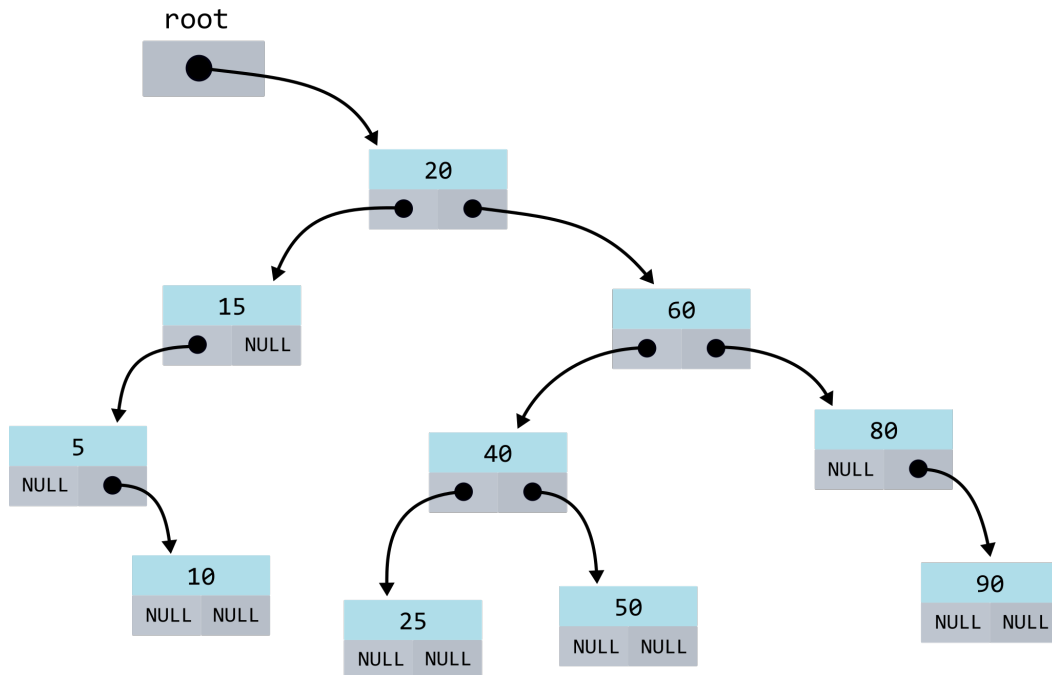


Семинар #11: Бинарные деревья поиска.



```
struct node
{
    int value;
    struct node* left;
    struct node* right;
}

typedef struct node Node;

Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
        root->left = bst_insert(root->left, x);
    else if (x > root->value)
        root->right = bst_insert(root->right, x);
    return root;
}
```

Стартовый код для этой части задания в файле `tree.c`.

Бинарное дерево - дерево, в котором у каждого узла может быть не более двух потомков.

Бинарное дерево поиска (binary search tree – bst)

- бинарное дерево со следующими условиями:

- У всех узлов левого поддерева `value` - меньше
- У всех узлов правого поддерева `value` - больше

Одинаковые элементы это дерево не хранит.

Глубина узла = количество предков узла + 1

Высота дерева = глубина самого глубокого узла

Сложность операций с bst:

- Поиск $O(h(n))$
- Добавление $O(h(n))$
- Удаление $O(h(n))$

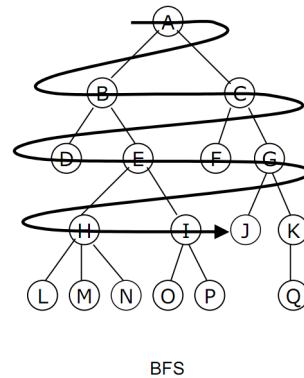
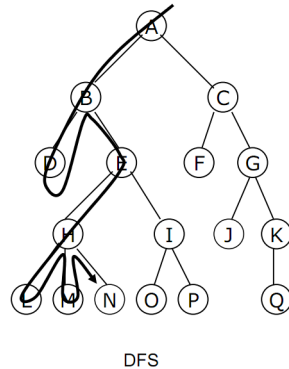
где $h(n)$ - высота дерева.

Для обычного дерева поиска $h(n)$ может достигать n .

Сбалансированное дерево - у *каждого* узла высоты левого поддерева и правого различаются не более, чем на 1. Для сбалансированного дерева $h(n) \approx \log(n)$.

AVL-деревья и **красно-черные деревья** - два способа балансировки деревьев.

- Написать функцию `int bst_size(const Node* root)`, вычисляющую количество элементов в данном дереве. Используйте рекурсию.
- Написать функцию `int bst_height(const Node* root)`, вычисляющую глубину бинарного дерева. Используйте рекурсию.
- Написать функцию `void bst_print_dfs(const Node* root)`, которая будет печатать все элементы дерева в порядке возрастания.



- Написать функцию `Node* bst_search(const Node* root, int val)`, которая ищет элемент в бинарном дереве и возвращает указатель на этот элемент. Если такого элемента нет, то функция должна вернуть `NULL`. Протестируйте эту функцию, печатая поддерево с помощью `print_ascii_tree`.
- Написать функцию `Node* bst_get_min(Node* root)`, которая возвращает указатель на минимальный элемент в этом дереве.
- Написать функцию `Node* bst_remove(Node* root, int x)`, которая удаляет элемент, содержащий `x`, из дерева поиска.

Нужно рассмотреть следующие случаи:

- Если `root == NULL`, то ничего не делаем
- Если `x > root->value`
- Если `x < root->value`
- Если `x == root->value` и у `root` нет детей
- Если `x == root->value` и `root` имеет одного левого ребёнка
- Если `x == root->value` и `root` имеет одного правого ребёнка
- Если `x == root->value` и `root` имеет двух детей. В этом случае делаем следующее:
 - * Находим минимальный элемент в правом поддереве.
 - * Копируем значение `val` из этого элемента в `root`.
 - * Удаляем этот минимальный элемент в правом поддереве, используя функцию `bst_remove`.

Протестируйте ваш код на всех случаях. Используйте функцию `print_ascii_tree` для проверки.

- Заполнить дерево миллионом случайных чисел и найти количество элементов и высоту этого дерева. Сравнить высоту с оптимальной $h_{optimal} = \lceil \log_2(n + 1) \rceil$
- Заполнить дерево миллионом последовательных чисел (от 1 до 10^6) и найти количество элементов и высоту этого дерева. Сравнить высоту с оптимальной.
- Написать функцию `void bst_print_bfs(const Node* root)`, которая будет печатать элементы в порядке их расстояния от узла (при равенстве расстояния печатать по возрастанию). Тут нужно использовать одну из реализаций абстрактного типа данных Очередь.

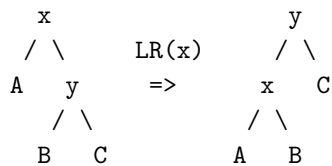
AVL-деревья

```
struct node
{
    int val;
    struct node* left;
    struct node* right;
    int height;
};
typedef struct node Node;
```

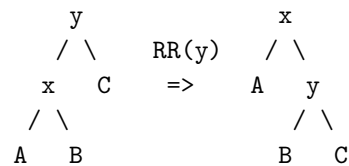
Стартовый код для этой части задания в файле `avl.c`.

- Написать функцию `void left_rotate(Node** proot)`. Проверьте функцию, используя `print_ascii_tree`.
- Написать функцию `void right_rotate(Node** proot)`.

левое вращение

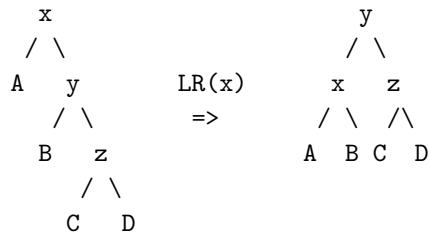


правое вращение

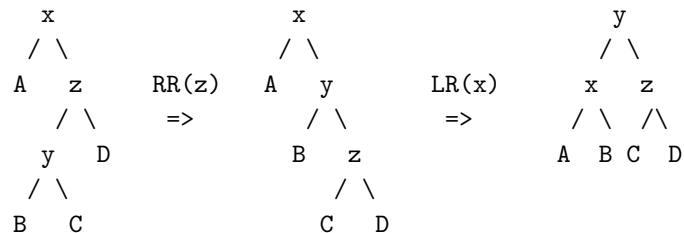


Тут `x` и `y` - узлы, а `A`, `B` и `C` - произвольные поддеревья.

- Написать функцию `void rebalance(Node** proot)`. Тут нужно рассмотреть 4 случая.
 - Если самая длинная ветвь справа-справа, то нужно сделать `LR(x)`.



- Если самая длинная ветвь справа-слева, то нужно сделать `RR(z)` и `LR(x)`.



- Два других случая для левой ветви зеркально симметричны.

- Написать функцию `tree_fix_height(Node* root)`, которая будет вычислять высоту узла `root`, при условии, что поле `height` у его детей задано правильно. Её нужно вызывать при каждом изменении дерева.
- Заполнить AVL-дерево миллионом случайных чисел и найти количество элементов и высоту этого AVL-дерева. Сравнить высоту с оптимальной.
- Заполнить AVL-дерево миллионом последовательный чисел (от 1 до 10^6) и найти количество элементов и высоту этого AVL-дерева. Сравнить высоту с оптимальной.