TREE

HARD SKILL

Деревья. Основные понятия.

- **Дерево<tree>** - структура данных с узлами, где каждый узел имеет ноль или более дочерних узлов.
- **Узел** - элемент дерева, содержащий данные и ссылки на дочерние узлы.
- **Корень<root>** - верхний узел дерева, от которого начинается дерево.
- **Листья** - узлы без дочерних узлов.
- **Родитель** - узел, от которого исходят ветви к другим узлам.
- **Высота** - максимальное количество уровней в дереве.
- **Глубина** - количество ребер от корня до узла.
- **Поддерево** - узел со всеми его потомками.

Базовая структура узла дерева.

```
struct TreeNode {
    int val;              // Значение узла
    TreeNode* left;       // Указатель на левого потомка
    TreeNode* right;      // Указатель на правого потомка

    // Конструктор для узла с заданным значением
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
```

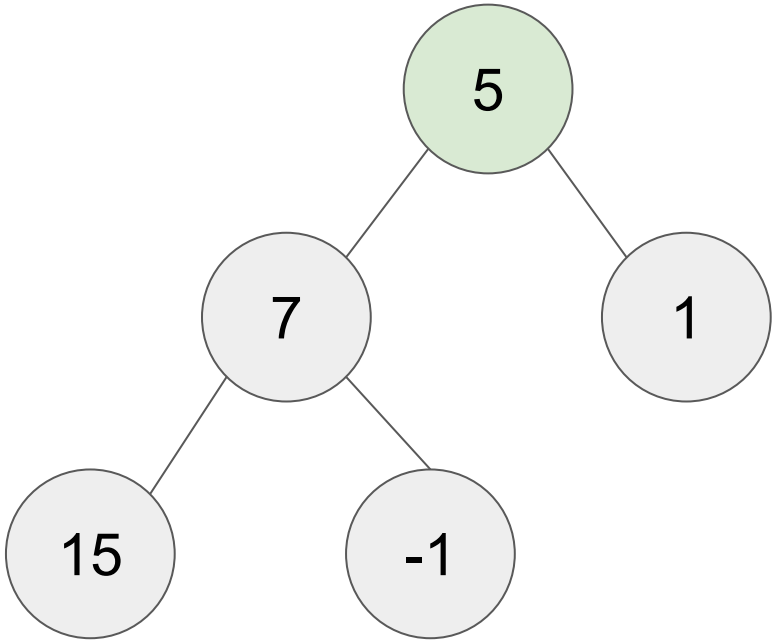
Виды деревьев

1. Binary Tree (BT)
2. Binary Search Tree (BST)
3. AVL-tree
4. Red-Black tree
5. B-tree
6. Splay-tree

и другие

Binary Tree.

Двоичное дерево - это структура данных, в которой каждый узел имеет не более двух дочерних узлов.



<base class binary-tree>

```
class BinaryTree {
private:
    struct TreeNode {
        ...
    };
    TreeNode* root;

public:
    BinaryTree() : root(nullptr) {}
    ~BinaryTree() {}
};
```

<insert node>

```
public:
    void insert(int value) {
        root = _insert(root, value);
    }
private:
    TreeNode* _insert(TreeNode* node, int value) {
        if (node == nullptr) return new TreeNode(value);
        if (node->left == nullptr) {
            node->left = new TreeNode(value);
        } else if (node->right == nullptr) {
            node->right = new TreeNode(value);
        } else {
            if (getHeight(node->left) <= getHeight(node->right))
                node->left = _insert(node->left, value);
            else
                node->right = _insert(node->right, value);
        } return node;
    }
```

<method GET-HEIGHT>

1. Берем узел.
2. Если узел пустой (*не существует*), его высота равна 0.
3. Рекурсивно вычислите высоту левого поддерева узла и сохраните результат.
4. Рекурсивно вычислите высоту правого поддерева узла и сохраните результат.
5. Верните 1 плюс максимальное значение высоты из левого и правого поддеревьев. Это определит высоту текущего узла.
6. Продолжайте этот процесс до листовых узлов, а затем верните значения высоты обратно к корню дерева.

```
int getHeight(TreeNode* node) {
    if (node == nullptr) return 0;
    // TODO: leftHeight, rightHeight;
    return 1 + std::max(leftHeight,
rightHeight);
}
```

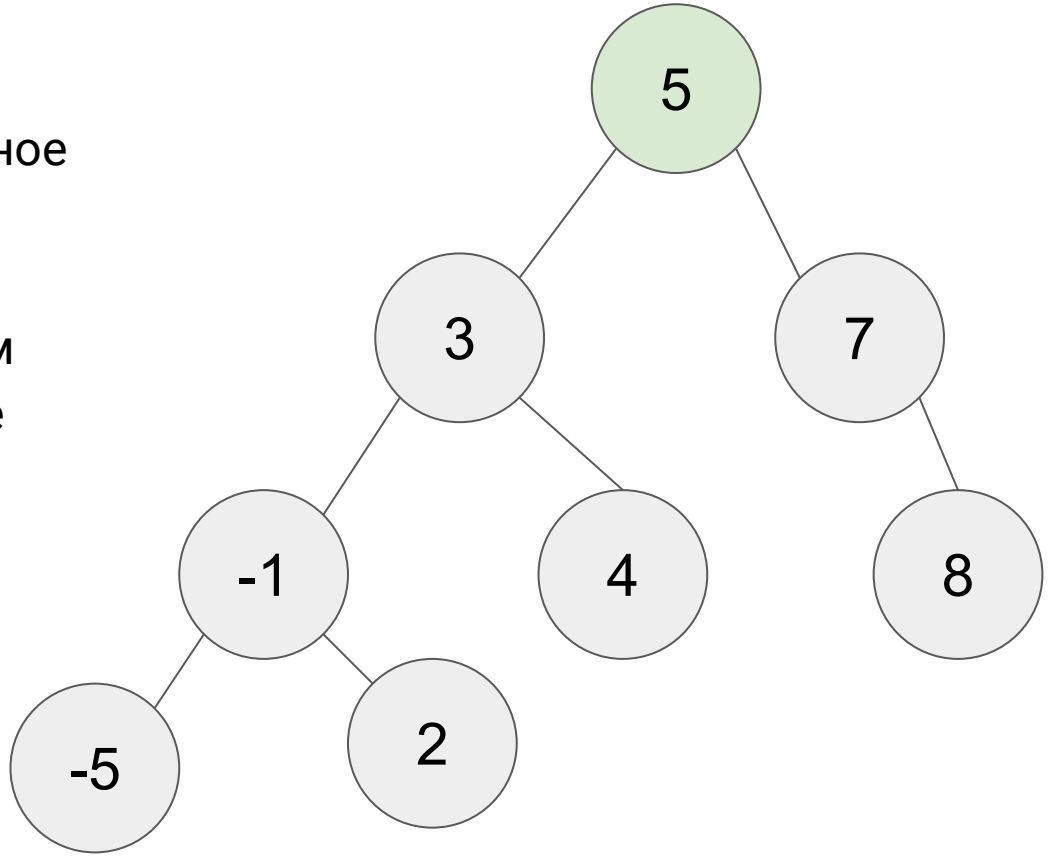
<print nodes>

```
public:
    void print() {
        _print(root);
    }

private:
    void _print(TreeNode* node) {
        if (node == nullptr) return;
        // вывод значение текущего узла
        std::cout << node->val << " ";
        // обходим левое поддерево
        _print(node->left);
        // обходим правое поддерево
        _print(node->right);
    }
```

Binary Search Tree.

Двоичное дерево поиска - двоичное дерево, где узлы упорядочены таким образом, что для каждого узла значения всех узлов в левом поддереве меньше, чем значение узла, и значения всех узлов в правом поддереве больше, чем значение узла.



<base class bst>

```
class BinarySearchTree {
private:
    struct TreeNode {
        ...
    };
    TreeNode* root;

public:
    BinarySearchTree() : root( nullptr) {}
    ~BinarySearchTree() {}
};
```

<insert node - private>

```
TreeNode* _insert(TreeNode* node, int value) {
    if (node == nullptr) return new TreeNode(value);
    if (value < node->val)
        node->left = _insert(node->left, value);
    else
        node->right = _insert(node->right, value);
    return node;
}
```

Remove Node Алгоритм

1. Берем root tree.
2. Если дерево пустое, завершите операцию удаления.
3. Иначе, если значение узла для удаления меньше значения текущего узла, перейдите к левому поддереву.
4. Иначе, если значение узла для удаления больше значения текущего узла, перейдите к правому поддереву.
5. Если значение узла для удаления равно значению текущего узла:
    - Если у текущего узла нет левого поддерева, просто замените его правым поддеревом (и наоборот).
    - Если у текущего узла есть оба поддерева, найдите наименьший узел в правом поддереве (или наибольший узел в левом поддереве), замените значение текущего узла значением найденного узла, а затем удалите найденный узел.
6. Повторяйте шаги 3-5 до тех пор, пока не найдете узел для удаления или не дойдете до конца дерева.

<remove node - private> part 1

```
TreeNode* _remove(TreeNode* node, int value) {
   if (node == nullptr) return nullptr;
   if (value < node->val) node->left = _remove(node->left, value);
   else if (value > node->val) node->right = _remove(node->right, value);
   else {
       // Узел для удаления найден
       // 1) если у узла нет дочерних узлов
       if (node->left == nullptr && node->right == nullptr) {
           delete node;
           return nullptr;
       }
       // 2) у узла есть только один дочерний узел
       if (node->left == nullptr) {
           TreeNode* temp = node->right;
           delete node;
           return temp;
       } ...
```

<remove node - private> part 2

```
}
        // 3) у узла есть оба дочерних узла
        TreeNode* temp = findMin(node->right);
        node->val = temp->val;
        node->right = _remove(node->right, temp->val);
    }
    return node;
}
```

<findMin>

```
TreeNode* findMin(TreeNode* node) {
        while (node->left != nullptr)
            node = node->left;
        return node;
}
```

<findMax>

```
TreeNode* findMax(TreeNode* node) {
    if (node == nullptr) return nullptr;
    while (node->right != nullptr)
        node = node->right;
    return node;
}
```

<search element>

```
private:
    TreeNode* _search(TreeNode* node, int value) {
        if (node == nullptr || node->val == value) return node;
        if (value < node->val)
            return _search(node->left, value);
        else
            return _search(node->right, value);
    }
```

Понятие балансировки.

Балансировка в деревьях - это процесс поддержания равномерного распределения узлов или ограничения высоты дерева, обеспечивающий эффективность операций добавления, удаления и поиска данных.

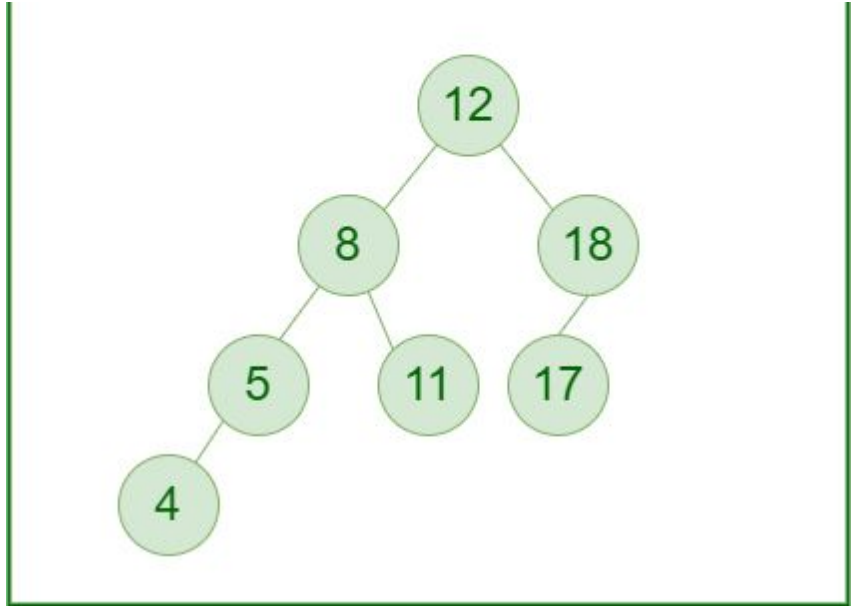
используют для: AVL tree, R/B tree, B tree, Splay-tree

Алгоритм балансировки дерева.

1. Рекурсивно вычислить высоту каждого поддерева, начиная с корня.
2. Для каждого узла вычислить разницу между высотами его левого и правого поддеревьев.
3. Проверить разницу на соответствие заданному критерию баланса (например, -1, 0 или 1 для AVL-деревьев).
4. Если критерий не выполнен, произвести балансировку дерева в соответствии с его типом (например, повороты для AVL-деревьев или перекрашивание узлов для красно-черных деревьев).

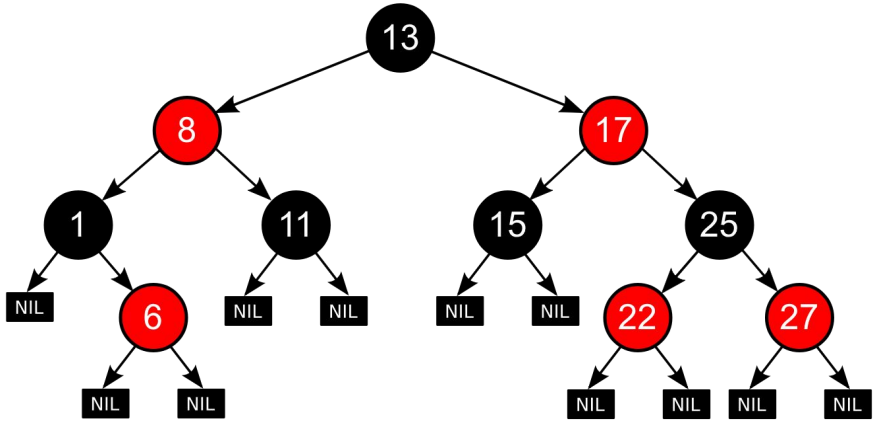
AVL tree

AVL-дерево - это сбалансированное двоичное дерево поиска, в котором разница между высотами левого и правого поддеревьев для каждого узла ограничена одним. Это обеспечивает быстрый поиск и вставку, так как высота дерева ограничена.



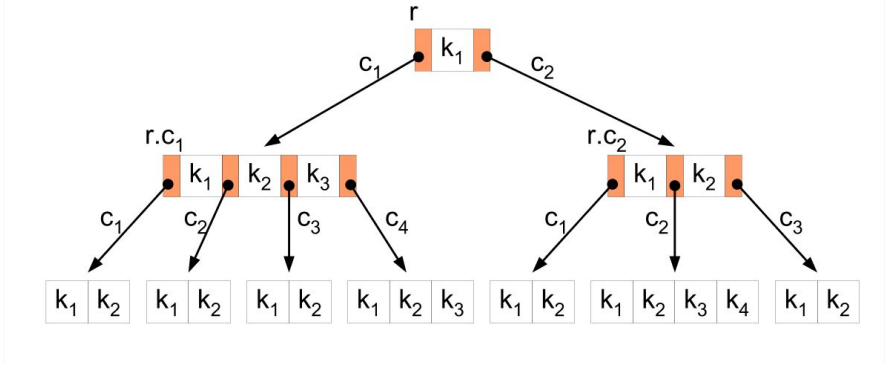
Red-Black tree

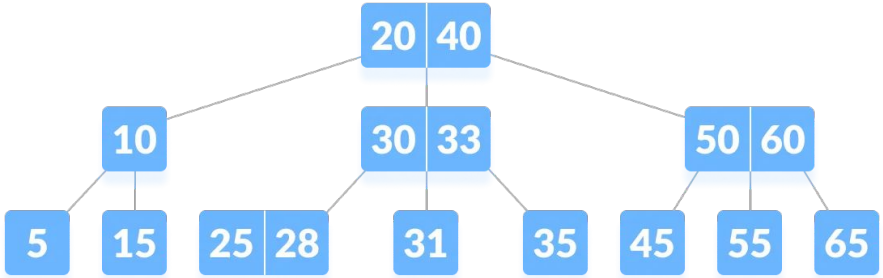
Красно-чёрное дерево - это сбалансированное двоичное дерево поиска, в котором каждый узел имеет ассоциированный с ним цвет (красный или черный), и выполняются определенные правила балансировки, гарантирующие ограничение высоты дерева.



B tree

B-дерево - это сбалансированное дерево, специально предназначенное для работы с большими объемами данных, часто используется в базах данных и файловых системах. Оно позволяет эффективно выполнять операции вставки, удаления и поиска блоков данных.





Splay tree

Splay-дерево - это самобалансирующееся двоичное дерево поиска, где после каждой операции узел перемещается в корень для улучшения быстродействия.