Data Structures Lab 9

Course: Data Structures (CL2001) **Semester:** Fall 2022
Instructor: Muhammad Monis **T.A:** N/A

Note:

- Maintain discipline during the lab.
- Lab manual cover following below elementary sorting algorithms **{AVL trees, Rotation, Balancing, Insertion, Deletion, sort}**
- Listen and follow the instructions as they are given.
- 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.

For Reference you can use this site
https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

AVL tree is a binary search tree in which the difference of heights of left and right subtrees of any node is less than or equal to one. The technique of balancing the height of binary trees was developed by Adelson, Velskii, and Landi and hence given the short form as AVL tree or Balanced Binary Tree.

An AVL tree can be defined as follows:

Let T be a non-empty binary tree with T_L and T_R as its left and right subtrees. The tree is height balanced if:

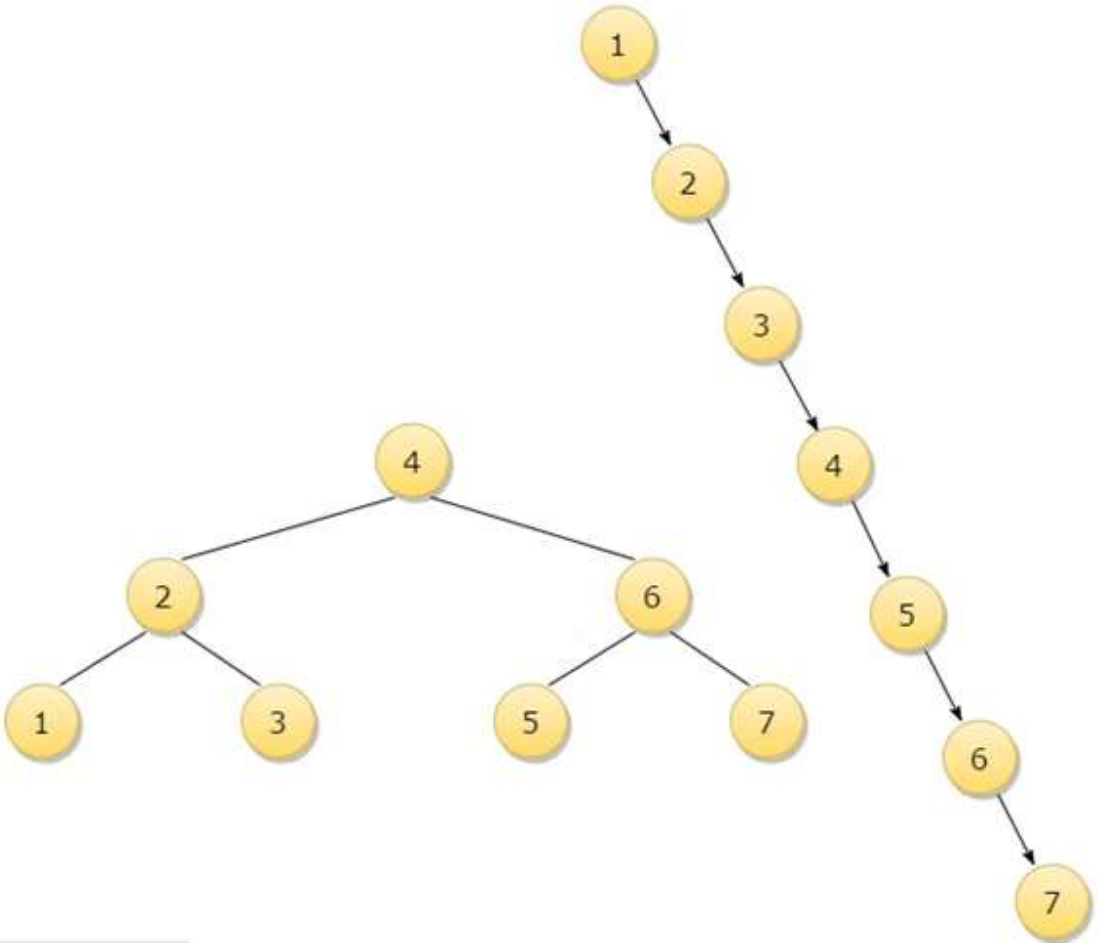
- T_L and T_R are height balanced
- $h_L - h_R <= 1$, where $h_L - h_R$ are the heights of T_L and T_R

The Balance factor of a node in a binary tree can have value 1, -1, 0, depending on whether the height of its left subtree is greater, less than or equal to the height of the right subtree.

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

```
Struct AVLNode
{
   int data;
   struct AVLNode *left, *right;
   int balfactor;
};
```

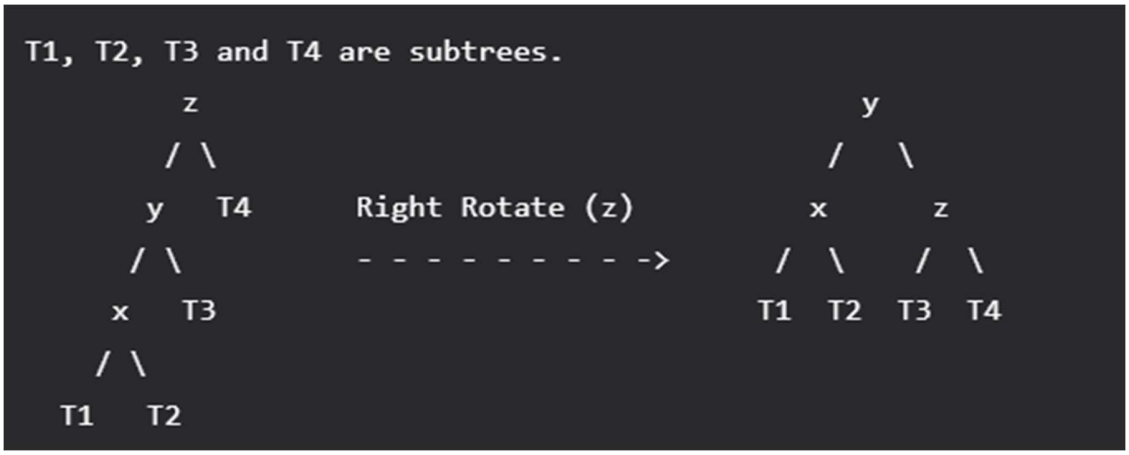
Insertion in AVL Tree:
To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing.
Following are two basic operations that can be performed to balance a BST without violating the BST property (keys(left) < key(root)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

Left Rotation
Right Rotation

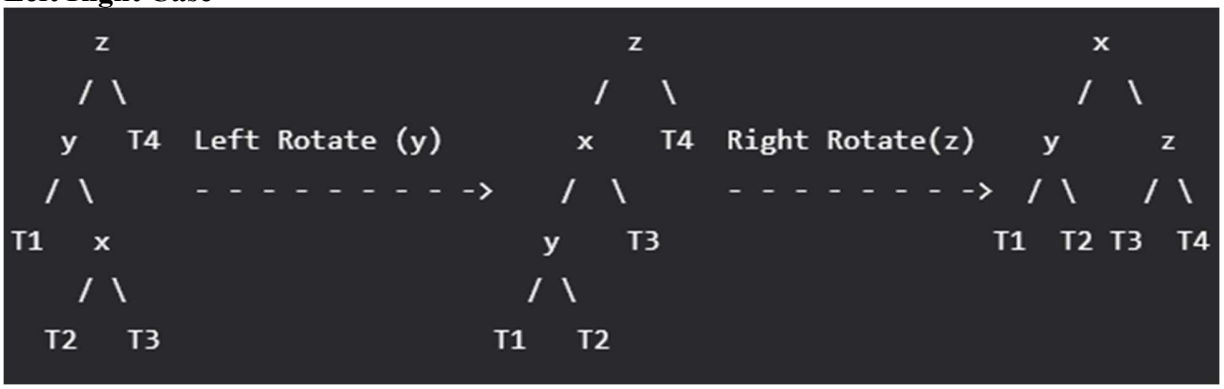
<u>Steps to follow for insertion:</u>
Let the newly inserted node be w

- **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)**

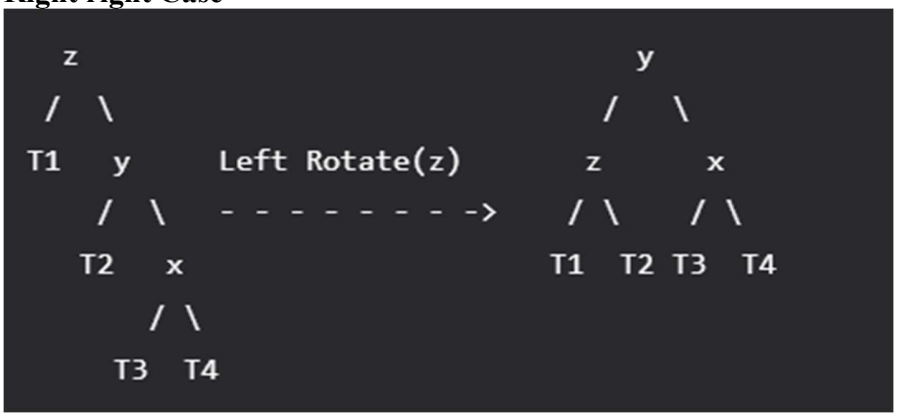
1. **Left Left Case**



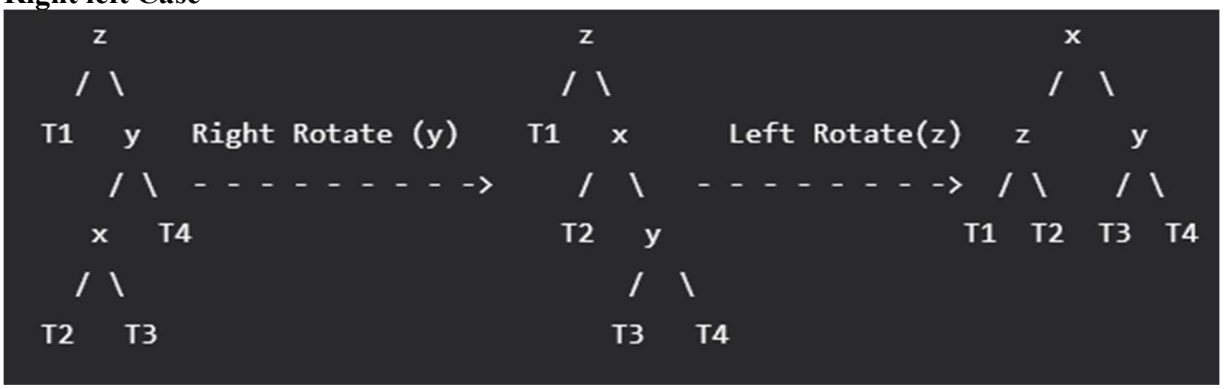
2. **Left Right Case**



3. **Right right Case**



4. **Right left Case**



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)).

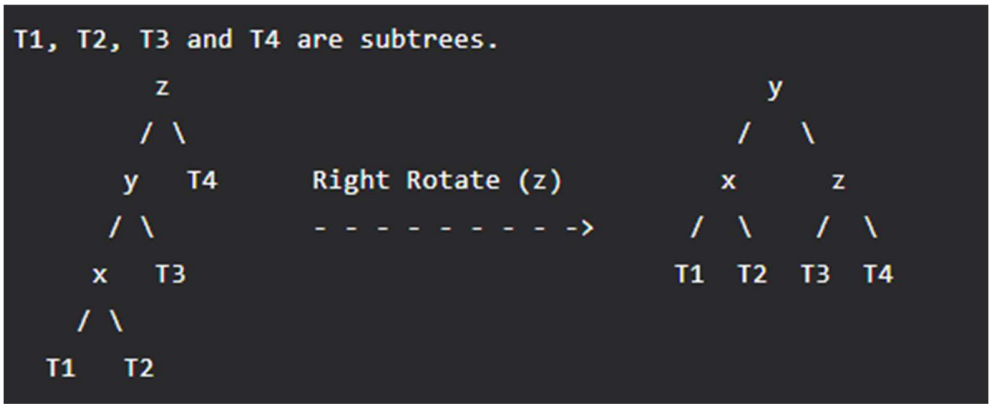
Left Rotation
Right Rotation

Let w be the node to be deleted

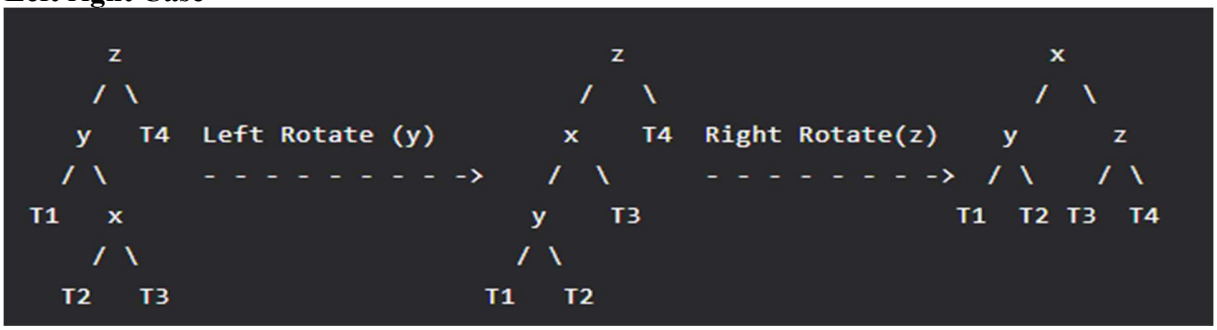
1. **Perform standard BST delete for w.**
2. **Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertion here.**
3. **Re-balance 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

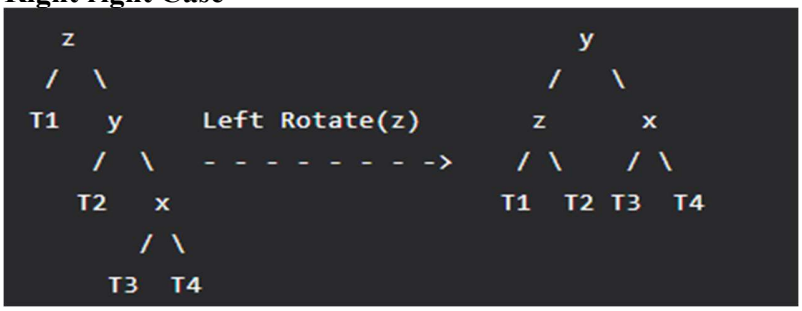
a) Left left Case



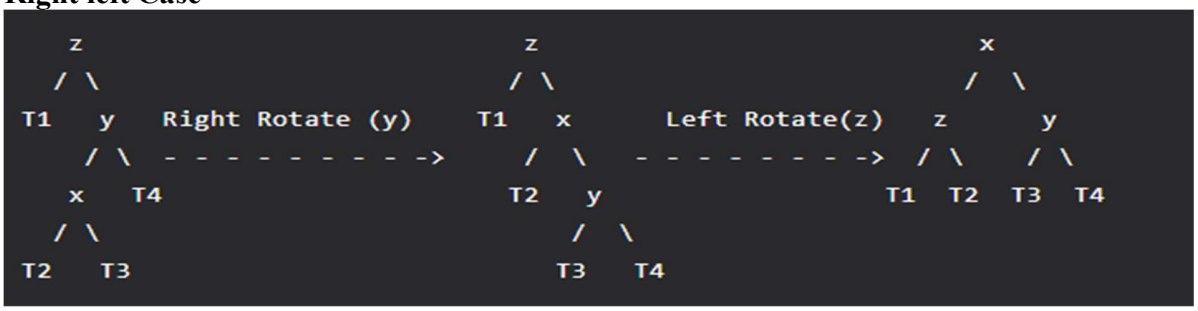
b) **Left right Case**



c) **Right right Case**



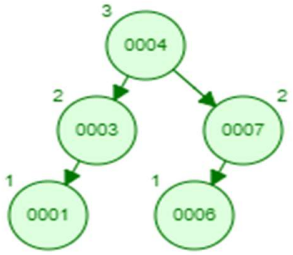
d) **Right left Case**



Tasks:

Implement an AVL Tree in the following sub tasks

a. **Insert Value 1,2,3 Make sure the tree balances in this arrangement with the latter 2-> root, 1-> left and 3-> right**
b. **Delete the Root node 2**
c. **Insert two more element 4,5,6,7,8**
d. **Delete Node 8 and 5**
   The Tree should look something like this after above operations



e. **Print the AVL Tree in (In-Order, Pre-Order and Post-Order)**