

## UNIT V

### GRAPHS

Graph Terminology, Graph Traversal, Topological sorting - Minimum spanning tree – Prims - Kruskals - Network flow problem - Shortest Path Algorithm: Dijkstra - Graph Search: Depth First Search, Breadth First Search - Hashing: Hash functions, Collision avoidance, Separate chaining - Open addressing: Linear probing, Quadratic Probing, Double hashing, Rehashing, Extensible Hashing

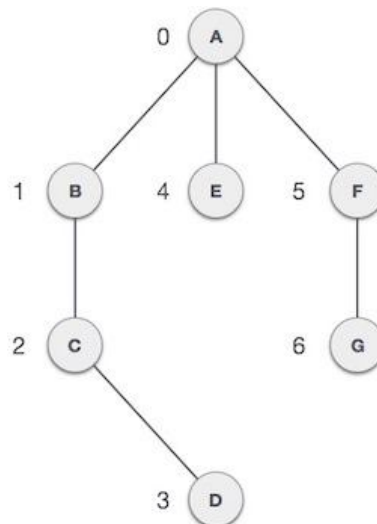
---

### GRAPHS

#### Graph Terminology

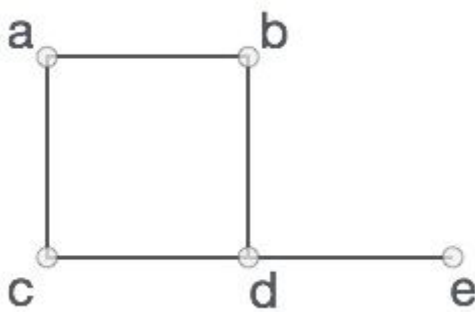
Mathematical graphs can be represented in data-structure. We can represent a graph using an array of vertices and a two dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms –

- **Vertex** – Each node of the graph is represented as a vertex. In example given below, labeled circle represents vertices. So A to G are vertices. We can represent them using an array as shown in image below. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** – Edge represents a path between two vertices or a line between two vertices. In example given below, lines from A to B, B to C and so on represents edges. We can use a two dimensional array to represent array as shown in image below. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- **Adjacency** – Two node or vertices are adjacent if they are connected to each other through an edge. In example given below, B is adjacent to A, C is adjacent to B and so on.
- **Path** – Path represents a sequence of edges between two vertices. In example given below, ABCD represents a path from A to D.



A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.

Formally, a graph is a pair of sets  $(V, E)$ , where  $V$  is the set of vertices and  $E$  is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



In the above graph,

$$V = \{a, b, c, d, e\}$$

$$E = \{ab, ac, bd, cd, de\}$$

## Basic Operations

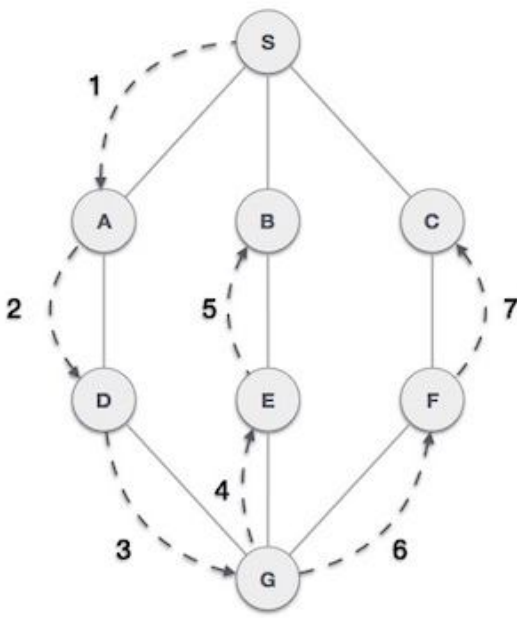
Following are basic primary operations of a Graph which are following.

- **Add Vertex** – add a vertex to a graph.
- **Add Edge** – add an edge between two vertices of a graph.
- **Display Vertex** – display a vertex of a graph.

## Graph Traversal

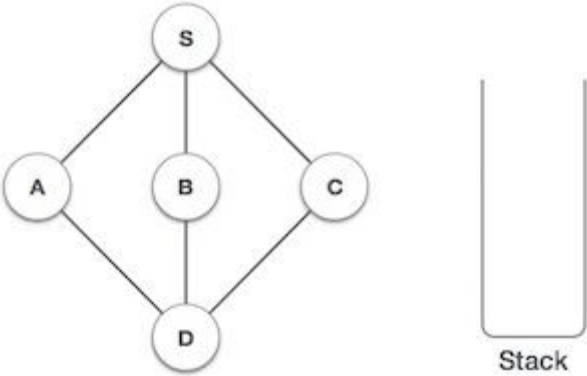
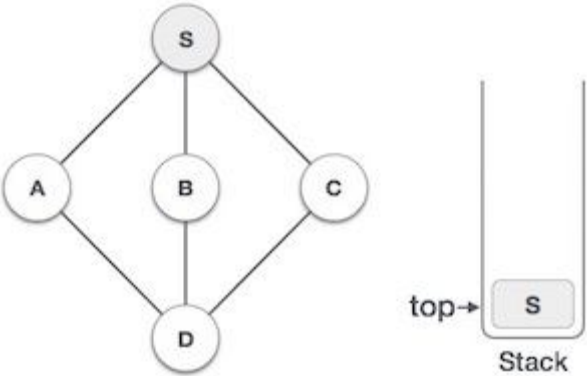
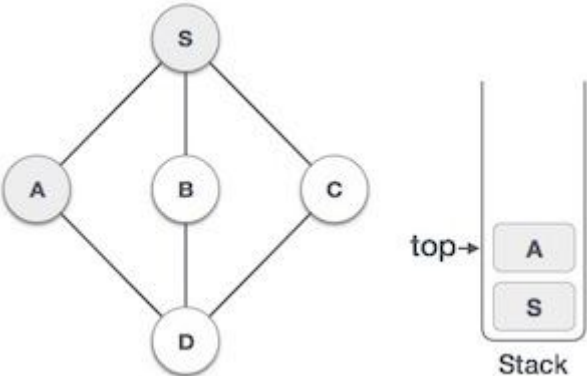
### 1. Depth First Traversal

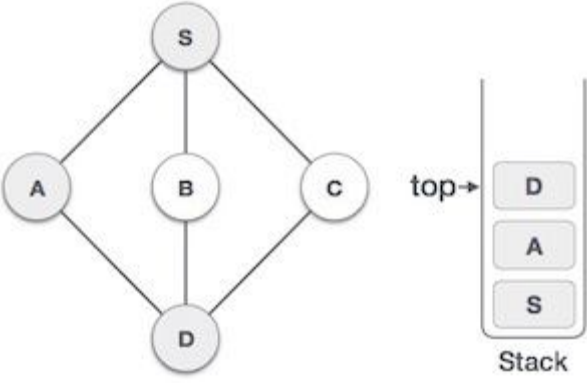
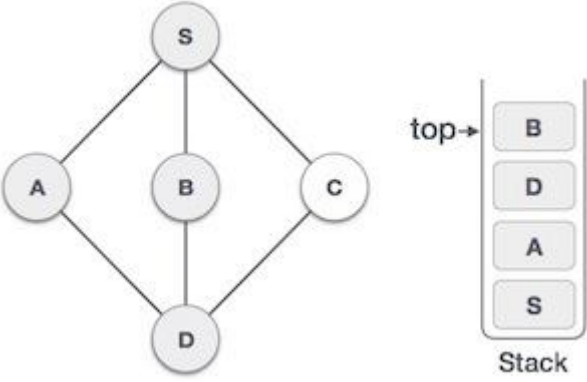
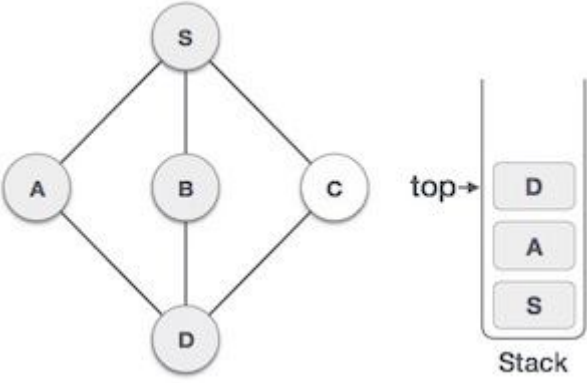
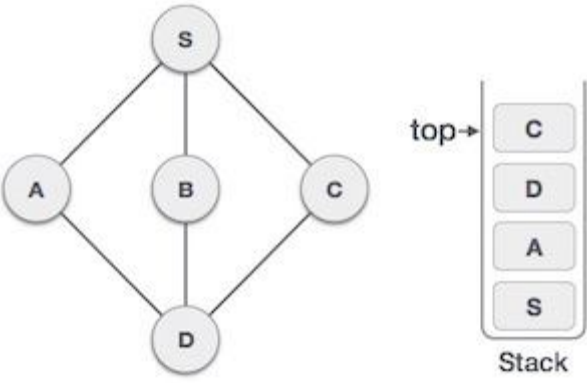
Depth First Search algorithm(DFS) traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.



As in example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. It employs following rules.

- **Rule 1** – Visit adjacent unvisited vertex. Mark it visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex found, pop up a vertex from stack. (It will pop up all the vertices from the stack which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until stack is empty.

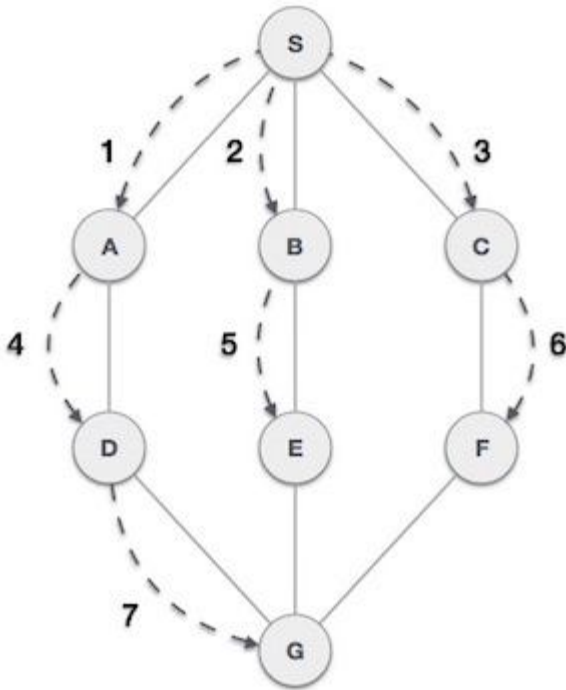
Step	Traversal	Description
1.		Initialize the stack
2.		Mark <b>S</b> as visited and put it onto the stack. Explore any unvisited adjacent node from <b>S</b> . We have three nodes and we can pick any of them. For this example, we shall take the node in alphabetical order.
3.		Mark <b>A</b> as visited and put it onto the stack. Explore any unvisited adjacent node from <b>A</b> . Both <b>S</b> and <b>D</b> are adjacent to <b>A</b> but we are concerned for unvisited nodes only.

4.		<p>Visit <b>D</b> and mark it visited and put onto the stack. Here we have <b>B</b> and <b>C</b> nodes which are adjacent to <b>D</b> and both are unvisited. But we shall again choose in alphabetical order.</p>
5.		<p>We choose <b>B</b>, mark it visited and put onto stack. Here <b>B</b> does not have any unvisited adjacent node. So we pop <b>B</b> from the stack.</p>
6.		<p>We check stack top for return to previous node and check if it has any unvisited nodes. Here, we find <b>D</b> to be on the top of stack.</p>
7.		<p>Only unvisited adjacent node is from <b>D</b> is <b>C</b> now. So we visit <b>C</b>, mark it visited and put it onto the stack.</p>

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node which has unvisited adjacent node. In this case, there's none and we keep popping until stack is empty.

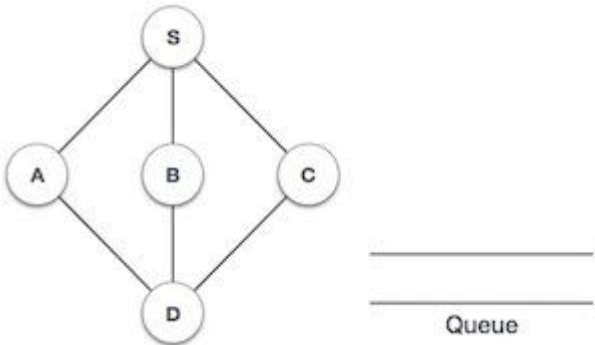
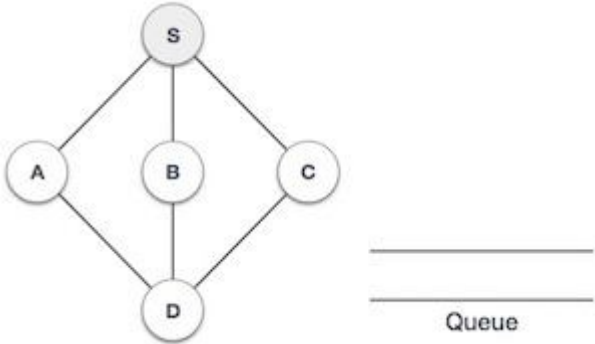
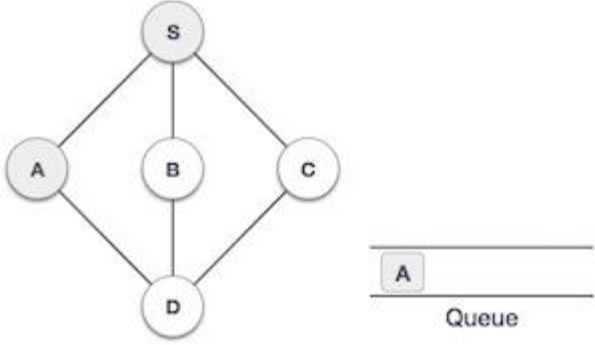
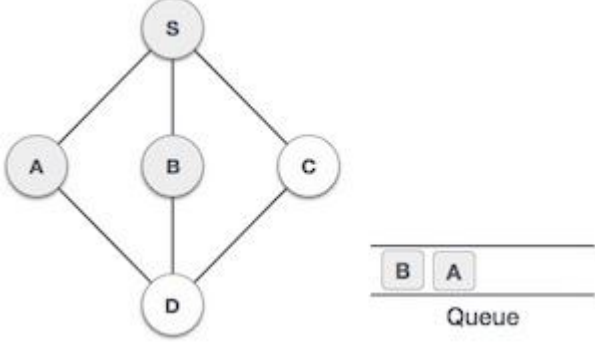
## 2. Breadth First Traversal

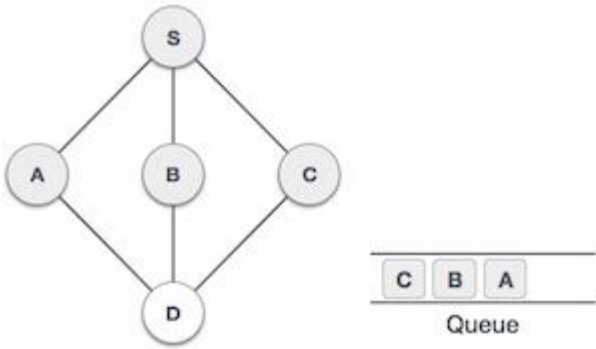
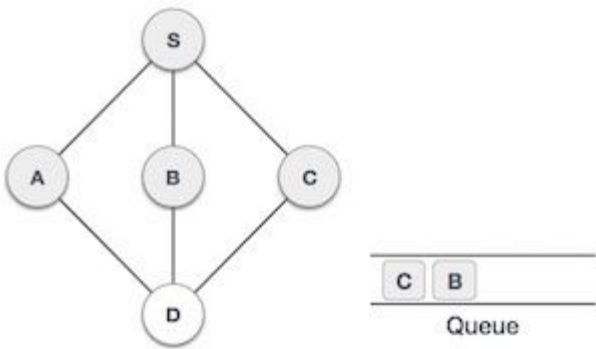
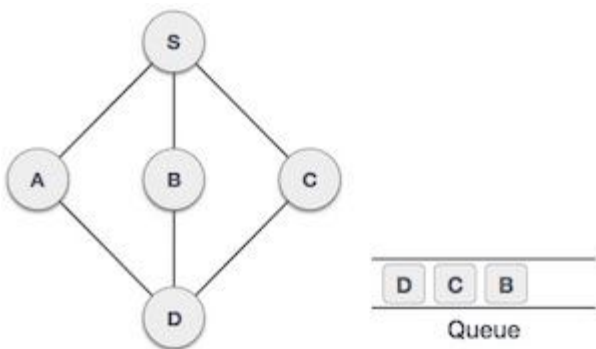
Breadth First Search algorithm(BFS) traverses a graph in a breadth wards motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.



As in example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs following rules.

- **Rule 1** – Visit adjacent unvisited vertex. Mark it visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex found, remove the first vertex from queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until queue is empty.

Step	Traversal	Description
1.		Initialize the queue.
2.		We start from visiting <b>S</b> (starting node), and mark it visited.
3.		We then see unvisited adjacent node from <b>S</b> . In this example, we have three nodes but alphabetically we choose <b>A</b> mark it visited and enqueue it.
4.		Next unvisited adjacent node from <b>S</b> is <b>B</b> . We mark it visited and enqueue it.

5.		<p>Next unvisited adjacent node from <b>S</b> is <b>C</b>. We mark it visited and enqueue it.</p>
6.		<p>Now <b>S</b> is left with no unvisited adjacent nodes. So we dequeue and find <b>A</b>.</p>
7.		<p>From <b>A</b> we have <b>D</b> as unvisited adjacent node. We mark it visited and enqueue it.</p>

At this stage we are left with no unmarked (unvisited) nodes. But as per algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied the program is over.

## Topological sort

**Topological sort:** an ordering of the vertices in a directed acyclic graph, such that:

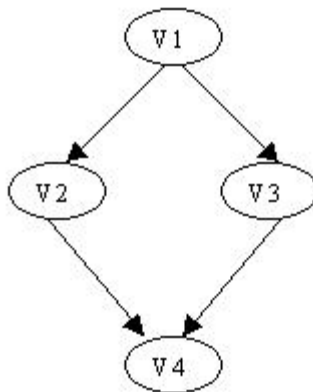
If there is a path from **u** to **v**, then **v** appears after **u** in the ordering.



**Types of graphs:**

The graphs should be **directed**: otherwise for any edge  $(u,v)$  there would be a path from  $u$  to  $v$  and also from  $v$  to  $u$ , and hence they cannot be ordered.

The graphs should be **acyclic**: otherwise for any two vertices  $u$  and  $v$  on a cycle  $u$  would precede  $v$  and  $v$  would precede  $u$ . The ordering may not be unique:



$V1, V2, V3, V4$  and  $V1, V3, V2, V4$  are legal orderings

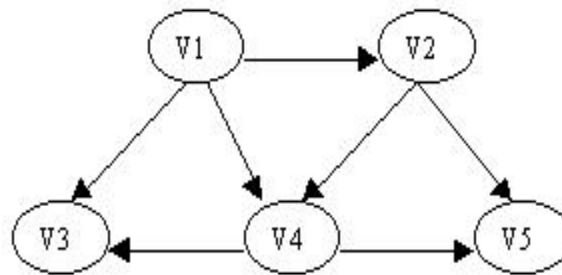
**Degree** of a vertex  $U$ : the number of edges  $(U,V)$  - outgoing edges

**Indegree** of a vertex  $U$ : the number of edges  $(V,U)$  - incoming edges

The algorithm for topological sort uses "indegrees" of vertices.

**Algorithm**

1. Compute the indegrees of all vertices
2. Find a vertex  $U$  with indegree 0 and print it (store it in the ordering)  
If there is no such vertex then there is a cycle  
and the vertices cannot be ordered. Stop.
3. Remove  $U$  and all its edges  $(U,V)$  from the graph.
4. Update the indegrees of the remaining vertices.
5. Repeat steps 2 through 4 while there are vertices to be processed.

**Example**

1. Compute the indegrees: V1: 0      V2: 1      V3: 2      V4: 2      V5: 2
2. Find a vertex with indegree 0: V1
3. Output V1 , remove V1 and update the indegrees:

Sorted: V1

Remove edges: (V1,V2) , (V1, V3) and (V1,V4)

Updated indegrees: V2: 0      V3: 1      V4: 1      V5: 2

The process is depicted in the following table:

	Indegree					
Sorted		V1	V1,V2	V1,V2,V4	V1,V2,V4,V3	V1,V2,V4,V3,V5
V1	0					
V2	1	0				
V3	2	1	1	0		
V4	2	1	0			
V5	2	2	1	0	0	

One possible sorting: V1, V2, V4, V3, V5

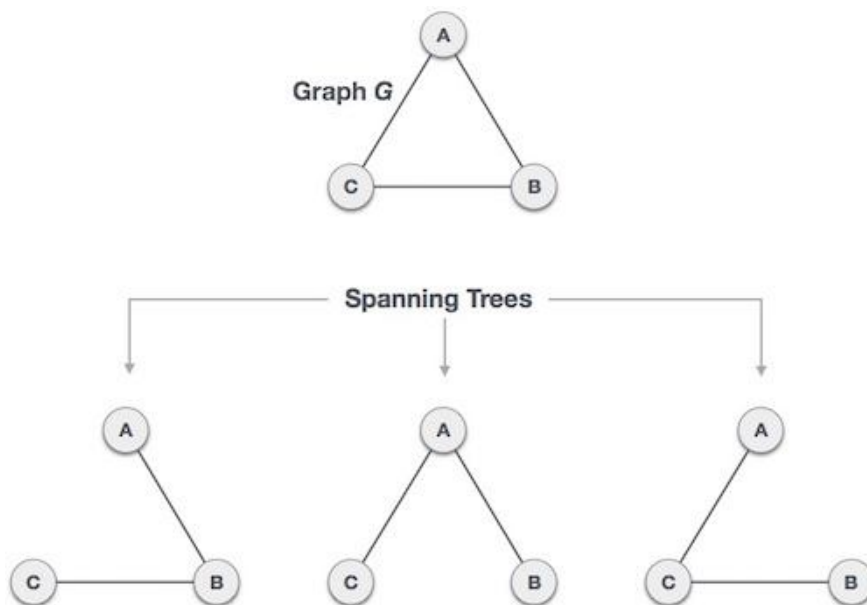
Another sorting: V1, V2, V4, V5, V3

**Complexity** of this algorithm:  $O(|V|^2)$ ,  $|V|$  - the number of vertices. To find a vertex of indegree 0 we scan all the vertices -  $|V|$  operations. We do this for all vertices:  $|V|^2$

## Minimum 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 can not be disconnected. By this definition we can draw a conclusion that every connected & undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it can not spanned to all its vertices.

We found three spanning trees off one complete graph. A complete undirected graph can have maximum  $n^{n-2}$  number of spanning trees, where n is number of nodes. In addressed example, n is 3, hence  $3^{3-2} = 3$  spanning trees are possible.



### General properties of spanning tree

We now understand that one graph can have more than one spanning trees. Below are few properties is spanning tree of given connected graph G –

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have same number of edges and vertices.
- Spanning tree does not have any cycle (loops)
- Removing one edge from spanning tree will make the graph disconnected i.e. spanning tree is **minimally connected**.
- Adding one edge to a spanning tree will create a circuit or loop i.e. spanning tree is **maximally acyclic**.

### Mathematical properties of spanning tree

- Spanning tree has  $n-1$  edges, where  $n$  is number of nodes (vertices)
- From a complete graph, by removing maximum  $e-n+1$  edges, we can construct a spanning tree.
- A complete graph can have maximum  $n^{n-2}$  number of spanning trees.

So we can conclude here that spanning trees are subset of a connected Graph  $G$  and disconnected Graphs do not have spanning tree.

### Application of Spanning Tree

Spanning tree is basically used to find minimum paths to connect all nodes in a graph. Common application of spanning trees are –

- **Civil Network Planning**
- **Computer Network Routing Protocol**
- **Cluster Analysis**

Lets understand this by a small example. Consider city network as a huge graph and now plan to deploy telephone lines such a way that in minimum lines we can connect to all city nodes. This is where spanning tree comes in the picture.

### Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight that all other spanning trees of the same graph. In real world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

### Minimum Spanning-Tree Algorithm

We shall learn about two most important spanning tree algorithms here –

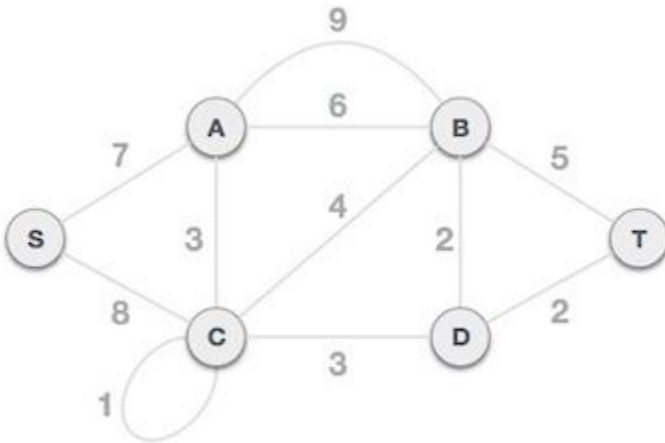
- Kruskal's Algorithm
- Prim's Algorithm

Both are greedy algorithms.

### Kruskal's Algorithm

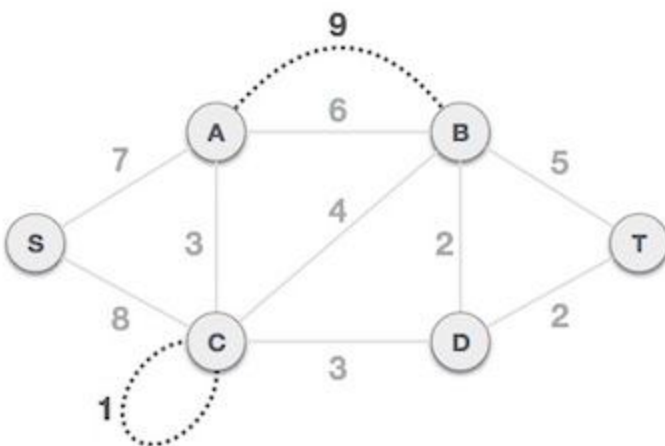
Kruskal's algorithm to find minimum cost spanning tree uses greedy approach. This algorithm treats the graph as a forest and every node it as an individual tree. A tree connects to another only and only if it has least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm we shall take the following example –

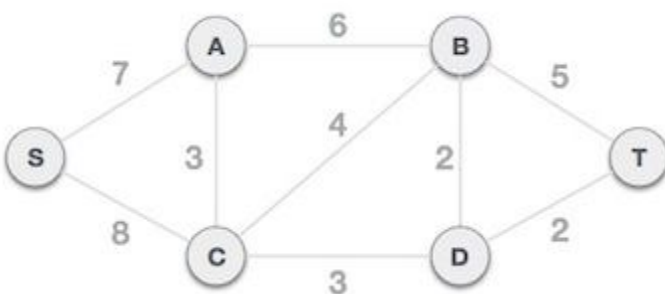


Step 1 - Remove all loops & Parallel Edges

Remove all loops and parallel edges from the given graph.



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



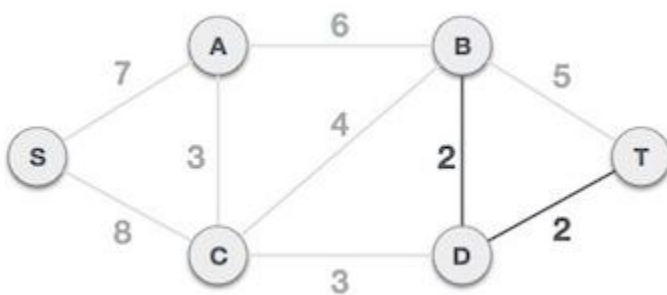
Step 2 - Arrange all edges in their increasing order of weight

Next step is to create a set of edges & weight and arrange them in 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

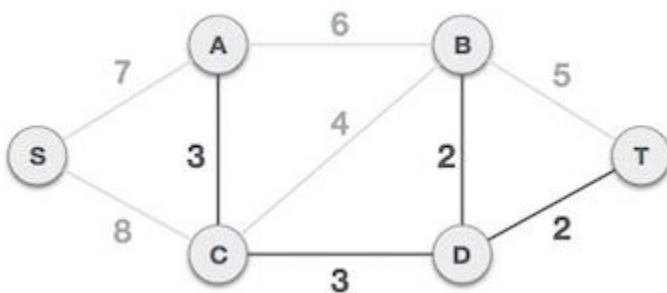
Step 3 - Add the edge which has least weightage

Now we start adding edges to graph beginning from the one which has least weight. At all time, we shall keep checking that the spanning properties are 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 graph.

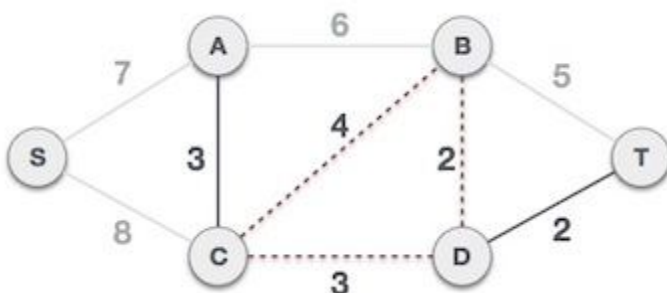


The least cost is 2 and edges involved are B,D and D,T so we add them. Adding them does not violate spanning tree properties so we continue to our next edge selection.

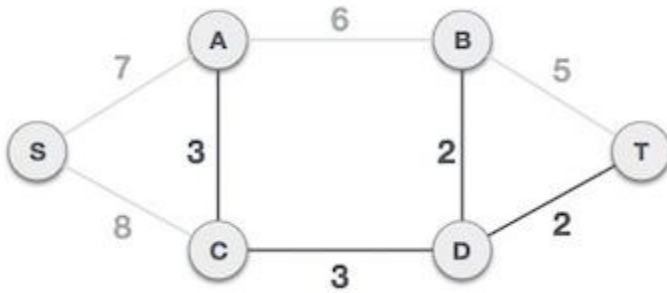
Next cost is 3, and associated edges are A,C and C,D. So we add them –



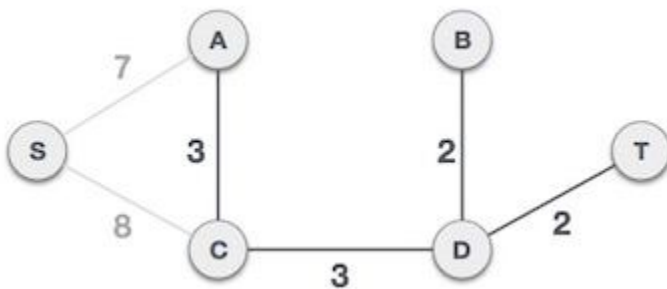
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph



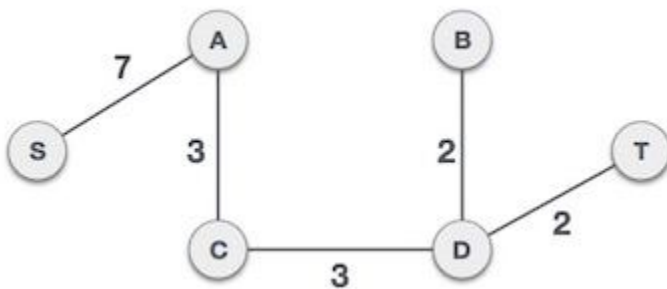
...and we ignore it. And in the process we shall ignore/avoid all edges which create circuit.



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



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



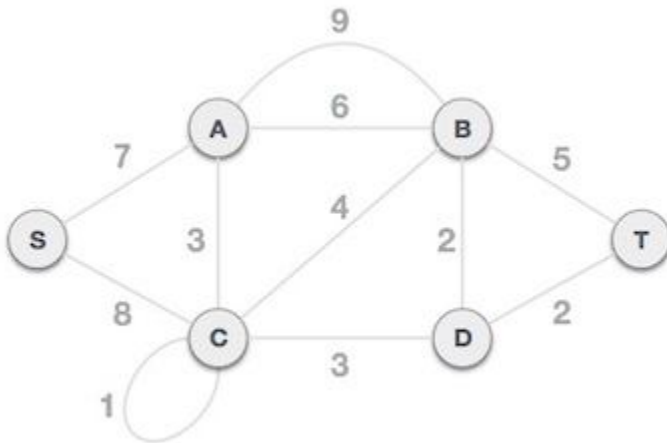
By adding edge S,A we have included all the nodes of the graph and we have minimum cost spanning tree.

### Algorithm

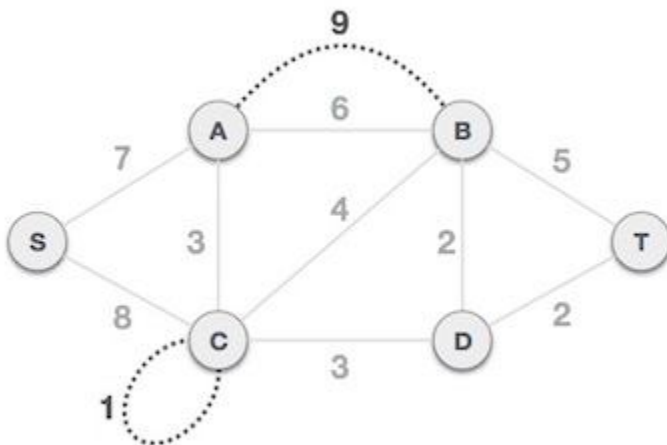
1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far.  
If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are  $(V-1)$  edges in the spanning tree.

### Prims Algorithm

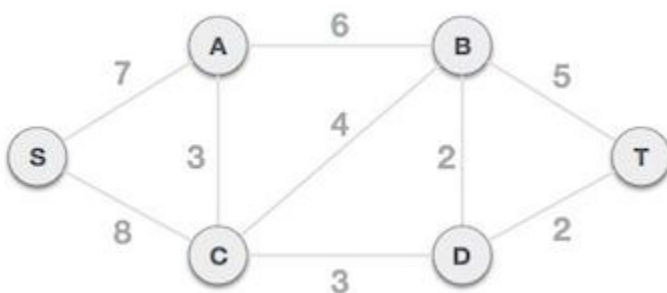
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses greedy approach. Prim's algorithm shares similarity with **shortest path first** algorithms. Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph. To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example –



**Step 1 - Remove all loops & Parallel Edges**



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



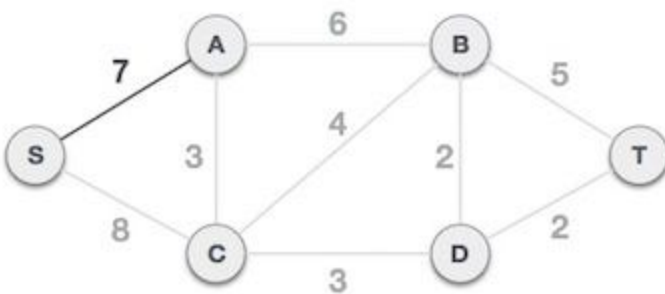
**Step 2 - Choose any arbitrary node as root node**



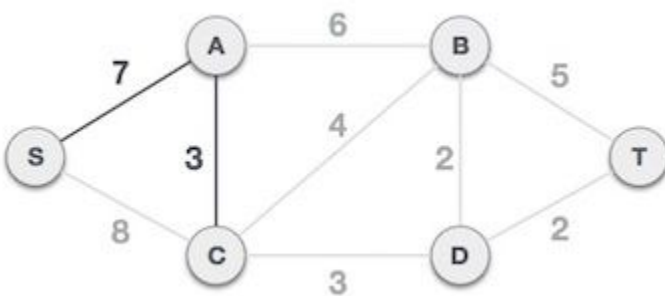
In this case, we choose **S** node as root node of Prim's spanning tree. This node is arbitrarily chosen so any node can be root node. One may wonder why can any video be a root node, so the answer is, in spanning tree all the nodes of a graph are included and because it is connected then there must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

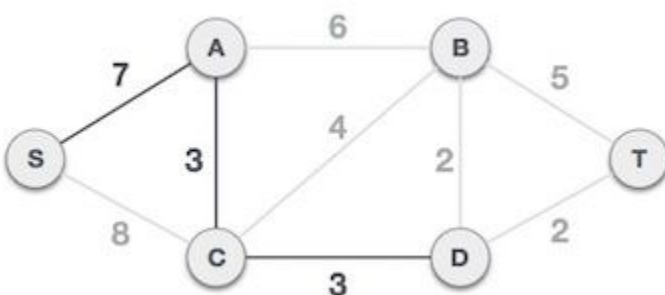
After choosing root node **S**, we see that S,A and S,C are two edges with weight 7 and 8 respectively. And we choose S,A edge as it is lesser than the other.



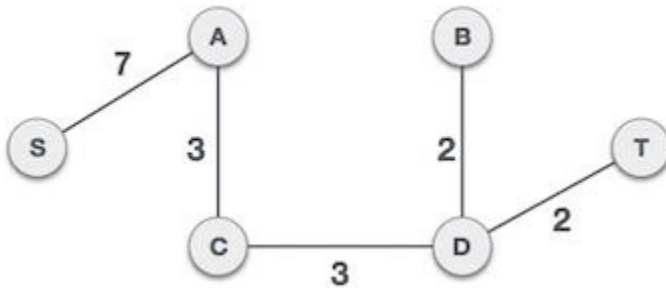
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



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 and will choose only the least cost edge one. Here 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 have same cost, i.e. D-2-T and D-2-B. So we can add either one. But the next step will again yield the edge 2 as the least cost. So here we are showing spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

### Algorithm

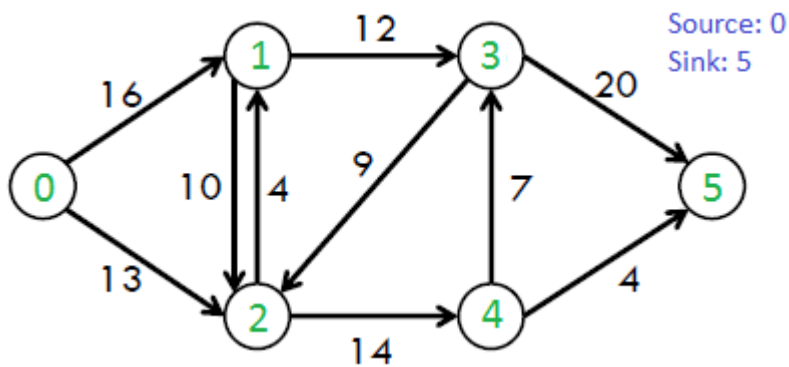
- 1) Create a set mstSet that keeps track of vertices already included in MST.
- 2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
- 3) While mstSet doesn't include all vertices
  - a) Pick a vertex *u* which is not there in mstSet and has minimum key value.
  - b) Include *u* to mstSet.
  - c) Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*.

### Network Flow Problem

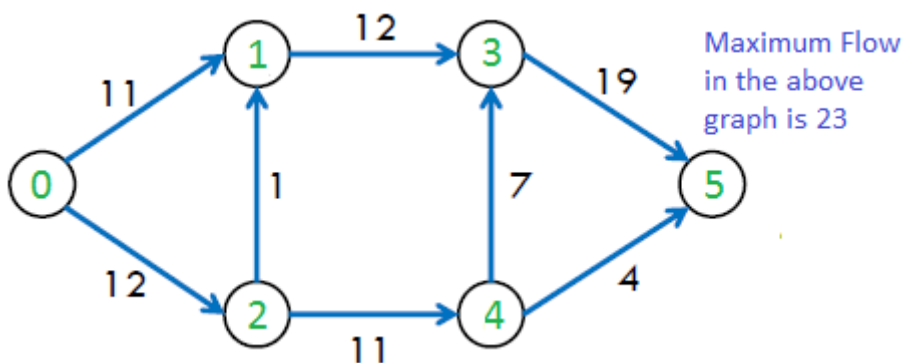
Given a graph which represents a flow network where every edge has a capacity. Also given two vertices *source's'* and *sink 't'* in the graph, find the maximum possible flow from *s* to *t* with following constraints:

- a) **Flow on an edge doesn't exceed the given capacity of the edge.**
- b) **Incoming flow is equal to outgoing flow for every vertex except *s* and *t*.**

For example, consider the following graph from CLRS book.



The maximum possible flow in the above graph is 23.



### Augmenting Path

The idea behind the algorithm is as follows: as long as there is a path from the source (start node) to the sink (end node), with available capacity on all edges in the path, we send flow along one of the paths. Then we find another path, and so on. A path with available capacity is called an augmenting path.

### Ford-Fulkerson Algorithm

The following is simple idea of Ford-Fulkerson algorithm:

- 1) Start with initial flow as 0.
- 2) While there is a augmenting path from source to sink.  
Add this path-flow to flow.
- 3) Return flow.

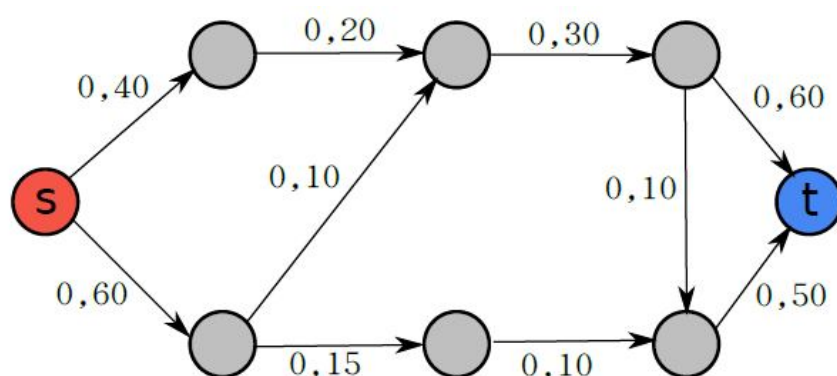
**Example**

Figure 3: A network with zero flow.

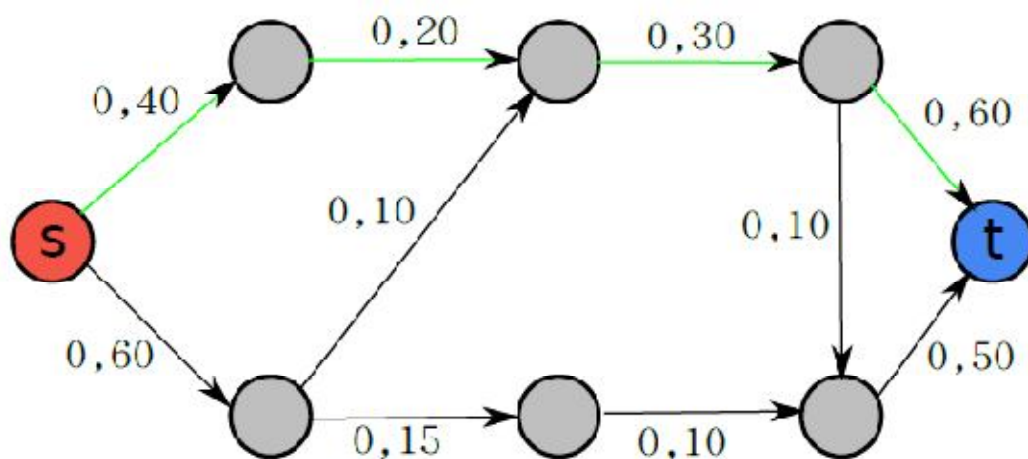


Figure 4: Find an augmenting path.

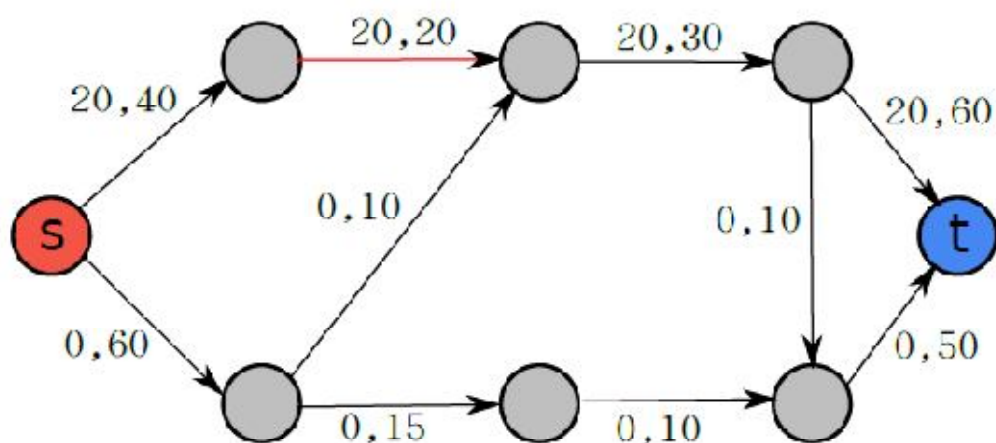


Figure 5: Send flow along the path.

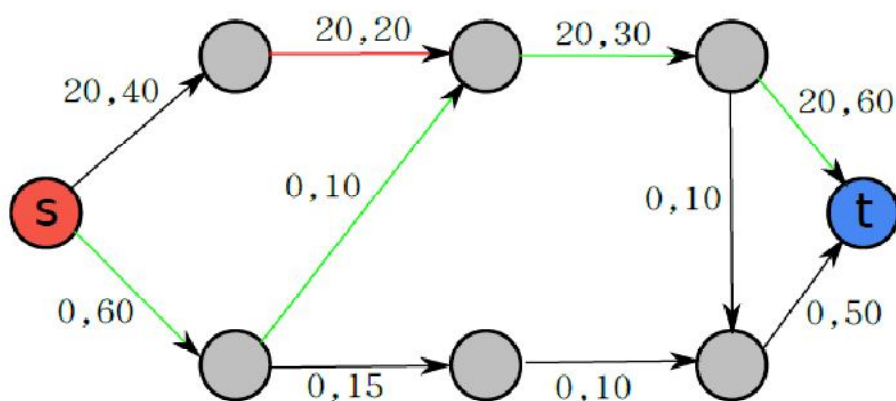


Figure 6: Find an augmenting path.

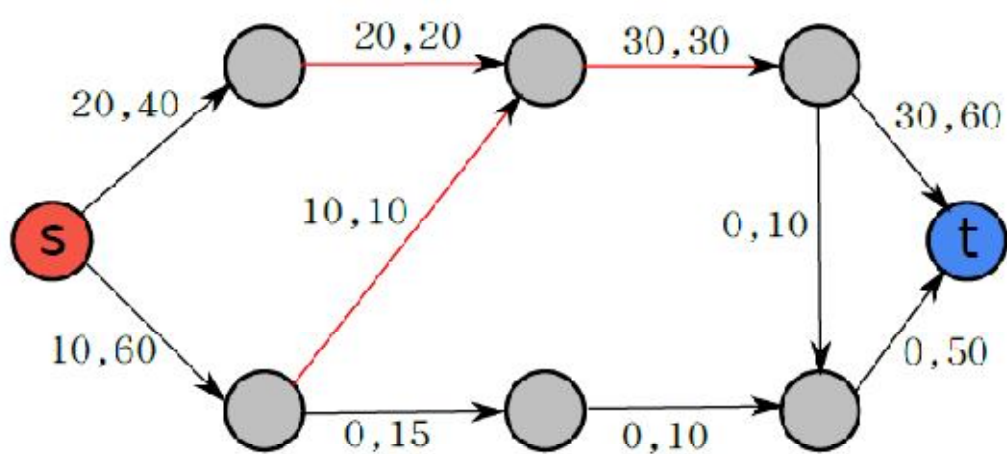


Figure 7: Send flow along the path.

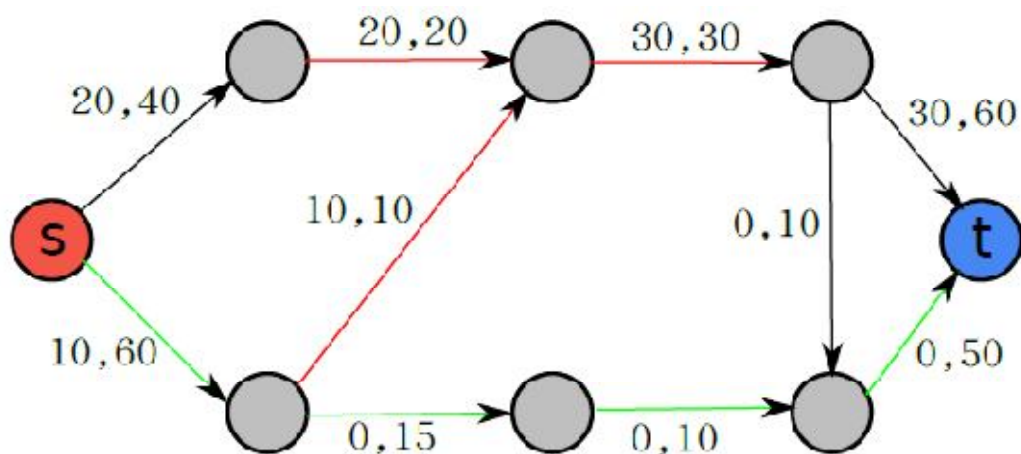


Figure 8: Find an augmenting path.

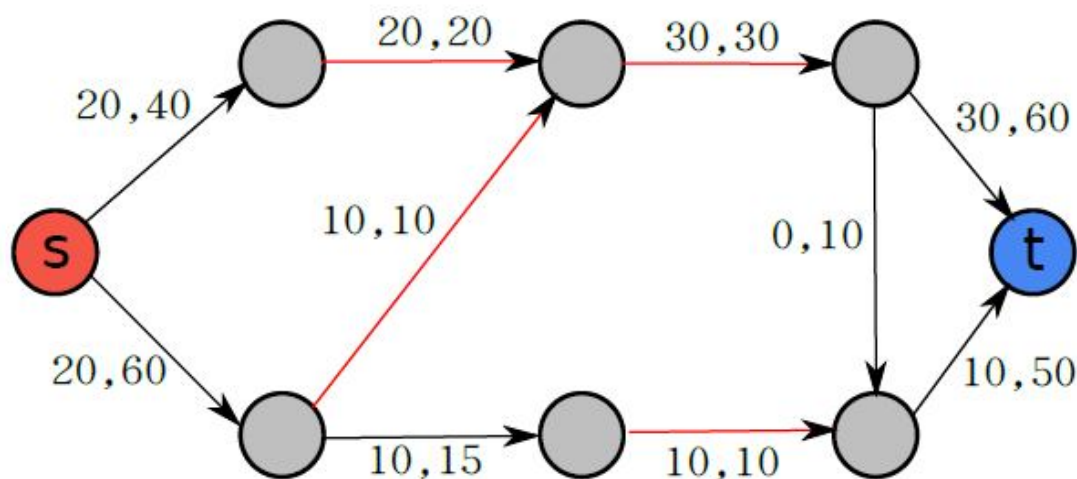


Figure 9: Send flow along the path.

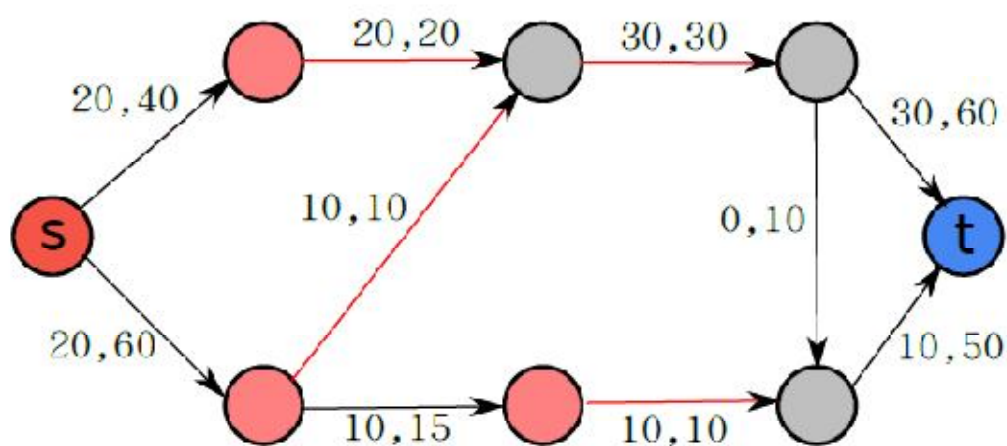


Figure 10: No more augmenting paths can be found. Label all vertices that can be reached via non-saturated edges as "belonging to the source".



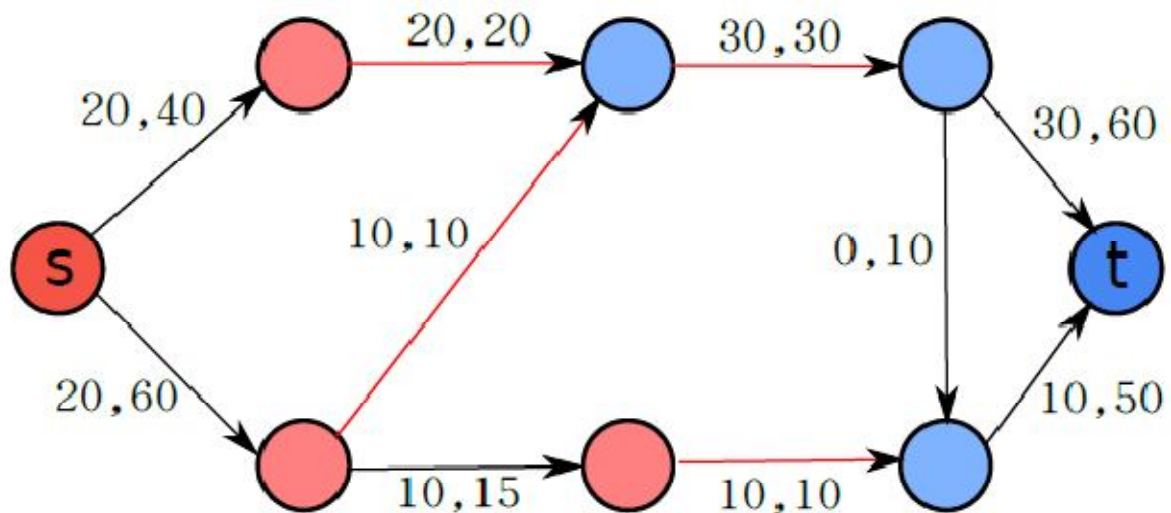


Figure 11: Label all remaining vertices as “belonging to the sink”.

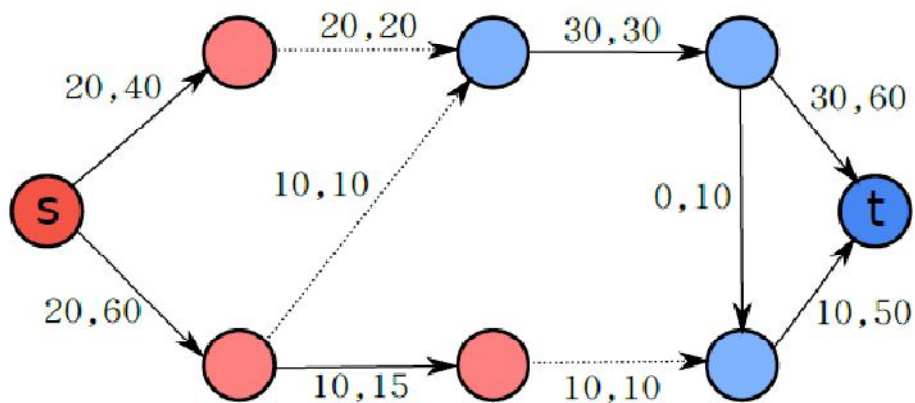


Figure 12: The edges on the boundary of this labeling form a minimum s-t cut.

## Shortest Path Algorithm: Dijkstra

### Objective:

We start with a weighted, directed graph with a weight function  $w$  defined on its edges. The objective is to find the shortest path to every vertices from the start vertex  $S$ .

### Graph Initialization

We will generally consider a distinguished vertex, called the **source**, and usually denoted by  $s$ .

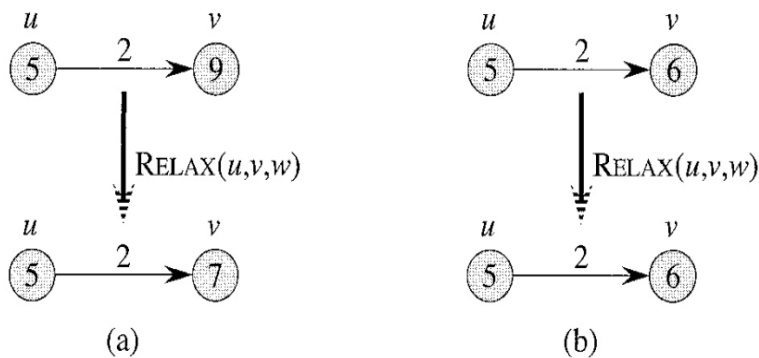
INITIALIZE-SINGLE-SOURCE( $G, s$ )

```

1  for each vertex  $v \in G.V$ 
2       $v.d = \infty$ 
3       $v.\pi = \text{NIL}$ 
4   $s.d = 0$ 

```

## Relaxing an Edge



**Figure 24.3** Relaxing an edge  $(u, v)$  with weight  $w(u, v) = 2$ . The shortest-path estimate of each vertex appears within the vertex. (a) Because  $v.d > u.d + w(u, v)$  prior to relaxation, the value of  $v.d$  decreases. (b) Here,  $v.d \leq u.d + w(u, v)$  before relaxing the edge, and so the relaxation step leaves  $v.d$  unchanged.

RELAX( $u, v, w$ )

```

1  if  $v.d > u.d + w(u, v)$ 
2       $v.d = u.d + w(u, v)$ 
3       $v.\pi = u$ 

```

## Algorithm

DIJKSTRA( $G, w, s$ )

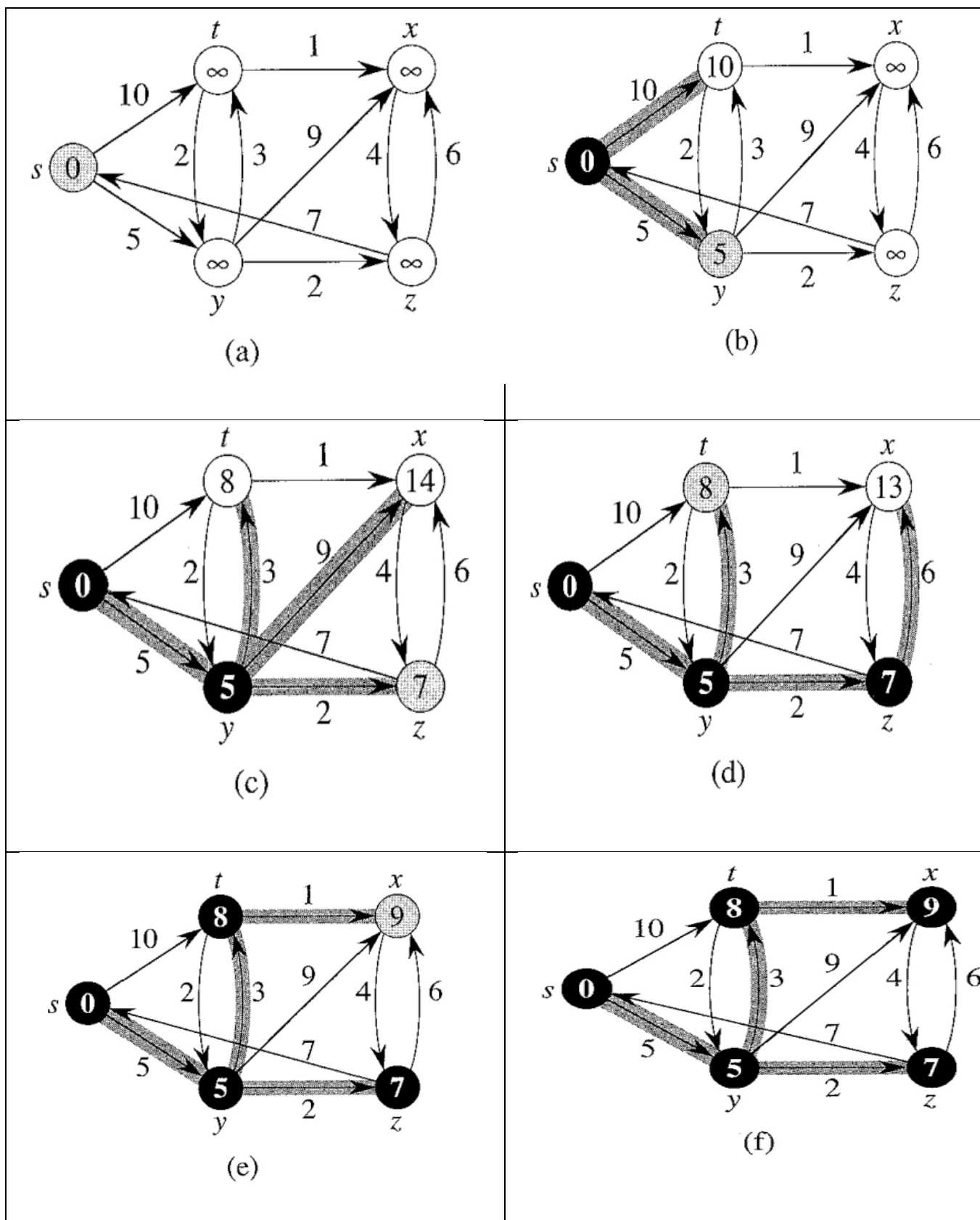
```

1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2   $S = \emptyset$ 
3   $Q = G.V$ 
4  while  $Q \neq \emptyset$ 
5       $u = \text{EXTRACT-MIN}(Q)$ 
6       $S = S \cup \{u\}$ 
7      for each vertex  $v \in G.\text{Adj}[u]$ 
8          RELAX( $u, v, w$ )

```



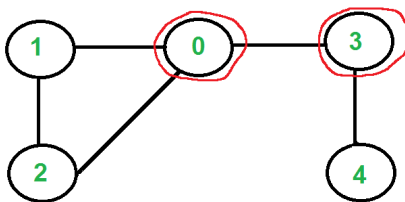
## Example



## Bi-connected components

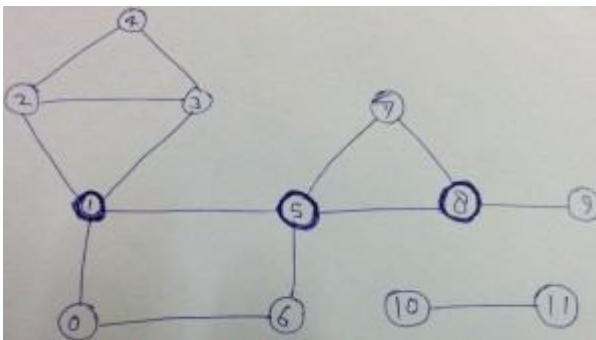
A biconnected component is a maximal biconnected subgraph.

Biconnected Graph: A biconnected graph is a connected graph on two or more vertices having no articulation vertices. A vertex in an undirected connected graph is an articulation point (or cut vertex) iff removing it (and edges through it) disconnects the graph. Articulation points represent vulnerabilities in a connected network – single points whose failure would split the network into 2 or more disconnected components.



Articulation points are 0 and 3

To find biconnected component in a graph using algorithm by John Hopcroft and Robert Tarjan.



In above graph, following are the biconnected components:

- 4-2 3-4 3-1 2-3 1-2
- 8-9
- 8-5 7-8 5-7
- 6-0 5-6 1-5 0-1
- 10-11

Idea is to store visited edges in a stack while DFS on a graph and keep looking for Articulation Points (highlighted in above figure). As soon as an Articulation Point  $u$  is

found, all edges visited while DFS from node  $u$  onwards will form one biconnected component. When DFS completes for one connected component, all edges present in stack will form a biconnected component. If there is no Articulation Point in graph, then graph is biconnected and so there will be one biconnected component which is the graph itself.

## Hashing

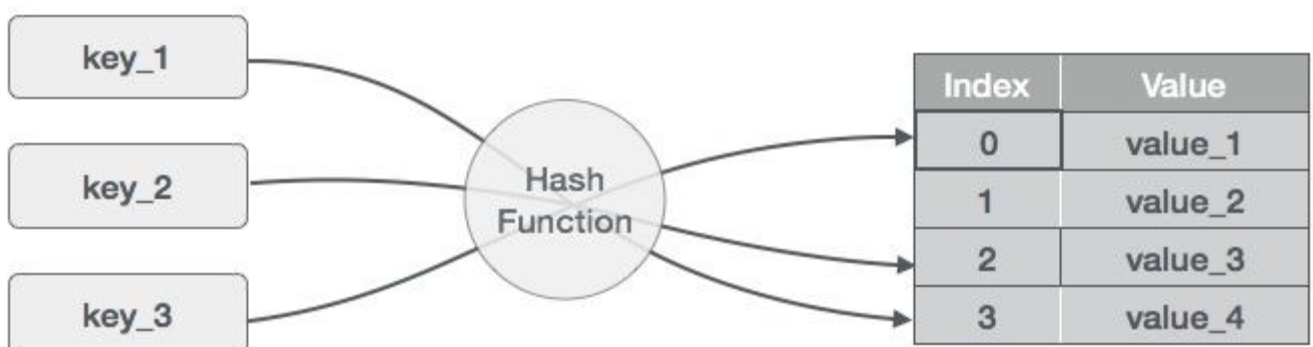
### Hash Table

Hash Table is a data structure which store data in associative manner. In hash table, data is stored in array format where each data value has its own unique index value. Access of data becomes very fast if we know the index of desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of size of data. Hash Table uses array as a storage medium and uses hash technique to generate index where an element is to be inserted or to be located from.

### Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and following items are to be stored. Item are in (key,value) format.



- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

S.n.	Key	Hash	Array Index
1	1	$1 \% 20 = 1$	1
2	2	$2 \% 20 = 2$	2
3	42	$42 \% 20 = 2$	2
4	4	$4 \% 20 = 4$	4
5	12	$12 \% 20 = 12$	12
6	14	$14 \% 20 = 14$	14
7	17	$17 \% 20 = 17$	17
8	13	$13 \% 20 = 13$	13
9	37	$37 \% 20 = 17$	17

### Linear Probing

As we can see, it may happen that the hashing technique used here, creates already used index of the array. In such case, we can search the next empty location in the array by looking into the next cell until we found an empty cell. This technique is called linear probing.

S.n.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	$1 \% 20 = 1$	1	1
2	2	$2 \% 20 = 2$	2	2
3	42	$42 \% 20 = 2$	2	3

4	4	$4 \% 20 = 4$	4	4
5	12	$12 \% 20 = 12$	12	12
6	14	$14 \% 20 = 14$	14	14
7	17	$17 \% 20 = 17$	17	17
8	13	$13 \% 20 = 13$	13	13
9	37	$37 \% 20 = 17$	17	18

### Basic Operations

Following are basic primary operations of a hashtable which are following.

- **Search** – search an element in a hashtable.
- **Insert** – insert an element in a hashtable.
- **delete** – delete an element from a hashtable.

### DataItem

Define a data item having some data, and key based on which search is to be conducted in hashtable.

```
struct DataItem {
    int data;
    int key;
};
```

### HashMethod

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){
    return key % SIZE;
}
```

### SearchOperation

Whenever an element is to be searched. Compute the hash code of the key passed and locate the element using that hashcode as index in the array. Use linear probing to get element ahead if element not found at computed hash code.

```
struct DataItem *search(int key){
    //get the hash
    int hashIndex = hashCode(key);
    //move in array until an empty
    while(hashArray[hashIndex] != NULL){
        if(hashArray[hashIndex]->key == key)
            return hashArray[hashIndex];
        //go to next cell
        ++hashIndex;
        //wrap around the table
        hashIndex %= SIZE;
    }
    return NULL;
}
```

### Insert Operation

Whenever an element is to be inserted. Compute the hash code of the key passed and locate the index using that hashcode as index in the array. Use linear probing for empty location if an element is found at computed hash code.

```
void insert(int key,int data){
    struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
    item->data = data;
    item->key = key;
    //get the hash
    int hashIndex = hashCode(key);
    //move in array until an empty or deleted cell
    while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1){
        //go to next cell
        ++hashIndex;
        //wrap around the table
        hashIndex %= SIZE;
    }
```

```
}  
hashArray[hashIndex] = item;  
}
```

### Delete Operation

Whenever an element is to be deleted. Compute the hash code of the key passed and locate the index using that hashcode as index in the array. Use linear probing to get element ahead if an element is not found at computed hash code. When found, store a dummy item there to keep performance of hashtable intact.

```
struct DataItem* delete(struct DataItem* item){  
    int key = item->key;  
    //get the hash  
    int hashIndex = hashCode(key);  
    //move in array until an empty  
    while(hashArray[hashIndex] != NULL){  
        if(hashArray[hashIndex]->key == key){  
            struct DataItem* temp = hashArray[hashIndex];  
            //assign a dummy item at deleted position  
            hashArray[hashIndex] = dummyItem;  
            return temp;  
        }  
        //go to next cell  
        ++hashIndex;  
        //wrap around the table  
        hashIndex %= SIZE;  
    }  
    return NULL;  
}
```

**What is Collision?**

Since a hash function gets us a small number for a key which is a big integer or string, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique.

**What are the chances of collisions with large table?**

Collisions are very likely even if we have big table to store keys. An important observation is Birthday Paradox. With only 23 persons, the probability that two people have same birthday is 50%.

**How to handle Collisions?**

There are mainly two methods to handle collision:

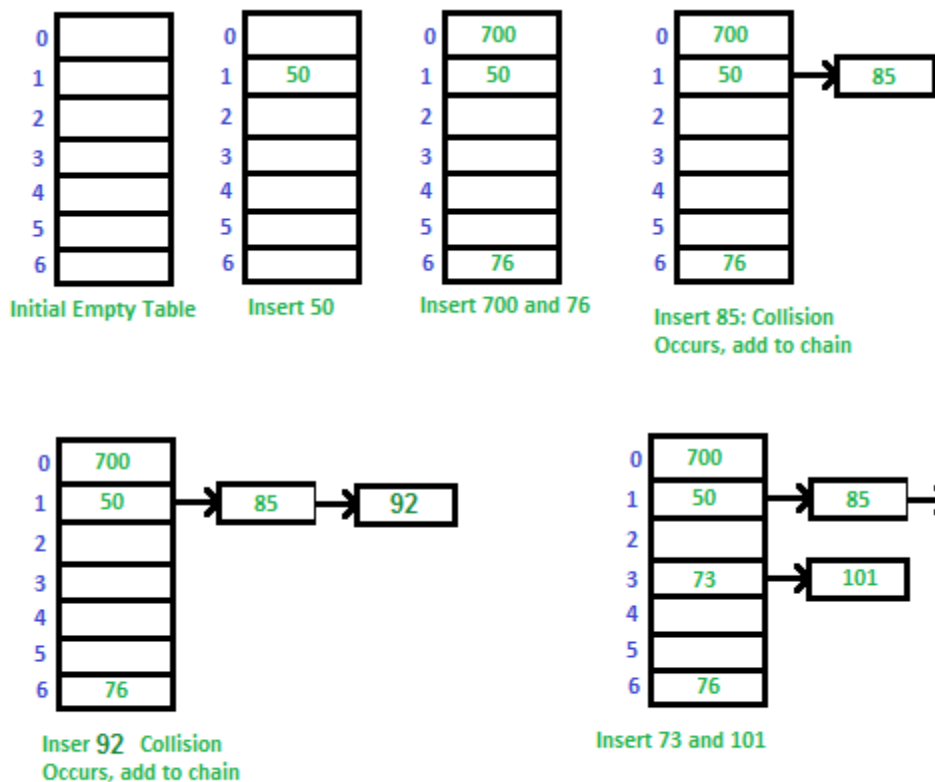
- 1) Separate Chaining
- 2) Open Addressing

**Separate Chaining:**

The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let us consider a simple hash function as "key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



**Advantages:**

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

**Disadvantages:**

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

## Open Addressing

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, size of table must be greater than or equal to total number of keys (Note that we can increase table size by copying old data if needed).

Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.

Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.

Delete(k): Delete operation is interesting. If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as "deleted".

Insert can insert an item in a deleted slot, but search doesn't stop at a deleted slot.

**Open Addressing is done following ways:**

### a) Linear Probing:

In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also.

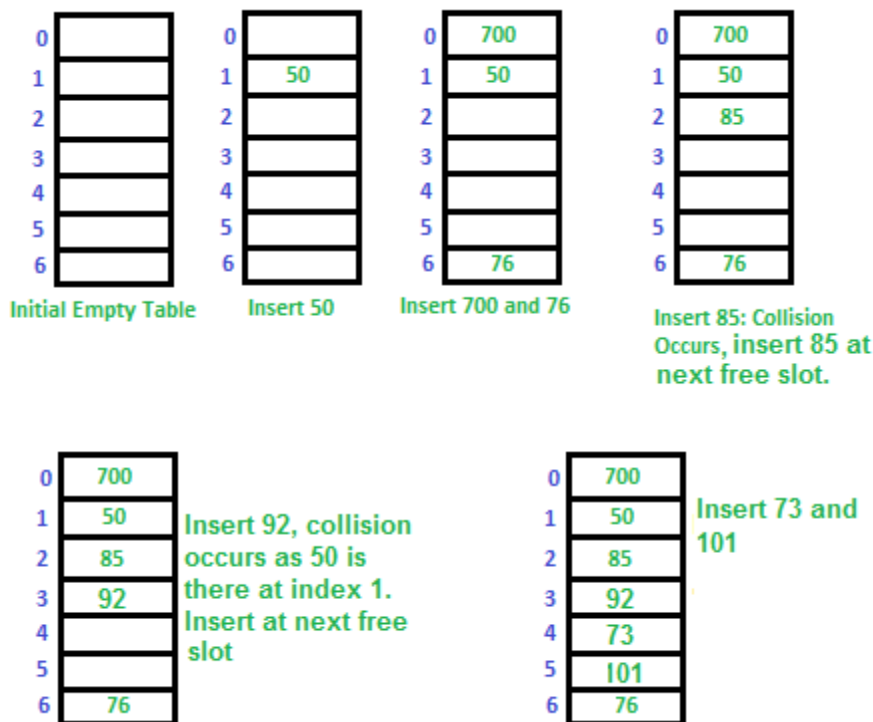
let  $\text{hash}(x)$  be the slot index computed using hash function and  $S$  be the table size

**If slot  $\text{hash}(x) \% S$  is full, then we try  $(\text{hash}(x) + 1) \% S$**

**If  $(\text{hash}(x) + 1) \% S$  is also full, then we try  $(\text{hash}(x) + 2) \% S$**

**If  $(\text{hash}(x) + 2) \% S$  is also full, then we try  $(\text{hash}(x) + 3) \% S$**

Let us consider a simple hash function as "key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



### Clustering:

The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

### b) Quadratic Probing

We look for  $i^2$ 'th slot in  $i$ 'th iteration.

let  $\text{hash}(x)$  be the slot index computed using hash function.

**If slot  $\text{hash}(x) \% S$  is full, then we try  $(\text{hash}(x) + 1^2) \% S$**

**If  $(\text{hash}(x) + 1^2) \% S$  is also full, then we try  $(\text{hash}(x) + 2^2) \% S$**

**If  $(\text{hash}(x) + 2^2) \% S$  is also full, then we try  $(\text{hash}(x) + 3^2) \% S$**

### c) Double Hashing

We use another hash function  $\text{hash}_2(x)$  and look for  $i * \text{hash}_2(x)$  slot in  $i$ 'th rotation.

let  $\text{hash}(x)$  be the slot index computed using hash function.

**If slot  $\text{hash}(x) \% S$  is full, then we try  $(\text{hash}(x) + 1 * \text{hash}_2(x)) \% S$**

**If  $(\text{hash}(x) + 1 * \text{hash}_2(x)) \% S$  is also full, then we try  $(\text{hash}(x) + 2 * \text{hash}_2(x)) \% S$**

**If  $(\text{hash}(x) + 2 * \text{hash}_2(x)) \% S$  is also full, then we try  $(\text{hash}(x) + 3 * \text{hash}_2(x)) \% S$**

**Comparison of above three:**

Linear probing has the best cache performance, but suffers from clustering. One more advantage of Linear probing is easy to compute.

Quadratic probing lies between the two in terms of cache performance and clustering.

Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

**Open Addressing vs. Separate Chaining****Advantages of Chaining:**

- 1) Chaining is Simpler to implement.
- 2) In chaining, Hash table never fills up, we can always add more elements to chain. In open addressing, table may become full.
- 3) Chaining is Less sensitive to the hash function or load factors.
- 4) Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.
- 5) Open addressing requires extra care for to avoid clustering and load factor.

**Advantages of Open Addressing**

- 1) Cache performance of chaining is not good as keys are stored using linked list. Open addressing provides better cache performance as everything is stored in same table.
- 2) Wastage of Space (Some Parts of hash table in chaining are never used). In Open addressing, a slot can be used even if an input doesn't map to it.
- 3) Chaining uses extra space for links.

**Rehashing:**

- It is a closed hashing technique.
- If the table gets too full, then the rehashing method builds new table that is about twice as big and scan down the entire original hash table, comparing the new hash value for each element and inserting it in the new table.
- Rehashing is very expensive since the running time is  $O(N)$ , since there are  $N$  elements to rehash and the table size is roughly  $2N$

Rehashing can be implemented in several ways like

1. Rehash , as soon as the table is half full
2. Rehash only when an insertion fails

Example : Suppose the elements 13, 15, 24, 6 are inserted into an open addressing hash table of size 7 and if linear probing is used when collision occurs.

0	6
1	15
2	
3	24
4	
5	
6	13

If 23 is inserted, the resulting table will be over 70 percent full.

0	6
1	15
2	23
3	24
4	
5	
6	13

A new table is created. The size of the new table is 17, as this is the first prime number that is twice as large as the old table size.

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	1
16	

#### Advantages

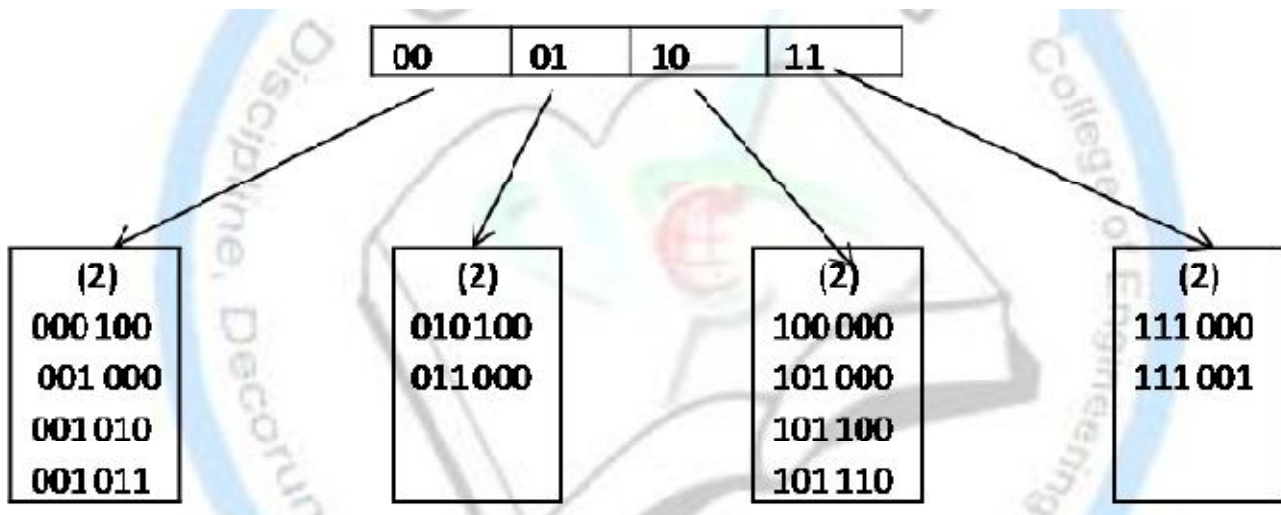
- Programmer doesn't worry about the table size
- Simple to implement

#### Extendible Hashing:

- When open addressing or separate hashing is used, collisions could cause several blocks to be examined during a Find operation, even for a well distributed hash table.
- Furthermore , when the table gets too full, an extremely expensive rehashing step must be performed, which requires  $O(N)$  disk accesses.
- These problems can be avoided by using extendible hashing.
- Extendible hashing uses a tree to insert keys into the hash table.

#### Example:

- Consider the key consists of several 6 bit integers.
- The root of the "tree" contains 4 pointers determined by the leading 2 bits.
- In each leaf the first 2 bits are identified and indicated in parenthesis.
- D represents the number of bits used by the root(directory)
- The number of entries in the directory is  $2^D$



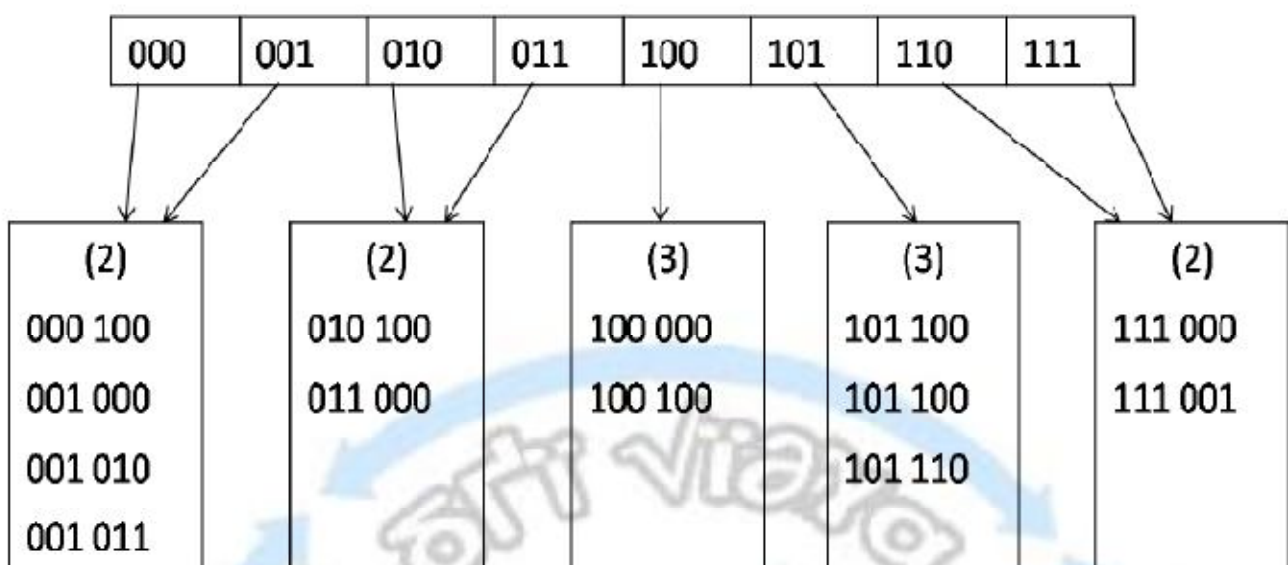
Suppose to insert the key 100100.

This would go to the third leaf but as the third leaf is already full.

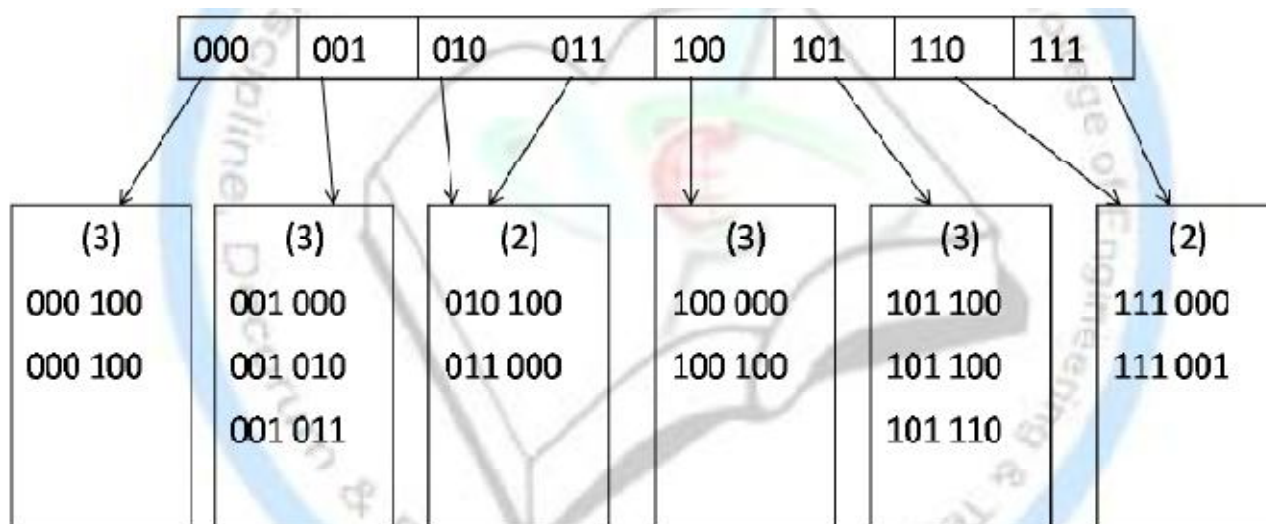
So split this

leaf into two leaves, which are now determined by the first three bits.

Now the directory size is increased to 3.



Similarly if the key 000100 is to be inserted, then the first leaf is split into 2 leaves.



Advantages & Disadvantages:

- Advantages
  - Provides quick access times for insert and find operations on large databases.
- Disadvantages
  - This algorithm does not work if there are more than M duplicates