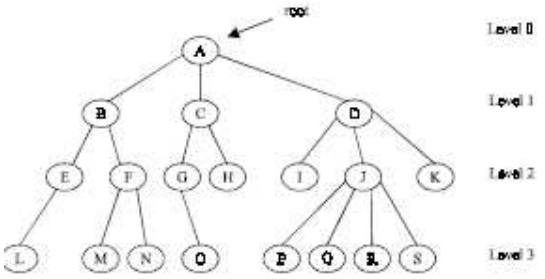
MODULE 3:

TREES BASIC TERMINOLOGY

Trees are very flexible, versatile and powerful non-liner data structure that can be used to represent data items possessing hierarchical relationship between the grand father and his children and grand children as so on.



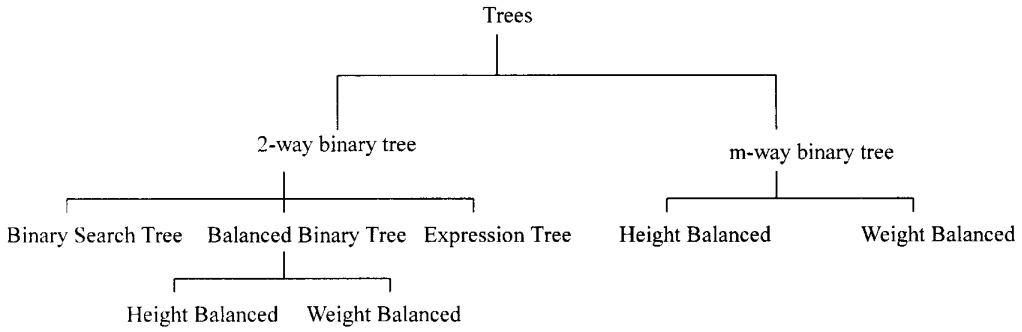
A tree is an ideal data structure for representing hierarchical data. A tree can be theoretically defined as a finite set of one or more data items (or nodes) such that :

1. There is a special node called the root of the tree.
2. Removing nodes (or data item) are partitioned into number of mutually exclusive (*i.e.*, disjoined) subsets each of which is itself a tree, are called sub tree.

BASIC TERMINOLOGIES

Root is a specially designed node (or data items) in a tree. It is the first node in the hierarchical arrangement of the data items. 'A' is a root node in the Fig. above. Each data item in a tree is called a *node*. It specifies the data information and links (branches) to other data items.

Trees can be divided in different classes as follows :



Binary trees:

A binary tree is a tree in which no node can have more than two children. Typically these children are described as "left child" and "right child" of the parent node.

A binary tree T is defined as a finite set of elements, called nodes, such that :

1. T is empty (*i.e.*, if T has no nodes called the *null tree* or *empty tree*).

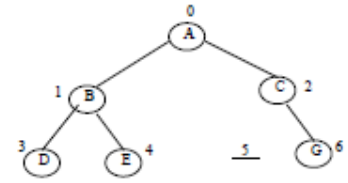
2. T contains a special node R, called root node of T, and the remaining nodes of T form an ordered pair of disjoined binary trees T1 and T2, and they are called left and right sub tree of R. If T1 is non empty then its root is called the left successor of R, similarly if T2 is non empty then its root is called the right successor of R.

BINARY TREE REPRESENTATION

This section discusses two ways of representing binary tree T in memory :
1. Sequential representation using arrays
2. Linked list representation

ARRAY REPRESENTATION

An array can be used to store the nodes of a binary tree. The nodes stored in an array of memory can be accessed sequentially. Suppose a binary tree T of depth *d*. Then at most 2^{d-1} nodes can be there in T.(*i.e.*, SIZE = 2^d–1) So the array of size "SIZE" to represent the binary tree. Consider a binary tree of depth 3. Then SIZE = 2^3 – 1 = 7. Then the array A[7] is declared to hold the nodes.



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

The array representation of the binary tree is shown. To perform any operation often we have to identify the father, the left child and right child of an arbitrary node.

1. The father of a node having index n can be obtained by (n – 1)/2. For example to find the father of D, where array index n = 3. Then the father nodes index can be obtained
= (n – 1)/2
= 3 – 1/2
= 2/2
= 1
i.e., A[1] is the father D, which is B.

2. The left child of a node having index n can be obtained by (2n+1). For example to find the left child of C, where array index n = 2. Then it can be obtained by
= (2n +1)
= 2*2 + 1
= 4 + 1
= 5
i.e., A[5] is the left child of C, which is NULL. So no left child for C.

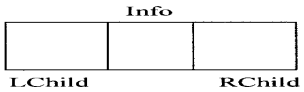
3. The right child of a node having array index n can be obtained by the formula (2n + 2). For example to find the right child of B, where the array index n = 1. Then
= (2n + 2)
= 2*1 + 2
= 4

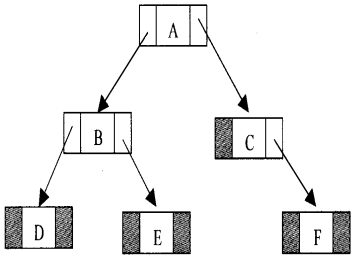
i.e., A[4] is the right child of B, which is E.

4. If the left child is at array index *n*, then its right brother is at (*n* + 1). Similarly, if the right child is at index *n*, then its left brother is at (*n* – 1). The array representation is more ideal for the complete binary tree. The tree in Fig. above is not a complete binary tree. Since there is no left child for node C, *i.e.*, A[5] is vacant. Even though memory is allocated for A[5] it is not used, so wasted unnecessarily.

LINKED LIST REPRESENTATION

The most popular and practical way of representing a binary tree is using linked list (or pointers). In linked list, every element is represented as nodes. A node consists of three fields such as :
(*a*) Left Child (LChild)
(*b*) Information of the Node (Info)
(*c*) Right Child (RChild)





If a node does not have left/right child, corresponding left/right child is assigned to NULL.

COMPLETE BINARY TREES

A complete binary tree of depth 'd' is strictly the binary tree where all the leaf nodes are at level d.

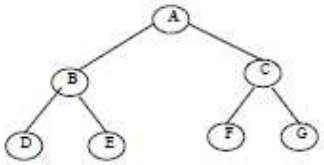


Fig: Complete binary tree

A binary tree with *n* nodes, $n > 0$, has exactly $n - 1$ edges. A binary tree of depth *d*, $d > 0$, has at least *d* and at mast $2^d - 1$ nodes in it. If a binary tree contains *n* nodes at level l, then it contains at most $2n$ nodes at level $l + 1$. A complete binary tree of depth *d* is the binary tree of depth *d* contains exactly 2 *l* nodes at each level *l* between 0 and *d*.

ALGEBARIC EXPRESSIONS

The main application of a 2-tree is to represent and compute any algebraic expression using binary operation. For example, consider an algebraic expression E.
$E = (a + b)/((c - d)*e)$

E can be represented by means of the extended binary tree T as shown in Fig. below. Each variable or constant in E appears as an internal node in T whose left and right sub tree corresponds to the operands of the operation.

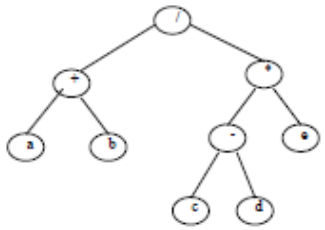
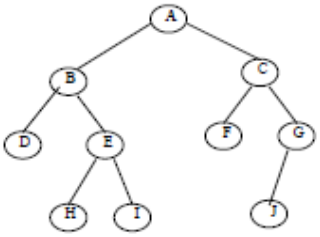


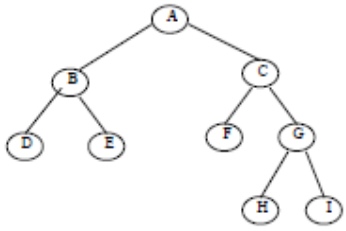
Fig: Expression tree

EXTENDED BINARY TREES

Consider a binary tree T in Fig. below. Here 'A' is the root node of the binary tree T. Then 'B' is the left child of 'A' and 'C' is the right child of 'A' *i.e.*, 'A' is a father of 'B' and 'C'. The node 'B' and 'C' are called brothers, since they are left and right child of the same father. If a node has no child then it is called a leaf node.



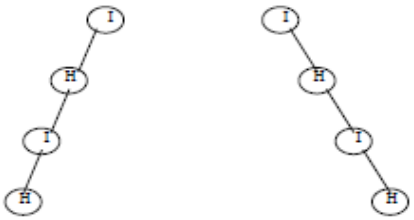
Nodes P,H,I,F,J are leaf nodes. The tree is said to be *strictly binary tree*, if every non-leaf made in a binary tree has non-empty left and right sub trees. A strictly binary tree with *n* leaves always contains 2*n*– 1 nodes. The tree in Fig. below is strictly binary tree, where as the tree in Fig. above is not. That is every node in the strictly binary tree can have either no children or two children. They are also called 2-*tree* or *extended binary tree.*



Finally, let us discuss in briefly the main difference between a binary tree and ordinary tree is:

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

Fig. below shows left skewed and right skewed binary trees respectively.



(a) Left skewed (b) Right skewed

The basic operations performed on binary trees are listed as follows:

1.Create an Empty Binary tree
2. Traversing a binary tree
3. Insert a new mode
4. Deleting a Node
5. Searching for a Node
6. Copying the mirror image of a tree
7. Determine the total no: of Nodes
8. Determine the total no: leaf Nodes
9. Determine the total no: non-leaf Nodes
10. Find the smallest element in a Node
11. Finding the largest element
12. Find the Height of the tree
13. Finding the Father/Left Child/Right Child/Brother of an arbitrary node
Some primitive operations are discussed in the next few sections.

TREE TRAVERSAL ALGORITHMS

Tree traversal is one of the most common operations performed on tree data structures. It is a way in which each node in the tree is visited exactly once in a systematic manner. There are three standard ways of traversing a binary tree. They are:

1. Pre Order Traversal (Node-left-right)
2. In order Traversal (Left-node-right)
3. Post Order Traversal (Left-right-node)

PRE ORDER TRAVERSAL

To traverse a non-empty binary tree in pre order following steps one to be processed
1. Visit the root node
2. Traverse the left sub tree in preorder
3. Traverse the right sub tree in preorder

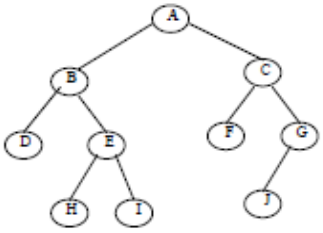


Fig 1

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

The preorder traversal non-recursively algorithms uses a variable PN (Present Node), which will contain the location of the node currently being scanned. Left(R) denotes the left child of the node R and Right(R) denoted the right child of R. A stack is used to hold the addresses of the nodes to be processed. Info(R) denotes the information of the node R.
Preorder traversal starts with root node of the tree *i.e*., PN = ROOT. Then repeat the following steps until PN = NULL.

Step 1: Process the node PN. If any right child is there for PN, push the Right (PN) into the top of the stack and proceed down to left by PN = Left (PN), if any left child is there (*i.e*., Left (PN) not equal to NULL). Repeat the step 2 until there is no left child (*i.e.*, Left (PN) = NULL).
Step 2: Now we have to go back to the right node(s) by backtracking the tree. This can be achieved by popping the top most element of the stack. Pop the top element from the stack and assigns to PN.
Step 3: If (PN is not equal to NULL) go to the Step 1
Step 4: Exit
The preorder traversal of a binary tree in Fig. 1 is A, B, D, E, H, I, C, F, G, J.

IN ORDER TRAVERSAL

The in order traversal of a non-empty binary tree is defined as follows :
1. Traverse the left sub tree in order
2. Visit the root node
3. Traverse the right sub tree in order

In order traversal, the left sub tree is traversed recursively, before visiting the root. After visiting the root the right sub tree is traversed recursively, in order fashion. The procedure for an in order traversal is given below :

The in-order traversal algorithm uses a variable PN, which will contain the location of the node currently being scanned. Info (R) denotes the information of the node R, Left (R) denotes the left child of the node R and Right (R) denotes the right child of the node R. In-order traversal starts from the ROOT node of the tree (*i.e.*, PN = ROOT). Then repeat the following steps until PN = NULL.

Step 1: Proceed down to left most node of the tree by pushing the root node onto the stack.
Step 2: Repeat the step 1 until there is no left child for a node.
Step 3: Pop the top element of the stack and process the node. PN = STACK[TOP]
Step 4: If the stack is empty then go to step 6.
Step 5: If the popped element has right child then PN = Right(PN). Then repeat the step from 1.
Step 6: Exit.

The in order traversal of a binary tree in Fig 1 is D, B, H, E, I, A, F, C, J, G.

POST ORDER TRAVERSAL

The post order traversal of a non-empty binary tree can be defined as :
1. Traverse the left sub tree in post order
2. Traverse the right sub tree in post order
3. Visit the root node
In Post Order traversal, the left and right sub tree(s) are recursively processed before visiting the root.

The post-order traversal algorithm uses a variable PN, which will contain the location of the node currently being scanned. Left (R) denotes the left child of the node R and Right (R) denotes the right child of the node R. Info (R) denotes the information of the node R. The post-order traversal algorithm is more complicated than the proceeding two algorithms, because here we have to push the information of the node PN to stack in two

different situations. These two situations are distinguished between by pushing Left(PN) and - Right(PN) on to stack. That is whenever a negative node sees in the stack; it means that it was a right child of a node. Post-order traversal starts from the ROOT node of the tree (*i.e.*, PN = ROOT).

Step 1: Proceed down to left most node of the tree by pushing the root node and - Right(PN) on the stack.
Step 2: Repeat the Step 1 until there is no left child for the node.
Step 3: Pop and display the positive nodes on the stack.
Step 4: If the stack is empty, go to Step 6
Step 5: If a negative node is popped, then PN = – PN (*i.e.*, to remove the negative sign in the node) and go to Step 1.
Step 6: Exit

The post order traversal of a binary tree in Fig.1 is D, H, I, E, B, F, J, G, C, A

THREADED BINARY TREE

Traversing a binary tree is a common operation and it would be helpful to find more efficient method for implementing the traversal. Moreover, half of the entries in the Lchild and Rchild field will contain NULL pointer. These fields may be used more efficiently by replacing the NULL entries by special pointers which points to nodes higher in the tree. Such types of special pointers are called threads and binary tree with such pointers are called threaded binary tree. Fig. below shows the threaded binary tree with threads replacing NULL pointer of a binary tree . The threads are drawn with dotted lines to differentiate then from tree pointers.

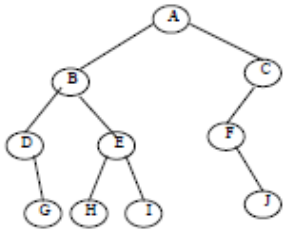


Fig: Binary tree

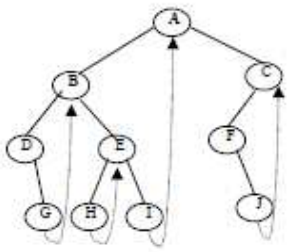


Fig: Threaded binary tree

There are many ways to thread a binary tree. Right most nodes in the threaded binary tree have a NULL right pointer (*i.e.*, in-order successor). Such trees are called right in threaded binary trees. A left in threaded binary tree may be defined similarly as one in which each NULL left pointer is altered to contain a thread (*i.e.*, in-order predecessor). An in-threaded binary tree may be defined as a binary tree that is both left-in-threaded and right-in-threaded. We can implement a right in threaded binary tree using arrays by distinguishing threads from ordinary pointers. Threads are denoted by negative numbers, when ordinary pointers are denoted by positive integers. The array representation of the right in thread binary tree in Fig above is shown in Table below.

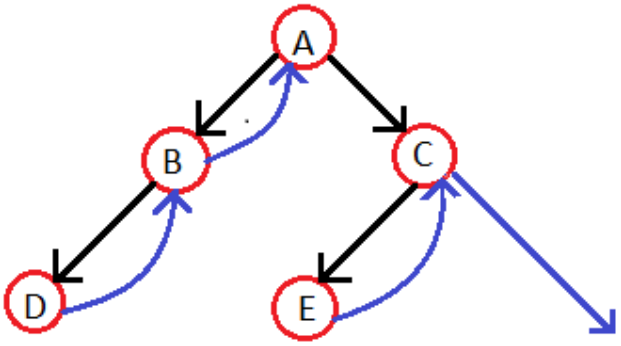
	Info	*Lchild*	*Rchild*
A[0]	A	1	2
A[1]	B	3	4
A[2]	C	5	
A[3]	D		8
A[4]	E	9	10
A[5]	F		12
A[6]			
A[7]			
A[8]	G		- 1
A[9]	H		- 4
A[10]	I		- 0
A[11]			
A[12]	J		- 2
A[13]			
A[14]			

To implement a right-in-threaded binary tree using dynamic memory allocation, an extra 1 bit logical field, rthread, is used to distinguish threads from ordinary pointers. If a right pointer of a node is threaded, then the rthread = TRUE otherwise FALSE.

TRAVERSING THREADED BINARY TREES

Non Recursive Inorder Traversal for a Threaded Binary Tree:

As this is 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. I'll consider the INORDER traversal again. Here, for every node, we'll visit the left sub-tree (if it exists) first (if and only if we haven't visited it earlier); then we visit (i.e. print its value, in our case) the node itself and then the right sub-tree (if it exists). If the right sub-tree is not there, we check for the threaded link and make the threaded node the current node in consideration. Please, follow the example given below.



Algorithm

Step-1: For the current node check whether it has a left child which is not there in the visited list. If it has then go to step-2 or else step-3.

Step-2: Put that left child in the list of visited nodes and make it your current node in consideration. Go to step-6.

Step-3: For the current node check whether it has a right child. If it has then go to step-4 else go to step-5

Step-4: Make that right child as your current node in consideration. Go to step-6.

Step-5: Check for the threaded node and if its there make it your current node.

Step-6: Go to step-1 if all the nodes are not over otherwise quit

Working of the algorithm for the figure given above:

step-1	'A' has a left child i.e. B, which has not been visited.So, we put B in our "list of visited nodes" and B becomes our current node in consideration.	B		
step-2	'B' also has a left child, 'D', which is not there in our list of visited nodes. So, we put 'D' in that list and make it our current node in consideration.	B D		
step-3	'D' has no left child, so we print 'D'. Then we check for its right child. 'D' has no right child and thus we check for its thread-link. It has a thread going till node	B D	D	

	'B'. So, we make 'B' as our current node in consideration.		
step-4	'B' certainly has a left child but its already in our list of visited nodes. So, we print 'B'. Then we check for its right child but it doesn't exist. So, we make its threaded node (i.e. 'A') as our current node in consideration.	B D	D B
step-5	'A' has a left child, 'B', but its already there in the list of visited nodes. So, we print 'A'. Then we check for its right child. 'A' has a right child, 'C' and it's not there in our list of visited nodes. So, we add it to that list and we make it our current node in consideration.	B D C	D B A
step-6	'C' has 'E' as the left child and it's not there in our list of visited nodes even. So, we add it to that list and make it our current node in consideration.	B D C E	D B A
step-7		and finally.....	D B A E C