UNIT 6 TREES

Structure **Page Nos.**

6.0 Introduction 31
6.1 Objectives 31
6.2 Abstract Data Type-Tree 31
6.3 Implementation of Tree 34
6.4 Tree Traversals 35
6.5 Binary Trees 37
6.6 Implementation of a Binary Tree 38
6.7 Binary Tree Traversals 40
6.7.1 Recursive Implementation of Binary Tree Traversals
6.7.2 Non-Recursive Implementation of Binary Tree Traversals
6.8 Applications 43
6.9 Summary 45
6.10 Solutions/Answers 45
6.11 Further Readings 46

6.0 INTRODUCTION

Have you ever thought how does the operating system manage our files? Why do we have a hierarchical file system? How do files get saved and deleted under hierarchical directories? Well, we have answers to all these questions in this section through a hierarchical data structure called Trees! Although most general form of a tree can be defined as an **acyclic graph**, we will consider in this section only rooted tree as general tree does not have a parent-child relationship.

Tree is a data structure which allows you to associate a parent-child relationship between various pieces of data and thus allows us to arrange our records, data and files in a hierarchical fashion. Consider a Tree representing your family structure. Let us say that we start with your grand parent; then come to your parent and finally, you and your brothers and sisters. In this unit, we will go through the basic tree structures first (general trees), and then go into the specific and more popular tree called binary-trees.

6.1 OBJECTIVES

After going through this unit, you should be able

- to define a tree as abstract data type (ADT);
- learn the different properties of a Tree and a Binary tree;
- to implement the Tree and Binary tree, and
- give some applications of Tree.

6.2 ABSTRACT DATA TYPE-TREE

Definition: A set of data values and associated operations that are precisely specified independent of any particular implementation.

Since the data values and operations are defined with mathematical precision, rather than as an implementation in a computer language, we may reason about effects of the operations, relationship to other abstract data types, whether a programming language implements the particular data type, etc.

Consider the following abstract data type:

```
Structure  Tree

   type Tree = nil | fork (Element , Tree , Tree)

Operations:

   null : Tree -> Boolean
   leaf : Tree -> Boolean
   fork : (Element , Tree , Tree) -> Tree
   left : Tree -> Tree                 // It depicts the properties of tree that left of a
   tree is also a tree.
   right: Tree -> Tree
   contents: Tree -> Element
   height (nil) = 0 |
   height (fork(e,T,T')) = 1+max(height(T), height(T'))
   weight (nil) = 0 |
   weight (fork(e,T,T')) = 1+weight(T)+weight(T')
```

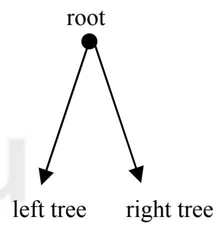


Figure 6.1: A binary tree

```
Rules:

   null(nil) = true                  // nil is an empty tree

   null(fork(e, T, T'))= false       //  e : element , T and T are two sub tree

   leaf(fork(e, nil, nil)) = true
   leaf(fork(e, T, T')) = false if not null(T) or not null(T')
   leaf(nil) =  error

   left(fork(e, T, T')) = T
   left(nil) = error

   right(fork(e, T, T')) = T'
   right(nil) = error

   contents(fork(e, T, T')) = e
   contents(nil) = error
```

Look at the definition of Tree (ADT). A way to think of a *binary tree* is that it is either empty (nil) or contains an element and two sub trees which are themselves binary trees (Refer to *Figure 6.1*). Fork operation joins two sub tree with a parent node and produces another Binary tree. It may be noted that a tree consisting of a single leaf is defined to be of height 1.

Definition : A tree is a connected, acyclic graph (Refer to *Figure 6.2*).

It is so connected that any node in the graph can be reached from any other node by exactly one path.

It does not contain any cycles (circuits, or closed paths), which would imply the existence of more than one path between two nodes. This is the most general kind of tree, and may be converted into the more familiar form by designating a node as the root. We can represent a tree as a construction consisting of nodes, and edges which represent a relationship between two nodes. In *Figure 6.3*, we will consider most common tree called **rooted tree**. A rooted tress has a single root node which has no parents.

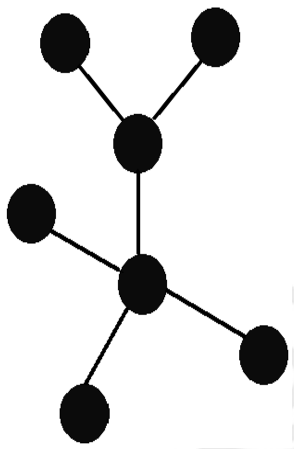


Figure 6.2 : Tree as a connected acyclic graph

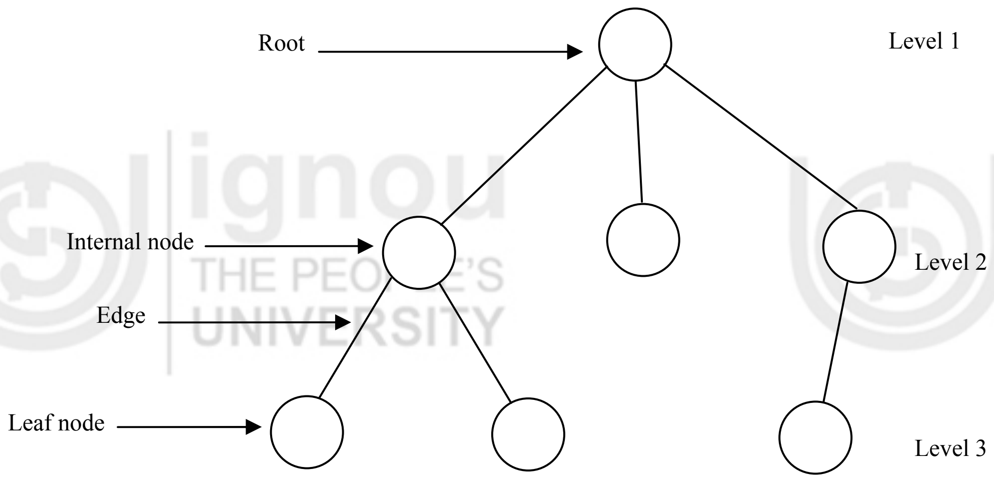


Figure 6.3 : A rooted tree

In a more formal way, we can define a tree T as a finite set of one or more nodes such that there is one designated node r called the root of T, and the remaining nodes in (T – { r }) are partitioned into n > 0 disjoint subsets T_1, T_2, ..., T_k each of which is a tree, and whose roots r1 , r2 , ..., rk , respectively, are children of r. The general tree is a generic tree that has one root node, and every node in the tree can have an unlimited number of child nodes. One popular use of this kind of tree is a Family Tree.
A tree is an instance of a more general category called graph.

- A tree consists of nodes connected by edges.
- A root is a node without parent.
- Leaves are nodes with no children.
- The root is at level 1. The child nodes of root are at level 2. The child nodes of nodes at level 2 are at level 3 and so on.
- The depth (height) of a Binary tree is equal to the number of levels in it.
- Branching factor defines the maximum number of children to any node. So, a branching factor of 2 means a binary tree.

- Breadth defines the number of nodes at a level.
- The depth of a node M in a tree is the length of the path from the root of the tree to M.
- A node in a Binary tree has at most 2 children.

The following are the properties of a Tree.

Full Tree : A tree with all the leaves at the same level, and all the non-leaves having the same degree

- Level h of a full tree has d^{h-1} nodes.
- The first h levels of a full tree have $1 + d + d^2 + d^3 + d^4 + \ldots\ldots + d^{h-1} = (d^h - 1)/(d - 1)$ nodes where d is the degree of nodes.
- The number of edges = the number of nodes – 1 (Why? Because, an edge represents the relationship between a child and a parent, and every node has a parent except the root.
- A tree of height *h* and degree *d* has at most $d^h - 1$ elements.

Complete Trees

A complete tree is a k-ary position tree in which all levels are filled from left to right. There are a number of specialized trees.

They are binary trees, binary search trees, AVL-trees, red-black trees, 2-3 trees.

Data structure- Tree

Tree is a dynamic data structures. Trees can expand and contract as the program executes and are implemented through pointers. A tree deallocates memory when an element is deleted.

Non-linear data structures: Linear data structures have properties of ordering relationship (can the elements/nodes of tree be sorted?). There is no first node or last node. There is no ordering relationship among elements of tree.

Items of a tree can be partially ordered into a hierarchy via parent-child relationship. Root node is at the top of the hierarchy and leafs are at the bottom layer of the hierarchy. Hence, trees can be termed as hierarchical data structures.

6.3 IMPLEMENTATION OF TREE

The most common way to add nodes to a general tree is to first find the desired parent of the node you want to insert, then add the node to the parent's child list. The most common implementations insert the nodes one at a time, but since each node can be considered a tree on its own, other implementations build up an entire sub-tree before adding it to a larger tree. As the nodes are added and deleted dynamically from a tree, tree are often implemented by link lists. However, it is simpler to write algorithms for a data representation where the numbers of nodes are fixed. *Figure* 6.4 depicts the structure of the node of a general k-ary tree.

Data	link1	link2	- · - · -	link k

Figure 6.4 : Node structure of a general k-ary tree

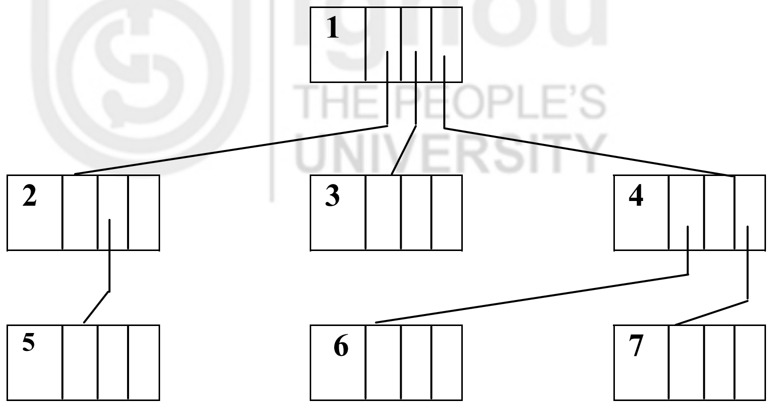


Figure 6.5: A linked list representation of tree (3-ary tree)

Figure 6.5 depicts a tree with one data element and three pointers. The number of pointers required to implement a general tree depend of the maximum degree of nodes in the tree.

6.4 TREE TRAVERSALS

There are three types of tree traversals, namely, Preorder, Postorder and Inorder.

Preorder traversal: Each node is visited before its children are visited; the root is visited first.

Algorithm for pre order traversal:

1. visit root node
2. traverse left sub-tree in preorder
3. traverse right sub-tree in preorder

Example of pre order traversal: Reading of a book, as we do not read next chapter unless we complete all sections of previous chapter and all it's sections (refer to *Figure 6.6*).

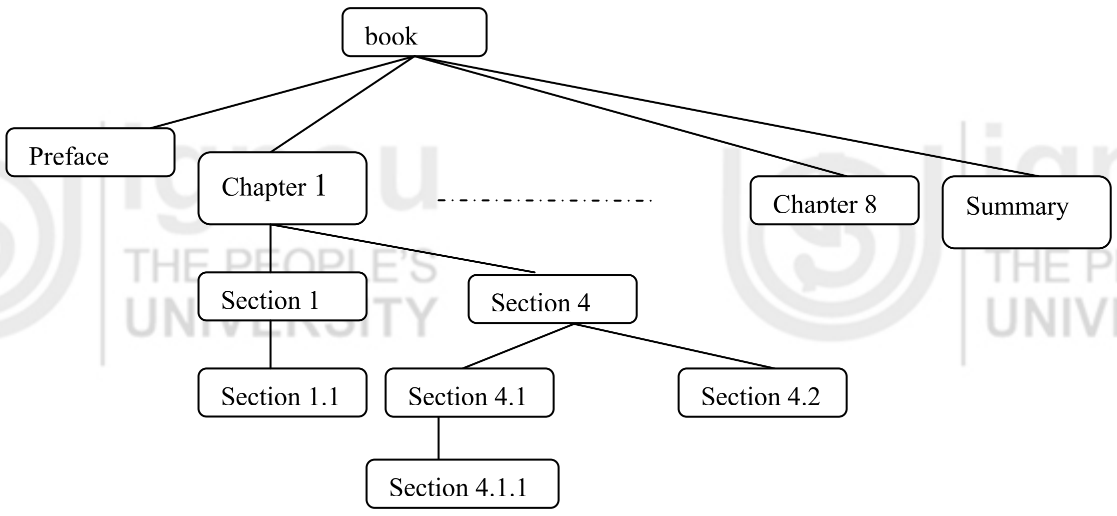


Figure 6.6 : Reading a book : A preorder tree traversal

As each node is traversed only once, the time complexity of preorder traversal is T(n) = O(n), where n is number of nodes in the tree.

Postorder traversal: The children of a node are visited before the node itself; the root is visited last. Every node is visited after its descendents are visited.

Algorithm for postorder traversal:

1. traverse left sub-tree in post order
2. traverse right sub-tree in post order
3. visit root node.

Finding the space occupied by files and directories in a file system requires a postorder traversal as the space occupied by directory requires calculation of space required by all files in the directory (children in tree structure) (refer to *Figure 6.7*)

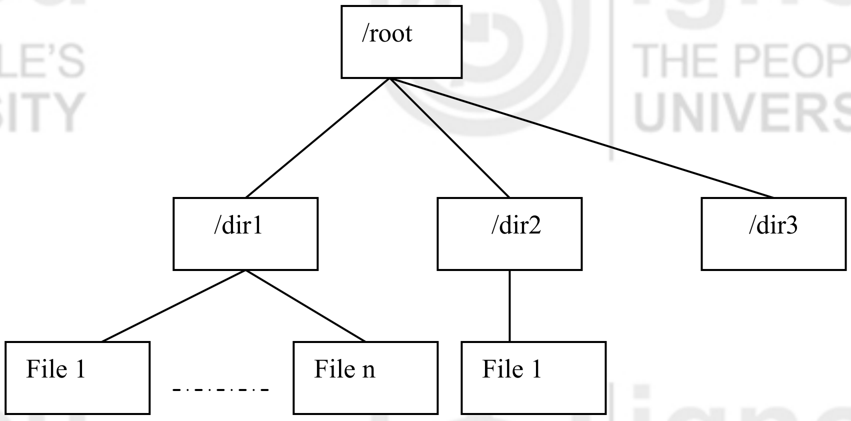


Figure 6.7 : Calculation of space occupied by a file system : A post order traversal

As each node is traversed only once, the time complexity of post order traversal is T(n) = O(n), where n is number of nodes in the tree.

Inorder traversal: The left sub tree is visited, then the node and then right sub-tree.

Algorithm for inorder traversal:
1. traverse left sub-tree
2. visit node
3. traverse right sub-tree

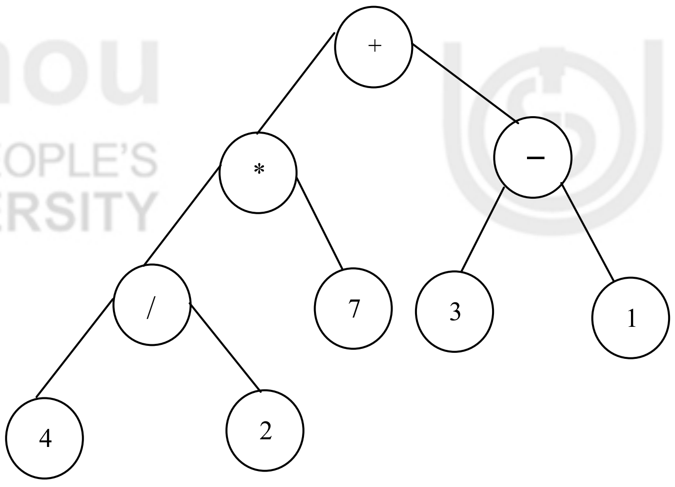


Figure 6.8 : An expression tree : An inorder traversal

Inorder traversal can be best described by an expression tree, where the operators are at parent node and operands are at leaf nodes.

Let us consider the above expression tree (refer to *Figure 6.8*). The preorder, postorder and inorder traversal are given below:

preorder Traversal : + * / 4 2 7 - 3 1
postorder traversal : 4 2 / 7 * 3 1 - +
inorder traversal : - ((((4 / 2) * 7) + (3 - 1))

There is another tree traversal (of course, not very common) is called level order, where all the nodes of the same level are travelled first starting from the root (refer to *Figure 6.9*).

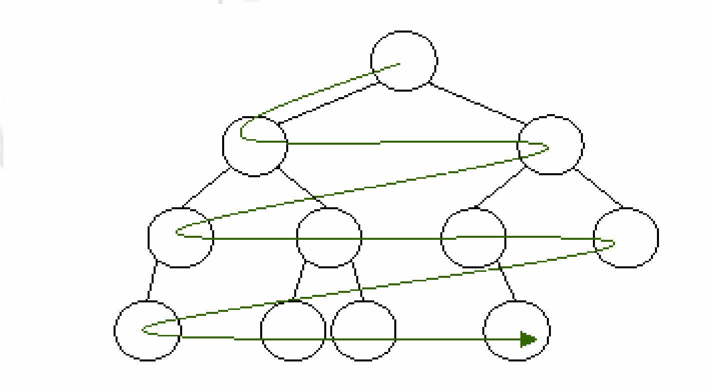


Figure 6.9: Tree Traversal: Level Order

☞ Check Your Progress 1

1) If a tree has 45 edges, how many vertices does it have?
2) Suppose a full 4-ary tree has 100 leaves. How many internal vertices does it have?
3) Suppose a full 3-ary tree has 100 internal vertices. How many leaves does it have?
4) Prove that if T is a full m-ary tree with v vertices, then T has ((m-1)v+1)/m leaves.

6.5 BINARY TREES

A binary tree is a special tree where each non-leaf node can have atmost two child nodes. Most important types of trees which are used to model yes/no, on/off, higher/lower, i.e., binary decisions are binary trees.

Recursive Definition: A binary tree is either empty or a node that has left and right sub-trees that are binary trees. Empty trees are represented as boxes (but we will almost always omit the boxes).

In a formal way, we can define a binary tree as a finite set of nodes which is either empty or partitioned in to sets of T_0, T_l, T_r, where T_0 is the root and T_l and Tr are left and right binary trees, respectively.

Properties of a binary tree

- If a binary tree contains ***n*** nodes, then it contains exactly $n - 1$ edges;
- A Binary tree of height h has $2^h - 1$nodes or less.
- If we have a binary tree containing n nodes, then the height of the tree is at most n and at least ceiling $\log_2(n + 1)$.
- If a binary tree has n nodes at a level l then, it has at most 2n nodes at a level l+1
- The total number of nodes in a binary tree with depth d (root has depth zero) is

$$N = 2^0 + 2^1 + 2^2 + \ldots\ldots + 2^d = 2^{d+1} - 1$$

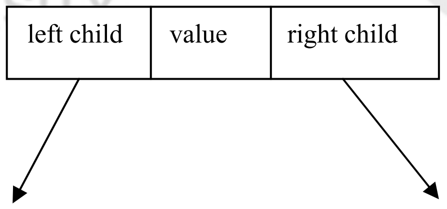
Full Binary Trees: A binary tree of height h which had $2^h - 1$ elements is called a Full Binary Tree.

Complete Binary Trees: A binary tree whereby if the height is d, and all levels, except possibly level d, are completely full. If the bottom level is incomplete, then it has all nodes to the left side. That is the tree has been filled in the level order from left to right.

6.6 IMPLEMENTATION OF A BINARY TREE

Like general tree, binary trees are implemented through linked lists. A typical node in a Binary tree has a structure as follows (refer to *Figure 6.10*):

```
struct NODE
 {
  struct NODE *leftchild;
  int nodevalue;              /* this can be of any data type */
  struct NODE *rightchild;
 };
```



The 'left child'and 'right child' are pointers to another tree-node. The "leaf node" (not shown) here will have NULL values for these pointers.

Figure 6.10 : Node structure of a binary tree

The binary tree creation follows a very simple principle. For the new element to be added, compare it with the current element in the tree. If its value is less than the current element in the tree, then move towards the left side of that element or else to its right. If there is no sub tree on the left, then make your new element as the left child of that current element or else compare it with the existing left child and follow the same rule. Exactly, the same has to done for the case when your new element is greater than the current element in the tree but this time with the right child. Though this logic is followed for the creation of a Binary tree, this logic is often suitable to search for a key value in the binary tree.

Algorithm for the implementation of a Binary tree:

Step-1: If value of new element < current element, then go to step-2 or else step -3
Step-2: If the current element does not have a left sub-tree, then make your new

```
        element the left child of the current element; else make the existing left child
        as your current element and go to step-1
Step-3: If the current element does not have a right sub-tree, then make your new
        element the right child of the current element; else make the existing right
        child as your current element and go to step-1
```

Program 6.1 depicts the segment of code for the creation of a binary tree.

```
struct NODE
{
 struct NODE *left;
 int value;
 struct NODE *right;
};

create_tree( struct NODE  *curr, struct NODE  *new )
{
  if(new->value <= curr->value)
    {
      if(curr->left != NULL)
      create_tree(curr->left, new);
      else
      curr->left = new;
    }
 else
  {
     if(curr->right != NULL)
    create_tree(curr->right, new);
    else
    curr->right = new;
  }
}
```

Program 6.1 : Binary tree creation

Array-based representation of a Binary Tree

Consider a complete binary tree T having n nodes where each node contains an item (value). Label the nodes of the complete binary tree T from top to bottom and from left to right 0, 1, ..., n-1. Associate with T the array A where the i^{th} entry of A is the item in the node labelled i of T, i = 0, 1, ..., n-1. Figure 6.11 depicts the array representation of a Binary tree of Figure 6.16.

Given the index **i** of a node, we can easily and efficiently compute the index of its parent and left and right children:

Index of Parent: (i – 1)/2, Index of Left Child: 2i + 1, Index of Right Child: 2i + 2.

Node #	Item	Left child	Right child
0	A	1	2
1	B	3	4
2	C	-1	-1
3	D	5	6
4	E	7	8
5	G	-1	-1
6	H	-1	-1
7	I	-1	-1
8	J	-1	-1
9	?	?	?

Figure 6.11 : Array Representation of a Binary Tree

First column represents index of node, second column consist of the item stored in the node and third and fourth columns indicate the positions of left and right children (–1 indicates that there is no child to that particular node.)

6.7 BINARY TREE TRAVERSALS

We have already discussed about three tree traversal methods in the previous section on general tree. The same three different ways to do the traversal – preorder, inorder and postorder are applicable to binary tree also.

Let us discuss the inorder binary tree traversal for following binary tree (refer to *Figure 6.12*):

We start from the root i.e. * We are supposed to visit its left sub-tree then visit the node itself and its right sub-tree. Here, root has a left sub-tree rooted at +. So, we move to + and check for its left sub-tree (we are suppose repaeat this for every node). Again, + has a left sub-tree rooted at 4. So, we have to check for 4's left sub-tree now, but 4 doesn't have any left sub-tree and thus we will visit node 4 first (print in our case) and check for its right sub-tree. As 4 doesn't have any right sub-tree, we'll go back and visit node +; and check for the right sub-tree of +. It has a right sub-tree rooted at 5 and so we move to 5. Well, 5 doesn't have any left or right sub-tree. So, we just visit 5 (print 5)and track back to +. As we have already visited + so we track back to * . As we are yet to visit the node itself and so we visit * before checking for the right sub-tree of *, which is 3. As 3 does not have any left or right sub-trees, we visit 3.
So, the inorder traversal results in 4 + 5 * 3

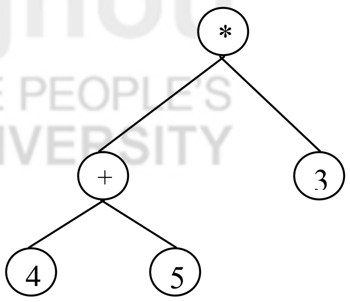


Figure 6.12 : A binary tree

```
Algorithm: Inorder
Step-1: For the current node, check whether it has a left
        child. If it has, then go to step-2 or
        else go to step-3
Step-2: Repeat step-1 for this left child
Step-3: Visit (i.e. printing the node in our case) the
        current node
Step-4: For the current node check whether it has a right
        child. If it has, then go to step-5
Step-5: Repeat step-1 for this right child
```

The preoreder and postorder traversals are similar to that of a general binary tree. The general thing we have seen in all these tree traversals is that the traversal mechanism is inherently recursive in nature.

6.7.1 Recursive Implementation of Binary Tree Traversals

There are three classic ways of recursively traversing a binary tree. In each of these, the left and right sub-trees are visited recursively and the distinguishing feature is **when the element in the root is visited or processed**.

Program 6.2, Program 6.3 and Program 6.4 depict the inorder, preorder and postorder traversals of a Binary tree.

```
struct NODE
{
 struct NODE *left;
 int value;      /* can be of any type */
 struct NODE *right;
};

inorder(struct NODE *curr)
{
 if(curr->left != NULL) inorder(curr->left);
 printf("%d", curr->value);

 if(curr->right != NULL) inorder(curr->right);

}
```

Program 6.2 : Inorder traversal of a binary tree

```
struct NODE
{
 struct NODE *left;
 int value;      /* can be of any type */
 struct NODE *right;
};

preorder(struct NODE *curr)
{
 printf("%d", curr->value);
 if(curr->left != NULL) preorder(curr->left);
 if(curr->right != NULL) preorder(curr->right);

}
```

Program 6.3 : Preorder traversal of a binary tree

```
struct NODE
{
 struct NODE *left;
 int value;      /* can be of any type */
 struct NODE *right;
};

postorder(struct NODE *curr)
{
  if(curr->left != NULL) postorder(curr->left);
 if(curr->right != NULL) postorder(curr->right);

printf("%d", curr->value);

}
```

Program 6.4 : Postorder traversal of a binary tree

In a preorder traversal, the root is visited first (pre) and then the left and right sub-trees are traversed. In a postorder traversal, the left sub-tree is visited first, followed by right sub-tree which is then followed by root. In an inorder traversal, the left sub-tree is visited first, followed by root, followed by right sub-tree.

6.7.2 Non-recursive implementation of binary tree traversals

As we have seen, as the traversal mechanisms were inherently recursive, the implementation was also simple through a recursive procedure. However, in the case of a non-recursive method for traversal, it has to be an iterative procedure; meaning, all the steps for the traversal of a node have to be under a loop so that the same can be applied to all the nodes in the tree.

Algorithm : Non-recursive preorder binary tree traversal

```
Stack S
push root onto S
repeat until S is empty
{
   v = pop S
   if v is not NULL
   visit v
   push v's right child onto S
   push v's left child onto S
}
```

Program 6.5 depicts the program segment for the implementation of non-recursive preorder traversal.

```
/* preorder traversal of a binary tree, implemented using a stack */
void preorder(binary_tree_type *tree)
{
  stack_type *stack;
  stack = create_stack();
  push(tree, stack);          /* push the first element of the tree to the stack */
  while (!empty(stack))
    {
    tree = pop(stack);
    visit(tree);
    push(tree->right, stack);  /* push right child to the stack */
    push(tree->left, stack);     /* push left child to the stack */
    }
 }
```

Program 6.5: Non-recursive implementation of preorder traversal

In the worst case, for preorder traversal, the stack will grow to size n/2, where n is number of nodes in the tree. Another method of traversing binary tree non-recursively which does not use stack requires pointers to the parent node (called threaded binary tree).

A threaded binary tree is a binary tree in which every node that does not have a right child has a THREAD (a third link) to its INORDER successor. By doing this threading we avoid the recursive method of traversing a tree and use of stack, which makes use of a lot of memory and time.

A node structure of threaded binary is :

The node structure for a threaded binary tree varies a bit and its like this –

```
struct NODE
{
```

```
struct NODE *leftchild;
int node_value;
struct NODE *rightchild;
struct NODE *thread;   /* third pointer to it's inorder successor */
}
```

6.8 APPLICATIONS

Trees are used enormously in computer programming. These can be used for improving database search times (binary search trees, 2-3 trees, AVL trees, red-black trees), Game programming (minimax trees, decision trees, pathfinding trees), 3D graphics programming (quadtrees, octrees), Arithmetic Scripting languages (arithmetic precedence trees), Data compression (Huffman trees), and file systems (B-trees, sparse indexed trees, tries). Figure 6.13 depicts a tic-tac-toe game tree showing various stages of game.

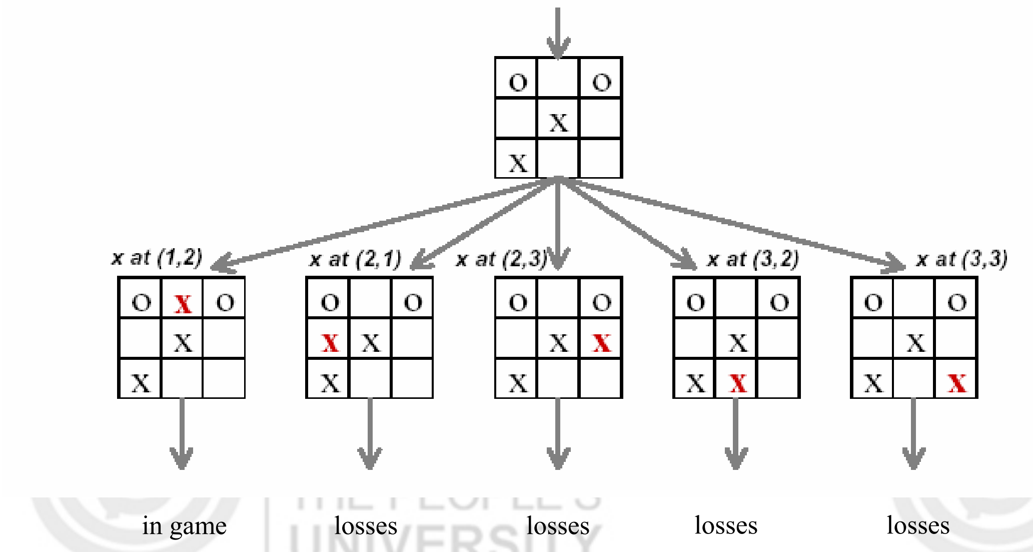


Figure 6.13 : A tic-tac-toe game tree showing various stages of game

In all of the above scenario except the first one, the player (playing with X) ultimately looses in subsequent moves.

The General tree (also known as Linked Trees) is a generic tree that has one root node, and every node in the tree can have an unlimited number of child nodes. One popular use of this kind of tree is in Family Tree programs. In game programming, many games use these types of trees for decision-making processes as shown above for tic-tac-toe. A computer program might need to make a decision based on an event that happened.

But this is just a simple tree for demonstration. A more complex AI decision tree would definitely have a lot more options. The interesting thing about using a tree for decision-making is that the options are cut down for every level of the tree as we go down, greatly simplifying the subsequent moves and improving the speed at which the AI program makes a decision.

The big problem with tree based level progressions, however, is that sometimes the tree can get too large and complex as the number of moves (level in a tree) increases. Imagine a game offering just two choices for every move to the next level at the end of each level in a ten level game. This would require a tree of 1023 nodes to be created.

Binary trees are used for searching keys. Such trees are called Binary Search trees(refer to *Figure 6.14*).

A Binary Search Tree (BST) is a binary tree with the following properties:

1. The key of a node is always greater than the keys of the nodes in its left sub-tree
2. The key of a node is always smaller than the keys of the nodes in its right sub-tree

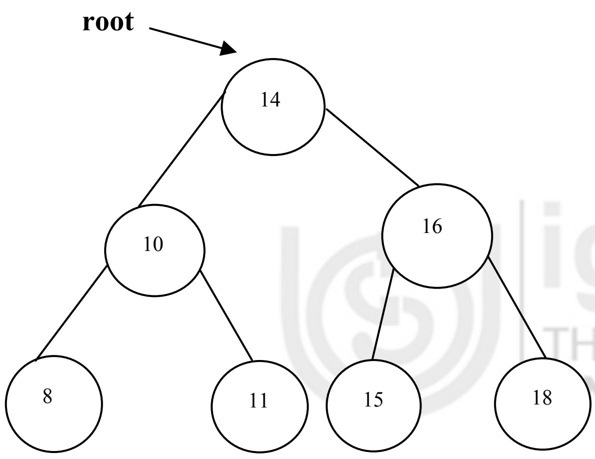


Figure 6.14 : A binary search tree (BST)

It may be seen that when nodes of a BST are traversed by inorder traversal, the keys appear in sorted order:

```
inorder(root)
{
inorder(root.left)
print(root.key)
inorder(root.right)
}
```

Binary Trees are also used for evaluating expressions.
A binary tree can be used to represent and evaluate arithmetic expressions.

1. If a node is a leaf, then the element in it specifies the value.
2. If it is not a leaf, then evaluate the children and combine them according to the operation specified by the element.

Figure 6.15 depicts a tree which is used to evaluate expressions.

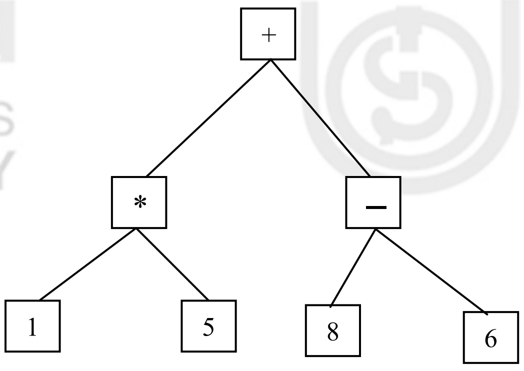


Figure 6.15 : Expression tree for 1 * 5 + 8 - 6

☞ **Check Your Progress 2**

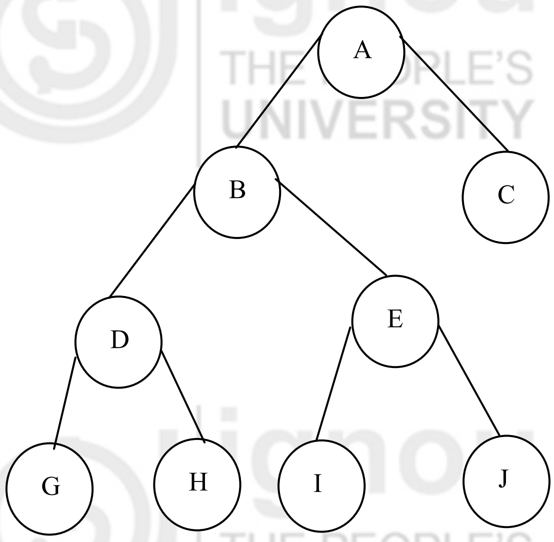


Figure 6.16 : A binary tree

1) With reference to *Figure 6.16*, find
   a) the leave nodes in the binary tree
   b) sibling of J
   c) Parent node of G
   d) depth of the binary tree
   e) level of node J
2) Give preorder, post order, inorder and level order traversal of above binary tree
3) Give array representation of the binary tree of *Figure 6.12*
4) Show that in a binary tree of N nodes, there are N+1 children with both the links as null (leaf node).

6.9 SUMMARY

Tree is one of the most widely used data structure employed for representing various problems. We studied tree as a special case of an acyclic graph. However, rooted trees are most prominent of all trees. We discussed definition and properties of general trees with their applications. Various tree traversal methods are also discussed.

Binary tree are the special case of trees which have at most two children. Binary trees are mostly implemented using link lists. Various tree traversal mechanisms include inorder, preorder and post order. These tree traversals can be implemented using recursive procedures and non-recursive procedures. Binary trees have wider applications in two way decision making problems which use yes/no, true/false etc.

6.10 SOLUTIONS / ANSWERS

Check Your Progress 1

1) If a tree has e edges and n vertices, then e=n – 1. Hence, if a tree has 45 edges, then it has 46 vertices.

2) A full 4-ary tree with 100 leaves has i=(100 – 1)/(4 – 1)=33 internal vertices.

3) A full 3-ary tree with 100 internal vertices has l = (3 – 1)*100+ 1=201 leaves

Check Your Progress 2

1) Answers
   a. G,H,I and J
   b. I
   c. D
   d. 4
   e. 4
2) Preorder : ABDCHEIJC Postorder : GHDIJEBCA Inorder : GDHBIEFAC level-order: ABCDEGHIJ
3) Array representation of the tree in *Figure 6.12*

Index of Node	Item	Left child	Right child
0	*	1	2
1	+	3	4
2	3	-1	-1
3	4	-1	-1
4	5	-1	-1
5	?	?	?

6.11 FURTHER READINGS

1. *Fundamentals of Data Structures in C++* by E.Horowitz, Sahni and D.Mehta; Galgotia Publications.
2. *Data Structures and Program Design in C* by Kruse, C.L.Tonodo and B.Leung; Pearson Education.

Reference websites

http://www.csee.umbc.edu
http://www.cse.ucsc.edu
http://www.webopedia.com