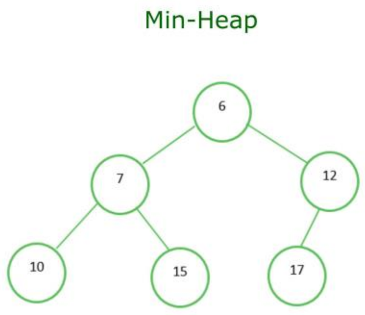
UNIT-II

Max Heap and Min Heap:

A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Since a heap is a complete binary tree, a heap with **N** nodes has **log N** height. It is useful to remove the highest or lowest priority element. It is typically represented as an array. There are two types of Heaps in the data structure.

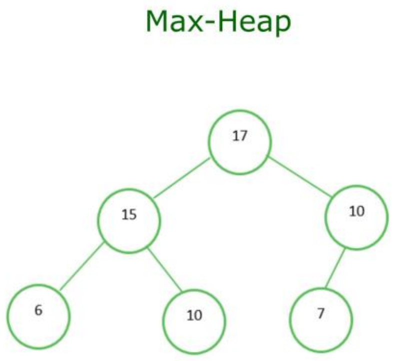
Min-Heap

In a Min-Heap the key present at the root node must be less than or equal among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree. In a Min-Heap the minimum key element present at the root. Below is the Binary Tree that satisfies all the property of Min Heap.



Max Heap

In a Max-Heap the key present at the root node must be greater than or equal among the keys present at all of its children. The same property must be recursively **true** for all sub-trees in that Binary Tree. In a Max-Heap the maximum key element present at the root. Below is the Binary Tree that satisfies all the property of Max Heap.



Difference between Min Heap and Max Heap

	Min Heap	**Max Heap**
1.	In a Min-Heap the key present at the root node must be less than or equal to among the keys present at all of its children.	In a Max-Heap the key present at the root node must be greater than or equal to among the keys present at all of its children.

	Min Heap	**Max Heap**
2.	In a Min-Heap the minimum key element present at the root.	In a Max-Heap the maximum key element present at the root.
3.	A Min-Heap uses the ascending priority.	A Max-Heap uses the descending priority.
4.	In the construction of a Min-Heap, the smallest element has priority.	In the construction of a Max-Heap, the largest element has priority.
5.	In a Min-Heap, the smallest element is the first to be popped from the heap.	In a Max-Heap, the largest element is the first to be popped from the heap.

Applications of Heaps:

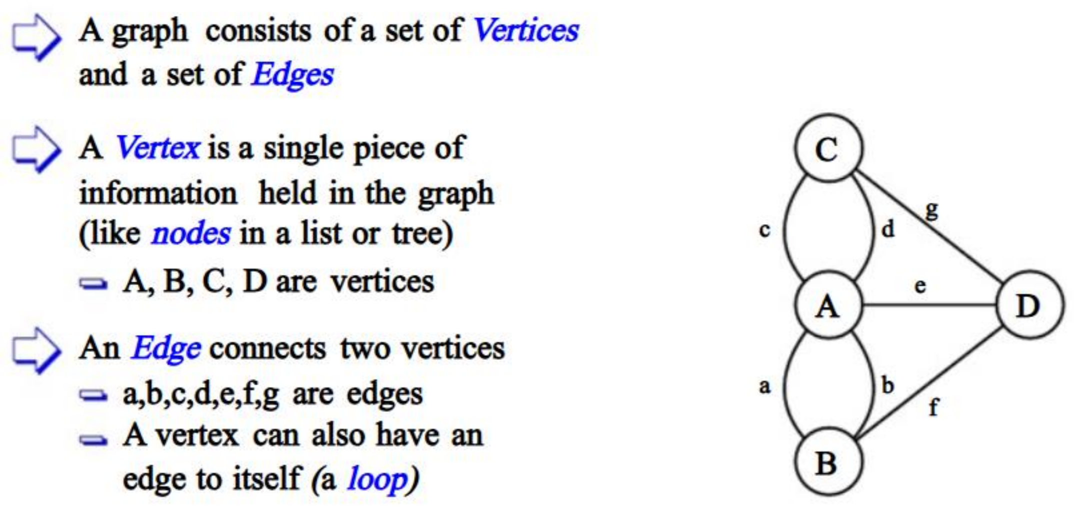
1. Heap Sort: Heap Sort is one of the best sorting algorithms that use Binary Heap to sort an array in **O(N*log N)** time.
2. Priority Queue: A priority queue can be implemented by using a heap because it supports **insert()**, **delete()**, **extractMax()**, **decreaseKey()** operations in **O(log N)** time.
3. Graph Algorithms: The heaps are especially used in Graph Algorithms like Dijkstra's Shortest Path and Prim's Minimum Spanning Tree.

Performance Analysis of Min-Heap and Max-Heap:

- Get Maximum or Minimum Element: O(1)
- Insert Element into Max-Heap or Min-Heap: O(log N)
- Remove Maximum or Minimum Element: O(log N)

4.Graphs:

Graph Data Structure is a collection of **nodes** connected by **edges**. It's used to represent relationships between different entities. **Graph algorithms** are methods used to manipulate and analyze graphs, solving various problems like **finding the shortest path** or **detecting cycles.**



- A graph consists of a set of *Vertices* and a set of *Edges*
- A *Vertex* is a single piece of information held in the graph (like *nodes* in a list or tree)
  - A, B, C, D are vertices
- An *Edge* connects two vertices
  - a,b,c,d,e,f,g are edges
  - A vertex can also have an edge to itself (a *loop*)

Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(**V**) and a set of edges(**E**). The graph is denoted by **G(V, E).**

Graph data structures are a powerful tool for representing and analyzing complex relationships between objects or entities. They are particularly useful in fields such as social network analysis, recommendation systems, and computer networks. In the field of sports data science, graph data structures can be used to analyze and understand the dynamics of team performance and player interactions on the field.

Components of a Graph:

- **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabeled.
- **Edges:** Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabelled.

Basic Operations on Graphs:

Below are the basic operations on the graph:

- Insertion of Nodes/Edges in the graph – Insert a node into the graph.
- Deletion of Nodes/Edges in the graph – Delete a node from the graph.
- Searching on Graphs – Search an entity in the graph.
- Traversal of Graphs – Traversing all the nodes in the graph.

Applications of Graph:

Following are the real-life applications:

- Graph data structures can be used to represent the interactions between players on a team, such as passes, shots, and tackles. Analyzing these interactions can provide insights into team dynamics and areas for improvement.
- Commonly used to represent social networks, such as networks of friends on social media.
- Graphs can be used to represent the topology of computer networks, such as the connections between routers and switches.
- Graphs are used to represent the connections between different places in a transportation network, such as roads and airports.

- Graphs are used in Neural Networks where vertices represent neurons and edges represent the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 10^11 neurons and close to 10^15 synapses.

Representations of Graph

Here are the two most common ways to represent a graph :

1. Adjacency Matrix
2. Adjacency List

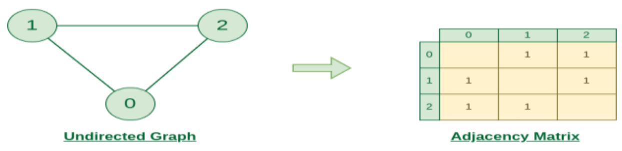
Adjacency Matrix

An adjacency matrix is a way of representing a graph as a matrix of boolean (0's and 1's).
Let's assume there are **n** vertices in the graph So, create a 2D matrix **adjMat[n][n]** having dimension n x n.

- If there is an edge from vertex **i** to **j**, mark **adjMat[i][j]** as **1**.
- If there is no edge from vertex **i** to **j**, mark **adjMat[i][j]** as **0**.

Representation of Undirected Graph to Adjacency Matrix:

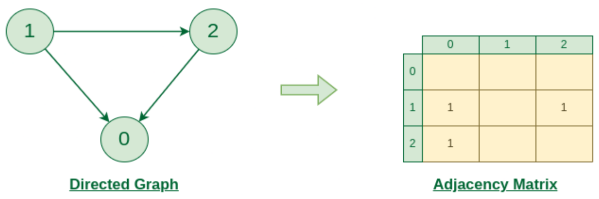
The below figure shows an undirected graph. Initially, the entire Matrix is initialized to **0**. If there is an edge from source to destination, we insert **1** to both cases (**adjMat[destination]** and **adjMat**[**destination])** because we can go either way.



Graph Representation of Undirected graph to Adjacency Matrix

Representation of Directed Graph to Adjacency Matrix:

The below figure shows a directed graph. Initially, the entire Matrix is initialized to **0**. If there is an edge from source to destination, we insert **1** for that particular **adjMat[destination]**.



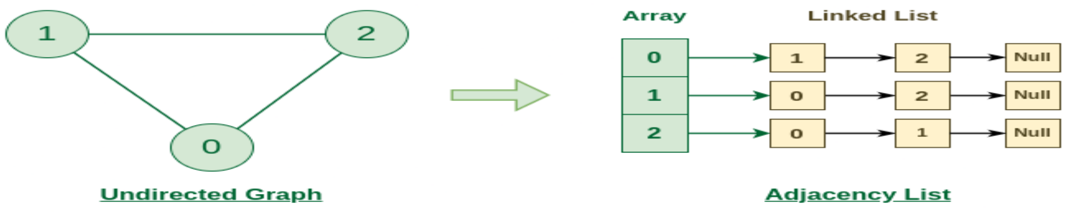
Graph Representation of Directed graph to Adjacency Matrix

Adjacency List:

An array of Lists is used to store edges between two vertices. The size of array is equal to the number of **vertices (i.e, n)**. Each index in this array represents a specific vertex in the graph. The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex **i**.
Let's assume there are **n** vertices in the graph So, create an **array of list** of size **n** as **adjList[n].**

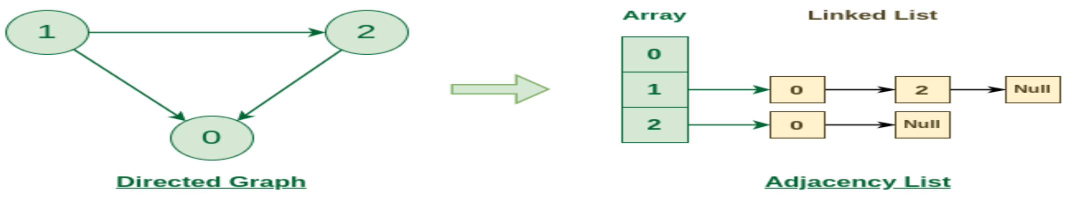
- adjList[0] will have all the nodes which are connected (neighbour) to vertex **0**.
- adjList[1] will have all the nodes which are connected (neighbour) to vertex **1** and so on.

Representation of Undirected Graph to Adjacency list:
The below undirected graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has two neighbours (i.e, 1 and 2). So, insert vertex 1 and 2 at indices 0 of array. Similarly, For vertex 1, it has two neighbour (i.e, 2 and 0) So, insert vertices 2 and 0 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



Graph Representation of Undirected graph to Adjacency List

Representation of Directed Graph to Adjacency list:
The below directed graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has no neighbours. For vertex 1, it has two neighbour (i.e, 0 and 2) So, insert vertices 0 and 2 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



Graph Representation of Directed graph to Adjacency List

Breadth First Search (BFS) for a Graph:
Breadth First Search (BFS) is a graph traversal algorithm that explores all the vertices in a graph at the current depth before moving on to the vertices at the next depth level. It starts at a specified vertex and visits all its neighbors before moving on to the next level of neighbors. **BFS** is commonly used in algorithms for pathfinding, connected components, and shortest path problems in graphs.
Relation between BFS for Graph and BFS for Tree:
Breadth-First Traversal (BFS) for a graph is similar to the Breadth-First Traversal of a tree . The only catch here is, that, unlike **trees** , **graphs** may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

- Visited and
- Not visited.

A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS **uses** a **queue data structure** for traversal.
Breadth First Search (BFS) for a Graph Algorithm:
Let's discuss the algorithm for the BFS:

1. **Initialization:** Enqueue the starting node into a queue and mark it as visited.
2. **Exploration:** While the queue is not empty:

- Dequeue a node from the queue and visit it (e.g., print its value).
- For each unvisited neighbor of the dequeued node:
    - Enqueue the neighbor into the queue.
    - Mark the neighbor as visited.

3. **Termination:** Repeat step 2 until the queue is empty.

This algorithm ensures that all nodes in the graph are visited in a breadth-first manner, starting from the starting node.

How Does the BFS Algorithm Work?

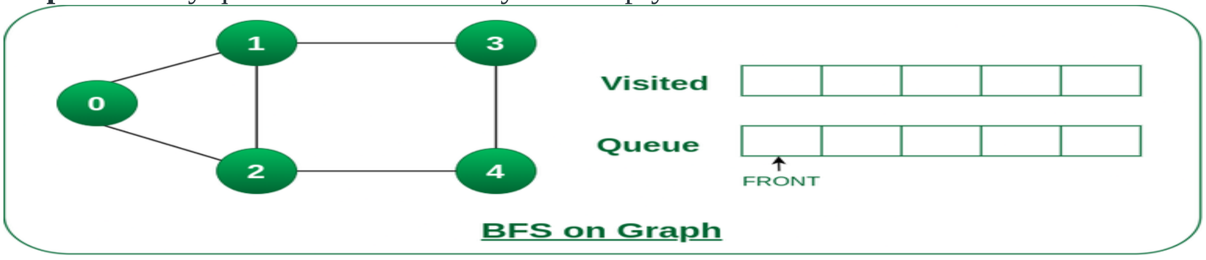
Starting from the root, all the nodes at a particular level are visited first and then the nodes of the next level are traversed till all the nodes are visited.

To do this a queue is used. All the adjacent unvisited nodes of the current level are pushed into the queue and the nodes of the current level are marked visited and popped from the queue.

Illustration:

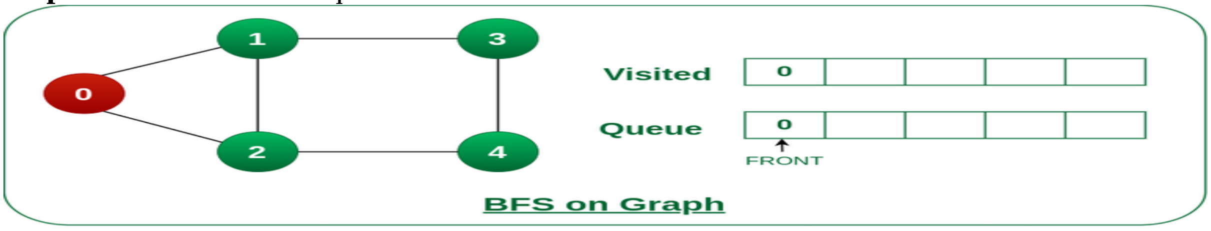
Let us understand the working of the algorithm with the help of the following example.

Step1: Initially queue and visited arrays are empty.



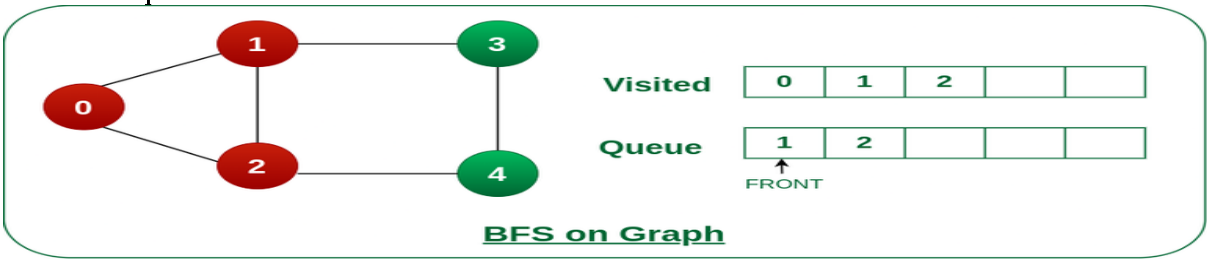
Queue and visited arrays are empty initially.

Step2: Push node 0 into queue and mark it visited.



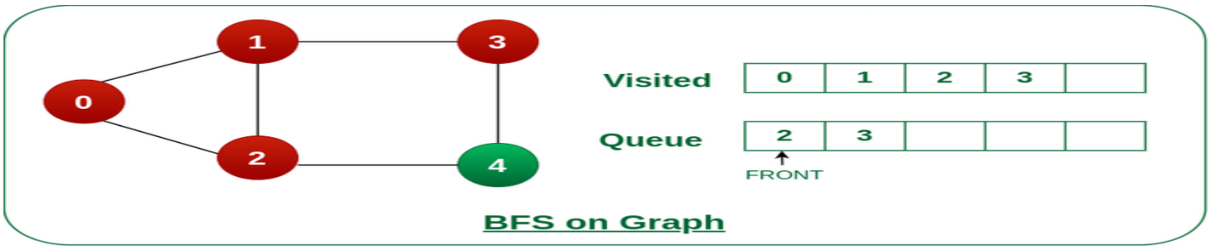
Push node 0 into queue and mark it visited.

Step 3: Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.



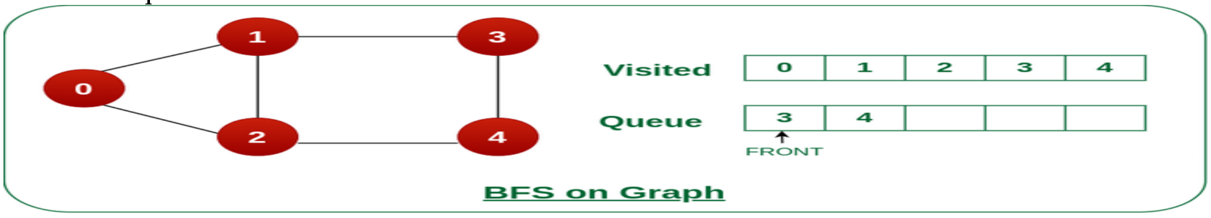
Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.

Step 4: Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.



Remove node 1 from the front of queue and visited the unvisited neighbours and push

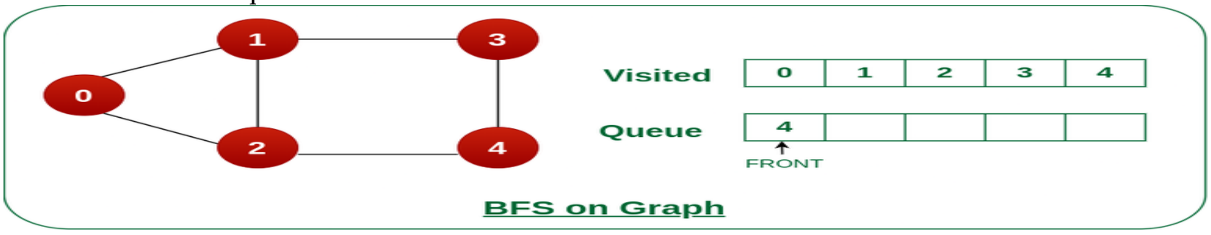
Step 5: Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.



Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.

Step 6: Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

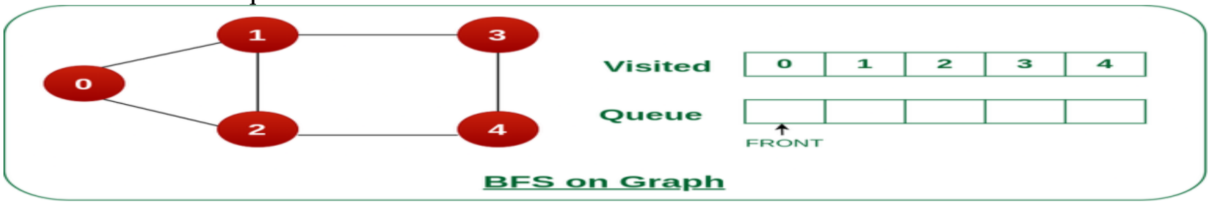
As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.



Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

Steps 7: Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.

As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.

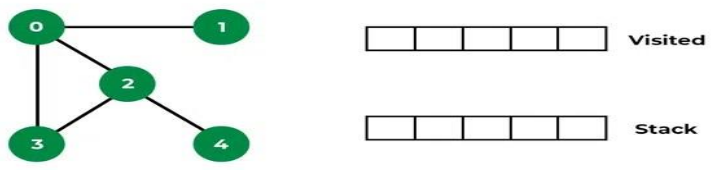


Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.

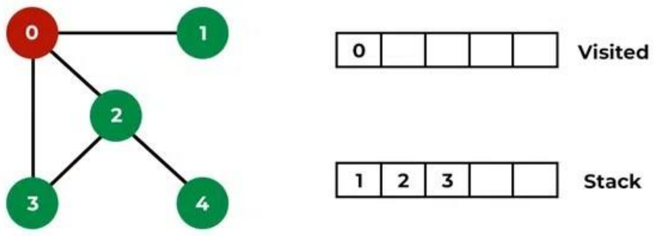
Now, Queue becomes empty, So, terminate these process of iteration.

Depth First Traversal (or DFS) for a graph is similar to Depth First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array. A graph can have more than one DFS traversal.

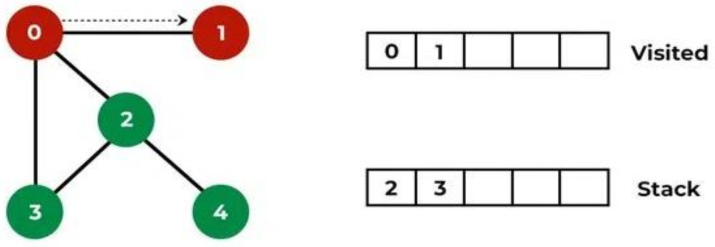
How does DFS work?
Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.
Let us understand the working of **Depth First Search** with the help of the following illustration:
Step1: Initially stack and visited arrays are empty.



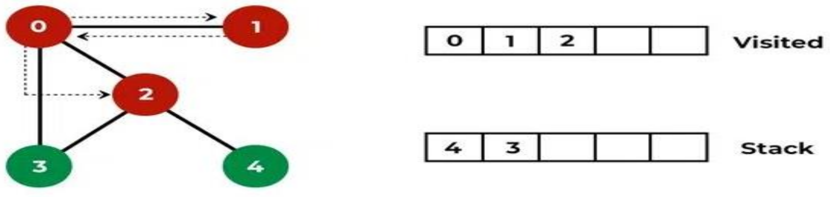
Stack and visited arrays are empty initially.
Step 2: Visit 0 and put its adjacent nodes which are not visited yet into the stack.



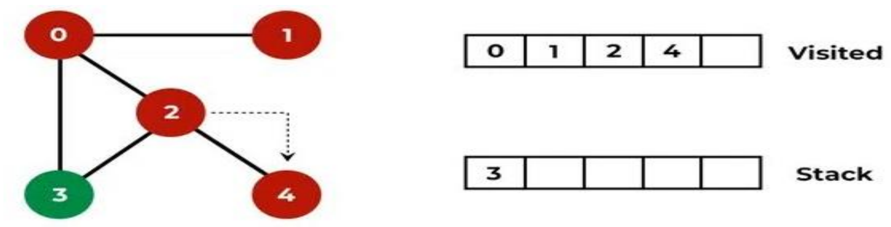
Visit node 0 and put its adjacent nodes (1, 2, 3) into the stack
Step 3: Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



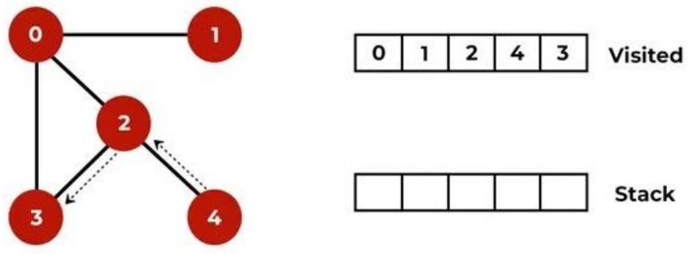
Visit node 1
Step 4: Now, Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited (i.e, 3, 4) in the stack.



Visit node 2 and put its unvisited adjacent nodes (3, 4) into the stack
Step 5: Now, Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.

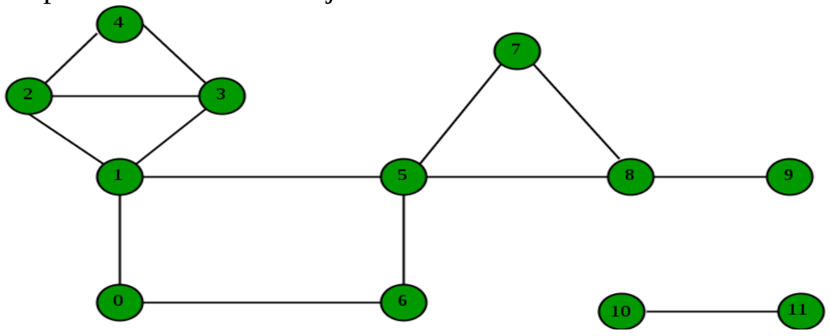


Visit node 4
Step 6: Now, Node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



Visit node 3
Now, Stack becomes empty, which means we have visited all the nodes and our DFS traversal ends.

Connectivity in a Graph:
Finding Bi connected components in a Graph:
we will see how to find biconnected component in a graph using algorithm by John Hopcroft and Robert Tarjan.



In above graph, following are the biconnected components:

- 4–2 3–4 3–1 2–3 1–2
- 8–9
- 8–5 7–8 5–7
- 6–0 5–6 1–5 0–1
- 10–11

The algorithm for finding biconnected components (BCC) in a graph by John Hopcroft and Robert Tarjan is a classical algorithm in graph theory. It efficiently finds all biconnected components of an undirected graph using depth-first search (DFS). Here’s a detailed description of the algorithm:

Algorithm for Finding Biconnected Components by Hopcroft and Tarjan

Definitions

- **Discovery time**: The time when a vertex is first visited during DFS.
- **Low value**: The smallest discovery time reachable from a vertex.

Steps

1. **Initialization**:
    - Start DFS from an arbitrary vertex.
    - Maintain an array disc[] to store discovery times of visited vertices.
    - Maintain an array low[] to store the smallest discovery time reachable from a vertex.
    - Use a stack to store edges for backtracking.
    - Initialize a counter for discovery times.
2. **DFS Traversal**:
    - For each vertex, perform DFS:
        - Assign discovery time and low value to the vertex.
        - For each adjacent vertex:
            - If the adjacent vertex is not visited, recursively perform DFS on it:
                - Update the low value of the current vertex.
                - If the subtree rooted with the adjacent vertex has a connection back to one of the ancestors of the current vertex, then the current vertex is an articulation point.
            - If the adjacent vertex is already visited and is not the parent of the current vertex, update the low value of the current vertex.
        - After visiting all adjacent vertices, if the current vertex is an articulation point, pop all edges from the stack till the current edge.
3. **Output Biconnected Components**:
    - After completing the DFS traversal, the edges left in the stack form a biconnected component.

5.Sorting:

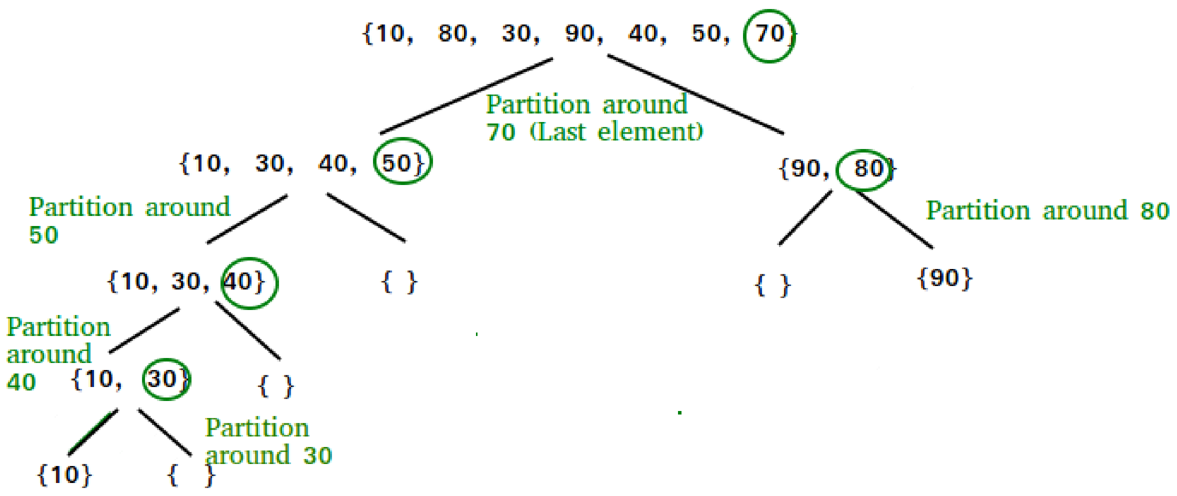
Quick Sort:

QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

How does QuickSort work?

The key process in **quickSort** is a **partition()**. The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.

Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.

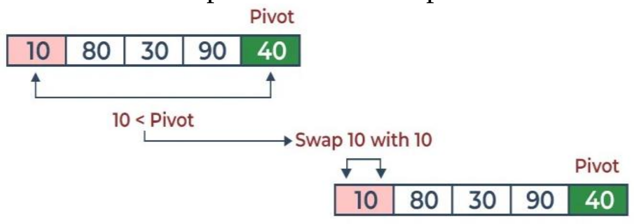


Choice of Pivot:
There are many different choices for picking pivots.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot (implemented below)
- Pick a random element as a pivot.
- Pick the middle as the pivot.

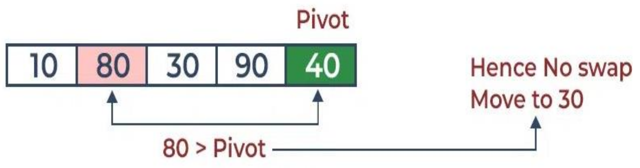
Partition Algorithm:
The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal) elements as **i**. While traversing, if we find a smaller element, we swap the current element with arr[i]. Otherwise, we ignore the current element.
Let us understand the working of partition and the Quick Sort algorithm with the help of the following example:
Consider: arr[] = {10, 80, 30, 90, 40}.

- Compare 10 with the pivot and as it is less than pivot arrange it accrodingly.



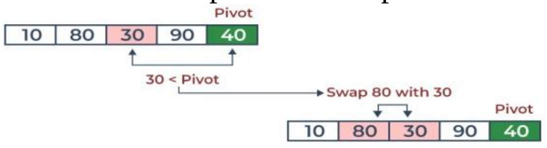
Partition in QuickSort: Compare pivot with 10

- Compare 80 with the pivot. It is greater than pivot.



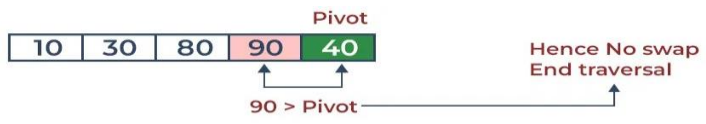
Partition in QuickSort: Compare pivot with 80

- Compare 30 with pivot. It is less than pivot so arrange it accordingly.



Partition in QuickSort: Compare pivot with 30

- Compare 90 with the pivot. It is greater than the pivot.



Partition in QuickSort: Compare pivot with 90

- Arrange the pivot in its correct position.

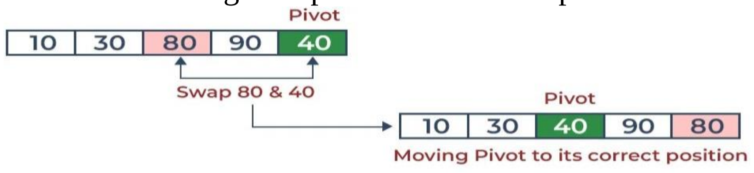
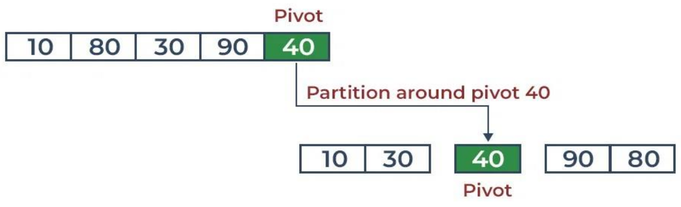


Illustration of Quicksort:

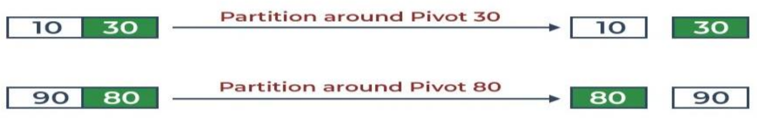
As the partition process is done recursively, it keeps on putting the pivot in its actual position in the sorted array. Repeatedly putting pivots in their actual position makes the array sorted. Follow the below images to understand how the recursive implementation of the partition algorithm helps to sort the array.

- Initial partition on the main array:



Quicksort: Performing the partition

- Partitioning of the subarrays:



Quick Sort Algorithm

Steps

1. Divide:
   - Select a pivot element from the array. This can be the first element, the last element, the middle element, or any random element. The choice of pivot can affect the performance of the algorithm.
   - Partition the array into two sub-arrays:
     - Elements less than the pivot.
     - Elements greater than the pivot.
2. Conquer:
   - Recursively apply the above steps to the sub-arrays.
3. Combine:
   - Since the sub-arrays are sorted in place, no additional steps are required to combine them.

Quick Sort: Best, Average, and Worst Cases

Quick sort's performance is significantly affected by the choice of the pivot element and the initial arrangement of elements in the array. Let's explore the best, average, and worst cases in detail.

Best Case: O(n log n)

- Scenario: The pivot element always divides the array into two nearly equal halves.
- Explanation:

    - When the pivot splits the array into two sub-arrays of equal (or nearly equal) length, the depth of the recursion tree is minimized.
    - At each level of recursion, we process n elements.
    - The number of levels in the recursion tree is log n (since the array is halved at each level).
    - Therefore, the total work done is n log n.
- Example: For an array [8, 2, 4, 1, 3, 7, 6, 5], if the pivot chosen at each step divides the array into two equal parts, we achieve the best case.

Average Case: O(n log n)

- Scenario: The pivot element divides the array into two sub-arrays that are not necessarily equal but not too unbalanced.
- Explanation:
    - In the average case, the pivot will partition the array such that the sizes of the two sub-arrays are fairly balanced over multiple recursive calls.
    - On average, each partition step reduces the size of the problem roughly by half.
    - The depth of the recursion tree is still proportional to log n, and at each level, the work done is n.
    - Therefore, the expected total work is n log n.
- Example: For an array [8, 2, 4, 1, 3, 7, 6, 5], if the pivot chosen results in sub-arrays that are not too unbalanced, the algorithm performs close to O(n log n).

Worst Case: $O(n^2)$

- Scenario: The pivot element is always the smallest or largest element, resulting in highly unbalanced partitions.
- Explanation:
    - In the worst case, the pivot does not split the array into balanced sub-arrays but instead results in one sub-array with $n-1$ elements and another with 0 elements.
    - This leads to the maximum depth of the recursion tree, which is n(since only one element is removed at each level).
    - At each level, we process n elements.
    - Therefore, the total work done is $n \times n = n^2$.
- Example: For an already sorted array [1, 2, 3, 4, 5, 6, 7, 8] or reverse sorted array [8, 7, 6, 5, 4, 3, 2, 1], if the pivot chosen is always the first or last element, the algorithm will exhibit the worst-case performance.

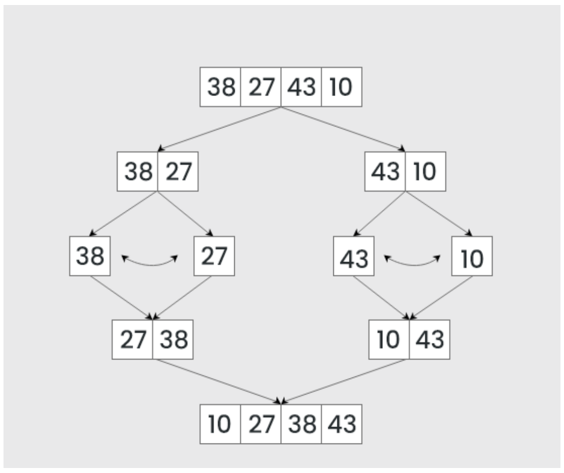
Summary

- Best Case: O(n log n)
    - Achieved when the pivot consistently splits the array into two nearly equal parts.
- Average Case: O(n log n)
    - Expected performance for a random array with good pivot selection.
- Worst Case: $O(n^2)$
    - Occurs when the pivot results in highly unbalanced partitions.

Merge Sort:

Merge sort is a sorting algorithm that follows the **divide-and-conquer** approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

In simple terms, we can say that the process of **merge sort** is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.



How does Merge Sort work?

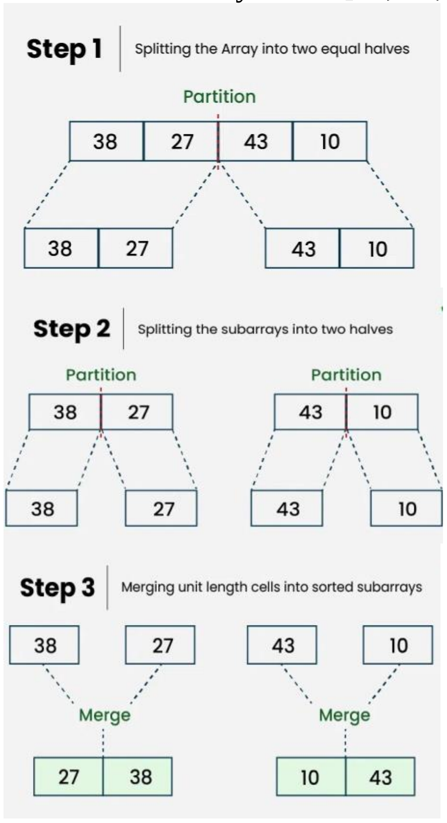
Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the **divide-and-conquer** approach to sort a given array of elements.

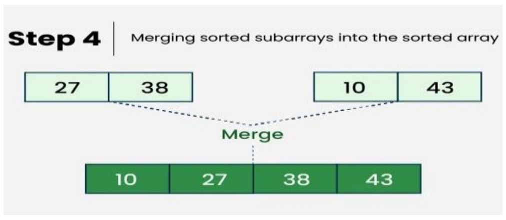
Here's a step-by-step explanation of how merge sort works:

1. **Divide:** Divide the list or array recursively into two halves until it can no more be divided.
2. **Conquer:** Each subarray is sorted individually using the merge sort algorithm.
3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

Illustration of Merge Sort:

Let's sort the array or list **[38, 27, 43, 10]** using Merge Sort


Step 1
Splitting the Array into two equal halves
Partition
38 27 43 10
38 27
43 10
Step 2
Splitting the subarrays into two halves
Partition
Partition
38 27
43 10
38
27
43
10
Step 3
Merging unit length cells into sorted subarrays
38
27
43
10
Merge
Merge
27 38
10 43



Let’s look at the working of above example:

Divide:

- **[38, 27, 43, 10]** is divided into **[38, 27**] and **[43, 10]**.
- **[38, 27]** is divided into **[38]** and **[27]**.
- **[43, 10]** is divided into **[43]** and **[10]**.

Conquer:

- **[38]** is already sorted.
- **[27]** is already sorted.
- **[43]** is already sorted.
- **[10]** is already sorted.

Merge:

- Merge **[38]** and **[27]** to get **[27, 38]**.
- Merge **[43]** and **[10]** to get **[10,43]**.
- Merge **[27, 38]** and **[10,43]** to get the final sorted list **[10, 27, 38, 43]**

Therefore, the sorted list is **[10, 27, 38, 43]**.

Algorithm of merge Sort:

Merge sort is a comparison-based sorting algorithm that uses the divide-and-conquer strategy. It divides the array into two halves, recursively sorts each half, and then merges the sorted halves to produce the sorted array.

Steps

1. **Divide**:
   - Divide the unsorted array into two roughly equal sub-arrays.
2. **Conquer**:
   - Recursively sort each sub-array.
3. **Combine**:
   - Merge the two sorted sub-arrays into one sorted array.

Merge sort's performance is generally consistent because it divides the array into two equal parts and then merges them in a sorted order. This results in a time complexity of O(n log n) for all cases.

Best Case: O(n log n)

- **Scenario**: The array is already sorted or any specific order.
- **Explanation**:
  - Merge sort will still divide the array into two halves recursively and merge them back together, regardless of the initial order of the elements.
  - The time complexity is based on the divide-and-conquer approach, which remains O(n log n) as it consistently divides the array and merges it.

Average Case: O(n log n)

- **Scenario**: The array elements are in random order.
- **Explanation**:
  - Similar to the best case, the array is divided and merged in a consistent manner.
  - The average-case time complexity remains O(n log n) due to the regular splitting and merging process.

Worst Case: O(n log n)

- **Scenario**: The array is in reverse order or any specific order that doesn't favor any specific divide or merge operation.
- **Explanation**:
  - The worst-case scenario still follows the same process of dividing and merging.
  - The time complexity for dividing the array into halves and merging them back remains O(n log n).

Space Complexity: O(n)

- Merge sort requires additional space proportional to the size of the array, as it needs to store the sub-arrays temporarily during the merge process.