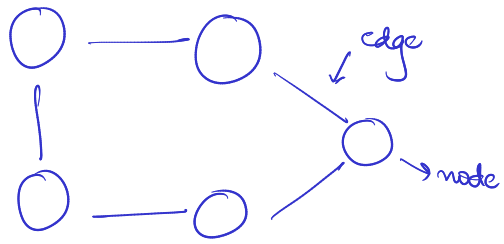
#Graphs

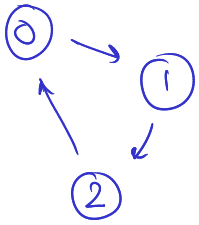


- → Directed
- → Undirected
- → Weighted
- → Non-weighted
- → Cyclic
- → Acyclic
- → Disconnected

Representation of Graph:-

- → Adjacency matrix
- → Adjacency list

	0	1	2
0	0	1	0
1	0	0	1
2	1	0	0



0 → 1, 4
1 → 0, 2, 3
2 → 1, 3
3 → 4, 1, 2
4 → 3, 0

Adj. List

↓

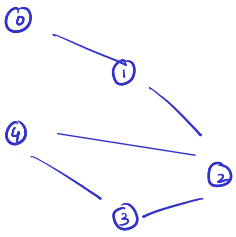
Implementation

↓

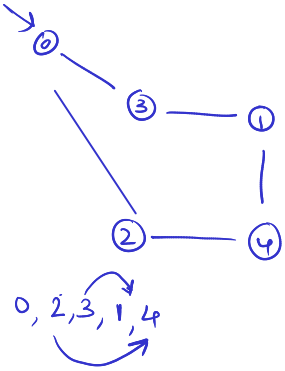
map<int, list<int>>

```cpp
#include <iostream>
#include <map>
#include <list>
using namespace std;

class graph{
public:
    unordered_map<int, list<int>> adj;

    void addEdge(int u, int v, bool direction) {
        // direction = 0 → undirected
        // direction = 1 → directed

        // create an edge from u to v
        adj[u].push_back(v);
        if(direction==0) adj[v].push_back(u);
    }
    void printAdjList() {
        for(auto i:adj) {
            cout<<i.first<<"→ ";
            for(auto j:i.second) {
                cout<<j<<", ";
            }
            cout<<endl;
        }
    }
};

int main() {
    int n;
    cout<<"Enter the number of nodes"<<endl;
    cin>>n;
    int m;
    cout<<"Enter the number of edges"<<endl;
    cin>>m;
    graph g;
    for(int i = 0; i < m; i++) {
        int u, v;
        cin>>u>>v;
        //creating undirected graph
        g.addEdge(u,v,0);
    }
    g.printAdjList();
    return 0;
}
```

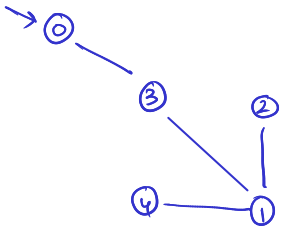
BFS



Print → 0, 1, 2, 3, 4



0, 2, 3, 1, 4



visited node

unordered_map
<node, bool>

0, 3, 1, 2, 4

Ans: ④ 0 3 1 2 4

adj list

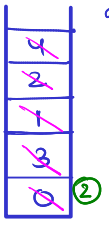
0 → 3
1 → 2, 3, 4
2 → 1
3 → 0, 1
4 → 1

visited

0 → ~~F~~ T
1 → ~~F~~ T
2 → ~~F~~ T
3 → ~~F~~ T ③
4 → ~~F~~ T

⑤

→ Now, after zero is added, push all the adjacent / neighbour nodes to queue.



queue

① frontNode = 0

frontNode = 3

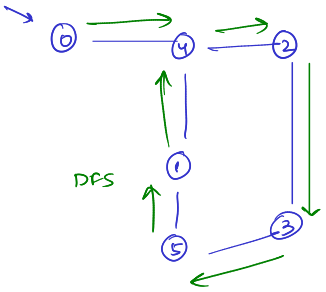
frontNode = 1

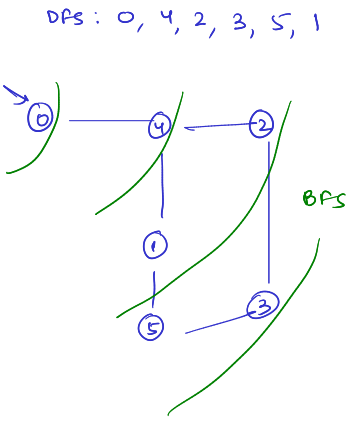
frontNode = 2

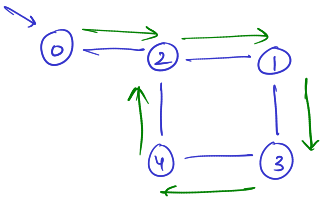
frontNode = 4

```cpp
#include <unordered_map>
#include<list>
#include<set>
#include<queue>

void prepareAdjList(unordered_map<int, list<int>> &adjList, vector<pair<int, int>> edges) {
    for(int i = 0; i < edges.size(); i++) {
        int u = edges[i][0];
        int v = edges[i][1];
        adjList.push_back(v);
        adjList.push_back(u);
    }
}

void bfs(unordered_map<int, list<int>> adjList, unordered_map<int, bool> visited, vector<int> &ans, int node) {
    queue<int> q;
    q.push(node);
    visited[node] = 1;
    while(!q.empty()) {
        int frontNode = q.front();
        q.pop();

        //store frontNode in ans;
        ans.push_back(frontNode);

        // traverse all neighbours of frontNode
        for(auto i: adjList[frontNode]) {
            if(!visited[i]) {
                q.push(i);
                visited[i] = 1;
            }
        }
    }
}

vector<int> BFS(int vertex, vector<pair<int, int>> edges) {
    unordered_map<int, list<int>> adjList;
    vector<int> ans;
    unordered_map<int, bool> visited;

    prepareAdjList(adjList, edges);

    // traverse all component of unconnected graph
    for(int i = 0; i < vertex; i++) {
        if(!visited[i]) bfs(adjList, visited, ans, i);
    }
    return ans;
}
```

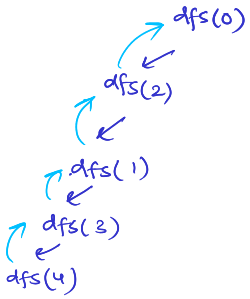
Dfs







0 2 1 3 4

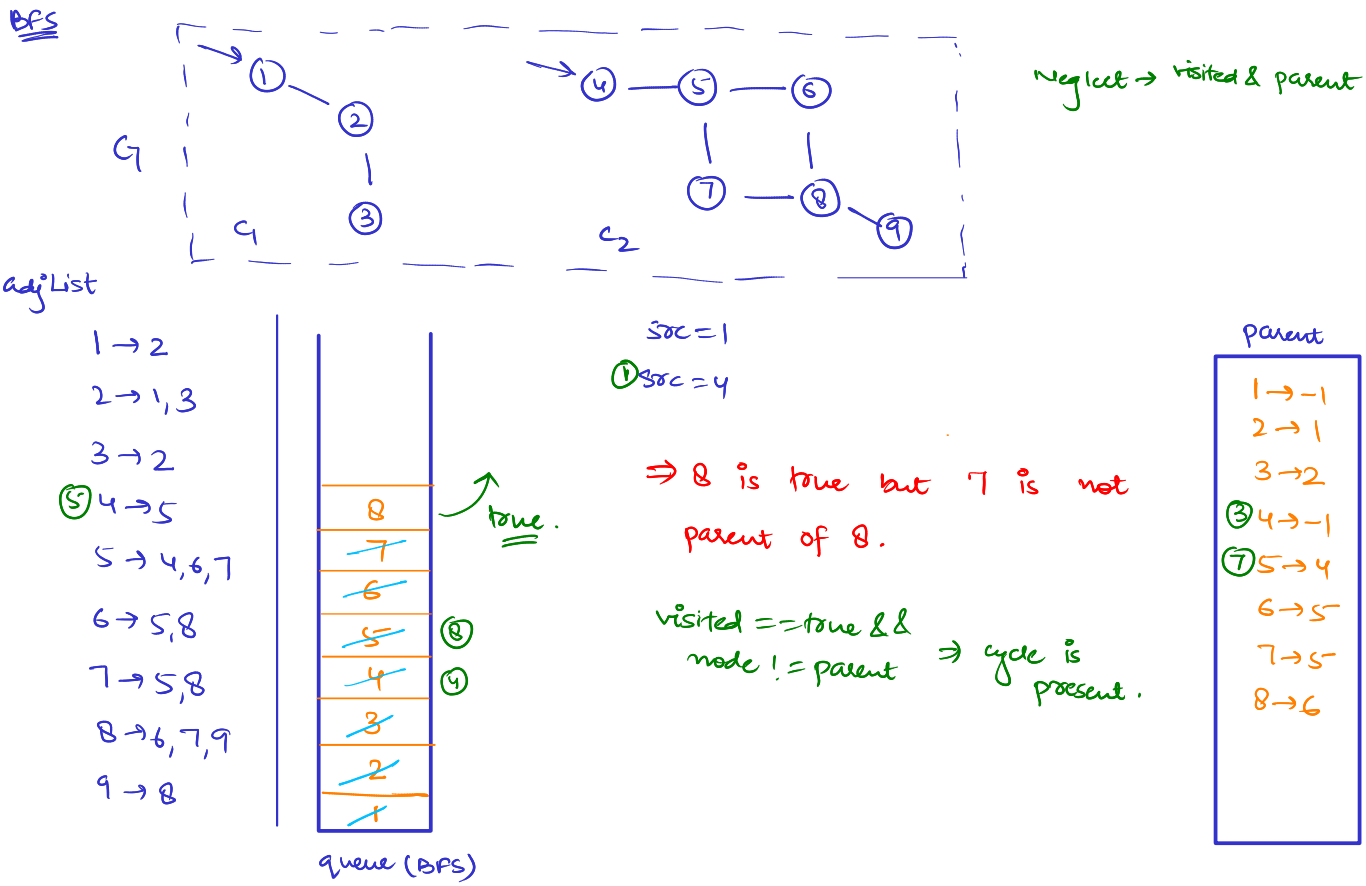



```cpp
#include <bits/stdc++.h>
using namespace std;

void dfs(int start, vector<int> adj[], vector<int> &innerAns, int vis[]) {
  vis[start] = 1;
  innerAns.push_back(start);
  for (auto it : adj[start]) {
    if (!vis[it]) {
      dfs(it, adj, innerAns, vis);
    }
  }
}

vector<vector<int>> depthFirstSearch(int V, int E, vector<vector<int>> &edges) {
  // Creating adjacency list
  vector<int> adj[V];
  for (int it = 0; it < edges.size(); it++) {
    int u = edges[it][0];
    int v = edges[it][1];
    adj[u].push_back(v);
    adj[v].push_back(u);
  }
  // Array for visited mark initially all are 0
  int vis[V] = {0};
  // Answer vetor to store the final ans
  vector<vector<int>> ans;

  // loop through out the vertices and look for unvisited one and then go dipper
  // and push ans for that
  for (int it = 0; it < V; it++) {
    if (!vis[it]) {
      vector<int> innerAns;
      // deeply visit the adjacency one by DFS
      dfs(it, adj, innerAns, vis);
      ans.push_back(innerAns);
    }
  }
  return ans;
}
```

→ If adjacency list is already given.

```
class Solution {
  public:
    void dfs(int i, vector<int> adj[], vector<int> &ans, int vis[]) {
        vis[i] = 1;
        ans.push_back(i);
        for(auto it:adj[i]) {
            if(!vis[it]) dfs(it, adj, ans, vis);
        }
    }

    vector<int> dfsOfGraph(int V, vector<int> adj[]) {
        int vis[V] = {0};
        vector<int> ans;

        for(int i = 0; i < V; i++) {
            if(!vis[i]) {
                dfs(i, adj, ans, vis);
            }
        }
        return ans;
    }
};
```

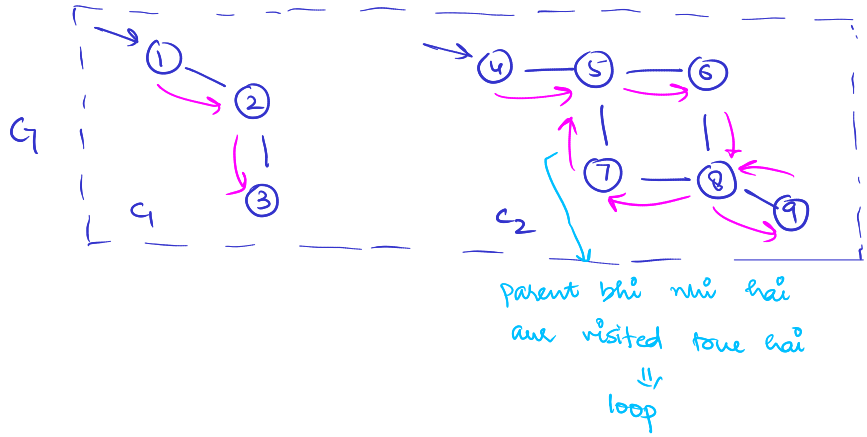
Cycle Detection in Undirected Graph




```cpp
#include <unordered_map>
#include <list>
#include <vector>
#include <queue>
#include <string>
using namespace std;

bool isCyclicBFS(int node, unordered_map<int, bool> &visited,  unordered_map<int, list<int>> &adj) {
  unordered_map<int, int> parent;

  parent[node] = -1;
  visited[node] = true;

  queue<int> q;
  q.push(node);

  while(!q.empty()) {
    int frontNode = q.front();
    q.pop();

    for(auto neighbour: adj[frontNode]) {
      if(visited[neighbour] == true && neighbour ≠ parent[frontNode]) {
        return true;
      }
      else if(!visited[neighbour]) {
        q.push(neighbour);
        parent[neighbour] = frontNode;
        visited[neighbour] = 1;
      }
    }
  }
  return false;
}
```

```cpp
string cycleDetection(vector<vector<int>>& edges, int n, int m)
{
    // Step 1: Create adjacency list
    unordered_map<int, list<int>> adj;
    for(int i = 0; i < m; i++) {
        adj[edges[i][0]].push_back(edges[i][1]);
        adj[edges[i][1]].push_back(edges[i][0]); // Add the reverse direction edge
    }

    // Step 2: Initialize visited map
    unordered_map<int, bool> visited;

    // Step 3: Search through the vertices, if not visited then go deeper
    for(int i = 1; i ⩽ n; i++) {
        if(!visited[i]) {
            bool ans = isCyclicBFS(i, visited, adj);
            if(ans) return "Yes";
        }
    }
    return "No";
}
```

DFS



→ Here also we will use parent & visited.

```cpp
#include <unordered_map>
#include <list>
#include <vector>
#include <string>
using namespace std;

bool isCyclicDFS(int node, int parent, unordered_map<int, bool> &visited, unordered_map<int, list<int>> &adj) {
    visited[node] = true;

    for(auto neighbour: adj[node]) {
        if(!visited[neighbour]) {
            bool cycleDetected = isCyclicDFS(neighbour, node, visited, adj);
            if(cycleDetected) return true;
        } else if(neighbour != parent) return true;
    }
    return false;
}

string cycleDetection(vector<vector<int>>& edges, int n, int m)
{
    // Step 1: Create adjacency list
    unordered_map<int, list<int>> adj;
    for(int i = 0; i < m; i++) {
        adj[edges[i][0]].push_back(edges[i][1]);
        adj[edges[i][1]].push_back(edges[i][0]); // Add the reverse direction edge
    }

    // Step 2: Initialize visited map
    unordered_map<int, bool> visited;

    // Step 3: Search through the vertices, if not visited then go deeper
    for(int i = 1; i <= n; i++) {
        if(!visited[i]) {
            bool ans = isCyclicDFS(i, -1, visited, adj);
            if(ans) return "Yes";
        }
    }
    return "No";
}
```

Cycle Detection in directed graph