Unit 6: Tree

a. Concept and Definition
b. Binary Tree
c. Introduction and application
d. Operation
e. Types of Binary Tree
   - Complete
   - Strictly
   - Almost Complete
f. Huffman algorithm
g. Binary Search Tree
   - Insertion
   - Deletion
   - Searching
h. Tree traversal
   - Pre-order traversal
   - In-order traversal
   - Post-order traversal

Concept and Definition

- The implementations of Stacks and Queues form linear data structures.
- They cannot represent data items possessing hierarchical relationship such as between the grandfather and his descendants and in turns their descendants and so on.
- We need non-linear data structures to deal with such applications in real life situations.
- Trees and graphs are two examples of non-linear data structures.

Concept and Definition- Tree

- A tree is a non-linear data structure in which items are arranged in a sorted sequence. It is used to represent hierarchical relationship existing amongst several data items.
- A tree is defined as a finite set of one or more data items (nodes) such that
  - There is a special data item called the **root** of the tree.
  - And its remaining data items are partitioned into a number of mutually exclusive (i.e. disjoint) subsets, each of which itself is a tree (called **subtrees**).
- Tree data structure grows downwards from top to bottom.

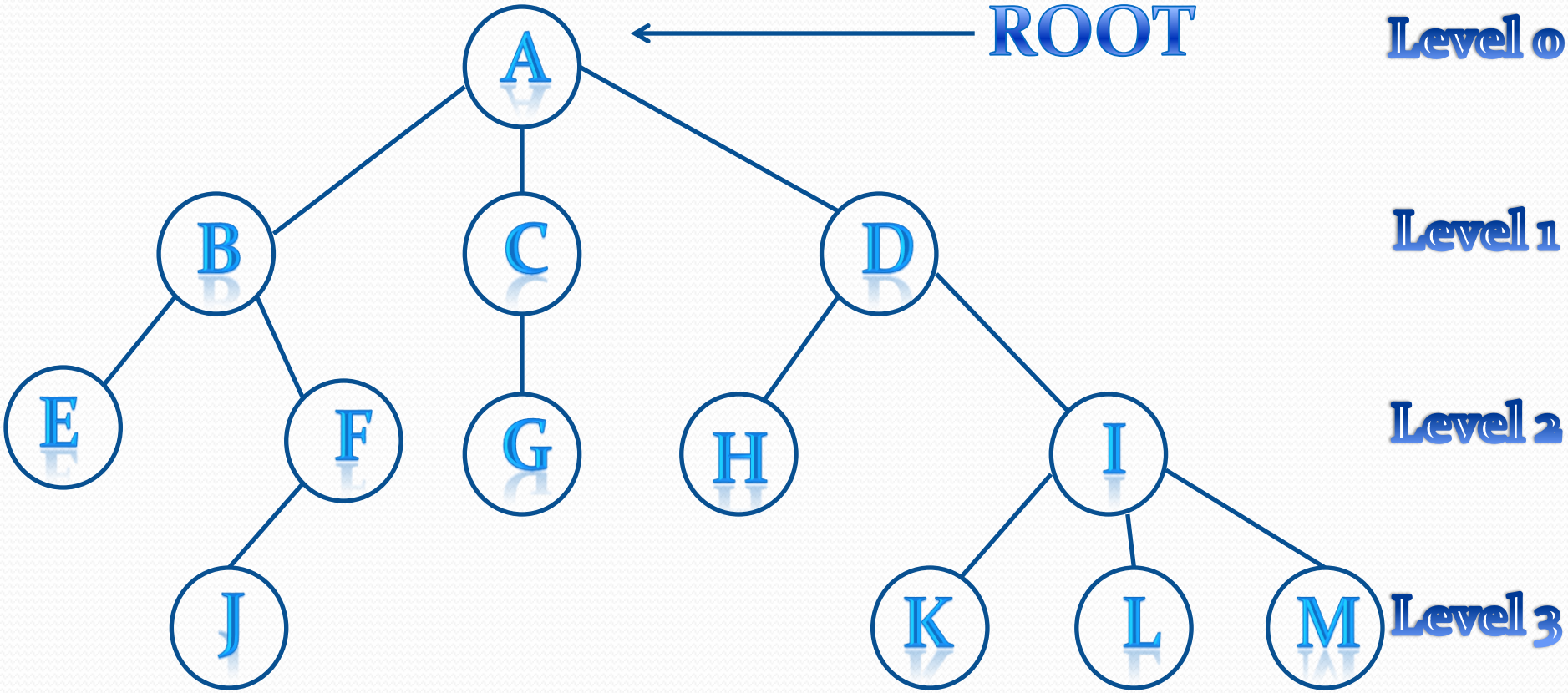


Fig: A Tree

Terminologies

- ***Node:*** Each data item in a tree is called a node. It is the basic structure in a tree. It specifies the data and links (branches) to other data items. There are 13 nodes in the above tree.

- ***Root:*** It is the first data item (or node) in the hierarchical arrangement of data items. In the above tree, **A** is the root item.

- ***Degree of a node:*** It is the number of subtrees of a node in a given tree. For example, in the above tree,
  - The degree of node A is 3
  - The degree of node B is 2
  - The degree of node C is 1
  - The degree of node H is 0
  - The degree of node I is 3

Terminologies…

- ***Degree of a tree:*** It is the maximum degree of nodes in a given tree. In the above tree, the degree of node A is 3 and another node I also have degree 3. In the whole tree, this value is the maximum. So, the degree of the above tree is 3.

- ***Terminal node(s):*** A node with degree zero is called a **terminal node** or **leaf**. In the above tree, there are 7 terminal nodes: **E**, **J**, **G**, **H**, **K**, **L** and **M**.

- ***Non-terminal node(s):*** Any node (except the root node) whose degree is not zero is called **non-terminal node**. **Non-terminal nodes** are the intermediate nodes in traversing the given tree from its root node to the terminal nodes (leaves). In the above tree, there are 5 **non-terminal nodes**: **B**, **C**, **D**, **F** and **I**.

Terminologies…

- ***Siblings:*** The children nodes of a given parent node are called **siblings**. They are also called **brothers**. In the above tree,
  - **E** and **F** are siblings of parent node **B**
  - **K**, **L** and **M** are siblings of parent node **I**

- ***Level:*** The entire tree structure is leveled in such a way that the root node is always at level 0. Then its immediate children are at level 1 and their immediate children are at level 2 and so on up to the terminal nodes. In general, if a node is at level n, then its children are at level n+1. In the above tree, there are four levels from level 0 to level 3.

- ***Edge:*** It is a connecting line of two nodes. That is, the line drawn from one node to another node is called an **edge**.

Terminologies…

- ***Path:*** It is a sequence of consecutive edges from the source to the destination node. In the above tree, the **path** between **A** and **J** is given by the node pairs: **(A, B)**, **(B, F)** and **(F, J)**.

- ***Depth:*** It is the maximum level of any leaf in a given tree. This equals the length of the longest path from root to the terminal nodes (leaves). The term **height** is also used to denote the depth. The **depth** of the above tree is 3.

- ***Forest:*** It is a set of disjoint trees. In a given tree, if we remove its root, then it becomes a forest. In the above tree, there is forest with three trees if we remove the root node.

Binary Tree

- A binary tree is the most commonly used non-linear data structure.
- A binary tree is a finite set of data items which is either empty or is partitioned into three disjoint subsets. The first subset consists of a single data item called the **root** of the binary tree. The other two subsets are themselves binary trees called the **left subtree** and **right subtree** of the original binary tree.
- In a binary tree, the maximum degree of any node is **at most two**. This means that there may be a zero degree node (usually an empty tree or leaf) or a one degree node or a two degree node.

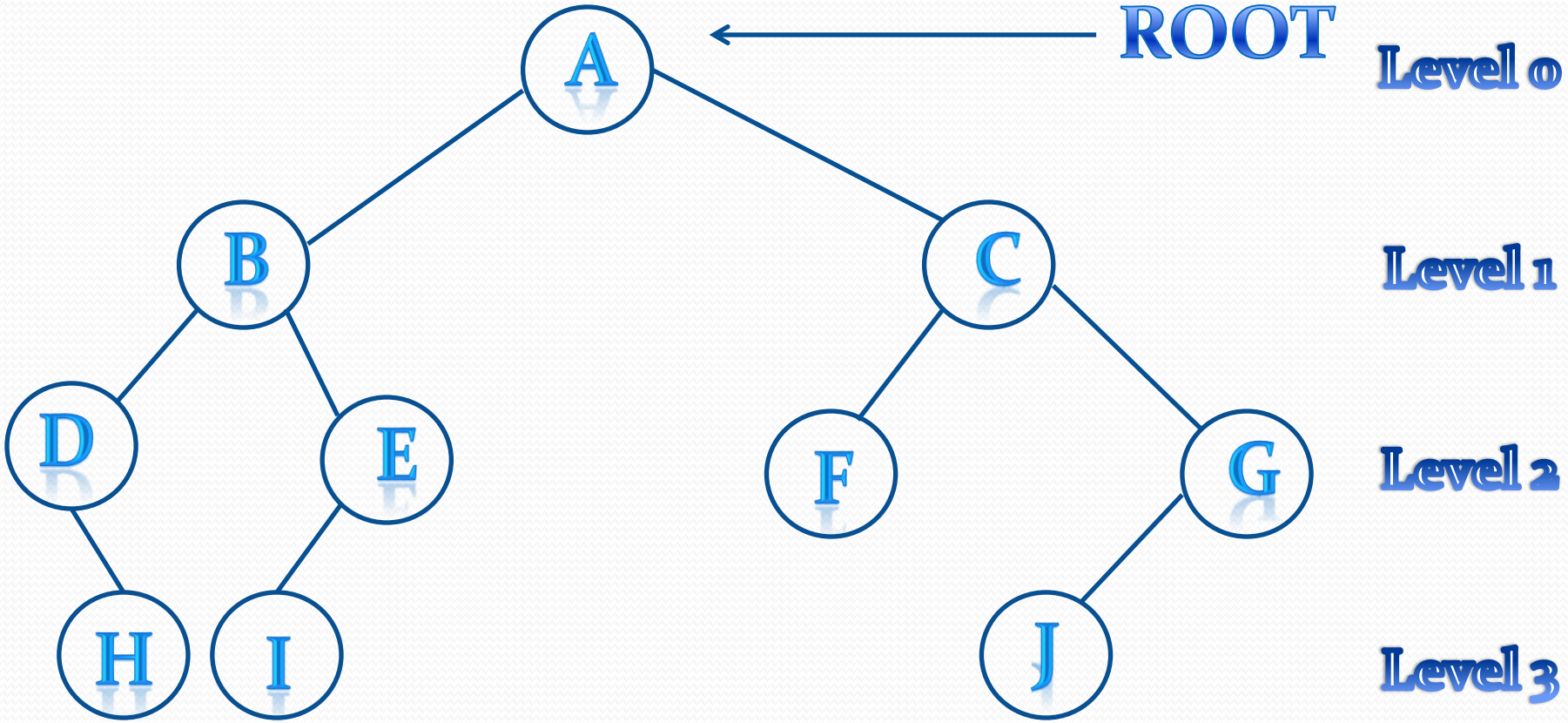


Fig: A Binary Tree

- In the above binary tree, **A** is the **root** of the **binary tree**. The **left subtree** consists of the tree with root **B** and the **right subtree** consists of the tree with root **C**. Further **B** has its **left subtree** with root **D** and **right subtree** with root **E**. Similarly, **C** has its **left subtree** with root **F** and its **right subtree** with root **G**. In the next level, **D** has an empty **left subtree** and its **right subtree** with root **H**. Similarly, **E** has a **right subtree** with root **I** and has no **left subtree**. **F** has neither its **left subtree** nor its **right subtree**. Finally, **G** has its **left subtree** with root **J** and it has no **right subtree**.

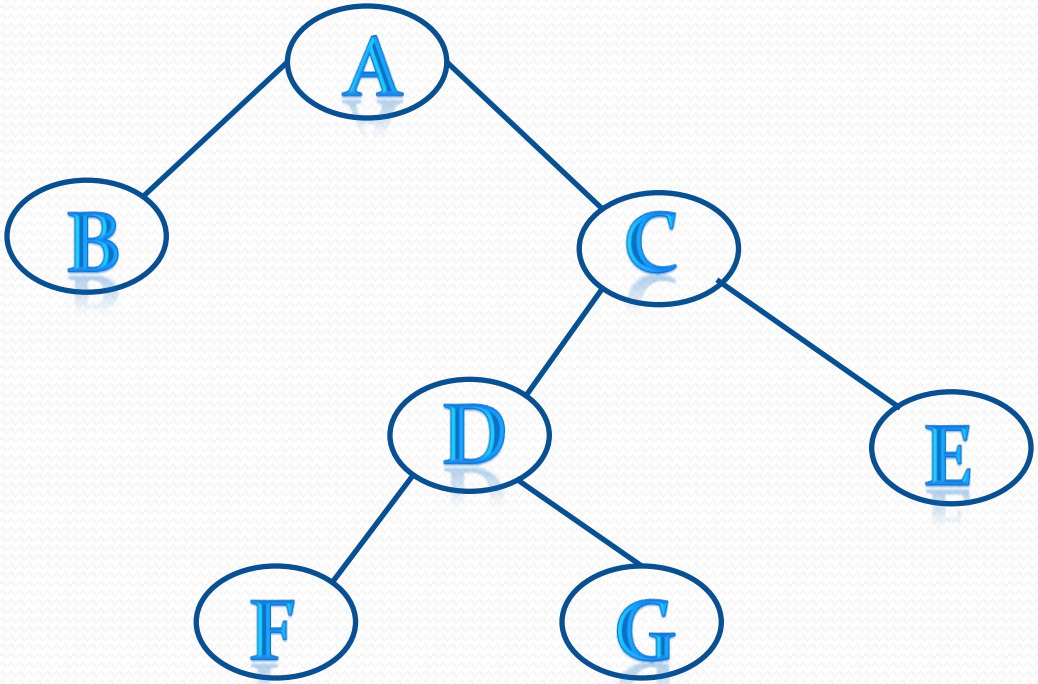
- ***Note:***
- If **A** is the root of a binary tree and **B** is the root of its left or right subtree, then **A** is called **father** of **B** and **B** is called **left** or **right son** of **A**.
- A node that has no sons (such as **H**, **I**, **F** or **J**) is called a **leaf**.
- Node n1 is an **ancestor** of node n2 (and n2 is a **descendant** of n1) if n1 is either the father of n2 or the father of some ancestor of n2. For example, in the above binary tree **A** is an ancestor of **I**, and **J** is a descendant of **C**, but **E** is neither an ancestor nor a descendant of **C**.
- A node n2 is a **left descendant** of node n1 if n2 is either the left son of n1 or a descendant of the left son of n1. Similarly **right descendant** can be defined**.**

Types of Binary Trees

- **Strictly Binary Tree**
- **Complete Binary Tree**
- **Almost Complete Binary Tree**

Strictly Binary Tree

- If every non-terminal node in a binary tree consists of non-empty left and right subtrees, then such a tree is called **strictly binary tree**.



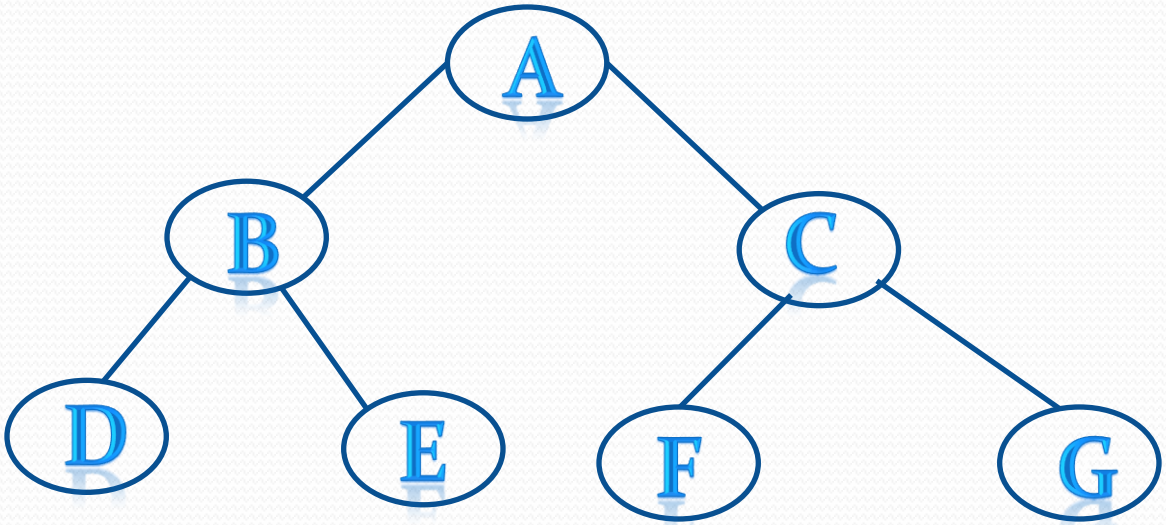
In this binary tree, all the non-terminal nodes such as C and D have non-empty left and right subtrees.

Note: A strictly binary tree with n leaves always contains (2n-1) nodes.

Fig: A Strictly Binary Tree

Complete Binary Tree

- A **complete binary tree** of depth ***d*** is the **strictly binary tree** all of whose leaves are at level ***d***.
- In a **complete binary tree**, there is exactly one node at level 0, two nodes at level 1, four nodes at level 2 and so on.



Note: A complete binary tree of depth d contains a total of $(2^{d+1}-1)$ nodes with 2^d leaves at level d and 2^d-1 non-leaf nodes.

Fig: A Complete Binary Tree

Almost Complete Binary Tree

- A binary tree of depth ***d*** is an **almost complete binary tree** if:
  1) Any node ***nd*** at level less than ***d-1*** has two sons.
  2) For any node ***nd*** in the tree with a **right descendant** at level ***d***, ***nd*** must have a **left son** and every **left descendant** of ***nd*** is either a **leaf** at level ***d*** or has **two sons**.
- The nodes of an **almost complete binary tree** are numbered in a way that the **root** is assigned the number 1, a **left son** is assigned twice the number assigned to its **father**, and a **right son** is assigned one more than twice the number assigned to its **father**.

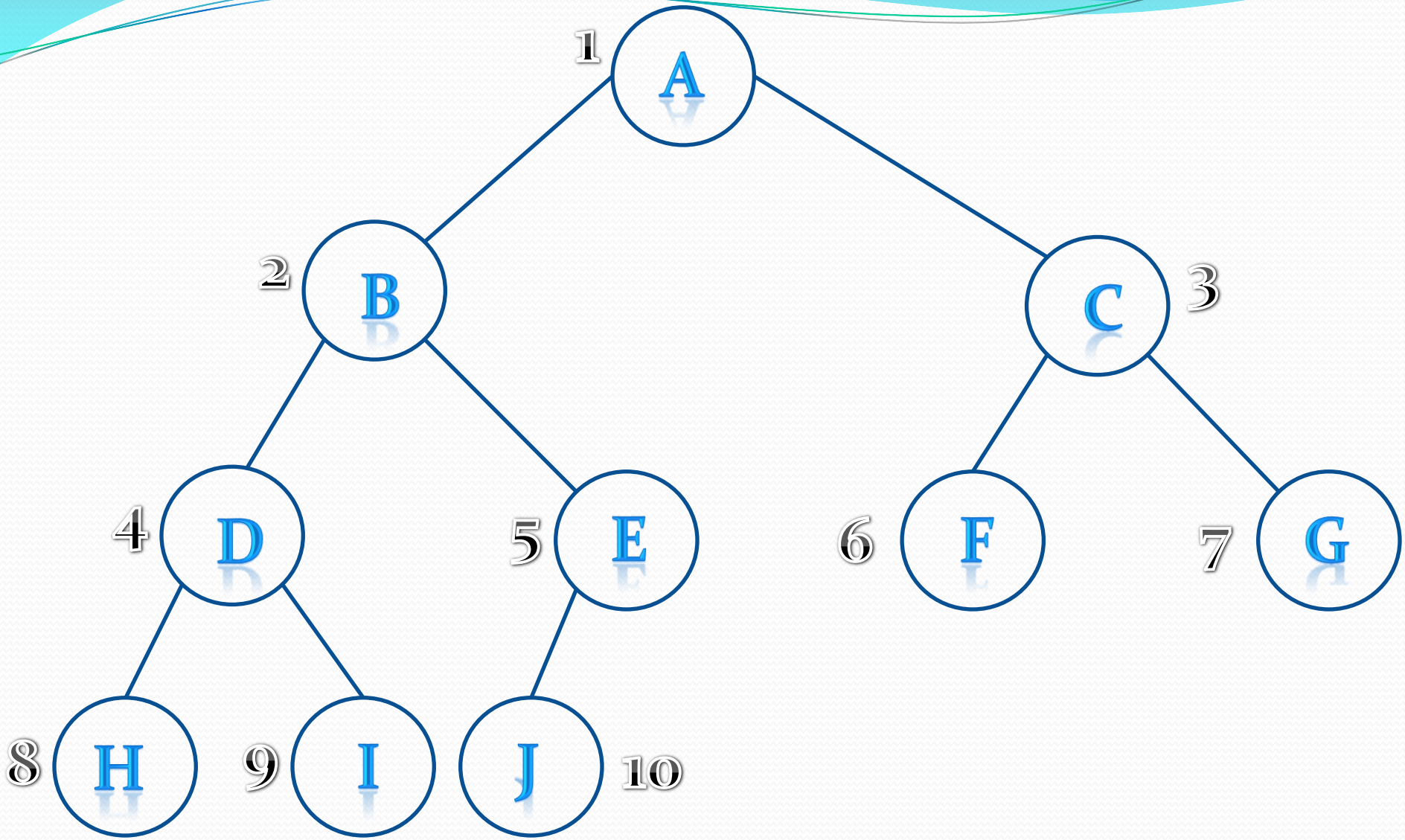
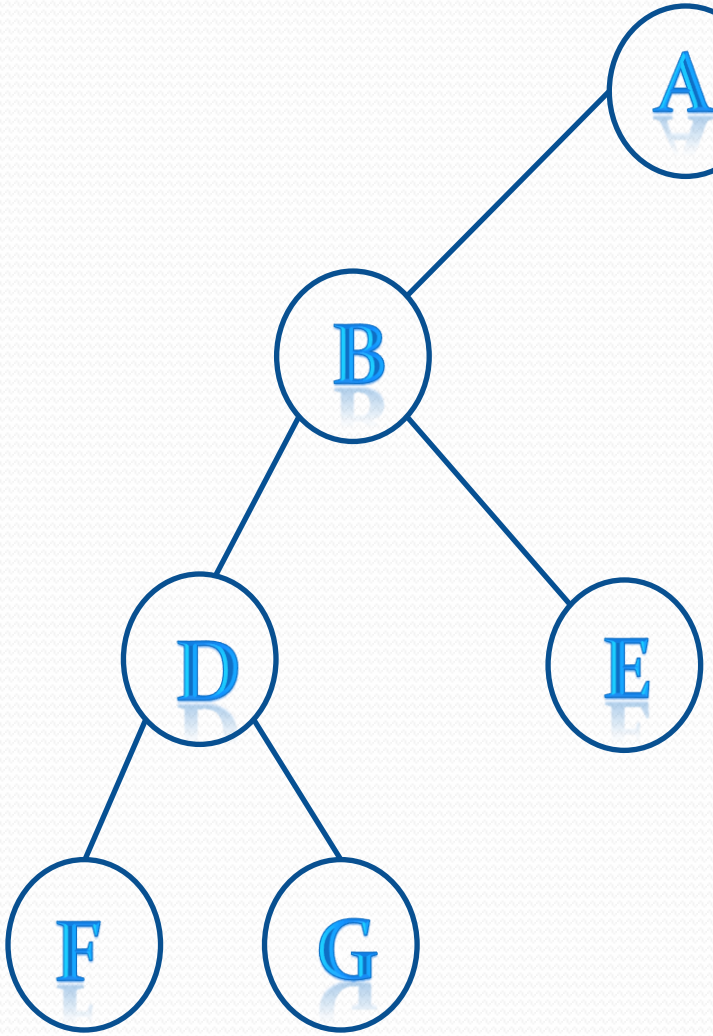


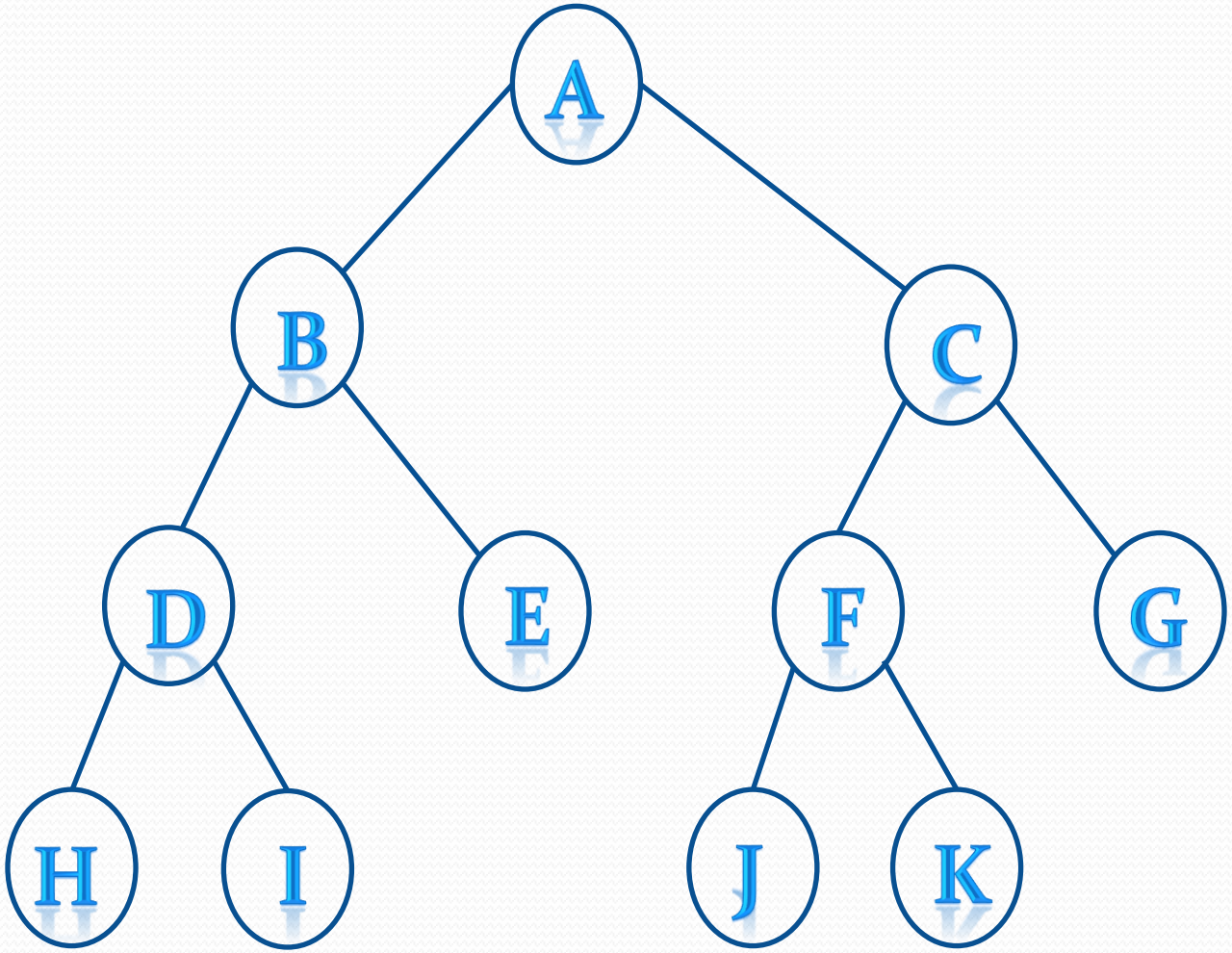
Fig: An Almost Complete Binary Tree

Is the following an **almost complete binary tree**?



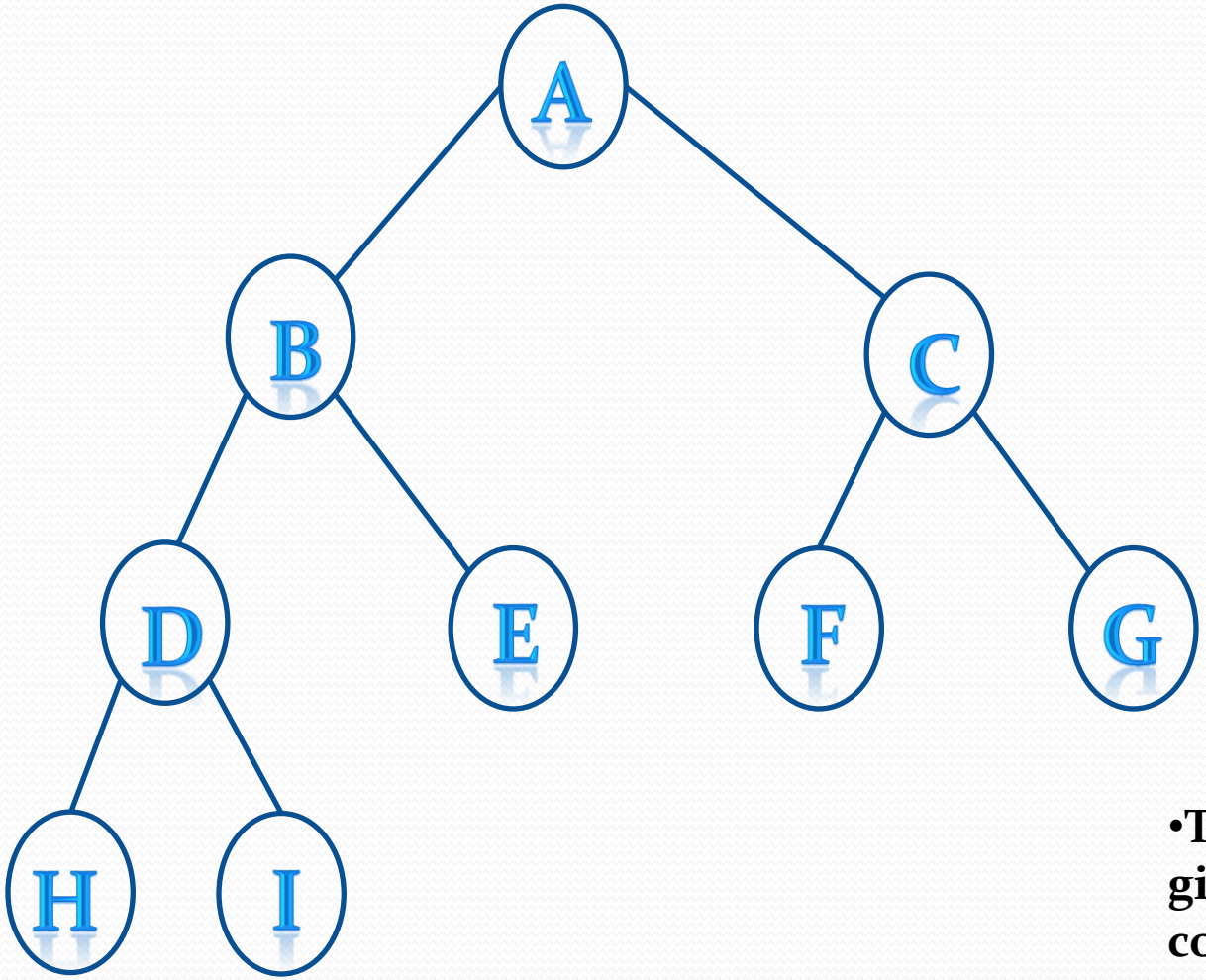
- **The binary tree given here is not almost complete because it contains leaves at levels 1, 2 and 3 thereby violating condition 1.**
- **But it is a strictly binary tree.**

Is the following an **almost complete binary tree**?



•**The strictly binary tree given here satisfies condition 1 , since every leaf is either at level 2 or at level 3. However, condition 2 is violated, since A has a right descendant at level 3 (J) but also has a left descendant that is a leaf at level 2 (E).**

Is the following an **almost complete binary tree**?



•**The strictly binary tree given here satisfies both conditions for an almost complete binary tree.**

Model Question (2008)

- Explain different types of binary tree.

Binary Search Tree (BST)

- A **binary search tree** is a ***binary tree*** that is either empty or in which each node possesses a key that satisfies the following three conditions:
  i. For every node X in the tree, the values of all the keys in its left subtree are smaller than the key value in X.
  ii. Keys in its right subtree are greater than the key value in X.
  iii. The left and right subtrees of the root are again binary search trees.

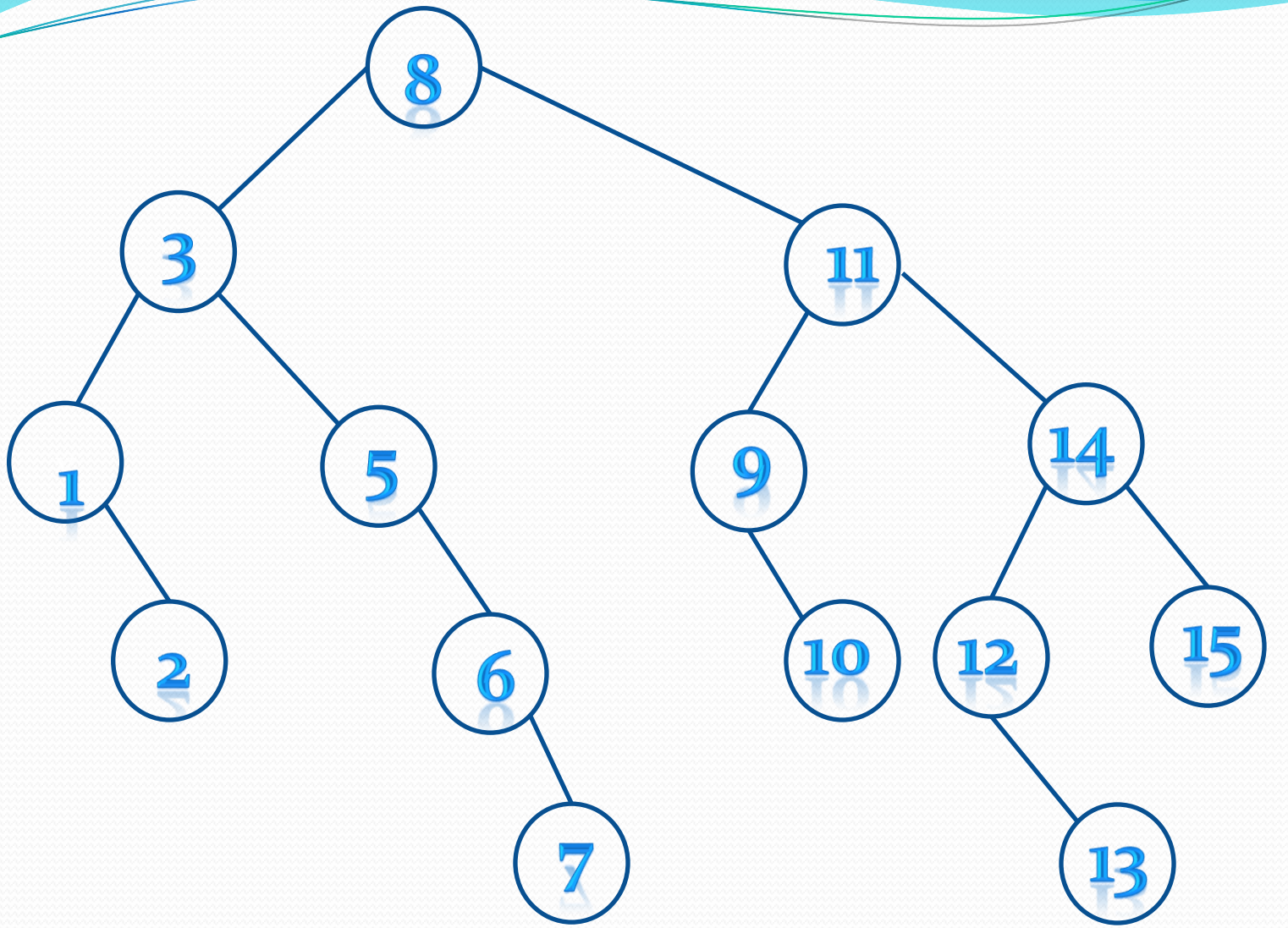


Fig: A Binary Search Tree

Operations on BST

- **Insertion:** Inserting a node into **BST** is done in two steps:
  I. The tree should be searched to determine where the node is to be inserted.
  II. On the completion of search, the node is inserted into the tree.
- **Deletion:** While deleting a node from **BST**, there are two major tasks to be done: First search or locate the node to be deleted and then delete.
- **Searching:** Traverse the whole **BST** with the key value comparing with the keys on all the nodes of **BST** until match is found. If the match is not found, then display "Unsuccessful search".

C Implementation of BST

- For the implementation of a binary search tree, a ***binarysearchtree*** template is defined using a self-referential structure in linked list format and after this, nodes of the binary search tree are inserted and removed using dynamic memory allocation functions like ***malloc()*** and ***free()***. We consider ***rootnode*** as an external pointer that keeps the address of the root node of the binary search tree. The following code defines the structure for binary search tree and creates a ***rootnode***.

```
struct binarysearchtree
        {
        int info;
        struct binarysearchtree *leftlink;
         struct binarysearchtree *rightlink;
        };
struct binarysearchtree *rootnode;
rootnode=NULL;
```

- Whenever ***rootnode=NULL***, the binary search tree is empty.

Insertion in BST

I. ***Inserting a node into an empty tree***: In this case, the node inserted into the tree is considered as the root node.

II. ***Inserting a node into a non-empty tree***: In this case, we compare the new node to the root node of the tree.

   a. If the value of the new node is less than the value of the root node, then if the left subtree is empty, the new node is appended as the left leaf of the root node else we search continuous down the left subtree.

   b. If the value of the new node is greater than the value of the root node, then if the right subtree is empty, the new node is appended as the right leaf of the root node else we search continuous down the right subtree.

   c. If the value of the new node is equal to the value of the root node, then print "**DUPLICATE ENTRY**" and return.

Recursive Function for Insertion in BST

Function Call: ***insertion(&rootnode,item);***

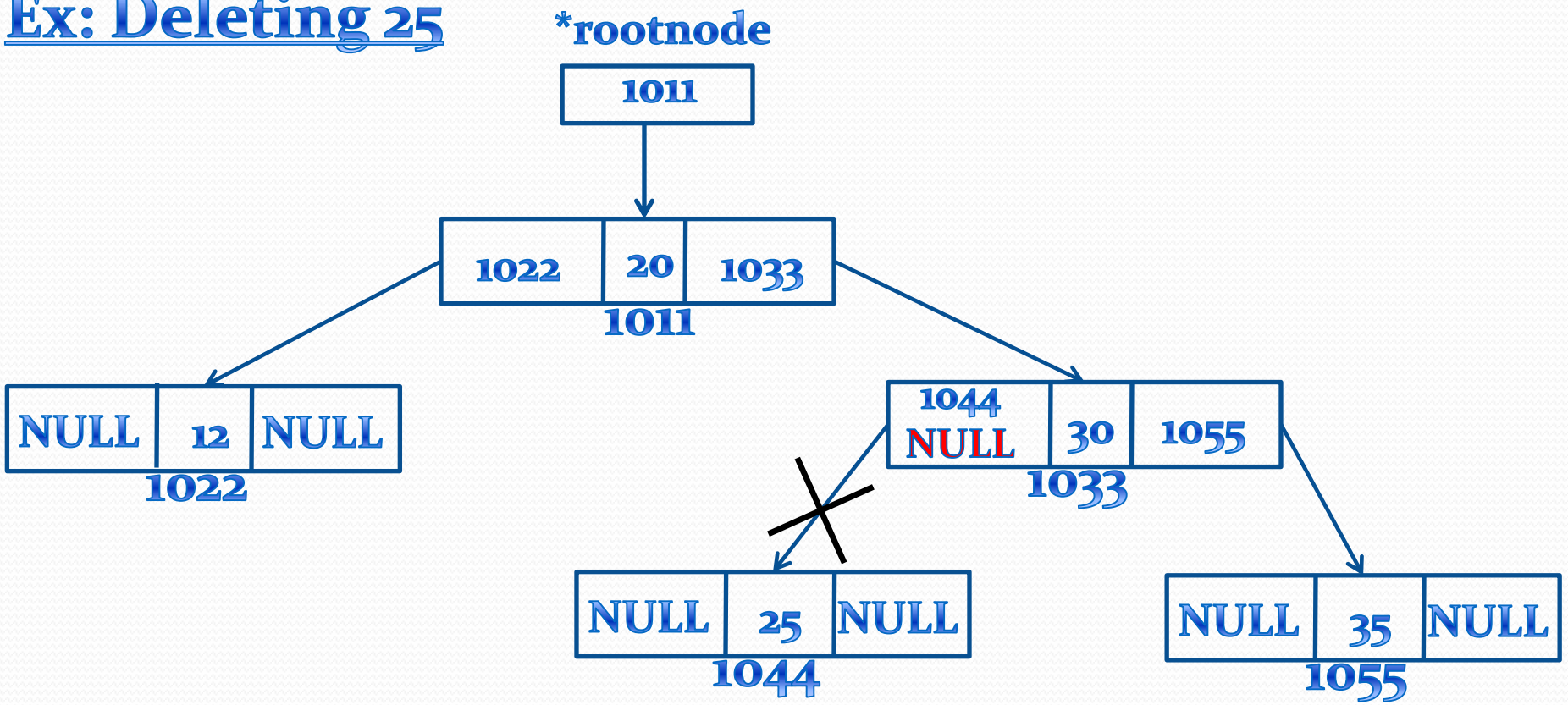
Function Definition:

```
void insertion(struct binarysearchtree **root, int x)
{
struct binarysearchtree *temp,*node;
temp=*root;
if(temp==NULL)                    //empty tree
      {
      node=(struct binarysearchtree *)malloc(sizeof(struct
                  binarysearchtree));
      node->info=x;
      node->leftlink=NULL;
      node->rightlink=NULL;
      *root=node;
      }
else if(temp->info>x)
      {
      if(temp->leftlink==NULL)
                  {
                  node=(struct binarysearchtree *)malloc(sizeof(struct
                        binarysearchtree));
                  node->info=x;
                  node->leftlink=NULL;
                  node->rightlink=NULL;
                  temp->leftlink=node;
                  }
      else
                  insertion(temp->leftlink, x);
      }

else if(temp->info<x)
      {
      if(temp->rightlink==NULL)
                  {
                  node=(struct binarysearchtree *)malloc(sizeof(struct
                        binarysearchtree));
                  node->info=x;
                  node->leftlink=NULL;
                  node->rightlink=NULL;
                  temp->rightlink=node;
                  }
      else
                  insertion(temp->rightlink, x);
      }

else
      {
      printf("\n DUPLICATE ENTRY!!!");
      return;
      }
}
```

Deletion in BST

- The node to be deleted may be a ***leaf node*** or a ***node with one child*** or a ***node with two children***.
  - *a)* ***Deleting a leaf node:*** In this case, the node is simply deleted from the tree and **NULL** is set to its parent pointer.

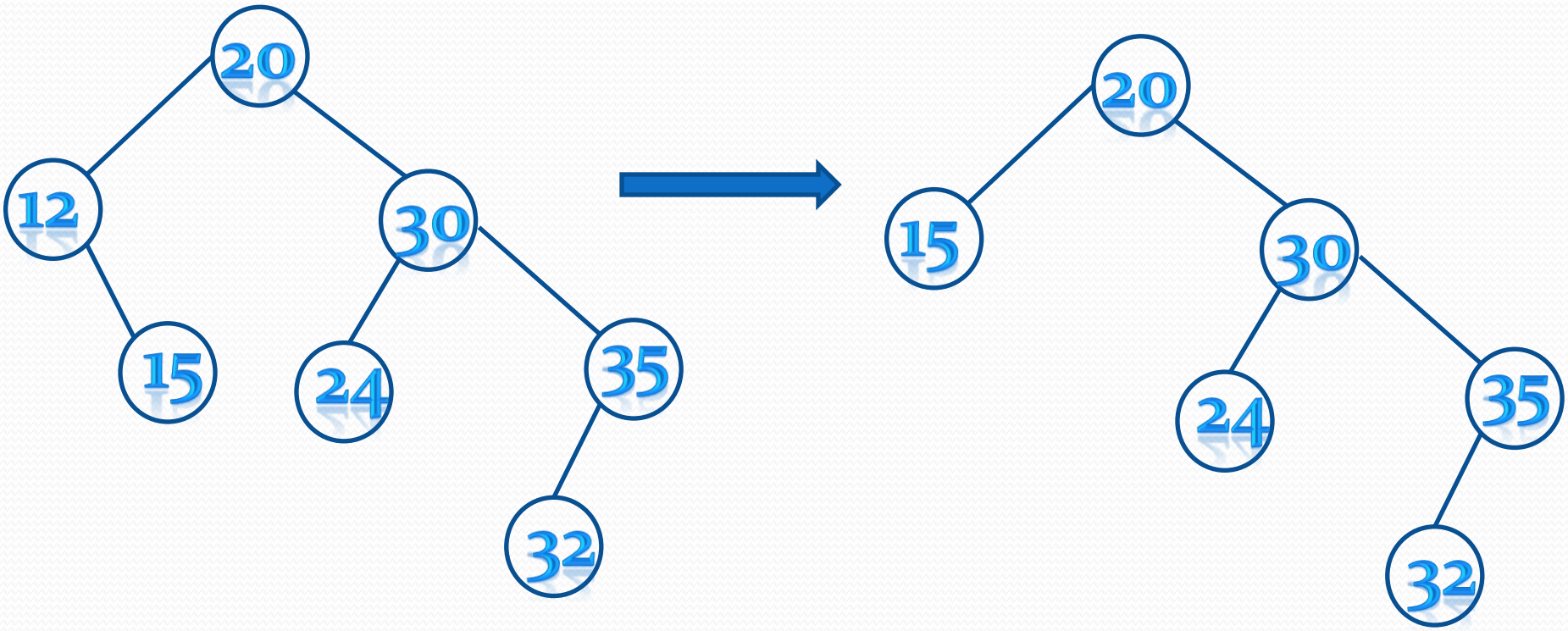
Ex: Deleting 25



Deletion in BST…

b) ***Deleting a node with one child:*** In this case, the child of the node is appended to its parent node.

Ex: Deleting 12

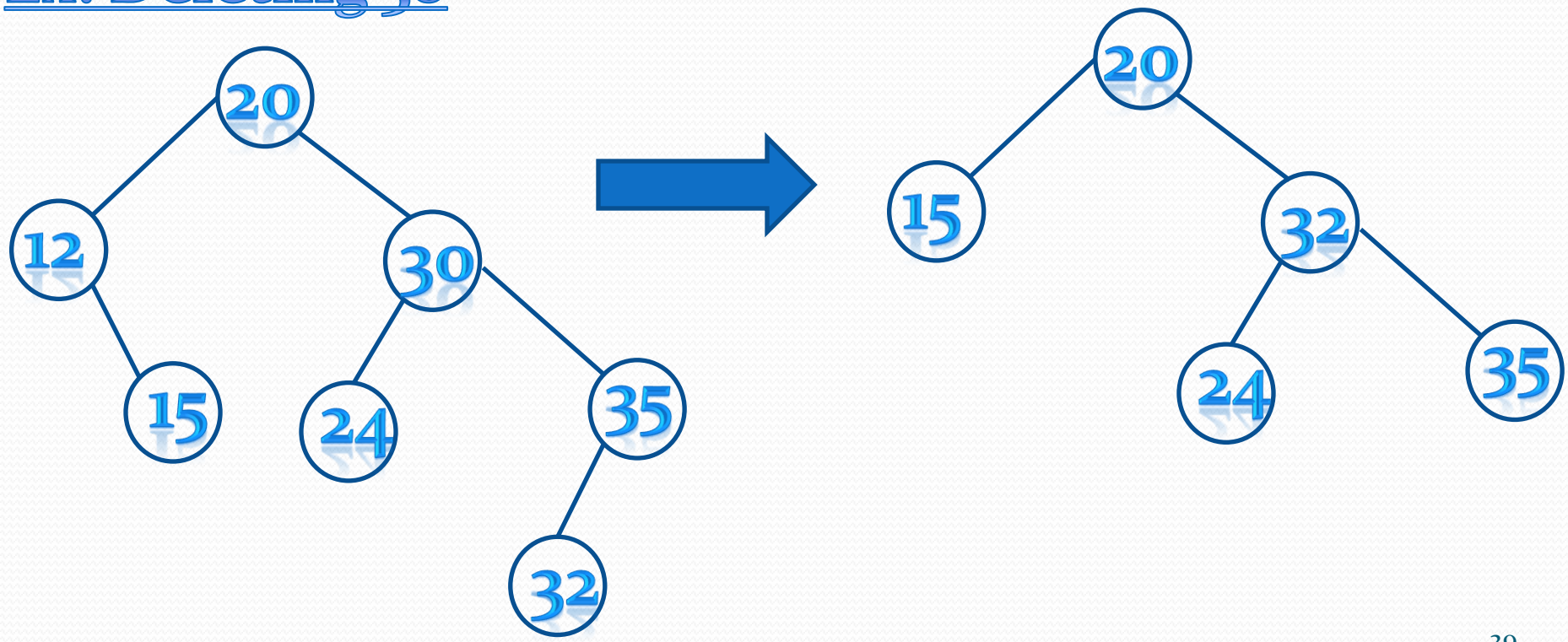


Deletion in BST…

c) ***Deleting a node with two children:*** In this case, the node being deleted is replaced by:

- leftmost node of its right child subtree
- **OR** rightmost node of its left child subtree

Ex: Deleting 30



Algorithm for Deletion in BST

1) If leaf node
   – set **NULL** to its parent pointer.
2) If node with one child
   – redirect its child pointer to its parent pointer.
3) If node with two child
   – replace node being deleted by
   I. Leftmost node of its right subtree
   II. **OR** rightmost node of its left subtree

See textbook page 405-406 for deletion details

TU Exam Question (2065)

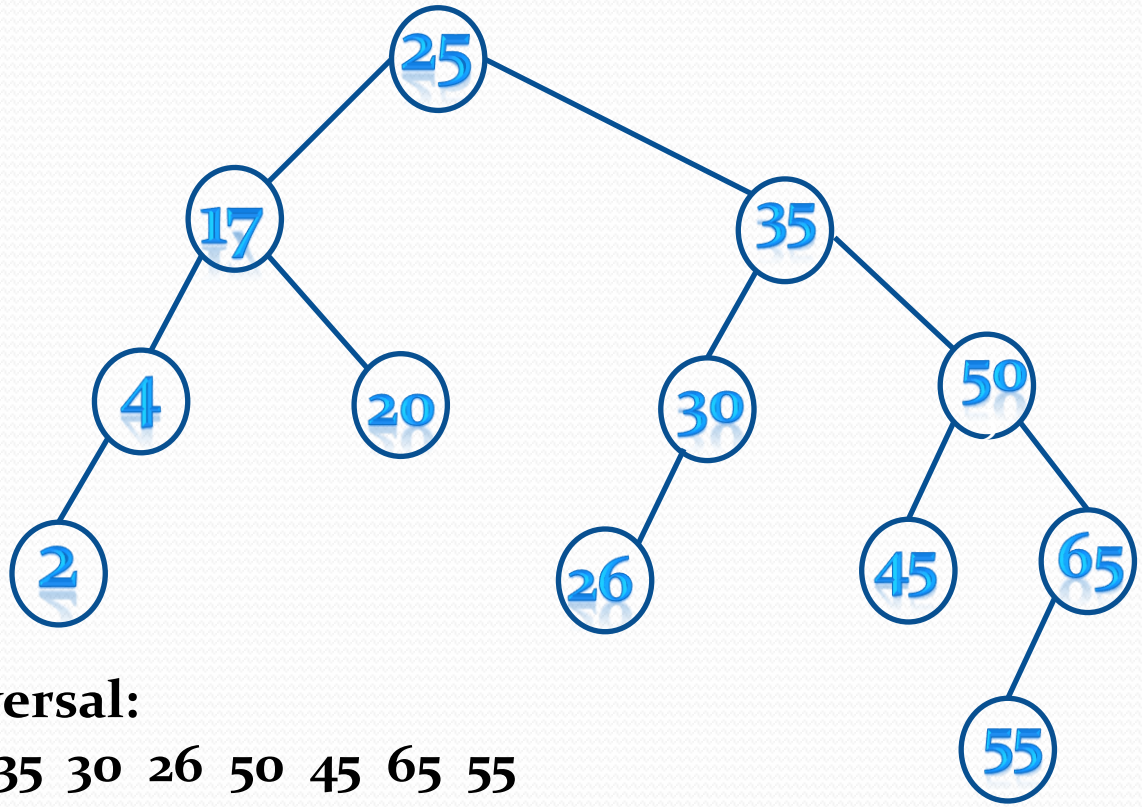
- What do you mean by binary tree? Explain the binary search tree with example.

Tree Traversal

- Tree traversal is one of the most common operations performed on tree data structures.
- ***Traversing a binary tree is a way in which each node in the tree is visited exactly once in a systematic manner.***
- We know that the order of traversal of nodes in a linear list is from first to last, however there is no such "natural" linear order for the nodes of a tree.
- We have the following three orderings or methods for traversing a **non-empty** binary tree:
  - ***Preorder (Depth-first order) Traversal***
  - ***Inorder (Symmetric order) Traversal***
  - ***Postorder Traversal***
- All theses traversal techniques are defined ***recursively***.

Preorder Traversal

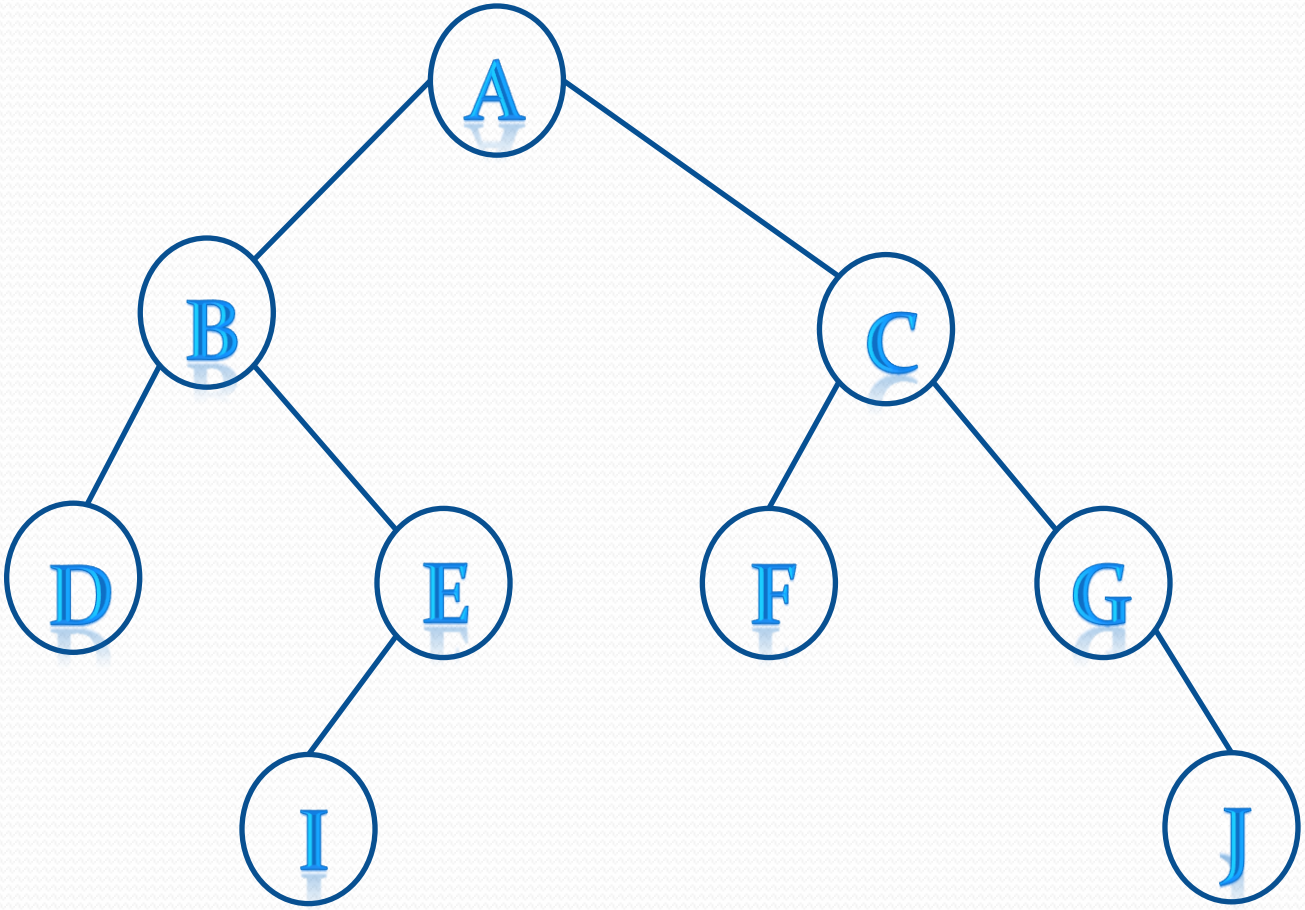
- *In this technique, first of all we process the root **R** of the binary tree **T**. Then, we traverse the left subtree **T1** of **R** in preorder (which means that we traverse root of subtree **T1** first and then its left subtree). After visiting left subtree of **R**, then we take over right subtree **T2** of **R** and process all the nodes in preorder.*



Preorder Traversal:

25 17 4 2 20 35 30 26 50 45 65 55

Traverse the given binary tree in preorder



Answer: A B D E I C F G J

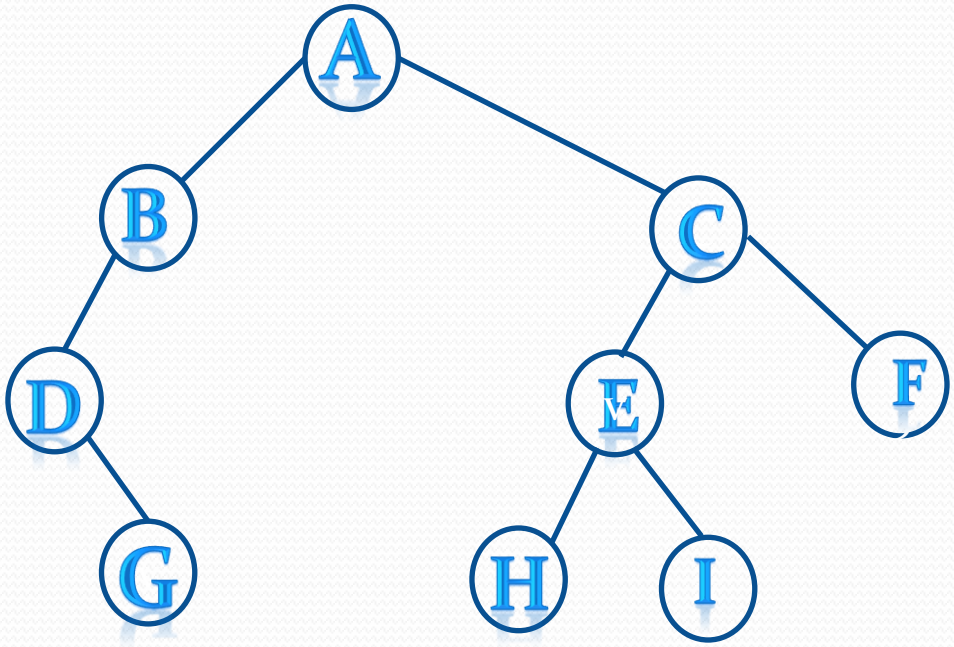
- ***Algorithm for preorder traversal***
  1. If root=NULL, return.
  2. Visit the root.
  3. Traverse the left subtree in preorder.
  4. Traverse the right subtree in preorder.

- ***C function for preorder traversal***
  - ***Function Call:*** *preorder(&rootnode);*
  - ***Function Definition***

```
void preorder(struct binarytree **root)
{
if(*root==NULL)
        return;
printf("%d\t", (*root)->info);
preorder((*root)->leftlink);
preorder((*root)->rightlink);
}
```

Postorder Traversal

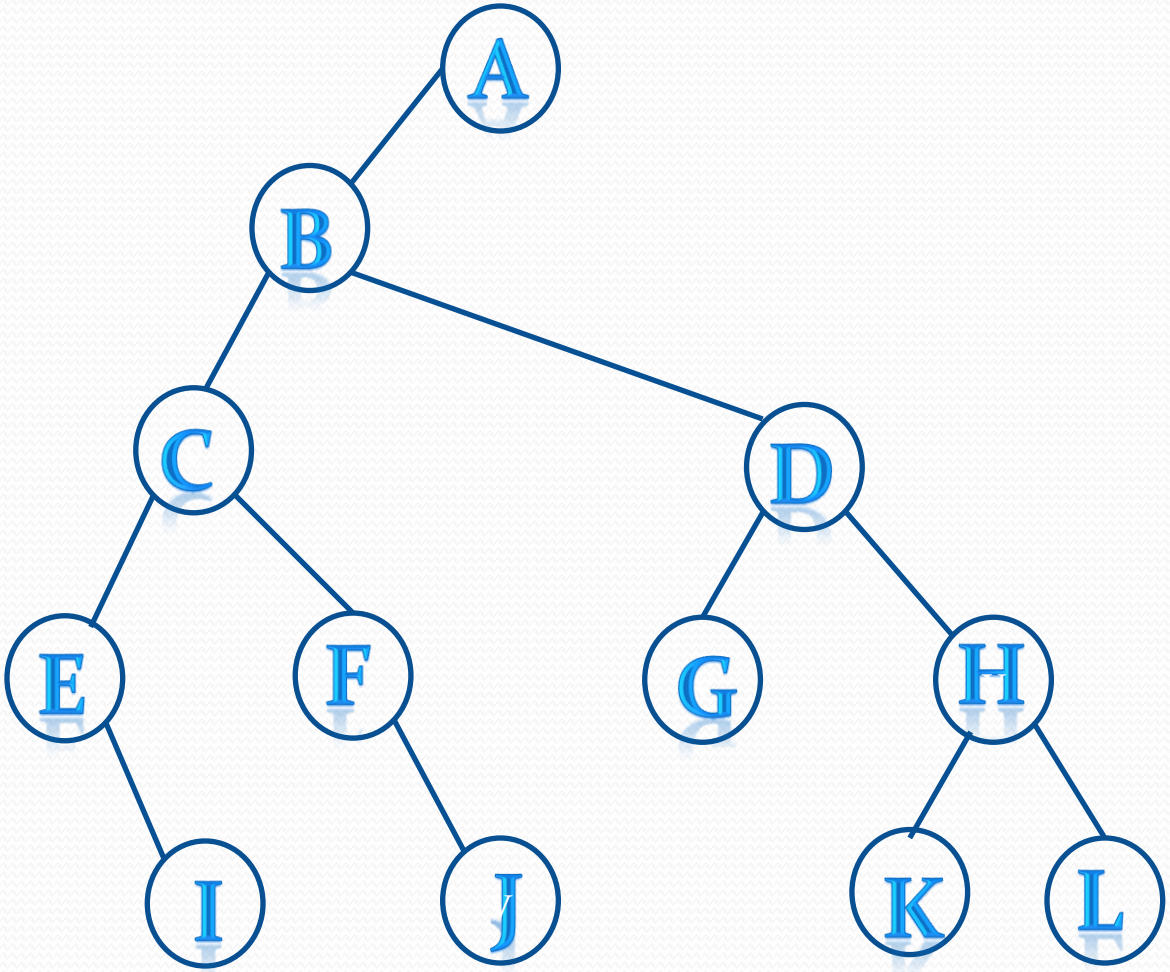
- *In this technique, first of all we process the left subtree **T1** of **R** in postorder, then right subtree **T2** of **R** in postorder and at the last, the root **R**.*



Postorder Traversal:
G D B H I E F C A

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

Traverse the given binary tree in postorder



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

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

- ***Algorithm for postorder traversal***
  1. If root=NULL, return.
  2. Traverse the left subtree in postorder.
  3. Traverse the right subtree in postorder.
  4. Visit the root.

- ***C function for postorder traversal***
  - ***Function Call:*** *postorder(&rootnode);*
  - ***Function Definition***

```
void postorder(struct binarytree **root)
{
if(*root==NULL)
        return;
postorder((*root)->leftlink);
postorder((*root)->rightlink);
printf("%d\t", (*root)->info);
}
```

TU Exam Question (2065)

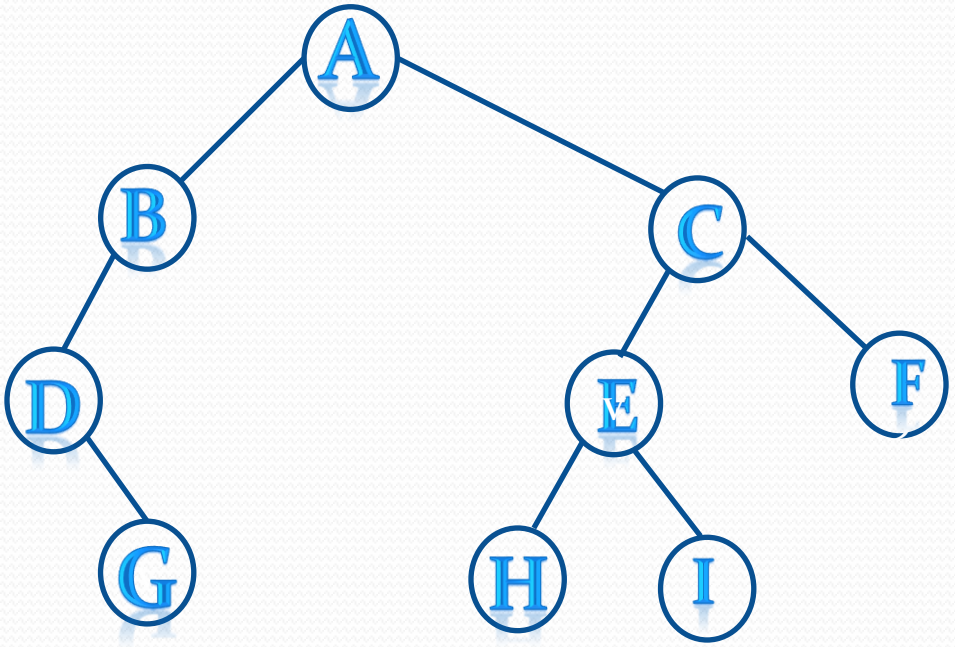
What is Post-order traversal?

TU Exam Question (2065)

Differentiate between Pre-order and In-order traversal?

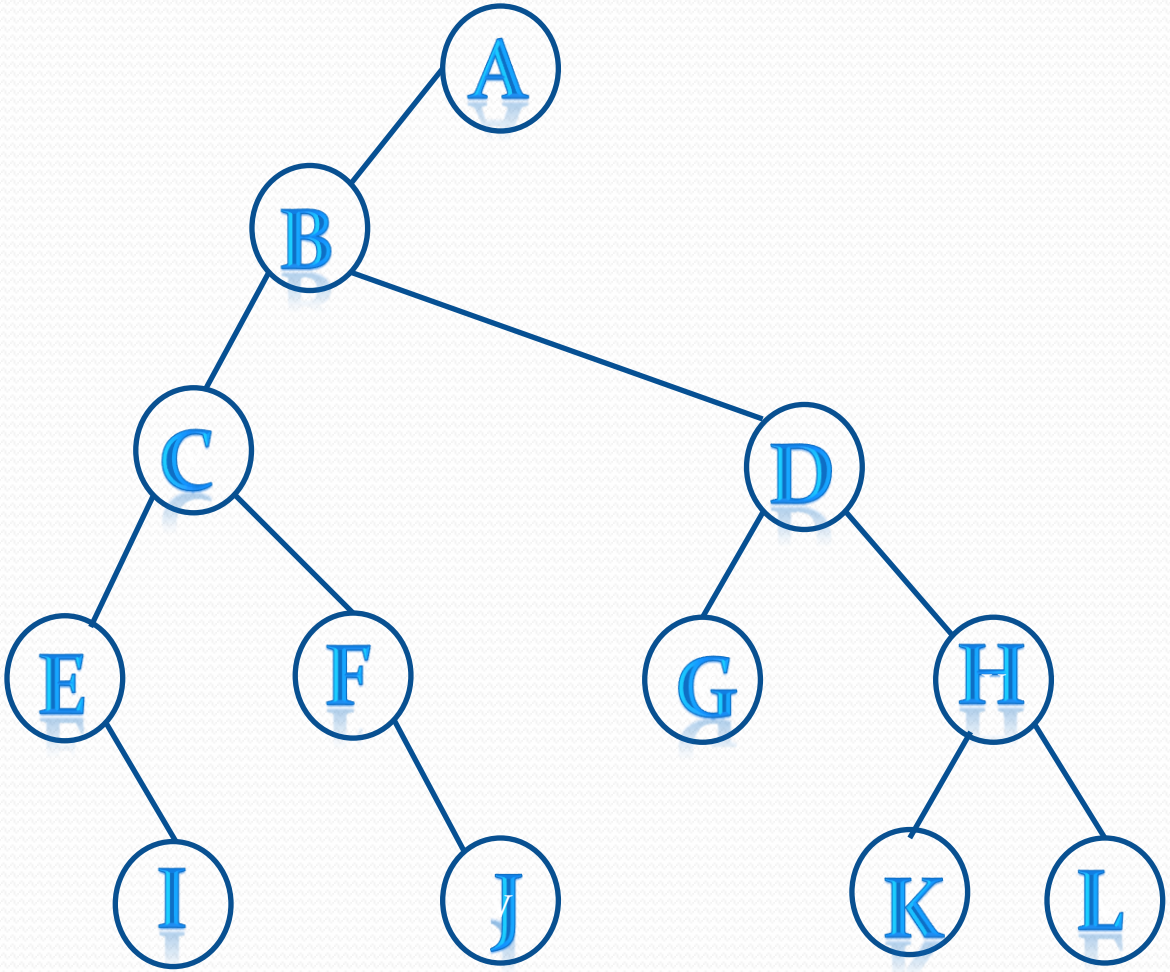
Inorder Traversal

- *In this traversal technique, first of all we process the left subtree **T1** of the root **R** in inorder, then we process the root and at last the right subtree **T2** of **R**.*



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

Perform inorder traversal on the given tree



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

- ***Algorithm for inorder traversal***
  1. If root=NULL, return.
  2. Traverse the left subtree in inorder.
  3. Visit the root.
  4. Traverse the right subtree in inorder.

- ***C function for inorder traversal***
  - ***Function Call:*** *inorder(&rootnode);*
  - ***Function Definition***

```
void inorder(struct binarytree **root)
{
if(*root==NULL)
        return;
postorder((*root)->leftlink);
printf("%d\t", (*root)->info);
postorder((*root)->rightlink);
}
```

- ***Note:***
  - By traversing a ***binary search tree*** (**BST**) in inorder, the nodes are always traversed (and therefore printed) in ascending order.
  - If we want to traverse and print data in descending order in a ***binary search tree***, then follow following steps:
    1. ***Traverse right subtree in inorder.***
    2. ***Visit the root.***
    3. ***Traverse the left subtree in inorder.***

TU Exam Question (2066)

- A binary tree T has 12 nodes. The in-order and pre-order traversals of T yield the following sequence of nodes:

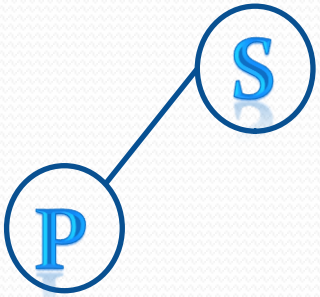
In-order:	VPNAQRSOKBTM
Pre-order:	SPVQNARTOKBM

Construct the binary tree T showing each step. Explain, how you arrive at solution in brief?

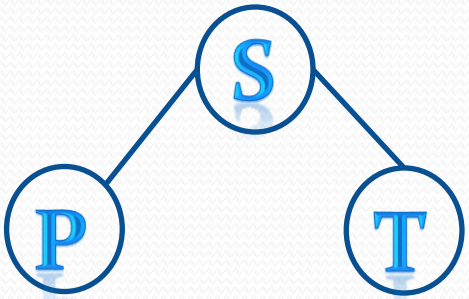
Answer: The binary tree **T** is constructed from its root downwards as follows:

a) The root of binary tree **T** is obtained by choosing the first node in its preorder (since preorder is ***root-left-right***). Thus the node **S** is the root of tree **T**.

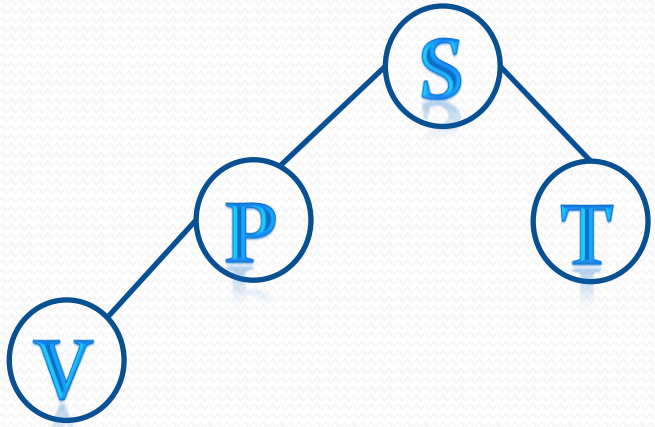
b) The left child of the root node **S** is obtained as follows: First we use the inorder of **T** to find the nodes in the left subtree **T1** of **S** (since inorder is ***left-root-right***). Thus the left subtree **T1** consists of the nodes **V**, **P**, **N**, **A**, **Q** and **R**. Then the left child of **S** is obtained by choosing the first node in the preorder of **T1** (which clearly appears in the preorder of **T**). Thus **P** is the left child of **S**.



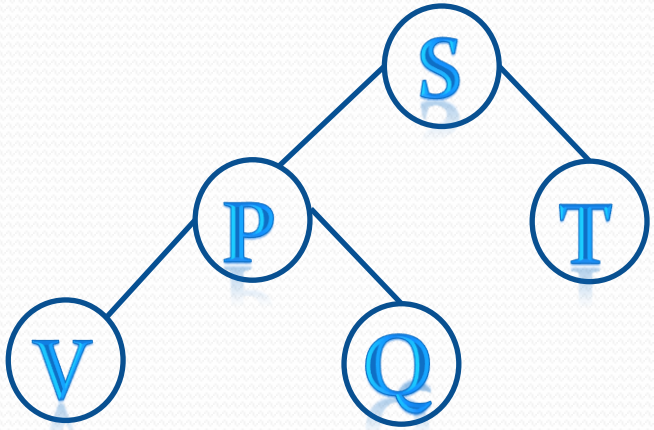
c) Similarly, the right subtree **T2** of **S** consists of the nodes **O**, **K**, **B**, **T** and **M**. Thus the right child of **S** is obtained by choosing the first node in the preorder of **T2**. Thus the node **T** is the right child of **S**.



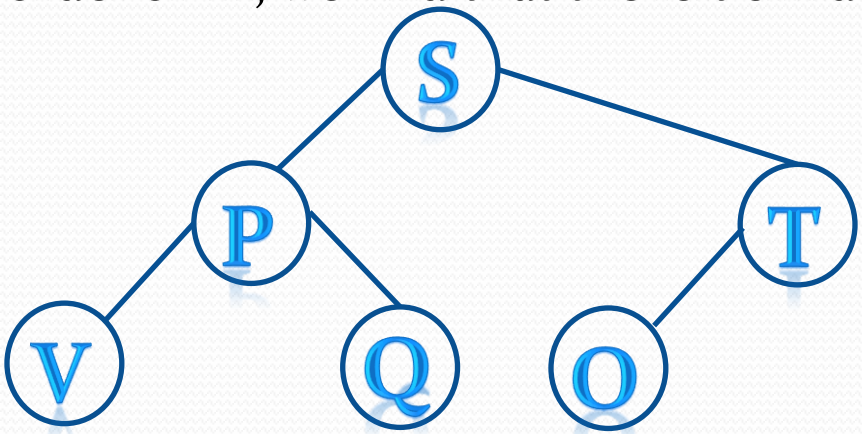
d) Now, we need to find the left child of **P**. By seeing the inorder of tree **T**, we find that the left subtree **T1** of **P** consists of only one node i.e. **V**. Thus the left child of **P** is **V**.



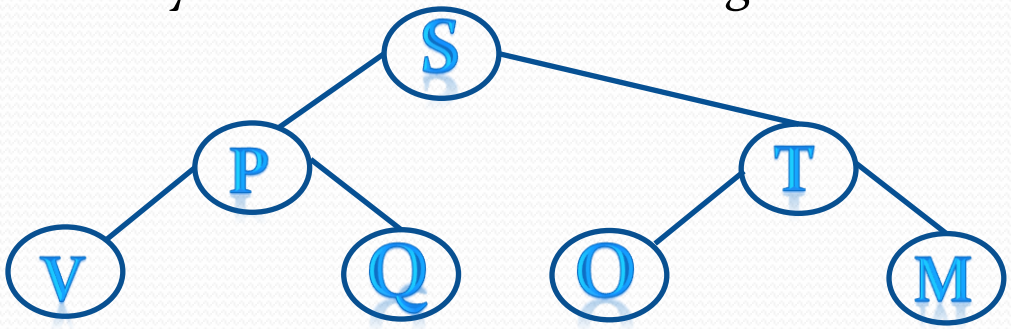
e) The right subtree **T2** of **P** consists of the nodes **N**, **A**, **Q** and **R**. Thus the right child of **P** is obtained by choosing the first node in the preorder of **T2**. Thus the node **Q** is the right child of **S**.



f) The left subtree **T1** of node **T** consists of the nodes **O**, **K** and **B**. By seeing the preorder of **T1**, we find that the left child of **T** is **O**.

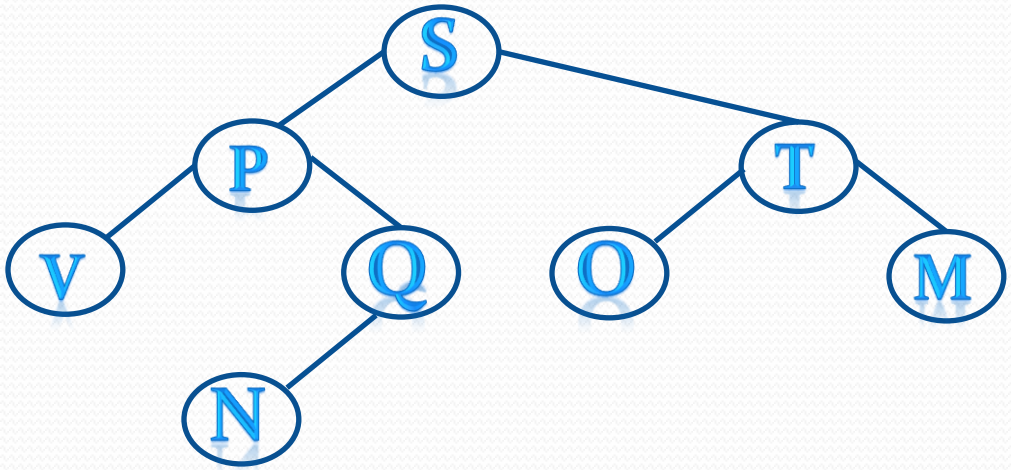


g) By seeing the inorder traversal, we find that the right subtree **T2** of node **T** consists of only one node **M**. Thus the right child of node **T** is **M**.

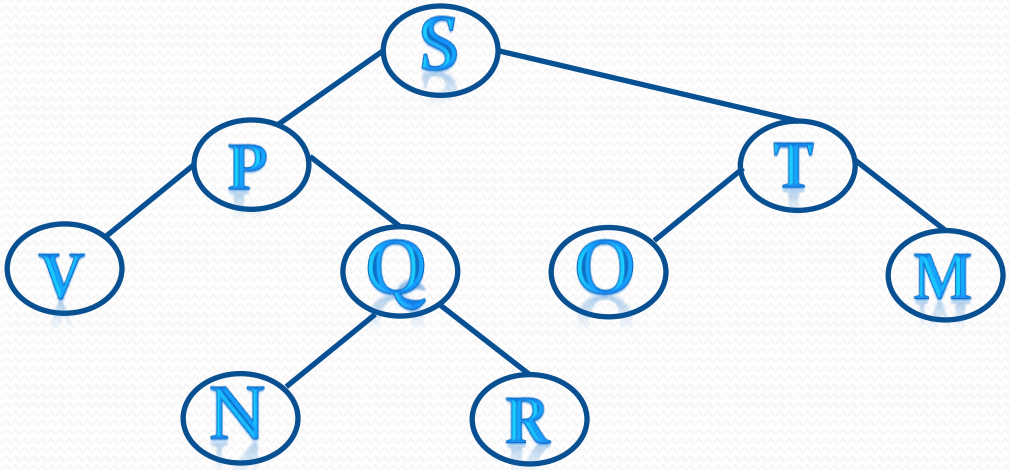


h) Now node **V** does not have any child because it is the leftmost node according to the inorder traversal. Therefore it's a leaf.

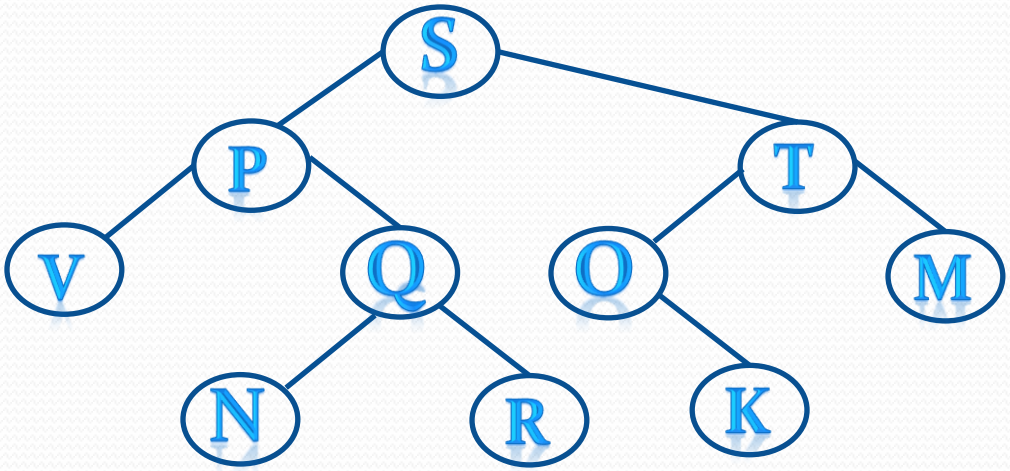
i) To find the left child of node **Q**, the left subtree **T1** of node **Q** consists of the nodes **N** and **A** (since **V** and **P** are already occupied). By seeing the preorder, it is clear that the left child of node **Q** is **N**.



j) The right subtree **T2** of node **Q** consists of only one node **R**. Thus the right child of node **Q** is **R**.



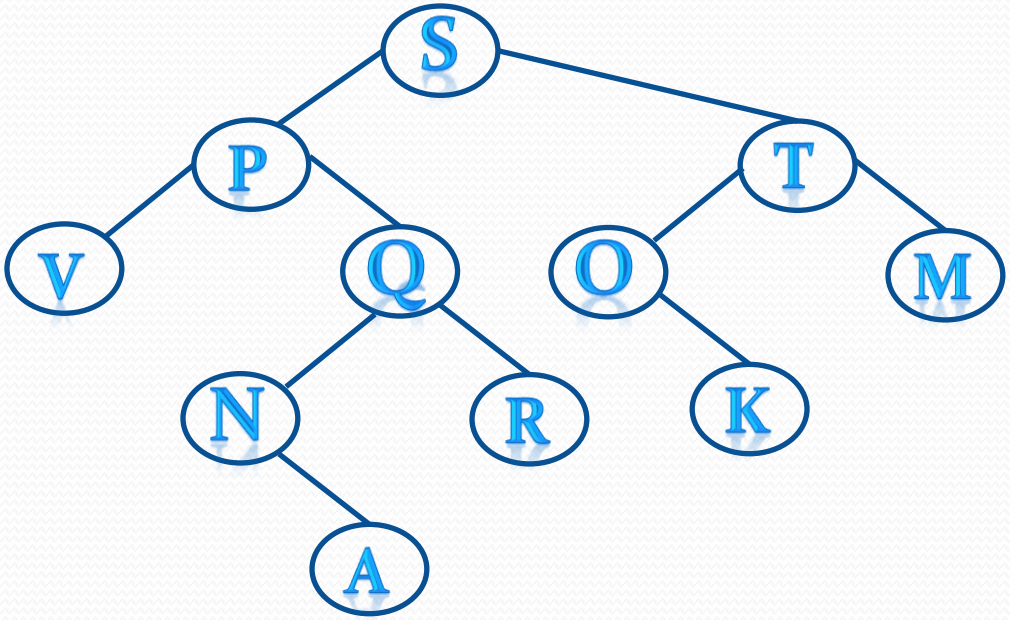
k) Now node **O** does not have a left child. By seeing the inorder, the right subtree of node **O** consists of nodes **K** and **B**. By seeing the preorder, node **K** is the right child of node **O**.



l) Node **M** is also a leaf node since it is the rightmost node in the inorder traversal.

m) Node **N** does not have a left child since its left subtree is empty.

n) The right subtree of node **N** consists of the only one node i.e. **A**.



o) Node **K** does not have a left child as its left subtree is empty which is obvious from its inorder.

p) The right subtree of node **K** contains the only remaining node i.e. **B**.

*Hence the final binary tree **T** is:*

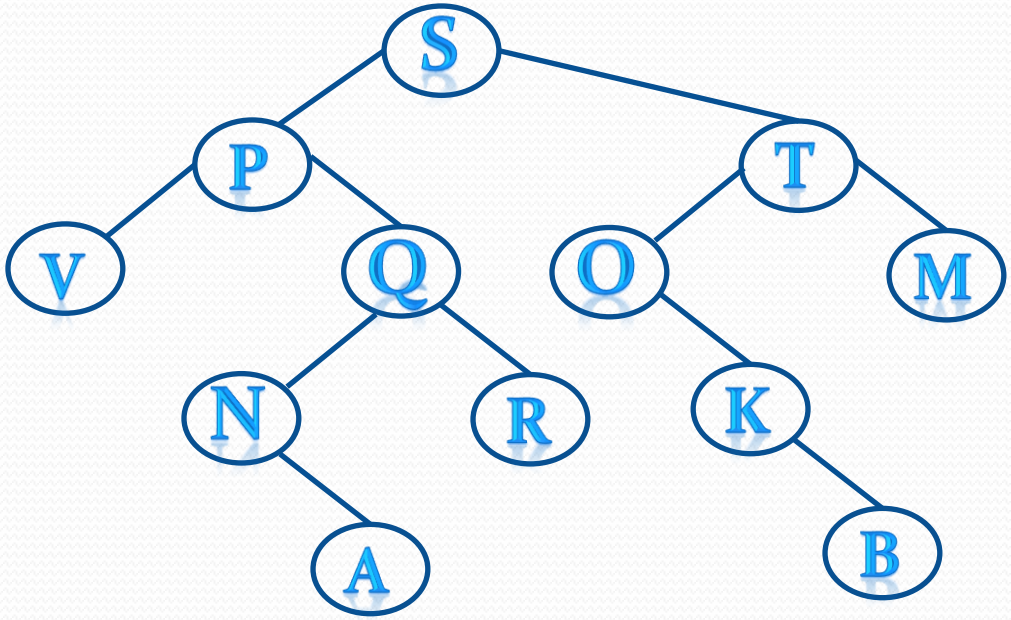


Fig: Final Binary Tree

Classwork

- A binary tree **T** has 9 nodes. The inorder and preorder traversals of **T** yield the following sequence of nodes:

  In-order: E A C K F H D B G

  Pre-order: F A E K C D H G B

Draw the tree **T**.

Answer:

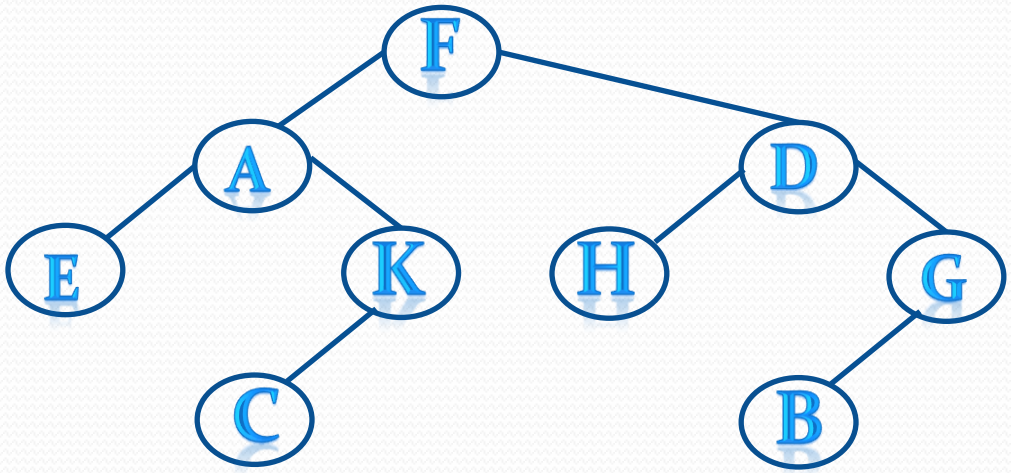


Fig: Final Binary Tree

Homework

- The following sequence gives the preorder and inorder of the binary tree **T**:

  Preorder: A B D G C E H I F

  Inorder: D G B A H E I C F

Draw the diagram of the tree **T**.

Answer:

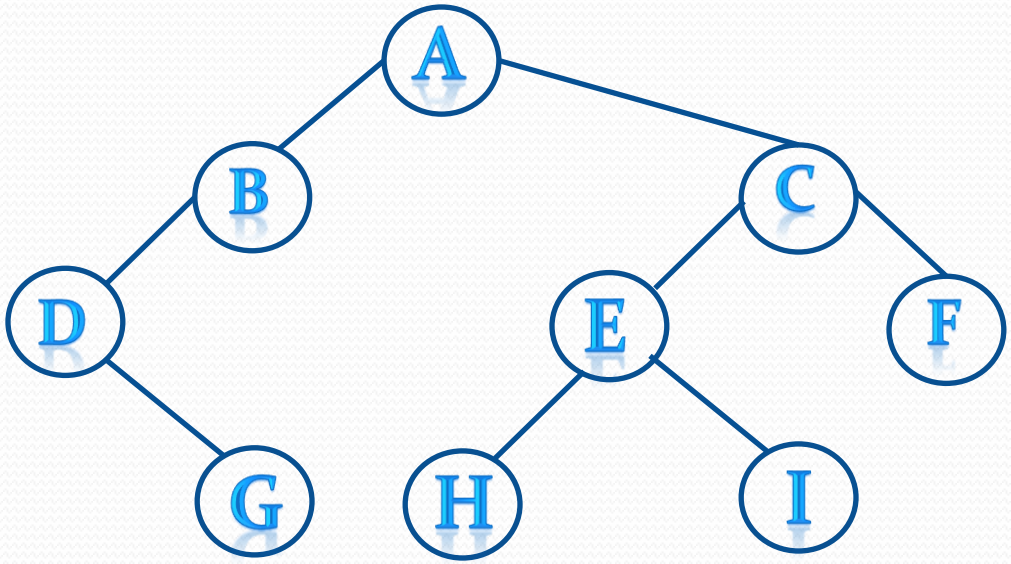


Fig: Final Binary Tree

Homework

1. Suppose the following list of letters is inserted in order into an empty ***binary search tree*** **T**:

   S, T, P, Q, M, N, O, R, K, V, A, B

   a) Draw the final tree **T**.

   b) Find the inorder traversal of tree **T**.

2. Suppose the following eight numbers are inserted in order into an empty ***binary search tree*** **T**:

   50, 33, 44, 77, 35, 60, 40, 100

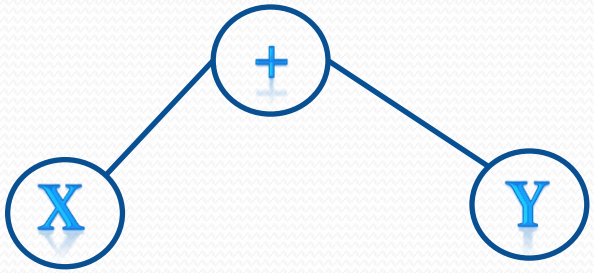
   Draw the tree **T**.

Model Question (2008)

- Describe, using an example, how an arithmetic expression can be represented using a binary tree. Once represented, how can the expression be output in postfix notation?

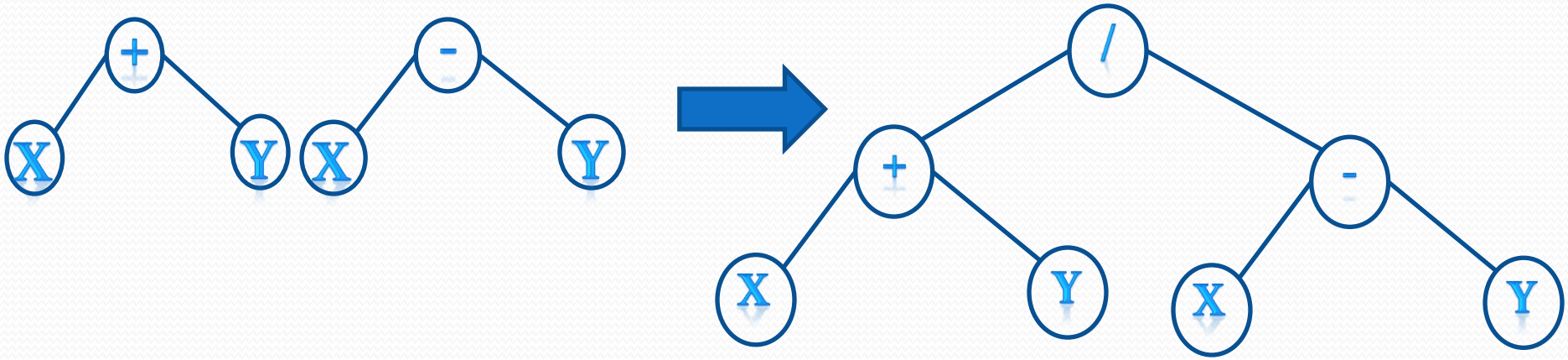
Application of Binary Tree

- A ***binary tree*** (or more appropriately ***strictly binary tree***) can be used to represent an expression containing operands and binary operators.
- A ***binary tree*** used to represent a binary expression is called a ***binary expression tree (BET)***.
- The root of **BET** contains the ***operator*** and its two children contains ***operands***.
- For example, **X + Y** can be represented as:



Application of Binary Tree…

- If we have an expression with more arithmetic operators like **(X+Y)/(X-Y)**, then we have to construct two subtrees and then these two subtrees are added to the root node as:



- When a ***binary expression tree*** is traversed in preorder (root-left-right), we get prefix form of the expression and when traversed in postorder (left-right-root), we get postfix form of the expression. Thus postfix form of above arithmetic expression is: **XY+XY-/**

Classwork

- Represent the following arithmetic expressions by their corresponding binary trees and find out their postfix equivalent:
  a) **A+B*C**
  b) **(A+B)*C**
  c) **A+(B-C)*D$(E*F)**
  d) **(A+B*C)$((A+B)*C)**

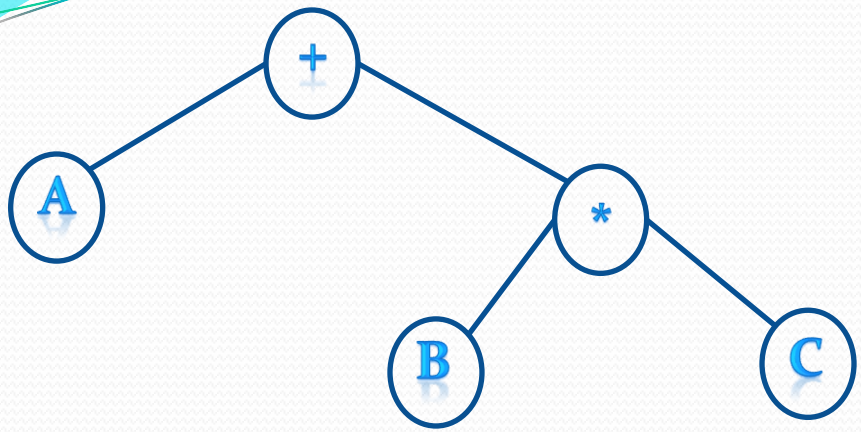


Fig (a): A+B*C

Postfix: ABC*+

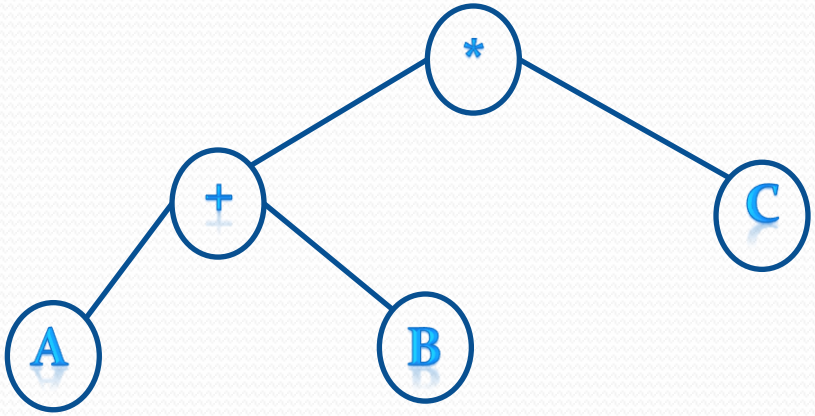


Fig (b): (A+B)*C

Postfix: ABC*+

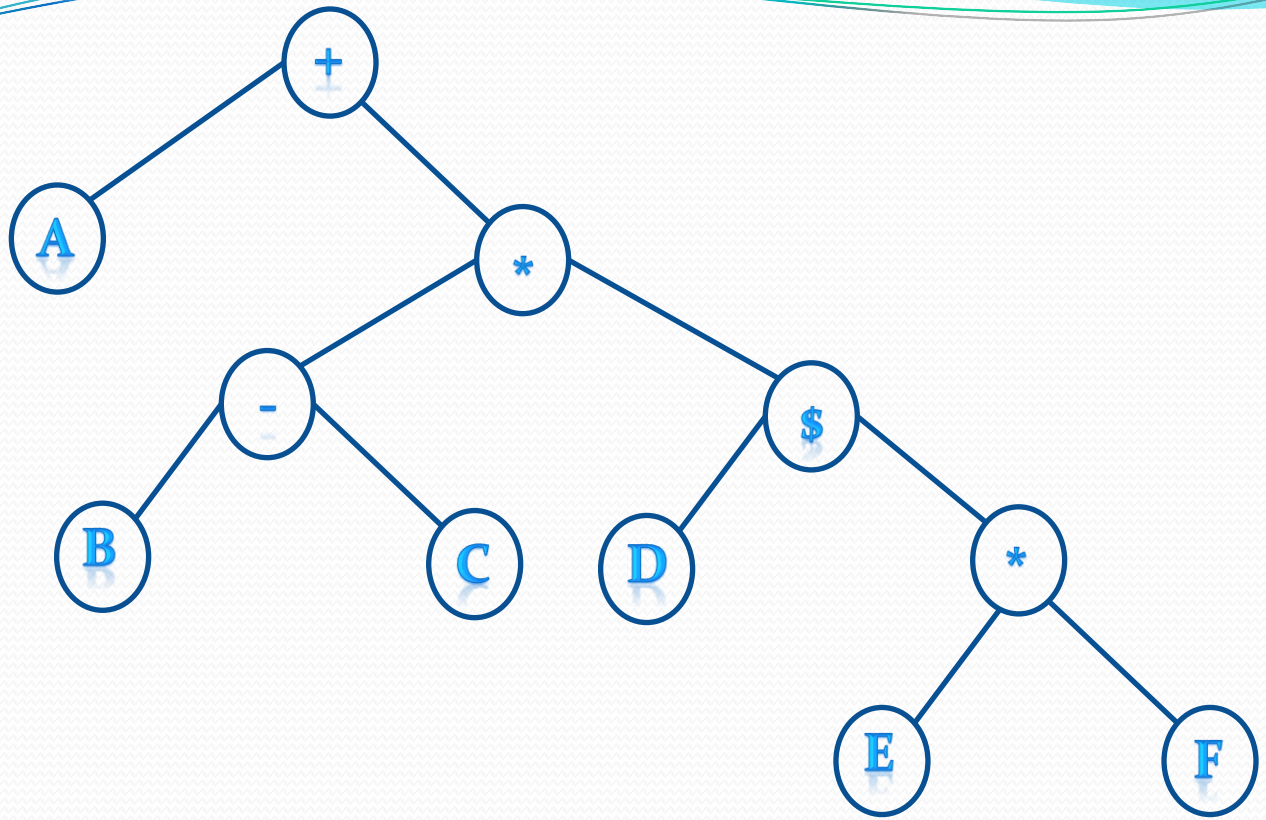


Fig (c): A+(B-C)*D$(E*F)

Postfix: ABC-DEF*$*+

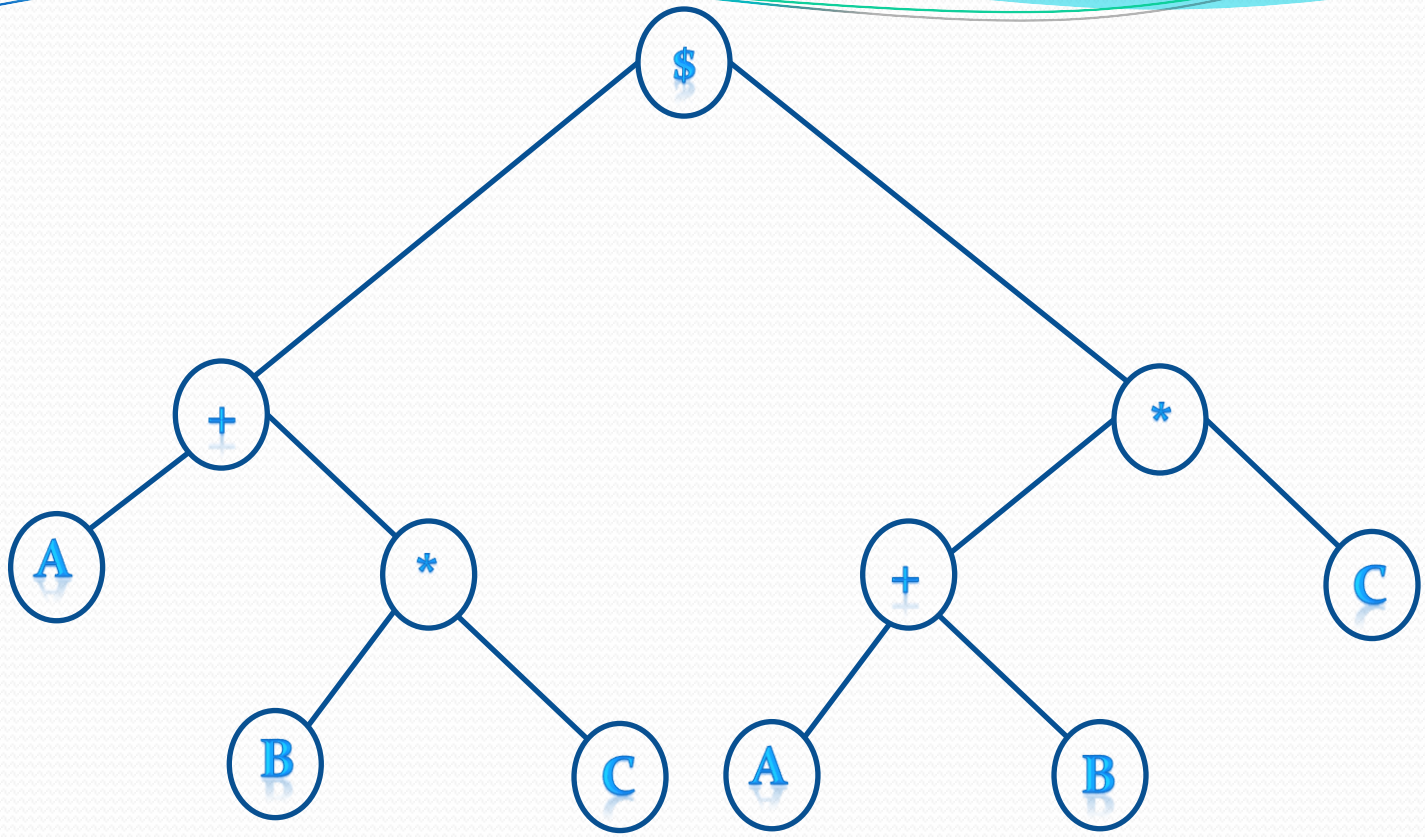


Fig (d): (A+B*C) $((A+B)*C)

Postfix: ABC*+AB+C*$