

# Структуры Данных

Фоменко Максим Юрьевич

# Что такое Структура Данных

- Структура данных — программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных в вычислительной технике. Нужны для организации данных в памяти компьютера для их эффективного использования.

Эффективность определяется:

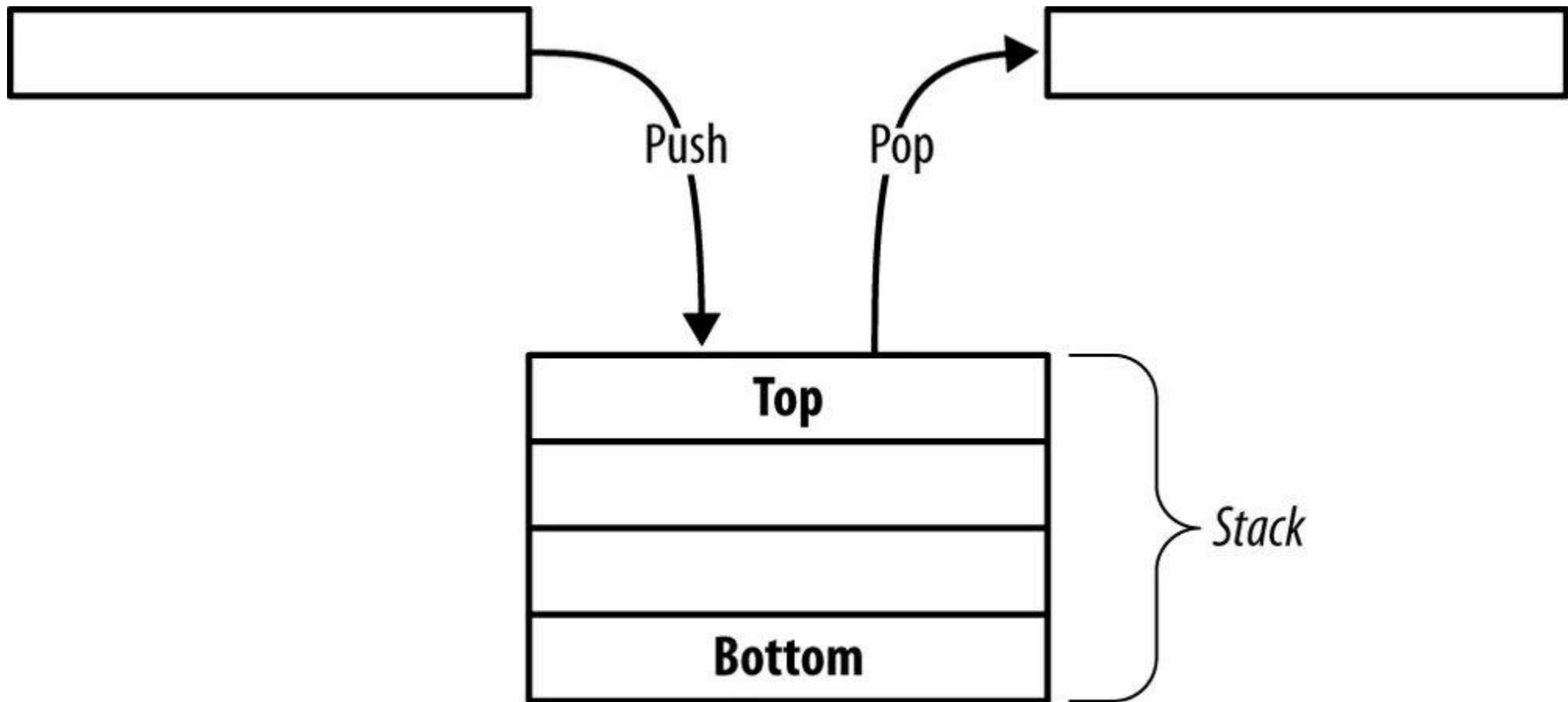
- Временной сложностью: Как быстро выполняются операции (добавление, поиск, удаление).
- Пространственной сложностью: Сколько памяти структура потребляет.

# Классификация Структур Данных

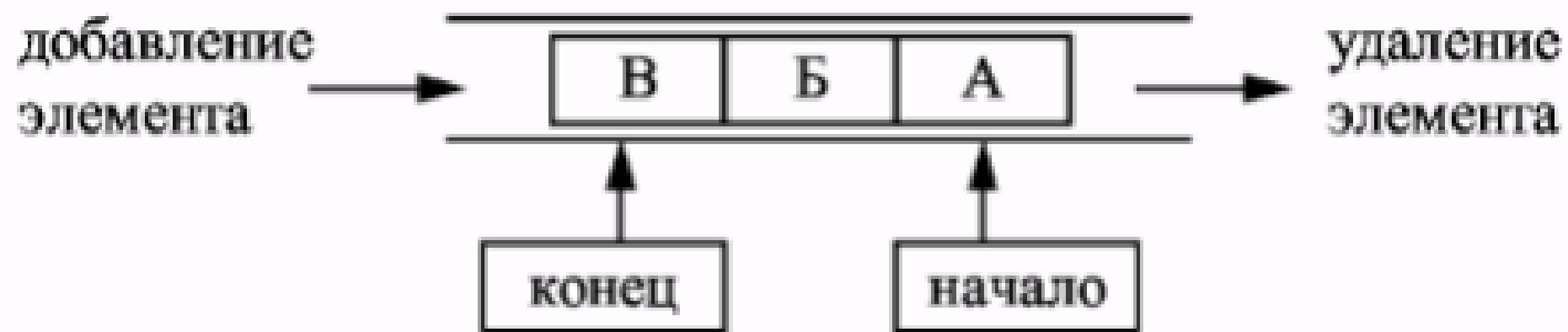
**Линейные структуры:** Элементы расположены последовательно, один за другим.

1. Массив (Array): Статическая, индексированная коллекция.
2. Список (List): Динамическая, индексированная коллекция.
3. Стек (Stack): LIFO (Last-In, First-Out).
4. Очередь (Queue): FIFO (First-In, First-Out).

# Классификация Структур Данных (стек)



# Классификация Структур Данных (очередь)

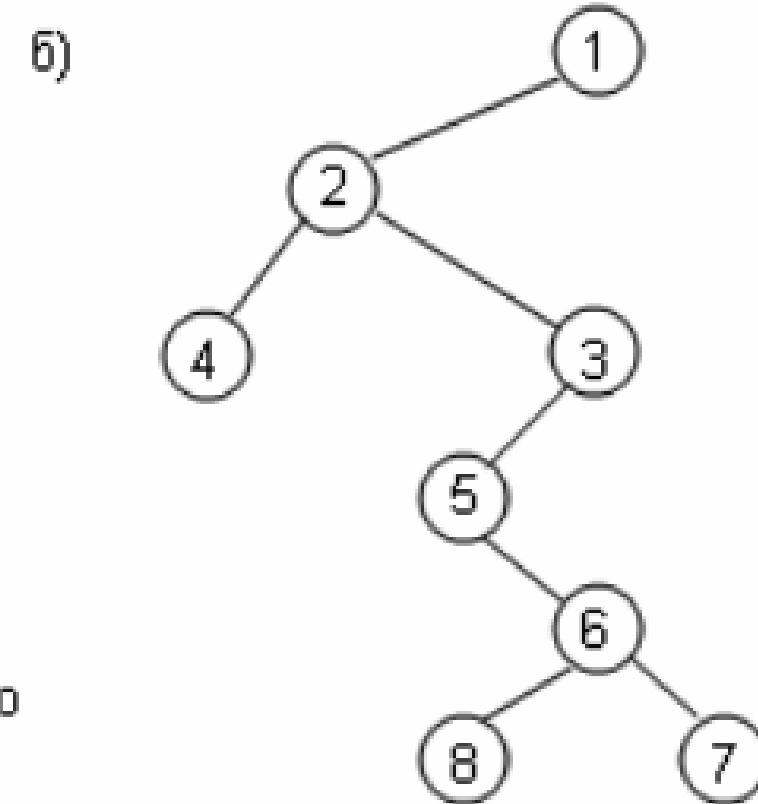
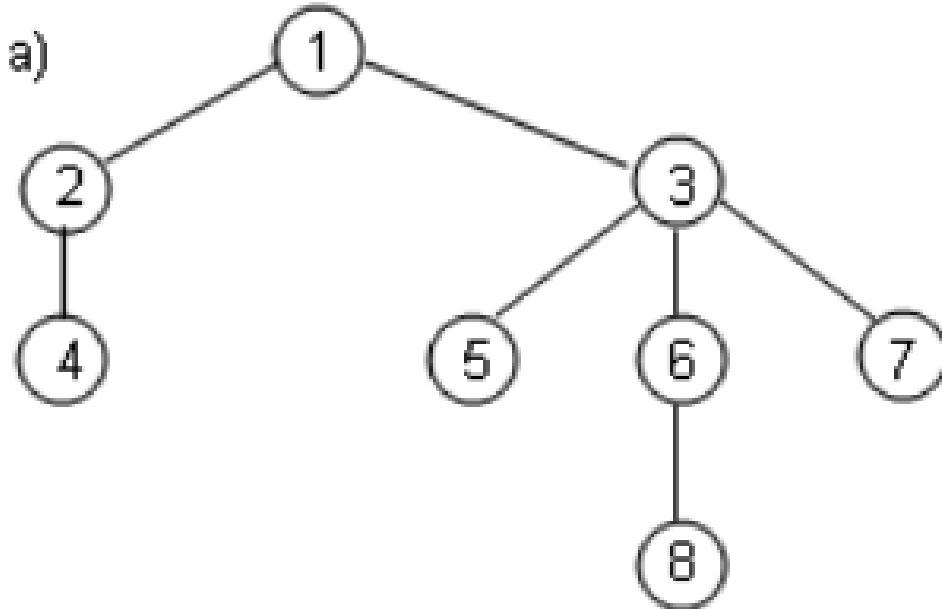


# Классификация Структур Данных

**Нелинейные структуры:** Элементы имеют более сложные, непоследовательные связи.

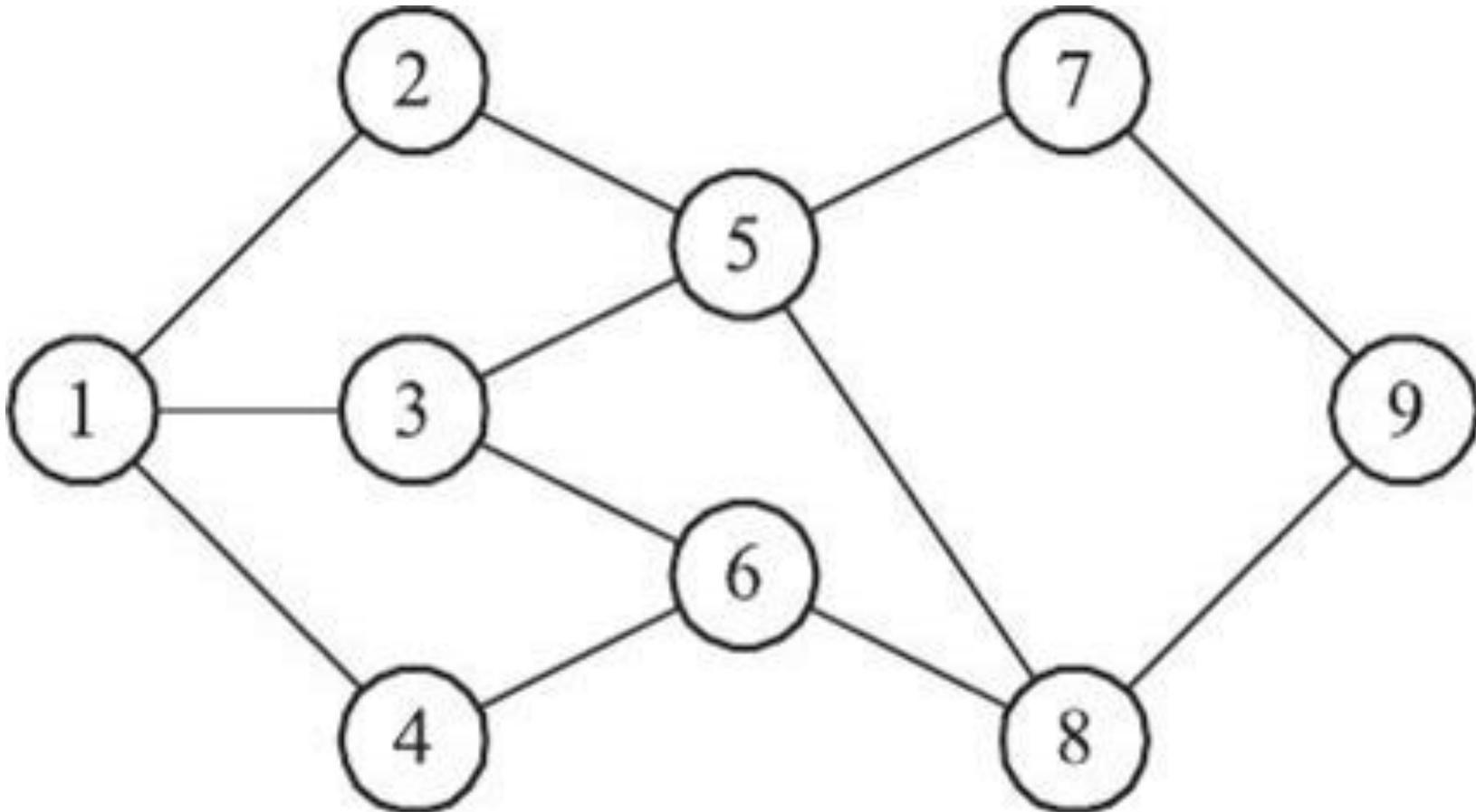
1. Деревья (Trees): Иерархическая структура.
2. Графы (Graphs): Сетевая структура.

# Классификация Структур Данных (дерево)



а) упорядоченное дерево  
б) бинарное дерево

# Классификация Структур Данных (граф)



# Классификация Структур Данных

**Ассоциативные структуры:** Хранение данных в виде пар «ключ-значение».

Словарь (Dictionary) / Хэш-таблица (Hashtable).

Множество (Set) - структура данных, которая представляет собой набор уникальных и неупорядоченных элементов.

Каждый элемент является уникальным, порядка нет.

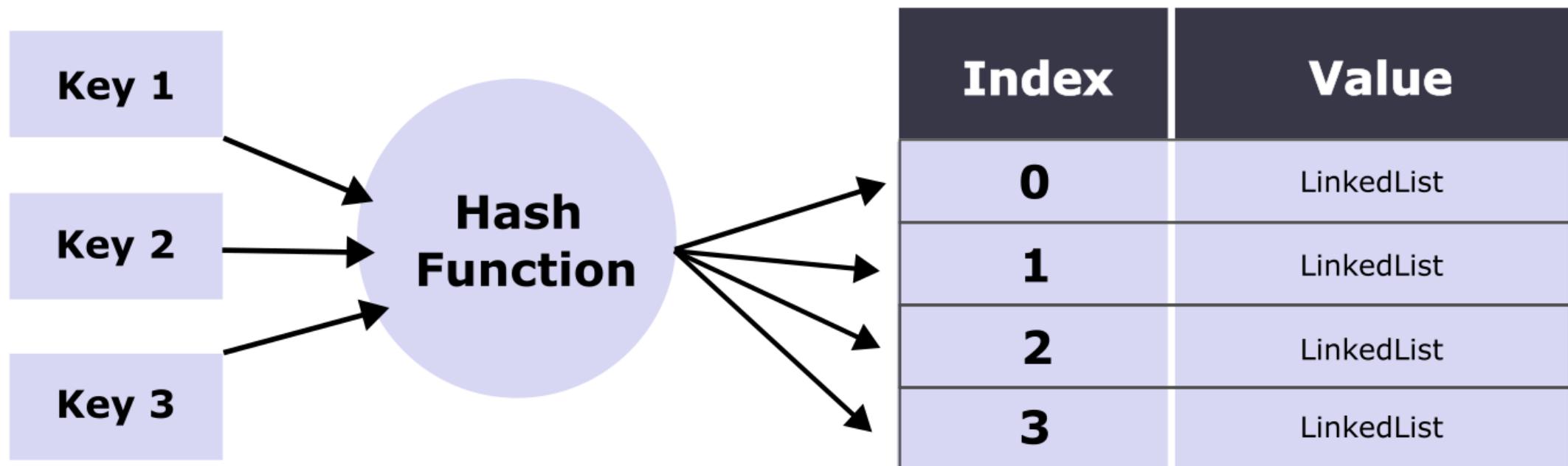
**Хэширование** (хеширование) широко применяется в программировании для создания **уникальных идентификаторов**, сравнения данных и обеспечения целостности информации. Для этого используются специальные алгоритмы — хеш-функции.

# Классификация Структур Данных

📦 "Amogus".GetHashCode()	996821276
📦 "Amogus".GetHashCode()	996821276
📦 "amogus".GetHashCode()	-277244297
📦 "AAmogus".GetHashCode()	418858756

**индекс = хэш(ключ) % размер массива**

# Классификация Структур Данных



# Сложность Алгоритмов (O-нотация)

Сказать «этот алгоритм быстрый, а этот медленный» как сказать «эта машина **красная**» - не информативно. Нам нужен инструмент, который измеряет не секунды (*они все таки зависят от процессора, а он у всех свой*), а **масштабируемость** алгоритма.

# Сложность Алгоритмов (O-нотация)

## «O» большое (Big O Notation):

O нотация описывает, насколько сильно замедлится алгоритм, если увеличить количество входных данных ( $N$ ) в  $X$  раз.

**Она описывает НАИХУДШИЙ сценарий.**

*(Наихудший сценарий – когда выполняются все потенциальные операции)*

# Сложность Алгоритмов (O-нотация)

## O(1) Константная сложность

- Кол-во операций НЕ ЗАВИСИТ от размера входных данных. 10 элементов или 10 миллиардов -- скорость одна и та же.
- Взять книгу с полки по известному номеру. Не важно, сколько всего книг в библиотеке.
- Пример: array[5], dictionary["ключ"], stack.Push().

## O(log n) Логарифмическая сложность

- Кол-во операций растёт очень медленно. Увеличил данные в 1000 раз — время увеличилось на 10 «шагов» ( $2^{10} = 1024$ ).
- Поиск слова в бумажном словаре. Открываем середину и отбрасываешь половину книги. С каждым шагом мы уменьшаем область поиска вдвое.
- Пример: Бинарный поиск в отсортированном массиве.

# Сложность Алгоритмов (O-нотация)

Search for 14:

$$12 > 8$$



1	3	4	7	8	11	12	14	25	30
---	---	---	---	---	----	----	----	----	----

$$12 < 14$$



11	12	14	25	30
----	----	----	----	----

$$12 > 11$$



11	12
----	----

12

# Сложность Алгоритмов (O-нотация)

## **O(n log n) Линеаритмическая сложность**

«Золотой стандарт» для большинства задач, которые нельзя решить за один проход ( $O(n)$ ). Это сложность эффективных алгоритмов сортировки (QuickSort, MergeSort, HeapSort) и многих других задач типа «разделяй и властвуй».

# Сложность Алгоритмов линеаритмич

Аналогия: **Идеальная сортировка бумаг.**

Есть гора из  $n$  бумажных листов, которые надо отсортировать по алфавиту.

**Если  $O(n^2)$  (Пузырёк):** Каждый лист сравнивается с последующим. И так со всеми листами.

**Если  $O(n \log n)$  (Быстрая сортировка):**

**Разделяй ( $\log n$  часть):** Берется один случайный (чаще из середины) лист (опорный элемент) и стопка расскладывается на две кучи: слева — все листы, которые по алфавиту раньше него, справа — все, что позже. С левой и правой кучами проделывается то же самое. Продолжаем делить кучи пополам, пока не останется стопка из одного листа (стопка из одного листа по определению уже отсортирована). Количество таких делений есть  $\log n$ .

**Властвуй ( $n$  часть):** На каждом уровне этого деления нам нужно один раз пройтись по всем  $n$  листам, чтобы раскидать их по двум кучам.

Итого: Мы делаем  $\log n$  «проходов» (уровней деления), и на каждом проходе мы обрабатываем  $n$  элементов. Получается  $n * \log n$ .

# Сложность Алгоритмов линеаритмич

N (элементов)	$n^2$ (Пузырёк)	$n * \log n$ (Быстрая сортировка)	Во сколько раз быстрее
10	100	~33	~3 раза
100	10 000	~664	~15 раз
1 000	1 000 000	~9 965	~100 раз
1 000 000	1 000 000 000 000	~19 931 568	~50 000 раз

# Сложность Алгоритмов (O-нотация)

## O(n) Линейная сложность

- Кол-во операций растёт прямо пропорционально количеству данных. Увеличили данные в 1000 раз — алгоритм стал выполнять в 1000 раз больше операций.
- Найти конкретного человека в очереди, спрашивая каждого по имени с самого начала.
- Пример: `list.Contains()`, поиск в неотсортированном массиве, простой цикл `for` по коллекции.

# Сложность Алгоритмов (O-нотация)

## $O(n^2)$ Квадратичная сложность

- Кол-во операций растёт катастрофически. Увеличили данные в 1\_000 раз — кол-во работы увеличилось в 1\_000\_000 раз.
- Рукопожатия на встрече. Каждый должен пожать руку каждому. Цикл в цикле.
- Пример: Тупые алгоритмы сортировки (пузырёк, вставками), вложенный цикл `for` по одной и той же коллекции.

## $O(2^n)$ Экспоненциальная сложность

Кол-во операций растёт так быстро, что задача становится нерешаемой уже на небольших  $n$ .

Подбор пароля полным перебором. Каждая новая буква умножает время на количество символов в алфавите.

Пример: Рекурсивный расчёт чисел Фибоначчи "в лоб", задача коммивояжёра.

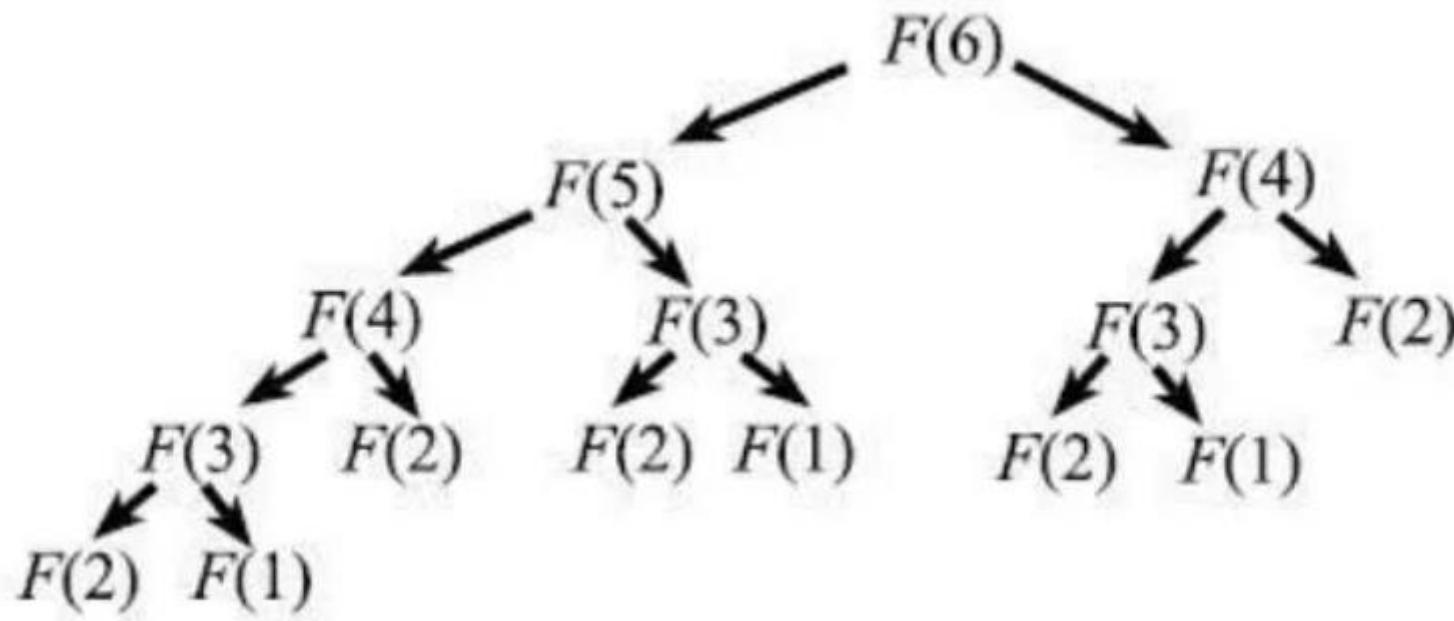
# Сложность Алгоритмов (рекурсия)

**Числа Фибоначчи** — последовательность чисел, где каждое следующее число равно сумме двух предыдущих. В классическом варианте последовательность начинается с 0 и 1.

Пример: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Самый прямой и очевидный способ посчитать N-ное число Фибоначчи ( $F(n) = F(n-1) + F(n-2)$ ).

# Сложность Алгоритмов (рекурсия)



$\text{Fib}(4)$  мы посчитали 2 раза.  $\text{Fib}(3)$  мы посчитали 3 раза.  $\text{Fib}(2)$  мы посчитали 5 раз.

# Сложность Алгоритмов (O-нотация)

**ВИЗУАЛИЗАЦИЯ В ДЕЗМОСЕ**

<https://www.desmos.com/calculator/xmtzsymmia?lang=ru>

# Сложность Алгоритмов (O-нотация)

**Не забудьте посмотреть в коде (`lectionSample` и `lectionSample2`)**

# Массив (System.Array)

Массив – индексированная коллекция элементов **одного типа**, расположенных в **непрерывной** области памяти.

Характеристики:

- Тип: Сылочный (объект в Куче, ссылка на Стеке)
- Размер: Фиксированный, задаётся при создании
- Доступ: Прямой (Random Access) по целочисленному индексу

# Массив (System.Array)

Массив – индексированная коллекция элементов **одного типа**, расположенных в **непрерывной** области памяти.

Временная сложность операций:

- Доступ по индексу  $\text{array}[i]$ :  $O(1)$  — константная, очень быстрая.
- Поиск элемента:  $O(n)$  — линейная, медленная (в худшем случае).
- Вставка/Удаление:  $O(n)$  — линейная, очень медленная (требует сдвига элементов).

# Список (System.Collections.Generic.List<T>)

Список - динамически изменяемая, строго типизированная, индексированная коллекция объектов. Представляет собой массив, который может «растягиваться» и «сжиматься».

Основан на внутреннем массиве (T[]).

*Внимание!!! Это про список в C#. В интернете вы можете наткнуться на определение связного списка – это другое.*

# Список (System.Collections.Generic.List<T>)

Временная сложность операций:

- Доступ по индексу `list[i]`:  $O(1)$  — константная (т.к. под капотом массив).
- Добавление в конец `list.Add()`:  $O(1)$  (амортизированная). В большинстве случаев быстро, но иногда может быть  $O(n)$  из-за необходимости переаллокации внутреннего массива.
- Поиск элемента:  $O(n)$  — линейная.
- Вставка/Удаление:  $O(n)$  — линейная.

# Словарь (System.Collections.Generic.Dictionary<TKey, TValue>)

Словарь - коллекция пар «ключ-значение», где каждый ключ должен быть уникальным. Позволяет ассоциировать одно значение с другим. Использует хэш-функцию для преобразования ключа в индекс внутреннего массива («корзины» или «buckets»), что позволяет избежать последовательного перебора.

Временная сложность операций (в среднем):

- Добавление, удаление, поиск по ключу:  $O(1)$  — константная, так как используется индекс.

# Множество (System.Collections.Generic.HashSet<T>)

Множество - Неупорядоченная коллекция, содержащая уникальные элементы. Также основан на хэш-таблице. Аналогичен Dictionary, но хранит только ключи, без ассоциированных значений.

Временная сложность операций (в среднем):

- Добавление, удаление, проверка наличия (Contains):  $O(1)$  — константная.

# Боль - массивы

Фиксированный размер: Создал на 10 элементов — 11-й не впишнёшь. Нужно заранее знать, сколько данных у тебя будет. А мы *почти никогда* не знаем.

Боль при изменении: Нужно вставить элемент в середину?  
Двигаем всю правую часть массива руками. Нужно удалить?  
Двигаем влево. (*массив непрерывный*)

*Массив — это просто, быстро, но негибко. Это фундамент, но жить в подвале неудобно.*

# List<T> Иллюзия бесконечности

`list.Add(item)` — добавляй сколько влезет.

`list.Insert(index, item)` – вставляй куда хочешь.

`list.Remove(item)` — удаляй и не думай о дырках.

# List<T> Иллюзия бесконечности

Под капотом у List<T> лежит ОБЫЧНЫЙ МАССИВ!

Переменная List<T> myList это ссылка, которая лежит на Стеке.  
Эта ссылка указывает на объект в Куче.

А внутри этого объекта в Куче лежит приватное поле — массив T[] \_items и счётчик int \_size.

```
public class List<T> : IList<T>, IList, IReadOnlyList<T>
{
    private const int DefaultCapacity = 4;

    internal T[] _items; // Do not rename (binary serialization)
    internal int _size; // Do not rename (binary serialization)
    internal int _version; // Do not rename (binary serialization)
```

# List<T> Иллюзия бесконечности

Что происходит, когда мы делаем myList.Add(5)?

List смотрит: `_size < _items.Length`

Да — он просто кладёт элемент в массив и увеличивает счётчик `_size`. Это быстро.  $O(1)$ .

Если нет —List понимает, что элемент не влезет. Он:

Создаёт в Куче НОВЫЙ массив, в два раза больше старого (`Capacity * 2`).

КОПИРУЕТ туда все элементы из старого массива. Это МЕДЛЕННО.  $O(n)$ .

Выкидывает старый массив на мороз (GC потом подберёт).

И только потом добавляет ваш новый элемент.

Вывод: List — это не бесконечная лента. Это серия массивов всё большего и большего размера. Удобство «бесконечного» добавления мы оплачиваем редкими, но дорогими операциями копирования.

# List<T> Иллюзия бесконечности (пруфы)

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public void Add(T item)
{
    _version++;
    T[] array = _items;
    int size = _size;
    if ((uint)size < (uint)array.Length)
    {
        _size = size + 1;
        array[size] = item;
    }
    else
    {
        AddWithResize(item);
    }
}
```

# List<T> Иллюзия бесконечности (пруфы)

```
private void AddWithResize(T item)
{
    Debug.Assert(_size == _items.Length);
    int size = _size;
    Grow(capacity: size + 1);
    _size = size + 1;
    _items[size] = item;
}

internal void Grow(int capacity)
{
    Capacity = GetNewCapacity(capacity);
}
```

# List<T> Иллюзия бесконечности (пруфы)

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
private int GetNewCapacity(int capacity)
{
    Debug.Assert(_items.Length < capacity);

    int newCapacity = _items.Length == 0 ? DefaultCapacity : 2 * _items.Length;

    // Allow the list to grow to maximum possible capacity (~2G elements) before encountering overflow.
    // Note that this check works even when _items.Length overflowed thanks to the (uint) cast
    if ((uint)newCapacity > Array.MaxLength) newCapacity = Array.MaxLength;

    // If the computed capacity is still less than specified, set to the original argument.
    // Capacities exceeding Array.MaxLength will be surfaced as OutOfMemoryException by Array.Resize.
    if (newCapacity < capacity) newCapacity = capacity;

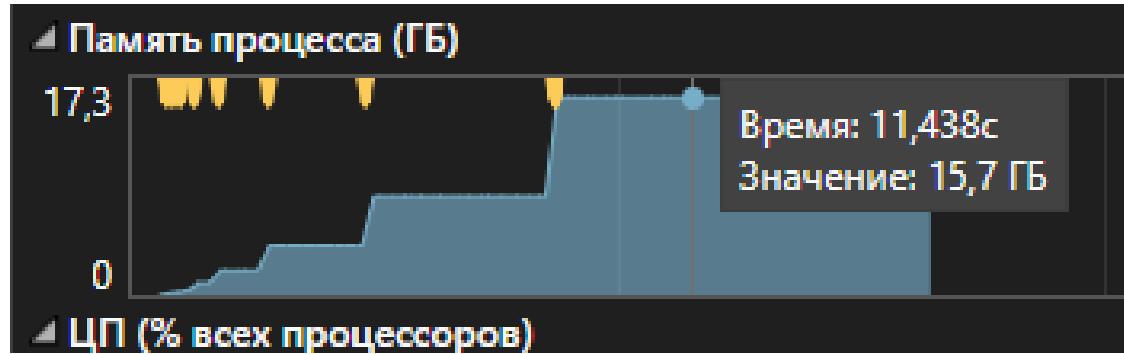
    return newCapacity;
}
```

# List<T> Иллюзия бесконечности (пруфы)

```
public int Capacity
{
    get => _items.Length;
    set
    {
        if (value < _size)
        {
            ThrowHelper.ThrowArgumentOutOfRangeException(ExceptionArgument.value, ExceptionResource.ArgumentOutOfRange_SmallCapacity);
        }

        if (value != _items.Length)
        {
            if (value > 0)
            {
                T[] newItems = new T[value];
                if (_size > 0)
                {
                    Array.Copy(sourceArray: _items, destinationArray: newItems, _size);
                }
                _items = newItems;
            }
            else
            {
                _items = s_emptyArray;
            }
        }
    }
}
```

# List<T> вор оперативки



MyListProject