SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI

18CSC201J

DATA STRUCTURES AND ALGORITHMS

Unit- V

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

- GRAPH TERMINOLOGY
- GRAPH TRAVERSAL

- A graph - an abstract data structure that is used to implement the graph concept from mathematics.
- A collection of vertices (also called nodes) and edges that connect these vertices.
- A graph is often viewed as a generalization of the tree structure, where instead of a having a purely parent-to-child relationship between tree nodes.
- Any kind of complex relationships between the nodes can be represented.

Why graphs are useful?

- Graphs are widely used to model any situation where entities or things are related to each other in pairs; for example, the following information can be represented by graphs:
- *Family trees* in which the member nodes have an edge from parent to each of their children.
- *Transportation networks* in which nodes are airports, intersections, ports, etc. The edges can be airline flights, one-way roads, shipping routes, etc.

Definition

- A graph G is defined as an ordered set (V, E), where V(G) represent the set of vertices and E(G) represents the edges that connect the vertices.
- The figure given shows a graph with V(G) = { A, B, C, D and E} and E(G) = { (A, B), (B, C), (A, D), (B, D), (D, E), (C, E) }. Note that there are 5 vertices or nodes and 6 edges in the graph.

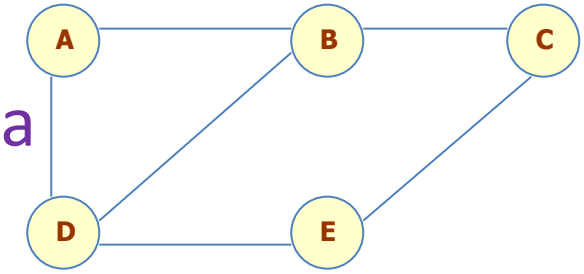


Fig:a

A graph can be directed (Fig a)or undirected (Fig b).

Fig:a

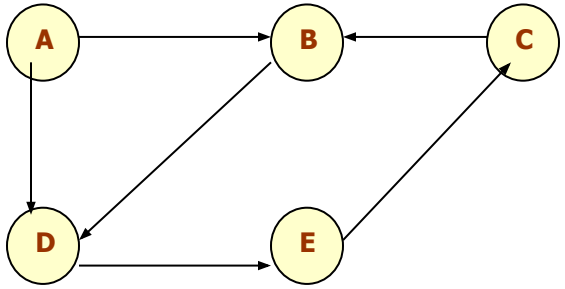
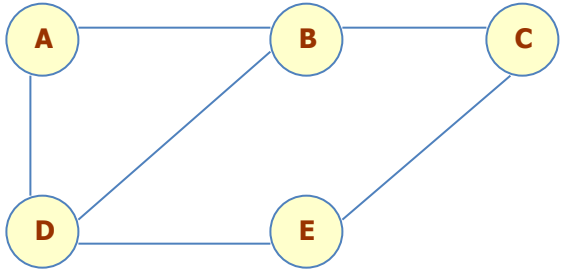


Fig:b



Graph Terminology:

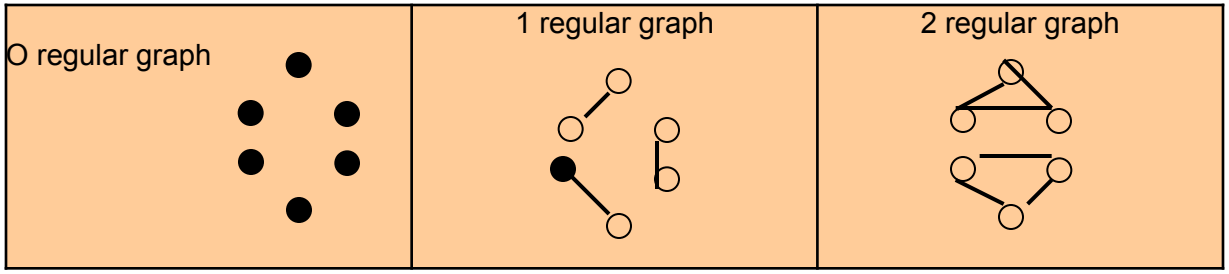
Adjacent Nodes or Neighbors:

For every edge, e = (u, v) that connects nodes u and v; the nodes u and v are the end-points and are said to be the adjacent nodes or neighbors.

Degree of a node:

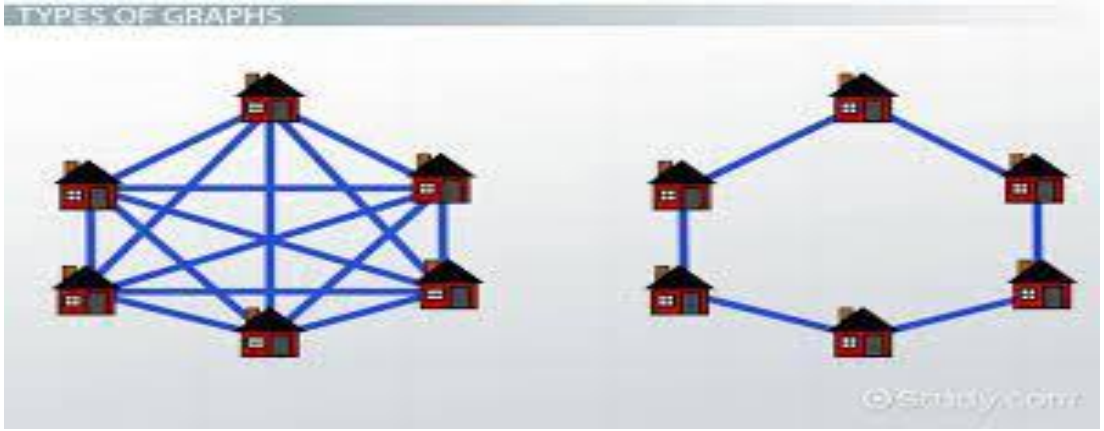
Degree of a node u, deg(u), is the total number of edges containing the node u. If deg(u) = 0, it means that u does not belong to any edge and such a node is known as an isolated node.

Regular graph: Regular graph is a graph where each vertex has the same number of neighbors. That is every node has the same degree. A regular graph with vertices of degree k is called a k-regular graph or regular graph of degree k.



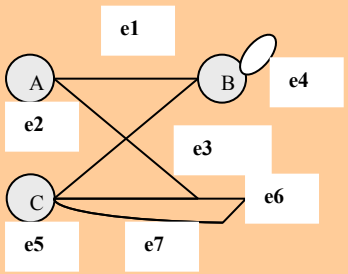
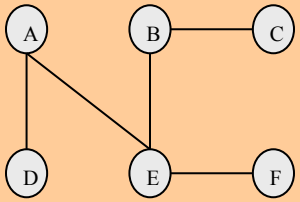
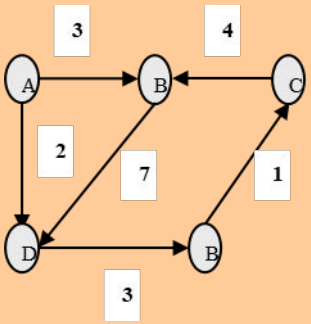
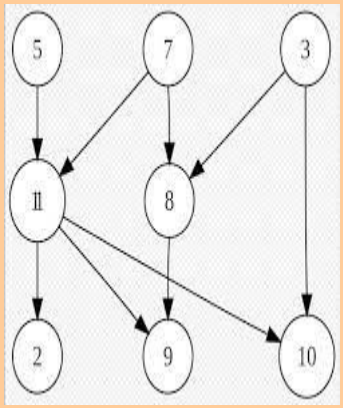
- *Path:* A path P, written as P = {v0, v1, v2,….., vn), of length n from a node u to v is defined as a sequence of (n+1) nodes. Here, u = v0, v = vn and vi-1 is adjacent to vi for i = 1, 2, 3, …, n.
- *Closed path:* A path P is known as a closed path if the edge has the same end-points. That is, if v0 = vn. (Cycle)
- *Simple path:* A path P is known as a simple path if all the nodes in the path are distinct with an exception that v0 may be equal to vn. If v0 = vn, then the path is called a closed simple path.
- *Cycle:* A closed simple path with length 3 or more is known as a cycle. A cycle of length *k* is called a *k* – cycle.

- *Connected graph:* A graph in which there exists a path between any two of its nodes is called a connected graph. That is to say that there are no isolated nodes in a connected graph. A connected graph that does not have any cycle is called a tree.
- *Complete graph:* A graph G is said to be a complete, if all its nodes are fully connected, that is, there is a path from one node to every other node in the graph. A complete graph has n(n-1)/2 edges, where n is the number of nodes in G.



- *Labeled graph or weighted graph:* A graph is said to be labeled if every edge in the graph is assigned some data. In a weighted graph, the edges of the graph are assigned some weight or length. Weight of the edge, denoted by w(e) is a positive value which indicates the cost of traversing the edge.
- *Multiple edges:* Distinct edges which connect the same end points are called multiple edges. That is, e = {u, v) and e' = (u, v) are known as multiple edges of G.
- *Loop:* An edge that has identical end-points is called a loop. That is, e = (u, u).
- *Multi- graph:* A graph with multiple edges and/or a loop is called a multi-graph.
- *Size of the graph:* The size of a graph is the total number of edges in it.

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

(a) Multi-graph	(b) Tree	(c) Weighted Graph	Simple Directed Graph
			

Directed Graph:

- A directed graph G, also known as a digraph, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G. For an edge (u, v)-
- The edge begins at u and terminates at v
- U is known as the origin or initial point of e. Correspondingly, v is known as the destination or terminal point of e
- U is the predecessor of v. Correspondingly, v is the successor of u nodes u and v are adjacent to each other.

Terminologies in Directed graph:

- *Out-degree of a node:* The out degree of a node u, written as outdeg(u), is the number of edges that originate at u.
- *In-degree of a node*: The in degree of a node u, written as indeg(u), is the number of edges that terminate at u.
- *Degree of a node:* Degree of a node written as deg(u) is equal to the sum of in-degree and out-degree of that node. Therefore, deg(u) = indeg(u) + outdeg(u)
- *Source:* A node u is known as a source if it has a positive out-degree but an in-degree = 0.
- *Sink:* A node u is known as a sink if it has a positive in degree but a zero out-degree.
- *Reachability:* A node v is said to be reachable from node u, if and only if there exists a (directed) path from node u to node v.

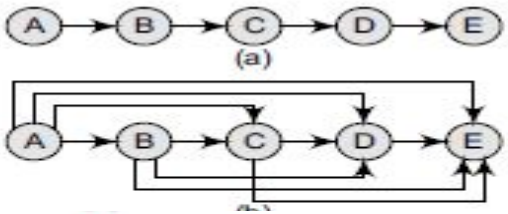
- *Strongly connected directed graph*: A digraph is said to be strongly connected if and only if there exists a path from every pair of nodes in G. That is, if there is a path from node u to v, then there must be a path from node v to u.
- *Unilaterally connected graph:* A digraph is said to be unilaterally connected if there exists a path from any pair of nodes u, v in G such that there is a path from u to v or a path from v to u but not both.
- *Parallel/Multiple edges:* Distinct edges which connect the same end points are called multiple edges. That is, e = {u, v) and e' = (u, v) are known as multiple edges of G.
- *Simple directed graph:* A directed graph G is said to be a simple directed graph if and only if it has no parallel edges. However, a simple directed graph may contain cycle with an exception that it cannot have more than one loop at a given node

Transitive Closure of a Directed Graph:

- It is same as the adjacency list in graph terminology.
- It is stored as a matrix T.

Definition

For a directed graph G = (V,E), where V is the set of vertices and E is the set of edges, the transitive closure of G is a graph G* = (V,E*). In G*, for every vertex pair v, w in V there is an edge (v, w) in E* if and only if there is a valid path from v to w in G.

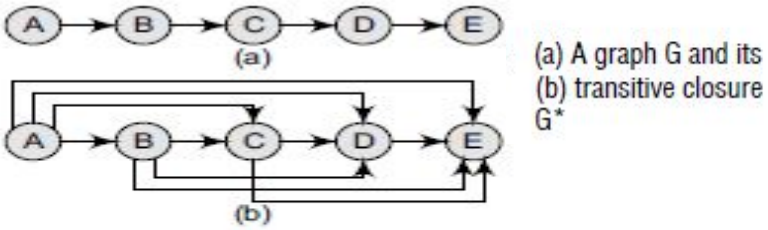


(a) A graph G and its
(b) transitive closure
G*

- A binary relation indicates only whether the node A is connected to node B, whether node B is connected to node C, etc.
- But once the transitive closure is constructed as shown in Fig.
- we can easily determine in O(1) time whether node E is reachable from node A or not.

Definition

For a directed graph G = (V,E), where V is the set of vertices and E is the set of edges, the transitive closure of G is a graph G* = (V,E*). In G*, for every vertex pair v, w in V there is an edge (v, w) in E* if and only if there is a valid path from v to w in G.



(a) A graph G and its (b) transitive closure G*

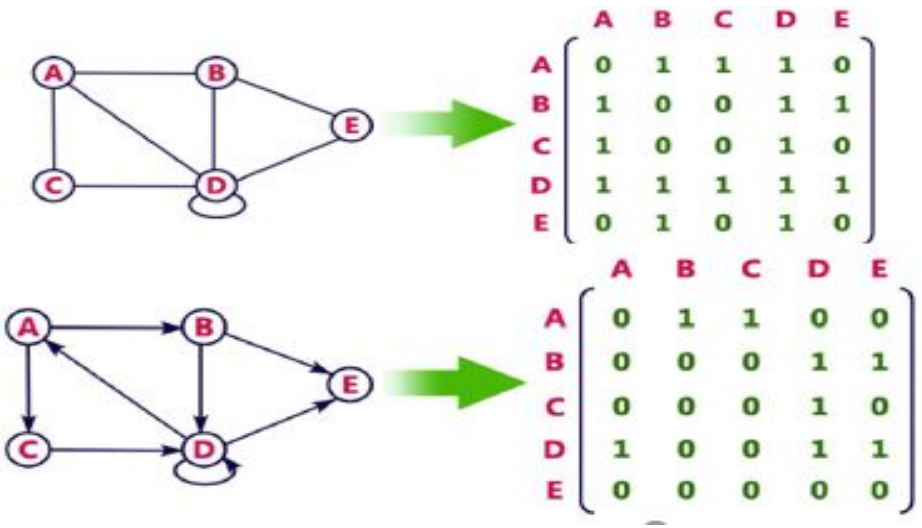
- A binary relation indicates only whether the node A is connected to node B, whether node B is connected to node C, etc. But once the transitive closure is constructed as shown in Fig. we can easily determine in O(1) time whether node E is reachable from node A or not.

Graph Representation:

- There are three common ways of storing graphs in the computer's memory.
- They are:
  - *Sequential representation* by using an adjacency matrix.
  - *Linked representation* by using an adjacency list that stores the neighbors of a node using a linked list.

Adjacency Matrix Representation:

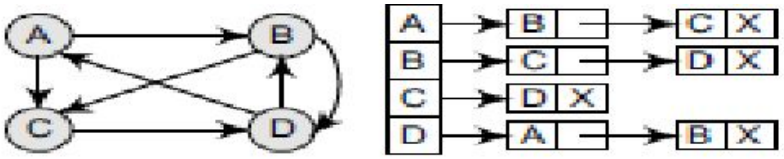
- An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting them.
- In this representation, graph can be represented using a matrix of size V*V.
- No matter how few edges the graph has, the matrix has O(n^2) in memory.



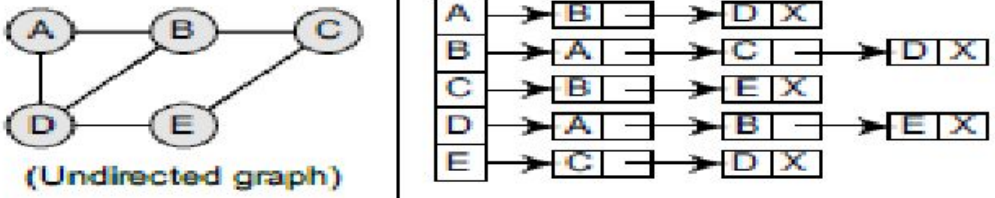
Adjacency List Representation:

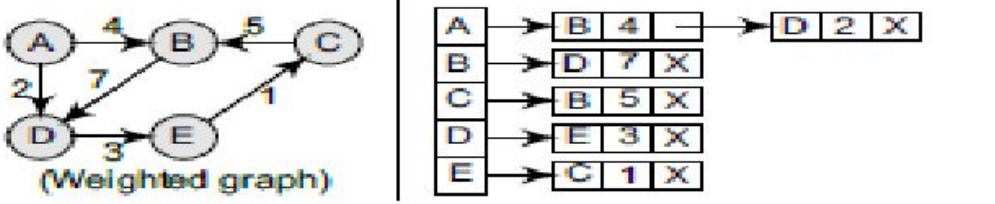
- An adjacency list is another way in which graphs can be represented in the computer's memory.
- This structure consists of a list of all nodes in G. Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.
- The key advantages of using an adjacency list are:
  - It is easy to follow and clearly shows the adjacent nodes of a particular node.
  - It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.
  - Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.

Adjacency List Representation:



Graph G and its adjacency list





Adjacency list for an undirected graph and a weighted graph

Breadth-First Search Algorithm:

- It is a graph search algorithm that begins at the root node and explores all the neighboring nodes.
- Then for each of those nearest nodes, the algorithm explores their unexplored neighbor nodes, and so on, until it finds the goal.
- This technique is used for searching a vertex in a graph.
- It produces a spanning tree as a final result.
- Spanning tree is a graph without any loops.
- Here we use Queue data structure with maximum size of total number of vertices in the graph.

Breadth-First Search Algorithm:

STEPS:

Step 1 - Define a Queue of size total number of vertices in the graph.

Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.

Step 3 - Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.

Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

Step 5 - Repeat steps 3 and 4 until queue becomes empty.

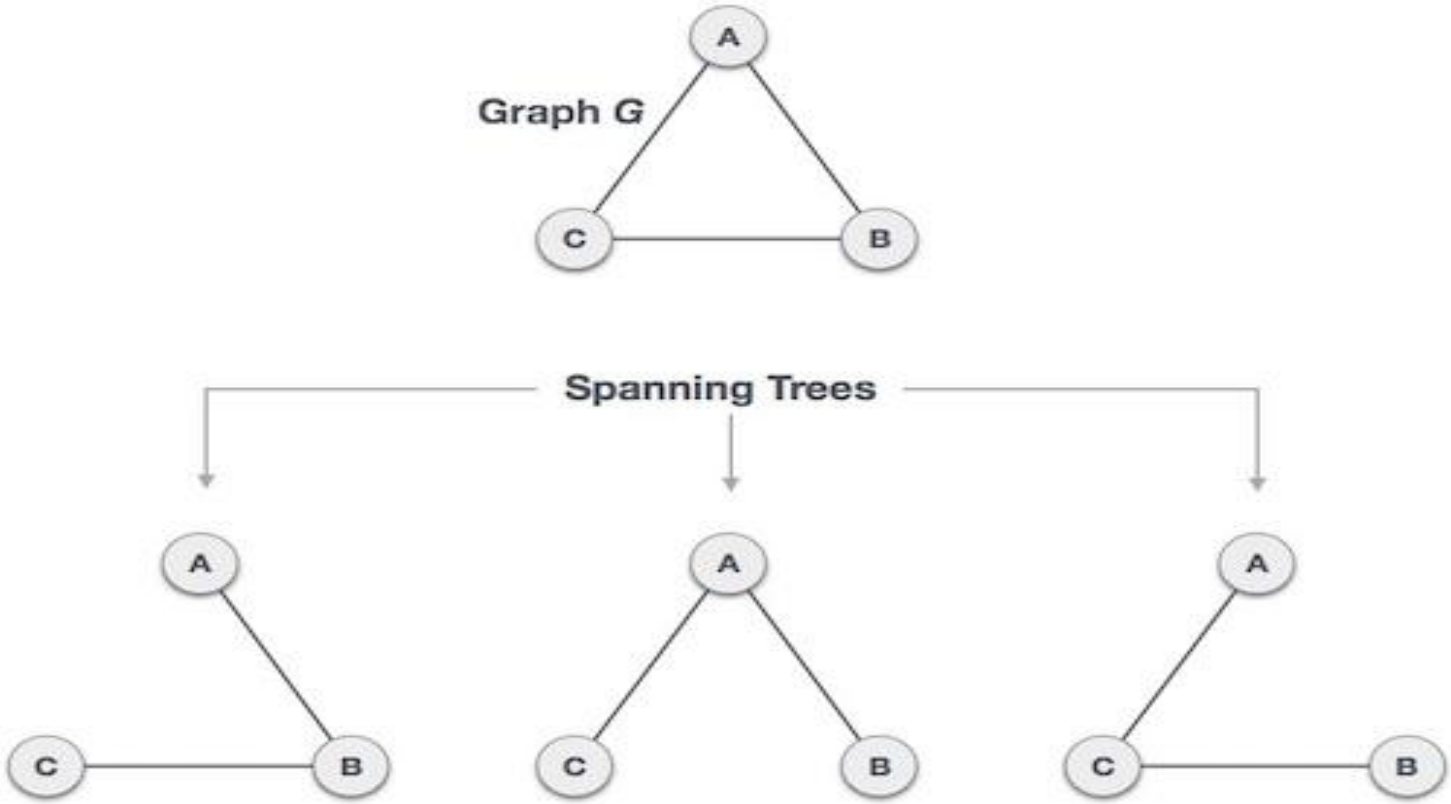
Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

Breadth-First Search Algorithm:

```
Step 1: SET STATUS = 1 (ready state)
        for each node in G
Step 2: Enqueue the starting node A
        and set its STATUS = 2
        (waiting state)
Step 3: Repeat Steps 4 and 5 until
        QUEUE is empty
Step 4: Dequeue a node N. Process it
        and set its STATUS = 3
        (processed state).
Step 5: Enqueue all the neighbours of
        N that are in the ready state
        (whose STATUS = 1) and set
        their STATUS = 2
        (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

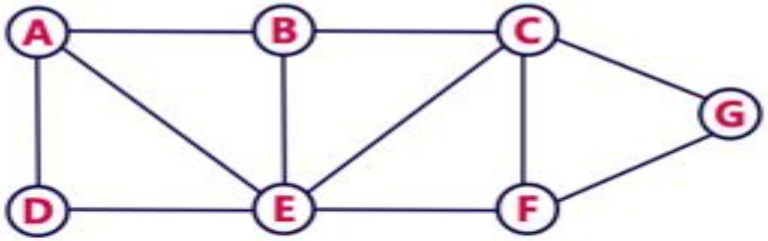
Spanning Tree

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..



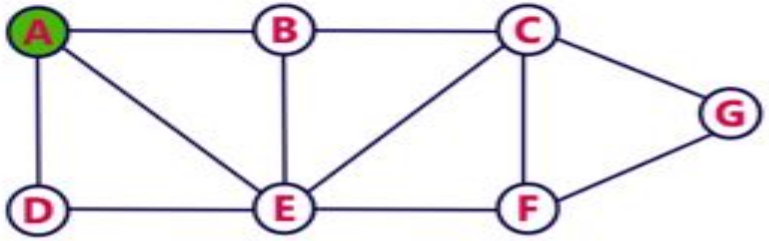
Breadth-First Search Algorithm:

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

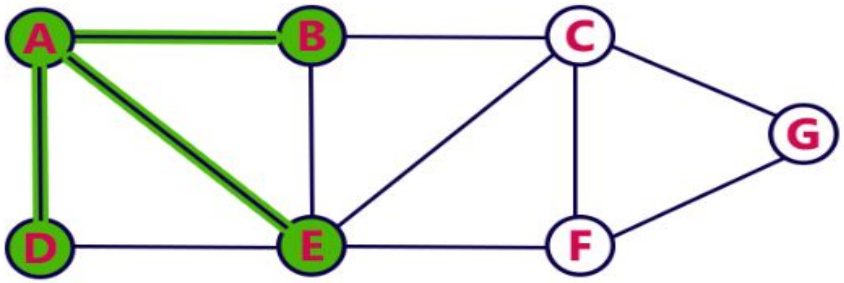


Queue

Breadth-First Search Algorithm:

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

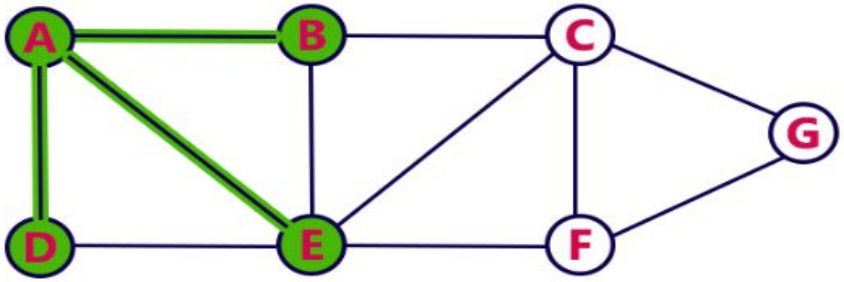


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



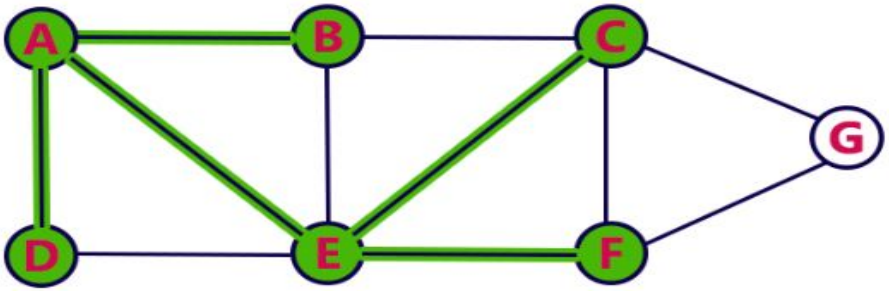
Queue

		E	B			

Breadth-First Search Algorithm:

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

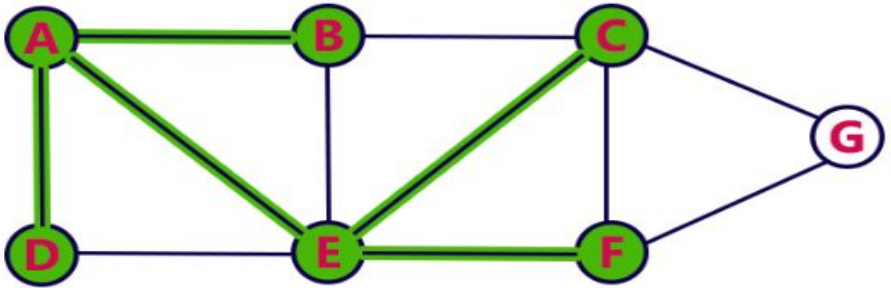


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



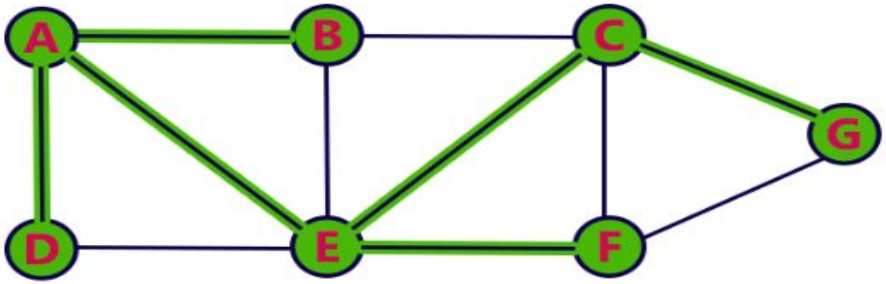
Queue

				C	F	

Breadth-First Search Algorithm:

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

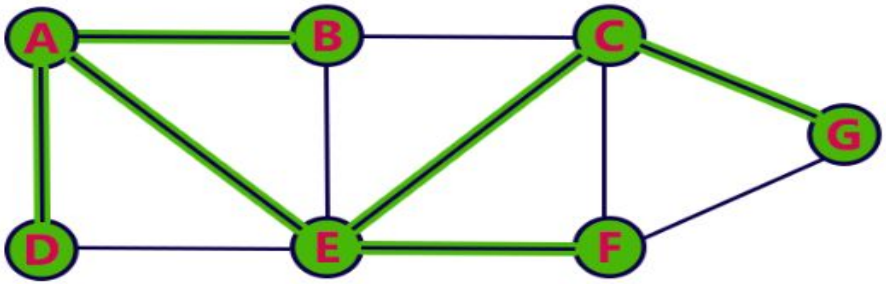


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.



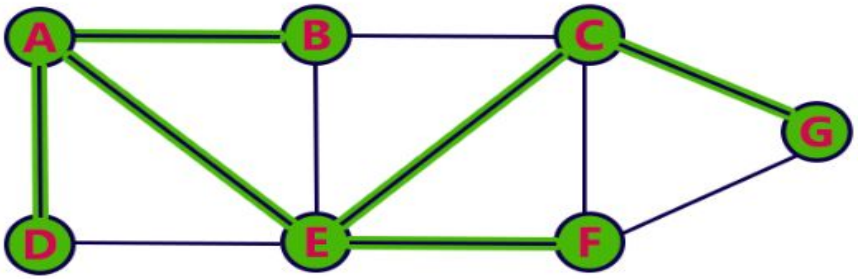
Queue

						G

Breadth-First Search Algorithm:

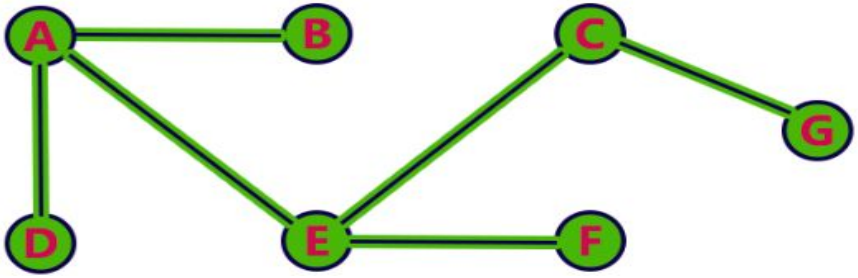
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Features of Breadth-First Search Algorithm:

- ***Space complexity***
  - In the breadth-first search algorithm, all the nodes at a particular level must be saved until their child nodes in the next level have been generated.
  - The space complexity is therefore proportional to the number of nodes at the deepest level of the graph. Given a graph with branching factor b (number of children at each node) and depth d, the asymptotic space complexity is the number of nodes at the deepest level O(bd).
- ***Time complexity***
  - In the worst case, breadth-first search has to traverse through all paths to all possible nodes, thus the time complexity of this algorithm asymptotically approaches O(bd).
  - However, the time complexity can also be expressed as O(| E | + | V |), since every vertex and every edge will be explored in the worst case.

- ***Completeness***
  - Breadth-first search is said to be a complete algorithm because if there is a solution, breadth-first search will find it regardless of the kind of graph.
  - But in case of an infinite graph where there is no possible solution, it will diverge.
- ***Optimality***
  - Breadth-first search is optimal for a graph that has edges of equal length, since it always returns the result with the fewest edges between the start node and the goal node.
  - But generally, in real-world applications, we have weighted graphs that have costs associated with each edge, so the goal next to the start does not have to be the cheapest goal available.

Applications of Breadth-First Search Algorithm:

- Breadth-first search can be used to solve many problems such as:
- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph.

Depth-first Search Algorithm:

- The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.
- In other words, depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbor of A, then a neighbor of neighbor of A, and so on.

Depth-first Search Algorithm:

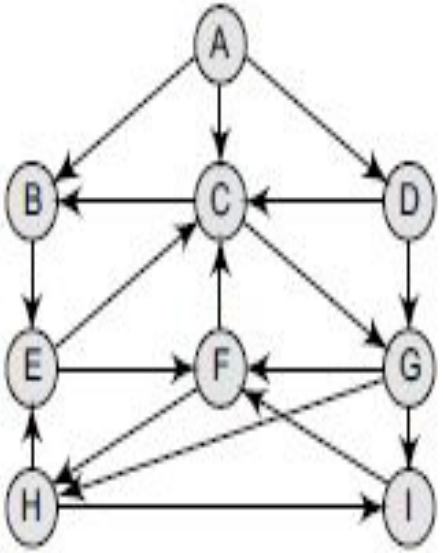
- During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.
- This technique is used for searching a vertex in a graph.
- It produces a spanning tree as a final result.
- Spanning tree is a graph without any loops.
- Here we use Stack data structure with maximum size of total number of vertices in the graph.

Depth-first Search Algorithm:

```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set
        its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4:   Pop the top node N. Process it and set its
          STATUS = 3 (processed state)
Step 5:   Push on the stack all the neighbours of N that
          are in the ready state (whose STATUS = 1) and
          set their STATUS = 2 (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

Depth-first Search Algorithm:

Consider the graph and find the spanning tree,



Adjacency lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Depth-first Search Algorithm:

(a) Push H onto the stack.

STACK: H

(b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H onto the stack that are in the ready state. The STACK now becomes

PRINT: H STACK: E, I

(c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I onto the stack that are in the ready state. The STACK now becomes

PRINT: I STACK: E, F

(d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. (Note F has two neighbours, C and H. But only C will be added, as H is not in the ready state.) The STACK now becomes

PRINT: F STACK: E, C

(e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes

PRINT: C STACK: E, B, G

Depth-first Search Algorithm:

(f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: G | STACK: E, B

(g) Pop and print the top element of the STACK, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: B | STACK: E

(h) Pop and print the top element of the STACK, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The STACK now becomes empty.

PRINT: E | STACK:

Since the STACK is now empty, the depth-first search of G starting at node H is complete and the nodes which were printed are:

H, I, F, C, G, B, E

These are the nodes which are reachable from the node H.

Features of Depth-First Search Algorithm

Space complexity The space complexity of a depth-first search is lower than that of a breadthfirst search.

Time complexity The time complexity of a depth-first search is proportional to the number of vertices plus the number of edges in the graphs that are traversed. The time complexity can be given as (O(|V| + |E|)).

Completeness Depth-first search is said to be a complete algorithm. If there is a solution, depthfirst search will find it regardless of the kind of graph. But in case of an infinite graph, where there is no possible solution, it will diverge.

Applications of Depth-First Search Algorithm

Depth-first search is useful for:

- Finding a path between two specified nodes, u and v, of an unweighted graph.
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph.

Minimum Spanning Tree

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

1. Cluster Analysis
2. Handwriting recognition
3. Image segmentation

Minimum Spanning Tree

Spanning Tree

Given an undirected and connected graph G=(V,E), a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)

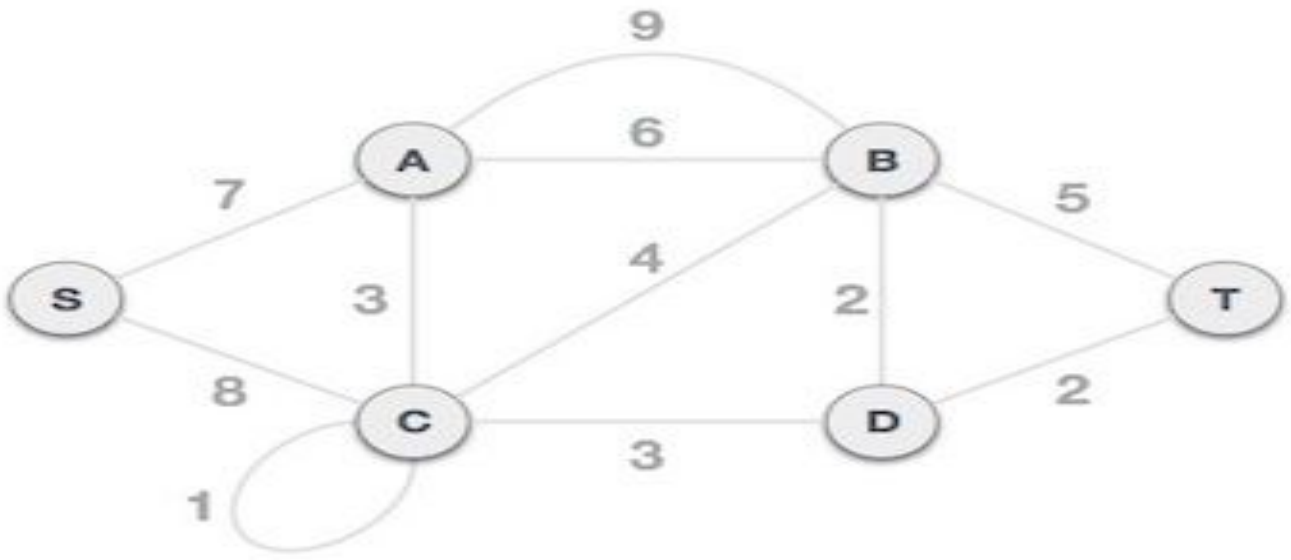
Minimum Spanning Tree

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Minimum Cost Spanning Tree

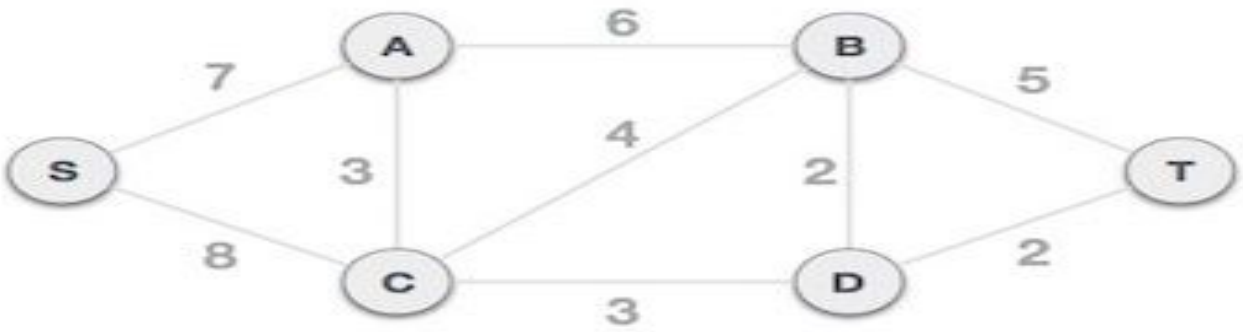
A **Minimum Spanning Tree (MST)** is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight. To derive an MST, **Prim's** algorithm or **Kruskal's** algorithm can be used.

Prims Algorithm: Prim's algorithm is a greedy approach to find the minimum spanning tree. In this algorithm, to form a MST we can start from an arbitrary vertex.



Minimum Cost Spanning Tree

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.

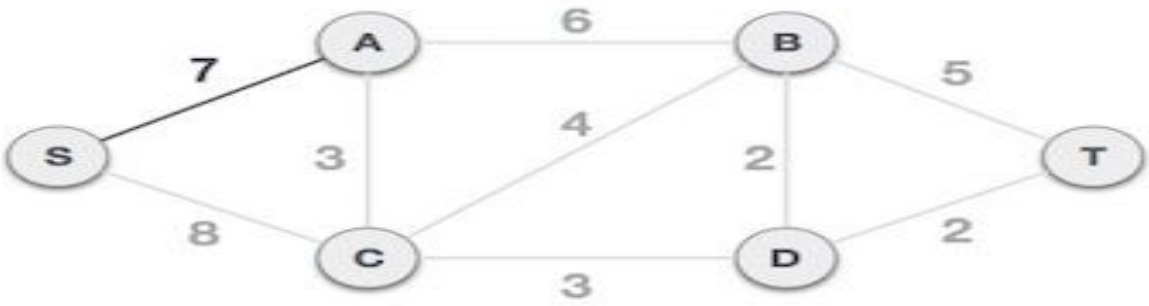


Step 2 - **Choose any arbitrary node as root node**

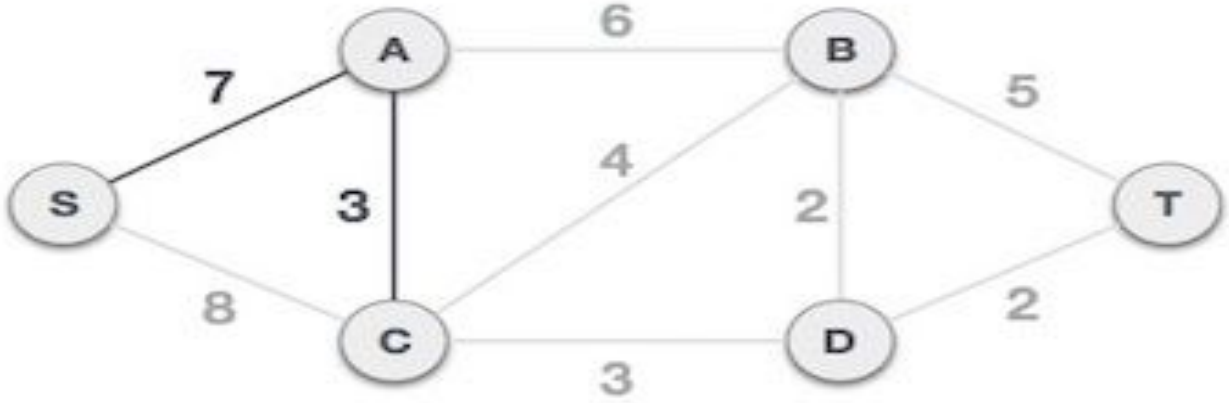
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Minimum Cost Spanning Tree

Step 3 - Check outgoing edges and select the one with less cost : After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.

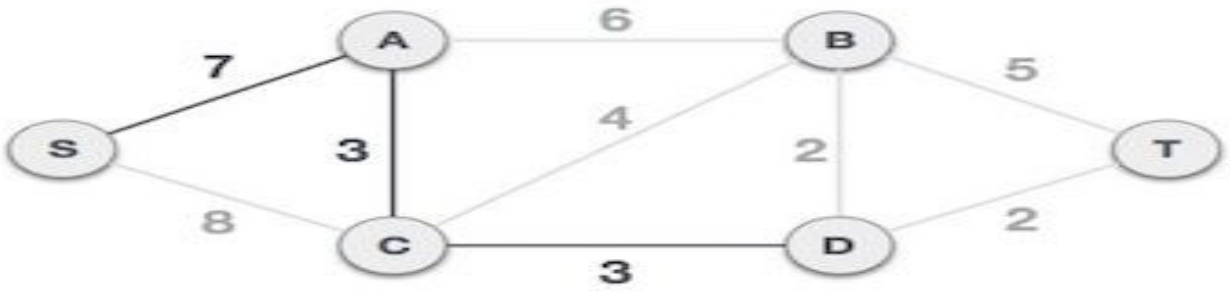


Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.

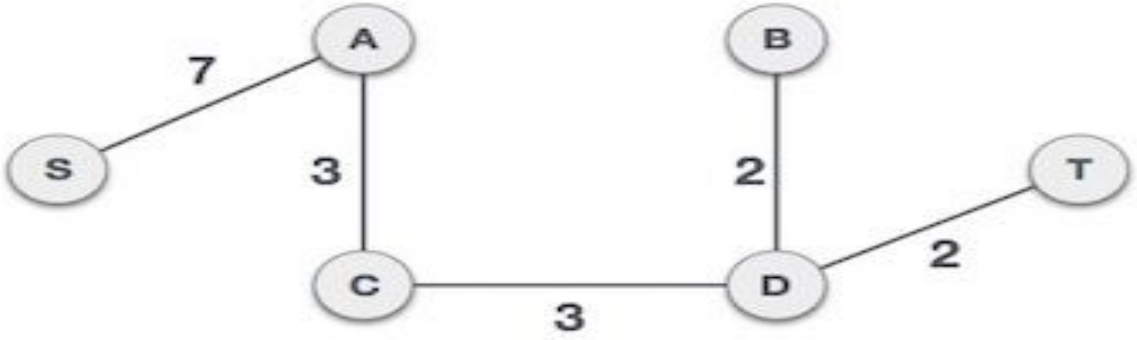


Minimum Cost Spanning Tree

After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will 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, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



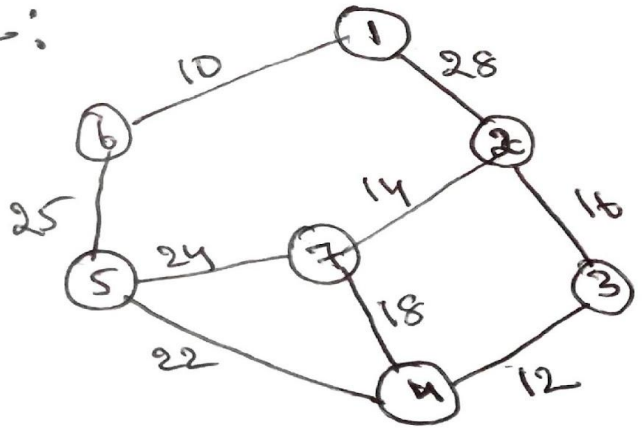
Minimum Cost Spanning Tree

Prim's Algorithm

A greedy method to obtain a minimum cost Spanning tree builds the tree edge by edge. The next edge to be included is choosen according to some optimization Criteria

- If A is the set of edges selected so far, then A forms a tree. The next edge (u, v) ~~be~~ is to be included in A is a minimum ~~cost~~ edge not in A, with the property that $A \cup \{(u, v)\}$ is also a tree.

Ex:



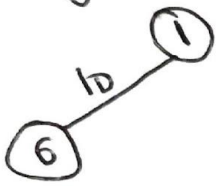
Minimum Cost Spanning Tree

- The prims algorithm will start with a tree that includes only a minimum cost edge of G.
- Then edges are added to this tree one by one.
- The next edge (i,j) to be added is such that i is a vertex already included in the tree, j is a vertex not yet included and cost of i,j, i.e cost(i,j) is min among all edges [k,l] such that k is in tree and l is not in the tree.

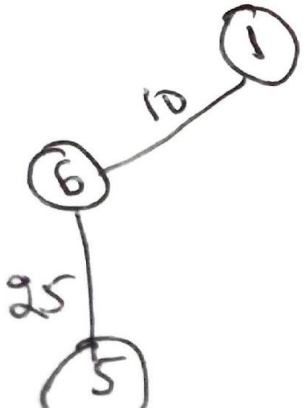
- Every time after selecting min-cost edge, check whether the inclusion of this edge making cycle in the Spanning tree of it makes cycle, then discard it and select next minimum cost edge. Otherwise add this edge into Spanning tree.

Minimum Cost Spanning Tree

- The min cost edge in the graph is (1,6) with cost 10. So include this edge into spanning tree. Now the tree is:



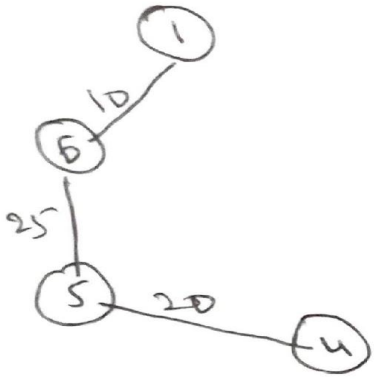
- Now consider nodes 1 and 6 and select nearest node from 1,6 in the graph. 1 has only one node 2 with cost 28 node 6 has nearest node 5 with cost 25.
- So (6,5) edge cost is less than (1,2). So include (6,5) edge into spanning tree.



Minimum Cost Spanning Tree

- now for the next edge, Consider nodes 1, 6 and 5 and select nearest nodes from these three nodes and select min edge cost among all.

so edge (5, 4) is included as next edge

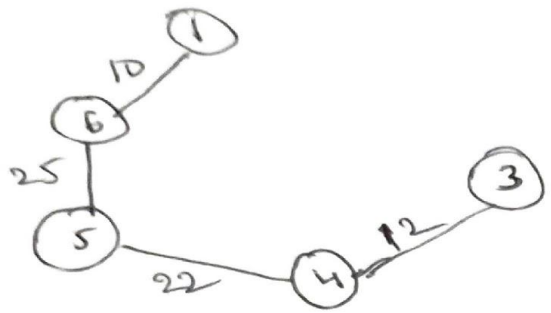


- now Consider the nodes in the tree 1, 6, 5 and 4 and apply the Same procedure.

So include (4, 3) as next min cost edge into Spanning tree

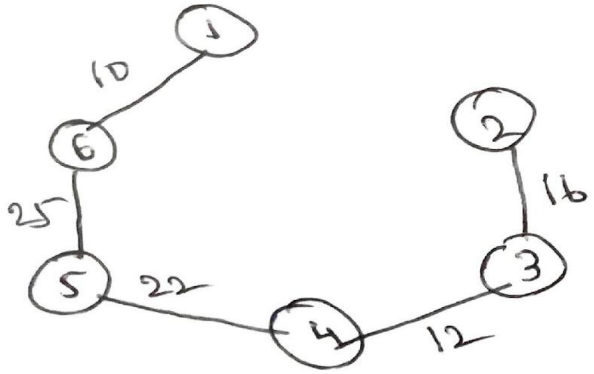
Minimum Cost Spanning Tree

The updated tree is:

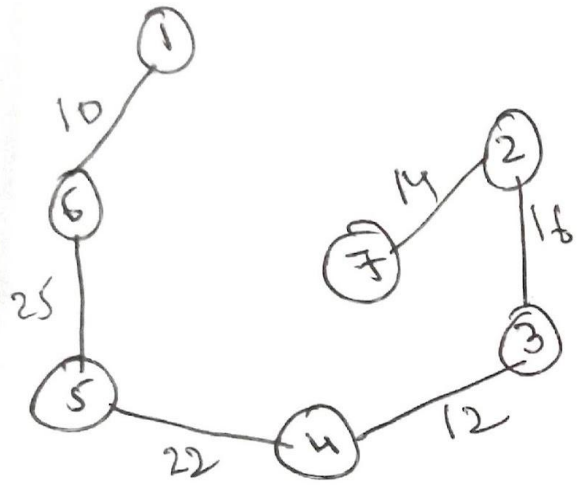


- Next Consider included Vertices in the tree and select nearest node from all the above Vertices and select min cost among ~~the~~ all.

min-cost edge is (3,2), add this edge into Spanning tree.



- for next edge repeat the same procedure. so the edge (2,7) is included as next min cost edge.



All the vertices are included into the spanning tree.

So this tree is the minimum-cost spanning tree.

The cost of the spanning tree is: 99

Minimum Cost Spanning Tree

Prim's Algorithm

```
Let (k, l) be an edge of min cost in E;
    mincost := cost[k, l].
t[1,1] := k; t[1,2] := l;
for i := 1 to n do // initialize near
if (cost[i, l] < cost[i, k] then near[i] := l;
else
near[i] := k;
near[k] := near[l] := 0;
```

Minimum Cost Spanning Tree

```
for i := 2 to n-1 do
  {
Let j be the index such that near[j] ≠ 0 and
(cost[j], near[j]) is minimum;
t[i,1] = j;  t[i,2] := near[j];
mincost := mincost + cost[j, near[j]];
  near[j] := 0;
for k := 1 to n do  // update near
if (near[k] ≠ 0) and cost[k, near[k]] > cost[k,j]
then  near[k] := j;
  }
return mincost;
}
```

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

MINIMUM SPANNING TREE
KRUSKAL'S ALGORITHM

Kruskal’s Algorithm

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

1. form a tree that includes every vertex
2. has the minimum sum of weights among all the trees that can be formed from the graph

Algorithm

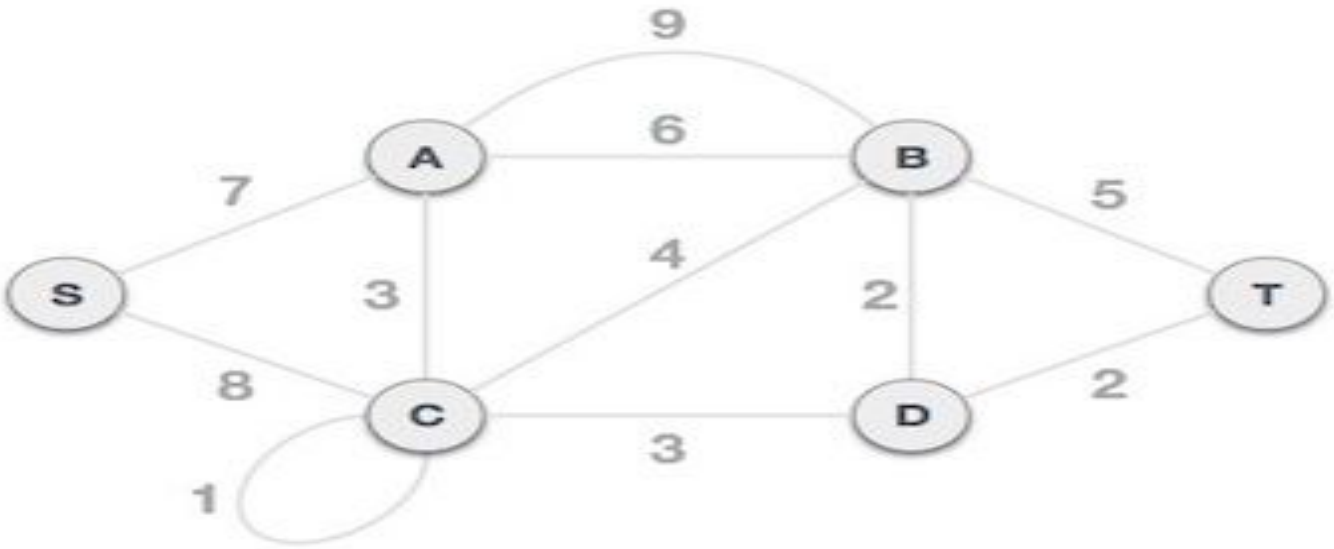
Input : Graph with V edges
Output : Minimum Spanning tree with V-1 edges

Step 1. Sort all the edges in non-decreasing order of their weight.
Step 2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
Step 3. Repeat step#2 until there are (V-1) edges in the spanning tree.

Minimum Cost Spanning Tree

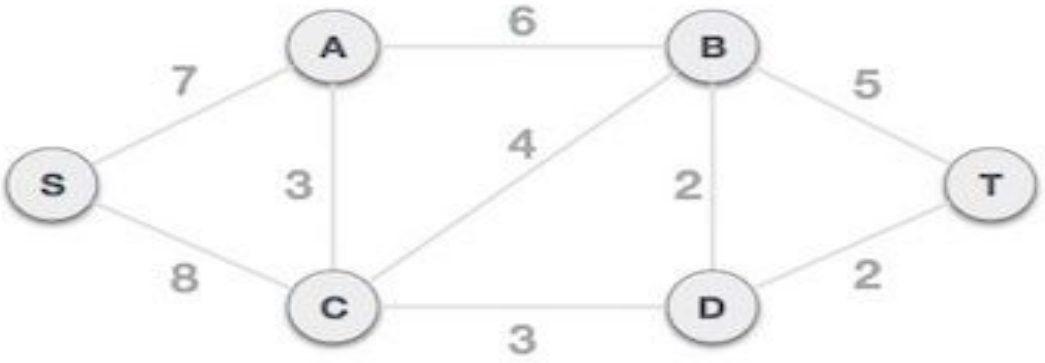
Kruskal's Algorithm: Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not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

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



Minimum Cost Spanning Tree

Step 1 - Remove all loops and Parallel Edges : Remove all loops and parallel edges from the given graph.



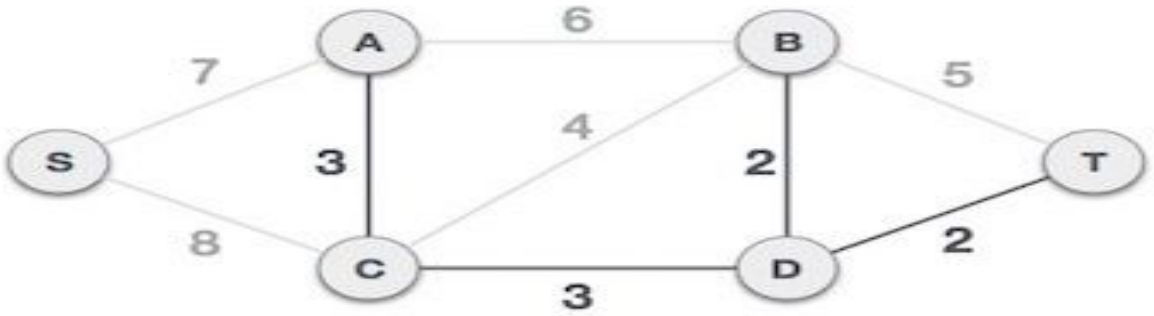
Step 2 - Arrange all edges in their increasing order of weight : The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

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

Minimum Cost Spanning Tree

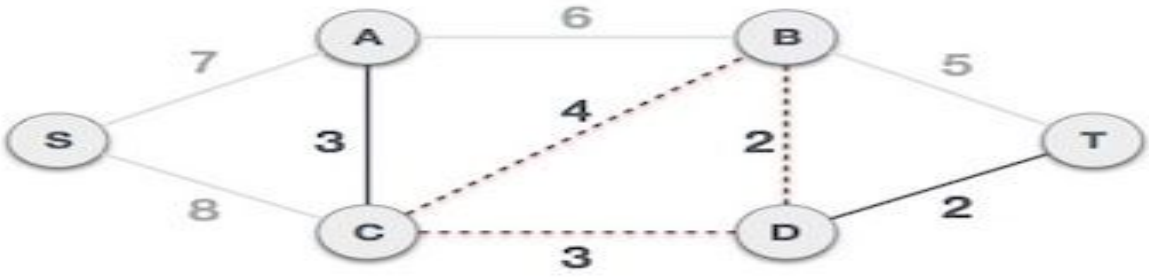
Step 3 - Add the edge which has the least weightage : Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection. Next cost is 3, and associated edges are A,C and C,D. We add them again −



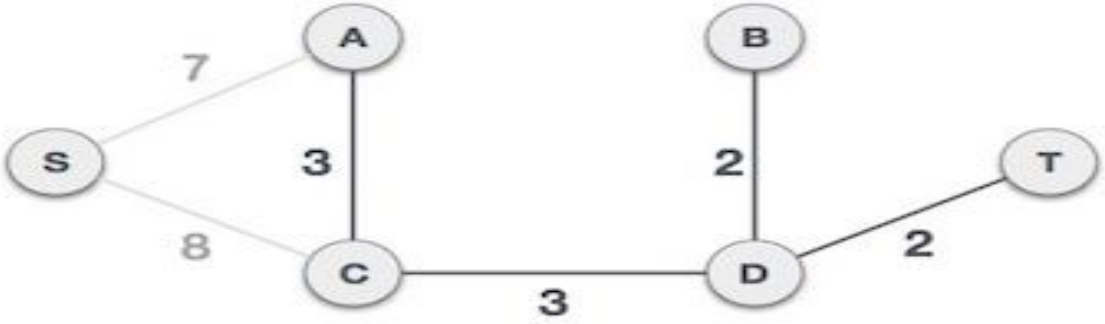
Minimum Cost Spanning Tree

Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



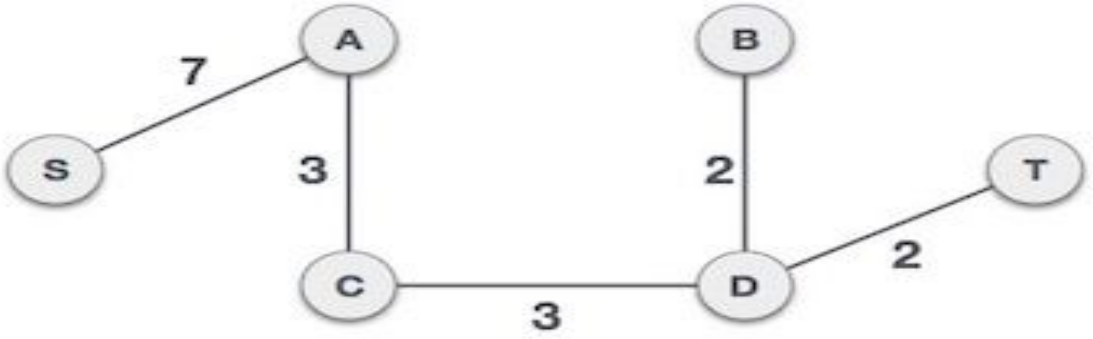
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.

We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Minimum Cost Spanning Tree

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



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

Minimum Cost Spanning Tree

Kruskal's Algorithm

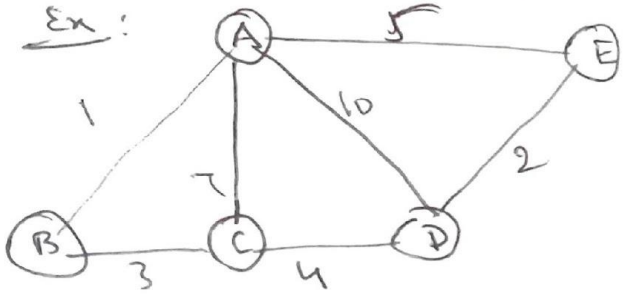
In this method, the edges of the graph are considered in non-decreasing order of edge costs. So arrange all the edges in decreasing order of its cost.

- Select minimum cost edge, verify whether the inclusion of this edge into spanning tree makes cycle. If not include it, otherwise discard it from consideration.
- Next select next minimum cost edge and do the same as above.
- Repeat the above selection procedure until including all the vertices ($n-1$ edges of n-vertex graph) into spanning tree.
- Before constructing spanning tree, first create forest with all the nodes and treat these nodes are as individual sets.

Minimum Cost Spanning Tree

- After selecting minimum cost edge, verify these two vertices of edge are there in same set or different sets
  - If these two nodes are there in the same set, then adding this node makes cycle in the spanning tree.
  - If both the nodes are there in the different sets then add this edge into spanning tree and also combine the two sets which are having the above two nodes.
- Repeat the procedure until adding n-1 edges into spanning tree (until adding all the nodes into the spanning tree.

Ex:



Minimum Cost Spanning Tree

The weight of the edges are given as

Edge	AE	AD	AC	AB	BC	CD	DE
weight	5	10	7	1	3	4	2

Sort the edges according to their weights

Edge	AB	DE	BC	CD	AE	AC	AD
weight	1	2	3	4	5	7	10

and Consider all the nodes as forest.

{A} {B} {C} {D} {E}

Minimum Cost Spanning Tree

- First min cost edge is (A, B) with edge cost 1. Both the nodes are there in two different sets. So add this edge into spanning tree and combine those two sets

{A, B} {C} {D} {E}

- Next min cost edge is (D, E) with edge cost 2. Both D and E are there in two different sets. So add (D, E) into spanning tree and combine these two sets

{A, B} {C} {D, E}

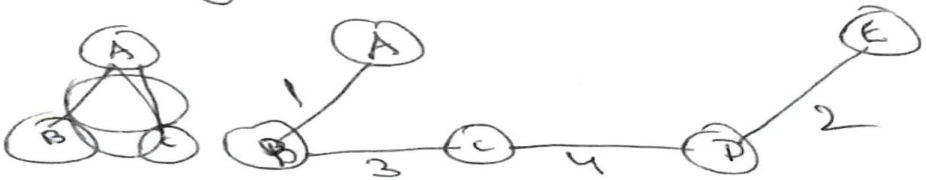
- Next min cost edge is (B, C). Both the nodes are there in two different sets, add them (combine two sets and add (B, C) into spanning tree

Minimum Cost Spanning Tree

{A, B, C} {D, E}

4

- Next minimum cost edge is (C, D), Both C and D are there in different sets, so add (C, D) into the spanning tree and Combine these two sets



{A, B, C, D, E}

- Next min cost edge is (A, E), But both A and E are there in the Same set so discard (A, E)
- Next min-cost edge is (A, C), Both A and C are in the Same set, inclusion of which makes cycle in the Spanning Tree.
- Next min-cost edge is (A, D), Both A and D are there in the Same Set, discard it.
- No other edge is available in the graph and all nodes have included into Spanning Tree
- So the Cost of the Spanning Tree is 10

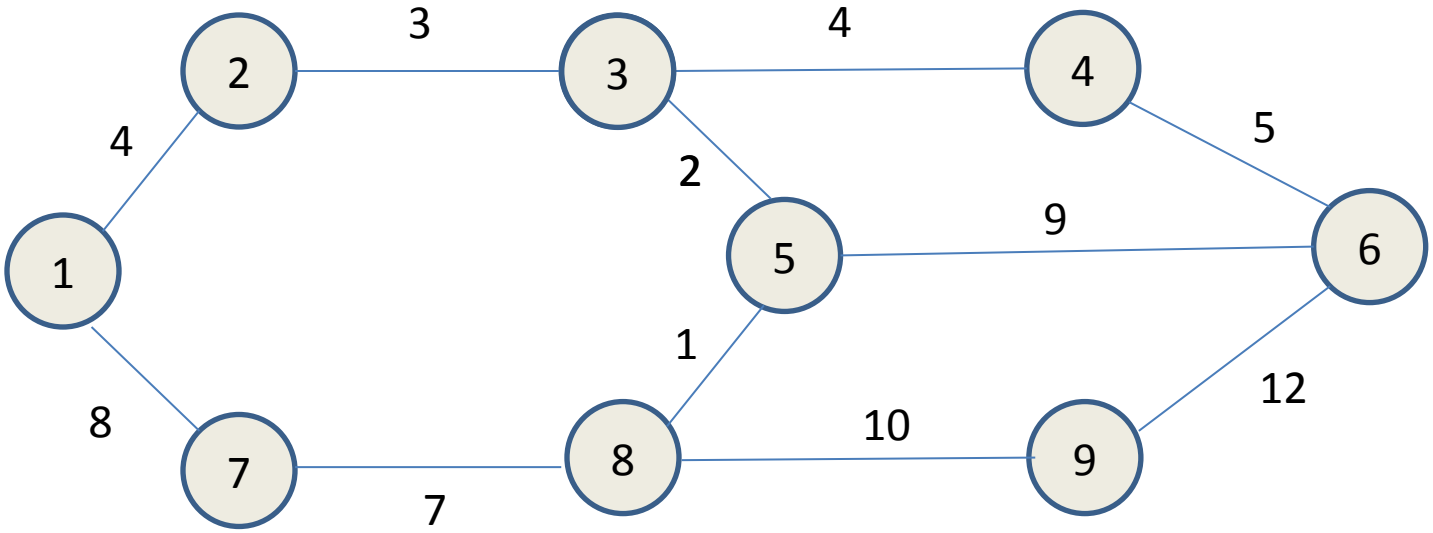
Minimum Cost Spanning Tree

```
Algorithm Kruskal (E, Cost, n, t)
{
Construct a heap out of the edge costs using Heapify;
for i := 1 to n do
  Parent[i] := -1;
i := 0; mincost := 0.0;
while (i < n-1) and (heap not empty) do
{
Delete a mincost edge (u, v) from the heap and
  reheapify using adjust.
j := Find(u); k := find(v);
```

Minimum Cost Spanning Tree

```
    if (j ≠ k) then
        {
        i := i+1;
    t[i,1] := u; t[i,2] := v;
    mincost := mincost + cost[u,v];
    union (j, k);
        }
}
if (i ≠ n-1) then write ("No Spanning Tree");
else
return mincost;
}
```

Example 2



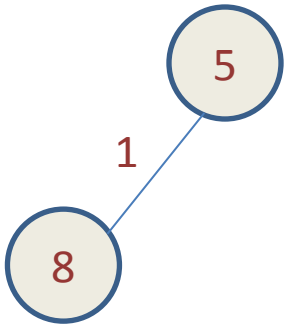
The graph contains 9 vertices and 11 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

Sort the weights of the graph

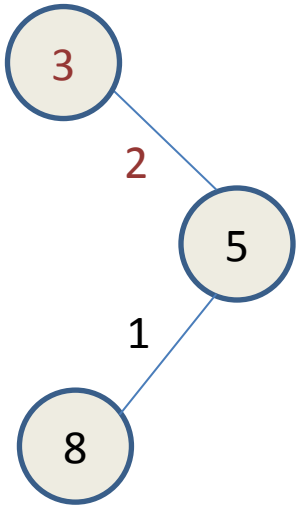
Weight	Source	Destination
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6

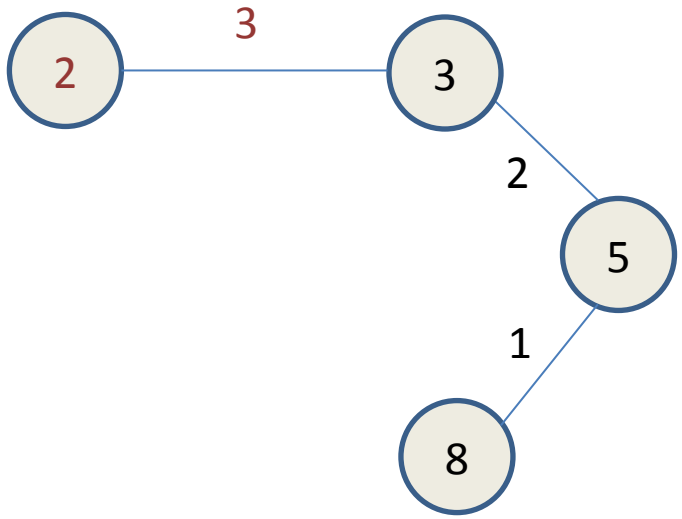
Now pick all edges one by one from sorted list of edges

Weigh t	Source	Destina tion
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6



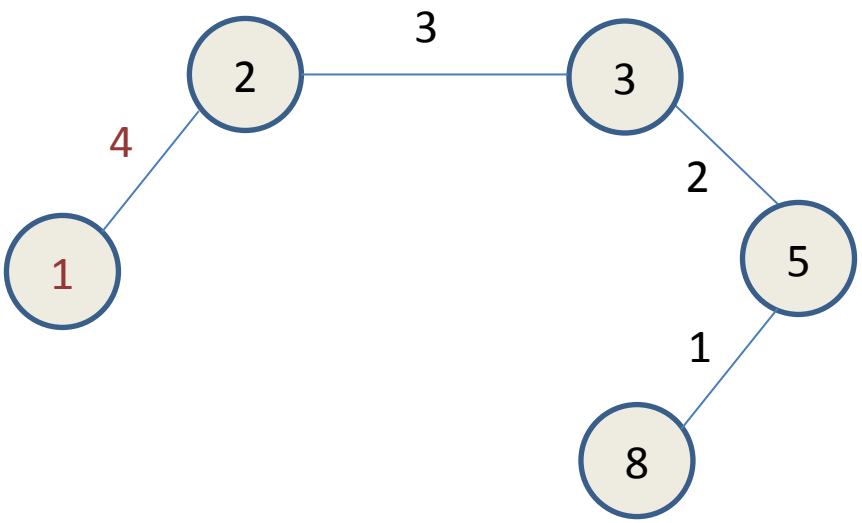
Weigh t	Source	Destina tion
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6



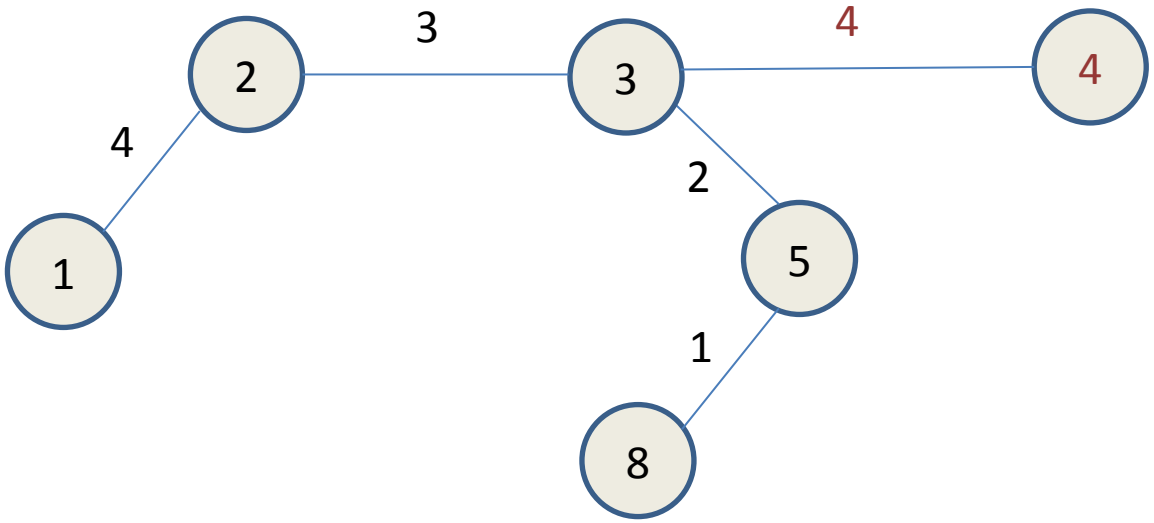


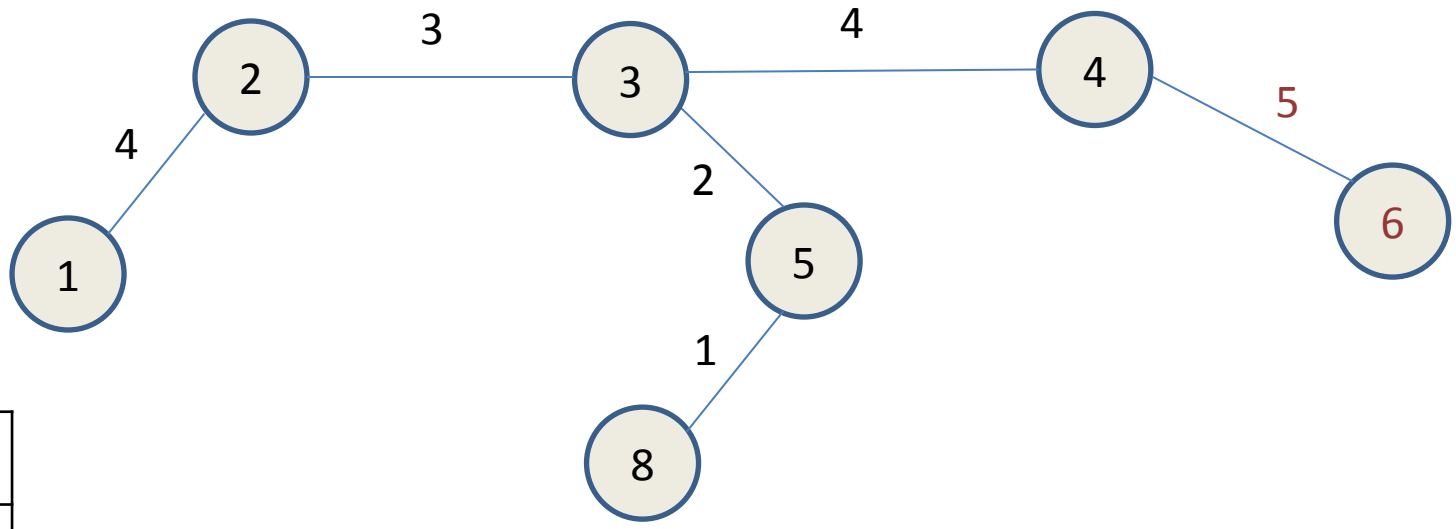
Weigh t	Source	Destina tion
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6

Weigh t	Source	Destina tion
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6

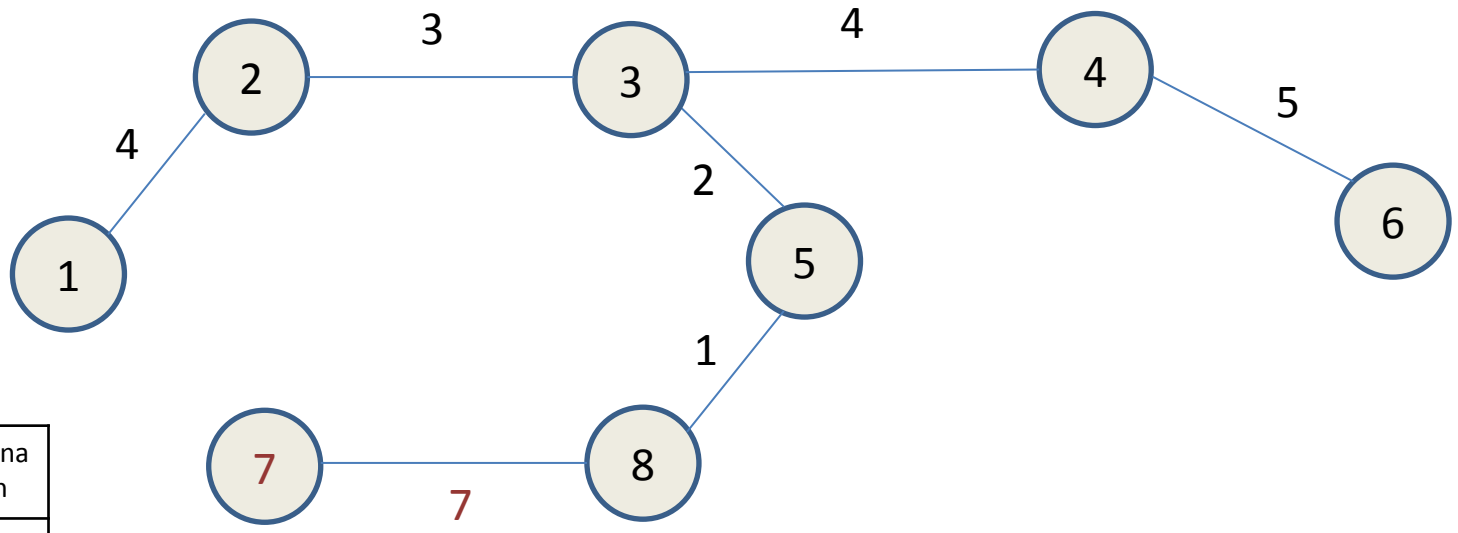


Weigh t	Source	Destina tion
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6

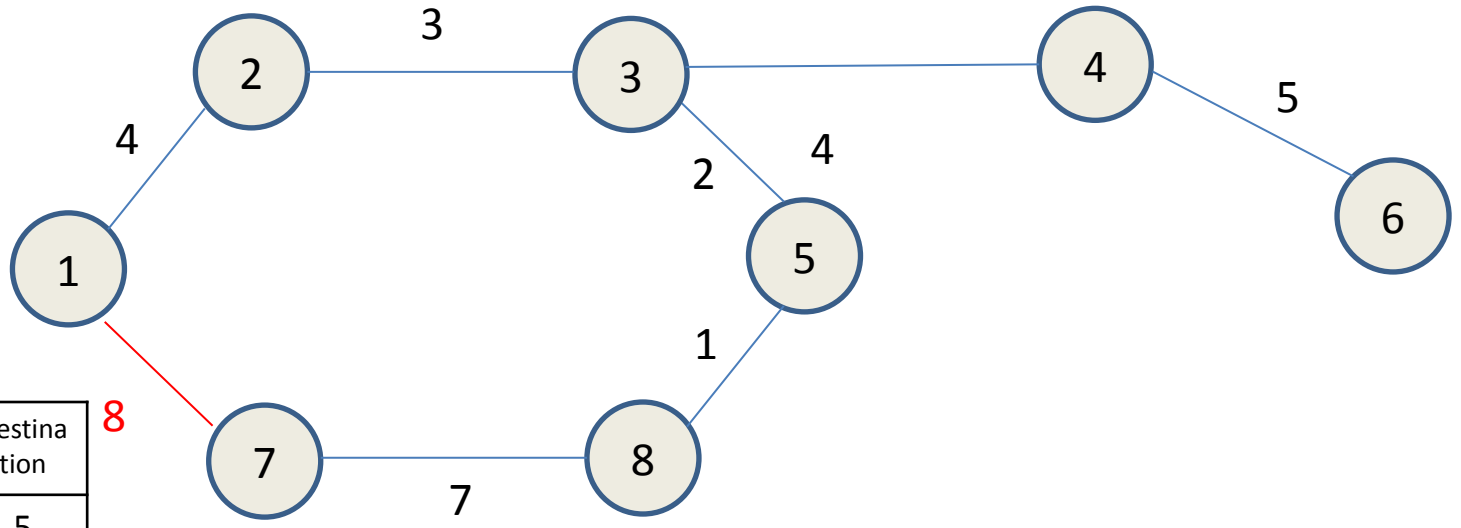




Weigh t	Source	Destina tion
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6

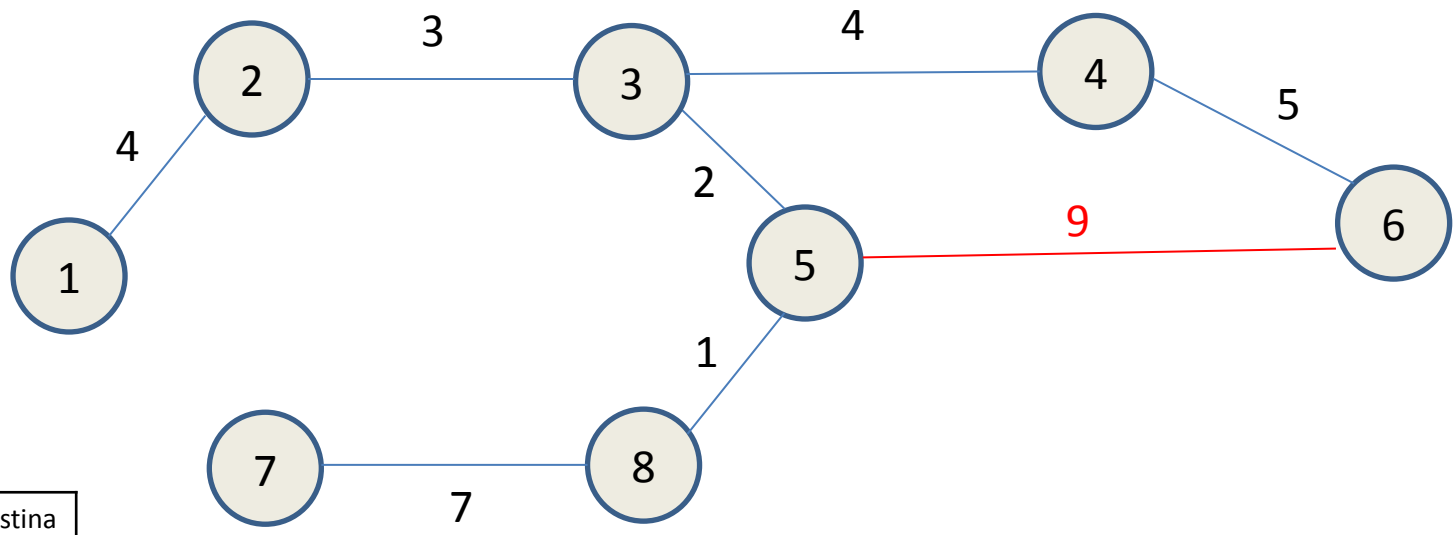


Weigh t	Source	Destina tion
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6



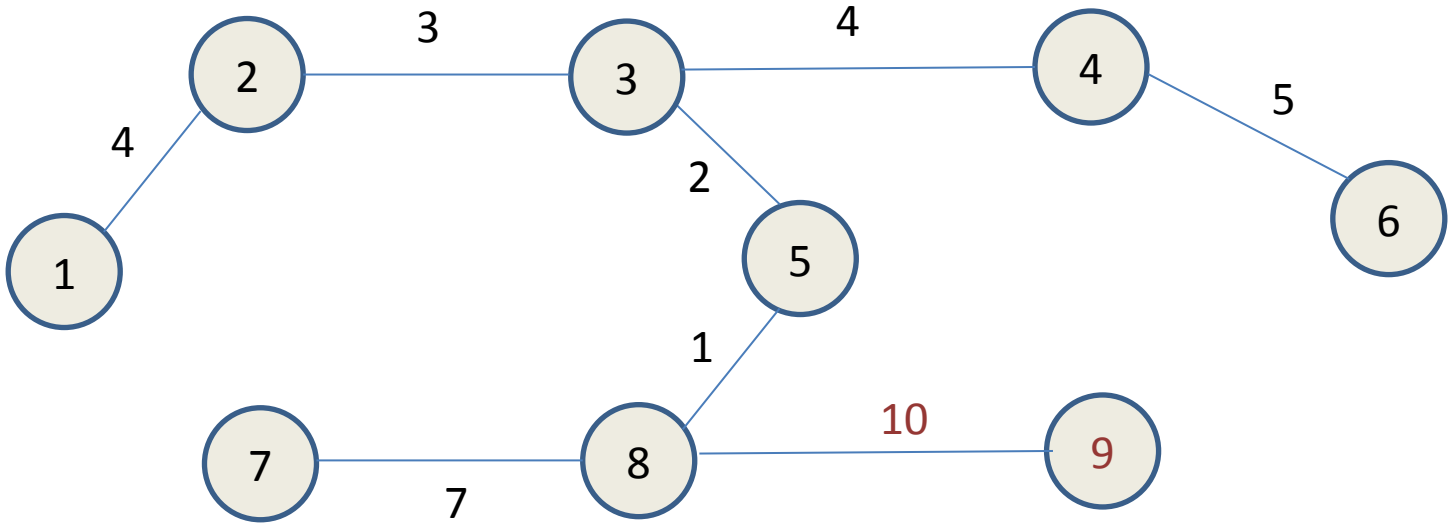
Weigh t	Source	Destina tion
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6

Cycle is formed because of weight 8. hence it is discarded



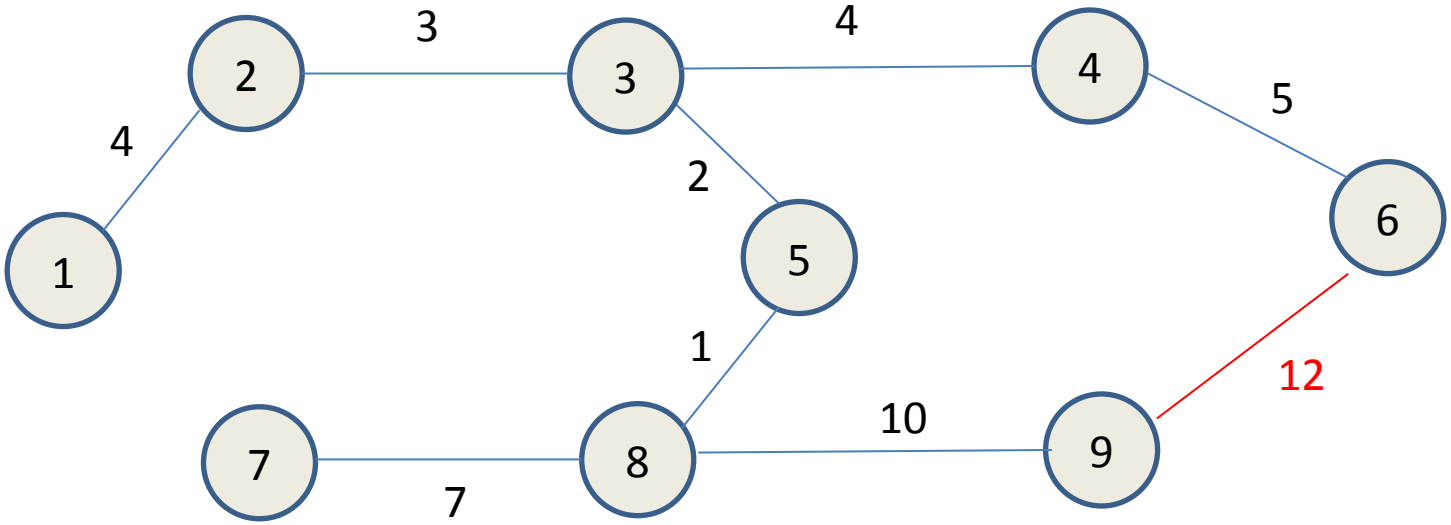
Weigh t	Source	Destina tion
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6

Cycle is formed because of weight 9. hence it is discarded.



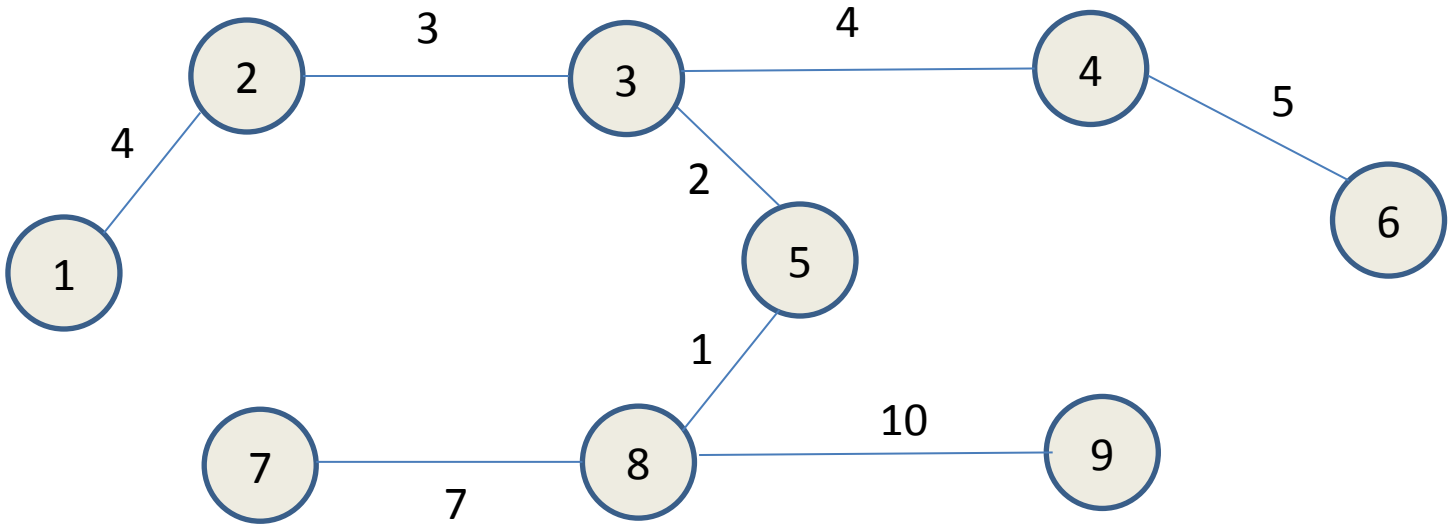
Weigh t	Source	Destina tion
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6

Cycle is formed because of weight 12. hence it is discarded.



Weigh t	Source	Destina tion
1	8	5
2	3	5
3	2	3
4	1	2
4	3	4
5	4	6
7	7	8
8	1	7
9	5	6
10	8	9
12	9	6

Cycle is formed because of weight 12. hence it is discarded.



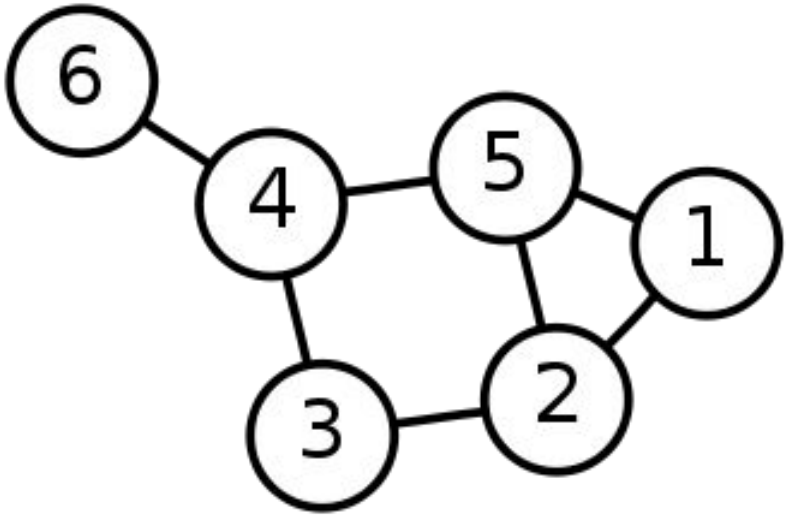
Minimum Spanning tree generated by Kruskal's algorithm

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

Dijkstra's algorithm

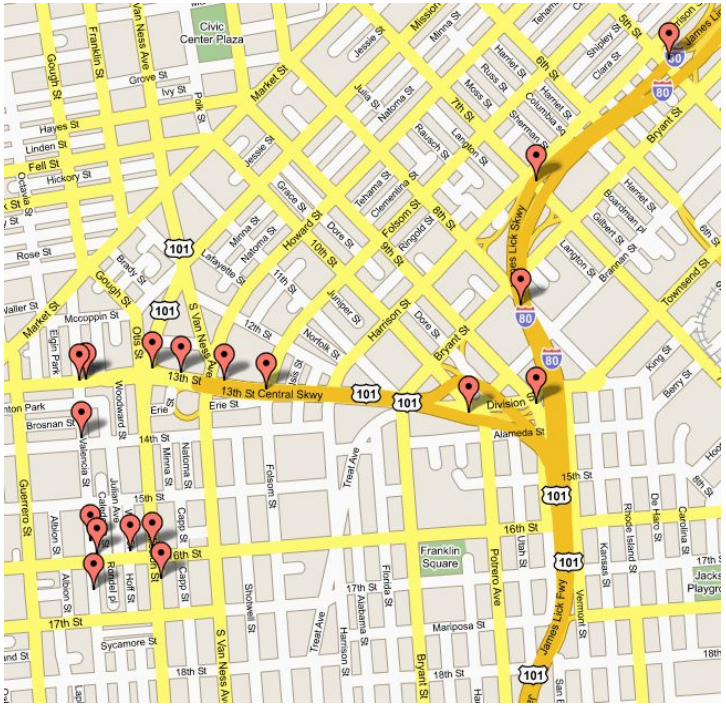
Shortest Path Problem

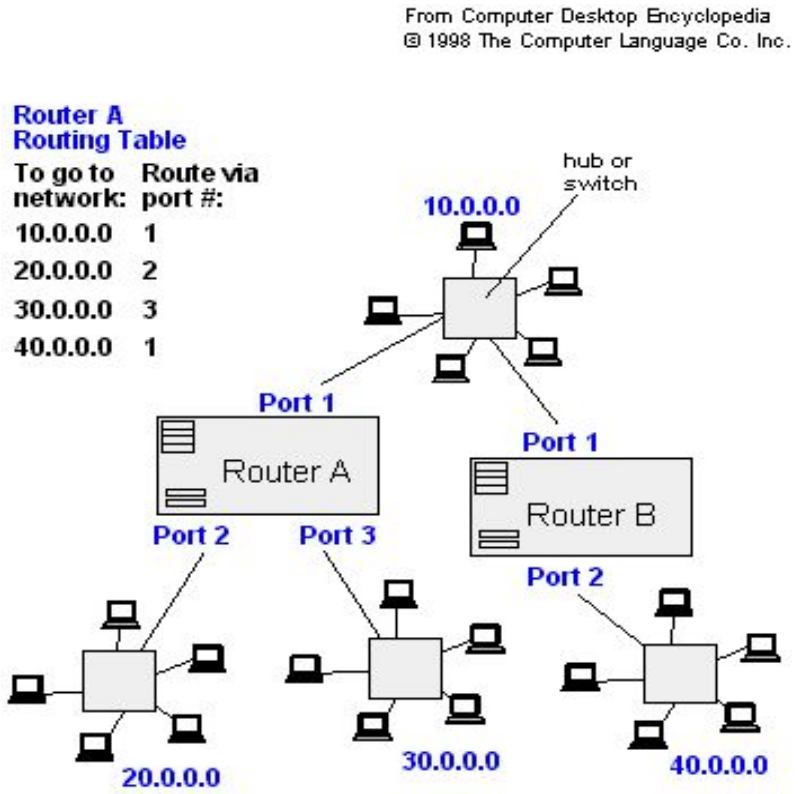
Single-Source Shortest Path Problem - The problem of finding shortest paths from a source vertex *v* to all other vertices in the graph.



Applications

- Maps (Map Quest, Google Maps)
- Routing Systems





Dijkstra's algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Input: Weighted graph G={E,V} and source vertex $v \in V$, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

Approach

1. The algorithm computes for each vertex u the distance to u from the start vertex v, that is, the weight of a shortest path between v and u.
2. The algorithm keeps track of the set of vertices for which the distance has been computed, called the cloud C
3. Every vertex has a label D associated with it. For any vertex u, D[u] stores an approximation of the distance between v and u. The algorithm will update a D[u] value when it finds a shorter path from v to u.
4. When a vertex u is added to the cloud, its label D[u] is equal to the actual (final) distance between the starting vertex v and vertex u.

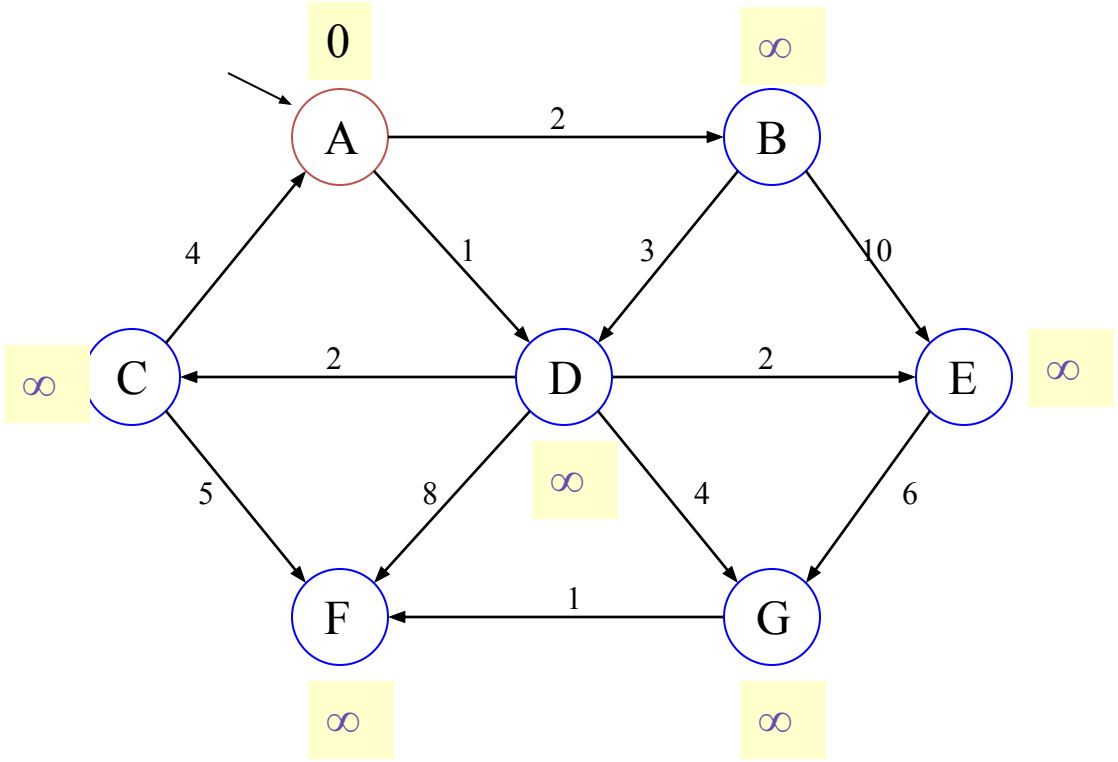
Dijkstra pseudocode

```
Dijkstra(v1, v2):
    for each vertex v:                          // Initialization
        v's distance := infinity.
        v's previous := none.
    v1's distance := 0.
    List := {all vertices}.

    while List is not empty:
        v := remove List vertex with minimum distance.
        mark v as known.
        for each unknown neighbor n of v:
            dist := v's distance + edge (v, n)'s weight.

            if dist is smaller than n's distance:
                n's distance := dist.
                n's previous := v.

    reconstruct path from v2 back to v1,
    following previous pointers.
```

Example: Initialization

Let us consider A as the source node. Distance of all vertices except source is ∞

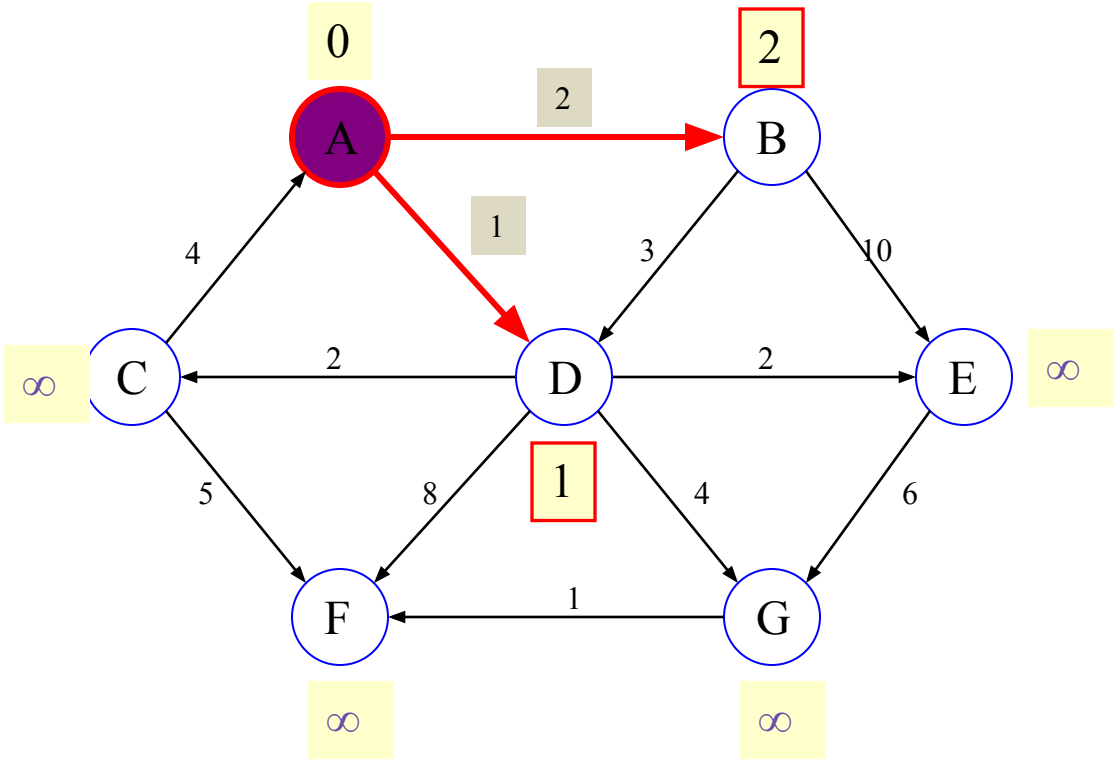
Distance(source) = 0



Pick vertex in List with minimum distance.

Update neighbour's distance

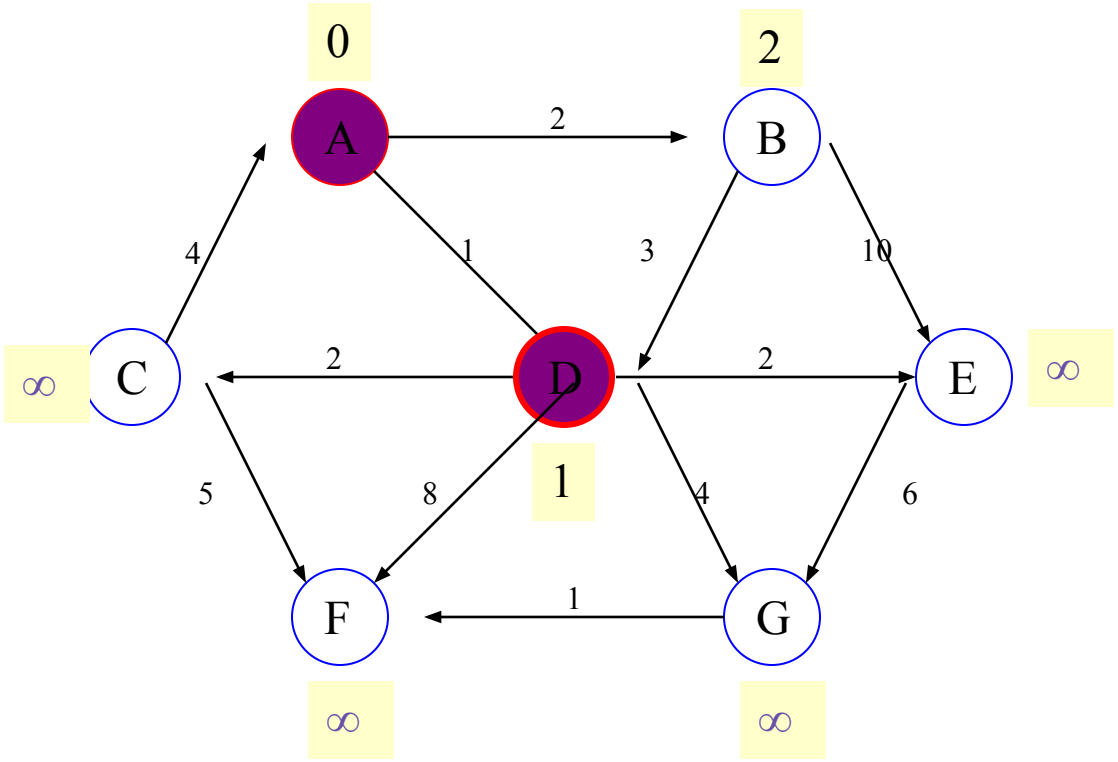
Select the source node A and check the neighbour nodes it reaches. Update the weights of B and D as 2 and 1.



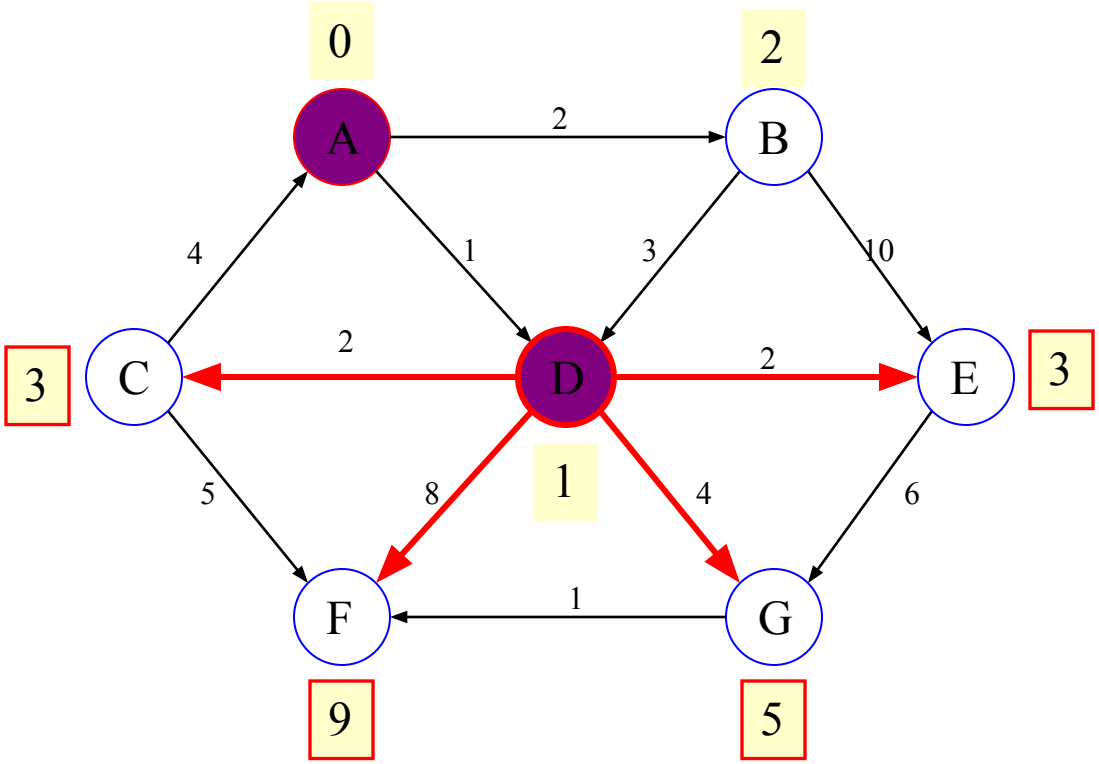
Distance(B) = 2

Distance(D) = 1

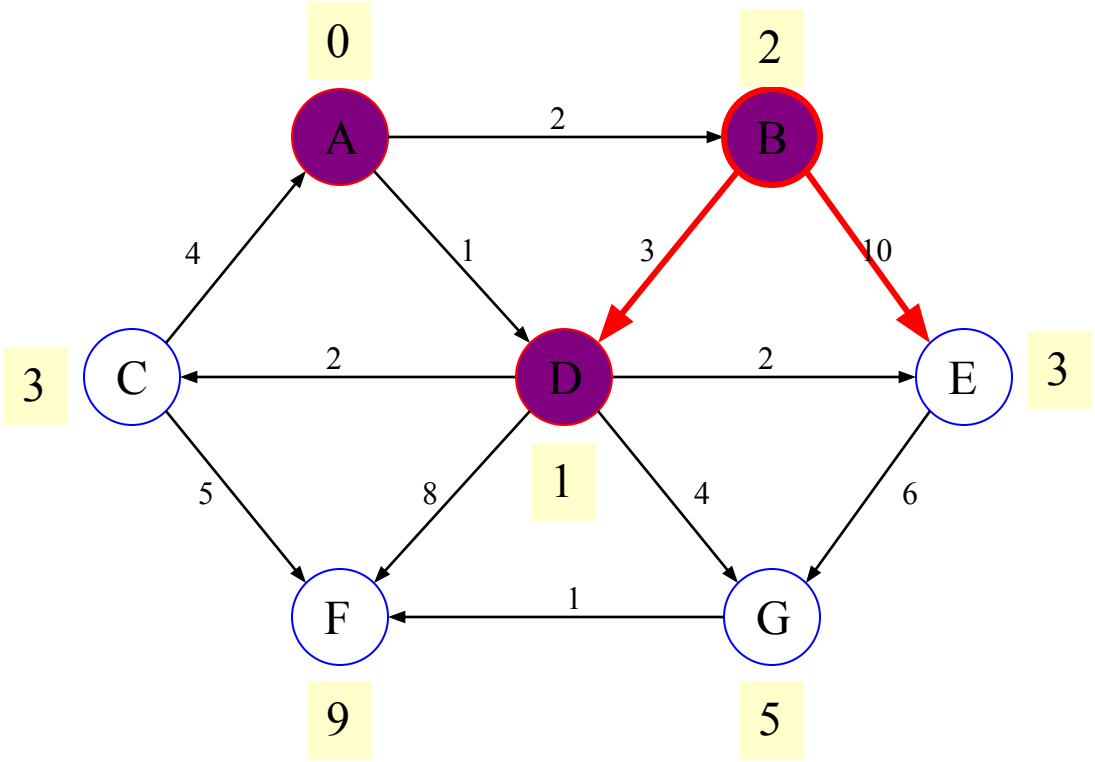
Now, select the node with minimum distance. Weights are 2 and 1 hence 1 is selected i.e., D is chosen



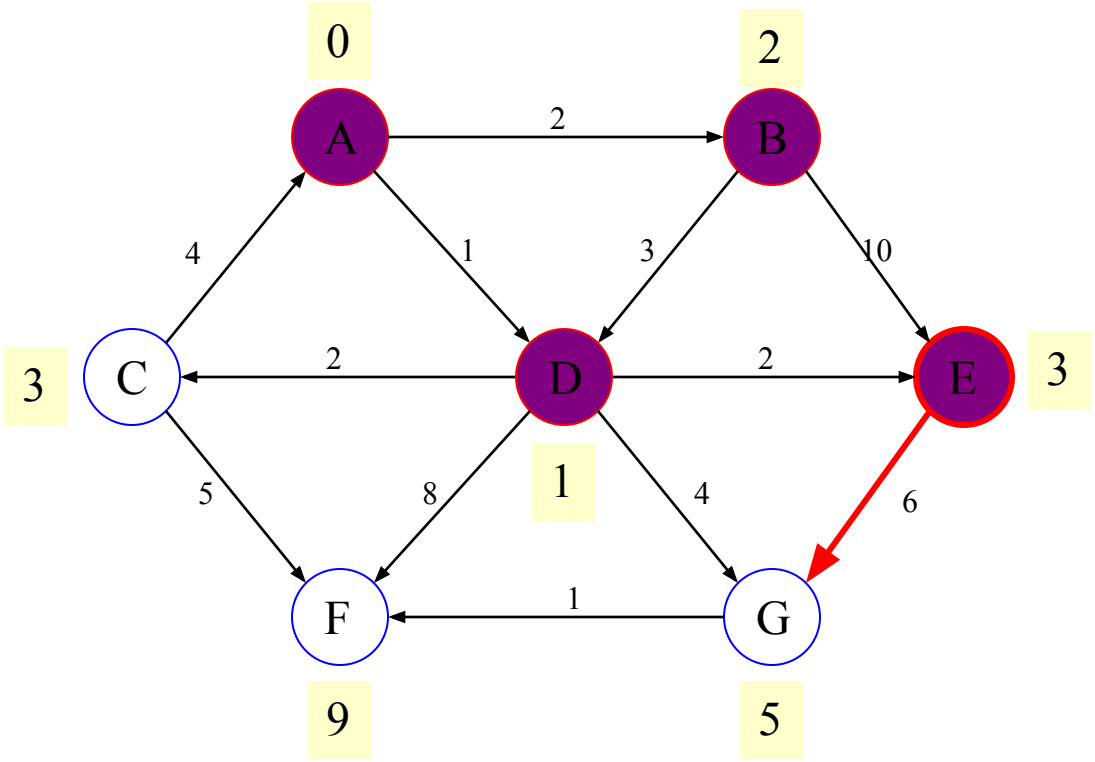
Neighbors of D are selected and their distances are updated.
Distance of C = Distance of D + weight of D to C = 1+ 2=3
Distance of E = Distance of D + weight of D to E = 1+ 2=3
Distance of F = Distance of D + weight of D to F = 1+ 8=9
Distance of G = Distance of D + weight of D to G = 1+ 4=5



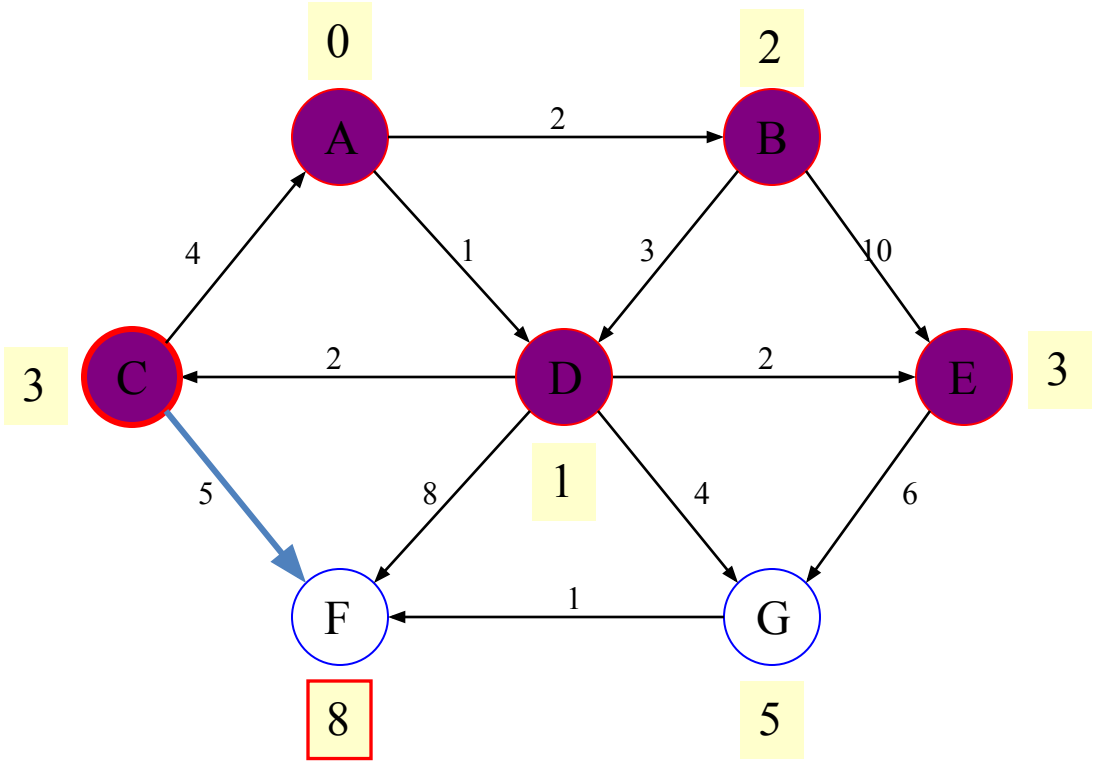
Pick vertex in List with minimum distance and update neighbors.
B has the minimum distance. Neighbours of B are D and E.
Distance of D is not updated since D is already visited.
Previous distance of E = 3
New distance of E = 2+10 = 12
Distance of E not updated since it is larger than previously computed.



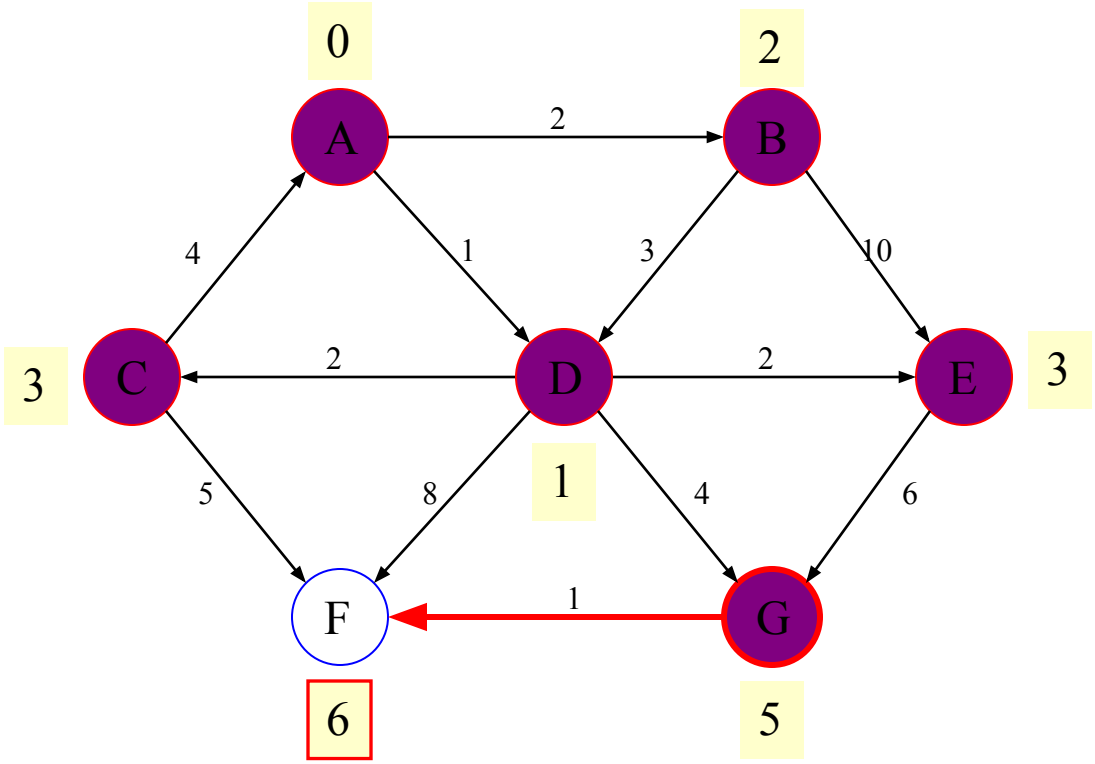
Pick vertex List with minimum distance and update neighbors. E is selected.
Distance of G not updated since it is larger than previously computed.



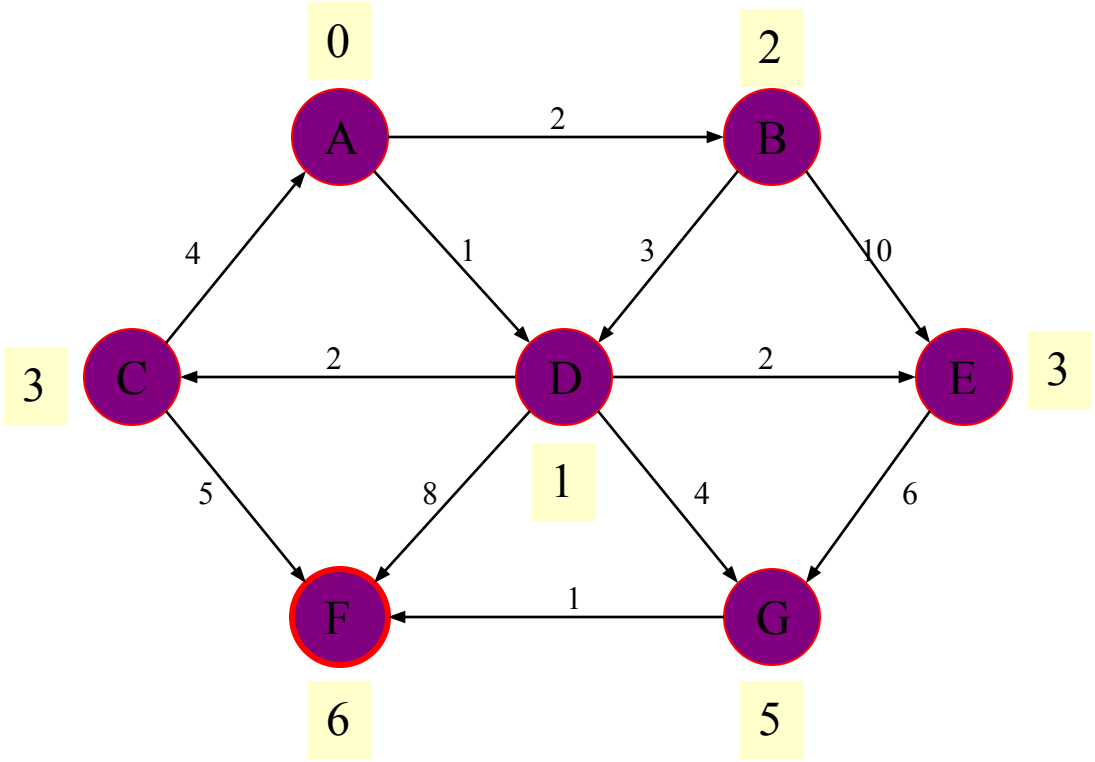
Pick vertex List with minimum distance and update neighbors.
C is selected. Neighbour is F.
Distance of F = Distance of C + weight of C to F = 3 + 5 = 8



Pick vertex List with minimum distance and update neighbors
G is selected. F is the neighbour.
Previous distance of F is 8.
Distance of F = Distance of G+weight from G to F = 5+1
Update the distance of F as 6.



F is selected and there are no neighbors to F. This is resultant the shortest path graph



Example : 2

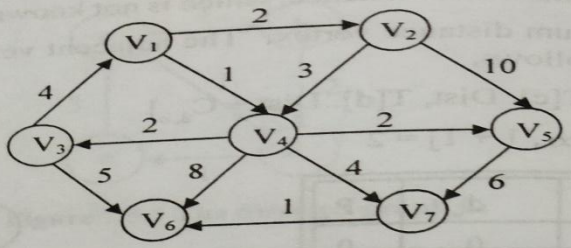


Fig. 7.5.4 The directed Graph G

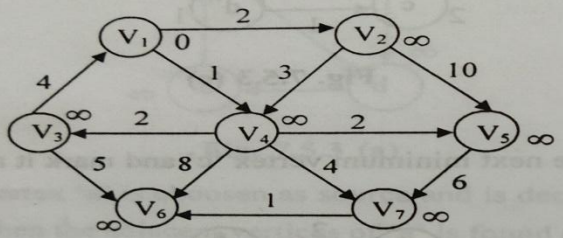


Fig. 7.5.4 (a)

V	known	d_v	P_v
V_1	0	0	0
V_2	0	∞	0
V_3	0	∞	0
V_4	0	∞	0
V_5	0	∞	0
V_6	0	∞	0
V_7	0	∞	0

INITIAL CONFIGURATION

V_1 is taken as source vertex, and is marked known. Then the d_v and P_v of its adjacent vertices are updated.

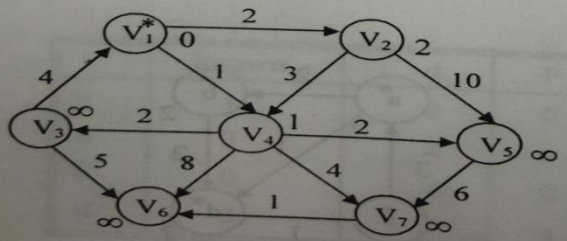


Fig. 7.5.4(b)

V	known	d_v	P_v
V_1	1	0	0
V_2	0	2	V_1
V_3	0	∞	0
V_4	0	1	V_1
V_5	0	∞	0
V_6	0	∞	0
V_7	0	∞	0

After V_1 is declared known

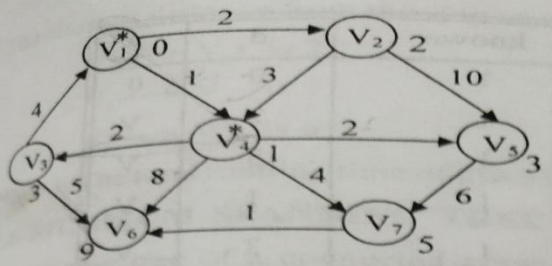


Fig. 7.5.4(c)

V	known	d_v	P_v
V_1	1	0	0
V_2	0	2	V_1
V_3	0	3	V_4
V_4	1	1	V_1
V_5	0	3	V_4
V_6	0	9	V_4
V_7	0	5	V_4

After V_4 is declared known

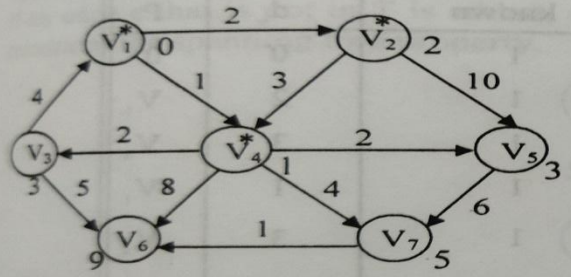


Fig. 7.5.4 (d)

V	known	d_v	P_v
V_1	1	0	0
V_2	1	2	V_1
V_3	0	3	V_4
V_4	1	1	V_1
V_5	0	3	V_4
V_6	0	9	V_4
V_7	0	5	V_4

After V_2 is declared known

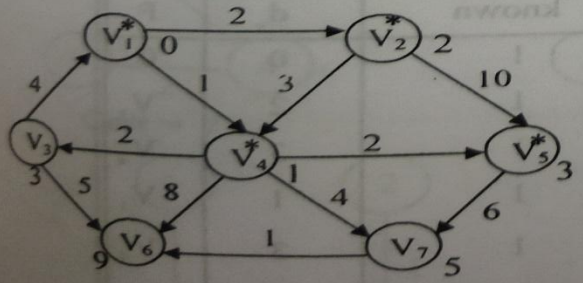


Fig. 7.5.4(e)

V	known	d_v	P_v
V_1	1	0	0
V_2	1	2	V_1
V_3	0	3	V_4
V_4	1	1	V_1
V_5	1	3	V_4
V_6	0	9	V_4
V_7	0	5	V_4

After V_5 is declared known

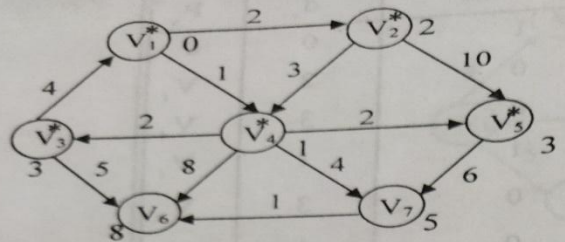


Fig. 7.5.4(f)

V	known	d_v	P_v
V_1	1	0	0
V_2	1	2	V_1
V_3	1	3	V_4
V_4	1	1	V_1
V_5	1	3	V_4
V_6	0	8	V_3
V_7	0	5	V_4

After V_3 is declared known

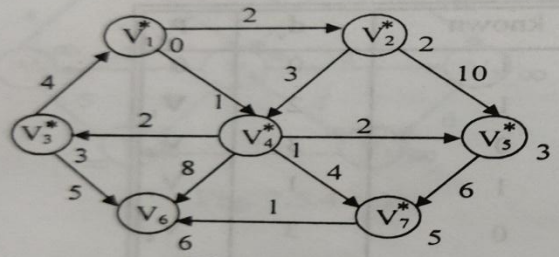


Fig. 7.5.4(g)

V	known	d_v	P_v
V_1	1	0	0
V_2	1	2	V_1
V_3	1	3	V_4
V_4	1	1	V_1
V_5	1	3	V_4
V_6	0	6	V_7
V_7	1	5	V_4

After V_7 is declared known

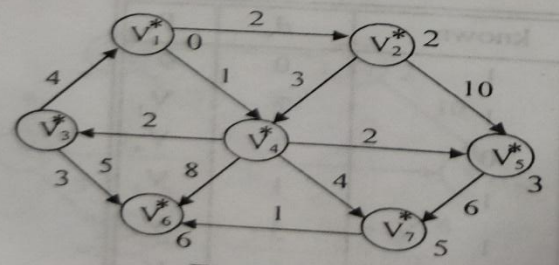


Fig. 7.5.4 (h)

V	known	d_v	P_v
V_1	1	0	0
V_2	1	2	V_1
V_3	1	3	V_4
V_4	1	1	V_1
V_5	1	3	V_4
V_6	1	6	V_7
V_7	1	5	V_4

After V_6 is declared known and algorithm terminates

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

Hashing and Collision

Introduction

In hashing, large keys are converted into small keys by using hash functions.

• The values are then stored in a data structure called hash table.

• The idea of hashing is to distribute entries (key/value pairs) uniformly across an array.

• Each element is assigned a key (converted key).

• By using that key you can access the element in O(1) time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

Hashing is implemented in two steps:

• An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.

• The element is stored in the hash table where it can be quickly retrieved using hashed key. • hash = hashfunc(key) index = hash % array_size.

INTRODUCTION

In this case we can directly access the record of any employee, once we know his Emp_ID, because array index is same as that of Emp_ID number. But practically, this implementation is hardly feasible.

Let us assume that the same company use a five digit Emp_ID number as the primary key. In this case, key values will range from 00000 to 99999. If we want to use the same technique as above, we will need an array of size 100,000, of which only 100 elements will be used.

HASH FUNCTION

- A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table

- The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

- To achieve a good hashing mechanism, It is important to have a good hash function with the following basic requirements:

- Easy to compute
- Uniform distribution
- Less Collision

HASH FUNCTION

Types

Division Method.
Mid Square Method.
Folding Method.
Multiplication Method.

HASH FUNCTION

1. Division Method

- Division method is the most simple method of hashing an integer *x.* The method divides *x* by *M* and then use the remainder thus obtained. In this case, the hash function can be given as

  h(x) = x mod M

- The division method is quite good for just about any value of *M* and since it requires only a single division operation, the method works very fast. However, extra care should be taken to select a suitable value for *M*.

HASH FUNCTION

- For example, *M* is an even number, then *h*(*x*) is even if *x* is even; and *h*(*x*) is odd if *x* is odd. If all possible keys are equi-probable, then this is not a problem. But if even keys are more likely than odd keys, then the division method will not spread hashed values uniformly.
- Generally, it is best to **choose *M* to be a prime number** because making *M* a prime increases the likelihood that the keys are mapped with a uniformity in the output range of values. Then M should also be not too close to exact powers of 2. if we have,

  h(k) = x mod 2k
- then the function will simply extract the lowest *k* bits of the binary representation of *x*

HASH FUNCTION

- A potential drawback of the division method is that using this method, consecutive keys map to consecutive hash values. While on one hand this is good as it ensures that consecutive keys do not collide, but on the other hand it also means that consecutive array locations will be occupied. This may lead to degradation in performance.

- Example: Calculate hash values of keys 1234 and 5462.

  Setting m = 97, hash values can be calculated as

  h(1234) = 1234 % 97 = 70

  h(5642) = 5642 % 97 = 16

HASH FUNCTION

2. Multiplication Method

The steps involved in the multiplication method can be given as below:

Step 1: Choose a constant A such that $0 < A < 1$.

Step 2: Multiply the key k by A

Step 3: Extract the fractional part of kA

Step 4: Multiply the result of Step 3 by m (size of the hash table i.e. M.)

Hence, the hash function can be given as,

$$h(K) = floor\ (M\ (kA \bmod 1))$$

where, kA mod 1 gives the fractional part of kA and m is the total number of indices in the hash table

The greatest advantage of the multiplication method is that it works practically with any value of A. Although the algorithm works better with some values than the others but the optimal choice depends on the characteristics of the data being hashed. Knuth has suggested that the best choice of A is

» (sqrt5 - 1) /2 = 0.6180339887

HASH FUNCTION

Multiplication Method – Example

$k = 12345$

$A = 0.357840$

$M = 100$

$h(12345) = floor[\ 100\ (12345*0.357840 \bmod 1)]$

$= floor[\ 100\ (4417.5348 \bmod 1)\ ]$

$= floor[\ 100\ (0.5348)\ ]$

$= floor[\ 53.48\ ]$

$= 53$

3. Mid Square Method:

Example: Calculate the hash value for keys 1234 and 5642 using the mid square method. The hash table has 100 memory locations.

Note the hash table has 100 memory locations whose indices vary from 0-99. this means, only two digits are needed to map the key to a location in the hash table, so $r = 2$.

When k = 1234, k2 = 1522756, h (k) = 27
When k = 5642, k2 = 31832164, h (k) = 21

Observe that 3rd and 4th digits starting from the right are chosen.

4. Folding Method

The folding method works in two steps.
Step 1: Divide the key value into a number of parts. That is divide k into parts, *k1, k2, ..., kn,* where each part has the same number of digits except the last part which may have lesser digits than the other parts.
Step 2: Add the individual parts. That is obtain the sum of *k1 + k2 + .. + kn*. Hash value is produced by ignoring the last carry, if any.

Note that the number of digits in each part of the key will vary depending upon the size of the hash table. For example, if the hash table has a size of 1000. Then it means there are 1000 locations in the hash table. To address these 1000 locations, we will need at least three digits, therefore, each part of the key must have three digits except the last part which may have lesser digits.

4. Folding Method

Formula:

k = k1, k2, k3, k4, ….., kn

s = k1+ k2 + k3 + k4 +….+ kn

h(K)= s

Here,

s is obtained by adding the parts of the key **k**

Example:

k = 12345

k1 = 12, k2 = 34, k3 = 5

s = k1 + k2 + k3

= 12 + 34 + 5

= 51

h(K) = 51

HASH TABLE

- A hash table is a data structure that is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an By using a good hash function, hashing can work well.

- Under reasonable assumptions, the average time required to search for an element in a hash table is O(1).

- Hash Table is a data structure in which keys are mapped to array positions by a hash function.

- A value stored in the Hash Table can be searched in O(1) time using a hash function to generate an address from the key (by producing the index of the array where the value is stored).

HASH TABLE

- When the set K of keys that are actually used is much smaller than that of U, a hash table consumes much less storage space. The storage requirement for a hash table is just O(k), where k is the number of keys actually used.

- In a hash table, an element with key k is stored at index h(k) not k. This means, a hash function h is used to calculate the index at which the element with key k will be stored. Thus, the process of mapping keys to appropriate locations (or indexes) in a hash table is called ***hashing***.

COLLISION

- If x1 and x2 are two different keys, it is possible that h(x1) = h(x2). This is called a collision.
- Collision resolution is the most important issue in hash table implementations.
- Choosing a hash function that minimizes the number of collisions and also hashes uniformly is another critical issue.

•Some of the Collision Resolution Techniques:

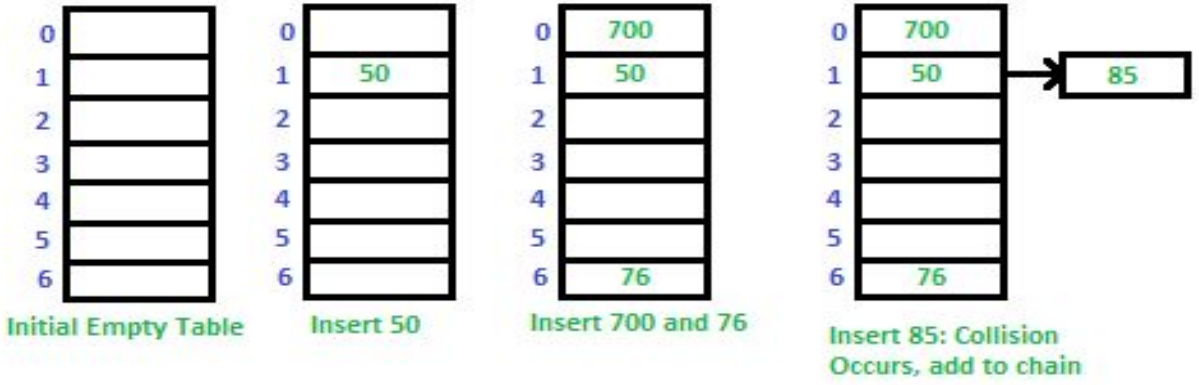
- Separate chaining (open hashing)
- Linear probing (open addressing or closed hashing)
- Multiple Hashing
- Quadratic Probing
- Double hashing

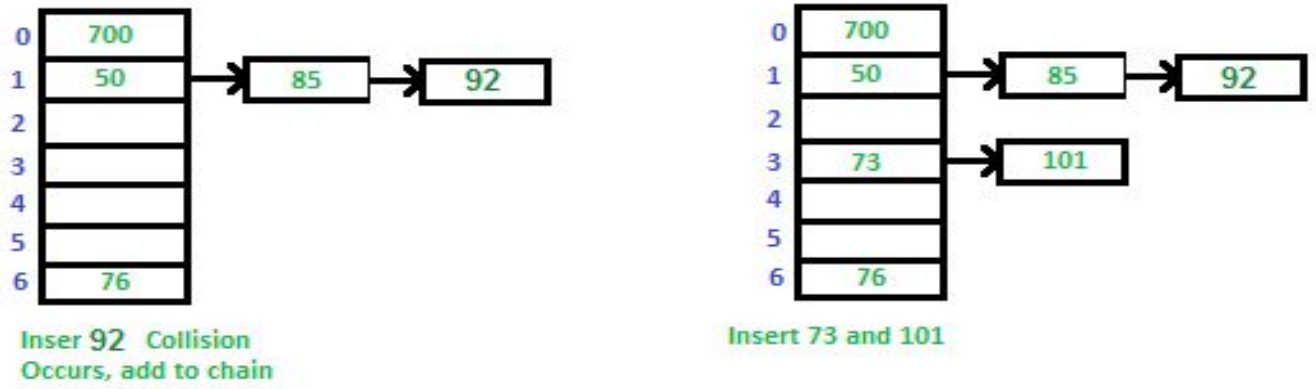
1. SEPARATE CHAINING

- Separate chaining is one of the most commonly used collision .
- It is usually implemented using linked lists. In separate chaining, each element of the hash table is a linked list.
- To store an element in the hash table you must insert it into a specific linked list.
- A pointer field is added to each record location.
- Whenever Overflow Occurs this pointer is set to point to overflow blocks making a linkedlist.
- If there is any collision (i.e. two different elements have same hash value) then store both the elements in the same linked list.

SEPARATE CHAINING

- **Example:** Let us consider a simple hash function as "**key mod 7**" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101
- Hash(k) = k % Tablesize Hash(50) = 50%7=1





SEPARATE CHAINING

Advantages:

- Simple to implement.
- Hash table never fills up, we can always add more elements to the chain.
- Less sensitive to the hash function or load factors.
- It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

- The cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
- Wastage of Space (Some Parts of the hash table are never used)
- If the chain becomes long, then search time can become O(n) in the worst case
- Uses extra space for links

2. Open Addressing

- In Open Addressing, all elements are stored in the **hash table** itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed). This approach is also known as closed hashing. This entire procedure is based upon probing. We will understand the types of probing ahead:

- The process of examining memory locations in the hash table is called probing. Open addressing technique can be implemented using- linear probing, quadratic probing and double hashing. We will discuss all these techniques in this section.

PROS AND CONS OF HASHING

- Advantage of hashing is that no extra space is required to store the index as in case of other data structures. In addition, a hash table provides fast data access and an added advantage of rapid updates.

- On the other hand, the primary drawback of using hashing technique for inserting and retrieving data values is that it usually lacks locality and sequential retrieval by key. This makes insertion and retrieval of data values even more random.

- All the more choosing an effective hash function is more an art than a science. It is not uncommon to (in open-addressed hash tables) to create a poor hash function.

APPLICATIONS OF HASHING

- Hash tables are widely used in situations where enormous amounts of data have to be accessed to quickly search and retrieve information. A few typical examples where hashing is used are given below.

- Hashing is used for database indexing. Some DBMSs store a separate file known as indexes. When data has to be retrieved from a file, the key information is first found in the appropriate index file which references the exact record location of the data in the database file. This key information in the index file is often stored as a hashed value.

- Hashing is used as symbol tables, for example, in Fortran language to store variable names. Hash tables speeds up execution of the program as the references to variables can be looked up quickly.

APPLICATIONS OF HASHING

- In many database systems, File and Directory hashing is used in high performance file systems. Such systems use two complementary techniques to improve the performance of file access. While one of these techniques is caching which saves information in memory, the other is hashing which makes looking up the file location in memory much quicker than most other methods.
- Hash tables can be used to store massive amount of information for example, to store driver's license records. Given the driver's license number, hash tables help to quickly get information about the driver (i.e. name, address, age)
- Hashing technique is for compiler symbol tables in C++. The compiler uses a symbol table to keep a record of the user-defined symbols in a C++ program. Hashing facilitates the compiler to quickly look up variable names and other attributes associated with symbols .

Open Addressing (Probing)

Open addressing / probing is carried out for insertion into fixed size hash tables (hash tables with 1 or more buckets). If the index given by the hash function is occupied, then increment the table position by some number.

There are three schemes commonly used for probing:

1. Linear Probing: The linear probing algorithm is detailed below:

```
Index := hash(key)
While Table(Index) Is Full do
index := (index + 1) MOD Table_Size
if (index = hash(key))
return table_full
else
Table(Index) := Entry
```

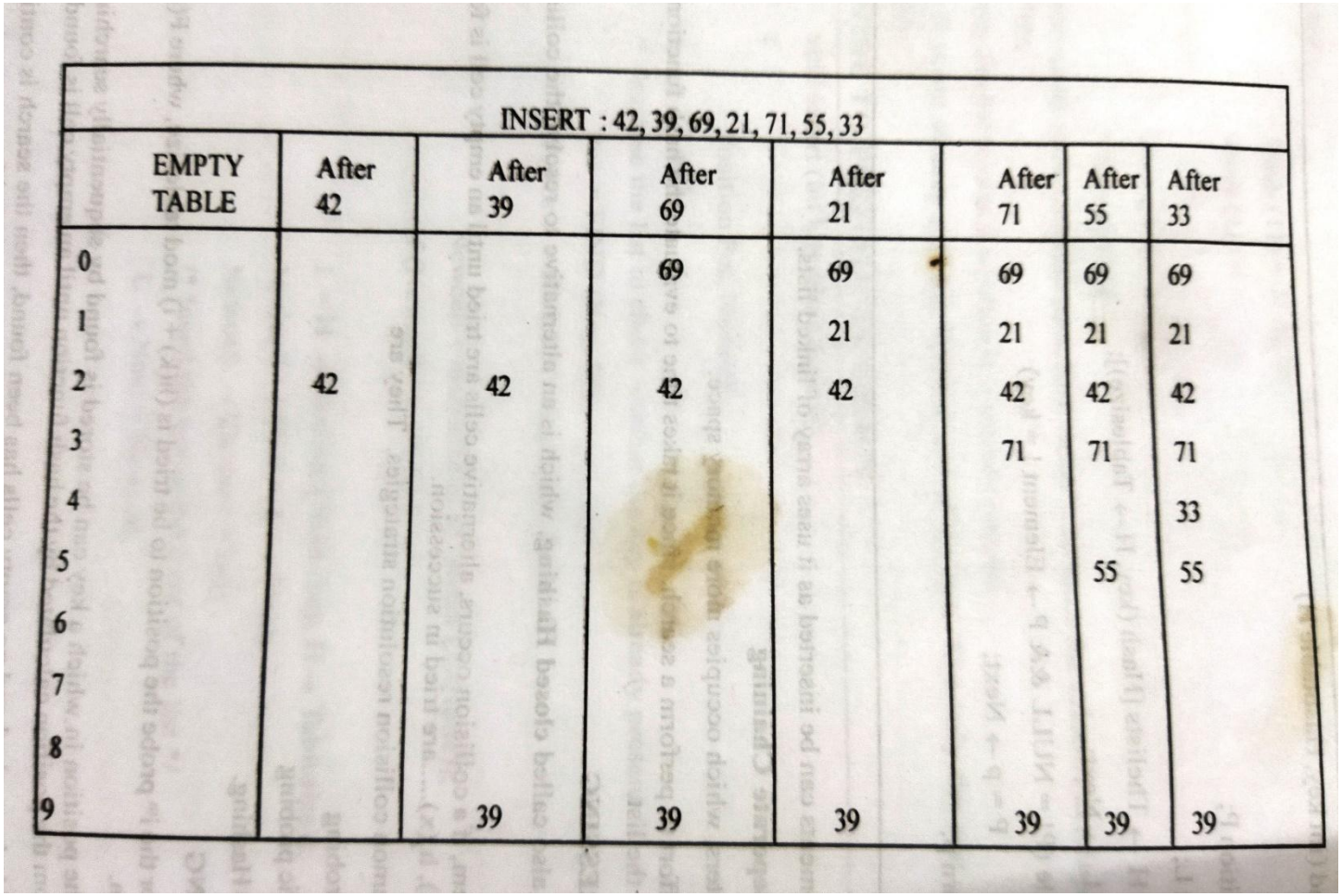
Formula : (H(k)+i) MOD Table_Size

2. Quadratic Probing: increment the position computed by the hash function in quadratic fashion i.e. increment by 1, 4, 9, 16, ….

3. Double Hash: compute the index as a function of two different hash functions.

2. Open Addressing – Linear Probing

: (H(k)+i) MOD Table_Size



INSERT : 42, 39, 69, 21, 71, 55, 33

	EMPTY TABLE	After 42	After 39	After 69	After 21	After 71	After 55	After 33
0				69	69	69	69	69
1					21	21	21	21
2		42	42	42	42	42	42	42
3						71	71	71
4								33
5							55	55
6								
7								
8								
9			39	39	39	39	39	39

2. Open Addressing – Linear Probing

Challenges in Linear Probing :

Primary Clustering: One of the problems with linear probing is Primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search for an element.

Secondary Clustering*:* Secondary clustering is less severe, two records only have the same collision chain (Probe Sequence) if their initial position is the same.

2. Open Addressing – Quadratic Probing

- Quadratic probing eliminates primary clusters.
- This method is also known as the **mid-square** method
- In this method, we look for the **i^2'th** slot in the **i^{th}** iteration.
- We always start from the original hash location. If only the location is occupied then we check the other slots.

let hash(x) be the slot index computed using hash function.

*If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S*
*If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S*
*If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S*

Properties of quadratic hashing

- Hashtable size should not be an even number; otherwise Property 2 will not be satisfied.
- Ideally, table size should be a prime of the form 4j+3,where j is an integer. This choice of table size guarantees Property 2.

Example:Quadratic Probing

✓Load the keys 23, 13, 21, 14, 7, 8, and 15, in this order, in a hash table of size 7 using quadratic probing with c(i) = ± i^2 and the hash function:

h(key) = key % 7

✓The required probe sequences are given by:

h_i (key) = (h(key) ± i^2) % 7 i = 0, 1, 2, 3

Computations - keys 23, 13, 21, 14, 7, 8, and 15

$hi(key) = (h(key) \pm i^2) \% 7 \quad i = 0, 1, 2, 3$

h0(23) = (23 % 7) % 7 = 2
h0(13) = (13 % 7) % 7 = 6
h0(21) = (21 % 7) % 7 = 0

h0(14) = (14 % 7) % 7 = 0 collision
h1(14) = (0 + 1^2) % 7 = 1

h0(7) = (7 % 7) % 7 = 0 collision
h1(7) = (0 + 1^2) % 7 = 1 collision
h2(7) = (0 + 2^2) % 7 = 4

h0(8) = (8 % 7)%7 = 1 collision
h1(8) = (1 + 1^2) % 7 = 2 collision
h2(8) = (1 + 2^2) % 7 = 5

h0(15) = (15 % 7)%7 = 1 collision
h1(15) = (1 + 1^2) % 7 = 2 collision
h2(15) = (1 + 2^2) % 7 = 5 collision
h2(15) = (1 + 3^2) % 7 = 3

0	O	21
1	O	14
2	O	23
3	O	15
4	O	7
5	O	8
6	O	13

Quadratic probing is better than linear probing because it eliminates primary clustering.

2. Open Addressing – Double Hashing

- Double hashing achieves this by having two hash functions that both depend on the hash key.
- In this technique, the increments for the probing sequence are computed by using another hash function. We use another hash function hash2(x) and look for the i*hash2(x) slot in the ith rotation.
- The probing sequence is:
- *Double hashing can be done using :*
  (hash1(key) + i * hash2(key)) % TABLE_SIZE
- *let hash(x) be the slot index computed using hash function.*
- *If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S*
  *If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S*
  *If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S*

Performance and example of Double hashing

Much better than linear or quadratic probing because it eliminates both primary and secondary clustering. BUT requires a computation of a second hash function h_p.

Example: Load the keys 18, 26, 35, 9, 64, 47, 96, 36, and 70 in this order, in an empty hash table of size 13

(a) using double hashing with the first hash function: h(key) = key % 13 and the second hash function: h_p (key) = 1 + key % 12

(b) using double hashing with the first hash function: h(key) = key % 13 and the second hash function: h_p (key) = 7 – key % 7

Computation

hi(key) = [h(key) + i*hp(key)]% 13
h(key) = key % 13
hp(key) = 1 + key % 12

h0(18) = (18%13)%13 = 5
h0(26) = (26%13)%13 = 0
h0(35) = (35%13)%13 = 9

h0(9) = (9%13)%13 = 9 collision
hp(9) = 1 + 9%12 = 10
h1(9) = (9 + 1*10)%13 = 6

h0(64) = (64%13)%13 = 12
h0(47) = (47%13)%13 = 8

h0(96) = (96%13)%13 = 5 collision
hp(96) = 1 + 96%12 = 1
h1(96) = (5 + 1*1)%13 = 6 collision
h2(96) = (5 + 2*1)%13 = 7

h0(36) = (36%13)%13 = 10

h0(70) = (70%13)%13 = 5 collision
hp(70) = 1 + 70%12 = 11
h1(70) = (5 + 1*11)%13 = 3

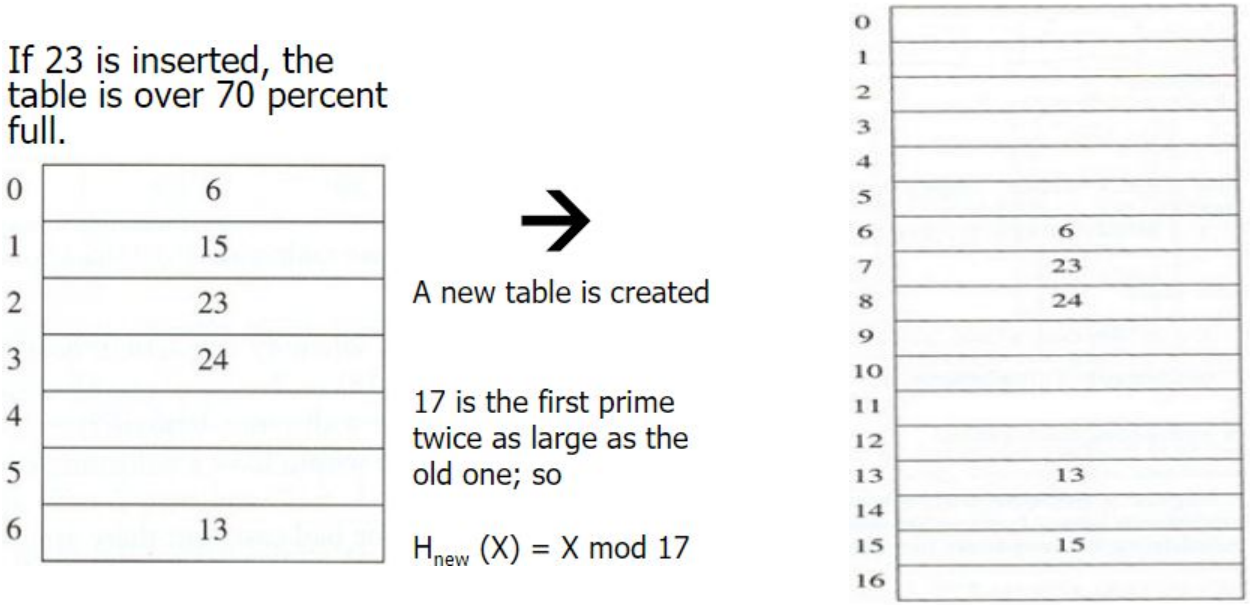
0	1	2	3	4	5	6	7	8	9	10	11	12
26			70		18	9	96	47	35	36		64

Rehashing

- Rehashing is with respect to closed hashing. When we try to store the record with Key1 at bucket Hash(Key1) position and find that it already holds a record, it is collision situation

- If the table gets too full, then the rehashing method builds a new table that is about twice as big and scan down the entire original hash table, computing the new hash value for each element and inserting it in the new table.

- Very Expensive, Running time is O(N)



Extendible hashing

- Extendible hashing is based on a radix-2 tree. The idea is to hash the key, yielding a long two-bit number. Then, use as many bits as desired to create a radix-2 tree.
- But, instead of building a tree, just collapse the tree, interpreting the 0/1 sequence along each branch as a binary number, used as the index into the bucket array.
- When collision occurs, more bits can be used to divide the buckets into a larger (by powers of 2) address space.

Illustration of Extendible hashing

An empty Extendible Hash Table w/One Bucket and an inital address space of 2 bits.

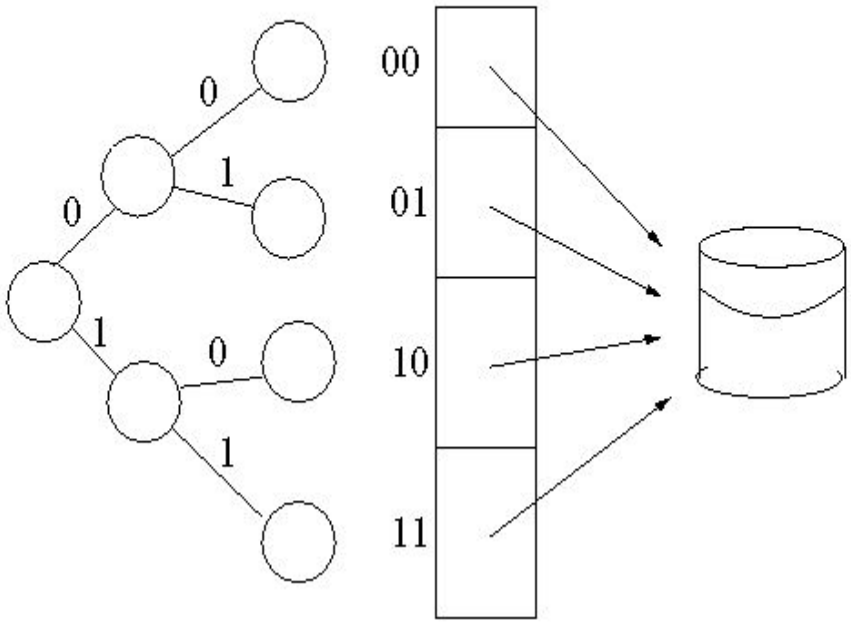


Illustration of Extendible hashing

The address space splits in response to the addition of two keys, one with prefix 00, the other 01

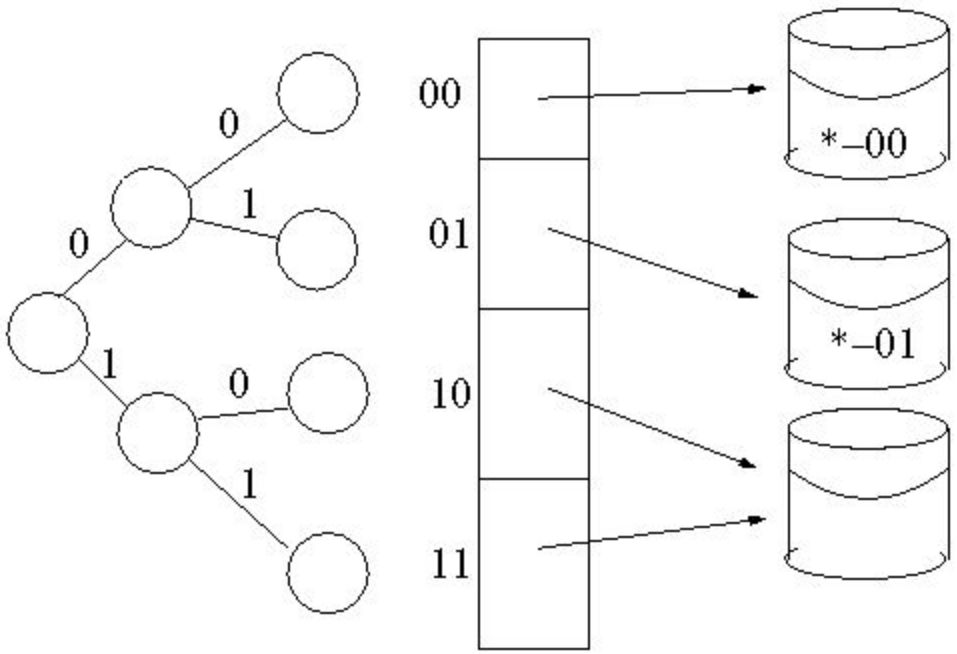
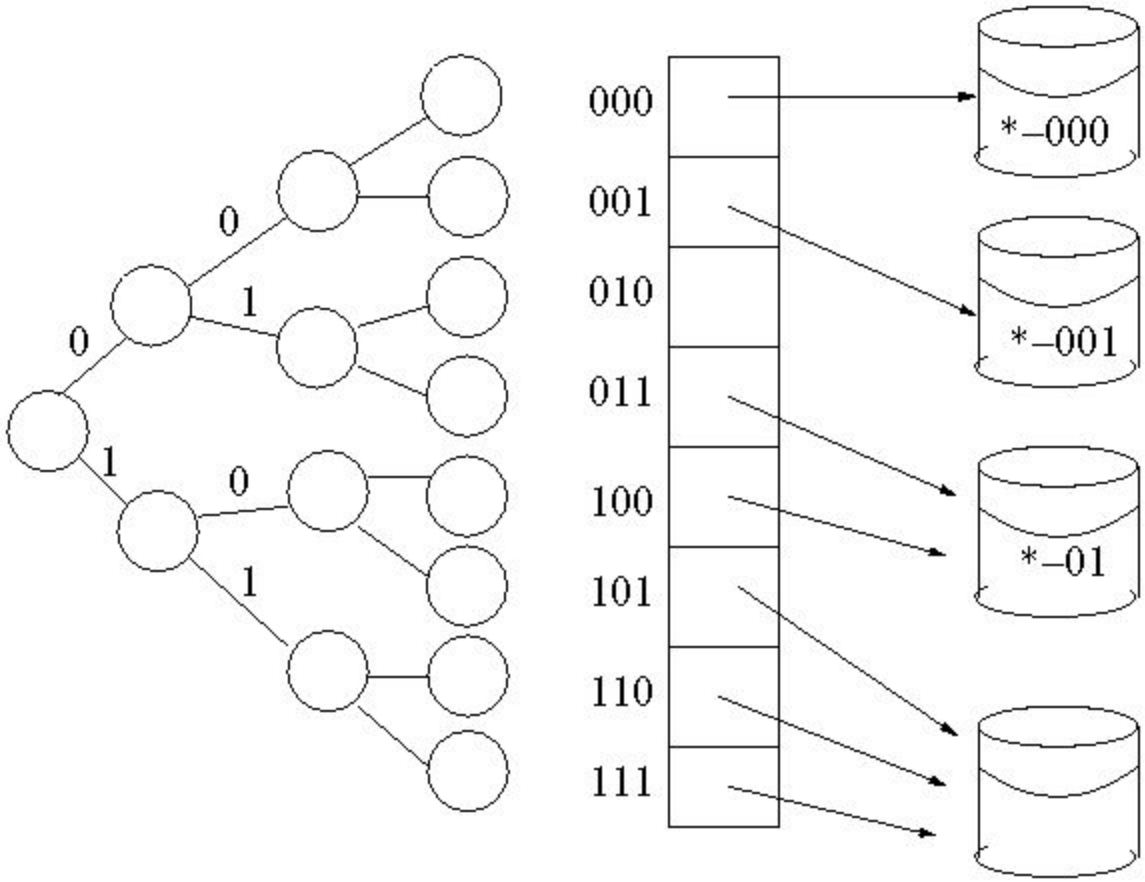


Illustration of Extendible hashing

The address space splits again, this time keys need to be distinguished by three bits: 000, 001, and 010



References

*Reema Thareja, “Data Structures Using C”, Oxford Higher Education, First Edition,
2011.

A.V.Aho, J.E Hopcroft , J.D.Ullman, Data structures and Algorithms, Pearson Education, 2003

*https://www.slideshare.net/HanifDurad/chapter-12-ds

*https://www.slideshare.net/sumitbardhan/358-33-powerpointslides15hashingcollisionchapter15

*andrew.cmu.edu/course/15-310/applications/ln/lecture12.html