UNIT IV

NON-LINEAR DATA STRUCTURES

Trees – Binary Trees – Binary tree representation and traversals – Application of trees: Set representation and Union-Find operations – Graph and its representations – Graph Traversals – Representation of Graphs – Breadth-first search – Depth-first search - Connected components.

4.1. Trees

A tree is a finite set of one or more nodes such that there is a specially designated node called the Root, and zero or more non empty sub trees T1, T2....Tk, each of whose roots are connected by a directed edge from Root R.

A tree is a collection of n nodes, one of which is the root, and n-1 edges.

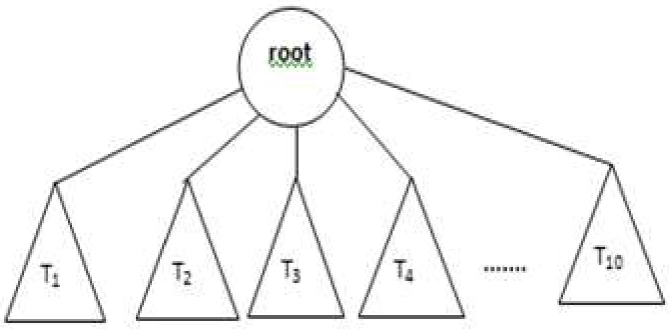


Fig.4.1(a)Generic Tree

Example: Figure shows a tree T with 13 nodes, A, B, C, D, E, F, G, H, I, J, K, L, M

The root of a tree T is the node at the top, and the children of a node are ordered from left to right. Accordingly, A is the root of T(Tree), and A has three children; the first child B, the second child C and the third child D.

Observe that,

a. The node C has three children.
b. Each of the nodes B and J has two children.
c. Each of the nodes D and H has only one child.
d. The nodes E, F, G, K, I, L and M have no children.

Root

- The node at the top of the tree is called the root.
- There is only one root in a tree.
- Example - From Fig.:4.1(b) A.

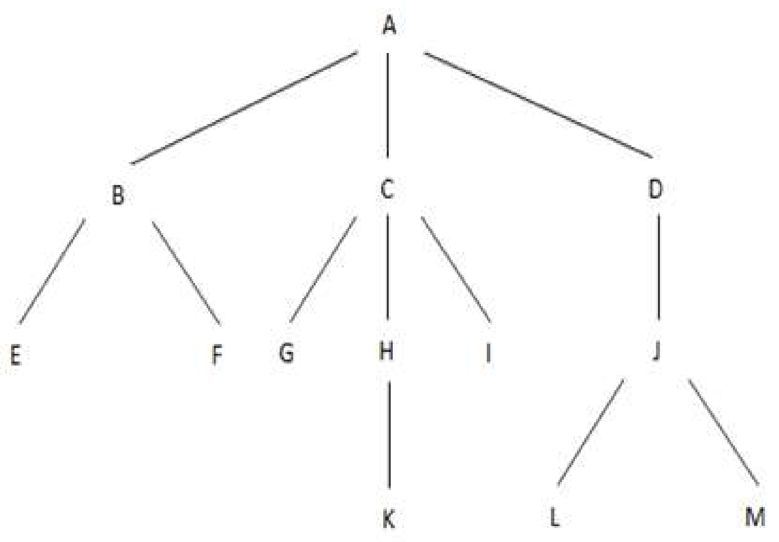


Fig. 4.1(b) General Tree

Observe that,

e. The node C has three children.

f. Each of the nodes B and J has two children.

g. Each of the nodes D and H has only one child.

h. The nodes E, F, G, K, I, L and M have no children.

Root

- The node at the top of the tree is called the root.
- There is only one root in a tree.
- Example - From Fig.:4.1(b) A.

Parent

- If there is an edge from node *R* to node *M*, then *R* is a Parent of *M*.
- Any node (except the root) has exactly one edge running upward to another node. The node above it is called the *parent* of the node. Parents, grandparents, etc. are ancestors.
- Example - From Fig.:4.1(b) B, C, D, H, J.

Child

- If there is an edge from node *R* to node *M*, then *M* is a child of *R*.

- Any node may have one or more lines running downward to other nodes. The nodes below a given node are called its *children*.

Ancestor

- If there is a path from node n_1 to node n_2, then n1 is an ancestor of n_2. Parents, grandparents, etc. are ancestors.
- Example - From Fig.:4.1(b) B is an ancestor of E.

Descendants

- If there is a path from node n_1 to node n_2, then n_2 is a child of n_1.
- Children, grandchildren, etc. are descendants
- Example - From Fig.:4.1(b) E is a child of B.

Siblings

- Children of the same parent are called as *siblings*.
- Example - From Fig.:4.1(b) G, H, I are the siblings of C.

Leaf and Internal node

- A node that has no children is called a leaf node or simply a leaf. There can be only one root in a tree, but there can be many leaves.
- A node (apart from the root) that has children is an internal node or *non-leaf node*.
- Example - From Fig.:4.1(b) Leaf Nodes: E, F, G, K, I, L, M and Non-leaf Nodes: B, C, D

Path

- A path from node n_1 to n_k is defined as a sequence of nodes n_1, n_2, …,n_k such that n_i, is the parent of n_{i+1} for $1 \leq i \geq k$,. The length of the path is $k - 1$.
- The *length* of the path is the number of edges on the path. There is path of length zero from every node to itself.
- In a tree there is exactly one path from the Root to each node.
- Example - From Fig.:4.1(b) A-B-E and Length is 2.

Levels

- The level of a particular node refers to how many generations the node is from the root.
- If the root is assumed to be on level 0, then its children will be on level 1, its grandchildren will be on level 2, and so on.

- Example - From Fig.:4.1(b) 3.

Depth

- The depth of a node n_i is the length of the unique path from the Root to n_i.
- The Root is at depth 0.
- Example - From Fig.:4.1(b) Depth(k)=3

Height

- The height of a node n_i is the length of the longest path from n_i to leaf.
- All leaves are at height 0.
- The height of the tree is equal to height o the Root.
- Example - From Fig.:4.1(b) Height(A)=3

Degree of a node

- The degree of a node is the number of subtrees of the node. The node with degree 0 is a leaf or terminal node**.**
- *Subtree :* A nodes subtree contains all its descendants.

Degree

- The number of subtrees of a node is called its degree.
- The degree of the tree is the maximum degree of any node in the tree.
- Example - From Fig.:4.1(b) 3

Terminal

- Those with no children, are called *terminal nodes***.**
- Example - From Fig.:4.1(b) E, F, G, I, K, L, M.

4.1.1. Implementation of Trees

Left child right sibling data structures for general trees

The best way to implement a tree is *linked list.*

For that each node can have the *data, a pointer* to each child of the node. But the number of children per node can vary so greatly and is not known in advance, so it is infeasible to make the children direct links in the data structure, because there would be too much wasted space.

The solution is simple: Keep the children of each node in a linked list of tree nodes. so the structure of the general tree contains the 3 fields

- Element
- Pointer to the Left Child

- Pointer to the Next Sibling

Example: Left Child/Right Sibling Representation of Fig.:4.1.1(a).

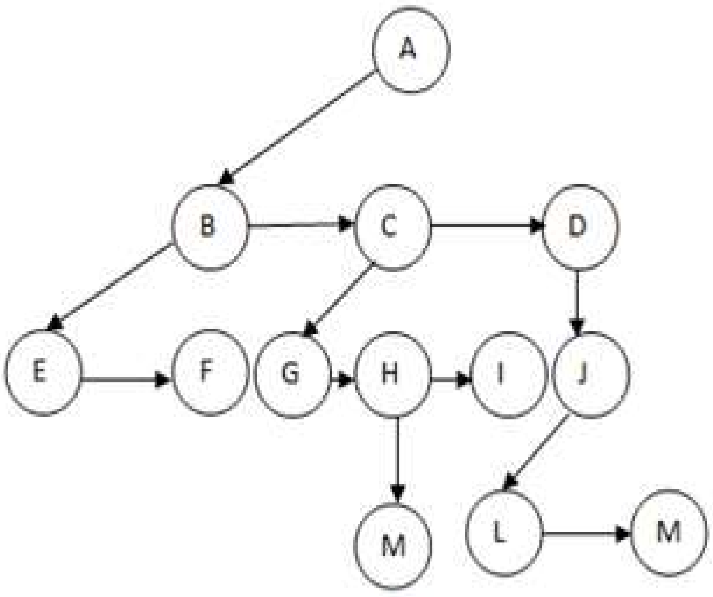


Fig. 4.1.1(a). Left Child/Right Sibling Representation

In this representation, the Arrow that point downward is FirstChild pointers. Arrow that go left to right are NextSibling pointers.

4.2. Binary Trees

Definition

A **binary tree** is a tree in which each node has at most two children.

Example The below shows that a binary tree consists of a root and two subtrees, T_l and T_r,

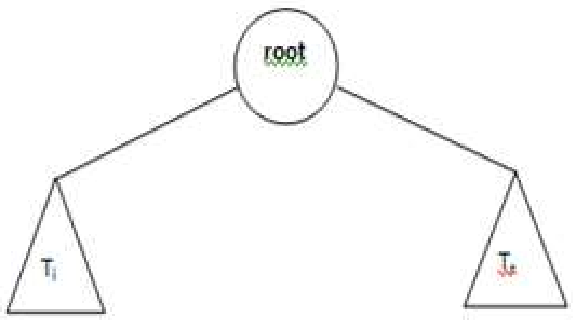


Fig.: 4.1.2(a) Left Child/Right Sibling Representation

Types of Binary Tree

1. **Skewed Binary Tree**

Skewed Binary tree is a binary tree in which all nodes other than the leaf node have only either the left or right child.

If it has only a left child it is called as left skewed binary tree.

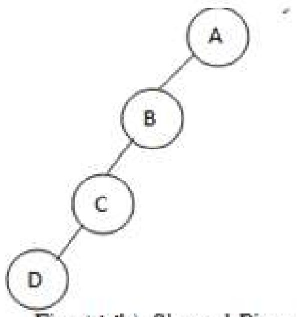


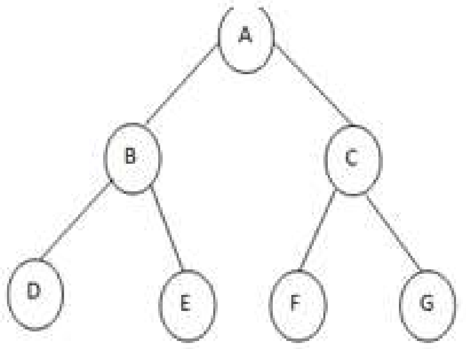
Fig.: 4.1.2b). Skewed Binary Tree

2. **Rooted Binary Tree**

It is a binary tree in which every node has at most two children

3. **Fully Binary Tree** A Binary Tree is a fully binary tree if it contains maximum possible number of nodes in all level.

Fig 4.1.2(c)Fully Binary Tree



From Fig.:1.2(c) Height(T)=3. The levels are,

Levels	*Nodes*
0	1
1	2
2	4

4. Perfect Binary tree

A binary tree is a perfect binary tree in which all leaves are at the same depth. Example - Fig.:4.1.2(c).

5. Complete Binary Tree

A binary tree is said to be a complete binary tree if all its level, except possibly the last level have the maximum number of possible nodes at the last level appear as far as left as possible.

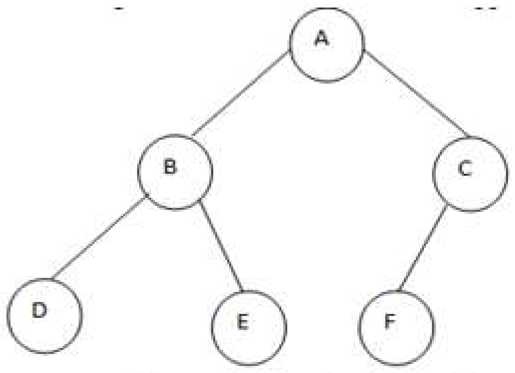


Fig4.1.2(d). Complete Binary Tree

6. Strict Binary Tree

Every non-terminal node in a binary tree consists of non-empty left subtree and right subtree then such a tree is called as strict binary tree

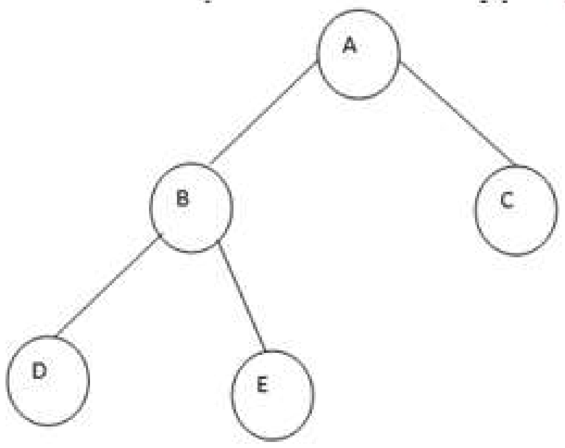


Fig 4.1.2 e). Fully Binary Tree

7. Extended Binary Tree

If each node of a tree has either 0 or 2children. In that case the nodes with 2 children are called

internal nodes and the nodes with 0 children are called external nodes.

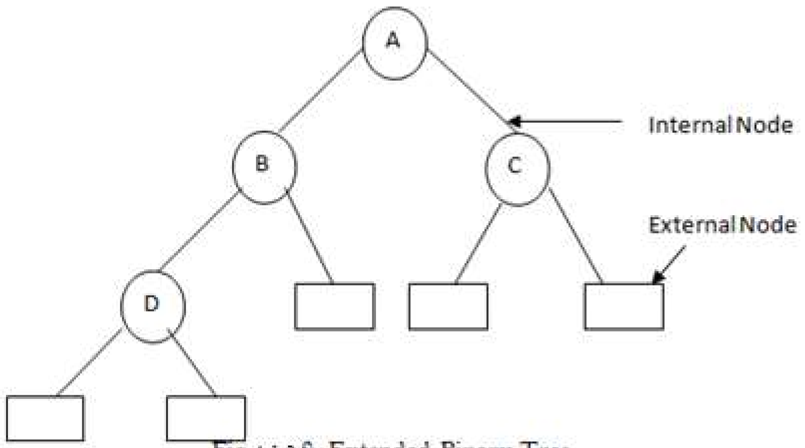


Fig 4.1.2 f). Extended Binary Tree

4.2.1. Binary Tree Representations

1. *Array Representation (Sequential Representation)*:

The elements in the tree are represented using arrays.

For any element in position i, the left child is in position 2i, the right child is in position (2i + 1), and the parent is in position (i/2).

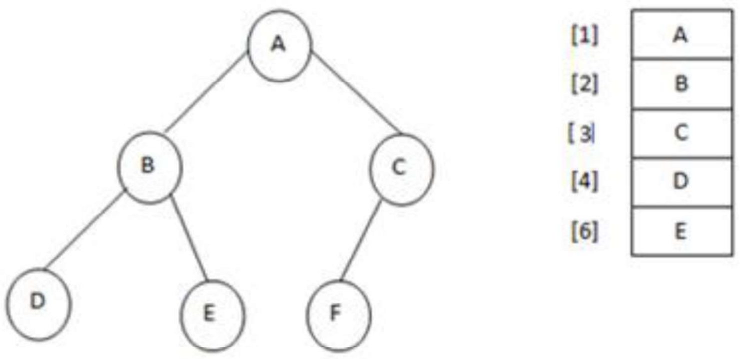


Fig 4.1.3.a). Array Representation of Binary Tree

Disadvantages

- Wastage of space
- Insertion/deletion is a tedious process.

2. *Linked Representation:*

A binary tree every node has atmost two children, so we can keep direct pointers to the children. Every node in the tree structure can have 3 fields.

i. Element
ii. Pointer to the left subtree
iii. Pointer to the right subtree

It can be shown below

Element	Left	Right

Fig.: 4.1.3(b) General Format for Linked List Representation of Binary Tree

Example

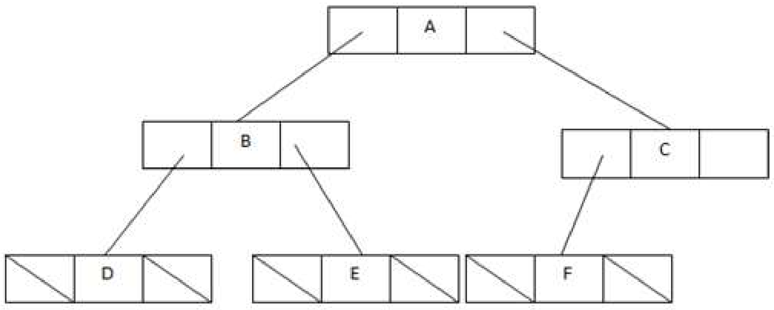


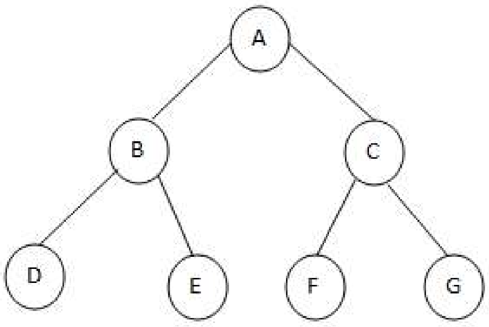
Fig. 4.1.3(c) Linked List Representation of Binary Tree

4.2.2. Tree Traversals

Tree traversal is a process of moving through a tree in a specified order to process each of the nodes. Each of the nodes is processed only once (although it may be visited more than once).

There are three standard ways of traversing a binary tree T with root R.

1. Preorder Traversal (Root - Left - Right)
2. Inorder Traversal (Left - Root - Right)
3. Postorder Traversal (Left - Right - Root)



1. Fig.:1.3(a). Binary Tree for Tree Traversal

From Fig.: 1.3(a),

1. The preorder traversal of T is **A B D E C F G.**
2. The Inorder traversal of T is **D B E A F C G.**
3. The Postorder traversal is **D E B F G C A.**

```
struct node
{
      ElementType item;
      Tree lchild;
      Tree rchild;
};
template<class itype>
void btree<itype>::inorder(node *t)
{
      if(t!=NULL)
      {
            inorder(t->lchild);
            cout<<t->item;
            inorder(t->rchild);
      }
}
void btree<itype>::preorder(node *t)
{
      if(t!=NULL)
      {
            cout<<t->item;
            preorder(t->lchild);
            preorder(t->rchild);
      }
}
void btree<itype>::postorder(node *t)
{
      if(t!=NULL)
      {
            postorder(t->lchild);
            postorder(t->rchild);
            cout<<t->item;
      }
}
```

4.2.3. Application of trees

4.2.3.1. Set representation

Assume that the elements of the sets are the numbers 0,1,2,3,4,…..,n-1. These numbers must be indices into a symbol table where the actual elements are stored.

The set being represented are pairwise disjoint (if S_i and S_j, $i \neq j$, are two sets).

Example, when n=10, the elements may be partitioned into three disjoint sets, S_1={0,6,7,8} S_2={1,4,9} and S_3={2,3,5}.

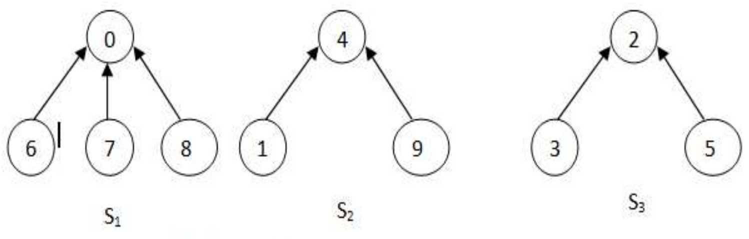
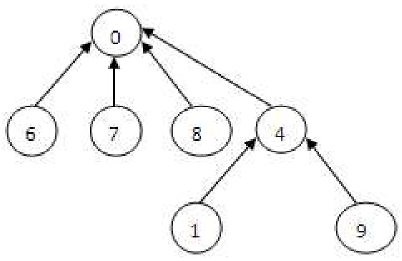


Fig 4.1.5(a) Possible tree representation of sets

Set operations are,

1. *Disjoint set union:* If S_i and S_j are two sets , then their union S_iUS_j ={all elements such that x is in S_i

or S_j }. Thus, S_1US_2={0,6,7,8,1,4,9}.



S_1US_2

Fig 4.1.5(b) Possible tree representation of sets S_1US_2

2. *Find(i) :* Find the set containing element i. Thus, 3 is in set S_3,and 8 is in set S_1.

4.2.3.2. Union-Find operations

1. ***Data Representation for*** **S_1, S_2, S_3**

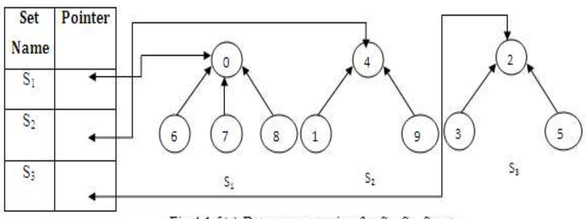


Fig 4.1.5(c) Data representation for S_1, S_2, S_3 sets

2. union of trees

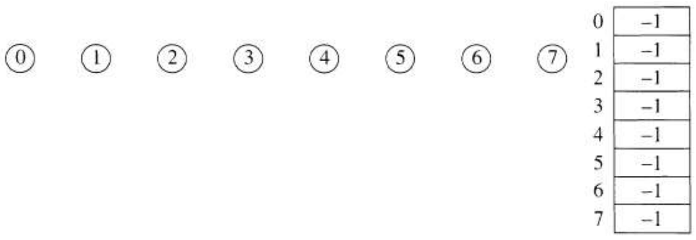


Fig. 4.1.5(d) A forest and its eight elements, initially in different sets.

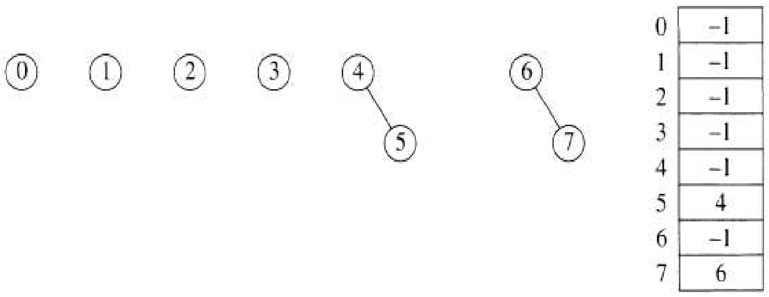


Fig. 4.1.5(e)The forest after the union of trees with roots 4 & 5, 6 & 7.

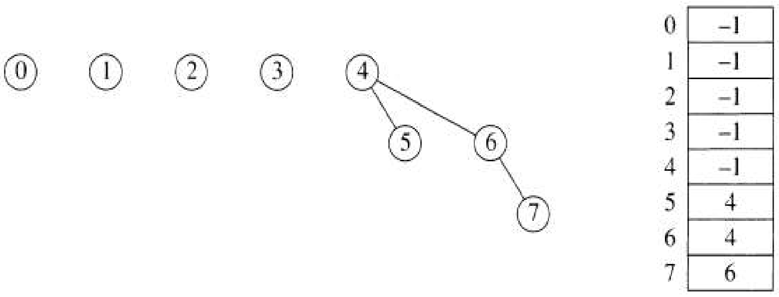


Fig. 4.1.5(f) The forest after the union of trees with roots 4 and 6.

3. Smart Union Algorithms

To make the smaller tree a subtree of the larger, breaking ties by any method, an approach called **union-by-size.** The preceding three union operations were all ties, so we can consider that they-were peLformed by size. If the next operation is union(3, 4), the forest shown in Figure 24.16 forms.

```
class DisjSets
{
    public:
    DisjSets( int numElements ) ;
    int find( int x ) const;
   int find( int x ) ;
    void unionsets( int rootl, int root2 ) ;
   private:
   vector<int> s;
   void assertIsRoot( int root ) const;
   void assertIsItem( int item ) const;
};
/ / Construct the disjoint sets object.
/ / numElements is the lnitial number of disjoint sets.
DisjSets::DisjSets(int numElements) : s(numElements)
{
      for( int i = 0; i < s.size( ) ; i++ )
      s[ i ] = -1;
}
/ / Union two disjoint sets.
/ / rootl is the root of set 1. root2 is the root of set 2.
void DisjSets::unionsets( int rootl, int root2 )
{
      assertIsRoot( root1 ) ;
      assertIsRoot( root2 ) ;
      if( s[ root2 ] < s[ root1 ] ) /; root2 is deeper
      S[ root1 ] = root2; / / Make root2 new root
      else
      {
            if( S[ root1 ] == S[ root2 ] )
            S[ root1 ] - - ; / / Update height if same
            S[ root2 ] = rootl; / / Make root1 new root
      }
}
/ / Perform a find without path compression.
int DisjSets: :find( int x ) const
{
      assertIsItem( x ) ;
            if( S[ x ] < 0 )
      retun x;
      else
            return find( s [ x ] ) ;
}
/ / Perform a find with path compression.
int DisjSets::find( int x )
{
      assertisItem( x ) ;
      if( S [ x ] < 0 )
            return x;
      else
            returns[ x ] = find( s[ x ] ) ;
}
```

4.3. Priority Queue (binary heap)

This is basically designed for the purpose of implementing the job scheduling.

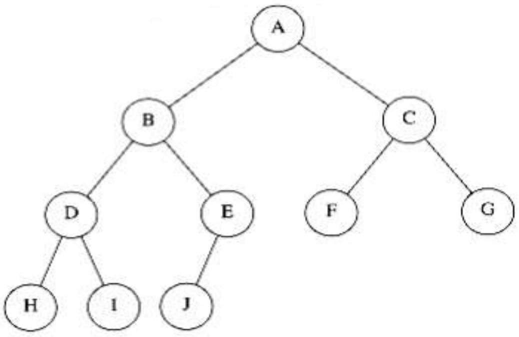
Heaps have two properties, namely,

1. Structure property
2. Heap order property.

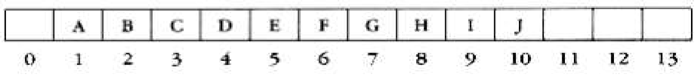
Structure Property

A heap is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right. Such a tree is known as a complete binary tree.

For any element in array position i, the left child is in position 2i, the right child is in the cell after the left child (2i + 1), and the parent is in position *i*/2.



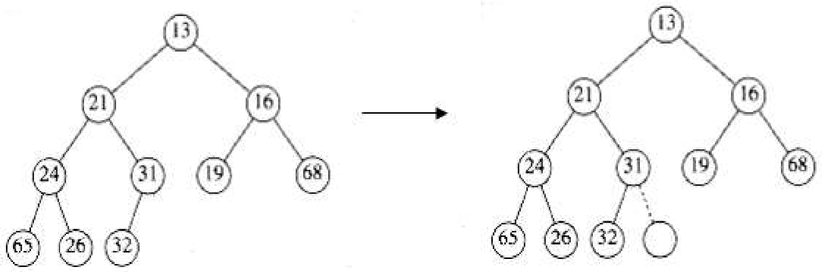
Array Implementation



	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Heap Order property

In a heap, for every node X, the key in the parent of X is smaller than (or equal to) the key in X, with the obvious exception of the root (which has no parent).



```
struct heap_struct
{
        unsigned int max_heap_size;
        unsigned int size;
        element_type *elements;
};
typedef struct heap_struct *PRIORITY_QUEUE;
```

Basic Heap Operation

3. Insert

4. Delete_min

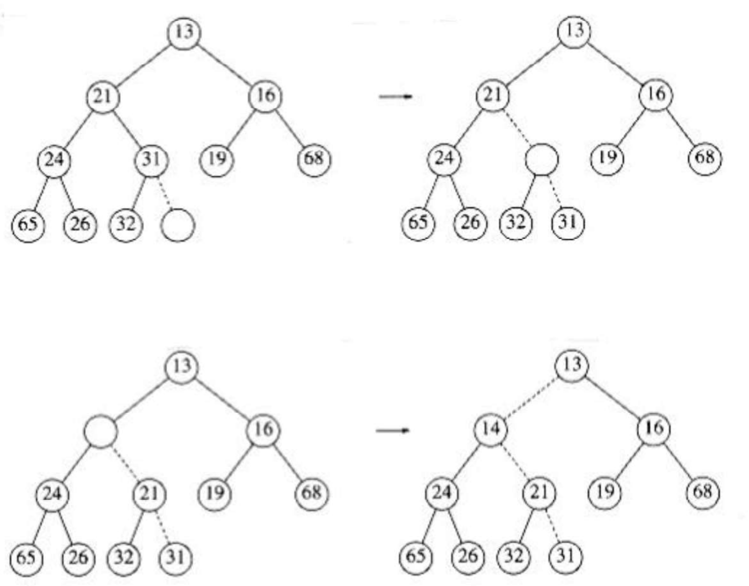
Insert

To insert an element x into the heap, we create a hole in the next available location, since otherwise the tree will not be complete.

If x can be placed in the hole without violating heap order, then we do so and are done. Otherwise we slide the element that is in the whole parent node into the hole, thus bubbling the hole up toward the root. We continue this process until x can be placed in the hole.

This general strategy is known as a percolate up; the new element is percolated up the heap until the correct location is found.

Insert 14



Routine

```
void insert( element_type x, PRIORITY_QUEUE H )
{
        unsigned int i;
        if( is_full( H ) )
                error("Priority queue is full");
        else
        {
                i = ++H->size;
                while( H->elements[i/2] > x )
                {
                        H->elements[i] = H->elements[i/2];
                        i /= 2;
                }
                H->elements[i] = x;
        }
}
```

DeleteMin

Delete_mins are handled in a similar manner as insertions. Finding the minimum is easy; the hard part is removing it. When the minimum is removed, a hole is created at the root.

Since the heap now becomes one smaller, it follows that the last element *x* in the heap must move somewhere in the heap.

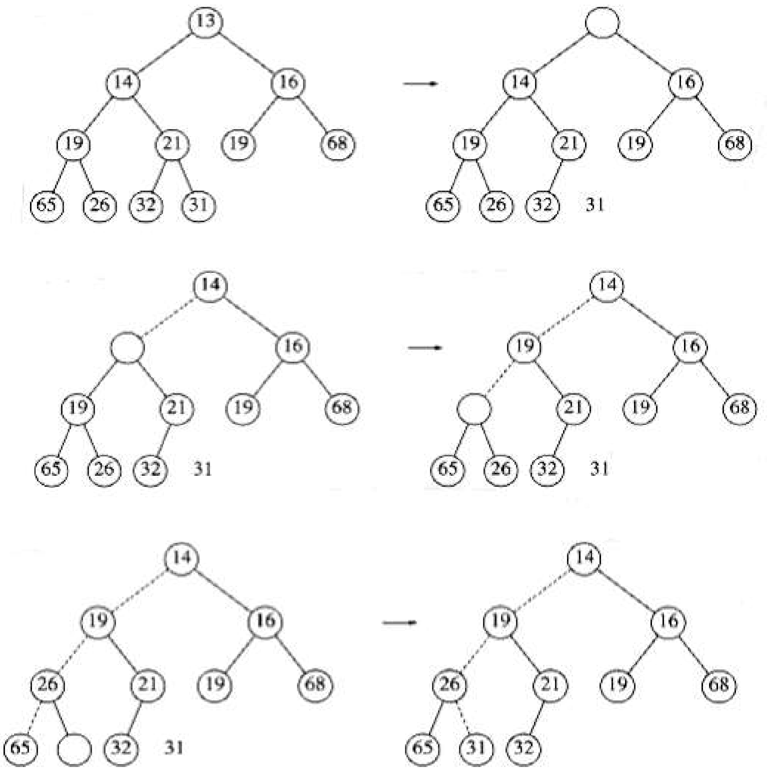
If *x* can be placed in the hole, then we are done.

We repeat this step until *x* can be placed in the hole. Thus, our action is to place *x* in its correct spot along a path from the root containing *minimum* children.

Routine

```
element_type delete_min( PRIORITY_QUEUE H )
{
        unsigned int i, child;
        element_type min_element, last_element;
        if( is_empty( H ) )
        {
                error("Priority queue is empty");
                return H->elements[0];
        }
        min_element = H->elements[1];
        for( i=1; i*2 <= H->size; i=child )
        { /* find smaller child */
                child = i*2;
                if((child!=H->size)&&(H->elements[child+1]<H->elements[child]))
```

```
                    child++; /* percolate one level */
                if( last_element > H->elements[child] )
                    H->elements[i] = H->elements[child];
                else
                    break;
            }
            H->elements[i] = last_element;
            return min_element;
}
```



Heap applications

The heap data structure has many applications.

- Heapsort: One of the best sorting methods being in-place and with no quadratic worst-case scenarios.

- Selection algorithms: Finding the min, max, both the min and max, median, or even the *k*-th largest element can be done in linear time (often constant time) using heaps. E.g. Telephone Call Processing
- Graph algorithms: By using heaps as internal traversal data structures, run time will be reduced by polynomial order. Examples of such problems are Prim's minimal spanning tree algorithm and Dijkstra's shortest path problem.
- Queuing Theory

Full and almost full binary heaps may be represented in a very space-efficient way using an array alone. The first (or last) element will contain the root. The next two elements of the array contain its children. The next four contain the four children of the two child nodes, etc.

Thus the children of the node at position n would be at positions 2n and 2n+1 in a one-based array, or 2n+1 and 2n+2 in a zero-based array. This allows moving up or down the tree by doing simple index computations. Balancing a heap is done by swapping elements which are out of order.

As we can build a heap from an array without requiring extra memory (for the nodes, for example), heapsort can be used to sort an array in-place.

One more advantage of heaps over trees in some applications is that construction of heaps can be done in linear time using Tarjan's algorithm.

4.4. Graph

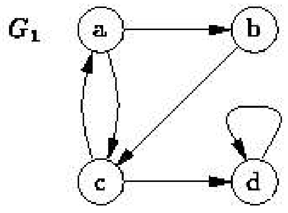
Graph Algorithms Definition

It is a nonlinear data structure. A graph G = (V, E) consists of a *set of vertices*, V, and *set of edges E*. Each edge is a pair (v, w) where v,wεv. Vertices are referred to as *nodes* and the arc between the nodes are referred to as *Edges*.

4.4.1. Types of Graph

1. Directed Graph (or) Digraph:

Directed graph is a graph which consists of directed edges, where each edge in E is unidirectional. It is also referred as Digraph. If (v, w) is a directed edge then (v, w) # (w, v).



The above graph comprised of four vertices and six edges:

V={a,b,c,d}

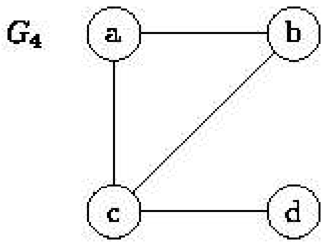
E={(a,b),(a,c),(b,c),(c,a),(c,d),(d,d)}

Adjacent Vertex :Vertex w is adjacent to v, if and only if there is an edge from vertex v to w (i.e. (v,w) ε E. In the above graph vertex c is adjacent to and vertex b is adjacent to a and so on.

2. **Undirected Graph:**

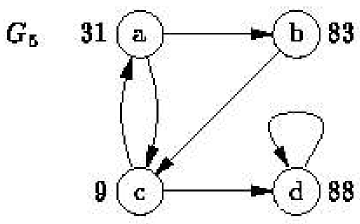
An undirected graph is a graph, which consists of undirected edges. If (v, w) is an undirected edge then (v,w) = (w, v).

consider the undirected graph G=(V_1,E_1) comprised of four vertices and four edges: V_1={a,b,c,d} E_1={{a,b}{a,c}{b,c},{c,d}} The graph can be represented *graphically* as shown below

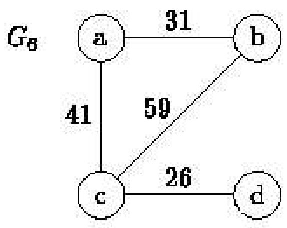


3. **Weighted Graph**

A graph is said to be weighted graph if every edge in the graph is assigned a weight or value. It can be directed or undirected graph.



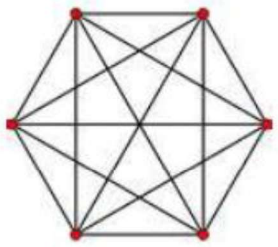
directed graph with labeled vertices



undirected graph with labeled edges

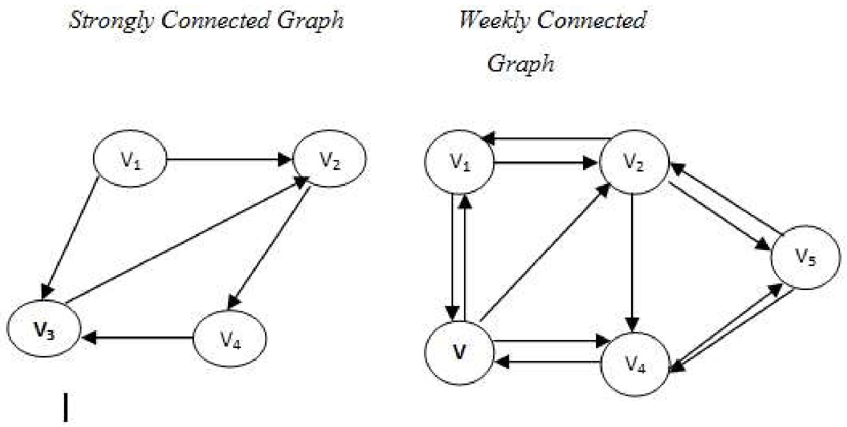
4. **Complete Graph**

A complete graph is a graph in which there is an edge between every pair of vertices. A complete graph with n vertices will have n (n - 1)/2 edges.



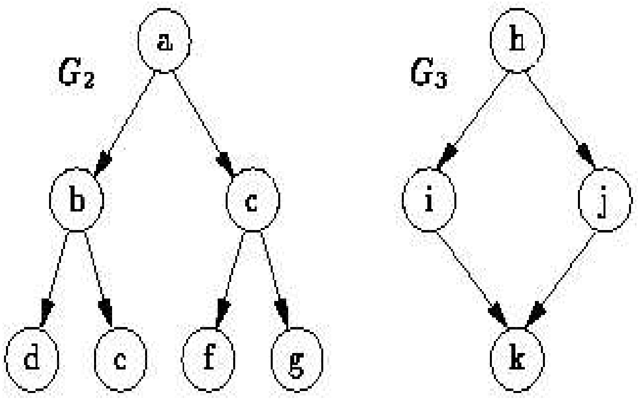
The above graph contains 6 vertices and 15 edges.

5. Strongly Connected Graph and Weekly Connected Graph



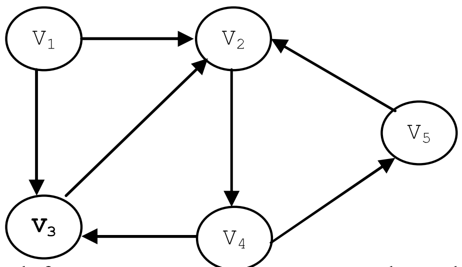
6. Acyclic Graph

A directed graph which has no cycles is referred to as acyclic graph. It is abbreviated as DAG (DAG - Directed Acyclic Graph).



Path

A path in a graph is a sequence of vertices $w_1, w_2, w_3, \ldots$, wn such that $(w_i , w_{i+1}) \varepsilon E$ for $1 \leq i < N$.



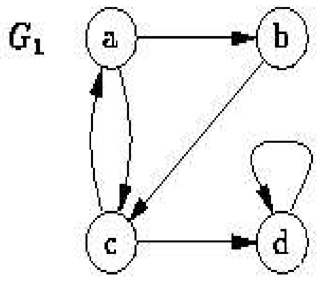
The path from vertex v_1 to v_4 as v_1, v_2, v_4 The path from vertex v_1 to v_5 as v_1, v_2, v_4, v_5.

Path Length

The length of the path is the number of edges on the path, which is equal to N-1, where N represents the number of vertices. The length of the above path v1 to v5 as 3 . (i.e) (V1, V2), (V2, V4) , (v4,v5). If there is a path from a vertex to itself, with no edges, then the path length is 0.

Loop

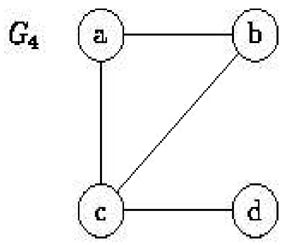
If the graph contains an edge (v, v) from a vertex to itself, then the path v,v is referred to as a loop.



The edge (d,d) is called as loop.

Simple Path

A simple path is a path such that all vertices on the path, except possibly the first and the last are distinct.



In the above graph the path (a ,b ,c, d) is a simple path. **Cycle** A cycle in a graph is a path in which the first and last vertexes are the same. In the above graph a, b, a is a cycle A graph which has cycles is referred to as cyclic graph.

Simple cycle

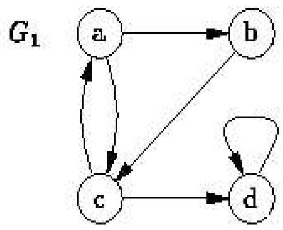
A simple cycle is the simple path of length at least one that begins and ends at the same vertex.

Degree

The number of edges incident on a vertex determines its degree. The degree of the vertex V is written as degree (V). The above graph degree of a vertex c is 3

Indegree and Outdegree

The indegree of the vertex V, is the number of edges entering into the vertex V. Similarly the out degree of the vertex V is the number of edges exiting from that vertex V.



The Indegree of vertex c as 3 The Outdegree of vertex C as 1

4.4.2. Graph Representations

Representation of Graph

Graph can be represented by two ways

i) Adjacency Matrix

ii) Adjacency list.

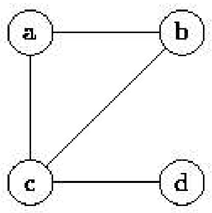
Adjacency Matrix One simple way to represents a graph is Adjacency Matrix. The adjacency Matrix A for a graph G = (V, E) with n vertices is an n x n matrix, such that

Aij = 1, if there is an edge Vi to Vj

Aij = 0, if there is no edge.

Example

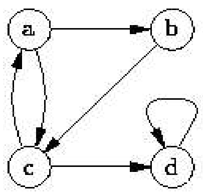
Adjacency Matrix for an Undirected Graph



$$A_1 = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 1 & 0 \\ b & 0 & 0 & 1 & 0 \\ c & 1 & 0 & 0 & 1 \\ d & 0 & 0 & 0 & 1 \end{array}$$

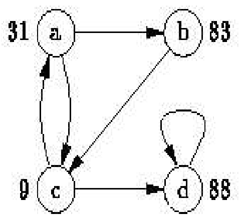
In the matrix (3,4) th data represent the presence of an edge between the vertices c and d.

Adjacency matrix for an directed graph



$$A_4 = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 1 & 0 \\ b & 1 & 0 & 1 & 0 \\ c & 1 & 1 & 0 & 1 \\ d & 0 & 0 & 1 & 0 \end{array}$$

Adjacency Matrix for an Weighted Graph



$$A_6 = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & \infty & 31 & 41 & \infty \\ b & 31 & \infty & 59 & \infty \\ c & 41 & 59 & \infty & 26 \\ d & \infty & \infty & 26 & \infty \end{array}$$

Comparing two Representation

Space Matrix:

Adjacency Matrix is O(|V|2) -Static Representation

Adjacency List is O(|V| + |E|) –Dynamic Representation

Advantage

Simple to implement.

Disadvantage

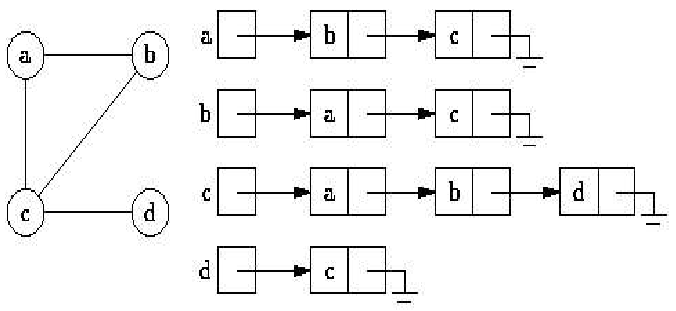
a. Takes O(n2) space to represents the graph

b. It takes O(n2) time to solve the most of the problems.

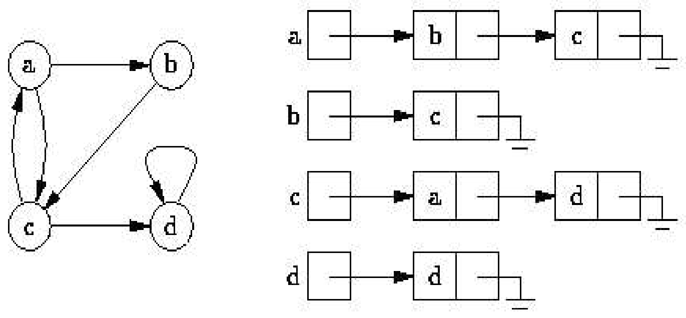
Adjacency List

In this representation for each vertex a list is maintained to keep all its adjacent vertices. It is the standard way to represent graphs

Example



Adjacency List For a Directed Graph



4.4.3. Graph Traversals

Traversing a graph is visiting each of its elements(nodes) in a systematic manner.

Condition is, when traversing a graph, we must be careful to avoid going round in circles. We do this by making all vertices which have already been visited.

Types of Traversal,

- Depth First Search
- Breadth First Search

4.4.3.1. Breadth First Traversal

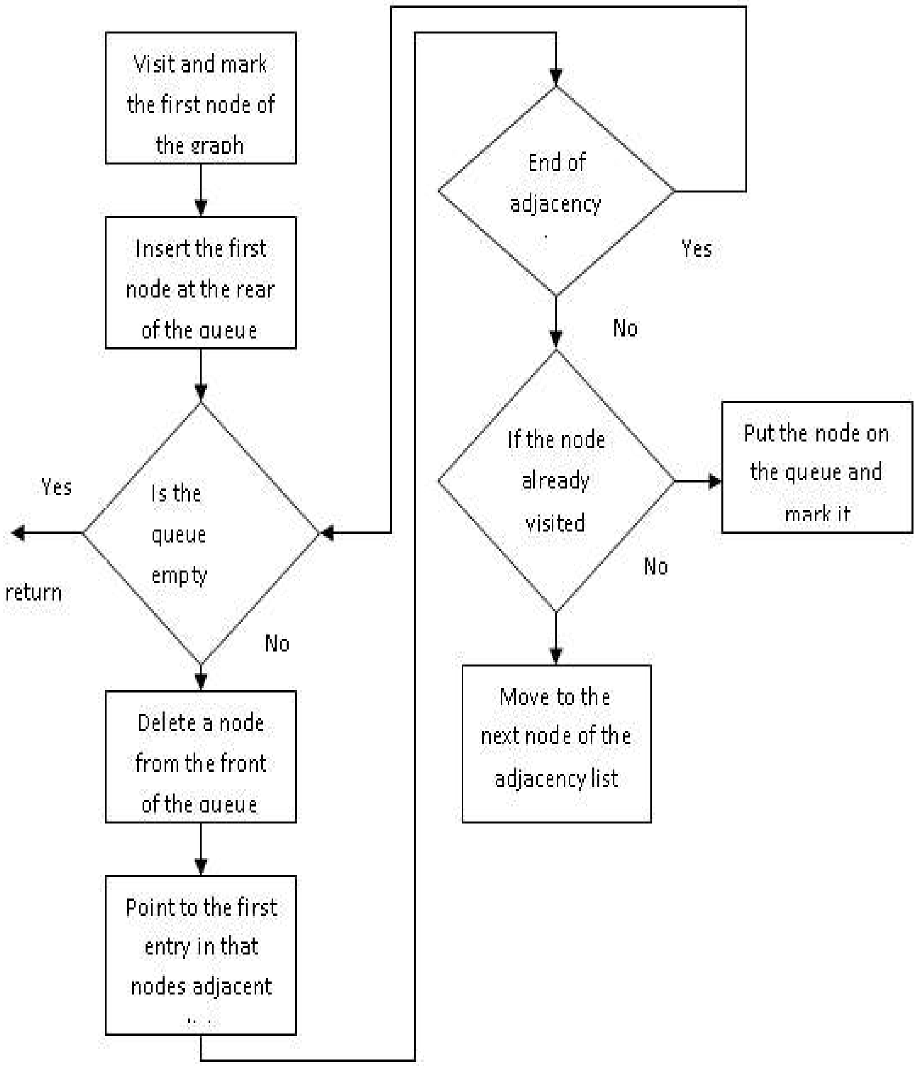
It uses a queue to keep track of which adjacent vertices might still be unproposed.

All unvisited vertices adjacent to v are visited after visiting the starting vertex v and narking it as visited.

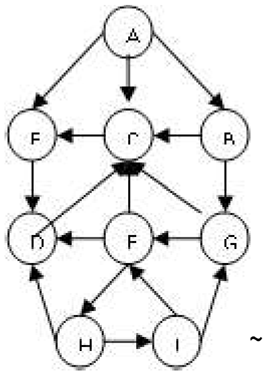
i. First we examine the starting node A.
ii. Then we examine all the neighbors of A.
iii. Then we examine all the neighbors of the neighbors of A and so on.
iv. Here, we need to keep track of the neighbors of a node, and we need to guarantee that no node is processed more than one.
v. This algorithm uses a queue to store the nodes of each level of graph as and when they are visited.
vi. These nodes are then taken one by one and their adjacent nodes are visited and so on until all nodes have been visited
vii. The algorithm terminate when the queue becomes empty.

Flow chart

Steps are,


Visit and mark the first node of the graph
Insert the first node at the rear of the queue
Is the queue empty
Yes
return
No
Delete a node from the front of the queue
Point to the first entry in that nodes adjacent
End of adjacency
Yes
No
If the node already visited
Put the node on the queue and mark it
No
Move to the next node of the adjacency list

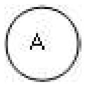
Example

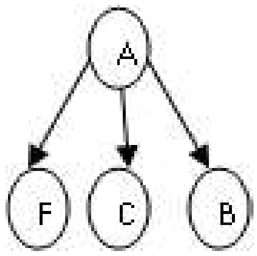


i. Initially add A to QUEUE and add NULL to ORIGIN(array) follows:

FRONT=1 QUEUE=A

REAR=1 ORIGIN=0

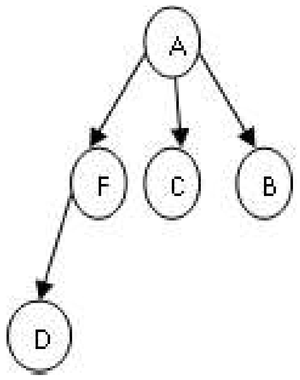




ii. Remove the front element A from QUEUE by setting FRONT:=FRONT+1 and add to QUEUE the neighbors of A as follows:

FRONT=1 QUEUE=A

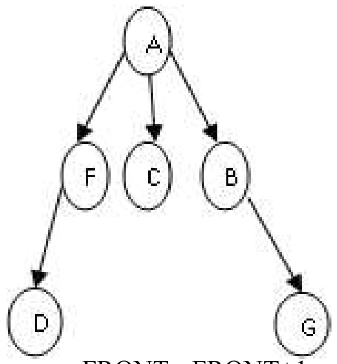
REAR=1 ORIGIN=0



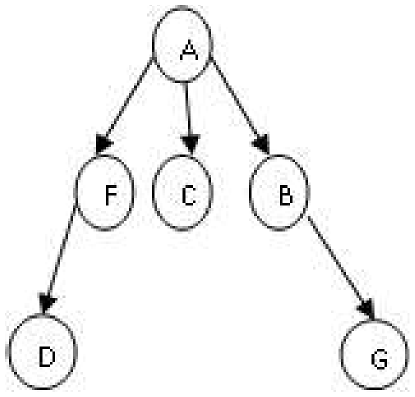
iii. Remove the front element F from QUEUE by setting FRONT:=FRONT+1 and add to QUEUE the neighbors of F as follows:

FRONT=3 QUEUE=A, F, C, B, D

REAR=5 ORIGIN=0, A, A, A, F



iv. Remove the front element B from QUEUE by setting FRONT:=FRONT+1 and add to QUEUE the neighbors of B as follows:



FRONT=5

QUEUE=A, F, C, B, D, G

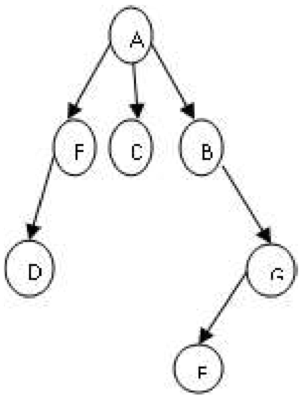
REAR=6 ORIGIN=0, A, A, A, F, B

B adjacent=C, G [Here C is already visited]

v. Remove the front element B from QUEUE by setting FRONT:=FRONT+1 and add to QUEUE the neighbors of B as follows:

FRONT=6 QUEUE=A, F, C, B, D, G

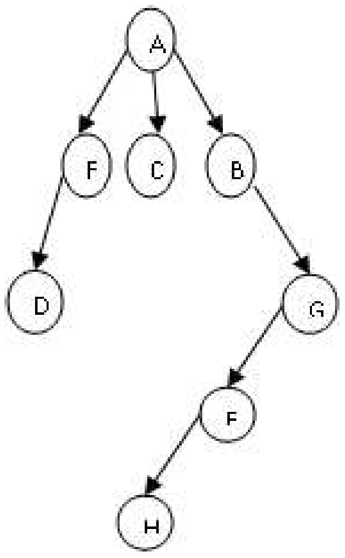
REAR=6 ORIGIN=0, A, A, A, F, B



vi. Remove the front element G from QUEUE by setting FRONT:=FRONT+1 and add to QUEUE the neighbors of G as follows:

FRONT=7 QUEUE=A, F, C, B, D, G, E

REAR=7 ORIGIN=0, A, A, A, F, B, G



vii. Remove the front element E from QUEUE by setting FRONT:=FRONT+1 and add to QUEUE the neighbors of E as follows:

FRONT=8

QUEUE=A, F, C, B, D, G, E, H

REAR=8

ORIGIN=0, A, A, A, F, B, G, E

H is added to QUEUE, This is the final destination.

From the above example ORIGIN array values are required path H<-E<-G<-B<-A.

Algorithm BFS(G). Here input is graph G=<V,E> and output is Graph G or with its vertices marked with consecutive integers.

```
count=0
for each vertex $v_1$ in V
do
        if $v_1$ is marked with 0
                BFS(V)
                        count=count+1
        while the queue is not empty do
                for each vertex w in v adjacent to the front's vertex $v_i$ do
                        if w is marked with 0
                                count=count+1
                add w to the queue
        remove vertex v from the front of the queue.
```

Application of BFT

i. To check whether the graph is connected or not.

ii. To determine whether a graph is cycle.

iii. The unweighted graph, BFS find the shortest path.

4.4.3.2. Depth First Traversal

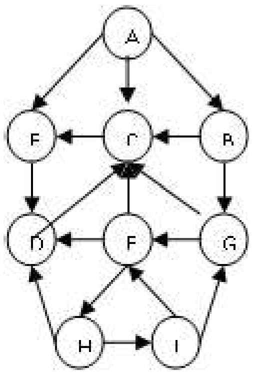
It keeps trying to move forward in the graph until searching a vertex with no outgoing edges to unmarked vertices.

- This done by STACK
- The DFS algorithm works as follows:
  i. First we examine the starting node(first vertex)
  ii. Then we examine each node N along a path P which begins at first vertex.

For example, the starting vertex V is visited first. Let w_1, w_2, w_3,…,w_n be the vertices adjacent to V. Then the vertex w_1 is visited next. After visiting w_1 all vertices adjacent to w_1 are visited in depth first manner before returning to traverse w_2,…,w_n.

The algorithm terminates when no visited vertex can be reached from any of the visited once.

Example



Suppose we want to find and print all the nodes reachable from the node H (including the node H itself). The DFS(G) starting from node H is as follows:

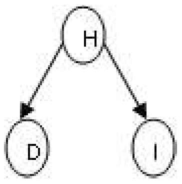
i. Push H on to the stack.

STACK : H

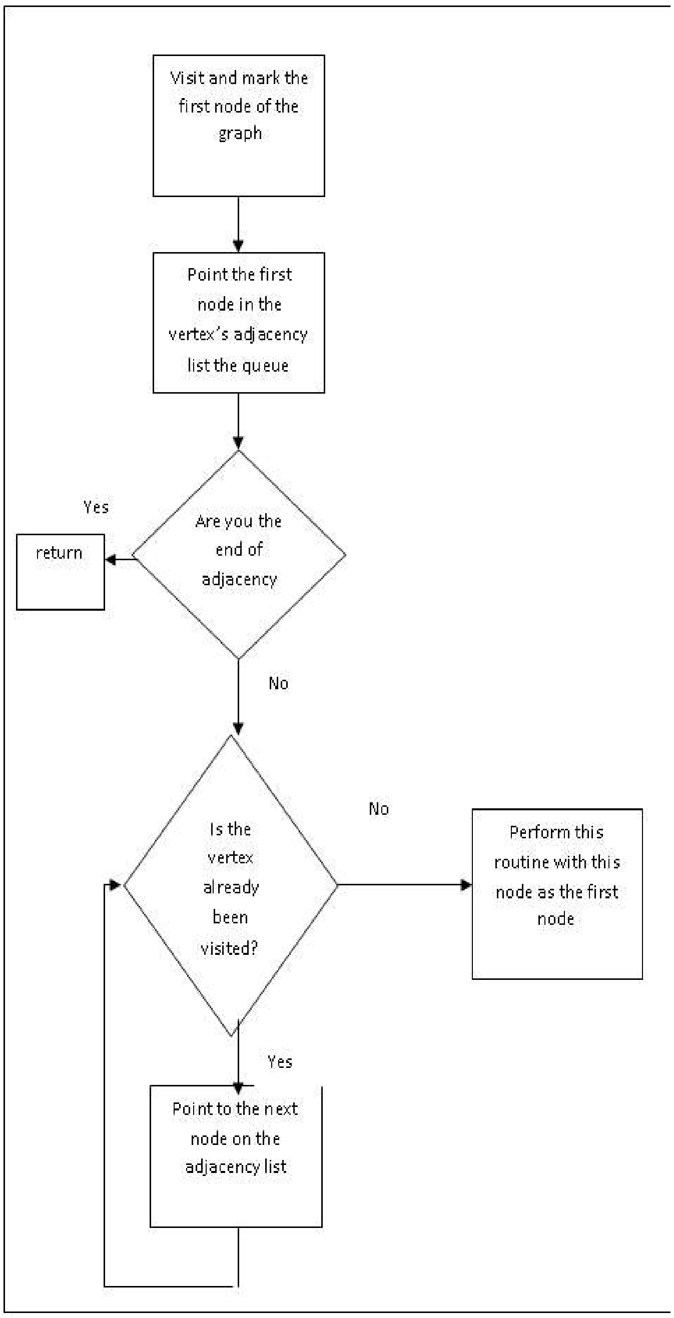
ii. Pop and print the top element H and then push onto the stack all the neighbors of H.

PRINT : H

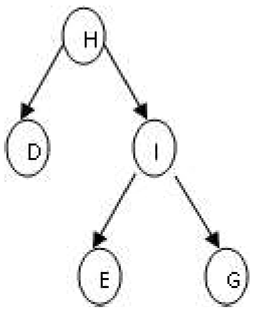
STACK : D, I



Flow chart


Visit and mark the first node of the graph
Point the first node in the vertex's adjacency list the queue
Are you the end of adjacency
Yes
return
No
Is the vertex already been visited?
No
Perform this routine with this node as the first node
Yes
Point to the next node on the adjacency list

iii. Pop and print the top element I and then push onto the stack all the neighbors of I.



PRINT : I

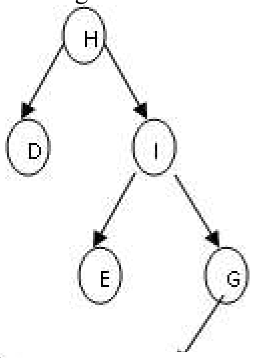
STACK : D, E, G

iv. Pop and print the top element G and then push onto the stack all the neighbors of G.

PRINT : G

STACK : D, E, C

- Here C is pushed onto the stack & E has already pushed on to the stack



v. Pop and print the top element C and then push onto the stack all the neighbors of C.

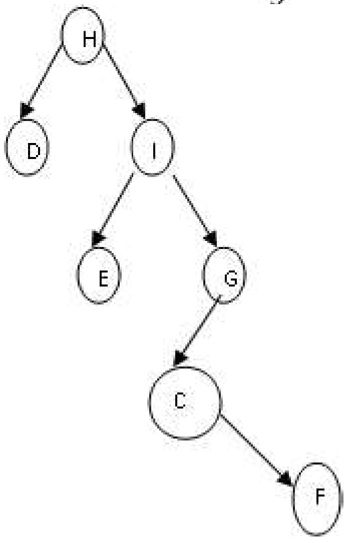
PRINT : C

STACK : D, E, F

vi. Pop and print the top element F and then push onto the stack all the neighbors of F.

PRINT : F

STACK : D, E



vii. Pop and print the top element E and then push onto the stack all the neighbors of E.

PRINT : E

STACK : D

viii. Pop and print the top element D and then push onto the stack all the neighbors of D.

PRINT : D

STACK : EMPTY

The stack is now empty and DFS(G) starting from H is now complete. The nodes which are reachable from H are: H, I, G, C, F, E, D

Algorithm DFS(G)

```
DFS(G)
    count=0
    for each vertex $v_1$ in V do
    if $v_1$ is marked with 0
        dfs(v)
```

```
            count=count+1
        for each vertex w in v adjacency to $v_1$ do
            if w is marked with 0
```

Efficiency of DFS (or) DFT

i. In an adjacency matrix implementation, traversing of all successor of a node is $O(n^2)$
ii. In an adjacency list representation, traversing of all successor of a node is O(n+e). here n represent nodes represent edge.

Difference between BFS and DFS

Sl. No.	Breadth first Traversal	Depth First Traversal
1	It uses a queue to keep track of which adjacent vertices might still be unprocessed	It keeps trying to move forward in the graph, until reaching a vertex with no outgoing edges to unmarked vertices.
2	It is done by Queue	It is done by Stack

4.4.4. Connected components.

A graph is ***connected*** if, for any two vertices, there is a path between them. If a graph *G* is not connected, its maximal connected subgraphs are called the ***connected components*** of *G*.

```
/* Components(G):: */ // count components
int operator()
{
      initialize(); // initialize DFS
      nComponents = 0; // init vertex count
      VertexList verts = graph.vertices();
     for(VertexItor pv=verts.begin();pv!=verts.end();++pv)
      {
            if (!isVisited(*pv))
            {       // not yet visited?
                  dfsTraversal(*pv); // visit
                  nComponents++; // one more component
            }
      }
      return nComponents;
}
```

Data structures

Data structure	Time Complexity: Avg: Indexing	Time Complexity: Avg: Search	Time Complexity: Avg: Insertion o(n2)	Time Complexity: Avg: Deletion	Time Complexity: Worst: Indexing	Time Complexity: Worst: Search	Time Complexity: Worst: Insertion	Time Complexity: Worst: Deletion	Space Complexity: Worst
Basic Array	O(1)	O(n)	-	-	O(1)	O(n)	-	-	O(n)
Dynamic array	O(1)	O(n)	O(n)	-	O(1)	O(n)	O(n)	-	O(n)
Singly linked list	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly linked list	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Hash table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary									

search tree	–	O((log n))	O((log n))	O((log n))	–	O(n)	O(n)	O(n)	O(n)
B-tree	–	O((log n))	O((log n))	O((log n))	–	O((log n))	O((log n))	O((log n))	O(n)
Red-black tree	–	O((log n))	O((log n))	O((log n))	–	O((log n))	O((log n))	O((log n))	O(n)
AVL tree	–	O((log n))	O((log n))	O((log n))	–	O((log n))	O((log n))	O((log n))	O(n)

Graph search

Node/Edge management	Storage	Add vertex	Add edge	Remove vertex	Remove edge	Query
Adjacency list	O(|V|+|E|)	O(1)	O(1)	O(|E|)	O(|E|)	O(|V|)
incidence list	O(|V|+|E|)	O(1)	O(1)	O(|E|)	O(|E|)	O(|E|)
Adjacency matrix	O(|V|^2)	O(|V|^2)	O(1)	O(|V|^2)	O(1)	O(1)
incidence matrix	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|E|)