National University

Of Computer & Emerging Sciences Peshawar Campus

CL-210
Data Structures
Lab # 11

Objectives:

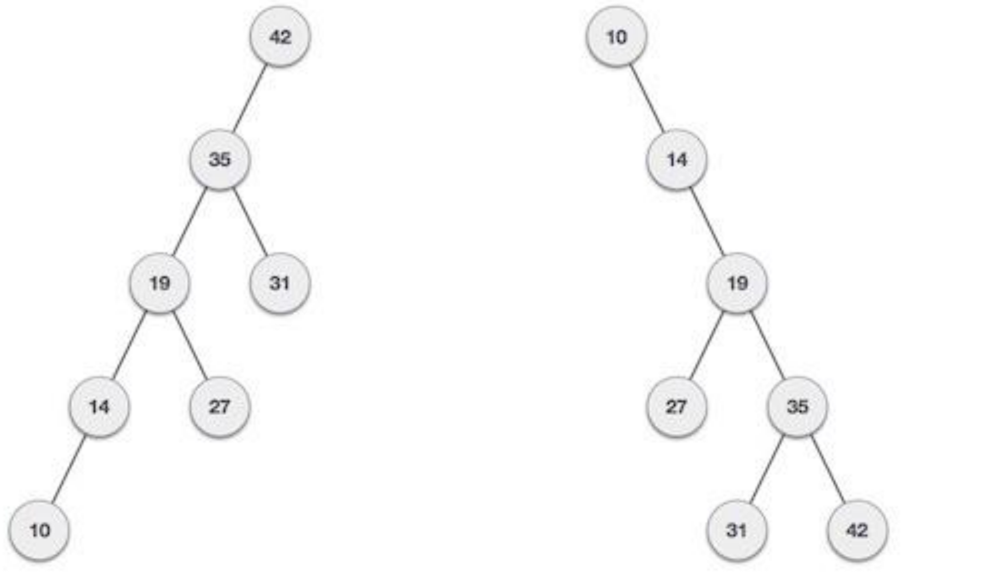
- AVL
- Insertion in AVL
- Deletion in AVL

Note: Carefully read the following instructions (*Each instruction contains a weightage***)**

1. There must be a block of comments at start of every question's code by students; the block should contain brief description about functionality of code.
2. Comment on every function and about its functionality.
3. Mention comments where necessary such as comments with variables, loop, classes etc to increase code understandability.
4. Use understandable name of variables.
5. Proper indentation of code is essential.
6. Write a code in C++ language.
7. Make a Microsoft Word file and paste all of your C++ code with all possible screenshots of every task **outputs in Microsoft Word and submit word file. Do not submit .cpp file.**
8. First think about statement problems and then write/draw your logic on copy.
9. After copy pencil work, code the problem statement on MS Studio C++ compiler.
10. At the end when you done your tasks, attached C++ created files in MS word file and make your submission on Google Classroom. (Make sure your submission is completed).
11. Please submit your file in this format **19F1234_L11**.
12. Do not submit your assignment after deadline. Late and email submission is not accepted.
13. Do not copy code from any source otherwise you will be penalized with negative marks.

Data Structure and Algorithms - AVL Trees

What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this −

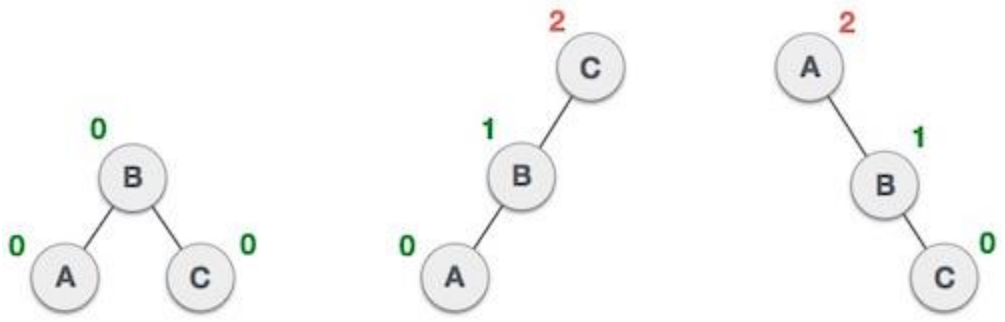


If input 'appears' non-increasing manner   If input 'appears' in non-decreasing manner

It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson**, **Velski** & **Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

Here we see that the first tree is balanced and the next two trees are not balanced −



Balanced   Not balanced   Not balanced

In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

```
BalanceFactor = height(left-sutree) − height(right-sutree)
```

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

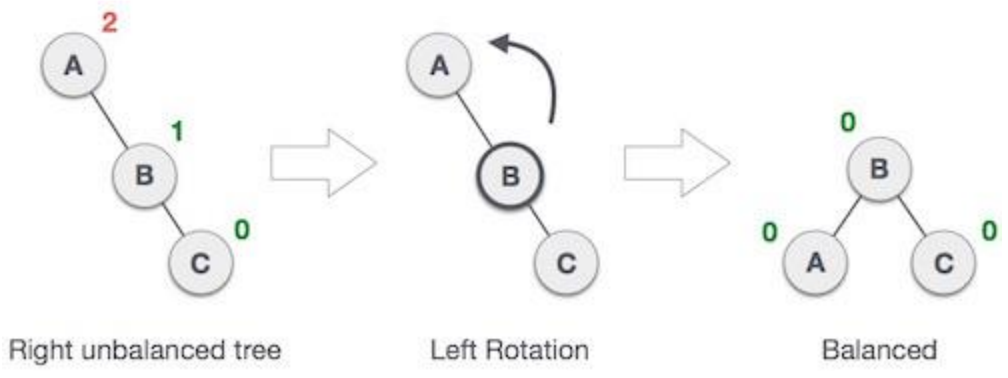
To balance itself, an AVL tree may perform the following four kinds of rotations −

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −

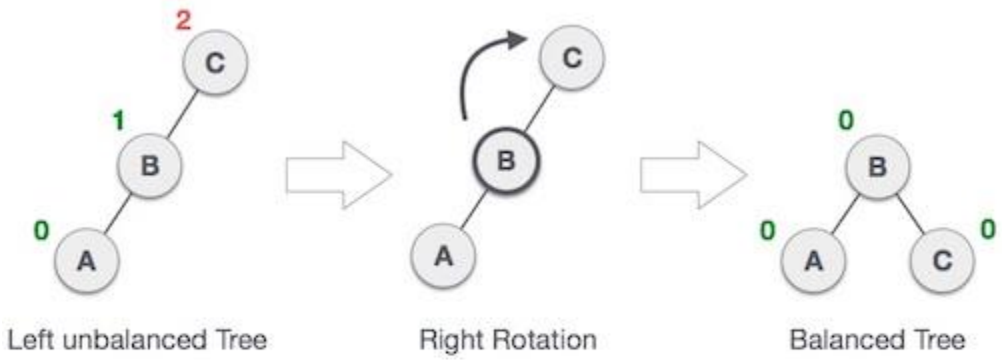


In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

```
node* AVLTree::singleLeftRotate(node* &A)
{
    node* newRoot = A->right;

    A->right = newRoot->left;

    newRoot->left = A;

    A->height = max(height(A ->left), height(A ->right)) + 1;

    newRoot ->height = max(height(newRoot->right), A->height) + 1;

    return newRoot;
}
```

Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

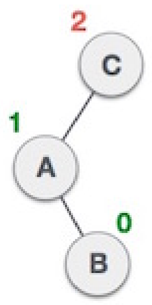
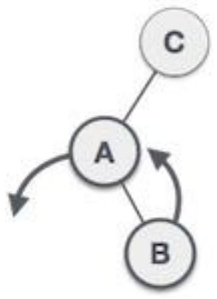
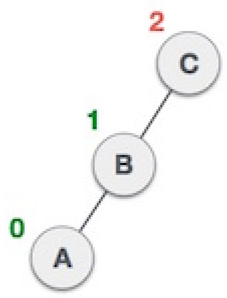


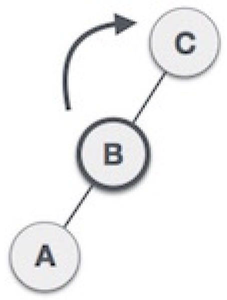
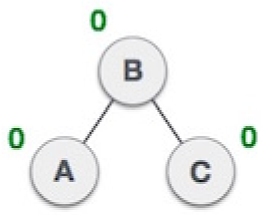
As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

```
node* AVLTree::singleRightRotate(node* &C)
{
    node* newRoot = C->left;

    C->left = newRoot->right;

    newRoot->right = C;

    C->height = max(height(C ->left), height(C ->right)) + 1;

    newRoot ->height = max(height(newRoot->left), C->height) + 1;

    return newRoot;
}
```

Left-Right Rotation

Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

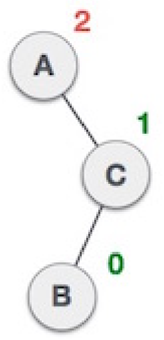
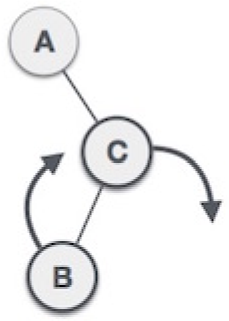
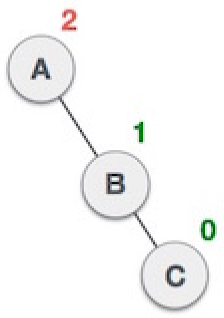
State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.

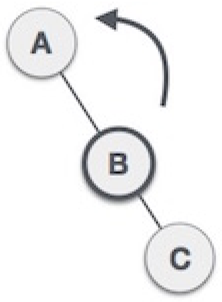
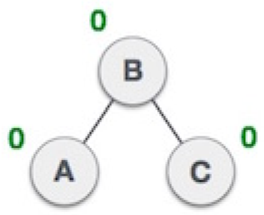
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.
	The tree is now balanced.

```
node* AVLTree::doubleLeftRightRotate(node*& t)
{
    t->left = singleLeftRotate(t->left);
    return singleRightRotate(t);
}
```

Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.
	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.
	Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.

	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.
	The tree is now balanced.

```
node* AVLTree::doubleRightLeftRotate(node*& t)
{
    t->right = singleRightRotate(t->right);
    return singleLeftRotate(t);
}
```

Problem: 1|

Provide a C++ implementation of AVL tree must include

- Recursive Height
- Finding Balancing Factor
- Recursive RR
- Recursive LL
- Recursive RL
- Recursive LR
- Apply on BST deletion
- Apply on BST insertion
- Display Nodes
- Test Your Code

☺ Best of luck