SRI MANAKULA VINAYAGAR ENGINEERING COLLEGE
(An Autonomous Institution)
(Approved by AICTE, New Delhi & Affiliated to Pondicherry University)
(Accredited by NBA-AICTE, New Delhi, ISO 9001:2000 Certified Institution &
Accredited by NAAC with "A" Grade)
Madagadipet, Puducherry - 605 107

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Subject Name: **Data Structures**
Subject Code:

Verified by: **Approved by:**

UNIT – IV
TREES

Trees: Basic Tree Terminologies, Different types of Trees: Binary Tree, Threaded Binary Tree, Binary Search Tree, Binary Tree Traversals, AVL Tree. Introduction to B-Tree and B+ Tree.

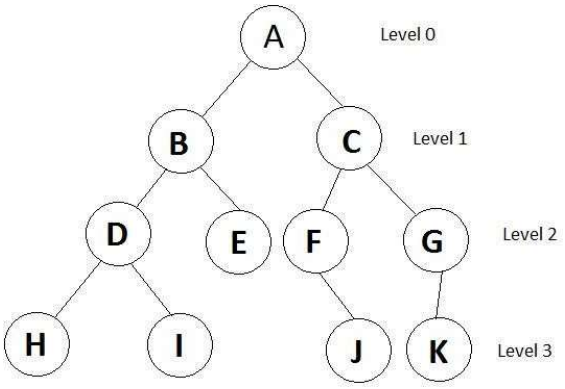
1. **Define tree.**

   Tree is a non linear data structure. There is no linear relation between the data items. It can be defined as finite set of more than one node.

   There is a special node designated as root node.

   The remaining nodes are partitioned into sub tree of a tree.

   The below diagram shows the tree



2. **Define path in tree?**

   The path in a tree is referred as the nodes in which the successive nodes are connected by the edge in a tree. For example: the path from A to l

   A-B, B-D, D-I.

3. **Define terminal nodes in a tree?**

   A node that has no children is called as a terminal node. It is also referred as a leaf node. These nodes have degree has zero.

4. **Define non-terminal nodes in a tree?**

   All intermediate nodes that traverse the given tree from its root node to the terminal nodes are referred as non-terminal nodes.

5. **Define branch, siblings & ancestors?**

   Branch or edge of a tree is called as the link or connection between two nodes. The nodes having the same parent are called siblings. The ancestor of a node is referred as all nodes along the path of root node to the path node.

6. **State the properties of the tree?**

   Any node can be the root of the tree.

   If the root is identified; then that tree is called as the rooted node.

   If it is not identified; then that tree is called as the free tree.

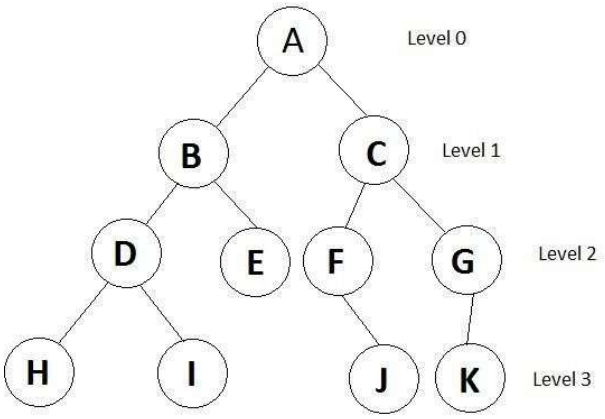
   Every node, expect the root node has a unique parent.

7. **Define degree?**

   The degree of a node is referred as the number of sub-trees of a particular node.

8. **Define a binary tree?**

   A binary tree is tree, which has nodes either empty or not more than two child nodes each of which may be a leaf node.



9. **Define a full binary tree?**

   A full binary tree is a tree in which all the leaves are on the same level and every non-leaf node has exactly two children.

10. Define complete binary tree?

A complete binary tree is a tree in which every non-leaf node has exactly two children not necessarily to be on the same level.

11. State the properties of binary tree?

The maximum number of nodes on level n of the binary tree is 2^{n-1} where n>=1 The maximum number of nodes in a binary tree of height n is 2^{n-1} where n>=1. For any non empty tree $n_l = n_d + 1$ where n_l is the number of leaf nodes and n_d is the number of nodes of degree 2.

12. What are the different ways of representing binary tree?

- Linear representation using arrays
- Linked representation using pointers

13. What is meant by binary tree traversal?

Traversing a binary tree means moving through all the nodes in the binary tree visiting each node in the tree only once.

14. What are the different types of binary tree traversal?

- Preorder
- Inorder
- Postorder

15. What are the tasks performed during preorder tree traversal?

Process the root node.

Traverse the left subtree

Traverse the right subtree

16. What are the tasks performed during inorder tree traversal?

Traverse the left subtree

Process the root node.

Traverse the right subtree

17. What are the tasks performed during postorder tree traversal?

Traverse the left subtree

Traverse the right subtree

Process the root node.

18. State the merits and demerits of linear representation of binary tree?

Merits:

Storage method is easy and can be easily implemented in arrays.
When the location of a parent /child node is known other one can be determined easily. It requires static memory allocation so it is easily implemented in all programming language

Demerits:

Insertions and deletions in a node, taker an excessive amount of processing time due to data movement up and down the array.

19. State the merits and demerits of linked representation of binary tree?

Merits:

Insertions and deletions in a node, involves no data movement except the re arrangement of pointers, hence less processing time.

Demerits:

Given a node structure, it is difficult to determine its parent node.

Memory spaces are wasted for storing null pointers for the nodes, which have one or no subtrees. It requires dynamic memory allocation, which is not possible in some programming languages.

20. What do you mean by general tree?

General tree is a tree with nodes having any number of children

21. What is the length of the path in a tree?

The path length of a tree is the sum of the levels of all the tree's nodes. The path length of a tree with N nodes is the sum of the path lengths of the subtrees of its root plus N-1.

22. Define B+ tree indexing?

A B+ tree can be viewed as a B-tree in which each node contains only keys (not key-value pairs), and to which an additional level is added at the bottom with linked leaves. The primary values of a B+ tree is in storing data for efficient retrieval in a block-oriented storage context — in particular, file systems.

23. What are the applications of binary trees?

1. Binary search tree
2. Binary tries
3. Hash trees
4. Heaps
5. T-tree
6. Syntax tree
7. Huffman coding tree

24. What is the use of indexing technique?

Indexing is an auxiliary data structure which speeds up the record retrieval.

25. Compare B- tree and B+ tree?(Feb/Mar 2021)

a. A B+ tree is the form of a balanced tree in which every path from the root of the tree to a leaf of the tree is the same length. B+ trees are good for searches but cause some overhead issues in wasted space.

b. B-trees are similar to B+ trees but it allows search-key values to appear only once, eliminates redundant storage of search keys. It is possible sometimes to find search-key value before reaching the leaf node. Implementation is harder than B+ trees.

26. Differentiate binary tree and binary search tree?

a. Binary is a specialized form of tree with two child (left child and right child). It is simply representation of data in tree structure.

b. Binary search tree is a special type of binary tree that follows the following condition:

   i. Left child is smaller than its parent node

   ii. Right child is greater than its parent node

27. Give some of the applications of B tree?

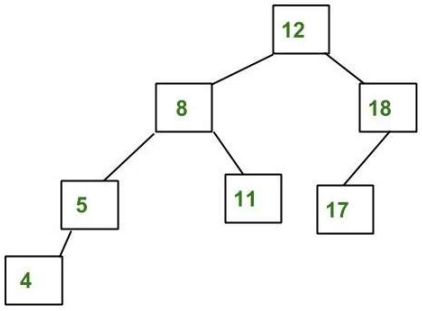
B tree is optimized for systems that read and write large blocks of data. It is commonly used in databases and file systems.

B tree also optimizes costly disk accesses that are of concern when dealing with large data sets.

28. Define AVL tree?

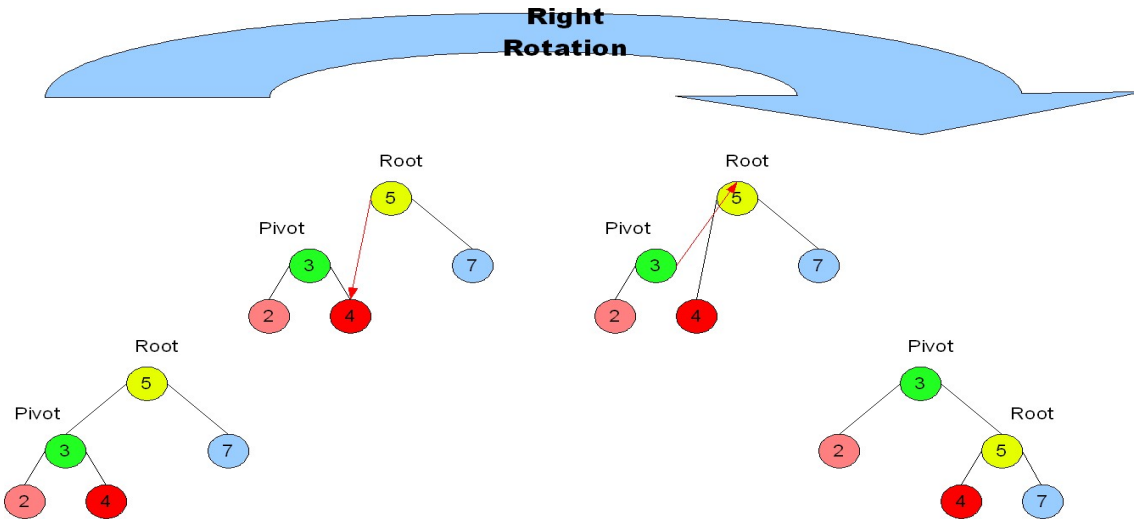
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

An Example Tree that is an AVL Tree



The above tree is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.

29. Perform RR rotation in AVL tree with an example.



5 MARKS

1. EXPLAIN THE TREE TERMINOLOGIES ROOT, EDGE, PARENT.

- In linear data structure data is organized in sequential order and in non-linear data structure data is organized in random order. A tree is a very popular non-linear data structure used in a wide range of applications. A tree data structure can be defined as follows...

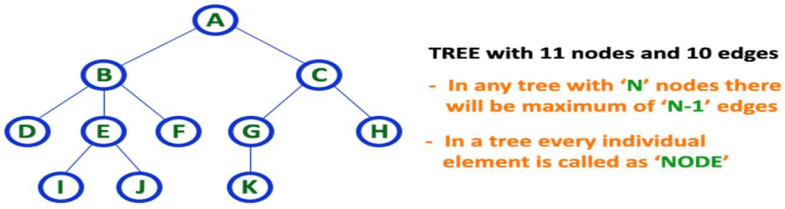
Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

- A tree data structure can also be defined as follows...

Tree data structure is a collection of data (Node) which is organized in hierarchical structure recursively

- In tree data structure, every individual element is called as **Node**. Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure. In a tree data structure, if we have **N** number of nodes then we can have a maximum of **N-1** number of links.

Example

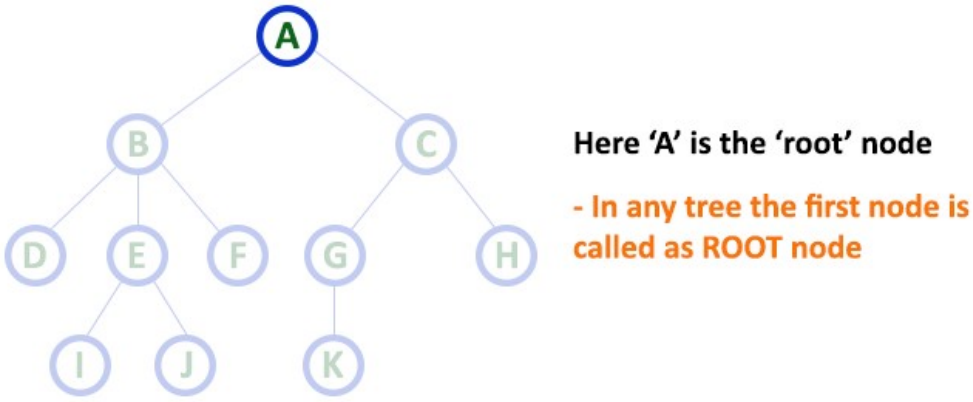


Terminology

In a tree data structure, we use the following terminology...

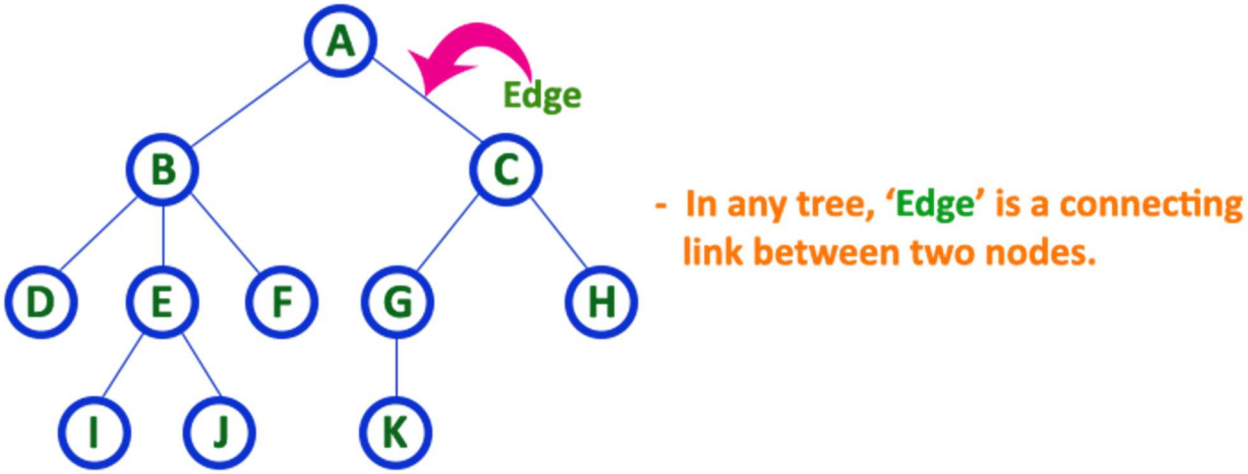
1. Root

In a tree data structure, the first node is called as **Root Node**. Every tree must have a root node. We can say that the root node is the origin of the tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.



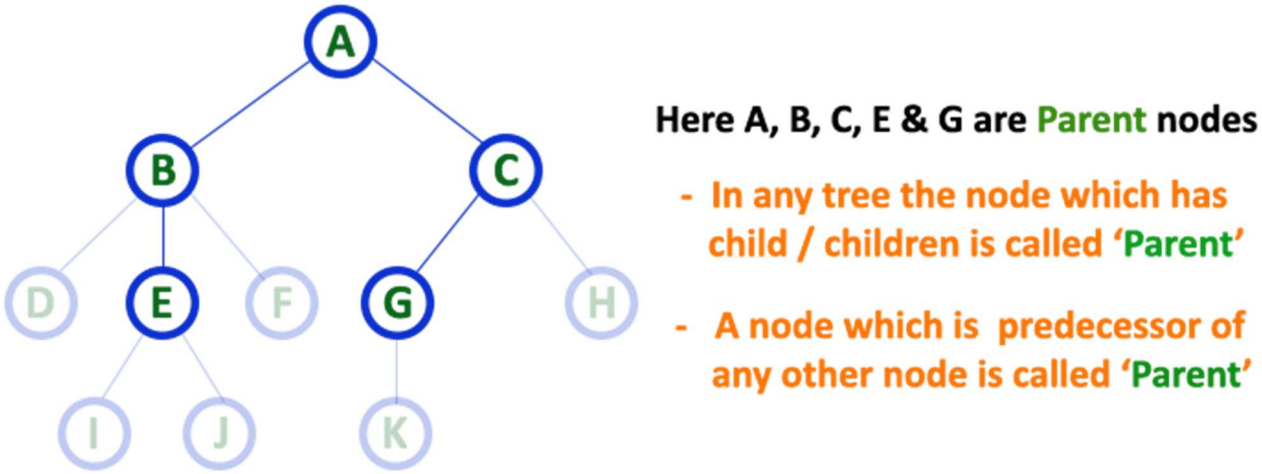
2. Edge

In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.



3. Parent

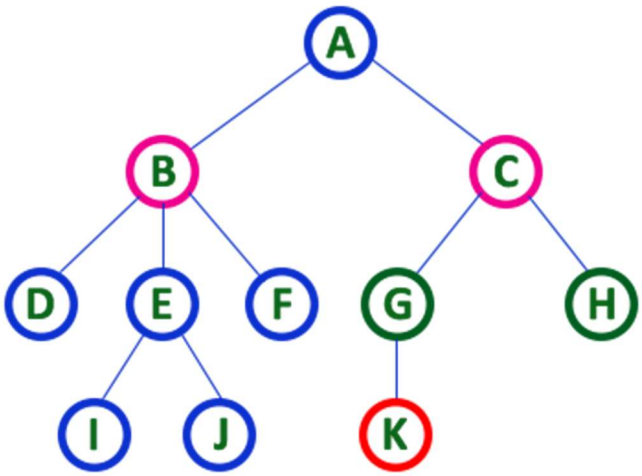
In a tree data structure, the node which is a predecessor of any node is called as **PARENT NODE**. In simple words, the node which has a branch from it to any other node is called a parent node. Parent node can also be defined as "**The node which has child / children**".



2. EXPLAIN THE TREE TERMINOLOGIES LEAF, SIBLING, CHILD.

1. Child

In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



Here B & C are Children of A

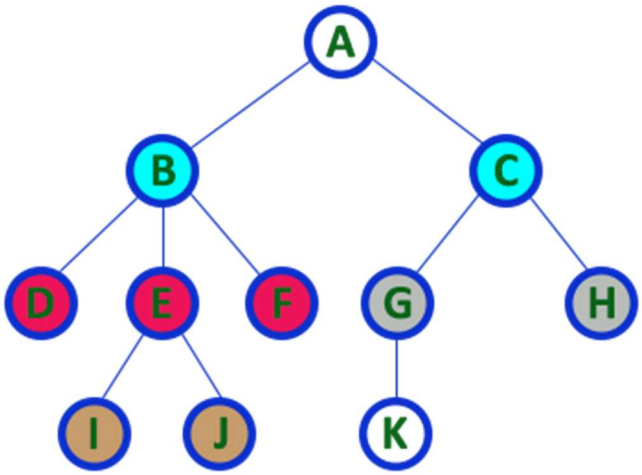
Here G & H are Children of C

Here K is Child of G

- descendant of any node is called as CHILD Node

2. Siblings

In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with the same parent are called Sibling nodes.



Here B & C are Siblings

Here D E & F are Siblings

Here G & H are Siblings

Here I & J are Siblings

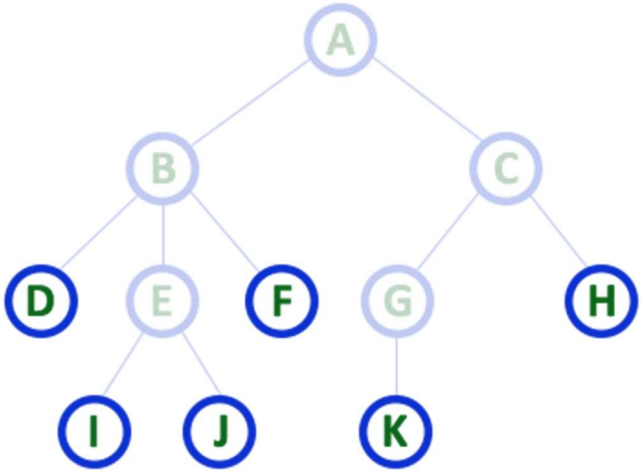
- In any tree the nodes which has same Parent are called 'Siblings'

- The children of a Parent are called 'Siblings'

3. Leaf

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.

In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as '**Terminal**' node.



Here D, I, J, F, K & H are Leaf nodes

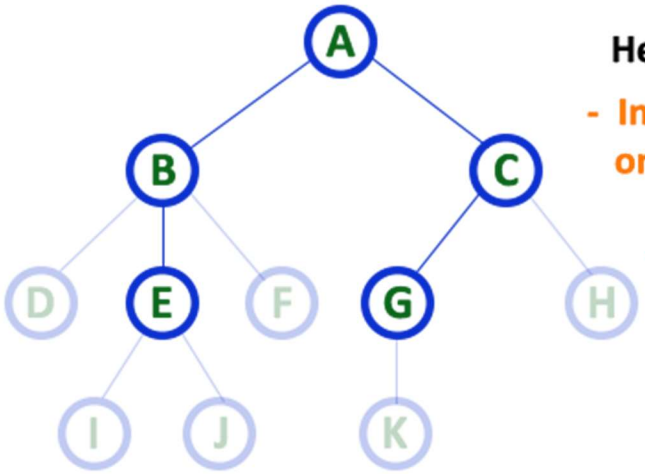
- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

3. EXPLAIN THE TREE TERMINOLOGIE INTERNAL NODES, DEGREE, LEVEL.

1. Internal Nodes

In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.

In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. **The root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as '**Non-Terminal**' nodes.

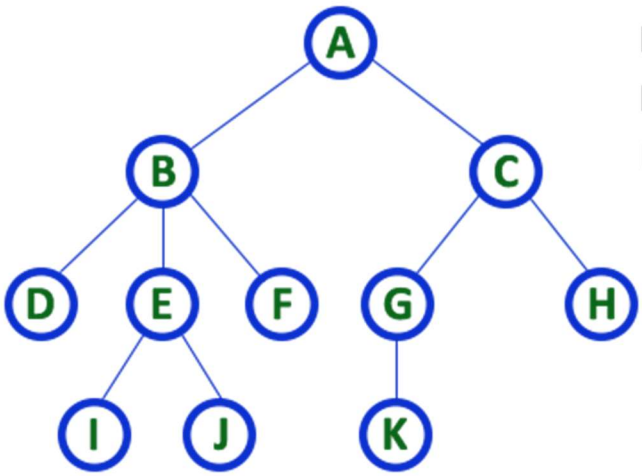


Here A, B, C, E & G are Internal nodes

- In any tree the node which has atleast one child is called 'Internal' node
- Every non-leaf node is called as 'Internal' node

2. Degree

In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'



Here Degree of B is 3

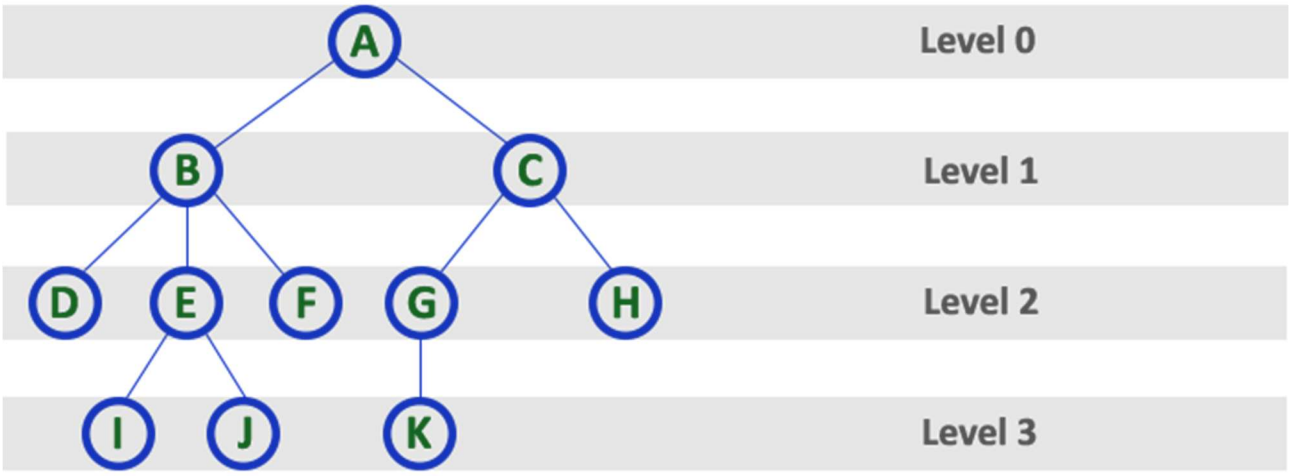
Here Degree of A is 2

Here Degree of F is 0

- In any tree, 'Degree' of a node is total number of children it has.

3. Level

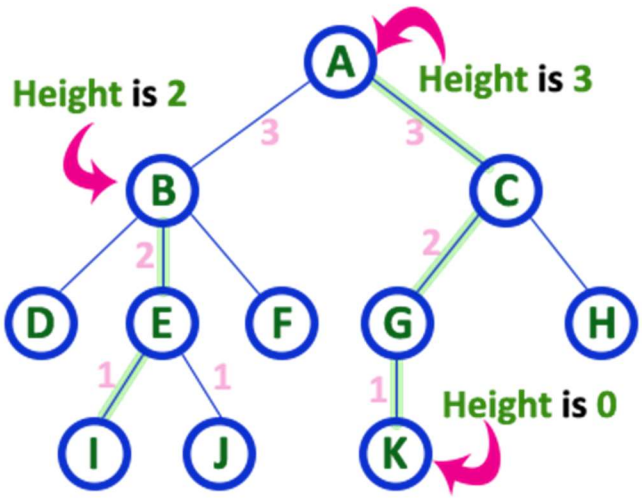
In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



4. EXPLAIN THE TREE TERMINOLOGIES HEIGHT, DEPTH, PATH, SUB TREE.

1. Height

In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. In a tree, height of the root node is said to be **height of the tree**. In a tree, **height of all leaf nodes is '0'.**

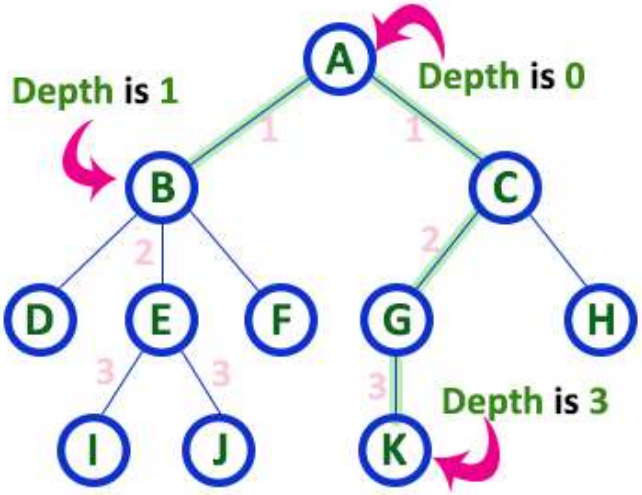


Here Height of tree is 3

- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.

2. Depth

In a tree data structure, the total number of egdes from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.**

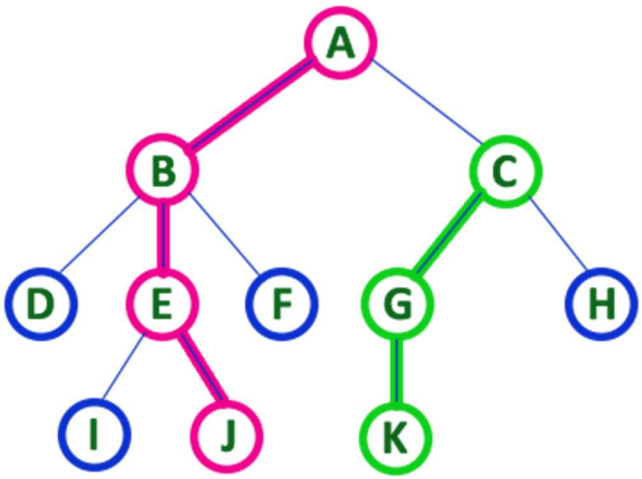


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

3. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4**.



- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is

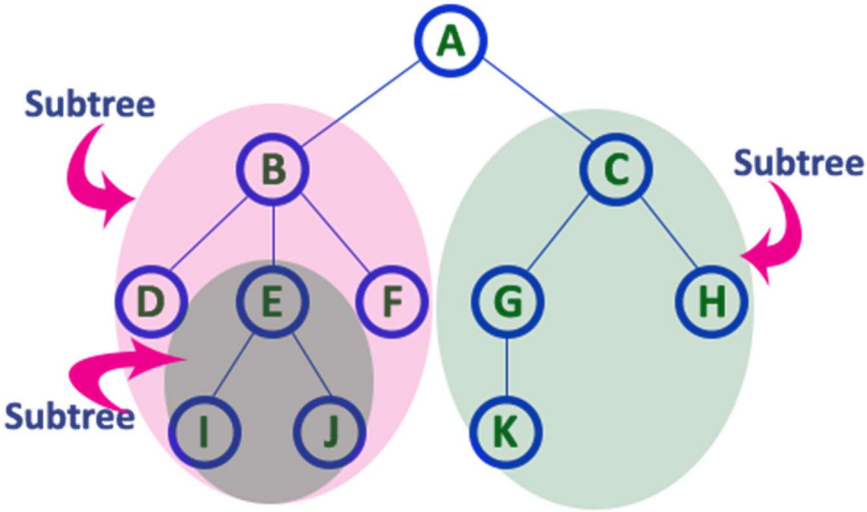
A - B - E - J

Here, 'Path' between C & K is

C - G - K

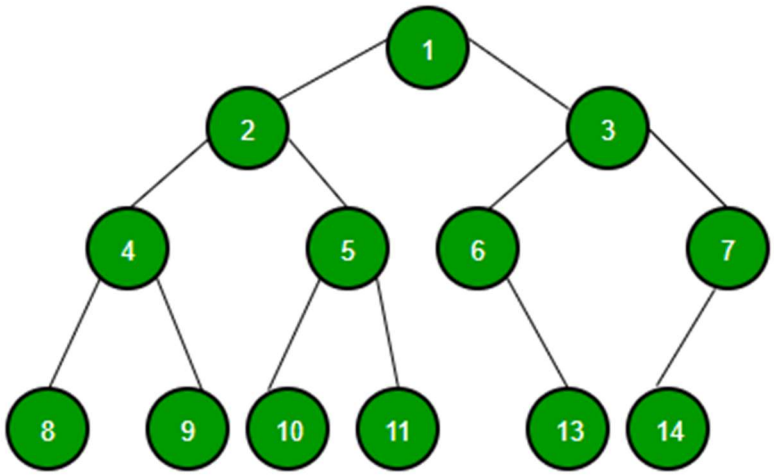
4. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



5. WHAT IS BINARY TREE AND EXPLAIN THE STRICTLY BINARY TREE.

- A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



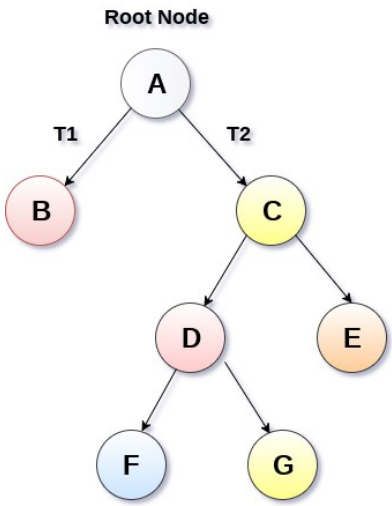
A Binary Tree node contains following parts.

- Data
- Pointer to left child
- Pointer to right child

Strictly Binary Tree

In Strictly Binary Tree, every non-leaf node contain non-empty left and right sub-trees. In other words, the degree of every non-leaf node will always be 2. A strictly binary tree with n leaves, will have (2n - 1) nodes.

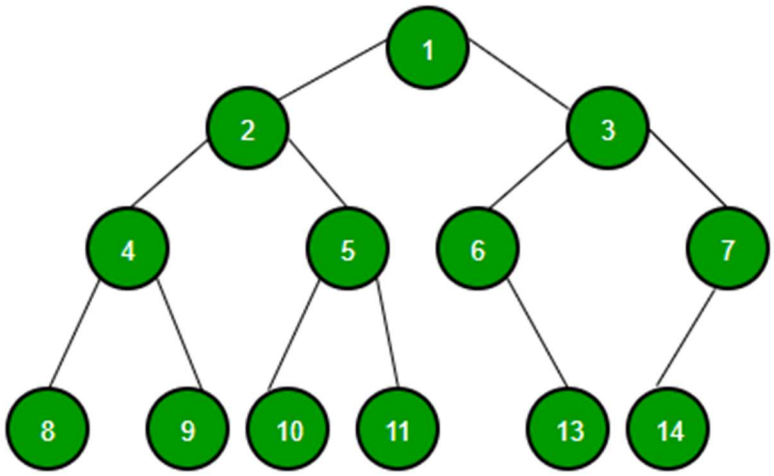
A strictly binary tree is shown in the following figure.



Strictly Binary Tree

6. **WHAT IS BINARY TREE AND EXPLAIN THE COMPLETE BINARY TREE.**

- A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

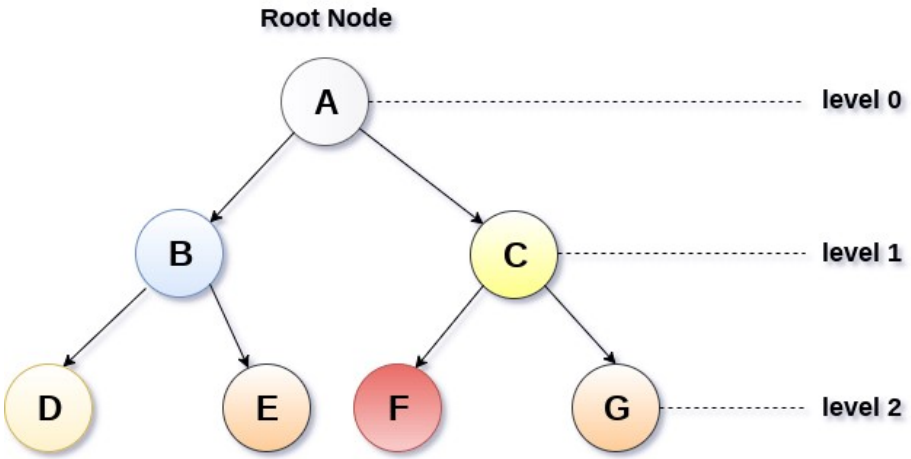


A Binary Tree node contains following parts.

- Data
- Pointer to left child
- Pointer to right child

Complete Binary Tree

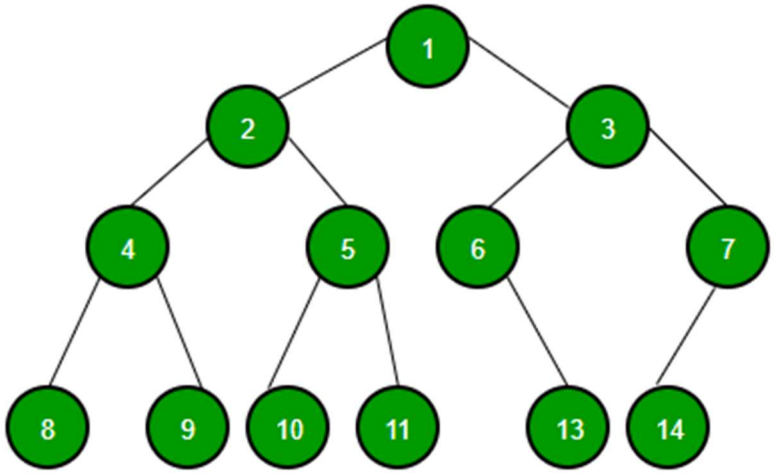
A Binary Tree is said to be a complete binary tree if all of the leaves are located at the same level d. A complete binary tree is a binary tree that contains exactly 2^l nodes at each level between level 0 and d. The total number of nodes in a complete binary tree with depth d is 2^{d+1}-1 where leaf nodes are 2^{d} while non-leaf nodes are 2^{d}-1.



Complete Binary Tree

7. **WHAT IS BINARY TREE AND EXPLAIN THE EXTENDED BINARY TREE**

- A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



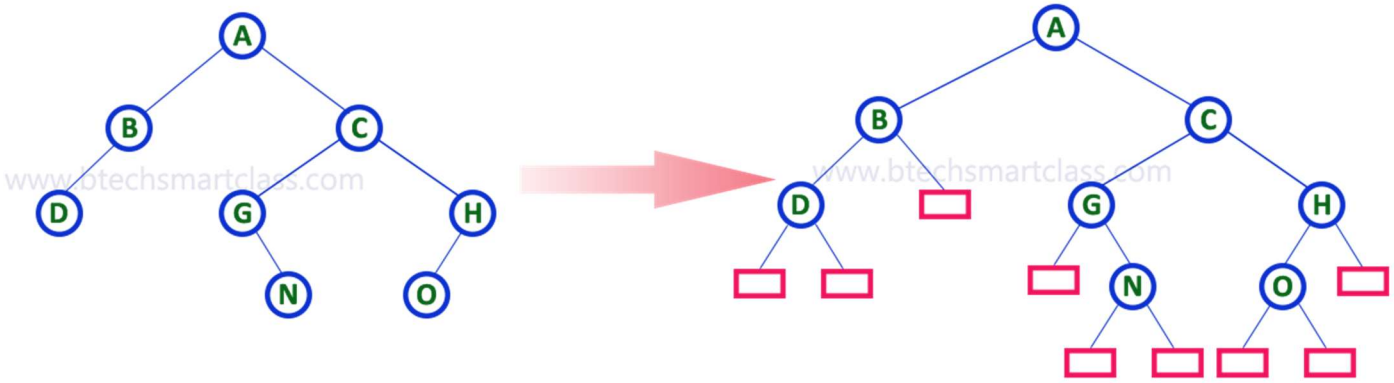
A Binary Tree node contains following parts.

- Data
- Pointer to left child
- Pointer to right child

Extended Binary Tree

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.

The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.



8. EXPLAIN THE BINARY TREE TRAVERSALS.

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

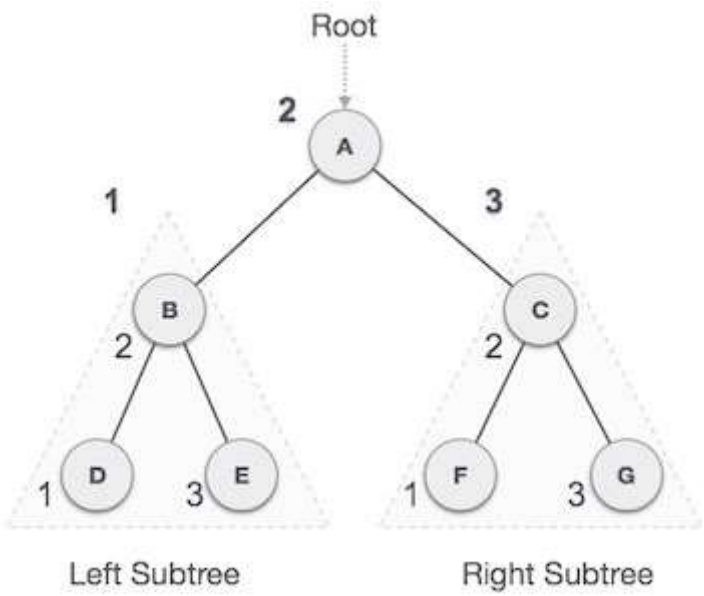
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

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 inorder traversal of this tree will be −

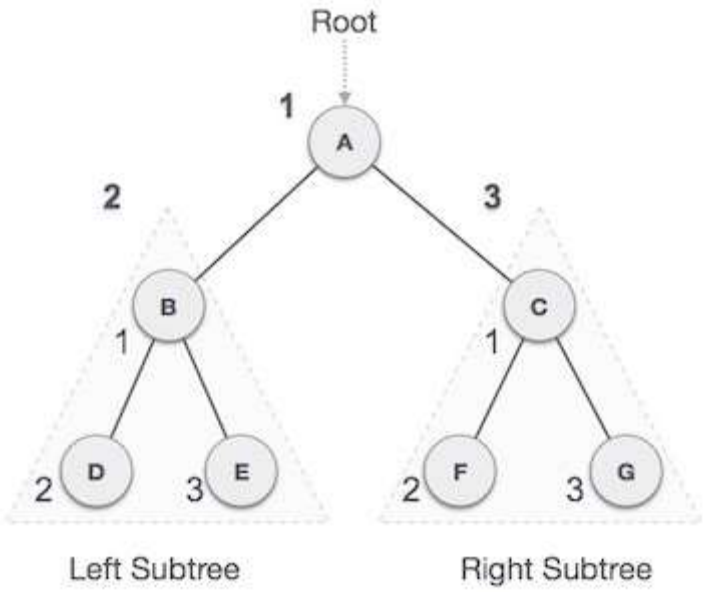
$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Algorithm

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.

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 −

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

Algorithm

Until all nodes are traversed −

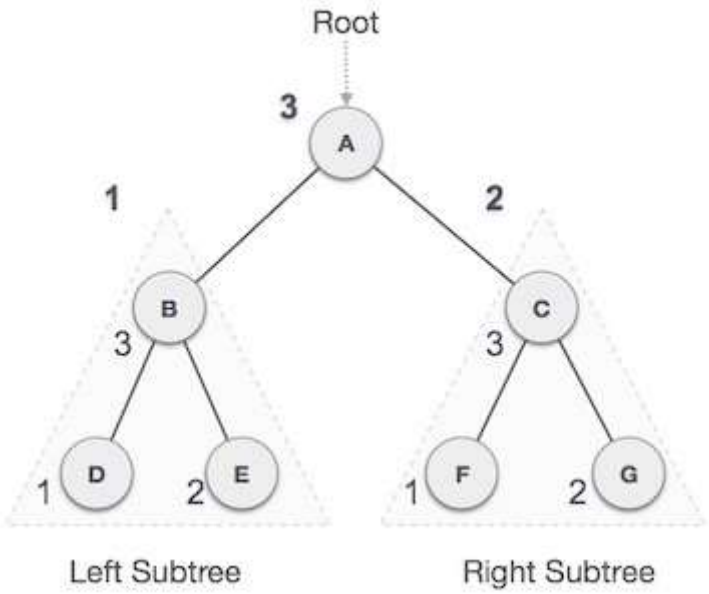
Step 1 − Visit root node.

Step 2 − Recursively traverse left subtree.

Step 3 − Recursively traverse right subtree.

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 −

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

Algorithm

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.

9. EXPLAIN B TREE.

- B-Tree is a self-balancing search tree. In most of the other self-balancing search trees , it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory.
- When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to the main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses.
- Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, the B-Tree node size is kept equal to the disk block size.
- Since the height of the B-tree is low so total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ..etc.
  Time Complexity of B-Tree:

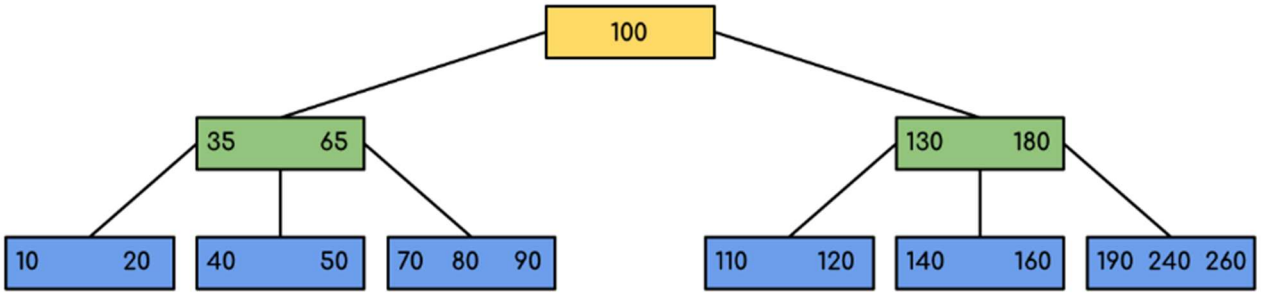
SR. NO.	ALGORITHM	TIME COMPLEXITY
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

“n” is the total number of elements in the B-tree.

Properties of B-Tree:

1. All leaves are at the same level.
2. A B-Tree is defined by the term *minimum degree* ‘t’. The value of t depends upon disk block size.
3. Every node except root must contain at least (ceiling)([t-1]/2) keys. The root may contain minimum 1 key.
4. All nodes (including root) may contain at most t – 1 keys.
5. Number of children of a node is equal to the number of keys in it plus 1.
6. All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
7. B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
8. Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(log n).

Following is an example of B-Tree of minimum order 5. Note that in practical B-Trees, the value of the minimum order is much more than 5.



We can see in the above diagram that all the leaf nodes are at the same level and all non-leaf have no empty sub-tree and have keys one less than the number of their children.

10. EXPLAIN THE B+ TREE.

- In order, to implement dynamic multilevel indexing, B-tree and B+ tree are generally employed. The drawback of B-tree used for indexing, however is that it stores the data pointer (a pointer to the disk file block containing the key value), corresponding to a particular key value, along with that key value in the node of a B-tree. This technique, greatly reduces the number of entries that can be packed into a node of a B-tree, thereby contributing to the increase in the number of levels in the B-tree, hence increasing the search time of a record.
- B+ tree eliminates the above drawback by storing data pointers only at the leaf nodes of the tree. Thus, the structure of leaf nodes of a B+ tree is quite different from the structure of internal nodes of the B tree. It may be noted here that, since data pointers are present only at the leaf nodes, the leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, in order to access them. Moreover, the leaf nodes are linked to provide ordered access to the records. The leaf nodes, therefore form the first level of index, with the internal nodes forming the other levels of a multilevel index. Some of the key values of the leaf nodes also appear in the internal nodes, to simply act as a medium to control the searching of a record.

From the above discussion it is apparent that a B+ tree, unlike a B-tree has two orders, ‘a’ and ‘b’, one for the internal nodes and the other for the external (or leaf) nodes.

The structure of the internal nodes of a B+ tree of order ‘a’ is as follows:

1. Each internal node is of the form :
   $<P_1, K_1, P_2, K_2, ….., P_{c-1}, K_{c-1}, P_c>$
   where c <= a and each **P_i is a tree pointer (i.e points to another node of the tree)** and, each **K_i is a key value** (see diagram-I for reference).

2. Every internal node has : $K_1 < K_2 < …. < K_{c-1}$
3. For each search field values ‘X’ in the sub-tree pointed at by P_i, the following condition holds :
   $K_{i-1} < X <= K_i$, for $1 < i < c$ and,
   $K_{i-1} < X$, for $i = c$
   (See diagram I for reference)
4. Each internal nodes has at most ‘a’ tree pointers.
5. The root node has, at least two tree pointers, while the other internal nodes have at least \ceil(a/2) tree pointers each.
6. If any internal node has ‘c’ pointers, c <= a, then it has 'c – 1' key values.

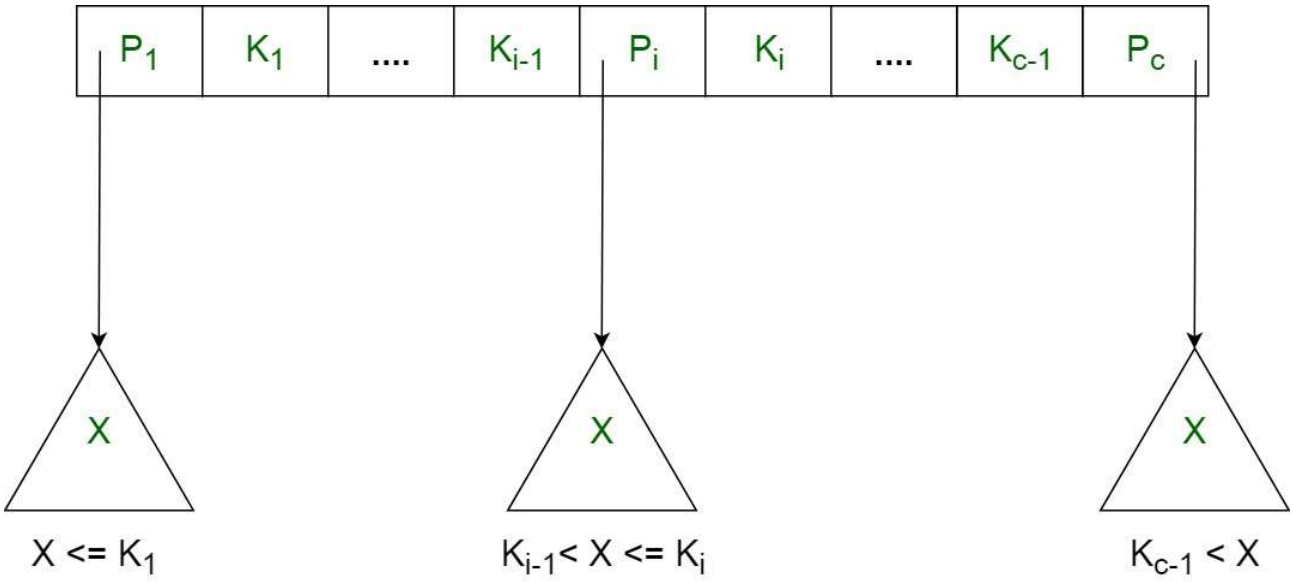


Diagram-I

The structure of the leaf nodes of a B+ tree of order 'b' is as follows:

1. Each leaf node is of the form :
   $<<K_1, D_1>, <K_2, D_2>, ..., <K_{c-1}, D_{c-1}>, P_{next}>$
   where c <= b and each **D_i is a data pointer (i.e points to actual record in the disk whose key value is K_i or to a disk file block containing that record)** and, each **K_i is a key value** and, **P_{next} points to next leaf node in the B+ tree** (see diagram II for reference).
2. Every leaf node has : $K_1 < K_2 < .... < K_{c-1}$, c <= b
3. Each leaf node has at least \ceil(b/2) values.
4. All leaf nodes are at same level.

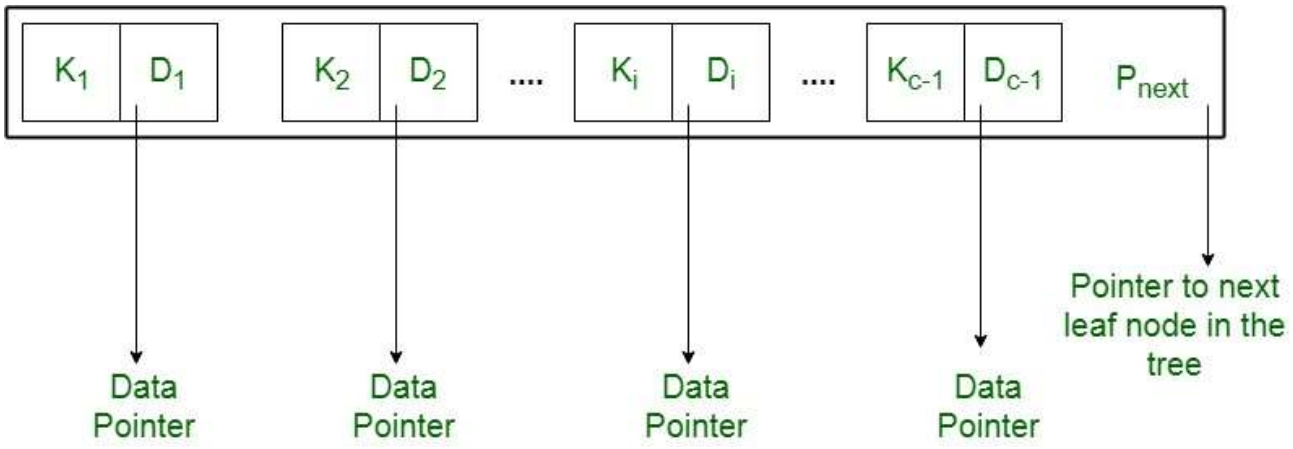
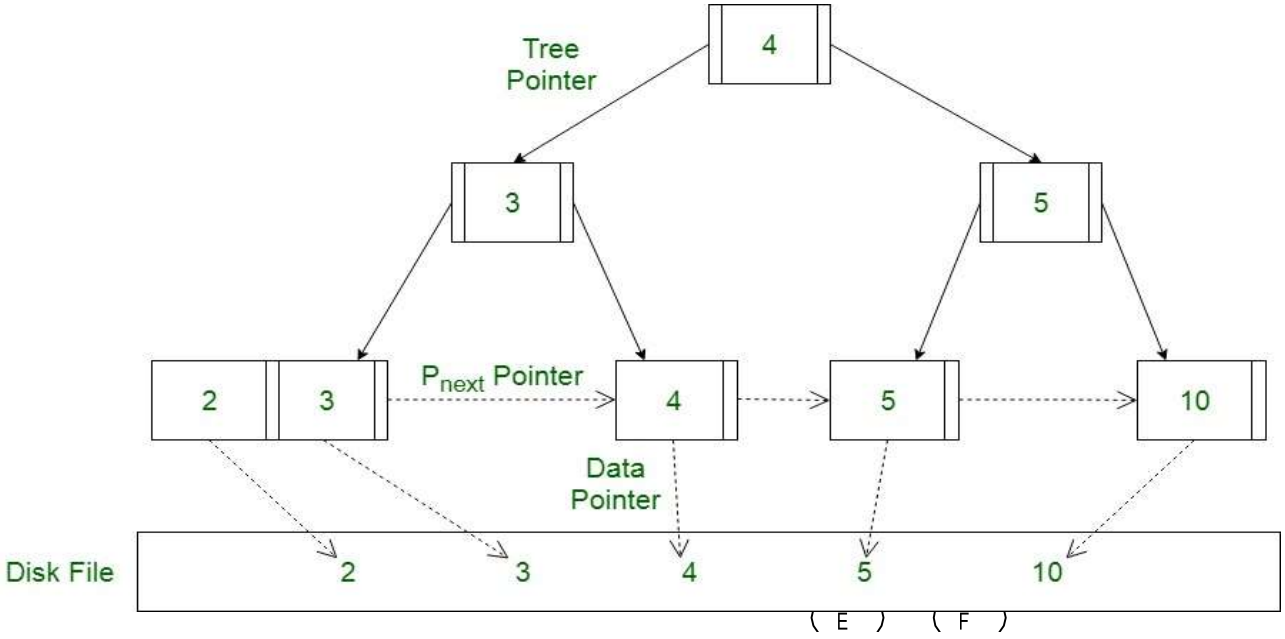


Diagram-II

Using the P_{next} pointer it is viable to traverse all the leaf nodes, just like a linked list, thereby achieving ordered access to the records stored in the disk.

A Diagram of B+ Tree –



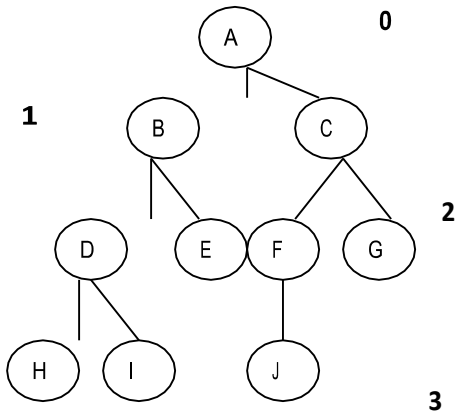
10 Marks

1. **Explain about Basic Tree Terminology with need diagrams.**

Tree: is defined as a finite set of one or more nodes such that

- ✓ There is one specially designated node called ROOT.
- ✓ The remaining nodes are partitioned into a collection of sub-trees of the root each of which is also a tree.

Example LEVEL



NODE: stands for item of information.

The nodes of a tree have a parent-child relationship. The root does not have a parent; but each one of the other nodes has a parent node associated to it. A node may or may not have children is called a **leaf node** or **terminal nodes**.

A line from a parent to a child node is called a branch. If a tree has n nodes, one of which is the root

there would be n-1 branches.

The number of sub-trees of a node is called its degree. The degree of A is 2, F is 1 and J is zero. The leaf node is having degree zero and other nodes are referred as **non-terminals.**

The degree of a tree is the maximum degree of the nodes in the tree.

Nodes with the same parent are called siblings. Here D & E are all **siblings.** H & I are also siblings.

The **level of a node** is defined by initially letting the root be at level zero. If a node is al level l, there its children are at level l+1.

The **height or depth** of a tree is defined to be the maximum level of any node in the tree.

A set of trees is called **forest;** if we remove the root of a tree we get a forest. In the above fig, if we remove A, we get a forest with three trees.

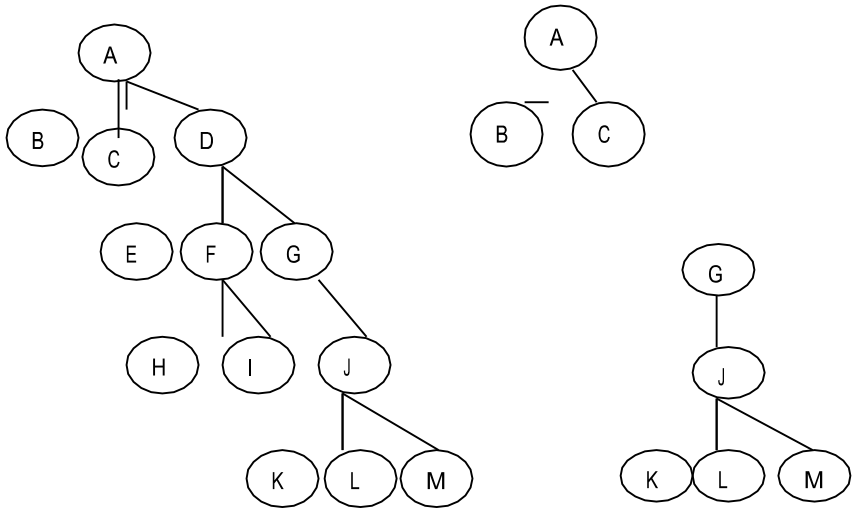


FIG: Tree FIG: Forest (Sub-trees)

Properties of a Tree

1. Any node can be the root of the tree and each node in a tree has the property that there is exactly one path connecting that node with every other node in the tree.

2. The tree in which the root is identified is called a rooted tree; a tree in which the root is not identified is called a free tree.

3. Each node, except the root, has a unique parent.

Binary Trees

A binary tree is a tree, which is, either empty or consists of a root node and two disjoint binary trees called the left sub-tree and right sub-tree.

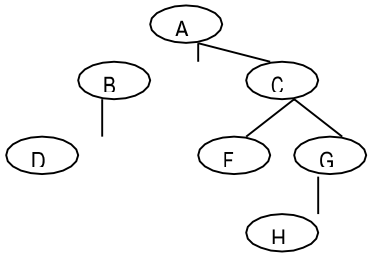


FIG: Binary Tree

In a binary tree, no node can have more than two children. So every binary tree is a tree, not every tree is a binary tree.

A **complete binary tree** is a binary tree in which all interval nodes have degree and all leaves are at the same level.

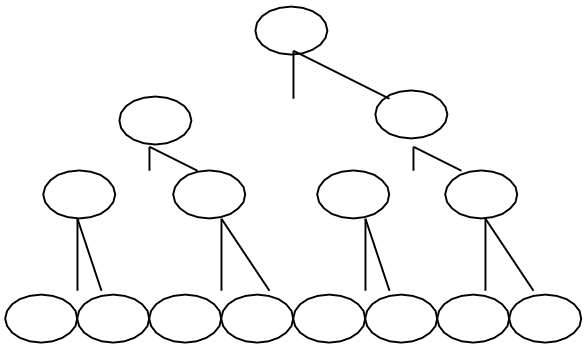


FIG: Complete binary tree

If every non-leaf node in a binary tree has non-empty left and right sub-trees, the tree is termed as **strictly binary tree.**

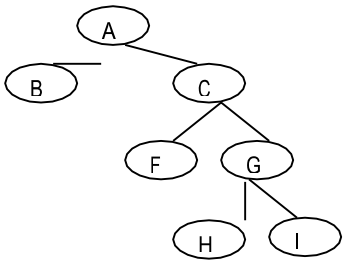


FIG: Strictly binary tree

If A is the root of a binary tree and B is the root of its left or right sub-tree, then A is said to be the parent of B and B is said to be the left or right child of A.

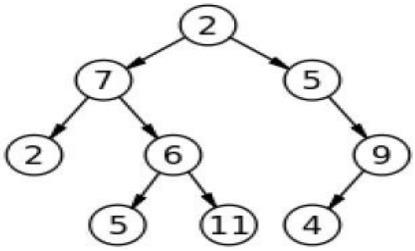
Node n1 is an **ancestor** of node n2, if n1 is either the parent of n2 or the parent of some ancestor of n2. Here n2 is a **descendant** of n1, a node n2 is a **left descendant** of node n1 if n2 is either the left child of n1 or a descendant of the left child ofn1.

A **right descendant** may be similarly defined the number of nodes at level i is 2^i. For a complete binary tree with k levels contains 2i **nodes.**

2. WRITE BRIEFLY ABOUT THE NODE REPRESENTATION OF BINARY TREES.

Binary Tree:

A binary tree is a tree in which each node has atmost 2 childrens. (i.e) a node can have 0 child or 1 child or 2 child. A node must not have more than 2 childrens.



1. Linear Representation Of A Binary Tree

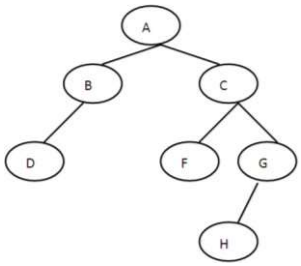
The linear representation method of implementing a binary tree uses a one-dimensional array of size ((2^d+1)-1) where d is the depth of the tree.

Once the size of the array has been determined the following method is used to represent the tree.

1. Store the root in 1st location of the array.
2. If a node is in location n of the array store its left child at location 2n and its right child at (2n+1).

In c, arrays start at position 0; therefore instead of numbering the trees nodes from 1 to n, we number them from 0 to n-1. The two child of a node at position P are in positions 2P+1 and 2P+2.

The following figure illustrates arrays that represent the almost complete binary trees.



0	1	2	3	4	5	6	7	8
A	B	C	D	E	F	G	H	

We can extend this array representation of almost complete binary trees to an array representation of binary trees generally.

The following fig (A) illustrates binary tree and fig (B) illustrates the almost complete binary tree of fig (a). Finally fig(C) illustrates the array implementation of the almost complete binary tree.

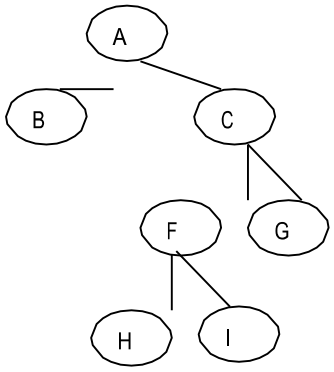


Fig (A) Binary tree

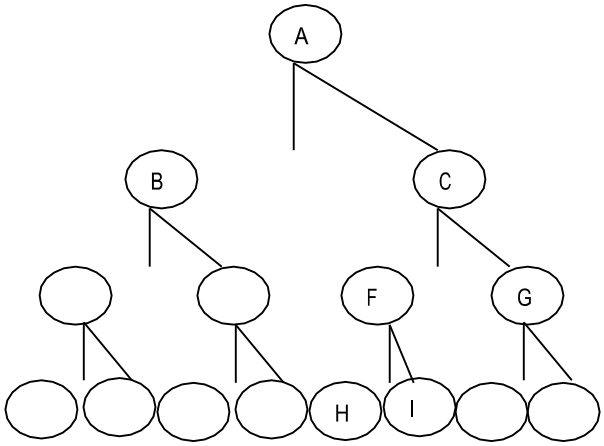


Fig (B) Almost a complete binary tree

0	1	2	3	4	5	6	7	8	9	10	11	12
A	B	C			F	G					H	I

Fig (C) Array Representation

Advantages

1. Given a child node, its parent node can be determined immediately. If a child node is at location N in the array, then its parent is at location N/2.
2. It can be implemented easily in languages in which only static memory allocation is directly available.

Disadvantages

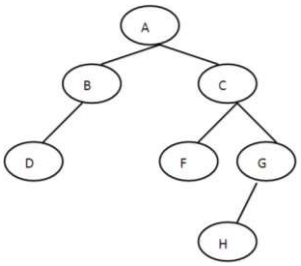
1. Insertion or deletion of a node causes considerable data movement up and down the array, using an excessive amount of processing time.
2. Wastage of memory due to partially filled trees.

2. Linked List Representation

Linked lists most commonly represent binary trees. Each node can be considered as having 3 elementary fields : a data field, left pointer, pointing to left sub-tree and right pointer pointing to the

right sub-tree.

The following figure is an example of linked storage representation of a binary tree.



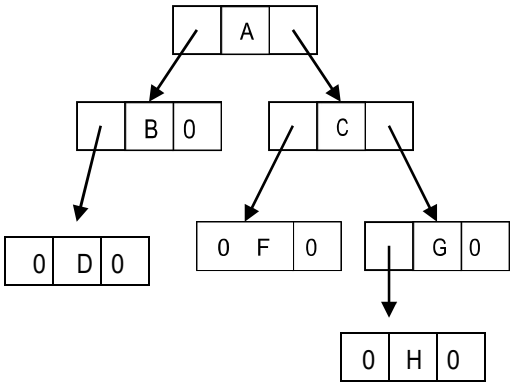


FIG:Binarytree

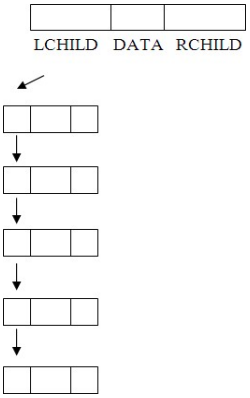
FIG: Linked representation of a binary tree

disadvantages

- Wasted memory space is well pointers.
- Given a node, it is difficult to determined to parent.
- Its implementation algorithm is more difficult in languages that do not offer dynamic storage techniques.

Linked Representation: -

The problems of sequential representation can be easily overcome through the use of a linked representation. Each node will have three fields LCHILD, DATA and RCHILD as represented below



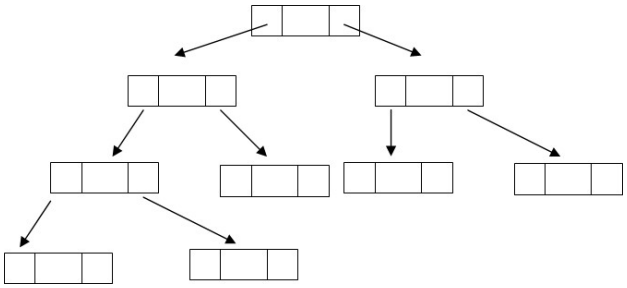


Fig (a)

Fig (b)

In most applications it is adequate. But this structure make it difficult to determine the parent of a node since this leads only to the forward movement of the links.

Using the linked implementation,

we may declare, stuct nodetype

```
{
int info;
struct nodetype *left; struct nodetype *right; struct
nodetype *father;
};
```

typedef struct nodetype *NODEPTR;

This representation is called **dynamic node representation.** Under this representation,

info(p) would be implemented by reference **p→info,**

left(p) would be implemented by reference **p→left,**

right(p) would be implemented by reference **p→right**,

father(p) would be implemented by reference **p→father**.

Array Representation:

Each node contains **info, left, right** and **father** fields. The left, right and father fields of a node point to the node's left son, right son and father respectively.

Using the array implementation, we may declare,

```
#define NUMNODES 100 struct nodetype
  {
  int info; int left; int right; int father;
  };
struct nodetype node[NUMNODES];
```

This representation is called linked array representation.

Under this representation,

info(p) would be implemented by reference **node[p].info,**

left(p) would be implemented by reference **node[p].left,**

right(p) would be implemented by reference **node[p].right,**

father(p) would be implemented by reference **node[p].father** respectively.

The operations,

isleft(p) can be implemented in terms of the operation **left(p)**

isright(p) can be implemented in terms of the operation **right(p)**

Example: -

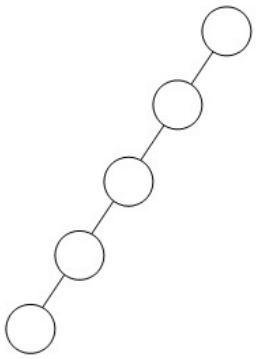


Fig (a)

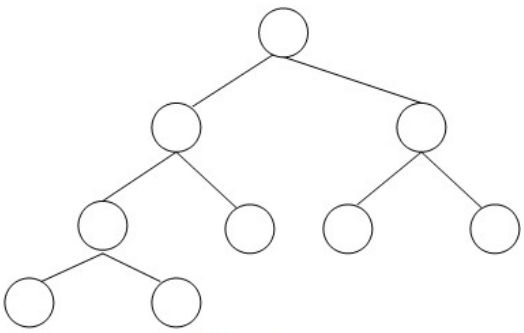
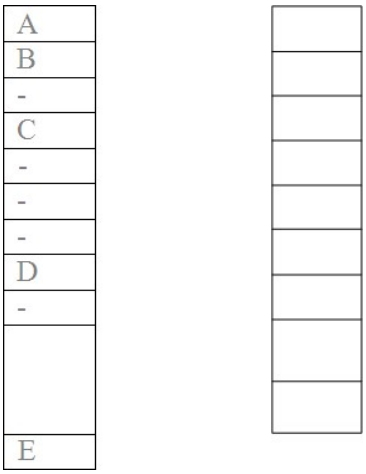


Fig (b)

The above trees can be represented in memory sequentially as follows



The above representation appears to be good for complete binary trees and wasteful for many other binary trees. In addition, the insertion or deletion of nodes from the middle of a tree requires the insertion of many nodes to reflect the change in level number of these nodes.

3. What is Binary Tree? Explain its operation in detail with example. (Nov 2012, April 2011, April 2012,April 2013)

Binary Tree:

A binary tree is a tree in which each node has atmost 2 childrens. (i.e) a node can have 0 child or 1 child or 2 child. A node must not have more than 2 childrens.

Eg:

BINARY SEARCH TREE OPERATIONS:

The basic operation on a binary search tree(BST) include, creating a BST, inserting an element into BST, deleting an element from BST, searching an element and printing element of BST in ascending order.

The ADT specification of a BST:

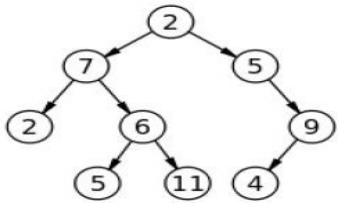
ADT BST

{

Create BST() : Create an empty BST;

Insert(elt) : Insert elt into the BST;

Search(elt,x) : Search for the presence of element elt and Set x=elt, return true if elt is found, else



return false.

FindMin() : Find minimum element;

FindMax() : Find maximum element;

Ordered Output() : Output elements of BST in ascending order; Delete(elt,x) : Delete elt and set x = elt;
}

Inserting an element into Binary Search Tree

```
Algorithm InsertBST(int elt, NODE *T)
[ elt is the element to be inserted and T is the pointer to the root of the
 tree] If (T= = NULL) then
    Create a one-node tree and
return Else if (elt<key) then
    InsertBST(elt,
T→lchild) Else


    if(elt>key) then
    InsertBST(elt, T→rchild)
Else
    " element is
already exist return T
End
```

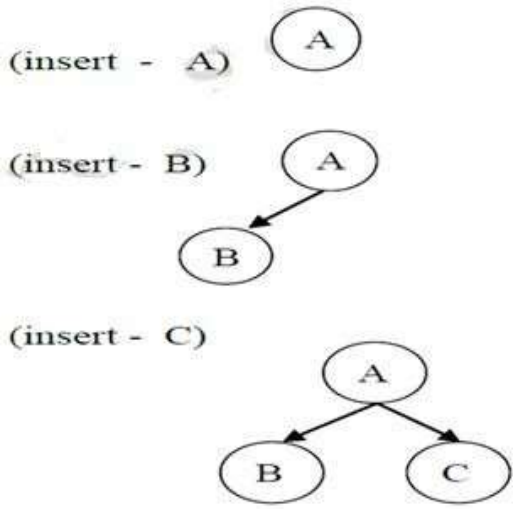
C coding to Insert element into a BST

```
struct node
    {
    int info;
    struct node *lchild; struct node *rchild;
    };
typedef stuct node NODE;
NODE *InsertBST(int elt, NODE *T)
    {
    if(T = = NULL)
      {
      T = (NODE *)malloc(sizeof(NODE)); if ( T = = NULL)
        printf ( " No memory error"); else
        {
        t→info = elt; t→lchild = NULL;
        t→rchild = NULL;
        }
      }
```

```
else if ( elt < T→info)
  t→lchild = InsertBST(elt, t→lchild); else if ( elt >T→info)
  t→rchild = InsertBST( elt, t→rchild); return T;
}
```

Inserting an External node:

The first node that we add is an external node'node-A'. To add a new node'node-B' after 'node-A', 'node-A' assigns the newly inserted node 'node-B' as one of its child and the added new node'node-B' assigns 'node-A' as its parent.



Searching an element in BST

Searching an element in BST is similar to insertion operation, but they only return the pointer to the node that contains the key value or if element is not, a NULL is return;

Searching start from the root of the tree;

If the search key value is less than that in root, then the search is left subtree;

If the search key value is greater than that in root, then the search is right subtree;

This searching should continue till the node with the search key value or null pointer(end of the branch) is reached.

In case null pointer(null left/righ chile) is reached, it is an indication of the absence of the node.

```
NODE * SearchBST(int elt, NODE *T)
{
if(T = = NULL)
  return NULL;

 if ( elt < T→info)
  return SearchBST(elt,
t→lchild); else if ( elt >
T→info)
```

```
    return SearchBST( elt, t→rchild); else
    return T;
  }
```

Finding Minimum Element in a BST

Minimum element lies as the left most node in the left most branch starting from the root. To reach the node with minimum value, we need to traverse the tree from root along the left branch till we get a node with a null / empty left subtree.

Algorithm FindMin(NODE * T)

1. If Tree is null then return NULL;
2. if lchild(Tree) is null then return tree

```
    else
    return FindMin(T→lchild)
```

3. End

```
NODE * FindMin( NODE *T )
  {
  if(T = = NULL)
    return NULL;

  if(T→lchild = = NULL) return tree;
  else
    return FindMin(Tree→lchild);
  }
```

Finding Maximum Element in a BST

Maximum element lies as the right most node in the right most branch starting from the root. To reach the node with maximum value, we need to traverse the tree from root along the right branch till we get a node with a null / empty right subtree.

Algorithm FindMax(NODE * T)

1. If Tree is null

then return

NULL;

2. if rchild(Tree) is null

then return tree

```
    else
    return FindMax(T→rchild)
```

3. End

```
NODE * FindMax( NODE *T )
```

```
{
if(T = = NULL)
  return NULL;
 if(T→rchild = =
  NULL) return
  tree;
else
  return FindMax(Tree→rchild);  }
```

DELETING AN ELEMENT IN A BST

The node to be deleted can fall into any one of the following categories;

1. Node may not have any children (ie, it is a leafnode)
2. Node may have only on child (either left / right child)
3. Node may have two children (both left and

right) Algorithm DeleteBST(int elt, NODE * T)

1. If Tree is null then

print "Element is not found"

2. If elt is less than info(Tree)then

locate element in left subtree and delete it else if elt is greater than info(Tree) then locate element in right subtree and delete it else if (both left and right child are not NULL) then /* node with two children */ begin

Locate minimum element in the right subtree Replace elt by this value

Delete min element in right subtree and move the remaining tree as its right childend else

if leftsubtree is Null then

/* has only right subtree or both subtree Null */ replace node by its rchild

else

if right subtree is Null then replace node by its left child\

end

free memory allocated to min node end

return Tree End

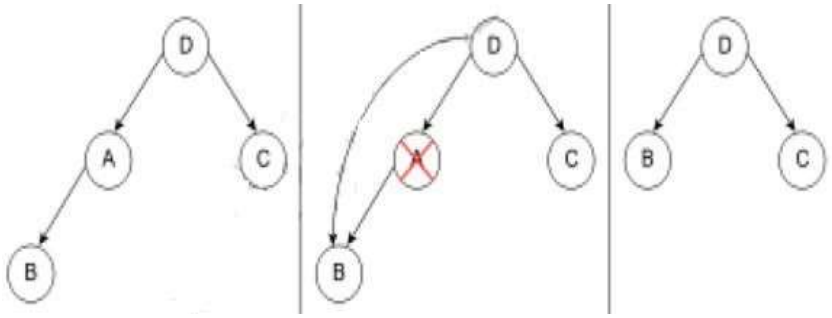
```
NODE *DeleteBST(int elt, NODE * T)
{
NODE * minElt;

if(T = = NULL)
    printf(“ element not found \n”); else if ( elt <T→info)
    T→lchild = DeleteBST(elt,
T→lchild); else if ( elt >T→info)
    T→rchild = DeleteBST(elt, T→rchild); else
    if(T→lchild && T→ rchild)
      {
      minElt = FindMin(T→rchild); T→info = minElt→info;
      T→rchild = DeleteBST(T→info, T→rchild);
      }
    else
      {
      minElt = Tree;
      if (T→lchild = = NULL) T = T→rchild;
      else if ( T→rchild = = NULL) T = T→lchild;
      Free (minElt);
      }
return T;
}
```

Deletion:

Deletion is the process whereby a node is deleted from the tree. Only certain nodes in a binary tree can be deleted easily.

(i.e) the node with 0 or 1 children can be deleted, but, the node with 2 childrens cannot be deleted easily



3. Explain Tree Traversal techniques with its implementation and example. April 2012) (Feb/Mar 2021)

Traversals of a Binary Tree:

Traversing a tree means processing the tree such that each node is visited only one.

Traversal of a binary tree is useful in many applications. For example, in searching for particulars nodes compilers commonly build binary trees in the process of scanning, parsing, generating code and evaluation of arithmetic expression.

Let T be a binary tree, there are a number of a different ways to proceed. The methods differ primarily in the order in which they visit the nodes. The three different traversals of T are Inorder, Post order and Preorder traversals.

In C, each node is defined as a structure of the following

form: struct node

```
{
int info;
struct node
*lchild; struct
node *rchild;
}
typedef struct node NODE;
```

I. Inorder Traversal:

It follows the general strategy of Left-Root-Right. In this traversal, if T is not empty, we first traverse (in order) the left sub-tree;

Then visit the root node of T and then traverse the right sub-tree. Consider the binary tree given in the following figure.

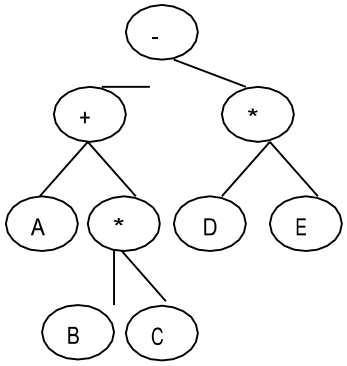


FIG: Expression Tree

This is an example of an expression tree for (A+B*C)-(D*E).

A binary tree can be used to represent arithmetic expressions if the node value can be either

operators or operand values and are such that

- Each operator node has exactly two branches.
- Each operand node has no branches; such trees are called expression trees.

Tree, T at the start is rooted at '-';

- Since left(T) is not empty; current T becomes rooted at '+';
- Since left(T) is not empty; current T becomes rooted at 'A';
- Since left (T) is empty; we visit root i.e. A.
- We access T root i.e. '+'.
- We now perform in-order traversal of right (T) current T becomes rooted at '*'.
- Since left(T) is not empty; current T becomes rooted at 'B' since left(T) is empty; we visit it to root i.e. B;

cluck for right (T) which is empty, therefore we move back to parent tree. We visit its root i.e. '*'.

Now Inorder traversal of right (T) is performed; which would give us
'C'. We visit T's root i.e., 'D' and perform in-order traversal of right (T); which would give us '* and E'.
Therefore the complete listing is
A+B*C-D*E
We may note that expression is in infix notation. The in-order traversal produces a left expression then prints out the operator at root and then a right expression.

Steps :

1. Traverse left subtree in inorder
4. Process root node
5. Traverse right subtree in inorder

Algorithm

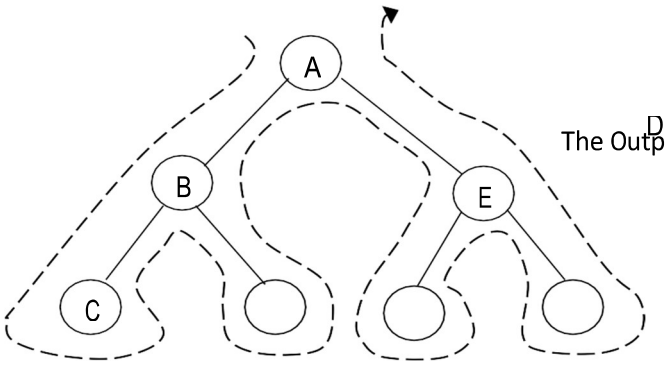
Algorithm inoder traversal (Bin-Tree T)
Begin
If (not empty (T)) then
Inorder_traversal (left subtree (T))
Print (info (T)) / * process node */
Inorder_traversal (right subtree (T))
E

C Coding

```
void inorder_traversal ( NODE * T)
{
if( T ! = NULL)
inorder_traversal(T->lchild);
        printf("%d\t",T->info); Begin
inorder_traversal(T->rchild);
}
}
```

II. Postorder Traversal:

In this traversal we first traverse Left(T) ; then traverse Right(T) ; and finally visit root. It is a Left-



The Output is : C → B → D →A →E → F

Right- Root strategy i.e.,

Traverse the left sub-tree in

Postorder Traverse the right sub-

tree in Postorder Visit the root

For example, a Postorder traversal of the fig: Expression tree

would be ABC*+DE*-

We can see that it is the prefix notation of the expression (A+(B*C))-

(D*E). Tree, T, at the start is rooted at '-' ;

- Since left (T) is not empty ; current T becomes rooted at '+';
- Since left (T) is not empty ; current T becomes rooted at 'A';
- Since left (T) is empty ; we visit right (T) '*'
- Since Right (T) is empty ; we visit root i.e. A
- We access right T's root it*
- Since left (T) is not empty ; we visit right (T)
- Since right (T) is empty we visit root i.e., B
- Since right (T) is not empty ; current T becomes rooted at 'C'
- Since left (T) & right (T) is empty , we visit root i.e. 'C'
- Visit root i.e. '*'
- Visit root i.e., '+'
- Since right (T) is not empty ; current T becomes rooted at '*'
- Since left (T) is not empty ; current T becomes rooted at 'D'
- Since left (T) and right (T) are empty we visit root i.e., 'D'
- Since right (T) is not empty ; current T becomes rooted at 'E'.
- Since left (T) & right (T) are empty ; we visit root i.e., 'E'
- Visit root is '*'
- Visit root is '-'

Therefore, the complete

listing is ABC*+DE*-

Steps : 1. Traverse left subtree in postorder

2. Traverse right subtree in postorder

3. process root node

Algorithm

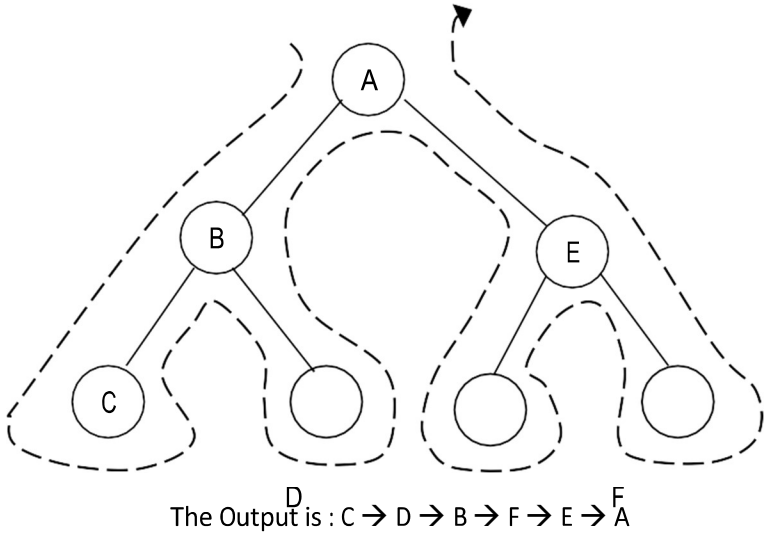
```
Postorder Traversal
                                        {
Algorithm postorder traversal (Bin-Tree T)
                                      {
Begin
If ( not empty (T) ) then
Begin
    Postorder_traversal ( left subtree ( T ) )
    Postorder_traversal ( right subtree( T))      }
    Print ( Info ( T ) ) / * process node */
End
End
```

C function

```
void postorder_traversal ( NODE * T)

  if( T ! = NULL)

  {
   postorder_traversal(T->lchild);
        postorder_traversal(T->rchile);
   printf("%d \t", T->info);
  }
```



The Output is : C → D → B → F → E → A

III. PREORDER TRAVERSAL:

In this traversal, we first visit the root; then traverse left (T) and finally right (T). It is a Root-Left-Right strategy i.e.

Visit the root

Traverse the left sub-tree in preorder

Traverse the right sub tree in preorder

For example, a preorder traversal of the fig: Expression tree would be

-+A*BC*DE

We can see that it is a prefix notation of the expression (A+(B*C))-(D*E)

Tree, T, at the start is rooted at '-';

- *since it visits the root first , it is displayed i.e. '-' is displayed*
- *Next left traversal is done. Here the current T becomes rooted at '+'*
- *Since left(T) is not empty; current T becomes rooted at A*
- *Since left (T) is empty we visit right (T) i.e. '*' and T is rooted here.*
- *Since left (T) is not empty , current T gets rooted at B*
- *Since left (T) is empty, we visit right (T) i.e. 'C' and T is at present rooted here*
- *Since left (T) is empty and no right (T) we move to the root '*'*
- *Visit root '+'*
- *Now the left sub tree is complete so we move to right sub tree i.e. T is rooted at '*'*
- *Since left (T) is not empty , current T becomes rooted at D*
- *Since left (T) is empty, T gets rooted at right (T) i.e. E.*

*Thus the complete listing is -+A*BC*DE*

Steps : 1. Process root node

2. Traverse left subtree in preorder
3. Traverse right subtree in preorder

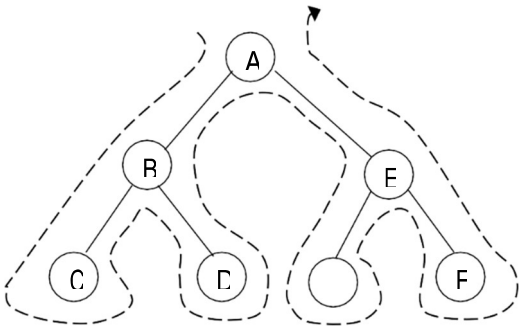
Algorithm

Algorithm preorder traversal (Bin-Tree T)

Begin
If (not empty (T)) then
Begin
    Print (info (T)) / * process node * /
    Preoder traversal (left subtree (T))
    Inorder traversal (right subtree (T))
End
End

C function

```
void preorder_traversal ( NODE * T)
{
  if( T ! = NULL)
     {
  printf("%d \t", T->info);
            preorder_traversal(T->lchild);
  preorder_traversal(T->rchild);
       }
  }
```



Output is : A →B → C → D → E → F

5. Explain in detail about Application of Tress with examples.

I. Set Representation

In this section we study the use of trees in the representation of sets. We shall assume that the elements of the sets are the numbers 1, 2, 3…n. These numbers might, in practice, be indices into symbol tables where the actual names of the elements are stored. We shall assume that the sets being represented are pair wise disjoint: i.e. if Si and Sj, I != j, are two sets then there is no element which is in both Si and Sj. For example if we have 10 elements numbered from 1 through 10, they may be partitioned into three disjoint sets S1 = {1,7,8,9}; S2 = {2, 3, 10}; S3 = {3, 4,6}. The operations we wish to perform on these sets are

- Disjoint set union…if Si and Sj are two disjoint sets, then their union on Si U Sj

  = {all elements x such is in Si or Sj)

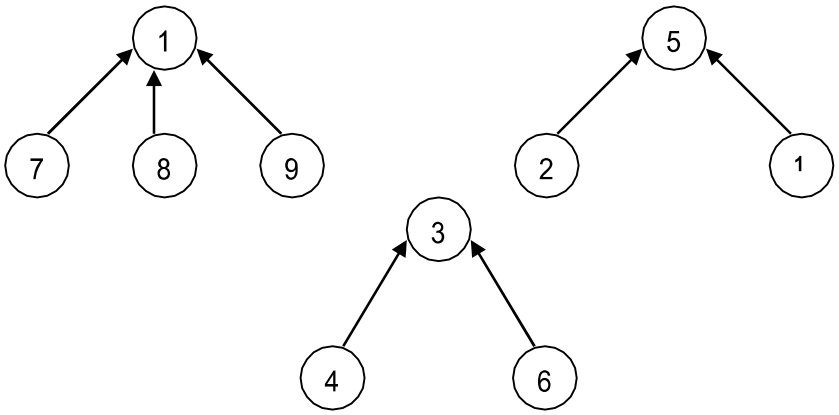
Thus,

S1 U S2 = {1, 7, 8, 9, 2, 5, 10}

Since we have assumed that all sets are disjoint. Following the union of Si and Sj we can assume that the sets Si and Sj no longer exist independently i.e., they are replaced by Si U Sj in the collection of sets.

- Find(i)…find the set containing element i. Thus, 4 is in set S3 and 9 is in set S1.

The sets will be represented by trees. One possible representation for the sets S1, S2 and S3 is is:



In presenting the UNION and FIND algorithms we shall ignore the actual set names and just identify sets by the roots of the trees representing them. This will simplify the discussion. The transition to set names is easy. If we determine that element I is in a tree with root j and I has a pointer to entry k in the set name table, then the set name is just(NAME(k)). If we wish to union sets Si and Sj then we wish to union the trees with roots POINTERS(Si) and POINTERS(Sj).

As we shall see in many application the set name is just the element at the root. The operation of FIND(i) now becomes; determine the root of the tree containing element i. UNION(I, j) requires two trees with roots I and j to be joined. We shall assume that the nodes in the trees are numbered 1 through n so that the node index corresponds to the element index. Thus element 6 is represented by the node with index 6. Consequently, each node needs only one field: The PARENT field to link to its parent. Root nodes have a PARENT field of zero. Based on the above discussion our first attempt arriving at UNION, FIND algorithms would result in the algorithms U and F below.

Algorithm for FIND / UNION operation

```
Procedure U(I, j)
    //replace the disjoint sets with roots I and j, i!=j with their union//
PARENT(i) ← j
End U

Procedure F(i)
    //Find the root j of the tree containing element i//
j← i
while PARENT(j) > 0 do
      j← PARENT(j)
end
return(j)
end F
```

We can do much better if care is taken to avoid the creation of degenerate trees. In order to accomplish this we shall use a Weighting Rule for UNION(I, j). If the number of nodes in tree I is less than the number of nodes in tree j, then make j the parent of i, otherwise make i the parent of j. using this rule on the sequence of set unions given before we obtain the trees on next page remember that the arqument of UNION must both be roots. The time required to process all the n finds is only O(m) since in tis case the maximum level of any node is 2. This, however, is not the worst case.

The maximum level for any node is [log n]+1. First let us see how easy it is to implement the weighting rule. We need to know how many nodes there are in any tree. To do this easily, we maintain a count field in the toot of every tree. If I is a root node, then COUNT(i) = number of nodes in that tree. The count can be maintained in the PARENT field as a negative number. This is equivalent to using a one bit field to distinguish a count from a pointer. No confusion is created as for all other nodes the PARENT is positive.

```
Procedure UNION(i, j)
    //union sets with roots i and j1 != j using the weighting rule//
x ← PARENT(i) + PARENT(j)

if PARENT(i) > PARENT(j) then
    [PARENT(i) ← j
     PARENT(j) ← x]
Else
    [PARENT(j) ←1
     PARENT(i) ← x]
End UNION
```

Similarly modifying the FIND algorithm by using the Collapsing Rule:

If j is a node on the path from I to its root and PARENT(j) != root(i) then set PARENT(j) ← root(i). The new algorithm becomes:

```
Procedure FIND(i)
    //find the root of the tree containing element i//
j←i
while PARENT(j)>0 do
    j←PARENT(j)
end
k←i
while k != j do
    t ← PARENT(k)
    PARENT(k) ←j
    K←t
end
return(j)
end FIND
```

II. Decision Trees

Another application of trees is decision making. Consider the well-known eight coins problem. Given coins a,b,c,d,e,f,g,h. we are told that one is a counterfeit and has a different weight than the others. We want to do so using a minimum number of comparisons and at the same time determine whether the false coin is heavier or lighter than the rest. The tree below represents a set of decisions by which we can get the answer to our problem. This is why it is called a decision tree.

The use of capital H or L means that counterfeit coin is heavier or lighter. Let us trace through one possible sequence. If a+b+c < d+e+f, then we know that the false coin is present among the six and is neither g nor h. If on our next measurement we find that a+d < b+e, then by interchanging d and b we have is not the culprit, and (ii) that b or d is also not the culprit. If a+d was equal to b+e, the c or f would be the counterfeit coin. Knowing at this point that either a or e is the counterfeit, we compare a with a good coin say. If a = b then e is heavy, otherwise a must be light.

By looking at this tree we see that all possibilities are covered, since there are 8 coins which can be heavy or light and there are 16 terminal nodes. Every path requires exactly 3 comparisons.

ALGORITHM:

```
Procedure COMP(x,y,z)
    // x is compared against the standard coin z//
if x > z then print (x 'heavy')
else print (y 'light')
end COMP
Procedure EIGHTCOINS
    //eight weights are input; the different one is discovered using only 3 comparison//
read (a,b,c,d,e,f,g,h)
case
    : a+b+c = d+e+f : if g>h then call COMP(g,h,a)
            else call COMP(h,g,a)
    : a+b+c > d+e+f : case
                : a+d = b+e : call COMP(c,f,a)
                :a+d > b+e : call COMP(a,e,b)
                :a+d < b+e : call COMP (b,d,a)
            end
    : a+b+c < d+e+f : case
                : a+d = b+e : call COMP(f,c,a)
                : a+d > b+e : call COMP(d,b,a)
                : a+d < b+e : call COMP(e,a,b)
             end
  end
end EIGHTCOINS
```

III. Game Tree

A game tree is like a search tree in many ways …

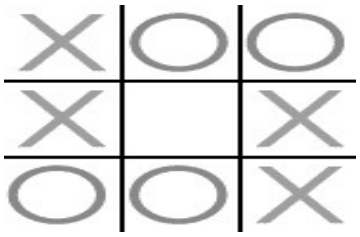
- Nodes are search states, with full details about a position
- characterize the arrangement of game pieces on the game board
- Edges between nodes correspond to moves
- leaf nodes correspond to a set of goals
- { win, lose, draw }
- Usually determined by a score for or against player
- at each node it is one or other player's turn to move
- A game tree is not like a search tree because you have an opponent!

Two-player games

The object of a search is to find a path from the starting state to a goal state

- In one-player games such as puzzle and logic problems you get to choose every move
  - e.g. solving a maze
- In two-player games you alternate moves with another player
  - competitive games
  - each player has their own goal
  - search technique must be different

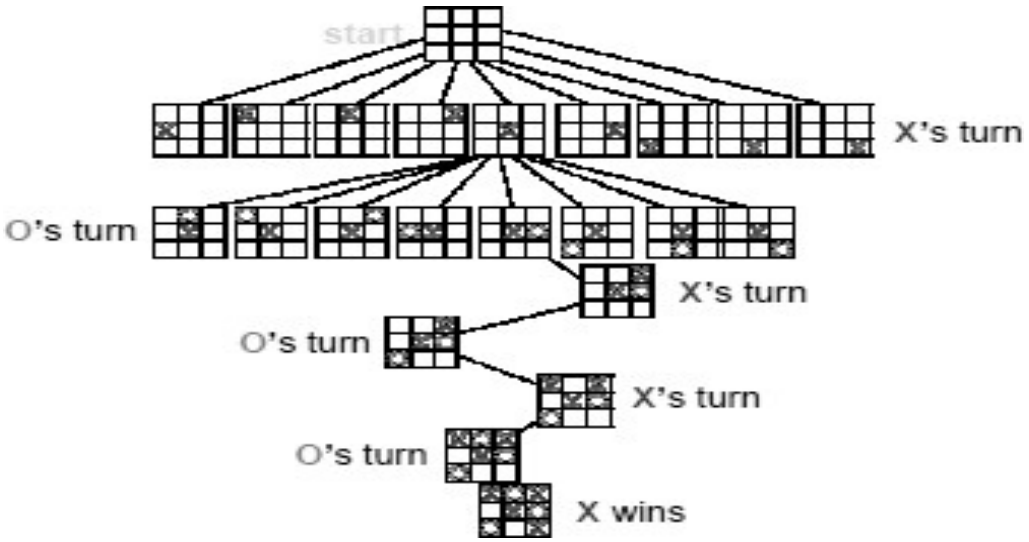
Eg. Tic-Tac-Toe



The first player is **X** and the second is **O**

- Object of game: get three of your symbol in a horizontal, vertical or diagonal row on a 3µ3 game board
- **X** always goes first
- Players alternate placing **X**s and **O**s on the game board
- Game ends when a player has three in a row (a wins) or all nine squares are filled (a draw)

Partial game tree for Tic-Tac-Toe



6. Write the Linked list implementation of Polynomial addition.

Polynomials:

One classic example of an ordered list is a polynomial.

Definition:

A polynomial is the sum of terms where each term consists of variable, coefficient and exponent.

Various operations which can be performed on the polynomial are,

- Addition of two polynomials
- Multiplication of two polynomials
- Evaluation of polynomials

The Polynomial Expression can be represented using Linked list by the following illustration,

The typical node will look like this,

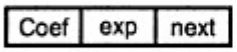
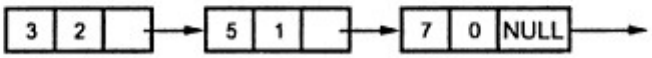


Fig. 3.13 Node of polynomial

For example : To represent $3x^2 + 5x + 7$ the link list will be,



Program for Addition of two polynomial using Linked List

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
struct list
{
    int coef;
    int pow;
    struct list *next;
};
struct list *p1=NULL,*p2=NULL,*p=NULL;

void generate(struct list *node)
{
    char c;
do
{
    printf("\n Enter a coefficient value:");
    scanf("%d",&node->coef);
    printf("\n Enter a power value:");
```

```
        scanf("%d", &node->pow);

        node->next=(struct list*)malloc(sizeof(struct list));
        node=node->next;
        node->next=NULL;

        printf("\n Want to continue? [Y/N]:");
        c=getch();
    }
    while(c=='y' || c=='Y');
    }
    void display(struct list *node)
    {
        while(node->next!=NULL)
        {
            printf("%dx^%d",node->coef,node->pow);
            node=node->next;
            if(node->next!=NULL)
            {
                    printf("+");
            }
        }
    }
    void add(struct list *p1,struct list *p2,struct list *p)
    {
        while(p1->next && p2->next)
        {
            if(p1->pow > p2-> pow)
            {
                    p-> pow = p1-> pow;
                    p->coef= p1-> coef;
                    p1= p1->next;
            }
            else if(p1->pow < p2->pow)
            {
                    p->pow=p2->pow;
                    p->coef=p2->coef;
                    p2=p2->next;
            }
            else
            {
                    p->pow=p1->pow;
                    p->coef=p1->coef+p2->coef;
```

```
                p1=p1->next;
                p2=p2->next;
        }

        p->next=(struct list *)malloc(sizeof(struct list));
        p=p->next;
        p->next=NULL;
    }

    while(p1->next || p2->next)
    {
        if(p1->next)
        {
                p->pow=p1->pow;
                p->coef=p1->coef;
                p1=p1->next;
        }
        if(p2->next)
        {
                p->pow=p2->pow;
                p->coef=p2->coef;
                p2=p2->next;
        }
        p->next=(struct list *)malloc(sizeof(struct list));
        p=p->next;
        p->next=NULL;
    }
}
void main() // Main method
{
    char ch;
    clrscr();
    do
    {
        p1=(struct list *)malloc(sizeof(struct list));
        p2=(struct list *)malloc(sizeof(struct list));
        p=(struct list *)malloc(sizeof(struct list));

        printf("\n Enter First No:");
        generate (p1);
        printf("\n Enter second No:");
        generate (p2);
```

```
        printf("\n First No is:");
        display(p1);
        printf("\n Second No is:");
        display(p2);

        add(p1,p2,p);
        printf("\n Addition of the polynomial is:");
        display(p);

        getch();


    }
    while(ch=='y' || ch=='Y');

}
```

OUTPUT:

```
Enter First No:
Enter a coefficient value:3

Enter a power value:3

Want to continue? [Y/N]:
Enter a coefficient value:
2

Enter a power value:2

Want to continue? [Y/N]:
Enter a coefficient value:6

Enter a power value:1

Want to continue? [Y/N]:
Enter a coefficient value:7

Enter a power value:0


Want to continue? [Y/N]:
```

```
Enter second No:
Enter a coefficient value:2

Enter a power value:3

Want to continue? [Y/N]:
Enter a coefficient value:6

Enter a power value:2

Want to continue? [Y/N]:
Enter a coefficient value:7

Enter a power value:1

Want to continue? [Y/N]:
Enter a coefficient value:
4

Enter a power value:0

Want to continue? [Y/N]:n

First No is:          3x^3+2x^2+6x^1+7x^0
Second No is:         2x^3+6x^2+7x^1+4x^0
Addition of the polynomial is:    5x^3+8x^2+13x^1+11x^0
```