Стандартные контейнеры

17/02/2023

Проблема

Как хранить данные?

Менеджер памяти

Можем считать, что ОС выдает нам некоторое место в памяти, куда можно писать данные, обычно некоторый непрерывный блок из n байт. Назовем это $malloc(n)$ (потому что, собственно, оно так и называется).

Хотим укладывать данные в эту память

Массив статического размера

Если наш тип данных T занимает b байт, и в нем n элементов, то мы можем решить задачу просто

```
def create(b, n):
    allocated = malloc(b * n)
    arr[i] = T(allocated[i * b: (i + 1) * b])
```

Но неужели нужно звать новый $alloc$ на каждое добавление нового элемента?

Capacity

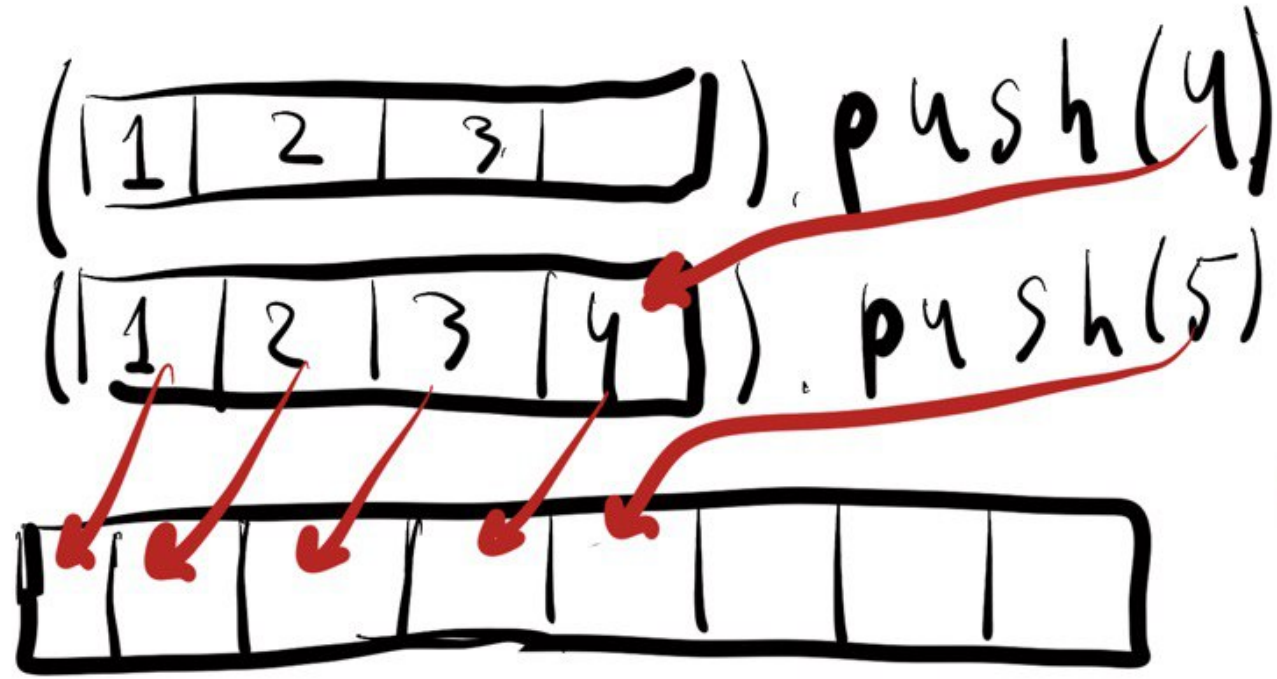
Давайте заранее оценим, сколько памяти нам может максимально понадобится, и выделим ровно столько памяти

```
create(b, MAXN)
```

Динамический массив

Давайте хранить про запас в два раза больше памяти, чем нам нужно, и удваивать количество, если не хватило

```
class Stack:
    def __init__(self):
        self.size, self.end = 1, 0
        self.data = create(b, self.size)

    def push(self, x):
        if self.end == self.size:
            old = self.data
            self.size *= 2
            self.data = create(b, self.size)
            copy(old, self.data)
        self.data[self.end] = x
        self.end += 1

    def pop(self):
        self.end -= 1
        return self.data[self.end]
```



O(1).

O(n).

Динамический массив, анализ

- Выделено памяти: $O(n)$
- А какая сложность у $push$?
- best-case: $O(1)$
- worst-case: $O(n)$
- А в среднем?

"Амортизированный" анализ

Иногда бывают "медленные" операции, иногда "быстрые".

Пусть был массив размера n и мы сделали еще n добавлений.

$$O_{avg} = \frac{\sum_{fast} O(1) + \sum_{slow} O(n)}{n} = \frac{n \cdot 1 + 1 \cdot n}{n} = O(1)$$

Ура!

Проверка

Как сделать метод `pop` в нашем массиве, чтобы среднее время выполнения было `pop` было $O(1)$, и мы чистили ненужную память (оставляя *какой-то* запас)?

В питоне

Это вам не нужно :)

```
a = []
a.append(1)
print(a.pop())
# 1
print(a)
# []
```

Контейнеры

Сделав массив, мы по совместительству сделали стек! Вопрос только в том, что это такое и зачем

Проблема: иногда мы хотим, чтобы данные в контейнере имели некоторый порядок, а мы могли просто "добавить" и "достать" (push/pop).

Стек - LIFO-структура. Элемент вытащится раньше, если его добавлляли позже. Пример - стопка книг, вызов функции.

Еще мы хотим FIFO-структуру, чтобы первый добавленный был вытащен первым. Это назвыается очередью. Пример - очередь в магазине, таск-менеджер.

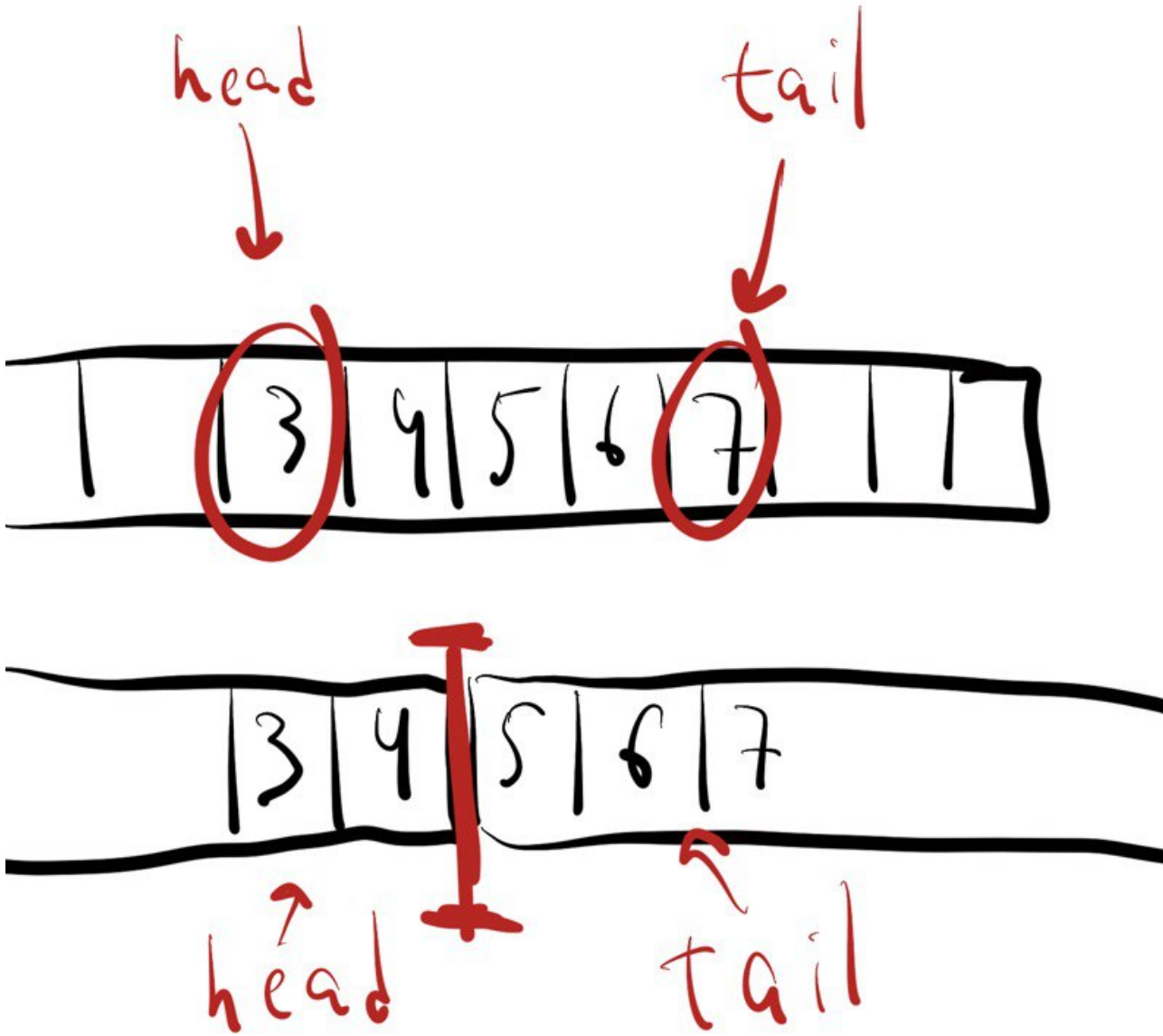
Очередь

```
class Queue:
    def __init__(self):
        self.data = []
        self.head = 0
        self.tail = 0
        self.size = 0

    def push(self, x):
        if self.tail == self.size:
            self.size = max(1, self.size * 2)
            # reallocate + copy
        self.data[self.tail] = x
        self.tail += 1

    def pop(self):
        if self.head == self.tail:
            return None
        res = self.data[self.head]
        self.head += 1
        return res
```

Очередь через стеки

```
class Queue:
    def __init__(self):
        self.head = Stack()
        self.tail = Stack()

    def push(self, x):
        self.tail.push(x)

    def pop(self):
        if self.head.empty():
            while not self.tail.empty():
                self.head.push(self.tail.pop())
        return self.head.pop()
```

Какая из реализаций лучше?



Проверка

Какое среднее времмя на `push` / `pop` в очереди? А лучшее/худшее?

Два указателя

Иногда мы хотим обработать *подотрезки* массива, обладающие некоторым свойством

Задача

Найти самый длинный подотрезок массива натуральных чисел с суммой меньше k

Решение

Заметим, что если у нас есть подотрезок, и мы добавляем в него еще один элемент, сумма увеличивается, а если удаляем, то уменьшается.

Давайте пройдем по массиву очередью и будем добавлять элементы в конец, пока сумма меньше k, и удалять из начала, если сумма больше или равна k. *Релаксируем* ответ `ans = max(ans, queue.size)`, когда сумма меньше k.

Очередь можно сделать неявной, с помощью двух указателей на начало и конец массива

Алгоритм

```
ans = 0
right = 0
current_sum = 0
for left in range(0, n):
    if current_sum < k:
        ans = max(ans, right - left)
    while right < n and current_sum < k:
        current_sum += a[right]
        right += 1
        if current_sum < k:
            ans = max(ans, right - left)
    current_sum -= a[left]
```

List

Вместо того, чтобы хранить данные последовательно, положим в элемент указание, где искать следующий.

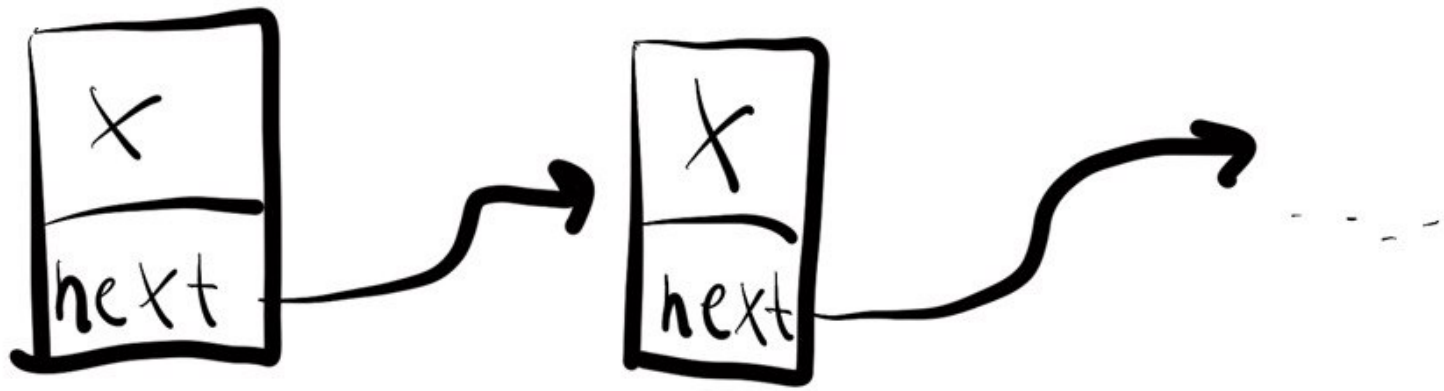
Если добавим указатель на предыдущий, то получим Linked List

Не путайте с List в python!

List

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

    def insert(self, node):
        node.next = self.next
        self.next = node

    def print(self):
        node = self
        while node != None:
            print(node.val)
            node = node.next
```

Можно отдельно создавать Node для вершин и структуру List, которая будет содержать указатели на начало и конец, например


x
next
x
next

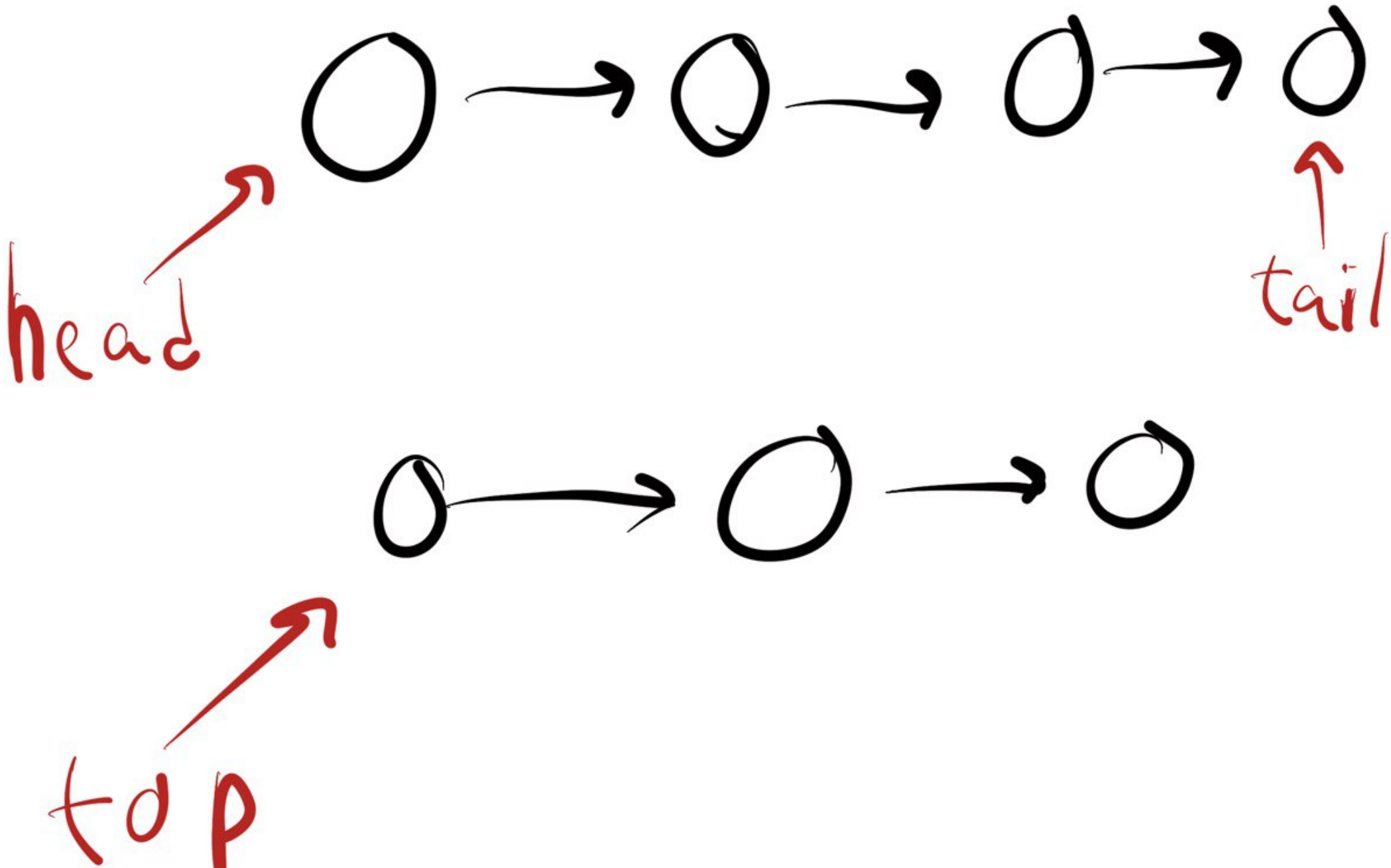
Вариации

Иногда кроме поля `node.next` добавляют поле `node.prev`. Это усложняет некоторые операции, но зато получится нормально удалять (попробуйте понять, почему без `node.prev` не получится сделать удаление `node` из списка).

Вопрос: Можно ли хоть как-то удалять без `node.prev`? Как будут выглядеть операции удаления? Когда в них есть смысл?

Stack, Queue

Можно ли реализовать через список? Какой трейд-офф?


head
tail
top

Трейдоффы

	Array	List
$find$	$O(1)$	$O(n)$
$push^*$	$O^*(1)$	$O(1)$
pop^*	$O^*(1)$	$O(1)$
$push^+$	$O(n)$	$find + O(1)$
pop^+	$O(n)$	$find + O(1)$

Аналогично для Stack/Queue построенных поверх Array/List.

Куда дальше: оптимизации

Мы столкнулись с фундаментальной проблемой - мы либо храним данные "плоско" и можем легко делать поиск за $O(1)$, либо храним элементы со связями - и тогда можем легко делать добавления и удаления.

Теперь хочется как-то объединить подходы, чтобы у нас не было операций за $O(n)$

Куда дальше: функционально

На самом деле, можно думать про Array и List как про оптимальные (*по Парето*) структуры - В массиве очень быстрая индексация, и нельзя улучшить произвольное добавление, не пожертвовав ей. В списке быстрое обновление, и нельзя сделать быстрый поиск, не пожертвовав им.

В структурах "посередине" индексы теряют смысл - либо отсутствует понятие "упорядоченности" (привет, хеш-таблицы!), либо перед элементом может встать другой элемент, чем изменит все остальные индексы (привет, деревья!)

Вопрос Почему нас это не волновало в массивах? а в списках?

Новая "упорядоченность"

- $push(x)$ делает то же самое
- $pop(x)$ делает то же самое
- $get(x)$ возвращает элемент, имеющий значение x.
- дополнительно: $lower\_bound(x)$ возвращает минимальный элемент, больше или равный x.

У нас пропадает индекс, теперь только поиск по значению. Ограничений на значение, в целом, нет. Обычно достаточно предоставить компаратор. Чаще всего используют числа.

Комбинируем

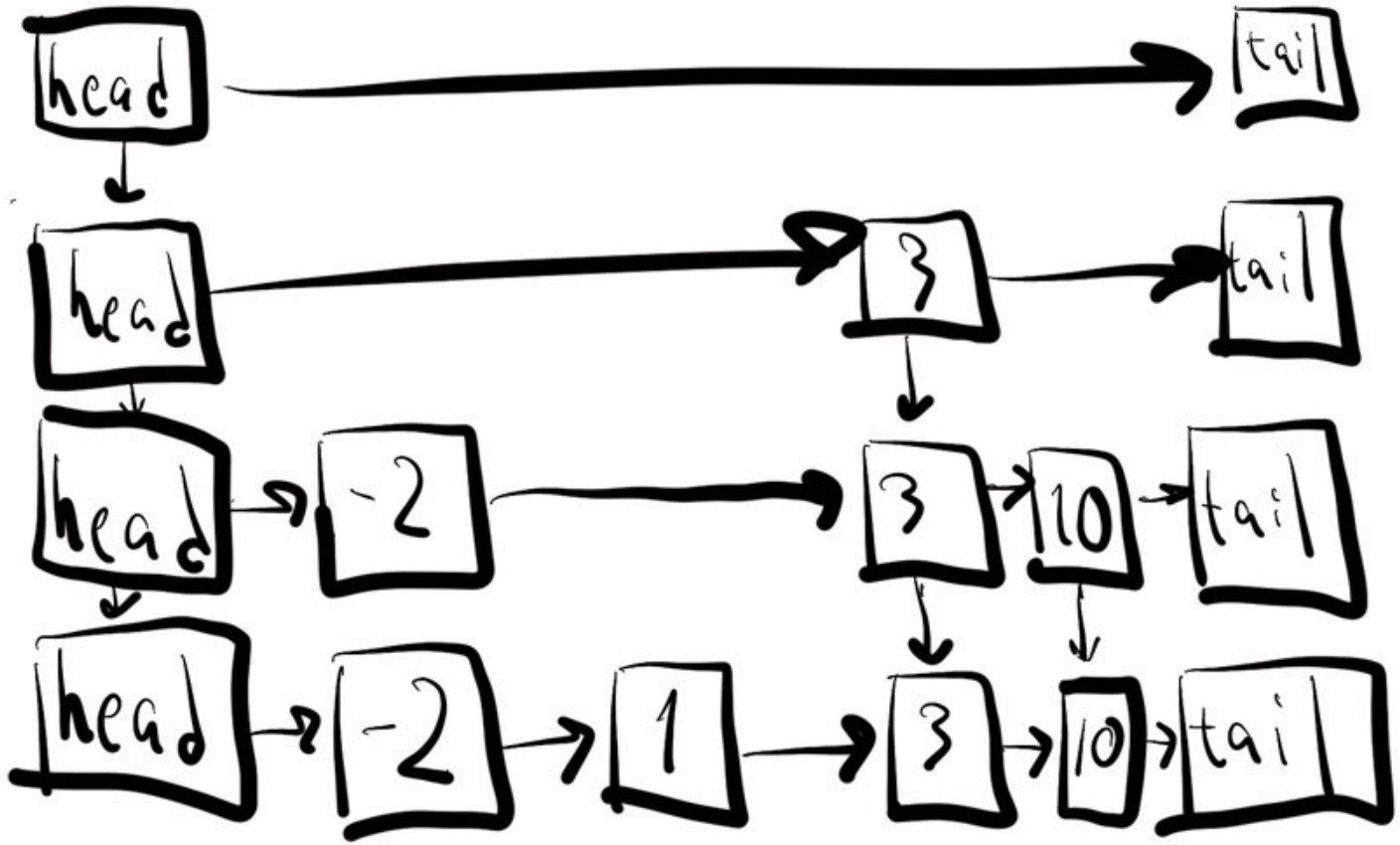
Давайте хранить данные связанно, чтобы мы могли делать быстрые обновления.

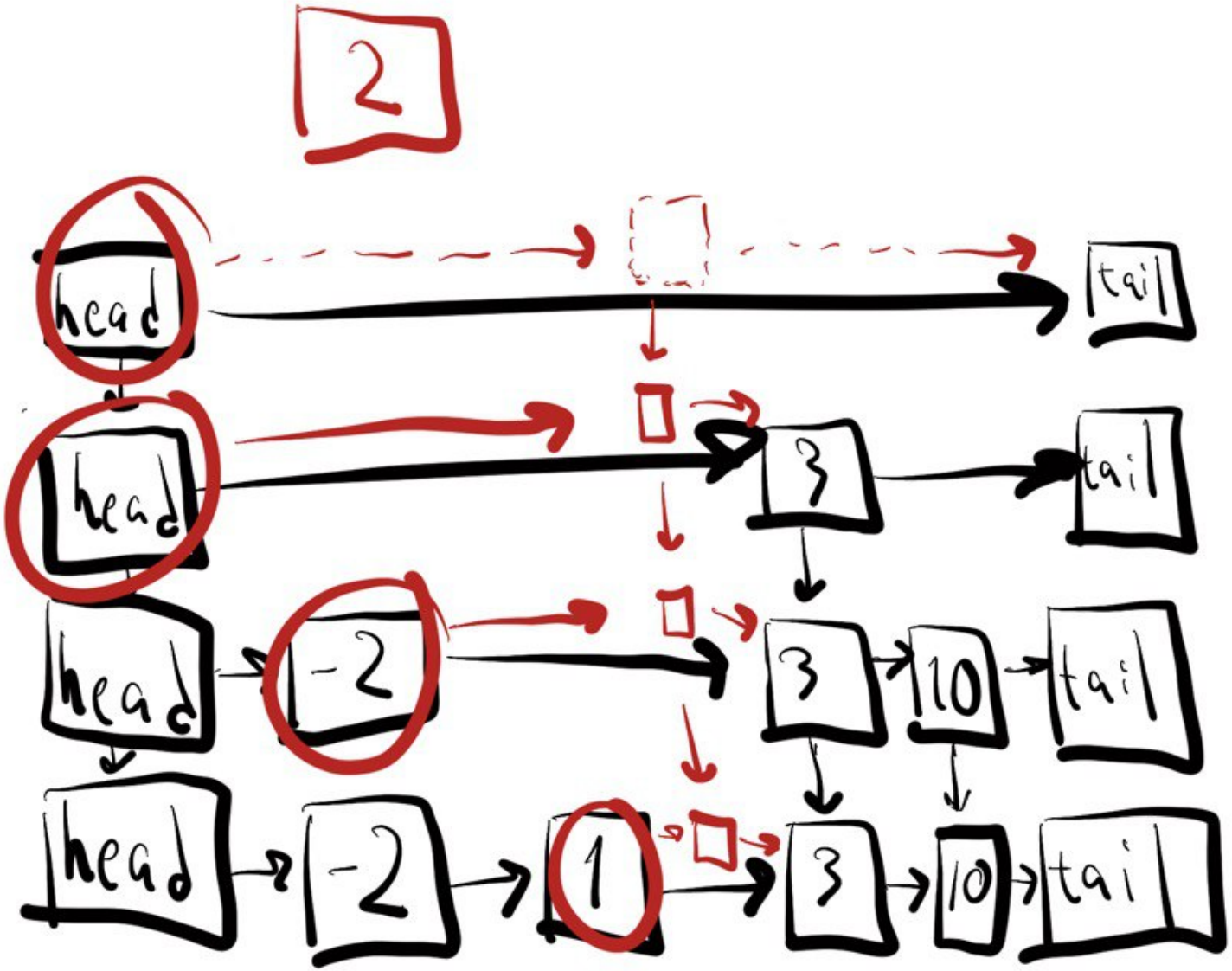
Но так, чтобы структура данных позволяла делать быстрый поиск.

Skip-list

Выкинем из списка половину элементов.

Поиск стал в два раза быстрее!


head
tail
head
3
tail
head
-2
3
10
tail
head
-2
1
3
10
tail


2
head
tail
head
3
tail
head
-2
3
10
tail
head
-2
1
3
10
tail

Асимптотика

На нижнем слое у нас все n узлов. На следующем $n/2$ и так далее. Матожидание количества узлов $O(n)$

Для добавления мы на каждом слое обновляем один узел, для поиска мы на каждом слое двигаемся на $O(1)$, поэтому обе операции за $O(\log n)$

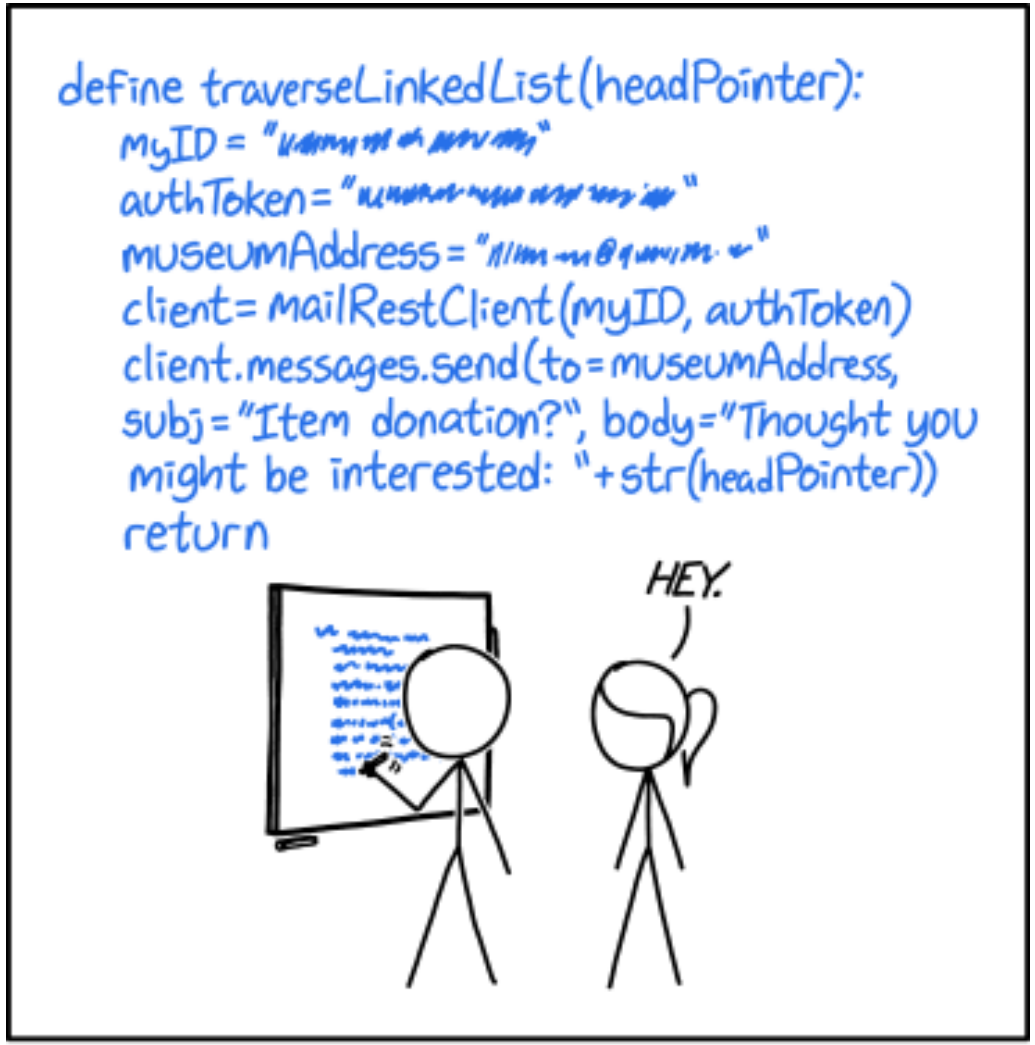
Псевдокод

```
HEAD = Node(-INF)

def find(x, head):
    node = head
    while node.next_level is not None:
        while node.next is not None and node.next.val <= x:
            node = node.next
        node = node.next_level
    return node

def insert(value):
    node = find(value, HEAD)
    new_node = Node(value)
    while node.prev_level is not None:
        new_node.prev, node.next.prev = node, new_node
        new_node.next, node.next = node.next, new_node
        if random.choice([0, 1]) == 1:
            new_node.prev_level = node.prev_level
            break
        node = node.prev_level
        new_node.prev_level = Node(value)
        new_node, new_node.next_level = new_node.prev_level, new_node
```

Redis: ordered set

- ZADD adds a new member and associated score to a sorted set. If the member already exists, the score is updated.
- ZCOUNT Returns the number of elements in the sorted set at key with a score between min and max.
- ZRANK returns the rank of the provided member, assuming the sorted is in ascending order.

Мем



CODING INTERVIEW TIP: INTERVIEWERS GET REALLY MAD WHEN YOU TRY TO DONATE THEIR LINKED LISTS TO A TECHNOLOGY MUSEUM.

Пет-проект

Создать свою очередь задач.

Вот тут можно найти данные по нашей таблице результатов https://algocode.ru/standings_data/students_2023/

Можно сделать телеграм-бота, который высчитывает пользователю, сколько надо решить задач до оценки "отлично". Только такие долгие задачи, как поход в веб-сервис и обработку данных, хочется делать изолированными. Хочется, чтобы один поток (воркер) постепенно "разбирал" задачи, а телеграм-бот эти задачи добавлял.

Пользователь отправляет сообщение в бота, оно уходит в очередь на обработку, а после обработки бот отправляет обратно результат.

Что еще почитать

- https://docs.python.org/3/library/collections.html#deque-objects
- Метод потенциалов
- https://mecha-mind.medium.com/redis-sorted-sets-and-skip-lists-4f849d188a33
- Очереди задач: Kafka, RabbitMQ