Data Structures Lab 10

Course: Data Structures (CL2001) **Semester:** Spring 2024
Instructor: Bushra Sattar **SLA:** M Anus

Note:

- Lab manual cover following below Stack and Queue topics
  {BST, Design and implementation, Traverse the tree with the three common orders, Operation such as searches, insertions, and removals on a binary search, AVL Tree, Rotation Cases}
- Maintain discipline during the lab.
- Just raise hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

BINARY SEARCH TREE

KeyPoint: A Binary Search Tree (BST) is a binary tree with the following properties:

- The left subtree of a particular node will always contain nodes whose keys are less than that node's key.
- The right subtree of a particular node will always contain nodes with keys greater than that node's key. The left and right subtree of a particular node will also, in turn, be binary search trees

BST Insertion

Sample Code of class Nodes

```
Create class Nodes
class Node {    private:
 int key;
 string name;

 Node leftChild;
  Node rightChild; public:
 Node(int key, string name) {

 this.key = key;
 this.name = name;

 }
string toString() {

 return cout<<name<< " has the key " <<key<<endl;
       } };
```

Task-1: Complete the following Code:
Create class BinaryTree and create a function which add nodes in BST

```
class BinaryTree {
  private:   Node root;
 public:
 void addNode(int key, string name) {

 ----------------------// Create a new Node and initialize it

 // If there is no root this becomes root

 if (root == NULL) {
  ---------------------------
```

```
 } else {

        // Set root as the Node we will start with as we traverse the tree

        ----------------------------
        // Future parent for new Node

        Node parent;

        while (true) {

               // root is the top set the parent node to the root node

               -------------------------

               // Check if the new node should go on
               // the left side of the parent node
Key is compared with that of root. If the key is less than root, it is compared with root's
left child key. If greater, it is compared with the root's right child. Continue this process
until the new node is compared with a leaf node and added either on the right or left child
depending on its key.

}
```

Tree Traversals: Inorder, PreOrder, PostOrder

Pseudo code For Inorder Traversal (iteration)

```
1) Create an empty stack S.
2) Initialize current node as root
3) Push the current node to S and set current = current->left until current is NULL
4) If current is NULL and stack is not empty then
     a) Pop the top item from stack.
     b) Print the popped item, set current = popped_item->right
     c) Go to step 3.
5) If current is NULL and stack is empty then we are done.
```

Task-2:
Write recursive algorithms that perform preorder and inorder tree walks.
Preorder Traversal approach.

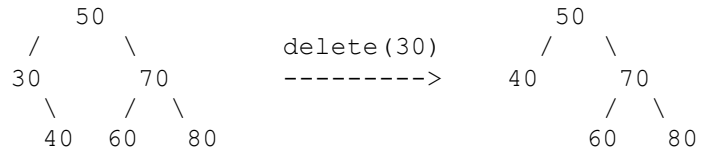
1. Visit Node.
2. Traverse Node's left sub-tree.
3. Traverse Node's right sub-tree.

BST Deletion

BST Deletion

1) ***Node to be deleted is the leaf:*** Simply remove from the tree.

2) ***Node to be deleted has only one child:*** Copy the child to the node and delete the child



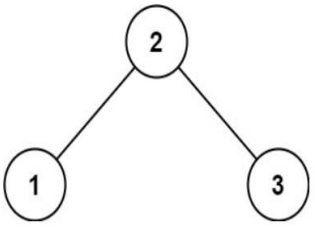
3) ***Node to be deleted has two children:*** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

The important thing to note is, inorder successor is needed only when the right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in the right child of the node.

Task:3

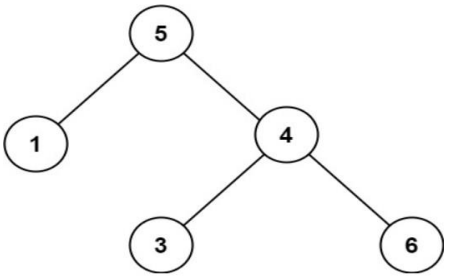
Given the root of the binary search tree, determine if it is a valid binary search tree.

Example 1:




```
Input: root = [2,1,3]
Output: true
```

Example 2:



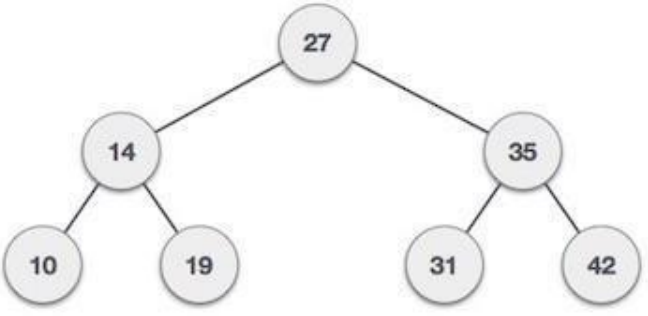

```
Input: root = [5,1,4,null,null,3,6]
Output: false
Explanation: The root node's value is 5 but its right child's value is 4.
```

Some points to Note:

Binary search tree (BST)

Binary search tree (BST) or a lexicographic tree is a binary tree data structure which has the following binary search tree properties:

- Each node has a value.
- The key value of the left child of a node is less than to the parent's key value.
- The key value of the right child of a node is greater than (or equal) to the parent's key value.
- And these properties hold true for every node in the tree.



- **Subtree**: any node in a tree and its descendants.
- **Depth of a node**: the number of steps to hop from the current node to the root node of the tree.
- **Depth of a tree**: the maximum depth of any of its leaves.
- **Height of a node**: the length of the longest downward path to a leaf from that node.
- **Full binary tree**: every leaf has the same depth and every nonleaf has two children.
- **Complete binary tree**: every level except for the deepest level must contain as many nodes as possible; and at the deepest level, all the nodes are as far left as possible.
- **Traversal**: an organized way to visit every member in the structure.

Traversals

The binary search tree property allows us to obtain all the keys in a binary search tree in a sorted order by a simple traversing algorithm, called an in order tree walk, that traverses the left sub tree of the root in in order traverse, then accessing the root node itself, then traversing in in-order the right sub tree of the root node.

The tree may also be traversed in preorder or post order traversals. By first accessing the root, and then the left and the right sub-tree or the right and then the left sub-tree to be traversed in preorder. And the opposite for the post order.
The algorithms are described below, with Node initialized to the tree's root.

Preorder Traversal

1. Visit Node.
2. Traverse Node's left sub-tree.
3. Traverse Node's right sub-tree.

In-order Traversal

1. Traverse Node's left sub-tree.
2. Visit Node.
3. Traverse Node's right sub-tree

Post-order Traversal

1. Traverse Node's left sub-tree.
2. Traverse Node's right sub-tree.
3. Visit Node

Searching

We use the following procedure to search for a node with a given key in a binary search tree. Given a pointer to the root of the tree and a key k, TREE-SEARCH returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

```
TREE-SEARCH (x, k)
1 if x = NIL or k = key[x]
2     then return x
3  if k < key[x]
4     then return TREE-SEARCH (left[x], k)
5     else return TREE-SEARCH (right[x], k)
```

The procedure begins its search at the root and traces a path downward in the tree, as shown in Figure 13.2. For each node x it encounters, it compares the key k with $key[x]$. If the two keys are equal, the search terminates. If k is smaller than $key[x]$, the search continues in the left subtree of x, since the binary-search-tree property implies that k could not be stored in the right subtree. Symmetrically, if k is larger than $key[k]$,

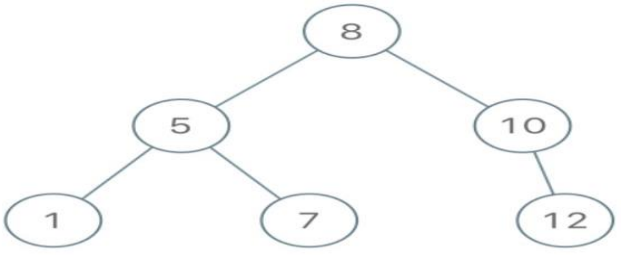
the search continues in the right subtree. The nodes encountered during the recursion form a path downward from the root of the tree, and thus the running time of TREE-SEARCH is $O(h)$, where h is the height of the tree.

Task 04: Write a recursive program to create a binary search tree. Your program should have the following functions:

- Insert
- DFS (Inorder, Pre-order, Post-order)
- Find_Height
- Find_Minimum
- Find_Maximum
- Find_Predecessor
- Find_Successor
- Delete (Leaf, Neither, and Full Node)
- Search

Task 05: Given an array of integers preorder, which represents the preorder traversal of a BST, construct the tree and return its root.

Example 1:




```
Input: preorder = [8,5,1,7,10,12]
Output: [8,5,10,1,7,null,12]
```

Task-6: Implement an AVL Tree in the following sub tasks

a. Insert Value 6,7,9 Make sure the tree balances in this arrangement with the latter 7-> root, 6-> left and 9-> right
b. Delete the Root node 7
c. Insert more element 12,13,14,4,2,5,8
d. Delete Node 8 and 5
e. Print the AVL Tree in (In-Order, Pre-Order and Post-Order)

Task-7: Write a recursive function which takes a number as parameter and finds out all numbers which are greater
than given number. Consider that all numbers are implemented in AVL tree.

Task-8: Write a complete code for AVL tree which satisfies below conditions:

1. If we insert a value in tree it must prints the message that which rotation will be performed either LL LR, RL,
RR or no Rotation.
2. If it requires RL or RL rotation, do not perform rotation just return the sum of all numbers presented in AVL
tree and prints a tree using in-order traversal without any rotation.**Task-8:** Write a program which construct the AVL tree on any 5 alphabetic characters (consider A is smallest and Z is largest). Write a function which checks that either given tree is palindrome or not. Use the BFS and DFS both for traversing.

AVL Tree

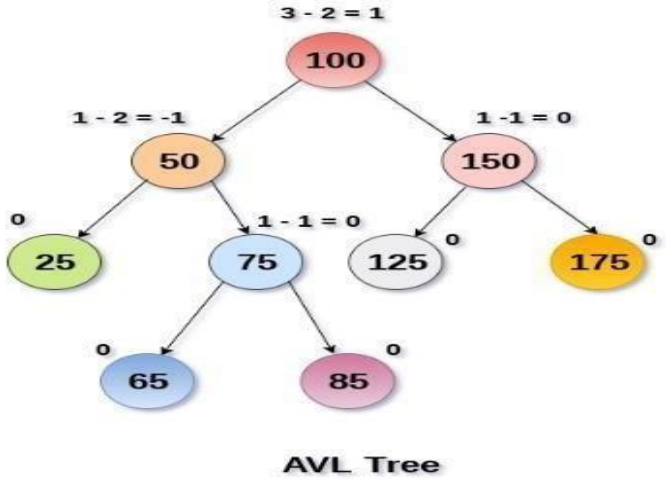
VL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

Balance Factor (k) = height (left(k)) - height (right(k))

If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.

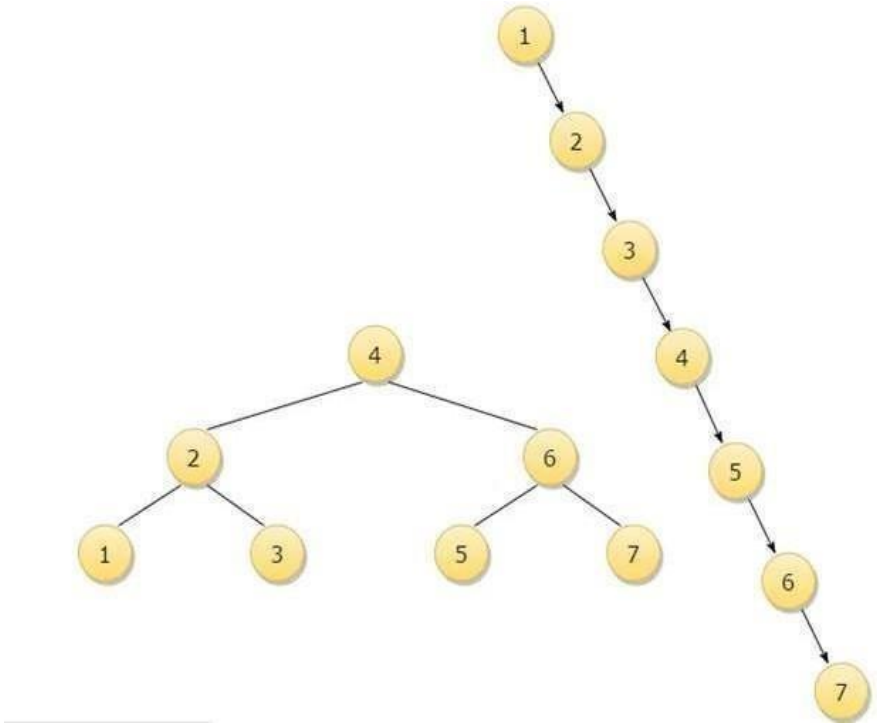


AVL Tree

Advantages of AVL tree

Since AVL trees are height balance trees, operations like insertion and deletion have low time complexity. Let us consider an example:

If you have the following tree having keys 1, 2, 3, 4, 5, 6, 7 and then the binary tree will be like the second figure:



To insert a node with a key Q in the binary tree, the algorithm requires seven comparisons, but if you insert the same key in AVL tree, from the above 1st figure, you can see that the algorithm will require three comparisons.

Representation of AVL Trees

```
class Node
{
    int element;
    int h;  //for height
    Node leftChild;
    Node rightChild;
}
```

Insertion in AVL Tree:

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing.

Following are two basic operations that can be performed to balance a BST without violating the BST

property (keys(left) < key(root) < keys(right)).

Steps to follow for insertion:

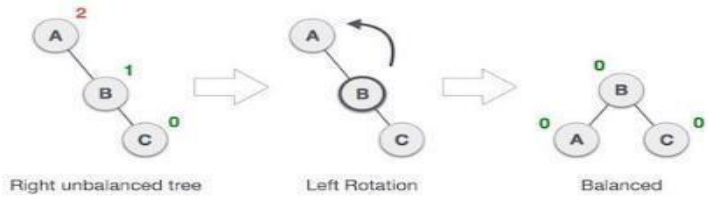
Let the newly inserted node be w

- **Perform standard BST insert for w.**
- **Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.**
- **Re-balance the tree by performing appropriate rotations on the subtree rooted with z.**
- **There can be 4 possible cases that need to be handled as x, y and z can be arranged in 4 ways.**
- **Following are the possible 4 arrangements:**
- **y is the left child of z and x is the left child of y (Left Left Case)**
- **y is the left child of z and x is the right child of y (Left Right Case)**
- **y is the right child of z and x is the right child of y (Right Right Case)**
- **y is the right child of z and x is the left child of y (Right Left Case)**

Rotation cases for insertion

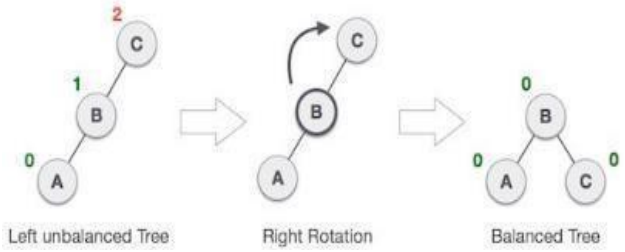
1. RR Rotation Case

When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



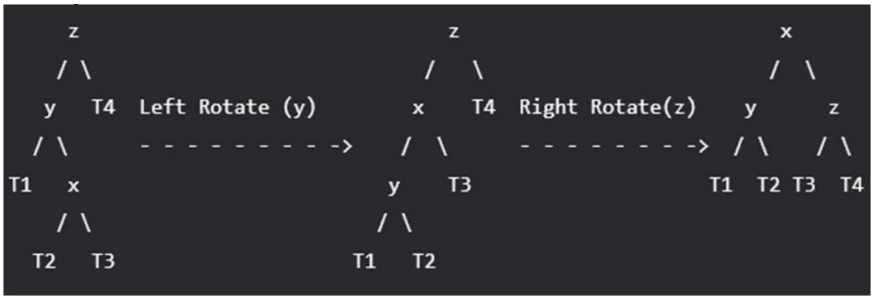
2. LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



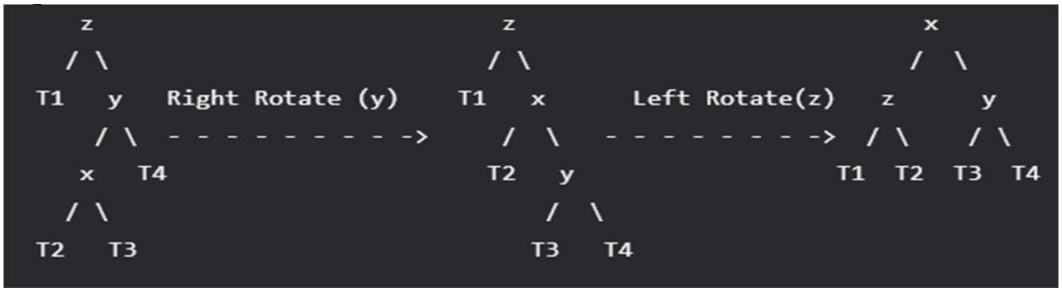
3. LR Rotation

Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.



4. RL Rotation

As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.



Deletion in an AVL Tree

To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) <keys(right)).

Let w be the node to be deleted

1. **Perform standard BST delete for w.**
2. **Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertion here.**
3. **Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:**

1. **y is left child of z and x is left child of y (Left Left Case)**
2. **y is left child of z and x is right child of y (Left Right Case)**
3. **y is right child of z and x is right child of y (Right Right Case)**
4. **y is right child of z and x is left child of y (Right Left Case)**

Like insertion, following are the operations to be performed in above mentioned 4 cases. Note that, unlike insertion, fixing the node z won't fix the complete AVL tree.

After fixing z, we may have to fix ancestors of z as well