DATA STURCTURES

BST & AVL Tree

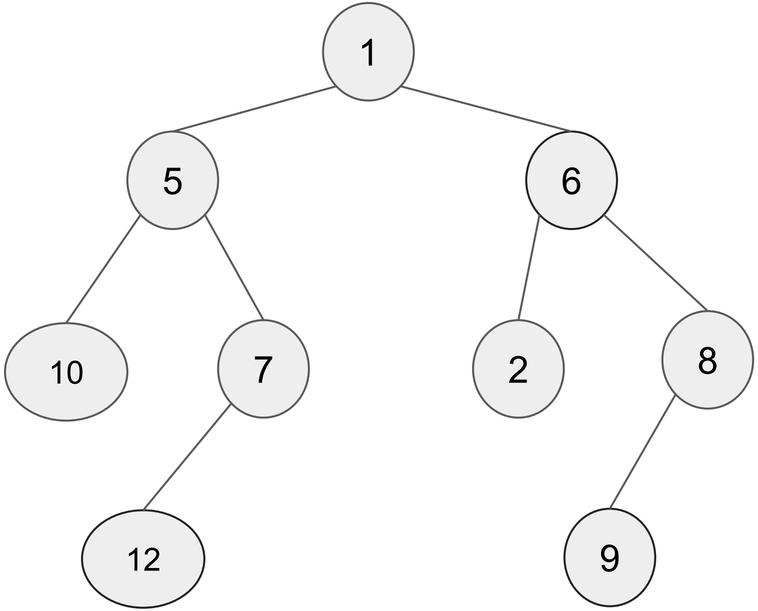
Agenda

- *Recap on Binary Tree*
- *What is Binary Search Tree?*
- *BST Operations with implementation*
- *What is Self-Balanced BST?*
- *Self-Balanced BST Examples*
- *What is AVL Tree?*
- *Tree Rotations & Unbalanced Types*
- *AVL Tree Operations with implementation*
- *BST STLs Applications*

Remember Binary Tree?

- *Logical non-linear DS* ⟶ Tree where each node has **at most** two children
- What is the complexity of searching in this?
    - This is unordered, so for each subtree should searching on element in left ***and*** right subtree
    - O(n) where n is number of nodes

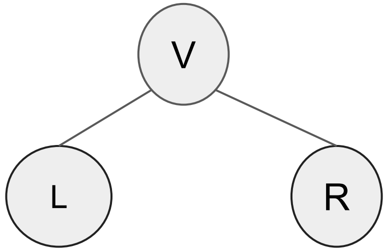
```
void search(BinaryTreeNode* cur, int target) {
    if (cur == nullptr) return;
    if (cur->data == target) {
        cout << "Found: " << target << endl;
        return;
    }
    search(cur->left, target);
    search(cur->right, target);
}
```

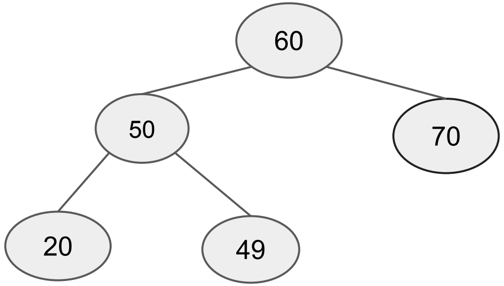


- O(*nodes*) is not **efficient**, Think in how to make this O(*Height*)

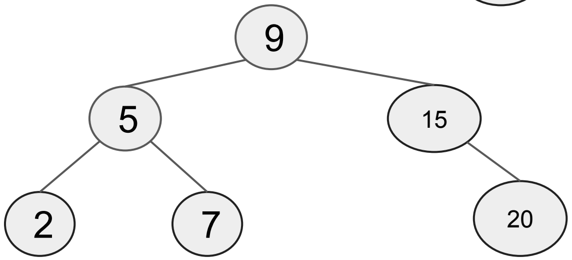
What is Binary Search Tree?

- *BST* ⟶ Binary Tree which the inorder traversal of it always **sorted.**
    - Think for 5 minutes how to what is the meaning of the inorder traversal is always sorted?
- Simply, L < V < R ⟶ for **all** subtrees
    - By above condition ⟶ BST has **unique** elements





Invalid



Valid

- So for every node i, i→val > all left nodes and i→val < all right nodes

Binary Search Tree ADT

- Main Operations
  - Insert element (No Duplications)
  - Delete element
  - Search
- Secondary Operations
  - findMax
  - successor
  - size
  - isEmpty
  - clear
  - printInorer

- For Simplicity, we implement this without template and with Node style implementation

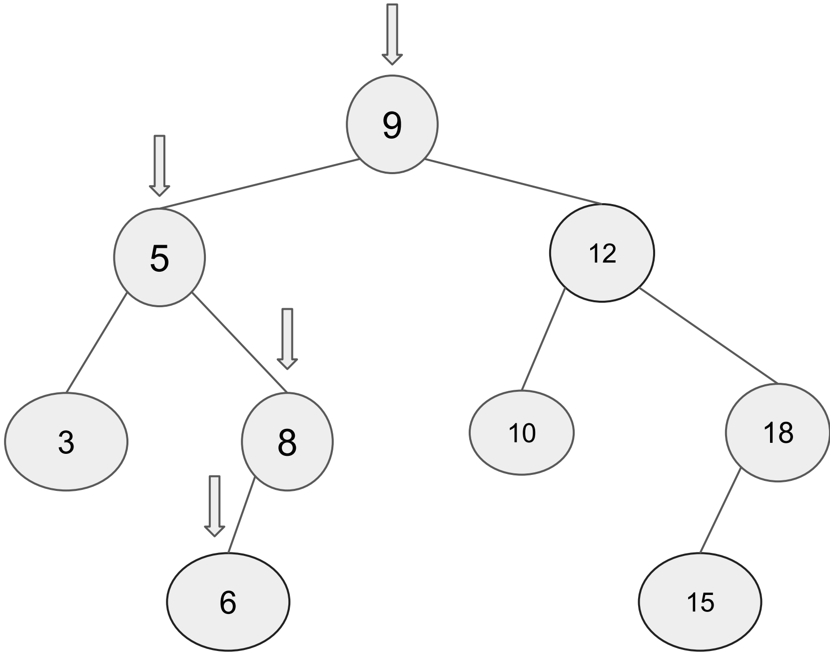
```
struct BinaryTreeNode {
    int data;
    BinaryTreeNode* left, * right;
    BinaryTreeNode(int val)
        : data(val), left(nullptr), right(nullptr) {}
};

class BinaryTree {
    BinaryTreeNode* root;

public:
    BinaryTree() : root(nullptr) {}

    //Main Operations
    void insert(int val);
    bool search(int target);
    void remove(int target);


    //Secondary Operations
    void inorderTraversal();
    int findMax();
    int findMin();
    int successor(int target);
    int size();
    bool isEmpty();
    void clear();
};
```

BST Searching

- search(6)
  - Start from root 9 > 6 → go left
  - 5 < 6 → go right
  - 8 > 6 → go left
  - 6 == 6 → then return true;

- Observation: for each time we go down **one level**
  - So the worst case is O(h)

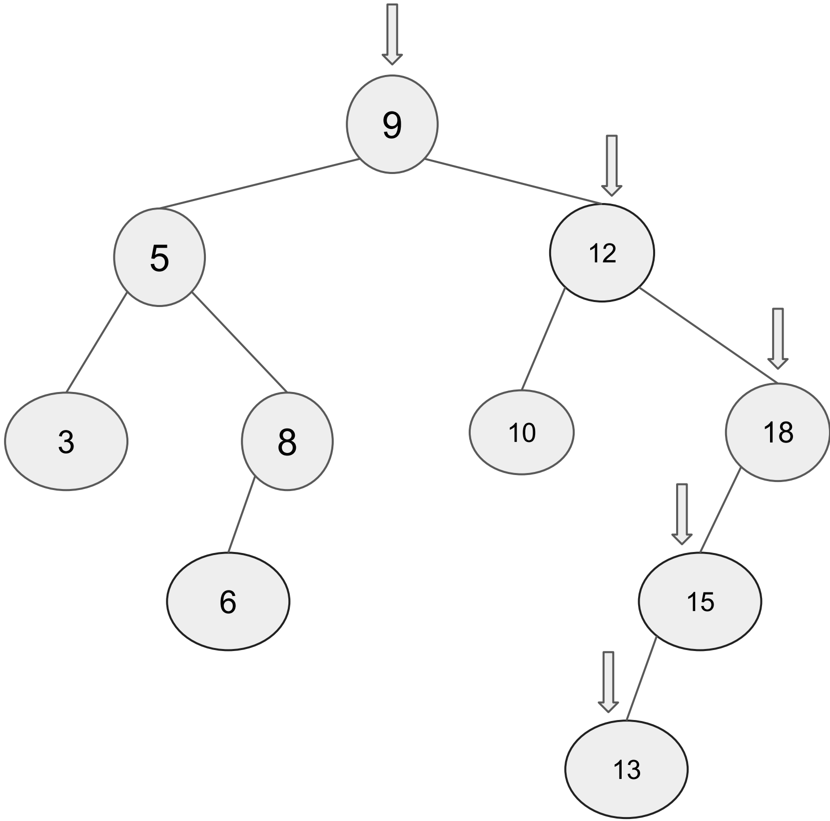
```
//This function can accessed by the user
bool BinaryTree::search(int target) {
    return root ? searchNode(root, target) : false;
}

//Private function to search for a node in the tree
bool BinaryTree::searchNode(BinaryTreeNode* cur, int target) {
    if (cur == nullptr) return false;

    if (cur->data == target) //found the target
        return true;
    else if (target < cur->data)  //go left subtree
        return searchNode(cur->left, target);
    else //go right subtree
        return searchNode(cur->right, target);
}
```



BST Insertion

- insert(13)
  - Start from root 9 < 13 → go right
  - 12 < 13 → go right
  - 18 > 13 → go left
  - 15 > 13 → go left
  - null → insert here 13

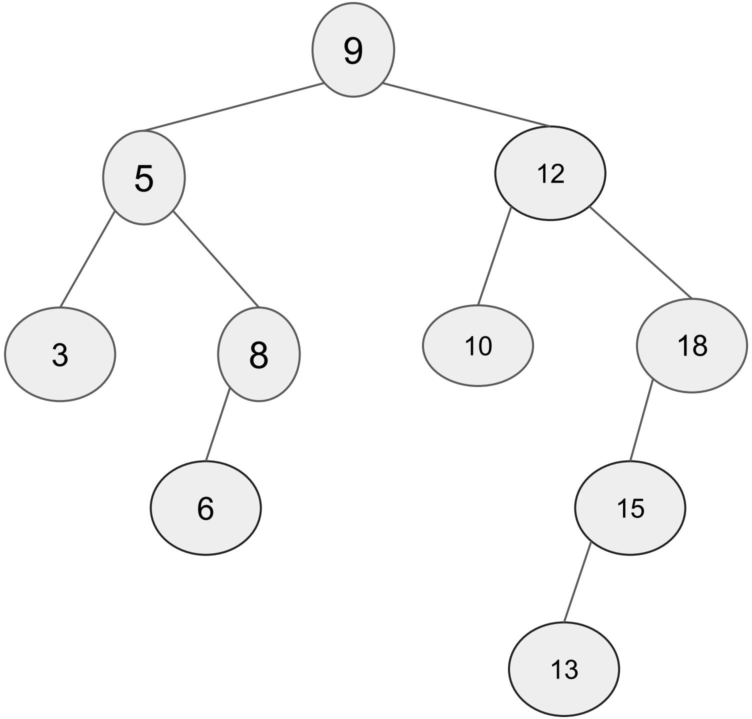
```cpp
//This function can accessed by the user
void BinaryTree::insert(int val) {
    if (root == nullptr)
        root = new BinaryTreeNode(val);
    else
        insertNode(root, val);
}

//Private function to insert a node in the tree
void BinaryTree::insertNode(BinaryTreeNode* cur, int val) {
    if (val < cur->data) {
        if (cur->left == nullptr)
            cur->left = new BinaryTreeNode(val);
        else
            insertNode(cur->left, val);
    }
    else if (val > cur->data) {
        if (cur->right == nullptr)
            cur->right = new BinaryTreeNode(val);
        else
            insertNode(cur->right, val);
    }
}
```



What is a Successor in BST?

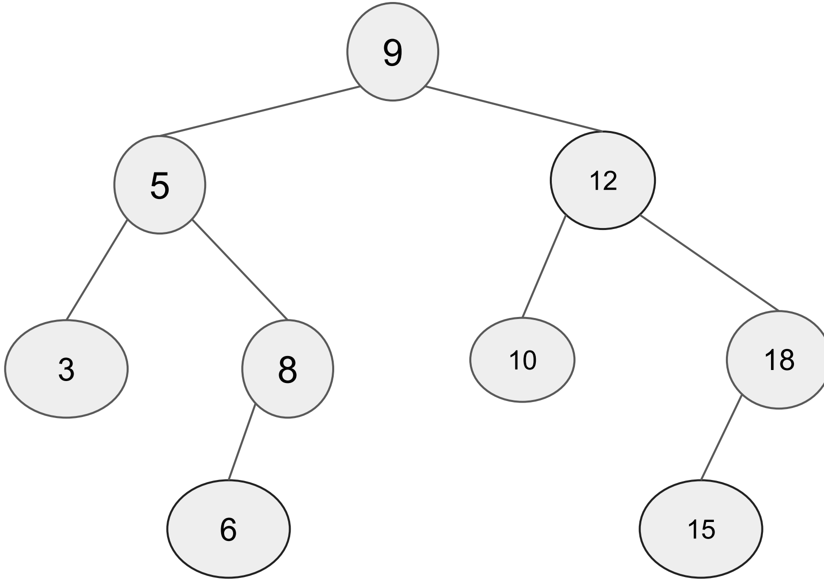
- The successor of a node is the next node in the in-order traversal sequence.
  - smallest element greater than the given node
- There are two cases:
  - successor is minimum node in right subtree of given node (**if exist**)
  - otherwise, the successor is the smallest ancestor where given node lies in the ancestor's left subtree.

```
int BinaryTree::successor(int target) {
    BinaryTreeNode* cur = root;
    BinaryTreeNode* succ = nullptr;// Track the potential successor

    while (cur != nullptr) {
        if (target < cur->data) {
            succ = cur;
            cur = cur->left;
        }
        else {
            cur = cur->right;
        }
    }

    if (!cur) return -1; //not found

    // Step 2: Case 1 (right subtree exists)
    if (cur->right) {
        cur = cur->right;
        while (cur->left) {
            cur = cur->left;
        }
        return cur->data;
    }

    // Step 3: Case 2 (return tracked successor)
    return succ->data;
}
```



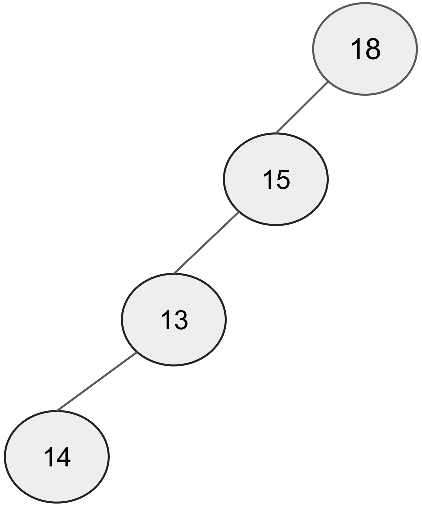
BST Deletion

- When we need to delete node in BST, we should think in 3 cases
  - node has one child or two child or it's leaf node? (think for 5 minutes for each case)
- 1st case: remove(3) → leaf case
  - Just go to 3 and delete it
- 2nd case: remove(8) → one child case
  - Go to 8 node and copy the data of his child.
  - Then, go to delete the child
- 2nd case: remove(12) → two child case
  - Go to 12 node and copy the data of his **successor** in his right **subtree** (**to keep it BST**)
  - Then, go to delete the successor



Remember Degenerate Tree?

- Now we know that the worst case for insertion,deletion and searching → O(h)
  - In Degenerate BST Case the h equal to the number of nodes

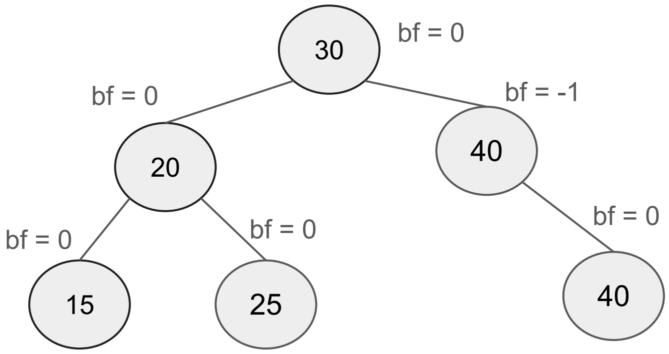


- BST is more efficient than Regular Binary Tree but still O(n)
  - How to make the worst case is always O(h) ~ O(logn)?

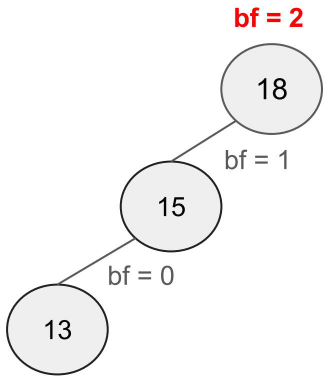
Self-Balanced BST

- A BST that **automatically maintains balance** (height ≈ O(log n)) during insertions/deletions
  - Ensures operations like search/insert/delete remain O(log n) in the worst case.
- **Examples:** Adelson-Velsky and Landis (**AVL** Tree), **Red-Black** Tree, **Splay** Tree, **B-Tree**, **Treaps**
- **AVL Tree:** **height balanced** BST which the **height diff** between left and right subtree **at most 1** for every node
  - Introduce the **balance factor** (**bf**) for every subtree = h(left subtree) - h(right subtree)
  - if the **bf** within {-1,0,1} -> this subtree is **balanced**
  - otherwise, **unbalanced**

```
struct AVLTreeNode {
    int data;
    AVLTreeNode* left, * right;
    //we use height frequently to calculate bf
    int height;
    AVLTreeNode(int val)
        : data(val), left(nullptr),
        right(nullptr), height(0) {}
};
```



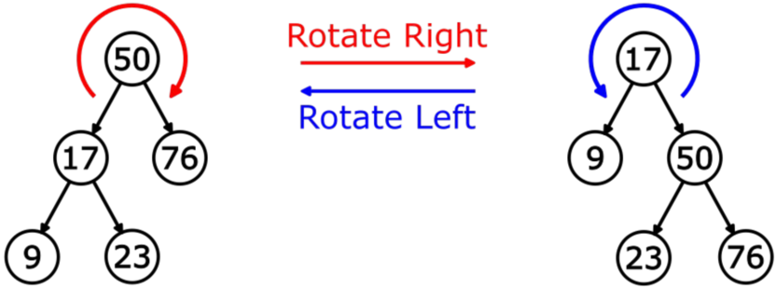
Balanced AVL Tree



Unbalanced Tree

Tree Rotations

- When insertion in AVL Tree -> the balance factor of new node within {-2,-1,0,1,2}
  - {-2,2} -> **unbalanced,** so we need to fix this **(rebalancing)**
- **Tree Rotations** will fix it
  - A tree rotation is a transformation on a binary tree that **preserves** inorder traversal of nodes.




```
AVLTreeNode* AVLTree::rightRotate(AVLTreeNode* y) {
    AVLTreeNode* x = y->left;
    // Perform rotation
    y->left = x->right;
    x->right = y;

    // Update heights
    updateHeight(y) , updateHeight(x);
    return x; // New root
}
```

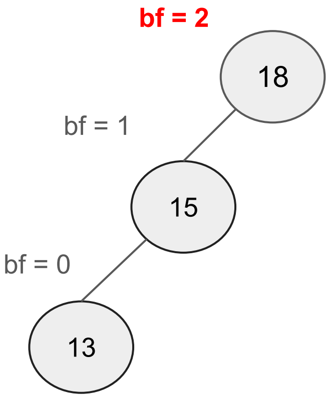
```
AVLTreeNode* AVLTree::leftRotate(AVLTreeNode* x) {
    AVLTreeNode* y = x->right;
    // Perform rotation
    x->right = y->left;
    y->left = x;

    // Update heights
    updateHeight(x) ,updateHeight(y);
    return y; // New root
}
```

Unbalanced Types

- **Single Rotation**

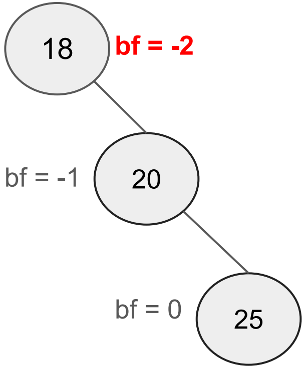
- **Double Rotation**

- **Left Rotation (LL)**



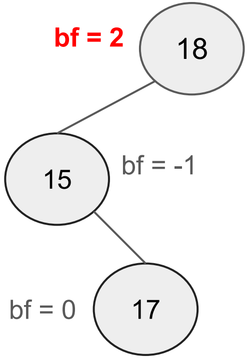
- Do Right Rotation

- **Right Rotation (RR)**



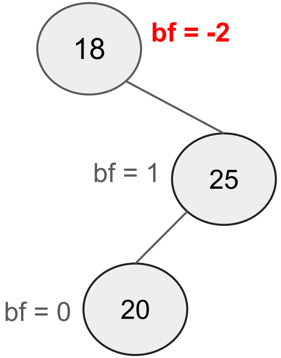
- Do Left Rotation

- **Left Right Rotation (LR)**



- Do Left Rotation (15) -> LL
- Then, Do Right Rotation

- **Right Left Rotation (RL)**



- Do Right Rotation (25) -> RR
- Then, Do Left Rotation

AVL Tree Helper Functions

```
//Update the height when insertion,deletion or rotation occurs
void AVLTree::updateHeight(AVLTreeNode* node) {
    if (node) {
        node->height = 1 + max(childHeight(node->left), childHeight(node->right));
    }
}
int AVLTree::childHeight(AVLTreeNode* node) {
    return node ? node->height : -1;
}
int AVLTree::balanceFactor(AVLTreeNode* node) {
    return node ? childHeight(node->left) - childHeight(node->right) : 0;
}
AVLTreeNode* AVLTree::rightRotate(AVLTreeNode* y) {
    AVLTreeNode* x = y->left;
    // Perform rotation
    y->left = x->right;
    x->right = y;

    // Update heights
    updateHeight(y) , updateHeight(x);
    return x; // New root
}
```

```
AVLTreeNode* AVLTree::leftRotate(AVLTreeNode* x) {
    AVLTreeNode* y = x->right;
    // Perform rotation
    x->right = y->left;
    y->left = x;

    // Update heights
    updateHeight(x) ,updateHeight(y);
    return y; // New root
}

//Used in insertion and deletion (rebalancing)
AVLTreeNode* AVLTree::balance(AVLTreeNode* node) {
    // balance the LL and LR case
    if (balanceFactor(node) == 2) {
        if (balanceFactor(node->left) == -1)
            node->left = leftRotate(node->left);
        node = rightRotate(node);
    }

    //balance the RR and RL case
    else if (balanceFactor(node) == -2) {
        if (balanceFactor(node->right) == 1)
            node->right = rightRotate(node->right);
        node = leftRotate(node);
    }

    return node;
}
```

AVL Tree Insertion

```cpp
//Private function to insert a node in the tree
AVLTreeNode* AVLTree::insertNode(AVLTreeNode* node, int val) {
    if (node == nullptr)
        return new AVLTreeNode(val);
    else if (val < node->data)
        node->left = insertNode(node->left, val);
    else if (val > node->data)
        node->right = insertNode(node->right, val);
    else
        return node; // Duplicate values are not allowed

    // Update height after insertion
    updateHeight(node);
    return balance(node); // Balance the tree
}
//This function can accessed by the user
void AVLTree::insert(int val) {
    root = insertNode(root, val);
}
```

AVL Tree Deletion

```
AVLTreeNode* AVLTree::removeNode(AVLTreeNode* node, int target) {
    if (node == nullptr) return node;
    if (target < node->data)
        node->left = removeNode(node->left, target);
    else if (target > node->data)
        node->right = removeNode(node->right, target);
    // Founded
    else { ... }

    // Update height after deletion
    updateHeight(node);

    // Balance the tree to assign it with parent
    return balance(node);
}

void AVLTree::remove(int target) {
    root = removeNode(root, target);
}
```

AVL Tree Deletion

```
// Founded
else {
    // Node with no child
    if (!node->left && !node->right) {
        delete node;
        return nullptr;
    }
    // Node with only one child or no child
    else if (!node->left) {
        AVLTreeNode* temp = node->right;
        delete node;
        return temp;
    }
    else if (!node->right) {
        AVLTreeNode* temp = node->left;
        delete node;
        return temp;
    }
    else {
        // Node with two children: Get the successor in the right subtree
        AVLTreeNode* temp = node->right;
        while (temp && temp->left) //get minimum node
            temp = temp->left;
        // Copy the successor's value
        node->data = temp->data;
        // Go delete the successor
        node->right = removeNode(node->right, temp->data);
    }
}
```

Set in STLs

```
void set_in_STLs() {
    //Example of Self-Balancing Binary Search Tree
    //based on Red-Black Tree
    set<int> bst;
    bst.insert(10);
    bst.insert(5);
    bst.insert(15);
    bst.insert(7);
    // Search for an element : return iterator on the found element of end()
    if (bst.find(7) != bst.end())
        cout << "Found 7 in the BST." << endl;
    else
        cout << "7 not found in the BST." << endl;

    // Remove an element
    bst.erase(5);
    if (bst.find(5) == bst.end())
        cout << "5 removed from the BST." << endl;
    else
        cout << "5 still exists in the BST." << endl;


    // Iterate through the BST
    cout << "Elements in the BST: ";
    for (const auto& elem : bst)
        cout << elem << " ";


    //another approach using iterators
    cout << "\nUsing iterators: ";
    for (auto it = bst.begin(); it != bst.end(); ++it)
        cout << *it << " ";


    //there is a multiset in STL
    multiset<int> mset;
}
```

Map in STLs

```
void map_in_STLs() {
    //Example of Self-Balancing Binary Search Tree
    //based on Red-Black Tree
    map<int, string> bst;
    bst[10] = "Ten";
    bst[5] = "Five";
    bst[15] = "Fifteen";
    bst[7] = "Seven";
    // Search for an element : return iterator on the found element of end()
    if (bst.find(7) != bst.end())
        cout << "Found 7 in the BST." << endl;
    else
        cout << "7 not found in the BST." << endl;

    // Remove an element
    bst.erase(5);
    if (bst.find(5) == bst.end())
        cout << "5 removed from the BST." << endl;
    else
        cout << "5 still exists in the BST." << endl;

    // Iterate through the BST
    cout << "Elements in the BST: ";
    for (const auto& elem : bst)
        cout << elem.first << ": " << elem.second << ", ";

    //another approach using iterators
    cout << "\nUsing iterators: ";
    for (auto it = bst.begin(); it != bst.end(); ++it)
        cout << it->first << ": " << it->second << ", ";

    //there is a multimap in STL
    multimap<int, string> mset;
}
```

Sheet link: (Link)

“Practice Makes Perfect”

Thank You!

Anas Elwkel