PROG
kiev.ua

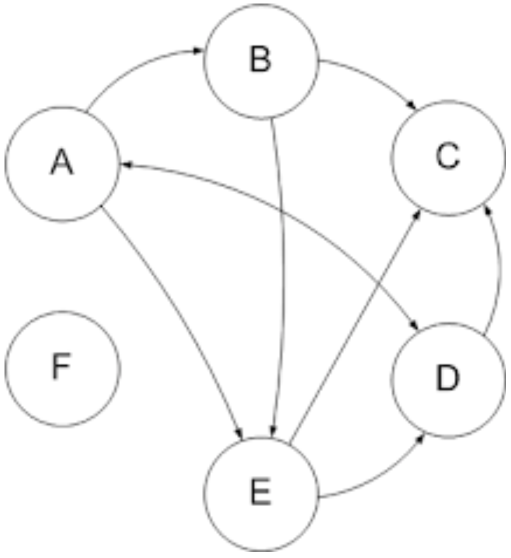
Древовидные структуры

Составитель: Рощупкин Александр

Древовидные структуры данных

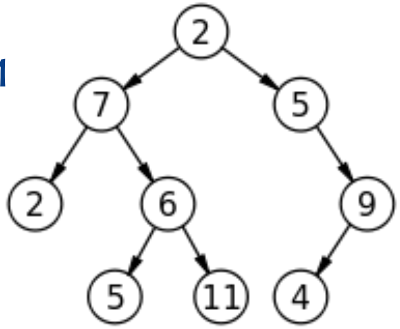
Графы

* Структура данных состоящая из множества вершин и рёбер
* **Вершины** – это основные объекты графа, служащие для хранения данных
* **Рёбра** – это связи между вершинами
* **Дуга** – ориентированное ребро
* Граф с рёбрами – ориентированный
* Цикл – дуга, исходящая и входящая в тот же узел



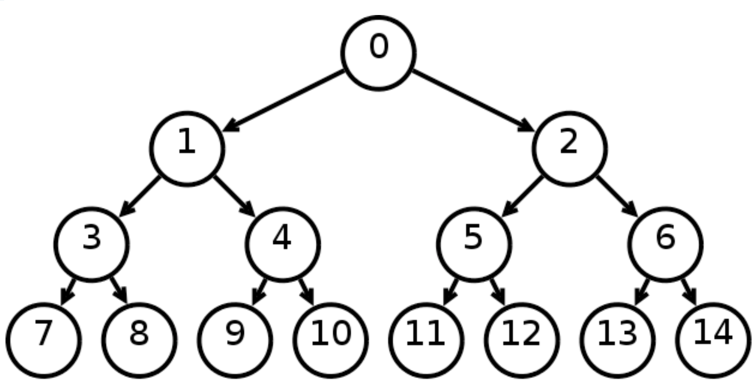
Понятие дерева

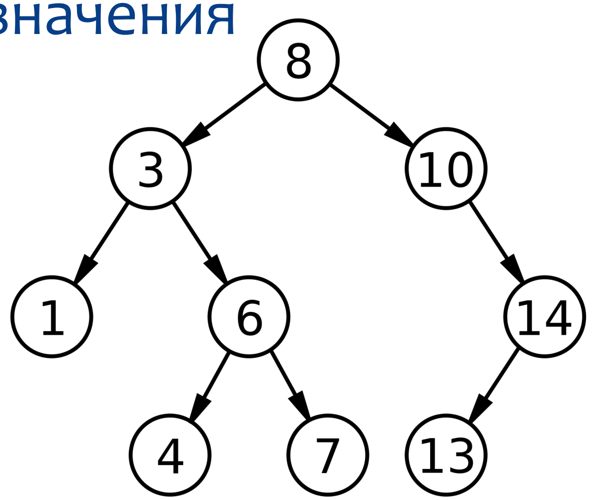
- Дерево – связанный ацикличный граф
- **Связность** означает наличие путей между любой парой вершин
- **Ацикличность** — отсутствие циклов и то, что между парами вершин имеется только по одному пути
- Взвешенное дерево – это дерево, вершинам которого задан определённый вес
- **Корень** – узел, в который не входят дуги
- **Лист** – концевой узел
- **Узел ветвления** – не концевой узел



Бинарное дерево поиска

* Это бинарное дерево, обладающее дополнительными свойствами:
* Значение левого потомка меньше значения родителя,
* Значение правого потомка больше значения родителя для каждого узла дерева





Пример бинарного дерева

- Бинарное дерево представляет собой класс, в котором есть вложенный класс узла дерева
- В классе узла находится две ссылки, на левое и правое поддерево

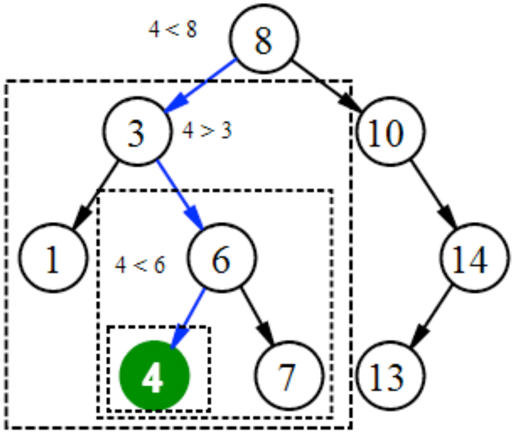
```
public class BinaryTree {

    private Node root;

    private static class Node {
        int value;
        Node left;
        Node right;
    }
}
```

Поиск элемента

* Для каждого узла метод сравнивает значение его ключа с искомым ключом
* Если ключи одинаковы, то метод возвращает текущий узел, в противном случае метод вызывается рекурсивно для левого или правого поддерева
* Узлы, которые посещает функция образуют нисходящий путь от корня, так что время ее работы O(h), где h — высота дерева



Поиск максимума и минимума

* Чтобы найти минимальный элемент в бинарном дереве поиска, необходимо просто следовать указателям left от корня дерева, пока не встретится значение null
* Если у вершины есть левое поддерево, то по свойству бинарного дерева поиска в нем хранятся все элементы с меньшим ключом
* Если его нет, значит эта вершина и есть минимальная
* Максимальный элемент находится движением вправо

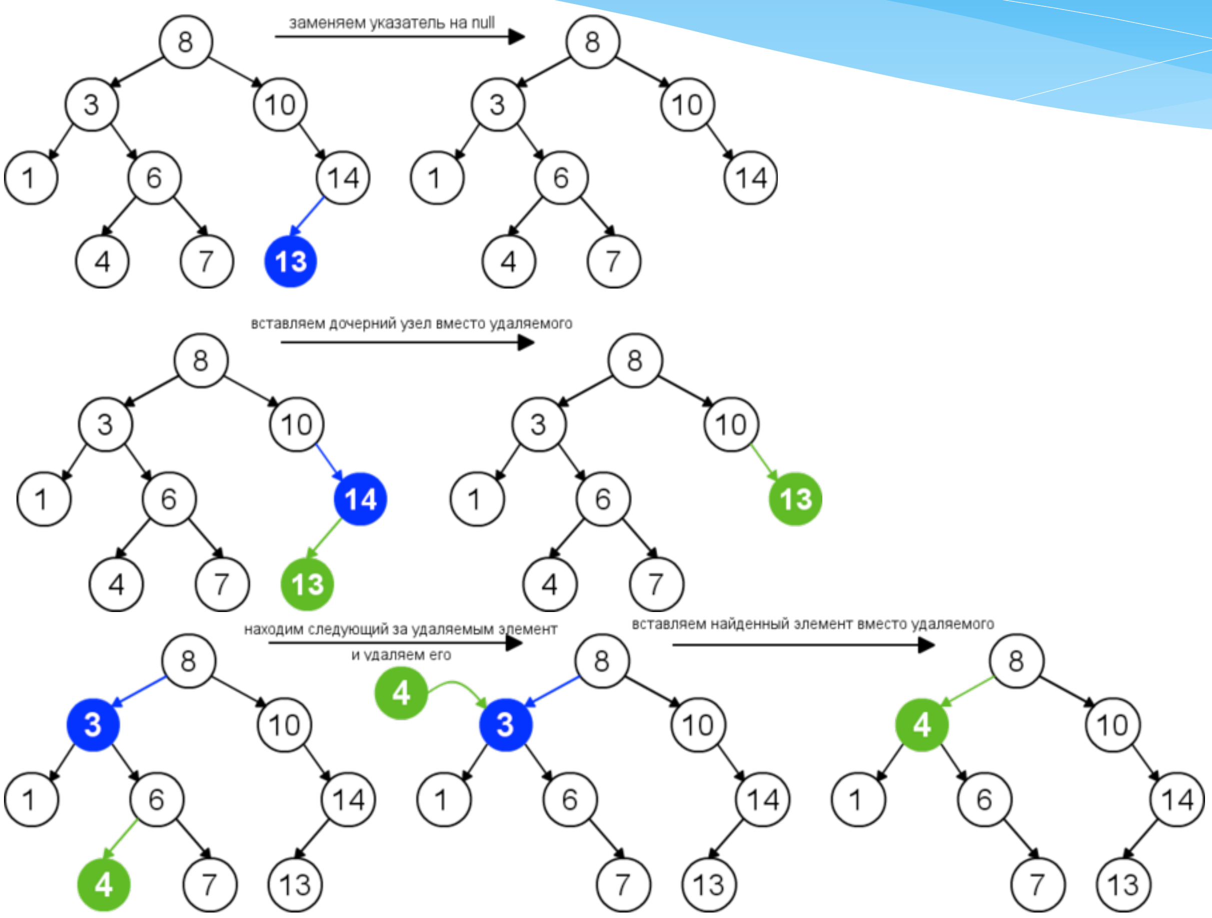
Добавление нового узла

* Для каждого узла метод сравнивает значение его ключа с искомым ключом
* Если нужно идти вправо и место правого потомка пусто, то вставляем элемент и выходим, иначе спускаемся на правый элемент
* Если нужно идти влево и место левого потомка пусто, то вставляем элемент и выходим, иначе спускаемся на левый элемент

Удаление узла

- При удалении узла есть 3 основных случая:
  - У узла нет дочерних узлов, то у его родителя нужно просто заменить указатель на null
  - Если у узла есть только один дочерний узел, то нужно создать новую связь между родителем удаляемого узла и его дочерним узлом
  - Если у узла два дочерних узла, то нужно найти следующий за ним элемент (у этого элемента не будет левого потомка), его правого потомка подвесить на место найденного элемента, а удаляемый узел заменить найденным узлом

Удаление узла



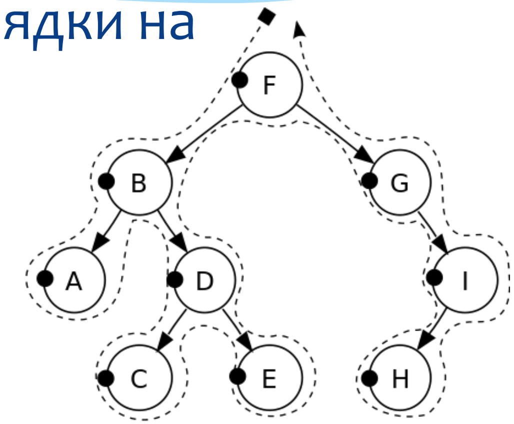
Пример удаление узла

```
public TreeNode deleteNode(TreeNode root, int value) {
    if (root == null)
        return null;
    if (root.data > value) {
        root.left = deleteNode(root.left, value);
    } else if (root.data < value) {
        root.right = deleteNode(root.right, value);
    } else {
        // у узла оба потомков
        if (root.left != null && root.right != null) {
            TreeNode temp = root;
            // находим минимальный элемент (самый левый) у правого потомка
            TreeNode minNodeForRight = minimumElement(temp.right);
            // заменяем текущий элемент минимальным элементом в правом поддереве
            root.data = minNodeForRight.data;
            // удаляем минимальный элемент из правого поддерева
            deleteNode(root.right, minNodeForRight.data);
        }
        // у узла только левый потомок
        else if (root.left != null) {
            root = root.left;
        }
        // у узла только правый потомок
        else if (root.right != null) {
            root = root.right;
        }
        // у узла нет потомков
        else
            root = null;
    }
    return root;
}
```

```
public TreeNode minimumElement(TreeNode root) {
    if (root.left == null)
        return root;
    else {
        return minimumElement(root.left);
    }
}
```

Обход в глубину

* **Обход в глубину** - идет из начальной вершины, посещая еще не посещенные вершины без оглядки на удаленность от начальной вершины



  * Прямой

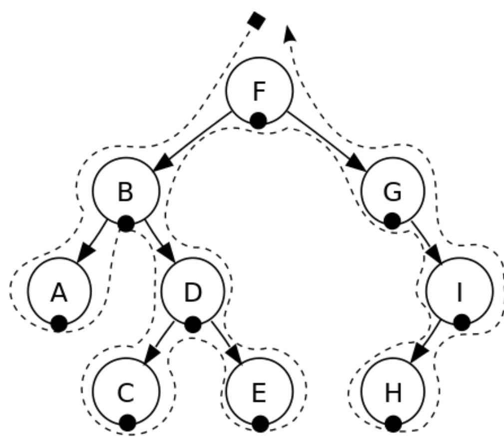
    (F, B, A, D, C, E, G, I, H)

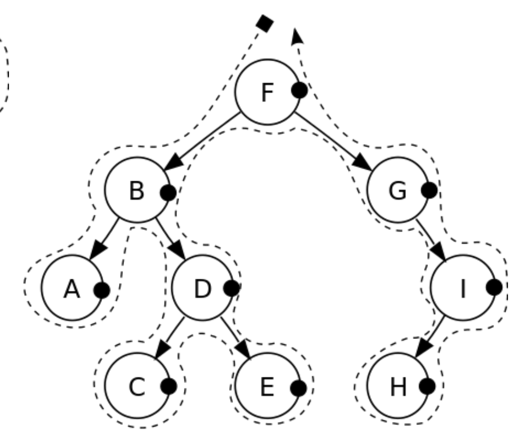
  * Центральный

    (A, B, C, D, E, F, G, H, I)

  * Обратный

    (A, C, E, D, B, H, I, G, F)





Примеры обход в глубину

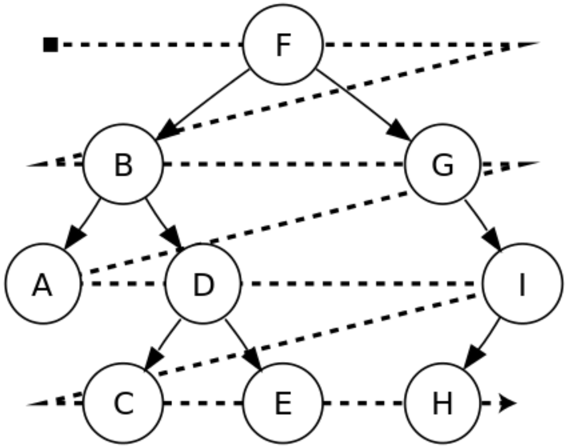
```java
public static void inOrder(TreeNode root) {
    if (root == null) {
        return;
    }
    inOrder(root.left);
    System.out.print(root.data + " ");
    inOrder(root.right);
}

public static void preOrder(TreeNode root) {
    if (root == null) {
        return;
    }
    System.out.print(root.data + " ");
    inOrder(root.left);
    inOrder(root.right);
}
```

```java
public static void postOrder(TreeNode root) {
    if (root == null) {
        return;
    }
    inOrder(root.left);
    inOrder(root.right);
    System.out.print(root.data + " ");
}
```

Обход дерева в ширину

- Обход всех вершин дерева можно совершать несколькими способами:
- В ширину и в глубину
  - **Обход в ширину** - посещаем каждый узел на уровне прежде чем перейти на следующий уровень

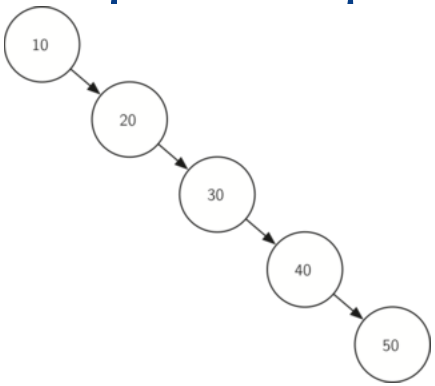


Пример обхода дерева в ширину

```java
private String across(TreeNode node) {
    StringBuilder sb = new StringBuilder();
    Queue<TreeNode> queue = new LinkedList<>(); // создать новую очередь
    queue.offer(node); // поместить в очередь первый уровень
    while (queue.size() != 0) { // пока очередь не пуста
        //если у текущей ветви есть листья, их тоже добавить в очередь
        final TreeNode current = queue.peek();
        if (current.left != null) {
            queue.offer(current.left);
        }
        if (current.right != null) {
            queue.offer(current.right);
        }
        //извлечь из очереди последний элемент
        sb.append(current.data + " ");
        queue.poll();
    }
    return sb.toString();
}
```

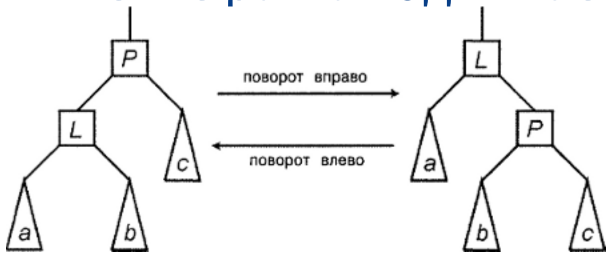
Вырожденное бинарное дерева

* Всегда желательно, чтобы все пути в дереве от корня до листьев имели примерно одинаковую длину, то есть чтобы глубина и левого, и правого поддеревьев была примерно одинакова в любом узле
* В противном случае теряется производительность



Баланс бинарного дерева

* Для балансировки дерева применяется операция «поворот дерева», при этом повышается ранг того узла, который становится выше, те повышение ранга поднимает узел на один уровень вверх



* Левый дочерний узел в позицию его родительского узла, правое дочернее дерево становится новым левым дочерним поддеревом родительского узла
* Для повышения ранга на 2 уровня используются спаренный двусторонний поворот

Домашнее задание

* Сделать поиск, вставку, удаление элемента в бинарное дерево поиска
* Посчитать количество вершин в дереве
* Определить высоту двоичного дерева
* *Определить, является ли заданное двоичное дерево деревом поиска*
* *Сбалансировать двоичное дерево поиска*

Балансирующиеся деревья

- **AVL дерево**

https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

- **Красно-чёрное дерево**

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

AVL дерево

* АВЛ-дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1
* АВЛ — аббревиатура, образованная первыми буквами фамилий создателей Адельсон-Вельского Георгия Максимовича и Ландиса Евгения Михайловича
* **Фактор баланса** – (или коэффициент сбалансированности узла) разница в высоте между правым и левым поддеревом

Пример АВЛ дерева

* В узел добавляется дополнительное поле – высота поддерева

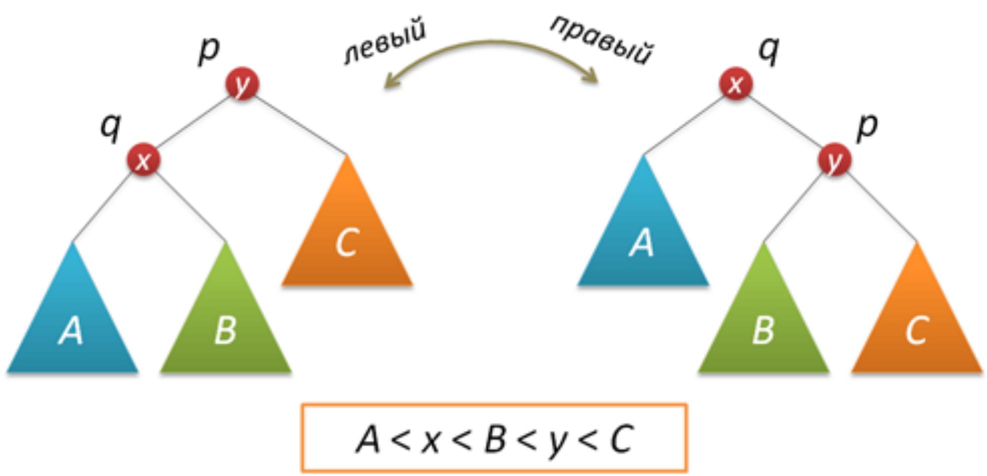
```
public class BinaryTree {

    private Node root;

    private static class Node {
        int value;
        int height;
        Node left;
        Node right;
    }
}
```

Балансировка дерева

* В процессе добавления или удаления узлов в АВЛ-дереве возможно возникновение ситуации, когда balance factor некоторых узлов оказывается равными 2 или -2, т.е. возникает *расбалансировка* поддерева. необходимо балансировать поворотами

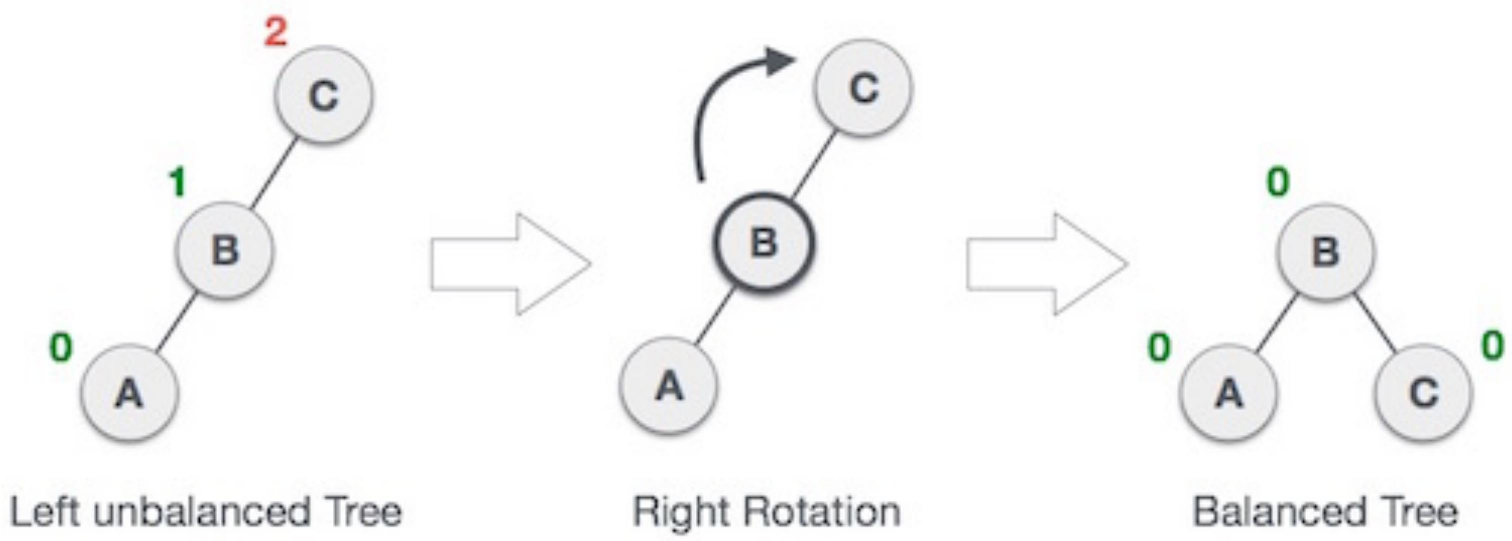


Типы поворотов

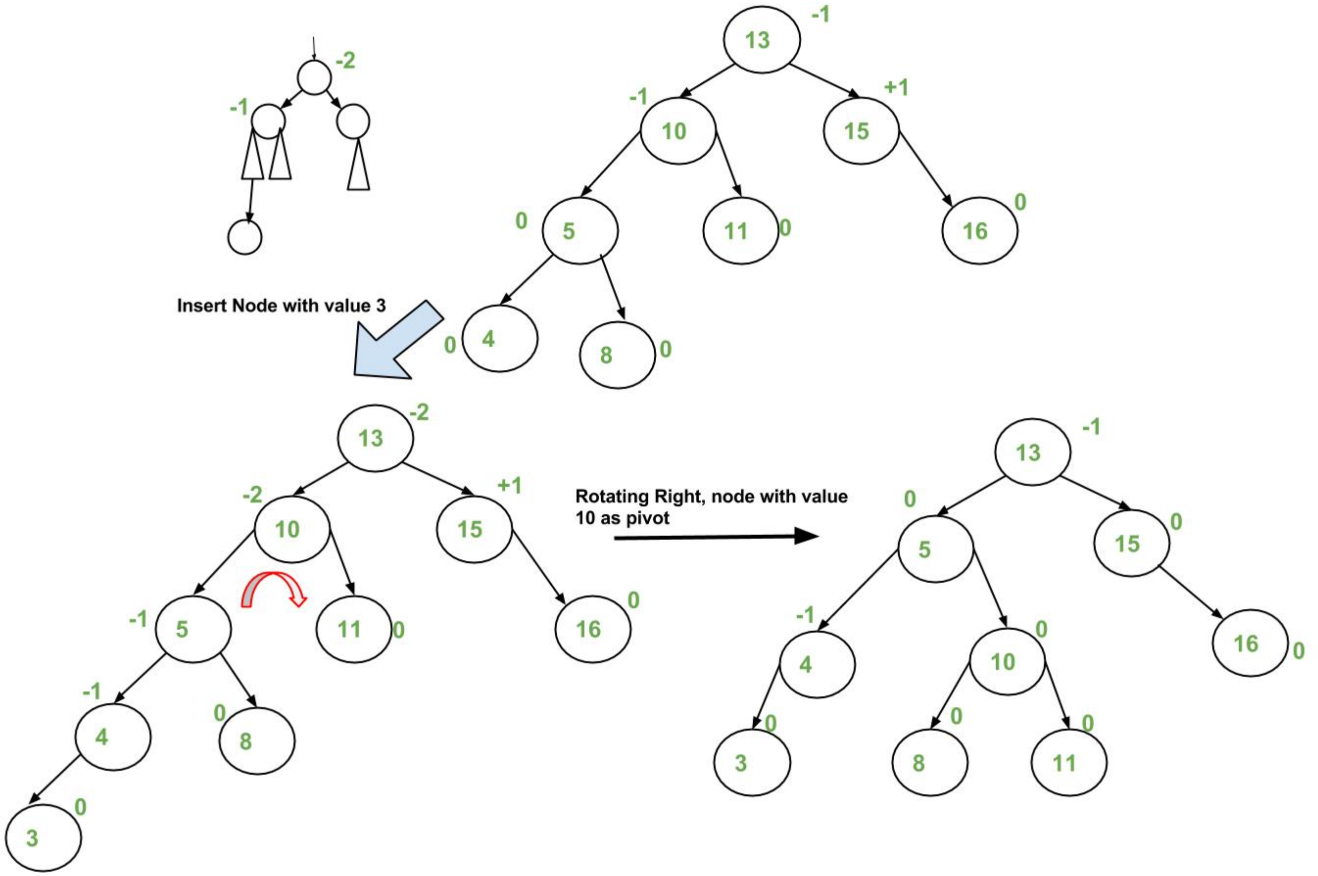
* Одиночный правый поворот
* Одиночный левый поворот
  Двойной лево-правый поворот
* Двойной право-левый поворот

Правый поворот

- В левое поддерево добавили элемент А
- Необходимо увеличить высоту правого поддерева

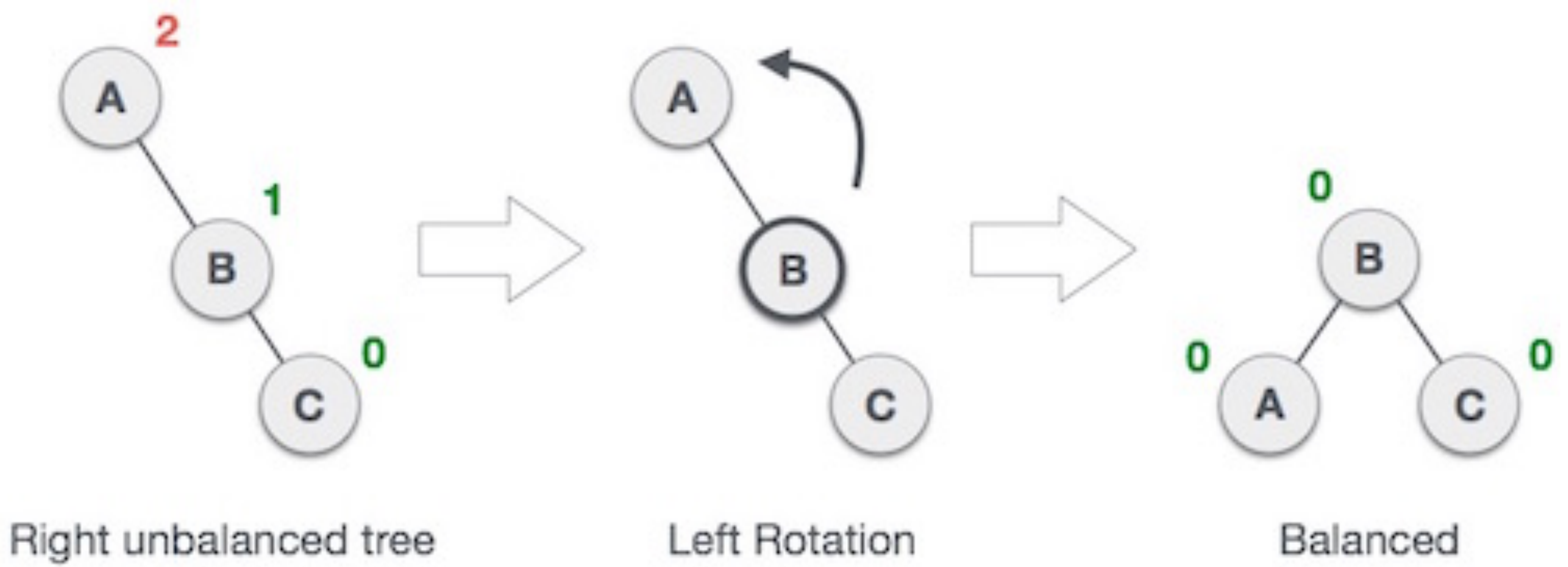


Правый поворот

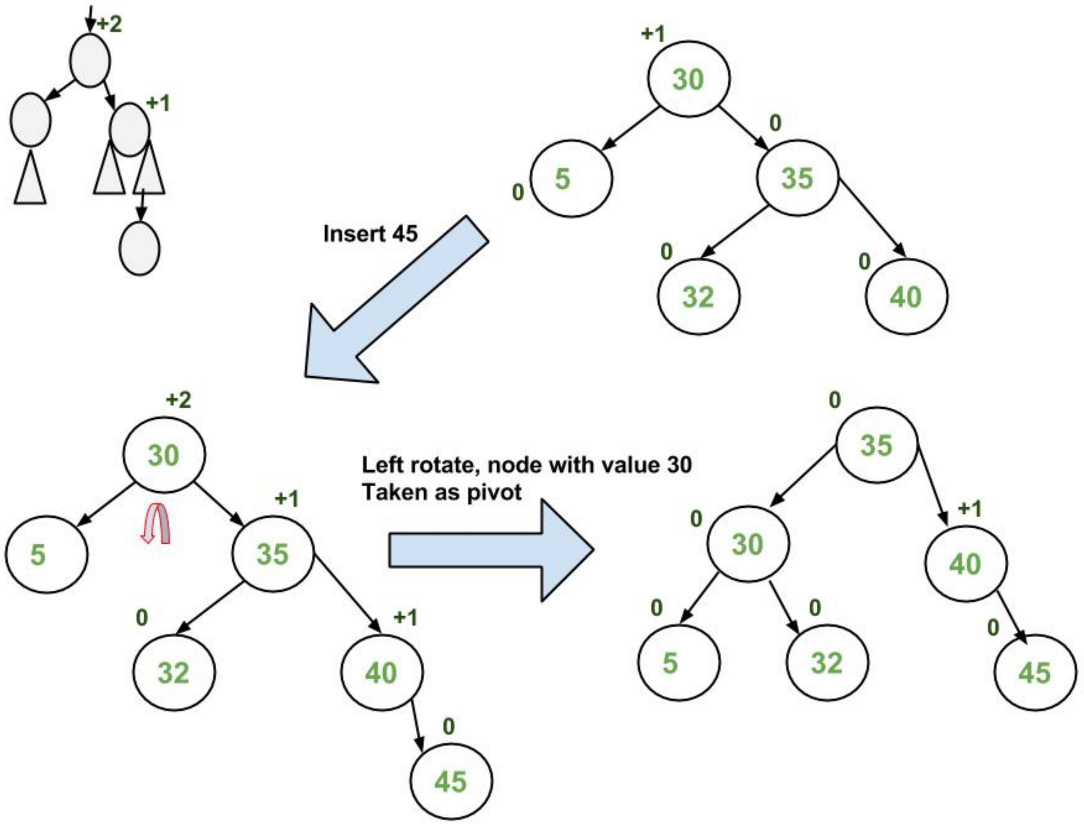


Левый поворот

* В правое поддерево добавили элемент C
* Необходимо увеличить высоту левого поддерева

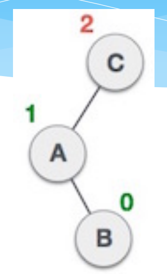


Левый поворот

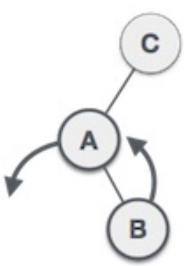


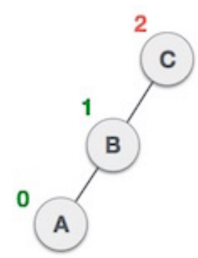
Лево-правый поворот

* Поворот выполняется после добавления элемента в правое поддерево левого дочернего узла дерева

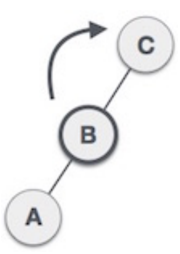


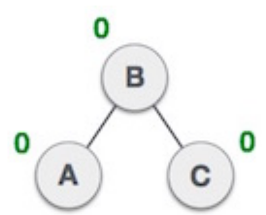
* Сначала делаем левый поворот левого поддерева, готовим к правому повороту





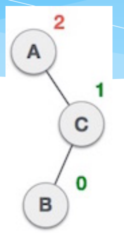
* Выполняем правый поворот, делаем B новым корнем



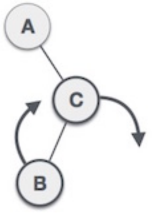


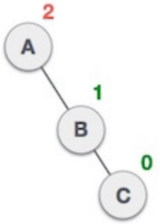
Право-левый поворот

* Выполняется после добавления элемента в левое поддерево правого дочернего узла дерева

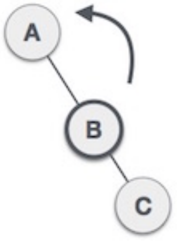


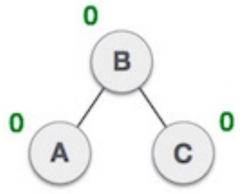
* Делаем подготовительный правый поворот правого поддерева



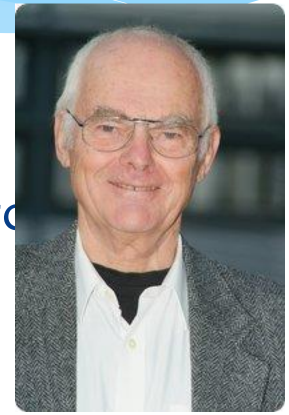


* Выполняем левый поворот, делаем B корнем поддерева



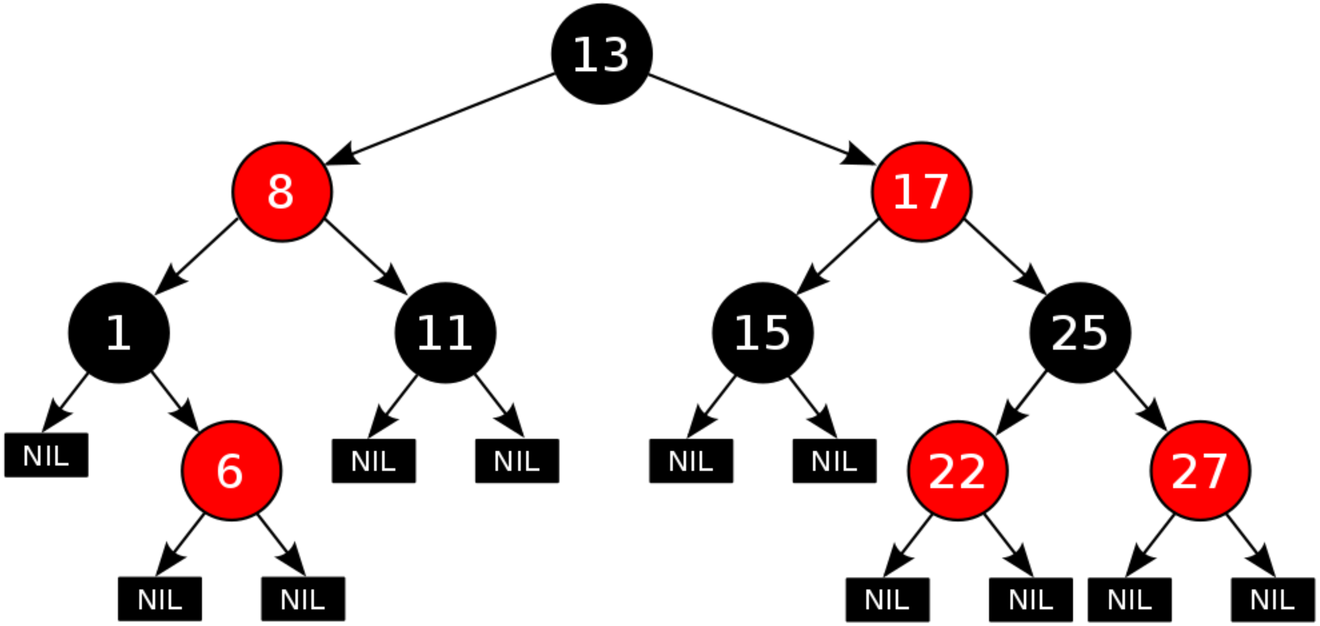


Красно-чёрное дерево



- Автор Rudolf Bayer
  Technical University of Munich, Germany, 1972
- Это бинарное дерево поиска, для которого выполняются красно-черные свойства:
  - Каждый узел дерева является либо красным (red), либо черным (black)
  - Корень дерева является черным узлом
  - Каждый лист дерева (NULL) является черным узлом
  - У красного узла оба дочерних узла – черные
  - У любого узла все пути от него до листьев, являющихся его потомками, содержат одинаковое количество черных узлов

Пример красно-чёрного дерева



Нарушение свойств дерева

* **Добавляемый узел всегда красный**
* После добавления нового элемента свойства 2 и 4 могут быть нарушены
* Корень дерева является черным – может быть нарушено (например, при добавление первого элемента)
* У красного узла оба дочерних узла являются черными – может быть нарушено
* Возможно 6 случаев, нарушающих свойства красно-черного дерева (3 случая симметричны другим трем)
* Восстановление свойств начинаем с нового элемента и продвигаемся вверх к корню дерева

Восстановление свойств

* Случай 1
* Дядя только что добавленного узла – красный
* Выравниваем баланс перекрашиванием:
  * Родителя в чёрный
  * Дядю в чёрный
  * Дедушку в красный

Восстановление свойств

* Случай 2
* Дядя чёрный и добавляемый узел является правым потомком своего родителя
* Выравниваем баланс поворотом дерева влево

Восстановление свойств

* Случай 3
* Дядя чёрный и добавляемый узел – левый потомок своего предка
* Восстанавливаем баланс перекрашиванием родителя в чёрный и дедушку в красный. Затем поворачиваем дерево с дедушки вправо