Data Structure
Chapter 4:Tree

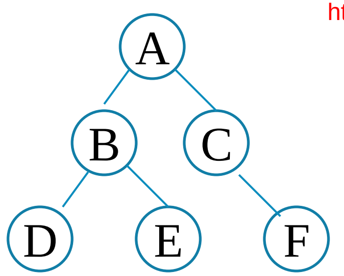
GATE CS Lectures

By

Monalisa Pradhan

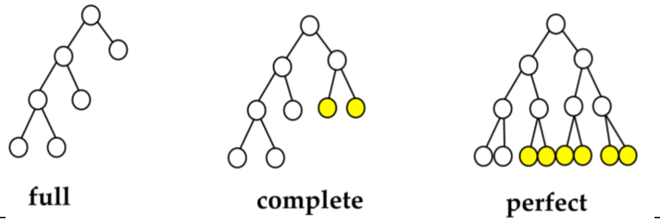
- **Section 4: Programming and Data Structures**
  Programming in C. Recursion.Arrays, stacks, queues, linked lists, trees, binary search trees, binary heaps, graphs.

- Chapter 1:Arrays
- Chapter 2: stacks, queues
- Chapter 3: linked lists
- Chapter 4: trees(tree traversal), binary search trees,AVL tree,
- Chapter 5: graphs

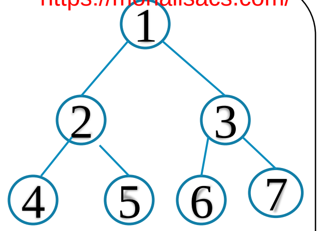


- **Tree:** It is a non linear data structure.
- Root: parent of all node, or level 0 node .
- Nodes , Edges
- Parent node ,Child node
- Leaf node, Non leaf node
- Internal node, external node
- Path: sequence of consecutive edges from source to destination
- Ancestor or Predecessor
- Descendant or Successor
- Subtree: subpart of tree.
- Sibling: children of same parent
- Degree :number of edges connected with that node.
- Height: longest path from root to leaf
- Level : nodes at same height
- Depth :length of path from root to that node
- Forest : collection of tree.
- A tree with n vertices has (n-1) edges.

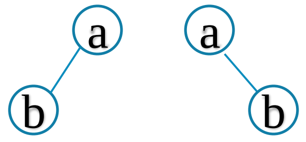
- **Binary Tree**
- A tree having at most 2 children is called a binary tree.
- Number of distinct binary tree can form $= {}^{2n}c_n/(n+1)$.
- Maximum number of node at ith level is 2^i node.
- Maximum number of nodes in a binary tree of height h is, $2^{h+1} - 1$.
- Minimum number of nodes in a binary tree of height h is h + 1 .
- Maximum height of n node is n-1.
- Minimum height of n node is ceil(log(n+1)-1).
- If every node has either 0 or 2 children, a binary tree is called **full.**
- If the lowest d-1 levels of a binary tree of height d are filled and level d is partially filled from left to right, the tree is called **complete**.
- If all d levels of a height-d binary tree are filled, the tree is called **perfect**
- A full binary tree with n internal nodes has n+1 external nodes.
- A binary tree with n leaf have exactly n-1 nodes having two children



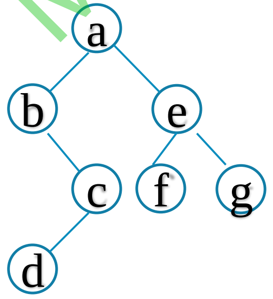
- **Tree Traversal:**
- InOrder :Traverse left subtree ,Visit Root, Traverse Right subtree
- PreOrder : Visit Root ,Traverse left subtree ,Traverse Right subtree
- PostOrder:Traverse left subtree ,Traverse Right subtree ,Visit Root
- InOrder :4 ,2 ,5 ,1 ,6 ,3 ,7
- PreOrder:1 ,2 ,4 ,5 ,3 ,6 ,7
- PostOrder:4 ,5 ,2 ,6 ,7 ,3 ,1
- Level Order traversal :Traverse level wise, Ex:1 ,2 ,3 ,4 ,5 ,6 ,7



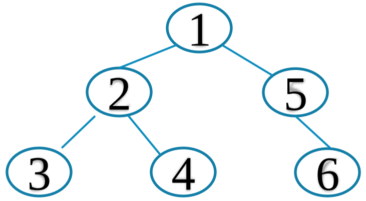
- **Construct Binary Tree from InOrder , PreOrder , PostOrder traversal :**
- To construct unique binary tree there most be InOrder with PreOrder or PostOrder.
- PreOrder: a,b PostOrder :b,a



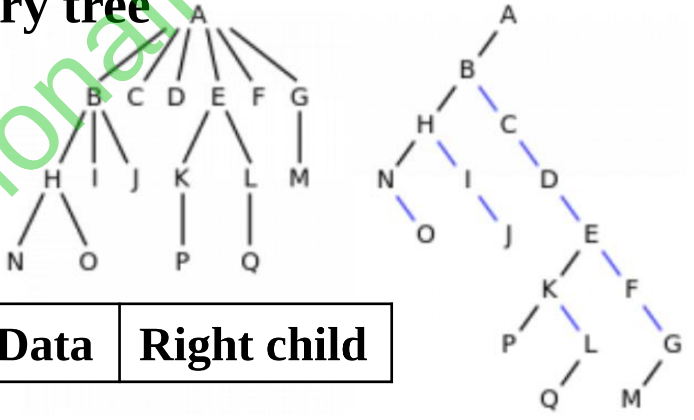
- Construct binary tree from preorder and inorder traversal
- Preorder: a ,b ,c ,d ,e ,f ,g
- Inorder : b ,d ,c ,a ,f ,e ,g
- Postorder:d ,c,b,f,g,e,a



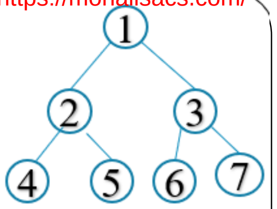
- Construct binary tree from postorder and inorder traversal
- Postorder: 3 ,4 ,2 ,6 ,5 ,1
- Inorder :   3 ,2 ,4 ,1 ,5 ,6
- Preorder:1 ,2 ,3 ,4 ,5 ,6



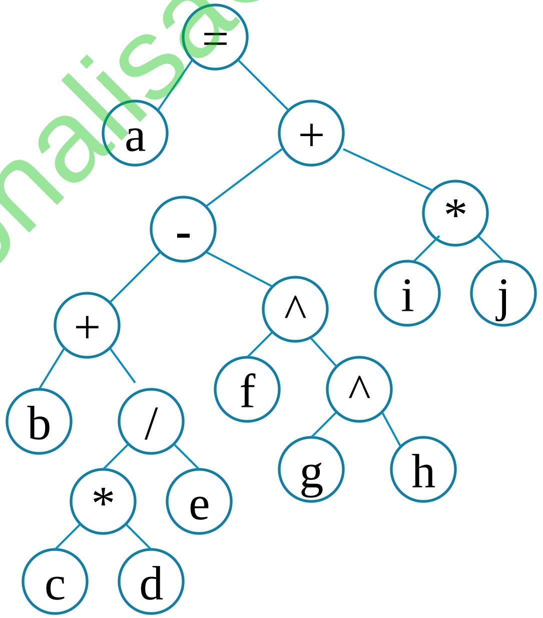
- **LeftMostChild-rightSibling representation**
- It is a different representation of an n-ary tree where instead of holding a reference to each and every child node, a node holds just two references, first a reference to it's first child, and the other to it's immediate next sibling.
- **Self referential structure of binary tree**
- *struct BTnode*
- *{ struct BTnode *leftchild;*
- *char data;*
- *struct BTnode *rightchild; };*
- *Struct Btnode *T*
- Access T→ data ,
- T → leftchild(Descendant left),
- T → rightchild(Descendant right)



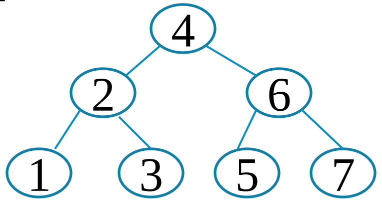
Left child	Data	Right child



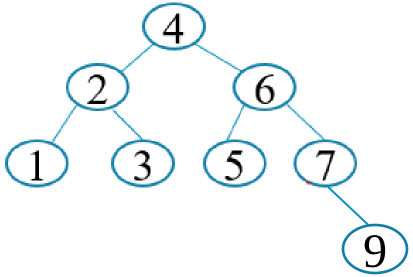
- Program for finding number of terminal node
- *Int do(struct Btnode *T)*
- *{if(!T) return 0;*
- *if (T→ lc = null && T → rc=null)*
- *return 1;*
- *else*
- *return do(T → lc)+do(T → rc)*
- *}*
- **Binary Tree Applications:**
- Arithmetic Tree or expression tree
- Parent: Operator ,Child =Operand
- Example a=b+c*d/e-f^g^h+i*j
- **Precedence   Associatively**
- (),[],{}
- ^   Right-Left
- *,/   Left-Right
- +,-   Left-Right



- **Binary search tree**
- A Binary Search Tree (BST) is a Binary tree in which all the nodes follows
- The value of the key of the left sub-tree is less than the value of its parent (root)
- The value of the key of the right sub-tree is greater than or equal to the value of its parent (root)
- *left_subtree (keys) < node (key) ≤ right_subtree (keys)*
- Traverse inorder get sorted list.
- **Create BST:** Very first node is root , for other node start comparing with root ,find proper position and insert.
- **Insert:** first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the proper location in the left subtree and insert the data. Else search for the proper location in the right subtree and insert the data
- Example : 4 ,2 ,3 ,6 ,5 ,7 ,1
- InOrder: 1 ,2 ,3 ,4 ,5 ,6 ,7

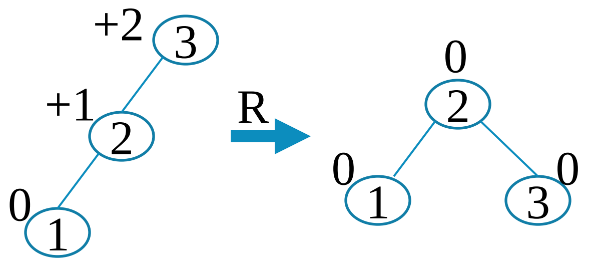


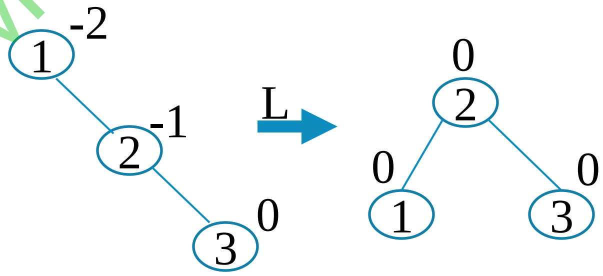
- **Delation:**
- After delation the order must remain same,the tree can be modified but not the order.
  - 0 children :delate it.
  - One children or one subtree :connect to grand parent .
  - Both children or both subtree: replace with inorder successor or predecessor .
- The time efficiency of searching, insertion, and deletion,which are all in *Θ(*log *n)*, but only in the average case.
- In the worst case, these operations are in *Θ(n)* because the tree can degenerate into a severely unbalanced one with its height equal to *n*-1.
- BST of inserting 4 ,2 ,3 ,6 ,5 ,7 ,1, 9 , Delate 5,7,4



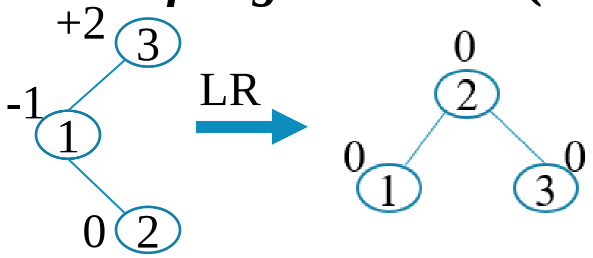
- **AVL Trees**
- AVL trees were invented in 1962 by two Russian scientists, G. M. **A**delson**V**elsky and E. M. **L**andis .
- An ***AVL tree*** is a binary search tree in which the *balance factor* of every node, which is defined as the difference between the heights of the node's left and right subtrees, is either 0 or +1 or -1.
- A ***rotation*** in an AVL tree is a local transformation of its subtree rooted at a node whose balance has become either +2 or -2.
- There are four types of rotations.

1 .*Single right rotation, or R-rotation* ***2****.single left rotation, or L-rotation*

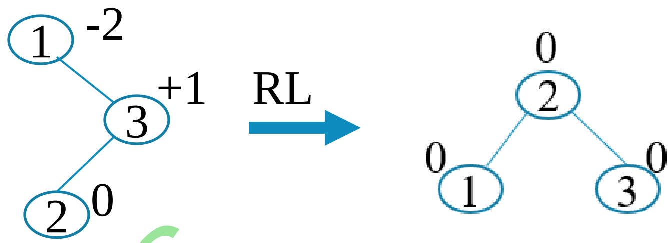




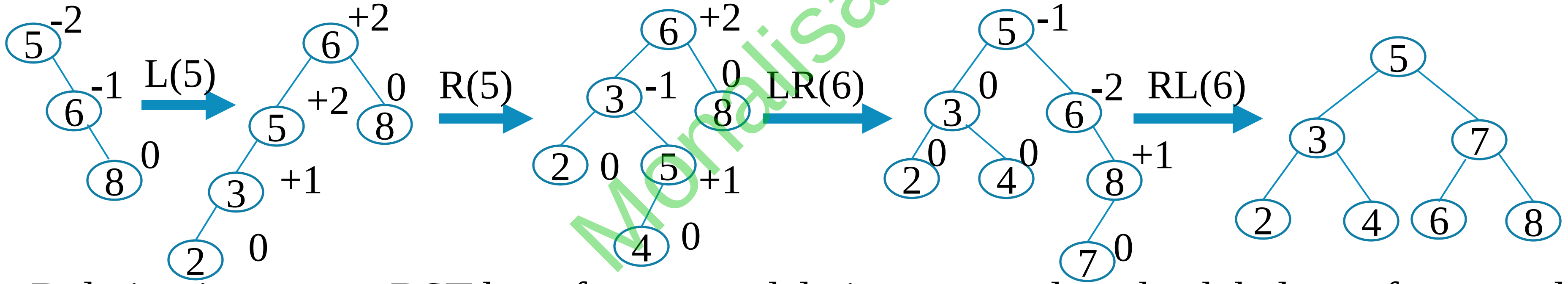
3.double left-right rotation (LR-rotation)



4.double right-left rotation (RL-rotation)



- The operations of searching and insertion and deletion are Θ(log n) in the worst case.
- Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions



- Deletion is same as BST but after every deletion we need to check balance factor and balance height by rotation .
- The drawbacks of AVL trees are frequent rotations and the need to maintain balances for its nodes.