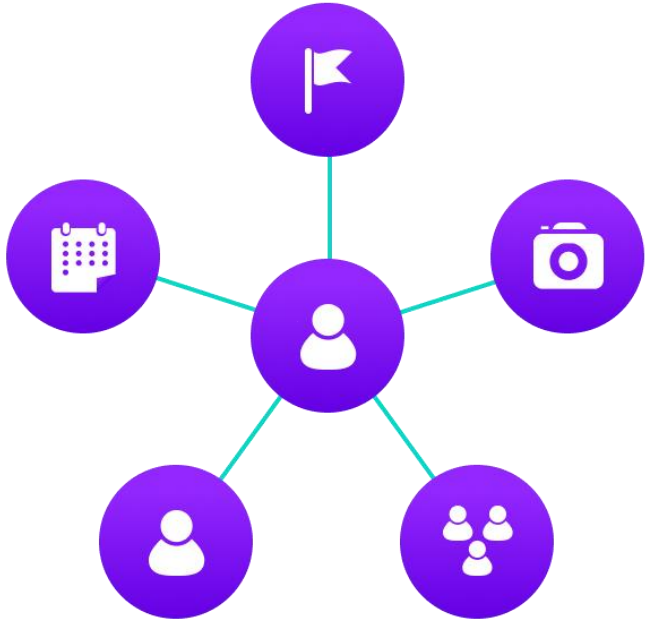
Data Structures and Algorithm



Moazzam Ali Sahi

Lecture # 25-26

Graphs

Last Lecture

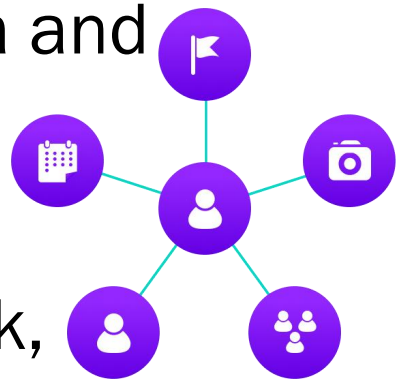
- AVL Trees
- BST to AVL
- AVL Tree Creation
- Delete a Node in AVL Tree

This Lecture

- Learn about graphs
- Discover how to represent graphs in computer memory
- Examine and implement various graph traversal algorithms
- Learn how to implement a shortest path algorithm
- Examine and implement the minimum spanning tree algorithm

WHAT IS Graph DATA STRUCTURE?

- A graph data structure is a collection of nodes that have data and are connected to other nodes.



- Let's try to understand this through an example. On Facebook, everything is a node. That includes User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link, Note...anything that has data is a node.

- Every relationship is an edge from one node to another. Whether you post a photo, join a group, like a page, etc., a new edge is created for that relationship.

Intuition Behind Graphs

- The nodes represent entities (such as people, cities, computers, words, etc.)
- Edges (x,y) represent relationships between entities x and y, such as:
  - “x loves y”
  - “x hates y”
  - “x is a friend of y” (note that this not necessarily reciprocal)
  - “x considers y a friend”
  - “x is a child of y”
  - “x is a half-sibling of y”
  - “x is a full-sibling of y”
- In those examples, each relationship is a different graph

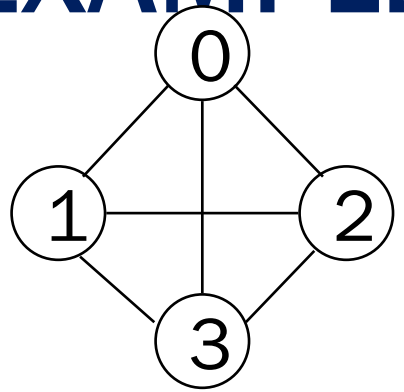
MORE EXAMPLES

- Web pages with links
- Facebook friends
- "Input data" for the Kevin Bacon game
- Methods in a program that call each other
- Road maps
- Airline routes
- Family trees
- Course pre-requisites

DEFINITION

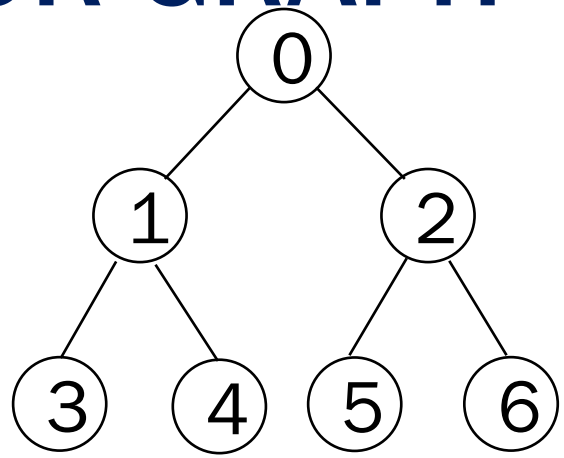
- A graph G consists of two sets
  - a finite, nonempty set of vertices V(G)
  - a finite, possible empty set of edges E(G)
  - G(V,E) represents a graph
- An **undirected graph** is one in which the pair of vertices in a edge is unordered, (v0, v1) = (v1,v0)
- A **directed graph** is one in which each edge is a directed pair of vertices, <v0, v1> != <v1,v0>

EXAMPLES FOR GRAPH



G_1

complete graph



G_2

G_3

incomplete graph

$V(G_1)=\{0,1,2,3\}$
$V(G_2)=\{0,1,2,3,4,5,6\}$
$V(G_3)=\{0,1,2\}$

$E(G_1)=\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$
$E(G_2)=\{(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)\}$
$E(G_3)=\{<0,1>,<1,0>,<1,2>\}$

complete undirected graph: n(n-1)/2 edges
complete directed graph: n(n-1) edges

COMPLETE GRAPH

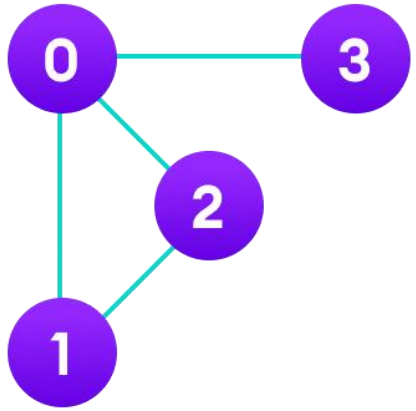
- A complete graph is a graph that has the maximum number of edges
  - for undirected graph with n vertices, the maximum number of edges is n(n-1)/2
  - for directed graph with n vertices, the maximum number of edges is n(n-1)
  - example: G1 is a complete graph

ADJACENT AND INCIDENT

- If (v0, v1) is an edge in an undirected graph,
  - v0 and v1 are **adjacent**
  - The edge (v0, v1) is **incident** on vertices v0 and v1
- If <v0, v1> is an edge in a directed graph
  - v0 is adjacent to v1, and v1 is adjacent from v0
  - The edge <v0, v1> is incident on v0 and v1

Vertices and Edges

- All of Facebook is then a collection of these nodes and edges. This is because Facebook uses a graph data structure to store its data.
- More precisely, a graph is a data structure (V, E) that consists of
  - A collection of vertices V
  - A collection of edges E, represented as ordered pairs of vertices (u,v)



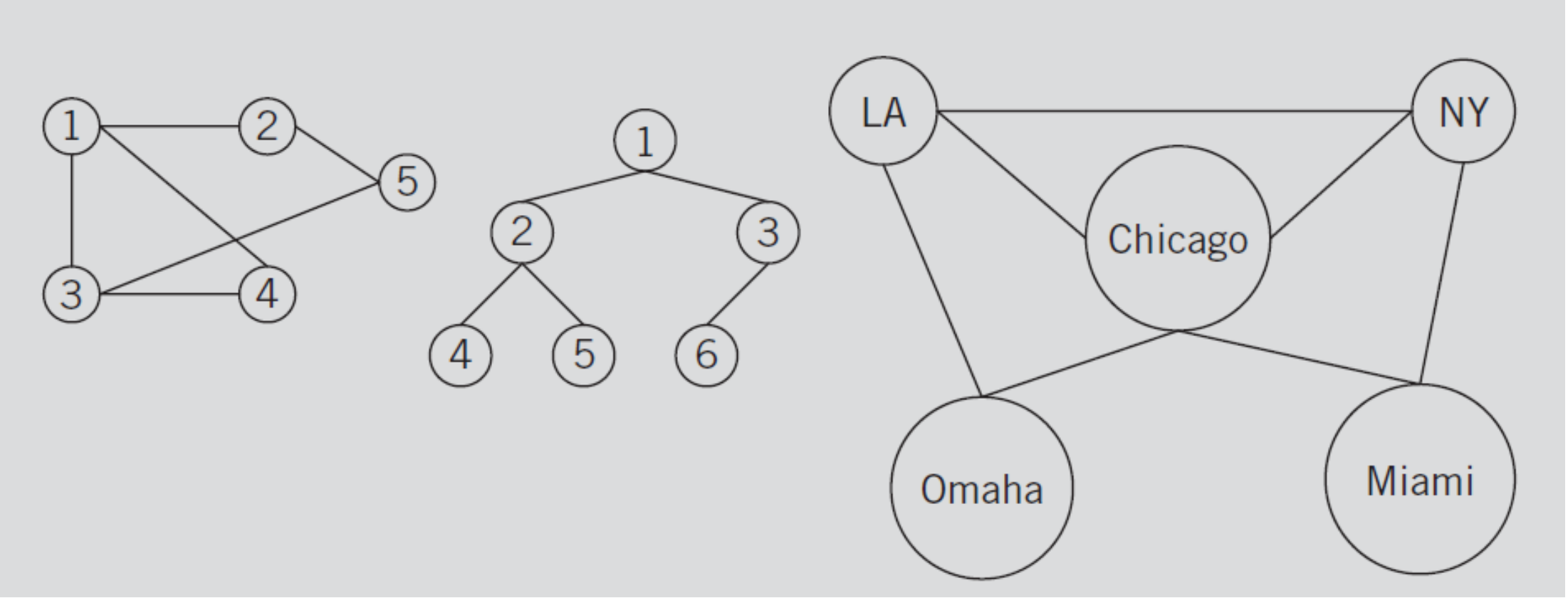

```
V = {0, 1, 2, 3}
E = {(0,1), (0,2), (0,3), (1,2)}
G = {V, E}
```

Graph Terminology

- **Adjacency**: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.
- **Path**: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.
- **Directed Graph:** A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.

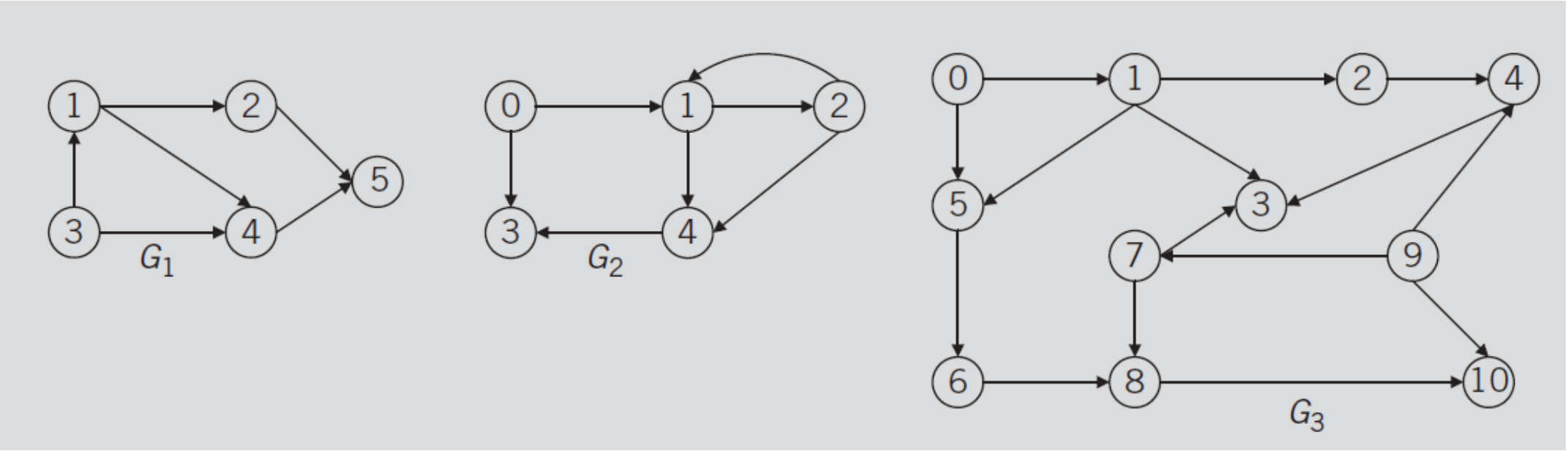
TYPES OF GRAPHS

- Undirected Graphs



TYPES OF GRAPHS

- Directed Graphs

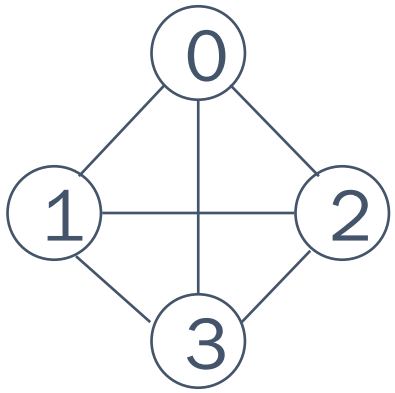


$V(G_1) = \{1, 2, 3, 4, 5\}$
$V(G_2) = \{0, 1, 2, 3, 4\}$
$V(G_3) = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

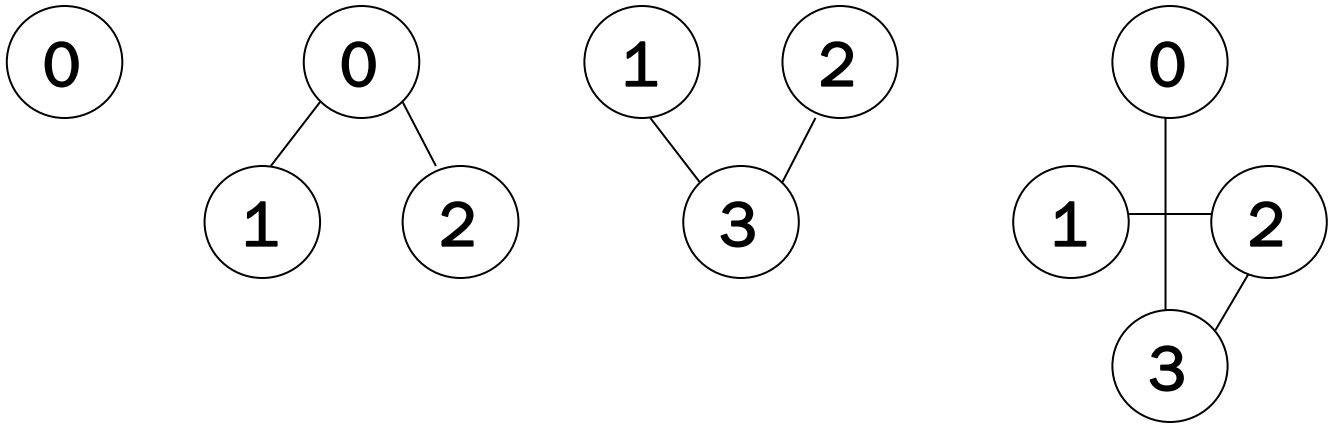
$E(G_1) = \{(1, 2), (1, 4), (2, 5), (3, 1), (3, 4), (4, 5)\}$
$E(G_2) = \{(0, 1), (0, 3), (1, 2), (1, 4), (2, 1), (2, 4), (4, 3)\}$
$E(G_3) = \{(0, 1), (0, 5), (1, 2), (1, 3), (1, 5), (2, 4), (4, 3),$
$(5, 6), (6, 8), (7, 3), (7, 8), (8, 10), (9, 4),$
$(9, 7), (9, 10)\}$

SUBGRAPH AND PATH

- A subgraph of G is a graph G' such that V(G') is a subset of V(G) and E(G') is a subset of E(G)
- A path from vertex v_p to vertex v_q in a graph G, is a sequence of vertices, v_p, v_{i1}, v_{i2}, ..., v_{in}, v_q, such that (v_p, v_{i1}), (v_{i1}, v_{i2}), ..., (v_{in}, v_q) are edges in an undirected graph
- The length of a path is the number of edges on it

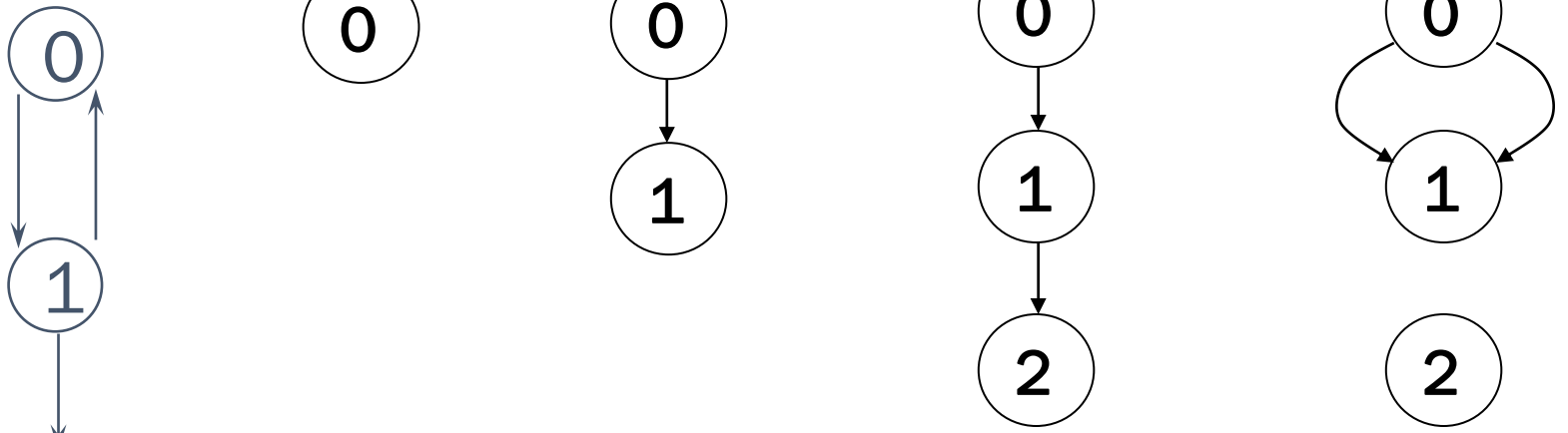


G_1



(i) (ii) (iii) (iv)

(a) Some of the subgraph of G_1



G_3

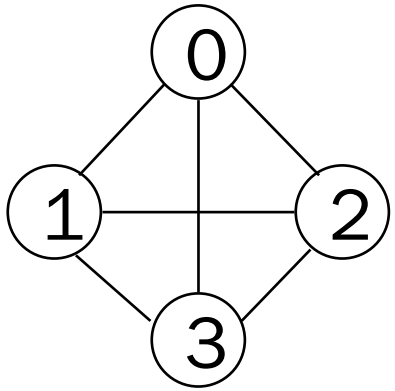
(i) (ii) (iii) (iv)

(b) Some of the subgraph of G_3

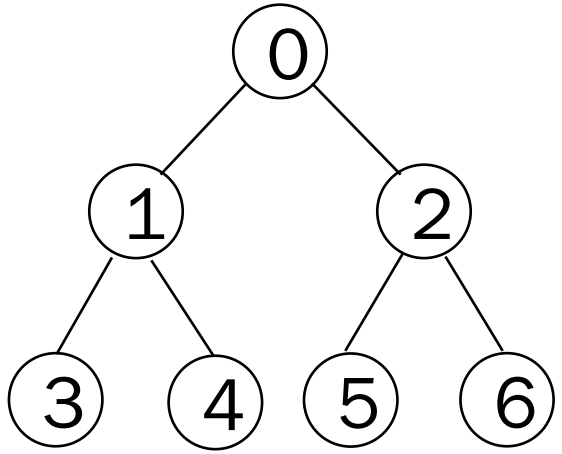
Simple Path and Style

- A **simple path** is a path in which all vertices, except possibly the first and the last, are distinct
- A **cycle** is a simple path in which the first and the last vertices are the same
- In an undirected graph G, two **vertices**, v_0 and v_1, are **connected** if there is a path in G from v_0 to v_1
- An undirected **graph** is **connected** if, for every pair of distinct vertices v_i, v_j, there is a path from v_i to v_j

CONNECTED GRAPHS



G_1



G_2

tree (acyclic graph)

What's the Data Structure?

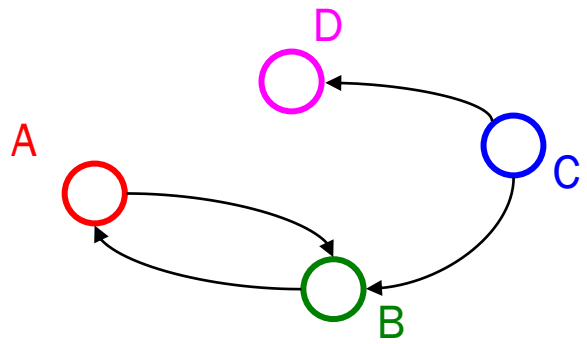
- Graphs are often useful for lots of data and questions
  - Example: "What's the lowest-cost path from x to y"
- But we need a data structure that represents graphs
- Which data structure is "best" can depend on:
  - properties of the graph (e.g., dense versus sparse)
  - the common queries about the graph ("is (u ,v) an edge?" vs "what are the neighbors of node u?")
- We will discuss two standard graph representations
- Adjacency Matrix and Adjacency List
- Different trade-offs, particularly time versus space

Adjacency Matrix

Assign each node a number from 0 to |V|-1

A |V| x |V| matrix of Booleans (or 0 vs. 1)

- Then M[u][v] == true means there is an edge from u to v



	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

Adjacency Matrix Properties

Running time to:

- Get a vertex's out-edges:
- Get a vertex's in-edges:
- Decide if some edge exists:
- Insert an edge:
- Delete an edge:

Space requirements:

Best for sparse or dense graphs?

	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

Adjacency Matrix Properties

Running time to:

- Get a vertex's out-edges: $O(|V|)$
- Get a vertex's in-edges: $O(|V|)$
- Decide if some edge exists: $O(1)$
- Insert an edge: $O(1)$
- Delete an edge: $O(1)$

Space requirements:

$O(|V|^2)$

Best for sparse or dense graphs? *dense*

	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

Adjacency Matrix Properties

How will the adjacency matrix vary for an undirected graph?

- Will be symmetric about diagonal axis
- Matrix: Could we save space by using only about half the array?

	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	T	F

- But how would you "get all neighbors"?

Adjacency Matrix Properties

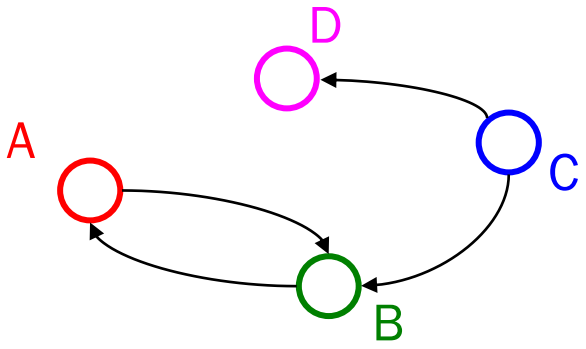
How can we adapt the representation for weighted graphs?

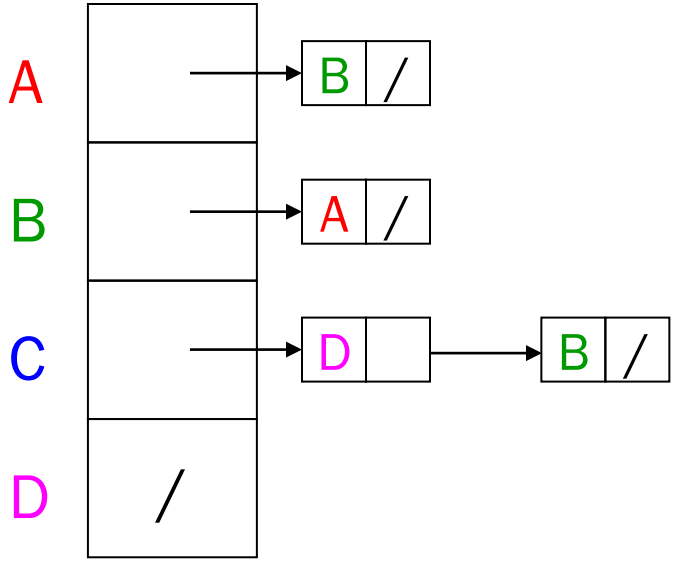
- Instead of Boolean, store a number in each cell
- Need some value to represent ‘not an edge’
  - 0, -1, or some other value based on how you are using the graph
  - Might need to be a separate field if no restrictions on weights

Adjacency List

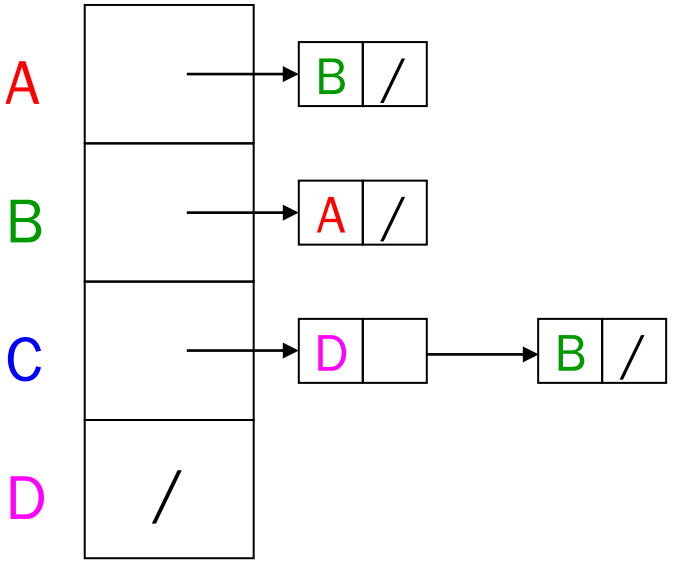
Assign each node a number from 0 to |V|-1

- An array of length |V| in which each entry stores a list of all adjacent vertices (e.g., linked list)





Adjacency List Properties



Running time to:

- Get a vertex's out-edges:
- Get a vertex's in-edges:
- Decide if some edge exists:
- Insert an edge:
- Delete an edge:

Space requirements:

Best for sparse or dense graphs?

Adjacency List Properties

Running time to:

- Get a vertex's out-edges: $O(d)$ where d is out-degree of vertex
- Get a vertex's in-edges: $O(|E|)$ (could keep a second adjacency list for this!)
- Decide if some edge exists: $O(d)$ where d is out-degree of source
- Insert an edge: $O(1)$ (unless you need to check if it's already there)
- Delete an edge: $O(d)$ where d is out-degree of source

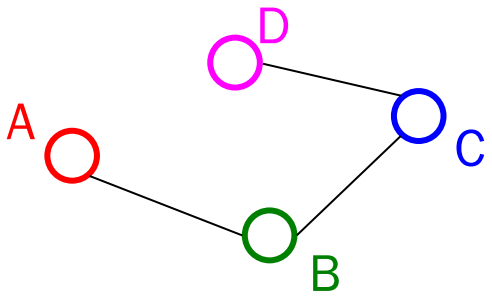
Space requirements: $O(|V|+|E|)$

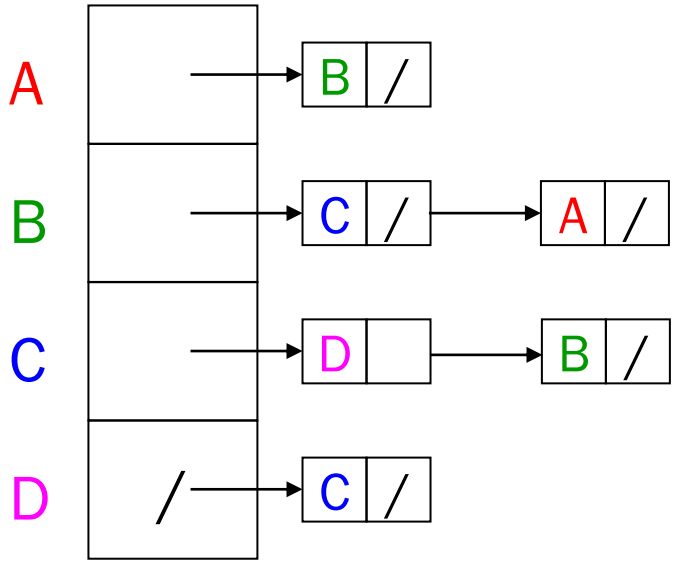
Best for sparse or dense graphs? *sparse*

Undirected Graphs

Adjacency lists also work well for undirected graphs with one caveat

- Put each edge in two lists to support efficient "get all neighbors"





Which is better?

Graphs are often sparse

- Streets form grids
- Airlines rarely fly to all cities

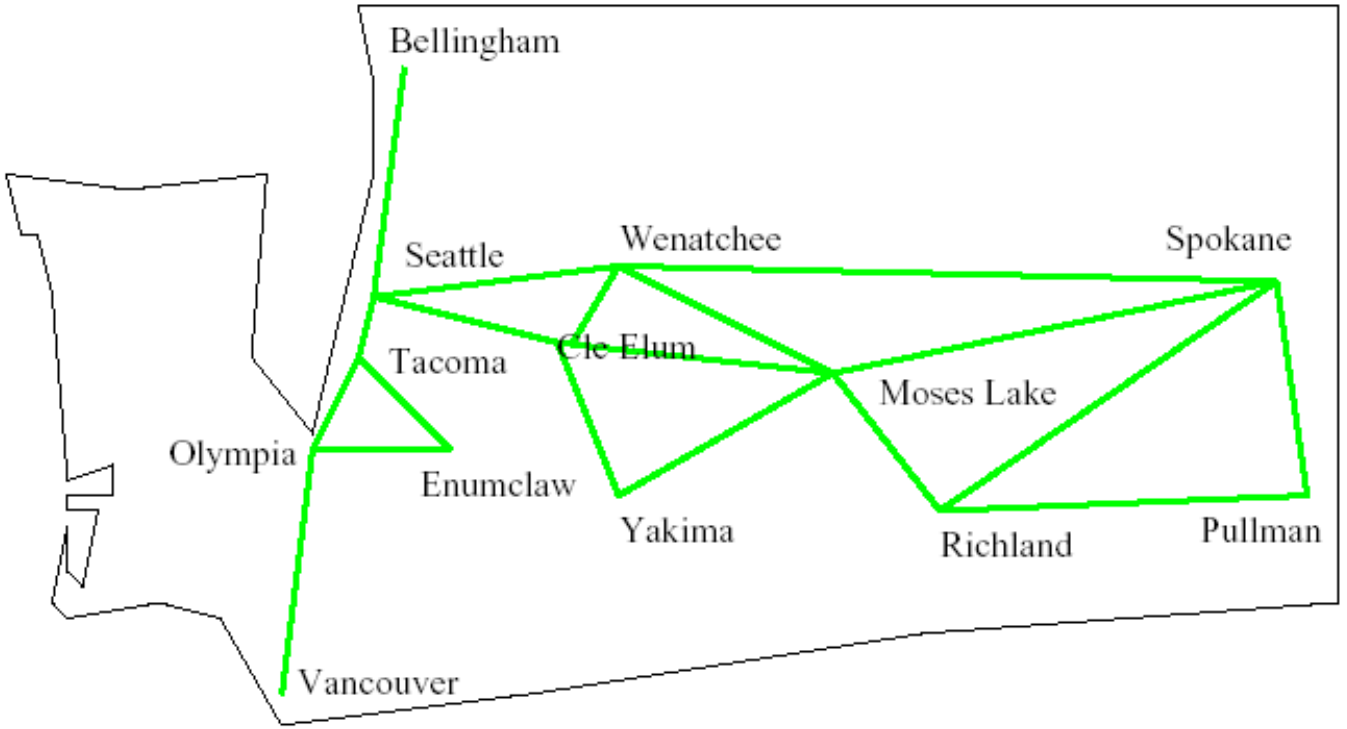
Adjacency lists should generally be your default choice

- Slower performance compensated by greater space savings

Applications of Graphs: Traversals

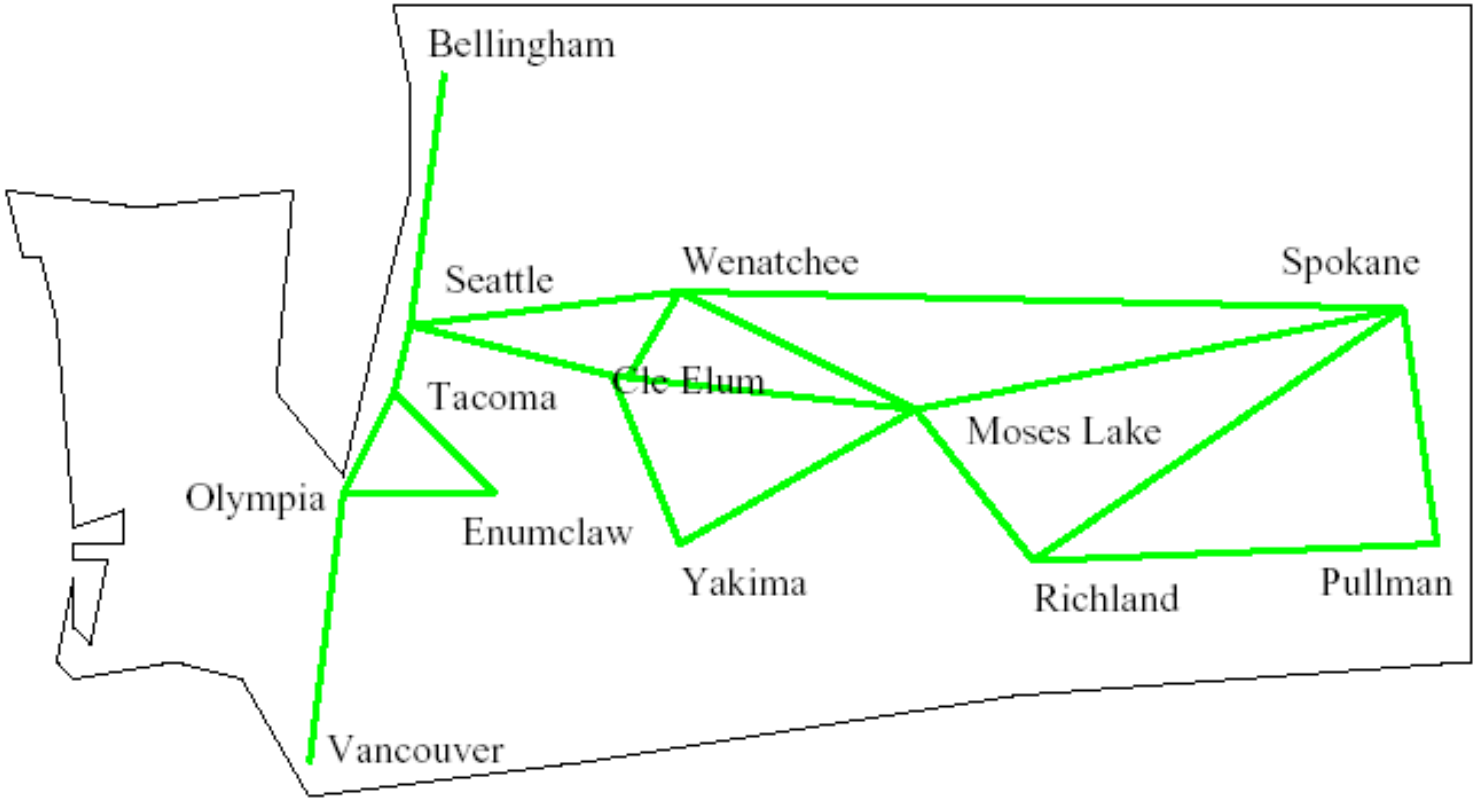
Might be easier to list what isn't a graph application...

Application: Moving Around WA State



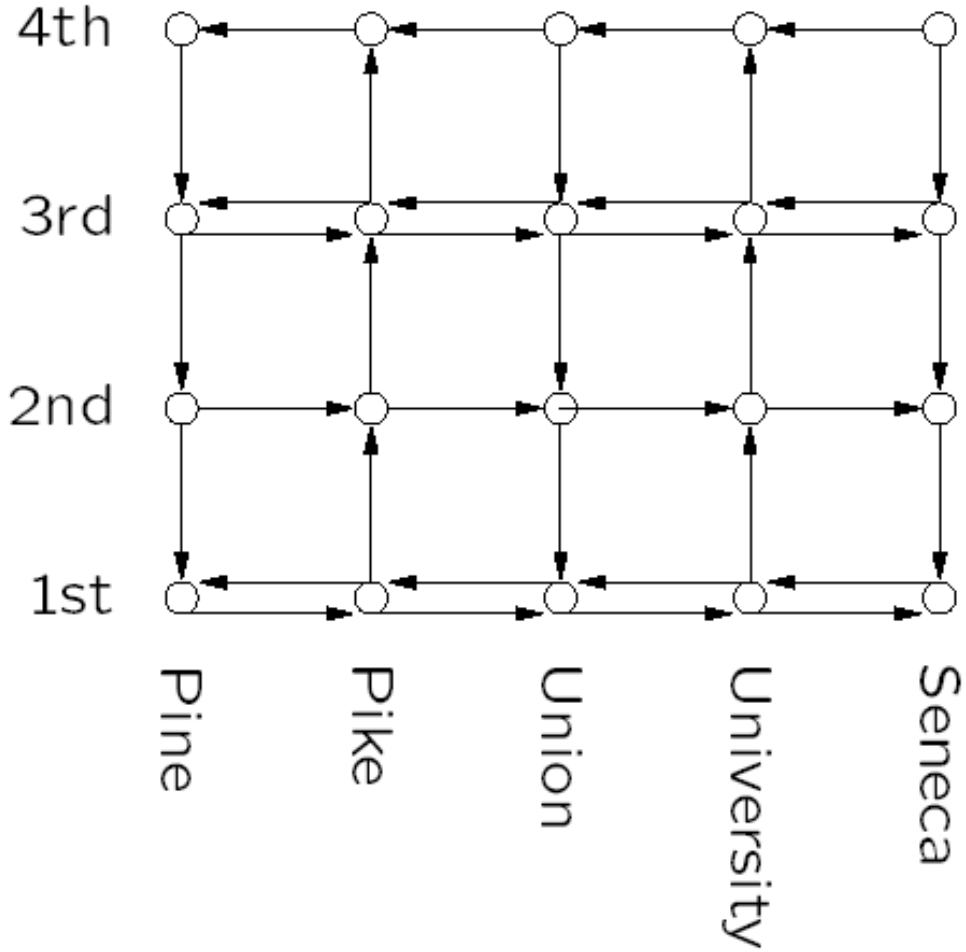
What's the *shortest way* to get from Seattle to Pullman?

Application: Moving Around WA State



What's the *fastest way* to get from Seattle to Pullman?

Applications: Bus Routes Downtown



If we're at 3rd and Pine, how can we get to 1st and University using Metro? How about 4th and Seneca?

Graph Traversals

For an arbitrary graph and a starting node v, find all nodes reachable from v (i.e., there exists a path)

- Possibly "do something" for each node (print to output, set some field, return from iterator, etc.)

Related Problems:

- Is an undirected graph connected?
- Is a digraph weakly/strongly connected?
  - For strongly, need a cycle back to starting node

Graph Traversals

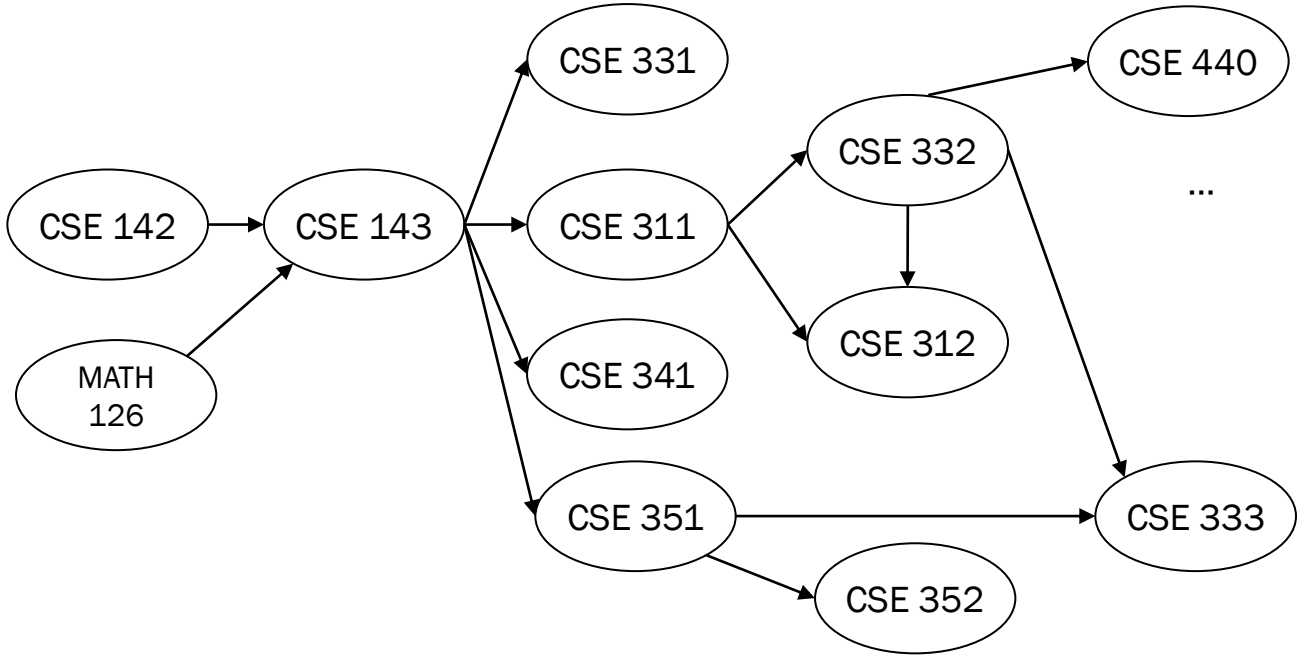
Basic Algorithm for Traversals:

- Select a starting node
- Make a set of nodes adjacent to current node
- Visit each node in the set but "mark" each nodes after visiting them so you don't revisit them (and eventually stop)
- Repeat above but skip "marked nodes"

Topological Sort

Problem: Given a DAG $G=(V, E)$, output all the vertices in order such that if no vertex appears before any other vertex that has an edge to it

Example input:



Example output:

- 142, 126, 143, 311, 331, 332, 312, 341, 351, 333, 440, 352

Questions and Comments

Terminology:

A DAG represents a partial order and a topological sort produces a total order that is consistent with it

Why do we perform topological sorts only on DAGs?

- Because a cycle means there is no correct answer

Is there always a unique answer?

- No, there can be one or more answers depending on the provided graph

What DAGs have exactly 1 answer?

- Lists

Uses Topological Sort

- Figuring out how to finish your degree
- Computing the order in which to recalculate cells in a spreadsheet
- Determining the order to compile files with dependencies
- In general, use a dependency graph to find an allowed order of execution

TYPES OF TRAVERSALS

- Breadth First Search BFS
- Depth First Search DFS

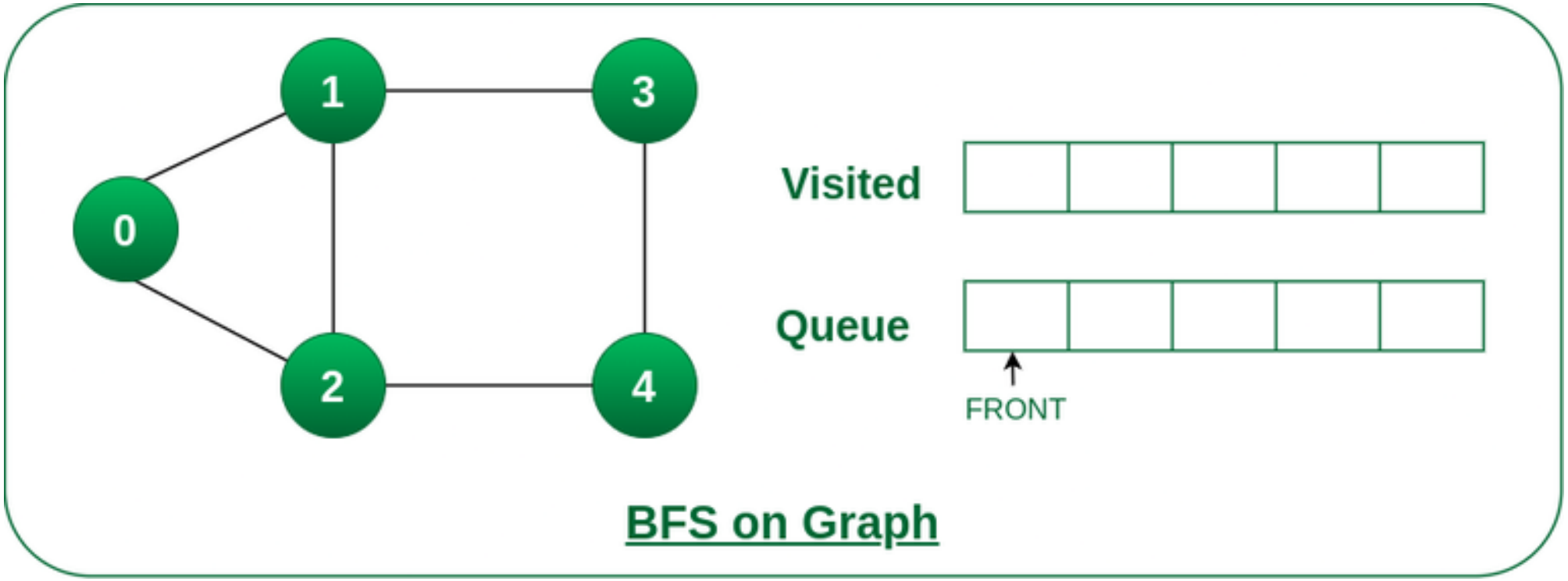
Algorithm of Breadth-First Search

- Utilize the following two data structures for traversing the graph.
  - Visited array(size of the graph)
  - Queue data structure

1) Consider the graph you want to navigate.
2) Select any vertex in your graph (say v1), from which you want to traverse the graph.
3) Add the starting vertex to the visited array, and afterward, you add v1's adjacent vertices to the queue data structure.
4) Now using the FIFO concept, remove the first element from the queue, put it into the visited array, and then add the adjacent vertices of the removed element to the queue.
5) Repeat step 4 until the queue is not empty and no vertex is left to be visited.

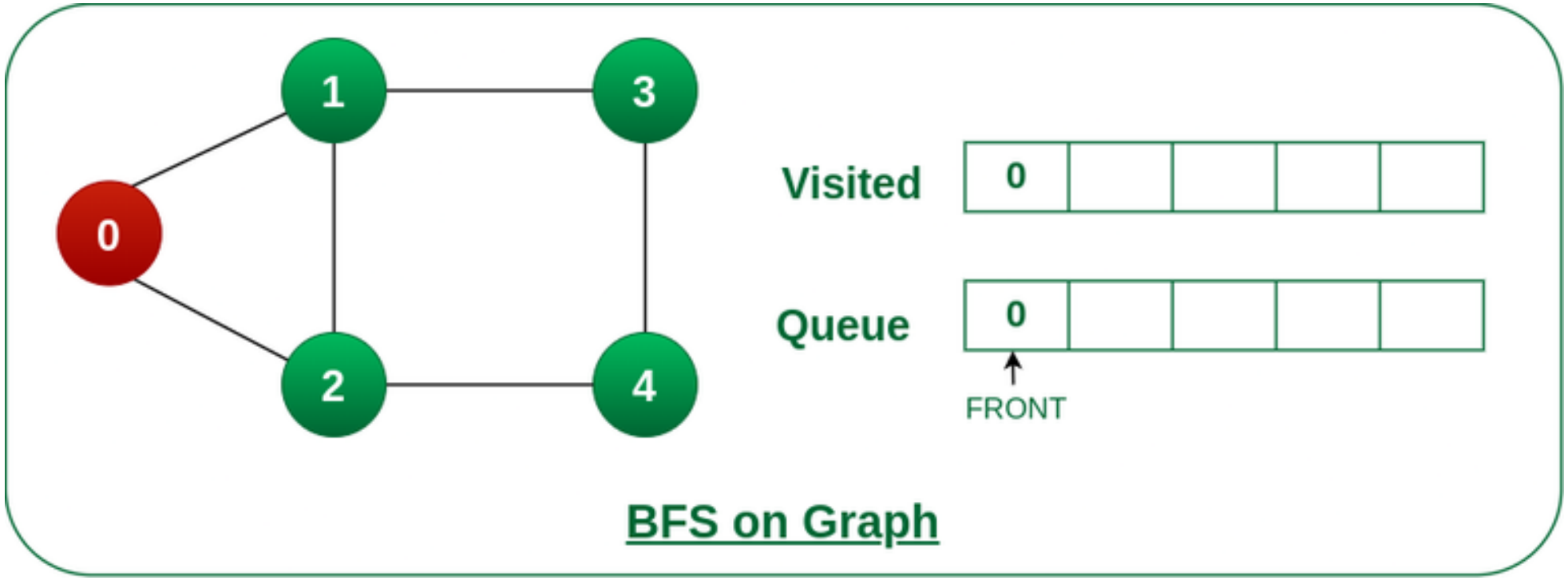
Breadth First Search

Step1: Initially queue and visited arrays are empty



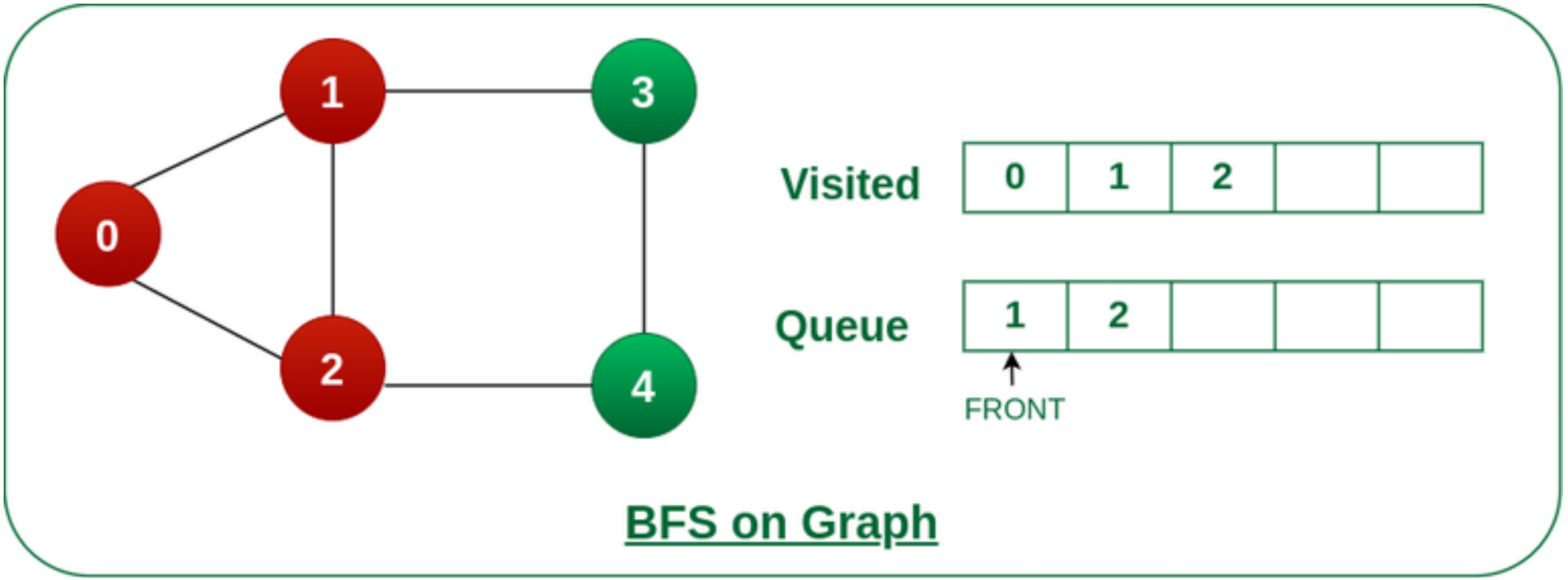
Breadth First Search

Step2: Push node 0 into queue and mark it visited.



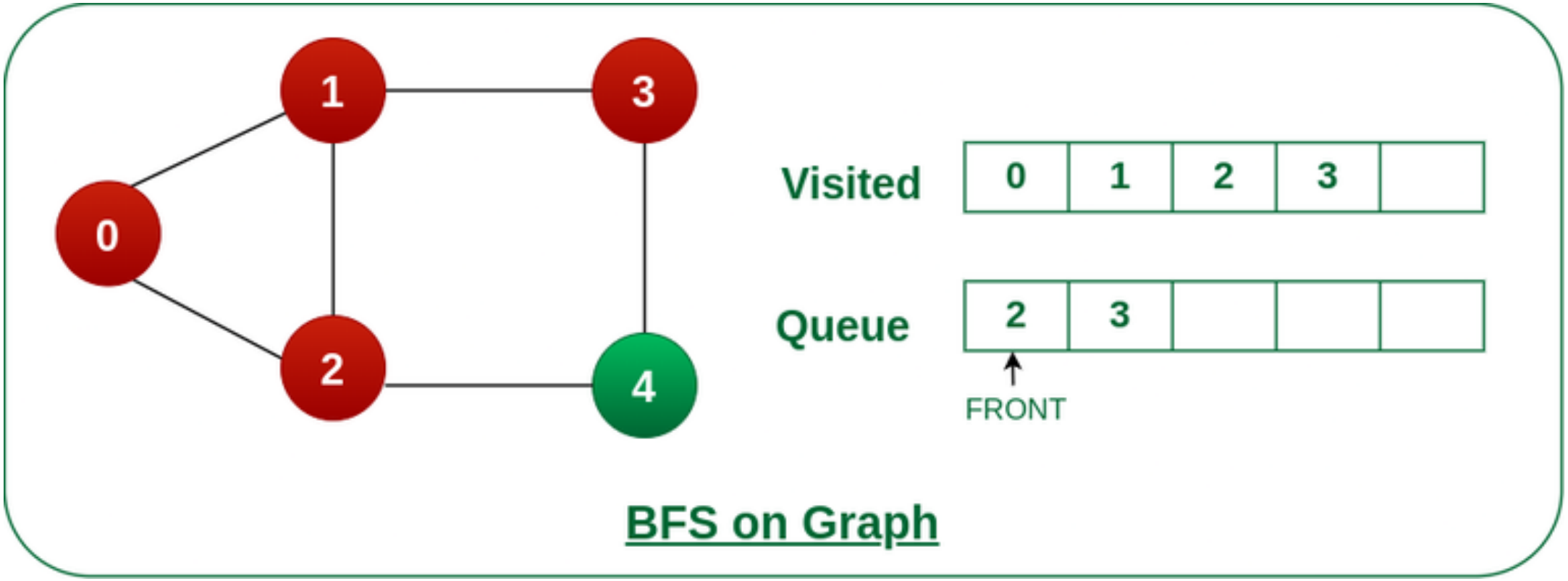
Breadth First Search

Step 3: Remove node 0 from the front of queue and visit the unvisited neighbors and push them into queue



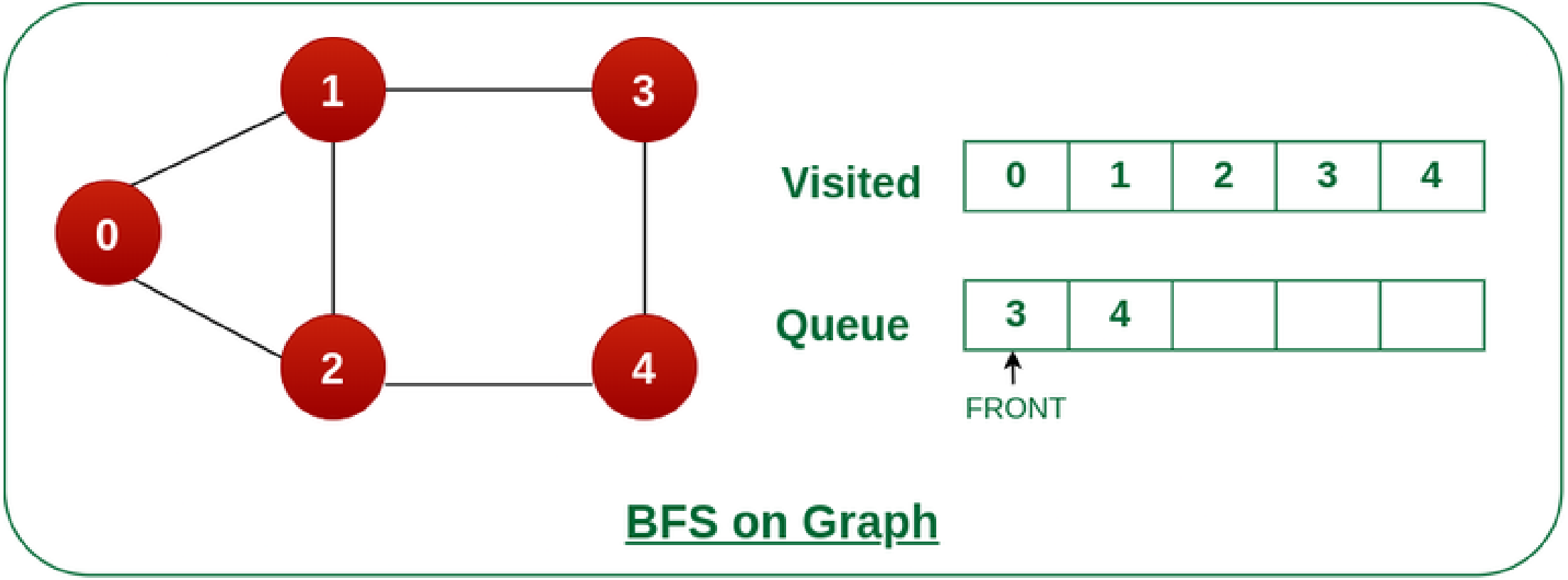
Breadth First Search

Step 4: Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.



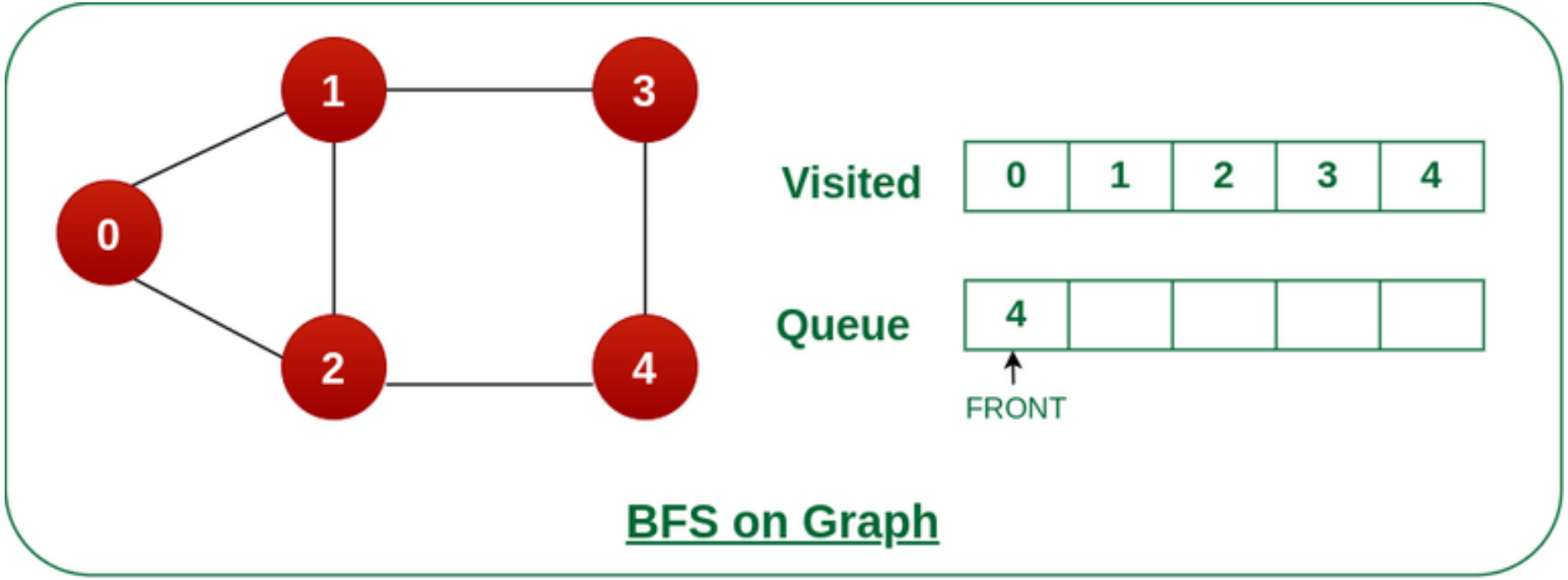
Breadth First Search

Step 5: Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue



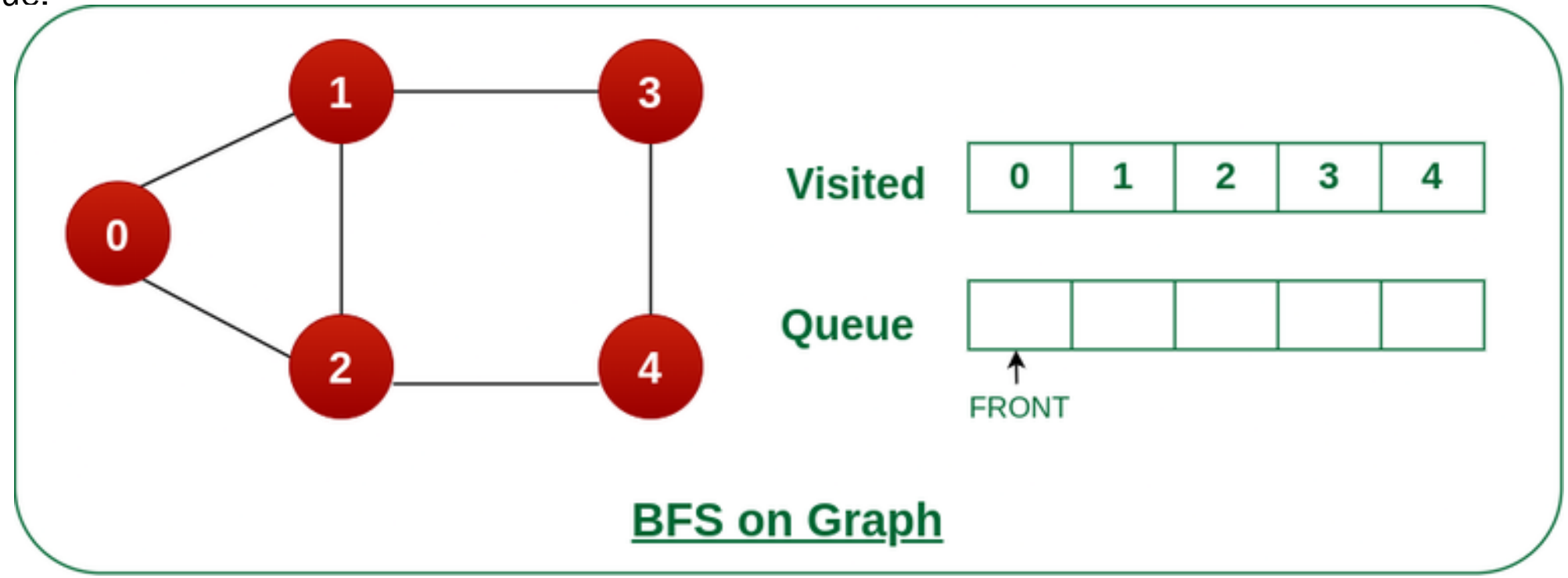
Breadth First Search

Step 6: Remove node 3 from the front of queue and visit the unvisited neighbors and push them into queue. As we can see that every neighbors of node 3 is visited, so move to the next node that are in the front of the queue



Breadth First Search

Steps 7: Remove node 4 from the front of queue and visit the unvisited neighbors and push them into queue. As we can see that every neighbors of node 4 are visited, so move to the next node that is in the front of the queue.



Now, Queue becomes empty, So, terminate these process of iteration.

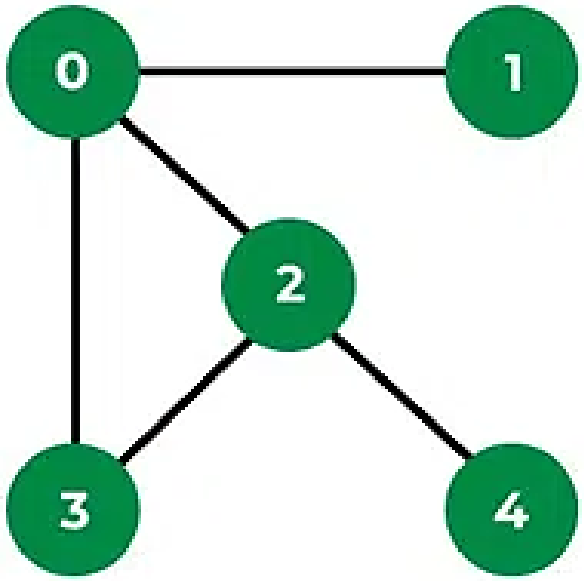
Algorithm of Depth First Search

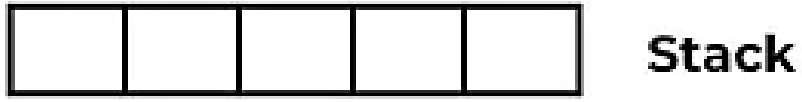
- Utilize the following two data structures for traversing the graph.
  - Visited array(size of the graph)
  - STACK data structure

1) Choose a starting node.
2) Push the starting node onto the stack.
3) Mark the starting node as visited.
4) While the stack is not empty, do the following:
5) Pop a node from the stack.
6) Process or perform any necessary operations on the popped node.
7) Get all the adjacent neighbors of the popped node.
8) For each adjacent neighbor, if it has not been visited, do the following:
   1) Mark the neighbor as visited.
   2) Push the neighbor onto the stack.
9) Repeat step 5 until the stack is empty.

Depth First Search

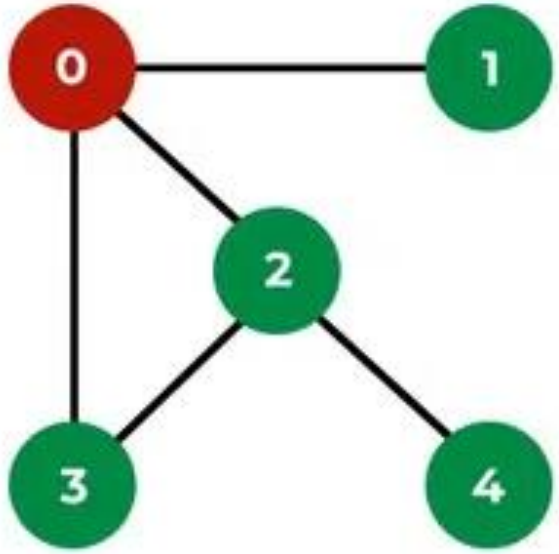
Step1: Initially stack and visited arrays are empty.





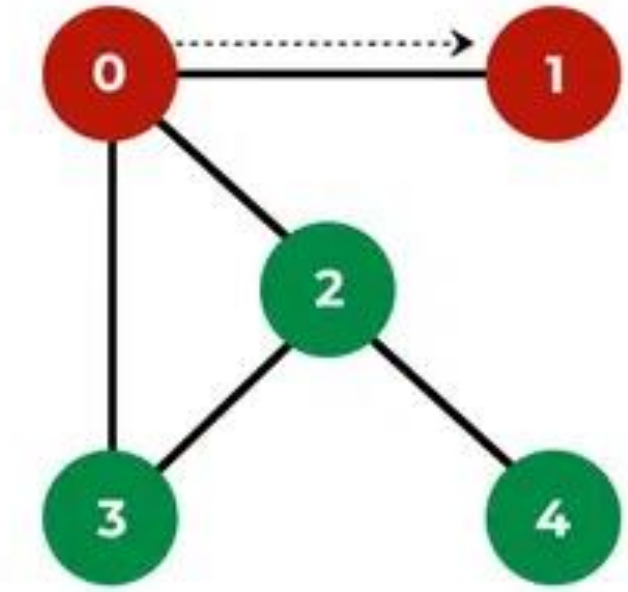
Depth First Search

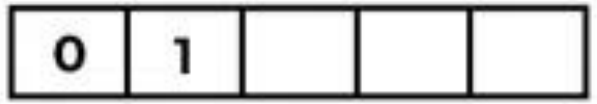
Step 2: Visit 0 and put its adjacent nodes which are not visited yet into the stack.



Depth First Search

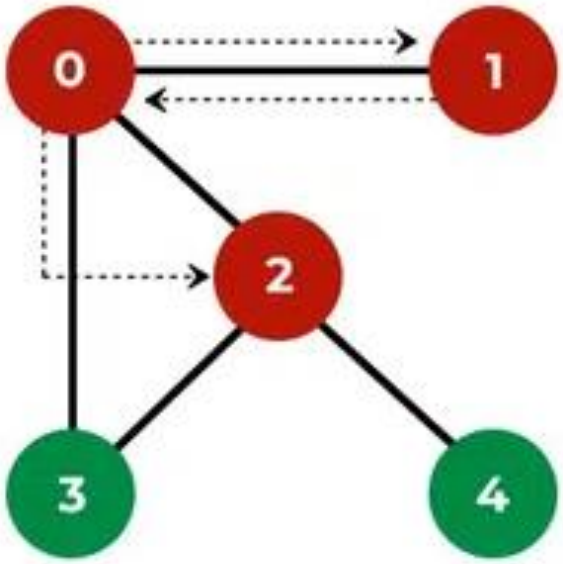
Step 3: Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.





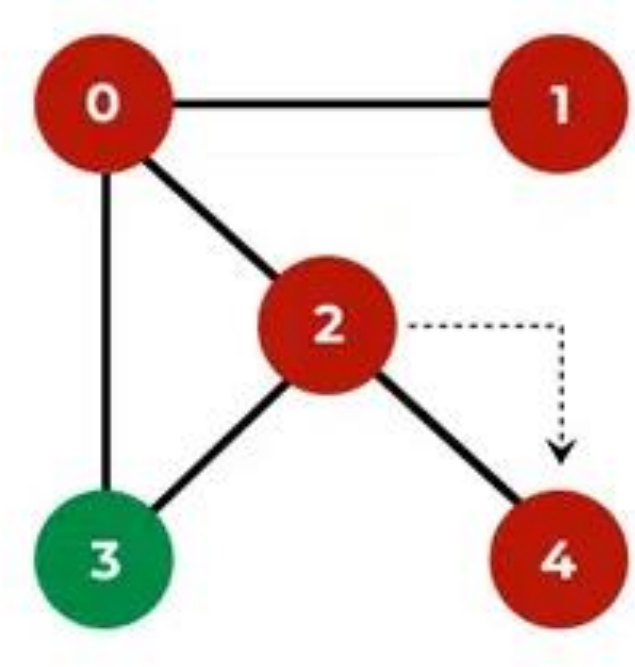
Depth First Search

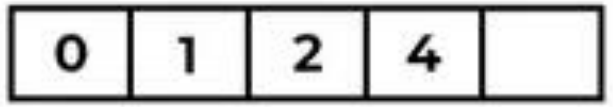
Step 4: Now, Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited (i.e, 3, 4) in the stack



Depth First Search

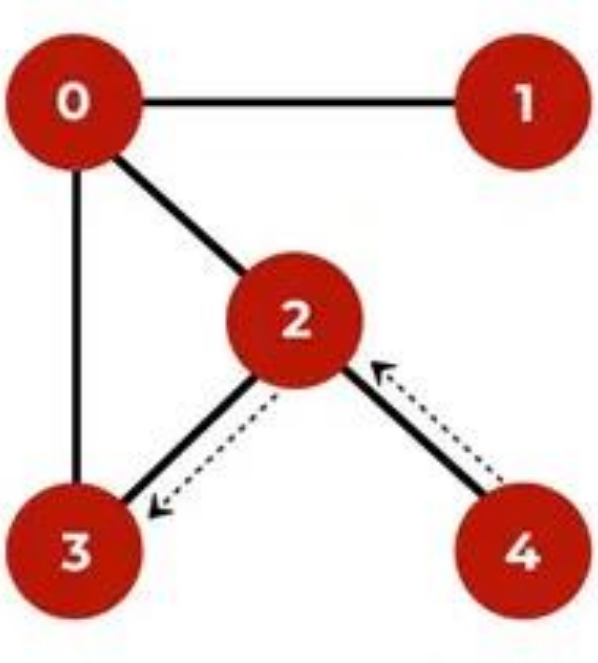
Step 5: Now, Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.

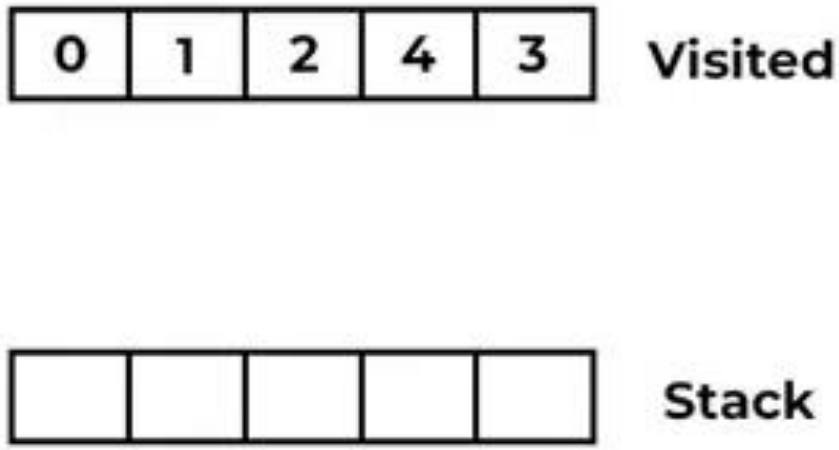




Depth First Search

Step 6: Now, Node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.





Now, Stack becomes empty, which means we have visited all the nodes and our DFS traversal ends.