```
1)vector<vector<int>> adj(n);

2)unordered_map< int, list<int> > adj;
```

what is similarites and difference between this for the graph explain step by step

Both `vector<vector<int>>` and `unordered_map<int, list<int>>` are used to represent graphs in C++, but they have different structures and use cases. Let's explore their similarities and differences step by step.

1. `vector<vector<int>>`

Structure:

- **Type**: `vector<vector<int>>`
- **Description**: A vector of vectors. Each element in the outer vector represents a node, and the inner vectors contain the list of adjacent nodes (neighbors) for that node.

Example:

For a graph with 4 nodes and edges:

```
0 - 1
|   |
3 - 2
```

The adjacency list representation using `vector<vector<int>>` would look like this:

```
vector<vector<int>> adj(4); // 4 nodes, indexed from 0 to 3
adj[0] = {1, 3}; // Node 0 is connected to 1 and 3
adj[1] = {0, 2}; // Node 1 is connected to 0 and 2
adj[2] = {1, 3}; // Node 2 is connected to 1 and 3
adj[3] = {0, 2}; // Node 3 is connected to 0 and 2
```

Operations:

- **Accessing neighbors**: Fast, as it uses direct indexing (`O(1)`).
- **Adding edges**: Requires updating the inner vector of the respective nodes.
- **Memory Usage**: Fixed size determined by the number of nodes.
- **Lookup**: For a node's neighbors, it's straightforward, but accessing neighbors for large graphs could be less efficient due to potential resizing or shifting operations if the graph is dynamic.

Use Case:

- Useful when the number of nodes is known in advance and the graph is relatively static (i.e., not frequently changing in size).

2. `unordered_map<int, list<int>>`

Structure:

- **Type**: `unordered_map<int, list<int>>`
- **Description**: An unordered map where each key is a node, and the value is a list of adjacent nodes. This provides a more flexible representation where nodes don't need to be indexed contiguously.

Example:

For the same graph:

```
unordered_map<int, list<int>> adj;
adj[0] = {1, 3};
adj[1] = {0, 2};
adj[2] = {1, 3};
adj[3] = {0, 2};
```

Operations:

- **Accessing neighbors**: Efficient, but involves hash table lookups which are typically `O(1)` on average.
- **Adding edges**: Adding or removing nodes and edges is more flexible and efficient for sparse graphs or when nodes can be added/removed dynamically.
- **Memory Usage**: More flexible as it doesn't require a fixed number of nodes. However, hash tables and lists have overheads compared to vectors.
- **Lookup**: Fast for individual nodes due to hash-based indexing. Efficient for dynamically changing graphs.

Use Case:

- Useful when the graph is sparse, or the number of nodes is not known in advance, or when nodes are dynamically added or removed. It is also beneficial when node indices are not consecutive integers.

Comparison

Similarities:

- **Purpose**: Both represent adjacency lists of a graph, storing which nodes are connected to which.
- **Edge Addition**: Both can be used to add edges between nodes.

Differences:

1. **Structure**:

- `vector<vector<int>>`: Indexed by node numbers directly, suitable for fixed-size or dense graphs.
- `unordered_map<int, list<int>>`: Uses node identifiers (keys) as indices, suitable for dynamic or sparse graphs.

2. **Access Speed**:
   - `vector<vector<int>>`: Direct access to nodes via indices (`O(1)`), but resizing can be costly.
   - `unordered_map<int, list<int>>`: Access via hash table (`O(1)` on average), better for non-contiguous indices.
3. **Memory Usage**:
   - `vector<vector<int>>`: Can be more memory efficient if nodes are densely packed.
   - `unordered_map<int, list<int>>`: More overhead due to hash table and list, but flexible for dynamic graphs.
4. **Flexibility**:
   - `vector<vector<int>>`: Fixed size; resizing and adding nodes require reallocation.
   - `unordered_map<int, list<int>>`: Dynamic size; nodes and edges can be added/removed easily.

Summary

- Use `vector<vector<int>>` when dealing with a fixed number of nodes, especially if nodes are consecutively numbered.
- Use `unordered_map<int, list<int>>` when nodes are dynamically added or removed, or if node indices are not continuous.

Each representation has its advantages and is suited to different types of graph-related problems.

Logical Operator(!visted[0])

In C++, `!visited[3]` is a boolean expression that checks whether the value at index `3` of the `visited` array (or vector) is `false`.

Here's a detailed breakdown:

- **`visited`**: This is typically a boolean array or vector used to track whether nodes in a graph have been visited.
- **`visited[3]`**: This accesses the value at index `3` of the `visited` array/vector. If `visited` is a boolean array, this value is either `true` or `false`.
- **`!visited[3]`**: The `!` operator is the logical NOT operator. It inverts the boolean value:
  - If `visited[3]` is `true`, `!visited[3]` evaluates to `false`.
  - If `visited[3]` is `false`, `!visited[3]` evaluates to `true`.

Example in Context

Consider the following example in the BFS algorithm:

```
if (!visited[neighbor]) {
    visited[neighbor] = true;
    q.push(neighbor);
}
```

- **Condition** `!visited[neighbor]`:
  - Checks if `neighbor` has not been visited yet.
  - If `visited[neighbor]` is `false` (i.e., the node has not been visited), `!visited[neighbor]` evaluates to `true`.
- **Action**:
  - If the condition is `true`, it means the `neighbor` has not been visited, so we mark it as visited and enqueue it for further exploration.

Summary

- `!`: The logical NOT operator in C++.
- `!visited[3]`: Checks if the value at index `3` in the `visited` array/vector is `false` (i.e., the node has not been visited).

This expression is commonly used in algorithms like BFS and DFS to determine whether a node has already been visited and, if not, to mark it as visited and proceed with further operations.