# Graph representation

In this article, we will discuss the ways to represent the graph. By Graph representation, we simply mean the technique to be used to store some graph into the computer's memory.

A graph is a data structure that consist a sets of vertices (called nodes) and edges. There are two ways to store Graphs into the computer's memory:

* **Sequential representation** (or, Adjacency matrix representation)
* **Linked list representation** (or, Adjacency list representation)

### Sequential representation

In sequential representation, there is a use of an adjacency matrix to represent the mapping between vertices and edges of the graph. We can use an adjacency matrix to represent the undirected graph, directed graph, weighted directed graph, and weighted undirected graph.

If adj[i][j] = w, it means that there is an edge exists from vertex i to vertex j with weight w.

An entry Aij in the adjacency matrix representation of an undirected graph G will be 1 if an edge exists between Vi and Vj. If an Undirected Graph G consists of n vertices, then the adjacency matrix for that graph is n x n, and the matrix A = [aij] can be defined as -

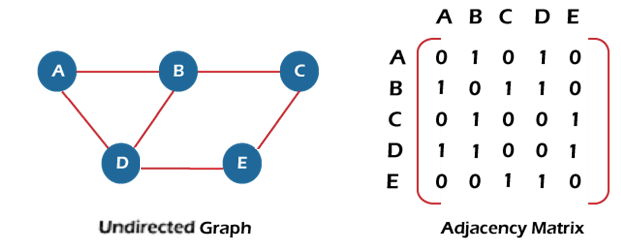
aij = 1 {if there is a path exists from Vi to Vj}

aij = 0 {Otherwise}

It means that, in an adjacency matrix, 0 represents that there is no association exists between the nodes, whereas 1 represents the existence of a path between two edges.

If there is no self-loop present in the graph, it means that the diagonal entries of the adjacency matrix will be 0.

Now, let's see the adjacency matrix representation of an undirected graph.



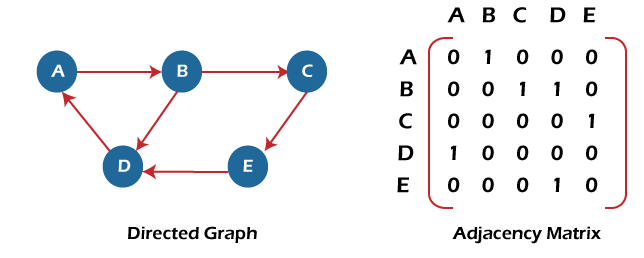
In the above figure, an image shows the mapping among the vertices (A, B, C, D, E), and this mapping is represented by using the adjacency matrix.

There exist different adjacency matrices for the directed and undirected graph. In a directed graph, an entry Aij will be 1 only when there is an edge directed from Vi to Vj.

### Adjacency matrix for a directed graph

In a directed graph, edges represent a specific path from one vertex to another vertex. Suppose a path exists from vertex A to another vertex B; it means that node A is the initial node, while node B is the terminal node.

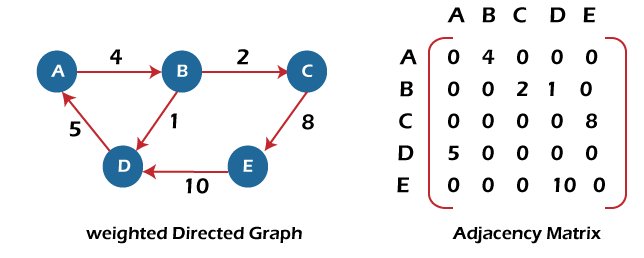
Consider the below-directed graph and try to construct the adjacency matrix of it.



In the above graph, we can see there is no self-loop, so the diagonal entries of the adjacent matrix are 0.

**Adjacency matrix for a weighted directed graph**

It is similar to an adjacency matrix representation of a directed graph except that instead of using the '1' for the existence of a path, here we have to use the weight associated with the edge. The weights on the graph edges will be represented as the entries of the adjacency matrix.



In the above image, we can see that the adjacency matrix representation of the weighted directed graph is different from other representations. It is because, in this representation, the non-zero values are replaced by the actual weight assigned to the edges.

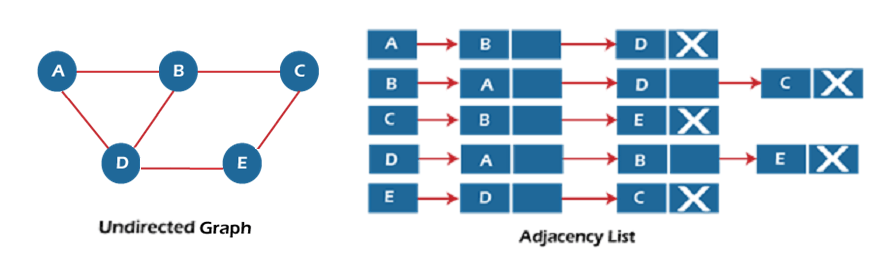
Adjacency matrix is easier to implement and follow. An adjacency matrix can be used when the graph is dense and a number of edges are large.

Though, it is advantageous to use an adjacency matrix, but it consumes more space. Even if the graph is sparse, the matrix still consumes the same space.

## Linked list representation

An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.

Let's see the adjacency list representation of an undirected graph.

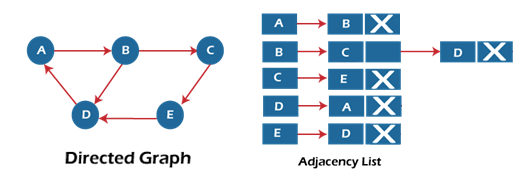


In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list.

An adjacency list is maintained for each node present in the graph, which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed, then store the NULL in the pointer field of the last node of the list.

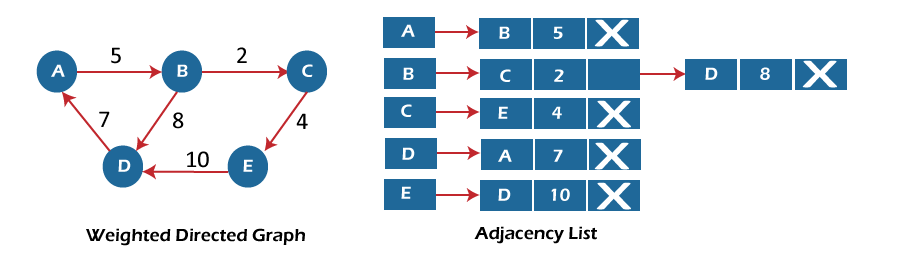
The sum of the lengths of adjacency lists is equal to twice the number of edges present in an undirected graph.

Now, consider the directed graph, and let's see the adjacency list representation of that graph.



For a directed graph, the sum of the lengths of adjacency lists is equal to the number of edges present in the graph.

Now, consider the weighted directed graph, and let's see the adjacency list representation of that graph.



In the case of a weighted directed graph, each node contains an extra field that is called the weight of the node.

In an adjacency list, it is easy to add a vertex. Because of using the linked list, it also saves space.

### Implementation of adjacency matrix representation of Graph

Now, let's see the implementation of adjacency matrix representation of graph in C.

In this program, there is an adjacency matrix representation of an undirected graph. It means that if there is an edge exists from vertex A to vertex B, there will also an edge exists from vertex B to vertex A.

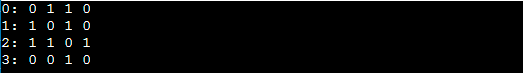
Here, there are four vertices and five edges in the graph that are non-directed.

/\* Adjacency Matrix representation of an undirected graph in C \*/

1. #include <stdio.h>
2. #define V 4 /\* number of vertices in the graph \*/
4. /\* function to initialize the matrix to zero \*/
5. **void** init(**int** arr[][V]) {
6. **int** i, j;
7. **for** (i = 0; i < V; i++)
8. **for** (j = 0; j < V; j++)
9. arr[i][j] = 0;
10. }
12. /\* function to add edges to the graph \*/
13. **void** insertEdge(**int** arr[][V], **int** i, **int** j) {
14. arr[i][j] = 1;
15. arr[j][i] = 1;
16. }
18. /\* function to print the matrix elements \*/
19. **void** printAdjMatrix(**int** arr[][V]) {
20. **int** i, j;
21. **for** (i = 0; i < V; i++) {
22. printf("%d: ", i);
23. **for** (j = 0; j < V; j++) {
24. printf("%d ", arr[i][j]);
25. }
26. printf("\n");
27. }
28. }
30. **int** main() {
31. **int** adjMatrix[V][V];
33. init(adjMatrix);
34. insertEdge(adjMatrix, 0, 1);
35. insertEdge(adjMatrix, 0, 2);
36. insertEdge(adjMatrix, 1, 2);
37. insertEdge(adjMatrix, 2, 0);
38. insertEdge(adjMatrix, 2, 3);
40. printAdjMatrix(adjMatrix);
42. **return** 0;
43. }

**Output:**

After the execution of the above code, the output will be -



### Implementation of adjacency list representation of Graph

Now, let's see the implementation of adjacency list representation of graph in C.

In this program, there is an adjacency list representation of an undirected graph. It means that if there is an edge exists from vertex A to vertex B, there will also an edge exists from vertex B to vertex A.

1. /\* Adjacency list representation of a graph in C \*/
2. #include <stdio.h>
3. #include <stdlib.h>
5. /\* structure to represent a node of adjacency list \*/
6. **struct** AdjNode {
7. **int** dest;
8. **struct** AdjNode\* next;
9. };
11. /\* structure to represent an adjacency list \*/
12. **struct** AdjList {
13. **struct** AdjNode\* head;
14. };
16. /\* structure to represent the graph \*/
17. **struct** Graph {
18. **int** V; /\*number of vertices in the graph\*/
19. **struct** AdjList\* array;
20. };

23. **struct** AdjNode\* newAdjNode(**int** dest)
24. {
25. **struct** AdjNode\* newNode = (**struct** AdjNode\*)malloc(**sizeof**(**struct** AdjNode));
26. newNode->dest = dest;
27. newNode->next = NULL;
28. **return** newNode;
29. }
31. **struct** Graph\* createGraph(**int** V)
32. {
33. **struct** Graph\* graph = (**struct** Graph\*)malloc(**sizeof**(**struct** Graph));
34. graph->V = V;
35. graph->array = (**struct** AdjList\*)malloc(V \* **sizeof**(**struct** AdjList));
37. /\* Initialize each adjacency list as empty by making head as NULL \*/
38. **int** i;
39. **for** (i = 0; i < V; ++i)
40. graph->array[i].head = NULL;
41. **return** graph;
42. }
44. /\* function to add an edge to an undirected graph \*/
45. **void** addEdge(**struct** Graph\* graph, **int** src, **int** dest)
46. {
47. /\* Add an edge from src to dest. The node is added at the beginning \*/
48. **struct** AdjNode\* check = NULL;
49. **struct** AdjNode\* newNode = newAdjNode(dest);
51. **if** (graph->array[src].head == NULL) {
52. newNode->next = graph->array[src].head;
53. graph->array[src].head = newNode;
54. }
55. **else** {
57. check = graph->array[src].head;
58. **while** (check->next != NULL) {
59. check = check->next;
60. }
61. // graph->array[src].head = newNode;
62. check->next = newNode;
63. }
65. /\* Since graph is undirected, add an edge from dest to src also \*/
66. newNode = newAdjNode(src);
67. **if** (graph->array[dest].head == NULL) {
68. newNode->next = graph->array[dest].head;
69. graph->array[dest].head = newNode;
70. }
71. **else** {
72. check = graph->array[dest].head;
73. **while** (check->next != NULL) {
74. check = check->next;
75. }
76. check->next = newNode;
77. }
78. }
79. /\* function to print the adjacency list representation of graph\*/
80. **void** print(**struct** Graph\* graph)
81. {
82. **int** v;
83. **for** (v = 0; v < graph->V; ++v) {
84. **struct** AdjNode\* pCrawl = graph->array[v].head;
85. printf("\n The Adjacency list of vertex %d is: \n head ", v);
86. **while** (pCrawl) {
87. printf("-> %d", pCrawl->dest);
88. pCrawl = pCrawl->next;
89. }
90. printf("\n");
91. }
92. }
94. **int** main()
95. {
97. **int** V = 4;
98. **struct** Graph\* g = createGraph(V);
99. addEdge(g, 0, 1);
100. addEdge(g, 0, 3);
101. addEdge(g, 1, 2);
102. addEdge(g, 1, 3);
103. addEdge(g, 2, 4);
104. addEdge(g, 2, 3);
105. addEdge(g, 3, 4);
106. print(g);
107. **return** 0;
108. }

**Output:**

In the output, we will see the adjacency list representation of all the vertices of the graph. After the execution of the above code, the output will be -

