Массивы и списки

education

Базовые структуры данных

1

Разберём линейные структуры данных

2

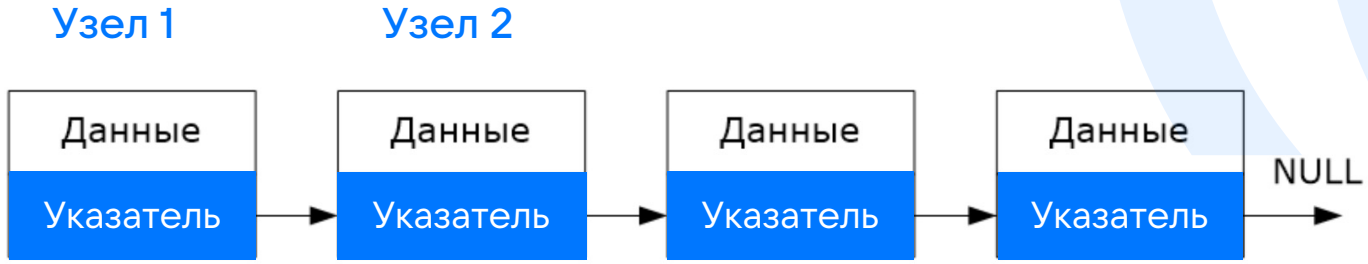
Поговорим о сложности основных операций в каждой из структур

3

Поймём отличия и особенности каждой из них

Введение

- Уже изучили О-нотацию и понимаете, как определить сложность того или иного алгоритма.
- В этом уроке мы будем экстраполировать эти знания на самые распространённые структуры данных.
- Исходя из О большого будем понимать, в какой ситуации какая структура будет лучше.



Массивы

Непрерывная область памяти заданного размера

Массив подразумевает под собой хранение однотипных данных, расположенных друг за другом в памяти

Доступ к элементу массива осуществляется посредством целочисленного индекса

Обращение к ячейке по индексу за константное время

Массивы

- Ёмкость массива (capasity)
- Размер (size) — количество элементов, находящихся в массиве
- Тип данных

Array:

size int
capacity int
type int

Структура массива

64 байта

Для массива из 8 элементов с типом int64 при условии, что на моей архитектуре 1 элемент с типом int занимает 8 байт, в памяти будет последовательно аллоцировано 64 байта.

- Допустим, нулевой элемент начался с ячейки памяти с номером 1 000.
- Тогда байты с 1 000 по 1 007 будут принадлежать первому элементу массива. Байты с 1 008 — второму и т. д.
- Зная, что все элементы располагаются последовательно, легко получить доступ к каждому из них.
- Элемент номер 4 можно найти по формуле: нулевой элемент плюс произведение индекса на размер каждой ячейки 1 000 + 4 × 8 = 1 032.

Действия над массивом

1. append() — вставка
2. get(index) — получение элемента по индексу
3. capacity — ёмкость массива, количество элементов, которое он может в себя вместить
4. size (size ≤ capacity) — количество заполненных ячеек массива

Следующий код призван лишь проиллюстрировать принцип работы массива

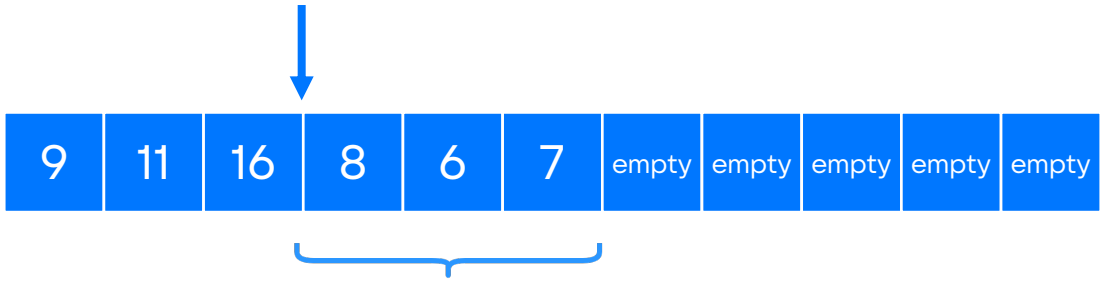
```
from typing import Any


class Array:
    def __init__(self, capacity: int, el_type: Any):
        self.array = [None] * capacity
        self.el_type = el_type


    def get(self, index: int):
        return self.array[index]


    def append(self, index: int, value: Any):
        # проверяем тип данных
        if not isinstance(value, self.el_type):
            raise ValueError(f"value is {type(value)}; but must be {self.el_type}")


        self.array[index] = value
```

Основные операции

1. Получение данных O(1)
2. Вставка в середину O(n)
3. Вставка в конец O(1)
4. Удаление O(n)

Вставка/удаление из середины массива

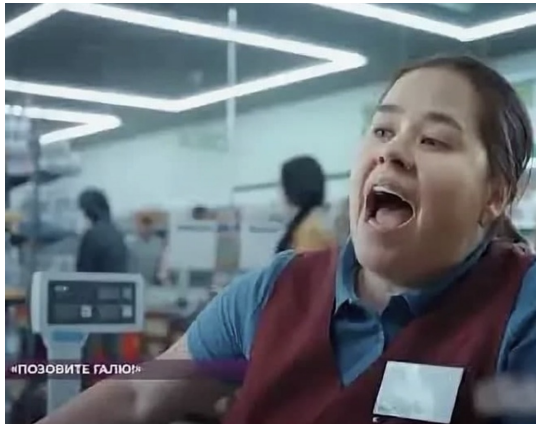
- Допустим, мы хотим вставить новый элемент в середину массива.
- Так как все элементы должны располагаться друг за другом, нам необходимо последовательно переместить каждый элемент вперёд.
- По схожей логике реализуется удаление из середины массива.



Элементы, которые необходимо перебрать для вставки в середину

Вставка/удаление из середины массива

- Массив отлично подходит, когда операций на получение элемента значительно больше, чем вставок, идеально для readonly-хранилища.
- Мы заранее должны знать его размер.
- Не подходит в ситуациях, когда необходимо производить много вставок в середину.



- Пример — организация хранения товаров в памяти
- Один раз добавили все товары
- Чтобы узнать цену товара, вы производите только выборку
- Операции вставки происходят гораздо реже

Саморасширяющийся массив

Саморасширяющийся массив

1

Как быть, если мы не всегда можем заранее знать размер массива?

2

При этом есть необходимость получения элементов по индексу

3

По-прежнему хотим O(1) при выборке

Как должна работать такая структура данных

- У массива есть два основных параметра: size и capacity.
- Если size становится равным capacity, то мы больше не можем вставлять в массив ничего.
- Следовательно, нам нужен алгоритм увеличения capacity.
- Этот процесс называется реалокацией памяти.

- Нам нужно заново найти свободное непрерывное пространство в памяти, чтобы мы могли перенести наши уже существующие элементы последовательно и чтобы там было место под новый элемент.
- Скопировать все значения из старого массива в новый.
- Удалить старый массив, чтобы избежать утечки по памяти.
- Попробуем реализовать похожую на описанную выше процедуру в коде.

Структура данных

```
import ctypes

class DynamicArray(object):
  def __init__(self):
    self.count = 1
    self.size = 0
    self.capacity = 1
    self.array = self.make_array(self.capacity)

  def cap(self):
    return self.capacity

  def append(self, element):
    if self.size == self.capacity:
      self.resize()

    self.array[self.size] = element
    self.size += 1

  def resize(self):
    new_cap = self.capacity * 2
    new_array = self.make_array(new_cap)

    for i in range(self.size):
      self.count += 1    # 16384
      new_array[i] = self.array[i]

    self.array = new_array
    self.capacity = new_cap
```

```
fast
slow: 6
fast
fast
fast
fast
slow: 11
fast
fast
fast
fast
slow: 16
fast
fast
fast
fast
slow: 21
fast
fast
fast
fast
slow: 26
fast
fast
fast
fast
slow: 31
```

- Сделаем capacity равным единице при инициализации.
- Попробуем понять, как лучше увеличивать ёмкость массива.
- Попробуем увеличивать на 5 всякий раз, когда мы пытаемся вставить элемент в заполненный массив.

Сложность реалокации

Должны различать две ситуации вставки в конец массива:

- когда нам не надо увеличивать capacity — сложность O(1);
- когда нужно увеличивать, мы копируем все значения в новый массив, а значит, сложность стремится к O(n).

- Как тогда правильно подсчитать сложность, понимая, что O(n) будет далеко не всегда?
- Проведём множество операций вставки, подсчитаем общее количество элементарных действий и время на их выполнение и разделим общую сложность на количество операций.
- Такая усреднённая сложность называется амортизированной сложностью, а анализ называется амортизационным.

Сложность реалокации

- При заполнении массива 10 000 элементов метод resize будет вызываться «2 000 раз». Это не совсем то, что нам нужно.
- При увеличении ёмкости на 10 элементов — 1 000 реалокаций и т. д.
- Сама по себе идея увеличивать на константу не очень хороша, так как чем больше элементов в массиве, тем больше будет реалокаций, а амортизационная сложность будет стремиться к O(n).
- Самым оптимальным будет увеличивать сложность в константное значение, например, в 2 раза.
- Количество вызовов функции resize будет всего 15.
- Количество простых операций при увеличении capacity на 5 = 9 997 001.
- Количество простых операций при увеличении capacity в 2 раза = 16 384.

```
def resize(self):
  new_cap = self.capacity * 2
  new_array = self.make_array(new_cap)

  for i in range(self.size):
     self.count += 1    # 16384
     new_array[i] = self.array[i]

  self.array = new_array
  self.capacity = new_cap
```

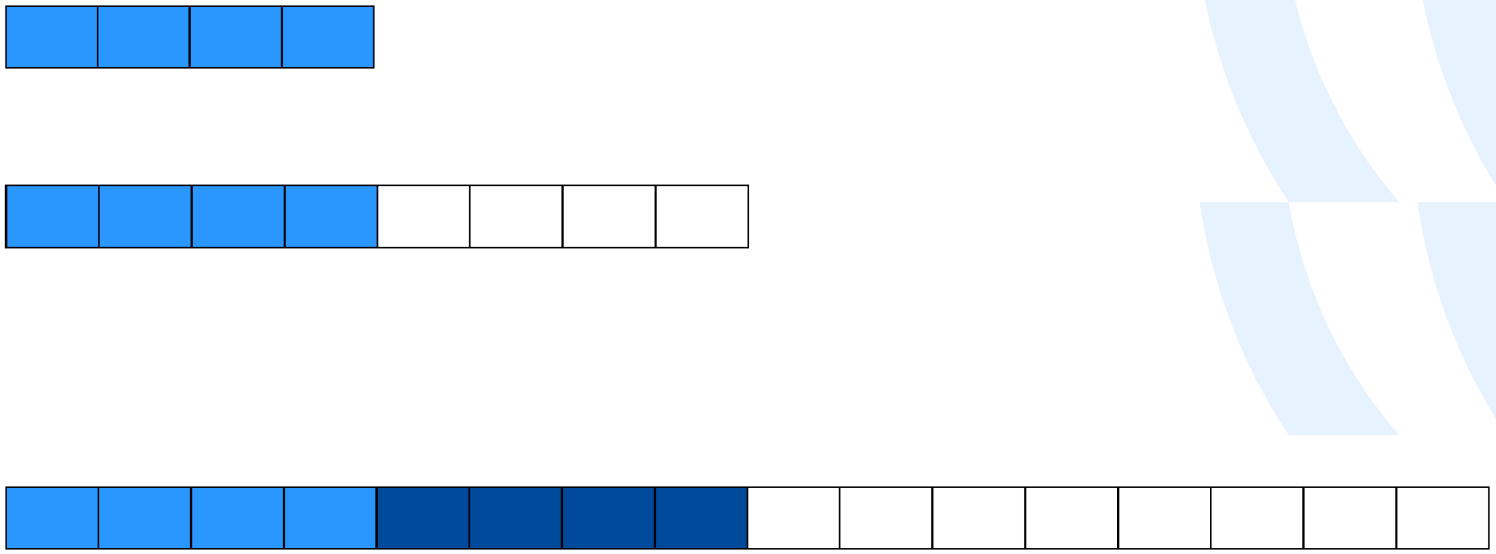
```
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
```

```
def resize(self):
  new_cap = self.capacity + 5
  new_array = self.make_array(new_cap)

  for i in range(self.size):
     self.count += 1    # 16384
     new_array[i] = self.array[i]

  self.array = new_array
  self.capacity = new_cap
```

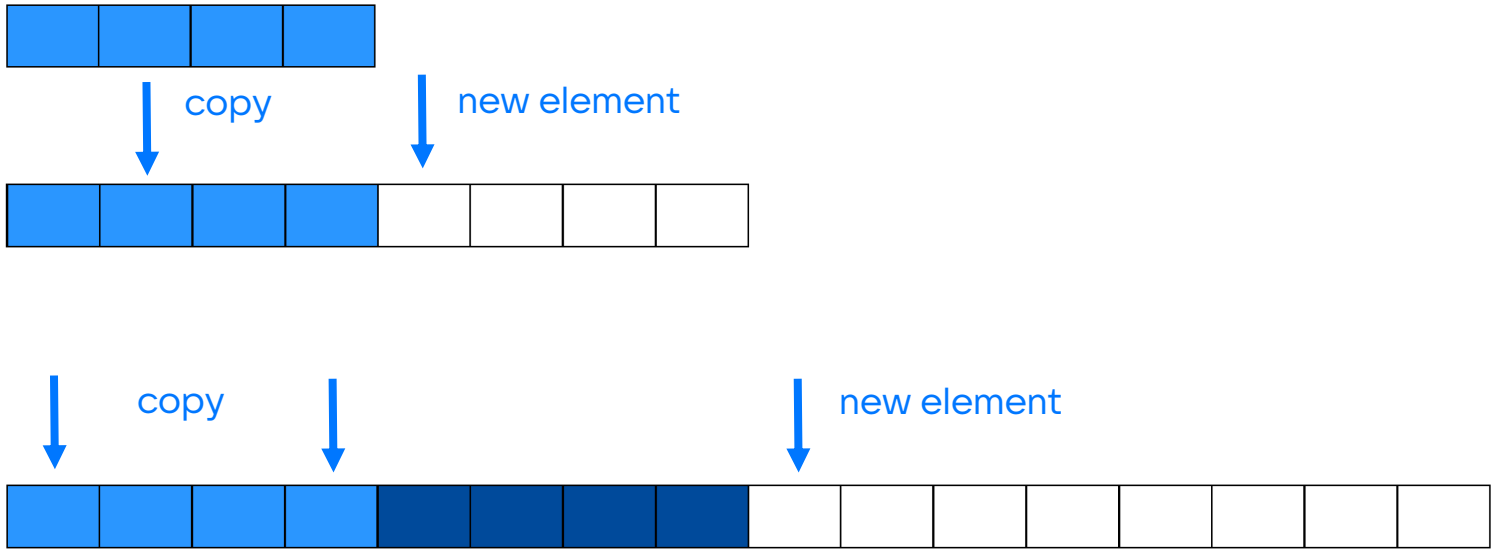
Краткое резюме по амортизационной сложности реализации в два раза

- До вызова метода resize, когда у нас есть место в массиве, мы произвели n вставок, каждая из которых занимала O(1).
- При реалокации мы произвели вставку, которая заняла O(n).
- В итоге у нас есть n + 1 операция, которые у нас заняли 2n (одно n до вызова resize, другое n непосредственно в момент вызова resize).
- Всё это можно представить в виде 2n ÷ (n + 1), что в нотации O большое эквивалентно O(1).

- Необязательно знать конечный размер массива
- При превышении capacity создаётся новый массив с capacity ×2



Аллокация памяти происходит каждый раз, когда мы пытаемся вставить элемент в массив, в котором size = capacity.



Помимо аллокации памяти, необходимо скопировать все элементы из предыдущего массива, вставка в таком случае — O(n).

Как удалять

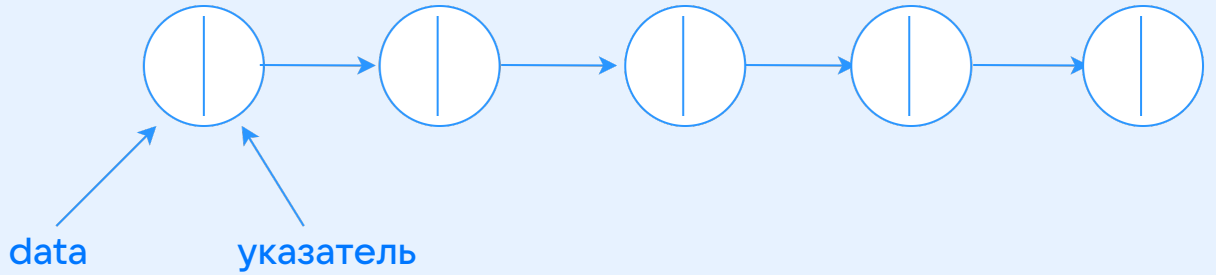
- Первое, что приходит на ум, — раз мы при добавлении увеличиваем ёмкость в два раза, то, возможно, при удалении надо следовать той же стратегии и уменьшать capacity в два раза, когда массив освободился наполовину?

- Тут надо быть аккуратным, сейчас мы почти попали в ловушку. Дело в том, что если после уменьшения capacity в два раза сразу последует вставка, то нам придётся вновь аллоцировать в два раз больше памяти. А если эти операции будут повторяться? То мы рискуем получить в своём алгоритме сложность O(n).

- Уменьшают в два раза объём, когда справедливо равенство size = capacity ÷ 4. То есть когда реальное количество элементов в массиве в 4 раза меньше, чем его ёмкость, тогда уменьшают capacity в 2 раза.

Связный список

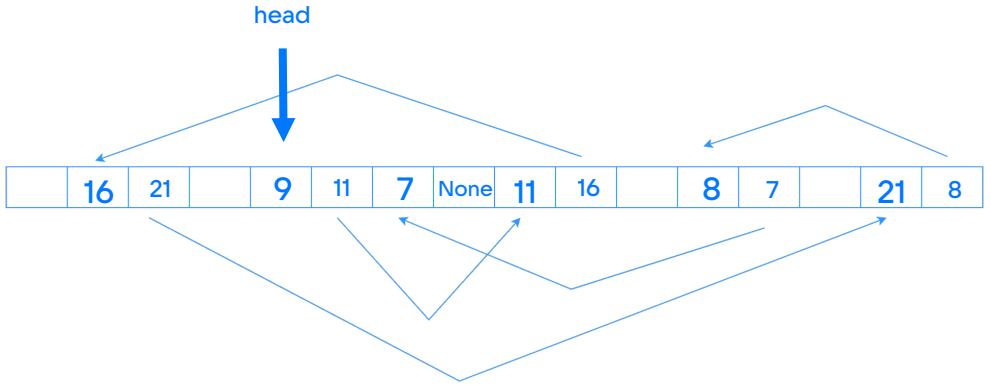
- Однонаправленный (односвязный)
- Двунаправленный (двусвязный)



Однонаправленный список

Структура отличается от изученного нами массива.

- У нас есть какой-то участок памяти. Где-то в нём хранится наш первый элемент списка, его принято называть head — головой списка.
- Список устроен таким образом, что head знает, где хранится второй элемент списка.
- Второй элемент знает, где хранится третий, и так далее.
- Последний элемент вместо указателя на следующий хранит в себе None (null, nil — в зависимости от языка). Так мы понимаем, что это последний элемент.
- Получившаяся структура, в которой каждый элемент знает, где хранится следующий, называется односвязным списком.
- Важно понимать, что в списке нет произвольного доступа по индексу к узлам, как в массивах, а это означает, что, чтобы найти элемент, надо пройти по всему списку.



Однонаправленный список

- Нет необходимости располагать последовательно элементы, что дает ряд преимуществ перед массивом (и некоторые ограничения).
- Каждый узел хранит в себе, помимо собственных данных, ссылку на следующий элемент.
- Аллоцирует память ровно столько, сколько элементов в себе содержит, плюс указатели на следующие элементы.
- Для вставки в любую точку списка необходимо лишь изменить ссылки у рядом стоящих элементов.

Расположение в памяти

В отличие от массива, нет необходимости хранить данные последовательно

Абстрактное представление в коде

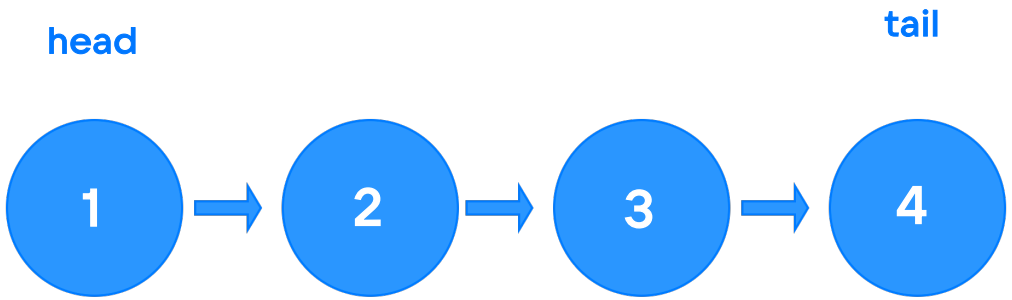
- Каждый элемент списка мы будем называть узлом или нодой от Node.
- Узел — основная часть списка, обычно определяющаяся классом или структурой.
- Структура каждого элемента представляет из себя какую-то полезную информацию: data и указатель на следующий элемент.
- Голова списка (Head): указатель на первый узел в списке. Это «начальная точка», откуда начинается список.
- Tail — указатель на последний узел списка. В простейших однонаправленных списках на него обычно не содержится отдельного указателя, но иногда он может быть полезен для оптимизации некоторых операций.
- Сам список будет представлять из себя структуру в виде головы и размера списка size, иногда добавляют указатель на последний элемент tail.

```
Node {
        data int
        next Node
}
```

Структура каждого узла

```
LinkedList {
        head Node
        tail Node
        size int
}
```

В общем виде список выглядит так



Схематичная иллюстрация односвязанного списка без привязки к памяти

Вставка в начало списка

- Самой простой операцией по добавлению элемента является вставка в начало списка.
- Нам просто нужно переопределить head.
- Три действия за константное время приводят эту операцию к O(1).

```
addNewHead(n) {
        node = Node{}
        node.data = n
        //если список был пустой
        if (head == null) {
                tail = node
        } else {
                //прежний head сдвигаем на один узел вперед
                node.next = head
        }
        //записываем новый узел в качестве head
        head = node
}
```

```
class Node(object):
  def __init__(self, data):
    self.data = data
    self.next = None


class LinkedList(object):
  def __init__(self):
    self.head = None

  def append_front(self, data):
    # создаем новый узел и добавляем в него новое значение data
    new_node = Node(data)
    # если ранее список был пуст, значит первый элемент и будет являться
головой (head)
    if not self.head:
      self.head = new_node
      return

    # если список не пуст, то устанавливаем head
     # в качестве параметра next для нового узла
    new_node.next = self.head
    # записываем в head новый узел
    self.head = new_node
    # new_node.next, self.head = self.head, new_node
```

Вставка в конец списка

- Первая половина метода идентична вставке в начало.
- В отличие от вставки в начало, нам необходимо пройти по всем элементам, что приводит нас к сложности O(n).

```
addNewTail(n) {
        node = Node{}
        node.data = n
        //если список был пустой
        if (tail == null) {
                head = node
        } else {
                tail.next = node
        }
        //записываем новый узел в качестве tail
        tail = node
}
```

```
def append_back(self, data):
  # создаем новый узел и добавляем в него новое значение data
  new_node = Node(data)
  # если ранее список был пуст, значит первый элемент и будет являться головой (head)
  if not self.head:
    self.head = new_node
    return

  # если список не был пустым - начинаем перебирать все элементы
  # до тех пор, пока не дойдем до узла у которого next пустой
  cur_node = self.head
  while cur_node.next:
    cur_node = cur_node.next
  # в элемент, который до вставки был последним, в поле next указываем новый узел
  cur_node.next = new_node
```

Перебор всего списка в цикле

- Начнём обход списка с головы, сохраняя значение head в промежуточную переменную.
- Сохраним весь наш список в переменной.
- До тех пор, пока мы не упёрлись в конец списка, то есть пока у элемента есть указатель на следующий узел.
- Как только мы дошли до узла, у которого поле next равно None, выводим наш список.

```
cur = linkedList.head
while cur != null {
        print(cur.data)
        cur = cur.next
}
```

```
def print_list(self):
  # начнем обход списка с головы, сохраняя значение head в промежуточную переменную
  cur_node = self.head
  # сохраним весь наш список в переменной
  output = ""
  # до тех пор, пока мы не уперлись в конец списка
  # пока у узла есть указатель на следующий узел
  while cur_node is not None:
    output += str(cur_node.data)
    # добавим проверку next, чтобы избежать в конце стрелки ведущей в никуда
    if cur_node.next:
      output += " -> "
    cur_node = cur_node.next
  # как только мы дошли до узла у которого поле next равно None выводим наш список
  print(output)
```

Вставка в середину

Попробуйте решить самостоятельно.
Ответ через 1, 2, 3...

```
insert(after, n) {
        //находим after
        search = linkedList.head
        while search != null {
                if search.data = after {
                        break
                }
                search = search.next
        }
        //если мы нашли элемент after
        if search != null {
                node = Node{}
                node.data = n
                if search == tail {
                        tail = node
                }
                node.next = search.next
                search.next = node
        }
}
```

Сложность

1

Вставка в начало
и конец списка O(1)

2

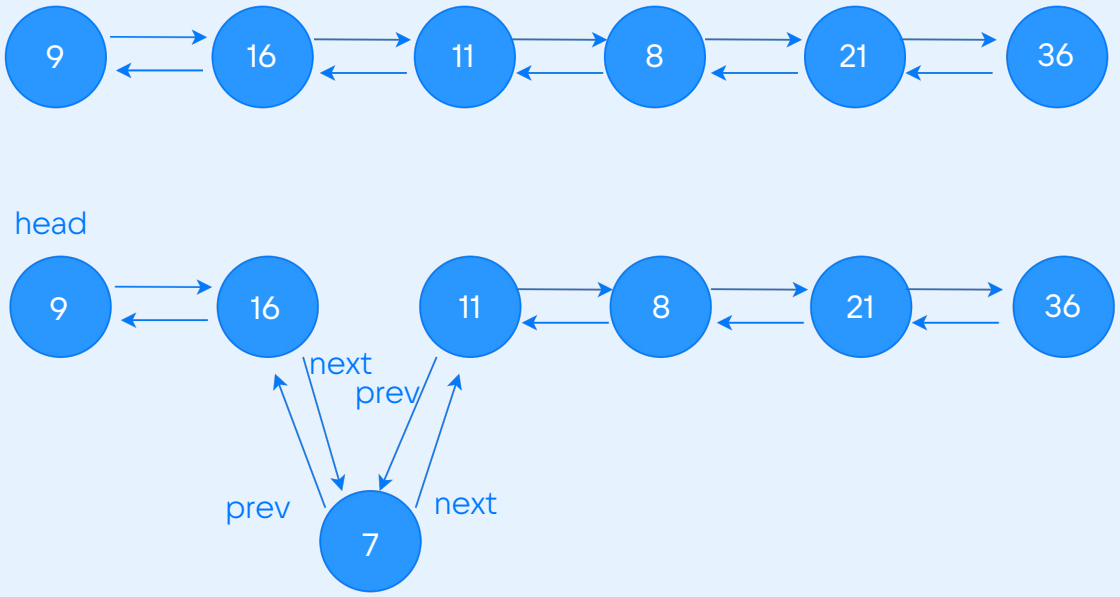
Вставка
в середину O(n)

3

Удаление
из середины O(n)

Двусвязный список

- Каждый узел, кроме первого и последнего, хранит указатели на следующий и на предыдущий узлы.
- Занимает больше памяти в сравнении с односвязным.
- Мы можем производить вставку не только после, но и перед элементом.
- При вставке или выборке необходимо обновлять два указателя: на следующий и на предыдущий узлы.



Вставка

- При вставке нам теперь надо следить за указателем на предыдущий элемент.
- append_front — создаём новый узел и добавляем в него новое значение data.
- Если ранее список был пуст, значит, первый элемент и будет являться головой (head).
- Если список не пуст, то устанавливаем head.
- В качестве параметра next для нового узла записываем в head новый узел.

- append_back — повторяем первые два пункта из append_front.
- Идём по списку до конца, начиная с головы.
- Элементу, который был последним, в поле next записываем новый созданный узел.
- В новый элемент, в поле prev, записываем узел, который до вставки был последнии.

```
def append_front(self, data):
  # создаем новый узел и добавляем в него новое значение data
  new_node = Node(data)
  if self.head is None:
    # если ранее список был пуст, значит первый элемент и будет являться головой (head)
    self.head = new_node
    return
  # если список не пуст, то устанавливаем head
  # в качестве параметра next для нового узла
  new_node.next = self.head
  self.head.prev = new_node
  # записываем в head новый узел
  self.head = new_node
```

```
def append_back(self, data):
  # создаем новый узел и добавляем в него новое значение data
  new_node = Node(data)
  if self.head is None:
    self.head = new_node
    return
  # пройдемся по списку до конца, начиная с головы
  cur_node = self.head
  while cur_node.next is not None:
    cur_node = cur_node.next
  # элементу, который был последним, в поле next записываем новый
  cur_node.next = new_node
  # в новый элемент, в поле prev записываем узел, который до вставки был последним
  new_node.prev = cur_node
```

Метод двух указателей

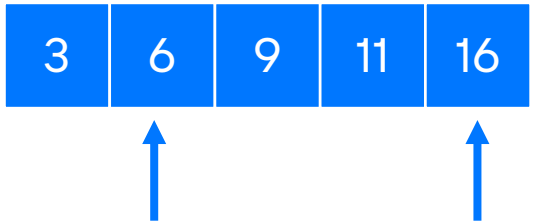
- Распространённый шаблон для решения задач, связанных с массивами и списками.
- Он позволяет свести сложность O(n^2) к O(n) за счёт исключения необходимости в дополнительном цикле. То есть мы проходимся по нашей последовательности лишь один раз, в то время как наивный метод решения подразумевал бы цикл в цикле.
- Применим, когда речь идёт об отсортированном массиве или о списке.
- Если речь идёт о поиске группы элементов, удовлетворяющих определённому свойству. Например, поиск двух чисел, сумма которых выше или эквивалентна заданной, или удаление повторяющихся значений.

Задача на примере массива

- Задан отсортированный массив целых чисел, необходимо вернуть индексы двух элементов, сумма которых равна заданному числу. Набор чисел задан таким образом, что результат может быть только один. Нельзя использовать одно и то же число дважды.
- Зададим значение двух указателей: первый — на нулевой позиции, второй — на последнем элементе.
- Складываем, если значение меньше заданного числа, то двигаем левый указатель, так как нам надо двигаться в сторону увеличения.
- Снова складываем два значения — перебор, значит, двигаем правый указатель влево, так как массив отсортирован и нам надо двигаться в сторону уменьшения суммы.
- Повторяем эти действия, пока не найдём заданную сумму или не убедимся, что данный массив не имеет решения.

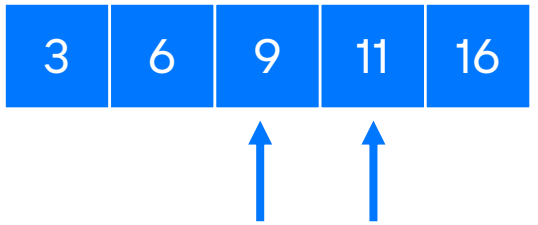
Задача на примере массива

3 + 16 < 20



6 + 16 > 20

3 + 11 < 20



9 + 11 = 20

Будем ВКонтакте!