Алгоритмы и структуры данных

Почуев Илья
Павлов Михаил

Немного организационных вопросов

- Обучение будет в формате лекция + семинар
- Лекция по вторникам, семинар по четвергам и пятницам.
- Во время лекции микрофон должен быть выключен
- Вопросы задаем в чат
- Актуальное расписание, ссылки на дз, описание занятий можно найти на VK Education
- Также на платформе нужно отмечать свое присутствие на занятии
- Памятка по работе с платформой https://elfin-fortnight-60b.notion.site/0880660474c04237ba7e3c2e19b42111

О чем курс?

- Структуры данных
- Алгоритмы
- Подходы к решению задач
- Поговорим о прикладной пользе

Темы лекций

- Массивы и списки
- Очереди и стек
- Хеш-таблица
- Алгоритмы поиск
- Сортировки
- Динамическое программирование
- Деревья и древовидные структуры
- Графы

Чего не будет в курсе

- Языки программирования
- Математическое обоснование

Зачем нам это?

- Хранение данных
- Обработка данных
- Понимание сложности и ресурсоемкости
- Часть профессиональной гигиены

Базовые структуры данных и оценка сложности алгоритма

- Как оценивать алгоритм, нотация О большое или Big O
- Разберем линейные структуры данных
- Поговорим о сложности основных операций в каждой из структур
- Поймем отличие и особенности каждой из них

Сложность алгоритма или Big O

- Как оценивать сложность?
- O(N) дает связь между входными значениями и итоговой оценкой сложности
- Виды сложности от O(1) до O(N!)
- Константы не влияют на сложность: O(2*N)~O(20*N)
- Учитываем все входные параметры
- Всегда “округляем” до наихудшей сложности
- Оцениваем сложность по потребляемой памяти

Еще один аспект

- алгоритмическая часть собеседований


Спустя 15 лет
я нашел его...
Robotatertotcomics
Вот он!!! Идеальный
алгоритм сортировки
const arr = [36, 18, 9, 11, 8]
for (let i = 0;i < arr.length - 1;i++) {
setTimeout(
() => {console.log(arr[i])},
arr[i]
)
}
Да чтоб
тебя

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

Структура массива

- size количество элементов
- capacity объем массива
- type тип данных

```
Array:
    size int
    capacity int
    type int
```

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 1000
- Тогда байты с 1000 по 1007 будут принадлежать первому элементу массива. Байты с 1008 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 1000 + 4*8 = 1032

Действия над массивом

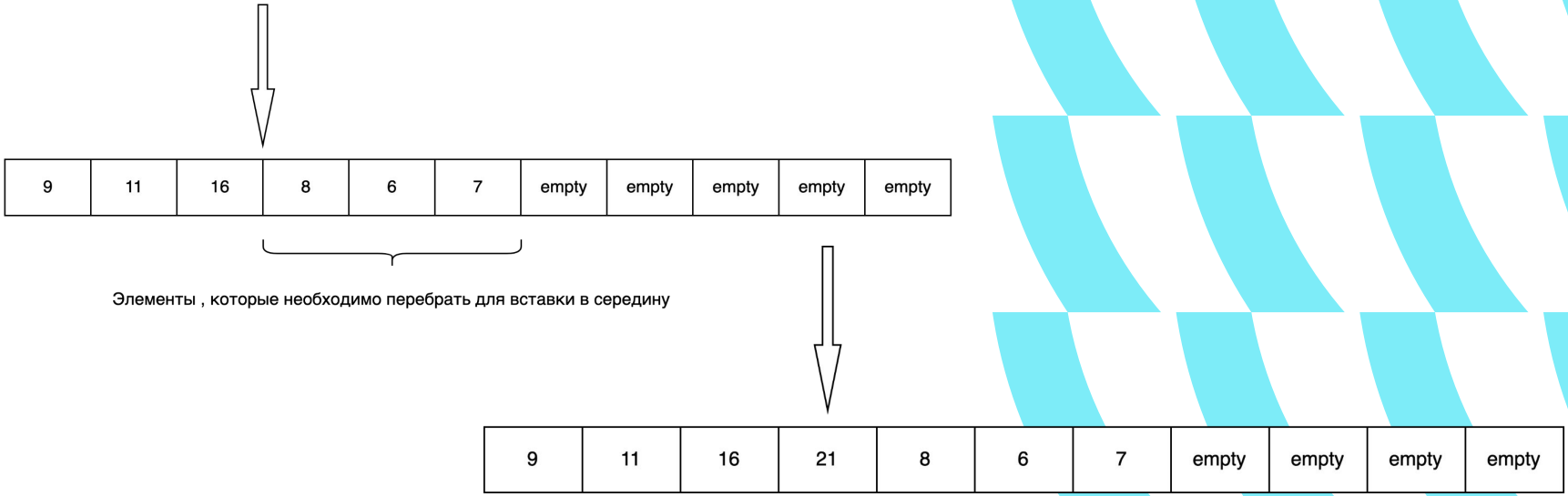
- append() - вставка
- get(index) - получение элемента по индексу
- capacity - ёмкость массива, количество элементов которое он может в себя вместить
- size (size<=capacity) - количество заполненных ячеек массива

Сложность основных операций

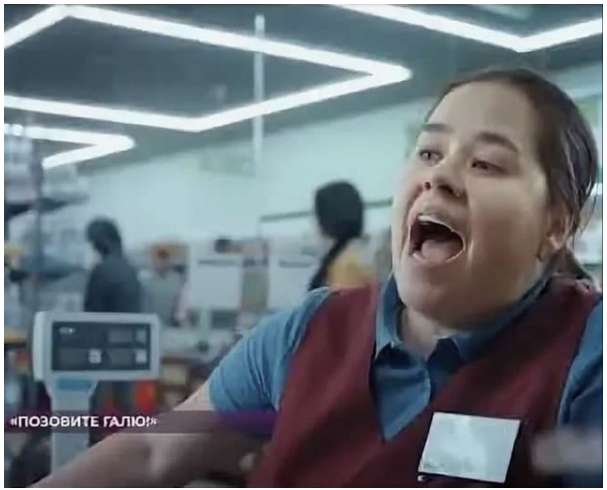
- Получение данных O(1)
- Вставка в середину O(n)
- Вставка в конец O(1)
- Удаление O(n)

Вставка/удаление из середины массива

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

- Как быть если мы не всегда можем заранее знать размер массива?
- При этом есть необходимость получения элементов по индексу
- По-прежнему хотим O(1) при выборке

Как должна работать такая структура данных?

- Как я уже говорил 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 плюс чтобы там было место под новый элемент.
- Скопировать все значения из старого массива в новый
- Вставить новый элемент
- Удалить старый массив, чтобы избежать утечки по памяти
- Попробуем реализовать похожий на описанную выше процедуру в коде.

Попробуем подобрать алгоритм увеличения capacity

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

```python
import ctypes


class DynamicArray(object):
    def __init__(self):
        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()
            print(f"slow: {self.cap()}")
        else:
            print('fast')

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

    def resize(self):
        new_cap = self.capacity + 5  # как правильно увеличивать capacity?
        new_array = self.make_array(new_cap)

        for i in range(self.size):
            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
```

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?
- Проведем множество операций вставки, подсчитаем общее количество элементарных действий и время на их выполнение и разделим общую сложность на количество операций.
- Такая усредненная сложность называется амортизированной сложностью, а анализ называется амортизационным.

- При заполнении массива 10000-ми элементов метод resize будет вызываться 2000 раз. Это не совсем то, что нам нужно
- При увеличении емкости на 10 элементов - 1000 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 стремится к O(n)
- Самым оптимальным будет увеличивать сложность в константное значение, например в 2.
- Количество вызовов функции resize будет всего 15
- Количество простых операций при увеличении capacity на 5 = 9997001
- Количество простых операций при увеличении capacity в 2 раза = 16384

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

    for i in range(self.size):
        self.count += 1     # 9997001
        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
```

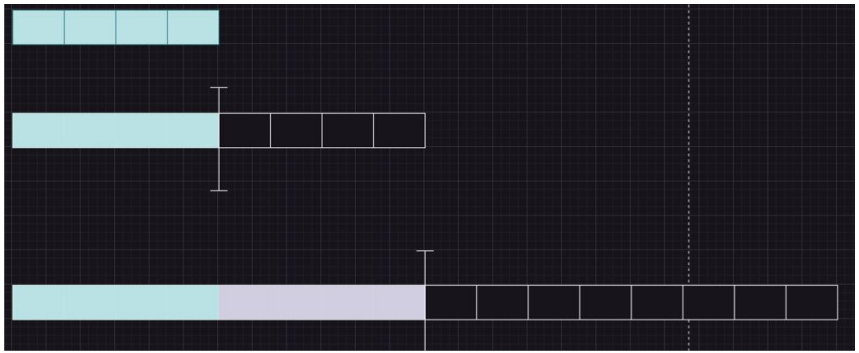
```python
def resize(self):
    new_cap = self.capacity * 2
    new_array = self.make_array(new_cap)

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

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

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 которой 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 о большое эквивалентно O(1)

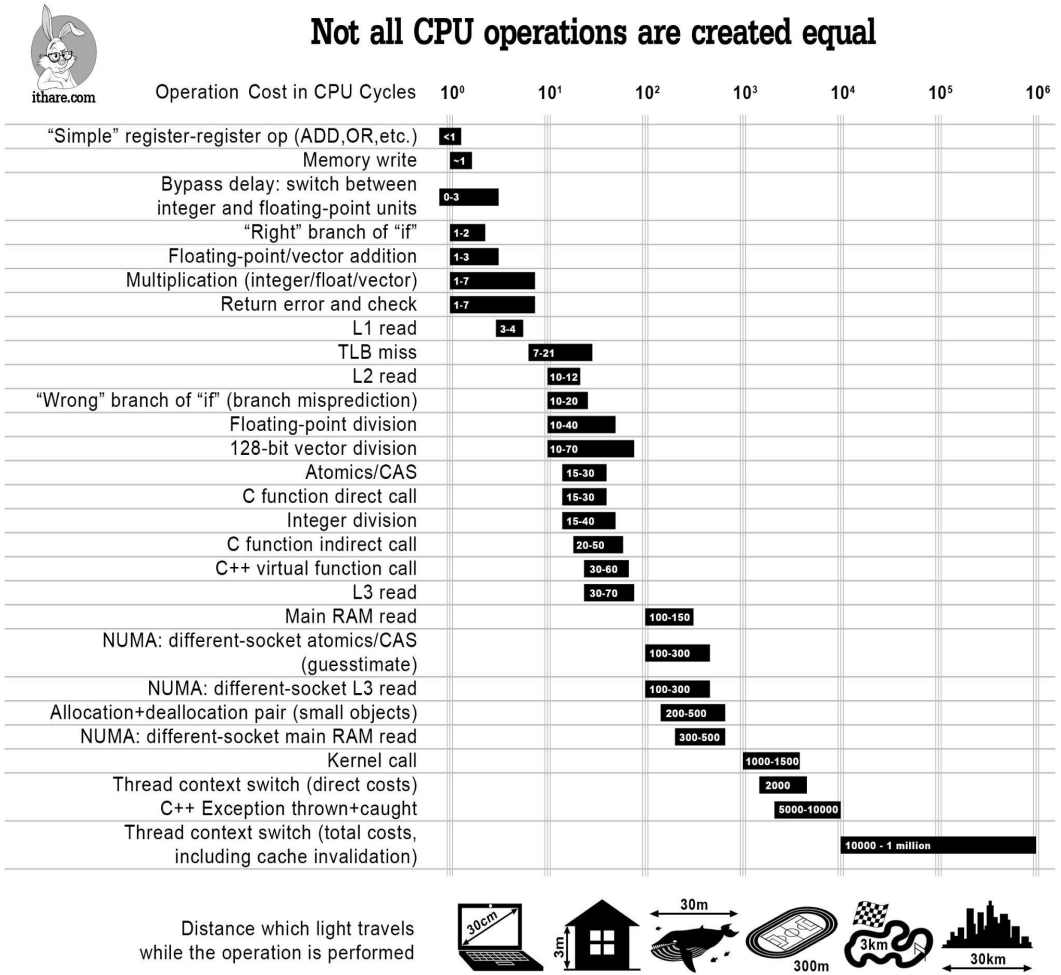
Резюме

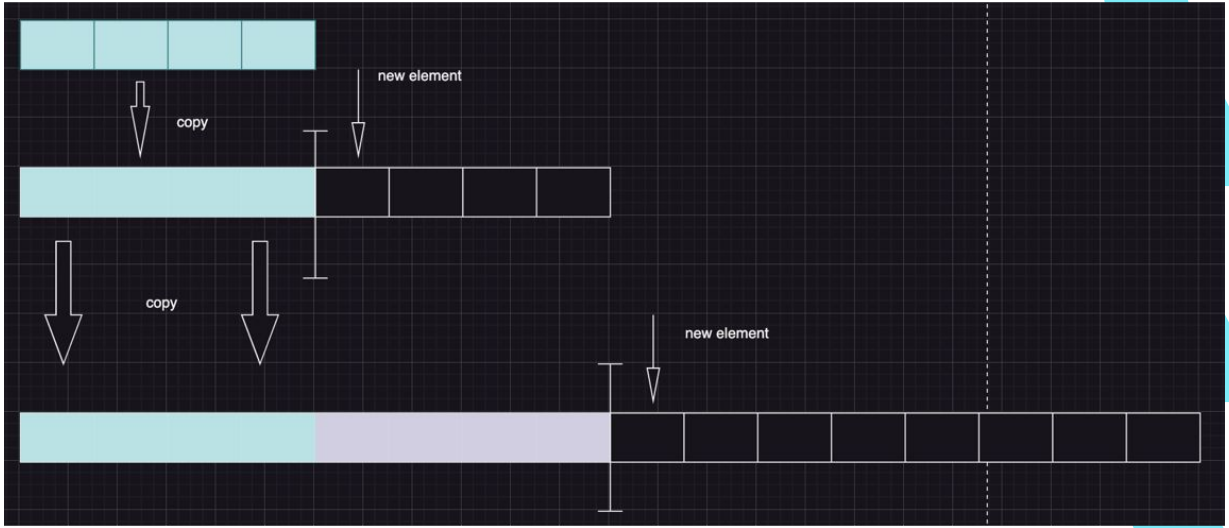
- не обязательно знать конечный размер массива
- при превышении capacity создается новый массив с capacity*2 (в какое-то константное значение)



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

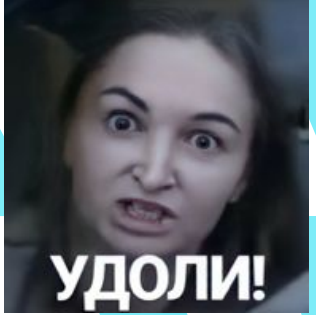
Почему надо избегать аллокаций





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

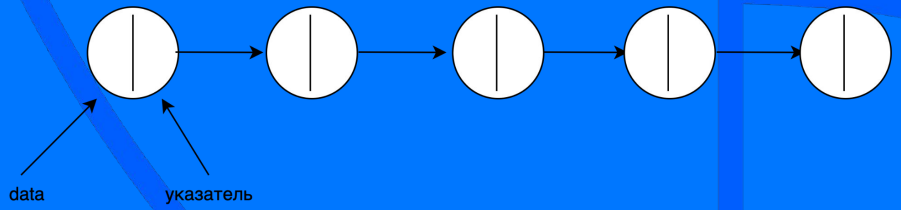
А как удалять



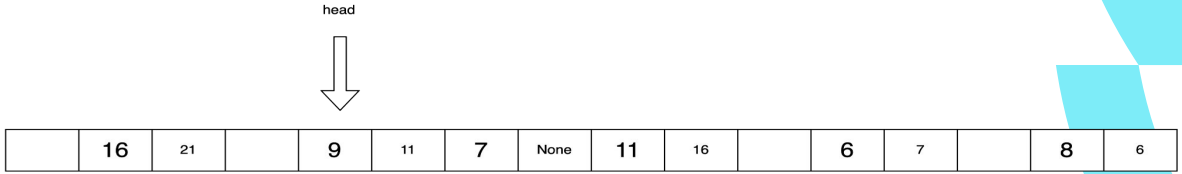
- Первое что приходит на ум - раз мы при добавлении увеличиваем емкость в два раза, то возможно при удалении надо следовать той же стратегии и уменьшать capacity в два раза когда массив освободился наполовину?
- Тут надо быть аккуратным, сейчас мы почти попали в ловушку. Дело в том, что если после уменьшения capacity в два раза, сразу последует вставка, то нам придется вновь аллоцировать в два раз больше памяти. А если эти операции будут повторяться? То мы рискуем получить в своем алгоритме сложность O(n)
- Уменьшают в два раз 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

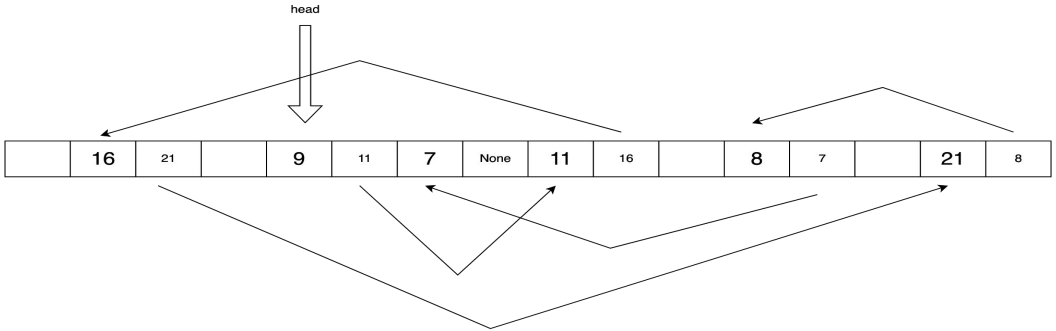
- Однонаправленный (односвязный)
- Двунаправленный (двусвязный)



Однонаправленный список



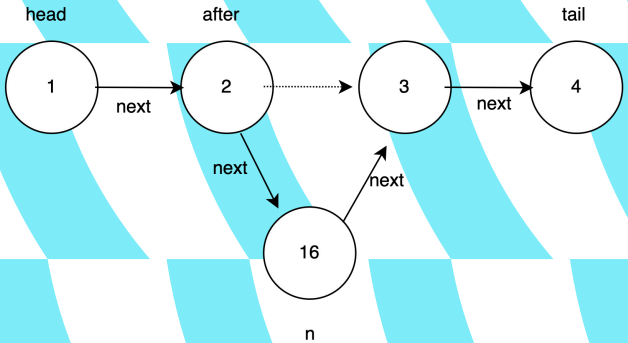
- Структура отличается от изученного нами массива
- У нас есть какой-то участок памяти. Где-то в нем хранится наш первый элемент списка, его принято называть head - голова списка
- Список устроен таким образом, что head знает где хранится второй элемент списка.
- Второй элемент знает где хранится третий 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.
- Получившееся структура, в которой каждый элемент знает, где хранится следующий называется односвязным списком.
- Важно понимать - 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. Правда и накладывает 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 отличии от массива нет необходимости хранить данные последовательно



Абстрактное представление в коде

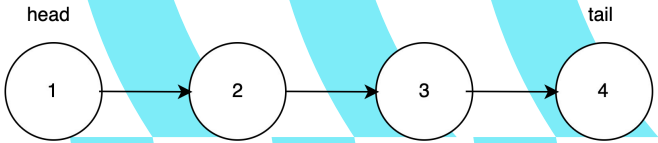
- Каждый элемент списка мы будем называть узлом или нодой от Node
- Узел - основная часть списка, обычно определяющаяся классом или структурой.
- Структура каждого элемента представляет из себя какую-то полезную информацию data и указатель на следующий элемент.
- Голова списка (Head): Указатель на первый 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
}
```

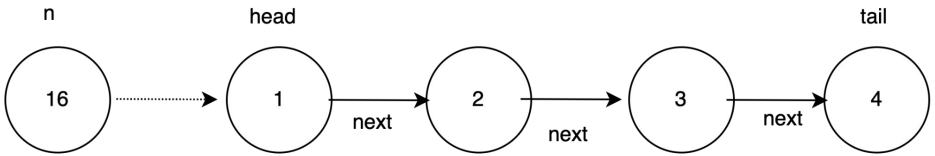
В общем виде список выглядит так

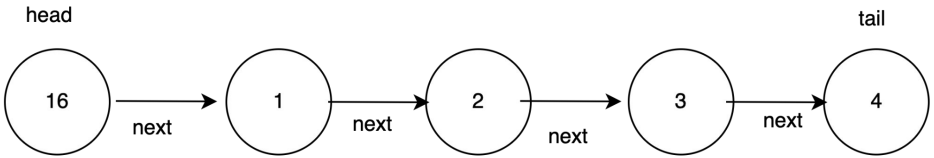


Вставка в начало списка

- 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)

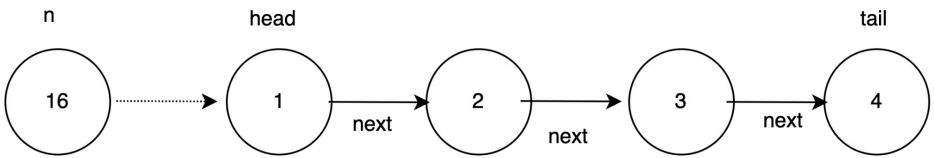
```
addNewHead(n) {
    node = Node{}
    node.data = n
    //если список был пустой
    if (head == null) {
        tail = node
    } else {
        //прежний head сдвигаем на один узел вперед
        node.next = head
    }
    //записываем новый узел в качестве head
    head = node
}
```

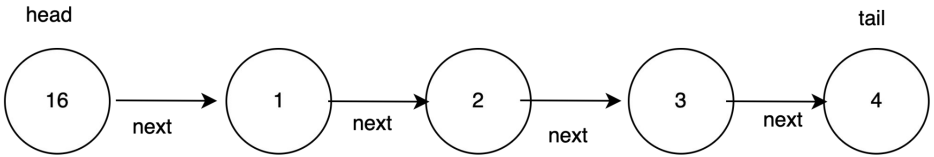




Вставка в начало списка

```
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 вставки в начало
- В отличии 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

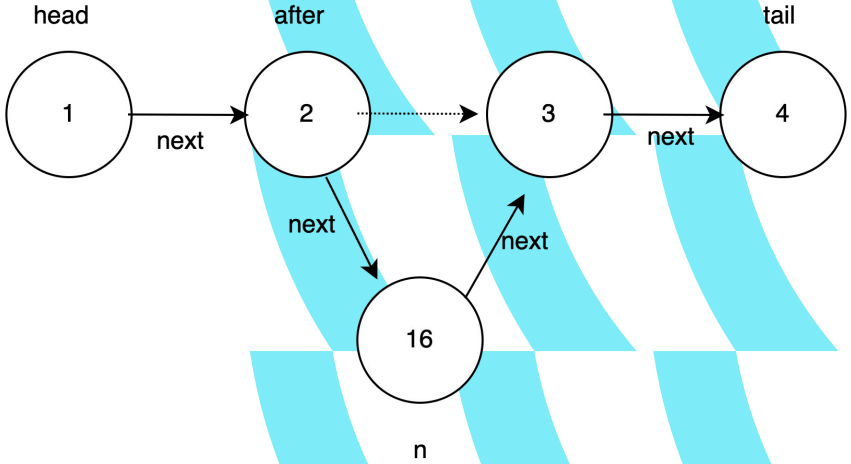
- начнем обход списка с головы, сохраняя значение head в промежуточную переменную
- сохраним весь наш список в переменной
- до тех пор, пока мы не уперлись в конец списка то есть пока у элемента есть указатель на следующий узел
- как только мы дошли до узла у которого поле next равно None выводим наш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

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

```python
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

```
insert(linkedList, 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
      }
}
```

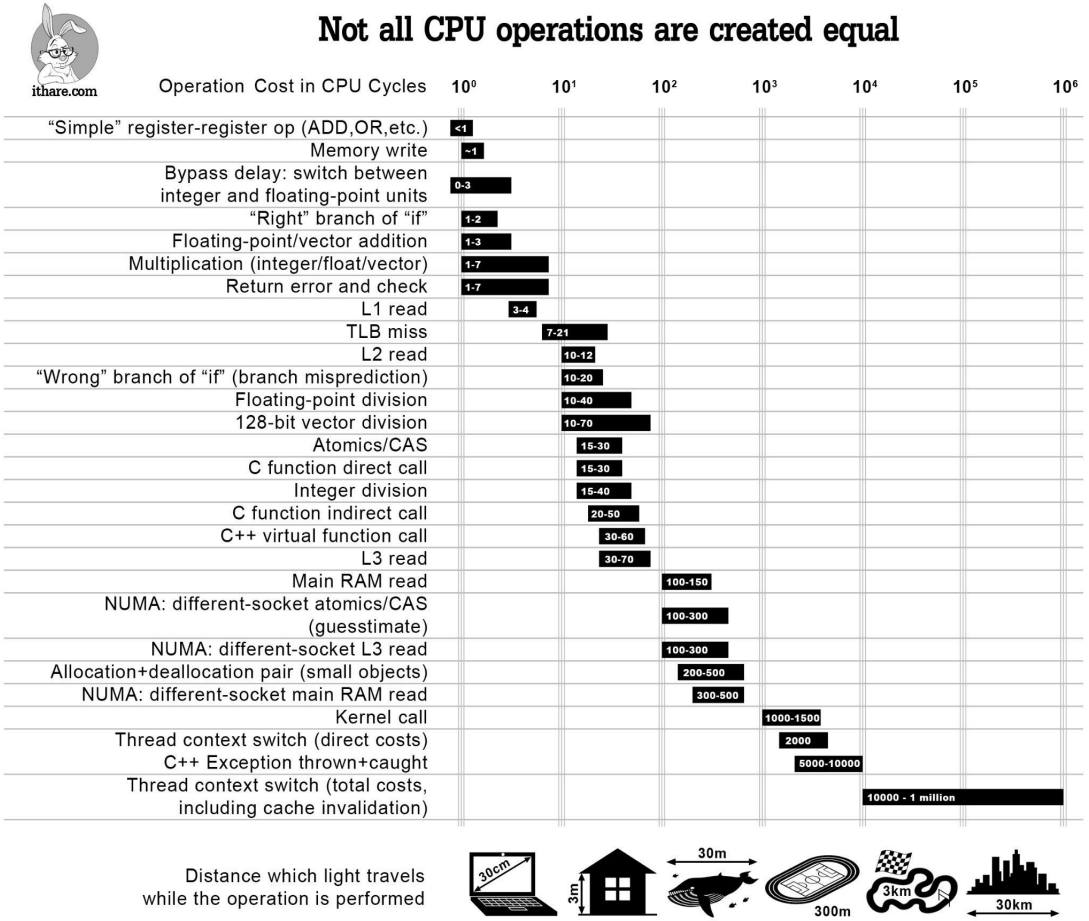


Сложность

- Вставка в начало O(1)
- Вставка в конец списка O(1) при наличии tail
- Вставка в середину O(n)
- Удаление из середины O(n)

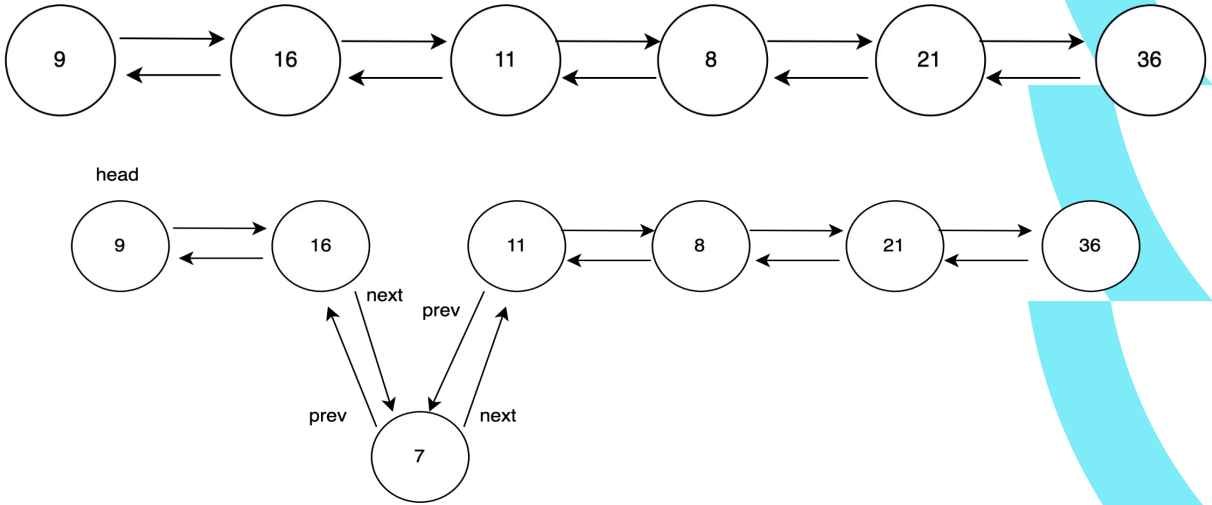
Массив или список?

Массив, если он может поместиться в кеше, за счет расположения последовательно в памяти, будет читаться из кеша.



Двусвязный список

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



Вставка

- При вставке нам надо теперь следить за указателем на предыдущий элемент
- **append_front** создаем 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
- идем 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 последним

```
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
```