Очередь, стек

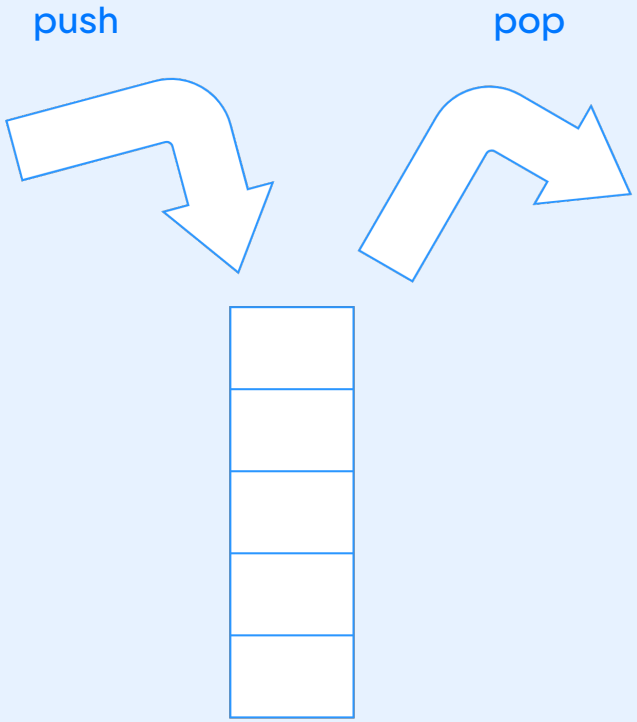
education

Стек, очередь. Двойная очередь

- В первом уроке мы изучили линейные структуры данных, обсудили преимущества
и недостатки того или иного представления в памяти.
- Поговорили о том, в каких ситуациях и что лучше выбрать.
- Сейчас мы изучим абстрактные типы данных, которые могут быть реализованы
при помощи уже известных нам массивов и списков.
- Мы поговорим с вами о стеке и об очередях. Узнаем, какие принципы лежат в их основе, а также решим хрестоматийную задачу, которая даст понимание того, для чего же все эти типы нужны.
- Начнём наш урок с абстрактного типа данных, который называется «стек».

Стек

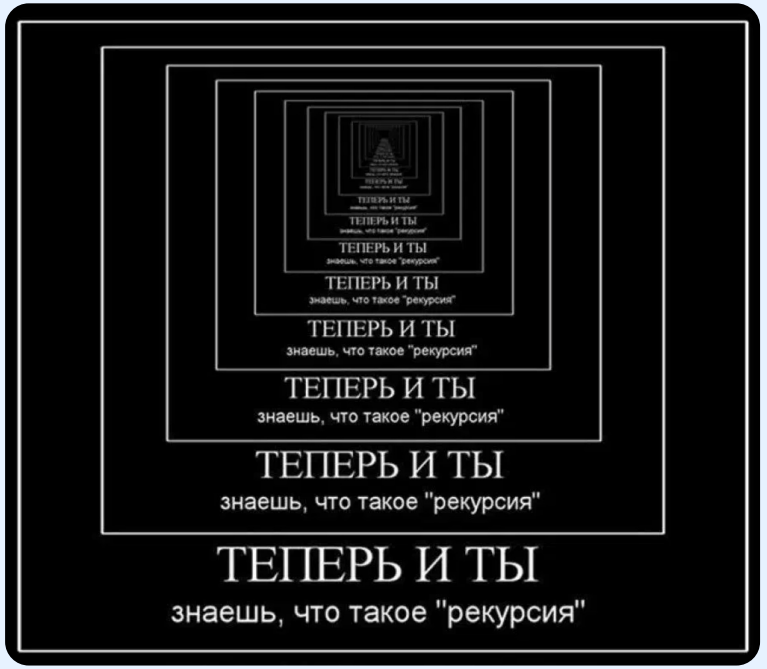
- LIFO (last in first out) — первый вошёл, последний вышел.
- Добавление и удаление в этой структуре возможно только с одного конца.
- Типичный пример для этой структуры данных — стопка бумаг.



education

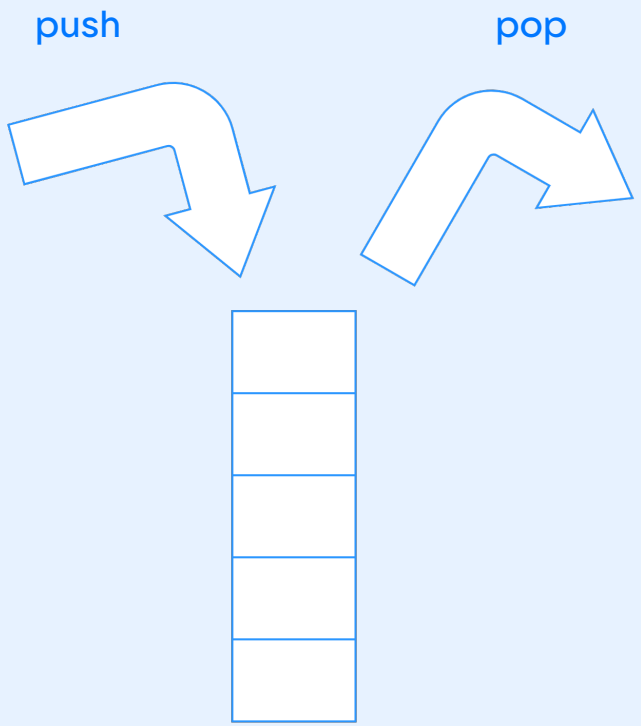
Области применения

- 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.
- 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.
- 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 ни 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 выполнять за O(1).



Реализация на списке

1

Реализация стека на списке

2

В нашем случае достаточно односвязного списка

3

Push будет писать данные, как это делал append_front в уже написанной нами функции

4

Pop просто будет возвращать head. Главное не забывать переписывать указатели

Реализация в коде

- Метод push можно скопировать из append_front, разве что заменим head на top.
- 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 нём хранится больше одного значения, переписываем значение вершины на следующий элемент.
- Если стек пуст, то устанавливаем вершину в значение None.
- Возвращаем значение, хранящиеся в вершине стека.

Код стека на списке

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

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
    # new_node.next, self.head = self.head, new_node
```

```
  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:
        # 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

1

- Так как мы не знаем, сколько данных будет приходить в наш стек, лучше всего использовать саморасширяющийся массив

2

- Принцип идентичен реализации на списке, но вставлять мы будем не в начало, а в конец, чтобы обеспечить амортизационную сложность O(1)

3

- Сокращение размера массива в этом случае будет эквивалентно сокращению саморасширяющегося массива

Реализация в коде

1

Должно пойти в Д/З

2

Плюс к Д/З реализовать метод empty(), который бы возвращал true в случае, если стек пуст, и false, если он полон

3

Также надо реализовать стек с методом size(), который будет возвращать текущий размер стека

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 скобочки
есть в нужном месте закрывающая)

[(]), ((()), ({[})], ((()), [[]]]

Невалидные последовательности

education

Решение

- Используем список как стек, только методы append (push) и pop.
- Идём в цикле по нашей последовательности.
- Если скобка открывающая, то пишем её в стек.
- Если на итерации мы встретили закрывающую скобку, но стек уже пустой, значит, последовательность невалидна
- Если скобка, которая в данный момент находится на вершине стека, не является открывающей для текущей скобки, последовательность также не валидна при этом во время проверки
- Если скобка закрывающая, то мы удаляем с вершины соответствующую открывающую скобку, то есть освобождаем стек на один элемент.

education

```
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

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

education

Основные операции

1

Push (enqueue) — кладёт данные в конец очереди

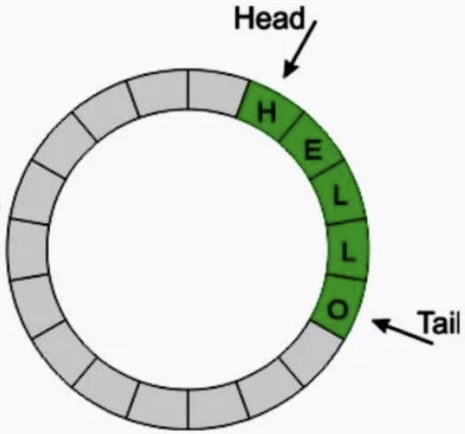
2

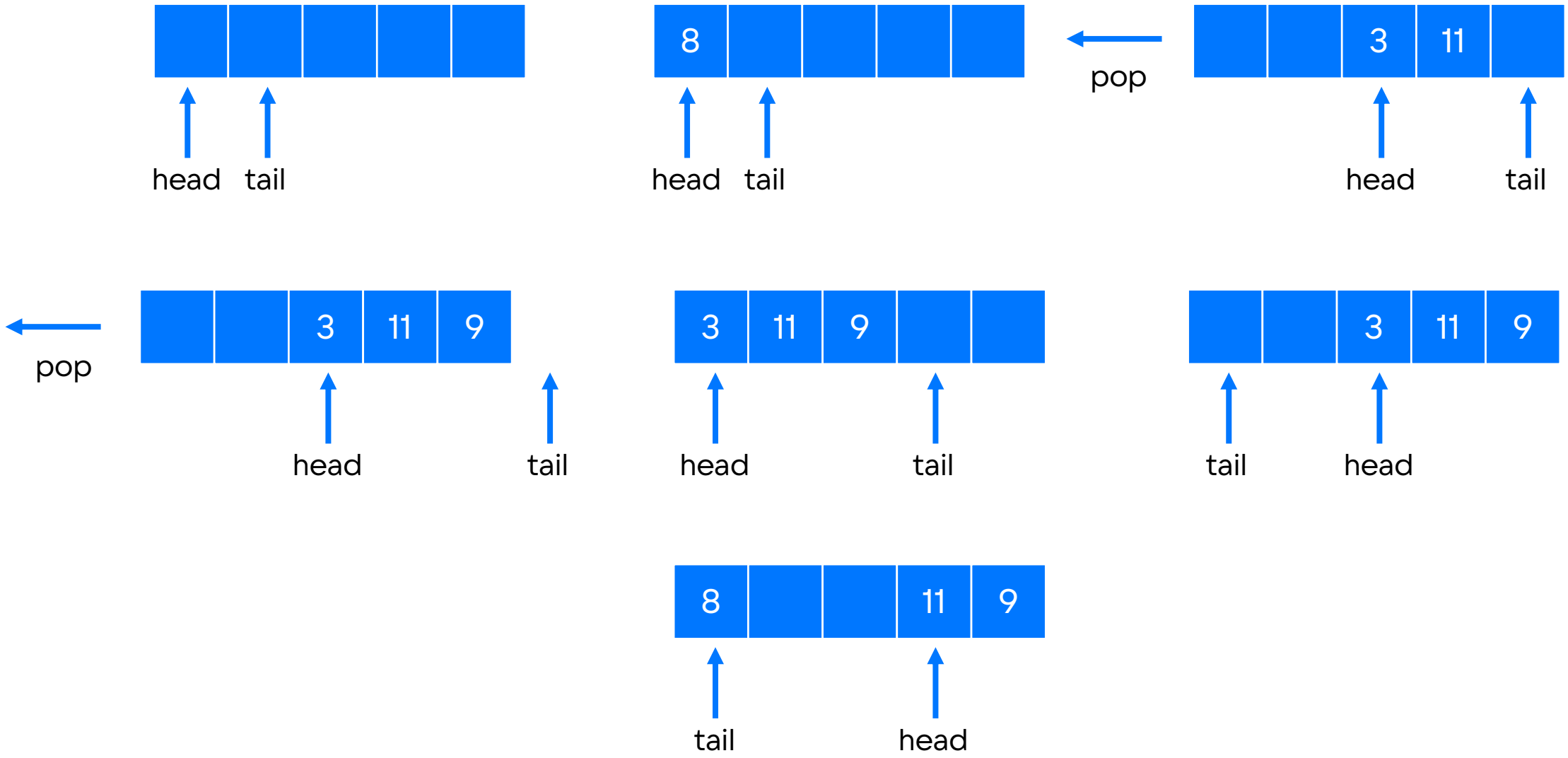
Pop (dequeue) — извлекает данные только из начала очереди, то есть первым вернётся только тот элемент, который был добавлен первым

Так же, как и у стека, сложность этих двух операций — O(1)

Кольцевой буфер (циклический массив)

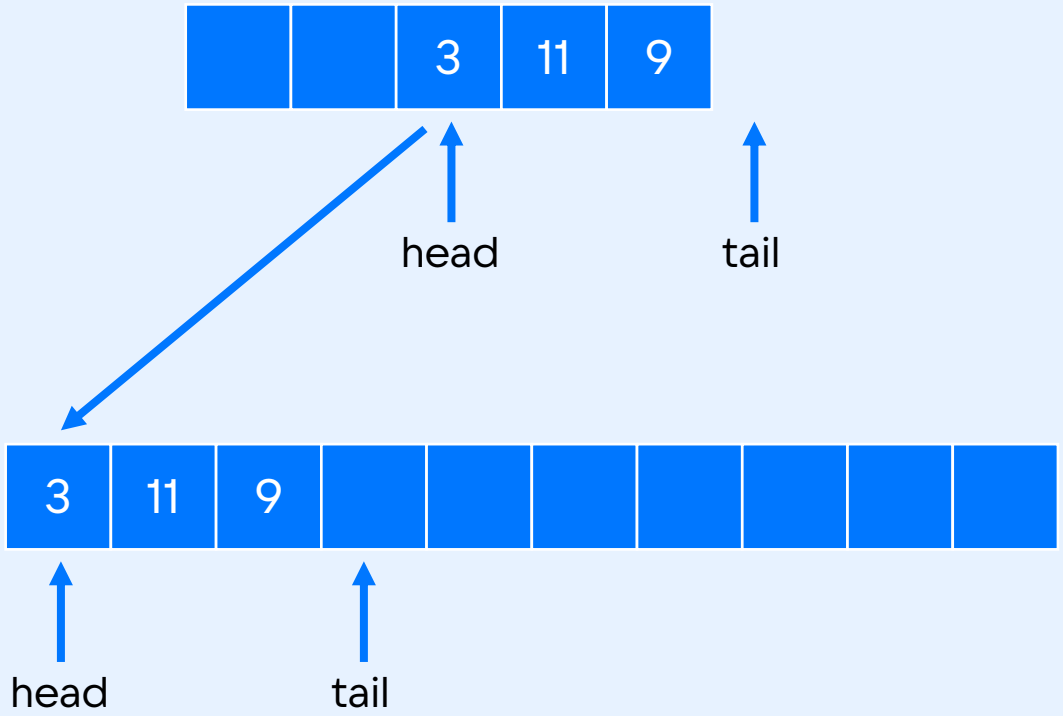
- Вводим две новые переменные — head и tail — индексы начала и конца очереди.
- В пустом массиве они указывают на нулевой элемент.
- При добавлении мы вставляем элемент в ячейку с индексом tail. После вставки инкриментируем tail. Выбираем из ячейки с индексом head.
- В какой-то момент при вставке tail может выйти за пределы массива, что недопустимо.
- Можно попробовать двигать все элементы в начало — тогда сложность начнёт стремиться к O(n).
- Выходом в такой ситуации может быть перемещение tail на нулевую ячейку при условии, когда заполненность массива, то есть size, будет равна capacity.




head
tail
8
head
tail
pop
3
11
head
tail
pop
3
11
9
head
tail
3
11
9
head
tail
3
11
9
tail
head
8
11
9
tail
head

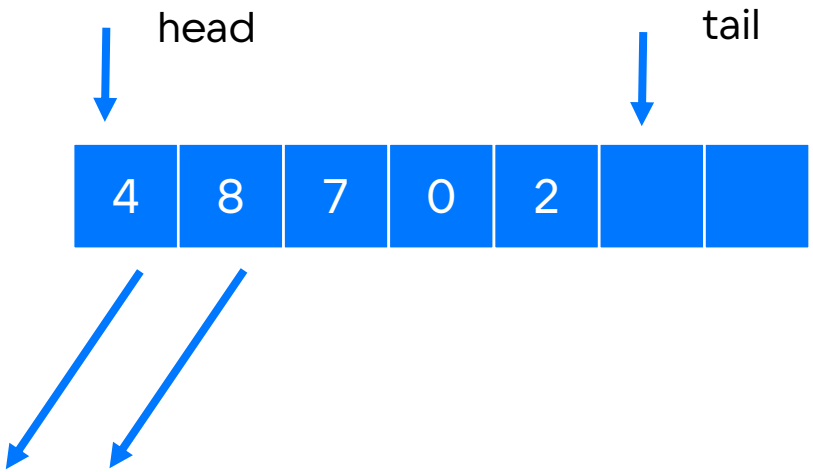
Реализация

- Используем саморасширяющийся массив.
- При чтении проверяем наличие элементов в очереди, если они есть, возвращаем их 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 учётом, что head должен указывать на нулевую ячейку.

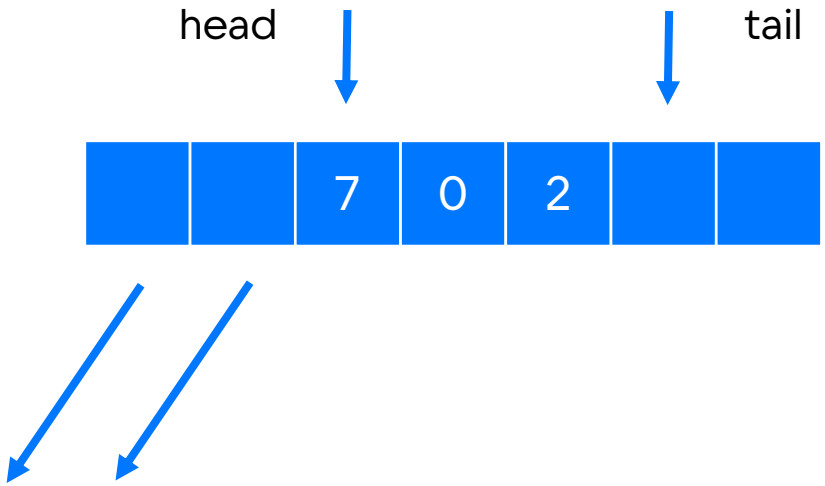


education

Пустая очередь — head и tail равны между собой



При выборке последовательно сдвигаем head вправо



Сложность операций

1

Вставка O(1). Вставляем только в конец очереди

2

Выборка O(1). Выбираем только из начала очереди

3

Учитываем необходимость увеличения массива

4

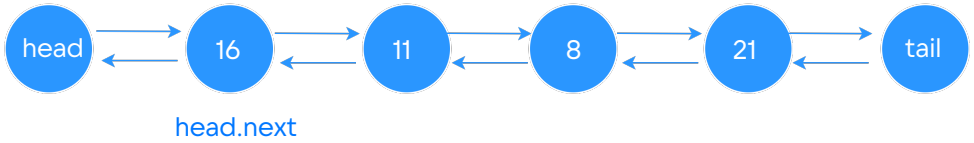
Не забываем очищать неиспользуемую память

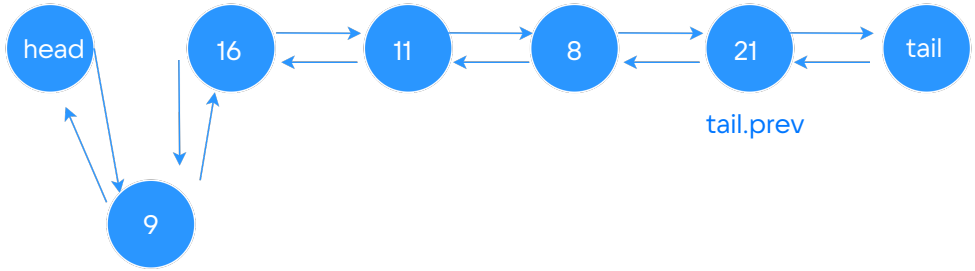
Домашнее задание

- Попробуйте дома реализовать очередь на расширяющемся массиве.
- Вам необходимо реализовать метод push(int), который будет принимать число и записывать его в очередь.
- Метод pop(), который будет возвращать следующий на очереди элемент и удалять его.
- Также реализуйте метод resize (array, int), который будет реалоцировать память — увеличивать вдвое capacity в случае, если массив заполнился и происходит вставка.
- Задача со звёздочкой — реализовать очередь на массиве таким образом, чтобы, в случае если size ÷ 4 ≤ capacity, уменьшалось бы capacity в два раза.

Очередь на основе двусвязного списка

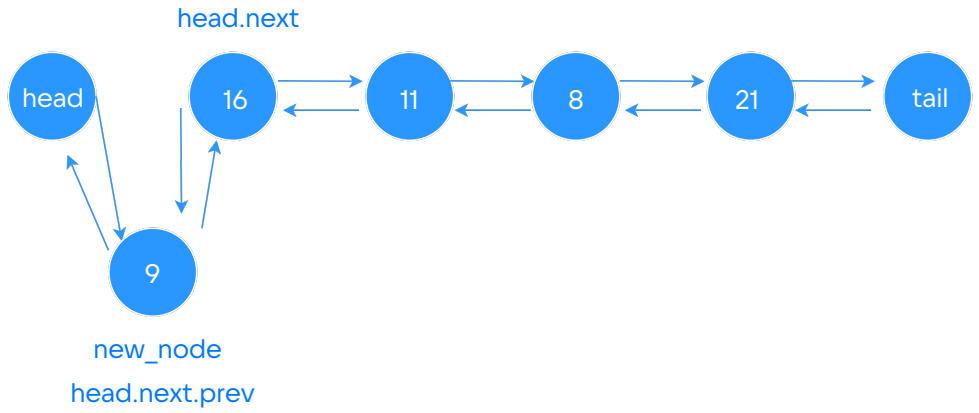
- Попробуем реализовать двусвязный список так, что эта сложность сведётся к O(1).
- Для этого мы введём новую переменную tail, которая будет указывать на конец списка. Будем её каждый раз изменять при вставке элемента.
- Теперь head и tail всегда указывают на один и тот же элемент, а при инициализации очереди у нас всегда по умолчанию есть два элемента, что избавляет нас от ряда проверок.
- head и tail теперь будут играть роль заглушек, то есть они не будут нести в себе полезную информацию, а только лишь указатели. Такие элементы ещё называют сторожевыми.
- Такая структура данных, в которой каждый узел имеет два указателя — на предыдущий и на следующий узлы — нам очень пригодится, когда мы коснёмся дека.

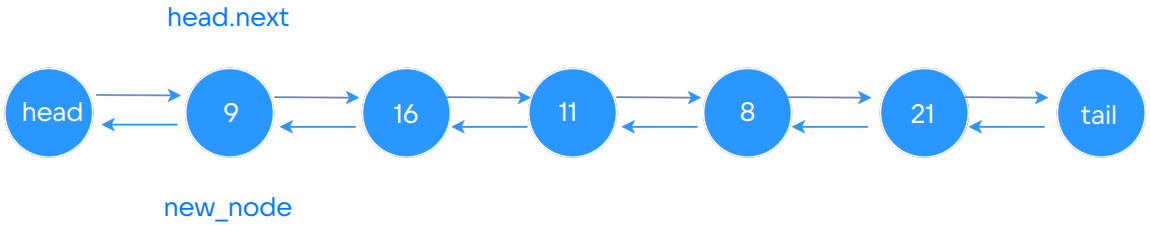




Попробуем реализовать это в коде: Вставка

- Создаём 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 остаётся заменить только последнюю ссылку: head теперь ссылается на новый элемент.
- Ну и, конечно, инкрементируем счётчик.





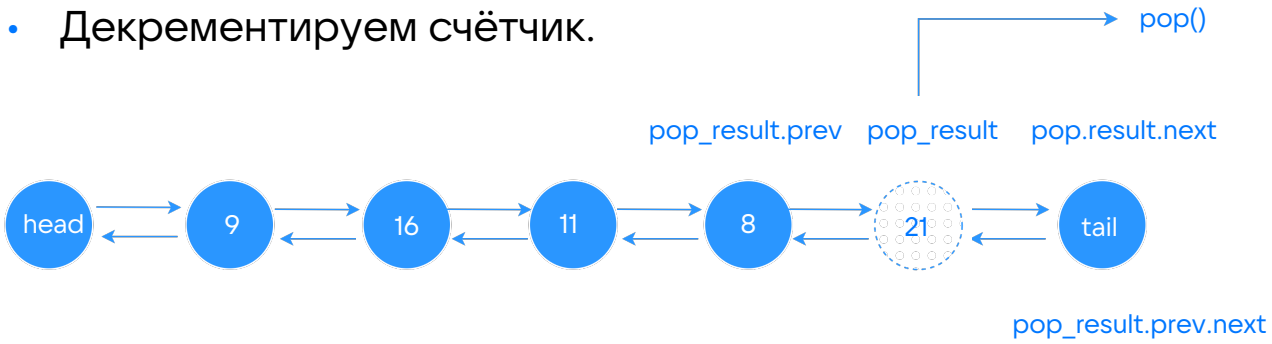
```
class Node:
  def __init__(self, data=None):
    self.data = data
    self.next = None
    self.prev = None


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
```

```
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
```

Попробуем реализовать это в коде: 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 второго элемента. Теперь next ссылается на tail.
- «Отцепляем» наш элемент от списка.
- Декрементируем счётчик.




```
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 связанного 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 кладём в конец очереди (начало списка), для этого поддерживаем указатель на него.
- Head и tail больше не несут в себе данных, а используются только в качестве сторожевых элементов.
- В отличие от реализации с массивом — минимальный контроль за памятью, но необходим контроль за указателями.

Домашнее задание

1

Для закрепления материала я предлагаю вам самостоятельно реализовать метод print_queue, который бы выводил в консоль все элементы очереди от tail к head

2

Идея метода довольно проста — пройтись в цикле до тех пор, пока next != null, и выполнить команду вывода в консоль

ДЕК

Duque — 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 понадобится 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

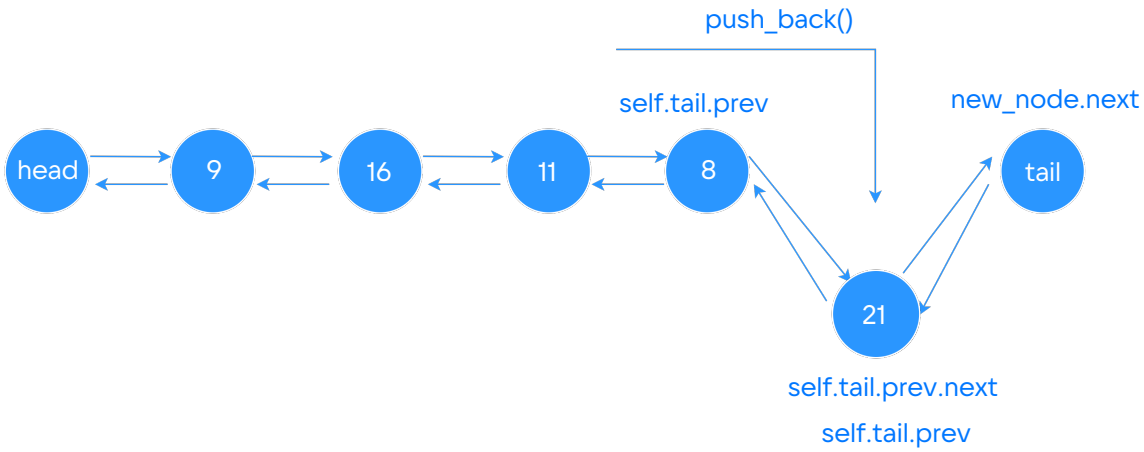
pop_front

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

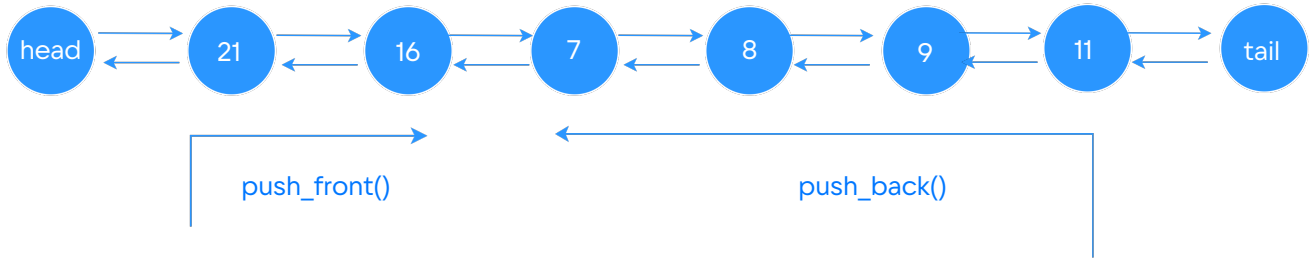
Реализуем вставку в конец push_back




```
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 ваш метод pop_front, который вы реализуете самостоятельно, должен вернуть сначала 21, затем 16, 7 и так далее до tail.




```
de_queue = Dequeue()

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


print(de_queue.print())

print(de_queue.pop_back())
print(de_queue.pop_back())
print(de_queue.pop_back())
print(de_queue.pop_back())
print(de_queue.pop_back())
print(de_queue.pop_back())
```

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

education

Будем ВКонтакте!