# Data Structures and Algorithms (CEN3016)

Dr. Muhammad Umair Khan
Assistant Professor
Department of Computer Engineering
National University of Technology

## Objectives Overview

- Introduction to Graph Data Structure.
- Depth First Traversal In Graph
- Breadth First Traversal in Graph

## Graphs

- More general structure than trees
  - Tree is a special type of graph

- Graphs can represent:
  - Roads
  - Airline flights from city to city
  - How the Internet is connected
- Once good representation for problem, graph algorithms can be applied

## Graph Data Structure

- 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.

## Graphs - Definitions

- **Vertex (node):**
  - Fundamental part of a graph.
  - Can have a name =>"key."
  - Additional information as "payload."

- **Edge:**
  - Connects two vertices
  - One-way or two-way
  - One-way => directed graph

## Graphs - Definitions

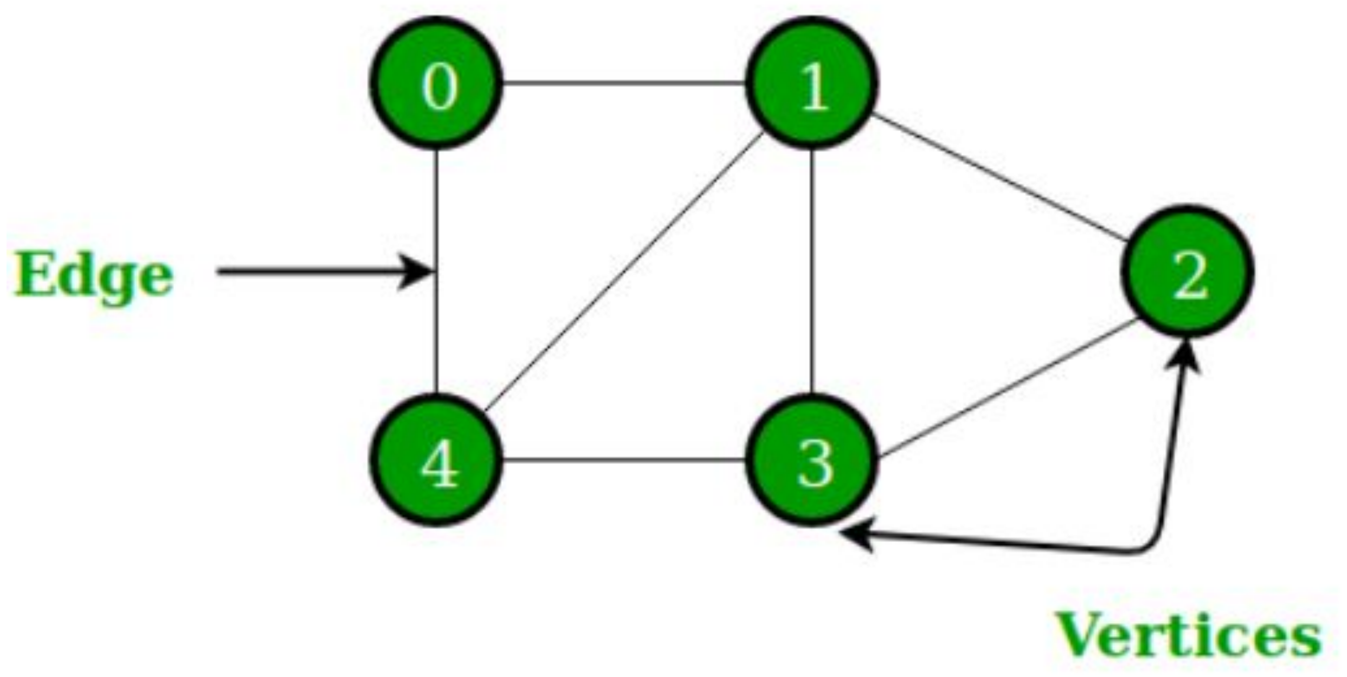
- **Weight:**
  - Edges may be weighted
  - Cost to go from on vertex to another
  - E.g. cost on edges of graph of roads might represent distance between two cities

# Formally Define Graph

- Graph can be represented by *G*, where $G=(V,E)$
- $V$ is a set of vertices, and $E$ is a set of edges
- Each edge is a set of tuples where $w, v \in V$
- Weight can be added as third component of tuple
- A subgraph is a set of edges *e* and vertices *v* such that

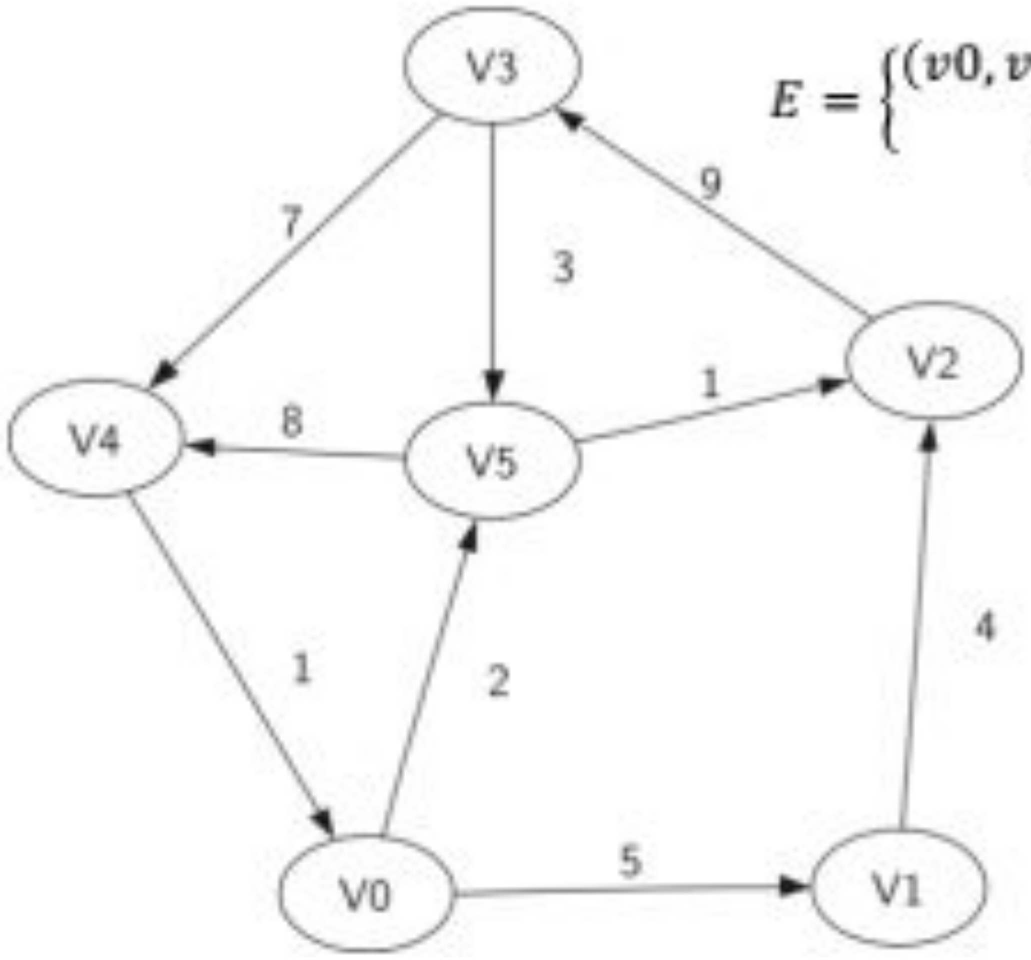
$e \subset E$ and $v \subset V$

## Graph Data Structure



**In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.**

## Graph Example

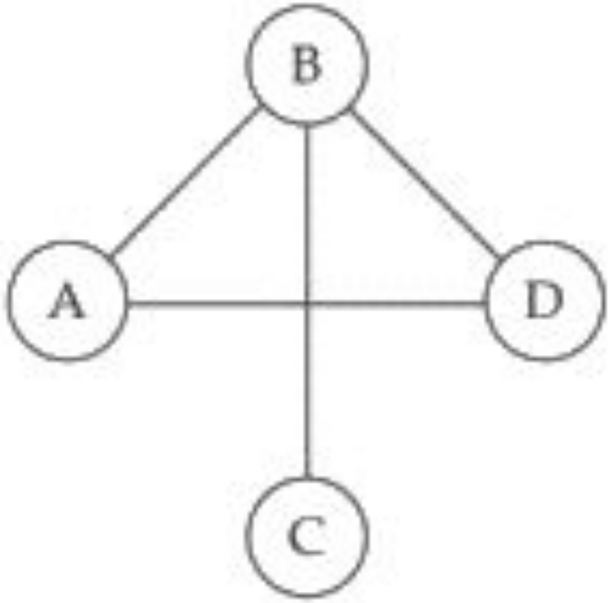


$$E = \left\{ \begin{array}{c} (v0, v1, 5), (v1, v2, 4), (v2, v3, 9), (v3, v4, 7), (v4, v0, 1) \\ (v0, v5, 2), (v5, v4, 8), (v3, v5, 3), (v5, v2, 1) \end{array} \right\}$$

$$V = \{V0, V1, V2, V3, V4, V5\}$$

## Directed vs. undirected graphs

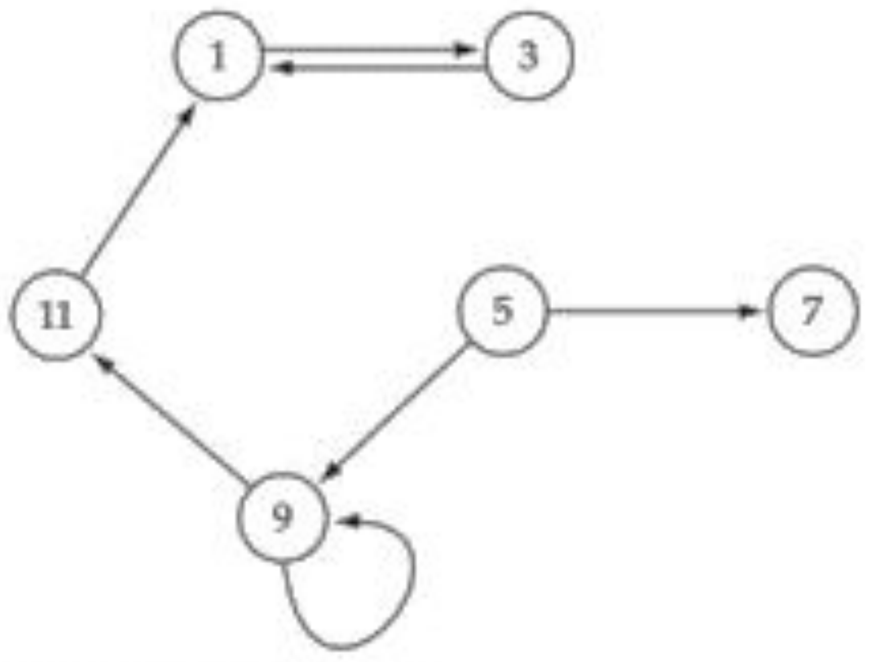
- When the edges in a graph have no direction, the graph is called *undirected*



V(Graph1) = { A, B, C, D }
E(Graph1) = { (A, B), (A, D), (B, C), (B, D) }

# Directed vs. undirected graphs

- When the edges in a graph have a direction, the graph is called *directed* (or *digraph*)

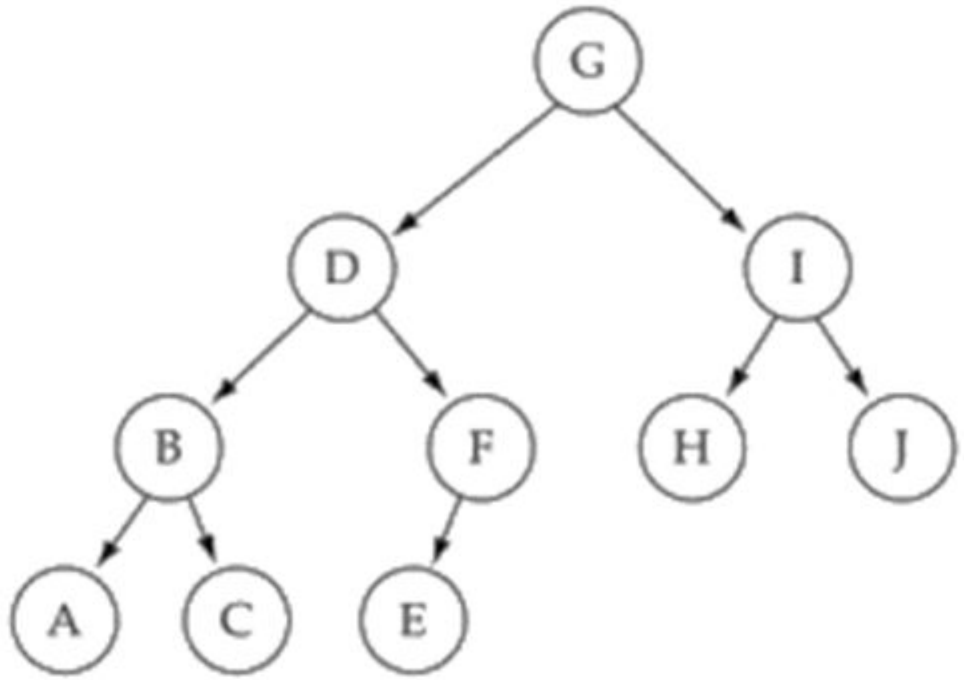


V(Graph2) = { 1, 3, 5, 7, 9, 11 }
E(Graph2) = {(1,3) (3,1) (5,9) (9,11) (5,7)), (9, 9), (11, 1) }

*Warning*: if the graph is directed, the order of the vertices in each edge is important !!

## Trees vs graphs

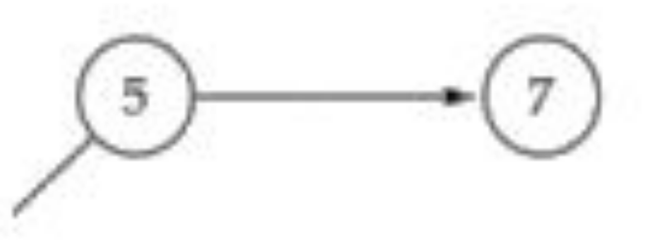
- Trees are special cases of graphs!!



V(Graph3) = { A, B, C, D, E, F, G, H, I, J }
E(Graph3) = { (G, D), (G, I), (D, B), (D, F) (I, H), (I, J), (B, A), (B, C), (F, E) }

## Graph terminology

- Adjacent nodes: two nodes are adjacent if they are connected by an edge



5 is adjacent to 7
7 is adjacent from 5

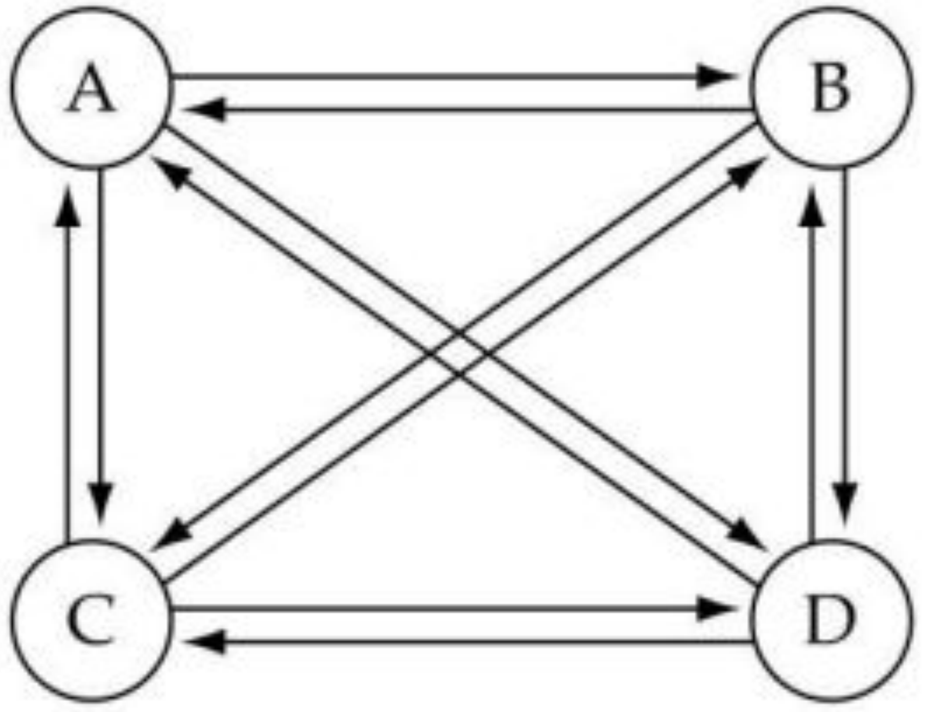
- Path: a sequence of vertices that connect two nodes in a graph
- Complete graph: a graph in which every vertex is directly connected to every other vertex

# Graph terminology

What is the number of edges in a complete directed graph with N vertices?

*N * (N-1)*

$O(N^2)$



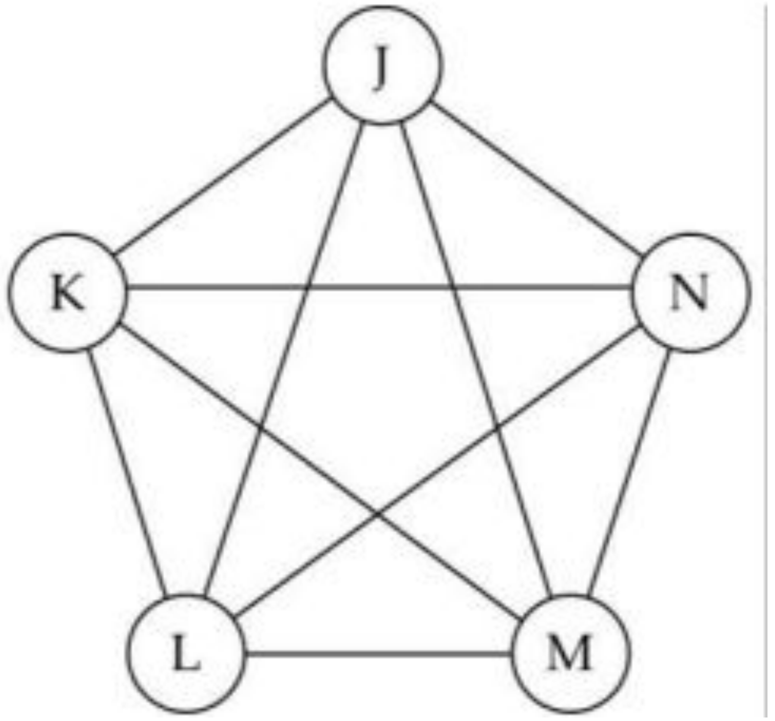
(a) Complete directed graph.

## Graph terminology

- What is the number of edges in a complete undirected graph with N vertices?

*N * (N-1) / 2*

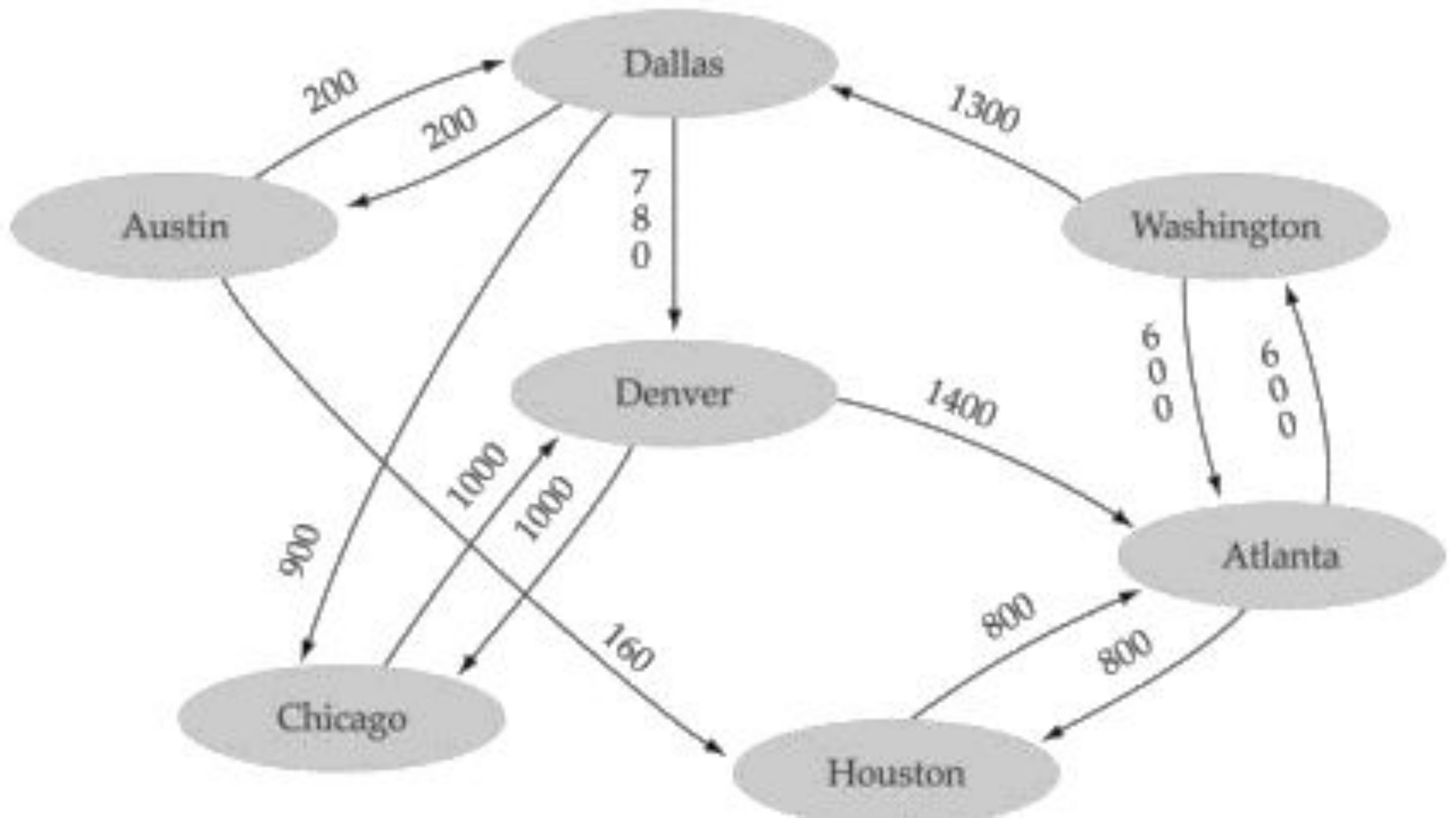
$O(N^2)$



(b) Complete undirected graph.

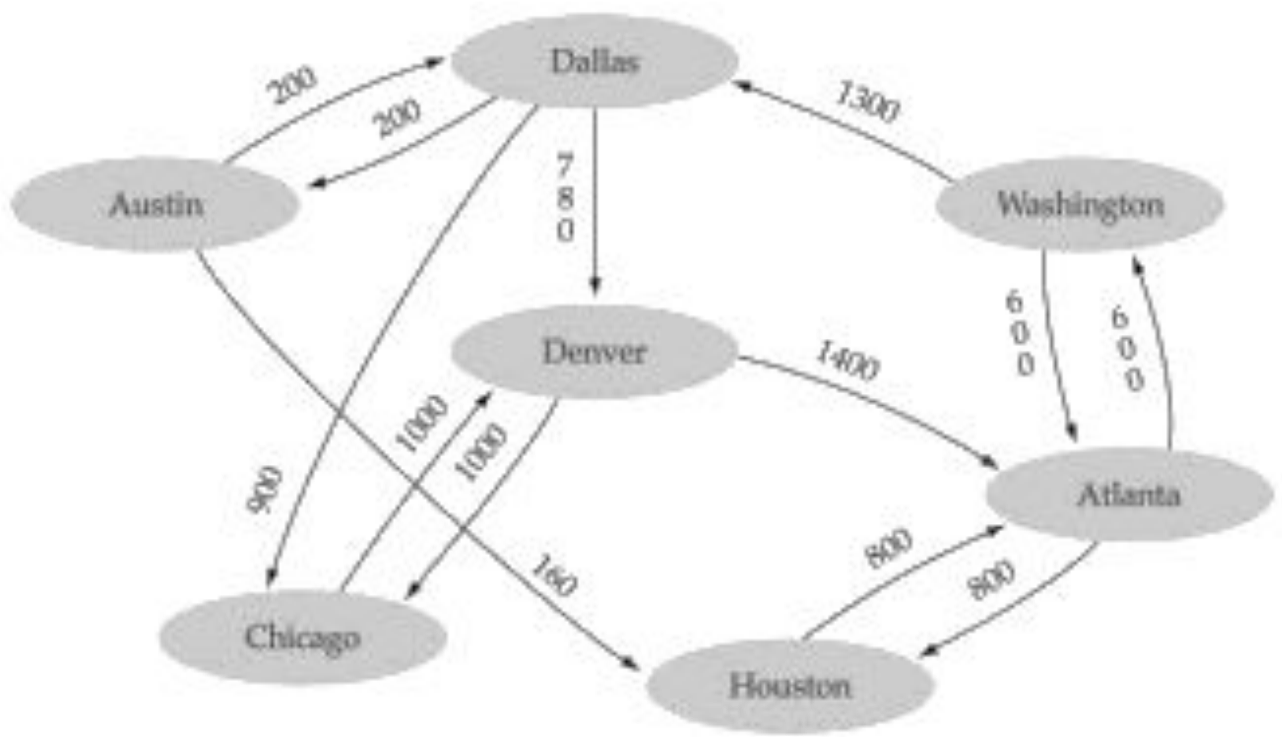
# Graph terminology

- Weighted graph: a graph in which each edge carries a value

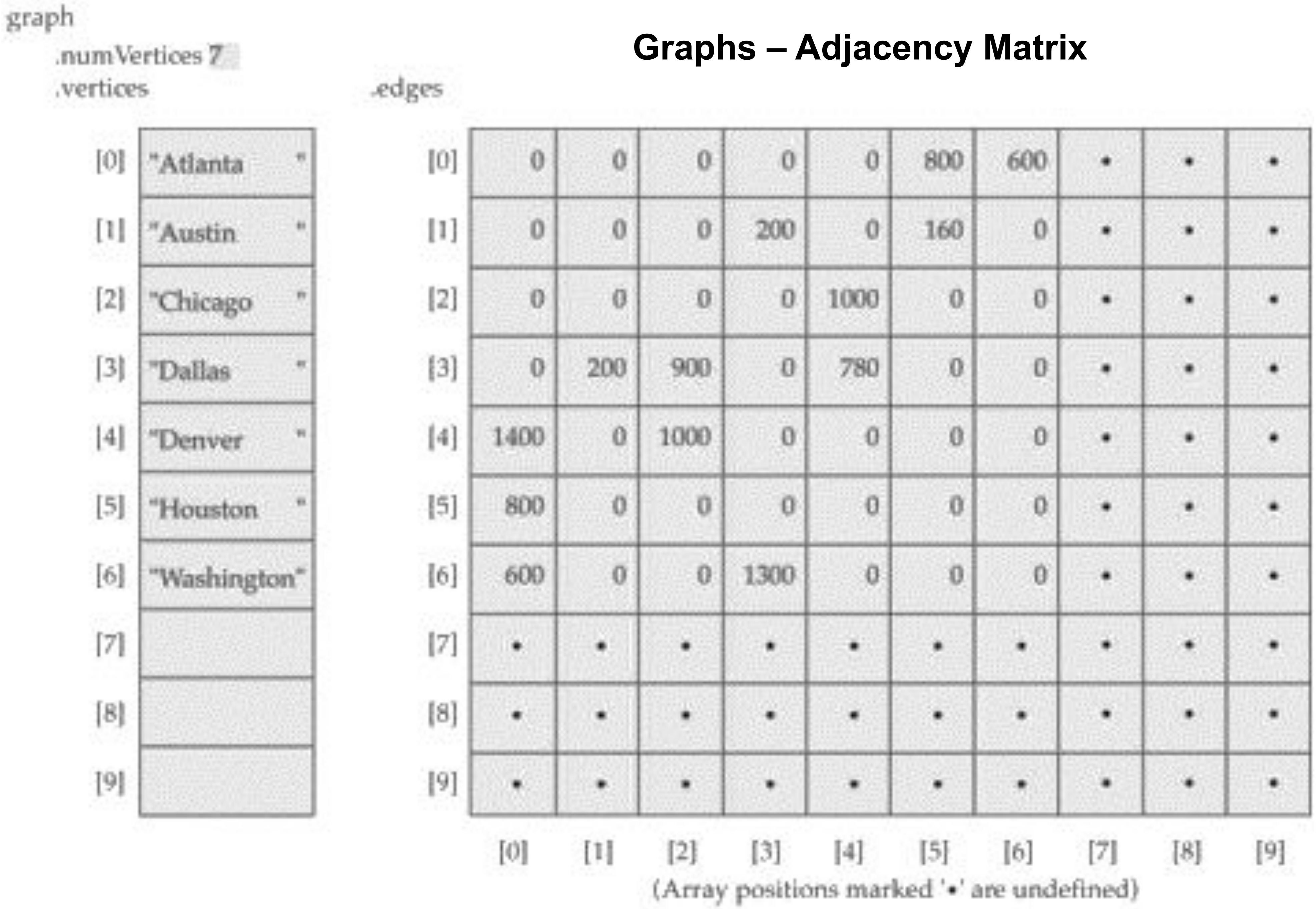


## Graph implementation

- Array-based implementation
  - A 1D array is used to represent the vertices
  - A 2D array (adjacency matrix) is used to represent the edges



## Graphs – Adjacency Matrix



graph
.numVertices 7
.vertices

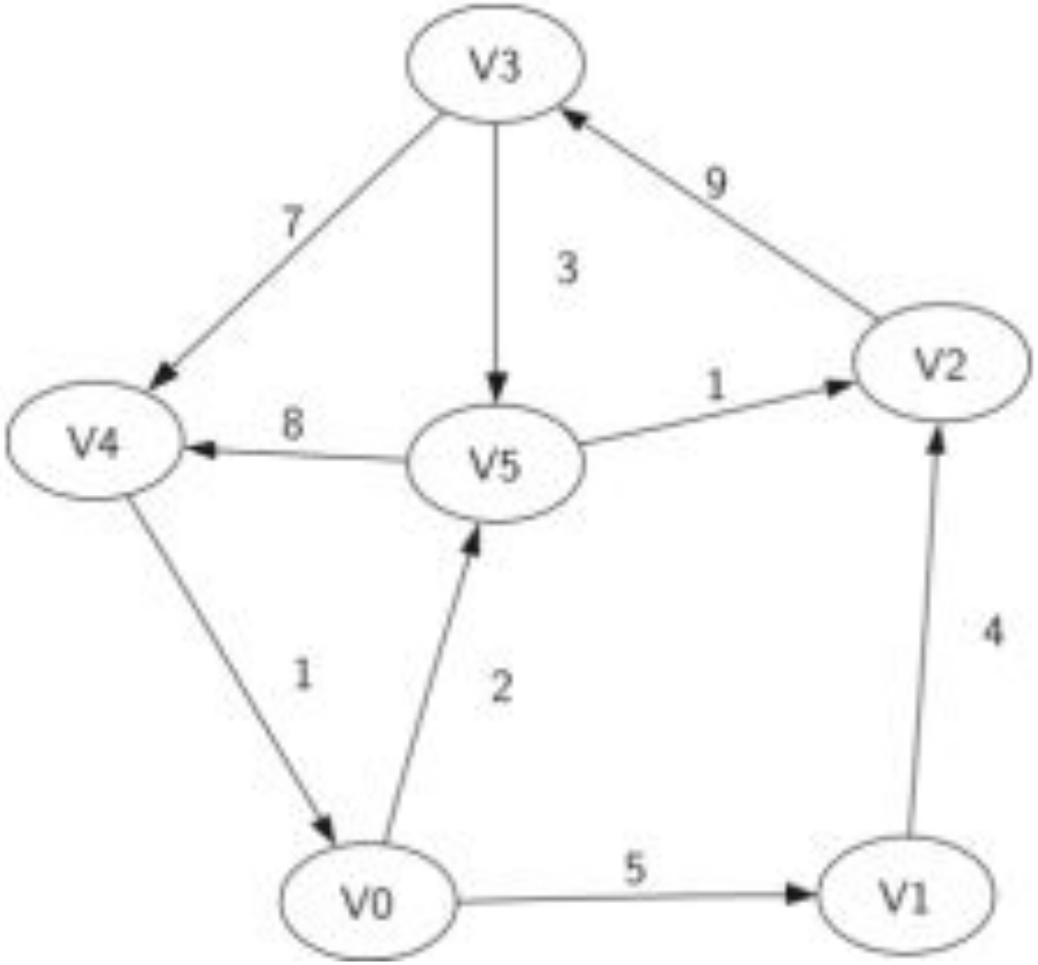
| | |
|---|---|
| [0] | "Atlanta " |
| [1] | "Austin " |
| [2] | "Chicago " |
| [3] | "Dallas " |
| [4] | "Denver " |
| [5] | "Houston " |
| [6] | "Washington" |
| [7] | |
| [8] | |
| [9] | |

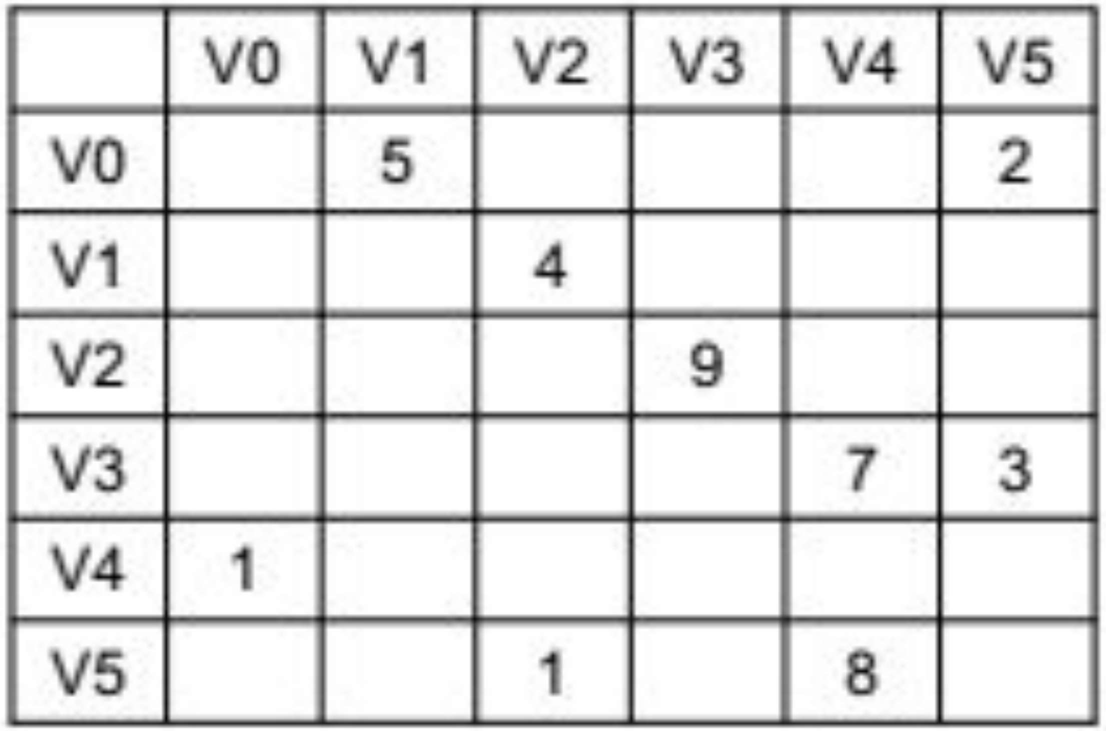
.edges

| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
|---|---|---|---|---|---|---|---|---|---|---|
| [0] | 0 | 0 | 0 | 0 | 0 | 800 | 600 | • | • | • |
| [1] | 0 | 0 | 0 | 200 | 0 | 160 | 0 | • | • | • |
| [2] | 0 | 0 | 0 | 0 | 1000 | 0 | 0 | • | • | • |
| [3] | 0 | 200 | 900 | 0 | 780 | 0 | 0 | • | • | • |
| [4] | 1400 | 0 | 1000 | 0 | 0 | 0 | 0 | • | • | • |
| [5] | 800 | 0 | 0 | 0 | 0 | 0 | 0 | • | • | • |
| [6] | 600 | 0 | 0 | 1300 | 0 | 0 | 0 | • | • | • |
| [7] | • | • | • | • | • | • | • | • | • | • |
| [8] | • | • | • | • | • | • | • | • | • | • |
| [9] | • | • | • | • | • | • | • | • | • | • |

(Array positions marked '•' are undefined)

# Graphs – Adjacency Matrix





|  | V0 | V1 | V2 | V3 | V4 | V5 |
|---|---|---|---|---|---|---|
| V0 |  | 5 |  |  |  | 2 |
| V1 |  |  | 4 |  |  |  |
| V2 |  |  |  | 9 |  |  |
| V3 |  |  |  |  | 7 | 3 |
| V4 | 1 |  |  |  |  |  |
| V5 |  |  | 1 |  | 8 |  |

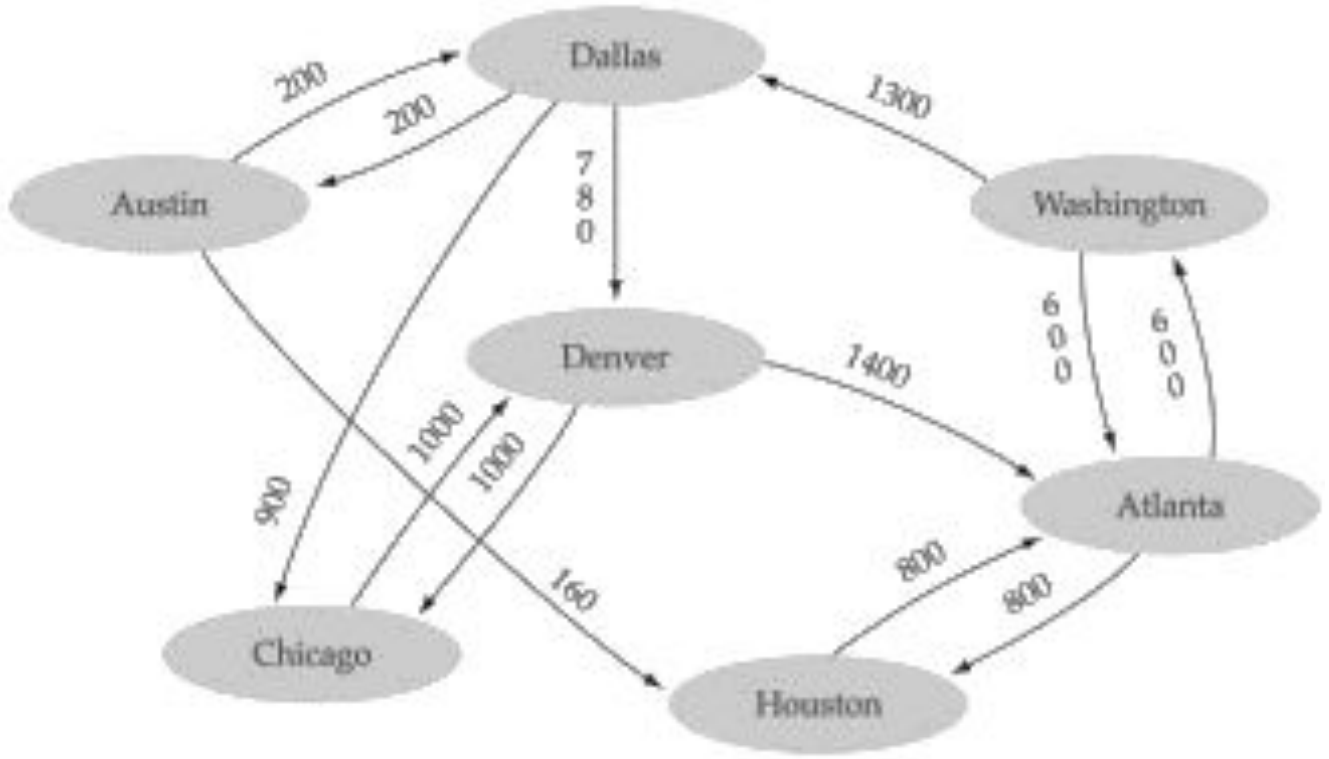
## Graphs – Adjacency Matrix

- For small graphs:
    - Easy to see which nodes are connected
    - But matrix is sparse (many empty cells), not efficient

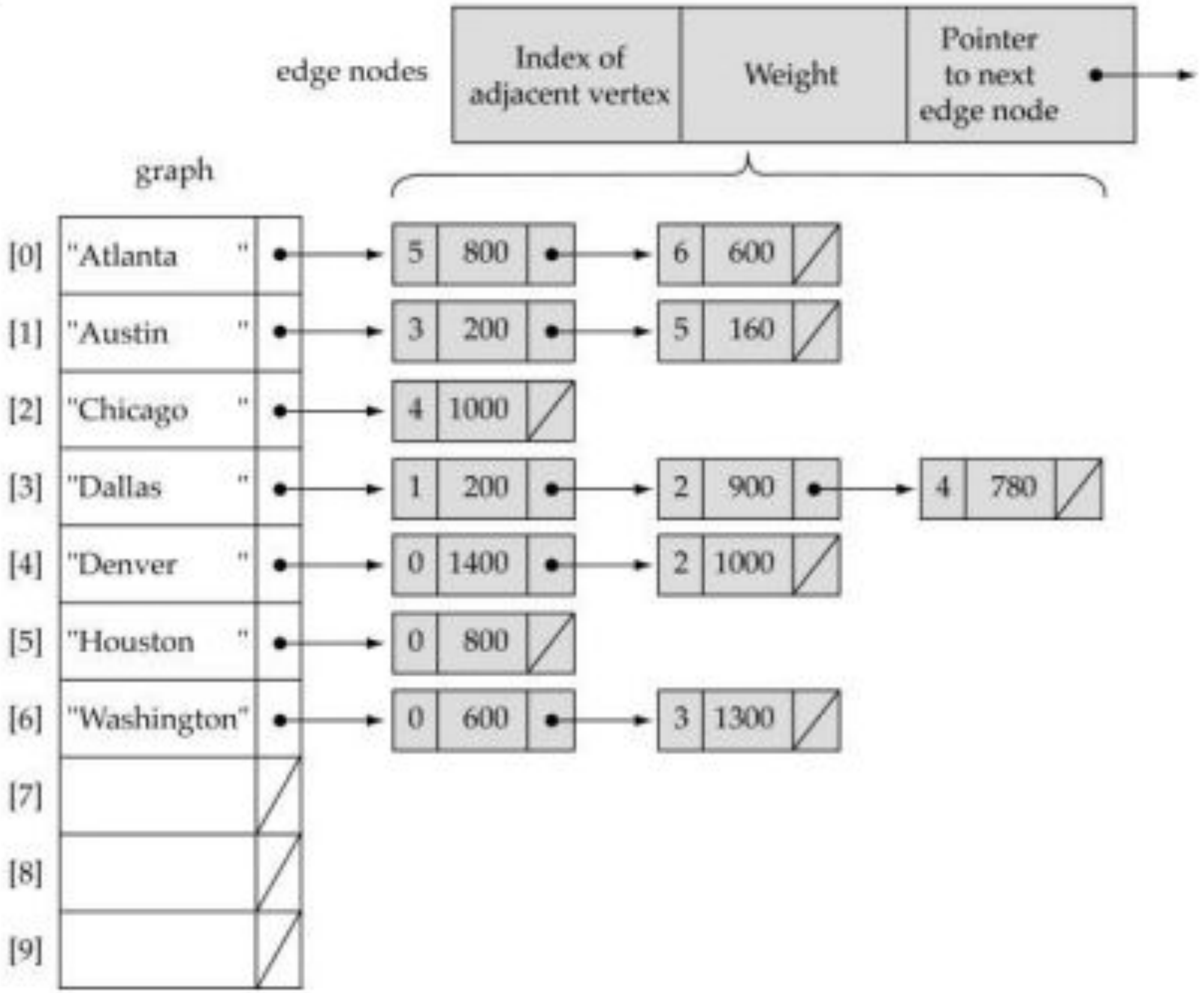
- Adjacency matrix is good fit when edges are large
    - Completely filled matrix would require to connect all edges with each other -> doesn't occur often

# Graph implementation

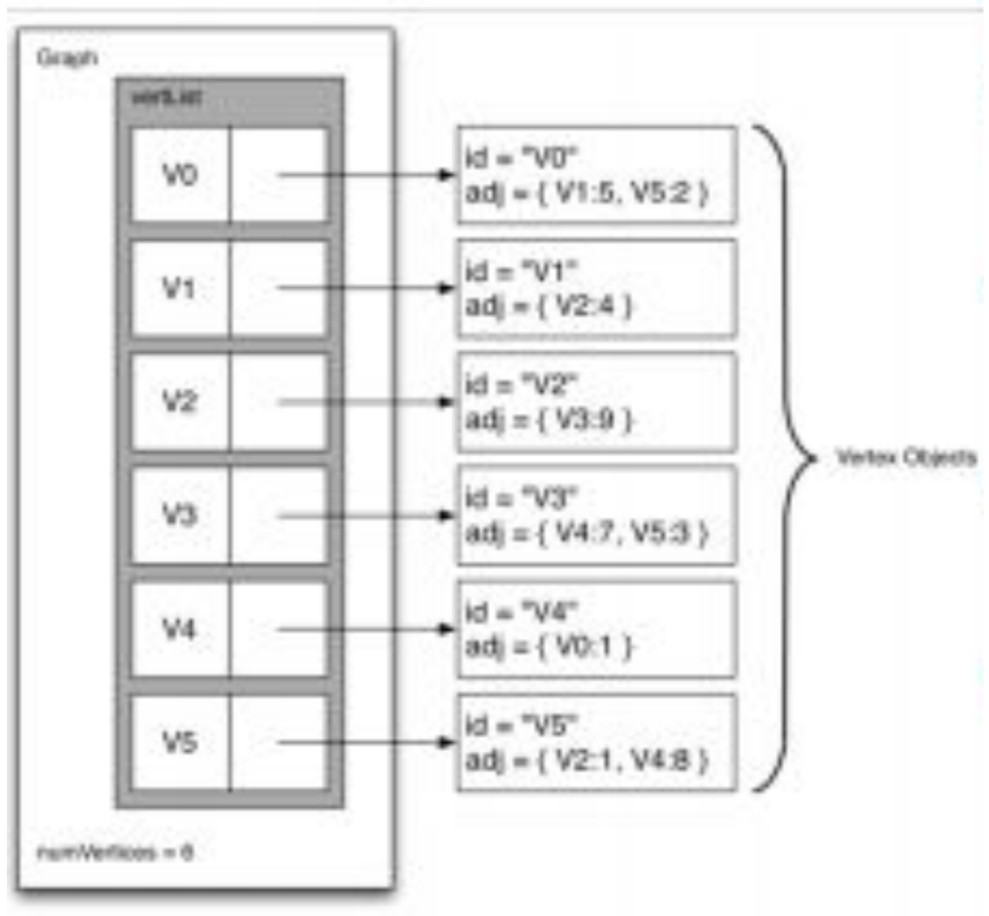
- Linked-list implementation
  - A 1D array is used to represent the vertices
  - A list is used for each vertex *v* which contains the vertices which are adjacent from *v* (adjacency list)



(a)


edge nodes
Index of adjacent vertex
Weight
Pointer to next edge node
graph
[0] "Atlanta "
[1] "Austin "
[2] "Chicago "
[3] "Dallas "
[4] "Denver "
[5] "Houston "
[6] "Washington"
[7]
[8]
[9]
5 800
6 600
3 200
5 160
4 1000
1 200
2 900
4 780
0 1400
2 1000
0 800
0 600
3 1300

# Adjacency List



- Master list of all vertices
- Each vertex maintains a list of other vertices it is connected to
- Keys are vertices and values are weights
- Benefits:
- Compactly represent sparse matrix
- Easily find links between vertices
- Efficient to find the weight between any two vertices
- Allows for quick addition and removal of vertices and edges
- Can be easily serialized for storage or transmission

## Graph searching

- Problem: find a path between two nodes of the graph (e.g., Austin and Washington)
- Methods: Depth-First-Search (DFS) or BreadthFirst-Search (BFS)

## Depth-First-Search (DFS)

- What is the idea behind DFS?
  - Travel as far as you can down a path
  - Back up as little as possible when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)
- DFS can be implemented efficiently using a stack.

Depth First Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.

# Depth First Search (DFS)

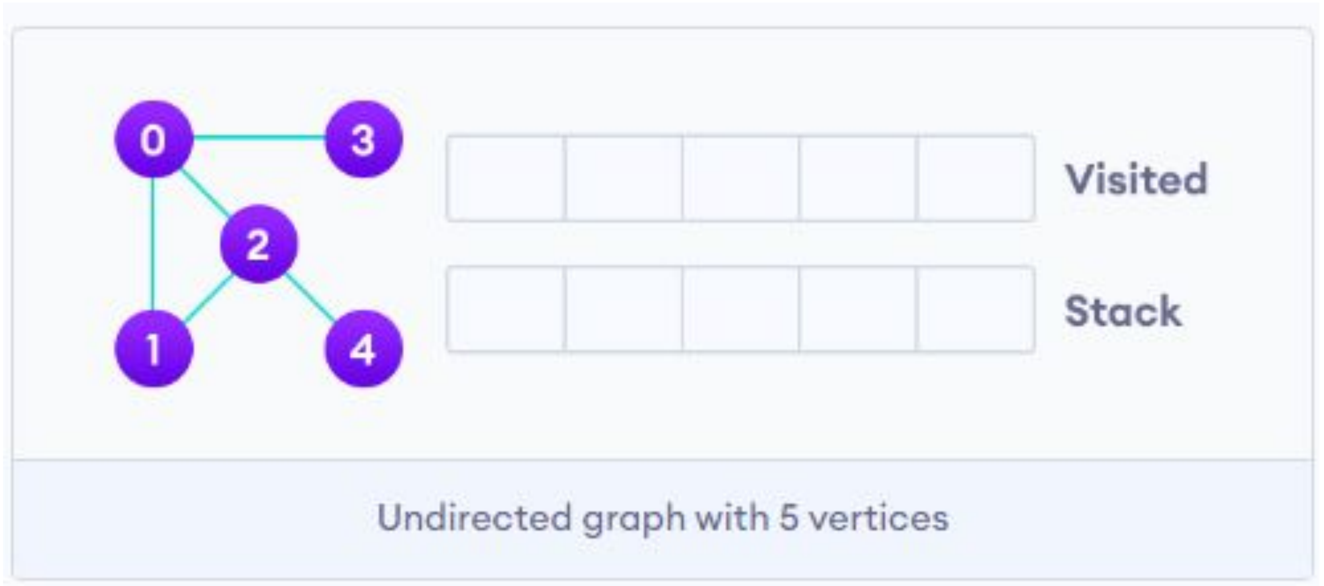
A standard DFS implementation puts each vertex of the graph into one of two categories:

- Visited
- Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.
The DFS algorithm works as follows:
Start by putting any one of the graph's vertices on top of a stack.
Take the top item of the stack and add it to the visited list.
Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
Keep repeating steps 2 and 3 until the stack is empty.

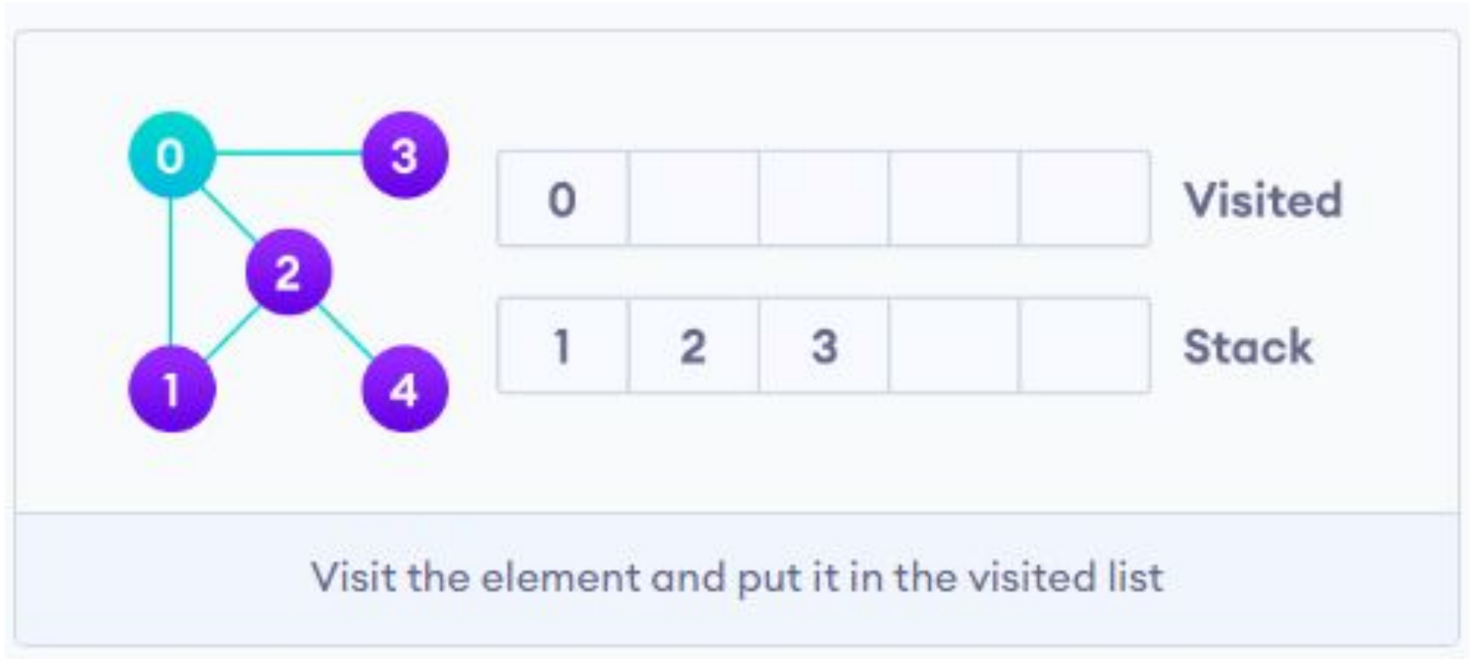
## Depth First Search Example

Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



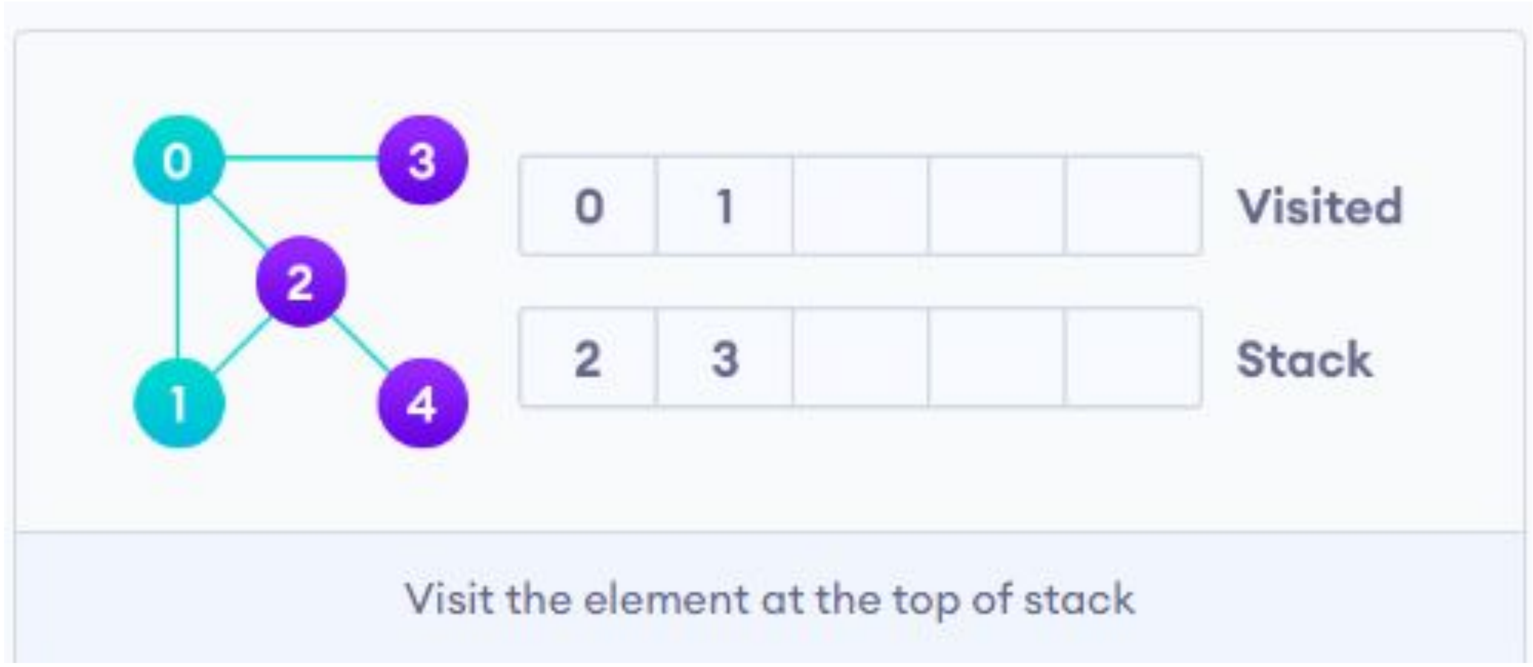
# Depth First Search Example

We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



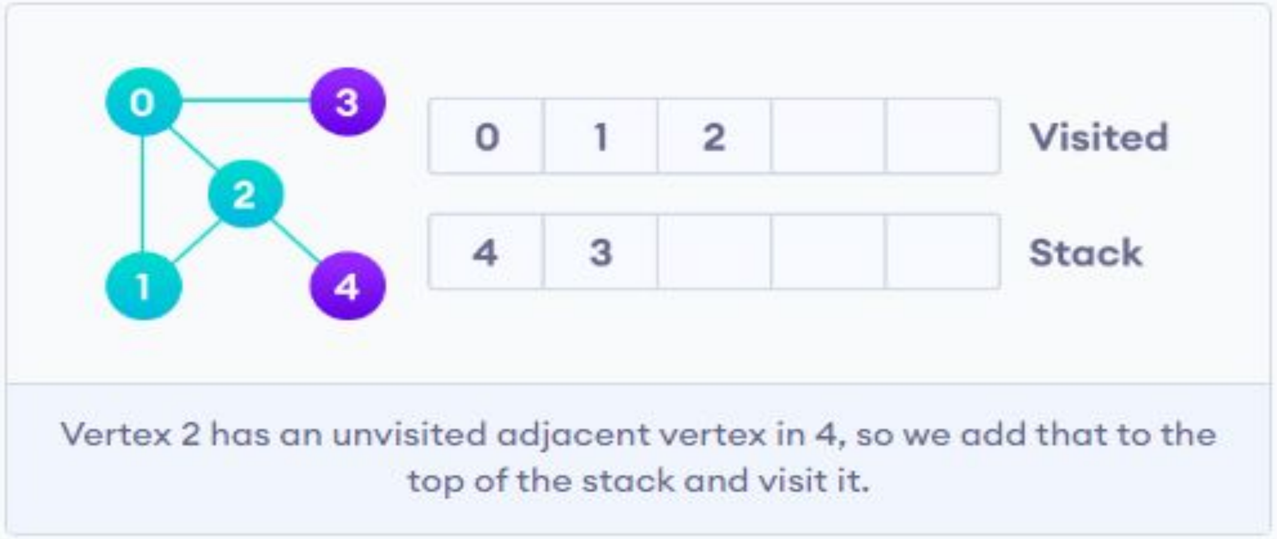
# Depth First Search Example

Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



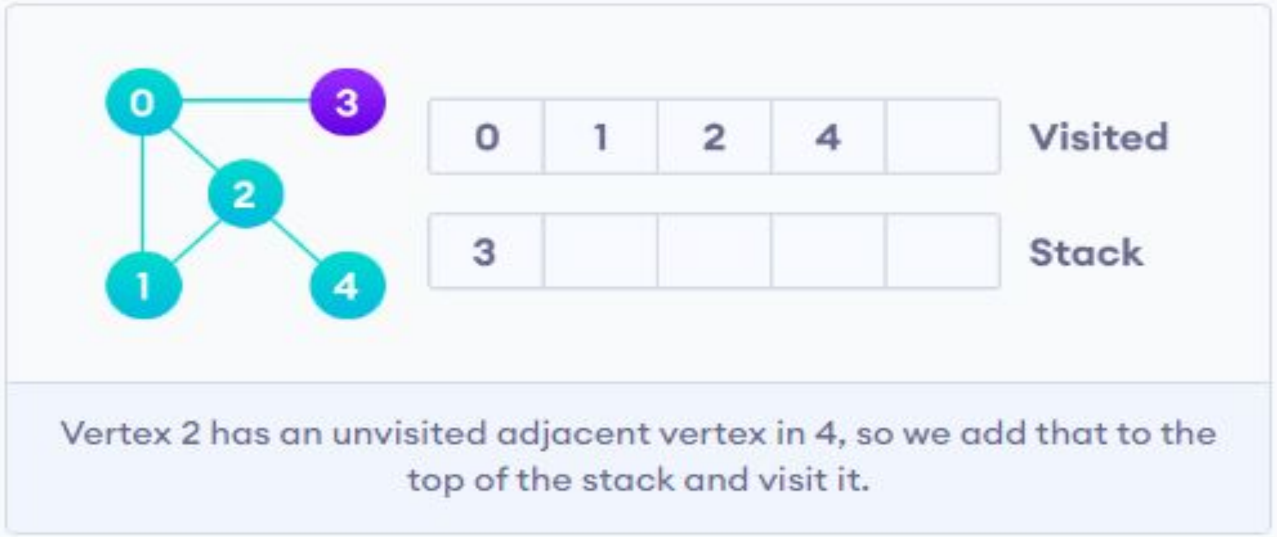
# Depth First Search Example

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



|  |  |  |  |  |  |
|---|---|---|---|---|---|
| 0 | 1 | 2 |  |  | Visited |
| 4 | 3 |  |  |  | Stack |

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

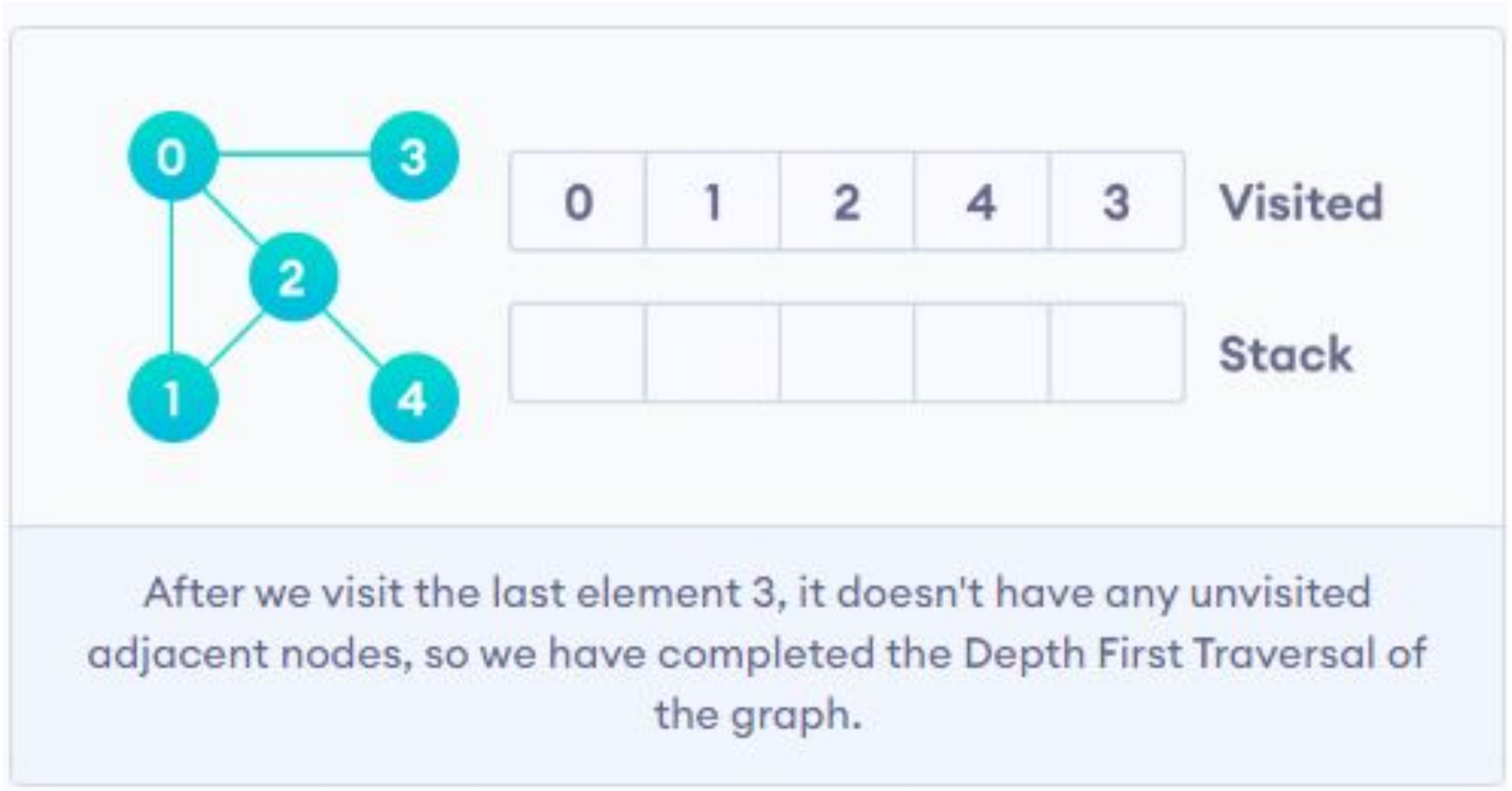


|  |  |  |  |  |  |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 4 |  | Visited |
| 3 |  |  |  |  | Stack |

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

# Depth First Search Example

After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 4 | 3 | Visited |
| | | | | | Stack |

After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

## Depth-First-Search (DFS)

```
1. Set found to false
2. stack.Push (startVertex)
3. do
4. stack.Pop (vertex)
5. if vertex == endVertex
6. Set found to true
7. else
8. Push all adjacent vertices onto stack
9. while !stack.IsEmpty() AND !found
10. if(!found)
11. Write "Path does not exist"
```

# Breadth-First-Searching (BFS)

- What is the idea behind BFS?
  - Look at all possible paths at the same depth before you go at a deeper level
  - Back up *as far as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)

Traversal means visiting all the nodes of a graph. Breadth First Traversal or Breadth First Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.

## Implementing Breadth First Search

• Breadth First Search (BFS) is one of the easiest algorithms to search a graph.
• Given a graph G and starting vertex s BFS explores edges in the graph to find all vertices for which there is a path from s.
• Note: BFS finds all vertices at distance k from s, before any vertices at distance k+1.
• To visualize BFS, imagine that it is building one level at a time.
• BFS adds all children of the starting vertex.
•Then it begins to discover any of the grandchildren.

# BFS algorithm

A standard BFS implementation puts each vertex of the graph into one of two categories:

Visited

Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

Start by putting any one of the graph's vertices at the back of a queue.

Take the front item of the queue and add it to the visited list.

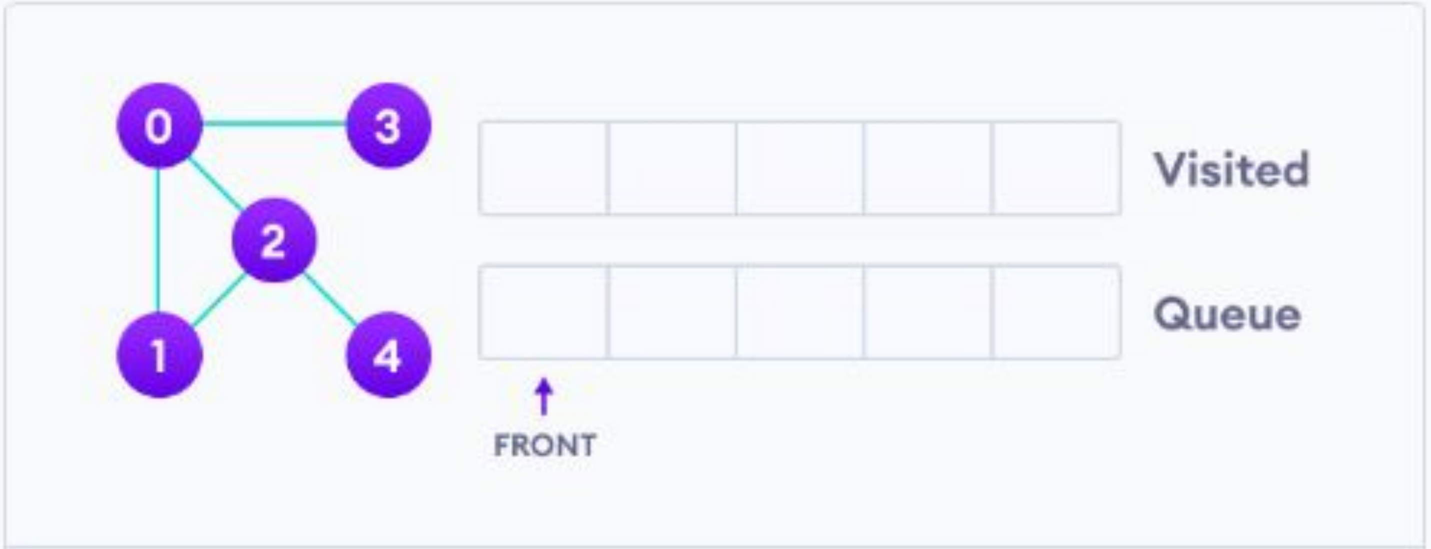
Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.

Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

## BFS example

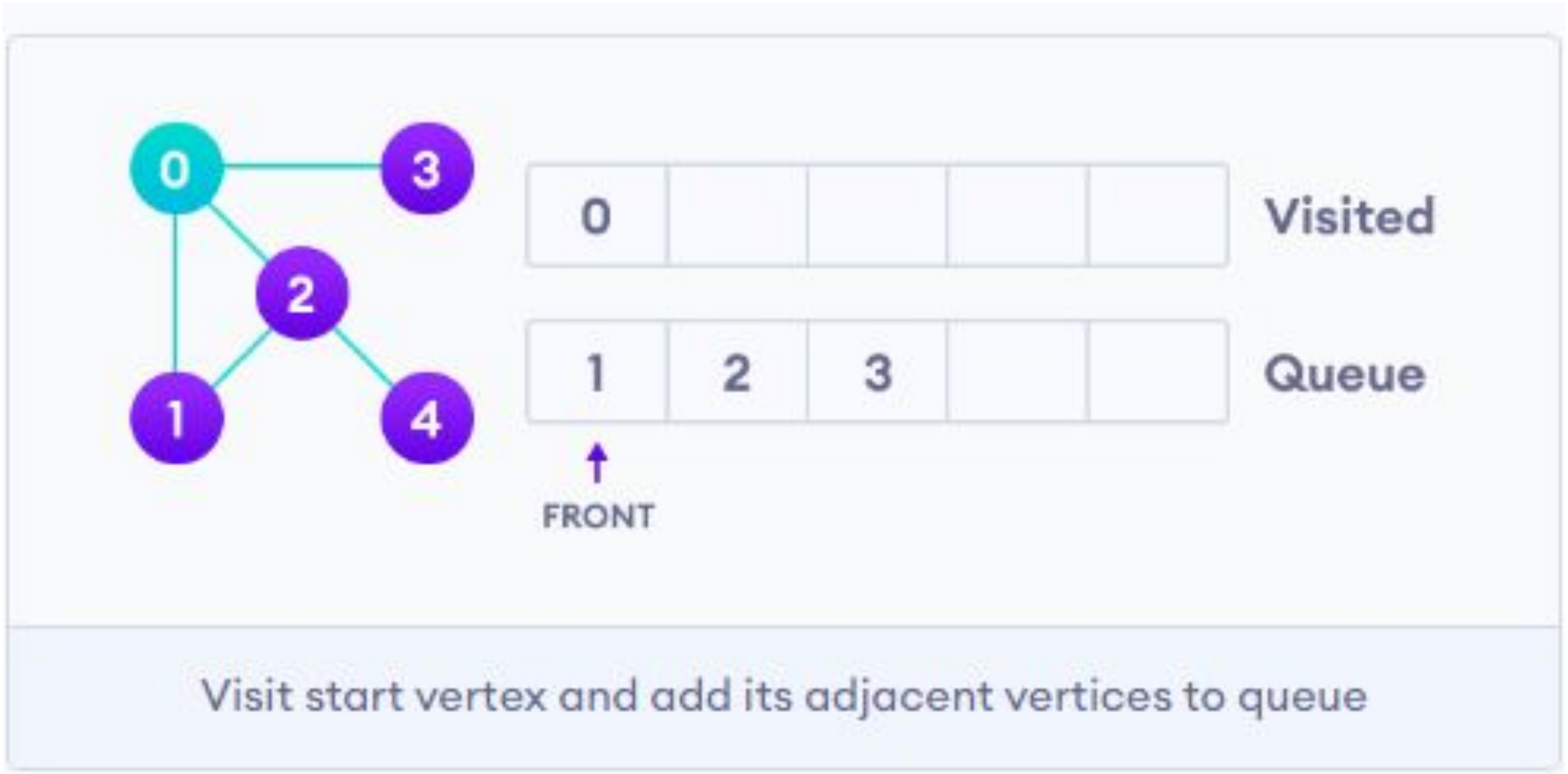
Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



Undirected graph with 5 vertices

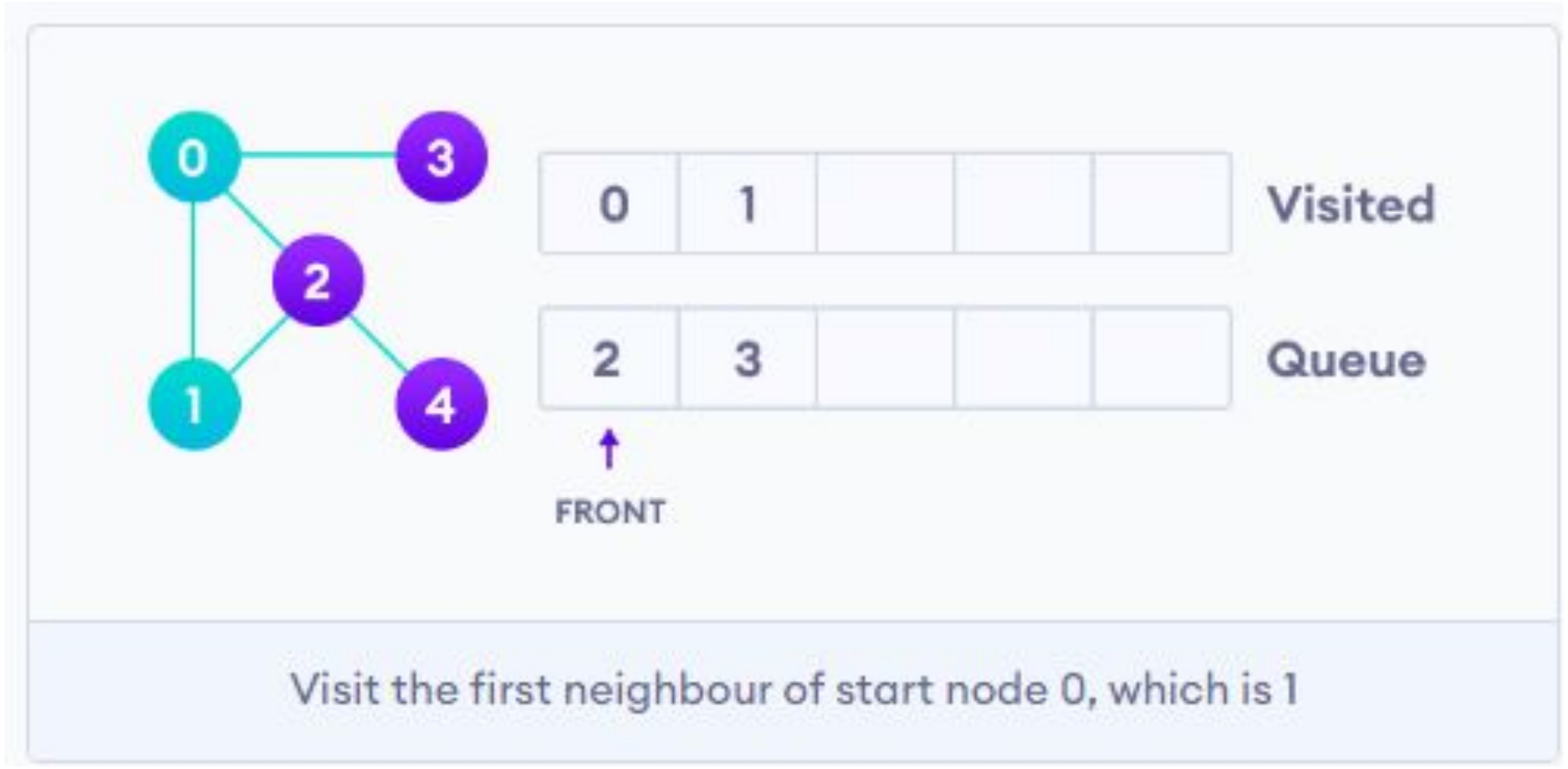
# BFS example

We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



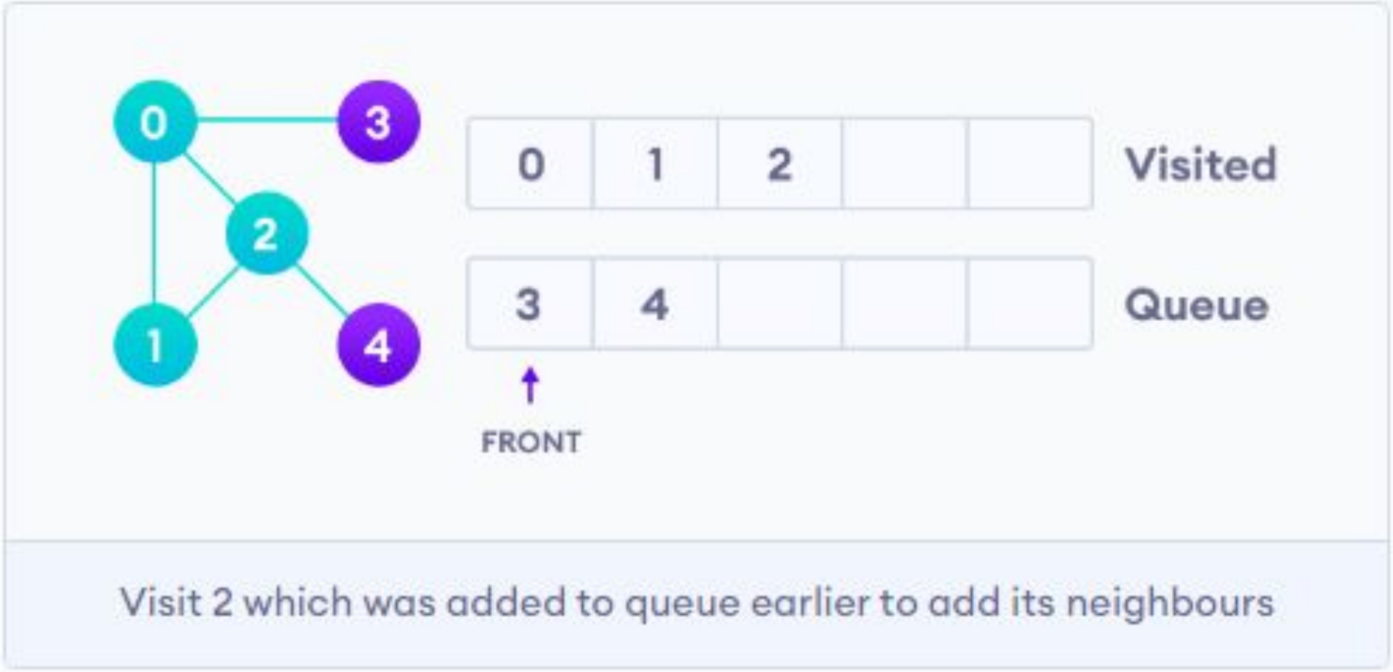
# BFS example

Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

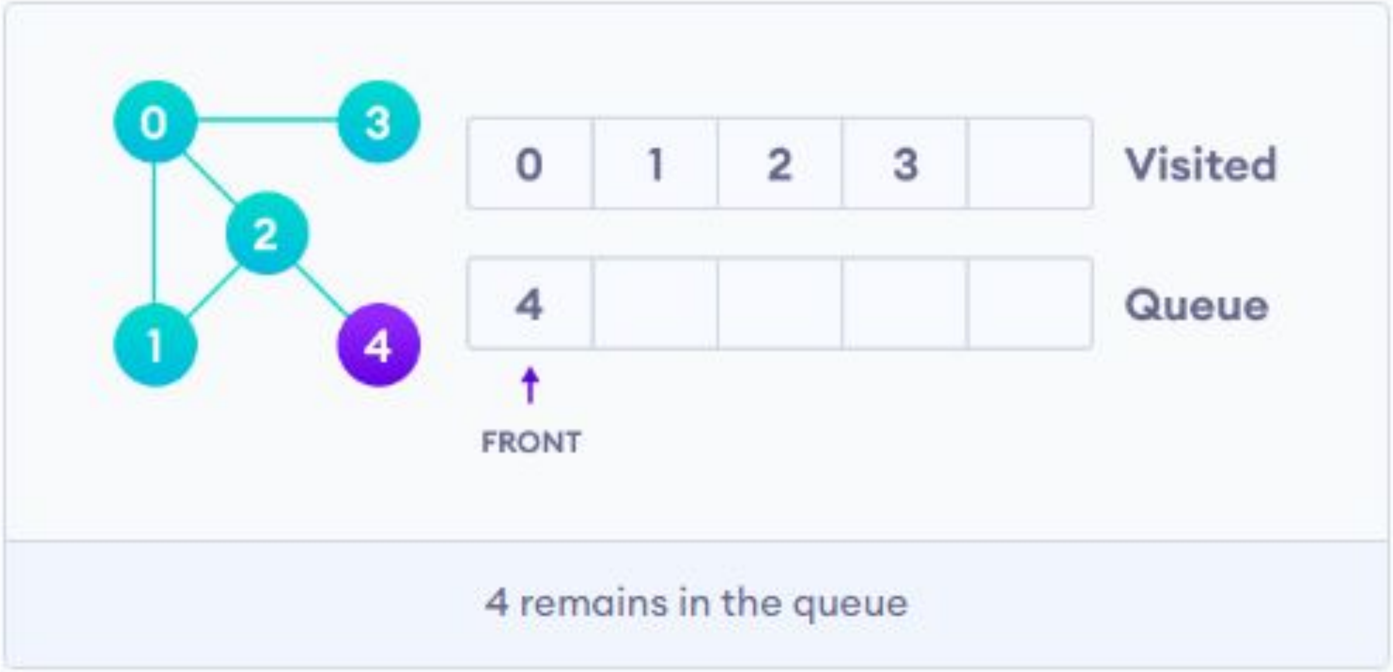


# BFS example

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.



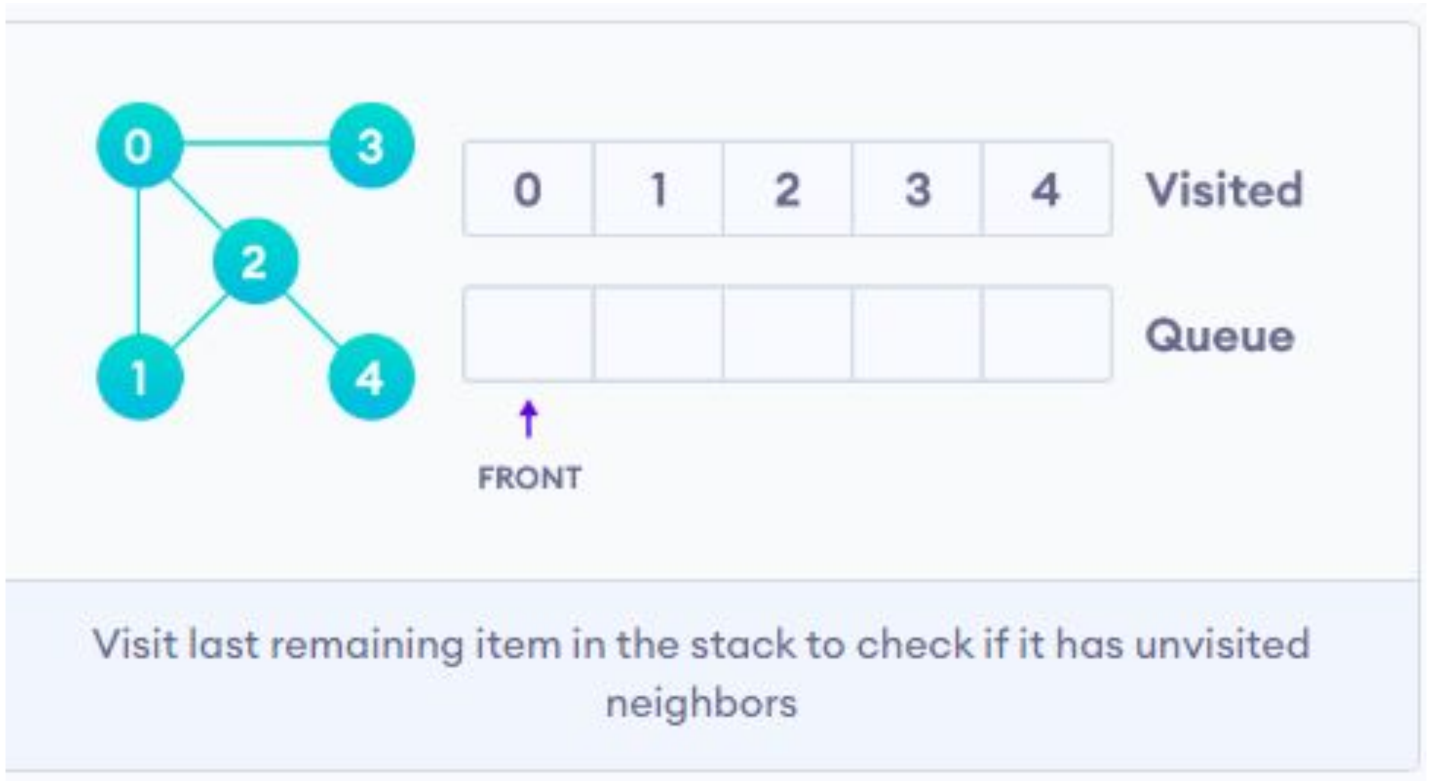
Visit 2 which was added to queue earlier to add its neighbours



4 remains in the queue

## BFS example

Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited. We visit it.



Visit last remaining item in the stack to check if it has unvisited neighbors

Since the queue is empty, we have completed the Breadth First Traversal of the graph.

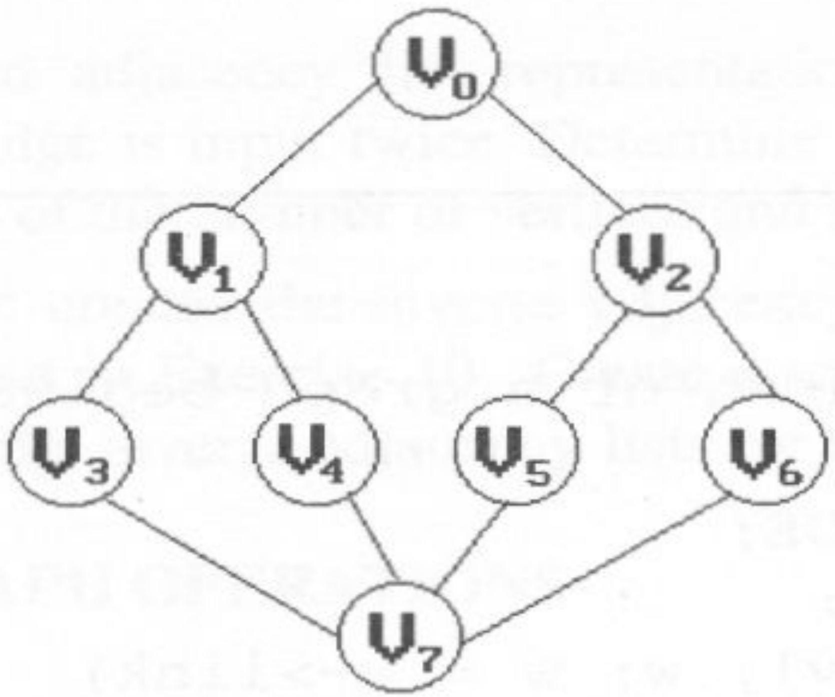
# Breadth-First-Searching (BFS)

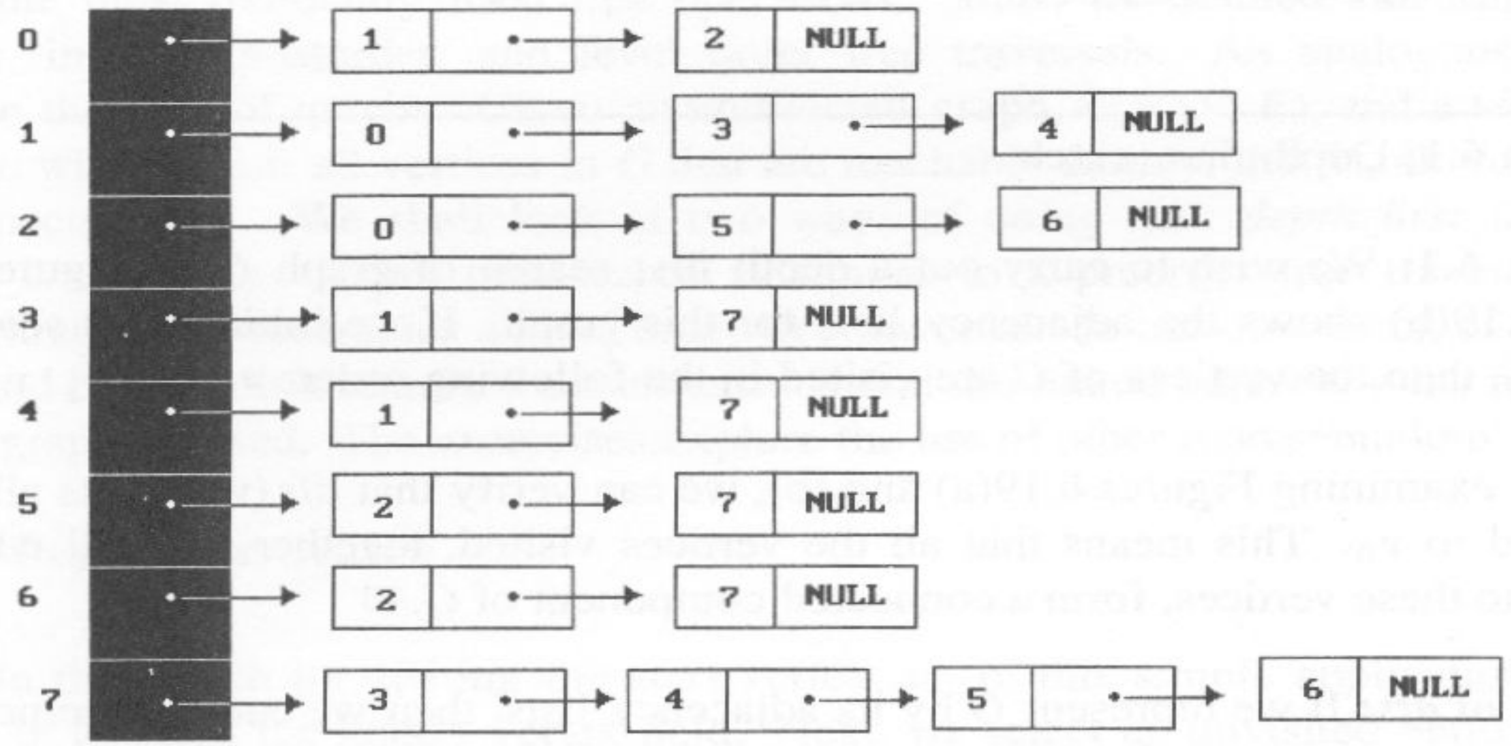
- BFS can be implemented efficiently using a *queue*

```
Set found to false
queue.Enqueue(startVertex)
do
    queue.Dequeue(vertex)
    if vertex == endVertex
        Set found to true
    else
        Enqueue all adjacent vertices onto queue
while !queue.IsEmpty() AND !found
```

- Should we mark a vertex when it is enqueued or when it is dequeued?

**depth first search:** v0, v1, v3, v7, v4, v5, v2, v6





(b)

**breadth first search:** v0, v1, v2, v3, v4, v5, v6, v7

## Single-source shortest-path problem

- There are multiple paths from a source vertex to a destination vertex
- Shortest path: the path whose total weight (i.e., sum of edge weights) is minimum
- Examples:
  - Austin->Houston->Atlanta->Washington: 1560 miles
  - Austin->Dallas->Denver->Atlanta->Washington: 2980 miles

## Single-source shortest-path problem

- Common algorithms: Dijkstra's algorithm, Bellman-Ford algorithm
- BFS can be used to solve the shortest graph problem when the graph is weightless or all the weights are the same (mark vertices before enqueue)