Деревья поиска

education

Дерево

1

Иерархическая структура данных

2

В отличие от уже нам известных массивов и списков, дерево — нелинейная структура

3

Состоит из нод (узлов)

4

Узел — хранилище для данных и указателей на следующие узлы (потомки)

5

Лист — узел, у которого нет потомков

6

Корневой узел — вершина дерева

7

Всегда может быть только один родитель и множество дочерних нод

8

Высота дерева — расстояние от корня до самого нижнего элемента

Где используются

1

set и map в C++
в java TreeMap
и TreeSet
в Lisp-списках

2

В качестве индекса
в базах данных

3

Управление
виртуальной памятью
в ядре (хранение
адресов и т.д.)

4

Структура папок
как пример
иерархической
структуры

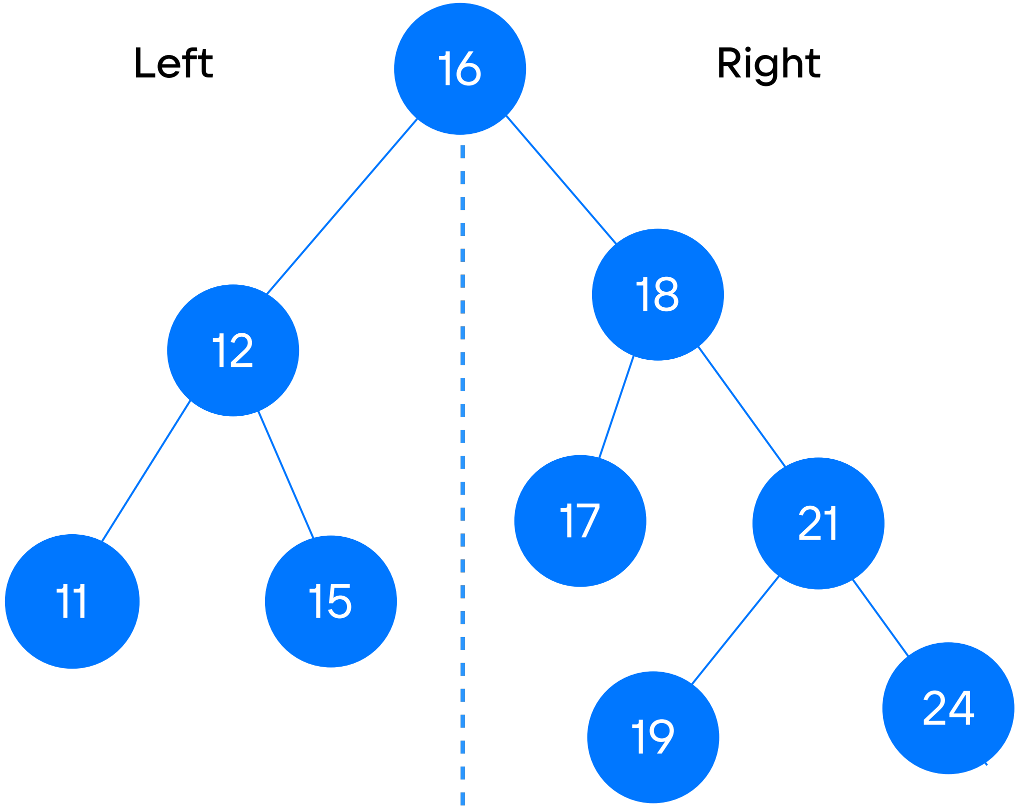
Бинарное дерево

- Разновидность деревьев.
- У каждого узла не может быть больше двух потомков (левый и правый).
- Меньше проверок при рекурсивном обходе 0 < n < 2.

```
Node {
        d: int
        l: *Node
        r: *Node
}
```

Бинарное дерево поиска

- Максимум два потомка.
- У каждой вершины в левом поддереве все элементы не больше, чем в самой вершине и в правом поддереве.
- Поддеревья каждого узла тоже являются бинарными деревьями поиска.

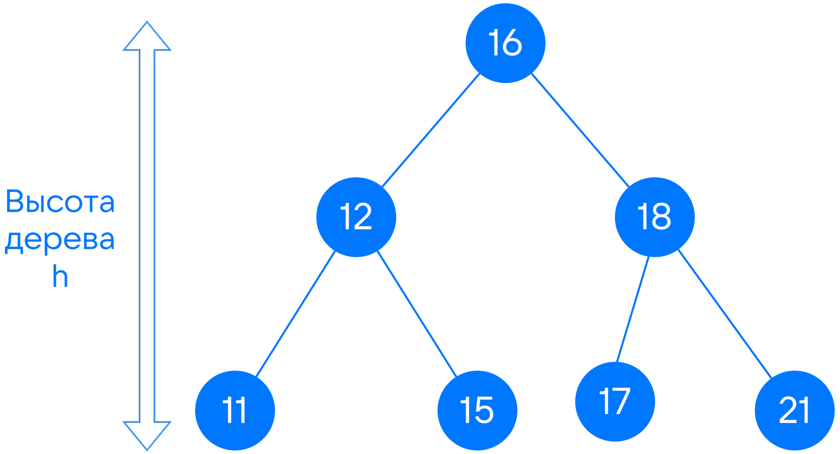


Сложность

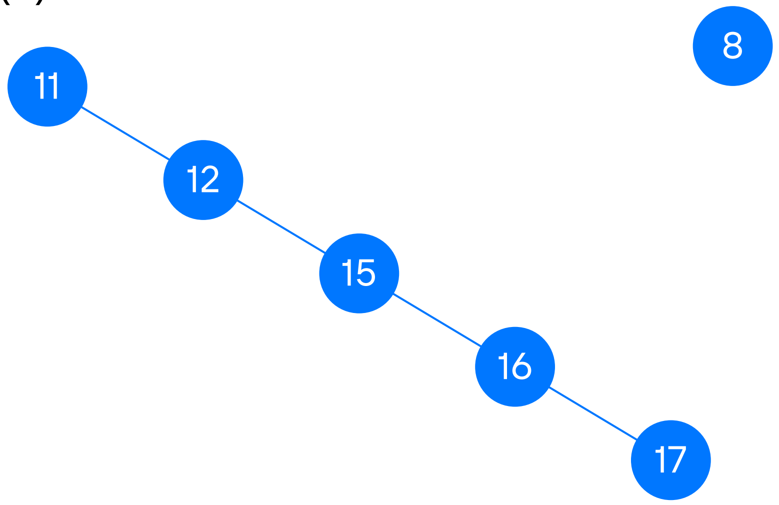
- В идеале у сбалансированного дерева, у каждого узла, кроме листовых потомков, ровно два дочерних, как на первом рисунке.
- Подсчитаем количество узлов n.
  n = 1 + 2 + 2^2 + 2^4 = 2^(h+1) – 1.

- Прологарифмируем результат:
  h = log(n+1) – 1.
- Сложность операций у нас всегда O(h).

O(log(n))



O(n)



Поиск минимума и максимума

- Из свойств бинарного дерева поиска следует, что минимум всегда внизу левого поддерева.
- Максимум всегда внизу правого поддерева.

```
min(root) {
 if root.left == null
   return root
 return min(root.left)
}

max(root) {
 if root.right == null
   return root
 return max(root.left)
}
```

Выборка элемента

- В «сбалансированном» дереве — O(log n).
- В худшем случае — O(n).

```
search(root, needle) {
   if root == null {
      return null
   }
   if needle == root.data {
      return root
   }
   if needle < root.data {
      return search(root.left)
   }
   if needle > root.data {
      return search(root.right)
   }
}
```

Вставка элемента

- Алгоритм схож с поиском.
- При нахождении null-элемента мы на его место вставляем свой.
- Сложность вставки также зависит от высоты дерева.

```
insert(root, data) {
   if root == NULL {
      return Node(data)
   }
   if (data < root.data) {
      root.left  = insert(root.left, data)
   }
   if (root > root.data) {
      root.right = insert(root.right, data)
   }

   return root;
}
```

Удаление

1. Удаление листа — удаление у родителя соответствующего указателя

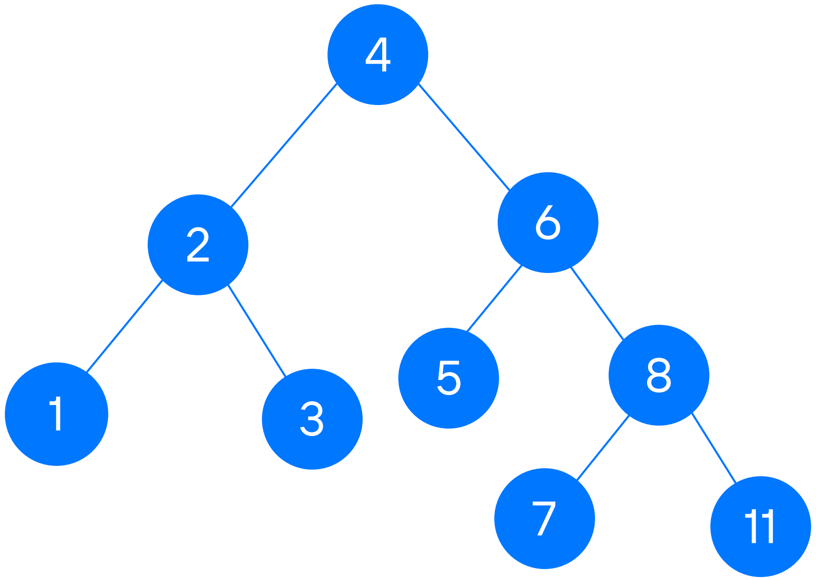
2. Удаление узла с одним дочерним элементом

3. Удаление узла с двумя дочерними узлами

```
def delete(self, node, key):
  # ищем узел по переданному значению
  if node is None:
     return node
  # если значение меньше текущего узла - идем в левое поддерево
  elif key < node.data:
     node.left = self.delete(node.left, key)
  # если значение больше - идем в правое поддерево
  elif key > node.data:
     node.right = self.delete(node.right, key)
  else:
     # если ключ нашли - начинаем процедуру удаления

     # 1. узел является листом
     if node.left is None and node.right is None:
        node = None

     # 2. узел имеет одного ребенка
     elif node.left is None:
        node = node.right

     elif node.right is None:
        node = node.left

     # 3. узел имеет двух детей
     else:
        min_val = minimum(node.right)  # ищем минимальное значение в правом поддереве
        node.data = min_val.data  # записываем вместо удаляемого значение найденное
минимальное
        node.right = self.delete(node.right, min_val.data)  # удаляем минимальное с его прежней
позиции

  return node
```

```
def minimum(node):
   while node.left is not None:
      node = node.left
   return node
```

```
def maximum(node):
   while node.right is not None:
      node = node.right
   return node
```



Обход дерева

1

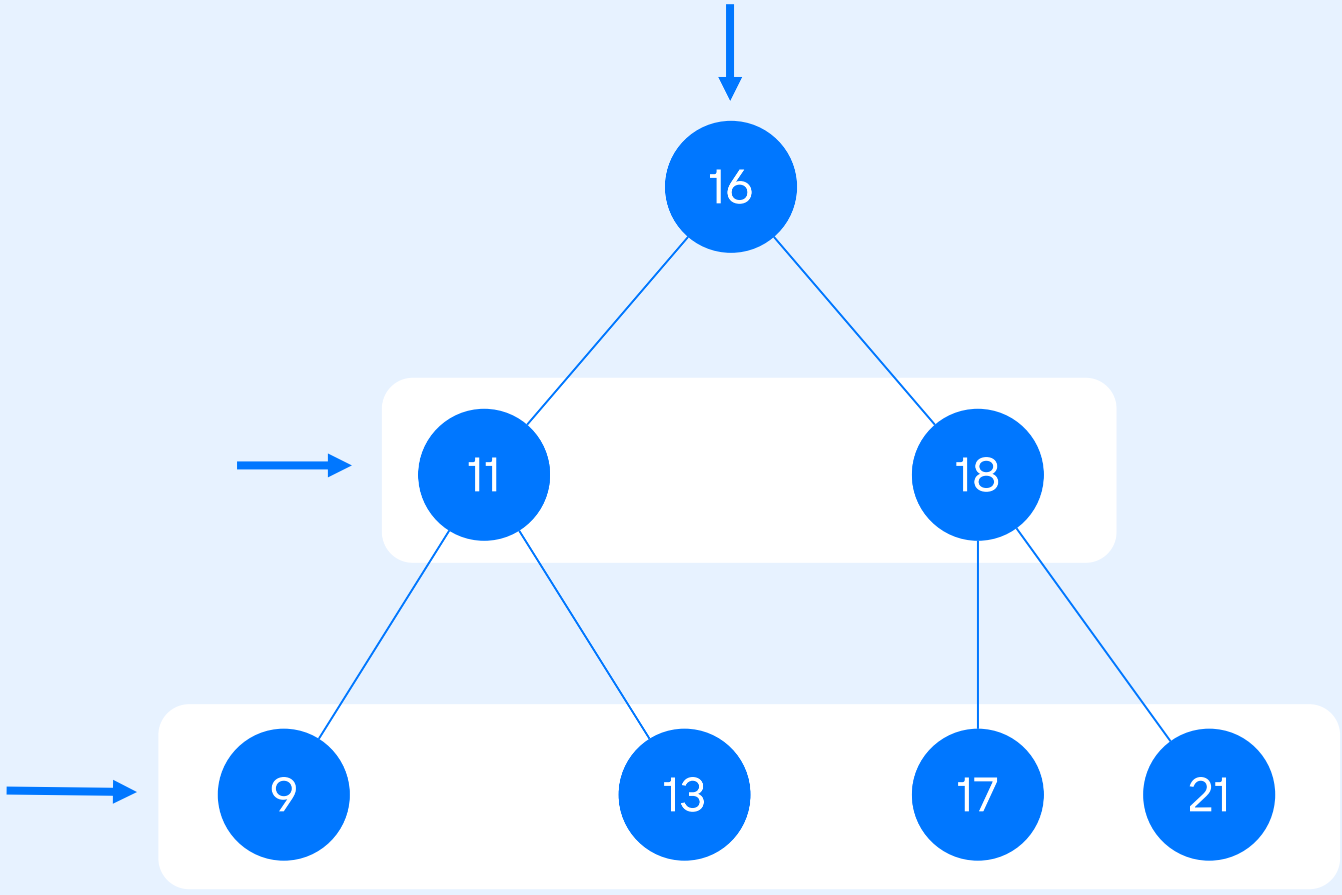
Обход в ширину

2

Обход в глубину

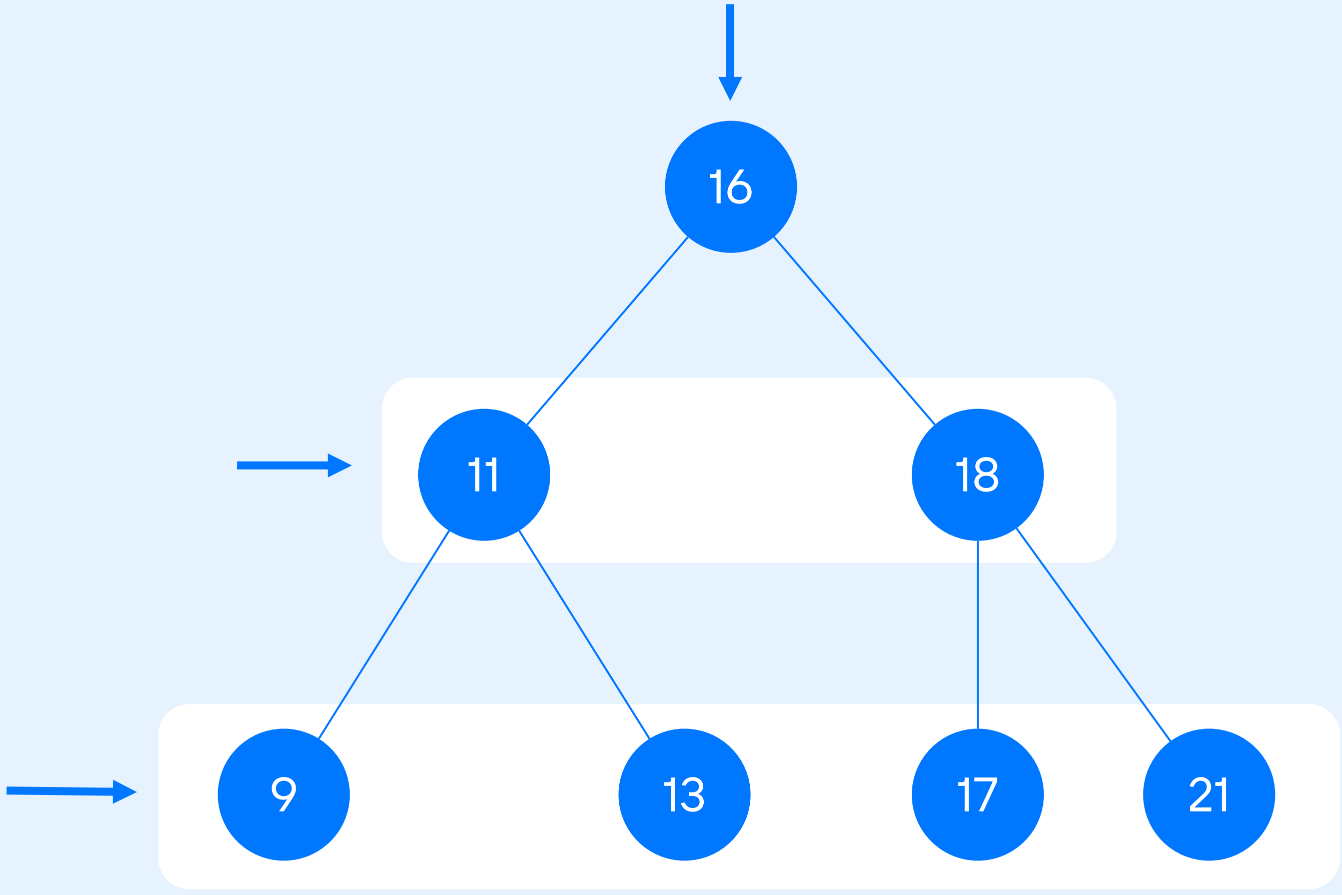
Обход в ширину (BFT)

- Последовательно проходим все вершины, начиная с корневого.
- Сверху вниз, слева направо.
- Вначале мы заходим в корень, следом за ним, слева направо, все узлы первого уровня, далее — все узлы второго уровня, и так вниз, пока не дойдём до последнего уровня.
- Итеративный алгоритм обхода.



Код обхода в ширину

```
def breadth_tree(node):
  root = [node]
  result = []
  # начинаем спускаться по дереву от корня,
   # заходим в каждый потомок корня,
   # кладем в наш контейнер (в данном случае очередь) в начале
текущий узел, затем его потомков,
   # непосредственно сам узел обрабатывается первым, затем его
дети. Иными словами, следуем принципу FIFO
  while root:
     queue = []
     for current in root:
        # print(current.data)
        result.append(current.data)
        if current.left:
           queue += [current.left]
        if current.right:
           queue += [current.right]
     root = queue

  return result

tree = Node(16)
tree.insert(11)
tree.insert(18)
tree.insert(9)
tree.insert(13)
tree.insert(17)
tree.insert(21)
print(breadth_tree(tree))
```

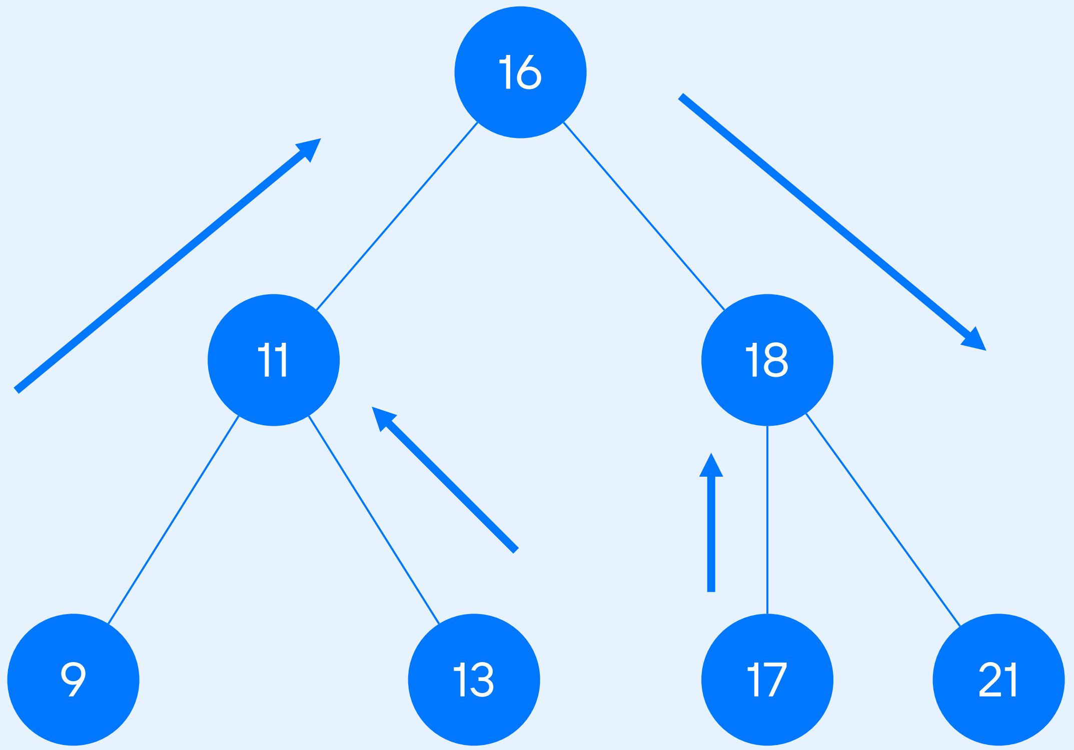



```
[16, 11, 18, 9, 13, 17, 21]
```

Обход в глубину (DFT)

- Рекурсивный обход всех вершин.
- Снизу вверх, слева направо, справа налево.
- Обходим сначала левое поддерево (L), потом вершину (N), затем правое поддерево (R): LNR (симметричный обход).
- Обходим сначала правое поддерево (R), потом вершину (N), затем левое поддерево (L): RNL (симметричный обход).
- Обходим сначала вершину (N), затем левое поддерево (L), потом правое поддерево (N): NLR (прямой обход).
- Результат симметричных обходов — отсортированная последовательность.

9	11	13	16	17	18	21



Dept-first traversal

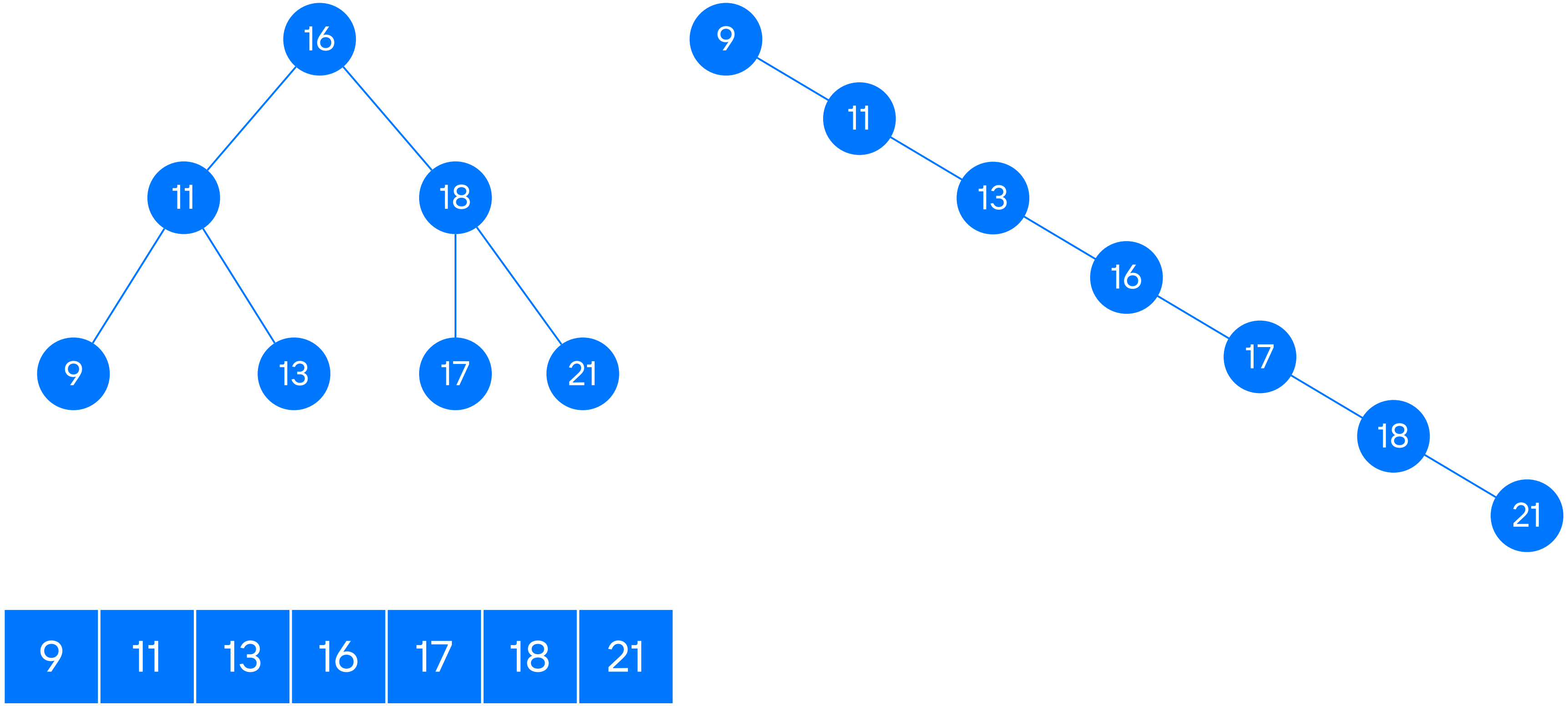
```
def dept_traversal(root):
	if root is None:
		return

	dept_traversal(root.left)
	print(root.data)
	dept_traversal(root.right)
```

LNR

```
def dept_traversal(root):
	if root is None:
		return

	dept_traversal(root.right)
	print(root.data)
	dept_traversal(root.left)
```

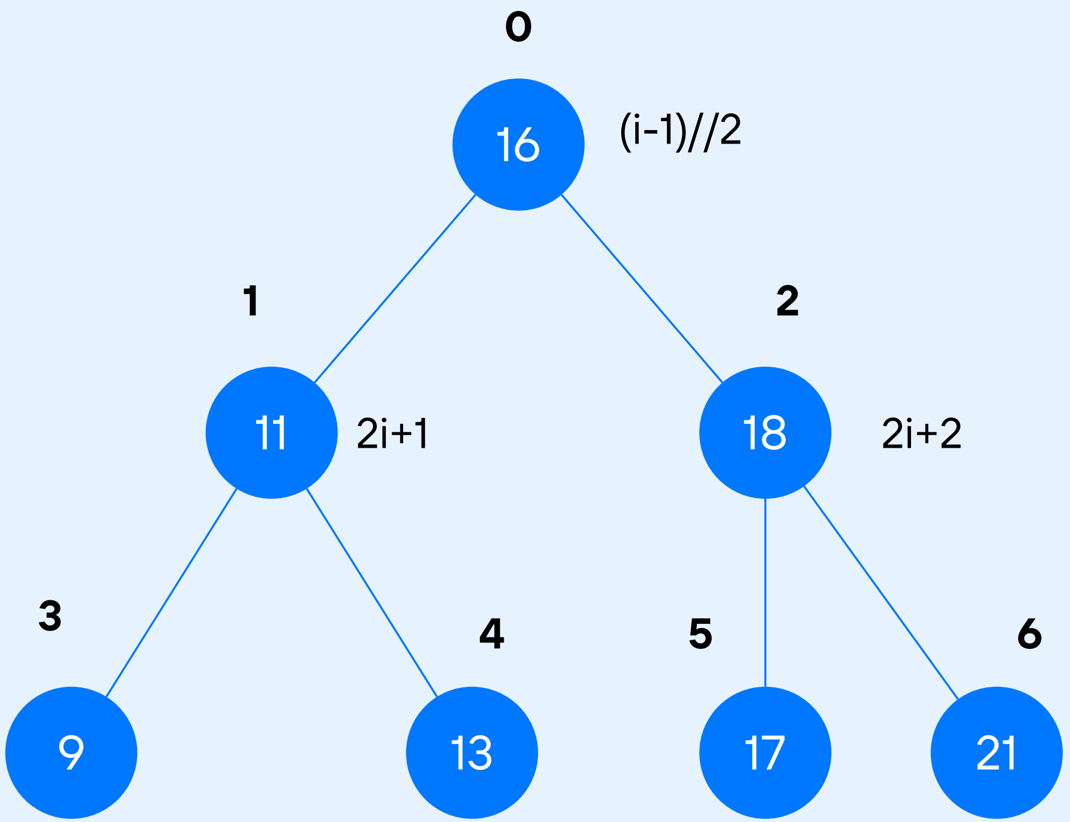
RNL

Представление бинарного дерева поиска в массиве



- Не надо хранить указатели.
- Родитель — (i – 1)/2.
- Левый потомок — 2i + 1.
- Правый потомок — 2i + 2.
- Напоминает ли это какую-нибудь операцию, которую мы разбирали ранее?)

16	11	18	9	13	17	21



Очередь с приоритетом

1

Принцип известной нам очереди — **FIFO**

2

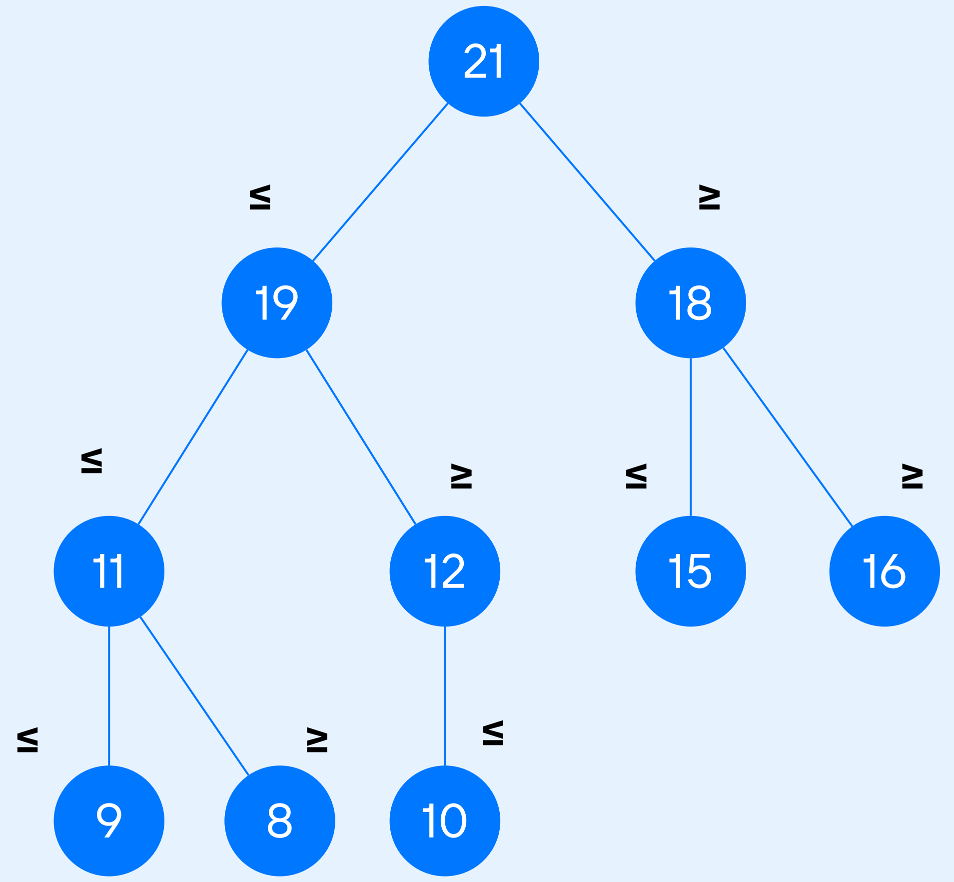
Вставка — **push** — и получение элемента с максимальным приоритетом — **pop**

3

Оптимальный вариант реализации с использованием **кучи**

Двоичная куча (max-куча)

- Бинарное дерево.
- Почти полное дерево. Все уровни, за исключением нижнего, должны быть заполнены.
- Все значения, находящиеся ниже узла, должны быть меньше него. То есть если мы возьмём узел 19, то все значения ниже него будут меньше 19.
- Отсюда вывод: приоритет (значение) любого узла должен быть не меньше, чем приоритет его потомков.
- Нижний уровень всегда заполняется слева направо. Если сейчас мы решим добавить узел, то мы поставим его в качестве правого потомка узла 12.
- Самый приоритетный элемент всегда хранится в корне.



Выборка

1

Получение максимального значения осуществляется за O(1)

2

Мы всегда возвращаем корень

Псевдокод получения максимального значения родителя и потомков

```
function pop() {
        return H.array[0]
}
```

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

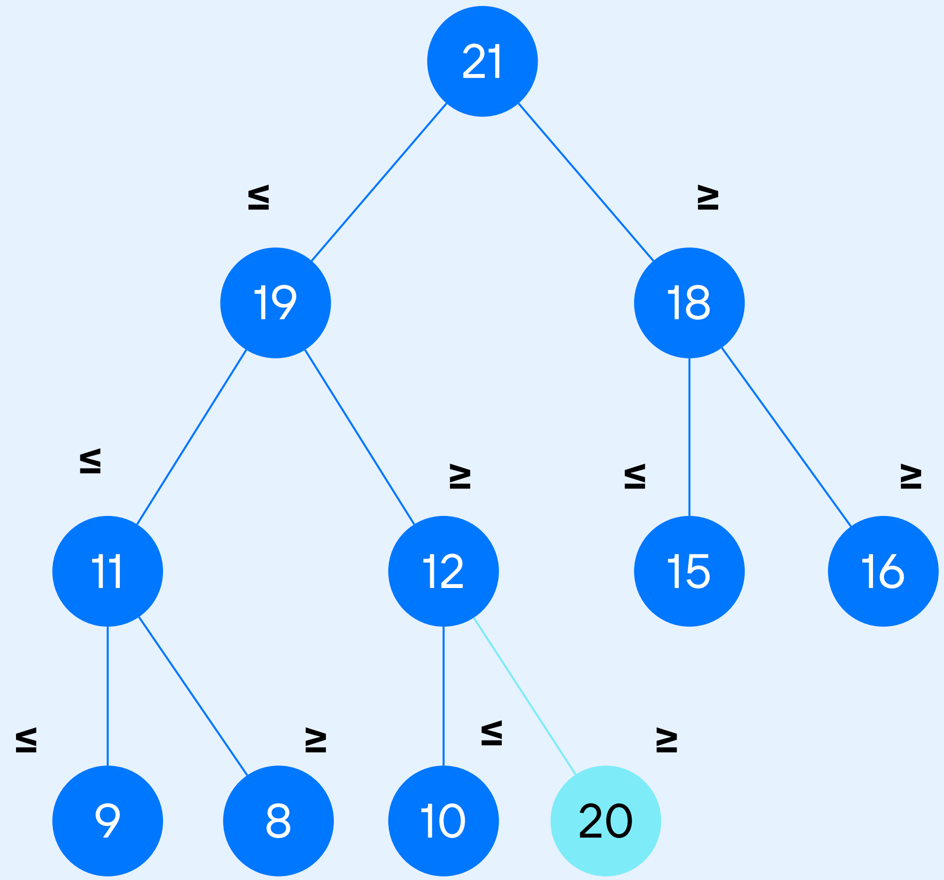
```
function parent(i) {
        if i == 0 {
                return i
        }
        return (i - 1)/2
}
```

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

Вставка элемента

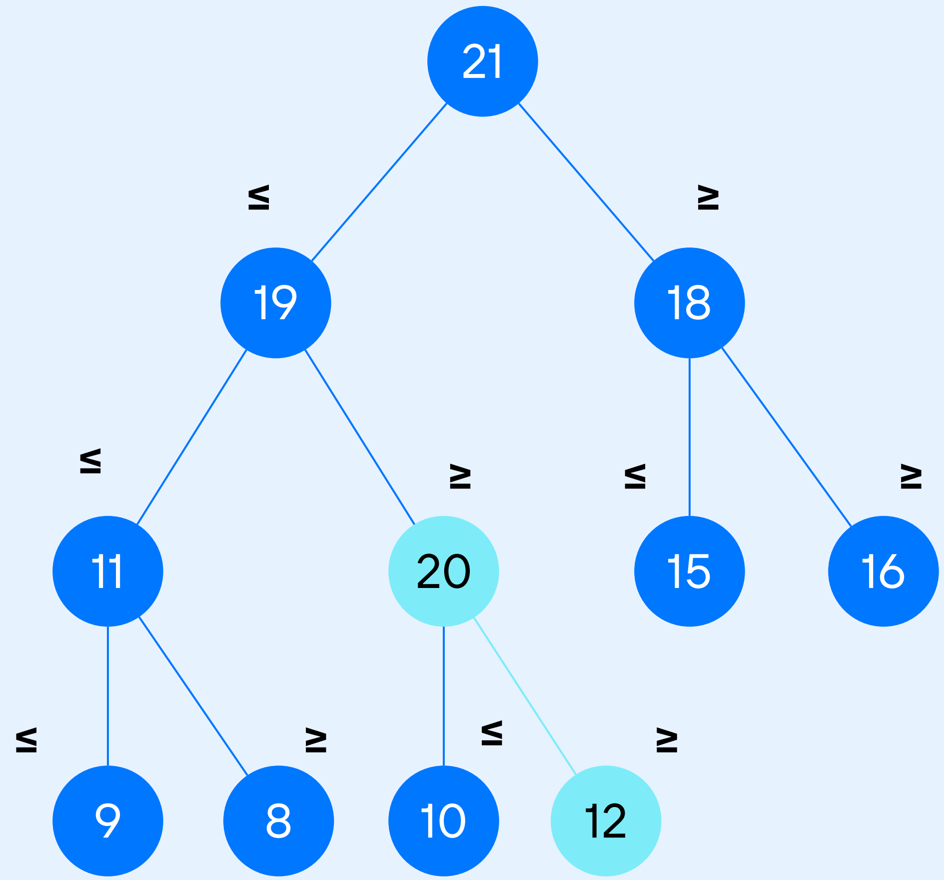
- Вставляем вниз, дорисовывая нижний уровень.
  В нашем случае — в конец массива.
- В случае если свойства кучи нарушаются, поднимаем элемент выше, меняя его местами с родителями.
- Делаем это до тех пор, пока наш элемент не дойдёт до узла, который больше него.
- Подъём и вставка элемента в худшем случае требуют количества перестановок, равного высоте дерева.
- Откуда такая сложность: количество элементов n, так как у каждого элемента по 2 потомка, получаем n ~ 2^h, отсюда h = log(n).

21	19	18	11	12	15	16	9	8	10	20



Вставка элемента

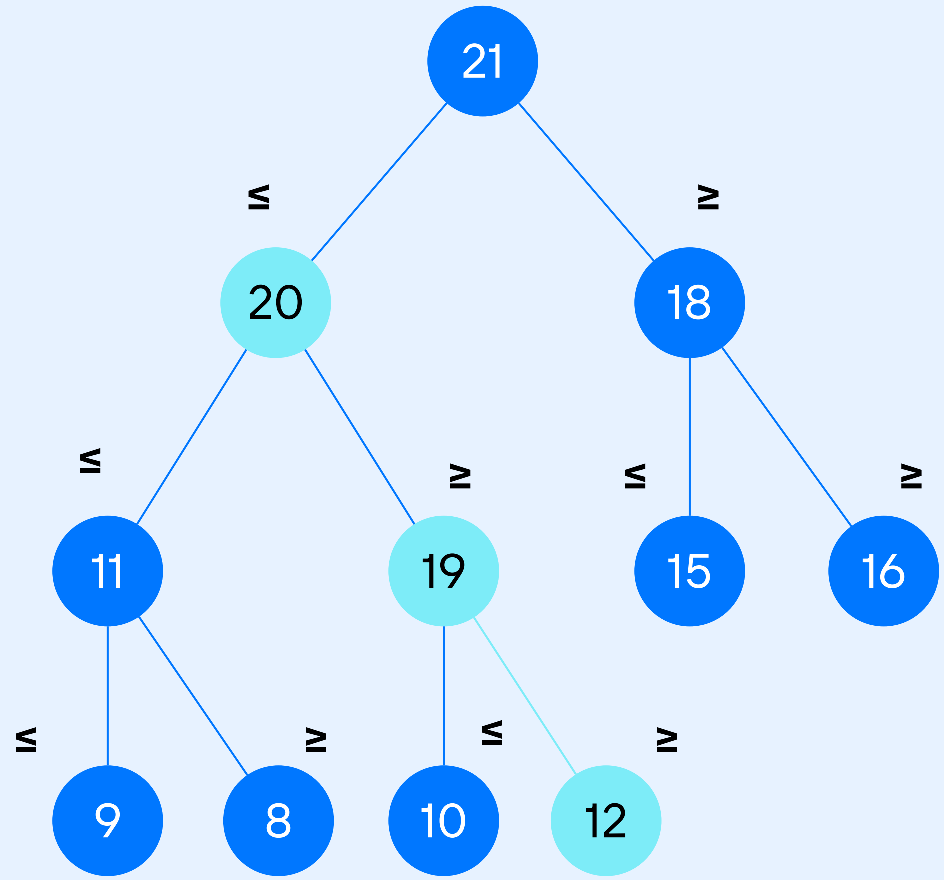
- Вставляем вниз, дорисовывая нижний уровень. В нашем случае — в конец массива.
- В случае если свойства кучи нарушаются, поднимаем элемент выше, меняя его местами с родителями.
- Делаем это до тех пор, пока наш элемент не дойдёт до узла, который больше него.
- Подъём и вставка элемента в худшем случае требуют количества перестановок, равного высоте дерева.
- Откуда такая сложность: количество элементов n, так как у каждого элемента по 2 потомка, получаем n ~ 2^h, отсюда h = log(n).



21	19	18	11	20	15	16	9	8	10	12

Вставка элемента

- Вставляем вниз, дорисовывая нижний уровень. В нашем случае — в конец массива.
- В случае если свойства кучи нарушаются, поднимаем элемент выше, меняя его местами с родителями.
- Делаем это до тех пор, пока наш элемент не дойдёт до узла, который больше него.
- Подъём и вставка элемента в худшем случае требуют количества перестановок, равного высоте дерева.
- Откуда такая сложность: количество элементов n, так как у каждого элемента по 2 потомка, получаем n ~ 2^h, отсюда h = log(n).



21	20	18	11	19	15	16	9	8	10	12

Код

- Как должна выглядеть функция push():
- Вставляем в конец массива и далее поднимаем вверх до тех пор, пока родитель не окажется больше добавляемого элемента.

```
function push(key) {
            H.array.appaend(key)
            up(H, H.array[length(H.array) - 1])
}
```

```
function up(H, index) {
    while H.array[index] > H.array[parent(index)] {
            swap(H, parent(index), parent(index))
            index = parent(index)
    }
}
```

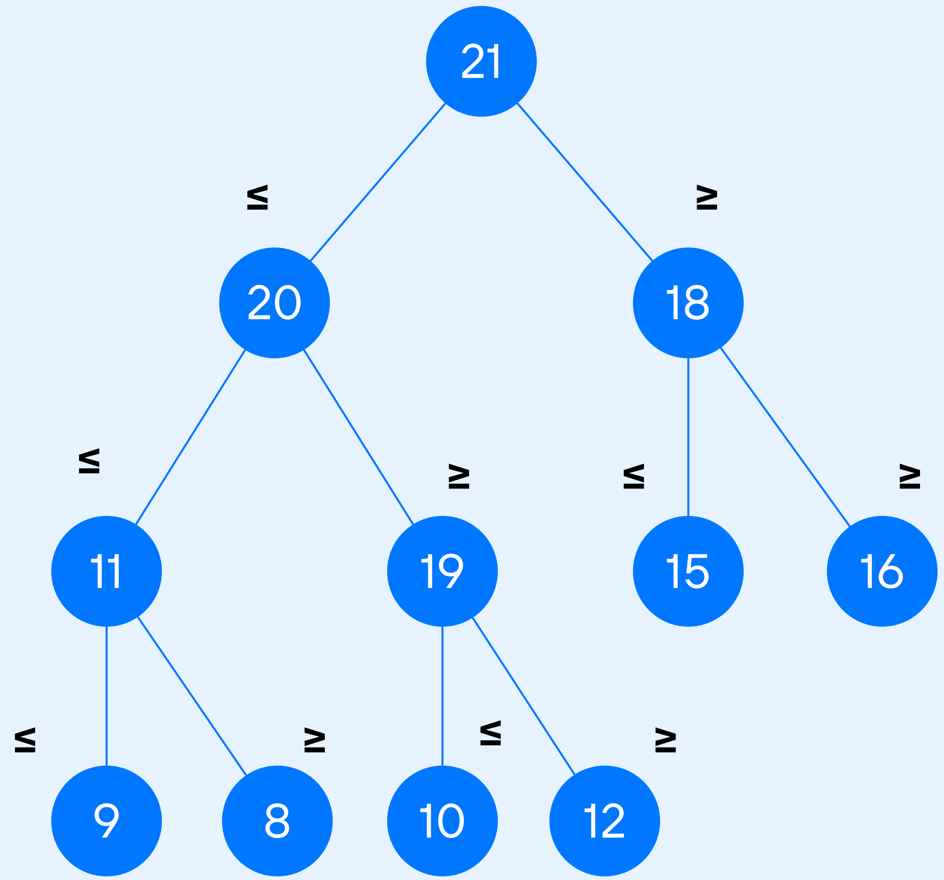

```
function push(key) {
            H.array.appaend(key)
            up(H, H.array[length(H.array) - 1])
}
```


- На Python код может показаться чуть сложнее, тем не менее суть остаётся той же:

```
def push(heap, element):
  size = len(heap)
  # вставляем элемент
  heap.append(element)
  if size != 0:
    # восстанавливаем свойства кучи проходя от середины к ее корню,
    # то есть к нулевому элементу
    up(heap, size - 1)


def recover(heap, border, i):
  largest = i
  # по уже известным нам формулам получим левого и правого потомков
  left = 2 * i + 1
  right = 2 * i + 2

  # проверяем что бы наши потомки не выходили за пределы массива
  # и продолжаем проталкивать наше значение
  if left < border and heap[i] < heap[left]:
    largest = left

  if right < border and heap[largest] < heap[right]:
    largest = right

  if largest != i:
    # swap: меняем значения местами
    heap[i], heap[largest] = heap[largest], heap[i]
    recover(heap, border, largest)
```

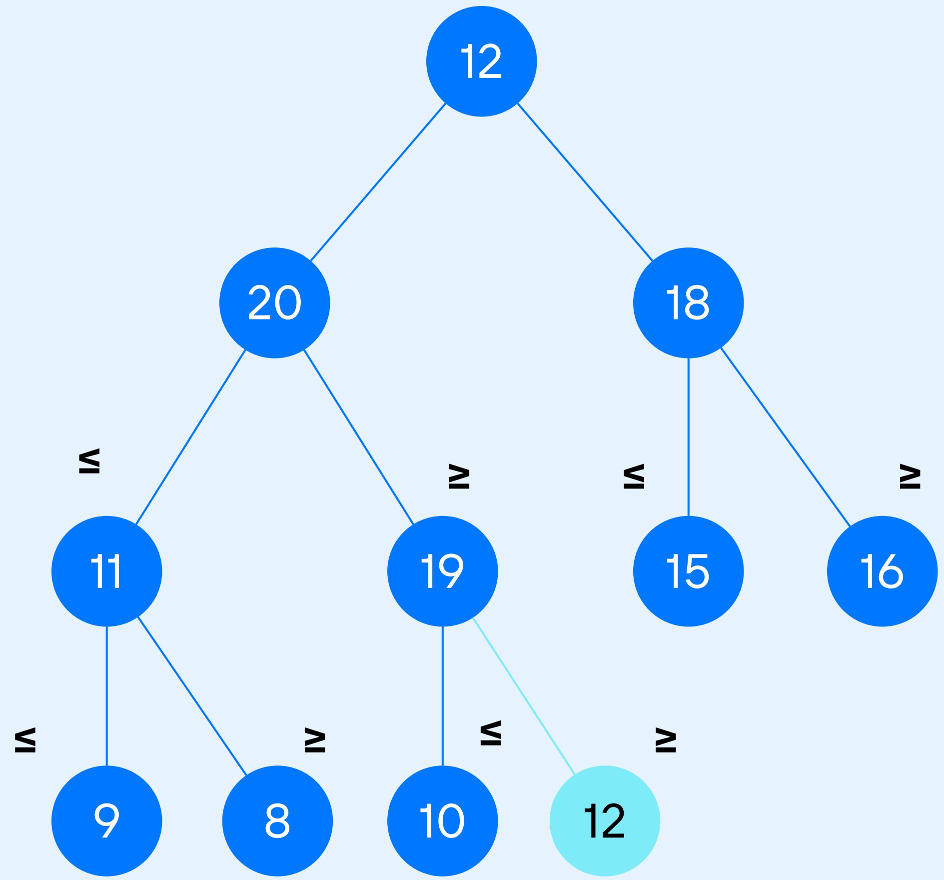
Удаление

- Метод pop всегда выбирает максимальный элемент — это всегда корень кучи.
- На его место передвигаем последний элемент.
- 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 поднимаем число, сравнивая его только с родителем, а при удалении мы опускаем его вниз, сравнивая с потомками.



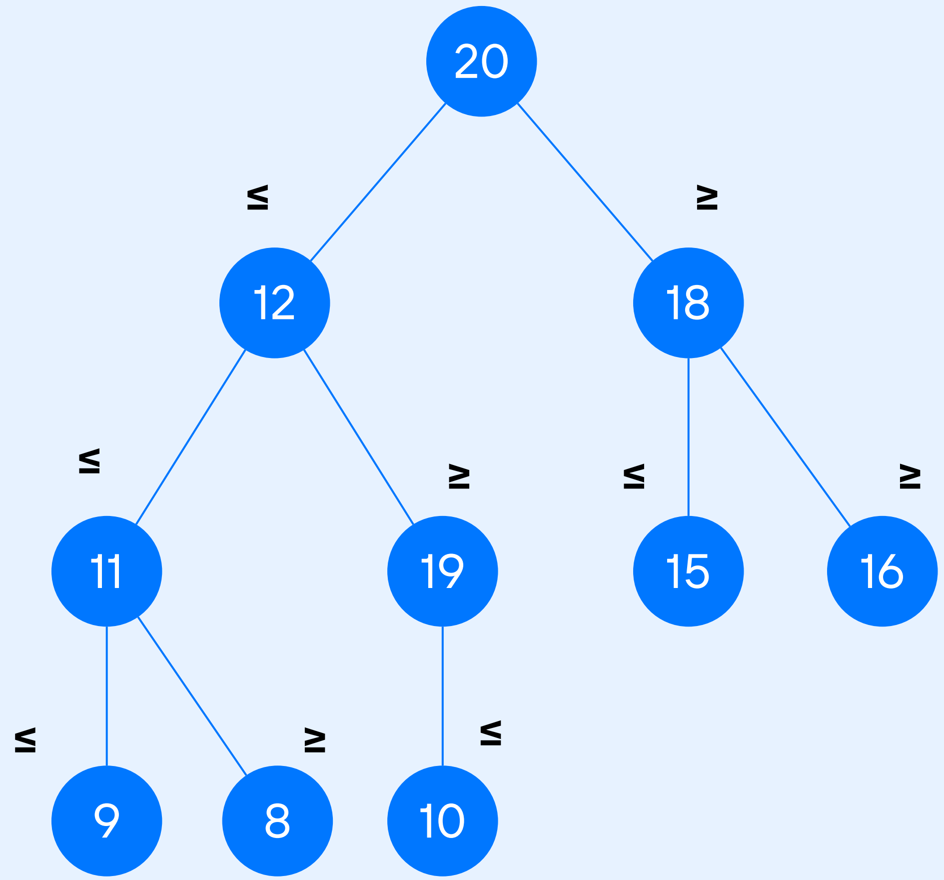
Удаление

- Метод pop всегда выбирает максимальный элемент — это всегда корень кучи.
- На его место передвигаем последний элемент.
- 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 поднимаем число, сравнивая его только с родителем, а при удалении мы опускаем его вниз, сравнивая с потомками.



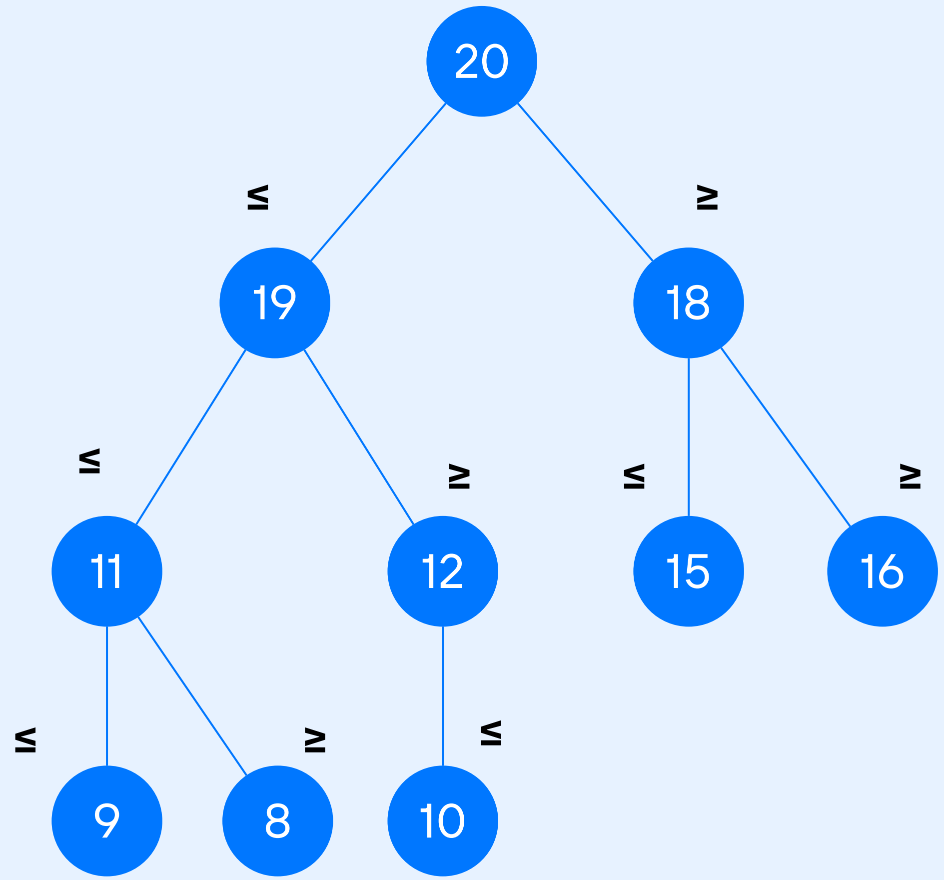
Удаление

- Метод pop всегда выбирает максимальный элемент — это всегда корень кучи.
- На его место передвигаем последний элемент.
- 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 поднимаем число, сравнивая его только с родителем, а при удалении мы опускаем его вниз, сравнивая с потомками.



Удаление

- Метод pop всегда выбирает максимальный элемент — это всегда корень кучи.
- На его место передвигаем последний элемент.
- 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 поднимаем число, сравнивая его только с родителем, а при удалении мы опускаем его вниз, сравнивая с потомками.



Подытожим

1

Куча — структура, которая возвращает за O(1) максимальный элемент.

2

Позволяет удалять и вставлять за **O(log(n))**.

Будем ВКонтакте!

Псевдокод получения максимального значения родителя и потомков

```
function pop() {
        return H.array[0]
}
```

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

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

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

Псевдокод получения максимального значения родителя и потомков

```
function pop() {
        return H.array[0]
}
```

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

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

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

Псевдокод получения максимального значения родителя и потомков

```
function pop() {
        return H.array[0]
}
```

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

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

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

Псевдокод получения максимального значения родителя и потомков

```
function pop() {
        return H.array[0]
}
```

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

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

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

Псевдокод получения максимального значения родителя и потомков

```
function pop() {
        return H.array[0]
}
```

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

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

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