

Chapter 5

Binary Trees

A data structure is said to be linear if its elements form a sequence or a linear list. Previous linear data structures that we have studied like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be non linear if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represents hierarchial relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

In this chapter in particular, we will explain special type of trees known as binary trees, which are easy to maintain in the computer.

5.1. TREES:

A tree is hierarchical collection of nodes. One of the nodes, known as the root, is at the top of the hierarchy. Each node can have at most one link coming into it. The node where the link originates is called the parent node. The root node has no parent. The links leaving a node (any number of links are allowed) point to child nodes. Trees are recursive structures. Each child node is itself the root of a subtree. At the bottom of the tree are leaf nodes, which have no children.

Trees represent a special case of more general structures known as graphs. In a graph, there is no restrictions on the number of links that can enter or leave a node, and cycles may be present in the graph. The figure 5.1.1 shows a tree and a non-tree.

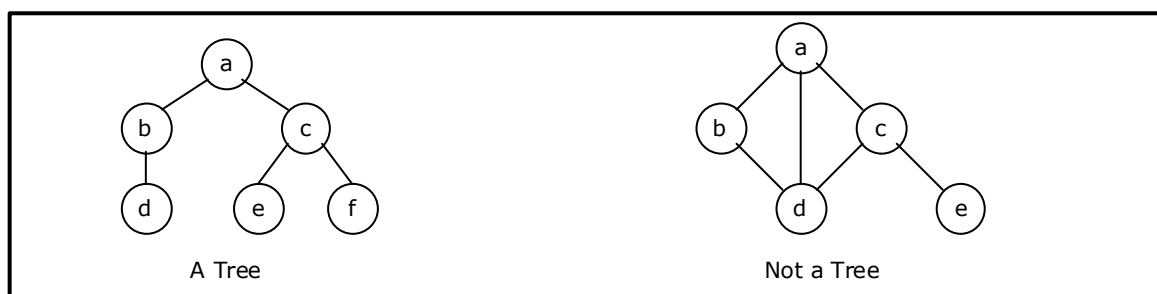


Figure 5.1.1 A Tree and a not a tree

In a tree data structure, there is no distinction between the various children of a node i.e., none is the "first child" or "last child". A tree in which such distinctions are made is called an **ordered tree**, and data structures built on them are called **ordered tree data structures**. Ordered trees are by far the commonest form of tree data structure.

5.2. BINARY TREE:

In general, tree nodes can have any number of children. In a binary tree, each node can have at most two children. A binary tree is either **empty** or consists of a node called the **root** together with two binary trees called the **left subtree** and the **right subtree**.

A tree with no nodes is called as a **null** tree. A binary tree is shown in figure 5.2.1.

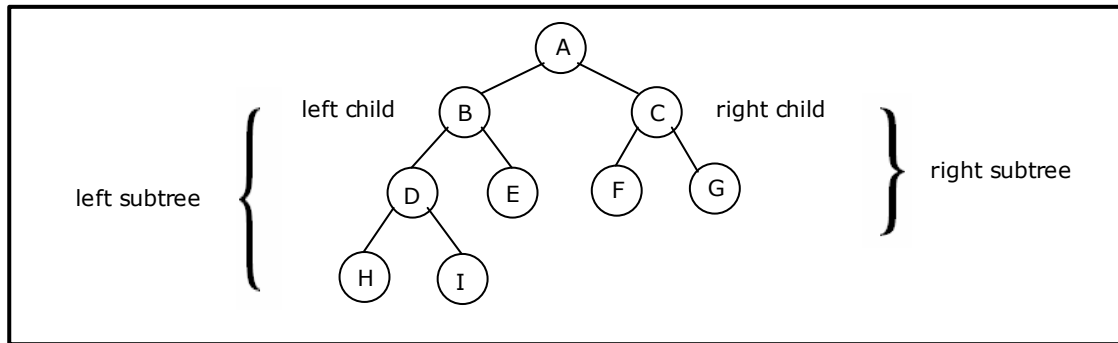


Figure 5.2.1. Binary Tree

Binary trees are easy to implement because they have a small, fixed number of child links. Because of this characteristic, binary trees are the most common types of trees and form the basis of many important data structures.

Tree Terminology:

Leaf node

A node with no children is called a *leaf* (or *external node*). A node which is not a leaf is called an *internal node*.

Path

A sequence of nodes n_1, n_2, \dots, n_k , such that n_i is the parent of n_{i+1} for $i = 1, 2, \dots, k - 1$. The length of a path is 1 less than the number of nodes on the path. Thus there is a path of length zero from a node to itself.

For the tree shown in figure 5.2.1, the path between A and I is A, B, D, I.

Siblings

The children of the same parent are called siblings.

For the tree shown in figure 5.2.1, F and G are the siblings of the parent node C and H and I are the siblings of the parent node D.

Ancestor and Descendent

If there is a path from node A to node B, then A is called an ancestor of B and B is called a descendent of A.

Subtree

Any node of a tree, with all of its descendants is a subtree.

Level

The level of the node refers to its distance from the root. The root of the tree has level 0, and the level of any other node in the tree is one more than the level of its parent. For example, in the binary tree of Figure 5.2.1 node F is at level 2 and node H is at level 3. *The maximum number of nodes at any level is 2^n .*

Height

The maximum level in a tree determines its height. The height of a node in a tree is the length of a longest path from the node to a leaf. The term depth is also used to denote height of the tree. The height of the tree of Figure 5.2.1 is 3.

Depth

The depth of a node is the number of nodes along the path from the root to that node. For instance, node 'C' in figure 5.2.1 has a depth of 1.

Assigning level numbers and Numbering of nodes for a binary tree:

The nodes of a binary tree can be numbered in a natural way, level by level, left to right. The nodes of a complete binary tree can be numbered so that the root is assigned the number 1, a left child is assigned twice the number assigned its parent, and a right child is assigned one more than twice the number assigned its parent. For example, see Figure 5.2.2.

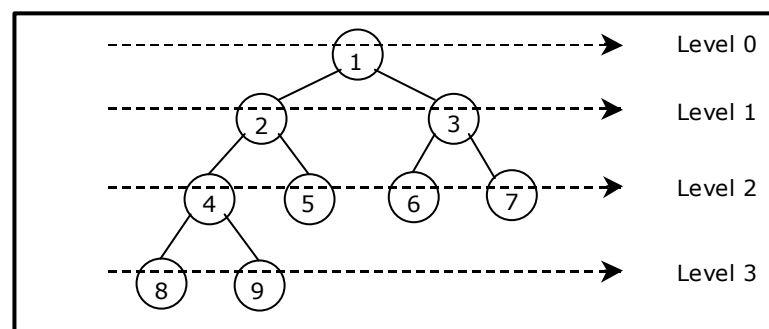


Figure 5.2.2. Level by level numbering of binary tree

Properties of binary trees:

Some of the important properties of a binary tree are as follows:

1. If h = height of a binary tree, then
 - a. Maximum number of leaves = 2^h
 - b. Maximum number of nodes = $2^{h+1} - 1$
2. If a binary tree contains m nodes at level l , it contains at most $2m$ nodes at level $l + 1$.
3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2^l nodes at level l .
4. The total number of edges in a full binary tree with n nodes is $n - 1$.

Strictly Binary tree:

If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed as strictly binary tree. Thus the tree of figure 5.2.3(a) is strictly binary. A strictly binary tree with n leaves always contains $2n - 1$ nodes.

Full Binary tree:

A full binary tree of height h has all its leaves at level h . Alternatively; All non leaf nodes of a full binary tree have two children, and the leaf nodes have no children.

A full binary tree with height h has $2^{h+1} - 1$ nodes. A full binary tree of height h is a *strictly binary tree* all of whose leaves are at level h . Figure 5.2.3(d) illustrates the full binary tree containing 15 nodes and of height 3. A full binary tree of height h contains 2^h leaves and, $2^h - 1$ non-leaf nodes.

Thus by induction, total number of nodes (tn) = $\sum_{l=0}^h 2^l = 2^{h+1} - 1$.

For example, a full binary tree of height 3 contains $2^{3+1} - 1 = 15$ nodes.

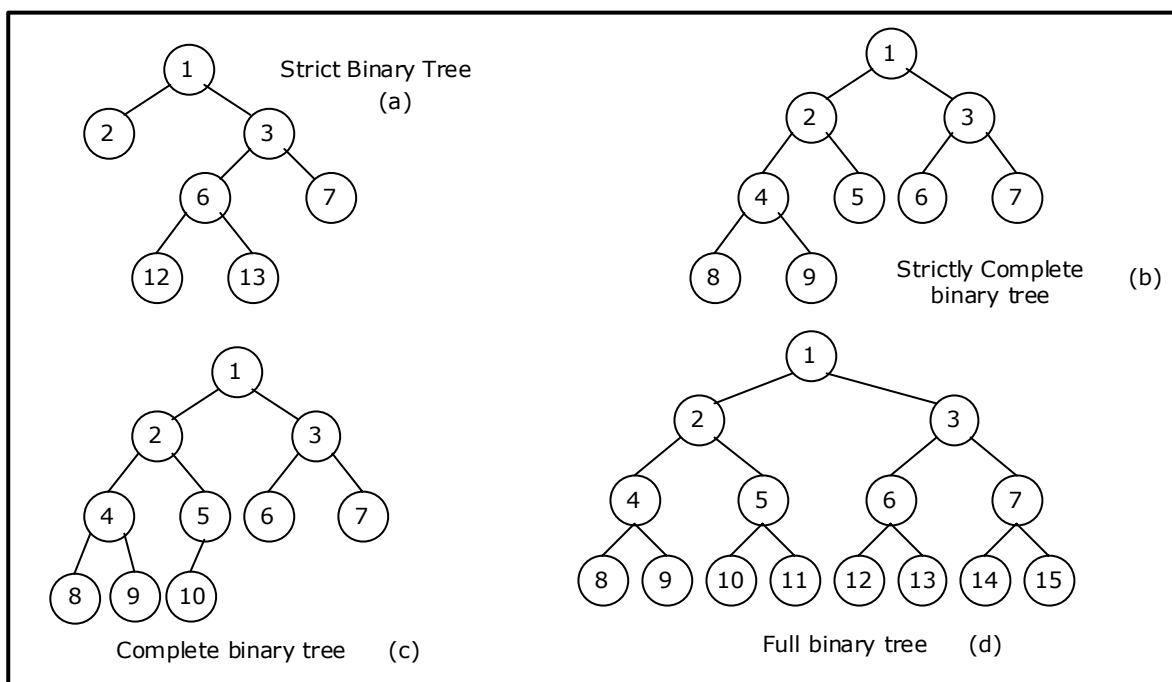


Figure 5.2.3. Examples of binary trees

Complete Binary tree:

A binary tree with n nodes is said to be **complete** if it contains all the first n nodes of the above numbering scheme. Figure 5.2.4 shows examples of complete and incomplete binary trees.

A complete binary tree of height h looks like a full binary tree down to level $h-1$, and the level h is filled from left to right.

A complete binary tree with n leaves that is *not strictly* binary has $2n$ nodes. For example, the tree of Figure 5.2.3(c) is a complete binary tree having 5 leaves and 10 nodes.

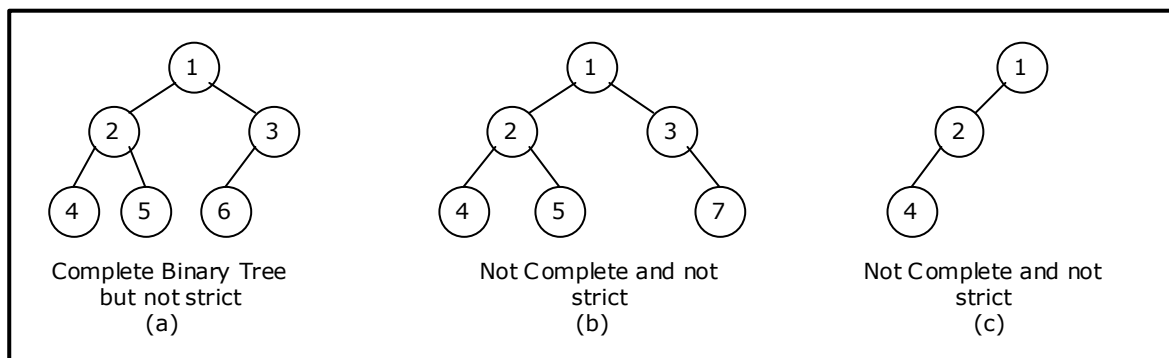


Figure 5.2.4. Examples of complete and incomplete binary trees

Internal and external nodes:

We define two terms: Internal nodes and external nodes. An internal node is a tree node having at least one-key and possibly some children. It is some times convenient to have another types of nodes, called an external node, and pretend that all null child links point to such a node. An external node doesn't exist, but serves as a conceptual place holder for nodes to be inserted.

We draw internal nodes using circles, with letters as labels. External nodes are denoted by squares. The square node version is sometimes called an extended binary tree. A binary tree with n internal nodes has $n+1$ external nodes. Figure 5.2.6 shows a sample tree illustrating both internal and external nodes.

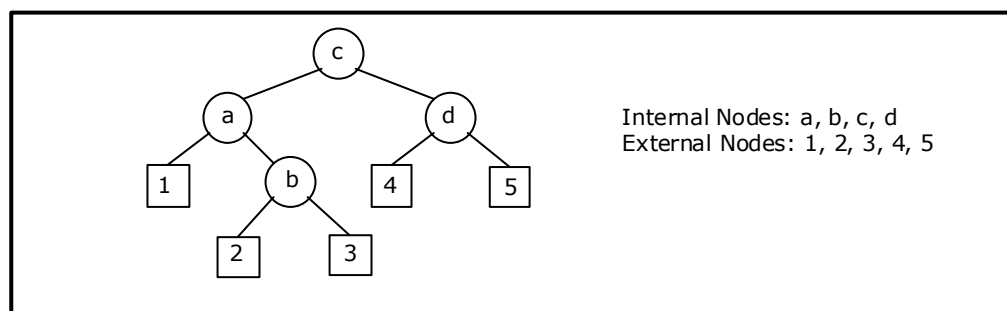


Figure 5.2.6. Internal and external nodes

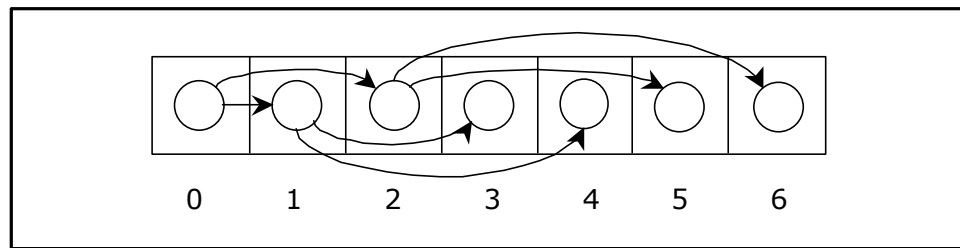
Data Structures for Binary Trees:

1. Arrays; especially suited for complete and full binary trees.
2. Pointer-based.

Array-based Implementation:

Binary trees can also be stored in arrays, and if the tree is a complete binary tree, this method wastes no space. In this compact arrangement, if a node has an index i , its children are found at indices $2i+1$ and $2i+2$, while its parent (if any) is found at index $\text{floor}((i-1)/2)$ (assuming the root of the tree stored in the array at an index zero).

This method benefits from more compact storage and better locality of reference, particularly during a preorder traversal. However, it requires contiguous memory, expensive to grow and wastes space proportional to $2^h - n$ for a tree of height h with n nodes.



Linked Representation (Pointer based):

Array representation is good for complete binary tree, but it is wasteful for many other binary trees. The representation suffers from insertion and deletion of node from the middle of the tree, as it requires the movement of potentially many nodes to reflect the change in level number of this node. To overcome this difficulty we represent the binary tree in linked representation.

In linked representation each node in a binary has three fields, the left child field denoted as *LeftChild*, data field denoted as *data* and the right child field denoted as *RightChild*. If any sub-tree is empty then the corresponding pointer's LeftChild and RightChild will store a NULL value. If the tree itself is empty the root pointer will store a NULL value.

The advantage of using linked representation of binary tree is that:

- Insertion and deletion involve no data movement and no movement of nodes except the rearrangement of pointers.

The disadvantages of linked representation of binary tree includes:

- Given a node structure, it is difficult to determine its parent node.
- Memory spaces are wasted for storing NULL pointers for the nodes, which have no subtrees.

The structure definition, node representation empty binary tree is shown in figure 5.2.6 and the linked representation of binary tree using this node structure is given in figure 5.2.7.

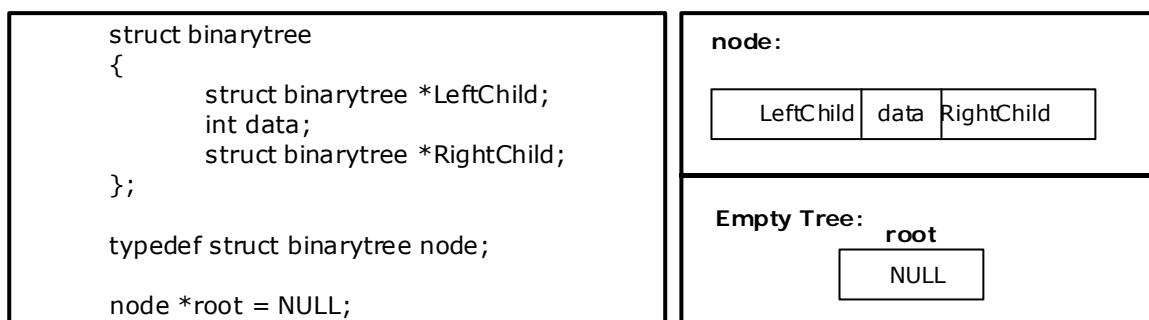


Figure 5.2.6. Structure definition, node representation and empty tree

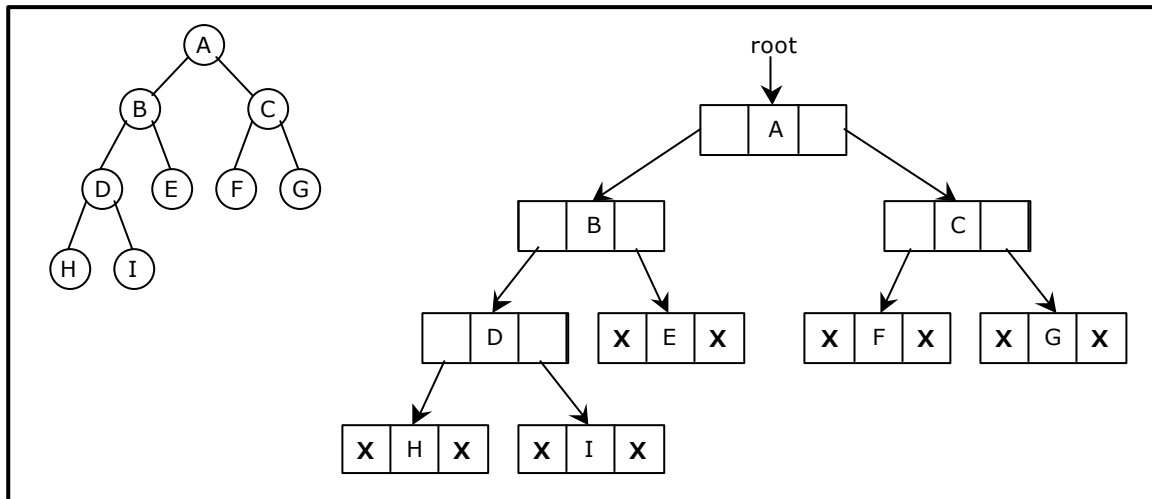


Figure 5.2.7. Linked representation for the binary tree

5.3. Binary Tree Traversal Techniques:

A tree traversal is a method of visiting every node in the tree. By visit, we mean that some type of operation is performed. For example, you may wish to print the contents of the nodes.

There are four common ways to traverse a binary tree:

1. *Preorder*
2. *Inorder*
3. *Postorder*
4. *Level order*

In the first three traversal methods, the left subtree of a node is traversed before the right subtree. The difference among them comes from the difference in the time at which a root node is visited.

5.3.1. Recursive Traversal Algorithms:

Inorder Traversal:

In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins. The steps for traversing a binary tree in inorder traversal are:

1. Visit the left subtree, using inorder.
2. Visit the root.
3. Visit the right subtree, using inorder.

The algorithm for inorder traversal is as follows:

```
void inorder(node *root)
{
    if(root != NULL)
    {
        inorder(root->lchild);
```

```

        print root -> data;
        inorder(root->rchild);
    }
}

```

Preorder Traversal:

In a preorder traversal, each root node is visited before its left and right subtrees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:

1. Visit the root.
2. Visit the left subtree, using preorder.
3. Visit the right subtree, using preorder.

The algorithm for preorder traversal is as follows:

```

void preorder(node *root)
{
    if( root != NULL )
    {
        print root -> data;
        preorder (root -> lchild);
        preorder (root -> rchild);
    }
}

```

Postorder Traversal:

In a postorder traversal, each root is visited after its left and right subtrees have been traversed. The steps for traversing a binary tree in postorder traversal are:

1. Visit the left subtree, using postorder.
2. Visit the right subtree, using postorder
3. Visit the root.

The algorithm for postorder traversal is as follows:

```

void postorder(node *root)
{
    if( root != NULL )
    {
        postorder (root -> lchild);
        postorder (root -> rchild);
        print (root -> data);
    }
}

```

Level order Traversal:

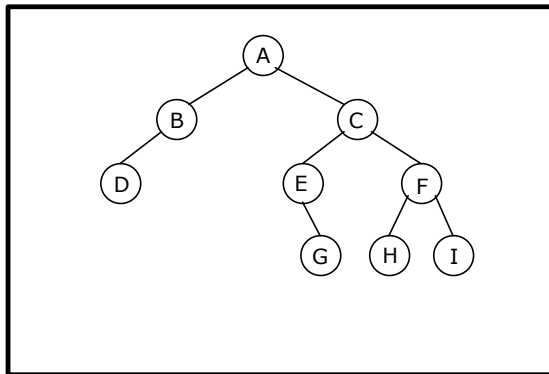
In a level order traversal, the nodes are visited level by level starting from the root, and going from left to right. The level order traversal requires a queue data structure. So, it is not possible to develop a recursive procedure to traverse the binary tree in level order. This is nothing but a breadth first search technique.

The algorithm for level order traversal is as follows:

```
void levelorder()
{
    int j;
    for(j = 0; j < ctr; j++)
    {
        if(tree[j] != NULL)
            print tree[j] -> data;
    }
}
```

Example 1:

Traverse the following binary tree in pre, post, inorder and level order.



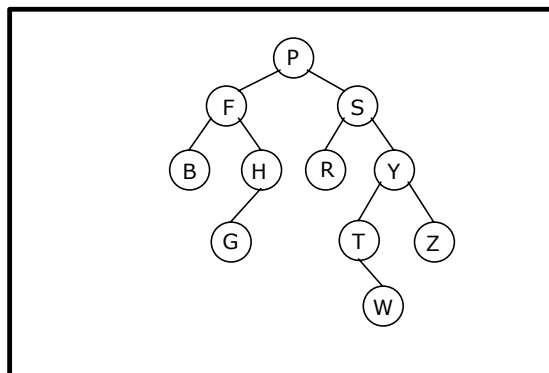
Binary Tree

- Preorder traversal yields:
A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
D, B, A, E, G, C, H, F, I
- Level order traversal yields:
A, B, C, D, E, F, G, H, I

Pre, Post, Inorder and level order Traversing

Example 2:

Traverse the following binary tree in pre, post, inorder and level order.



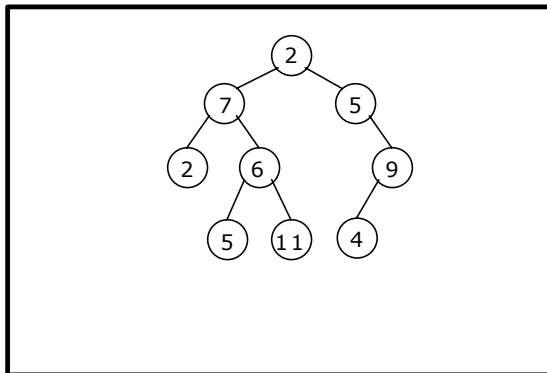
Binary Tree

- Preorder traversal yields:
P, F, B, H, G, S, R, Y, T, W, Z
- Postorder traversal yields:
B, G, H, F, R, W, T, Z, Y, S, P
- Inorder traversal yields:
B, F, G, H, P, R, S, T, W, Y, Z
- Level order traversal yields:
P, F, S, B, H, R, Y, G, T, Z, W

Pre, Post, Inorder and level order Traversing

Example 3:

Traverse the following binary tree in pre, post, inorder and level order.



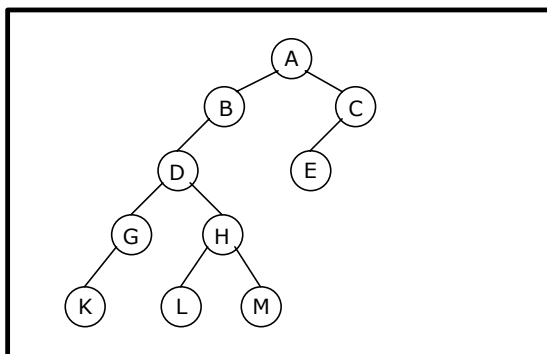
Binary Tree

- Preorder traversal yields:
2, 7, 2, 6, 5, 11, 5, 9, 4
- Postorder traversal yields:
2, 5, 11, 6, 7, 4, 9, 5, 2
- Inorder traversal yields:
2, 7, 5, 6, 11, 2, 5, 4, 9
- Level order traversal yields:
2, 7, 5, 2, 6, 9, 5, 11, 4

Pre, Post, Inorder and level order Traversing

Example 4:

Traverse the following binary tree in pre, post, inorder and level order.



Binary Tree

- Preorder traversal yields:
A, B, D, G, K, H, L, M, C, E
- Postorder traversal yields:
K, G, L, M, H, D, B, E, C, A
- Inorder traversal yields:
K, G, D, L, H, M, B, A, E, C
- Level order traversal yields:
A, B, C, D, E, G, H, K, L, M

Pre, Post, Inorder and level order Traversing

5.3.2. Building Binary Tree from Traversal Pairs:

Sometimes it is required to construct a binary tree if its traversals are known. From a single traversal it is not possible to construct unique binary tree. However any of the two traversals are given then the corresponding tree can be drawn uniquely:

- Inorder and preorder
- Inorder and postorder
- Inorder and level order

The basic principle for formulation is as follows:

If the preorder traversal is given, then the first node is the root node. If the postorder traversal is given then the last node is the root node. Once the root node is identified, all the nodes in the left sub-trees and right sub-trees of the root node can be identified using inorder.

Same technique can be applied repeatedly to form sub-trees.

It can be noted that, for the purpose mentioned, two traversal are essential out of which one should be inorder traversal and another preorder or postorder; alternatively, given preorder and postorder traversals, binary tree cannot be obtained uniquely.

Example 1:

Construct a binary tree from a given preorder and inorder sequence:

Preorder: A B D G C E H I F

Inorder: D G B A H E I C F

Solution:

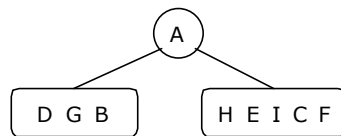
From Preorder sequence **A** B D G C E H I F, the root is: A

From Inorder sequence D G B **A** H E I C F, we get the left and right sub trees:

Left sub tree is: D G B

Right sub tree is: H E I C F

The Binary tree upto this point looks like:

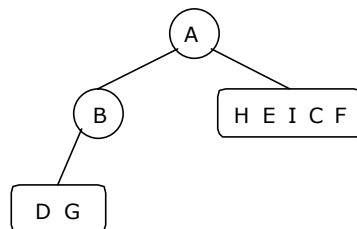


To find the root, left and right sub trees for D G B:

From the preorder sequence **B** D G, the root of tree is: B

From the inorder sequence D G **B**, we can find that D and G are to the left of B.

The Binary tree upto this point looks like:

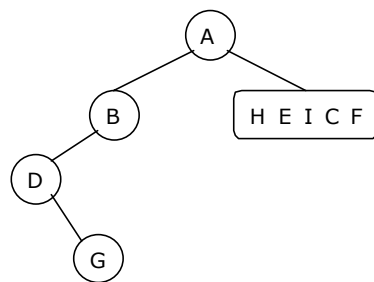


To find the root, left and right sub trees for D G:

From the preorder sequence **D** G, the root of the tree is: D

From the inorder sequence **D** G, we can find that there is no left node to D and G is at the right of D.

The Binary tree upto this point looks like:

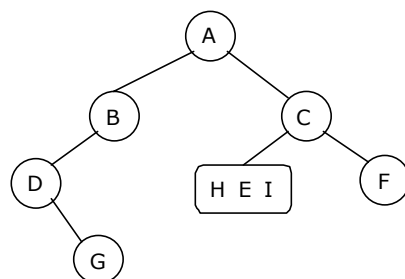


To find the root, left and right sub trees for H E I C F:

From the preorder sequence C E H I F, the root of the left sub tree is: C

From the inorder sequence H E I C F, we can find that H E I are at the left of C and F is at the right of C.

The Binary tree upto this point looks like:

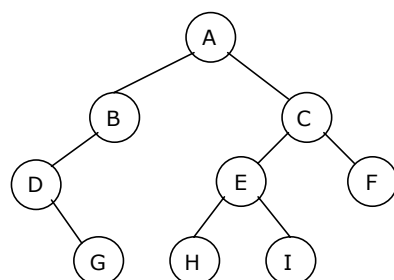


To find the root, left and right sub trees for H E I:

From the preorder sequence E H I, the root of the tree is: E

From the inorder sequence H E I, we can find that H is at the left of E and I is at the right of E.

The Binary tree upto this point looks like:



Example 2:

Construct a binary tree from a given postorder and inorder sequence:

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

Solution:

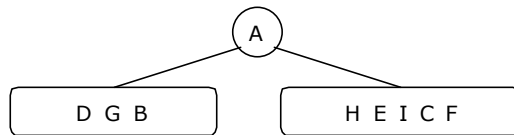
From Postorder sequence $G D B H I E F C \mathbf{A}$, the root is: A

From Inorder sequence $\underline{D G B} \mathbf{A} \underline{H E I C F}$, we get the left and right sub trees:

Left sub tree is: $D G B$

Right sub tree is: $H E I C F$

The Binary tree upto this point looks like:

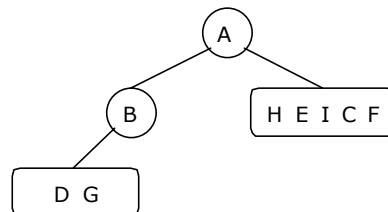


To find the root, left and right sub trees for $D G B$:

From the postorder sequence $G D B$, the root of tree is: B

From the inorder sequence $\underline{D G} \mathbf{B}$, we can find that $D G$ are to the left of B and there is no right subtree for B .

The Binary tree upto this point looks like:

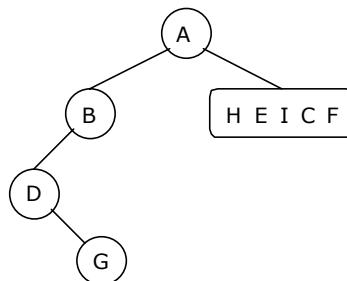


To find the root, left and right sub trees for $D G$:

From the postorder sequence $G \mathbf{D}$, the root of the tree is: D

From the inorder sequence $\mathbf{D} \underline{G}$, we can find that there is no left subtree for D and G is to the right of D .

The Binary tree upto this point looks like:

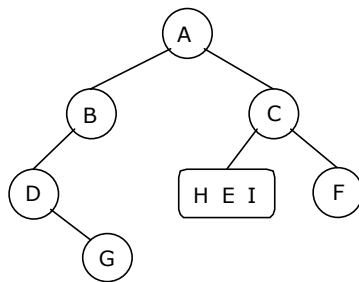


To find the root, left and right sub trees for $H E I C F$:

From the postorder sequence $H I E F \mathbf{C}$, the root of the left sub tree is: C

From the inorder sequence $\underline{H E I} \mathbf{C} \underline{F}$, we can find that $H E I$ are to the left of C and F is the right subtree for C .

The Binary tree upto this point looks like:

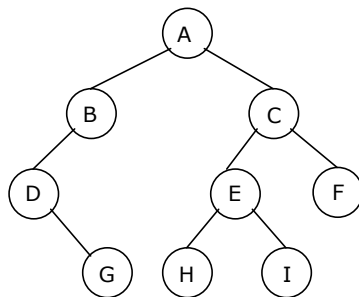


To find the root, left and right sub trees for H E I:

*From the postorder sequence **H I E**, the root of the tree is: E*

From the inorder sequence **H E I**, we can find that H is left subtree for E and I is to the right of E.

The Binary tree upto this point looks like:



Example 3:

Construct a binary tree from a given preorder and inorder sequence:

Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9

Preorder: n6 n2 n1 n4 n3 n5 n9 n7 n8

Solution:

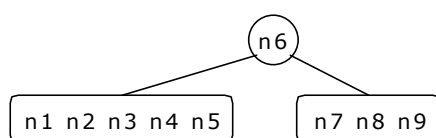
*From Preorder sequence **n6** n2 n1 n4 n3 n5 n9 n7 n8, the root is: n6*

From Inorder sequence **n1 n2 n3 n4 n5** **n6** **n7 n8 n9**, we get the left and right sub trees:

Left sub tree is: n1 n2 n3 n4 n5

Right sub tree is: n7 n8 n9

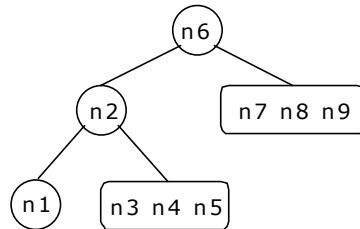
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n1 n2 n3 n4 n5:

From the preorder sequence **n2** n1 n4 n3 n5, the root of tree is: n2

From the inorder sequence n1 **n2** n3 n4 n5, we can find that n1 is to the left of n2 and n3 n4 n5 are to the right of n2. The Binary tree upto this point looks like:

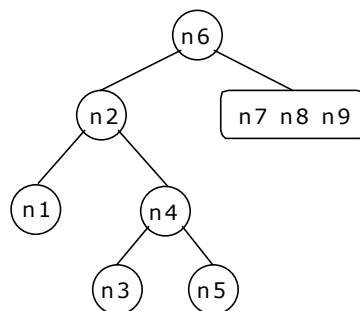


To find the root, left and right sub trees for n3 n4 n5:

From the preorder sequence **n4** n3 n5, the root of the tree is: n4

From the inorder sequence n3 **n4** n5, we can find that n3 is to the left of n4 and n5 is at the right of n4.

The Binary tree upto this point looks like:

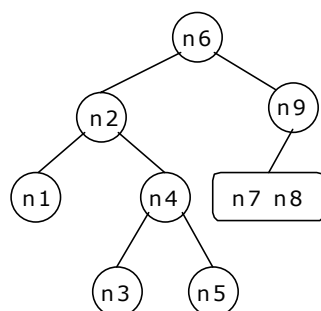


To find the root, left and right sub trees for n7 n8 n9:

From the preorder sequence **n9** n7 n8, the root of the left sub tree is: n9

From the inorder sequence n7 n8 **n9**, we can find that n7 and n8 are at the left of n9 and no right subtree of n9.

The Binary tree upto this point looks like:

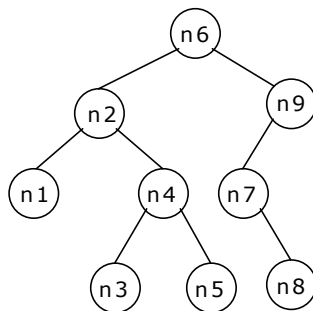


To find the root, left and right sub trees for n7 n8:

From the preorder sequence **n7** n8, the root of the tree is: n7

From the inorder sequence **n7** n8, we can find that is no left subtree for n7 and n8 is at the right of n7.

The Binary tree upto this point looks like:



Example 4:

Construct a binary tree from a given postorder and inorder sequence:

Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9

Postorder: n1 n3 n5 n4 n2 n8 n7 n9 n6

Solution:

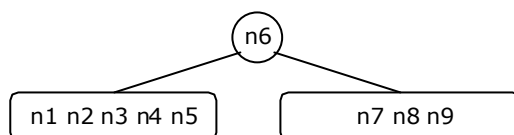
From Postorder sequence *n1 n3 n5 n4 n2 n8 n7 n9* **n6**, the root is: n6

From Inorder sequence n1 n2 n3 n4 n5 **n6** n7 n8 n9, we get the left and right sub trees:

Left sub tree is: n1 n2 n3 n4 n5

Right sub tree is: n7 n8 n9

The Binary tree upto this point looks like:

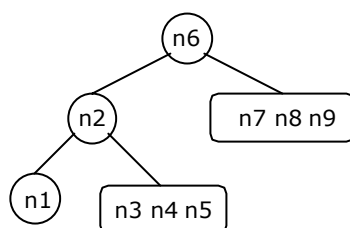


To find the root, left and right sub trees for n1 n2 n3 n4 n5:

From the postorder sequence *n1 n3 n5 n4* **n2**, the root of tree is: n2

From the inorder sequence n1 **n2** n3 n4 n5, we can find that n1 is to the left of n2 and n3 n4 n5 are to the right of n2.

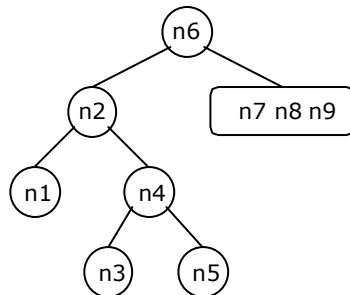
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n3 n4 n5:

From the postorder sequence $n3\ n5\ \mathbf{n4}$, the root of the tree is: $n4$

From the inorder sequence $\underline{n3}\ \mathbf{n4}\ \underline{n5}$, we can find that $n3$ is to the left of $n4$ and $n5$ is to the right of $n4$. The Binary tree upto this point looks like:

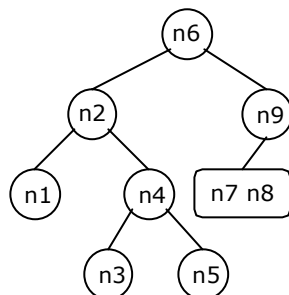


To find the root, left and right sub trees for n7 n8 and n9:

From the postorder sequence $n8\ n7\ \mathbf{n9}$, the root of the left sub tree is: $n9$

From the inorder sequence $\underline{n7}\ \underline{n8}\ \mathbf{n9}$, we can find that $n7$ and $n8$ are to the left of $n9$ and no right subtree for $n9$.

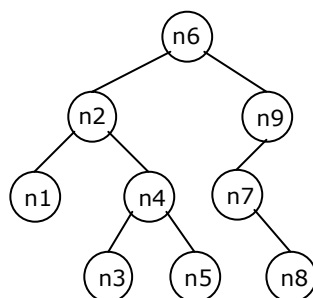
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n7 and n8:

From the postorder sequence $n8\ \mathbf{n7}$, the root of the tree is: $n7$

From the inorder sequence $\mathbf{n7}\ \underline{n8}$, we can find that there is no left subtree for $n7$ and $n8$ is to the right of $n7$. The Binary tree upto this point looks like:



5.3.3. Binary Tree Creation and Traversal Using Arrays:

This program performs the following operations:

1. Creates a complete Binary Tree
2. Inorder traversal
3. Preorder traversal
4. Postorder traversal
5. Level order traversal
6. Prints leaf nodes
7. Finds height of the tree created

```
# include <stdio.h>
# include <stdlib.h>

struct tree
{
    struct tree* lchild;
    char data[10];
    struct tree* rchild;
};

typedef struct tree node;
int ctr;
node *tree[100];

node* getnode()
{
    node *temp ;
    temp = (node*) malloc(sizeof(node));
    printf("\n Enter Data: ");
    scanf("%s",temp->data);
    temp->lchild = NULL;
    temp->rchild = NULL;
    return temp;
}

void create_fbinarytree()
{
    int j, i=0;
    printf("\n How many nodes you want: ");
    scanf("%d",&ctr);
    tree[0] = getnode();
    j = ctr;
    j--;
    do
    {
        if( j > 0 )                                /* left child */
        {
            tree[ i * 2 + 1 ] = getnode();
            tree[i]->lchild = tree[ i * 2 + 1 ];
            j--;
        }
        if( j > 0 )                                /* right child */
        {
            tree[ i * 2 + 2 ] = getnode();
            j--;
            tree[i]->rchild = tree[ i * 2 + 2 ];
        }
        i++;
    } while( j > 0 );
}
```

```

void inorder(node *root)
{
    if( root != NULL )
    {
        inorder(root->lchild);
        printf("%3s",root->data);
        inorder(root->rchild);
    }
}

```

```

void preorder(node *root)
{
    if( root != NULL )
    {
        printf("%3s",root->data);
        preorder(root->lchild);
        preorder(root->rchild);
    }
}

```

```

void postorder(node *root)
{
    if( root != NULL )
    {
        postorder(root->lchild);
        postorder(root->rchild);
        printf("%3s",root->data);
    }
}

```

```

void levelorder()
{
    int j;
    for(j = 0; j < ctr; j++)
    {
        if(tree[j] != NULL)
            printf("%3s",tree[j]->data);
    }
}

```

```

void print_leaf(node *root)
{
    if(root != NULL)
    {
        if(root->lchild == NULL && root->rchild == NULL)
            printf("%3s ",root->data);
        print_leaf(root->lchild);
        print_leaf(root->rchild);
    }
}

```

```

int height(node *root)
{
    if(root == NULL)
    {
        return 0;
    }
}

```

```

        if(root->lchild == NULL && root->rchild == NULL)
            return 0;
        else
            return (1 + max(height(root->lchild), height(root->rchild)));
    }

void main()
{
    int i;
    create_fbinarytree();
    printf("\n Inorder Traversal: ");
    inorder(tree[0]);
    printf("\n Preorder Traversal: ");
    preorder(tree[0]);
    printf("\n Postorder Traversal: ");
    postorder(tree[0]);
    printf("\n Level Order Traversal: ");
    levelorder();
    printf("\n Leaf Nodes: ");
    print_leaf(tree[0]);
    printf("\n Height of Tree: %d ", height(tree[0]));
}

```

5.3.4. Binary Tree Creation and Traversal Using Pointers:

This program performs the following operations:

1. Creates a complete Binary Tree
2. Inorder traversal
3. Preorder traversal
4. Postorder traversal
5. Level order traversal
6. Prints leaf nodes
7. Finds height of the tree created
8. Deletes last node
9. Finds height of the tree created

```

#include <stdio.h>
#include <stdlib.h>

struct tree
{
    struct tree* lchild;
    char data[10];
    struct tree* rchild;
};

typedef struct tree node;
node *Q[50];
int node_ctr;

node* getnode()
{
    node *temp ;
    temp = (node*) malloc(sizeof(node));
    printf("\n Enter Data: ");
    fflush(stdin);
    scanf("%s",temp->data);
    temp->lchild = NULL;
    temp->rchild = NULL;
    return temp;
}

```

```

void create_binarytree(node *root)
{
    char option;
    node_ctr = 1;
    if( root != NULL )
    {
        printf("\n Node  %s has Left SubTree(Y/N)",root->data);
        fflush(stdin);
        scanf("%c",&option);
        if( option=='Y' || option == 'y')
        {
            root->lchild = getnode();
            node_ctr++;
            create_binarytree(root->lchild);
        }
        else
        {
            root->lchild = NULL;
            create_binarytree(root->lchild);
        }

        printf("\n Node  %s has Right SubTree(Y/N) ",root->data);
        fflush(stdin);
        scanf("%c",&option);
        if( option=='Y' || option == 'y')
        {
            root->rchild = getnode();
            node_ctr++;
            create_binarytree(root->rchild);
        }
        else
        {
            root->rchild = NULL;
            create_binarytree(root->rchild);
        }
    }
}

```

```

void make_Queue(node *root,int parent)
{
    if(root != NULL)
    {
        node_ctr++;
        Q[parent] = root;
        make_Queue(root->lchild,parent*2+1);
        make_Queue(root->rchild,parent*2+2);
    }
}

```

```

delete_node(node *root, int parent)
{
    int index = 0;
    if(root == NULL)
        printf("\n Empty TREE ");
    else
    {
        node_ctr = 0;
        make_Queue(root,0);
        index = node_ctr-1;
        Q[index] = NULL;
        parent = (index-1) /2;
        if( 2* parent + 1 == index )
            Q[parent]->lchild = NULL;
    }
}

```

```

        else
            Q[parent]->rchild = NULL;
    }
    printf("\n Node Deleted ..");
}

void inorder(node *root)
{
    if(root != NULL)
    {
        inorder(root->lchild);
        printf("%3s",root->data);
        inorder(root->rchild);
    }
}

void preorder(node *root)
{
    if( root != NULL )
    {
        printf("%3s",root->data);
        preorder(root->lchild);
        preorder(root->rchild);
    }
}

void postorder(node *root)
{
    if( root != NULL )
    {
        postorder(root->lchild);
        postorder(root->rchild);
        printf("%3s", root->data);
    }
}

void print_leaf(node *root)
{
    if(root != NULL)
    {
        if(root->lchild == NULL && root->rchild == NULL)
            printf("%3s ",root->data);
        print_leaf(root->lchild);
        print_leaf(root->rchild);
    }
}

int height(node *root)
{
    if(root == NULL)
        return -1;
    else
        return (1 + max(height(root->lchild), height(root->rchild)));
}

void print_tree(node *root, int line)
{
    int i;
    if(root != NULL)
    {
        print_tree(root->rchild,line+1);
        printf("\n");
        for(i=0;i<line;i++)

```

```

        printf(" ");
        printf("%s", root->data);
        print_tree(root->lchild,line+1);
    }
}

void level_order(node *Q[],int ctr)
{
    int i;
    for( i = 0; i < ctr ; i++)
    {
        if( Q[i] != NULL )
            printf("%5s",Q[i]->data);
    }
}

int menu()
{
    int ch;
    clrscr();
    printf("\n 1. Create Binary Tree ");
    printf("\n 2. Inorder Traversal ");
    printf("\n 3. Preorder Traversal ");
    printf("\n 4. Postorder Traversal ");
    printf("\n 5. Level Order Traversal");
    printf("\n 6. Leaf Node ");
    printf("\n 7. Print Height of Tree ");
    printf("\n 8. Print Binary Tree ");
    printf("\n 9. Delete a node ");
    printf("\n 10. Quit ");
    printf("\n Enter Your choice: ");
    scanf("%d", &ch);
    return ch;
}

void main()
{
    int i,ch;
    node *root = NULL;
    do
    {
        ch = menu();
        switch( ch)
        {
            case 1 :
                if( root == NULL )
                {
                    root = getnode();
                    create_binarytree(root);
                }
                else
                {
                    printf("\n Tree is already Created ..");
                }
                break;
            case 2 :
                printf("\n Inorder Traversal: ");
                inorder(root);
                break;
            case 3 :
                printf("\n Preorder Traversal: ");
                preorder(root);
                break;
        }
    }
}

```

```

        case 4 :
            printf("\n Postorder Traversal: ");
            postorder(root);
            break;
        case 5:
            printf("\n Level Order Traversal ..");
            make_Queue(root,0);
            level_order(Q,node_ctr);
            break;
        case 6 :
            printf("\n Leaf Nodes: ");
            print_leaf(root);
            break;
        case 7 :
            printf("\n Height of Tree: %d ", height(root));
            break;
        case 8 :
            printf("\n Print Tree \n");
            print_tree(root, 0);
            break;
        case 9 :
            delete_node(root,0);
            break;
        case 10 :
            exit(0);
    }
    getch();
}while(1);
}

```

5.3.5. Non Recursive Traversal Algorithms:

At first glance, it appears that we would always want to use the flat traversal functions since they use less stack space. But the flat versions are not necessarily better. For instance, some overhead is associated with the use of an explicit stack, which may negate the savings we gain from storing only node pointers. Use of the implicit function call stack may actually be faster due to special machine instructions that can be used.

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.
2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

Algorithm inorder()

```

{
    stack[1] = 0
    vertex = root
top: while(vertex ≠ 0)
    {
        push the vertex into the stack
        vertex = leftson(vertex)
    }
}

```



```

    }

    pop the element from the stack and make it as vertex

    while(vertex ≠ 0)
    {
        print the vertex node
        if(rightson(vertex) ≠ 0)
        {
            vertex = rightson(vertex)
            goto top
        }
        pop the element from the stack and made it as vertex
    }
}

```

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.
2. Pop the vertex from stack, if vertex ≠ 0 then return to step one otherwise exit.

Algorithm preorder()

```

{
    stack[1] = 0
    vertex = root.
    while(vertex ≠ 0)
    {
        print vertex node
        if(rightson(vertex) ≠ 0)
            push the right son of vertex into the stack.
        if(leftson(vertex) ≠ 0)
            vertex = leftson(vertex)
        else
            pop the element from the stack and made it as vertex
    }
}

```

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push -(right son of vertex) onto stack.
2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

Algorithm postorder()

```

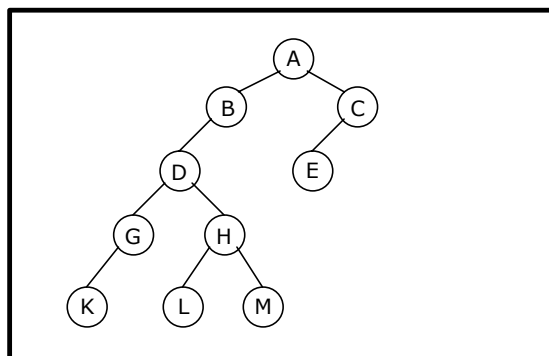
{
    stack[1] = 0
    vertex = root

top: while(vertex ≠ 0)
    {
        push vertex onto stack
        if(rightson(vertex) ≠ 0)
            push - (vertex) onto stack
        vertex = leftson(vertex)
    }
    pop from stack and make it as vertex
    while(vertex > 0)
    {
        print the vertex node
        pop from stack and make it as vertex
    }
    if(vertex < 0)
    {
        vertex = - (vertex)
        goto top
    }
}

```

Example 1:

Traverse the following binary tree in pre, post and inorder using non-recursive traversing algorithm.



Binary Tree

- Preorder traversal yields:
A, B, D, G, K, H, L, M, C, E
- Postorder traversal yields:
K, G, L, M, H, D, B, E, C, A
- Inorder traversal yields:
K, G, D, L, H, M, B, A, E, C

Pre, Post and Inorder Traversing

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.
2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
A	0		PUSH 0
	0 A B D G K		PUSH the left most path of A
K	0 A B D G	K	POP K
G	0 A B D	K G	POP G since K has no right son
D	0 A B	K G D	POP D since G has no right son
H	0 A B	K G D	Make the right son of D as vertex
	0 A B H L	K G D	PUSH the leftmost path of H
L	0 A B H	K G D L	POP L
H	0 A B	K G D L H	POP H since L has no right son
M	0 A B	K G D L H	Make the right son of H as vertex
	0 A B M	K G D L H	PUSH the left most path of M
M	0 A B	K G D L H M	POP M
B	0 A	K G D L H M B	POP B since M has no right son
A	0	K G D L H M B A	Make the right son of A as vertex
C	0 C E	K G D L H M B A	PUSH the left most path of C
E	0 C	K G D L H M B A E	POP E
C	0	K G D L H M B A E C	Stop since stack is empty

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push -(right son of vertex) onto stack.
2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
A	0		PUSH 0
	0 A -C B D -H G K		PUSH the left most path of A with a -ve for right sons
	0 A -C B D -H	K G	POP all +ve nodes K and G
H	0 A -C B D	K G	Pop H

	0 A -C B D H -M L	K G	PUSH the left most path of H with a -ve for right sons
L	0 A -C B D H -M	K G L	POP all +ve nodes L
M	0 A -C B D H	K G L	Pop M
	0 A -C B D H M	K G L	PUSH the left most path of M with a -ve for right sons
	0 A -C	K G L M H D B	POP all +ve nodes M, H, D and B
C	0 A	K G L M H D B	Pop C
	0 A C E	K G L M H D B	PUSH the left most path of C with a -ve for right sons
	0	K G L M H D B E C A	POP all +ve nodes E, C and A
	0	K G L M H D B E C A	Stop since stack is empty

Preorder Traversal:

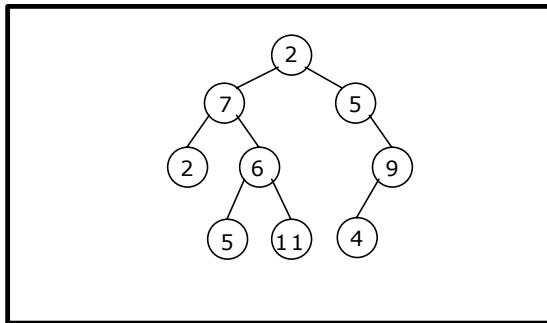
Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.
2. Pop the vertex from stack, if vertex $\neq 0$ then return to step one otherwise exit.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
A	0		PUSH 0
	0 C H	A B D G K	PUSH the right son of each vertex onto stack and process each vertex in the left most path
H	0 C	A B D G K	POP H
	0 C M	A B D G K H L	PUSH the right son of each vertex onto stack and process each vertex in the left most path
M	0 C	A B D G K H L	POP M
	0 C	A B D G K H L M	PUSH the right son of each vertex onto stack and process each vertex in the left most path; M has no left path
C	0	A B D G K H L M	Pop C
	0	A B D G K H L M C E	PUSH the right son of each vertex onto stack and process each vertex in the left most path; C has no right son on the left most path
	0	A B D G K H L M C E	Stop since stack is empty

Example 2:

Traverse the following binary tree in pre, post and inorder using non-recursive traversing algorithm.



Binary Tree

- Preorder traversal yields:
2, 7, 2, 6, 5, 11, 5, 9, 4
- Postorder traversal yields:
2, 5, 11, 6, 7, 4, 9, 5, 2
- Inorder traversal yields:
2, 7, 5, 6, 11, 2, 5, 4, 9

Pre, Post and In order Traversing

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.
2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
2	0		
	0 2 7 2		
2	0 2 7	2	
7	0 2	2 7	
6	0 2 6 5	2 7	
5	0 2 6	2 7 5	
6	0 2	2 7 5 6	
11	0 2 11	2 7 5 6	
11	0 2	2 7 5 6 11	
2	0	2 7 5 6 11 2	
5	0 5	2 7 5 6 11 2	
5	0	2 7 5 6 11 2 5	
9	0 9 4	2 7 5 6 11 2 5	
4	0 9	2 7 5 6 11 2 5 4	
9	0	2 7 5 6 11 2 5 4 9	Stop since stack is empty

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push -(right son of vertex) onto stack.
2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
2	0		
	0 2 -5 7 -6 2		
2	0 2 -5 7 -6	2	
6	0 2 -5 7	2	
	0 2 -5 7 6 -11 5	2	
5	0 2 -5 7 6 -11	2 5	
11	0 2 -5 7 6 11	2 5	
	0 2 -5	2 5 11 6 7	
5	0 2 5 -9	2 5 11 6 7	
9	0 2 5 9 4	2 5 11 6 7	
	0	2 5 11 6 7 4 9 5 2	Stop since stack is empty

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.
2. Pop the vertex from stack, if vertex $\neq 0$ then return to step one otherwise exit.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
2	0		
	0 5 6	2 7 2	
6	0 5 11	2 7 2 6 5	
11	0 5	2 7 2 6 5 11	
	0 5	2 7 2 6 5 11	
5	0 9	2 7 2 6 5 11 5	
9	0	2 7 2 6 5 11 5 9 4	
	0	2 7 2 6 5 11 5 9 4	Stop since stack is empty

5.4. Expression Trees:

Expression tree is a binary tree, because all of the operations are binary. It is also possible for a node to have only one child, as is the case with the unary minus operator. The leaves of an expression tree are operands, such as constants or variable names, and the other (non leaf) nodes contain operators.

Once an expression tree is constructed we can traverse it in three ways:

- Inorder Traversal
- Preorder Traversal
- Postorder Traversal

Figure 5.4.1 shows some more expression trees that represent arithmetic expressions given in infix form.

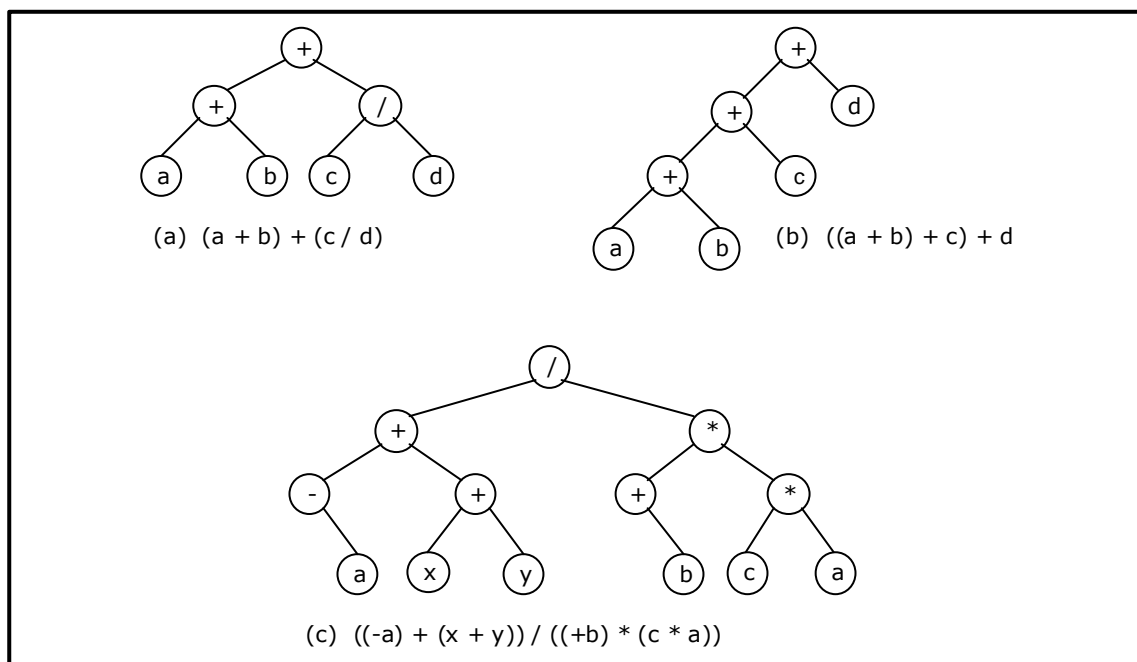


Figure 5.4.1 Expression Trees

An expression tree can be generated for the infix and postfix expressions.

An algorithm to convert a postfix expression into an expression tree is as follows:

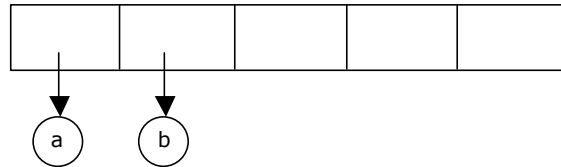
1. Read the expression one symbol at a time.
2. If the symbol is an operand, we create a one-node tree and push a pointer to it onto a stack.
3. If the symbol is an operator, we pop pointers to two trees T1 and T2 from the stack (T1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T2 and T1 respectively. A pointer to this new tree is then pushed onto the stack.

Example 1:

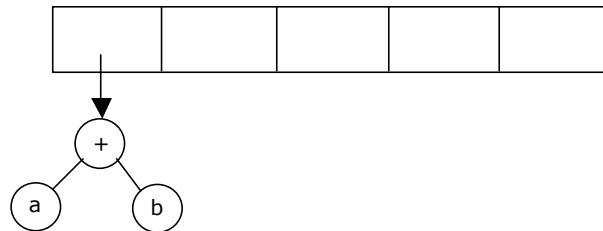
Construct an expression tree for the postfix expression: $a\ b\ +\ c\ d\ e\ +\ *\ *$

Solution:

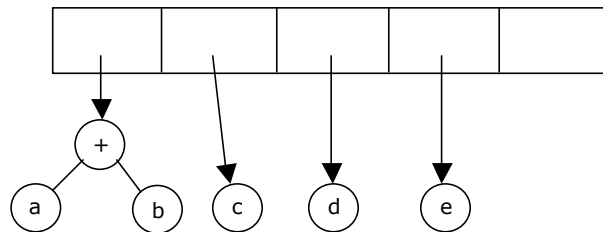
The first two symbols are operands, so we create one-node trees and push pointers to them onto a stack.



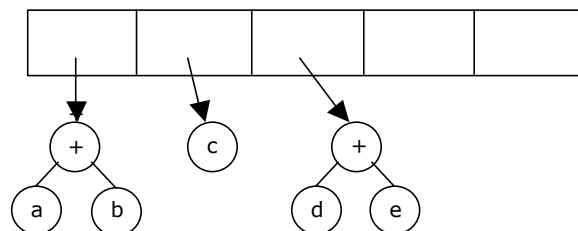
Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



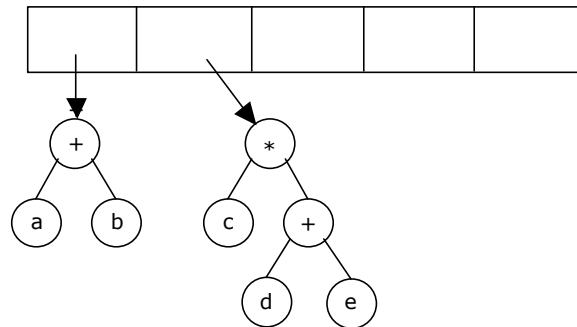
Next, c, d, and e are read, and for each one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



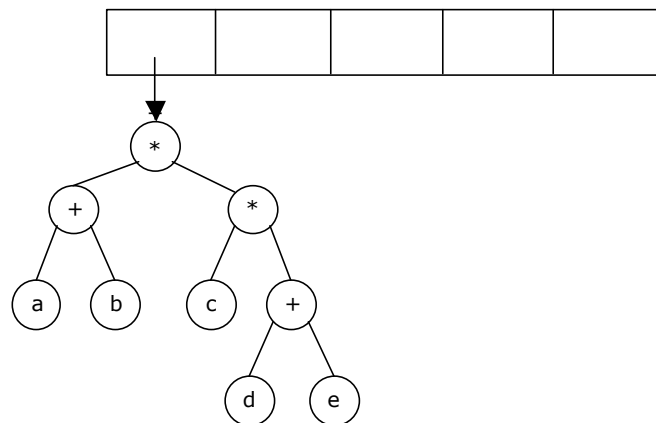
Now a '+' is read, so two trees are merged.



Continuing, a '*' is read, so we pop two tree pointers and form a new tree with a '*' as root.



Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack.



For the above tree:

Inorder form of the expression: $a + b * c * d + e$

Preorder form of the expression: $* + a b * c + d e$

Postorder form of the expression: $a b + c d e + * *$

Example 2:

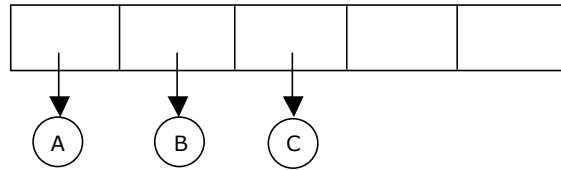
Construct an expression tree for the arithmetic expression:

$$(A + B * C) - ((D * E + F) / G)$$

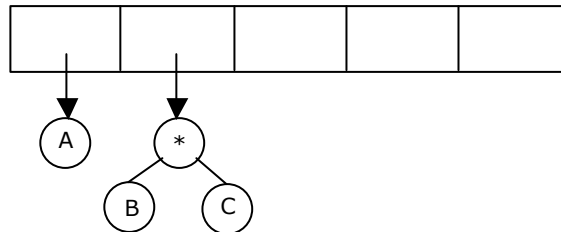
Solution:

First convert the infix expression into postfix notation. Postfix notation of the arithmetic expression is: $A B C * + D E * F + G / -$

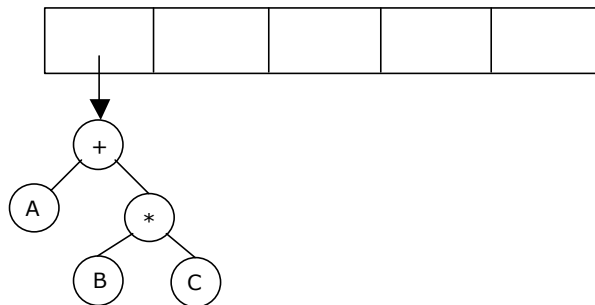
The first three symbols are operands, so we create one-node trees and pointers to three nodes pushed onto the stack.



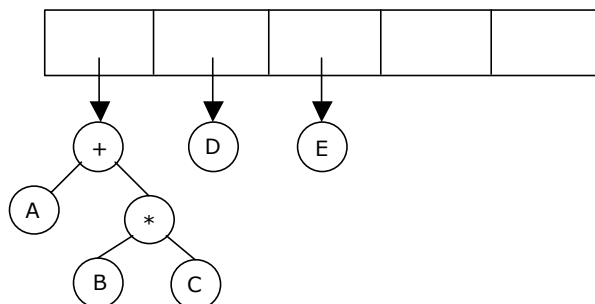
Next, a '*' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



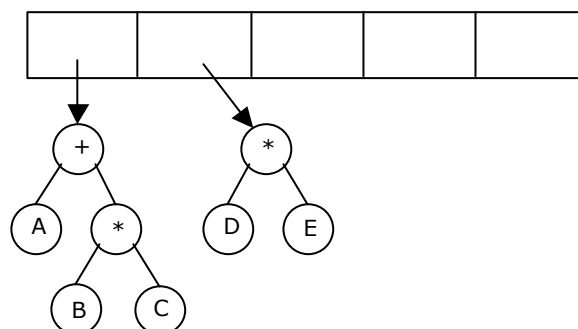
Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



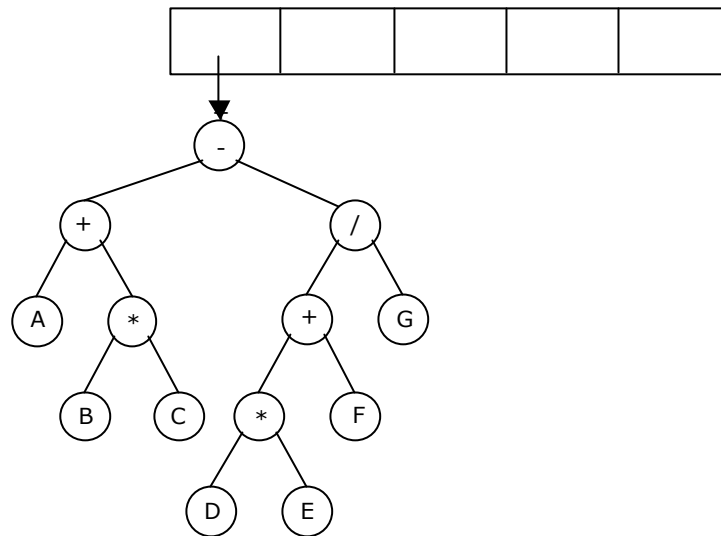
Next, D and E are read, and for each one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



Continuing, a '*' is read, so we pop two tree pointers and form a new tree with a '*' as root.



Proceeding similar to the previous steps, finally, when the last symbol is read, the expression tree is as follows:



5.4.1. Converting expressions with expression trees:

Let us convert the following expressions from one type to another. These can be as follows:

1. Postfix to infix
2. Postfix to prefix
3. Prefix to infix
4. Prefix to postfix

1. Postfix to Infix:

The following algorithm works for the expressions whose infix form does not require parenthesis to override conventional precedence of operators.

- A. Create the expression tree from the postfix expression
- B. Run inorder traversal on the tree.

2. Postfix to Prefix:

The following algorithm works for the expressions to convert postfix to prefix:

- A. Create the expression tree from the postfix expression
- B. Run preorder traversal on the tree.

3. Prefix to Infix:

The following algorithm works for the expressions whose infix form does not require parenthesis to override conventional precedence of operators.

- A. Create the expression tree from the prefix expression
- B. Run inorder traversal on the tree.

4. Prefix to postfix:

The following algorithm works for the expressions to convert postfix to prefix:

- A. Create the expression tree from the prefix expression
- B. Run postorder traversal on the tree.

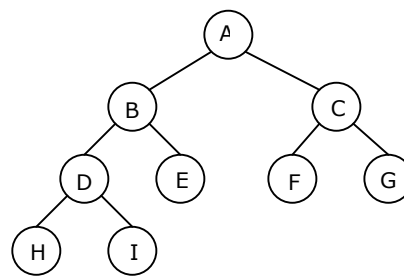
5.5. Threaded Binary Tree:

The linked representation of any binary tree has more null links than actual pointers. If there are $2n$ total links, there are $n+1$ null links. A clever way to make use of these null links has been devised by A.J. Perlis and C. Thornton.

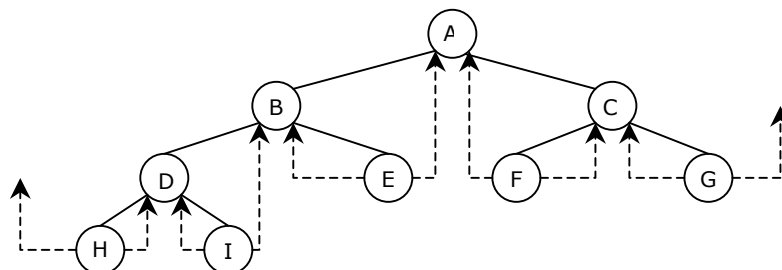
Their idea is to replace the null links by pointers called Threads to other nodes in the tree.

If the $RCHILD(p)$ is normally equal to zero, we will replace it by a pointer to the node which would be printed after P when traversing the tree in inorder.

A null $LCHILD$ link at node P is replaced by a pointer to the node which immediately precedes node P in inorder. For example, Let us consider the tree:



The Threaded Tree corresponding to the above tree is:



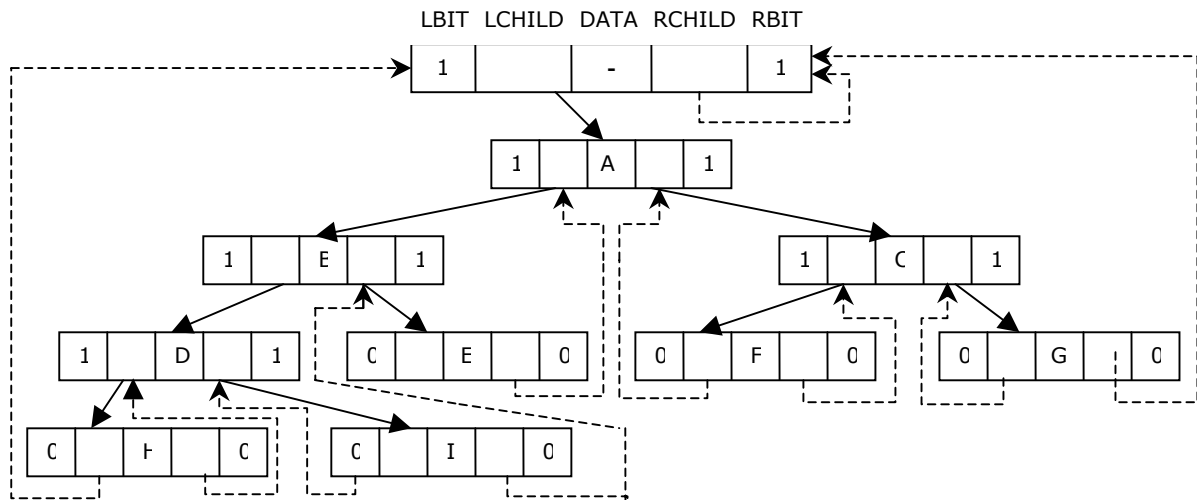
The tree has 9 nodes and 10 null links which have been replaced by Threads. If we traverse T in inorder the nodes will be visited in the order H D I B E A F C G.

For example, node 'E' has a predecessor Thread which points to 'B' and a successor Thread which points to 'A'. In memory representation Threads and normal pointers are distinguished between as by adding two extra one bit fields LBIT and RBIT.

$LBIT(P) = 1$ if $LCHILD(P)$ is a normal pointer
 $LBIT(P) = 0$ if $LCHILD(P)$ is a Thread

$RBIT(P) = 1$ if $RCHILD(P)$ is a normal pointer
 $RBIT(P) = 0$ if $RCHILD(P)$ is a Thread

In the above figure two threads have been left dangling in LCHILD(H) and RCHILD(G). In order to have no loose Threads we will assume a head node for all threaded binary trees. The Complete memory representation for the tree is as follows. The tree T is the left sub-tree of the head node.



5.6. Binary Search Tree:

A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

1. Every element has a key and no two elements have the same key.
2. The keys in the left subtree are smaller than the key in the root.
3. The keys in the right subtree are larger than the key in the root.
4. The left and right subtrees are also binary search trees.

Figure 5.2.5(a) is a binary search tree, whereas figure 5.2.5(b) is not a binary search tree.

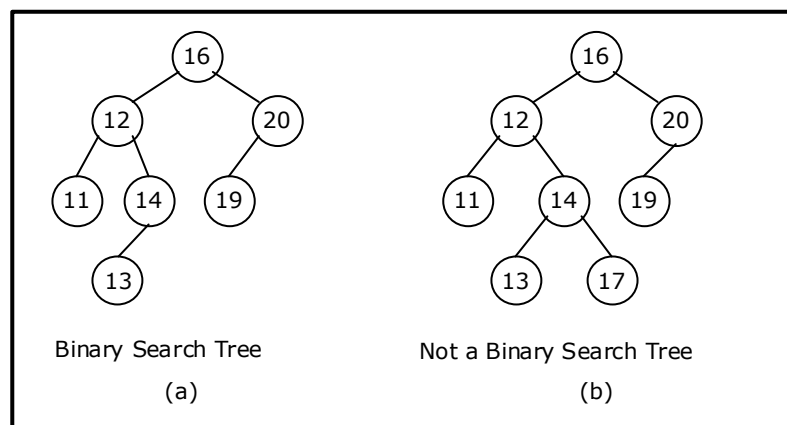


Figure 5.2.5. Examples of binary search trees

5.7. General Trees (m-ary tree):

If in a tree, the outdegree of every node is less than or equal to m , the tree is called general tree. The general tree is also called as an m -ary tree. If the outdegree of every node is exactly equal to m or zero then the tree is called a *full or complete m-ary tree*. For $m = 2$, the trees are called *binary* and *full binary trees*.

Differences between trees and binary trees:

TREE	BINARY TREE
Each element in a tree can have any number of subtrees.	Each element in a binary tree has at most two subtrees.
The subtrees in a tree are unordered.	The subtrees of each element in a binary tree are ordered (i.e. we distinguish between left and right subtrees).

5.7.1. Converting a m -ary tree (general tree) to a binary tree:

There is a one-to-one mapping between general ordered trees and binary trees. So, every tree can be uniquely represented by a binary tree. Furthermore, a forest can also be represented by a binary tree.

Conversion from general tree to binary can be done in two stages.

Stage 1:

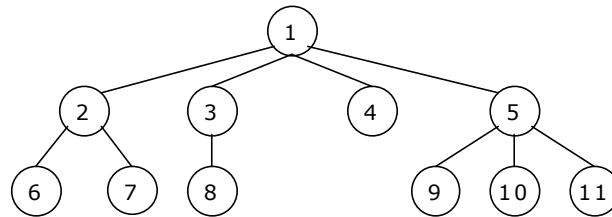
- As a first step, we delete all the branches originating in every node except the left most branch.
- We draw edges from a node to the node on the right, if any, which is situated at the same level.

Stage 2:

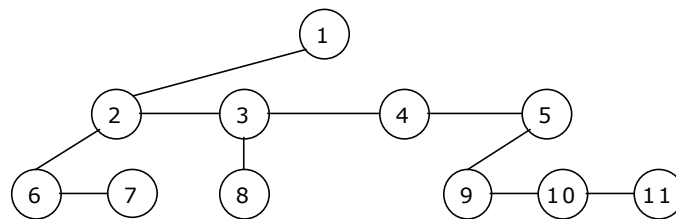
- Once this is done then for any particular node, we choose its left and right sons in the following manner:
 - The left son is the node, which is immediately below the given node, and the right son is the node to the immediate right of the given node on the same horizontal line. Such a binary tree will not have a right subtree.

Example 1:

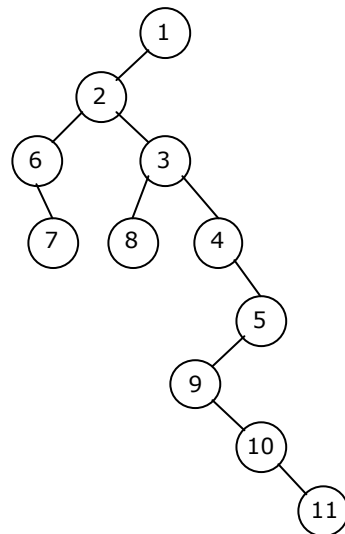
Convert the following ordered tree into a binary tree:

**Solution:**

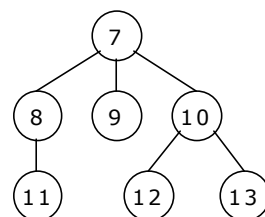
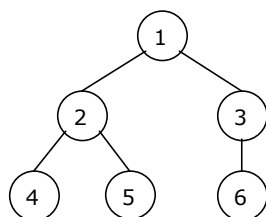
Stage 1 tree by using the above mentioned procedure is as follows:



Stage 2 tree by using the above mentioned procedure is as follows:

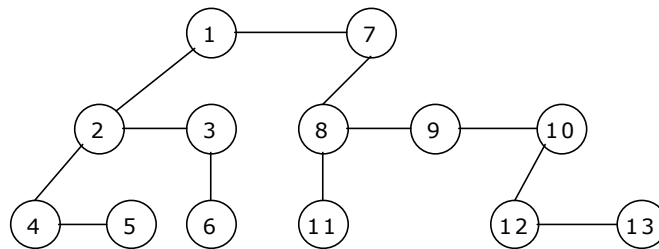
**Example 2:**

Construct a unique binary tree from the given forest.

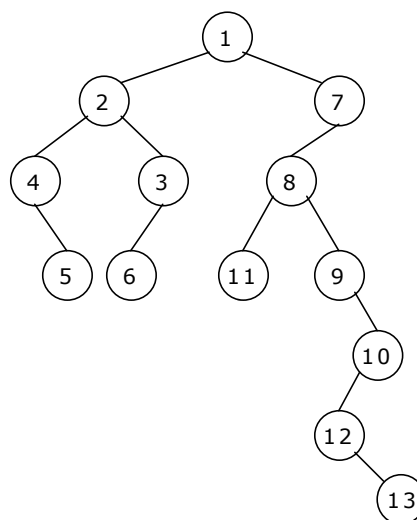


Solution:

Stage 1 tree by using the above mentioned procedure is as follows:

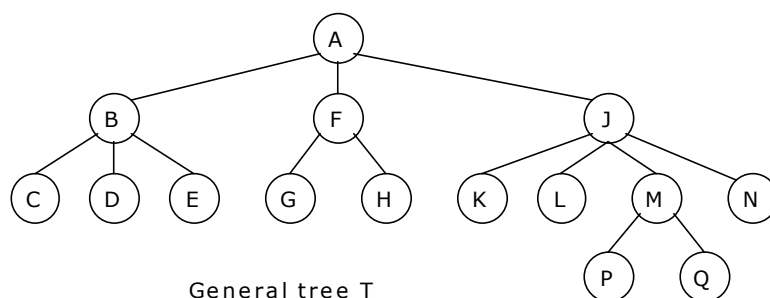


Stage 2 tree by using the above mentioned procedure is as follows (binary tree representation of forest):

**Example 3:**

For the general tree shown below:

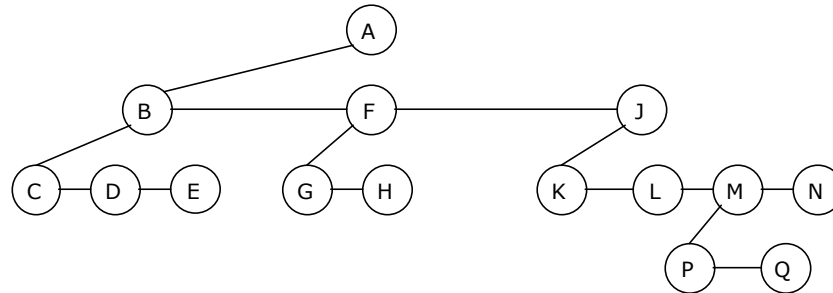
1. Find the corresponding binary tree T' .
2. Find the preorder traversal and the postorder traversal of T .
3. Find the preorder, inorder and postorder traversals of T' .
4. Compare them with the preorder and postorder traversals obtained for T' with the general tree T .



Solution:

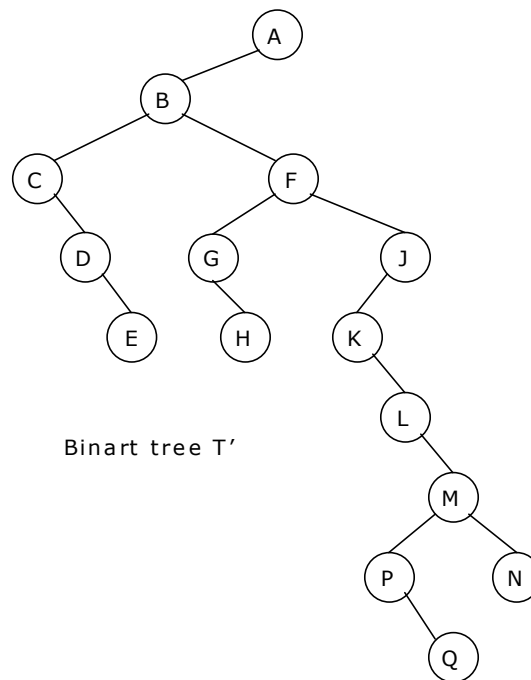
1. Stage 1:

The tree by using the above-mentioned procedure is as follows:



Stage 2:

The binary tree by using the above-mentioned procedure is as follows:



2. Suppose T is a general tree with root R and subtrees T_1, T_2, \dots, T_M . The preorder traversal and the postorder traversal of T are:

Preorder: 1) Process the root R.
2) Traverse the subtree T_1, T_2, \dots, T_M in preorder.

Postorder: 1) Traverse the subtree T_1, T_2, \dots, T_M in postorder.
2) Process the root R.

The tree T has the root A and subtrees T_1, T_2 and T_3 such that:

T_1 consists of nodes B, C, D and E.

T_2 consists of nodes F, G and H.

T_3 consists of nodes J, K, L, M, N, P and Q.

- A. The preorder traversal of T consists of the following steps:
- (i) Process root A.
 - (ii) Traverse T_1 in preorder: Process nodes B, C, D, E.
 - (iii) Traverse T_2 in preorder: Process nodes F, G, H.
 - (iv) Traverse T_3 in preorder: Process nodes J, K, L, M, P, Q, N.

The preorder traversal of T is as follows:

A, B, C, D, E, F, G, H, J, K, L, M, P, Q, N

- B. The postorder traversal of T consists of the following steps:
- (i) Traverse T_1 in postorder: Process nodes C, D, E, B.
 - (ii) Traverse T_2 in postorder: Process nodes G, H, F.
 - (iii) Traverse T_3 in postorder: Process nodes K, L, P, Q, M, N, J.
 - (iv) Process root A.

The postorder traversal of T is as follows:

C, D, E, B, G, H, F, K, L, P, Q, M, N, J, A

3. The preorder, inorder and postorder traversals of the binary tree T' are as follows:

Preorder: A, B, C, D, E, F, G, H, J, K, M, P, Q, N

Inorder: C, D, E, B, G, H, F, K, L, P, Q, M, N, J, A

Postorder: E, D, C, H, G, Q, P, N, M, L, K, J, F, B, A

4. Comparing the preorder and postorder traversals of T' with the general tree T:

We can observe that the preorder of the binary tree T' is identical to the preorder of the general T.

The inorder traversal of the binary tree T' is identical to the postorder traversal of the general tree T.

There is no natural traversal of the general tree T which corresponds to the postorder traversal of its corresponding binary tree T' .

5.8. Search and Traversal Techniques for m-ary trees:

Search involves visiting nodes in a tree in a systematic manner, and may or may not result into a visit to all nodes. When the search necessarily involved the examination of every vertex in the tree, it is called the traversal. Traversing of a tree can be done in two ways.

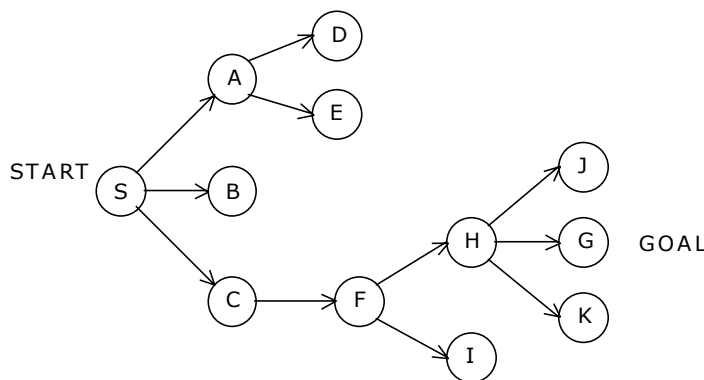
1. Depth first search or traversal.
2. Breadth first search or traversal.

5.8.1. Depth first search:

In Depth first search, we begin with root as a start state, then some successor of the start state, then some successor of that state, then some successor of that and so on, trying to reach a goal state. One simple way to implement depth first search is to use a stack data structure consisting of root node as a start state.

If depth first search reaches a state S without successors, or if all the successors of a state S have been chosen (visited) and a goal state has not get been found, then it "backs up" that means it goes to the immediately previous state or predecessor formally, the state whose successor was 'S' originally.

To illustrate this let us consider the tree shown below.



Suppose S is the start and G is the only goal state. Depth first search will first visit S, then A, then D. But D has no successors, so we must back up to A and try its second successor, E. But this doesn't have any successors either, so we back up to A again. But now we have tried all the successors of A and haven't found the goal state G so we must back to 'S'. Now 'S' has a second successor, B. But B has no successors, so we back up to S again and choose its third successor, C. C has one successor, F. The first successor of F is H, and the first of H is J. J doesn't have any successors, so we back up to H and try its second successor. And that's G, the only goal state.

So the solution path to the goal is S, C, F, H and G and the states considered were in order S, A, D, E, B, C, F, H, J, G.

Disadvantages:

1. It works very fine when search graphs are trees or lattices, but can get struck in an infinite loop on graphs. This is because depth first search can travel around a cycle in the graph forever.

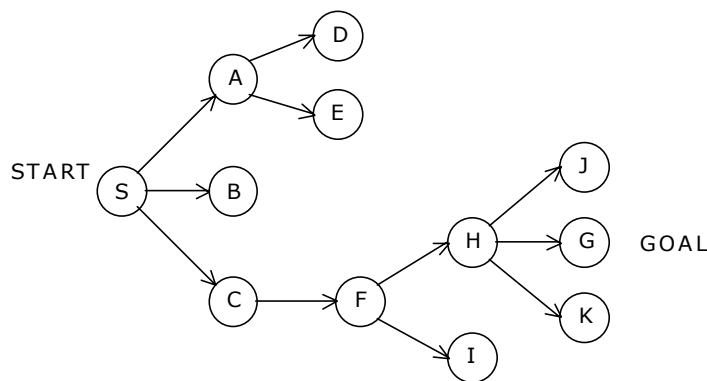
To eliminate this keep a list of states previously visited, and never permit search to return to any of them.

2. We cannot come up with shortest solution to the problem.

5.8.2. Breadth first search:

Breadth-first search starts at root node S and "discovers" which vertices are reachable from S. Breadth-first search discovers vertices in increasing order of distance. Breadth-first search is named because it visits vertices across the entire breadth.

To illustrate this let us consider the following tree:



Breadth first search finds states level by level. Here we first check all the immediate successors of the start state. Then all the immediate successors of these, then all the immediate successors of these, and so on until we find a goal node. Suppose S is the start state and G is the goal state. In the figure, start state S is at level 0; A, B and C are at level 1; D, e and F at level 2; H and I at level 3; and J, G and K at level 4.

So breadth first search, will consider in order S, A, B, C, D, E, F, H, I, J and G and then stop because it has reached the goal node.

Breadth first search does not have the danger of infinite loops as we consider states in order of increasing number of branches (level) from the start state.

One simple way to implement breadth first search is to use a queue data structure consisting of just a start state.

5.9. Sparse Matrices:

A sparse matrix is a two-dimensional array having the value of majority elements as null. The density of the matrix is the number of non-zero elements divided by the total number of matrix elements. The matrices with very low density are often good for use of the sparse format. For example,

$$A = \begin{pmatrix} 0 & 0 & 0 & 5 \\ 0 & 2 & 0 & 0 \\ 1 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \end{pmatrix}$$

As far as the storage of a sparse matrix is concerned, storing of null elements is nothing but wastage of memory. So we should devise technique such that only non-null elements will be stored. The matrix A produces:

$$S = \begin{pmatrix} (3, 1) & 1 \\ (2, 2) & 2 \\ (3, 2) & 3 \\ (4, 3) & 4 \\ (1, 4) & 5 \end{pmatrix}$$

The printed output lists the non-zero elements of S, together with their row and column indices. The elements are sorted by columns, reflecting the internal data structure.

In large number of applications, sparse matrices are involved. One approach is to use the linked list.

The program to represent sparse matrix:

```
/*      Check whether the given matrix is sparse matrix or not, if so then print in
        alternative form for storage.      */

#include <stdio.h>
#include <conio.h>

main()
{
    int matrix[20][20], m, n, total_elements, total_zeros = 0, i, j;
    clrscr();
    printf("\n Enter Number of rows and columns: ");
    scanf("%d %d",&m, &n);
    total_elements = m * n;
    printf("\n Enter data for sparse matrix: ");
    for(i = 0; i < m ; i++)
    {
        for( j = 0; j < n ; j++)
        {
            scanf("%d", &matrix[i][j]);
            if( matrix[i][j] == 0)
            {
                total_zeros++;
            }
        }
    }
    if(total_zeros > total_elements/2 )
    {
        printf("\n Given Matrix is Sparse Matrix..");
        printf("\n The Representaion of Sparse Matrix is: \n");
        printf("\n Row \t Col \t Value ");
        for(i = 0; i < m ; i++)
        {
            for( j = 0; j < n ; j++)
            {
                if( matrix[i][j] != 0)
                {
                    printf("\n %d \t %d \t %d",i,j,matrix[i][j]);
                }
            }
        }
    }
    else
        printf("\n Given Matrix is Not a Sparse Matrix..");
}
```

EXERCISES

1. How many different binary trees can be made from three nodes that contain the key value 1, 2, and 3?
2.
 - a. Draw all the possible binary trees that have four leaves and all the nonleaf nodes have no children.
 - b. Show what would be printed by each of the following.
An inorder traversal of the tree
A postorder traversal of the tree
A preorder traversal of the tree
3.
 - a. Draw the binary search tree whose elements are inserted in the following order:
50 72 96 94 107 26 12 11 9 2 10 25 51 16 17 95
 - b. What is the height of the tree?
 - c. What nodes are on level?
 - d. Which levels have the maximum number of nodes that they could contain?
 - e. What is the maximum height of a binary search tree containing these nodes?
Draw such a tree?
 - f. What is the minimum height of a binary search tree containing these nodes?
Draw such a tree?
 - g. Show how the tree would look after the deletion of 29, 59 and 47?
 - h. Show how the (original) tree would look after the insertion of nodes containing 63, 77, 76, 48, 9 and 10 (in that order).
4. Write a "C" function to determine the height of a binary tree.
5. Write a "C" function to count the number of leaf nodes in a binary tree.
6. Write a "C" function to swap a binary tree.
7. Write a "C" function to compute the maximum number of nodes in any level of a binary tree. The maximum number of nodes in any level of a binary tree is also called the width of the tree.
8. Construct two binary trees so that their postorder traversal sequences are the same.
9. Write a "C" function to compute the internal path length of a binary tree.
10. Write a "C" function to compute the external path length of a binary tree.
11. Prove that every node in a tree except the root node has a unique parent.
12. Write a "C" function to reconstruct a binary tree from its preorder and inorder traversal sequences.
13. Prove that the inorder and postorder traversal sequences of a binary tree uniquely characterize the binary tree. Write a "C" function to reconstruct a binary tree from its postorder and inorder traversal sequences.

14. Build the binary tree from the given traversal techniques:

A. Inorder: g d h b e i a f j c
Preorder: a b d g h e i c f j

B. Inorder: g d h b e i a f j c
Postorder: g h d i e b j f c a

C. Inorder: g d h b e i a f j c
Level order: a b c d e f g h i j

15. Build the binary tree from the given traversal techniques:

A. Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Preorder: n6 n2 n1 n4 n3 n5 n9 n7 n8

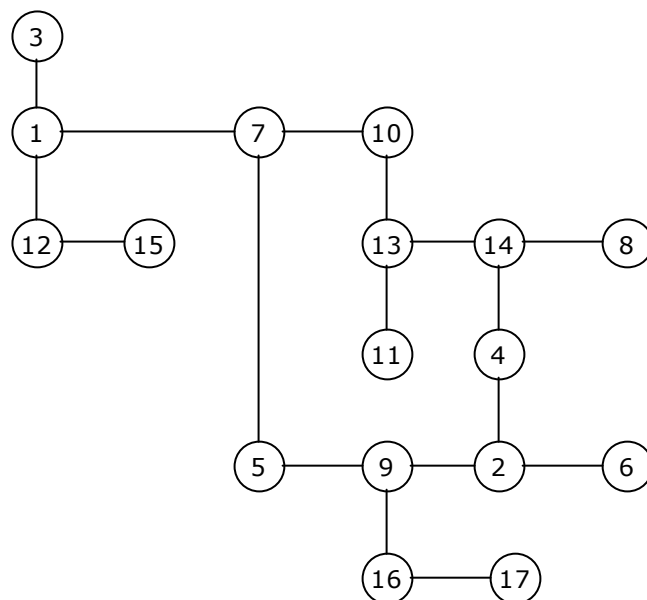
B. Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Postorder: n1 n3 n5 n4 n2 n8 n7 n9 n6

C. Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Level order: n6 n2 n9 n1 n4 n7 n3 n5 n8

16. Build the binary tree for the given inorder and preorder traversals:

Inorder: E A C K F H D B G
Preorder: F A E K C D H G B

17. Convert the following general tree represented as a binary tree:



Multiple Choice Questions

1. The node that has no children is referred as: [C]
A. Parent node
B. Root node
C. Leaf node
D. Siblings
2. A binary tree in which all the leaves are on the same level is called as: [B]
A. Complete binary tree
B. Full binary tree
C. Strictly binary tree
D. Binary search tree
3. How can the graphs be represented? [C]
A. Adjacency matrix
B. Adjacency list
C. Incidence matrix
D. All of the above
4. The children of a same parent node are called as: [C]
A. adjacent node
B. non-leaf node
C. Siblings
D. leaf node
5. A tree with n vertices, consists of _____ edges. [A]
A. $n - 1$
B. $n - 2$
C. n
D. $\log n$
6. The maximum number of nodes at any level is: [B]
A. n
B. 2^n
C. $n + 1$
D. $2n$

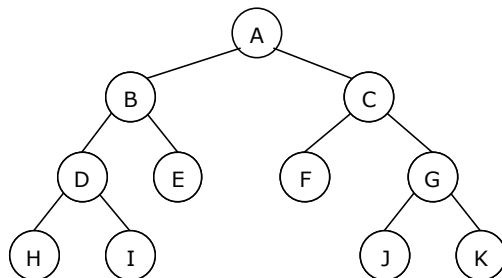
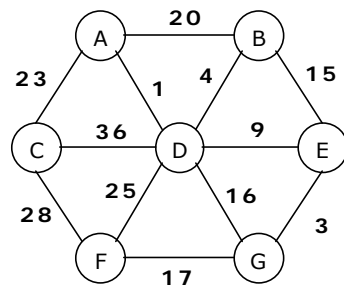


FIGURE 1

7. For the Binary tree shown in fig. 1, the in-order traversal sequence is: [C]
 A. A B C D E F G H I J K C. H D I B E A F C J G K
 B. H I D E B F J K G C A D. A B D H I E C F G J K
8. For the Binary tree shown in fig. 1, the pre-order traversal sequence is: [D]
 A. A B C D E F G H I J K C. H D I B E A F C J G K
 B. H I D E B F J K G C A D. A B D H I E C F G J K
9. For the Binary tree shown in fig. 1, the post-order traversal sequence is: [B]
 A. A B C D E F G H I J K C. H D I B E A F C J G K
 B. H I D E B F J K G C A D. A B D H I E C F G J K



Node	Adjacency List
A	B C D
B	A D E
C	A D F
D	A B C E F G
E	B D G
F	C D G
G	F D E

FIGURE 2 and its adjacency list

10. Which is the correct order for Kruskal's minimum spanning tree algorithm to add edges to the minimum spanning tree for the figure 2 shown above: [B]
- A. (A, B) then (A, C) then (A, D) then (D, E) then (C, F) then (D, G)
 B. (A, D) then (E, G) then (B, D) then (D, E) then (F, G) then (A, C)
 C. both A and B
 D. none of the above
11. For the figure 2 shown above, the cost of the minimal spanning tree is: [A]
- A. 57
 B. 68
 C. 48
 D. 32

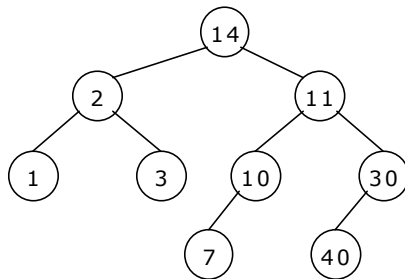


FIGURE 3

12. For the figure 3, how many leaves does it have? [B]
- A. 2
 B. 4
 C. 6
 D. 8
13. For the figure 3, how many of the nodes have at least one sibling? [A]
- A. 5
 B. 6
 C. 7
 D. 8
14. For the figure 3, How many descendants does the root have? [D]
- A. 0
 B. 2
 C. 4
 D. 8
15. For the figure 3, what is the depth of the tree? [B]
- A. 2
 B. 3
 C. 4
 D. 8
16. For the figure 3, which statement is correct? [A]
- A. The tree is neither complete nor full.
 B. The tree is complete but not full.
 C. The tree is full but not complete.
 D. The tree is both full and complete.

17. There is a tree in the box at the top of this section. What is the order of nodes visited using a pre-order traversal? []
 A. 1 2 3 7 10 11 14 30 40 C. 1 3 2 7 10 40 30 11 14
 B. 1 2 3 14 7 10 11 40 30 D. 14 2 1 3 11 10 7 30 40
18. There is a tree in the box at the top of this section. What is the order of nodes visited using an in-order traversal? []
 A. 1 2 3 7 10 11 14 30 40 C. 1 3 2 7 10 40 30 11 14
 B. 1 2 3 14 7 10 11 40 30 D. 14 2 1 3 11 10 7 30 40
19. There is a tree in the box at the top of this section. What is the order of nodes visited using a post-order traversal? []
 A. 1 2 3 7 10 11 14 30 40 C. 1 3 2 7 10 40 30 11 14
 B. 1 2 3 14 7 10 11 40 30 D. 14 2 1 3 11 10 7 30 40
20. What is the minimum number of nodes in a full binary tree with depth 3? [D]
 A. 3 C. 8
 B. 4 D. 15
21. Select the one true statement. [C]
 A. Every binary tree is either complete or full.
 B. Every complete binary tree is also a full binary tree.
 C. Every full binary tree is also a complete binary tree.
 D. No binary tree is both complete and full.
22. Suppose T is a binary tree with 14 nodes. What is the minimum possible depth of T? [B]
 A. 0 C. 4
 B. 3 D. 5
23. Select the one FALSE statement about binary trees: [A]
 A. Every binary tree has at least one node.
 B. Every non-empty tree has exactly one root node.
 C. Every node has at most two children.
 D. Every non-root node has exactly one parent.
24. Consider the node of a complete binary tree whose value is stored in data[i] for an array implementation. If this node has a right child, where will the right child's value be stored? [C]
 A. data[i+1] C. data[2*i + 1]
 B. data[i+2] D. data[2*i + 2]

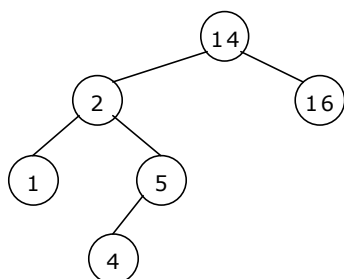
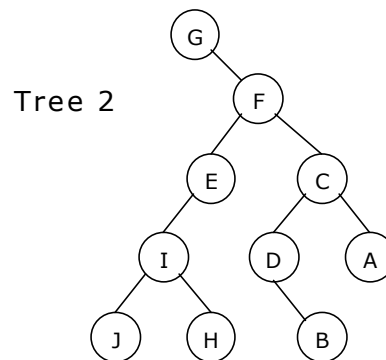
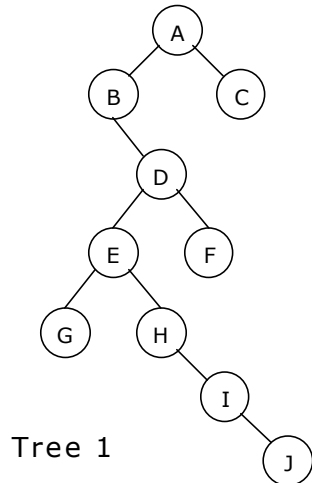
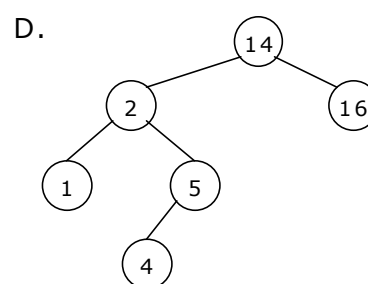
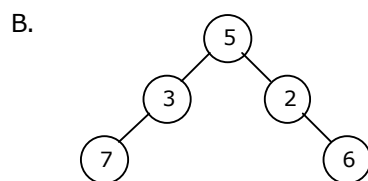
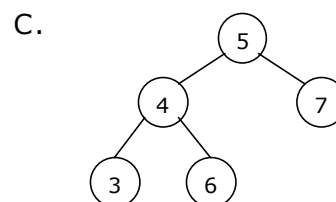
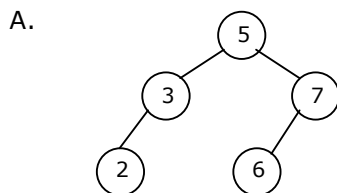


Figure 4

25. For the binary search tree shown in figure 4, Suppose we remove the root, replacing it with something from the left subtree. What will be the new root? [D]
- A. 1
B. 2
C. 4
D. 5
E. 16



26. Which traversals of tree 1 and tree 2, will produce the same sequence of node names? [C]
- A. Preorder, Postorder
B. Postorder, Postorder
C. Postorder, Inorder
D. Inorder, Inorder
27. Which among the following is not a binary search tree? [C]



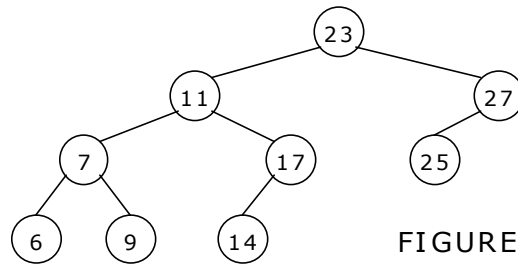


FIGURE 5

28. For the binary search tree shown in figure 5, after deleting 23 from the binary search tree what node will be at the root? []
 A. 11
 B. 25
 C. 27
 D. 14
29. For the binary search tree shown in figure 5, after deleting 23 from the binary search tree what parent → child pair does not occur in the tree? [B]
 A. 25 → 27
 B. 27 → 11
 C. 11 → 7
 D. 7 → 9
30. The number of nodes in a complete binary tree of depth d is: [B]
 A. $2d$
 B. $2^k - 1$
 C. 2^k
 D. none of the above
31. The depth of a complete binary tree with n nodes is: [C]
 A. $\log n$
 B. n^2
 C. $\lfloor \log_2 n \rfloor + 1$
 D. $2n$
32. If the inorder and preorder traversal of a binary tree are D, B, F, E, G, H, A, C and A, B, D, E, F, G, H, C respectively then, the postorder traversal of that tree is: [A]
 A. D, F, H, G, E, B, C, A
 B. D, F, G, A, B, C, H, E
 C. F, H, D, G, E, B, C, A
 D. D, F, H, G, E, B, C, A
33. The data structure used by level order traversal of binary tree is: [A]
 A. Queue
 B. Stack
 C. linked list
 D. none of the above

Chapter 6

Graphs

6.1. Introduction to Graphs:

Graph G is a pair (V, E) , where V is a finite set of vertices and E is a finite set of edges. We will often denote $n = |V|$, $e = |E|$.

A graph is generally displayed as figure 6.5.1, in which the vertices are represented by circles and the edges by lines.

An edge with an orientation (i.e., arrow head) is a directed edge, while an edge with no orientation is our undirected edge.

If all the edges in a graph are undirected, then the graph is an undirected graph. The graph in figure 6.5.1(a) is an undirected graph. If all the edges are directed; then the graph is a directed graph. The graph of figure 6.5.1(b) is a directed graph. A directed graph is also called as digraph. A graph G is connected if and only if there is a simple path between any two nodes in G .

A graph G is said to be complete if every node a in G is adjacent to every other node v in G . A complete graph with n nodes will have $n(n-1)/2$ edges. For example, Figure 6.5.1.(a) and figure 6.5.1.(d) are complete graphs.

A directed graph G is said to be connected, or strongly connected, if for each pair (u, v) for nodes in G there is a path from u to v and also a path from v to u . On the other hand, G is said to be unilaterally connected if for any pair (u, v) of nodes in G there is a path from u to v or a path from v to u . For example, the digraph shown in figure 6.5.1 (e) is strongly connected.

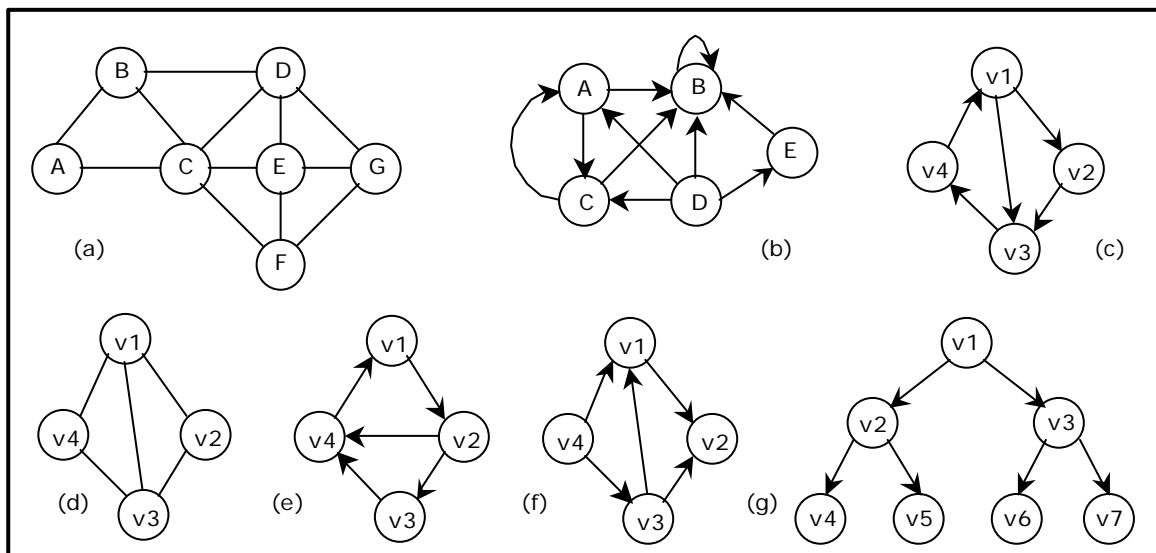


Figure 6.5.1 Various Graphs

We can assign weight function to the edges: $w_G(e)$ is a weight of edge $e \in E$. The graph which has such function assigned is called weighted graph.

The number of incoming edges to a vertex v is called in-degree of the vertex (denote $\text{indeg}(v)$). The number of outgoing edges from a vertex is called out-degree (denote $\text{outdeg}(v)$). For example, let us consider the digraph shown in figure 6.5.1(f),

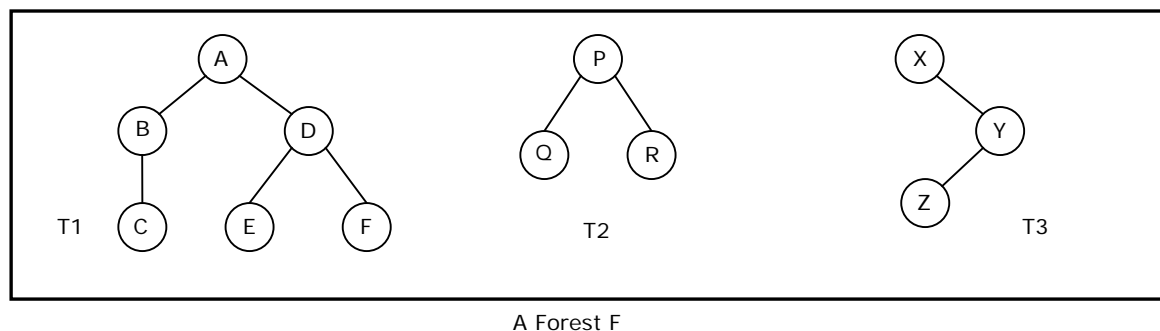
$$\begin{array}{ll} \text{indegree}(v_1) = 2 & \text{outdegree}(v_1) = 1 \\ \text{indegree}(v_2) = 2 & \text{outdegree}(v_2) = 0 \end{array}$$

A path is a sequence of vertices (v_1, v_2, \dots, v_k) , where for all i , $(v_i, v_{i+1}) \in E$. A path is simple if all vertices in the path are distinct. If there is a path containing one or more edges which starts from a vertex V_i and terminates into the same vertex then the path is known as a cycle. For example, there is a cycle in figure 6.5.1(a), figure 6.5.1(c) and figure 6.5.1(d).

If a graph (digraph) does not have any cycle then it is called **acyclic graph**. For example, the graphs of figure 6.5.1 (f) and figure 6.5.1 (g) are acyclic graphs.

A graph $G' = (V', E')$ is a sub-graph of graph $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E$.

A **Forest** is a set of disjoint trees. If we remove the root node of a given tree then it becomes forest. The following figure shows a forest F that consists of three trees T_1 , T_2 and T_3 .



A graph that has either self loop or parallel edges or both is called **multi-graph**.

Tree is a connected acyclic graph (there aren't any sequences of edges that go around in a loop). A spanning tree of a graph $G = (V, E)$ is a tree that contains all vertices of V and is a subgraph of G . A single graph can have multiple spanning trees.

Let T be a spanning tree of a graph G . Then

1. *Any two vertices in T are connected by a unique simple path.*
2. *If any edge is removed from T , then T becomes disconnected.*
3. *If we add any edge into T , then the new graph will contain a cycle.*
4. *Number of edges in T is $n-1$.*

6.2. Representation of Graphs:

There are two ways of representing digraphs. They are:

- Adjacency matrix.
- Adjacency List.
- Incidence matrix.

Adjacency matrix:

In this representation, the adjacency matrix of a graph G is a two dimensional $n \times n$ matrix, say $A = (a_{i,j})$, where

$$a_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed. This matrix is also called as Boolean matrix or bit matrix.

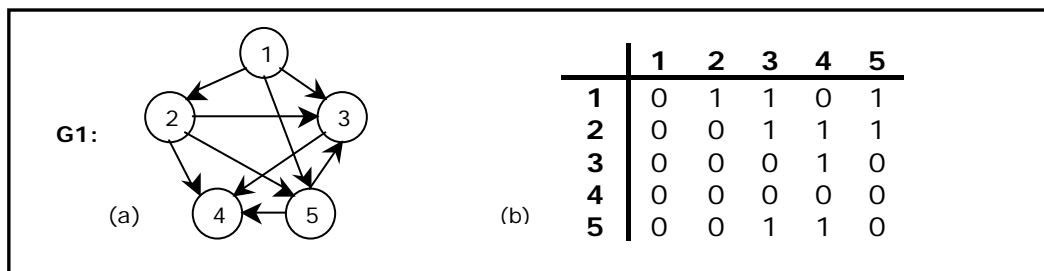


Figure 6.5.2. A graph and its Adjacency matrix

Figure 6.5.2(b) shows the adjacency matrix representation of the graph G_1 shown in figure 6.5.2(a). The adjacency matrix is also useful to store multigraph as well as weighted graph. In case of multigraph representation, instead of entry 0 or 1, the entry will be between number of edges between two vertices.

In case of weighted graph, the entries are weights of the edges between the vertices. The adjacency matrix for a weighted graph is called as cost adjacency matrix. Figure 6.5.3(b) shows the cost adjacency matrix representation of the graph G_2 shown in figure 6.5.3(a).

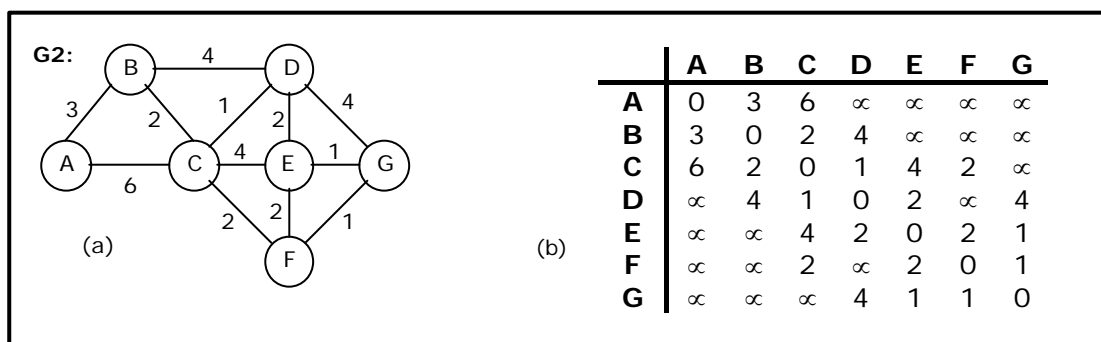


Figure 6.5.3 Weighted graph and its Cost adjacency matrix

Adjacency List:

In this representation, the n rows of the adjacency matrix are represented as n linked lists. An array $\text{Adj}[1, 2, \dots, n]$ of pointers where for $1 \leq v \leq n$, $\text{Adj}[v]$ points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements. For the graph G in figure 6.5.4(a), the adjacency list is shown in figure 6.5.4 (b).

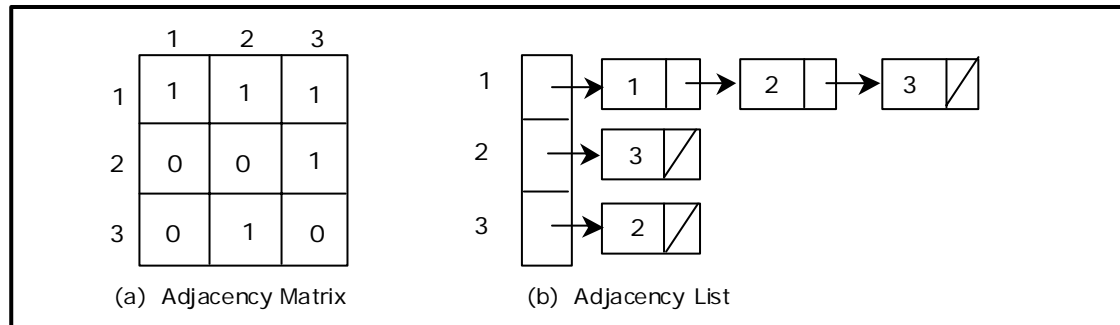


Figure 6.5.4 Adjacency matrix and adjacency list

Incidence Matrix:

In this representation, if G is a graph with n vertices, e edges and no self loops, then incidence matrix A is defined as an n by e matrix, say $A = (a_{i,j})$, where

$$a_{i,j} = \begin{cases} 1 & \text{if there is an edge } j \text{ incident to } v_i \\ 0 & \text{otherwise} \end{cases}$$

Here, n rows correspond to n vertices and e columns correspond to e edges. Such a matrix is called as vertex-edge incidence matrix or simply incidence matrix.

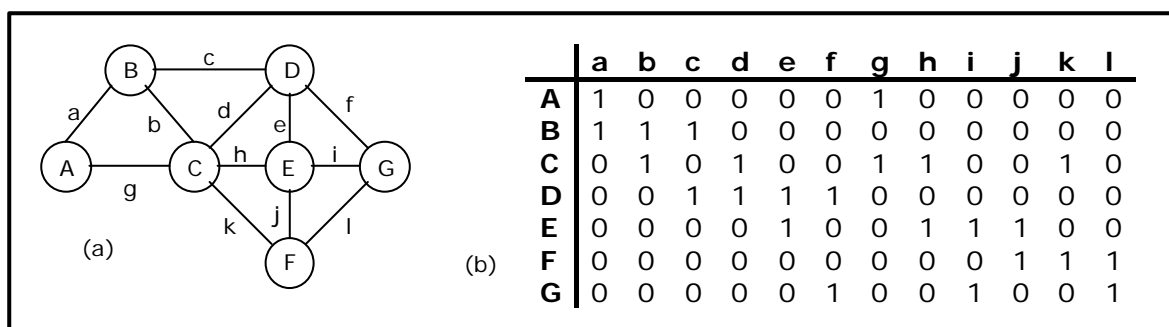


Figure 6.5.4 Graph and its incidence matrix

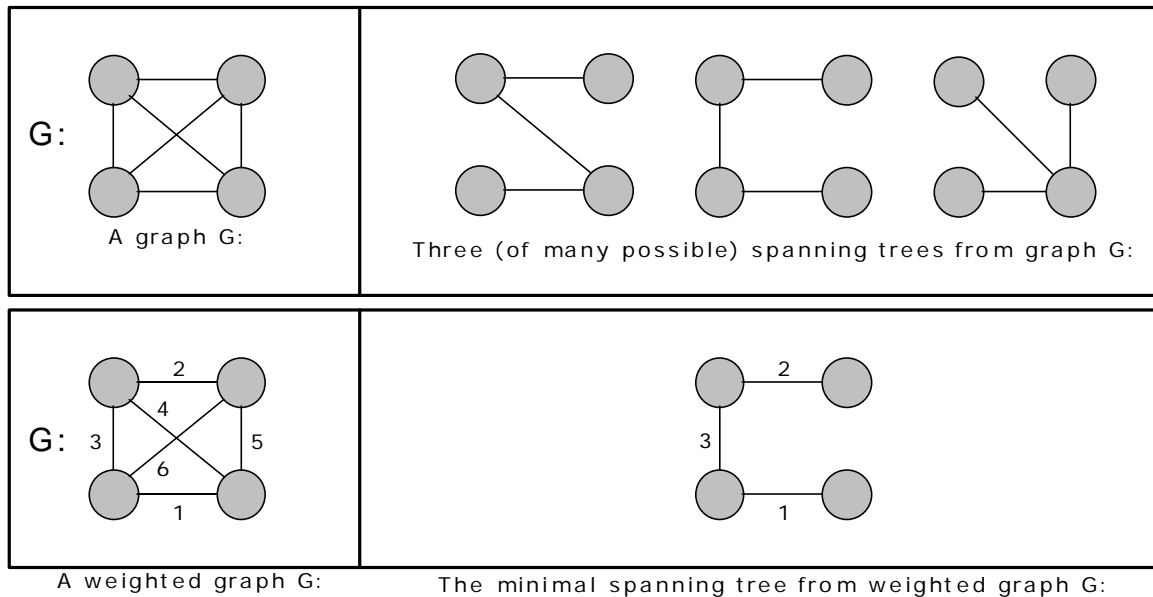
Figure 6.5.4(b) shows the incidence matrix representation of the graph G_1 shown in figure 6.5.4(a).

6.3. Minimum Spanning Tree (MST):

A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph. i.e., any connected graph will have a spanning tree.

Weight of a spanning tree $w(T)$ is the sum of weights of all edges in T . Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.

Example:



Let's consider a couple of real-world examples on minimum spanning tree:

- One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least cost paths with no cycles in this network, thereby connecting everyone at a minimum cost.
- Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

Minimum spanning tree, can be constructed using any of the following two algorithms:

1. Kruskal's algorithm and
2. Prim's algorithm.

Both algorithms differ in their methodology, but both eventually end up with the MST. *Kruskal's algorithm uses edges, and Prim's algorithm uses vertex connections in determining the MST. In Prim's algorithm at any instance of output it represents tree whereas in Kruskal's algorithm at any instance of output it may represent tree or not.*

6.3.1. Kruskal's Algorithm

This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an edge with the least weight in a MST).

Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until $(n - 1)$ edges have been added. Sometimes two or more edges may have the same cost.

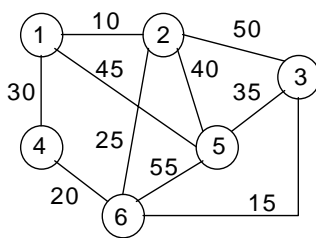
The order in which the edges are chosen, in this case, does not matter. Different MST's may result, but they will all have the same total cost, which will always be the minimum cost.

Kruskal's Algorithm for minimal spanning tree is as follows:

1. Make the tree T empty.
2. Repeat the steps 3, 4 and 5 as long as T contains less than $n - 1$ edges and E is not empty otherwise, proceed to step 6.
3. Choose an edge (v, w) from E of lowest cost.
4. Delete (v, w) from E.
5. If (v, w) does not create a cycle in T
 then Add (v, w) to T
 else discard (v, w)
6. If T contains fewer than $n - 1$ edges then print no spanning tree.

Example 1:

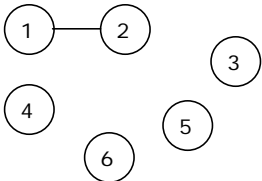
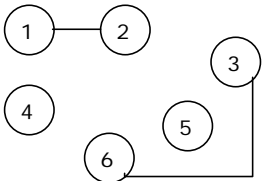
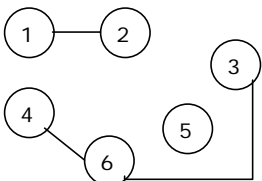
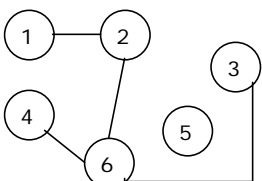
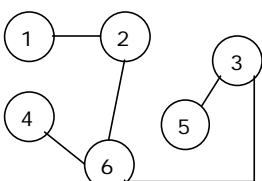
Construct the minimal spanning tree for the graph shown below:



Arrange all the edges in the increasing order of their costs:

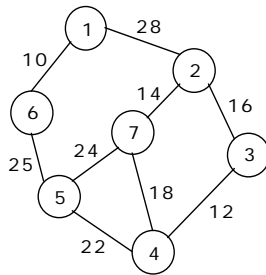
Cost	10	15	20	25	30	35	40	45	50	55
Edge	(1, 2)	(3, 6)	(4, 6)	(2, 6)	(1, 4)	(3, 5)	(2, 5)	(1, 5)	(2, 3)	(5, 6)

The stages in Kruskal's algorithm for minimal spanning tree is as follows:

EDGE	COST	STAGES IN KRUSKAL'S ALGORITHM	REMARKS
(1, 2)	10		The edge between vertices 1 and 2 is the first edge selected. It is included in the spanning tree.
(3, 6)	15		Next, the edge between vertices 3 and 6 is selected and included in the tree.
(4, 6)	20		The edge between vertices 4 and 6 is next included in the tree.
(2, 6)	25		The edge between vertices 2 and 6 is considered next and included in the tree.
(1, 4)	30	Reject	The edge between the vertices 1 and 4 is discarded as its inclusion creates a cycle.
(3, 5)	35		Finally, the edge between vertices 3 and 5 is considered and included in the tree built. This completes the tree. The cost of the minimal spanning tree is 105.

Example 2:

Construct the minimal spanning tree for the graph shown below:

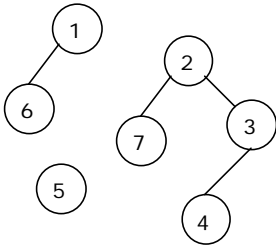
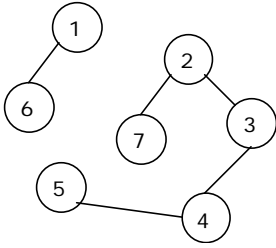
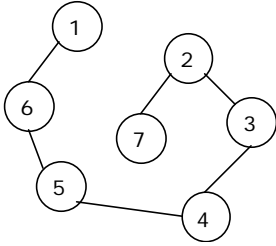
**Solution:**

Arrange all the edges in the increasing order of their costs:

Cost	10	12	14	16	18	22	24	25	28
Edge	(1, 6)	(3, 4)	(2, 7)	(2, 3)	(4, 7)	(4, 5)	(5, 7)	(5, 6)	(1, 2)

The stages in Kruskal's algorithm for minimal spanning tree is as follows:

EDGE	COST	STAGES IN KRUSKAL'S ALGORITHM	REMARKS
(1, 6)	10		The edge between vertices 1 and 6 is the first edge selected. It is included in the spanning tree.
(3, 4)	12		Next, the edge between vertices 3 and 4 is selected and included in the tree.
(2, 7)	14		The edge between vertices 2 and 7 is next included in the tree.

(2, 3)	16		The edge between vertices 2 and 3 is next included in the tree.
(4, 7)	18	Reject	The edge between the vertices 4 and 7 is discarded as its inclusion creates a cycle.
(4, 5)	22		The edge between vertices 4 and 7 is considered next and included in the tree.
(5, 7)	24	Reject	The edge between the vertices 5 and 7 is discarded as its inclusion creates a cycle.
(5, 6)	25		Finally, the edge between vertices 5 and 6 is considered and included in the tree built. This completes the tree. The cost of the minimal spanning tree is 99.

6.3.2. MINIMUM-COST SPANNING TREES: PRIM'S ALGORITHM

A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an Minimal cost spanning tree. Prim's algorithm is an example of a greedy algorithm.

Prim's Algorithm:

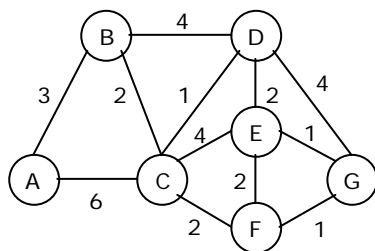
E is the set of edges in G . $\text{cost}[1:n, 1:n]$ is the cost adjacency matrix of an n vertex graph such that $\text{cost}[i, j]$ is either a positive real number or ∞ if no edge (i, j) exists. A minimum spanning tree is computed and stored as a set of edges in the array $t[1:n-1, 1:2]$. $(t[i, 1], t[i, 2])$ is an edge in the minimum-cost spanning tree. The final cost is returned.

Algorithm Prim (E, cost, n, t)

```
{
  Let  $(k, l)$  be an edge of minimum cost in  $E$ ;
   $\text{mincost} := \text{cost}[k, l]$ ;
   $t[1, 1] := k$ ;  $t[1, 2] := l$ ;
  for  $i := 1$  to  $n$  do // Initialize near
    if  $(\text{cost}[i, l] < \text{cost}[i, k])$  then  $\text{near}[i] := l$ ;
    else  $\text{near}[i] := k$ ;
   $\text{near}[k] := \text{near}[l] := 0$ ;
  for  $i := 2$  to  $n - 1$  do // Find  $n - 2$  additional edges for  $t$ .
  {
    Let  $j$  be an index such that  $\text{near}[j] \neq 0$  and
     $\text{cost}[j, \text{near}[j]]$  is minimum;
     $t[i, 1] := j$ ;  $t[i, 2] := \text{near}[j]$ ;
     $\text{mincost} := \text{mincost} + \text{cost}[j, \text{near}[j]]$ ;
     $\text{near}[j] := 0$ ;
    for  $k := 1$  to  $n$  do // Update  $\text{near}[]$ .
      if  $((\text{near}[k] \neq 0) \text{ and } (\text{cost}[k, \text{near}[k]] > \text{cost}[k, j]))$ 
        then  $\text{near}[k] := j$ ;
  }
  return  $\text{mincost}$ ;
}
```

EXAMPLE:

Use Prim's Algorithm to find a minimal spanning tree for the graph shown below starting with the vertex A.



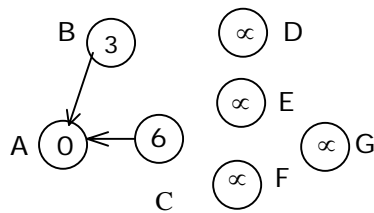
Solution:

The cost adjacency matrix is

$$\begin{pmatrix} 0 & 3 & 6 & \infty & \infty & \infty & \infty \\ 3 & 0 & 2 & 4 & \infty & \infty & \infty \\ 6 & 2 & 0 & 1 & 4 & 2 & \infty \\ \infty & 4 & 1 & 0 & 2 & \infty & 4 \\ \infty & \infty & 4 & 2 & 0 & 2 & 1 \\ \infty & \infty & 2 & \infty & 2 & 0 & 1 \\ \infty & \infty & \infty & 4 & 1 & 1 & 0 \end{pmatrix}$$

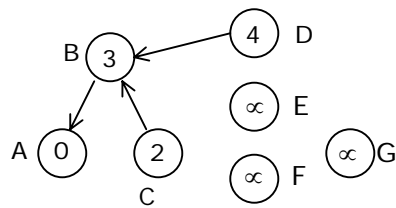
The stepwise progress of the prim's algorithm is as follows:

Step 1:



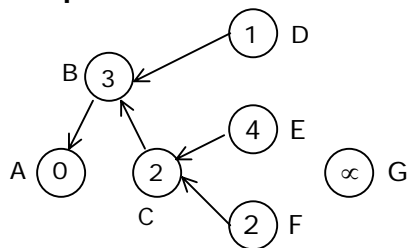
Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6	∞	∞	∞	∞
Next	*	A	A	A	A	A	A

Step 2:



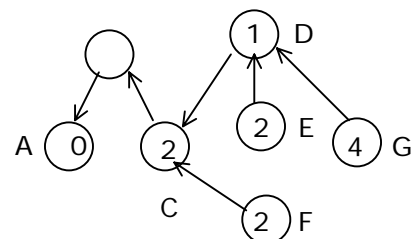
Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	2	4	∞	∞	∞
Next	*	A	B	B	A	A	A

Step 3:



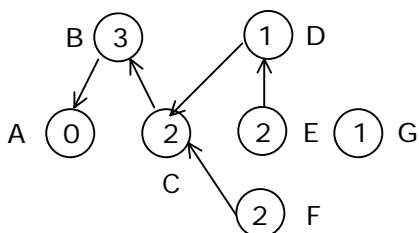
Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	2	1	4	2	∞
Next	*	A	B	C	C	C	A

Step 4:



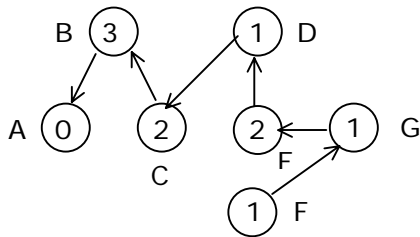
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	2	1	2	2	4
Next	*	A	B	C	D	C	D

Step 5:



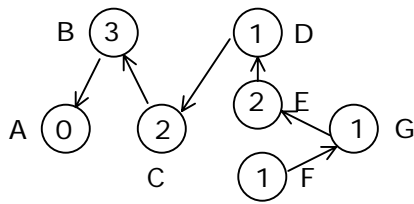
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	2	1	2	2	1
Next	*	A	B	C	D	C	E

Step 6:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	1	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

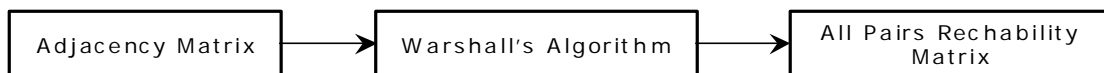
Step 7:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

6.4. Reachability Matrix (Warshall's Algorithm):

Warshall's algorithm requires knowing which edges exist and which does not. It doesn't need to know the lengths of the edges in the given directed graph. This information is conveniently displayed by adjacency matrix for the graph, in which a '1' indicates the existence of an edge and '0' indicates non-existence.



It begins with the adjacency matrix for the given graph, which is called A_0 , and then updates the matrix 'n' times, producing matrices called A_1, A_2, \dots, A_n and then stops.

In warshall's algorithm the matrix A_i contains information about the existence of i-paths. A one entry in the matrix A_i will correspond to the existence of i-paths and zero entry will correspond to non-existence. Thus when the algorithm stops, the final matrix A_n , contains the desired connectivity information.

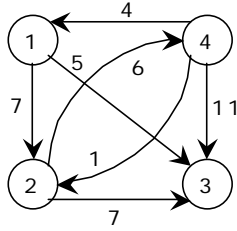
A one entry indicates a pair of vertices, which are connected and zero entry indicates a pair, which are not. This matrix is called a *reachability matrix* or *path matrix* for the graph. It is also called the *transitive closure* of the original adjacency matrix.

The update rule for computing A_i from A_{i-1} in warshall's algorithm is:

$$A_i[x, y] = A_{i-1}[x, y] \vee (A_{i-1}[x, i] \wedge A_{i-1}[i, y]) \quad \text{----} \quad (1)$$

Example 1:

Use warshall's algorithm to calculate the reachability matrix for the graph:



We begin with the adjacency matrix of the graph 'A₀'

$$A_0 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix} \end{matrix}$$

The first step is to compute 'A₁' matrix. To do so we will use the updating rule – (1).

Before doing so, we notice that only one entry in A₀ must remain one in A₁, since in Boolean algebra 1 + (anything) = 1. Since these are only nine zero entries in A₀, there are only nine entries in A₀ that need to be updated.

$$A_1[1, 1] = A_0[1, 1] \vee (A_0[1, 1] \wedge A_0[1, 1]) = 0 \vee (0 \wedge 0) = 0$$

$$A_1[1, 4] = A_0[1, 4] \vee (A_0[1, 1] \wedge A_0[1, 4]) = 0 \vee (0 \wedge 0) = 0$$

$$A_1[2, 1] = A_0[2, 1] \vee (A_0[2, 1] \wedge A_0[1, 1]) = 0 \vee (0 \wedge 0) = 0$$

$$A_1[2, 2] = A_0[2, 2] \vee (A_0[2, 1] \wedge A_0[1, 2]) = 0 \vee (0 \wedge 1) = 0$$

$$A_1[3, 1] = A_0[3, 1] \vee (A_0[3, 1] \wedge A_0[1, 1]) = 0 \vee (0 \wedge 0) = 0$$

$$A_1[3, 2] = A_0[3, 2] \vee (A_0[3, 1] \wedge A_0[1, 2]) = 0 \vee (0 \wedge 1) = 0$$

$$A_1[3, 3] = A_0[3, 3] \vee (A_0[3, 1] \wedge A_0[1, 3]) = 0 \vee (0 \wedge 1) = 0$$

$$A_1[3, 4] = A_0[3, 4] \vee (A_0[3, 1] \wedge A_0[1, 4]) = 0 \vee (0 \wedge 0) = 0$$

$$A_1[4, 4] = A_0[4, 4] \vee (A_0[4, 1] \wedge A_0[1, 4]) = 0 \vee (1 \wedge 0) = 0$$

$$A_1 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix} \end{matrix}$$

Next, A₂ must be calculated from A₁; but again we need to update the 0 entries,

$$A_2[1, 1] = A_1[1, 1] \vee (A_1[1, 2] \wedge A_1[2, 1]) = 0 \vee (1 \wedge 0) = 0$$

$$A_2[1, 4] = A_1[1, 4] \vee (A_1[1, 2] \wedge A_1[2, 4]) = 0 \vee (1 \wedge 1) = 1$$

$$A_2[2, 1] = A_1[2, 1] \vee (A_1[2, 2] \wedge A_1[2, 1]) = 0 \vee (0 \wedge 0) = 0$$

$$A_2[2, 2] = A_1[2, 2] \vee (A_1[2, 2] \wedge A_1[2, 2]) = 0 \vee (0 \wedge 0) = 0$$

$$A_2[3, 1] = A_1[3, 1] \vee (A_1[3, 2] \wedge A_1[2, 1]) = 0 \vee (0 \wedge 0) = 0$$

$$A_2[3, 2] = A_1[3, 2] \vee (A_1[3, 2] \wedge A_1[2, 2]) = 0 \vee (0 \wedge 0) = 0$$

$$A_2[3, 3] = A_1[3, 3] \vee (A_1[3, 2] \wedge A_1[2, 3]) = 0 \vee (0 \wedge 1) = 0$$

$$A_2[3, 4] = A_1[3, 4] \vee (A_1[3, 2] \wedge A_1[2, 4]) = 0 \vee (0 \wedge 1) = 0$$

$$A_2[4, 4] = A_1[4, 4] \vee (A_1[4, 2] \wedge A_1[2, 4]) = 0 \vee (1 \wedge 1) = 1$$

$$A_2 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{matrix}$$

This matrix has only seven 0 entries, and so to compute A_3 , we need to do only seven computations.

$$A_3[1, 1] = A_2[1, 1] \vee (A_2[1, 3] \wedge A_2[3, 1]) = 0 \vee (1 \wedge 0) = 0$$

$$A_3[2, 1] = A_2[2, 1] \vee (A_2[2, 3] \wedge A_2[3, 1]) = 0 \vee (1 \wedge 0) = 0$$

$$A_3[2, 2] = A_2[2, 2] \vee (A_2[2, 3] \wedge A_2[3, 2]) = 0 \vee (1 \wedge 0) = 0$$

$$A_3[3, 1] = A_2[3, 1] \vee (A_2[3, 3] \wedge A_2[3, 1]) = 0 \vee (0 \wedge 0) = 0$$

$$A_3[3, 2] = A_2[3, 2] \vee (A_2[3, 3] \wedge A_2[3, 2]) = 0 \vee (0 \wedge 0) = 0$$

$$A_3[3, 3] = A_2[3, 3] \vee (A_2[3, 3] \wedge A_2[3, 3]) = 0 \vee (0 \wedge 0) = 0$$

$$A_3[3, 4] = A_2[3, 4] \vee (A_2[3, 3] \wedge A_2[3, 4]) = 0 \vee (0 \wedge 0) = 0$$

$$A_3 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{matrix}$$

Once A_3 is calculated, we use the update rule to calculate A_4 and stop. This matrix is the reachability matrix for the graph.

$$A_4[1, 1] = A_3[1, 1] \vee (A_3[1, 4] \wedge A_3[4, 1]) = 0 \vee (1 \wedge 1) = 0 \vee 1 = 1$$

$$A_4[2, 1] = A_3[2, 1] \vee (A_3[2, 4] \wedge A_3[4, 1]) = 0 \vee (1 \wedge 1) = 0 \vee 1 = 1$$

$$A_4[2, 2] = A_3[2, 2] \vee (A_3[2, 4] \wedge A_3[4, 2]) = 0 \vee (1 \wedge 1) = 0 \vee 1 = 1$$

$$A_4[3, 1] = A_3[3, 1] \vee (A_3[3, 4] \wedge A_3[4, 1]) = 0 \vee (0 \wedge 1) = 0 \vee 0 = 0$$

$$A_4[3, 2] = A_3[3, 2] \vee (A_3[3, 4] \wedge A_3[4, 2]) = 0 \vee (0 \wedge 1) = 0 \vee 0 = 0$$

$$A_4[3, 3] = A_3[3, 3] \vee (A_3[3, 4] \wedge A_3[4, 3]) = 0 \vee (0 \wedge 1) = 0 \vee 0 = 0$$

$$A_4[3, 4] = A_3[3, 4] \vee (A_3[3, 4] \wedge A_3[4, 4]) = 0 \vee (0 \wedge 1) = 0 \vee 0 = 0$$

$$A_4 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{matrix}$$

Note that according to the algorithm vertex 3 is not reachable from itself 1. This is because as can be seen in the graph, there is no path from vertex 3 back to itself.

6.5. Traversing a Graph

Many graph algorithms require one to systematically examine the nodes and edges of a graph G . There are two standard ways to do this. They are:

- Breadth first traversal (BFT)
- Depth first traversal (DFT)

The BFT will use a queue as an auxiliary structure to hold nodes for future processing and the DFT will use a STACK.

During the execution of these algorithms, each node N of G will be in one of three states, called the *status* of N , as follows:

1. STATUS = 1 (Ready state): The initial state of the node N .
2. STATUS = 2 (Waiting state): The node N is on the QUEUE or STACK, waiting to be processed.
3. STATUS = 3 (Processed state): The node N has been processed.

Both BFS and DFS impose a tree (the BFS/DFS tree) on the structure of graph. So, we can compute a spanning tree in a graph. The computed spanning tree is not a minimum spanning tree. The spanning trees obtained using depth first search are called depth first spanning trees. The spanning trees obtained using breadth first search are called Breadth first spanning trees.

6.5.1. Breadth first search and traversal:

The general idea behind a breadth first traversal beginning at a starting node A is as follows. First we examine the starting node A . Then we examine all the neighbors of A . Then we examine all the neighbors of neighbors of A . And so on. We need to keep track of the neighbors of a node, and we need to guarantee that no node is processed more than once. This is accomplished by using a QUEUE to hold nodes that are waiting to be processed, and by using a field STATUS that tells us the current status of any node. The spanning trees obtained using BFS are called Breadth first spanning trees.

Breadth first traversal algorithm on graph G is as follows:

This algorithm executes a BFT on graph G beginning at a starting node A .

Initialize all nodes to the ready state (STATUS = 1).

1. Put the starting node A in QUEUE and change its status to the waiting state (STATUS = 2).
2. Repeat the following steps until QUEUE is empty:
 - a. Remove the front node N of QUEUE. Process N and change the status of N to the processed state (STATUS = 3).
 - b. Add to the rear of QUEUE all the neighbors of N that are in the ready state (STATUS = 1), and change their status to the waiting state (STATUS = 2).
3. Exit.

6.5.2. Depth first search and traversal:

Depth first search of undirected graph proceeds as follows: First we examine the starting node V. Next an unvisited vertex 'W' adjacent to 'V' is selected and a depth first search from 'W' is initiated. When a vertex 'U' is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited, which has an unvisited vertex 'W' adjacent to it and initiate a depth first search from W. The search terminates when no unvisited vertex can be reached from any of the visited ones.

This algorithm is similar to the inorder traversal of binary tree. DFT algorithm is similar to BFS except now use a STACK instead of the QUEUE. Again field STATUS is used to tell us the current status of a node.

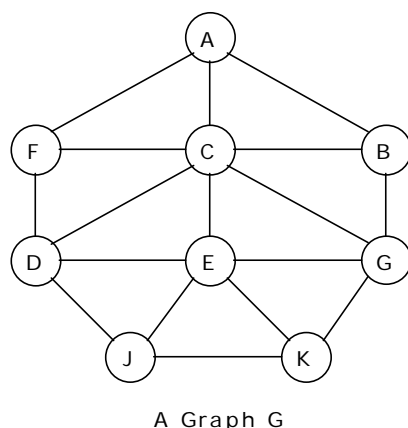
The algorithm for depth first traversal on a graph G is as follows.

This algorithm executes a DFT on graph G beginning at a starting node A.

1. Initialize all nodes to the ready state (STATUS = 1).
2. Push the starting node A into STACK and change its status to the waiting state (STATUS = 2).
3. Repeat the following steps until STACK is empty:
 - a. Pop the top node N from STACK. Process N and change the status of N to the processed state (STATUS = 3).
 - b. Push all the neighbors of N that are in the ready state (STATUS = 1), and change their status to the waiting state (STATUS = 2).
4. Exit.

Example 1:

Consider the graph shown below. Traverse the graph shown below in breadth first order and depth first order.



<i>Node</i>	<i>Adjacency List</i>
A	F, C, B
B	A, C, G
C	A, B, D, E, F, G
D	C, F, E, J
E	C, D, G, J, K
F	A, C, D
G	B, C, E, K
J	D, E, K
K	E, G, J

Adjacency list for graph G

Breadth-first search and traversal:

The steps involved in breadth first traversal are as follows:

Current Node	QUEUE	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	F C B	A	3	2	2	1	1	2	1	1	1
F	C B D	A F	3	2	2	2	1	3	1	1	1
C	B D E G	A F C	3	2	3	2	2	3	2	1	1
B	D E G	A F C B	3	3	3	2	2	3	2	1	1
D	E G J	A F C B D	3	3	3	3	2	3	2	2	1
E	G J K	A F C B D E	3	3	3	3	3	3	2	2	2
G	J K	A F C B D E G	3	3	3	3	3	3	3	2	2
J	K	A F C B D E G J	3	3	3	3	3	3	3	3	2
K	EMPTY	A F C B D E G J K	3	3	3	3	3	3	3	3	3

For the above graph the breadth first traversal sequence is: **A F C B D E G J K**.

Depth-first search and traversal:

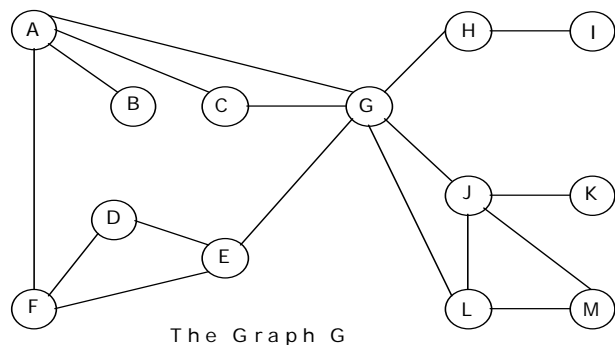
The steps involved in depth first traversal are as follows:

Current Node	Stack	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	B C F	A	3	2	2	1	1	2	1	1	1
F	B C D	A F	3	2	2	2	1	3	1	1	1
D	B C E J	A F D	3	2	2	3	2	3	1	2	1
J	B C E K	A F D J	3	2	2	3	2	3	1	3	2
K	B C E G	A F D J K	3	2	2	3	2	3	2	3	3
G	B C E	A F D J K G	3	2	2	3	2	3	3	3	3
E	B C	A F D J K G E	3	2	2	3	3	3	3	3	3
C	B	A F D J K G E C	3	2	3	3	3	3	3	3	3
B	EMPTY	A F D J K G E C B	3	3	3	3	3	3	3	3	3

For the above graph the depth first traversal sequence is: **A F D J K G E C B**.

Example 2:

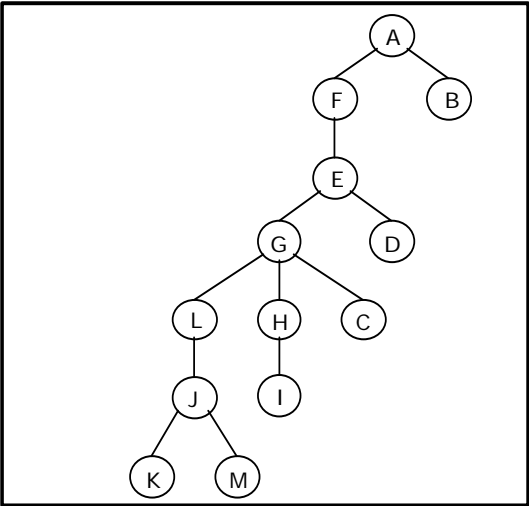
Traverse the graph shown below in breadth first order, depth first order and construct the breadth first and depth first spanning trees.



Node	Adjacency List
A	F, B, C, G
B	A
C	A, G
D	E, F
E	G, D, F
F	A, E, D
G	A, L, E, H, J, C
H	G, I
I	H
J	G, L, K, M
K	J
L	G, J, M
M	G, J, L

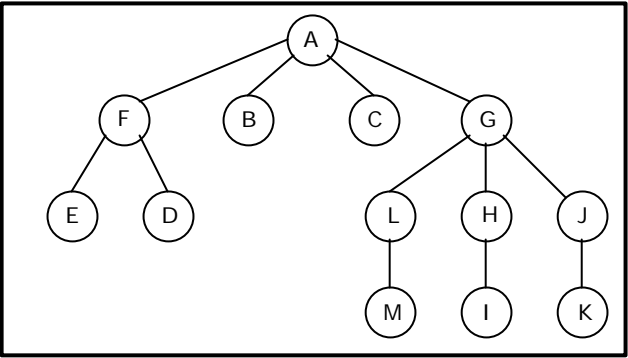
The Adjacency list for the graph G

If the depth first traversal is initiated from vertex A, then the vertices of graph G are visited in the order: **A F E G L J K M H I C D B**. The depth first spanning tree is shown in the figure given below:



Depth first Traversal

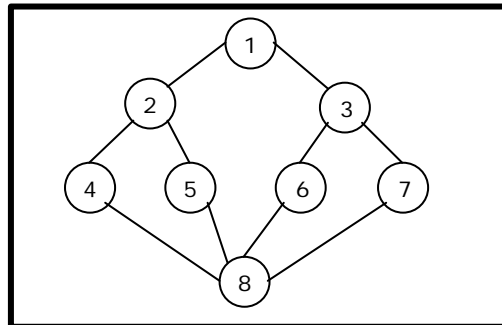
If the breadth first traversal is initiated from vertex A, then the vertices of graph G are visited in the order: **A F B C G E D L H J M I K**. The breadth first spanning tree is shown in the figure given below:



Breadth first traversal

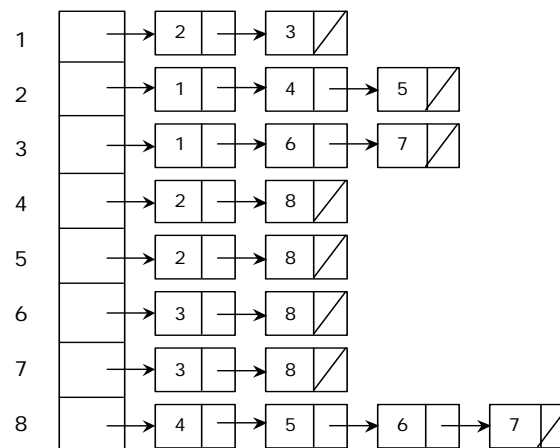
Example 3:

Traverse the graph shown below in breadth first order, depth first order and construct the breadth first and depth first spanning trees.



Graph G

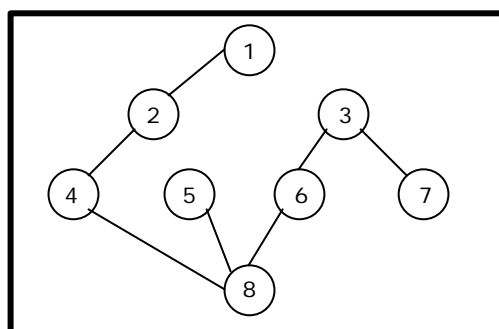
Head Nodes



Adjacency list for graph G

Depth first search and traversal:

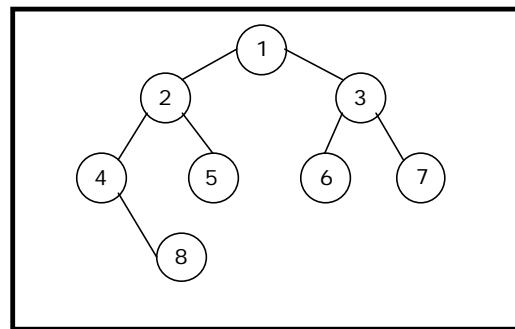
If the depth first is initiated from vertex 1, then the vertices of graph G are visited in the order: 1, 2, 4, 8, 5, 6, 3, 7. The depth first spanning tree is as follows:



Depth First Spanning Tree

Breadth first search and traversal:

If the breadth first search is initiated from vertex 1, then the vertices of G are visited in the order: 1, 2, 3, 4, 5, 6, 7, 8. The breadth first spanning tree is as follows:

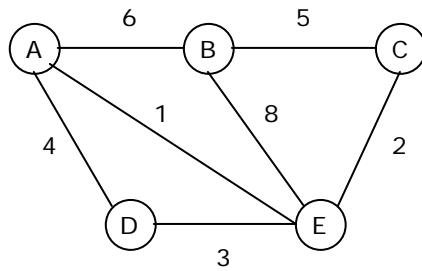


Breadth First Spanning Tree

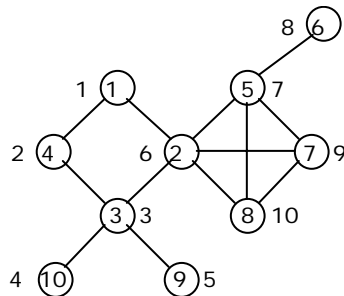
EXERCISES

1. Show that the sum of degrees of all vertices in an undirected graph is twice the number of edges.
2. Show that the number of vertices of odd degree in a finite graph is even.
3. How many edges are contained in a complete graph of "n" vertices.
4. Show that the number of spanning trees in a complete graph of "n" vertices is $2^{n-1} - 1$.
5. Prove that the edges explored by a breadth first or depth first traversal of a connected graph form a tree.
6. Explain how existence of a cycle in an undirected graph may be detected by traversing the graph in a depth first manner.
7. Write a "C" function to generate the incidence matrix of a graph from its adjacency matrix.
8. Give an example of a connected directed graph so that a depth first traversal of that graph yields a forest and not a spanning tree of the graph.
9. Rewrite the algorithms "BFSearch" and "DFSearch" so that it works on adjacency matrix representation of graphs.
10. Write a "C" function to find out whether there is a path between any two vertices in a graph (i.e. to compute the transitive closure matrix of a graph)
11. Write a "C" function to delete an existing edge from a graph represented by an adjacency list.
12. Construct a weighted graph for which the minimal spanning trees produced by Kruskal's algorithm and Prim's algorithm are different.

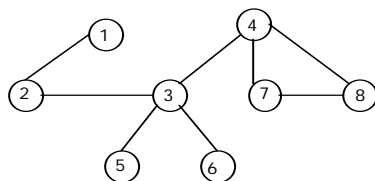
13. Describe the algorithm to find a minimum spanning tree T of a weighted graph G . Find the minimum spanning tree T of the graph shown below.



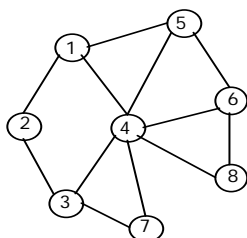
14. For the graph given below find the following:
- Linked representation of the graph.
 - Adjacency list.
 - Depth first spanning tree.
 - Breadth first spanning tree.
 - Minimal spanning tree using Kruskal's and Prim's algorithms.



15. For the graph given below find the following:
- Linked representation of the graph.
 - Adjacency list.
 - Depth first spanning tree.
 - Breadth first spanning tree.
 - Minimal spanning tree using Kruskal's and Prim's algorithms.

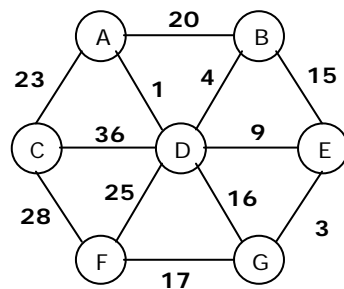


16. For the graph given below find the following:
- Linked representation of the graph.
 - Adjacency list.
 - Depth first spanning tree.
 - Breadth first spanning tree.
 - Minimal spanning tree using Kruskal's and Prim's algorithms.



Multiple Choice Questions

1. How can the graphs be represented? [D]
 A. Adjacency matrix C. Incidence matrix
 B. Adjacency list D. All of the above
2. The depth-first traversal in graph is analogous to tree traversal: [C]
 A. In-order C. Pre-order
 B. Post-order D. Level order
3. The children of a same parent node are called as: [C]
 A. adjacent node C. Sibblings
 B. non-leaf node D. leaf node
4. Complete graphs with n nodes will have _____ edges. [C]
 A. $n - 1$ C. $n(n-1)/2$
 B. $n/2$ D. $(n - 1)/2$
5. A graph with no cycle is called as: [C]
 A. Sub-graph C. Acyclic graph
 B. Directed graph D. none of the above
6. The maximum number of nodes at any level is: [B]
 A. n C. $n + 1$
 B. 2^n D. $2n$



Node	Adjacency List
A	B C D
B	A D E
C	A D F
D	A B C E F G
E	B D G
F	C D G
G	F D E

FIGURE 1 and its adjacency list

7. For the figure 1 shown above, the depth first spanning tree visiting sequence is: [B]
 A. A B C D E F G C. A B C D E F G
 B. A B D C F G E D. none of the above
8. For the figure 1 shown above, the breadth first spanning tree visiting sequence is: [B]
 A. A B D C F G E C. A B C D E F G
 B. A B C D E F G D. none of the above
9. Which is the correct order for Kruskal's minimum spanning tree algorithm to add edges to the minimum spanning tree for the figure 1 shown above: [B]
 A. (A, B) then (A, C) then (A, D) then (D, E) then (C, F) then (D, G)
 B. (A, D) then (E, G) then (B, D) then (D, E) then (F, G) then (A, C)
 C. both A and B
 D. none of the above
10. For the figure 1 shown above, the cost of the minimal spanning tree is: [A]
 A. 57 C. 48
 B. 68 D. 32

11. A simple graph has no loops. What other property must a simple graph have? [D]
 A. It must be directed. C. It must have at least one vertex.
 B. It must be undirected. D. It must have no multiple edges.
12. Suppose you have a directed graph representing all the flights that an airline flies. What algorithm might be used to find the best sequence of connections from one city to another? [D]
 A. Breadth first search. C. A cycle-finding algorithm.
 B. Depth first search. D. A shortest-path algorithm.
13. If G is an directed graph with 20 vertices, how many boolean values will be needed to represent G using an adjacency matrix? [D]
 A. 20 C. 200
 B. 40 D. 400
14. Which graph representation allows the most efficient determination of the existence of a particular edge in a graph? [B]
 A. An adjacency matrix. C. Incidence matrix
 B. Edge lists. D. none of the above
15. What graph traversal algorithm uses a queue to keep track of vertices which need to be processed? [A]
 A. Breadth-first search. C Level order search
 B. Depth-first search. D. none of the above
16. What graph traversal algorithm uses a stack to keep track of vertices which need to be processed? [B]
 A. Breadth-first search. C Level order search
 B. Depth-first search. D. none of the above
17. What is the expected number of operations needed to loop through all the edges terminating at a particular vertex given an adjacency matrix representation of the graph? (Assume n vertices are in the graph and m edges terminate at the desired node.) [D]
 A. $O(m)$ C. $O(m^2)$
 B. $O(n)$ D. $O(n^2)$
18. What is the expected number of operations needed to loop through all the edges terminating at a particular vertex given an adjacency list representation of the graph? (Assume n vertices are in the graph and m edges terminate at the desired node.) [A]
 A. $O(m)$ C. $O(m^2)$
 B. $O(n)$ D. $O(n^2)$
19. [B]

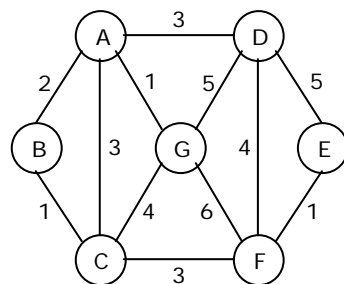


FIGURE 3

For the figure 3, starting at vertex A, which is a correct order for Prim's minimum spanning tree algorithm to add edges to the minimum spanning tree?

Chapter 7

Searching and Sorting

There are basically two aspects of computer programming. One is data organization also commonly called as data structures. Till now we have seen about data structures and the techniques and algorithms used to access them. The other part of computer programming involves choosing the appropriate algorithm to solve the problem. Data structures and algorithms are linked each other. After developing programming techniques to represent information, it is logical to proceed to manipulate it. This chapter introduces this important aspect of problem solving.

Searching is used to find the location where an element is available. There are two types of search techniques. They are:

1. Linear or sequential search
2. Binary search

Sorting allows an efficient arrangement of elements within a given data structure. It is a way in which the elements are organized systematically for some purpose. For example, a dictionary in which words are arranged in alphabetical order and telephone directory in which the subscriber names are listed in alphabetical order. There are many sorting techniques out of which we study the following.

1. Bubble sort
2. Quick sort
3. Selection sort and
4. Heap sort

There are two types of sorting techniques:

1. Internal sorting
2. External sorting

If all the elements to be sorted are present in the main memory then such sorting is called **internal sorting** on the other hand, if some of the elements to be sorted are kept on the secondary storage, it is called **external sorting**. Here we study only internal sorting techniques.

7.1. Linear Search:

This is the simplest of all searching techniques. In this technique, an ordered or unordered list will be searched one by one from the beginning until the desired element is found. If the desired element is found in the list then the search is successful otherwise unsuccessful.

Suppose there are 'n' elements organized sequentially on a List. The number of comparisons required to retrieve an element from the list, purely depends on where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two comparisons are necessary and so on. On an average you need $[(n+1)/2]$ comparison's to search an element. If search is not successful, you would need 'n' comparisons.

The time complexity of linear search is $O(n)$.

Algorithm:

Let array a[n] stores n elements. Determine whether element 'x' is present or not.

```
linsrch(a[n], x)
{
    index = 0;
    flag = 0;
    while (index < n) do
    {
        if (x == a[index])
        {
            flag = 1;
            break;
        }
        index ++;
    }
    if(flag == 1)
        printf("Data found at %d position", index);
    else
        printf("data not found");
}
```

Example 1:

Suppose we have the following unsorted list: 45, 39, 8, 54, 77, 38, 24, 16, 4, 7, 9, 20

If we are searching for:	45, we'll look at 1 element before success
	39, we'll look at 2 elements before success
	8, we'll look at 3 elements before success
	54, we'll look at 4 elements before success
	77, we'll look at 5 elements before success
	38 we'll look at 6 elements before success
	24, we'll look at 7 elements before success
	16, we'll look at 8 elements before success
	4, we'll look at 9 elements before success
	7, we'll look at 10 elements before success
	9, we'll look at 11 elements before success
	20, we'll look at 12 elements before success

For any element not in the list, we'll look at 12 elements before failure.

Example 2:

Let us illustrate linear search on the following 9 elements:

Index	0	1	2	3	4	5	6	7	8
Elements	-15	-6	0	7	9	23	54	82	101

Searching different elements is as follows:

1. Searching for x = 7 Search successful, data found at 3rd position.
2. Searching for x = 82 Search successful, data found at 7th position.
3. Searching for x = 42 Search un-successful, data not found.

7.1.1. A non-recursive program for Linear Search:

```
# include <stdio.h>
# include <conio.h>

main()
{
    int number[25], n, data, i, flag = 0;
    clrscr();
    printf("\n Enter the number of elements: ");
    scanf("%d", &n);
    printf("\n Enter the elements: ");
    for(i = 0; i < n; i++)
        scanf("%d", &number[i]);
    printf("\n Enter the element to be Searched: ");
    scanf("%d", &data);
    for( i = 0; i < n; i++)
    {
        if(number[i] == data)
        {
            flag = 1;
            break;
        }
    }
    if(flag == 1)
        printf("\n Data found at location: %d", i+1);
    else
        printf("\n Data not found ");
}
```

7.1.2. A Recursive program for linear search:

```
# include <stdio.h>
# include <conio.h>

void linear_search(int a[], int data, int position, int n)
{
    if(position < n)
```

```

    {
        if(a[position] == data)
            printf("\n Data Found at %d ", position);
        else
            linear_search(a, data, position + 1, n);
    }
    else
        printf("\n Data not found");
}

void main()
{
    int a[25], i, n, data;
    clrscr();
    printf("\n Enter the number of elements: ");
    scanf("%d", &n);
    printf("\n Enter the elements: ");
    for(i = 0; i < n; i++)
    {
        scanf("%d", &a[i]);
    }
    printf("\n Enter the element to be seached: ");
    scanf("%d", &data);
    linear_search(a, data, 0, n);
    getch();
}

```

7.2. BINARY SEARCH

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \dots < x_n$. When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that $a[j] = x$ (successful search). If 'x' is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key $a[mid]$, and compare 'x' with $a[mid]$. If $x = a[mid]$ then the desired record has been found. If $x < a[mid]$ then 'x' must be in that portion of the file that precedes $a[mid]$. Similarly, if $a[mid] > x$, then further search is only necessary in that part of the file which follows $a[mid]$.

If we use recursive procedure of finding the middle key $a[mid]$ of the un-searched portion of a file, then every un-successful comparison of 'x' with $a[mid]$ will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved after each comparison between 'x' and $a[mid]$, and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$.

Algorithm:

Let array $a[n]$ of elements in increasing order, $n \geq 0$, determine whether 'x' is present, and if so, set j such that $x = a[j]$ else return 0.


```

binsrch(a[], n, x)
{
    low = 1; high = n;
    while (low ≤ high) do
    {
        mid = ⌊ (low + high)/2 ⌋
        if (x < a[mid])
            high = mid - 1;
        else if (x > a[mid])
            low = mid + 1;
        else return mid;
    }
    return 0;
}

```

low and *high* are integer variables such that each time through the loop either '*x*' is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if '*x*' is not present.

Example 1:

Let us illustrate binary search on the following 12 elements:

Index	1	2	3	4	5	6	7	8	9	10	11	12
Elements	4	7	8	9	16	20	24	38	39	45	54	77

If we are searching for *x* = 4: (This needs 3 comparisons)

low = 1, high = 12, mid = $13/2 = 6$, check 20

low = 1, high = 5, mid = $6/2 = 3$, check 8

low = 1, high = 2, mid = $3/2 = 1$, check 4, **found**

If we are searching for *x* = 7: (This needs 4 comparisons)

low = 1, high = 12, mid = $13/2 = 6$, check 20

low = 1, high = 5, mid = $6/2 = 3$, check 8

low = 1, high = 2, mid = $3/2 = 1$, check 4

low = 2, high = 2, mid = $4/2 = 2$, check 7, **found**

If we are searching for *x* = 8: (This needs 2 comparisons)

low = 1, high = 12, mid = $13/2 = 6$, check 20

low = 1, high = 5, mid = $6/2 = 3$, check 8, **found**

If we are searching for *x* = 9: (This needs 3 comparisons)

low = 1, high = 12, mid = $13/2 = 6$, check 20

low = 1, high = 5, mid = $6/2 = 3$, check 8

low = 4, high = 5, mid = $9/2 = 4$, check 9, **found**

If we are searching for *x* = 16: (This needs 4 comparisons)

low = 1, high = 12, mid = $13/2 = 6$, check 20

low = 1, high = 5, mid = $6/2 = 3$, check 8

low = 4, high = 5, mid = $9/2 = 4$, check 9

low = 5, high = 5, mid = $10/2 = 5$, check 16, **found**

If we are searching for *x* = 20: (This needs 1 comparison)

low = 1, high = 12, mid = $13/2 = 6$, check 20, **found**

If we are searching for $x = 24$: (This needs 3 comparisons)
low = 1, high = 12, mid = $13/2 = 6$, check 20
low = 7, high = 12, mid = $19/2 = 9$, check 39
low = 7, high = 8, mid = $15/2 = 7$, check 24, **found**

If we are searching for $x = 38$: (This needs 4 comparisons)
low = 1, high = 12, mid = $13/2 = 6$, check 20
low = 7, high = 12, mid = $19/2 = 9$, check 39
low = 7, high = 8, mid = $15/2 = 7$, check 24
low = 8, high = 8, mid = $16/2 = 8$, check 38, **found**

If we are searching for $x = 39$: (This needs 2 comparisons)
low = 1, high = 12, mid = $13/2 = 6$, check 20
low = 7, high = 12, mid = $19/2 = 9$, check 39, **found**

If we are searching for $x = 45$: (This needs 4 comparisons)
low = 1, high = 12, mid = $13/2 = 6$, check 20
low = 7, high = 12, mid = $19/2 = 9$, check 39
low = 10, high = 12, mid = $22/2 = 11$, check 54
low = 10, high = 10, mid = $20/2 = 10$, check 45, **found**

If we are searching for $x = 54$: (This needs 3 comparisons)
low = 1, high = 12, mid = $13/2 = 6$, check 20
low = 7, high = 12, mid = $19/2 = 9$, check 39
low = 10, high = 12, mid = $22/2 = 11$, check 54, **found**

If we are searching for $x = 77$: (This needs 4 comparisons)
low = 1, high = 12, mid = $13/2 = 6$, check 20
low = 7, high = 12, mid = $19/2 = 9$, check 39
low = 10, high = 12, mid = $22/2 = 11$, check 54
low = 12, high = 12, mid = $24/2 = 12$, check 77, **found**

The number of comparisons necessary by search element:

20 – requires 1 comparison;
8 and 39 – requires 2 comparisons;
4, 9, 24, 54 – requires 3 comparisons and
7, 16, 38, 45, 77 – requires 4 comparisons

Summing the comparisons, needed to find all twelve items and dividing by 12, yielding $37/12$ or approximately 3.08 comparisons per successful search on the average.

Example 2:

Let us illustrate binary search on the following 9 elements:

Index	0	1	2	3	4	5	6	7	8
Elements	-15	-6	0	7	9	23	54	82	101

Solution:

The number of comparisons required for searching different elements is as follows:

1. If we are searching for $x = 101$: (Number of comparisons = 4)

low	high	mid
1	9	5
6	9	7
8	9	8
9	9	9

found

2. Searching for $x = 82$: (Number of comparisons = 3)

low	high	mid
1	9	5
6	9	7
8	9	8

found

3. Searching for $x = 42$: (Number of comparisons = 4)

low	high	mid
1	9	5
6	9	7
6	6	6
7	6	not found

4. Searching for $x = -14$: (Number of comparisons = 3)

low	high	mid
1	9	5
1	4	2
1	1	1
2	1	not found

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

Index	1	2	3	4	5	6	7	8	9
Elements	-15	-6	0	7	9	23	54	82	101
Comparisons	3	2	3	4	1	3	2	3	4

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding $25/9$ or approximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x .

If $x < a(1)$, $a(1) < x < a(2)$, $a(2) < x < a(3)$, $a(5) < x < a(6)$, $a(6) < x < a(7)$ or $a(7) < x < a(8)$ the algorithm requires 3 element comparisons to determine that ' x ' is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons.

Thus the average number of element comparisons for an unsuccessful search is:

$$(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 = 3.4$$

Time Complexity:

The time complexity of binary search in a successful search is $O(\log n)$ and for an unsuccessful search is $O(\log n)$.

7.2.1. A non-recursive program for binary search:

```
# include <stdio.h>
# include <conio.h>

main()
{
    int number[25], n, data, i, flag = 0, low, high, mid;
    clrscr();
    printf("\n Enter the number of elements: ");
    scanf("%d", &n);
    printf("\n Enter the elements in ascending order: ");
    for(i = 0; i < n; i++)
        scanf("%d", &number[i]);
    printf("\n Enter the element to be searched: ");
    scanf("%d", &data);
    low = 0; high = n-1;
    while(low <= high)
    {
        mid = (low + high)/2;
        if(number[mid] == data)
        {
            flag = 1;
            break;
        }
        else
        {
            if(data < number[mid])
                high = mid - 1;
            else
                low = mid + 1;
        }
    }
    if(flag == 1)
        printf("\n Data found at location: %d", mid + 1);
    else
        printf("\n Data Not Found ");
}
```

7.2.2. A recursive program for binary search:

```
# include <stdio.h>
# include <conio.h>

void bin_search(int a[], int data, int low, int high)
{
    int mid ;
    if( low <= high)
    {
        mid = (low + high)/2;
        if(a[mid] == data)
            printf("\n Element found at location: %d ", mid + 1);
        else
        {
            if(data < a[mid])
                bin_search(a, data, low, mid-1);
            else

```

```

        bin_search(a, data, mid+1, high);
    }
}
else
    printf("\n Element not found");
}
void main()
{
    int a[25], i, n, data;
    clrscr();
    printf("\n Enter the number of elements: ");
    scanf("%d", &n);
    printf("\n Enter the elements in ascending order: ");
    for(i = 0; i < n; i++)
        scanf("%d", &a[i]);
    printf("\n Enter the element to be searched: ");
    scanf("%d", &data);
    bin_search(a, data, 0, n-1);
    getch();
}

```

7.3. Bubble Sort:

The bubble sort is easy to understand and program. The basic idea of bubble sort is to pass through the file sequentially several times. In each pass, we compare each element in the file with its successor i.e., $X[i]$ with $X[i+1]$ and interchange two element when they are not in proper order. We will illustrate this sorting technique by taking a specific example. Bubble sort is also called as exchange sort.

Example:

Consider the array $x[n]$ which is stored in memory as shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]
33	44	22	11	66	55

Suppose we want our array to be stored in ascending order. Then we pass through the array 5 times as described below:

Pass 1: (first element is compared with all other elements).

We compare $X[i]$ and $X[i+1]$ for $i = 0, 1, 2, 3$, and 4 , and interchange $X[i]$ and $X[i+1]$ if $X[i] > X[i+1]$. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	Remarks
33	44	22	11	66	55	
	22	44				
		11	44			
			44	66		
				55	66	
33	22	11	44	55	66	

The biggest number 66 is moved to (bubbled up) the right most position in the array.

Pass 2: (second element is compared).

We repeat the same process, but this time we don't include X[5] into our comparisons. i.e., we compare X[i] with X[i+1] for i=0, 1, 2, and 3 and interchange X[i] and X[i+1] if $X[i] > X[i+1]$. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	Remarks
33	22	11	44	55	
22	33				
	11	33			
		33	44		
			44	55	
22	11	33	44	55	

The second biggest number 55 is moved now to X[4].

Pass 3: (third element is compared).

We repeat the same process, but this time we leave both X[4] and X[5]. By doing this, we move the third biggest number 44 to X[3].

X[0]	X[1]	X[2]	X[3]	Remarks
22	11	33	44	
11	22			
	22	33		
		33	44	
11	22	33	44	

Pass 4: (fourth element is compared).

We repeat the process leaving X[3], X[4], and X[5]. By doing this, we move the fourth biggest number 33 to X[2].

X[0]	X[1]	X[2]	Remarks
11	22	33	
11	22		
	22	33	

Pass 5: (fifth element is compared).

We repeat the process leaving X[2], X[3], X[4], and X[5]. By doing this, we move the fifth biggest number 22 to X[1]. At this time, we will have the smallest number 11 in X[0]. Thus, we see that we can sort the array of size 6 in 5 passes.

For an array of size n, we required (n-1) passes.

7.3.1. Program for Bubble Sort:

```
#include <stdio.h>
#include <conio.h>
void bubblesort(int x[], int n)
{
    int i, j, temp;
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n-i-1 ; j++)
        {
            if (x[j] > x[j+1])
            {
                temp = x[j];
                x[j] = x[j+1];
                x[j+1] = temp;
            }
        }
    }
}

main()
{
    int i, n, x[25];
    clrscr();
    printf("\n Enter the number of elements: ");
    scanf("%d", &n);
    printf("\n Enter Data:");
    for(i = 0; i < n ; i++)
        scanf("%d", &x[i]);
    bubblesort(x, n);
    printf ("\n Array Elements after sorting: ");
    for (i = 0; i < n; i++)
        printf ("%5d", x[i]);
}
```

Time Complexity:

The bubble sort method of sorting an array of size n requires $(n-1)$ passes and $(n-1)$ comparisons on each pass. Thus the total number of comparisons is $(n-1) * (n-1) = n^2 - 2n + 1$, which is $O(n^2)$. Therefore bubble sort is very inefficient when there are more elements to sorting.

7.4. Selection Sort:

Selection sort will not require no more than $n-1$ interchanges. Suppose x is an array of size n stored in memory. The selection sort algorithm first selects the smallest element in the array x and place it at array position 0; then it selects the next smallest element in the array x and place it at array position 1. It simply continues this procedure until it places the biggest element in the last position of the array.

The array is passed through $(n-1)$ times and the smallest element is placed in its respective position in the array as detailed below:

Pass 1: Find the location j of the smallest element in the array $x[0], x[1], \dots, x[n-1]$, and then interchange $x[j]$ with $x[0]$. Then $x[0]$ is sorted.

Pass 2: Leave the first element and find the location j of the smallest element in the sub-array $x[1], x[2], \dots, x[n-1]$, and then interchange $x[1]$ with $x[j]$. Then $x[0], x[1]$ are sorted.

Pass 3: Leave the first two elements and find the location j of the smallest element in the sub-array $x[2], x[3], \dots, x[n-1]$, and then interchange $x[2]$ with $x[j]$. Then $x[0], x[1], x[2]$ are sorted.

Pass (n-1): Find the location j of the smaller of the elements $x[n-2]$ and $x[n-1]$, and then interchange $x[j]$ and $x[n-2]$. Then $x[0], x[1], \dots, x[n-2]$ are sorted. Of course, during this pass $x[n-1]$ will be the biggest element and so the entire array is sorted.

Time Complexity:

In general we prefer selection sort in case where the insertion sort or the bubble sort requires exclusive swapping. In spite of superiority of the selection sort over bubble sort and the insertion sort (there is significant decrease in run time), its efficiency is also $O(n^2)$ for n data items.

Example:

Let us consider the following example with 9 elements to analyze selection Sort:

1	2	3	4	5	6	7	8	9	Remarks
65	70	75	80	50	60	55	85	45	find the first smallest element
i								j	swap $a[i]$ & $a[j]$
45	70	75	80	50	60	55	85	65	find the second smallest element
	i			j					swap $a[i]$ and $a[j]$
45	50	75	80	70	60	55	85	65	Find the third smallest element
		i				j			swap $a[i]$ and $a[j]$
45	50	55	80	70	60	75	85	65	Find the fourth smallest element
			i		j				swap $a[i]$ and $a[j]$
45	50	55	60	70	80	75	85	65	Find the fifth smallest element
				i				j	swap $a[i]$ and $a[j]$
45	50	55	60	65	80	75	85	70	Find the sixth smallest element
					i			j	swap $a[i]$ and $a[j]$
45	50	55	60	65	70	75	85	80	Find the seventh smallest element
						i j			swap $a[i]$ and $a[j]$
45	50	55	60	65	70	75	85	80	Find the eighth smallest element
							i	J	swap $a[i]$ and $a[j]$
45	50	55	60	65	70	75	80	85	The outer loop ends.

7.4.1. Non-recursive Program for selection sort:

```
# include<stdio.h>
# include<conio.h>

void selectionSort( int low, int high );

int a[25];

int main()
{
    int num, i= 0;
    clrscr();
    printf( "Enter the number of elements: " );
    scanf("%d", &num);
    printf( "\nEnter the elements:\n" );
    for(i=0; i < num; i++)
        scanf( "%d", &a[i] );
    selectionSort( 0, num - 1 );
    printf( "\nThe elements after sorting are: " );
    for( i=0; i< num; i++ )
        printf( "%d  ", a[i] );
    return 0;
}

void selectionSort( int low, int high )
{
    int i=0, j=0, temp=0, minindex;
    for( i=low; i <= high; i++ )
    {
        minindex = i;
        for( j=i+1; j <= high; j++ )
        {
            if( a[j] < a[minindex] )
                minindex = j;
        }
        temp = a[i];
        a[i] = a[minindex];
        a[minindex] = temp;
    }
}
```

7.4.2. Recursive Program for selection sort:

```
#include <stdio.h>
#include<conio.h>

int x[6] = {77, 33, 44, 11, 66};
selectionSort(int);

main()
{
    int i, n = 0;
    clrscr();
    printf ( " Array Elements before sorting: " );
    for (i=0; i<5; i++)
```

```

        printf ("%d ", x[i]);
    selectionSort(n);          /* call selection sort */
    printf ("\n Array Elements after sorting: ");
    for (i=0; i<5; i++)
        printf ("%d ", x[i]);
}

selectionSort( int n)
{
    int k, p, temp, min;
    if (n== 4)
        return (-1);
    min = x[n];
    p = n;
    for (k = n+1; k<5; k++)
    {
        if (x[k] <min)
        {
            min = x[k];
            p = k;
        }
    }
    temp = x[n];          /* interchange x[n] and x[p] */
    x[n] = x[p];
    x[p] = temp;
    n++ ;
    selectionSort(n);
}

```

7.5. Quick Sort:

The quick sort was invented by Prof. C. A. R. Hoare in the early 1960's. It was one of the first most efficient sorting algorithms. It is an example of a class of algorithms that work by "divide and conquer" technique.

The quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value. The chosen value is known as the *pivot* element. Once the array has been rearranged in this way with respect to the *pivot*, the same partitioning procedure is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers up and down which are moved toward each other in the following fashion:

1. Repeatedly increase the pointer 'up' until $a[up] \geq pivot$.
2. Repeatedly decrease the pointer 'down' until $a[down] \leq pivot$.
3. If $down > up$, interchange $a[down]$ with $a[up]$
4. Repeat the steps 1, 2 and 3 till the 'up' pointer crosses the 'down' pointer. If 'up' pointer crosses 'down' pointer, the position for pivot is found and place pivot element in 'down' pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' and 'high'.

1. It terminates when the condition $low \geq high$ is satisfied. This condition will be satisfied only when the array is completely sorted.
2. Here we choose the first element as the 'pivot'. So, $pivot = x[low]$. Now it calls the partition function to find the proper position j of the element $x[low]$ i.e. pivot. Then we will have two sub-arrays $x[low], x[low+1], \dots, x[j-1]$ and $x[j+1], x[j+2], \dots, x[high]$.
3. It calls itself recursively to sort the left sub-array $x[low], x[low+1], \dots, x[j-1]$ between positions low and $j-1$ (where j is returned by the partition function).
4. It calls itself recursively to sort the right sub-array $x[j+1], x[j+2], \dots, x[high]$ between positions $j+1$ and $high$.

The time complexity of quick sort algorithm is of $O(n \log n)$.

Algorithm

Sorts the elements $a[p], \dots, a[q]$ which reside in the global array $a[n]$ into ascending order. The $a[n + 1]$ is considered to be defined and must be greater than all elements in $a[n]$; $a[n + 1] = +\infty$

quicksort (p, q)

```
{
    if ( p < q ) then
    {
        call j = PARTITION(a, p, q+1);    // j is the position of the partitioning element
        call quicksort(p, j - 1);
        call quicksort(j + 1 , q);
    }
}
```

partition(a, m, p)

```
{
    v = a[m]; up = m; down = p;           // a[m] is the partition element
    do
    {
        repeat
            up = up + 1;
        until (a[up] ≥ v);

        repeat
            down = down - 1;
        until (a[down] ≤ v);
        if (up < down) then call interchange(a, up, down);
    } while (up ≥ down);

    a[m] = a[down];
    a[down] = v;
    return (down);
}
```

```

interchange(a, up, down)
{
    p = a[up];
    a[up] = a[down];
    a[down] = p;
}

```

Example:

Select first element as the pivot element. Move 'up' pointer from left to right in search of an element larger than pivot. Move the 'down' pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped.

This process continues till the 'up' pointer crosses the 'down' pointer. If 'up' pointer crosses 'down' pointer, the position for pivot is found and interchange pivot and element at 'down' position.

Let us consider the following example with 13 elements to analyze quick sort:

1	2	3	4	5	6	7	8	9	10	11	12	13	Remarks
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				up						down			swap up & down
pivot				04						79			
pivot					up			down					swap up & down
pivot					02			57					
pivot						down	up						swap pivot & down
(24	08	16	06	04	02)	38	(56	57	58	79	70	45)	
pivot					down	up							swap pivot & down
(02	08	16	06	04)	24								
pivot, down	up												swap pivot & down
02	(08	16	06	04)									
	pivot	up		down									swap up & down
	pivot	04		16									
	pivot		down	Up									
	(06	04)	08	(16)									swap pivot & down
	pivot	down	up										
	(04)	06											swap pivot & down
	04												
	pivot, down, up												
				16									
				pivot, down, up									
(02	04	06	08	16	24)	38							

							(56	57	58	79	70	45)	
							pivot	up				down	swap up & down
							pivot	45				57	
							pivot	down	up				swap pivot & down
							(45)	56	(58	79	70	57)	
							45 pivot, down, up						swap pivot & down
									(58 pivot	79 up	70	57) down	swap up & down
										57		79	
										down	up		
									(57)	58	(70	79)	swap pivot & down
									57 pivot, down, up				
											(70	79)	
											pivot, down	up	swap pivot & down
											70		
												79 pivot, down, up	
							(45	56	57	58	70	79)	
02	04	06	08	16	24	38	45	56	57	58	70	79	

7.5.1. Recursive program for Quick Sort:

```
# include<stdio.h>
# include<conio.h>

void quicksort(int, int);
int partition(int, int);
void interchange(int, int);
int array[25];

int main()
{
    int num, i = 0;
    clrscr();
    printf( "Enter the number of elements: " );
    scanf( "%d", &num);
    printf( "Enter the elements: " );
    for(i=0; i < num; i++)
        scanf( "%d", &array[i] );
    quicksort(0, num -1);
    printf( "\nThe elements after sorting are: " );
}
```

```

        for(i=0; i < num; i++)
            printf("%d ", array[i]);
        return 0;
    }

void quicksort(int low, int high)
{
    int pivotpos;
    if( low < high )
    {
        pivotpos = partition(low, high + 1);
        quicksort(low, pivotpos - 1);
        quicksort(pivotpos + 1, high);
    }
}

int partition(int low, int high)
{
    int pivot = array[low];
    int up = low, down = high;

    do
    {
        do
            up = up + 1;
        while(array[up] < pivot );

        do
            down = down - 1;
        while(array[down] > pivot);

        if(up < down)
            interchange(up, down);

    } while(up < down);
    array[low] = array[down];
    array[down] = pivot;
    return down;
}

void interchange(int i, int j)
{
    int temp;
    temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}

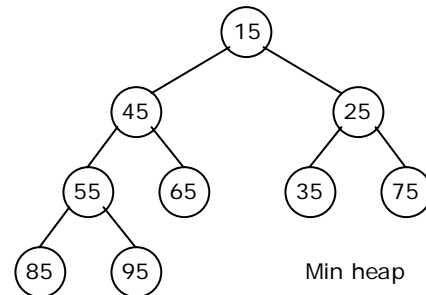
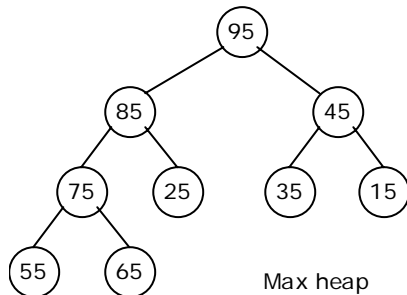
```

7.6. Priority Queue, Heap and Heap Sort:

Heap is a data structure, which permits one to insert elements into a set and also to find the largest element efficiently. A data structure, which provides these two operations, is called a priority queue.

7.6.1. Max and Min Heap data structures:

A max heap is an almost complete binary tree such that the value of each node is greater than or equal to those in its children.



A min heap is an almost complete binary tree such that the value of each node is less than or equal to those in its children.

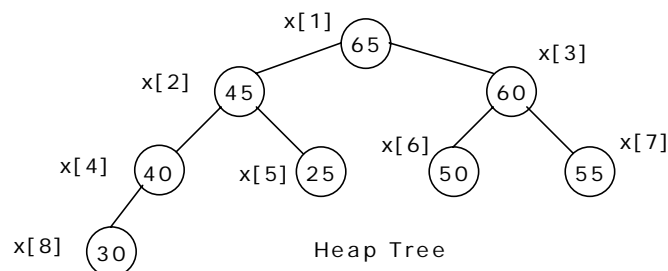
7.6.2. Representation of Heap Tree:

Since heap is a complete binary tree, a heap tree can be efficiently represented using one dimensional array. This provides a very convenient way of figuring out where children belong to.

- The root of the tree is in location 1.
- The left child of an element stored at location i can be found in location $2*i$.
- The right child of an element stored at location i can be found in location $2*i+1$.
- The parent of an element stored at location i can be found at location $\text{floor}(i/2)$.

The elements of the array can be thought of as lying in a tree structure. A heap tree represented using a single array looks as follows:

X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]
65	45	60	40	25	50	55	30



7.6.3. Operations on heap tree:

The major operations required to be performed on a heap tree:

1. Insertion,
2. Deletion and
3. Merging.

Insertion into a heap tree:

This operation is used to insert a node into an existing heap tree satisfying the properties of heap tree. Using repeated insertions of data, starting from an empty heap tree, one can build up a heap tree.

Let us consider the heap (max) tree. The principle of insertion is that, first we have to adjoin the data in the complete binary tree. Next, we have to compare it with the data in its parent; if the value is greater than that at parent then interchange the values. This will continue between two nodes on path from the newly inserted node to the root node till we get a parent whose value is greater than its child or we reached the root.

For illustration, 35 is added as the right child of 80. Its value is compared with its parent's value, and to be a max heap, parent's value greater than child's value is satisfied, hence interchange as well as further comparisons are no more required.

As another illustration, let us consider the case of insertion 90 into the resultant heap tree. First, 90 will be added as left child of 40, when 90 is compared with 40 it requires interchange. Next, 90 is compared with 80, another interchange takes place. Now, our process stops here, as 90 is now in root node. The path on which these comparisons and interchanges have taken places are shown by *dashed line*.

The algorithm Max_heap_insert to insert a data into a max heap tree is as follows:

```
Max_heap_insert (a, n)
{
    //inserts the value in a[n] into the heap which is stored at a[1] to a[n-1]
    int i, n;
    i = n;
    item = a[n];
    while ( (i > 1) and (a[ ⌊ i/2 ⌋ ] < item ) do
    {
        a[i] = a[ ⌊ i/2 ⌋ ] ;           // move the parent down
        i = ⌊ i/2 ⌋ ;
    }
    a[i] = item ;
    return true ;
}
```

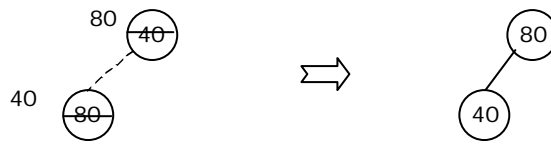
Example:

Form a heap using the above algorithm for the data: 40, 80, 35, 90, 45, 50, 70.

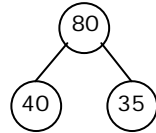
1. Insert 40:



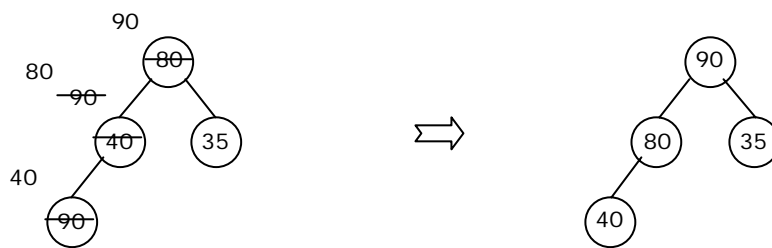
2. Insert 80:



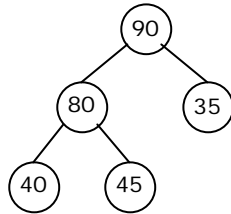
3. Insert 35:



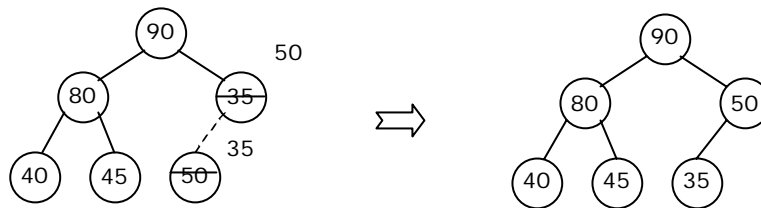
4. Insert 90:



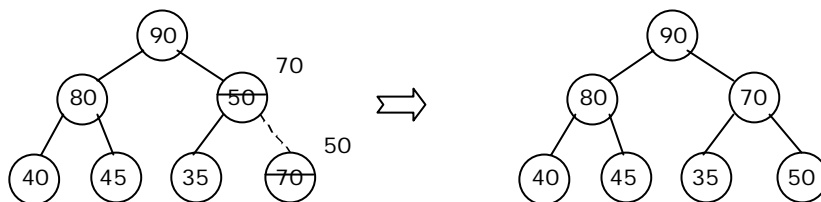
5. Insert 45:



6. Insert 50:



7. Insert 70:



Deletion of a node from heap tree:

Any node can be deleted from a heap tree. But from the application point of view, deleting the root node has some special importance. The principle of deletion is as follows:

- Read the root node into a temporary storage say, ITEM.
- Replace the root node by the last node in the heap tree. Then re-heap the tree as stated below:
 - Let newly modified root node be the current node. Compare its value with the value of its two child. Let X be the child whose value is the largest. Interchange the value of X with the value of the current node.
 - Make X as the current node.
 - Continue re-heap, if the current node is not an empty node.

The algorithm for the above is as follows:

delmax (a, n, x)

// delete the maximum from the heap a[n] and store it in x

```
{
    if (n = 0) then
    {
        write ("heap is empty");
        return false;
    }
    x = a[1]; a[1] = a[n];
    adjust (a, 1, n-1);
    return true;
}
```

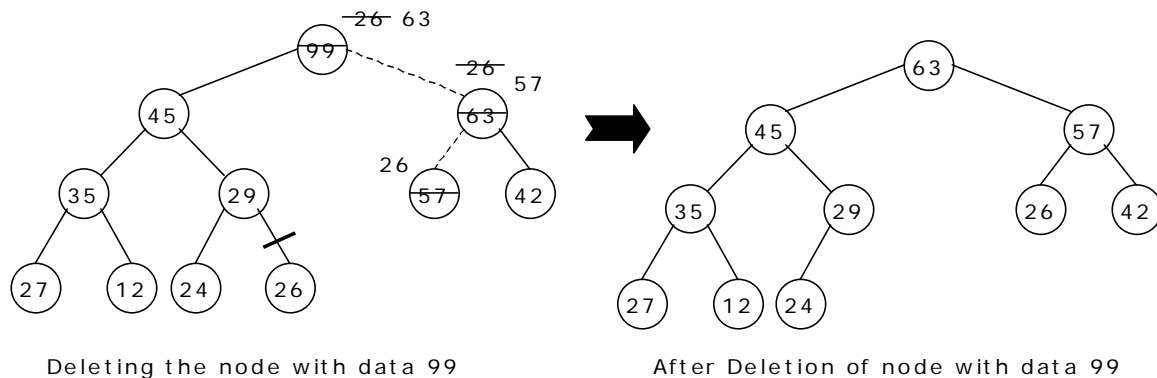
adjust (a, i, n)

// The complete binary trees with roots $a(2*i)$ and $a(2*i + 1)$ are combined with $a(i)$ to form a single heap, $1 \leq i \leq n$. No node has an address greater than n or less than 1. //

```
{
    j = 2 * i ;
    item = a[i] ;
    while (j ≤ n) do
    {
        if ((j < n) and (a (j) < a (j + 1))) then j ← j + 1;
        // compare left and right child and let j be the larger child
        if (item ≥ a (j)) then break;
        // a position for item is found
        else a[⌊ j / 2 ⌋] = a[j] // move the larger child up a level
        j = 2 * j;
    }
    a [⌊ j / 2 ⌋] = item;
}
```

Here the root node is 99. The last node is 26, it is in the level 3. So, 99 is replaced by 26 and this node with data 26 is removed from the tree. Next 26 at root node is compared with its two child 45 and 63. As 63 is greater, they are interchanged. Now,

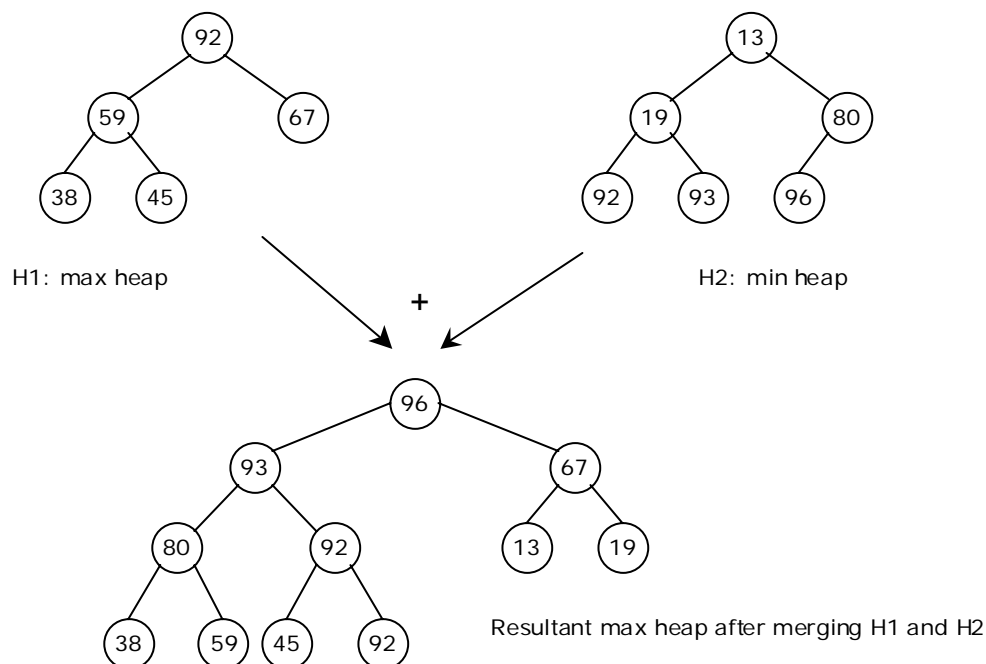
26 is compared with its children, namely, 57 and 42, as 57 is greater, so they are interchanged. Now, 26 appears as the leaf node, hence re-heap is completed.



7.6.4. Merging two heap trees:

Consider, two heap trees H1 and H2. Merging the tree H2 with H1 means to include all the node from H2 to H1. H2 may be min heap or max heap and the resultant tree will be min heap if H1 is min heap else it will be max heap. Merging operation consists of two steps: Continue steps 1 and 2 while H2 is not empty:

1. Delete the root node, say x, from H2. Re-heap H2.
2. Insert the node x into H1 satisfying the property of H1.



7.6.5. Application of heap tree:

They are two main applications of heap trees known are:

1. Sorting (Heap sort) and
2. Priority queue implementation.

7.7. HEAP SORT:

A heap sort algorithm works by first organizing the data to be sorted into a special type of binary tree called a heap. Any kind of data can be sorted either in ascending order or in descending order using heap tree. It does this with the following steps:

1. Build a heap tree with the given set of data.
2.
 - a. Remove the top most item (the largest) and replace it with the last element in the heap.
 - b. Re-heapify the complete binary tree.
 - c. Place the deleted node in the output.
3. Continue step 2 until the heap tree is empty.

Algorithm:

This algorithm sorts the elements $a[n]$. Heap sort rearranges them in-place in non-decreasing order. First transform the elements into a heap.

heapsort(a, n)

```
{
    heapify(a, n);
    for i = n to 2 by -1 do
    {
        temp = a[i];
        a[i] = a[1];
        a[1] = temp;
        adjust(a, 1, i - 1);
    }
}
```

heapify(a, n)

//Readjust the elements in $a[n]$ to form a heap.

```
{
    for i  $\leftarrow$   $\lfloor n/2 \rfloor$  to 1 by -1 do adjust(a, i, n);
}
```

adjust(a, i, n)

// The complete binary trees with roots $a(2*i)$ and $a(2*i + 1)$ are combined with $a(i)$ to form a single heap, $1 \leq i \leq n$. No node has an address greater than n or less than 1. //

```
{
    j = 2 * i ;
    item = a[i] ;
    while (j  $\leq$  n) do
    {
        if ((j < n) and (a(j) < a(j + 1))) then j  $\leftarrow$  j + 1;
        // compare left and right child and let j be the larger child
        if (item  $\geq$  a(j)) then break;
        // a position for item is found
        else a[ $\lfloor j / 2 \rfloor$ ] = a[j] // move the larger child up a level
        j = 2 * j;
    }
    a[ $\lfloor j / 2 \rfloor$ ] = item;
}
```

Time Complexity:

Each 'n' insertion operations takes $O(\log k)$, where 'k' is the number of elements in the heap at the time. Likewise, each of the 'n' remove operations also runs in time $O(\log k)$, where 'k' is the number of elements in the heap at the time.

Since we always have $k \leq n$, each such operation runs in $O(\log n)$ time in the worst case.

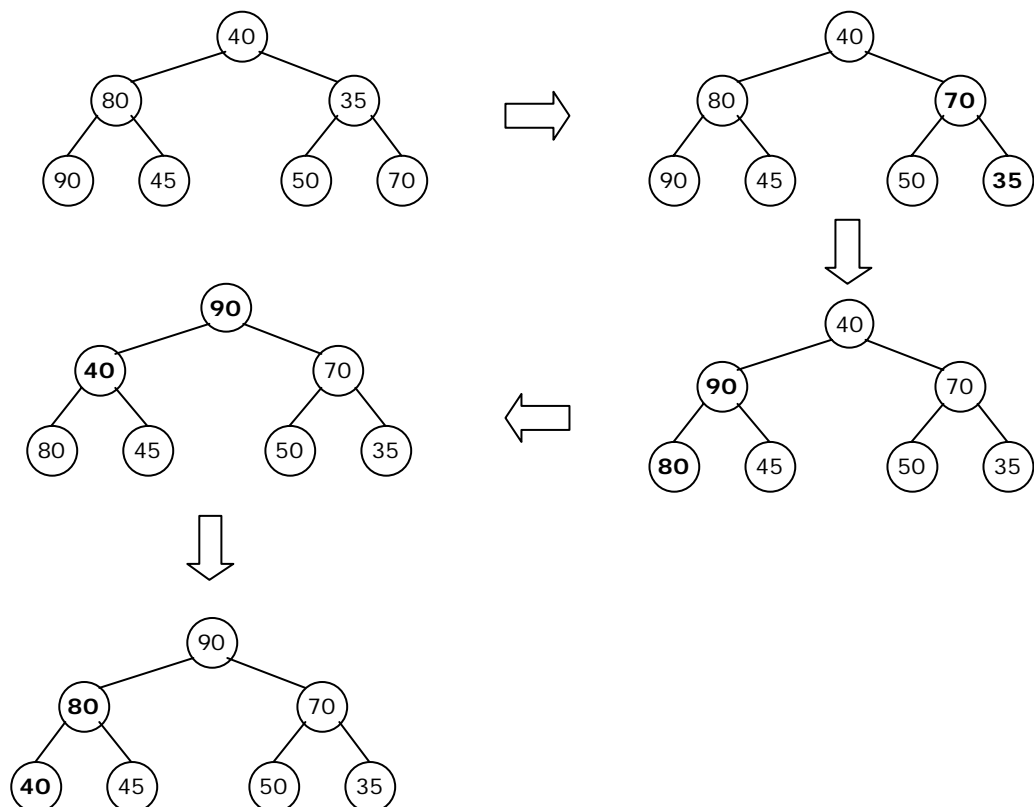
Thus, for 'n' elements it takes $O(n \log n)$ time, so the priority queue sorting algorithm runs in $O(n \log n)$ time when we use a heap to implement the priority queue.

Example 1:

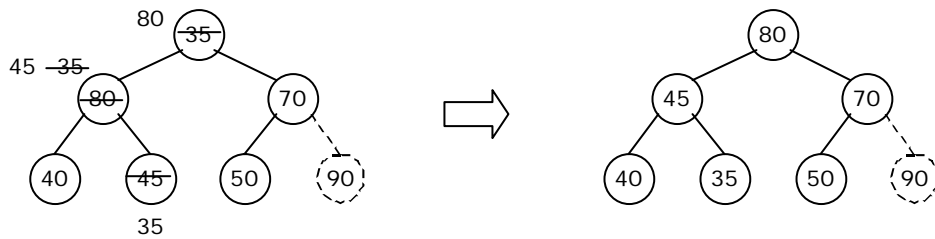
Form a heap from the set of elements (40, 80, 35, 90, 45, 50, 70) and sort the data using heap sort.

Solution:

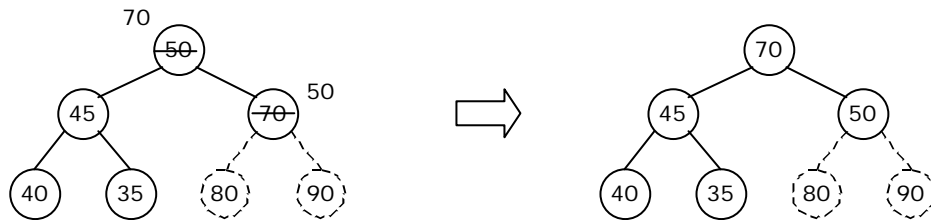
First form a heap tree from the given set of data and then sort by repeated deletion operation:



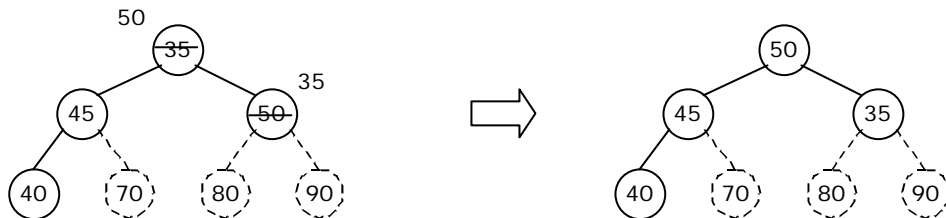
1. Exchange root 90 with the last element 35 of the array and re-heapify



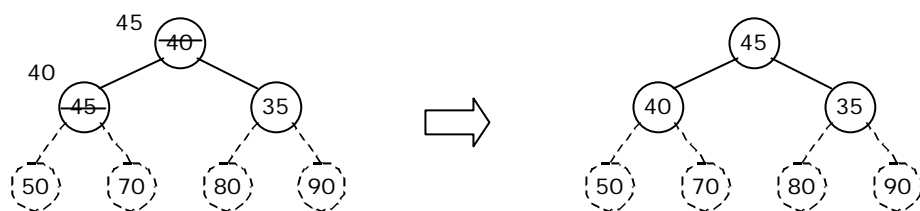
2. Exchange root 80 with the last element 50 of the array and re-heapify



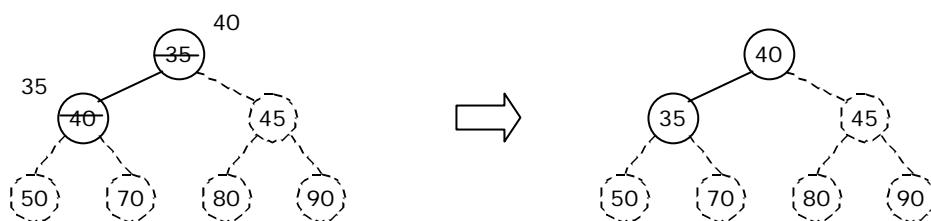
3. Exchange root 70 with the last element 35 of the array and re-heapify



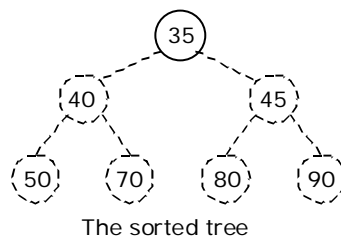
4. Exchange root 50 with the last element 40 of the array and re-heapify



5. Exchange root 45 with the last element 35 of the array and re-heapify



6. Exchange root 40 with the last element 35 of the array and re-heapify



7.7.1. Program for Heap Sort:

```
void adjust(int i, int n, int a[])
{
    int j, item;
    j = 2 * i;
    item = a[i];
    while(j <= n)
    {
        if((j < n) && (a[j] < a[j+1]))
            j++;
        if(item >= a[j])
            break;
        else
        {
            a[j/2] = a[j];
            j = 2*j;
        }
    }
    a[j/2] = item;
}

void heapify(int n, int a[])
{
    int i;
    for(i = n/2; i > 0; i--)
        adjust(i, n, a);
}

void heapsort(int n, int a[])
{
    int temp, i;
    heapify(n, a);
    for(i = n; i > 0; i--)
    {
        temp = a[i];
        a[i] = a[1];
        a[1] = temp;
        adjust(1, i - 1, a);
    }
}

void main()
{
    int i, n, a[20];
    clrscr();
    printf("\n How many element you want: ");
    scanf("%d", &n);
    printf("Enter %d elements: ", n);
    for (i=1; i<=n; i++)
        scanf("%d", &a[i]);
    heapsort(n, a);
    printf("\n The sorted elements are: \n");
    for (i=1; i<=n; i++)
        printf("%5d", a[i]);
    getch();
}
```

7.8. Priority queue implementation using heap tree:

Priority queue can be implemented using circular array, linked list etc. Another simplified implementation is possible using heap tree; the heap, however, can be represented using an array. This implementation is therefore free from the complexities of circular array and linked list but getting the advantages of simplicities of array.

As heap trees allow the duplicity of data in it. Elements associated with their priority values are to be stored in from of heap tree, which can be formed based on their priority values. The top priority element that has to be processed first is at the root; so it can be deleted and heap can be rebuilt to get the next element to be processed, and so on. As an illustration, consider the following processes with their priorities:

Process	P ₁	P ₂	P ₃	P ₄	P ₅	P ₆	P ₇	P ₈	P ₉	P ₁₀
Priority	5	4	3	4	5	5	3	2	1	5

These processes enter the system in the order as listed above at time 0, say. Assume that a process having higher priority value will be serviced first. The heap tree can be formed considering the process priority values. The order of servicing the process is successive deletion of roots from the heap.

Exercises

1. Write a recursive "C" function to implement binary search and compute its time complexity.
2. Find the expected number of passes, comparisons and exchanges for bubble sort when the number of elements is equal to "10". Compare these results with the actual number of operations when the given sequence is as follows: 7, 1, 3, 4, 10, 9, 8, 6, 5, 2.
3. An array contains "n" elements of numbers. The several elements of this array may contain the same number "x". Write an algorithm to find the total number of elements which are equal to "x" and also indicate the position of the first such element in the array.
4. When a "C" function to sort a matrix row-wise and column-wise. Assume that the matrix is represented by a two dimensional array.
5. A very large array of elements is to be sorted. The program is to be run on a personal computer with limited memory. Which sort would be a better choice: Heap sort or Quick sort? Why?
6. Here is an array of ten integers: 5 3 8 9 1 7 0 2 6 4
Suppose we partition this array using quicksort's partition function and using 5 for the pivot. Draw the resulting array after the partition finishes.
7. Here is an array which has just been partitioned by the first step of quicksort: 3, 0, 2, 4, 5, 8, 7, 6, 9. Which of these elements could be the pivot? (There may be more than one possibility!)
8. Show the result of inserting 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13, and 2, one at a time, into an initially empty binary heap.
9. Sort the sequence 3, 1, 4, 5, 9, 2, 6, 5 using insertion sort.

10. Show how heap sort processes the input 142, 543, 123, 65, 453, 879, 572, 434, 111, 242, 811, 102.
11. Sort 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5 using quick sort with median-of-three partitioning and a cutoff of 3.

Multiple Choice Questions

1. What is the worst-case time for serial search finding a single item in an array? [D]
 A. Constant time
 B. Quadratic time
 C. Logarithmic time
 D. Linear time
2. What is the worst-case time for binary search finding a single item in an array? [B]
 A. Constant time
 B. Quadratic time
 C. Logarithmic time
 D. Linear time
3. What additional requirement is placed on an array, so that binary search may be used to locate an entry? [C]
 A. The array elements must form a heap.
 B. The array must have at least 2 entries
 C. The array must be sorted.
 D. The array's size must be a power of two.
4. Which searching can be performed recursively ? [B]
 A. linear search
 B. both
 C. Binary search
 D. none
5. Which searching can be performed iteratively ? [B]
 A. linear search
 B. both
 C. Binary search
 D. none
6. In a selection sort of n elements, how many times is the swap function called in the complete execution of the algorithm? [B]
 A. 1
 B. n^2
 C. $n - 1$
 D. $n \log n$
7. Selection sort and quick sort both fall into the same category of sorting algorithms. What is this category? [B]
 A. $O(n \log n)$ sorts
 B. Interchange sorts
 C. Divide-and-conquer sorts
 D. Average time is quadratic
8. Suppose that a selection sort of 100 items has completed 42 iterations of the main loop. How many items are now guaranteed to be in their final spot (never to be moved again)? [C]
 A. 21
 B. 41
 C. 42
 D. 43
9. When is insertion sort a good choice for sorting an array? [B]
 A. Each component of the array requires a large amount of memory
 B. The array has only a few items out of place
 C. Each component of the array requires a small amount of memory
 D. The processor speed is fast

10. What is the worst-case time for quick sort to sort an array of n elements? [D]
 A. $O(\log n)$ C. $O(n \log n)$
 B. $O(n)$ D. $O(n^2)$
11. Suppose we are sorting an array of eight integers using quick sort, and we have just finished the first partitioning with the array looking like this:
 2 5 1 7 9 12 11 10 Which statement is correct? [A]
 A. The pivot could be either the 7 or the 9.
 B. The pivot is not the 7, but it could be the 9.
 C. The pivot could be the 7, but it is not the 9.
 D. Neither the 7 nor the 9 is the pivot
12. What is the worst-case time for heap sort to sort an array of n elements? [C]
 A. $O(\log n)$ C. $O(n \log n)$
 B. $O(n)$ D. $O(n^2)$
13. Suppose we are sorting an array of eight integers using heap sort, and we have just finished one of the reheapifications downward. The array now looks like this: 6 4 5 1 2 7 8
 How many reheapifications downward have been performed so far? [B]
 A. 1 C. 2
 B. 3 or 4 D. 5 or 6
14. Time complexity of inserting an element to a heap of n elements is of the order of [A]
 A. $\log_2 n$ C. $n \log_2 n$
 B. n^2 D. n
15. A min heap is the tree structure where smallest element is available at the [B]
 A. leaf C. intermediate parent
 B. root D. any where
16. In the quick sort method , a desirable choice for the portioning element will be [C]
 A. first element of list C. median of list
 B. last element of list D. any element of list
17. Quick sort is also known as [D]
 A. merge sort C. heap sort
 B. bubble sort D. none
18. Which design algorithm technique is used for quick sort . [A]
 A. Divide and conqueror C. backtrack
 B. greedy D. dynamic programming
19. Which among the following is fastest sorting technique (for unordered data) [C]
 A. Heap sort C. Quick Sort
 B. Selection Sort D. Bubble sort
20. In which searching technique elements are eliminated by half in each pass . [C]
 A. Linear search C. Binary search
 B. both D. none
21. Running time of Heap sort algorithm is ----- [B]
 A. $O(\log_2 n)$ C. $O(n)$
 B. $O(n \log_2 n)$ D. $O(n^2)$

22. Running time of Bubble sort algorithm is ----- [D]
 A. $O(\log_2 n)$ C. $O(n)$
 B. $O(n \log_2 n)$ D. $O(n^2)$
23. Running time of Selection sort algorithm is ----- [D]
 A. $O(\log_2 n)$ C. $O(n)$
 B. $O(n \log_2 n)$ D. $O(n^2)$
24. The Max heap constructed from the list of numbers 30,10,80,60,15,55 is [C]
 A. 60,80,55,30,10,15 C. 80,55,60,15,10,30
 B. 80,60,55,30,10,15 D. none
25. The number of swappings needed to sort the numbers 8,22,7,9,31,19,5,13 in ascending order using bubble sort is [D]
 A. 11 C. 13
 B. 12 D. 14
26. Time complexity of insertion sort algorithm in best case is [C]
 A. $O(\log_2 n)$ C. $O(n)$
 B. $O(n \log_2 n)$ D. $O(n^2)$
27. Binary search algorithm performs efficiently on a [C]
 A. linked list C. array
 B. both D. none
28. Which is a stable sort ? [D]
 A. Bubble sort C. Quick sort
 B. Selection Sort D. none
29. Heap is a good data structure to implement [A]
 A. priority Queue C. linear queue
 B. Deque D. none
30. Always Heap is a [A]
 A. complete Binary tree C. Full Binary tree
 B. Binary Search Tree D. none