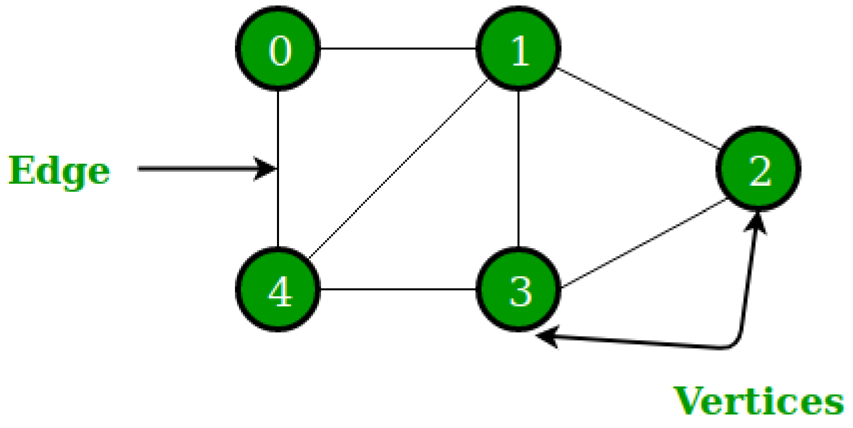
Graphs:

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as,

A Graph consists of a finite set of vertices (or nodes) and set of Edges which connect a pair of nodes.



In the above Graph, the set of vertices V = {0, 1, 2, 3, 4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.

Vertex

Each node of the graph is represented as a vertex.

Edge

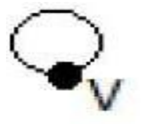
Edge represents a path between two vertices or a line between two vertices.

Path

Path represents a sequence of edges between the two vertices.

Loop

In a graph, if an edge is drawn from vertex to itself, it is called a loop.



In this example, we have a path from a vertex ‘V’ to itself. Such can be said as a loop.

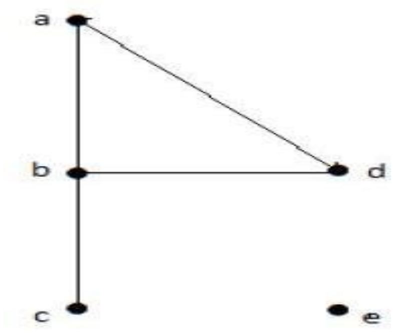
Degree of Vertex

It is the number of vertices adjacent to a vertex V.
Notation − deg (V)

Degree of vertex can be considered under two cases of graphs −

- Undirected Graph
- Directed Graph

Degree of Vertex in an Undirected Graph



In the above Undirected Graph,

- Deg (a) = 2, as there are 2 edges meeting at vertex ‘a’.
- Deg (b) = 3, as there are 3 edges meeting at vertex ‘b’.
- Deg(c) = 1, as there is 1 edge formed at vertex ‘c’
  So ‘c’ is a **pendent vertex**.
- Deg (d) = 2, as there are 2 edges meeting at vertex ‘d’.
- Deg (e) = 0, as there are 0 edges formed at vertex ‘e’.
  So ‘e’ is an **isolated vertex**.

Degree of Vertex in a Directed Graph

In a directed graph, each vertex has an **in degree** and an **out degree**.

In degree

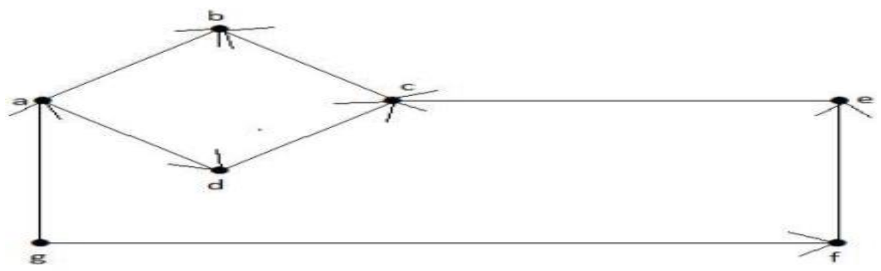
- In degree of vertex V is the number of edges which are coming into the vertex V.
- **Notation** − $\deg^{-}$ (V).

Out degree

- Out degree of vertex V is the number of edges which are going out from the vertex V.
- **Notation** − $\deg^{+}$ (V).

Consider the following example:

Take a look at the following directed graph. Vertex ‘a’ has two edges, ‘ad’ and ‘ab’, which are going outwards. Hence its out degree is 2. Similarly, there is an edge ‘ga’, coming towards vertex ‘a’. Hence the in degree of ‘a’ is 1.



The in degree and out degree of other vertices are shown in the following table −

Vertex	In degree	Out degree
a	1	2
b	2	0
c	2	1
d	1	1
e	1	1
f	1	1
g	0	2

Pendent Vertex

By using degree of a vertex, we have a two special types of vertices. A vertex with degree one is called a pendent vertex.

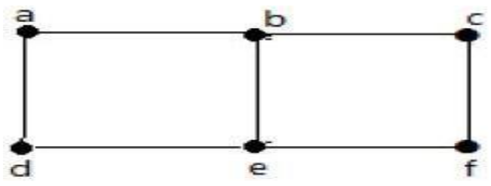
Isolated Vertex

A vertex with degree zero is called an isolated vertex.

Adjacency

- In a graph, two vertices are said to be **adjacent,** if there is an edge between the two vertices. Here, the adjacency of vertices is maintained by the single edge that is connecting those two vertices.

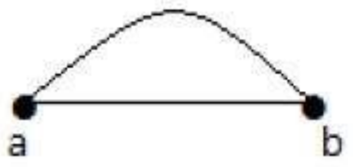
- In a graph, two edges are said to be **adjacent**, if there is a common vertex between the two edges. Here, the adjacency of edges is maintained by the single vertex that is connecting two edges.



In the above graph −

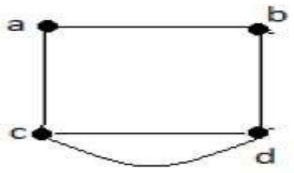
- 'a' and 'b' are the adjacent vertices, as there is a common edge 'ab' between them.
- 'a' and 'd' are the adjacent vertices, as there is a common edge 'ad' between them.
- ab' and 'be' are the adjacent edges, as there is a common vertex 'b' between them.
- 'be' and 'de' are the adjacent edges, as there is a common vertex 'e' between them.

Parallel Edges



In a graph, if a pair of vertices is connected by more than one edge, then those edges are called parallel edges.

Multi Graph



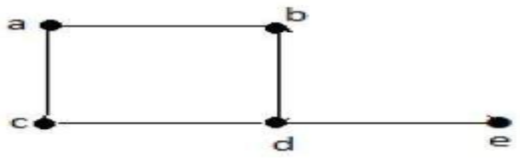
A graph having parallel edges or self-loops or both is known as a Multigraph.

Directed Acyclic Graph:

A graph with no cycles is called a DAG. A graph with no cycles is called a **Forest**.

Degree Sequence of a Graph

If the degrees of all vertices in a graph are arranged in descending or ascending order, then the sequence obtained is known as the degree sequence of the graph.



In the above graph, for the vertices {d, a, b, c, e}, the degree sequence is {3, 2, 2, 2, 1}.The below table gives a better understanding. ‘

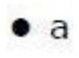
Vertex	a	b	c	d	e
Connecting to	b,c	a,d	a,d	c,b,e	D
Degree	2	2	2	3	1

Null Graph

A **graph having no edges** is called a Null Graph.

In the above graph, there are three vertices named ‘a’, ‘b’, and ‘c’, but there are no edges among them. Hence it is a Null Graph.

Trivial Graph



A **graph with only one vertex** is called a Trivial Graph.

In the above shown graph, there is only one vertex ‘a’ with no other edges. Hence it is a trivial graph.

Simple Graph

A graph **with no loops** and **no parallel edges** is called a simple graph.

- The maximum number of edges possible in a single graph with ‘n’ vertices is ${}^nC_2$ where ${}^nC_2 = n(n-1)/2$.
- The number of simple graphs possible with ‘n’ vertices = $2^{{}^nc_2} = 2^{n(n-1)/2}$.

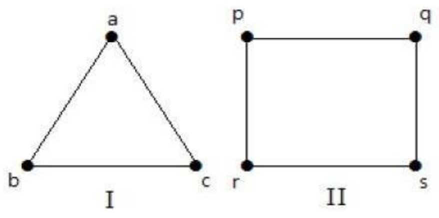
Connected Graph

A graph G is said to be connected **if there exists a path between every pair of vertices**. There should be at least one edge for every vertex in the graph. So that we can say that it is connected to some other vertex at the other side of the edge.

Disconnected Graph

A graph G is disconnected, if it does not contain at least two connected vertices.

Regular Graph



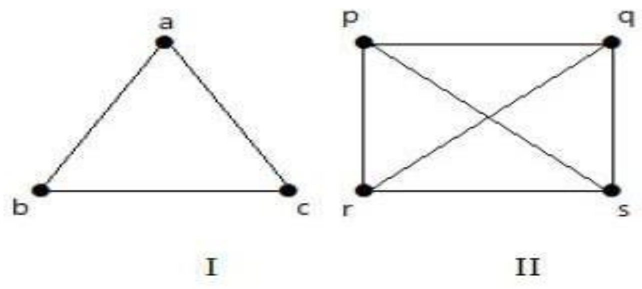
A graph G is said to be regular, **if all its vertices have the same degree**. In a graph, if the degree of each vertex is 'k', then the graph is called a 'k-regular graph'.

Complete Graph

A simple graph with 'n' mutual vertices is called a complete graph and it is **denoted by 'K_n'**. In the graph, **a vertex should have edges with all other vertices,** then it called a complete graph.

In other words, if a vertex is connected to all other vertices in a graph, then it is called a complete graph.

In the following graphs, each vertex in the graph is connected with all the remaining vertices in the graph except by itself.



In graph I,

	a	**b**	**c**
a	Not Connected	Connected	Connected
b	Connected	Not Connected	Connected
c	Connected	Connected	Not Connected

In graph II,

	p	q	r	s
p	Not Connected	Connected	Connected	Connected
q	Connected	Not Connected	Connected	Connected
r	Connected	Connected	Not Connected	Connected
s	Connected	Connected	Connected	Not Connected

Cycle Graph

A simple graph with ‘n’ vertices (n >= 3) and ‘n’ edges is called a cycle graph if all its edges form a cycle of length ‘n’.

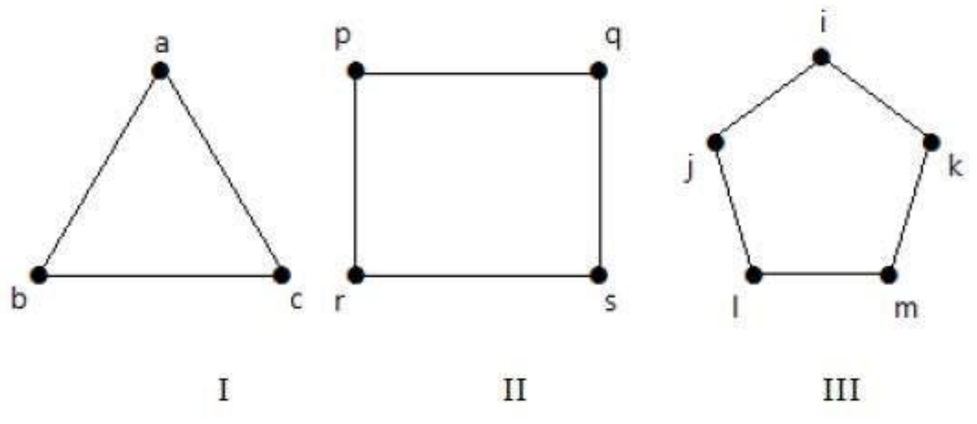
If the **degree of each vertex in the graph is two,** then it is called a Cycle Graph.

Notation − C_n

Example

Take a look at the following graphs −

- Graph I has 3 vertices with 3 edges which is forming a cycle ‘ab-bc-ca’.
- Graph II has 4 vertices with 4 edges which is forming a cycle ‘pq-qs-sr-rp’.
- Graph III has 5 vertices with 5 edges which is forming a cycle ‘ik-km-ml-lj-ji’.



Acyclic Graph

A graph **with no cycles** is called an acyclic graph.

Representation of Graphs

There are mainly two ways to represent a graph −

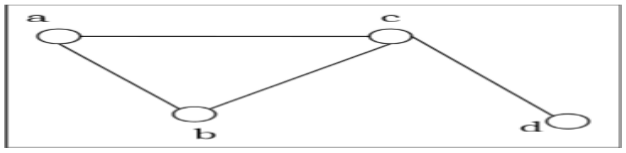
- Adjacency Matrix
- Adjacency List

Adjacency Matrix

An Adjacency Matrix A [V] [V] is a 2D array of size V×V where V is the number of vertices in an undirected graph. If there is an edge between Vx to Vy then the value of A [Vx] [Vy] =1 and A [Vy] [Vx] =1(this is because it is an undirected graph and if it is possible to traverse from Vx to Vy, the reverse also is possible), otherwise the value will be zero. And for a directed graph, if there is an edge between Vx to Vy, then the value of A [Vx] [Vy] =1, otherwise the value will be zero.

Adjacency Matrix of an Undirected Graph

Let us consider the following undirected graph and construct the adjacency matrix −

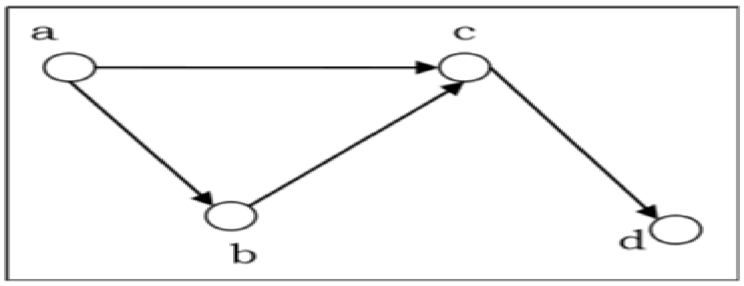


Adjacency matrix of the above undirected graph will be

	a	**b**	**c**	**d**
a	0	1	1	0
b	1	0	1	0
c	1	1	0	1
d	0	0	1	0

Adjacency Matrix of a Directed Graph

Let us consider the following directed graph and construct its adjacency matrix −



Adjacency matrix of the above directed graph will be −

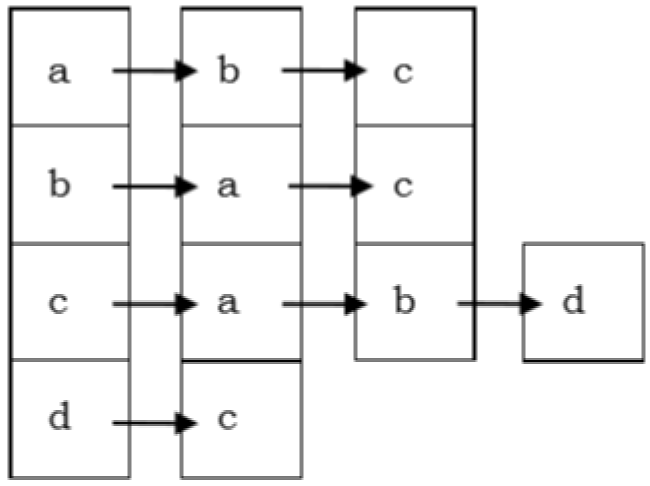
	a	**b**	**c**	**d**
a	0	1	1	0
b	0	0	1	0
c	0	0	0	1
d	0	0	0	0

***Pros**:* Representation is easier to implement and follow. Removing an edge takes O (1) time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done O(1).

***Cons**:* Consumes more space O (V^2). Even if the graph is sparse (contains less number of edges), it consumes the same space. Adding a vertex is O (V^2) time.

Adjacency List

In adjacency list, an array (A [V]) of linked lists is used to represent the graph G with V number of vertices. An entry $A\ [V_x]$ A [Vx] represents the linked list of vertices adjacent to the 'Vx' vertex. The adjacency list of the undirected graph is as shown in the figure below −



Simply considering the same figure as the above undirected graph, an array of list is considered. Now as we can see the Vertex ‘a’ is connected to vertices ‘b’ & ‘c’. Similarly all the other vertices are connected and they are represented as an array of lists as represented above.

Graph Traversal - DFS

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

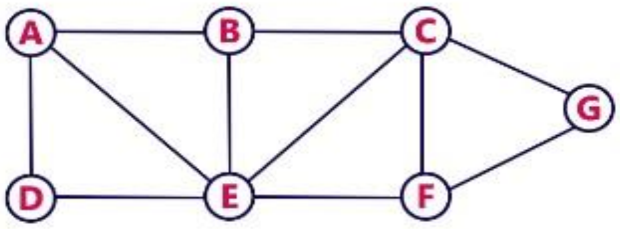
Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

---

Back tracking is coming back to the vertex from which we reached the current vertex.

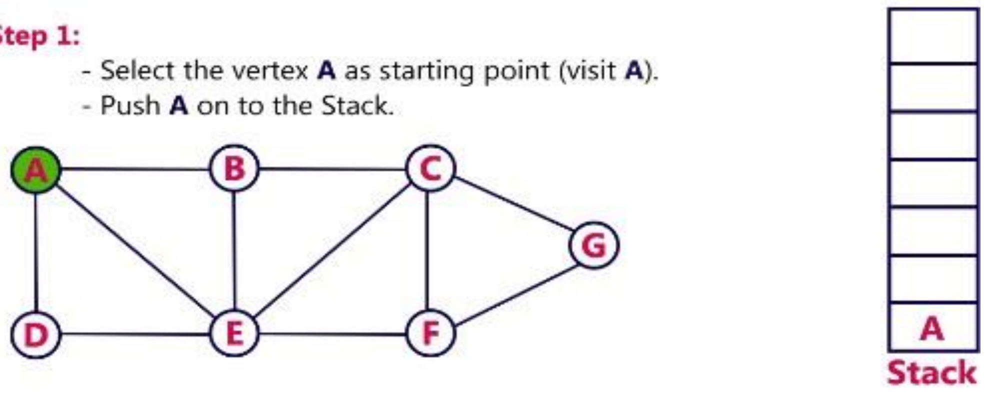
Example:

Consider the following example graph to perform DFS traversal



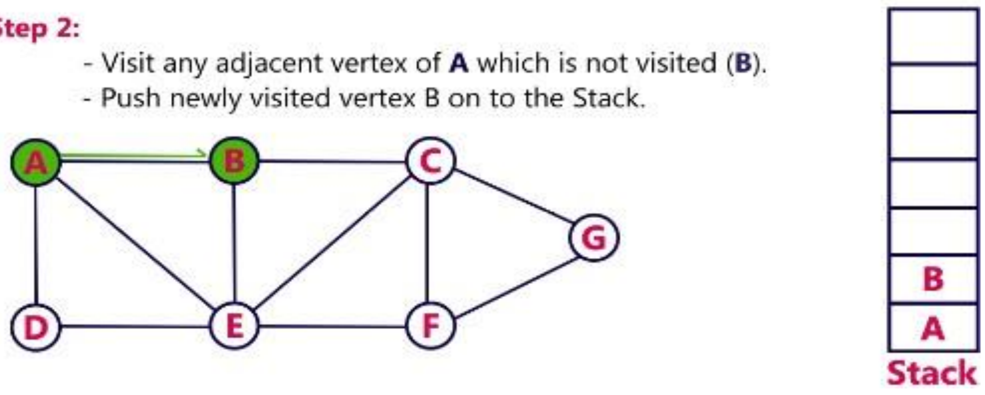
Step 1:

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



Step 2:

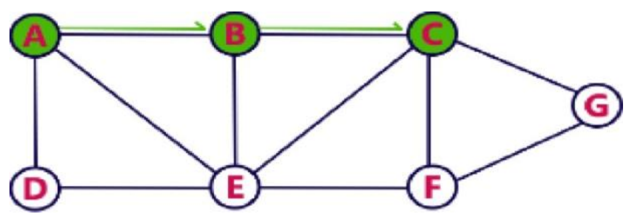
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

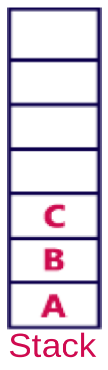


Step 3:

Visit a ny adja cent vertext of B which is not visited (C).

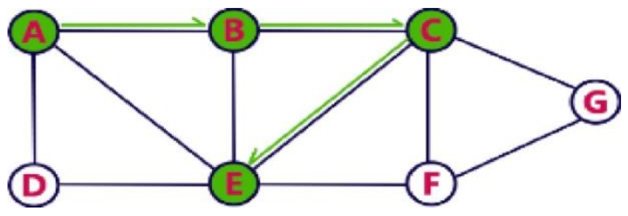
- Push C on to the Stack

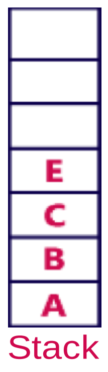




Step 4:

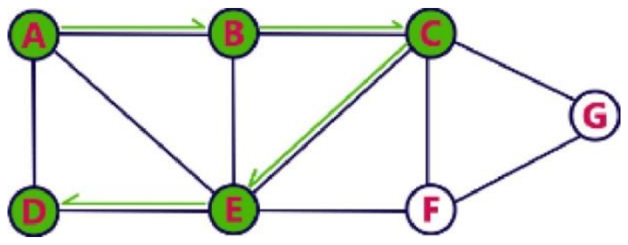
- Visit any adjacent vertext of C w hich is not VISit 'd (E).
  Push E an to the Stack

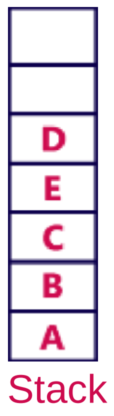




Step 5:

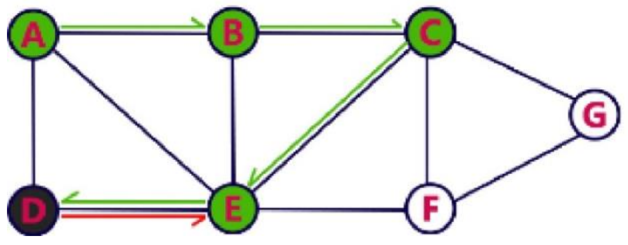
- Visit any adjacent vertext of E which is not visited (D).
- Push D on to the Stack

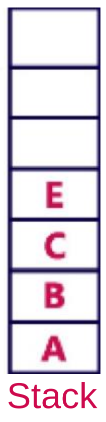




Step 6:

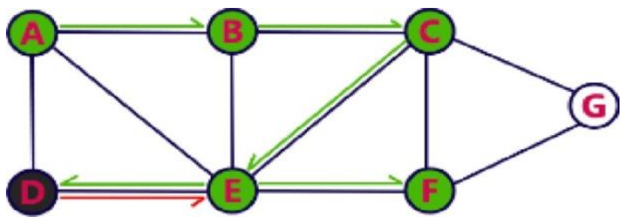
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

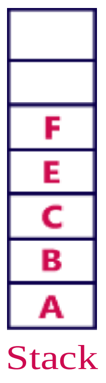




Step 7:

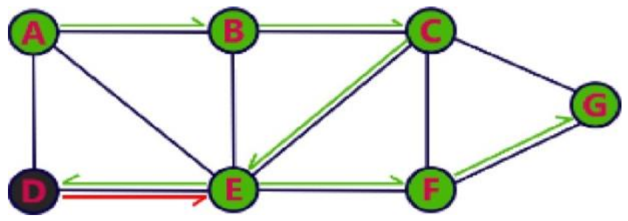
- Vi3 it any adjacent *ve rtex* of E which in n ot visited (F).
- Push F on to the Stack.

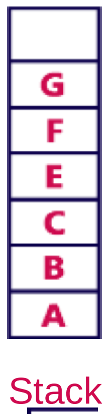




Step 8:

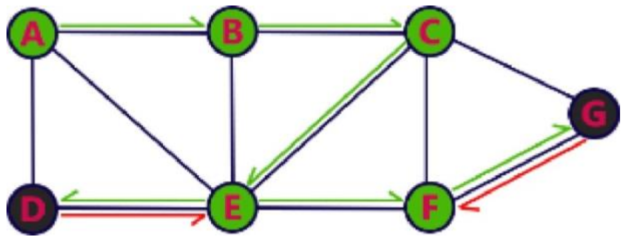
- Visit any adjacent vertex of E which is not visited (G).
- Push G on to the Stack.

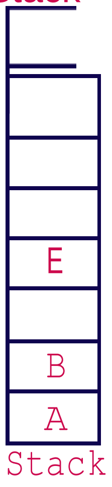




Step 9:

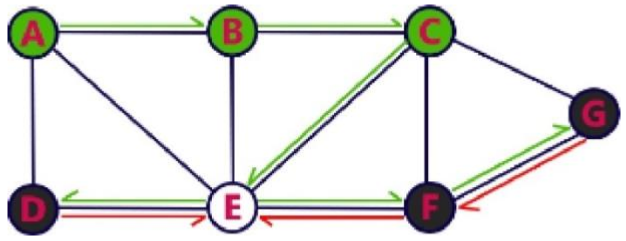
- There is no new vertiex to be visited from G. So use back track.
- Pnp G frnm the Stark.

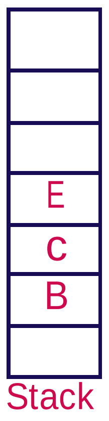




S1ep 10:

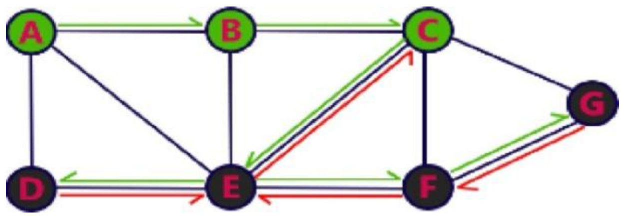
- Tt›ere is no new v+rtiex to be visited from F. So use back track.
- Pop F from the Stack.

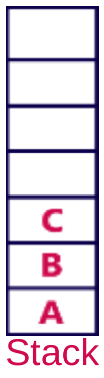




Step 11:

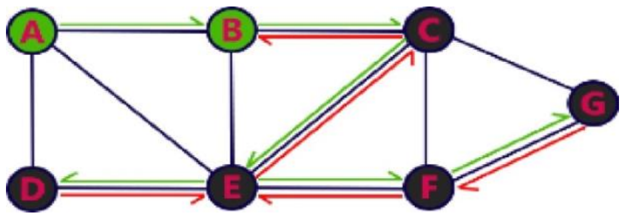
- There is no new vortiex to be visited from E. So use back track.
- Pop E from the Stack.

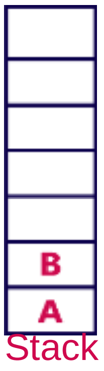




Step LZ:

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

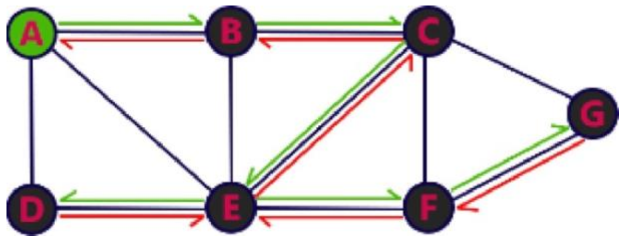


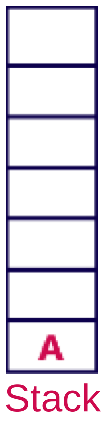


Step 13:

There is no new vertiex to be visited from B. So use back track.

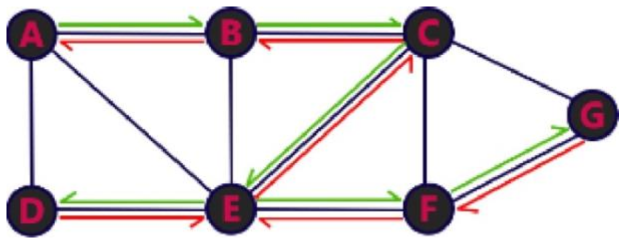
- Pop B from the Stack.

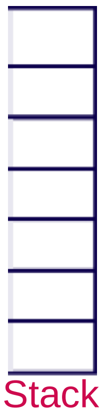




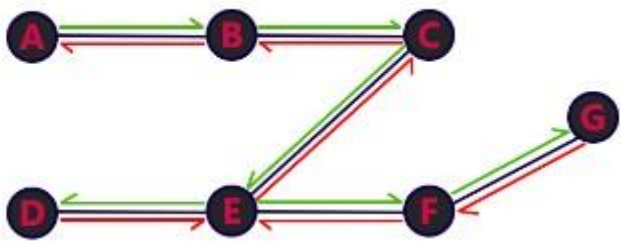
Step 14:

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





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



BFS (Breadth First Search)

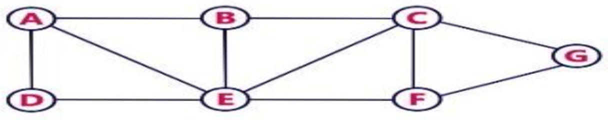
BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

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

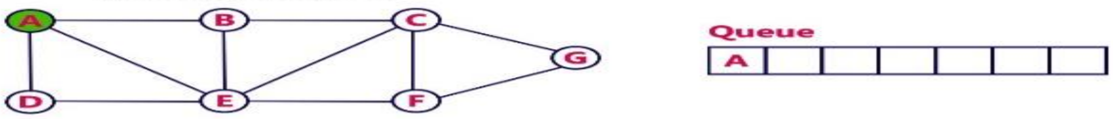
- **Step 1 -** Define a Queue of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3 -** Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue 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:

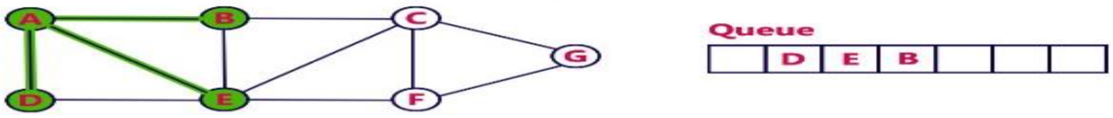
Consider the following example graph to perform BFS traversal



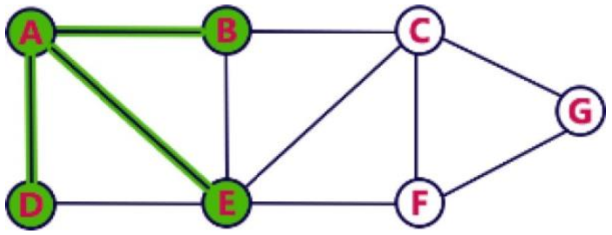
Step 1:
- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



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..



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

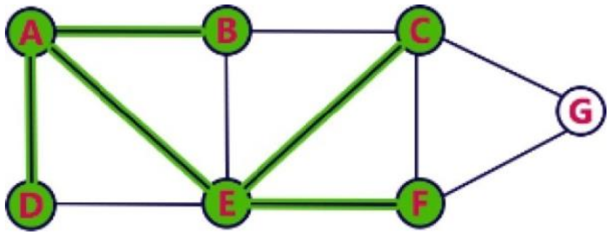


Queue

EJB

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.

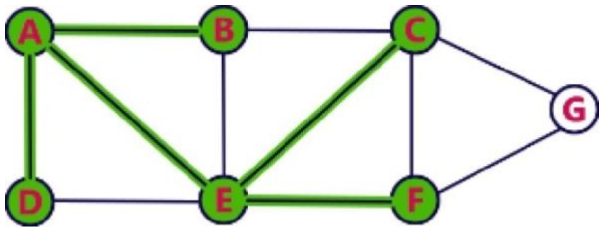


Queue

QB C F

Step 5:

- Visit all adjacent vertices of B which are nat visited (there is no vertex).
- Delete B from the Queue.

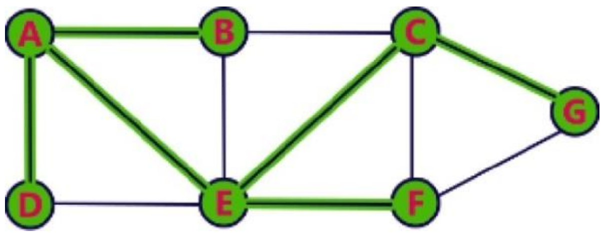


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of C which are nat visited (G).
- Insert newly visited vertex into the Queue and delete C from the Queue.

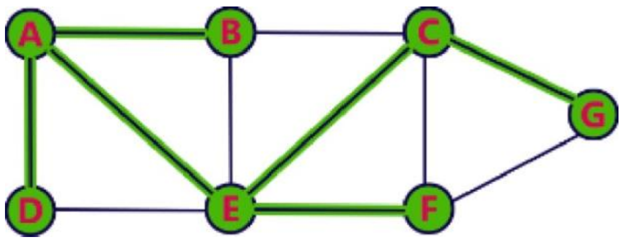


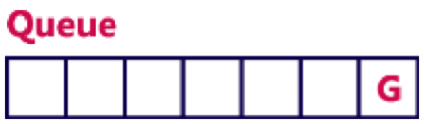
Queue

					F	G

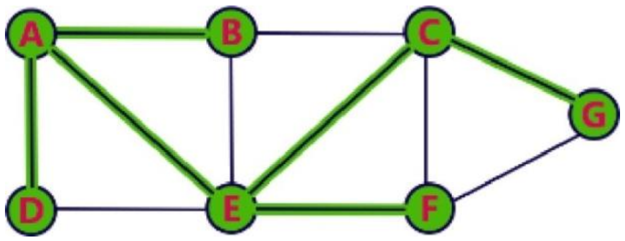
Step 7:

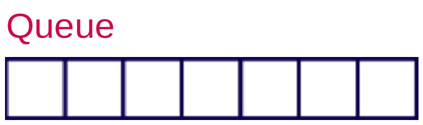
- Visit all adjacent vertices of F which are nat vi5ited (there is no vertex).
- Delete F from the Queue.



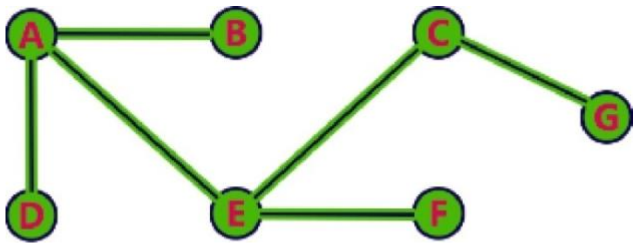


- Visit all adjacent vertices of G which are not visited (there is no vertex).
- Delete G from the Queue.





- Queue becan\e End pty. So, stop the BFS |uroCess.
  Final result of BFS is a Spa nning Tree as shown below...

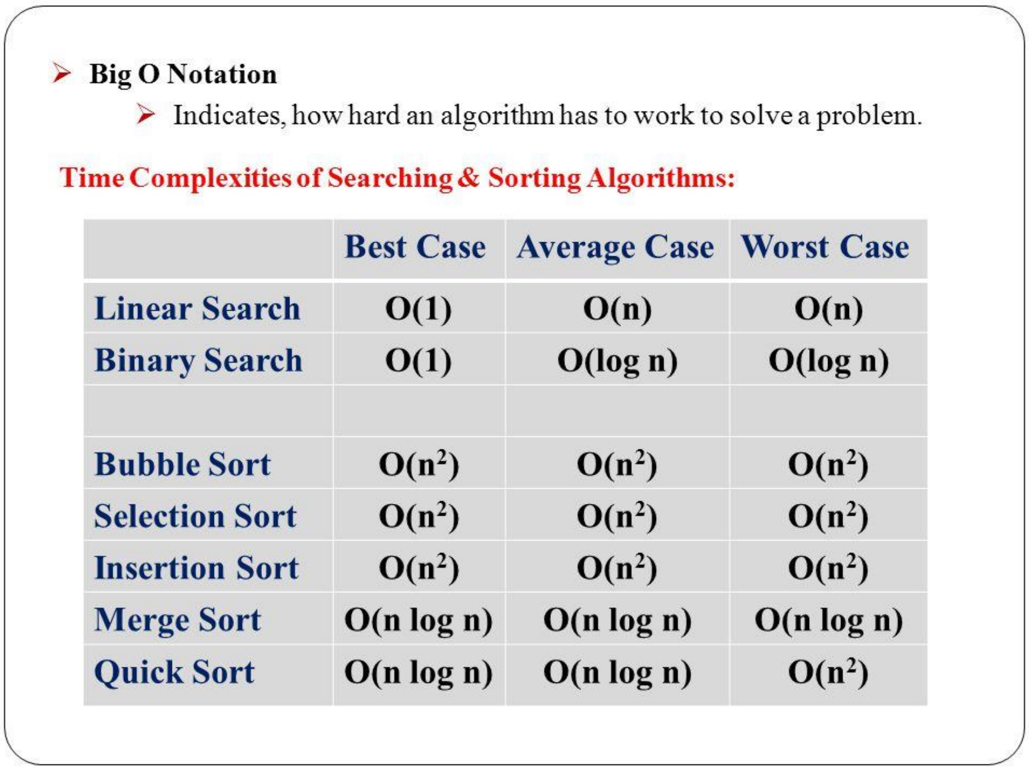


Applications of Graph Theory

Graph theory has its applications in diverse fields of engineering −

- **Electrical Engineering** − the concepts of graph theory is used extensively in designing circuit connections. The types or organization of connections are named as topologies. Some examples for topologies are star, bridge, series, and parallel topologies.
- **Computer Science** − Graph theory is used for the study of algorithms. For example,
    - Kruskal's Algorithm
    - Prim's Algorithm
    - Dijkstra's Algorithm
- **Computer Network** − the relationships among interconnected computers in the network follows the principles of graph theory.
- **Science** − the molecular structure and chemical structure of a substance, the DNA structure of an organism, etc., are represented by graphs.
- **Linguistics** − the parsing tree of a language and grammar of a language uses graphs.
- **General** − Routes between the cities can be represented using graphs. Depicting hierarchical ordered information such as family tree can be used as a special type of graph called tree.

UNIT – V (Introduction to Algorithms & Searching and Sorting)



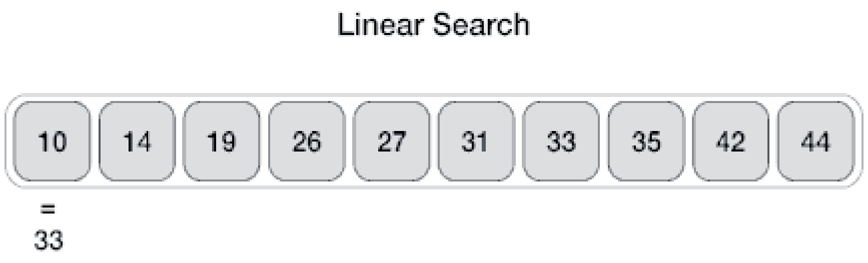
- **Big O Notation**
  - Indicates, how hard an algorithm has to work to solve a problem.

Time Complexities of Searching & Sorting Algorithms:

	Best Case	Average Case	Worst Case
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$

Linear search

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection. Time complexity of linear search is O(n).



Algorithm

Linear Search (Array A, Value x)

```
Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit
```

```
//linear search
#include<stdio.h>
#include<stdlib.h>
```

```
void  main()
{
      int a[50],n,i,element,flag=0;
      printf("\n Enter no-of elements:");
      scanf("%d",&n);
      printf("\n Enter array elements:");
```

```
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
       printf("\n Enter element to be searched:");
     scanf("%d",&element);
             for(i=0;i<n;i++)
             {
                 if(a[i]==element)
                  {
                          printf("\n element is found at index: %d\n",i);
                           flag=1;
                           break;
                          }
                  }
             if(flag==0)
                           printf("\n element not found");
   }
```

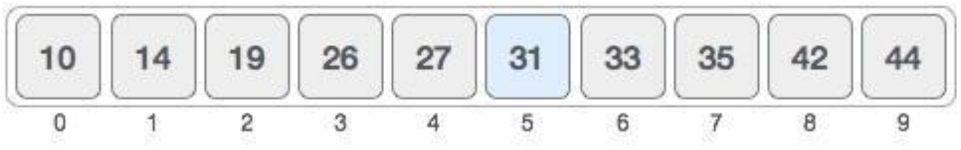
Binary search

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

How Binary Search Works?

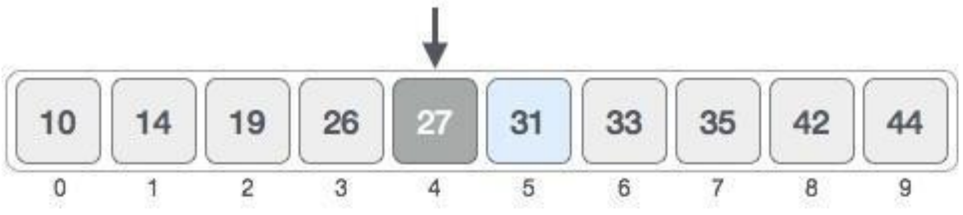
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



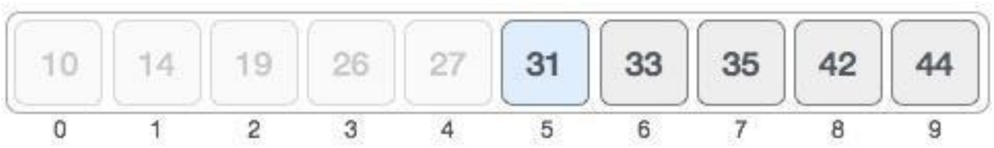
First, we shall determine half of the array by using this formula −

n=10;
Low=0;
High=n-1;
mid = (low + high) / 2

Here it is, (0 + 9)/ 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



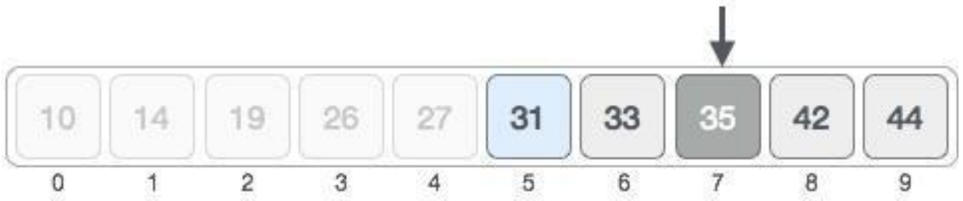
Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.



We change our low to mid + 1 and find the new mid value again. low

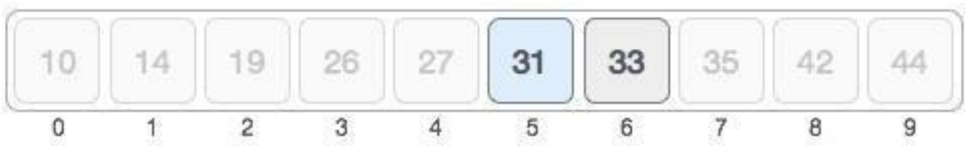
= mid + 1
mid = (low + high) / 2

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

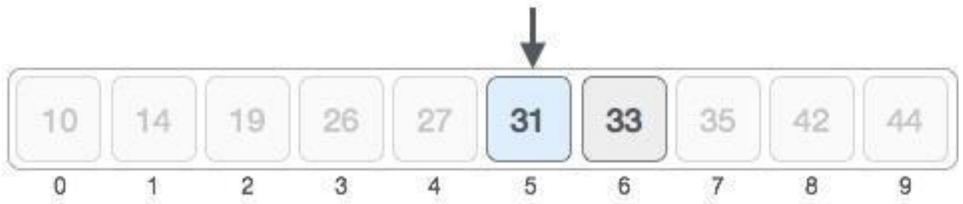


High=mid-1=7-1=6;

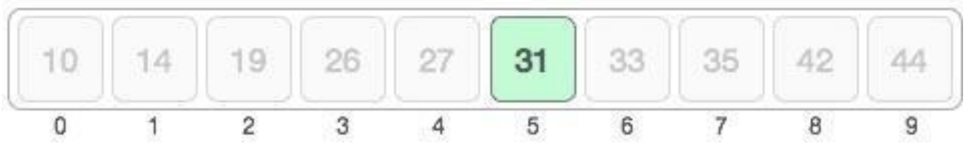
The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers

<u>Search for x=20</u>
<u>Iteration1`:</u>
L=0 ; h=9; m=4;
Here x<a[4] so h=m-1=4-1=3;
No change in l. i.e l=0;

Iteration2`:
 L=0; h=3; m=1;
 Here x>a[1] so l=m+1=1+1=2;
No change in h=3;
Iteration3`:
 L=2;h=3;m=2;
 Here x>a[2] so l=m+1=2+1=3
No change in h=3;
Iteration4`:
 L=3;h=3;m=3;
 Here x<a[3] so h=m-1=3-1=2;
No change in l=3.
Since L>h loop terminates. Here element not found.

```
//binary search
#include<stdio.h>
#include<stdlib.h>
void  main()
{

     int      a[50],n,i,element,flag=0,l,m,h;
     printf("\n   Enter   no-of   elements:");
     scanf("%d",&n);
             printf("\n   Enter   array   elements:");
       for(i=0;i<n;i++)
       scanf("%d",&a[i]);

       printf("\n Enter element to be searched:");
               scanf("%d",&element);
                l=0;h=n-1;
         while(l<=h)
          {
                            m=(l+h)/2;
                if(element==a[m])
                {
                   printf("\n element is found at index: %d ",m);
                   flag=1; break;
                               }
                              else if(element<a[m])
                                       h=m-1;
                              else
                                       l=m+1;
                   }

               if(flag==0)
                 printf("\n Element not found");

   }
```

BUBBLE SORT

In Bubble sort, Each element of the array is compared with its adjacent element. The algorithm processes the list in passes. A list with n elements requires n-1 passes for sorting. Consider an array

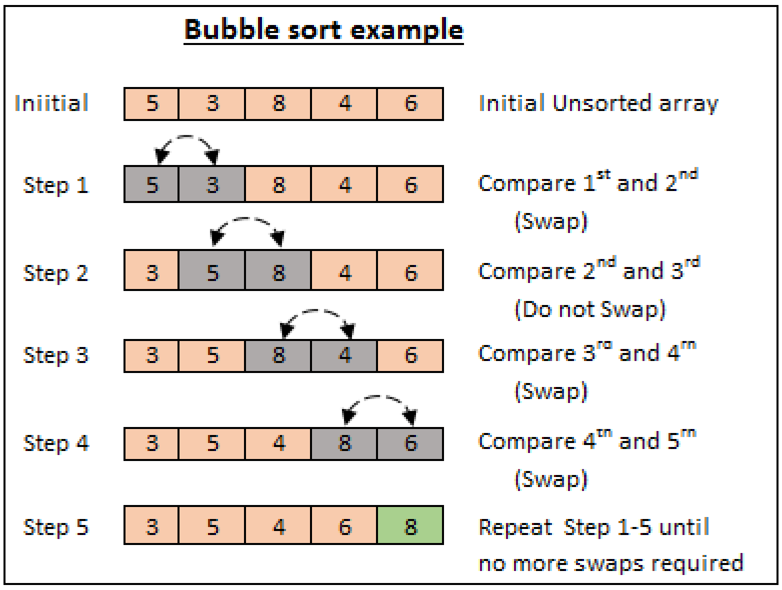
A of n elements whose elements are to be sorted by using Bubble sort. The algorithm processes like following.

1. In Pass 1, A[0] is compared with A[1], A[1] is compared with A[2], A[2] is compared with A[3] and so on. At the end of pass 1, the largest element of the list is placed at the highest index of the list.
2. In Pass 2, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of Pass 2 the second largest element of the list is placed at the second highest index of the list.
3. In pass n-1, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of this pass. The smallest element of the list is placed at the first index of the list.

Algorithm :

- **Step 1**: Repeat Step 2 For i = 0 to N-1
- **Step 2**: Repeat For J = i + 1 to N - I
- **Step 3**: IF A[J] > A[i]
  SWAP A[J] and A[i]
  [END OF INNER LOOP]
  [END OF OUTER LOOP
- **Step 4**: EXIT

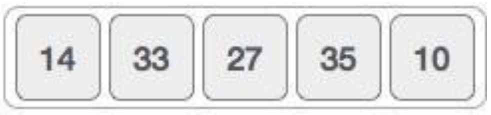
first pass:


Bubble sort example
Iniitial 5 3 8 4 6 Initial Unsorted array
Step 1 5 3 8 4 6 Compare 1st and 2nd (Swap)
Step 2 3 5 8 4 6 Compare 2nd and 3rd (Do not Swap)
Step 3 3 5 8 4 6 Compare 3rd and 4th (Swap)
Step 4 3 5 4 8 6 Compare 4th and 5th (Swap)
Step 5 3 5 4 6 8 Repeat Step 1-5 until no more swaps required

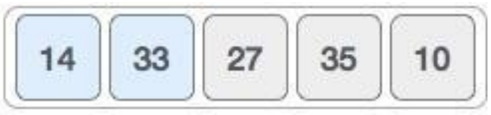
Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

How Bubble Sort Works?

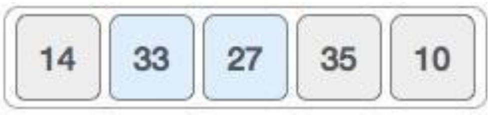
We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.



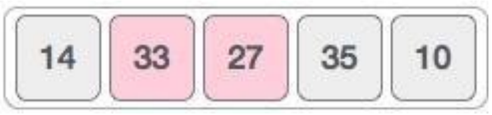
Bubble sort starts with very first two elements, comparing them to check which one is greater.



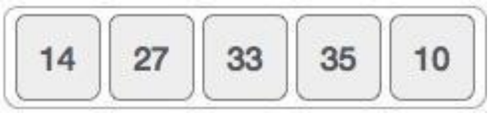
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



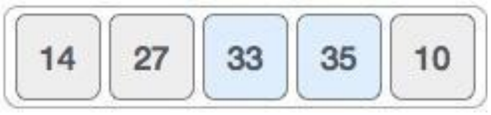
We find that 27 is smaller than 33 and these two values must be swapped.



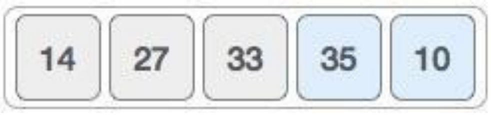
The new array should look like this −



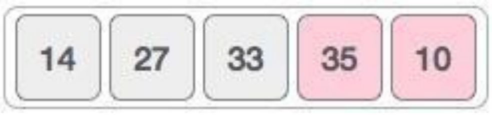
Next we compare 33 and 35. We find that both are in already sorted positions.



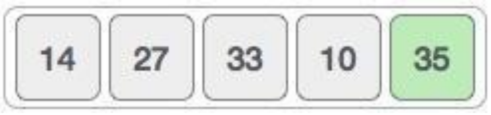
Then we move to the next two values, 35 and 10.



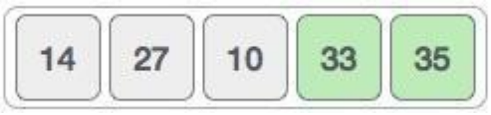
We know then that 10 is smaller 35. Hence they are not sorted.



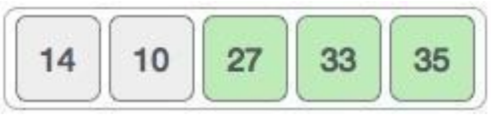
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −



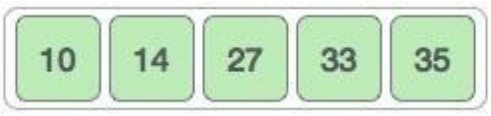
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −



Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



45,23,32,10,16
(J=0;j<n-1;j++) ~~(J=0;j<n-i-1;j++)~~

Pass1: i=0

23, 45, 32, 10, 16
23, 32, 45, 10, 16
23, 32, 10, 45, 16
23, 32, 10, 16, 45

Pass2: i=1
23, 32, 10, 16, 45
23, 10.32, 16, 45
23, 10, 16, 32, 45
~~23, 10, 16, 32, 45~~

Pass 3: i=2
10, 23, 16, 32, 45
10, 16, 23, 32, 45
~~10, 16, 23, 32, 45~~
~~10, 16, 23, 32, 45~~

Pass4: i=3
10, 16, 23, 32, 45
~~10, 16, 23, 32, 45~~
~~10, 16, 23, 32, 45~~
~~10, 16, 23, 32, 45 (sorted list)~~

```
// C program for Bubble sort
#include <stdio.h>
void main()
 {
  int a[100], i, j, n,  temp;

  printf("\nEnter number of elements\n");
  scanf("%d",  &n);
  printf("\nEnter elements:");
    for(i=0;i<n;i++)
    scanf("%d",  &a[i]);
    printf("\nBefore sorting:\n");
   for(i=0;i<n;i++)
```

```
    printf("%d\t",  a[i]);

   for(i=0;i<(n-1);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 in ascending order:\n");
   for(i=0;i<n;i++)
    printf("%d\t",  a[i]);

  }
```

Selection Sort

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.

First, find the smallest element of the array and place it on the first position. Then, find the second smallest element of the array and place it on the second position. The process continues until we get the sorted array. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

The array with n elements is sorted by using n-1 pass of selection sort algorithm.

- In 1st pass, smallest element of the array is to be found along with its index **pos**. then, swap A[0] and A[pos]. Thus A[0] is sorted, we now have n -1 elements which are to be sorted.
- In 2nd pas, position pos of the smallest element present in the sub-array A[n-1] is found. Then, swap, A[1] and A[pos]. Thus A[0] and A[1] are sorted, we now left with n-2 unsorted elements.
- In n-1th pass, position pos of the smaller element between A[n-1] and A[n-2] is to be found. Then, swap, A[pos] and A[n-1].

Therefore, by following the above explained process, the elements A[0], A[1], A[2], ..., A[n- 1] are sorted.

Example

Consider the following array with 6 elements. Sort the elements of the array by using selection sort.

A = {10, 2, 3, 90, 43, 56}.

Pass	Pos	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

1, i=0	1	2	10	3	90	43	56
2, i=1	2	2	3	10	90	43	56
3, i=2	2	2	3	10	90	43	56
4, i=3	4	2	3	10	43	90	56
5, i=4	5	**2**	**3**	**10**	**43**	**56**	**90**

Sorted A = {2, 3, 10, 43, 56, 90}

Example2:

34 67 21 55 37

Pass1:

I=0, pos=2
Swap(34, 21)
21 67 34 55 37

Pass2:

I=1, pos=2
Swap(67, 34)
21 34 67 55 37

Pass3:

I=2, pos=4
Swap(67, 37)
21 34 37 55 67

Pass 4:

I=3, pos=3
Swap(55, 55)
21 34 37 55 67 (sorted list final)

Algorithm

SELECTION SORT(ARR, N)

- **Step 1**: Repeat Steps 2 and 3 for K = 1 to N-1
- **Step 2**: CALL SMALLEST(ARR, K, N, POS)
- **Step 3**: SWAP A[K] with ARR[POS]
  [END OF LOOP]
- **Step 4**: EXIT

SMALLEST (ARR, K, N, POS)

- **Step 1**: [INITIALIZE] SET SMALL = ARR[K]
- **Step 2**: [INITIALIZE] SET POS = K

- **Step 3**: Repeat for J = K+1 to N -1
```
IF SMALL > ARR[J]
SET SMALL = ARR[J]
SET POS = J
[END OF IF]
[END OF LOOP]
```
- **Step 4**: RETURN POS

```
// C program for selection sort
#include <stdio.h>
void main()
 {
  int a[100], i, j, n, temp,  pos;

  printf("\nEnter number of elements\n");
  scanf("%d",  &n);
  printf("\nEnter elements:");
    for(i=0;i<n;i++)
    scanf("%d",  &a[i]);
    printf("\nBefore sorting:\n");
   for(i=0;i<n;i++) printf("%d\t",
   a[i]);

  for(i=0;i<n-1;i++)
     {
      pos=i;
        for(j=i+1;j<n;j++)
        {
          if(a[j]<a[pos])
            pos=j;
        }
          temp=a[i];
          a[i]=a[pos];
          a[pos]=temp;
      }

   printf("\nSorted list in ascending order:\n");
   for(i=0;i<n;i++)
    printf("%d\t",  a[i]);


 }
```

Insertion Sort

Insertion sort is the simple sorting algorithm which is commonly used in the daily lives while ordering a deck of cards. In this algorithm, we insert each element onto its proper place in the sorted array. This is less efficient than the other sort algorithms like quick sort, merge sort, etc.

Technique

Consider an array A whose elements are to be sorted. Initially, A[0] is the only element on the sorted set. In pass 1, A[1] is placed at its proper index in the array.

In pass 2, A[2] is placed at its proper index in the array. Likewise, in pass n-1, A[n-1] is placed at its proper index into the array.

To insert an element A[k] to its proper index, we must compare it with all other elements i.e. A[k- 1], A[k-2], and so on until we find an element A[j] such that, A[j]<=A[k].

All the elements from A[k-1] to A[j] need to be shifted and A[k] will be moved to A[j+1].

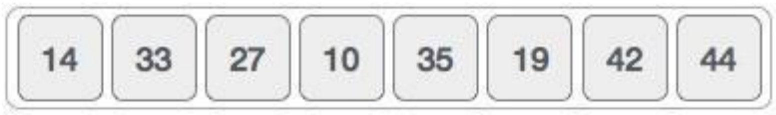
Algorithm

- **Step 1**: Repeat Steps 2 to 5 for K = 1 to N-1
- **Step 2**: SET TEMP = ARR[K]
- **Step 3**: SET J = K - 1
- **Step 4**: Repeat while TEMP <=ARR[J]
  SET ARR[J + 1] = ARR[J]
  SET J = J - 1
  [END OF INNER LOOP]
- **Step 5**: SET ARR[J + 1] = TEMP
  [END OF LOOP]
- **Step 6**: EXIT

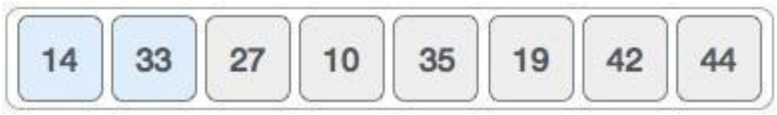
This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n^2), where **n** is the number of items.

How Insertion Sort Works?

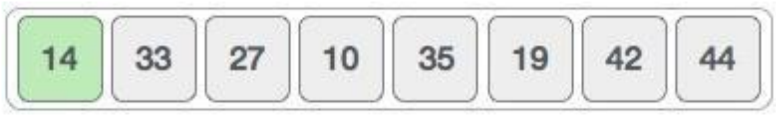
We take an unsorted array for our example.



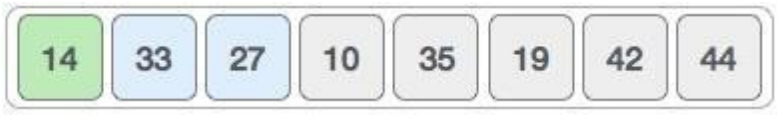
Insertion sort compares the first two elements.



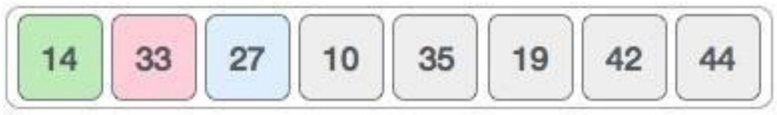
It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



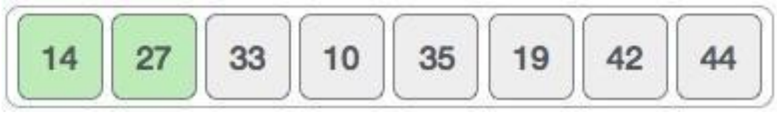
Insertion sort moves ahead and compares 33 with 27.



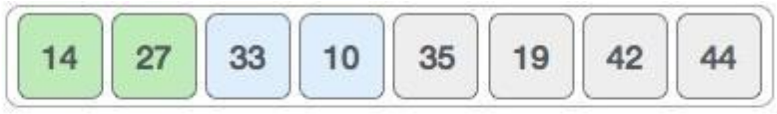
And finds that 33 is not in the correct position.



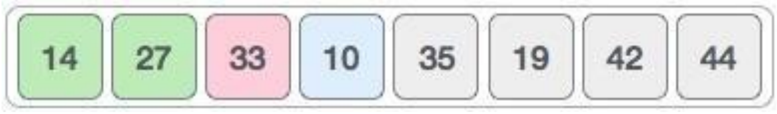
It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



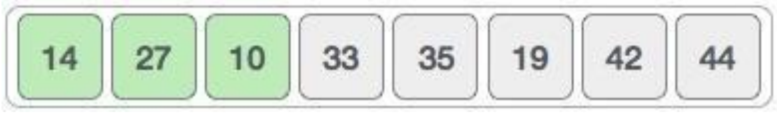
By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



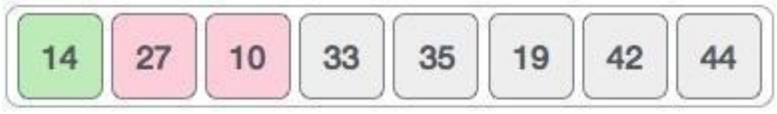
These values are not in a sorted order.



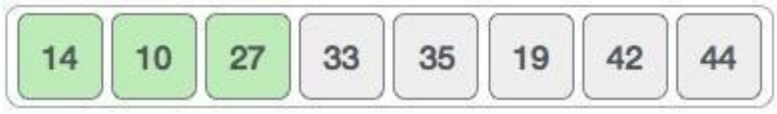
So we swap them.



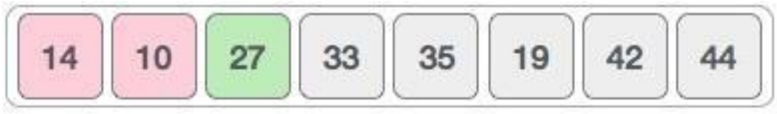
However, swapping makes 27 and 10 unsorted.



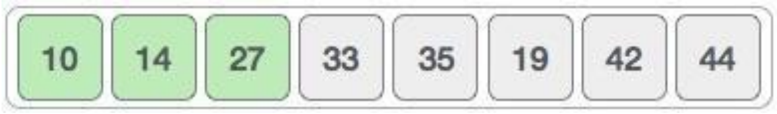
Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

```
// C program for insertion sort
#include <stdio.h>
void main()
```

```
{
 int a[100],i,j,n,temp;

 printf("\nEnter number of elements\n");
 scanf("%d", &n);
 printf("\nEnter elements:");
   for(i=0;i<n;i++)
   scanf("%d", &a[i]);
   printf("\nBefore sorting:\n");
  for(i=0;i<n;i++)
  printf("%d\t",a[i]);

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


}
```

Example2:
4, 2, 5, 1, 3

Pass 1:

I=1, temp=a[1]=2, j=0
A[1]=a[0]=4 inner loop
A[0]=2 outer loop
2, 4, 5, 1, 3

Pass 2:

I=2, temp=a[2]=5, j=1 Condition
false in inner loop A[2]=5 outer
loop
2, 4, 5, 1, 3

Pass 3:

I=3, temp=a[3]=1, j=2
A[3]=a[2] A[2]=a[1] A[1]=a[0]
inner loop A[0]=1
outer loop 1,2, 4, 5,
3

Pass 4:

I=4, temp=a[4]=3, j=3
  A[4]=a[3] A[3]=a[2]
  inner loop
A[2]=3 outer loop
1 , 2, 3 , 4, , 5 (SORTED LIST)

Heap Sort

- Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.
- A Binary Heap is a complete Binary Tree where items are stored in a special order such that value in a parent node is greater (or smaller) than the values in its two children nodes. If the parent node is greater than the children nodes it is called a max heap and if the parent node is smaller than the children nodes it is called a min heap. The heap can be represented by binary tree or array.
- We use an array because Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).

Procedure:

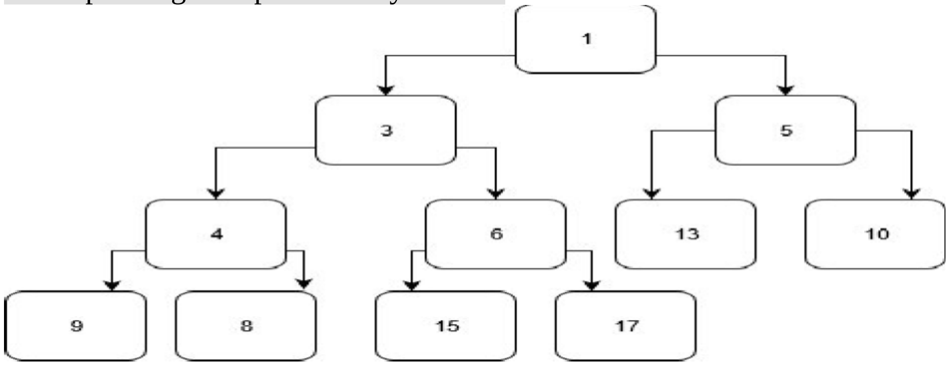
- Initially build a max heap of elements in Array.(Refer 2nd point for what is a Max Heap)
- The root element, that is Array [1], will contain maximum element of Array (since it is a max heap).
- After that, swap this element with the last element of Array and heapify the max heap excluding the last element which is already in its correct position and then decrease the length of heap by one.
- Repeat the step 2, until all the elements are in their correct position.

Building a Max heap:

Consider an array,

Array = {1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17}

Corresponding Complete Binary Tree is:

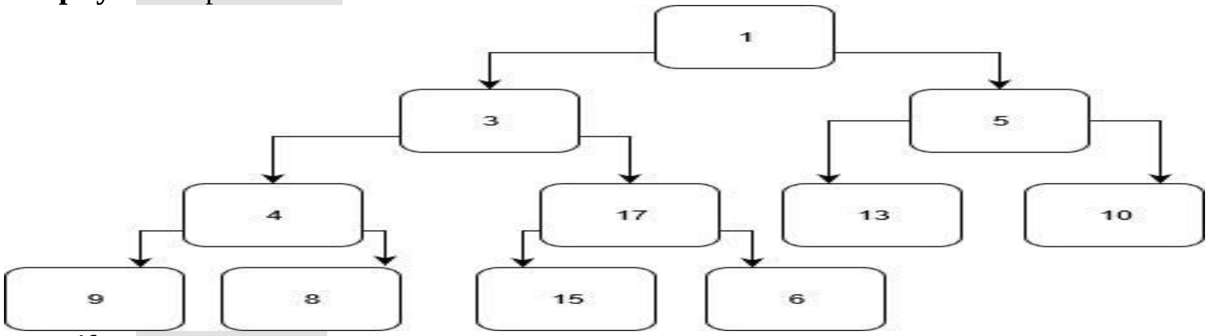


The task to build a Max-Heap from above array

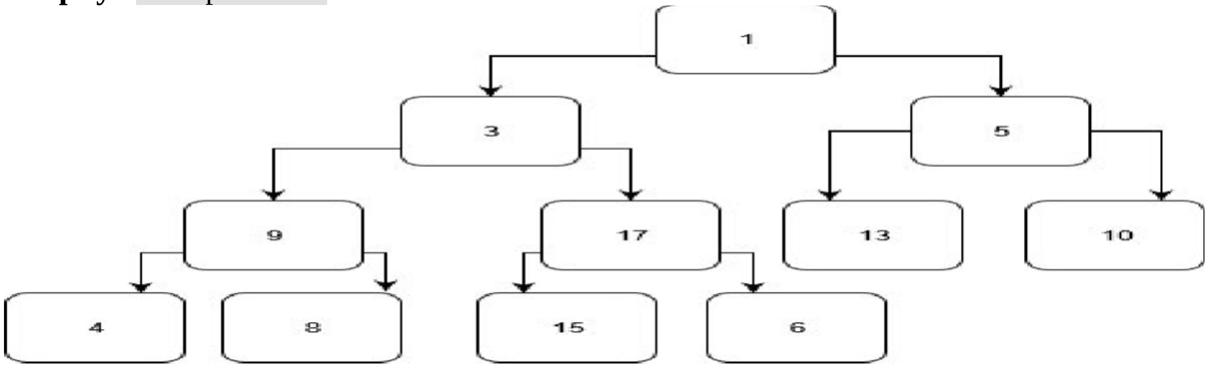
Total Nodes = 11.

To build the heap, heapify only the nodes: [1, 3, 5, 4, 6] in **reverse order** as the rest of the nodes are leaf nodes [13, 10, 9, 8, 15, 17].

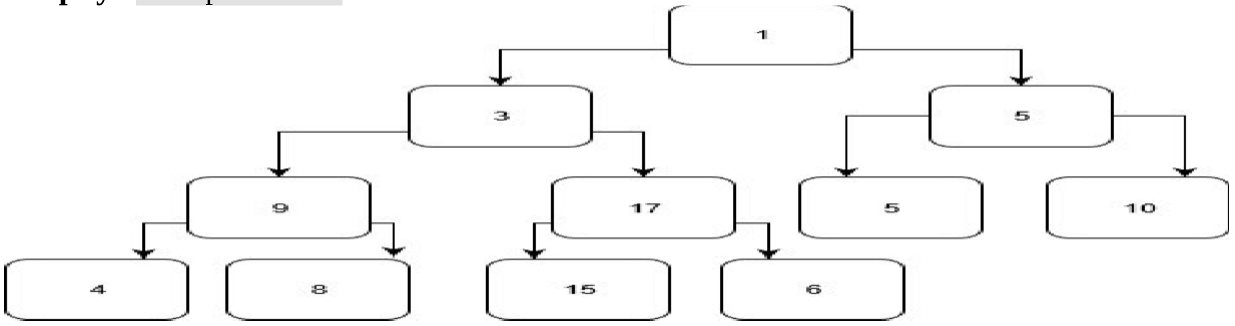
Heapify 6: Swap 6 and 17.



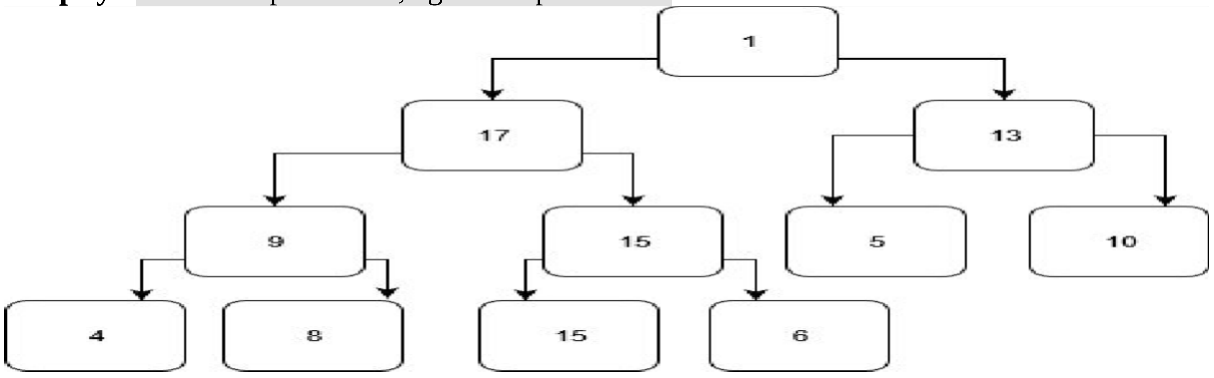
Heapify 4: Swap 4 and 9.



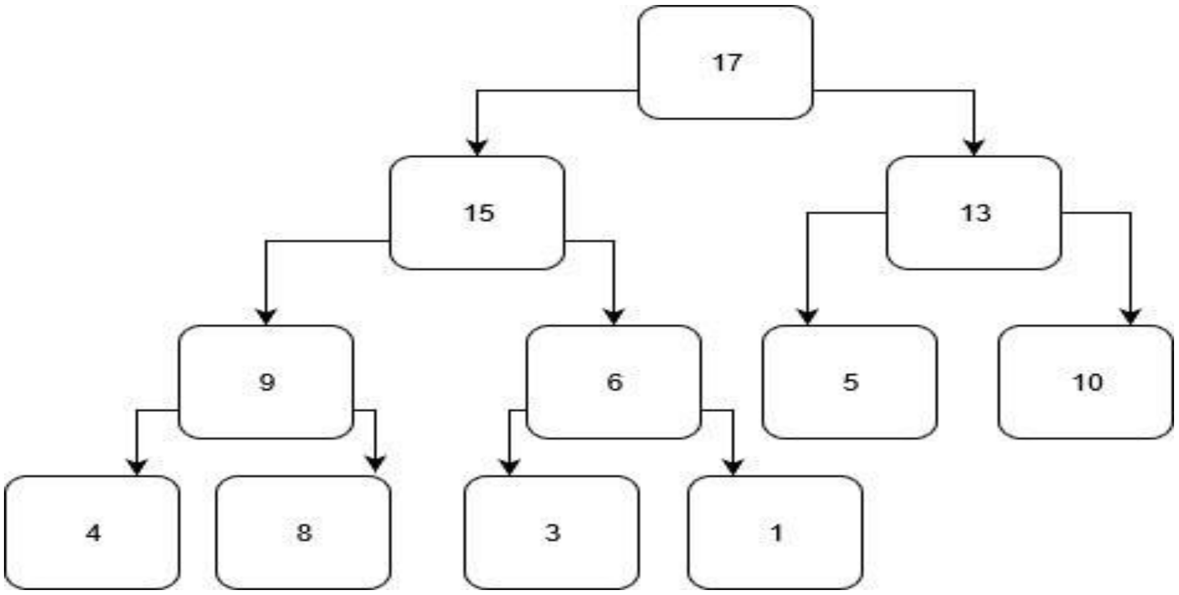
Heapify 5: Swap 13 and 5.



Heapify 3: First Swap 3 and 17, again swap 3 and 15.



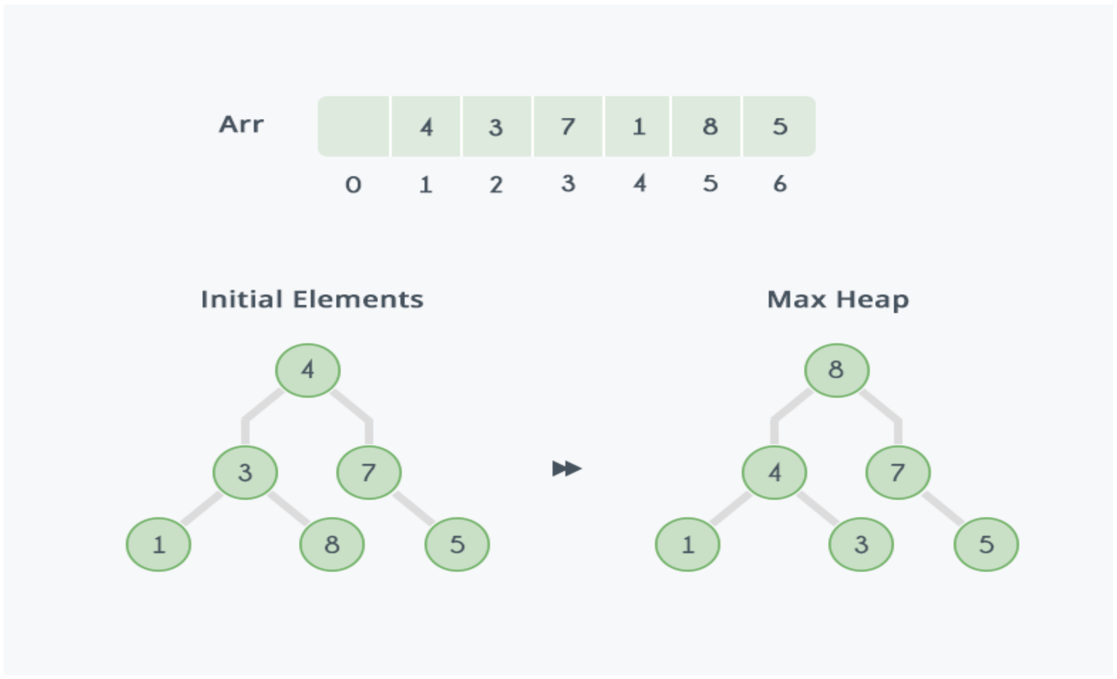
Heapify 1: First Swap 1 and 17, again swap 1 and 15,
Finally swap 1 and 6.



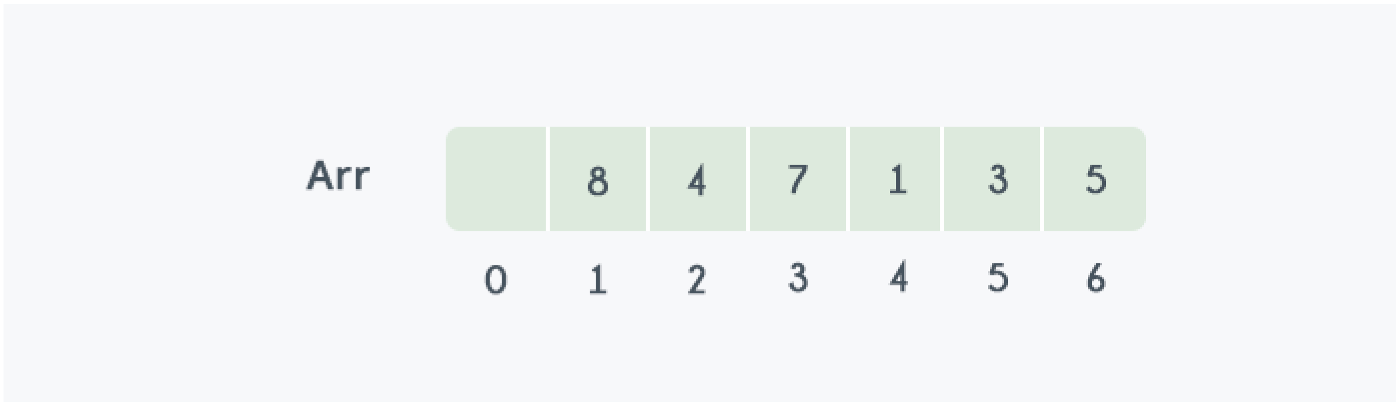
Hence the elements are arranged in a max heap.

Example for heap sort:

In the diagram below, initially there is an unsorted array having 6 elements and then max-heap will be built.

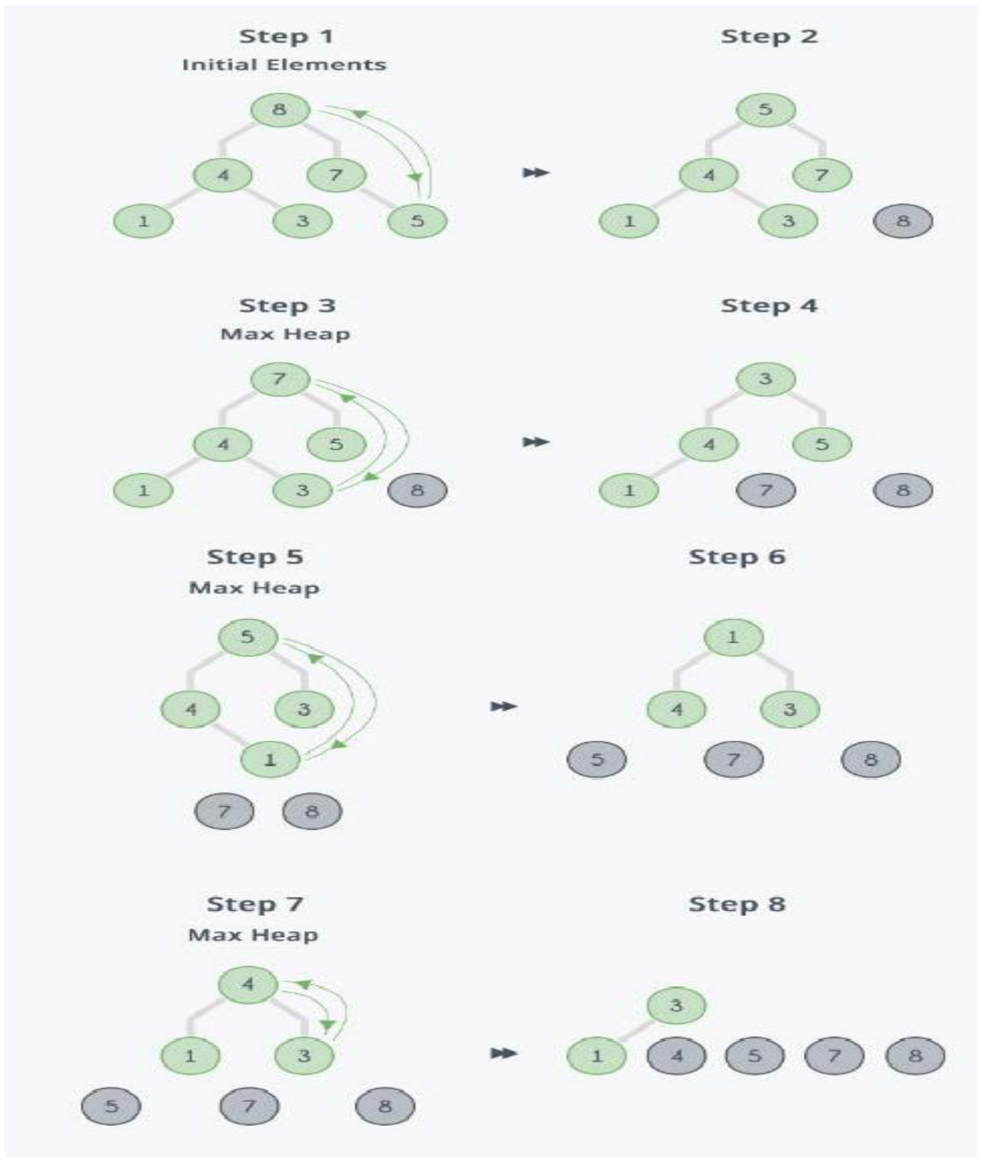


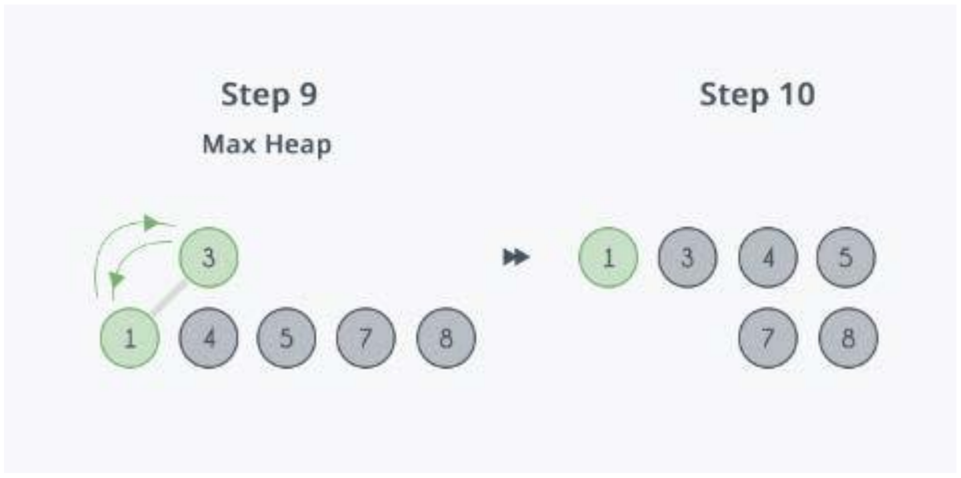
After building max-heap, the elements in the array will be:



Step 1: 8 is swapped with 5.
Step 2: 8 is disconnected from heap as 8 is in correct position now.
Step 3: Max-heap is created and 7 is swapped with 3.
Step 4: 7 is disconnected from heap.
Step 5: Max heap is created and 5 is swapped with 1.
Step 6: 5 is disconnected from heap.
Step 7: Max heap is created and 4 is swapped with 3.
Step 8: 4 is disconnected from heap.
Step 9: Max heap is created and 3 is swapped with 1.
Step 10: 3 is disconnected.

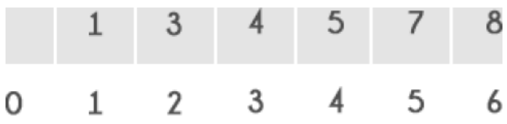
After all the steps, we will get a sorted array.





After all the steps, we will get a sorted array.

Arr



External Sorting

External sorting is a term for a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive). External sorting typically uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub-files are combined into a single larger file.

One example of external sorting is the external merge sort algorithm, which sorts chunks that each fit in RAM, then merges the sorted chunks together. We first divide the file into **runs** such that the size of a run is small enough to fit into main memory. Then sort each run in main memory using merge sort sorting algorithm. Finally merge the resulting runs together into successively bigger runs, until the file is sorted.

Merge Sort

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sub list consists of a single element and merging those sub lists in a manner that results into a sorted list.

Idea:

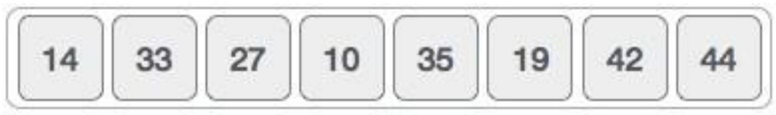
- Divide the unsorted list into N sub lists, each containing 1 element.
- Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. N will now convert into N/2 lists of size 2.
- Repeat the process till a single sorted list of obtained.

Time Complexity**:**

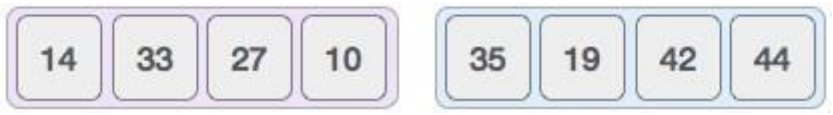
The list of size N is divided into a max of logN parts, and the merging of all sub lists into a single list takes O (N) time, the worst case run time of this algorithm is O (NLogN).

How Merge Sort Works?

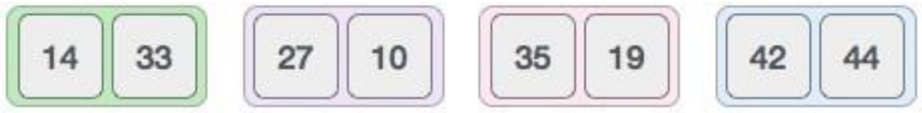
To understand merge sort, we take an unsorted array as the following −



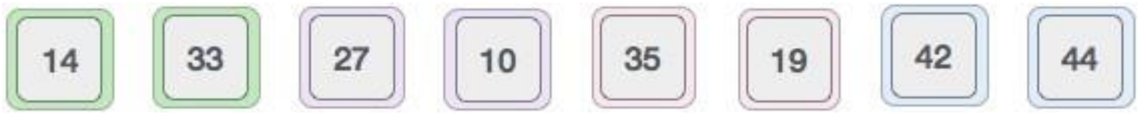
We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

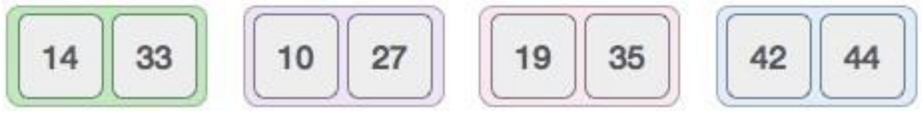


We further divide these arrays and we achieve atomic value which can no more be divided.

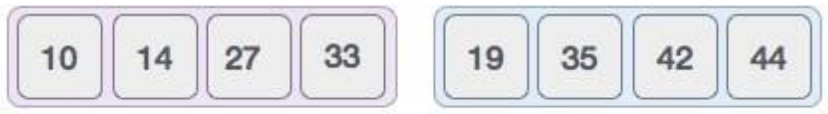


Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

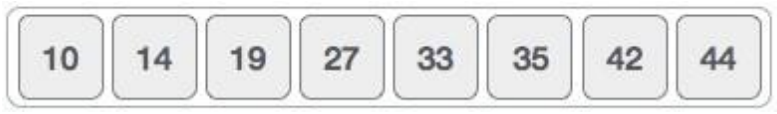
We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



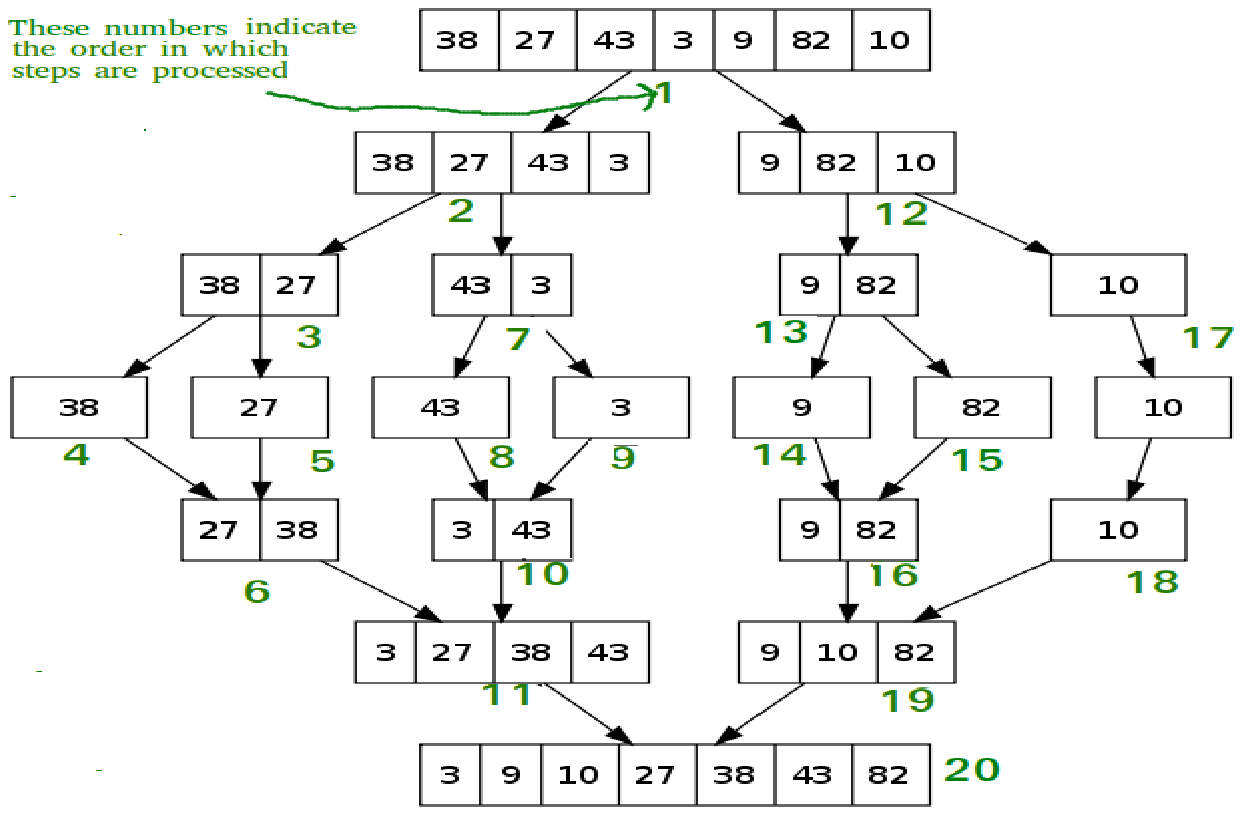
In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



After the final merging, the list should look like this −



Consider another example,



Applications of Merge Sort

- Merge sort is useful for sorting linked lists in O (nlogn) time.
- Inversion Count Problem.
- Used in External Sorting.