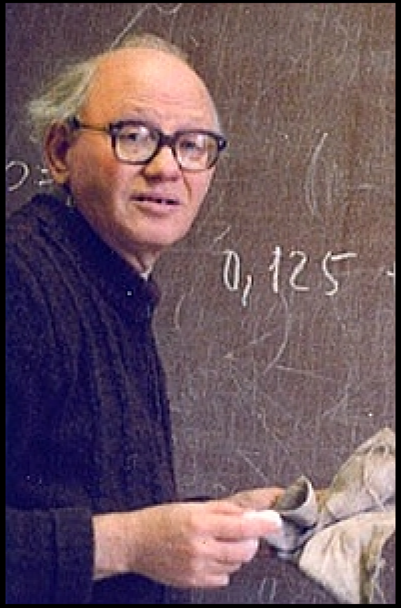
АВЛ-деревья.Оценка высоты АВЛ-дерева

Выполнил: Миронов Егор БИВТ-23-8

История создания

В 1962 году советские учёные Георгий Максимович Адельсон-Вельский и Евгений Михайлович Ландис опубликовали статью, в которой описали новый тип двоичного дерева поиска. Они назвали его «АВЛ-деревом» в честь своих инициалов.

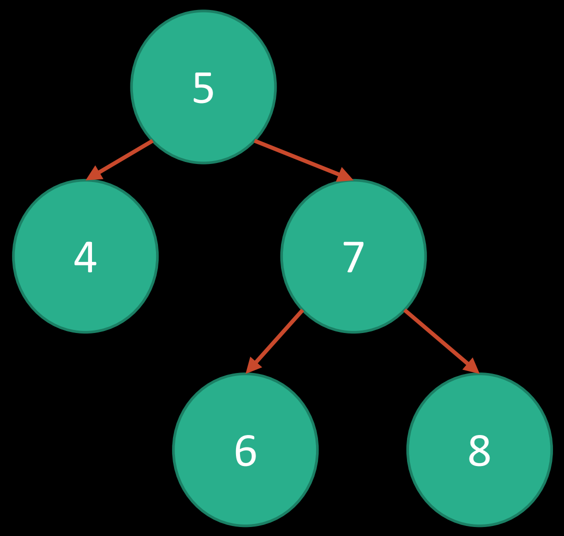


АВЛ-деревья

— это сбалансированное по высоте двоичное дерево поиска. Для каждой вершины высота её двух поддеревьев не различается более чем на 1.

Поддерживаются следующие основные операции:

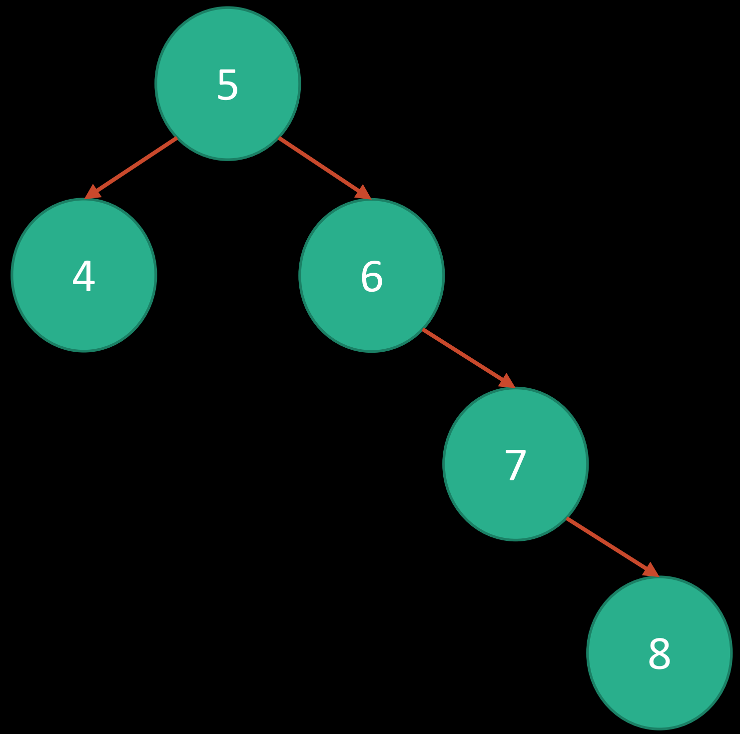
- Поиск значения
- Подвешивание новой вершины
- Удаление вершины



BST – Binary search tree

Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами:

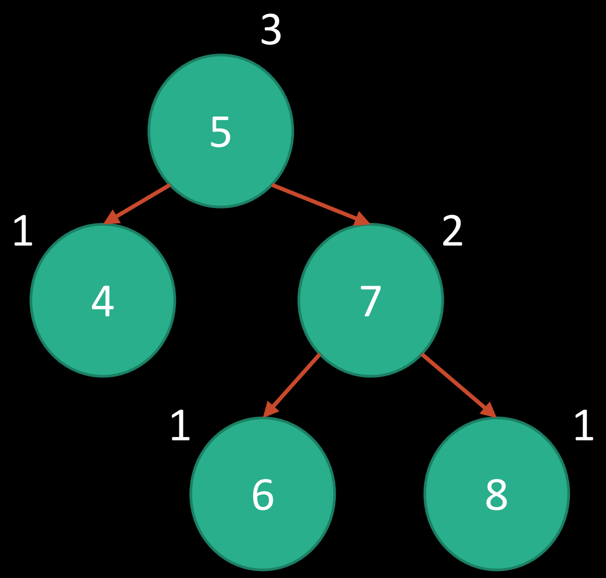
- Значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева.
- При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется.
- Рекурсивная реализация поиска

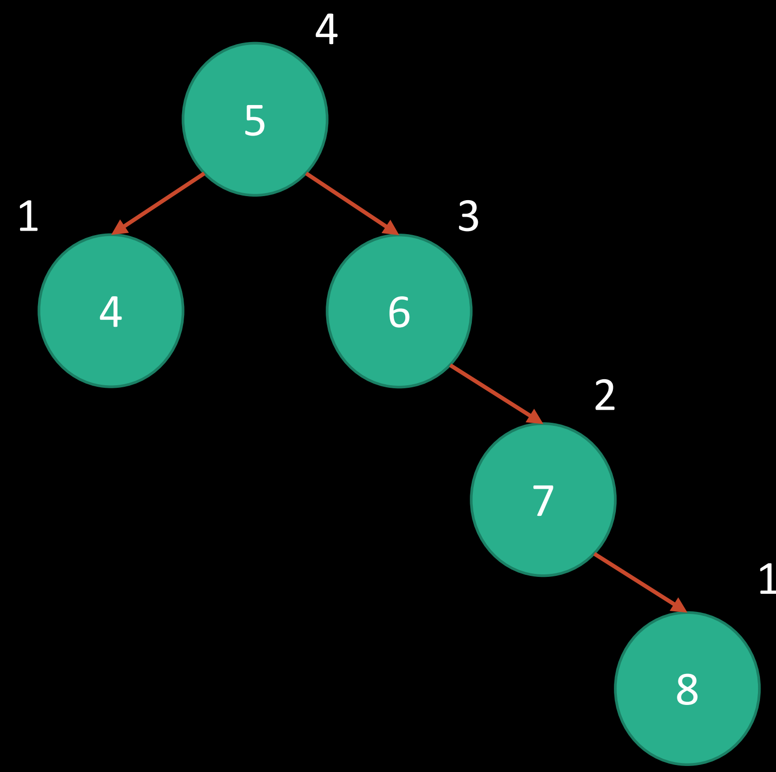


АВЛ-дерево

VS

BST





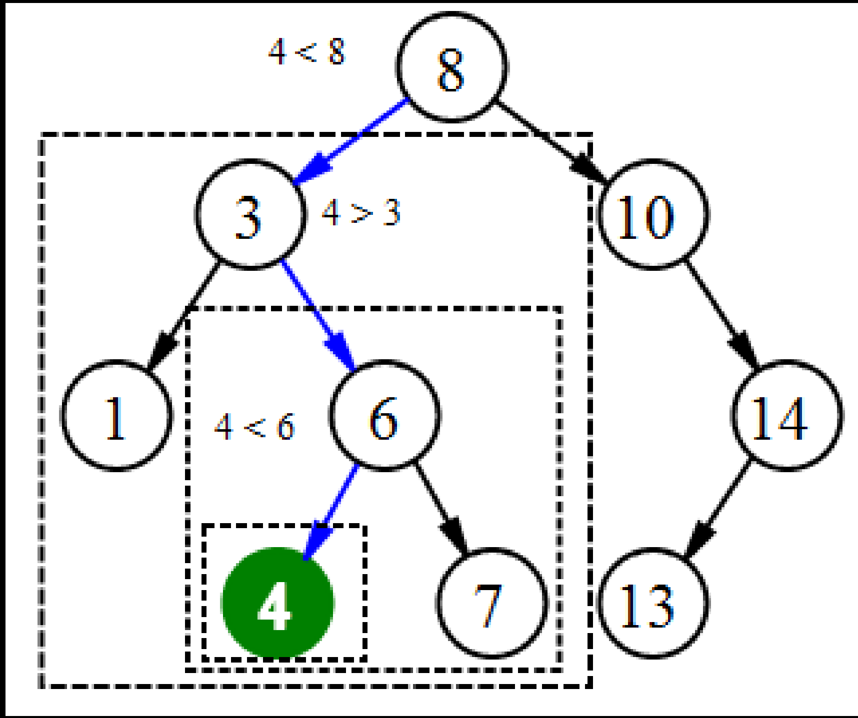
Реализация на

Сложность хранения: O(n), n – количество вершин
Хранимые атрибуты вершины:

- значение вершины(ключ)
- вес вершины (уровень/высота) = max(вес левой вершины, вес правой вершины) + 1
- указатель(ссылка) на левую вершину (значение левой вершины < значение текущей)
- указатель(ссылка) на правую вершину (значение правой вершины > значение текущей)

```python
class Node:
    def __init__(self, d):
        self.key = d
        self.height = 1
        self.left = None
        self.right = None
```

Поиск в АВЛ-дереве



Временная сложность: O(log(n)), n – количество вершин
Доказательство:
При поиске нужной вершины обход дерева ведётся всё время по правым и левым поддеревьям вершин.
Поскольку дерево сбалансировано, то глубина каждого узла не превышает log(n)

```
def find_value(key:int, node:Node)->bool:
    if key == node.key:
        return True
    if node.height == 1:
        return key == node.key
    return find_value(key, node.left) if node.key > key else find_value(key, node.right)
```

Балансировка АВЛ дерева

Балансировкой вершины называется операция, которая в случае разницы весов левого и правого поддеревьев вершины n $|h(n.l) - h(n.r)| = 2$ изменяет связь «предок-потомок» для восстановления баланса $|h(n.l) - h(n.r)| \leq 1$

4 вида балансировки:

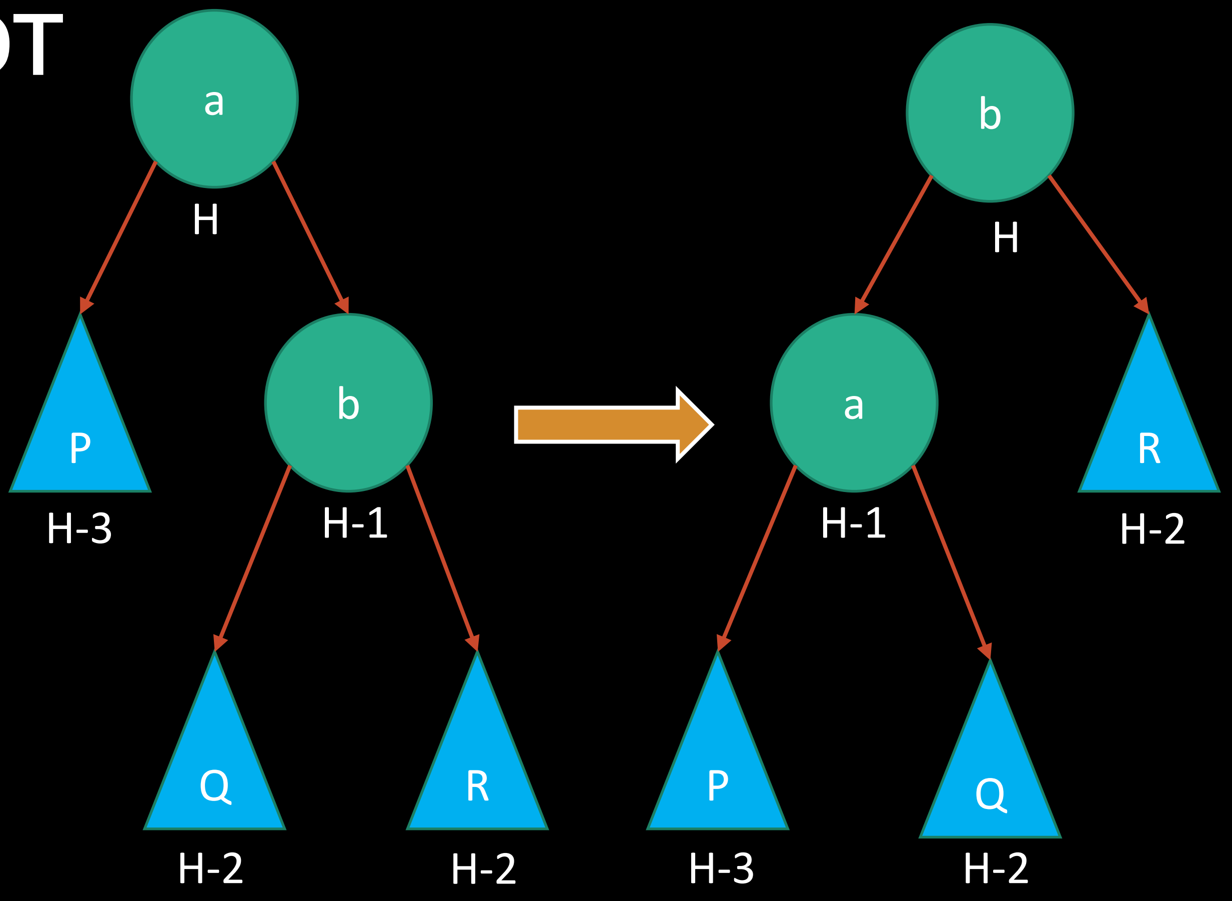
1. Малый левый поворот
2. Большой левый поворот
3. Малый правы поворот (отзеркаленный малый левый поворот)
4. Большой правый поворот (отзеркаленный большой левый поворот)

Малый левый поворот

Используется когда:

b.H = P.H + 2 и a.H = b.H + 1

Временная сложность: O(1)

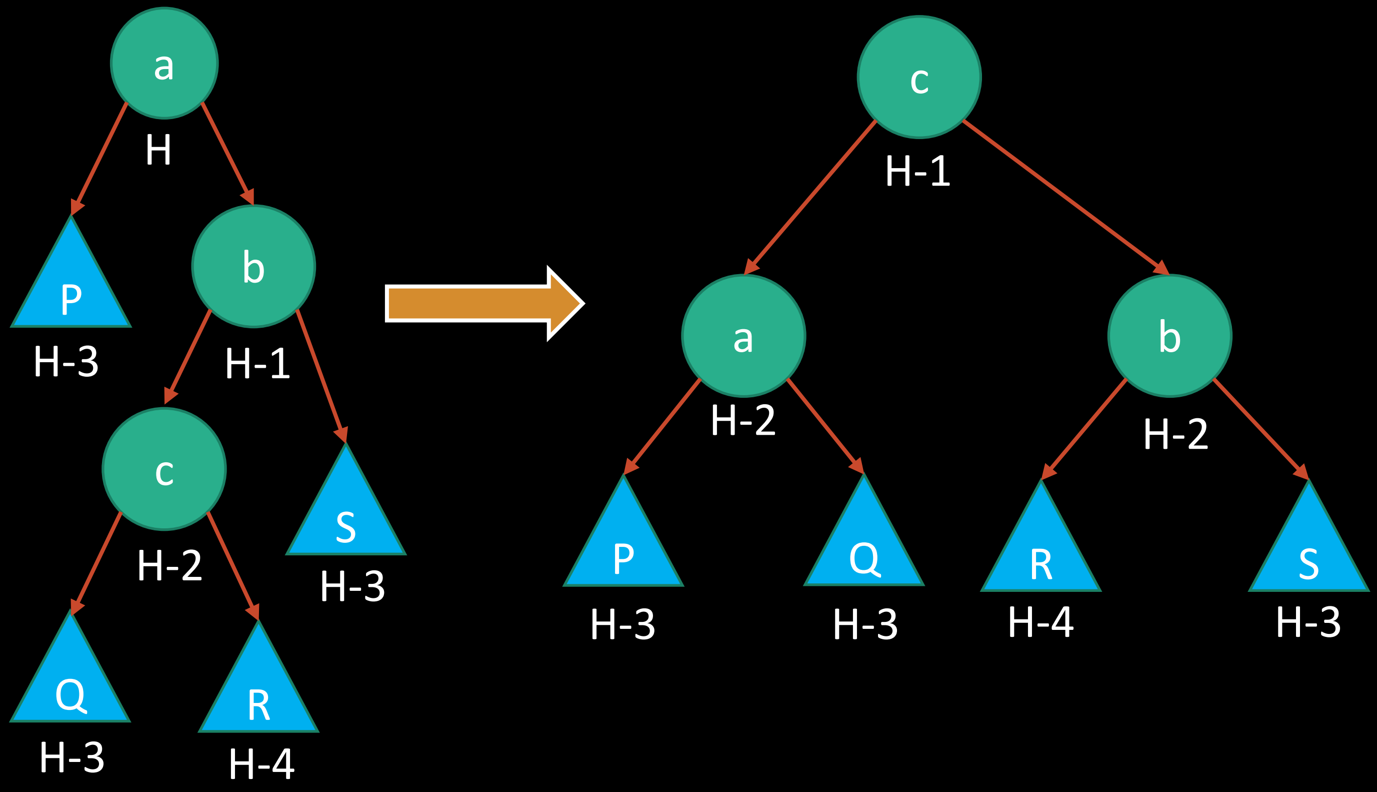



```
def left_rotate(x:Node)->Node:
    y = x.right
    x.right = y.left
    y.left = x
    x.height = max(height(x.left), height(x.right)) + 1
    y.height = max(height(y.left), height(y.right)) + 1
    return y
```

Большой левый поворот

Используется когда:
c.H = S.H + 1 и a.H = c.H + 2
Временная сложность: O(1)

```
def big_left_rotate(x:Node)->Node:
    return left_rotate(right_rotate(x.right))
```



Подвешивание новой вершины

Временная сложность: O(log(n))

Доказательство:

Подвязывание новой вершины происходит в ходе рекурсивного спуска на самые нижние уровни дерева для подвязывания в соответствии правилами BST к подходящей вершине, что занимает по времени O(log(n)). После добавления новой вершины происходит балансировка дерева сложностью O(1).

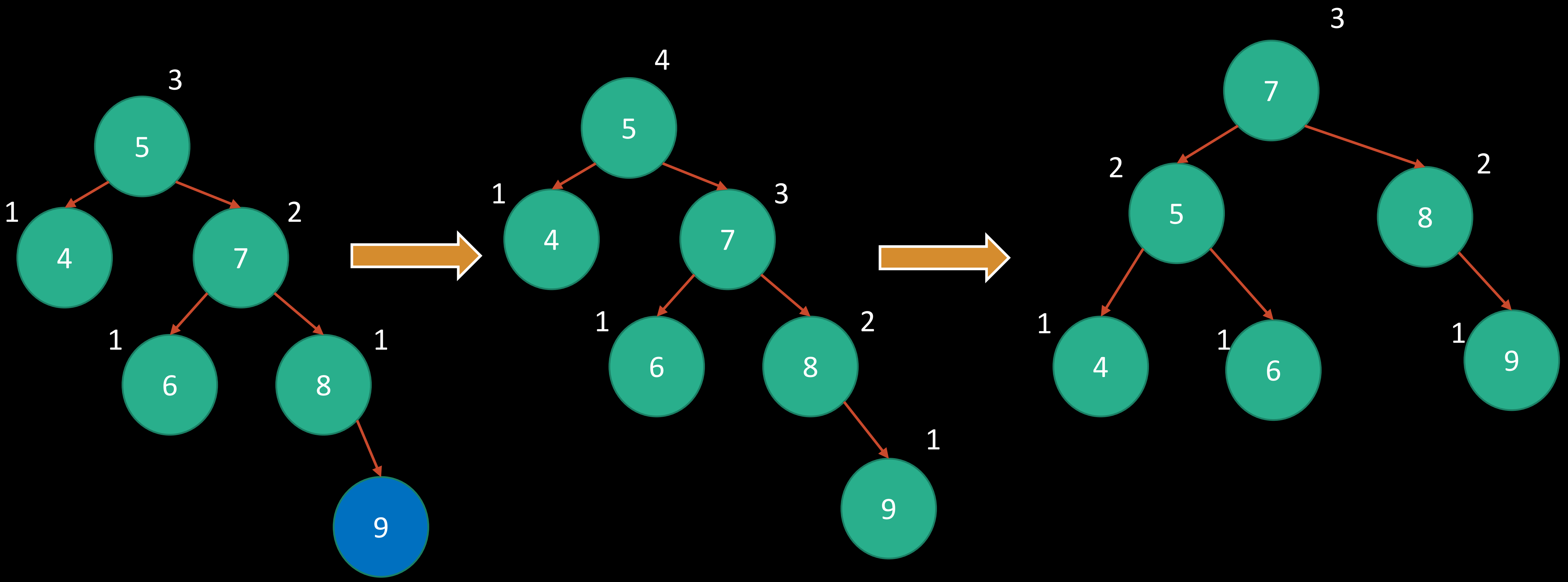
```python
def insert(key:int, node:Node=None)->Node:
    if node is None:
        return Node(key)

    if key < node.key:
        node.left = insert(key, node.left)
    elif key > node.key:
        node.right = insert(key, node.right)
    else:
        return node

    node.height = 1 + max(height(node.left), height(node.right))
    balance = get_balance(node)

    if balance > 1 and key < node.left.key:
        return right_rotate(node)
    if balance < -1 and key > node.right.key:
        return left_rotate(node)
    if balance > 1 and key > node.left.key:
        node.left = left_rotate(node.left)
        return right_rotate(node)
    if balance < -1 and key < node.right.key:
        node.right = right_rotate(node.right)
        return left_rotate(node)
    return node
```

Пример добавления вершины 9



Удаление вершины

Временная сложность: O(log(n))

Доказательство:

Удаление вершины происходит в ходе рекурсивного спуска до самой вершины, что имеет сложность O(log(n)). Далее ищется вершина «преемник», которая максимально приближена к удаляемой по значению, её подвесят вместо удаленной, что займёт O(log(n)). Потом дерево стабилизируется за O(1).

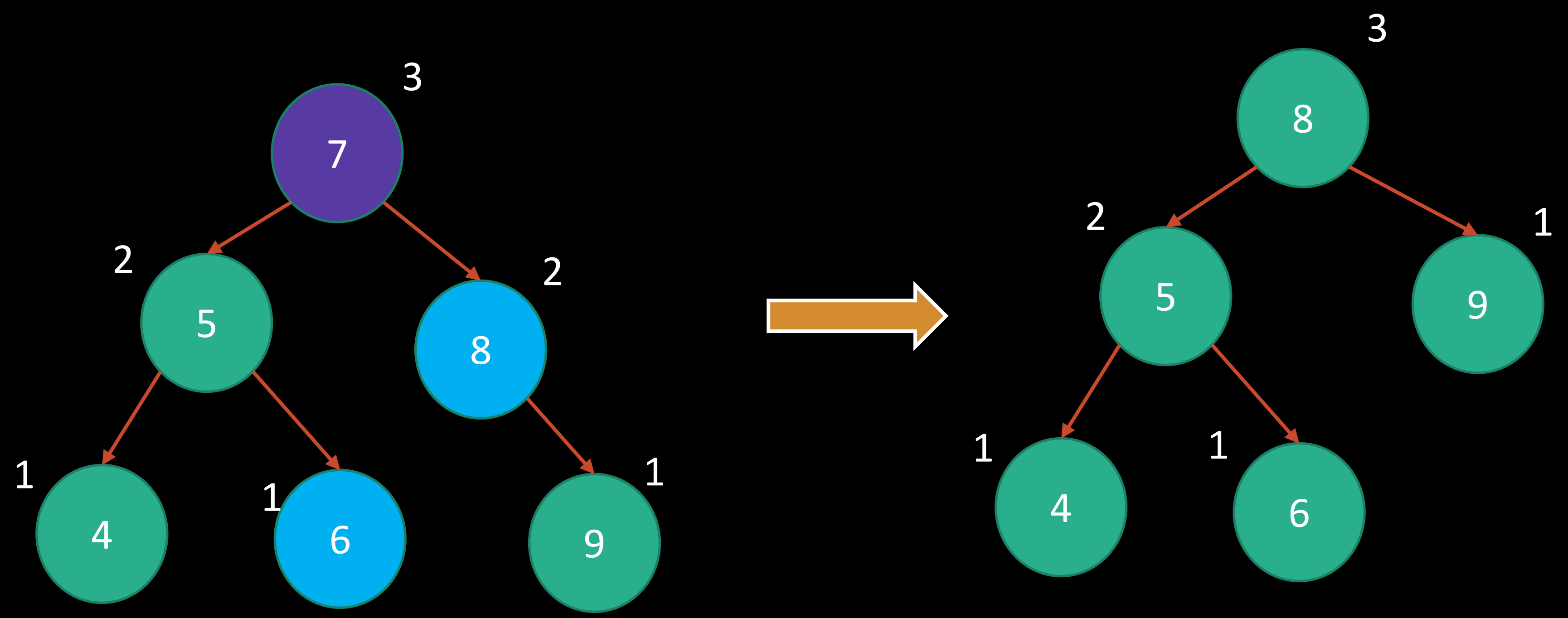
```python
def delete(key:int, node:Node=None)->Node:
    if not node:
        return node
    if key < node.key:
        node.left = delete(key, node.left)
    elif key > node.key:
        node.right = delete(key, node.right)
    else:
        if not node.left:
            return node.right
        elif not node.right:
            return node.left
        temp = get_min_value_node(node.right)
        node.key = temp.key
        node.right = delete(temp.key, node.right)

    node.height = 1 + max(height(node.left), height(node.right))
    balance = get_balance(node)

    if balance > 1 and get_balance(node.left) >= 0:
        return right_rotate(node)
    if balance > 1 and get_balance(node.left) < 0:
        node.left = left_rotate(node.left)
        return right_rotate(node)
    if balance < -1 and get_balance(node.right) <= 0:
        return left_rotate(node)
    if balance < -1 and get_balance(node.right) > 0:
        node.right = right_rotate(node.right)
        return left_rotate(node)
    return node


def get_min_value_node(node:Node)->str:
    current = node
    while current.left is not None:
        current = current.left
    return current
```

Пример удаления вершины 7

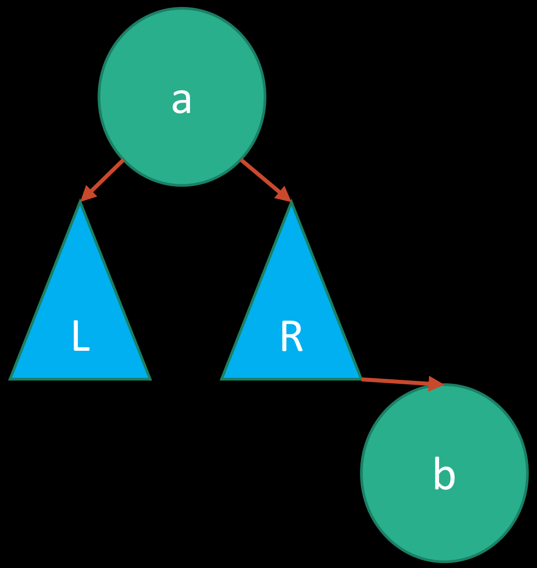


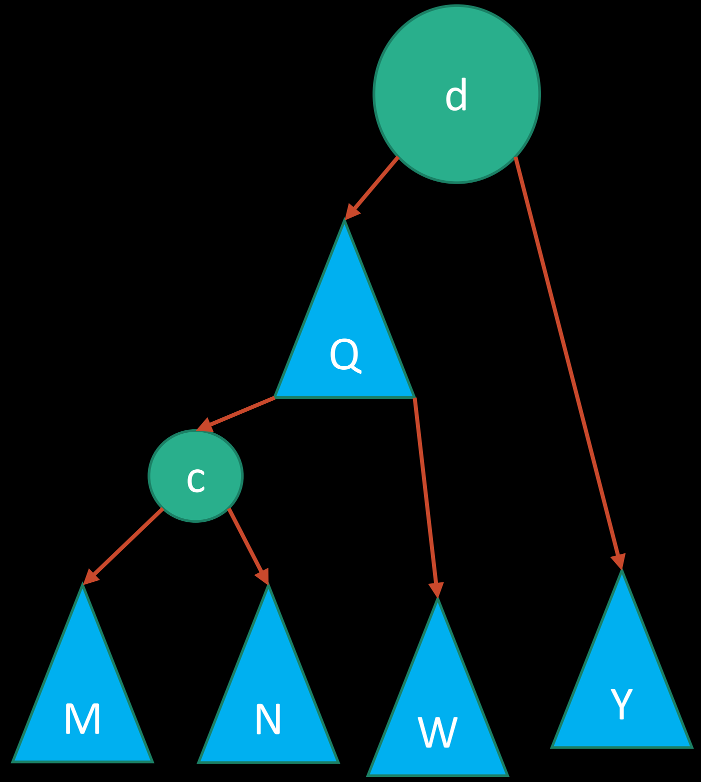
Объединение АВЛ-деревьев

Слияние двух AVL-деревьев: T1 и T2, где T1 имеет меньше ключей и меньшую высоту.

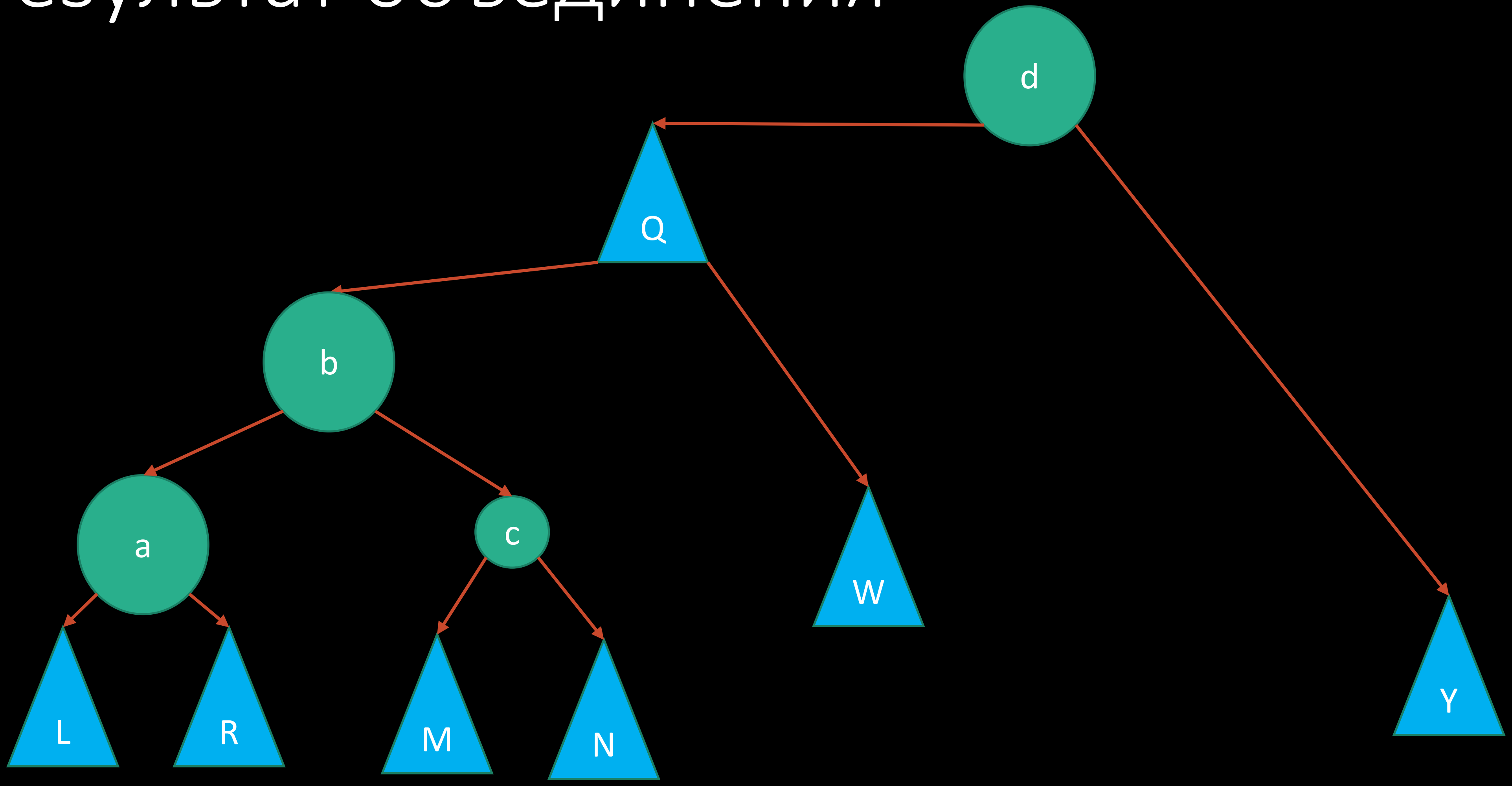
1. В T1 удаляется самая правая вершина b.
2. В T2 идем от корня в левое поддерево, пока высота поддерева не сравняется с высотой T1.
3. Создаем новое дерево S с корнем b, левым поддеревом T1 и правым поддеревом P.
4. В T2 делаем левое поддерево S и запускаем балансировку.

Временная сложность: O(log(len(T1))) * 2 = O(log(n))

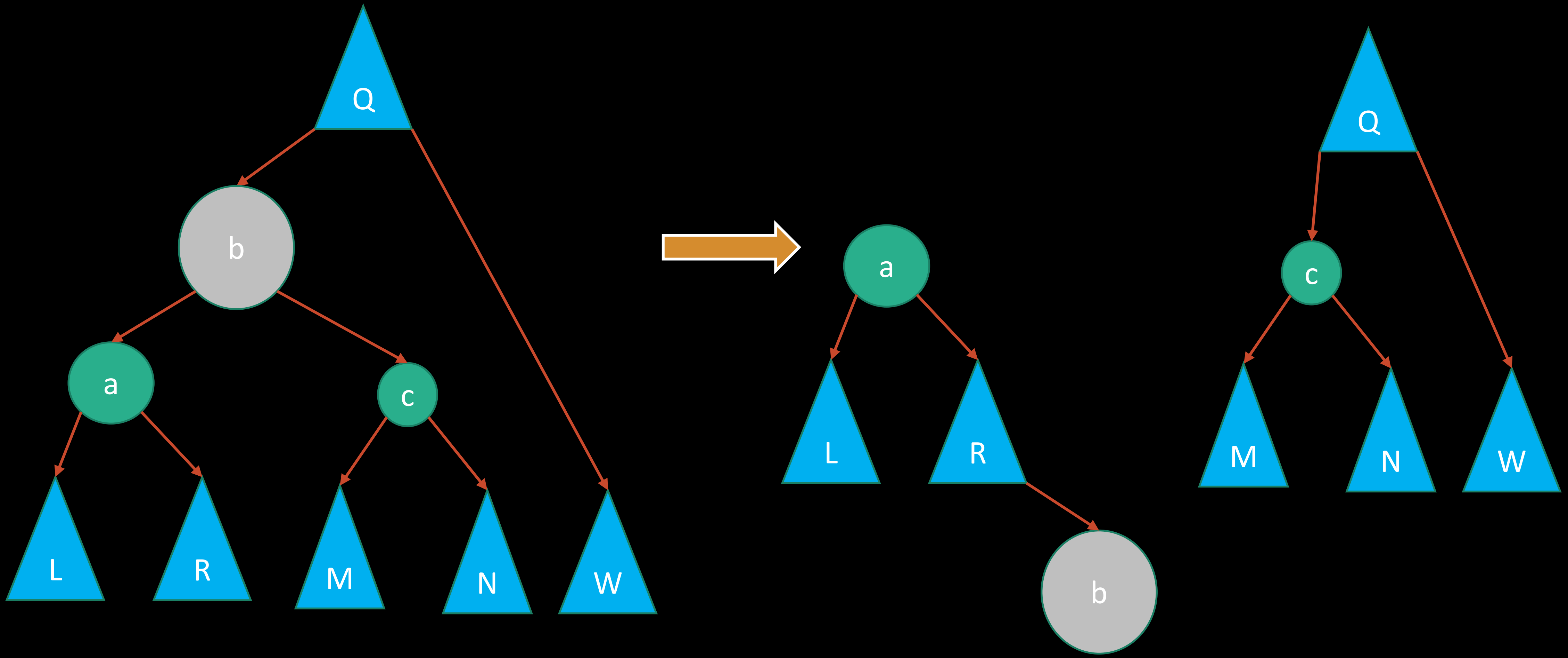




Результат объединения



Разбиение АВЛ-дерева



Тесты

```
tree = insert(5)
tree = insert(4, tree)
tree = insert(6, tree)
tree = insert(7, tree)
tree = insert(8, tree)
print(print_tree_dop(tree))
tree = insert(9, tree)
print(print_tree_dop(tree))
tree = delete(7, tree)
print(print_tree_dop(tree))
print(find_value(7, tree))
print(find_value(5, tree))
```

```
(k:4, h:1)<=(k:5, h:3)=>((k:6, h:1)<=(k:7, h:2)=>(k:8, h:1))
((k:4, h:1)<=(k:5, h:2)=>(k:6, h:1))<=(k:7, h:3)=>((k:8, h:2)=>(k:9, h:1))
((k:4, h:1)<=(k:5, h:2)=>(k:6, h:1))<=(k:8, h:3)=>(k:9, h:1)
False
True
```

Сферы применения АВЛ-деревьев

1. **Базы данных.** AVL-деревья используются для индексации данных в базах данных, что позволяет ускорить поиск информации.
2. **Графические интерфейсы.** В графических интерфейсах пользователя AVL-деревья могут использоваться для организации элементов интерфейса, таких как меню, панели инструментов и окна.
3. **Сжатие данных.** AVL-деревья могут применяться для сжатия данных путём кодирования часто встречающихся последовательностей символов в виде узлов дерева.
4. **Обработка изображений.** В обработке изображений AVL-деревья используются для быстрого доступа к пикселям изображения и выполнения операций над ними.
5. **Компьютерная графика.** В компьютерной графике AVL-деревья применяются для моделирования трёхмерных объектов и сцен.
6. **Криптография.** В криптографии AVL-деревья используются для генерации и проверки цифровых подписей.

Аналоги

1. **Красно-чёрные деревья.** Временная сложность основных операций (поиск, вставка, удаление) в красно-черных деревьях — O(log n), как и в АВЛ-деревьях. Однако красно-чёрные деревья используют меньше памяти для хранения информации о балансе узлов.
2. **B-деревья.** B-деревья обычно имеют более высокую степень ветвления, что приводит к лучшей производительности при работе с большими объёмами данных. Временная сложность операций в B-деревьях также O(log n). Так же более оптимальнее используется память при балансировке. Однако B-деревья могут быть сложнее реализовать и поддерживать.
3. **Splay-деревья.** Splay-деревья адаптируются к частоте доступа к элементам, перемещая часто используемые элементы ближе к корню дерева. Это может привести к улучшению производительности для часто используемых операций, но также может вызвать ухудшение производительности для редко используемых операций. Временная сложность операций в Splay-деревьях зависит от частоты использования элементов и может варьироваться от O(1) для часто используемых элементов до O(n) для редко используемых элементов.
4. **Бинарные деревья поиска.** Бинарные деревья поиска не являются самобалансирующимися, поэтому их временная сложность может варьироваться от O(n) в худшем случае до O(log n) в лучшем случае. АВЛ-деревья обеспечивают лучшую производительность в среднем случае, но требуют больше вычислений для поддержания баланса.

Реализация

Python 3.9

Vs Code