Absolutely, Vishwas! Here's a **detailed line-by-line explanation** of the entire C++ program that implements **Depth First Search (DFS)** and **Breadth First Search (BFS)** on an **undirected graph**. This will cover all key concepts, data structures, flow, and how each part works together.

**🔰 OVERVIEW**

* You have a Graph class that:
  + Builds an **undirected graph** using an adjacency list.
  + Implements **DFS using recursion**.
  + Implements **BFS using a queue**.
* The main() function:
  + Creates a graph.
  + Adds edges.
  + Runs both DFS and BFS from a starting node (0 in this case).

**📦 HEADERS AND NAMESPACES**

#include <iostream>

#include <vector>

#include <unordered\_map>

#include <unordered\_set>

#include <queue>

using namespace std;

**📌 What These Do:**

* #include <iostream> – For input/output (cout, cin).
* #include <vector> – To use vector for adjacency list.
* #include <unordered\_map> – To store the graph (node → list of neighbors).
* #include <unordered\_set> – To track visited nodes efficiently.
* #include <queue> – For implementing BFS using a queue.
* using namespace std; allows using cout, vector, etc., without prefixing std::.

**🏗️ GRAPH CLASS DEFINITION**

class Graph {

private:

unordered\_map<int, vector<int>> adj;

**🧱 Explanation:**

* adj is an **adjacency list**, stored as:
  + Key: A node (integer).
  + Value: A vector of its connected neighbors.

This is a space-efficient way to represent a graph, especially when the graph is **sparse** (few edges).

**🔗 ADD EDGE FUNCTION**

void addEdge(int u, int v) {

adj[u].push\_back(v);

adj[v].push\_back(u);

}

**📌 Explanation:**

* Adds an edge between node u and node v.
* Since the graph is **undirected**, we add:
  + v to u’s list.
  + u to v’s list.

💡 For example, addEdge(1, 3) makes:

* adj[1] = [3]
* adj[3] = [1]

**🔍 DFS UTIL FUNCTION (Recursive)**

void dfsUtil(int node, unordered\_set<int>& visited) {

visited.insert(node);

cout << node << " ";

for (int neighbor : adj[node]) {

if (visited.find(neighbor) == visited.end()) {

dfsUtil(neighbor, visited);

}

}

}

**🎯 Purpose:**

To recursively visit all nodes reachable from node using Depth First Search.

**🔄 Step-by-Step:**

1. visited.insert(node);
   * Mark the current node as visited so we don’t visit it again.
2. cout << node << " ";
   * Output the current node (in the order it's visited).
3. for (int neighbor : adj[node])
   * Loop through all neighbors of the current node.
4. if (visited.find(neighbor) == visited.end())
   * Check if the neighbor hasn’t been visited.
   * If **not visited**, call dfsUtil recursively.

**💡 Behavior:**

* DFS dives **deep** into each branch before backtracking.
* Uses the **call stack** for tracking traversal path.

**🚀 DFS FUNCTION (Main Entry Point)**

void dfs(int start) {

unordered\_set<int> visited;

cout << "DFS Traversal:\n";

dfsUtil(start, visited);

cout << endl;

}

**🎯 Purpose:**

Public interface for performing DFS from a given start node.

**🔄 What It Does:**

1. Creates an empty set visited to keep track of visited nodes.
2. Prints "DFS Traversal:"
3. Calls the recursive helper dfsUtil(start, visited)
4. Prints a newline after traversal is done.

**🚂 BFS FUNCTION**

void bfs(int start) {

unordered\_set<int> visited;

queue<int> q;

visited.insert(start);

q.push(start);

cout << "BFS Traversal:\n";

while (!q.empty()) {

int node = q.front();

q.pop();

cout << node << " ";

for (int neighbor : adj[node]) {

if (visited.find(neighbor) == visited.end()) {

visited.insert(neighbor);

q.push(neighbor);

}

}

}

cout << endl;

}

**🎯 Purpose:**

Performs **Breadth First Search** starting from node start.

**🔄 Step-by-Step:**

1. unordered\_set<int> visited; – To mark visited nodes.
2. queue<int> q; – Queue to process nodes level by level.
3. visited.insert(start); – Mark starting node as visited.
4. q.push(start); – Begin BFS from the starting node.

**📘 BFS Traversal:**

While the queue is not empty:

1. int node = q.front(); q.pop(); – Get the current node from front of the queue.
2. cout << node << " "; – Print it.
3. Loop through its neighbors:
   * If neighbor is unvisited:
     + Mark it visited.
     + Add it to the queue.

**💡 Behavior:**

* Visits all neighbors of a node **before moving deeper**.
* Uses a **queue** (FIFO), unlike DFS which uses recursion (call stack).

**🧪 MAIN FUNCTION**

int main() {

Graph g;

// Add edges to the graph

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(1, 4);

g.addEdge(2, 5);

g.addEdge(2, 6);

g.dfs(0);

g.bfs(0);

return 0;

}

**🔧 What This Does:**

1. Creates a Graph object g.
2. Adds these edges:

0

/ \

1 2

/ \ / \

3 4 5 6

This is a binary-tree-like undirected graph.

1. Calls g.dfs(0);
   * Starts DFS from node 0.
   * Output will depend on the structure and traversal order of neighbors.
2. Calls g.bfs(0);
   * Starts BFS from node 0.
   * Visits level by level: 0 → 1 2 → 3 4 5 6

**🧾 SAMPLE OUTPUT**

DFS Traversal:

0 1 3 4 2 5 6

BFS Traversal:

0 1 2 3 4 5 6

(Exact DFS order might differ slightly depending on adjacency list ordering, which is based on insertion.)

**🔚 IN SUMMARY:**

| **Feature** | **DFS** | **BFS** |
| --- | --- | --- |
| **Approach** | Depth-first (go deep first) | Breadth-first (explore level by level) |
| **Data Structure** | Recursion (Call Stack) | Queue |
| **Use Case** | Pathfinding, Topological Sort | Shortest path in unweighted graph |
| **Time Complexity** | O(V + E) | O(V + E) |
| **Space Complexity** | O(V) | O(V) |

Would you like me to provide a graphical visualization of DFS and BFS traversal on this exact graph?

**✅ PART 1: Expected Viva Questions (Based on Your Code)**

**1. What is the purpose of the addEdge(int u, int v) function?**

**Answer:**  
This function adds an undirected edge between two nodes u and v in the graph. It updates the adjacency list so that v becomes a neighbor of u and vice versa.

**2. How does the DFS algorithm work in your code?**

**Answer:**  
DFS is implemented recursively in the dfsUtil function. It:

* Starts from a given node,
* Marks it as visited,
* Recursively visits all unvisited neighbors of that node.

**3. Why do we use unordered\_set<int> visited in DFS and BFS?**

**Answer:**  
The visited set is used to keep track of all the nodes that have already been visited during traversal. This prevents visiting the same node again and avoids infinite loops in cyclic graphs.

**4. How is BFS implemented in your code?**

**Answer:**  
BFS uses a queue. Starting from the source node, it:

* Enqueues the node,
* Marks it as visited,
* Then processes each node level by level,
* Visiting all its unvisited neighbors and adding them to the queue.

**5. Why is DFS implemented using recursion but BFS using a queue?**

**Answer:**  
DFS is naturally implemented using recursion (or a stack) because it dives deep into a node’s neighbors before backtracking. BFS is level-order and requires a queue to maintain the order of visiting nodes.

**6. Is the graph directed or undirected? How can you tell?**

**Answer:**  
The graph is undirected. This is evident from the addEdge function where both adj[u].push\_back(v) and adj[v].push\_back(u) are called, meaning both directions are stored.

**7. What will happen if you don’t check if a node is visited before recursing in DFS?**

**Answer:**  
It may lead to **infinite recursion** or visiting the same node multiple times, especially if the graph has cycles.

**8. What is the time complexity of DFS and BFS in your code?**

**Answer:**  
Both DFS and BFS have time complexity **O(V + E)**, where V is the number of vertices and E is the number of edges, since every node and edge is visited once.

**9. What would happen if the graph had disconnected components?**

**Answer:**  
If DFS or BFS is called from one node, it will only visit the **connected component** of that node. To visit the entire graph, you’d need to run DFS/BFS from all unvisited nodes.

**10. Can you write DFS iteratively using a stack instead of recursion?**

**Answer:**  
Yes, DFS can be implemented iteratively using a stack by simulating the call stack used in recursion.

**✅ PART 2: Basic Graph and Algorithm Questions**

**1. What is a graph?**

**Answer:**  
A graph is a non-linear data structure consisting of nodes (vertices) and edges. It can be directed or undirected and may or may not contain cycles.

**2. What is the difference between DFS and BFS?**

**Answer:**

| **Feature** | **DFS** | **BFS** |
| --- | --- | --- |
| Data Structure | Stack/Recursion | Queue |
| Traversal | Depth-wise | Level-wise |
| Use Case | Topological sort, pathfinding | Shortest path in unweighted graphs |

**3. What is an adjacency list?**

**Answer:**  
An adjacency list is a graph representation where each node stores a list of its neighbors. It is memory-efficient, especially for sparse graphs.

**4. What is the difference between an adjacency list and an adjacency matrix?**

**Answer:**

| **Feature** | **Adjacency List** | **Adjacency Matrix** |
| --- | --- | --- |
| Storage | O(V + E) | O(V²) |
| Ideal for | Sparse Graphs | Dense Graphs |
| Access time | Slower for edge checks | Faster for edge checks |

**5. Can DFS and BFS be used to detect cycles in a graph?**

**Answer:**  
Yes.

* In **undirected graphs**, cycles can be detected in DFS if a visited node is found again and it's not the parent node.
* In **directed graphs**, both DFS and BFS (using Kahn's algorithm) can be used with modifications.

**6. What is the space complexity of DFS and BFS?**

**Answer:**

* **DFS:** O(V) due to recursion stack and visited set.
* **BFS:** O(V) due to queue and visited set.

**7. How can you modify the code to handle disconnected graphs?**

**Answer:**  
Wrap the DFS/BFS call inside a loop that runs for all nodes. If a node hasn't been visited, start a new DFS/BFS from that node.

for (int i = 0; i < total\_nodes; ++i) {

if (visited.find(i) == visited.end()) {

dfsUtil(i, visited);

}

}

**8. What are some real-life applications of DFS and BFS?**

**Answer:**

* **DFS:**
  + Solving puzzles (like mazes)
  + Detecting cycles
  + Topological sorting
* **BFS:**
  + Shortest path (like Google Maps)
  + Web crawling
  + Network broadcast

**9. What is the difference between a tree and a graph?**

**Answer:**

| **Feature** | **Tree** | **Graph** |
| --- | --- | --- |
| Cycles | No | Can have cycles |
| Connected | Always | Not always |
| Edges | V - 1 | ≤ V(V - 1)/2 |
| Structure | Hierarchical | Non-hierarchical |

**10. What happens in DFS if the graph contains a cycle?**

**Answer:**  
If cycles exist and visited nodes are not tracked, DFS may enter an **infinite loop**. This is why the visited set is essential.