SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

TREES

UNIT - 4

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Syllabus

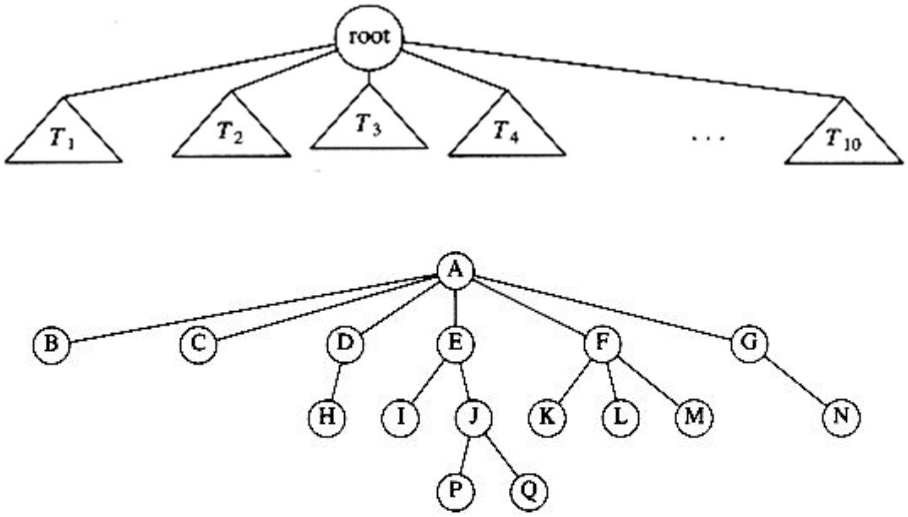
- General Trees , Tree Terminologies
- Tree Representation , Tree Traversal
- Binary Tree Representation , Expression Trees
- Binary Tree Traversal , Threaded Binary Tree
- Binary Search Tree :Construction, Searching Insertion and Deletion
- AVLTrees: Rotations , Insertions
- B-Trees Constructions , B-Trees Search
- B-Trees Deletions, Splay Trees
- Red Black Trees , Red Black Trees Insertion

General Trees

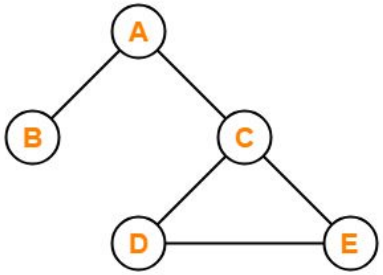
- Linear access time for linked lists too high.
- Solution – group data into trees.
- Trees – non linear data structure
- Used to represent data contains a hierarchical relationship among elements example: family, record
- Worst Case time – O(log n)
- Tree can be defined in many ways, eg: recursively

General Trees

- Tree consists of collection of nodes arranged in hierarchical pattern



Tree - Examples



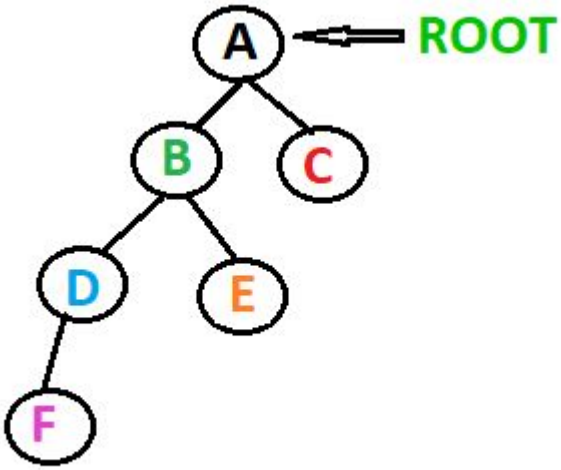
Not a tree

Tree Terminology

- Root
- Edge
- Parent
- Child
- Sibling
- Degree
- Internal node
- Leaf node
- Level
- Height
- Depth
- Subtree

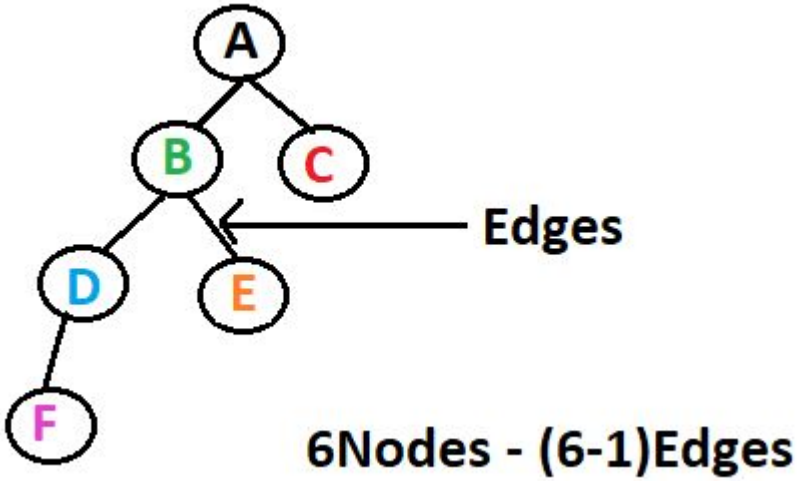
Tree Terminology - Root

- The stating node of a tree is root node
- Only one root node



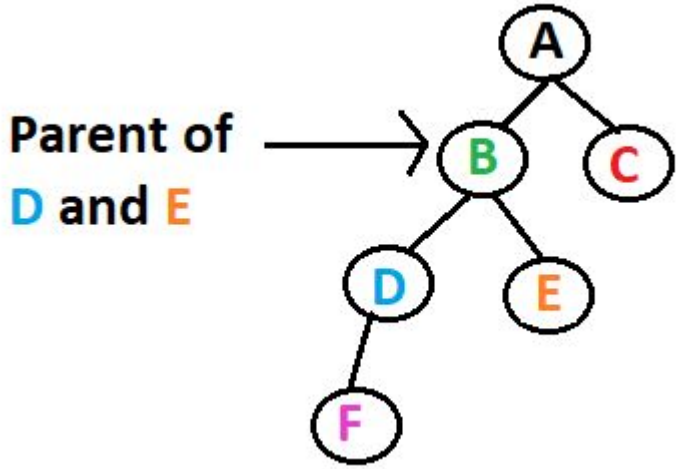
Tree Terminology - Edge

- Nodes are connected using the link called edges
- Tree with n nodes exactly have (n-1) edges



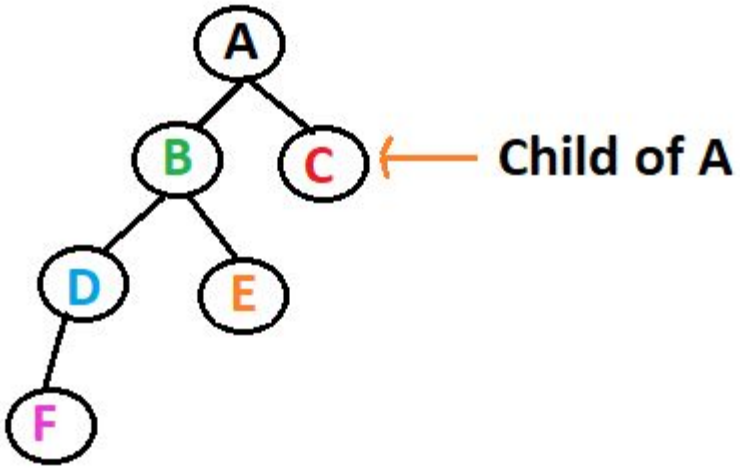
Tree Terminology - Parent

- Node that have children nodes or have branches connecting to other nodes
- Parental node have one or more children nodes



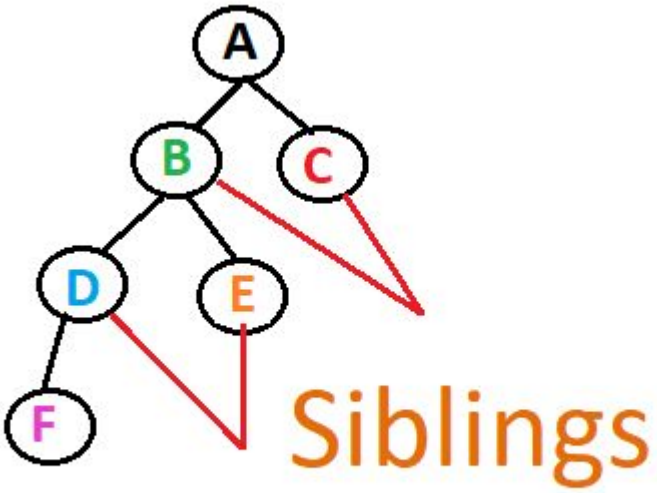
Tree Terminology - Child

- Node that is descendant of any node is child
- All nodes except root node are child node



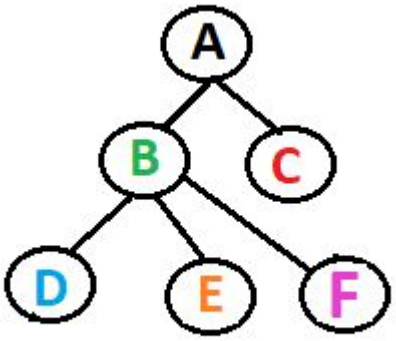
Tree Terminology - Siblings

- Nodes with same parents are siblings



Tree Terminology - Degree

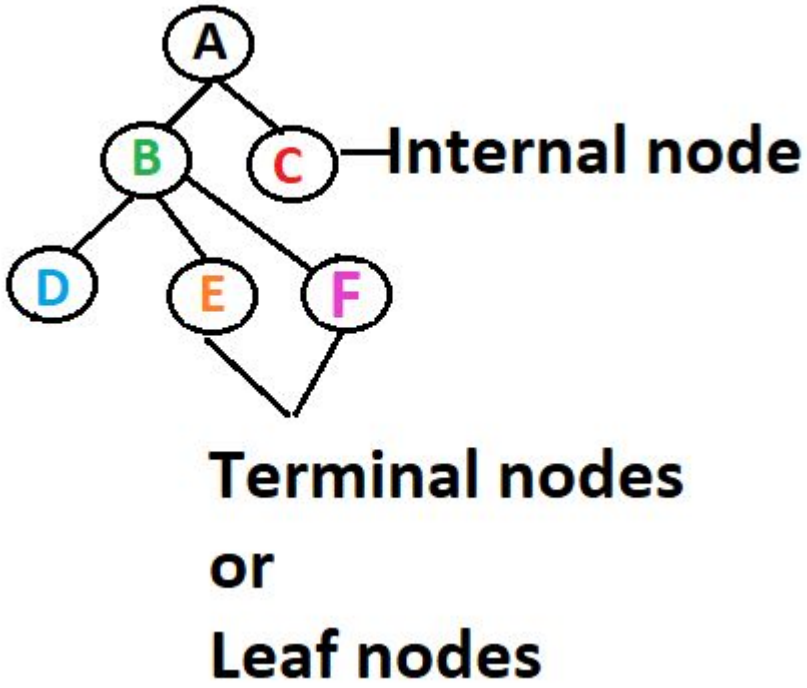
- Degree of node – number of children per node
- Degree of tree – highest degree of a node among all nodes in tree



Degree A – 2
Degree B – 3
Degree C – 0
Degree D – 0
Degree E – 0
Degree F – 0
Degree of entire tree - 3

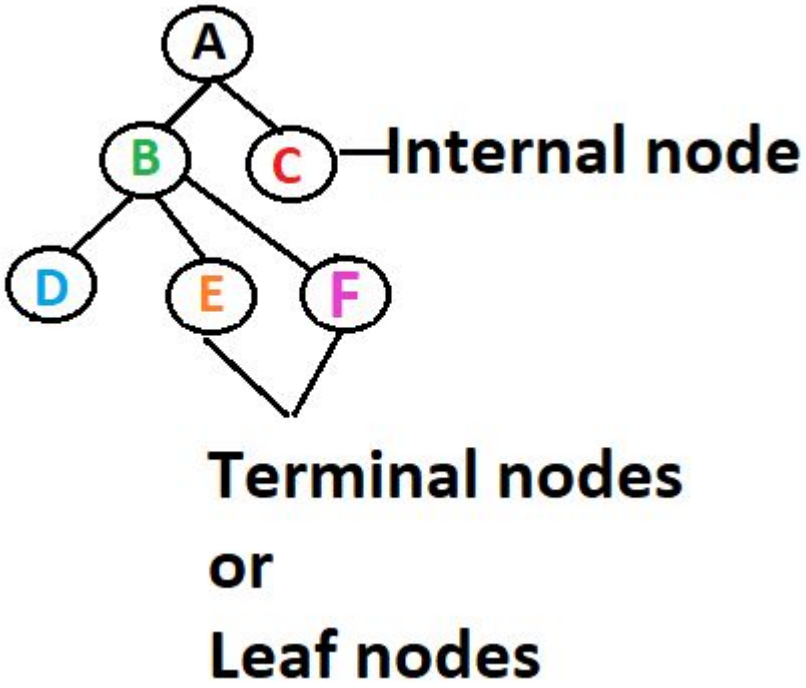
Tree Terminology – Internal Node

- Node with minimum one child – internal node
- Also known as **non – terminal nodes**



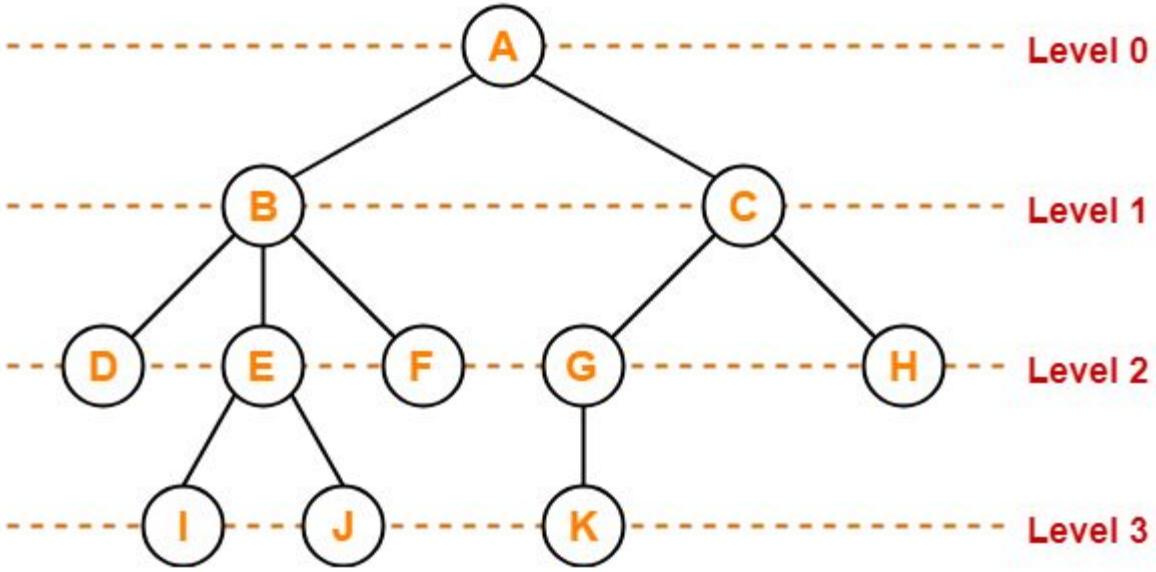
Tree Terminology – Leaf Node

- Node with no child – Leaf node
- Also known as **External nodes or Terminal nodes**



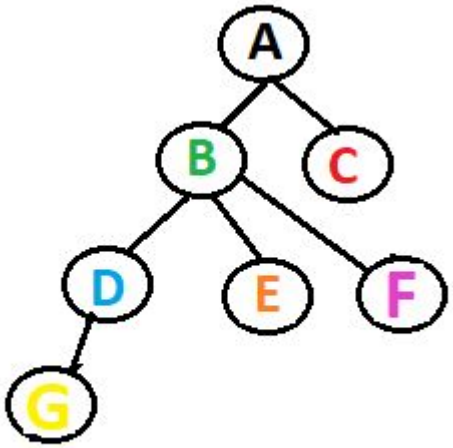
Tree Terminology – Level

- Each step of tree is level number
- Staring with root as 0



Tree Terminology – Height

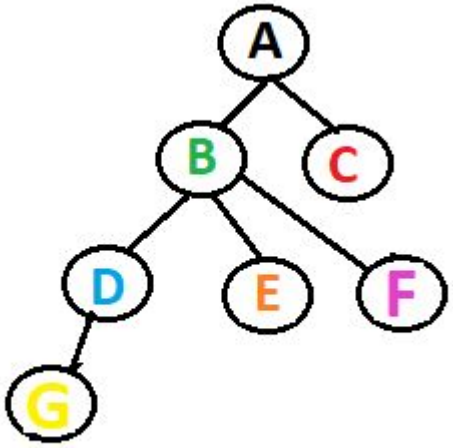
- Number of edges in longest path from the node to any leaf
- Height of any leaf node is 0
- Height of Tree = Height of Root node



Height A – 3
Height B – 2
Height D – 1
Height C,G,E,F – 0
Height of tree - 3

Tree Terminology – Depth

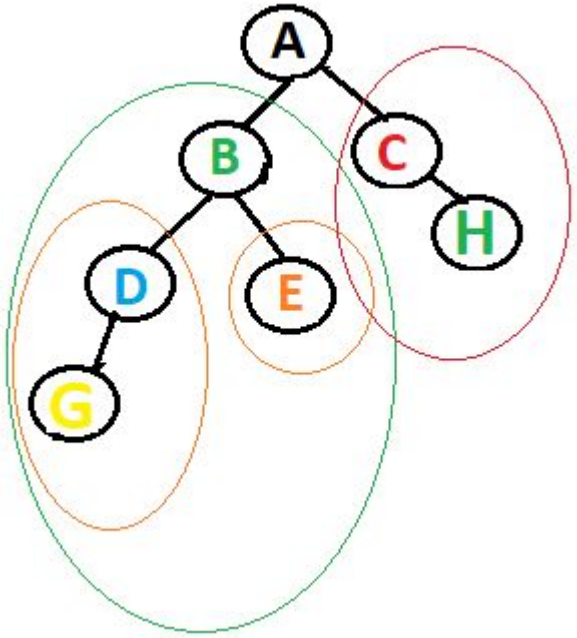
- Number of edges from root node to particular node is Depth
- Depth of root node is 0
- Depth of Tree = Depth of longest path from root to leaf



Depth A – 0
Depth B ,C – 1
Depth D, E, F – 2
Depth G – 3
Depth of tree - 3

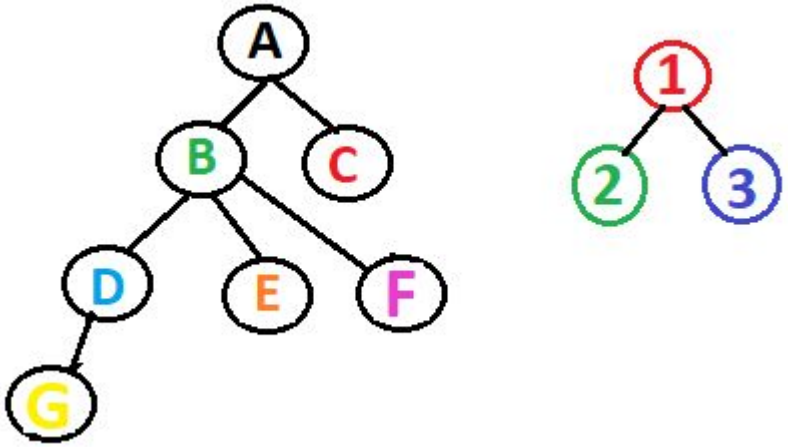
Tree Terminology – Subtree

- Each child of a tree forms a subtree recursively



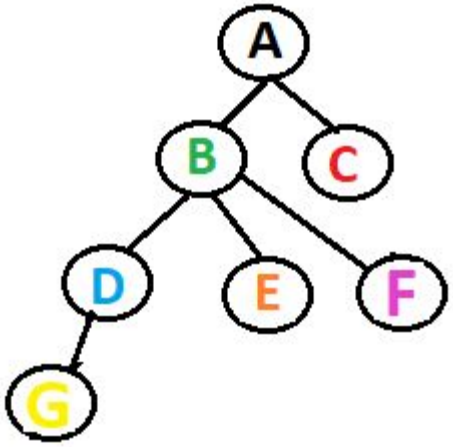
Tree Terminology – Forest

- Set of disjoint trees



Tree Terminology - Path

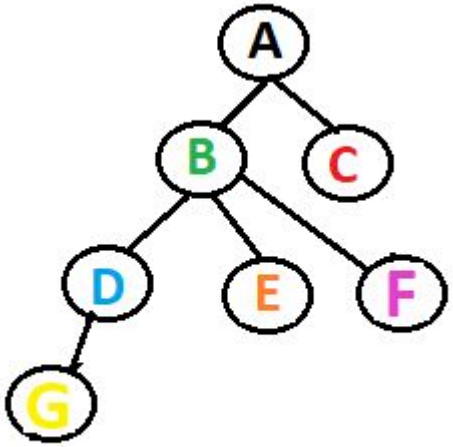
- A path from node a^1 to a^k is defined as a sequence of nodes $a^1, a^2, \ldots, a^k$ such that a^i is the parent of a^{i+1} for $1 \leq i < k$



Path A-G: A,B,D,G

Tree Terminology – length of Path

- Number of edges in the path

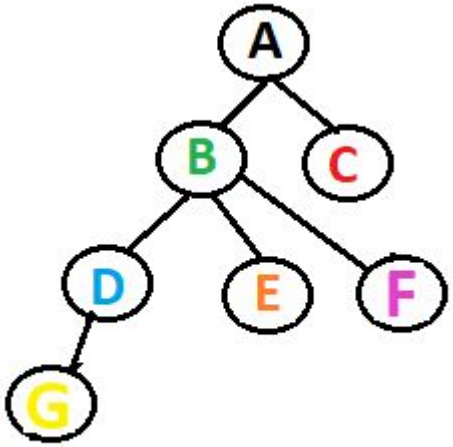


Path A-G: A – B – D – G

Length of path A-G : 3

Tree Terminology – Ancestor & Descendant

- If there is a path from A to B, then A is an ancestor of B and B is a descendant of A



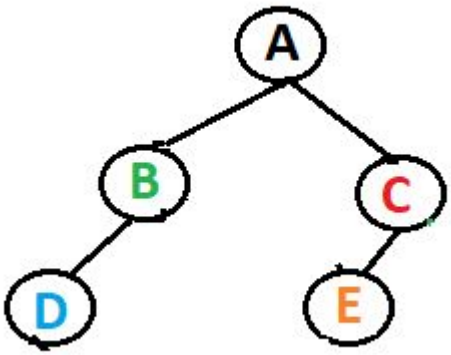
Path A-G: A – B – D – G

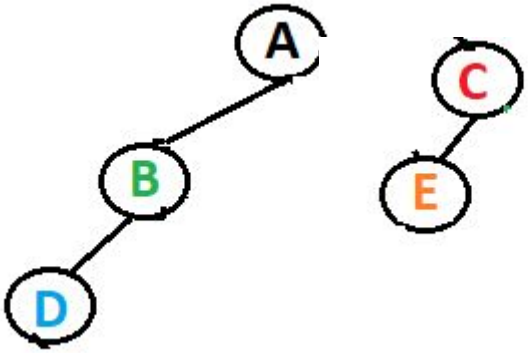
A – Ancestor for G, B, D…

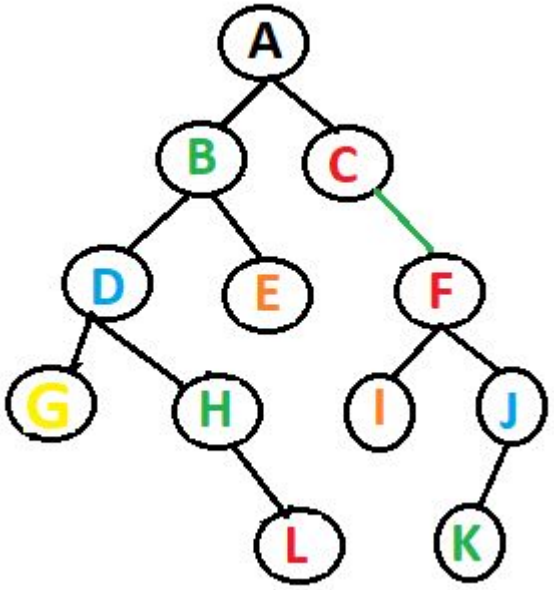
G – Descendant of D,B,A

Binary Tree

- A binary tree is a data structure specified as a set of so-called node elements. The topmost element in a binary tree is called the root node, and each node has 0, 1, or at most 2 kids.

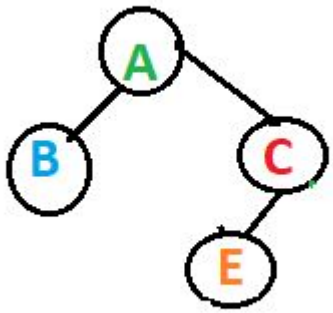


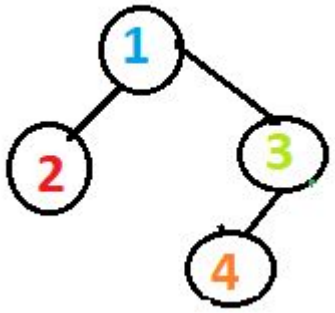




Similar binary tree

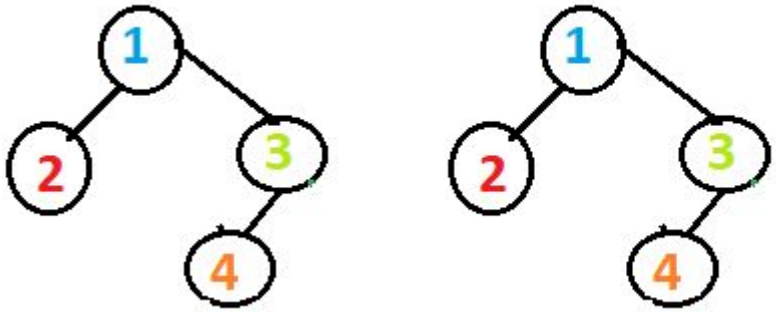
- Tree with same structure





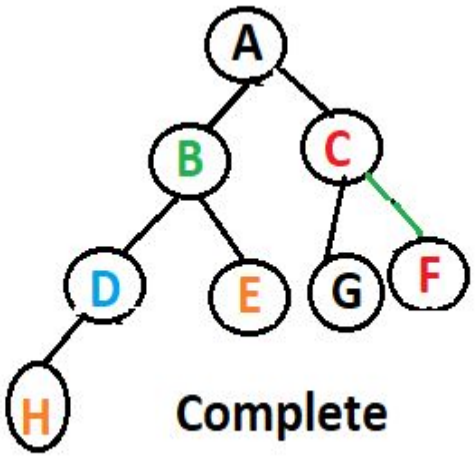
Copies of Binary Tree

- Same structure and same data node

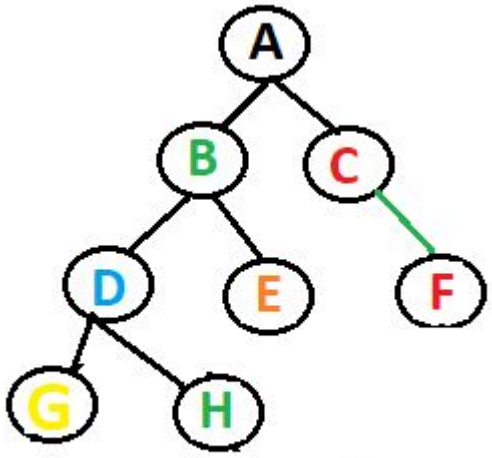


Complete Binary Tree

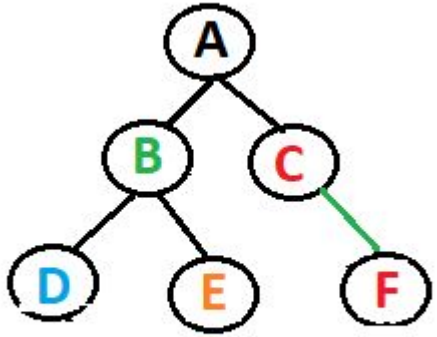
- Except last level, all the nodes need to completely filled
- All nodes filled from left to right



Complete



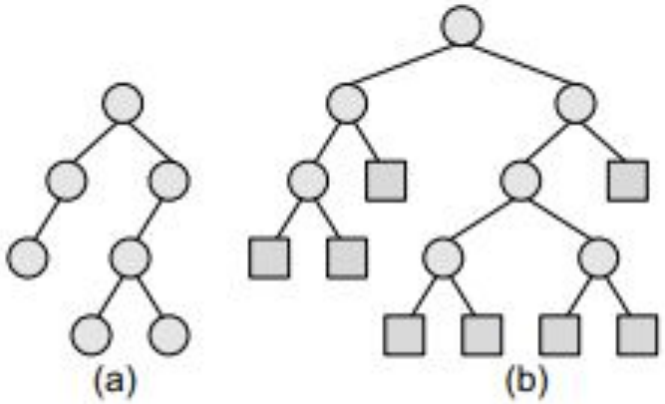
Not - Complete



Not- Complete

Extended Binary Tree

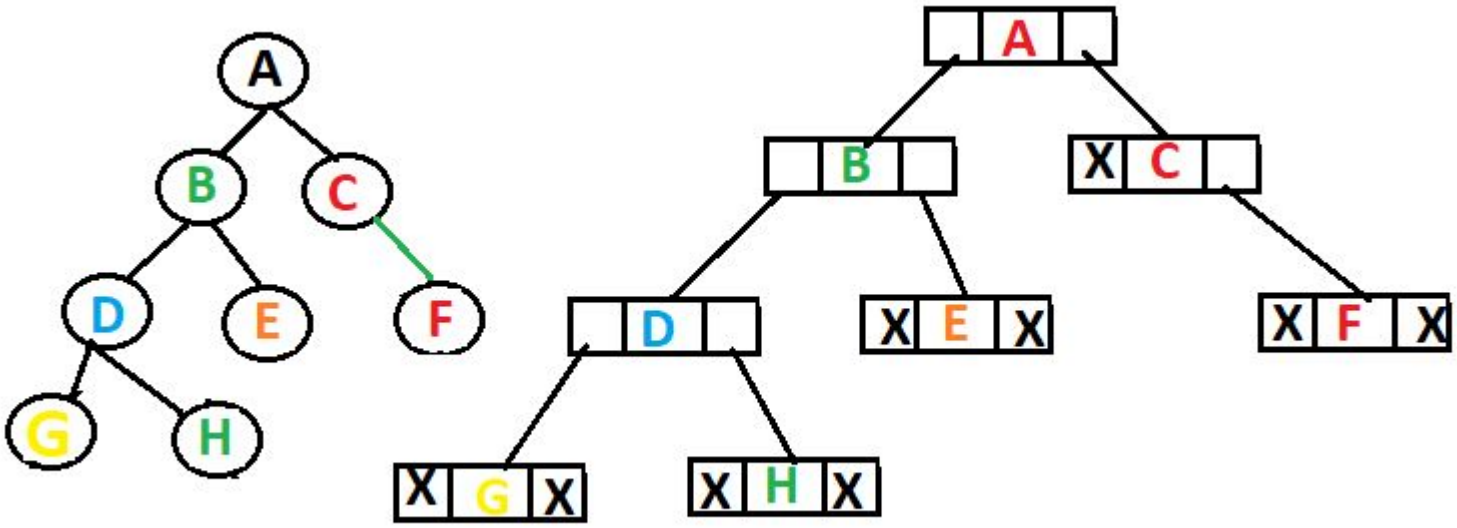
- Each node will have either two child or zero child
- Nodes with two children – Internal nodes
- Node with no child – External nodes



(a) Binary tree and (b) extended binary tree

Representation of Binary Tree

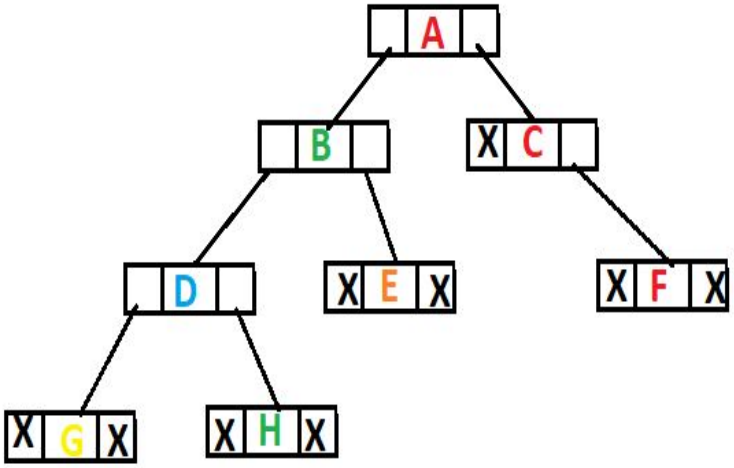
- Each node – three portion – Data portion, left child pointer, Right child pointer



Linked List Representation of Tree

```
Struct NODE
{
Struct NODE *leftchild;
Int value;
Struct NODE *rightchild;
};
```

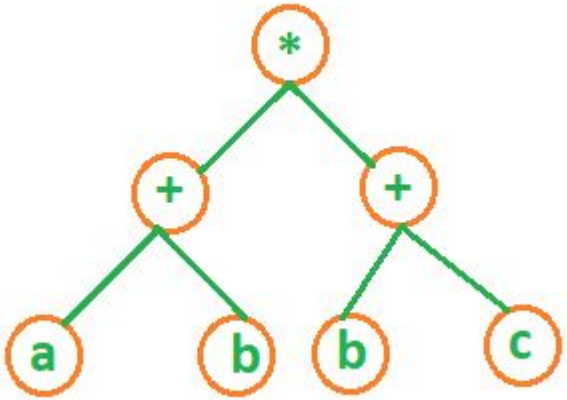
Linked List Representation of Tree



	Index	Left child	Node	Right child
	1	-1	G	-1
Root - 2	2	5	A	8
	3			
	4	-1	F	-1
	5	10	B	12
	6			
	7			
	8	-1	C	4
	9			
	10	1	D	14
	11			
	12	-1	E	-1
	13			
	14	-1	H	-1

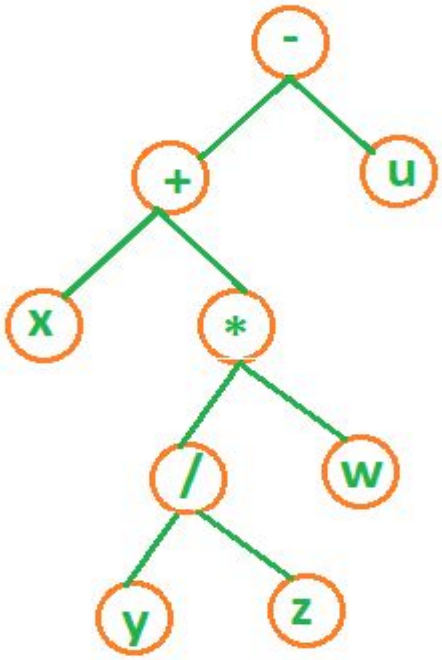
Expression Tree

- Used to store algebraic expression
- Example 1: exp = (a+b)*(b+c)



Expression Tree

- Used to store algebraic expression
- Example 1: exp = x+y/z*w-u
- Can be written as: exp = ((x+ ((y/z)*w)-u)

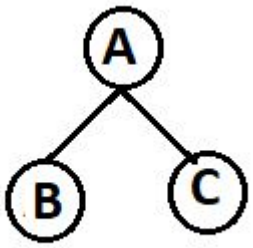


Binary Tree Traversal

- Traversal – visiting all node only once
- Based on the order of visiting :
  - In – order traversal
  - Pre – order traversal
  - Post – order traversal

In-order Traversal

- **Traverse left node , visit root node, Traverse right node**



B – A – C

Traverse Left node– B
No left child for B subtree
Visit root node B
No right child for B subtree
Visit root node A
Traverse Right node– C
No left child for C subtree
Visit root node C
No right child for C subtree

In-order Traversal

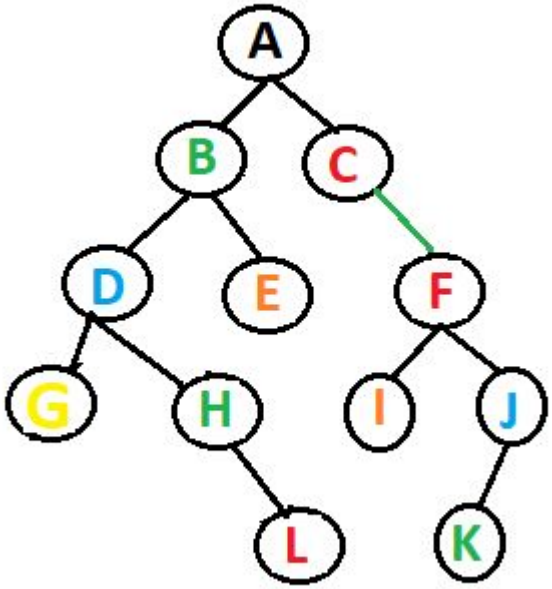
Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

In-order Traversal

Algorithm: Inorder(Tree)

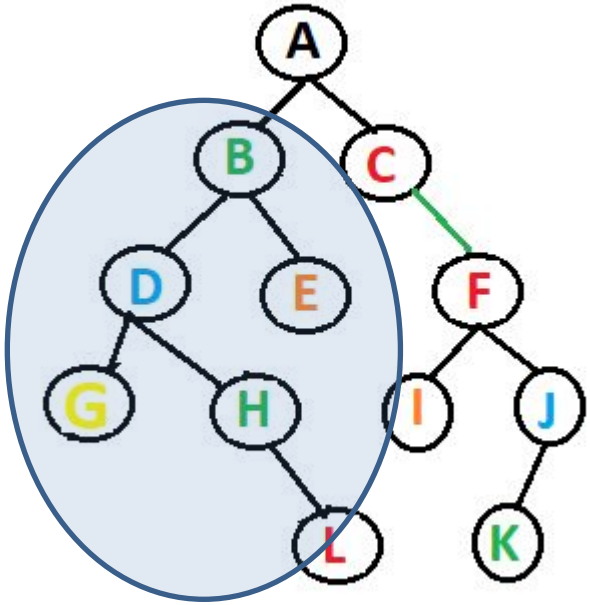
1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End



In-order Traversal

Algorithm: Inorder(Tree)

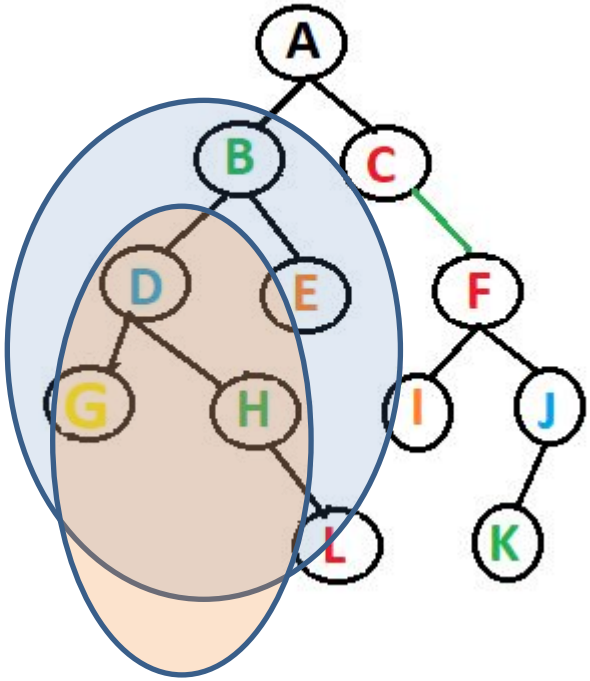
1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End



In-order Traversal

Algorithm: Inorder(Tree)

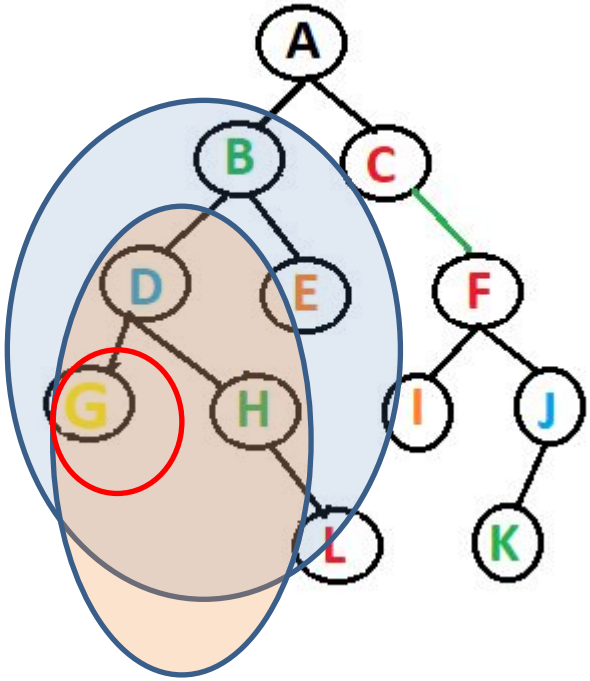
1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End



In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

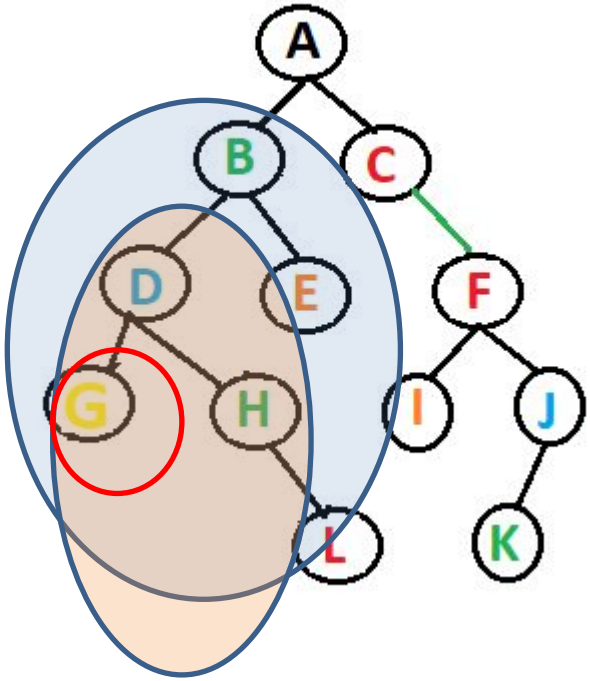


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G,

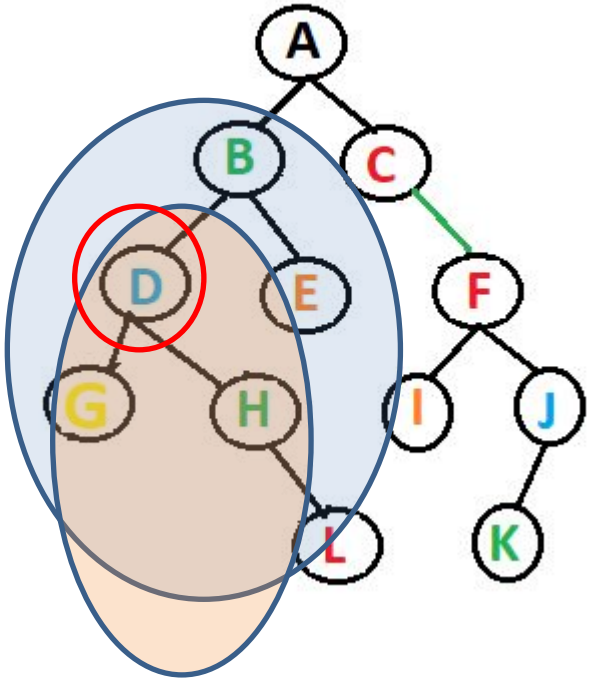


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D,

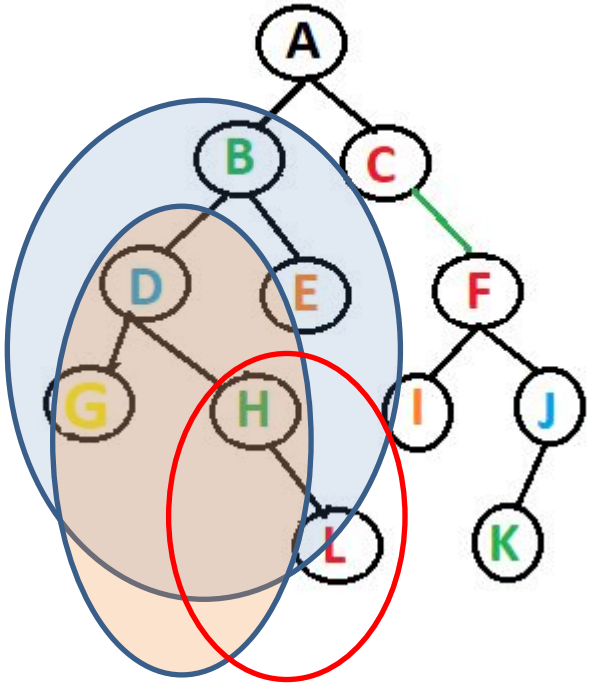


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D,

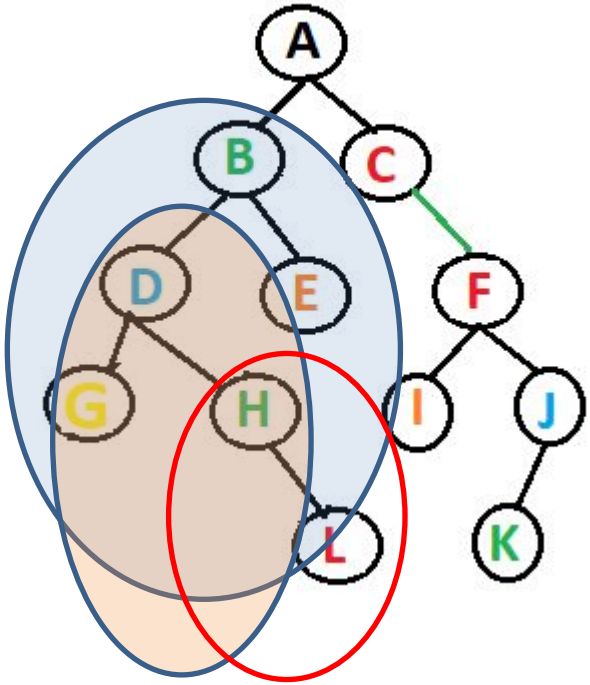


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H,

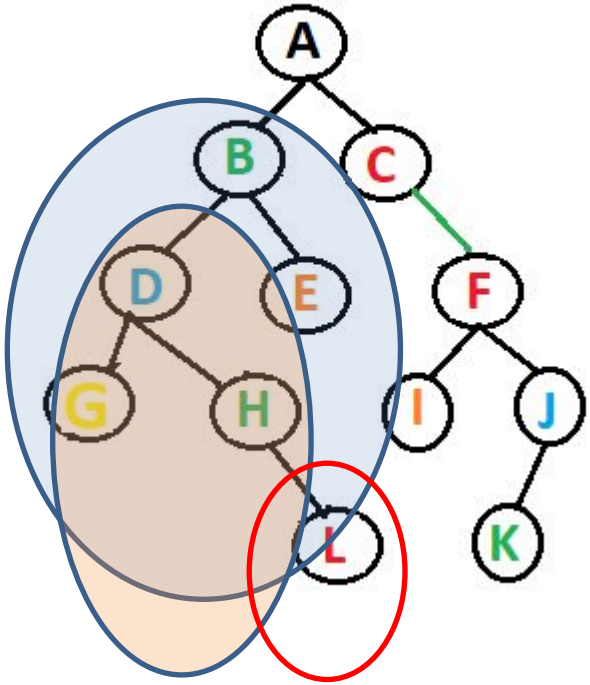


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L,

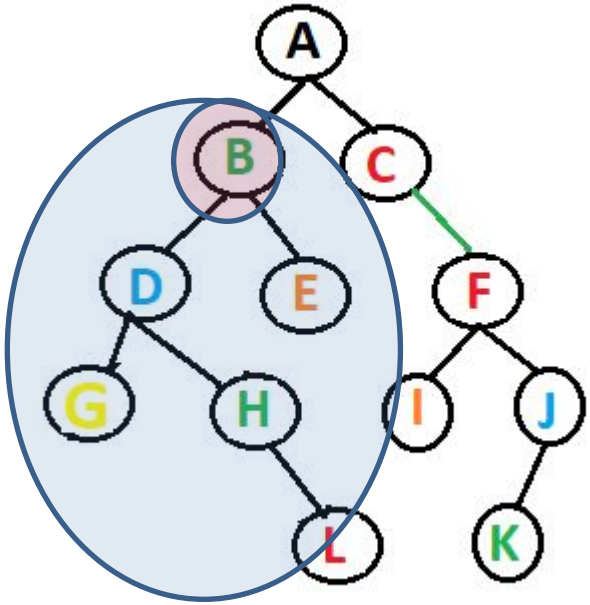


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L, B

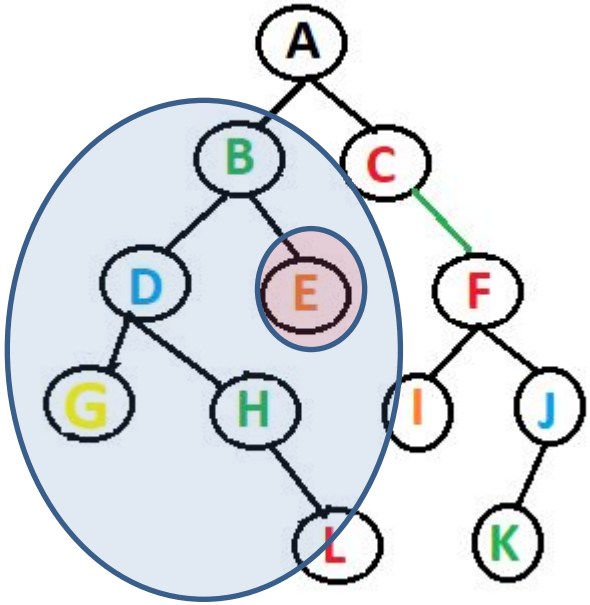


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L, B, E

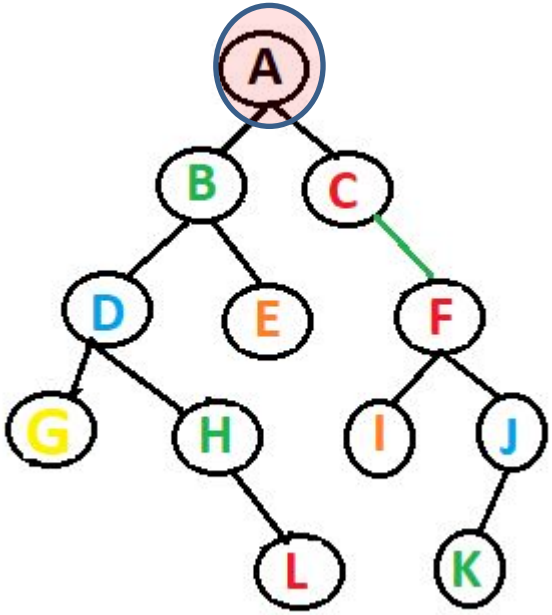


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L, B, E, A,

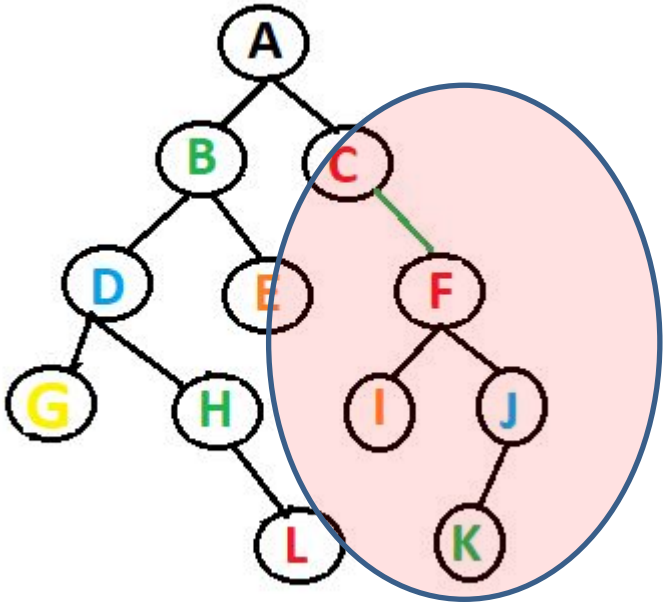


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L, B, E, A,

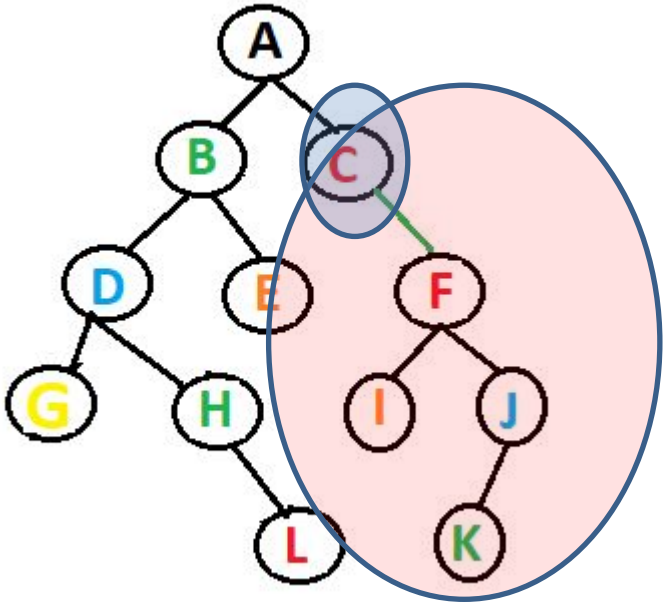


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L, B, E, A, C

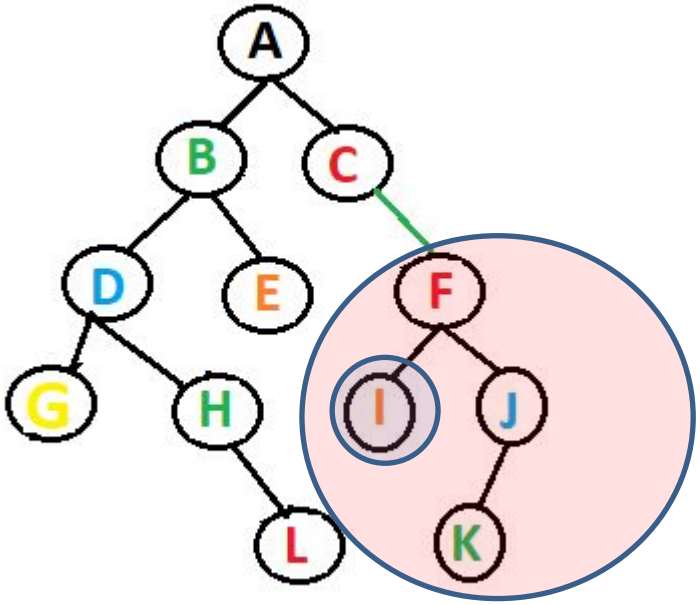


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L, B, E, A, C, I,

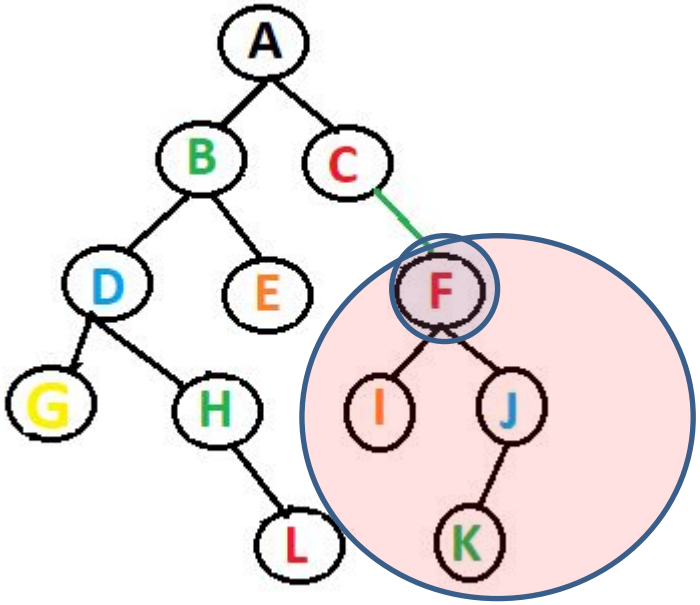


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L, B, E, A, C, I,
F,

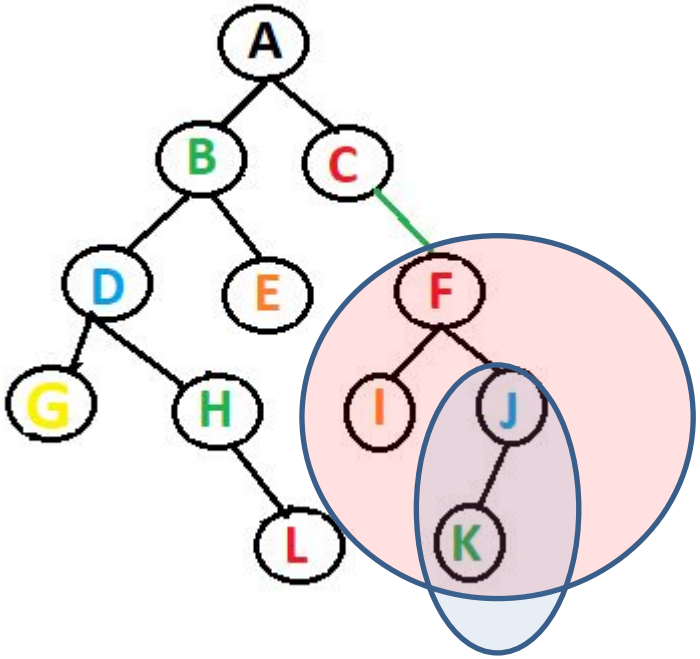


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L, B, E, A, C, I,
F,

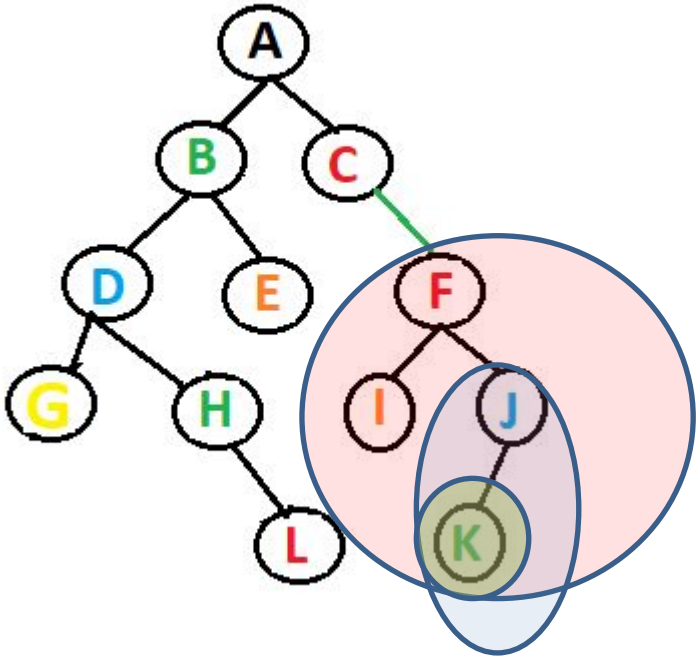


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

Result : G, D, H, L, B, E, A, C, I,
F, K,

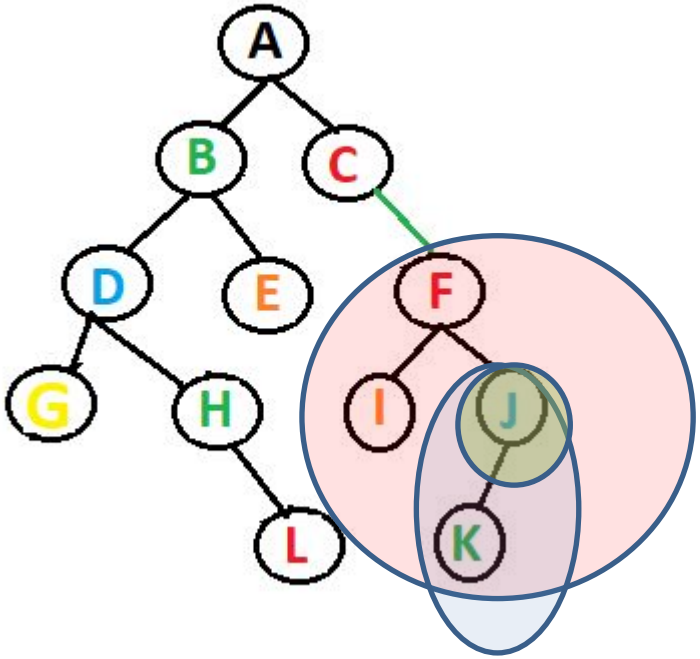


In-order Traversal

Algorithm: Inorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

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

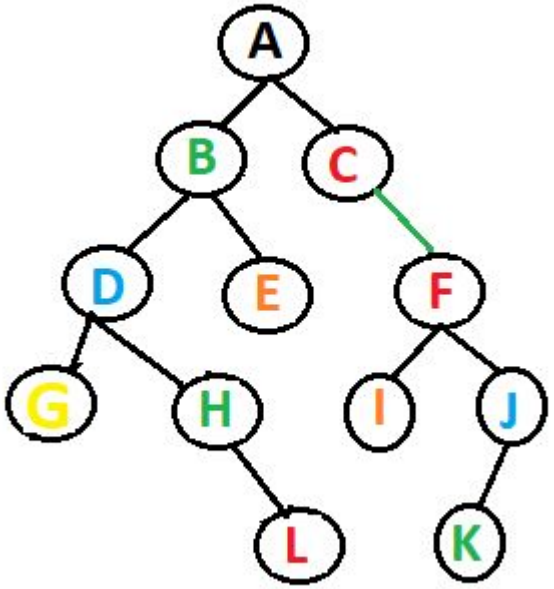


In-order Traversal

Algorithm: Inorder(Tree)

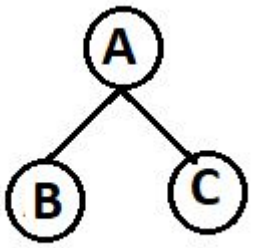
1. Repeat step 2 – 4 while Tree != Null
2. Inorder(Tree->left)
3. Write(Tree->Data)
4. Inorder(Tree->right)
5. End

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



Pre – order Traversal

- **Visit root node, Traverse left node , Traverse right node**



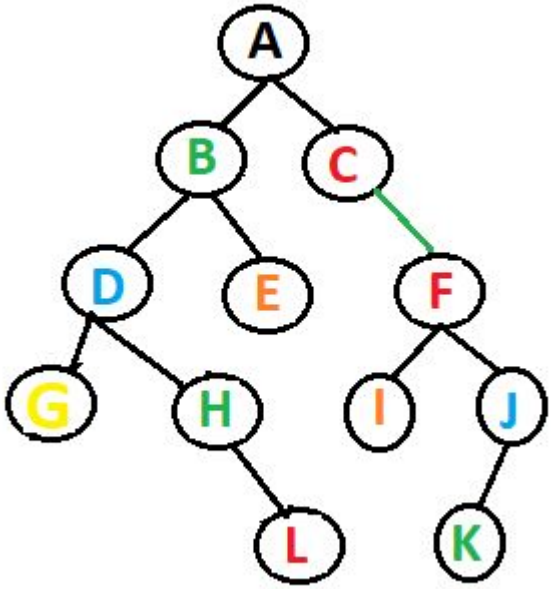
A – B – C

Visit root node A
Traverse Left node– B
Visit root node B
No left child for B subtree
No right child for B subtree
Traverse Right node– C
Visit root node C
No left child for C subtree
No right child for C subtree

Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

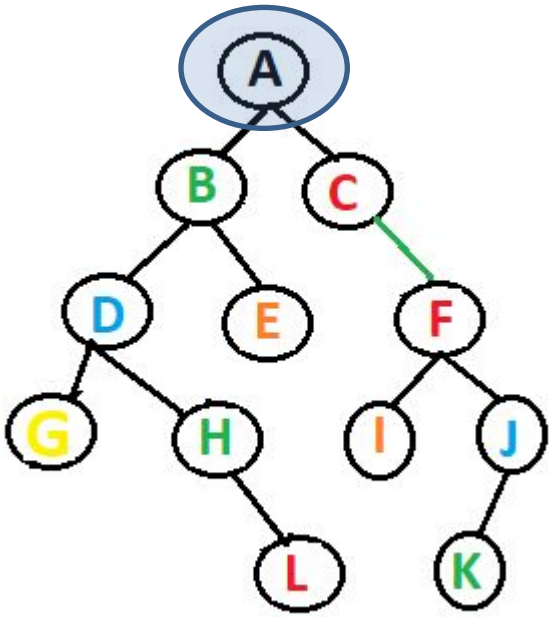


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A,

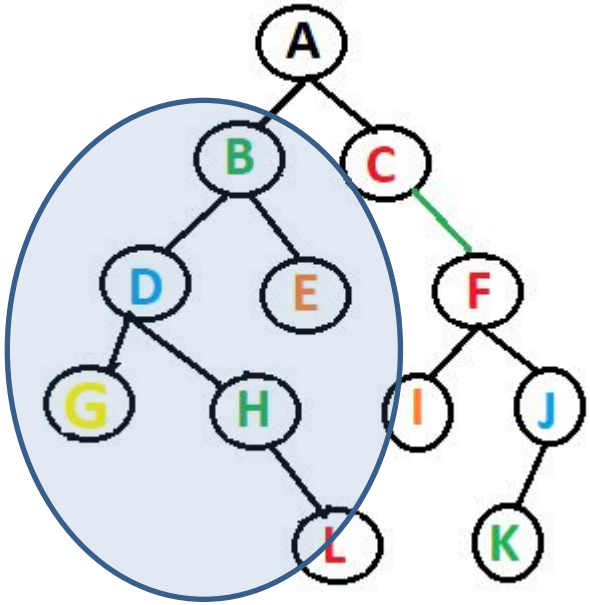


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A,

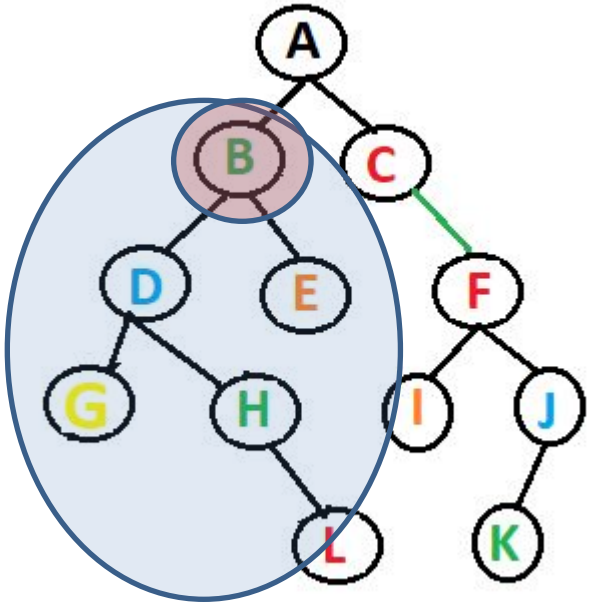


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B,

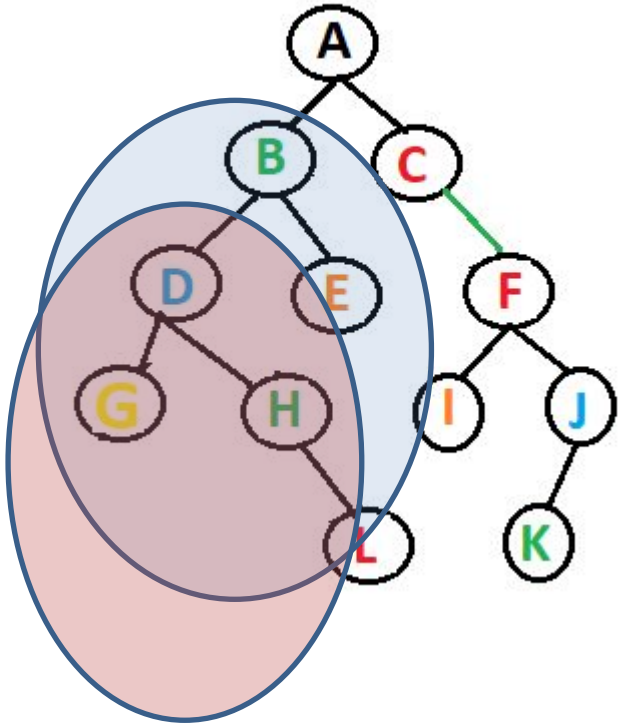


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B,

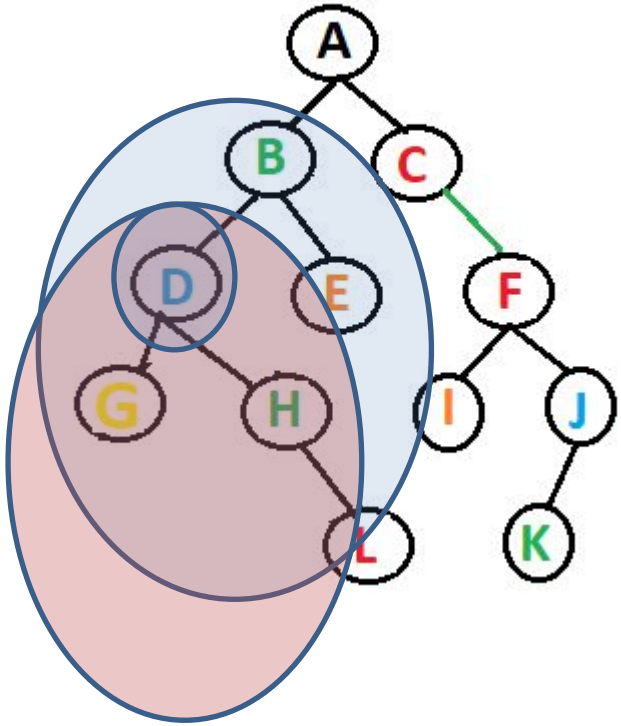


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D,

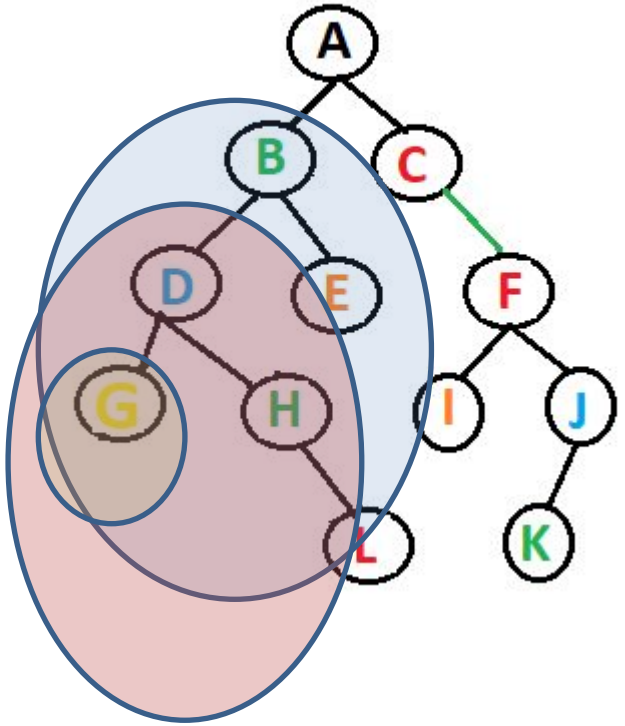


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D,G,

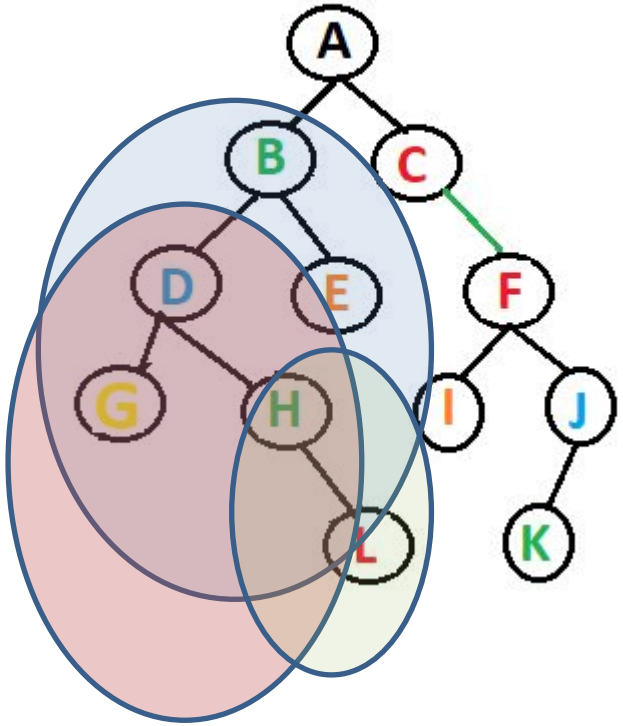


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D, G,

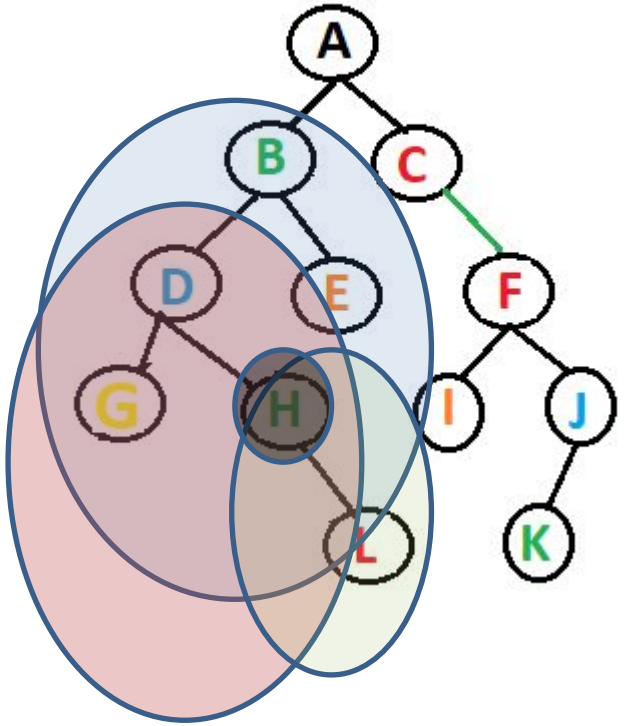


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D, G, H,

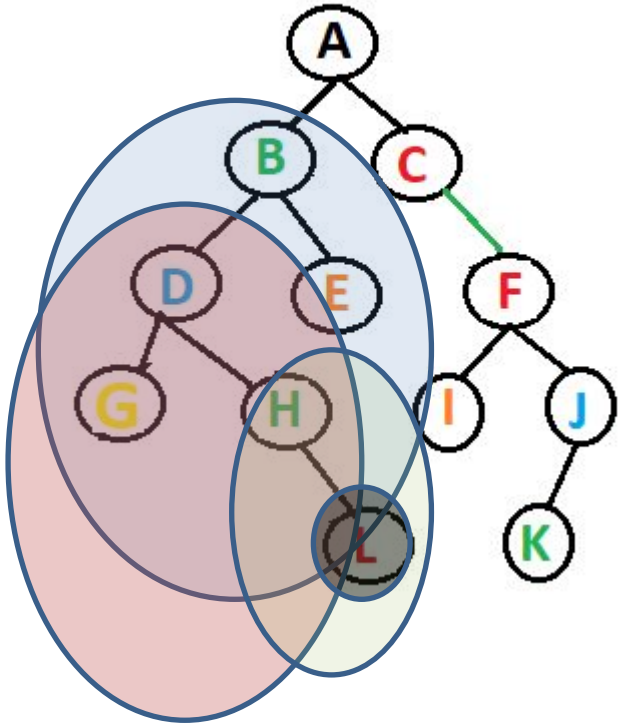


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D, G, H, L,

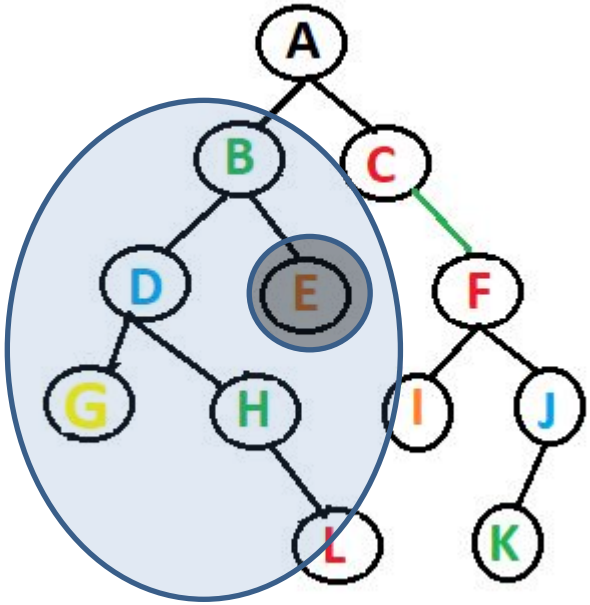


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D, G, H, L, E,

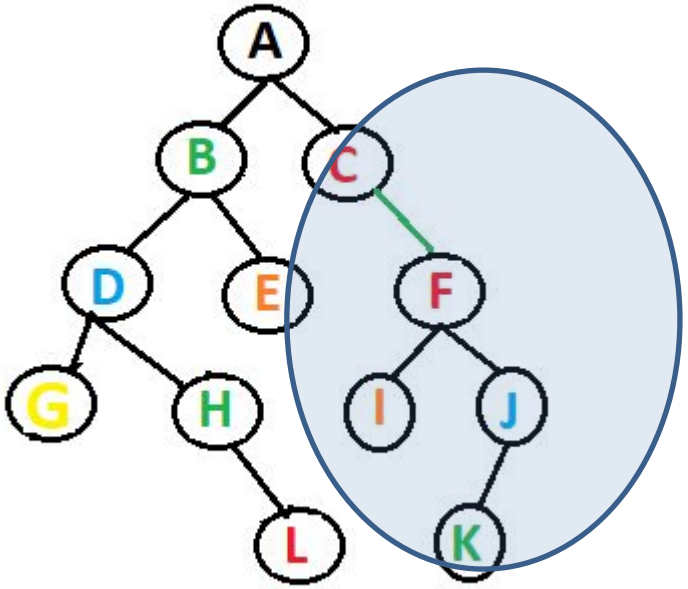


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D, G, H, L, E,

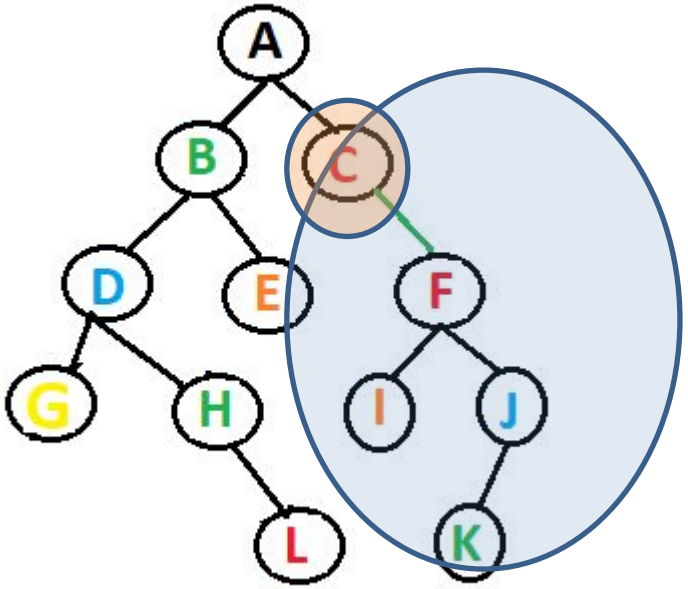


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D, G, H, L, E, C,

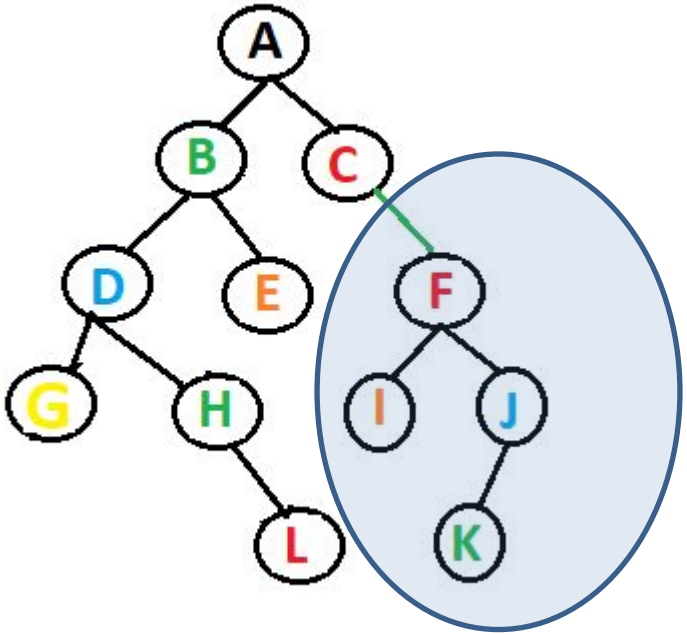


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D, G, H, L, E, C,

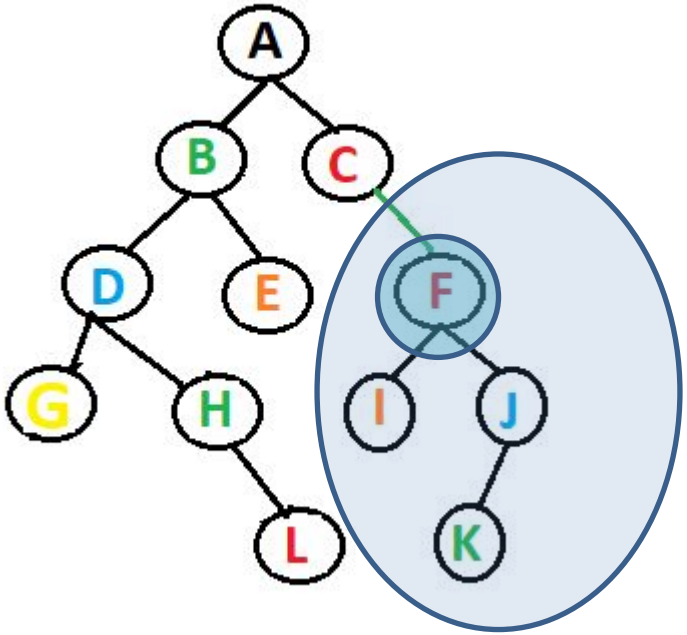


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D, G, H, L, E, C, F,

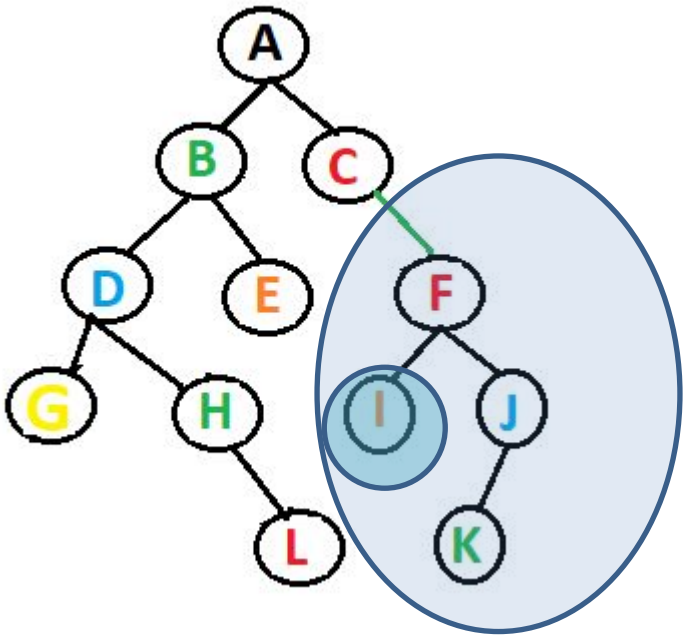


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D, G, H, L, E, C, F, I,

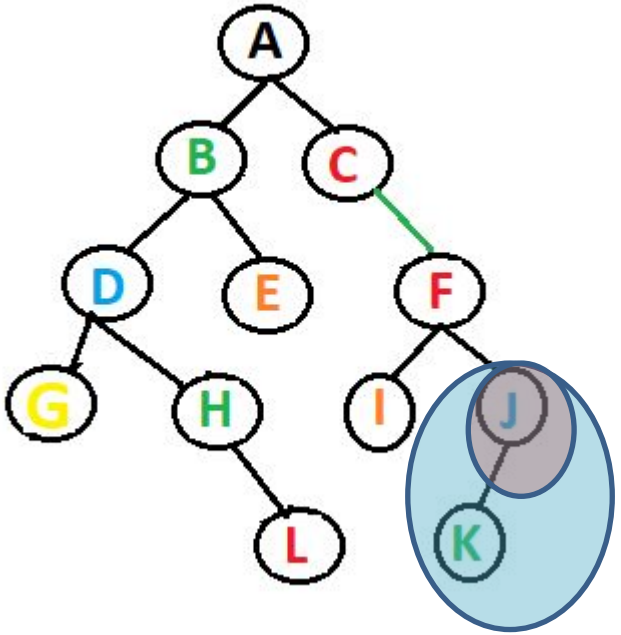


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

Result: A, B, D, G, H, L, E, C, F, I, J,

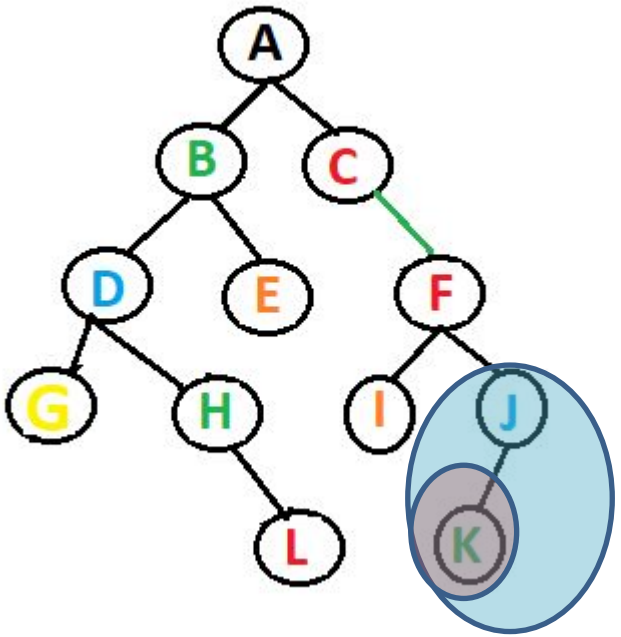


Pre-order Traversal

Algorithm: Preorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End

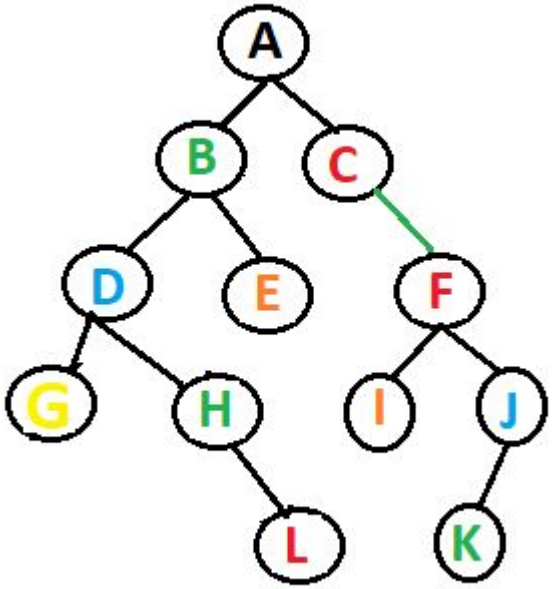
Result: A, B, D, G, H, L, E, C, F, I, J, K



Pre-order Traversal

Algorithm: Preorder(Tree)

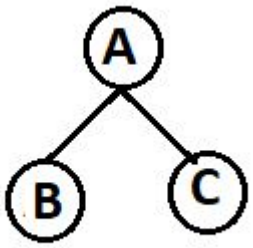
1. Repeat step 2 – 4 while Tree != Null
2. Write(Tree->Data)
3. Preorder(Tree->left)
4. Preorder(Tree->right)
5. End



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

Post– order Traversal

- **Traverse left node, Traverse right node , visit root node**



B – C – A

Traverse Left node – B
No left child for B subtree
No right child for B subtree
Visit root node B
Traverse Right node– C
No left child for C subtree
No right child for C subtree
Visit root node C
Visit root node A

Post-order Traversal

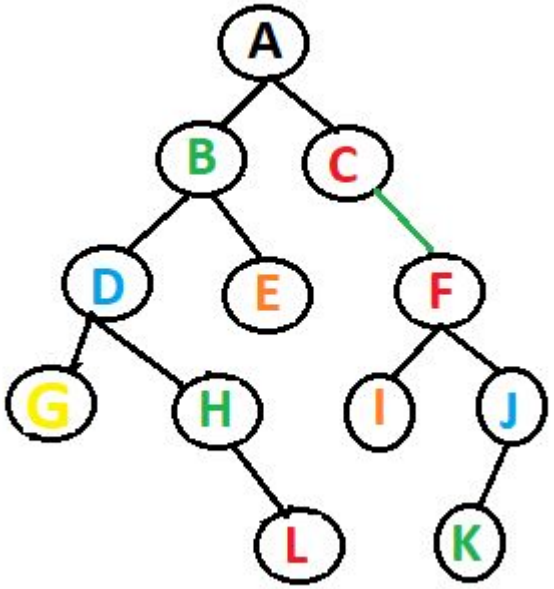
Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Post-order Traversal

Algorithm: Postorder(Tree)

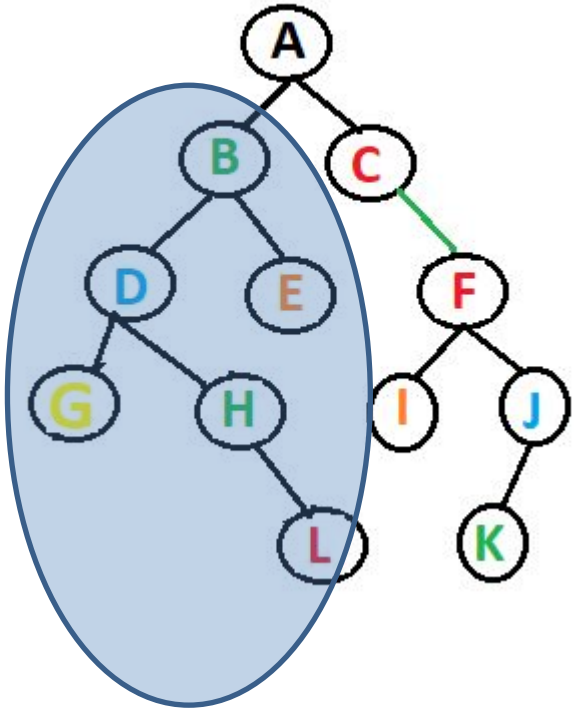
1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End



Post-order Traversal

Algorithm: Postorder(Tree)

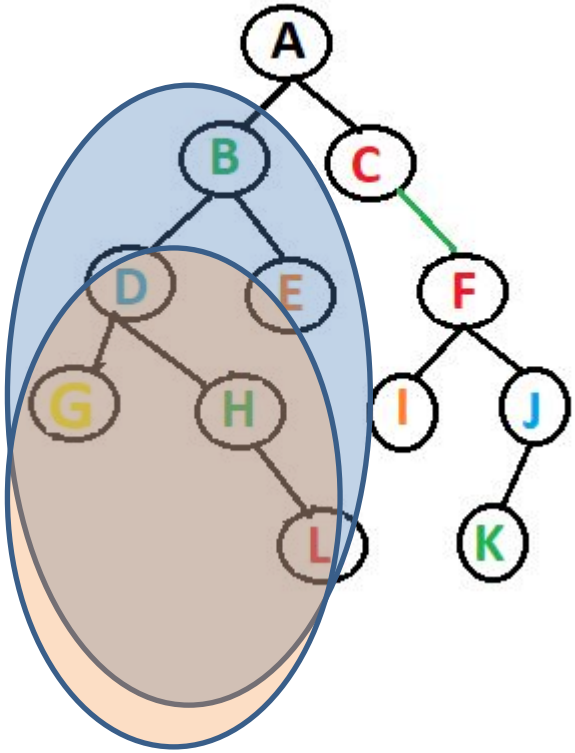
1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End



Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

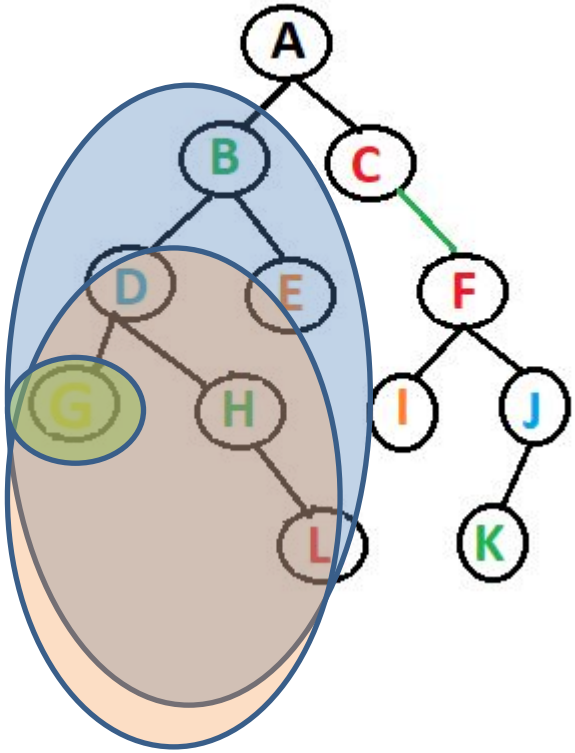


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G,

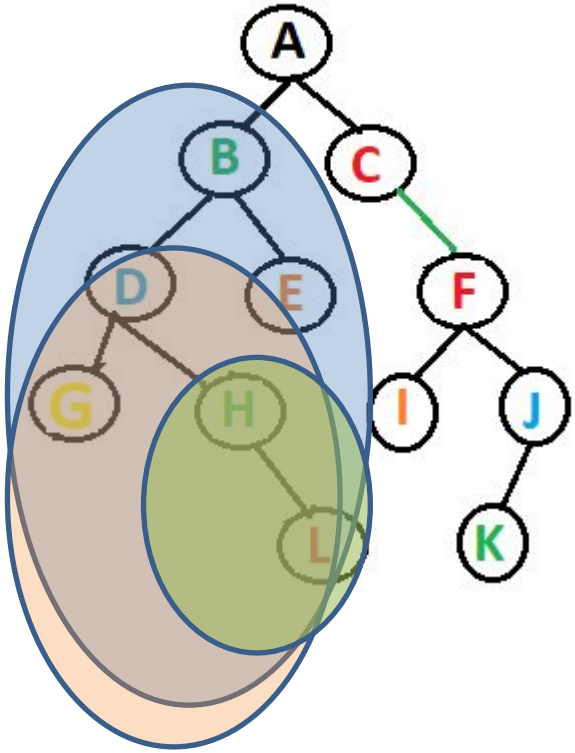


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G,

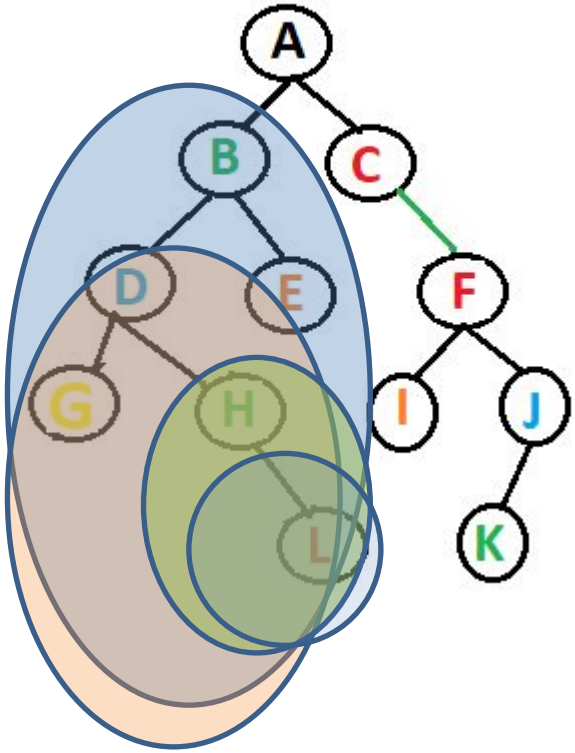


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L,

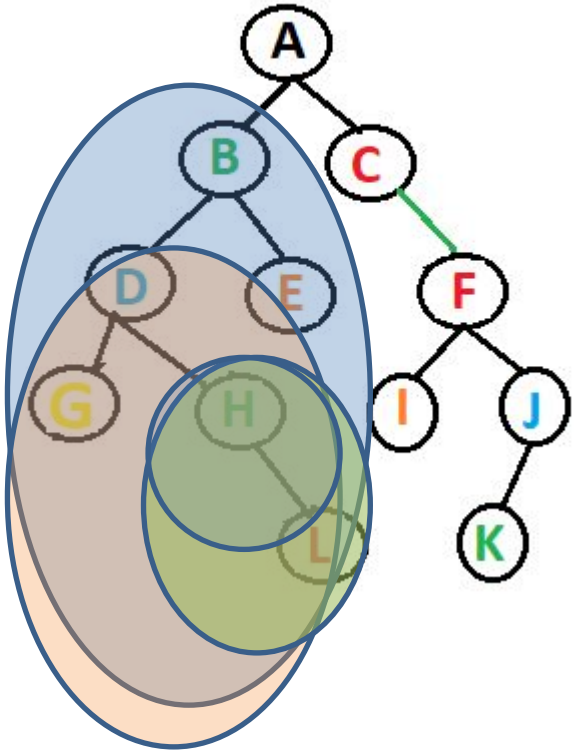


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H,

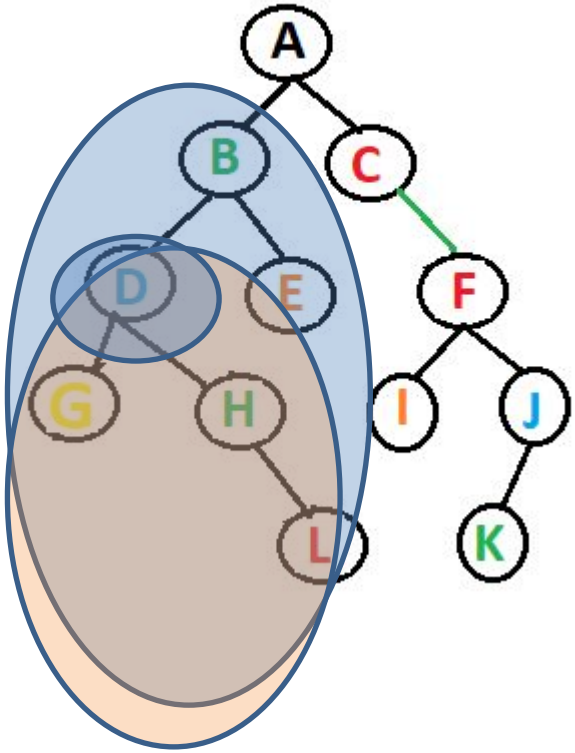


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H, D

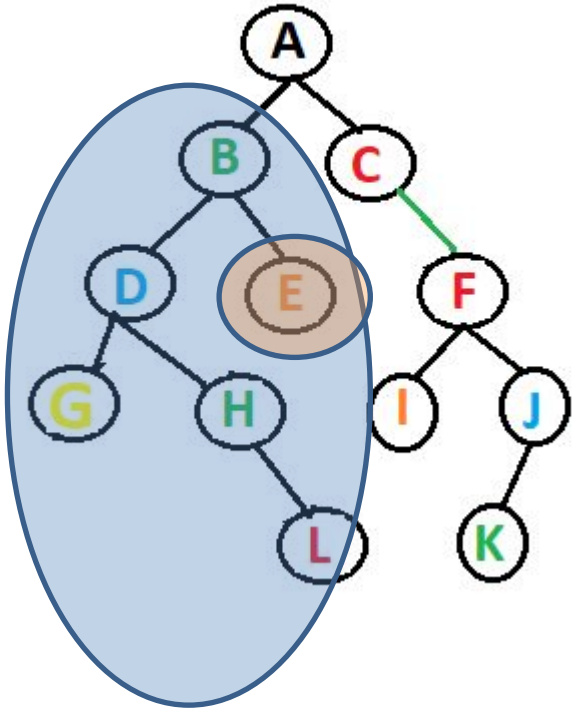


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H, D, E,

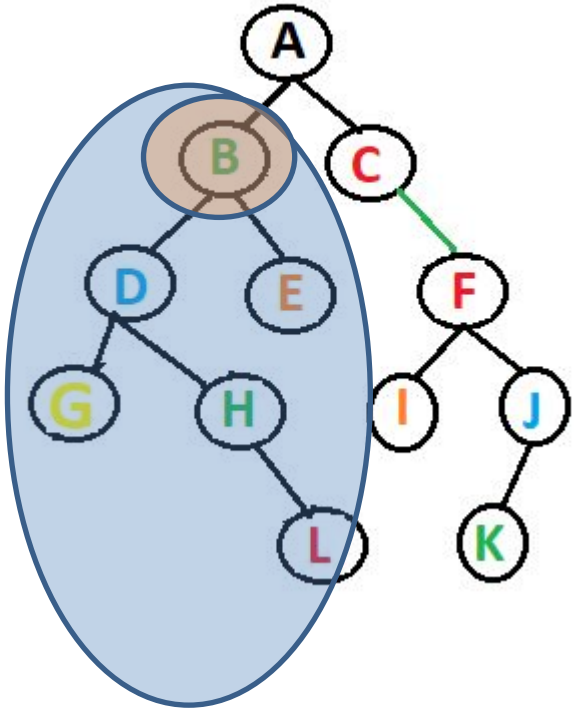


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H, D, E, B,

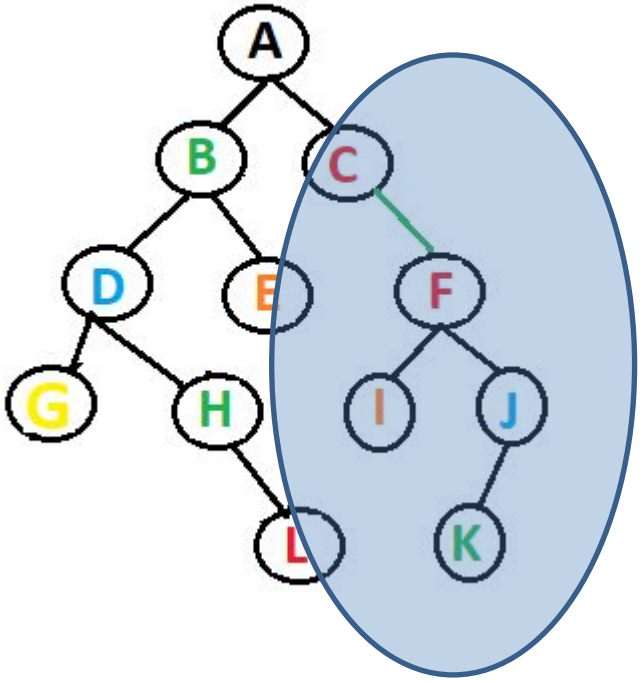


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H, D, E, B,

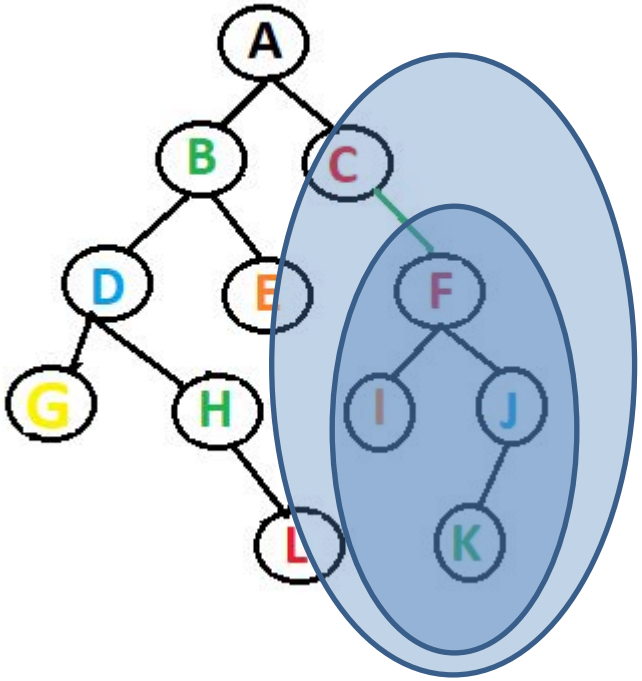


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H, D, E, B,

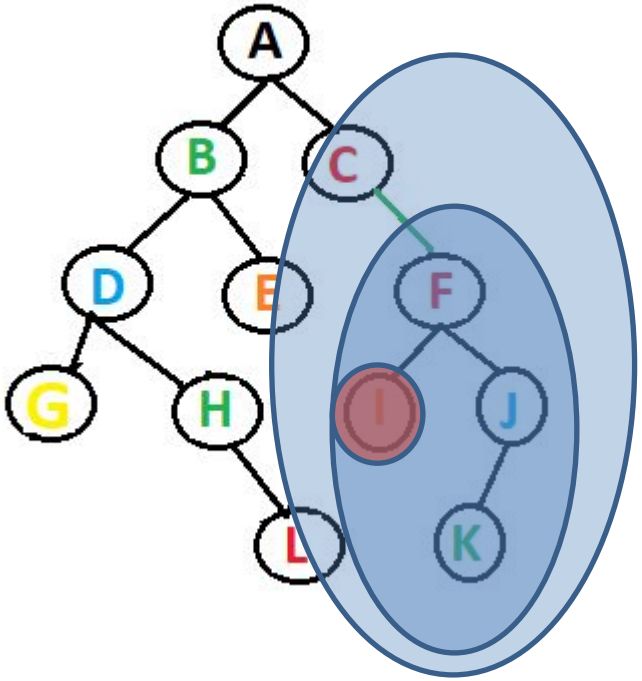


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H, D, E, B, I,

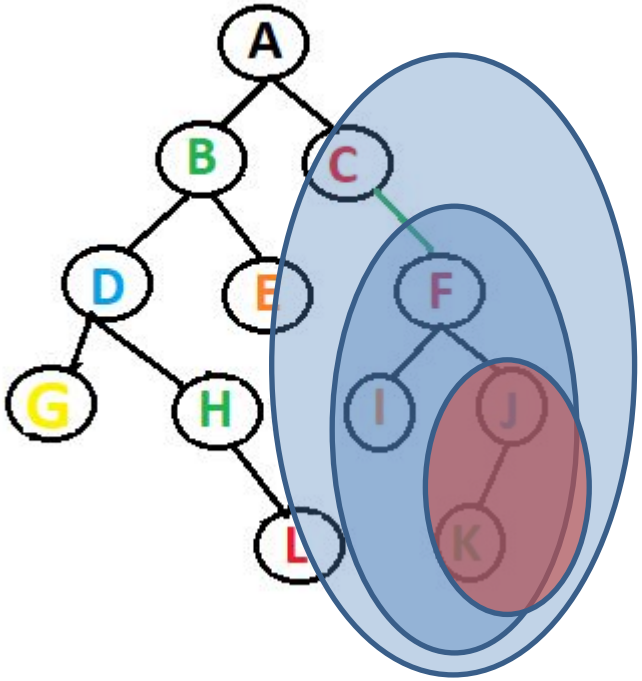


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H, D, E, B, I,

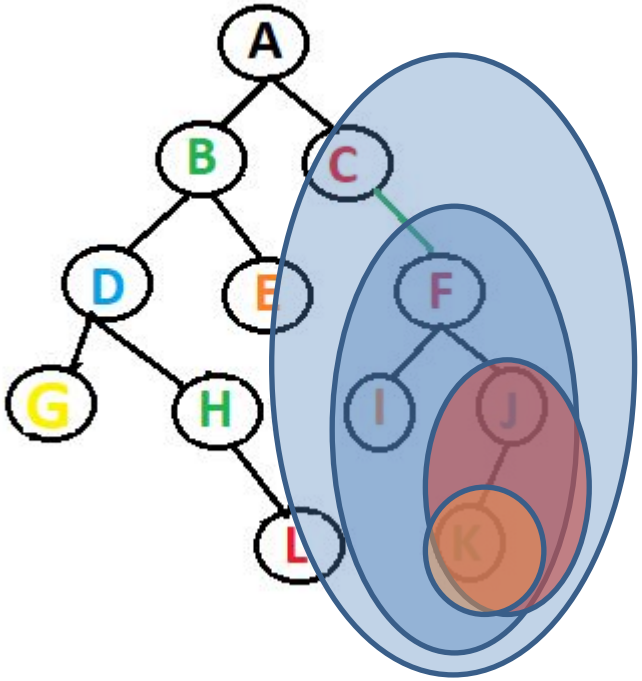


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H, D, E, B, I, K,

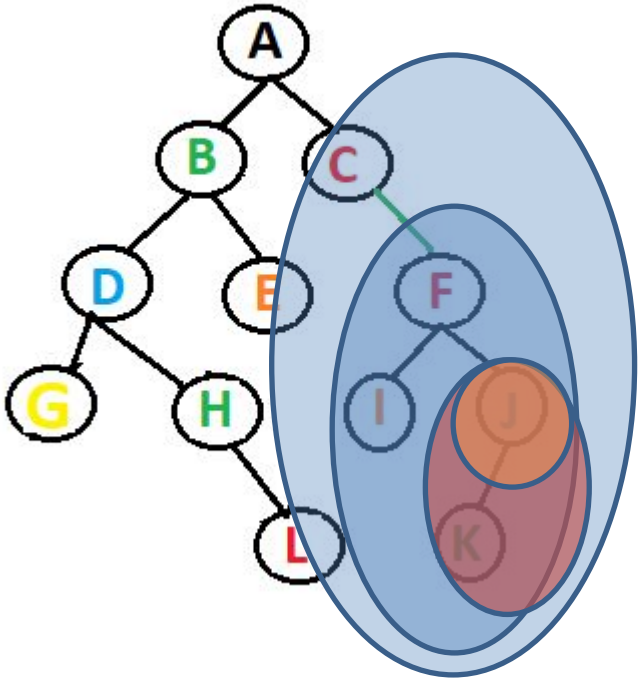


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H, D, E, B, I, K, J,

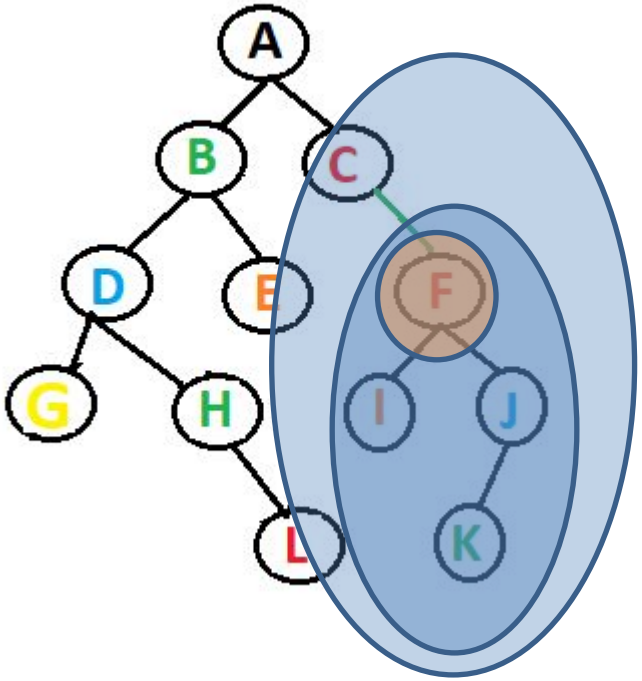


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H, D, E, B, I, K, J, F,

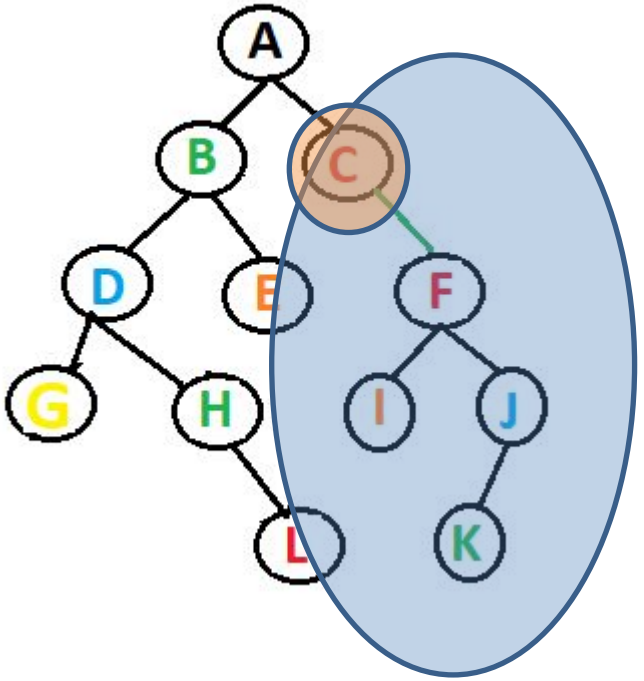


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

Result : G, L, H, D, E, B, I, K, J, F, C,

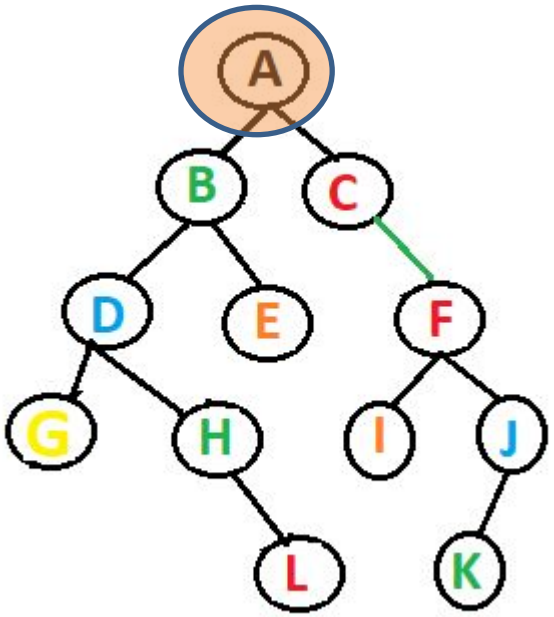


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

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

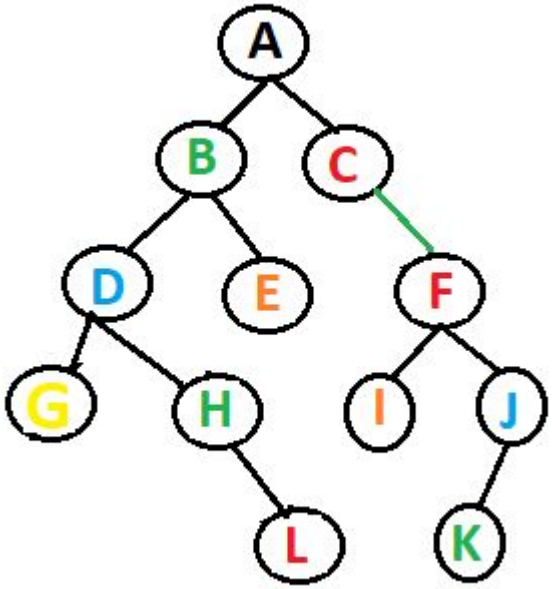


Post-order Traversal

Algorithm: Postorder(Tree)

1. Repeat step 2 – 4 while Tree != Null
2. Postorder(Tree->left)
3. Postorder(Tree->right)
4. Write(Tree->Data)
5. End

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



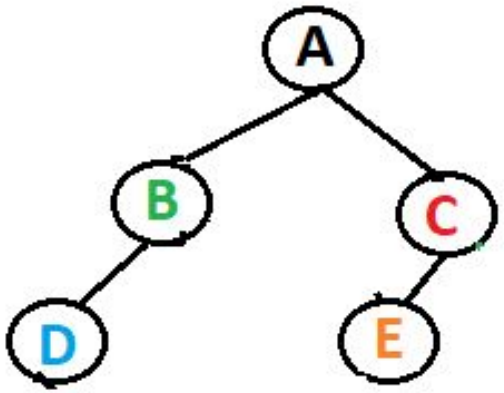
Threaded Binary Trees

- Nodes that does not have right child, have a thread to their inorder successor.
- Node structure:

```
Struct NODE1
{
struct NODE1 *lchild;
int Node1_data;
Struct NODE1 *rchild;
Struct NODE1*trd;
}
```

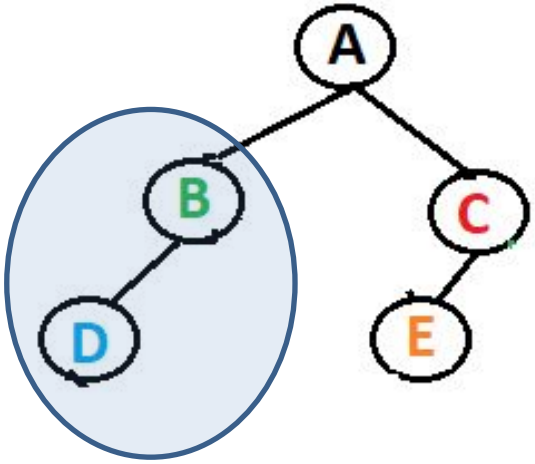
Threaded Binary Trees

- Consider the following tree:
- Inorder traversal : D B A E C



Threaded Binary Trees

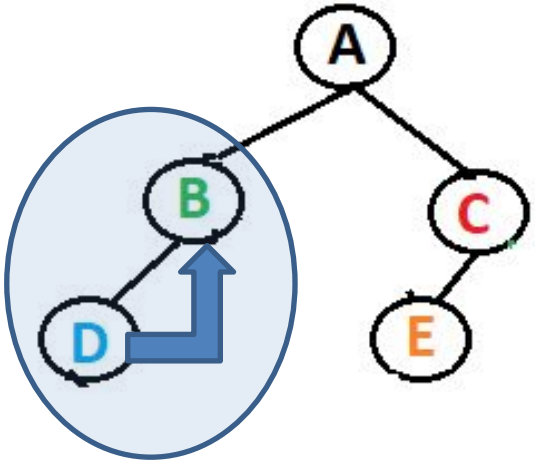
- Consider the following tree:
- Inorder traversal : D B A E C



No Right Child for D – inorder traversal B comes after D

Threaded Binary Trees

- Consider the following tree:
- Inorder traversal : D B A E C

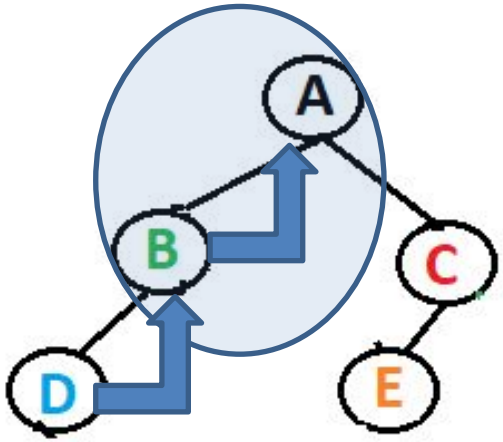


No Right Child for D – inorder traversal B comes after D

Create a thread from D to B

Threaded Binary Trees

- Consider the following tree:
- Inorder traversal : D B A E C

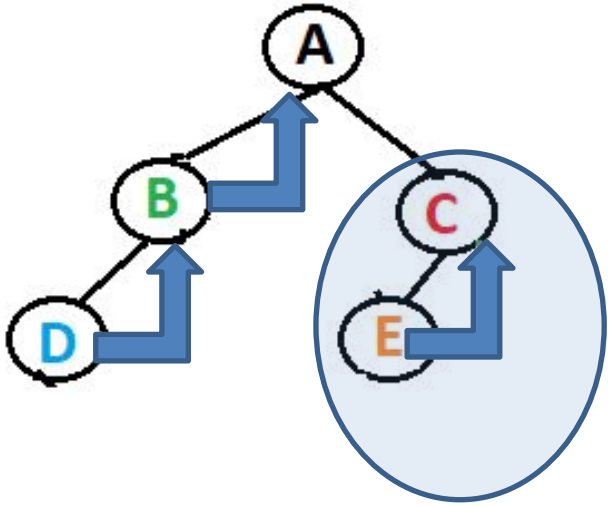


No Right Child for B – inorder traversal A comes after B

Create a thread from B to A

Threaded Binary Trees

- Consider the following tree:
- Inorder traversal : D B A E C

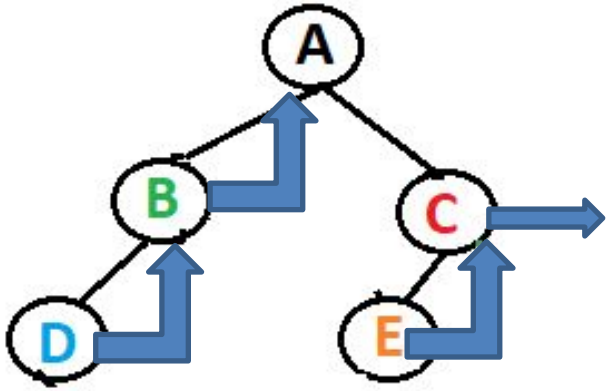


No Right Child for E – inorder traversal C comes after E

Create a thread from E to C

Threaded Binary Trees

- Consider the following tree:
- Inorder traversal : D B A E C



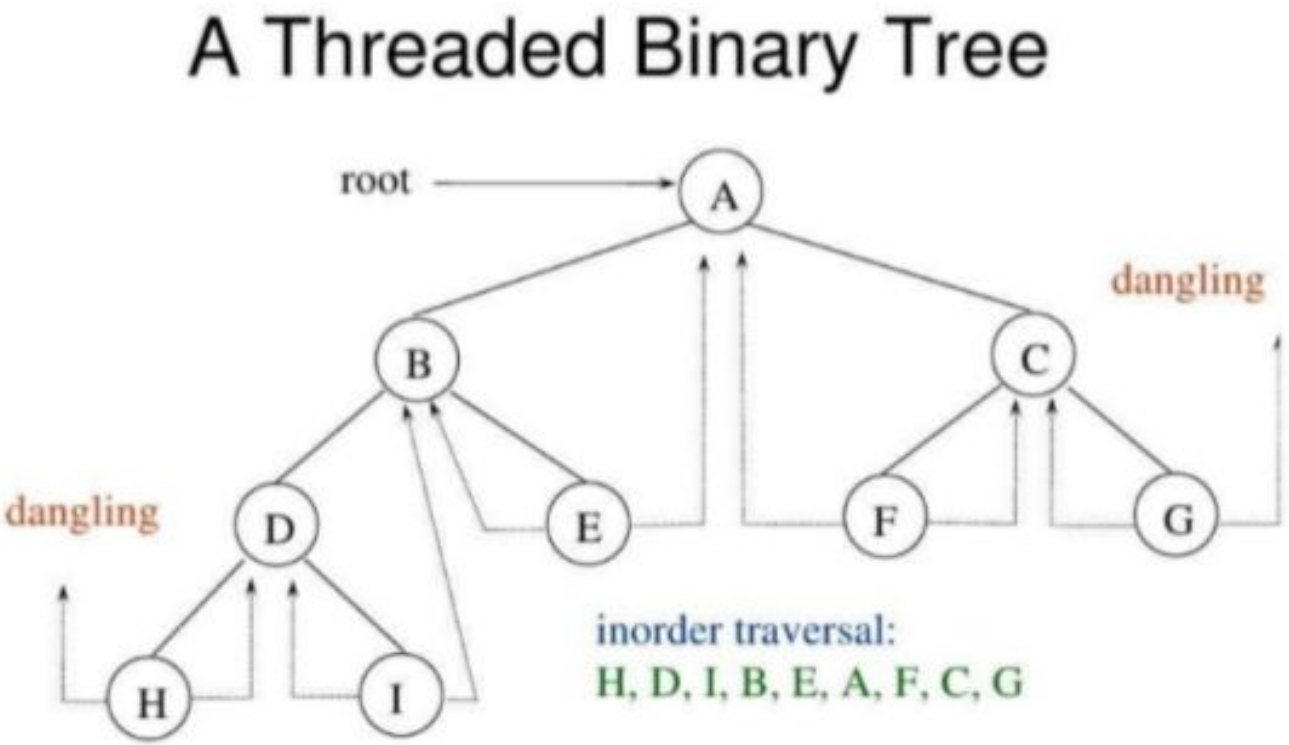
No Right Child for C – inorder traversal C comes at end

Create a hanging tread (dangling thread)

Threaded Binary Tree

- Left Null pointer points to Inorder Predecessor
- Right Null pointer points to Inorder Successor

Threaded Binary Trees



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Session- 7

BINARY SEARCH TREE

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

BINARY SEARCH TREE

- Binary Search Tree is a binary tree in which every **node** contains only smaller values in its left sub tree and only larger values in its right sub tree**.**
- Also called as ***ORDERED*** binary tree

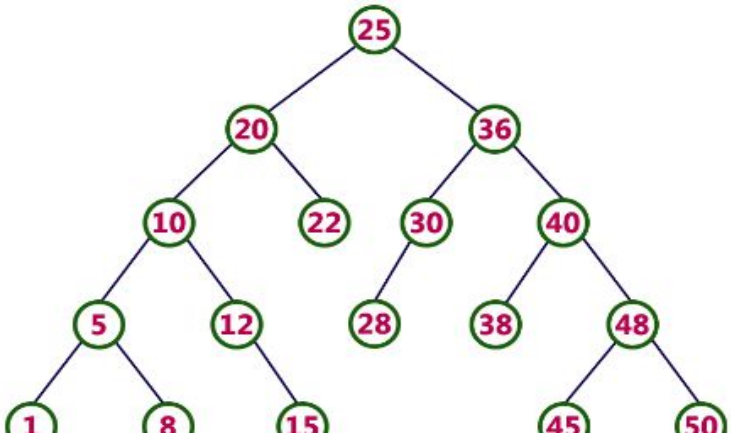
BST– properties:

- It should be Binary tree.
- Left subtree **< Root Node < =** Right subtree

(or)

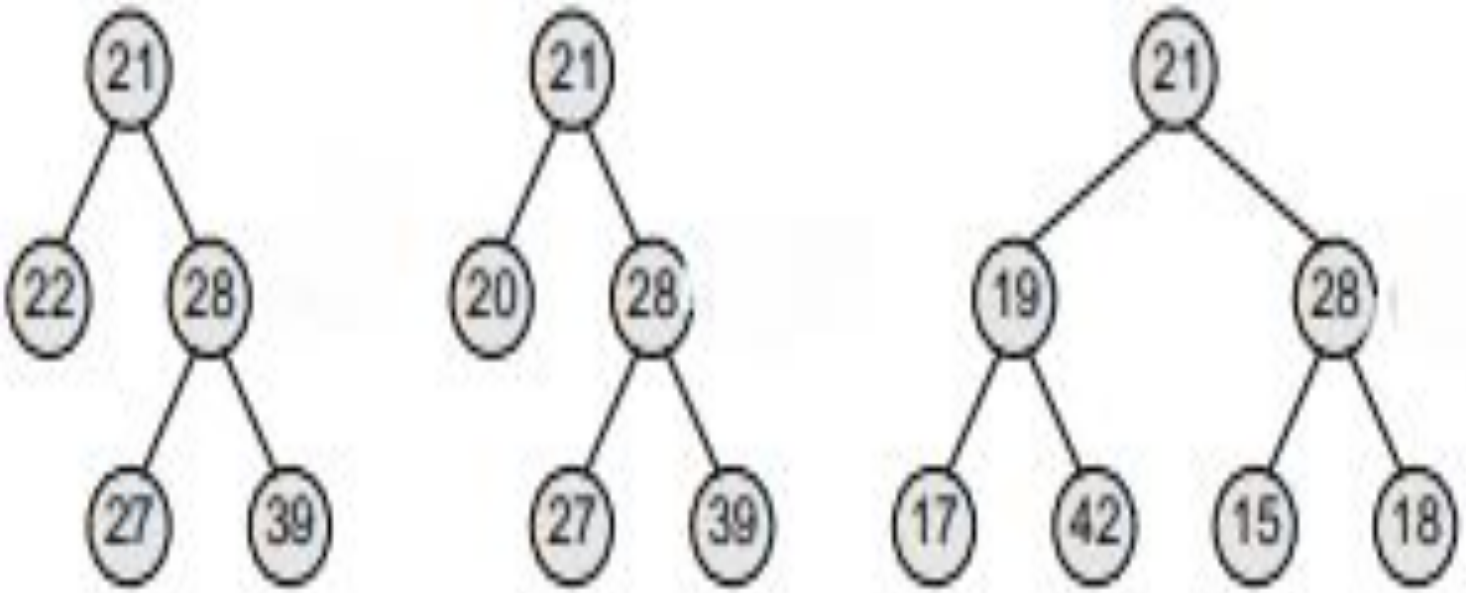
Left subtree < =**Root Node <** Right subtree

Example:

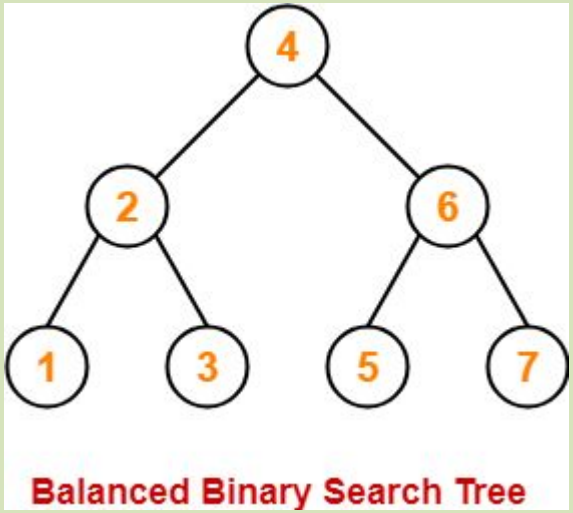
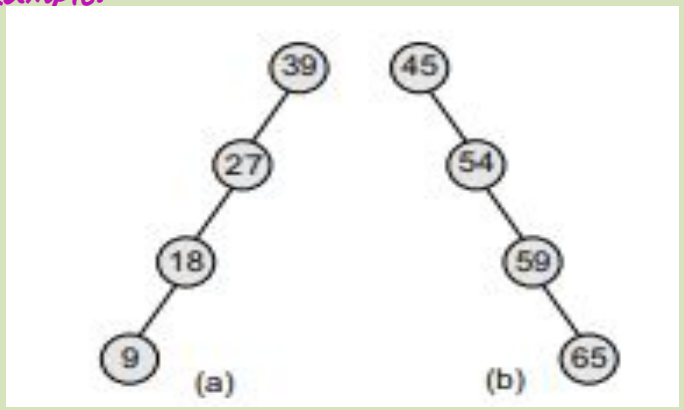


SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Binary search trees or not ?



- **Operations:** Searching, Insertion, Deletion of a Node
- **TimeComplexity :**

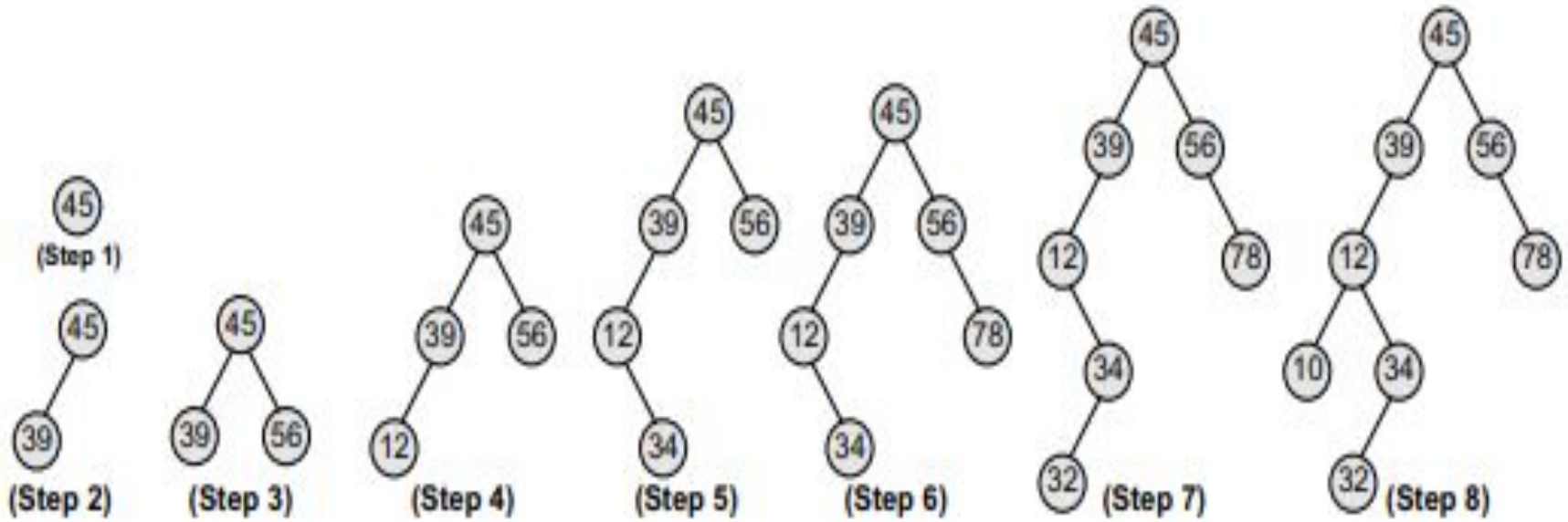
BEST CASE	WORST CASE
Search Operation - O(log n) Insertion Operation- O(log n) Deletion Operation - O(log n) Example:  Balanced Binary Search Tree	Search Operation - O(n) Insertion Operation- O(1) Deletion Operation - O(n) (note:Height of the binary search tree becomes n) Example:  (a) Left skewed, and (b) right skewed binary search trees

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

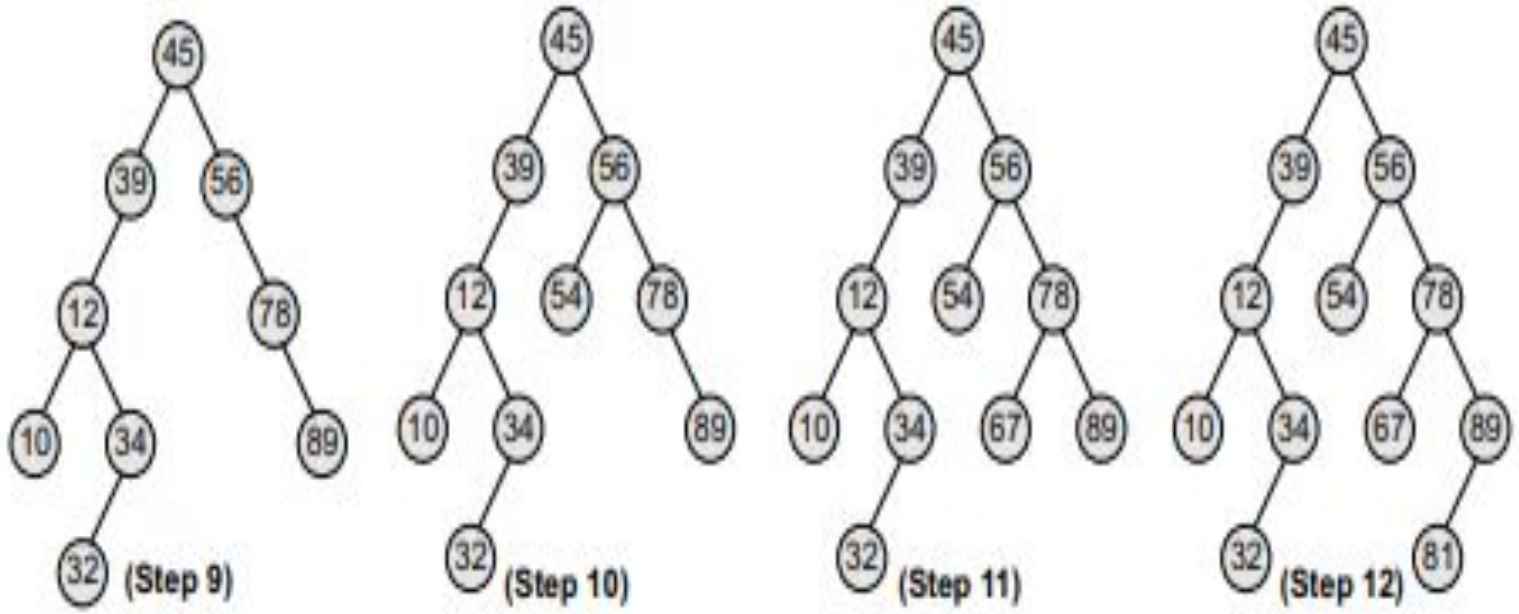
Binary search tree Construction

- Create a binary search tree using the following data elements: 45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81

Solution



Remaining: 89, 54, 67, 81



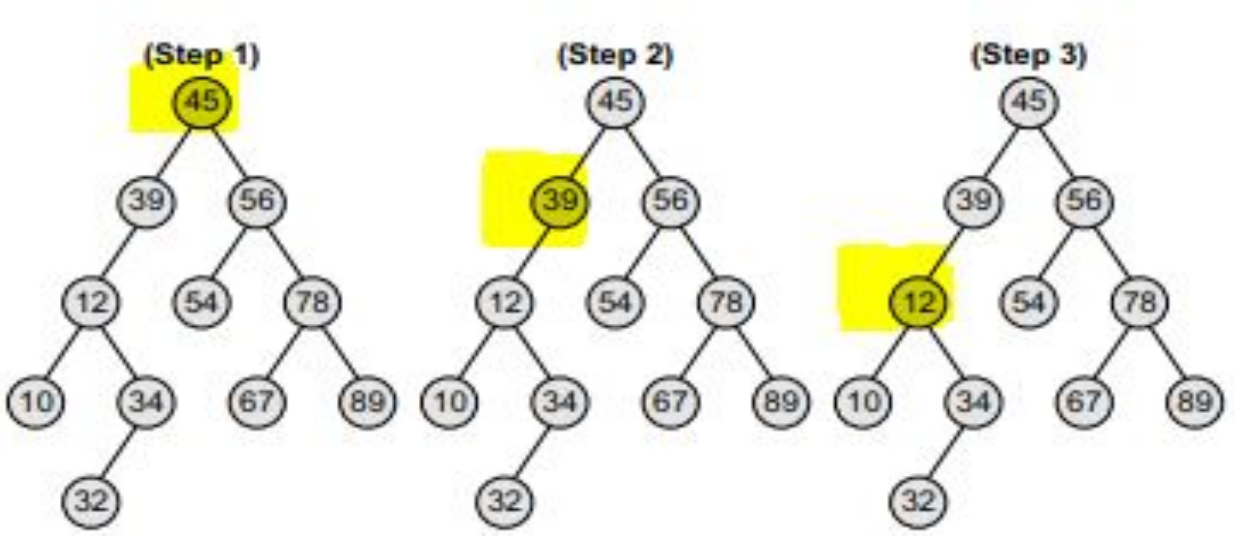
SEARCHING A NODE IN BST

SearchElement (TREE, VAL)

Step 1: IF TREE DATA = VAL OR TREE = NULL
    Return TREE
ELSE
    IF VAL < TREE DATA
    Return searchElement(TREE LEFT, VAL)
ELSE
    Return searchElement(TREE RIGHT, VAL)
    [END OF IF]
    [END OF IF]
Step 2: END

EXAMPLE 1:

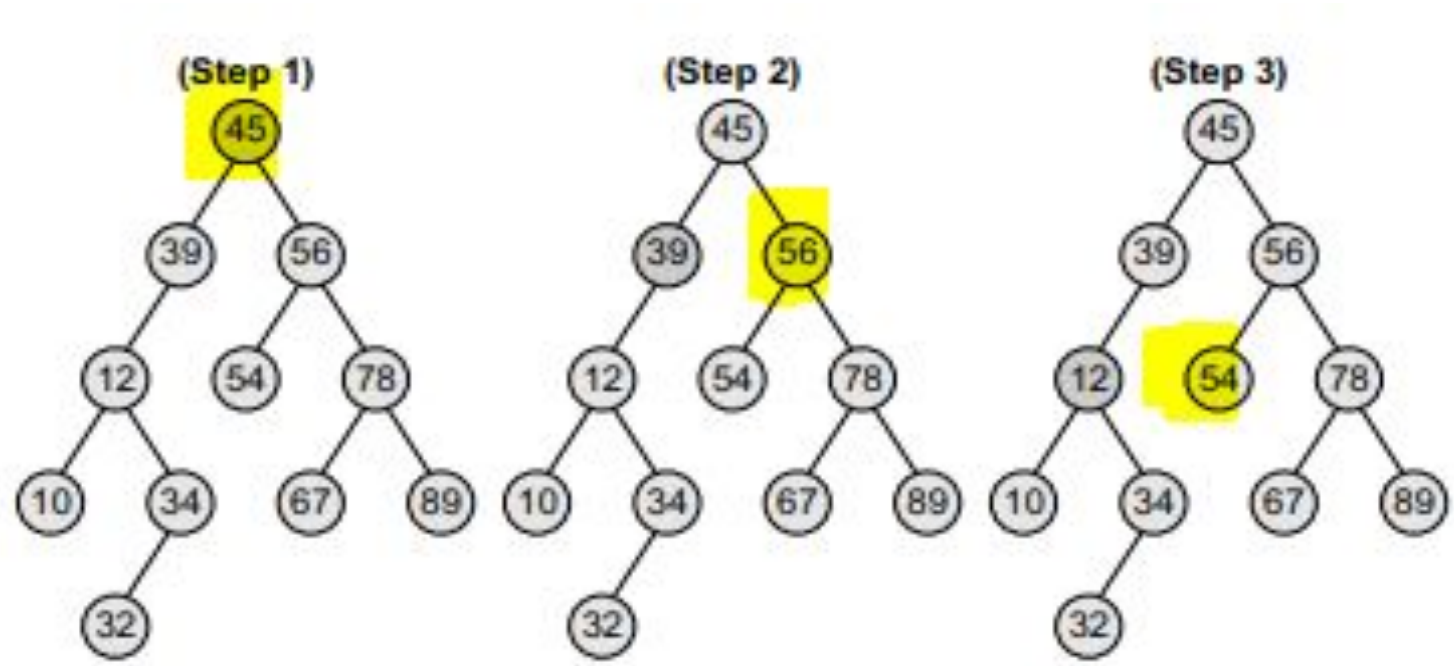
Searching a node with value **12** in the given binary search tree



We start our search from the root node 45.
As 12 < 45, so we search in 45's LEFT subtree.
As 12 < 39, so we search in 39's LEFT subtree.
So, we conclude that 12 is present in the above BST.

EXAMPLE 2:

Searching a node with value **52** in the given binary search tree



We start our search from the root node 45.

As 52 > 45, so we search in 45's RIGHT subtree.

As 52 < 56 so we search in 56's LEFT subtree.

As 52 < 54 so we search in 54's LEFT subtree.

But **54 is leaf node**

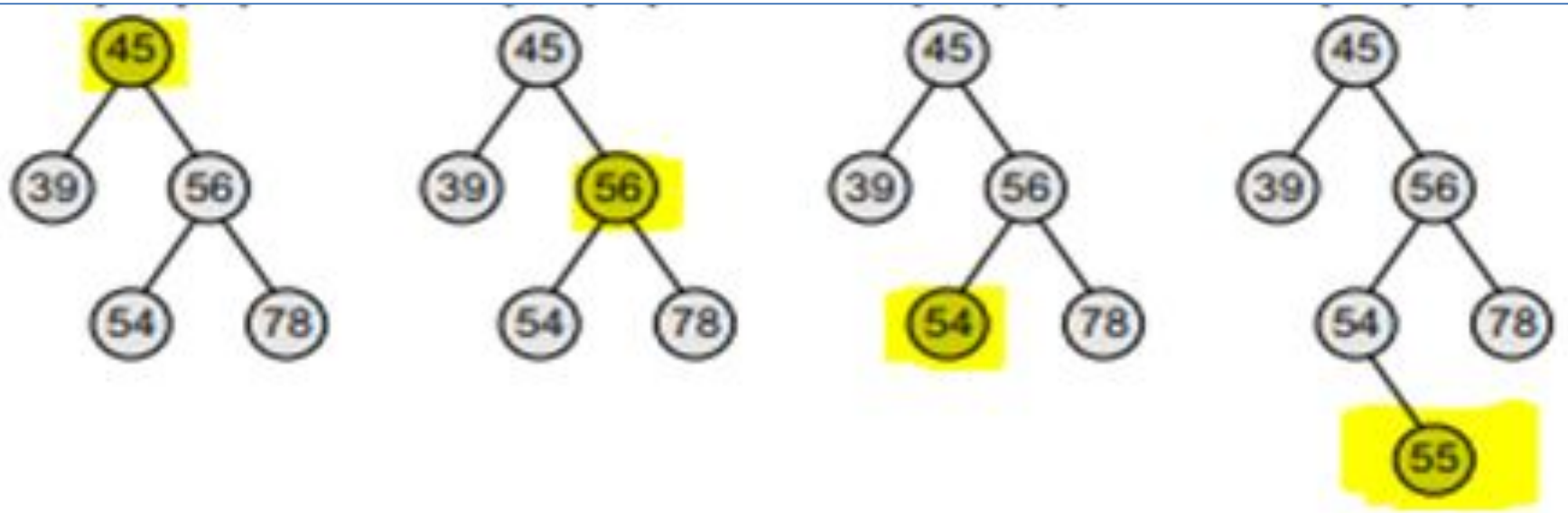
So, we conclude that 52 is not present in the above BST.

INSERTING A NODE IN BST

Insert (TREE, VAL)
Step 1: IF TREE = NULL
Allocate memory for TREE
SET TREE DATA = VAL
SET TREE LEFT = TREE RIGHT = NULL
ELSE
IF VAL < TREE DATA
Insert(TREE LEFT, VAL)
ELSE
Insert(TREE RIGHT, VAL)
[END OF IF]
[END OF IF]

Step 2: END

EXAMPLE : Inserting nodes with values 55 in the given binary search tree



We start searching for value 55 from the root node 45.
As 55 > 45, so we search in 45's RIGHT subtree.
As 55 < 56 so we search in 56's LEFT subtree.
As 55 > 54 so so we **add 55 to 54's right** subtree.

Deletion Operation in BST

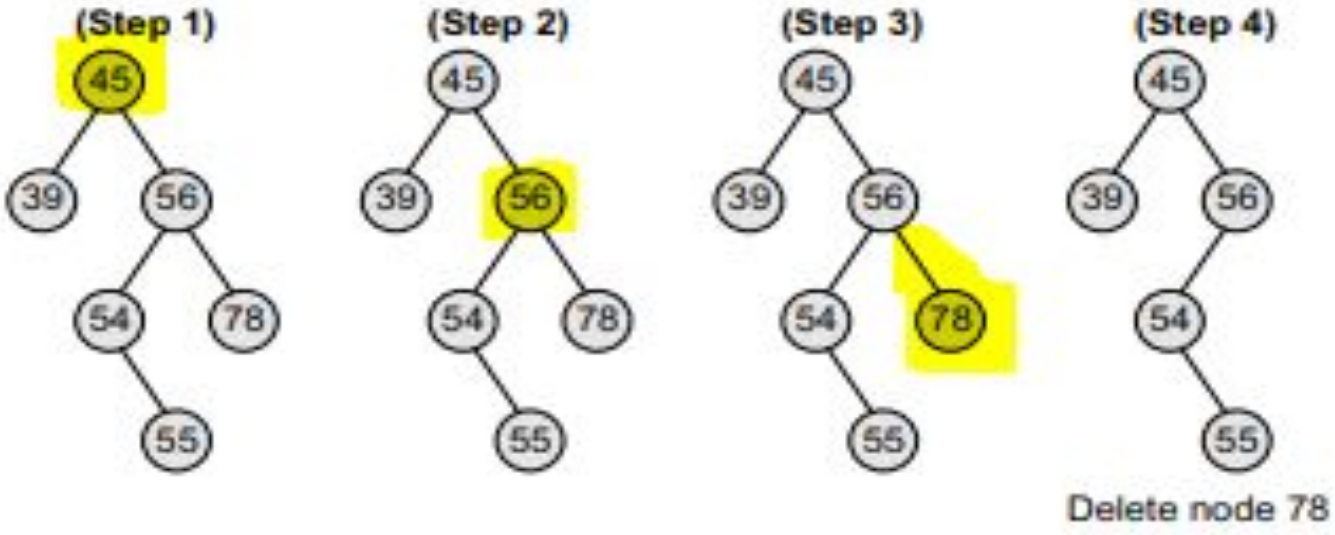
- **Case 1:** Deleting a Leaf node (A node with no children)
- **Case 2:** Deleting a node with one child
- **Case 3:** Deleting a node with two children

```
Delete (TREE, VAL)
Step 1: IF TREE = NULL
        Write "VAL not found in the tree"
ELSE IF VAL < TREE DATA
        Delete(TREE->LEFT, VAL)
ELSE IF VAL > TREE DATA
        Delete(TREE RIGHT, VAL)
ELSE IF TREE LEFT AND TREE RIGHT
        SET TEMP = findLargestNode(TREE LEFT)    ( INORDER  PREDECESSOR)
                                                 (OR)
        SET TEMP = findSmallestNode(TREE RIGHT)    ( INORDER SUCESSOR)
        SET TREE DATA = TEMP DATA
        Delete(TREE LEFT, TEMP DATA) (OR) Delete(TREE RIGHT, TEMP DATA)
ELSE
        SET TEMP = TREE
IF TREE LEFT = NULL AND TREE RIGHT = NULL
        SET TREE = NULL

ELSE IF TREE LEFT != NULL
        SET TREE = TREE LEFT
ELSE
        SET TREE = TREE RIGHT
[END OF IF]
FREE TEMP
[END OF IF]
Step 2: END
```

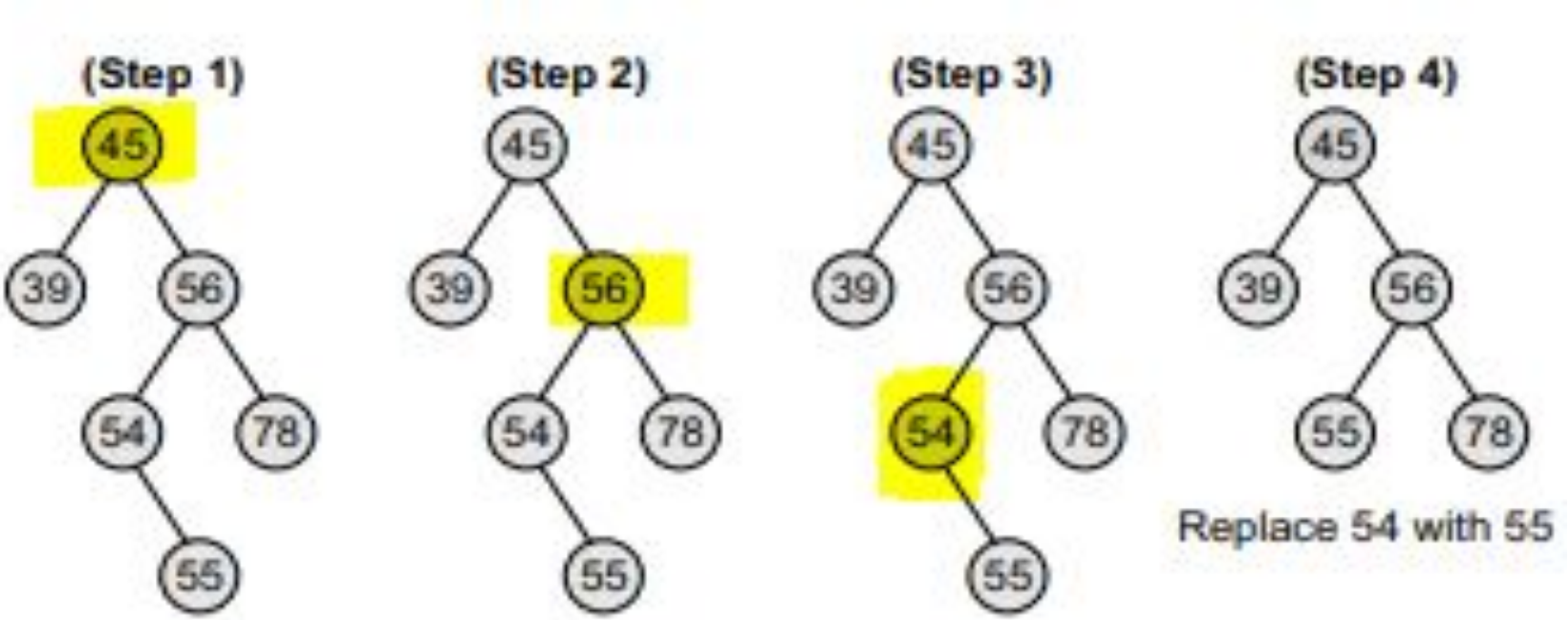
Case 1: Deleting a Leaf node (A node with no children)

EXAMPLE : Deleting node 78 from the given binary search tree



Case 2: Deleting a node with one child

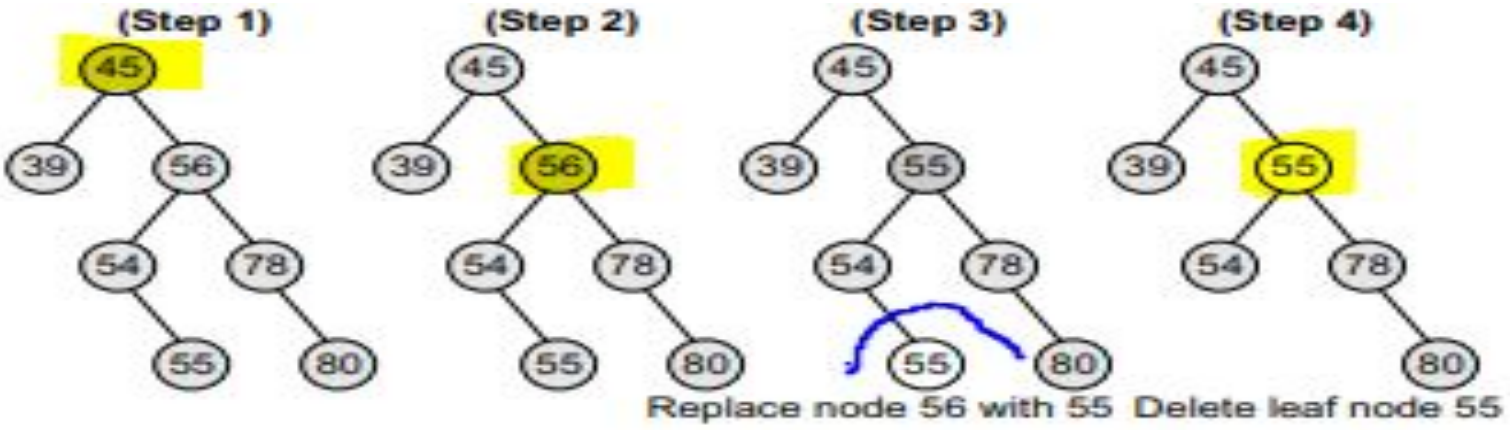
EXAMPLE : Deleting node 54 from the given binary search tree



Case 3: Deleting a node with two children

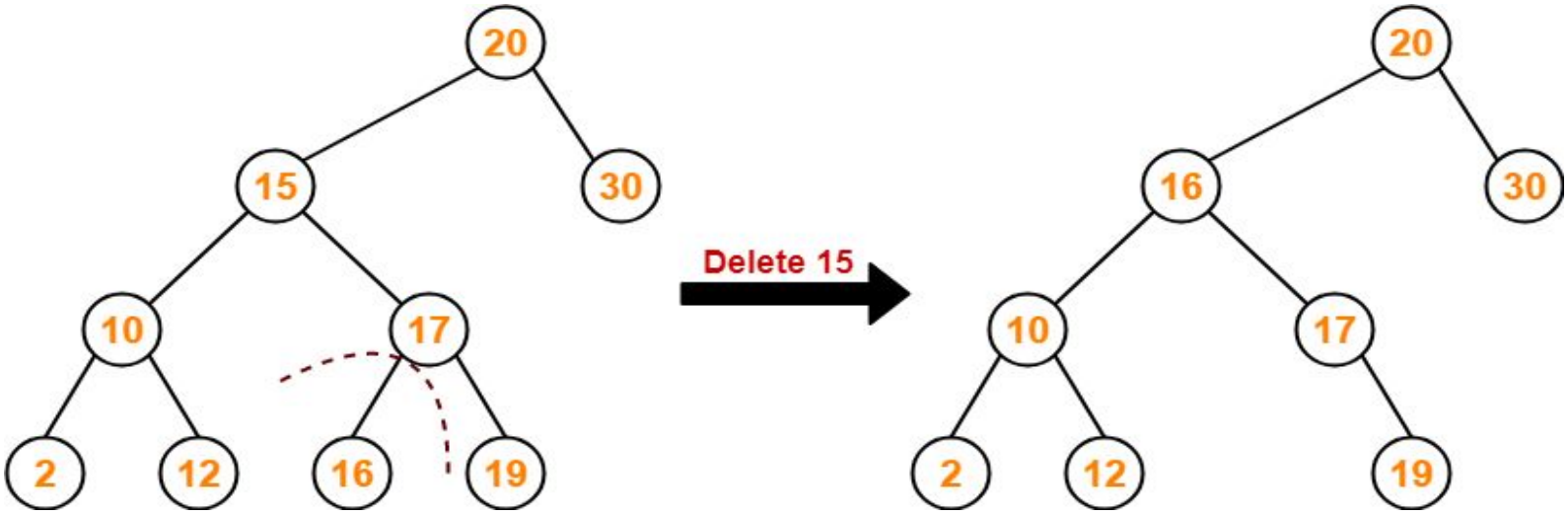
EXAMPLE 1 : Deleting node 56 from the given binary search tree

- Visit to the left sub tree of the deleting node.
- Grab the greatest value element called as in-order predecessor.
- Replace the deleting element with its in-order predecessor.



EXAMPLE 2 : Deleting node 15 from the given binary search tree

- Visit to the right sub tree of the deleting node.
- Pluck the least value element called as inorder successor.
- Replace the deleting element with its inorder successor.



Other possible operations:

- Determining the height of a tree
- Determining the no of nodes a tree
- Determining the no of internal nodes
- Determining the no of external nodes
- Determining the mirror images
- Removing the tree
- Finding the smallest node
- Finding the largest node

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Session-8

AVL TREE

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

AVL TREE

- Named after Adelson-Velskii and Landis as AVL tree
- Also called as *self-balancing* binary search tree

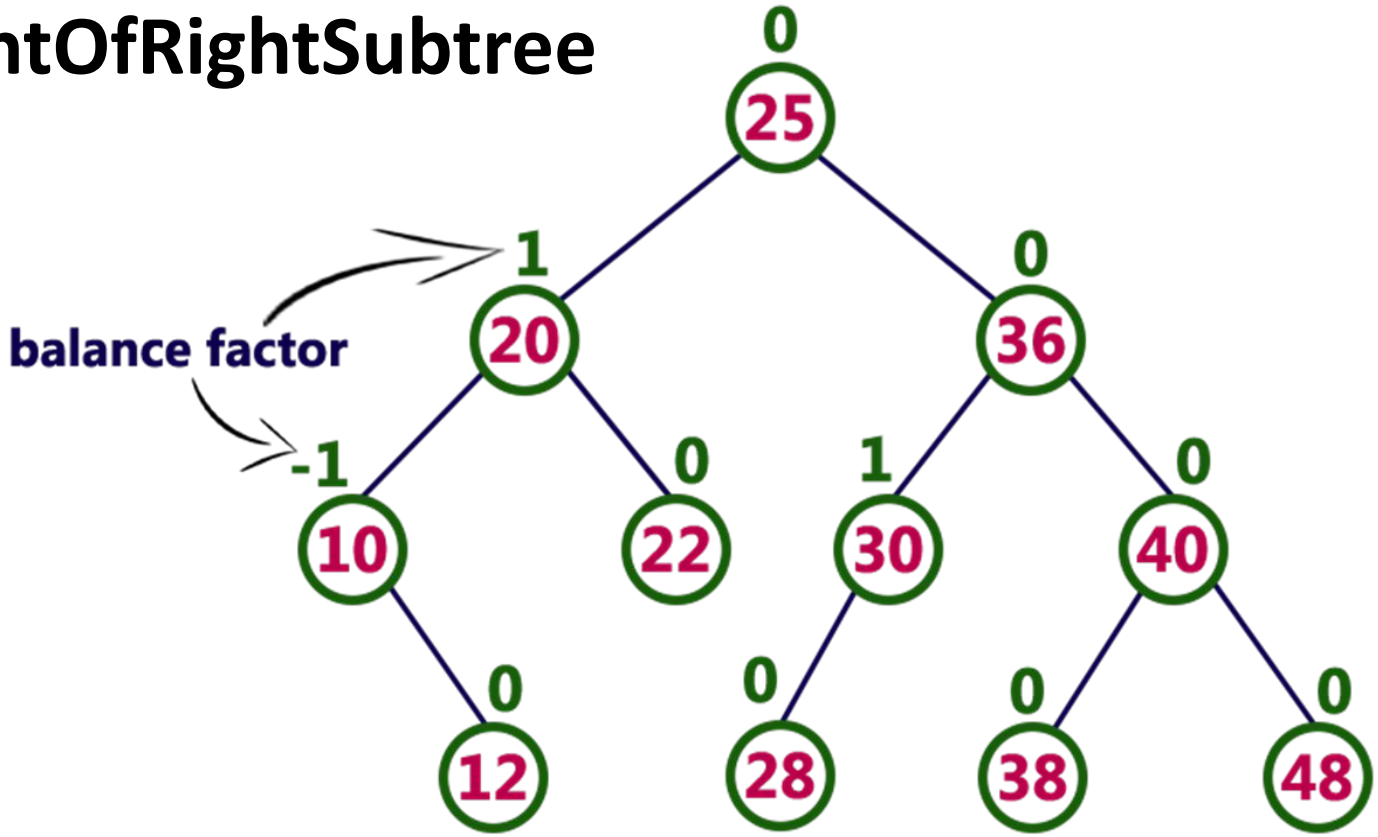
AVL tree – properties:

- It should be Binary search tree
- **Balancing factor:** balance of every node is either -1 or 0 or 1
  where **balance**(*node*) = height(*node*.left subtree) – height(*node*.right subtree)
  - Maximum possible number of nodes in AVL tree of height H
    $= 2^{H+1} - 1$
- **Operations:** Searching, Insertion, Deletion of a Node
- **TimeComplexity : O(log n)**

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

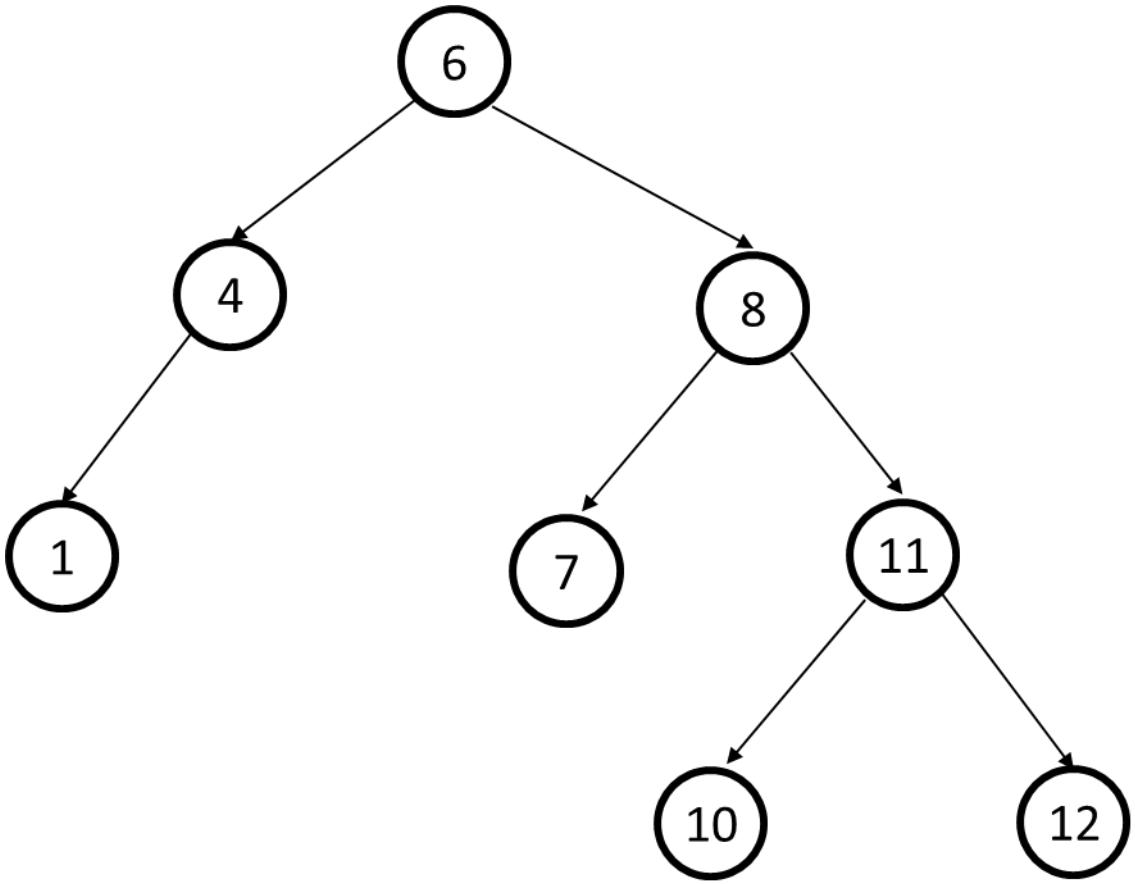
Balancing Factor

- **Balance factor = heightOfLeftSubtree – heightOfRightSubtree**



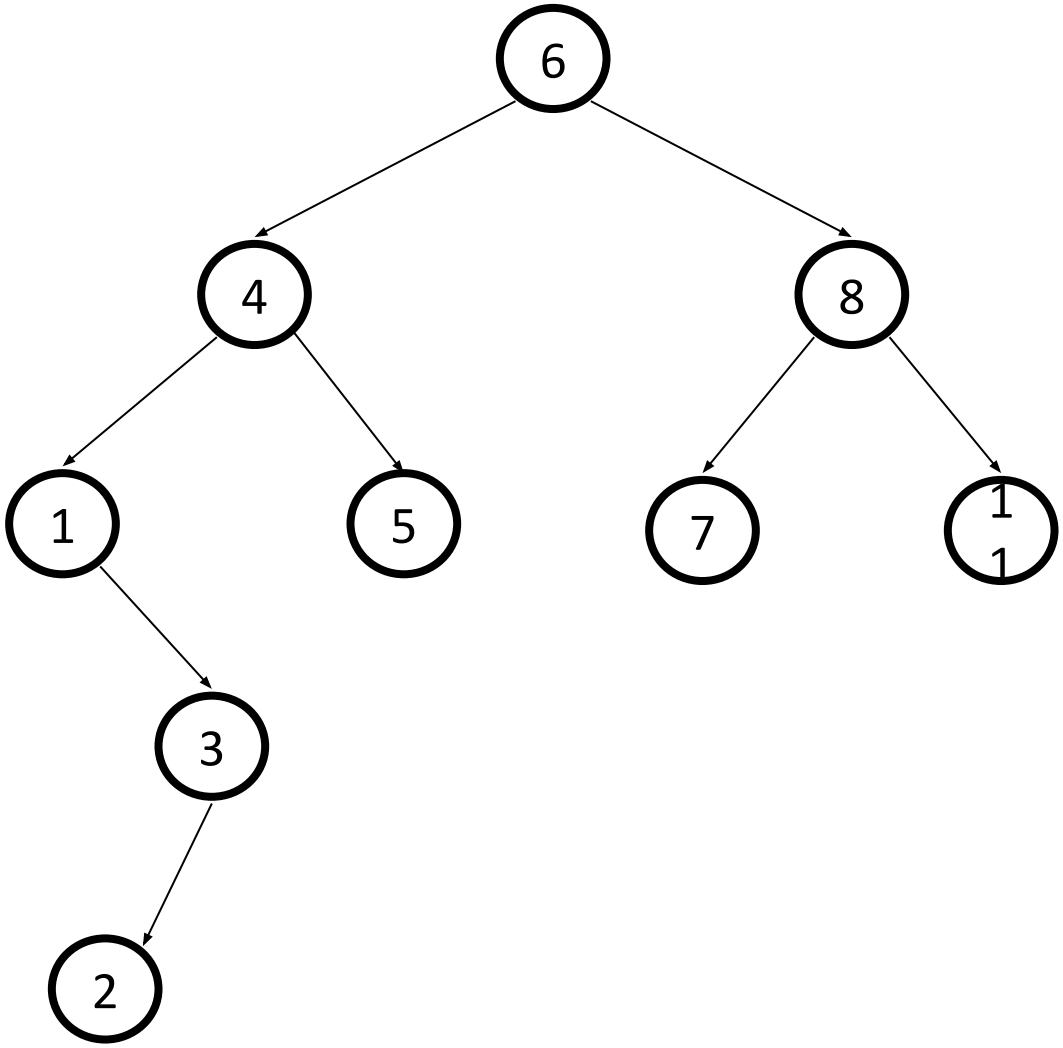
SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Example 1 : Check - AVL Tree?



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Example 2: Check - AVL Tree?



operations on AVL tree

1. SEARCHING
2. INSERTION
3. DELETION

Search Operation in AVL Tree

ALGORITHM:

STEPS:

1 : Get the search element

2 : check search element **==** root node in the tree.

3 : If both are **exact match**, then display "**element found**" and end.

4 : If both are **not matched**, then check whether search element is **<** or **>** than that node value.

5: If search element is **<** then continue search in **left sub tree**.

6: If search element is **>** then continue search in **right sub tree**.

7: **Repeat** the step from **1 to 6** until we find the **exact element**

8: Still the search element is not found after reaching the leaf node display "**element not found**".

INSERTION or DELETION

- After performing any operation on AVL tree - the **balance factor** of **each node** is to be **checked**.
- After insertion or deletion there exists either any one of the following:

Scenario 1:

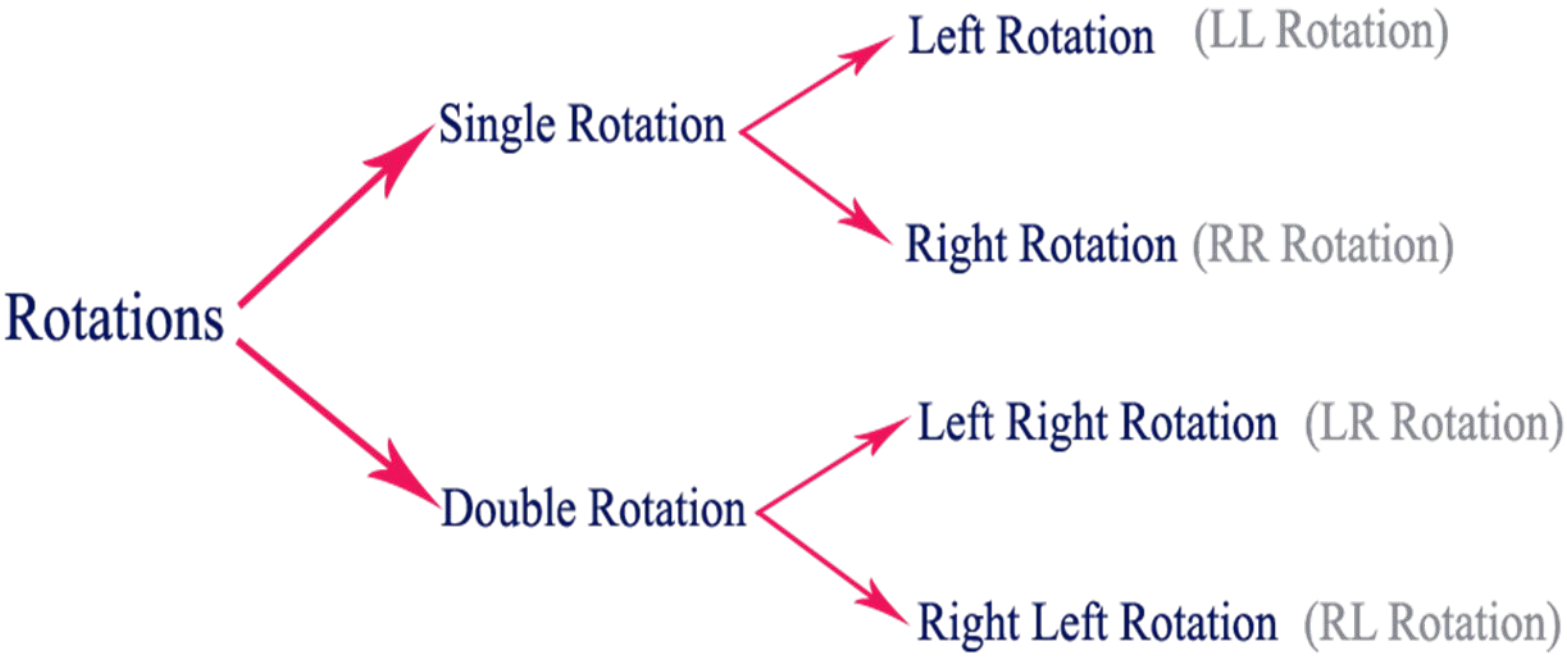
- After insertion or deletion , the balance factor of each node **is either 0 or 1 or -1.**
- If so AVL tree is considered to be **balanced**.
- The operation ends.

Scenario 1:

- After insertion or deletion, the balance factor **is not 0 or 1 or -1** for at least one node then
- The AVL tree is considered to be **imbalanced**.
- If so, **Rotations** are need to be performed **to balance** the tree in order to **make it as AVL TREE.**

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

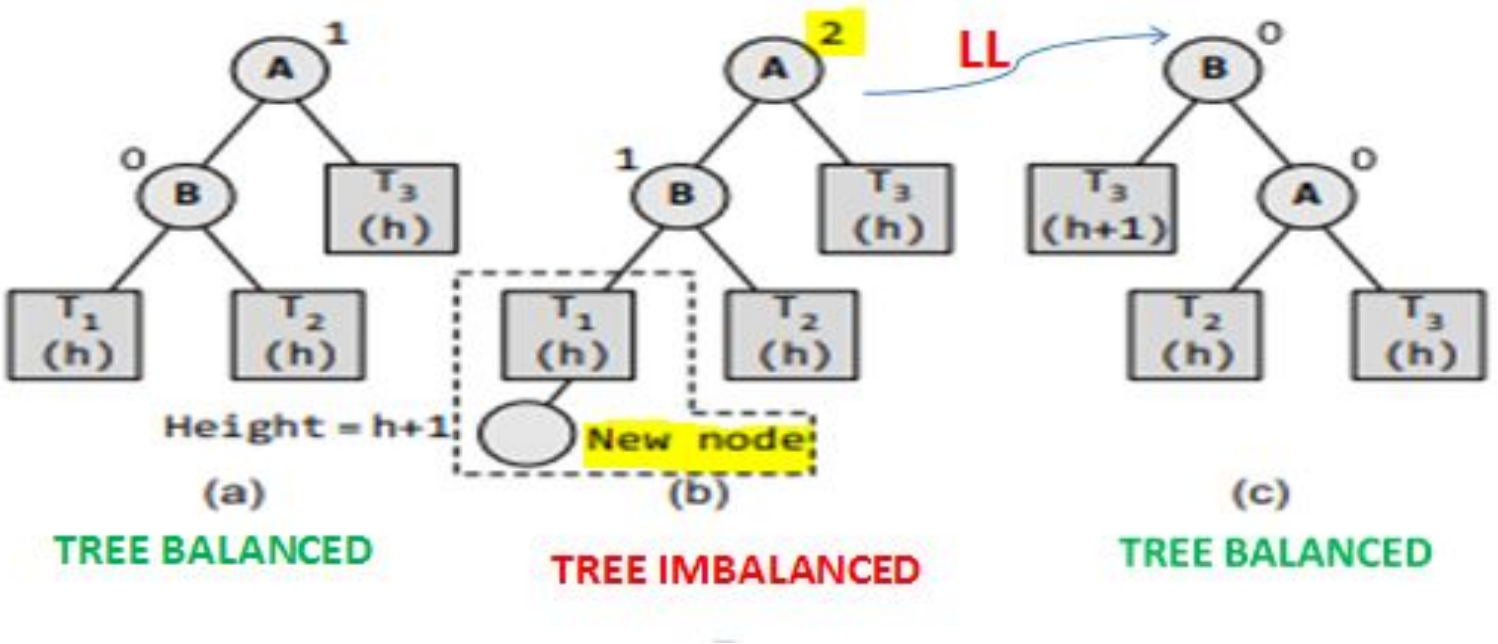
AVL TREE ROTATION



SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

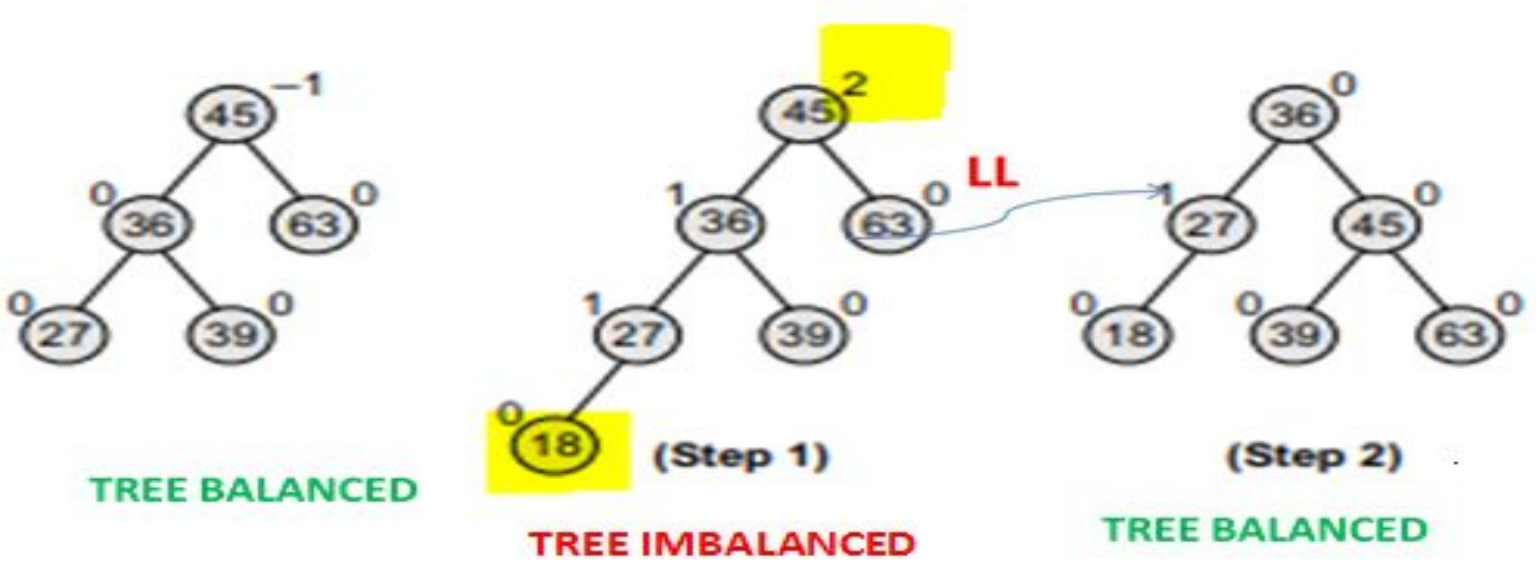
LL ROTATION

When new node is inserted in the left sub-tree of the left sub-tree of the critical



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

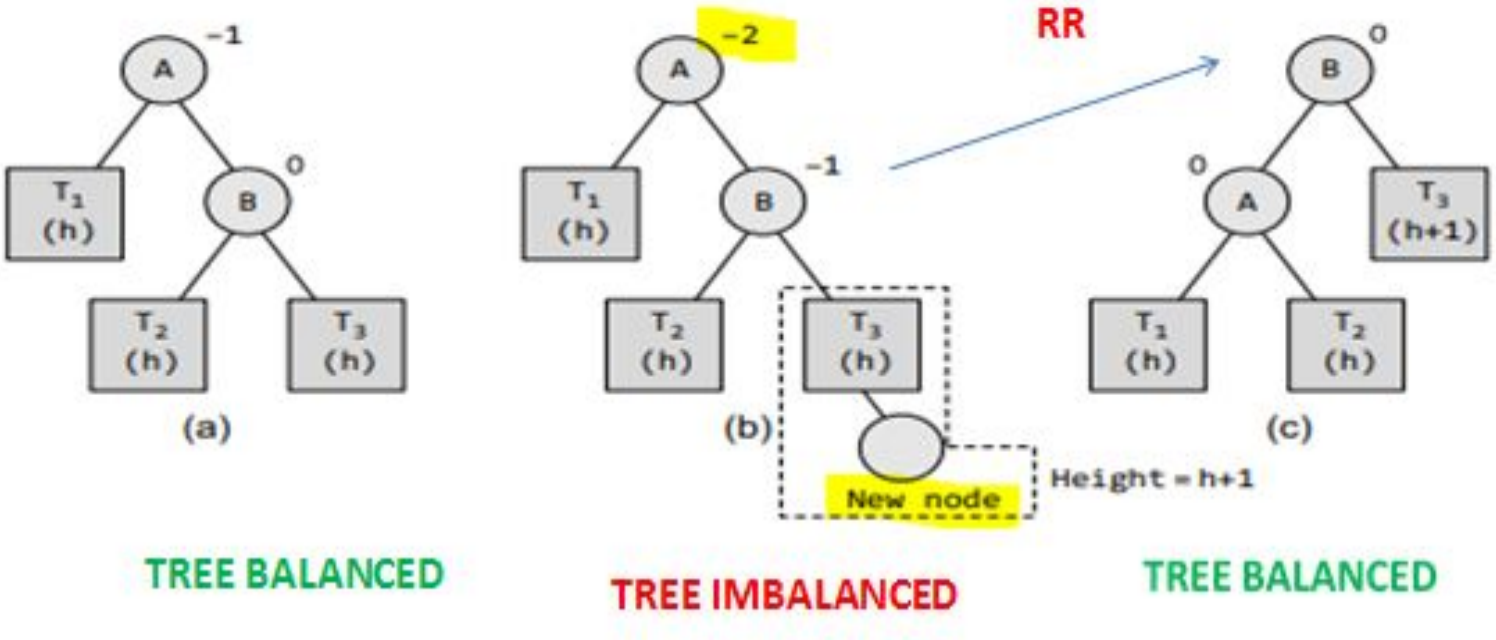
LL ROTATION- Example



SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

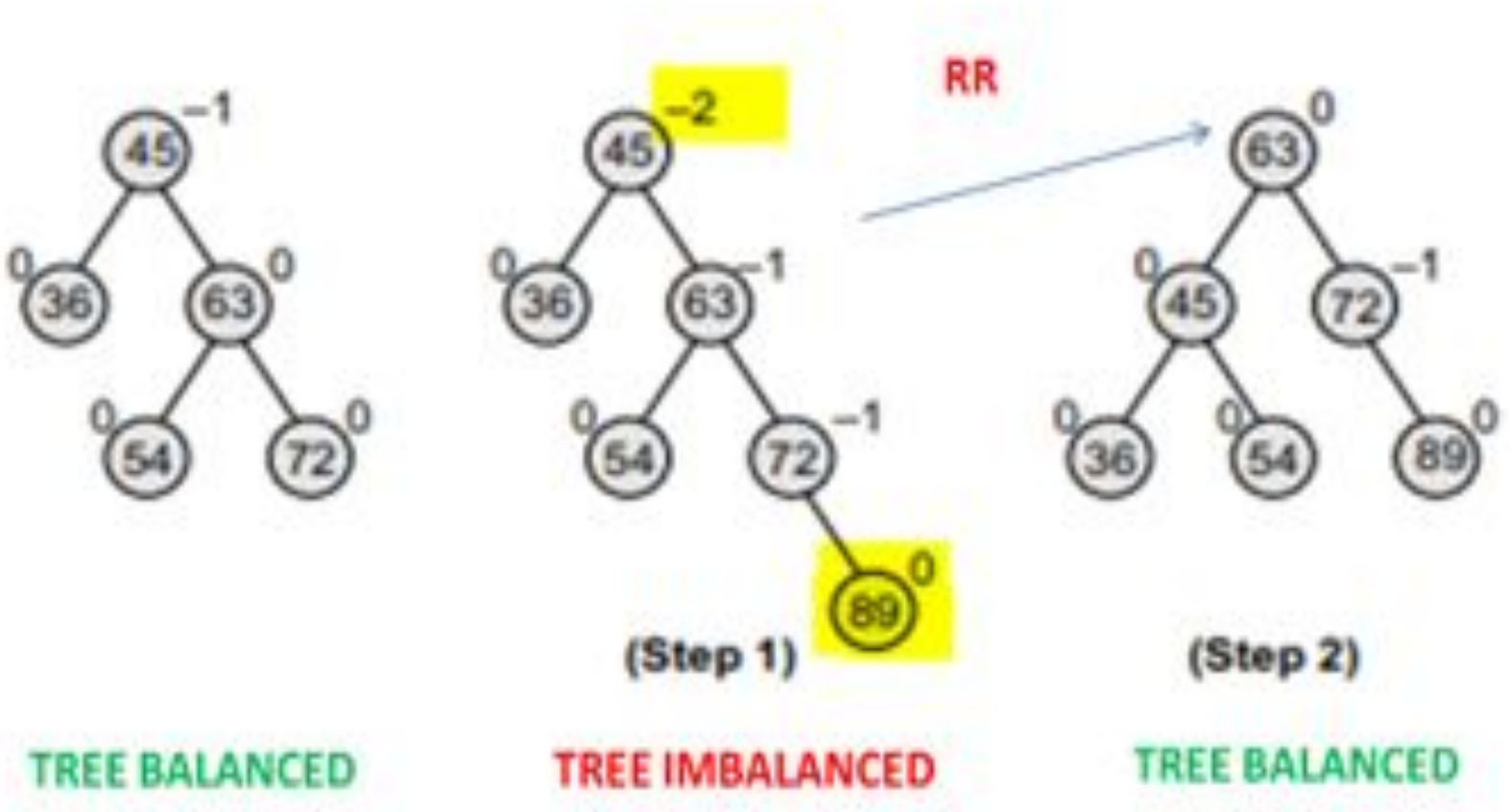
RR ROTATION

When new node is inserted in the right sub-tree of the right sub-tree of the critical



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

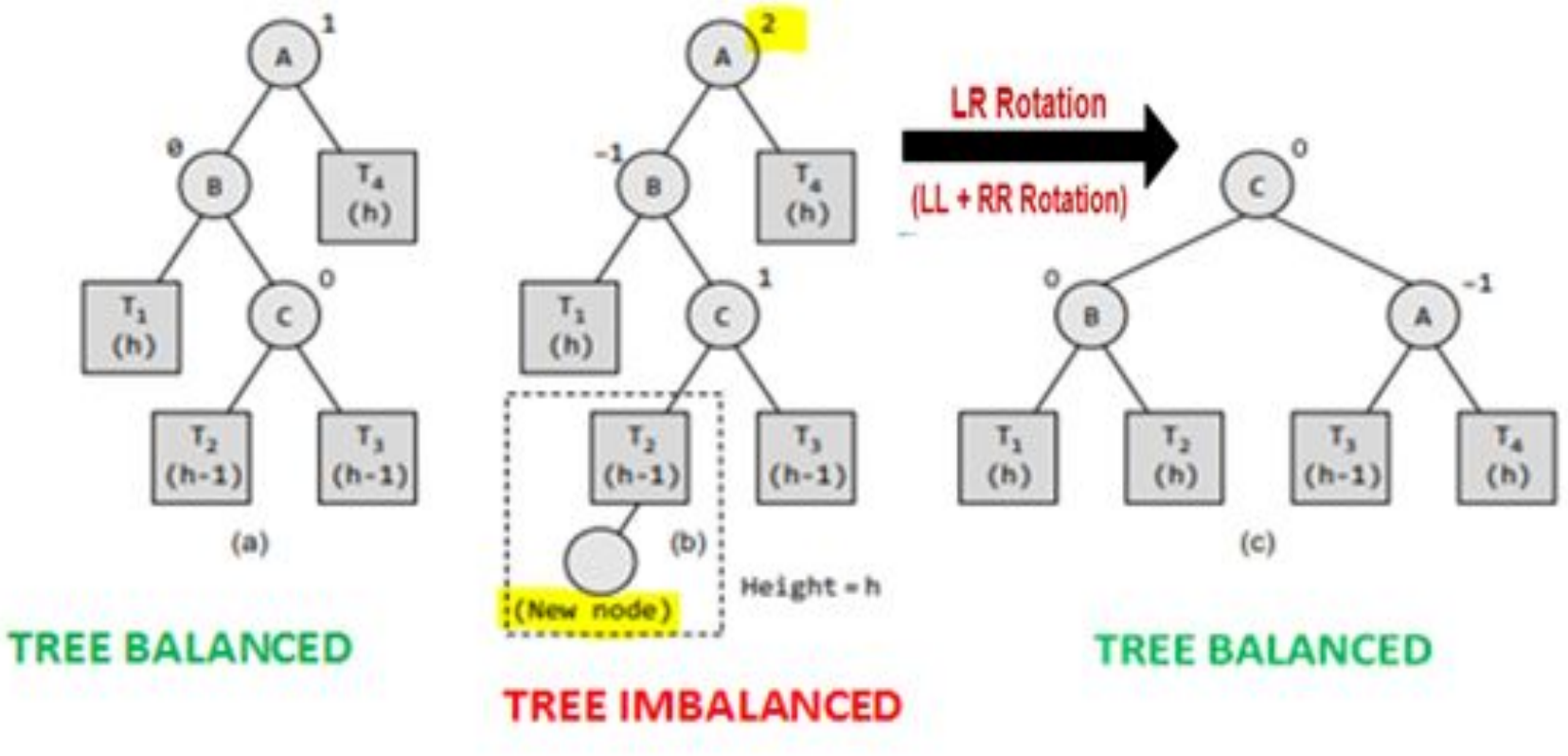
RR Rotation - Example



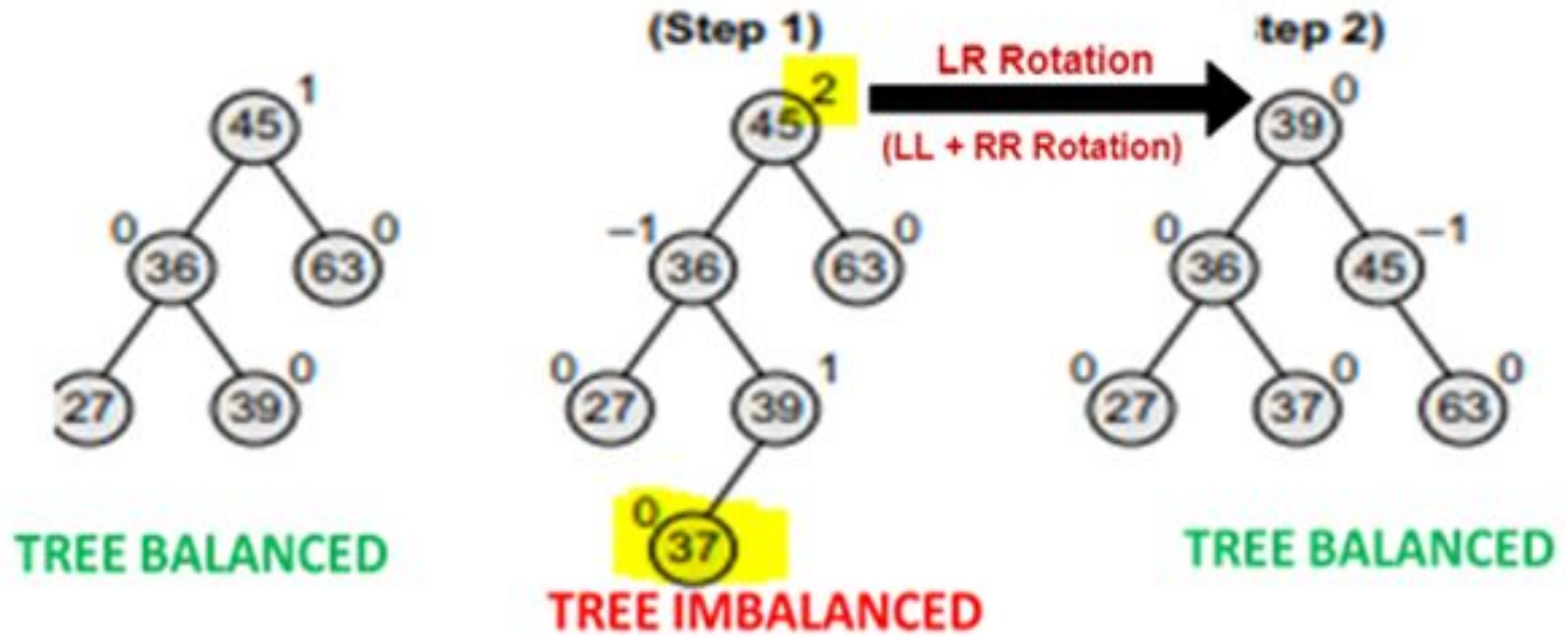
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

LR ROTATION

When new node is inserted in the left sub-tree of the right sub-tree of the critical



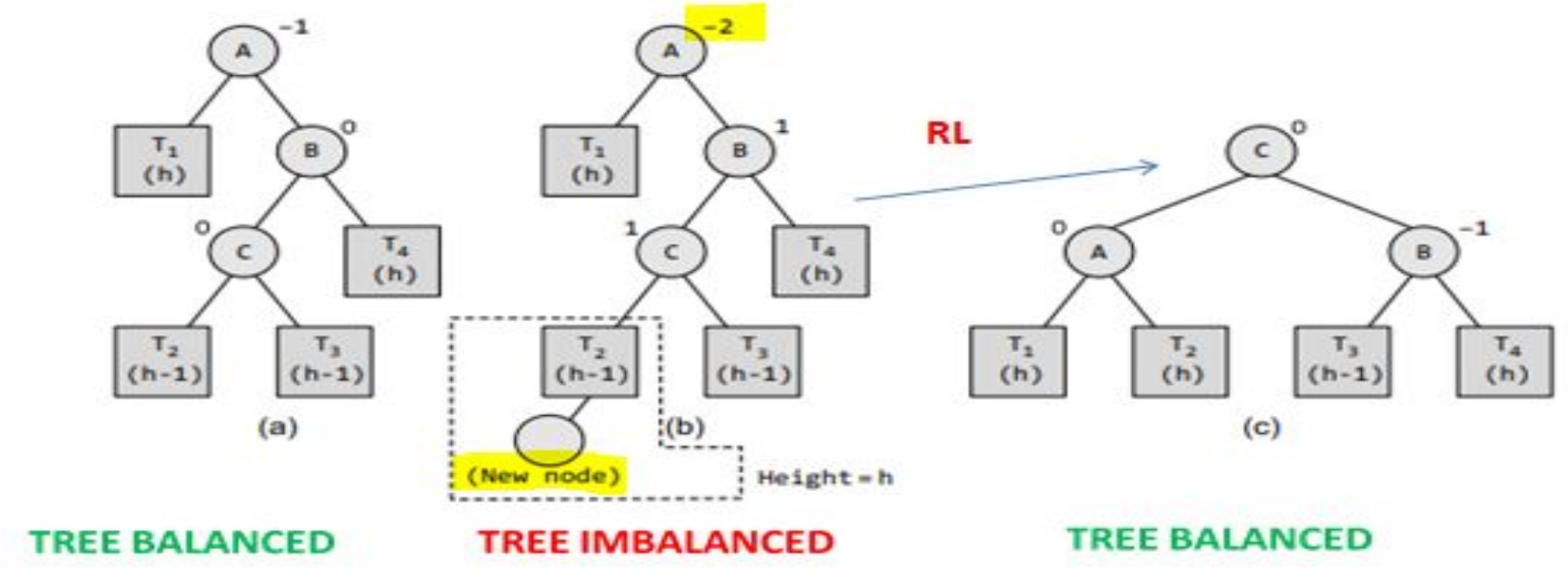
LR Rotation - Example



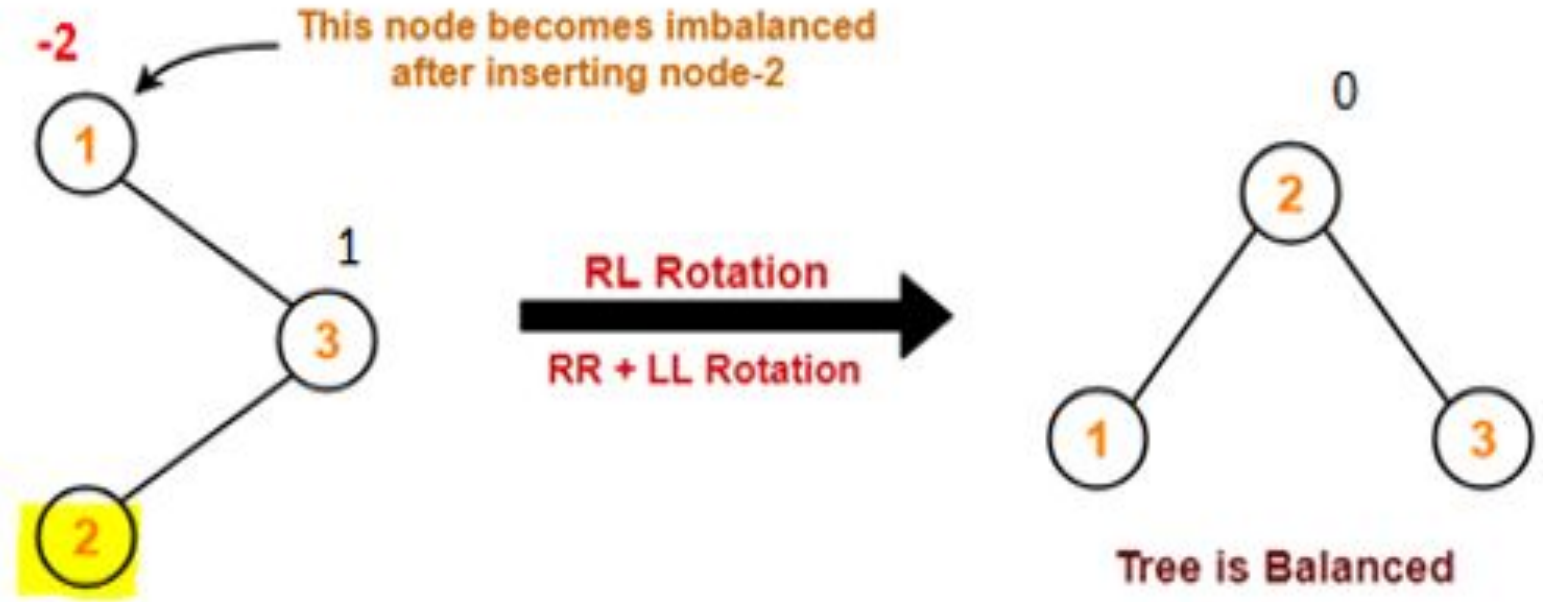
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

RL ROTATION

When new node is inserted in the right sub-tree of the left sub-tree of the critical



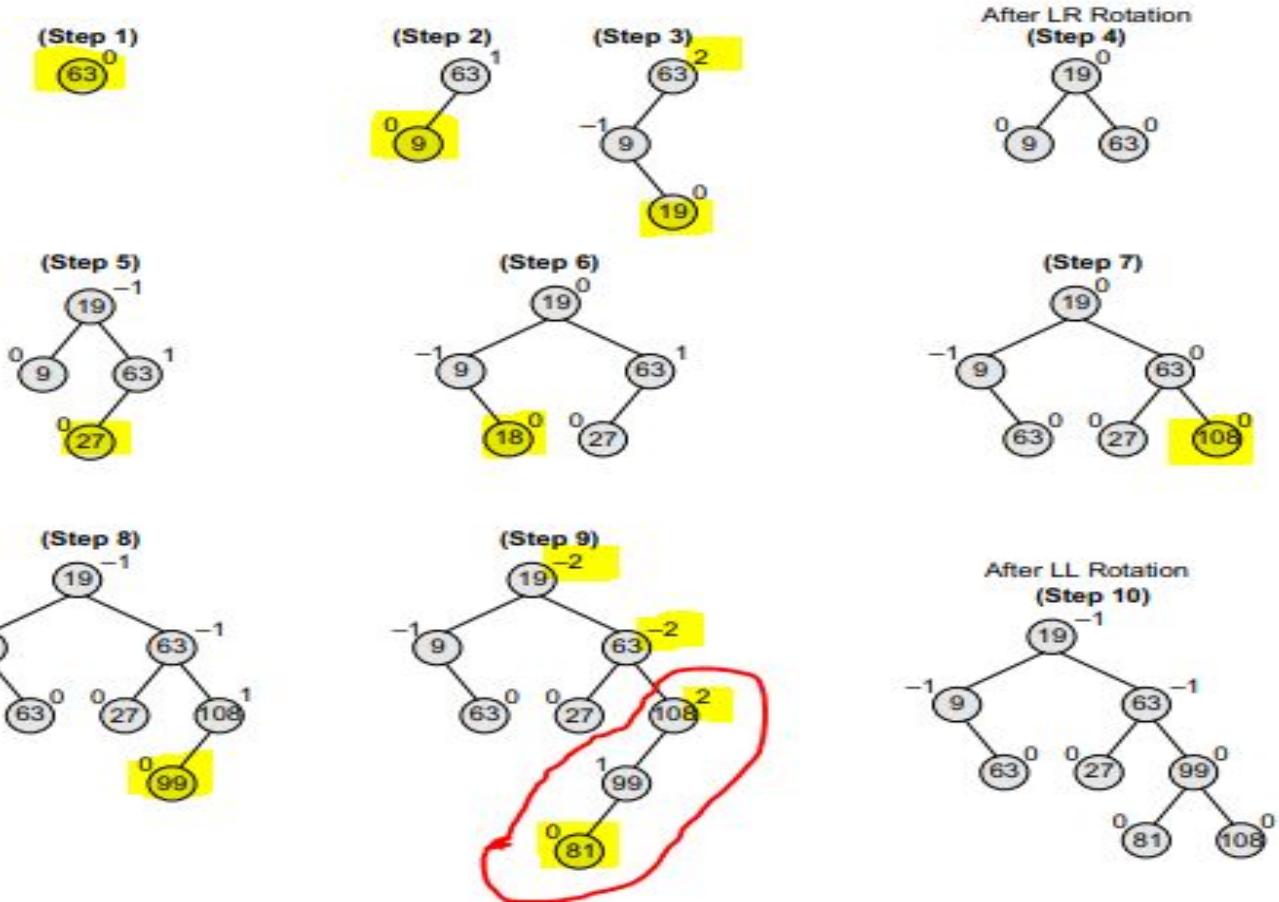
RL Rotation - Example



SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

AVL TREE CONSTRUCTION / NODE INSERTION - Example

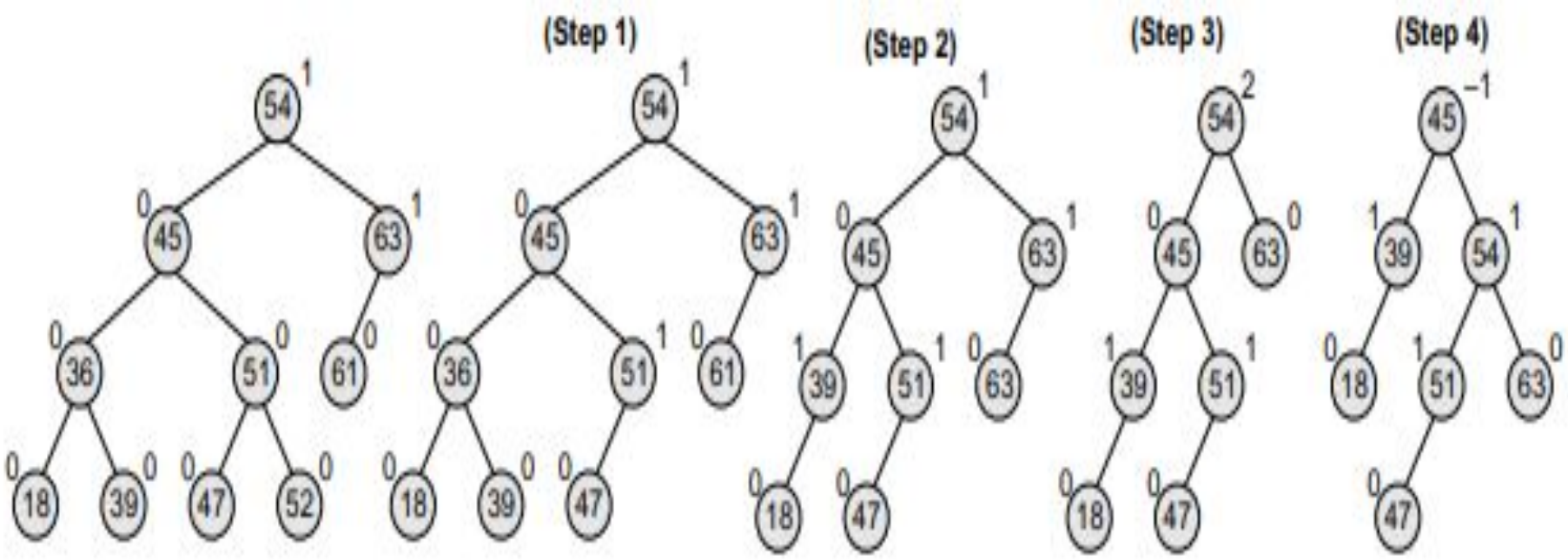
Construct an AVL tree by inserting the following elements in the given order 63, 9, 19, 27, 18, 108, 99, 81.



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

AVL TREE – NODE DELETION - Example

- Delete nodes 52, 36, and 61 from the AVL tree given



Construct AVL Tree

- 35, 15,5,20,25
- ,17,45


SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

B-Trees

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

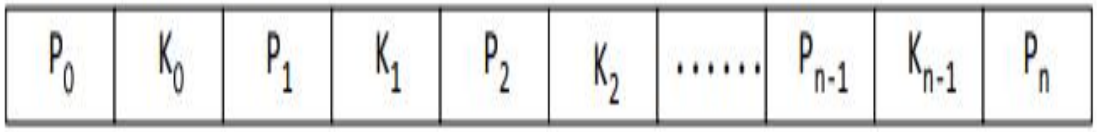
B-Trees

- A B-tree is called as an *m*-way tree where each node is allowed to have a maximum of *m* children.
- Its order is *m.*
- The number of keys in each non-leaf node is *m-1.*
- Leaves should be in the same level and should contain no more than *m-1* keys.
- Non-leaf nodes except the root have at least $\lceil m / 2 \rceil$ children.
- The root can be either leaf node or it can have two to *m* children.

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Structure of an *m*-way search tree node

- The structure of an *m*-way search tree node is shown in figure

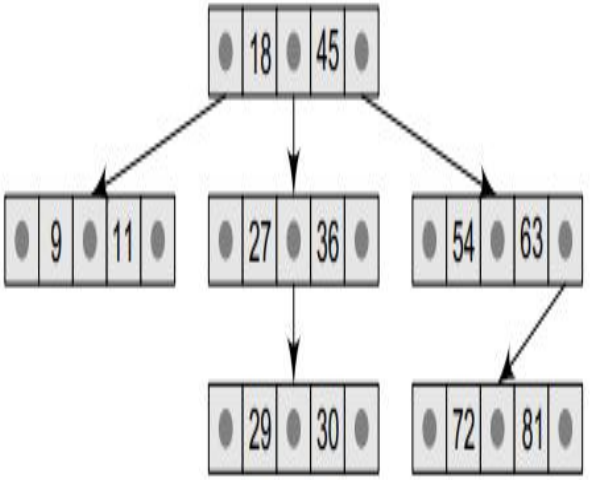


- Where P0 , P1 , P2 , ..., Pn are pointers to the node's sub-trees and K0 , K1 , K2 , ..., Kn–1 are the key values of the node.
- All the key values are stored in ascending order.
- A B tree is a specialized *m*-way tree developed by Rudolf Bayer and Ed McCreight in 1970 that is widely used for disk access.

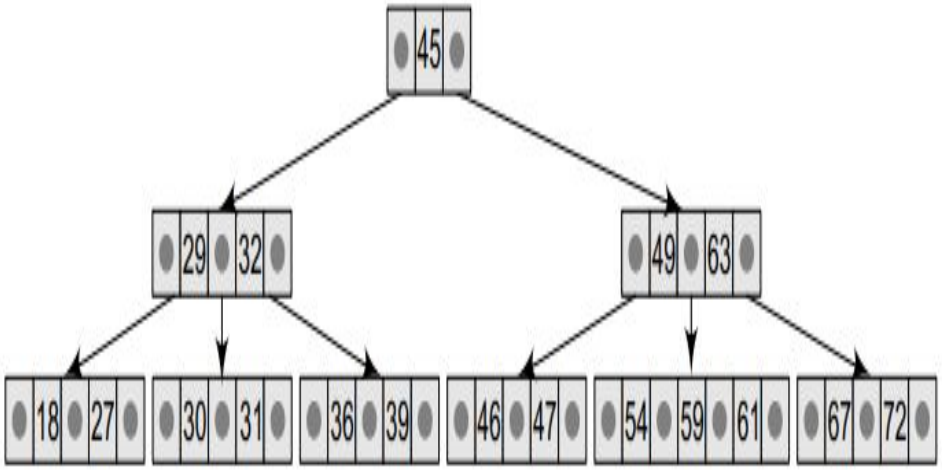
Source :
http://masterraghu.com/subjects/Datastructures/ebooks/rema%20thareja.pdf

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

B-Trees - Examples



B-Tree of order 3

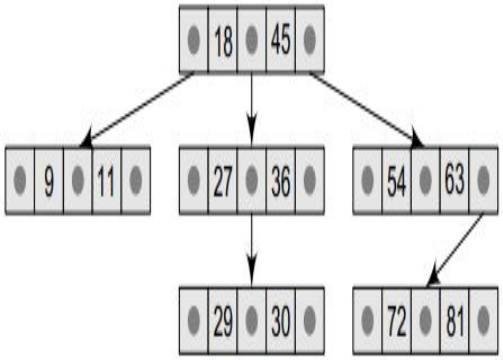


B-Tree of order 4

Source :
http://masterraghu.com/subjects/Datastructures/ebooks/rema%20thareja.pdf

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Searching in a B-Tree



- Similar to searching in a binary search tree.
- Consider the B-Tree shown here.
- If we wish to search for 72 in this tree, first consider

the root, the search key is greater than the values in the root node. So go to the right sub-tree.

- The right sub tree consists of two values and again 72 is greater than 63 so traverse to right sub tree of 63.
- The right sub tree consists of two values 72 and 81. So we found our value 72.

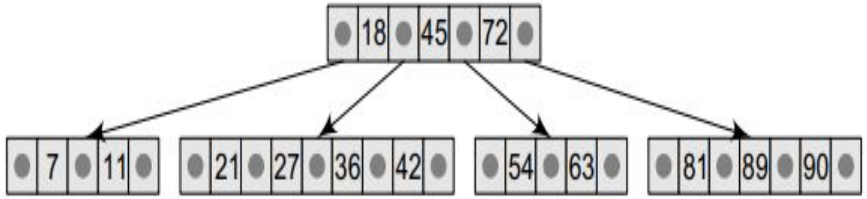
Insertion in a B-Tree

- Insertions are performed at the leaf level.
- Search the B-Tree to find suitable place to insert the new element.
- If the leaf node is not full, the new element can be inserted in the leaf level.
- If the leaf is full,
  - insert the new element in order into the existing set of keys.
  - split the node at its median into two nodes.
  - push the median element up to its parent's node. If the parent's node is already full, then split the parent node by following the same steps.

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

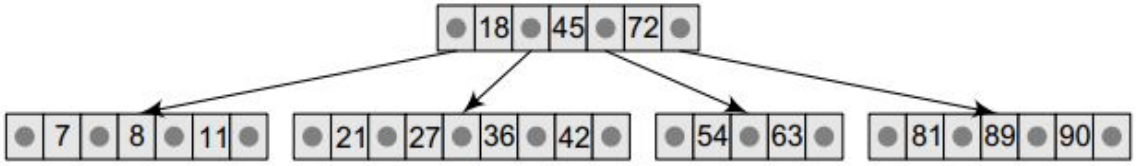
Insertion - Example

- Consider the B-Tree of order 5

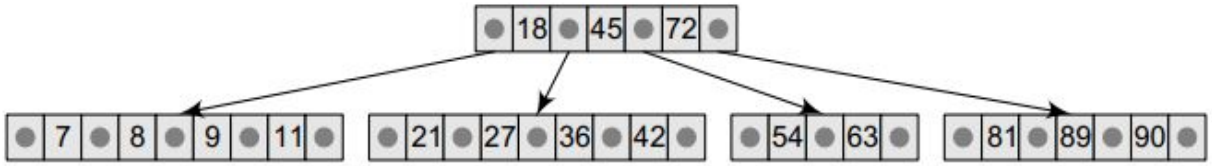


- Try to insert 8, 9, 39 and 4 into it.

Step 1: Insert 8



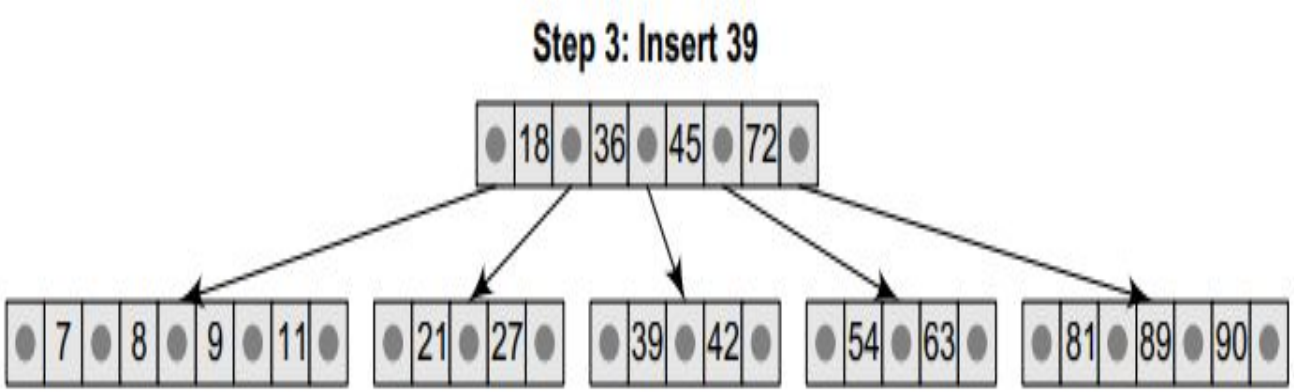
Step 2: Insert 9

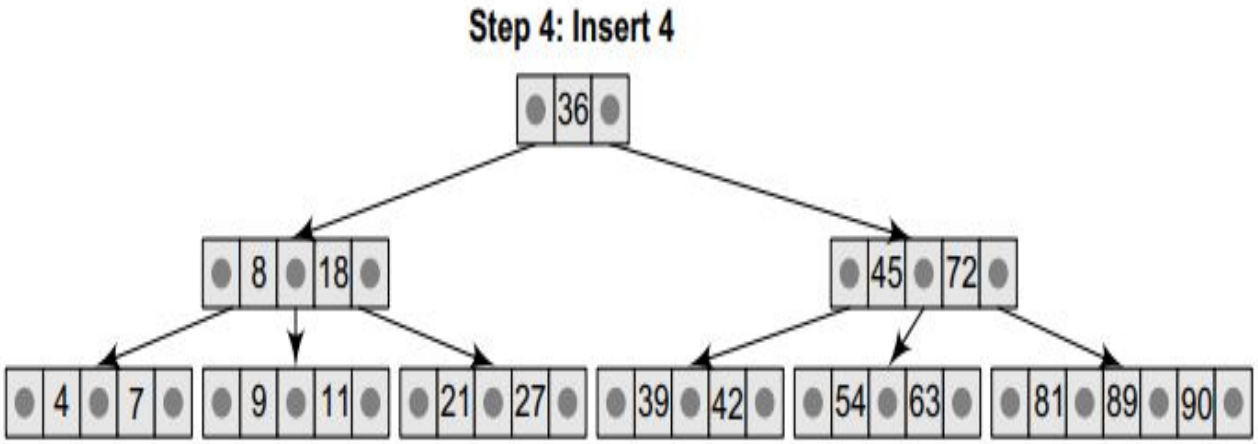


Source :
http://masterraghu.com/subjects/Datastructures/ebooks/rema%20thareja.pdf

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Insertion - Example



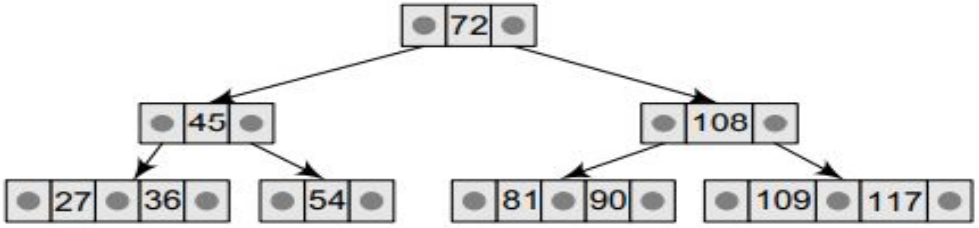


Source :
http://masterraghu.com/subjects/Datastructures/ebooks/rema%20thareja.pdf

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Exercise

- Consider the B-Tree of order 3, try to insert 121 and 87.



- Create a B-Tree of order 5, by inserting the following elements
- 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, and 19.

Deletion in a B-Tree

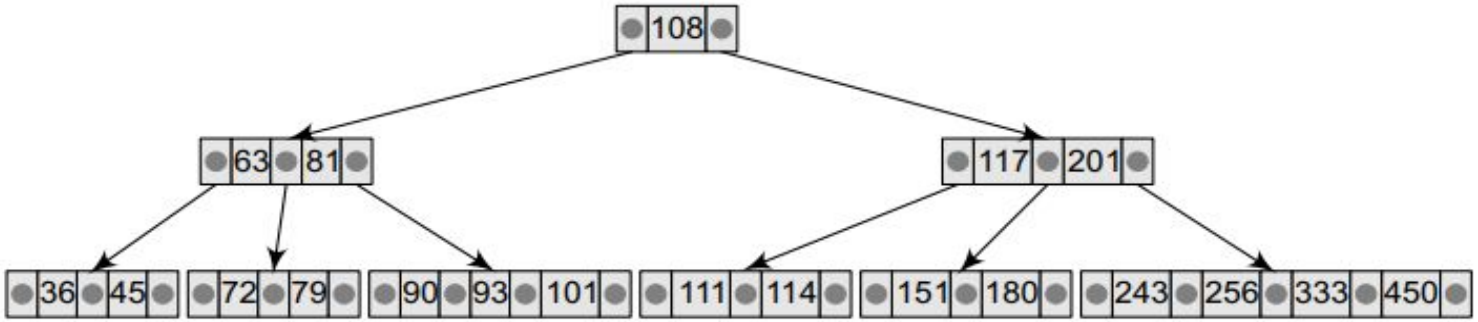
- Like insertion, deletion also should be performed from leaf nodes.
- There are two cases in deletion.
  - To delete a leaf node.
  - To delete an internal node.
- Deleting leaf node
  - Search for the element to be deleted, if it is in the leaf node and the leaf node has more than m/2 elements then delete the element.
  - If the leaf node does not contain m/2 elements then take an element from either left or right sub tree.
  - If both the left and right sub tree contain only minimum number of elements then create a new leaf node.

Deletion in a B-Tree

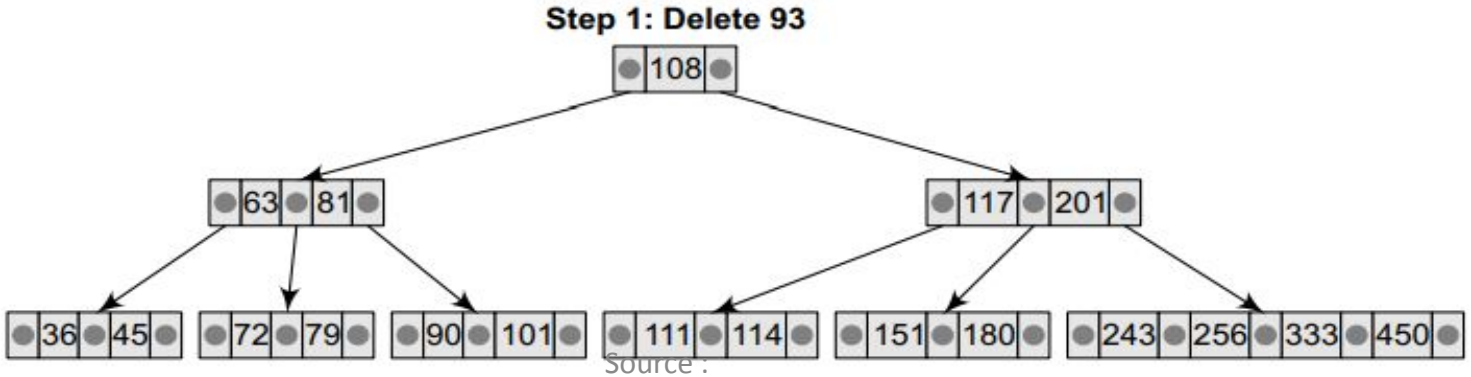
- Deleting an internal node
  - If the element to be deleted is in an internal node then find the predecessor or successor of the element to be deleted and place it in the deleted element position.
  - The predecessor or successor of the element to be deleted will always be in the leaf node.
  - So the procedure will be similar to deleting an element from the leaf node.

Deletion - Example

- Consider the B-Tree of order 5



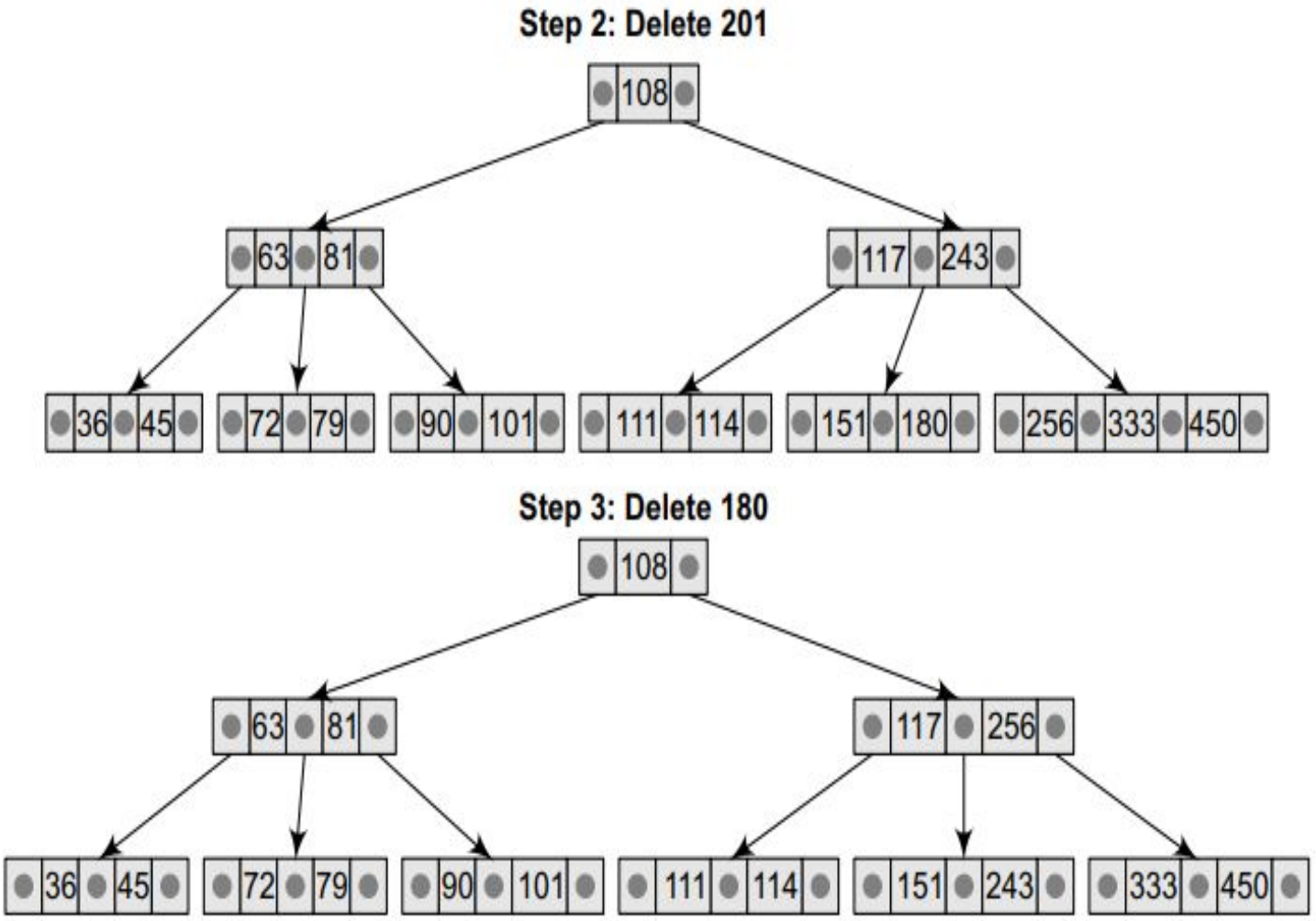
- Try to delete values 93, 201, 180 and 72 from it.



Source :
http://masterraghu.com/subjects/Datastructures/ebooks/rema%20thareja.pdf

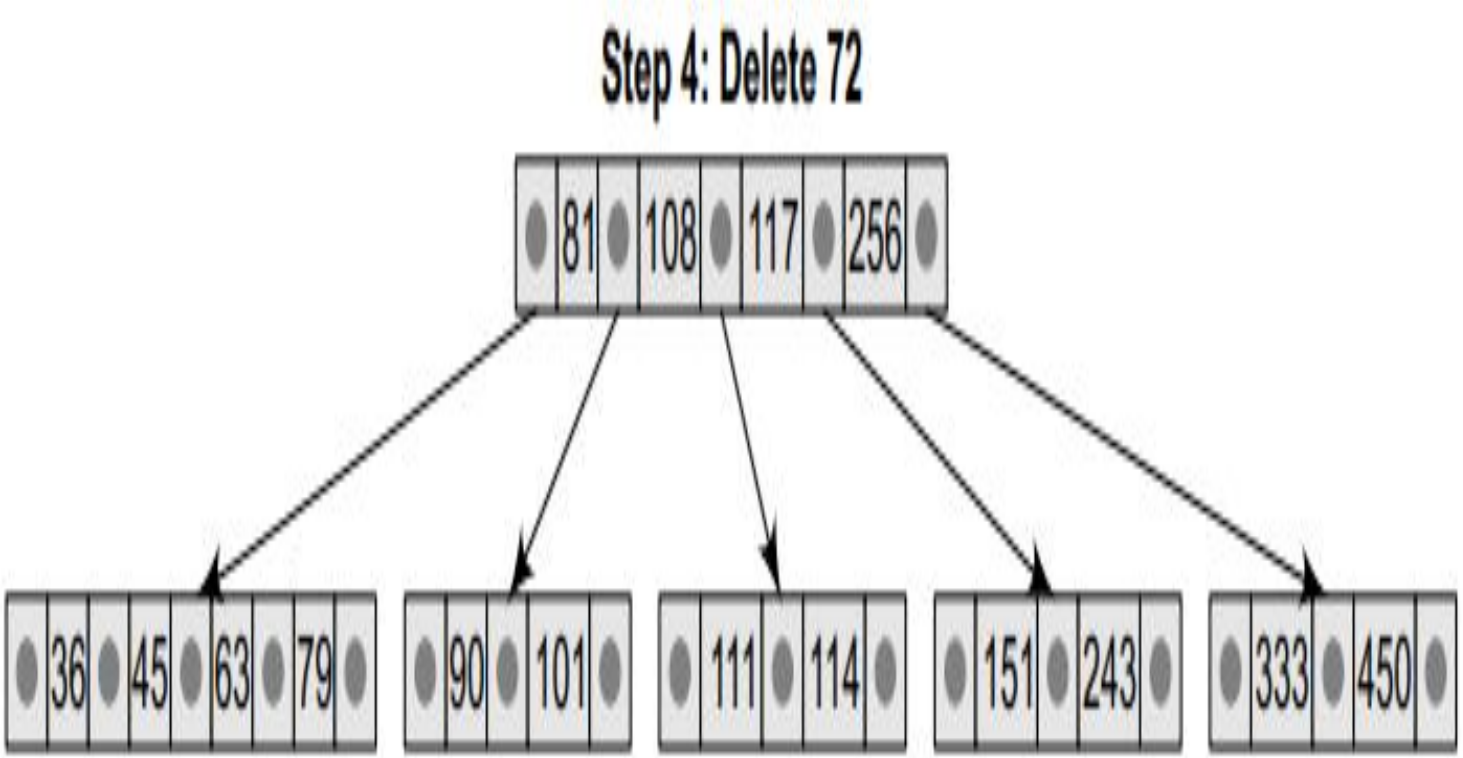
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Deletion - Example



Source :
http://masterraghu.com/subjects/Datastructures/ebooks/rema%20thareja.pdf

Deletion - Example

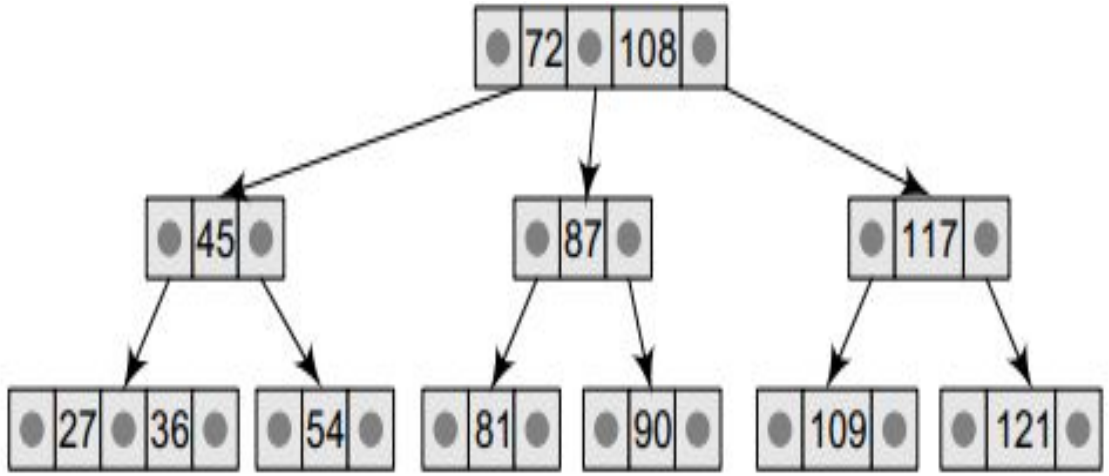


Source :
http://masterraghu.com/subjects/Datastructures/ebooks/rema%20thareja.pdf

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Exercise

- Consider the B-Tree of order 3, try to delete 36 and 109



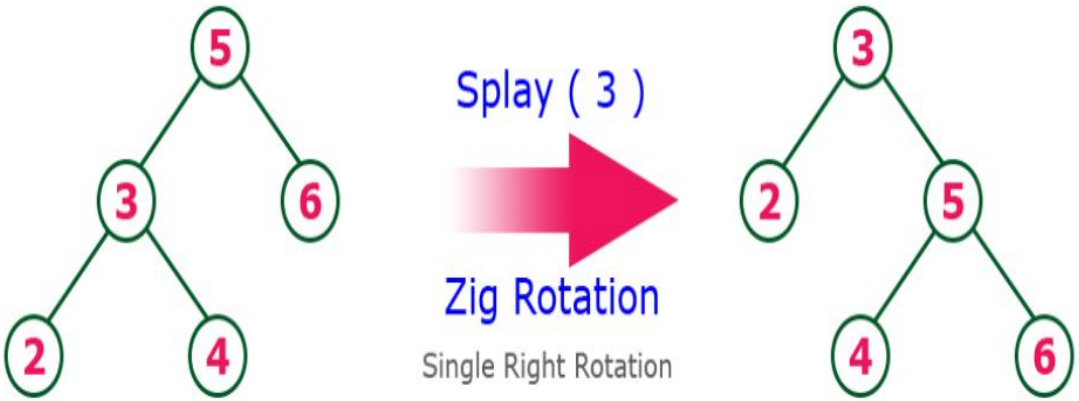
SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Splay Trees

- Self-balancing BST with an additional property that recently accessed elements can be re-accessed fast.
- All operations can be performed in O(log n) time.
- All operations can be performed by combining with the basic operation called splaying.
- Splaying the tree for a particular node rearranges the tree to place that node at the root.
- Each splay step depends on three factors:
- Whether N is the left or right child of its parent P,
- Whether P is the root or not, and if not,
- Whether P is the left or right child of its parent, G (N's grandparent).

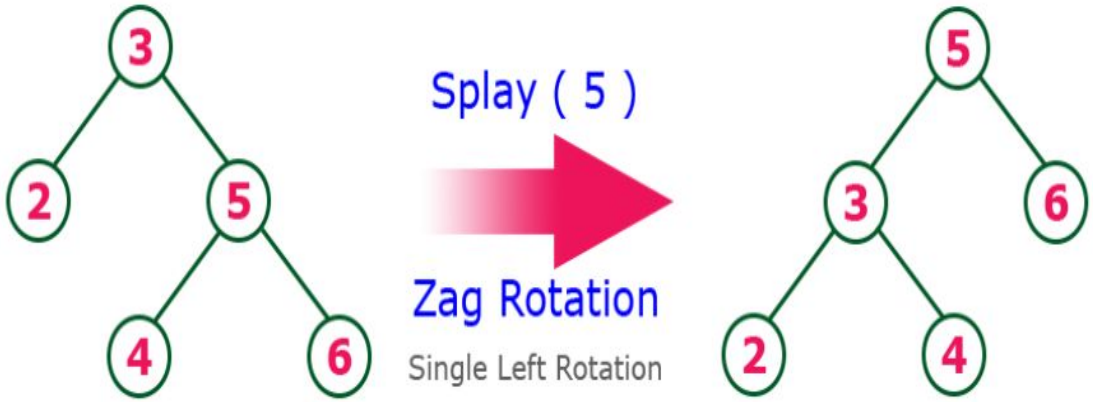
Zig Rotation

The **Zig Rotation** in splay tree is similar to the single right rotation in AVL Tree rotations. In zig rotation, every node moves one position to the right from its current position. Consider the following example...



Zag Rotation

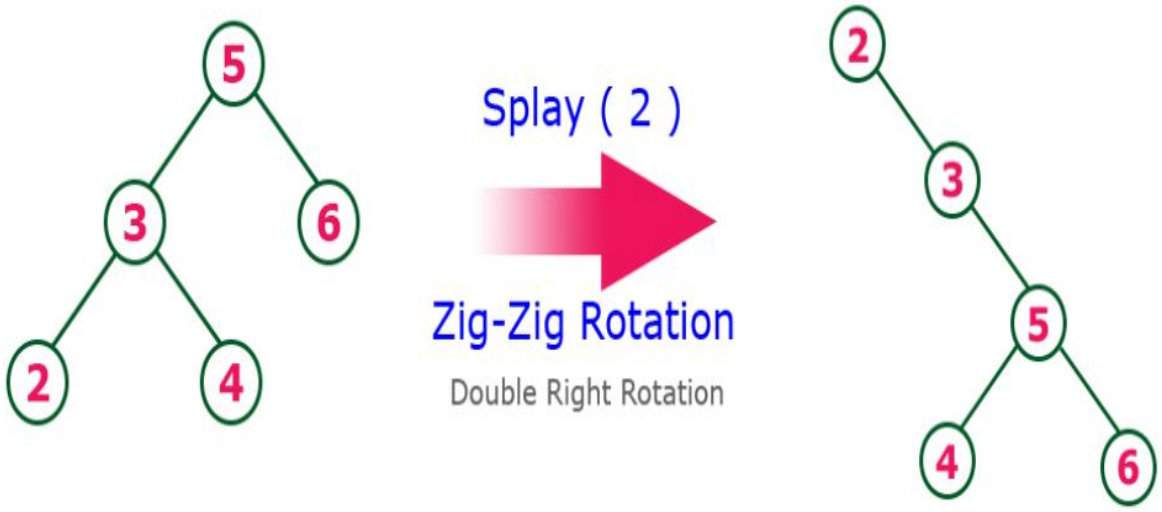
The **Zag Rotation** in splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation, every node moves one position to the left from its current position. Consider the following example...



Zig-Zig Rotation

The **Zig-Zig Rotation** in splay tree is a double zig rotation. In zig-zig rotation, every node moves two positions to the right from its current position.

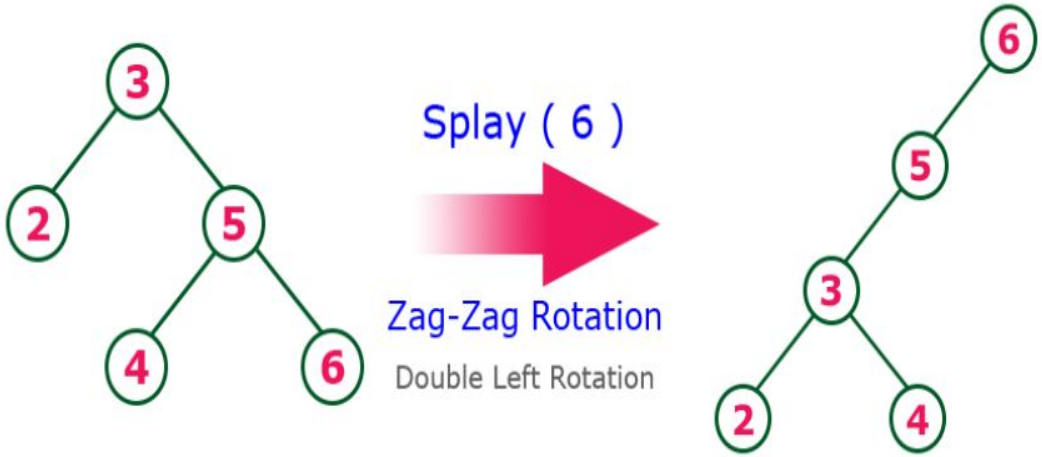
Consider the following example...



Zag-Zag Rotation

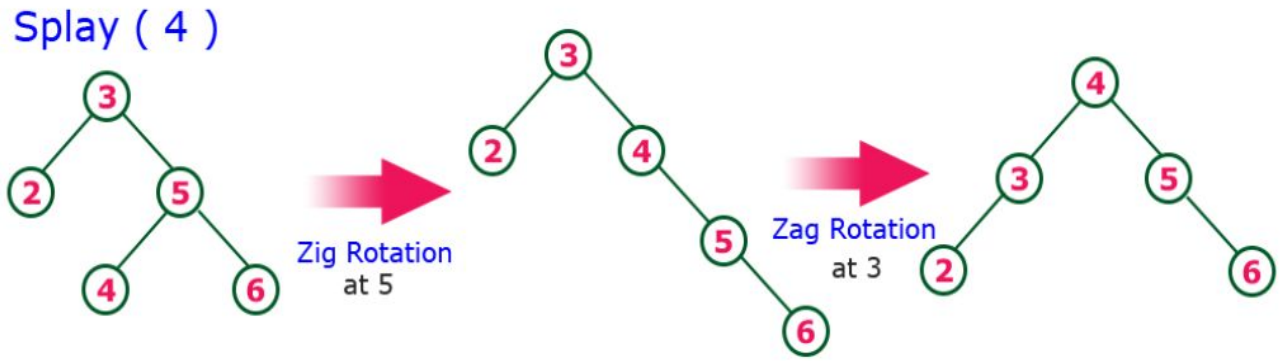
The **Zag-Zag Rotation** in splay tree is a double zag rotation. In zag-zag rotation, every node moves two positions to the left from its current position.

Consider the following example...



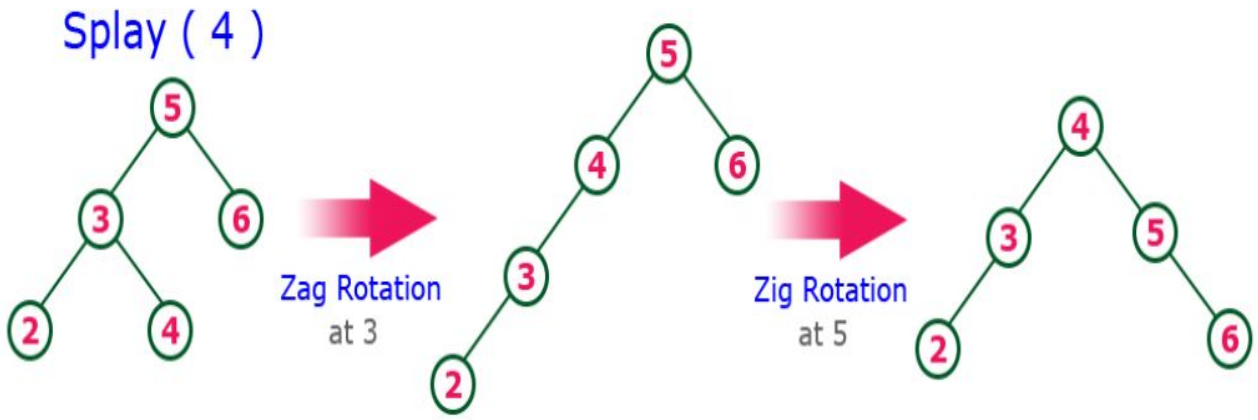
Zig-Zag Rotation

The **Zig-Zag Rotation** in splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position. Consider the following example...



Zag-Zig Rotation

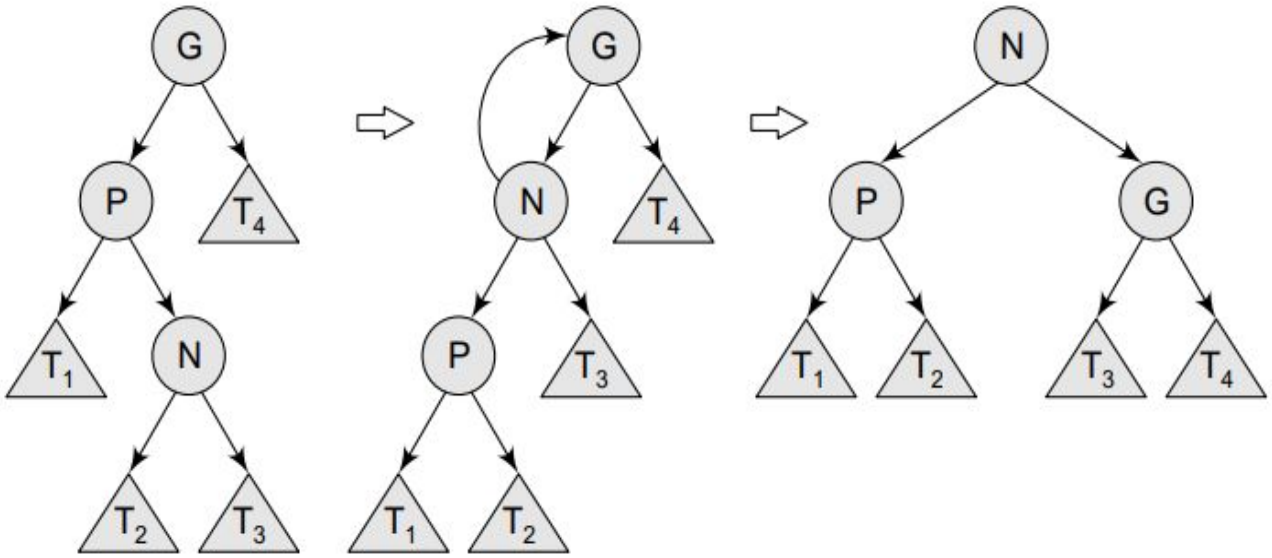
The **Zag-Zig Rotation** in splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position. Consider the following example...



SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Splay Trees

- Zig – Zag Step



- RED -BLACK TREE

Properties of Red Black Tree

- **Property #1:** Red - Black Tree must be a Binary Search Tree.
- **Property #2:** The ROOT node must be colored BLACK.
- **Property #3:** The children of Red colored node must be colored BLACK. (There should not be two consecutive RED nodes).
- **Property #4:** In all the paths of the tree, there should be same number of BLACK colored nodes.
- **Property #5:** Every new node must be inserted with RED color.
- **Property #6:** Every leaf (e.i. NULL node) must be colored BLACK.

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Insertion into RED BLACK Tree

- 1. Recolor
- 2. Rotation
- 3. Rotation followed by Recolor

The insertion operation in Red Black tree is performed using the following steps...

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node with color Red.

Step 4 - If the parent of newNode is Black then exit from the operation.

Step 5 - If the parent of newNode is Red then check the color of parentnode's sibling of newNode.

Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.

Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

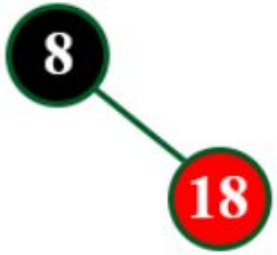
Create a RED BLACK Tree by inserting following sequence of number 8, 18, 5, 15, 17, 25, 40 & 80.

insert (8)

Tree is Empty. So insert newNode as Root node with black color.

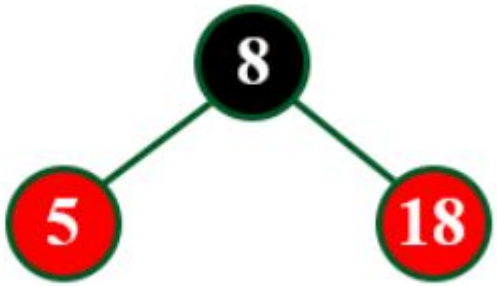
insert (18)

Tree is not Empty. So insert newNode with red color.



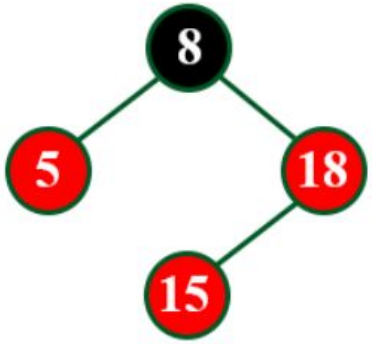
insert (5)

Tree is not Empty. So insert newNode with red color.



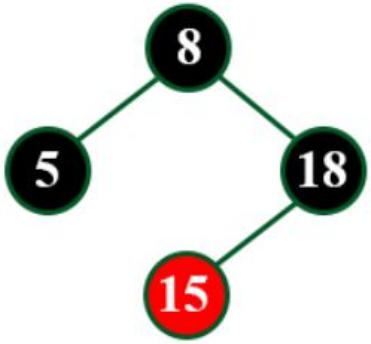
insert (15)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (18 & 15).
The newnode's parent sibling color is Red
and parent's parent is root node.
So we use RECOLOR to make it Red Black Tree.

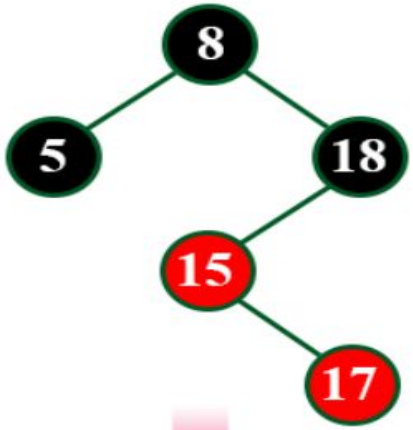
After RECOLOR



After Recolor operation, the tree is satisfying all Red Black Tree properties.

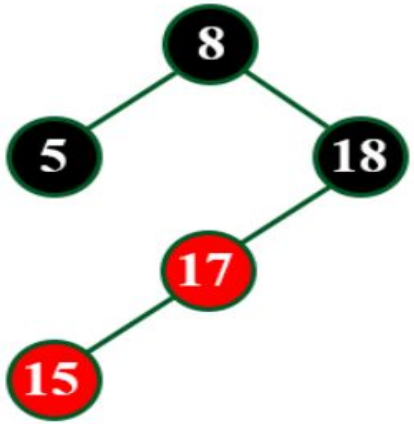
insert (17)

Tree is not Empty. So insert newNode with red color.

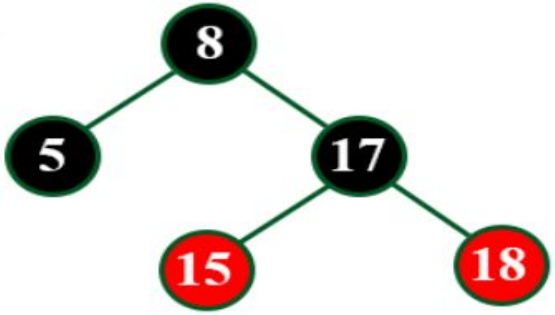


Here there are two consecutive Red nodes (15 & 17). The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

After Left Rotation

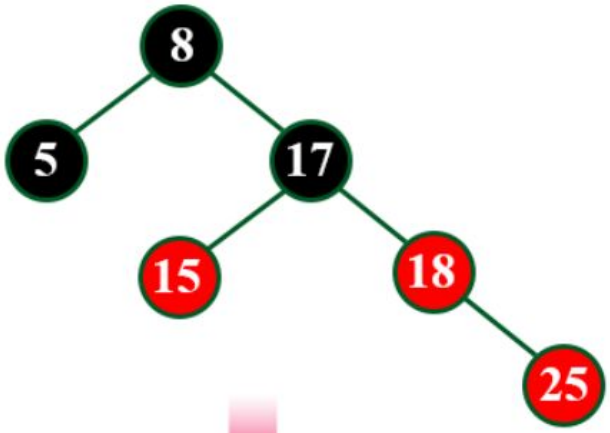


After Right Rotation & Recolor



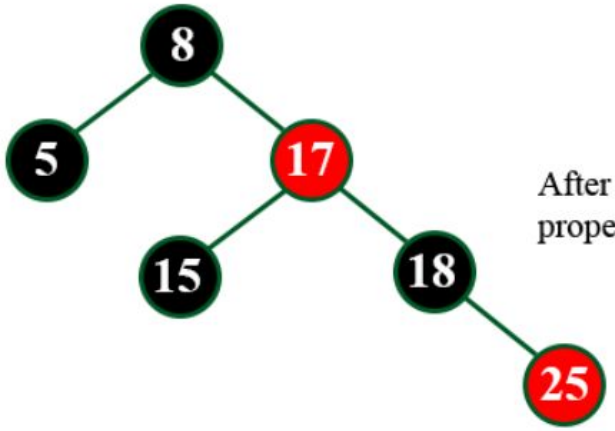
insert (25)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (18 & 25).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

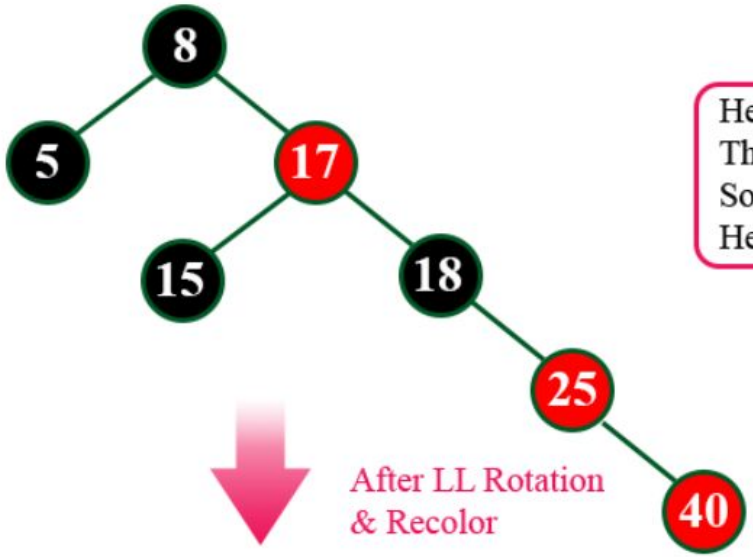
After Recolor



After Recolor operation, the tree is satisfying all Red Black Tree properties.

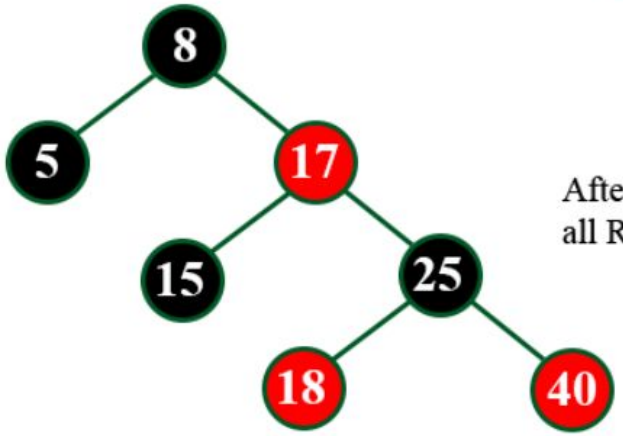
insert (40)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (25 & 40).
The newnode's parent sibling is NULL
So we need a Rotation & Recolor.
Here, we use LL Rotation and Recheck.

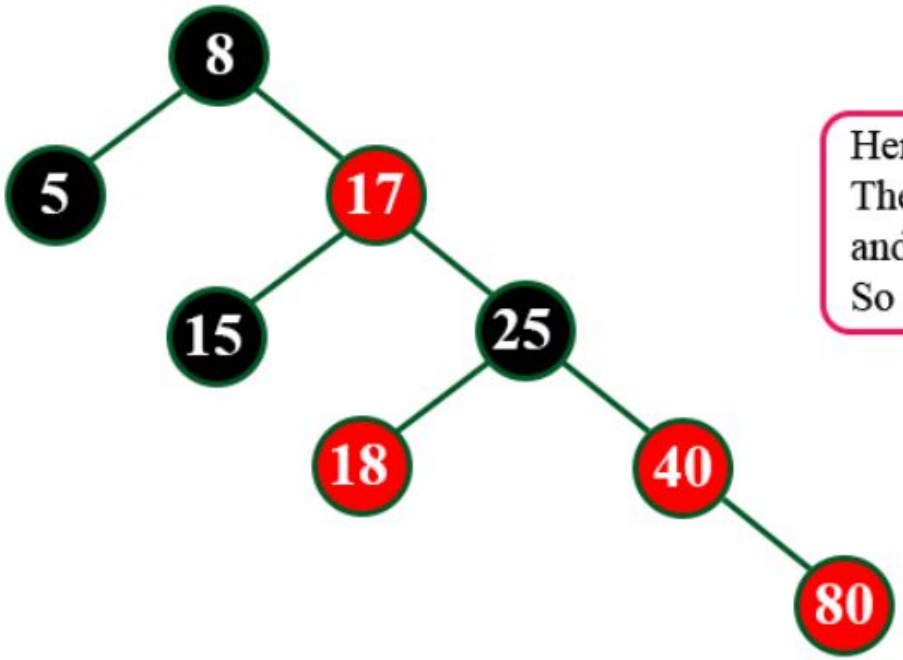
After LL Rotation & Recolor



After LL Rotation & Recolor operation, the tree is satisfying all Red Black Tree properties.

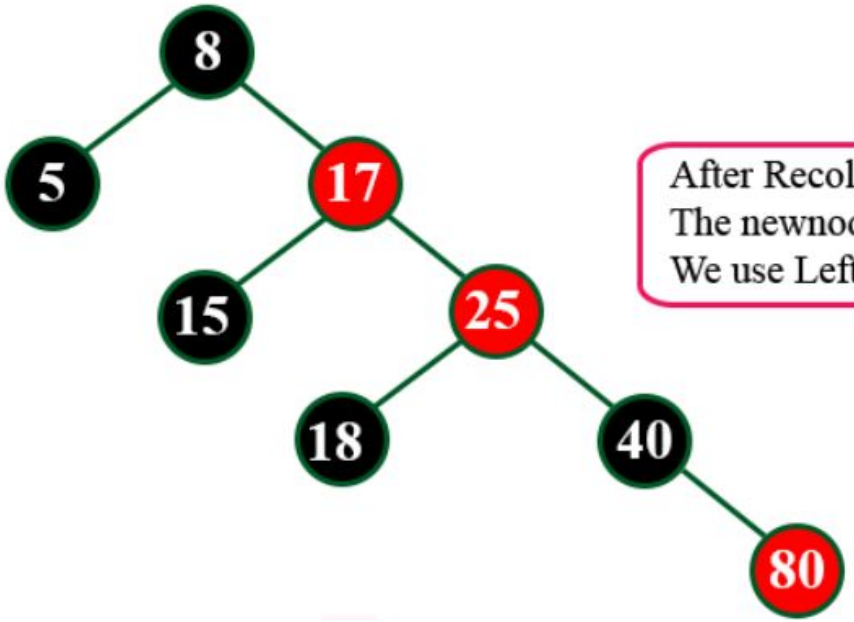
insert (80)

Tree is not Empty. So insert newNode with red color.

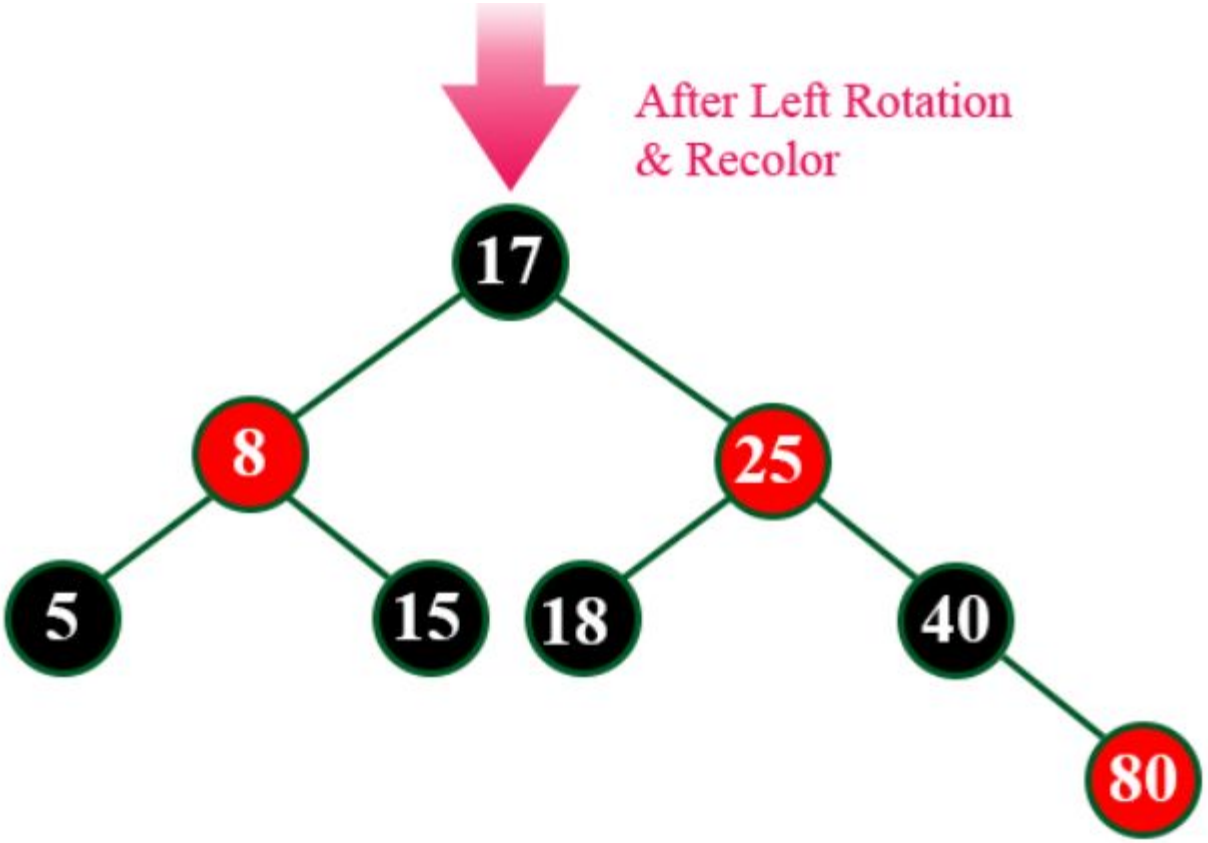


Here there are two consecutive Red nodes (40 & 80)
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor

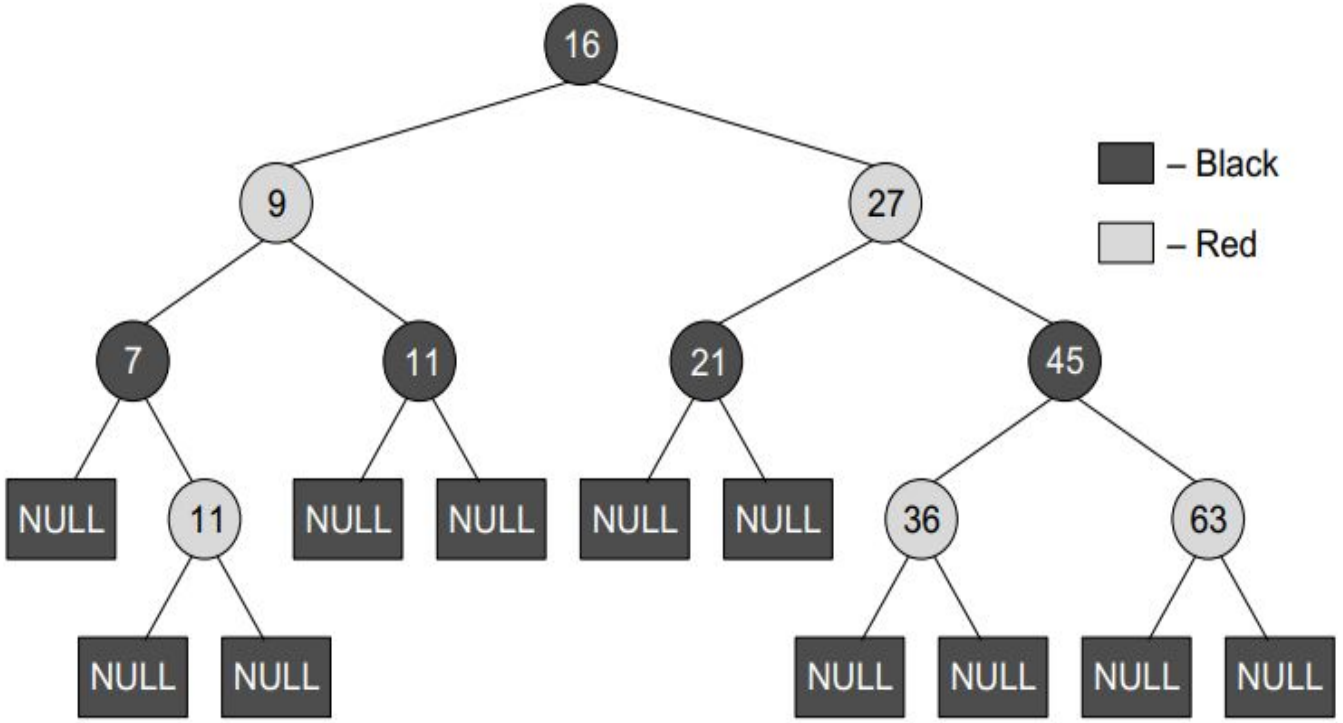


After Recolor again there are two consecutive Red nodes (17 & 25). The newnode's parent sibling color is Black. So we need Rotation. We use Left Rotation & Recolor.


After Left Rotation
& Recolor
17
8
25
5
15
18
40
80

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Red-Black Tree - Example

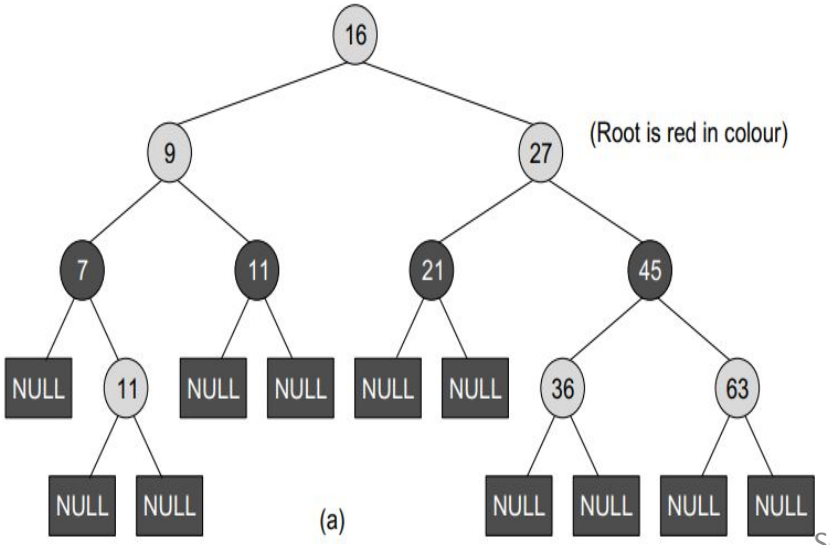


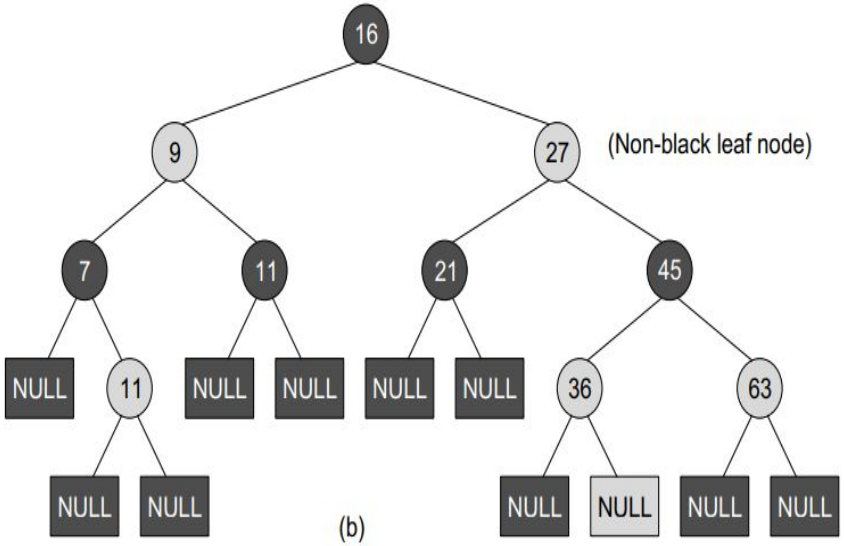
Source :
http://masterraghu.com/subjects/Datastructures/ebooks/rema%20thareja.pdf

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Exercise

- Say whether the following trees are red-black or not.





Source :
http://masterraghu.com/subjects/Datastructures/ebooks/rema%20thareja.pdf

Red Black Trees - Insertion

- Insertion operation starts in the same way as we add new node in the BST.
- The difference is, in BST the new node is added as a leaf node whereas in red-black tree there is no data in the leaf node.
- So we add the new node as a red interior node which has two black child nodes.
- When a new node is added, it may violate some properties of the red-black tree.
- So in order to restore their property, we check for certain cases and restore the property depending on the case that turns up after insertion.
- Terminology
- Grandparent node (G) of node (N) - parent of N’s parent (P)
- Uncle node (U) of node (N) - sibling of N’s parent (P)

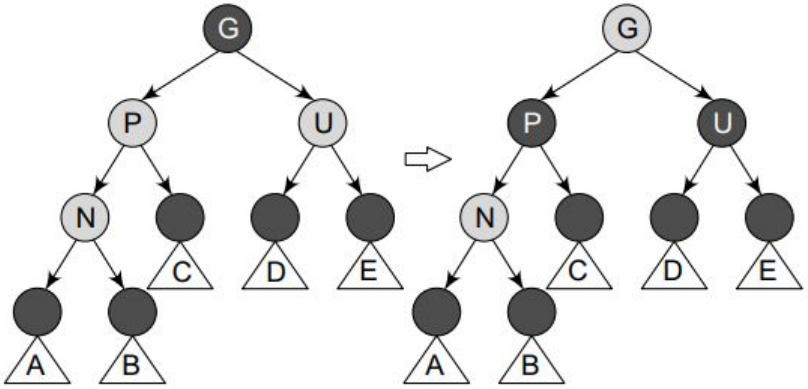
Red Black Trees - Insertion

- When we insert a new node in a red-black tree, note the following:
- All leaf nodes are always black. So property 3 always holds true.
- Property 4 (both children of every red node are black) is threatened only by adding a red node, repainting a black node red, or a rotation.
- Property 5 (all paths from any given node to its leaf nodes has equal number of black nodes) is threatened only by adding a black node, repainting a red node black, or a rotation.
- Case 1: The new node N is added as the root of the Tree
  - In this case, N is repainted black, as the root should be black always.
- Case 2: The new node's parent P is black
  - In this case, both children of every red node are black, so Property 4 is not invalidated. Property 5 is also not threatened.

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Red Black Trees - Insertion

- Case 3: If Both the Parent (P) and the Uncle (U) are Red
- In this case, Property 5 which says all paths from any given node to its leaf nodes have an equal number of black nodes is violated.

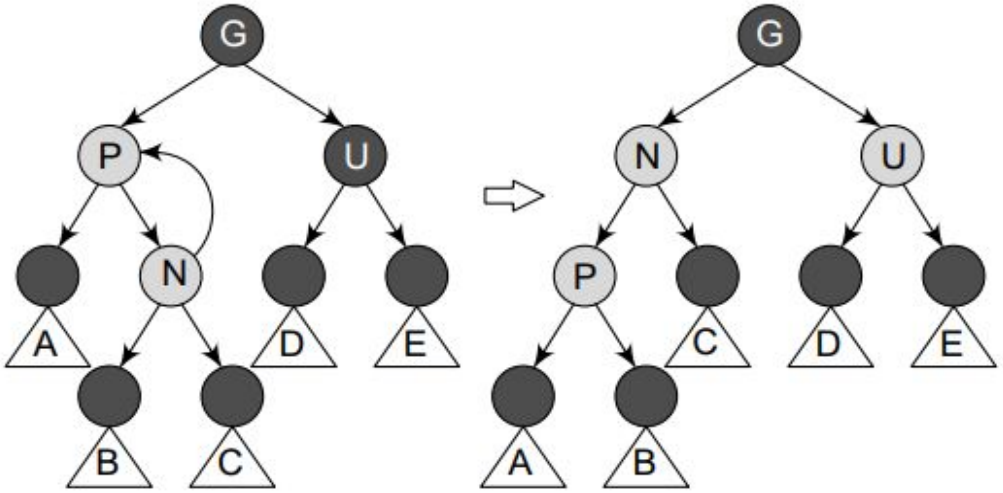


Source :
http://masterraghu.com/subjects/Datastructures/ebooks/rema%20thareja.pdf

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Red Black Trees - Insertion

- Case 4: The Parent P is Red but the Uncle U is Black and N is the Right Child of P and P is the Left Child of G

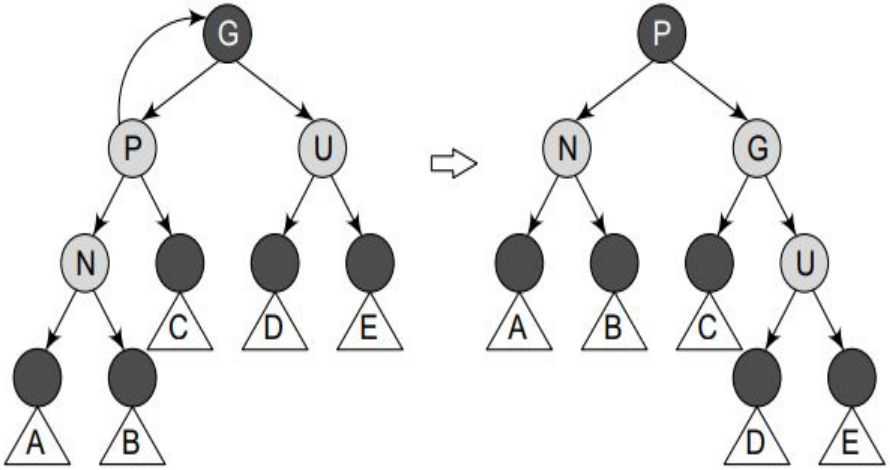


Source :
http://masterraghu.com/subjects/Datastructures/ebooks/rema%20thareja.pdf

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Red Black Trees - Insertion

- Case 5: The parent P is red but the uncle U is black and the new node N is the left child of P, and P is the left child of its parent G.



Source :
http://masterraghu.com/subjects/Datastructures/ebooks/rema%20thareja.pdf

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

References

1. Reema Thareja, Data Structures Using C, 1st ed., Oxford Higher Education, 2011

2. Thomas H Cormen, Charles E Leiserson, Ronald L Revest, Clifford Stein, Introduction to Algorithms 3rd ed., The MIT Press Cambridge, 2014

3. Mark Allen Weiss, Data Structures and Algorithm Analysis in C, 2nd ed., Pearson Education, 2015

4.http://masterraghu.com/subjects/Datastructures/ebooks/rema%20thareja.pdf