Data Structure and Algorithm chapter 06

Presented by: Er. Aruna Chhatkuli

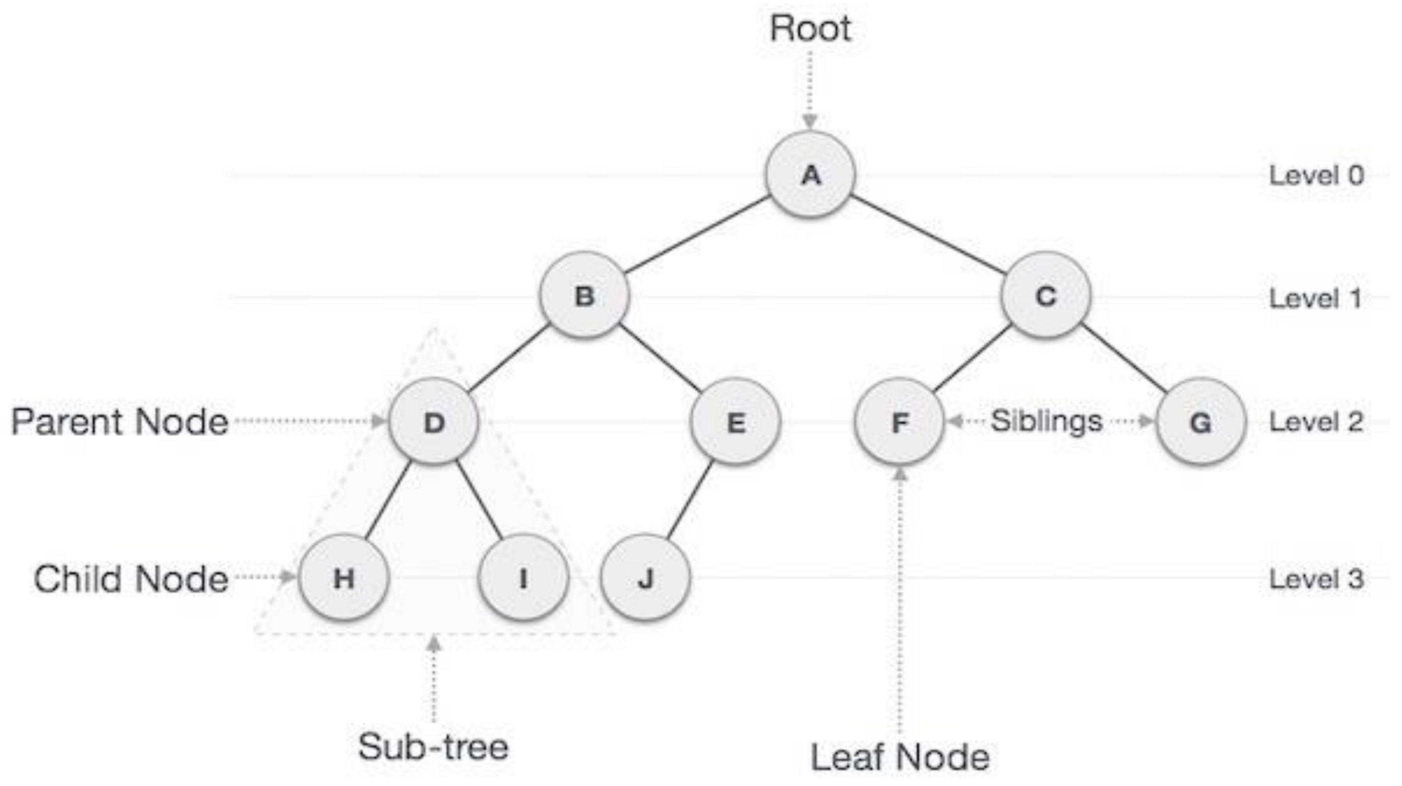
Nepal College of Information Technology, Balkumari, Lalitpur

Tree

- A tree is as non linear data structure in which **items are arranged in a sorted sequence,** It is used to represent hierarchical relationship existing among several data items.
- Each node of a tree may or may not pointing more than one node.
- It is a finite set of one or more data items (nodes) such that there is a special data item called the root.
- It’s remaining data items are portioned into numbers of mutually exclusive subsets, each of which is itself a tree and they are called subtrees.

Tree terminology:-

Tree



Tree terminology:-

Root:- The first node in a hierarchical arrangement of data items is root.

Node:- Each data item in a tree is called a node.

Degree of a node:- It is the no. of subtrees of a node in a given tree from fig. Here,

Degree of node A = 3

Degree of node D = 2

Degree of node F = 1

Degree of node I = 3

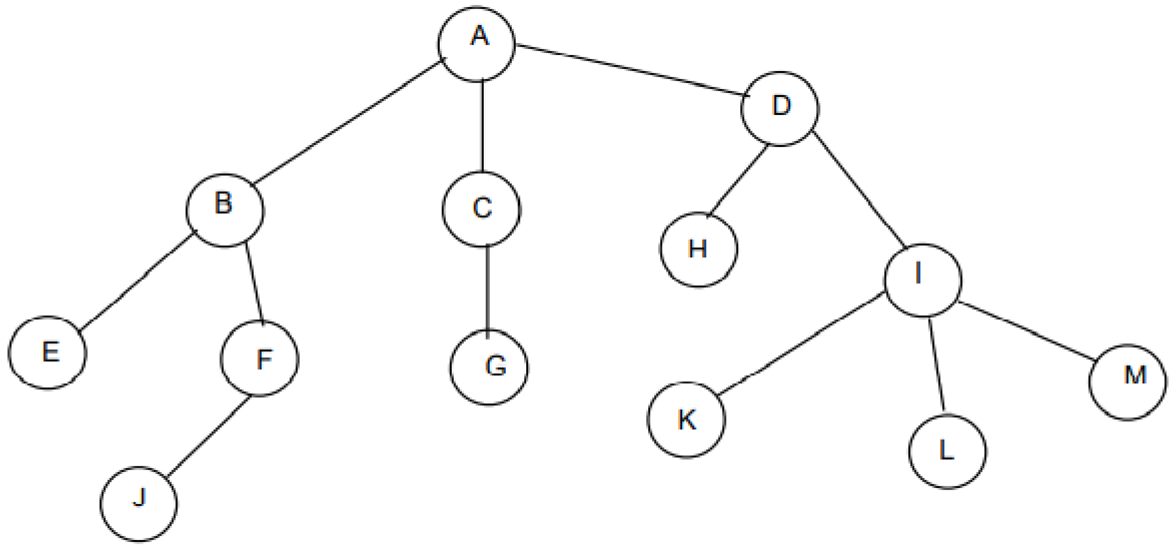


Fig. Tree

Tree terminology:-

Degree of a tree:- It is the maximum degree of nodes in a given tree here degree of tree is 3.

Terminal node:- A node with degree 0 is terminal node here, S,I,G,H,X,L,M are terminal node.

Non terminal node:- Any node except the root whose degree is not zero is call non terminal node.

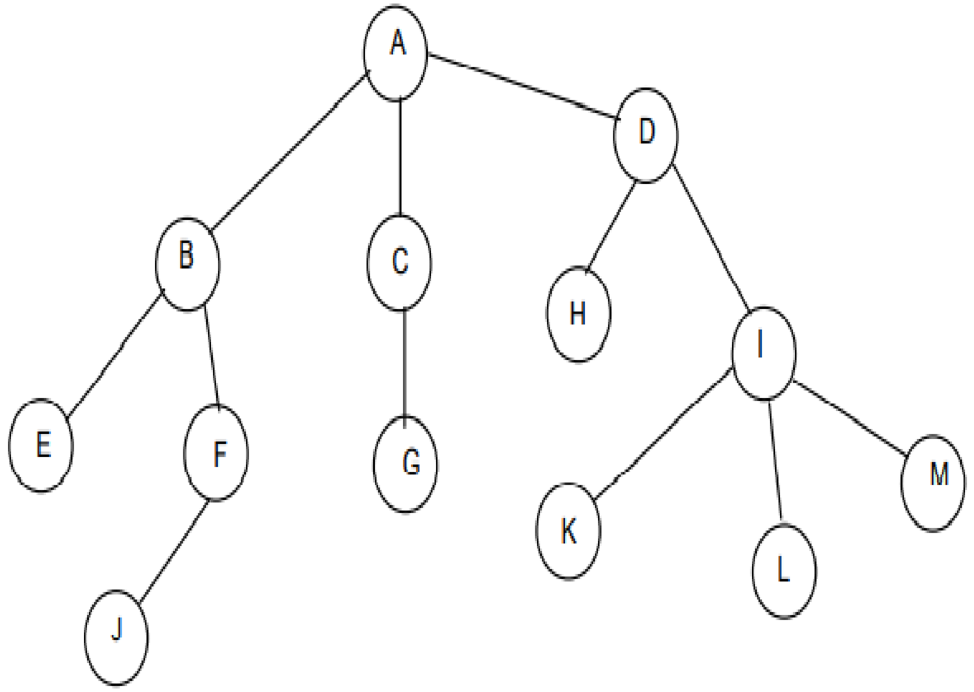


Fig. Tree

Tree terminology:-

Siblings:- The children nodes of a given parent nodes are called siblings. E and F are siblings of parent node B and K, L, M are siblings of parents node I & so on.

Level:- The entire tree structure is leveled in such a way that the root node is always at level zero. Then it's immediate children are at level 1 and there immediate children are at level 2 and so on up to the terminal nodes. Here, the level of a tree is 3.

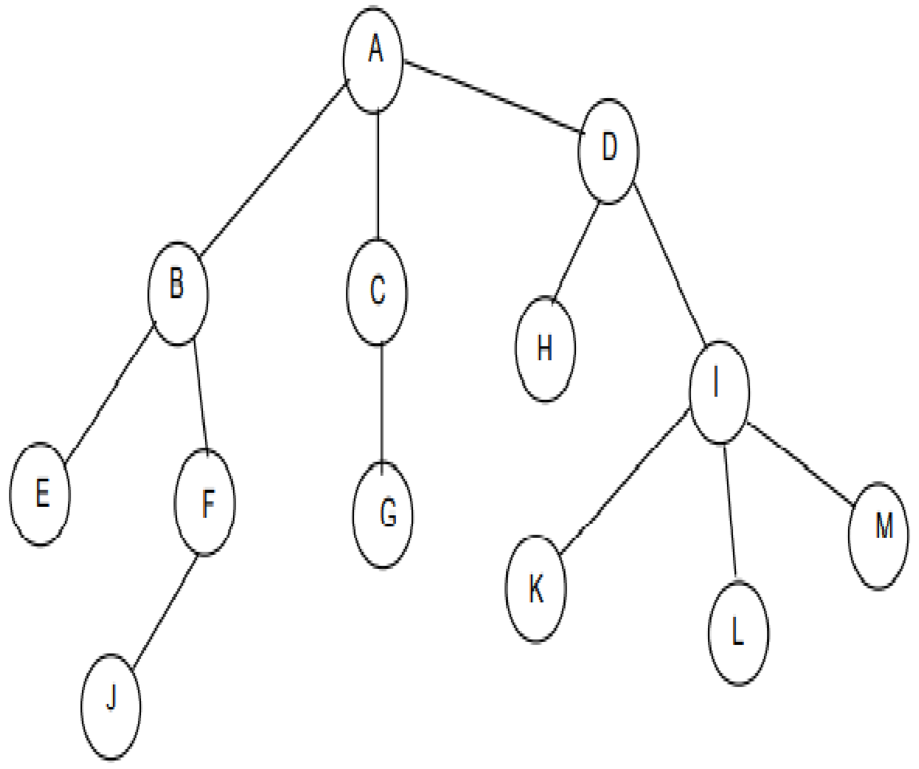


Fig. Tree

Tree terminology:-

Path:- It is a sequence of consecutive edge from the source node to the destination node.

Edge:- It is a connecting line of two nodes i.e. line drawn from one node to another.

Depth or height:- It is the maximum label of may node in a given tree here, the depth of a tree is 4.

Forest:- It is a set of disjoint trees. In a given tree if we remove it's root then it becomes forest.

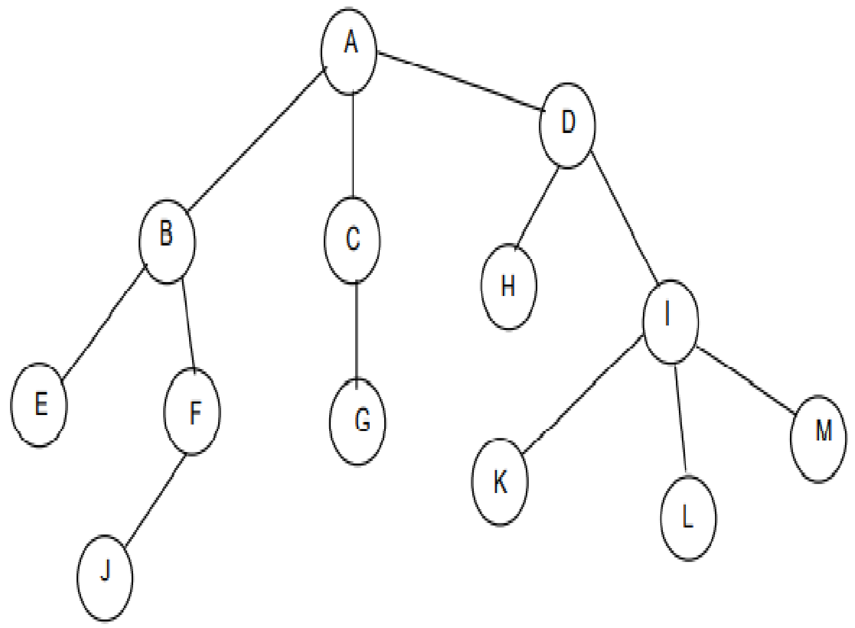


Fig. Tree

Basic Operation on Binary Tree

The basic operations that can be performed on a binary search tree data structure, are the following -

- **Insert** - Inserts an element in a tree/create a tree.
- **Deletion –** Delete an element in a tree.
- **Search** - Searches an element in a tree.
- **Preorder Traversal** - Traverses a tree in a pre-order manner.
- **In order Traversal** - Traverses a tree in an in-order manner.
- **Post order Traversal** - Traverses a tree in a post-order manner.

Binary Tree

Binary tree:-

- A binary tree is a finite set of data items which is either empty or consist of a single item called the root and two disjoint binary trees the lefts subs tree and right subtree.
- In binary tree the maximum degree of any node is almost 2 i.e. each node having 0, 1 or 2 degree node.

Binary Tree

- A binary tree is a tree in which no node can have more than two children.
- Typically these children are described as “left child” and “right child” of the parent’s node.
- A binary tree T is defined as a finite set of elements, called nodes, such that:
  - ❖ T is empty (i.e. if T has no node called null tree or empty tree).
  - ❖ T contains a special node R, called root node of T, and the remaining nodes of T from an ordered pair of disjoined binary trees T1 and T2, and they are called left and right sub tree of R. If T1 is non empty then its root is called the left successor of R, Similarly if T2 is non empty then its root is called the right successor of R.

Binary Tree

- ❖ Consider a binary tree T in Fig 2. Here ‘A’ is the root node of the binary tree T.
- ❖ Then ‘B’ is the left child of ‘A’ and ‘C’ is the right child of ‘A’ i.e., ‘A’ is a father of ‘B’ and ‘C’.
- ❖ The node ‘B’ and ‘C’ are called brothers. If a node has no child then it is called a leaf node. Nodes D, H, I, F, J are leaf node in Fig 2

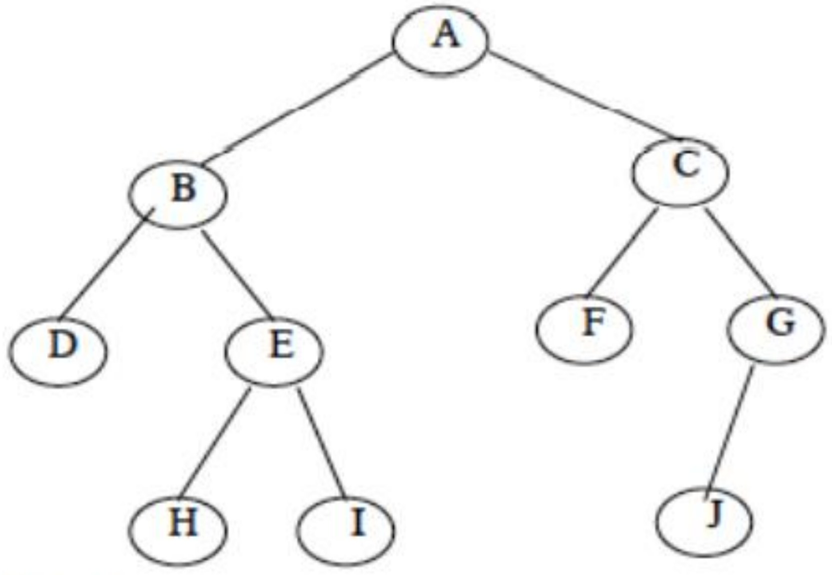


Fig 2: Binary Tree

Binary Tree

Strictly Binary Tree:

- The tree is said to be strictly binary tree, if every non-leaf node in a binary tree has non-empty left and right sub trees.
- A strictly binary tree with n leafs always contains 2n–1 node.
- The tree in Fig 3 is strictly binary tree; where as the tree in Fig 2 is not.
- That is every node in the strictly binary tree can have either no children or two children.
- They are also called 2-tree or extended binary tree.

Binary Tree

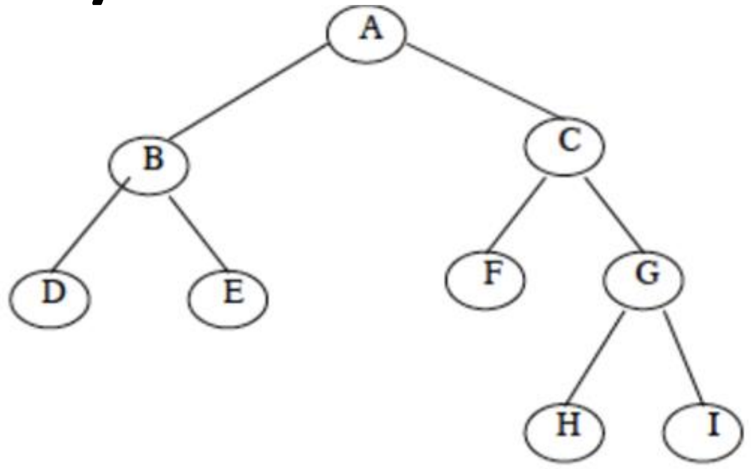


Fig 3: Strictly Binary Tree

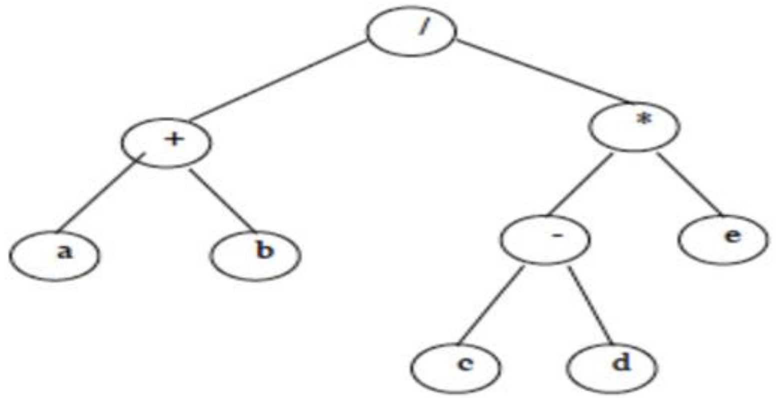


Fig 4: Expression Tree

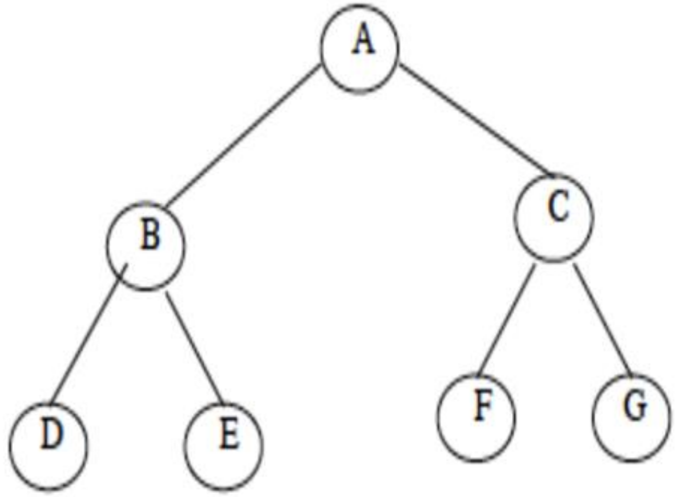


Fig5: Complete Binary Tree

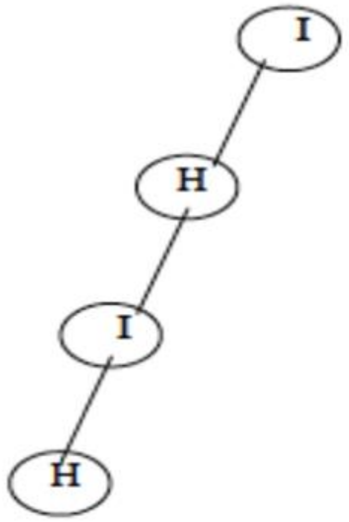


Fig6: Left Skewed

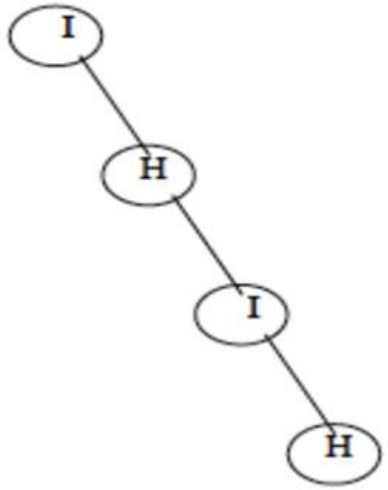


Fig7: Right Skewed

Complete Binary Tree

Complete Binary Tree:

- ❖ A complete binary tree at depth ‘d’ is the strictly binary tree, where all the leaf's are at level d.
- ❖ Fig 5 Ilustre's the complete binary tree of depth 2.
- ❖ A binary tree with n nodes, n > 0, has exactly n – 1 edges.
- ❖ If a binary tree contains n nodes at level l, then it contains at most 2n nodes at level l + 1.
- ❖ A complete binary tree of depth d is the binary tree of depth d contains exactly (2^l) nodes at each level l between 0 and d

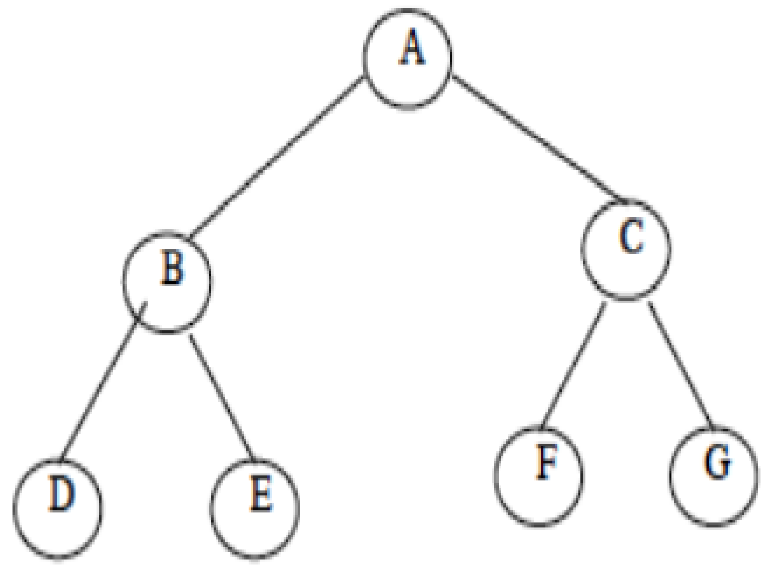


Fig5: Complete Binary Tree

Complete Binary Tree

The main difference between a binary tree and ordinary tree is:

1. A binary tree can be empty where as a tree cannot.
2. Each element in binary tree has exactly two sub trees (one or both of these sub trees may be empty). Each element in a tree can have any number of sub trees.
3. The sub tree of each element in a binary tree is ordered, left and right sub trees. The sub trees in a tree are unordered.

Complete Binary Tree

- ❖ If a binary tree has only left sub trees, then it is called left skewed binary tree.
- ❖ Fig 6 is a left skewed binary tree.
- ❖ If a binary tree has only right sub trees, then it is called right skewed binary tree. Fig 7 is a right skewed binary tree.

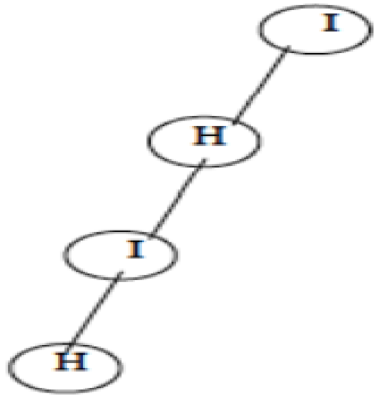


Fig6: Left Skewed

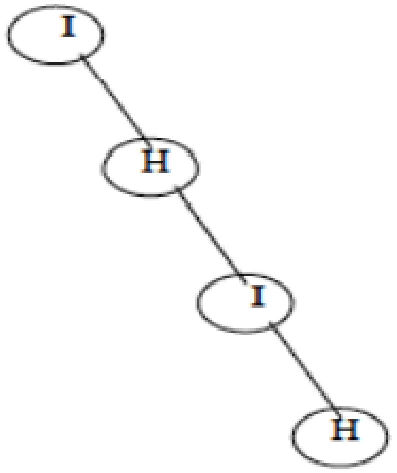


Fig7: Right Skewed

BINARY TREE REPRESENTATION

There are two ways of representing binary tree T in memory:

1. Sequential representation using arrays
2. Linked list representation

BINARY TREE REPRESENTATION

❖ An array can be used to store the nodes of a binary tree.

❖ The nodes stored in an array of memory can be accessed sequentially. Suppose a binary tree T of depth d. Then at most (2^d -1) nodes can be there in T. (i.e.,SIZE = 2^d –1).

❖ Consider a binary tree in Fig 8 of depth 3. Then SIZE = 2*3 – 1 = 7.

❖ Then the array A [7] is declared to hold the nodes.

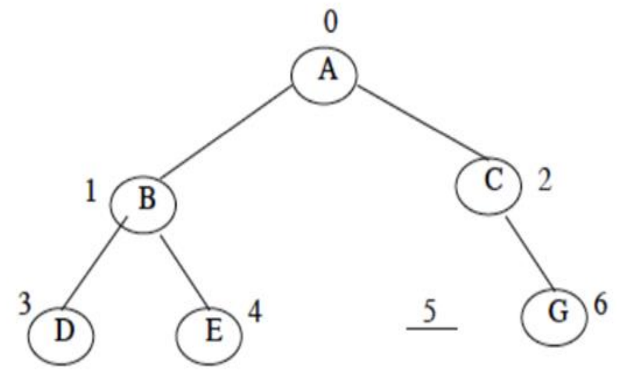


Fig8: Binary Tree of Depth 3

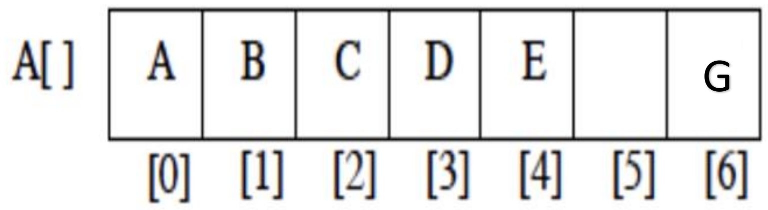


Fig9: Array Representation of Binary Tree of Fig8

BINARY TREE REPRESENTATION

To perform any operation on Binary tree we have to identify the father, the left child and right child of node.

1. **The father of a node having index n can be obtained by (n – 1)/2.**

   For example to find the father of D, where array index n = 3.

   i. Then the father nodes index can be obtained = (n – 1)/2 = 3 – 1/2 = 2/2 = 1 i.e., A[1] is the father D, which is B. 2.

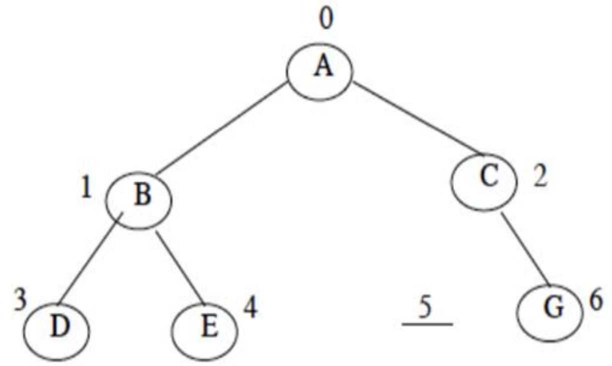
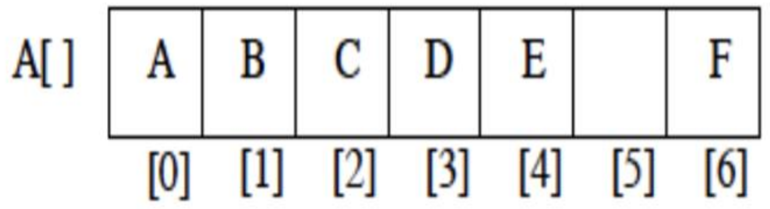


Fig8: Binary Tree of Depth 3



A[ ]	A	B	C	D	E		F
	[0]	[1]	[2]	[3]	[4]	[5]	[6]

Fig9: Array Representation of Binary Tree of Fig8

BINARY TREE REPRESENTATION

To perform any operation on Binary tree we have to identify the father, the left child and right child of node.

2. The left child of a node having index n can be obtained by (2n+1).

i. For example to find the left child of C, where array index n = 2.

ii. Then it can be obtained by = (2n +1) = 2*2 + 1 = 4 + 1 = 5; i.e., A[5] is the left child of C, which is NULL. So no left child for C.

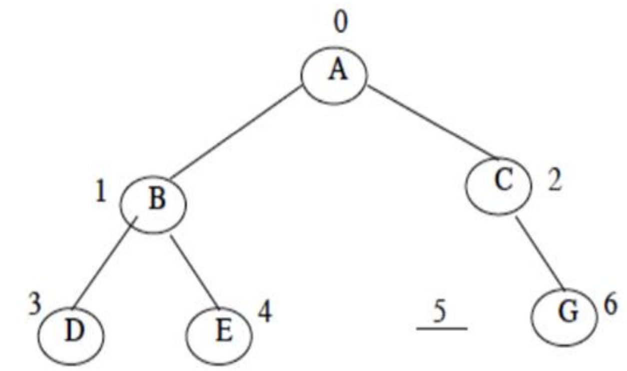


Fig8: Binary Tree of Depth 3

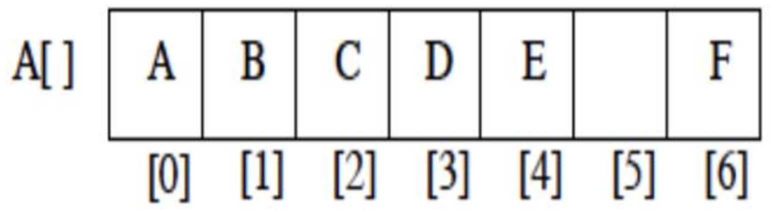


Fig9: Array Representation of Binary Tree of Fig8

BINARY TREE REPRESENTATION

3. The right child of a node having array index n can be obtained by the formula (2n+ 2).

i. For example to find the right child of B, where the array index n = 1.

ii. Then = (2n + 2) = 2*1 + 2 = 4 i.e., A[4] is the right child of B, which is E.

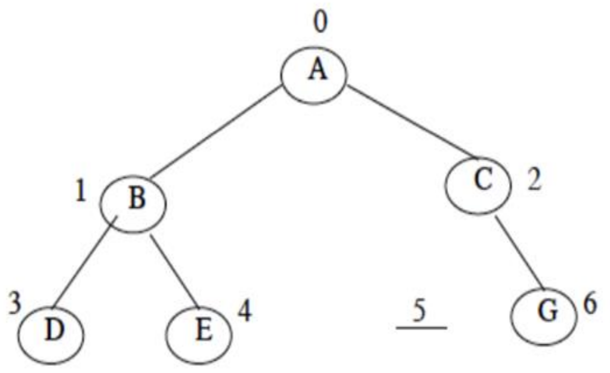


Fig8: Binary Tree of Depth 3

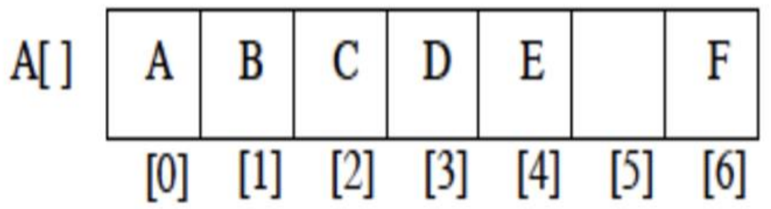


Fig9: Array Representation of Binary Tree of Fig8

4. If the left child is at array index n, then its right brother is at (n + 1).

i. Similarly, if the right child is at index n, then its left brother is at (n – 1).

BINARY TREE REPRESENTATION

- The array representation is more ideal for the complete binary tree.
- The Fig8 is not a complete binary tree. Since there is no left child for node C, i.e., A[5] is vacant.
- Even though memory is allocated for A [5] it is not used, so wasted unnecessarily.

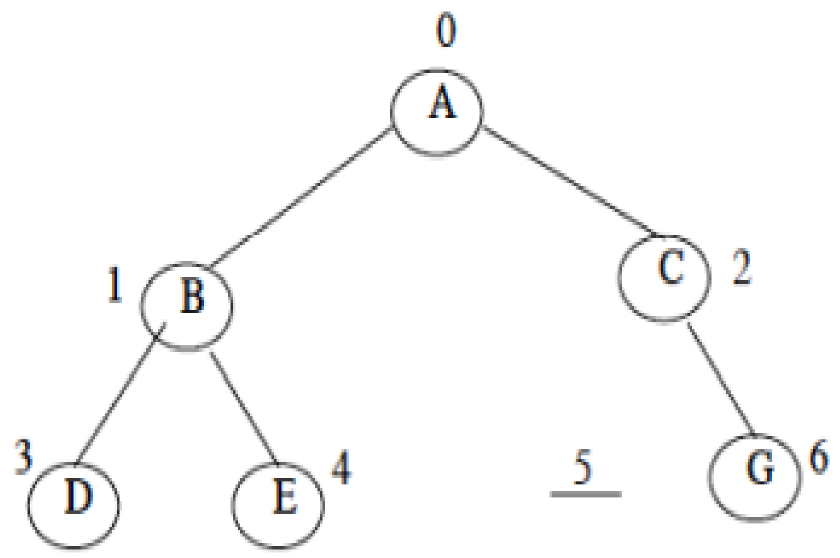


Fig8: Binary Tree of Depth 3

LINKED LIST REPRESENTATION

- The most popular and practical way of representing a binary tree is using linked list (or pointers).
- In linked list, every element is represented as nodes.
- A node consists of three fields such as:
  - i. Left Child (LChild)
  - ii. Information of the Node (Info)
  - iii. Right Child (RChild)
- The LChild links to the left child of the parent's node, info holds the information of every node and RChild holds the address of right child node of the parent node.
- The following figures shows that the Linked List representation of the binary tree of Fig 8.

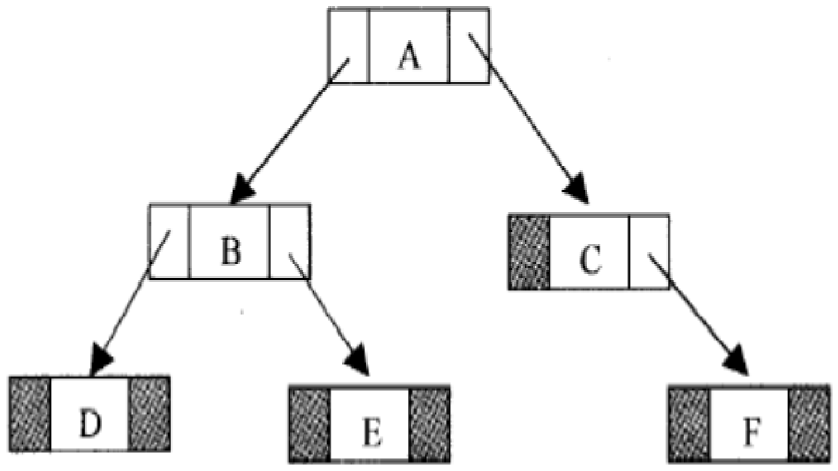


Fig9: Linked List representations of Binary Tree

LINKED LIST REPRESENTATION

- ❖ If a node has no left or/and right node.
- ❖ Corresponding LChild and RChild is assigned to NULL.
- ❖ The node structure in C can be represented as:

```
truct Node {
        int info;
        struct Node *LChild;
        struct Node *RChild;
}; typedef struct Node *NODE;
```

BASIC OPERATIONS IN BINARY TREE (Primitive Operations)

The basic operations that are commonly performed in binary tree are:

1. Create an empty binary tree
2. Traversing a binary tree
3. Inserting a new node
4. Deleting a Node
5. Searching for a Node
6. Determine the total no: of Nodes
7. Determine the total no: leaf Nodes
8. Determine the total no: non-leaf Nodes
9. Find the smallest element in a Node
10. Finding the largest element
11. Find the Height of the tree
12. Finding the Father/Left Child/Right Child/Brother of an arbitrary node.

BINARY SEARCH TREE (BST)

A Binary Search Tree is a binary tree, which is either empty or satisfies the following properties:

1. Every node has a value and no two nodes have the same value (i.e., all the values are unique).
2. If there exists a left child or left sub tree then its value is less than the value of the root.
3. The value(s) in the right child or right sub tree is larger than the value of the root node.

BINARY SEARCH TREE (BST)

- The Fig 11 shows a typical binary search tree. Here the root node information is 50.
- Note that the right sub tree node's value is greater than 50, and the left sub tree nodes value is less than 50.
- Again right child node of 25 has large values than 25 and left child node has small values than 25.
- Similarly right child node of 75 has large values than 75 and left child node has small values that 75 and so on.

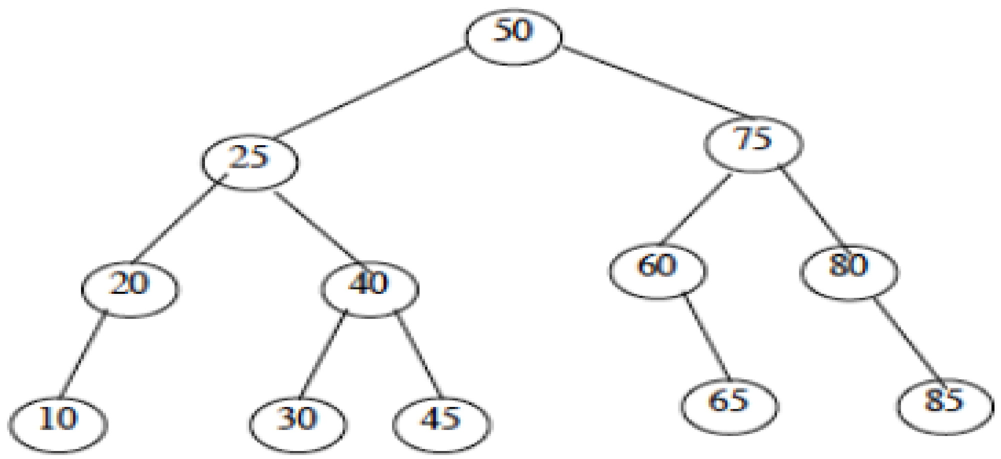


Figure 11: Binary Search Tree

BINARY SEARCH TREE (BST)

- All the primitives operation performed on Binary tree can also performed in Binary Search Tree (BST).

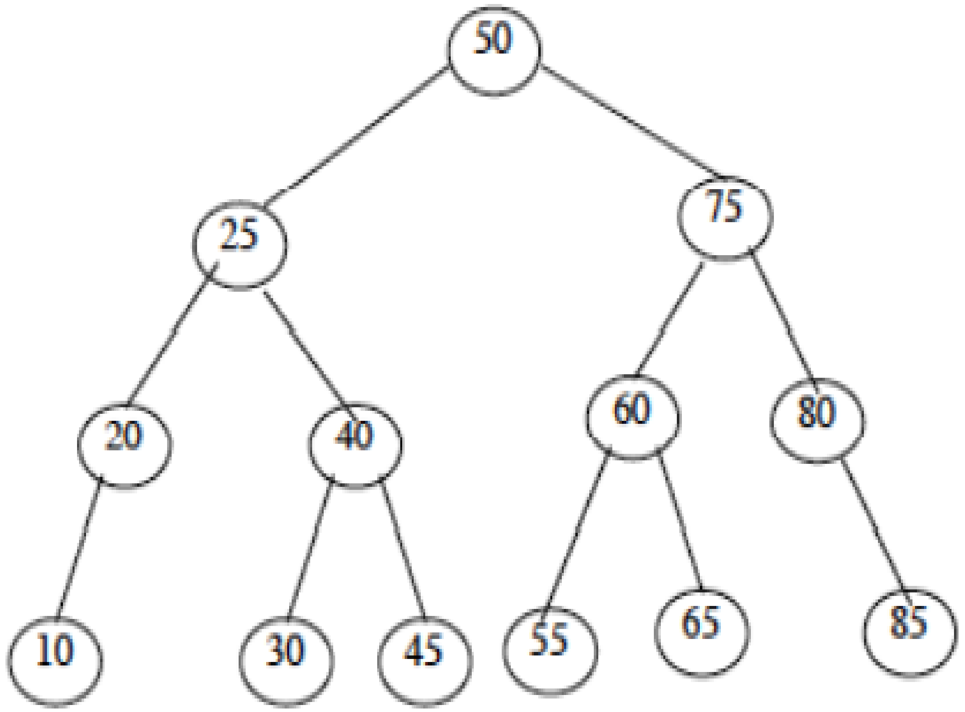


Figure 12: 55 is Inserted.

Inserting A Node In Binary Search Tree

A BST is constructed by the repeated insertion of new nodes to the tree structure. Inserting a node in to a tree is achieved by performing two separate operations.

i. The tree must be searched to determine where the node is to be inserted.

ii. Then the node is inserted into the tree.

Suppose a “DATA” is the information to be inserted in a BST.

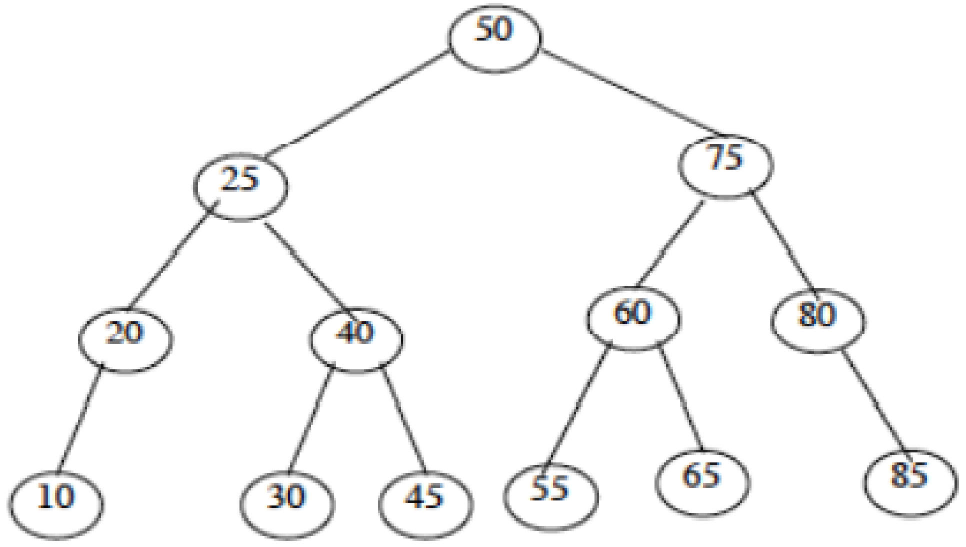
Always insert new node as a leaf node

- **Step 1:** Compare DATA with root node information of the tree
  i. If (DATA < ROOT → Info) Proceed to the left child of ROOT
  ii. If (DATA > ROOT → Info) Proceed to the right child of ROOT
- **Step 2:** Repeat the Step 1 until we meet an empty sub tree, where we can insert the DATA in place of the empty sub tree by creating a new node.
- **Step 3:** Exit

Inserting A Node In Binary Search Tree

For example, consider a binary search tree in Fig 11. Suppose we want to insert a DATA = 55 in to the tree, then following steps one obtained:

1. Compare 55 with root node info (i.e., 50) since 55 > 50 proceed to the right sub tree of 50.
2. The root node of the right sub tree contains 75. Compare 55 with 75. Since 55 < 75 proceed to the left sub tree of 75.
3. The root node of the left sub tree contains 60. Compare 55 with 60. Since 55 < 60 proceed to the right sub tree of 60.
4. Since left sub tree is NULL place 55 as the left child of 60 as shown in above Fig12.



Algorithm For Inserting An Element Into BST

NEWNODE is a pointer variable to hold the address of the newly created node. DATA is the information to be pushed and TEMP is the Temporary pointer variable of type NODE

```
1. Input the DATA to be pushed and ROOT node of the tree.
2. NEWNODE = Create a New Node.
3. If (ROOT == NULL)
   i.   ROOT=NEWNODE
4. ELSE
   i.   TEMP = ROOT
   ii.  Repeat until true (while (1))
        a. IF (DATA < TEMP →info)
           i.   IF (TEMP →LChild not equal
                to NULL)TEMP=TEMP→Lchild
           ii.  ELSE TEMP→LChild =
                NewNode
           iii. Return
        b. ELSE IF (DATA > TEMP →info)
           i.   IF (TEMP →RChild not equal
                to NULL) TEMP =
                TEMP→RChild
           ii.  ELSE TEMP→RChild =
                NewNode
           iii. Return
        c. ELSE
           i.   Display Duplicate node
           ii.  Return
5. Exit
```

Algorithm For Deleting a Node from BST

First search and locate the node to be deleted. Then any one of the following conditions arises:

1. The node to be deleted has no children
2. The node has exactly one child (or sub tress, left or right sub tree)
3. The node has two children (or two sub tress, left and right sub tree)

Algorithm For Deleting a Node from BST

Suppose the node to be deleted is N.

1. If N has no children then simply delete the node and place its parent node by the NULL pointer.
2. If N has one child, check whether it is a right or left child.
    1. If it is a right child, **then find the smallest element from the corresponding right sub tree**. Then replace the smallest node information with the deleted node.
3. If N has a left child, **find the largest element from the corresponding left sub tree.** Then replace the largest node information with the deleted node.
4. The same process is repeated if N has two children, i.e., left and right child. Randomly select a child and find the small/large node and replace it with deleted node.

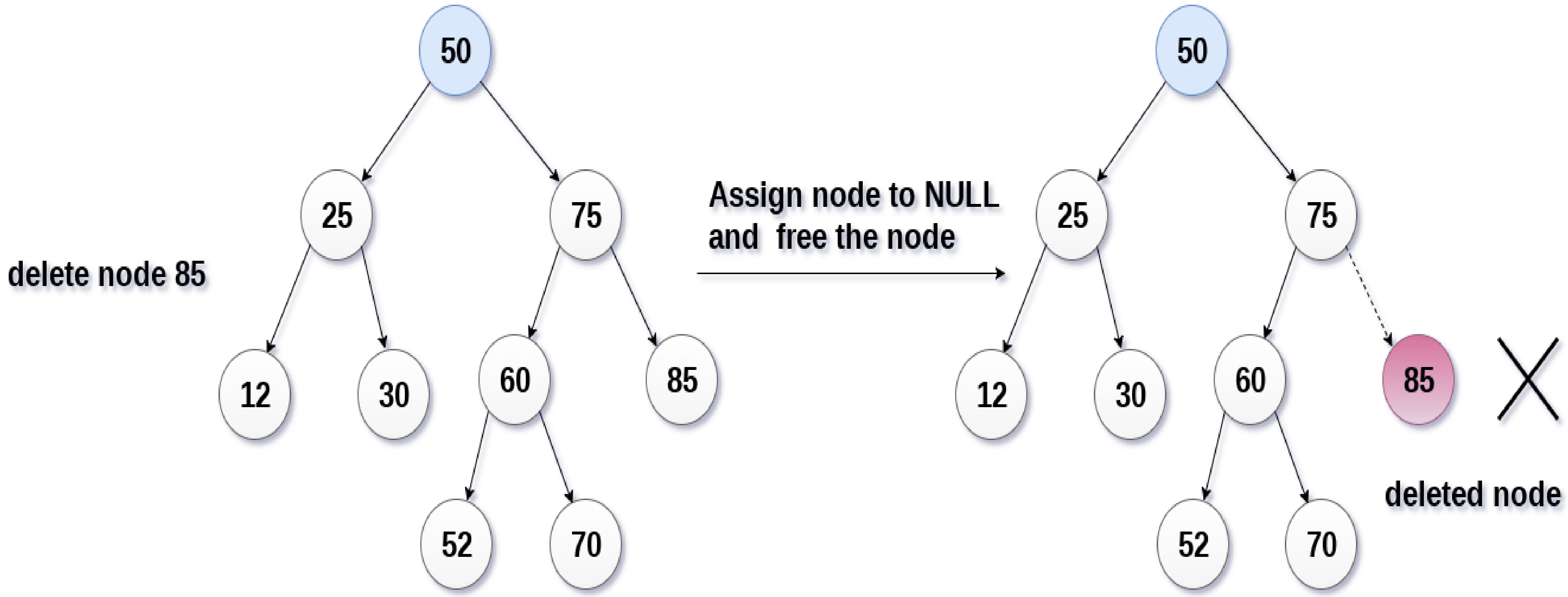
NOTE that the tree that we get after deleting a node should also be a binary search tree.

Algorithm For Deleting a Node from BST

- Delete function is used to delete the specified node from a binary search tree.
- However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate.
- There are three situations of deleting a node from binary search tree.
  1. The node to be deleted is a leaf node
  2. The node to be deleted has only one child.
  3. The node to be deleted has two children.

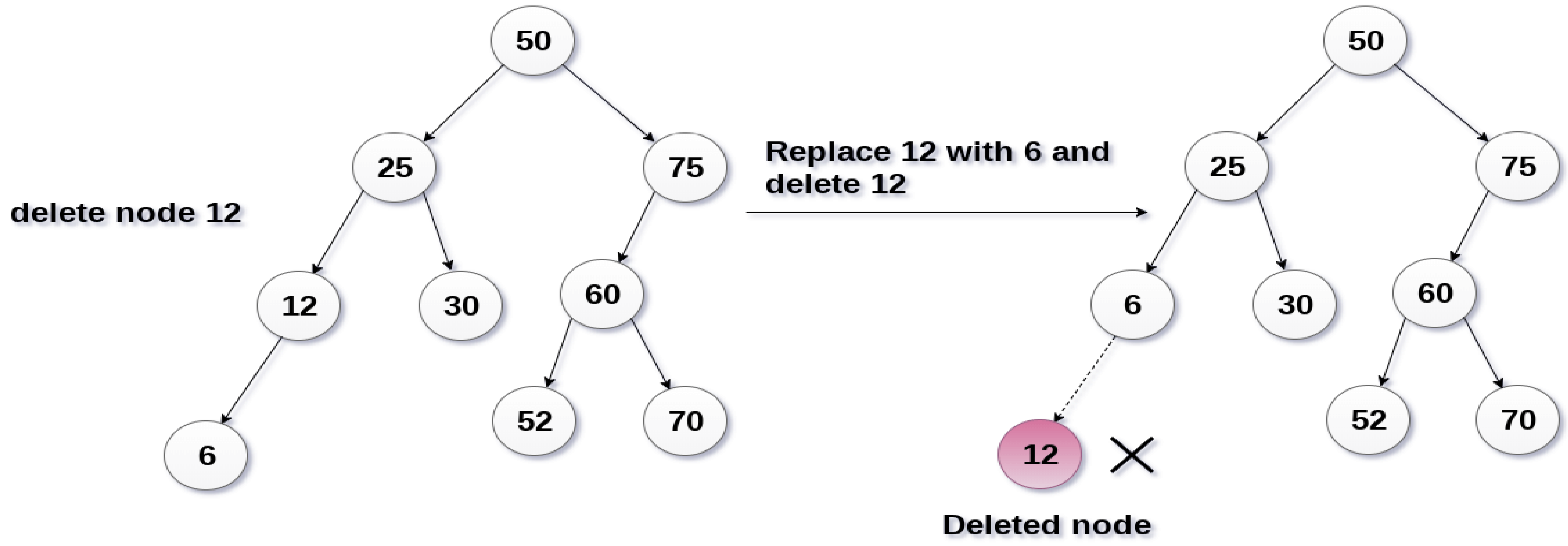
The node to be deleted is a leaf node

- It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space.
- In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



The node to be deleted has only one child.

- In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.
- In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted.

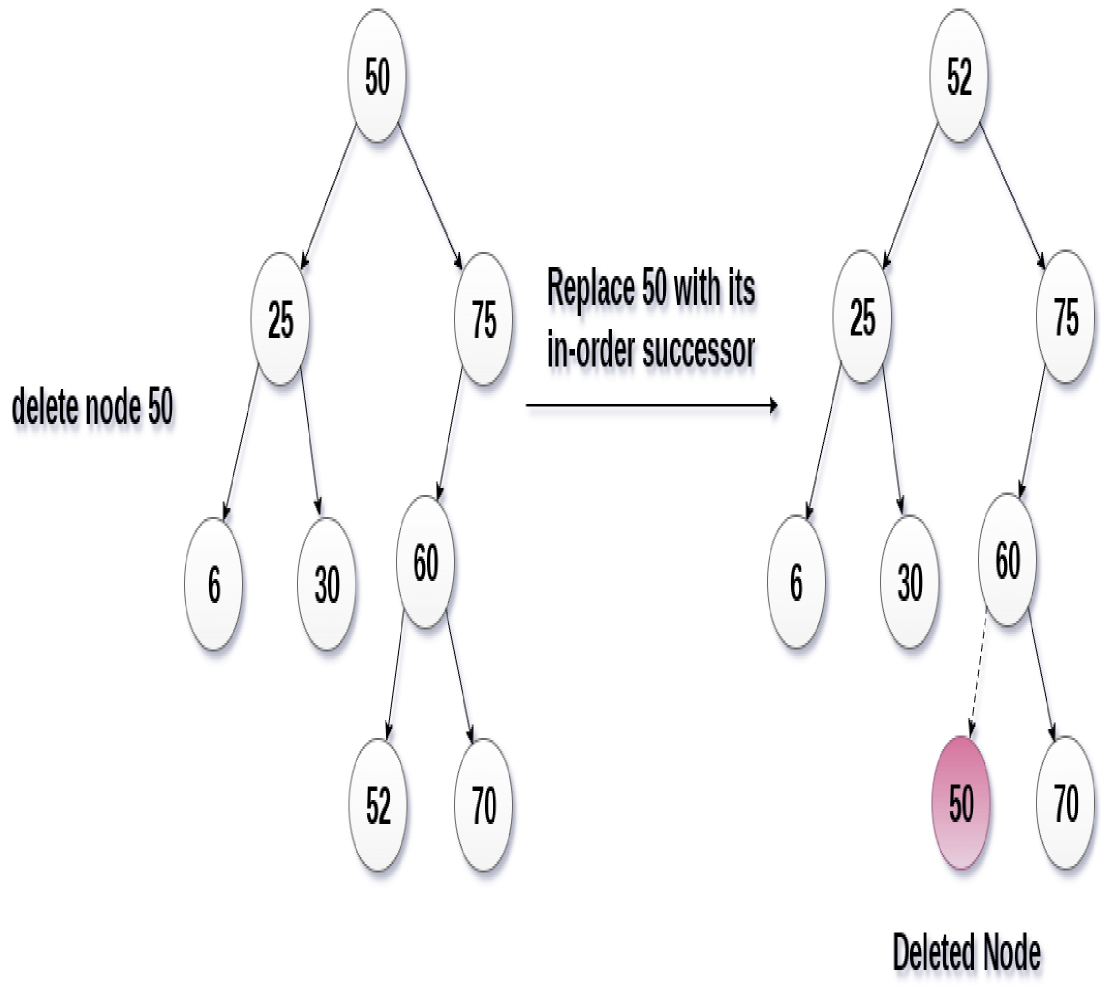


The node to be deleted has two children.

- It is a bit complexed case compare to other two cases. However, the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.
- In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below
- 6, 25, 30, 50, 52, 60, 70, 75
- Replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.

The node to be deleted has two children.

- It is a bit complexed case compare to other two cases. However, the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.
- In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below
- 6, 25, 30, 50, 52, 60, 70, 75
- Replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



An algorithm to delete a node in a BST

1. Input the number of nodes.
2. Input the nodes of the tree.
3. Consider the first element as the root element and insert all the elements.
4. Input the data of the node to be deleted.
5. If the node is a leaf node, delete the node directly.
6. Else if the node has one child, copy the child to the node to be deleted and delete the child node.
7. Else if the node has two children, find the in order successor of the node.
8. Copy the contents of the inorder successor to the node to be deleted and delete the inorder successor.

The node to be deleted has two children.

Delete (TREE, ITEM)

```
•Step 1: IF TREE = NULL
   Write "item not found in the tree" ELSE IF
ITEM < TREE -> DATA
  Delete(TREE->LEFT, ITEM)
  ELSE IF ITEM > TREE -> DATA
   Delete(TREE -> RIGHT, ITEM)
  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
```

Algorithm for Searching A Node from BST

```
1. Input the DATA to be searched and assign the address of the root node to ROOT.
2. TEMP = ROOT
3. Repeat Until DATA is not Equal to TEMP → Info)
    a. IF (DATA == TEMP → Info )
        i. Display “The DATA exist in the tree”
        ii. Return
    b. If (TEMP == NULL)
        i. Display “The DATA does not exist”
        ii. Return
    c. If(DATA > TEMP→Info)
        i. TEMP = TEMP→Rchild
    d. Else If(DATA < TEMP→Info)
        a. TEMP = TEMP→LChild
4. Exit
```

TRAVERSING BINARY TREE

❖ Tree traversal is one of the most common operations performed on tree data structures.

❖ It is a way in which each node in the tree is visited exactly once in a systematic manner. There are three standard ways of traversing a binary tree.

❖ They are:

i. Pre Order Traversal (Node-left-right)

ii. In order Traversal (Left-node-right)

iii. Post Order Traversal (Left-right-node)

TRAVERSING BINARY TREE

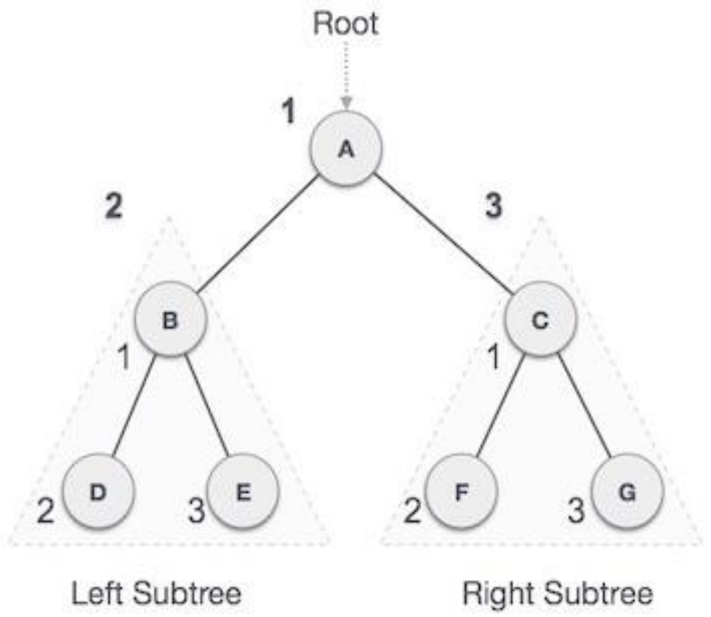
Pre Order Traversal (Root-Left-Right)

- A systematic way of visiting all nodes in a binary tress that visits a node, then visits the nodes in the left sub tree of the node, and then visits the nodes in the right sub tree of the node.
  1. Visit the root node
  2. Traverse the left sub tree in preorder
  3. Traverse the right sub tree in preorder

That is in preorder traversal, the root node is visited (or processed) first, before traveling through left and right sub trees recursively.

Pre order Traversal

- In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.
- We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**.
- **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be -$\boldsymbol{A} \rightarrow \boldsymbol{B} \rightarrow \boldsymbol{D} \rightarrow \boldsymbol{E} \rightarrow \boldsymbol{C} \rightarrow \boldsymbol{F} \rightarrow \boldsymbol{G}$



TRAVERSING BINARY TREE

Pre Order Traversal (Root-Left-Right)

C/C++ implementation of preorder traversal technique is as follows:

```
void preorder (Node * Root) {
       If (Root != NULL) {
                   printf (“%d\n”,Root → Info);
                   preorder(Root → Lchild);
                   preorder(Root → Rchild);
        }
}
```

TRAVERSING BINARY TREE

The preorder traversal of the following Binary tree is: A,B,D,E,H,I,C,F,G,J

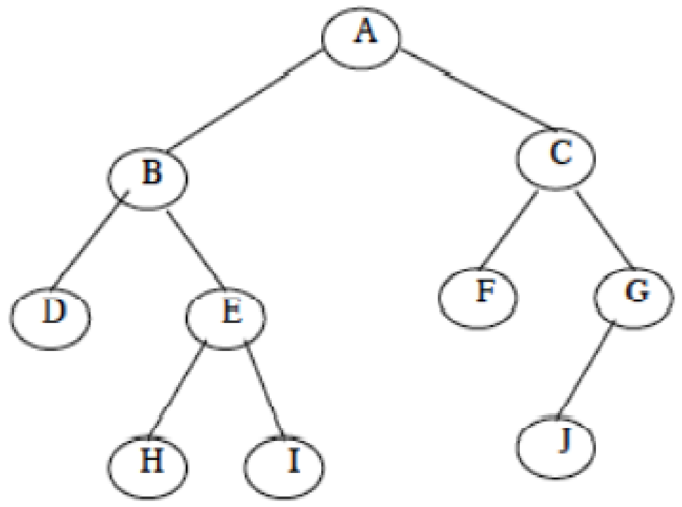


Fig10: Binary Tree

TRAVERSING BINARY TREE

In Order Traversal (Left-Root-Right)

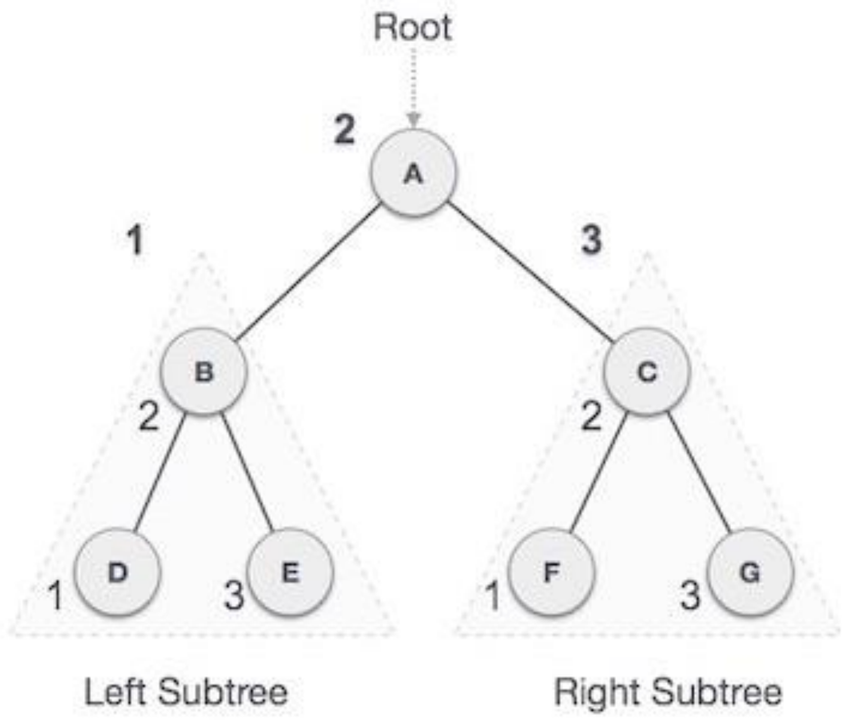
A systematic way of visiting all nodes in a binary tress that visits the nodes in the left sub tree of a node, then visits the node, and then visits the nodes in the right sub tree of the node.

1. Traverse the left sub tree in order
2. Visit the root node
3. Traverse the right sub tree in order

In Order Traversal

In-order Traversal

- In this traversal method, the left subtree is visited first, then the root and later the right sub-tree.
- We should always remember that every node may represent a subtree itself.
- If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.
- We start from **A**, and following in-order traversal, we move to its left subtree **B**.
- **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of in order traversal of this tree will be -***D*** → ***B*** → ***E*** → ***A*** → ***F*** → ***C*** → ***G***



In order traversal

Algorithm

Until all nodes are traversed

Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.

In order Traversal

- In order traversal, the left sub tree is traversed recursively, before visiting the root. After visiting the root the right sub tree is traversed recursively.
- C/C++ implementation of inorder traversal technique is as follows:

```
void inorder (NODE *Root) {
        If (Root != NULL){
                inorder(Root → Lchild);
                printf (“%d\n”,Root → info);
                inorder(Root → Rchild);
} }
```

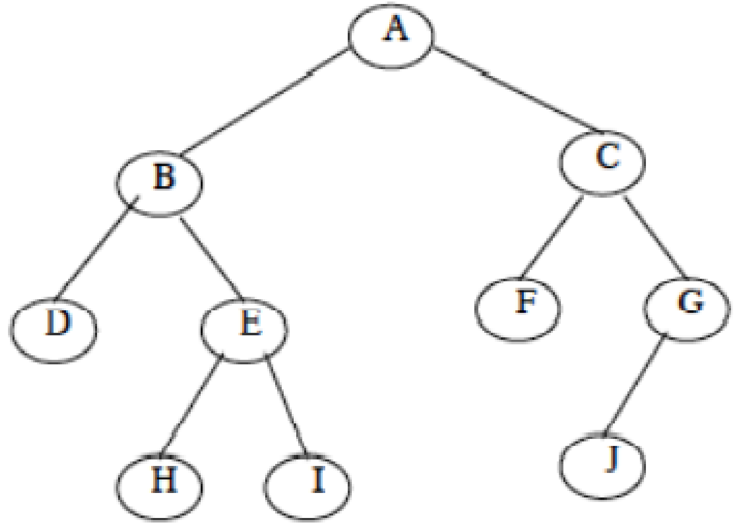


Fig10: Binary Tree

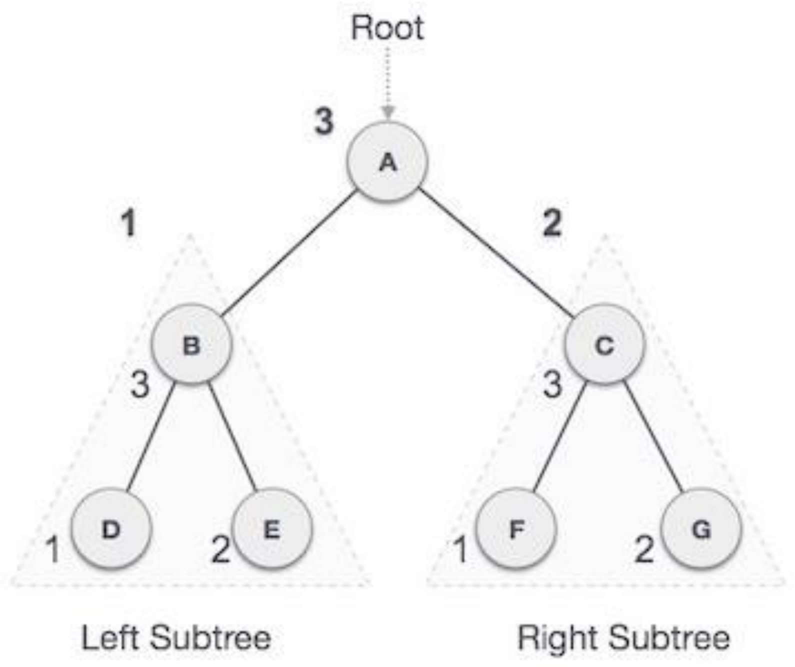
The in order traversal of a binary tree in Fig10 is: D, B, H, E, I, A, F, C, J, G

Post order Traversal

- In this traversal method, the root node is visited last, hence the name.
- First we traverse the left subtree, then the right subtree and finally the root node.
- We start from **A**, and following Post-order traversal, we first visit the left subtree **B**.
- **B** is also traversed post-order. The process goes on until all the nodes are visited.

- The output of post-order traversal of this tree will be -

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$



Post order traversal

Algorithm

Until all nodes are traversed

Step 1 − Recursively traverse left subtree.

Step 2 − Recursively traverse right subtree.

Step 3 − Visit root node.

TRAVERSING BINARY TREE

Post Order Traversal (Left-Right-Root)

- In Post Order traversal, the left and right sub tree(s) are recursively processed before visiting the root.
- C/C++ implementation of post order traversal technique is as follows:

```
void postorder (NODE *Root) {
        If (Root != NULL) {
                postorder(Root → Lchild);
                postorder(Root → Rchild);
                printf (“%d\n”, Root → info);
        } }
```

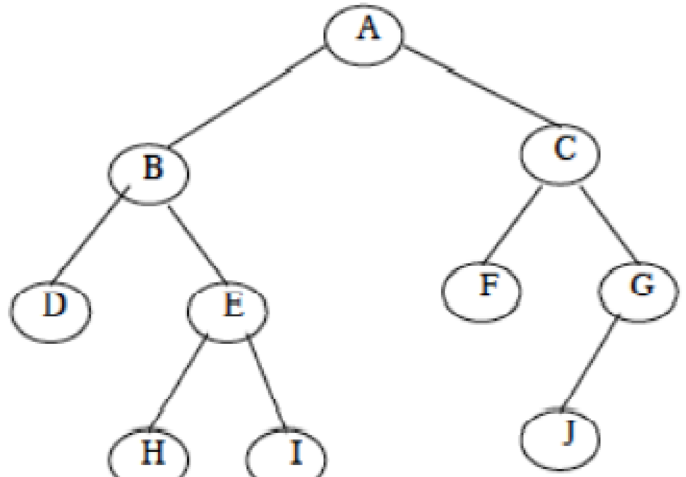


Fig10: Binary Tree

The post order traversal of a binary tree in Fig 10 is: D, H, I, E, B, F, J, G, C, A.

TRAVERSING BINARY TREE

Post Order Traversal (Left-Right-Root)

A systematic way of visiting all nodes in a binary tress that visits the node in the left sub tree of the node, then visits the node in the right sub tree in the node, and then visits the node.

1. Traverse the left sub tree in post order
2. Traverse the right sub tree in post order
3. Visit the root node

BALANCED BINARY TREE

- ❖ A balanced binary tree is one in which the largest path through the left sub tree is the same length as the largest path of the right sub tree, i.e., from root to leaf.
- ❖ Searching time is very less in balanced binary trees compared to unbalanced binary tree. i.e., balanced trees are used to maximize the efficiency of the operations on the tree. There are two types of balanced trees:
  1. Height Balanced Trees
  2. Weight Balanced Trees

Balanced Binary Tree

A balanced binary tree is one in which the **largest path** through the left sub tree is the same length as the **largest path** of the right sub tree, i.e., from root to leaf.

Searching time is very less in balanced binary trees compared to unbalanced binary tree. i.e., balanced trees are used to maximize the efficiency of the operations on the tree.

AVL Trees

Two Russian Mathematicians, G.M. Adel’son Vel’sky and E.M. Landis developed algorithm in 1962; here the tree is called AVL Tree.

An AVL tree is a binary tree in which the left and right sub tree of any node may differ in height by at most 1, and in which both the sub trees are themselves AVL Trees.

Each node in the AVL Tree possesses any one of the following properties:

- A node is called **left heavy**, if the largest path in its left sub tree is one level larger than the largest path of its right sub tree

- A node is called **right heavy**, if the largest path in its right sub tree is one level larger than the largest path of its left sub tree.

- The node is called **balanced**, if the largest paths in both the right and left sub trees are equal.

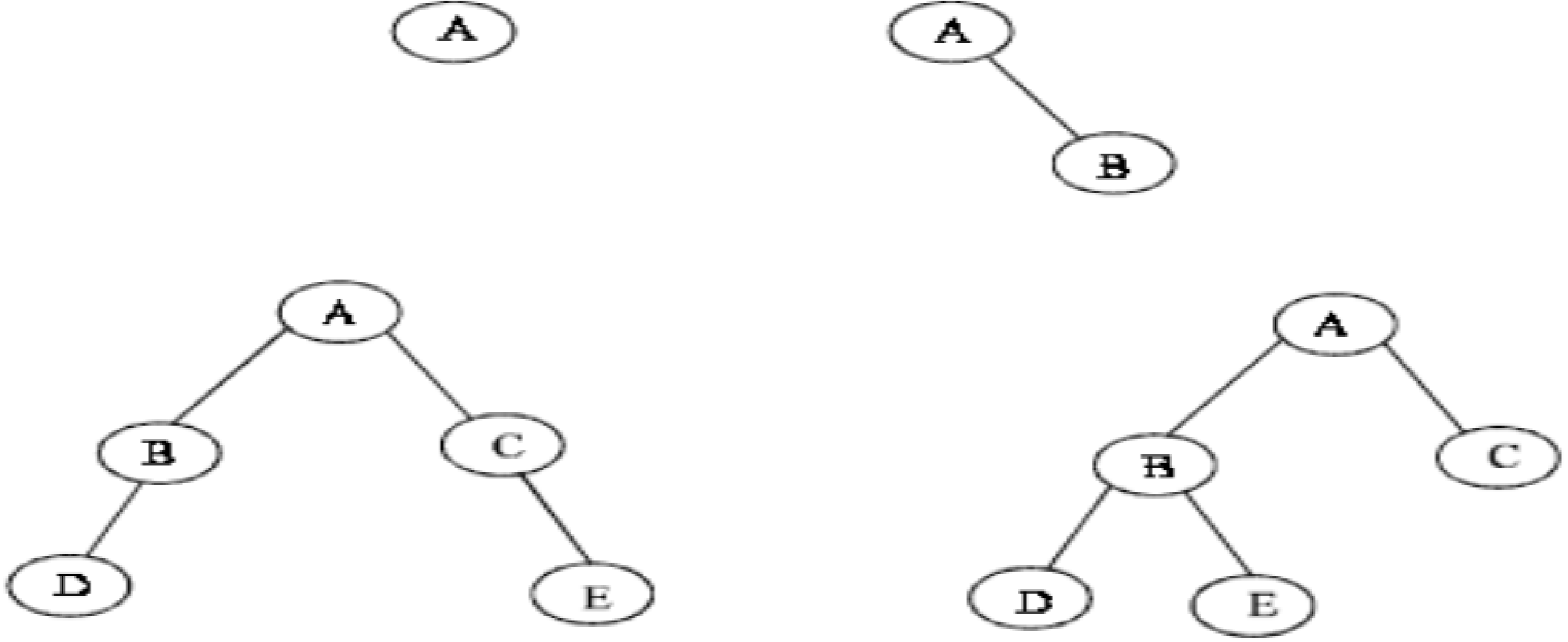


Fig17: AVL Trees

- Fig17 shows some example for AVL trees.
- The construction of an AVL Tree is same as that of an ordinary binary tree except that after the addition of each new node, a check must be made to ensure that the AVL balance conditions have not been violated.
- If the new node causes an imbalance in the tree, some rearrangement of the tree's nodes must be done.

Inserting a node in an AVL tree

Main point: **balanced factor = Height of left subtree – height of right sub tree.**

1. Insert the node in the same way as in an ordinary binary tree.

2. Trace a path from the new nodes, back towards the root for checking the height difference of the two sub trees of each node along the way.

3. Consider the node with the imbalance and the two nodes on the layers immediately below.

4. If these three nodes lie in a straight line, apply a single rotation to correct the imbalance.

5. If these three nodes lie in a dogleg pattern (i.e., there is a bend in the path) apply a double rotation to correct the imbalance.
6. Exit.

- The above algorithm will be illustrated with an example shown in Fig18, which is an unbalance tree.
- We have to apply the rotation to the nodes 40, 50 and 60 so that a balance tree is generated.
- Since the three nodes are lying in a straight line, single rotation is applied to restore the balance.

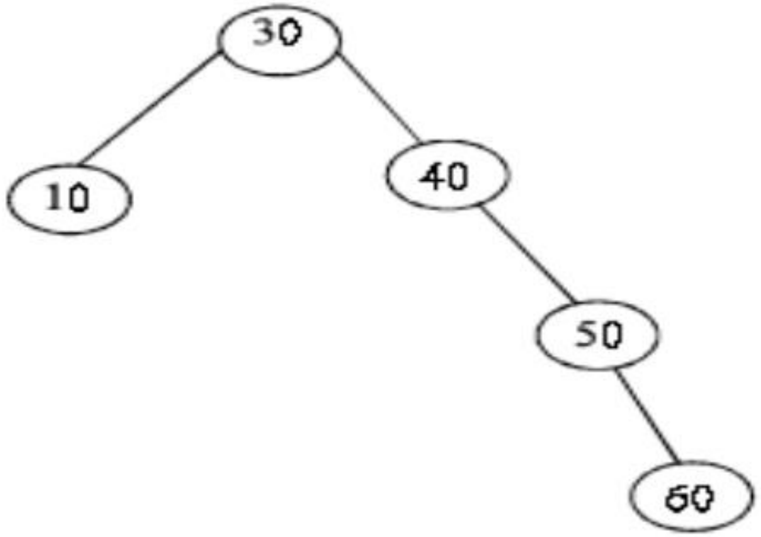


Fig 18

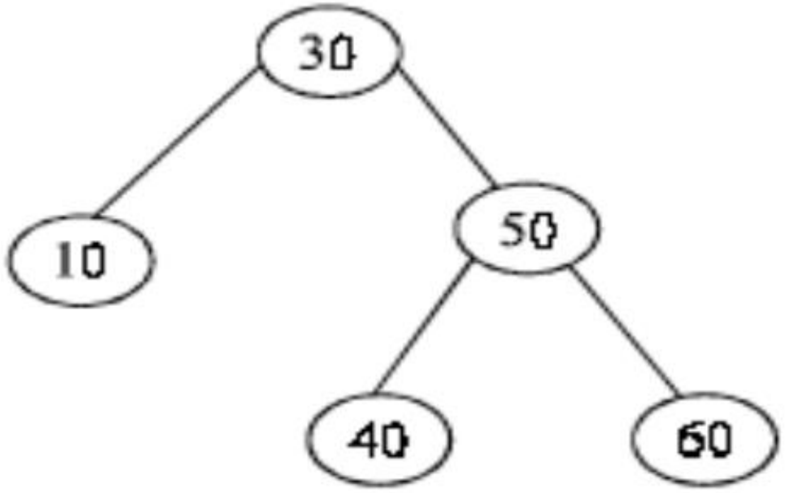


Fig19

Fig19 is a balance tree of the unbalanced tree in Fig18.

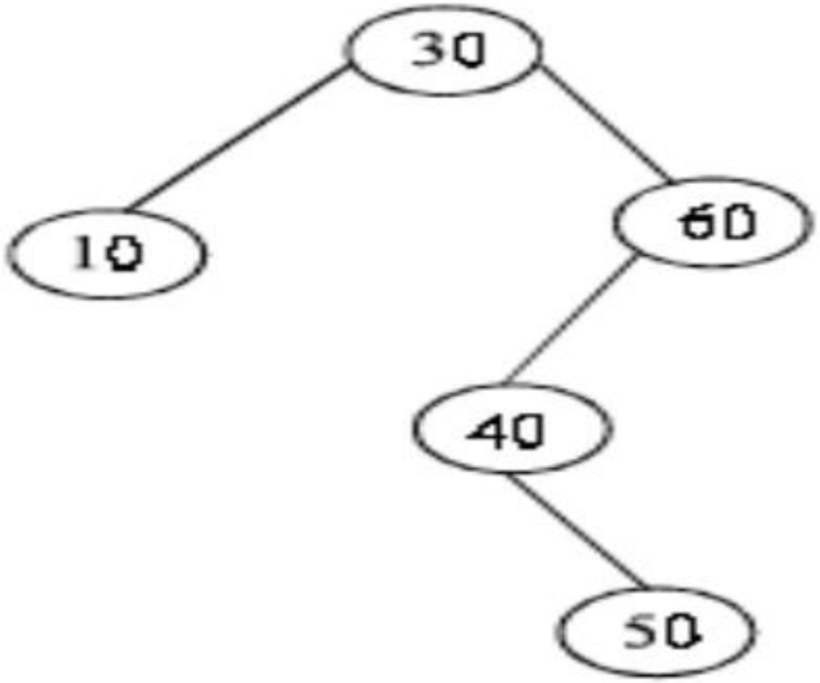


Fig20

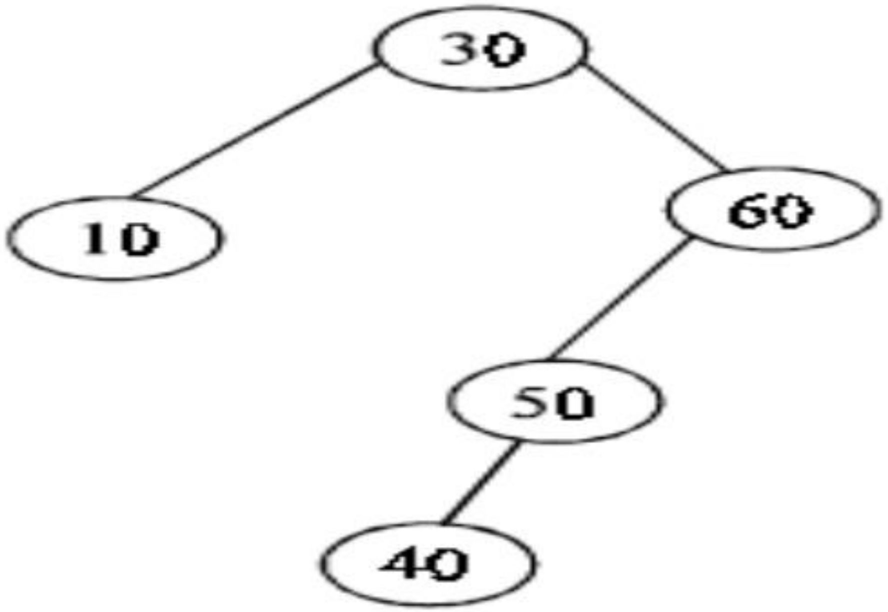


Fig21

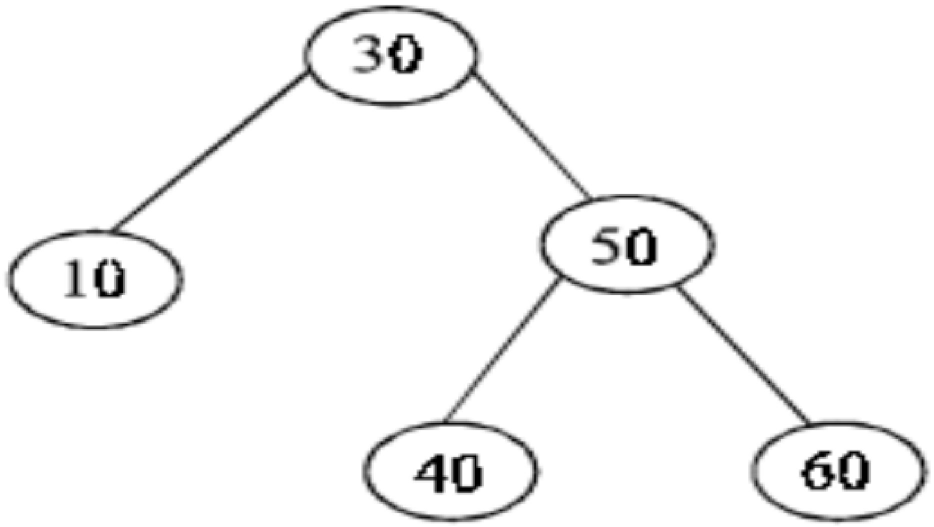


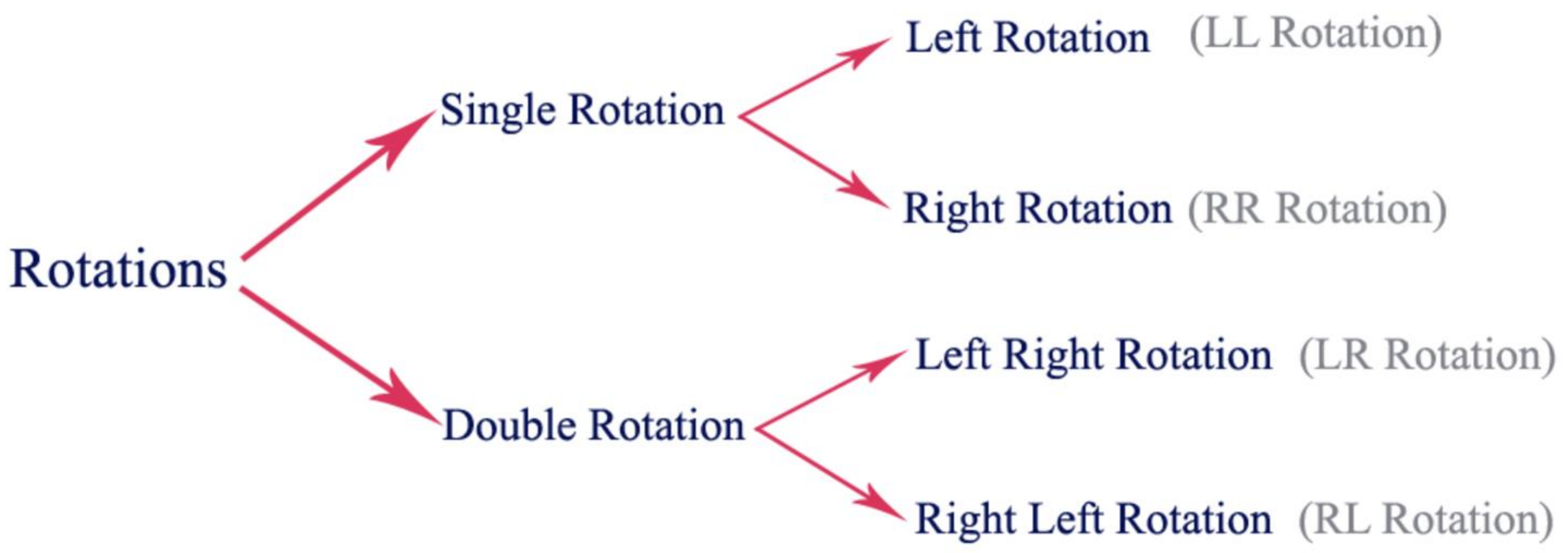
Fig22

Consider a tree in Fig20 to explain the double rotation.

AVL Tree Rotation

In AVL tree, after performing operations like insertion and deletion we need to check the **balance factor** of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation we use **rotation** operations to make the tree balanced.

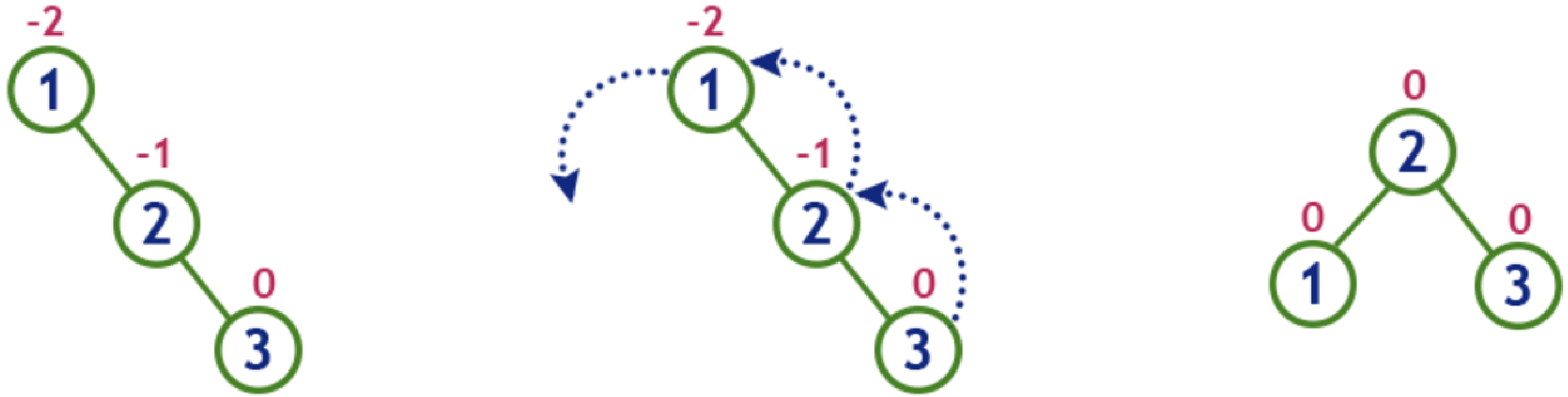
Rotation operations are used to make the tree balanced.


Rotations
Single Rotation
Double Rotation
Left Rotation (LL Rotation)
Right Rotation (RR Rotation)
Left Right Rotation (LR Rotation)
Right Left Rotation (RL Rotation)

Single left rotation

In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree...

insert 1, 2 and 3



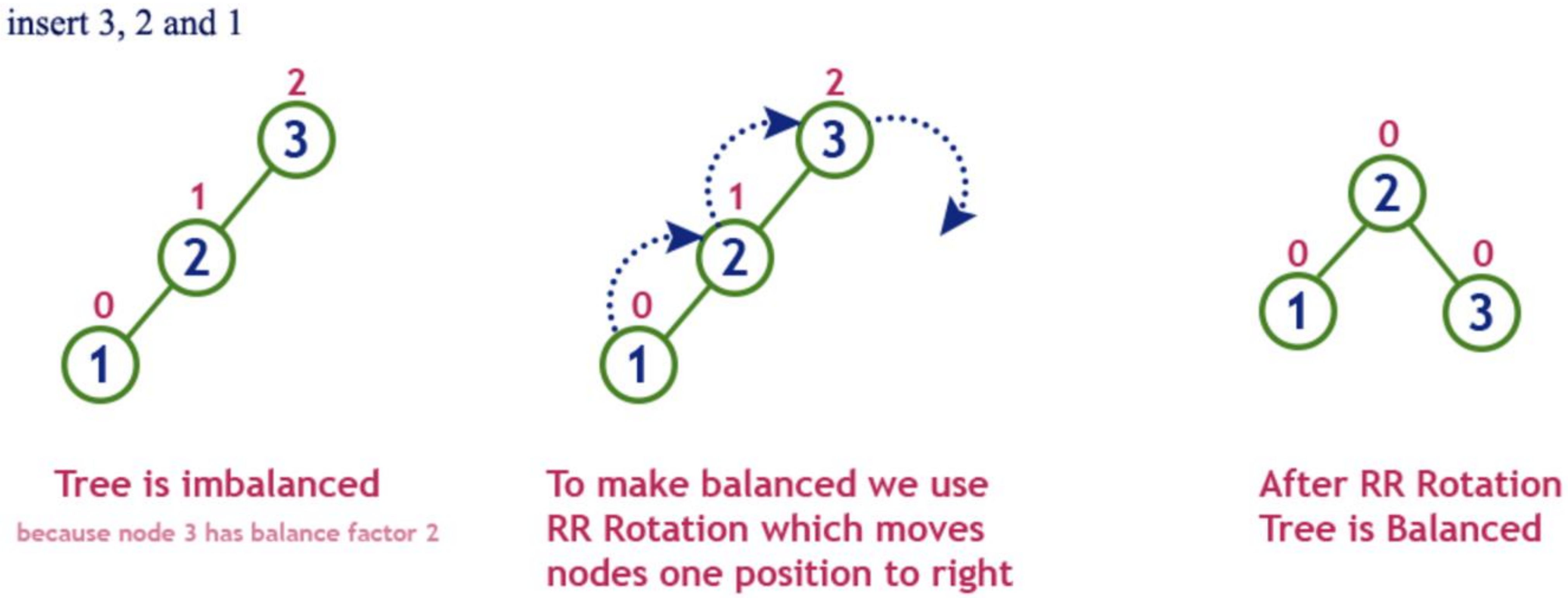
Tree is imbalanced

To make balanced we use LL Rotation which moves nodes one position to left

After LL Rotation Tree is Balanced

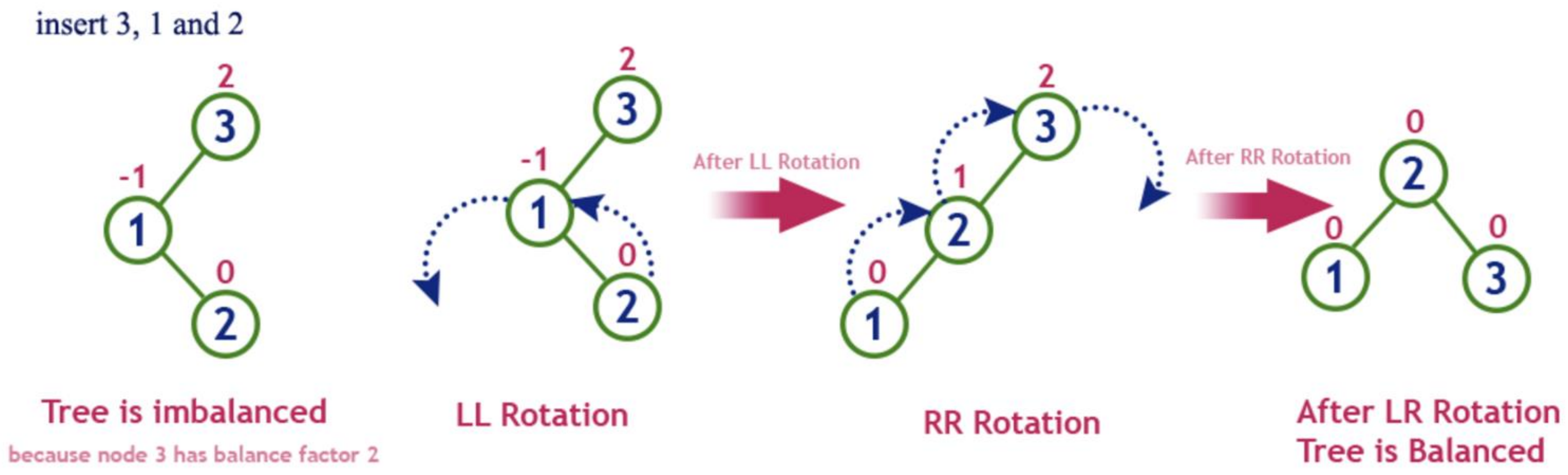
Single right rotation

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...



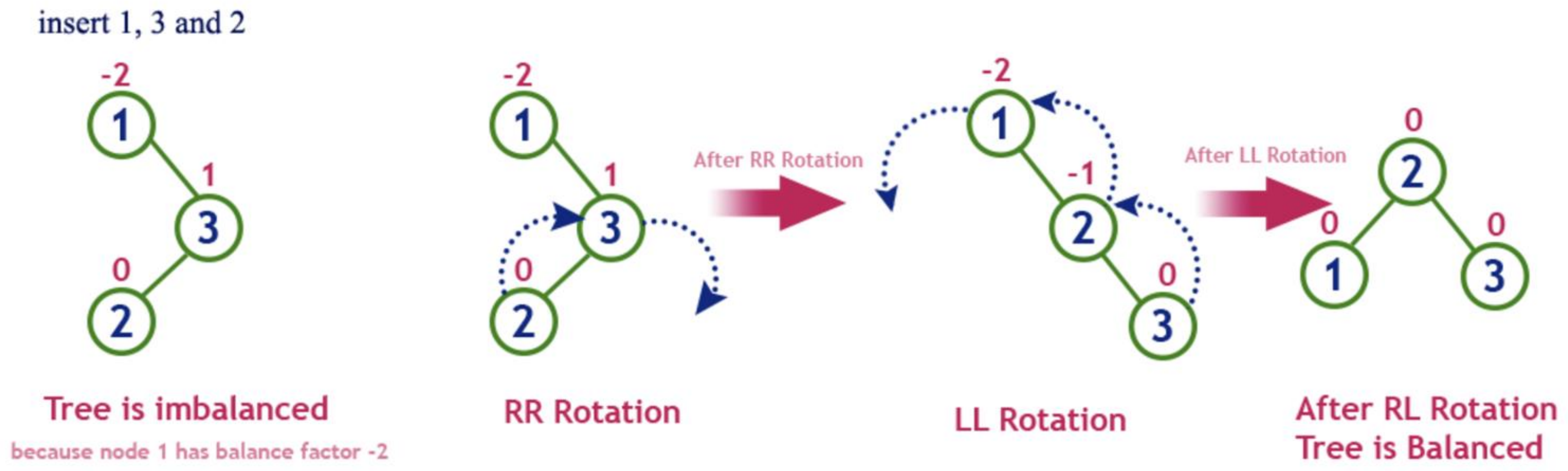
Left-Right Rotation (LR)

The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...



Right Left Rotation (RL)

The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...

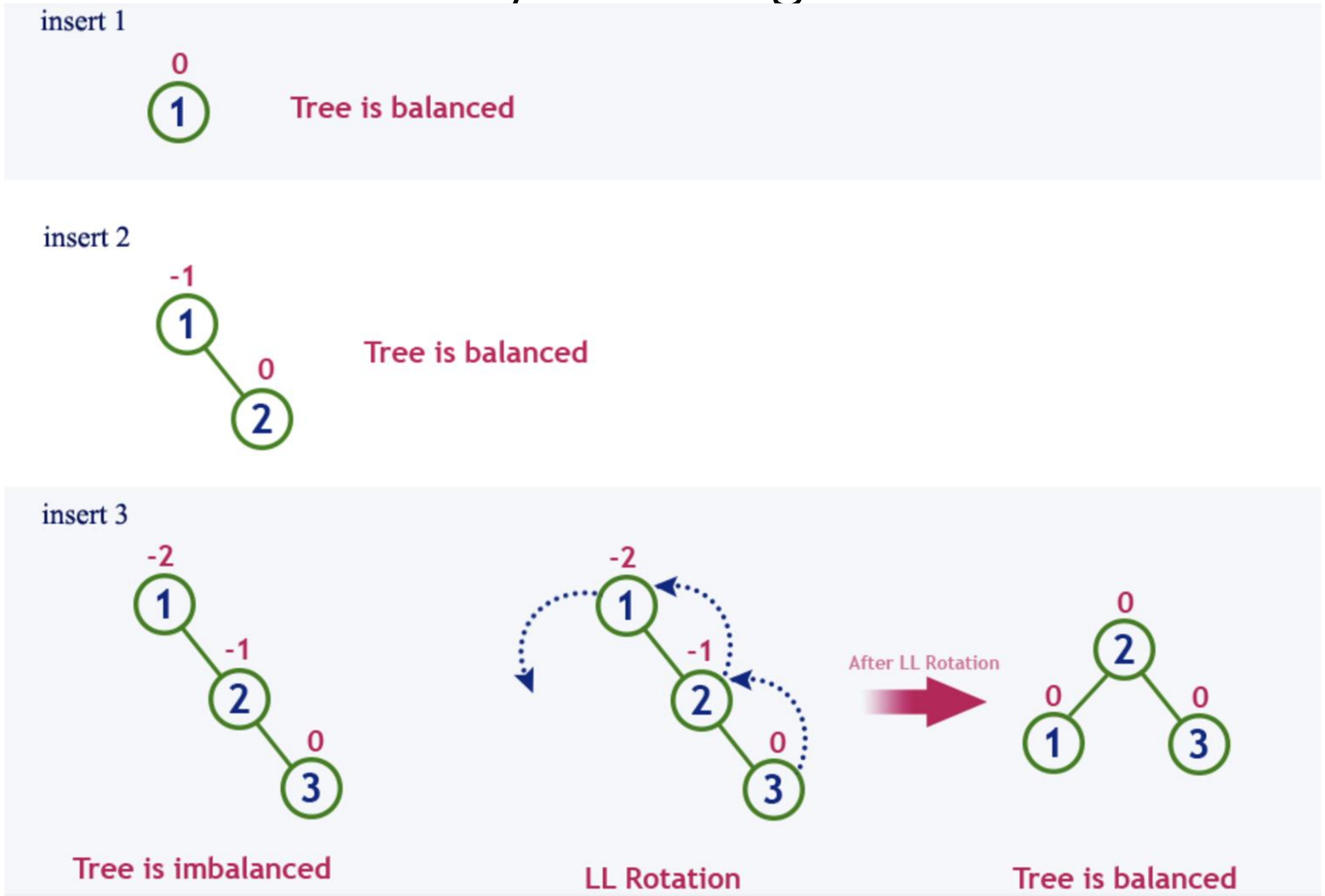


Insertion in AVL

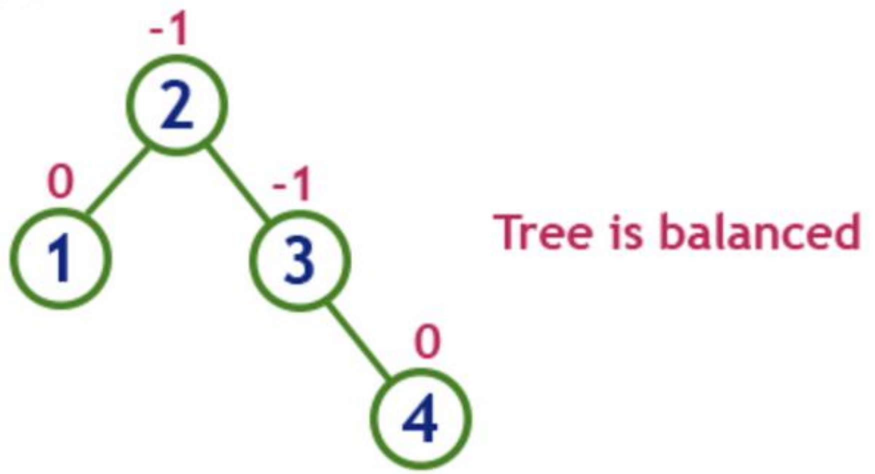
In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

- Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.
- Step 2 - After insertion, check the **Balance Factor** of every node.
- Step 3 - If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
- Step 4 - If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

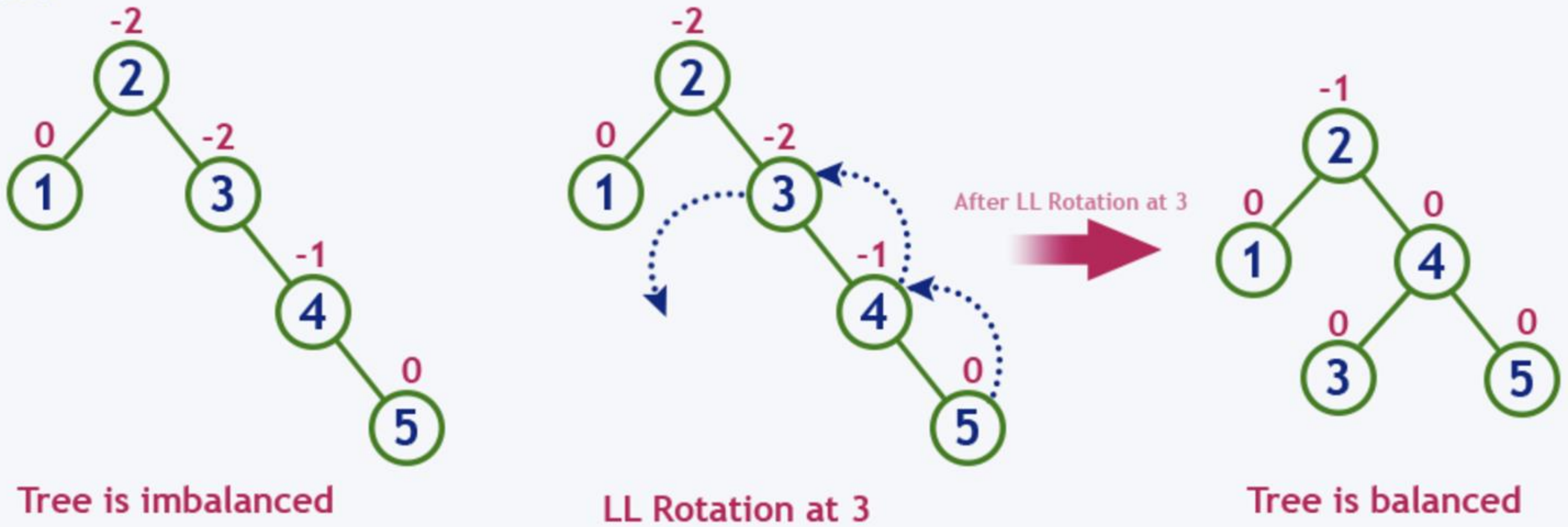
Construct an AVL tree by inserting number from 1 to 8.



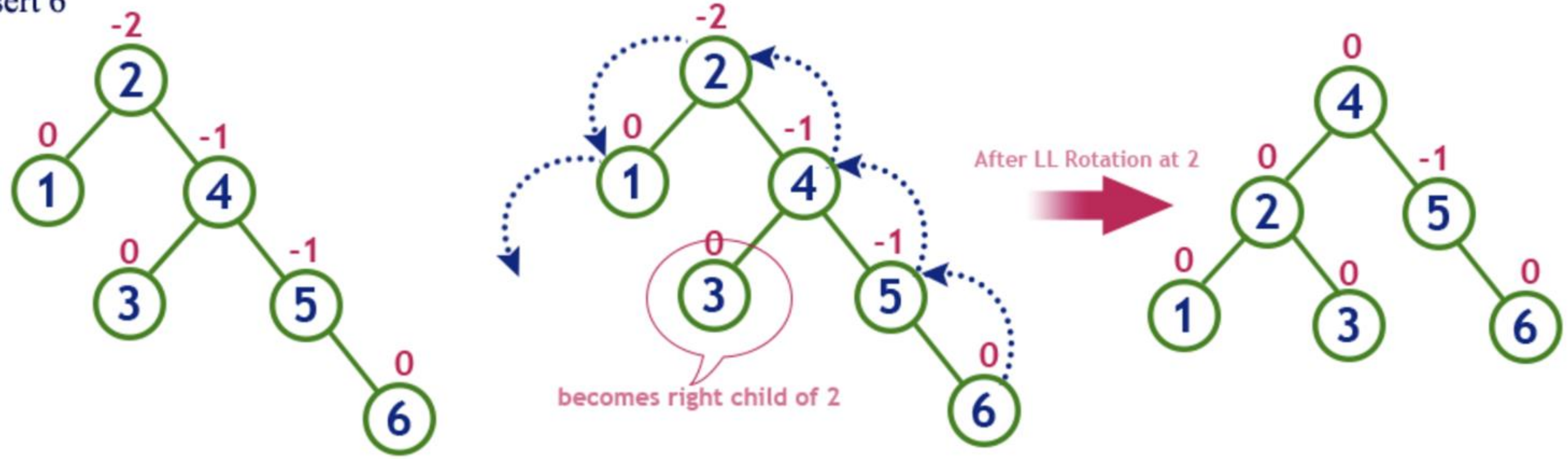
insert 4



insert 5



insert 6

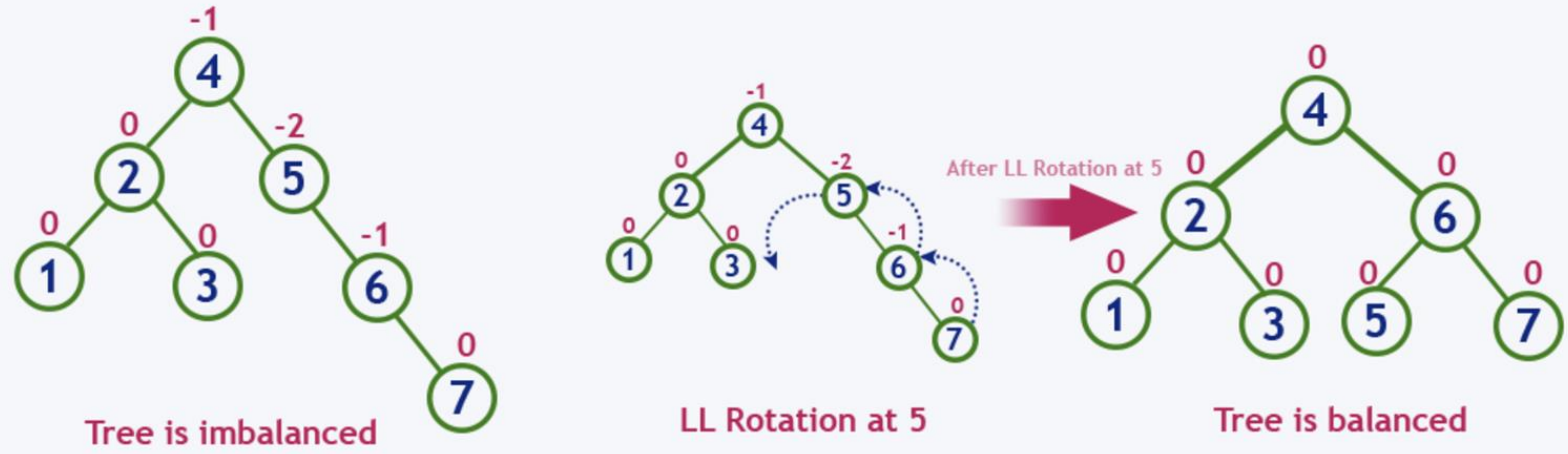


Tree is imbalanced

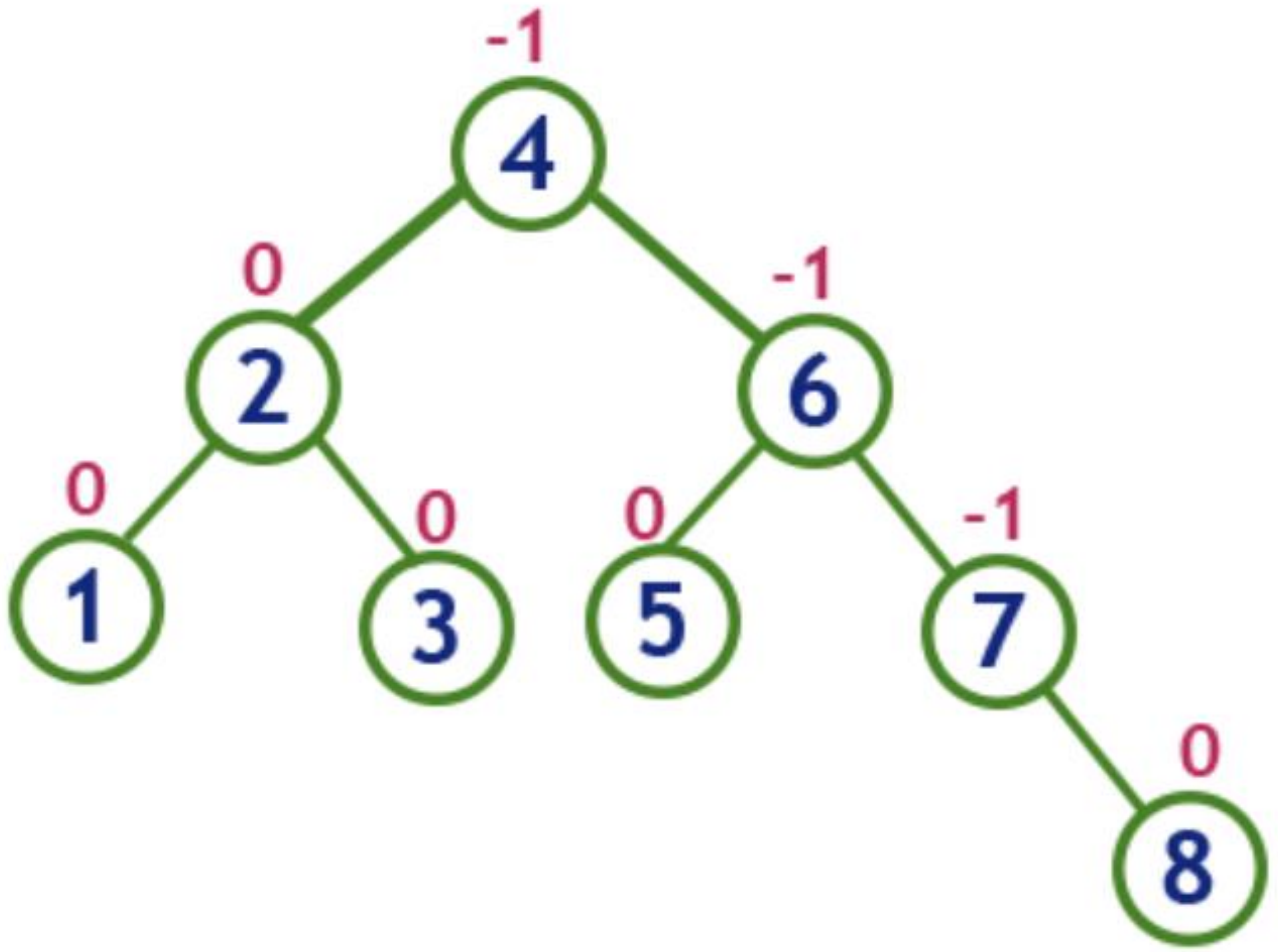
LL Rotation at 2

Tree is balanced

insert 7


-1
4
0
2
-2
5
0
1
0
3
-1
6
0
7
Tree is imbalanced
-1
4
0
2
-2
5
0
1
0
3
-1
6
0
7
LL Rotation at 5
After LL Rotation at 5
0
4
0
2
0
6
0
1
0
3
0
5
0
7
Tree is balanced

insert 8



Tree is balanced

Deletion in AVL trees

The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition.

If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

B-trees

- 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-trees

Here, the number of keys in a node and number of children for a node depends on the order of B-Tree. Every B-Tree has an order.

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.**

Insertion in B-tree

In a B-Tree, a new element must be added only at the leaf node. That means, the new keyValue is always attached to the leaf node only. The insertion operation is performed as follows...

- **Step 1 -** Check whether tree is Empty.
- **Step 2 -** If tree is **Empty**, then create a new node with new key value and insert it into the tree as a root node.
- **Step 3 -** If tree is **Not Empty**, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.
- **Step 4 -** If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.
- **Step 5 -** If that leaf node is already full, **split** that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.
- **Step 6 -** If the spilting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

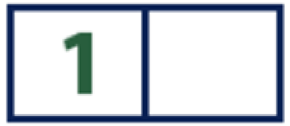
Example

Construct a B-Tree of Order 3 by inserting numbers from 1 to 10.

Construct a B-Tree of order 3 by inserting numbers from 1 to 10.

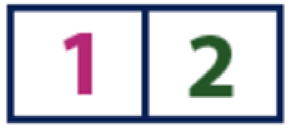
insert(1)

Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.



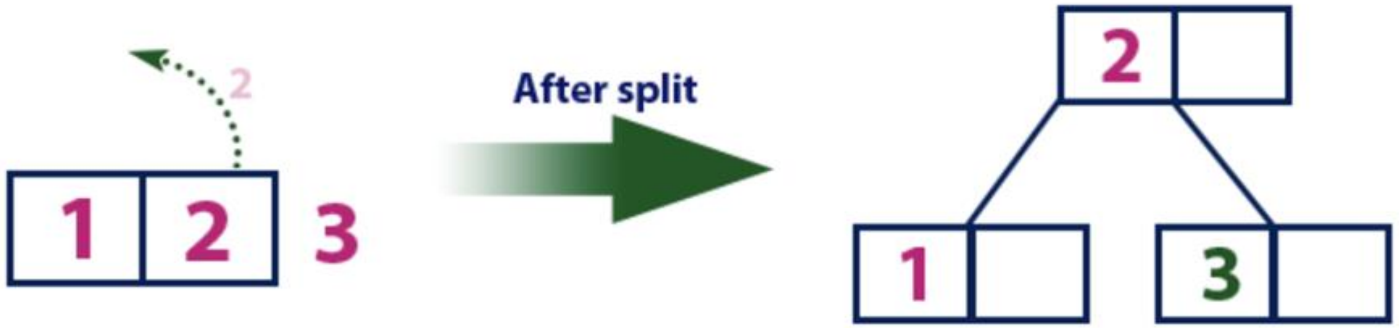
insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.



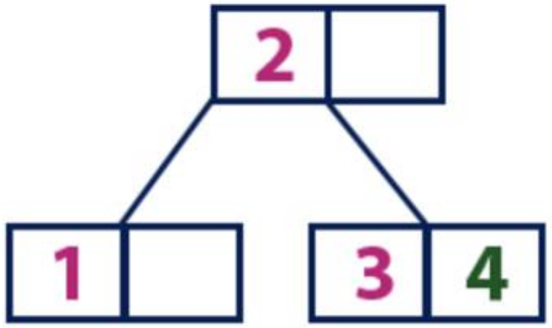
insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't has an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't has parent. So, this middle value becomes a new root node for the tree.



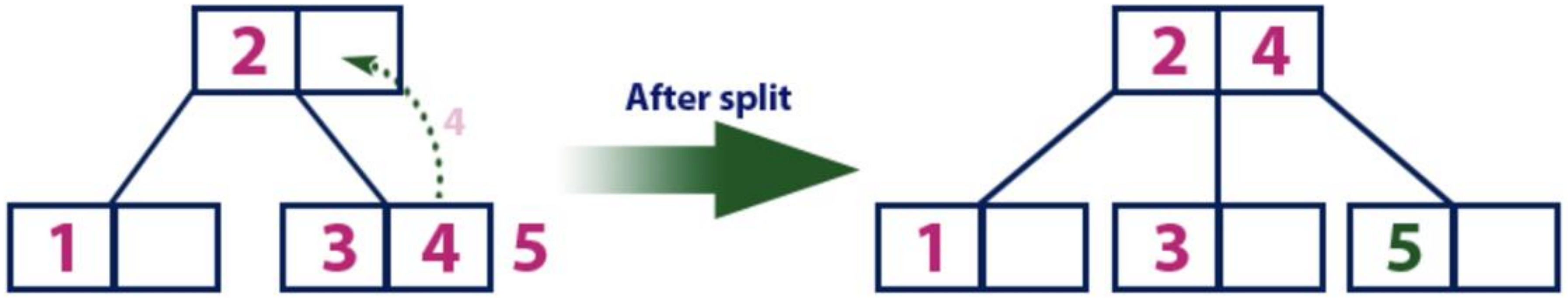
insert(4)

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.



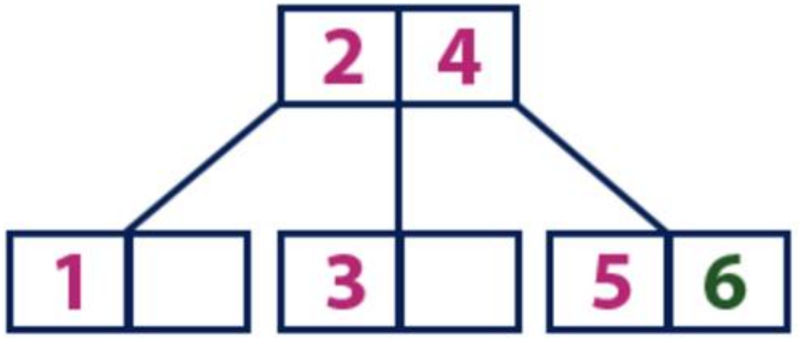
insert(5)

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.



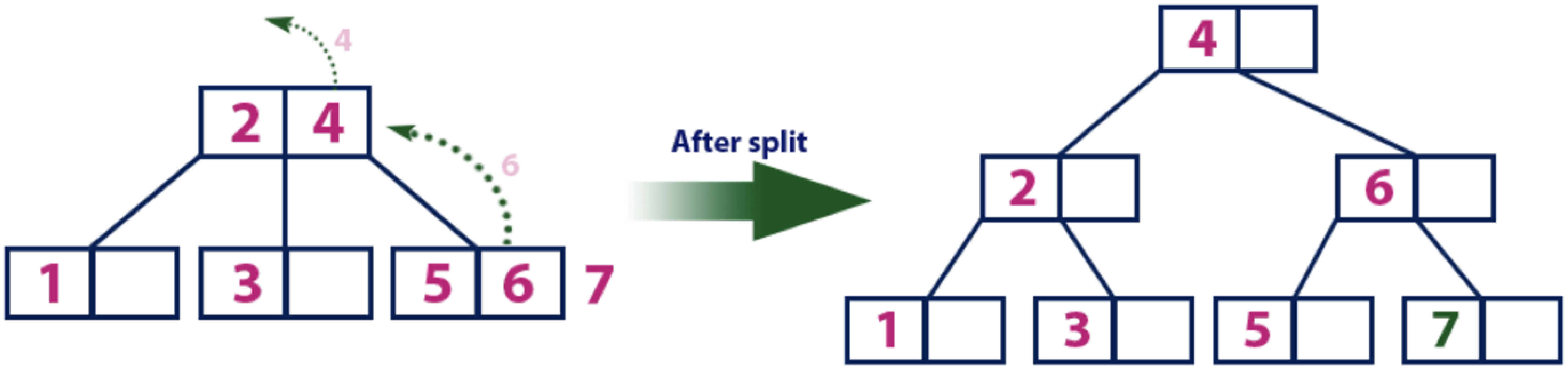
insert(6)

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



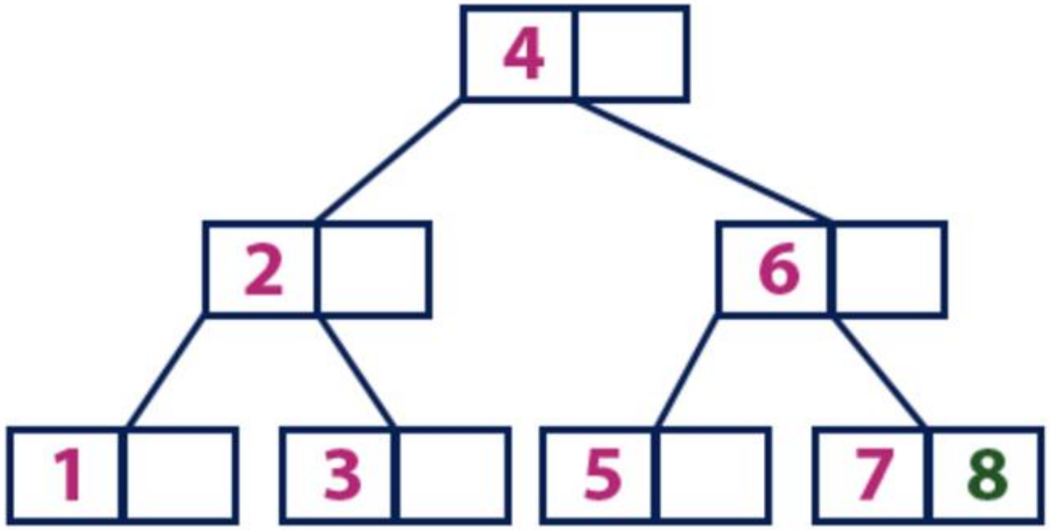
insert(7)

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



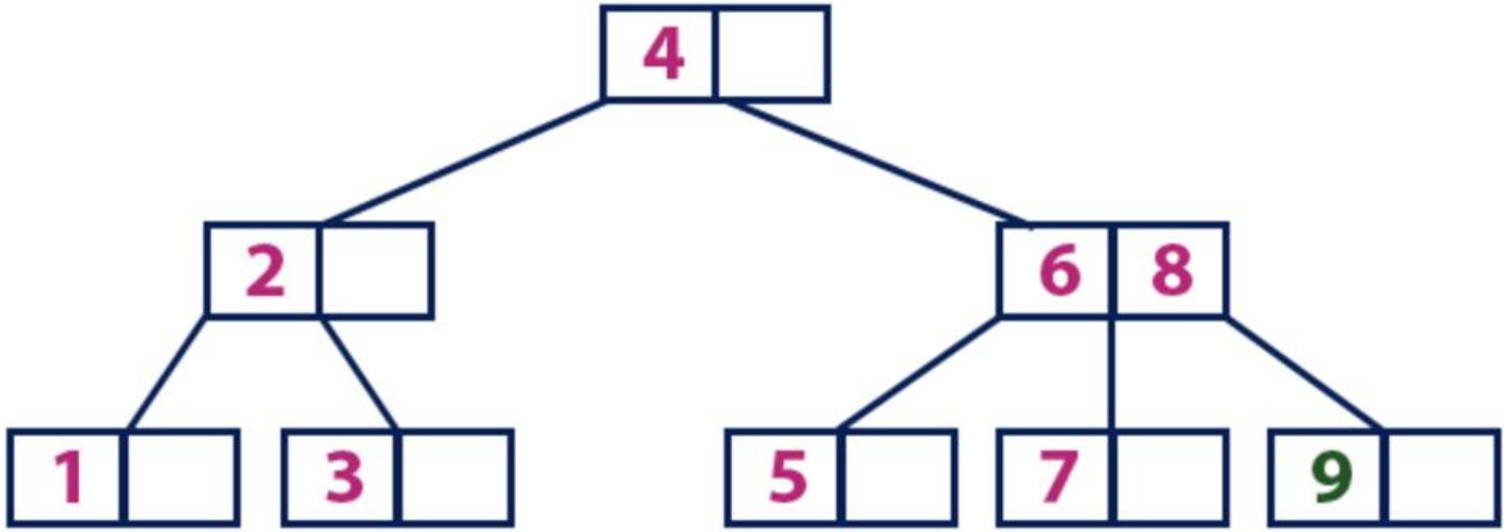
insert(8)

Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.



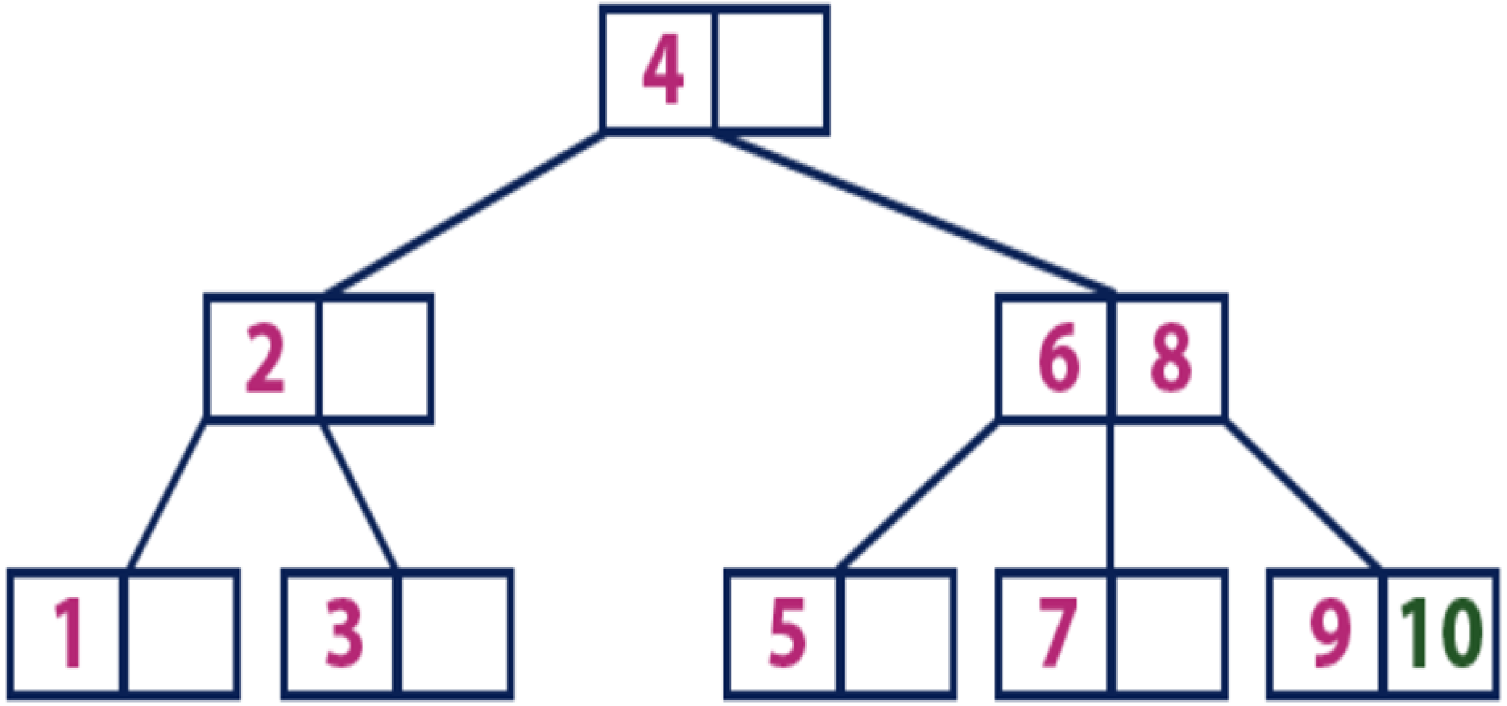
insert(9)

Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



insert(10)

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8 '. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.



Huffman Algorithm

Huffman code is a technique for compressing data. Huffman's greedy algorithm looks at the occurrence of each character and it as a binary string in an optimal way.

Huffman coding

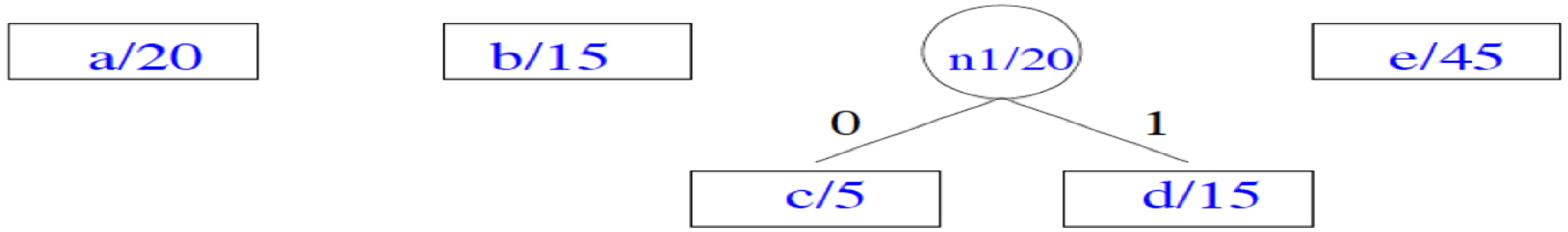
Step 1: Pick two letters x and *y* from Alphabet A with the smallest frequencies and create a subtree that has these two characters as leaves. (greedy idea) Label the root of this subtree as *z.*

Step 2: Set frequency *f(z) = f(x) + f(y) :*Remove *x,y* and add *z* creating new alphabet . A′′ = A ∪{*Z*} – {*x,y*}

Note that mod$|A''| = |A|$ -1. Repeat this procedure, called merge, with new alphabet A′′ until an alphabet with only one symbol is left. Then the resulting tree is the Huffman.

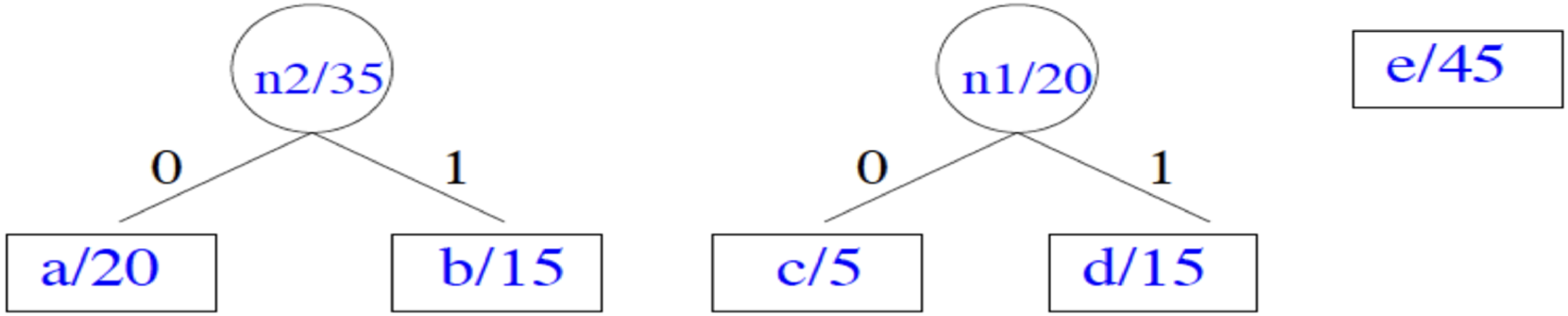
Example of Huffman Coding

Let A = {a/20, b/15, c/5, d/15, e/45} be the alphabet and its frequency distribution. In the first step Huffman coding **merge c and d.**



Huffman Algorithm

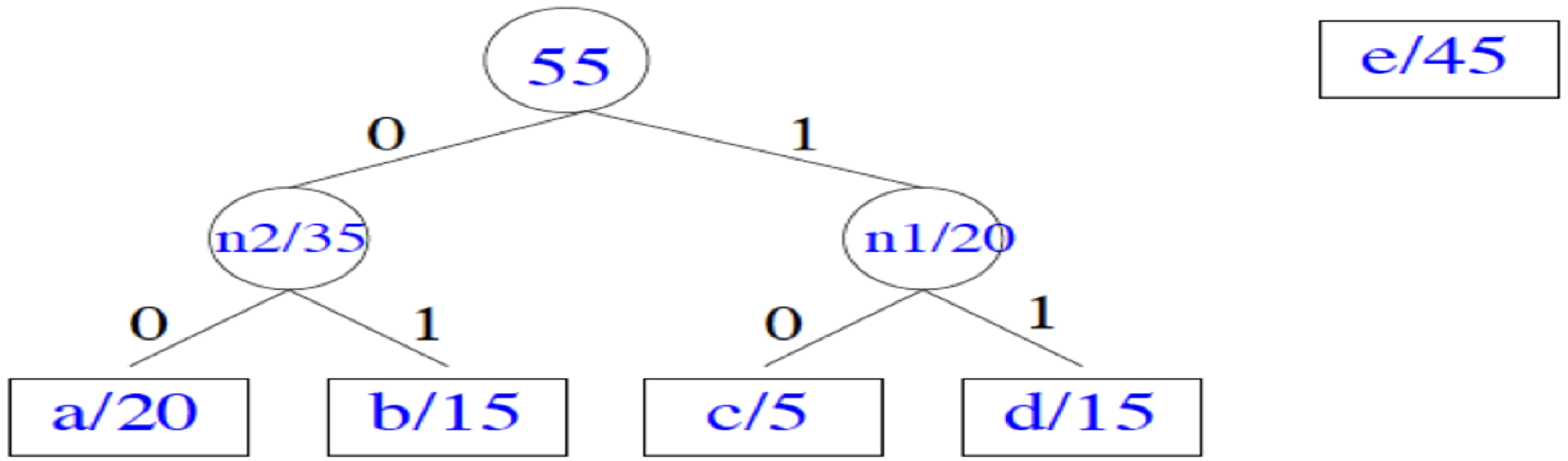
Alphabet is now A1 = {a/20, b/15, n1/20,e/45}. Algorithm **merges a and b**



Alphabet is now A2 = { n2/35,n1/20, e/45}. Algorithm **merge n1 and n2**.

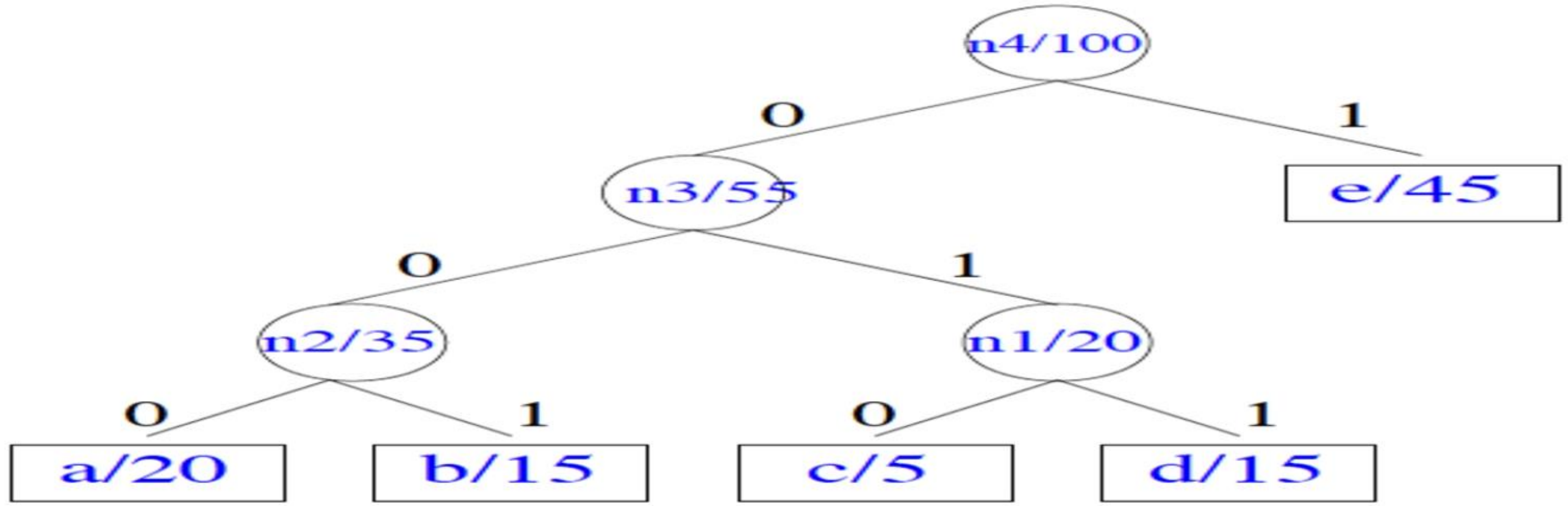
Huffman Algorithm

Alphabet is now A2 = { n2/35,n1/20, e/45}. Algorithm **merge n1 and n2**.



Huffman Algorithm

Alphabet is now A3 = {n3/55, e/45}. Algorithm merges n3 and e and finished.



Huffman code a=000, b=001, c = 010, d= 011, e=1

Game Tree

- A **game tree** is a graph representing all possible game states within such a game. Such games include well-known ones such as chess, checkers, Go, and tic-tac-toe. This can be used to measure the complexity of a game, as it represents all the possible ways a game can pan out.

Game Tree

- A game tree isn't a special new data structure—it's a name for any regular tree that maps how a discrete game is played.
- The computer begins evaluating a move by calculating all possible moves. In Tic-Tac-Toe, it is filling any empty square on the grid, thus the number of possible moves will be equal to the number of empty squares.
- Once it has found these moves, it loops over each of the possible moves, and tries to find out whether the move will result in a win for the computer or not. It does this by running this same algorithm (recursion) over the position (obtained by performing one of the computer's original possible moves) but trying to calculate the *opponent's* best move.

Game Tree

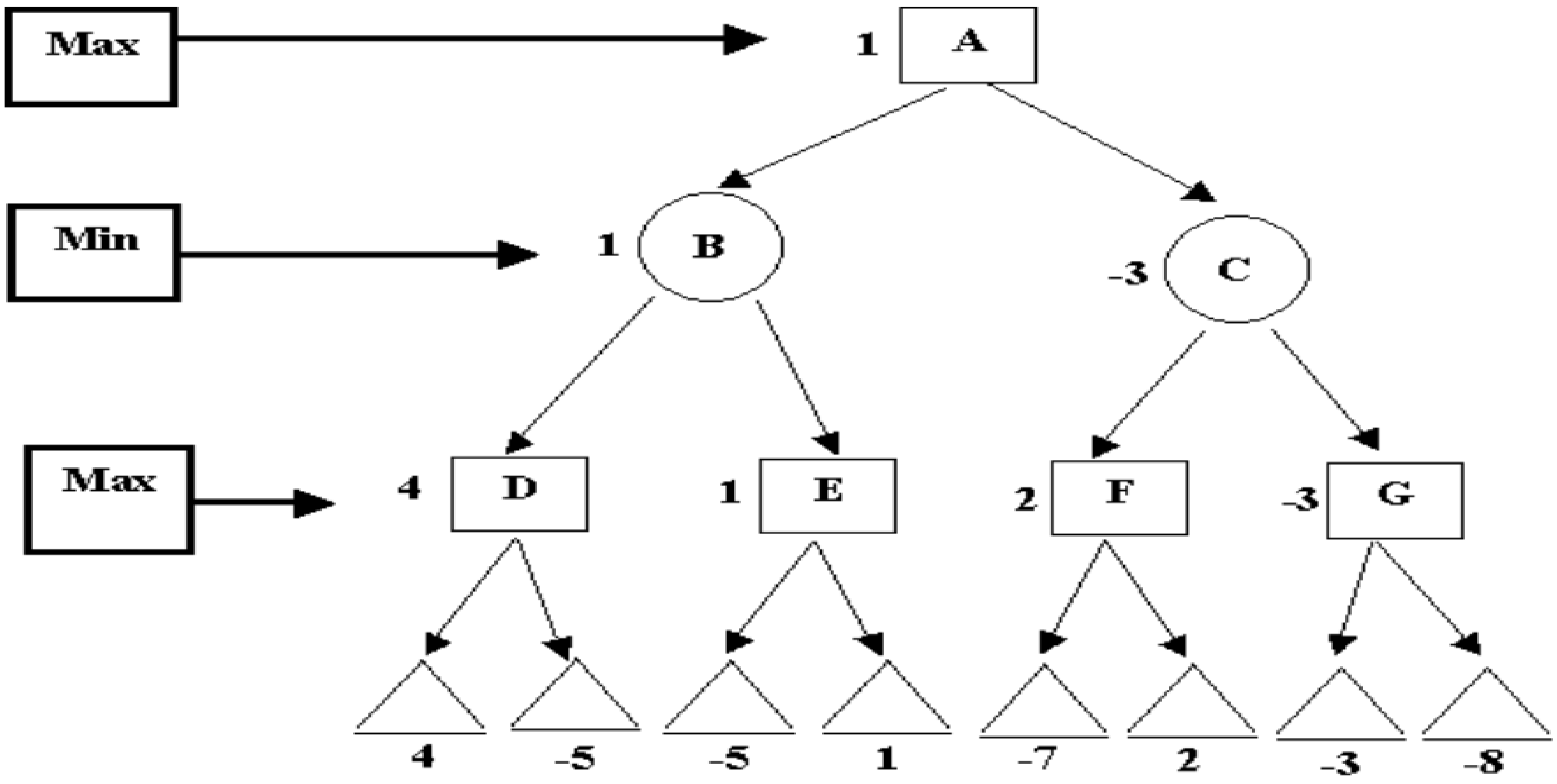
- To find if a move is "good" or "bad" the game tree is extended and continues to branch out until the game ends (a "terminal node").
- Terminal nodes are then assigned values based on the result of the game; the higher the value, the better the result is for the computer.
- Because there are only two possible results (a win or a loss), the values of "-1" and "1" can be used to represent who has won (the example below offers a greater variety of values than "-1" and "1").
- Now that the terminal nodes' values have been determined, the node above them (nodes D, E, F, and G in the picture below).
- If the node represents the computer's choice of moves it is a "max node", if it represents the player's choice of moves it is a "min" node.

Game Tree

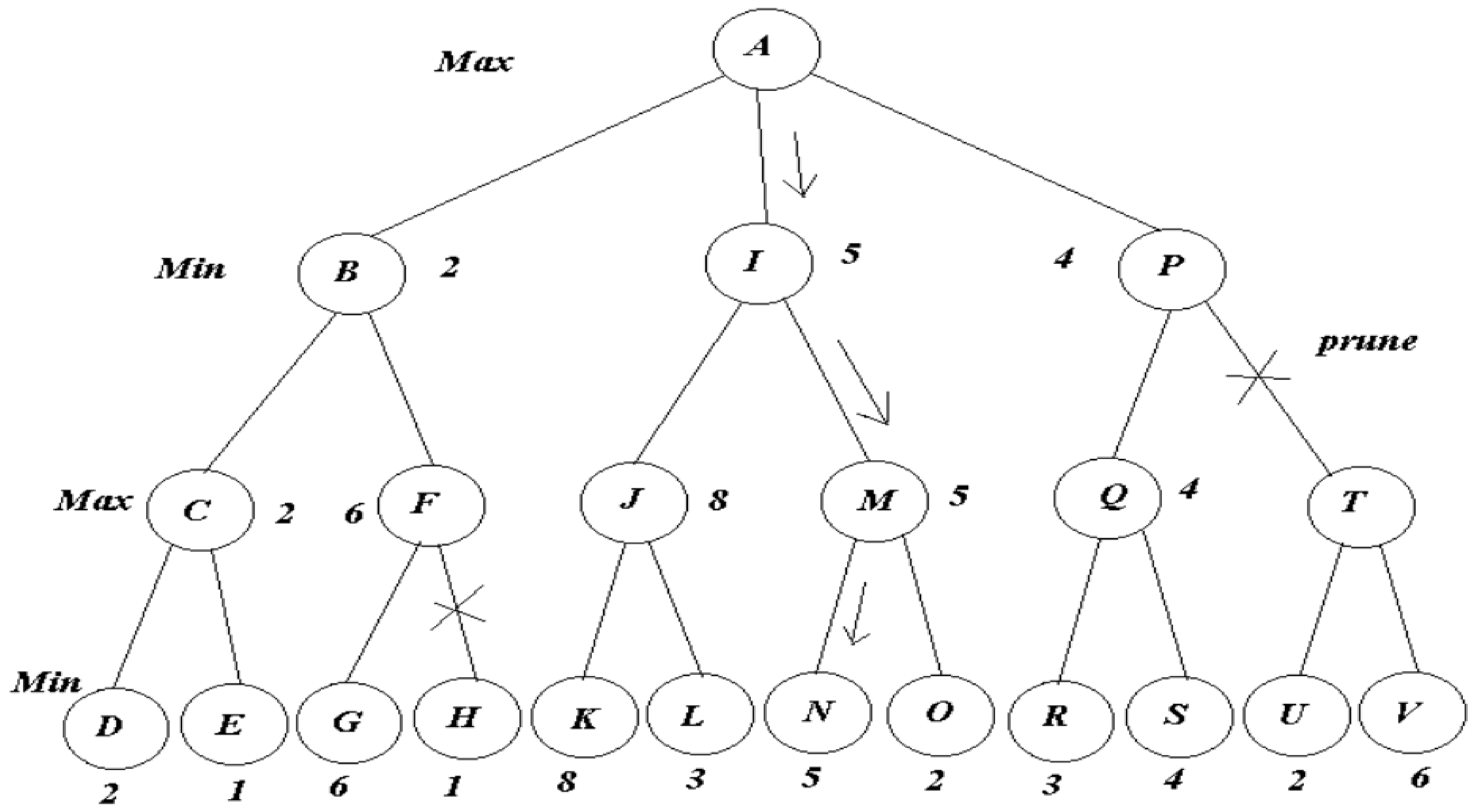
- The value of a max node becomes that of the highest node beneath it.
- The value of a min node becomes that of the lowest node beneath it. The value D in the example below is 4 and the value of E is 1.
- The value of B becomes 1 (the lowest of 4 and 1). By continuing to move up, each node is given a value.
- The top node (A) then assumes the highest value of the nodes beneath it; the node with the highest value is the move that the computer should make.

Note : If the position is that of a finished game, and the winner is the opponent, the position is regarded as a winning position for the opponent (i.e. a "bad" move for the computer), and vice-versa.

Game Tree(Min Max)



Alpha Beta pruning(cut off)



Other Applications

1. *Binary Search Tree*: Used in many search applications where data is constantly entering/leaving, such as the map and set objects in many languages' libraries.

2. *Hash Trees* - used in p2p programs and specialized image-signatures in which a hash needs to be verified, but the whole file is not available.

3. *Heaps* - Used in heap-sort; fast implementations of Dijkstra's algorithm; and in implementing efficient priority-queues, which are used in scheduling processes in many operating systems, Quality-of-Service in routers, and A* (path-finding algorithm used in AI applications, including video games).

4. *Huffman Coding Tree* - used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats.

5. Syntax Tree - Constructed by compilers and (implicitly) calculators to parse expressions.

6. Treap - Randomized data structure used in wireless networking and memory allocation.

7. T-tree - Though most databases use some form of B-tree to store data on the drive, databases which keep all (most) their data in memory often use T-trees to do so

Thank you!!!!!