

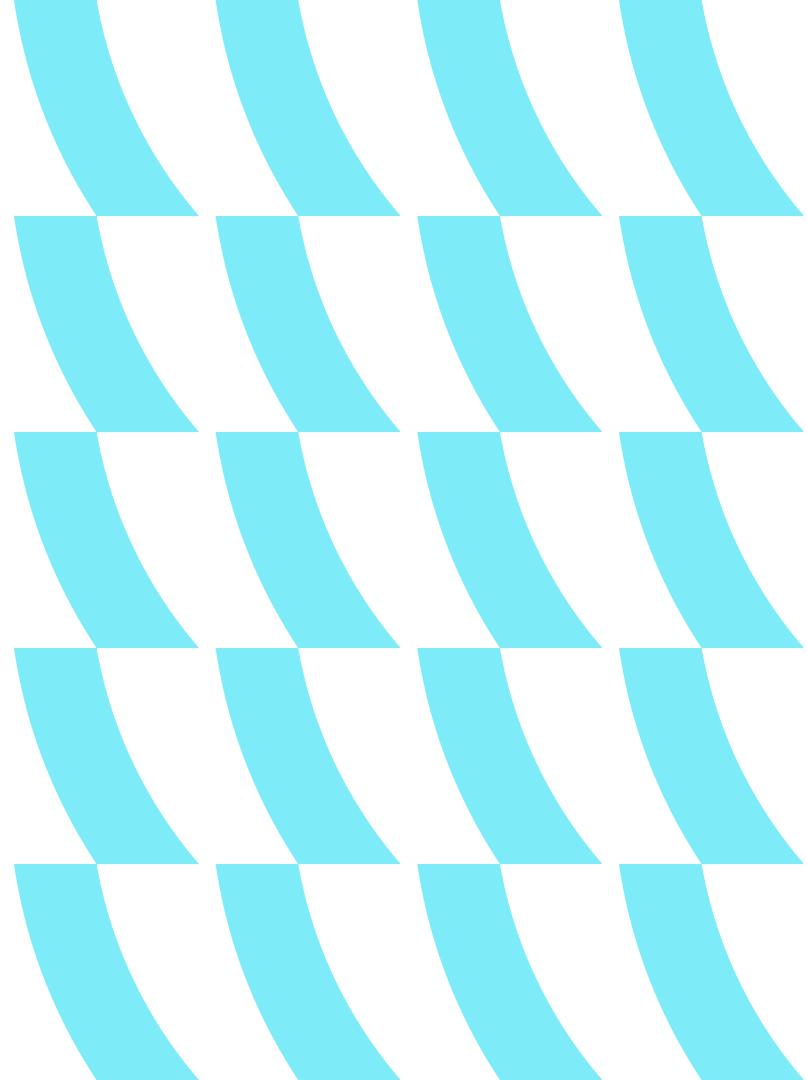
Кучи

max-куча
пирамидальная сортировка
min-куча
Виды куч
Красно-чёрное дерево



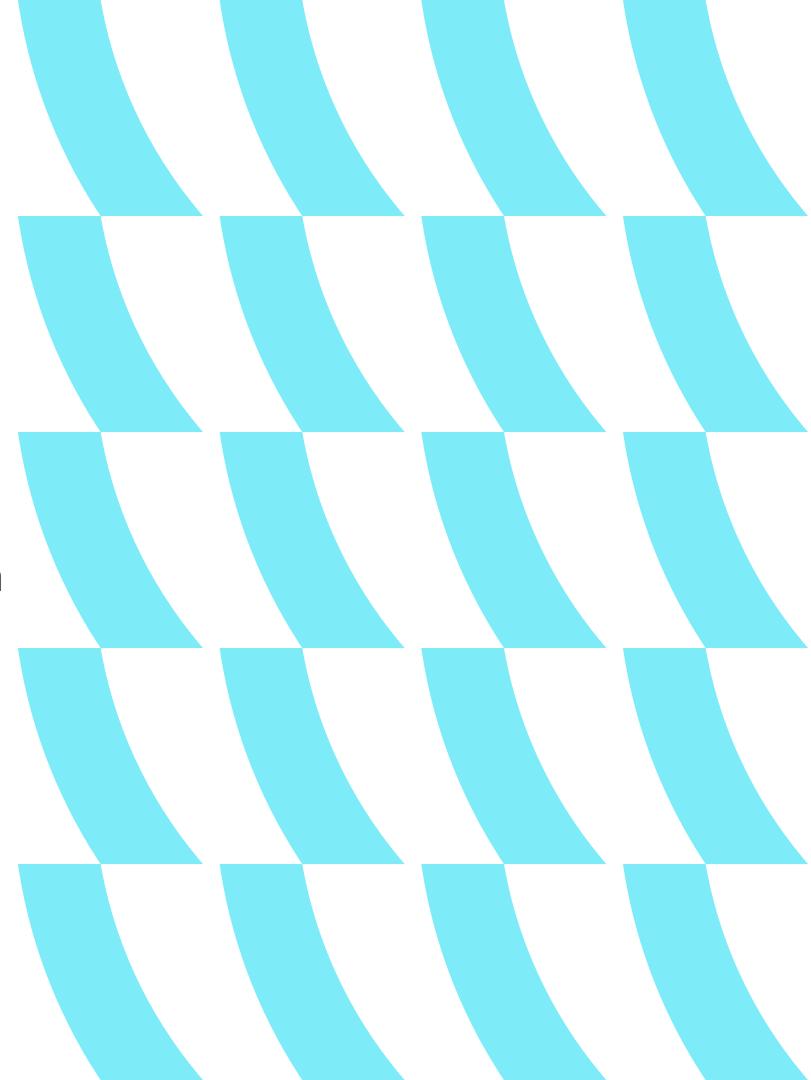
Определим проблему

- Необходимо иметь быстрый доступ к максимальному элементу множества
- Поиск максимального элемента $O(n)$ - линейный поиск
- Вставка $O(1)$
- Можно хранить отсортированный массив
- Получение $O(1)$
- Вставка: $O(\log n)$ - на поиск плюс $O(n)$ на вставку



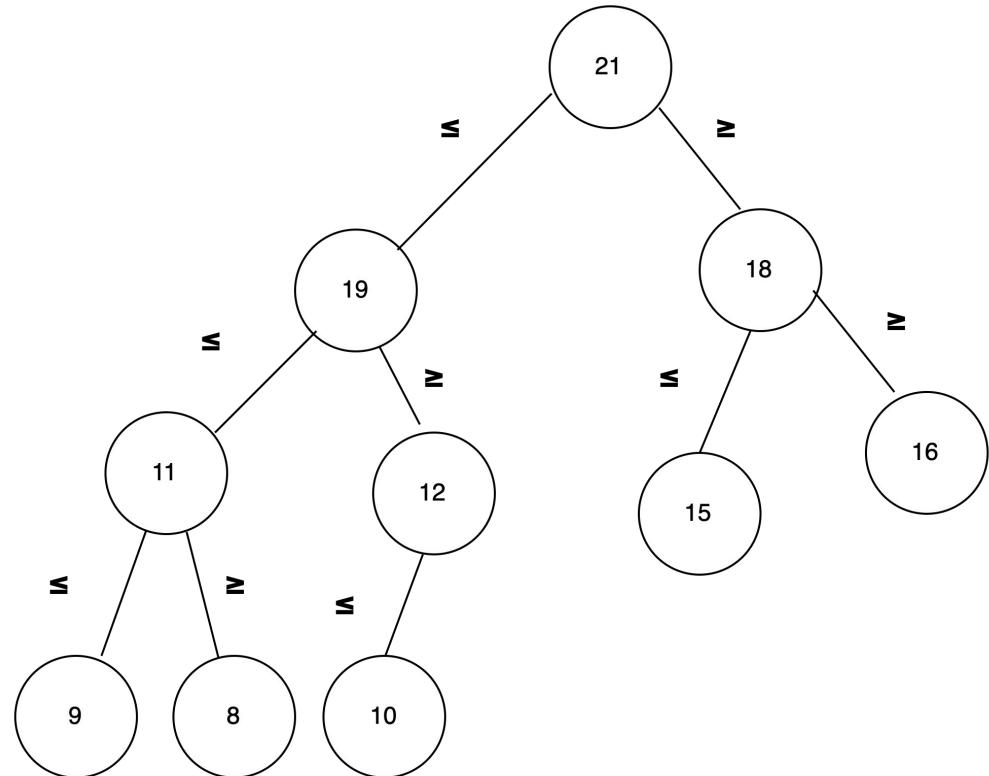
Двоичная куча (max-куча)

- Бинарное дерево
- Почти полное дерево. Все уровни, за исключением нижнего должны быть заполнены
- Все значения находящиеся ниже любого узла должны быть меньше него.
- Отсюда вывод: приоритет (значение) любого узла должен быть не меньше чем приоритет его потомков
- Нижний уровень всегда заполняется слева направо.
- Самый приоритетный элемент всегда хранится в корне



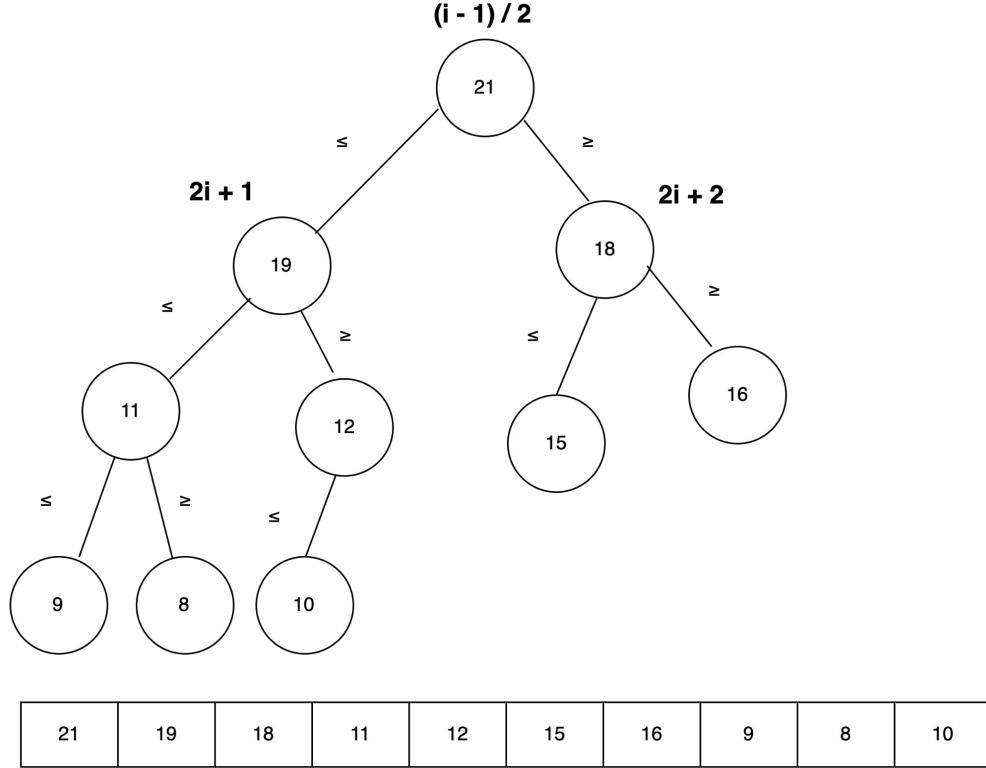
Двоичная куча

- Если мы возьмем узел 19, то все значения ниже него будут меньше 19
- Если сейчас мы решим добавить узел, то мы поставим его в качестве правого потомка узла 12
- Отсюда и название max-куча
- Это одна из возможных реализаций структуры данных куча



Представление кучи в массиве

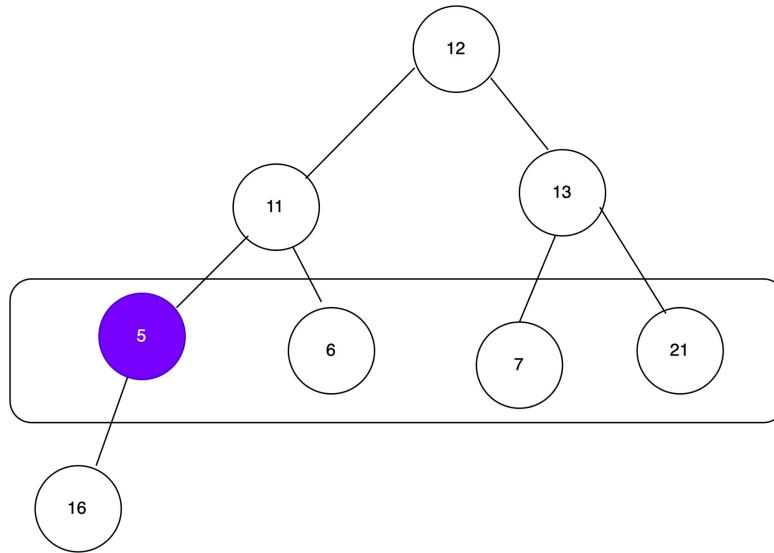
- Это по-прежнему бинарное дерево
- Это почти полное бинарное дерево, его представление в массиве будет немного проще, так как все уровни за исключением самого нижнего заполнены равномерно
- Корень: всегда находится под нулевым индексом
- Родительский узел: $(i - 1) / 2$
- Левый потомок: $2i + 1$
- Для узла 12 (индекс 4) его левый потомок, если он есть, будет находиться под индексом $2 \cdot 4 + 1 = 9$
- Правый потомок: $2i + 2$
- Для узла 11 (индекс 3) его правый потомок, если он есть, будет находиться под индексом $2 \cdot 3 + 2 = 8$



Создание кучи из массива

- [12, 11, 13, 5, 6, 7, 21, 16]
- Построим из нашего массива, по известным нам формулам бинарное дерево
- начинаем с узла 5 так как у него единственного есть потомки
- на первом вызове **siftDown** он поменяется местами с 16

```
def buildMaxHeap(data):
    n = len(data)
    # начинаем с последнего уровня,
    # который имеет потомков
    # идем от последнего узла до корня и
    # выполняем siftDown для каждого узла
    for i in range(n // 2 - 1, -1, -1):
        siftDown(data, n, i)
```



Восстановление свойств кучи

Функция **siftDown (heapify)** выполняет корректировку максимальной кучи (текущего поддерева) с корнем в узле i , чтобы удовлетворить свойство максимальной кучи после удаления или изменения элемента.

Параметры:

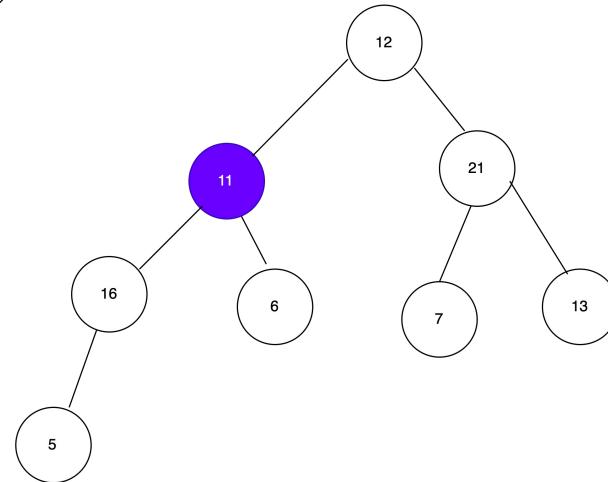
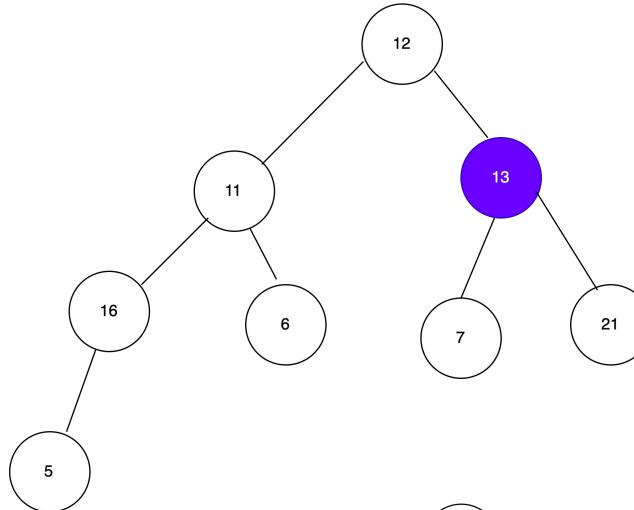
- data: массив, представляющий максимальную кучу
- n: размер кучи
- i: индекс узла, в котором нужно выполнить корректировку

```
def siftDown(data, n, i):  
    # запоминаем индекс наибольшего элемента  
    largest = i  
    # Индекс левого потомка в массиве  
    left = 2 * i + 1  
    # Индекс правого потомка в массиве  
    right = 2 * i + 2  
  
    # Если левый потомок существует  
    # и больше текущего элемента  
    if left < n and data[left] > data[i]:  
        # Обновляем индекс наибольшего элемента  
        largest = left  
  
    # Если правый потомок существует и больше  
    # текущего наибольшего элемента  
    if right < n and data[right] > data[largest]:  
        # Обновляем индекс наибольшего элемента  
        largest = right  
  
    if largest != i:  
        # Если после всех проверок  
        # индекс наибольшего элемента изменился  
        data[i], data[largest] = data[largest], data[i]  
        siftDown(data, n, largest)
```

Создание кучи из массива

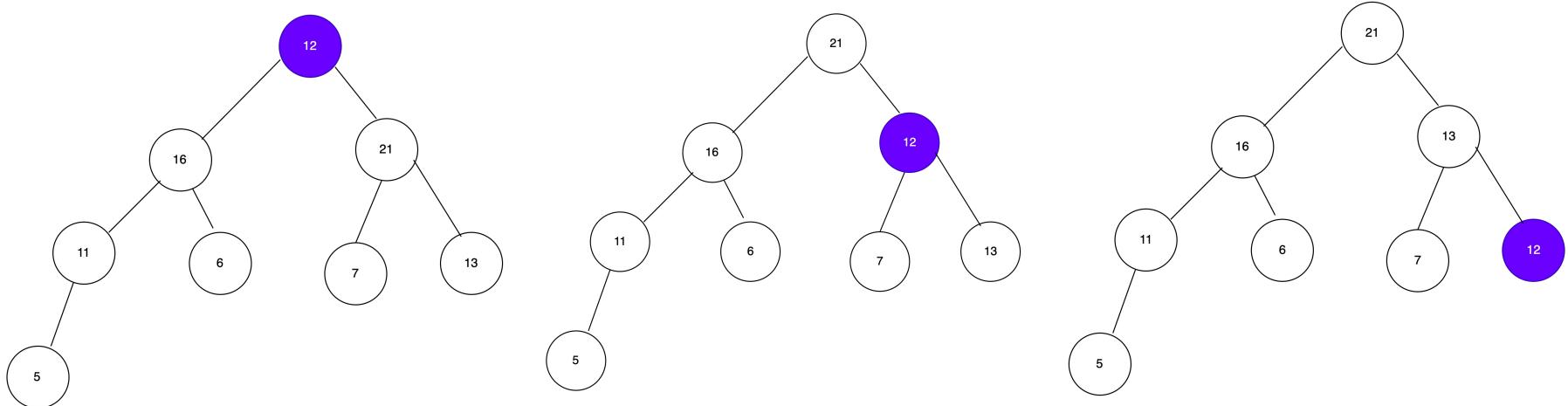
- [12, 11, 13, 5, 6, 7, 21, 16]
- далее вызываем **siftDown** для узла 13, 11
- доходим до корня 12 и в две итерации спускам его на место узла 13

```
def buildMaxHeap(data):
    n = len(data)
    # начинаем с последнего уровня,
    # который имеет потомков
    # идем от последнего узла до корня и
    # выполняем siftDown для каждого узла
    for i in range(n // 2 - 1, -1, -1):
        siftDown(data, n, i)
```



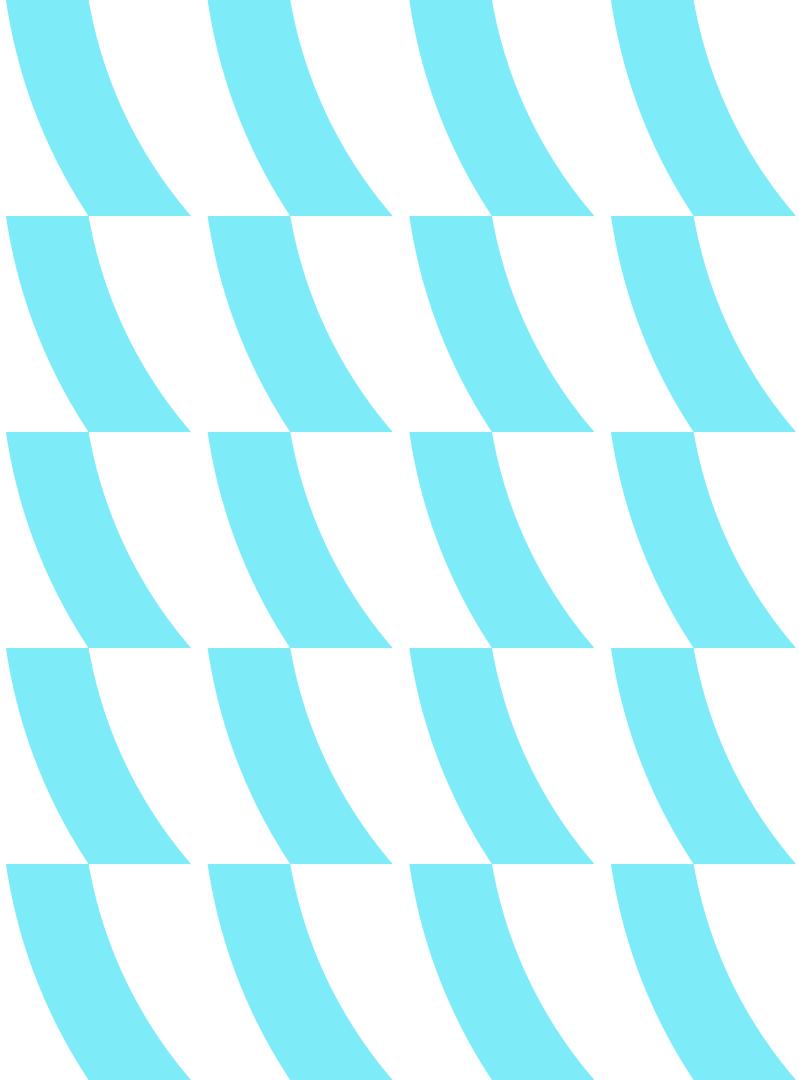
Создание кучи из массива

[12, 11, 13, 5, 6, 7, 21, 16]
[21, 16, 13, 11, 6, 7, 12, 5]



Сложность

- Листы не требуют просеивания, а значит просеивание надо начинать с последнего узла имеющего потомка
- Цикл, перебирающий узлы массива, выполняется $n/2$ раз, где n - количество элементов в массиве
- Узлы массива с индексами от $n/2$ до $n-1$ являются листовыми узлами
- Сложность для каждого узла в худшем случае $O(\log n)$
- Таким образом, общая времененная сложность алгоритма построения максимальной бинарной кучи составляет $O(n \log n)$
- Сложность по памяти $O(1)$



Операции над max-кучей

Очередь с приоритетом

- Операция получения значения **pop()**.
- Как следует из вышеперечисленных свойств, получение максимального значения осуществляется за $O(1)$
- Всегда возвращаем корень
- Операция добавления **push(data)**
- Восстановление свойств кучи - **heapify**

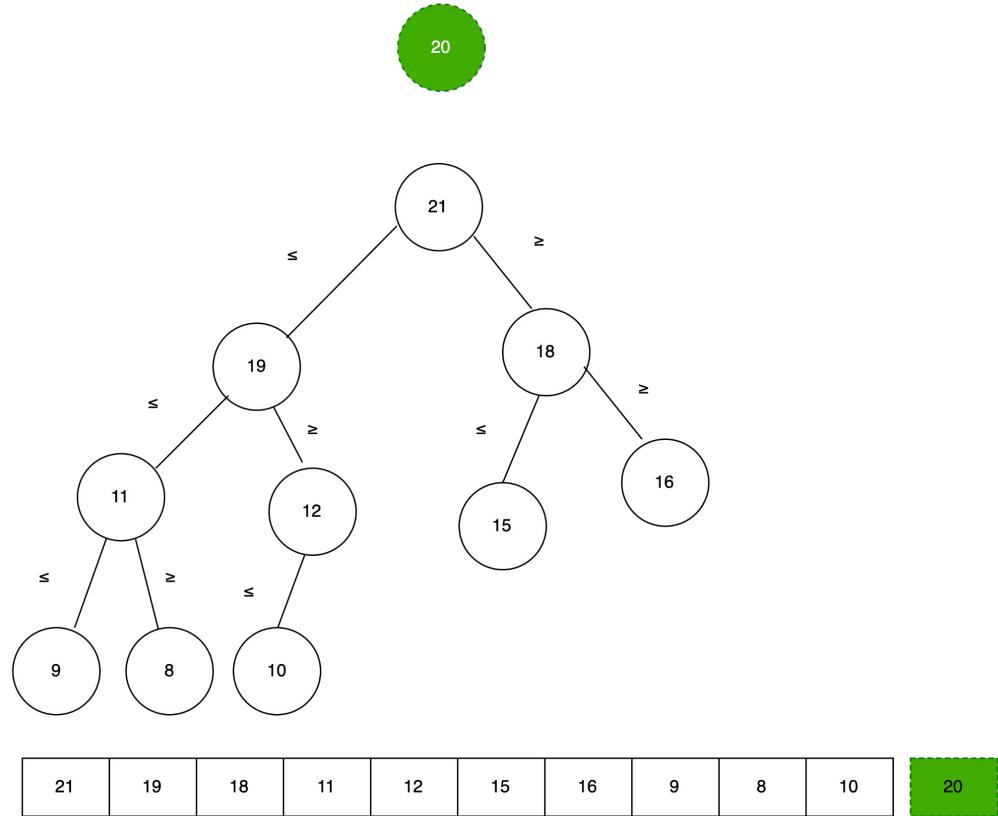
```
function parent(i) {  
    // находимся в корне  
    if i == 0 {  
        return i  
    }  
    return (i - 1)/2  
}
```

```
function pop() {  
    return heap[0]  
}
```

```
function left(i) {  
    return 2i + 1  
}  
  
function right(i) {  
    return 2i + 2  
}
```

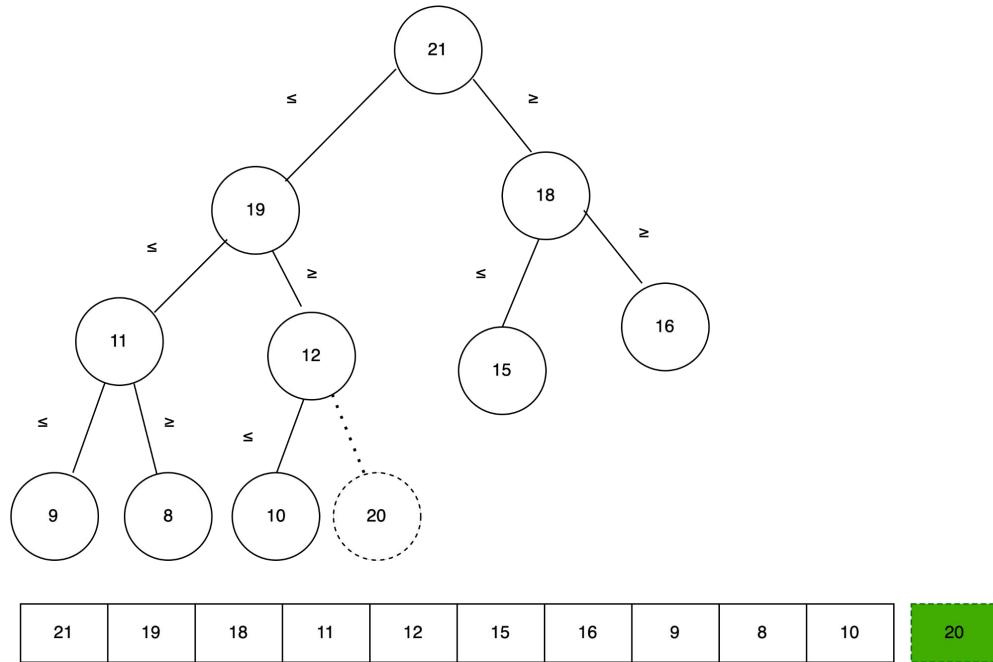
Добавление элемента

- Допустим, мы хотим вставить узел со значением 20
- Когда мы говорили про бинарное дерево поиска, мы осуществляли вставку, начиная с поиска подходящего места сверху вниз.
- В случае необходимости проводили балансировку
- Какой должен быть в данном случае алгоритм?
- Главное требование после любой операции над кучей: все уровни должны быть заполнены (кроме последнего) и в корне должен находиться максимальный элемент



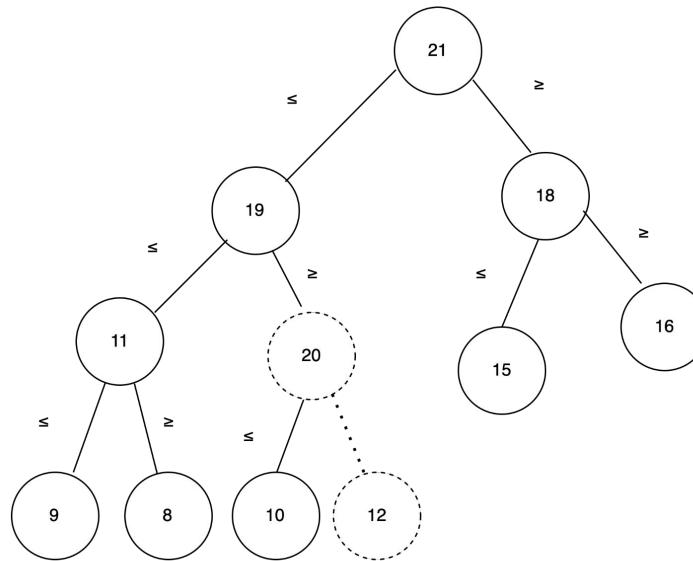
Добавление элемента

- Вставляем вниз, дорисовывая нижний уровень.
- То есть наш новый элемент должен оказаться следующим справа в нижнем ряду
- Экстраполируя это на массив - добавляем элемент в самый конец.
- В случае если свойства кучи нарушаются - поднимаем элемент выше, меняя его местами с его родителями



Добавление элемента

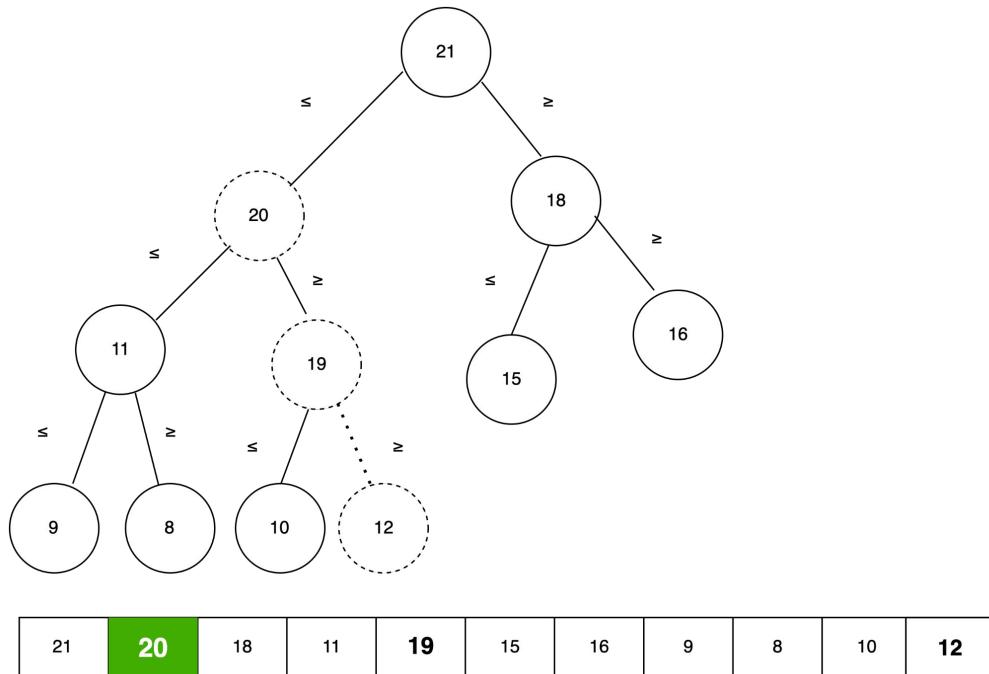
- Меняем местами с родительскими узлами до тех пор пока наш элемент не дойдет до узла, который больше него
- Какая сложность операции будет в данном случае?



21	19	18	11	20	15	16	9	8	10	12
----	----	----	----	----	----	----	---	---	----	----

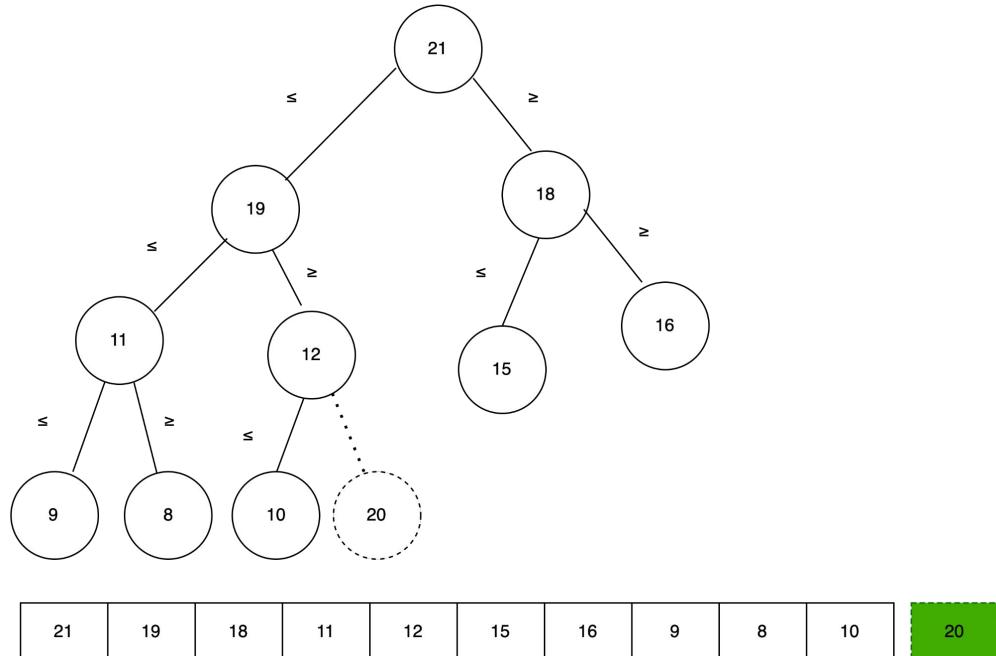
Добавление элемента

- Вставка и последующий подъем элемента в худшем случае требует количество перестановок равное высоте дерева
- Если бы мы, например, вставляли узел с значением 22, то мы прошлись бы по всей высоте кучи
- Откуда такая сложность: количество элементов n , так как у каждого элемента по 2 потомка получаем $n \sim 2^h$ отсюда $h = \log(n)$



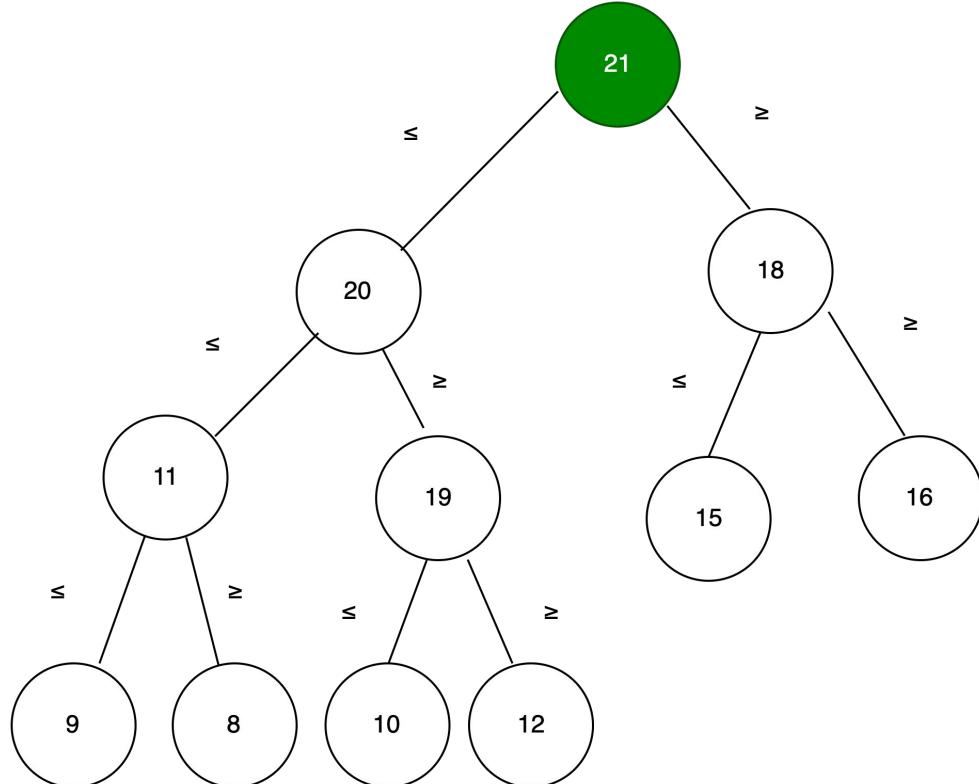
Добавление элемента

```
function push(key) {  
    // вставляем новый элемент  
    // в конец массива  
    heap.append(key)  
    // поднимаем его вверх  
    up(heap, heap.length(H.array) - 1)  
}  
  
function up(heap, index) {  
    // в цикле пока наш элемент больше  
    // родительского значения  
    // меняем их местами  
    // и смещаем индекс на родительский узел  
    while heap[index] > heap[parent(index)] {  
        swap(heap[index], heap[parent(index)])  
        index = parent(index)  
    }  
}
```



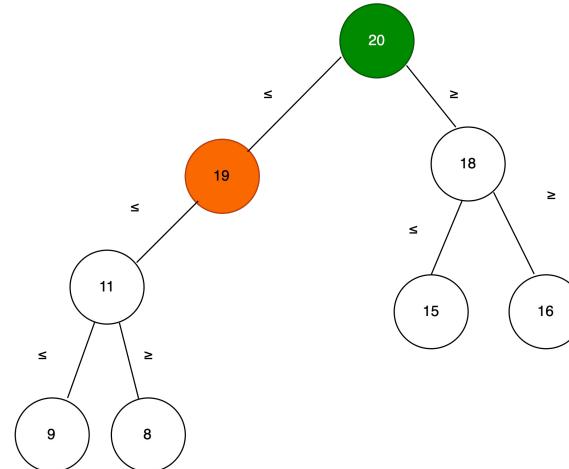
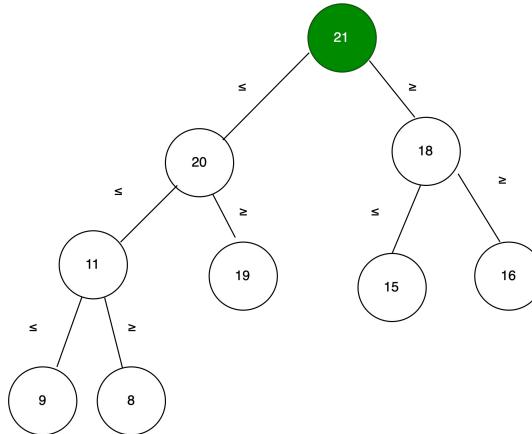
Удаление

- Операция `pop()` всегда возвращает элемент, находящийся на вершине нашей кучи
- Какой элемент должен оказаться теперь в корне?
- Как вариант можем выбрать максимальный из дочерних, то есть в данном случае это будет 20
- Но тогда после подъема всех узлов наверх, дерево может перестать быть сбалансированным



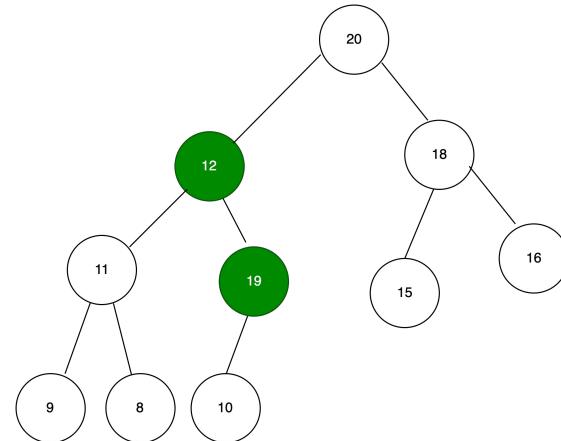
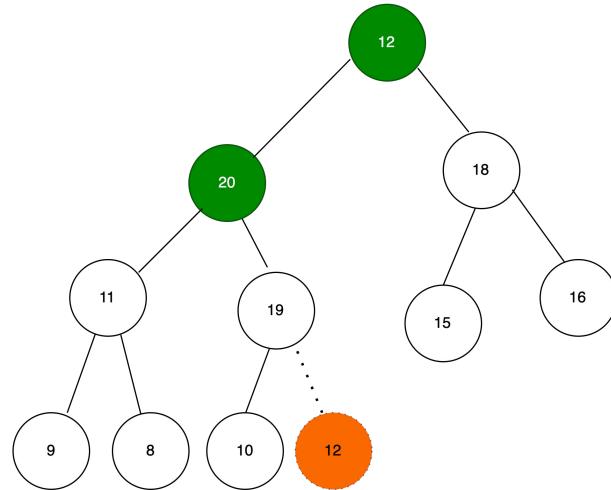
Удаление

- Например, если у 19 нет детей
- После операции `pop()` удаляем 21 и поднимаем 20 вверх
- Следом должен пойти узел 19
- И вот у нас уже не все уровни заполнены



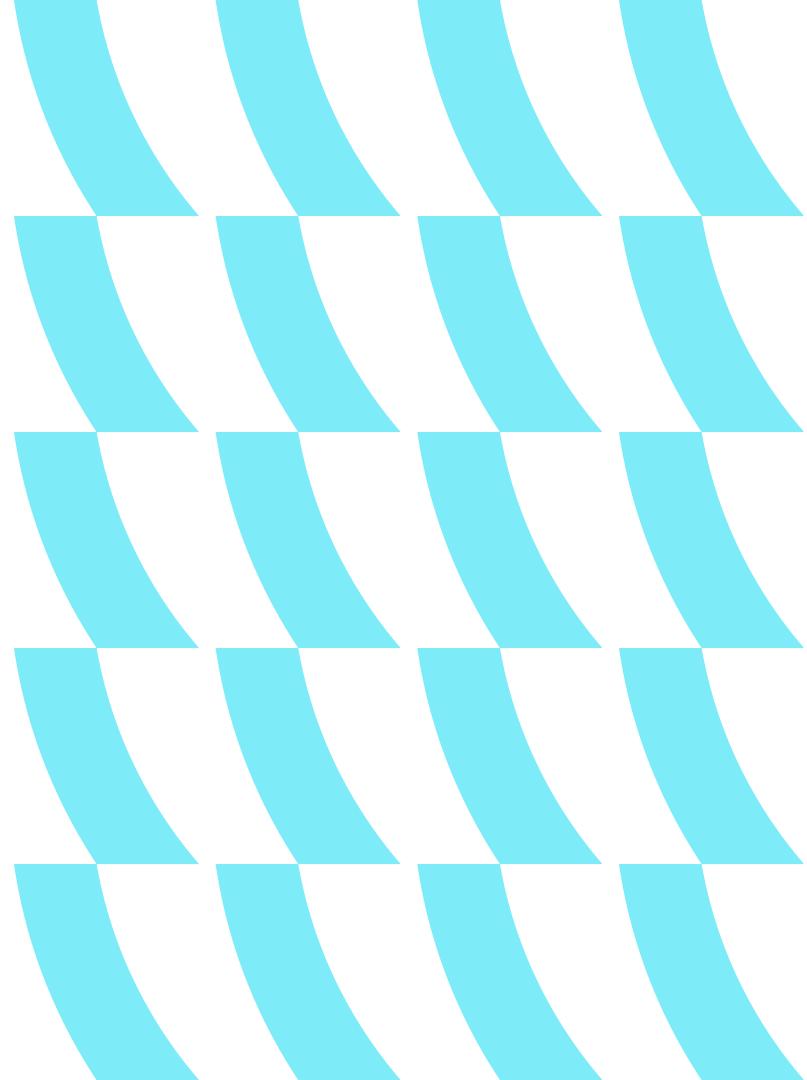
Удаление

- На место удаленного узла передвигаем последний, то есть из самого нижнего ряда справа
- Опускаем этот элемент вниз, меняя его местами с максимальным потомком
- Удаляем 21, на его место ставим 12, далее 12 опускаем вниз, меняя каждый раз с наибольшим значением из потомков
- Повторяем это пока есть куда опускать.
- $O(\log(n))$
- Восстановление свойств кучи при удалении будет немного отличаться от восстановления при вставке. Так как при вставке мы подымаем число, сравнивая его только с родителем, а при удалении мы опускаем его вниз, сравнивая с потомками.



Удаление

```
def deleteMax(data):  
    # Проверка, что куча не пустая  
    if not data:  
        return  
  
    n = len(data)  
    # Заменяем корневой элемент  
    # на последний элемент в куче  
    # самый правый на нижнем уровне  
    data[0] = data[n - 1]  
    # удаляем последний элемент из кучи  
    data.pop()  
    # Вызываем siftDown для корректировки  
    # кучи с корнем в 0-м индексе  
    # иногда это называют просеиванием  
    siftDown(data, n - 1, 0)
```



Пирамидальная сортировка

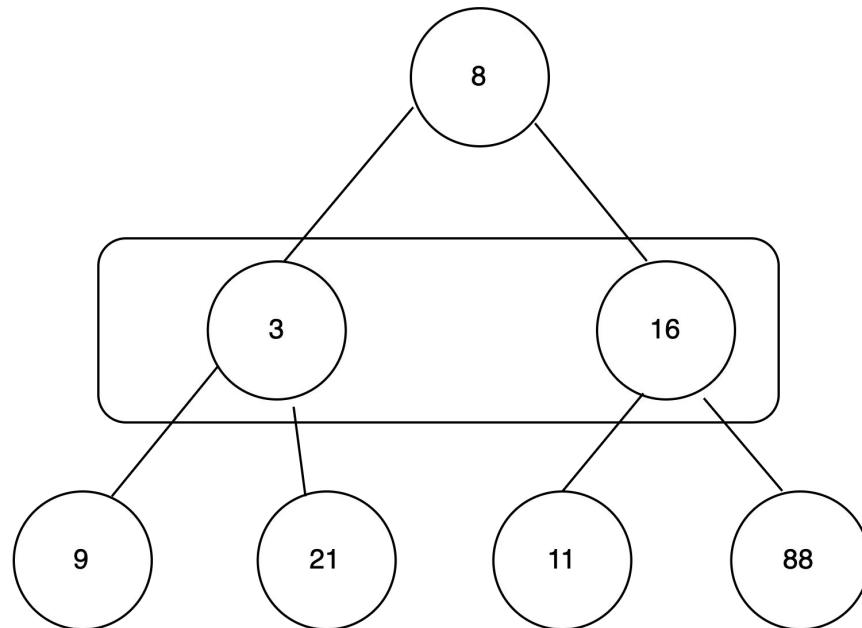
Heap sort

- Пирамидальная сортировка - это алгоритм сортировки, который использует "кучу" для хранения и упорядочивания элементов.
- Строим полное бинарное дерево из массива
- Преобразовываем получившееся дерево в кучу
- Начинаем проталкивать элементы снизу вверх

Heap sort

- Из массива, по известным нам формулам строим дерево
- Начинаем с последнего родительского уровня. То есть с уровня, узлы которого, в качестве потомков имеют только листья
- Сравниваем значения каждого родителя с его потомками
- Например, 16 сравниваем с 88 и 11
- Самое большое значение поднимается наверх.
- 88 меняем местами с 16

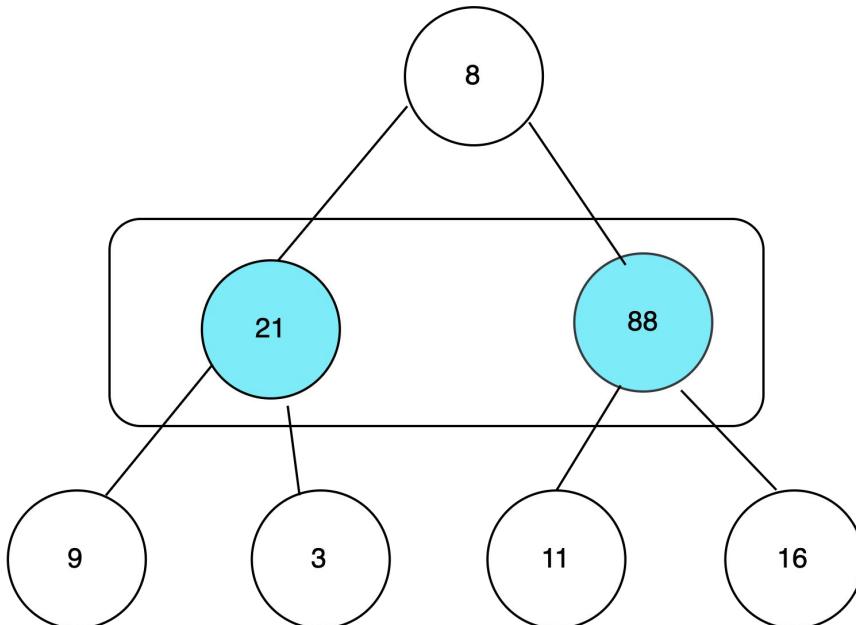
8	3	16	9	21	11	88
---	---	----	---	----	----	----



Heap sort

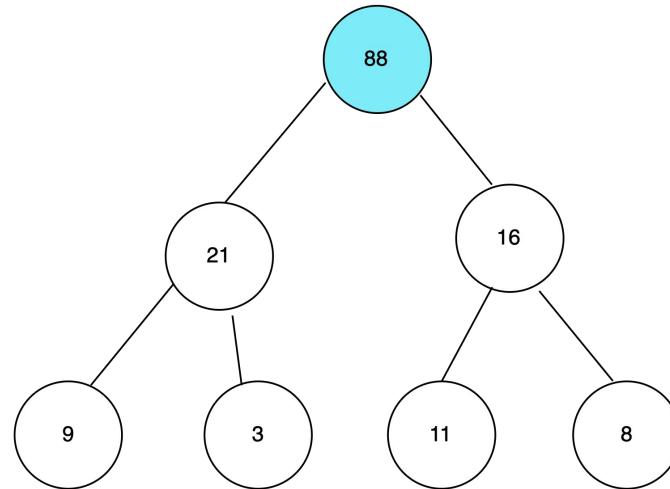
- Переходим к следующему родительскому узлу
- Поднимаем 21 вверх, меняя местами с узлом 3
- Далее будем сравнивать 8, 21 и 88
- Наверх пойдет 88

8	3	16	9	21	11	88
---	---	----	---	----	----	----



Heap sort

- В первой части алгоритма мы получили массив, которому еще далеко до сортированного
- Сейчас это бинарное дерево обладает свойствами кучи
- Теперь знаем, что его первый элемент самый большой
- Меняем первый и последний элементы местами. Теперь у нас в конце самый большой элемент
- Другими словами справа у нас будет отсортированная последовательность



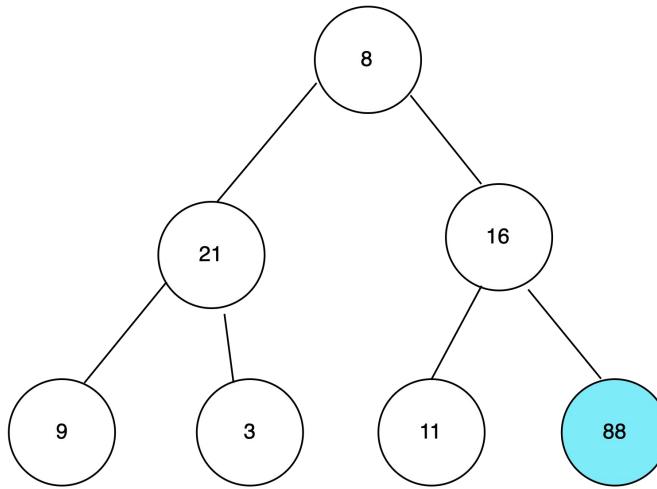
88	21	16	9	3	11	8
----	----	----	---	---	----	---

8	21	16	9	3	11	88
---	----	----	---	---	----	----

Heap sort

- Продолжаем алгоритм, но уже с учетом прошлой итерации
- Теперь у нас в вершине узел со значением 8, а наш массив сузился на один элемент
- Теперь нам надо спустить 8 вниз, как мы это делали, когда восстанавливали кучу после удаления элемента
- Меняем 8 местами с наибольшим потомком
- Теперь 21 должно находиться в вершине кучи

8	21	16	9	3	11	88
---	----	----	---	---	----	----

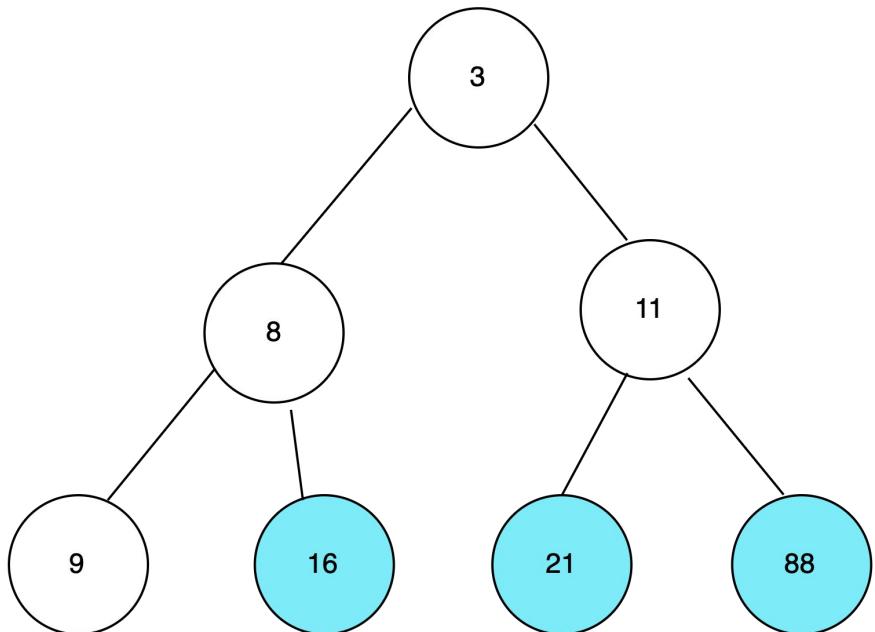


21	8	16	9	3	11	88
----	---	----	---	---	----	----

Heap sort

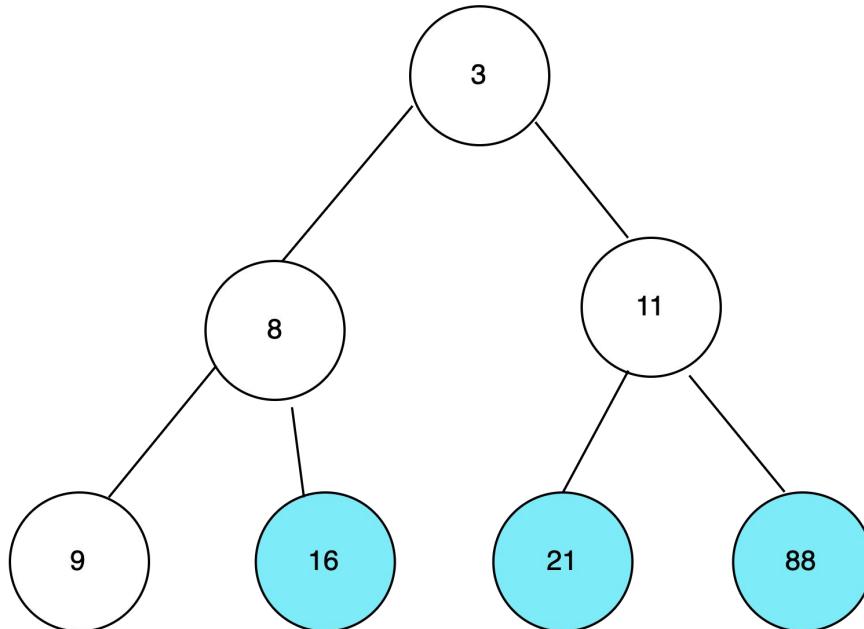
- Снова меняем вершину и последний из неотсортированной части элемент
- Получаем дерево следующего вида
- Продолжаем обмен и на этот раз меняем 11 и 16
- 16 становится корнем и впоследствии меняется с тройкой местами

3	8	11	9	16	21	88
---	---	----	---	----	----	----



Heap sort

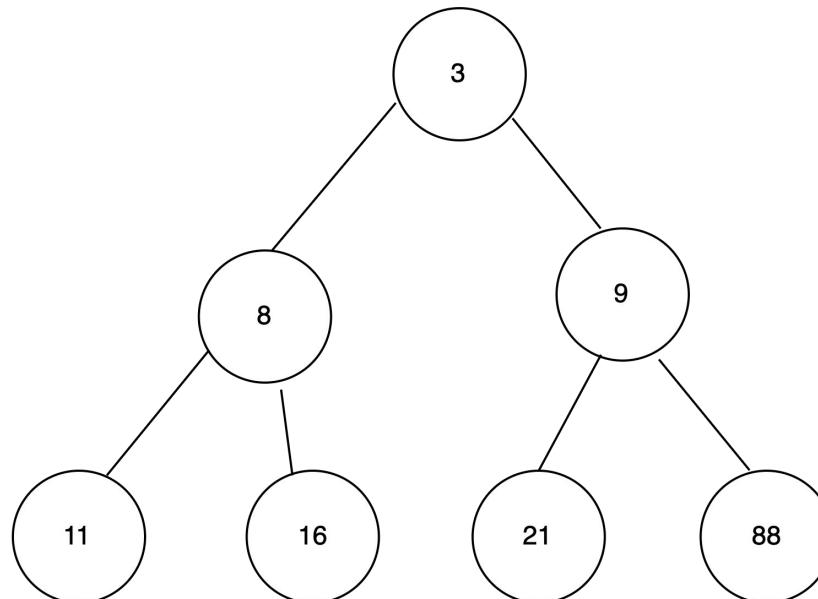
- Сейчас меняем местами 3 и 11
- Узел 11 уходит в начало отсортированной части, меняясь с 9
- Продолжаем алгоритм пока полностью не отсортируем массив



Heap sort

- В итоге наш массив стал отсортированным
- Структуру данных, к которой мы пришли можно назвать *min-кучей*.
- Каждый потомок меньше своего родителя
- В корне находится минимальный элемент

3	8	9	11	16	21	88
---	---	---	----	----	----	----



Операции над min-кучей

- Операция получения минимального значения **pop()** за $O(1)$
- Всегда возвращаем корень
- Операция добавления **push(data)**
- Восстановление свойств минимальной кучи происходит схожим образом.
- Сортировка min-кучей - на выходе массив отсортирован по убыванию

Heap sort

- Создаем бинарную кучу из массива
- Отправляем максимальный элемент, который лежит в вершине кучи, в самый конец массива
- Восстанавливаем свойства кучи для

```
# Пример использования
arr = [12, 11, 13, 5, 6, 7, 21, 16]
sorted_arr = heapSort(arr)
print(sorted_arr)
```

```
def heapSort(data):
    n = len(data)

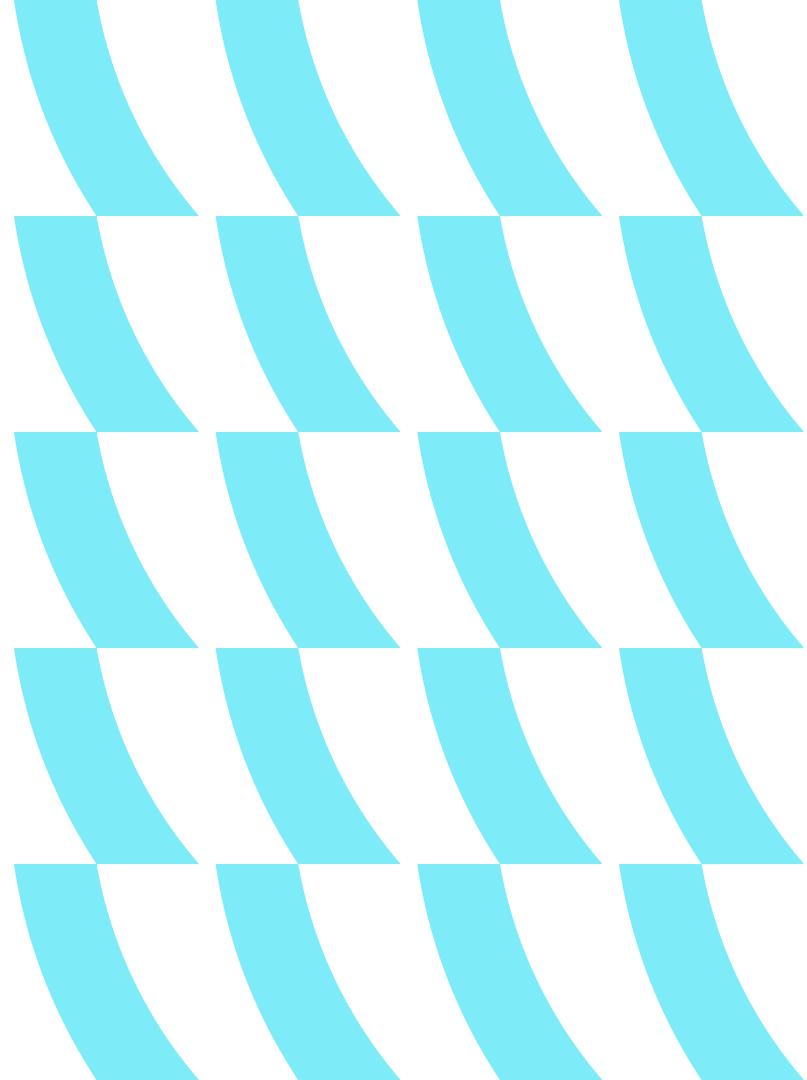
    for i in range(n // 2 - 1, -1, -1):
        siftDown(data, n, i)

    for i in range(n - 1, 0, -1):
        data[i], data[0] = data[0], data[i]
        siftDown(data, i, 0)

    return data
```

Сложность

- Создание создания бинарной кучи - $O(n \log n)$
- Для каждого элемента кучи мы выполнили количество перестановок равное высоте кучи
- $n \sim 2^h$ другими словами для каждого из n элементов мы выполнили $\log(n)$ действий
- Результирующая сложность как и у быстрых сортировок $O(n \log(n))$
- Эта сложность будет и в лучшем и в среднем и в худшем случае.



Красно-чёрное дерево



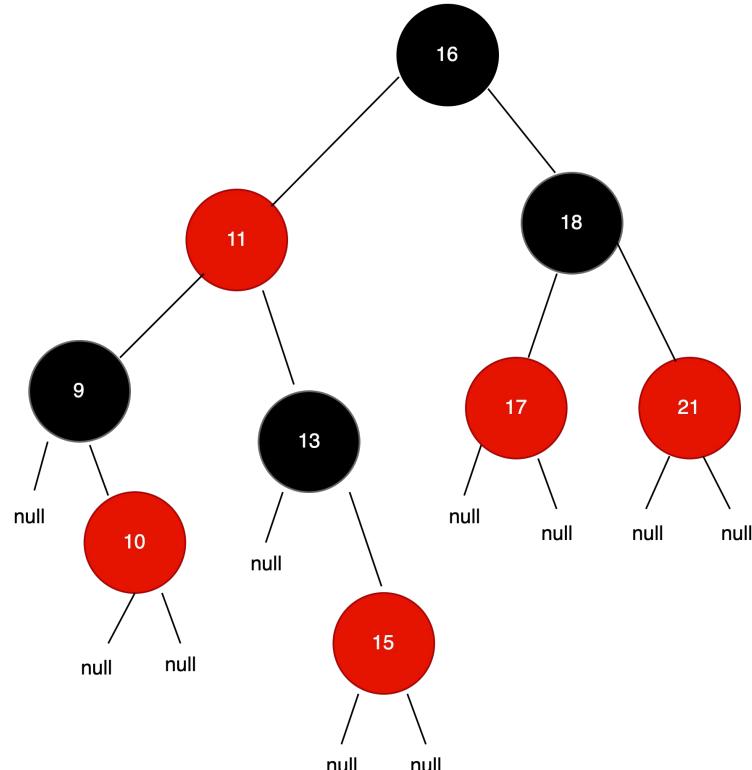
История

- Автор - немец Рудольф Байер изначально назвал его «symmetric binary B-tree» (B - не от *binary* а от *balanced*)
- Именно это название нам даст определенное понимание того что там под капотом
- И лишь позже в одной из научных работ это дерево назвали красно-черным.
- Цветовая гамма не имеет под собой никакой физической нагрузки



Что ты такое

- Это бинарное дерево и на него распространяются все соответствующие правила и ограничения
- Это так называемое самобалансирующиеся дерево
- На всякий случай проговорим, что это значит: дерево называется сбалансированным, или полным, если длины всех путей от корня к внешним вершинам равны между собой. Дерево называется почти сбалансированным, если длины всевозможных путей от корня к внешним вершинам отличаются не более, чем на единицу.
- Каждому узлу дается цвет - либо черный либо красный. Но не рандомно, а по определенным правилам.
- И именно это дает возможность дереву достигать сбалансированности, так как алгоритм «покраски», определение узла как черный или красный, при выполнении основных операций гарантирует логарифмический рост высоты дерева от количества узлов. А следовательно логарифмическую сложность.
- Иными словами цвет используется для балансировки узлов.
- Подобного рода гарантии баланса дают возможность производить быстро операции вставки, поиска и удаления узла



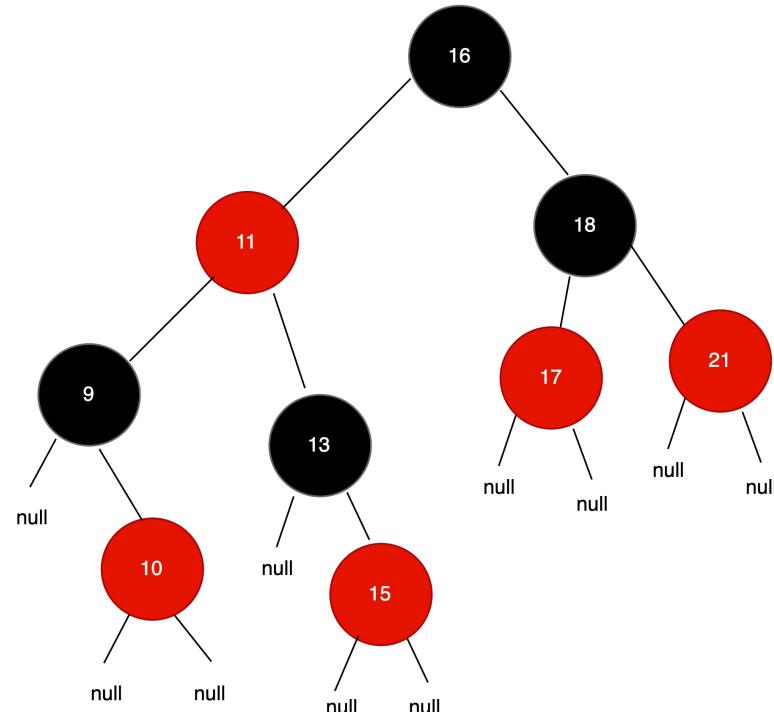
Снова про сложность

- Раз мы уже знаем, что дерево сбалансировано, то уже мы можем поговорить о сложности
- Подобного рода вывод можно посмотреть в предыдущей лекции

Операция	Средний случай	Худший случай
insert	$\log(n)$	$\log(n)$
search	$\log(n)$	$\log(n)$
delete	$\log(n)$	$\log(n)$
min	$\log(n)$	$\log(n)$
max	$\log(n)$	$\log(n)$

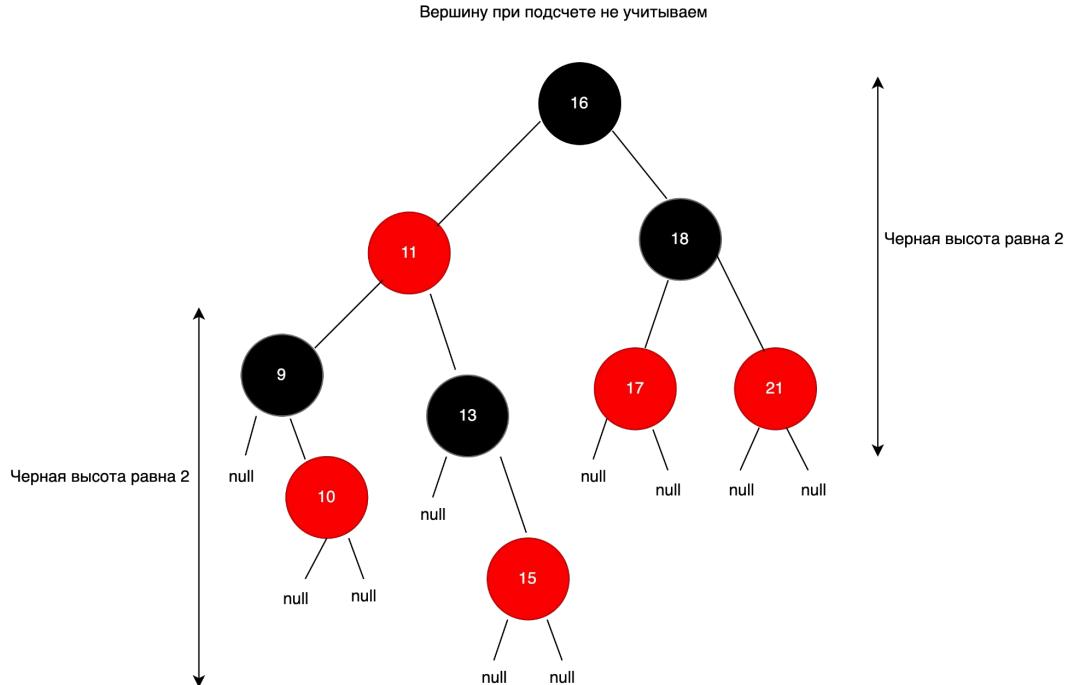
Основные правила КЧ дерева

- Это, как я уже сказал бинарное дерево. Но каждому узлу добавляется атрибут в виде цвета.
- Сформулируем 4 основных правила, которым будем следовать в дальнейшем при балансировке.
- 1. Каждый узел имеет цвет: красный или черный
- 2. Корень и листья, то есть узлы которые не имеют потомков - всегда черные
- 3. Красный узел не может иметь красного родителя, иными словами не может быть два красного узла подряд
- 4. Все пути от любого узла до листа содержит одинаковое количество черных узлов. Это так называемая черная высота.
- Пару слов про листья: как видно из рисунка, каждый конечный узел имеет значение null. Мы их специально добавляем. Более того они имеют цвет - они всегда черные.



Черная высота

- Корень не учитывается при подсчете черной высоты
- Считаем все черные узлы и листья
- Она всегда должна быть одинаковой.
- Если она равна от корня, то она будет равна от каждого узла



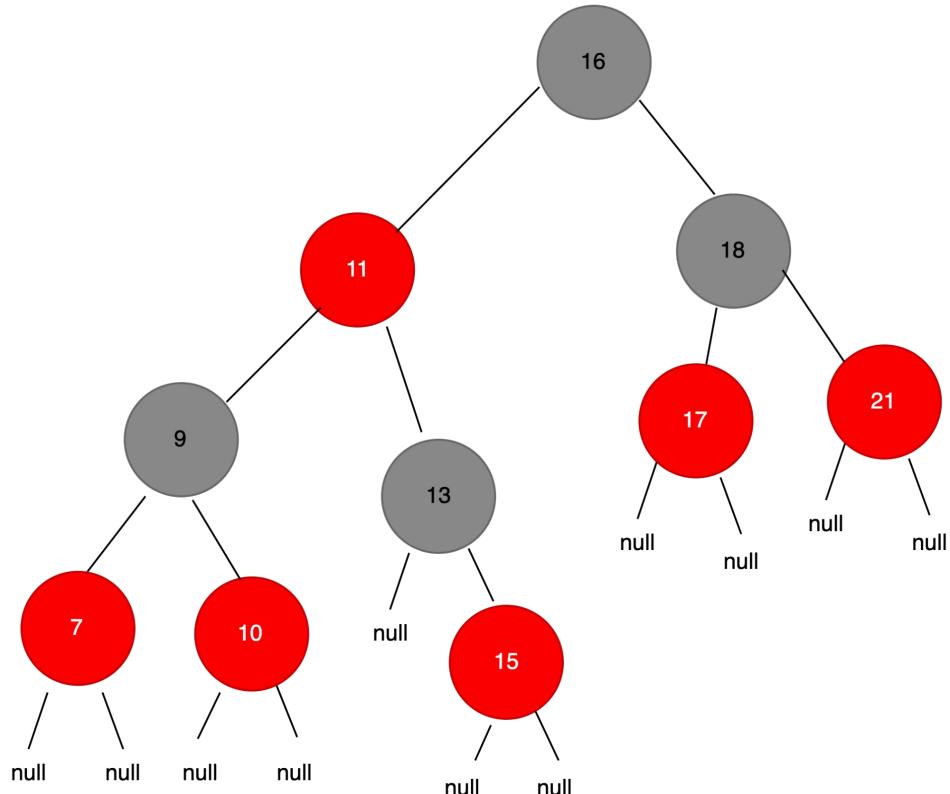
Структура дерева в коде

- Структура очень схожа с уже известным нам бинарным деревом поиска
- Добавлен атрибут color и листья по умолчанию.
- В качестве оптимизации можно всегда ссылаться на один и тот листок

```
class RedBlackTree:  
    class Node:  
        def __init__(self, data):  
            self.data = data  
            # у каждого узла есть  
            # по умолчанию левый и  
            # правый потомки  
            self.left = None  
            self.right = None  
            self.parent = None  
            # Изначально все узлы красные  
            # 1 – красный  
            # 0 – черный  
            self.color = "RED"
```

Вставка

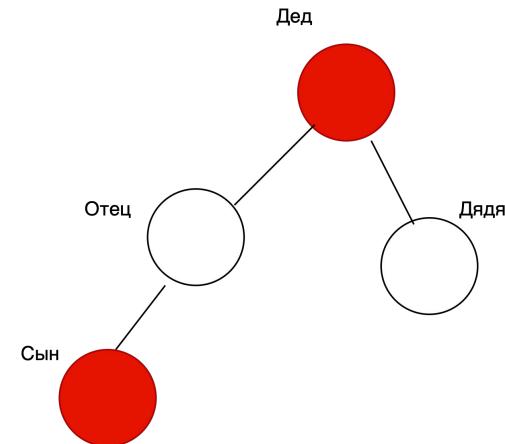
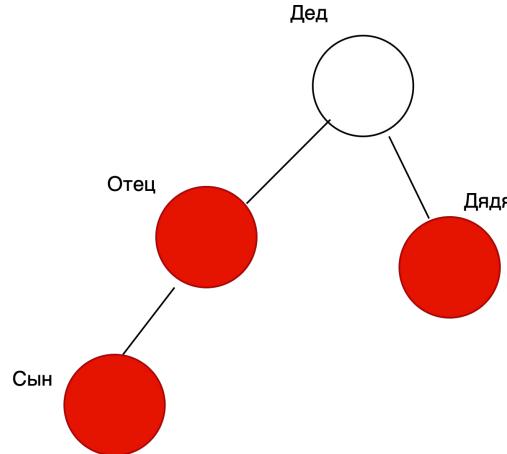
- Так же как и раньше мы спускаемся вниз в поисках подходящего места для нашего нового узла
- Ищем пока не дойдем до NULL то есть до листа дерева
- Вставляем вместо этого листа новый узел. Добавляем ему два NULL листа.
- Новый элемент всегда окрашивается в красный цвет
- В примере ниже у нас самый тривиальный кейс - вставляем семерку
- В нашем случае у 7 отец черный, поэтому тут проблем нет. Как быть если отец красный?
- Тут будем рассматривать следующие случаи.



Вставка

Вариант с рекурсивным подъемом

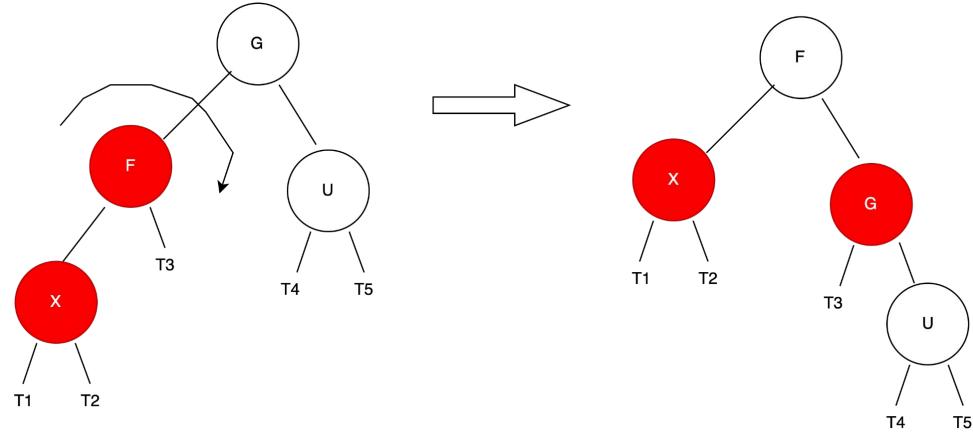
- Перекрашиваем отца и дядю в черный цвет
- Если отец или дядя были красными, то дед был черным. Так как два подряд красных узла быть не может, то дед в нашей ситуации в любом случае оказался бы черным.
- Зачем красить дядю? Чтобы не нарушать 4-ое правило, которое гарантирует нам равенство черных высот. Перекрашиваем деда в красный по той же причине.
- Если после такой перекраски дед стал нарушать свойства дерева, в свою очередь уже из-за своего отца, то есть прадеда, который может оказаться красным, то идем вверх рекурсивно, пока не сможем соблюсти все свойства. Узел за узлом восстанавливая необходимые свойства.
- Дойдя до корня красим его в любом случае в черный цвет. Корень, как мы знаем не влияет на черную высоту.



Вставка

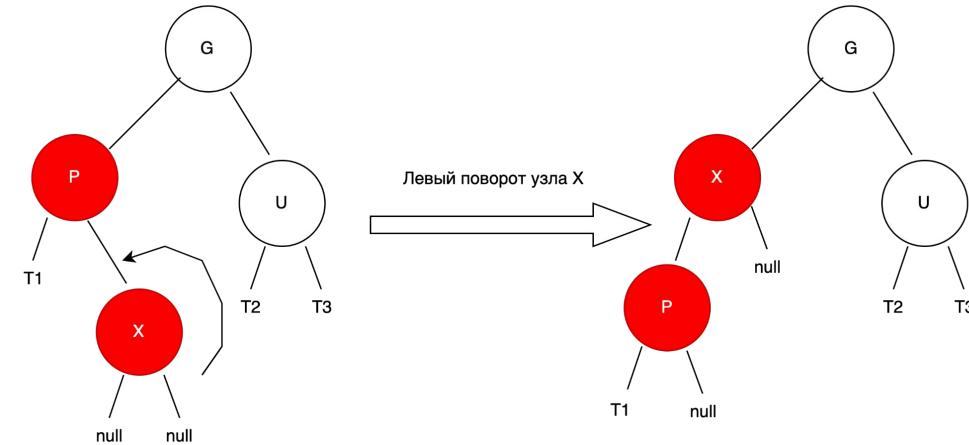
Вариант с поворотом

- Разберем случай посложнее - дядя черный, он находится справа. Так как мы сейчас будем вращать дерево это важно.
- В данном случае X необязательно вставляемый элемент, это может быть ситуация, когда мы балансируем дерево после вставки и подымаемся рекурсивно вверх. Возможно это чей-то дед. Как например в предыдущей ситуации.
- Для полноты картины, так как у нашего элемента могут быть потомки, в схеме добавлены T_i
- Просто перекрасить тут уже не получится - нарушится правило черной высоты.
- Тут нам потребуется **правый поворот** относительно отца (вращаем деда вправо).
- После поворота видно, что количество черных узлов от F, который теперь стал корнем поддерева равно первоначальному.
- Обратите внимание, что T_3 перекинулось от отца к деду, чтобы сохранить уже свойства бинарного дерева поиска.
- Количество черных узлов от T_i до любой вершины не изменилось



Вариант с двумя поворотами

- X - правый потомок родительского узла P, а дядя черный
- Выполняем левый поворот узла X, а дальше приходим ко 2 варианту с одним поворотом
- В случае, если дядя левый а X правый потомок - ситуация будет симметричной, а поворот должен быть соответственно левым.
- Запоминаем: пришли из левого узла - делаем правый поворот. Из правого узла - левый.



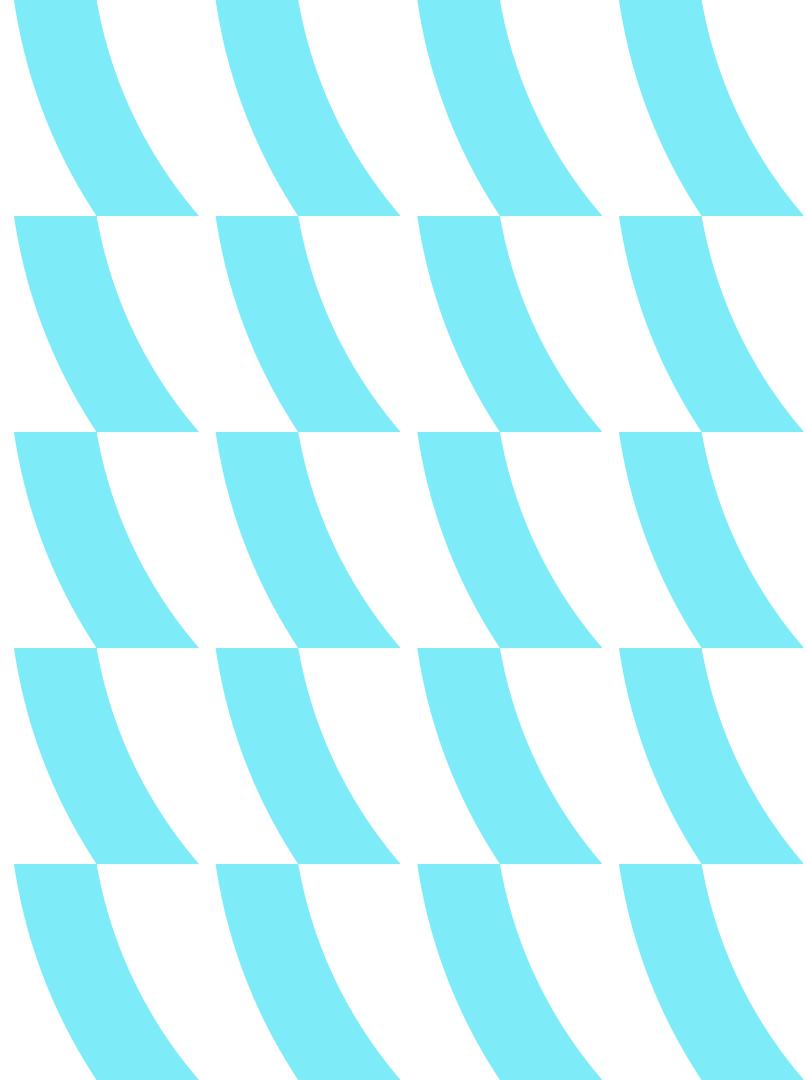
Какие могут быть ситуации

- Кейс с рекурсивным подъемом может выполняться как один раз, так и до тех пор пока не дошли до корня или черного родителя.
- В результате рекурсивного подъема мы можем дойти до случая когда нам понадобится правый/левый поворот. На этом вставка закончится.
- То есть первый случай может перейти во второй, второй же не перейдет в первый.
- Самый худший вариант это когда нам придется дойти до корня, но и тогда сложность будет $O(\log(n))$

```
def insert(self, data):  
    node = self.Node(data)  
    # Вставка элемента как в обычное  
    # бинарное дерево поиска  
    self._insert_node(data)  
  
    # Проверка и восстановление  
    # свойств красно-черного дерева  
    self._fix_insert(node)  
    :  
    :
```

Преимущества

Красно-черные деревья имеют особенности, позволяющие эффективно выполнять операции вставки и удаления элементов без нарушения баланса. В бинарных деревьях поиска операции вставки и удаления могут потребовать перебалансировки всего дерева, что может быть менее эффективным.



Восстановление свойств КЧ дерева

```
def _fix_insert(self, node):
    """
    Восстановление свойств красно-черного дерева после вставки
    """

    while node.parent and node.parent.color == "RED":
        if node.parent == node.parent.parent.left:
            # Родительский узел является левым потомком своего родителя
            uncle = node.parent.parent.right

            if uncle and uncle.color == "RED":
                # Дядя также красный - перекрашиваем
                # родителя, дадо и дедушку
                node.parent.color = "BLACK"
                uncle.color = "BLACK"
                node.parent.parent.color = "RED"
                # Переходим к дедушке
                node = node.parent.parent
            else:
                if node == node.parent.right:
                    # Узел является правым потомком
                    # своего родителя - делаем левый поворот
                    node = node.parent
                    self._left_rotate(node)

                    # Изменяем цвета и делаем правый поворот
                    node.parent.color = "BLACK"
                    node.parent.parent.color = "RED"
                    self._right_rotate(node.parent.parent)
                else:
                    # Родительский узел является правым потомком
                    # своего родителя (аналогично случаю выше)
                    uncle = node.parent.parent.left

                    if uncle and uncle.color == "RED":
                        node.parent.color = "BLACK"
                        uncle.color = "BLACK"
                        node.parent.parent.color = "RED"
                        node = node.parent.parent
                    else:
                        if node == node.parent.left:
                            node = node.parent
                            self._right_rotate(node)

                        node.parent.color = "BLACK"
                        node.parent.parent.color = "RED"
                        self._left_rotate(node.parent.parent)

        self.root.color = "BLACK"
```

Всем спасибо

и хорошего вечера:)

