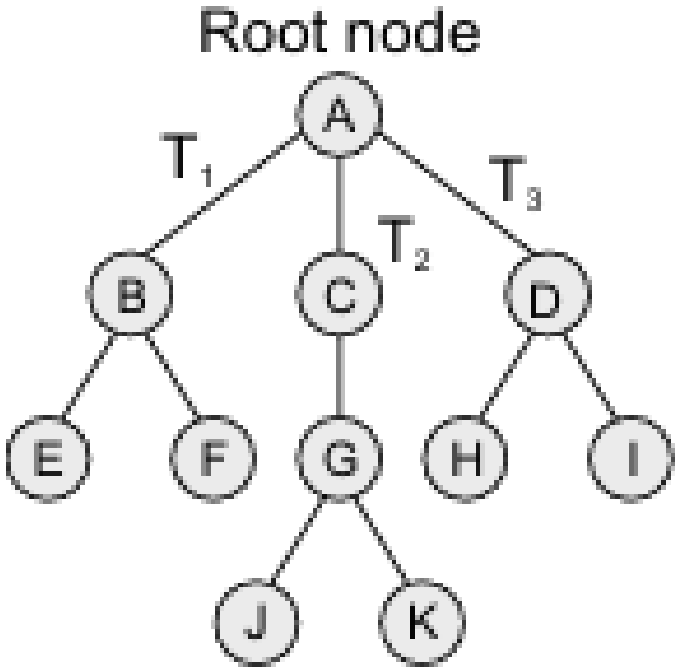
Unit 5

Trees

Contents

- Definition of Tree
- Binary Tree and their types
- Representation of Binary Tree
- Operations on Binary Tree
- Binary Search Tree (BST)
- Traversal of Binary Tree
  - Pre-order Traversal
  - In-order Traversal
  - Post-order Traversal
- Introduction of Threaded Binary Tree
- AVL Tree and B-Tree

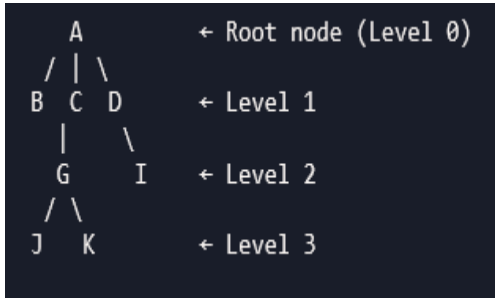
Introduction



- A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree and all the remaining nodes can be partitioned into non-empty sets each of which is a sub-tree of the root.
- Figure shows a tree where node A is the root node; nodes B, C, and D are children of the root node and form sub-trees of the tree rooted at node A.

Basic Terminology

- **Root Node**: Topmost node in the tree
  - Node **A** is the root. It has no parent and is at level 0.
- **Sub-trees**: If the root node R is not NULL, then the trees B , C , and D are called the sub-trees of R.
- **Leaf Node**: A node that has no children is called the leaf node or the terminal node
  - Nodes **B**, **J**, **K**, and **I** have no children, so they are leaf nodes.
- **Path**: A sequence of consecutive edges is called a path
  - A path from **A** to **I** is: A → D → I.



- **Ancestor Node**: An ancestor of a node is any predecessor node on the path from root to that node. The root node does not have any ancestors.
  - For node **K**, ancestors are: A → C → G.
- **Descendant Node**: A descendant node is any successor node on any path from the node to a leaf node. Leaf nodes do not have any descendants.
  - For node **A**, descendants include: B, C, D, G, I, J, K.
- **Level Number**: Every node in the tree is assigned a level number in such a way that the root node is at level 0, children of the root node are at level number 1. Thus, every node is at one level higher than its parent. So, all child nodes have a level number given by parent's level number + 1.
  - A: Level 0
  - B, C, D: Level 1
  - G, I: Level 2
  - J, K: Level 3

- **Degree**: Degree of a node is equal to the number of children that a node has. The degree of a leaf node is zero
  - A has degree 3 (children: B, C, D)
  - G has degree 2 (children: J, K)
  - Leaf nodes like B, J, K, I have degree 0
- **In-degree**: In-degree of a node is the number of edges arriving at that node
  - Number of edges arriving at a node.
  - Example: G has in-degree 1 (from C)
- **Out-degree**: Out-degree of a node is the number of edges leaving that node
- Number of edges leaving a node.
- Example: C has out-degree 1 (to G)

Types of Trees

Trees are of following 6 types:

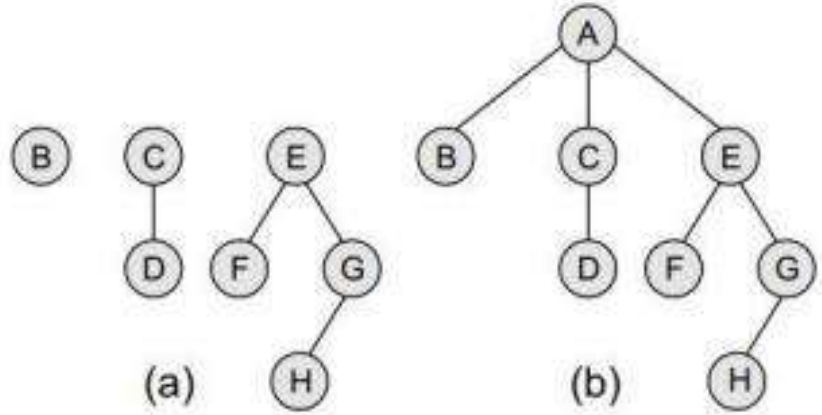
1. General trees
2. Forests
3. Binary trees
4. Binary search trees
5. Expression trees
6. Tournament trees

1.General Trees

- General trees are data structures that store elements hierarchically.
- The top node of a tree is the root node and each node, except the root, has a parent.
- A node in a general tree(except the leaf nodes)may have zero or more sub-trees.
- General trees which have3 sub-trees per node are called ternary trees. However, the number of sub-trees for any node may be variable. For example, a node can have 1 sub-tree, whereas some other node can have 3 sub-trees.

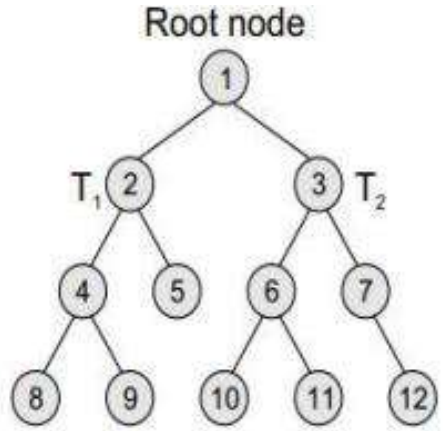
2.Forests

- Forest is a disjoint union of trees.
- A set of disjoint trees(or forests) is obtained by deleting the root and the edges connecting the root node to nodes at level1. Every node of a tree is the root of some sub-tree. Therefore, all the sub-trees immediately below a node forma forest.
- We can convert a forest into a tree by adding a single node as the root node of the tree. For example, below Fig (a)shows a forest and Fig.(b) shows the corresponding tree. Similarly, we can convert a general tree into a forest by deleting the root node of the tree.

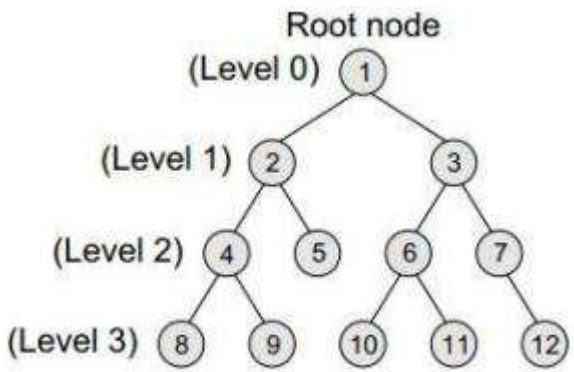


3.Binary tree

- A binary tree is a data structure that is defined as a collection of elements called nodes. In a binary tree, the topmost element is called the root node, and each node has 0, 1, or at the most 2 children.
- Every node contains a data element, a left pointer which points to the left child, and a right pointer which points to the right child. The root element is pointed by a 'root' pointer.
- If root = NULL, then it means the tree is empty.
- R is the root node and the two trees T1 and T2 are called the left and right sub-trees ofR.T1 is said to be the left successor ofR.Likewise,T2 is called the right successor of R.

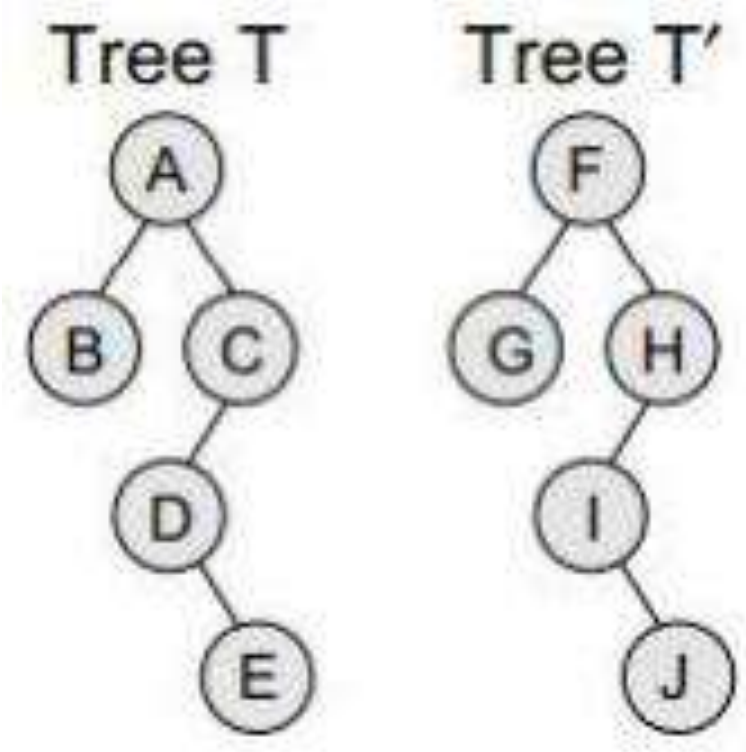


- **Parent :** If N is any node in T that has left successor S1 and right successor S2 , then N is called the parent of S1 and S2.Correspondingly, S1 and S2 are called the left child and the right child of N. Every node other than the root node has a parent.
- **Level number :**Every node in the binary tree is assigned a level number The root node is defined to be at level 0.
- The left and the right child of the root node have a level number 1.
- Similarly, every node is at one level higher than its parents. So all child nodes are defined to have level number as parent's level number + 1.
- **Degree of a node :** It is equal to the number of children that a node has. The degree of a leaf node is zero.
- For example, in the tree, degree of node 4 is 2, degree of node 5 is zero and degree of node 7 is 1.
- **Sibling:** All nodes that are at the same level and share the same parent are called siblings (brothers). For example, nodes 2 and 3; nodes 4 and 5; nodes 6 and 7; nodes 8 and 9; and nodes 10 and 11 are siblings.
- **Path :**A sequence of consecutive edges .the path from the root node to the node 8 is given as: 1, 2, 4, and 8.
- **Leaf node :** A node that has no children is called a leaf node or a terminal node. The leaf nodes in the tree are: 8, 9, 5, 10, 11, and 12.



Similar binary trees

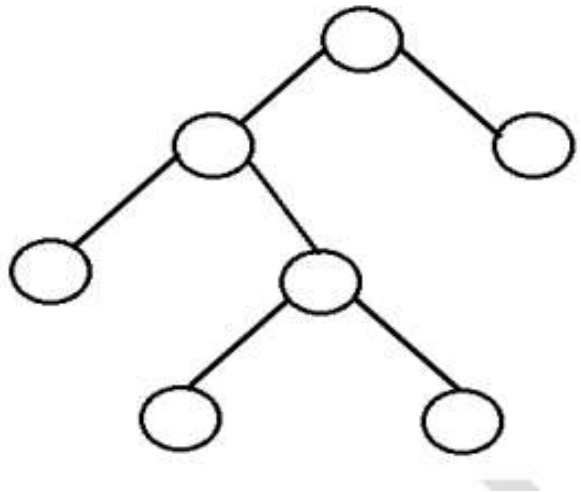
- Two binary trees T and T” are said to be similar if both these trees have the same structure. Figure shows two similar binary trees.



- Edge : It is the line connecting a node N to any of its successors. A binary tree of n nodes has exactly n-1 edges because every node except the root node is connected to its parent via an edge.
- Depth :The depth of a node N is given as the length of the path from the root R to the node N. The depth of the root node is zero.
- Height of a tree :It is the total number of nodes on the path from the root node to the deepest node in the tree. A tree with only a root node has a height of 1

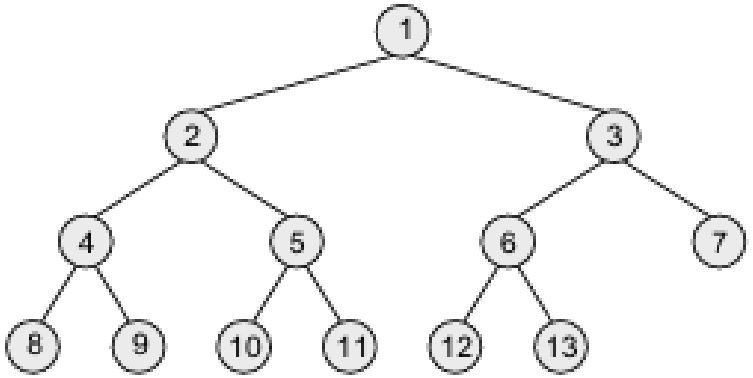
Strictly binary tree

- It is a tree in which every node in the tree has either 0 or 2 children.



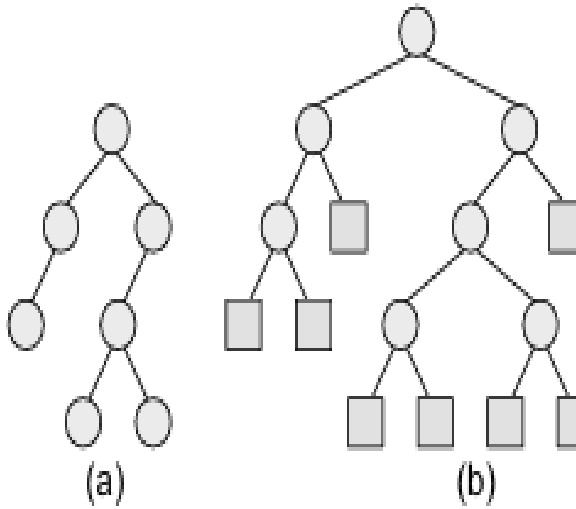
COMPLETE BINARY TREES

- A complete binary tree is a binary tree that satisfies two properties. First, in a complete binary tree, every level, except possibly the last, is completely filled.
- Second, all nodes appear as far left as possible.
- In a complete binary tree Tn, there are exactly n nodes and level r of T can have at most 2r nodes.



- level 0 has 2^0 = 1 node, level 1 has 2^1 = 2 nodes, level 2 has 2^2 = 4 nodes, level 3 has 6 nodes which is less than the maximum of 2^3 = 8 nodes.
- Tree has exactly 13 nodes. They have been purposely labelled from 1 to 13, so that it is easy for the reader to find the parent node, the right child node, and the left child node of the given node. The formula can be given as—if K is a parent node, then its left child can be calculated as 2 × K and its right child can be calculated as 2 × K + 1.
- For example, the children of the node 4 are 8 (2 × 4) and 9 (2 × 4 + 1). Similarly, the parent of the node K can be calculated as | K/2 |.
- Given the node 4, its parent can be calculated as | 4/2 | = 2. The height of a tree Tn having exactly n nodes is given as:
- Hn= | log2(n + 1) |
- This means, if a tree T has 10,00,000 nodes, then its height is 21.

5.Extended Binary Trees



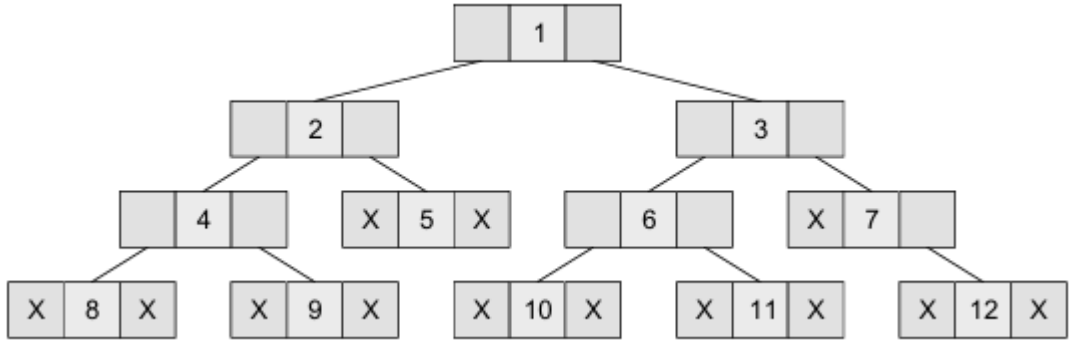
- A binary tree T is said to be an extended binary tree (or a 2-tree) if each node in the tree has either no child or exactly two children.
- how an ordinary binary tree is converted into an extended binary tree?
- In an extended binary tree, nodes having two children are called internal nodes and nodes having no children are called external nodes.
- The internal nodes are represented using circles and the external nodes are represented using squares. To convert a binary tree into an extended tree, every empty sub-tree is replaced by a new node. The original nodes in the tree are the internal nodes, and the new nodes added are called the external nodes.

Representation of Binary Trees in the Memory

- Linked representation of binary trees In the linked representation of a binary tree, every node will have three parts: the data element, a pointer to the left node, and a pointer to the right node.
- In C, the binary tree is built with a node type given below.

```
struct node {
struct node *left;
int data;
struct node *right;
};
```

Link Representation

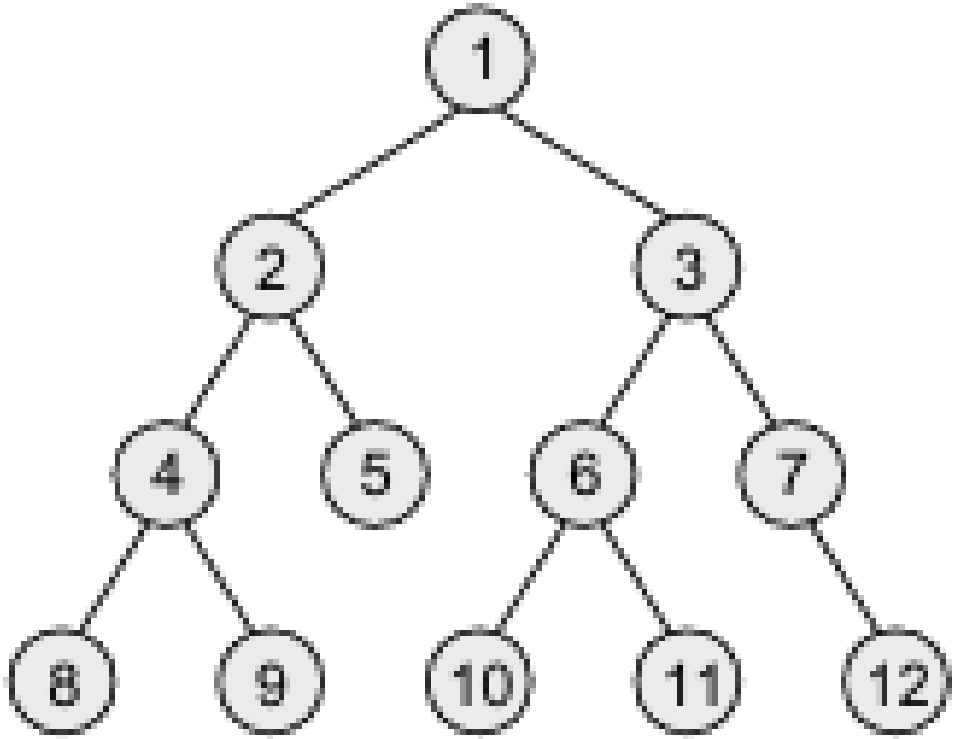


Every binary tree has a pointer ROOT, which points to the root element (topmost element) of the tree.
If ROOT = NULL, then the tree is empty.
In the left position is used to point to the left child of the node or to store the address of the left child of the node.
The middle position is used to store the data.
Finally, the right position is used to point to the right child of the node or to store the address of the right child of the node.
Empty sub-trees are represented using X (meaning NULL).

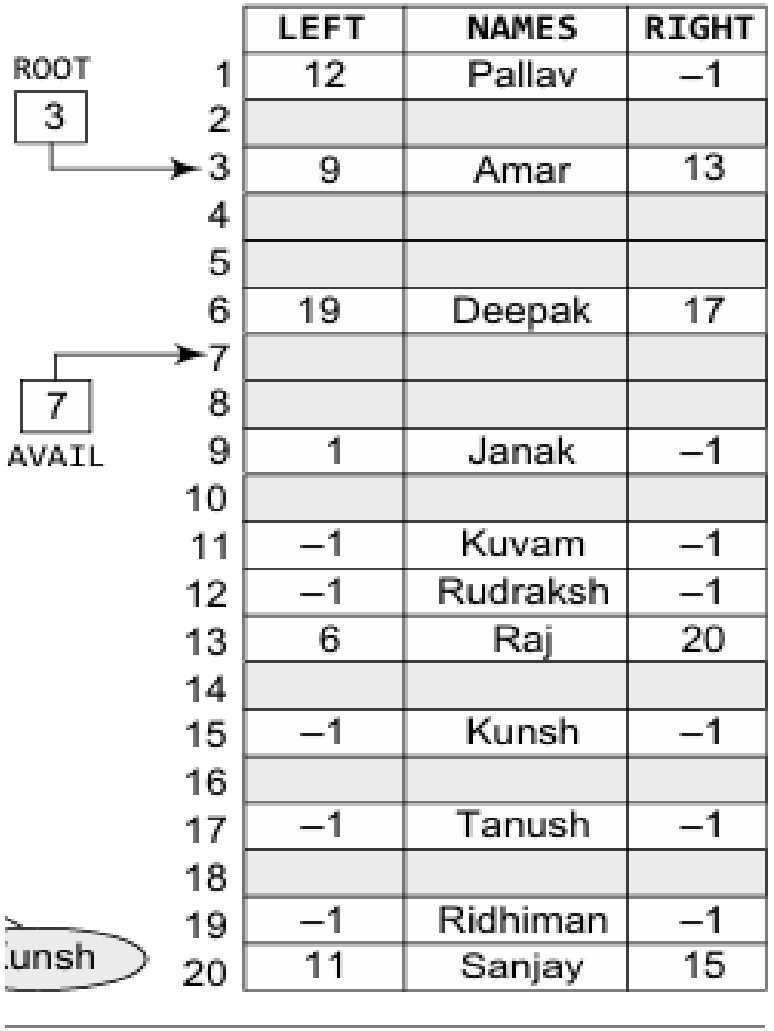
ROOT
3

15
AVAIL

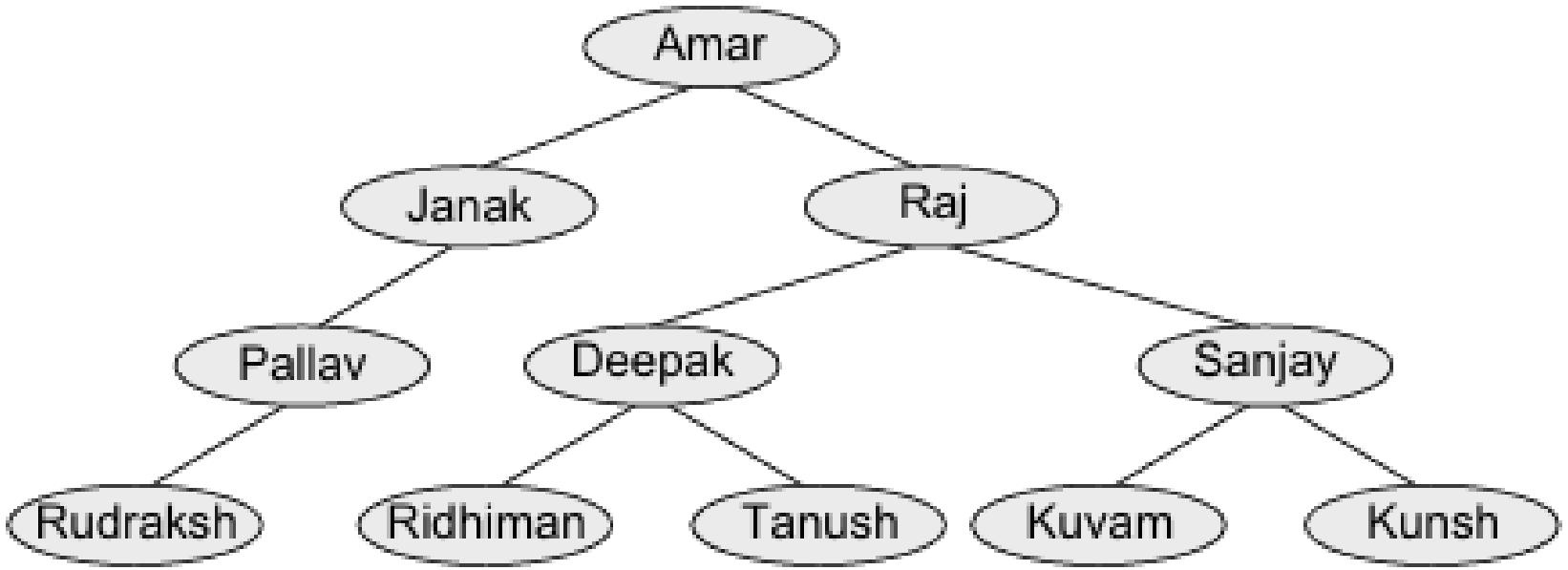
	LEFT	DATA	RIGHT
1	–1	8	–1
2	–1	10	–1
3	5	1	8
4			
5	9	2	14
6			
7			
8	20	3	11
9	1	4	12
10			
11	–1	7	18
12	–1	9	–1
13			
14	–1	5	–1
15			
16	–1	11	–1
17			
18	–1	12	–1
19			
20	2	6	16



Given the memory representation of a tree that stores the names of family members, construct the corresponding tree from the given data.

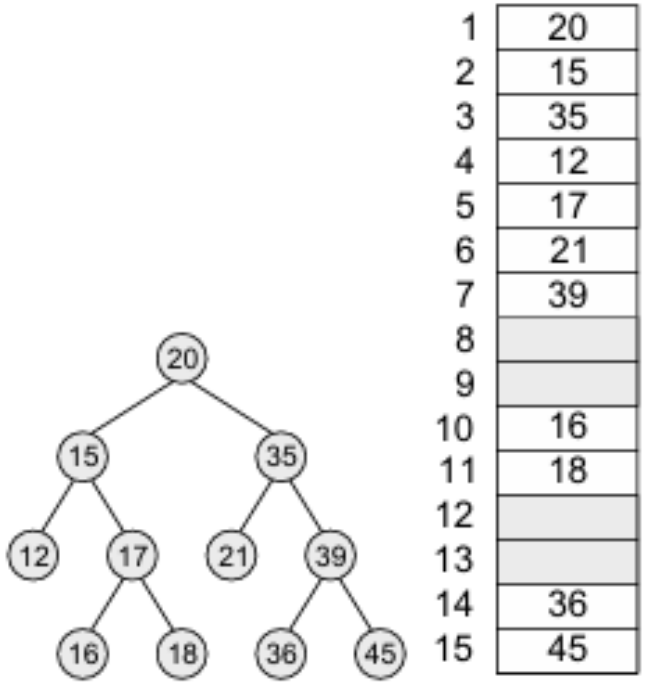


	LEFT	NAMES	RIGHT
1	12	Pallav	–1
2			
3	9	Amar	13
4			
5			
6	19	Deepak	17
7			
8			
9	1	Janak	–1
10			
11	–1	Kuvam	–1
12	–1	Rudraksh	–1
13	6	Raj	20
14			
15	–1	Kunsh	–1
16			
17	–1	Tanush	–1
18			
19	–1	Ridhiman	–1
20	11	Sanjay	15


Amar
Janak
Raj
Pallav
Deepak
Sanjay
Rudraksh
Ridhiman
Tanush
Kuvam
Kunsh

Sequential representation of binary trees

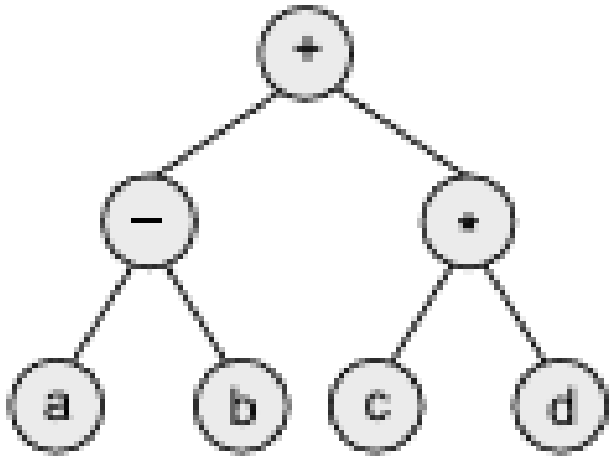
- Sequential representation of trees is done using single or one-dimensional arrays.
- Though it is the simplest technique for memory representation, it is inefficient as it requires a lot of memory space.
- A sequential binary tree follows the following rules:
- ∑ A one-dimensional array, called TREE, is used to store the elements of tree.
- ∑ The root of the tree will be stored in the first location. That is, TREE[1] will store the data of the root element.
- ∑ The children of a node stored in location K will be stored in locations (2 × K) and (2 × K+1).
- ∑ The maximum size of the array TREE is given as (2h–1), where h is the height of the tree.
- ∑ An empty tree or sub-tree is specified using NULL. If TREE[1] = NULL, then the tree is empty.
- Figure shows a binary tree and its corresponding sequential representation. The tree has 11 nodes and its height is 4.

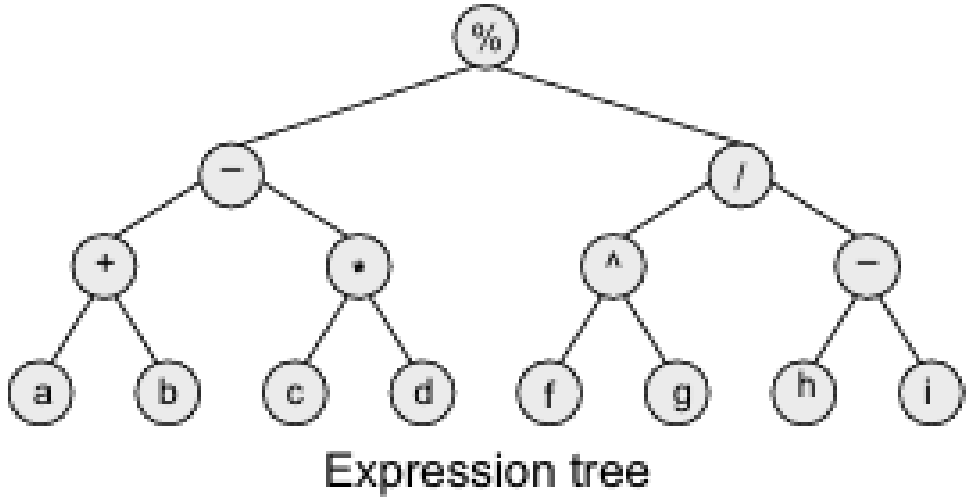


Expression Trees

Exp = ((a + b) – (c * d)) % ((e ^f) / (g – h))

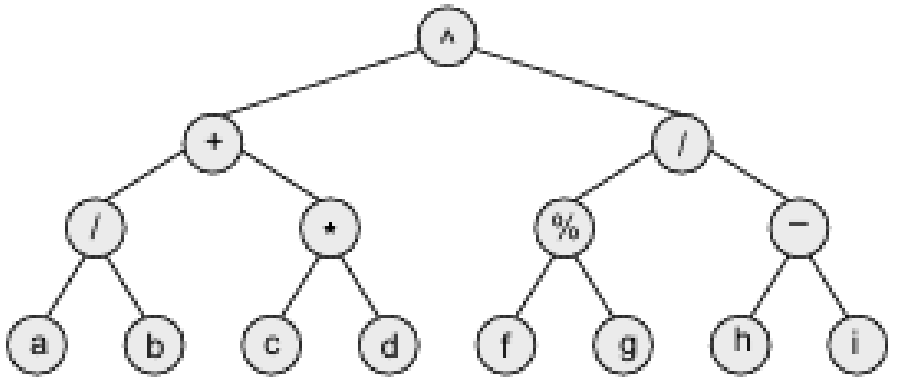
Exp = (a – b) + (c * d)





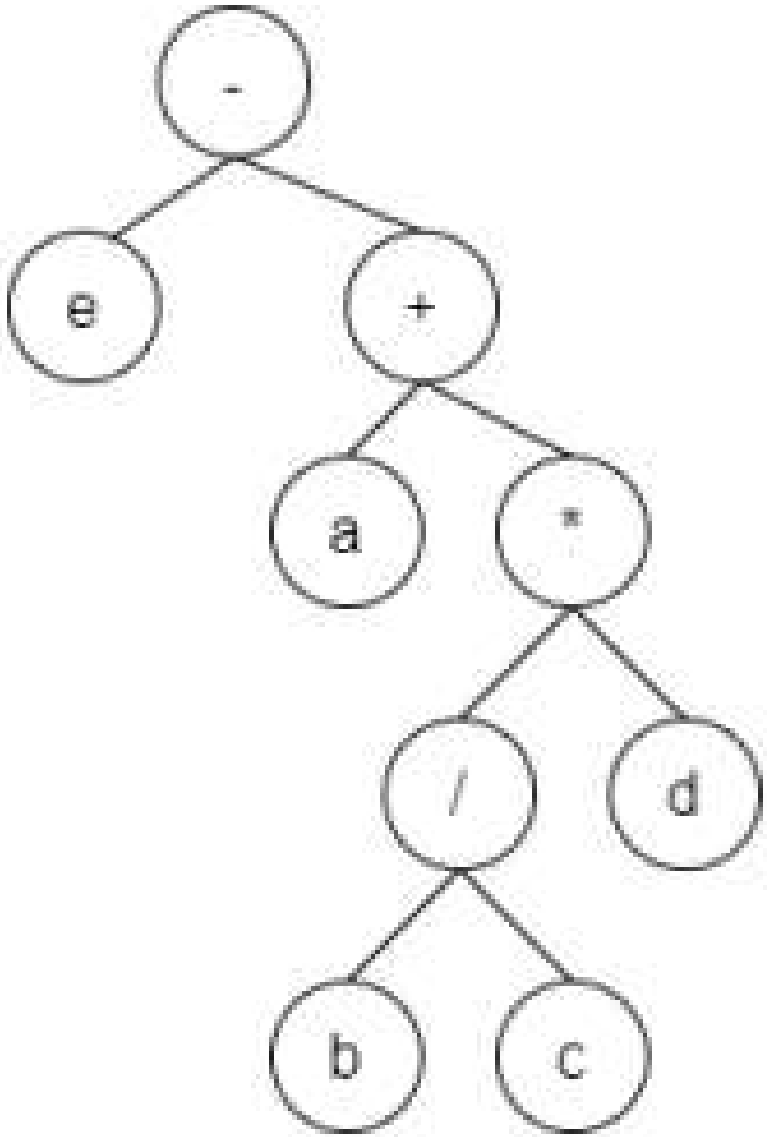
Expression tree

Given the binary tree, write down the expression that it represents



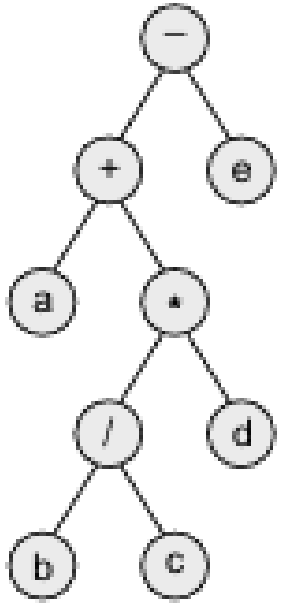
[{(a/b) + (c*d)} ^ {(f % g)/(h – i)}]

- (a + ((b/c) * d)) - e



- Given the expression, Exp = a + b / c * d – e, construct the corresponding binary tree.

Use the operator precedence chart to find the sequence in which operations will be performed. The given expression can be written as Exp = ((a + ((b/c) * d)) – e)



Expression tree

6. Tournament Trees

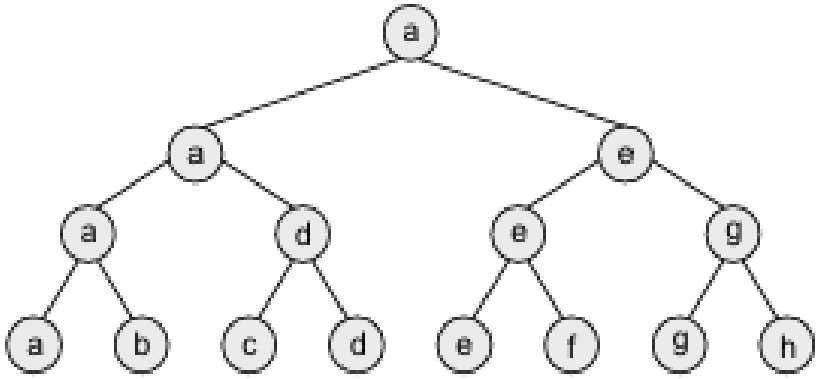
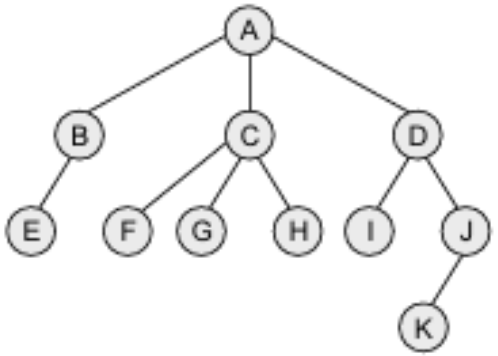


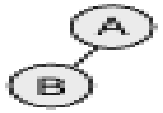
Figure 9.14 Tournament tree

- In a tournament tree (also called a selection tree), each external node represents a player and each internal node represents the winner of the match played between the players represented by its children nodes. These tournament trees are also called winner trees because they are being used to record the winner at each level. We can also have a loser tree that records the loser at each level.
- Consider the tournament tree given in Fig. There are 8 players in total whose names are represented using , b, c, d, e, f, g, and h. In round 1, a and b; c and d; e and f; and finally g and h play against each other. In round 2, the winners of round 1, that is, a, d, e, and g play against each other. In round 3, the winners of round 2, a and e play against each other. Whosoever wins is declared the winner. In the tree, the root node a specifies the winner.

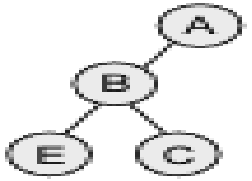
Creating a Binary Tree From a general Tree



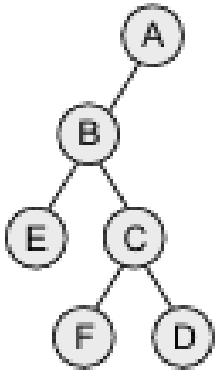
Step 1



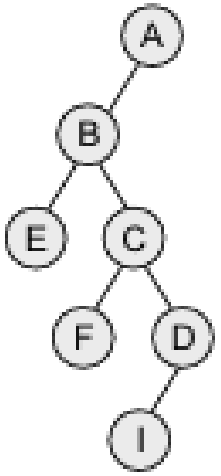
Step 2



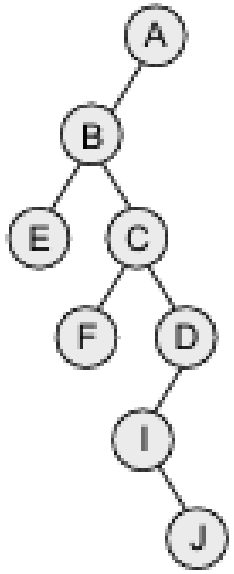
Step 3



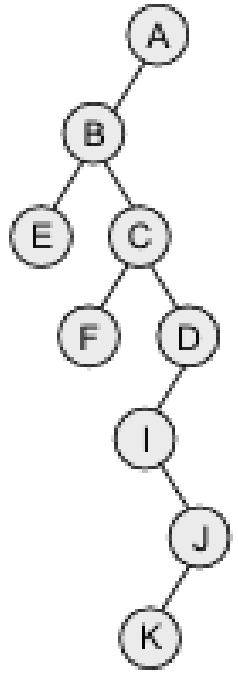
Step 4



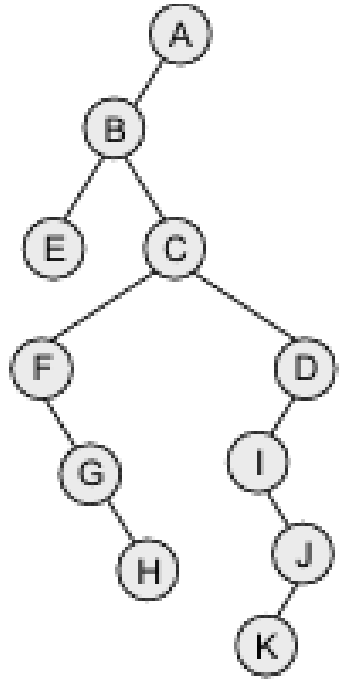
Step 5



Step 6



Step 7



Step 8

Creating a Binary Tree From a general Tree

- The rules for converting a general tree to a binary tree are given below.
- Note that a general tree is converted into a binary tree and not a binary search tree.

Rule 1: Root of the binary tree = Root of the general tree

Rule 2: Left child of a node = Leftmost child of the node in the binary tree B C D in the general tree

Rule 3: Right child of a node in the binary tree = Right sibling of the node in the general tree

let us build the binary tree.

Step 1: Node A is the root of the general tree, so it will also be the root of the binary tree.

Step 2: Left child of node A is the leftmost child of node A in the general tree and right child of node A is the right sibling of the node A in the general tree. Since node A has no right sibling in the general tree, it has no right child in the binary tree.

Step 3: Now process node B. Left child of B is E and its right child is C (right sibling in general tree).

Step 4: Now process node C. Left child of C is F (leftmost child) and its right child is D (right sibling in general tree).

Step 5: Now process node D. Left child of D is I (leftmost child). There will be no right child of D because it has no right sibling in the general tree.

Step 6: Now process node I. There will be no left child of I in the binary tree because I has no left child in the general tree. However, I has a right sibling J, so it will be added as the right child of I.

Step 7: Now process node J. Left child of J is K (leftmost child). There will be no right child of J because it has no right sibling in the general tree.

Step 8: Now process all the unprocessed nodes (E, F, G, H, K) in the same fashion, so the resultant binary tree can be given as follows.

Traversing a Binary Tree

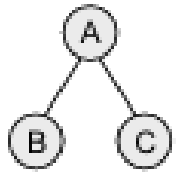
- Traversing a binary tree is the process of visiting each node in the tree exactly once in a systematic way.
- Linear data structures in which the elements are traversed sequentially, tree is a nonlinear data structure in which the elements can be traversed in many different ways.

Pre-order Traversal

- Binary tree

To traverse a non-empty binary tree in pre-order, the following operations are performed recursively at each node. The algorithm works by:

1. Visiting the root node,
2. Traversing the left sub-tree, and finally
3. Traversing the right sub-tree.

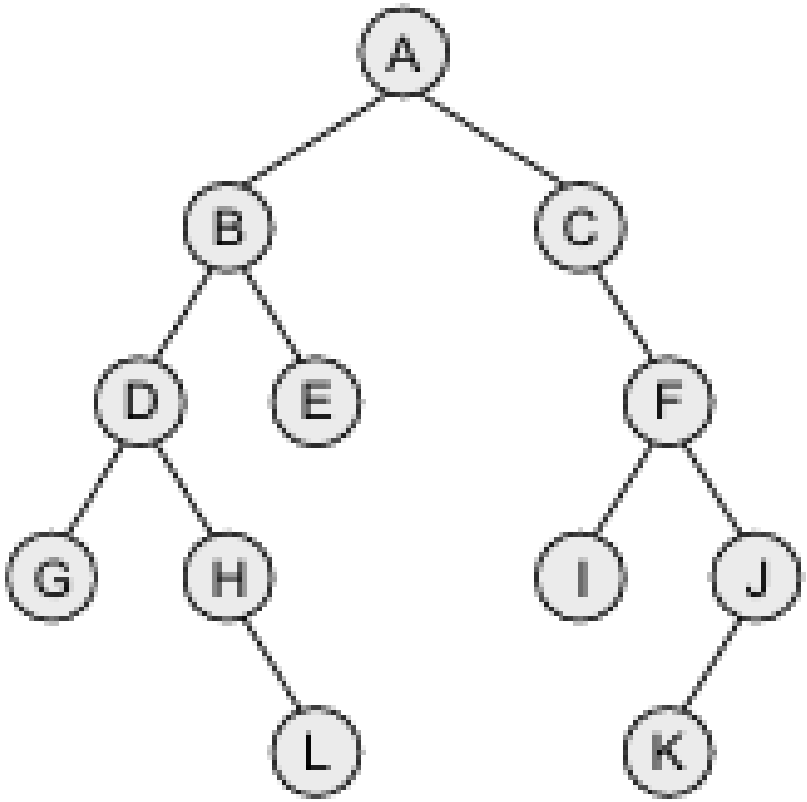


The pre-order traversal of the tree is given as A, B, C. Root node first, the left sub-tree next, and then the right sub-tree. Pre-order traversal is also called as depth-first traversal.

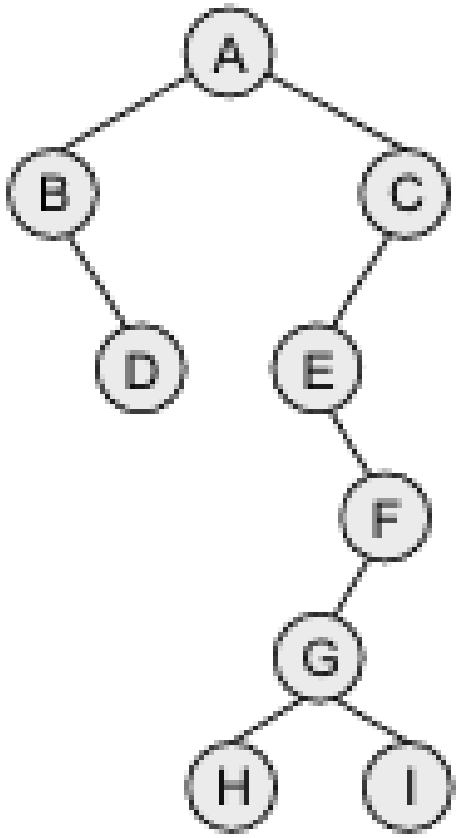
In this algorithm, the left sub-tree is always traversed before the right sub-tree.

The word ‘pre’ in the pre-order specifies that the root node is accessed prior to any other nodes in the left and right sub-trees. Pre-order algorithm is also known as the NLR traversal algorithm (Node-Left-Right)

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:           Write TREE -> DATA
Step 3:           PREORDER(TREE -> LEFT)
Step 4:           PREORDER(TREE -> RIGHT)
       [END OF LOOP]
Step 5: END
```



(a)



(b)

Pre-Order

TRAVERSAL ORDER: A, B, D, G, H, L, E, C, F, I, J, and K (fig (a))

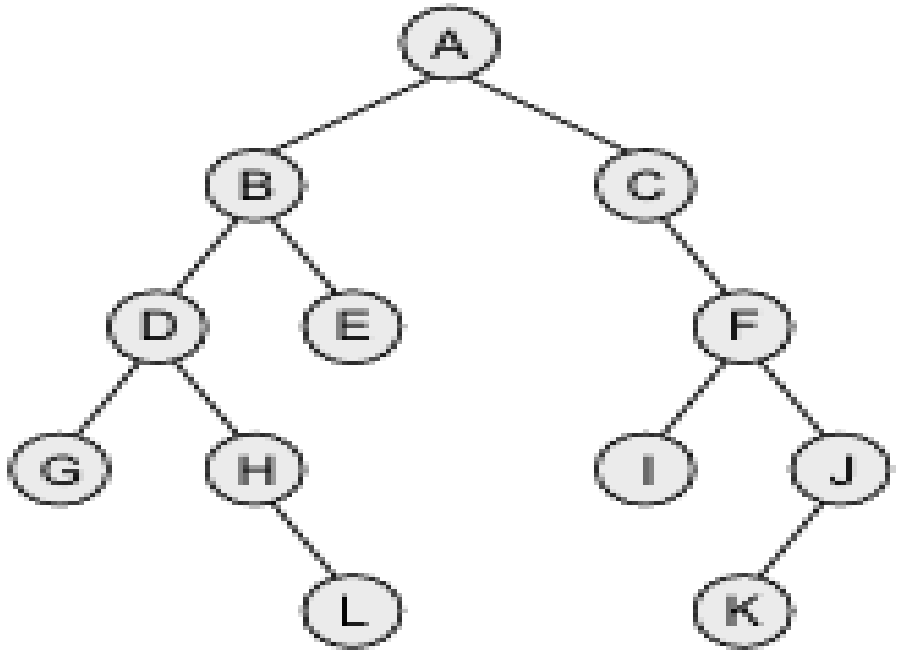
TRAVERSAL ORDER: A, B, D, C, E, F, G, H, and I (fig (b))

In-order Traversal

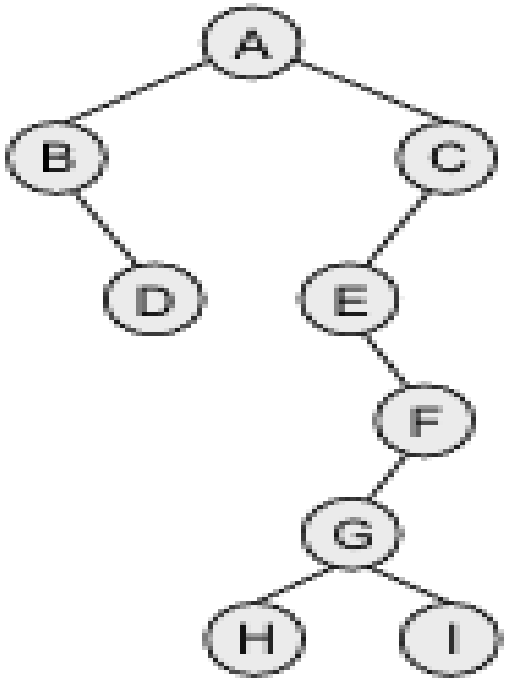
- To traverse a non-empty binary tree in in-order, the following operations are performed recursively at each node. The algorithm works by:

1. Traversing the left sub-tree,
2. Visiting the root node, and finally
3. Traversing the right sub-tree

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            INORDER(TREE -> LEFT)
Step 3:            Write TREE -> DATA
Step 4:            INORDER(TREE -> RIGHT)
       [END OF LOOP]
Step 5: END
```



(a)



(b)

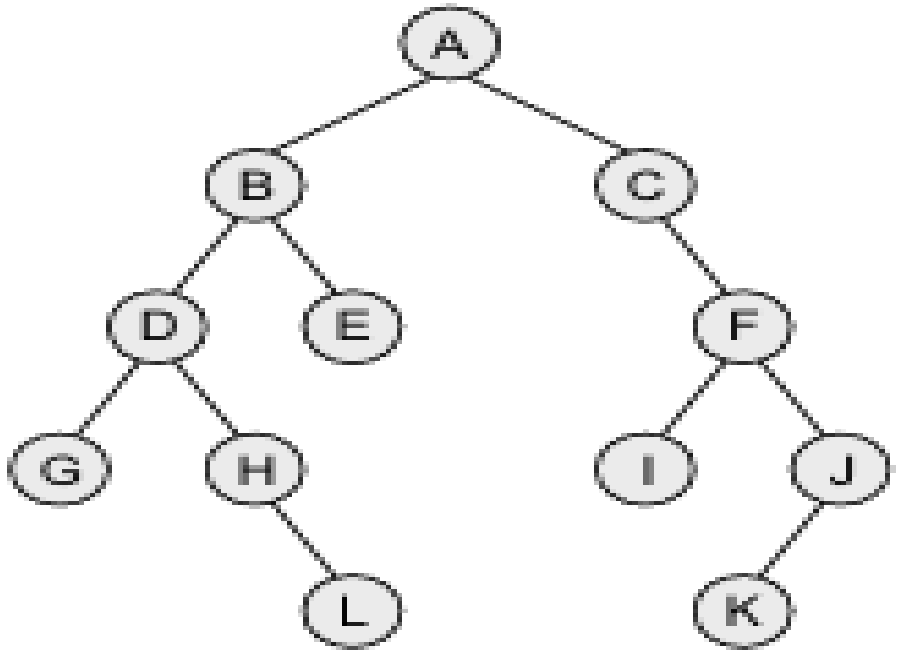
Inorder (Left-Root-Right)
TRAVERSAL ORDER: G, D, H, L, B, E, A, C, I, F, K, and J
TRAVERSAL ORDER: B, D, A, E, H, G, I, F, and C

Post-order Traversal

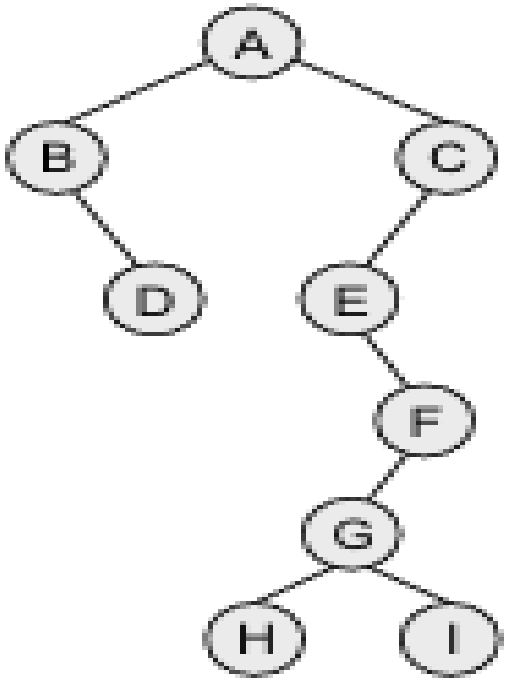
- To traverse a non-empty binary tree in post-order, the following operations are performed recursively at each node. The algorithm works by:

1. Traversing the left sub-tree,
2. Traversing the right sub-tree, and finally
3. Visiting the root node.

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:           POSTORDER(TREE -> LEFT)
Step 3:           POSTORDER(TREE -> RIGHT)
Step 4:           Write TREE -> DATA
       [END OF LOOP]
Step 5: END
```



(a)

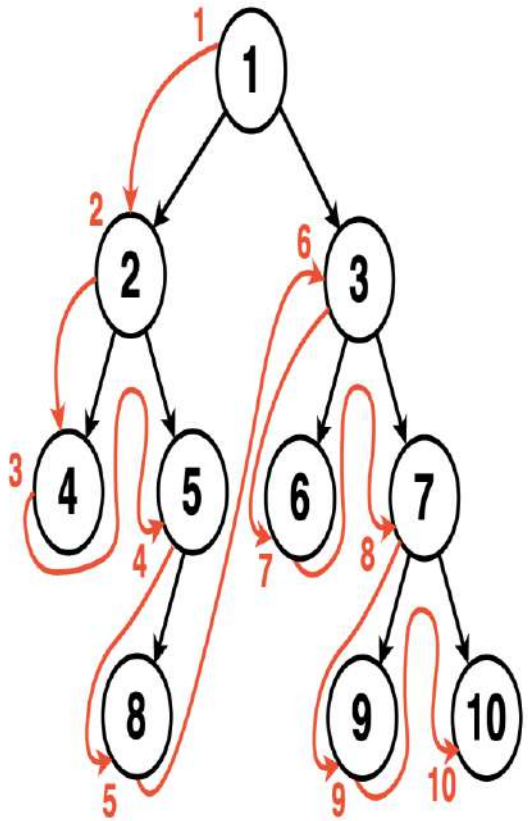


(b)

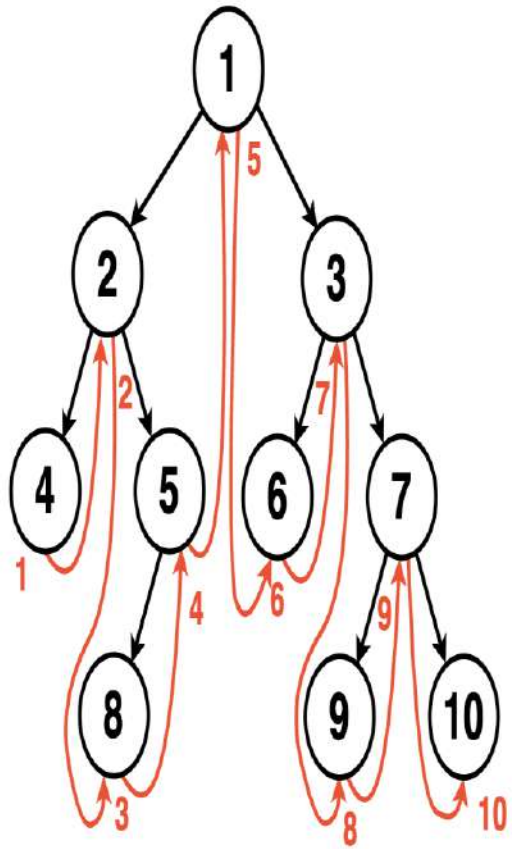
Postorder(left-Right-Root):

TRAVERSAL ORDER: G, L, H, D, E, B, I, K, J, F, C, and A

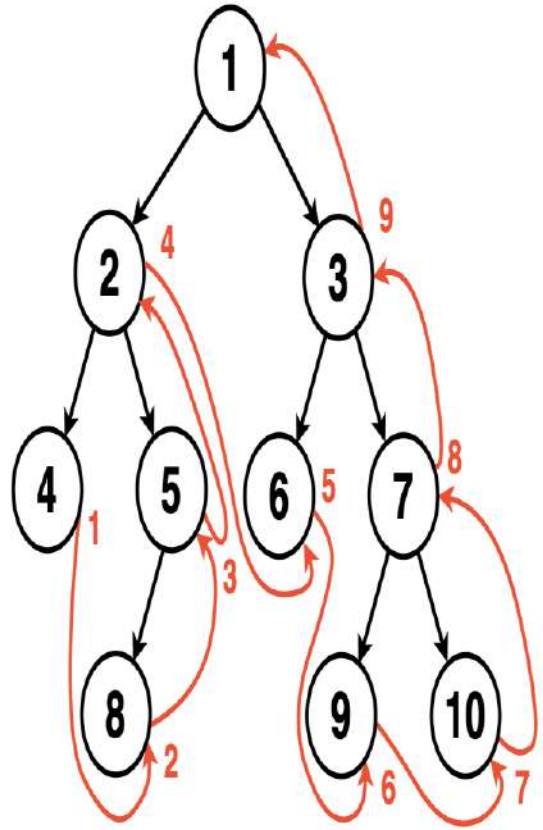
TRAVERSAL ORDER: D, B, H, I, G, F, E, C, and A



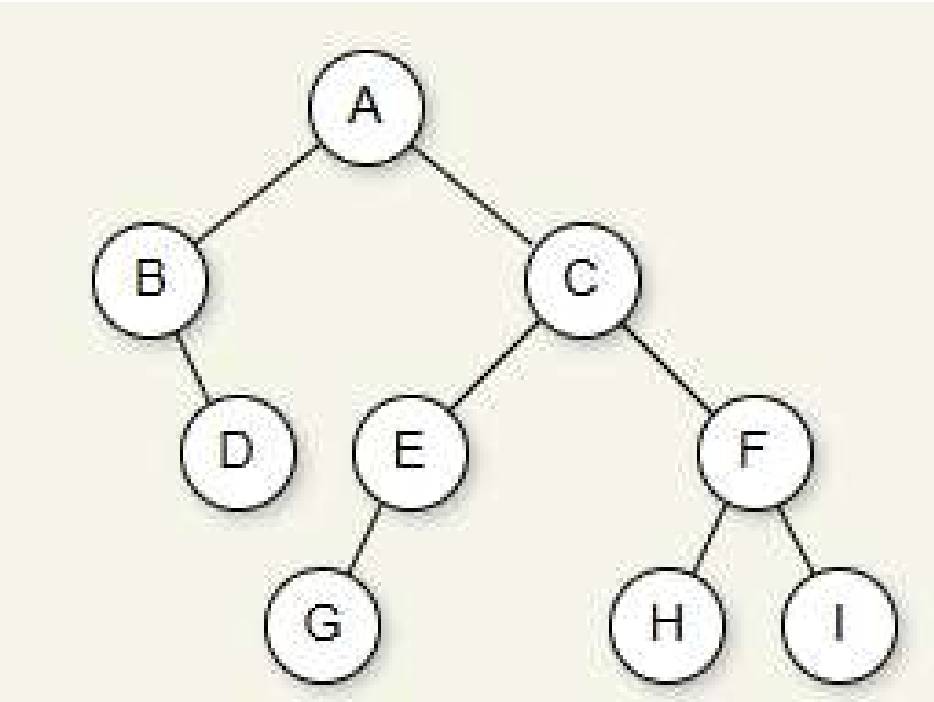
preorder arr = [1, 2, 4, 5, 8, 3, 6, 7, 9, 10]



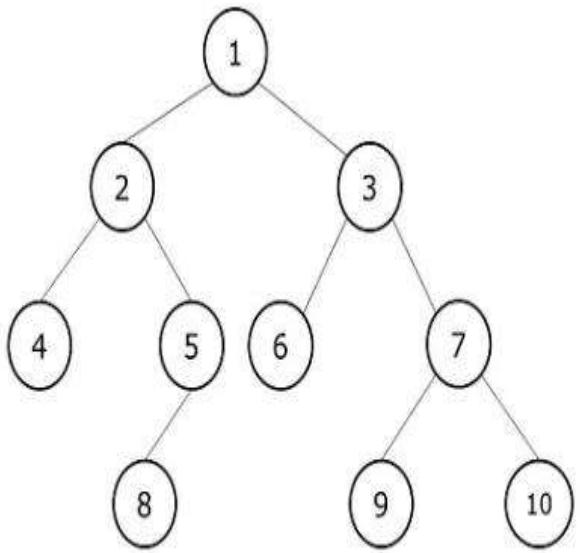
inorder arr = [4, 2, 8, 5, 1, 6, 3, 9, 7, 10]



postorder arr = [4, 8, 5, 2, 6, 9, 10, 7, 3, 1]



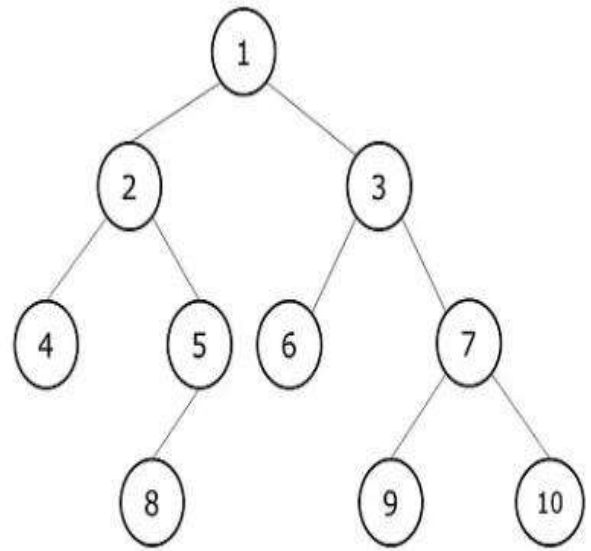
Preorder: **A B D C E G F H I**.
Postorder: **D B G E H I F C A**.
Inorder: **B D A G E C H F I**.



Preorder Traversal:

[root, left, right]

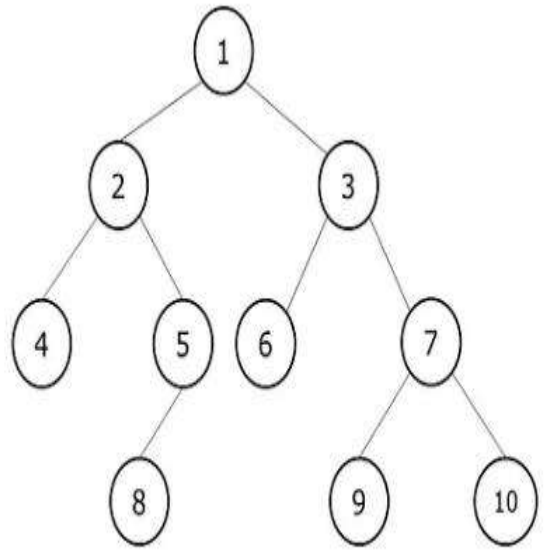
1	2	4	5	8	3	6	7	9	10



Postorder Traversal:

[left, right, root]

4	8	5	2	6	9	10	7	3	1



Inorder Traversal:

[left,root,right]

4	2	8	5	1	6	3	9	7	10

InOrder(root) visits nodes in the following order:

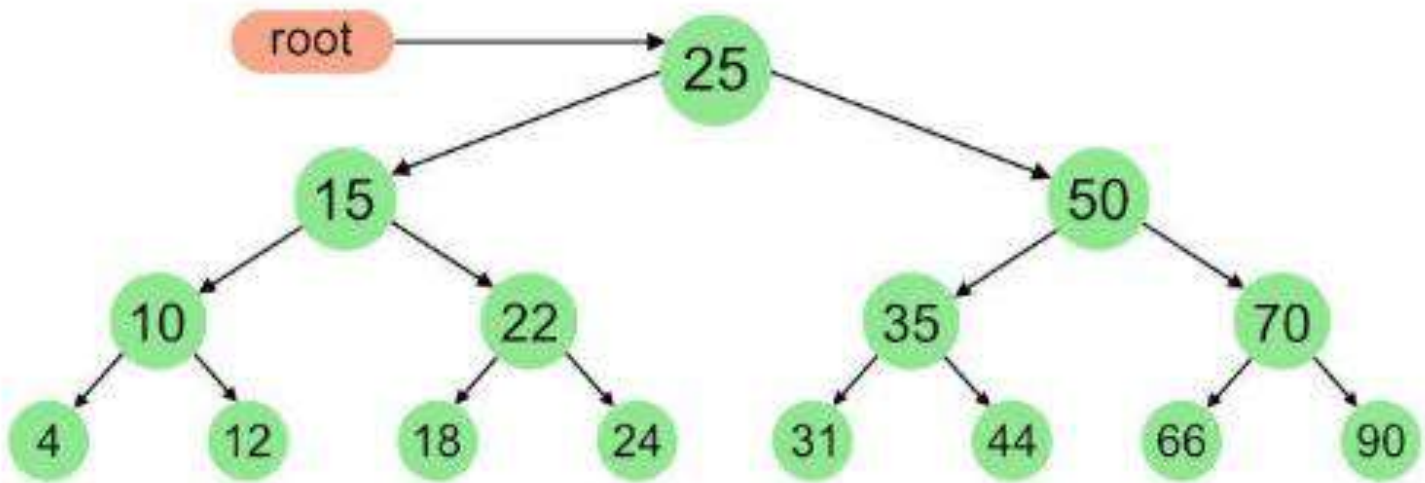
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

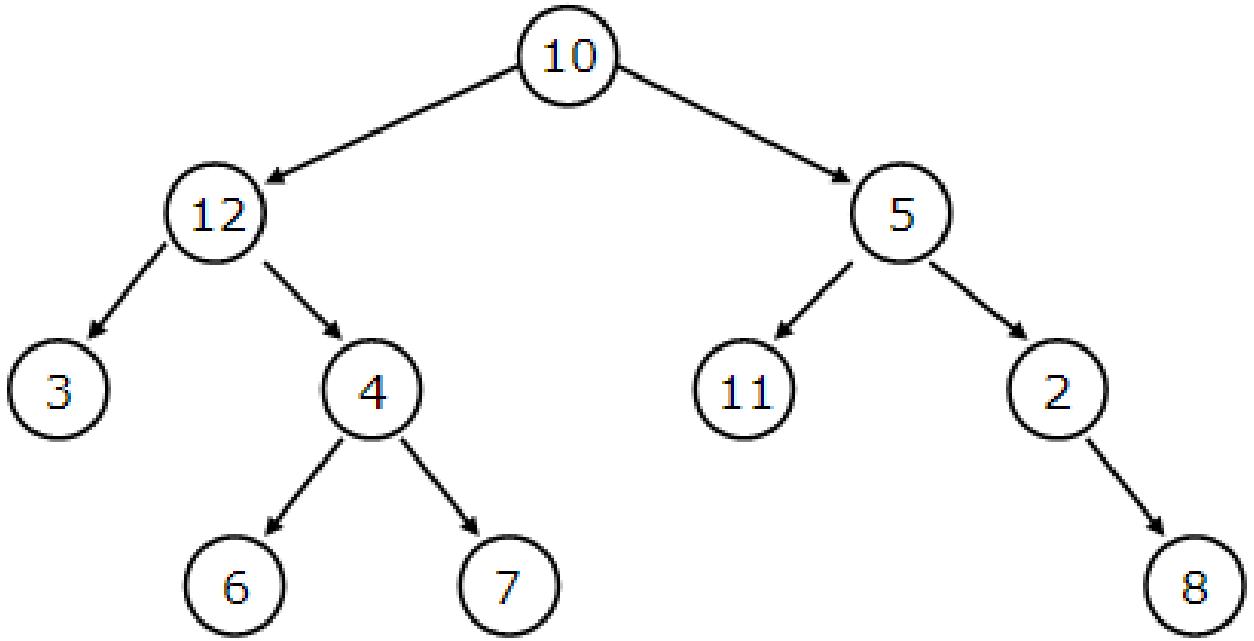
A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25





Levelorder tree traversal

10, 12, 5, 3, 4, 11, 2, 6, 7, 8

Inorder tree traversal

3, 12, 6, 4, 7, 10, 11, 5, 2, 8

Preorder tree traversal

10, 12, 3, 4, 6, 7, 5, 11, 2, 8

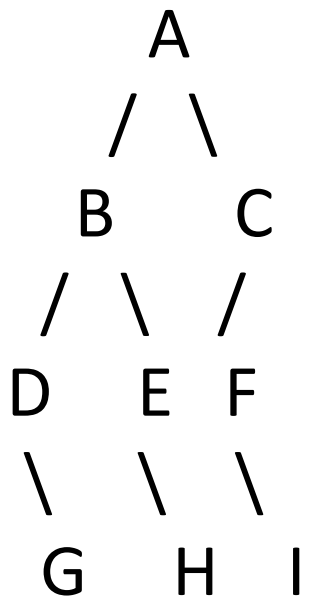
Postorder tree traversal

3, 6, 7, 4, 12, 11, 8, 2, 5, 10

Given the following inorder and preorder traversal reconstruct a binary tree

Preorder – A, B, D, G, E, H, C, F, I

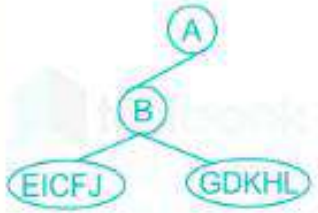
Inorder – D, G, B, E, H, A, F, I, C

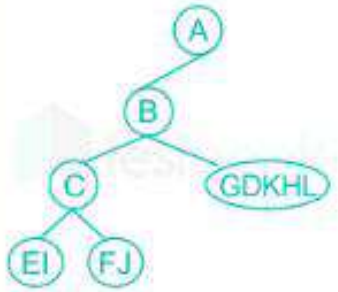


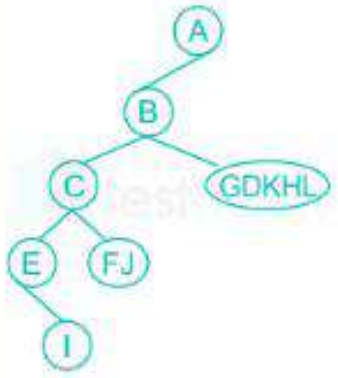
- A is the root (first in preorder).
- In inorder, everything left of A is its left subtree, everything right is right subtree.
- For the left subtree, B becomes the next root; splitting around B gives its children (D–G on left, E on right).
- or the right subtree, **C** becomes the next root; splitting around C gives **F** (left) and **I** (right child of F).

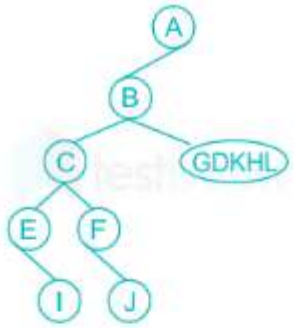
- Generate Binary Tree

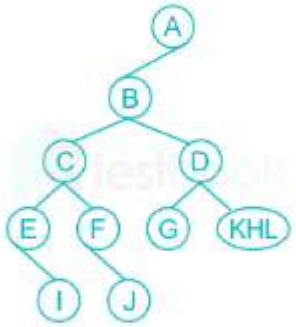
Consider the traversal of a tree
Preorder → ABCEIFJDGHKL
Inorder → EICFJBGDKHLA

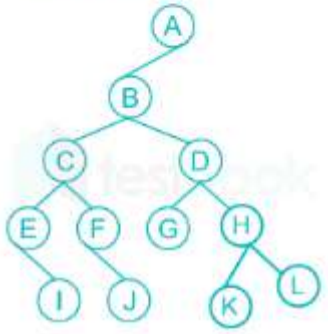












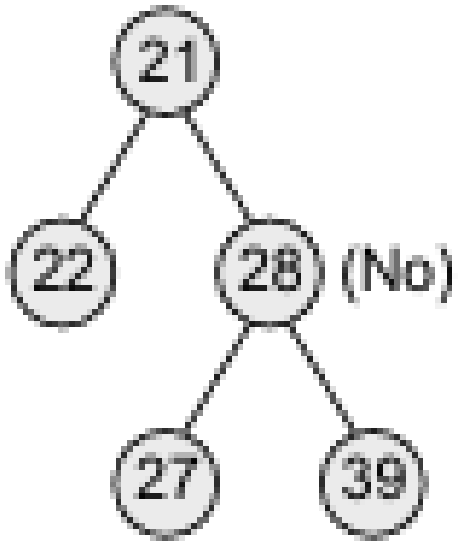
Applications of Tree

- Trees are used to store simple as well as complex data. Here simple means an integer value, character value and complex data means a structure or a record.
- Trees are often used for implementing other types of data structures like hash tables, sets, and maps.
- A self-balancing tree, Red-black tree is used in kernel scheduling, to preempt massively multi processor computer operating system use.
- Another variation of tree, B-trees are prominently used to store tree structures on disc. They are used to index a large number of records.
- B-trees are also used for secondary indexes in databases, where the index facilitates a select operation to answer some range criteria.
- Trees are an important data structure used for compiler construction.
- Trees are also used in database design.
- Trees are used in file system directories.
- Trees are also widely used for information storage and retrieval in symbol tables

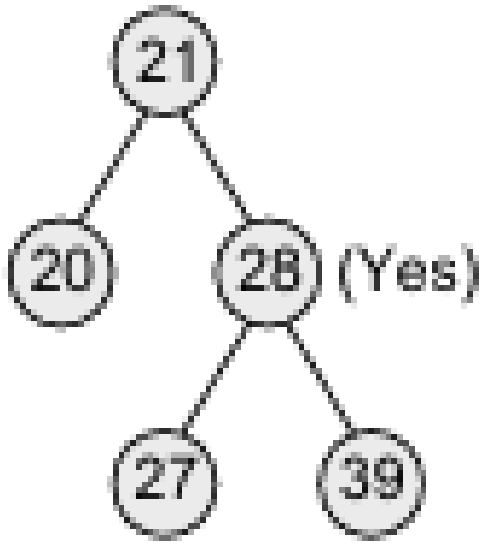
BINARY SEARCH TREES

- A binary search tree, also known as an ordered binary tree, is a variant of binary trees in which the nodes are arranged in an order.
- In a binary search tree, all the nodes in the left sub-tree have a value less than that of the root node. Correspondingly, all the nodes in the right sub-tree have a value either equal to or greater than the root node.

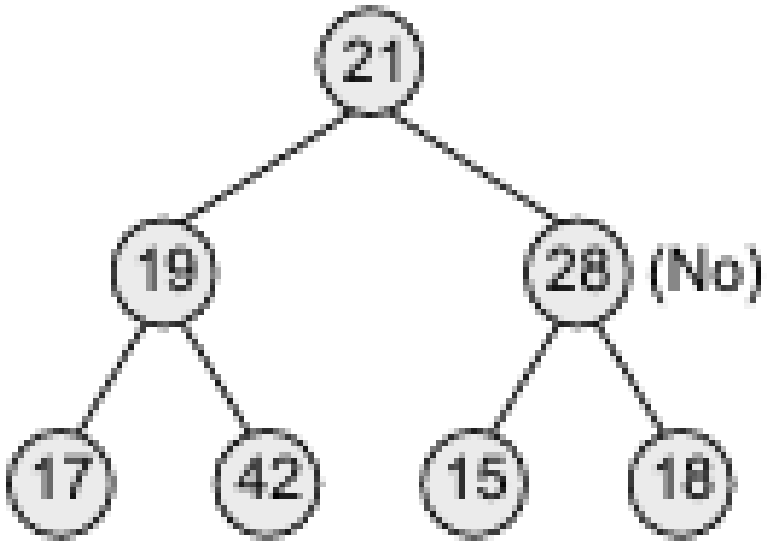
State whether the binary trees in are binary search trees or not.



(No)

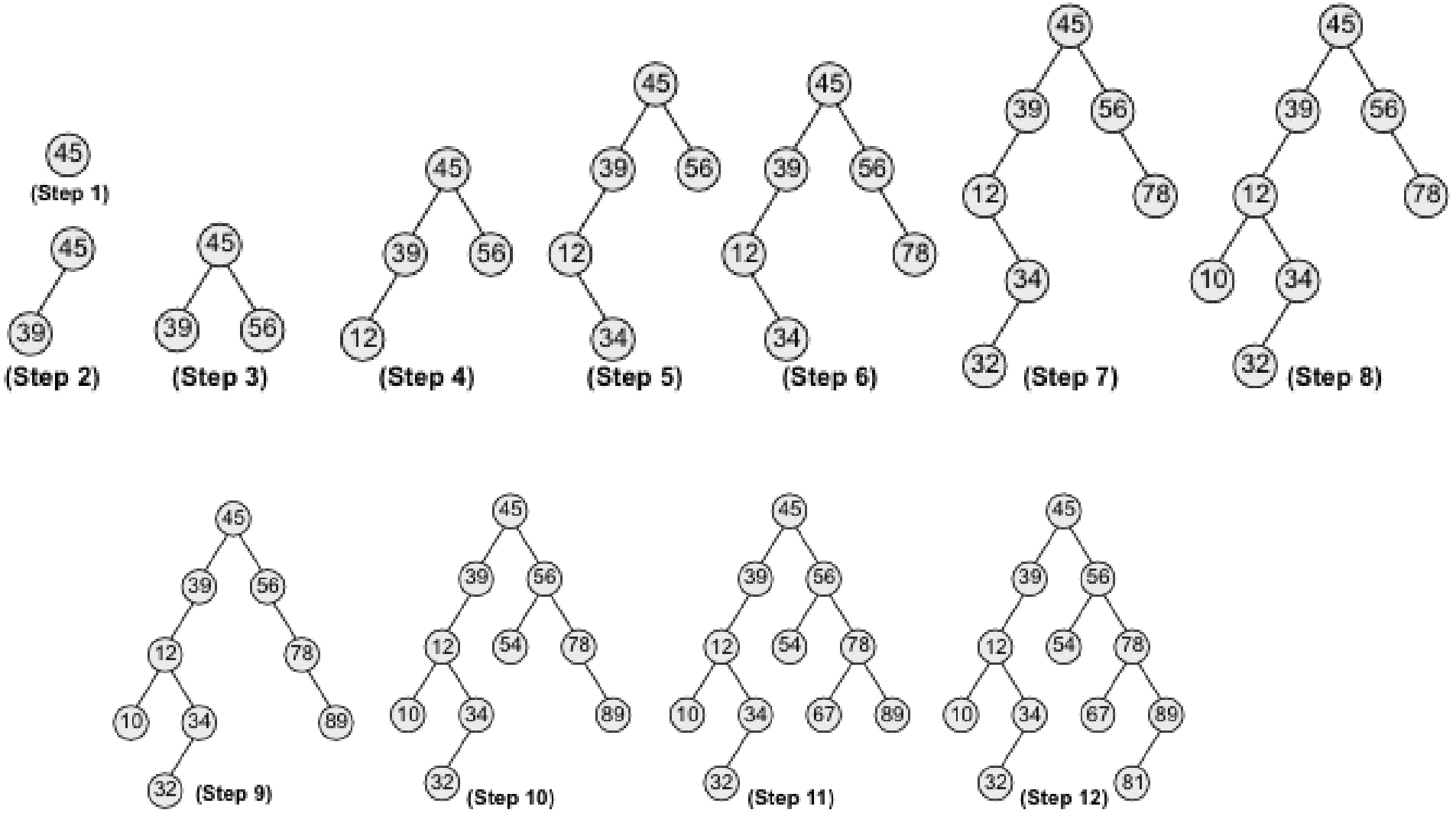


(Yes)



(No)

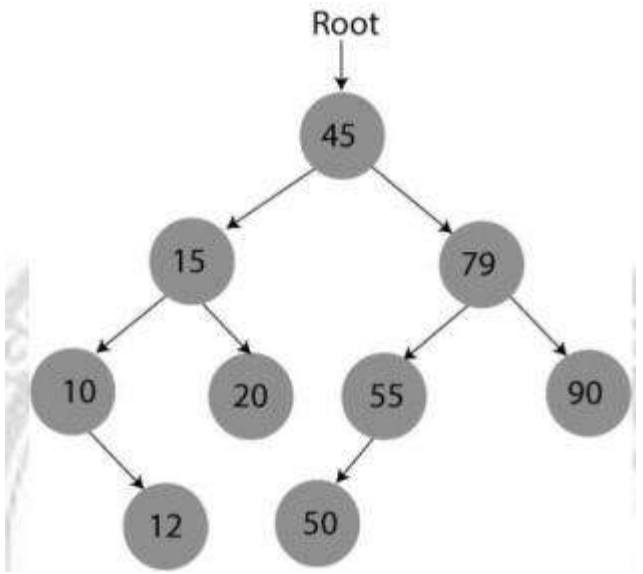
- Create a binary search tree using the following data elements: 45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81



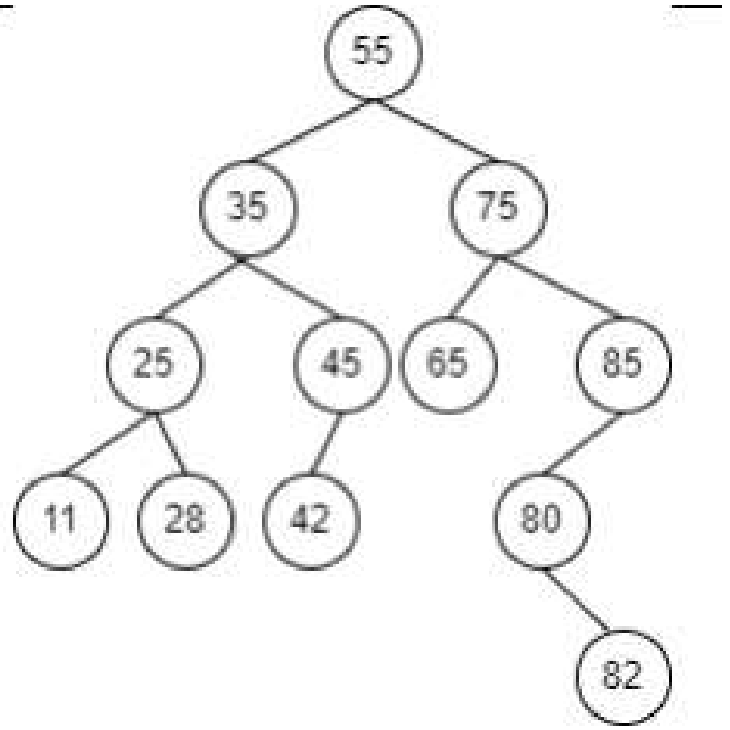
- **Binary Search Tree (BST) for the following sequence of numbers-**

47, 12, 75, 88, 90, 73, 57, 1, 85, 50, 62

45, 15, 79, 90, 10, 55, 12, 20, 50

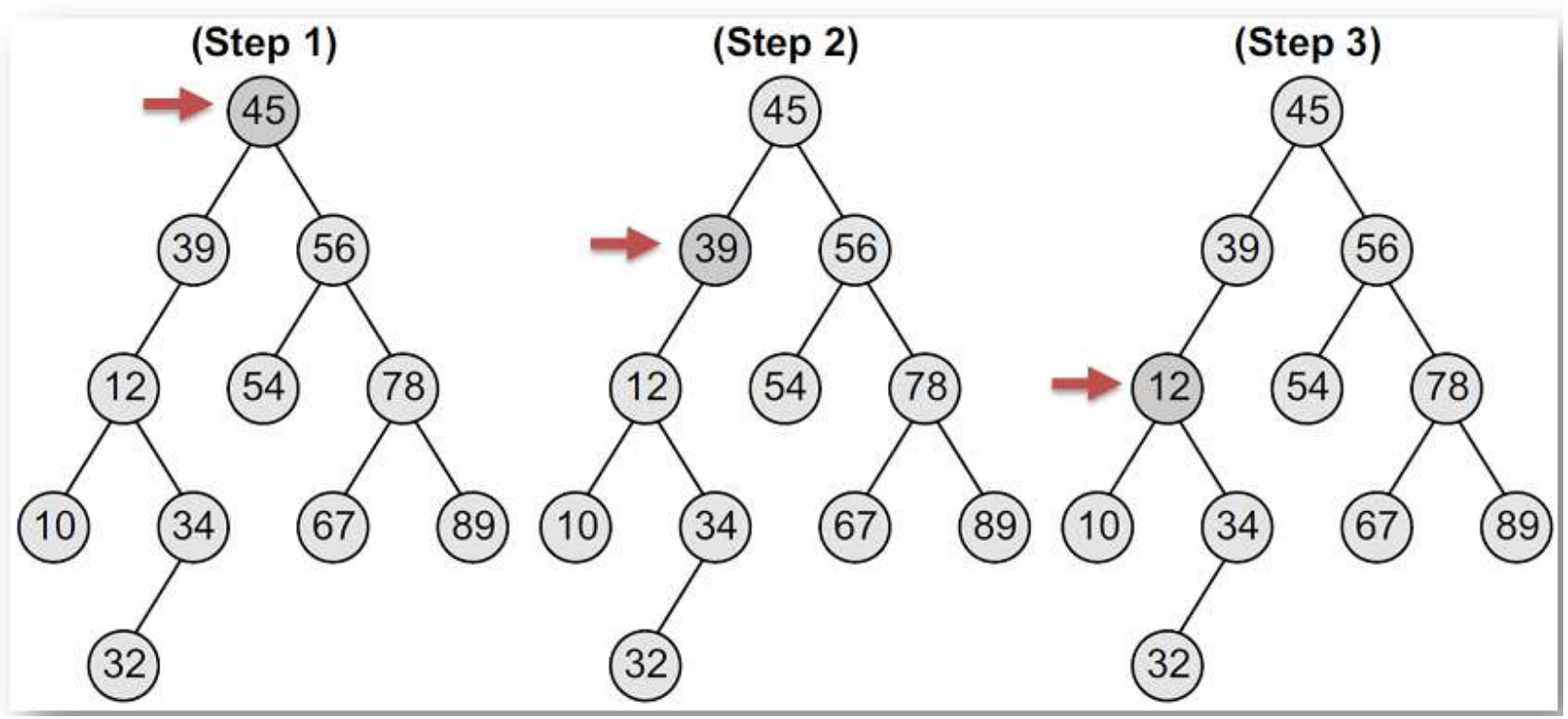


55, 35, 75, 25, 45, 85, 65, 42, 11, 80, 85, 82, 28



OPERATIONS ON BINARY SEARCH TREES

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

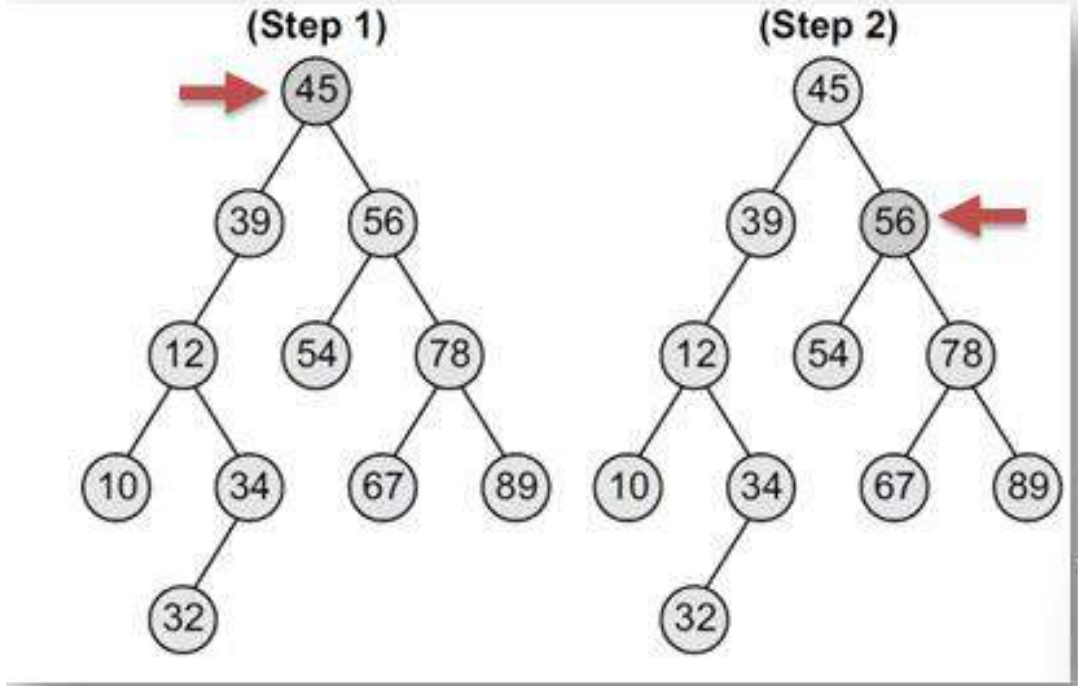


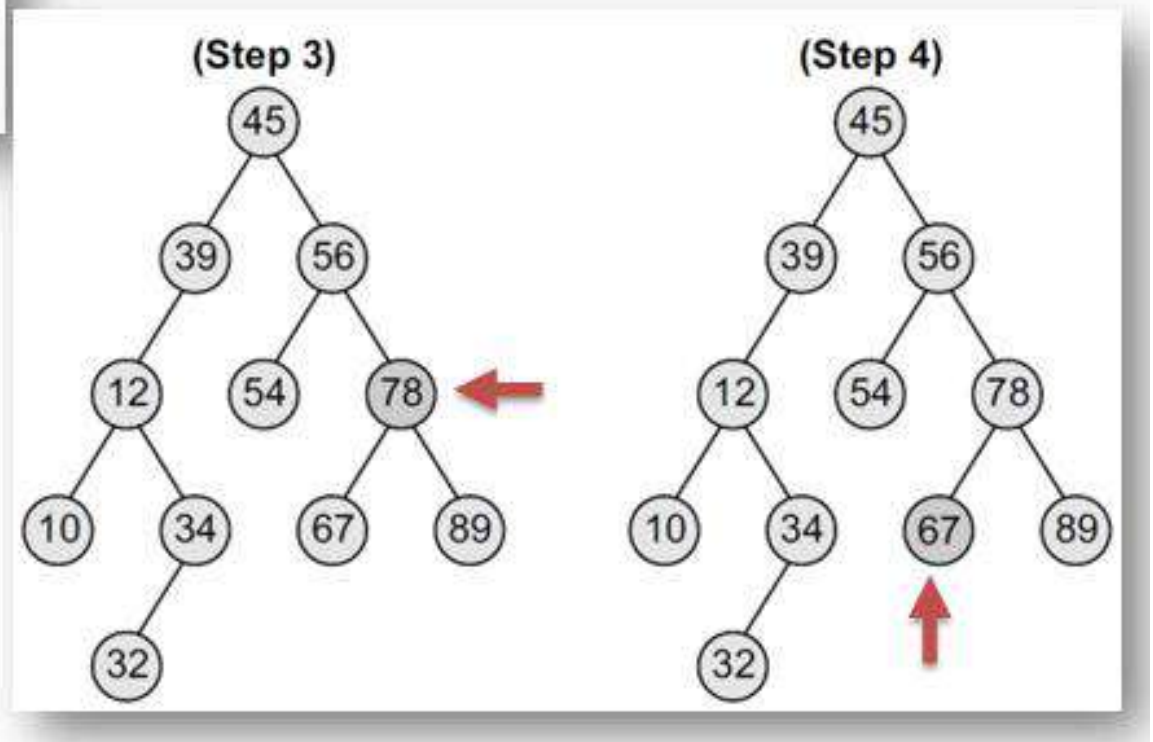
OPERATIONS ON BINARY SEARCH TREES

- Searching for a Node in a Binary Search Tree
- The search function is used to find whether a given value is present in the tree or not– Checks if the binary search tree is empty– Compare the value
  - Find
  - Go left
  - Go right

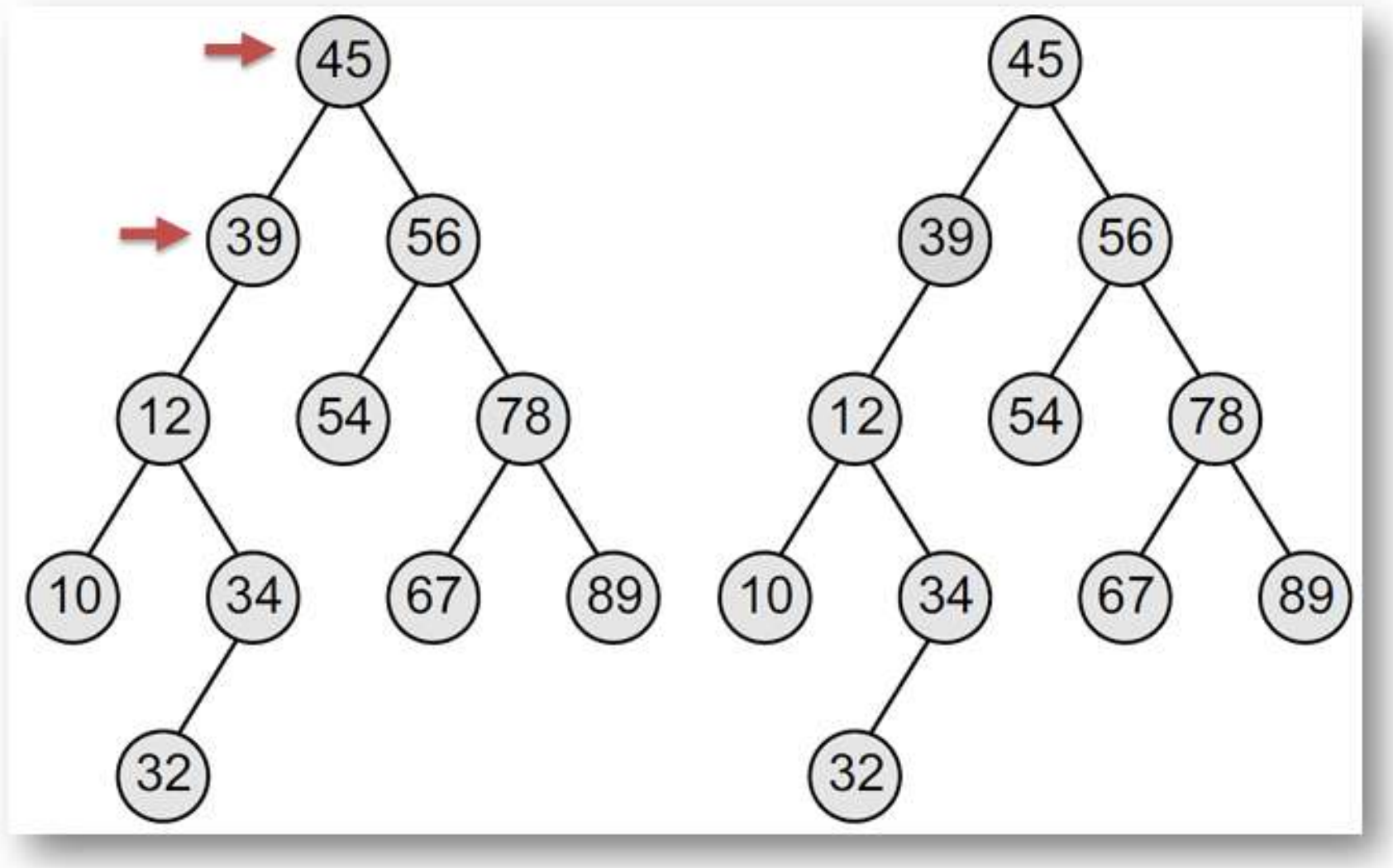
```
searchElement (TREE, VAL)

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
```

- Searching a node with value 67





- Searching a node with the value 40

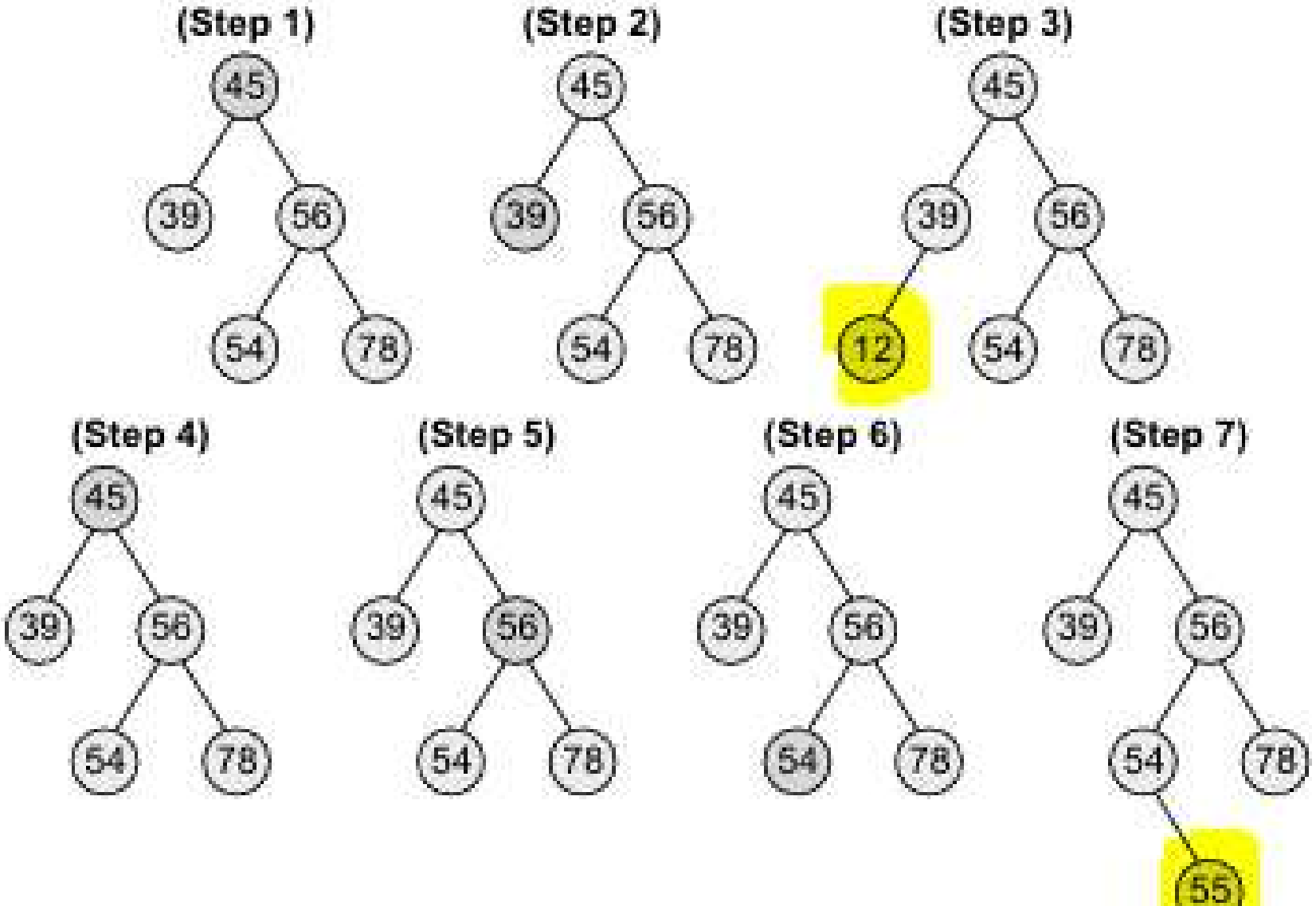


Inserting a New Node in a Binary Search Tree

- The insert function is used to add a new node with a given value at the correct position in the binary search tree. Adding the node at the correct position means that the new node should not violate the properties of the binary search tree.

```
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
```

Inserting nodes with values 12 and 55 in the given binary search tree



Deleting a Node from a Binary Search Tree

- The delete function deletes a node from the binary search tree– In order to take care the properties of binary search tree, we can divide the deleting functions into three categories
  - Deleting a node that has no children
  - Deleting a node with one child
  - Deleing a node with two children

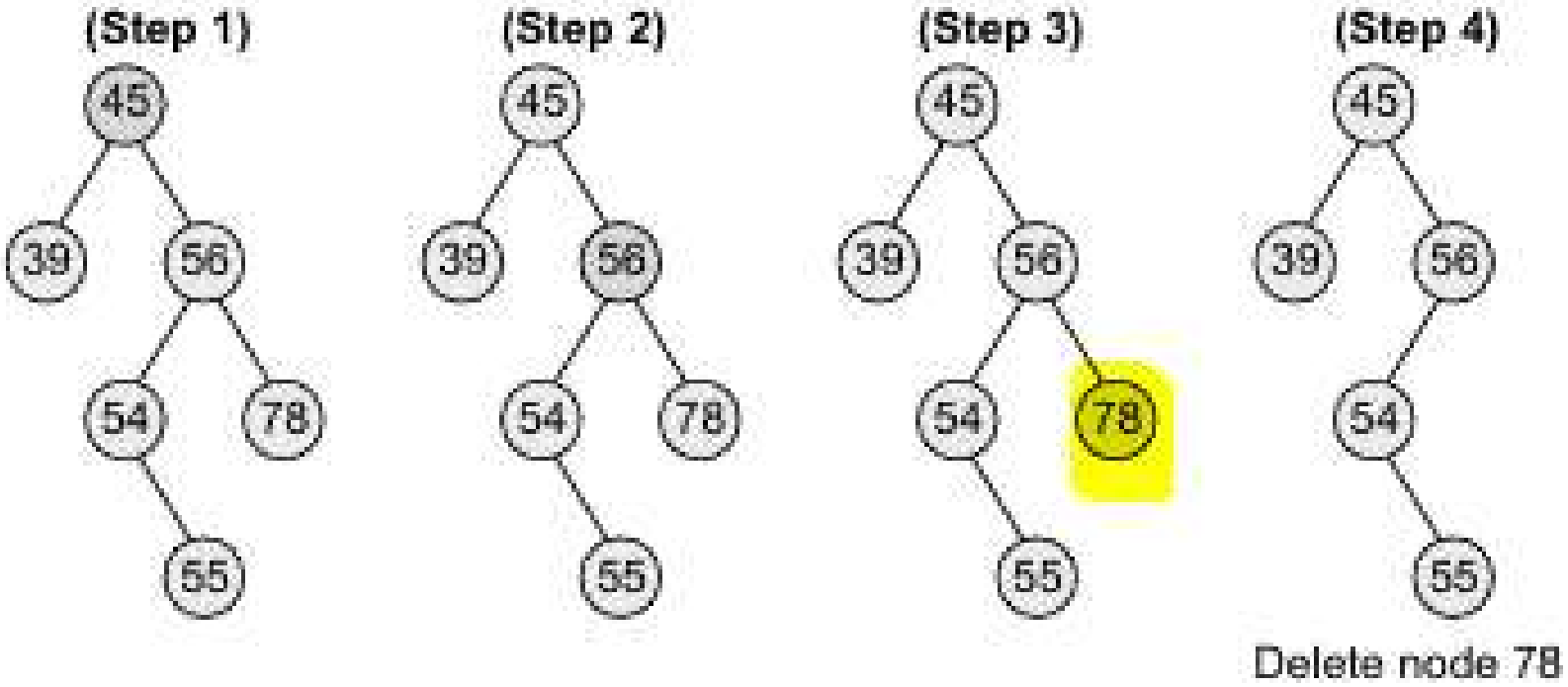
```
Delete (TREE, VAL)

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)
          SET TREE -> DATA = TEMP -> DATA
          Delete(TREE -> LEFT, 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
```

Deleing a node with two children

Deleting a node that has no children

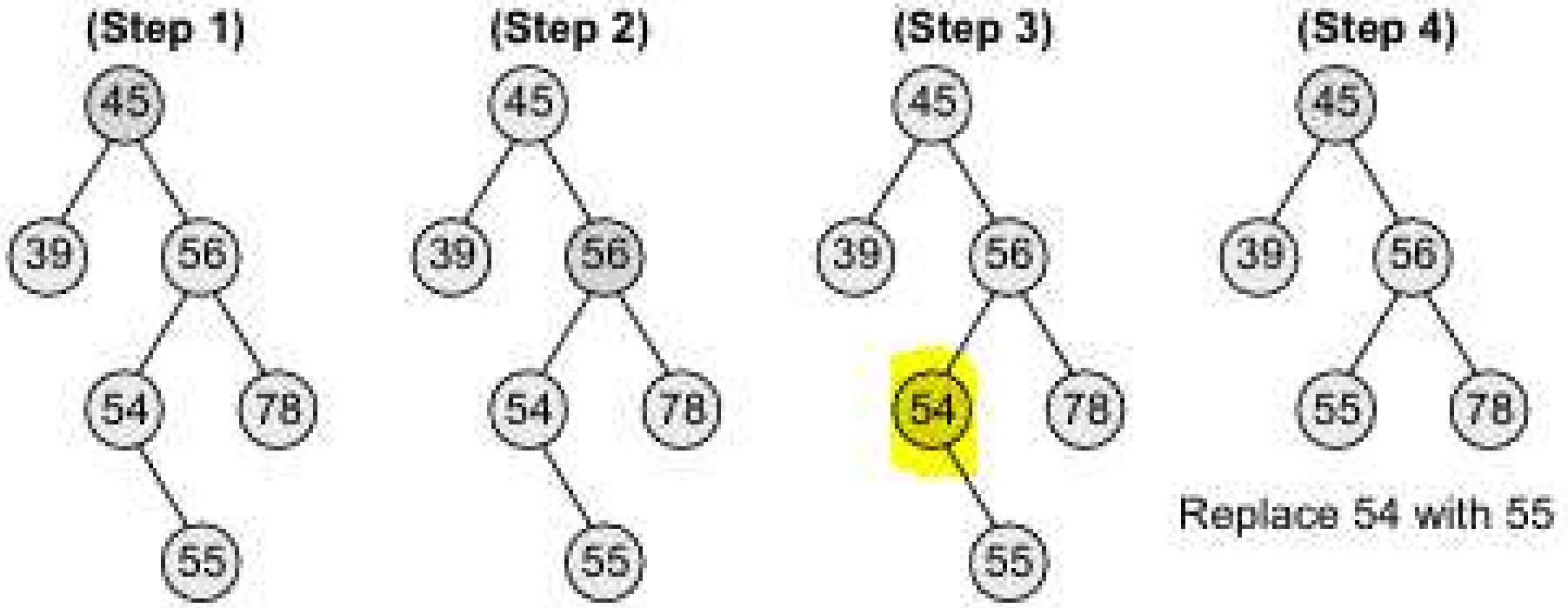
Deleting a node with one child

Case 1: Deleting node 78 from the given BST(Node that has no children)



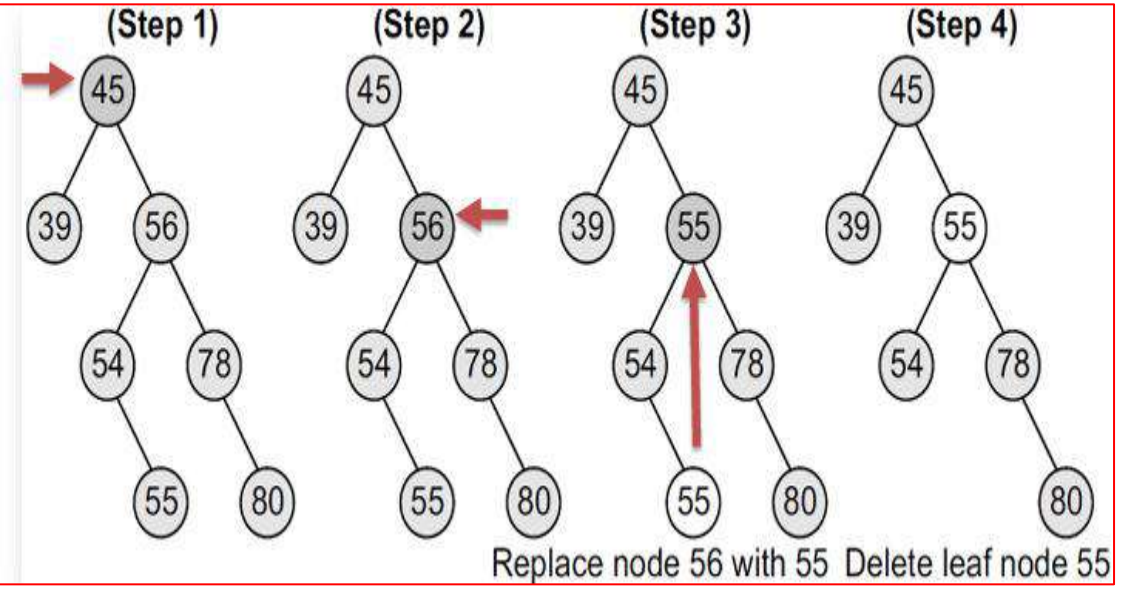
Case 2: Deleting a Node with One Child

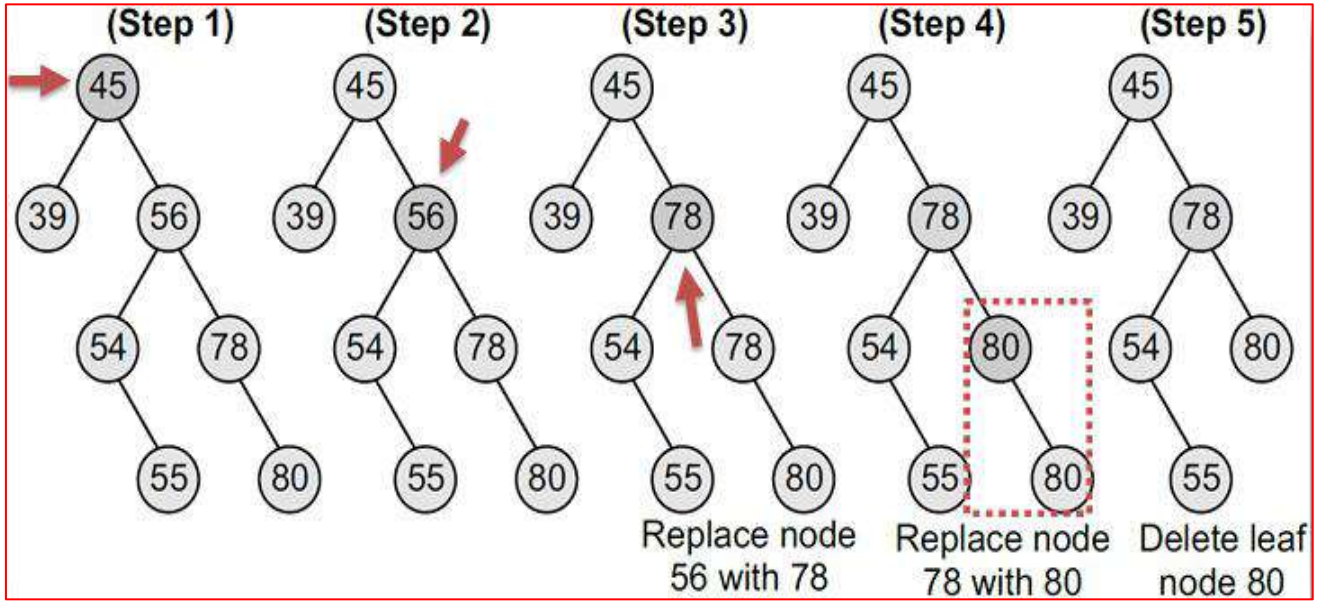
Deleting node 54 from the given binary search tree



Case 3: Deleting a Node with Two Children

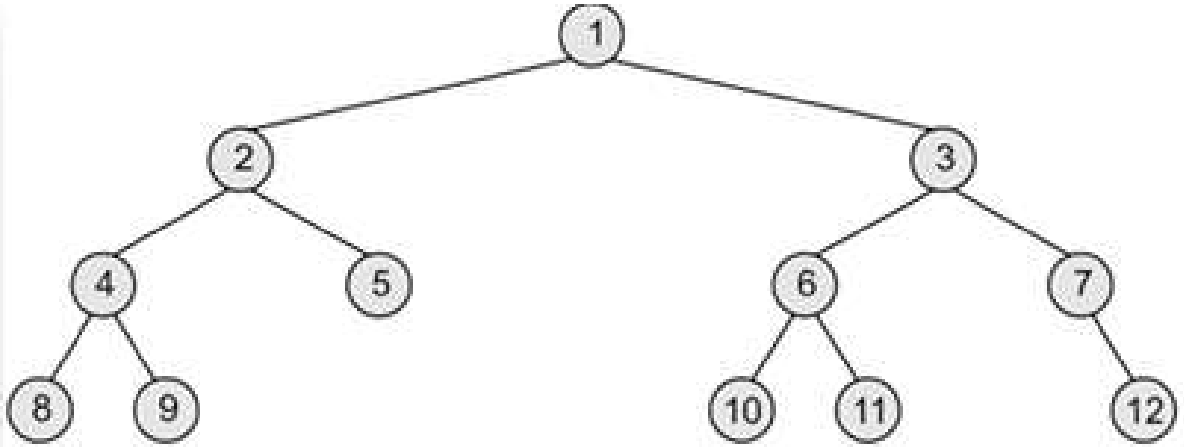
- Deleing a node with two children– Replace the node's value with its in-order predecessor (largest value in the left sub-tree) or in-order successor (smallest value in the right sub-tree)
- Deleting node 56 from the given binary search tree

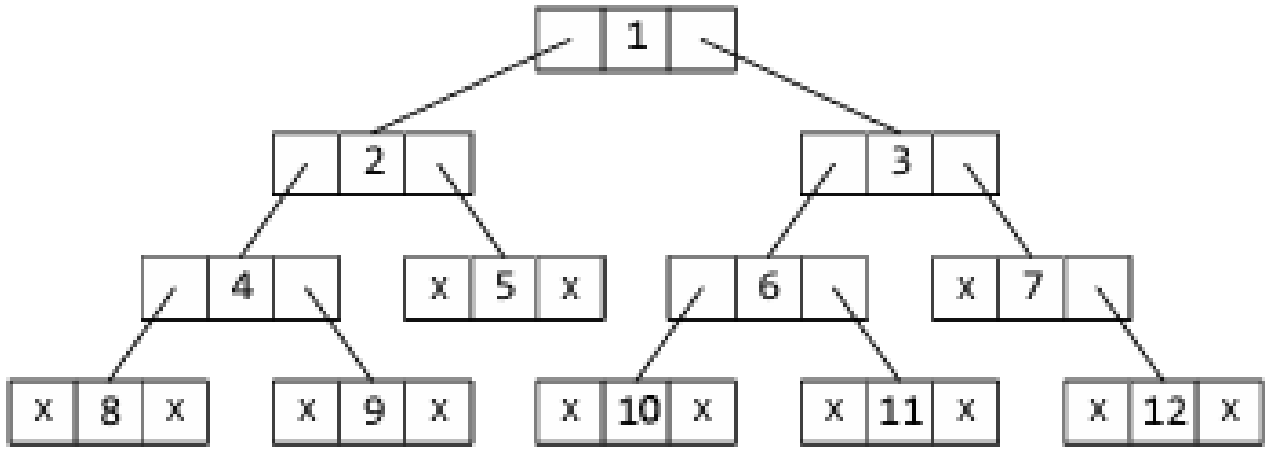




THREADED BINARY TREES

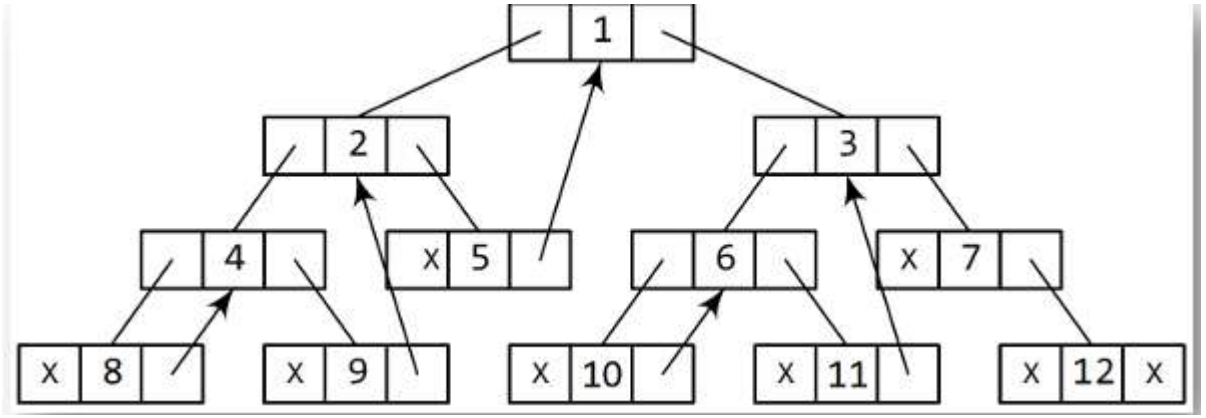
- A threaded binary tree is the same as that of a binary tree but with a difference in storing the NULL pointers– The space that is wasted in storing a NULL pointer can be efficiently used to store some other useful piece of information





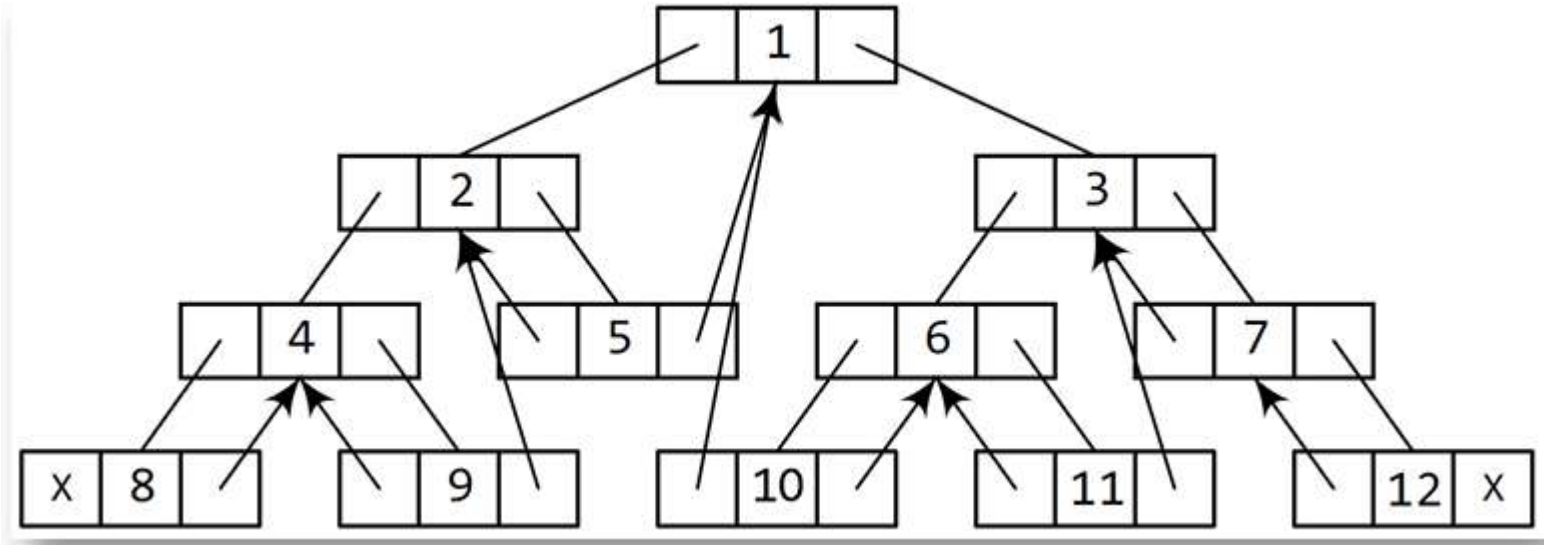
One way Threaded Trees

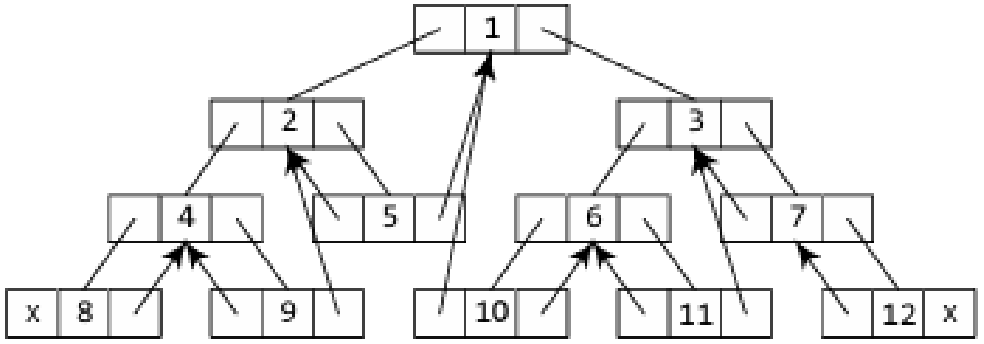
- A one-way threaded tree is also called a single-threaded tree– If the thread appears in the left field, then the left field will be made to point to the in-order predecessor of the node Such a tree is called a left-threaded binary tree
- – If the thread appears in the right field, then it will point to the in-order successor of the node Such a tree is called a right-threaded binary tree



Two way Threaded Trees

- In a two-way threaded tree, also called a double-threaded tree, threads will appear in both the left and the right field of the node– The left field will point to the in-order predecessor of the node, and the right field will point to its successor– A two-way threaded binary tree is also called a fully threaded binary tree





Memory representation of binary trees: (a) without threading, (b) with one-way, and (c) two-way threading

		LEFT	DATA	RIGHT
ROOT 3	1	-1	8	-1
	2	-1	10	-1
→	3	5	1	8
	4			
	5	9	2	14
	6			
	7			
	8	20	3	11
	9	1	4	12
	10			
	11	-1	7	18
	12	-1	9	-1
	13			
	14	-1	5	-1
→	15			
15 AVAIL	16	-1	11	-1
	17			
	18	-1	12	-1
	19			
	20	2	6	16

(a)

		LEFT	DATA	RIGHT
ROOT 3	1	-1	8	9
	2	-1	10	20
→	3	5	1	8
	4			
	5	9	2	14
	6			
	7			
	8	20	3	11
	9	1	4	12
	10			
	11	-1	7	18
	12	-1	9	5
	13			
	14	-1	5	3
→	15			
15 AVAIL	16	-1	11	8
	17			
	18	-1	12	-1
	19			
	20	2	6	16

(b)

		LEFT	DATA	RIGHT
ROOT 3	1	-1	8	9
	2	3	10	20
→	3	5	1	8
	4			
	5	9	2	14
	6			
	7			
	8	20	3	11
	9	1	4	12
	10			
	11	8	7	18
	12	9	9	5
	13			
	14	5	5	3
→	15			
15 AVAIL	16	20	11	8
	17			
	18	11	12	-1
	19			
	20	2	6	16

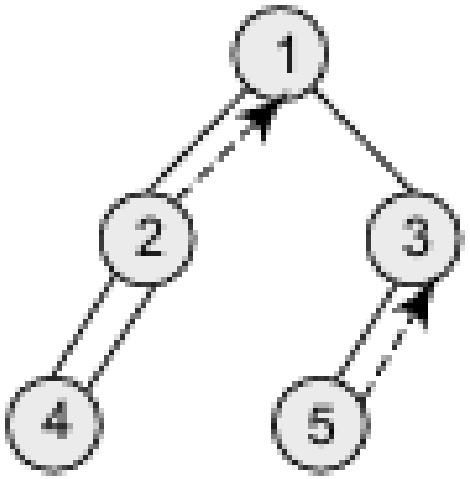
(c)

Traversing a Threaded Binary Tree

For every node, visit the left sub-tree first, provided if one exists and has not been visited earlier. Then the node (root) itself is followed by visiting its right sub-tree (if one exists).

Step 1: Check if the current node has a left child that has not been visited. If a left child exists that has not been visited, go to Step 2, else go to Step 3.
Step 2: Add the left child in the list of visited nodes. Make it as the current node and then go to Step 6.
Step 3: If the current node has a right child, go to Step 4 else go to Step 5.
Step 4: Make that right child as current node and go to Step 6.
Step 5: Print the node and if there is a threaded node make it the current node.
Step 6: If all the nodes have visited then END else go to Step 1.

Algorithm for in-order traversal of a threaded binary tree



1. Node 1 has a left child i.e., 2 which has not been visited. So, add 2 in the list of visited nodes, make it as the current node.
2. Node 2 has a left child i.e., 4 which has not been visited. So, add 4 in the list of visited nodes, make it as the current node.
3. Node 4 does not have any left or right child, so print 4 and check for its threaded link. It has a threaded link to node 2, so make node 2 the current node.
4. Node 2 has a left child which has already been visited. However, it does not have a right child. Now, print 2 and follow its threaded link to node 1. Make node 1 the current node.
5. Node 1 has a left child that has been already visited. So print 1. Node 1 has a right child 3 which has not yet been visited, so make it the current node.
6. Node 3 has a left child (node 5) which has not been visited, so make it the current node.
7. Node 5 does not have any left or right child. So print 5. However, it does have a threaded link which points to node 3. Make node 3 the current node.
8. Node 3 has a left child which has already been visited. So print 3.
9. Now there are no nodes left, so we end here. The sequence of nodes printed is—4 2 1 5 3.

Advantages of Threaded Binary Tree

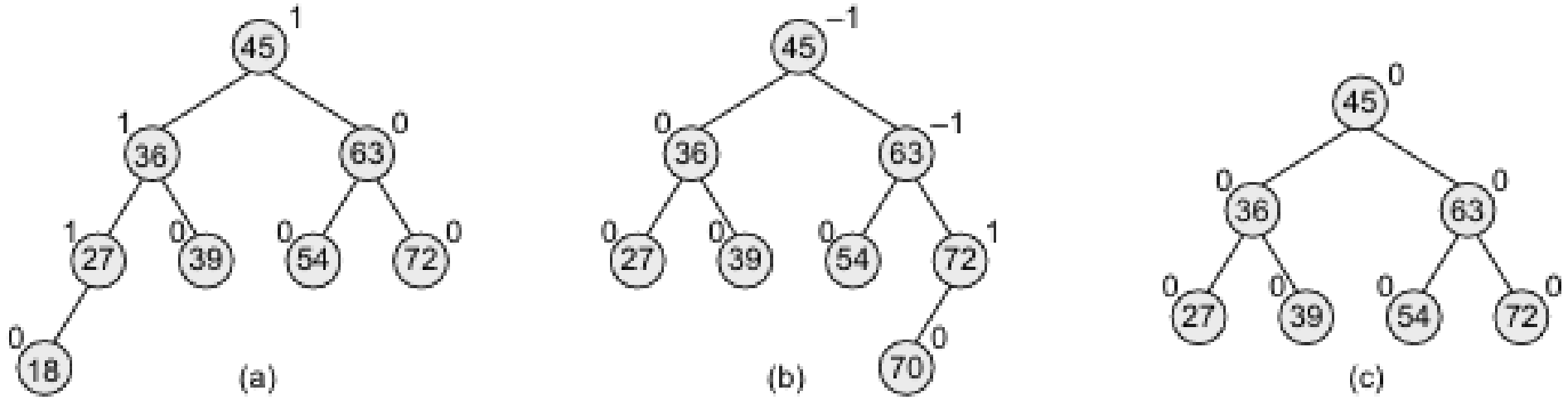
- It enables linear traversal of elements in the tree.
- Linear traversal eliminates the use of stacks which in turn consume a lot of memory space and computer time.
- It enables to find the parent of a given element without explicit use of parent pointers.
- Since nodes contain pointers to in-order predecessor and successor, the threaded tree enables forward and backward traversal of the nodes as given by in-order fashion.

AVL Tree

- AVL tree is a self-balancing binary search tree invented by G.M. Adelson-Velsky and E.M. Landis in 1962.
- In an AVL tree, the heights of the two sub-trees of a node may differ by at most one. Due to this property, the AVL tree is also known as a height-balanced tree. The key advantage of using an AVL tree is that it takes O(log n) time to perform search, insert, and delete operations in an average case as well as the worst case because the height of the tree is limited to O(log n).
- The structure of an AVL tree is the same as that of a binary search tree but with a little difference.
- In its structure, it stores an additional variable called the BalanceFactor. Thus, every node has a balance factor associated with it. The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree.
- A binary search tree in which every node has a balance factor of –1, 0, or 1 is said to be height balanced.
- A node with any other balance factor is considered to be unbalanced and requires rebalancing of the tree.

Balance factor = Height (left sub-tree) – Height (right sub-tree)

- If the balance factor of a node is 1, then it means that the left sub-tree of the tree is one level higher than that of the right sub-tree. Such a tree is therefore called as a left-heavy tree.
- If the balance factor of a node is 0, then it means that the height of the left sub-tree (longest path in the left sub-tree) is equal to the height of the right sub-tree.
- If the balance factor of a node is –1, then it means that the left sub-tree of the tree is one level lower than that of the right sub-tree. Such a tree is therefore called as a right-heavy tree.



(a) Left-heavy AVL tree, (b) right-heavy tree, (c) balanced tree

Nodes 18, 39, 54, and 72 have no children, so their balance factor = 0. Node 27 has one left child and zero right child. So, the height of left sub-tree = 1, whereas the height of right sub-tree = 0. Thus, its balance factor = 1. Look at node 36, it has a left Efficient Binary Trees 317 sub-tree with height = 2, whereas the height of right sub-tree = 1. Thus, its balance factor = 2 – 1 = 1. Similarly, the balance factor of node 45 = 3 – 2 =1; and node 63 has a balance factor of 0 (1 – 1).

Searching for a Node in an AVL

- Tree Searching in an AVL tree is performed exactly the same way as it is performed in a binary search tree. Due to the height-balancing of the tree, the search operation takes O(log n) time to complete. Since the operation does not modify the structure of the tree, no special provisions are required.

Inserting a New Node in an AVL Tree

- Insertion in an AVL tree is also done in the same way as it is done in a binary search tree.
- In the AVL tree, the new node is always inserted as the leaf node. But the step of insertion is usually followed by an additional step of rotation.
- Rotation is done to restore the balance of the tree.
- if insertion of the new node does not disturb the balance factor, that is, if the balance factor of every node is still –1, 0, or 1, then rotations are not required.
- During insertion, the new node is inserted as the leaf node, so it will always have a balance factor equal to zero.
- The only nodes whose balance factors will change are those which lie in the path between the root of the tree and the newly inserted node. The possible changes which may take place in any node on the path are as follows:
  - Initially, the node was either left- or right-heavy and after insertion, it becomes balanced.
  - Initially, the node was balanced and after insertion, it becomes either left- or right-heavy.
  - Initially, the node was heavy (either left or right) and the new node has been inserted in the heavy sub-tree, thereby creating an unbalanced sub-tree. Such a node is said to be a critical node.

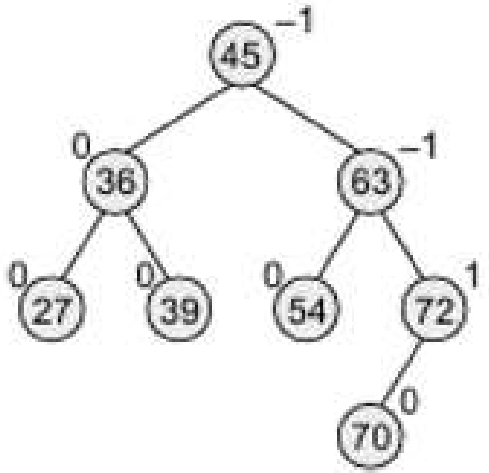


Figure AVL tree

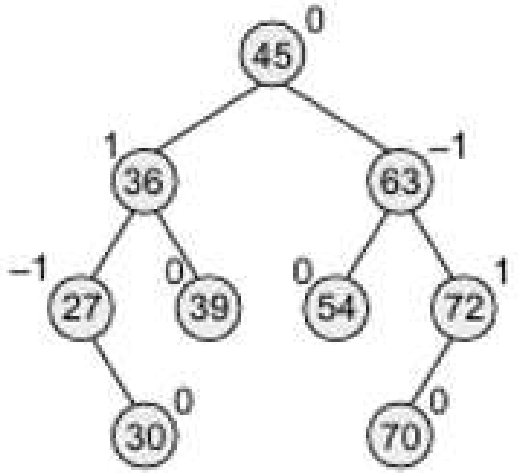


Figure AVL tree after inserting a node with the value 30

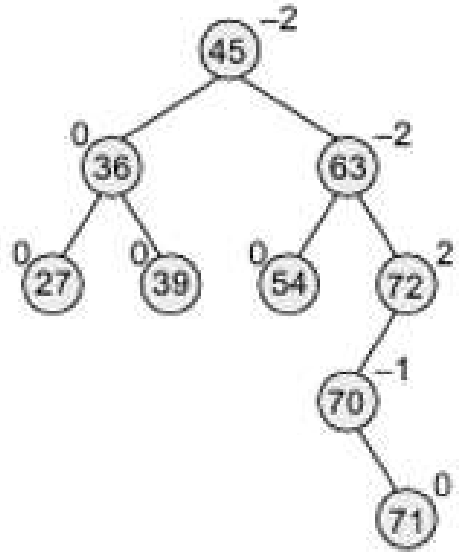
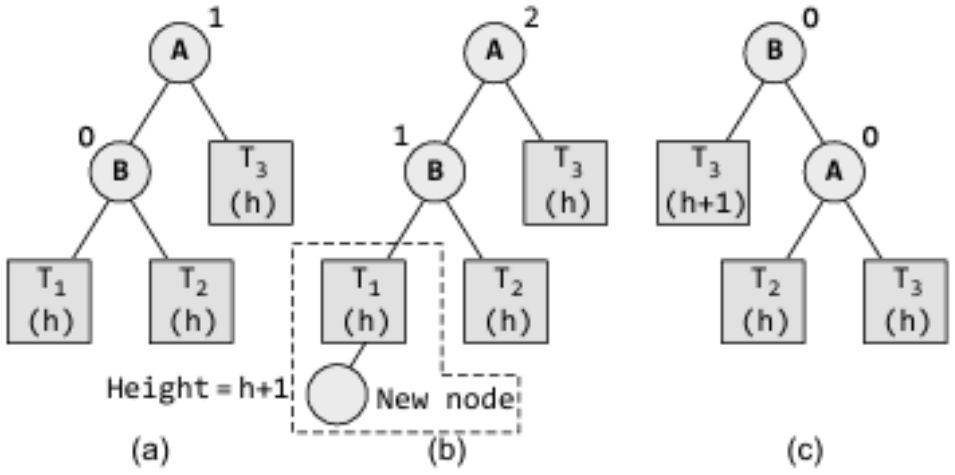


Figure AVL tree after inserting a node with the value 71

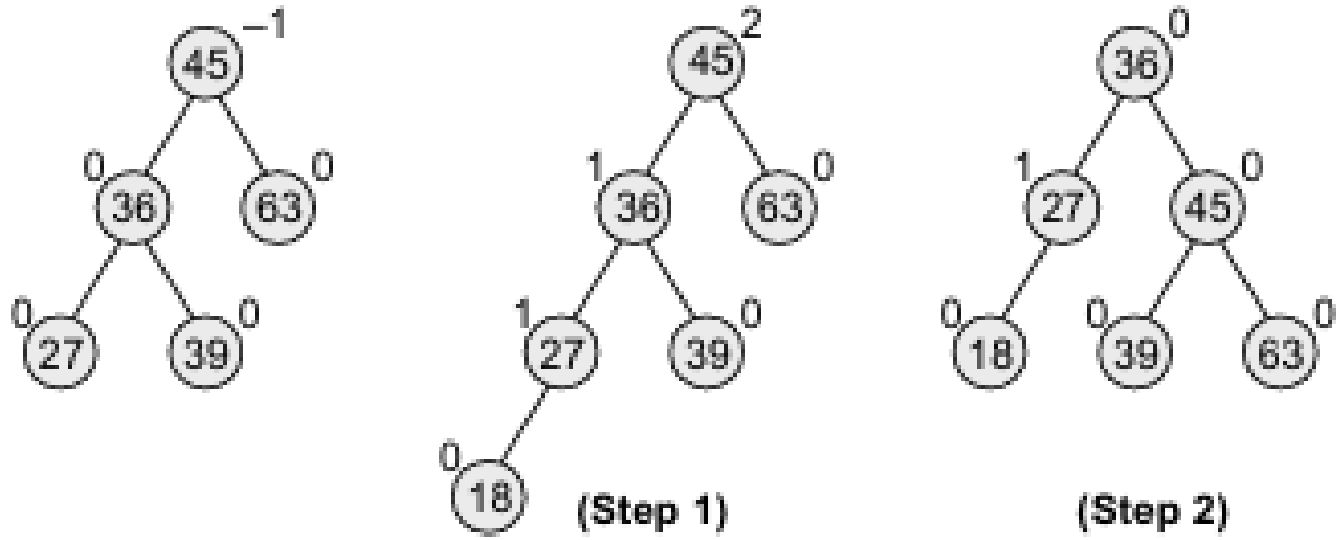
Rotation required

- There are four types of rotations:
  - LL rotation
  - RR rotation
  - LR rotation
  - RL rotation

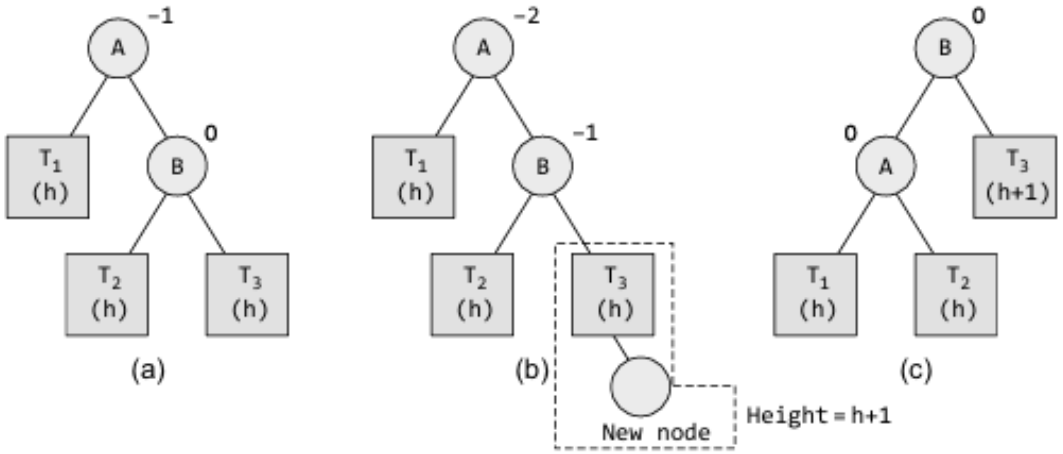
LL Rotation

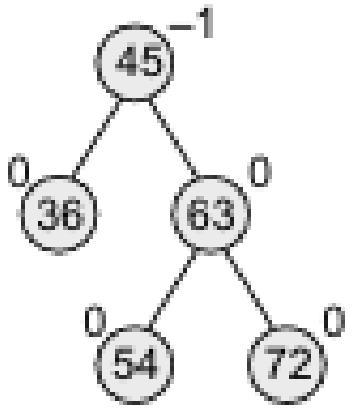


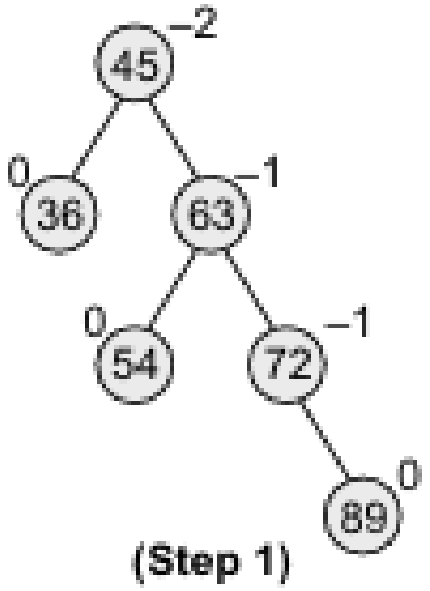
Consider the AVL tree given in Fig. and insert 18 into it

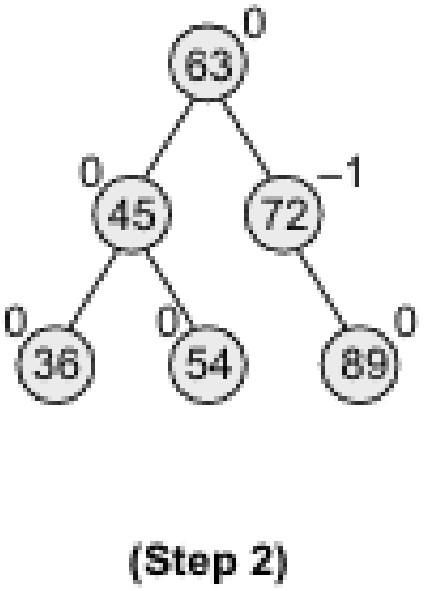


RR Rotation

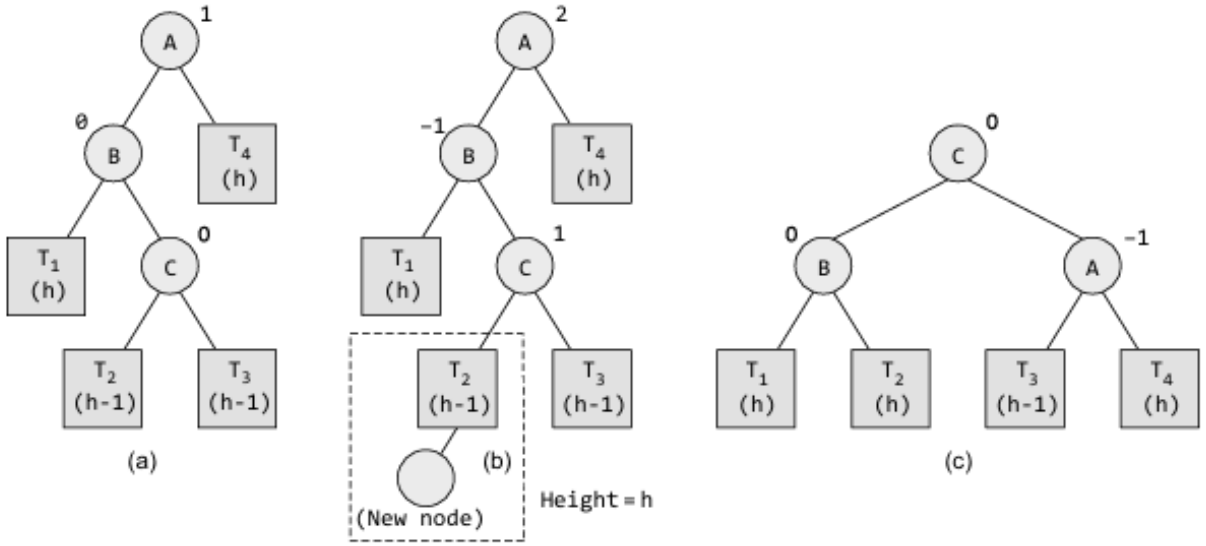

A
−1
T1
(h)
B
0
T2
(h)
T3
(h)
(a)
A
−2
T1
(h)
B
−1
T2
(h)
T3
(h)
New node
Height = h+1
(b)
B
0
A
0
T3
(h+1)
T1
(h)
T2
(h)
(c)


45
−1
36
0
63
0
54
0
72
0

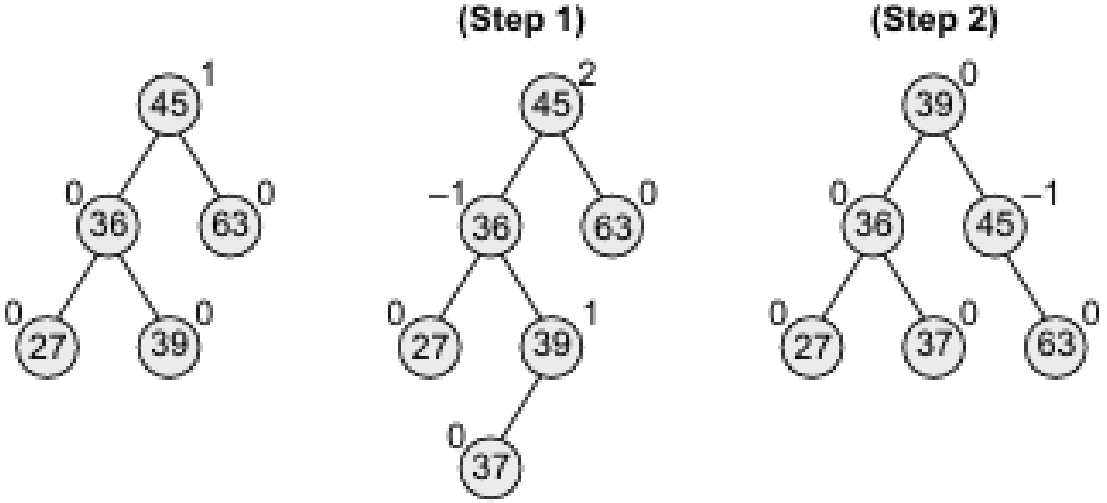

45
−2
36
0
63
−1
54
0
72
−1
89
0
(Step 1)


63
0
45
0
72
−1
36
0
54
0
89
0
(Step 2)

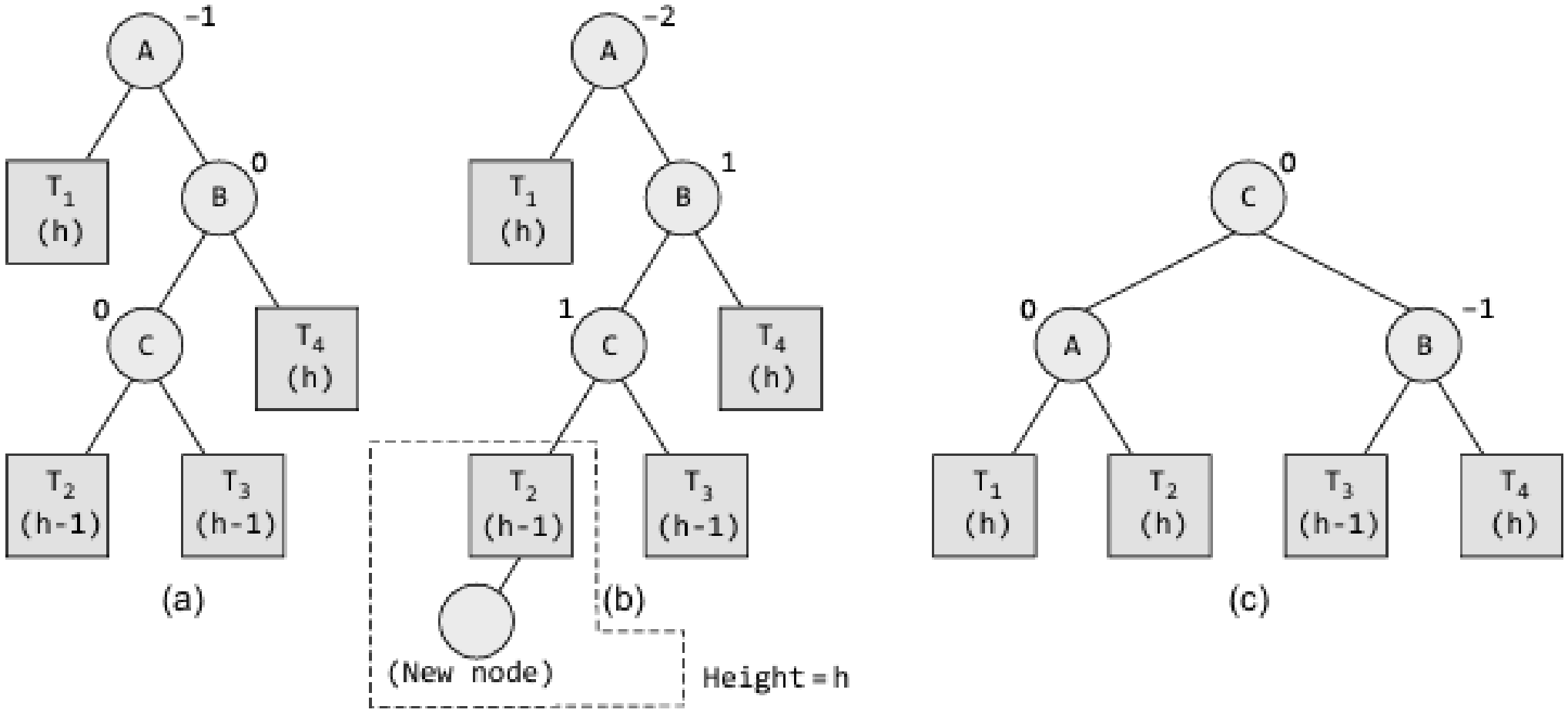
LR and RL Rotations



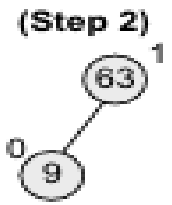
Insert 37

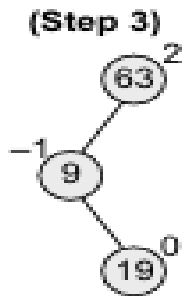


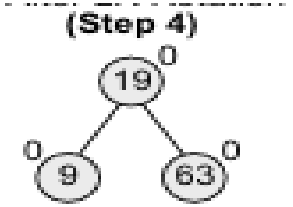
RL rotation in an AVL tree

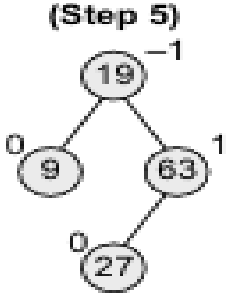


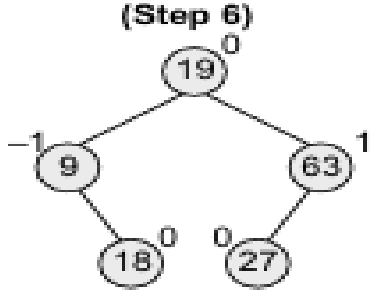
Construct an AVL tree by inserting the following elements in the given order. 63, 9, 19, 27, 18, 108, 99, 81

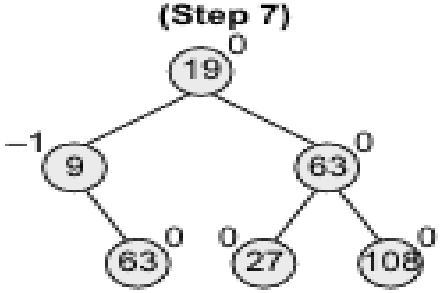


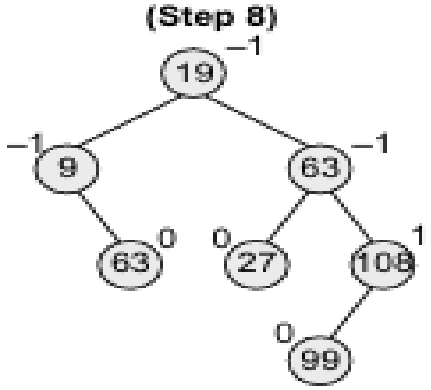


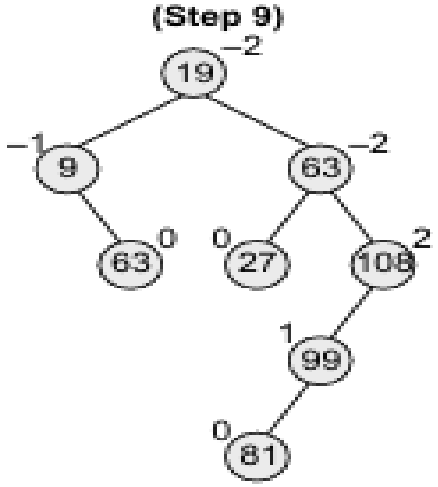


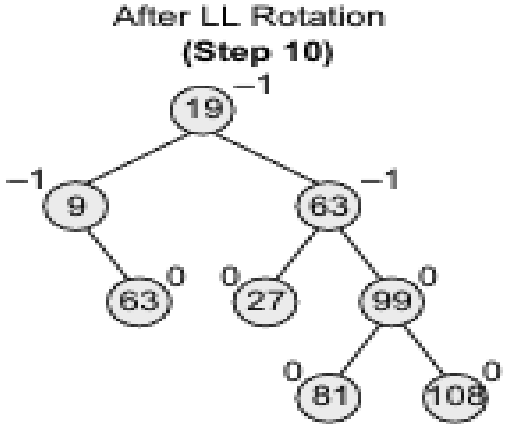






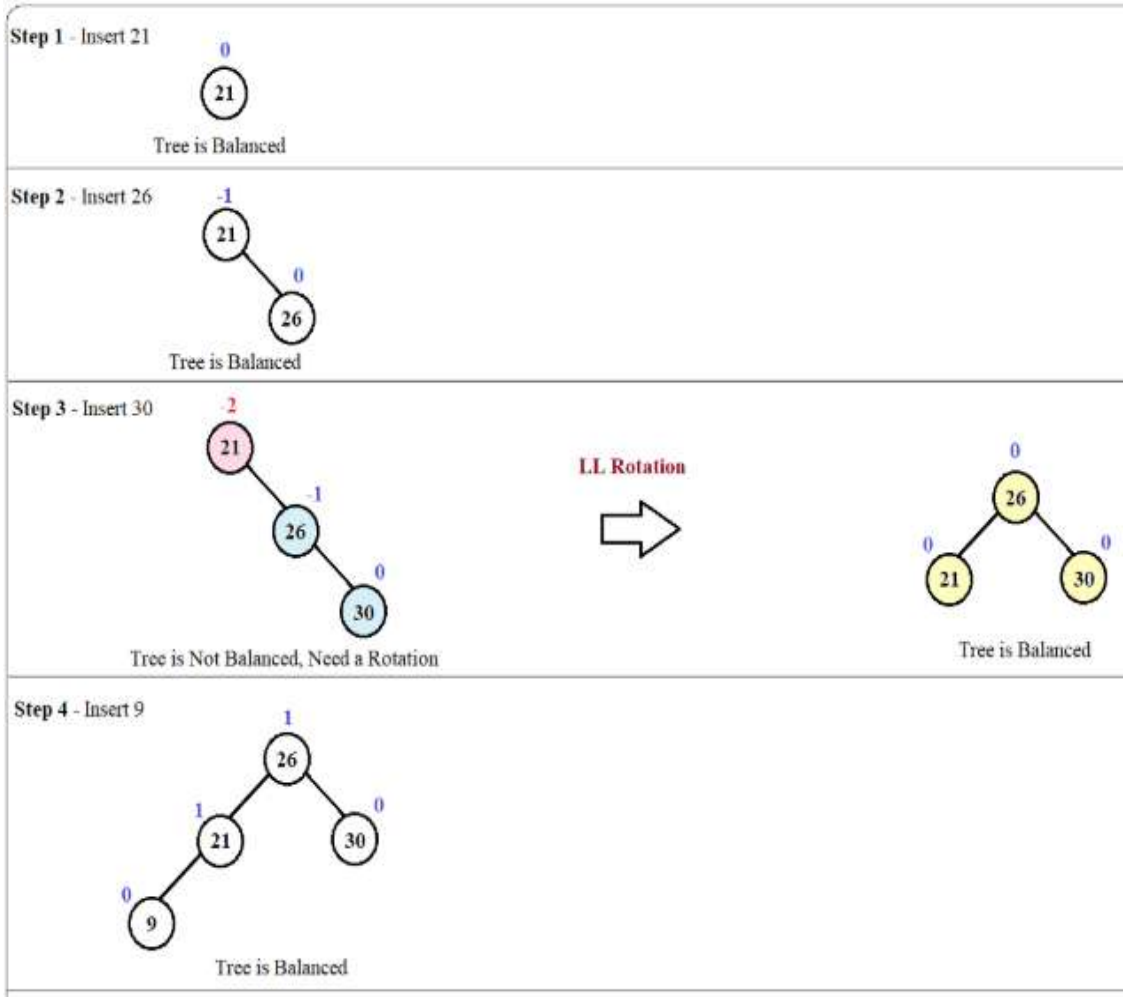


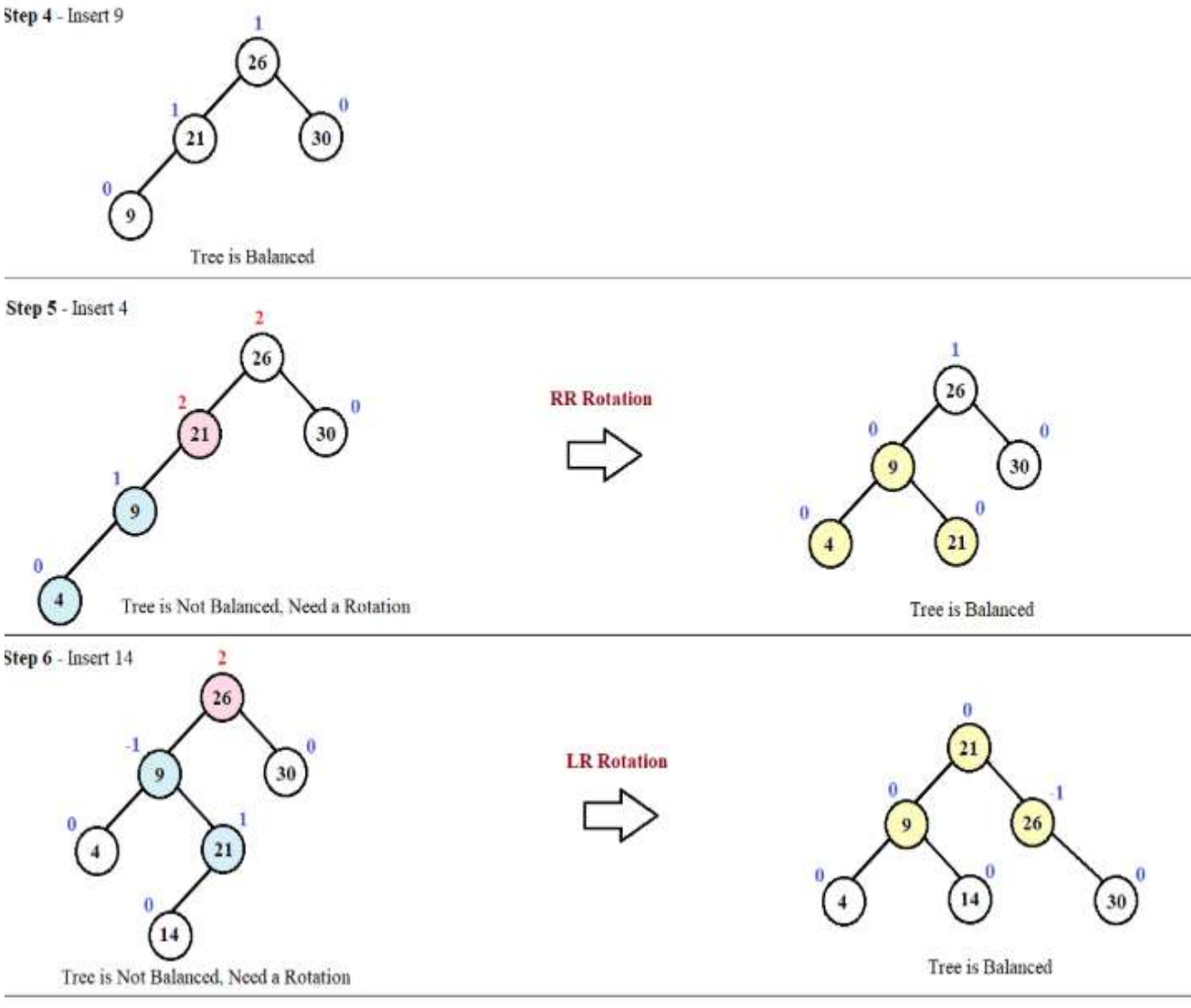




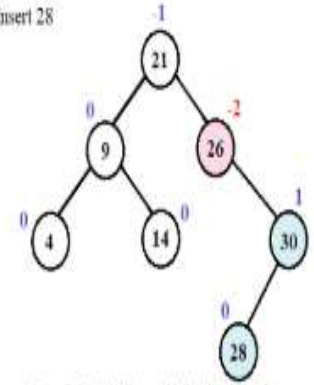
Construction of the AVL Tree for the given Sequence

21, 26, 30, 9, 4, 14, 28, 18,15,10, 2, 3, 7

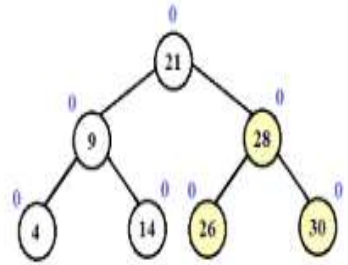




Step 7 - Insert 28



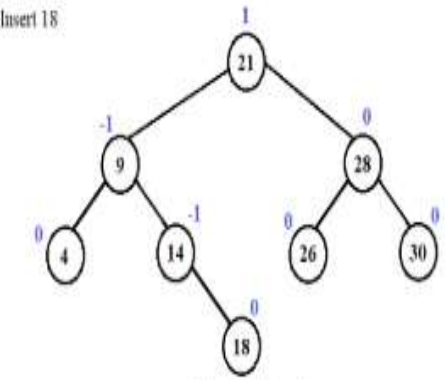
RL Rotation



Tree is Not Balanced, Need a Rotation

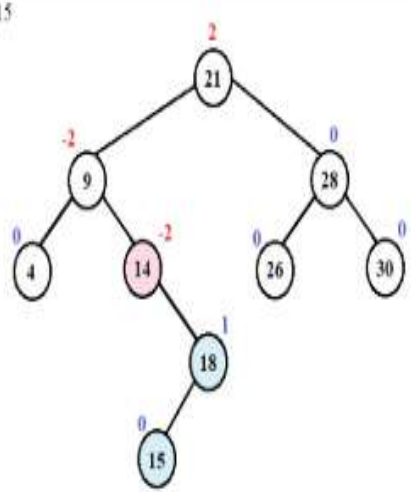
Tree is Balanced

Step 8 - Insert 18

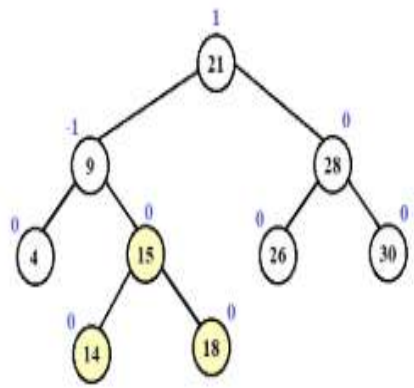


Tree is Balanced

Step 9 - Insert 15



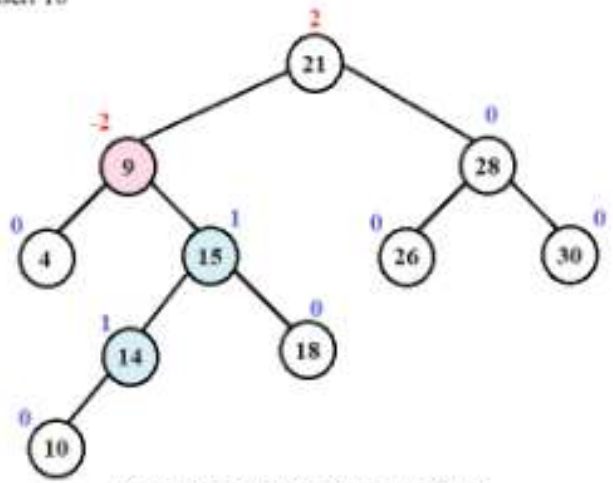
RL Rotation



Tree is Not Balanced, Need a Rotation

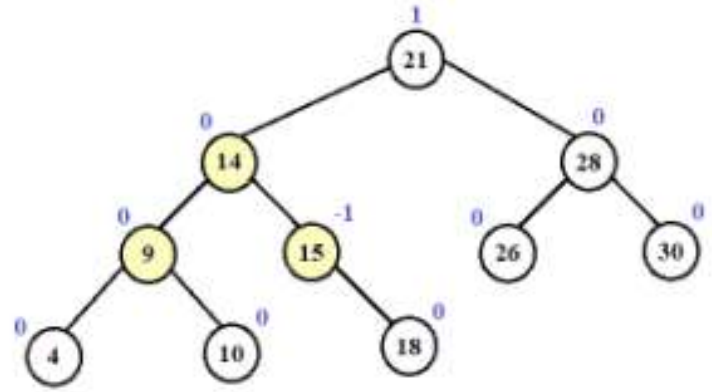
Tree is Balanced

Step 10 - Insert 10



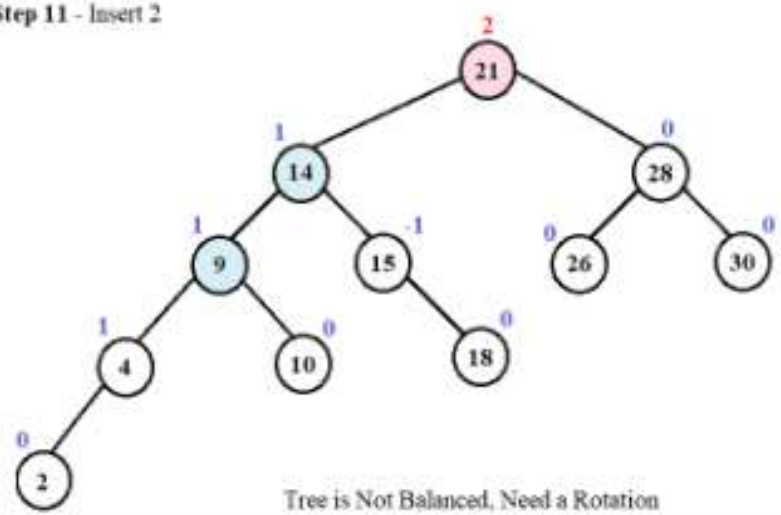
Tree is Not Balanced, Need a Rotation

RL Rotation



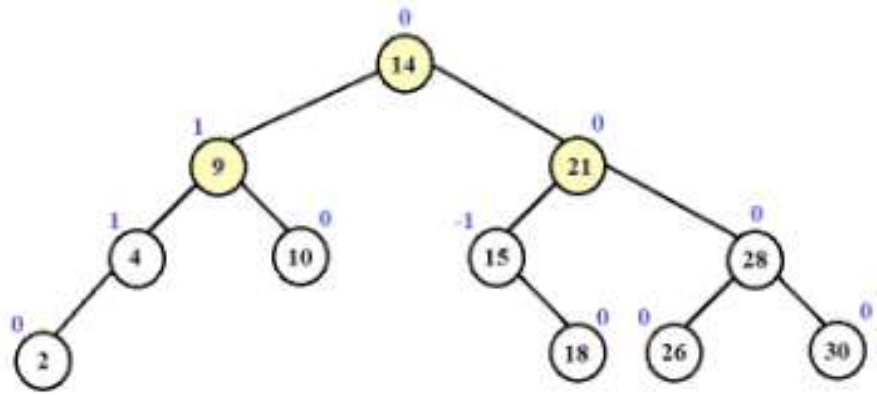
Tree is Balanced

Step 11 - Insert 2



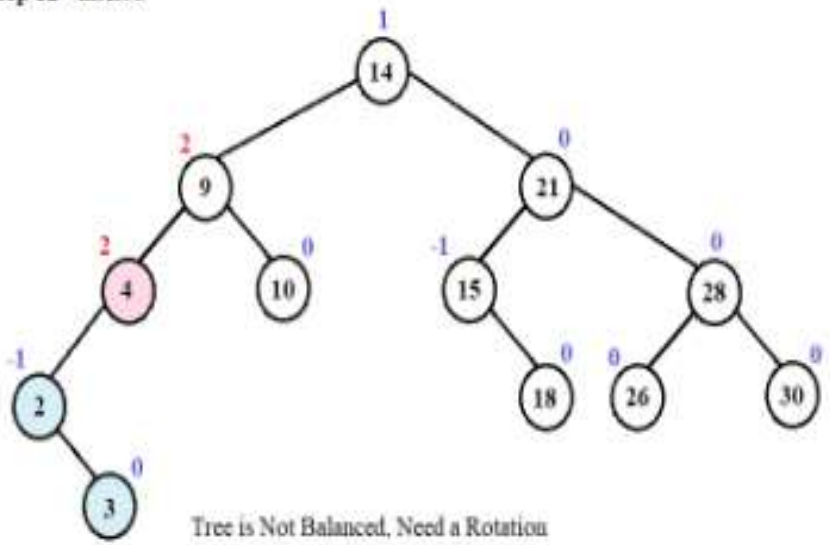
Tree is Not Balanced, Need a Rotation

RR Rotation



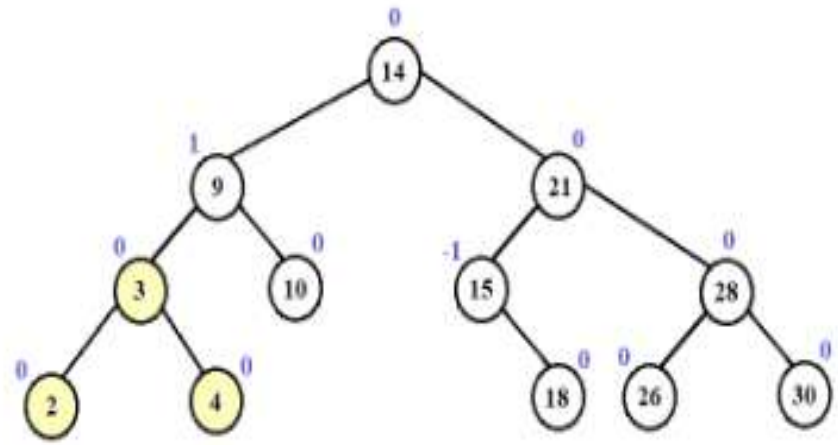
Tree is Balanced

Step 12 - Insert 3



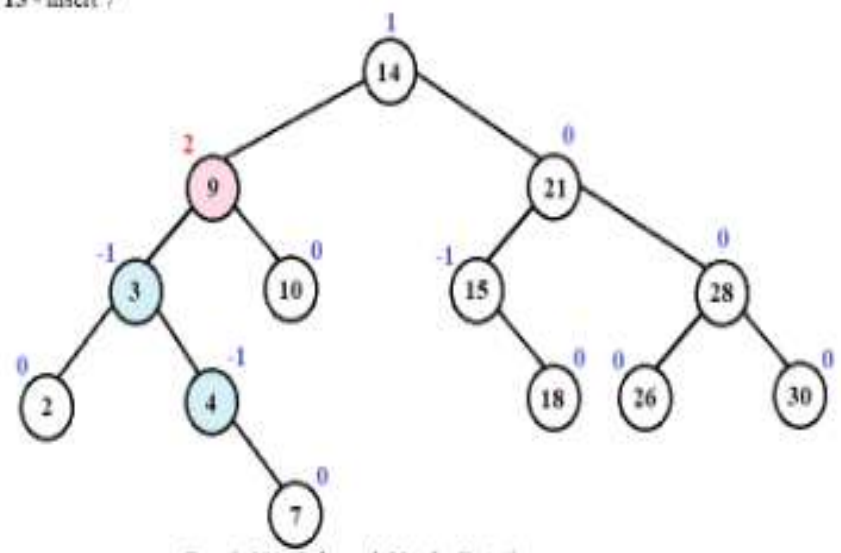
Tree is Not Balanced, Need a Rotation

LR Rotation



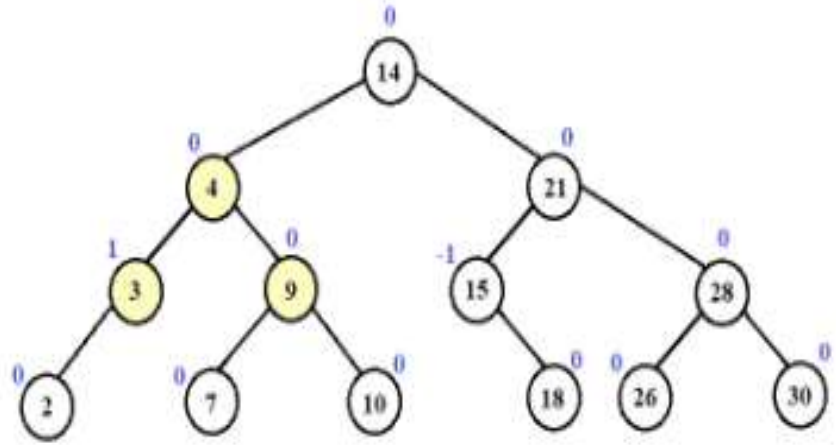
Tree is Balanced

Step 13 - Insert 7



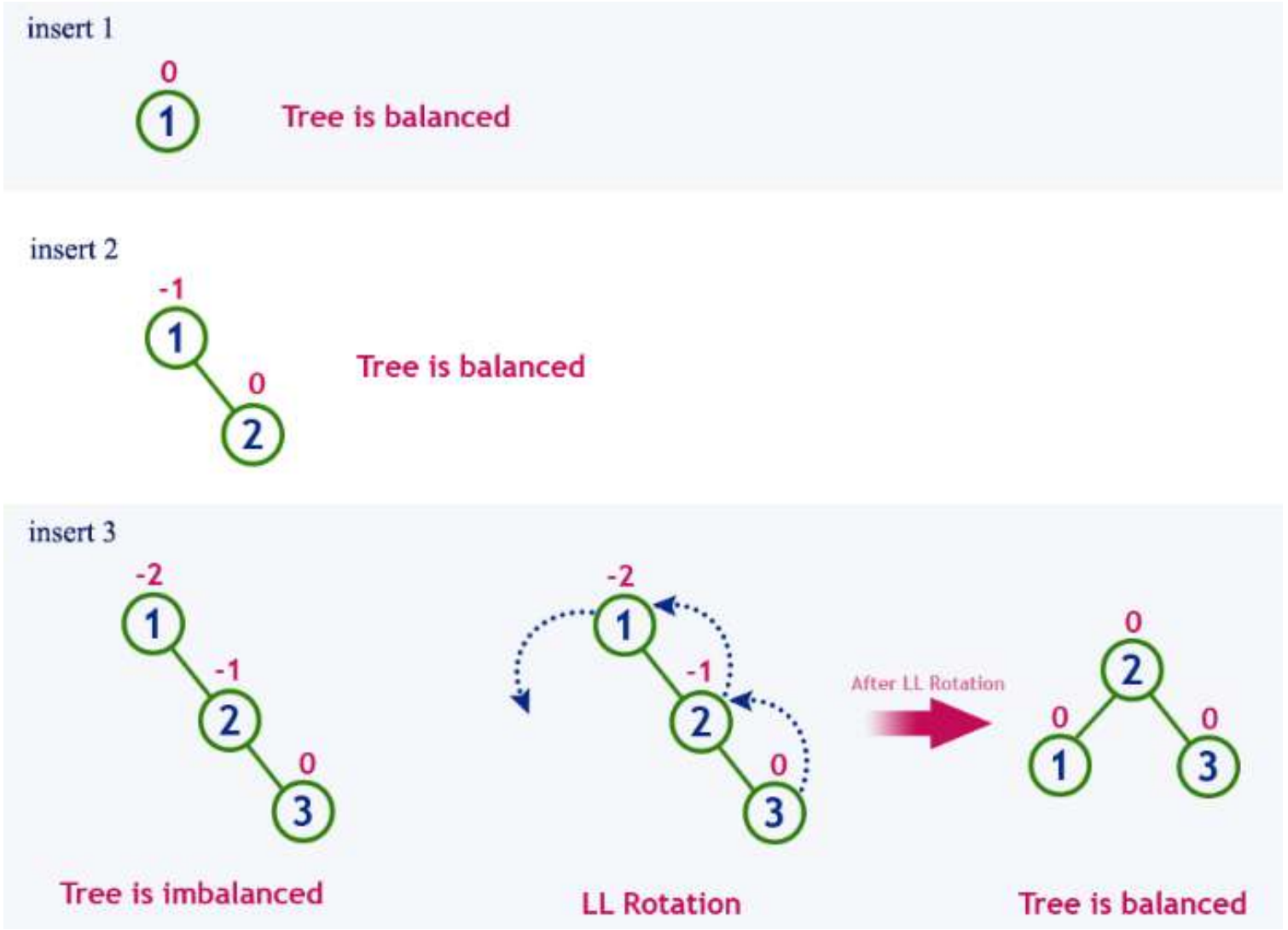
Tree is Not Balanced, Need a Rotation

LR Rotation

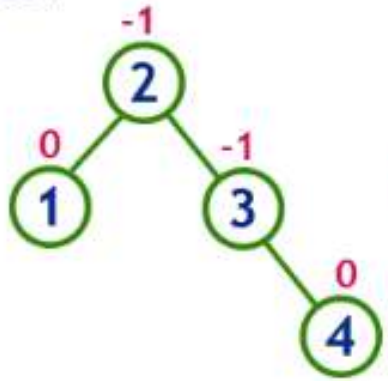


Tree is Balanced

Insert 1 to 8 in AVL Tree

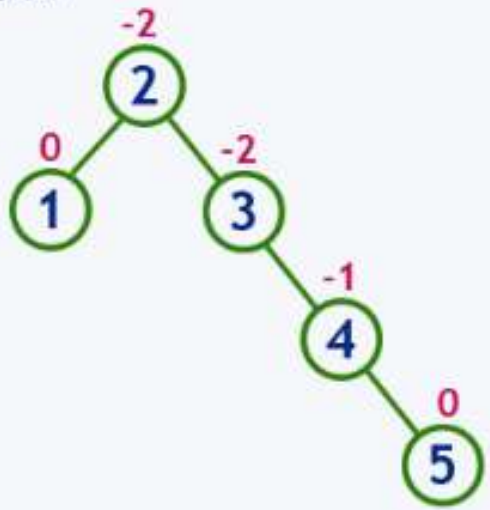


insert 4

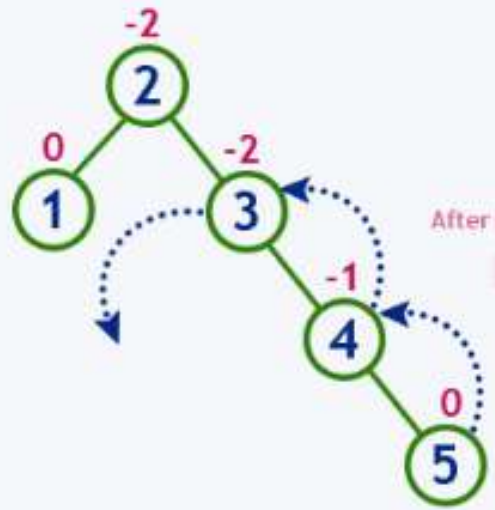


Tree is balanced

insert 5

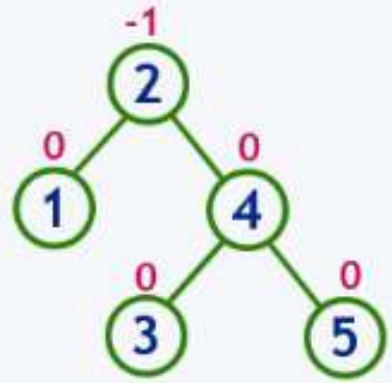


Tree is imbalanced



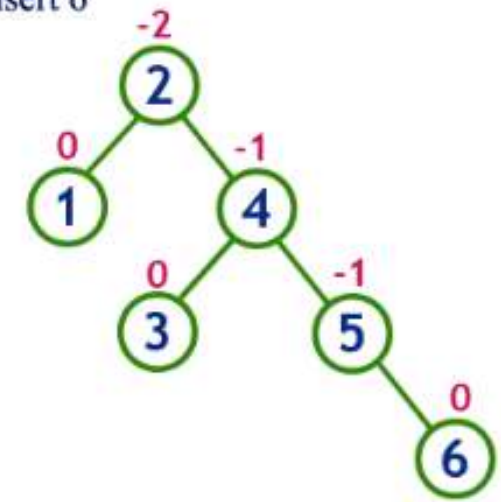
LL Rotation at 3

After LL Rotation at 3

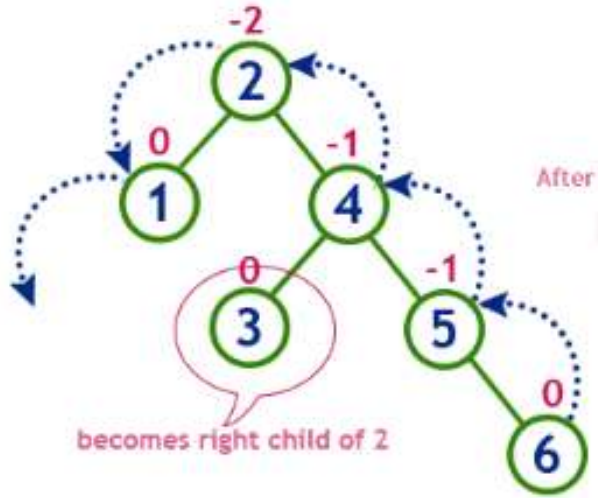


Tree is balanced

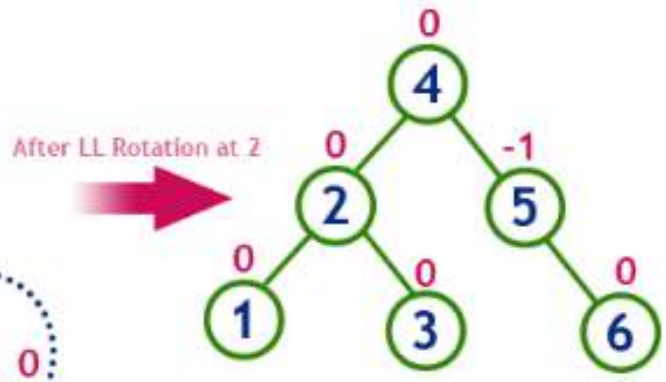
insert 6



Tree is imbalanced

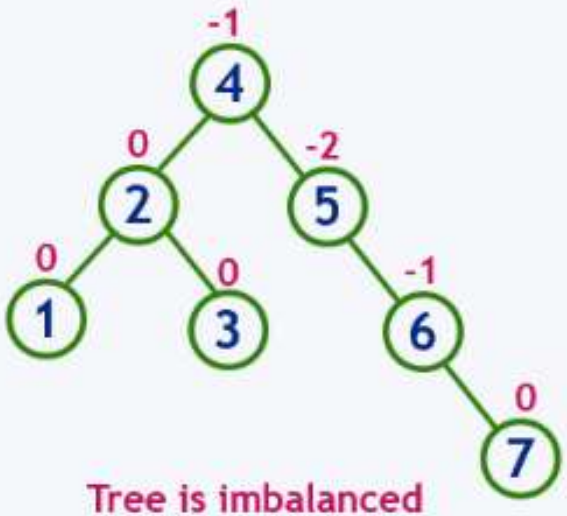


LL Rotation at 2

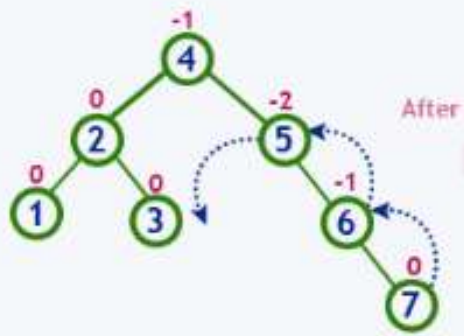


Tree is balanced

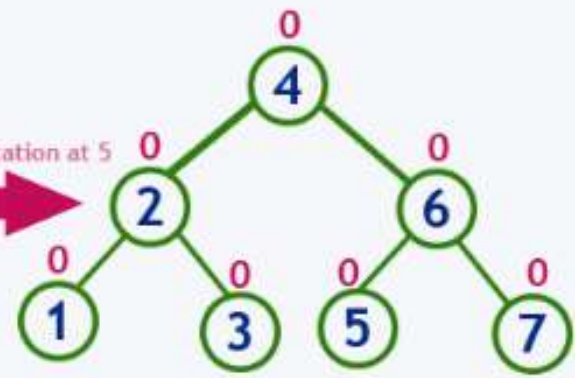
insert 7



Tree is imbalanced

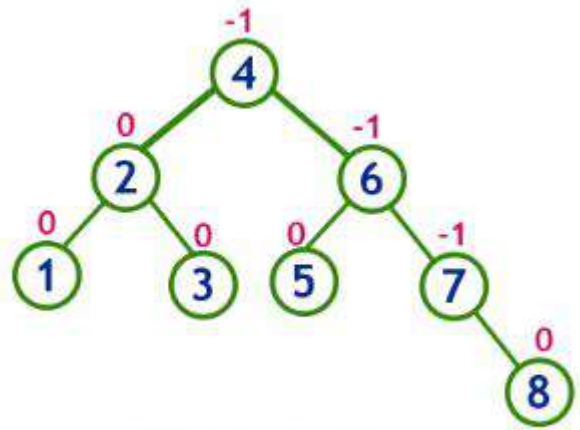


LL Rotation at 5



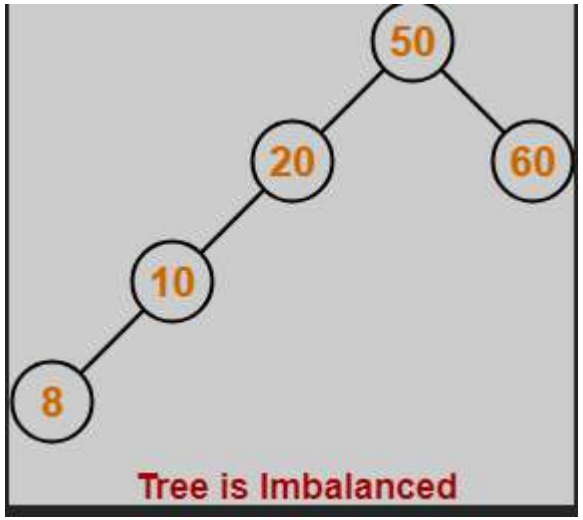
Tree is balanced

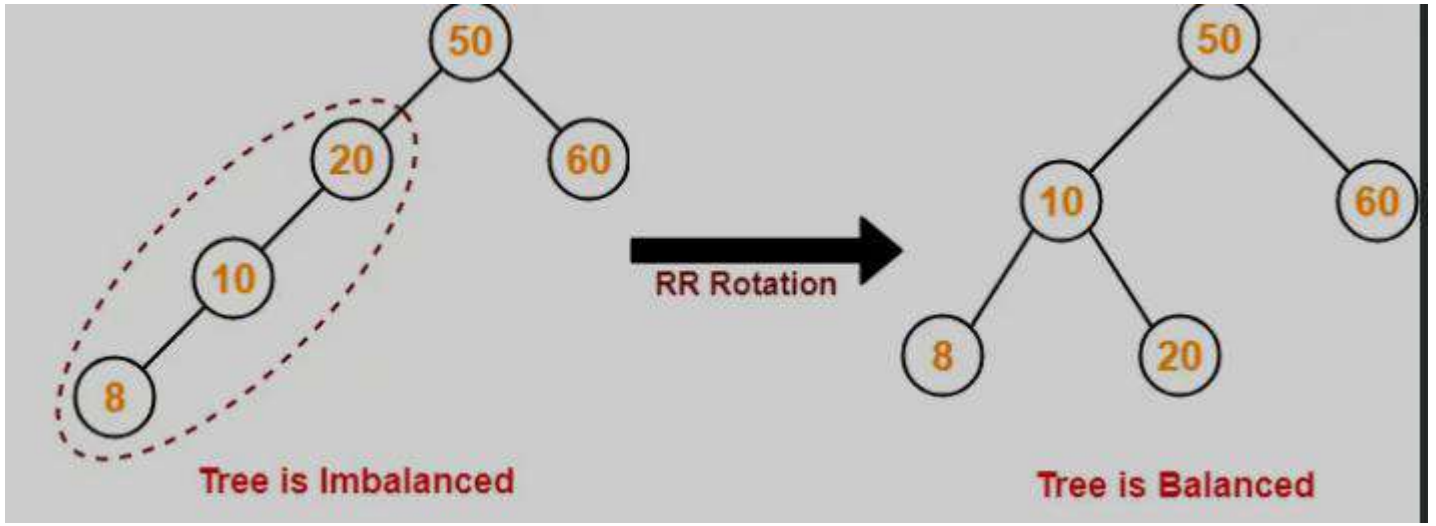
insert 8

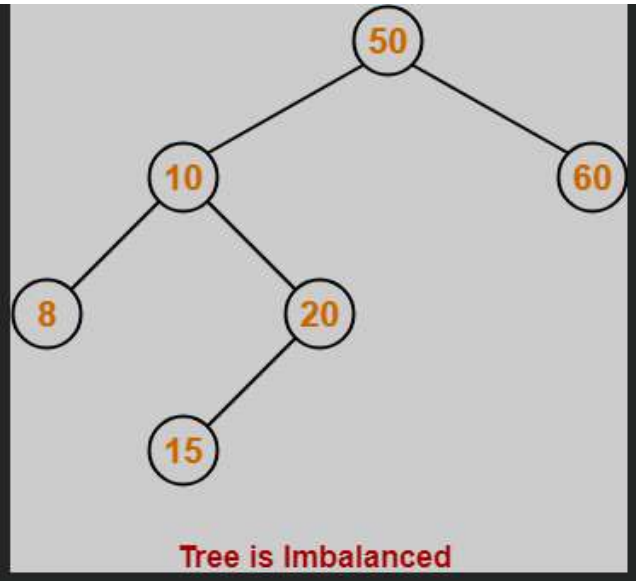


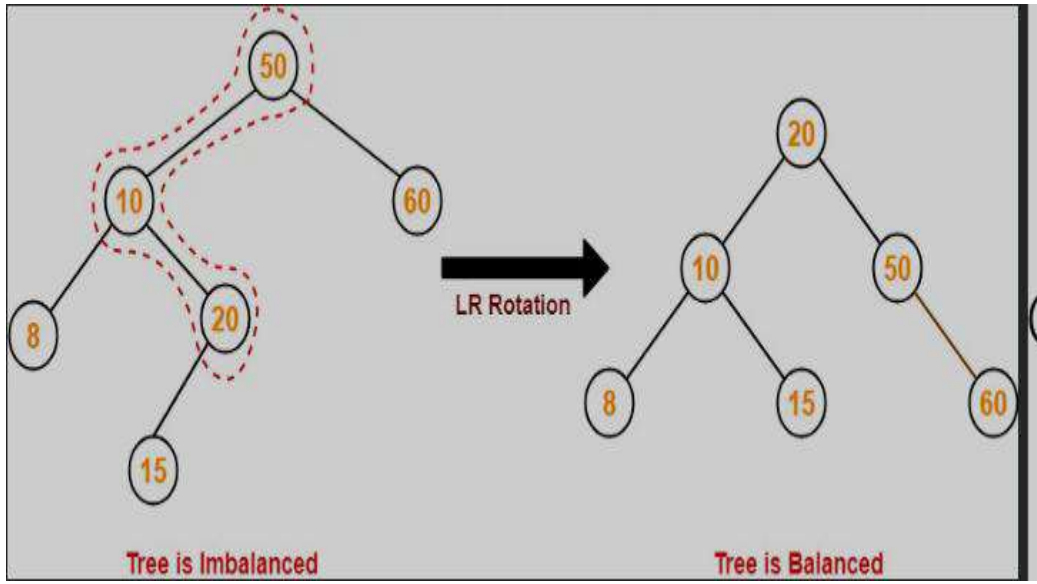
Tree is balanced

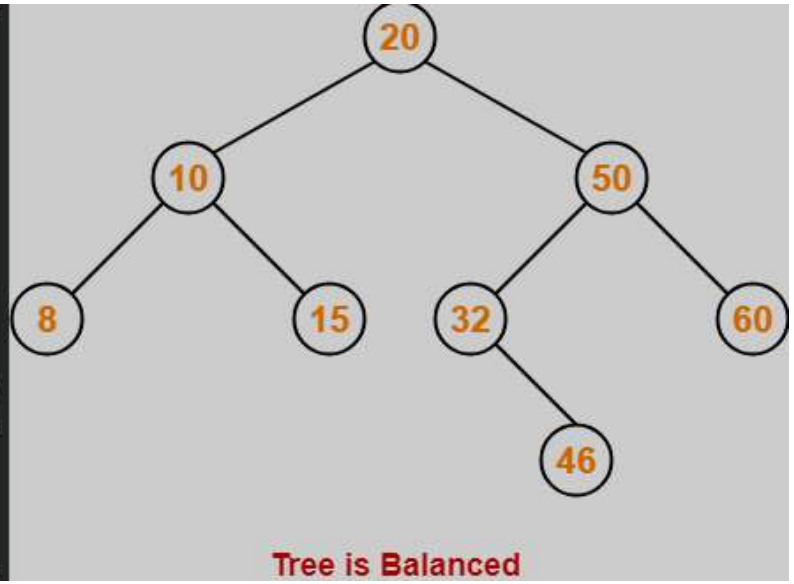
- Construct AVL Tree for the following sequence of numbers-
- 50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
-

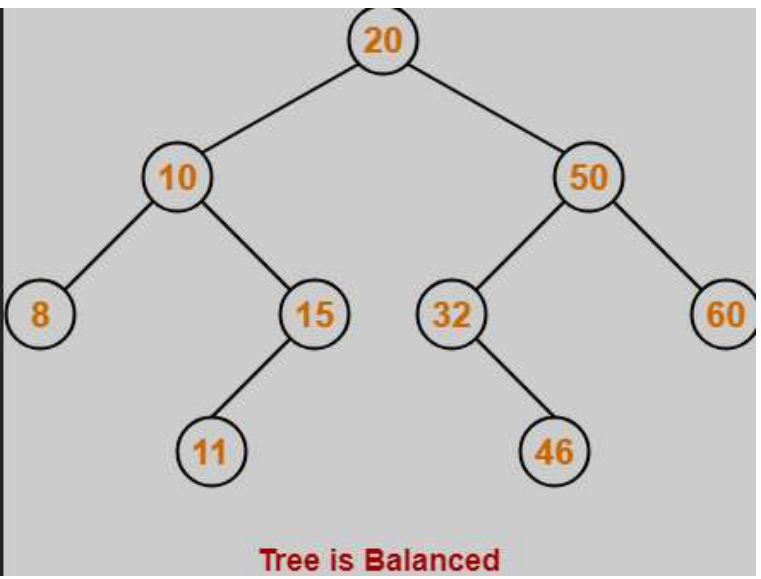

50
20
60
10
8
Tree is Imbalanced


50
20
60
10
8
Tree is Imbalanced
RR Rotation
50
10
60
8
20
Tree is Balanced

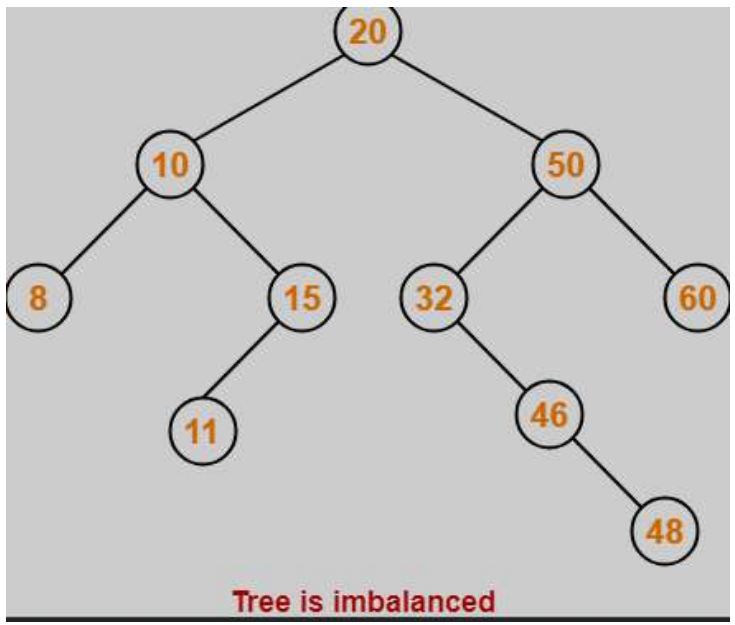

50
10
60
8
20
15
Tree is Imbalanced


50
10
60
8
20
15
Tree is Imbalanced
LR Rotation
20
10
50
8
15
60
Tree is Balanced

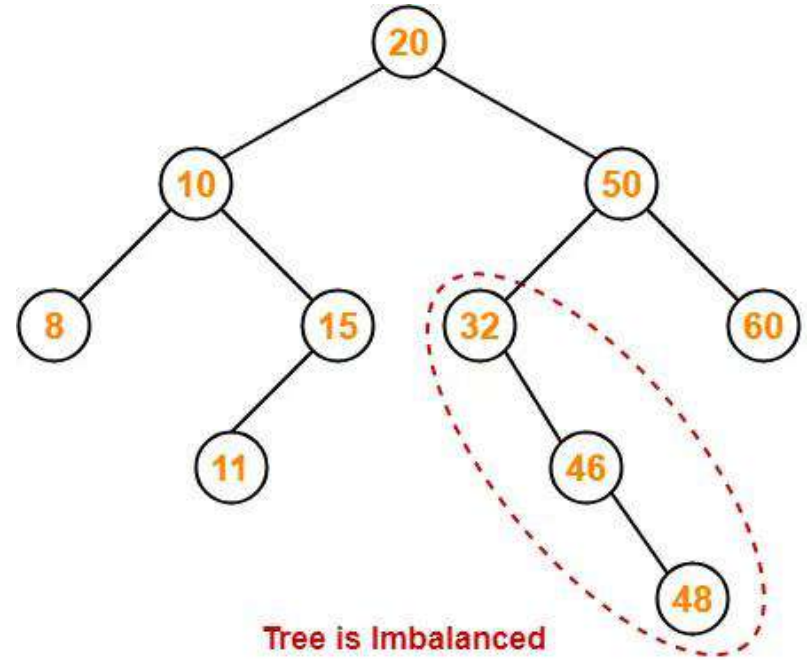

20
10
50
8
15
32
60
46
Tree is Balanced



Tree is Balanced

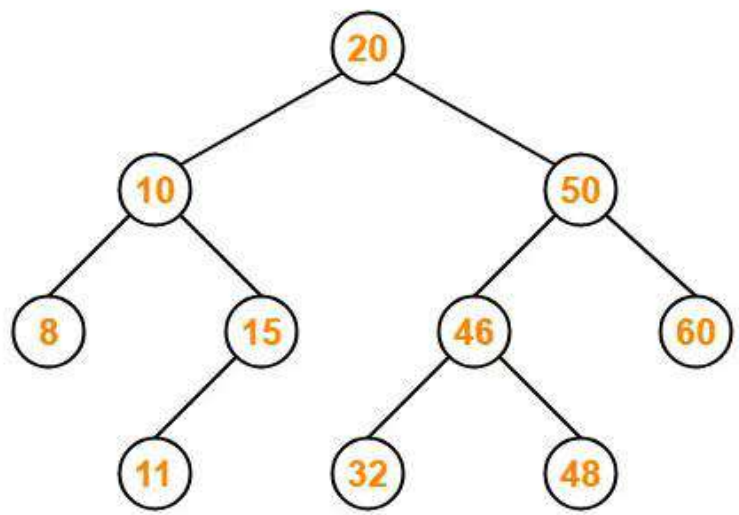


Tree is imbalanced

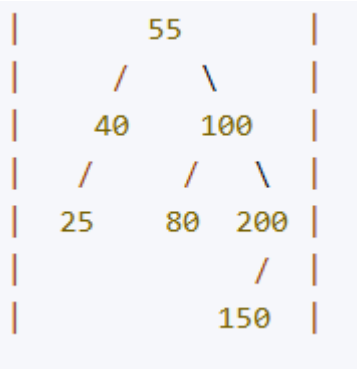


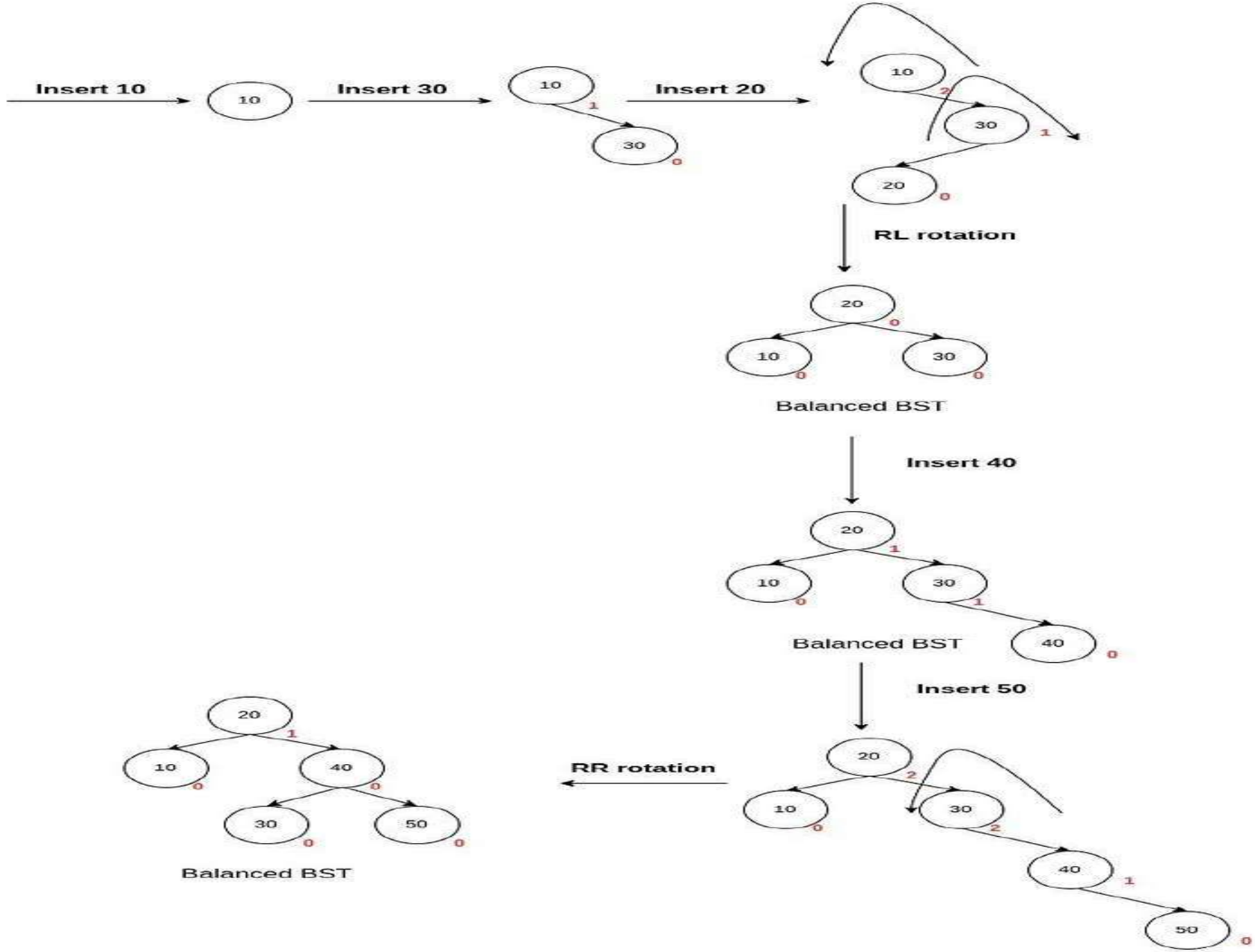
Tree is Imbalanced

LL Rotation



- 55, 40, 25,100,80,200,150




Insert 10
10
Insert 30
10
1
30
0
Insert 20
10
2
30
1
20
0
RL rotation
20
0
10
0
30
0
Balanced BST
Insert 40
20
1
10
0
30
1
40
0
Balanced BST
Insert 50
20
2
10
0
30
2
40
1
50
0
RR rotation
20
1
10
0
40
0
30
0
50
0
Balanced BST

Deletion in AVL tree

Case 1: When the node to be deleted is a leaf node: Then the node can be deleted directly as it does not have any children, i.e. it is a leaf node.

Case 2: When the node to be deleted has one subtree: In this case, the node to be deleted is replaced by its only child.

Case 3: When the node to be deleted has both the subtrees: This is a tricky case, as we need to decide with which node the node to be deleted has to be replaced. We can choose either a node from the left or right subtree.

- **Deleting a Leaf Node (No Rotations Needed)**

```
    20
   /  \
  10   30
```

Delete Node: `10`

Resulting Tree:

Code

```
    20
     \
      30
```

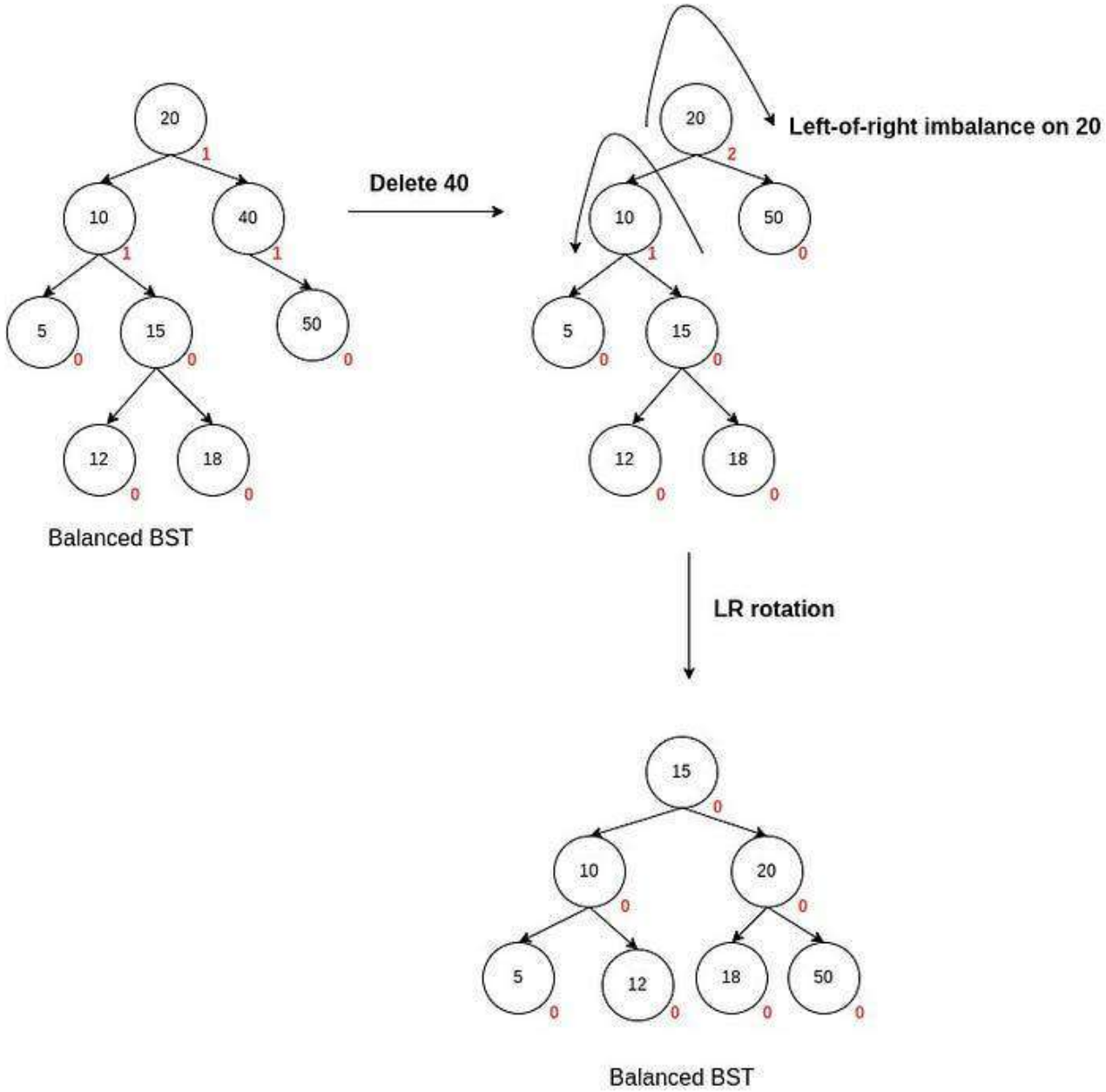
Deleting a Node with One Child (No Rotations Needed)

```
    20
   /  \
  10   30
         \
          40
```

Delete Node: `30`

- **Resulting Tree:**

```
    20
   /  \
  10   40
```


20
1
10
40
1
1
5
15
50
0
0
0
12
18
0
0
Balanced BST
Delete 40
20
Left-of-right imbalance on 20
2
10
50
1
0
5
15
0
0
12
18
0
0
LR rotation
15
0
10
20
0
0
5
12
18
50
0
0
0
0
Balanced BST

Time Complexity

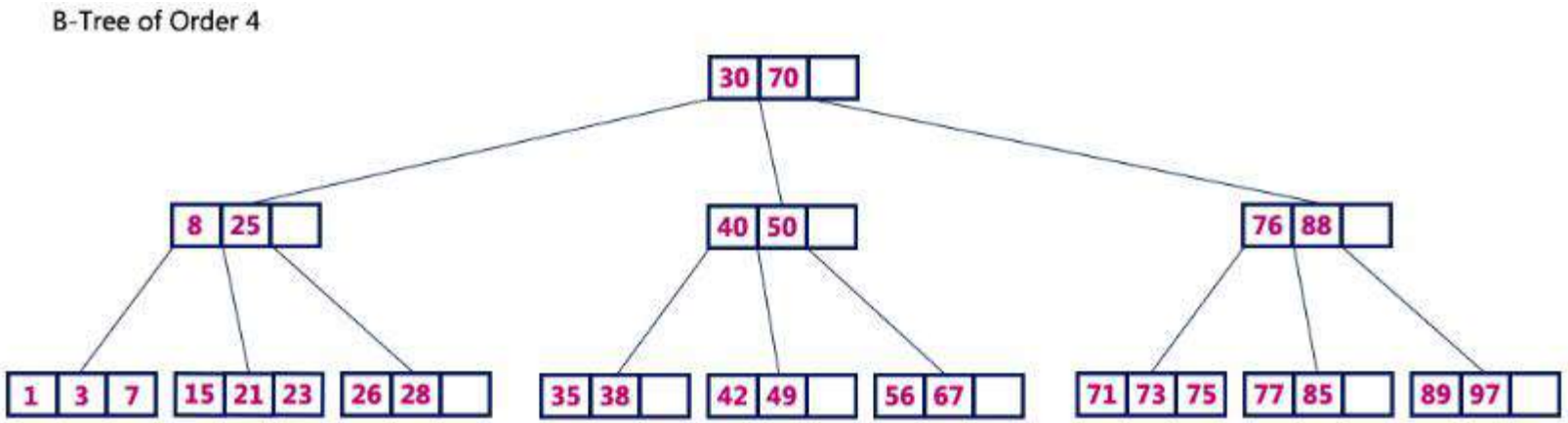
	AVL Tree		Binary Search Tree	
	Average	**Worst**	**Average**	**Worst**
Insertion	O(logn)	O(logn)	O(logn)	O(n)
Deletion	O(logn)	O(logn)	O(logn)	O(n)
Searching	O(logn)	O(logn)	O(logn)	O(n)
Traversal	O(n)	O(n)	O(n)	O(n)

B Tree

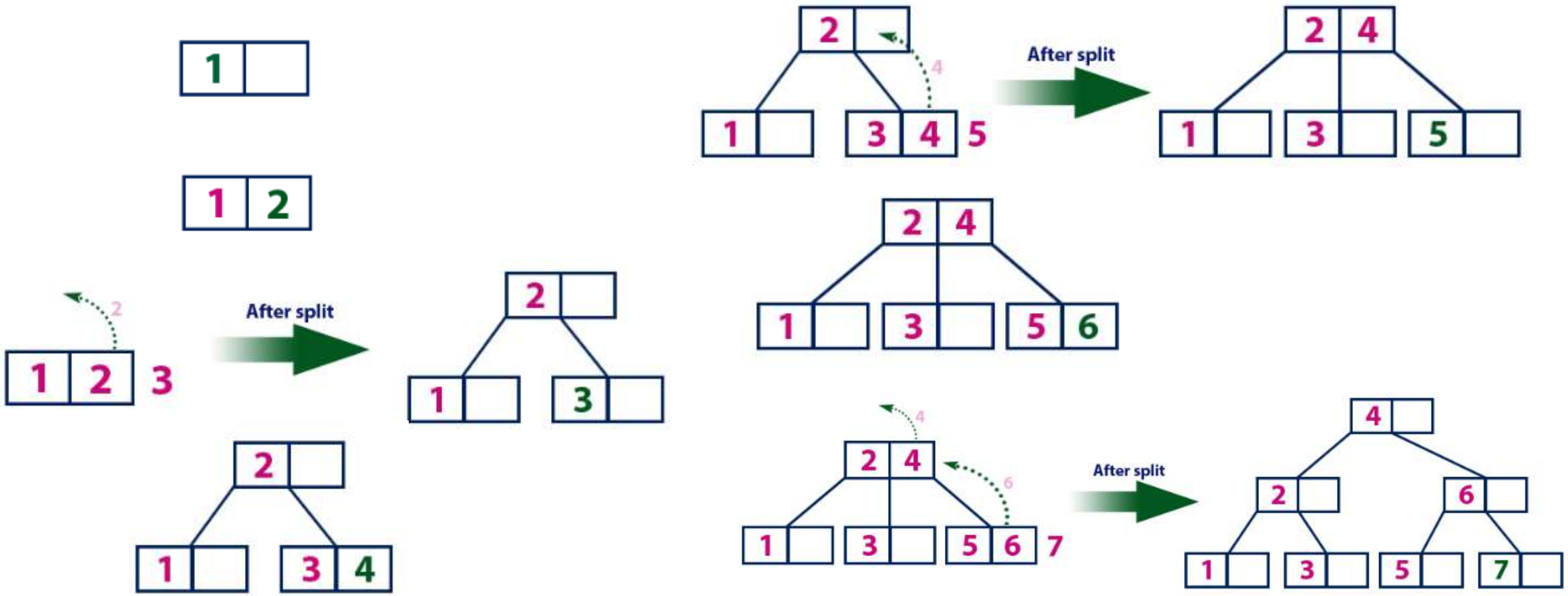
- In search trees like binary search tree, AVL Tree, Red-Black tree, etc., every node contains only one value (key) and a maximum of two children. But there is a special type of search tree called B-Tree in which a node contains more than one value (key) and more than two children.
- B-Tree was developed in the year 1972 by **Bayer and McCreight** with the name ***Height Balanced m-way Search Tree***. Later it was named as B-Tree.
- **B-Tree is a self-balanced search tree in which every node contains multiple keys and has more than two children.**

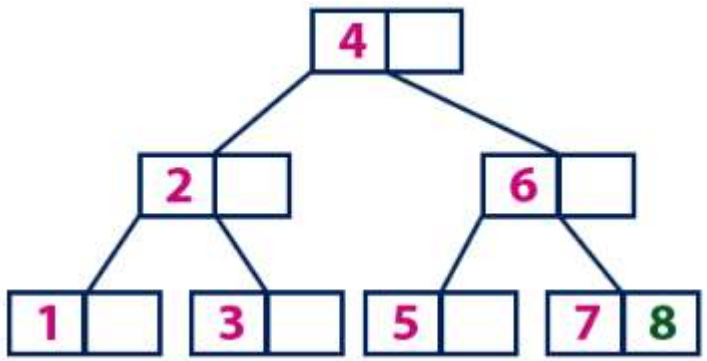
- **B-Tree of Order m** has the following properties...
  - **Property #1** - All **leaf nodes** must be **at same level**.
  - **Property #2** - All nodes except root must have at least **[m/2]-1** keys and maximum of **m-1** keys.
  - **Property #3** - All non leaf nodes except root (i.e. all internal nodes) must have at least **m/2** children.
  - **Property #4** - If the root node is a non leaf node, then it must have **atleast 2** children.
  - **Property #5** - A non leaf node with **n-1** keys must have **n** number of children.
  - **Property #6** - All the **key values in a node** must be in **Ascending Order**.

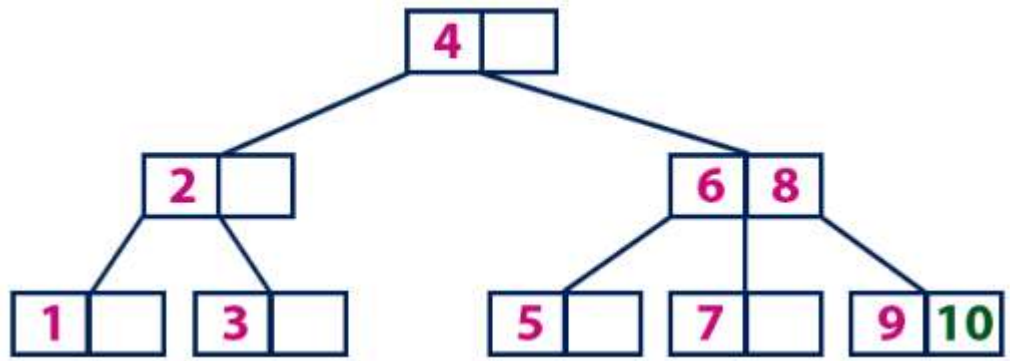
- B-Tree of Order 4 contains a maximum of 3 key values in a node and maximum of 4 children for a node.

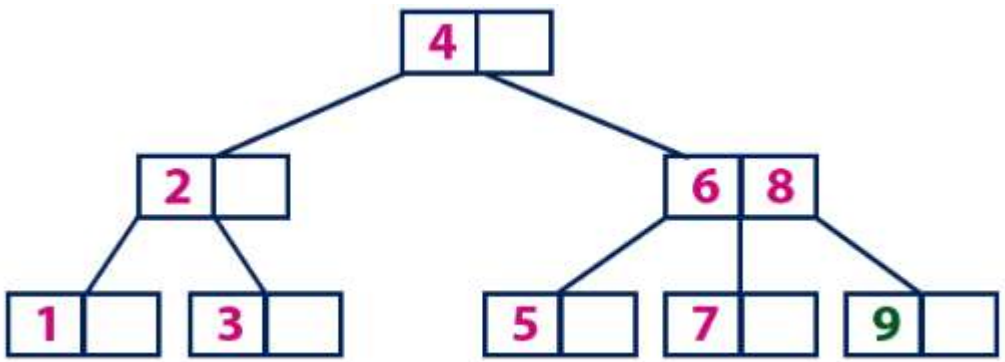


- Construct BTree of order 3 from elements 1 to 10



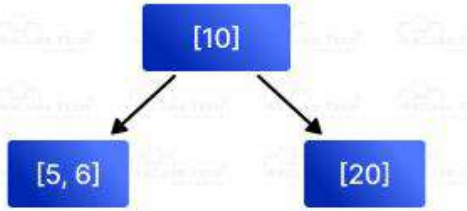

4
2
6
1
3
5
7
8


4
2
6
8
1
3
5
7
9
10

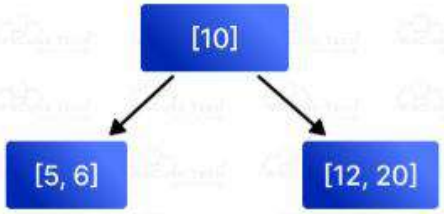

4
2
6
8
1
3
5
7
9

- Consider a B-Tree of order 3 (m=3), which means each node can have at most 2 keys. Insert **the keys 10, 20, 5, 6, and 12:**
-

- **Insert 6 (node splits):**



- **Insert 12:**



- Construct a B-Tree of order 3 for following data: 50, 30, 21, 90, 10, 13, 20, 70, 25, 92, 80.

B-Tree – order 3.

So No. of keys for holding – 2

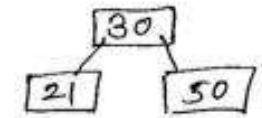
Overflow condition when keys in node is 3

No. of childrens – 3 for each node possible.

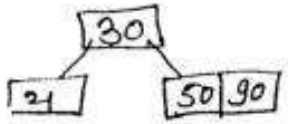
1) 50 — 50

2) 30 | 50 — 30

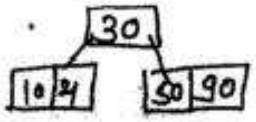
3) 21 | 30 | 50 — 21 — overflow so split node



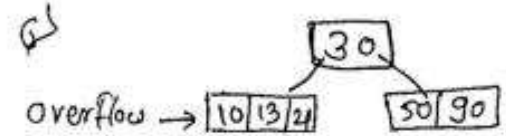
4) 90



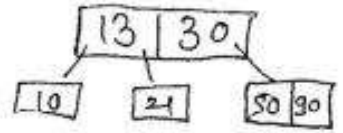
5) 10



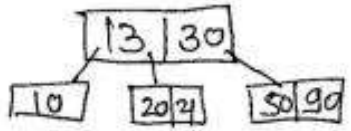
6)



13

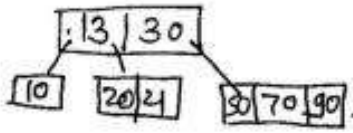


7)

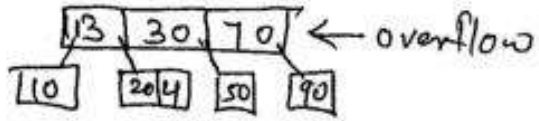


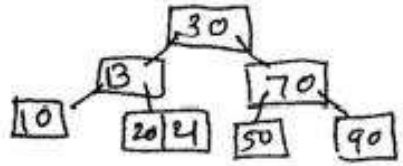
20

8)



70





Construct a B-Tree of order 5 for following data: 78, 21, 14, 11, 97, 85, 74, 63, 45, 42, 57, 20, 16, 19, 32, 30, 31.

- [11, 14, 21, 78,95]→split

[21]
 ├── [11, 14]
 └── [78, 97]

- Insert 85,74

[21]
 ├── [11, 14]
 └── [74, 78, 85, 97]

Insert 63→split required

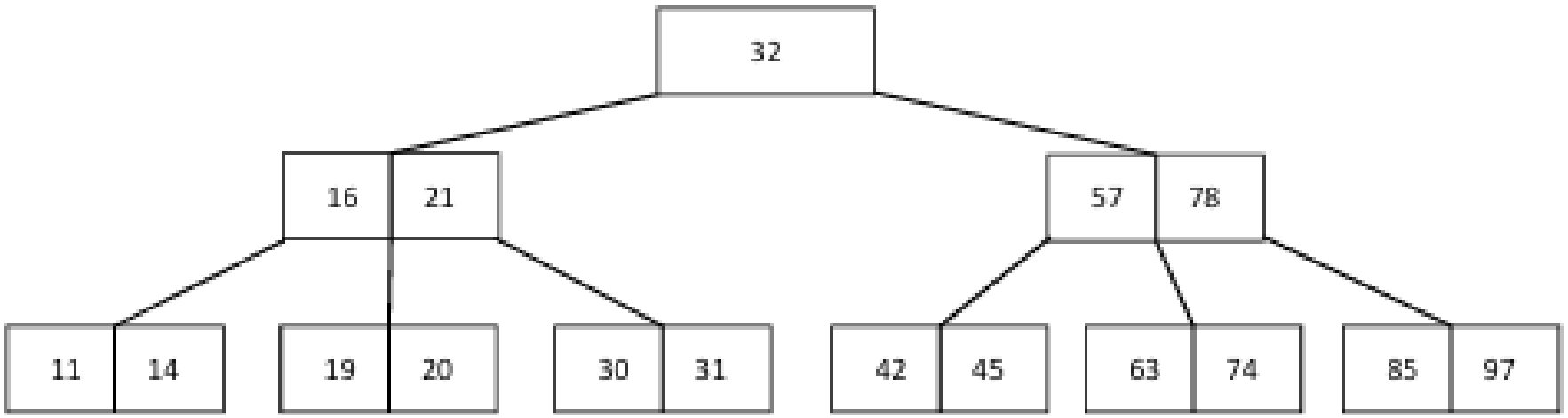
[21, 78]
 ├── [11, 14]
 ├── [63, 74]
 └── [85, 97]

Insert 45,42,57

[21, 57, 78]
 ├── [11, 14]
 ├── [42, 45]
 ├── [63, 74]
 └── [85, 97]

Insert 20,16,19

[11,14,16,19,20]



B – Tree.

- Construct a B^+-Tree of order 5 for following data: 30, 31, 23, 32, 22, 28, 24, 29, 15, 26, 27, 34, 39, 36.