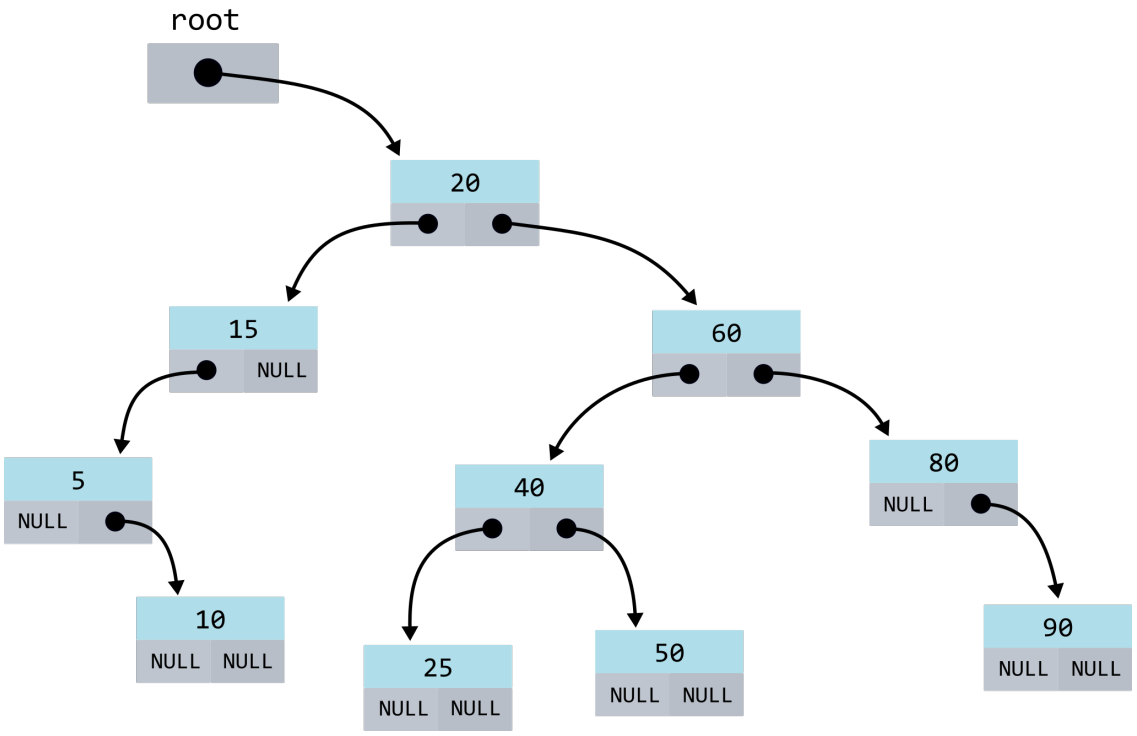
# Семинар #14: Деревья.

## Часть 1: 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;
}
```

**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 такое 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.

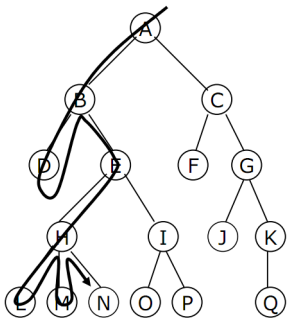
Стартовый код для этой части задания в файле `tree.c`.

- Написать функцию `size_t bst_size(const Node* root)`, вычисляющую количество элементов в данном дереве. Используйте рекурсию:

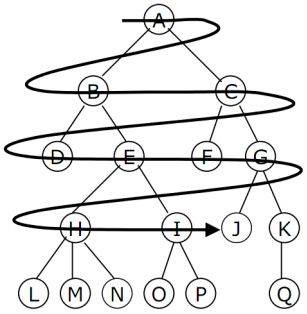
  `size(корня) = size(левого ребёнка) + size(правого ребёнка) + 1`

- Написать функцию `size_t 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.




DFS




BFS

- Написать функцию `Node* bst_search(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 рекурсивную функцию `Node* bst_remove(Node* root, int x)`, которая удаляет элемент, содержащий `x`, из дерева поиска. Функция должна возвращать указатель на узел, который встал на место удалённого узла. Если у удаляемого узла нет детей, то функция должна вернуть `NULL`.
  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 ребёнков. 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`.

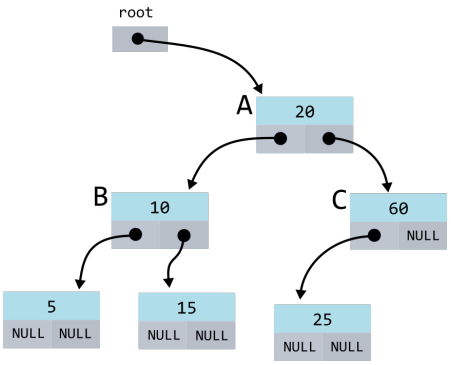
  Протестируйте ваш код на всех случаях. Используйте функцию `print_ascii_tree` для проверки.

- 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.

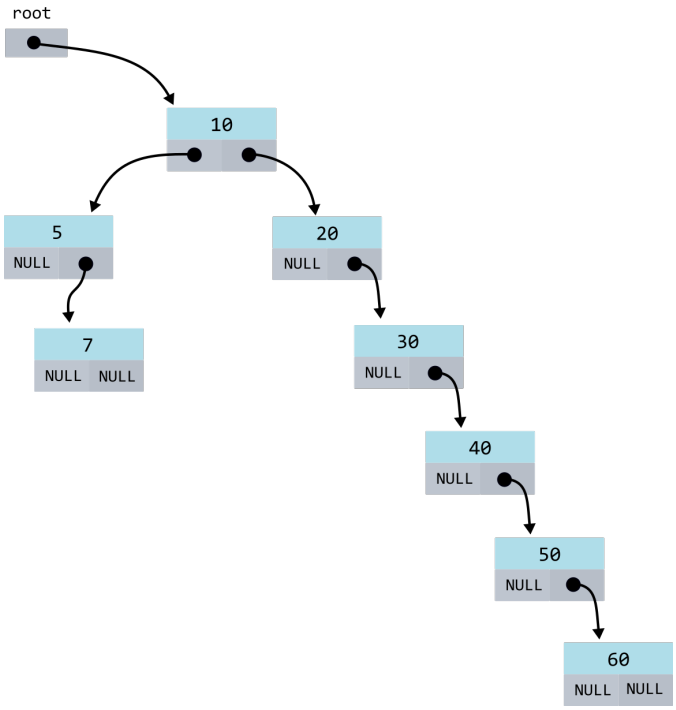
# Часть 2: Сбалансированные деревья

Сбалансированное дерево – это дерево у которого для *каждого* узла высоты левого поддерева и правого различаются не более, чем на 1.

**Пример 1:**
Это дерево сбалансированное, так как для узлов `A` и `B` глубины левого и правого поддерева равны, а для узла `C` глубины левого и правого поддерева отличается всего на `1`.




**Пример 2:**
Это дерево несбалансированное.




Можно показать, что для любого сбалансированного дерева его высота пропорциональна логарифму количества элементов и, как следствие, $h(n) = O(\log(n))$. Для несбалансированных деревьев высота может быть намного больше чем $\log(n)$. В худшем случае дерево вырождается в связный список.

Так как вычислительная сложность операций с деревом зависит от высоты дерева, то нам очень важно, чтобы дерево было сбалансированным. В ином случае, операции поиска/добавления/удаления будут работать намного медленнее. К сожалению, обычное бинарное дерево поиска, написанное нами в предыдущей части, не является сбалансированным. Это можно понять, если представить, что будет при добавлении в дерево последовательно возврастающих элементов с помощью функции `bst_insert`.

Однако, можно модифицировать бинарное дерево поиска так, чтобы дерево всегда оставалось сбалансированным. Есть два основных способа такой модификации:

1. AVL-деревья
2. Красно-черные деревья

Для самобалансирующихся деревьев поиска гарантируется, что вычислительная сложноть основных операций с ними будет равна $O(\log(n))$.

## Задачи:

- Заполнить дерево $n = 20000$ случайных чисел и найти количество элементов и высоту этого дерева. Сравнить высоту с оптимальной $h_{optimal} = \lceil log_2(n+1) \rceil = 14.3$. Помните, что вычислительная сложность операций с деревом равна $O(h)$, где $h$ – высота дерева.
- Заполнить дерево $n = 20000$ последовательными числами и найти количество элементов и высоту такого дерева. Сравнить высоту с оптимальной. Что будет, если увеличить $n$ до миллиона?