

# UNIT 5 GRAPHS

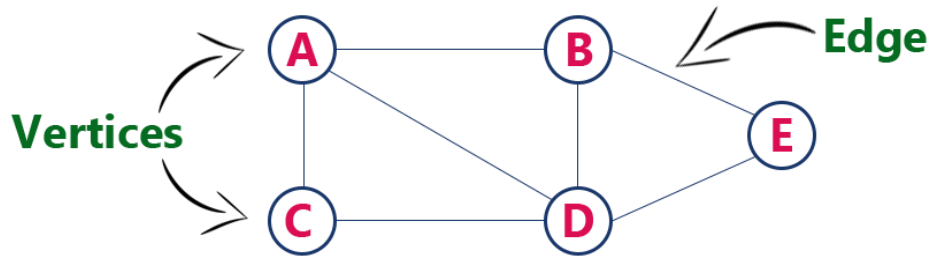
## Introduction to graphs

Graph is a non-linear data structure. It contains a set of points known as nodes (or vertices) and a set of links known as edges (or Arcs). Here edges are used to connect the vertices. A graph is defined as follows...

**Graph is a collection of vertices and arcs in which vertices are connected with arcs**

Generally, a graph **G** is represented as **G = ( V , E )**, where **V** is set of vertices and **E** is set of edges.

The following is a graph with 5 vertices and 6 edges. This graph G can be defined as  $G = (V, E)$  Where  $V = \{A, B, C, D, E\}$  and  $E = \{(A, B), (A, C), (A, D), (B, D), (C, D), (B, E), (E, D)\}$ .



## Graph Terminology

We use the following terms in graph data structure...

### Vertex

Individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

### Edge

An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex). For example, in above graph the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

1. **Undirected Edge** - An undirected edge is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
2. **Directed Edge** - A directed edge is a unidirectional edge. If there is directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
3. **Weighted Edge** - A weighted edge is a edge with value (cost) on it.

## Undirected Graph

A graph with only undirected edges is said to be undirected graph.

## Directed Graph

A graph with only directed edges is said to be directed graph.

## Mixed Graph

A graph with both undirected and directed edges is said to be mixed graph.

## End vertices or Endpoints

The two vertices joined by edge are called end vertices (or endpoints) of that edge.

## Origin

If a edge is directed, its first endpoint is said to be the origin of it.

## Destination

If a edge is directed, its first endpoint is said to be the origin of it and the other endpoint is said to be the destination of that edge.

## Adjacent

If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, vertices A and B are said to be adjacent if there is an edge between them.

## Incident

Edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

## Outgoing Edge

A directed edge is said to be outgoing edge on its origin vertex.

## Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

## Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

## Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

## Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

## Parallel edges or Multiple edges

If there are two undirected edges with same end vertices and two directed edges with same origin and destination, such edges are called parallel edges or multiple edges.

## Self-loop

Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.

## Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

## Path

A path is a sequence of alternate vertices and edges that starts at a vertex and ends at other vertex such that each edge is incident to its predecessor and successor vertex.

---

# Graph Representations

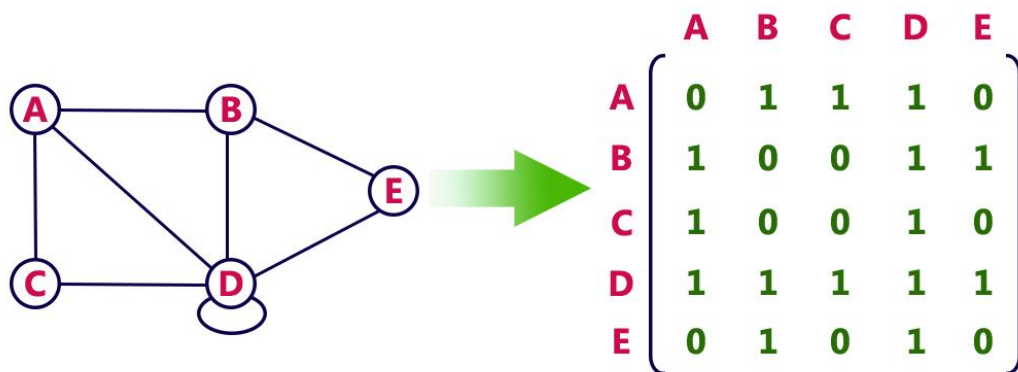
Graph data structure is represented using following representations...

1. Adjacency Matrix
2. Adjacency List

## Adjacency Matrix

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

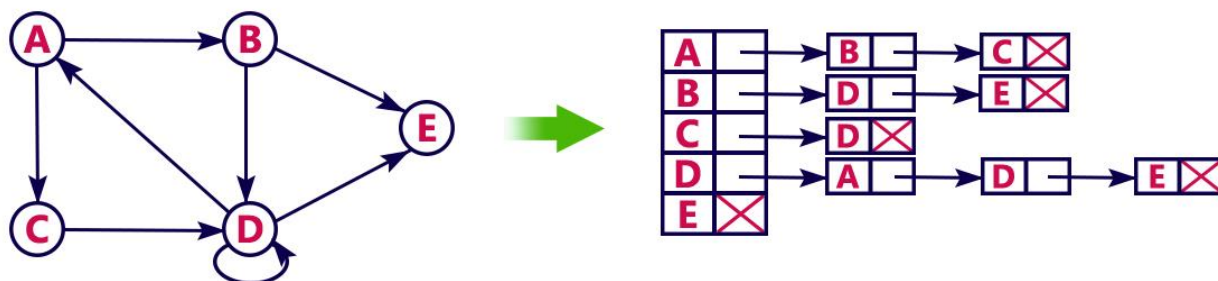
For example, consider the following undirected graph representation...



## Adjacency List

In this representation, every vertex of a graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



# Graph Traversal

**Graph traversal** is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

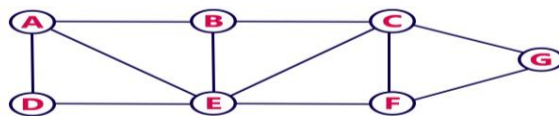
1. DFS (Depth First Search)
2. BFS (Breadth First Search)

## DFS (Depth First Search)

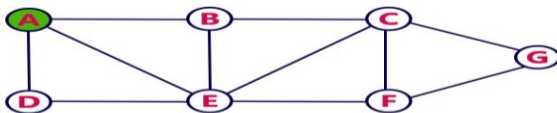
DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.

We use the following steps to implement DFS traversal...

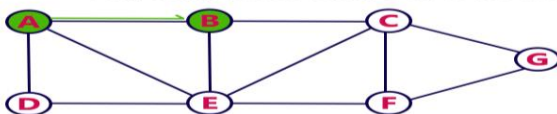
- **Step 1** - Define a Stack of size total number of vertices in the graph.
- **Step 2** - Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3** - Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.
- **Step 4** - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
- **Step 5** - When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.



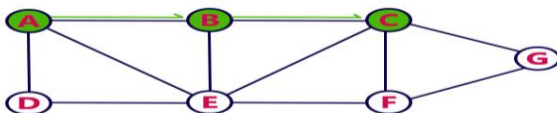
- Step 1:**
- Select the vertex **A** as starting point (visit **A**).
  - Push **A** on to the Stack.



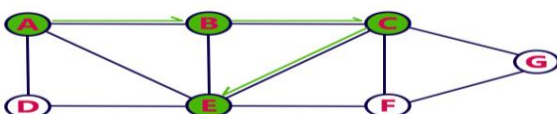
- Step 2:**
- Visit any adjacent vertex of **A** which is not visited (**B**).
  - Push newly visited vertex **B** on to the Stack.



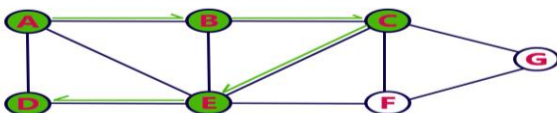
- Step 3:**
- Visit any adjacent vertex of **B** which is not visited (**C**).
  - Push **C** on to the Stack.



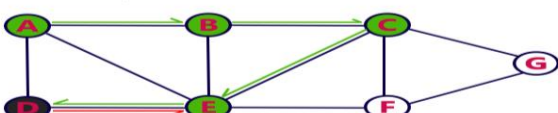
- Step 4:**
- Visit any adjacent vertex of **C** which is not visited (**E**).
  - Push **E** on to the Stack



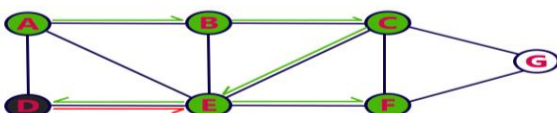
- Step 5:**
- Visit any adjacent vertex of **E** which is not visited (**D**).
  - Push **D** on to the Stack



- Step 6:**
- There is no new vertex to be visited from **D**. So use back track.
  - Pop **D** from the Stack.



- Step 7:**
- Visit any adjacent vertex of **E** which is not visited (**F**).
  - Push **F** on to the Stack.

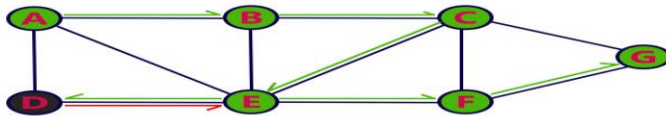


- **Step 6** - Repeat steps 3, 4 and 5 until stack becomes Empty.

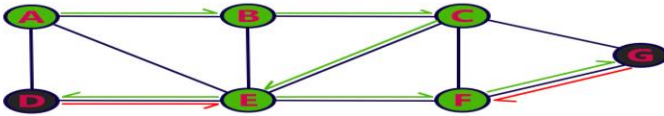
- **Step 7** - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

**Step 8:**

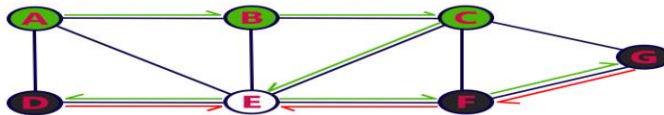
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

**Stack****Stack****Step 9:**

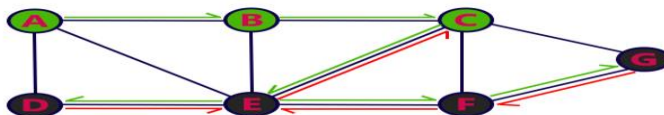
- There is no new vertex to be visited from **G**. So use back track.
- Pop **G** from the Stack.

**Stack****Step 10:**

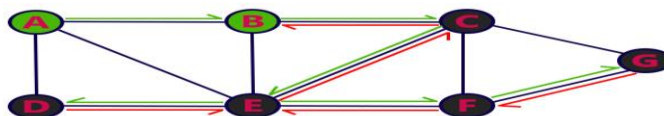
- There is no new vertex to be visited from **F**. So use back track.
- Pop **F** from the Stack.

**Stack****Step 11:**

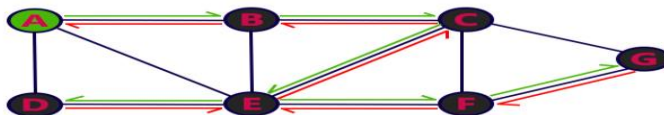
- There is no new vertex to be visited from **E**. So use back track.
- Pop **E** from the Stack.

**Stack****Step 12:**

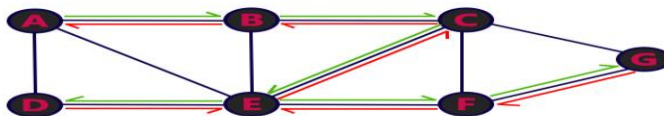
- There is no new vertex to be visited from **C**. So use back track.
- Pop **C** from the Stack.

**Stack****Step 13:**

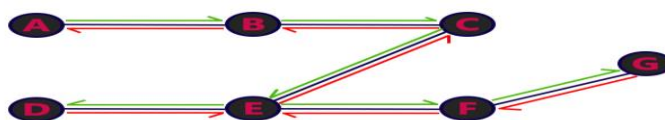
- There is no new vertex to be visited from **B**. So use back track.
- Pop **B** from the Stack.

**Stack****Step 14:**

- There is no new vertex to be visited from **A**. So use back track.
- Pop **A** from the Stack.

**Stack**

- Stack became Empty. So stop DFS Traversal.
- Final result of DFS traversal is following spanning tree.



## BFS (Breadth First Search)

BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

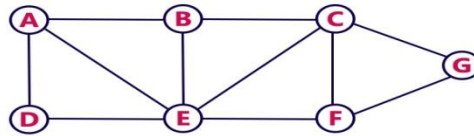
We use the following steps to implement BFS traversal...

- **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

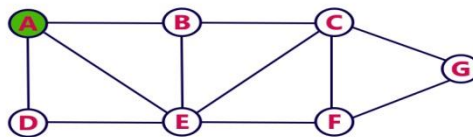
## Example

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.

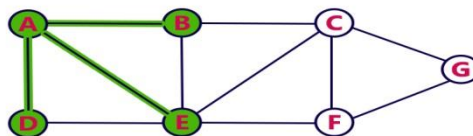


Queue



### 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..

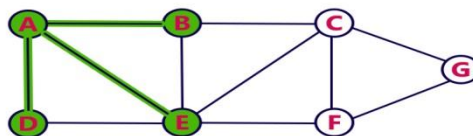


Queue



### Step 3:

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

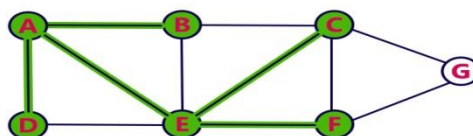


Queue



### 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.



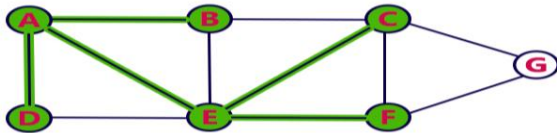
Queue



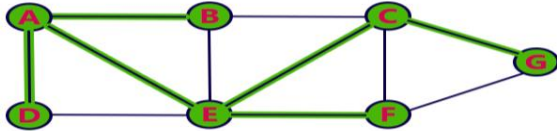


**Step 5:**

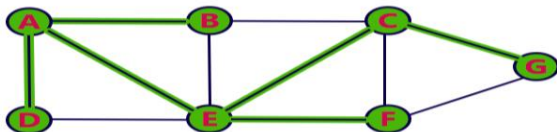
- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

**Queue****Step 6:**

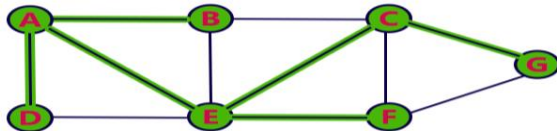
- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

**Queue****Step 7:**

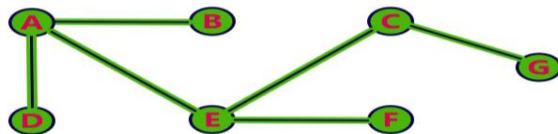
- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

**Queue****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...



## Minimum Spanning Tree

### What is a Spanning Tree?

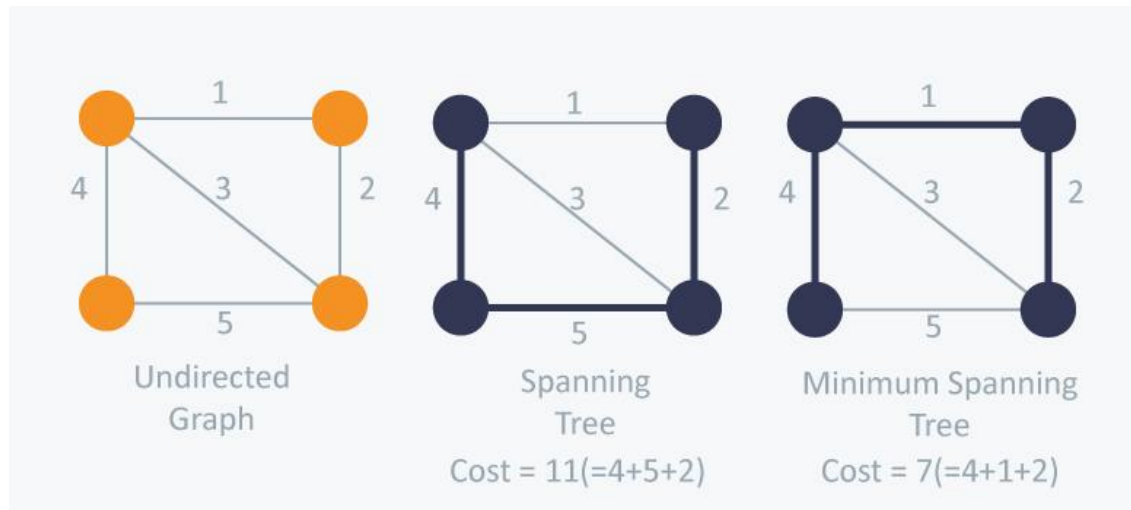
Given an undirected and connected graph  $G=(V,E)$ , a spanning tree of the graph  $G$  is a tree that spans  $G$  (that is, it includes every vertex of  $G$ ) and is a subgraph of  $G$  (every edge in the tree belongs to  $G$ )

### What is a Minimum Spanning Tree?

The minimum 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.

## Applications

1. Used in the design of networks.
2. Used to travelling salesman problem
3. Cluster Analysis
4. Handwriting recognition
5. Image segmentation



There are two famous algorithms for finding the Minimum Spanning Tree:

### Kruskal's Algorithm

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

#### Algorithm Steps:

1. Sort the graph edges with respect to their weights.
2. Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
3. Only add edges which doesn't form a cycle, edges which connect only disconnected components.

```

1. KRUSKAL(G):
2. A = ∅
3. For each vertex v ∈ G.V:
4.   MAKE-SET(v)
5. For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):
6.   if FIND-SET(u) ≠ FIND-SET(v):
7.     A = A ∪ {(u, v)}
8.     UNION(u, v)
9. return A
  
```

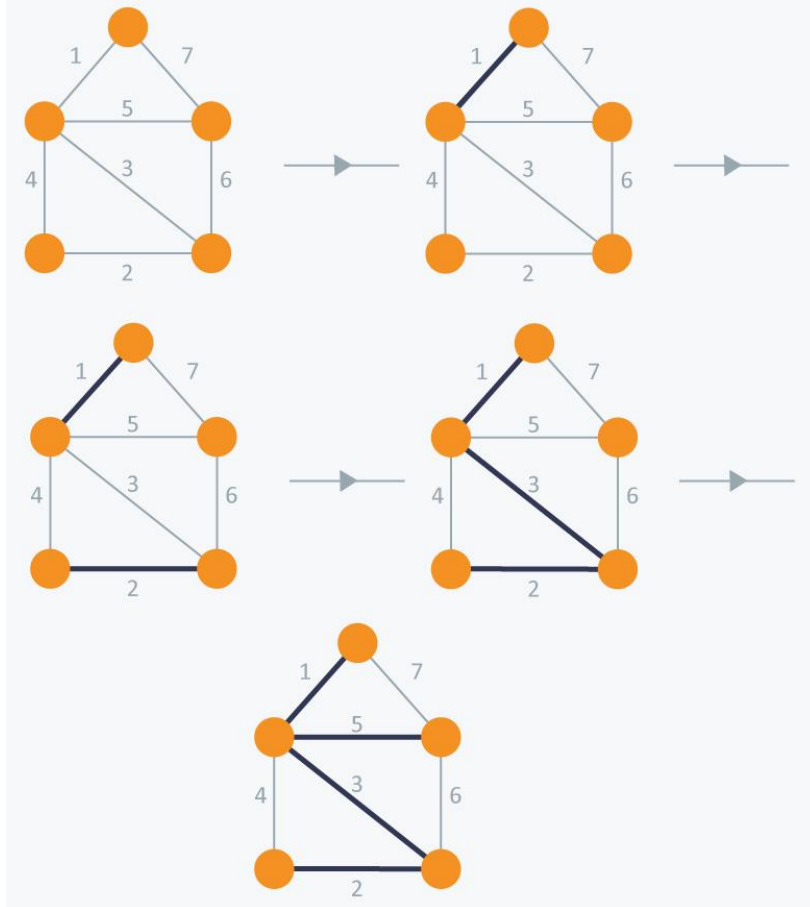
This could be done using DFS which starts from the first vertex, then check if the second vertex is visited or not.

But DFS will make time complexity of  $O(V+E)$  where  $V$  is the number of vertices,  $E$  is the number of edges

Consider following example:



### Kruskal's Algorithm



In Kruskal's algorithm, at each iteration we will select the edge with the lowest weight.

1. So, we will start with the lowest weighted edge first i.e., the edges with weight 1.
2. After that we will select the second lowest weighted edge i.e., edge with weight 2. Notice these two edges are totally disjoint.
3. Now, the next edge will be the third lowest weighted edge i.e., edge with weight 3, which connects the two disjoint pieces of the graph.
4. Now, we are not allowed to pick the edge with weight 4, that will create a cycle and we can't have any cycles. So we will select the fifth lowest weighted edge i.e., edge with weight 5.
5. Now the other two edges will create cycles so we will ignore them. In the end, we end up with a minimum spanning tree with total cost 11 ( $= 1 + 2 + 3 + 5$ ).

### Prim's Algorithm

Prim's Algorithm also use Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an **edge** in Kruskal's, we add **vertex** to the growing spanning tree in Prim's.

#### Algorithm Steps:

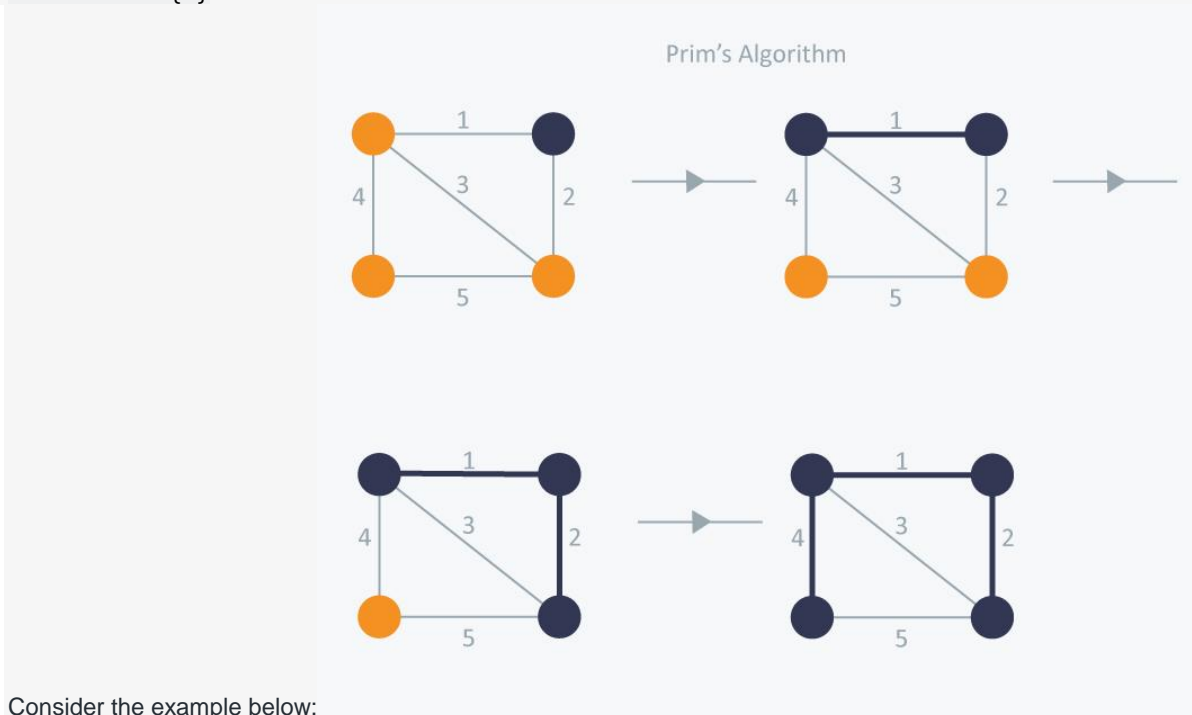
1. Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
2. Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
3. Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

1.  $T = \emptyset$ ;

```

2.  $U = \{1\};$ 
3. while ( $U \neq V$ )
4.   let  $(u, v)$  be the lowest cost edge such that  $u \in U$  and  $v \in V - U;$ 
5.    $T = T \cup \{(u, v)\}$ 
6.    $U = U \cup \{v\}$ 

```



In Prim's Algorithm, we will start with an arbitrary node (it doesn't matter which one) and mark it. In each iteration we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex.

1. we will simply choose the edge with weight 1.
2. In the next iteration we have three options, edges with weight 2, 3 and 4. So, we will select the edge with weight 2 and mark the vertex.
3. Now again we have three options, edges with weight 3, 4 and 5. But we can't choose edge with weight 3 as it is creating a cycle. So we will select the edge with weight 4 and we end up with the minimum spanning tree of total cost 7 ( $= 1 + 2 + 4$ ).

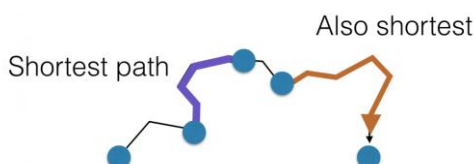
## Dijkstra's Algorithm

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

It differs from minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

### How Dijkstra's Algorithm works

Dijkstra's Algorithm works on the basis that any subpath  $B \rightarrow D$  of the shortest path  $A \rightarrow D$  between vertices A and D is also the shortest path between vertices B and D.



Dijkstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbours to find the shortest subpath to those neighbours.

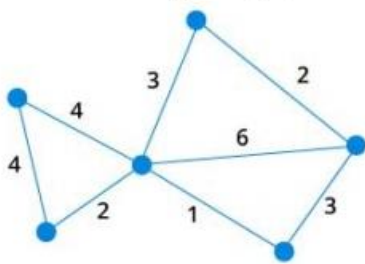
The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

## Example of Dijkstra's algorithm

It is easier to start with an example and then think about the algorithm.

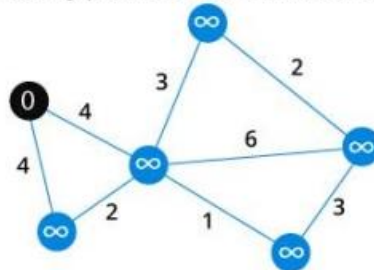
1

Start with a weighted graph



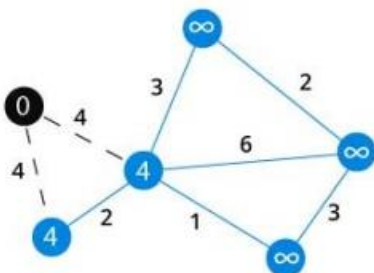
2

Choose a starting vertex and assign infinity path values to all other vertices



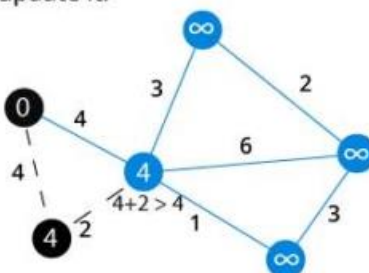
3

Go to each vertex adjacent to this vertex and update its path length



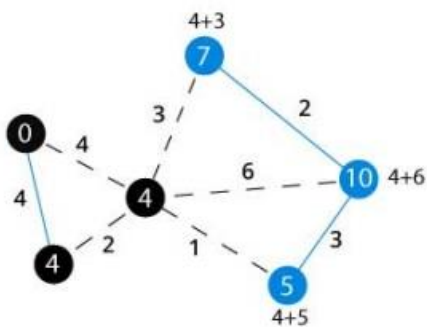
4

If the path length of adjacent vertex is lesser than new path length, don't update it.



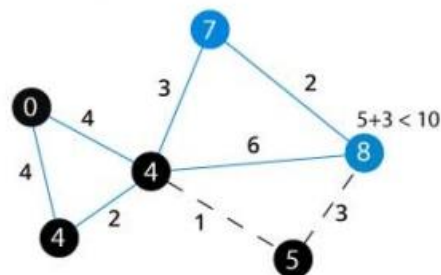
5

Avoid updating path lengths of already visited vertices



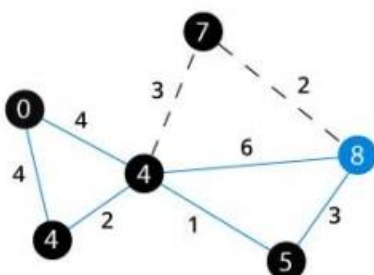
6

After each iteration, we pick the unvisited vertex with least path length. So we chose 5 before 7



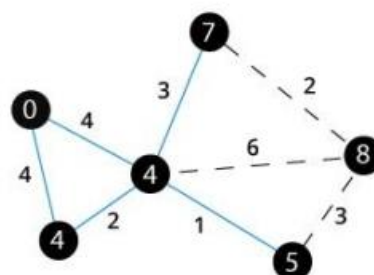
7

Notice how the rightmost vertex has its path length updated twice



8

Repeat until all the vertices have been visited



## Dijkstra's algorithm pseudocode

We need to maintain the path distance of every vertex. We can store that in an array of size  $v$ , where  $v$  is the number of vertices.

We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

A minimum priority queue can be used to efficiently receive the vertex with least path distance.

```
1. function dijkstra(G, S)
2.   for each vertex V in G
3.     distance[V] <- infinite
4.     previous[V] <- NULL
5.     If V != S, add V to Priority Queue Q
6.   distance[S] <- 0
7.
8.   while Q IS NOT EMPTY
9.     U <- Extract MIN from Q
10.    for each unvisited neighbour V of U
11.      tempDistance <- distance[U] + edge_weight(U, V)
12.      if tempDistance < distance[V]
13.        distance[V] <- tempDistance
14.        previous[V] <- U
15.   return distance[], previous[]
```

## Bellman Ford's Algorithm

Bellman Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph.

It is similar to [Dijkstra's algorithm](#) but it can work with graphs in which edges can have negative weights.

### Why would one ever have edges with negative weights in real life?

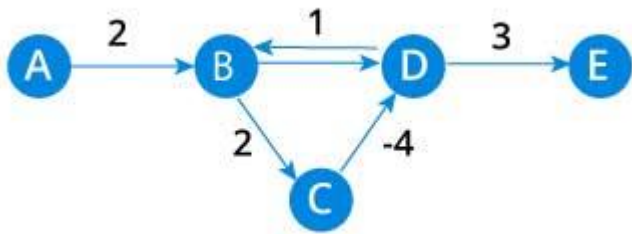
Negative weight edges might seem useless at first but they can explain a lot of phenomena like cashflow, heat released/absorbed in a chemical reaction etc.

For instance, if there are different ways to reach from one chemical A to another chemical B, each method will have sub-reactions involving both heat dissipation and absorption.

If we want to find the set of reactions where minimum energy is required, then we will need to be able to factor in the heat absorption as negative weights and heat dissipation as positive weights.

### Why we need to be careful with negative weights?

Negative weight edges can create negative weight cycles i.e. a cycle which will reduce the total path distance by coming back to the same point.



Shortest path algorithms like Dijkstra's Algorithm that aren't able to detect such a cycle can give an incorrect result because they can go through a negative weight cycle and reduce the path length.

## How Bellman Ford's algorithm works

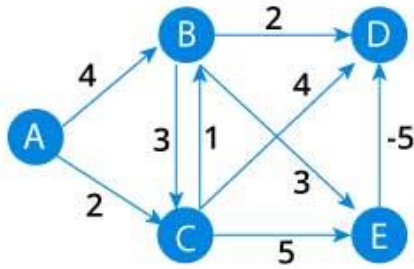
Bellman Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths.

By doing this repeatedly for all vertices, we are able to guarantee that the end result is optimized.



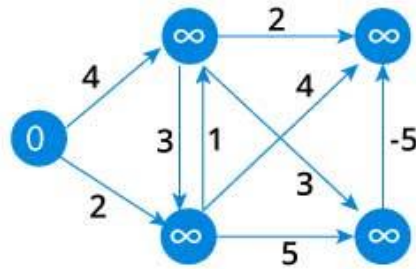
1

Start with a weighted graph



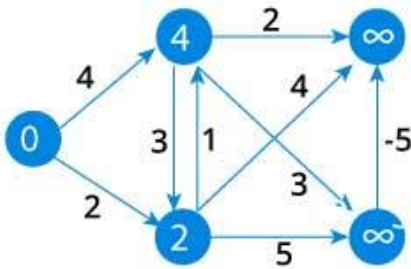
2

Choose a starting vertex and assign infinity path values to all other vertices



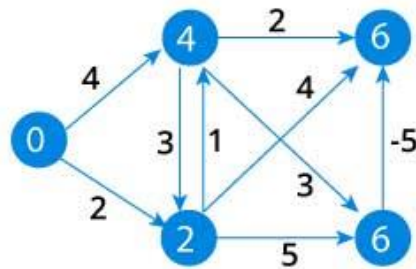
3

Visit each edge and relax the path distances if they are inaccurate



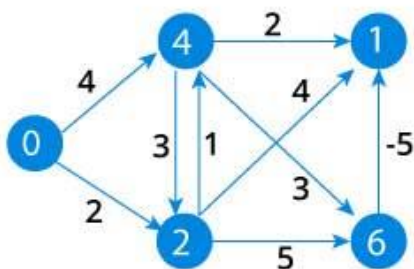
4

We need to do this V times because in the worst case, a vertex's path length might need to be readjusted V times



5

Notice how the vertex at the top right corner had its path length adjusted



6

After all the vertices have their path lengths, we check if a negative cycle is present.

A	B	C	D	E
0	∞	∞	∞	∞
0	4	2	∞	∞
0	3	2	6	6
0	3	2	1	6
0	3	2	1	6

## Bellman Ford Pseudocode

We need to maintain the path distance of every vertex. We can store that in an array of size  $v$ , where  $v$  is the number of vertices.

We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.


```
1. function bellmanFord(G, S)
2.   for each vertex V in G
3.     distance[V] <- infinite
4.     previous[V] <- NULL
5.   distance[S] <- 0
6.   for each vertex V in G
7.     for each edge (U,V) in G
8.       tempDistance <- distance[U] + edge_weight(U, V)
9.       if tempDistance < distance[V]
10.        distance[V] <- tempDistance
11.        previous[V] <- U
12.
13.   for each edge (U,V) in G
14.     If distance[U] + edge_weight(U, V) < distance[V]
15.       Error: Negative Cycle Exists
16.
17.   return distance[], previous[]
```

---


# SELECTION SORT

Selection sort is a simplest method of sorting technique. To sort the given list in ascending order, we will compare the first element with all the other elements. If the first element is found to be greater than the compared element, then they are interchanged. Thus at the end of first interaction, the smallest element will be stored in first position, which is its proper position. Then in the second interaction, we will repeat the procedure from second element to last element. The algorithm is continued till we get sorted list. If there are  $n$  elements, we require  $(n-1)$  iterations, in general.

**Consider the example----**

25   12   30   8   7   43   32  


First: Iteration

25   12   30   8   7   43   32  


12   25   30   8   7   43   32

12   25   30   8   7   43   32

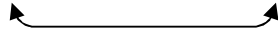
8   25   30   12   7   43   32

7   25   30   12   8   43   32

7   25   30   12   8   43   32

7   25   30   12   8   43   32







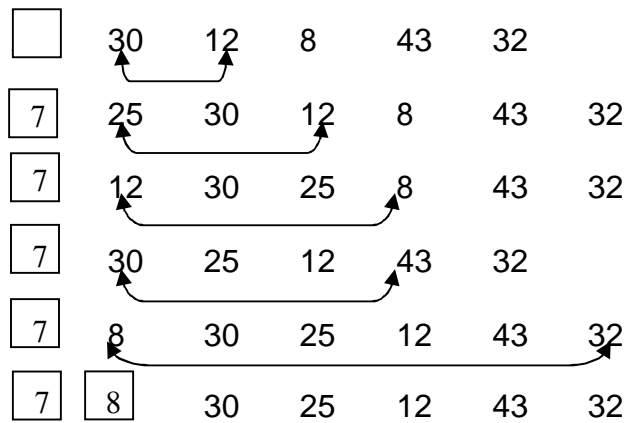




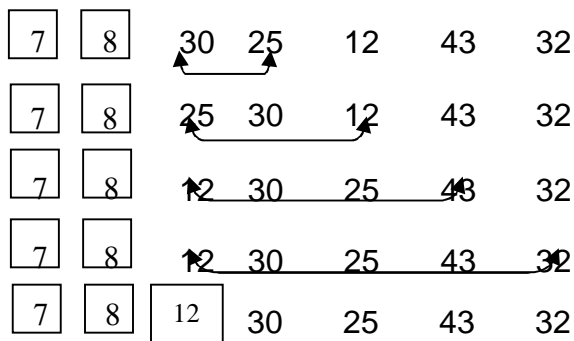


---

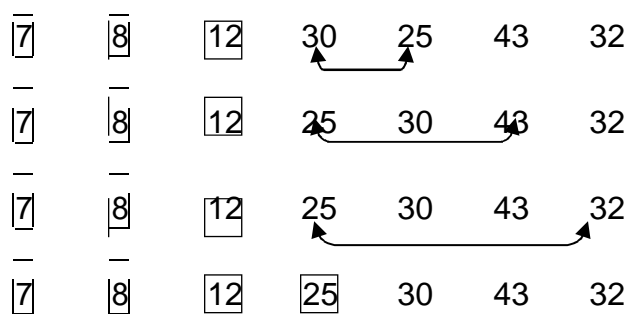
### Second Iteration



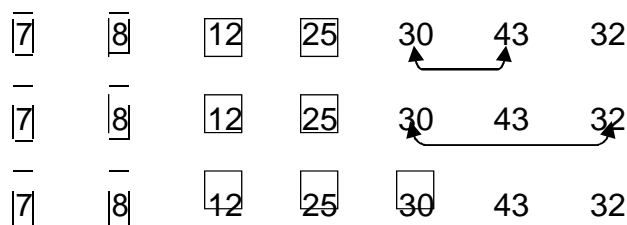
### Third Iteration



### Fourth Iteration



### Fifth Iteration



---

### Sixth Iteration

7	8	12	25	30	43	32
7	8	12	25	30	32	43

Thus sorted list is:

7, 8, 12, 25, 30, 32, 43

### Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[10],n,i,temp,j;
    clrscr();
    printf("Enter the size of the array:");
    scanf("%d",&n);
    printf("\nEnter array elements:\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    for(i=0;i<n-1;i++)
    {
        for(j=i+1;j<n;j++)
        {
            if(a[i]>a[j])
            {
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
            }
        }
    }
    printf("\nSorted list is:\n");
    for(i=0;i<n;i++)
        printf("%d\t",a[i]);
    getch();
}
```

### BUBBLE SORT

The bubble sort technique for sorting a list of data in ascending order is as follows: In the first iteration, the first element of the array is compared with the second element. If the first element is found to be greater than the second element, they are interchanged. Now, the

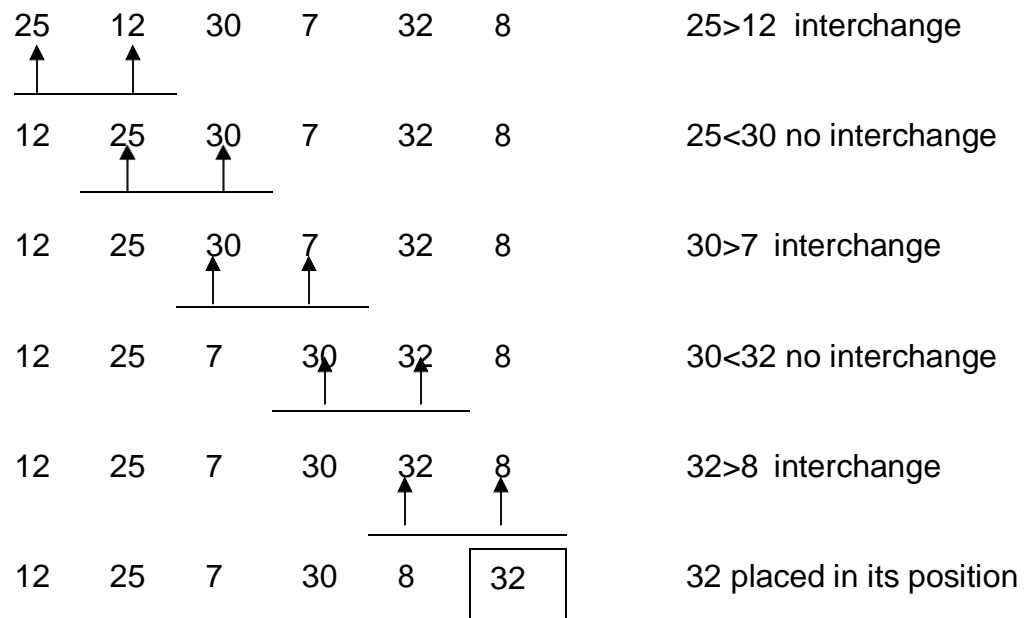
---

second element is compared with the third and interchanged if required. In the same way, comparison is done till the last element. At the end of first iteration, the largest element will be stored at the last position. In the second iteration, again the comparison is done from the first element to last-but-one element. At the end of this iteration, the second largest element will be placed in its proper position. If there are 'n' elements in the given list, then after (n-1) iterations, the array gets sorted.

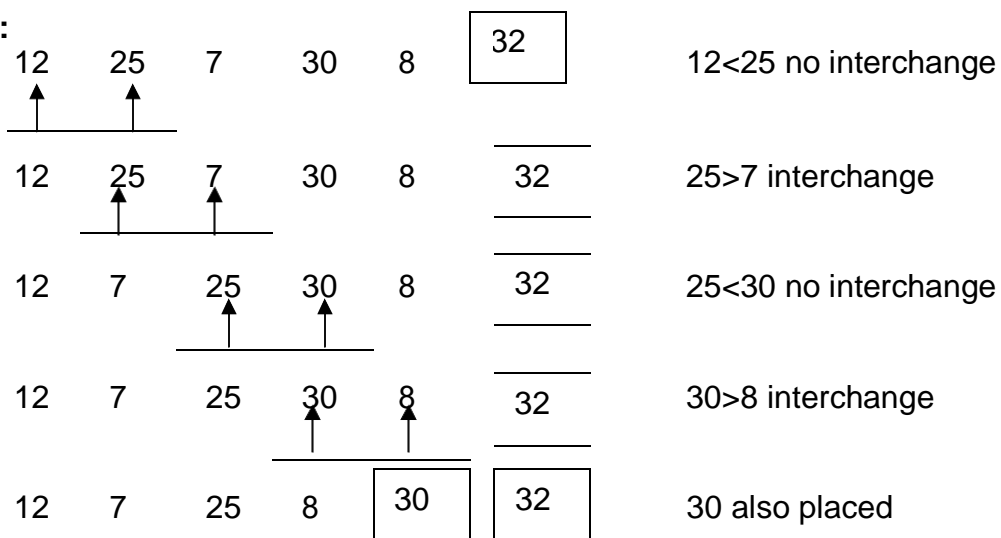
Consider the following list of integers to be sorted:

25    12    30    7    32    8

**1<sup>st</sup> Iteration:**

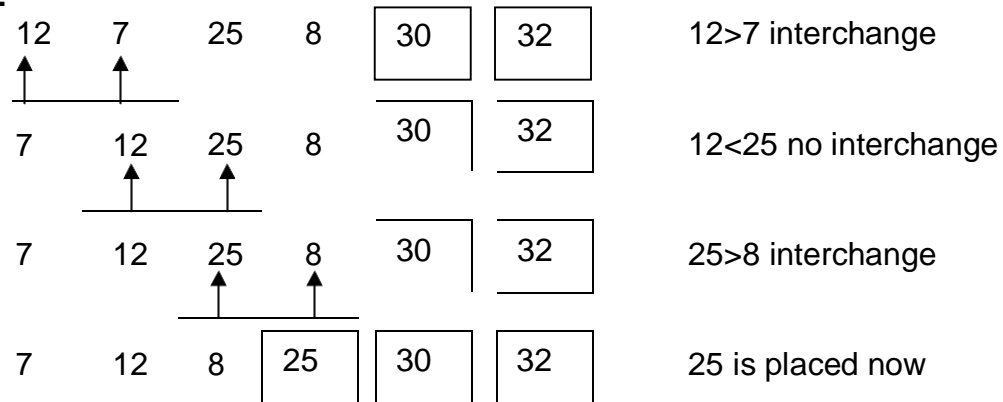
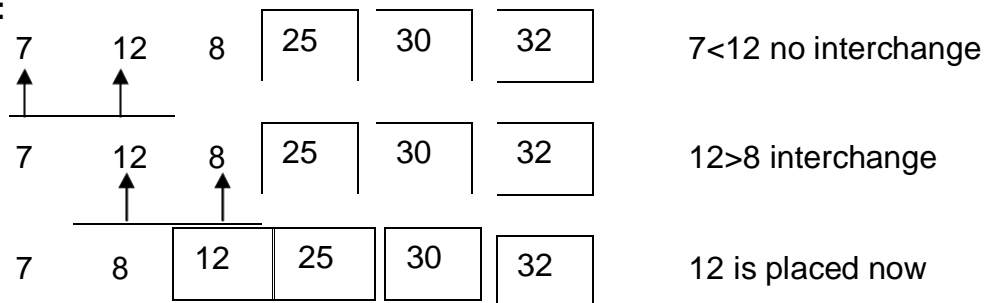
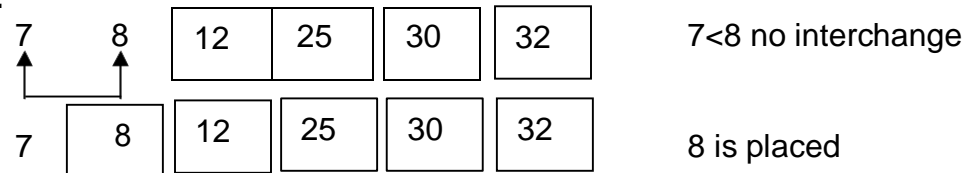


**2<sup>nd</sup> Iteration:**





---

**3<sup>rd</sup> Iteration:****4<sup>th</sup> Iteration:****5<sup>th</sup> Iteration:**

Thus, the sorted list is:

7      8      12      25      30      32

**Program:**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[10],n,i,temp,j;
    clrscr();
    printf("Enter the size of the array:");
    scanf("%d",&n);
    printf("\nEnter array elements:\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
```

---

---

```

for(i=0;i<n;i++)
{
    for(j=0;j<n-i-1;j++)
    {
        if(a[j]>a[j+1])
        {
            temp=a[j];
            a[j]=a[j+1];
            a[j+1]=temp;
        }
    }
}
printf("\nSorted list is:\n");
for(i=0;i<n;i++)
    printf("%d\t",a[i]);
getch();
}

```

## INSERTION SORT

This sorting technique involves inserting a particular element in proper position. In the first iteration, the second element is compared with the first. In second iteration, the third element is compared with second and then the first. Thus in every iteration, the element is compared with all the elements before it. If the element is found to be greater than any of its previous elements, then it is inserted at that position and all other elements are moved to one position towards right, to create the space for inserting element. The procedure is repeated till we get the sorted list.

### Consider an example

25    12    30    8    7    43    32

#### I iteration


 25    12    30    8    7    43    32    (12<25, so insert 12 at first position)

12    25    30    8    7    43    32

#### II iteration


 12    25    30    8    7    43    32    (30>25, so don't compare 30 with 12)

12    25    30    8    7    43    32

---

---

### III iteration

12 25 30 8 7 43 32 (8<30,25,12 so insert 8 at 1<sup>st</sup> position)



### IV iteration

8 12 25 30 7 43 32 (7<30,25,12,8. So insert 7 at 1<sup>st</sup> pos)




### V iteration

7 8 12 25 30 43 32 (43>30. So, don't compare 43 with other)



### VI iteration

7 8 12 25 30 43 32 (32<43 but 32>30. So, insert in-between)



Sorted list:

7 8 12 25 30 32 43

### Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[10],n,i,item,j;
    clrscr();
    printf("Enter the size of the array:");
    scanf("%d",&n);
    printf("\nEnter array elements:\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    for(i=1;i<n;i++)
    {
        item=a[i];
        for(j=i-1;j>=0 && item<a[j];j--)
            a[j+1]=a[j];
        a[j+1]=item;
    }
}
```

---

---

```

printf("Enter the size of second array:");
scanf("%d",&n);
printf("\nEnter second array(in sorted order):\n");
for(i=0;i<n;i++)
    scanf("%d",&b[i]);

MergeSort(a,b,c,m,n);

printf("\nSorted list is:\n");
for(i=0;i<m+n;i++)
    printf("%d\t",c[i]);
getch();
}

```

## RADIX SORT

The radix sort is based on the idea that the number which is having the highest significant digit greater than the corresponding digit of another number will be larger. That is, if we consider two numbers viz. 673 and 512, compare highest significant digits i.e. 6 and 5. Then as 6 is greater than 5, the number 673 is greater than 512. If the two digits are same, we will take next digit and compare. The process continues till we get the result. For a sorting method by radix sort; we assume that all the numbers to be sorted are having equal number of digits i.e. all are two digit numbers or all are three digit numbers etc. There will be ten packets numbering from 0 to 9. Initially all the packets are empty. We will scan all the numbers to be sorted one by one and separate the least significant digit and insert that number into appropriate packet. If the packet is non-empty, the element is inserted into rear-end of the packet. Once all the elements have been scanned, they are removed from the packets. The process is repeated till the array gets sorted.

Consider an example for radix sort. We have to sort the numbers-  
212, 310, 451, 117, 256, 813, 514, 315, 789, 618, 912, 513.

**Step (i):** Scan each number and put it into the appropriate packets based on last digit (least significant digit) of a number as below-

0	1	2	3	4	5	6	7	8	9
310	451	212	813	514	315	256	117	618	789
		912	513						

**Step (ii):** Now, remove elements from the packets-  
310, 451, 212, 912, 813, 513, 514, 315, 256, 117, 618, 789

Put these elements into packets based on 2<sup>nd</sup> digit as follows –

---

---

0	1	2	3	4	5	6	7	8	9
	310				451			789	
	212				256				
	912								
	813								
	513								
	315								
	117								
	618								

Again remove elements from packets:

310, 212, 912, 813, 513, 514, 315, 117, 618, 451, 256, 789.

**Step (iii)** Now consider highest significant digit (first digit) and put the elements into packets.

0	1	2	3	4	5	6	7	8	9
	117	212	313	451	513	618	789	813	912
		256	315		514				

Remove from the packets-

117, 212, 256, 313, 315, 451, 513, 514, 618, 789, 813, 912

This is a sorted list.

To implement radix sort in C programming, we need a function which will separate the required significant digits, a function to insert an element at the rear-end of the list and to find the largest. For these, we go for linked list data structure as below –

```
#include<stdio.h>
#include<math.h>
#include<process.h>
#include<alloc.h>

struct node
{
    int data;
    struct node *link;
};
typedef struct node *NODE;

/* Use a function getnode() from linked list example*/
/* Use a function insert_rear() which is used for single list program*/

int separate(int item, int j)
{
    return item/(int) pow(10, j-1)%10;
}
```

---

---

```

int largest(int a[], int n)
{
    int i, big;
    big=a[0];
    for(i=1;i<n;i++)
        if(a[i]>big)
            big=a[i];
    return big;
}

void radix_sort(int a[], int n)
{
    int i, j, k, m, big, digit;
    NODE p[10], temp;
    big=largest(a,n);
    m=log 10(big)+1;
    For(j=1;j<=m;j++)
    {
        for(i=0;i<=9;i++)    //0-9 packets
            p[i]=NULL;
        for(i=0;i<n;i++)
        {
            digit=separate (a[i],j);
            p[digit]=insert_rear(a[i], p[digit]);
        }
        k=0;
        for(i=0;i<=9;i++)
        {
            temp=p[i];
            while(temp!=NULL)
            {
                a[k++]=temp->data;
                temp=temp->link;
            }
        }
    }
}

void main()
{
    int n, i, a[20];
    printf("enter array size");
    scanf("%d",&n);
    printf("enter elements:\n");
    for(i=0;i<n;i++)
        scanf("%d", &a[i]);
}

```

---



---

```
radix_sort(a,n);
printf("\n sorted list");
for(i=0;i<n;i++)
    printf("%d",a[i]);
}
```

## ADDRESS CALCULATION SORT

**NOTE:** The readers are advised to understand the Hashing technique which is discussed in the next chapter (Searching) before reading this sorting technique.

The address calculation sort is also known as sorting by hashing. In this method, a function  $f$  is applied on each element, where the function  $f$  should be such that,  
if  $x \leq y$ , then  $f(x) \leq f(y)$

Such function is called as order preserving function. In this sorting method also, we assume that all the numbers to be sorted are having equal number of digits i.e. all are two digit numbers or all are three digit numbers etc. That is, if we consider two numbers viz. 673 and 512, compare highest significant digits i.e. 6 and 5. Then as 6 is greater than 5, the number 673 is greater than 512. If the two digits are same, we will take next digit and compare. The process continues till we get the result.

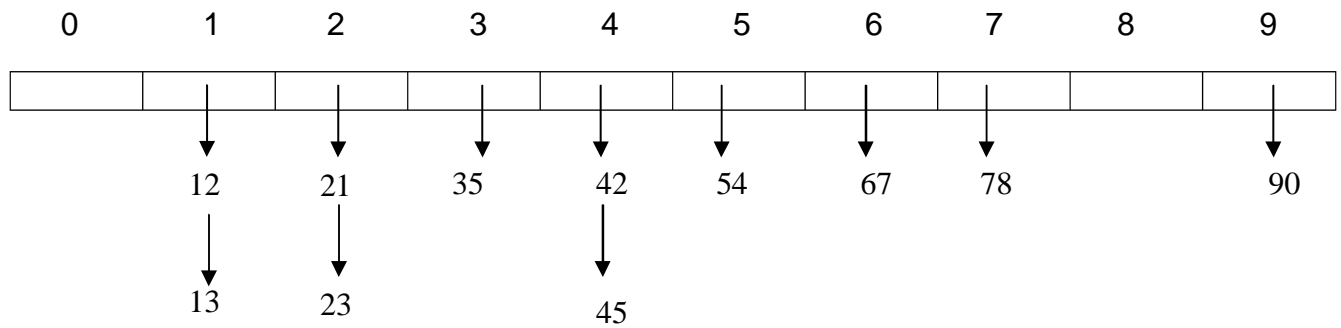
With the above assumption, the simplest function satisfying the above given equation, would be considering most significant digit of every number. That is,  
 $f(452) = 4$ ,  
 $f(127) = 1$  etc.

The sorting procedure is as follows: There will be ten packets numbering from 0 to 9 (similar to hash table). Initially all the packets are empty. Then, most significant digit of every item is computed, the item is hashed into respective packet. In case of hash collision, we use, open hashing (or separate chaining) using linked list. While inserting new item into a packet, we should see that the items in that packet are inserted in a sorted manner. After hashing all items into a hash table, just remove all the numbers from packet 0 till packet 9 sequentially. This will be a sorted list.

For example, consider the numbers: 21, 45, 13, 67, 42, 90, 78, 23, 54, 12, 35

Considering a hash function which results in most significant digit of a given number, the hash table would look like:

---



Observe that, while inserting 42, the number 45 was already hashed. But, 42 is inserted before 45. So, during each insertion into a packet, we should maintain an ordered list. Now, remove all the elements from packet 0 in an order. We will get a sorted list as:  
 12, 13, 21, 23, 35, 42, 45, 54, 67, 78, 90

```
#include<stdio.h>
#include<alloc.h>
#include<conio.h>
#include<stdlib.h>

struct node                                     //taken from linked list chapter
{
    int data;
    struct node *link;
};
typedef struct node *NODE;

NODE list[10]={NULL};

NODE getnode()                                 //taken from linked list chapter
{
    NODE x;
    x=(NODE) malloc(sizeof(struct node));
    if(x==NULL)
    {
        printf("no memory in heap");
        exit(0);
    }
    return x;
}

NODE insert(int item, NODE start)              //insert_order() function of linked lists
{
    NODE temp, cur, prev;
    temp = getnode();
```

---

---

```

temp->data=item;
temp->link=NULL;

if(start == NULL)
    return temp;

if(item < start->data)
{
    temp->link =start;
    return temp;
}
prev = NULL;
cur = start;
while(cur != NULL && item >= cur->data)
{
    prev = cur;
    cur = cur->link;
}
prev->link = temp;
temp->link=cur;
return start;
}

int msd(int x)           //hash function to retrieve most significant digit of  number
{
    while(x>10)
        x=x/10;

    printf("\n%d",x);
    return x;
}

void addcal(int arr[], int n)
{
    int i, j=0, pos;

    for(i=0;i<n;i++)
    {
        pos=msd(arr[i]);
        list[pos]=insert(arr[i],list[pos]);
    }

    for(i=0;i<10;i++)
    {
        while(list[i]!=NULL)
        {

```

---

---

```
        arr[j++]=list[i]->data;
        list[i]=list[j]->link;
    }
}

printf("\nSorted list is:\n");
for(i=0;i<n;i++)
    printf("%d\t",arr[i]);

}

void main()
{
    int n, a[20], i;
    clrscr();
    printf("Enter size of the array:");
    scanf("%d",&n);
    printf("\nEnter array elements:\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    addcal(a,n);
}
```

---

---

```

printf("Enter the key element:");
scanf("%d",&item);

pos=interpol(item,a,0,n-1);
if(pos==-1)
    printf("\nItem not found");
else
    printf("\nItem found at %d",pos);
}

```

## HASHING

Hashing is a way of representing dictionaries. Dictionary is an abstract data type with a set of operations searching, insertion and deletion defined on its elements. The elements of dictionary can be numeric or characters or most of the times, records. Usually, a record consists of several fields; each may be of different data types. For example, student record may contain student id, name, gender, marks etc. Every record is usually identified by some **key**.

Here we will consider the implementation of a dictionary of  $n$  records with keys  $k_1, k_2 \dots k_n$ . Hashing is based on the idea of distributing keys among a one-dimensional array

$H[0 \dots m-1]$ , called **hash table**.

For each key, a value is computed using a predefined function called **hash function**. This function assigns an integer, called **hash address**, between 0 to  $m-1$  to each key. Based on the hash address, the keys will be distributed in a hash table.

For example, if the keys  $k_1, k_2, \dots, k_n$  are integers, then a hash function can be

$$h(K) = K \bmod m.$$

Let us take keys as 65, 78, 22, 30, 47, 89. And let hash function be,

$$h(k) = k \% 10.$$

Then the hash addresses may be any value from 0 to 9. For each key, hash address will be computed as –

$$h(65) = 65 \% 10 = 5$$

$$h(78) = 78 \% 10 = 8$$

$$h(22) = 22 \% 10 = 2$$

$$h(30) = 30 \% 10 = 0$$

$$h(47) = 47 \% 10 = 7$$

$$h(89) = 89 \% 10 = 9$$

Now, each of these keys can be hashed into a hash table as –

0	1	2	3	4	5	6	7	8	9
30		22			65		47	78	89

---

---

In general, a hash function should satisfy the following requirements:

- A hash function needs to distribute keys among the cells of hash table as evenly as possible.
- A hash function has to be easy to compute.

### Hash Collisions

Let us have  $n$  keys and the hash table is of size  $m$  such that  $m < n$ . As each key will have an address with any value between 0 to  $m-1$ , it is obvious that more than one key will have same hash address. That is, two or more keys need to be hashed into the same cell of hash table. This situation is called as **hash collision**.

In the worst case, all the keys may be hashed into same cell of hash table. But, we can avoid this by choosing proper size of hash table and hash function. Anyway, every hashing scheme must have a mechanism for resolving hash collision. There are two methods for hash collision resolution, viz.

- Open hashing
- closed hashing

### Open Hashing (or Separate Chaining)

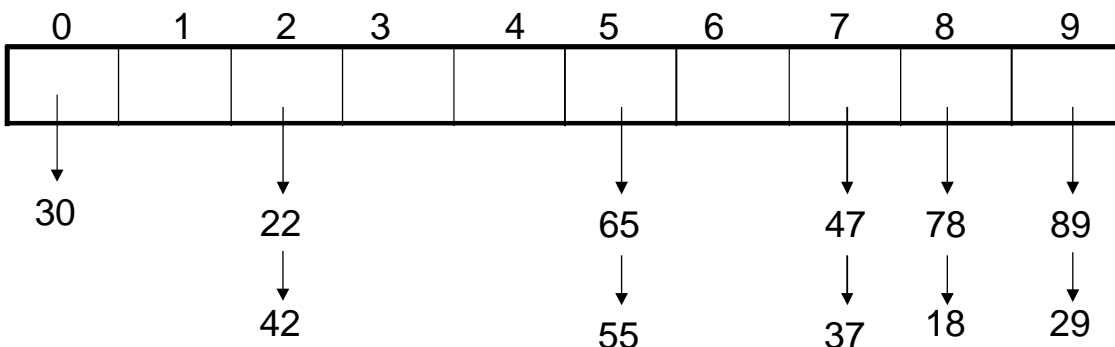
In open hashing, keys are stored in linked lists attached to cells of a hash table. Each list contains all the keys hashed to its cell. For example, consider the elements

65, 78, 22, 30, 47, 89, 55, 42, 18, 29, 37.

If we take the hash function as  $h(k) = k \% 10$ , then the hash addresses will be –

$h(65) = 65 \% 10 = 5$	$h(78) = 78 \% 10 = 8$
$h(22) = 22 \% 10 = 2$	$h(30) = 30 \% 10 = 0$
$h(47) = 47 \% 10 = 7$	$h(89) = 89 \% 10 = 9$
$h(55) = 55 \% 10 = 5$	$h(42) = 42 \% 10 = 2$
$h(18) = 18 \% 10 = 8$	$h(29) = 29 \% 10 = 9$
$h(37) = 37 \% 10 = 7$	

The hash table would be –





---

### Operations on Hashing:

- **Searching:** Now, if we want to search for the key element in a hash table, we need to find the hash address of that key using same hash function. Using the obtained hash address, we need to search the linked list by tracing it, till either the key is found or list gets exhausted.
- **Insertion:** Insertion of new element to hash table is also done in similar manner. Hash key is obtained for new element and is inserted at the end of the list for that particular cell.
- **Deletion:** Deletion of element is done by searching that element and then deleting it from a linked list.

### Closed Hashing (or Open Addressing)

In this technique, all keys are stored in the hash table itself without using linked lists. Different methods can be used to resolve hash collisions. The simplest technique is **linear probing**.

This method suggests to check the next cell from where the collision occurs. If that cell is empty, the key is hashed there. Otherwise, we will continue checking for the empty cell in a circular manner. Thus, in this technique, the hash table size must be at least as large as the total number of keys. That is, if we have  $n$  elements to be hashed, then the size of hash table should be greater or equal to  $n$ .

Example:

Consider the elements 65, 78, 18, 22, 30, 89, 37, 55, 42

Let us take the hash function as  $h(k) = k \% 10$ , then the hash addresses will be –

$h(65) = 65 \% 10 = 5$	$h(78) = 78 \% 10 = 8$
$h(18) = 18 \% 10 = 8$	$h(22) = 22 \% 10 = 2$
$h(30) = 30 \% 10 = 0$	$h(89) = 89 \% 10 = 9$
$h(37) = 37 \% 10 = 7$	$h(55) = 55 \% 10 = 5$
$h(42) = 42 \% 10 = 2$	

Since there are 9 elements in the list, our hash table should at least be of size 9. Here we are taking the size as 10.

Now, hashing is done as below –

0	1	2	3	4	5	6	7	8	9
30	89	22	42		65	55	37	78	18

### Drawbacks:

- Searching may become like a linear search and hence not efficient.
-