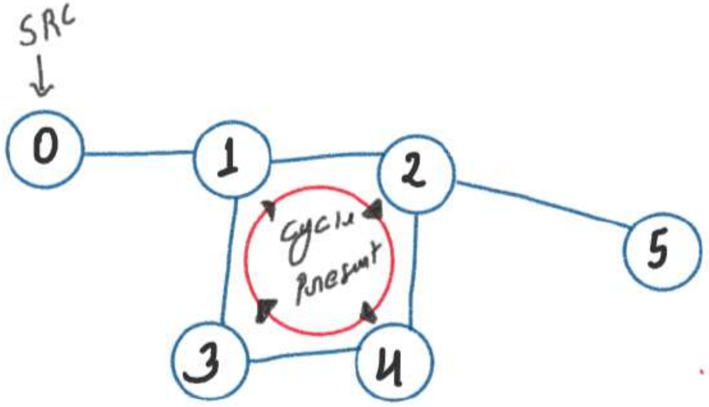
26/01/2024

GRAPHS CLASS - 2

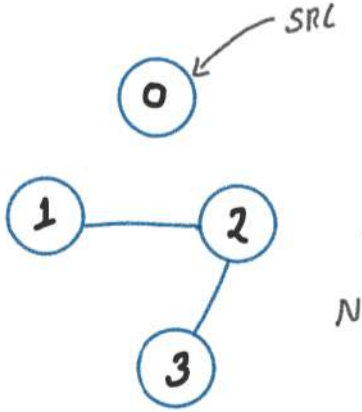
1. Detect cycle in an undirected graph using BFS

EX1



Output TRUE

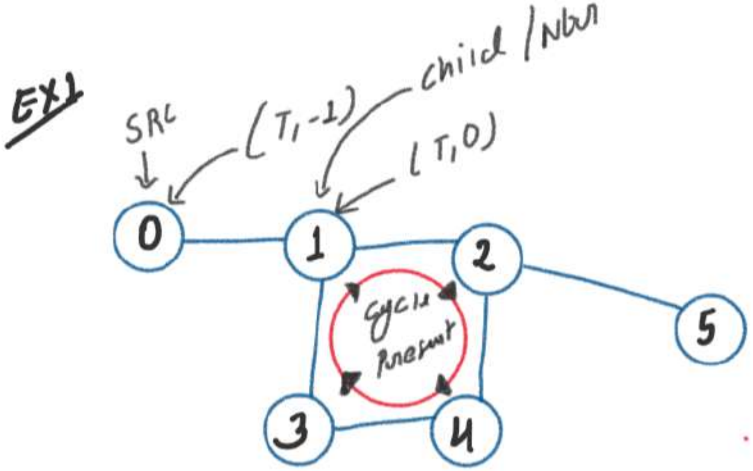
EX2



No cycle present

Output FALSE

LOGIC BUILDING



Intuition: *The intuition behind this is to check for the visited element if it is found again, this means the cycle is present in the given undirected graph.*

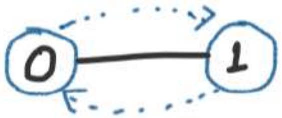
0 ← parent of 0 = -1

1 ← parent of 1 = 0

0 ← parent of 0 = 1

अगर such me 0 Repeat Ho Rha Hai TO 1 का parent Diff.n Hona chaiye 0 से

Check करलो



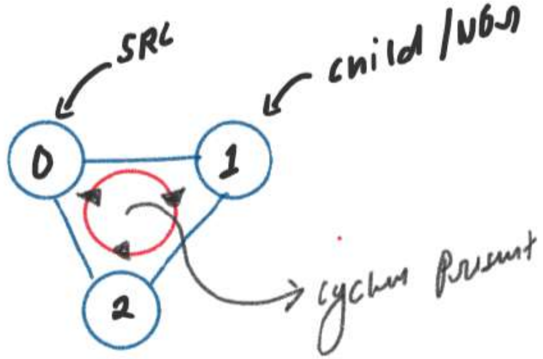
Actual me 0 Repeat Nahi Ho Rha Hai

(1 का child != 1 का parent)

0 != 0

False

EX



Edges

0—1

0—2

1-0

1-2

2-1

2-0

SRC

↳ 0 ⟵ Parent of 0 = -1   visited = T

child ↳ 1 ⟵ Parent of 1 = 0   visited = T

0 ⟵ parent of 0 = 1   0 Already visited

0 != Parent[1]

0 != 0 ✗

2 ⟵ parent of 2 = 1   visited = T

1 ⟵ Parent of 1 = 2   1 Already visited

1 != Parent[2]

1 != 1 ✗

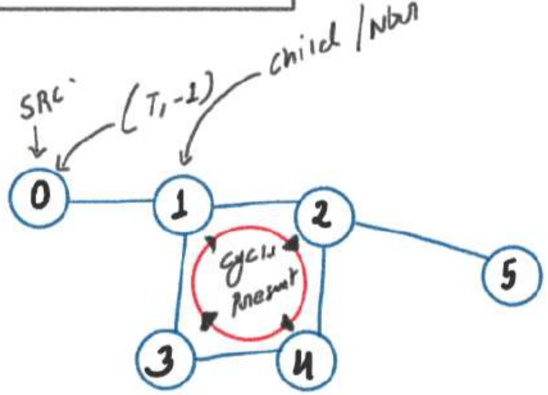
0 ⟵ parent of 0 = 2   0 Already visited

0 != Parent[2]

0 != 1 TRUE → cycle present Hai

DRY RUN EX1

EX1



Initial state

q.push(src)
visited[src] = T
parent[src] = -1

AdjList

key	value
0	{1}
1	{0,2,3}
2	{4,1,5}
3	{1,4}
4	{2,3}
5	{2}

visited

key	value
0	~~F~~ T
1	F
2	F
3	F
4	F
5	F

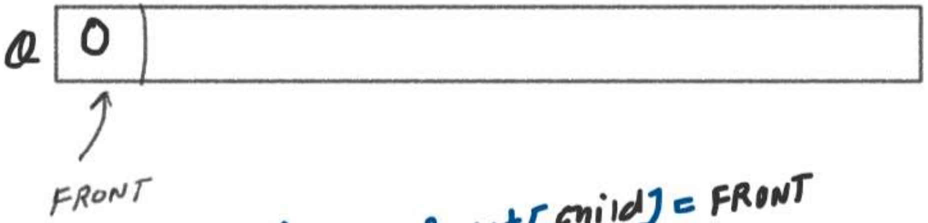
parent

key	value
0	-1
1	
2	
3	
4	
5	

STEP1 SAVE Front and pop it

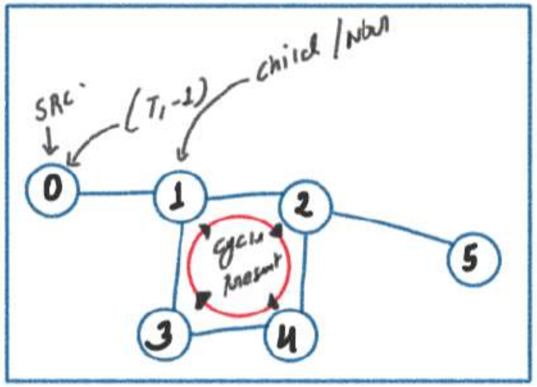
STEP2 Go to AdjList and Get the child of Front

STEP3 Check child is visited OR NOT



- if NOT visited → q.push(child)
  visited[child] = T
  parent[child] = FRONT
- if visited → if(child != parent[Front])
  ↳ cycle present hai

ITERATION 1



STEP1 Front = 0 , q.pop()

STEP2 child = {1}

STEP3

- if NOT visited → q.push(1)

  visited[1] = T

  parent[1] = 0

AdjList

key	value
0	{1}
1	{0,2,3}
2	{4,1,5}
3	{1,4}
4	{2,3}
5	{2}

visited

key	value
0	~~F~~ T
1	~~F~~ T
2	F
3	F
4	F
5	F

parent

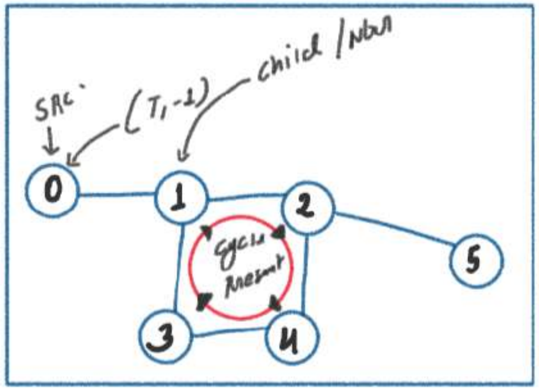
key	value
0	-1
1	0
2	
3	
4	
5	

Q			
	~~0~~	1	

FRONT

PARENT

ITERATION 2



AdjList

Key	value
0	{1}
1	{0,2,3}
2	{4,1,5}
3	{1,4}
4	{2,3}
5	{2}

visited

Key	value
0	~~F~~ T
1	~~F~~ T
2	~~F~~ T
3	~~F~~ T
4	F
5	F

parent

Key	value
0	-1
1	0
2	1
3	1
4	
5	

STEP1 Front = 1 , q.pop()

STEP2 child = {0,2,3}

Q

~~1~~	2	3	

FRONT PARENT

STEP3

child = 0

- if visited → if(0 != parent[1])
  0 != 0 ✗

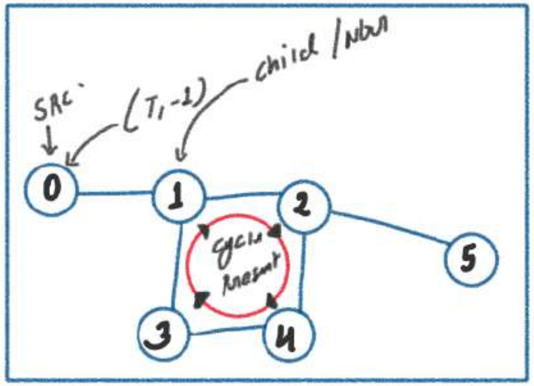
child = 2

- if NOT visited → q.push(2)
  visited[2] = T
  parent[2] = 1

child = 3

- if NOT visited → q.push(3)
  visited[3] = T
  parent[3] = 1

ITERATION 3



Adjlist

key	value
0	{1}
1	{0,2,3}
2	{4,1,5}
3	{1,4}
4	{2,3}
5	{2}

visited

key	value
0	~~F~~ T
1	~~F~~ T
2	~~F~~ T
3	~~F~~ T
4	~~F~~ T
5	~~F~~ T

parent

key	value
0	-1
1	0
2	1
3	1
4	2
5	2

STEP1 Front = 2 , q.pop()

STEP2 child = {4, 1,5}

Q

~~2~~	3	4	5	

FRONT PARENT

STEP3

child = 1

- if visited → if(1 != parent[2])

  1 != 1 ×

child = 4

- if NOT visited → q.push(4)

  visited[4] = T

  parent[4] = 2

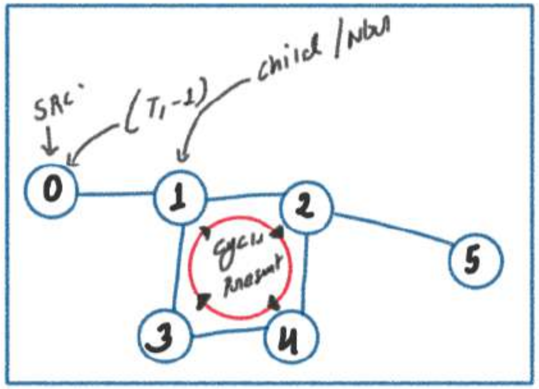
child = 5

- if NOT visited → q.push(5)

  visited[5] = T

  parent[5] = 2

ITERATION 4



Adjlist

Key	value
0	{ 1 }
1	{0,2,3}
2	{4,1,5}
3	{1,4}
4	{2,3}
5	{ 2 }

visited

Key	value
0	~~F~~ T
1	~~F~~ T
2	~~F~~ T
3	~~F~~ T
4	~~F~~ T
5	~~F~~ T

parent

Key	value
0	-1
1	0
2	1
3	1
4	2
5	2

STEP1 Front = 3 , q.pop()

STEP2 child = { 1,4 }

STEP3

child = 1

- if visited → if(1 != parent[3])

  1 != 1 x

child = 4

- if visited → if(4 != parent[3])

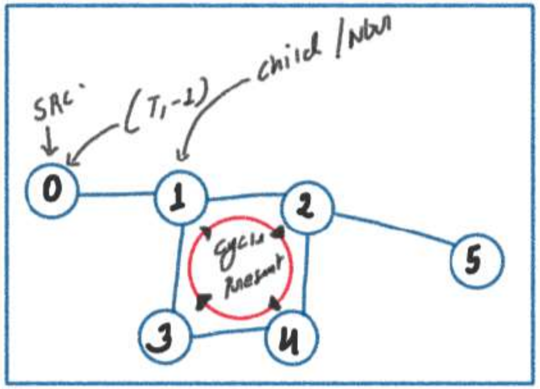
  4 != 1 x

Q				
Q	~~3~~	4	5	

FRONT

PARENT

ITERATION 5



Adjlist

Key	value
0	{ 1 }
1	{0,2,3}
2	{4,1,5}
3	{1,4}
4	{2,3}
5	{ 2 }

visited

Key	value
0	~~F~~ T
1	~~F~~ T
2	~~F~~ T
3	~~F~~ T
4	~~F~~ T
5	~~F~~ T

parent

Key	value
0	-1
1	0
2	1
3	1
4	2
5	2

STEP1 Front = 4 , q.pop()

STEP2 child = { 2,3 }

Q	~~4~~	5

FRONT PARENT

STEP3

child=2

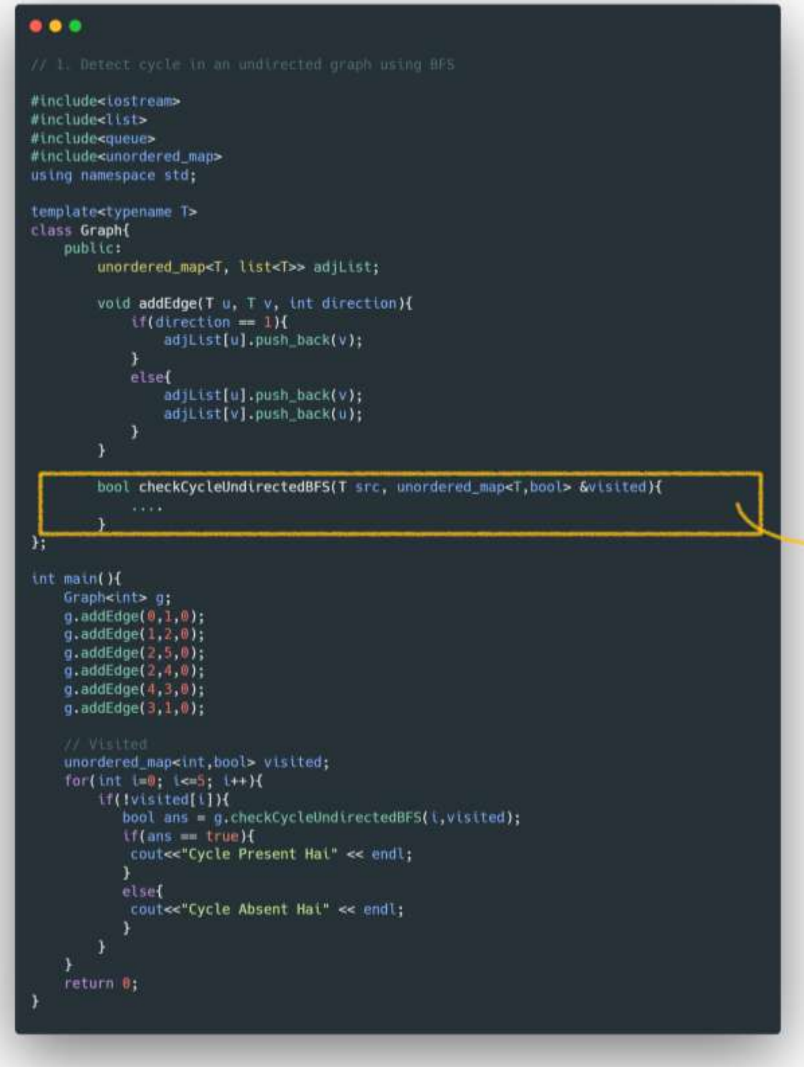
- if visited → if(2 != parent(4))

  2 != 2 ×

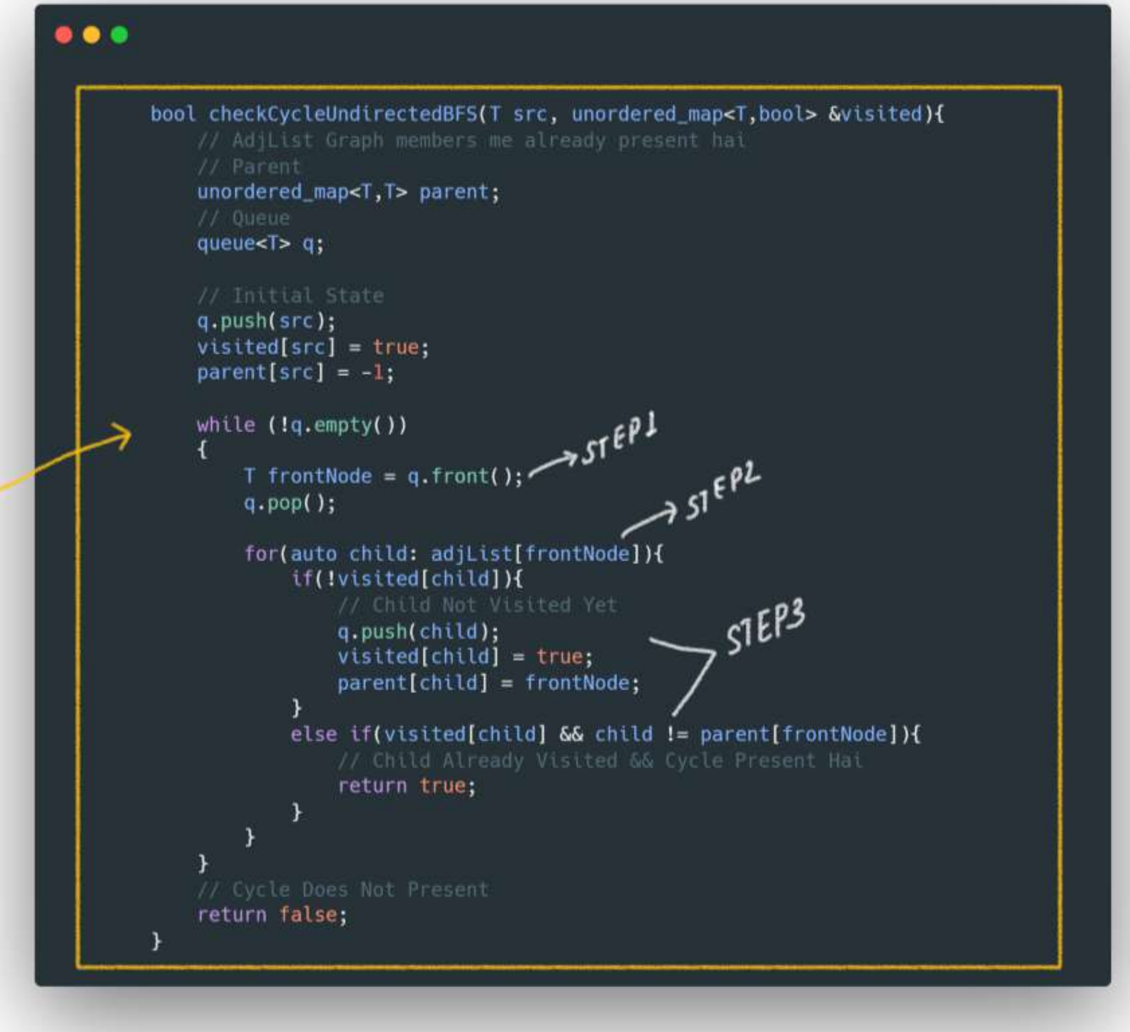
child=3

- if visited → if(3 != parent(4))

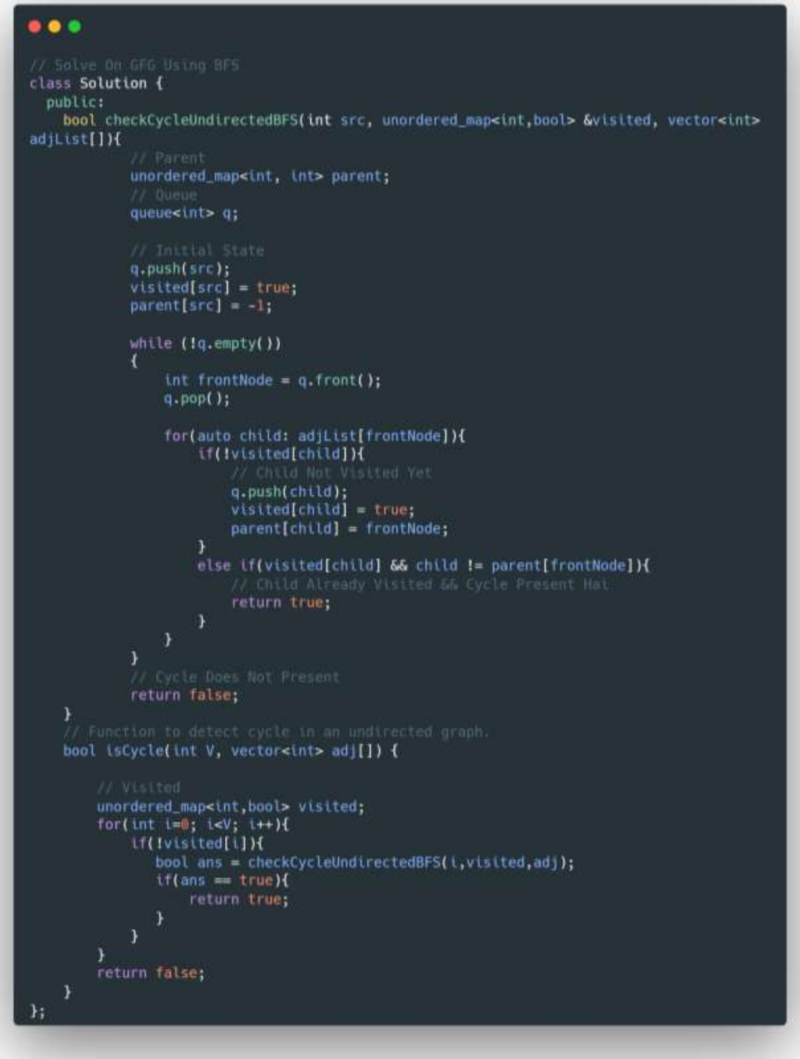
  3 != 2 TRUE → RETURN TRUE CYCLE PRESENT 🙂



```cpp
// 1. Detect cycle in an undirected graph using BFS

#include<iostream>
#include<list>
#include<queue>
#include<unordered_map>
using namespace std;

template<typename T>
class Graph{
    public:
        unordered_map<T, list<T>> adjList;

        void addEdge(T u, T v, int direction){
            if(direction == 1){
                adjList[u].push_back(v);
            }
            else{
                adjList[u].push_back(v);
                adjList[v].push_back(u);
            }
        }

        bool checkCycleUndirectedBFS(T src, unordered_map<T,bool> &visited){
            ....
        }
};

int main(){
    Graph<int> g;
    g.addEdge(0,1,0);
    g.addEdge(1,2,0);
    g.addEdge(2,5,0);
    g.addEdge(2,4,0);
    g.addEdge(4,3,0);
    g.addEdge(3,1,0);

    // Visited
    unordered_map<int,bool> visited;
    for(int i=0; i<=5; i++){
        if(!visited[i]){
           bool ans = g.checkCycleUndirectedBFS(i,visited);
           if(ans == true){
            cout<<"Cycle Present Hai" << endl;
           }
           else{
            cout<<"Cycle Absent Hai" << endl;
           }
        }
    }
    return 0;
}
```



```cpp
bool checkCycleUndirectedBFS(T src, unordered_map<T,bool> &visited){
    // AdjList Graph members me already present hai
    // Parent
    unordered_map<T,T> parent;
    // Queue
    queue<T> q;

    // Initial State
    q.push(src);
    visited[src] = true;
    parent[src] = -1;

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

        for(auto child: adjList[frontNode]){
            if(!visited[child]){
                // Child Not Visited Yet
                q.push(child);
                visited[child] = true;
                parent[child] = frontNode;
            }
            else if(visited[child] && child != parent[frontNode]){
                // Child Already Visited && Cycle Present Hai
                return true;
            }
        }
    }
    // Cycle Does Not Present
    return false;
}
```

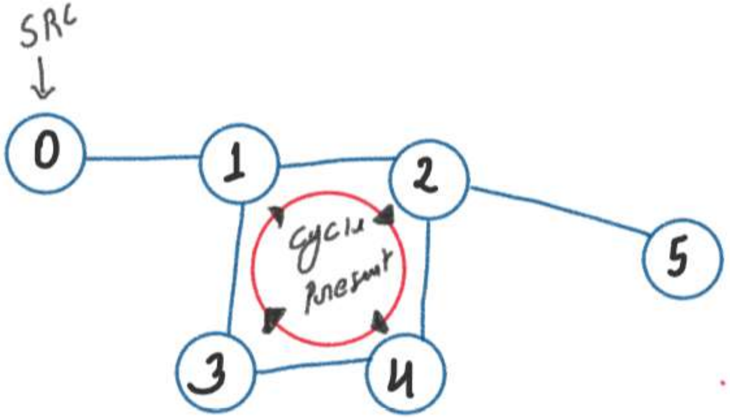


```
// Solve On GFG Using BFS
class Solution {
  public:
    bool checkCycleUndirectedBFS(int src, unordered_map<int,bool> &visited, vector<int>
adjList[]){
            // Parent
            unordered_map<int, int> parent;
            // Queue
            queue<int> q;

            // Initial State
            q.push(src);
            visited[src] = true;
            parent[src] = -1;

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

                for(auto child: adjList[frontNode]){
                    if(!visited[child]){
                        // Child Not Visited Yet
                        q.push(child);
                        visited[child] = true;
                        parent[child] = frontNode;
                    }
                    else if(visited[child] && child != parent[frontNode]){
                        // Child Already Visited && Cycle Present Hai
                        return true;
                    }
                }
            }
            // Cycle Does Not Present
            return false;
    }
    // Function to detect cycle in an undirected graph.
    bool isCycle(int V, vector<int> adj[]) {

        // Visited
        unordered_map<int,bool> visited;
        for(int i=0; i<V; i++){
            if(!visited[i]){
               bool ans = checkCycleUndirectedBFS(i,visited,adj);
               if(ans == true){
                   return true;
               }
            }
        }
        return false;
    }
};
```

Time and space complexity = ?

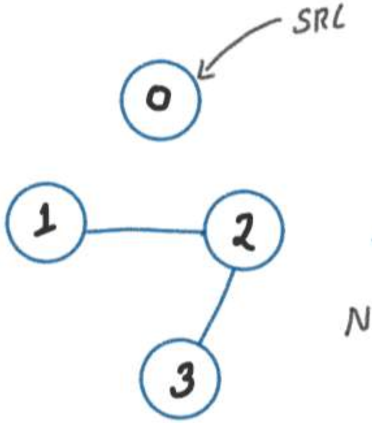
📂 2. Detect cycle in an undirected graph using DFS

EX1



OUTPUT=TRUE

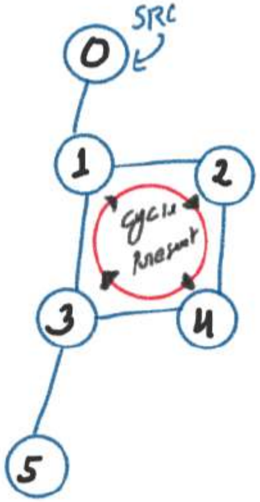
EX2



No cycle Present

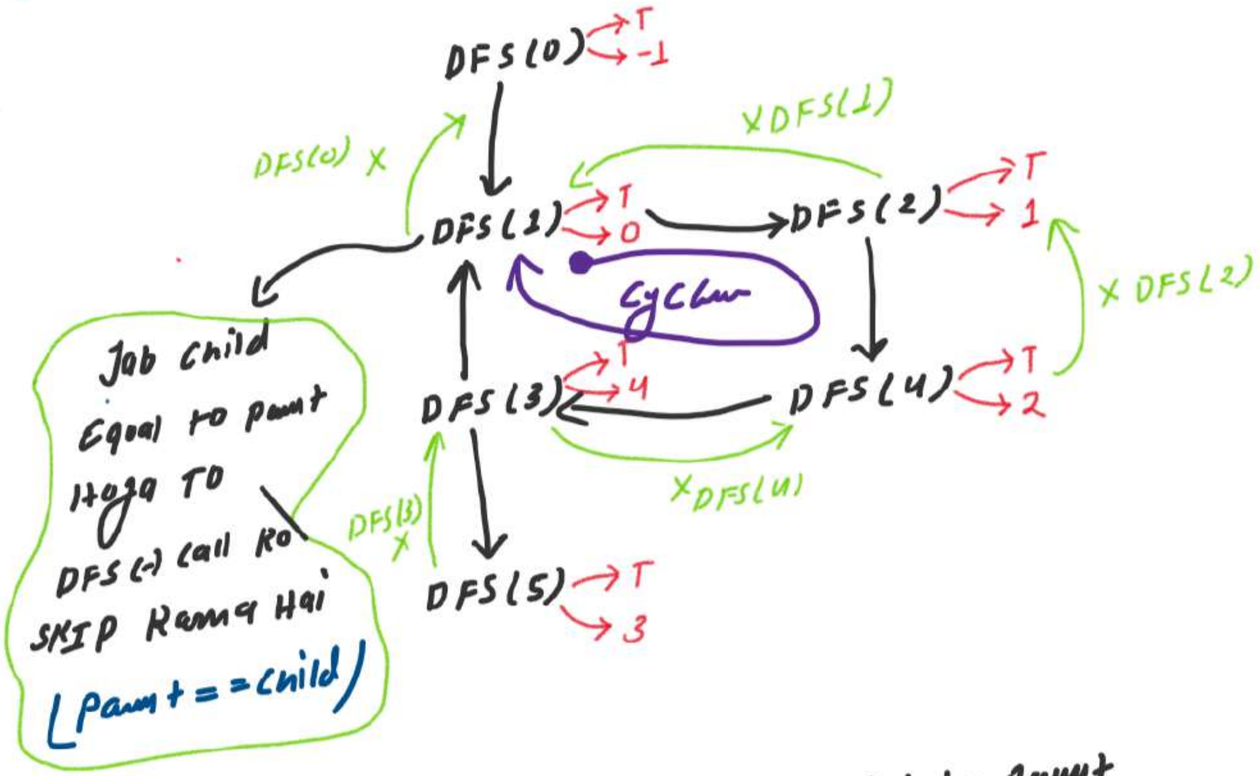
OUTPUT=FALSE

Ex1



SEquence of DFS calls

1 DFS(0)
2 DPS(1)
3 DFS(0) X
4 DFS(2)
5 DFS(1) X
6 DFS(4)
7 DFS(2) X
8 DFS(3)
9 DFS(4) X
10 DFS(5)
11 DFS(3) X
12 DFS(1) → 1 is Already visited && child != parent
1 3
↳ return TRUE



Jab child Equal to parent Hoga To DFS(-) call Ko SKIP Karna Hai

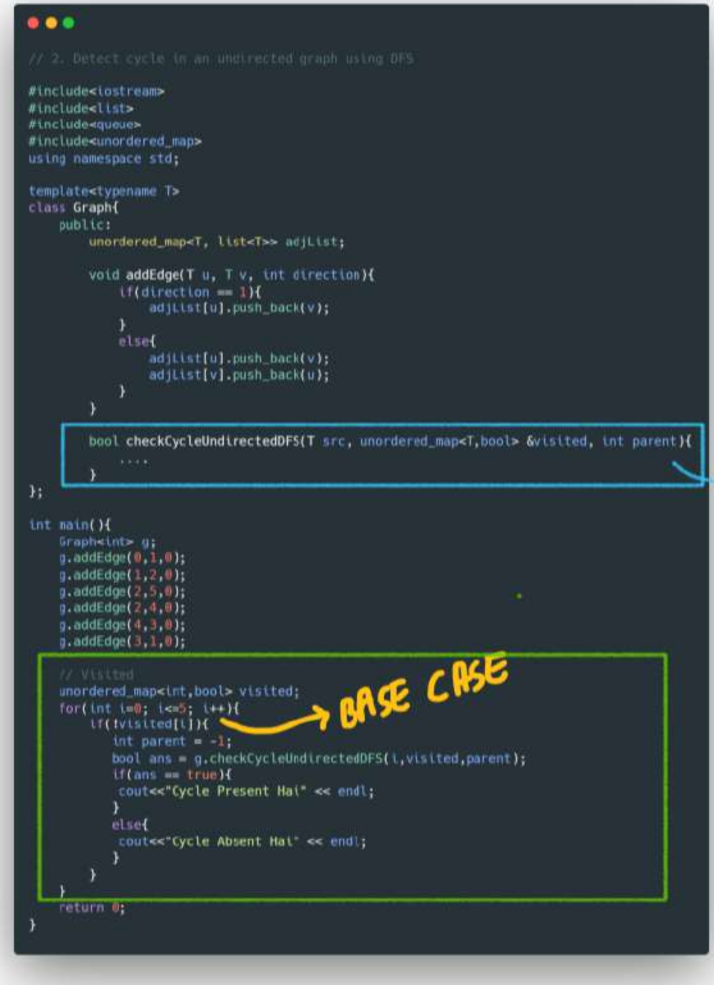
(Parent == child)

Adjlist

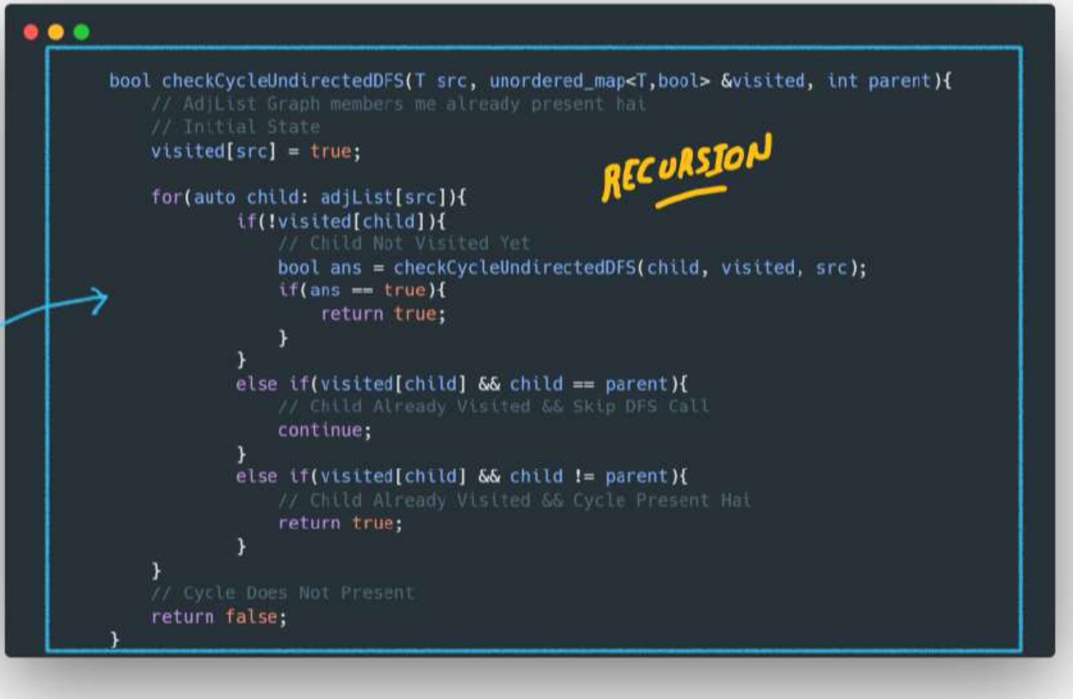
Key	value
0	{1}
1	{0,2,3}
2	{4,1}
3	{1,4,5}
4	{2,3}
5	{3}

visited

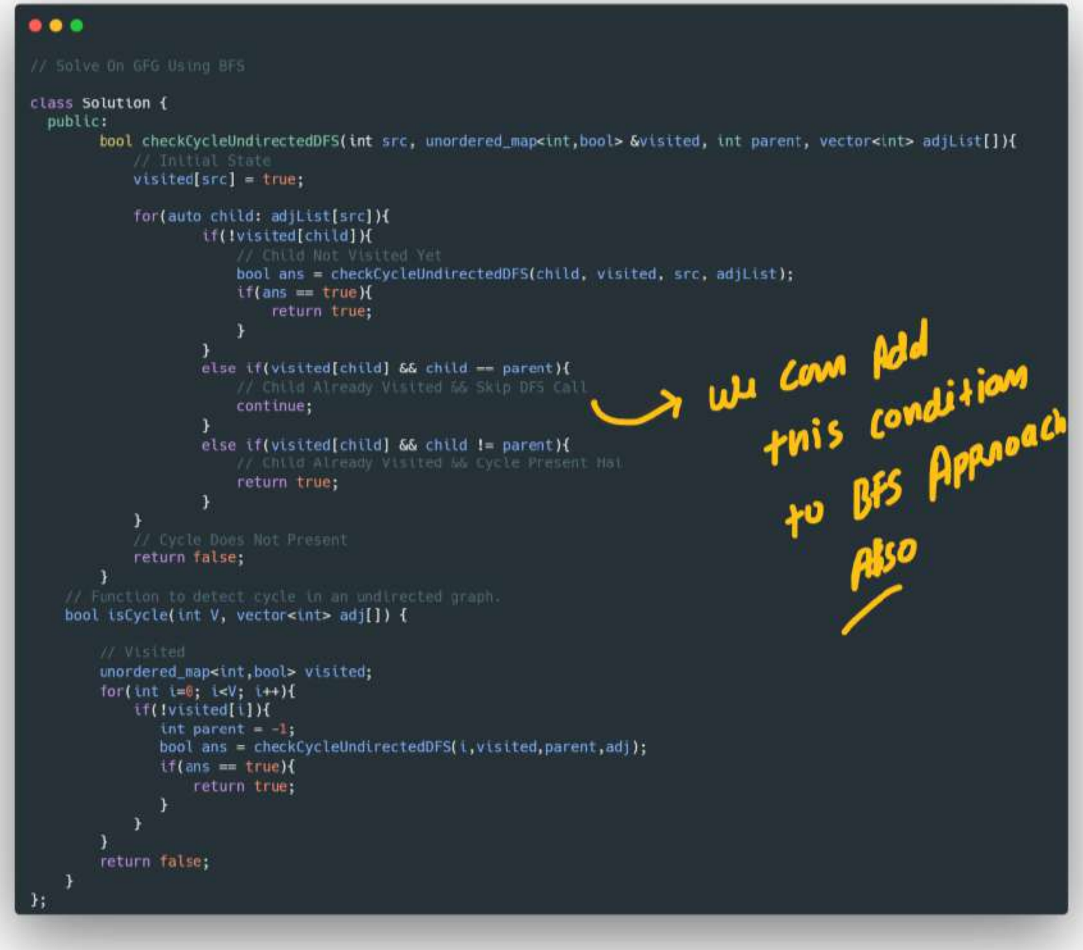
Key	value
0	~~F~~ T
1	F
2	F
3	F
4	F
5	F



```cpp
// 2. Detect cycle in an undirected graph using DFS

#include<iostream>
#include<list>
#include<queue>
#include<unordered_map>
using namespace std;

template<typename T>
class Graph{
    public:
        unordered_map<T, list<T>> adjList;

        void addEdge(T u, T v, int direction){
            if(direction == 1){
                adjList[u].push_back(v);
            }
            else{
                adjList[u].push_back(v);
                adjList[v].push_back(u);
            }
        }

        bool checkCycleUndirectedDFS(T src, unordered_map<T,bool> &visited, int parent){
            ....
        }
};

int main(){
    Graph<int> g;
    g.addEdge(0,1,0);
    g.addEdge(1,2,0);
    g.addEdge(2,5,0);
    g.addEdge(2,4,0);
    g.addEdge(4,3,0);
    g.addEdge(3,1,0);

    // Visited
    unordered_map<int,bool> visited;
    for(int i=0; i<=5; i++){
        if(!visited[i]){
            int parent = -1;
            bool ans = g.checkCycleUndirectedDFS(i,visited,parent);
            if(ans == true){
             cout<<"Cycle Present Hai" << endl;
            }
            else{
             cout<<"Cycle Absent Hai" << endl;
            }
        }
    }
    return 0;
}
```



```cpp
bool checkCycleUndirectedDFS(T src, unordered_map<T,bool> &visited, int parent){
    // AdjList Graph members me already present hai
    // Initial State
    visited[src] = true;

    for(auto child: adjList[src]){
            if(!visited[child]){
                // Child Not Visited Yet
                bool ans = checkCycleUndirectedDFS(child, visited, src);
                if(ans == true){
                    return true;
                }
            }
            else if(visited[child] && child == parent){
                // Child Already Visited && Skip DFS Call
                continue;
            }
            else if(visited[child] && child != parent){
                // Child Already Visited && Cycle Present Hai
                return true;
            }
    }
    // Cycle Does Not Present
    return false;
}
```



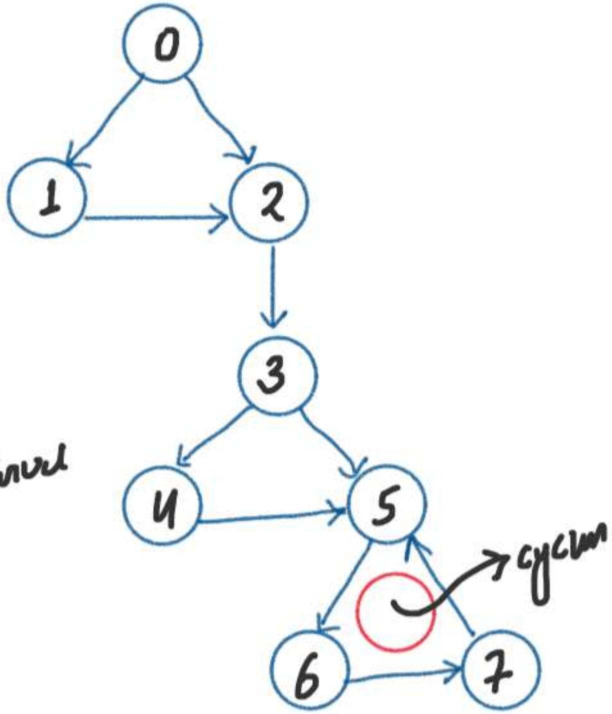
```cpp
// Solve On GFG Using BFS

class Solution {
  public:
        bool checkCycleUndirectedDFS(int src, unordered_map<int,bool> &visited, int parent, vector<int> adjList[]){
            // Initial State
            visited[src] = true;

            for(auto child: adjList[src]){
                    if(!visited[child]){
                        // Child Not Visited Yet
                        bool ans = checkCycleUndirectedDFS(child, visited, src, adjList);
                        if(ans == true){
                            return true;
                        }
                    }
                    else if(visited[child] && child == parent){
                        // Child Already Visited && Skip DFS Call
                        continue;
                    }
                    else if(visited[child] && child != parent){
                        // Child Already Visited && Cycle Present Hai
                        return true;
                    }
            }
            // Cycle Does Not Present
            return false;
        }
    // Function to detect cycle in an undirected graph.
    bool isCycle(int V, vector<int> adj[]) {

        // Visited
        unordered_map<int,bool> visited;
        for(int i=0; i<V; i++){
            if(!visited[i]){
               int parent = -1;
               bool ans = checkCycleUndirectedDFS(i,visited,parent,adj);
               if(ans == true){
                   return true;
               }
            }
        }
        return false;
    }
};
```

we can Add this condition to BFS Approach Also

Time and space complexity = ?

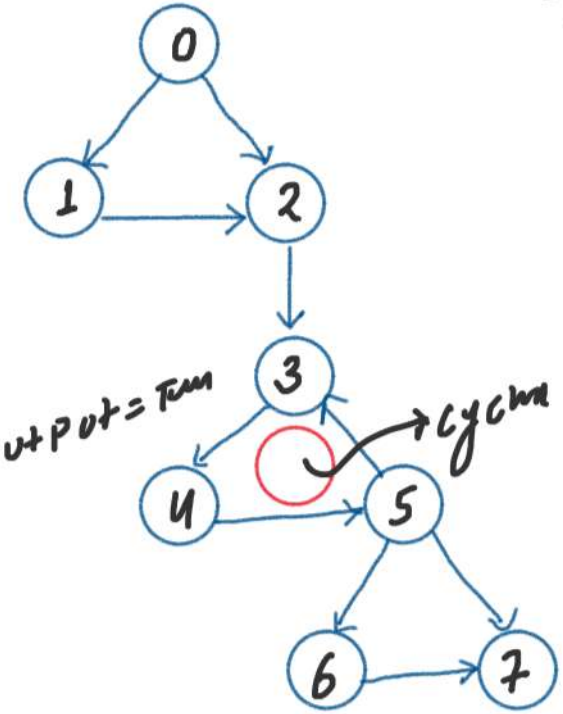
3. Detect cycle in an directed graph using DFS

CASE1



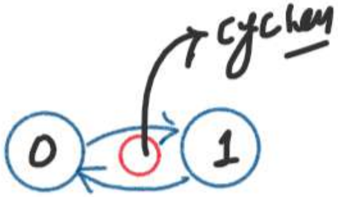
output = True

CASE 2



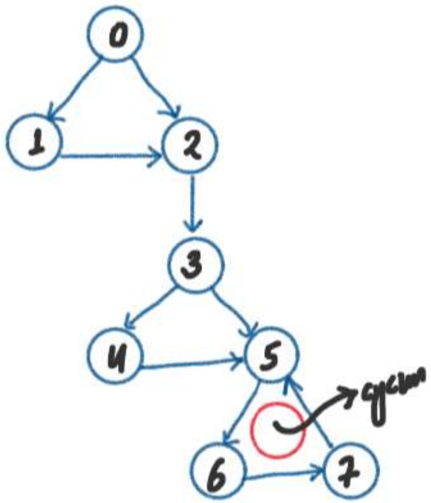
output = True

CASE 3



output = True

CASE1



DFS(0) → DFST= T
→ visi = T

↓

DFS(1) → DFST = T
→ visi = T

↓

DFS(2) → DFST = T
→ visi = T

↓

DFS(3) → DFST = T
→ visi = T

↓

DFS(4) → DFST = T
→ visi = T

↓

DFS(5) → DFST = T
→ visi = T

↓

DFS(6) → DFST = T
→ visi = T

↓

DFS(7) → DFST = T
→ visi = T

DFS(7) → DFS(5) →

ANS

Already DFST = T and vis = T

cycle present Hai

Adjlist

0: {1,2}
1: {2}
2: {3}
3: {4,5}
4: {5}
5: {6}
6: {7}
7: {5}

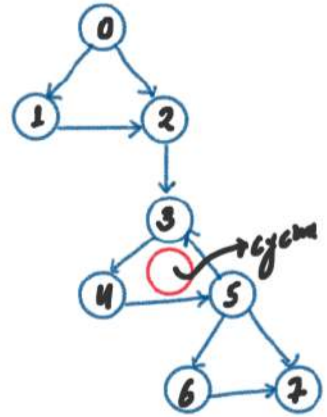
DFSTRACK

KEY	VALUE
0	~~F~~ T
1	~~F~~ T
2	~~F~~ T
3	~~F~~ T
4	~~F~~ T
5	~~F~~ T
6	~~F~~ T
7	~~F~~ T

visited

KEY	VALUE
0	~~F~~ T
1	~~F~~ T
2	~~F~~ T
3	~~F~~ T
4	~~F~~ T
5	~~F~~ T
6	~~F~~ T
7	~~F~~ T

CASE2



1 DFS(0) → DFST = T
→ vis = T

2 DFS(1) → DFST = T
→ vis = T

3 DFS(2) → DFST = T
→ vis = T

4 DFS(3) → DFST = T
→ vis = T

5 DFS(4) → DFST = T
→ vis = T

6 DFS(5) → DFST = T
→ vis = T

7 DFS(6) → DFST = ~~T~~ F
→ vis = T

8 DFS(7) → DFST = ~~T~~ F
→ vis = T

→ No child NO DFS call

BackTrack and return to parent

9 DFST(7) → DFST = ~~T~~ F

→ No child of (7)

DFS T = False
vis = True

→ No cycle

10 DFS (3)

ANS

DFST = TRUE
vis = TRUE

→ cycle Present hai

AdjList

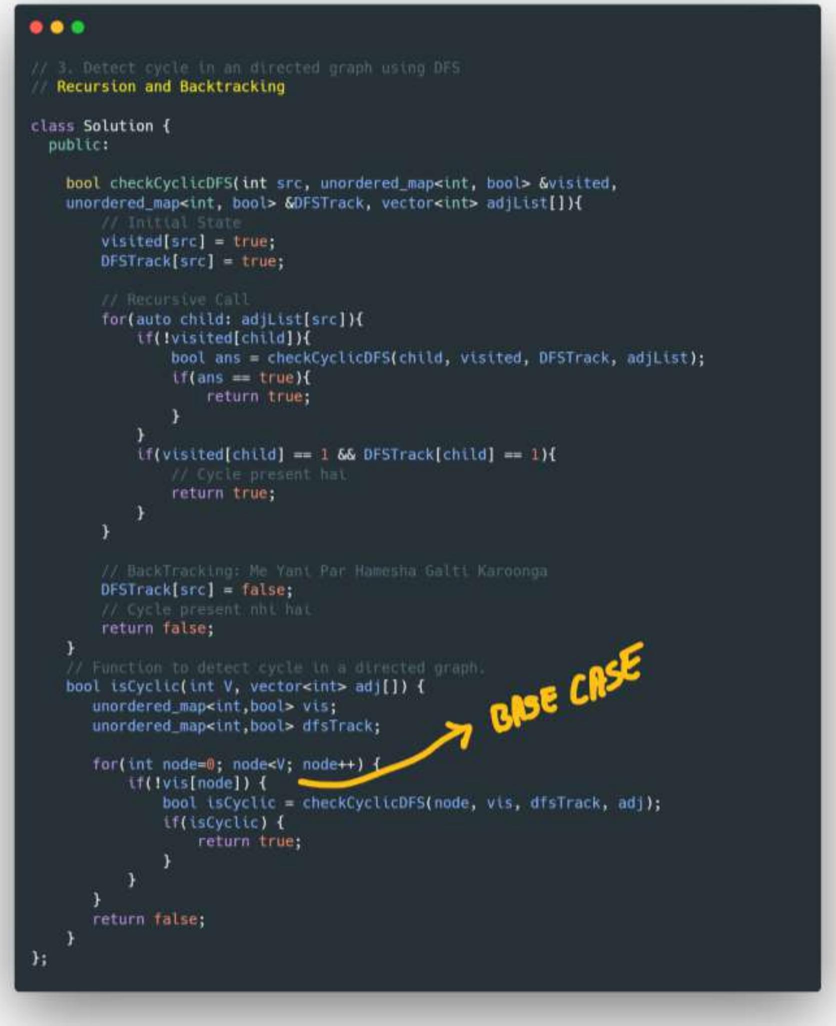
0 → {1, 2}
1 → {2}
2 → {3}
3 → {4}
4 → {5}
5 → {3, 6, 7}
6 → {7}
7 → NO child

DFSTRACK

KEY	VALUE
0	~~F~~ T
1	~~F~~ T
2	~~F~~ T
3	~~F~~ T
4	~~F~~ T
5	~~F~~ T
6	~~F~~ ~~T~~ F
7	~~F~~ ~~T~~ ~~F~~ ~~T~~ F

visited

KEY	VALUE
0	~~F~~ T
1	~~F~~ T
2	~~F~~ T
3	~~F~~ T
4	~~F~~ T
5	~~F~~ T
6	~~F~~ T
7	~~F~~ T



```
// 3. Detect cycle in an directed graph using DFS
// Recursion and Backtracking

class Solution {
  public:

    bool checkCyclicDFS(int src, unordered_map<int, bool> &visited,
    unordered_map<int, bool> &DFSTrack, vector<int> adjList[]){
        // Initial State
        visited[src] = true;
        DFSTrack[src] = true;

        // Recursive Call
        for(auto child: adjList[src]){
            if(!visited[child]){
                bool ans = checkCyclicDFS(child, visited, DFSTrack, adjList);
                if(ans == true){
                    return true;
                }
            }
            if(visited[child] == 1 && DFSTrack[child] == 1){
                // Cycle present hai
                return true;
            }
        }

        // BackTracking: Me Yani Par Hamesha Galti Karoonga
        DFSTrack[src] = false;
        // Cycle present nhi hai
        return false;
    }
    // Function to detect cycle in a directed graph.
    bool isCyclic(int V, vector<int> adj[]) {
       unordered_map<int,bool> vis;
       unordered_map<int,bool> dfsTrack;

       for(int node=0; node<V; node++) {
           if(!vis[node]) {
               bool isCyclic = checkCyclicDFS(node, vis, dfsTrack, adj);
               if(isCyclic) {
                   return true;
               }
           }
       }
       return false;
    }
};
```

BASE CASE

Time and space complexity = ?