

# Структуры данных

# Что такое Big O Notation? (Анализ сложности алгоритмов)

Представьте, что у вас есть две программы, которые решают одну и ту же задачу. Какая из них "лучше"? Та, что работает быстрее. Но как измерить скорость?

**Запускать и засекать время?** Плохой способ. Результат будет зависеть от мощности компьютера, языка программирования и даже от загруженности системы.

**Считать количество операций?** Уже лучше! Это более объективный показатель.

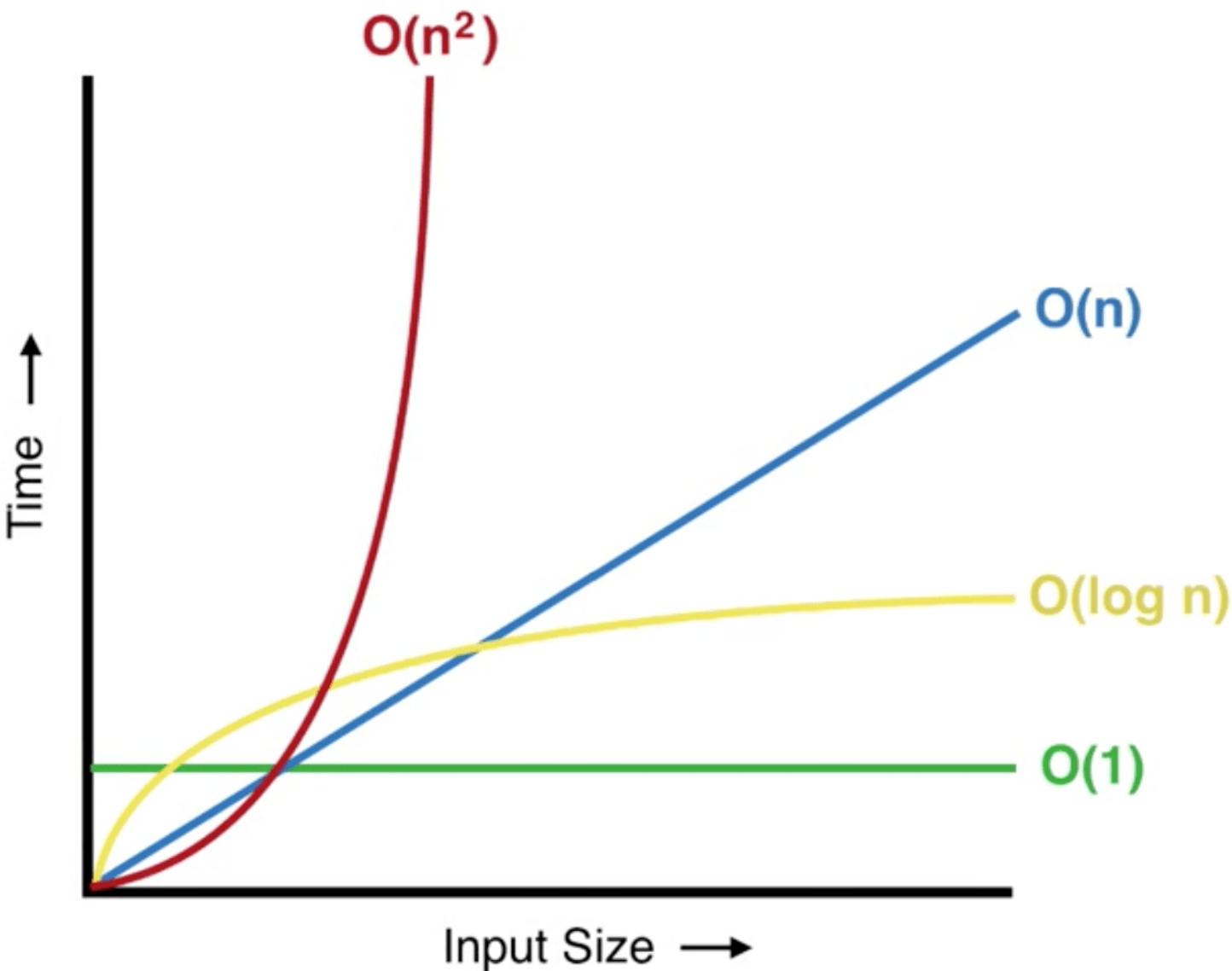
- **Big O Notation** — это способ описать, насколько сильно **растет** время выполнения (или количество операций) алгоритма при **увеличении** объема входных данных.
- Проще говоря, Big O показывает, станет ли ваша программа "тормозить", когда вы дадите ей обработать 1000 элементов вместо 10. Это "язык" для описания эффективности алгоритмов.

# Самые частые виды Big O



- **$O(1)$  — Константная сложность**
- **$O(\log n)$  — Логарифмическая сложность**
- **$O(n)$  — Линейная сложность**
- **$O(n^2)$  — Квадратичная сложность**

## Big O Notation



# $O(1)$ — Константная сложность

- Алгоритм работает одинаково быстро, независимо от размера входных данных.
- **Пример:** Взять первый элемент из списка. Неважно, 10 в нем элементов или 10 миллионов, это всегда одна операция.



```
1 def get_first_element(data: list):  
2     return data[0] # Всегда одна операция
```

# $O(\log n)$ – Логарифмическая сложность

Время выполнения растет очень медленно. При увеличении данных вдвое, время увеличивается всего на одну операцию.

**Пример:** Бинарный поиск в отсортированном массиве  
(как в аналогии со справочником).

# $O(n)$ — Линейная сложность

Время выполнения растет прямо пропорционально количеству данных. Если данных в 10 раз больше, то и времени потребуется примерно в 10 раз больше.

- **Пример:** Найти самый большой элемент в списке. Нужно пройтись по каждому элементу один раз.

# $O(n^2)$ – Квадратичная сложность

Время выполнения растет очень быстро. Если данных стало в 2 раза больше, время увеличится в 4 раза ( $2^2$ ).

**Пример:** Найти дубликаты в списке с помощью вложенных циклов.

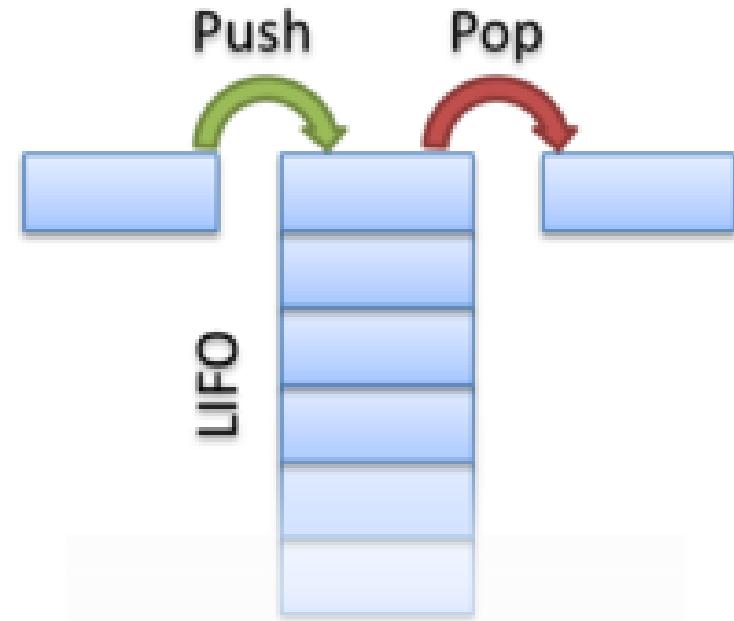
```
def has_duplicates(data: list):
    for i in range(len(data)):
        for j in range(len(data)):
            if i != j and data[i] == data[j]:
                return True # Найден дубликат
    return False
```

# Stack (Стек)

Last In First Out (LIFO)

# Стек (Stack) - LIFO

- Стек — это линейная структура данных, работающая по принципу «последним пришёл — первым ушёл» (LIFO - Last In, First Out).
- Добавление новых элементов и удаление существующих происходит только с одного конца, называемого вершиной стека, что можно сравнить со стопкой тарелок или колодой карт.



# Основные принципы и операции

- **Принцип LIFO:** Элемент, который был добавлен в стек последним, будет первым из него извлечён.
- **Вершина стека:** Это единственный конец, где можно выполнять операции. Доступ к другим элементам напрямую невозможен.
- **Основные операции:**
  - **Push (Вставка):** Добавление нового элемента на вершину стека.
  - **Pop (Удаление):** Извлечение и удаление верхнего элемента из стека.
  - **Peek (Просмотр):** Возвращение верхнего элемента без его удаления.
  - **IsEmpty (Проверка на пустоту):** Проверка, пуст ли стек.

# Задача: Реализуй стек на Python

**Цель:** Создать класс Stack, который поддерживает основные операции со стеком:

- Добавление элемента в верхушку стека (**push**)
- Удаление и возврат верхнего элемента (**pop**)
- Просмотр верхнего элемента без удаления (**peek** или **top**)
- Проверка, пуст ли стек (**is\_empty**)
- Получение размера стека (**size**)

## Дополнение к предыдущей задаче

- К классу стек **Stack** добавьте метод **get\_min()** который возвращает минимальный элемент в стеке.

# Дополнение к предыдущей задаче

- К классу стек **Stack** добавьте метод **get\_min()** который возвращает минимальный элемент в стеке.

**Примечание:** реализуйте метод **get\_min()** так чтобы работал в **O(1)** времени

## Задача: Удаление всех смежных дубликатов в строке

Дана строка  $s$ , состоящая из строчных латинских букв.

За одну операцию мы можем удалить два одинаковых соседних символов.

После удаления таких символов строка может образовать новые пары одинаковых соседних символов — их тоже нужно удалять. Процесс продолжается до тех пор, пока больше нет соседних одинаковых символов.

Верни окончательную строку после всех возможных удалений. Гарантируется, что результат уникален.



- 1 Ввод: `s = "abbaca"`
- 2 Вывод: `"ca"`
- 3 Объяснение:
- 4 - Сначала удаляем `"bb"`, получаем `"aaca"`.
- 5 - Затем удаляем `"aa"`, получаем `"ca"`.



- 1 Ввод: `s = "azxxzy"`
- 2 Вывод: `"ay"`
- 3 Объяснение:
- 4 - Удаляем `"xx"` → `"azzy"`
- 5 - Удаляем `"zz"` → `"ay"`

# Задача: Окончательные цены со специальной скидкой в магазине

Дан целочисленный массив `prices`, где `prices[i]` — цена  $i$ -го товара в магазине.

В магазине действует специальное предложение: для каждого товара ты можешь получить скидку, равную цене ближайшего следующего товара, у которого цена меньше или равна текущей.

Если такого следующего товара нет, то скидка не применяется, и ты платишь полную цену.

Верни массив `answer`, где `answer[i]` — окончательная цена на  $i$ -й товар после применения скидки.



1 Ввод: `prices = [8,4,6,2,3]`  
2 Вывод: `[4,2,4,2,3]`

3

4 Объяснение:

- 5 - Для товара 0 (цена 8): ближайший следующий товар со скидкой – цена 4 (индекс 1)  $\rightarrow 8 - 4 = 4$
- 6 - Для товара 1 (цена 4): следующая цена  $\leq 4$  – это 2  $\rightarrow 4 - 2 = 2$
- 7 - Для товара 2 (цена 6): следующая цена  $\leq 6$  – это 2  $\rightarrow 6 - 2 = 4$
- 8 - Для товара 3 (цена 2): нет следующей цены  $\leq 2 \rightarrow$  остаётся 2
- 9 - Для товара 4 (цена 3): последний товар  $\rightarrow$  остаётся 3



1 Ввод: prices = [1,2,3,4,5]

2 Вывод: [1,2,3,4,5]

3 Объяснение: Нет товара с меньшей или равной ценой после любого из них → без скидок.



1 Ввод: `prices = [10,1,1,6]`

2 Вывод: `[9,0,1,6]`

3 Объяснение:

4 -  $10 - 1 = 9$  (берём первый товар со скидкой)

5 -  $1 - 1 = 0$  (следующая единица)

6 -  $1 \rightarrow$  нет следующей  $\leq 1 \rightarrow$  остаётся 1

7 - 6 → остаётся 6

# Правильные скобки

На вход подаётся одна строка с кодом программы, нужно проверить правильность расстановки парных скобок.

Возможные пары скобок: [ ], { }, ( ).

- Эту задачу следует решить с использованием структуры данных **stack**.
- Если на Python, то с использованием методов списка **append** и **pop**.



```
1 s = '[]{}()'  
2  
3 mydict = {")": "(", "]": "[", "}": "{"}  
4 mystack = []  
5  
6 result = True  
7  
8 for i in s:  
9     if i in mydict:  
10         if not mystack or mystack.pop() != mydict[i]:  
11             result = False  
12             break  
13     else:  
14         mystack.append(i)  
15  
16 print(result)
```

# Queue (Очереди)

First In First Out (FIFO)

# Очередь (Queue) - FIFO

- Очередь — это линейная структура данных, которая работает по принципу FIFO (англ. "first-in, first-out" — "первый пришел, первый вышел").
- Элементы добавляются в конец очереди (хвост), а удаляются из начала (голова). Примером может служить обычная очередь людей, где первым обслуживаются те, кто пришел раньше всех.
- **Реализация:** Может быть реализована с помощью массивов

# Основные операции

- **Добавление (enqueue):**
- Новый элемент помещается в конец очереди.
- **Извлечение (dequeue):**
- Удаляется самый первый добавленный элемент из начала очереди.
- **Реализация:** Может быть реализована с помощью массивов (часто с использованием кольцевого буфера) или связанных списков

## **Задача:** Реализация очереди с использованием двух стеков

Твоя задача — реализовать структуру данных очередь (Queue), используя только два стека.

Ты должен поддерживать следующие операции:

- `push(x)` — добавить элемент `x` в конец очереди.
- `pop()` — удалить и вернуть первый элемент из очереди.
- `peek()` — вернуть первый элемент очереди без удаления.
- `empty()` — возвращает `True`, если очередь пуста, иначе `False`.

**Ограничение:** ты можешь использовать только структуру стека, то есть добавлять и удалять элементы только с одного конца (LIFO). Но поведение всей структуры должно быть как у очереди (FIFO).

# Number of Students Unable to Eat Lunch

- <https://leetcode.com/problems/number-of-students-unable-to-eat-lunch/description/?envType=problem-list-v2&envId=queue>

# Первый уникальный символ в строке

Дана строка s, состоящая из строчных латинских букв.

Нужно найти индекс первого символа, который встречается только один раз в строке.

Если такого символа нет, верни -1.

# Хеширование

(Hash map, Hash table, Hashing)

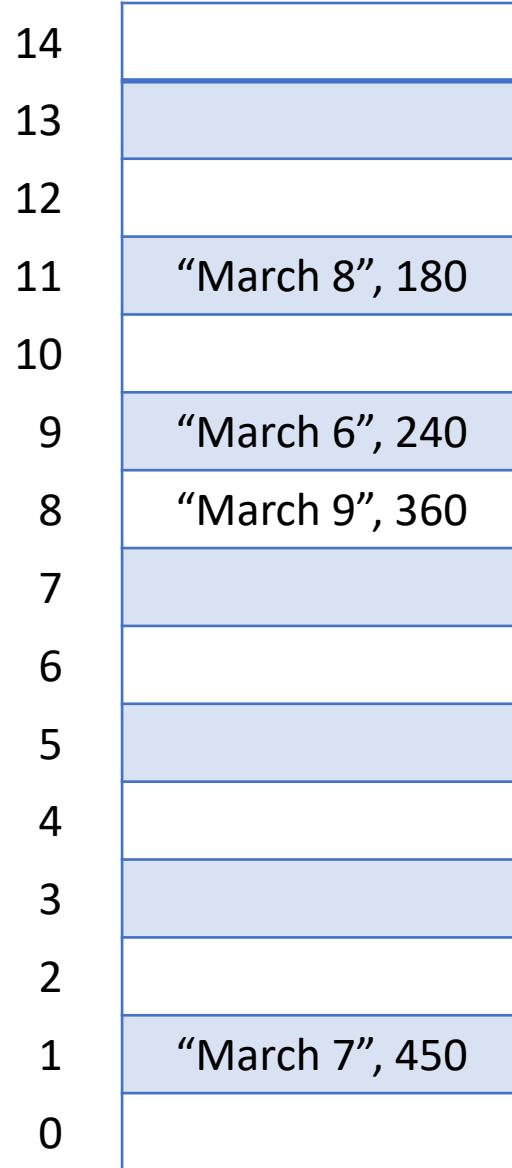
# Что такое хеширование

Хеширование — это стратегия управления данными. Представьте, что пытаетесь найти иголку в стоге сена — это кажется довольно сложной задачей, верно?

Хеширование — широко используемый в информатике метод, позволяющий эффективно хранить и извлекать данные. Диапазон значений ключа преобразуется в диапазон значений индекса с помощью специальной функции, называемой хеш-функцией.

Основная цель хеширования — минимизировать время поиска независимо от размера данных и их сложности. Использование хеш-кода в качестве индекса позволяет получить практически мгновенный доступ к данным и хранить их таким образом, чтобы их можно было находить быстро и легко. Это особенно важно при работе с большими массивами данных, поскольку помогает обеспечить эффективность процесса поиска.

Date	Price
March 6	240
March 7	450
March 8	<b>180</b>
March 9	360

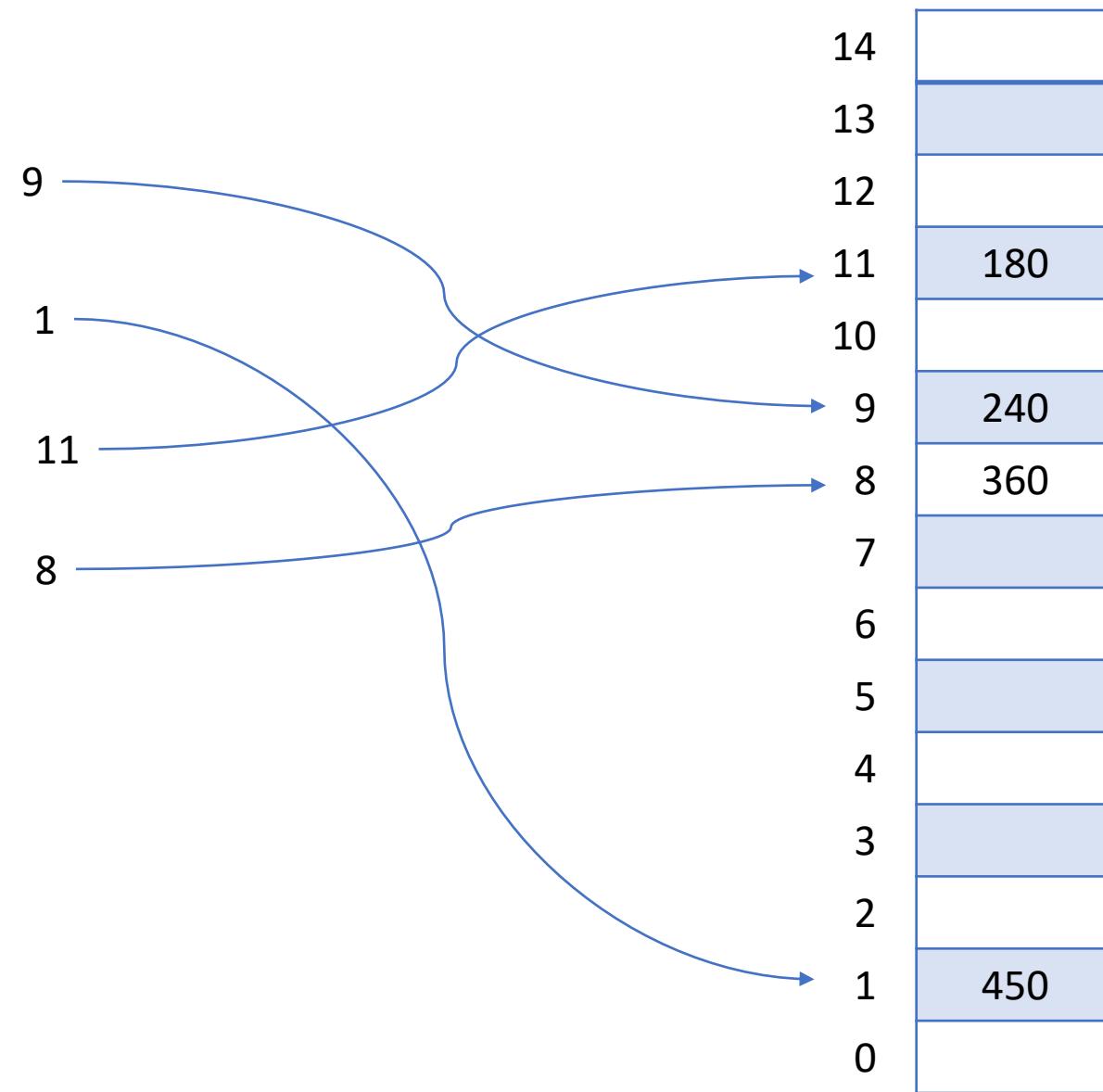


March 6 → Hash Function

March 7 → Hash Function

March 8 → Hash Function

March 9 → Hash Function



# Хеш функция

March 6 → Hash Function

M	109
A	97
R	114
C	99
H	104
	32
6	54
SUM	609

$$\text{MOD}(609,10) \rightarrow 609\%10 = 9$$

Home	ASCII Table ▾	ASCII Characters	ASCII Art	Articles	FAQ	Facts	History	Feedback
96	140	60	01100000					&#96;
97	141	61	01100001	a				&#97;
98	142	62	01100010	b				&#98;
99	143	63	01100011	c				&#99;
100	144	64	01100100	d				&#100;
101	145	65	01100101	e				&#101;
102	146	66	01100110	f				&#102;
103	147	67	01100111	g				&#103;
104	150	68	01101000	h				&#104;
105	151	69	01101001	i				&#105;
106	152	6A	01101010	j				&#106;
107	153	6B	01101011	k				&#107;
108	154	6C	01101100	l				&#108;
109	155	6D	01101101	m				&#109;
110	156	6E	01101110	n				&#110;
111	157	6F	01101111	o				&#111;
112	160	70	01110000	p				&#112;
113	161	71	01110001	q				&#113;
114	162	72	01110010	r				&#114;
115	163	73	01110011	s				&#115;

# Задачи

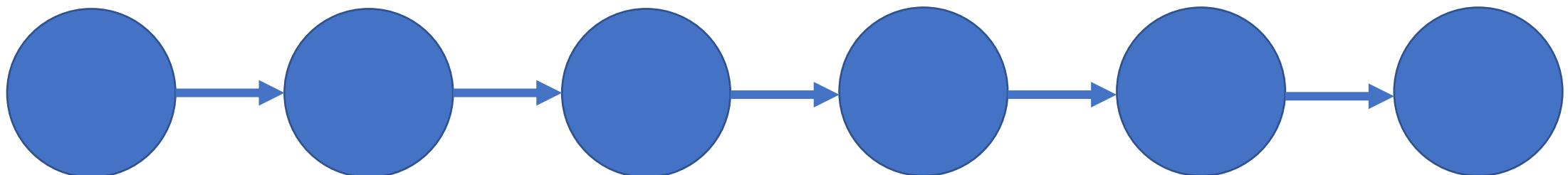
- <https://leetcode.com/problems/valid-anagram/description/?envType=problem-list-v2&envId=hash-table>
- <https://leetcode.com/problems/two-sum/description/?envType=problem-list-v2&envId=hash-table>

# Linked List

Связанные списки (указатели, узлы и их применение)

# Что такое связанные списки

- Связанный список — это линейная структура данных, состоящая из последовательности элементов. В связанном списке каждый элемент, который также называют узлом, содержит данные и ссылку на следующий элемент в последовательности.  
Представьте цепочку узлов, каждый из которых имеет значение и указатель, направляющий вас к следующему узлу.



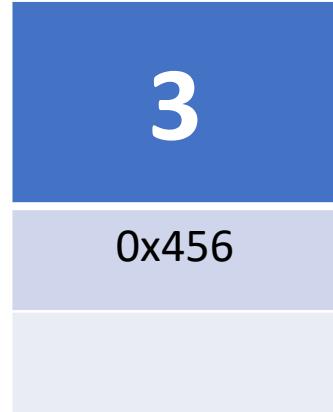
# Node (Узел)

- Узел — основной компонент связанного списка.
- Узел выступает в роли контейнера, который хранит важную информацию, а также поддерживает ссылку, обычно известную как `next`, на следующий узел в связанном списке.

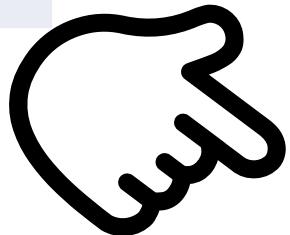
# Память



# Память



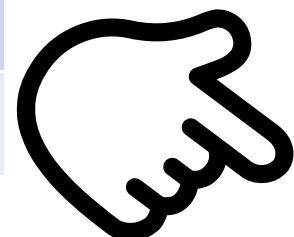
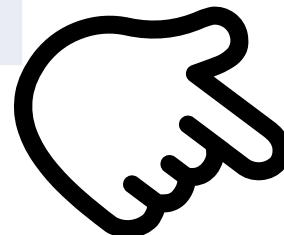
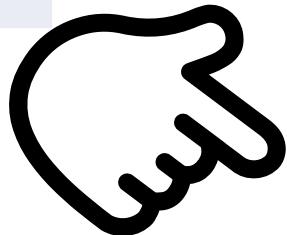
# Память



# Память



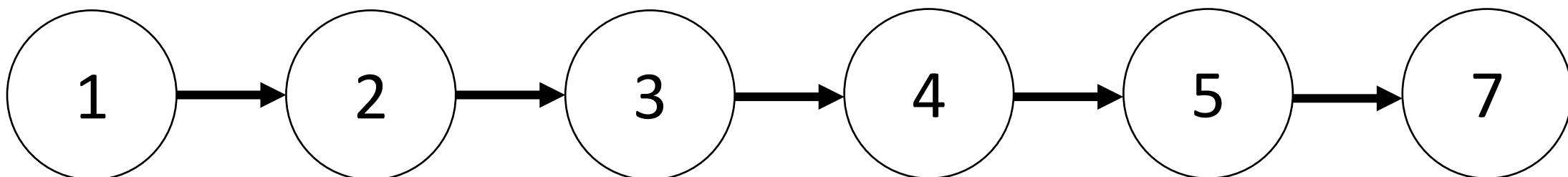
# Память



# Пример

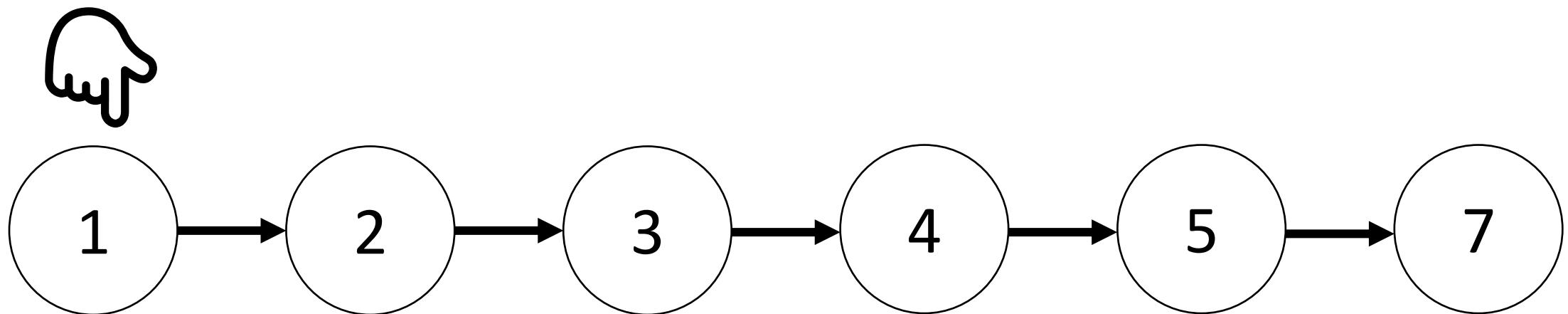


```
1 class Node:  
2     def __init__(self, val=0):  
3         self.val = val  
4         self.next = None
```



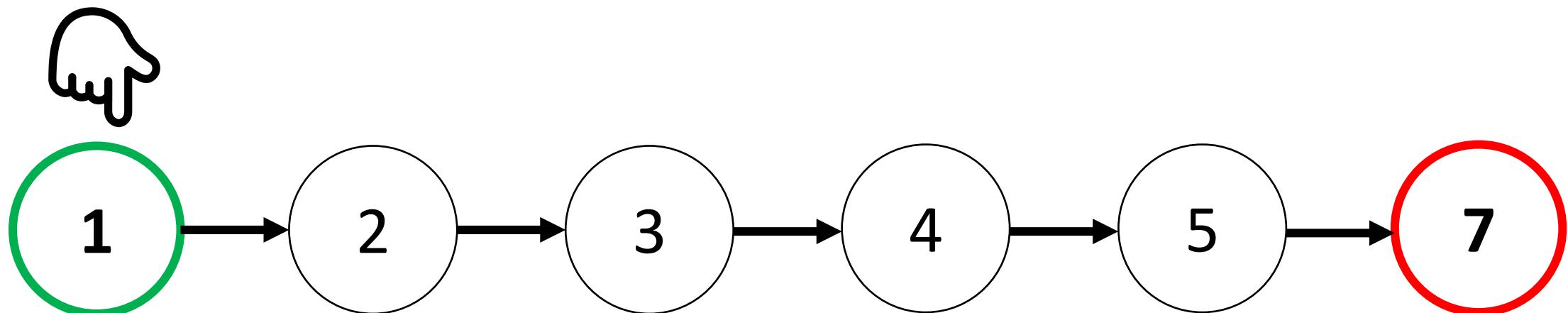
# Голова

- Голова в связанным списке — это начальный узел, служащий входом в список. Это указатель на первый узел, содержащий данные.



# Обход

- Обход — операция, которая позволяет перемещаться по списку и получать доступ к каждому узлу. Она играет важную роль в переборе элементов списка, начиная работать с головы и продолжая до тех пор, пока не будет достигнут последний узел.





```
1 def print_list(self):  
2     temp = self.head  
3     while temp:  
4         print(temp.data)  
5         temp = temp.next
```