UNIT-IV

Graph : Graph is a non-linear data structure. Generally, A graph G is represented as G = (V , E), **where**

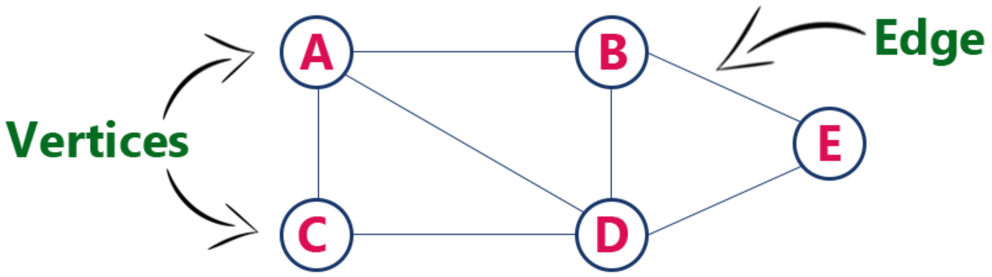
- V is set of vertices
- E is set of edges.

The most common representation of a graph is a diagram with Vertex and edges.

Vertex : Individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. Vertex are represented as points (or) small circles.

Edges : An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. Edges are represented as line segment (or) curve joining of its end vertices.

Example :



In The above graph with 5 vertices and 7 edges.

This graph G can be defined as G = (V , E)

Where

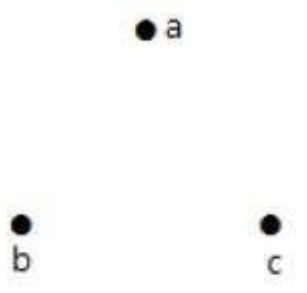
V = {A,B,C,D,E} and

E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.

Graph Terminology

1. Null Graph : Agraph in which number of edges is zero is called Null Graph.

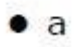
Example :



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.

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

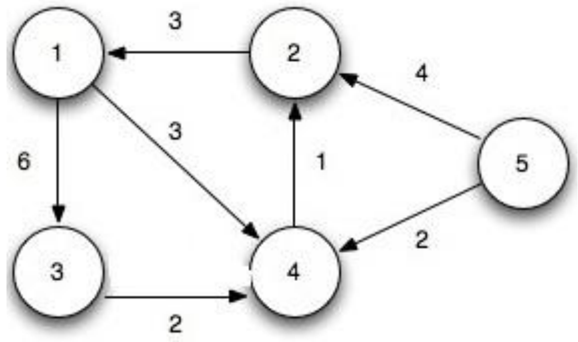
Example :



In the above graph, there is only one vertex 'a' with no other edges. Hence it is a Trivial graph.

3. Weighted Graphs: A graph G=(V, E) is called a weighted graph if each edge of graph G is assigned a positive number w called the weight of the edge e.

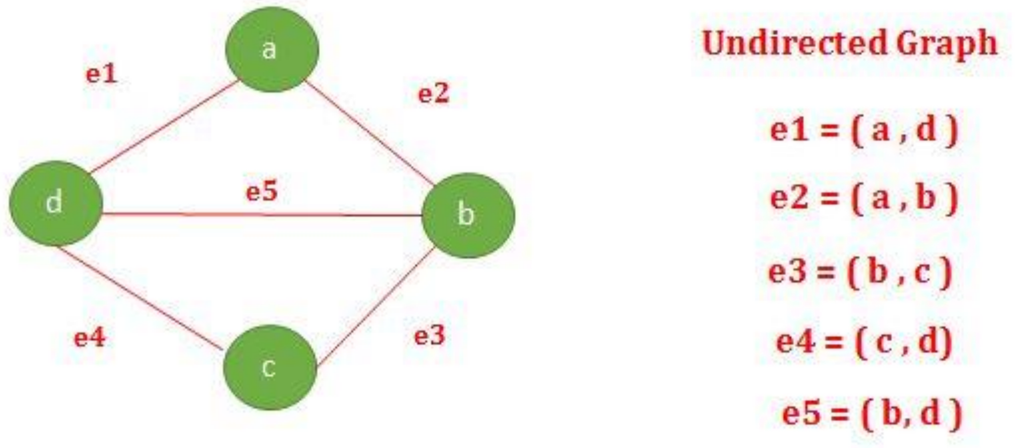
Example :



Here 1, 2, 3, 4, 6 are weights assigned to each edge respectively.

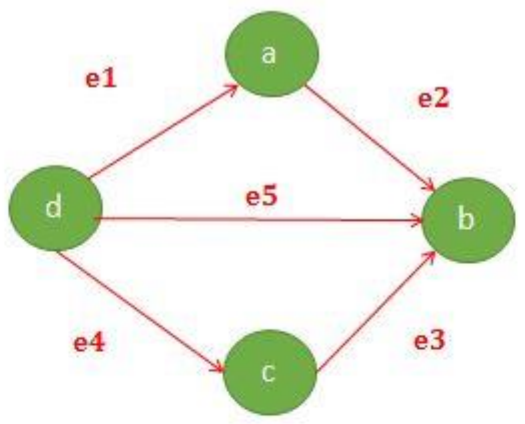
4. Undirected Graph : There is no direction associated with each node is known as Undirected Graph. In a undirected graph each edge is represented as an unordered pair of vertices, that is (a, b) and (b, a) represents the same edge.

Example : Undirected Graph



5. Directed Graph : If a direction associated if each edge is known as directed graph. In a directed graph each edge is represented by an order pair of vertices. If (a, b) and (b, a) represents two different edges.

Example : Directed Graph



Directed Graph

e1 = (d , a)

e2 = (a , b)

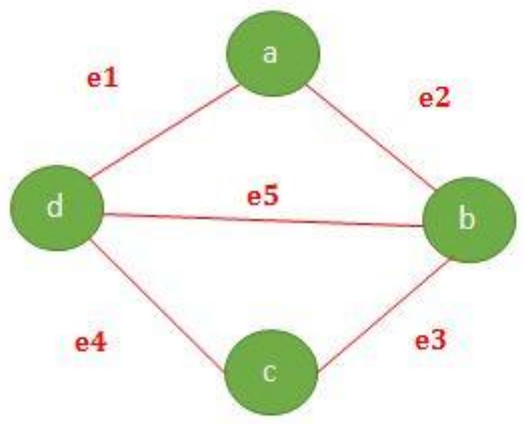
e3 = (c , b)

e4 = (d , c)

e5 = (d , b)

6. Degree of vertex : The graph must be undirected graph. Total number of edges connected to a vertex is said to be degree of that vertex.

Example : Undirected Graph



Degree of Vertex

d (a) = 2

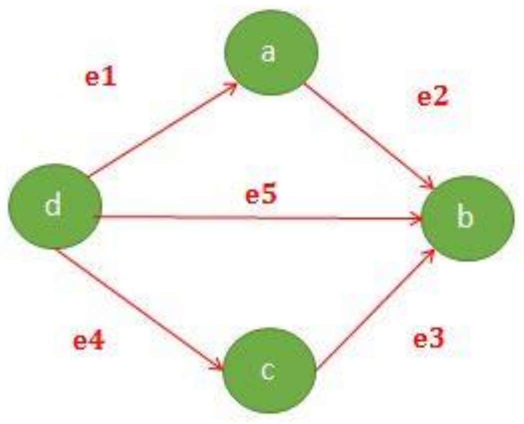
d (b) = 3

d (c) = 2

d (d) = 3

7. Indegree of vertex : The graph must be directed graph. Total number of incoming edges connected to a vertex is said to be indegree of that vertex. . It is denoted by $d^+(v)$.

Example : Directed Graph



In-Degree of Vertex

$d^+(a) = 1$

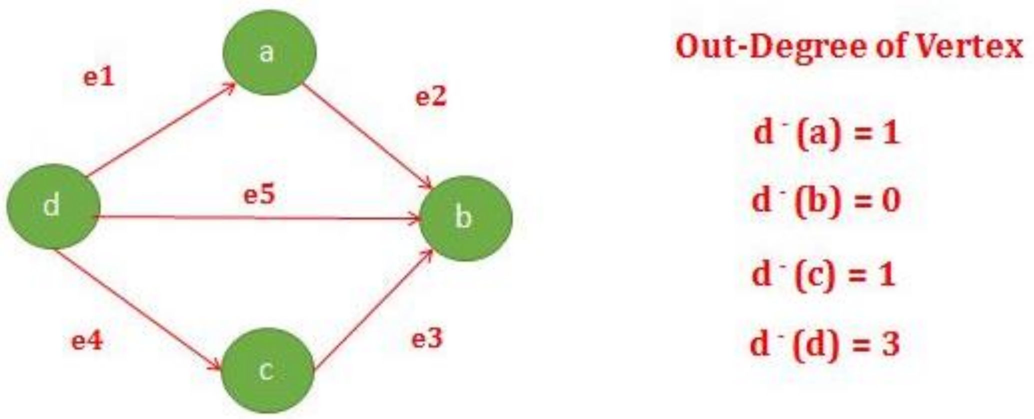
$d^+(b) = 3$

$d^+(c) = 1$

$d^+(d) = 0$

8. Outdegree of Vertex : The graph must be directed graph. Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex. . It is denoted by $d^{-}(v)$.

Example : Directed Graph



Out-Degree of Vertex

$d^{-}(a) = 1$

$d^{-}(b) = 0$

$d^{-}(c) = 1$

$d^{-}(d) = 3$

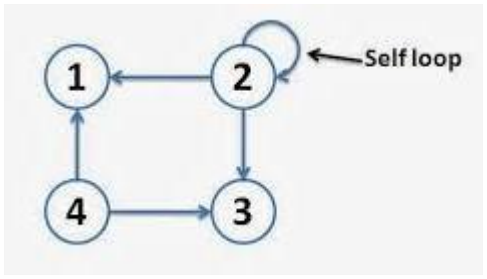
9. Isolated Vertex : if degree of vertex is 0 then that vertex is called Isolated vertex.

10. Pendant Vertex : if the degree of vertex is 1 then that vertex is called Pendant vertex.

11. Even Vertex : if the degree of a vertex is even no then that vertex is called Even vertex.

12. Odd Vertex : if the degree of a vertex is odd no then that vertex is called Odd vertex.

13. Self-loop : An edge joining a vertex to itself is called self-loop.



Graph Representations (or) Graph Implementation Methods

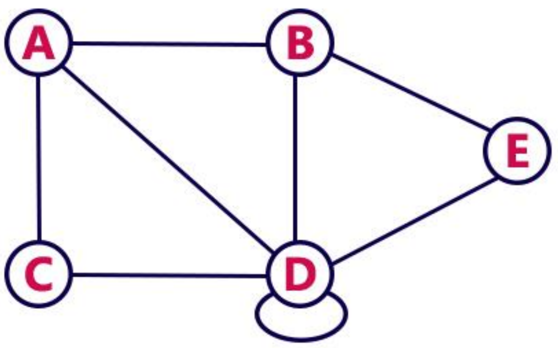
Graph data structure is represented using following representations...

1. Adjacency Matrix
2. Incidence Matrix
3. Adjacency List

1. Adjacency Matrix

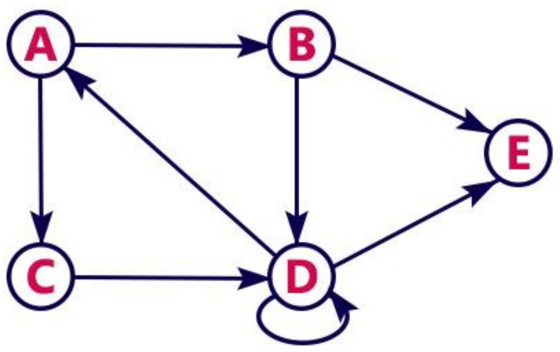
In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. This matrix is filled with either 1 or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

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



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

Directed graph representation...



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

2. Incidence Matrix

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of edges. That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6. In this matrix, rows represent vertices and columns represents edges. This matrix is filled with 0 or 1 or -1.

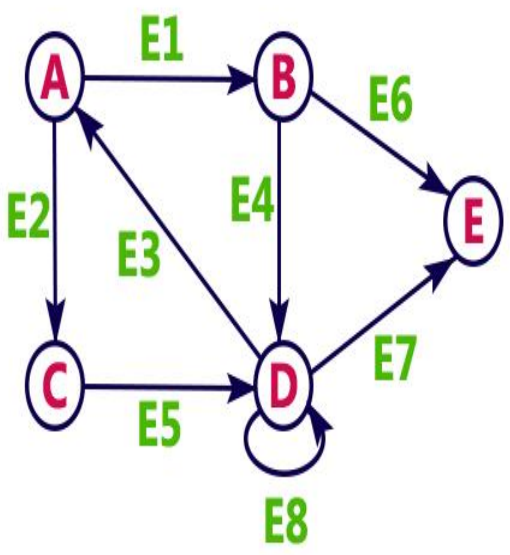
Here,

0 represents that the row edge is not connected to column vertex,

1 represents that the row edge is connected as the outgoing edge to column vertex and

-1 represents that the row edge is connected as the incoming edge to column vertex.

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

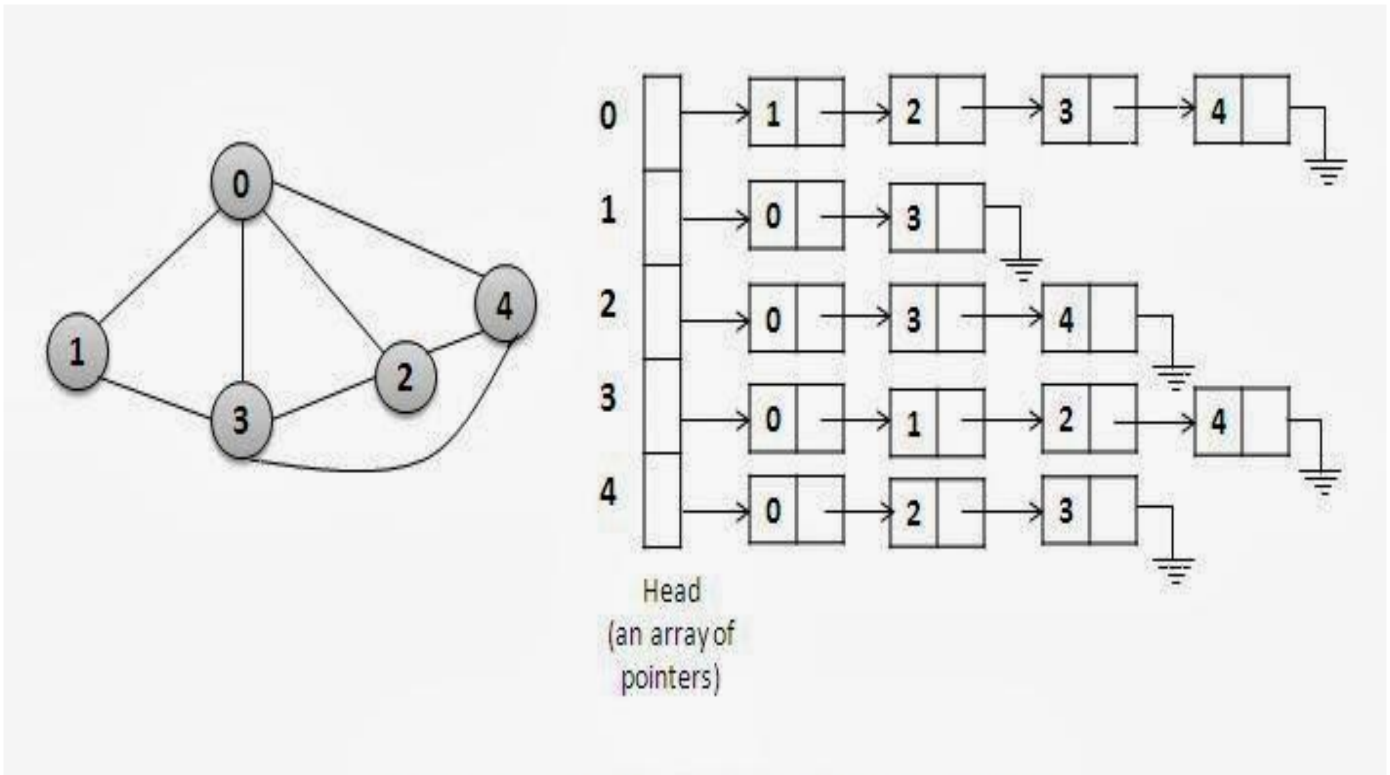


$$\begin{array}{c|cccccccc} & E1 & E2 & E3 & E4 & E5 & E6 & E7 & E8 \\ \hline A & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ B & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ C & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 1 & -1 & -1 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \end{array}$$

3. Adjacency List

A graph can also be represented using a linked list. For each vertex, a list of adjacent vertices is maintained using a linked list. It creates a separate linked list for each vertex Vi in the graph G = (V, E). Adjacency list of a graph with n nodes can be represented by an array of pointers. Each pointer points to a linked list of the corresponding vertex.

The adjacency list representation of a graph.



Adjacency List Representation of Graph

Graph Traversal Methods

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. BFS (Breadth First Search) - Queue
2. DFS (Depth First Search) - Stack

1. 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.

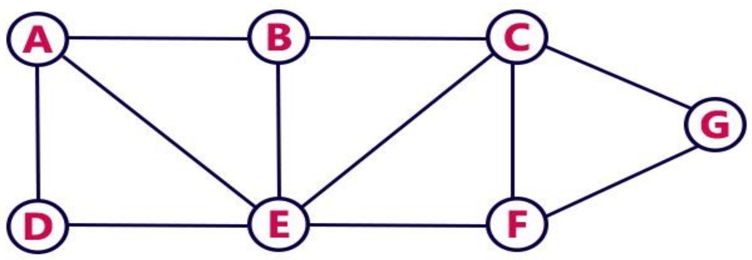
Step 3 . Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.

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

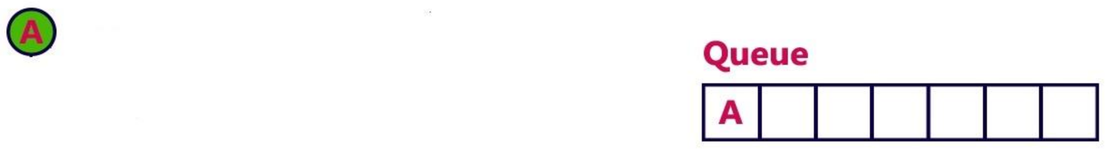
Example : Consider the following example graph to perform BFS Traversal.



Solution :

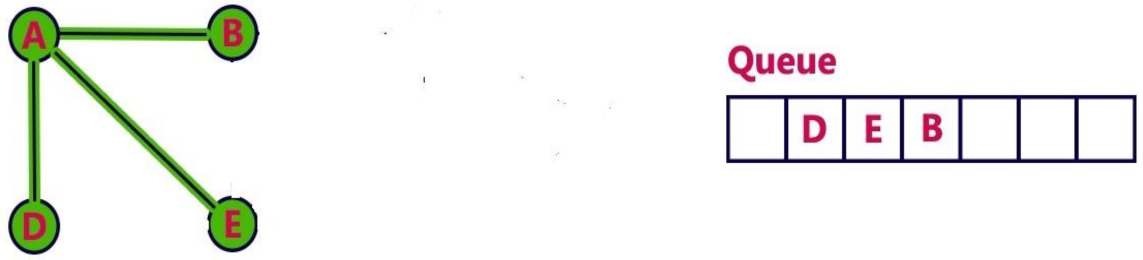
Step 1:

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



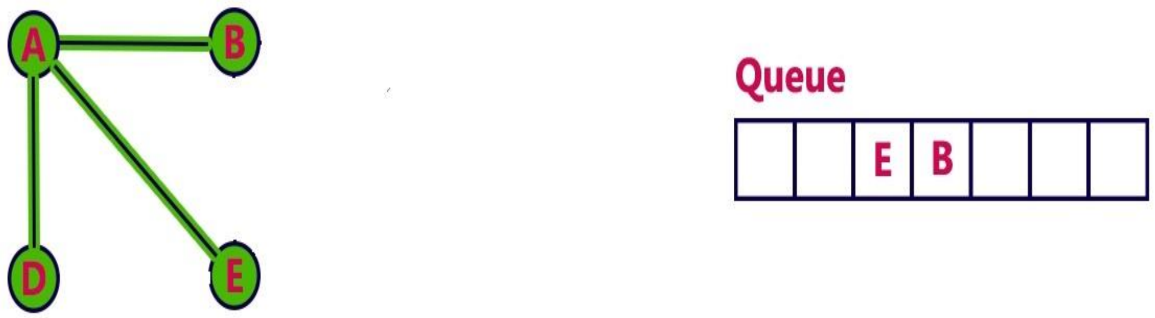
Step 2:

- 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..



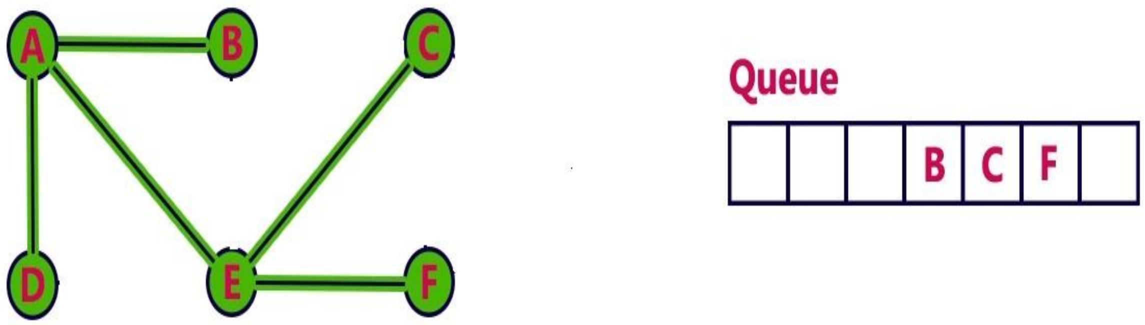
Step 3:

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



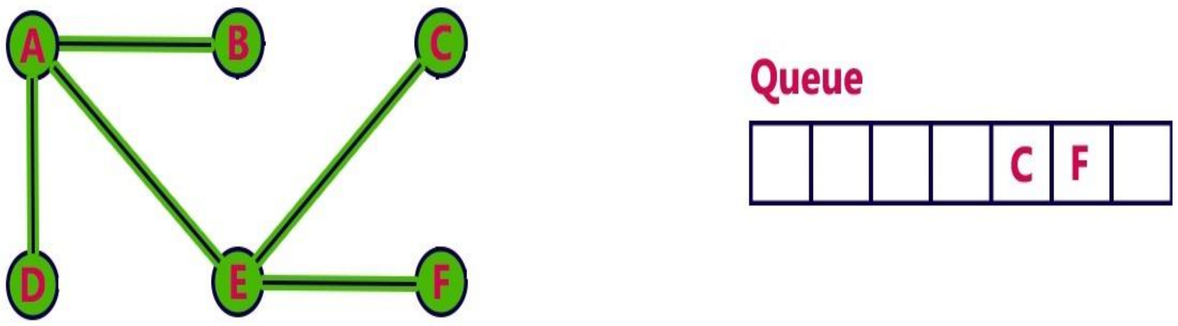
Step 4:

- 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.



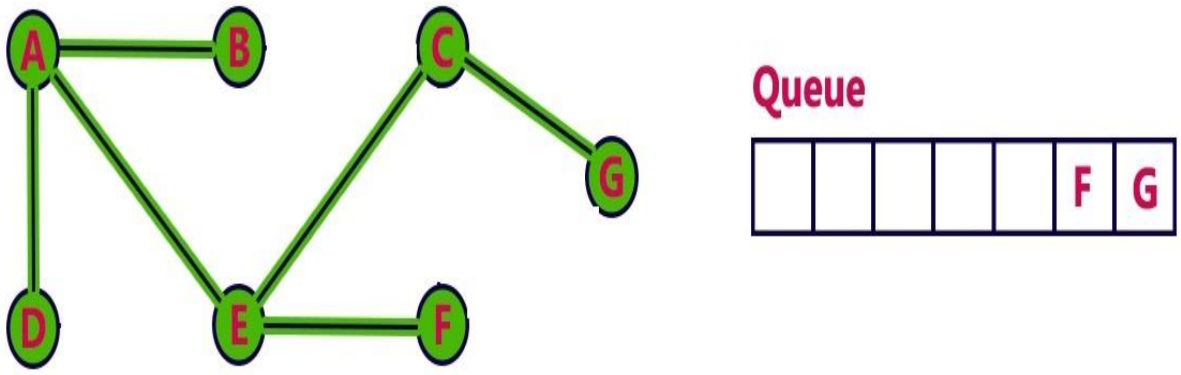
Step 5:

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



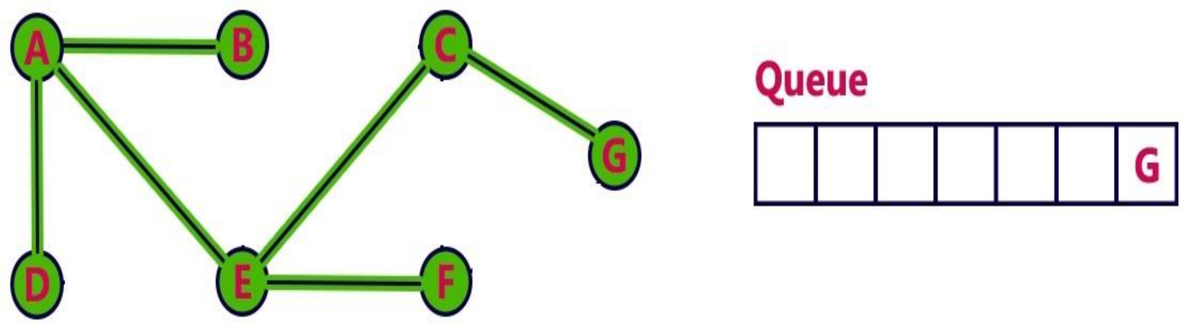
Step 6:

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



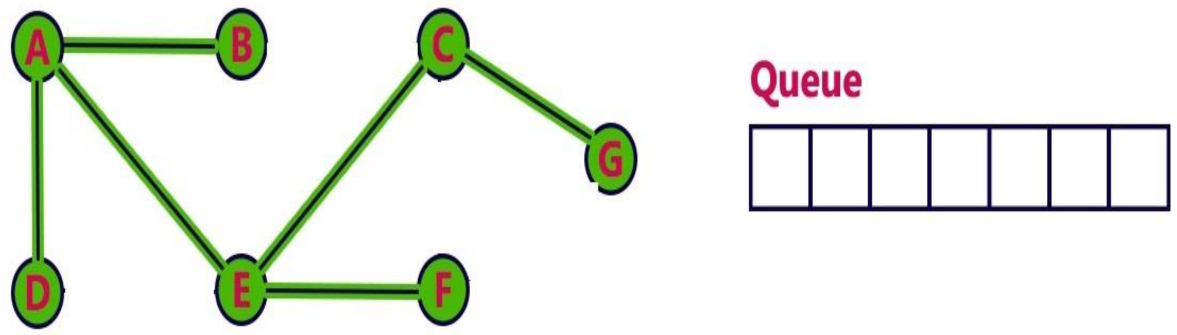
Step 7:

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



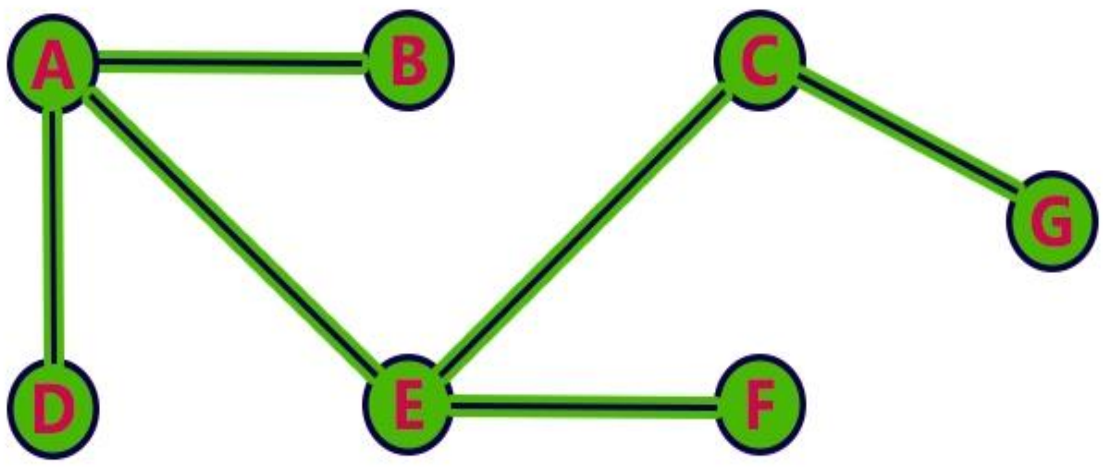
Step 8:

- 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.



2 .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.

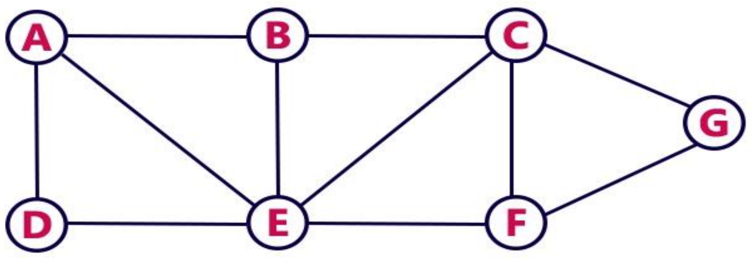
Step 4. Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

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

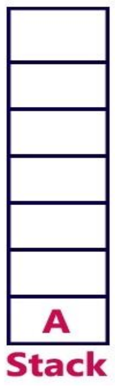
Example : Consider the following example graph to perform DFS Traversal.



Solution :

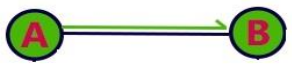
Step 1:

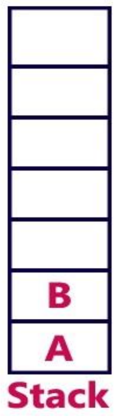
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



Step 2:

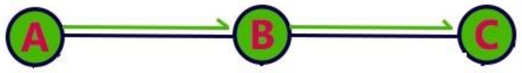
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

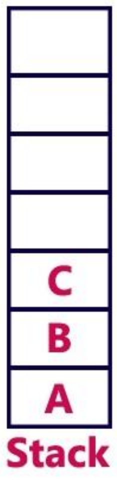




Step 3:

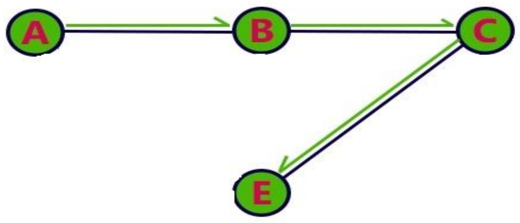
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

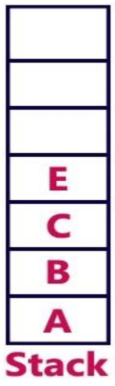




Step 4:

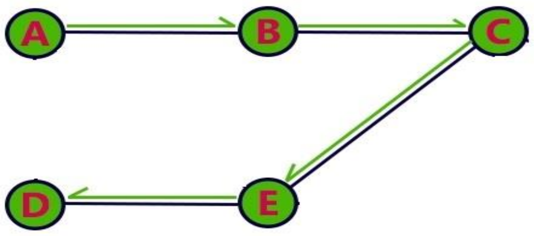
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

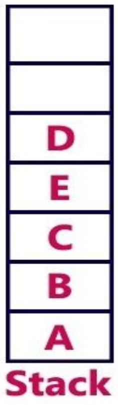




Step 5:

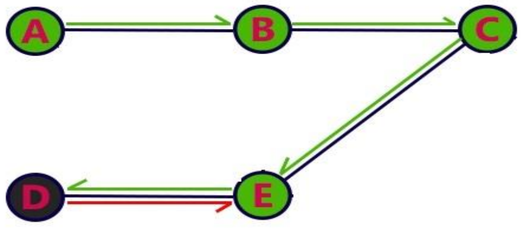
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

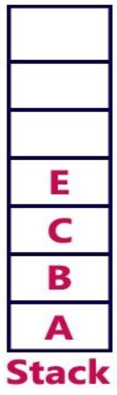




Step 6:

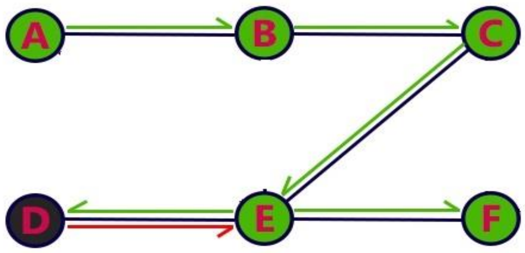
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

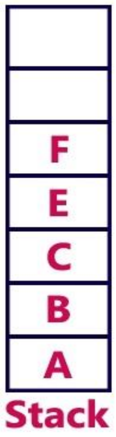




Step 7:

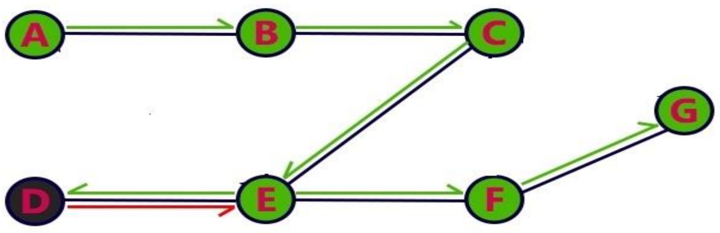
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

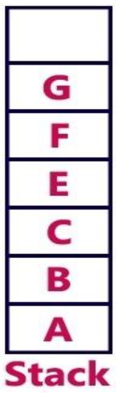




Step 8:

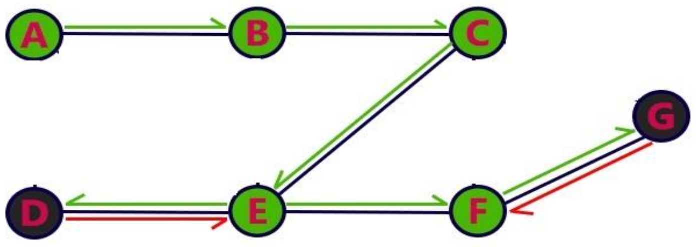
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

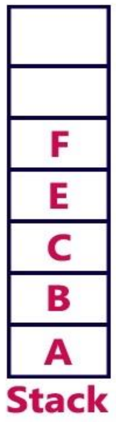




Step 9:

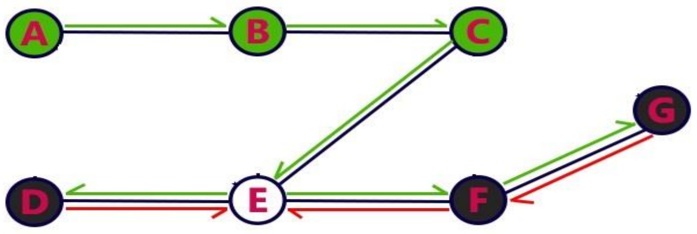
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

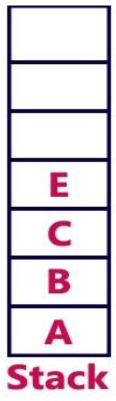




Step 10:

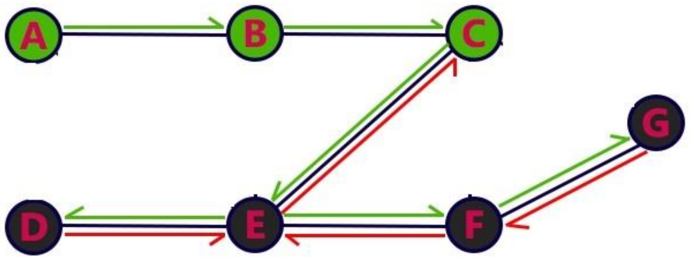
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

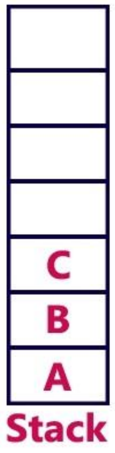




Step 11:

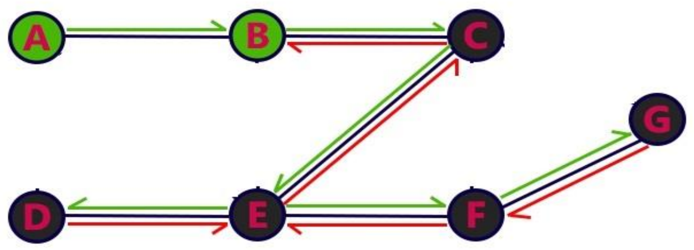
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

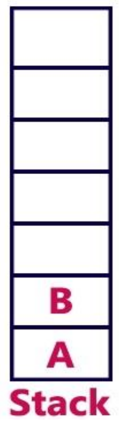




Step 12:

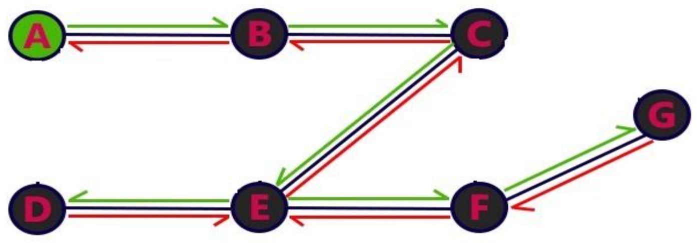
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

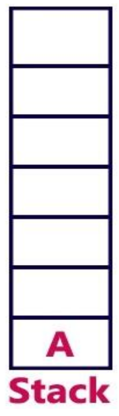




Step 13:

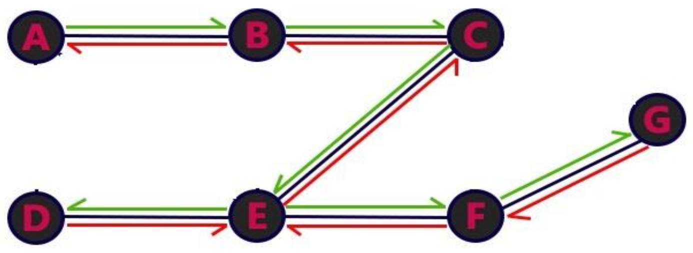
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

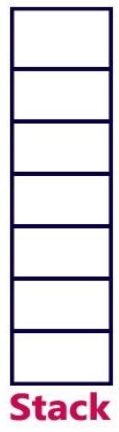




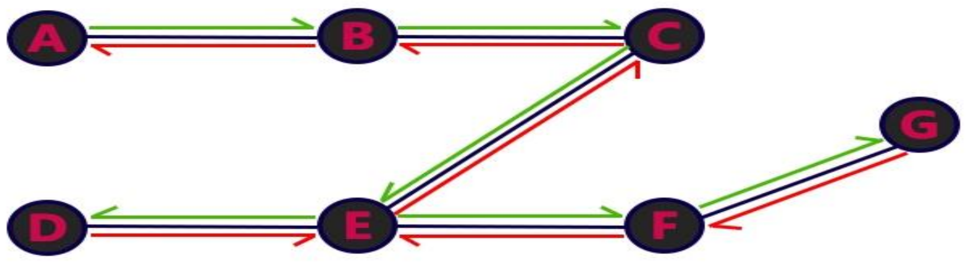
Step 14:

- 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.



UNIT- IV

Sorting :

- Sorting is nothing but arranging the elements in ascending or descending order.
- Sorting arranges data in a sequence which makes searching easier.

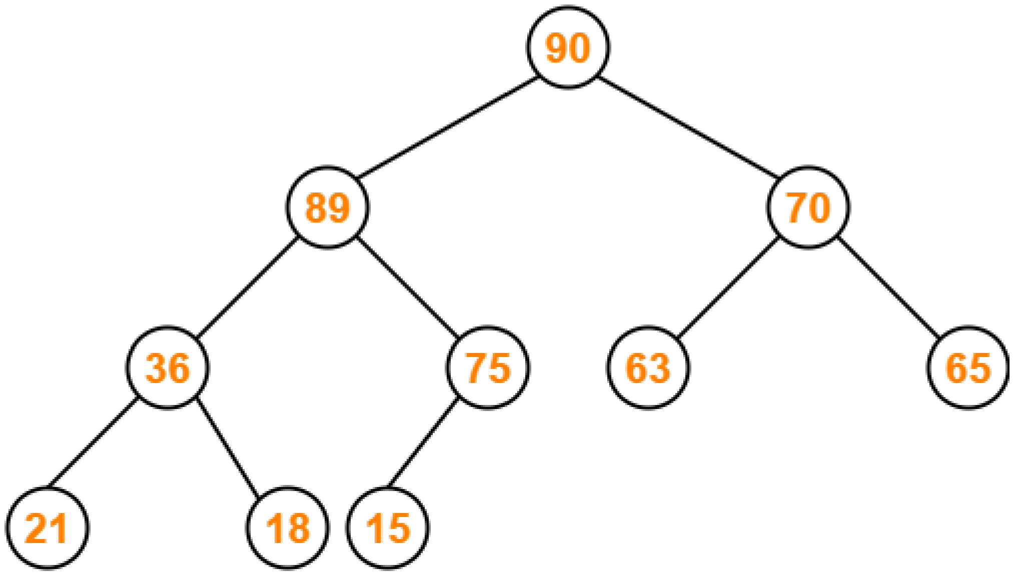
Heap : Heap is a data structure. A complete binary tree in which the value node is either greater than or equal to values in child nodes or less than or equal to values in child nodes is called a heap.

Heap can be classified into two types

1. Max Heap
2. Min Heap

1. Max Heap : In max heap, every node contains greater or equal value element than its child nodes are called Max Heap. Thus, root node contains the largest value element.

Example : Consider the following example of max heap-



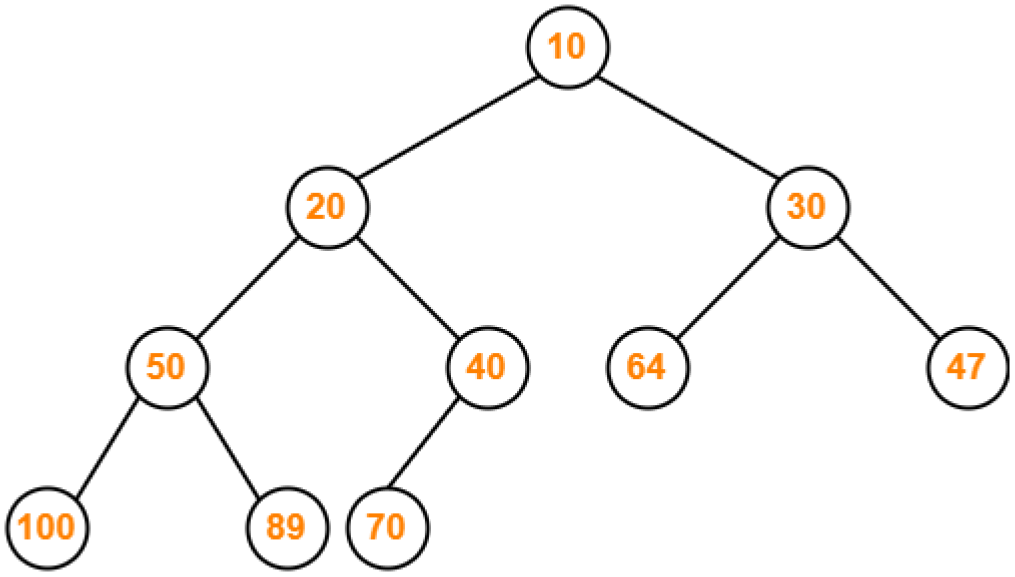
Max Heap Example

This is max heap because-

- Every node contains greater or equal value element than its child nodes.
- It is an almost complete binary tree with its last level strictly filled from left to right.

2. Min Heap : In min heap, every node contains lesser value element than its child nodes are called Min Heap. Thus, root node contains the smallest value element.

Example : Consider the following example of min heap-



Min Heap Example

This is min heap because-

- Every node contains lesser value element than its child nodes.
- It is an almost complete binary tree with its last level strictly filled from left to right.

Insertion Operation in Max Heap

Insertion Operation in max heap is performed as follows...

Step 1 - Insert the **newNode** as **last leaf** from left to right.

Step 2 - Compare **newNode value** with its **Parent node**.

Step 3 - If **newNode value is greater** than its parent, then **swap** both of them.

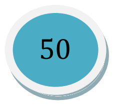
Step 4 - Repeat step 2 and step 3 until newNode value is less than its parent node (or) newNode reaches to root.

Example : Consider the following max heap-

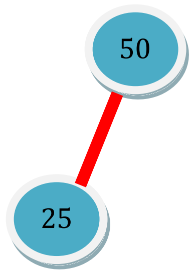
50, 25, 30, 75, 100, 45, 80

Solution :

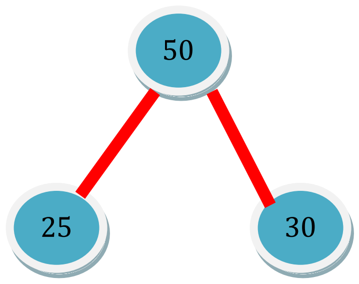
Step 1 : Insert 50



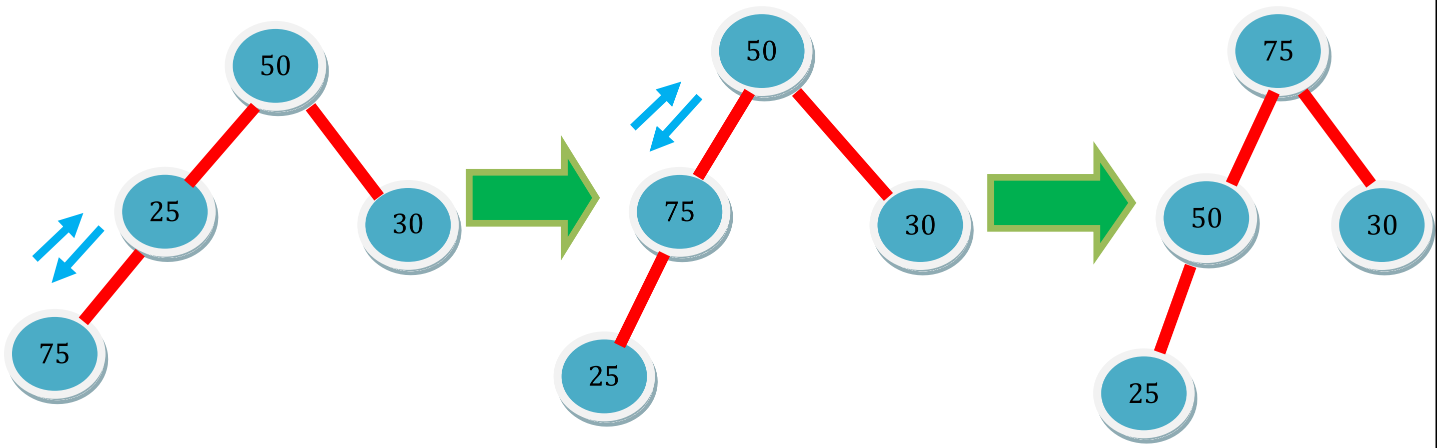
Step 2 : Insert 25



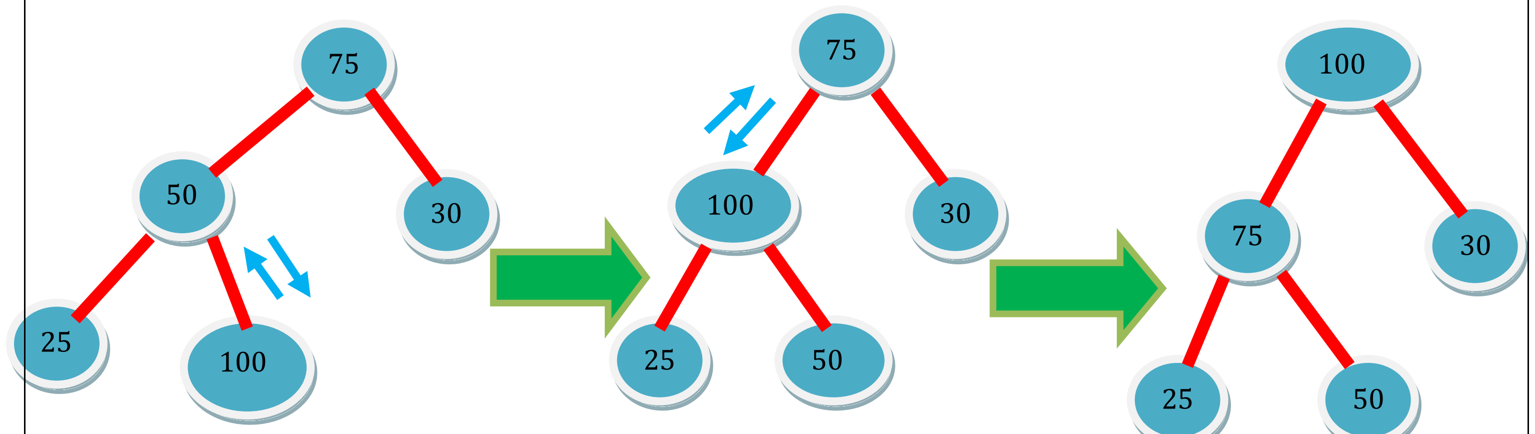
Step 3 : Insert 30



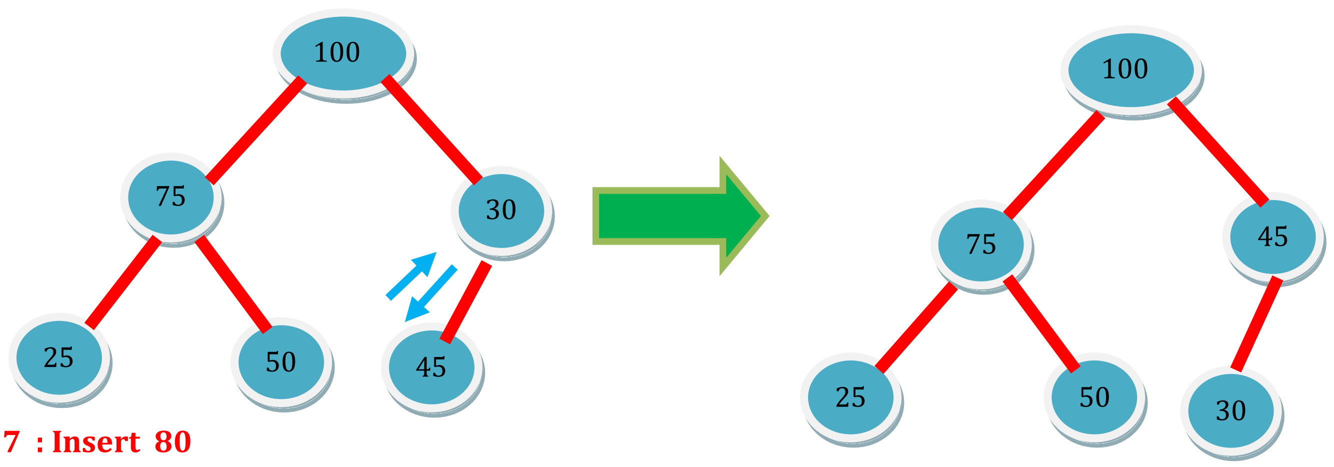
Step 4 : insert 75



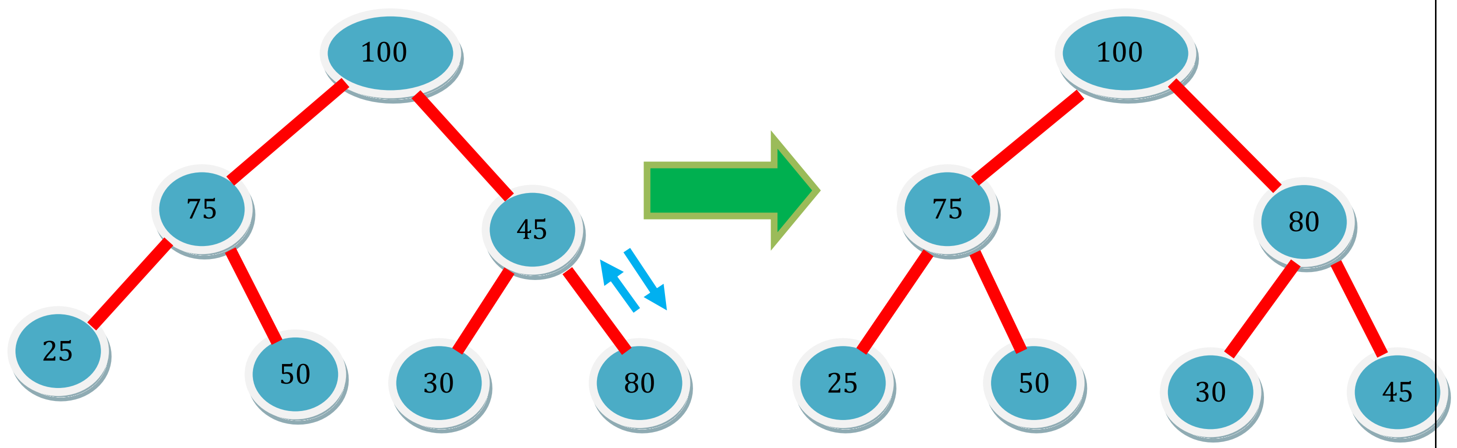
Step 5 : Insert 100



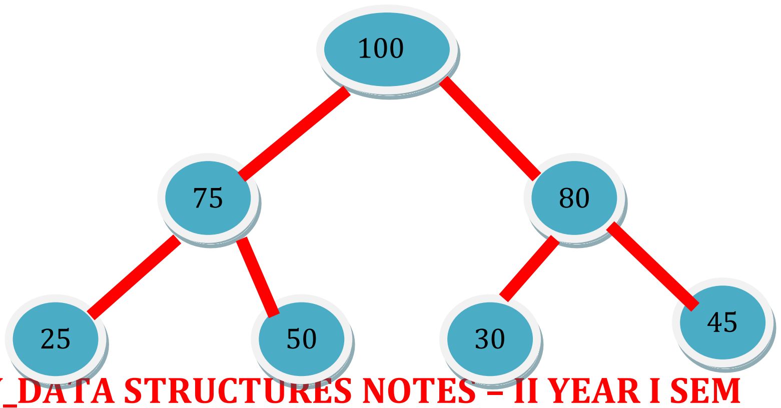
Step 6 : Insert 45



Step 7 : Insert 80



The Final Max Heap is

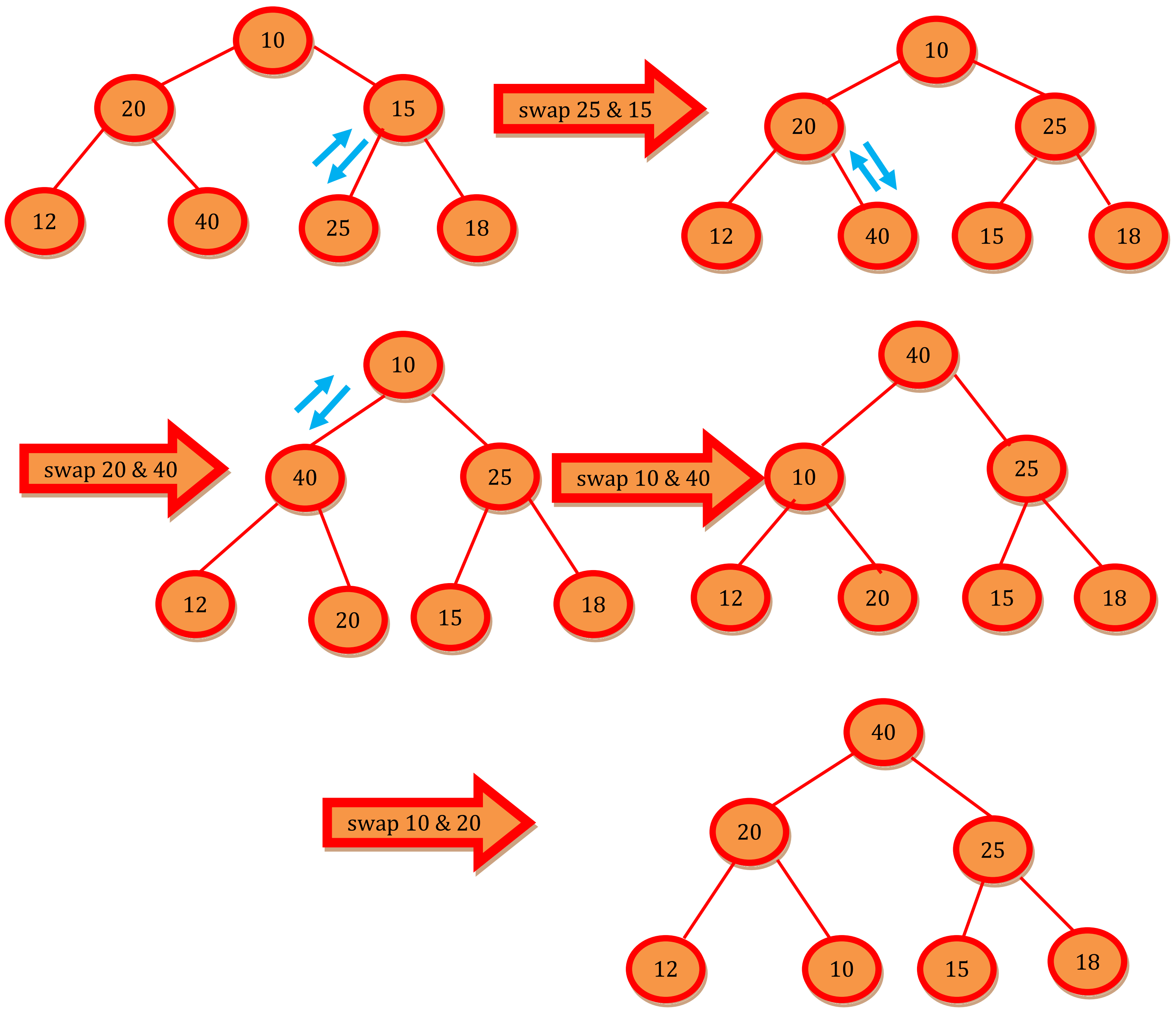


Heap Sort : 10 , 20 , 15 , 12 , 40 , 25 , 18

Heap sort algorithm requires two phases.

1 : Constructing a heap.
2 : Sorting Heap.

1 : Constructing a Heap : Construct a heap with given list of unsorted numbers and convert to max heap.



Sorting Heap: The following steps sort the heap.

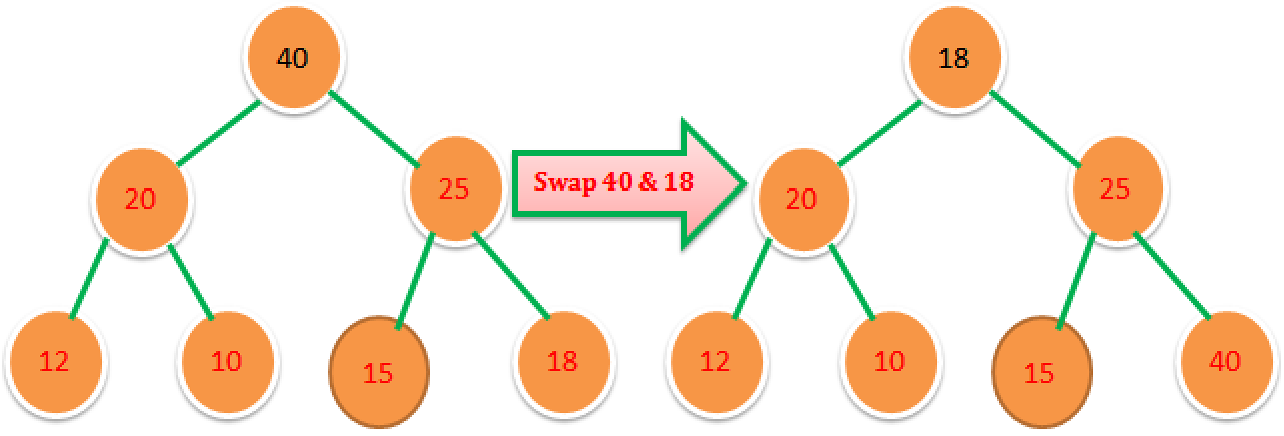
1 : Swap the Root node with the last node of the heap.Now, last node is in its proper place.

2: Delete last element , Put the delete element into the sorted list and consider the remaining elements as the new list.

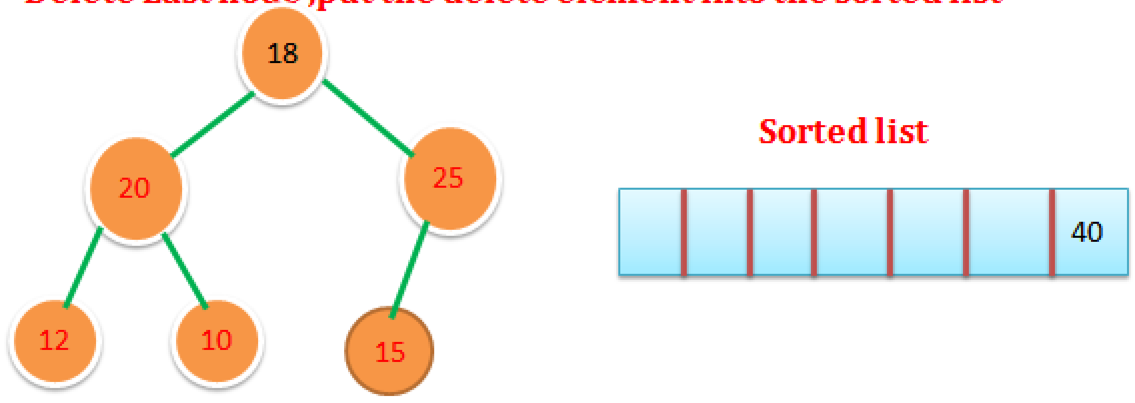
3: Heap after deleted element from max heap, The new list may not be in heap form. Therefore, construct a heap of new list.

4: Repeat steps (1) , (2) and (3) until all elements are placed at their proper place i.e, until all elements are sorted.

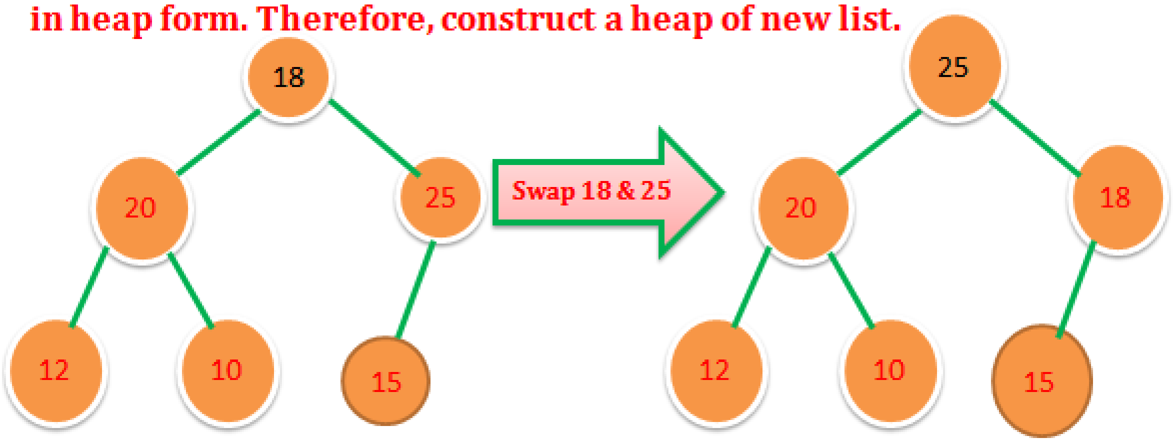
Step1 : Swap the root node with the last node



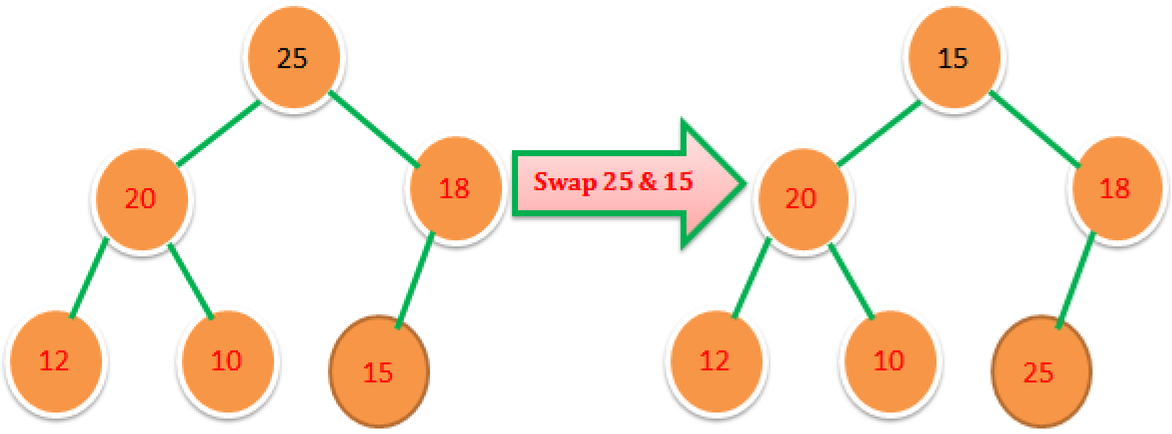
Delete Last node ,put the delete element into the sorted list



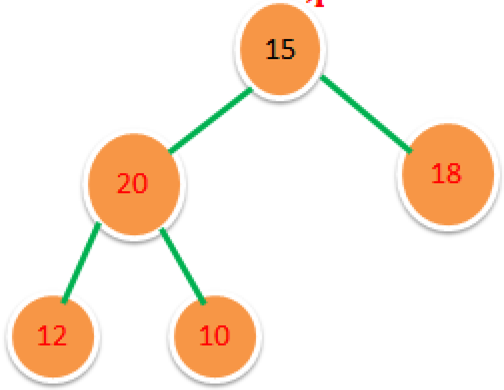
Heap after deleted element from max heap, The new list may not be in heap form. Therefore, construct a heap of new list.



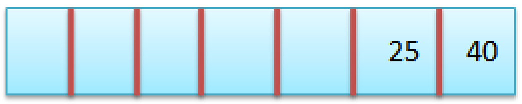
Step2 : Swap the root node with the last node



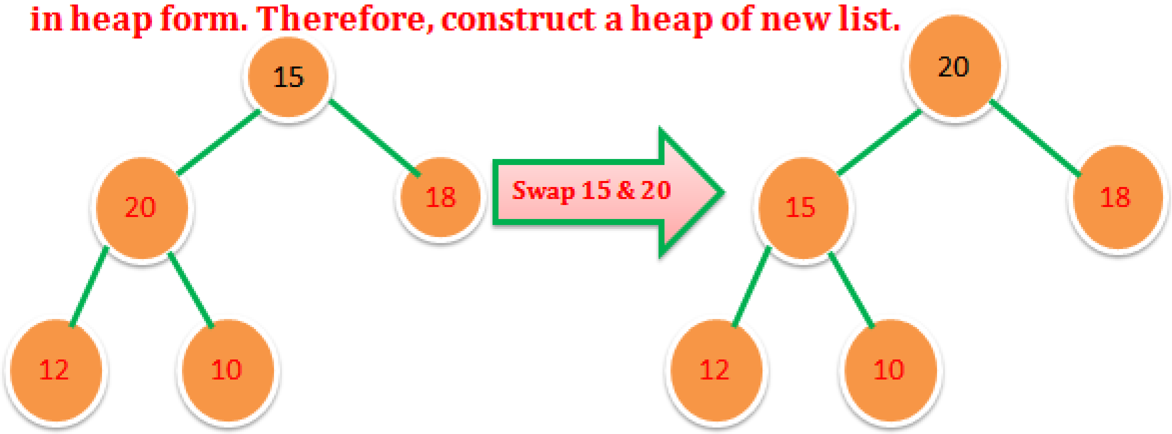
Delete Last node ,put the delete element into the sorted list



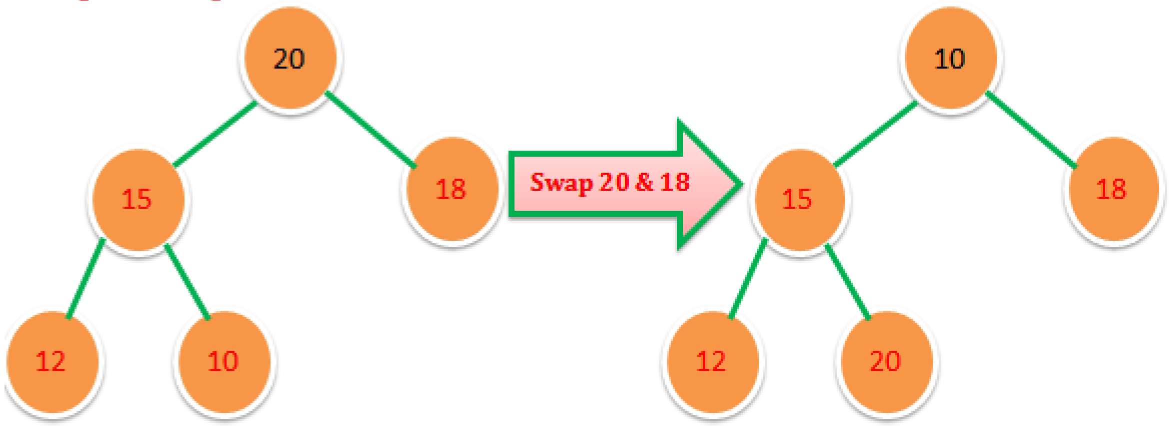
Sorted list



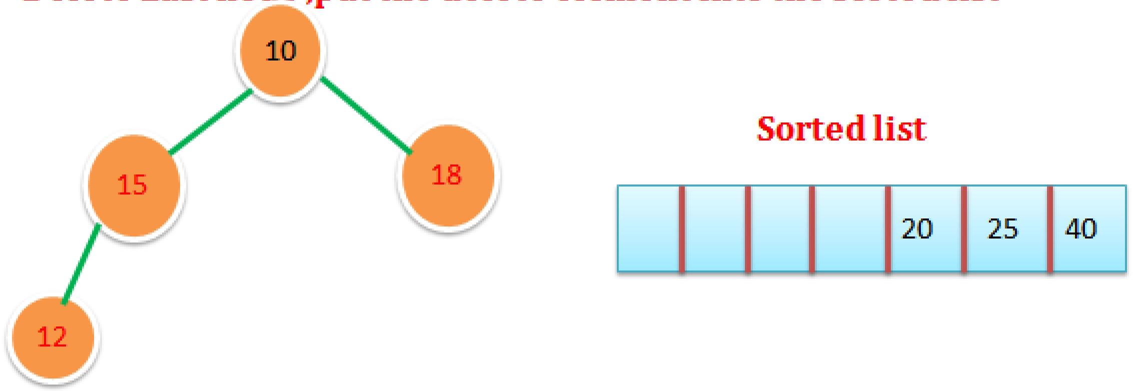
Heap after deleted element from max heap, The new list may not be in heap form. Therefore, construct a heap of new list.



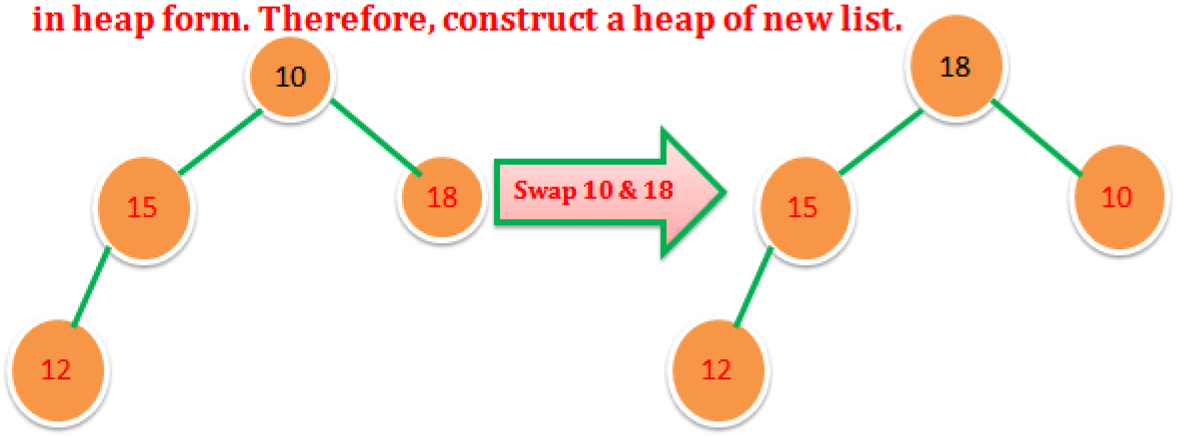
Step3 : Swap the root node with the last node



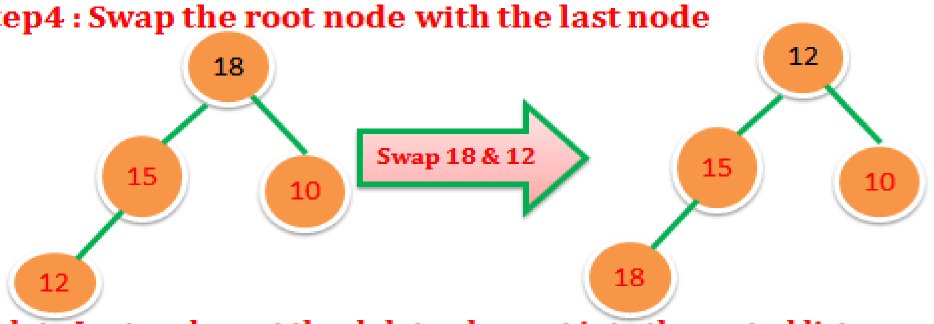
Delete Last node ,put the delete element into the sorted list



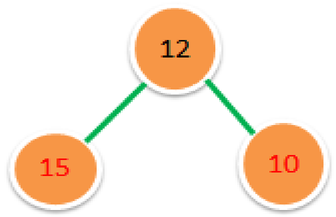
Heap after deleted element from max heap, The new list may not be in heap form. Therefore, construct a heap of new list.



Step4 : Swap the root node with the last node



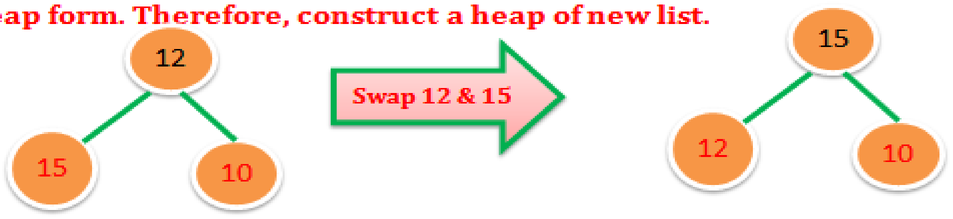
Delete Last node ,put the delete element into the sorted list



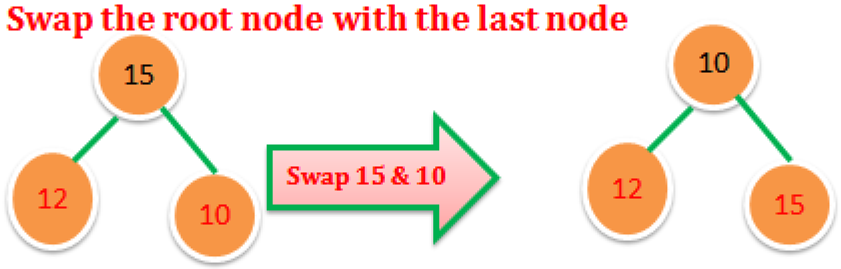
Sorted list

			18	20	25	40

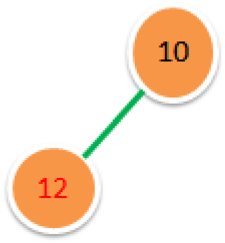
Heap after deleted element from max heap, The new list may not be in heap form. Therefore, construct a heap of new list.



Step 5 : Swap the root node with the last node



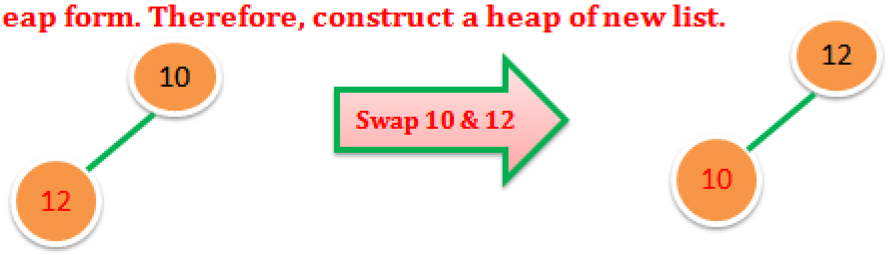
Delete Last node ,put the delete element into the sorted list



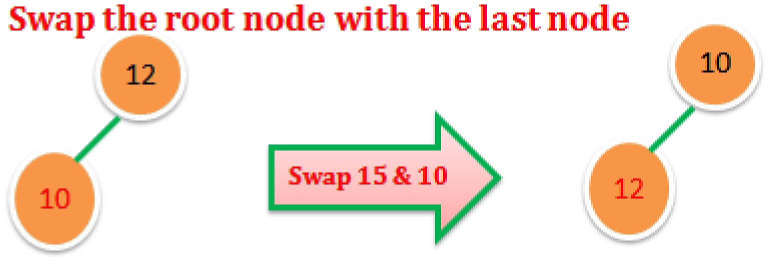
Sorted list

		15	18	20	25	40

Heap after deleted element from max heap, The new list may not be in heap form. Therefore, construct a heap of new list.



Step 6 : Swap the root node with the last node



Delete Last node ,put the delete element into the sorted list

Sorted list

	12	15	18	20	25	40

Heap after deleted element from max heap, The new list may not be in heap form. Therefore, construct a heap of new list.

Finally the Sorted list is

10	12	15	18	20	25	40

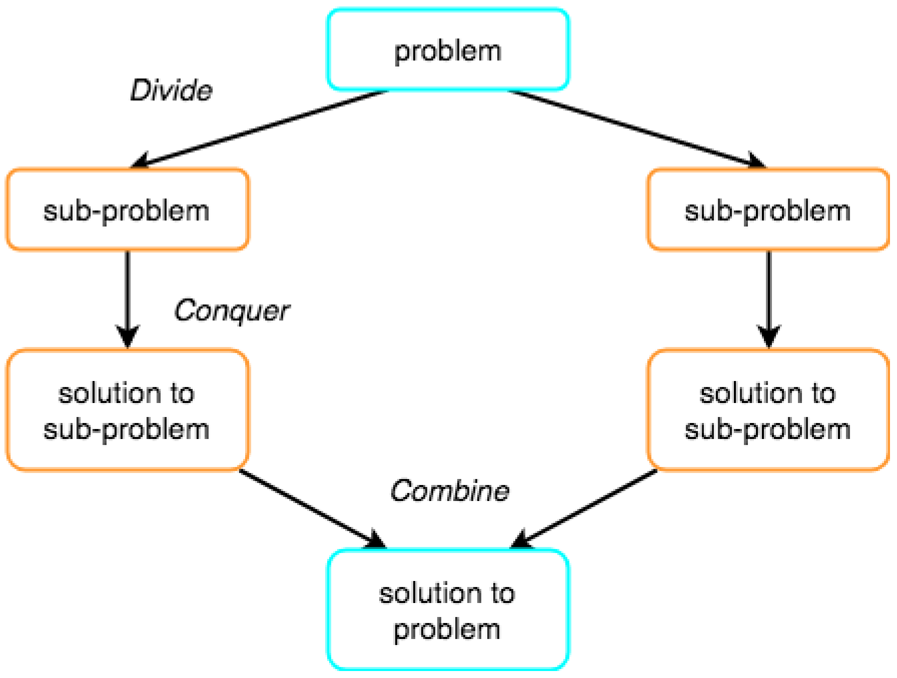
Merge Sort : Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.

The concept of Divide and Conquer involves three steps:

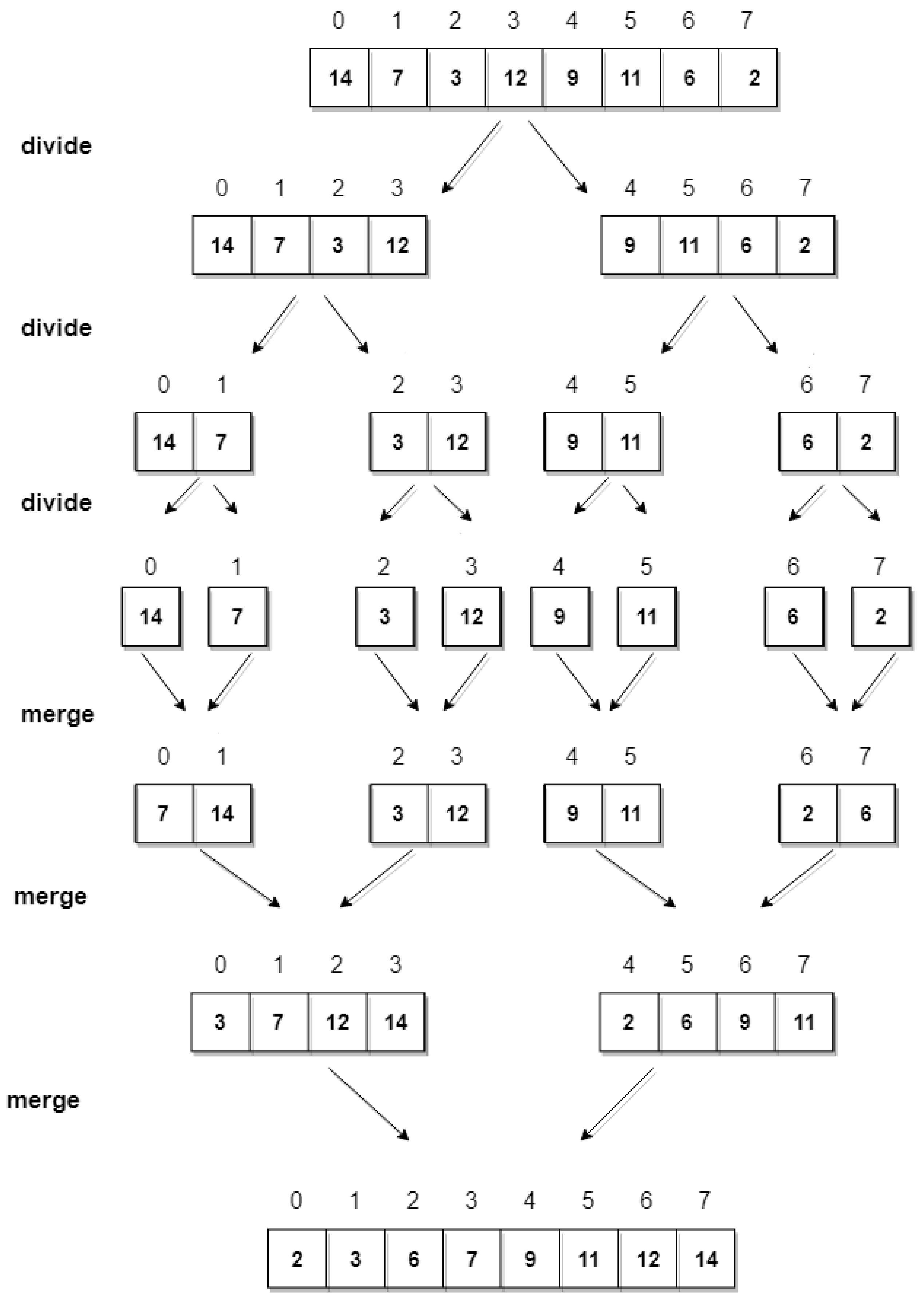
1. Divide : A problem is divided into multiple sub-problems.

2. Conquer : Each sub-problem is solved individually.

3 : Combine : sub-problems are combined to form the final solution.



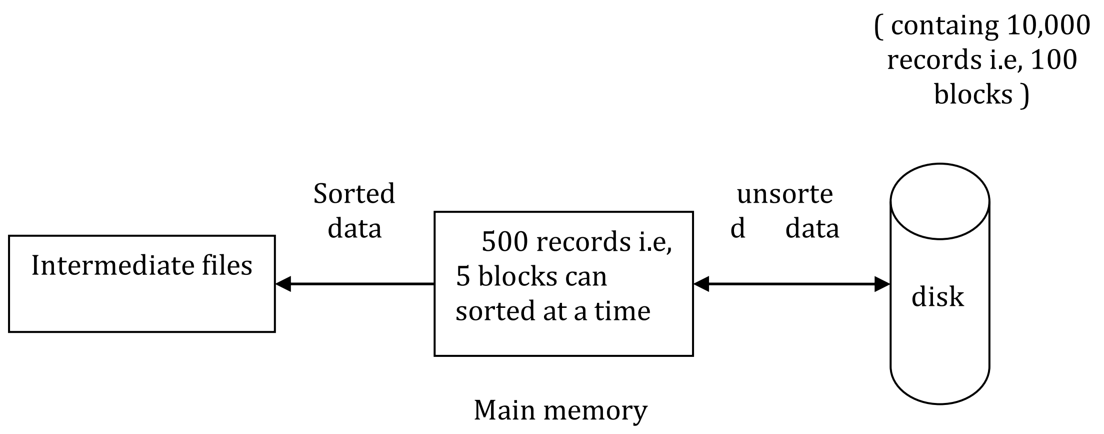
Example : Consider an array with values 14, 7, 3, 12, 9, 11, 6, 2



External Sorting : All the sorting algorithm which are applied on smaller data stored in the main memory. Because main memory can accommodate the data. Some applications need to sort the larger amount of data but it cannot be accommodated into a memory. In such case, we need to use the disk memory or secondary memory to perform sorting.

The data stored on secondary memory is part by part load into main memory and then sorting can be done over there. The sorted data is further stored in the intermediate files. These intermediate files can be done over there. The sorted data is further intermediate files can be merged repeatedly to get sorted data. The data is sorted in the external devices are like disk, tape etc. This is called external sorting.

Example :


(containg 10,000 records i.e, 100 blocks)
Sorted data
unsorted data
Intermediate files
500 records i.e, 5 blocks can sorted at a time
disk
Main memory

Consider that there are 10,000 records those has to be sorted . suppose main memory has capacity to store 500 records in blocks with each block size of 100 records. The sorted 5 blocks (I,e, 500 records) are then sorted in intermediate file. This process will be repeated 20 times to get all records are sorted.

2 - Way Merge Sort

The basic external sorting algorithm uses the *merge* routine from merge sort. Suppose we have four tapes, *Ta*1, *Ta*2, *Tb*1, *Tb*2, which are two input and two output tapes. Depending on the point in the algorithm, the *a* and *b* tapes are either input tapes or output tapes.

Procedure:

Step 1: The data is initially on Ta1.

Step 2: To read *m* records at a time from the input tape, sort the records internally, and then write the Sorted records alternately to *Tb*1 and *Tb*2. We will call each set of sorted records a *run*.

Step 3: Merging of runs

Step 4: Repeat the process until only one RUN is obtained. This would be the sorted file.

Example : 81, 94, 11, 96, 12, 35, 17, 99, 28, 58, 41, 75, 15

N = 13, M = 3 , K = 2.

Solution : K = 2, It requires four tapes, *Ta*1, *Ta*2, *Tb*1, *Tb*2, which are two input and two output tapes. Depending on the point in the algorithm, the *a* and *b* tapes are either input tapes or output tapes.

Step 1 : The data is initially on Ta1.

Ta1	**81 94 11 96 12 35 17 99 28 58 41 75 15**
Ta2	
Tb1	
Tb2	

Step 2 : To read 3 records at a time from the input tape, sort the records internally, and then write the Sorted records alternately to *Tb*1 and *Tb*2.

If M=3 then after the runs are constructed. The tapes will contain the data indicated in the following

Ta1			
Ta2			
Tb1	**11 81 94**	**17 28 99**	**15**
Tb2	**12 35 96**	**41 58 75**	

Step 3 : Merge first Run of Tb1 and the first Run of Tb2, Sort and store the result on Ta1.

Merge second Run of Tb1 and the second Run of Tb2, Sort and store the result on Ta2.

Third run of Tb1, No merging is required.

Ta1	**11 12 35 81 94 96**	**15**
Ta2	**17 28 41 58 75 99**	
Tb1		
Tb2		

Step 4 : Merge first Run of Ta1 and the first Run of Ta2, Sort and store the result on Tb1.

Second run of Ta1, No merging is required.

Ta1												
Ta2												
Tb1	**11**	**12**	**17**	**28**	**35**	**41**	**58**	**81**	**75**	**94**	**96**	**99**
Tb2	**15**											

Step 5 : Merge first Run of Tb1 and the first Run of Tb2, Sort and store the result on Ta1.

Ta1	**11**	**12**	**15**	**17**	**28**	**35**	**41**	**58**	**81**	**75**	**94**	**96**	**99**
Ta2													
Tb1													
Tb2													

The final Sorted Values are : 11 12 15 17 28 35 41 58 81 75 94 96 99

Multiway Merge (or) K-way Merge

If we have extra tapes, then we can expect to reduce the number of passes required to sort our input. We do this by extending the basic (two-way) merge to a *k*-way merge.

Example : 81, 94, 11, 96, 12, 35, 17, 99, 28, 58, 41, 75, 15

N = 13, M = 3 , K = 3.

Solution : K = 3, It requires six tapes, *Ta*1, *Ta*2, Ta3, *Tb*1, *Tb*2, Tb3 which are three input and three output tapes. Depending on the point in the algorithm, the *a* and *b* tapes are either input tapes or output tapes.

Step 1 : The data is initially on Ta1.

Ta1	**81 94 11 96 12 35 17 99 28 58 41 75 15**
Ta2	
Ta3	
Tb1	
Tb2	
Tb3	

Step 2 : To read 3 records at a time from the input tape, sort the records internally, and then write the Sorted records alternately to *Tb*1 , Tb2 and *Tb*3.

If M=3 then after the runs are constructed. The tapes will contain the data indicated in the following

Ta1		
Ta2		
Ta3		
Tb1	**11 81 94**	**41 58 75**
Tb2	**12 35 96**	**15**
Tb3	**17 28 99**	

Step 3 : Merge first Run of Tb1,Tb2, Tb3 Sort and store the result on Ta1.

Merge second Run of Tb1 and Tb2, Sort and store the result on Ta2.

Ta1	**11 12 17 28 35 81 94 96 99**
Ta2	**15 41 58 75**
Ta3	
Tb1	
Tb2	
Tb3	

Step 4 : Merge first Run of Ta1 and Tb2 Sort and store the result on Tb1.

Ta1	
Ta2	
Ta3	
Tb1	**11 12 15 17 28 35 41 58 75 81 94 96 99**
Tb2	
Tb3	

The final Sorted Values are : 11 12 15 17 28 35 41 58 81 75 94 96 99

Poly Phase Merge

Many applications restrict the use of multiway merge, as it uses 2k tapes to perform k-way merging. Instead they preper polyphase merging, as it uses only k+1 tapes for n-way merging.

Example : In performing a 2-way merge, 2+1 tapes are required.

Suppose we have three tapes, *T*1, *T*2, and *T*3, and an input file on *T*1 that will produce 34 runs. This method is to split the original 34 runs unevenly. Suppose we put 21 runs on *T*2 and 13 runs on *T*3. We would then merge 13 runs onto *T*1 before *T*3 was empty. At this point, we could rewind *T*1 and *T*3, and merge *T*1, with 13 runs, and *T*2, which has 8 runs, onto *T*3. We could then merge 8 runs until *T*2 was empty, which would leave 5 runs left on *T*1 and 8 runs on *T*3. We could then merge *T*1 and *T*3, and so on.

The following table below shows the number of runs on each tape after each pass.

	Run Construction	After T2 + T3	After T1 + T2	After T1 + T3	After T2 + T3	After T1 + T2	After T1 + T3	After T2 + T3
T1	0	13	5	0	3	1	0	1
T2	21	8	0	5	2	0	1	0
T3	13	0	8	3	0	2	1	0