<string, int>();
using (StreamReader sr = new StreamReader(@"book.txt"))
{
    while (!sr.EndOfStream)
    {
        foreach (var word in sr.ReadLine().Split())
        {
            if (!words.ContainsKey(word)) {
                words.Add(word, 0);
            }
            words[word]++;
        }
    }
}
foreach (var pair in words) {
    Console.WriteLine($"{pair.Key} = {pair.Value}");
}
```