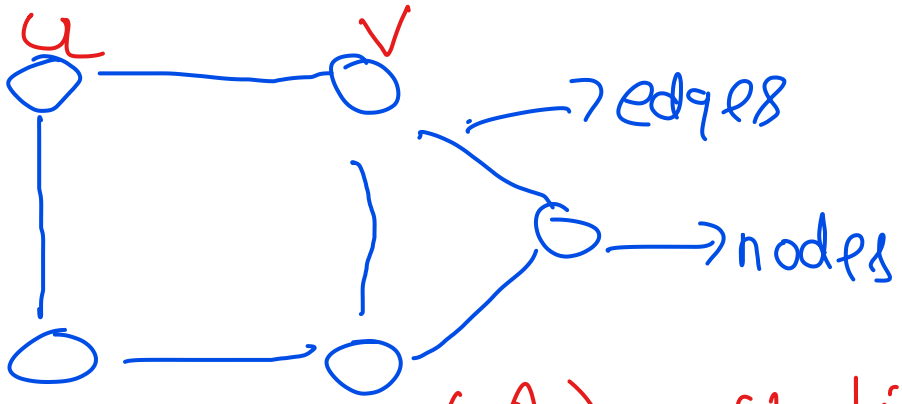
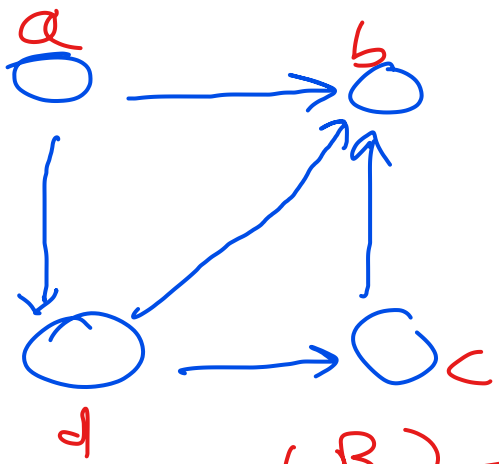
Graph



What = ?

(A) = Undirected

⇒ Type of Data Structure having node & edges.



a — b

b ≠ a

(B) = Directed

→ Node : Entity to store data

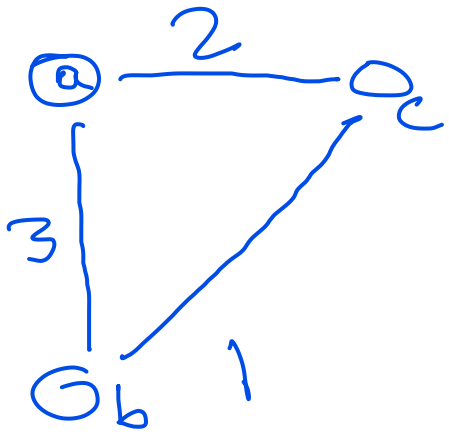
→ Edge : Connects nodes.

- Degree = ?
- Degree(v) = 3
- Degree(u) = 2

Directed graph.

- Indegree(a) = 0
- Outdegree(a) = 2
- Indegree(b) = 3

* Weighted Graph:



"If weights are not given in a graph, we will assume them to be '1' by default."

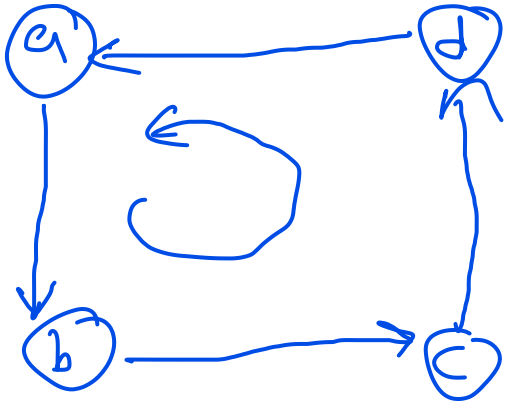
- Path: u - v - p

  p - q - v

  p - q - v - p (x) → not a path

  p repeated here.

- Cyclic Graph:



- Cyclic directed graph
- IP cycle doesn't form, means Acyclic Graph.

* Graph:

- Adjacency matrix
- Adjacency list.

① Adjacency matrix:

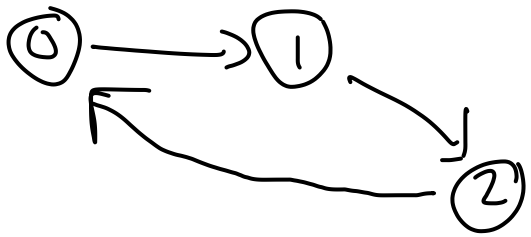
i/p: → no of nodes (n)
→ no of edges (m)

n = 3, m = 3

0 → 1

1 → 2

2 → 0



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

Space = $O(n^2)$

② Adjacency list:

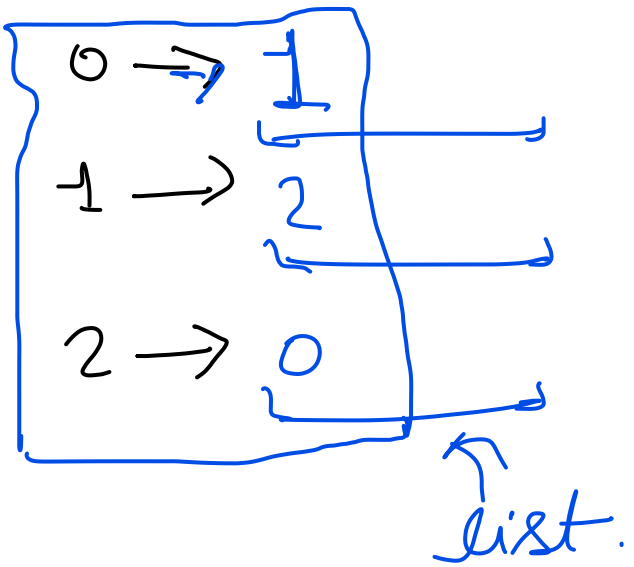
n = 3

m = 3

↳ 0 - 1

1 - 2

2 - 0



→ Implementation:

map < int, list <int> >;

or

vector<vector<int>>;

↓ Unordered

⇒ We will create a class for graph

① addEdge

② printadjList()

adj[u].pushback(v);
if (direction = 0)
{ v.pushback(u);
}

for (i : adj) { i.first & -> }
for (auto j : i.second) { j, , }

Creating a simple graph:::::::

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

class graph{
  public:
   unordered_map<int, list<int>>adj;
  void addedge(int u, int v, bool direction){
      adj[u].push_back(v);
      if(direction == 0){
        adj[v].push_back(u);
      }
  }

  void printadj(){
    for(auto i:adj){
      cout<<i.first<<"---->";
      for(auto j:i.second){
        cout<<j<<",";
      }
      cout<<endl;
    }
  }
};

int main()
{
  int n,m,a,b;
  bool direction;
  cout<<"enter n and m";
  cin>>n>>m;
  graph g;
  cout<<"connections";
  for(int i=0;i<n;i++){
    cin>>a>>b;
    g.addedge(a,b,direction);
  }

  g.printadj();
```

We can make it generic using —

→ template<typename T> } → Declare

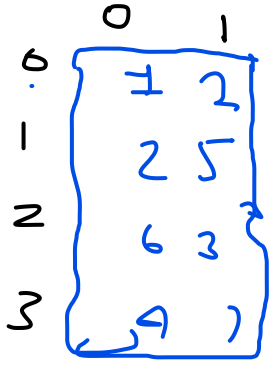
→ unordered_map< T, list<T>> adj;

→ void addedge (T u, T v, bool direction) } - use

→ graph <int> g } → tell.

```
#include<iostream>
#include<bits/stdc++.h>
using namespace std;
template<typename T>

class graph{
  public:
   unordered_map<T, list<T>>adj;
   void addedge(T u, T v, bool direction){
       adj[u].push_back(v);
       if(direction == 0){
         adj[v].push_back(u);
       }
    }


void printadj(){
      for(auto i:adj){
        cout<<i.first<<"---->";
        for(auto j:i.second){
          cout<<j<<",";
        }
        cout<<endl;
      }
      }
};

int main()
{
  int n,m,a,b;
  bool direction;
  cout<<"enter n and m";
  cin>>n>>m;
  graph<int> g;
  cout<<"connections";
  for(int i=0;i<n;i++){
    cin>>a>>b;
    g.addedge(a,b,direction);
  }

  g.printadj();

}
```

- given vector <vector < int >> & edges



2-D vector

u = [i] [0]

v = [i] [1].

→ make a vector, cnt → 1-D array

→ यहाँ v & V में u डाल दो

→

Important

Vectors :

① `vector<int> v(a,b);` or `vector<int> v(a)` → represents size.

↳ Behaves like 1-D array

a, b are the parameters given.

⇒ `v.push_back(5)`
`v.push_back(6)`
`v.push_back(3)`

so [5, 6, 3].

2-D Declaration-

② `vector < vector <int> > V;`

works like a 2D array.

[[ ], [ ], [ ], [ ]]
0 1 2 4.

→ so { `V[0].push_back(1)` → add 1 to 0th

v[-1].push-back(5)
v[0].push-back(6) → add 6 to v[0]
v[2].push-back(7).

→ So we will get

[[-1, 6), [5), [7), [ ))

→ again

v[1].push-back(4)
v[1].push-back(2)
v[3].push-back(9)

→ we will get -

[[1, 6), [5, 4, 2), [7), [9))

③ Vector<int> v[n]:

↳ This represents array of 'n' ,

6 vector<int>'. objects.

→ Each element of this array is a vector<int>.

↳ So typically, this behaves like vector<vector<int>> v, the only diff is, with 'n'.

So vector<int> v[3]

$\Rightarrow$ v = [[], [], []] (0, 1, 2)

$\Rightarrow$ v[2].push_back(5)

v[2].push_back(5)

v[1].push_back(6)

v[0].push_back(2)

↳ v = [[2], [5, 6], [5]]

Problem

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

0 1 2

ans, u → v

v → u

0 → 0, 1

1 → 2, 1

2 → 3, 1

we want a sequence,

0
1
2
3

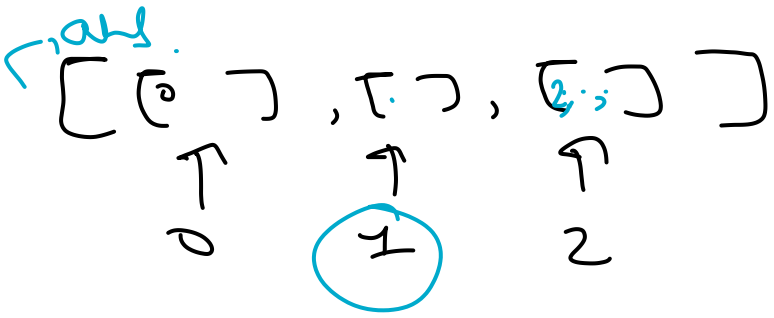
so for (i=0, < n)

adj[i].push_back(i);

for (j=0, j< ans[i].size ; j++)

↳ 2 → 3, 1

adj[i].push_back(ans[i][j])



i ans[i].size

20 array with some inputs.

```
vector < vector < int >> printAdjacency(int n, int m, vector < vector < int >> & edges) {
    // Write your code here.
    // n= nodes
    // m= edges
    vector<int> ans[n];
    for(int i=0;i<m;i++){
        int u = edges[i][0];
        int v = edges[i][1];

        ans[u].push_back(v);
        ans[v].push_back(u);

    }

    vector<vector<int>> adj(n);
    for(int i=0;i<n;i++){
        adj[i].push_back(i);

        for(int j=0; j<ans[i].size(); j++){
            adj[i].push_back(ans[i][j]);
        }
    }
    return adj;
}
```

↳ represents
0 - 1
1 - 2
3 - 4 type

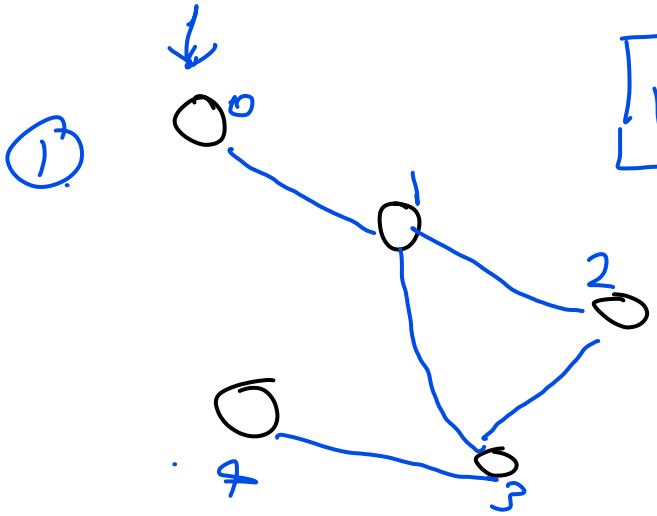
2D array refer above

input.

Lecture 2:

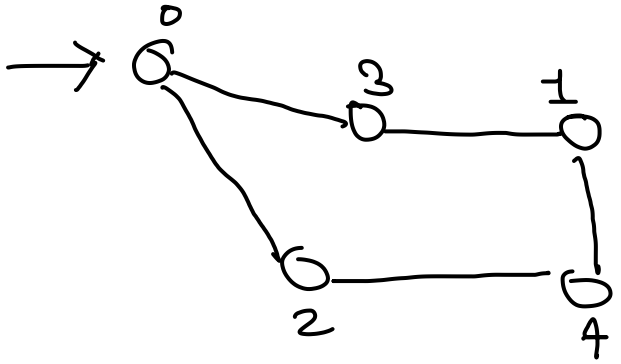
* BFS Traversal:

(Breadth First Search)



print → 0, 1, 2, 3, 4

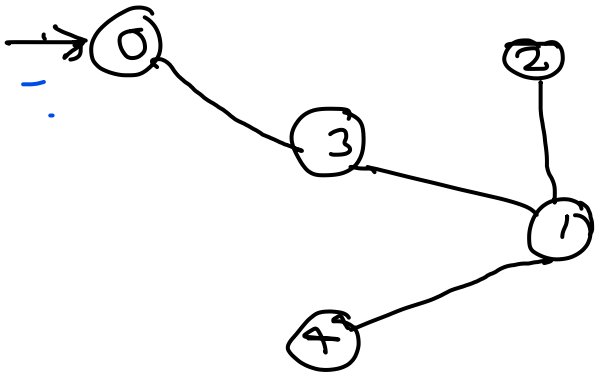
Check at 3, if 2 & 4 visited then print.



0, 2, 3, 4, 1

→ let's explore the methodology -

① Need of visited map ds:



(i) First visit a vertex 0, then 3.

(ii) So now check neighbours of 3. → (0, 1)

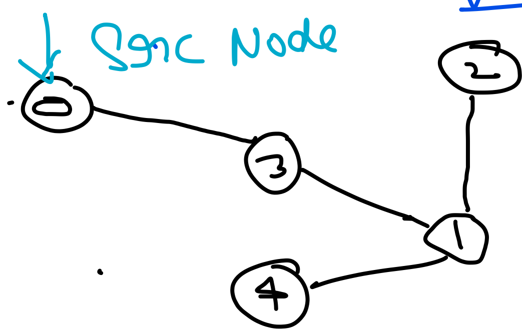
(iii) but we printed '0' already so no need to print it again,

Hence, we need to maintain

a 'visited' map, to store the index of visited element & to avoid infinite loop.

→ so `unordered_map<node, bool> visited;`
(node → int.)

② Need of queue DS:



Adj List –

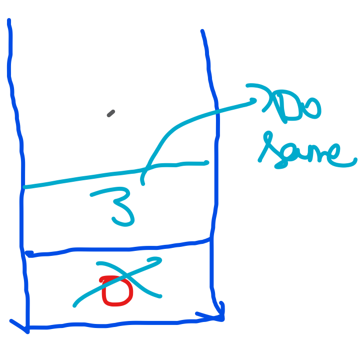
- 0 → 3
- 1 → 2, 3, 4
- 2 → 1
- 3 → 1, 0
- 4 → 1.

so

```
if(!visited[node])
    ↳ bfs()
```

Visited

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



① Put src Node in queue
② take it in front

→ Front Node = 0

④ print the node value.

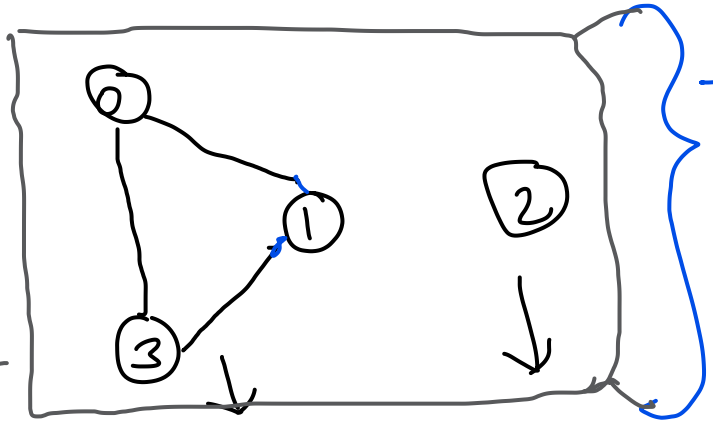
node & delete from queue.

⑤ mark this 'visited'

⑤ Check the neighbours of the node & put them in queue.

⑥ Again find the 'Front Node'.

* Gr:



→ Disconnected Graph.

Component I Comp II

TO handle this case -

```
for (int i=0; i<n; i++)
    { if (!visited(node))
        bfs()
    }
```

Problem -

- Given Disconnected Graph, make BFS traversal -
- Vector <int> BFS(int Vertex, vector<pair<int, int>> edges)
  - Vertex → no of nodes
  - return type
  - edges

{

}

Concept of Pair :

1. Used to store 2 values, which can be of different types.
2. pair <int, int>
   - Each pair consists two integers.

Let try to solve the problem -

given int V = vertex.

→ unordered_map<int, list<int>>adj;

→ for(i=0; i< ; i++) ← edges.size()

{

u = edges[i].first;

v = edges[i].second;

adj[u].pushback(v);

adj[v].pushback(u);

}

* Concept of Set:

① Works like list, the only diff is, stores values in sorted way

② We use insert function in place of push_back.

Eg = `adj[u].insert(v);`

Note - For a given -

unordered_map<int, list<int>>

⇒ For (auto i : adj){

→ Print first ← cout<< i.first <<"->";

element of pair.

→ print second elements of pair.

for(auto j : i.second){
cout << j << ",";
}
cout << endl;
}

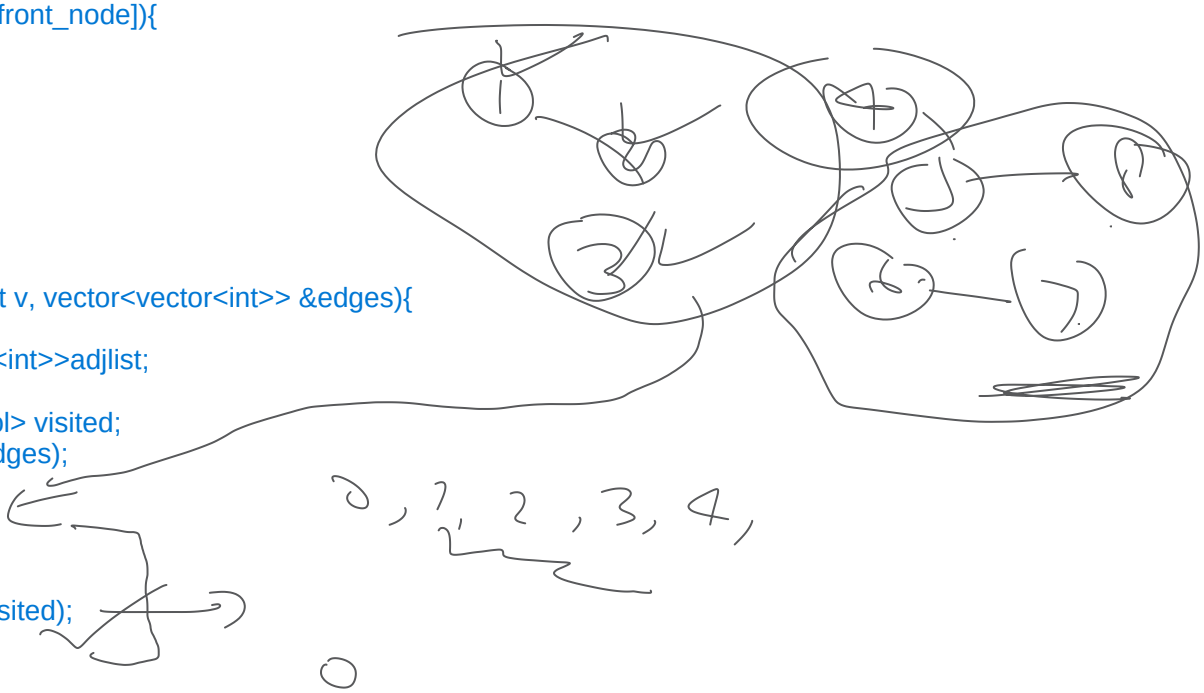
So - Basic concept is

(1)

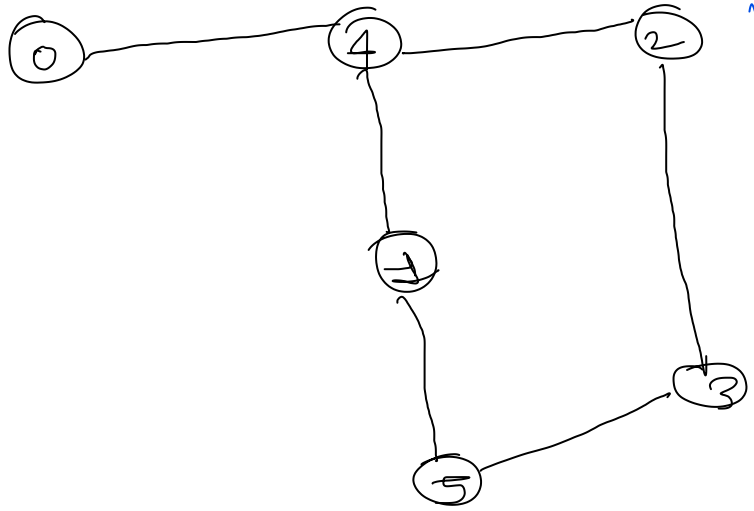
(Code for BFS

```cpp
#include<bits/stdc++.h>
#include<map>
void prepareAdjList(unordered_map<int, list<int>> &adjlist, vector<vector<int>> &edges ){
    //empty adjlist is given
    //edges given

    for(int i=0;i<edges.size();i++){
        int u = edges[i][0];
        int v = edges[i][1];

        adjlist[u].push_back(v);
        adjlist[v].push_back(u);

    }

}

void printAdjlist(unordered_map<int, list<int>> &adjlist){
    for(auto i:adjlist){
        cout<<i.first<<"--->";
        for(auto j:i.second){
            cout<<j<<",";
        }
        cout<<endl;
    }
}

void bfs(unordered_map<int, list<int>> &adjlist, vector<int> &ans, int node, unordered_map<int, bool> &visited ){
        queue<int> q;
        q.push(node);
        visited[node] = 1;

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

            ans.push_back(front_node);

            for(auto i: adjlist[front_node]){
                if(!visited[i]){
                    q.push(i);
                    visited[i]=1;
                }
            }
        }

}
vector<int> bfsTraversal(int v, vector<vector<int>> &edges){
    // Write your code here.
    unordered_map<int, list<int>>adjlist;
    vector<int> ans;
    unordered_map<int, bool> visited;
    prepareAdjList(adjlist, edges);
    printAdjlist(adjlist);

    for(int i=0;i<v;i++){
        if(!visited[i]){
            bfs(adjlist, ans, i, visited);
        }
    }
    return ans;
}
```

(error is there while running in code - Studio).



* DFS: (Depth first Search):

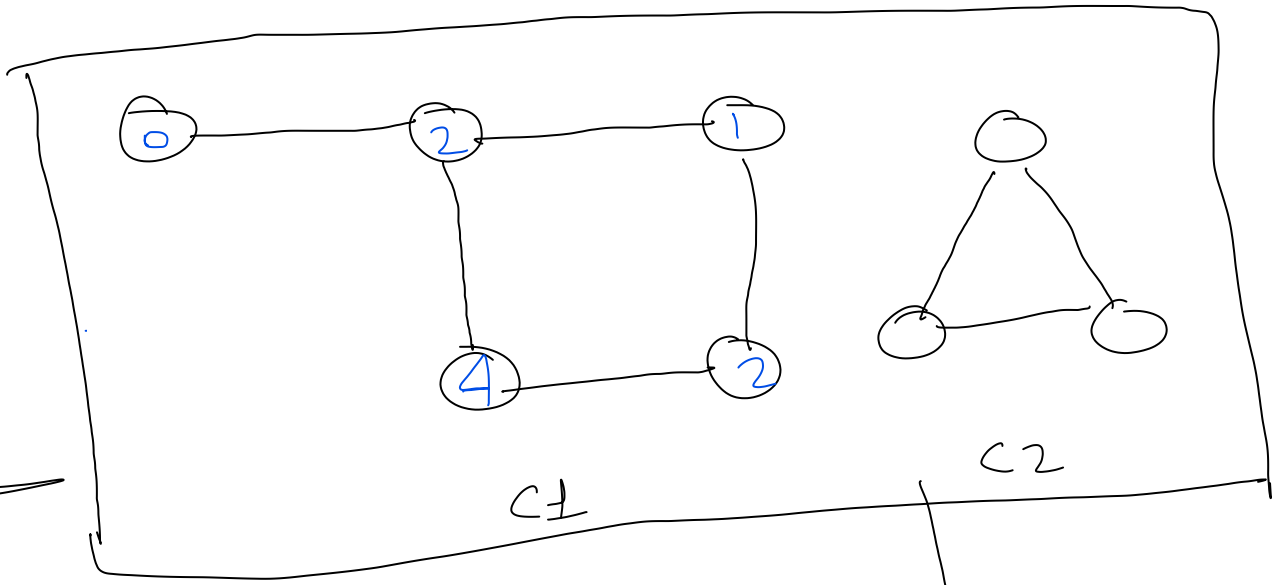


adjacent

BFS:

0 4 2 1 3 5

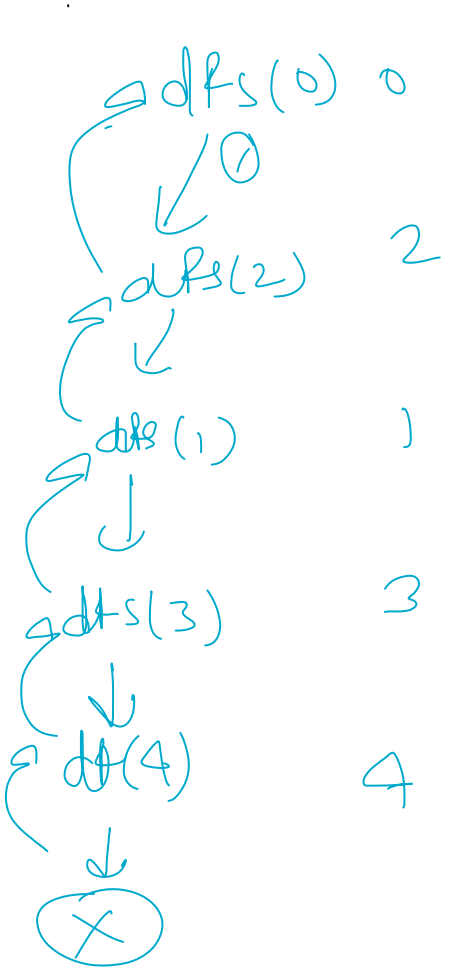
DFS: 0, 4, 2, 3, 5, 1



Graph.

```
for(int i=0;i<n)
  { if(!visited[i])
      { dfs()
      }
  }
```

→ Visited : unordered_map <int, bool>



② 0 → T
2 → T

adjlist
0 → 2
1 → 2, 3
2 → 0, 1, 4
3 → 1, 4
4 → 2, 3.

- Important things to do while writing code -

① Prepare adj list → as usual

② Create dfs

↳ By calling recursion -

```
for(auto i: adj[node])
{ if(visited[i])
  {
    dfs(i, visited, adj,
        component);
  }
}
```

dfs → parent_node = node.

↳ ans(node)

visited[node] = true

① → 1, 2, 3

for(i : adj[node]) {

↳ visited ⇒ dfs call.

⇒ vector<vector<int>> ans;

for (int i=0; i<V; i++) {

if (!visited[i]){

vector<int> component;

dfs( );

ans.pushback(component);

return ans;

```cpp
#include<bits/stdc++.h>
#include<map>

void prepareAdj(unordered_map<int, list<int>> &adjlist, vector<vector<int>> &edges){
    for(int i=0;i<edges.size();i++){
        int u =edges[i][0];
        int v =edges[i][1];

        adjlist[u].push_back(v);
        adjlist[v].push_back(u);
    }
}


void dfs( unordered_map<int, list<int>> &adjlist,  vector<int> &ans, unordered_map<int, bool> &vistited, int node){
     ans.push_back(node);

     vistited[node] = true;

     for(auto i:adjlist[node]){
         if(!vistited[i]){
             dfs(adjlist,ans, vistited, i);
         }
     }
}

vector<vector<int>> depthFirstSearch(int V, int E, vector<vector<int>> &edges)
{
    unordered_map<int, list<int>> adjlist;
    unordered_map<int, bool> visited;

    vector<vector<int>>ans;

    prepareAdj(adjlist,edges);

// dfs(adjlist, ans, vistited, 0);

for(int i=0;i<V;i++){
    if(!visited[i]){

vector<int> component;
          dfs(adjlist, component, visited, i );

ans.push_back(component);

    }
}

return ans;
```

→ Prepare list.

→ Call dfs function

→ to handle case for disconnected graph

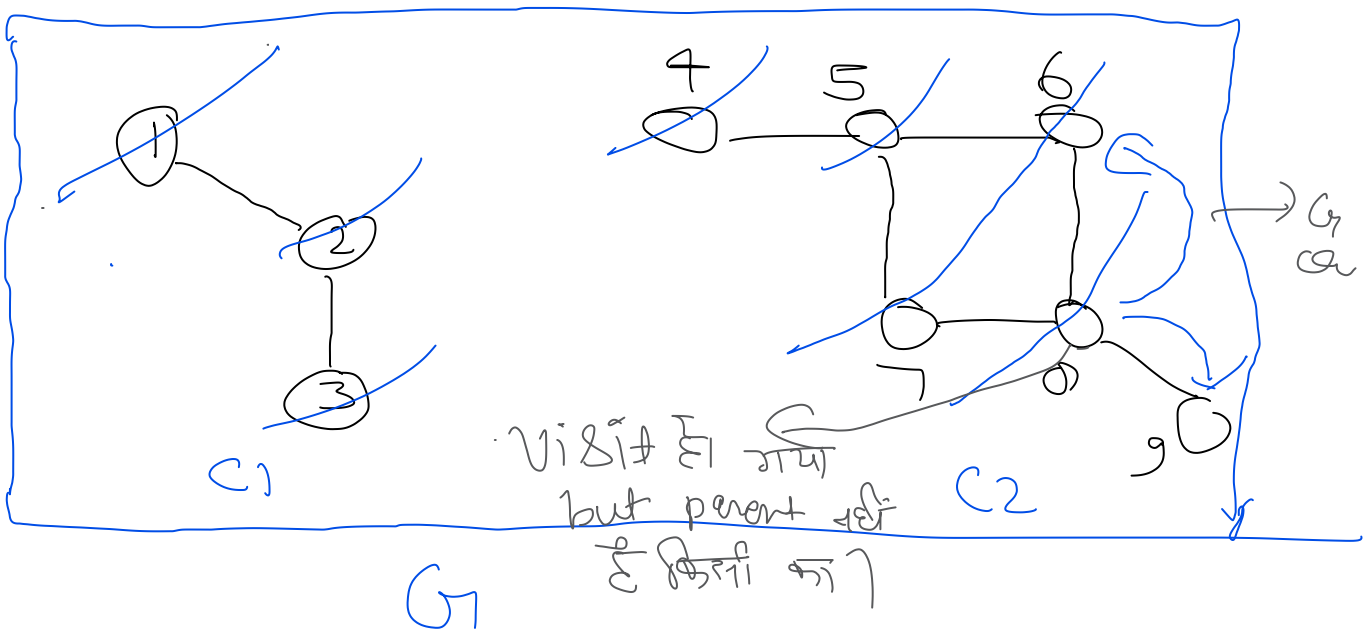
Otherwise we can normally call dfs.

(4)

Cycle Detection.

→ Cycle detection is undirected graph.

① BFS (By BFS):



⇒ We need to know, if Cycle is present in C1 or C2?

→ Adj:

1 → 2
2 → 1, 3
3 → 2

4 → 5
5 → 4, 6, 7
6 → 5, 8
7 → 5, 8
8 → 6, 7, 9
9 → 8

- We are going with BFS, so we need to maintain visited array & queue.

- Analysis-

1 → 2
2 → 1,3
3 → 2
4 → 5
5 → 4,6,7
6 → 5,8
7 → 5,8
8 → 6,7,9
9 → 8

Src = 1

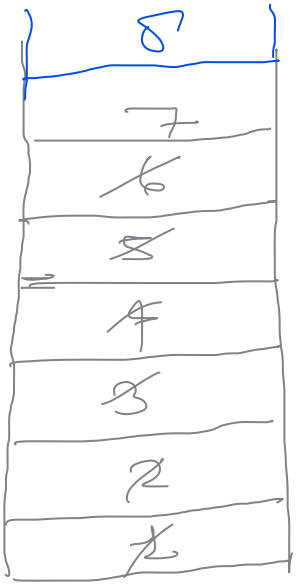
Parent (parent: x → y)

Visited

Set -1, if parent doesn't exist.

x	Parent	Visited
1	-1	True
2	1	True
3	2	True
4	-1	True
5	4	True
6	5	True
7	5	True
8	6	True

Parent[i] = previous one.




```
IF (visited & parent)
{ neglect;
}
```

⇒ (1) take src node, put in visited mark True then, mark its parent → 2 put this in queue

(2) Now check, what is there in adjlist Check that element (2) if not visited,

mark it "True" & 2 come from 1 so mark '1' as 2's parent. (2→1) & put this in queue.

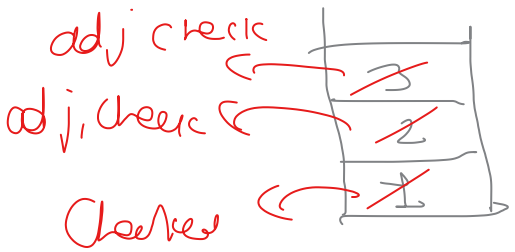
(3) Go in list of 2, check there exist 1 & 3, 1 is already visited & parent of 2 so neglect this. ↳ 2→1,3.

So one condition

if (visited[i] & parent-of-previous one)
neglect this.

So mark 3 as True.

(4) Go in list of (3→2), check 2, 2 is already visited & parent of 3, so neglect this no need to put this in the queue.



→ nothing remaining in the queue

(5) let's move to the other component — So check 4, mark it true &

parent of 4 = -1

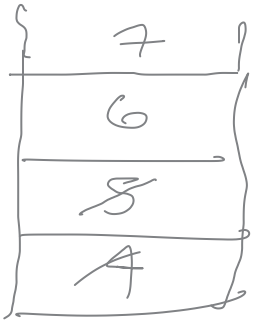
(6) Check list of 4 (& pop it), we (4 → 5), will get 5, which is not visited so mark it True & set parent of (5 = 4) (5 → 4).

(7) Check list of 5, so pop it, we we get (4, 6, 7), (5 → 4, 6, 7)

(a) 4 is already visited & parent of 5, & reject it. → ignore

(b) 6 is not visited so mark it True & set parent of it (6 → 5) & put this in the queue

(c) 7 is not visited so mark is True & set its parent (7 → 5) & put this in the queu



(8) Now check the list for 6 → 5, 8, so first 'pop 6'.

(a) 5 is visited & parent of 6 | 7 so negled it.

(b) 8 is not visited mark this True & put this in the queue

(9) Check list of 7 → 5, 8, 'pop 7'

(a) ignore 5

(b) 8 is visited but not parent of any element or node

→ So the problem comes here.
So cycle detected

so we got to the conclusion if

→ Element is visited & not parent of someone means cycle is present there

So typically 2 condition, we arrived -

① if (visited[i] && parent(i)
{ ignore it
}

② if (visited[i] && !parent[i])
{ Cycle detected;
}

③ if (!visited[i])
{
visited[i] = true;
}.

So 2 things, we need to put more
① → DS for the parent

(3) → Condition.

So.

```
while(!q.empty())
{
    int front = q.front();
    q.pop();
    for(auto i : adjlist[front])
    {
        if(visited[i] && i != parent[front])
            return true
        else if(!visited[i])
        {
            visited[i] = true;
            parent[i] = front;
            q.push(i);
        }
    }
```

* T.C = linear

⇒ To solve case of disconnected components -

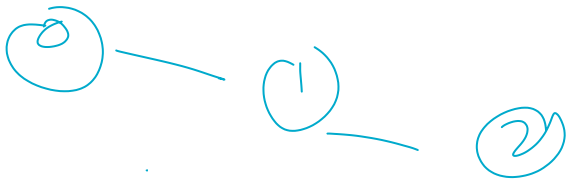
```
for(int i=0; i<n; i++) {
    if(!visited[i])
    {
        bool ans =
            isCyclic(i, visi, adj);
        if(ans == 1)
            return "yes";
    }
}
```

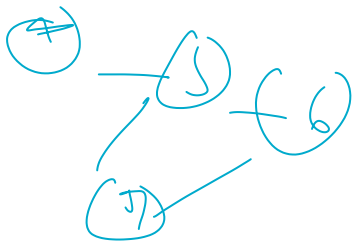
```cpp
#include<bits/stdc++.h>

void praparelist(unordered_map<int, list<int>> &adjlist, vector<vector<int>>& edges){
    for(int i=0; i<edges.size(); i++){
        int u=edges[i][0];
        int v = edges[i][1];

        adjlist[u].push_back(v);
        adjlist[v].push_back(u);
    }
}

bool cyclecheck(unordered_map<int, list<int>> &adjlist, unordered_map<int, bool> &visited, int src){
    queue<int> q;
    unordered_map<int, int> parent;
    q.push(src);
    visited[src] = 1;
    parent[src] = -1;

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

      for(auto i: adjlist[front]){
          if(visited[i] && i != parent[front]){
              return true;
          }else if (!visited[i]){
              visited[i] = true;
              parent[i] = front;
              q.push(i);
          }
      }
  }
}
string cycleDetection (vector<vector<int>>& edges, int n, int m)
{
    // Write your code here.
    unordered_map<int, list<int>> adjlist;
    unordered_map<int, bool> visited;
    praparelist(adjlist, edges);

    // To handle Disconnected graph case
    for(int i=0;i<n;i++){
        if(!visited[i]){
            bool ans = cyclecheck(adjlist, visited, i);
            if(ans){
                return "Yes";
            }
        }
    }
    return "No";
}
```

0, 1, 2, - - - n.

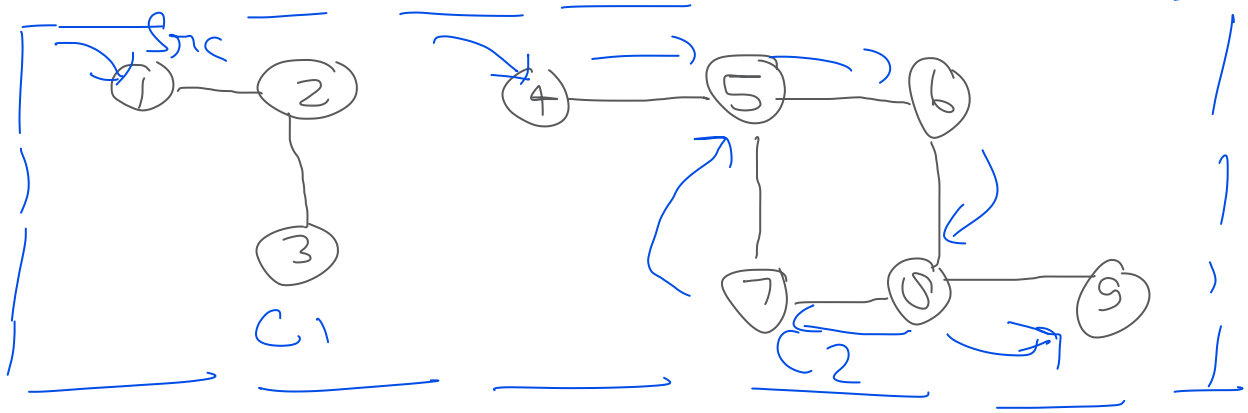
*Disconnected Case explanation:-
we will send 0 which is not visited till now after passing in function this will get visited & the operations inside it will make other nodes visited till

graph is connected as per bfs ex





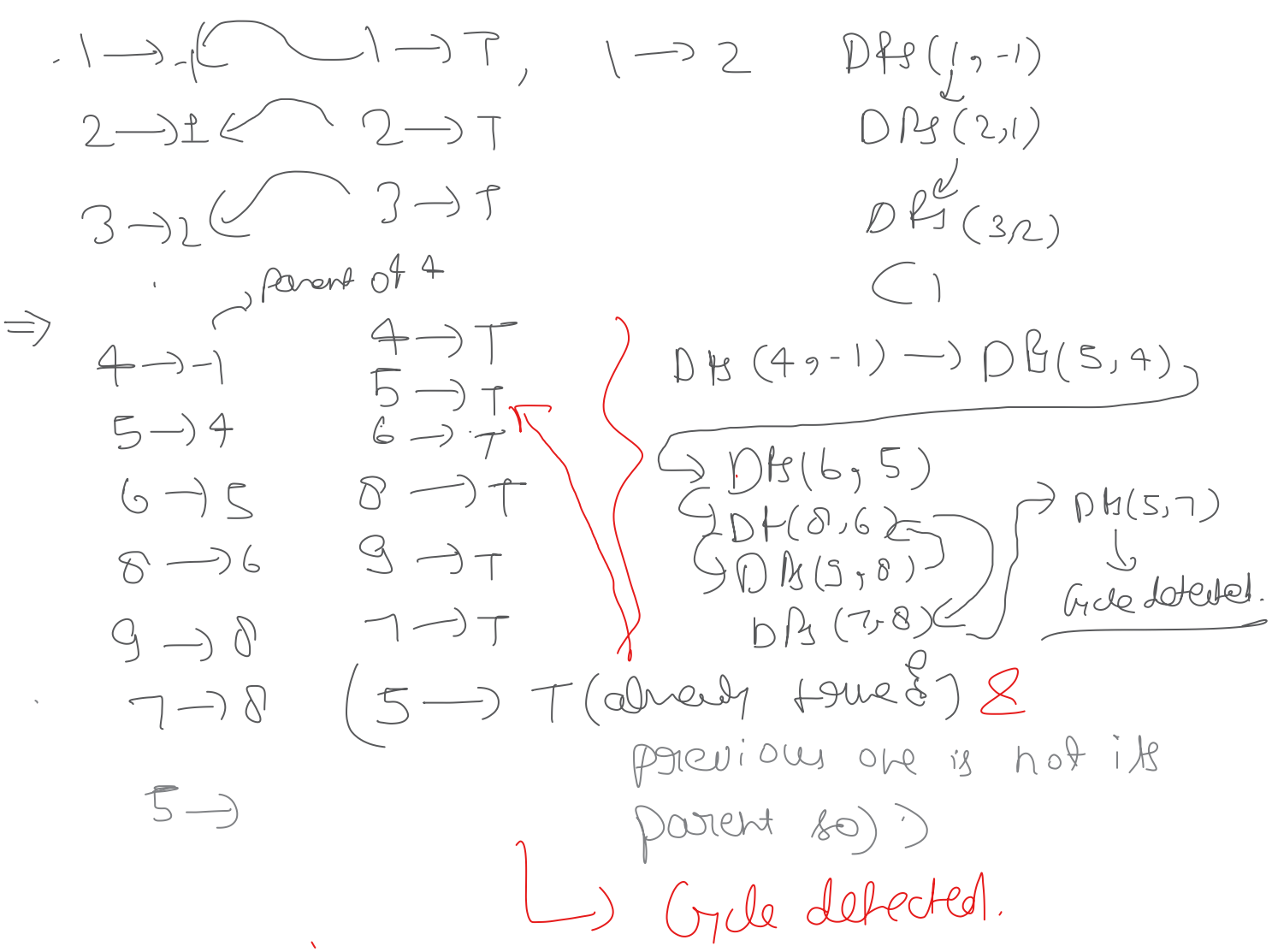
* DFS:



⇒ Visited = True, i != Parent : Cycle

⇒ Understanding of code.

Parent Visited



Code for DFS —

```
string cycleDetection (vector<vector<int>>& edges, int n, int m)
{
    // Write your code here.
    unordered_map<int, list<int>> adjlist;
    unordered_map<int, bool> visited;
    praparelist(adjlist, edges);

    // To handle Disconnected graph case
    for(int i=0;i<n;i++){
        if(!visited[i]){
            // bool ans1 = cyclecheckBfs(adjlist, visited, i);
            bool ans2 = cyclecheckDfs(adjlist, visited, i, -1);
            if(ans2){
                return "Yes";
            }
        }
    }
    return "No";
}
```

→ To handle great Disconnected graph.

```
bool cyclecheckDfs(unordered_map<int, list<int>> &adjlist, unordered_map<int, bool> &visited, int src, int prev){
    // int prev = -1;
    visited[src] = true;

    for(auto i: adjlist[src]){
        if(!visited[i]){
            bool cycle = cyclecheckDfs(adjlist, visited, i, src);
            if(cycle){
                return true;
            }
        }else if (visited[i] && i != prev ){
            return true;
        }
    }
    return false;
}
```

5 → 6

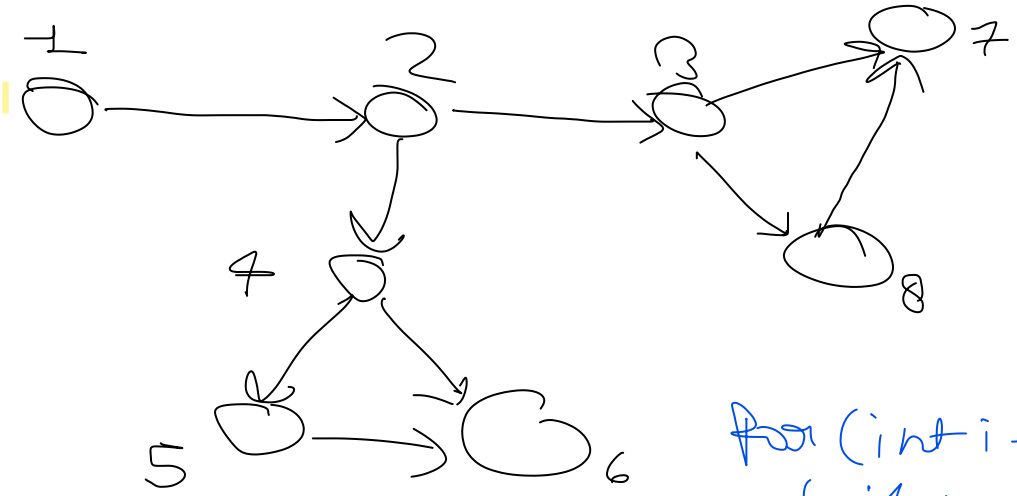
i != prev ?

i should not be parent.

Lecture 5:

* Cycle Detection in —

'Directed Graph'




```
for (int i -> n)
{ if(!vis[i])
    dfs()
}
```

* Graph. G1 —

Adjlist

1 → 2

Visited

1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0

2 → 3, 4

3 → 7, 8

4 → 5

5 → 6

6 → 4

7 →

8 → 7

f(1) → 1 dfT vis mark kri fun.

↓

f(2) → visited mark

↓

f(3) → visited

↓

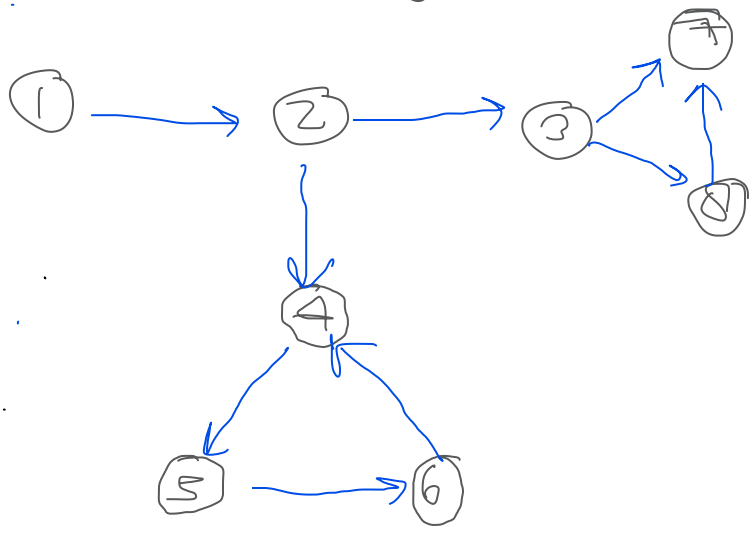
f(7) → visited (nothing is there so return)

f(8) → visited

↓

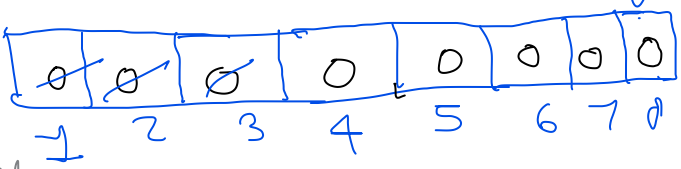
f(7) → already visited & 7 is not parent of 8 (so cycle present)

So lets understand the algo in deep-



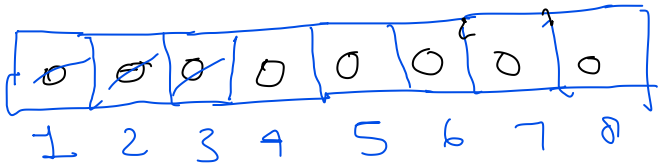
So Here we will maintain 2 array -

(1) Visited



↳ works generally as previously

(2) dfs visited



↳ marks

(a) true if function call for an element happens Here on respective index

(b) If function takes back then respective index will again be treated as false

So For the graph -

dfs(1): visited = true, dfs visited = True

↓

dfs(2): V = true, dv = True

↓

dfs(3): V = true, dv = True

↓

dfs(4): V = true, dv = true.

↓

dfs(-