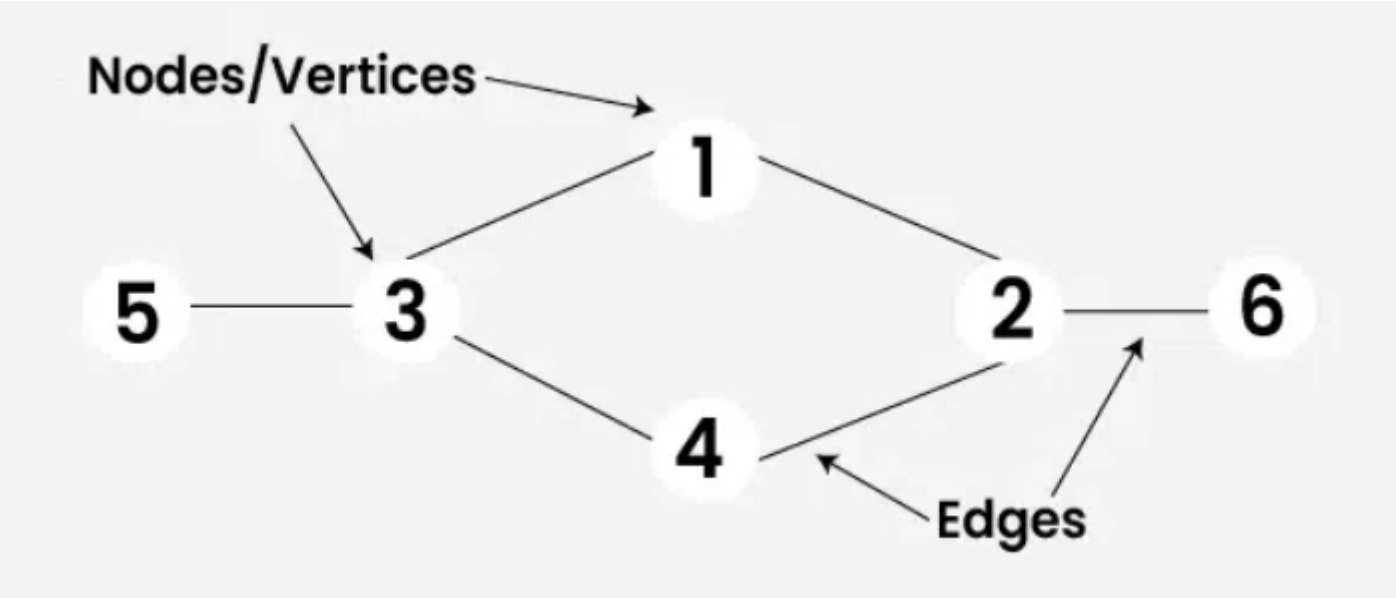
Module 8 - Graph, Stack, and Queue

1. Graphs Concept



A **Graph** is a non-linear data structure consisting of **nodes** and **edges**. The nodes are formally called vertices, and the edges are lines or arcs that connect any two nodes in the graph.

Graphs are used to solve many real-world problems. They are used to represent networks, such as social networks, transportation networks (roads and cities), or computer networks.

1.1 Graph Terminology

- **Vertex (or Node):** The fundamental unit of the graph.
- **Edge:** A line that connects two vertices.
- **Path:** A sequence of vertices connected by edges.
- **Cycle:** A path that starts and ends at the same vertex. A graph with no cycles is acyclic.
- **Connected Graph:** An undirected graph where there is a path between any two vertices.
- **Weighted Graph:** Each edge is assigned a numerical value or "weight," often representing cost or distance.
- **Unweighted Graph:** Edges do not have weights.

1.2 Graph Analogy

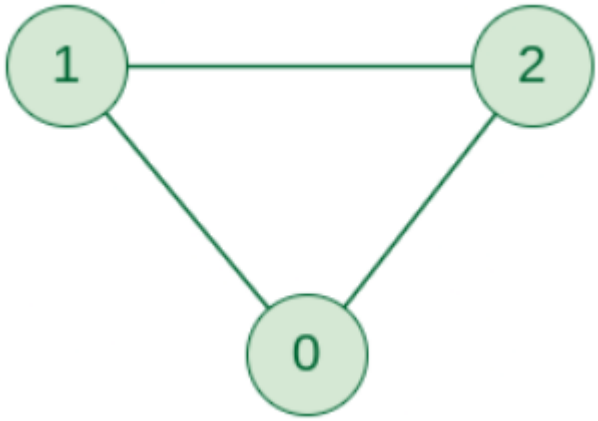
Imagine a social network like Facebook or Twitter.

- Each **person** is a **Vertex (Node)**.

- The **friendship** or **connection** between two people is an **Edge**.
- If the friendship is mutual (i.e. Facebook), it's an **undirected** graph.
- If you can "follow" someone without them following you back (i.e. Twitter), it's a **directed** graph.

1.3 Types of Graph

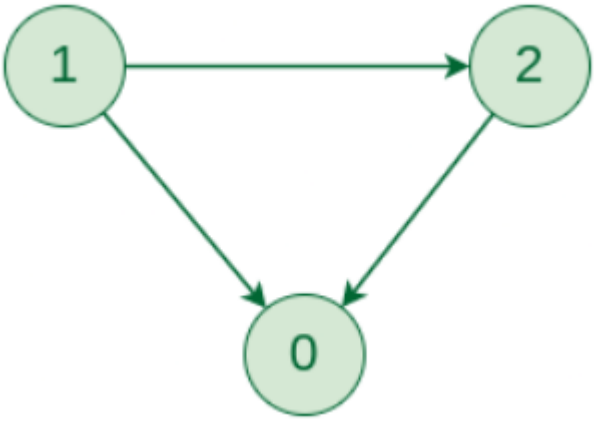
1.3.1 Undirected Graph



In an **Undirected Graph**, edges represent a mutual, bidirectional relationship. An edge `(u, v)` is identical to an edge `(v, u)`. There is no "direction" to the connection.

- **Analogy:** A friendship on Facebook or a two-way street. If `A` is friends with `B`, then `B` is automatically friends with `A`.
- **Degree:** The "degree" of a vertex is simply the total number of edges connected to it.

1.3.2 Directed Graph



In a **Directed Graph (or Digraph)**, edges are one-way streets. An edge `(u, v)` represents a connection from `u` to `v`, but it does not imply a connection from `v` to `u`.

- **Analogy:** Following someone on Twitter or a one-way street. `A` can follow `B` (an edge `A -> B`), but `B` does not have to follow `A` back. An edge `B -> A` would be a separate, distinct edge.
- **Degree:** Degree is split into two types:
  - **In-degree:** The number of edges pointing to the vertex.
  - **Out-degree:** The number of edges pointing away from the vertex.

2. Graph Representations

A graph is an abstract concept. To store one in a computer's memory, we must translate this idea of vertices and edges into a concrete data structure. The choice of representation is critical, as it dictates the performance and space efficiency of our graph algorithms.

A good representation must efficiently answer two fundamental questions:

1. "**Is vertex `u` connected to vertex `v`?**"
2. "**What are all the neighbors of vertex `u`?**"

The two most common methods to solve this are the Adjacency Matrix and the Adjacency List.

2.1. Adjacency Matrix

An **Adjacency Matrix** is a 2D array, typically of size `V x V`, where `V` is the number of vertices. Each cell `adj[u][v]` in the matrix stores information about the edge between vertex `u` and vertex `v`.

2.1.1. Concept

Imagine a flight chart. The rows represent the "source" vertex and the columns represent the "destination" vertex.

- For an **unweighted graph**, we store a `1` (or `true`) at `adj[u][v]` if an edge exists, and `0` (or `false`) if it does not.
- For a **weighted graph**, instead of storing `0` or `1`, we store the weight of the edge.
- For an **undirected graph**, the matrix will be symmetric about the diagonal (i.e., `adj[u][v] == adj[v][u]`).
- For a **directed graph**, the matrix is not necessarily symmetric. `matrix[u][v]` can be `1` while `matrix[v][u]` is `0`

2.1.2. Example

Consider this simple undirected graph with 4 vertices:

```
(0) --- (1)
| \       |
|  \      |
```

```
|   \     |
|    \    |
|     \   |
|      \  |
|       \ |
(3) --- (2)
```

Edges: (0,1), (0,2), (0,3), (1,2), (2,3)

The **Adjacency Matrix** for this graph would be:

```
  0 1 2 3
0[0,1,1,1]
1[1,0,1,0]
2[1,1,0,1]
3[1,0,1,0]
```

2.1.3 Implementation (C++)

In C++, you would implement this using a `vector` of `vector`s.

```
// Assume V is the number of vertices
int V = 4;

// 1. Initialize a V x V matrix with all zeros
std::vector<std::vector<int>> adjMatrix(
    V,
    std::vector<int>(V, 0)
);

// 2. Add an edge (e.g., between 0 and 2)
void addEdge(int u, int v) {
    adjMatrix[u][v] = 1;
    adjMatrix[v][u] = 1; // For undirected
}

// 3. To check for an edge (e.g., between 1 and 3):
bool hasEdge = (adjMatrix[1][3] == 1); // false

// 4. To find all neighbors of a vertex (e.g., vertex 0):
```

```
for (int j = 0; j < V; ++j) {
    if (adjMatrix[0][j] == 1) {
        // j is a neighbor of 0
        // This will find j=1, j=2, and j=3
    }
}
```

2.1.4. Pros and Cons

Pros:

- **Fast Edge Lookup:** Checking if an edge `(u, v)` exists is `O(1)`. You just access the `adjMatrix[u][v]` cell.
- **Simple to Implement:** The logic is straightforward, as it's just a 2D array lookup.

Cons:

- **Space Inefficient:** This is the biggest drawback. The matrix always requires `O(V^2)` space, regardless of how many edges the graph has.
- **Slow Neighbor Iteration:** Finding all neighbors of a vertex requires iterating through its entire row, which is an `O(V)` operation, even if the vertex only has one neighbor.

2.2. Adjacency List

An **Adjacency List** is an array (or `std::vector`) of lists. The main array has `V` entries, where each entry `adj[u]` corresponds to vertex `u`. The entry `adj[u]` then holds a list of all other vertices `v` that are neighbors of `u`.

2.2.1. Concept

Imagine a phone's contacts list. The main array (of size `V`) acts like your address book index. Each entry in this index (e.g., `adj[u]`) doesn't hold a single value, but rather points to a list of all that vertex's direct neighbors.

- For an **unweighted graph**, the list for `adj[u]` simply stores the indices (like `v1`, `v2`) of all its neighbors.
- For a **weighted graph**, the list for `adj[u]` stores pairs, where each pair contains the neighbor's index and the edge's weight (e.g., `{v1, w1}`, `{v2, w2}`).
- For an **undirected graph**, when you add an edge `(u, v)`, you must add `v` to `adj[u]`'s list and add `u` to `adj[v]`'s list.
- For a **directed graph**, When you add an edge `(u, v)`, you only add `v` to `adj[u]`'s list.

2.2.2. Example

Using the same 4-vertex graph:

```
(0) --- (1)
| \       |
|  \      |
|   \     |
|    \    |
|     \   |
|      \  |
|       \ |
(3) --- (2)
```

The **Adjacency List** for this graph would be:

```
0: -> [1, 2, 3]
1: -> [0, 2]
2: -> [0, 1, 3]
3: -> [0, 2]
```

2.2.3. Implementation (C++)

In C++, you would implement this using a `vector` of `list`s (or a `vector` of `vector`s).

```
// Assume V is the number of vertices
int V = 4;

// 1. Initialize a vector of size V. Each element
//    is an empty list.
std::vector<std::list<int>> adjList(V);

// 2. Add an edge (e.g., between 0 and 2)
void addEdge(int u, int v) {
    adjList[u].push_back(v);
    adjList[v].push_back(u); // For undirected
}

// 3. To check for an edge (e.g., between 1 and 3):
```

```
// This is slower than the matrix.
bool hasEdge = false;
for (auto& neighbor : adjList[1]) {
    if (neighbor == 3) {
        hasEdge = true;
        break;
    }
} // hasEdge is false

// 4. To find all neighbors of a vertex (e.g., vertex 0):
// This is extremely fast.
for (auto& neighbor : adjList[0]) {
    // neighbor is a neighbor of 0
    // This will find neighbor=1, neighbor=2, and neighbor=3
}
```

2.2.4. Handling Weighted Graphs

To store weights, the list cannot just be of `int`s. It must store pairs (or a custom `struct`).

- `std::vector<std::list<std::pair<int, int>>> adjList(V);`
- The `pair<int, int>` would store `{neighbor, weight}`.
- `addEdge` would look like: `adjList[u].push_back({v, weight});`

2.3.5. Pros and Cons

Pros:

- **Space Efficient:** This is the primary advantage. The total space required is `O(V + E)`.
- **Fast Neighbor Iteration:** Finding all neighbors of a vertex u is `O(deg(u))` (where `deg(u)` is the degree, or number of neighbors). This is optimally fast.

Cons:

- **Slow Edge Lookup:** Checking if a specific edge `(u, v)` exists is `O(deg(u))` in the worst case, because we must iterate through the list `adj[u]` to see if `v` is in it.

2.3. Comparison: Matrix vs. List

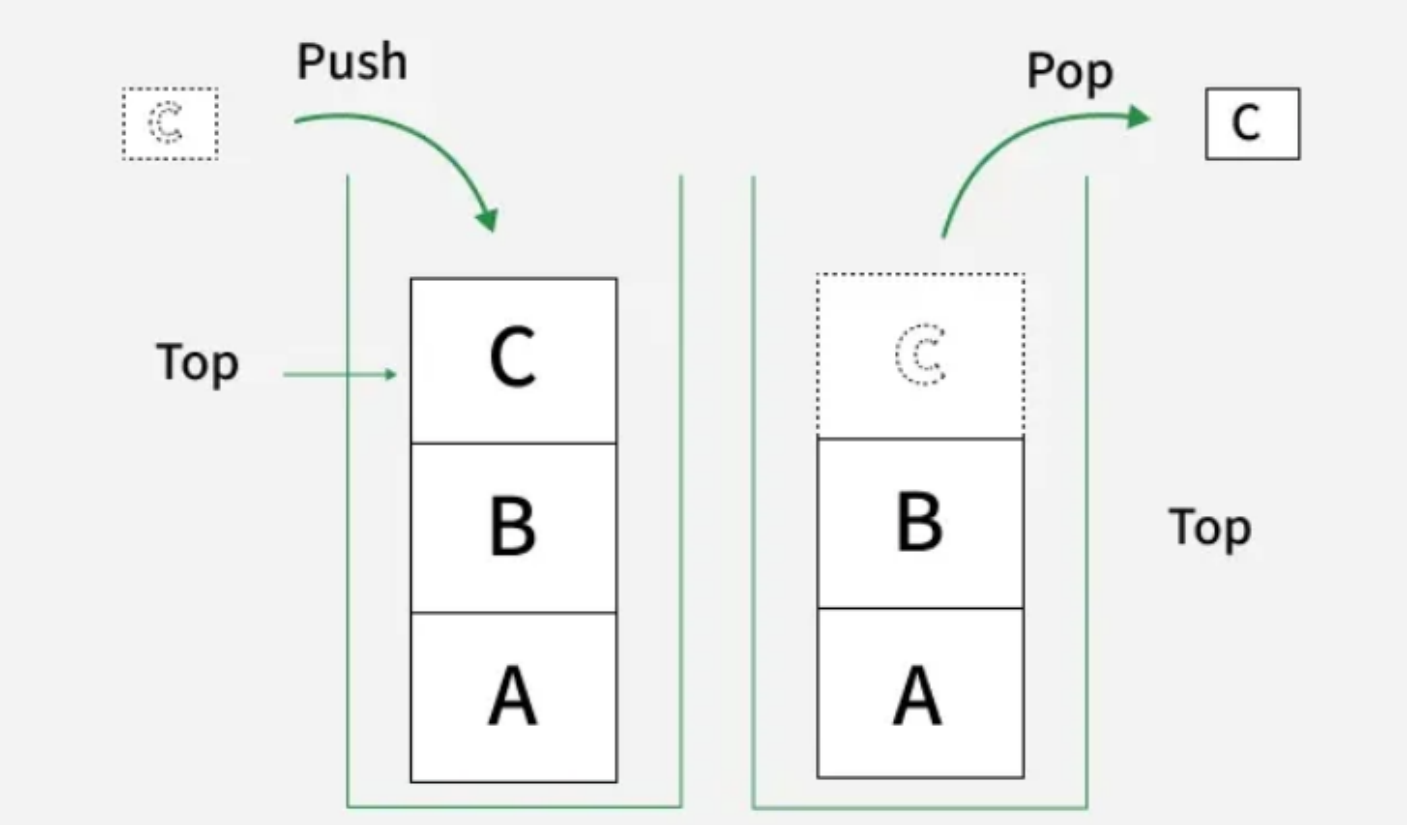
Operation	Adjacency Matrix	Adjacency List

Space	`O(V²)`	`O(V + E)`
Add Edge	`O(1)`	`O(1)`
Check Edge (u, v)	`O(1)` (Fast)	`O(deg(u))` (Slow)
Find Neighbors of u	`O(V)` (Slow)	`O(deg(u))` (Fast)
Remove Edge (u, v)	`O(1)`	`O(deg(u))`
Best For	**Dense Graphs** (where `E` ≈ `V²`)	**Sparse Graphs** (most common case)

3. Stack and Queue

Before diving into graph traversal, we must understand the two key data structures that power them: `std::stack` (for DFS) and `std::queue` (for BFS).

3.1 Introduction to `std::stack`



`std::stack` is a **container adapter** in the C++ STL. It's not a container itself, but a "wrapper" that provides a specific **LIFO (Last-In, First-Out)** interface on top of another container (like `std::deque` by default). It's like a stack of plates. You add new plates to the top and remove plates from the top.

Key Operations:

Operation	Description
`push(item)`	Adds an item to the top of the stack.
`pop()`	Removes the item from the top of the stack.
`top()`	Returns a reference to the item at the top.
`empty()`	Returns true if the stack is empty.

Operation	Description
`size()`	Returns the number of items in the stack.

Example:

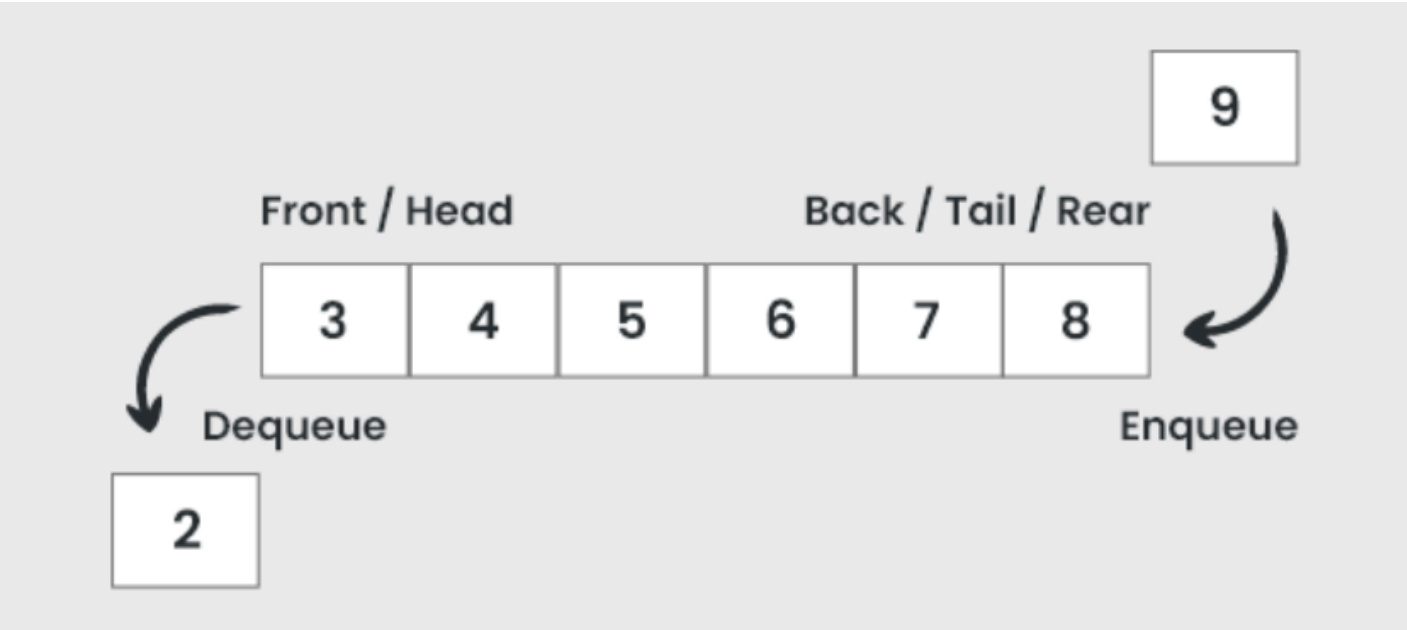
```
#include <stack>
#include <iostream>

int main() {
    std::stack<int> myStack;
    myStack.push(10); // Stack: [10]
    myStack.push(20); // Stack: [10, 20]
    myStack.push(30); // Stack: [10, 20, 30]

    std::cout << "Top: " << myStack.top() << std::endl; // Prints 30
    myStack.pop();   // Removes 30. Stack: [10, 20]
    std::cout << "Top: " << myStack.top() << std::endl; // Prints 20

    return 0;
}
```

3.2 Introduction to `std::queue`



`std::queue` is also a **container adapter**. It provides a **FIFO (First-In, First-Out)** interface. It's like a line at a ticket counter. The first person to get in line is the first person to be served.

Key Operations:

Operation	Description
push(item)	Adds an item to the back of the queue.
pop()	Removes the item from the front of the queue.
front()	Returns a reference to the item at the front.
back()	Returns a reference to the item at the back.
empty()	Returns true if the queue is empty.
size()	Returns the number of items in the queue.

Example:

```
#include <queue>
#include <iostream>

int main() {
    std::queue<int> myQueue;
    myQueue.push(10); // Queue: [10]
    myQueue.push(20); // Queue: [10, 20]
    myQueue.push(30); // Queue: [10, 20, 30]

    std::cout << "Front: " << myQueue.front() << std::endl; // Prints 10
    myQueue.pop();    // Removes 10. Queue: [20, 30]
    std::cout << "Front: " << myQueue.front() << std::endl; // Prints 20

    return 0;
}
```

4. Graph Traversal: Breadth-First Search (BFS)

Breadth-First Search (BFS) is a traversal algorithm that explores the graph "**level by level**." It starts at a source vertex, explores all of its immediate neighbors, then all of their neighbors, and so on.

4.1 BFS Analogy and Illustration

Imagine a **ripple effect in a pond**. When you drop a stone (the **source vertex**), the ripple expands:

- It first hits all the water molecules immediately next to it (Level 1 Neighbors).
- Then, the ripple expands to hit the next layer of molecules (Level 2 Neighbors).

BFS works exactly this way. It is excellent for finding the **shortest path** in an **unweighted graph**.

Example: We want to trace BFS starting from node `0` on the following graph:

```
      0
     / \
    1---2
   /     \
  3       4
```

The traversal order will be `0, 1, 2, 3, 4`.

4.2 The Role of `std::queue`

The FIFO (First-In, First-Out) nature of a queue is perfect for BFS.

1. We add the `source` (Level 0) to the queue.
2. We dequeue the `source`, and add all its neighbors (Level 1) to the queue
3. We dequeue all the Level 1 nodes one by one, and as we do, we add their neighbors (Level 2) to the **back** of the queue.

This ensures we process all nodes at the current level **before** moving to the next level, just like the ripple effect.

4.3 BFS Implementation

BFS is implemented using an **iterative approach** with a `std::queue`. This approach is natural to the algorithm's "level-by-level" logic. The queue ensures that nodes are processed in the order they are discovered. We also use a visited array (or `std::vector<bool>`) to keep track of nodes we have already processed or added to the queue, which is crucial for preventing infinite loops in graphs with cycles.

```
void BFS(int s) { // 's' is the source vertex
    // 1. Create a visited array, initialized to all false
    std::vector<bool> visited(V, false);

    // 2. Create a Queue for BFS
    std::queue<int> q;

    // 3. Mark the source vertex as visited and enqueue it
    visited[s] = true;
    q.push(s);

    std::cout << "BFS Traversal: ";

    // 4. Loop as long as the queue is not empty
    while (!q.empty()) {
        // 5. Dequeue a vertex from the front and print it
        int u = q.front();
        q.pop();
        std::cout << u << " ";

        // 6. Get all neighbors of the dequeued vertex 'u'
        for (auto& neighbor : adj[u]) {

            // 7. If a neighbor has not been visited:
            if (!visited[neighbor]) {
                // Mark it as visited
                visited[neighbor] = true;
                // Enqueue it (add to the back of the queue)
                q.push(neighbor);
```

```
            }
        }
    }
    std::cout << std::endl;
}
```

Code Explanation:

- Initialize a `visited` array (all `false`) and an empty `queue`.
- Mark the starting node `s` as visited and add it to the queue.
- While the queue is **not empty**:
  - Dequeue `a` node `u` (this is the **current node**).
  - Print `u`.
  - Iterate through all neighbors of `u`.
  - If a neighbor has not been visited, mark it as visited and enqueue it.

5. Graph Traversal: Depth-First Search (DFS)

Depth-First Search (DFS) is a traversal algorithm that explores the graph by going as "**deep**" as possible down one path before backtracking.

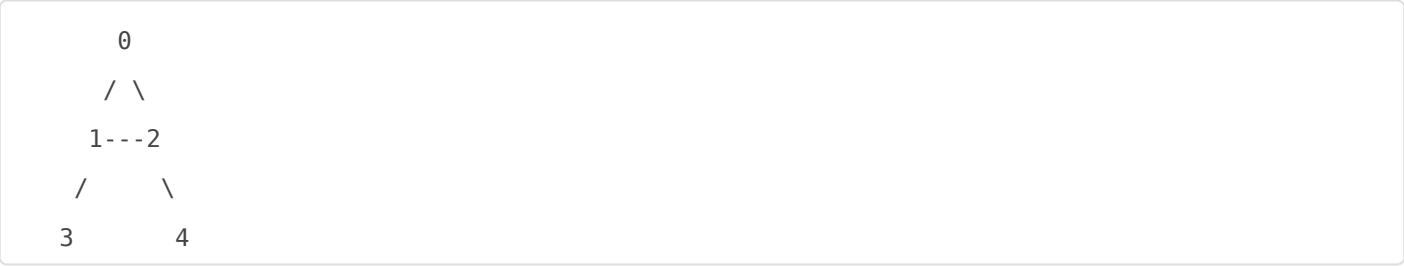
5.1 DFS Analogy and Illustration

Imagine exploring a **maze** with only one path.

- You pick a direction (an **edge**) and follow it.
- You keep going, taking the **first available turn** at each junction (visiting the first unvisited neighbor).
- You continue until you hit a **dead end** (a node with no unvisited neighbors).
- At the dead end, you **backtrack** to the previous junction and **try a different path**.

DFS works this way. It is excellent for detecting cycles, topological sorting, and solving puzzles.

Example: We want to trace BFS starting from node `0` on the following graph:



The traversal order will be `0, 1, 2, 4, 3`.

5.2 The Role of `std::stack`

The LIFO (Last-In, First-Out) nature of a stack is perfect for DFS.

- **Recursive DFS:** The system's call stack is used implicitly. When you make a **recursive call** `DFSUtil(neighbor)`, you **"push" the new function** onto the stack. When you hit a dead end, the function returns, **"popping" itself off** the stack and automatically "backtracking" to the previous node.

- **Iterative DFS:** We manually use a `std::stack`. We **push a node**, then **push its neighbors**. The last neighbor pushed is on **top**, so it will be the first one we explore next, perfectly mimicking the "go deep" behavior of recursion.

5.3 DFS Implementation

5.3.1 Iterative Approach

This approach uses an **explicit** `std::stack` and avoids recursion.

```
void DFS(int s) {
    // 1. Create a visited array, initialized to all false
    std::vector<bool> visited(V, false);

    // 2. Create a Stack for DFS
    std::stack<int> stack;

    // 3. Push the source vertex onto the stack
    stack.push(s);

    std::cout << "DFS Traversal (Iterative): ";

    // 4. Loop as long as the stack is not empty
    while (!stack.empty()) {
        // 5. Pop a vertex from the top of the stack
        int u = stack.top();
        stack.pop();

        // 6. If it hasn't been visited yet:
        if (!visited[u]) {
            visited[u] = true;
            std::cout << u << " ";
        }

        // 7. Get all neighbors of 'u'
        for (auto& neighbor : adj[u]) {
            // 8. If the neighbor is unvisited, push it
            if (!visited[neighbor]) {
                stack.push(neighbor);
```

```
            }
        }
    }
    std::cout << std::endl;
}
```

Code Explanation:

- Initialize a `visited` array (all `false`) and an empty `stack`.
- Push the starting node `s` onto the stack.
- While the stack is not empty:
  - Pop a node `u` from the stack.
  - If `u` has not been visited:
    - Mark `u` as visited and print it.
    - Iterate through all neighbors of `u`.
    - If a neighbor has not been visited, push it onto the stack (to be processed next).

5.3.2 Recursive Approach

This is the most common and intuitive way to implement DFS. It uses a **helper function**.

```
// Private helper function for recursive DFS
void DFSUtil(int u, std::vector<bool>& visited) {
    // 1. Mark the current node as visited and print it
    visited[u] = true;
    std::cout << u << " ";

    // 2. Iterate over all neighbors
    for (auto& neighbor : adj[u]) {
        // 3. If a neighbor is unvisited:
        if (!visited[neighbor]) {
            // 4. Make a recursive call to "go deeper"
            DFSUtil(neighbor, visited);
        }
    }
    // 5. When all neighbors are visited, the function returns.
    //    This is the implicit "backtracking" step.
}

// Public wrapper function to start the DFS
```

```
void DFSRecursive(int s) {
    // 1. Create a visited array
    std::vector<bool> visited(V, false);
    std::cout << "DFS Traversal (Recursive): ";

    // 2. Call the helper function to start the recursion
    DFSUtil(s, visited);
    std::cout << std::endl;
}
```

Code Explanation:

- The `DFSRecursive` function initializes the `visited` array.
- It calls the helper function `DFSUtil` with the starting node `s`.
- Inside `DFSUtil` (the recursive part):
  - Marks the current node `u` as visited and prints it.
  - Iterates through all neighbors of `u`.
  - If a neighbor has not been visited, it immediately calls `DFSUtil` on that neighbor, "**going deeper**."
  - When a node has no unvisited neighbors, its function call finishes and "**backtracks**" to the previous node's loop.

5.3.3 Comparison: Iterative vs. Recursive

Both recursive and iterative DFS **accomplish the same goal**, but they differ in implementation and performance characteristics.

Aspect	Recursive DFS	Iterative DFS
Logic	Intuitive and clean. Backtracking is handled **implicitly** by function returns.	Backtracking is handled **explicitly** using a `std::stack`.
Data Structure	Uses the **Call Stack** (managed by the system).	Uses an explicit `std::stack` (managed by the programmer on the heap).
Memory Usage	Can cause a **Stack Overflow** error if the graph is very deep (long paths).	Safer for very deep graphs, as heap memory is much larger than stack memory.
Code Simplicity	Often shorter and easier to write and understand the core "go deep" logic.	Can be more complex to write, but the process is more transparent.

In summary:

- Use **Recursive DFS** for simplicity and clarity, especially when the graph depth is **known to be manageable**.

- Use **Iterative DFS** when you need to avoid stack overflow on **very deep graphs** or when you need **finer control over the traversal**.

6. Example: Full Code Implementation

This chapter combines **all the concepts** into a **single Graph class** using the C++ STL for the adjacency list.

```
#include <iostream>
#include <vector>
#include <list>     // For Adjacency List
#include <queue>    // For BFS
#include <stack>    // For Iterative DFS
#include <vector>   // For visited tracker

class Graph {
private:
    int V; // Number of vertices
    // Adjacency List: A vector of lists
    std::vector<std::list<int>> adj;

    // Helper for recursive DFS
    void DFSUtil(int u, std::vector<bool>& visited) {
        visited[u] = true;
        std::cout << u << " ";
        for (auto& neighbor : adj[u]) {
            if (!visited[neighbor]) {
                DFSUtil(neighbor, visited);
            }
        }
    }

public:
    // Constructor
    Graph(int V) {
        this->V = V;
        adj.resize(V); // Resize the vector to hold V lists
    }
```

```
// Add an edge (for an undirected graph)
void addEdge(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u); // Comment this line for a directed graph
}

// Breadth-First Search (Iterative)
void BFS(int s) {
    std::vector<bool> visited(V, false);
    std::queue<int> q;
    visited[s] = true;
    q.push(s);

    std::cout << "BFS Traversal: ";
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        std::cout << u << " ";

        for (auto& neighbor : adj[u]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
    std::cout << std::endl;
}

// Depth-First Search (Iterative)
void DFS(int s) {
    std::vector<bool> visited(V, false);
    std::stack<int> stack;
    stack.push(s);

    std::cout << "DFS Traversal (Iterative): ";
    while (!stack.empty()) {
        int u = stack.top();
        stack.pop();

        if (!visited[u]) {
```

```
                visited[u] = true;
                std::cout << u << " ";
            }

            for (auto& neighbor : adj[u]) {
                if (!visited[neighbor]) {
                    stack.push(neighbor);
                }
            }
        }
        std::cout << std::endl;
    }

    // Depth-First Search (Recursive)
    void DFSRecursive(int s) {
        std::vector<bool> visited(V, false);
        std::cout << "DFS Traversal (Recursive): ";
        DFSUtil(s, visited);
        std::cout << std::endl;
    }
};
```