

Деревья поиска

- Бинарное дерево поиска
- АВЛ дерева



Где используются

- set и map в C++ в java TreeMap и TreeSet в Lisp
списки
- В качестве индекса в базах данных
- Управление виртуальной памятью в ядре
(хранение адресов и т.д.)
- Структура папок - как пример иерархической
структурой

Дерево

- Иерархическая структура данных
- В отличии от уже нам известных массивов и списков дерево - нелинейная структура
- Состоит из нод (узлов)
- Узел - хранилище для данных и указателей на следующие узлы (потомки)
- Лист - узел у которого нет потомков
- Корневой узел - вершина дерева
- Всегда может быть только один родитель и множество дочерних нод
- Высота дерева - расстояние от корня до самого нижнего элемента

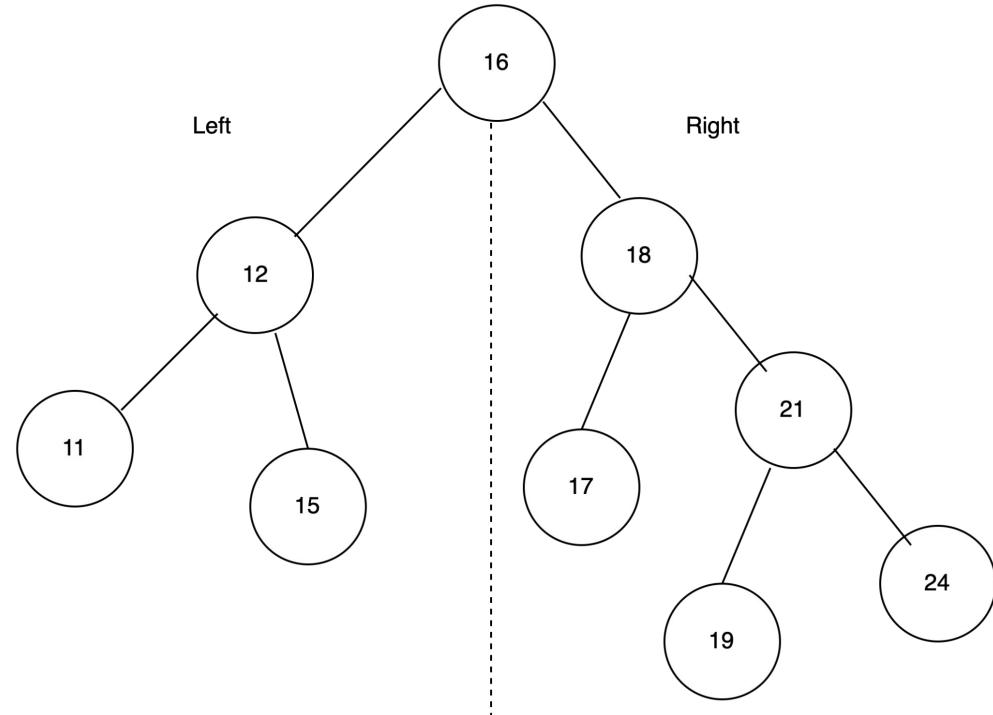
Бинарное дерево

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

```
Node {  
    data int  
    left *Node  
    Right *Node  
}
```

Бинарное дерево поиска

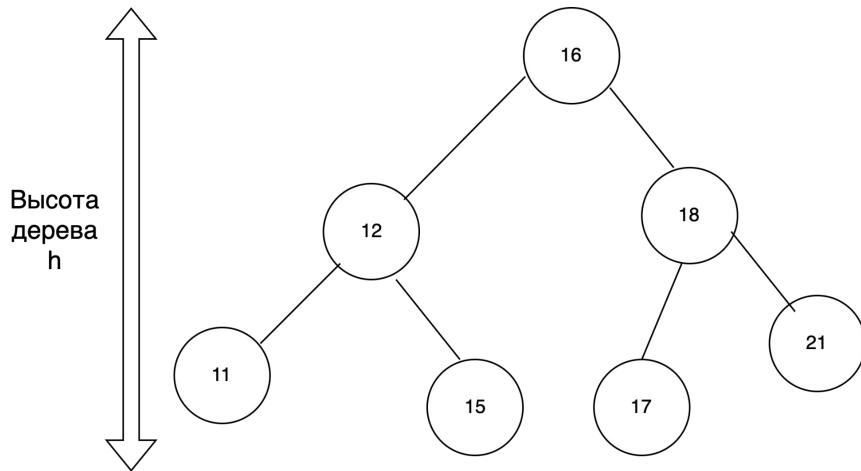
- Максимум два потомка
- У каждой вершины в левом поддереве все элементы не больше чем в самой вершине и в правом поддереве
- Поддерево каждого узла тоже являются бинарными деревьями поиска



Проекция бинарного дерева, является отсортированной последовательностью

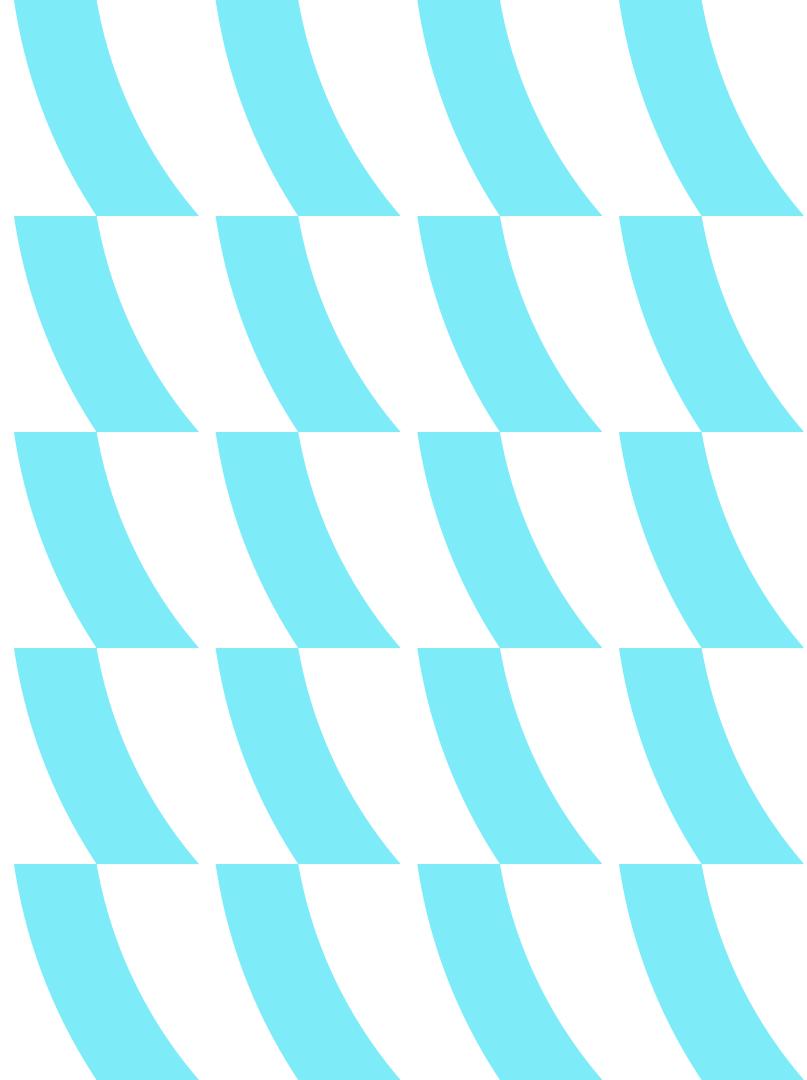
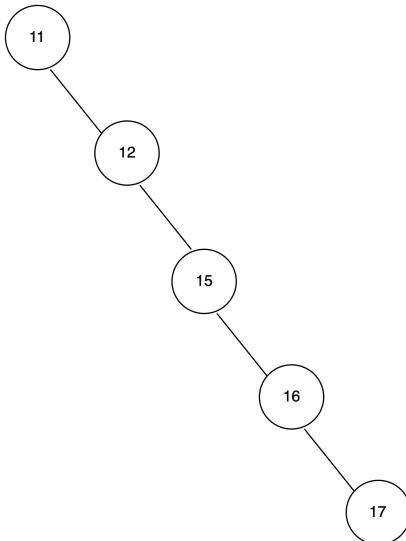
Сложность

- В идеале, у сбалансированного дерева, у каждого узла кроме листовых потомков ровно два дочерних
- Подсчитаем количество узлов n для высоты h : $n = 1 + 2 + 2^2 + 2^4 = 2^{(h+1)} - 1$
- Прологарифмируем результат: $h = \log(n+1) - 1$
- Сложность операций у нас всегда $O(h)$, то есть $O(\log n)$



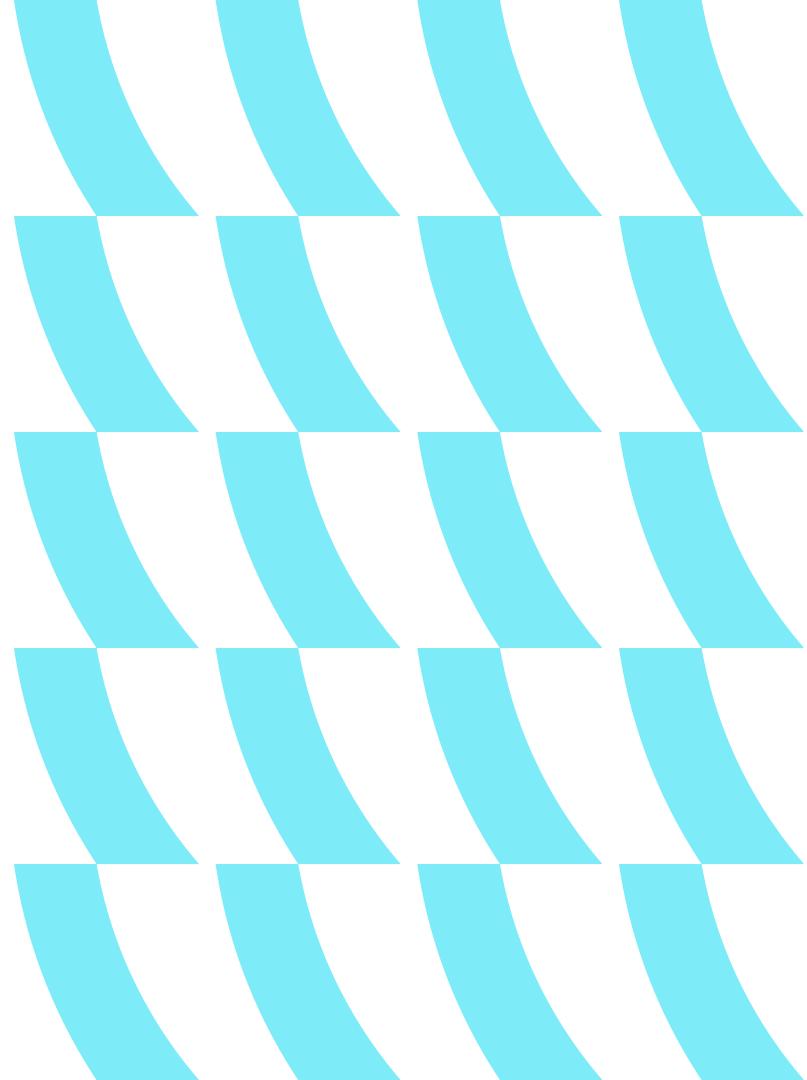
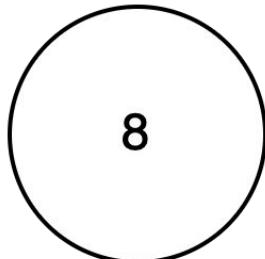
Сложность

- Сложность $O(\log n)$ возможна не всегда
- Если дерево вырождено или стремится к таковому, то сложность стремится к $O(n)$



Сложность

- У нас по прежнему не более двух потомков



Поиск минимума и максимума

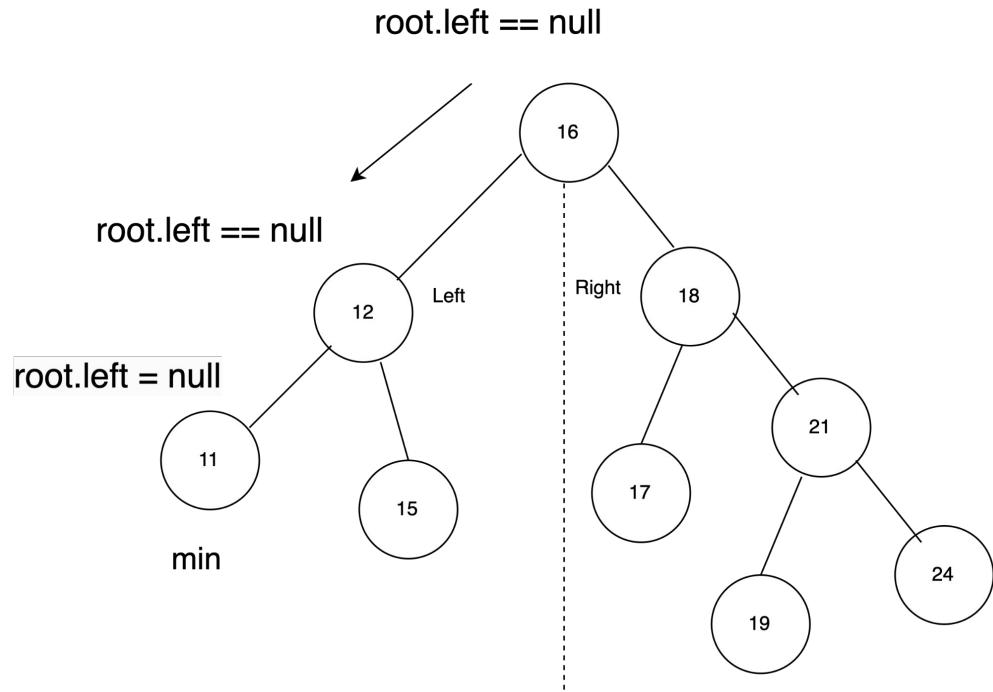
- Из свойств бинарного дерева поиска следует, что минимум всегда внизу левого поддерева
- Максимум всегда внизу правого поддерева

```
function minimum(root) {  
    // если спускаясь вниз  
    // влево, у узла нет левого  
    // потомка значит мы нашли  
    // минимальный элемент  
    if root.left == null {  
        return root  
    }  
    return minimum(root.left)  
}  
  
function maximum(root) {  
    // если спускаясь вниз  
    // вправо у узла нет левого  
    // потомка значит мы нашли  
    // минимальный элемент  
    if root.right == null {  
        return root  
    }  
    return maximum(root.right)  
}
```

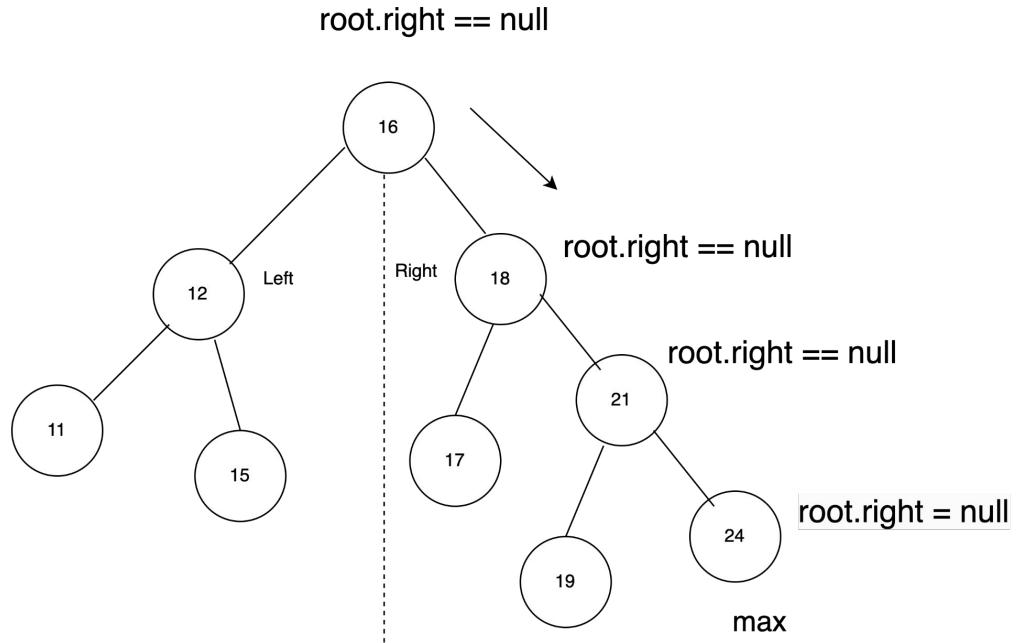
Итерационный подход

```
function minimum(node) {  
    while node.left != null {  
        node = node.left  
    }  
    return node  
}  
  
function maximum(node) {  
    while node.right != null {  
        node = node.right  
    }  
    return node  
}
```

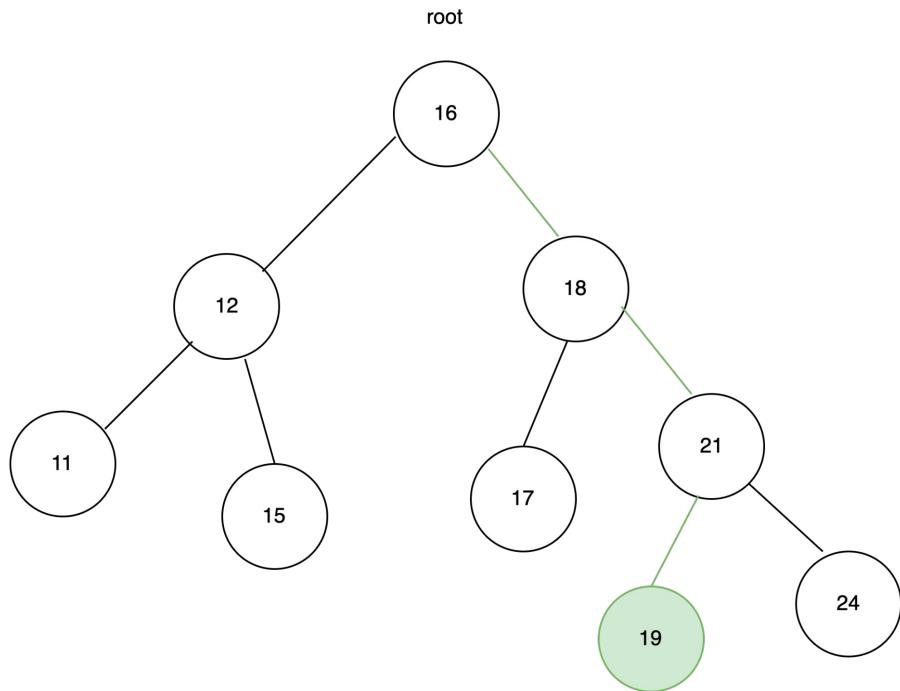
Поиск минимума



ПОИСК МАКСИМУМА



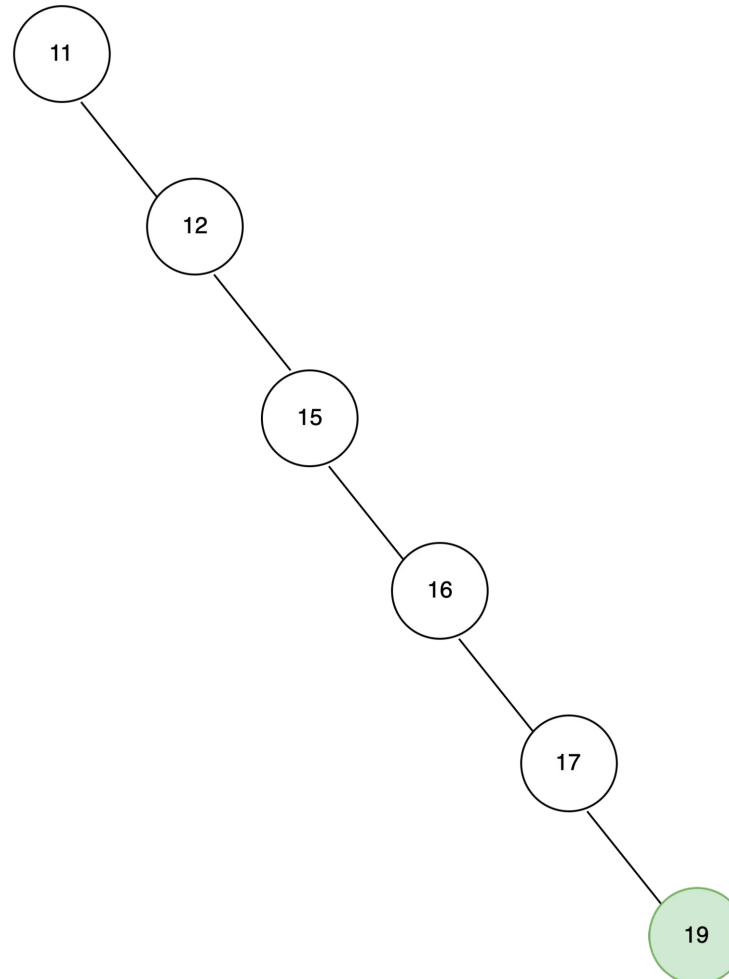
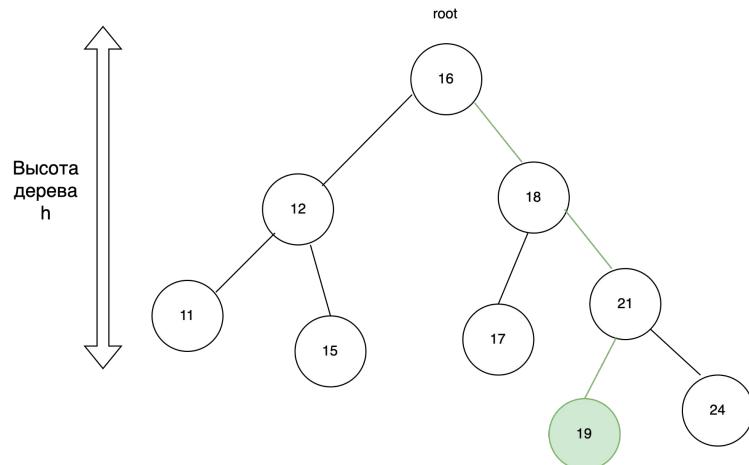
Поиск элемента



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

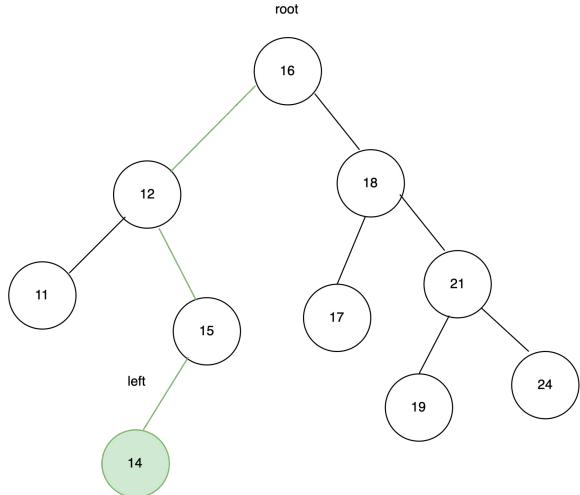
Поиск элемента

- В «сбалансированном» дереве $O(\log n)$
- В худшем случае $O(n)$
- Дерево называется сбалансированным, если высоты двух поддеревьев каждого из его узлов отличаются не более чем на единицу



Вставка элемента

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



```
function insert(root, data) {  
    // создаем новый узел  
    if root == null {  
        return Node(data)  
    }  
    // определяем каким узлом будет новый элемент  
    if data < root.data {  
        root.left = insert(root.left, data)  
    }  
    if data > root.data {  
        root.right = insert(root.right, data)  
    }  
    return root  
}
```

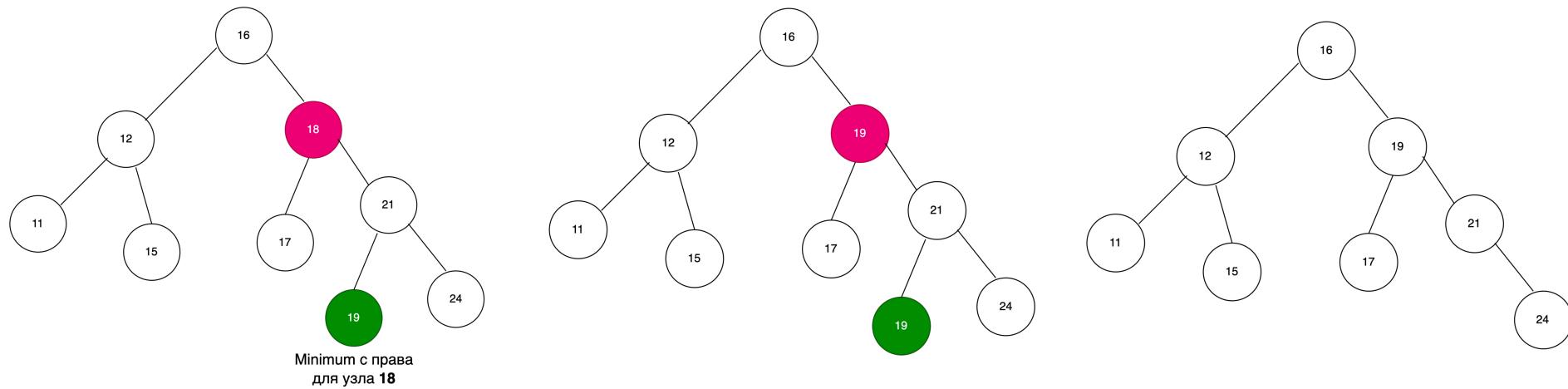
Удаление

- Удаление листа. Тут все просто, мы у родителя должны удалить соответствующий указатель.
- Удаление узла с одним дочерним элементом:
 - а. Определяем, каким потомком является узел
 - б. Переписываем соответствующий указатель
- Удаление узла с двумя дочерними узлами:
 - а. Ищем минимальное значение в правом поддереве
 - б. Ставим это значение на место удаляемого
 - с. Удаляем найденное значение с прежней позиции

Удаление

```
function delete(node, key){  
    // ищем узел по переданному значению  
    if node == null {  
        return node  
    }  
    // если значение меньше текущего узла – идем в левое поддерево  
    if key < node.data {  
        node.left = delete(node.left, key)  
    }  
    // если значение больше – идем в правое поддерево  
    if key > node.data {  
        node.right = delete(node.right, key)  
    }  
    // если ключ нашли – начинаем процедуру удаления  
    // 1. узел является листом  
    if node.left == null && node.right == null {  
        node = null  
        return node  
    }  
    // 2. узел имеет одного ребенка  
    if node.left == null {  
        node = node.right  
        return node  
    }  
    if node.right == null {  
        node = node.left  
        return node  
    }  
    // 3. узел имеет двух детей  
    min_val = minimum(node.right) // ищем минимальное значение в правом поддереве  
    node.data = min_val.data // записываем вместо удаляемого значение найденное минимальное  
    node.right = delete(node.right, min_val.data) // удаляем минимальное с его прежней позиции  
  
    return node  
}
```

Удаление



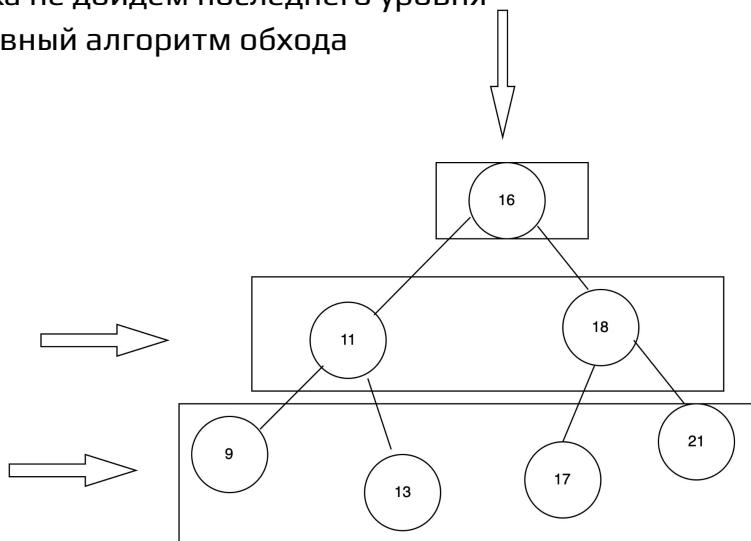
Обход дерева

- Обход в ширину
- Обход в глубину

Обход в ширину (BFT)

Breadth-First Traversal

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



Обход в ширину

```
def breadth_tree(node):
    root = [node]
    result = []
    # начинаем спускаться по дереву от корня
    while root:
        queue = []
        for current in root:
            # текущий элемент записываем
            # в результирующий массив
            # в result всегда попадает
            # корень данного поддерева
            result.append(current.data)
            # его детей записываем в очередь
            if current.left:
                queue += [current.left]
            if current.right:
                queue += [current.right]
        # продолжаем цикл для левого и правого потомков
        root = queue

    return result
```

```
class Node:
    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data

    def insert(self, data):
        if self.data is None:
            self.data = data
        if data < self.data:
            if self.left is None:
                self.left = Node(data)
            else:
                self.left.insert(data)
        elif data > self.data:
            if self.right is None:
                self.right = Node(data)
            else:
                self.right.insert(data)
```

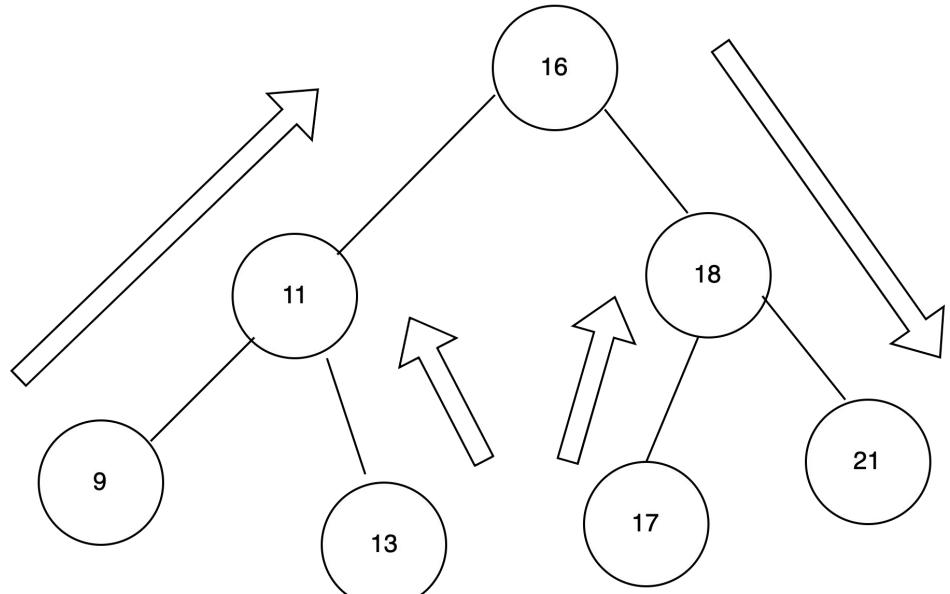
```
tree = Node(16)
tree.insert(11)
tree.insert(18)
tree.insert(9)
tree.insert(13)
tree.insert(17)
tree.insert(21)
print(breadth_tree(tree))
```

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



Обход в глубину DFT Depth-First Traversal

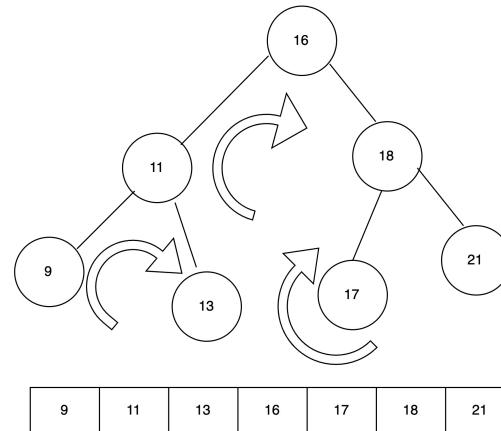
- Рекурсивный обход всех вершин
- Снизу вверх слева направо/справа налево
- Обходим в начале левое поддерево (L), потом вершину (N), затем правое поддерево (R): LNR (симметричный обход)
- Обходим в начале правое поддерево (R), потом вершину (N), затем левое поддерево (L): RNL (симметричный обход)
- Обходим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



9	11	13	16	17	18	21
---	----	----	----	----	----	----

Обход в глубину DFT LNR

- In-order обход
- Если текущий узел равен null, заканчиваем обход.
- Рекурсивно проходим левое поддерево.
- Выводим текущий узел.
- Обходим правое поддерево рекурсивно, вызывая нашу функцию.



```
def dept_traversal(root):  
    if root is None:  
        return  
  
    # в начале левое поддерево  
    dept_traversal(root.left)  
    print(root.data)  
    # затем правое  
    dept_traversal(root.right)
```

Обход в глубину DFT LNR

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

    # в начале левое поддерево
    dept_traversal(root.left, res)
    # сохраняем узел
    res.append(root.data)
    # затем правое
    dept_traversal(root.right, res)
    return res
```

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

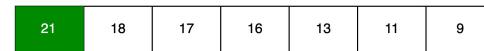
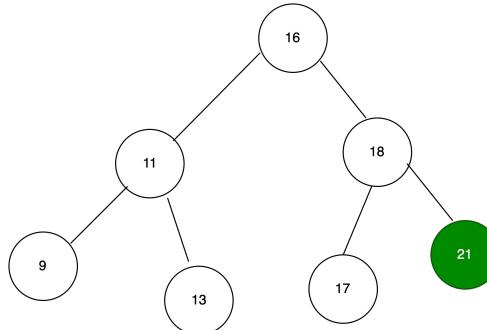
lnr = []
print(dept_traversal(tree, lnr))
```

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



Обход в глубину DFT RNL

- Reverse in-order обход
- Если текущий узел равен null, заканчиваем обход.
- Рекурсивно проходим правое поддерево.
- Выводим текущий узел.
- Обходим левое поддерево рекурсивно, вызывая нашу функцию.



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

        # в начале правое поддерево
    dept_traversal(root.right)
    print(root.data)
        # затем левое
    dept_traversal(root.left)
```

Обход в глубину DFT RNL

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

    # в начале правое поддерево
    dept_traversal(root.right, res)
    # сохраняем узел
    res.append(root.data)
    # затем левое
    dept_traversal(root.left, res)
    return res
```

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

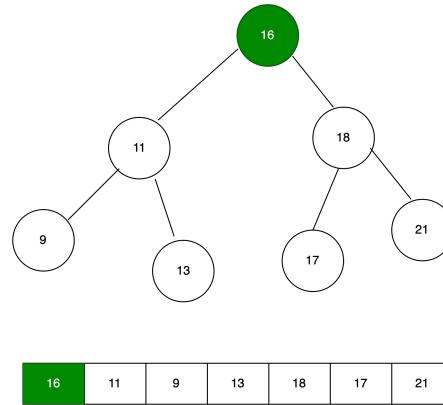
rnl = []
print(dept_traversal(tree, rnl))
```

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



Обход в глубину DFT NLR

- Pre-order обход
- прямой обход с приоритетом обхода потомков слева направо
- В начале заходим в корень дерева
- Далее обходим левое и правое поддеревья



```
def dept_traversal(root):  
    if root is None:  
        return  
  
    # в начале выводим корень  
    print(root.data)  
    # далее левое поддерево  
    dept_traversal(root.left)  
    # затем правое  
    dept_traversal(root.right)
```

Обход в глубину DFT NLR

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

    # в начале добавляем корень
    res.append(root.data)
    # далее левое поддерево
    dept_traversal(root.left, res)
    # затем правое
    dept_traversal(root.right, res)
    return res
```

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

nlr = []
print(dept_traversal(tree, nlr))
```

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



Варианты

Pre-order, NLR

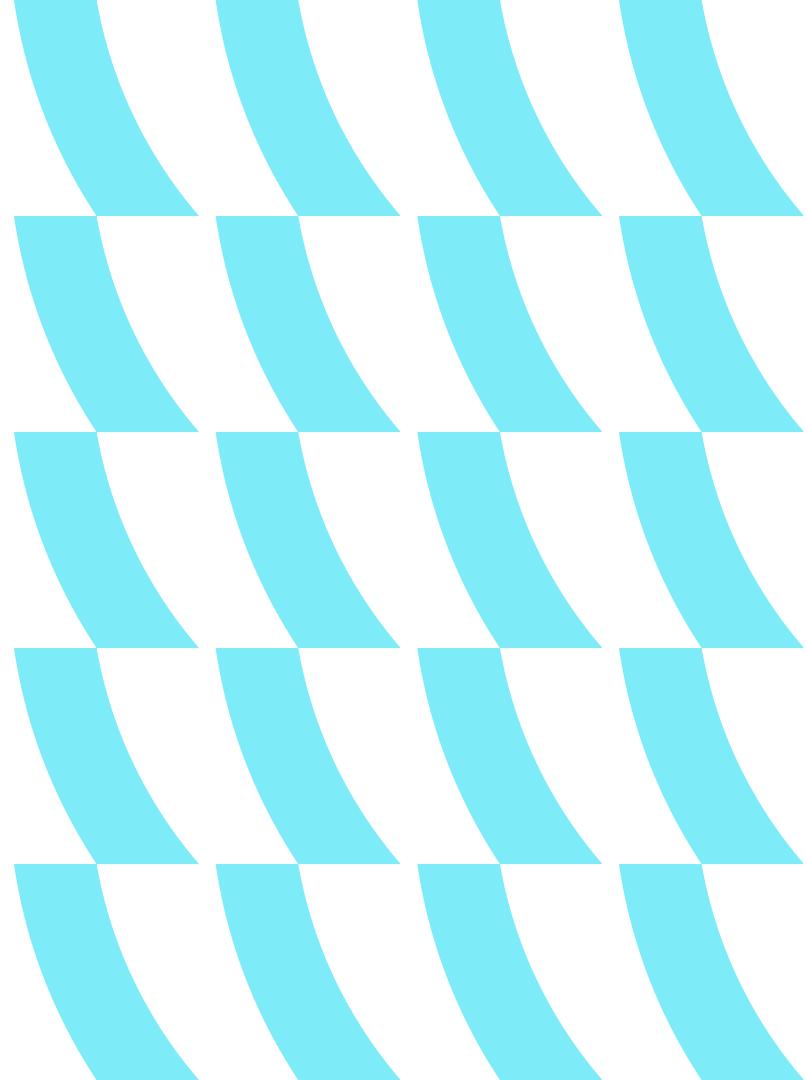
Post-order, LRN

In-order, LNR

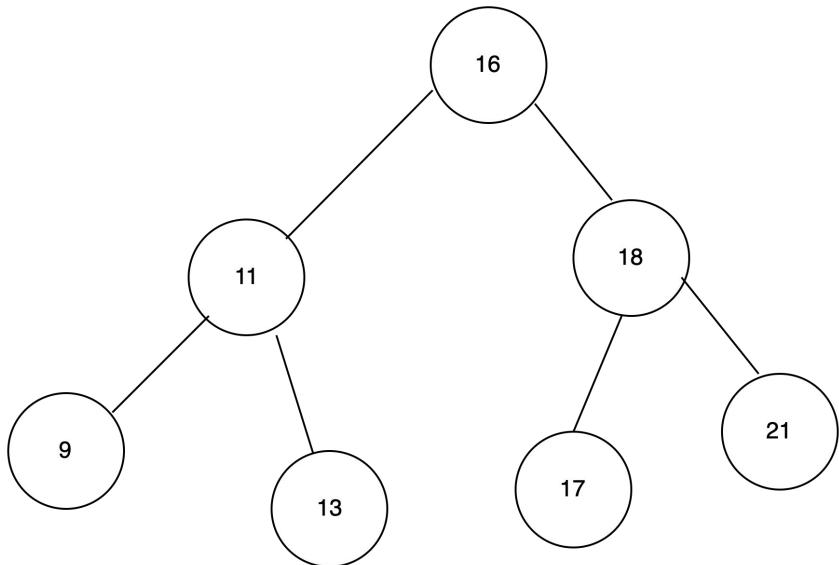
Reverse pre-order, NRL

Reverse post-order, RLN

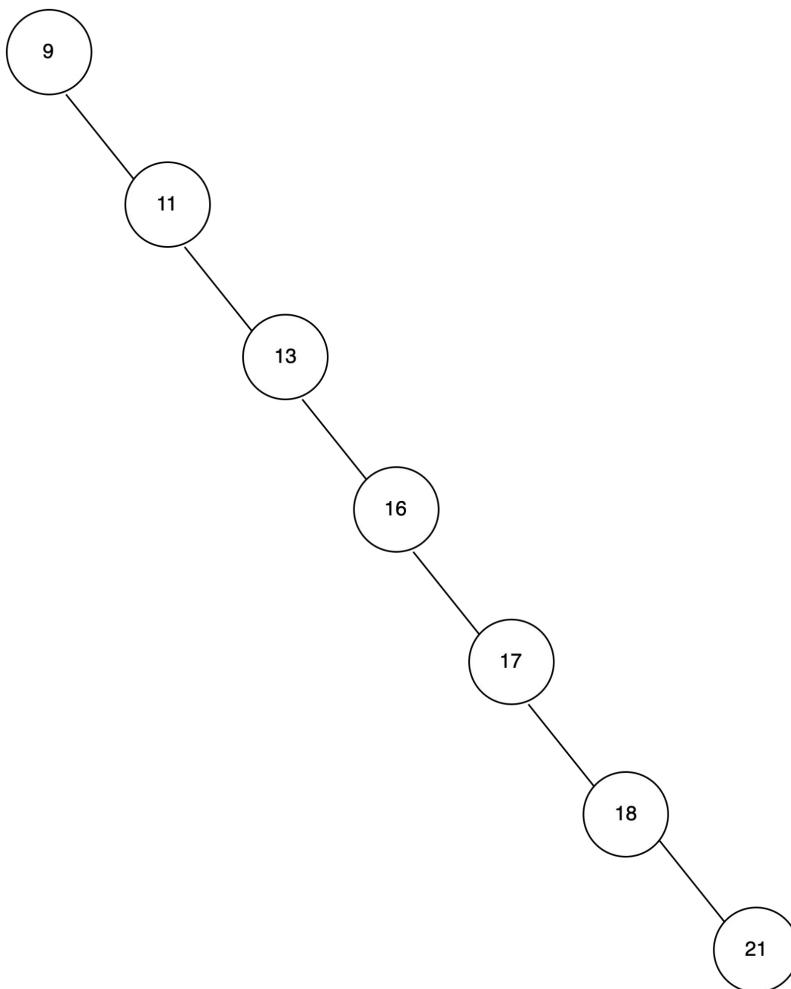
Reverse in-order, RNL



Представление бинарного дерева в массиве

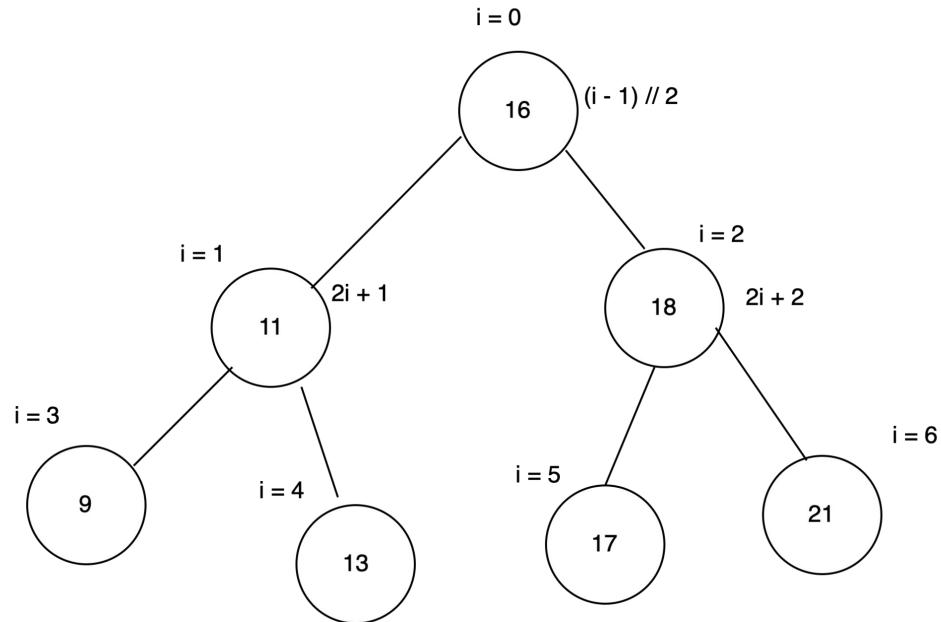


9	11	13	16	17	18	21
---	----	----	----	----	----	----



Представление бинарного дерева в массиве

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

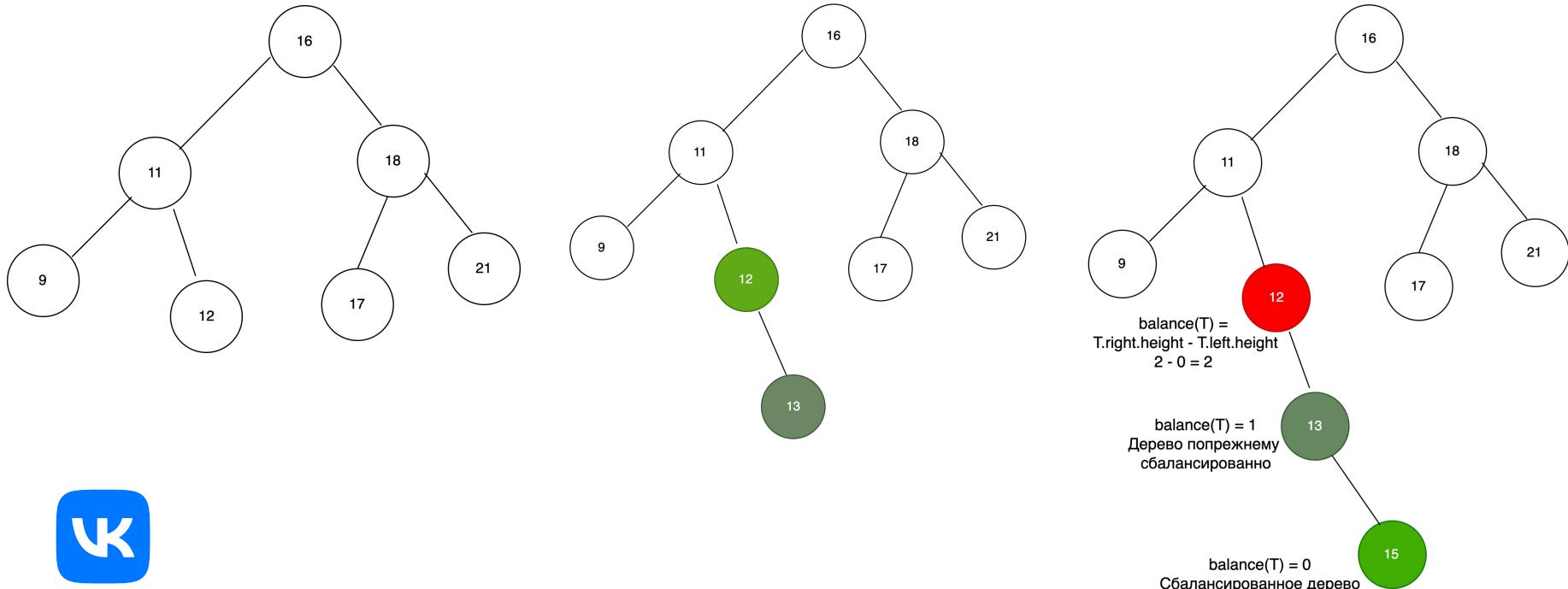


16	11	18	9	13	17	21
----	----	----	---	----	----	----

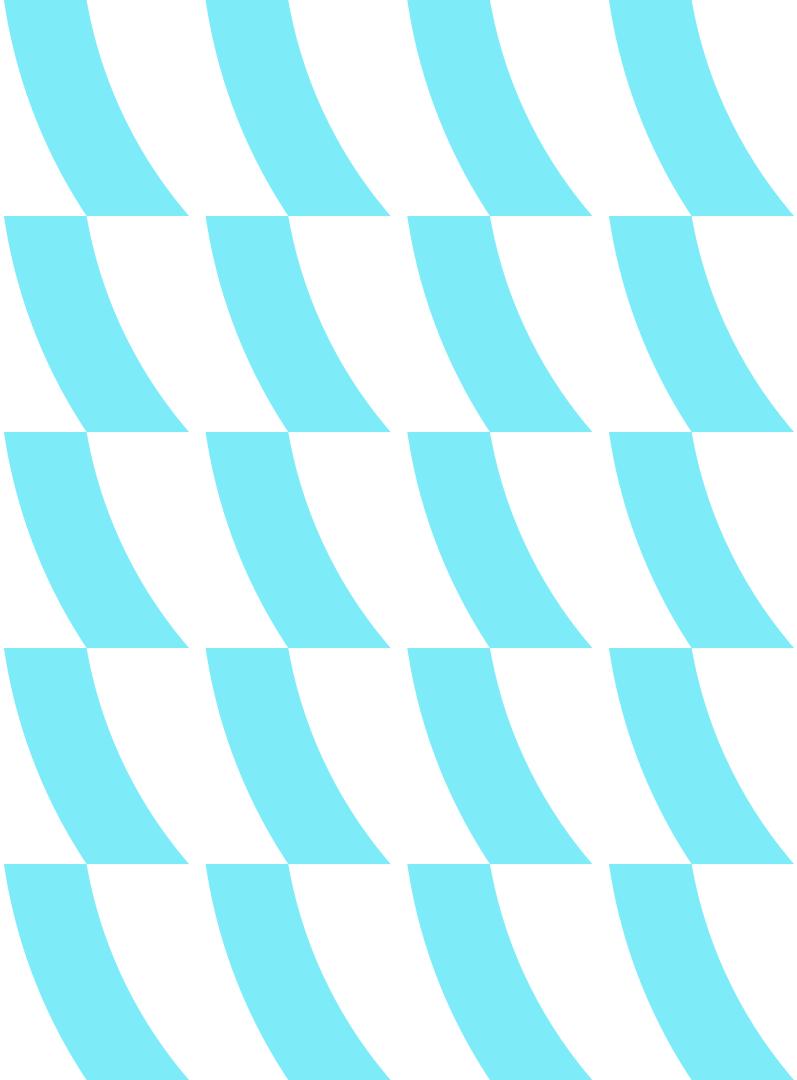
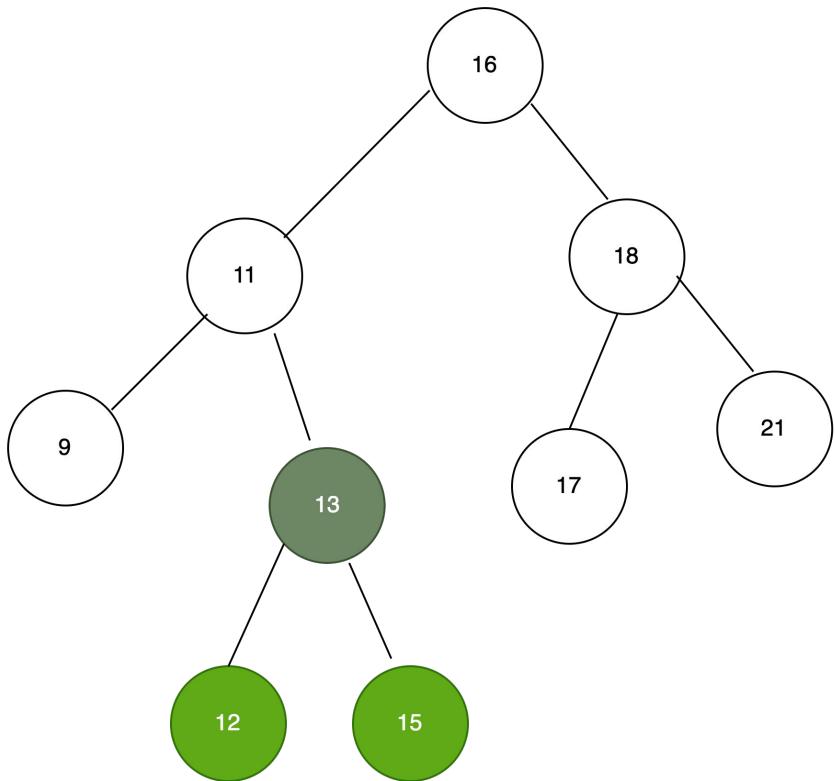
Поговорим немного о сбалансированных деревьях (AVL деревья)

- AVL — аббревиатура, образованная первыми буквами создателей (советских учёных) Адельсон-Вельского Георгия Максимовича и Ландиса Евгения Михайловича.
- Добавим новое понятие: баланс $\text{balance}(T) = T.\text{right.height} - T.\text{left.height}$
- Дерево называется сбалансированным если для каждого узла справедливо выражение $|\text{balance}(T)| \leq 1$. Иными словами высота левого и правого поддерева для каждого узла не будет различаться больше чем на 1.
- В случае если после операции вставки/удаления баланс стал $>= 2$ необходима балансировка (поворот дерева)

Баланс дерева



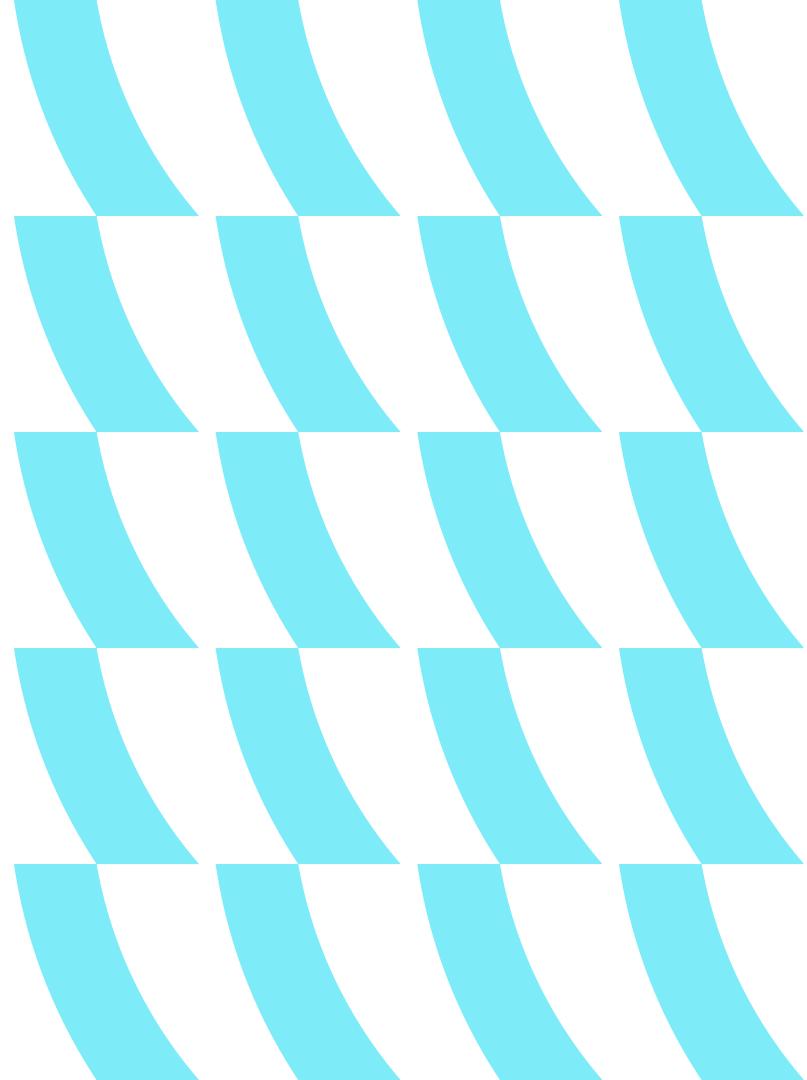
Баланс дерева



Поворот дерева

Виды вращений

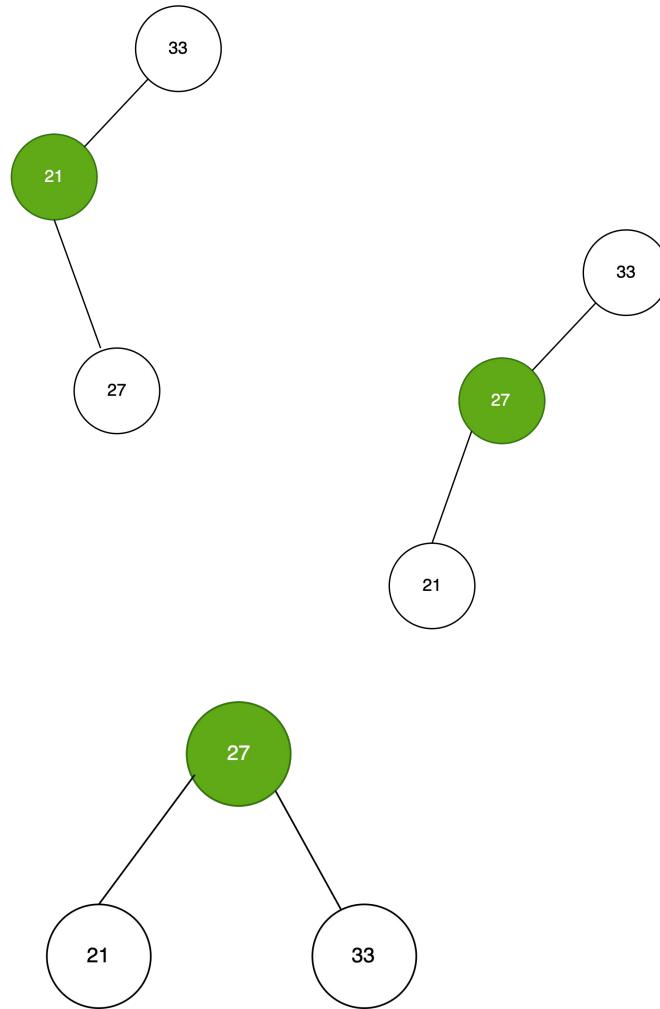
- Левое - малое левое вращение **L**
- Правое - малое правое вращение **R**
- Влево-вправо - большое правое вращение **LR**
- Вправо-влево - большое левое вращение **RL**
- Для отслеживания разницы высот добавим поле **balance_factor**. По этому значению будем понимать когда нам необходим поворот.



Большое правое вращение

- Появился указатель на родителя, чтобы можно было осуществлять вращение
- Появились поле баланса, для отслеживания необходимости вызывать функцию вращение

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.parent = None  
        self.left = None  
        self.right = None  
        self.balance_factor = 0  
  
class Tree:  
    def __init__(self):  
        self.root = None
```



Пример вращения

```
def insert(self, key):
    # создаем новый узел
    node = Node(key)
    p = None
    cur = self.root

    # поиск родителя для нового узла
    while cur is not None:
        p = cur
        if node.data < cur.data:
            cur = cur.left
        else:
            cur = cur.right

    # p - родитель нового элемента
    node.parent = p
    if p is None:
        # новый узел становится
        # корнем дерева
        self.root = node
    # определяем каким именно потомком
    # станет новый узел
    elif node.data < p.data:
        p.left = node
    else:
        p.right = node
    # при необходимости
    # балансируем дерево
    self.updateBalance(node)

# проставляем balance_factor для узла
def updateBalance(self, node):
    if node.balance_factor < -1 or node.balance_factor > 1:
        self.rebalance(node)
    return

if node.parent is not None:
    # если вставляем левый узел
    # декрементируем баланс
    if node == node.parent.left:
        node.parent.balance_factor -= 1
    # если вставляем правый узел
    # инкрементируем баланс
    if node == node.parent.right:
        node.parent.balance_factor += 1
    if node.parent.balance_factor != 0:
        # вызываем рекурсивно, чтобы понять
        # нужна ли нам балансировка дерева
        self.updateBalance(node.parent)
```

Пример вращения

```
def reBalance(self, node):
    print("До вращения")
    self.prettyPrint()
    if node.balance_factor > 0:
        if node.right.balance_factor < 0:
            self.rightRotate(node.right)
            self.leftRotate(node)
        else:
            self.leftRotate(node)
    # кейс нашего примера
    # у узла 33 баланс меньше 0
    elif node.balance_factor < 0:
        # у узла 21 баланс больше 0
        if node.left.balance_factor > 0:
            # повернули 21 влево
            self.leftRotate(node.left)
            # повернули 33 вправо
            self.rightRotate(node)
        else:
            self.rightRotate(node)

    print("После вращения")
    self.prettyPrint()

def leftRotate(self, node):
    print("Левое вращение")
    right_child = node.right
    node.right = right_child.left
    if right_child.left is not None:
        right_child.left.parent = node

    right_child.parent = node.parent
    if node.parent is None:
        self.root = right_child
    elif node == node.parent.left:
        node.parent.left = right_child
    else:
        node.parent.right = right_child
    right_child.left = node
    node.parent = right_child

    node.balance_factor = node.balance_factor - 1 - max(0, right_child.balance_factor)
    right_child.balance_factor = right_child.balance_factor - 1 + min(0, node.balance_factor)
```



Пример вращения

```
bst = Tree()  
bst.insert(9)  
bst.insert(11)  
bst.insert(13)  
bst.insert(16)  
bst.insert(17)  
bst.insert(18)  
bst.insert(21)
```

До вращения

R----9

R----11

R----13

Левое вращение

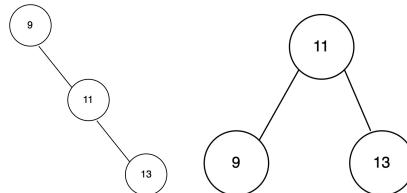
После вращения

R----11

L----9

R----13

рисунок 1



До вращения

R----11

L----9

R----13

R----16

R----17

рисунок 2

Левое вращение

После вращения

R----11

L----9

R----16

L----13

R----17

R----18

До вращения

R----11

L----9

R----16

L----13

R----17

R----18

Левое вращение

После вращения

R----16

L----11

| L----9

| R----13

R----17

R----18

рисунок 3

До вращения

R----16

L----11

| L----9

| R----13

R----17

R----18

R----21

Левое вращение

После вращения

R----16

L----11

| L----9

| R----13

R----18

L----17

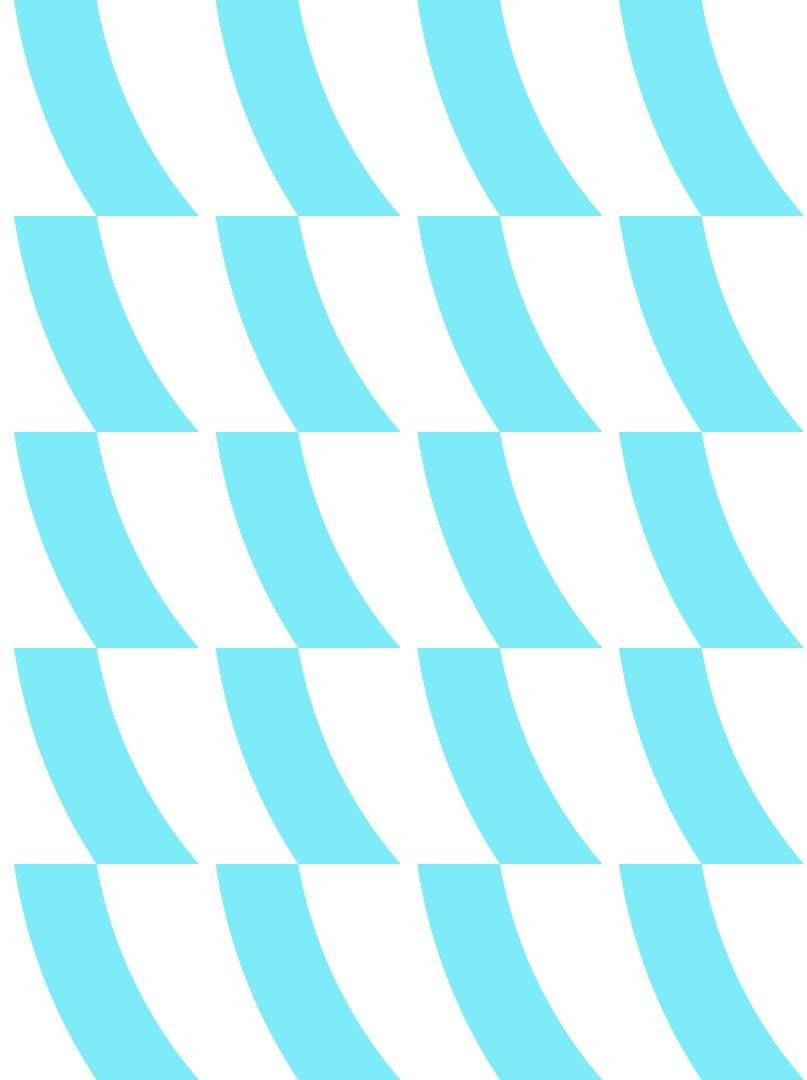
R----21

рисунок 4



Сложность

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	$\log(n)$	$\log(n)$



Резюме

- Операции за $O(\log(n))$
- Обход дерева в ширину и глубину
- Вращение дерева
- Когда нужна балансировка дерева



Всем спасибо и хорошего вечера:)

