19/02/2024

GRAPHS CLASS - 3

1. What is topological sort and where to use it?

Topological sorting *for* ***Directed Acyclic Graph (DAG)*** *is a linear ordering of vertices(Nodes) such that for every directed edge* ***u->v,*** *vertex* ***u*** *comes before* ***v*** *in the ordering.*

Note: Topological Sorting for a graph is not possible if the graph is not a DAG.

Where to use in real life?

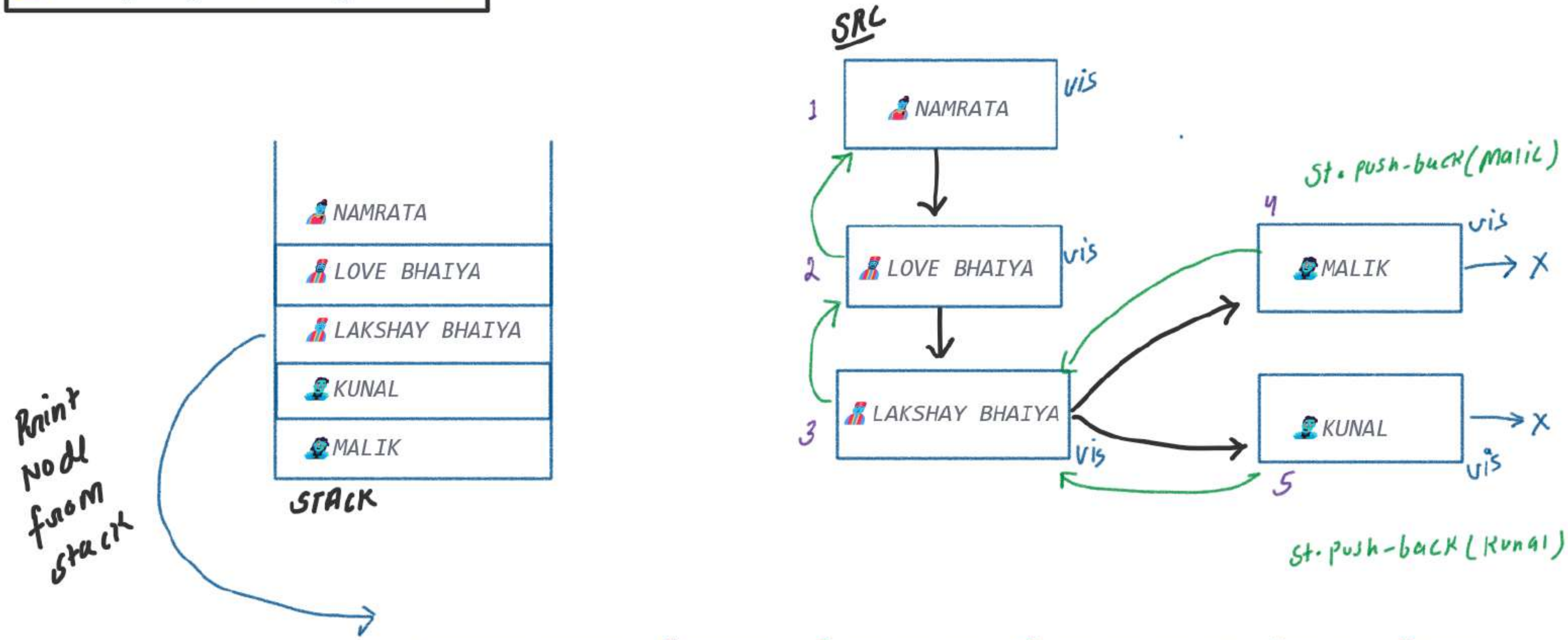
JAB KABHI BHI ***DEPENDENCY ORDER*** *KI BAT KI JA RAHI HOGI TAB HUM* ***TOPOLOGICAL SORT*** *KO APPLY KAR SKTE HAI.*

LIKE: NAMRATA -> LOVE BHAIYA -> LAKSHAY BHAIYA -> MALIK AND KUNAL

EXPLANATION:

- *NAMRATA KISI PAR BHI DEPEND NHI KARTI HAI BUT LOVE BHAIYA ISS LADKI PAR DEPEND HAI KI YEH PAHLE KUCH KARGI TABHI ME KUCH KAR PAUNGA*
- *LAKSHAY BHAIYA DEPENDS ON LOVE BHAIYA*
- *MALIK AND KUNAL DEPENDS ON LAKSHAY BHAIYA*
- *NO ONE DEPENDS ON MALIK AND KUNAL*

2. Topological sorting with DFS

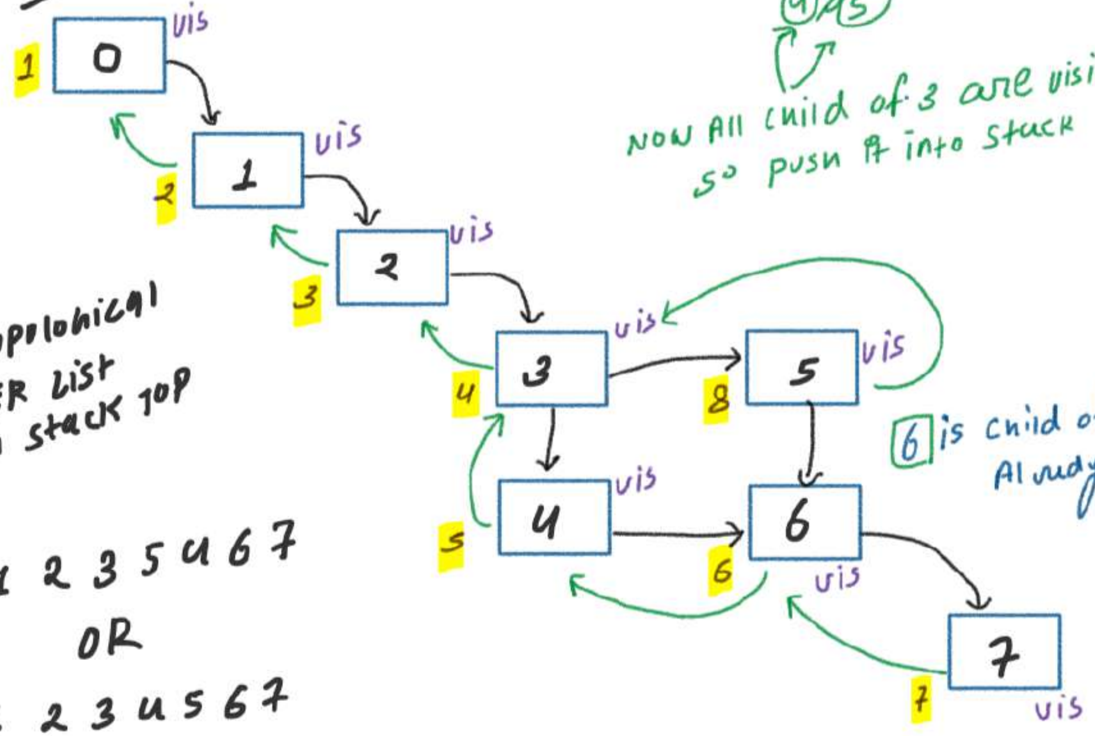


EXPECTED OUTPUT: NAMRATA, LOVE BHAIYA, LAKSHAY BHAIYA, MALIK, KUNAL

Independent Ladki

Ex

SRC



④ & ⑤

Now All child of 3 are visited so push it into stack

6 is child of 5 which is Already visited so can't do DFS(6) Again

No one Depends on 7 So push into stack first

OUTPUT

Print Topological ORDER List from stack Top

0 1 2 3 5 4 6 7

OR

0 1 2 3 4 5 6 7

AdjList

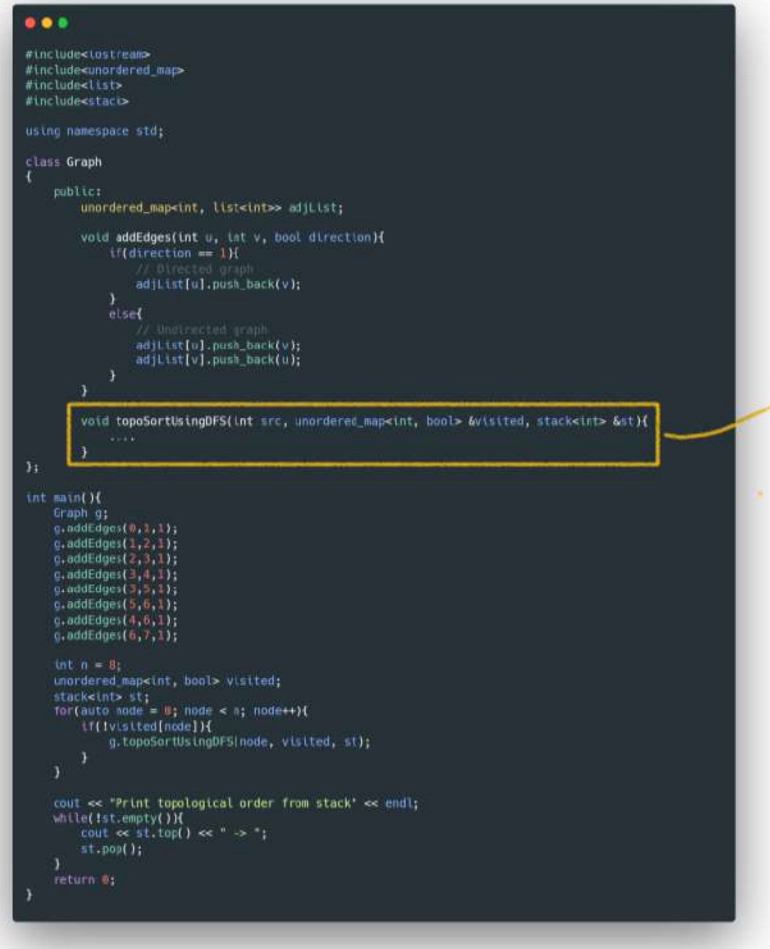
Key	value
0	{1}
1	{2}
2	{3}
3	{4,5}
4	{6}
5	{6}
6	{7}
7	X

Visited

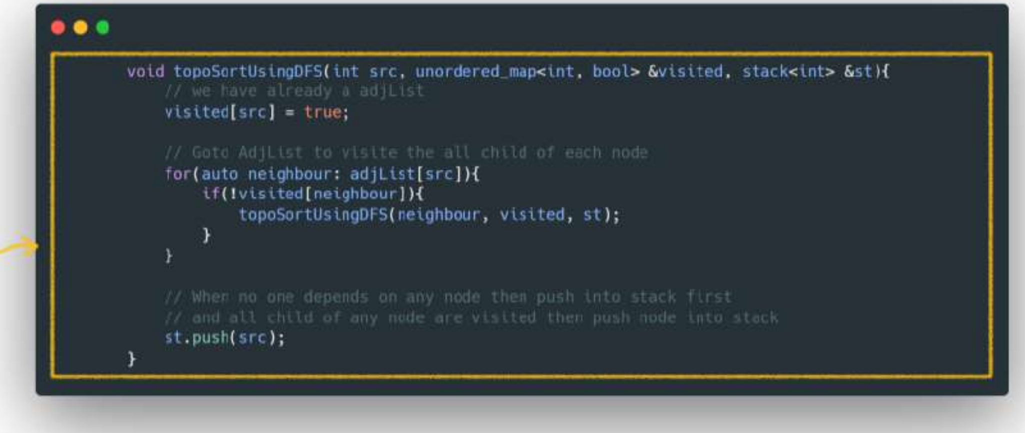
Key	value
0	~~F~~ T
1	~~F~~ T
2	~~F~~ T
3	~~F~~ T
4	~~F~~ T
5	~~F~~ T
6	~~F~~ T
7	~~F~~ T

0
1
2
3
5
4
6
7

STACK



```cpp
#include<iostream>
#include<unordered_map>
#include<list>
#include<stack>

using namespace std;

class Graph
{
    public:
        unordered_map<int, list<int>> adjList;

        void addEdges(int u, int v, bool direction){
            if(direction == 1){
                // Directed graph
                adjList[u].push_back(v);
            }
            else{
                // Undirected graph
                adjList[u].push_back(v);
                adjList[v].push_back(u);
            }
        }

        void topoSortUsingDFS(int src, unordered_map<int, bool> &visited, stack<int> &st){
            ....
        }
};

int main(){
    Graph g;
    g.addEdges(0,1,1);
    g.addEdges(1,2,1);
    g.addEdges(2,3,1);
    g.addEdges(3,4,1);
    g.addEdges(3,5,1);
    g.addEdges(5,6,1);
    g.addEdges(4,6,1);
    g.addEdges(6,7,1);

    int n = 8;
    unordered_map<int, bool> visited;
    stack<int> st;
    for(auto node = 0; node < n; node++){
        if(!visited[node]){
            g.topoSortUsingDFS(node, visited, st);
        }
    }

    cout << "Print topological order from stack" << endl;
    while(!st.empty()){
        cout << st.top() << " -> ";
        st.pop();
    }
    return 0;
}
```



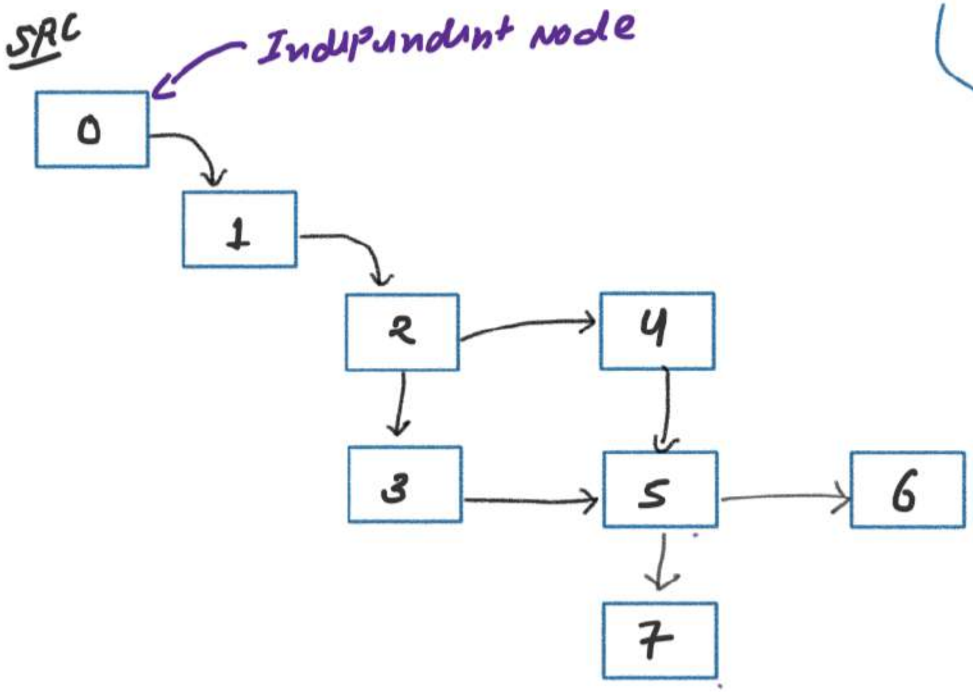
```cpp
        void topoSortUsingDFS(int src, unordered_map<int, bool> &visited, stack<int> &st){
            // we have already a adjList
            visited[src] = true;

            // Goto AdjList to visite the all child of each node
            for(auto neighbour: adjList[src]){
                if(!visited[neighbour]){
                    topoSortUsingDFS(neighbour, visited, st);
                }
            }

            // When no one depends on any node then push into stack first
            // and all child of any node are visited then push node into stack
            st.push(src);
        }
```

why use stack?

↳ Because we have to print Independent node First.

↳ But we can also use vector, DE-QUEUE, Array.

3. Topological sorting with BFS



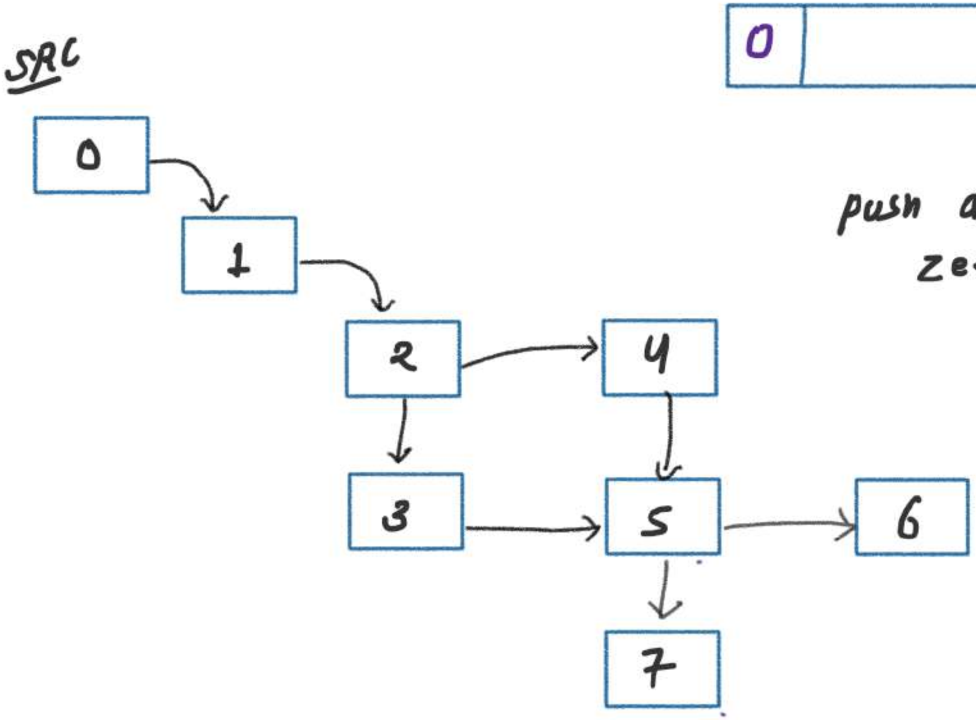
Note: jis node ki indegree = 0 Hai us NODE KO QUEUE me push karodo because this node is indipendent.

why? we have to print indipendent node first.

Key	valu
0	0
1	1
2	1
3	1
4	1
5	2
6	1
7	1

Indegree

output [0 1 2 3 4 5 6 7]



Queue

push all Nodes(jinki indegree zero Hai) into Queue

Key	value
0	{1}
1	{2}
2	{3,4}
3	{5}
4	{5}
5	{6,7}
6	X
7	X

AdjList

Key	value
0	0
1	1
2	1
3	1
4	1
5	2
6	1
7	1

Indegree

STEP1 initialize indegree

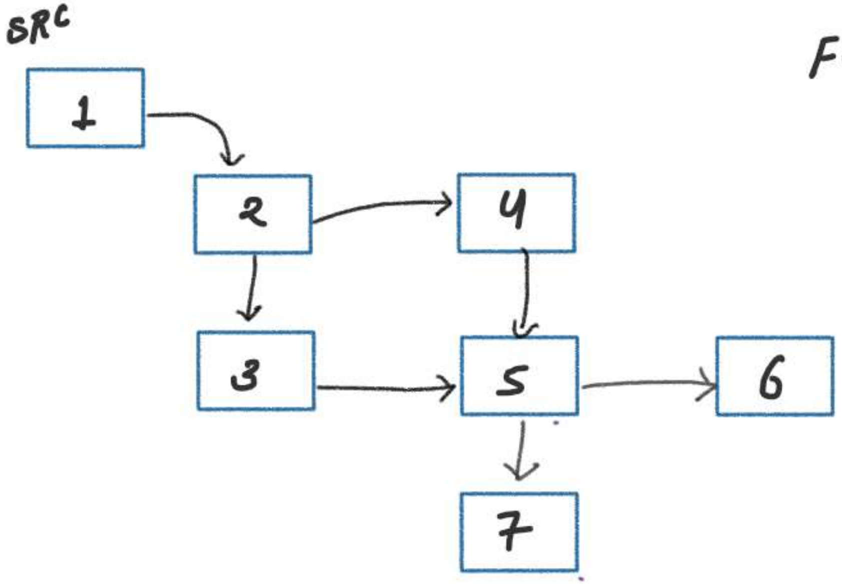
STEP2 push All Nodes of zero indegree into Queu

STEP3 BFS ON Queue

Iteration 1

Queue

~~0~~	1	



FrontNode = 0 and pop()

Key	valu
0	{1}
1	{2}
2	{3,4}
3	{5}
4	{5}
5	{6,7}
6	X
7	X

Adjlist

Key	valu
0	0
1	~~1~~ 0
2	1
3	1
4	1
5	2
6	1
7	1

Indegree

output

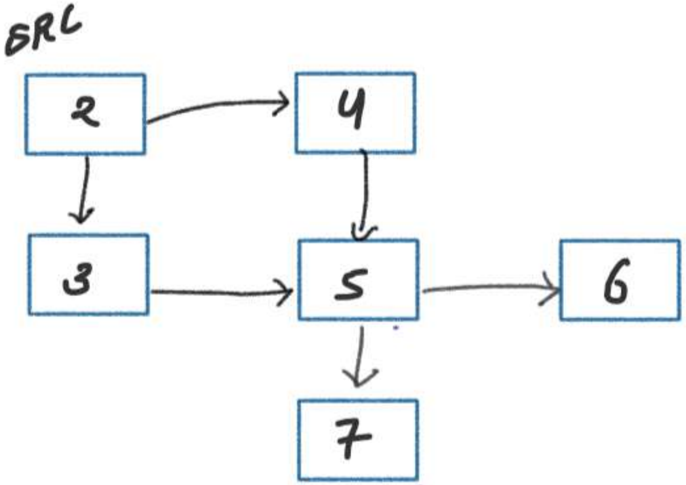
0

Iteration 2

Queue

~~1~~	2

FrontNode = 1 and pop()



Key	valu
0	{1}
1	{2}
2	{3,4}
3	{5}
4	{5}
5	{6,7}
6	X
7	X

AdjList

Key	valu
0	0
1	~~1~~ 0
2	~~1~~ 0
3	1
4	1
5	2
6	1
7	1

Indegree

Output

0 1

Iteration3

Queue

~~2~~	3	4

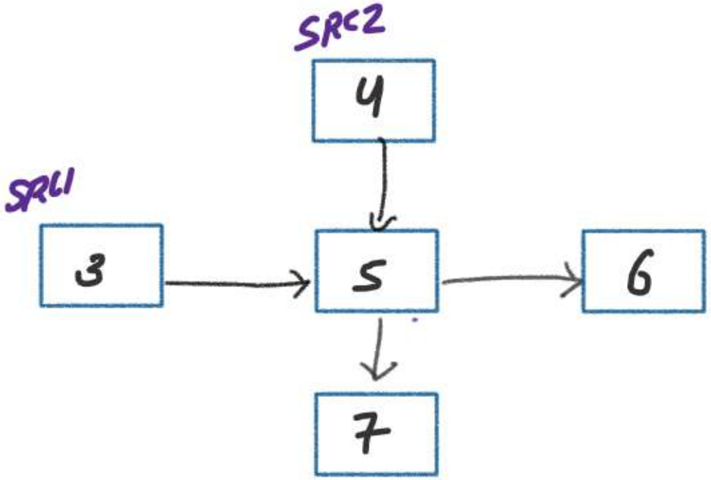
FrontNode =2 and pop()

Key	value
0	{1}
1	{2}
2	{3,4}
3	{5}
4	{5}
5	{6,7}
6	X
7	X

AdjList

Key	value
0	0
1	~~1~~ 0
2	~~1~~ 0
3	~~1~~ 0
4	~~1~~ 0
5	2
6	1
7	1

Indegree



output

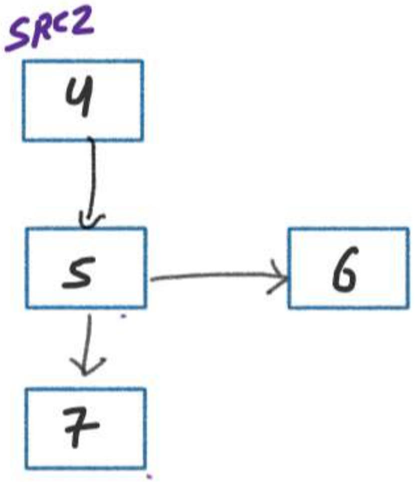
0 1 2

Iteration 4

Queue

~~3~~	4

FrontNode = 3 and pop()



Key	valu
0	{1}
1	{2}
2	{3,4}
3	{5}
4	{5}
5	{6,7}
6	X
7	X

Adjlist

Key	valu
0	0
1	~~1~~ 0
2	~~1~~ 0
3	~~1~~ 0
4	~~1~~ 0
5	~~2~~ 1
6	1
7	1

Indegree

Output

0 1 2 3

Iteration 5

Queue

~~4~~	5

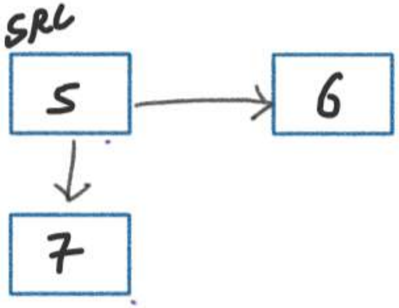
FrontNode = 4 and pop()

Key	value
0	{1}
1	{2}
2	{3,4}
3	{5}
4	{5}
5	{6,7}
6	X
7	X

AdjList

Key	value
0	0
1	~~1~~ 0
2	~~1~~ 0
3	~~1~~ 0
4	~~1~~ 0
5	~~2~~ ~~1~~ 0
6	1
7	1

Indegree



output

0 1 2 3 4

Iteration6

Queue

~~5~~	6	7

FrontNode =5 and pop()

Key	Value
0	{1}
1	{2}
2	{3,4}
3	{5}
4	{5}
5	{6,7}
6	X
7	X

Adjlist

Key	Value
0	0
1	~~1~~ 0
2	~~1~~ 0
3	~~1~~ 0
4	~~1~~ 0
5	~~2~~ ~~1~~ 0
6	~~1~~ 0
7	~~1~~ 0

Indegree

SRC

6

SRC

7

output

0 1 2 3 4 5

Iteration 7

Queue

~~6~~	7

FrontNode = 6 and pop()

Key	value
0	{1}
1	{2}
2	{3,4}
3	{5}
4	{5}
5	{6,7}
6	X
7	X

AdjList

Key	value
0	0
1	~~1~~ 0
2	~~1~~ 0
3	~~1~~ 0
4	~~1~~ 0
5	~~2~~ ~~1~~ 0
6	~~1~~ 0
7	~~1~~ 0

Indegree

SRC

7

output

0 1 2 3 4 5 6

Iteration 8

Queue

~~7~~

FrontNode = 7 and pop()

Key	value
0	{1}
1	{2}
2	{3,4}
3	{5}
4	{5}
5	{6,7}
6	X
7	X

Adjlist

Key	value
0	0
1	~~1~~ 0
2	~~1~~ 0
3	~~1~~ 0
4	~~1~~ 0
5	~~2~~ ~~1~~ 0
6	~~1~~ 0
7	~~1~~ 0

Indegree

Now Empty Queue

STOP

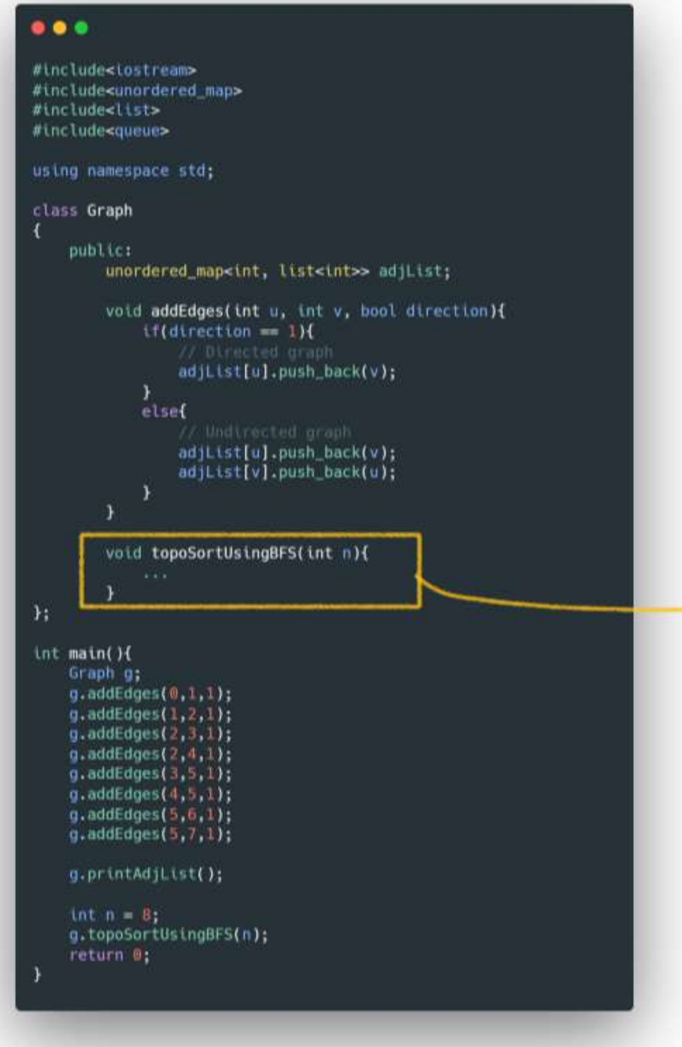
output →

Output

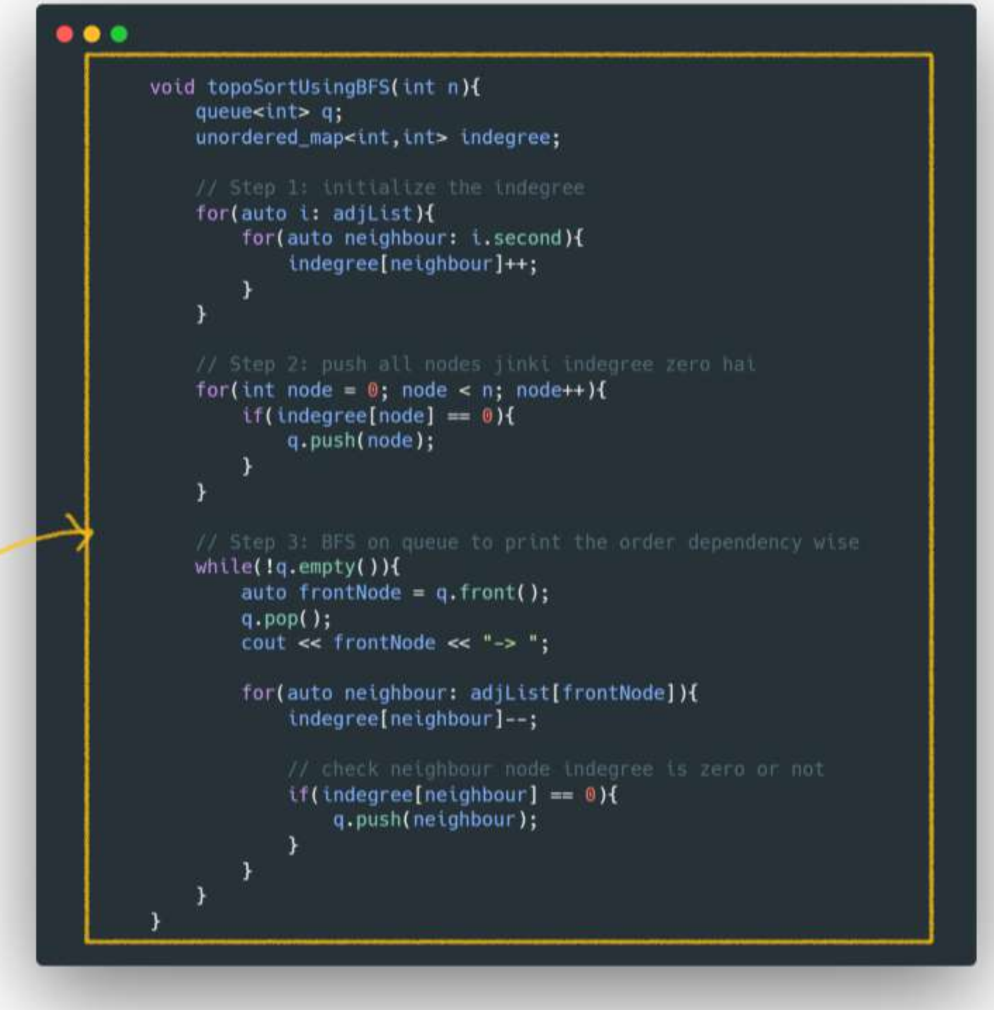
0 1 2 3 4 5 6 7

OR

0 1 2 4 3 5 7 6



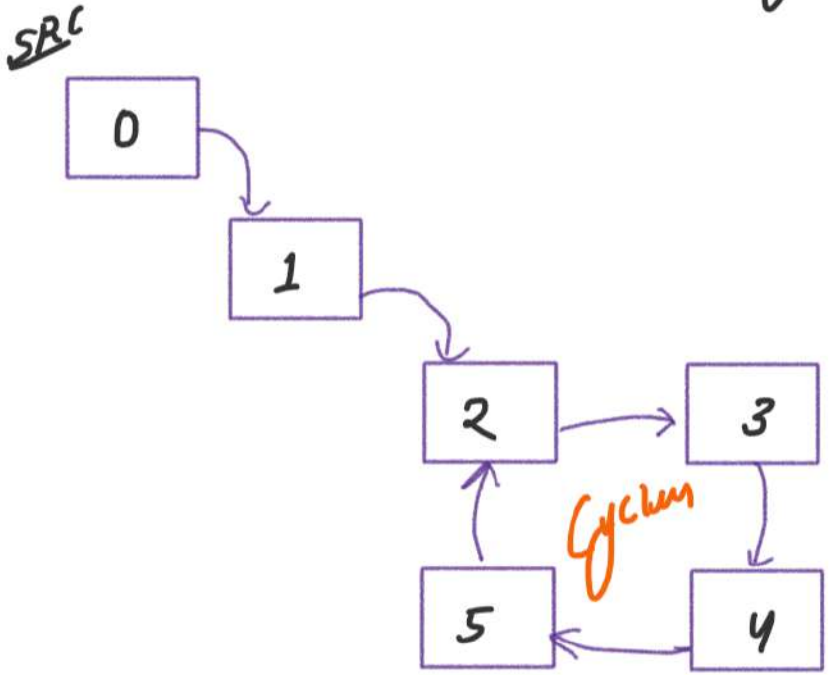
```
#include<iostream>
#include<unordered_map>
#include<list>
#include<queue>

using namespace std;

class Graph
{
    public:
        unordered_map<int, list<int>> adjList;

        void addEdges(int u, int v, bool direction){
            if(direction == 1){
                // Directed graph
                adjList[u].push_back(v);
            }
            else{
                // Undirected graph
                adjList[u].push_back(v);
                adjList[v].push_back(u);
            }
        }

        void topoSortUsingBFS(int n){
            ...
        }
};

int main(){
    Graph g;
    g.addEdges(0,1,1);
    g.addEdges(1,2,1);
    g.addEdges(2,3,1);
    g.addEdges(2,4,1);
    g.addEdges(3,5,1);
    g.addEdges(4,5,1);
    g.addEdges(5,6,1);
    g.addEdges(5,7,1);

    g.printAdjList();

    int n = 8;
    g.topoSortUsingBFS(n);
    return 0;
}
```



```
void topoSortUsingBFS(int n){
    queue<int> q;
    unordered_map<int,int> indegree;

    // Step 1: initialize the indegree
    for(auto i: adjList){
        for(auto neighbour: i.second){
            indegree[neighbour]++;
        }
    }

    // Step 2: push all nodes jinki indegree zero hai
    for(int node = 0; node < n; node++){
        if(indegree[node] == 0){
            q.push(node);
        }
    }

    // Step 3: BFS on queue to print the order dependency wise
    while(!q.empty()){
        auto frontNode = q.front();
        q.pop();
        cout << frontNode << "-> ";

        for(auto neighbour: adjList[frontNode]){
            indegree[neighbour]--;

            // check neighbour node indegree is zero or not
            if(indegree[neighbour] == 0){
                q.push(neighbour);
            }
        }
    }
}
```

4. Detect cycle in a directed graph using BFS

→ using TopoOrder



Create TopoOrder using Topological SORT

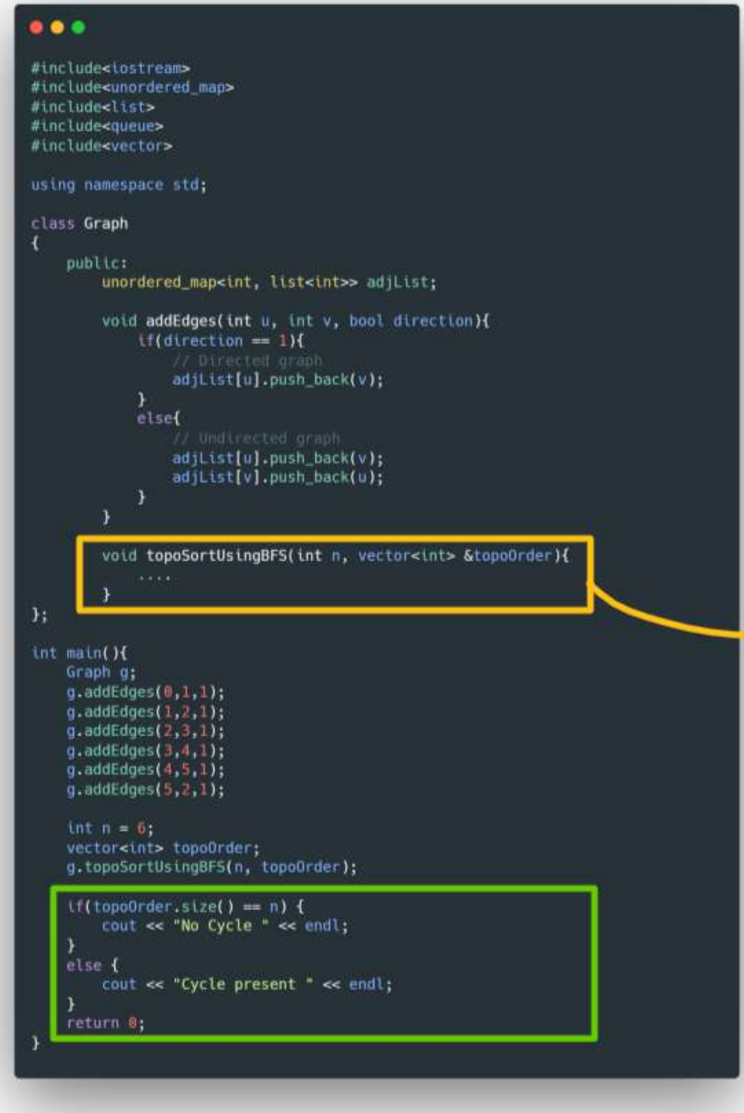
0	1

Key	Value
0	0
1	~~1~~ 0
2	~~2~~ 1
3	1
4	2
5	1

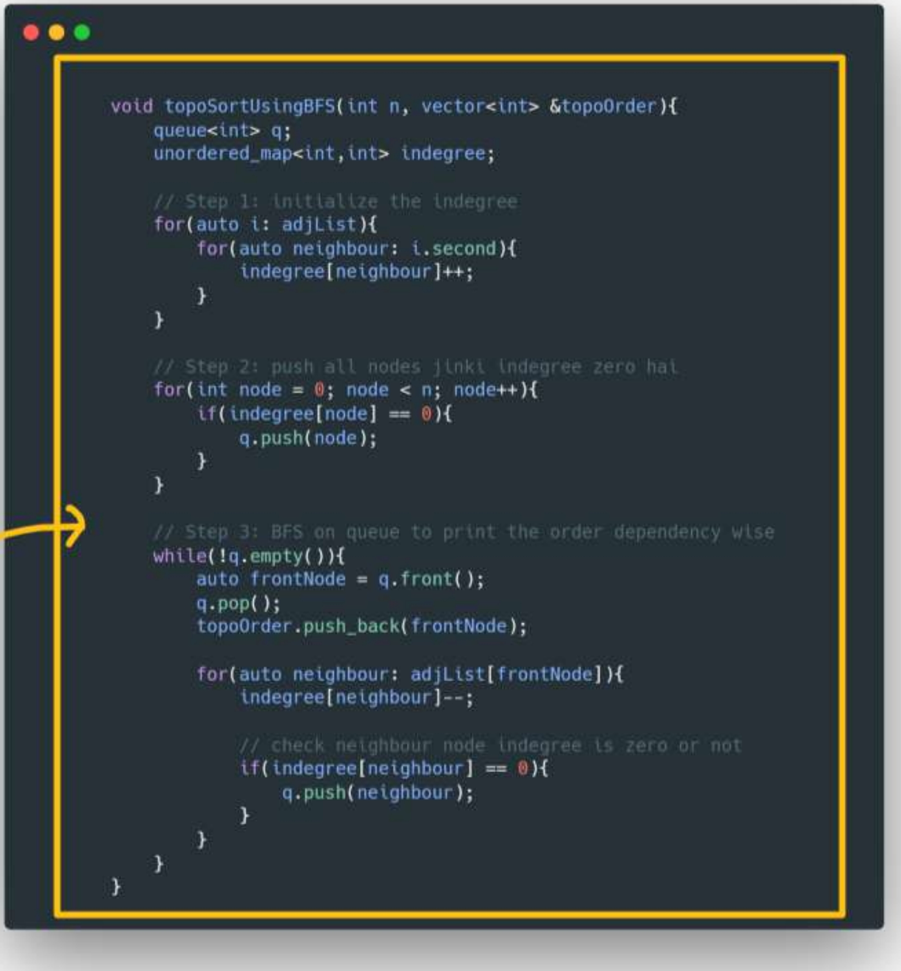
Indegree

Total node > TP.size()

↳ Cycle present Hai

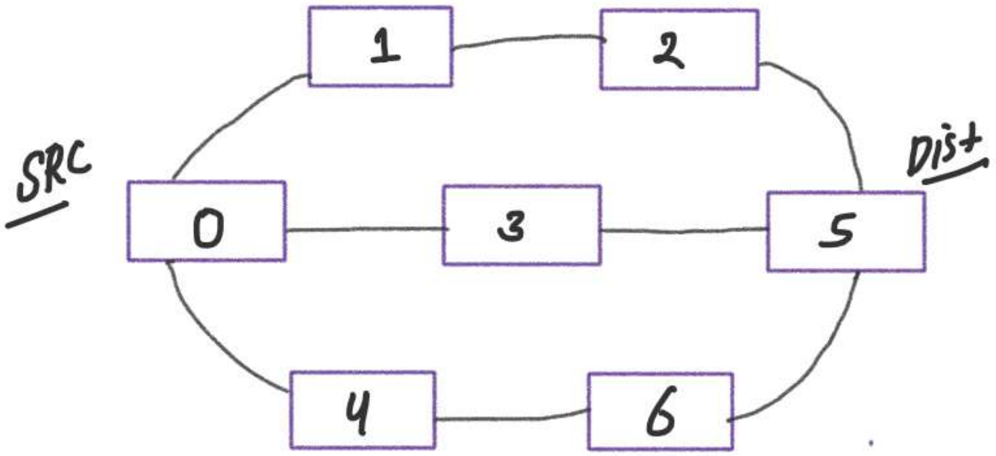


```cpp
#include<iostream>
#include<unordered_map>
#include<list>
#include<queue>
#include<vector>

using namespace std;

class Graph
{
    public:
        unordered_map<int, list<int>> adjList;

        void addEdges(int u, int v, bool direction){
            if(direction == 1){
                // Directed graph
                adjList[u].push_back(v);
            }
            else{
                // Undirected graph
                adjList[u].push_back(v);
                adjList[v].push_back(u);
            }
        }

        void topoSortUsingBFS(int n, vector<int> &topoOrder){
            ....
        }
};

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

    int n = 6;
    vector<int> topoOrder;
    g.topoSortUsingBFS(n, topoOrder);

    if(topoOrder.size() == n) {
        cout << "No Cycle " << endl;
    }
    else {
        cout << "Cycle present " << endl;
    }
    return 0;
}
```



```cpp
void topoSortUsingBFS(int n, vector<int> &topoOrder){
    queue<int> q;
    unordered_map<int,int> indegree;

    // Step 1: initialize the indegree
    for(auto i: adjList){
        for(auto neighbour: i.second){
            indegree[neighbour]++;
        }
    }

    // Step 2: push all nodes jinki indegree zero hai
    for(int node = 0; node < n; node++){
        if(indegree[node] == 0){
            q.push(node);
        }
    }

    // Step 3: BFS on queue to print the order dependency wise
    while(!q.empty()){
        auto frontNode = q.front();
        q.pop();
        topoOrder.push_back(frontNode);

        for(auto neighbour: adjList[frontNode]){
            indegree[neighbour]--;

            // check neighbour node indegree is zero or not
            if(indegree[neighbour] == 0){
                q.push(neighbour);
            }
        }
    }
}
```

5. Shortest path in an undirected graph using BFS



Tips To get the shortest path:

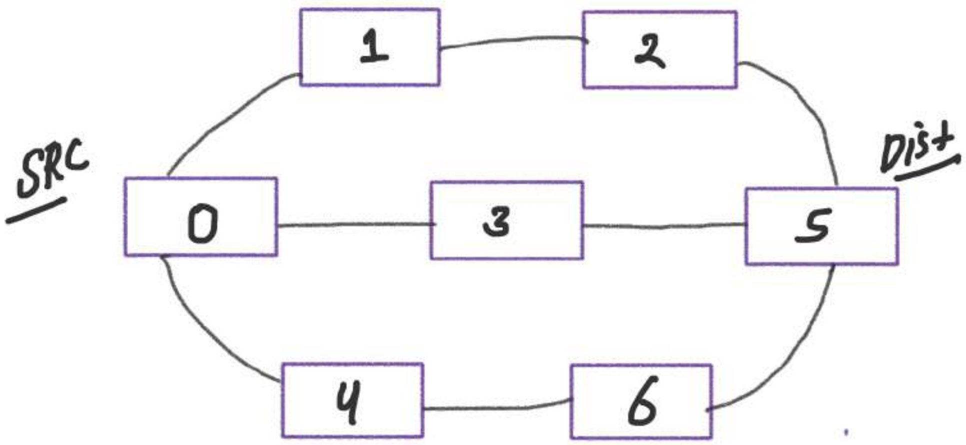
↳ jab Bhi pahli bar SRC Node se kisi dusre Node ko Hum Traverse karenge wohi Hamare graph ko shortest path Hoga in case of undirected graph-

Path1 ⇒ 0 — 1 — 2 — 5

path2 ⇒ 0 — 3 — 5 → output = 0 3 5

path3 ⇒ 0 — 4 — 6 — 5

Total Edges = 2



initial state of BFS

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

key	value
0	{1,4,3}
1	{0,2}
2	{1,5}
3	{0,5}
4	{0,6}
5	{2,6}
6	{4,5}

AdjList

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

visited

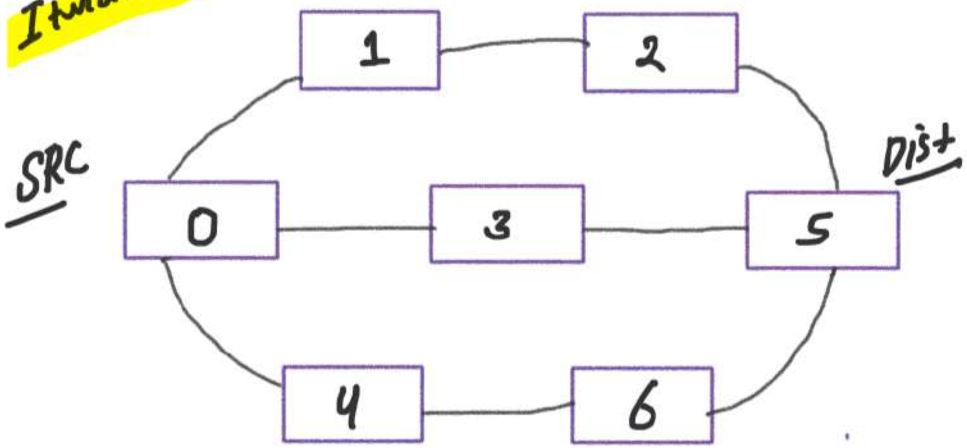
key	value
0	-1
1	
2	
3	
4	
5	
6	

Parent

0	

Queue

Iteration 1



key	value
0	{1,4,3}
1	{0,2}
2	{1,5}
3	{0,5}
4	{0,6}
5	{2,6}
6	{4,5}

AdjList

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

Visited

key	value
0	-1
1	0
2	
3	0
4	0
5	
6	

Parent

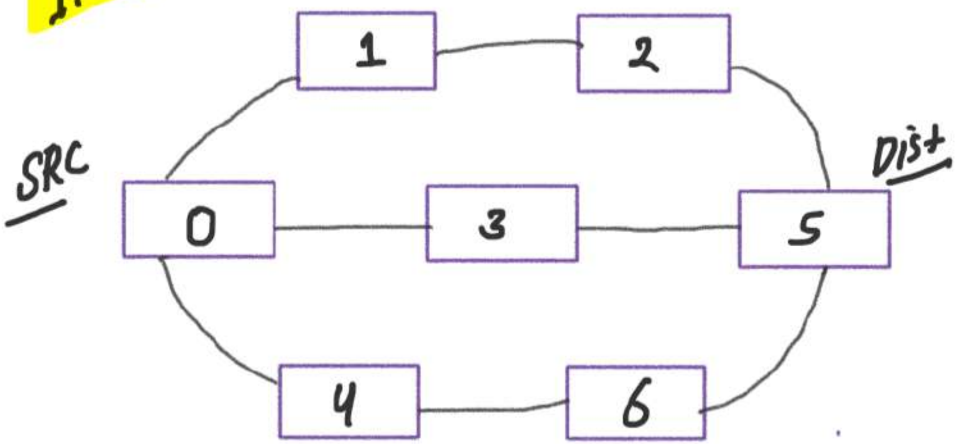
0^{x}	1	4	3

Queue

child of frontNode not visited
→ q.push(child)
visited[child]=T
parent[child]=frontNode

child visited
→ Ignore

Iteration2



key	value
0	{1,4,3}
1	{0,2}
2	{1,5}
3	{0,5}
4	{0,6}
5	{2,6}
6	{4,5}

AdjList

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

visited

key	value
0	-1
1	0
2	1
3	0
4	0
5	
6	

parent

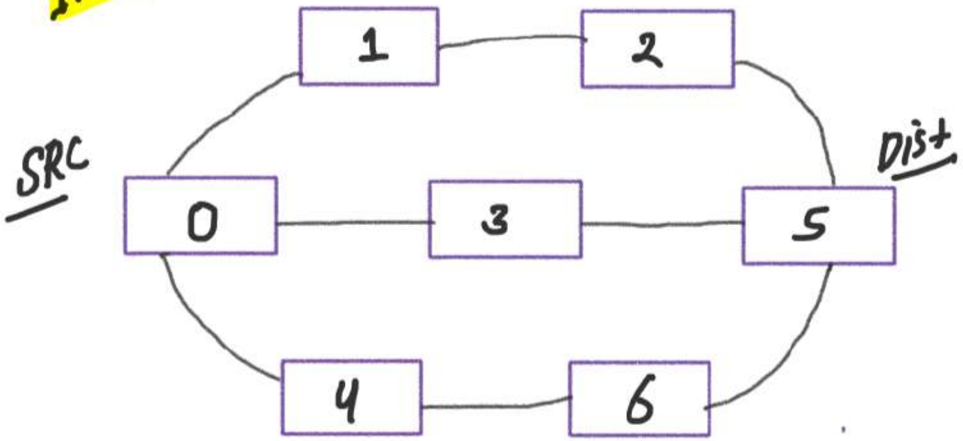
child of frontNode not visited
- q.push(child)
- visited[child]=T
- parent[child]=frontNode

child visited
- Ignore

~~1~~	4	3	2

Queue

Iteration3



key	value
0	{1,4,3}
1	{0,2}
2	{1,5}
3	{0,5}
4	{0,6}
5	{2,6}
6	{4,5}

AdjList

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

visited

key	value
0	-1
1	0
2	1
3	0
4	0
5	
6	4

Parent

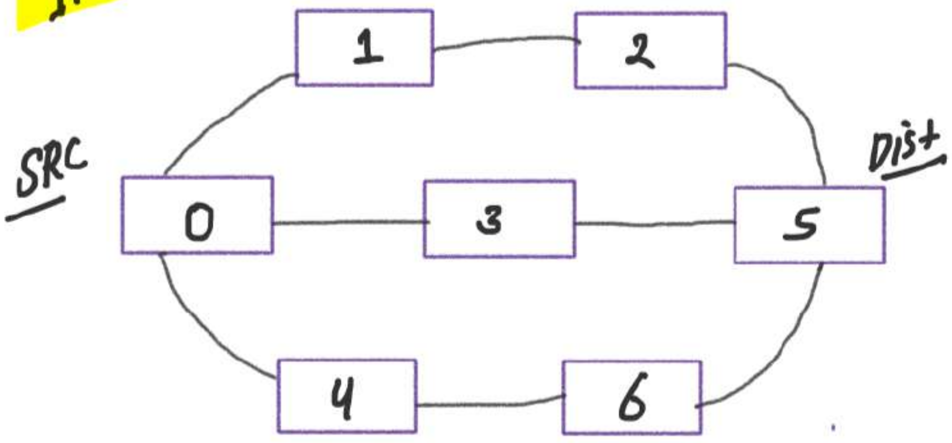
~~4~~	3	2	6	

Queue

child of frontNode not visited
→ q.push(child)
visited(child)=T
parent[child]=frontNode

child visited
→ Ignore

Itrationn



```
child of frontNode not visited
    ↳ q.push(child)
      visited[child]=T
      parent[child]=frontNode

child visited
    ↳ Ignore
```

key	value
0	{1,4,3}
1	{0,2}
2	{1,5}
3	{0,5}
4	{0,6}
5	{2,6}
6	{4,5}

AdjList

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

Visited

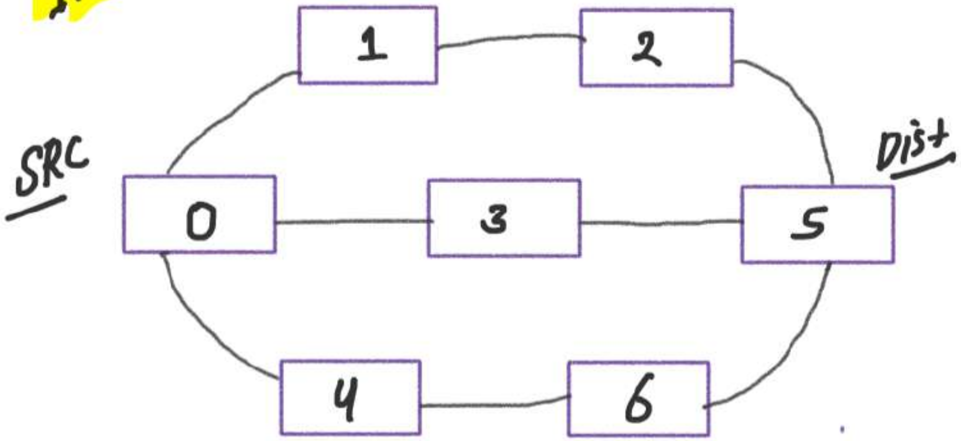
key	value
0	-1
1	0
2	1
3	0
4	0
5	3
6	4

Parent

3^{x}	2	6	5

Queue

Iterations 5, 6, 7



key	value
0	{1,4,3}
1	{0,2}
2	{1,5}
3	{0,5}
4	{0,6}
5	{2,6}
6	{4,5}

AdjList

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

visited

key	value
0	-1
1	0
2	1
3	0
4	0
5	3
6	4

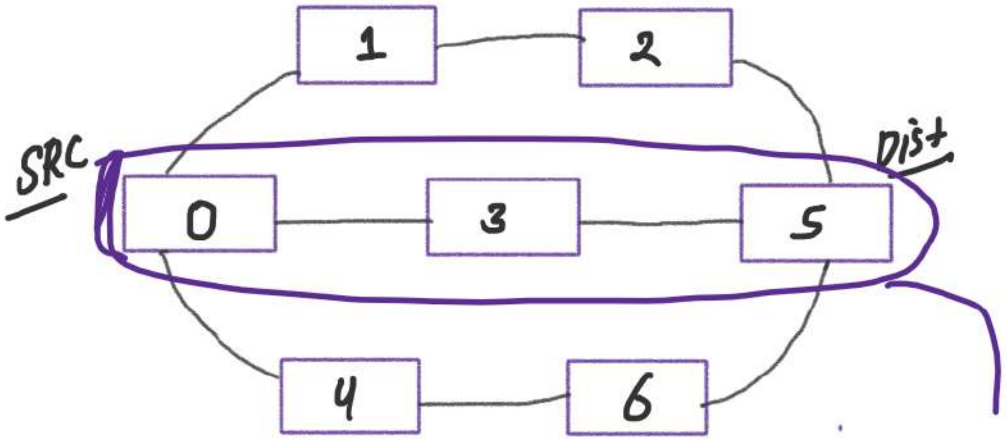
Parent

[Parent map created Ho Chuka Hai] Now Time start to make a shortest path ↓

~~2~~	~~6~~	~~5~~ Empty

Queue

Make shortest path using parent map



```
while(dist != -1){
    Ans.push_back(dist);
    dist = parent[dist];
}
```

key	value
0	-1
1	0
2	1
3	0
4	0
5	3
6	4

parent

Ans

5	3	0

① dist = 5

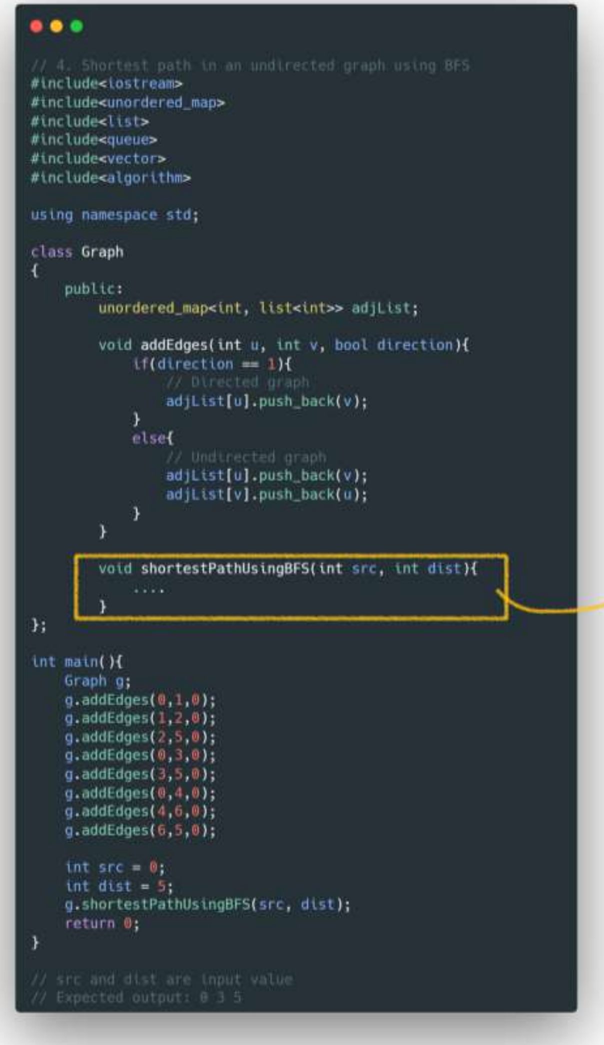
② dist = parent[5] ⇒ 3

③ dist = parent[3] ⇒ 0

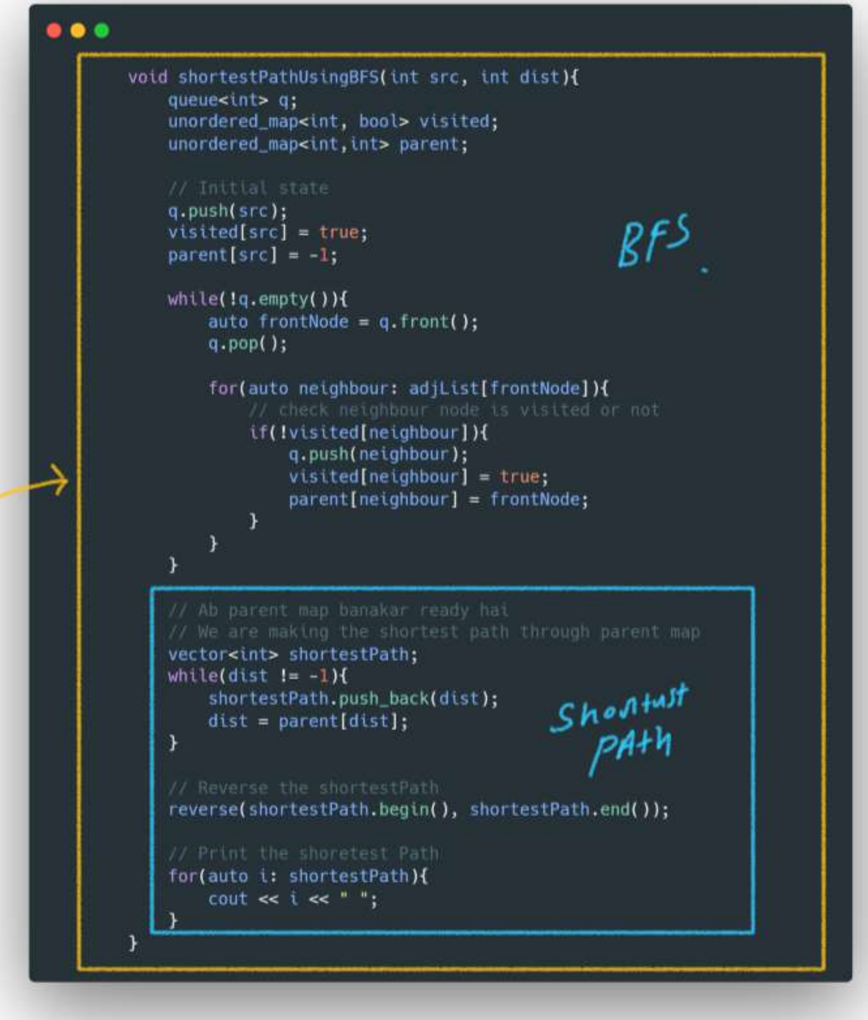
④ dist = parent [0] ⇒ -1 → STOP

Return the reverse of Array

0 → 3 → 5



```cpp
// 4. Shortest path in an undirected graph using BFS
#include<iostream>
#include<unordered_map>
#include<list>
#include<queue>
#include<vector>
#include<algorithm>

using namespace std;

class Graph
{
    public:
        unordered_map<int, list<int>> adjList;

        void addEdges(int u, int v, bool direction){
            if(direction == 1){
                // Directed graph
                adjList[u].push_back(v);
            }
            else{
                // Undirected graph
                adjList[u].push_back(v);
                adjList[v].push_back(u);
            }
        }

        void shortestPathUsingBFS(int src, int dist){
            ....
        }
};

int main(){
    Graph g;
    g.addEdges(0,1,0);
    g.addEdges(1,2,0);
    g.addEdges(2,5,0);
    g.addEdges(0,3,0);
    g.addEdges(3,5,0);
    g.addEdges(0,4,0);
    g.addEdges(4,6,0);
    g.addEdges(6,5,0);

    int src = 0;
    int dist = 5;
    g.shortestPathUsingBFS(src, dist);
    return 0;
}

// src and dist are input value
// Expected output: 0 3 5
```



```cpp
void shortestPathUsingBFS(int src, int dist){
    queue<int> q;
    unordered_map<int, bool> visited;
    unordered_map<int,int> parent;

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

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

        for(auto neighbour: adjList[frontNode]){
            // check neighbour node is visited or not
            if(!visited[neighbour]){
                q.push(neighbour);
                visited[neighbour] = true;
                parent[neighbour] = frontNode;
            }
        }
    }

    // Ab parent map banakar ready hai
    // We are making the shortest path through parent map
    vector<int> shortestPath;
    while(dist != -1){
        shortestPath.push_back(dist);
        dist = parent[dist];
    }

    // Reverse the shortestPath
    reverse(shortestPath.begin(), shortestPath.end());

    // Print the shoretest Path
    for(auto i: shortestPath){
        cout << i << " ";
    }
}
```