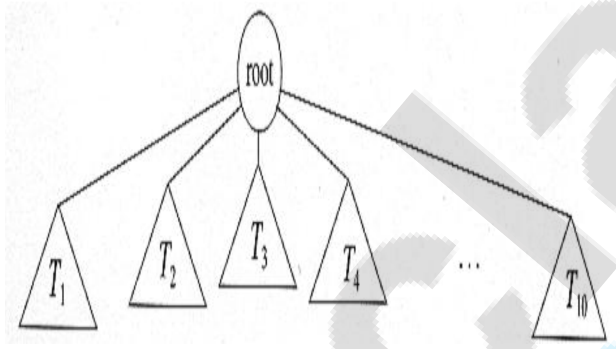
Data structure **Sem II Batch A** **By . Rupali Jadhav.**
Academic year 2016-17
Unit 5 Non Linear Data Structure

==

Trees/ General Trees

- Non linear data structure
- A tree is either
  - empty (no nodes), or
  - If not empty, a tree consists of a distinguished root node

(r), and zero or more nonempty sub trees T_1, T_2, ..., T_k



Terminology:

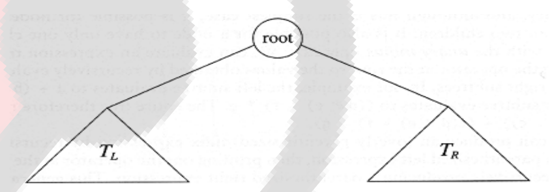
- Root
  - Base node (vertex) of the tree with no predecessor
- Leaves
  - Nodes with no children / Nodes (i.e. with outdegree zero)
- Internal nodes
  - Nodes that are not root or leaf
- Indegree
  - Number of edges (branches) arriving at that vertex.
- Outdegree
  - Number of edges leaving that vertex
- Degree
  - Number of branches associated with a node
  - Parent

==

    - Node with out degree greater than zero. Every node except the root has one parent .
    - Child
    - A node which has a predecessor
    - Sibling
    - Nodes with same parent
    - Path
    - Sequence of nodes in which each node is adjacent to next
- Ancestor and descendant
    - If a path exists from node p to node q, where node p is closer to the root node than q, then p is an ancestor of q and q is a descendant of p.
- Level of a node
    - Distance of node from root. Root at level 0.
    - Depth of a tree
    - Maximum level of any leaf in a tree.
- Height of tree
    - The height of a tree is the level of the leaf in the longest path from the root plus 1. Height of empty tree is -1.
    - It is a maximum number of nodes in longest path of tree.

Binary Trees:

- A tree in which no node can have more than two children

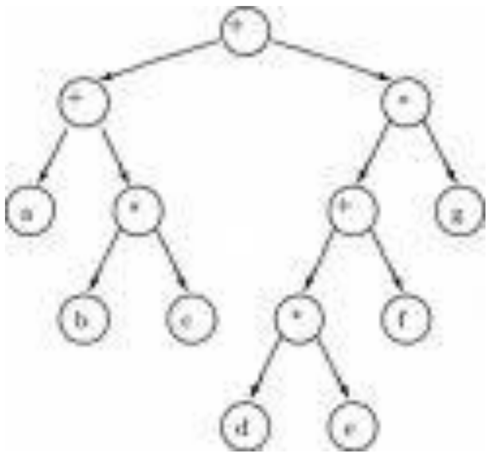


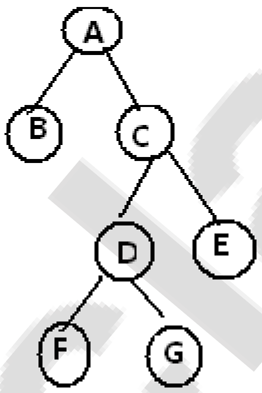
- M-ary Tree
    - A tree in which no node can have more than M children

===

Strict (Full) Binary tree:

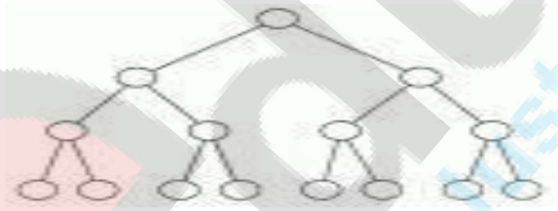
- If every non terminal node consist of non empty left and right sub tree or children, then such a tree is known as strict binary tree.
- Every node has zero or two children.
- n leaves => 2n-1 nodes.





Complete (perfect) binary tree:

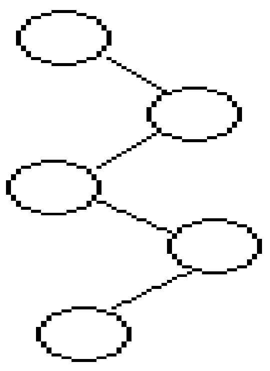
- Strict binary tree in which all leaves are at the same depth d. Total no. of nodes $2^{d+1}-1$. For every level d, 2^{d} nodes.



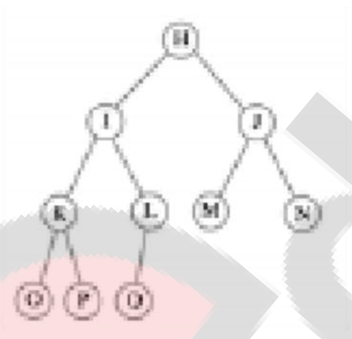
- **Degenerate tree :**

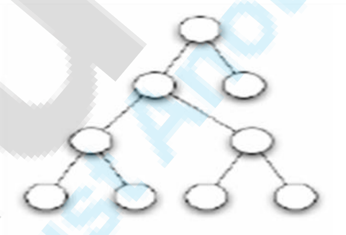
Every node has only one child. It behaves like a linked list data structure.

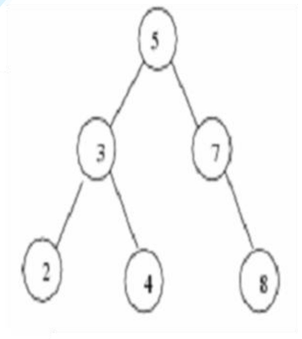
==



- **Almost complete binary tree:**
    - The tree is almost complete if all its levels, except possibly the last, has max. no. of nodes, and if all nodes at the last level appear as far left as possible.
    - Each node that has a right child also has a left child.
    - A node having a left child does not require a node to have a right child.
    - Leaves are at level d or d-1







Almost complete binary tree

Not Almost complete binary tree

Storage Representation of Binary Tree:

Q. Write a short note on Storage Representation of Binary Tree?

- Array Representation
- Linked Storage Representation

Array Representation:

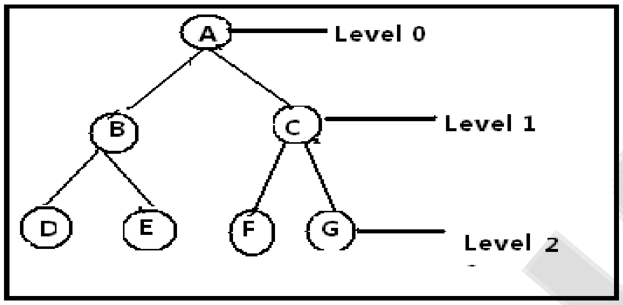
- Suppose h is height of tree

==

- Size of array is 2h-1
    - Root of the tree : At position zero
    - Left child: At position 1
    - Right child : At position 2
- Left child of node at position k is at array position

  (2* array index of parent node +1)
- Right child of node at position k is at array position

(2* array index of parent node +2)



0	1	2	3	4	5	6
A	**B**	**C**	**D**	**E**	**F**	**G**

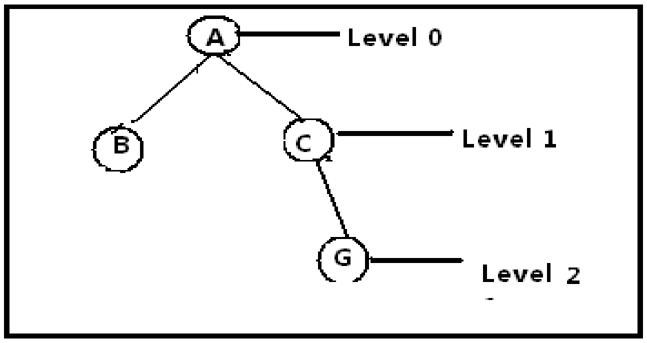
Disadvantage:

- Using this strategy , tree with N nodes does not necessarily occupy the first N position in the array.

Its lead to wastage of memory location.

- For example ,

===

- 

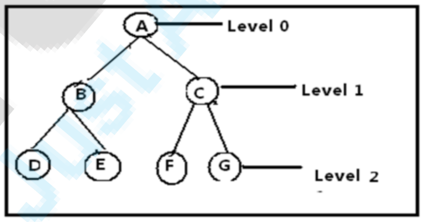
0	1	2	3	4	5	6
A	B	C	–	–	–	G

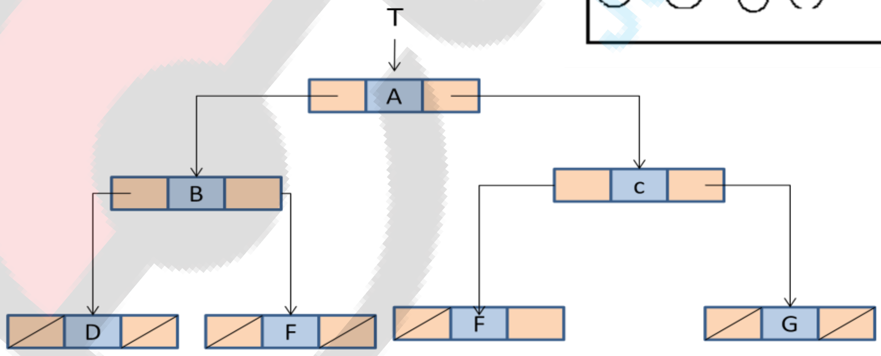
Linked list representation:

- We used node to represent tree using linked list.
- Each tree node can have left or right or both sub tree.

Structure of node is

Pointer to left child	Data	Pointer to right child





Binary Tree Traversal:

1. Depth First Traversal

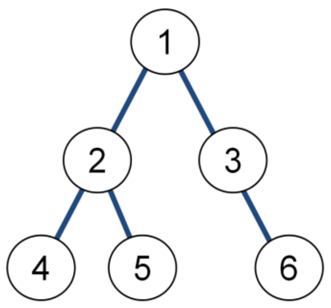
==

1. Pre order
2. In order
3. Post order

2. Breadth First Traversal :Traverse the tree horizontally from level 0 to last level in tree.

- **Preorder:**

1. **Visit the node.**
2. **Traverse the left subtree in preorder**
3. **Traverse the right subtree in preorder**



Root →Left→ Right

preorder : 1 2 4 5 3 6

Algorithm preOrder (root <tree ptr>)

Pre: root points to root of the tree

Return : Traverses the tree in preorder

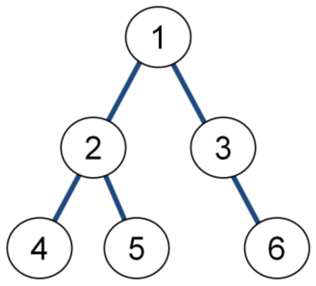
```
1. if (root <>NULL)
        1. process(root->data)
        2. preOrder(root->left)
        3. preOrder(root->right)
2. return
```

Inorder:

1. **Traverse the left subtree in inorder.**

==

2. **Visit the node.**
3. **Traverse the right subtree in inorder.**



Left→ Root → Right

Inorder : 4 2 5 1 3 6

Algorithm inOrder (root <tree ptr>)

Pre: root points to root of the tree

Return : Traverses the tree in inorder

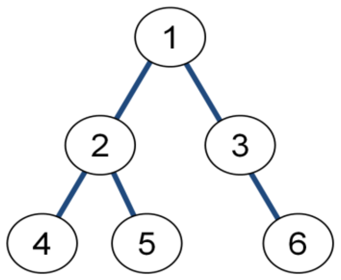
```
1. if (root <>NULL)
      1. inOrder(root->left)
      2. process(root->data)
      3. inOrder(root->right)
2. return
```

Postorder:

1. **Traverse the left subtree in postorder.**
2. **Traverse the right subtree in postorder.**
3. **Visit the node.**

==

Left→ Right → Root



Postorder : 4 5 2 6 3 1

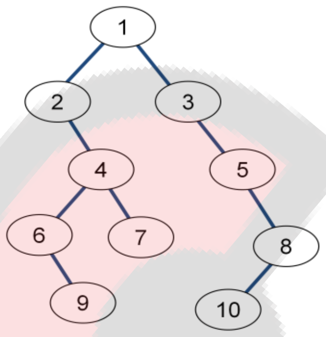
Algorithm postOrder (root <tree ptr>)

Pre: root points to root of the tree

Return : Traverses the tree in postorder

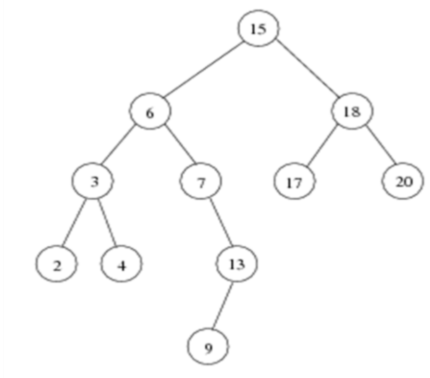
```
1. if (root <>NULL)
    1. postOrder(root->left)
    2. postOrder(root->right)
    3. process(root->data)
2. return
```

Q. Traverse the following binary tree in preorder, inorder and in postorder.

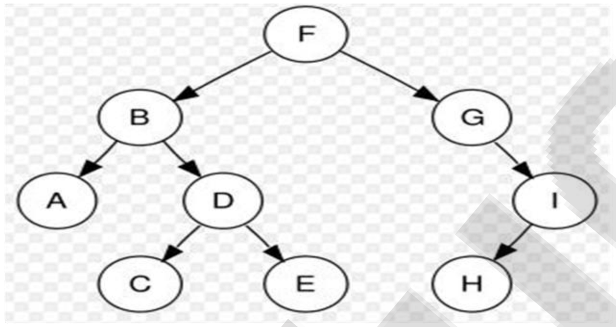


preorder : 1, 2, 4, 6, 9, 7, 3, 5, 8, 10
inorder : 2, 6, 9, 4, 7, 1, 3, 5, 10, 8
postorder : 9, 6, 7, 4, 2, 10, 8, 5, 3, 1

===



preorder : 15, 6, 3, 2, 4, 7, 13, 9, 18, 17, 20
inorder : 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20
postorder : 2, 4, 3, 9, 13, 7, 6, 17, 20, 18, 15



preorder : F, B, A, D, C, E, G, I, H
inorder : A, B, C, D, E, F, G, H, I
postorder : A, C, E, D, B, H, I, G, F

Question:

A binary tree has 10 nodes. The inorder and preorder traversal are shown below.

Inorder: A B C E D F J G I H

Preorder: J C B A D E F I G H

Show a step-wise reconstruction of the binary tree along with its postorder traversal.

Answer:

==

Inorder:	A B C E D F J G I H
Preorder:	J C B A D E F I G H

Preorder(tree) = root | preorder(left ST) | preorder(right ST)
Inorder(tree) = Inorder(left ST) | root | Inorder(right ST)
Postorder(tree) = Postorder(left ST) | postorder(right ST) | root

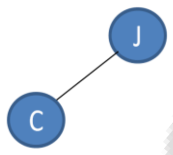
From preorder,
Root: J
Left subtree: A B C E D F
Right subtree: G I H

Preorder making subset [Cleft, Bleft, Aleft, Dleft, Eleft, Fleft, Iright, Gright, Hright]

All lefts are before all right nodes.

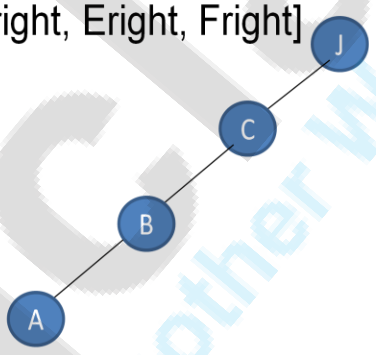
==

For left subtree of J:
Inorder: A B C E D F
Preorder: C B A D E F



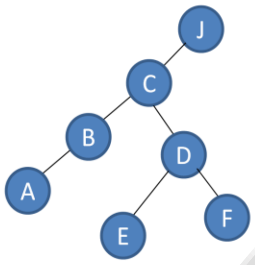
From preorder,
Root: C
Inorder finds
Left subtree: A B
Right subtree: E D F

Preorder making subset [Bleft, Aleft, Dright, Eright, Fright]
All lefts are before all right nodes.
For left subtree of C:
Inorder: A B
Preorder: B A
Root: B
[A] B i.e. A is at the left of B



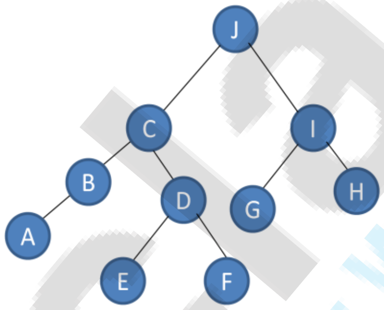
==

For right subtree of C:
Inorder: E D F
Preorder: D E F
Root: D
Preorder making subset [Eleft, Fright]
All lefts are before all right nodes.
[E] D [F]



For right subtree of J:
Inorder: G I H
Preorder: I G H
Root: I
Preorder making subset [Gleft, Hright]
All lefts are before all right nodes.

[G] I [H]



Exercise:

The inorder and postorder traversal of binary tree are as follows.

Inorder: C G A D B J L E H K I F

Preorder: L J G C D A B K H E I F

Draw binary tree and find its postorder traversal.

==

Q. Draw binary tree. Write its post order traversal and ites algorithm.

Preorder: a b d g h e I c f l k

In order : g d h b e I a c j f k

Q. what is binary tree? Construct binary tree.

Inorder: 4 7 2 1 5 3 6

Preorder: 1 2 4 7 3 5 6

==

Q. Construct binary tree. Write its postorder traversal and algorithm.

Inorder: D H B E A F C G

Preorder: A B D H E C F G

Q .Construct binary tree. Write its postorder traversal and algorithm.

Preorder: G B Q A C K F P D E R H

Inorder: Q B K C F A G P E D H R

==

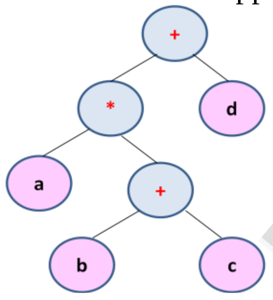
Q . Define expression tree? Draw tree and find prefix and postfix expression for given postfix expression?

(S + P) * (Z- W + D / X)

Ans:

Expression Trees: A special kind of binary tree in which:

1. Each leaf node contains a single operand
2. Each non-leaf node contains a single binary operator
3. The left and right subtrees of an operator node represent sub expressions that must be evaluated before applying the operator at the root of the subtree.



- Infix expression: a * (b + c) + d
  = [a * (b + c)] + d

(S + P) * (Z- W + D / X)

Q. Convert following expressions into expression trees.

X + Y – Z * P ^ 2

==

(A + B * (C / D) – E)* (F/(G + H – J))

23 * 54 - 67 + 89 - 76

General tree:

- It is a tree in which each node can have an unlimited outdegree. Each node may have as many children as necessary.
- It is easier to represent binary trees in program than representing general tree**.**

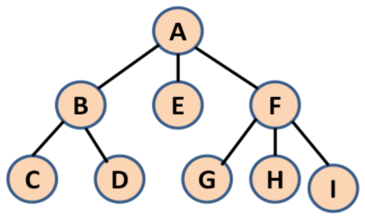
Steps to convert General tree into binary tree

1. Identify branches from parents to their first or leftmost child. These branches become left pointers in the binary tree.

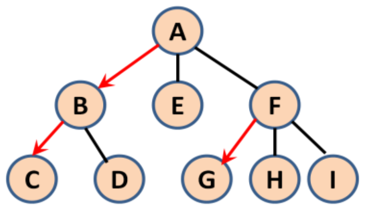
==

2. Connect siblings, starting with the leftmost child, using a branch for each sibling to its right sibling.

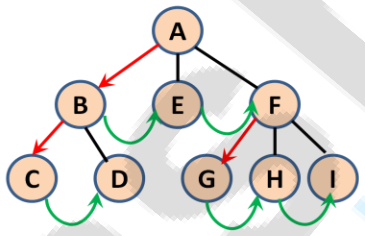
3. Remove all unneeded branches from the parent to its children.



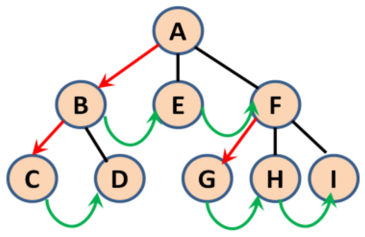
a) The general tree



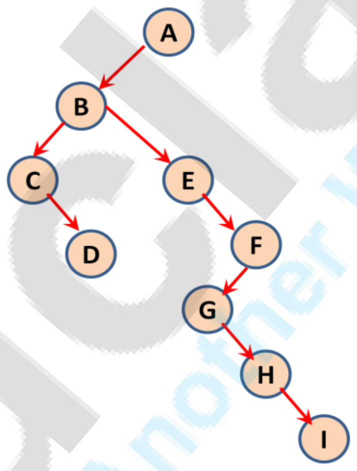
b) Identify leftmost children



c) Connect siblings

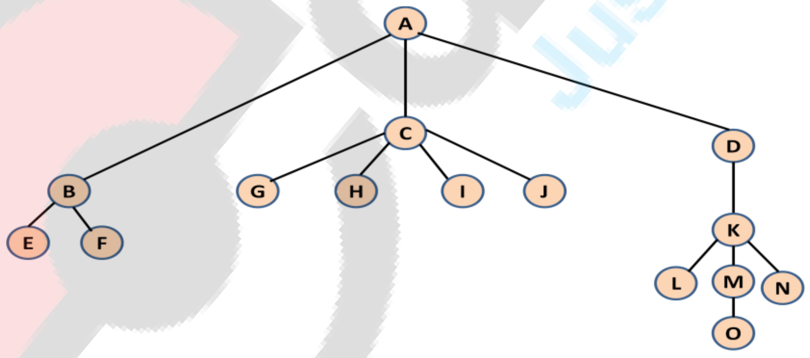


d) Delete unnecessary branches



e) The resulting binary tree

Q. Convert following general tree into binary tree.



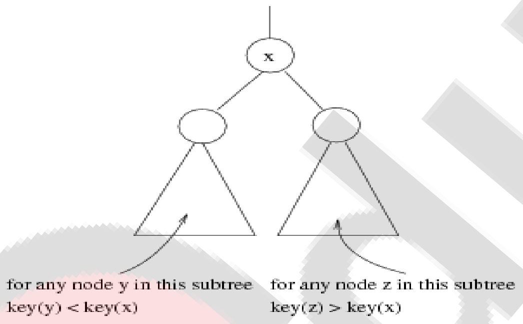
Binary Search Trees:

Define binary search tree (BST) write an algorithm to insert element into binary search tree. Construct BST from following nodes

45 23 29 85 92 7 11 35 49 51

Answer:

For every node X, all the keys in its left subtree are smaller than the key value in X, and all the keys in its right subtree are larger than the key value in X


x
for any node y in this subtree
key(y) < key(x)
for any node z in this subtree
key(z) > key(x)

Algorithm recInsert (root <tree ptr>, new<pointer>)

Pre: root points to root of the tree

Return : new node inserted in tree

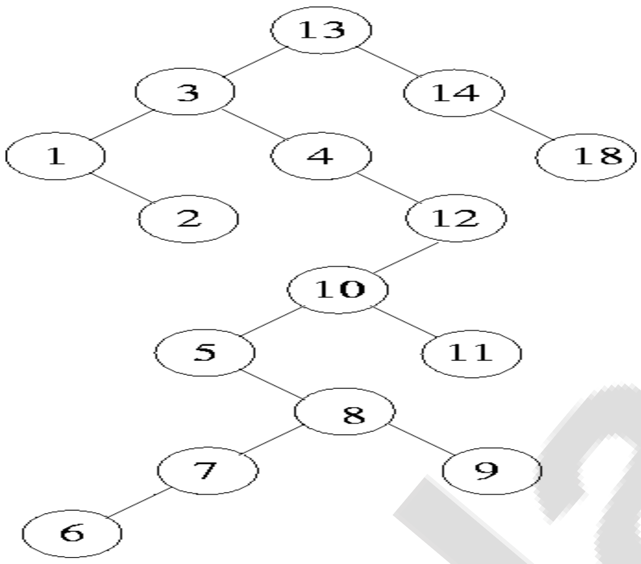
```
1. if (root == NULL)
    1. root = new
    2. root->left = NULL
    3. root->right = NULL
```

```
        4. return
    2. else
        1. if ( new->data < root->data)
            1. recInsert (root->left, new)
        2. else
            1. recInsert (root->right, new)
```

BST: 45 23 29 85 92 7 11 35 49 51

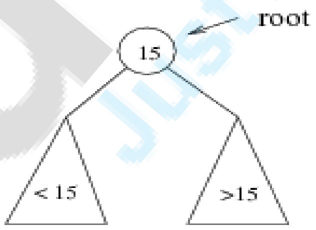
Question: Insert 13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18 in that order, starting from an empty tree.

==

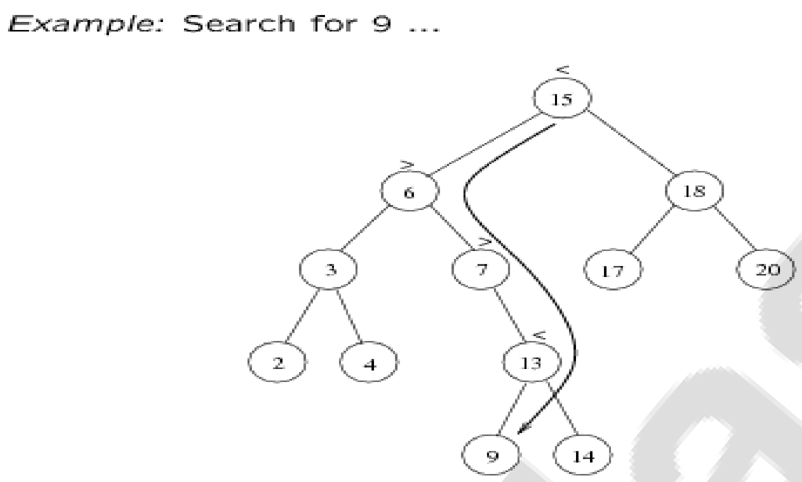


Searching BST:

- ‘If we are searching for 15, then we are done.
- If we are searching for a key < 15, then we should search in the left subtree.
- If we are searching for a key > 15, then we should search in the right subtree.
- Recursively repeat the same process, till you get your key or reach leaf level without finding key.



==



Algorithm recSearch (root <tree ptr>, target<keyType>)

Pre: root points to root of the tree

Return : Node address where key is found, NULL if not found

```
1. if (root == NULL)
        return NULL
2. if (target < root->key)
        return recSearch (root->left, target)
3. else if (target > root->key)
        return recSearch (root->right, target)
4. else
        return root
```

Deletion of node from BST:

1. **Deleting a leaf**

   Simply remove it from the tree.

1. **Deleting a node with one child**

Delete it and replace it with its child.

1. **Deleting a node with two children**

   Suppose the node to be deleted is called N. Replace node N with either.

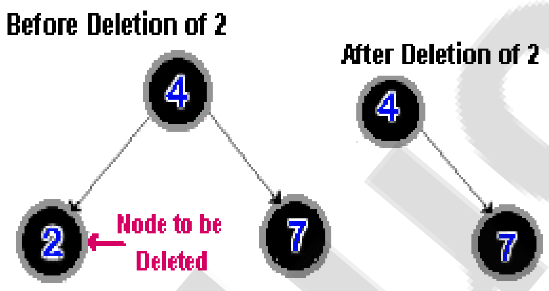
   ✓ In-order successor (the left-most child of the right subtree) i.e. Smallest node in right subtree

or

   ✓ In-order predecessor (the right-most child of the left subtree). i.e. Largest node in left subtree

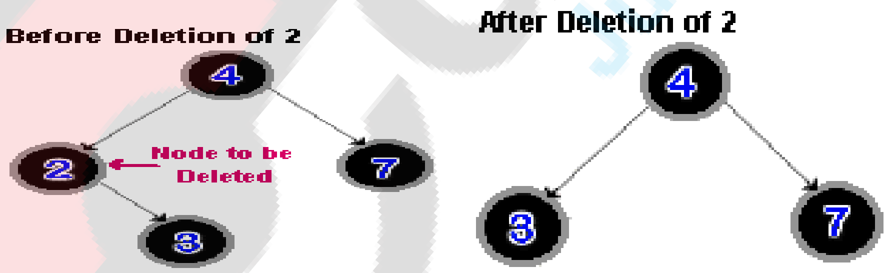
1. The node to be deleted has no children.

   In this case the node may simply be deleted from the tree.



2. The node has one child.

   The child node is appended to its grandparent.

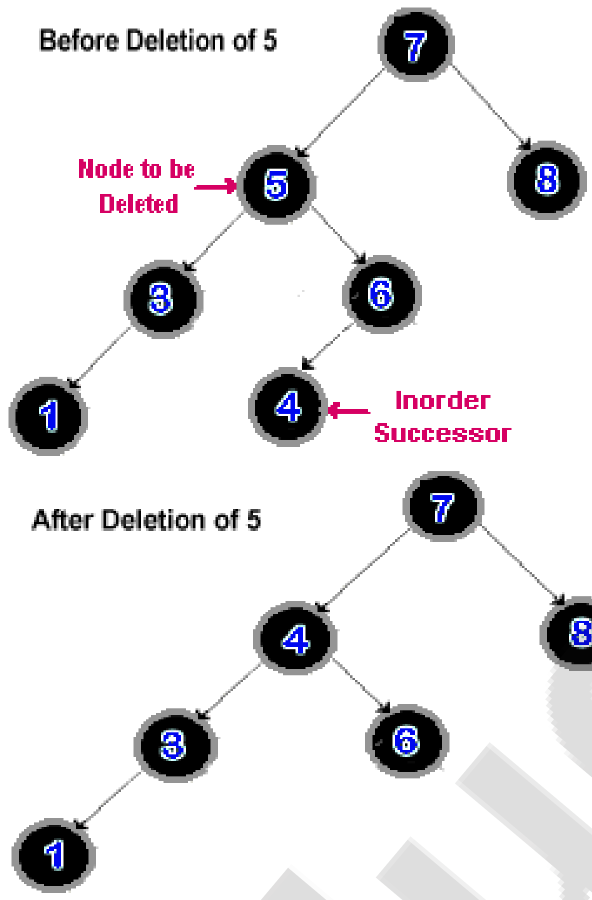


3 The node to be deleted has two children.

a. Replace node to be deleted with its inorder successor.

   ✓ In-order successor (the left-most child of the right subtree) i.e. Smallest node in right subtree

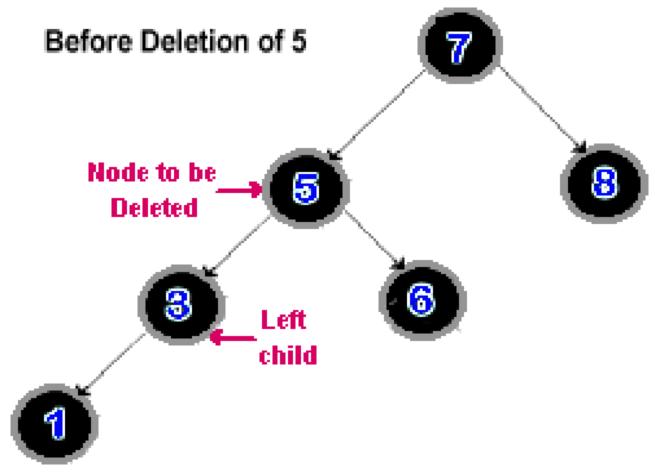
==

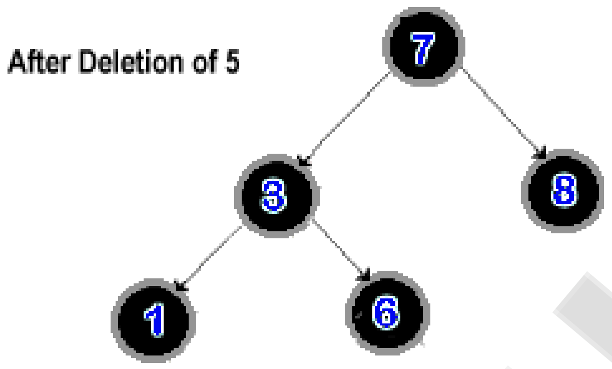


b. Replace node to be deleted with its inorder predecessor.

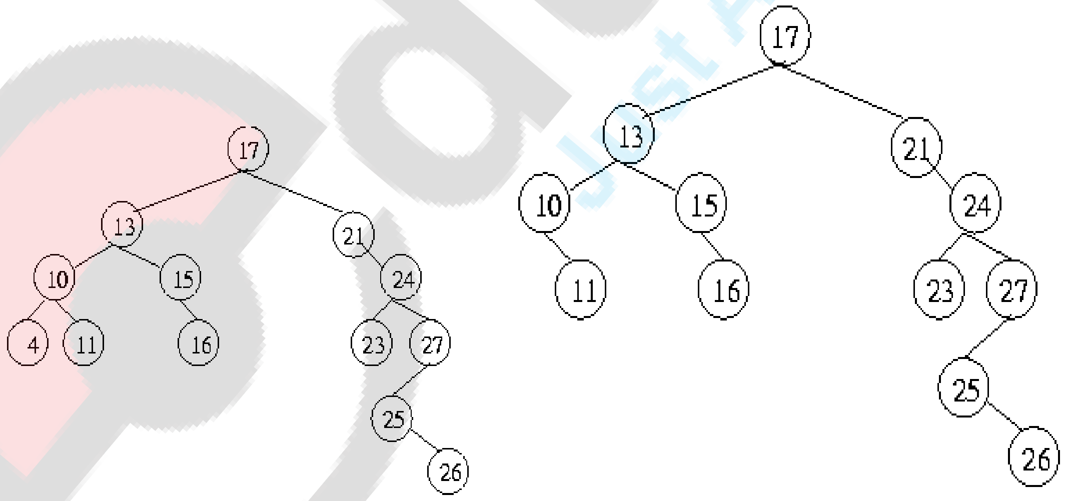
- ✓ In-order predecessor (the right-most child of the left subtree). i.e. Largest node in left subtree

==

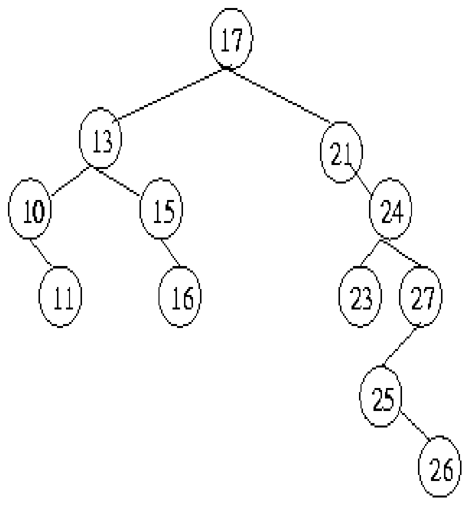


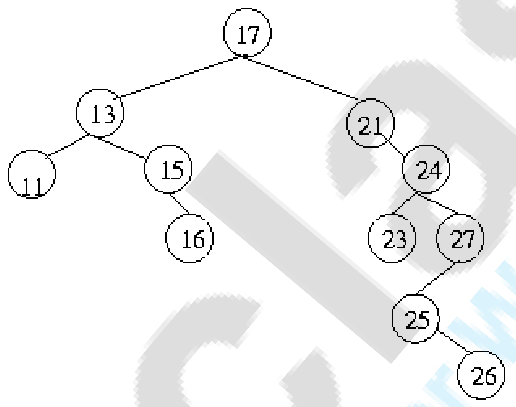


<u>Deletion Exercise</u>: Delete node 4

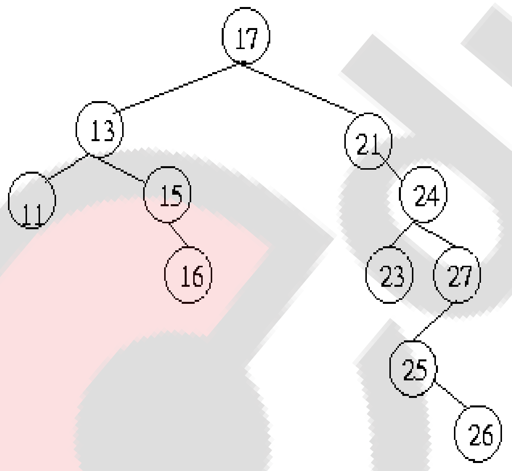


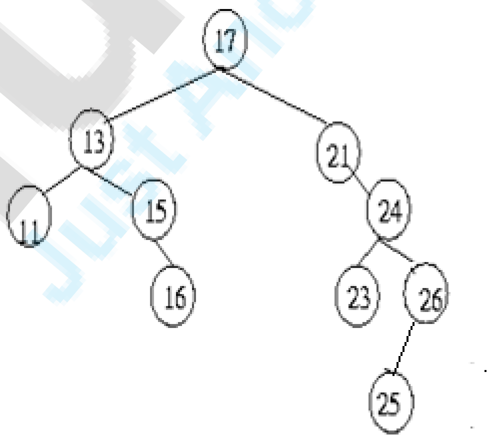
Delete node 10



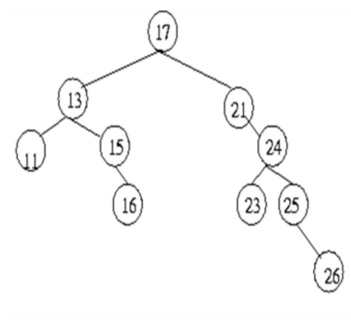


Delete node 27

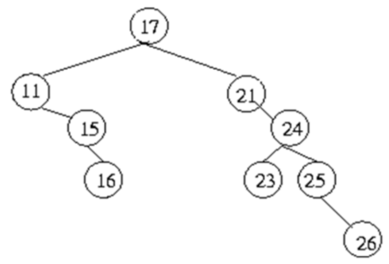


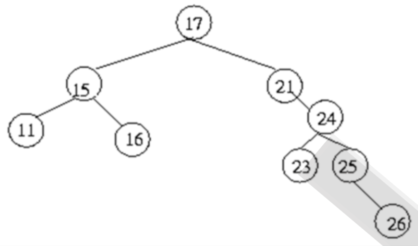


==



Delete 13





Algorithm delete (root <tree ptr>, key<key value>)

Pre: root points to root of the tree, key is data to be deleted

Return : Deletes specified node, if found

```
1. if (root == NULL)
        1. return false
2. if (key < root->data)
        1. return delete (root->left, key)
3. else if (key > root->data)
            1. return delete (root->right, key)
4. else
        1. if (root->left == NULL)   //if node has only right child    //or node
           has no child
                1. p = root
                2. root = root->right
```

===

```
            3.  free p
            4.  return true
    2.  else if (root->right == NULL)    //if node has only left child
            1.  p = root
            2.  root = root->left
            3.  free p
            4.  return true
    3.  else    // find inorder predecessor, replace node to be deleted with
        inorder predecessor value
        1.  p = root->left
        2.  while ( p->right <> NULL)
            1.  p = p->right
        3.  root->data = p->data
        4.  return delete (root->left, p->data)        //delete inorder
            predecessor
```

<u>Minimum and maximum node in BST:</u>

TREE-MINIMUM (x)

```
while x->left <> NULL do
   x = x->left
return x
```

TREE-MAXIMUM (x)

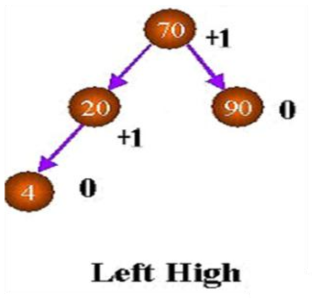
```
while x->right<> NULL do
   x = x->right
return x
```

AVL Trees

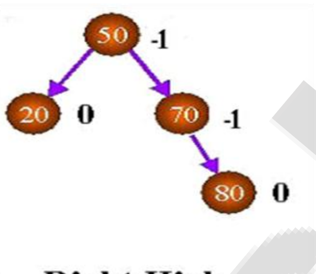
What is a height balanced tree? Why do we require a height balanced tree? Write algorithm for left balancing a tree.

- Height Balanced Tree:
- An AVL tree is a binary tree that has the following properties:
  - The sub-trees of every node differ in height by at most one.
  - Every sub-tree is an AVL tree.
- Height of a tree is number of levels in the tree.
- Balance factor of a node $|H_L - H_R| <= 1$, where H_L, H_R are heights of left and right subtree respectively
- Permissible balance factors:
  - Left-High (balance factor +1)
    The left-sub tree is one level taller than the right-sub tree
  - Balanced (balance factor 0)
    The left and right sub-trees are both the same heights
  - Right-High (balance factor -1)
    The right sub-tree is one level taller than the left-sub tree.
  - If the balance of a node becomes -2 or +2 it will require re-balancing

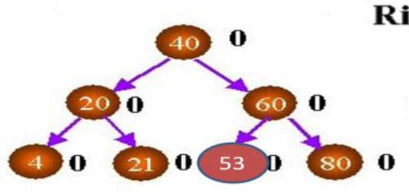
==



Left High



Right High



Balanced

What is the need for balancing a binary search tree? What are the advantages of the AVL tree? Write a pseudo code for Rotate AVL tree right and explain with the help of an example.

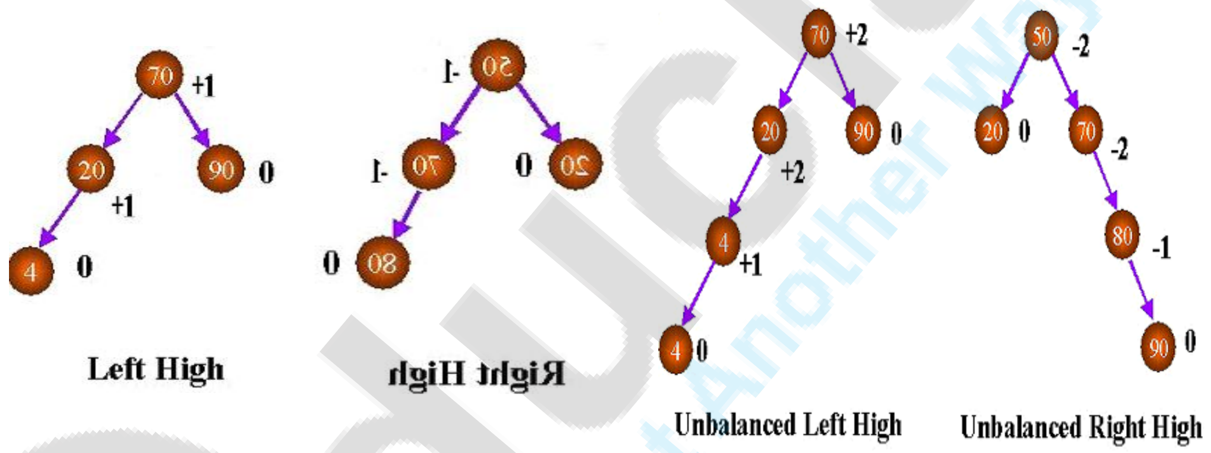
<u>Advantages of Height-Balanced Trees</u>

- If data is inserted in order then tree created is unbalanced and can create a degenerate tree which reduces it to a link list and search efficiency becomes O(n).
- AVL trees solve this problem by ensuring that, for each node, the difference in height between its subtrees (the balance factor) is not greater than 1.
- A rotation is performed whenever the balance factor would increase above 1 retaining O(log(n)) efficiency.
- By balancing a tree whenever an insertion is made, the speed of a subsequent search for an item will be improved. In each iteration of the look up loop the number of elements to be searched through are halved.

<u>Insertion:</u>

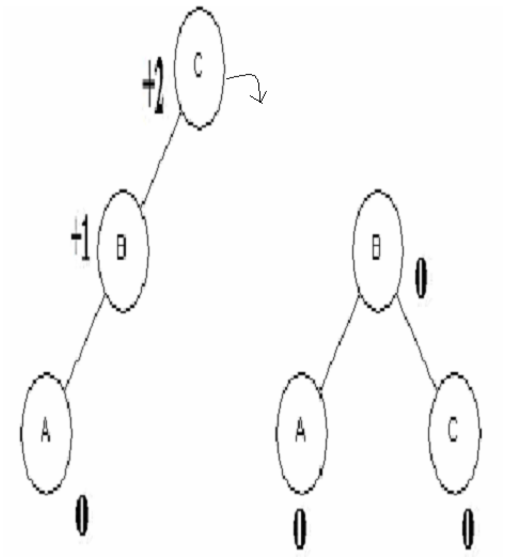
===

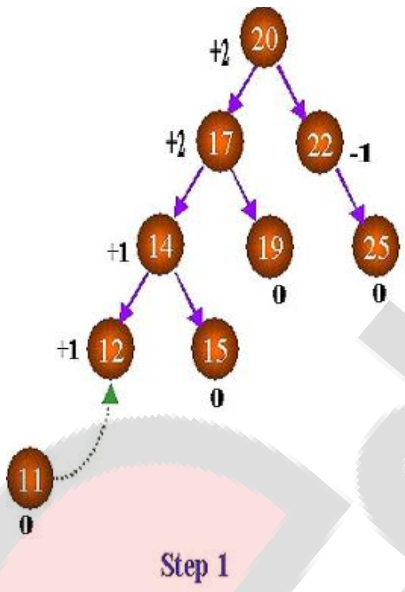
- Process to insert node is same as in an ordinary binary search tree.
- The AVL trees insertion algorithm travels back along the path, in which it took to find the point of insertion, and checks the balance at each node along the path. If a node is found unbalanced (balance factor -2 or +2), rotation is performed based on the position of the inserted node relative to the unbalanced node being examined.
- There will only ever be at most rotation of one node required after an insert operation.

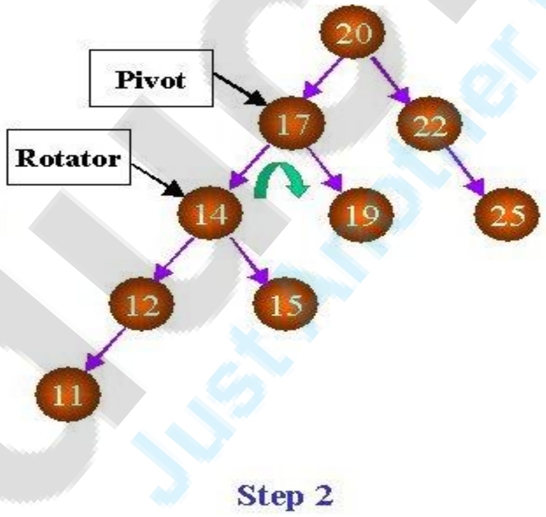


Left of Left – Single Rotation Right:

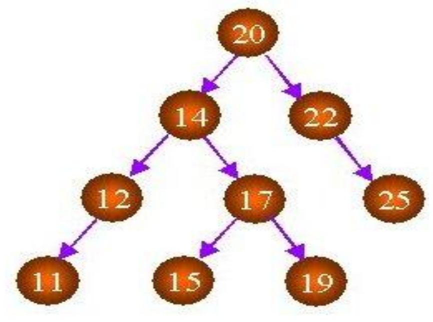
==


+2
C
+1
B
A
0
B
0
A
C
0
0


+2
20
+2
17
22
-1
+1
14
19
25
0
0
+1
12
15
0
11
0
Step 1

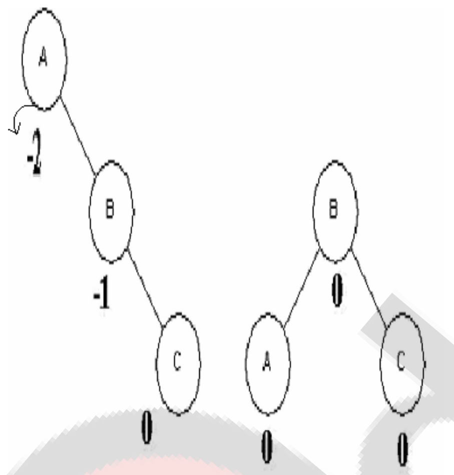

20
Pivot
17
22
Rotator
14
19
25
12
15
11
Step 2

==

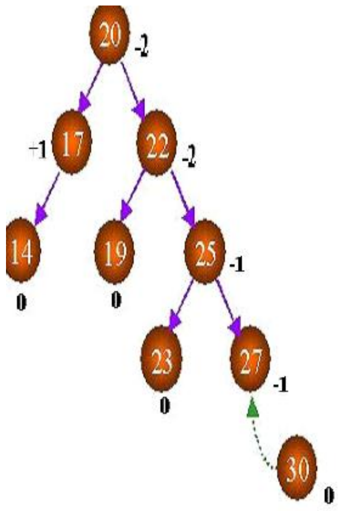


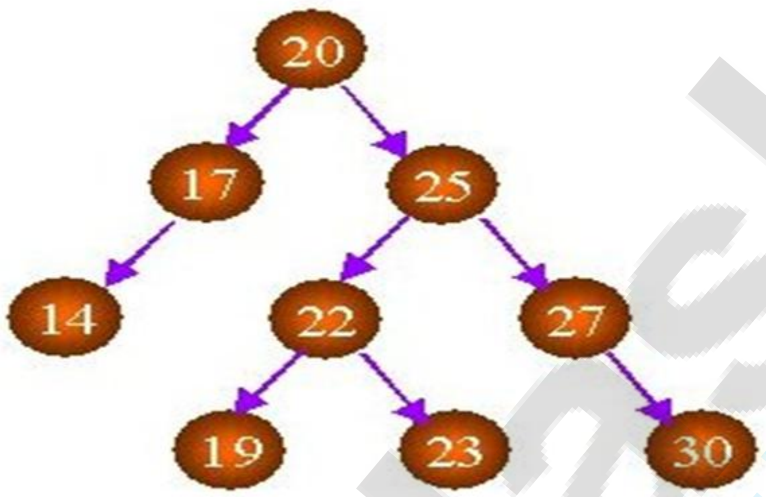
Step 3

Right of Right – Single Rotation Left

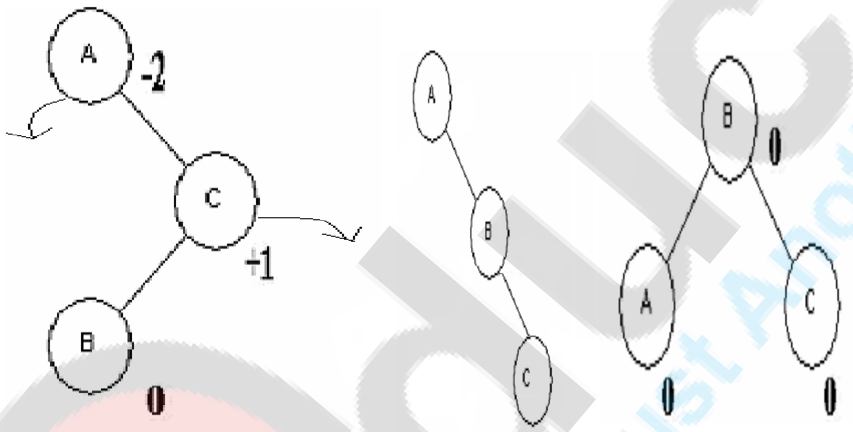


==

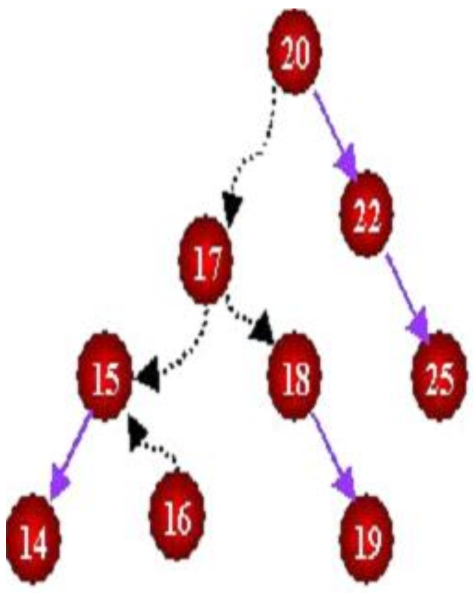


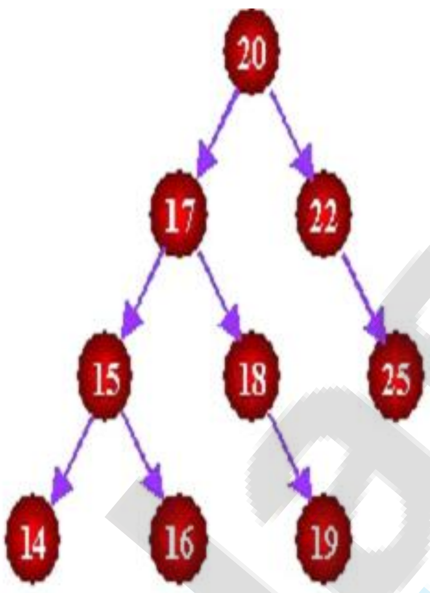


Left of Right - Double Rotation



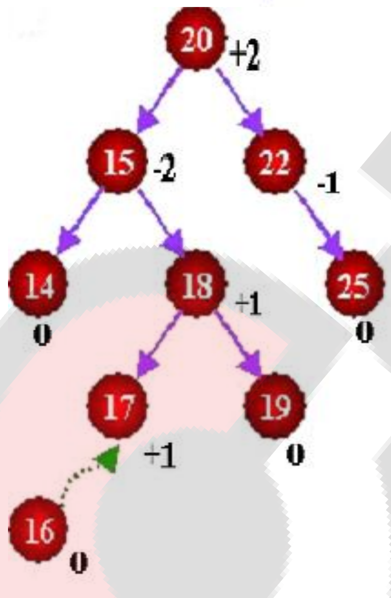
==

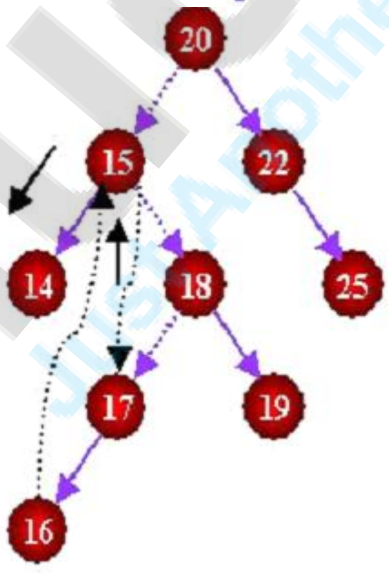




Step 3

Step 4



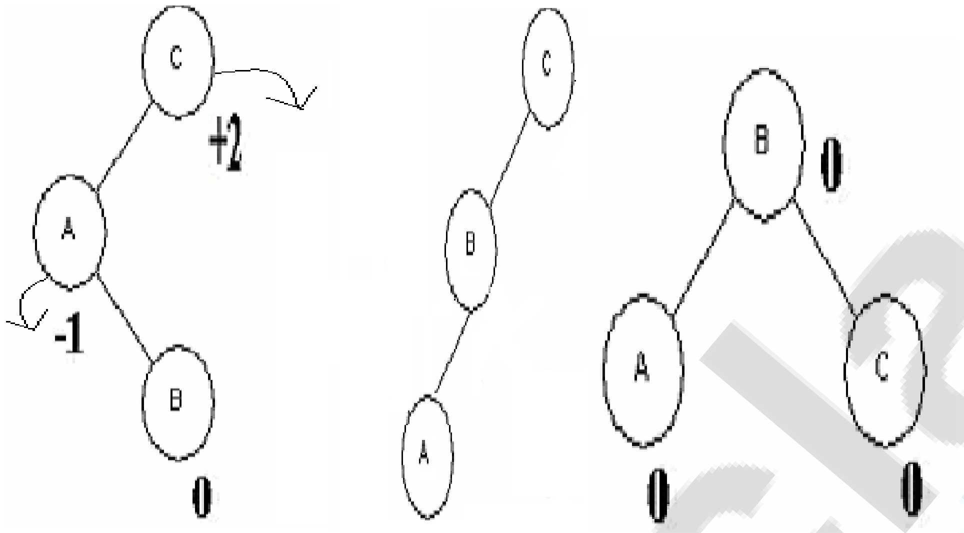


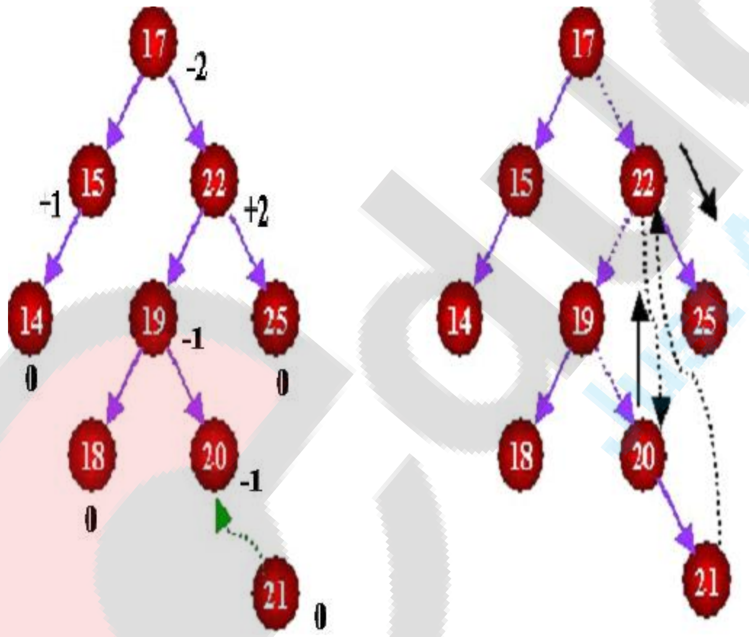
Step 1

Step 2

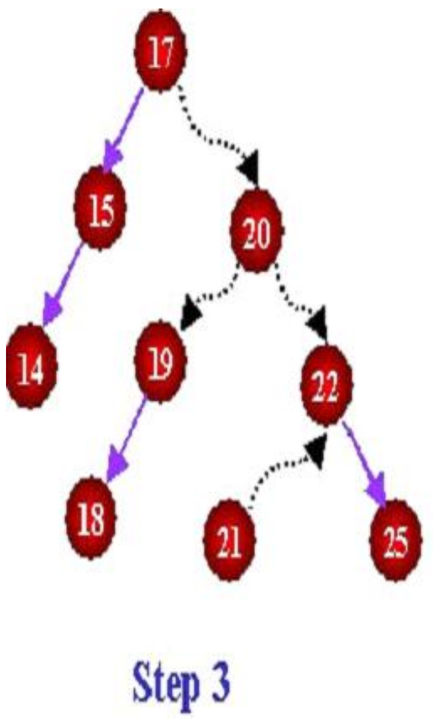
==

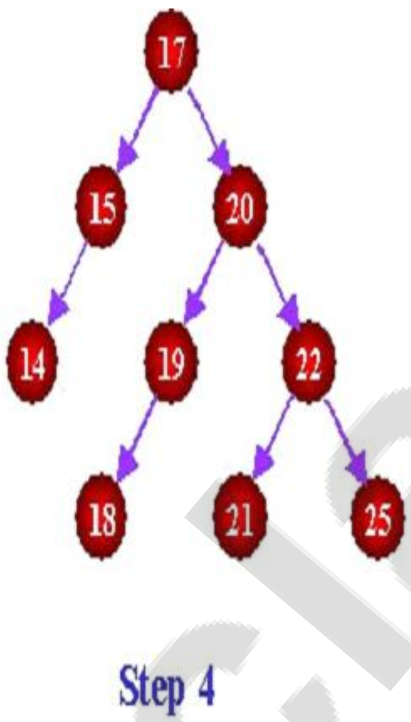
Right of Left - Double Rotation:





==





Q. Define an AVL tree. Write an algorithm to rotate AVL tree left and illustrate with the help of an example.

Algorithm : Rotate Right

Algorithm rotateright (root <tree pointer>)

Pre: root points to tree to be rotated

Post: Node rotated and root updated

```
1. temp = root->left
2. root->left = temp->right
3. temp->right = root
4. root = temp
5. return
```

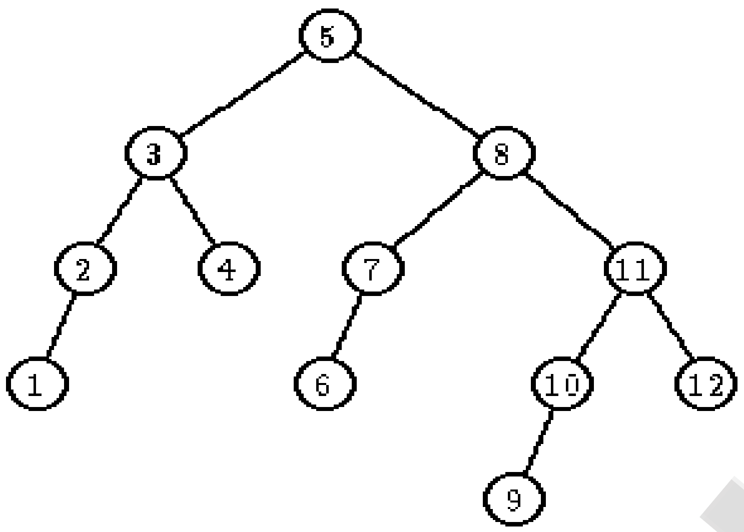
Algorithm : Rotate Right

```
Algorithm rotateleft (root <tree pointer>)
Pre:  root points to tree to be rotated
Post: Node rotated and root updated
    1. temp = root-> right
    2. root-> right = temp->left
    3. temp->left = root
    4. root = temp
    5. return
```

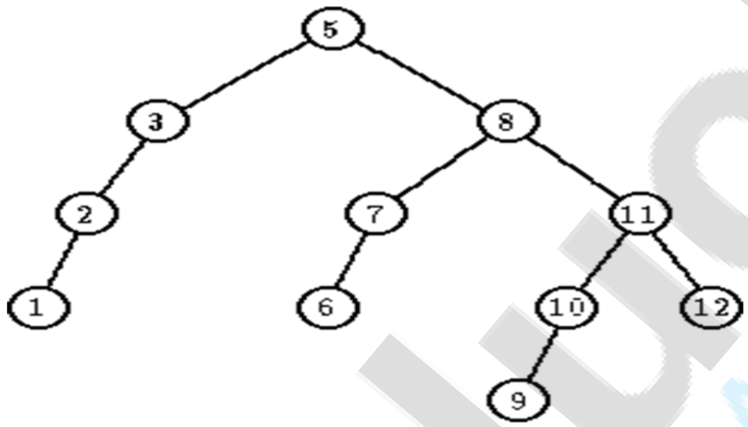
Deletion:

- Unlike the insertion algorithm, more than one rotation may be required after a delete operation, so in some cases we will have to continue back up the tree after a rotation.

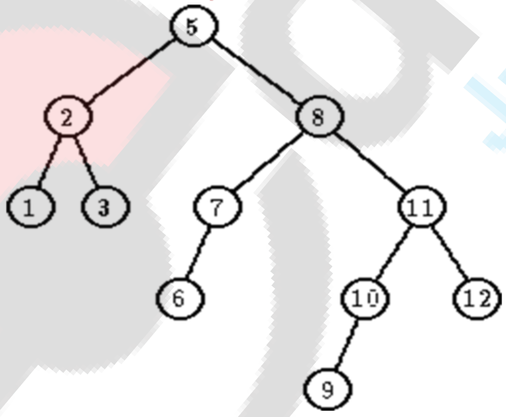
==



Delete 4

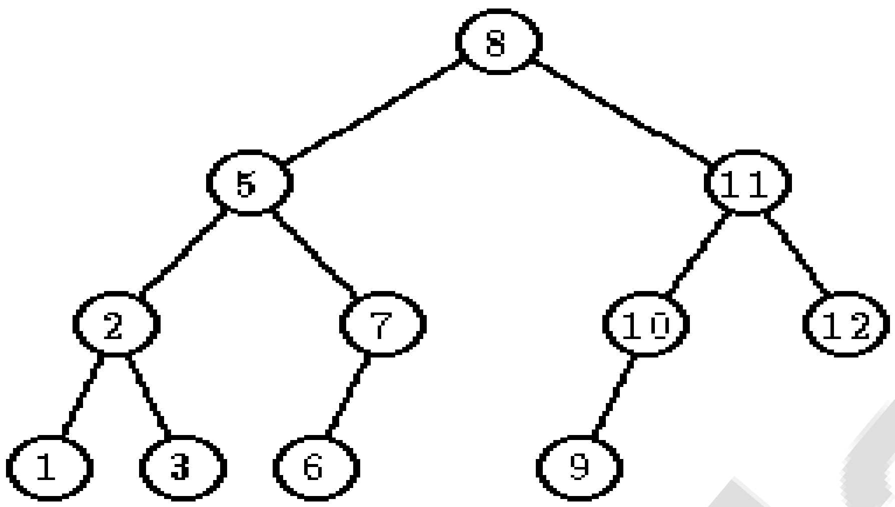


Imbalance at ‘3’ implies a LL rotation with ‘2’



Still imbalance at ‘5’. RR rotation with ‘8’.

==


8
5
11
2
7
10
12
1
3
6
9