U N I T - 2

Trees – Binary tree representations – Tree Traversal – Threaded Binary Trees – Binary Tree Representation of Trees – Graphs and Representations – Traversals, Connected Components and Spanning Trees – Shortest Paths and Transitive closure – Activity Networks – Topological Sort and Critical Paths .

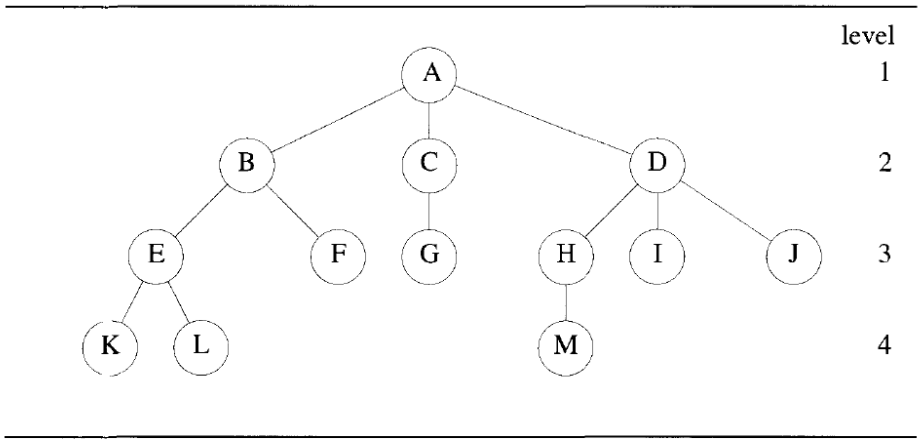
TREES

BASIC TERMINOLOGY

A *tree* is a finite set of one or more nodes such that: (i) there is a specially designated node called the *root*; (ii) the remaining nodes are partitioned into $n \geq 0$ disjoint sets $T1, ..., Tn$ where each of these sets is a tree. $T1, ..., Tn$ are called the *subtrees* of the root.

Terminoloy

A *node* stands for the item of information plus the branches to other items.



This tree has 13 nodes. The root contains A and we normally draw trees with their roots at the top.

The number of subtrees of a node is called its *degree*. The degree of A is 3, of *C* is 1, and of F is 0.

Nodes that have degree zero are called *leaf* or *terminal* nodes. The set **{K,** L, F, G, M, I, J} is the set of leaf nodes.

The other nodes are referred to as *nonterminals.*

The roots of the subtrees of a node *X* arc the *children* of *X*. The node *X* is the *parent* of its children. Thus the children of **D** are H, I, and J, and the parent of **D** is A.

Children of the same parent are said to be *siblings.* For example H, I, and J are siblings.

The *degree* of a tree is the maximum degree of the nodes in the tree. The above tree 5 has degree 3.

The *ancestors* of a node are all the nodes along the path from the root to that node. The ancestors of M are A, **D,** and H.

The *level* of a node is defined by initially letting the root be at level one. If a node is at level *p*, then its children are at level $p + 1$.

The *height* or *depth* of a tree is defined to be the maximum level of any node in the tree.

A *forest* is a set of $n \geq 0$ disjoint trees.

BINARY TREES

A *binary tree* is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the *left* and *right* subtrees.

Two sample binary trees

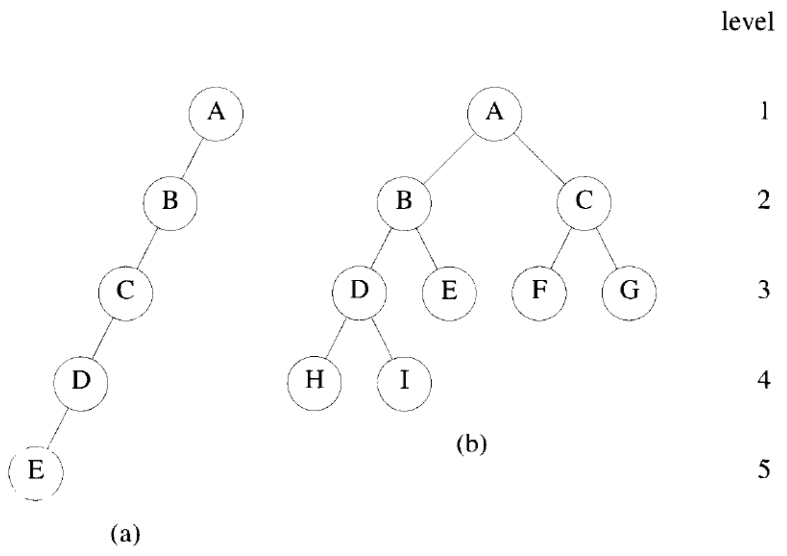


Figure (a) is a *skewed* tree, skewed to the left.
The tree in Figure (b) is called a *complete* binary tree.

The maximum number of nodes on level i of a binary tree is 2^{i-1}. Also, the maximum number of nodes in a binary tree of depth *k* is 2^{k-1}, $k > 0$.

The binary tree of depth *k* that has exactly 2k - 1 nodes is called a *full* binary tree of depth *k*. *A* full binary tree of depth 4 is shown below.

A binary tree with *n* nodes and depth *k* is *complete* iff its nodes correspond to the nodes that are numbered one to *n* in the full binary tree of depth *k*.

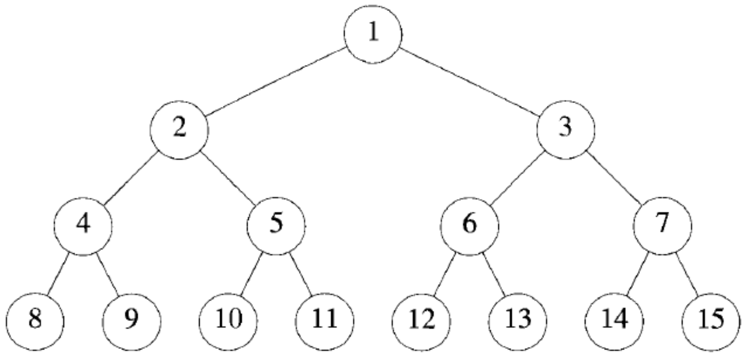
For any nonempty binary tree, *T*, if n_0 is the number of terminal nodes and n_2 the numberof nodes of degree 2, then $n_0 = n_2 + 1$.

A *full* binary tree of depth *k* is a binary tree of depth *k* having $2^k - 1$ nodes. This is the maximum number of nodes such a binary tree can have.

BINARY TREE REPRESENTATIONS

A binary tree data structure is represented using two methods. Those methods are as follows:

1. Array Representation
2. Linked List Representation



Array Representation

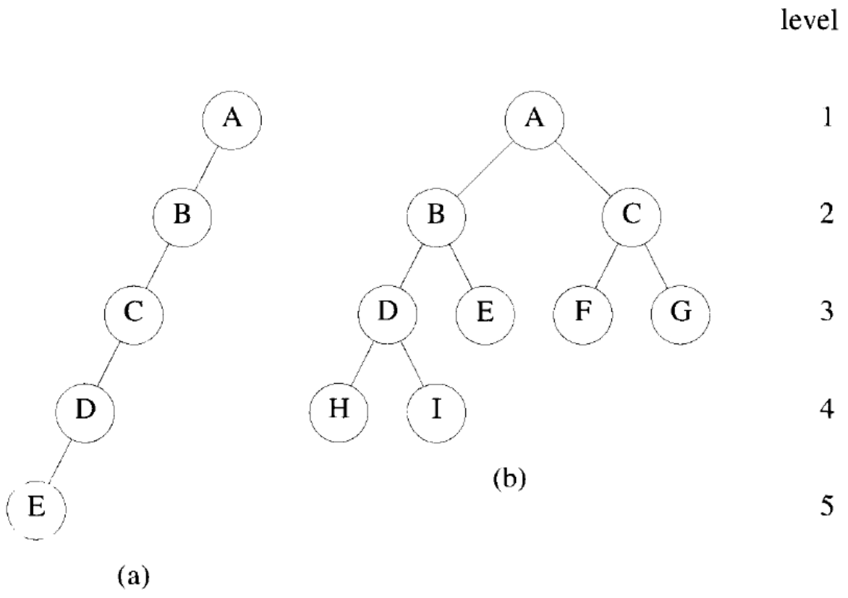
If a complete binary tree with n nodes (i.e., depth= [$\log_2 n$] + 1) is represented sequentially as above then for any node with index i, $1 \le i \le n$ we have:

(i) PARENT(i) is at [i/2] if $i \neq 1$. When $i = 1$, i is the root and has no parent.

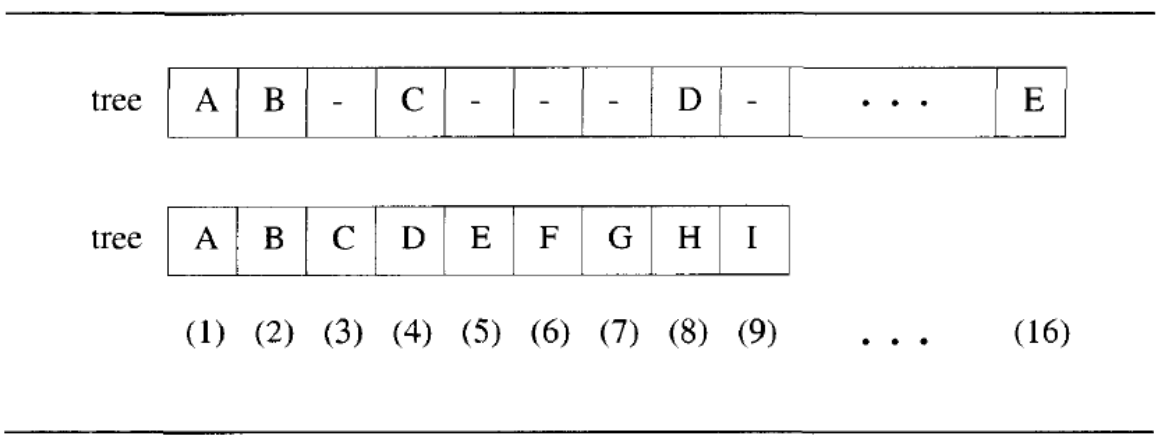
(ii) LCHILD(i) is at 2i if $2i \le n$. If $2i > n$, then i has no left child.

(iii) RCHILD(i) is at $2i + 1$ if $2i + 1 \le n$. If $2i + 1 > n$, then i has no right child.

Example tree:



In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree. Consider the above example of a binary tree and it is represented as follows:



tree	A	B	-	C	-	-	-	D	-	. . .	E
tree	A	B	C	D	E	F	G	H	I		
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	. . .	(16)

For complete binary trees the representation is ideal as no space is wasted. For the skewed tree less than half the array is utilized.

Linked List Representation of Binary Tree:

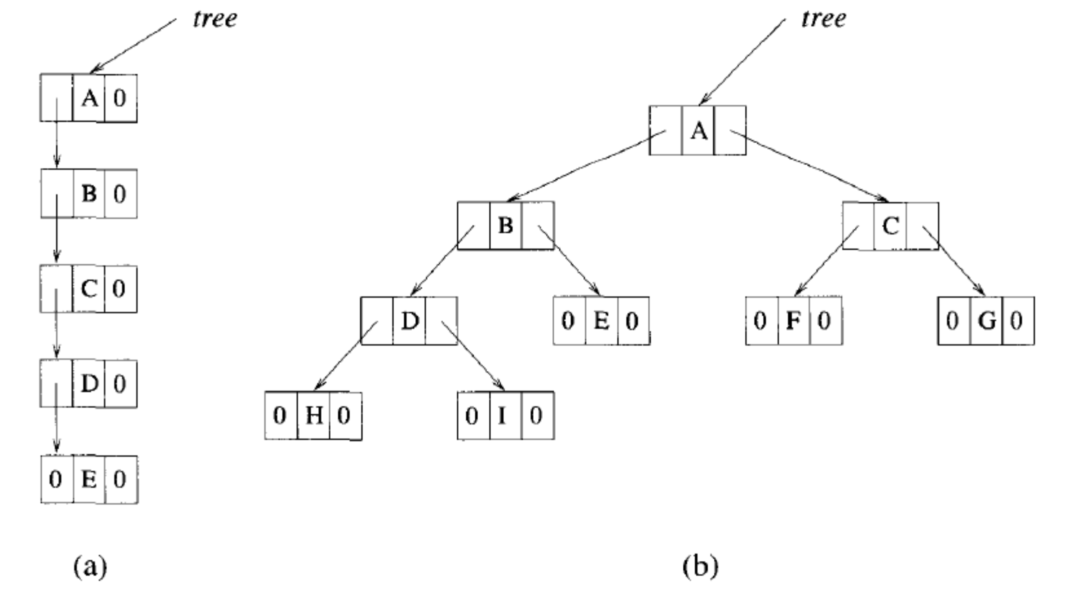
While the above representation appears to be good for complete binary trees it is wasteful for many other binary trees.

In addition, the representation suffers from the general inadequacies of sequential representations.

Insertion or deletion of nodes from the middle of a tree requires the movement of potentially many nodes to reflect the change in level number of these nodes.

These problems can be easily overcome through the use of a linked representation. Each node will have three fields LCHILD, DATA and RCHILD as below:

LCHILD	DATA	RCHILD



BINARY TREE TRAVERSAL

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree.

Let L, D, R stand for moving left, printing the data, and moving right.

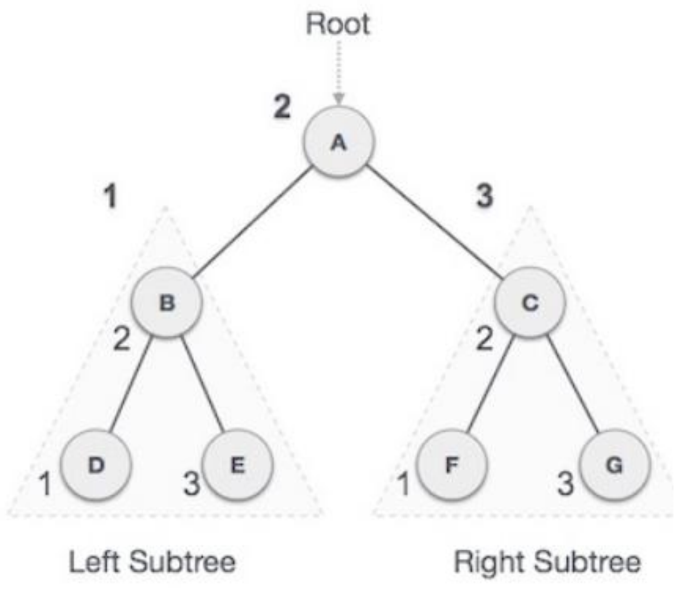
- LDR → Inorder traversal
- LRD → Postorder traversal
- DLR → Preorder traversal

Inorder Traversal:

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree.

```
procedure INORDER(T)
//T is a binary tree where each node has three fields LCHILD, DATA,RCHILD//
if T≠ 0 then
        [call INORDER(LCHILD(T))
        print(DATA(T))
        call(INORDER(RCHILD(T))]
end INORDER
```

Example:



The output of inorder traversal of this tree will be:

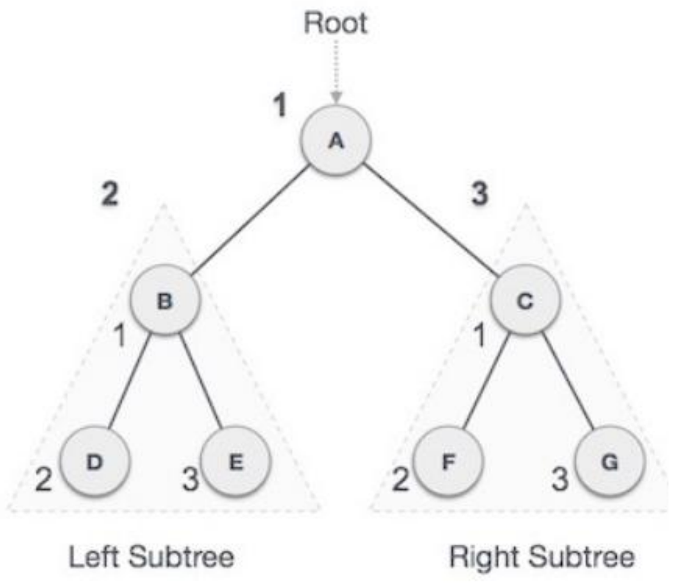
D → B → E → A → F → C → G

Preorder Traversal:

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

```
procedure PREORDER (T)
//T is a binary tree where each node has three fields LCHILD, DATA,RCHILD//
if T≠ 0 then
        [print (DATA(T))
        call PREORDER(LCHILD(T))
        call PREORDER(RCHILD(T))]]
end PREORDER
```

Example:



The output of pre-order traversal of this tree will be:

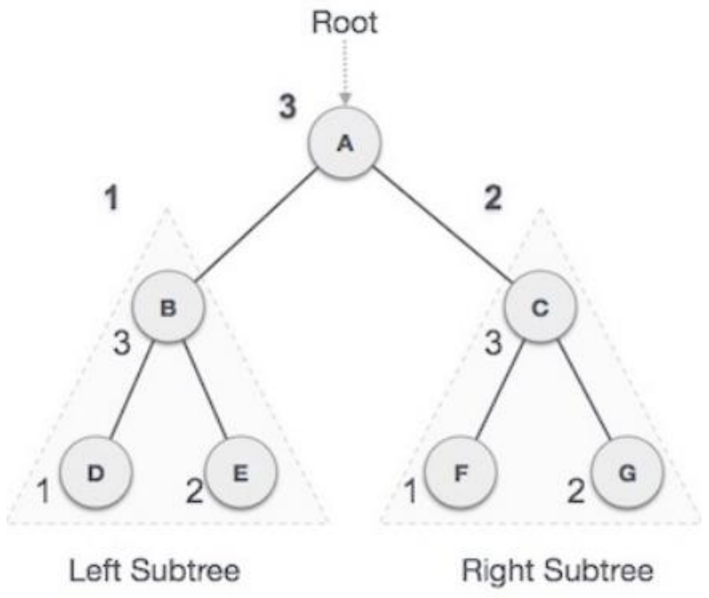
A → B → D → E → C → F → G

Postorder Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.

```
procedure POSTORDER (T)
//T is a binary tree where each node has three fields LCHILD, DATA,RCHILD//
if T≠ 0 then
        [call POSTORDER(LCHILD(T))
        call POSTORDER(RCHILD(T))
        print (DATA(T))]
end POSTORDER
```

Example:



The output of post-order traversal of this tree will be:

D → E → B → F → G → C → A

<u>THREADED BINARY TREE</u>

In the linked representation of binary trees, more than one half of the link fields contain NULL values which results in wastage of storage space.

If a binary tree consists of **n** nodes then **n+1** link fields contain NULL values.

So in order to effectively manage the space, a method was devised by Perlis and Thornton in which the NULL links are replaced with special links known as threads.

Such binary trees with threads are known as **threaded binary trees**. Each node in a threaded binary tree either contains a link to its child node or thread to other nodes in the tree.

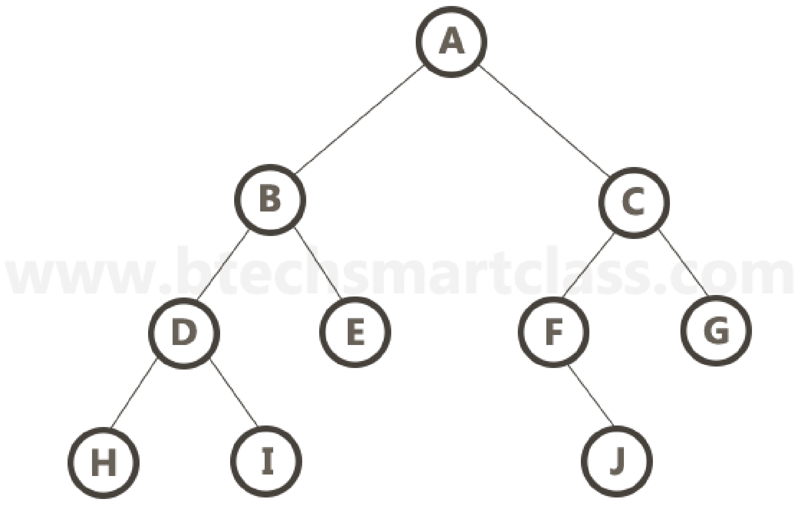
If the RCHILD(P) is normally equal to zero, we will replace it by a pointer to the node which would be printed after P when traversing the tree in inorder.

A null LCHILD link at node P is replaced by a pointer to the node which immediately precedes node P in inorder.

In the memory representation we must be able to distinguish between threads and normal pointers. This is done 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

Example:



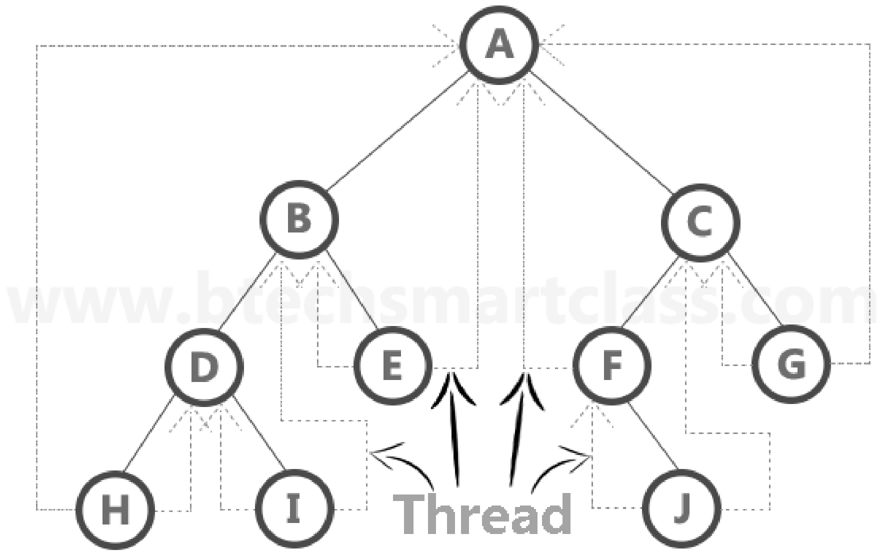
In-order traversal of above binary tree is H - D - I - B - E - A - F - J - C – G

When we represent the above binary tree using linked list representation, nodes H, I, E, F, J and G left child pointers are NULL.

This NULL is replaced by address of its in-order predecessor respectively (I to D, E to B, F to A, J to F and G to C), but here the node H does not have its in-order predecessor, so it points to the root node A.

And nodes H, I, E, J and G right child pointers are NULL. These NULL pointers are replaced by address of its in-order successor respectively (H to D, I to B, E to A, and J to C), but here the node G does not have its in-order successor, so it points to the root node A.

Above example binary tree is converted into threaded binary tree as follows.



APPLICATIONS OF TREES

- Set Representation
- Decision Trees
- Game Trees

GRAPHS:

TERMINOLOGY

A graph, G, consists of two sets V and E.

V is a finite non-empty set of vertices.

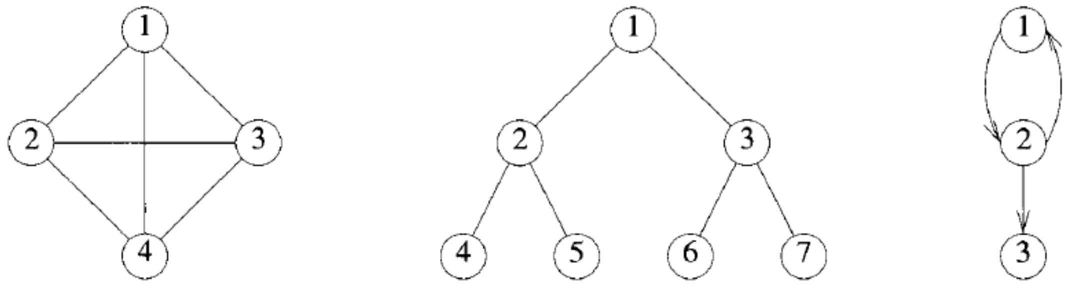
E is a set of pairs of vertices, these pairs are called edges.

V(G) and E(G) will represent the sets of vertices and edges of graph G. We will also write G = (V,E) to represent a graph.

In an undirected graph the pair of vertices representing any edge is unordered . Thus, the pairs (v1, v2) and (v2, v1) represent the same edge.

In a directed graph each edge is represented by a directed pair (v1, v2). v1 is the tail and v2 the head of the edge. Therefore <v2, v1> and <v1, v2> represent two different edges.

Figure shown below has three graphs G1, G2 and G3.



(a) G_1 (b) G_2 (c) G_3

The graphs G1 and G2 are undirected. G3 is a directed graph.

V (G1) = {1,2,3,4};	E(G1) = {(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)}
V (G2) = {1,2,3,4,5,6,7};	E(G2) = {(1,2),(1,3),(2,4),(2,5),(3,6),(3,7)}
V (G3) = {1,2,3};	E(G3) = {<1,2>, <2,1>, <2,3>}

An n vertex undirected graph with exactly n(n - 1)/2 edges is said to be complete.

G1 is the complete graph on 4 vertices while G2 and G3 are not complete graphs.

In the case of a directed graph on n vertices the maximum number of edges is n (n - 1).

If (v1,v2) is an edge in E(G), then we shall say the vertices v1 and v2 are adjacent and that the edge (v1,v2) is incident on vertices v1 and v2.

The vertices adjacent to vertex 2 in G2 are 4, 5 and 1. The edges incident on vertex 3 in G2 are (1,3), (3,6) and (3,7). If <v1,v2> is a directed edge, then vertex v1 will be said to be adjacent to v2 while v2 is adjacent from v1.

The edge <v1,v2> is incident to v1 and v2. In G3 the edges incident to vertex 2 are <1,2>, <2,1> and <2,3>.

A path from vertex vp to vertex vq in graph G is a sequence of vertices vp,vi1,vi2, ...,vin,vq such that (vp,vi1),(vi1,vi2), ...,(vin,vq) are edges in E(G). If G' is directed then the path consists of <vp,vi1>,<vi,vi2>, ..., <vin,vq>, edges in E(G'). The length of a path is the number of edges on it. A simple path is a path in which all vertices except possibly the first and last are distinct.

A connected component or simply a component of an undirected graph is a maximal connected subgraph. G4 has two components H1 and H2 (see figure 6.5). A tree is a connected acyclic (i.e., has no cycles) graph.

The degree of a vertex is the number of edges incident to that vertex. The degree of vertex 1 in G1 is 3.

In case G is a directed graph, we define the in-degree of a vertex v to be the number of edges for which v is the head.

The out-degree is defined to be the number of edges for which v is the tail. Vertex 2 of G3 has in-degree 1, out-degree 2 and degree 3.

A graph with weighted edges is called a ***network***.

GRAPH REPRESENTATIONS

There are several representations for graphs are possible. The three most commonly used graph representations are:

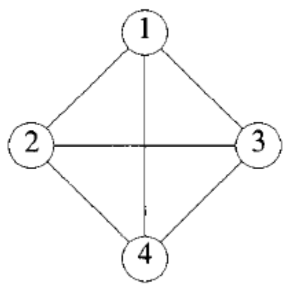
- Adjacency matrices
- Adjacency lists
- Adjacency multilists.

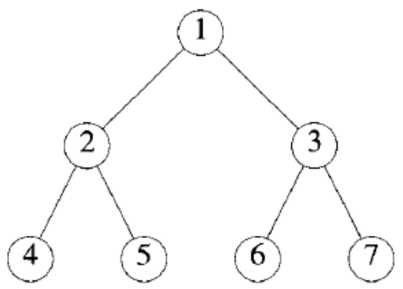
Adjacency Matrix

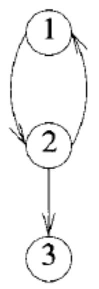
Let G = (V,E) be a graph with n vertices, n ≥1. The adjacency matrix of G is a 2-dimensional n x n array, say A, with the property that A(i,j) = 1 iff the edge (vi,vj) (<vi,vj> for a directed graph) is in E(G). A(i,j) = 0 if there is no such edge in G.

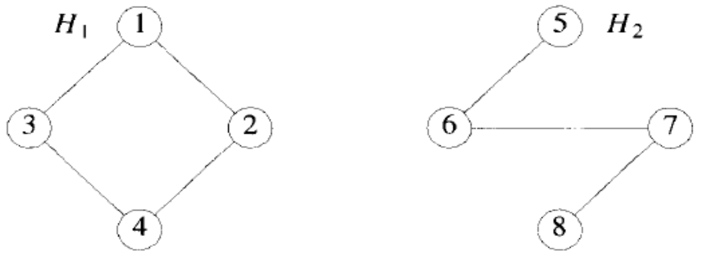
The adjacency matrix for an undirected graph is symmetric as the edge (vi,vj) is in E(G) iff the edge (vj,vi) is also in E(G). The adjacency matrix for a directed graph need not be symmetric.

The adjacency matrices for the graphs G1, G3 and G4 are as follows:


(a) G_1


(b) G_2


(c) G_3



G_4

$$\begin{array}{c|cccc|}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 1 & 1 & 1 \\
2 & 1 & 0 & 1 & 1 \\
3 & 1 & 1 & 0 & 1 \\
4 & 1 & 1 & 1 & 0
\end{array}$$

(a) G_1

$$\begin{array}{c|ccc|}
 & 1 & 2 & 3 \\
\hline
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 1 \\
3 & 0 & 0 & 0
\end{array}$$

(b) G_3

$$\begin{array}{c|cccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
3 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
4 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
7 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{array}$$

(c) G_4

For an undirected graph the degree of any vertex *i* is its row sum. For a directed graph the row sum is the out-degree while the column sum is the in-degree.

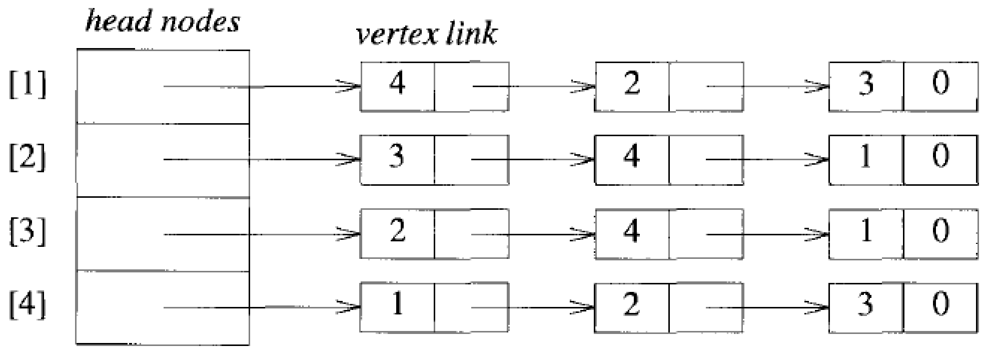
Adjacency Lists

Using adjacency matrices all algorithms will require at least $O(n^2)$ time as $n^2 - n$ entries of the matrix (diagonal entries are zero) have to be examined. When graphs are sparse, i.e., most of the terms in the adjacency matrix are zero. To overcome this problem, the linked representation is used.

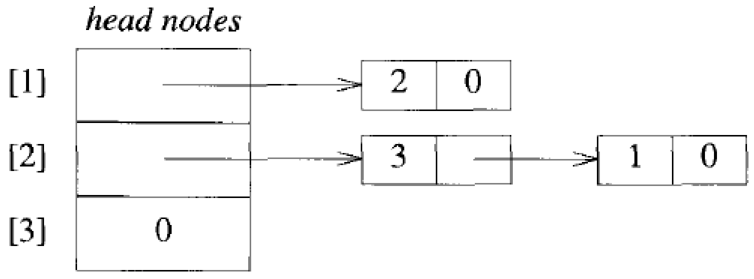
In this representation the n rows of the adjacency matrix are represented as n linked lists. There is one list for each vertex in G.

The nodes in list i represent the vertices that are adjacent from vertex i. Each node has at least two fields: VERTEX and LINK.

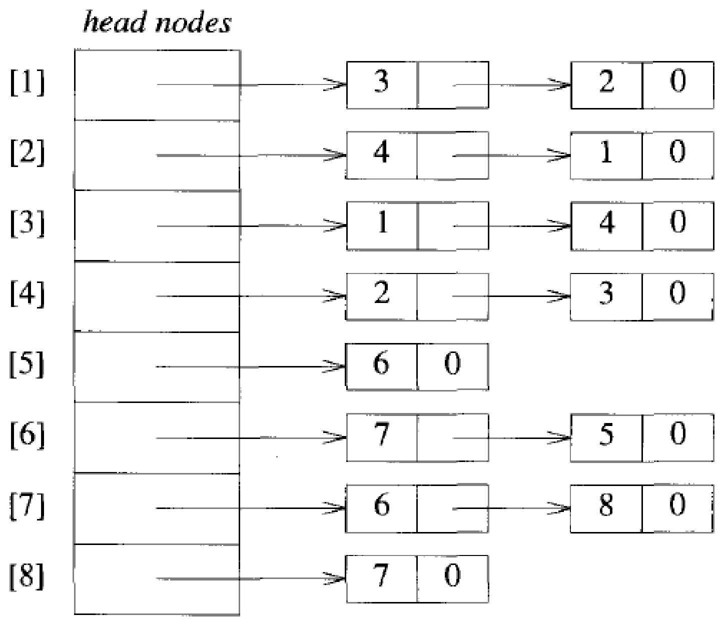
The VERTEX fields contain the indices of the vertices adjacent to vertex i. The adjacency lists for G1, G3 and G4 are shown below. Each list has a headnode. The headnodes are sequential providing easy random access to the adjacency list for any particular vertex.



(a) G_1



(b) G_3



(c) G_4

Adjacency Multilists

In the adjacency list representation of an undirected graph each edge (v_i,v_j) is represented by two entries, one on the list for v_i and the other on the list for v_j.

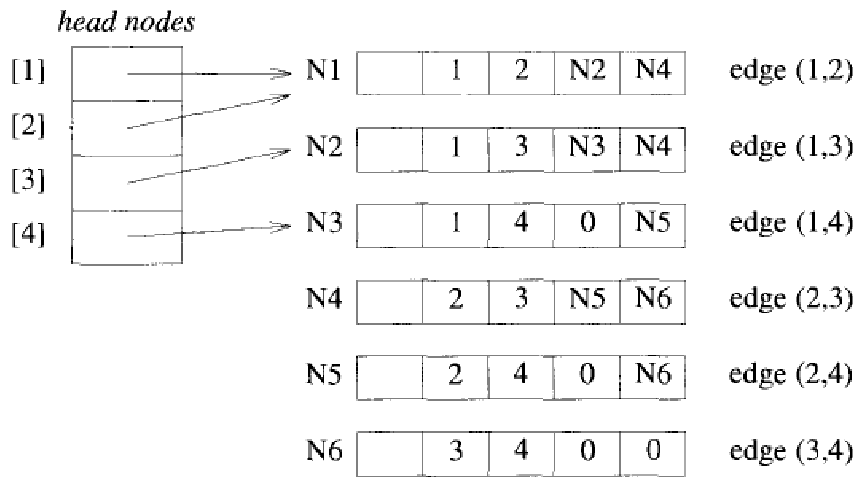
The node structure now becomes

m	*vertex1*	*vertex2*	*list1*	*list2*

where M is a one bi mark field that may be used to indicate whether or not the edge has been examined.

The storage requirements are the same as for normal adjacency lists except for the addition of the mark bit M.

The adjacency multilists for G1 is shown below:



The lists are vertex 1: N1 → N2 → N3
vertex 2: N1 → N4 → N5
vertex 3: N2 → N4 → N6
vertex 4: N3 → N5 → N6

TRAVERSALS, CONNECTED COMPONENTS AND SPANNING TREES

Given the root node of a binary tree, one of the most common things one wishes to do is to traverse the tree and visit the nodes in some order.

Given an undirected graph G = (V,E) and a vertex v in V(G) we are interested in visiting all vertices in G that are reachable from v (i.e., all vertices connected to v).

Graph traversal techniques are:

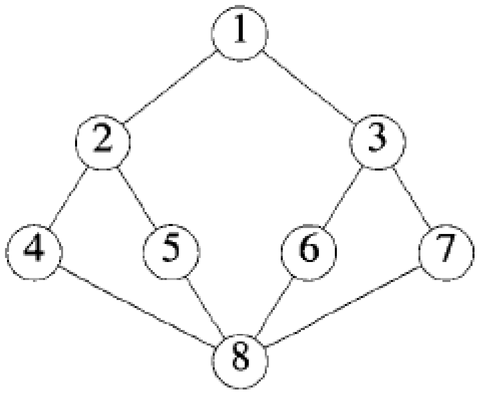
- ✓ Depth First Search
- ✓ Breadth First Search

Depth First Search

Depth first search of an undirected graph proceeds as follows. The start vertex v is visited. Next an unvisited vertex w adjacent to v is selected and a depth first search from w 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 one.

Graph G is shown below:



```
Algorithm DFS(v)
// Given an undirected (directed) graph G = (V, E) with
// n vertices and an array visited[ ] initially set
// to zero, this algorithm visits all vertices
// reachable from v. G and visited[ ] are global.
{
    visited[v] := 1;
    for each vertex w adjacent from v do
    {
        if (visited[w] = 0) then DFS(w);
    }
}
```

If a depth first search is initiated from vertex v1, then the vertices of G are visited in the order: v1, v2, v4, v8, v5, v6, v3, v7.

Breadth First Search

Starting at vertex v and marking it as visited, breadth first search differs from depth first search in that all unvisited vertices adjacent to v are visited next. Then unvisited vertices adjacent to these vertices are visited and so on.

```
Algorithm BFS(v)
// A breadth first search of G is carried out beginning
// at vertex v. For any node i, visited[i] = 1 if i has
// already been visited. The graph G and array visited[ ]
// are global; visited[ ] is initialized to zero.
{
    u := v; // q is a queue of unexplored vertices.
    visited[v] := 1;
    repeat
    {
        for all vertices w adjacent from u do
        {
            if (visited[w] = 0) then
            {
                Add w to q; // w is unexplored.
                visited[w] := 1;
            }
        }
        if q is empty then return; // No unexplored vertex.
        Delete u from q; // Get first unexplored vertex.
    } until(false);
}
```

A breadth first search beginning at vertex v1 of the graph G would first visit v1 and then v2 and v3. Next vertices v4, v5, v6 and v7 will be visited and finally v8.

Finally the order is: v1, v2, v3, v4, v5, v6, v7 and v8.

Connected Components

If G is an undirected graph, then one can determine whether or not it is connected by simply making a call to either DFS or BFS and then determining if there is any unvisited vertex.

A connected component or simply component of an undirected graph is a subgraph in which each pair of nodes is connected with each other via a path.

A set of nodes forms a connected component in an undirected graph if any node from the set of nodes can reach any other node by traversing edges. The main point here is reachability.

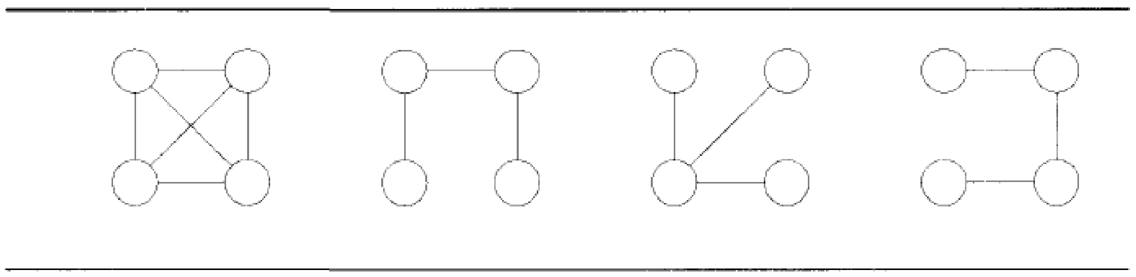
In connected components, all the nodes are always reachable from each other.

<u>Spanning Trees and Minimum Cost Spanning Trees</u>

Any tree consisting solely of edges in G and including all vertices in G is called a spanning tree.

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

A graph and its spanning trees are shown below:



In any practical situation, however, the edges will have weights assigned to them. These weights might represent the cost of construction, the length of the link, etc.

Given such a weighted graph one would then wish to select for construction a set of communication links that would connect all the cities and have *minimum total cost* or be of minimum total length.

In either case the links selected will have to form a tree.

We are, therefore, interested in finding a spanning tree of *G* with minimum cost. The cost of a spanning tree is the sum of the costs of the edges in that tree.

One approach to determining a minimum cost spanning tree of a graph has been given by Kruskal. It is shown below:

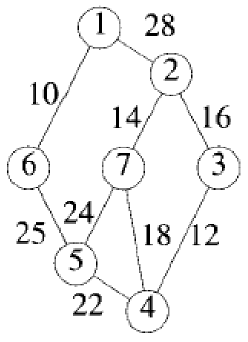
In this approach a minimum cost spanning tree, *T*, is built edge by edge.

Edges are considered for inclusion in *T* in nondecreasing order of their costs. An edge is included in *T* if it does not form a cycle with the edges already in *T*.

Since *G* is connected and has *n* > 0 vertices, exactly *n* - 1 edges will be selected for inclusion in *T*.

```
t := ∅;
while ((t has less than n − 1 edges) and (E ≠ ∅)) do
{
    Choose an edge (v, w) from E of lowest cost;
    Delete (v, w) from E;
    if (v, w) does not create a cycle in t then add (v, w) to t;
    else discard (v, w);
}
```

Example:



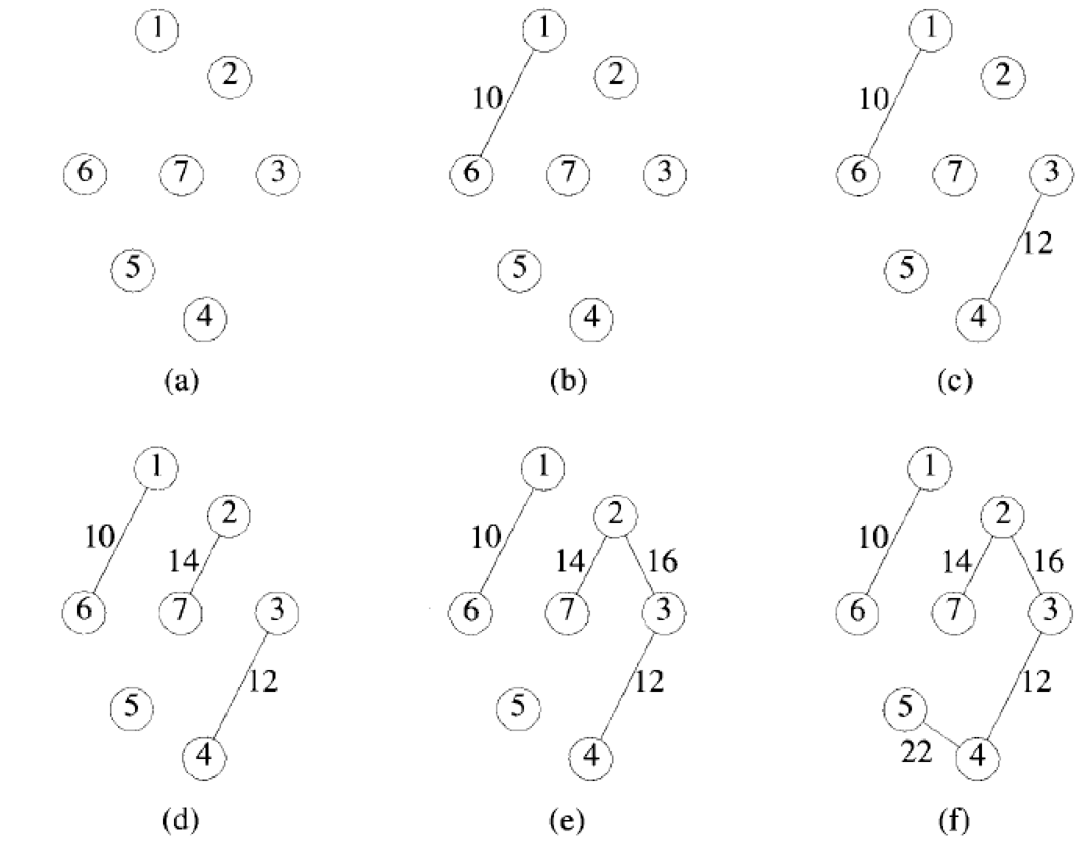
(a)

Initially, E is the set of all edges in G. The only functions we wish to perform on this set are:

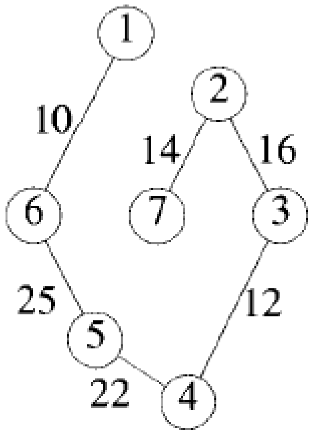
(i) determining an edge with minimum cost

(ii) deleting that edge

Both these functions can be performed efficiently if the edges in E are maintained as a sorted sequential list. The stages of the Kruskal's algorithm:



The resultant spanning tree is shown below:



AOV Network:

A directed graph G in which the vertices represent tasks or activities and the edges represent precedence relations between tasks is an activity on vertex network or AOV-network.

Transitive Closure

Transitive Closure it the reachability matrix to reach from vertex u to vertex v of a graph. One graph is given, we have to find a vertex v which is reachable from another vertex u, for all vertex pairs (u, v).

Topological Sort

This linear ordering will have the property that if i is a predecessor of j in the network then i precedes j in the linear ordering. A linear ordering with this property is called a topological order.

Topological sort gives a linear ordering of vertices in a *directed acyclic graph* such that, for every directed edge a -> b, vertex 'a' comes before vertex 'b'.

Critical Path

The critical path is the longest path of the network diagram. The activities in the critical path have an effect on the deadline of the project. If an activity of this path is delayed, the project will be delayed.

SHORTEST PATHS

Graphs may be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway.

The edges may then be assigned weights which might be either the distance between the two cities connected by the edge or the average time to drive along that section of highway.

A motorist wishing to drive from city A to city B would be interested in answers to the following questions:

(i) Is there a path from A to B?

(ii) If there is more than one path from A to B, which is the shortest path?

The length of a path is now defined to be the sum of the weights of the edges on that path rather than the number of edges.

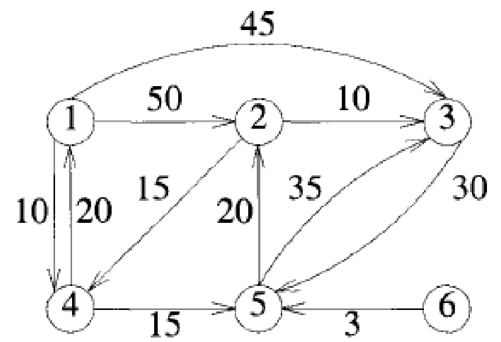
The starting vertex of the path will be referred to as the source and the last vertex the destination. The graphs will be digraphs to allow for one way streets. Assume that all weights are positive.

Single Source All Destinations

In this problem we are given a directed graph *G* = (*V,E*), a weighting function *w*(*e*) for the edges of *G* and a source vertex *vo*. The problem is to determine the shortest paths from *vo* to all the remaining vertices of *G*. It is assumed that all the weights are positive.

```
Algorithm ShortestPaths($v, cost, dist, n$)
// $dist[j]$, $1 \le j \le n$, is set to the length of the shortest
// path from vertex $v$ to vertex $j$ in a digraph $G$ with $n$
// vertices. $dist[v]$ is set to zero. $G$ is represented by its
// cost adjacency matrix $cost[1:n, 1:n]$.
{
    for $i := 1$ to $n$ do
    { // Initialize $S$.
        $S[i]$ := false; $dist[i] := cost[v, i]$;
    }
    $S[v]$ := true; $dist[v] := 0.0$; // Put $v$ in $S$.
    for $num := 2$ to $n - 1$ do
    {
        // Determine $n - 1$ paths from $v$.
        Choose $u$ from among those vertices not
        in $S$ such that $dist[u]$ is minimum;
        $S[u]$ := true; // Put $u$ in $S$.
        for (each $w$ adjacent to $u$ with $S[w]$ = false) do
            // Update distances.
            if ($dist[w] > dist[u] + cost[u, w]$)) then
                    $dist[w] := dist[u] + cost[u, w]$;
    }
}
```

Example:



(a) Graph

	Path	*Length*
1)	1, 4	10
2)	1, 4, 5	25
3)	1, 4, 5, 2	45
4)	1, 3	45

(b) Shortest paths from 1