ЛЕКЦИЯ 11

- Модуль 2
- 06.12.2023
- Абстрактные типы данных

ЦЕЛИ ЛЕКЦИИ

- Получить представление об абстрактных типах данных (АТД)
- Перестать путать АТД с абстрактными классами
- Перестать путать структуры данных и АТД

НЕМНОГО ОБ АБСТРАКТНЫХ ТИПАХ ДАННЫХ

Понятие АТД, контейнеры, последовательности

АБСТРАКТНЫЕ ТИПЫ ДАННЫХ

- **Абстрактный тип данных** (АТД) [abstract logic design] – функциональное описание некоторого класса сущностей в терминах операций, которые могут выполняться над ними

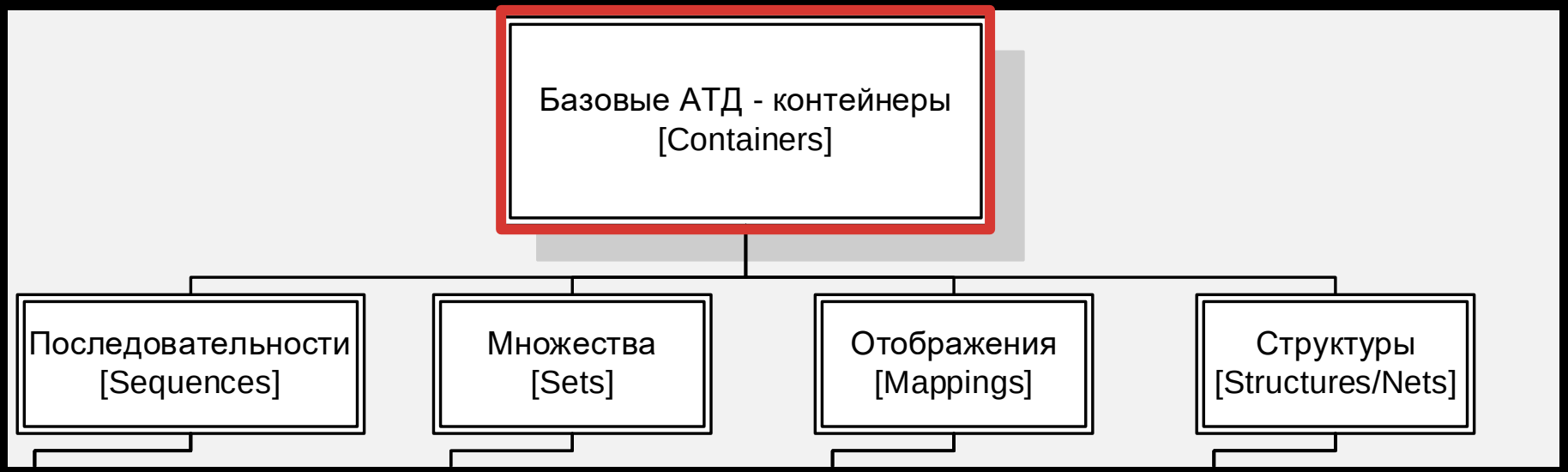
- **Интерфейс АТД** – формальное и однозначное описание синтаксиса и семантики операций, которые могут выполняться над экземплярами АТД
- **Реализация АТД** [ADT implementation] – конкретный тип данных, обеспечивающий заданный интерфейс АТД

Так же, как описание языка программирования не определяет особенности его реализации, так и интерфейс АТД не определяет реализацию АТД

КОНТЕЙНЕР

Контейнер [container] – абстрактный тип данных, представляющий собой структурированную коллекцию информационных элементов, доступ к которым определяется структурой контейнера

Добавление и удаление элементов контейнера - **трансформация**
Доступ к элементу контейнера – операция получения или изменения значения этого элемента

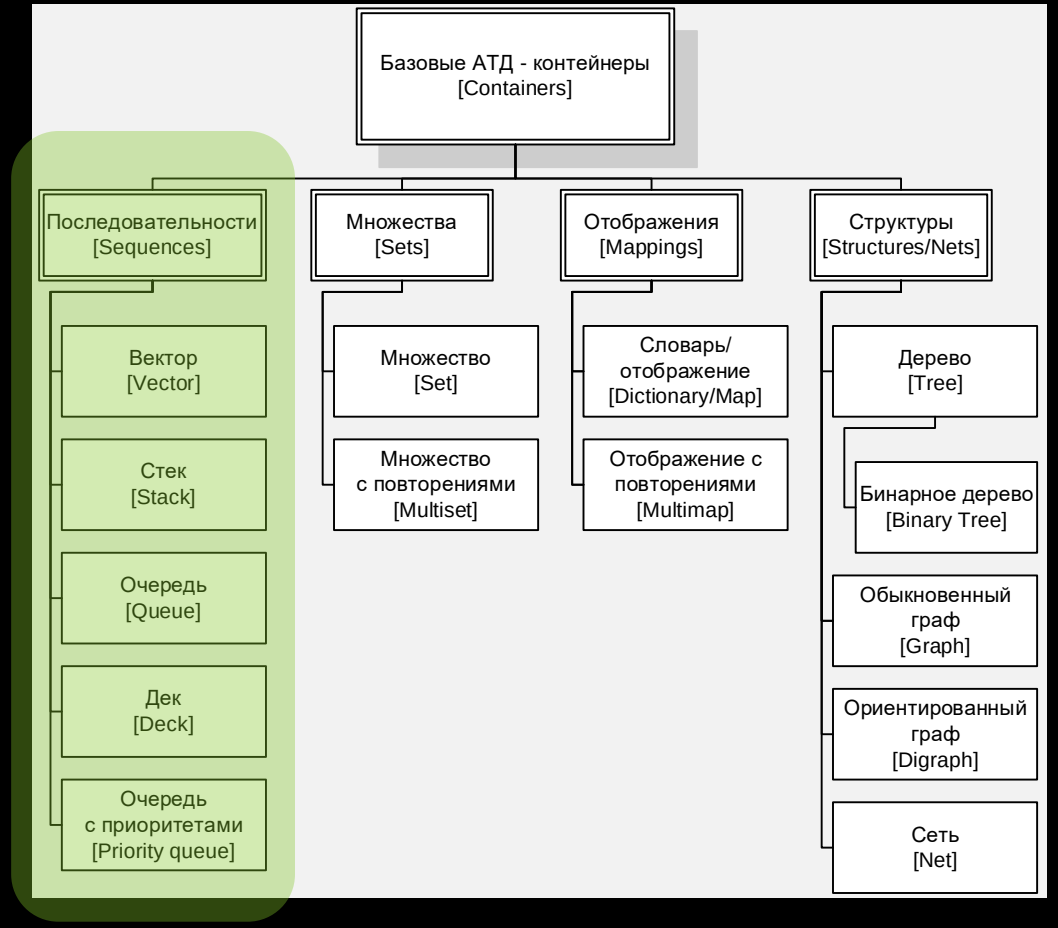


ПОСЛЕДОВАТЕЛЬНОСТИ

Вектор, стек, дек и очередь

ПОСЛЕДОВАТЕЛЬНОСТЬ

Последовательность [sequence] – контейнер, в котором элементы упорядочены по индексам (пронумерованы)



ВЕКТОР

- **Вектор** [vector]
  - последовательность, в которой возможен **доступ** к любому элементу **по индексу** элемента

РЕАЛИЗАЦИИ ВЕКТОРА

- **Вектора в C# реализуются (с ограничениями) несколькими конкретными типами данных (КТД):**
- Array (https://learn.microsoft.com/ru-ru/dotnet/api/system.array?view=net-7.0)
- ArrayList (https://learn.microsoft.com/ru-ru/dotnet/api/system.collections.arraylist?view=net-7.0)
  - Мы его в своих приложениях не используем!
- List<T> (https://learn.microsoft.com/ru-ru/dotnet/api/system.collections.generic.list-1?view=net-7.0#performance-considerations)
  - в статье рассмотрены вопросы производительности ArrayList по отношению к List<T> и требования к типу T, для обеспечения повышенной производительности за счёт использования List<T>
- ArraySegment<T> Struct (https://learn.microsoft.com/en-us/dotnet/api/system.arraysegment-1?view=net-7.0)

ПРИМЕР LIST<T>

Асимптотика:

- Добавление: O(1)
- Удаление, поиск: O(n)

```
public struct Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Person(string firstName, string lastName) =>
        (FirstName, LastName) = (firstName, lastName);
}
```

```
List<Person> list = new List<Person>() { new Person("Joe", "Doe"),
    new Person("Mary", "Smith"), new Person ("John", "White"),
    new Person("Jill", "Collins"), new Person ("Ann", "Black")
};
Console.WriteLine($"Capacity={list.Capacity}");
Console.WriteLine($"Count={list.Count}");
list.TrimExcess();
Console.WriteLine("After TrimExcess()");
Console.WriteLine($"Capacity={list.Capacity}");
Console.WriteLine($"Count={list.Count}");
```

Размер динамически увеличивается по мере добавления элементов

Ёмкость

Фактическое количество элементов

ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ И АВТОМАТИЧЕСКОЕ РАСШИРЕНИЕ

```
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
```

```
List<int> list = new List<int>();
Console.WriteLine(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}");
    }
}
```

СРАВНЕНИЕ ARRAY И LIST: СОРТИРОВКА

```
const int N = 10000;
static Random rnd = new Random();
```

```
int[] ints = new int[N];
 List<int> intList = new List<int>();
 for(int i = 0; i < ints.Length; i++)
 {
     int tmp = rnd.Next(10000);
     ints[i] = rnd.Next(tmp);
     intList.Add(tmp);
 }
```

Массив

```
Stopwatch sw = new Stopwatch();
sw.Start();
Array.Sort(ints);
sw.Stop();
Console.WriteLine($"Array::{sw.Elapsed}");
```

Список

```
 sw.Start();
 intList.Sort();
 sw.Stop();
 Console.WriteLine($"List::{sw.Elapsed}");
```

```
Array::00:00:00.0015311
List::00:00:00.0027566
```

СРАВНЕНИЕ ARRAY И LIST: ВСТАВКА ЭЛЕМЕНТА

```
const int N = 10000;
static Random rnd = new Random();
```

```
int[] ints = new int[N];
List<int> intList = new List<int>();
for(int i = 0; i < ints.Length; i++)
{
    int tmp = rnd.Next(10000);
    ints[i] = rnd.Next(tmp);
    intList.Add(tmp);
}
```

```
const int N = 10_000_000;
```

```
Array::00:00:00.0201310
List::00:00:00.0201314
```

Массив

```
Stopwatch sw = new Stopwatch();
sw.Start();
Array.Resize(ref ints, ints.Length + 1);
ints[ints.Length - 1] = 3;
sw.Stop();
Console.WriteLine($"Array::{sw.Elapsed}");
```

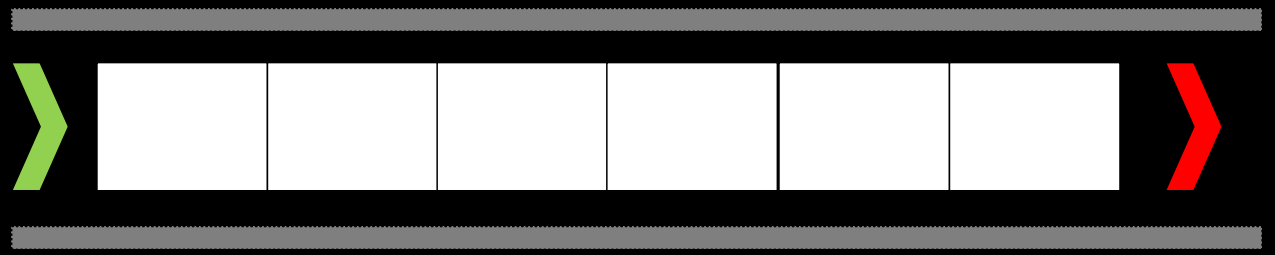
Список

```
sw.Start(); sw.Start();
intList.Add(3);
sw.Stop();
Console.WriteLine($"List::{sw.Elapsed}");
```

```
Array::00:00:00.0000650
List::00:00:00.0000657
```

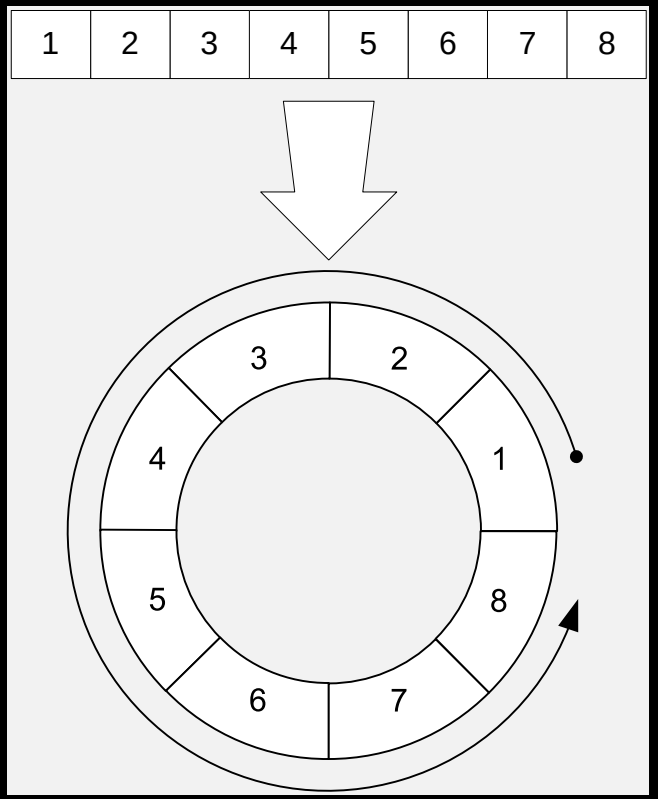
ОЧЕРЕДЬ

- **Очередь** [queue] – дек, в котором возможны только:
  - **доступ** [peek]: к начальному элементу
  - **добавление** [enqueue]: после конечного элемента
  - **удаление** [dequeue] : начального элемента
- Конечный элемент очереди часто называют **хвостом очереди**, а начальный – **головой очереди**



ОЧЕРЕДИ C#

- Класс Queue - коллекция объектов, основанная на принципе «первым поступил — первым обслужен»
  - Мы его в своих приложениях не используем!
- Класс Queue<T> – универсальная очередь, основанная на циклическом массиве
  - Queue<T> операции:
    - Enqueue добавляет элемент в конец элемента Queue<T>
    - Dequeue удаляет самый старый элемент из начала Queue<T>элемента
    - Peek При просмотре возвращается самый старый элемент, который находится в начале, Queue<T> но не удаляет его из Queue<T>

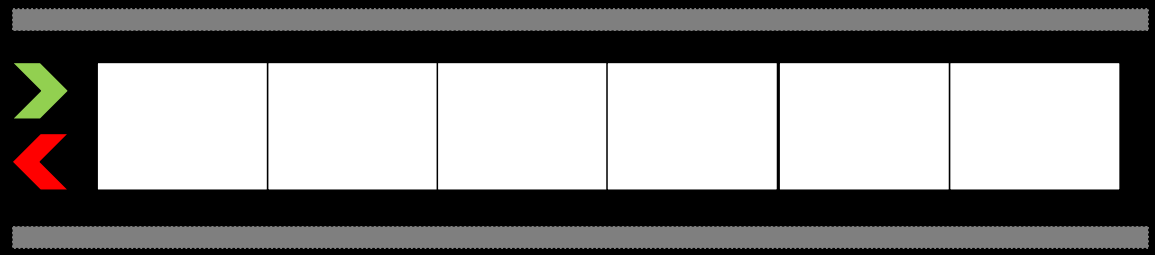


ОЧЕРЕДЬ С ПРИОРИТЕТАМИ

- PriorityQueue<TElement,TPriority> Class (https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.priorityqueue-2?view=net-8.0)

СТЕК

- **Стек** [stack] – дек, в котором возможны только:
  - **Доступ** [peek]: к конечному элементу
  - **Добавление** [push]: после конечного элемента
  - **Удаление** [pop]: конечного элемента
- Конечный элемент стека называют вершиной стека



СТЕКИ C#

- Класс Stack (https://learn.microsoft.com/ru-ru/dotnet/api/system.collections.stack?view=net-7.0)
  - Мы его в своих приложениях не используем!
- Класс Stack<T> (https://learn.microsoft.com/ru-ru/dotnet/api/system.collections.generic.stack-1?view=net-7.0)
- Класс ConcurrentStack<T> (https://learn.microsoft.com/ru-ru/dotnet/api/system.collections.concurrent.concurrentstack-1?view=net-7.0)

СТЕК СВОИМИ РУКАМИ

```
class Stack {
   // вершина стека - последний элемент массива
   int[] stack; // массив для элементов
   public void Push(int x) {
      if (stack == null) {
         stack = new int[] { x };
         return;
      }
      Array.Resize(ref stack, stack.Length + 1);
      stack[stack.Length - 1] = x;
   }
   public void Pop () {
      if (stack == null) return;
         if (stack.Length == 1) { stack = null; return; }
         Array.Resize(ref stack, stack.Length - 1);
      }
}
```

```
public Stack() { stack = null; }
public string ToString() {
   if (stack == null) return "Стек пуст";
   // слева на право
   string tmp = "Stack: ";
   for (int i = stack.Length - 1; i > 0; i--)
      tmp += (stack[i]+" -> ");
   tmp += stack[0];
   return tmp;
}
```

Пример реализации стека в рамках структурного подхода [http://neerc.ifmo.ru/wiki/index.php?title=Стек]

КОД ДЛЯ ОТЛАДКИ ОБЪЕКТА КЛАССА STACK

```
Stack example = new Stack();
Console.WriteLine(example.ToString());
example.Push(13);
Console.WriteLine(example.ToString());
example.Pop();
Console.WriteLine(example.ToString());
example.Push(1);
example.Push(2);
example.Push(3);
Console.WriteLine(example.ToString());
```

```
Стек пуст
Stack: 13
Стек пуст
Stack: 3 -> 2 -> 1
```

ТРАНСФОРМАЦИИ СТЕКА И ОЧЕРЕДИ

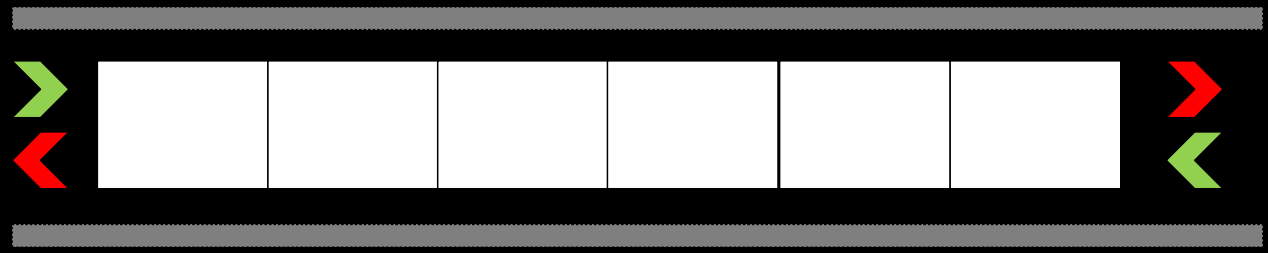
- **Типы трансформаций стека и очереди** часто обозначают аббревиатурами:
  - **LIFO** (Last In – First Out, последним пришёл – первым вышел)
  - **FIFO** (First In – First Out, первым пришёл – первым вышел)

Peek() – возвращает информацию об элементе

Enqueue()
Dequeue()
Push()
Pop()
Изменяют состояние контейнера

ДЕК

- **Дек** [*deque, double ended queue*] – последовательность, в которой возможны только:
  - **доступ** [peek]**:** к концевым элементам
  - **добавление:** до начального (push front)и после конечного элемента (push back)
  - **удаление:** концевых элементов (pop back, pop front)



МНОЖЕСТВА

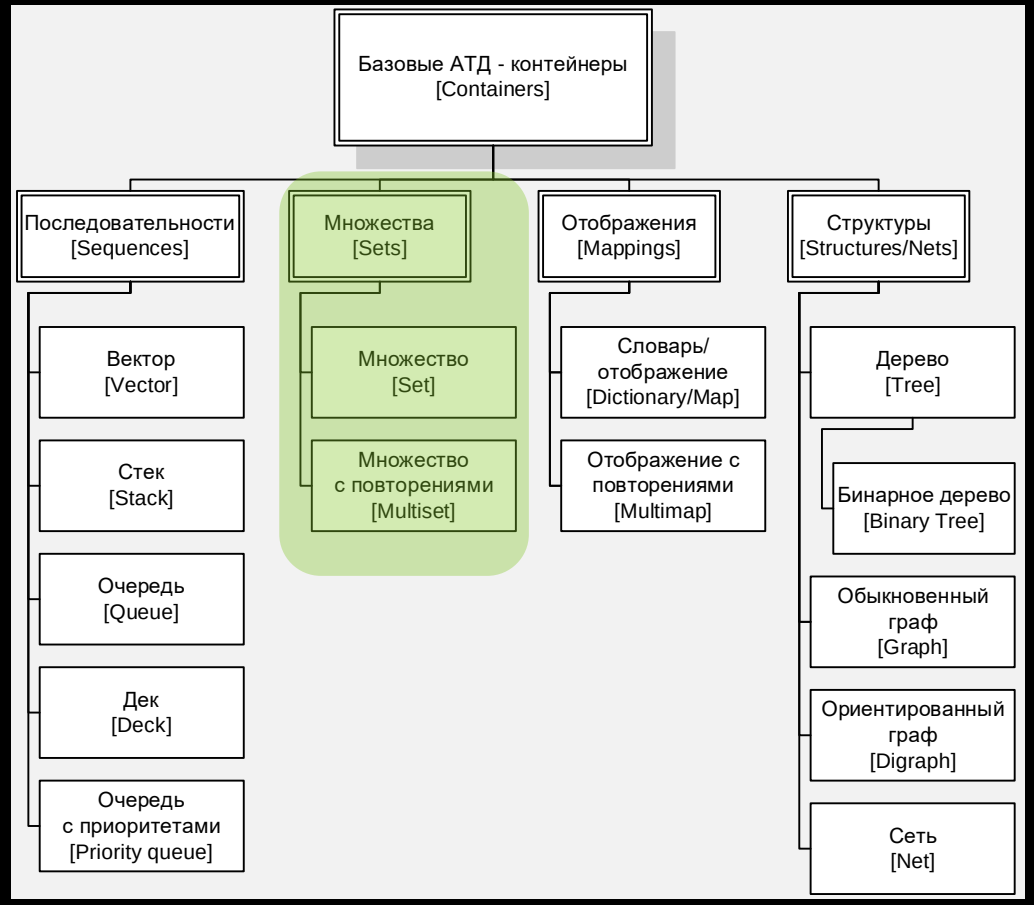
Множество и мультимножество

МНОЖЕСТВО

Множество [set] – АТД, соответствующий математическому определению множества, но, как правило, однотипных элементов

Операции над множествами:

- проверка принадлежности элемента с данным значением множеству
- добавление элемента
  - если элемент с данным значением уже принадлежит множеству, то ничего не происходит
- удаление элемента
  - если элемент с данным значением не принадлежит множеству, то ничего не происходит
- нахождение объединения, пересечения и разности с другим множеством



ПРИМЕР HASHSET<T> (1)

```
// Создаем HashSet для хранения уникальных значений.
HashSet<int> numbers = new HashSet<int>();

// Добавляем элементы в numbers.
numbers.Add(1);
numbers.Add(2);
numbers.Add(3);
numbers.Add(4);
```

```
Console.WriteLine("элементы:");
foreach (int number in numbers)
{
    Console.WriteLine(number);
}
```

```
// Попытка добавить дубликат элемента.
if (numbers.Add(3))
{
    Console.WriteLine("Элемент 3 успешно добавлен во множество.");
}
else
{
    Console.WriteLine("Элемент 3 уже существует во множестве.");
}
```

ПРИМЕР HASHSET<T> (2)

```
// Удаляем элемент из HashSet
numbers.Remove(2);

Console.WriteLine("HashSet после удаления элемента 2:");
foreach (var number in numbers)
{
    Console.WriteLine(number);
}
```

```
// Проверяем наличие элемента в HashSet
int searchNumber = 4;
if (numbers.Contains(searchNumber))
{
    Console.WriteLine($"HashSet содержит элемент {searchNumber}.");
}
else
{
    Console.WriteLine($"HashSet не содержит элемент {searchNumber}.");
}
```

КОЛЛЕКЦИИ И МУЛЬТИМНОЖЕСТВА

Коллекция [collection] – множество элементов произвольных типов

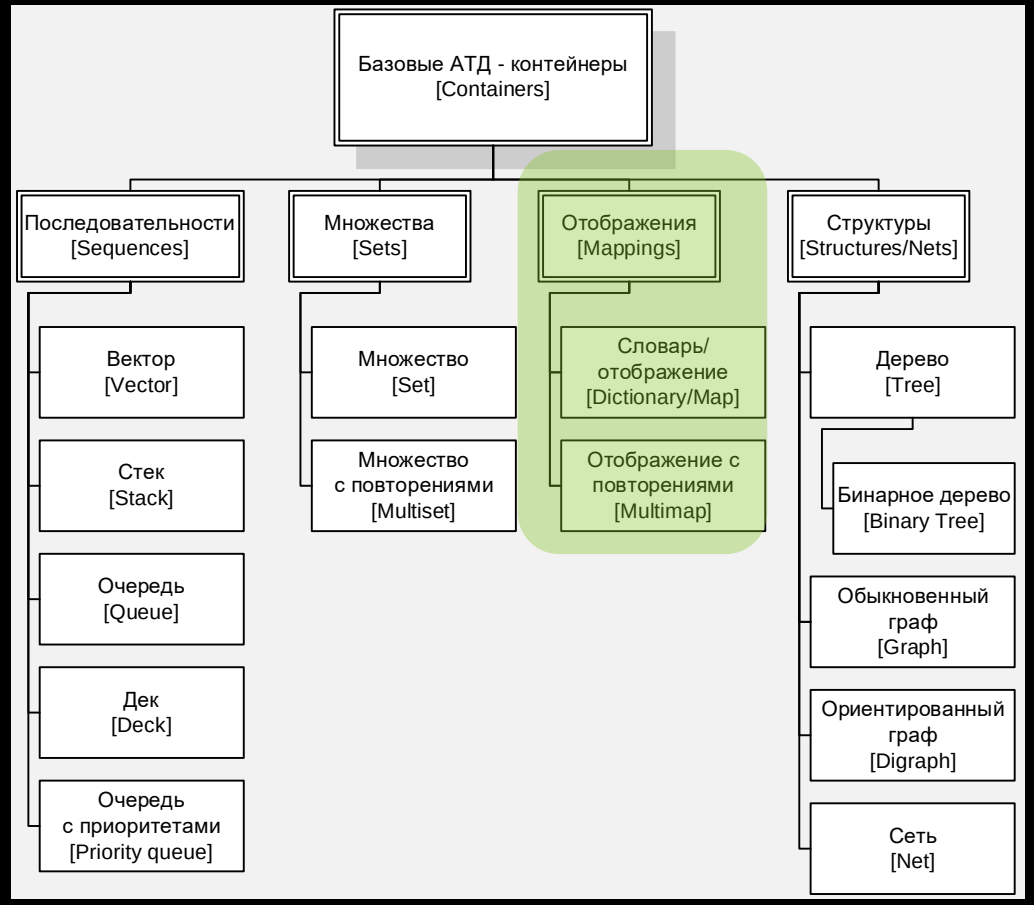
Мультимножество (множество с повторениями, сумка) [multiset, bag] – расширение понятия множества, позволяющее включать несколько одинаковых элементов

ОТОБРАЖЕНИЯ

Словарь [dictionary] или **отображение** [map] – ассоциативный контейнер, элементы которого разделены на ключевое поле (ключ) и информационное поле, и доступ к элементам возможен по значению ключевого поля.

Операции над словарями:

- проверка того, что в словаре есть элемент с данным значением ключа
- доступ к информационному полю элемента с заданным значением ключа
- добавление элемента (если элемент с данным значением ключа уже присутствует, то его информационное поле изменяется)
- удаление элемента (если элемент с данным значением ключа отсутствует, то ничего не происходит)



РЕАЛИЗАЦИЯ СЛОВАРЕЙ В C#

- Класс Hashtable (ключ понимается как хеш-код)
  - Мы его в своих приложениях не используем!
- Обобщённый класс Dictionary<TKey,TValue>

ПРИМЕР ИСПОЛЬЗОВАНИЯ DICTIONARY<>

```
struct Person
{
    public string name;
    public string lastname;
    public Person(string n, string ln) => (name, lastname) = (n, ln);
    public override string ToString() => $"{name} {lastname}";
}
```

Используем структуры, как ключ

```
Dictionary<Person, string> persTown = new Dictionary<Person, string>();

persTown.Add(new Person("Ivan", "Ivanov"), "Moscow");
persTown.Add(new Person("Joe", "Doe"), "New Orlean");
persTown.Add(new Person("I", "Wan"), "Peking");

foreach(var p in persTown)
{
    Console.WriteLine($"{p.Key} is living in {p.Value} ");
}
```

КОЛЛЕКЦИИ В C#

System.Array

System.Collections

System.Collections.Generic

System.Collections.Concurrent

System.Collections.Immutable

СВОЙСТВА КОЛЛЕКЦИЙ C#

- **Позволяют добавлять, удалять и искать элемент**
  - Add(), Remove(), Find()
- **Перечислимы** (как следствие ICollection)
  - реализуют интерфейс System.Collections.IEnumerable или System.Collections.Generic.IEnumerable<T>
- **Допускают копирование элементов в массив**
  - CopyTo()

РЕКОМЕНДАЦИИ ПО ВЫБОРУ КОЛЛЕКЦИИ C#

Требуемое действие	**Обобщенная**	**Необобщенная**
Хранение в виде пар «ключ-значение» для быстрого поиска по ключу	Dictionary<TKey,TValue>	Hashtable
Доступ к элементам по индексу	List<T>	Array ArrayList
Доступ по принципу FIFO	Queue<T>	Queue
Доступ по принципу LIFO	Stack<T>	Stack
Последовательный доступ к элементам	LinkedList<T>	
Получение оповещений при удалении \ добавлении элементов в коллекцию	ObservableCollection<T>	
Упорядоченная коллекция	SortedList<TKey,TValue>	SortedList
Множество с поддержкой математических множественных операций	HashSet<T> SortedSet<T>	

АЛГОРИТМИЧЕСКАЯ СЛОЖНОСТЬ

Изменяемые	Средняя	Худший случай	Неизменяемая	Сложность
`Stack<T>.Push`	O(1)	O(n)	`ImmutableStack<T>.Push`	O(1)
`Queue<T>.Enqueue`	O(1)	O(n)	`ImmutableQueue<T>.Enqueue`	
`List<T>.Add`	O(1)	O(n)	`ImmutableList<T>.Add`	O(log n)
`List<T>.Item[Int32]`	O(1)	O(1)	`ImmutableList<T>.Item[Int32]`	O(log n)
`List<T>.Enumerator`	O(n)	O(n)	`ImmutableList<T>.Enumerator`	O(n)
`HashSet<T>.Add, lookup`	O(1)	O(n)	`ImmutableHashSet<T>.Add`	O(log n)
`SortedSet<T>.Add`	O(log n)	O(n)	`ImmutableSortedSet<T>.Add`	O(log n)
`Dictionary<T>.Add`	O(1)	O(n)	`ImmutableDictionary<T>.Add`	O(log n)
`Dictionary<T> lookup`	O(1)	O(1) или строго O(n)	`ImmutableDictionary<T> lookup`	O(log n)
`SortedDictionary<T>.Add`	O(log n)	O(n log n)	`ImmutableSortedDictionary<T>.Add`	O(log n)

СТРУКТУРЫ ДАННЫХ ДЛЯ РЕАЛИЗАЦИИ АТД

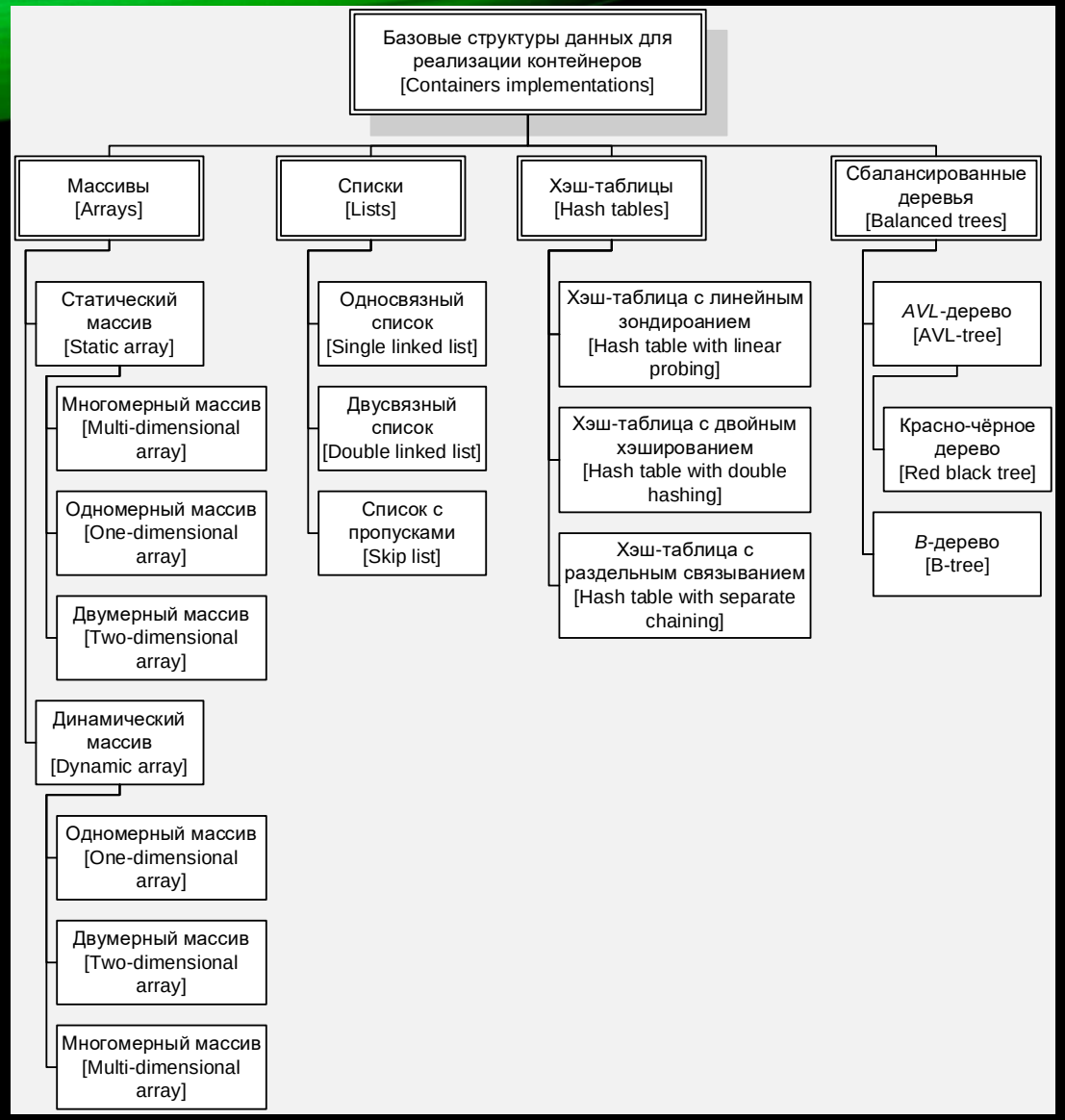
Односвязный и двусвязный список

СТРУКТУРЫ ДАННЫХ

Массив [array] и **связный список** [linked list] (как структуры данных) – соответственно базовые индексная и ссылочная структуры данных в памяти

- **Массив** [array] – последовательность однотипных элементов, к каждому из которых можно получить доступ по его индексу за константное время (примерно одинаковое время, не зависящее от значения индекса). Массив – базовая структура данных с индексацией
  - Массив представляет собой непрерывный участок памяти, а список – кусочки памяти для каждого элемента, связанные указателями (ссылками в виде адресов). Массив идентифицируется адресом первого элемента. Список идентифицируется адресом одного из своих элементов, от которого по указателям можно добраться до всех остальных элементов

СТРУКТУРЫ ДАННЫХ



- **Все более сложные структуры данные представляют собой «переплетение» набора массивов и списков**
  - как элементом массива, так и информационной частью элемента списка может быть адрес другого массива или элемента списка

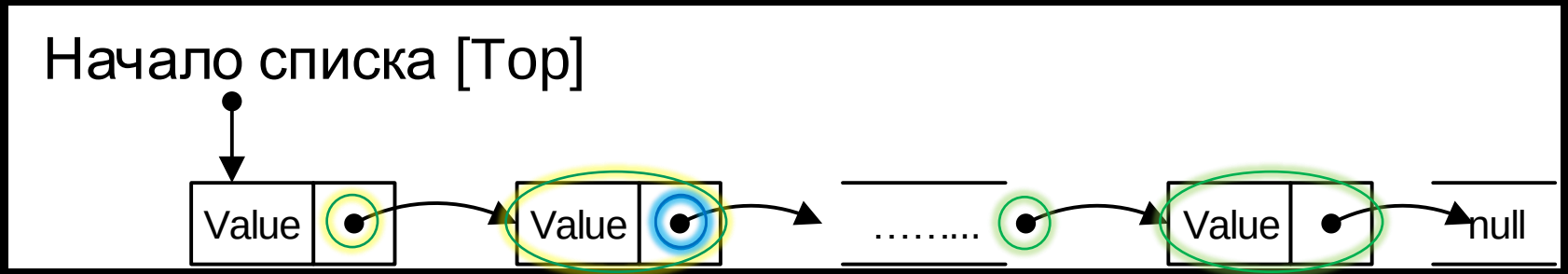
СПИСОК

- **Список** представляет собой такую реализацию последовательности однотипных элементов, когда элементы связаны друг с другом посредством ссылок
  - Список – базовая ссылочная структура
- Каждый элемент списка содержит не только информационное поле, но одно или более полей со ссылками на другие элементы

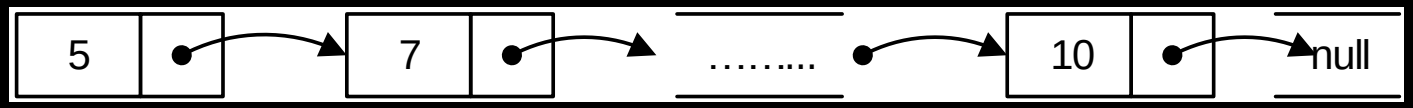
НАИБОЛЕЕ РАСПРОСТРАНЁННЫЕ СПИСКИ

- **Односвязные списки:** каждый элемент, кроме последнего, содержит ссылку на следующий, последний элемент ссылается на *null*
- **Двухсвязные списки:** каждый элемент, кроме первого и последнего, содержит ссылки на предыдущий и следующий

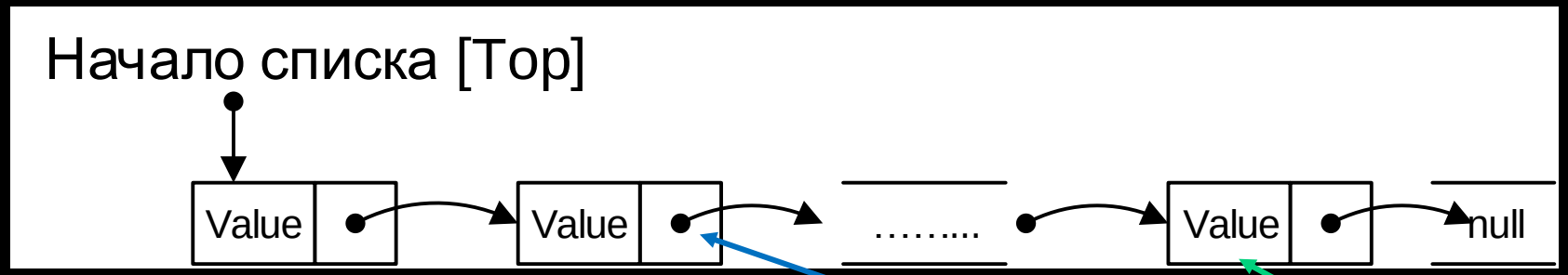
ОДНОСВЯЗНЫЙ СПИСОК



Примеры



МОДЕЛЬ ЭЛЕМЕНТА ОДНОСВЯЗНОГО СПИСКА



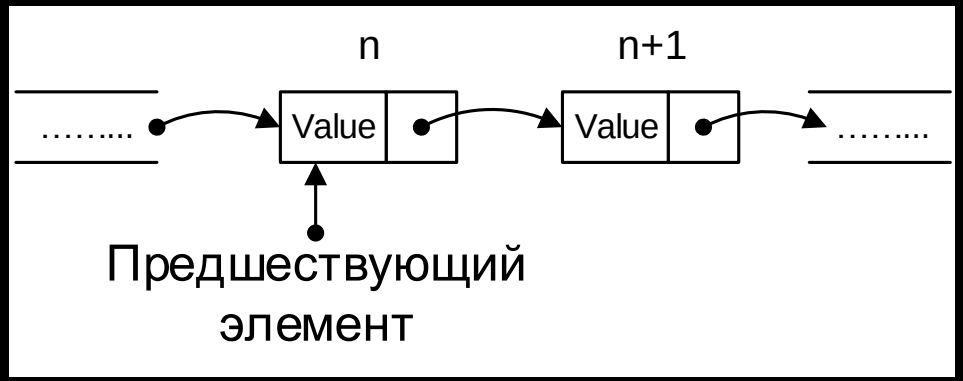

```
public class MyListItem    {
        // значение элемента
        public int Value { get; set; }
        //ссылка на следующий элемент списка
        public MyListItem Next { get; set; }
        public MyListItem() { }
        public MyListItem(int val)    {
            Value = val;
            Next = null;
        }
}
```

ВСТАВКА ЭЛЕМЕНТА В ОДНОСВЯЗНЫЙ СПИСОК

- Пусть имеется некоторый односвязный список и ссылка p на элемент, после которого мы хотим вставить новый элемент
- *Алгоритм добавления элемента*

1. создание нового элемента
2. присвоение значений его полям
3. добавление связи с элементом, следующим за **P**
4. добавление связи с элементом **P**

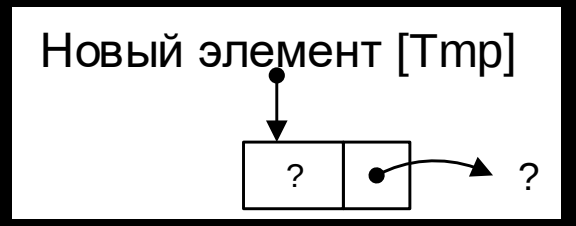
Состояние списка перед добавление элемента



ВСТАВКА ЭЛЕМЕНТА В ОДНОСВЯЗНЫЙ СПИСОК

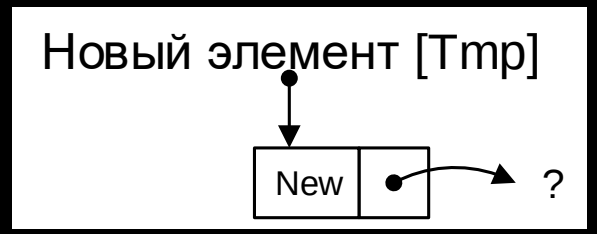
```
MyListItem head = new MyListItem(1);
head.Next = new MyListItem(2);
head.Next.Next = new MyListItem(3);
head.Next.Next.Next = new MyListItem(4);

MyListItem tmp = head;
While (tmp != null) {
    Console.WriteLine(tmp.Value);
    tmp = tmp.Next;
}
```

1. Создание нового элемента




```
MyListItem Tmp = new MyListItem();
```

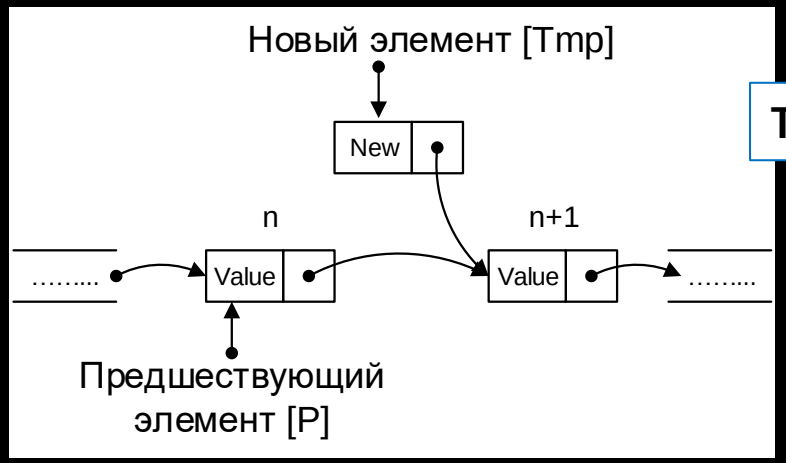
2. Присвоение значений его полям




```
Tmp.Value = 2;
```

ВСТАВКА ЭЛЕМЕНТА В ОДНОСВЯЗНЫЙ СПИСОК

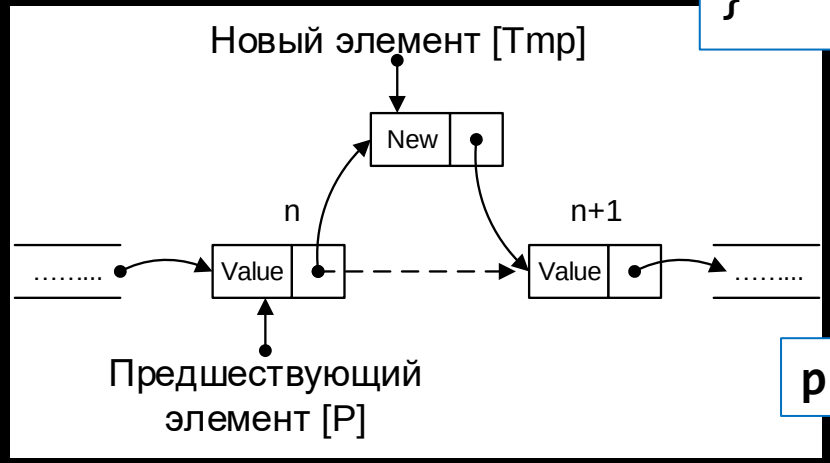
3. добавление связи с элементом, следующим за P




```
Tmp.Next = p.Next;
```

```
MyListItem head = new MyListItem(1);
head.Next = new MyListItem(2);
head.Next.Next = new MyListItem(3);
head.Next.Next.Next = new MyListItem(4);

MyListItem tmp = head;
While (tmp != null) {
    Console.WriteLine(tmp.Value);
    tmp = tmp.Next;
}
```

4. добавление связи с элементом P




```
p.Next = Tmp;
```

ДОБАВЛЕНИЕ ЭКЗЕМПЛЯРНОГО МЕТОДА В КЛАСС

```
 public void AddNext(MyListItem newItem) {
            // нужно заменить ссылки
            newItem.Next = this.Next;
            this.Next = newItem;
}
```

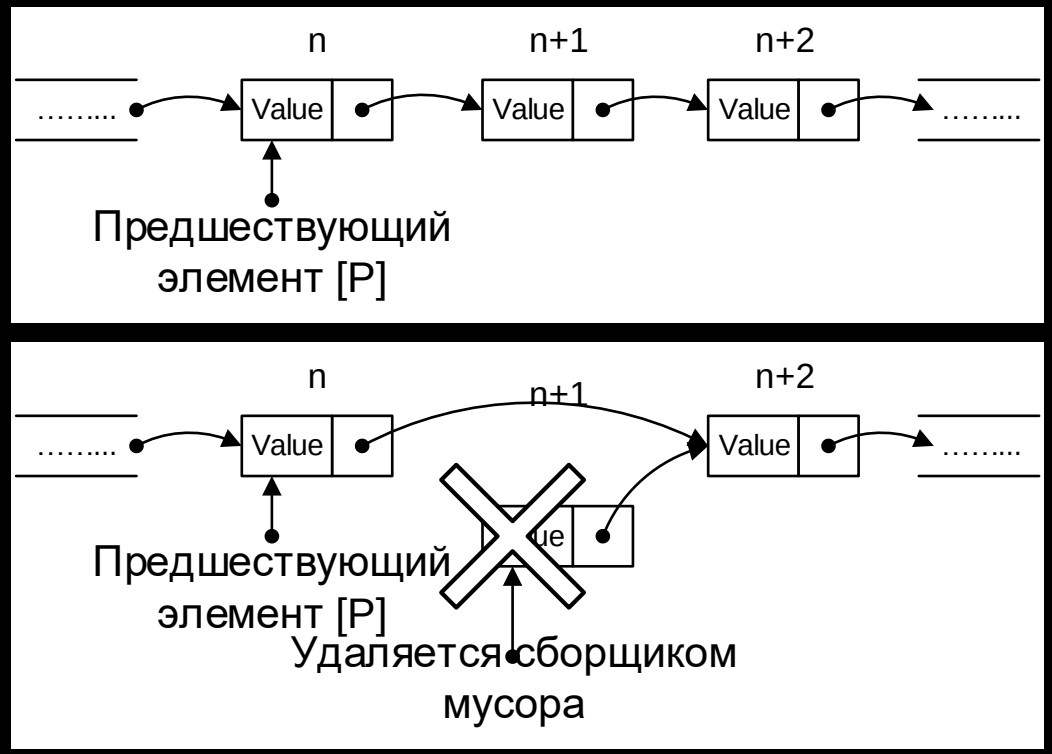
УДАЛЕНИЕ ЭЛЕМЕНТА ИЗ ОДНОСВЯЗНОГО СПИСКА

- Пусть имеется некоторый односвязный список и ссылка p на элемент, предшествующий тому, который мы хотим удалить.

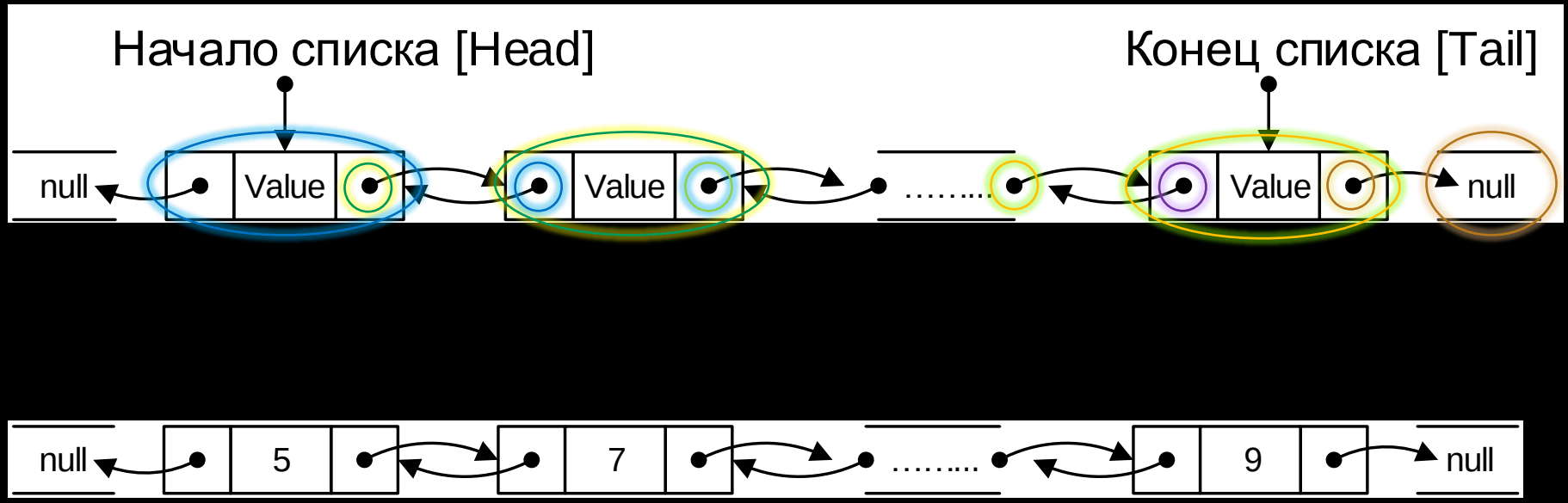
Алгоритм удаления элемента

1. добавление связи между **P** и следующим за удаляемым элементом

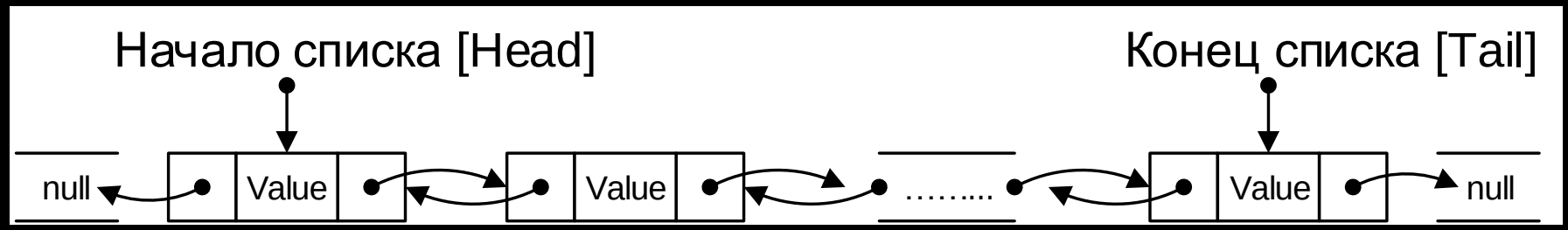
Состояние списка перед удалением элемента



ДВУСВЯЗНЫЙ СПИСОК



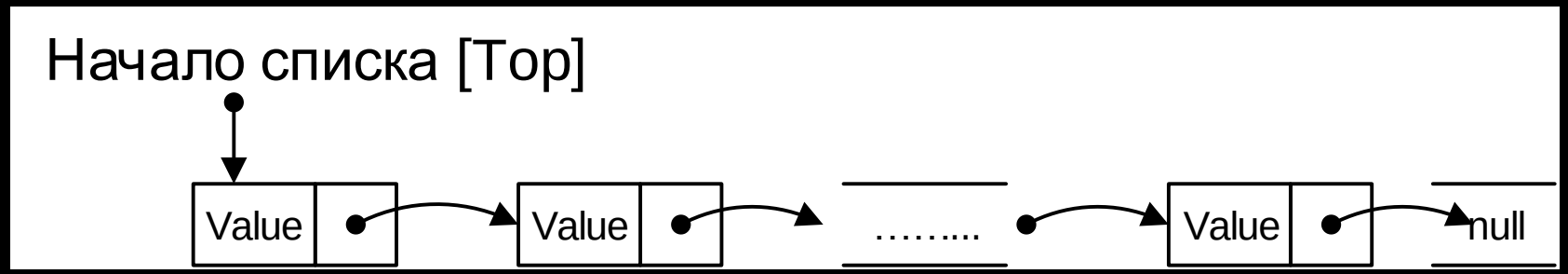
ЭЛЕМЕНТ ДВУСВЯЗНОГО СПИСКА

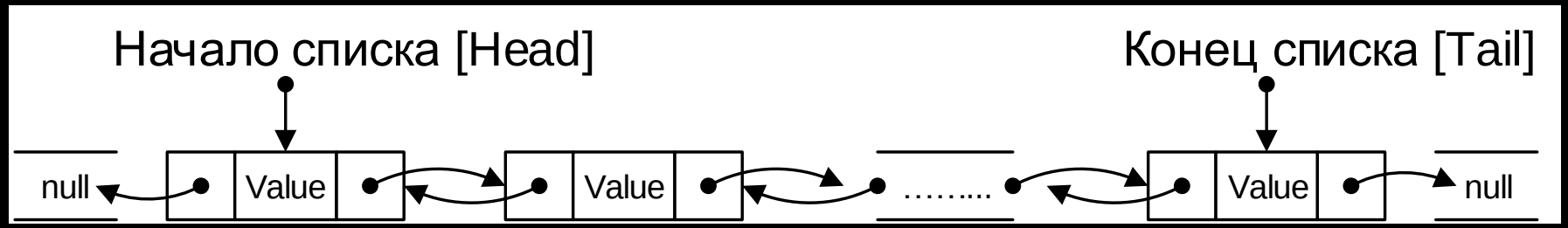



```
public class DoublleLinkyItem {
        public int Value { get; set; }
        public DoublleLinkyItem Next { get; set; }
        public DoublleLinkyItem Prev { get; set; }
}
```

ЗАКОЛЬЦОВАННЫЕ СПИСКИ

- Закольцевать список можно путём изменения ссылки последнего элемента таким образом, чтобы она указывала на первый элемент и наоборот





ИСПОЛЬЗОВАННАЯ ЛИТЕРАТУРА

- Очередь [http://neerc.ifmo.ru/wiki/index.php?title=Очередь]
- Очередь: классы Queue и Queue<T> (https://professorweb.ru/my/csharp/charp_theory/level12/12_7.php)
- Класс Queue (https://learn.microsoft.com/ru-ru/dotnet/api/system.collections.queue?redirectedfrom=MSDN&view=net-7.0)
- Класс Queue<T> (https://learn.microsoft.com/ru-ru/dotnet/api/system.collections.generic.queue-1?view=net-7.0)
- Класс ConcurrentQueue<T> (https://learn.microsoft.com/ru-ru/dotnet/api/system.collections.concurrent.concurrentqueue-1?view=net-7.0)
- Класс Stack (https://learn.microsoft.com/ru-ru/dotnet/api/system.collections.stack?view=net-7.0)
- Класс Stack<T> (https://learn.microsoft.com/ru-ru/dotnet/api/system.collections.generic.stack-1?view=net-7.0)
- Класс ConcurrentStack<T> (https://learn.microsoft.com/ru-ru/dotnet/api/system.collections.concurrent.concurrentstack-1?view=net-7.0)
- Collections and Data Structures (https://learn.microsoft.com/en-us/dotnet/standard/collections/)
- Choose a collection