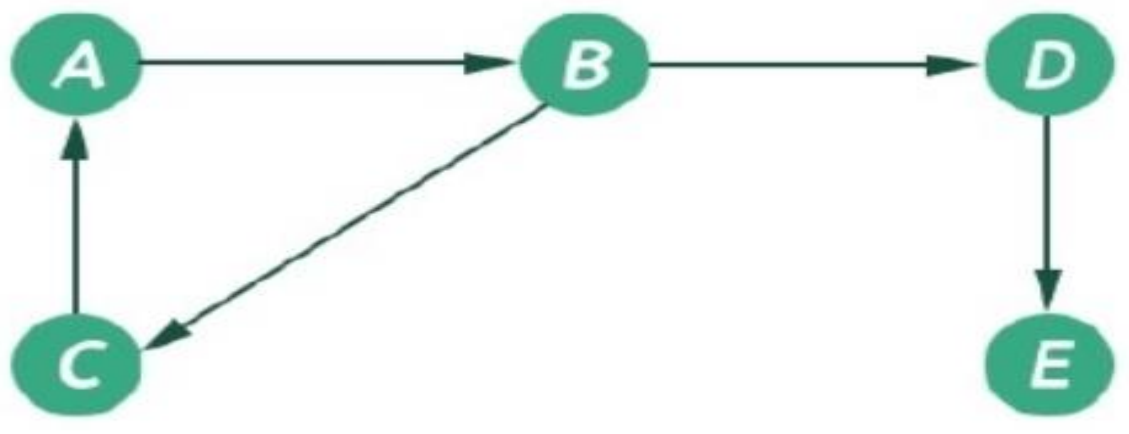
TRAVERSAL TECHNIQUE FOR GRAPHS

GRAPH TRAVERSAL - **Graph traversal refers to the process of visiting each vertex or node in a graph. A smaller graph seems relatively easy to traverse. But when it comes to traversing a graph with a huge number of nodes, the process needs to be automated. Doing things manually increases the chances of missing some vertices or so. And visiting nodes of a graph becomes important when you need to change something for some nodes, or just need to retrieve the value present there or something else. Sequences of steps that are used to traverse a graph are known as graph traversal algorithms.The graph is one non-linear data structure. That is consists of some nodes and their connected edges. The edges may be director or undirected. This graph can be represented as G(V, E). The following graph can be represented as G({A, B, C, D, E}, {(A, B), (B, D), (D, E), (B, C), (C, A)}).**



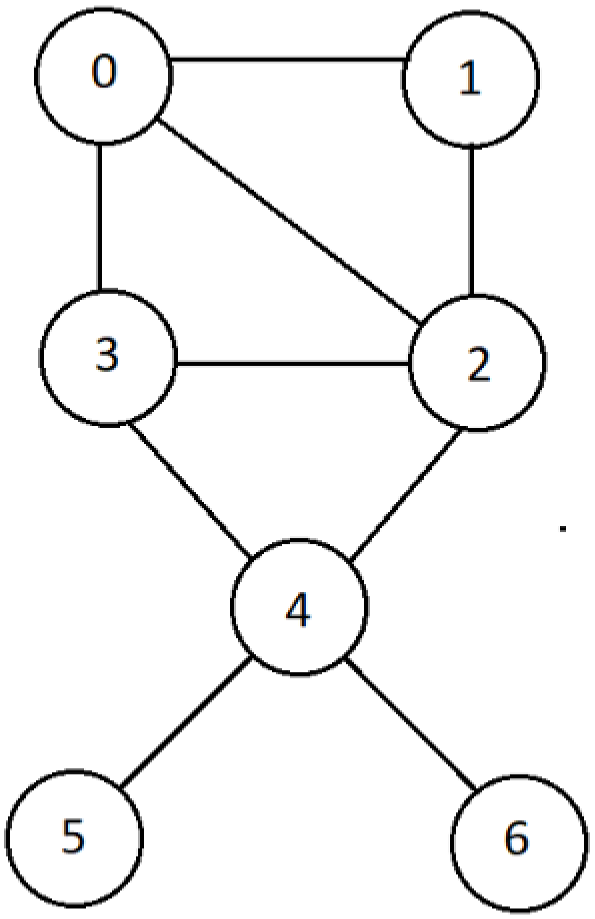
Sequences of steps that are used to traverse a graph are known as graph traversal algorithms. There are two algorithms that are used for traversing a graph. They are:

1. Breadth-First Search (BFS)
2. Depth First Search (DFS)

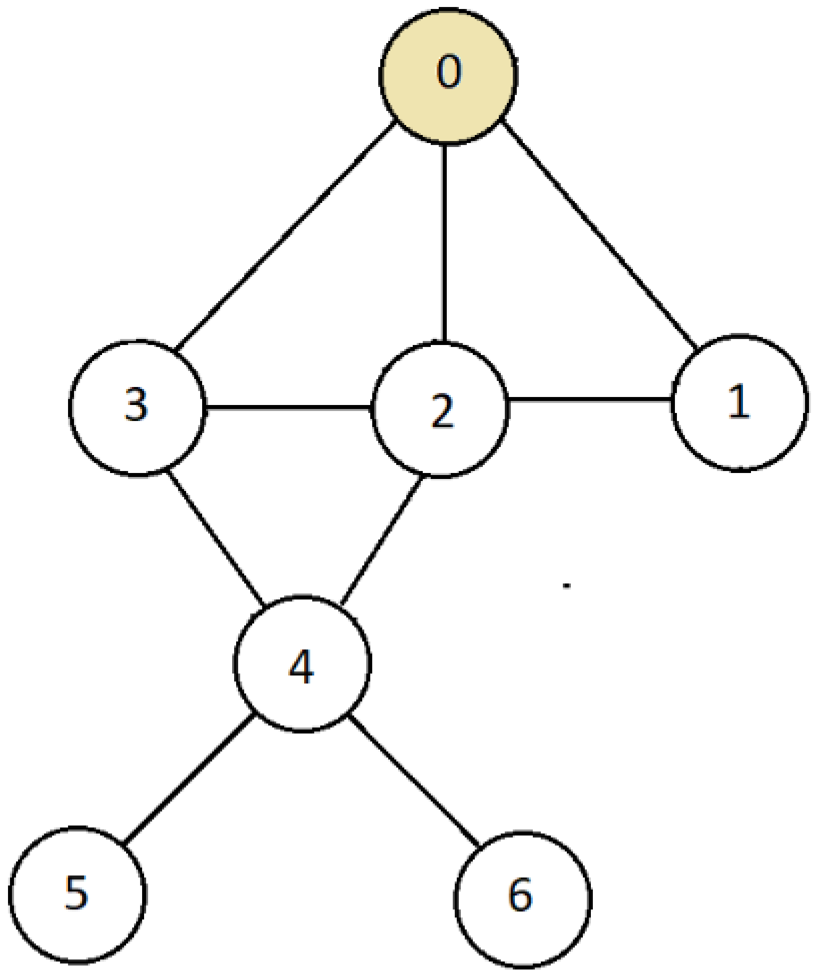
Breadth-First Search - In Breadth-First Search, we start with a node (not necessarily the smallest or the largest) and start exploring its connected nodes. The same process is repeated with all the connecting nodes until all the nodes are visited.

Method 1: BFS Spanning Tree:

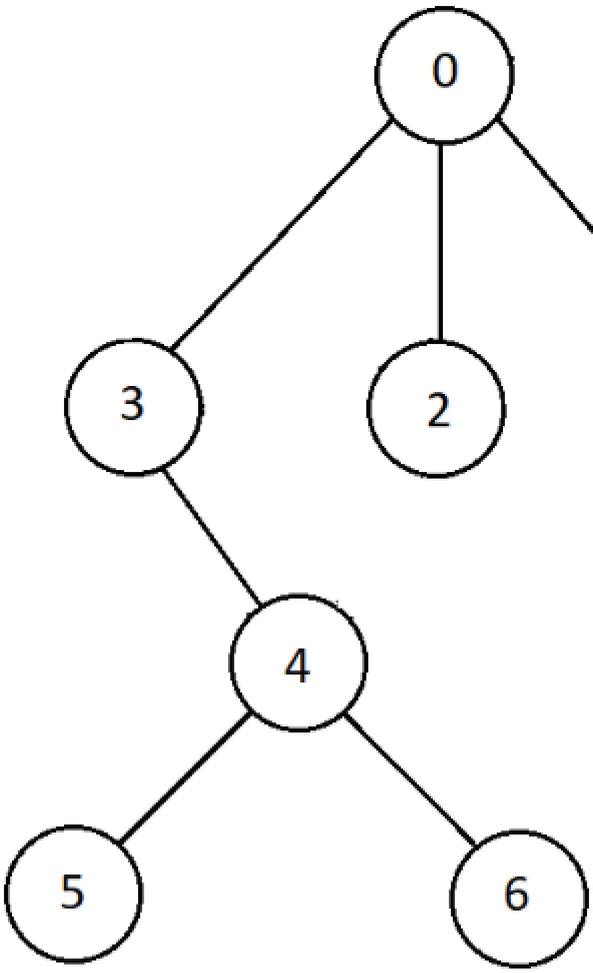
To understand what BFS Spanning Tree means, consider the graph.



Now, choose any node, say 0, and try to construct a tree with this chosen node as its root. So, the graph would now look something like this.



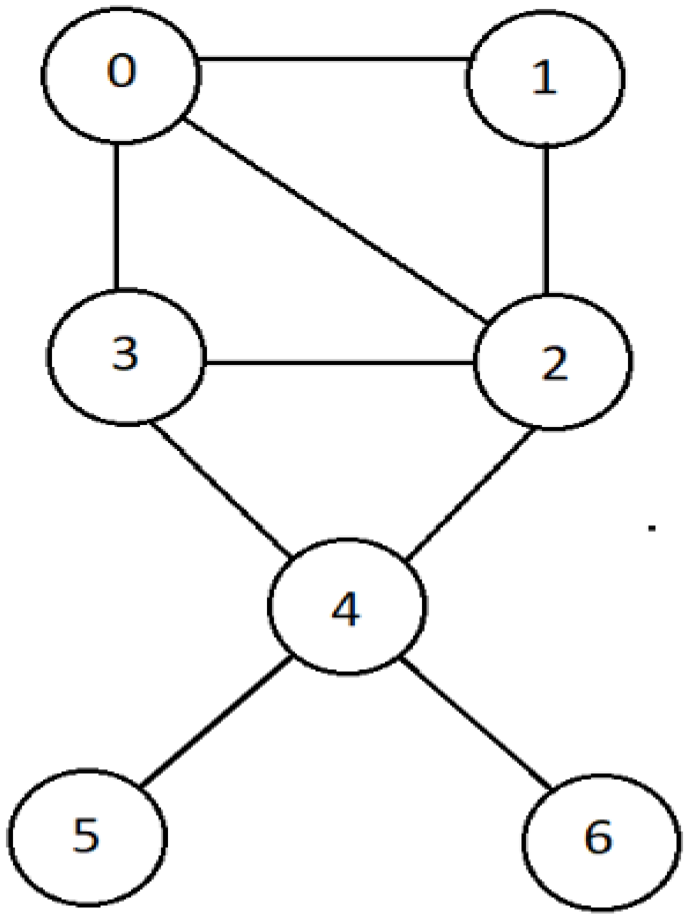
Now, mark dashed or simply remove all the edges which are either sideways or duplicate (above a node) to turn this graph into a valid tree, and as you know for a graph to be a tree, it shouldn’t have any cycle. So, we can remove the edges between nodes 2 and 3, and then between nodes 1 and 2 being sideways. Then also between 2 and 4. You could have rather removed the one between node 3 and 4 instead of 2 and 4, but both ways work since these are duplicate to node 4. The tree we receive after we do these above-mentioned changes is,



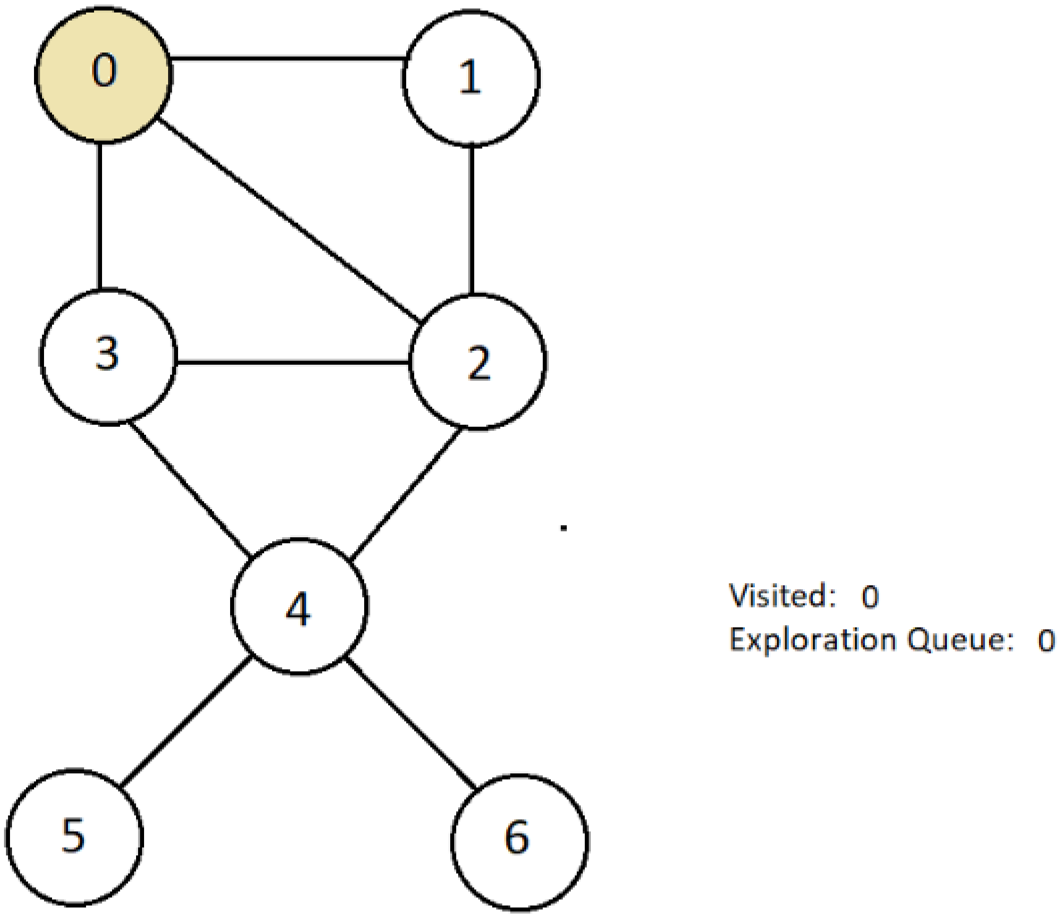
This constructed tree above is known as a BFS Spanning Tree , the level order traversal of this BFS spanning tree gives us the Breadth-First Search traversal of the graph and the level order traversal of a tree is, we simply write the nodes in the same level from left to right. So, the level order traversal of the above BFS Spanning Tree is 0, 3, 2, 1, 4, 5, 6.
And a BFS Spanning Tree is not unique to a graph. We could have removed, as discussed above, the edge between nodes 3 and 4, instead of nodes 2 and 4. That would have yielded a different BFS Spanning Tree.

Method 2: Conventional Breadth-First Search Traversal Algorithm:

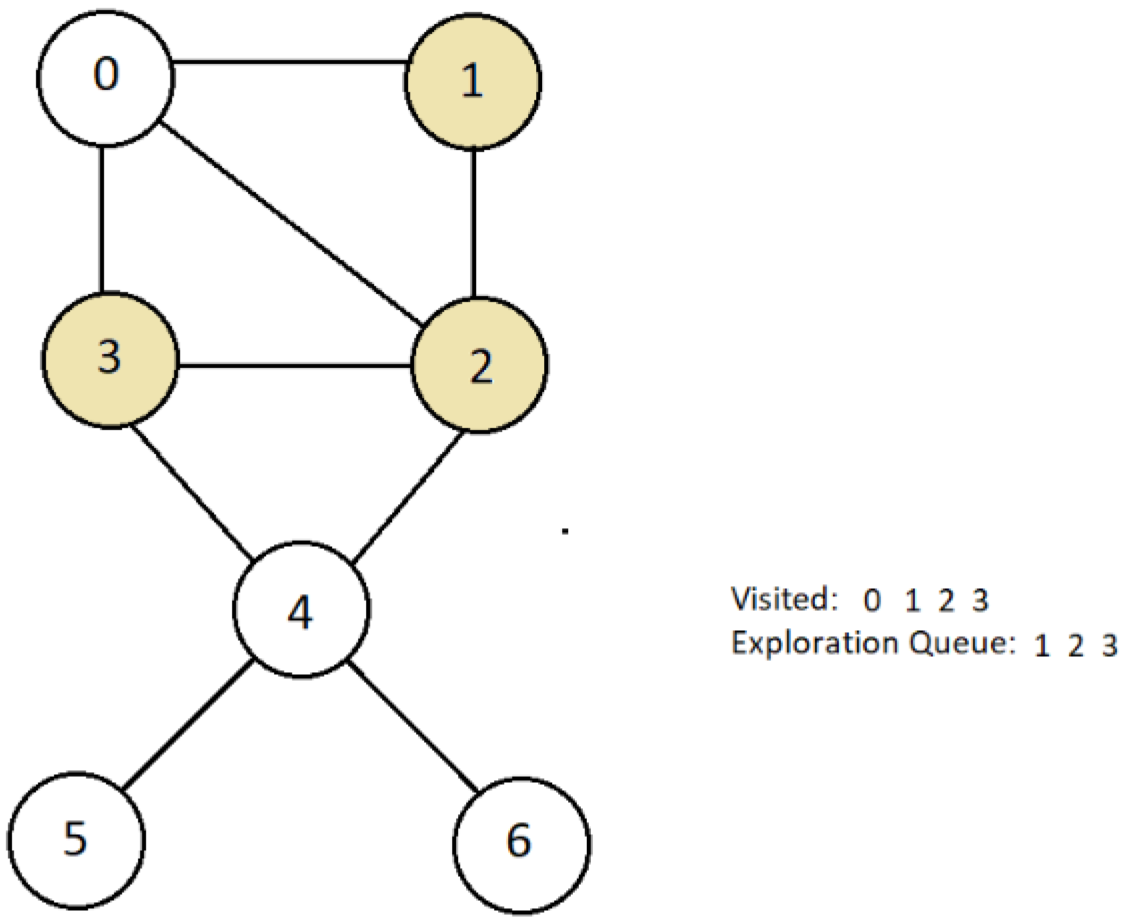
Consider the same graph we've seen above.



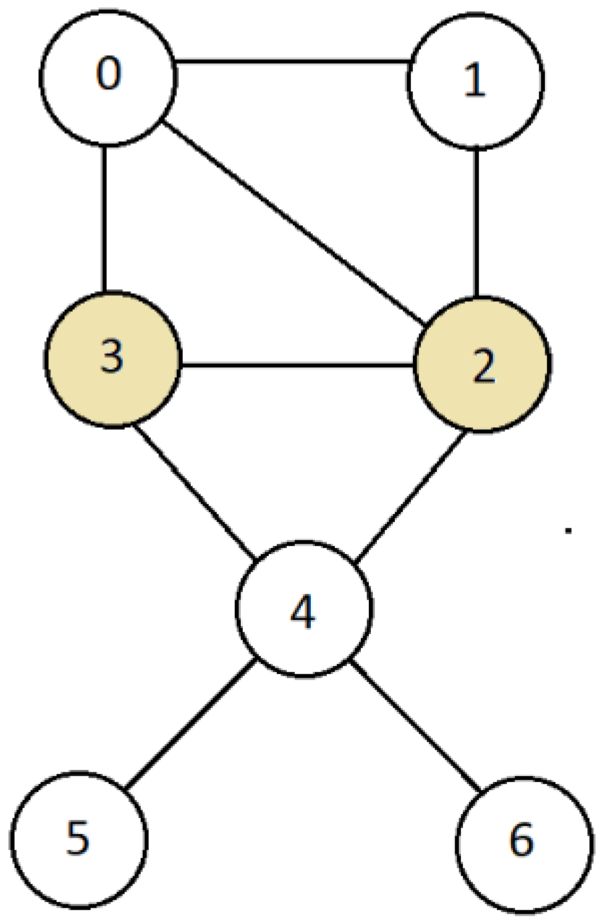
Considering we could begin with any source node; we'll start with 0 only. Let's define a queue named **exploration queue** which would hold the nodes we'll be exploring one by one. We would maintain another array holding the status of whether a node is already **visited** or not. Since we are starting with node 0, we would enqueue 0 into our exploration queue and mark it visited.



Now, we'll start visiting all the nodes connected to node 0, and remove node 0 from the exploration queue, enqueuing all the currently visited nodes which were nodes 1, 2, and 3. We are pushing them inside the exploration queue because these might further have some unvisited nodes connected to them.

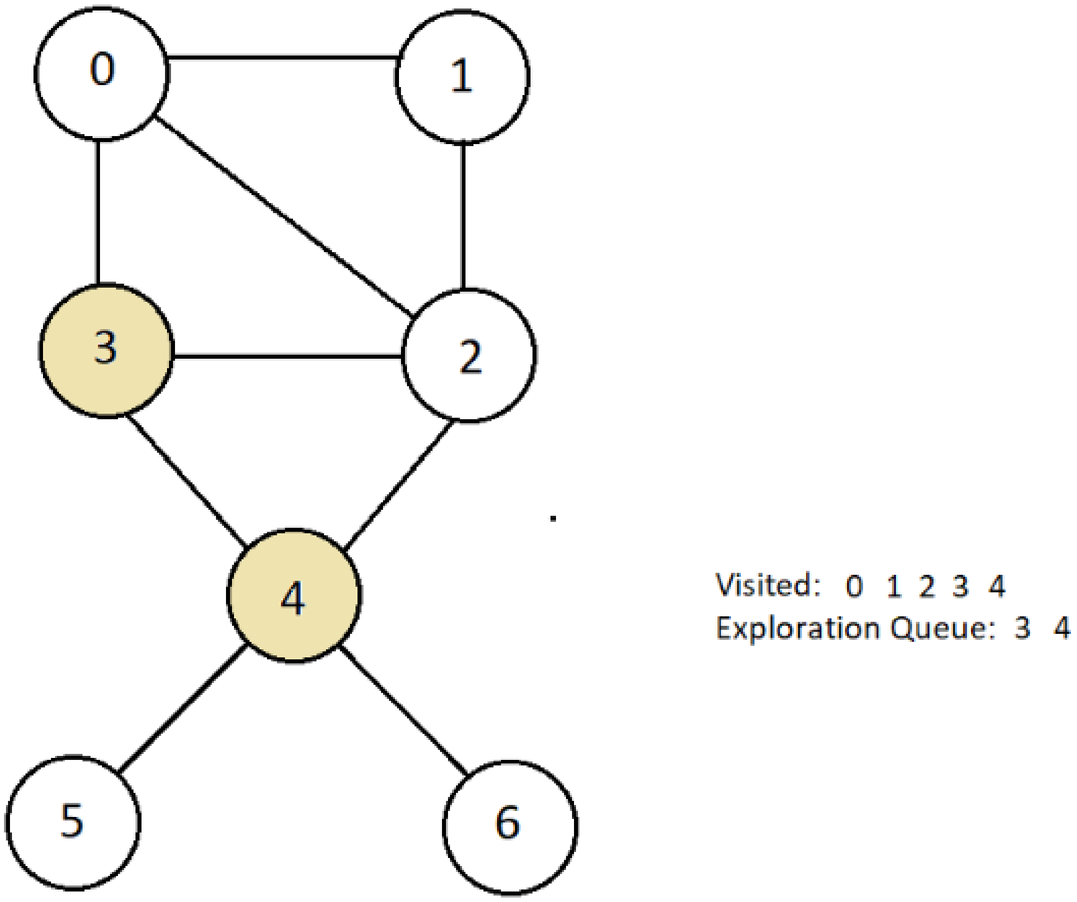


After this, we have node 1 at the top in the exploration queue, so we'll take it out and visit all unvisited nodes connected to it.



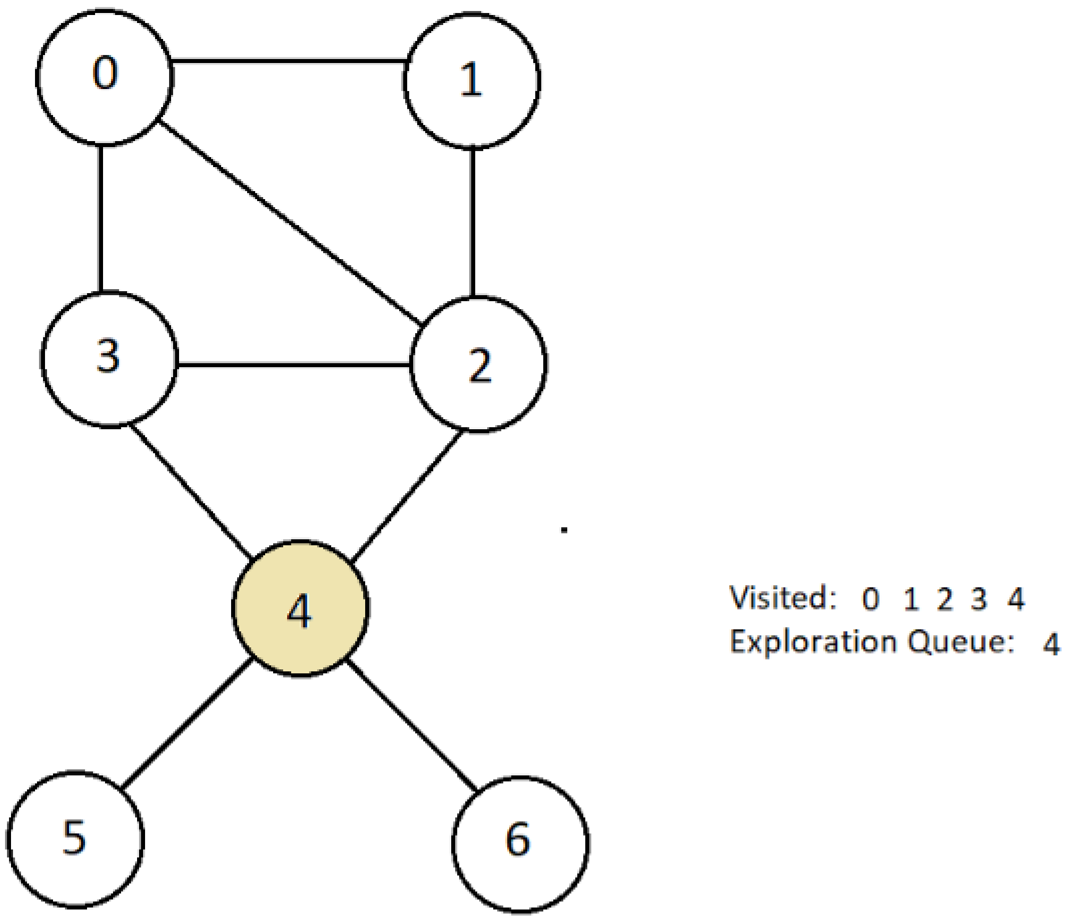
Visited: 0 1 2 3
Exploration Queue: 2 3

Next, we have node 2. And the only unvisited node connected to node 2 is node 4. So, we'll mark it visited and will also enqueue it in our exploration queue.

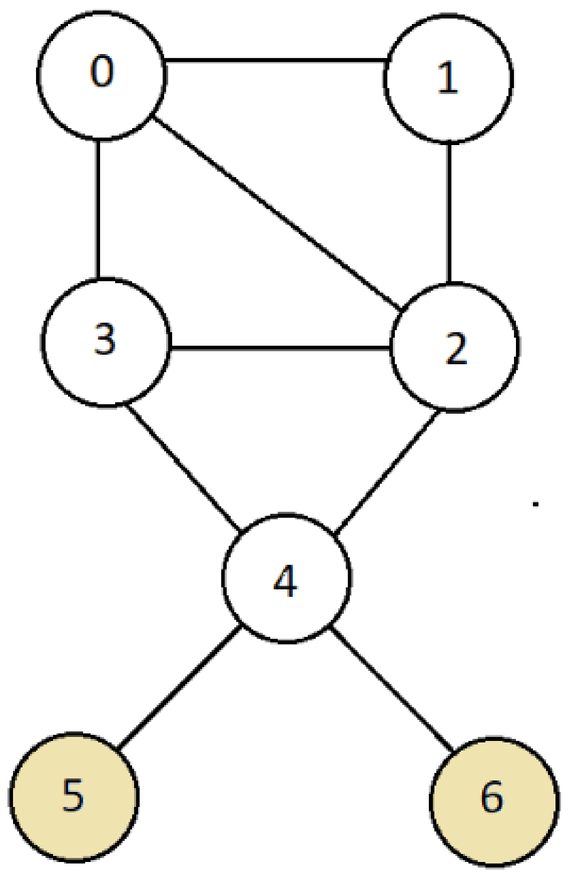


Visited: 0 1 2 3 4
Exploration Queue: 3 4

Node 3 is the next in line. Since, all nodes 1, 2, and 4 which are the nodes connected to it are already visited.

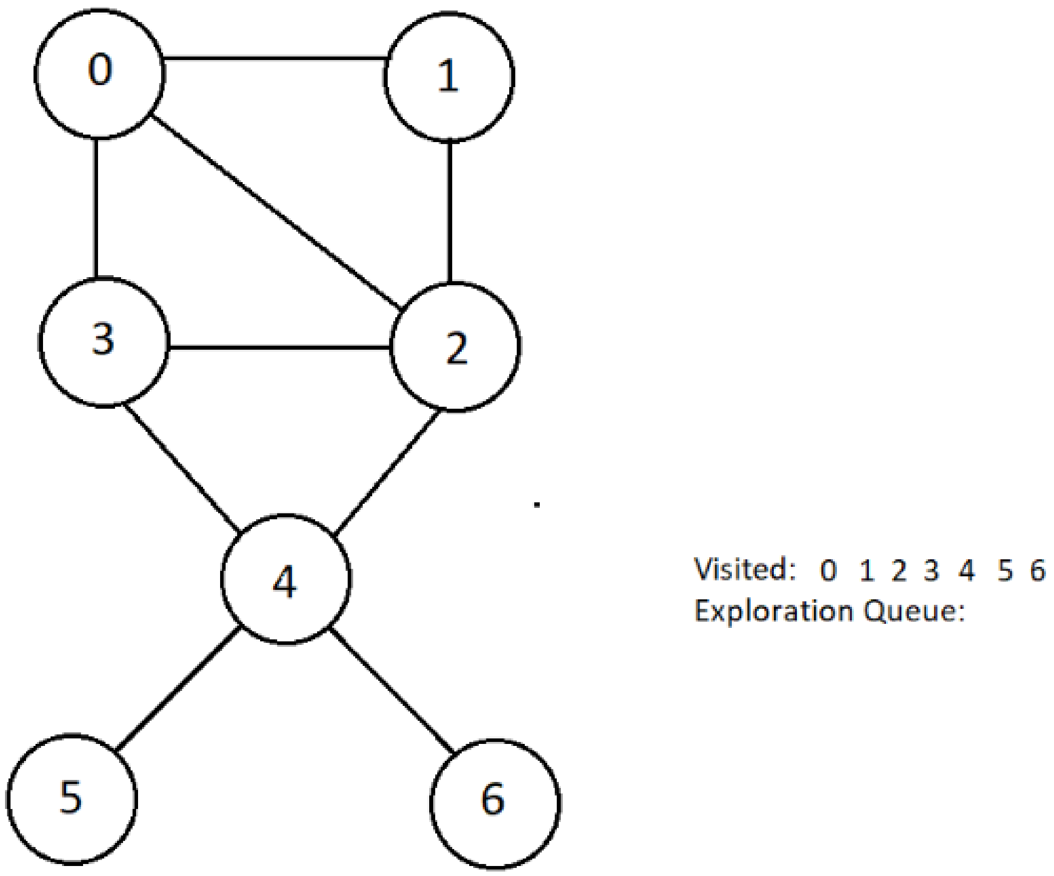


Next, we have node 4 on the top in the exploration queue. Let's get it out and see what nodes are connected and unvisited to it. So, we got nodes 5 and 6. Mark them visited and push them inside the exploration queue.



Visited: 0 1 2 3 4 5 6
Exploration Queue: 5 6

And now we can explore the other two nodes left in the queue, and since all nodes are already visited. And this got our queue emptied and every node traversed in Breadth-First Search manner.



And the order in which we marked our nodes visited is the Breadth-First Search traversal order. Here, it is **0, 1, 2, 3, 4, 5, 6.** So basically, the visited array maintains whether the node itself is visited or not, and the exploration queue maintains whether the nodes connected to a node are visited or not.

ALGORITHM:

1)Now, the first thing we did was, we took the input. The input comprises the information concerning the graph, its nodes/vertices, and edges, and the source node we'll start the traversal with.

2)Then, we'll mark the source node s visited and then create an exploration queue, and enqueue the source code s in it.

3)We'll then initiate a while loop which will run until the queue is not empty. At each iteration, we will take out the first element out of the queue using the utility function dequeue and suppose that element is u. Then, we visit all the vertices which are directly connected to u and are already not visited.

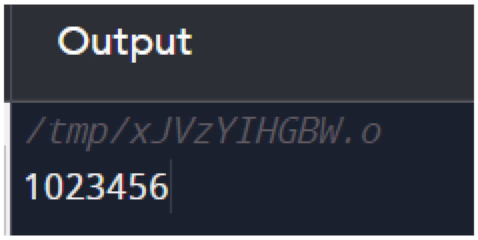
4)And every time we visit a new node, we enqueue them in our exploration queue. Eventually, the queue becomes empty, and our traversal ends. And the visited array is the order of our Breadth-First Search traversal.

CODE FOR BFS:

```
#include<stdio.h>

#include<stdlib.h>


struct queue

{

    int size;

    int f;

    int r;

    int* arr;

};




int isEmpty(struct queue *q){

    if(q->r==q->f){

        return 1;
    }
    return 0;
}

int isFull(struct queue *q){
    if(q->r==q->size-1){
        return 1;
    }
    return 0;
}

void enqueue(struct queue *q, int val){
    if(isFull(q)){
        printf("This Queue is full\n");
    }
    else{
        q->r++;
        q->arr[q->r] = val;
        // printf("Enqued element: %d\n", val);
    }
}

int dequeue(struct queue *q){
    int a = -1;
    if(isEmpty(q)){
        printf("This Queue is empty\n");
    }
    else{
        q->f++;
        a = q->arr[q->f];
    }
    return a;
}

int main(){
    // Initializing Queue (Array Implementation)
    struct queue q;
    q.size = 400;
    q.f = q.r = 0;
    q.arr = (int*) malloc(q.size*sizeof(int));

    // BFS Implementation
    int node;
    int i = 1;
    int visited[7] = {0,0,0,0,0,0,0};
    int a [7][7] = {
    }


        {0,1,1,1,0,0,0},

        {1,0,1,0,0,0,0},

        {1,1,0,1,1,0,0},

        {1,0,1,0,1,0,0},

        {0,0,1,1,0,1,1},

        {0,0,0,0,1,0,0},

        {0,0,0,0,1,0,0}

    };

    printf("%d", i);

    visited[i] = 1;

    enqueue(&q, i); // Enqueue i for exploration

    while (!isEmpty(&q))

    {

        int node = dequeue(&q);

        for (int j = 0; j < 7; j++)

        {

            if(a[node][j] ==1 && visited[j] == 0){

                printf("%d", j);

                visited[j] = 1;

                enqueue(&q, j);

            }

        }

    }
```

```
    return 0;
}
```

OUTPUT:



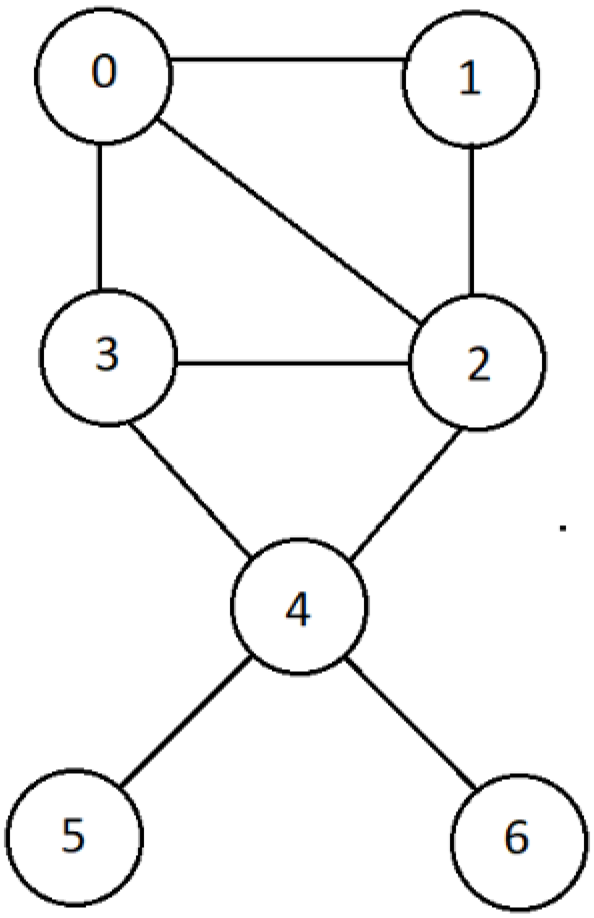
```
/tmp/xJVzYIHGBW.o
1023456
```

Depth First Search (DFS) Graph Traversal:

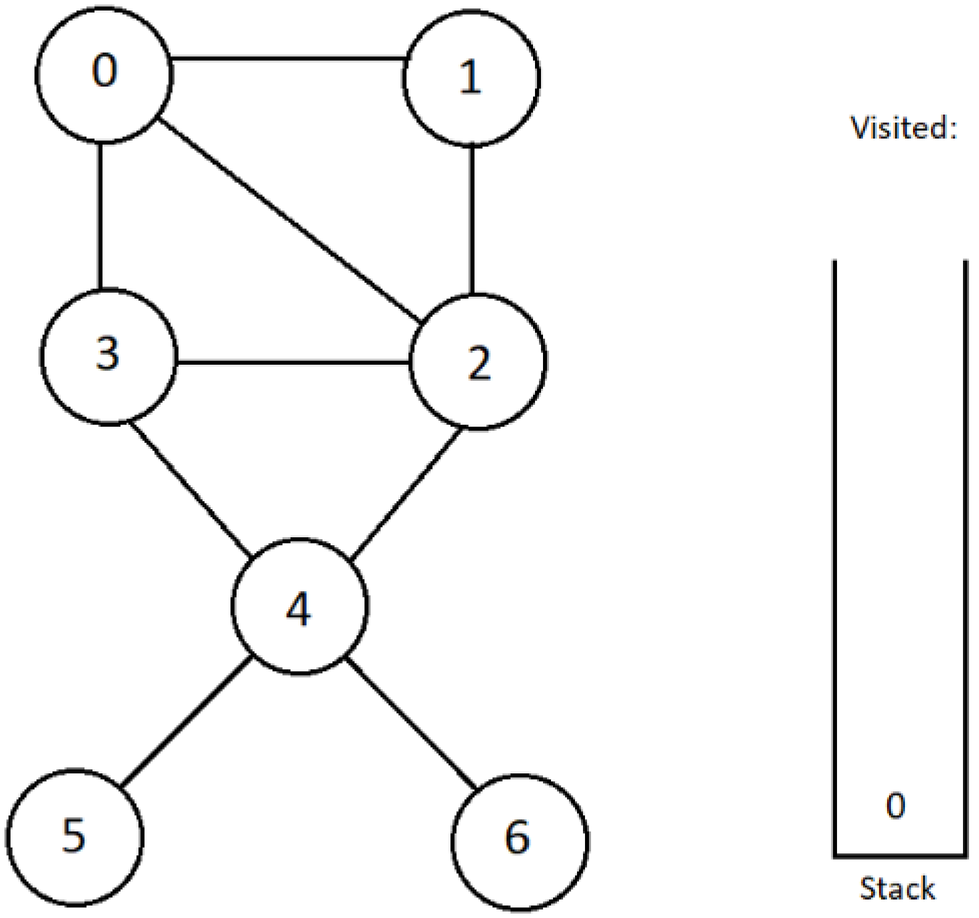
In Depth First Search, we start with a node and start exploring its connected nodes, keeping on suspending the exploration of previous vertices. And those suspended nodes are explored once we finish exploring the node below. And this way we explore all the nodes of the graph.

But In BFS, we explore all the nodes connected to a node, then explore the nodes we visited in a horizontal manner, while in DFS, we start with the first connected node, and similarly go deep, so this looks like visiting the nodes vertically.

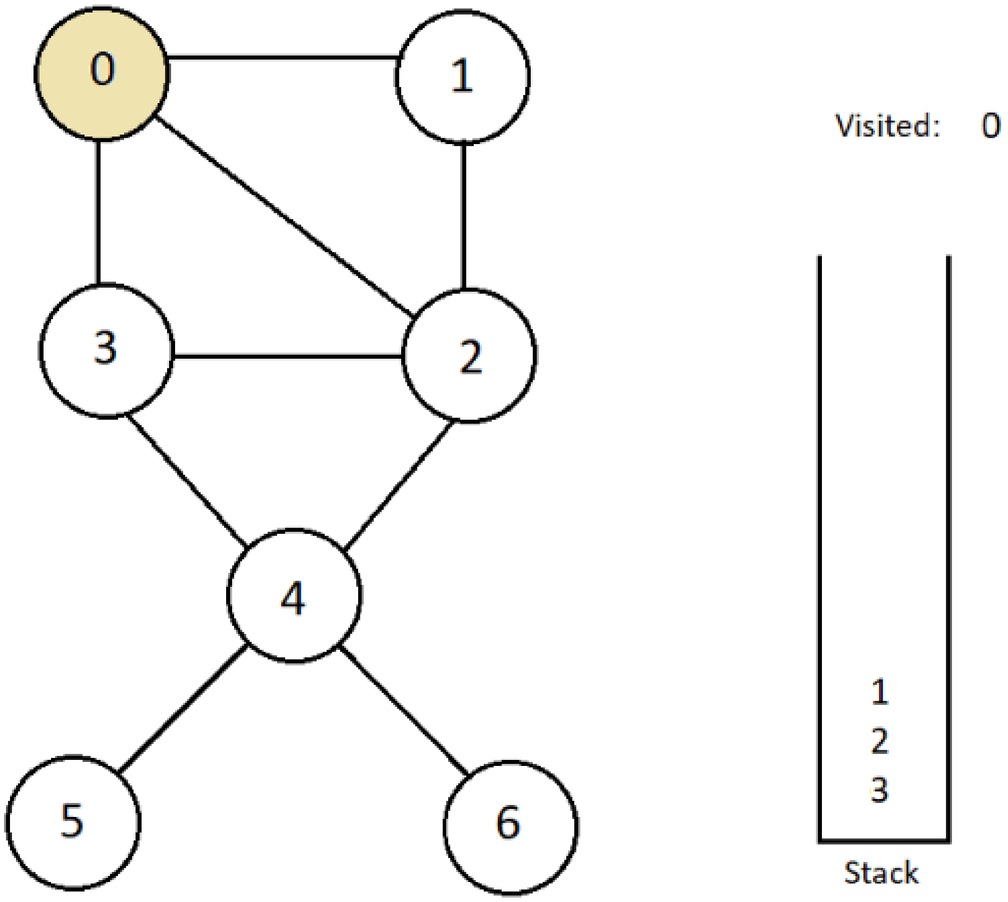
To understand the procedure of the Depth First Search, consider the graph.



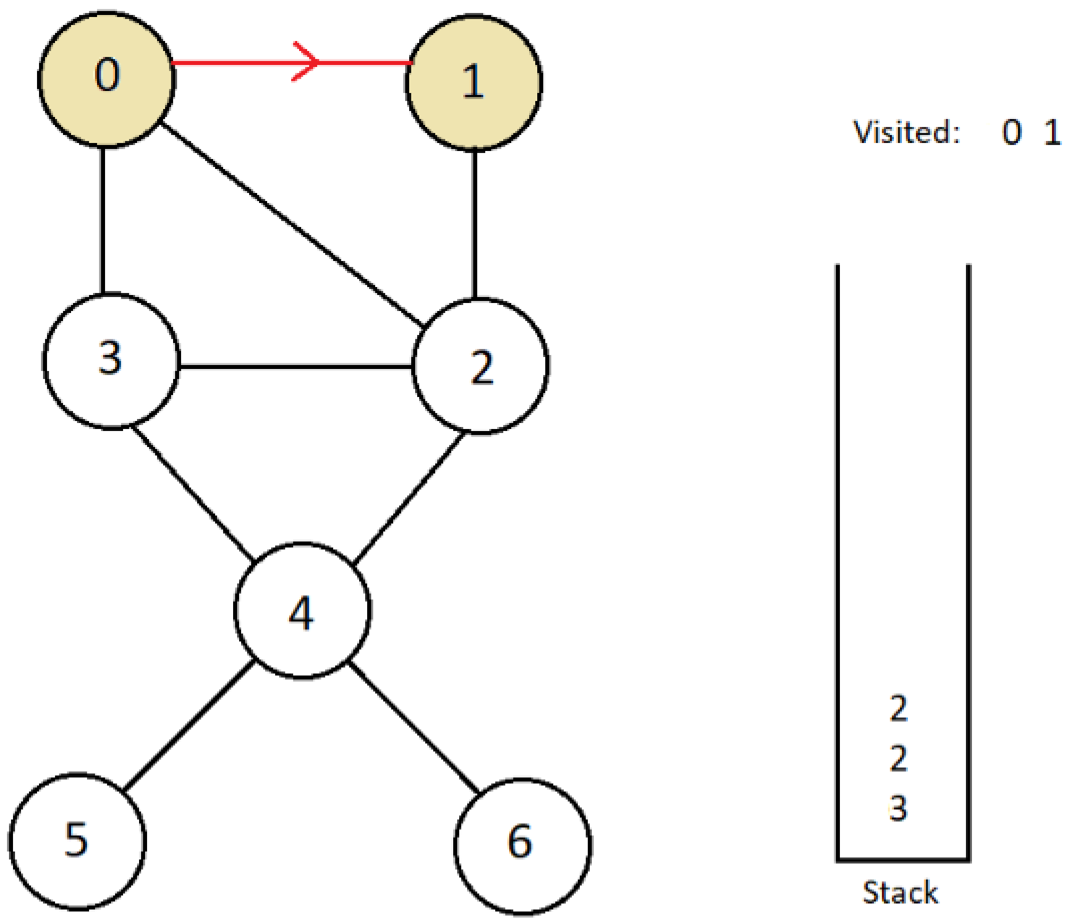
Considering the fact that we could begin traversing with any source node; we'll start with 0 only. So, following step1, we would push this node into the stack and begin our Depth First Search traversal.



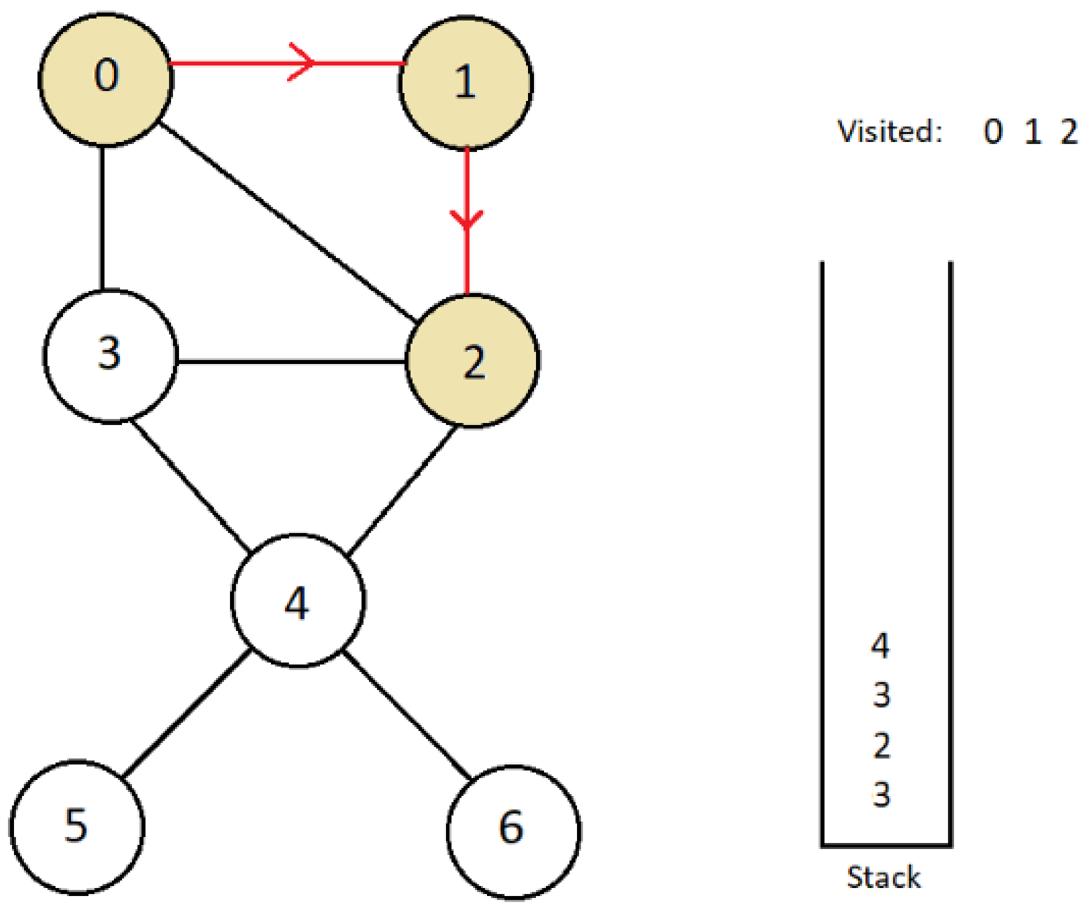
The next step is, pop the top element from the stack which is node 0 here, and mark it visited. Then, we'll start visiting all the nodes connected to node 0 which are not visited already, we are asked to push them all into the stack, and the order in which you push doesn't matter at all. Therefore, we will push nodes 3, 2, and 1 into the stack.



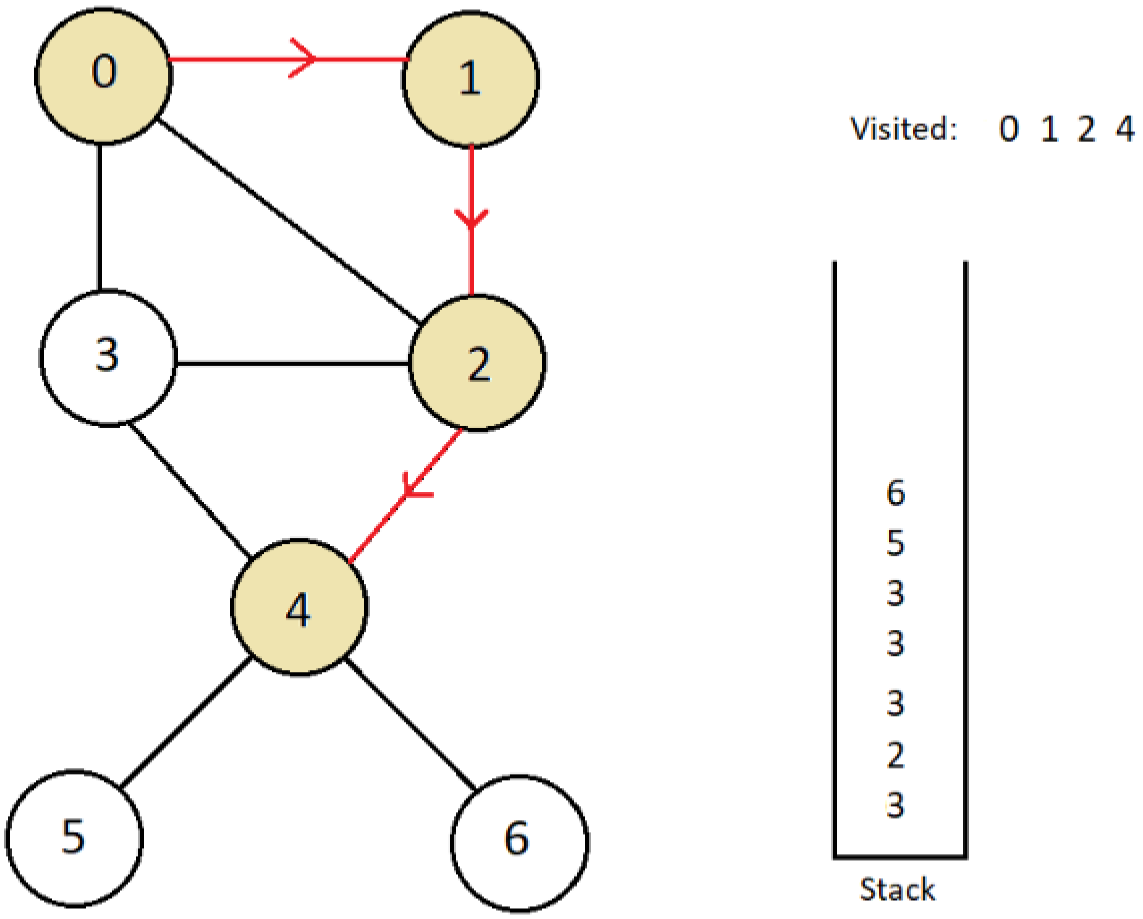
Repeating the steps above, we'll now pop the top element from the stack which is node 1, and mark it visited. Only nodes connected to node 1 were nodes 0 and 2, and since the only unvisited one is node 2. Although node 2 is in the stack, it is not visited. So, we'll push it into the stack again.



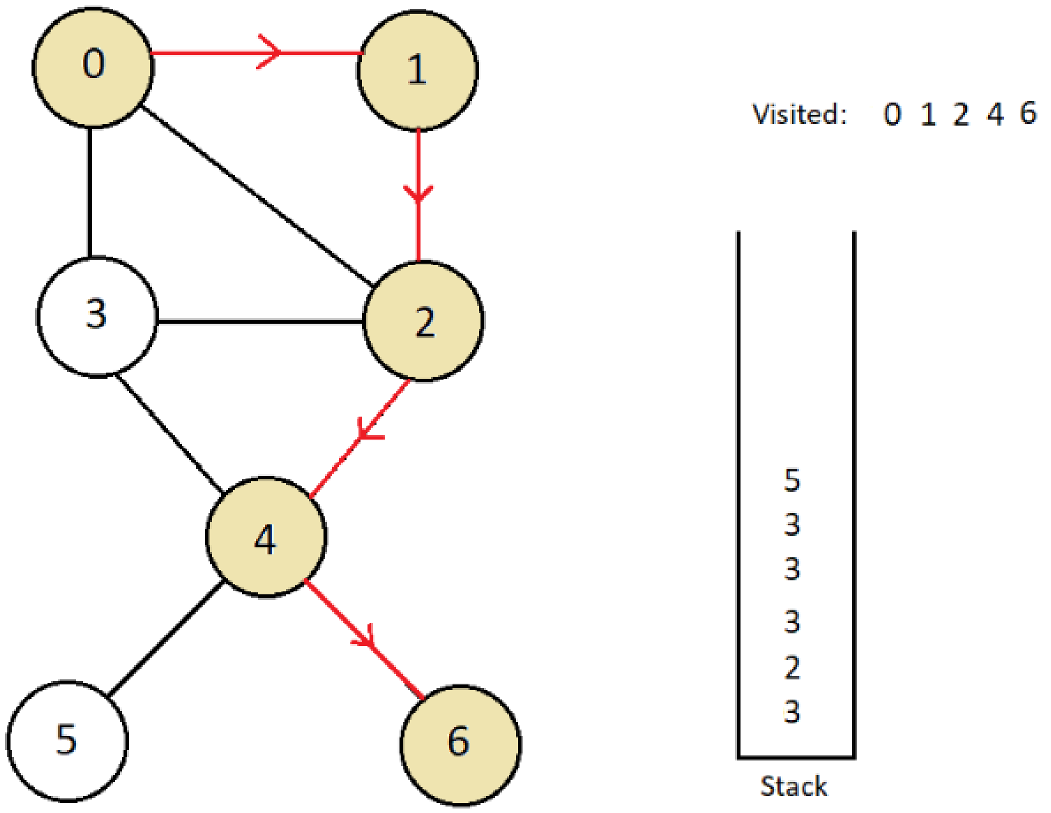
Next, we have node 2 at the top of the stack. We'll mark node 2 visited and unvisited nodes connected to node 2 are nodes 3 and 4, regardless of the fact that 3 is already there in the stack. So, we'll just push nodes 3 and 4 into the stack.



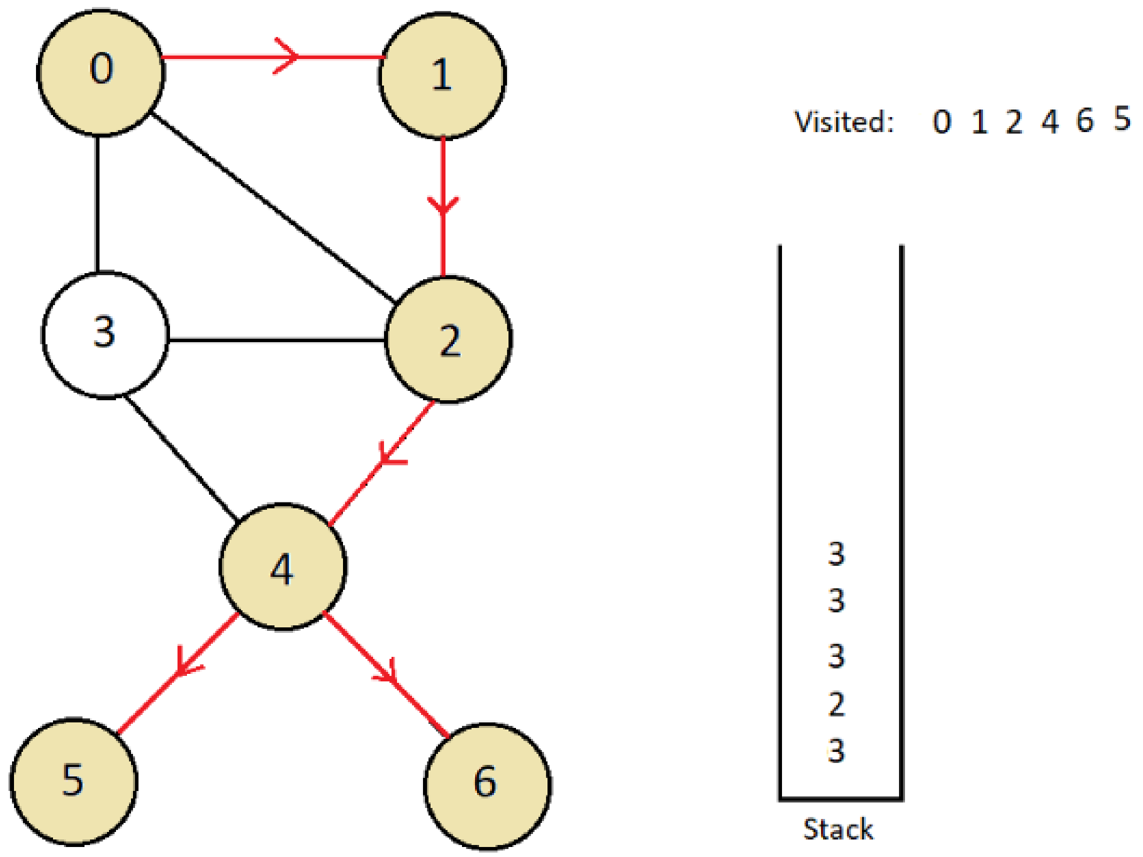
Node 4 is the next we have on the top. So, just mark it as visited. Since, all nodes 3, 5, and 6, except node 2, which are directly connected to it are not visited, we'll push them into the stack.



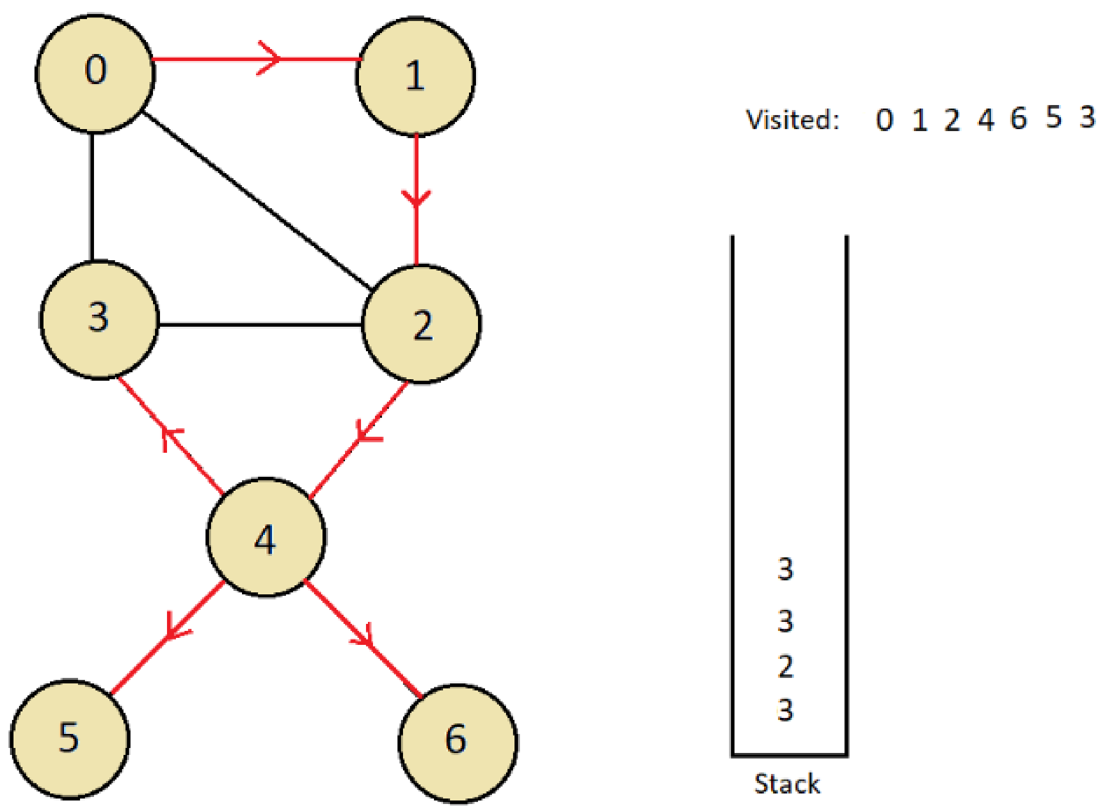
Next, we have node 6 on the top of the stack. Pop it and mark it visited. Since there are no nodes that are directed connected to node 6 and unvisited, we'll continue further without doing anything.



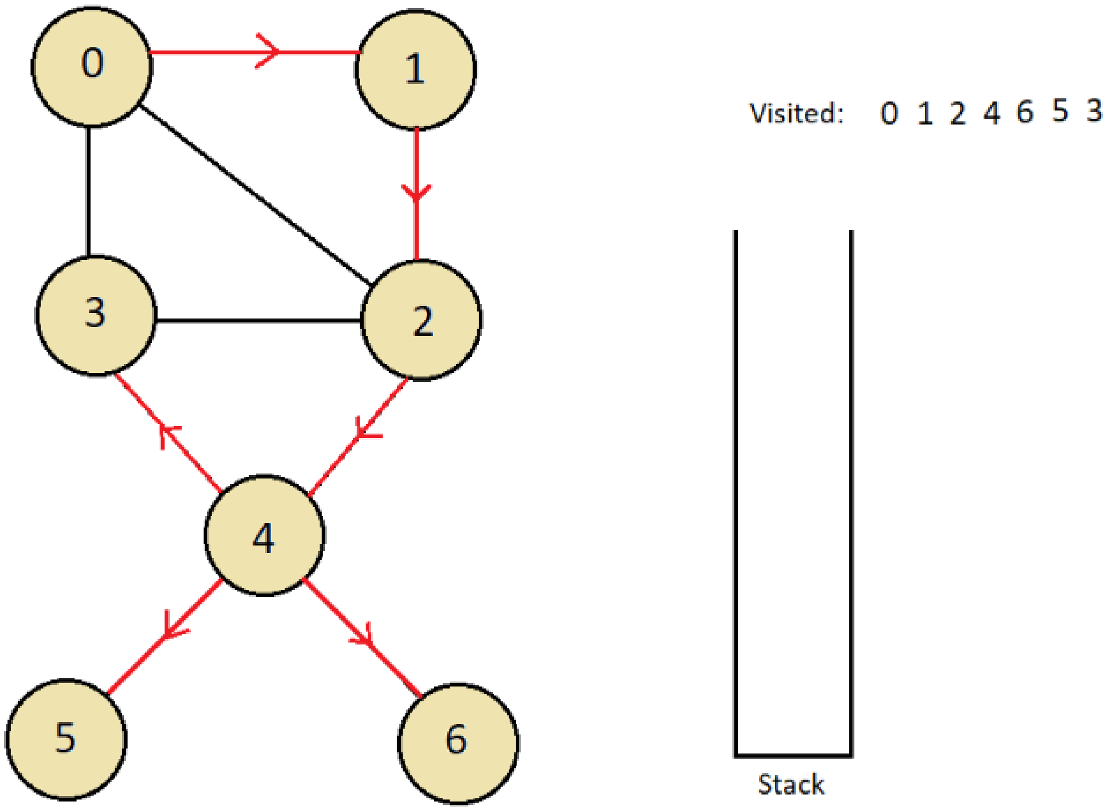
Next, we pop node 5 out of the stack and mark it visited. And since there is no unvisited node connected to it.



Node 3 comes next to be visited, being on the top in the stack. Mark node 3 visited and again there are no nodes left unvisited and connected to node 3. So, we just continue popping out elements from the stack.



Now, if you could observe, there are no nodes left to be visited. Although there are elements in the stack to be explored. So, we just pop them one by one and ignore finding them already visited. And this gets our stack emptied and every node traversed in Depth First Search manner.



And the order in which we marked our nodes visited is the Depth First Search traversal order. Here, it is **0, 1, 2, 4, 6, 5, 3.** So basically, the visited array maintains whether the node itself is visited or not, and the stack maintains nodes whose exploration got suspended earlier

ALGORITHM:

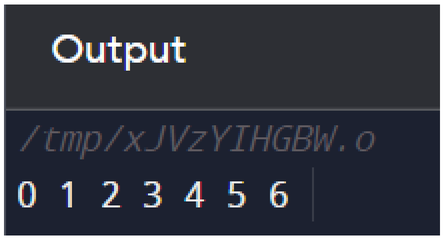
1)Now, the first thing we did was, we took the input. The input comprises the information concerning the graph, its nodes/vertices, and edges, and the source node we'll start the traversal with.
2)Then, we'll call the function DFS feeding a source node into it.
3)In the function, we'll mark the node visited, and then initiate a for loop which will access the connected nodes to this node, and see if they are visited or not.
4)And every time we find a node connected and not visited already, we call the DFS function recursively filling out our function stack. Eventually, the

function stack becomes empty, and our traversal ends. And the visited array is the order of our Depth First Search traversal.

CODE FOR DFS:

```
#include<stdio.h>
#include<stdlib.h>

int visited[7] = {0,0,0,0,0,0,0};
    int A [7][7] = {
        {0,1,1,1,0,0,0},
        {1,0,1,0,0,0,0},
        {1,1,0,1,1,0,0},
        {1,0,1,0,1,0,0},
        {0,0,1,1,0,1,1},
        {0,0,0,0,1,0,0},
        {0,0,0,0,1,0,0}
    };

void DFS(int i){
    printf("%d ", i);
    visited[i] = 1;
    for (int j = 0; j < 7; j++)
    {
        if(A[i][j]==1 && !visited[j]){
            DFS(j);
        }
    }
}

int main(){
    // DFS Implementation
    DFS(0);
    return 0;
}
```

OUTPUT:



Output

```
/tmp/xJVzYIHGBW.o
0 1 2 3 4 5 6
```