**Graphs and its Algorithms**

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as,

*A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes.*

Diagram

Description automatically generated

The following two are the most commonly used representations of a graph.   
**1.** Adjacency Matrix   
**2.** Adjacency List   
There are other representations also like, Incidence Matrix and Incidence List. The choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and ease of use.   
**Adjacency Matrix:**   
Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w. 

The adjacency matrix for the above example graph is:

Adjacency Matrix Representation

*Pros:* Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).  
*Cons:* Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.   
Please see [this](https://ide.geeksforgeeks.org/9je5j6jJ13) for a sample Python implementation of adjacency matrix.

**Adjacency List:**   
An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the list of vertices adjacent to the***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph. 

Adjacency List Representation of Graph

Note that in the below implementation, we use dynamic arrays (vector in C++/ArrayList in Java) to represent adjacency lists instead of the linked list. The vector implementation has advantages of cache friendliness. 

|  |
| --- |
| // A simple representation of graph using STL  #include<bits/stdc++.h>  **using** **namespace** std;    // A utility function to add an edge in an  // undirected graph.  **void** addEdge(vector<**int**> adj[], **int** u, **int** v)  {      adj[u].push\_back(v);      adj[v].push\_back(u);  }    // A utility function to print the adjacency list  // representation of graph  **void** printGraph(vector<**int**> adj[], **int** V)  {  **for** (**int** v = 0; v < V; ++v)      {          cout << "\n Adjacency list of vertex "               << v << "\n head ";  **for** (**auto** x : adj[v])             cout << "-> " << x;  **printf**("\n");      }  }    // Driver code  **int** main()  {  **int** V = 5;      vector<**int**> adj[V];      addEdge(adj, 0, 1);      addEdge(adj, 0, 4);      addEdge(adj, 1, 2);      addEdge(adj, 1, 3);      addEdge(adj, 1, 4);      addEdge(adj, 2, 3);      addEdge(adj, 3, 4);      printGraph(adj, V);  **return** 0;  } |

*Pros:* Saves space O(|V|+|E|) . In the worst case, there can be C(V, 2) number of edges in a graph thus consuming O(V^2) space. Adding a vertex is easier.  
*Cons:* Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(V).

# Add and Remove Edge in Adjacency List representation of a Graph

*The graph and the corresponding adjacency matrix after insertion of edges:*

*[Calendar

Description automatically generated](https://media.geeksforgeeks.org/wp-content/uploads/20200604170814/add-and-remove-edge-in-adjacency-matrix-representation-initial1.jpg)*

*The graph after removal and adjacency matrix after removal of edge between vertex****X****and****Y****:*

*[Calendar

Description automatically generated with medium confidence](https://media.geeksforgeeks.org/wp-content/uploads/20200604170842/add-and-remove-edge-in-adjacency-matrix-representation-final2.jpg)*

*If the edge between****1****and****4****has to be removed, then the above graph and the adjacency list transforms to:*

*A picture containing text, clipart, clock

Description automatically generatedDiagram

Description automatically generated*

**Approach:** The idea is to represent the graph as an [array of vectors](https://www.geeksforgeeks.org/2d-vector-in-cpp-with-user-defined-size/) such that every vector represents adjacency list of the vertex. 

* **Adding an edge:** Adding an edge is done by inserting both of the vertices connected by that edge in each others list. For example, if an edge between **(u, v)** has to be added, then **u** is stored in **v’s vector list** and **v** is stored in **u’s vector list**. ([push\_back](https://www.geeksforgeeks.org/vectorpush_back-vectorpop_back-c-stl/))
* **Deleting an edge:** To delete edge between **(u, v)**, **u’s adjacency list** is traversed until **v** is found and it is removed from it. The same operation is performed for **v**.([erase](https://www.geeksforgeeks.org/vectorclear-vectorerase-c-stl/))

Below is the implementation of the approach: 

|  |
| --- |
| // C++ implementation of the above approach    #include <bits/stdc++.h>  **using** **namespace** std;    // A utility function to add an edge in an  // undirected graph.  **void** addEdge(vector<**int**> adj[], **int** u, **int** v)  {      adj[u].push\_back(v);      adj[v].push\_back(u);  }    // A utility function to delete an edge in an  // undirected graph.  **void** delEdge(vector<**int**> adj[], **int** u, **int** v)  {      // Traversing through the first vector list      // and removing the second element from it  **for** (**int** i = 0; i < adj[u].size(); i++) {  **if** (adj[u][i] == v) {              adj[u].erase(adj[u].begin() + i);  **break**;          }      }        // Traversing through the second vector list      // and removing the first element from it  **for** (**int** i = 0; i < adj[v].size(); i++) {  **if** (adj[v][i] == u) {              adj[v].erase(adj[v].begin() + i);  **break**;          }      }  }    // A utility function to print the adjacency list  // representation of graph  **void** printGraph(vector<**int**> adj[], **int** V)  {  **for** (**int** v = 0; v < V; ++v) {          cout << "vertex " << v << " ";  **for** (**auto** x : adj[v])              cout << "-> " << x;  **printf**("\n");      }  **printf**("\n");  }    // Driver code  **int** main()  {  **int** V = 5;      vector<**int**> adj[V];        // Adding edge as shown in the example figure      addEdge(adj, 0, 1);      addEdge(adj, 0, 4);      addEdge(adj, 1, 2);      addEdge(adj, 1, 3);      addEdge(adj, 1, 4);      addEdge(adj, 2, 3);      addEdge(adj, 3, 4);        // Printing adjacency matrix      printGraph(adj, V);        // Deleting edge (1, 4)      // as shown in the example figure      delEdge(adj, 1, 4);        // Printing adjacency matrix      printGraph(adj, V);    **return** 0;  } |

**Output:**

vertex 0 -> 1-> 4

vertex 1 -> 0-> 2-> 3-> 4

vertex 2 -> 1-> 3

vertex 3 -> 1-> 2-> 4

vertex 4 -> 0-> 1-> 3

vertex 0 -> 1-> 4

vertex 1 -> 0-> 2-> 3

vertex 2 -> 1-> 3

vertex 3 -> 1-> 2-> 4

vertex 4 -> 0-> 3

# Add and Remove vertex in Adjacency List representation of Graph

**Removing a Vertex in the Graph:** To remove a vertex from the graph, we need to check if that vertex exists in the graph or not and if that vertex exists then we need to shift the rows to the left and the columns upwards of the adjacency matrix so that the row and column values of the given vertex gets replaced by the values of the next vertex and then decrease the number of vertices by 1.In this way that particular vertex will be removed from the adjacency matrix.

|  |
| --- |
| **void** removeVertex(**int** x)  {      // checking if the vertex is present  **if** (x > n) {          cout << "\nVertex not present!";  **return**;      }  **else** {  **int** i;            // removing the vertex  **while** (x < n) {              // shifting the rows to left side  **for** (i = 0; i < n; ++i) {                  g[i][x] = g[i][x + 1];              }                // shifting the columns upwards  **for** (i = 0; i < n; ++i) {                  g[x][i] = g[x + 1][i];              }              x++;          }            // decreasing the number of vertices          n--;      }  } |

The above method is a public member function of the class Graph which removes an existing vertex from the graph by shifting the rows to the left and shifting the columns up to replace the row and column values of that vertex with the next vertex and then decreases the number of vertices by 1 in the graph.

* **Following is a complete program that uses all of the above methods in a Graph.**

|  |
| --- |
| // C++ program to add and remove Vertex in Adjacency Matrix    #include <iostream>    **using** **namespace** std;    **class** Graph {  **private**:      // number of vertices  **int** n;        // adjacency matrix  **int** g[10][10];    **public**:      // constructor      Graph(**int** x)      {          n = x;            // initializing each element of the adjacency matrix to zero  **for** (**int** i = 0; i < n; ++i) {  **for** (**int** j = 0; j < n; ++j) {                  g[i][j] = 0;              }          }      }    **void** displayAdjacencyMatrix()      {          cout << "\n\n Adjacency Matrix:";            // displaying the 2D array  **for** (**int** i = 0; i < n; ++i) {              cout << "\n";  **for** (**int** j = 0; j < n; ++j) {                  cout << " " << g[i][j];              }          }      }    **void** addEdge(**int** x, **int** y)      {            // checks if the vertex exists in the graph  **if** ((x >= n) || (y > n)) {              cout << "Vertex does not exists!";          }            // checks if the vertex is connecting to itself  **if** (x == y) {              cout << "Same Vertex!";          }  **else** {              // connecting the vertices              g[y][x] = 1;              g[x][y] = 1;          }      }    **void** addVertex()      {          // increasing the number of vertices          n++;  **int** i;            // initializing the new elements to 0  **for** (i = 0; i < n; ++i) {              g[i][n - 1] = 0;              g[n - 1][i] = 0;          }      }    **void** removeVertex(**int** x)      {          // checking if the vertex is present  **if** (x > n) {              cout << "\nVertex not present!";  **return**;          }  **else** {  **int** i;                // removing the vertex  **while** (x < n) {                  // shifting the rows to left side  **for** (i = 0; i < n; ++i) {                      g[i][x] = g[i][x + 1];                  }                    // shifting the columns upwards  **for** (i = 0; i < n; ++i) {                      g[x][i] = g[x + 1][i];                  }                  x++;              }                // decreasing the number of vertices              n--;          }      }  };    **int** main()  {      // creating objects of class Graph      Graph obj(4);        // calling methods      obj.addEdge(0, 1);      obj.addEdge(0, 2);      obj.addEdge(1, 2);      obj.addEdge(2, 3);      // the adjacency matrix created      obj.displayAdjacencyMatrix();        // adding a vertex to the graph      obj.addVertex();      // connecting that vertex to other existing vertices      obj.addEdge(4, 1);      obj.addEdge(4, 3);      // the adjacency matrix with a new vertex      obj.displayAdjacencyMatrix();        // removing an existing vertex in the graph      obj.removeVertex(1);      // the adjacency matrix after removing a vertex      obj.displayAdjacencyMatrix();    **return** 0;  } |

**Output:**

Adjacency Matrix:

0 1 1 0

1 0 1 0

1 1 0 1

0 0 1 0

Adjacency Matrix:

0 1 1 0 0

1 0 1 0 1

1 1 0 1 0

0 0 1 0 1

0 1 0 1 0

Adjacency Matrix:

0 1 0 0

1 0 1 0

0 1 0 1

0 0 1 0

Adjacency matrices waste a lot of memory space. Such matrices are found to be very sparse. This representation requires space for n\*n elements, the time complexity of the addVertex() method is O(n), and the time complexity of the removeVertex() method is O(n\*n) for a graph of n vertices.

From the output of the program, the Adjacency Matrix is:

A picture containing application

Description automatically generated

And the Graph depicted by the above Adjacency Matrix is:

A picture containing text, clipart, vector graphics

Description automatically generated