BST - Height Balanced Tree - AVL Implementation

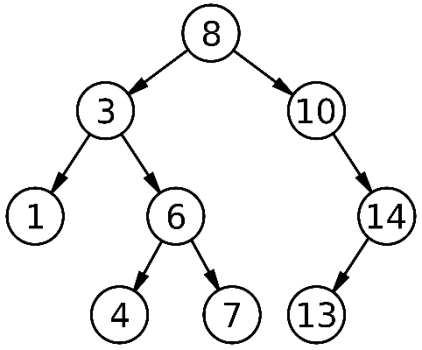
Abdul Mateen
Assistant Professor
PUCIT

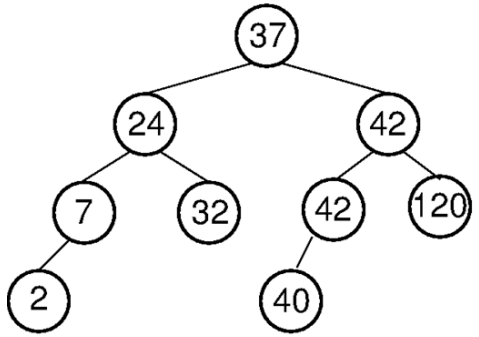
Binary Search Tree

Binary search tree (BST) is a binary tree, where each node has larger values on the right side of the node, whereas smaller values on the left hand side. You can see examples of BST at bottom. For example, consider BST on left hand side, on the right of node 8, we have 10, 14 & 13; whereas; on the left, we have 3, 1, 6, 4, & 7. Again, consider node 3, we have 1 on the left and 6, 4 & 7 on right hand side.

This representation of BST makes search efficient, for example if we have to search 7, we will start from root and proceed 8 -> 3 -> 6 -> 7. Similarly, to search 13 we have path 8 -> 10 -> 14 -> 13. Finally, to search 5, we will proceed 8 -> 3 -> 6 -> 4 and found that 5 does not exist in BST. Therefore, in first BST, we have 9 values and maximum number of comparisons are 4 that is height + 1 or you can say count of nodes on longest branch.

We can say that BST has search complexity O(height) of the tree. The minimum possible height of binary tree for (n) number of nodes is $\lfloor lg_2 n \rfloor$. For number of nodes 8 – 15, minimum possible height is 3, where $lg_2 8$ is 3 and $lg_2 15$ is 3.906891, the result of floor(3.906891).



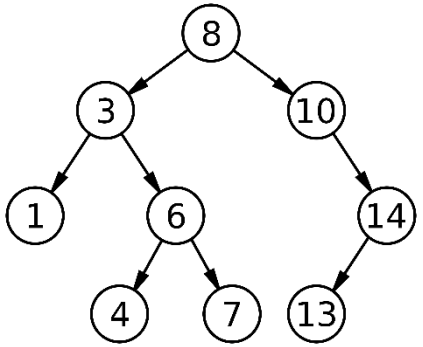


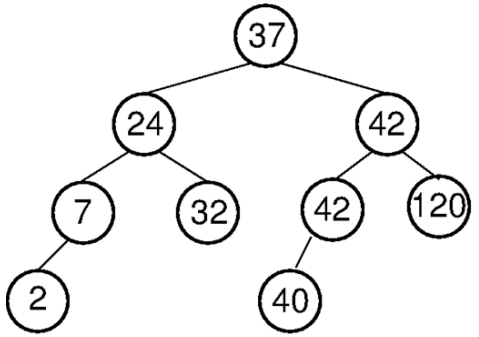
Binary Search Tree Height

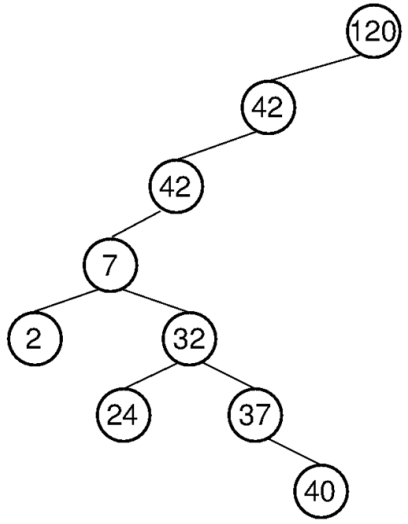
If the height of BST remain close to $\lfloor lg_2 n \rfloor$, then we have search complexity O($lg_2 n$), which is same as of Binary search of sorted array and BST is better because sorting operation requires O($nlgn$) time, which is too high and impractical for online data (means new data is coming and existing data can be modified, which means data becomes unsorted and to be sorted again. However, BST with complexity of insertion, deletion, searching $O(height)$ is affordable because statistics shows that more than **95%** times search operation is performed, whereas less than **5%** times other operations addition, deletion, modification etc. is performed.

However, BST may suffer from near worst/ max height that is $O(n)$. On the right hand side you can see third BST having height 6 with only 9 nodes. The reason is order of arrival of data or modification/ deletion ordering, which we can't restrict. For example, we can't say that first we will create ID cards of persons starting name with letter 'A' then of the persons starting name with letter 'B' etc. and we will not allow change in name / spelling error etc. once ID card is created. Therefore a BST may be skewed, (either on left or right it doesn't matter) and in result height will increase. As we already discuss a lot that all the operations are bounded by height (in BST search, addition, deletion etc. have worst time complexity order of height).

Finally, we need height balanced BST having height approximately order of $lg_2 n$.





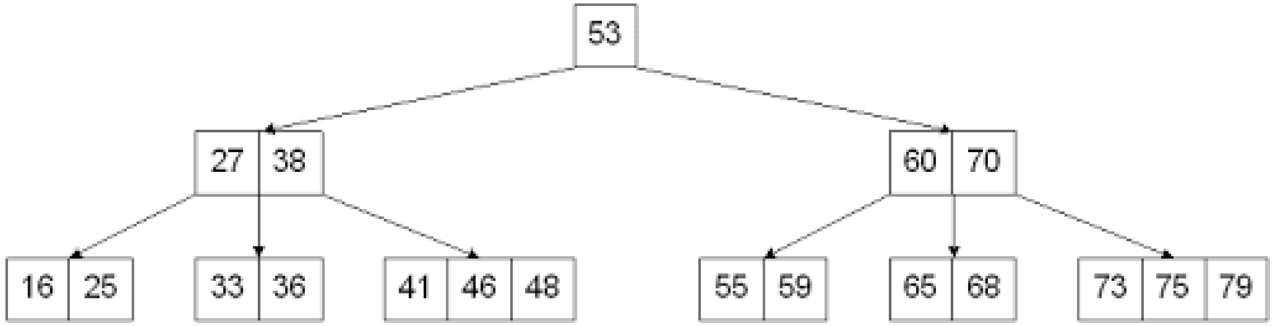


Height Balanced Tree

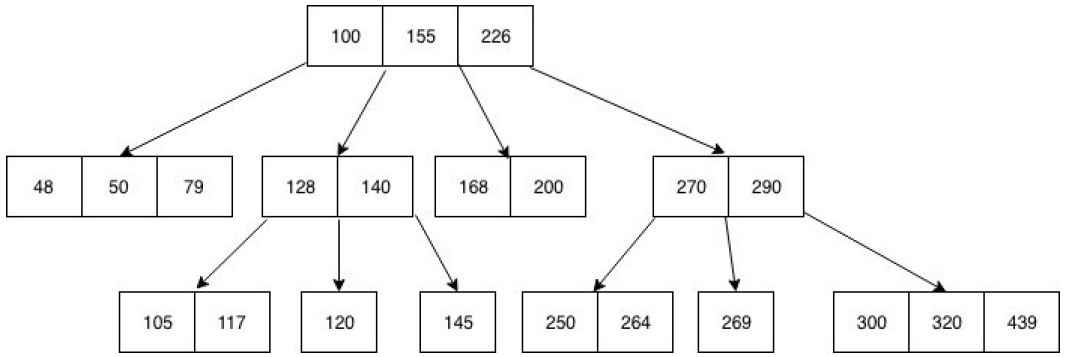
Height balanced tree has height approximately lgn, where base may be some positive integer depends on number of maximum child nodes, which in case of binary tree (most of the times discussed in academics) and we will mention some non-binary trees but mostly we will mainly we will focus on binary trees only. (Non-binary height balanced trees are not in the scope of our course)

There are number of binary height balanced trees like AVL, Red Black Tree, Splay Tree and there are non-binary height balanced trees like 2-3 trees, B-Trees etc.

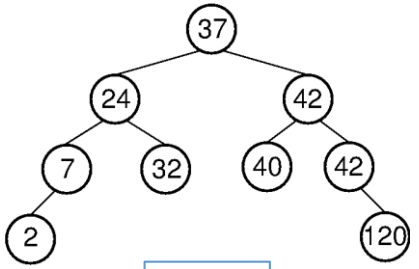
You can see examples of all of height balanced trees .



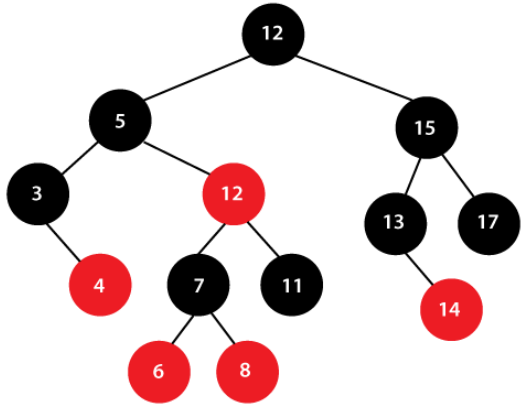
2-3 Trees



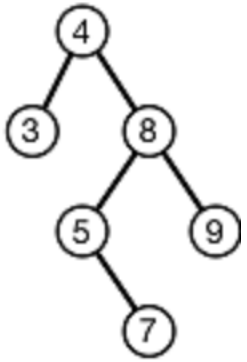
B Trees



AVL



Red Black Tree

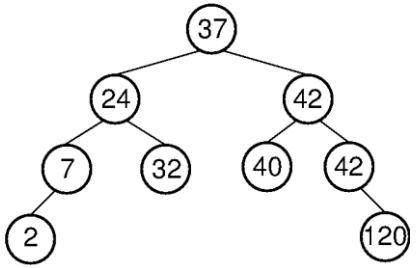


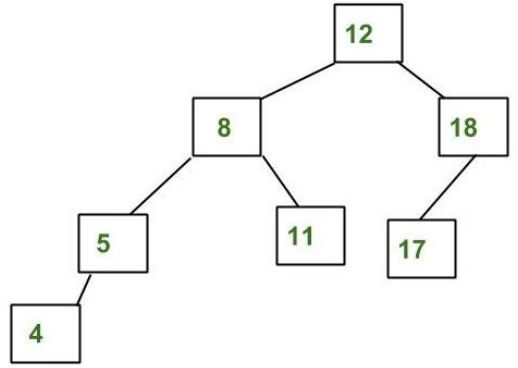
Splay Tree

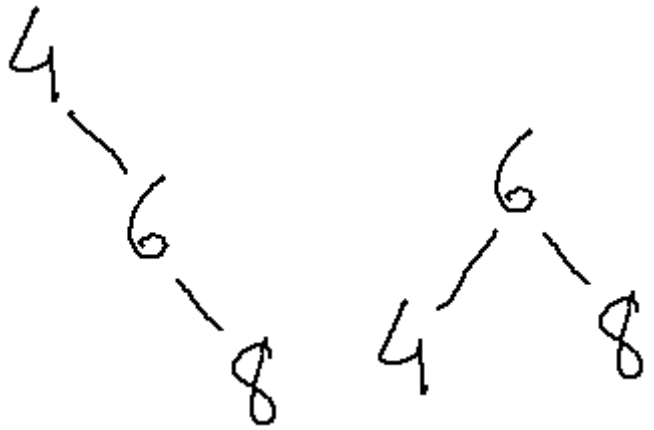
AVL Tree

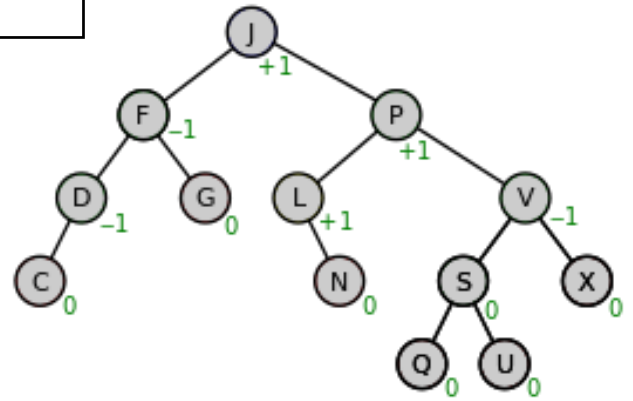
AVL is height balanced BST, where each node has balance factor +1, 0 or -1. Trivially, a tree can have minimum 1 node, therefore each node and its child nodes can be viewed as trees (sub-tree). The Balance factor is difference between height of left sub-tree and height of right sub-tree. On left hand side we have 3 examples of AVL, whereas in third example, you can see balance factor at each node.

As AVL is BST, therefore when insertion or deletion operation is performed, tree may become unbalanced, see left most figure at bottom. The balance at node 4 is -2 (-1-1), where -1 is the height on left sub-tree (no node) and +1 is height on right sub-tree, hence difference is -2. After insertion or deletion operation, balance is checked at each node on the path upside till root node (balanced is checked from bottom to top, for example 8 is inserted and balance is checked at 6 then 4 and so on). The first unbalanced node found is called **Pivot** node. Rotation is performed at pivot node to balance the tree. For example, in second figure right rotation is performed and tree is now balanced.





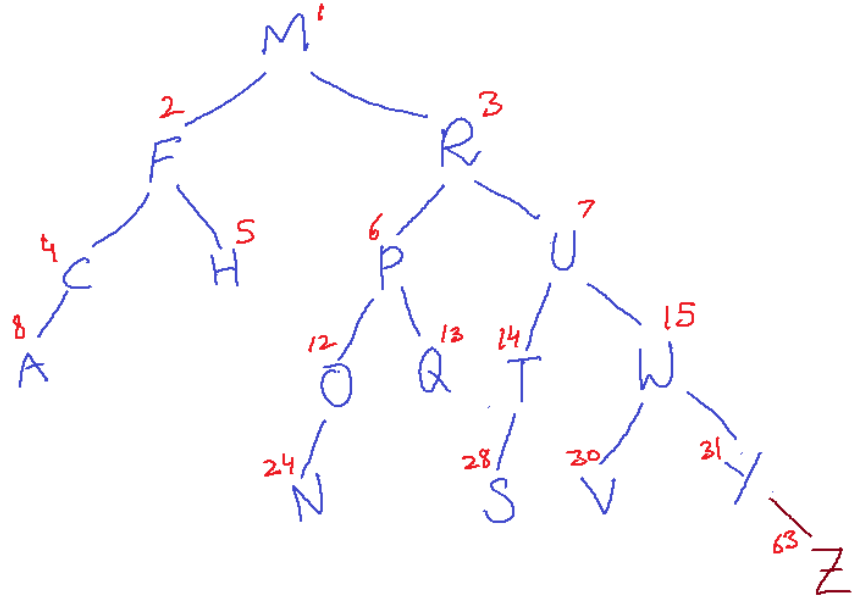


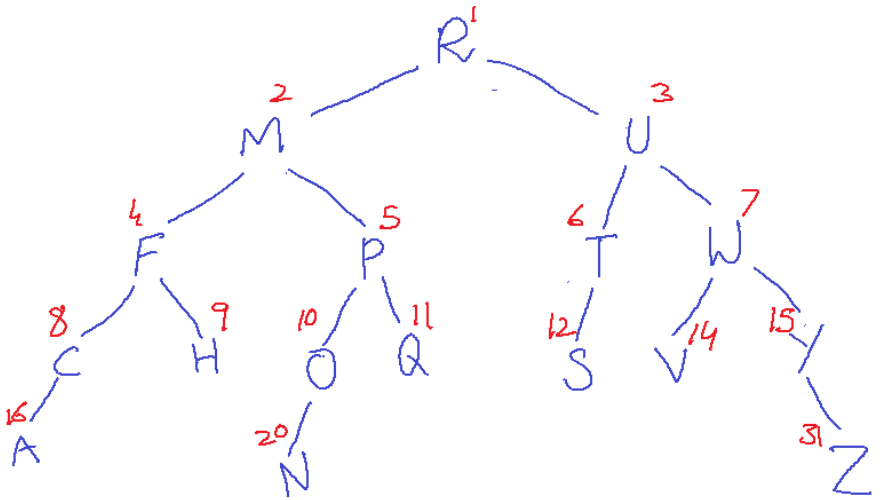


AVL Linked vs Array Implementation

As discussed in previous slide that for balance, rotation operation is required in AVL. Consider array based implementation of AVL and two figures on the bottom, where array indexes are given with each node. The left side tree was balanced when Z is inserted and tree has become unbalanced, where **M** (root node) is pivot node having balance factor (2-4=-2), therefore rotation is required and on the right hand side, you can see same tree is balanced after rotation, however, index of all the nodes are changed, which means we have to change indexes of all nodes (in case root node is pivot node), which is not at all possible, therefore we can't do array implementation of AVL.

Hence, we have linked based implementation of AVL. Also to reduce time to find balance factor, BST node is redefined as AVL node with an addition factor height of each node, by default each node has height 0 (zero), whereas all the nodes on the top of leaf have height one plus max(height of left sub-tree, height of right sub-tree).





Generation of BST from Binary Traversals

We can construct/ generate BST from binary traversal, where pre-order & in-order or pre-order & post-order two traversals required. Yes, we can say (as pointed by one of my student during class) that we can generate In-order by sorting, because in-order traversal always give sorted data in BST. So, we need pre-order or post-order for generation of BST. This is required to verify that tree has the shape, we are thinking of, for example BST (with same nodes) may have height between n to lgn, therefore to verify generation of Binary Ttree from traversals is handy.

Here, we will learn construction of Binary Tree from Pre-order and In-order traversals. Consider, the example:

In Order:

A B D G E H I L C F J M N K

Pre Order:

D G B H E I L A C M J N F K

Find and Locate Root

Pre Order:

In pre-order, first node is root of binary tree

A B D G E H I L C F J M N K

In Order:

Locating root in in-order, we can find left sub tree on left of node and right sub tree on right side of the node

D G B H E I L A C M J N F K

Left sub tree

Root

Right sub tree

Consider Left Subtree

Pre Order:

Again in pre-order, first node is root of left sub tee

B D G E H I L

In Order:

Locating root in-order, we can find left sub tree on left of node and right sub tree on right side of the node

D G B H E I L

Left sub tree | Root | Right sub tree

Consider Right Subtree

Pre Order:

Again in pre-order, first node is root of left sub tee

C F J M N K

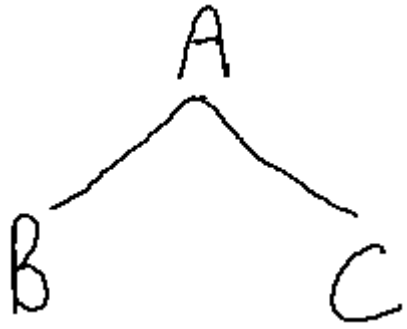
In Order:

Locating root in-order, we can find left sub tree on left of node and right sub tree on right side of the node

C M J N F K

Root

Right sub tree



Consider Right Subtree of Node B

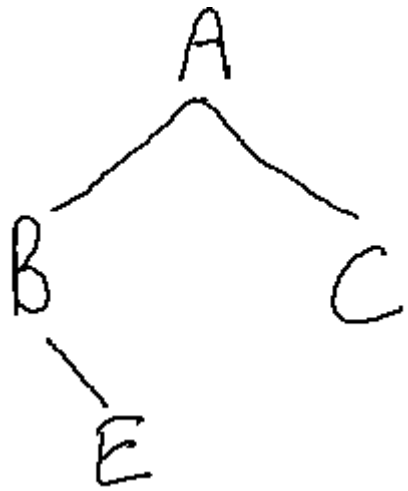
Again in pre-order, first node is root

Pre Order: E H I L

Locating root in in-order, we can find left sub tree on left of node and right sub tree on right side of the node

In Order: H E I L

Left sub tree | Root | Right sub tree

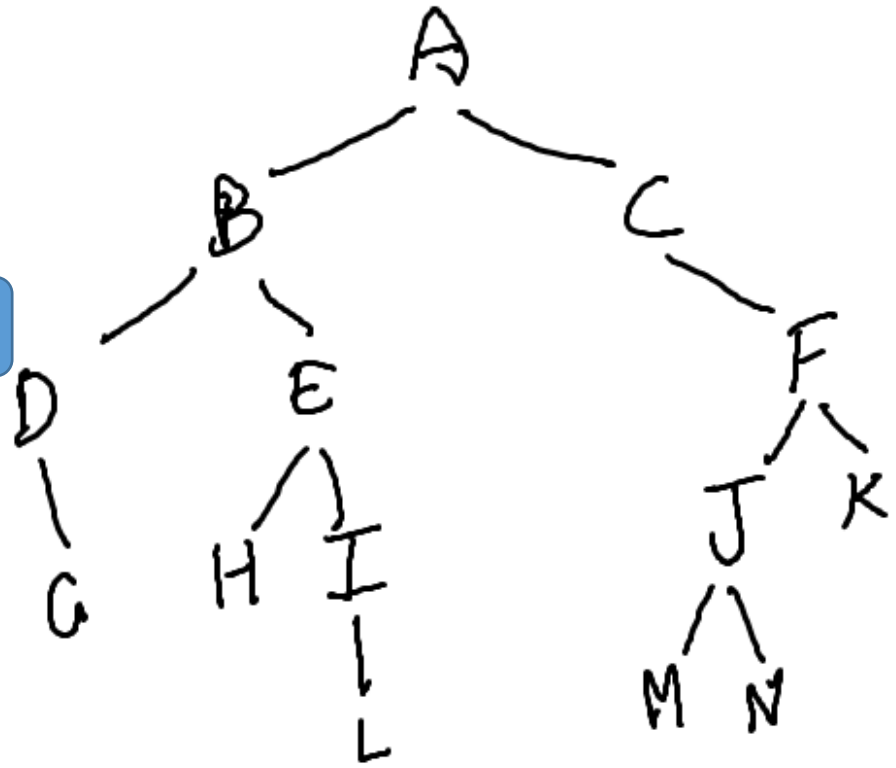


Complete Binary Tree Constructed

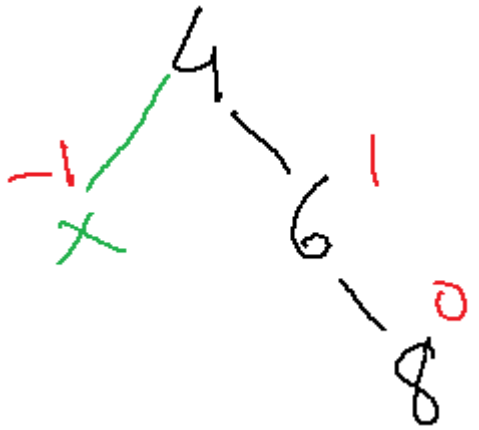
A B D G E H I L C F J M N K

D G B H E I L A C M J N F K

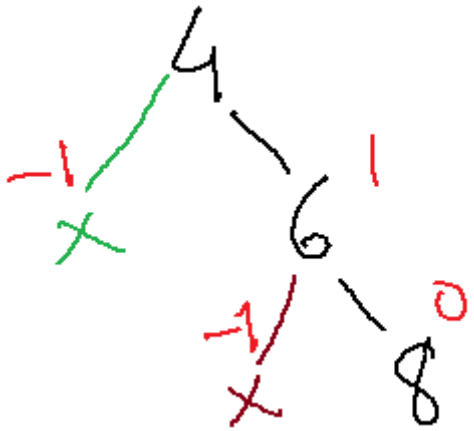
Binary Tree constructed from In-order (first line) and Pre-order (second line)



AVL Single Rotation

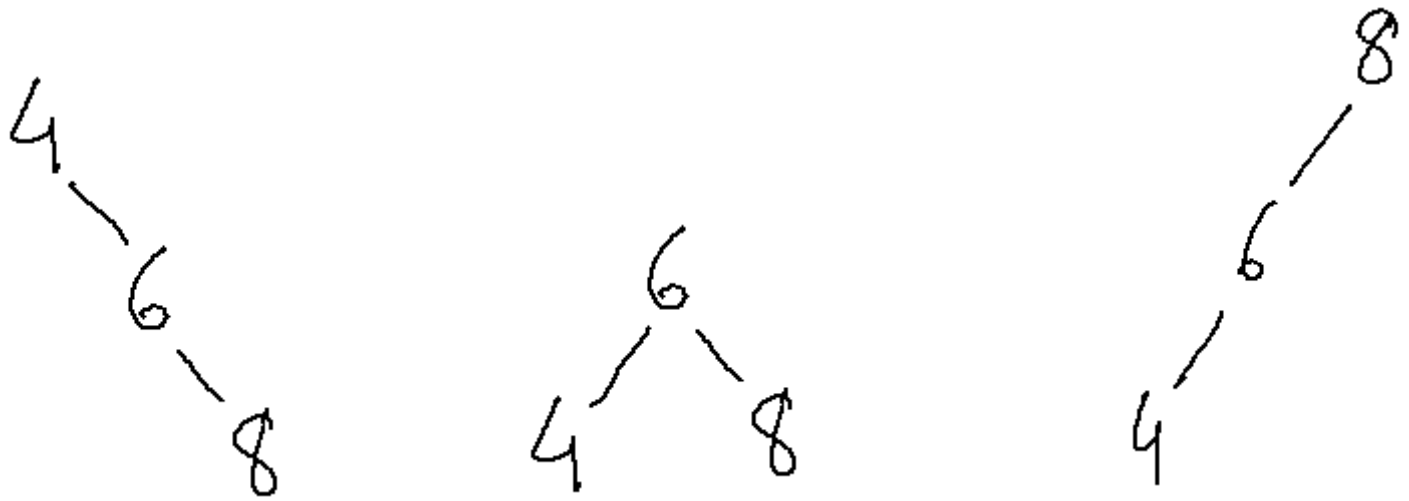


-1-1=-2 requires left rotation
Single or double rotation?



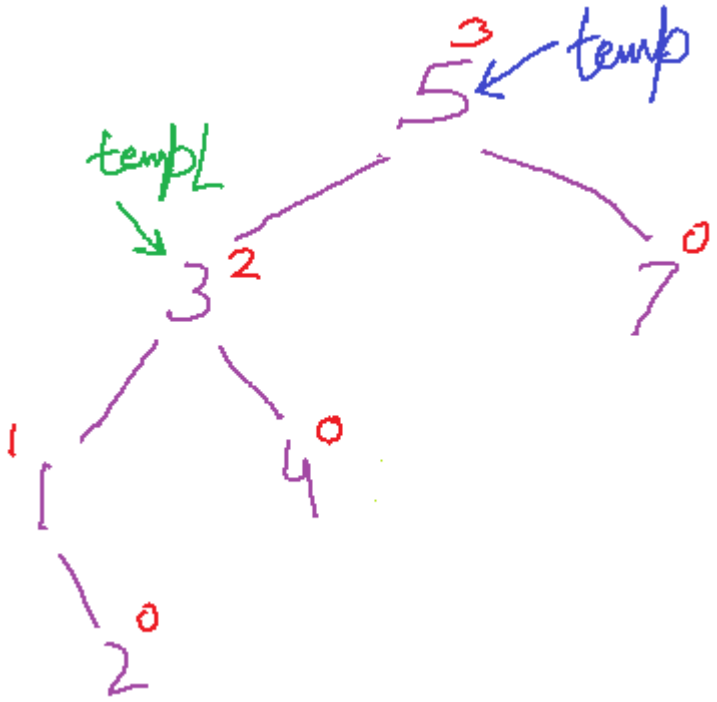
-1-0=-1, compare -2 above and -1 here, both have same signs, therefore single rotation required

AVL Single Left & Right Rotation



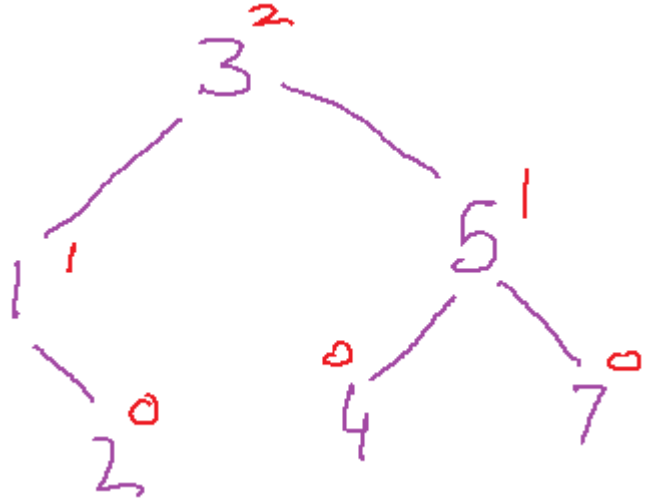
On left and right two un-balanced trees given, whereas left and right rotation of both will give balanced tree in the middle

AVL Left Rotation with Node Height Plus Code

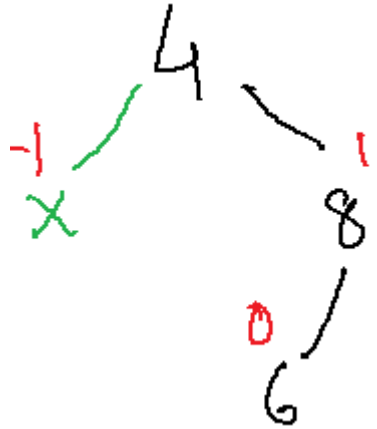



```
tempL = temp -> left;
temp -> left  = tempL -> right;
tempL -> right = temp ;
Temp -> height = temp -> height -1;
return temp;
```

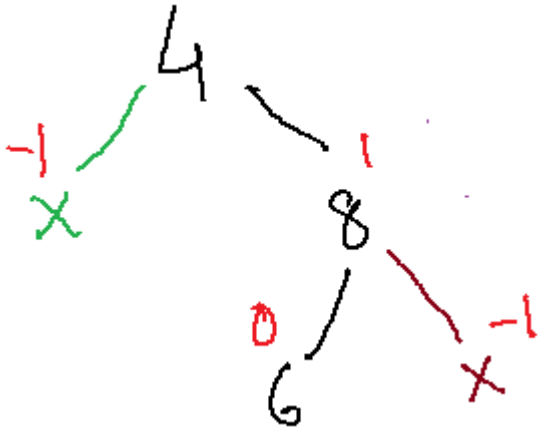
We will add one more statement later



AVL Double Rotation

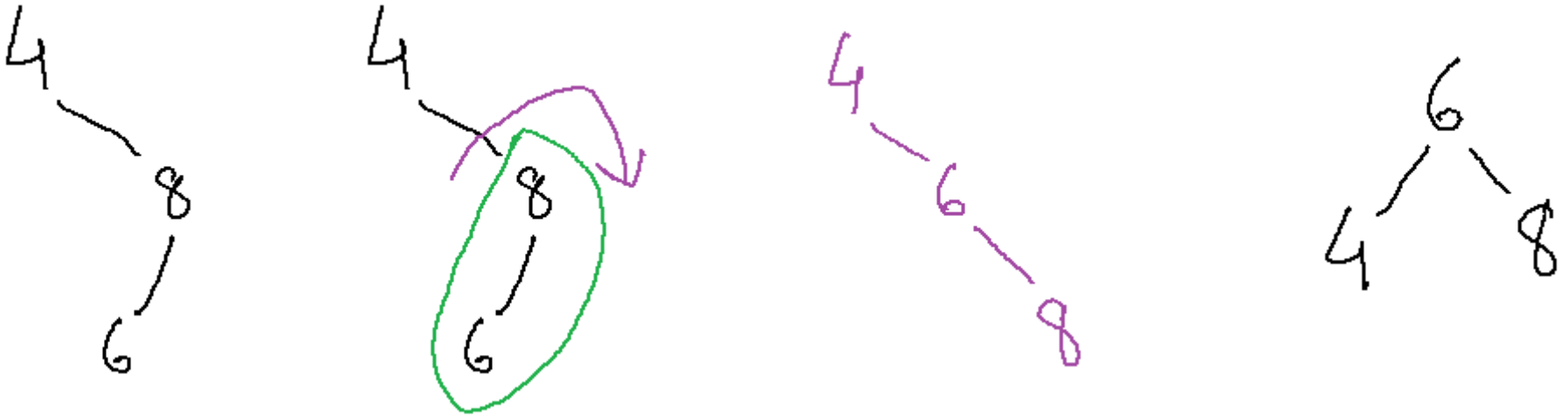


-1-1=-2 requires left rotation again
Single or double rotation?



0-(-1)=+1, compare -2 above and +1 here, they have different signs, therefore double rotation required

AVL Double Rotation

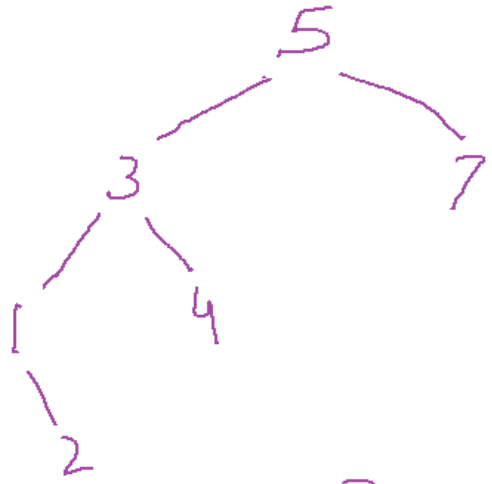


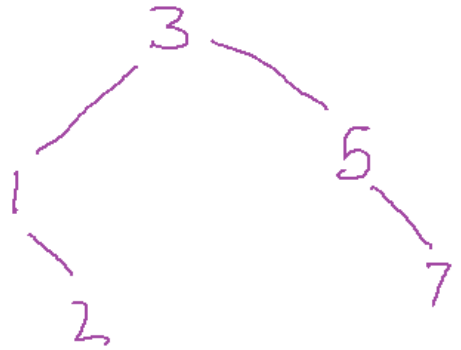
Consider first figure on left hand side, requires double rotation to maintain binary search property of tree. Double rotations are always required in opposite direction. That is if left rotation is required (at Pivot as in above example tree) then right rotation is required at node on right of pivot node, hence you can see in third figure (from left to right) right rotation is performed. Finally, in last right most figure left rotation is performed and the resultant tree is height balanced binary search tree (AVL)

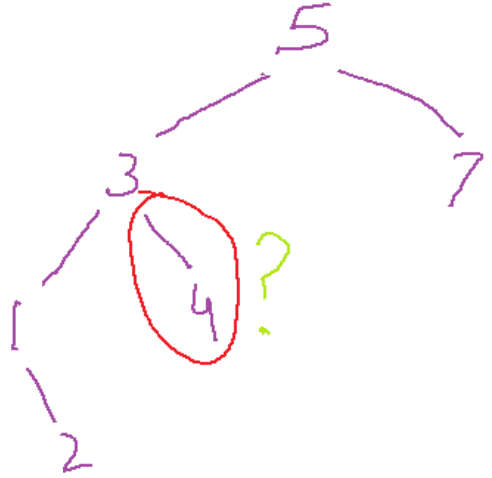
Complete Rotation Steps

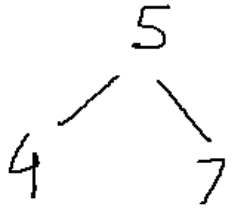
In previous slides, we have tried to give simple examples to understand rotation. We may have a AVL tree on upper left most figure, where 2 is inserted and node 5 (may be root, may have more nodes on top) is now pivot node. Here, if we simply assign node 5 as right child of node 3, node 4 will be logically removed from tree. Therefore we will do following steps:

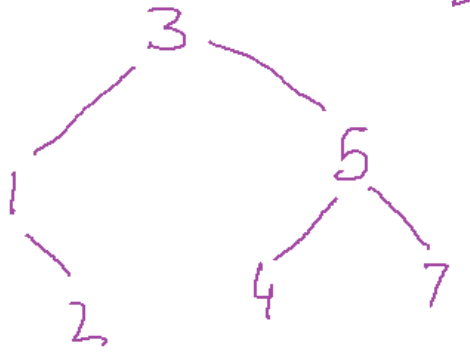
1. Assign node 3 (left of pivot node, case for right rotation) to some pointer
2. Make node 4 as left child of node 5 (see small figure with black colour)
3. Assign node 5 as left child of node 3 & return node 3 (it may replace as left or right child for previous parent of node 5



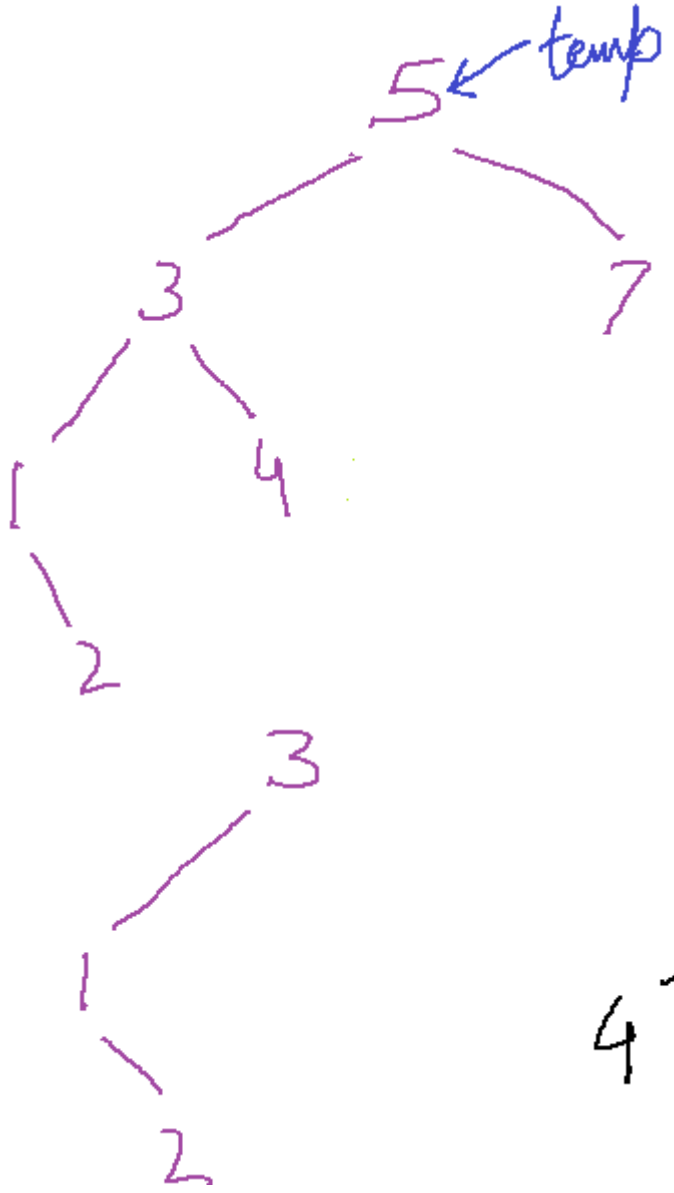


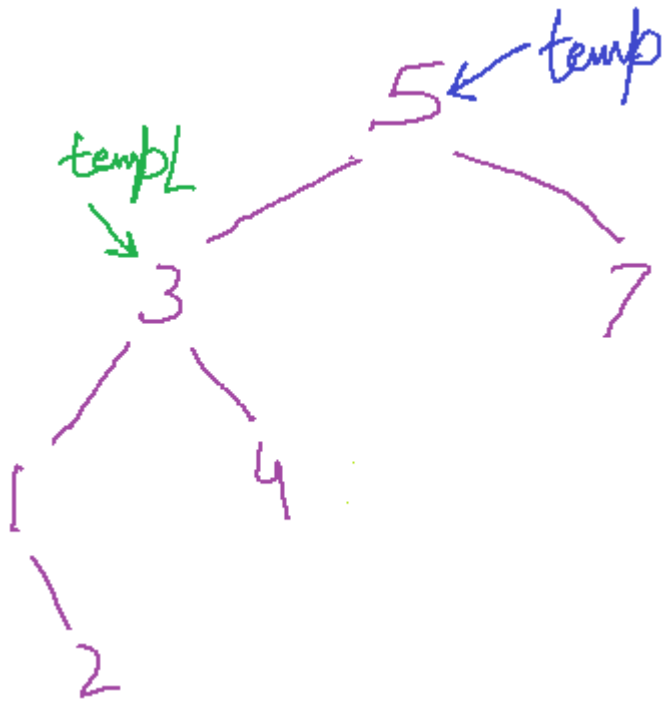






Code Right Rotation






```
templ = temp -> left;
temp -> left  = templ -> right;
templ -> right = temp ;
return templ;
```

We will add one more statement later in code