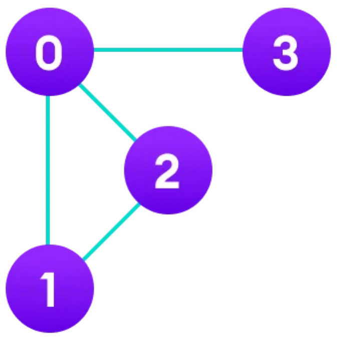
Adjacency List

In this tutorial, you will learn what an adjacency list is. Also, you will find working examples of adjacency list in C, C++, Java and Python.

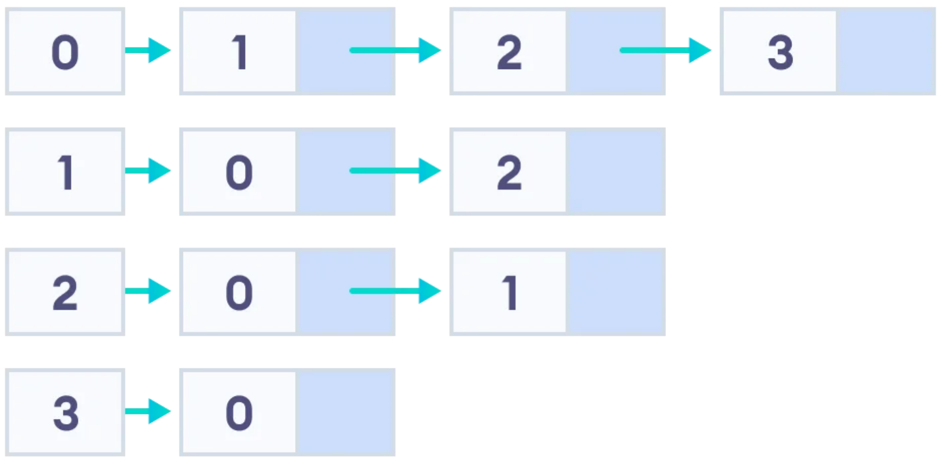
An adjacency list represents a graph as an array of linked lists. The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

For example, we have a graph below.



An undirected graph

We can represent this graph in the form of a linked list on a computer as shown below.



Linked list representation of the graph

Here, **0**, **1**, **2**, **3** are the vertices and each of them forms a linked list with all of its adjacent vertices. For instance, vertex 1 has two adjacent vertices 0 and 2. Therefore, 1 is linked with 0 and 2 in the figure above.

Pros of Adjacency List

- An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a sparse graph with millions of vertices and edges, this can mean a lot of saved space.
- It also helps to find all the vertices adjacent to a vertex easily.

Cons of Adjacency List

- Finding the adjacent list is not quicker than the adjacency matrix because all the connected nodes must be first explored to find them.

Adjacency List Structure

The simplest adjacency list needs a node data structure to store a vertex and a graph data structure to organize the nodes.

We stay close to the basic definition of a graph - a collection of vertices and edges `{V, E}`. For simplicity, we use an unlabeled graph as opposed to a labeled one i.e. the vertices are identified by their indices 0,1,2,3.

Let's dig into the data structures at play here.

```
struct node{
    int vertex;
    struct node* next;
};

struct Graph{
    int numVertices;
    struct node** adjLists;
};
```

Don't let the `struct node** adjLists` overwhelm you.

All we are saying is we want to store a pointer to `struct node*`. This is because we don't know how many vertices the graph will have and so we cannot create an array of Linked Lists at compile time.

Adjacency List C++

It is the same structure but by using the in-built list STL data structures of C++, we make the structure a bit cleaner. We are also able to abstract the details of the implementation.

```
class Graph{
    int numVertices;
    list<int> *adjLists;

  public:
    Graph(int V);
    void addEdge(int src, int dest);
};
```

Adjacency List Java

We use Java Collections to store the Array of Linked Lists.

```
class Graph{
    private int numVertices;
    private LinkedList<integer> adjLists[];
}
```

The type of LinkedList is determined by what data you want to store in it. For a labeled graph, you could store a dictionary instead of an Integer

Adjacency List Python

There is a reason Python gets so much love. A simple dictionary of vertices and its edges is a sufficient representation of a graph. You can make the vertex itself as complex as you want.

```
graph = {'A': set(['B', 'C']),
         'B': set(['A', 'D', 'E']),
         'C': set(['A', 'F']),
         'D': set(['B']),
         'E': set(['B', 'F']),
         'F': set(['C', 'E'])}
```

Adjacency List Code in Python, Java, and C/C++

Python Java C C++

```
// Adjacency List representation in C

#include <stdio.h>
#include <stdlib.h>

struct node {
  int vertex;
  struct node* next;
};
struct node* createNode(int);

struct Graph {
  int numVertices;
  struct node** adjLists;
};

// Create a node
struct node* createNode(int v) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
}

// Create a graph
struct Graph* createAGraph(int vertices) {
  struct Graph* graph = malloc(sizeof(struct Graph));
  graph->numVertices = vertices;
```

Applications of Adjacency List

- It is faster to use adjacency lists for graphs having less number of edges.