GRAPHS

Non-linear data structures are used to represent the data containing a network or hierarchical relationship among the elements. Graphs are one of the most important nonlinear data structures. In non-linear data structures, every data element may have more than one predecessor as well as successor. Elements do not form any particular linear sequence.

GRAPH ABSTRACT DATA TYPE:

Graphs as non-linear data structures represent the relationship among data elements, having more than one predecessor and/or successor. A graph *G* is a collection of nodes (*vertices*) and arcs joining a pair of the nodes (*edges*). Edges between two vertices represent the relationship between them. For finite graphs, *V* and *E* are finite. We can denote the graph as $G = (V, E)$.

Let us define the graph ADT. We need to specify both sets of vertices and edges. Basic operations include creating a graph, inserting and deleting a vertex, inserting and deleting an edge, traversing a graph, and a few others.

A graph is a set of vertices and edges $\{V, E\}$ and can be declared as follows:

Create, insert, delete, insert, and delete, is_empty.

Create:

The create() function is used to create an empty graph. An empty graph has both *V* and *E* as null sets. The empty graph has the total number of vertices and edges as zero. However, while implementing, we should have *V* as a non-empty set and *E* as an empty set as the mathematical notation normally requires the set of vertices to be non-empty.

Insert Vertex:

The insert vertex operation inserts a new vertex into a graph and returns the modified graph. When the vertex is added, it is isolated as it is not connected to any of the vertices in the graph through an edge. If the added vertex is related with one (or more) vertices in the graph, then the respective edge(s) are to be inserted.

A graph $G(V, E)$, where $V = \{a, b, c\}$ and $E = \{(a, b), (a, c), (b, c)\}$,and the resultant graph after inserting the node ***d***. The resultant graph *G* is shown. It shows the inserted vertex with resultant $V = \{a, b, c, d\}$. We can show the adjacency relation with other vertices by adding the edge. So now, *E* would be $E = \{(a, b),(a, c), (b,c), (b, d)\}$

Delete Vertex:

The delete vertex operation deletes a vertex and all the incident edges on that vertex and returns the modified graph. Fig Shows a graph $G(V, E)$ where $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, c),(b, c),(b, d)\}$, and the resultant graph after deleting the node *c* is shown with $V = \{a, b, d\}$ and $E = \{(a, b), (b, d)\}$.

Insert Edge:

The insert edge operation adds an edge incident between two vertices. In an undirected graph, for adding an edge, the two vertices *u* and *v* are to be specified, and for a directed graph specify, the start vertex and the end vertex. Fig shows a graph *G*(*V*, *E*) where *V*={*a*, *b*, *c*, *d*} and *E* ={(*a*, *b*), (*a*, *c*), (*b*, *c*), (*b*, *d*)} and the resultant graph after inserting the edge(*c*, *d*) is *V* ={*a*, *b*, *c*, *d*} and *E* ={(*a*, *b*), (*a*, *c*), (*b*, *c*), (*b*, *d*), (*c*, *d*)}.

Delete Edge:

The delete edge operation removes one edge from the graph. Let the graph *G* be *G*(*V*, *E*). Now, deleting the edge (*u, v*) from *G* deletes the edge incident between vertices *u* and *v* and keeps the incident vertices *u, v*. shows a graph *G*(*V*, *E*),where *V* ={*a*, *b*, *c*, *d*} and *E* ={(*a*, *b*),(*a*, *c*), (*b*, *c*), (*b*, *d*)}. The resultant graph after deleting the edge (*b*, *d*) is shown In fig with *V* ={*a*, *b*, *c*, *d*} and *E* ={(*a*, *b*), (*a*, *c*), (*b*, *c*)}.

Is_empty:

The is_empty operation checks whether the graph is empty and returns true if empty else returns false. An empty graph is one where the set *V* is a null set. *Graph traversal* is also known as searching through a graph. It means systematically passing through the edges and visiting the vertices of the graph. A graph search algorithm can help in listing all vertices, checking connectivity, and discovering the structure of a graph.

Representation of Graph:

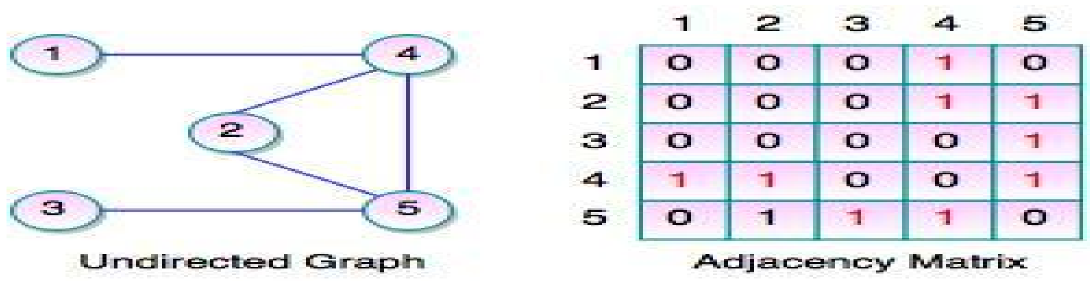
There are two standard representations of a graph given as follows:
1. Adjacency matrix (sequential representation) and
2. Adjacency list (linked representation)

Using these two representations, graphs can be realized using the adjacency matrix, adjacency list, or adjacency multilist.

Adjacency Matrix:

Adjacency matrix is a square, two-dimensional array with one row and one column for each vertex in the graph. An entry in row *i* and column *j* is 1 if there is an edge incident between vertex *i* and vertex *j*, and is 0 otherwise. If a graph is a weighted graph, then the entry 1 is replaced with the weight. It is one of the most common and simple representations of the edges of a graph. For a graph *G* =(*V*, *E*), suppose *V* ={1, 2, …, *n*}. The adjacency matrix for *G* is a two dimensional *n* X *n* Boolean matrix ***A*** and can be represented as

$$\boldsymbol{A[i][j]} = \{1 \text{ if there exists an edge } <i, j> , \quad 0 \text{ if edge } <i, j> \text{ does not exist}\}$$

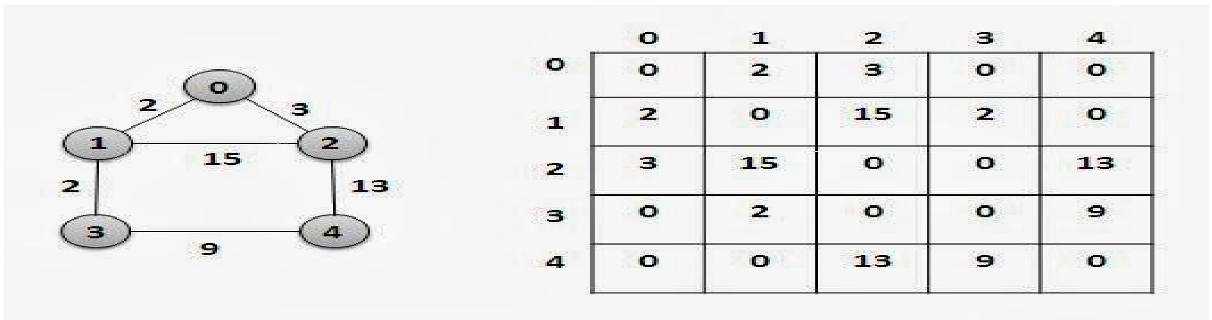


	1	2	3	4	5
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0

Fig. Adjacency Matrix Representation of Undirected Graph

For a ***weighted graph***, the matrix ***A*** is represented as

***A*[*i*][*j*]** ={weight if the edge <*i*, *j*> exists, 0 if there exists no edge <*i*, *j*>}



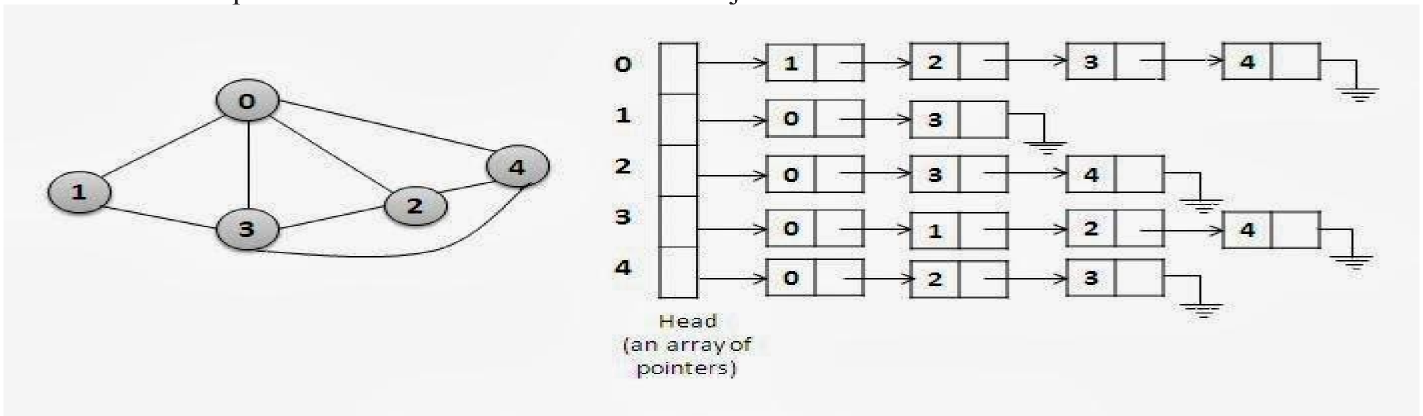
	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

Adjacency Matrix Representation of Weighted Graph

<u>Adjacency List:</u>

The *n* rows of the adjacency list are represented as *n*-linked lists, one list per vertex of the graph. The adjacency list for a vertex ***i*** is a list of all vertices adjacent to it. One way of achieving this is to go for an array of pointers, one per vertex.

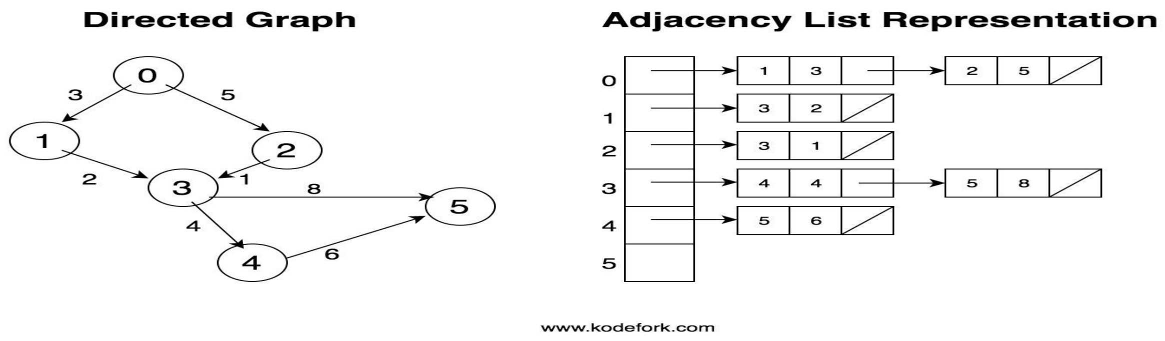
For example, a graph G by an array Head, where Head[i] is a pointer to the adjacency list of vertex i. For list each node of the list has at least two fields: vertex and link. The vertex field contains the vertex id, and the link field stores a pointer to next stores another vertex adjacent to i.



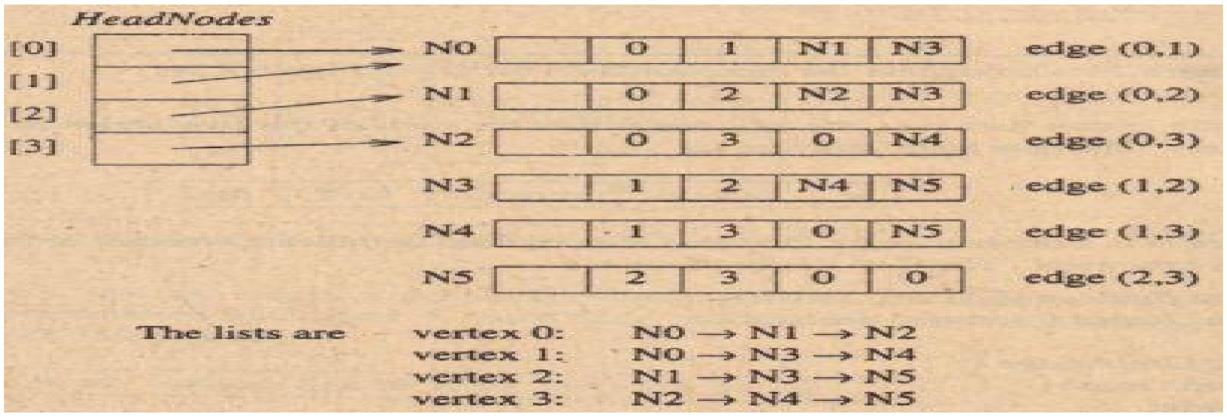
Adjacency List Representation of Graph

This graph is a directed graph. If a graph is a weighted graph, a weight field can be added in the node structure of the list.

Fig: Adjacency list of weighted graph



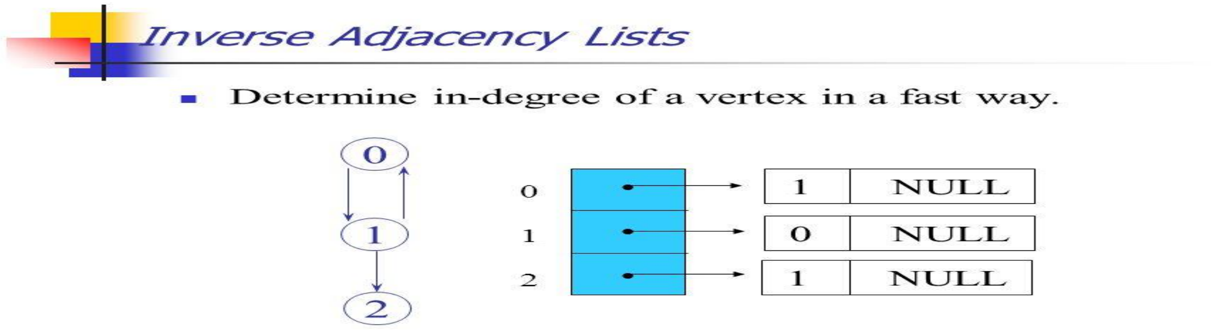
Adjacency Multilist:
For the graph *G*1 the edge connecting the vertices 1 and 2 is represented twice, in the lists of vertices 1 and 2. In applications such as minimum spanning tree computation, if we process any edge once, then it has to be marked as a processed one. If the adjacency list is maintained as multilists such that the nodes are shared among several lists.


HeadNodes
[0]
[1]
[2]
[3]

N0		0	1	N1	N3	edge (0,1)
N1		0	2	N2	N3	edge (0,2)
N2		0	3	0	N4	edge (0,3)
N3		1	2	N4	N5	edge (1,2)
N4		1	3	0	N5	edge (1,3)
N5		2	3	0	0	edge (2,3)

The lists are
vertex 0: N0 → N1 → N2
vertex 1: N0 → N3 → N4
vertex 2: N1 → N3 → N5
vertex 3: N2 → N4 → N5

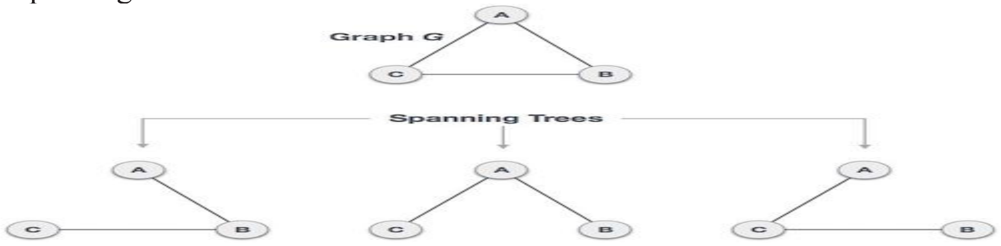
Inverse Adjacency List:
An *inverse adjacency list* is a set of lists that contains one list for each vertex. Each list contains a node per vertex adjacent to the vertex it represents. fig represents the inverse adjacency list for the graph *G.*


Inverse Adjacency Lists

- Determine in-degree of a vertex in a fast way.

SPANNING TREE:

A tree is a connected graph with no cycles. A spanning tree is a sub-graph of *G* that has all vertices of *G* and is a tree. A minimum spanning tree of a weighted graph *G* is the spanning tree of *G* whose edges sum to minimum weight.
There can be more than one minimum spanning tree for a graph. Fig shows a graph, one of its spanning trees, and a minimum spanning tree.



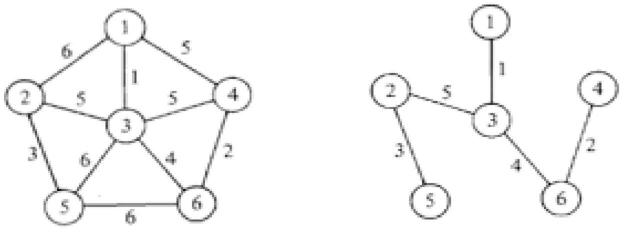


Fig. 7.4. A graph and spanning tree.

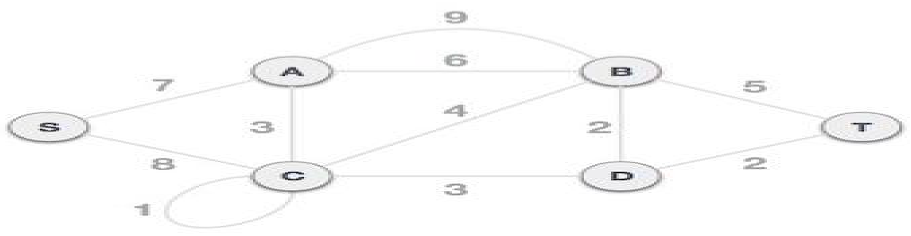
A minimum spanning tree minimizes the total length over all possible spanning trees.
These two popular methods used to compute the minimum spanning tree of a graph are
1. Prim's algorithm
2. Kruskal's algorithm

Connected Components:
An undirected graph is *connected* if there is at least one path between every pair of vertices in the graph. A *connected component* of a graph is a *maximal connected* sub-graph, that is, every vertex in a connected component is *reachable* from the vertices in the component. Consider the graph *G*1 in Fig
In this undirected graph, there is only one connected component, the graph *G*1 itself. If we delete the edges *e*4 and *e*5 from the graph *G*1, we get a graph *G*2 with two connected components: ({*V*1, *V*2, *V*3}, {*E*1, *E*2, *E*3}) and ({*V*4}, Ø). This is represented in Fig.

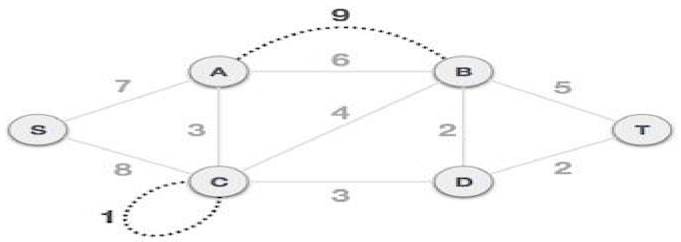
PRIM'S ALGORITHM:
All vertices of any connected graph are included in a minimum cost spanning tree of a graph *G*. Prim's algorithm starts from one vertex and grow the rest of the tree by adding one vertex at a time, by adding the associated edges.
This algorithm builds a tree by iteratively adding edges until a minimal spanning tree is obtained, that is, when all nodes are added. At each iteration, a next minimum weight edge is added that adds a new vertex to the tree, if adding that edge does not form a cycle.
Prim's algorithm looks for the shortest possible edge <u, v> such that u belongs A and v belongs to v-A.
Prim's algorithm to find minimum cost spanning tree uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

Prim's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

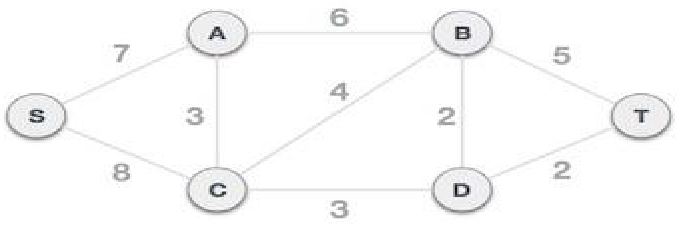
Consider an example: weighted graph G



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

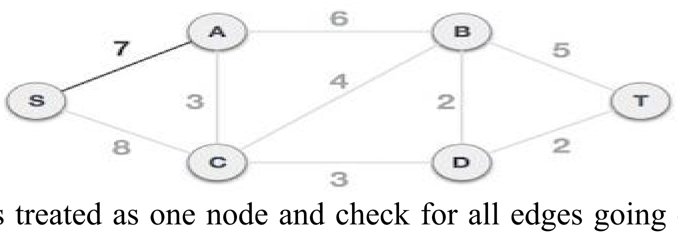


Step 2 - Choose any arbitrary node as root node

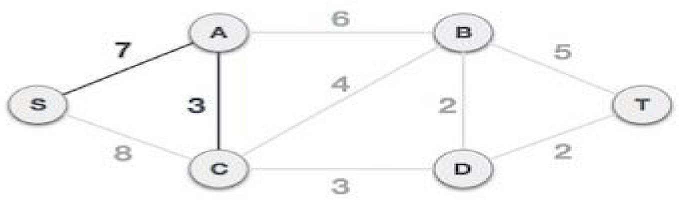
Let choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. In the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

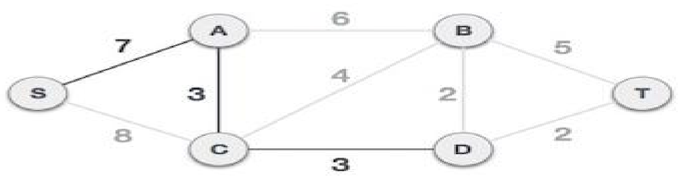
After choosing the root node **S**, see that S,A and S,C are two edges with weight 7 and 8, respectively. Choose the edge S,A as it is lesser than the other.



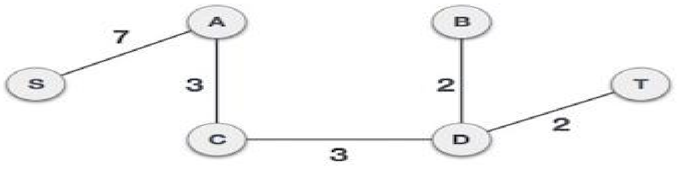
Now, the tree S-7-A is treated as one node and check for all edges going out from it. Select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now treat it as a node and check all the edges again. However, choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, add either one. But the next step will again yield edge 2 as the least cost. Hence, showing a spanning tree with both edges included.

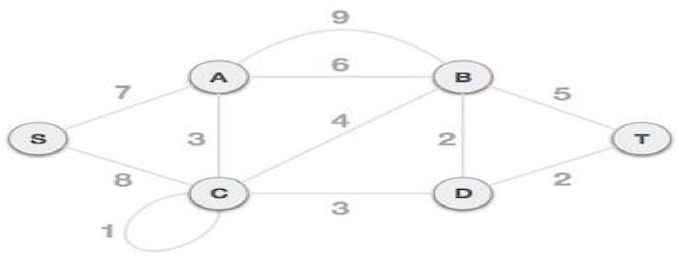


As all the vertices are added, the algorithm ends. The resultant minimum spanning tree with total weight is 7+3+3+2+2=17.

KRUSKAL’S ALGORITHM:

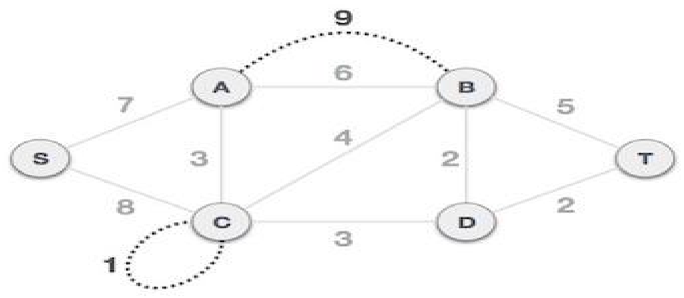
kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm let us consider the following example −

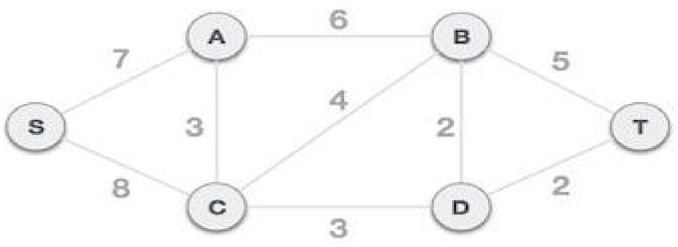


Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.



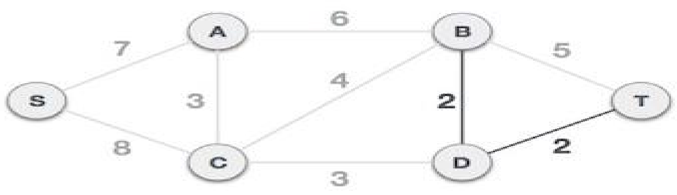
Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

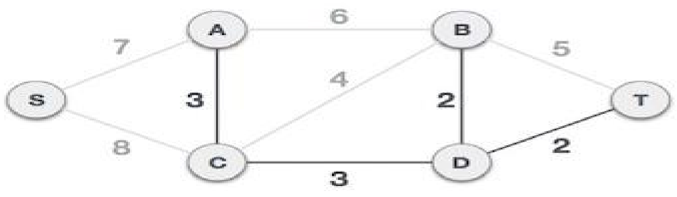
Step 3 - Add the edge which has the least weightage

Now start adding edges to the graph beginning from the one which has the least weight. Throughout, keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then consider not to include the edge in the graph.

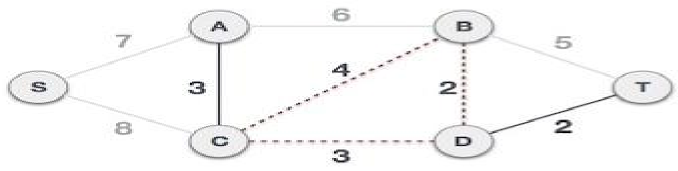


The least cost is 2 and edges involved are B, D and D,T. Add them. Adding them does not violate spanning tree properties, so continue to next edge selection.

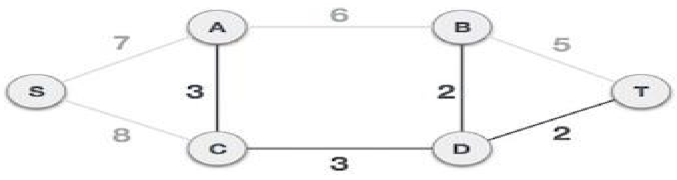
Next cost is 3, and associated edges are A,C and C,D. Add them again −



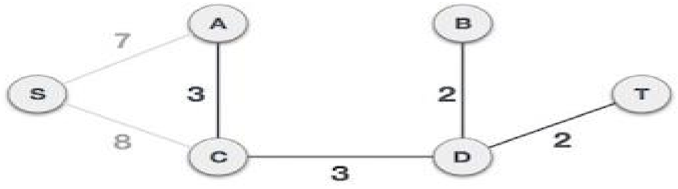
Next cost in the table is 4, and observes that adding it will create a circuit in the graph. –



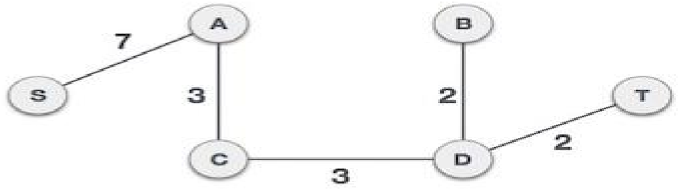
Ignore it. In the process ignore/avoid all edges that create a circuit.



Observe that edges with cost 5 and 6 also create circuits. Ignore them and move on.



Now left with only one node to be added. Between the two least cost edges available 7 and 8, add the edge with cost 7.



By adding edge S,A include all the nodes of the graph and now have minimum cost spanning tree.