UNIT 14 TREES

Structure **Page No.**

14.1 INTRODUCTION

In the previous block we discussed Arrays, Lists, Stacks and Queues. In this we will discuss trees. The concept of trees is one of the most fundamental and useful concepts in computer science. Trees have many variations, implementations and applications. Trees find their use in applications such as compiler construction, database design, windows, operating system programs, etc. What is a tree? A tree structure is one in which items of data are related by edges. More formally, a Tree is a particular kind of graph, an **acyclic**, **connected** graph. A Tree contains no loops or cycles. If all these are Greek and Latin to you, do not worry. To make the discussion and definition of trees understandable, we will discuss graphs briefly in this Unit. You will study graphs in greater detail in MMTE-001, Graph theory course in the 3rd semester. In this Unit our attention will be restricted to rooted trees. In Sec. 14.2, we will introduce you to the basic terminology related to trees. In Sec. 14.3 we will discuss binary trees, a special type of trees. In Sec. 14.4, we will discuss how to traverse a binary tree. In Sec. 14.5, we will see how to search a binary tree.

Objectives

After studying this unit, you should be able to

- define a tree, a rooted tree, a binary tree, and a binary search tree
- differentiate between a general tree and a binary tree
- describe the properties of a binary search tree
- write programs for insertion, deletion and searching of an element in a binary search tree
- show how an arithmetic expression may be stored in a binary tree
- build and evaluate an expression tree
- write programs for preorder, in order, and post order traversal of a tree

14.2 BASIC TERMINOLOGY

Before we formally discuss trees formally, we will discuss some common examples of trees in an informal way. Trees are encountered frequently in everyday life. An example is found in the

organisational chart of a large corporation. Computer Science in particular makes extensive use of trees. For example, in databases it is useful in organising and relating data. It is also used for scanning, parsing, generation of code and evaluation of arithmetic expressions in compiler design.

A very common example is the ancestor tree as given in Fig. 1. This tree shows the ancestors of LAKSHMI. Her parents are VIDYA and RAMKRISHNA; RAMKRISHNA'S PARENTS are SUMATHI and VIJAYANANDAN who are also the grand parents of LAKSHMI (on father's side); VIDYA'S parents are JAYASHRI and RAMAN and so on.

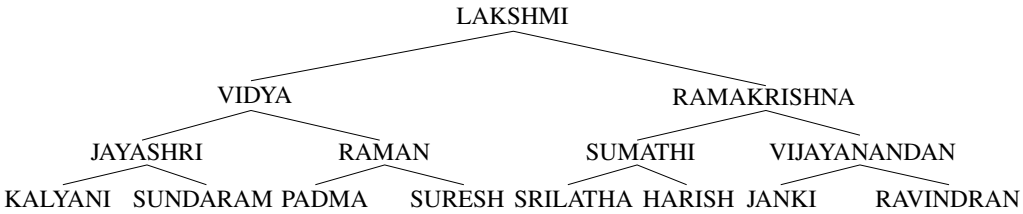


Fig. 1: A Family Tree I.

We can also have another form of ancestor tree as given in Fig. 2.

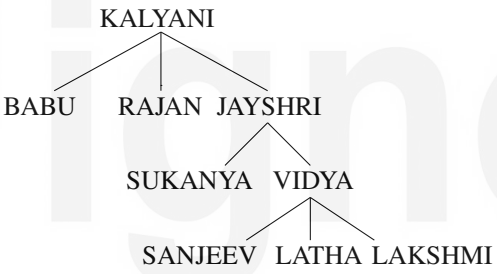


Fig. 2: A Family Tree II

We could have also generated the image of tree in Fig. 1 as in Fig. 3.

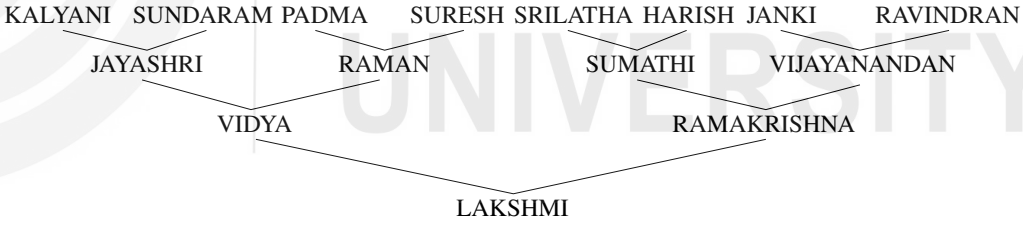


Fig. 3: A Family Tree III

All the above structures are called **rooted** trees. A tree is said to be **rooted** if it has one node, called the **root** that is distinguished from the other nodes. In Fig. 1, the root is LAKSHMI, in Fig. 2 the root is KALYANI and in Fig. 3 the root is LAKSHMI. We usually draw trees with the root at the top. Each node (except the root) has exactly one node above it, which is called its **parent**; the nodes directly below a node are called its children. We sometimes carry the analogy to family trees further and refer to the **grandparent** or the **sibling** of a **node**.

Let us now discuss some basic concepts in graph theory to prepare the ground for the study of trees. We start with a formal definition of a graph.

Definition 2: A (simple) **graph** G consists of a set V of vertices (or nodes) and a set E of edges (or arcs). We write G = (V, E) where V is a finite non-empty set of vertices. E is a subset of $V \times V$, the set of (unordered) pairs of elements in V.

Therefore, V(G), read as 'V of G' is the set of vertices and E(G), read as 'E of G' is the set of edges. An edge e = (v,w) is a pair of vertices v and w, and is said to be incident with v and w. A graph may be pictorially represented as in Fig. 4.

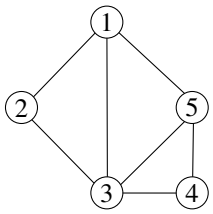


Fig. 4

We have numbered the nodes as 1, 2, 3, 4 and 5. So,

$$V(G) = \{1,2,3,4,5\}$$

and

$$E(G) = \{(1,2),(2,3),(3,4),(4,5),(1,5),(1,3),(3,5)\}$$

You may notice that we wrote the edge incident with node 1 and node 5 as $(1,5)$; we could have also written (5,1) instead. The same applies to all edges. Here, we do not attach ordering of the vertices. This is an **unordered graph** or a **simple graph**.

Definition 3: By a **subgraph** of a graph $(V(G),E(G))$, we mean a graph $(V(H),E(H))$, where $V(H) \subset V(G)$ and $E(H) \subset E(G)$.

For example, the graph in Fig. 5 is a subgraph of the graph in Fig. 4 with V(H) = 3,4,5} and E(H) = $\{(3,4),(3,5),(4,5)\}$

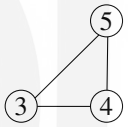


Fig. 5

We can also attach importance to the order of the vertices. We then get an **ordered graph**. In this, each vertex is represented by an ordered pair. So, we consider (1,5) and (5,1) as different edges. We can represent a directed graph pictorially as in Fig. 6.

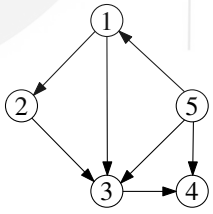


Fig. 6

We indicate the direction by an arrow. The set of vertices for this graph remains the same as that of the earlier example, i.e.

V(G) = 1,2,3,4,5}

However, the set of edges would be

$$E(G) = \{(1,2),(2,3),(3,4),(5,4),(5,1),(1,3),(5,3)\}$$

Did you notice the difference? Also, note that arrow is always from tail vertex to head vertex. In our further discussion on graphs, we will refer to directed graphs as digraphs and undirected graphs as simply graphs.

Definition 4: Two vertices, v and w, in a graph are **adjacent** if (v,w) is in E(G). In the case of digraphs, v and w are adjacent if either (v,w) or (w,v) is in E(G).

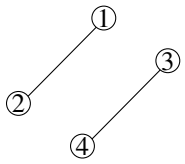


Fig. 7

In Fig. 7, vertices 1 and 2 are adjacent, but 1 and 4 are not adjacent.

Definition 5: A **path** from a vertex v to a vertex w is a sequence of vertices starting with v and ending in w with each vertex adjacent to the next. v is called the **starting** vertex and w is called the **end** vertex. We say that vertices in the sequences **lie** on the path or simply on the path joining the vertices v and w. A path in which the starting vertex and the end vertex are the same is called a **cycle.**

In Fig. 6, 1, 3, 4 is a path joining 1 and 4. In Fig. 4 on the preceding page, 5,4,3,2 is a path joining 5 and 2. In Fig. 4 on the facing page, 1,2,3,1 is a cycle.

Notice that, in Fig. 4, we can always find a path joining any two vertices. Such graphs are called connected graphs. If there are two vertices in a graph which are not connected by path, we say that the graph is disconnected. For example the graph in Fig. 7 is disconnected.

Notice that, although the graph in Fig. 7 looks like two different graphs, it is a single graph with V(G) = 1,2,3,4} and E(G) = (1,2), (3,4)}. In this graph there is no path connecting 1 and 4. Also, you can see that the graph has two 'pieces'. They are called connected components of the graph. A connected component of a graph is a maximal connected subgraph of a graph. In other words, a connected component must be a connected graph and it should not be a proper subgraph of any other connected subgraph of G. For example, $V(G_1)$ = 1,2}, $E(G_1) = \{(1,2)\}$ is a connected component of G.

Definition 6: The **degree** of a vertex of v a graph is the number of edges incident on the vertex v. In the case of a digraph, the **in degree** of a vertex v is the number of edges that end in v and the **out degree** is the number of vertices that start in v.

For example, in Fig. 4, the degree of 1 is three and the degree of 3 is four.

Definition 7: A graph is called a **tree** if it is connected and does contain any cycle.

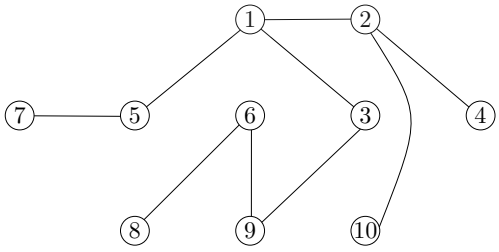


Fig. 8

Fig. 8 is an example of a tree.

In this Unit, we will discuss only a special class of trees called **rooted trees**. In what follows, we will also use the term **node** for the vertex of a tree and **branch** for the edge of a tree.

In Fig. 2 root is KALYANI. The three sub-trees are rooted at BABU, RAJAN and JAYASHRI. Sub-trees with JAYASHRI as root has two sub-trees rooted at SUKANYA and VIDYA and so on. 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, i.e. it may be of degree 1. A node that has no children is called a **leaf** node.

If a tree has n nodes, one of which is the root then there would be n – 1 branches. It follows from the fact that each branch connects some node to its parent, and every node except the root has one parent. Nodes with the same parent are called **siblings**. Consider the tree given in Fig. 9.

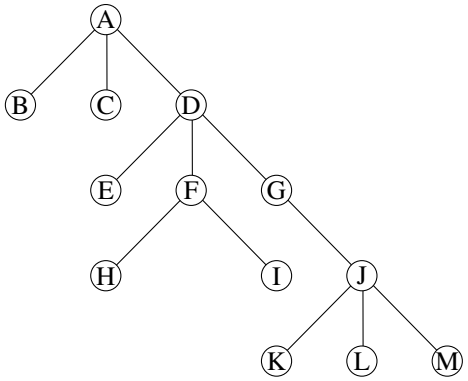


Fig. 9

K, L, and M are all siblings. B, C, D are also siblings.

There is exactly one path between any two nodes and between the root and each of the other nodes in the tree in particular. If there is more than one path between 2 nodes in a graph, the graph will contain a cycle; the graph will not be acyclic. If there is no path between two nodes, the graph will not be connected. In either case the graph will not be a tree. Nodes with no children are called **leaves**, or **terminal** nodes. Nodes with at least one child are sometimes called **non-terminal** nodes. We sometime refer to non-terminal nodes as **internal** nodes and terminal nodes as **external** nodes.

The length of a path is the number of branches (edges) on the path. Further if n lies on the unique path from the root to i, the n is an **ancestor** of i and i is a **descendant** of n. Also there is a path of length zero from every node to itself, and there is exactly one path from the root to each node.

Let us now see how these terms apply to the tree given in Fig. 9. A path from A to K is A-D-G-J-K and the length of this path is 4.

A is ancestor of K and K is descendant of A. All the other nodes on the path are also descendants of A and ancestors of K.

The **depth** of any node n_i is the length of the path from the root to n_i. Thus, the root is at depth 0(zero). The **height** of a node n_i is the longest path from n_i to a leaf. Thus all leaves are at height zero. Further, the height of a tree is same as the height of the root. For the tree in Fig. 9, F is at height 1 and depth 2. D is at height 3 and depth 1. The height of the tree is 4. Depth of a node is sometimes also referred to as **level** of a node.

An acyclic graph which is not connected is called a forest. Each component of such a graph will be a tree; for example, if we remove the root and the edges connecting it from the tree in Fig. 9, we are left with a forest consisting of three trees rooted at A, D and G, as shown in Fig. 10. Let us now list some of the properties of a tree:

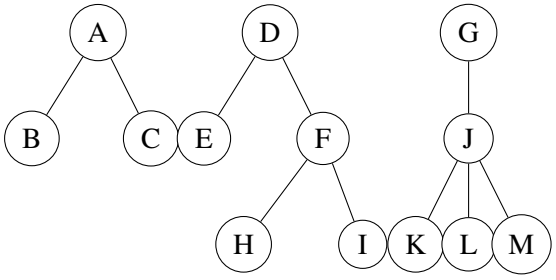


Fig. 10: A Forest (sub-tree)

Properties of a tree

1. Any node can be root of the tree 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. Each node, except the root, has a unique parent and every edge connects a node to its parents .Therefore, a tree with N nodes has N-1 edges.

We close the discussion of general trees here. In the next section, we will discuss binary trees.

14.3 BINARY TREES

By definition, a **Binary tree** is a tree which is either empty or consists of a root node and two disjoint binary trees called the left subtree and right subtree. In Fig. 11, a binary tree T is depicted with a left subtree, L(T) and a right subtree R(T).

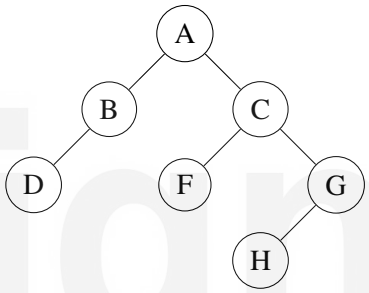


Fig. 11: A Binary Tree

a binary tree, no node can have more than two children. So, every node in a binary tree has 0, 1 or no children. Binary trees are special cases of general trees. The terminology we have discussed in the previous section applies to binary trees also.

Let us list the properties of binary trees:

1. Recall from the previous section the definition of internal and external nodes.- A binary tree with N internal nodes has maximum of (N + 1) **external nodes** : Root is considered as an internal node.
2. The external path length of any binary tree with N internal nodes is 2N greater than the internal path length.
3. The height of a full binary tree with N internal nodes is about $\log_2 N$

As we shall see, binary trees appear extensively in computer applications, and performance is best when the binary trees are full (or nearly full). You should note carefully that, while every binary tree is a tree, not every tree is a binary tree.

A **full binary tree** or a **complete binary tree** is a binary tree in which all internal nodes have degree and all leaves are at the same level. The Fig. 11 illustrates a full binary tree.

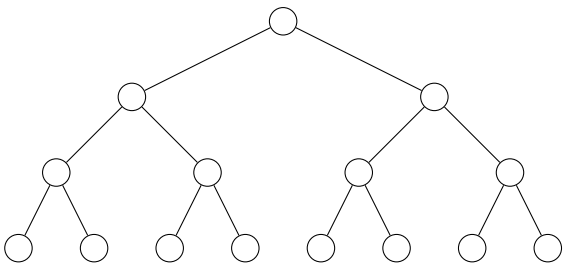


Fig. 12: A full binary tree.

The **degree** of a node is the number of non empty sub trees it has. A leaf node has a degree zero.

Implementation

A binary tree can be implemented as an array of nodes or a linked list. The most common and easiest way to implement a tree is to represent a node as a struct consisting of the data and pointer to each child of the node. Because a binary tree has at most two children, we can keep direct pointers to them. A binary tree node declaration in change may look like.

```
struct tnode {
Struct tnode *left;
Int data;
Struct tnode *right;};
typedef struct tnode Tnode;
typedef TNode *TNodePtr;
```

Let us now consider a special case of binary tree. it is called a 2-tree or a strictly binary tree. It is a non-empty binary tree in which either both sub trees are empty or both sub trees are 2-trees. For example, the binary trees in Fig. 13a and Fig. 13b are 2-trees, but the trees in Fig. 13c and 13d are not 2- trees.

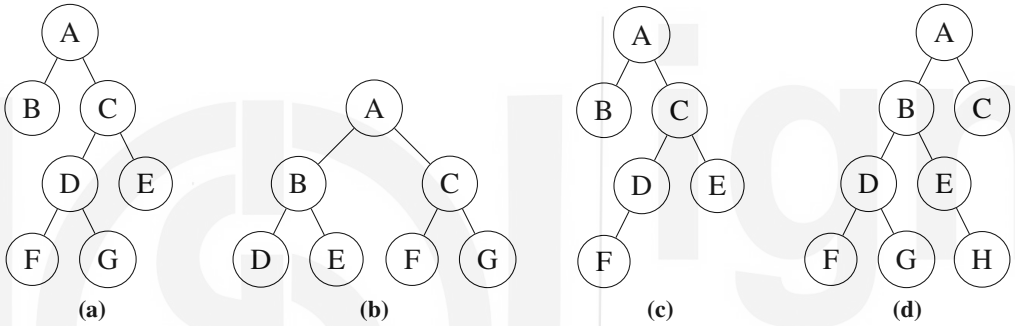


Fig. 13

Binary trees are most commonly represented by linked lists. 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.

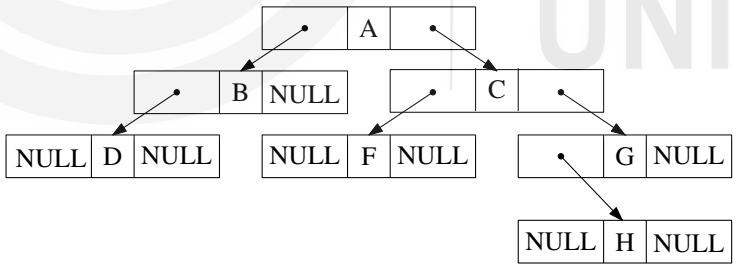


Fig. 14: Linked list representation of a Binary Tree

Fig. 14 contains the linked storage representation of a binary tree Fig. 11. A binary tree is said to be complete (See Fig. 11) if it contains the maximum number of nodes possible for its height. In a complete binary tree:

1. The number of nodes at level 0 is 1.
2. The number of nodes at level 1 is 2.
3. The number of nodes at level 2 is 4, and so on.
4. The number of nodes at level i is 2^i. Therefore for a complete binary tree with k- levels contains $\sum_{i=0}^{k} 2^i$ nodes.

E6) How many different trees are there with three nodes? Draw each.

E7) Give level, degree and height of each node of the tree in Fig. 15.

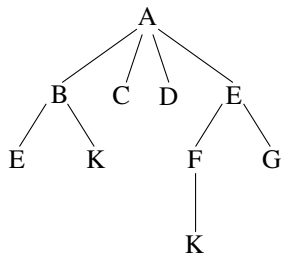


Fig. 15

We conclude this section here. In the next section, we will take up tree traversal.

14.4 TRAVERSALS OF A BINARY TREE

By a **traversal** of a graph is to visit each node exactly once. In this section we shall discuss traversal of a binary tree. It is useful in many applications. For example, in searching for particular 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 different ways to proceed. The methods differ primarily in the order in which they visit the nodes. The four different traversals of T are In order, Post order, Preorder and Level-by-level traversal.

14.4.1 In order Traversal

It follows the general strategy of Left-Root-Right. In this traversal, if T is not empty, we first traverse

1. the left sub tree and recursively traverse in order this tree;
2. then visit the root node of T; and
3. then traverse (in order) the right sub tree.

Consider the binary tree given in Fig. 11. Let us see how this is traversed in in order traversal.

1. The root node is A. In in order traversal, we first go to node B, the node to the left of A.
2. Recursively, B is now the root node, so we go to D. The node traversed so far are B, D.
3. D is a leaf node. So we have to go to the right node of B. B does not have a right node. We have exhausted all the descendents of B and so we go to the root of B which is A. The nodes traversed are D, B, A.
4. Now, we start traversing from C. C is the root node which is a left node. We go to its left node E. The nodes traversed are D, B, A, E.
5. E is a leaf node. So, we go to root node which is C. The nodes traversed so far are D, B, A, E, C.
6. Next, we go G. G is now the root node with a left descendent H. We go there.D, B, A, E, C, H.
7. G does not have a descendent on the right. So, we go to G. The nodes traversed are D, B, A, E, C, H, G.
8. From G, we move up to its root node C. We have exhausted all descendents of C. We go to the root of C, which is A. So, the final list of nodes in the order in which they are traversed is D, B, A, E, C, H, G.

Here is an exercise for you.

E8) Give the order of vertices in which the vertices are traversed in a inorder traversal of the graph in Fig. 13a.

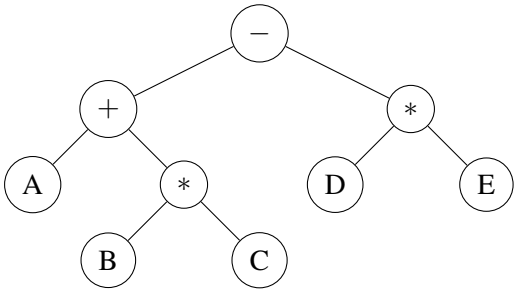


Fig. 16: Expression Tree

Fig. 16 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:

1. each operator node has exactly two branches
2. each operand node has no branches, such trees are called **expression trees**.

Let us traverse this tree in in order traversal.

1. Tree, T, at the start is rooted at '−';
2. Since left(T) is not empty; current T becomes rooted at +;
3. Since left(T) is not empty; current T becomes rooted at 'A'.
4. Since left(T) is empty; we visit root i.e. A.
5. We access T' root i.e. '+'.
6. We now perform in order traversal of right(T).
7. Current T becomes rooted at '*'.
8. Since left(T) is not empty; Current T becomes rooted at 'B' since left(T) is empty; we visit its root i.e. B; check for right(T) which is empty, therefore, we move back to parent tree. We visit its root i.e. '*'.
9. Now in order 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

$$A + B * C - D * E$$

You may note that expression is in infix notation. The in order traversal produces a (parenthesized) left expression, then prints out the operator at root and then a (parenthesized) right expression. This method of traversal is probably the most widely used. The following is a C function for in order traversal of a binary tree

```
void inorder (TNorePtr tptr)
        if (tptr ! = NULL){
             inorder (tptr->left);
             printf(''\%d'', tptr->data);
             inorder (tptr->right);
        }
```

Please notice that this procedure, like the definition for traversal is recursive.

14.4.2 Post order Traversal

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

Traverse the left sub tree In Post order.

Traverse the right sub tree in Post order.

Visit the root.

For example, a post order traversal of the tree given in Fig. 16 would be

$$ABC*+DE*-$$

You may notice that it is the postfix notation of the expression

$$(A+(B*C))-(D*E)$$

We leave the details of the post order traversal method as an exercise. Here is an exercise for you.

E9) Write a C function for post order traversal.

14.4.3 Preorder Traversal

In this traversal, we visit root first; then recursively perform preorder traversal of Left(T); followed by pre order. traversal of Right(T) i.e. a Root-Left-Right traversal, i.e.

Visit the root

Traverse the left sub tree preorder.

Traverse the right sub tree preorder.

A preorder traversal of the tree given in Fig. 16 would yield

$$-+A*BC*DE$$

It is the prefix notation of the expression

$$(A+(B*C))-(D*E)$$

Here is an exercise for you.

E10) Write a C function for preorder traversal.

14.4.4 Level by Level traversal

In this method we traverse level-wise i.e. we first visit node root at level 'O' i.e. root. There would be just one. Then we visit nodes at level one from left to right. There would be at most two. Then we visit all the nodes at level '2' from left to right and so on. For example the level by level traversal of the binary tree given in Fig. 11 will yield

A B C D E F G H I J K

This traversal is different from other three traversals in the sense that it need not be recursive, therefore, we may use queue kind of a data structure to implement it, while we need stack kind of data structure for the earlier three traversals.

E11) Traverse the tree given in Fig. 17 in preorder, in order, post order and level by level giving a list of nodes visited.

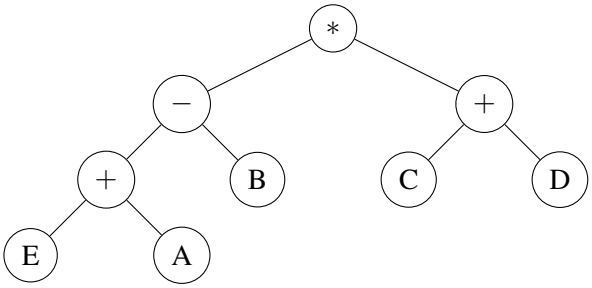


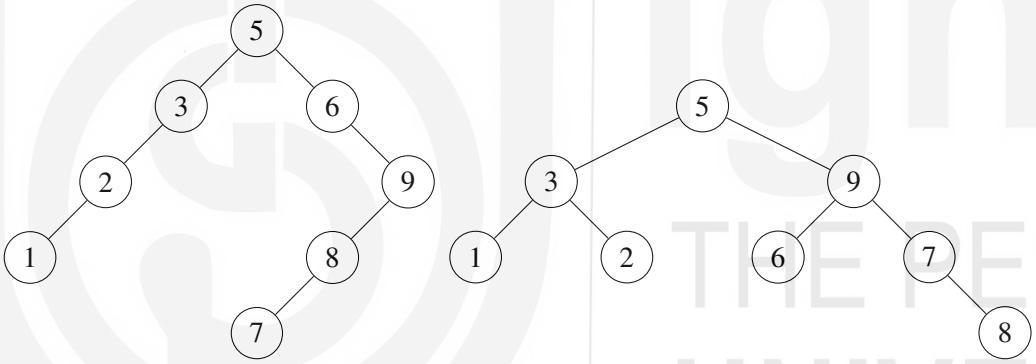
Fig. 17

We close this section here. In the next section, we discuss binary search trees.

14.5 BINARY SEARCH TREES (BST)

A **Binary Search Tree, BST**, is an ordered binary tree T such that either it is an empty tree or

1. each data value in its left sub tree is less than the root value,
2. each data value. in its right sub tree is greater than the root value, and
3. left and right sub trees are again binary search trees.



(a) Binary Search Tree **(b) Binary tree but not binary search tree**

Fig. 18

Fig. 18b depicts a binary search tree, while the one in Fig. 18b is not a binary search tree. (Why ?) Clearly, duplicate items are not allowed in a binary search tree. You may also notice that an in order traversal of a BST yields a sorted list in ascending order.

14.5.1 Operations on a BST

We now give a list of the operations that are usually performed on a BST.

1. Initialization of a **BST:** This operation makes an empty tree.
2. Cheek whether **BST** is Empty: This operation cheeks whether the tree is empty.
3. Create a node for the **BST:** This operation allocates memory space for the new node; returns with error if no spade is available.
4. Retrieve a node's data.
5. Update a node's data.
6. Insert a node in **BST**.
7. Delete a node (or sub tree) of a **BST**.

8. Search for a node in **BST**.
9. Traverse (in inorder, preorder, or post order) a **BST**.

We shall describe some of the operations in detail.

14.5.2 Insertion in a BST

Inserting a node to the tree: To insert a node in a BST, we must check whether the tree already contains any nodes. If tree is empty, the node is placed in the root node. If the tree is not empty, then the proper location is found and the added node becomes either a left or a right child of an existing node. The logic works this way:

```
add-node(node, value)
{
        if (two values are  same)
                duplicate;
                return (FAILURE)
        }
        else if (value < value in current node)
                if (left child exists)
                        add-node (left child, value);
                else{
                        allocate new node and make left
                        child point to it;
                        return (SUCCESS);
                }
        }
        else if (value > value in current node)
                if (right child exists)
                        add-node (right child, value);
                else{
                        allocate new node and make right
                        child point to it;
                        return (SUCCESS);
                }
        }
}
```

The function continues recursively until either it finds a duplicate (no duplicate strings are allowed) or it hits a dead end. If it determines that the value to be added belongs to the left-child sub tree and there is no left-child node, it creates one. If a left-child node exists, then it begins its search with the sub tree beginning at this node. If the function determines that the value to be added belongs to the right of the current node, a similar process occurs.

Let us consider the BST given in Fig. 19a.

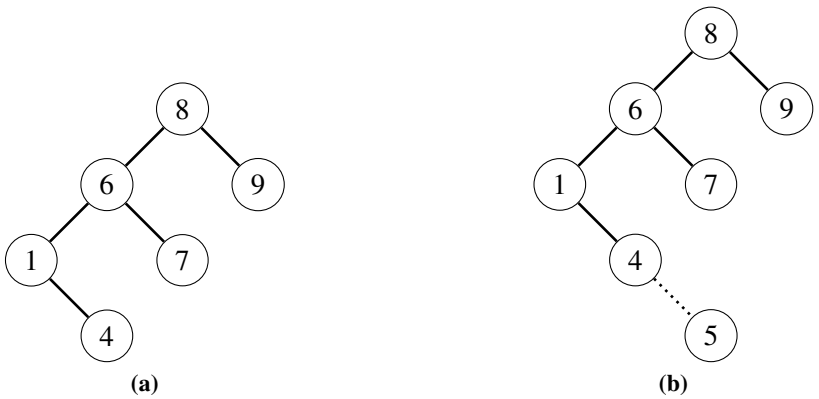


Fig. 19: Insertion in a Binary Search Tree

If we want to insert 5 in the BST in Fig. 19a, we first search the tree. If the key to be inserted is found in tree, we do nothing (since duplicates are not allowed), otherwise a nil is returned. In case a nil is returned, we insert the data at the last point traversed. In the example above a search operation will return nil on not finding a right, sub tree of tree rooted at 4. Therefore, 5 must be inserted as a right child of 4.

14.4.3 Deletion of a node

Once again the node to be deleted is searched in BST. If found, we need to consider the following possibilities:

(i) If node is a leaf, it can be deleted by making its parent pointing to nil. The deleted node is now unreferenced and may be disposed off. For example, we if we delete the node 4 in Fig. 19a the resulting BST will be the one in Fig. Fig. 20.

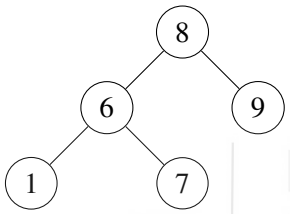


Fig. 20: Deletion of a Terminal Node

(ii) If the node has one child, its parent's pointer needs to be adjusted. For example for node 1 to be deleted from BST given in Fig. 19a the left pointer of node 6 is made to point to child of node 1 i.e. node 4 and the new structure would be as in Fig. Fig. 21.

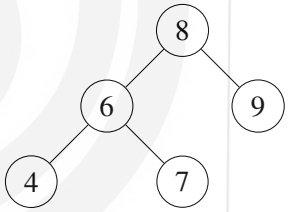
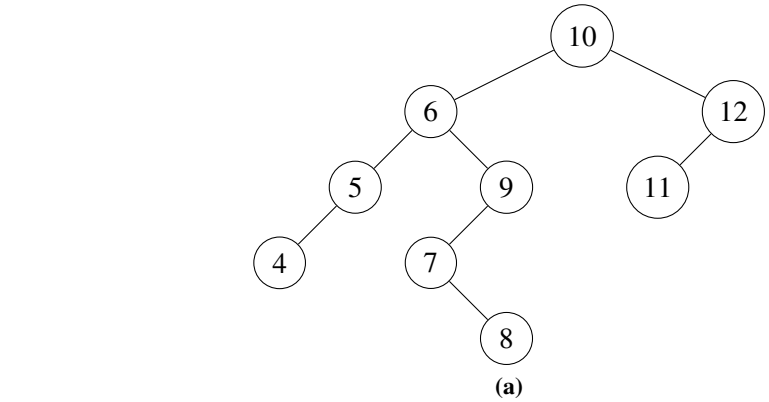


Fig. 21: Deletion of a Node with one child

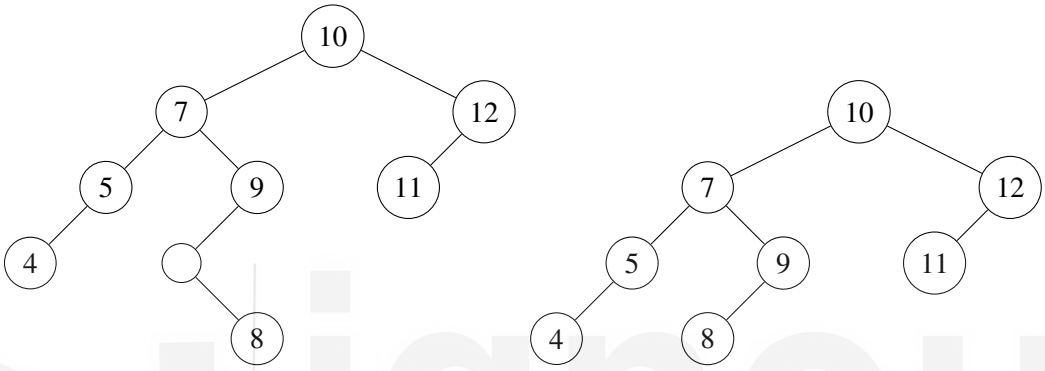
(iii) If the node to be deleted has two children; then the value is replaced by the smallest value in the right sub tree or the largest key value in the left sub tree; subsequently the empty node is recursively deleted. Consider the BST in Fig. 22a on the following page. If the node 6 is to be deleted then first its value is replaced by smallest value in its right subtree i.e. by 7. After we do this, the tree will be as in Fig. 22b on the next page. Now we need to, delete this empty node as explained in (iii). Therefore, the final structure would be as in Fig. 22c.

14.5.3 Search for a key in a BST

To search the binary tree for a particular node, we use procedures similar to those we used when adding to it. Beginning at the root node, the current node and the entered key are compared. If the values are equal success is output. If the entered value is less than the value in the node, then it must be in the left-child sub tree. If there is no left-child sub tree, the value is not in the tree i.e. a failure is reported. If there is a left-child subtree, then it is examined the same way. Similarly, if the entered value is greater than the value in the current node, the right child is searched. Fig. 23 on the following page shows the path through the tree followed in the search for the key H.



(a)



(b) (c)

Fig. 22: Removing a key with two children in a BST.

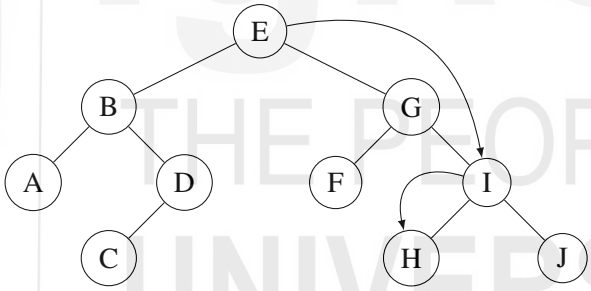


Fig. 23: Searching for a key in a BST.

```
find-key (key value, node){
     if (two values are same){
     print value stored in node;
          return (SUCCESS);
     }
     else if (key value < value stored in current node){
          if (left child exists)
          {
               find-key (key-value, left child);
          }
          else
          {
               there is no left subtree.,
               return (string not found)
          }
     }
          else if (key-value > value stored in current node){
               if (right child exists)
               {
                    find-key (key-value, right child);
               }
               else{
```

```
                there is no right subtree;
                return (string not found);
        }
```

14.6 SUMMARY

This unit introduced the tree data structure which is an **acyclic, connected, simple** graph. Terminology pertaining to trees was introduced. A special case of general case of general tree, a binary tree was focussed on. In a binary tree, each node has a maximum of two subtrees, left and right subtree. Sometimes it is necessary to traverse a tree, that is, to visit all the tree's nodes. Four methods of tree traversals were presented in order, post order, preorder and level by level traversal. These methods differ in the order in which the root, left subtree and right subtree are traversed. Each ordering is appropriate for a different **type of applications.**

An important class of binary trees is a complete or full binary tree. A full binary tree is one in which internal nodes completely fill every level, except possibly the. last. A complete binary tree where the internal nodes on the bottom level all appear to the left of the external nodes on that level. Fig. 6a shows an example of a complete binary tree. We conclude this section here. In the next section, we will summarise this unit.

14.7 SOLUTIONS/ANSWERS

E1) The Total number of different trees with 3 nodes are 5. See Fig. 24.

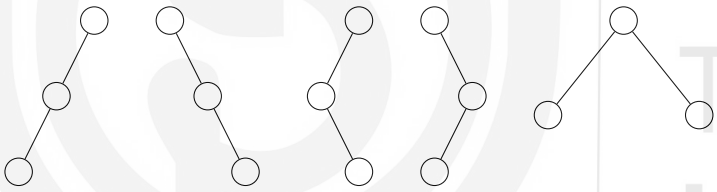


Fig. 24

E2)

Node	Level	Degree	Height
A	0	4	0
B	1	2	1
C	1	0	1
D	1	0	1
E	2	0	1
K	2	0	1
l	1	2	1
f	2	1	1
g	2	0	1
h	3	0	1

E3) B, A, F, D, G, C, E

E4)
```
Postorder(TnodePtr tptr)
{
     If (tptr ! = NULL) {
          Postorder (tptr-> left);
          Postorder (tptr-> right);
          printf (‘‘\%d’’, tptr-> data);
     }
}
```

E5)
```
Preorder (TnodePtr tptr)
{
     if (tptr ! = NULL) {
          printf (‘‘\%d’’, data);
          Preorder (tptr-$\mathrm{>}$left);
          Preorder (tptr-$\mathrm{>}$right);
     }
}
```

E6) Preorder: *-+EAB+CD

Inorder: E+A-B*C+D

Postorder: EA+B-CD + *-

Level by level: *- + +BCDEA