UNIT-II

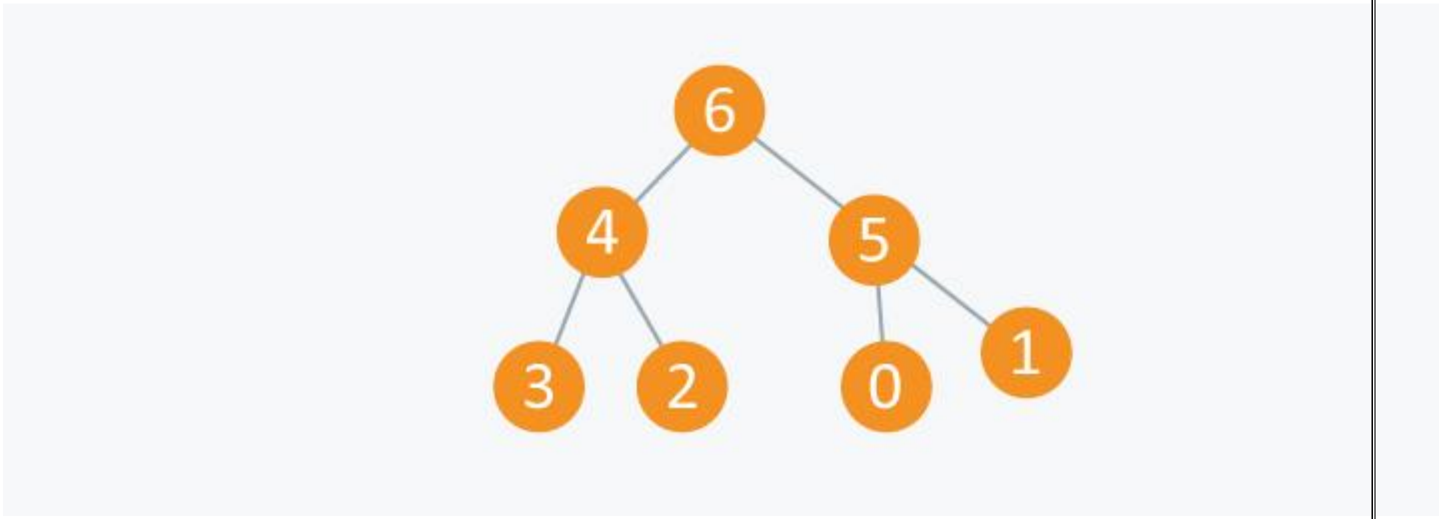
Heap Trees (Priority Queues) – Min and Max Heaps, Operations and Heap Sort .

Graphs – Terminology, Representations, Basic Search and Traversals, topological sort.

Heaps

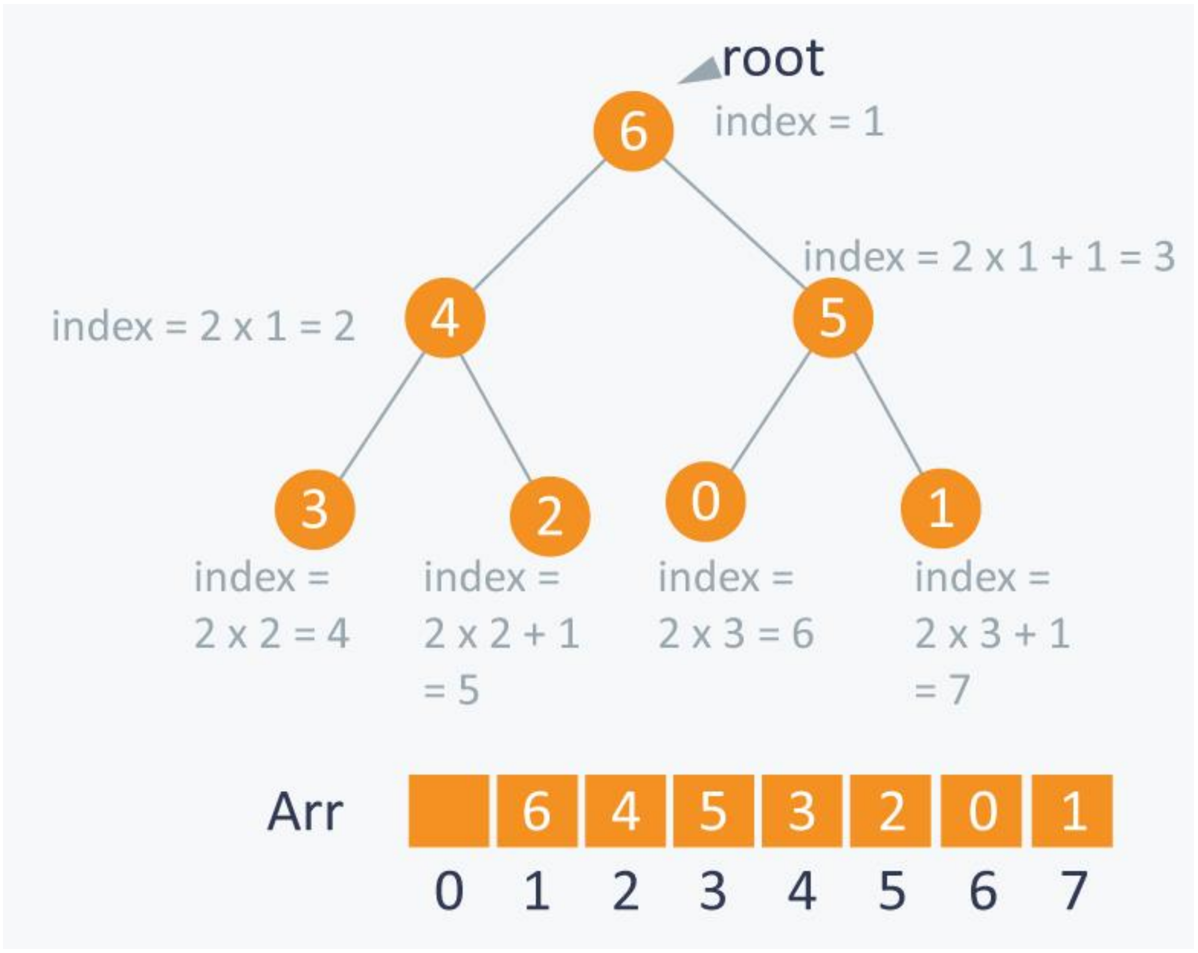
A heap is a tree-based data structure in which all the nodes of the tree are in a specific order.

For example, if X is the parent node of Y, then the value of X follows a specific order with respect to the value of Y and the same order will be followed across the tree.
The maximum number of children of a node in a heap depends on the type of heap. However, in the more commonly-used heap type, there are at most 2 children of a node and it's known as a Binary heap.
In binary heap, if the heap is a complete binary tree with N nodes, then it has smallest possible height which is log2N .



In the diagram above, you can observe a particular sequence, i.e each node has greater value than any of its children.

Suppose there are N Jobs in a queue to be done, and each job has its own priority. The job with maximum priority will get completed first than others. At each instant, we are completing a job with maximum priority and at the same time we are also interested in inserting a new job in the queue with its own priority.
So at each instant we have to check for the job with maximum priority to complete it and also insert if there is a new job. This task can be very easily executed using a heap by considering N jobs as N nodes of the tree.
As you can see in the diagram below, we can use an array to store the nodes of the tree. Let's say we have 7 elements with values {6, 4, 5, 3, 2, 0, 1}.
Note: An array can be used to simulate a tree in the following way. If we are storing one element at index i in array Arr, then its parent will be stored at index i/2 (unless its a root, as root has no parent) and can be accessed by Arr[i/2], and its left child can be accessed by Arr[2∗i] and its right child can be accessed by Arr[2∗i+1]. Index of root will be 1 in an array.



There can be two types of heap:

Max Heap: In this type of heap, the value of parent node will always be greater than or equal to the value of child node across the tree and the node with highest value will be the root node of the tree.

Implementation:

Let's assume that we have a heap having some elements which are stored in array Arr. The way to convert this array into a heap structure is the following. We pick a node in the array, check if the left sub-tree and the right sub-tree are max heaps, in themselves and the node itself is a max heap (it's value should be greater than all the child nodes)

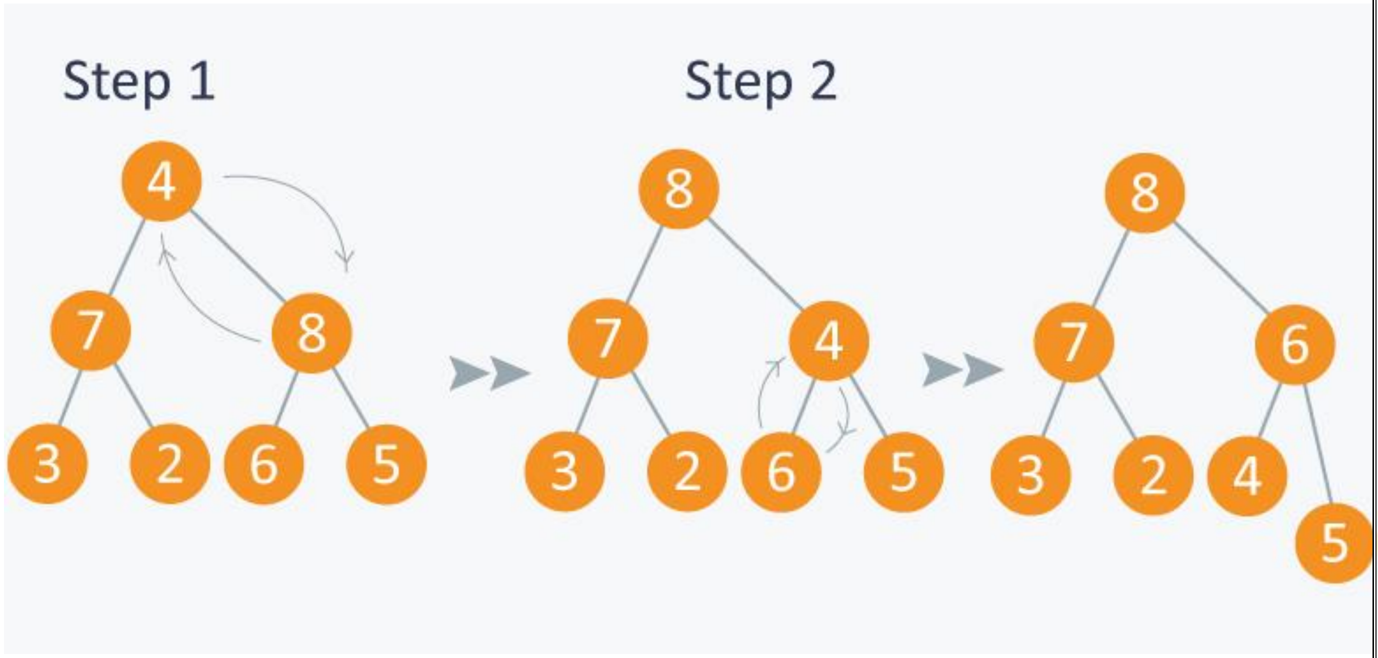
To do this we will implement a function that can maintain the property of max heap (i.e each element value should be greater than or equal to any of its child and smaller than or equal to its parent)

```
void max_heapify (int Arr[ ], int i, int N)
  {
     int left = 2*i                  //left child
     int right = 2*i +1          //right child
     if(left<= N and Arr[left] > Arr[i] )
          largest = left;
```

```
        else
             largest = i;
        if(right <= N and Arr[right] > Arr[largest] )
            largest = right;
        if(largest != i )
        {
            swap (Arr[i] , Arr[largest]);
            max_heapify (Arr, largest,N);
        }
    }
```

Complexity: O(logN)

Example:
In the diagram below,initially 1st node (root node) is violating property of max-heap as it has smaller value than its children, so we are performing max_heapify function on this node having value 4.



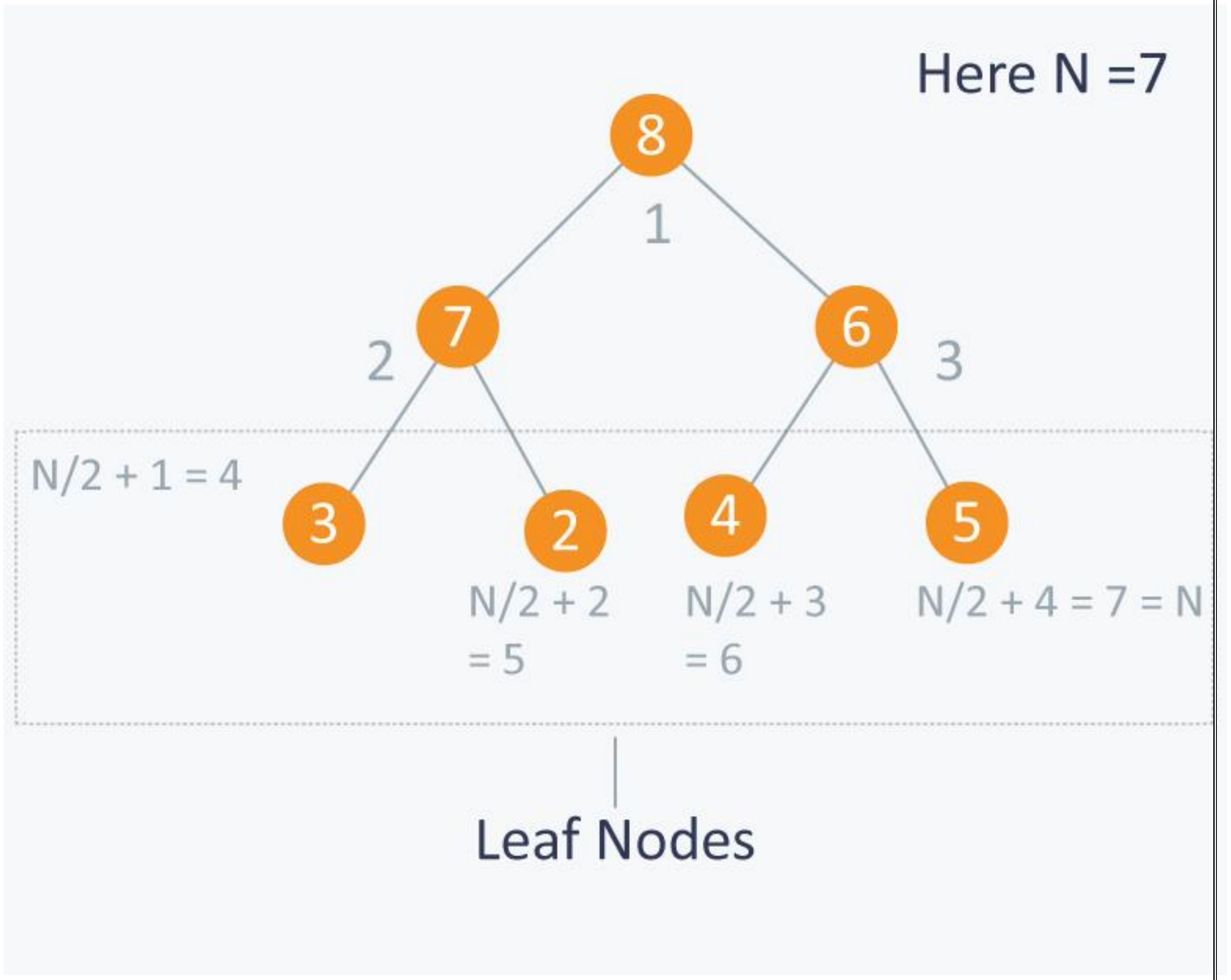
As 8 is greater than 4, so 8 is swapped with 4 and max_heapify is performed again on 4, but on different position. Now in step 2, 6 is greater than 4, so 4 is swapped with 6 and we will get a max heap, as now 4 is a leaf node, so further call to max_heapify will not create any effect on heap.

Now as we can see that we can maintain max- heap by using **max_heapify** function.

Before moving ahead, lets observe a property which states: A N element heap stored in an array has leaves indexed by N/2+1, N/2+2 , N/2+3 …. upto N.

Let's observe this with an example:

Lets take above example of 7 elements having values {8, 7, 6, 3, 2, 4, 5}.



So you can see that elements 3, 2, 4, 5 are indexed by N/2+1 (i.e 4), N/2+2 (i.e 5) and N/2+3 (i.e 6) and N/2+4 (i.e 7) respectively.

Building MAX HEAP:

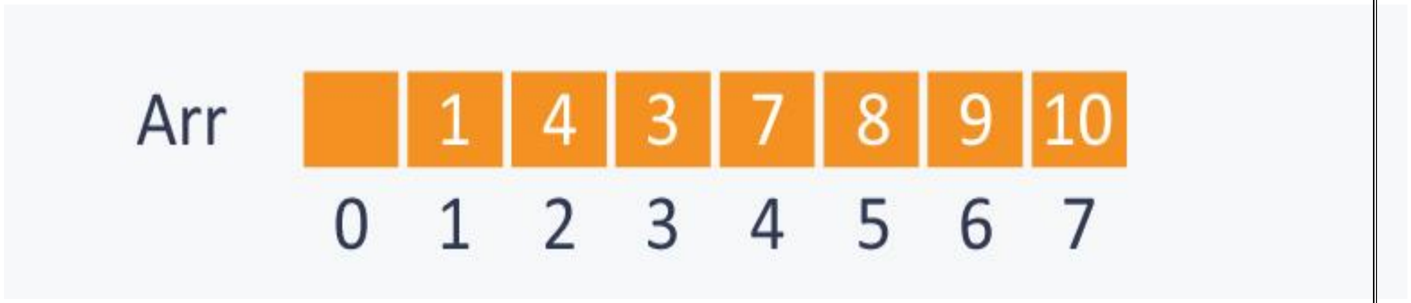
Now let's say we have N elements stored in the array Arr indexed from 1 to N. They are currently not following the property of max heap. So we can use max-heapify function to make a max heap out of the array.

How?

From the above property we observed that elements from Arr[N/2+1] to Arr[N] are leaf nodes, and each node is a 1 element heap. We can use max_heapify function in a bottom up manner on remaining nodes, so that we can cover each node of tree.

```
void build_maxheap (int Arr[ ])
   {
      for(int i = N/2 ; i >= 1 ; i-- )
      {
            max_heapify (Arr, i) ;
        }
    }
```
Complexity: O(N). **max_heapify** function has complexity logN and the **build_maxheap** functions runs only N/2 times, but the amortized complexity for this function is actually linear.

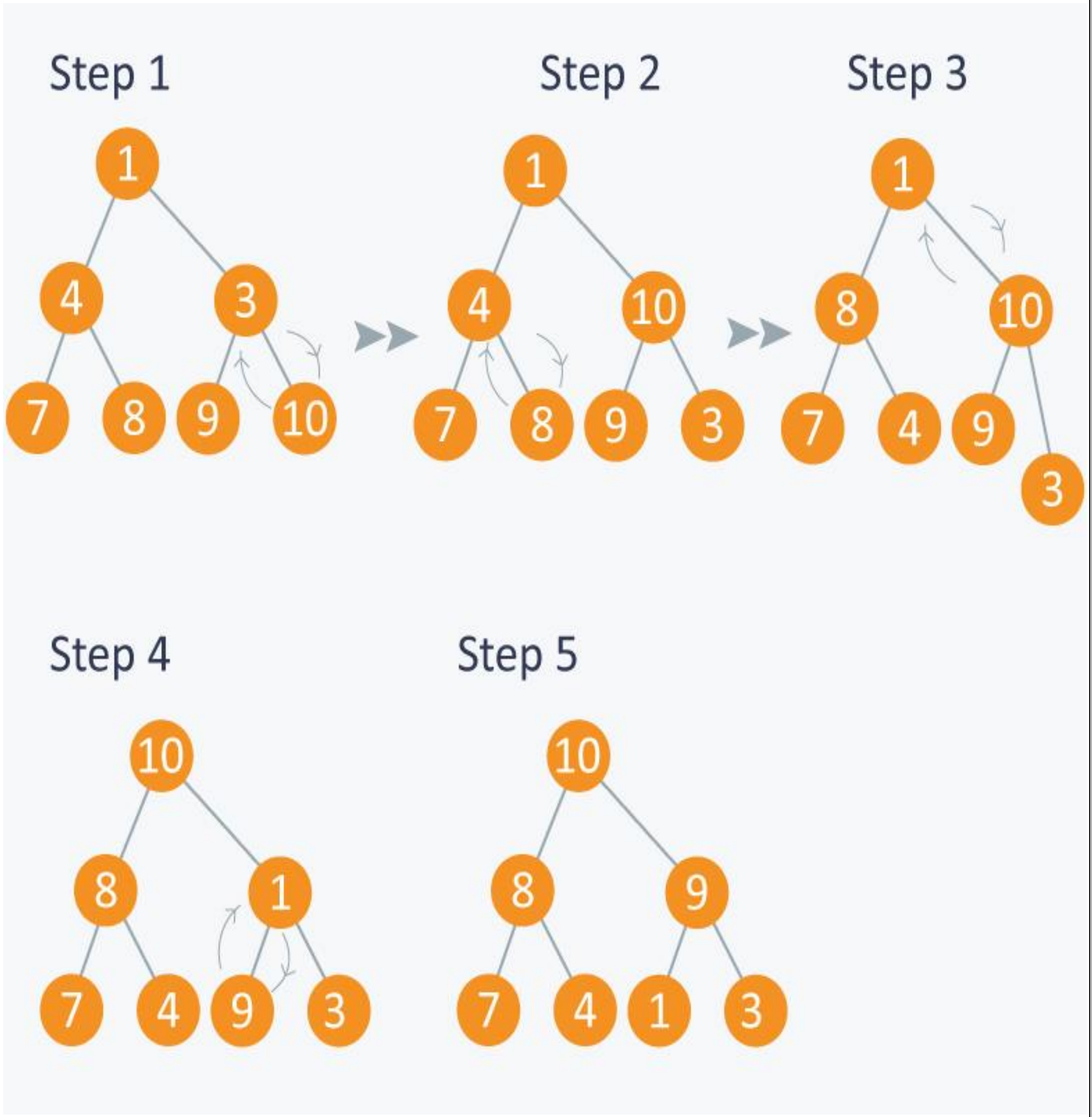
Example:
Suppose you have 7 elements stored in array Arr.



Here N=7, so starting from node having index N/2=3, (also having value 3 in the above diagram), we will call max_heapify from index N/2 to 1.

In the diagram below:

In step 1, in max_heapify(Arr, 3), as 10 is greater than 3, 3 and 10 are swapped and further call to max_heap(Arr, 7) will have no effect as 3 is a leaf node now.

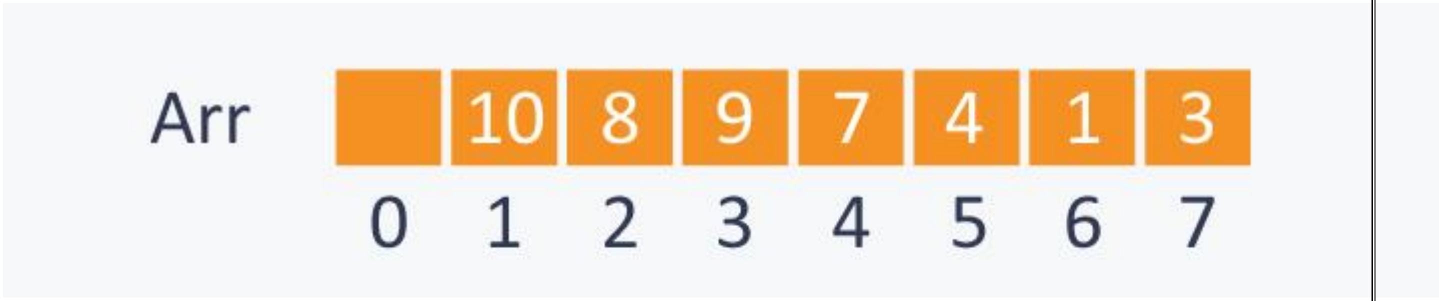


In step 2, calling max_heapify(Arr, 2) , (node indexed with 2 has value 4) , 4 is swapped with 8 and further call to max_heap(Arr, 5) will have no effect, as 4 is a leaf node now.

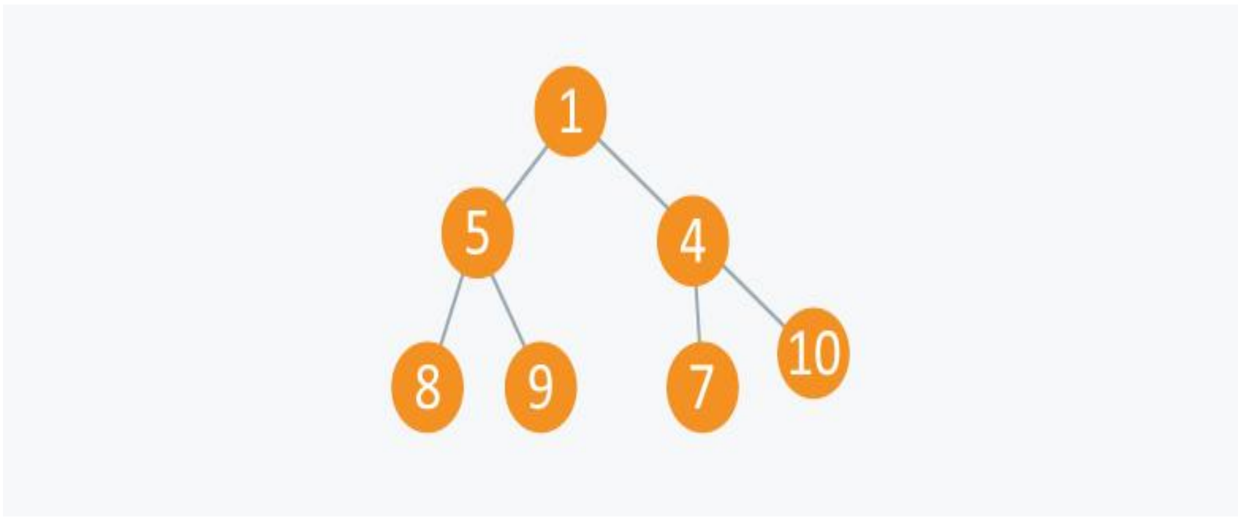
In step 3, calling max_heapify(Arr, 1) , (node indexed with 1 has value 1), 1 is swapped with 10 .

Step 4 is a subpart of step 3, as after swapping 1 with 10, again a recursive call of max_heapify(Arr, 3) will be performed , and 1 will be swapped with 9. Now further call to max_heapify(Arr, 7) will have no effect, as 1 is a leaf node now.

In step 5, we finally get a max- heap and the elements in the array Arr will be :



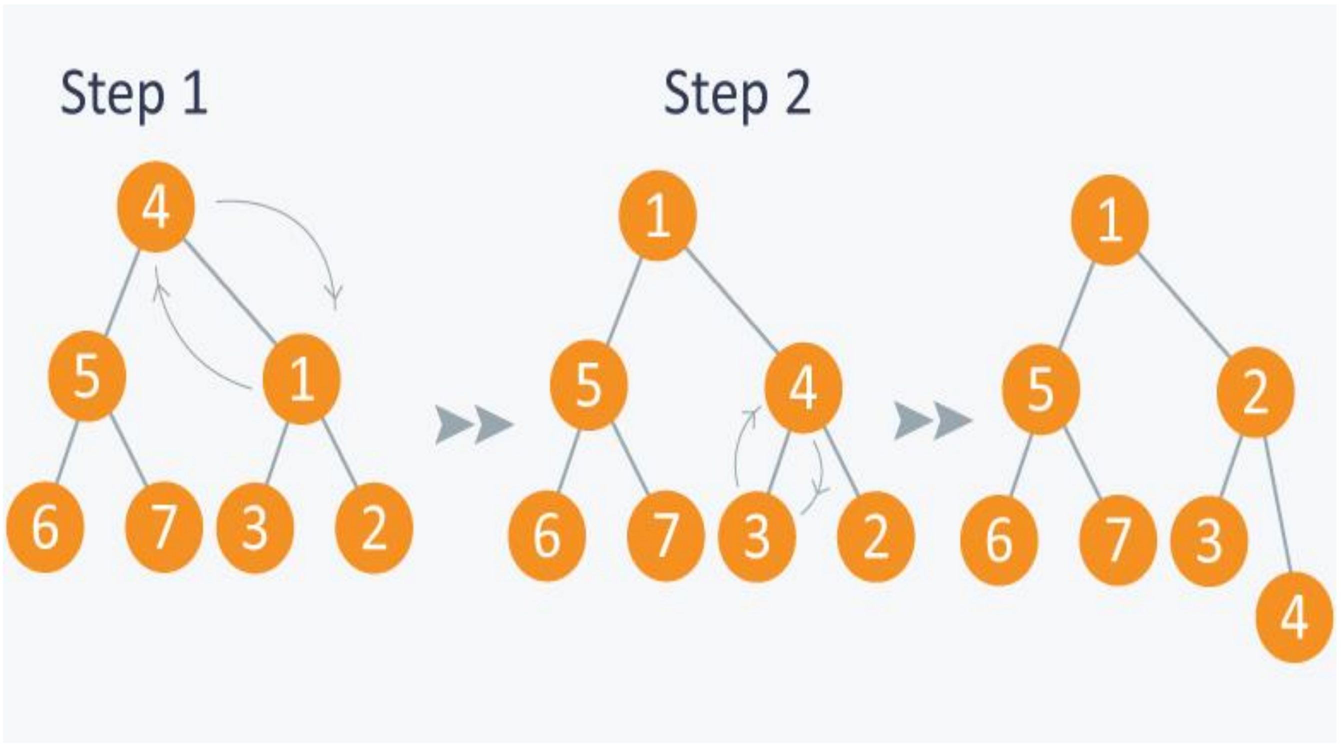
Min Heap: In this type of heap, the value of parent node will always be less than or equal to the value of child node across the tree and the node with lowest value will be the root node of tree.



As you can see in the above diagram, each node has a value smaller than the value of their children.
We can perform same operations as performed in building max_heap.
First we will make function which can maintain the min heap property, if some element is violating it.

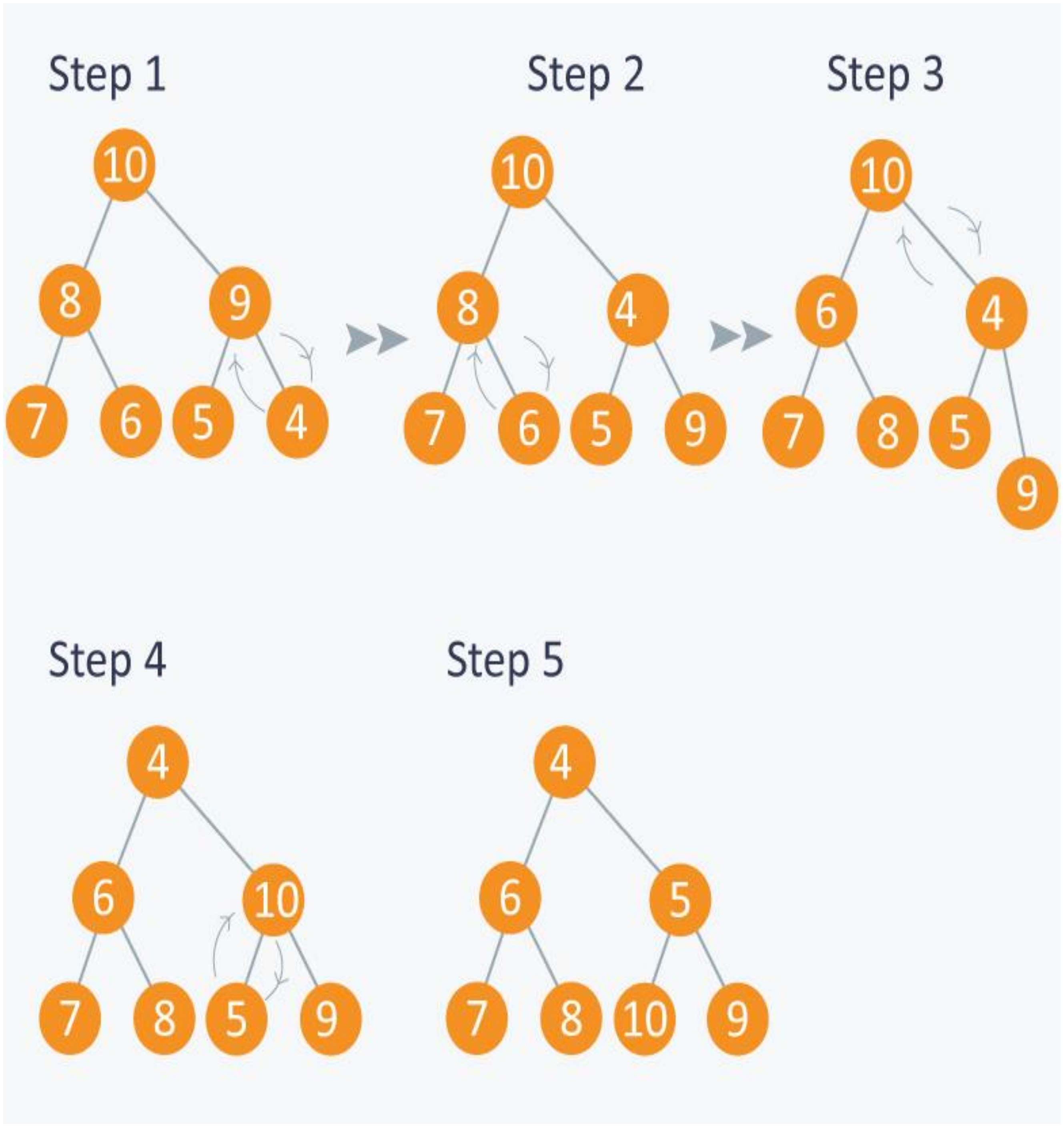
```
void min_heapify (int Arr[ ] , int i, int N)
  {
  int left  = 2*i;
  int right = 2*i+1;
  int smallest;
  if(left <= N and Arr[left] < Arr[ i ] )
       smallest = left;
  else
      smallest = i;
  if(right <= N and Arr[right] < Arr[smallest] )
      smallest = right;
  if(smallest != i)
  {
      swap (Arr[ i ], Arr[ smallest ]);
```

```
        min_heapify (Arr, smallest,N);
    }
    }
```
Complexity: O(logN) .
Example:
Suppose you have elements stored in array Arr {4, 5, 1, 6, 7, 3, 2}. As you can see in the diagram below, the element at index 1 is violating the property of min -heap, so performing min_heapify(Arr, 1) will maintain the min-heap.



Now let’s use above function in building min-heap. We will run the above function on remaining nodes other than leaves as leaf nodes are 1 element heap.

```
  void build_minheap (int Arr[ ])
    {
        for( int i = N/2 ; i >= 1 ; i--)
        min_heapify (Arr, i);
    }
```
Complexity: O(N). The complexity calculation is similar to that of building max heap.
Example:
Consider elements in array {10, 8, 9, 7, 6, 5, 4} . We will run min_heapify on nodes indexed from N/2 to 1. Here node indexed at N/2 has value 9. And at last, we will get a min_heap.



Heaps can be considered as partially ordered tree, as you can see in the above examples that the nodes of tree do not follow any order with their siblings(nodes on the same level). They can be mainly used when we give more priority to smallest or the largest node in the tree as we can extract these node very efficiently using heaps.

APPLICATIONS:

1) **Heap Sort:**

We can use heaps in sorting the elements in a specific order in efficient time.
Let’s say we want to sort elements of array Arr in ascending order. We can use max heap to perform this operation.
Idea: We build the max heap of elements stored in Arr, and the maximum element of Arr will always be at the root of the heap.

Leveraging this idea we can sort an array in the following manner.

Processing:

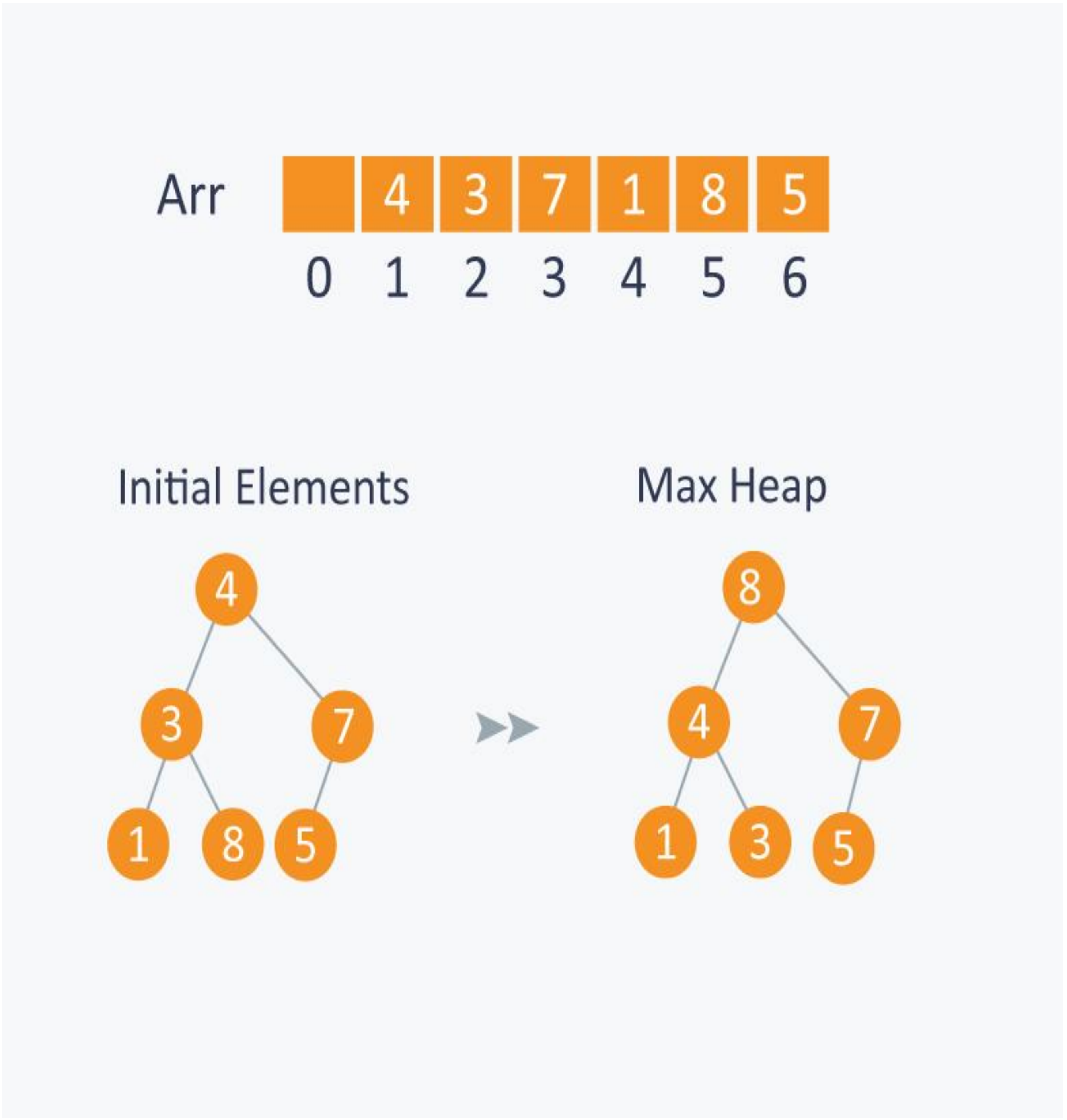
- Initially we will build a max heap of elements in Arr.
- Now the root element that is Arr[1] contains maximum element of Arr. After that, we will exchange this element with the last element of Arr and will again build a max heap excluding the last element which is already in its correct position and will decrease the length of heap by one.
- We will repeat the step 2, until we get all the elements in their correct position.
- We will get a sorted array.

Implementation:

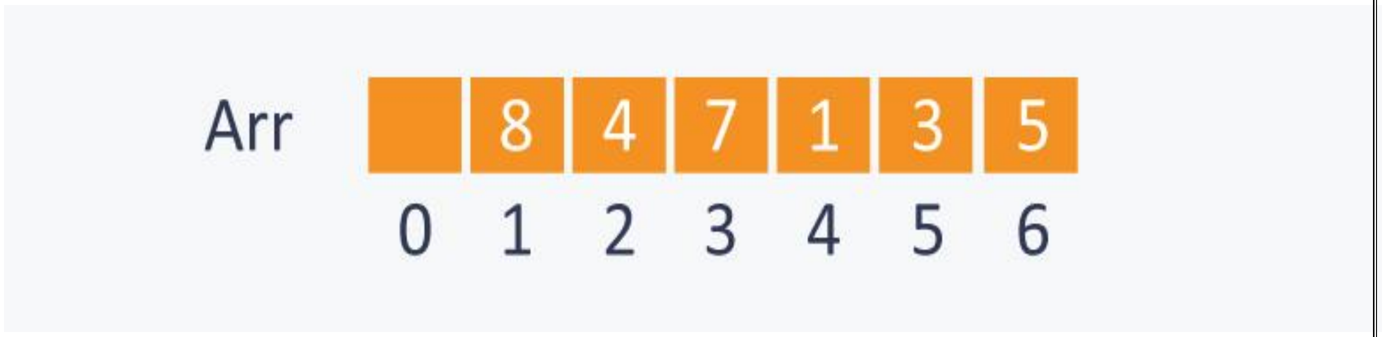
Suppose there are N elements stored in array Arr.

```
void heap_sort(int Arr[ ])
  {
     int heap_size = N;
     build_maxheap(Arr);
     for(int i = N; i>=2 ; i-- )
     {
        swap(Arr[ 1 ], Arr[ i ]);
        heap_size = heap_size-1;
        max_heapify(Arr, 1, heap_size);
     }
  }
```

Complexity: As we know max_heapify has complexity O(logN), build_maxheap has complexity O(N) and we run max_heapify N−1 times in heap_sort function, therefore complexity of heap_sort function is O(NlogN).
Example:
In the diagram below,initially there is an unsorted array Arr having 6 elements. We begin by building max-heap.

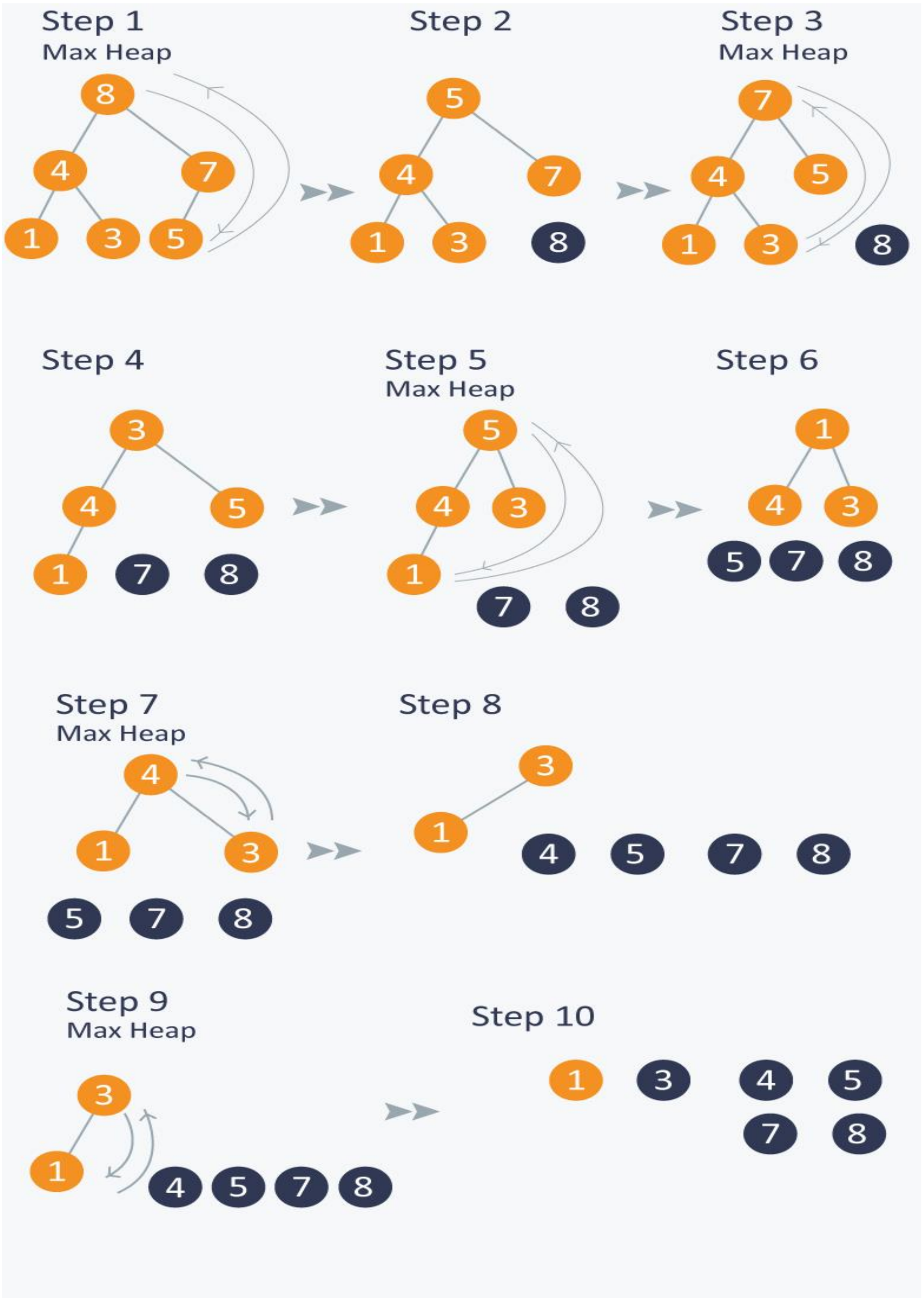


After building max-heap, the elements in the array Arr will be:

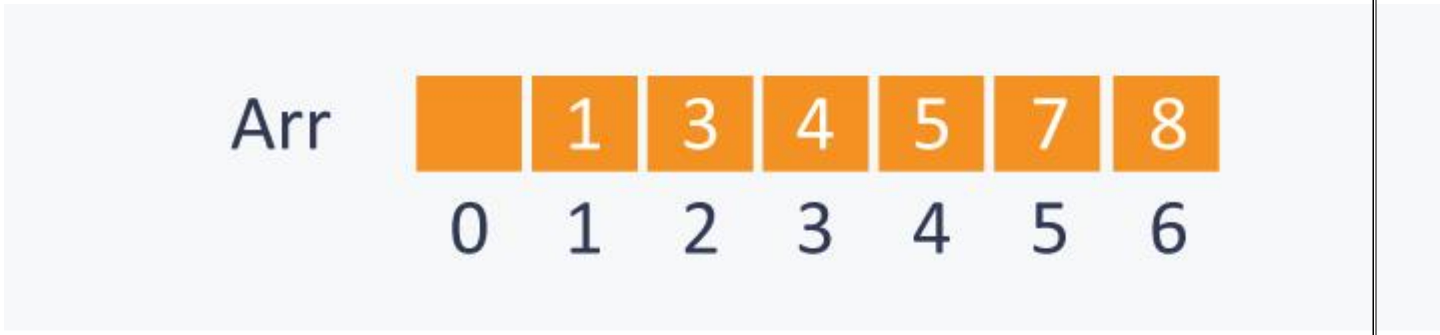


Processing:

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.


Step 1
Max Heap
Step 2
Step 3
Max Heap
Step 4
Step 5
Max Heap
Step 6
Step 7
Max Heap
Step 8
Step 9
Max Heap
Step 10

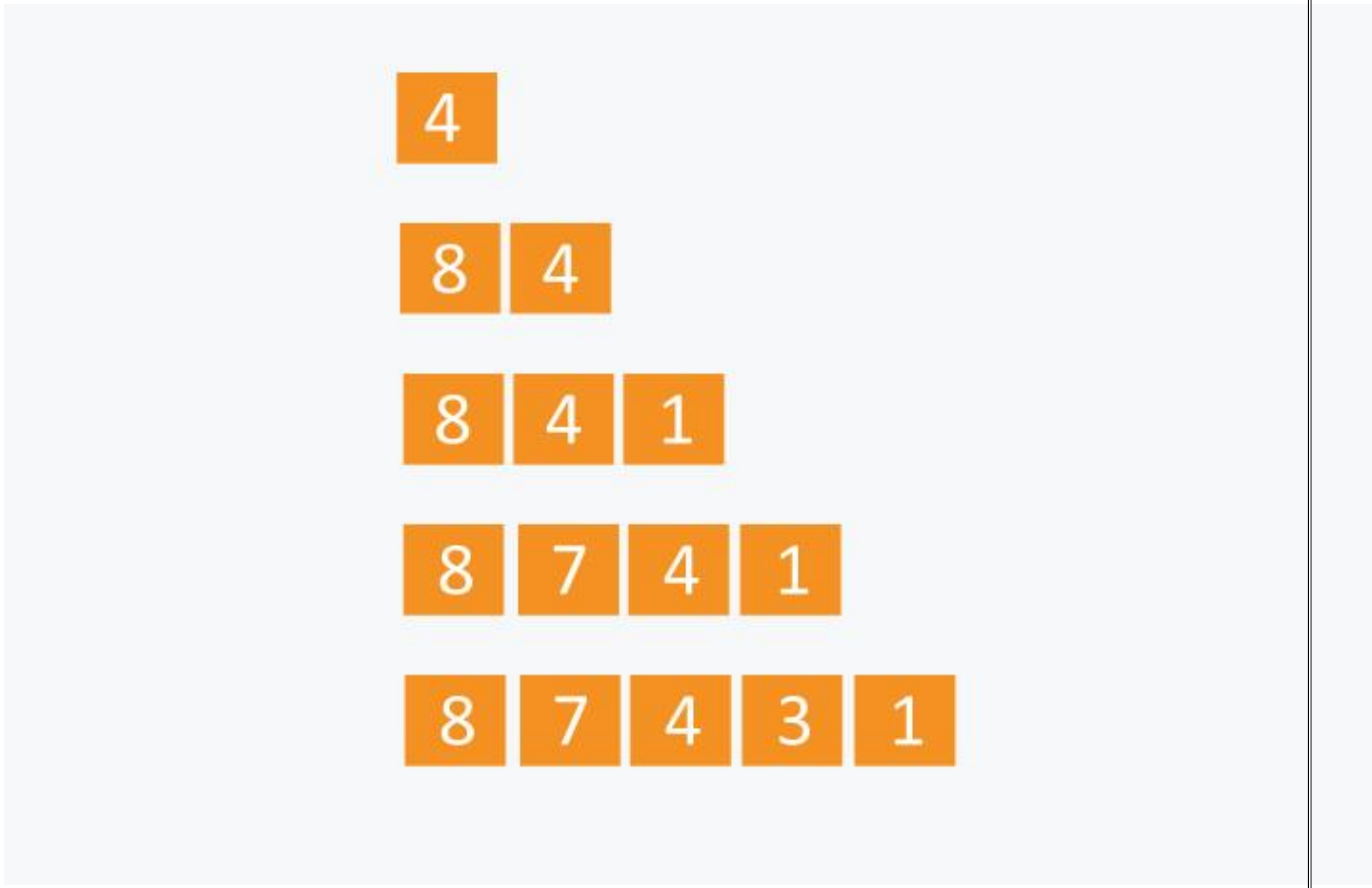
After all the steps, we will get a sorted array.



2) **Priority Queue:**

Priority Queue is similar to queue where we insert an element from the back and remove an element from front, but with a difference that the logical order of elements in the priority queue depends on the priority of the elements. The element with highest priority will be moved to the front of the queue and one with lowest priority will move to the back of the queue. Thus it is possible that when you enqueue an element at the back in the queue, it can move to front because of its highest priority.

Example:
Let’s say we have an array of 5 elements : {4, 8, 1, 7, 3} and we have to insert all the elements in the max-priority queue.
First as the priority queue is empty, so 4 will be inserted initially.
Now when 8 will be inserted it will moved to front as 8 is greater than 4.
While inserting 1, as it is the current minimum element in the priority queue, it will remain in the back of priority queue.
Now 7 will be inserted between 8 and 4 as 7 is smaller than 8.
Now 3 will be inserted before 1 as it is the 2nd minimum element in the priority queue. All the steps are represented in the diagram below:



We can think of many ways to implement the priority queue.

Naive Approach:
Suppose we have N elements and we have to insert these elements in the priority queue. We can use list and can insert elements in O(N) time and can sort them to maintain a priority queue in O(NlogN) time.
Efficient Approach:
We can use heaps to implement the priority queue. It will take O(logN) time to insert and delete each element in the priority queue.

Based on heap structure, priority queue also has two types max- priority queue and min - priority queue.

Let’s focus on Max Priority Queue.

Max Priority Queue is based on the structure of max heap and can perform following operations:

maximum(Arr) : It returns maximum element from the Arr.
extract_maximum (Arr) - It removes and return the maximum element from the Arr.
increase_val (Arr, i , val) - It increases the key of element stored at index i in Arr to new value **val**.
insert_val (Arr, val) - It inserts the element with value **val** in Arr.

Implementation:

length = number of elements in Arr.

Maximum :

```
int maximum(int Arr[ ])
    {
       return Arr[ 1 ];  //as the maximum element is the root element in the max heap.
    }
```

Complexity: O(1)

Extract Maximum: In this operation, the maximum element will be returned and the last element of heap will be placed at index 1 and max_heapify will be performed on node 1 as placing last element on index 1 will violate the property of max-heap.

```
int extract_maximum (int Arr[ ])
{
    if(length == 0)
    {
cout<< "Can't remove element as queue is empty";
       return -1;
    }
    int max = Arr[1];
    Arr[1] = Arr[length];
    length = length -1;
    max_heapify(Arr, 1);
    return max;
}
```

Complexity: O(logN).

Increase Value: In case increasing value of any node, it may violate the property of max-heap, so we may have to swap the parent's value with the node's value until we get a larger value on parent node.

```
void increase_value (int Arr[ ], int i, int val)
 {
    if(val < Arr[ i ])
    {
       cout<<"New value is less than current value, can't be inserted" <<endl;
       return;
    }
    Arr[ i ] = val;
    while( i > 1 and Arr[ i/2 ] < Arr[ i ])
    {
       swap(Arr[ i/2 ], Arr[ i ]);
       i = i/2;
    }
 }
```

Complexity : O(logN).

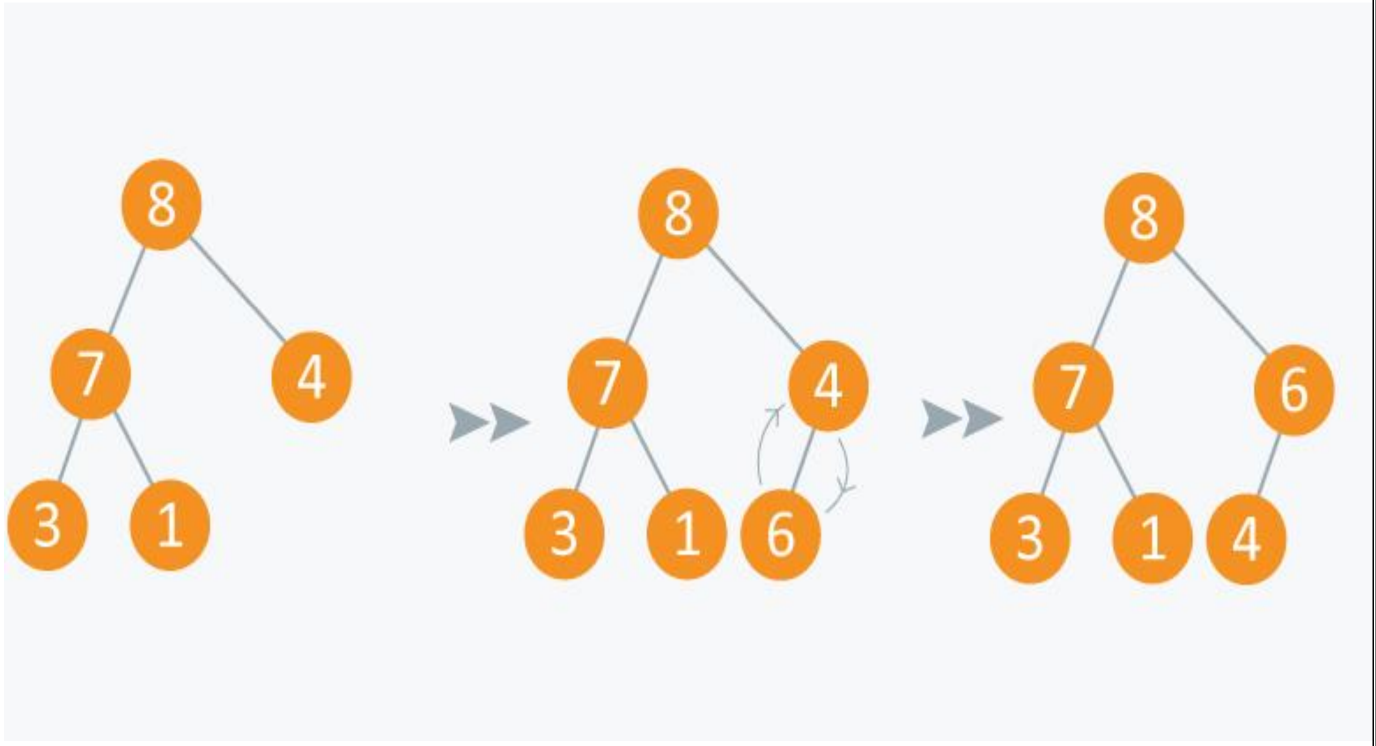
Insert Value :

```
void insert_value (int Arr[ ], int val)
  {
     length = length + 1;
     Arr[ length ] = -1;  //assuming all the numbers greater than 0 are to be inserted in queue.
     increase_val (Arr, length, val);
  }
```

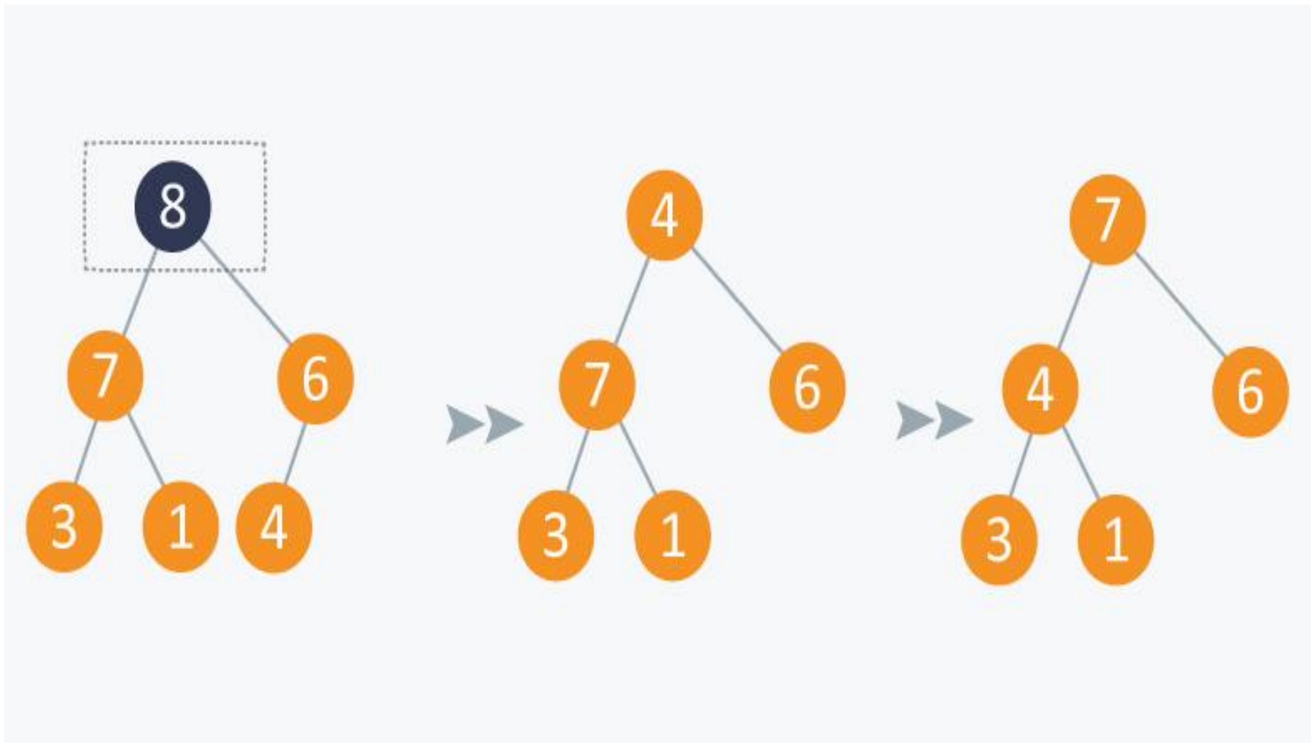
Complexity: O(logN).

Example:

Initially there are 5 elements in priority queue.
Operation: Insert Value(Arr, 6)
In the diagram below, inserting another element having value 6 is violating the property of max-priority queue, so it is swapped with its parent having value 4, thus maintaining the max priority queue.



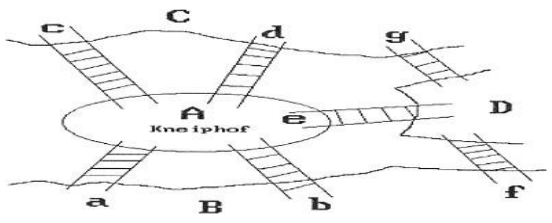
Operation: Extract Maximum:
In the diagram below, after removing 8 and placing 4 at node 1, violates the property of max-priority queue. So max_heapify(Arr, 1) will be performed which will maintain the property of max - priority queue.



As discussed above, like heaps we can use priority queues in scheduling of jobs. When there are N jobs in queue, each having its own priority. If the job with maximum priority will be completed first and will be removed from the queue, we can use priority queue's operation extract_maximum here. If at every instant we have to add a new job in the queue, we can use **insert_value** operation as it will insert the element in O(logN) and will also maintain the property of max heap.

Introduction to Graphs

Graph is a non linear data structure; A map is a well-known example of a graph. In a map various connections are made between the cities. The cities are connected via roads, railway lines and aerial network. We can assume thatthe graph is the interconnection of cities by roads. Euler used graph theory to solve Seven Bridges of Königsbergproblem. Is there a possible way to traverse every bridge exactly once – Euler Tour



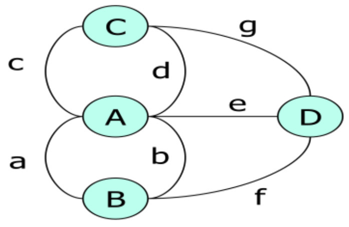


Figure: Section of the river Pregal in Koenigsberg and Euler's graph.

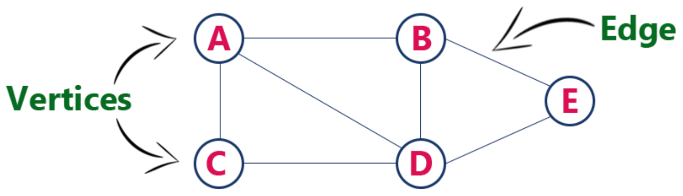
Defining the degree of a vertex to be the number of edges incident to it, Euler showed that there is a walk starting at any vertex, going through each edge exactly once and terminating at the start vertex iff the degree of each, vertex is even. A walk which does this is called Eulerian. There is no Eulerian walk for the Koenigsberg bridge problem as all four vertices are of odd degree.

A graph contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) whichconnects the vertices.

A graph is defined as Graph is a collection of vertices and arcs which connects vertices in the graph. A graph G isrepresented as G = (V , E), where V is set of vertices and E is set of edges.

Example: 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)}. This is a graph with 5 vertices and 6 edges.



Graph Terminology

1.**Vertex** : An individual data element of a graph is called as Vertex. Vertex is also known as node. In aboveexample graph, A, B, C, D & E are known as vertices.

2.**Edge** : An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as(starting Vertex, ending Vertex).

In above graph, the link between vertices A and B is represented as

(A,B).Edges are three types:

1.Undirected Edge - An undirected edge is a bidirectional edge. If there is an undirected edge between vertices Aand B then edge (A , B) is equal to edge (B , A).

2.Directed Edge - A directed edge is a unidirectional edge. If there is a directed edge between vertices A and Bthen edge (A , B) is not equal to edge (B , A).

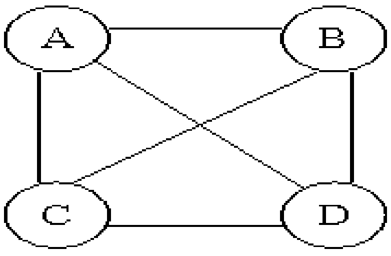
3.Weighted Edge - A weighted edge is an edge with cost on it.

Types of Graphs

1.Undirected

Graph

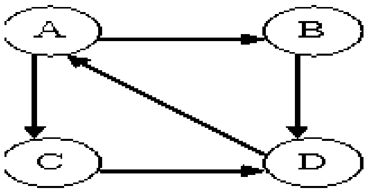
A graph with only undirected edges is said to be undirected graph.



Undirected Graph.

2.Directed Graph

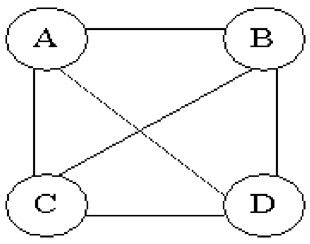
A graph with only directed edges is said to be directed graph.



Directed Graph.

3.Complete Graph

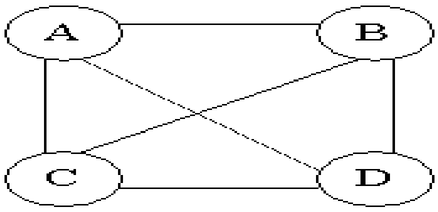
A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. Anundirected graph contains the edges that are equal to edges = n(n-1)/2 where n is the number of vertices present inthe graph. The following figure shows a complete graph.



A complete graph.

4.Regular Graph

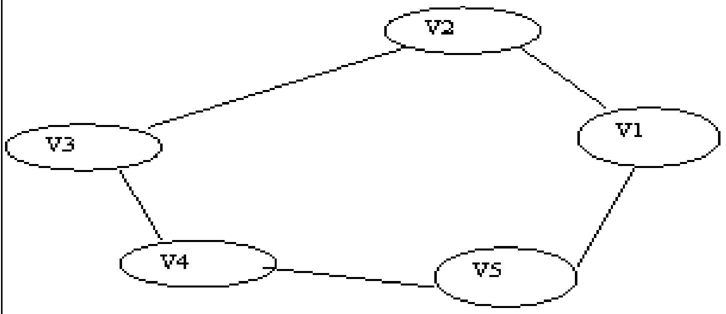
Regular graph is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any othernode.



A regular graph

5. Cycle Graph

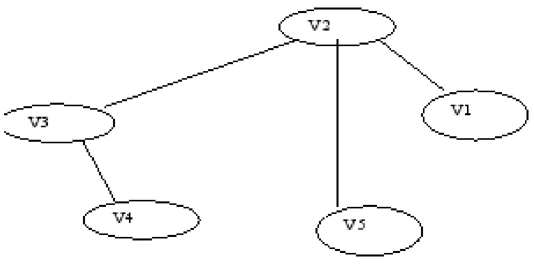
A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple pathis a cycle.



A cycle graph

6. Acyclic Graph

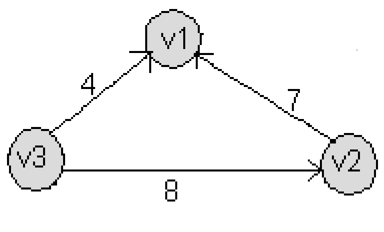
A graph without cycle is called acyclic graphs.



A acyclic graph

7. Weighted Graph

A graph is said to be weighted if there are some non negative value assigned to each edges of the graph. Thevalue is equal to the length between two vertices. Weighted graph is also called a network.



A weighted graph

Outgoing Edge

A directed edge is said to be outgoing edge on its orign 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 to have the same end vertices, and for two directed edges to have the sameorigin and the same destination. Such edges are called parallel edges or multiple edges.

Self-loop

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

Simple Graph

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

Adjacent nodes

When there is an edge from one node to another then these nodes are called adjacent nodes.

Incidence

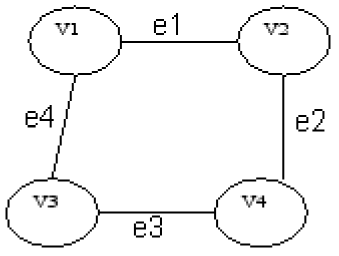
In an undirected graph the edge between v1 and v2 is incident on node v1 and v2.

Walk

A walk is defined as a finite alternating sequence of vertices and edges, beginning and ending with vertices, suchthat each edge is incident with the vertices preceding and following it.

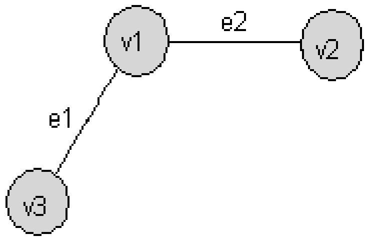
Closed walk

A walk which is to begin and end at the same vertex is called close walk. Otherwise it is an open walk.



If e1,e2,e3,and e4 be the edges of pair of vertices (v1,v2),(v2,v4),(v4,v3) and (v3,v1) respectively ,then v1 e1 v2e2 v4 e3 v3 e4 v1 be its closed walk or circuit.

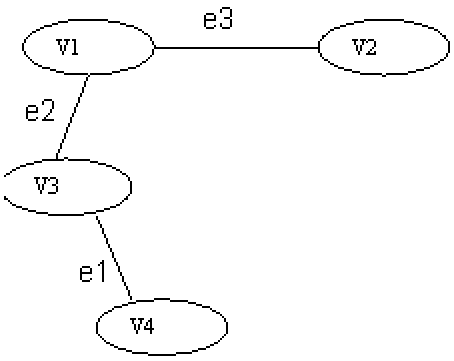
Path:A open walk in which no vertex appears more than once is called a path.



If e1 and e2 be the two edges between the pair of vertices (v1,v3) and (v1,v2) respectively, then v3 e1 v1 e2 v2 beits path.

Length of a path

The number of edges in a path is called the length of that path. In the following, the length of the path is 3.

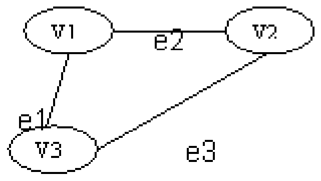


An open walk Graph

Circuit

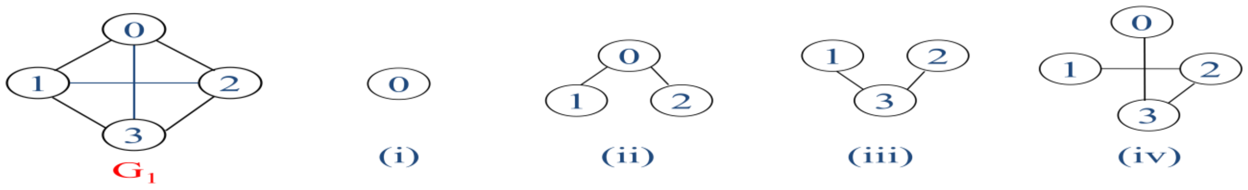
A closed walk in which no vertex (except the initial and the final vertex) appears more than once is called acircuit.

A circuit having three vertices and three edges.



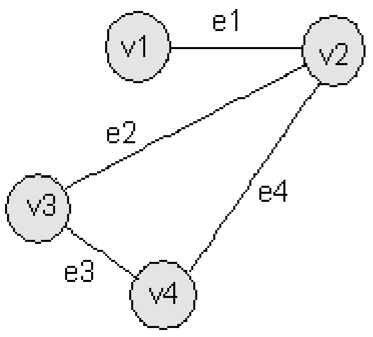
Sub Graph

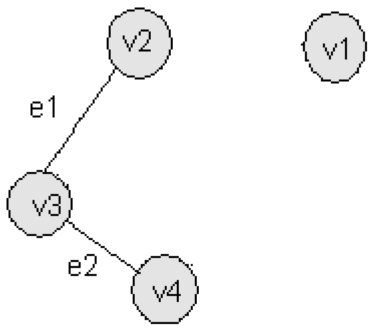
A graph S is said to be a sub graph of a graph G if all the vertices and all the edges of S are in G, and each edge ofS has the same end vertices in S as in G. A subgraph of G is a graph G' such that V(G') ⊆ V(G) and E(G') ⊆ E(G)



Connected Graph

A graph G is said to be connected if there is at least one path between every pair of vertices in G. Otherwise,G isdisconnected.





A connected graph G A disconnected graph G

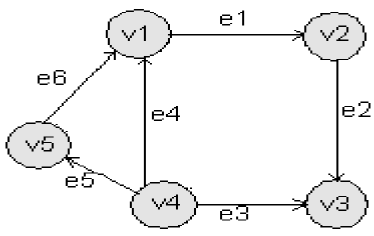
This graph is disconnected because the vertex v1 is not connected with the other vertices of the graph.

Degree

In an undirected graph, the number of edges connected to a node is called the degree of that node or the degree ofa node is the number of edges incident on it.

In the above graph, degree of vertex v1 is 1, degree of vertex v2 is 3, degree of v3 and v4 is 2 in a connectedgraph.

Indegree: The indegree of a node is the number of edges connecting to that node or in other words edges incident to it.



In the above graph,the indegree of vertices v1, v3 is 2, indegree of vertices v2, v5 is 1 and indegree of v4 is zero.

Outdegree

The outdegree of a node (or vertex) is the number of edges going outside from that node or in other words the

ADT of Graph:

Structure Graph is

objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of

verticesfunctions: for all *graph* ∈ *Graph*, *v*, v_1 and v_2 ∈ *Vertices*

Graph Create()::=return an empty graph

Graph InsertVertex(*graph*, *v*)::= return a graph with *v* inserted. *v* has no edge.

Graph InsertEdge(*graph*, *v1*,*v2*)::= return a graph with new edge between *v1* and *v2*

Graph DeleteVertex(*graph*, *v*)::= return a graph in which *v* and all edges incident to it are removed

Graph DeleteEdge(*graph*, *v1*, *v2*)::=return a graph in which the edge (*v1*, *v2*) is

removed*Boolean* IsEmpty(*graph*)::= if (*graph*==*empty graph*) return TRUE else

return FALSE *List* Adjacent(*graph*,*v*)::= return a list of all vertices that are adjacent

to *v*

Graph Representations

Graph data structure is represented using following representations

1. **Adjacency Matrix**
2. **Adjacency List**
3. **Adjacency**

Multilists1.Adjacency

Matrix

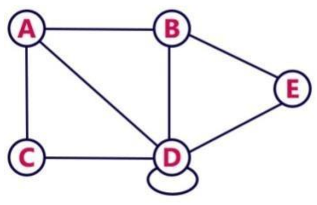
In this representation, graph can be represented using a matrix of size total number of vertices by total number ofvertices; means if a graph with 4 vertices can be represented using a matrix of 4X4 size.

In this matrix, rows and columns both represent vertices.

This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0represents there is no edge from row vertex to column vertex.

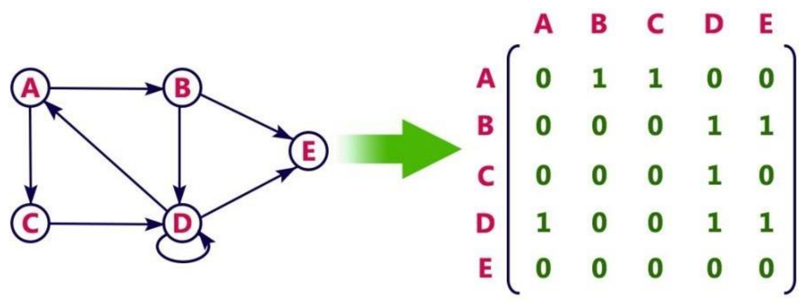
Adjacency Matrix : let G = (V, E) with n vertices, n ≥ 1. The adjacency matrix of G is a 2-dimensional n × nmatrix, A, A(i, j) = 1 iff $(v_i, v_j) \in E(G)$ $(\langle v_i, v_j \rangle$ for a diagraph), A(i, j) = 0 otherwise.

example : for undirected graph



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

For a Directed graph



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 & 0 \\ D & 1 & 0 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$

The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not besymmetric.

Merits of Adjacency Matrix:

From the adjacency matrix, to determine the connection of vertices is easy

The degree of a vertex is $\sum_{j=0}^{n-1} adj\_mat[i][j]$

For a digraph, the row sum is the out_degree, while the column sum is the in_degree

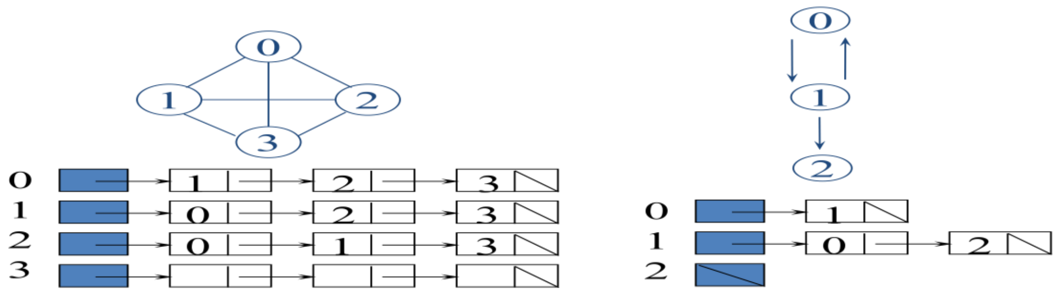
$ind(vi) = \sum_{j=0}^{n-1} A[j,i]$ $\qquad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$

The space needed to represent a graph using adjacency matrix is n^2 bits. To identify the edges in a graph,adjacency matrices will require at least $O(n^2)$ time.

2. Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices. The n rows of the adjacencymatrix are represented as n chains. The nodes in chain I represent the vertices that are adjacent to vertex i.

It can be represented in two forms. In one form, array is used to store n vertices and chain is used to store itsadjacencies. Example:



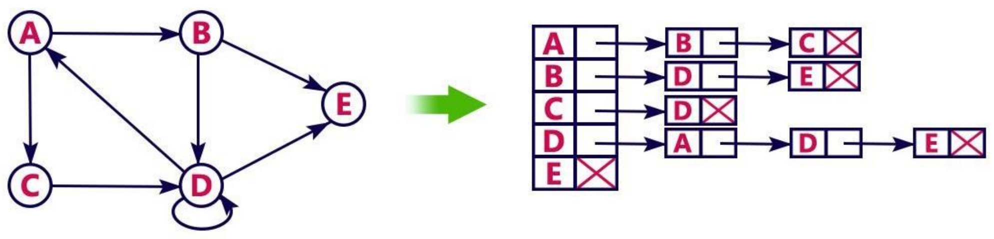
So that we can access the adjacency list for any vertex in O(1) time. Adjlist[i] is a pointer to to first node in theadjacency list for vertex i. Structure is

```
#define MAX_VERTICES 50
typedef struct node
*node_pointer;typedef struct
node {
   int vertex;
   struct node *link;
};
node_pointer
```

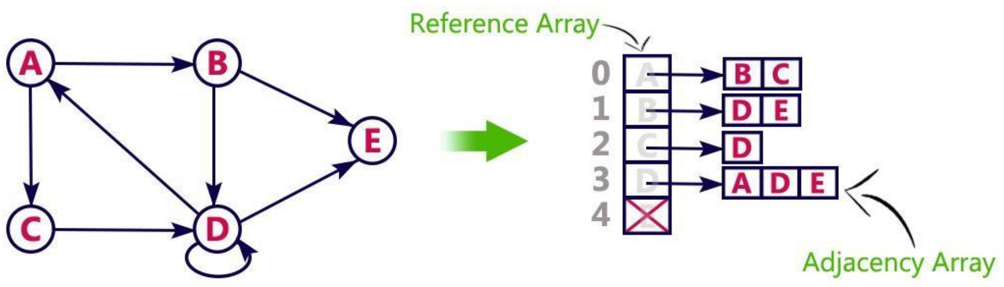
```
graph[MAX_VERTICES];int n=0;
/* vertices currently in use */
```

Another type of representation is given below.

example: consider the following directed graph representation implemented using linked list

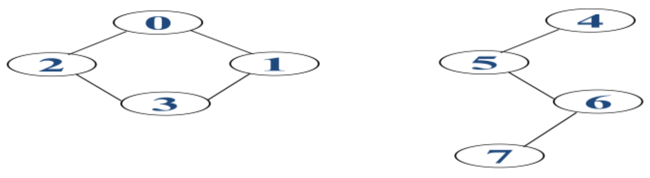


This representation can also be implemented using array



Sequential representation of adjacency list is

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	2	1	3	0	0	3	1	2	5	6	4	5	7	6

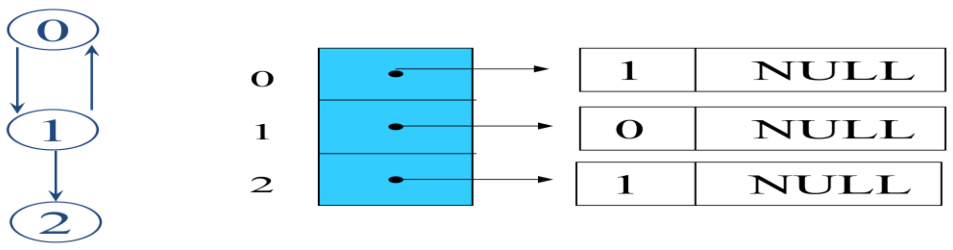


Graph

Instead of chains, we can use sequential representation into an integer array with size n+2e+1. For 0<=i<n, Array[i] gives starting point of the list for vertex I, and array[n] is set to n+2e+1. The adjacent vertices of node Iare stored sequentially from array[i].

For an undirected graph with n vertices and e edges, linked adjacency list requires an array of size n and 2e chain nodes. For a directed graph, the number of list nodes is only e. the out degree of any vertex may be determined bycounting the number of nodes in its adjacency list. To find in-degree of vertex v, we have to traverse complete list.

To avoid this, inverse adjacency list is used which contain in-degree.



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

3.Adjacency Multilists

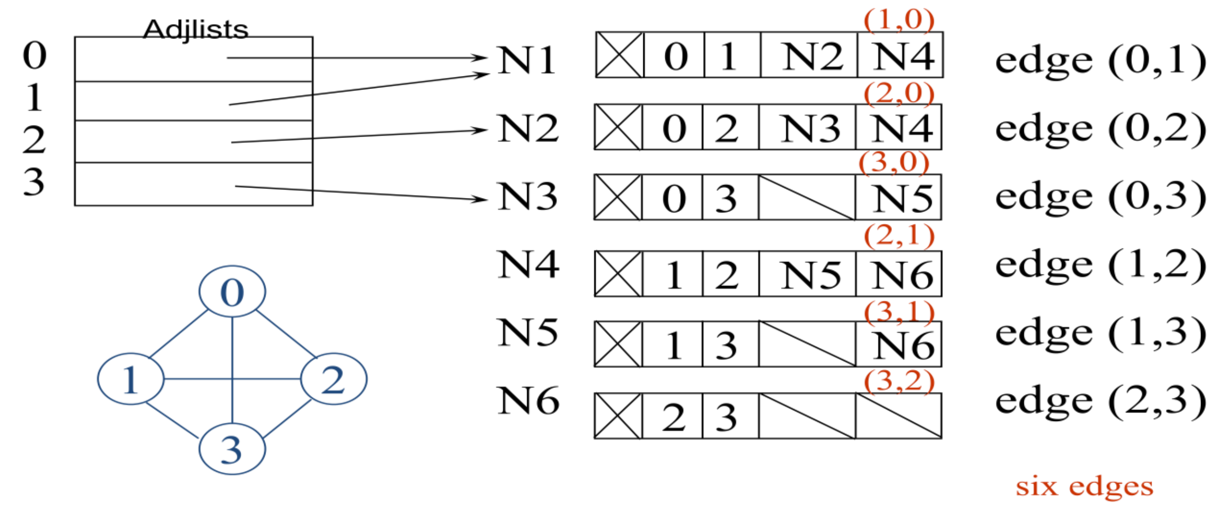
In the adjacency-list representation of an undirected graph each edge (u, v) is represented by two entries one on the list for u and the other on tht list for v. As we shall see in some situations it is necessary to be able to determin ie ~ nd enty for a particular edge and mark that edg as having been examined. This can be accomplished easilyif the adjacency lists are actually maintained as multilists (i.e., lists in which nodes may be shared among several lists). For each edge there will be exactly one node but this node will be in two lists (i.e. the adjacency lists for each of the two nodes to which it is incident).

For adjacency multilists, node

```
structure istypedef struct edge
*edge_pointer; typedef struct edge {
   short int
   marked; int
   vertex1,
   vertex2;
   edge_pointer path1, path2;
};
edge_pointer graph[MAX_VERTICES];
```

marked	vertex1	vertex2	path1	path2

Lists: vertex 0: N0->N1->N2, vertex 1: N0->N3->N4vertex 2: N1->N3->N5, vertex 3: N2-

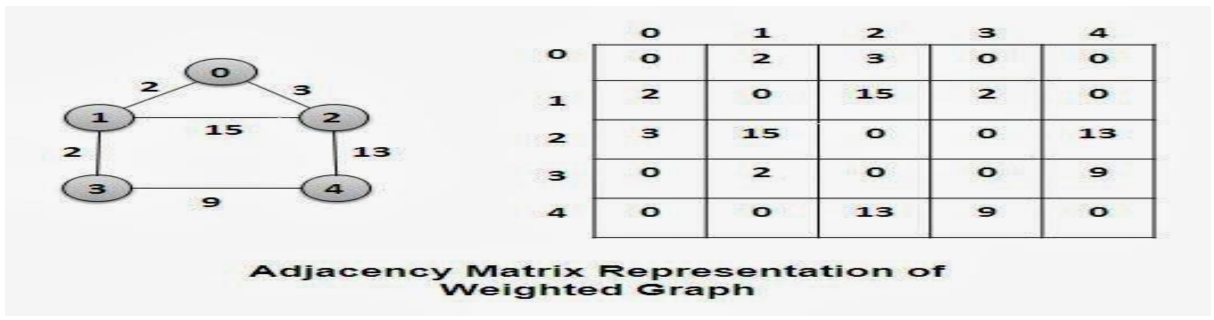


>N4->N5

Figure: Adjacency multilists for given graph

4. Weighted edges

In many applications the edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or the cost of going from one; vertex to an adjacent vertex In these applications the adjacency matrix entries A [i][j] would keep this information too. When adjacency lists are usedthe weight information may be kept in the list'nodes by including an additional field weight. A graph with weighted edges is called a network.



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

Adjacency Matrix Representation of Weighted Graph

ELEMENTARY GRAPH OPERATIONS

Given a graph G = (V E) and a vertex v in V(G) we wish to visit all vertices in G that are reachable from v (i.e., all vertices that are connected to v). We shall look at two ways of doing this: depth-first search and breadth-first search. Although these methods work on both directed and undirected graphs the following discussion assumes that the graphs are undirected.

Depth-First Search

- Begin the search by visiting the start vertex v
  - If v has an unvisited neighbor, traverse it recursively

    - Otherwise, backtrack
- Time complexity
    - Adjacency list: O(|E|)
    - Adjacency matrix: O($|V|^2$)

We begin by visiting the start vertex v. Next an unvisited vertex w adjacent to v is selected, and a depth-first search from w is initiated. When a vertex u is reached such that all its adjacent vertices have been visited, we backup to the last vertex visited that has an unvisited vertex w adjacent to it and initiate a depth-first search from w.
The search terminates when no unvisited vertex can be reached from any of the visited vertices.

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

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 adjacent vertex of the verex which is at top of the stack which is not visited and pushit on to the stack.

Step 4: Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.

Step 5: When there is no new vertex to be visit then use back tracking and pop one vertex from the

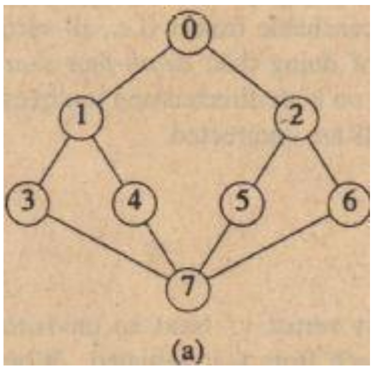
stack.Step 6: Repeat steps 3, 4 and 5 until stack becomes Empty.

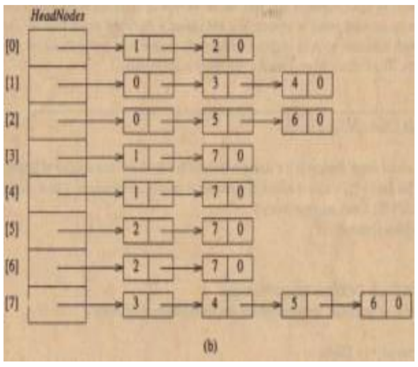
Step 7: When stack becomes Empty, then produce final spanning tree by removing unused edges from the

graphThis function is best described recursively as in Program.

```
#define FALSE 0
#define TRUE 1
 int
visited[MAX_VERTICES];
void dfs(int v)
{
  node_pointer
  w; visited[v]=
  TRUE;
  printf("%d",
  v);
  for (w=graph[v]; w; w=w-
    >link)if (!visited[w-
    >vertex])
      dfs(w->vertex);
}
```
Consider the graph G of Figure 6.16(a), which is represented by its adjacency lists as in Figure 6.16(b). If a

depth-first search is initiated from vertex 0 then the vertices of G are visited in the following order: **0 1 3 7 4 5 2 6.** Since DFS(O) visits all vertices that can be reached from 0 the vertices visited, together with all edges inG incident to these vertices form a connected component of G.





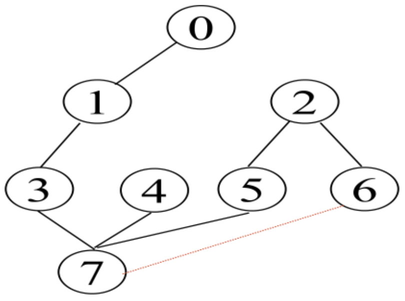


Figure: Graph and its adjacency list representation, DFS spanning tree

Analysis or DFS:
When G is represented by its adjacency lists, the vertices w adjacent to v can be determined by following a chain of links. Since DFS examines each node in the adjacency lists at most once and there are 2e list nodes the time to complete the search is O(e). If G is represented by its adjacency matrix then the time to determine all vertices adjacent to v is O(n). Since at most n vertices are visited the total time is O(n^2).

Breadth-First Search
In a breadth-first search, we begin by visiting the start vertex v. Next all unvisited vertices adjacent to v are visited. Unvisited vertices adjacent to these newly visited vertices are then visited and so on. Algorithm BFS (Program 6.2) gives the details.

```
typedef          struct          queue
*queue_pointer;  typedef   struct
queue {
   int vertex;
   queue_pointer link;
};
void
      addq(queue_pointe
      r *,queue_pointer
      *, int);
int deleteq(queue_pointer
*);void bfs(int v)
{
 node_pointer w;
 queue_pointer front,
 rear;front = rear =
 NULL; printf("%d",
 v); visited[v] =
 TRUE; addq(&front,
 &rear, v);
while (front) {
   v= deleteq(&front);
   for (w=graph[v]; w; w=w-
     >link)if (!visited[w-
```

```
      >vertex]) { printf("%d", w-
      >vertex);
        addq(&front, &rear, w-
        >vertex);visited[w->vertex]
        = TRUE;
      }
  }
}
```

Steps:

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

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 adjacent vertices of the vertex which is at front of the Queue which is not visited and insertthem into the Queue.

Step 4: When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex fromthe Queue.

Step 5: Repeat step 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

Analysis Of BFS:

Each visited vertex enters the queue exactly once. So the while loop is iterated at most n times If an adjacency matrix is used the loop takes O(n) time for each vertex visited. The total time is therefore, O(n^2). If adjacency listsare used the loop has a total cost of $d_0 + \dots + d_{n-1} = O(e)$, where d is the degree of vertex i. As in the case of DFS all visited vertices together with all edges incident to them, form a connected component of G.

3.Connected Components

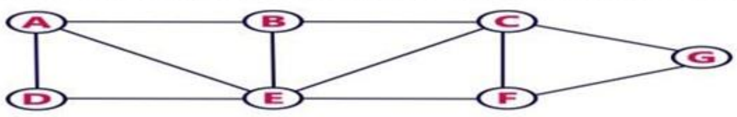
If G is an undirected graph, then one can determine whether or not it is connected by simply making a call to either DFS or BFS and then determining if there is any unvisited vertex. The connected components of a graphmay be obtained by making repeated calls to either DFS(v) or BFS(v); where v is a vertex that has not yet beenvisited. This leads to function Connected(Program 6.3), which determines the connected components of G. Thealgorithm uses DFS (BFS may be used instead if desired). The computing time is not affected. Function connected –Output outputs all vertices visited in the most recent invocation of DFS together with all edges incident on these vertices.

```
void
   connected(void)
   {for (i=0; i<n;
   i++) {
       if
           (!visited[
           i]) {
           dfs(i);
  printf("\n");        }     } }
```

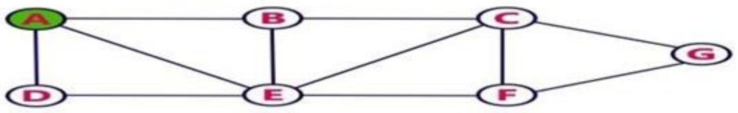
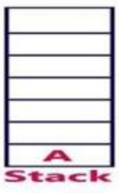
Analysis of Components:

If G is represented by its adjacency lists, then the total time taken by dfs is O(e). Since the for loops take O(n) time, the total time to generate all the Connected components is O(n+e). If adjacency matrices are used,then thetime required is O(n^2)

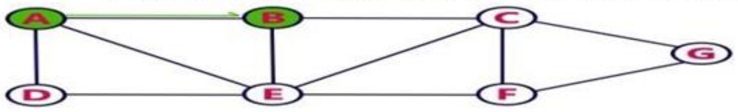
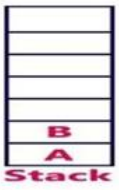
Consider the following example graph to perform DFS traversal



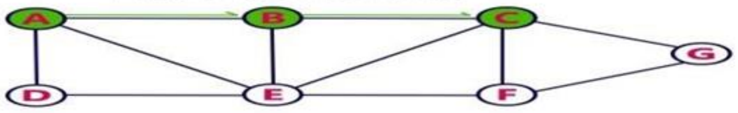
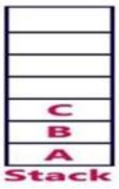
Step 1:
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

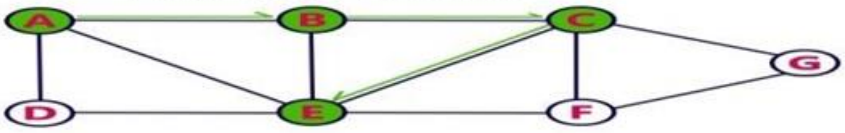
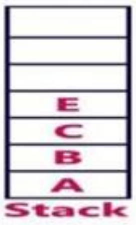
Step 2:
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

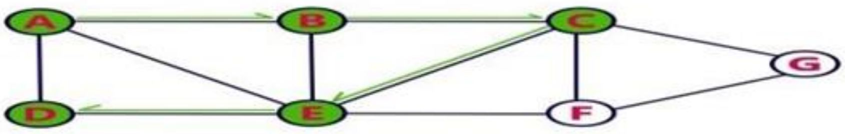
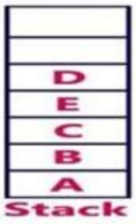
Step 3:
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

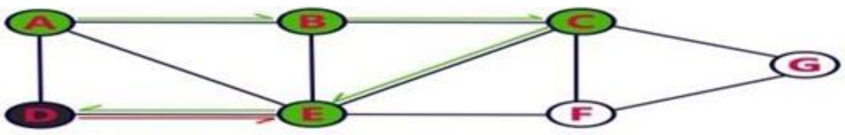
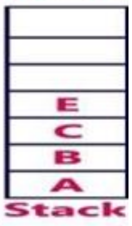
Step 4:
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

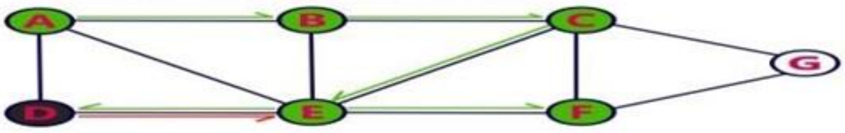
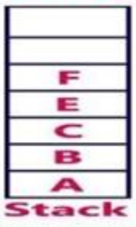
Step 5:
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

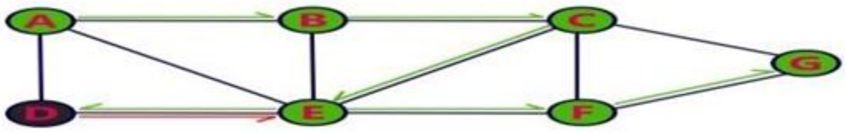
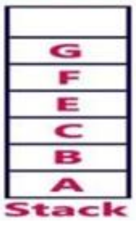
Step 6:
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

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

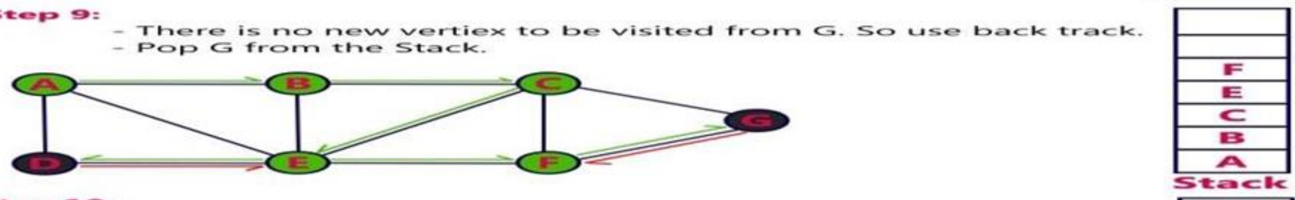



Step 8:
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

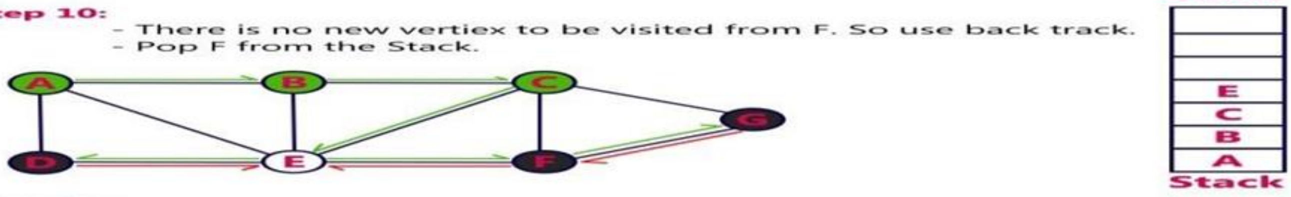
Step 9:

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



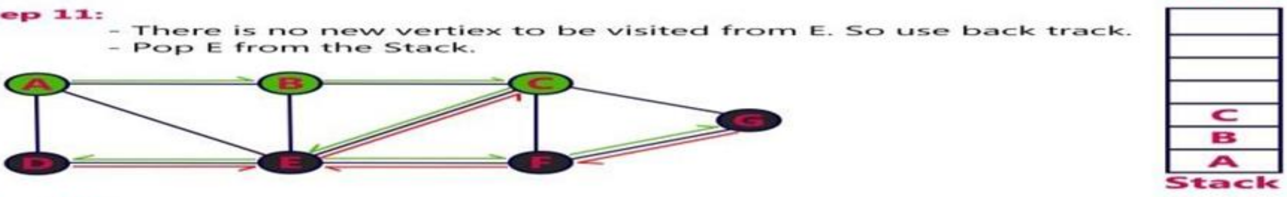
Step 10:

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



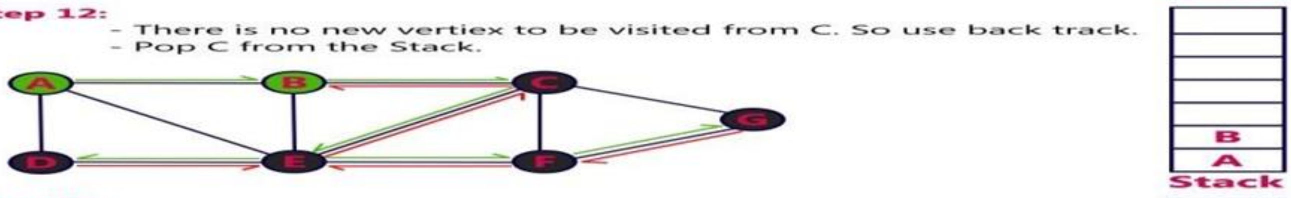
Step 11:

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



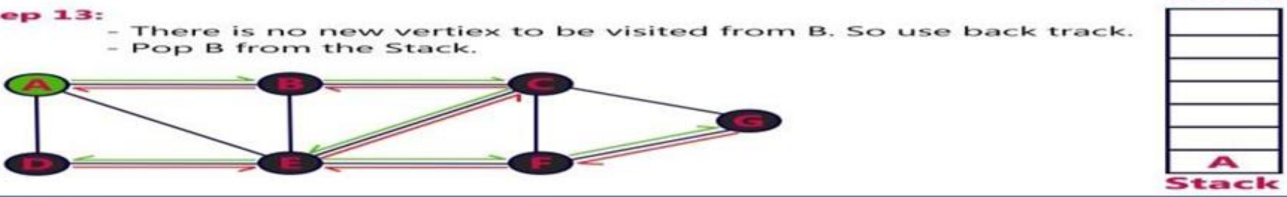
Step 12:

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



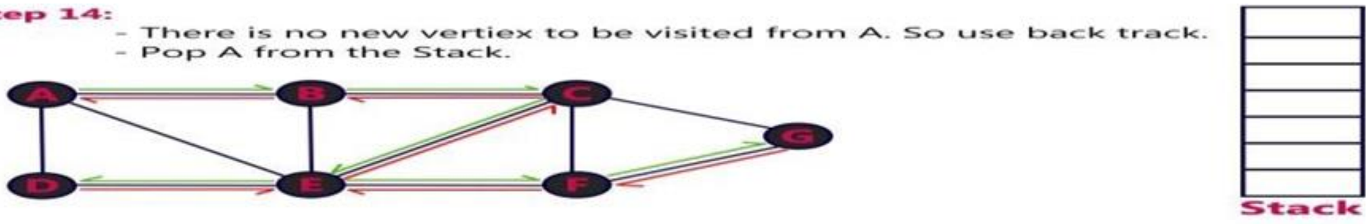
Step 13:

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

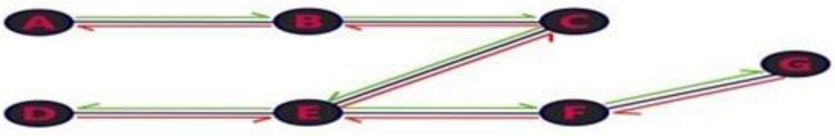


Step 14:

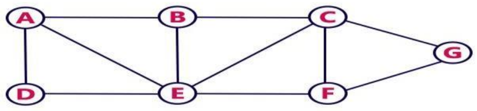
- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.



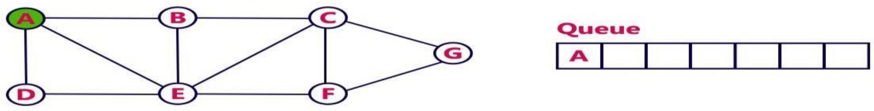
- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



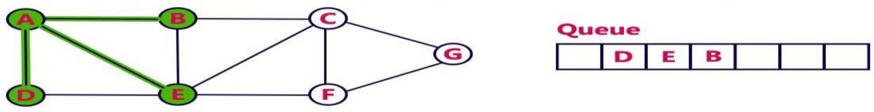
Consider the following example graph to perform BFS traversal



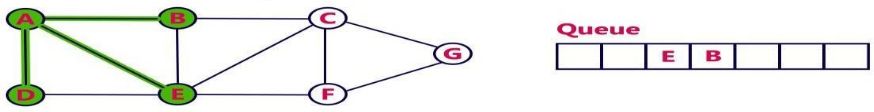
Step 1:
- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



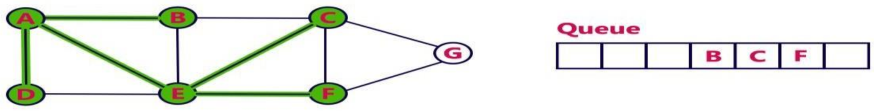
Step 2:
- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



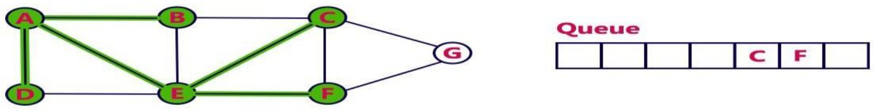
Step 3:
- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



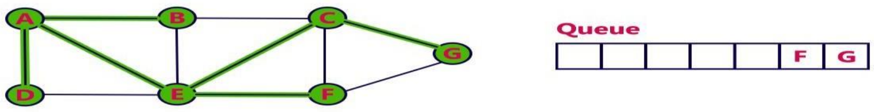
Step 4:
- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



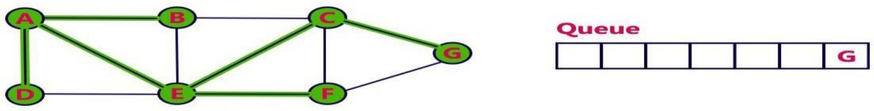
Step 5:
- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



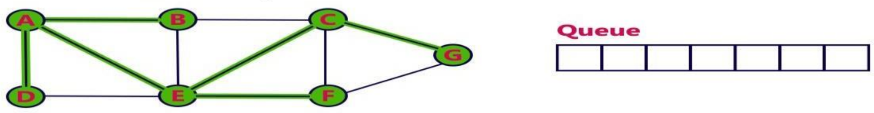
Step 6:
- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.



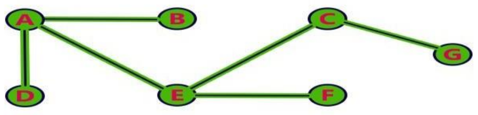
Step 7:
- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.



Step 8:
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



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



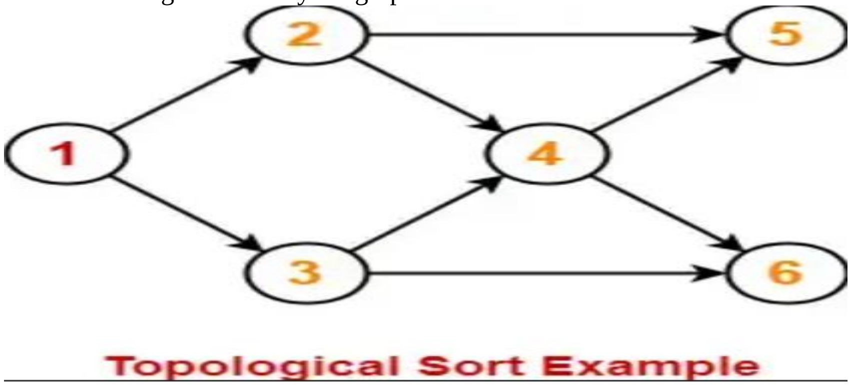
Topological Sort | Topological Sort Examples

Topological Sort is a linear ordering of the vertices in such a way that if there is an edge in the DAG going from vertex ‘u’ to vertex ‘v’,then ‘u’ comes before ‘v’ in the ordering.

It is important to note that-

- Topological Sorting is possible if and only if the graph is a Directed Acyclic Graph.
- There may exist multiple different topological orderings for a given directed acyclic graph.

Consider the following directed acyclic graph-



For this graph, following 4 different topological orderings are possible-

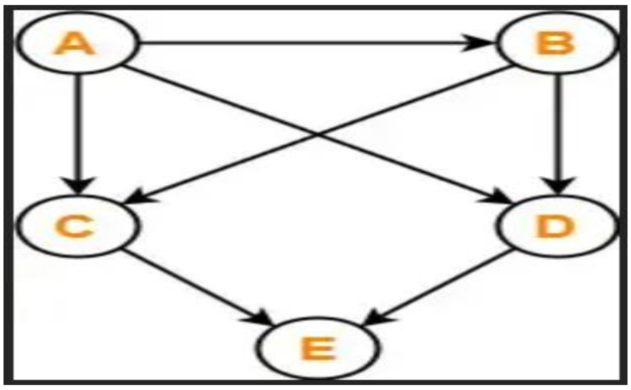
1 2 3 4 5 6
1 2 3 4 6 5
1 3 2 4 5 6
1 3 2 4 6 5

Applications of Topological Sort-

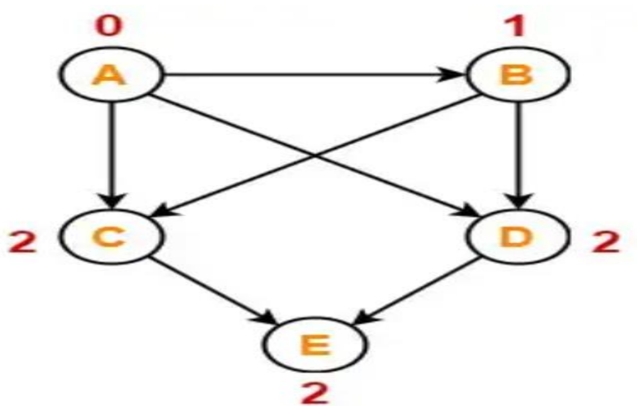
Few important applications of topological sort are-

- Scheduling jobs from the given dependencies among jobs
- Instruction Scheduling
- Determining the order of compilation tasks to perform in make files
- Data Serialization

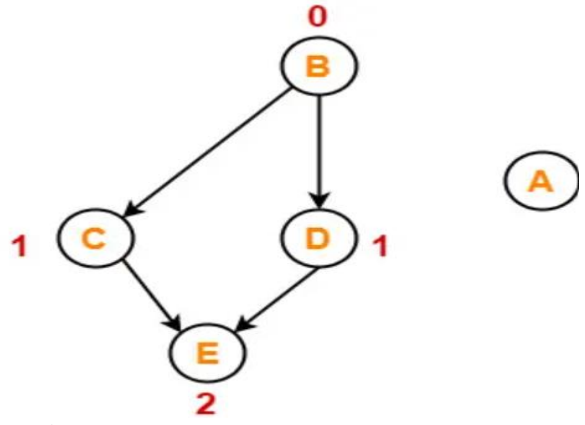
Find the number of different topological orderings possible for the given graph-



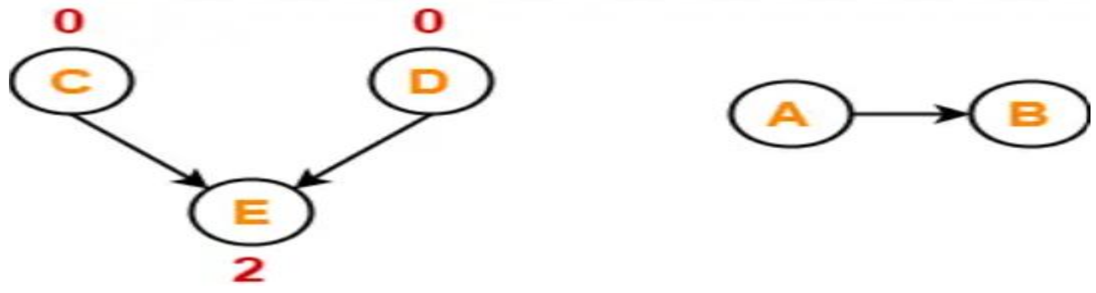
Step-01: Write in-degree of each vertex-



Step-02: Vertex-A has the least in-degree.
So, remove vertex-A and its associated edges. Now, update the in-degree of other vertices.

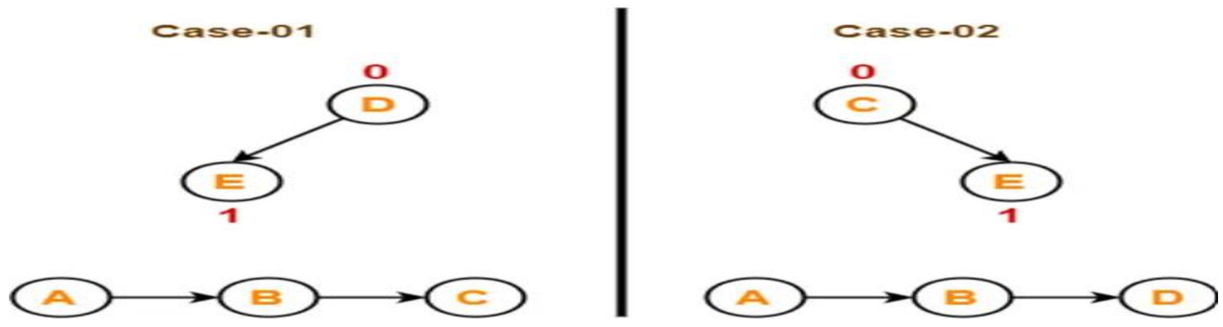


Step-03:
Vertex-B has the least in-degree. So, remove vertex-B and its associated edges. Now, update the in-degree of other vertices.

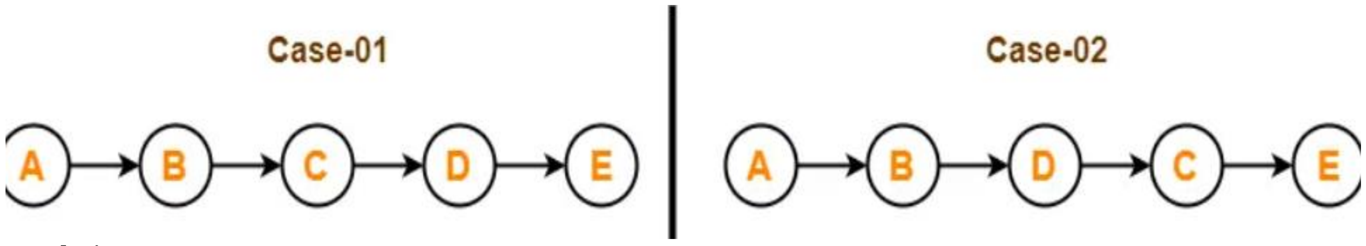


Step-04:

There are two vertices with the least in-degree. So, following 2 cases are possible-
In case-01,
Remove vertex-C and its associated edges Then, update the in-degree of other vertices.
In case-02,
Remove vertex-D and its associated edges. Then, update the in-degree of other vertices.



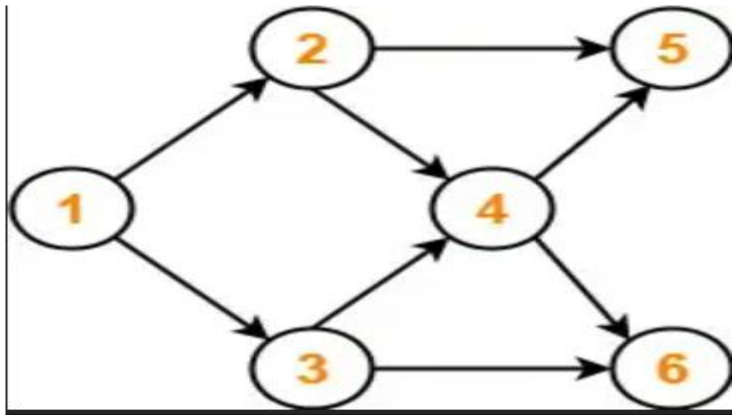
Step-05:
Now, the above two cases are continued separately in the similar manner.
In case-01,
Remove vertex-D since it has the least in-degree. Then, remove the remaining vertex-E.
In case-02,
Remove vertex-C since it has the least in-degree. Then, remove the remaining vertex-E.



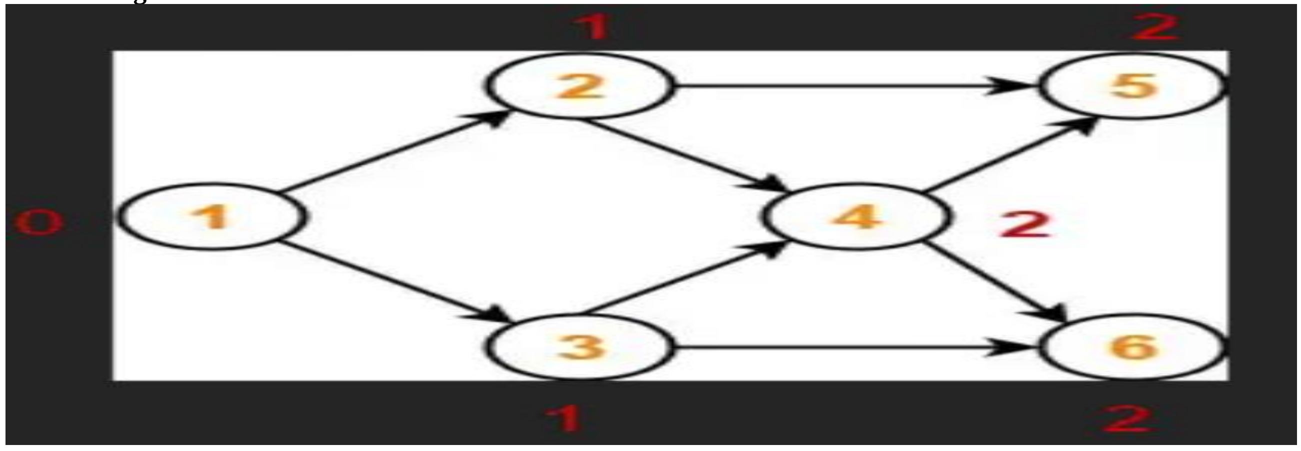
Conclusion-
For the given graph, following 2 different topological orderings are possible-

A B C D E or A B D C E

Problem-02: Find the number of different topological orderings possible for the given graph-

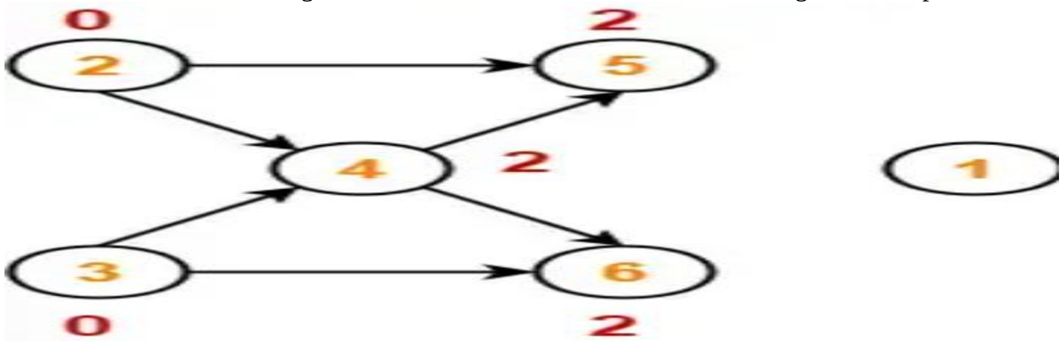


Solution-
The topological orderings of the above graph are found in the following steps-
Step-01:
Write in-degree of each vertex-

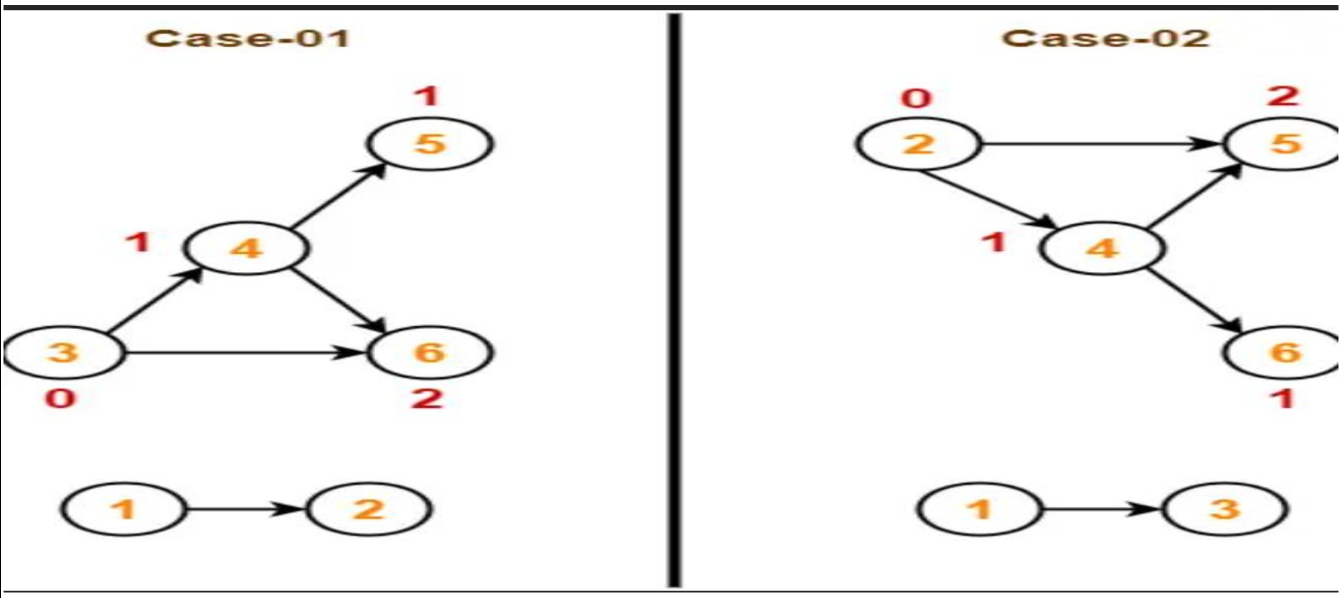


Step-02:

Vertex-1 has the least in-degree.So, remove vertex-1 and its associated edges. Now, update the in-degree of other vertices.

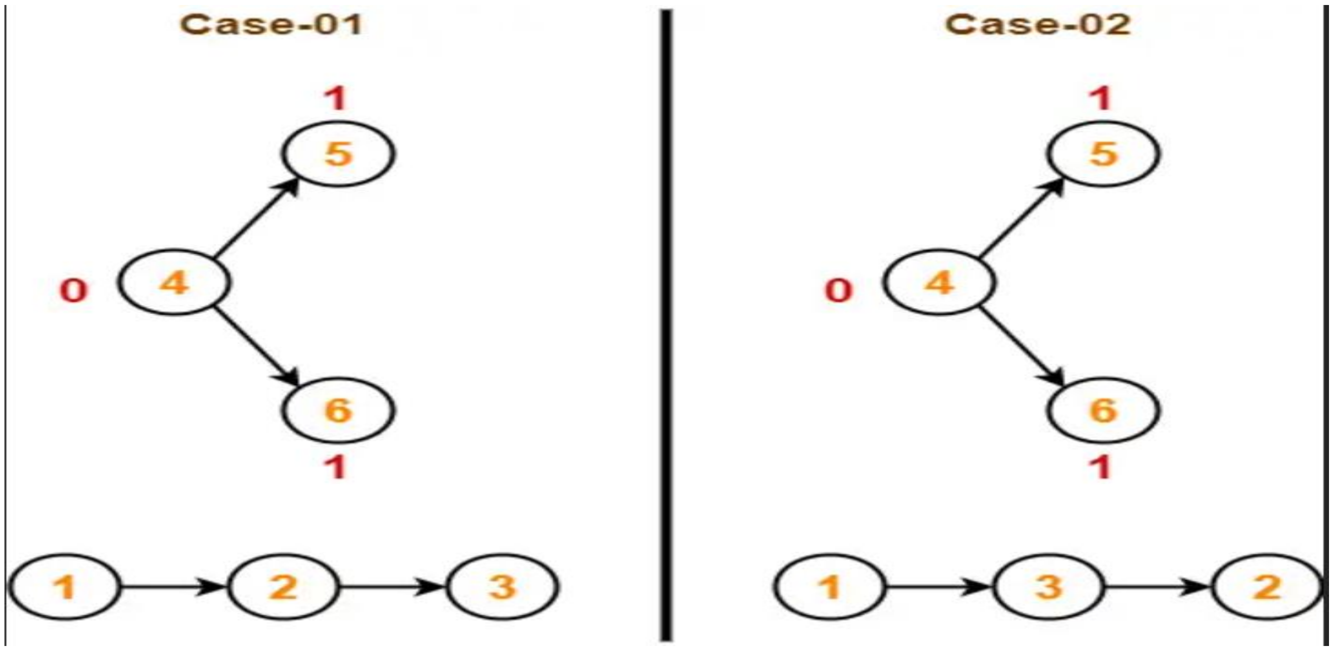


Step-03: There are two vertices with the least in-degree. So, following 2 cases are possible-
In case-01, Remove vertex-2 and its associated edges. Then, update the in-degree of other vertices.
In case-02,Remove vertex-3 and its associated edges. Then, update the in-degree of other vertices.

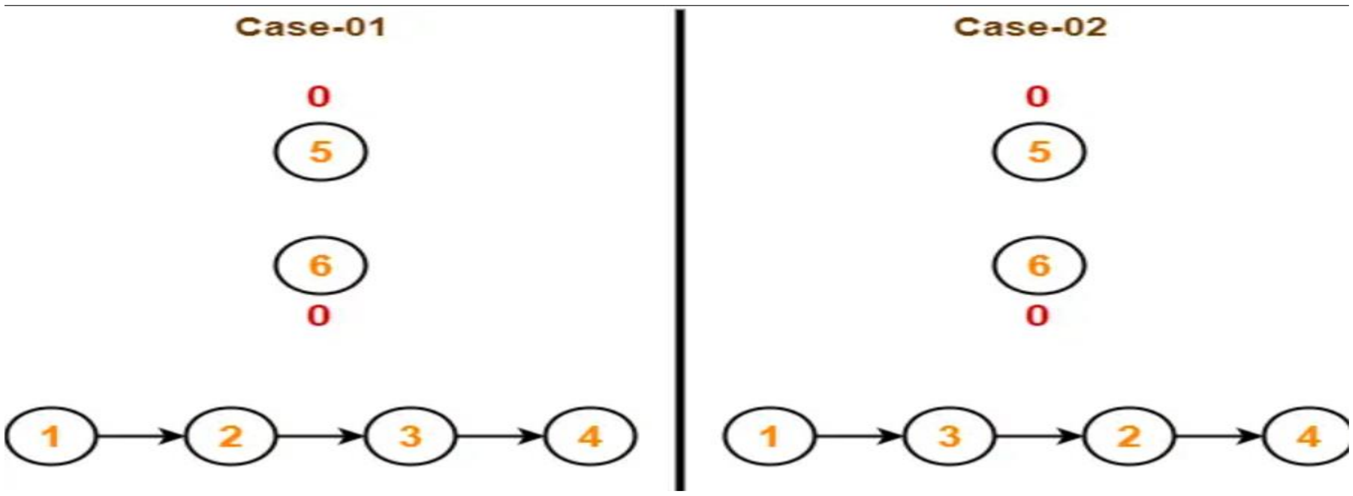


Step-04:

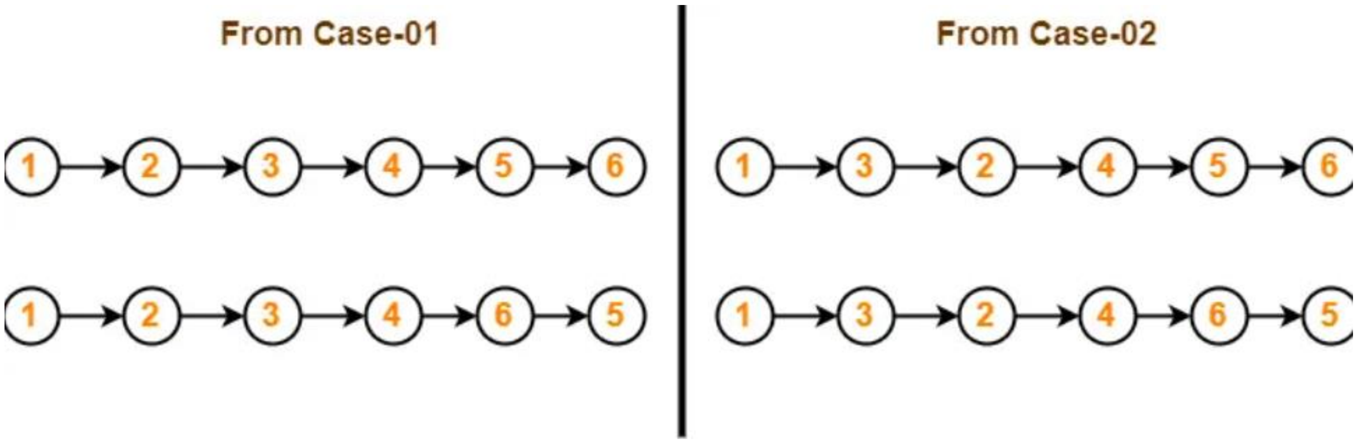
Now, the above two cases are continued separately in the similar manner.
In case-01,Remove vertex-3 since it has the least in-degree.Then, update the in-degree of other vertices.
In case-02, Remove vertex-2 since it has the least in-degree.Then, update the in-degree of other vertices.



Step-05:
In case-01, Remove vertex-4 since it has the least in-degree.Then, update the in-degree of other vertices.
In case-02, Remove vertex-4 since it has the least in-degree. Then, update the in-degree of other vertices.



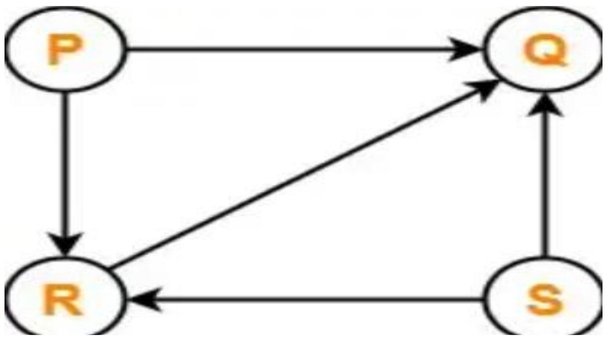
Step-06:
In case-01,There are 2 vertices with the least in-degree.So, 2 cases are possible.
Any of the two vertices may be taken first.
Same is with case-02.



Conclusion-

For the given graph, following 4 different topological orderings are possible-

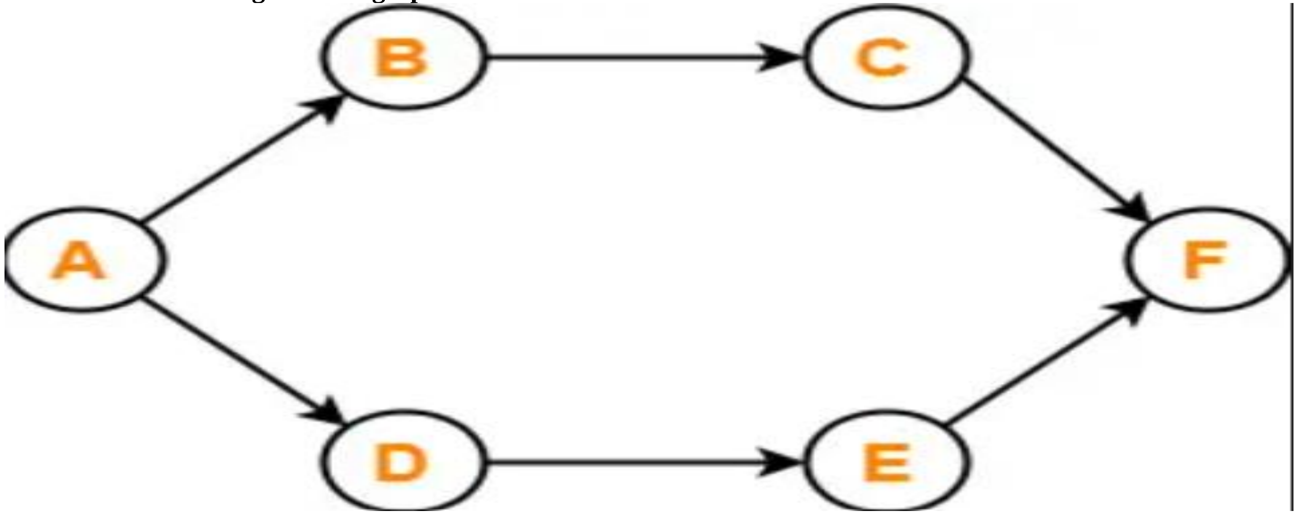
1 2 3 4 5 6
1 2 3 4 6 5
1 3 2 4 5 6
1 3 2 4 6
Problem-03:
Consider the directed graph given below. Which of the following statements is true?



1.The graph does not have any topological ordering.
2.Both PQRS and SRPQ are topological orderings.
3.Both PSRQ and SPRQ are topological orderings.
4.PSRQ is the only topological ordering.

Solution-
The given graph is a directed acyclic graph. So, topological orderings exist. P and S must appear before R and Q in topological orderings as per the definition of topological sort.
Thus, Correct option is (C).

Problem-04:
Consider the following directed graph-



The number of different topological orderings of the vertices of the graph is ________ ?
Solution-

Number of different topological orderings possible = 6.

Thus, Correct answer is 6.