[DATA STRUCTURES]

Chapter - 07 : Trees

"A Tree is a non-linear data structure in which items are arranged in a sorted sequence. It is used to represent hierarchical relationship existing amongst several data items."

The graph theoretic definition of tree is : it is a finite set of one or more data items (nodes) such that

1. There is a special data item called the root of the tree.
2. And its remaining data items are partitioned into number of mutually exclusive (i.e. disjoint) subsets, each of which is itself a tree. And they are called **subtrees**.

A tree is shown in Fig. (1).

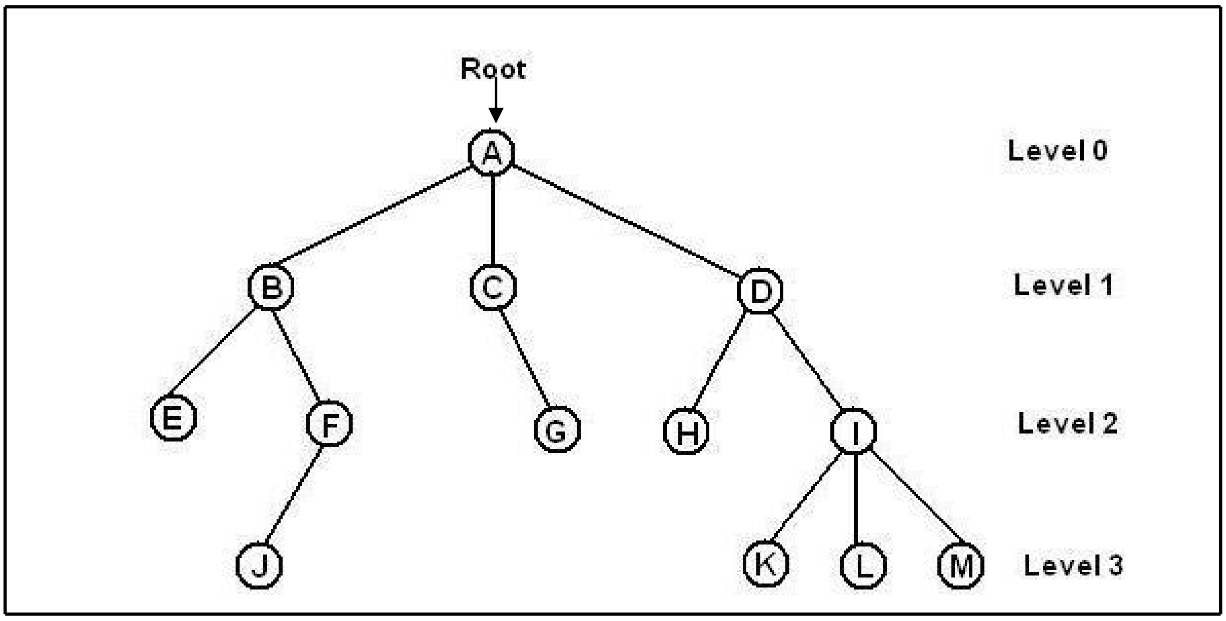


Fig. (1) : A Tree

Natural trees grow upwards from the ground into the air. But, tree data structure grows downwards from top to bottom.

TREE TERMINOLOGY

There are number of terms associated with the threes which are listed below :

1. Root

2. Node
3. Degree of a node
4. Degree of a tree
5. Terminal node (s)
6. Non-terminal node (s)
7. Siblings
8. Level
9. Edge
10. Path
11. Depth
12. Forest

Root

It is specially designed data item in a tree. It is the first in the hierarchical arrangement of data items. In the above tree, A is the **root** item.

Node

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

Degree of a node

It is the number of subtrees of a node in a given tree. In the above tree

- The degree of node A is 3
- The degree of node C is 1
- The degree of node B is 2

- The degree of node H is 0
- The degree of node I is 3.

Degree of a tree

It is the maximum degree of nodes in a given tree. In the above tree, the node A has degree 3 and another node I is also having its degree 3. In all this value is the maximum. So, the degree of the above tree is 3.

Terminal node (s)

A node with degree zero is called a terminal node or a leaf. In the above tree, there are 7 terminal nodes. They are E, J, G, H, K , L and M.

Non-Terminal Node (s)

Any node (except the root node) whose degree is not zero is called non-terminal node. Non-terminal nodes are the intermediate nodes in traversing the given tree from its root node to the terminal nodes (leaves). There are 5 non-terminal nodes.

Siblings

The children nodes of a given parent node are called **siblings**. They are also called brothers. In the above tree,

- E and F are siblings of parent node B.
- K, L and M are siblings of parent node I.

Level

The entire tree structure is leveled in such a way that the root node is always at level 0. Then, its immediate children are at level 1 and their immediate children are at level 2 and so on upto the terminal nodes. In general, if a node is at level n, then the children will be at level n + 1 in the four levels.

Edge

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

Path

It is a sequence of consecutive edges from the source node to the destination node. IN the above tree, the path between A and J is given by the node pairs,

(A, B), (B, F) and (F, J)

Depth

It is the maximum level of any node in a given tree. In the above tree, the root node A has the maximum level i.e., the number of levels one can descend the tree from its root to the terminal nodes (leaves). The term height is also used to denote the depth.

Forest

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

BINARY TREES

Binary tree :

A binary tree consists of a finite set of elements that can be partitioned into three distinct sub-sets called the **root**, the **left sub-tree** and the **right sub-tree**. If there are no elements in the binary tree it is called an **empty binary tree**.

Node :

Each element present in a binary tree is called a **node** of that tree.

Root :

The element that represents the base node of the tree is called the **root** of the tree.

The left and right sub-tree :

Apart from the root, the other two sub-sets of a binary tree are binary trees. They are called the **left** and **right sub-trees** of the original tree. Any of these sub-sets can be empty.

The tree shown in fig. (2) consists of nine nodes. This tree has a root node **A**, a left sub-tree formed by nodes **B, D** and the right sub-tree formed by nodes **C, E, F, G, H** and **I**. The left sub-tree is itself a binary with **B** as the root node, **D** as the left sub-tree and an empty right sub-tree. Similarly, the right sub-tree has **C** as the root right sub-tree. The left and right sub-trees are always shown using branches coming out from the root node. If the root node doesn't have a left or a right branch then that sub-tree is empty. For example, the binary trees with root nodes as **D, G, H** and **I** have empty right and left sub-trees.

Fig. (3) shows some structures that are not binary trees :

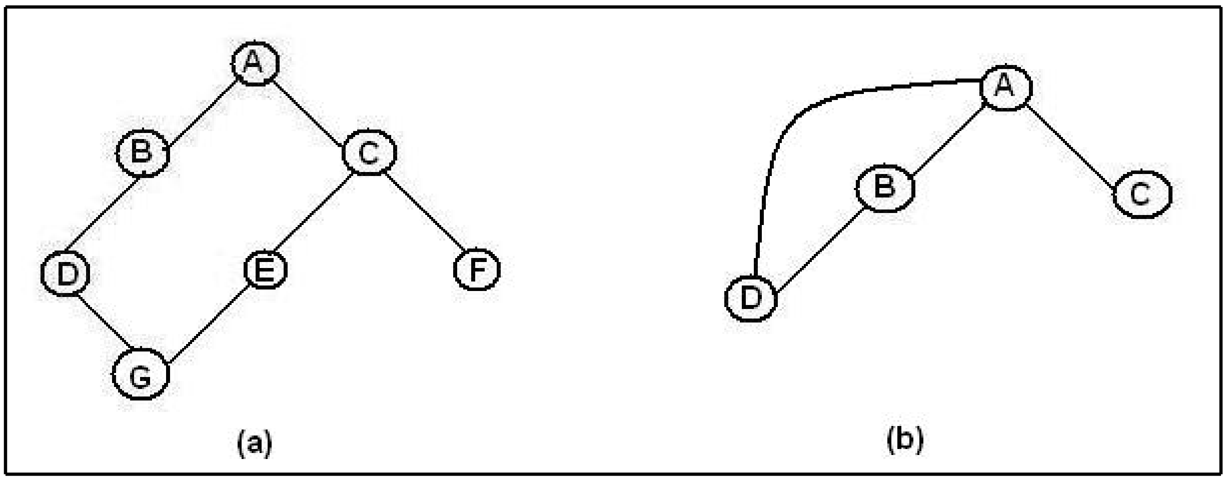


Fig. (3) : Trees that are not binary trees.

Consider some definitions that are used in association with binary trees.

Father and son :

Suppose **A** is the root node of a binary tree and **B** is the root of its left or right sub-tree. In this case, **A** is said to be the **father** of **B** and **B** is said to be the **left son** and **C** is said to be **right son** of **A**.

Leaf node :

A node that does not have any sons (such as **D, G, H** or **I** shown in Fig. (2) is called a **leaf** node.

Ancestor and descendant : A node **A** is said to be an **ancestor** of node **B**, if **A** is either the father of **B** or the father of some **ancestor** of **B**. For example, in the tree shown in Fig. (2), **A** is an ancestor of **C**. A node **B** is said to be a **left descendant** of node **A** if **B** is either the left son of **A** or a descendant of the left son of **A**. In a similar fashion we can define the right descendant.

Unlike natural trees, the tree data structures are depicted with root node at the top and the leaves at the bottom. The common convention used about direction is *"down"* from the root node to leaf nodes and *"up"* from leaf node to root node.

Climbing and descending :

When we are traversing the tree from the leaf node to the root node the operation is **climbing**. Similarly, traversing the tree from the root to the leaves is called **descending** the tree.

Strictly binary trees :

A binary tree is called a **strictly binary tree** if every non-leaf node in a binary tree has non-empty left and right sub-trees. For example, the tree shown in Fig. (4) is a strictly binary tree, whereas, the tree shown in Fig. (2) is not a strictly binary tree since nodes **B** and **E** in it have one son each.

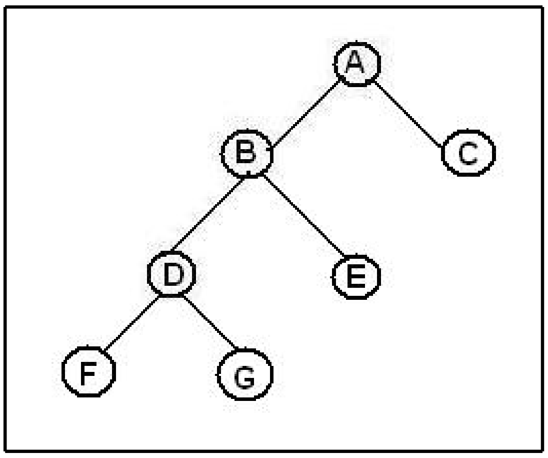


Fig. (4) Strictly binary tree

Degree :

The number of nodes connected to a particular node is called the **degree** of that node. For example, in Fig. (4) the node containing data **D** has a degree 3. The degree of a leaf node is always one.

Level :

The root node of the tree has level 0. The level of any other child node is one more than the level of its father. For example, in the binary tree shown in Fig. (2), node **E** is at level 2 and node **H** is at level 3.

Depth :

The maximum level of any leaf node in the tree is called the **depth** of the binary tree. For example, the depth of the tree shown in Fig. (2) is 3.

Complete binary tree :

A strictly binary tree all of whose leaf nodes are at the same level is called a **complete binary tree**. Fig. (5) shows the complete binary tree. The depth of this tree is 2.

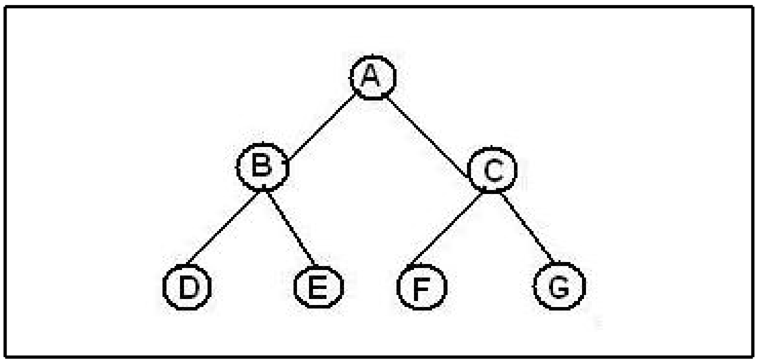


Fig. (5) Complete binary tree

Extended Binary tree :

A binary tree can be converted to an extended binary tree by adding new nodes to its leaf nodes and to the nodes that have only one child. These new nodes are added in such a way that all the nodes in the resultant tree have either **0** or **2** children. The extended tree is also known as a **2-tree**. The nodes of the original tree are called **internal** nodes and the new nodes that are added to binary tree, to make it an extended binary tree are called **external** nodes.

Fig. 6 shows how extended binary tree (Fig 6(b) can be obtained by adding new nodes to original tree (Fig. 6(a)). In Fig. 6, all the nodes with circular shape are internal nodes and all the nodes with square shape are external nodes.

A few important points about Extended Binary tree are :

(a) If a tree has **n** nodes then the number of branches it has is (**n-1**).
(b) Except the root node every node in a tree has exactly one parent.
(c) Any two nodes of a tree are connected by only one single path.
(d) For a binary tree of height **h** the maximum number of nodes can be $\mathbf{2^{h+1} - 1}$.
(e) Any binary tree with n internal nodes has **(n + 1)** external nodes.

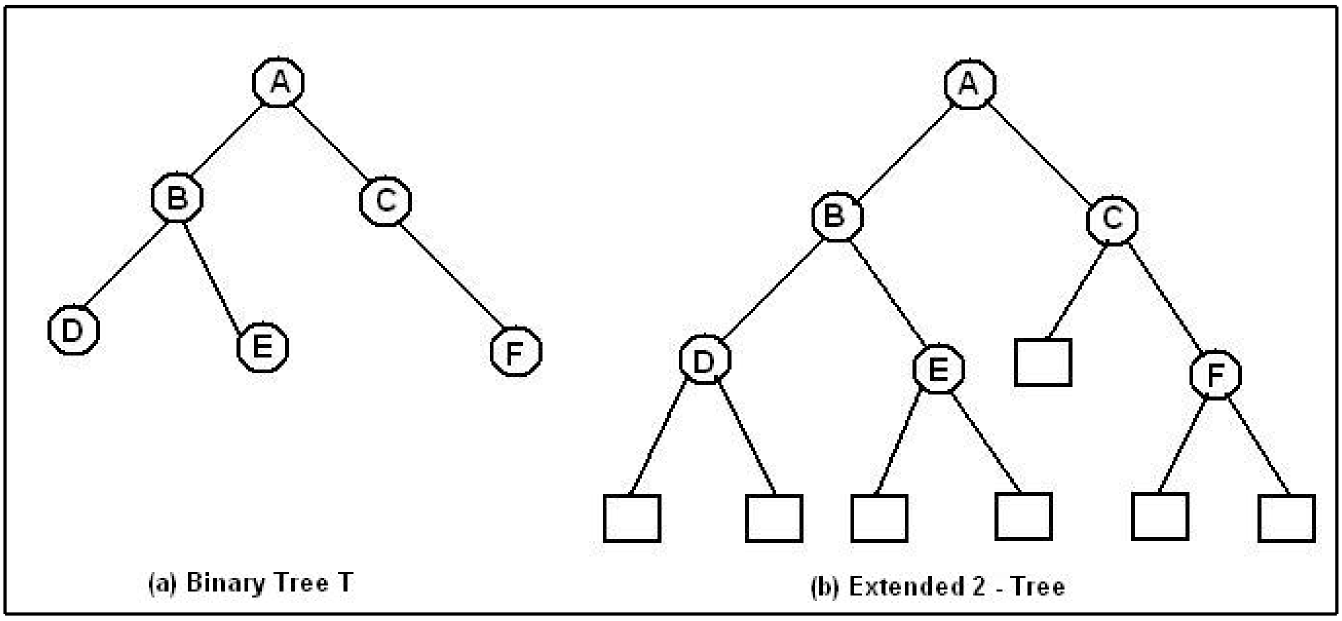


Fig. (6) Converting a binary tree T into a 2-tree

BINARY TREE REPRESENTATION

Array representation of a Binary Tree

An array can be used to store the nodes of a binary tree. The nodes stored in an array are accessible sequentially. The maximum number of nodes is specified by **MAXSIZE**. In C, arrays start with index **0** to (**MAXSIZE – 1**). Here, numbering of binary tree nodes start from **0**.

The root node is always at index **0**. Then, in successive memory locations the left child and right child are stored. Consider a binary tree with only three nodes as shown. Let **BT** denote the binary tree.

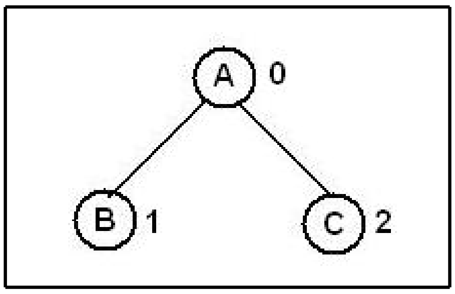


Fig. 7

The array representation of this binary tree is as follows :

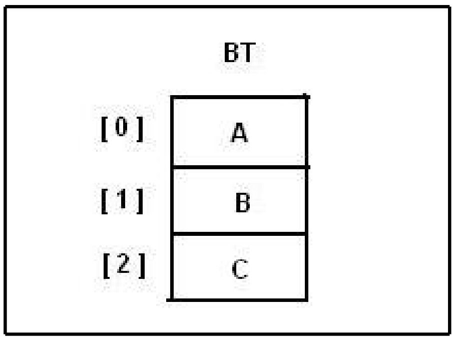


Fig. 8

Here, A is the father of B and C, B is the left child of A and C is the right child of A. Let us extend the above tree by one more level as shown below :

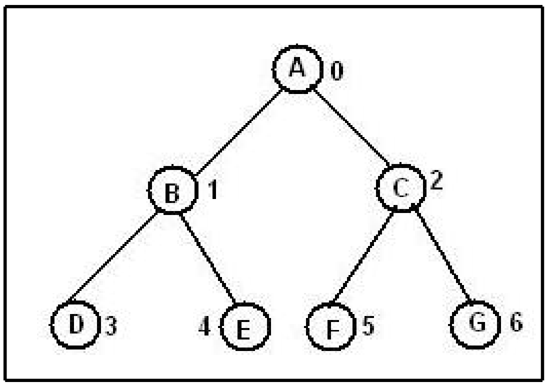


Fig. 9

The array representation of this binary tree is as follows :

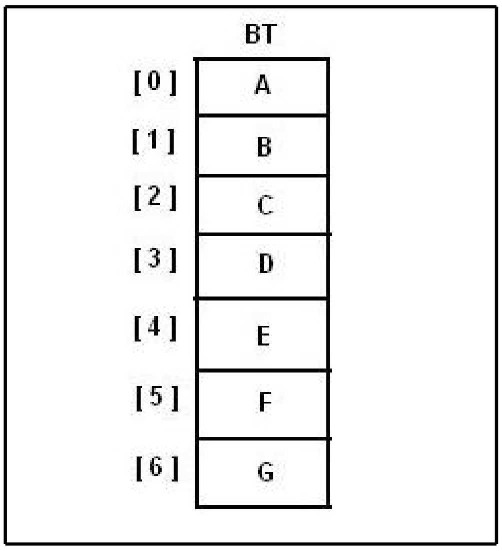


Fig. 10

How to identify the father, the left child and the right child of an random node in such representation ? It is very simple, to identify the father and the children of a node. For any node *n*, $0 \leq n \leq$ (MAXSIZE – 1), then we have

1. **father (n) :** The father of node having index **n** is at floor ((n - 1) / 2), if *n* is not equal to 0. If *n* = 0, then it is the root node and has no father.
   Example : Consider a node numbered 3 (i.e. D). The father of D, no doubt, is B whose index is 1.
   This is obtained from
   floor ((3 – 1) / 2 = floor (2/2)
   = 1
2. **Lchild (*n*) :** The left child of node numbered *n* is at (2*n* + 1). For example, in the above binary tree
   (a) Lchild (A) = Lchild(0)
   $= 2 \times 0 + 1$
   = 1
   i.e., the node with index 1 which is nothing but B.
   (b) Lchild (C) = Lchild(2)
   $= 2 \times 2 + 1$
   = 5
   i.e., the node with index 5 which is nothing but F.

3. **Rchild (*n*) :** The right child of node numbered *n* is index (2*n* + 2). For example, in the above binary tree,
   (a) Rchild(A) = Rchild(0)
   $= 2 \times 0 + 1$
   = 2
   i.e., the node with index 2 which is nothing but C.
   (b) Rchild(B) = Rchild(1)
   $= 2 \times 1 + 2$
   = 4
   i.e. the node with index 4 which is nothing but E.

4. **Siblings :** If the left child at index *n* is given then its right sibling (or brother) is at (*n* + 1). And, Similarly, if the right child at index *n* is given, then its left sibling is at (*n* - 1). For example, the right sibling is at (n – 1) of node indexed 4 is at index 5 in an array representation.

When a binary tree is represented using arrays, one array **A** stores the data fields of the trees. For this, numbers are given to each node starting from the root node – 0 to root node, 1 to the left node of the first level, then 2 to the second node from left of the first level and so on. In other words, the nodes are numbered from left to right level by level from top to bottom. Fig. ( ) shows the numbers given to each node in the tree. Note that while numbering the nodes of the tree, empty nodes are also taken into account.

Linked representation of a Binary tree :

Binary tree can be represented either using array representation or using a linked list representation. The basic component to be represented in a binary tree is a node. The node consists of three fields such as

- Data
- Left child
- Right child

The data field holds the value to be given. The left child is a link field which contains the address of its left node and the right child contains the address of the right node. Fig. (11) shows a structure of a node.

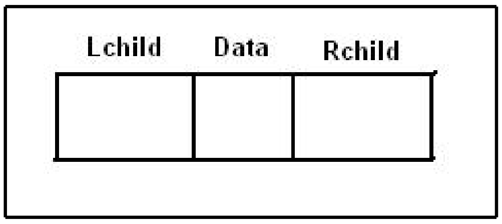


Fig. (11) A node structure of a binary tree

The logical representation of the node in C is given below :

```
struct tree
{
  char info;
  struct node *left;
  struct node *right;
};
struct tree *ptr;
```

Consider the following binary tree :

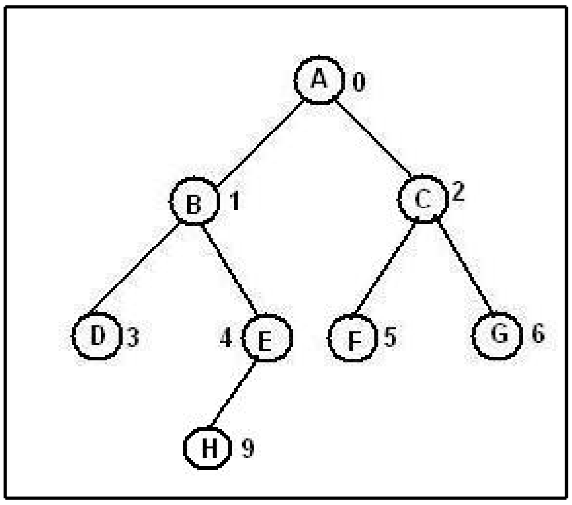


Fig. (12) A Binary tree

And, its linked representation is shown in Fig. (13). In the above binary tree all the data items are of type char.

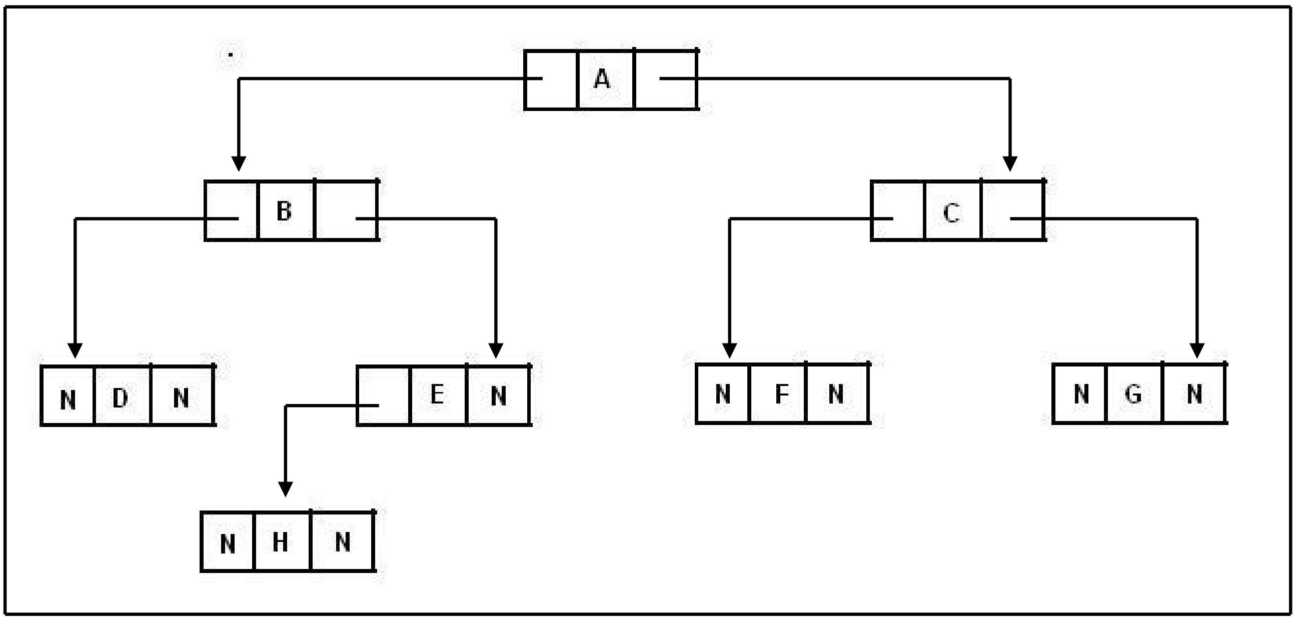


Fig. (13) : Linked representation of a Binary Tree

In some applications, it is necessary to include the **father** (or parent) field. In such situation, one more field to represent the **father** of a node is inserted into the structure definition of a binary tree node. i.e.,

```
Struct tree
{
  char data;
  struct node *father;
  struct node *lchild;
  struct node *rchild;
} ;

Struct tree *ptr;
```

A binary tree contains one root node and some non-terminal and terminal nodes (leaves). It is clear from the observation of a binary tree that the non-terminal nodes have their left child and right child nodes. But, the terminal nodes have no left child and right child nodes. Their **lchild** and **rchild** pointer are set to **NULL**. Here, the non-terminal nodes are called **internal nodes** and terminal nodes are called **external nodes**. In a specific application user may maintain two sets of nodes for both internal nodes and external nodes.

OPERATIONS ON BINARY TREES

The basic operations to be performed on a binary tree are listed in Table (1).

S.No.	Operation	Description
1.	Create	It create an empty binary tree.
2.	MakeBT	It creates a new binary tree having a single node with data field set to some value.
3.	EmptyBT	It returns true if the binary tree is empty. Otherwise if returns false.
4.	Lchild	It returns a pointer to the left child of the node. If the node has no left child, it returns a null pointer.
5.	Rchild	It returns a pointer to the right child of the node. If the node has no right child, a returns a null pointer.
6.	Father	It returns a pointer to the father of the node. Otherwise returns the null pointer.
7.	Brother (Sibling)	It returns a pointer to the brother of the node. Otherwise returns the null pointer.
8.	Data	It returns the contents of the node.

Apart from these primitive operations, other operations that can be applied to the binary tree are :

1. Tree traversal
2. Insertion of nodes.
3. Deletion of nodes.
4. Searching for the node
5. Copying the binary tree.

BINARY TREE TRAVERSALS

(A) Recursive Traversals

The traversal of a binary tree involves visiting each node in the tree exactly one. In several applications involving a binary tree we need to go to each node in the tree systematically. In a linear list, nodes can be visited in a systematic manner from beginning to end. However, such an order is not possible while traversing a tree. There are many applications that essentially require traversal of binary trees.

There are three methods commonly used for binary tree traversal. These methods are

(a) Pre-order traversal
(b) In-order traversal
(c) Post-order traversal.

The methods differ primarily in the order in which they visit the root node the nodes in the left sub-tree and the nodes in the right sub-tree. Note that a binary tree is of recursive nature, i.e. each sub-tree is a binary tree itself. Hence the functions used to traverse a tree using these methods can use recursion. Consider the following binary Tree :

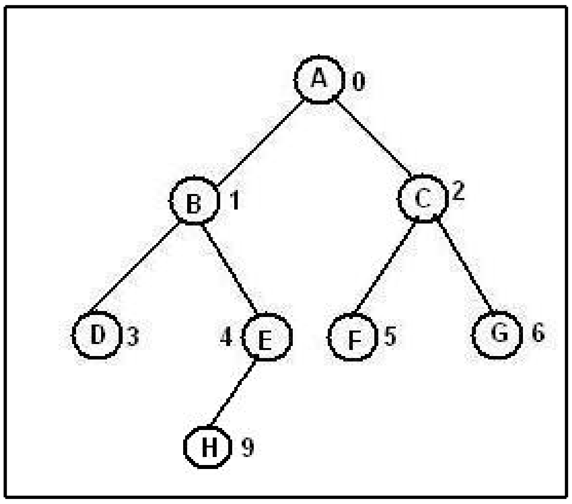


Fig. (14) A Binary tree

(I) Pre-order Traversal :

The algorithm to traverse a non-empty binary tree in **pre-order**, is given below :

(1) Visit the root **R**
(2) Traverse the left sub-tree of root **R** in **pre-order**
(3) Traverse the right sub-tree of root **R** in **pre-order.**

i.e. in a pre-order traversal the root node R is visited (or processed) before traversing its left and right sub-trees. The pre-order notion is recursive in nature, so even within the left sub-tree and right sub-tree the above three steps are followed. The function for pre-order traversal of binary tree shown in Fig. (14) is

```
void preorder(struct tree *root)
{
  if(root!=NULL)
  {
   printf("%c\t",root->info);
   preorder(root->left);
   preorder(root->right);
  }
}
```

After the pre-order traversal of Fig. (14), we get

A B D E H C F G

(II) In-Order Traversal

The algorithm to traverse a non-empty binary tree in **in-order**, is given below :

(1) Traverse the left sub-tree of root **R** in **in-order**.
(2) Visit the root **R**.
(3) Traverse the right sub-tree of root **R** in **in-order**.

i.e., in an **in-order** traversal, the left sub-tree is traversed recursively in in-order before visiting the root node. After visiting the root node, the right sub-tree is taken up and it is traversed recursively again in **in-order**. The function for **in-order** traversal of binary tree shown in Fig. (14) is

```
void inorder(struct tree *root)
{
  if(root!=NULL)
  {
   inorder(root->left);
   printf("%c\t",root->info);
   inorder(root->right);
  }
}
```

The function **inorder( )** is called to traverse the tree in **in-order** traversal. This function receives only one parameter ***root** as the address of the root node. Then a condition is checked whether the pointer if **NULL**. If the pointer is not **NULL**, then a recursive call is made first for the *left child* and then for the *right child*. The values passed are the addresses of the left and right children that are present in the pointers **left** and **right** respectively. In-between these two calls the data of the current node is printed by **root->info**.

After the **in-order** traversal of Fig. (14), we get

D B H E A F C G

(III) Post-order Traversal :

The algorithm to traverse a non-empty binary tree in **post-order**, is given below :

(1) Traverse the left sub-tree of root **R** in **in-order**.
(2) Traverse the right sub-tree of root **R** in **in-order**.
(3) Visit the root **R**.

i.e., in a post-order traversal, the left and the right sub-tree are recursively processed before visiting the root. The left sub-tree is taken up first and is traversed in **post-order**. Then the right sub-tree is taken up and is traversed in **post-order**. Finally, the data of the root node is displayed. The function for **in-order** traversal of binary tree shown in Fig. (14) is

```
void postorder(struct tree *root)
{
  if(root!=NULL)
  {
   postorder(root->left);
   postorder(root->right);
   printf("%c\t",root->info);
  }
}
```

After the **post-order** traversal of Fig. (14), we get

```
D    H    E    B    F    G    C    A
```

Program 1 : Static Implementation of binary tree using arrays by giving index numbers to each node.

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
struct tree
{
 struct tree *left;
 char info;
 struct tree *right;
};

struct tree * insert(int);
void inorder(struct tree *);
void preorder(struct tree *);
void postorder(struct tree *);

char A[ ]={'A','B','C','D','E','F','G','\0','\0','H','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0'};

void main()
{
 struct tree *root;
 clrscr();
 root=insert(0);
 printf("\nPre-order traversal : \n");
 preorder(root);
 printf("In-order traversal : \n");
 inorder(root);
 printf("\nPost-order traversal : \n");
 postorder(root);
 getch();
}

struct tree *insert(int n)
{
 struct tree *ptr=NULL;
 if(A[n]!='\0')
 {
  ptr=(struct tree *)malloc(sizeof(struct tree));
  ptr->left=insert(2*n+1);
  ptr->info=A[n];
  ptr->right=insert(2*n+2);
}
return ptr;
}

void preorder(struct tree *root)
{
  if(root!=NULL)
  {
   printf("%c\t",root->info);
   preorder(root->left);
   preorder(root->right);
  }
}

void inorder(struct tree *root)
{
  if(root!=NULL)
  {
   inorder(root->left);
   printf("%c\t",root->info);
   inorder(root->right);
  }
}

void postorder(struct tree *root)
{
  if(root!=NULL)
  {
   postorder(root->left);
   postorder(root->right);
   printf("%c\t",root->info);
  }
}
```

OUTPUT

```
Pre-order traversal :
A    B    D    E    H    C    F    G
In-order traversal :
D    B    H    E    A    F    C    G
Post-order traversal :
D    H    E    B    F    G    C    A
```

Example 1 : Consider the binary Tree T shown in Fig. (15). Traverse it using,

(a) Preorder Traversal
(b) Inorder Traversal
(c) Postorder Traversal

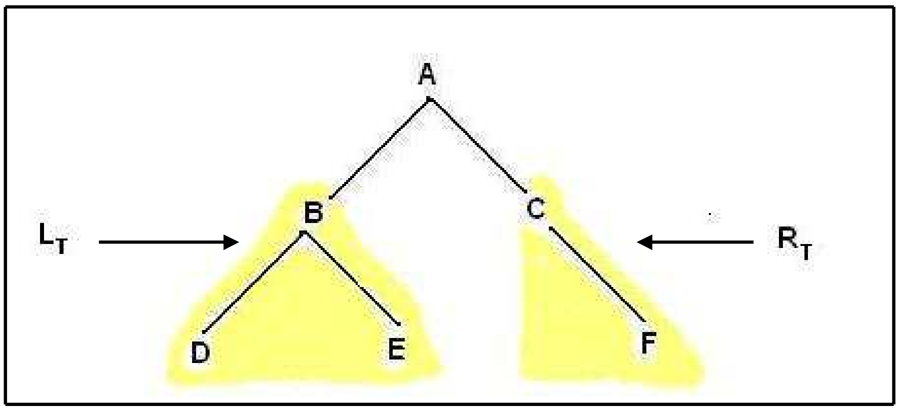


Fig. (15)

Sol.

(a) The **preorder** traversal of **T** processes **A**, traverses **L_T** and traverses **R_T**. However, the preorder traversal of **L_T** processes the root **B** and then **D** and **E**, and the preorder traversal of **R_T** processes the root **C** and then **F**. Hence,

Preorder Traversal **:** A B D E C F

(b) The **inorder** traversal of **T** traverses **L_T**, processes **A** and traverses **R_T**. However, the inorder traversal of **L_T** processes **D**, **B** and then **E**, and the inorder traversal of **R_T** processes **C** and then **F**. Hence,

Inorder Traversal **:** D B E A C F

(c) The **postorder** traversal of **T** traverses **L_T**, traverses **R_T** and then **A**. However, the postorder traversal of **L_T** processes **D**, **E** and then **B**, and the postorder traversal of **R_T** processes **F** and then **C**. Hence,

Postorder Traversal **:** D E B F C A

Example 2 : Consider the binary Tree T shown in Fig. (16). Traverse it using,

(a) Preorder Traversal
(b) Inorder Traversal
(c) Postorder Traversal

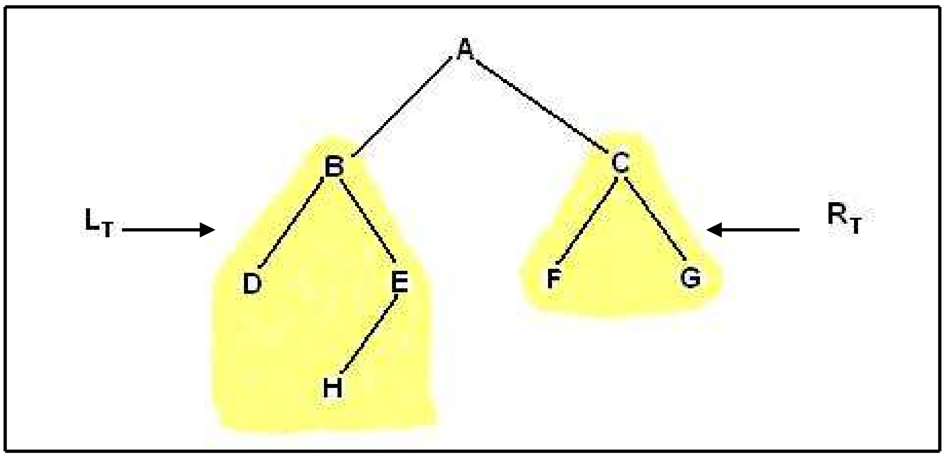


Fig. (16)

Sol.

(a) The **preorder** traversal of **T** processes **A**, traverses **L_T** and traverses **R_T**. However, the preorder traversal of **L_T** processes the root **B, D, E** & **H**, and the preorder traversal of **R_T** processes the root **C, F & G**. Hence,

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

(b) The **inorder** traversal of **T** traverses **L_T**, processes **A** and traverses **R_T**. However, the inorder traversal of **L_T** processes **D**, **B, H** & **E**, and the inorder traversal of **R_T** processes **F, C** & **G**. Hence,

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

(c) The **postorder** traversal of **T** traverses **L_T**, traverses **R_T** and then **A**. However, the postorder traversal of **L_T** processes **D**, **E** and then **B**, and the postorder traversal of **R_T** processes **F** and then **C**. Hence,

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

Example 3 : Consider the binary Tree T shown in Fig. (17). Traverse it using,

(a) Preorder Traversal
(b) Inorder Traversal
(c) Postorder Traversal

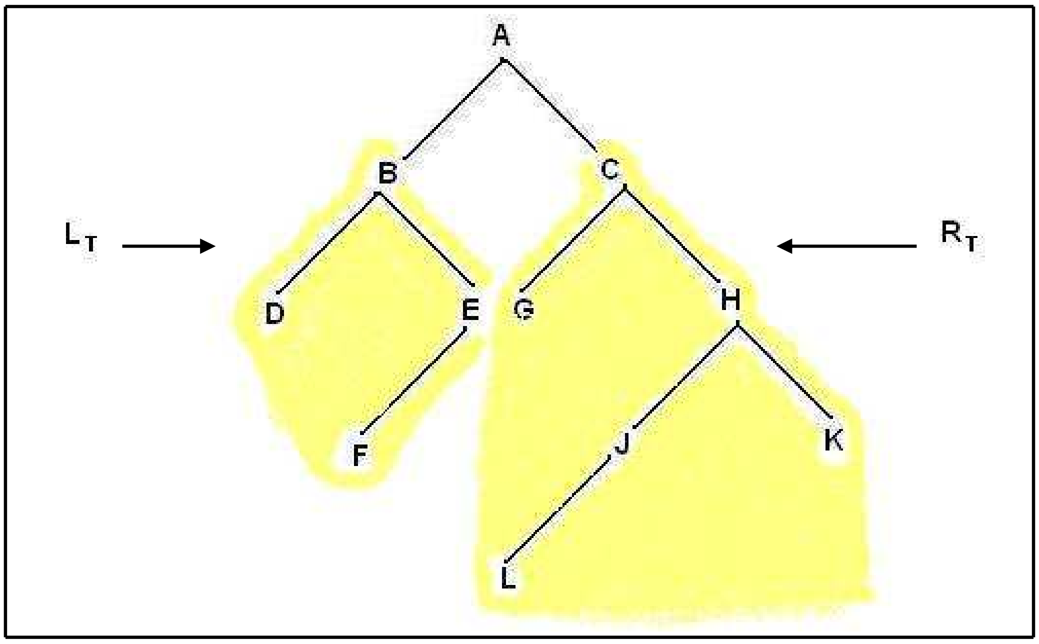


Fig. (17)

(a) The **preorder** traversal of **T** processes **A**, traverses $\mathbf{L_T}$ and traverses $\mathbf{R_T}$. However, the preorder traversal of $\mathbf{L_T}$ processes the root **B, D, E & F** and the preorder traversal of $\mathbf{R_T}$ processes the root **C, G, H, J, L** & **K**. Hence,

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

(b) The **inorder** traversal of **T** traverses $\mathbf{L_T}$, processes **A** and traverses $\mathbf{R_T}$. However, the inorder traversal of $\mathbf{L_T}$ processes **D**, **B, F** & **E**, and the inorder traversal of $\mathbf{R_T}$ processes **G, C, L, J, H** & **K**. Hence,

Inorder Traversal **:** D B F E A G C L J H K

(c) The **postorder** traversal of **T** traverses $\mathbf{L_T}$, traverses $\mathbf{R_T}$ and then **A**. However, the postorder traversal of $\mathbf{L_T}$ processes **D, F, E** & **B**, and the postorder traversal of $\mathbf{R_T}$ processes **G, L, J, K, H & C**. Hence,

Postorder Traversal **:** D F E B G L J K H C A

Example 4 : A binary Tree T has 9 nodes. The inorder and preorder traversals results into following sequences of nodes :

Inorder : E A C K F H D B G

Preorder : F A E K C D H G B

Draw the tree T.

Sol. The tree T is drawn from its root downward as follows :

(a) The root of T is obtained by choosing the first node in its preorder. Thus F is the root of T.

(b) The left child of the node F is obtained as follows. First use the inorder of T to find the nodes in the left subtree L_T of F. Thus L_T consists of the nodes E, A, C and K. Then the left child of F is obtained by choosing the first node in the preorder of L_T (which appears in the preorder of T). Thus A is the left son of F.

(c) Similarly, the right subtree T2 of F consists of the nodes H, D, B & G, and D is the root of R_T. Thus, D is the right son of F.
Repeating the above process with each node, we finally obtain the required tree in Fig.(18).

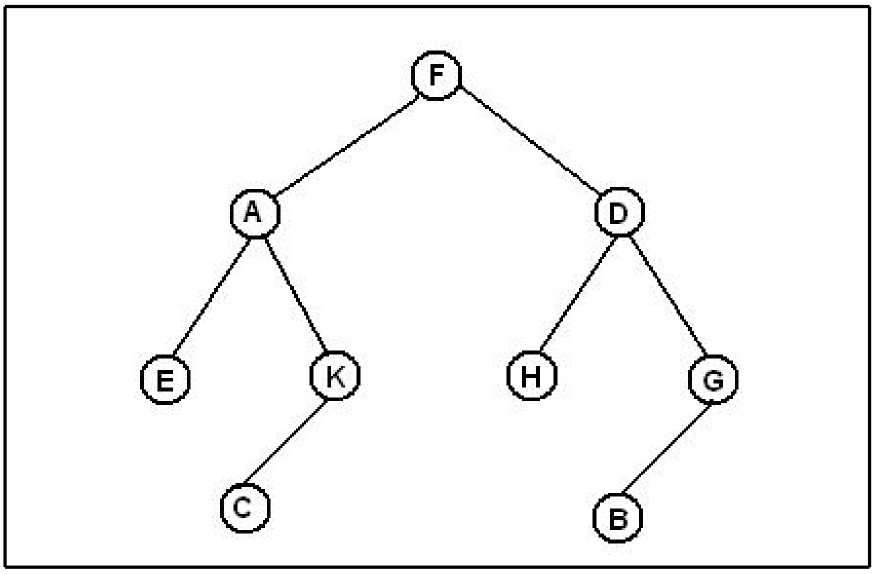


Fig. (19)

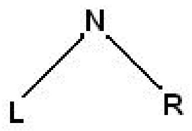
Example 5 : The following sequence gives the preorder and inorder of the Binary tree T :

Preorder : A B D G C E H I F

Inorder : D G B A H E I C F

Draw the binary tree T.

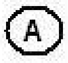
Sol. From the preorder we come to know that A is the root as in the preorder , we follow (NLR) sequence.



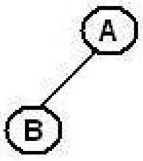
Therefore, from inorder, now we can say that the left subtree consists of elements DGB and right subtree consists of elements H E I C F.

D G B	A	H E I C F
Left	Node	Right

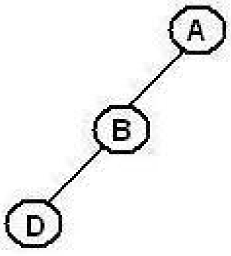
Now, we trace each node into tree step by step. Start with the root node A,



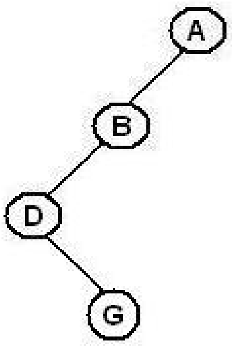
The next element in the preorder is B i.e., next Node on the left tree i.e., to be traversed therefore it comes on the left of A.



The next element is D to be fixed at proper position, according to the preorder notation the next element is D and from Inorder expression, the leftmost is D. Therefore, we put D at left of B.



Then only left element is G in the left subtree. Therefore, we think for G. As in the preorder, the (NLR) is BDG, it means G is on the right of D.

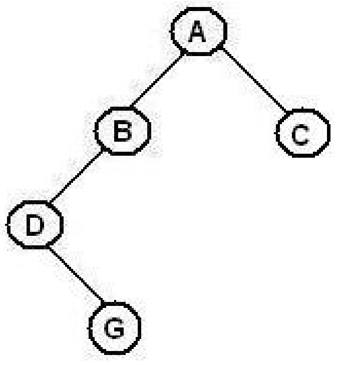


Now , we are left with the :

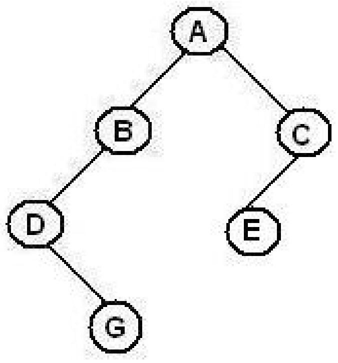
Preorder : C E H I F

Inorder : H E I C F

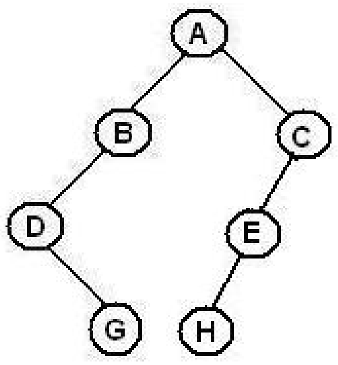
According to preorder traversal, C is the first node on the right side of tree after the Root node A.



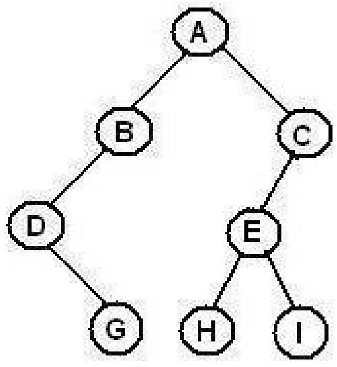
The next element is E in the preorder i.e., when we have to left from C then the next node to be printed is E on the left of C.



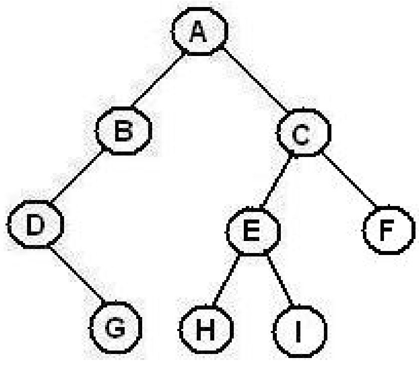
Next element is H in the preorder and H is the leftmost in the remaining Inorder expression. So, it will go on the left of E and the tree is :



The next element is I in the preorder expression and before C in the Inorder expression therefore it will go to the right of E.



The next element is F both in Inorder and Preorder which is the rightmost element. Therefore, it will be in the right of C.



To cross check the result, Traverse the tree and we get,

```
Preorder  : A  B  D  G  C  E  H  I  F
Inorder   : D  G  B  A  H  E  I  C  F
Postorder : G  D  B  H  I  E  F  C  A
```

Non-Recursive Traversals using Stacks :

Suppose a binary tree T is maintained in memory by some linked representation :

TREE (INFO,LEFT,RIGHT,ROOT)

The implementation of the three standard traversals of **T**, is done by means of non recursive procedures using stacks.

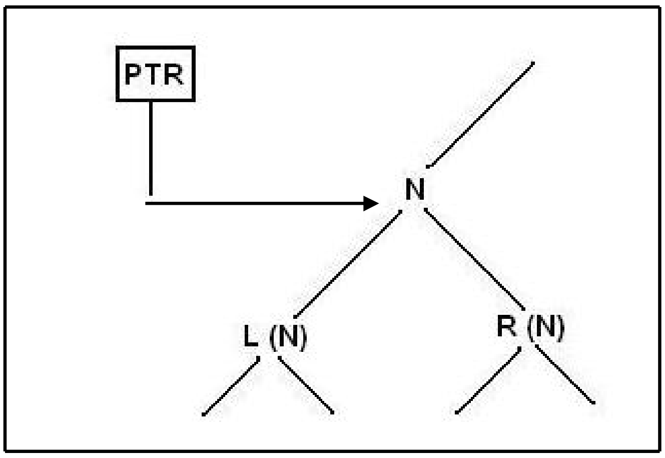


Fig. (20)

Preorder Traversal

The **Preorder** traversal algorithm uses a variable PTR (pointer) which will contain the location of the node N currently being scanned. This is shown in Fig. (20), where L (N) denotes the left child of node N and R(N) denotes the right child. The algorithm also uses an array STACK, which will hold the address of nodes for future processing.

Algorithm

Initially push NULL onto stack and then set PTR:=ROOT. Then repeat the following steps until PTR=NULL, or equivalently, while PTR ≠ NULL.

(d) Proceed down the left-most path rooted at PTR, processing each node N on the path and pushing each right child R(N), if any, onto STACK. The traversing ends after a node N with no left child L (N) is processed. (Thus PTR is updated using the assignment PTR := LEFT[PTR], and the traversing stops when LEFT[PTR] = NULL).

(b) [Backtracking] Pop and assign to PTR the top element on STACK. If PTR ≠ NULL, then return to step (a), other Exit.
(We note that initial element NULL on STACK is used as a sentinel.).

Inorder Traversal

The **Inorder** traversal algorithm also uses a variable pointer PTR, which will contain the location of the node N currently being scanned, an array STACK, which will hold the addresses of nodes for future processing. In fact, with this algorithm, a node is processed only when it is popped from STACK.

Algorithm

Initially push NULL onto STACK (for a sentinel) and then set PTR := ROOT. Then repeat the following steps until NULL is popped from STACK.

(a) Proceed down the left-most path rooted at PTR, pushing each node N onto STACK and stopping when a node N with no left child is pushed onto STACK.
(b) [Backtracking] Pop and process the nodes on STACK. If NULL is popped, then Exit. If a node N with a right child R(N) is processed, set PTR = R(N) (by assigning PTR := RIGHT[PTR] and return to step (a).
We emphasize that a node N is processed only when it is popped from STACK.

Postorder traversal

The **Postorder** traversal algorithm is more complicated than the preceding two algorithms, because here we may have to save a node N in two different situations. We distinguish between the two cases by pushing either N or its negative, – N, onto STACK. Again, a variable PTR (pointer) is used which contains the location of the node N that is currently being scanned, as in Fig. ( ).

Algorithm

Initially, push NULL onto STACK (as a sentinel) and then set PTR := ROOT. Then repeat the following steps until NULL is popped from STACK.

(a) Proceed down the left-most path rooted at PTR. At each node N of the path, push N onto STACK, and , if N has a right child R(N), push – R(N) onto STACK.
(b) [Backtracking] Pop and process positive nodes on STACK. If NULL is popped, then Exit. If a negative node is popped, i.e., if PTR = –N for some node N, set PTR = N (by assigning PTR := – PTR) and return to step (a).

We emphasize that a node N is processed only when it is popped from STACK and it is positive.

BINARY SEARCH TREE

Many algorithms that use binary trees proceed in two phases. The first phase builds a binary tree and the second traverses the tree. As an example of such an algorithm, consider the following sorting methods. Given a list of numbers in an input file, we wish to print them in ascending order. As we read the numbers, they can be inserted into a binary tree such as the one shown in Fig. (21). When a number is compared with the contents of a node in the tree, a left branch is taken. If the number is smaller then the contents of the node and a right branch if it is greater or equal to the contents of the node. Thus if the input list is

20 17 6 8 10 7 18 13 12 5

Then the binary tree shown in Fig. (21) is produced.

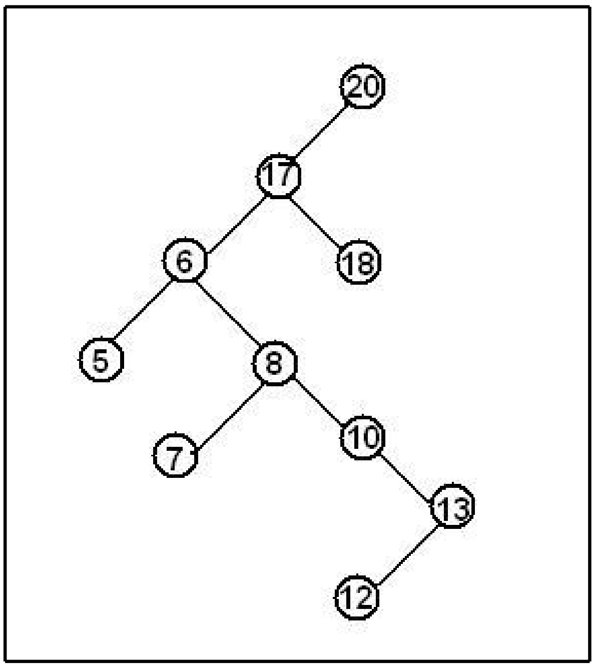


Fig. (21) : Binary search tree

Such a binary tree has the property that all the elements in the left sub-tree of a node **n** are less than the contents of **n,** and all the elements in the right sub-tree of n are greater than or equal to the contents of n.

A binary tree that has these properties is called a binary search tree. If a binary search tree is traversed in **in-order** (left, root and right) and the contents of each node are printed as the node is visited, the numbers are printed in ascending order.

B – TREE

A B-tree is a balanced N-way tree. A node of the tree contains many records or key and pointers to children. A B tree is also known as the balanced sort tree. It finds its use in external sorting. It is not a binary tree. To reduce disk accesses, several conditions of the tree must be true.

- The height of the tree must be kept to a minimum.
- There must be no empty sub-trees above the leaves of the tree.
- The leaves of the tree must all be on the same level.
- All nodes except the leaves must have some minimum number of children.

B - tree is a multi-way search tree of order n that satisfies the following conditions :

1. All the non-leaf nodes (except the root node) have at-least n/2 children and at the most n children.
2. The non-leaf root nodes may have at the most n non-empty child and at-least two child nodes.

3. A B - tree can exist with only one node i.e. the root node containing no child.
4. If a node has n children, then it must have n – 1 values. All values of a particular node are in increasing order.
5. All the values that appear on the leaf most child of a node are smaller than the first value of that node. All the values that appear on the right most child of a node are greater than the last value of that node.
6. If x and y are any two ith and (i + 1)th values of a node, where x < y, then all the values appearing on the (i + 1)th sub-tree of that node are greater than x and less than y.
7. All the leaf nodes should appear on the same level.

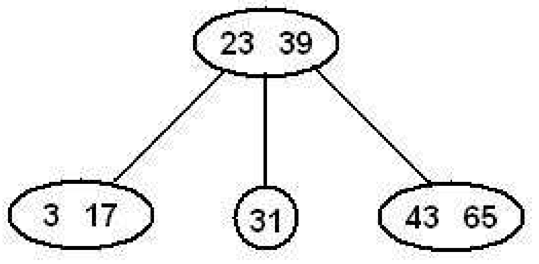


Fig. (22) : B – Tree of order 3

Fig. (22) shows a B – tree of order 3.
From Fig. (22), it can be observed that a B – tree of order 3 is a 2 – 3 tree.
The structure of a node of a B - tree is similar to the structure of the node of 2 – 3 tree.

B $^+$ TREE

One of the popular techniques for implementing indexed sequential organization is to use a variation on the basic known as B$^+$ tree.

In B$^+$ tree, all the leaves have been connected to form a linked list of the keys in sequential order. The B$^+$ tree has two parts : the indexed part is the interior node, the sequence is the leaves. Nodes **a** through **i** form the indexed part, nodes **j** through **y** form the sequence set. The linked leaves are an excellent aspect of B$^+$ tree, the keys can be accessed efficiently both directly and sequentially.

B$^+$ tree is used to provide indexed sequential file organization, the key value in the sequence set are the key values in the record collection, the key values in the indexed part exist solely for internal purposes of directing access to the sequence set.

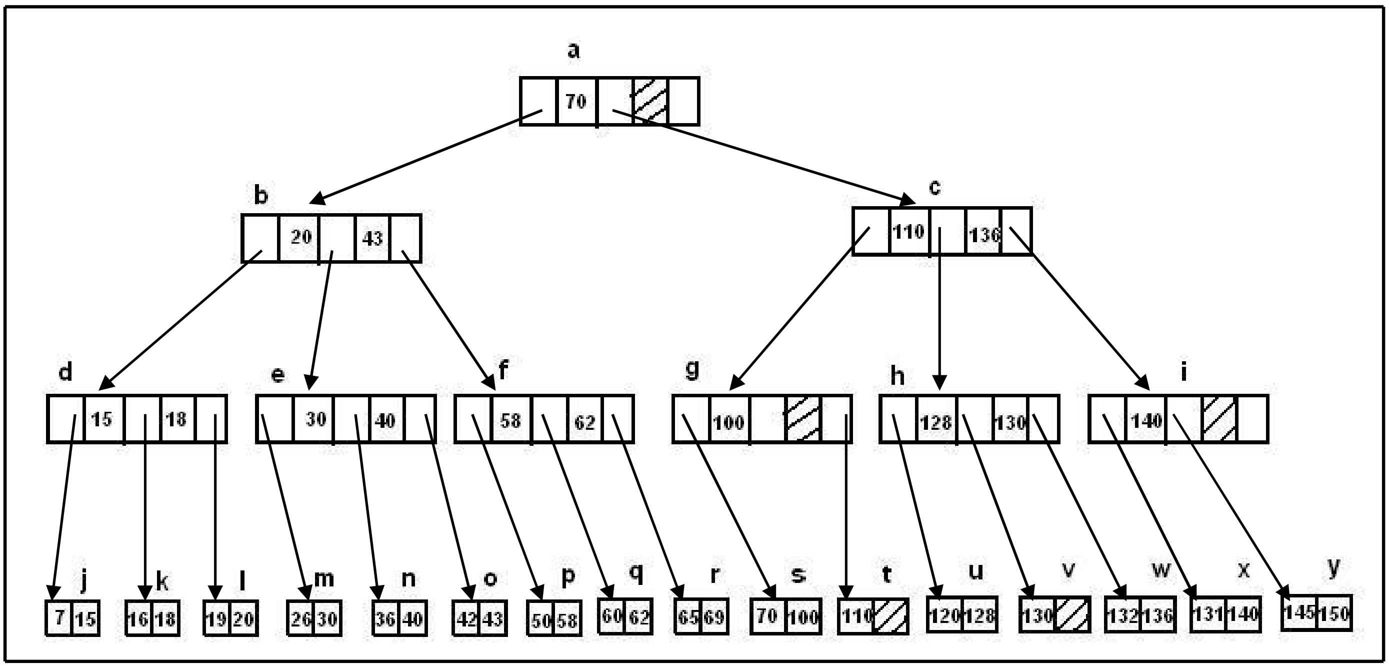


Fig. (23) : B+ Tree.

AVL TREES

Searching in a binary search tree is efficient if the heights of both left and right sub-trees of any node are equal. However, frequent insertions and deletions in a BST is like to make it unbalanced. The efficiency of searching is ideal if the difference between the heights of left and right sub-tree of all the nodes in a binary search tree is at the most one. Such a binary search tree is called as balanced binary tree. It was invented in the year 1962 by two Russian Mathematicians – G. M. Adelson – Velskii and E.M. Landis. Hence such trees as AVL trees.
Fig (24) shows some examples of AVL trees.

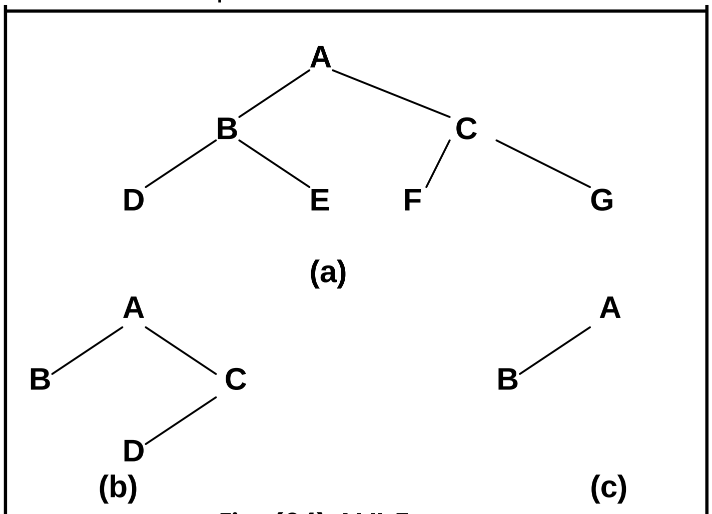


Fig. (24) AVL Trees

To represent a node of an AVL tree four fields are required, one for data, two for storing the address of left and right child and an additional field is required to hold the **balance factor**. The **balance factor** of any node is calculated by subtracting the height of the right sub-tree of the tree from the height of the left sub-tree.

THREADED BINARY TREE

In the recursive as well as non-recursive procedures for binary tree traversal, we are required to keep the pointers to the node temporarily on the stack. It is possible to write binary tree traversal procedure that does not require any pointers to the node, we put on the stack. Such procedures eliminate the overhead (time and memory) involved in initializing, **pushing** and **popping** the stack.

In order to get an idea of how such binary tree traversal procedures work, let us look at the tree shown in Fig.(25).

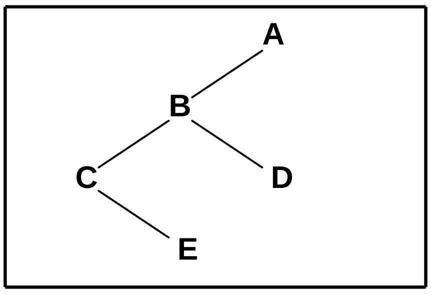


Fig. (25) Binary tree

Here, first we follow the left pointers until we reach node **C**, without, however, pushing the pointers to **A**, **B** and **C** into a stack. For in-order traversal the data for node **C** is then printed, after which **C**'s right pointer is followed to node **E**. Then the data from node **E** is printed. The next step in our in-order traversal is to go back to node **B** and print its data, However, we didn't save any pointers. But suppose that when we created the tree, we had replaced the **NULL** right pointer of node **E** with a pointer back to node **B**. We could then easily follow this pointer back to node **B.** This is shown in Fig. (26) .

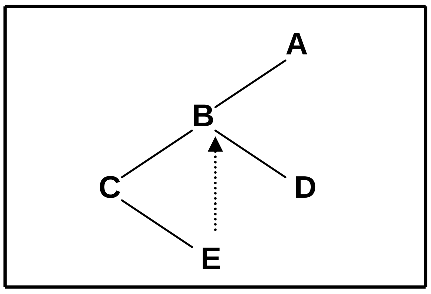


Fig. (26) Threaded Binary tree

Similarly, suppose we replace the NULL right pointer of **D** with a pointer back upto **A**, as shown in Fig. (27). Then after printing the data in **D**, we can easily jump upto **A** and print its data.

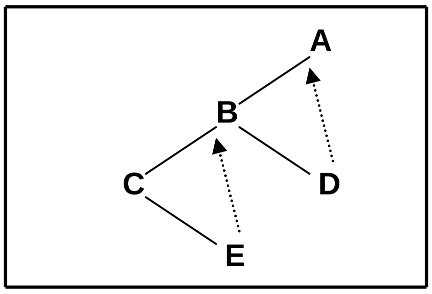


Fig. (27) Threaded Binary tree

The pointers that point in-order successor of a node are called **right threads**. Each **right thread** replaces a normal right pointer in a tree node. Likewise, we can have **left threads** that points to the in-order predecessor of a node.