Немного орг. вопросов

- Семинары: форкаемся, кто еще этого не сделал
- Дедлайн до следующего понедельника
- Дз - если не все тест кейсы проходят, пишем мне.
- Дедлайн - следующий понедельник.

Немного орг. вопросов

- сегодняшняя группа, у которой был семинар переходят на четверг в 18.40
- все вопросы по проблемам с all cups - помогут в чате
- во время лекции микрофон должен быть выключен
- вопросы отправляем в чат, либо обсуждаем между темами

Стек, Очередь, Двойная очередь

Stack, Queue, Deque

О чем поговорим сегодня?

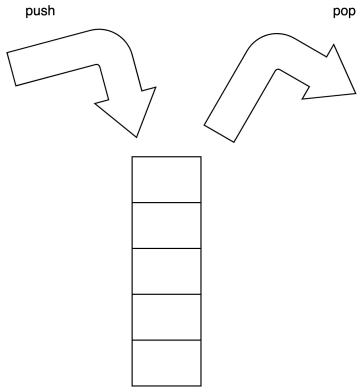
- Стек - что из себя представляет, какой основной принцип работы, основные операции и на чем его можно реализовать
- Очередь - что из себя представляет, какой основной принцип работы, основные операции и на чем её можно реализовать
- Двойная очередь по тому же сценарию

Стек, очередь.
Двойная очередь

- В первой лекции мы изучили линейные структуры данных, обсудили преимущества и недостатки того или иного представления в памяти.
- Поговорили о том, в каких ситуациях что лучше выбрать. Массив - read only хранилище, список для вставок.
- Сейчас мы изучим абстрактные типы данных , которые могут быть реализованы при помощи уже известных нам массивам и спискам
- Мы поговорим с вами о стеке и очередях. Узнаем какие принципы лежат в их основе, а так же решим хрестоматийную задачу, которая даст понимание для чего же все типы нужны.
- Начнем нашу лекцию с абстрактного типа данных, который называется стек.

Стек

- LIFO (last in first out) - первый вошел, последний вышел
- Добавление и удаление в этой структуре возможно только с одного конца
- Типичный пример для этой структуры данных - стопка бумаг

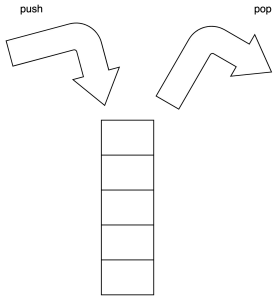


Области применения

- **История браузера**: каждый раз, когда вы посещаете новую страницу, URL-адрес помещается в стек, а когда вы нажимаете кнопку "Назад", предыдущий URL-адрес извлекается из стека
- **ctrl+Z** в текстовом редакторе
- **Вызовы функций и рекурсия**: при вызове функции текущее состояние программы помещается в стек. Когда функция заканчивает выполнение, состояние извлекается из стека, чтобы возобновить выполнение предыдущей функции.
- **Оператор defer в golang:** defer добавляет вызов функции, которая указана после него в стеке приложения

Основные операции

- Какие операции должен поддерживать стек исходя из его принципа работы?
- **Push** - добавление элемента в вершину стека.
- **Pop** - извлечение элемента. Всегда возвращается вершина стека. Сколько бы элементов мы не добавили, всегда будет возвращаться последний.
- Несмотря на то, что это абстрактный тип данных и у него может быть множество реализаций, ключевое в каждой из них это то, что они эти две операции должны выполняться за O(1)



Реализация на списке

- Одна из возможных реализаций
- В нашем случае достаточно односвязного списка
- **Push** будет писать данные, как это делал append_front в уже написанной нами функции
- **Pop** просто будет возвращать head. Главное не забывать переписывать указатели

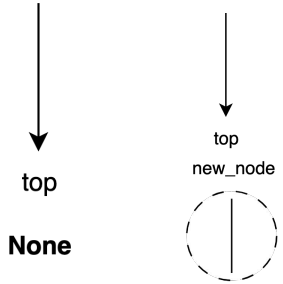
Вставка.

Реализация на списке

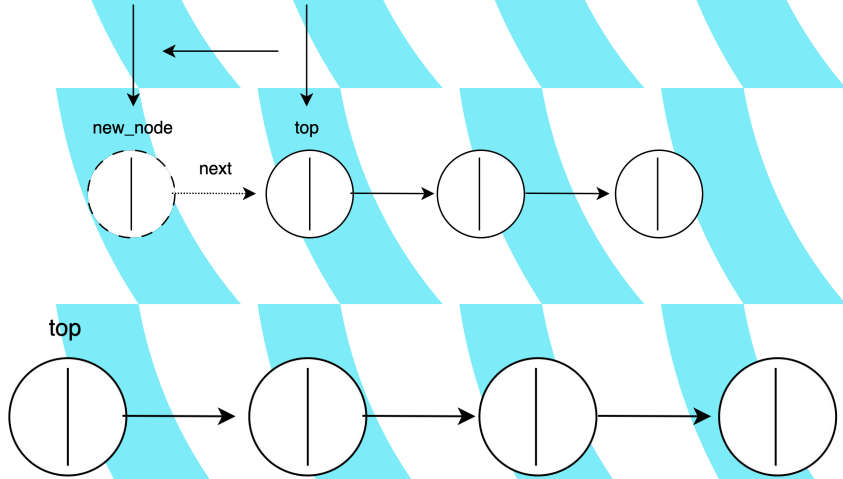
```
class Stack(object):
    def __init__(self):
        self.top = None

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

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

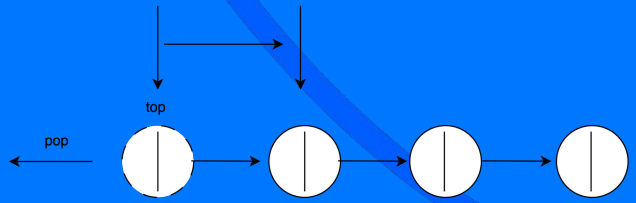


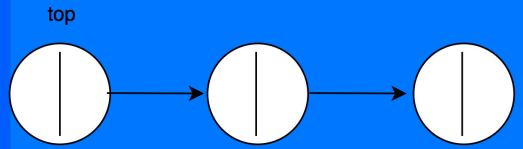

```
class Node(object):
    def __init__(self, data):
        self.data = data
        self.next = None
```



Выборка

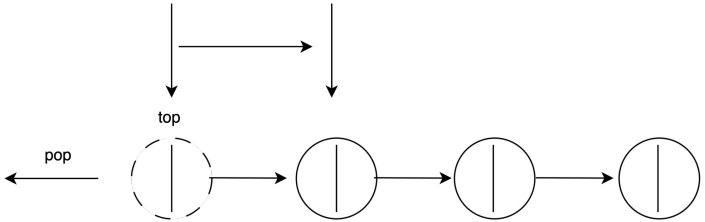
- **Pop:** со вставкой немного сложнее
- проверяем, что наш стек содержит хотя бы вершину
- Если стек пуст - возвращаем любое значение, по которому мы будем определять это состояние.
- Если стек не пуст и в нем хранится больше одного значения - переписываем значение вершины на следующий элемент
- если стек пуст, то устанавливаем вершину в значение None.

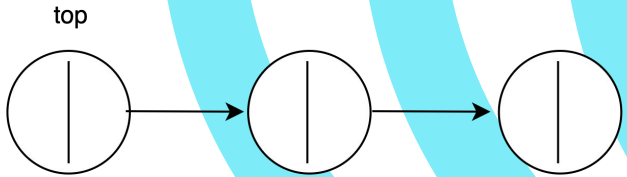




Выборка

```
def pop(self):
    # проверяем, что наш стек содержит хотя бы вершину
    if not self.top:
        # можно возвращать -1
        return None
    top = self.top
    if self.top.next is not None:
        # переписываем значение вершины на следующий элемент
        self.top = self.top.next
    else:
        # если стек пуст, то устанавливаем вершину в значение None
        # можно указать любое другое значение, которое было бы удобно нашей реализации
        self.top = None
    # возвращаем значение, хранящиеся в вершине стека
    return top.data
```





Реализация на массиве

- В нашем случае, так как мы не знаем, сколько данных будет приходить в наш стек, то лучше всего использовать саморасширяющийся массив
- Принцип тут по-сути идентичен реализации на списке, разве что вставлять мы будем не в начало, а в конец, чтобы обеспечить амортизационную сложность O(1)
- Разберем пример на простом массиве

```
class Stack:
    // Инициализация стека
    function initialize(size):
        this.stack = new Array[size]
        this.top = null
    // Проверка, пуст ли стек
    function isEmpty():
        return this.top == null
    // Проверка, полон ли стек
    function isFull(size):
        return this.top == size - 1
    // Добавление элемента в стек
    function push(element, size):
        if this.isFull(size):
            print "Стек полон"
        else:
            this.top = this.top + 1
            this.stack[this.top] = element
    // Удаление элемента из стека
    function pop():
        if this.isEmpty():
            print "Стек пуст"
        else:
            element = this.stack[this.top]
            this.top = this.top - 1
            return element
    // Получение верхнего элемента стека без его удаления
    function peek():
        if this.isEmpty():
            print "Стек пуст"
        else:
            return this.stack[this.top]
```

Пример для понимания стека

- Хрестоматийная задача, для понимания где можно использовать стек это ПСП - правильная скобочная последовательность.
- Необходимо по переданной строке, состоящей из открывающих и закрывающих скобок понять, является ли последовательность скобок - правильной.
- [], {()}[], ([{}]), (({})), ([(({}))]) - валидные последовательности. У каждой открывающей скобки есть в нужном месте закрывающая
- [(]), ((()), ({[})], ((()), [[]]] - невалидные последовательности

Решение

- используем список как стек
- идем в цикле по нашей последовательности
- если скобка открывающая, то пишем ее в стек
- если на итерации мы встретили закрывающую скобку, но стек уже пустой, значит последовательность не валидна.

```
for each bracket in expression {
    if bracket is opening {
        stack.push(bracket)
    } else if bracket is closing {
        if stack.isEmpty() or stack.top() is not corresponding opening bracket {
            return false
```

Решение

- при этом, во время проверки, если скобка закрывающая, то мы удаляем с вершины, соответствующую открывающую скобку, то есть освобождаем стек на один элемент
- если скобка, которая в данный момент находится на вершине стека, не является открывающей для текущей скобки - последовательность также не валидна
- когда мы прошлись по всей последовательности наш стек должен быть пустым, в противном случае открывающих скобок больше, а значит последовательность не валидна.
- сам по себе алгоритм подойдет не только для скобочной последовательности, но и например для валидации html или xml документов, так как там есть открывающие и закрывающие теги.

```
        if stack.isEmpty() or stack.top() is not corresponding opening bracket {
            return false
        }else{
            //выбираем вершину у стека
            stack.pop()
        }
    }
}
//если после цикла стек пуст, то последовательность правильная
if stack.isEmpty(){
    return true
} else{
    return false
}
```

Решение на псевдокоде

```
function isValid(expression) {
    stack = []
    //идем в цикле по скобочной последовательности
    for each bracket in expression {
        //если скобка открывающая - записываем ее в стек
        if bracket is opening {
            stack.push(bracket)
        //если скобка закрывающая
        } else if bracket is closing {
            //если мы попали на закрывающую скобку и стек при этом пуст
            //или закрывающая скобка не соответствует открывающей
            if stack.isEmpty() or stack.top() is not corresponding opening bracket {
                return false
            }else{
                //выбираем вершину у стека
                stack.pop()
            }
        }
    }
    //если после цикла стек пуст, то последовательность правильная
    if stack.isEmpty(){
        return true
    } else{
        return false
    }
}
```

Решение на python

```python
def isValid(bracket_sequence):
    stack = []  # используем список как стек, только методы append (push) и pop
    brackets_dict = {
        '[': ']',
        '{': '}',
        '(': ')'
    }
    for bracket in bracket_sequence:
        if bracket in brackets_dict:
            # если скобка открывающая, то пишем ее в стек
            stack.append(bracket)
        # если на итерации мы встретили закрывающую скобку, но стек уже пустой, значит последовательность не валидна
        # или, если скобка, которая в данный момент находится на вершине стека, не является открывающей для текущей
        # скобки - последовательность так же не валидна при этом, во время проверки, если скобка закрывающая,
        # то мы удаляем с вершины, соответствующую открывающую скобку
        elif len(stack) == 0 or bracket != brackets_dict[stack.pop()]:
            return False

    # когда мы прошлись по всей последовательности наш стек должен быть пустым, в противном случае
    # открывающих скобок больше, а значит последовательность не валидна
    return len(stack) == 0
```

Очередь

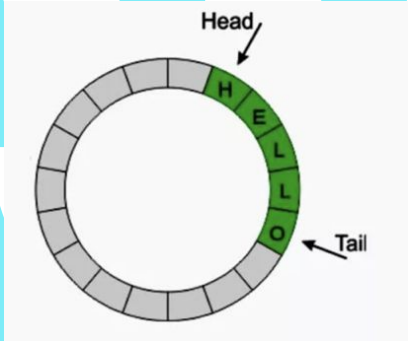
- Еще один абстрактный тип данных который мы с вами изучим - очередь
- FIFO - first in first out (первый пришел - первый вышел)
- Примером из жизни может являться обычная очередь на кассе
- Элемент, который мы положили первым, при запросе в очередь будет первым удален

Основные операции

- У очереди как и у стека есть две основных операции
- **Push** (enqueue) - кладет данные в конец очереди
- **Pop** (dequeue) - извлекает данные только из начала очереди, то есть первым вернется только тот элемент, который был добавлен первым.
- Так же как и у стека, сложность этих двух операций O(1)

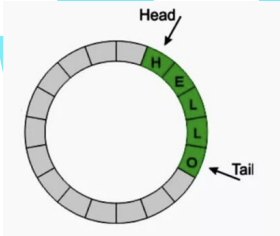
Кольцевой буфер (циклический массив)

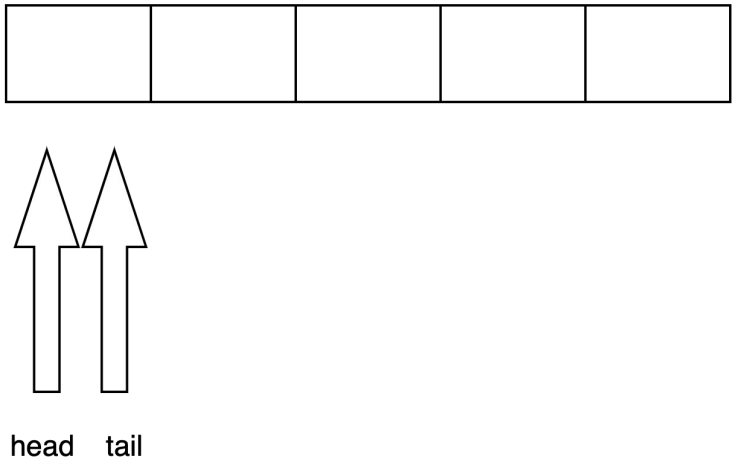
- Для начала давайте определим проблему, с чем мы столкнемся если будем использовать простой массив.
- Итак, попробуем реализовать на простом массиве
- Вводим две новые переменные head и tail - индексы начала и конца очереди.
- В пустом массиве они указывают на нулевой элемент
- При добавлении мы вставляем элемент в ячейку с индексом tail. После вставки инкрементируем tail. Выбираем из ячейки с индексом head.
- В какой-то момент при вставке tail может выйти за пределы массива, что недопустимо.
- Можно попробовать двигать все элементы в начало - тогда сложность начнет стремится к O(n)
- Выходом в такой ситуации может быть перемещение tail на нулевую ячейку.

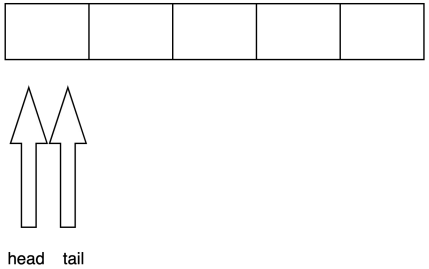


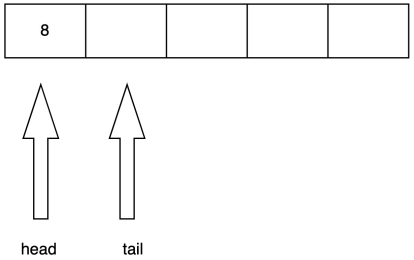
Кольцевой буфер (циклический массив)

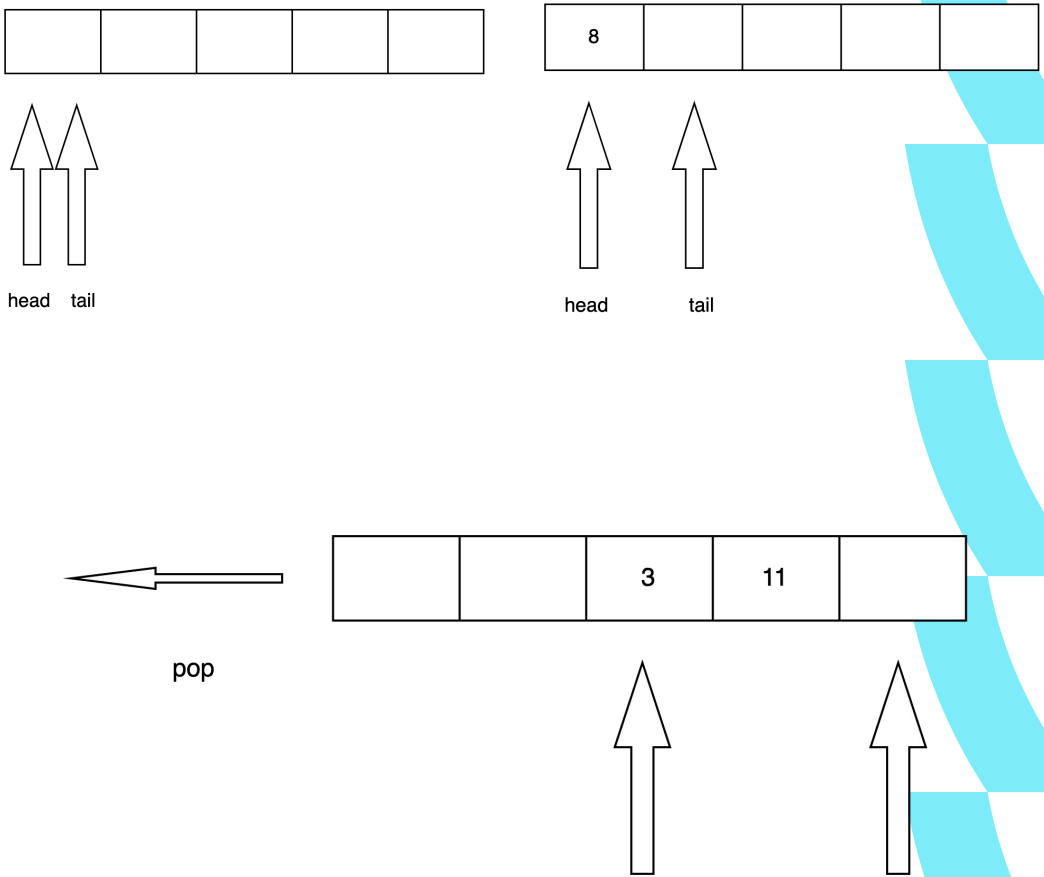
- создаем массив с фиксированным размером и инициализируем указатели начала и конца на первую позицию в буфере.
- вставка: если мы хотим записать данные в буфер, сначала проверяем, есть ли свободное место в буфере. Если указатель конца равен указателю начала минус один или указатель начала равен 0, это означает, что буфер полон и мы не можем записать новые данные. В противном случае, записывайте данные в текущую позицию tail и перемещайте его на следующую позицию в кольцевом порядке (например, если tail указывает на последнюю позицию в массиве, переместите его на первую позицию).
- Выборка: при чтении из буфера, сначала проверяем, есть ли доступные данные для чтения. Если head == tail, это означает, что буфер пуст и нет данных для чтения. В противном случае, читаем данные из текущей позиции head и перемещаем указатель начала на следующую позицию в кольцевом порядке.
- Повторение: повторяем шаги 2 и 3 для записи и чтения данных в кольцевом буфере по мере необходимости.

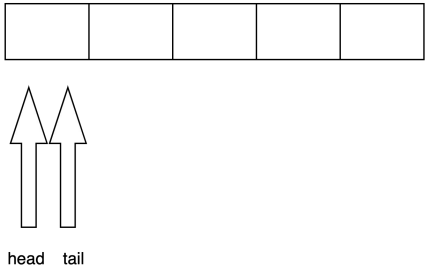


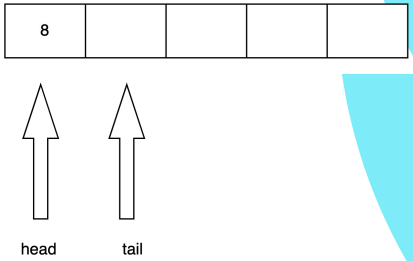

head
tail

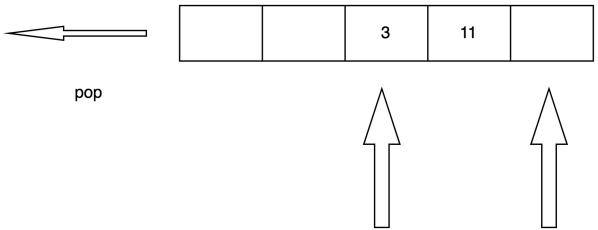

head
tail

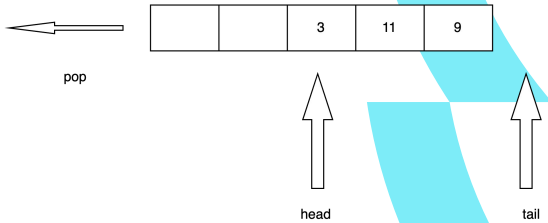

8
head
tail

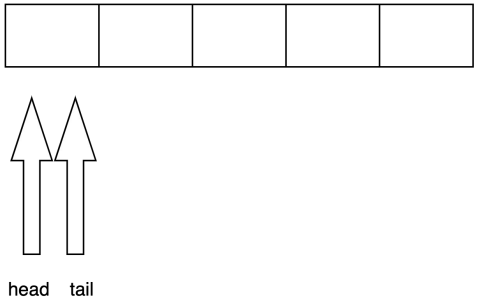

8
head
tail
head
tail
3
11
pop

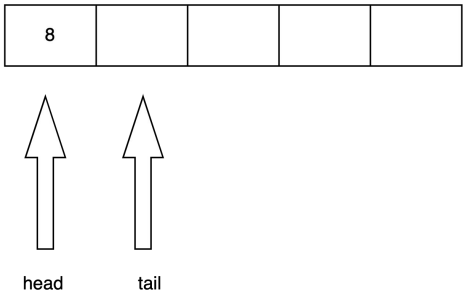

head
tail

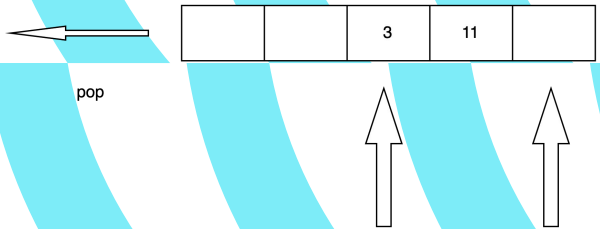

8
head
tail

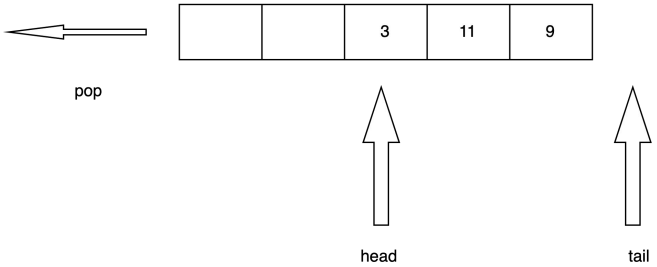

3
11
pop


3
11
9
pop
head
tail

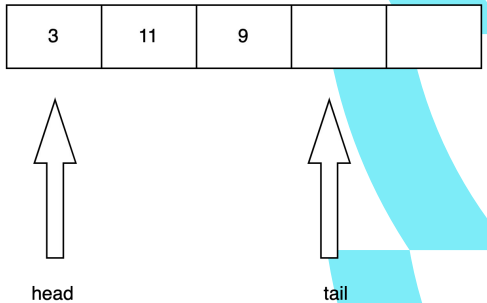


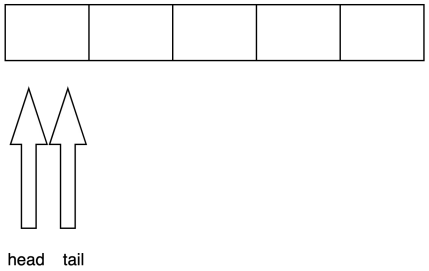


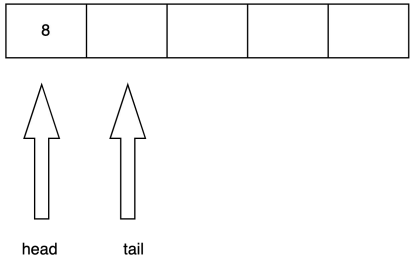


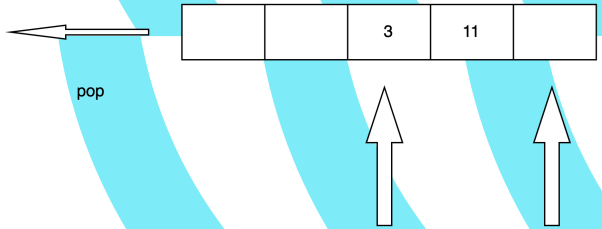


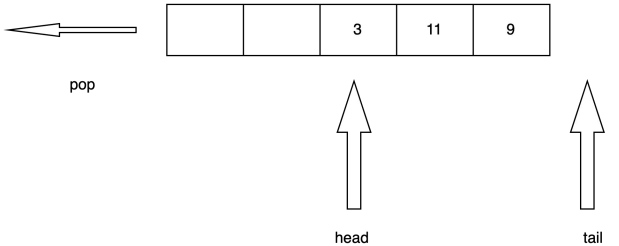
Сдвигаем все элементы к нулевой ячейке.
Тогда сложность стремится к O(n)

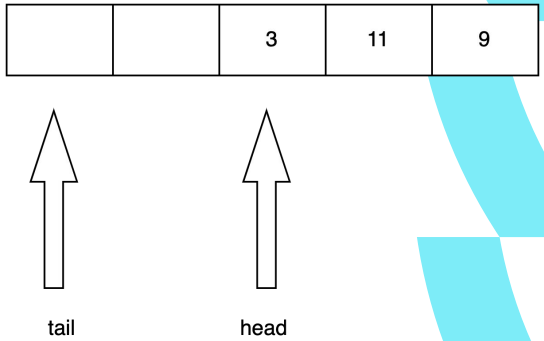


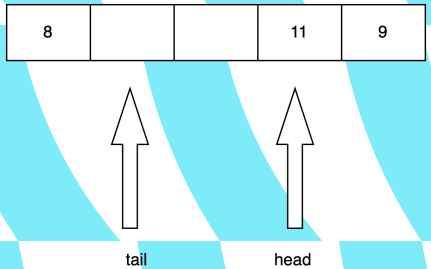

head
tail


8
head
tail


3
11
pop

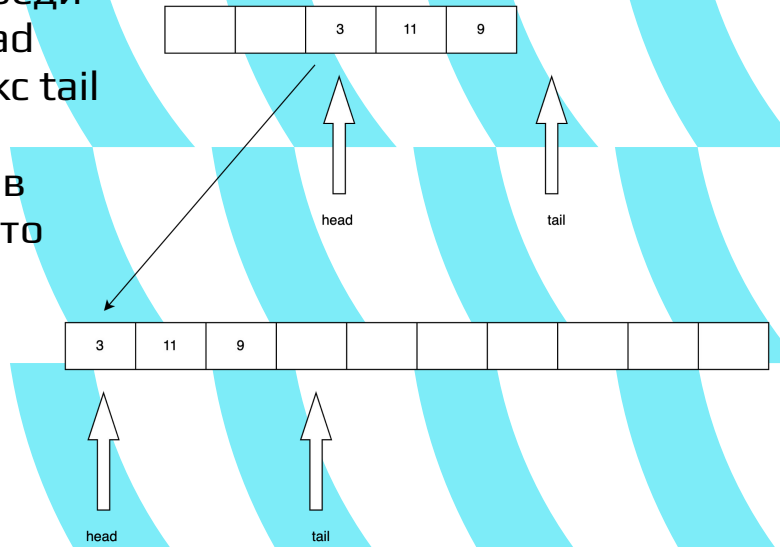

3
11
9
pop
head
tail


3
11
9
tail
head


8
11
9
tail
head

Реализация на саморасширяющемся массиве

- массив с фиксированным размером далеко не всегда подойдет для реализации очереди
- при чтении проверяем наличие элементов в очереди и если они есть, то возвращаем его и двигаем head
- при добавлении устанавливаем элемент на индекс tail и двигаем tail на +1 к концу массива
- при заполнении массива копируем все элементы в новый, но индексы переписываем с тем учетом, что head должен указывать на нулевую ячейку
- освобождаем память по тому же принципу, что

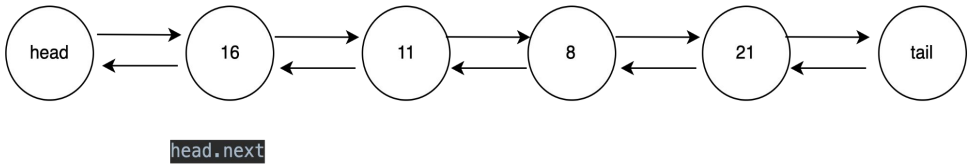


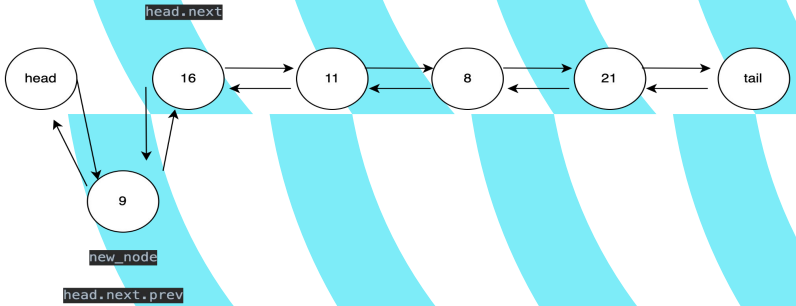
Сложность операций

- Вставка O(1). Вставляем только в конец очереди.
- Выборка O(1). Выбираем только из начала очереди.
- Учитываем необходимость увеличения массива
- Не забываем очищать неиспользуемую память

Очередь на основе двусвязного списка

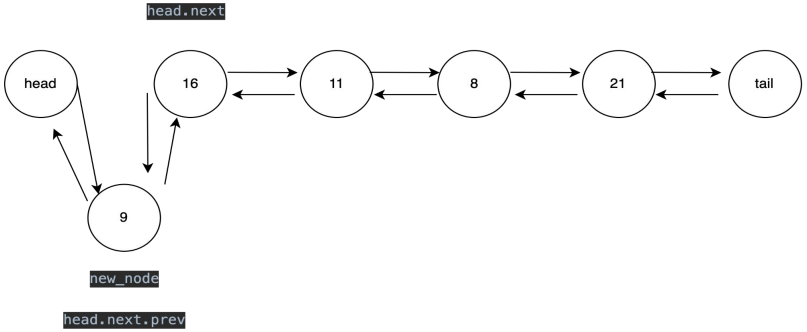
- Раньше мы реализовали двусвязный список так, что сложность вставки в конец был O(n)
- Сейчас мы попробуем реализовать двусвязный список так, что эта сложность сведётся к O(1)
- Для этого мы введем новую переменную tail, которая будет указывать на конец списка. Будем ее каждый раз изменять при вставке элемента.
- Теперь head и tail всегда указывают на один и тот же элемент, а при инициализации очереди у нас всегда по умолчанию есть два элемента, что избавляет нас от ряда проверок.
- Также добавим еще пару штрихов в нашу реализацию списка: head и tail теперь будут играть роль заглушек, то есть они не будут нести в себе полезную информацию, а только лишь указатели. Такие элементы еще называют сторожевыми. Такая структура данных, в которой каждый узел имеет два указателя - на предыдущий и на следующий узлы нам очень пригодится, когда мы коснемся Дека. Но обо всём по порядку)
- При вставке/удалении мы не двигаем ни tail ни head

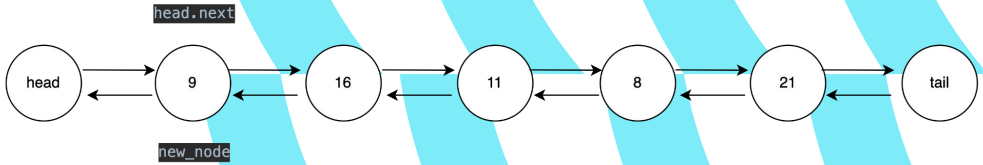




Вставка

- создаем новый узел
- теперь нам надо поменять 4 ссылки:
- новый элемент в качестве следующего ссылается на некогда первый элемент в списке (последний в очереди)
- новый элемент в качестве предыдущего ссылается на head
- некогда первый элемент в списке (последний элемент в очереди) теперь в качестве предыдущего элемента ссылается не на head, а на новый элемент
- нам остается заменить только последнюю ссылку: head теперь ссылается на новый элемент
- новый элемент всегда будет в качестве значения у head.next





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

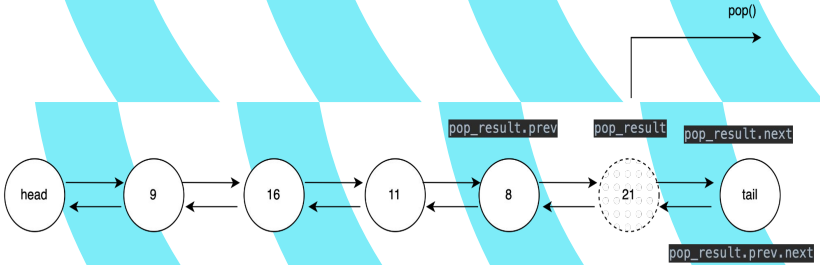
```python
class Queue:
    def __init__(self):
        self.head = Node()
        self.tail = Node()
        # при инициализации head указывает на tail
        self.head.next = self.tail
        # tail, в свою очередь, ссылается на head
        self.tail.prev = self.head
        # размер очереди при ее создания ставим в 0
        self.size = 0
```

```python
def push(self, value):
    # создаем новый узел
    new_node = Node(value)
    # теперь нам надо поменять 4 ссылки:
    # новый элемент в качестве следующего ссылается на некогда
    # первый элемент в списке (последний в очереди)
    new_node.next = self.head.next
    # новый элемент в качестве предыдущего ссылается на head
    new_node.prev = self.head
    # некогда первый элемент в списке
    # (последний элемент в очереди) теперь
    # в качестве предыдущего элемента ссылается не на head,
    # а на новый элемент
    self.head.next.prev = new_node
    # нам остается заменить только последнюю ссылку:
    # head теперь ссылается на новый элемент
    self.head.next = new_node
    self.size += 1
```

Выборка

- если head в качестве next имеет tail - значит список пуст и возвращать нечего
- извлекаем всегда только из начала очереди
- теперь tail в качестве prev (предпоследнего элемента) ссылается на следующий до предпоследнего элемент (tail.prev.prev)
- переписываем next у нового элемента. Теперь next ссылается на tail
- "отцепляем" наш элемент от списка
- Ну и конечно же, декрементируем счетчик

```
def pop(self):
    if self.head.next == self.tail:
        return
    # извлекаем всегда только из начала очереди
    pop_result = self.tail.prev
    # теперь tail в качестве prev (первого элемента очереди) ссылается на второй элемент
    self.tail.prev = pop_result.prev
    # переписываем next у второго элемента. Теперь next ссылается на tail
    pop_result.prev.next = pop_result.next
    # "отцепляем" наш элемент от списка
    pop_result.next = None
    pop_result.prev = None
    # уменьшаем счетчик
    self.size -= 1
    return pop_result.data
```



Очередь на основе связного списка

- Только что мы поговорили про то как реализовать очередь на основе двусвязного списка
- O(1) вставка и удаление на обоих концах это то что нам очень пригодится в следующей теме.
- Новые элементы кладем в конец очереди (начало списка).
- Извлекаем элементы из начала очереди (конец списка).
- Head и tail больше не несут в себе данных, а используются только в качестве сторожевых элементов.
- В отличии от реализации с массивом минимальный контроль за памятью, но необходим контроль за указателями

ДЕК

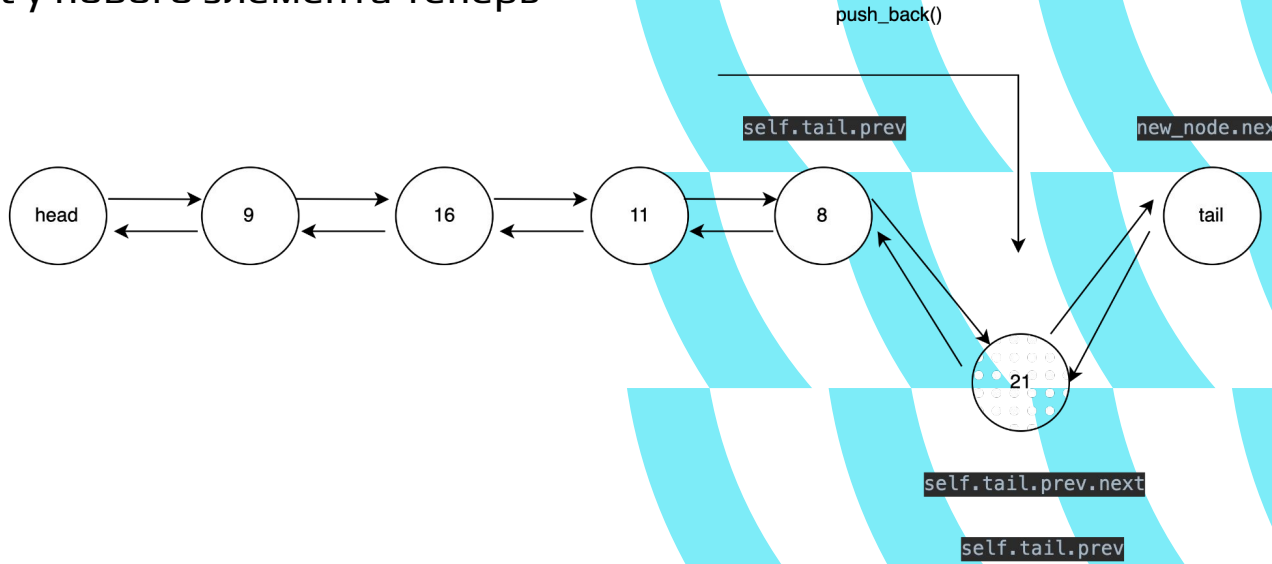
- Deque - double ended queue
- Можем добавлять и извлекать с любой стороны (левая и правая стороны дека)
- Добавляются и извлекаются с каждой стороны в порядке очереди
- Реализуем на двусвязном списке
- И именно на примере дека нам понадобиться реализация списка со сложностью вставки и выборки из начала и конца O(1)

Основные операции

- **push_front** - вставка в начало очереди. Этот метод мы только что с вами реализовали, только назывался он просто push
- **push_back** - вставка в конец очереди. Этот вид вставки нам предстоит с вами разобрать прямо сейчас.
- **pop_front** - извлечение из начала очереди. Тоже уже известный нам как метод pop
- **pop_back** - извлечение из конца очереди. Это вы попробуете реализовать самостоятельно. В качестве подсказки вам будет служить метод pop_front.

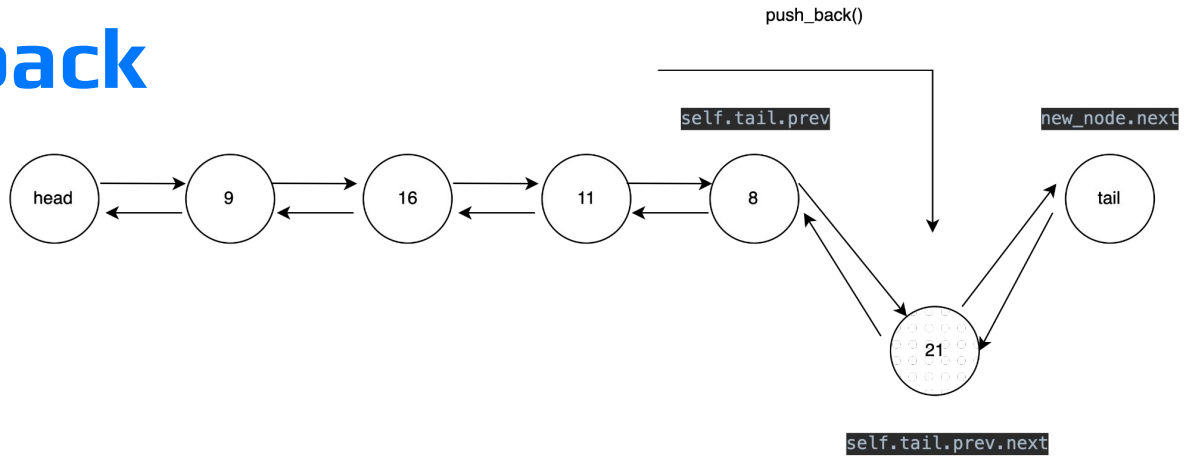
Реализуем вставку в конец push_back

- prev который раньше был у конца списка становится prev для нового элемента
- next который раньше был у предпоследнего элемента должен ссылаться на новый узел
- сам prev у tail теперь указывает на новый элемент
- в свою очередь next у нового элемента теперь ссылается на tail



Реализуем вставку в конец push_back

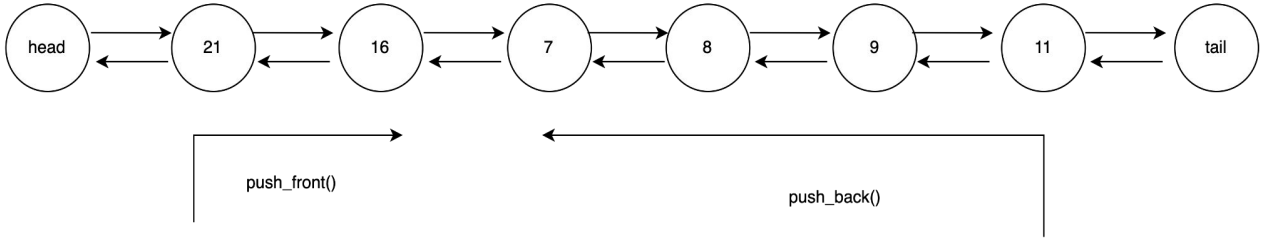
- создаем новый узел
- prev который раньше был у конца списка становится prev для нового элемента
- next который раньше был у предпоследнего элемента должен ссылаться на новый узел
- сам prev у tail теперь указывает на новый элемент
- в свою очередь next у нового элемента теперь ссылается на tail




```
def push_back(self, value):
    # создаем новый узел
    new_node = Node(value)
    # prev который раньше был у конца списка
    # становится prev для нового элемента
    new_node.prev = self.tail.prev
    # next который раньше был у предпоследнего
    # элемента должен ссылаться на новый узел
    self.tail.prev.next = new_node
    # сам prev у tail теперь указывает на новый элемент
    self.tail.prev = new_node
    # в свою очередь next у нового элемента
    # теперь ссылается на tail
    new_node.next = self.tail
```

Как это работает

- вызывая push_back() мы «проталкиваем» 7 ближе к head с каждым вызовом
- каждый вызов push_front() проталкивает к tail 16
- в этой ситуации наш метод pop_front, который вы реализуете самостоятельно, должен вернуть в начале 21, затем 16, 7 и так далее до tail.




```
deque = Deque()

deque.push_back(7)
deque.push_back(8)
deque.push_back(9)
deque.push_back(11)
deque.push_front(16)
deque.push_front(21)


print(deque.print())

print(deque.pop_back())
print(deque.pop_back())
print(deque.pop_back())
print(deque.pop_back())
print(deque.pop_back())
print(deque.pop_back())
```

```
[ 21 16 7 8 9 11 ]
11
9
8
7
16
21
```