**⭐GRAPH⭐**

**WHAT ARE GRAPHS**

A **graph data structure** consists of a finite (and possibly mutable) set of vertices or nodes or points, together with a set of unordered pairs of these vertices for an **undirected graph** or a set of ordered pairs for a **directed graph**.

Basically a Graph is a collection of nodes and connections between those nodes.

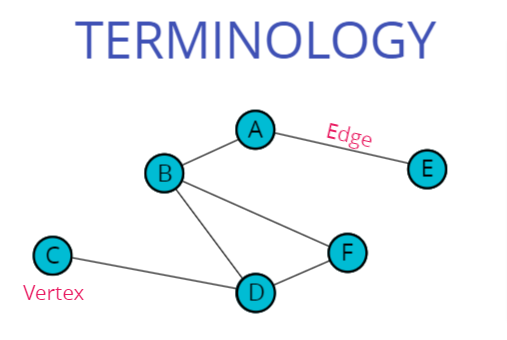
A tree is a type of graph, but a Graph doesn’t consist any starting node or ending node, doesn’t have root, child or leaf node.   
A tree is an undirected graph in which any two vertices/node are connected by exactly one path. On contrast, A graph might have multiple path to react to another node.

**USES FOR GRAPHS**

* Social Networks
* Location / Mapping
* Routing Algorithms
* Visual Hierarchy
* File System Optimizations
* EVERYWHERE!

**ESSENTIAL GRAPH TERMINOLOGY (Terms)**

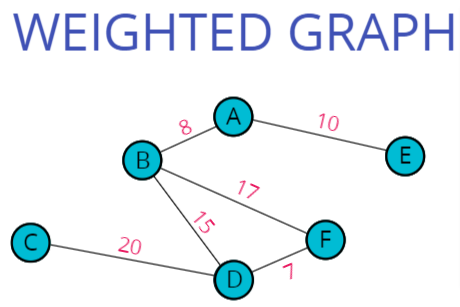
* **Vertex** - a node
* **Edge** - connection between nodes

  
**Unweighted Undirected Graph**

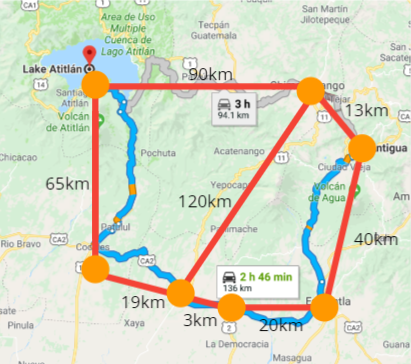
* **Weighted/Unweighted** - values assigned to distances between vertices.

Unweighted graph where each connection between the nodes/ each edge has no value associated with it.

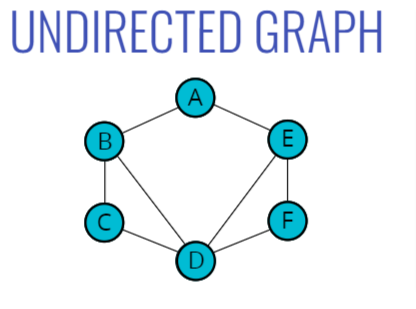
Weighted graph where each connection between the nodes/ each edge has value associated with it. When we assign a value to the edges, it becomes a weighted graph.



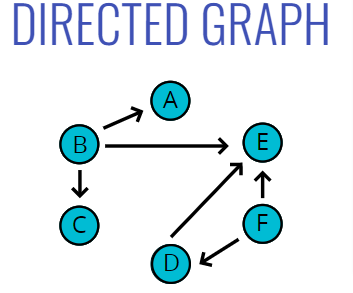
This is the **Weighted Undirected Graph**



* **Directed/Undirected** - Directions assigned to distanced between vertices.  
  In an undirected graph there is no direction to the edges/connections. So, they are two way connections.  
  For Ex: There is Two way connection in Facebook. As I sent a friend request to someone, and we both locked our profile. Then as he/she accepted my request, we both will able to see each other’s content posted on the profile.



In a Directed graph there is a direction to the edges/connections. So, they is only One way connections.  
For Ex: There is One way connection in Instagram. As I follow Justin Bieber, I can see the content of Justin but as Justin doesn’t follow me, So, he will not able to see my content because off course he doesn’t interested in my content.



**There are many different types of Graphs:**

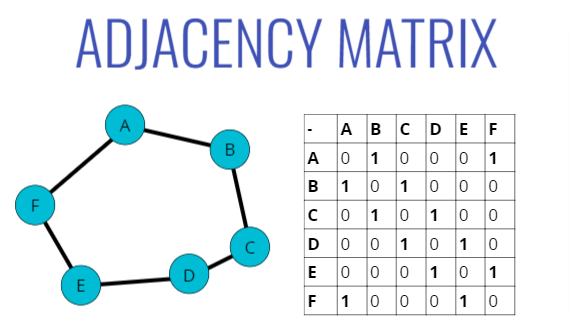
1. Based on Distinction in terms of the main definition  
   Undirected Graph  
   Directed Graph   
   Weighted graph, Simply Graph, Oriented Graph etc.
2. Import classes of graph  
   Regular graph  
   Complete Graph  
   Tree   
   Finite Graph etc.
3. And a lot more…

There are basically Two Approaches of Storing a graph:  
1. Adjancency Matrix

2. Adjancency List

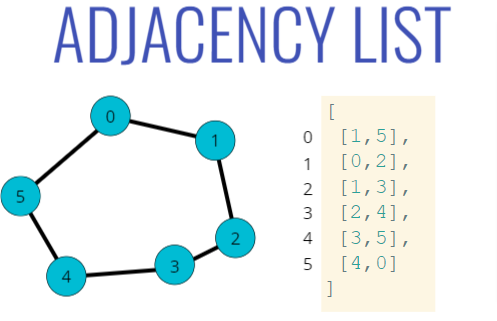
**ADJANCENCY MATRIX:-**

A matrix, is just a 2-Dimensional structure, usually implemented with nested arrays, but not always. And basically we store information in rows and columns.

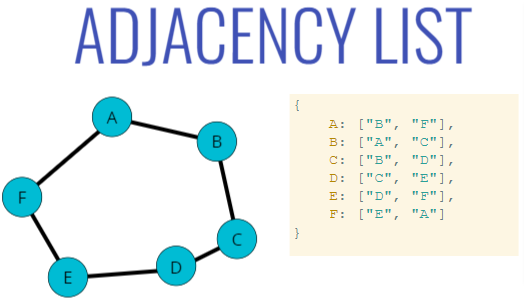


So, here we’re going to represent the connections of a graph using matrix.

**ADJANCENCY LIST:-**



If we’ve numbers in the Node, we’ll use .

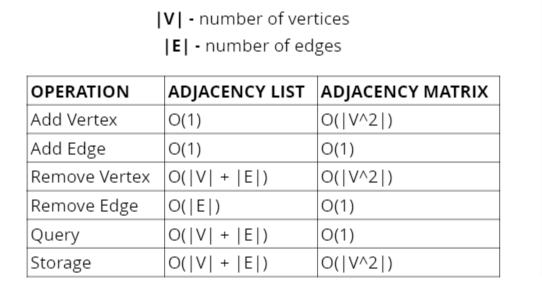


If we’ve String in the Node, we’ll use **Hash Table/Map** (use a key value pair of data structure)

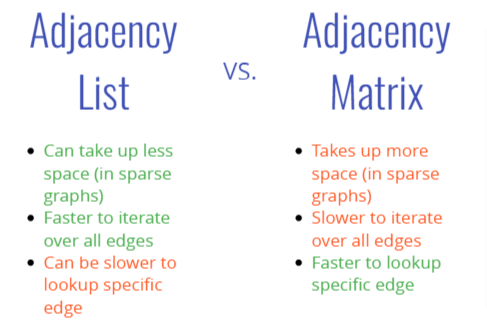
So, here we’re going to represent the connections of a graph using list.

**Differences & Big O between Adjacency List & Adjacency Matrix:-**

If we think about the storage



V: Vertex or Node  
E: Edge or Connection



**Note:**

Adjacency List is slower to query for the existence or to check it something, if an edge is in there.

Here, we’re going **to use Adjacency List to build Graphs**.

**Undirected Graph**

**Adding a Vertex:-**

**Pseudo Code:**

* Write a method called addVertex, which accepts a name of a vertex
* It should add a key to the adjacency list with the name of the vertex and set its value to be an empty array

g.addVertex("Tokyo")

{

"Tokyo": []

}

**Approach:**

* Adding vertex/node is the First thing to build any graph. We're going to store graph in adjacencyList, which will be hash Table/ hash Map.
* We're going to create a addingVertex function to add vertex in the Graph which accept a vertex and get store as key in adjacencyList.
* To avoid duplicate Vertex getting store in the Graph, we'll have the following Condition.
* IF(The vertex accepted from the paramenter is not already existed in adjacencyList, then only enter into this block. !adjacencyList[vertex])

As, vertex is unique, assign an Empty array[] as a value to that vertex (which get stored as key in adjacencyList).

**Code Implementation:**

class Graph {

constructor() {

this.adjacencyList = {};

}

addVertex(vertex){

**//If the vertex isn't already existed in the adjacencyList, then only add it as a key and assign an empty array corresponding to it. Otherwise gives undefined.**

if(!this.adjacencyList[vertex])

this.adjacencyList[vertex] = [];

}

}

const graph = new Graph();

Execute addVertex:-

graph.addVertex("Tokyo");  
 graph.addVertex("India");  
 graph.addVertex("India");

**Output:-**

graph ***//Checking adjacencyList by calling graph Object***

adjacencyList {  
 India: []  
 Tokyo: []  
}

undefined ***//Because India has already added in adjacencyList***

**ADDING AN EDGE:-**

**Pseudo Code:**

* This function should accept two vertices, we can call them vertex1 and vertex2
* The function should find in the adjacency list the key of vertex1 and push vertex2 to the array
* The function should find in the adjacency list the key of vertex2 and push vertex1 to the array
* Don't worry about handling errors/invalid vertices.

**Visualization:**

Available in the code File.

**Approach:**

* For adding an Edge between Two Vertices, We consider that, we already added the vertices.
* To Adding an Edge between two Vertices, we're going to create a function called 'addEdge(vertex1, vertex2)' accepting Two Vertices.
* If (As we got the vertex1 as the existed key (consisting an empty array []) in adjacencyList this.adjacencyList[vertex1])
* We're going to push vertex2 to its empty array[].
* If (As we got the vertex2 as the existed key (consisting an empty array []) in adjacencyList this.adjacencyList[vertex2])
* We're going to push vertex2 to its empty array[].

**Code Implementation:**

class Graph {

constructor() {

this.adjacencyList = {};

}

addVertex(vertex){

**//If the vertex isn't already existed in the adjacencyList, then only add it as a key and assign an empty array corresponding to it. Otherwise gives undefined.**

if(!this.adjacencyList[vertex])

this.adjacencyList[vertex] = [];

}

addEdge(vertex1, vertex2){

if(this.adjacencyList[vertex1])  
 this.adjacencyList[vertex1].push(vertex2);

if(this.adjacencyList[vertex2])  
 this.adjacencyList[vertex2].push(vertex1);

}

}

const graph = new Graph();

graph.addVertex("Tokyo");  
graph.addVertex("Dallas");  
graph.addVertex("Aspen");

**Execute addEdge:-**

graph.addEdge("Tokyo", "Dallas");

graph.addEdge("Dallas", "Aspen");

**Output:-**

graph

adjacencyList{

Aspen: ['Dallas']

Dallas: (2) ['Tokyo', 'Aspen']

Tokyo: ['Dallas']

}

**REMOVING AN EDGE:-**

**Pseudo Code:**

* This function should accept two vertices, we'll call them vertex1 and vertex2
* The function should reassign the key of vertex1 to be an array that does not contain vertex2
* The function should reassign the key of vertex2 to be an array that does not contain vertex1
* Don't worry about handling errors/invalid vertices

**Visualization:**

Available in the code File.

**Approach:**

* For removing an Edge between Two Vertices, We consider that, we already added the vertices and corresponding Edges between them.
* To Removing an Edge between two Vertices, we're going to create a function called 'removeEdge(vertex1, vertex2)' accepting Two Vertices.
* If (As we got the vertex1 as the existed key (consisting vertex2) in adjacencyList this.adjacencyList[vertex1])
  + We're going to assign a new array to that key, that array would be a filtered array, consisting only vertex1 and vertex2 will be removed to this array.
* If (As we got the vertex2 as the existed key (consisting vertex2) in adjacencyList this.adjacencyList[vertex2])
  + We're going to assign a new array to that key, that array would be a filtered array, consisting only vertex2 and vertex1 will be removed to this array.

**Code Implementation:**

class Graph {

constructor() {

this.adjacencyList = {};

}

addVertex(vertex){

**//If the vertex isn't already existed in the adjacencyList, then only add it as a key and assign an empty array corresponding to it. Otherwise gives undefined.**

if(!this.adjacencyList[vertex])

this.adjacencyList[vertex] = [];

}

addEdge(vertex1, vertex2){

if(this.adjacencyList[vertex1])

this.adjacencyList[vertex1].push(vertex2);

if(this.adjacencyList[vertex2])

this.adjacencyList[vertex2].push(vertex1);

}

removeEdge(vertex1, vertex2){

if(this.adjacencyList[vertex1])

**//Return a Filter Array, by filterring all the values that is not equal to vertex2 (Means returning array consisting only vertex2)**

this.adjacencyList[vertex1] = this.adjacencyList[vertex1].filter(element => element!==vertex2);

if(this.adjacencyList[vertex2])

**//Return a Filter Array, by filterring all the values that is not equal to vertex1 (Means returning array consisting only vertex1)**

this.adjacencyList[vertex2] = this.adjacencyList[vertex2].filter(element=>element!==vertex1);

}

}

const graph = new Graph();

**//Adding Vertex**

graph.addVertex("Tokyo");  
graph.addVertex("Dallas");  
graph.addVertex("Aspen");

**//Adding Edges between vertices**

graph.addEdge("Tokyo", "Dallas");  
graph.addEdge("Dallas", "Aspen");

**Execute removeEdge:-**

graph.removeEdge('Tokyo', 'Dallas');

**Output:-**

graph;

adjacencyList{

Aspen: ['Dallas']  
 Dallas: ['Aspen']  
 Tokyo: []

}

**REMOVING A VERTEX:-**

**Pseudo Code:**

* The function should accept a vertex to remove
* The function should loop as long as there are any other vertices in the adjacency list for that vertex
* Inside of the loop, call our **removeEdge** function with the vertex we are removing and any values in the adjacency list for that vertex
* delete the key in the adjacency list for that vertex

**Visualization:**

Available in the code File.

**Approach:**

* For removing a Vertex from the Graph, Firsly, we remove all its adjacentVertex (Means, Vertices connected with its Edges), then completely remove the required Vertex.
* To Removing a Vertex from the Graph, we're going to create a function called 'removeVertex(vertex)' accepting the require 'vertex' need to remove.
* If (As we got the vertex as the existed key (consisting adjacent Vertices) in adjacencyList this.adjacencyList[vertex])
* Since, the require vertex consisitinng adjacent Vertices inside an array as a value of the require Vertex. So, we've to store array of adjacent Vertices in 'adjVertexArr'
* Start a Loop till the length of adjVertexArr.length
  + Store each adjacent Vertex in a varible called 'adjacentVertex';
  + Call removeEdge(vertex, adjacentVertex) method by passing, the require 'vertex' that need to remove & 'adjacentVertex' that have a connection between 'vertex' and 'adjacentVertex' by an Edge.
* after removing all edges of require removed Vertex with its adjacent Vertices. Now only an Empty array[] will remain as a value of the require Vertex.
* Now, we can completely remove the require vertex from the Graph or say, the require key from the adjacencyList, by doing the following statament:
  + delete this.adjacencyList[vertex];

**Code Implementation:**

class Graph {

constructor() {

this.adjacencyList = {};

}

addVertex(vertex){

**//If the vertex isn't already existed in the adjacencyList, then only add it as a key and assign an empty array corresponding to it. Otherwise gives undefined.**

if(!this.adjacencyList[vertex])

this.adjacencyList[vertex] = [];

}

addEdge(vertex1, vertex2){

if(this.adjacencyList[vertex1])

this.adjacencyList[vertex1].push(vertex2);

if(this.adjacencyList[vertex2])

this.adjacencyList[vertex2].push(vertex1);

}

removeEdge(vertex1, vertex2){

if(this.adjacencyList[vertex1])

**//Return a Filter Array, by filterring all the values that is not equal to vertex2 (Means returning array consisting only vertex2)**

this.adjacencyList[vertex1] = this.adjacencyList[vertex1].filter(element => element!==vertex2);

if(this.adjacencyList[vertex2])

**//Return a Filter Array, by filterring all the values that is not equal to vertex1 (Means returning array consisting only vertex1)**

this.adjacencyList[vertex2] = this.adjacencyList[vertex2].filter(element=>element!==vertex1);

}

removeVertex(vertex){

if(this.adjacencyList[vertex]){

let adjVertexArr = this.adjacencyList[vertex];

for (let i = 0; i < adjVertexArr.length; i++) {

**//adjacentVertex are the vertices that have connections with 'Hong Kong' vertex** let adjacentVertex = adjVertexArr[i];

**// Remove all the edges of the required vertex (vertex going to be removed from the graph) with their corresponding vertices. And after that, In require vertex, only an Empty Array[] will remain as the value.**

this.removeEdge(vertex, adjacentVertex);

}

**//Delete the Hong Kong Vertex at the End, when only an empty array is left over there.**

delete this.adjacencyList[vertex];

}

}

}

const graph = new Graph();

**//Adding Vertex**

graph.addVertex("Tokyo");  
graph.addVertex("Dallas");  
graph.addVertex("Aspen");  
graph.addVertex("Hong Kong");  
graph.addVertex("Los Angeles");

**//Adding Edges between vertices**

graph.addEdge("Tokyo", "Dallas");  
graph.addEdge("Tokyo", "Hong Kong");  
graph.addEdge("Hong Kong", "Los Angeles");  
graph.addEdge("Dallas", "Aspen");  
graph.addEdge("Dallas", "Hong Kong");  
graph.addEdge("Dallas", "Los Angeles");

**// Execute removeVertex:-**

graph.removeVertex('Hong Kong');

**Output:-**

graph.adjacencyList;

adjacencyList{

Aspen: ['Dallas']

Dallas: (3) ['Tokyo', 'Aspen', 'Los Angeles']

Los Angeles: ['Dallas']

Tokyo: ['Dallas']

}

**⭐GRAPH TRAVERSAL⭐**

**GRAPH TRAVERSAL USES:**

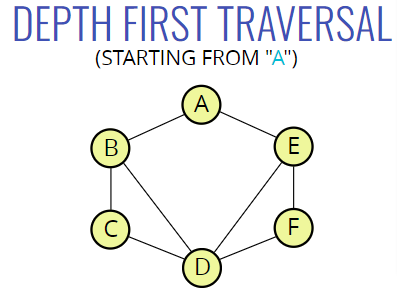
* Peer to peer networking
* Web crawlers
* Finding "closest" matches/recommendations
* Shortest path problems
* GPS Navigation
* Solving mazes
* AI (shortest path to win the game)

**DEPTH FIRST TRAVERSAL / SEARCH:**

In DFS, we prioritize deepening of the traversal rather than broad or say broadening or rather than widening. What it really means is just following the neighbours and continuing to follow neighbours before backtracking.

Note:

* There is no root node in Graph like there is on a tree.



Here, we’re starting from ‘A’. We want to explore as far as possible down one branch, Before we backtrack.

So, first of all we’ve to decide the order we’re going to visit things.

Then go with neighbours. Then, decide from which neighbour we’ve to go first. Here, we could visit either 1/1 (B or E).

**DFS Traversal: (Moving Alphabetical Order)**

Starting from ‘A’, then go with any neighbour. Let’s go with ‘B’, then going to explore B’s neighbour i.e. ‘C’. Off course we could explore ‘D’ but we’ve to be consistent in alphabetical order for now.

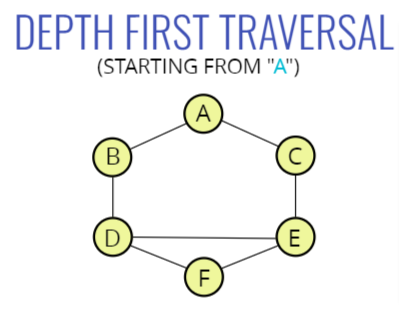
Now from ‘C’ we’ve two neighbours and we already visited ‘B’, this is not an important idea. We’ve to remember where we’ve been, otherwise we just go back to ‘B’. So we’ve done ‘B’ so that leaves us ‘D’. Now from ‘D’, we’ve two choices (‘E’ or ‘F’), and if we’re going alphabetically, we’re going to go to E and then to F.

A 🡪 B 🡪 C 🡪 D 🡪 E 🡪 F

Note: we’ve to keep storing those nodes that we’ve already been traversed otherwise, we’ll start backtracking to that nodes again.

So, We’re going to start off by implementing depth first traversal (DFS) of graph in two ways,

1. Recursively
2. Iteratively

  
We’re going to use this Graph.

DFS in Alphabetical Order:

[‘A’, ‘B’, ‘D’, ‘E’, ‘C’, ‘F’ ]

**1. Recursive (DFS)**

**Pseudo Code: (Basic)**

DFS(vertex):

if vertex is empty

return (this is base case)

add vertex to results list

mark vertex as visited

for each neighbor in vertex's neighbors:

if neighbor is not visited:

recursively call DFS on neighbour

**Pseudo Code: (More Detailed)**

* The function should accept a starting node
* Create a list to store the end result, to be returned at the very end
* Create an object to store visited vertices
* Create a helper function which accepts a vertex
* The helper function should return early if the vertex is empty
* The helper function should place the vertex it accepts into the visited object and push that vertex into the result array.
* Loop over all of the values in the adjacencyList for that vertex
* If any of those values have not been visited, recursively invoke the helper function with that vertex
* Invoke the helper function with the starting vertex
* Return the result array

**Approach:-**

* We're going to Depth First Traverse through the graph.

Means, we've to continuously following through the vertex neighbour and continous to follow neighbours before backtracking.

Create a function called 'depthFirstRecursive(start)' which accept a starting vertex for Depth First Traversal.

* Declared an empty 'result' array, which is going to be the end result after Completion of Depth First Traversal across the Graph.

Declared an empty 'visited' ojbect, which is going to store the visited vertex by setting the visited vertex as true.

Assign Complete AdjacencyList to a varible called adjacencyList.

* Define an Immediately Invoked function, which is going to call with the starting vertex.
* Inside it, Define a dfs(vertex) function, which is going to accept a vertex and Recursively call this function.
  + dfs(vertex) function will return null, if their is no vertex passing through it.
* Just push the vertex in the result array
* Just assign the getting vertex as true while storing it to the visited{} object.
* If we got the vertex, then At adjacencyList their would be an adjacent array(consisitinng adjacentVertex) corresponding to each vertex.
* Start looping over the adjacentArray of corresponding vertex to get every adjacentVertex in it.
  + if(the adjacentVertex is not found in visited{} as key )
    - Recursively call dfs function with the adjacentVertex. reutrn dfs(adjacentVertex);

**Code Implementation:**

class Graph {

constructor() {

this.adjacencyList={};

}

addVertex(vertex){

if(!this.adjacencyList[vertex])

this.adjacencyList[vertex] = [];

}

addEdge(vertex1, vertex2){

if(this.adjacencyList[vertex1])

this.adjacencyList[vertex1].push(vertex2);

if(this.adjacencyList[vertex2])

this.adjacencyList[vertex2].push(vertex1);

}

depthFirstRecursive(start){

const result = [],

visited = {},

adjacencyList = this.adjacencyList;

**// We're going to call this 'Immediately Invoked Function' Recursively.**

(function dfs(vertex) {

**// Edge Case**

if(!vertex) return null;

result.push(vertex);

visited[vertex] = true;

**// Here, current context of 'this' pointing to dfs(), where, this.adjacencyList is not defined. That's we defined it in adjacencyList.**

adjacencyList[vertex].forEach(neighbour =>{

if(!visited[neighbour])

return dfs(neighbour);

});

}

)(start);

return result;

}

}

const graph = new Graph();

**// Adding Vertex**

graph.addVertex("A");  
graph.addVertex("B");  
graph.addVertex("C");  
graph.addVertex("D");  
graph.addVertex("E");  
graph.addVertex("F");

**// Adding Edge between vertices**

graph.addEdge("A","B");  
graph.addEdge("A","C");  
graph.addEdge("B","D");  
graph.addEdge("C","E");  
graph.addEdge("D","E");  
graph.addEdge("D","F");  
graph.addEdge("E","F");

**// Depth First Traversal Recursively**

graph.depthFirstRecursive('A');

**Output:**['A', 'B', 'D', 'E', 'C', 'F']

**2. Iterative (DFS)**

**Pseudo Code: (Basic)**

DFS-iterative(start):

let S be a stack

S.push(start)

while S is not empty

vertex = S.pop()

if vertex is not labeled as discovered:

visit vertex (add to result list)

label vertex as discovered

for each of vertex's neighbors, N do

S.push(N)

**Pseudo Code: (More Detailed)**

* The function should accept a starting node
* Create a stack to help use keep track of vertices (use a list/array)
* Create a list to store the end result, to be returned at the very end
* Create an object to store visited vertices
* Add the starting vertex to the stack, and mark it visited
* While the stack has something in it:
* Pop the next vertex from the stack
* If that vertex hasn't been visited yet:
* Mark it as visited
* Add it to the result list
* Push all of its neighbors into the stack
* Return the result array

**Approach:**

* We're going to Depth First Traverse through the graph in iterative way.
* Means, we've to continuously following through the vertex neighbour and continous to follow neighbours before backtracking.
* Create a function called 'depthFirstIterative(start)' which accept a starting vertex for Depth First Traversal.
* Create a 'stack' with an empty array, which is going to keep track of vertices & ADD the Starting vertex to the stack;
* Declared an empty 'result' array or say list, to store the end result, to be return at the very end.
* Declared an empty 'visited' ojbect, to store the visited vertex by assigning the visited vertex as true.
* Just initialize the 'currentVertex';
* Mark the starting vertex as visited by assigning it as true in visited object
* Run a loop till their is something in the stack.
* Popout the next vertex from the stack and assign it in the 'currentVertex';
* Push the currentVertex to the result list.
* Since, the currentVertex has its own adjacentArray consisitinng adjacentVertices or say neighbours.
  + So, If(the neighbour is not visited yet !visited[neighbour], means undefined and !undefined is === true)
    - Mark it as visited by assigning it as true.
    - Push it to the stack.
* return the result list.

**Code Implementation:**

class Graph {

constructor() {

this.adjacencyList={};

}

addVertex(vertex){

if(!this.adjacencyList[vertex])

this.adjacencyList[vertex] = [];

}

addEdge(vertex1, vertex2){

if(this.adjacencyList[vertex1])

this.adjacencyList[vertex1].push(vertex2);

if(this.adjacencyList[vertex2])

this.adjacencyList[vertex2].push(vertex1);

}

depthFirstRecursive(start){

const result = [],  
 visited = {},

adjacencyList = this.adjacencyList;

**// We're going to call this 'Immediately Invoked Function' Recursively.** (function dfs(vertex) {

**// Edge Case** if(!vertex) return null;

result.push(vertex);

visited[vertex] = true;

**// Here, current context of 'this' pointing to dfs(), where, this.adjacencyList is not defined. That's we defined it in adjacencyList.**

adjacencyList[vertex].forEach(neighbour =>{

if(!visited[neighbour])

return dfs(neighbour);

});

}

)(start);

return result;

}

depthFirstIterative(start){

const stack = [start], ***//keep track of vertices & Added the Starting vertex to the stack;*** result = [], ***//list to store the end result***  visited = {}; ***//object to store visited vertices***

let currentVertex;

visited[start] = true; ***// Marked the starting vertex as visited***

while(stack.length){

currentVertex = stack.pop();

result.push(currentVertex);

this.adjacencyList[currentVertex].forEach(neighbour => {

if(!visited[neighbour]){

visited[neighbour] = true;

stack.push(neighbour);

}

});

}

return result;

}

}

const graph = new Graph();

**// Adding Vertex**  
graph.addVertex("A");  
graph.addVertex("B");  
graph.addVertex("C");  
graph.addVertex("D");  
graph.addVertex("E");  
graph.addVertex("F");

**// Adding Edge between vertices**  
graph.addEdge("A","B");  
graph.addEdge("A","C");  
graph.addEdge("B","D");  
graph.addEdge("C","E");  
graph.addEdge("D","E");  
graph.addEdge("D","F");  
graph.addEdge("E","F");

**// Depth First Traversal Iteratively**

graph.depthFirstIterative('A');

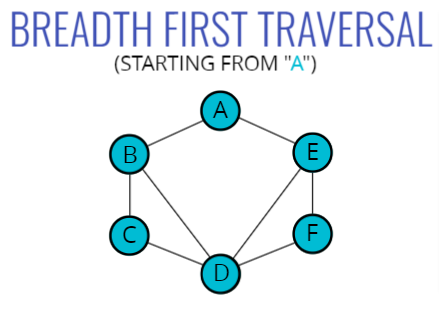
**Output:-**['A', 'C', 'E', 'F', 'D', 'B']

**BREADTH FIRST TRAVERSAL / SEARCH:**

Breadth first prioritizes visiting all of the neighbours at a given depth/level/height before moving downwards or visiting neighbours of neighbours.

We can just say horizontal before vertical, Before descending down.

**Example:**



**[‘A’, ‘B’, ‘E’, ‘C’, ‘D’, ‘F’]** (Alphabetical Order)

Or

**[‘A’, ‘E’, ‘B’, ‘F’, ‘D’, ‘C’,]**

Or

**[‘A’, ‘E’, ‘B’, ‘C’, ‘F’, ‘D’,]**

Or

**[‘A’, ‘E’, ‘B’, ‘D’, ‘F’, ‘C’,]**

**Note:**

* ‘A’, ‘B’, ‘E’ 🡪 Level-1 / Height-1 Completed
* ‘B’🡪‘C’, ‘B’🡪‘D’, ‘E’🡪‘D’, ‘E’🡪‘F’ are placed at level-2 or say Height-2
* The Order of visiting vertices at the same level/height doesn’t matter.  
  It could be like this as well, ‘E’🡪‘D’ , ‘E’🡪‘F’, ‘B’🡪‘C’,
* It would be very **similar to** **DFS of Iterative Version**, but rather than using stack we’re going to use Queue. Although Data structure would be the same i.e an Array. Just the algo. That going to implement is FIFO.

**Pseudo Code:**

* This function should accept a starting vertex
* Create a queue (you can use an array) and place the starting vertex in it
* Create an array to store the nodes visited
* Create an object to store nodes visited
* Mark the starting vertex as visited
* Loop as long as there is anything in the queue
* Remove the first vertex from the queue and push it into the array that stores nodes visited
* Loop over each vertex in the adjacency list for the vertex you are visiting.
* If it is not inside the object that stores nodes visited, mark it as visited and enqueue that vertex
* Once you have finished looping, return the array of visited nodes.

**Approach:**

* We're going to Breadth First Traverse through the graph.

Means, We're going to visit all of the neighbours at a given depth/level/height before moving downwards or visiting neighbours of neighbours.

Create a function called 'breadthFirstTraversal(start)' which accept a starting vertex for Breadth First Traversal.

* Create a 'queue' with an empty array, which is going to keep track of vertices & ADD the Starting vertex to the queue;

Declared an empty 'result' array or say list, to store the end result, to be return at the very end.

Declared an empty 'visited' object, to store the visited vertex by assigning the visited vertex as true.

Just initialize the 'currentVertex';

* Mark the starting vertex as visited by assigning it as true in visited object
* Run a loop till their is something in the queue.
* shift() the previous vertex from the queu and assign it in the 'currentVertex';
* Push the currentVertex to the result list.
* Since, the currentVertex has its own adjacentArray consisiting adjacentVertices or say neighbours in the AdjacencyList.
  + So, If(the neighbour is not visited yet !visited[neighbour], means undefined and !undefined is === true)
* Mark it as visited by assigning it to true.
* Push it to the queue.
* return the result list.

**Code Implementation:**

class Graph {

constructor() {

this.adjacencyList={};

}

addVertex(vertex){

if(!this.adjacencyList[vertex])

this.adjacencyList[vertex] = [];

}

addEdge(vertex1, vertex2){

if(this.adjacencyList[vertex1])

this.adjacencyList[vertex1].push(vertex2);

if(this.adjacencyList[vertex2])

this.adjacencyList[vertex2].push(vertex1);

}

breadthFirstTraversal(start){

const queue = [start], ***//keep track of vertices & Added the Starting vertex to the queue;***

result = [], ***//list to store the end result***

visited = {}; ***//object to store visited vertices***

let currentVertex;

visited[start] = true; ***// Marked the starting vertex as visited***

while(queue.length){

currentVertex = queue.shift();

result.push(currentVertex);

this.adjacencyList[currentVertex].forEach(neighbour=>{

if(!visited[neighbour]){

**// Marking the neighbour as visited**

visited[neighbour] = true;

queue.push(neighbour);

}

});

**// Reversing the Traversing (means, Rather than going with left neighbour, Starting vertex will go with the right neighbour by reverse the adjacentArray).**

// this.adjacencyList[currentVertex].reverse().forEach(neighbour=>{

// if(!visited[neighbour]){

// // Marking the neighbour as visited

// visited[neighbour] = true;

// queue.push(neighbour);

// }

// });

}

return result;

}

}

const graph = new Graph();

**// Adding Vertex**

graph.addVertex("A");  
graph.addVertex("B");  
graph.addVertex("C");  
graph.addVertex("D");  
graph.addVertex("E");  
graph.addVertex("F");

**// Adding Edge between vertices**graph.addEdge("A","B");  
graph.addEdge("A","E");  
graph.addEdge("B","C");  
graph.addEdge("B","D");  
graph.addEdge("C","D");  
graph.addEdge("E","F");  
graph.addEdge("E","D");  
graph.addEdge("D","F");

***// Breadth First Traversal***graph.breadthFirstTraversal('A');

**Output:-**

[‘A’, ‘B’, ‘E’, ‘C’, ‘D’, ‘F’]

Or

['A', 'E', 'B', 'D', 'F', 'C'] ***//After reversing the adjacentArray. (means Starting vertex will go with the right neighbour first.***