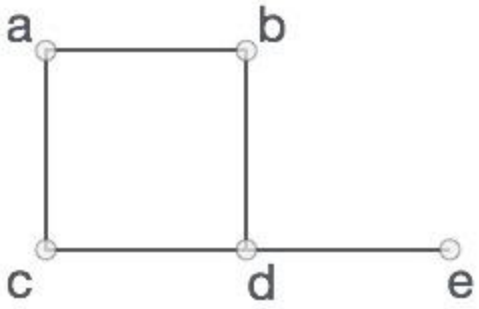
Graph:

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

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



In the above graph,

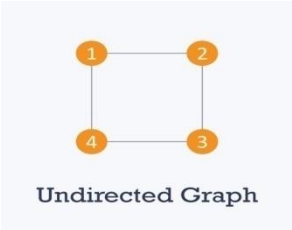
V = {a, b, c, d, e}

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

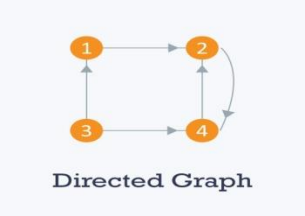
Basics of Graphs:

Types of graphs

- Undirected: An undirected graph is a graph in which all the edges are bi-directional i.e. the edges do not point in any specific direction.
- dges do not point in any specific direction.



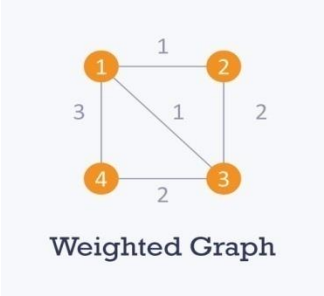
- Directed: A directed graph is a graph in which all the edges are uni-directional i.e. the edges point in a single direction.



Directed Graph

- Weighted: In a weighted graph, each edge is assigned a weight or cost. Consider a graph of 4 nodes as in the diagram below. As you can see each edge has a weight/cost assigned to it. If you want to go from vertex 1 to vertex 3, you can take one of the following 3 paths:
    - 1 -> 2 -> 3
    - 1 -> 3
    - 1 -> 4 -> 3

  Therefore the total cost of each path will be as follows: - The total cost of 1 -> 2 -> 3 will be (1 + 2) i.e. 3 units - The total cost of 1 -> 3 will be 1 unit - The total cost of 1 -> 4 -> 3 will be (3 + 2) i.e. 5 units



Weighted Graph

- Cyclic: A graph is cyclic if the graph comprises a path that starts from a vertex and ends at the same vertex. That path is called a cycle. An acyclic graph is a graph that has no cycle.

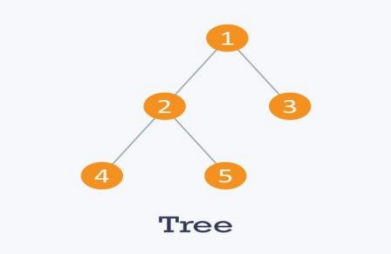
- a**djacency**: A vertex is said to be adjacent to another vertex if there is an edge connecting them. In the above directed graph Vertex 2 is adjacent because there is an edge between them. but Vertex 3 is not adjacent because there is no edge between them.

- **Path**: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.
- **Degree:** The number of edges connected to a node is known as degree of a node. The degree of node 1 in the above undirected graph is 2, degree of node 2 is 2, the degree of node 3 is 2. And the degree of node 4 are 2.

A **tree** is an undirected graph in which any two vertices are connected by only one path. A tree is an acyclic graph and has N - 1 edges where N is the number of vertices. Each node in a graph may have one or multiple parent nodes. However, in a tree, each node (except the root node) comprises exactly one parent node.

Note: A root node has no parent.

A tree cannot contain any cycles or self loops, however, the same does not apply to graphs.



Graph representations

You can represent a graph in many ways. The two most common ways of representing a graph is as follows:

- Adjacency matrix
- Adjacency list

Adjacency matrix

An adjacency matrix is a **VxV** binary matrix **A**. Element is 1 if there is an edge from vertex i to vertex j else is 0.

Note: A binary matrix is a matrix in which the cells can have only one of two possible values - either a 0 or 1.

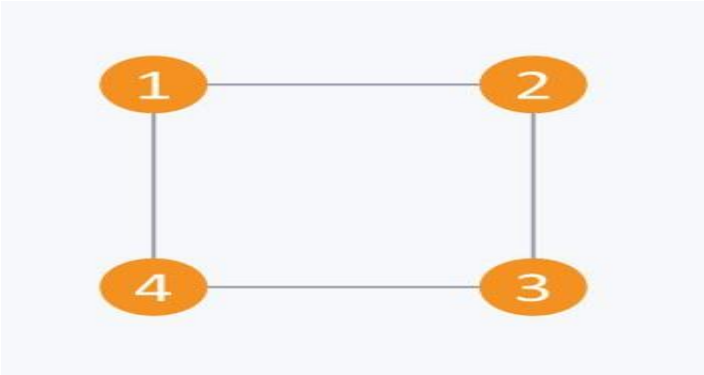
The adjacency matrix can also be modified for the weighted graph in which instead of storing 0 or 1 in , the weight or cost of the edge will be stored.

In an undirected graph, if = 1, then = 1. In a directed graph, if = 1, then may or may not be 1.

Adjacency matrix provides **constant time access (O(1))** to determine if there is an edge between two nodes. Space complexity of the adjacency matrix is **O()**.

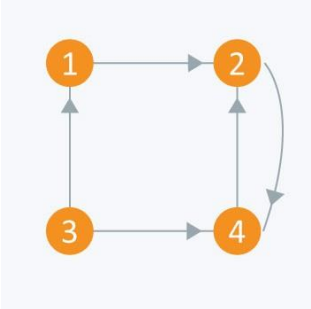
The adjacency matrix of the following graph is:
i/j : **1 2 3 4**

1 : 0 1 0 1
2 : 1 0 1 0
3 : 0 1 0 1
4 : 1 0 1 0



The adjacency matrix of the following graph is:

i/j: **1 2 3 4**
1 : 0 1 0 0
2 : 0 0 0 1
3 : 1 0 0 1
4 : 0 1 0 0



Adjacency list

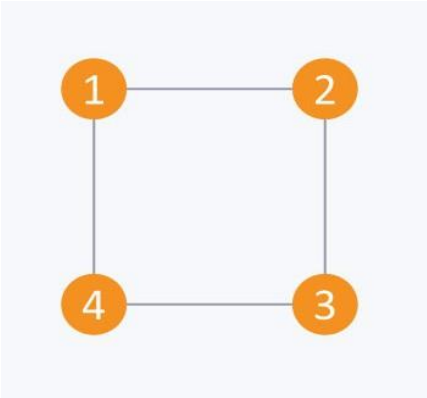
The other way to represent a graph is by using an adjacency list. An adjacency list is an array A of separate lists. Each element of the array $\mathbf{A_i}$ is a list, which contains all the vertices that are adjacent to vertex i.

For a weighted graph, the weight or cost of the edge is stored along with the vertex in the list using pairs. In an undirected graph, if vertex j is in list

then vertex i will be in list The space complexity of adjacency list is **O(V + E)** because in an adjacency list information is stored only for those edges that actually exist in the graph. In a lot of cases, where a matrix is sparse using an adjacency matrix may not be very useful. This is

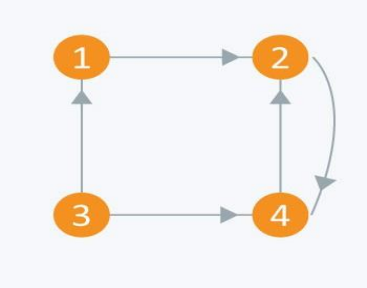
because using an adjacency matrix will take up a lot of space where most of the elements will be 0, anyway. In such cases, using an adjacency list is better.

Note: A sparse matrix is a matrix in which most of the elements are zero, whereas a dense matrix is a matrix in which most of the elements are non-zero.



Consider the same undirected graph from an adjacency matrix. The adjacency list of the graph is as follows:

A1 → 2 → 4
A2 → 1 → 3
A3 → 2 → 4
A4 → 1 → 3



Consider the same directed graph from an adjacency matrix. The adjacency list of the graph is as follows:

A1 → 2
A2 → 4
A3 → 1 → 4
A4 → 2

Graph Traversal Techniques

The process of visiting and exploring a graph for processing is called graph traversal. To be more specific it is all about visiting and exploring each vertex and edge in a graph such that all the vertices are explored exactly once.

During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

There are several graph traversal techniques such as

- Breadth-First Search
- Depth First Search and so on.

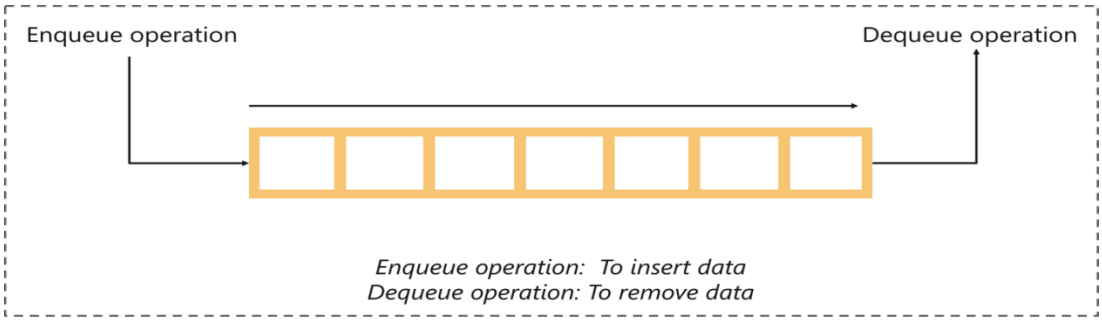
Breadth First Search (BFS)

There are many ways to traverse graphs. BFS is the most commonly used approach.

Breadth-First Search algorithm is a graph traversing technique, where we can select a random initial node (source or root node) and start traversing the graph layer-wise in such a way that all the nodes and their respective children nodes are visited and explored.

Before we get started, you must be familiar with the main data structure involved in the Breadth-First Search algorithm.

A queue is an abstract data structure that follows the First-In-First-Out methodology (data inserted first will be accessed first). It is open on both ends, where one end is always used to insert data (enqueue) and the other is used to remove data (dequeue).



The steps involved in traversing a graph by using Breadth-First Search:

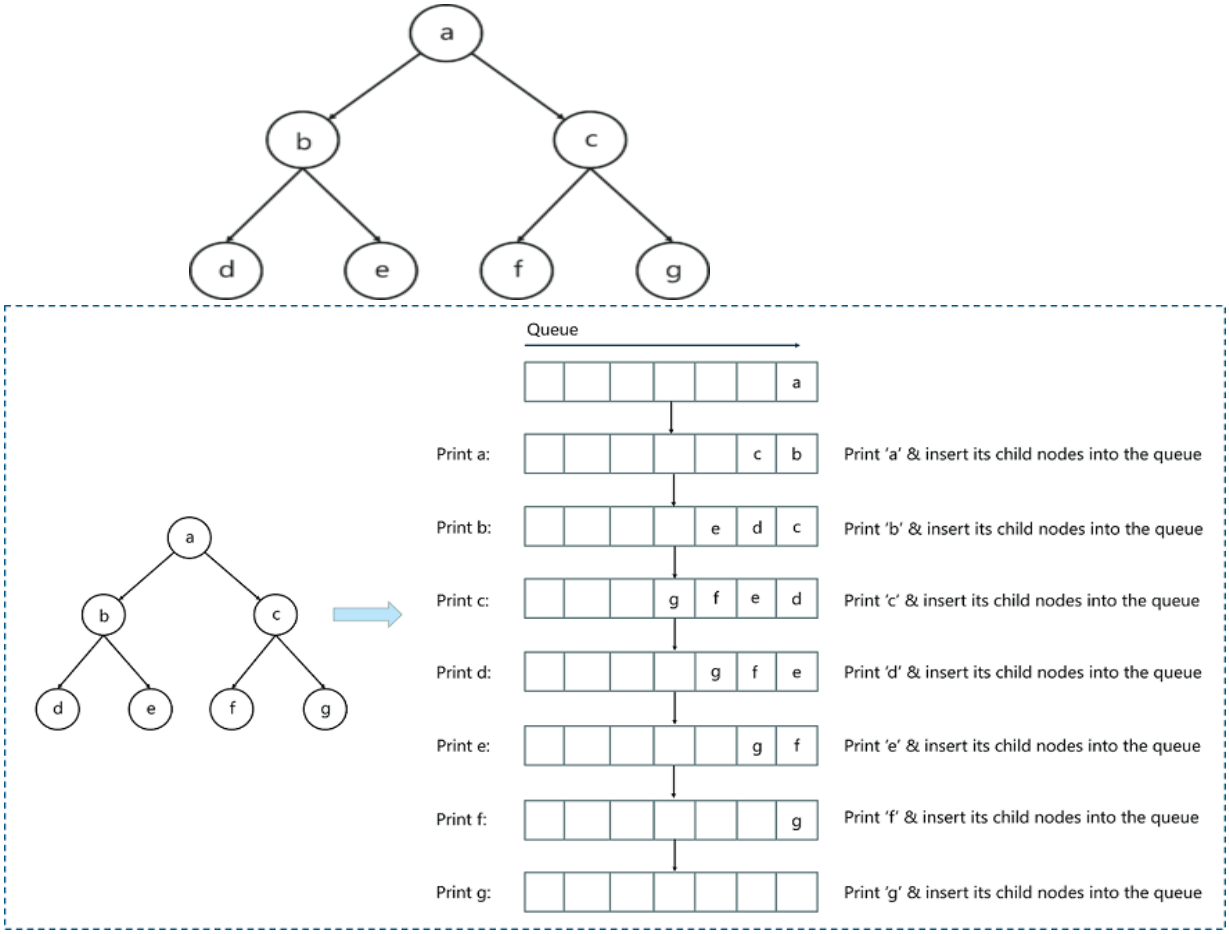
Step 1: Take an Empty Queue.

Step 2: Select a starting node (visiting a node) and insert it into the Queue.

Step 3: Provided that the Queue is not empty, extract the node from the Queue and insert its child nodes (adjacency nodes) into the Queue.

Step 4: Print the extracted node.

Ex:



Explanation:

1. Assign ‘a’ as the root node and insert it into the Queue.
2. Extract node ‘a’ from the queue and insert the child nodes of ‘a’, i.e., ‘b’ and ‘c’.
3. Print node ‘a’.
4. The queue is not empty and has node ‘b’ and ‘c’. Since ‘b’ is the first node in the queue, let’s extract it and insert the child nodes of ‘b’, i.e., node ‘d’ and ‘e’.

5. Repeat these steps until the queue gets empty. Note that the nodes that are already visited should not be added to the queue again.

Applications Of Breadth-First Search Algorithm

Breadth-first Search is a simple graph traversal method that has a surprising range of applications.

GPS Navigation systems: Breadth-First Search is one of the best algorithms used to find neighboring locations by using the GPS system.

Broadcasting:This algorithm is used to communicate broadcasted packets across all the nodes in a network.

Peer to Peer Networking: Breadth-First Search can be used as a traversal method to find all the neighboring nodes in a Peer to Peer Network. For example, BitTorrent uses Breadth-First Search for peer to peer communication.

etc..

Depth First Search (DFS) Algorithm

Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

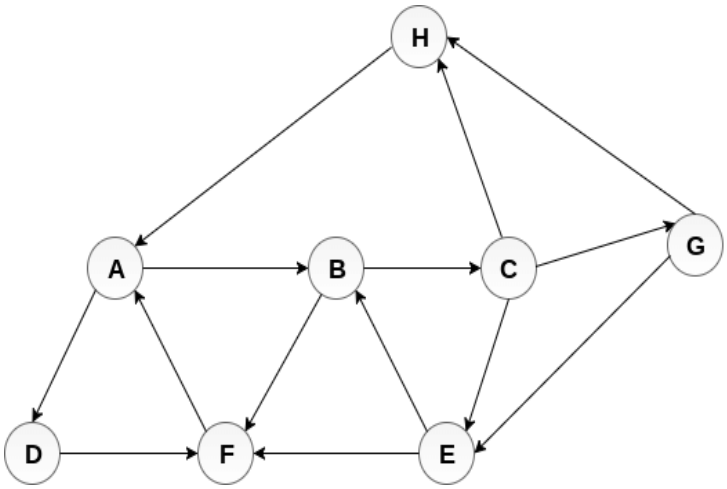
The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.

Algorithm

- **Step 1:** SET STATUS = 1 (ready state) for each node in G
- **Step 2:** Push the starting node A on the stack and set its STATUS = 2 (waiting state)
- **Step 3:** Repeat Steps 4 and 5 until STACK is empty
- **Step 4:** Pop the top node N. Process it and set its STATUS = 3 (processed state)
- **Step 5:** Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their
  STATUS = 2 (waiting state)
  [END OF LOOP]
- **Step 6:** EXIT

Example :

Consider the graph G along with its adjacency list, given in the figure below. Calculate the order to print all the nodes of the graph starting from node H, by using depth first search (DFS) algorithm.



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Solution :

Push H onto the stack

1. STACK : H

POP the top element of the stack i.e. H, print it and push all the neighbours of H onto the stack that are is ready state.

1. Print H
2. STACK : A

Pop the top element of the stack i.e. A, print it and push all the neighbours of A onto the stack that are in ready state.

1. Print A
2. Stack : B, D

Pop the top element of the stack i.e. D, print it and push all the neighbours of D onto the stack that are in ready state.

1. Print D
2. Stack : B, F

Pop the top element of the stack i.e. F, print it and push all the neighbours of F onto the stack that are in ready state.

1. Print F
2. Stack : B

Pop the top of the stack i.e. B and push all the neighbours

1. Print B
2. Stack : C

Pop the top of the stack i.e. C and push all the neighbours.

1. Print C
2. Stack : E, G

Pop the top of the stack i.e. G and push all its neighbours.

1. Print G
2. Stack : E

Pop the top of the stack i.e. E and push all its neighbours.

1. Print E
2. Stack :

Hence, the stack now becomes empty and all the nodes of the graph have been traversed.

The printing sequence of the graph will be :

1. H → A → D → F → B → C → G → E

Applications of graphs

1)In **Computer science** graphs are used to represent the flow of computation.

2)Google maps uses graphs for building transportation systems,

3)In **Facebook**, users are considered to be the vertices and if they are friends then there is an edge running between them

4)In **World Wide Web**, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u.

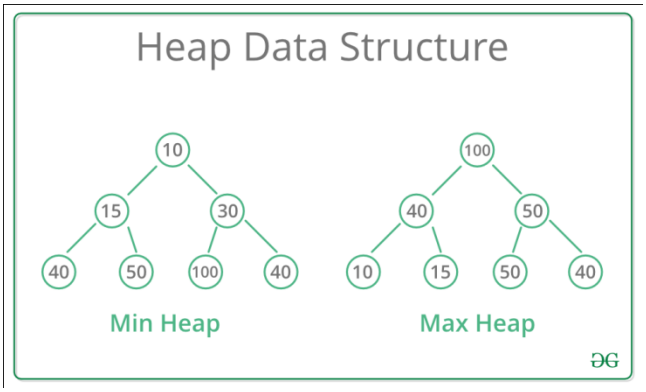
5)In **Operating System**, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated

process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occurs

A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Generally, Heaps can be of two types:

1. **Max-Heap**: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

2. **Min-Heap**: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.


Heap Data Structure
10
15
30
40
50
100
40
Min Heap
100
40
50
10
15
50
40
Max Heap

Heap Sort Algorithm

Heap sort is one of the sorting algorithms used to arrange a list of elements in order. Heapsort algorithm uses one of the tree concepts called **Heap Tree**. In this sorting algorithm, we use **Max Heap** to arrange list of elements in Descending order and **Min Heap** to arrange list elements in Ascending order.

Step by Step Process

The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

- Step 1 - Construct a **Binary Tree** with given list of Elements.
- Step 2 - Transform the Binary Tree into **Min Heap.**
- Step 3 - Delete the root element from Min Heap using **Heapify** method.
- Step 4 - Put the deleted element into the Sorted list.

Complexity of the Heap Sort Algorithm

To sort an unsorted list with **'n'** number of elements, following are the complexities...

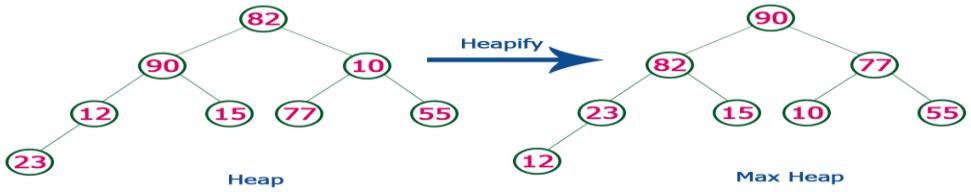
Worst Case : O(n log n)
Best Case : O(n log n)
Average Case : O(n log n)

Ex 1:

Consider the following list of unsorted numbers which are to be sort using Heap Sort

82, 90, 10, 12, 15, 77, 55, 23

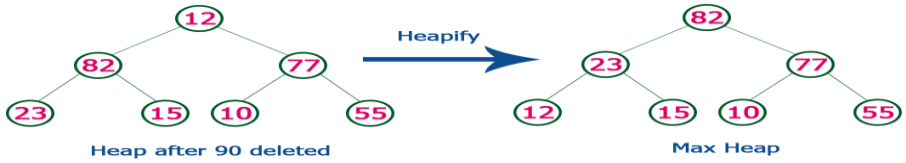
Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap



list of numbers after heap converted to Max Heap

90, 82, 77, 23, 15, 10, 55, 12

Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

12, 82, 77, 23, 15, 10, 55, 90

Step 3 - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 82 with 55.

12, 55, 77, 23, 15, 10, 82, 90

Step 4 - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 77 with 10.

12, 55, 10, 23, 15, 77, 82, 90

Step 5 - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 55 with 15.

12, 15, 10, 23, 55, 77, 82, 90

Step 6 - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 23 with 12.

12, 15, 10, 23, 55, 77, 82, 90

Step 7 - Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.

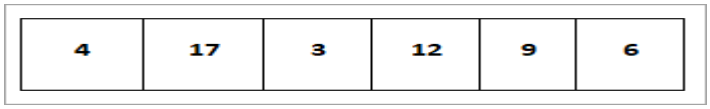
list of numbers after Deleting 15, 12 & 10 from the Max Heap.

10, 12, 15, 23, 55, 77, 82, 90

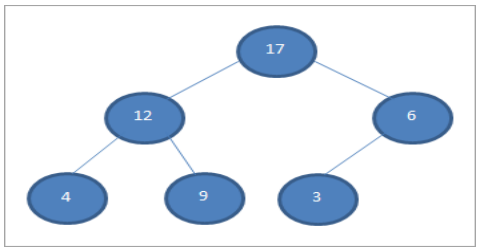
Whenever Max Heap becomes Empty, the list get sorted in Ascending order

Ex: 2

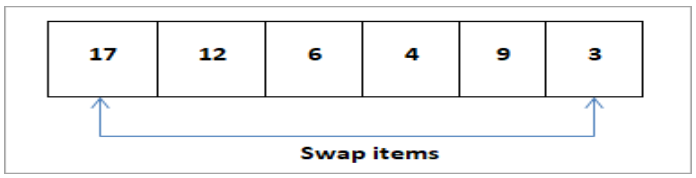
Sort the following array elements using the heap sort technique.



First **construct a max-heap as shown below for the array to be sorted.**

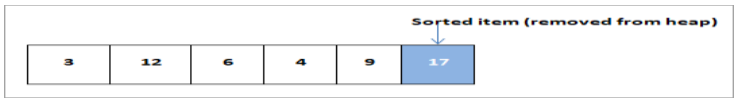


Once the heap is constructed, we represent it in an Array form as shown below.



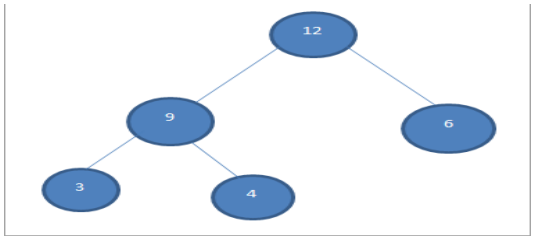
Now we compare the 1st node (root) with the last node and then swap them. Thus, as shown above, we swap 17 and 3 so that 17 is at the last position and 3 is in the first position.

Now we remove the node 17 from the heap and put it in the sorted array as shown in the shaded portion below.



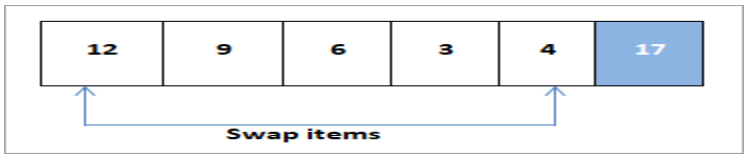
Now we again construct a heap for the array elements. This time the heap size is reduced by 1 as we have deleted one element (17) from the heap.

The heap of the remaining elements is shown below.

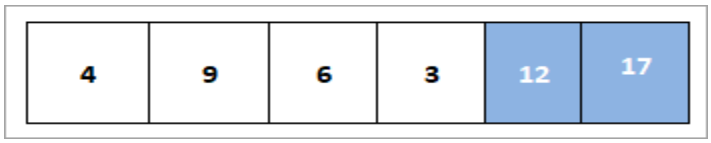


In the next step, we will repeat the same steps.

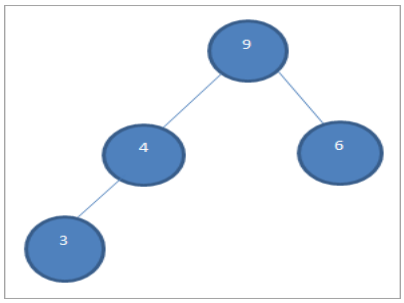
We compare and swap the root element and last element in the heap.



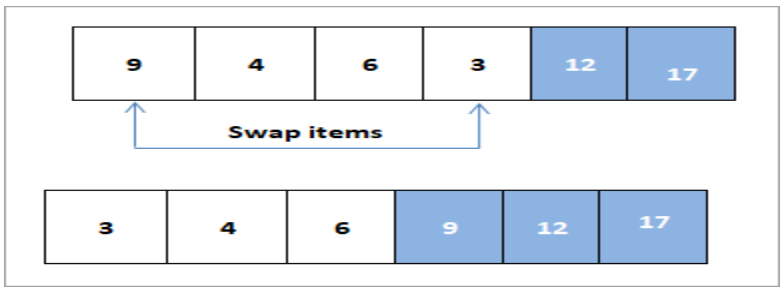
After swapping, we delete the element 12 from the heap and shift it to the sorted array.



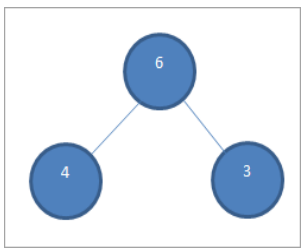
Once again we construct a max heap for the remaining elements as shown below.



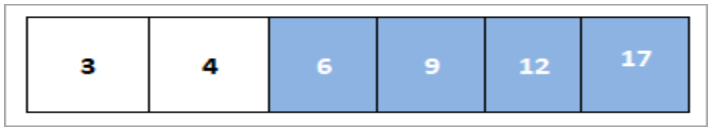
Now we swap the root and the last element i.e. 9 and 3. After swapping, element 9 is deleted from the heap and put in a sorted array.



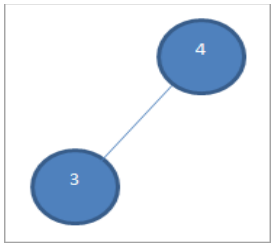
At this point, we have only three elements in the heap as shown below.



We swap 6 and 3 and delete the element 6 from the heap and add it to the sorted array.

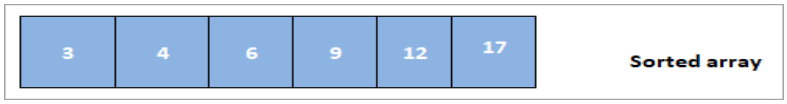


Now we construct a heap of the remaining elements and then swap both with each other.



After swapping 4 and 3, we delete element 4 from the heap and add it to the sorted array. Now we have only one node remaining in the heap as shown below**.**

So now with only one node remaining, we delete it from the heap and add it to the sorted array.



Thus the above shown is the sorted array t

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

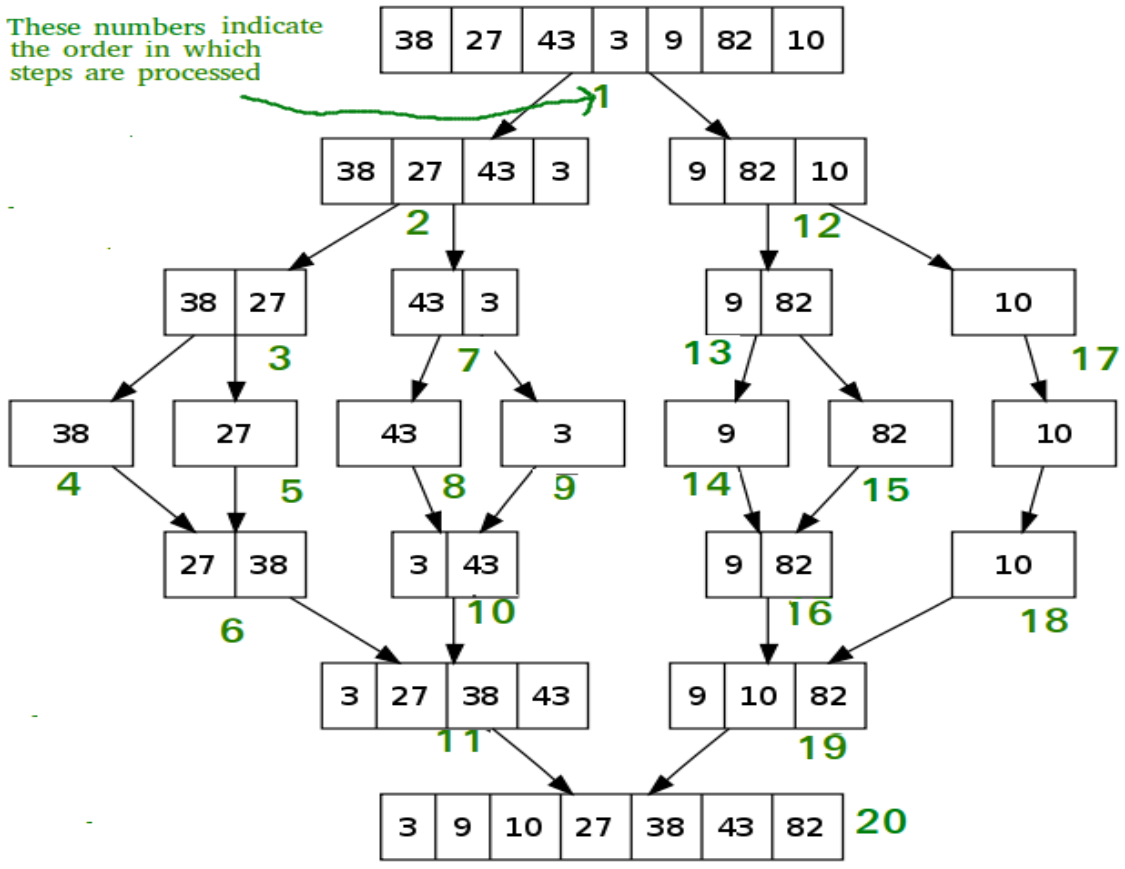
Algorithm:

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

While comparing two sublists for merging, the first element of both lists is taken into consideration. While sorting in ascending order, the element that is of a lesser value becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the new combined sublist comprises all the elements of both the sublists.

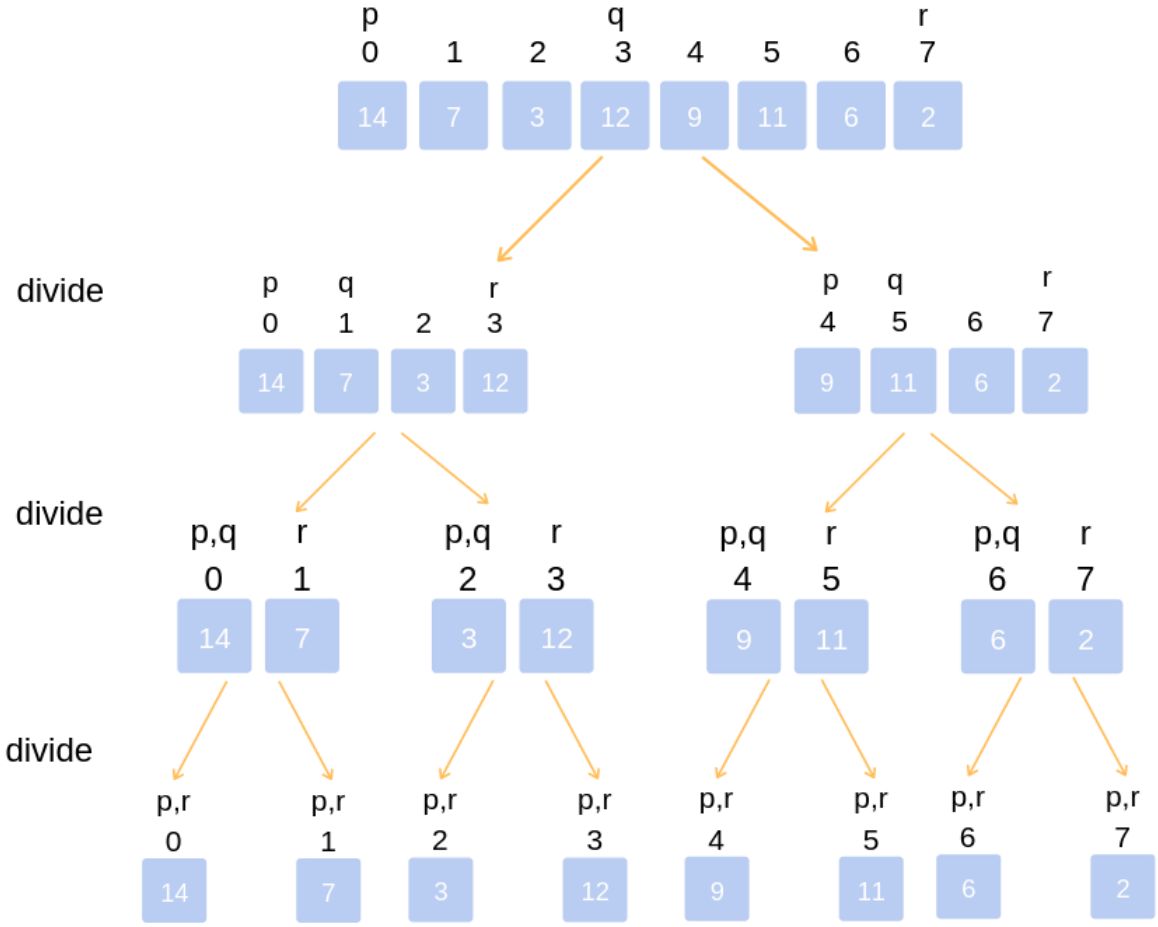
Let's consider the following Example:

Sort the following elements using merge sort 38, 27, 43, 3, 9, 82, 10

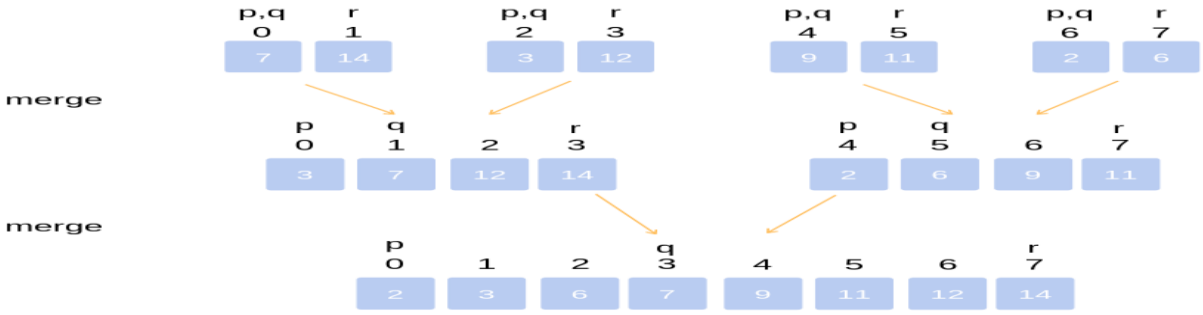


Ex: 2

Sort the following list of elements 14,7,3,12,9,11,6,2 using Merge Sort



First the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



Merge Sort Psuedocode

```
void mergesort(int a[],int i,int j)
{
int mid;
if(i<j)
{
mid=(i+j)/2;
mergesort(a,i,mid);
mergesort(a,mid+1,j);
merge(a,i,mid,mid+1,j);
}
}
void merge(int a[],int i1,int j1,int i2,int j2)
{
int temp[50]; //array used for merging
int i,j,k;
i=i1;
j=i2;
k=0;
while(i<=j1 && j<=j2) //while elements in both lists
{
if(a[i]<a[j])
temp[k++]=a[i++];
else
temp[k++]=a[j++];
}
while(i<=j1)
temp[k++]=a[i++];

while(j<=j2)
temp[k++]=a[j++];


//Transfer elements from temp[] back to a[]
for(i=i1,j=0;i<=j2;i++,j++)
a[i]=temp[j];

}
```

External sorting

External sorting is a technique in which the data is stored on the secondary memory, in which part by part data is loaded into the main memory and then sorting can be done over there. Then this sorted data will be stored in the intermediate files. Finally, these files will be merged to get a sorted data. Thus by using the external sorting technique, a huge amount of data can be sorted easily. In case of external sorting, all the data cannot be accommodated on the single memory, in this case, some amount of memory needs to be kept on a memory such as hard disk, compact disk and so on.

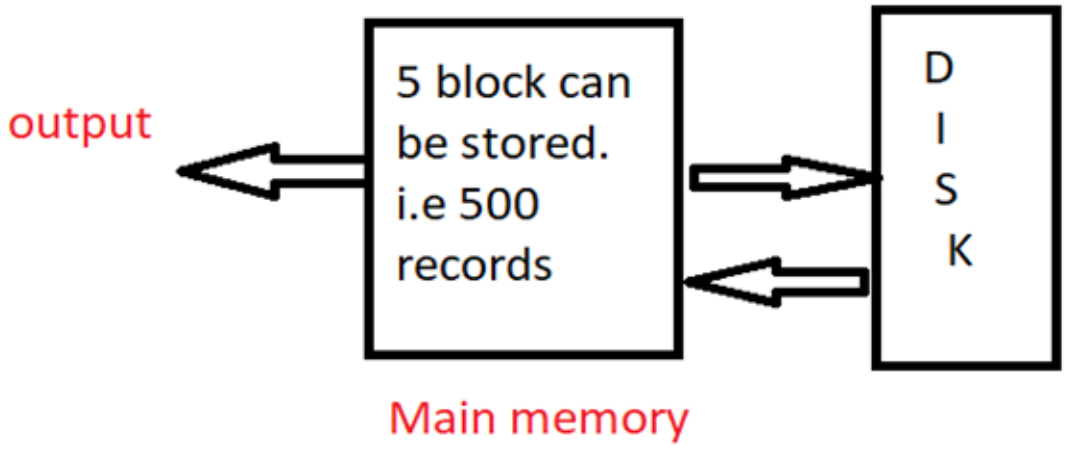
The requirement of **external sorting** is there, where the data we have to store in the main memory does not fit into it. Basically, it consists of two phases that are:

1. **Sorting phase:** This is a phase in which a large amount of data is sorted in an intermediate file.
2. **Merge phase:** In this phase, the sorted files are combined into a single larger file.

one of the best examples of external sorting is external merge sort.

External merge sort

3. The **external merge sort** is a technique in which the data is stored in intermediate files and then each intermediate files are sorted independently and then combined or merged to get a sorted data.
4. **For example:** Let us consider there are 10,000 records which have to be sorted. For this, we need to apply the external merge sort method. Suppose the main memory has a capacity to store 500 records in a block, with having each block size of 100 records.



In this example, we can see 5 blocks will be sorted in intermediate files. This process will be repeated 20 times to get all the records. Then by this, we start merging a pair of intermediate files in the main memory to get a sorted output.

Two-Way Merge Sort

Two-way merge sort is a technique which works in two stages which are as follows here:

Stage 1: Firstly break the records into the blocks and then sort the individual record with the help of two input tapes.

Stage 2: In this merge the sorted blocks and then create a single sorted file with the help of two output tapes.

By this, it can be said that **two-way merge sort** uses the two input tapes and two output tapes for sorting the data.

Algorithm for Two-Way Merge Sort:

Step 1) Divide the elements into the blocks of size M. Sort each block and then write on disk.

Step 2) Merge two runs

1. Read first value on every two runs.
2. Then compare it and sort it.
3. Write the sorted record on the output tape.

Step 3) Repeat the step 2 and get longer and longer runs on alternates tapes. Finally, at last, we will get a single sorted list.