# Jadavpur University

Department of Electronics and Telecommunication Engineering,
Faculty of Engineering & Technology

DSA LAB REPORT
2nd Year Second Semester 2020

Name : RAHUL SAHA
Roll: 001910701009

Group 1

## IMPLEMENTATION OF VARIOUS GRAPH TRAVERSAL SCHEMES

Date Of Submission: 19/05/2021

# BREADTH FIRST TRAVERSAL

## Q. Given a graph G= (V, E) and a source vertex s, implement Breadth-First-Search (BFS) and grow a BFS tree with root as s. Show some intermediate outputs as well.

### Introduction:

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree . The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

### Principle

The algorithm discovers all vertices at distance k from source vertex s before discovering any vertices at distance k+1.

### Algorithm for BFS:

```
1. SET STATUS = 1 (ready state) for each node in G
2. Enqueue the starting node A and set its STATUS = 2 (waiting state)
3. Repeat Steps 4 and 5 until QUEUE is empty
4. Dequeue a node N. Process it and set its STATUS = 3 (processed state).
5. Enqueue all the neighbours of N that are in the ready state (whose STATUS =
1) and set their STATUS = 2 (waiting state) [END OF LOOP]
6. EXIT
```

### Source Code:

```c
#include <stdio.h>
#include <stdlib.h>
#define SIZE 40
#define MAX 100
int array[MAX];

typedef struct n_ary_tree{
    int data;
    struct n_ary_tree *firstChild;
    struct n_ary_tree *nextSibling;
}tree;

typedef struct queue_list {
    int items[SIZE];
```

```
    int front;
    int rear;
}queue;

typedef struct adjacency_list_node
{
    int dest;
    struct adjacency_list_node* next;
}node;

typedef struct Graph_Structure
{
    int Vertices;
    node** adj_list;
    int* visited;
}Graph;

node* create_node(int dest)
{
    node* newNode = (node*) malloc(sizeof(node));
    newNode->dest = dest;
    newNode->next = NULL;
    return newNode;
}



Graph* create_graph(int vertices) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->Vertices = vertices;

    graph->adj_list = (node**)malloc(vertices * sizeof(node*));
    graph->visited = (int*)malloc(vertices * sizeof(int));

    int i;
    for (i = 0; i < vertices; i++) {
        graph->adj_list[i]= NULL;
        graph->visited[i] = 0;
    }
    return graph;
}

void add_edge(Graph* graph, int src, int dest)
{
    node* newNode = create_node(dest);
```

```
    newNode->next = graph->adj_list[src];
    graph-> adj_list[src]= newNode;

    newNode = create_node(src);
    newNode->next = graph->adj_list[dest];
    graph->adj_list[dest]= newNode;
}

//Function to create a queue
queue* create_queue() {
    queue* q = (queue*)malloc(sizeof(queue));
    q->front = -1;
    q->rear = -1;
    return q;
}

// Check if the queue is empty
int isEmpty(queue* q) {
    if (q->rear == -1)
        return 1;
    else
        return 0;
}

// Adding elements into queue
void enqueue(queue* q, int value) {
    if (q->rear == SIZE - 1)
        printf("\nQueue is Overflow");
    else {
        if (q->front == -1)
        q->front = 0;
        q->rear++;
        q->items[q->rear] = value;
    }
}

// Removing elements from queue
int dequeue(queue* q) {
    int item;
    if (isEmpty(q)) {
        printf("\nQueue is empty\n");
        item = -1;
    } else {
        item = q->items[q->front];
        printf("\nThe Value Dequeued is:%d",item);
```

```
        q->front++;
        if (q->front > q->rear) {
        printf("\nResetting queue");
        q->front = q->rear = -1;
        }
    }
    return item;
}

// Print the queue
void print_queue(queue* q) {
    int i = q->front;

    if (isEmpty(q)) {
        printf("\nQueue is empty\n");
    } else {
        printf("\nQueue contains: ");
        for (i = q->front; i < q->rear + 1; i++) {
        printf("%d ", q->items[i]);
        }
    }
}

void bfs(Graph* graph, int start_vertex) {
    queue* q = create_queue();
    graph->visited[start_vertex] = 1;
    enqueue(q, start_vertex);
    int i=0;
    while (!isEmpty(q)) {
        print_queue(q);
        int current_vertex = dequeue(q);
        printf("\nVisited %d\n\n", current_vertex);
        array[i]=current_vertex;
        i++;
        node* temp = graph->adj_list[current_vertex];

        while (temp) {
        int adj_vertex = temp->dest;
        if(graph->visited[adj_vertex]==1){
            printf("\nVertex %d which is adjacent to %d is already
visited",adj_vertex,current_vertex);
        }
        else if(graph->visited[adj_vertex] == 0) {
            graph->visited[adj_vertex] = 1;
            printf("\nThe Value %d is a child to the node
```

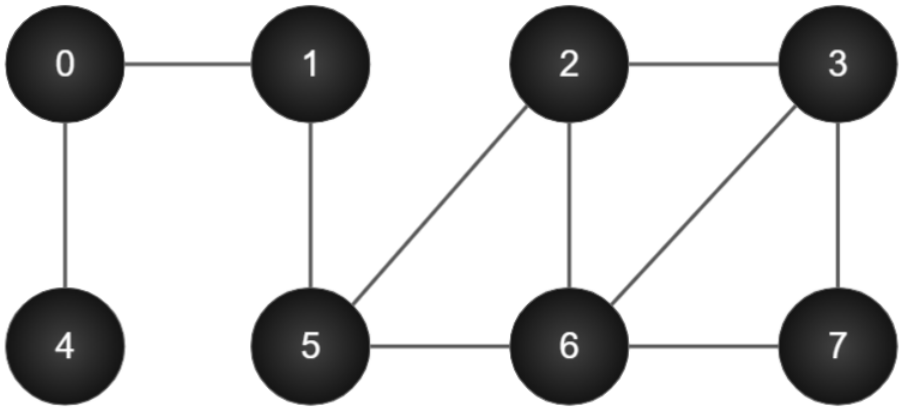
```
%d",adj_vertex,current_vertex);
            enqueue(q, adj_vertex);
        }
        temp = temp->next;
        }
    }
}

//Fuction to print the graph
void print_graph(Graph* graph) {
  int v;
  for (v = 0; v < graph->Vertices; v++) {
    node* temp = graph->adj_list[v];
    printf("\n Adjacency list of vertex %d\n ", v);
    while (temp) {
      printf("%d -> ", temp->dest);
      temp = temp->next;
    }
    printf("\n");
  }
}

//driver code
int main() {
    system("cls");
    printf("\t\tBreadth First Search in Graph");
    int vertices,ver1,ver2;
    char ch;
    printf("\n\nEnter the number of Vertices: ");
    scanf("%d",&vertices);
    Graph* graph = create_graph(vertices);
    do{
        printf("\nEnter the vertices you want to enter an edge in
between: ");
        scanf("%d %d",&ver1,&ver2);
        add_edge(graph,ver1,ver2);
        printf("Do you want to insert more edges->Y to continue and N
to Exit:");
        fflush(stdin);
        scanf("%c",&ch);
    }while(ch=='Y'||ch=='y');
    printf("\nDisplaying the Graph in adjacency List Format\n");
    print_graph(graph);
    printf("\n\n");
    bfs(graph, 0);
```

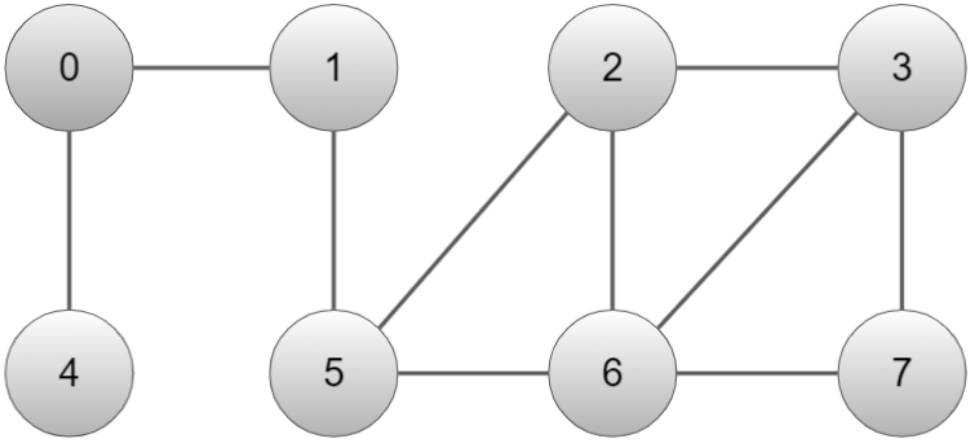
```
    printf("\n\nThe Breadth first search for the Given Graph is:\n");
    for(int i=0;i<vertices;i++){
        printf("%d ",array[i]);
    }
    printf("\n\n");
    return 0;
}
```

*Let us take this example:*

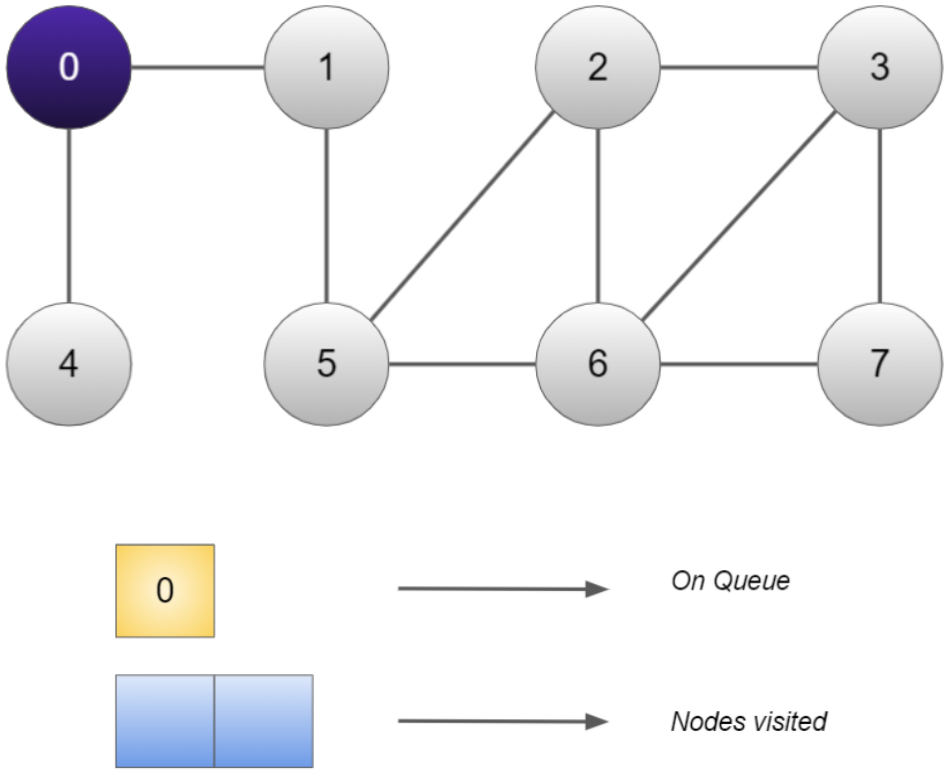



Let,
White nodes represent all the nodes that are not visited yet.
Purple nodes represent the nodes whose adjacency list is being checked and scanned
Red nodes represent the nodes which have been traversed.

Initially all the nodes are white

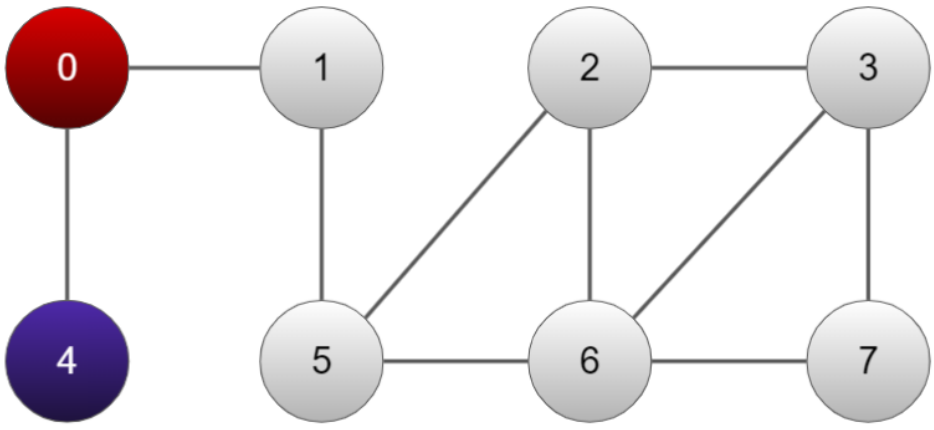


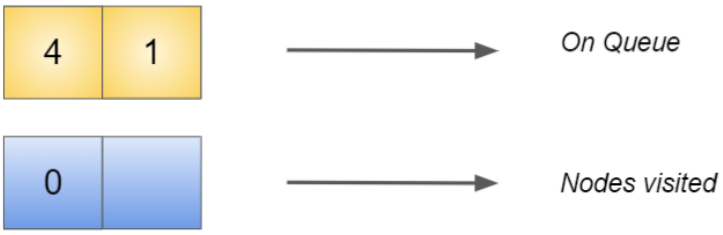
First, vertex 0 is scanned and its adjacency list is checked




Adj[0] → 4,1

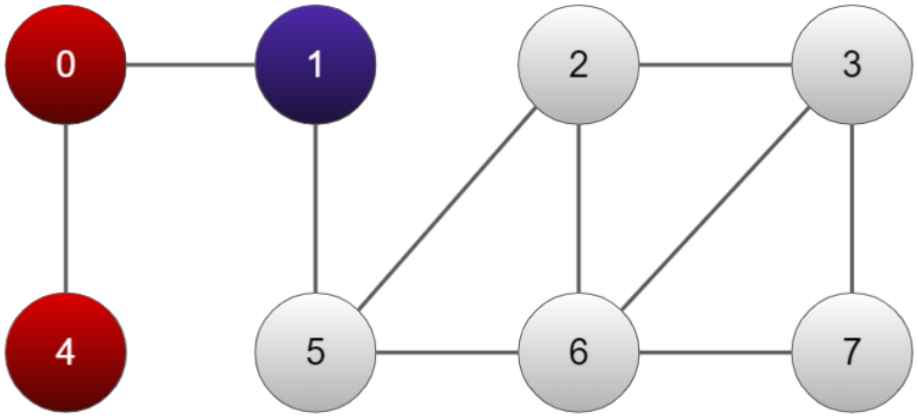
Now, 4 being the adjacent node of 0 is scanned and node 0 is completely explored.

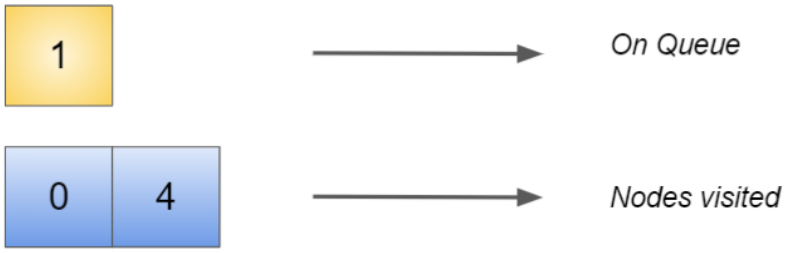





Adj[4] → Φ

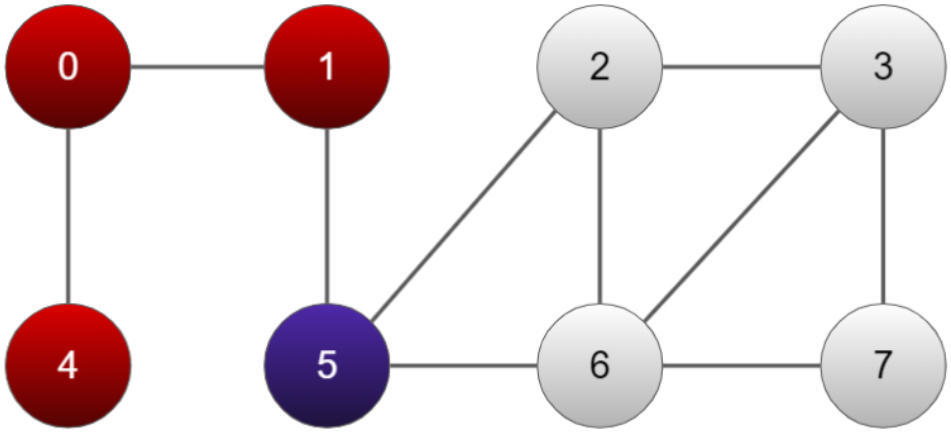
Next, 1 being the adjacent node of 0 is scanned and node 4 is visited completely.

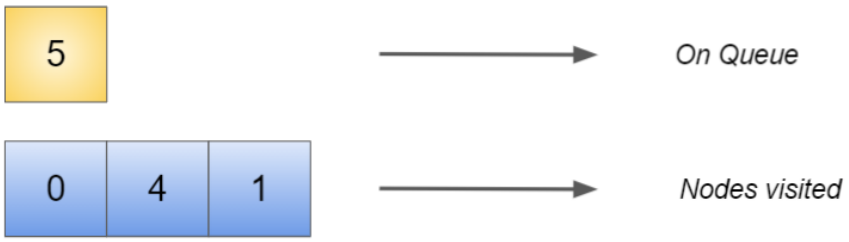



Adj[1] → 5

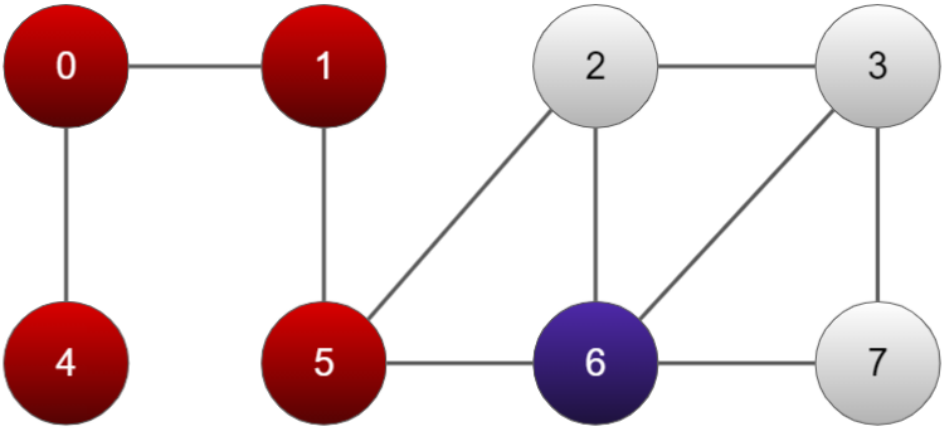
Next, 5 being the adjacent node of 1 is scanned and node 1 is visited completely.

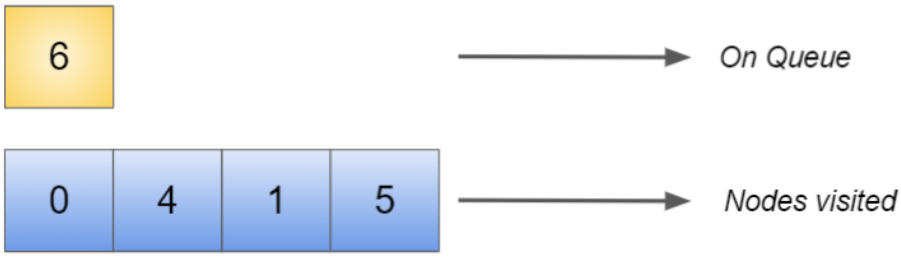





Adj[5] → 6,2

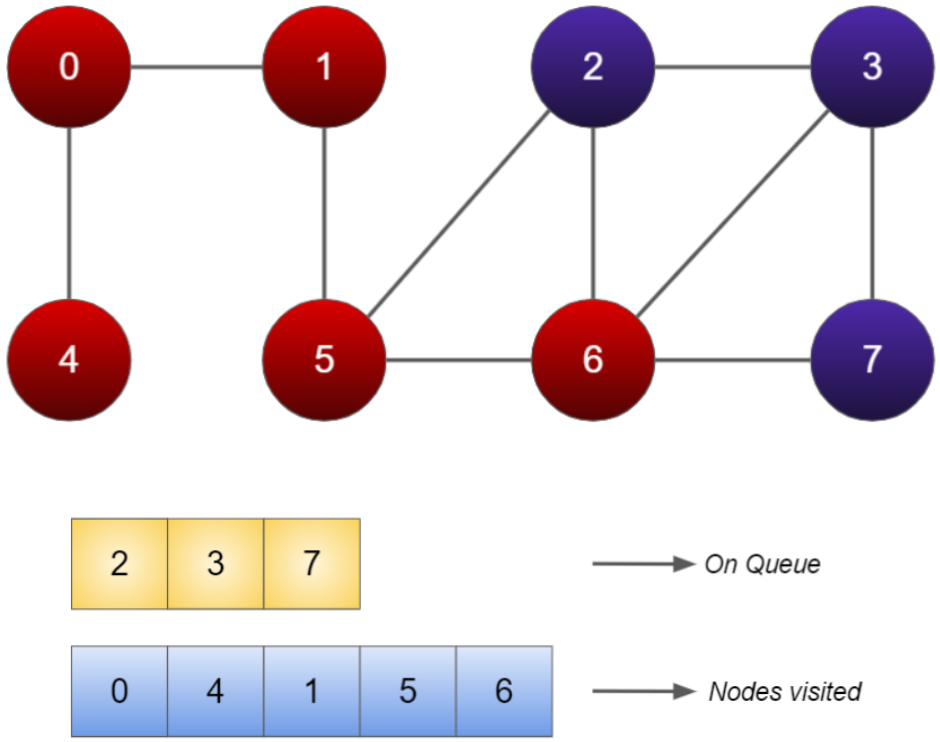
Next, 6 being the adjacent node of 5 is scanned and node 5 is visited completely.



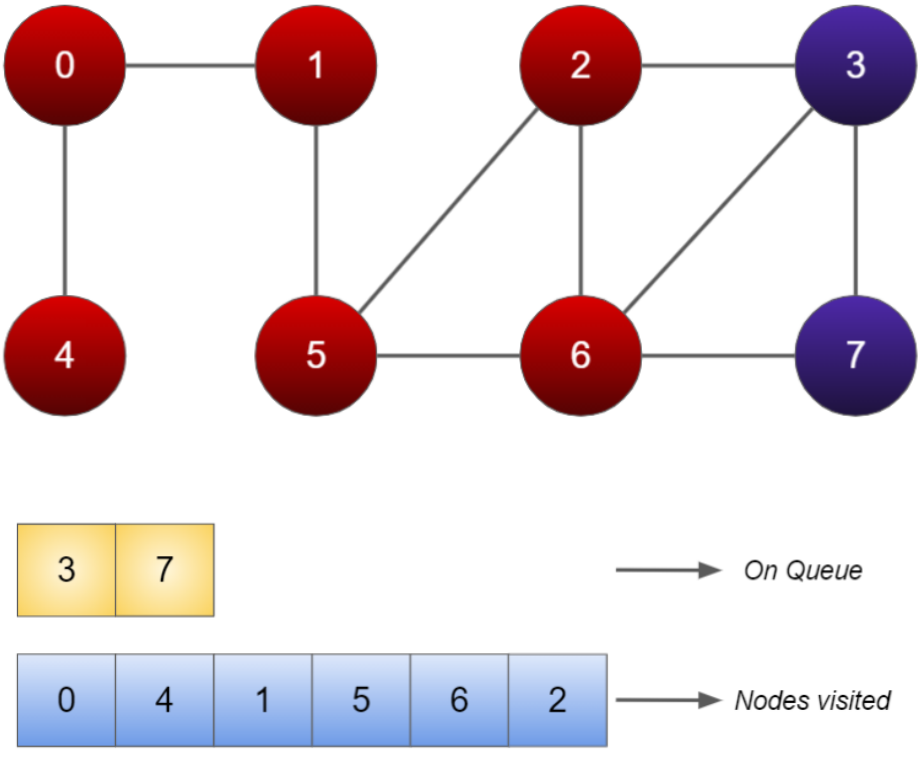

Adj[6] → 2,3,7

Next, 2,3,7 being the adjacent nodes of 6 is scanned one by one
First, 2 is scanned

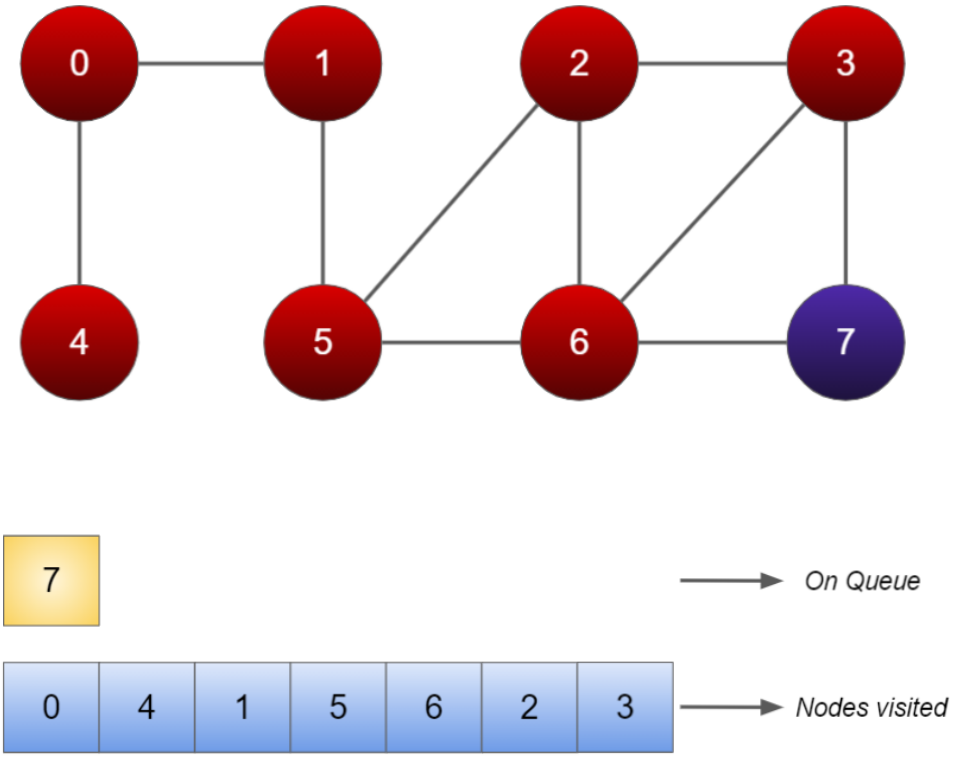



Adj[2] → 3

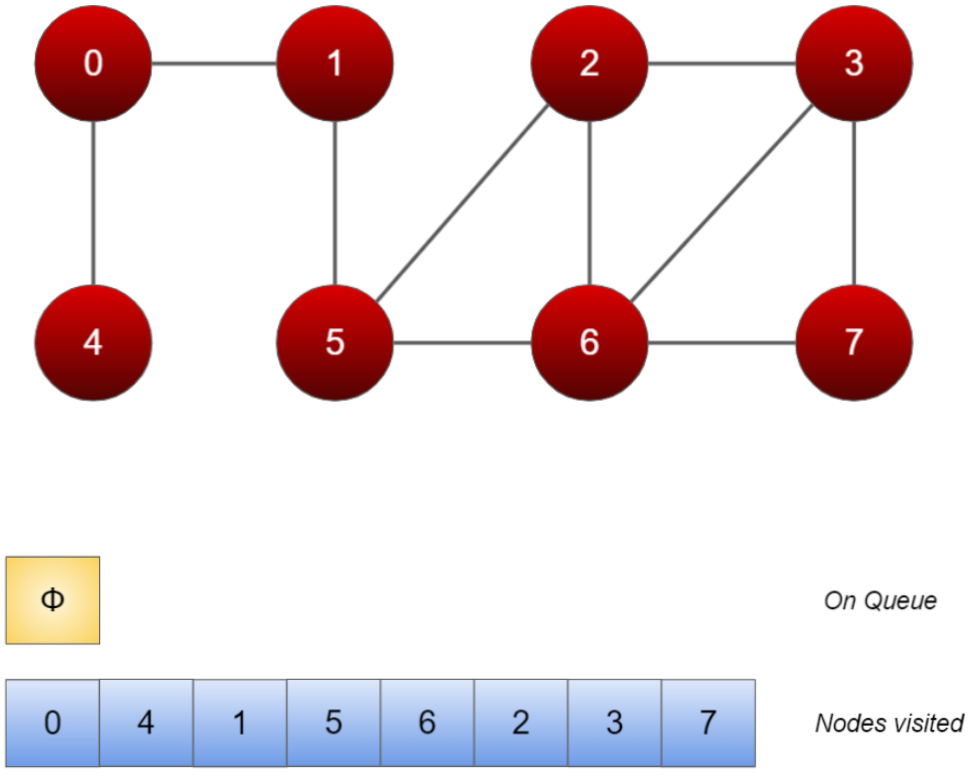
Now, 3 is scanned



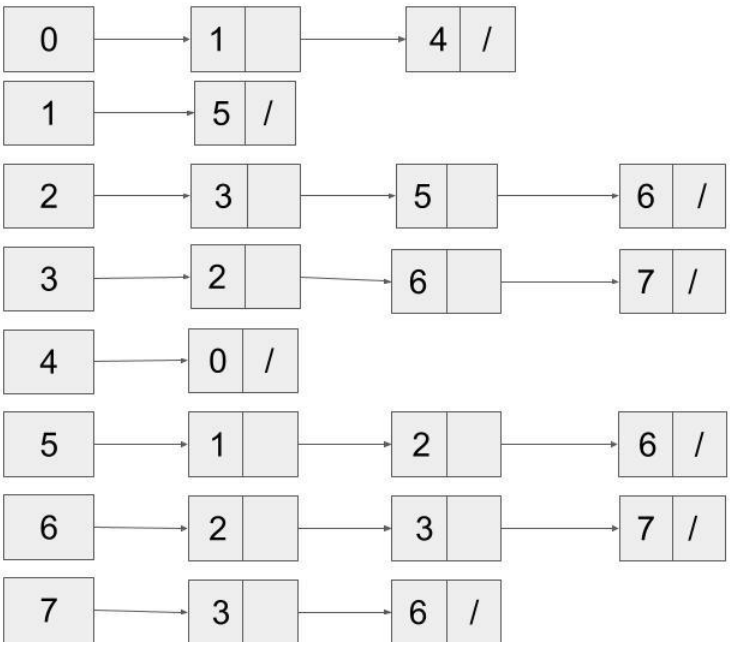
Now 7 is scanned




At last, 7 is traversed



Here is the adjacency list of all the nodes



## OUTPUT CONSOLE

### Vertices and edges are being inserted

```
                Breadth First Search in Graph

Enter the number of Vertices: 8

Enter the vertices you want to enter an edge in between: 0 1
Do you want to insert more edges->Y to continue and N to Exit:Y

Enter the vertices you want to enter an edge in between: 0 4
Do you want to insert more edges->Y to continue and N to Exit:Y

Enter the vertices you want to enter an edge in between: 1 5
Do you want to insert more edges->Y to continue and N to Exit:Y

Enter the vertices you want to enter an edge in between: 5 2
Do you want to insert more edges->Y to continue and N to Exit:Y

Enter the vertices you want to enter an edge in between: 5 6
Do you want to insert more edges->Y to continue and N to Exit:Y

Enter the vertices you want to enter an edge in between: 6 2
Do you want to insert more edges->Y to continue and N to Exit:Y

Enter the vertices you want to enter an edge in between: 6 3
Do you want to insert more edges->Y to continue and N to Exit:Y

Enter the vertices you want to enter an edge in between: 6 7
Do you want to insert more edges->Y to continue and N to Exit:Y

Enter the vertices you want to enter an edge in between: 6 3
Do you want to insert more edges->Y to continue and N to Exit:Y

Enter the vertices you want to enter an edge in between: 2 3
Do you want to insert more edges->Y to continue and N to Exit:Y

Enter the vertices you want to enter an edge in between: 3 7
Do you want to insert more edges->Y to continue and N to Exit:N
```

**The adjacency list**

```
Displaying the Graph in adjacency List Format

 Adjacency list of vertex 0
4 -> 1 ->

 Adjacency list of vertex 1
5 -> 0 ->

 Adjacency list of vertex 2
3 -> 6 -> 5 ->

 Adjacency list of vertex 3
7 -> 2 -> 6 -> 6 ->

 Adjacency list of vertex 4
0 ->

 Adjacency list of vertex 5
6 -> 2 -> 1 ->

 Adjacency list of vertex 6
3 -> 7 -> 3 -> 2 -> 5 ->

 Adjacency list of vertex 7
3 -> 6 ->
```

**Graph traversal (BFS)**

```
Queue contains: 0
The Value Dequeued is:0
Resetting queue
Visited 0


The Value 4 is a child to the node 0
The Value 1 is a child to the node 0
Queue contains: 4 1
The Value Dequeued is:4
Visited 4


Vertex 0 which is adjacent to 4 is already visited
Queue contains: 1
The Value Dequeued is:1
Resetting queue
Visited 1


The Value 5 is a child to the node 1
Vertex 0 which is adjacent to 1 is already visited
Queue contains: 5
The Value Dequeued is:5
Resetting queue
Visited 5


The Value 6 is a child to the node 5
The Value 2 is a child to the node 5
Vertex 1 which is adjacent to 5 is already visited
Queue contains: 6 2
The Value Dequeued is:6
Visited 6


The Value 3 is a child to the node 6
The Value 7 is a child to the node 6
Vertex 3 which is adjacent to 6 is already visited
Vertex 2 which is adjacent to 6 is already visited
Vertex 5 which is adjacent to 6 is already visited
Queue contains: 2 3 7
The Value Dequeued is:2
Visited 2
```

```
Vertex 3 which is adjacent to 2 is already visited
Vertex 6 which is adjacent to 2 is already visited
Vertex 5 which is adjacent to 2 is already visited
Queue contains: 3 7
The Value Dequeued is:3
Visited 3


Vertex 7 which is adjacent to 3 is already visited
Vertex 2 which is adjacent to 3 is already visited
Vertex 6 which is adjacent to 3 is already visited
Vertex 6 which is adjacent to 3 is already visited
Queue contains: 7
The Value Dequeued is:7
Resetting queue
Visited 7


Vertex 3 which is adjacent to 7 is already visited
Vertex 6 which is adjacent to 7 is already visited
```

**The Breadth First Traversal**

```
The Breadth first search for the Given Graph is:
0 4 1 5 6 2 3 7
```

# Depth First Traversal

**Q. Given a directed graph G=(V, E), implement Depth-First-Search (DFS). Your program should invoke DFS_VISIT(u) for any vertex u. Show some outputs as well.**

## Introduction

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, a node may be visited twice. To avoid processing a node more than once, use a boolean visited array.

## Principle

Edges are explored out of the most recently discovered vertex v, that still has unexplored edges having it. When all of v's edges have been explored, the search backtracks to explore edges having the vertex from which v was discovered. The process continues until we have discovered all the vertices that are reachable from the original source vertex.

**Algorithm for DFS:**

```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting
state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)
Step 5: Push on the stack all the neighbours of N that are in the ready state
(whose STATUS = 1) and set their
STATUS = 2 (waiting state)
[END OF LOOP]
Step 6: EXIT
```

**Source Code:**

```
// DFS algorithm in C

#include <stdio.h>
#include <stdlib.h>
#define MAX 100
int stack[MAX];
int top=-1;

typedef struct adjacency_list_node
{
    int dest;
    struct adjacency_list_node* next;
}node;

typedef struct Graph_Structure
{
    int Vertices;
    node** adj_list;
    int* visited;
}Graph;

void push(int item) {

    stack[++top] = item;
}

int pop() {
    printf("\nThe value poped is: %d",stack[top]);
    return stack[top--];
}

void print_stack(){
    printf("\nThe Stack contains: ");
    for(int k=top;k>=0;k--){
        printf("%d ",stack[k]);
    }
}

int isEmpty() {
   if (top == -1)
        return 1;
    else
        return 0;
```

```
}

// Create a node
node* create_node(int dest)
{
    node* newNode = (node*) malloc(sizeof(node));
    newNode->dest = dest;
    newNode->next = NULL;
    return newNode;
}

// Create graph
Graph* create_graph(int vertices) {
    Graph* graph =(Graph*)malloc(sizeof(Graph));
    graph->Vertices = vertices;

    graph->adj_list =(node**)malloc(vertices * sizeof(node*));

    graph->visited =(int*)malloc(vertices * sizeof(int));

    int i;
    for (i = 0; i < vertices; i++) {
        graph->adj_list[i] = NULL;
        graph->visited[i] = 0;
    }
    return graph;
}

// Add edge
void add_edge(Graph* graph, int src, int dest) {
    // Add edge from src to dest
    node* newNode = create_node(dest);
    newNode->next = graph->adj_list[src];
    graph->adj_list[src] = newNode;

    // Add edge from dest to src
    newNode = create_node(src);
    newNode->next = graph->adj_list[dest];
    graph->adj_list[dest] = newNode;
}

// DFS algo
void DFS_visit(Graph* graph, int start_vertex) {
    graph->visited[start_vertex] = 1;
    push(start_vertex);
```

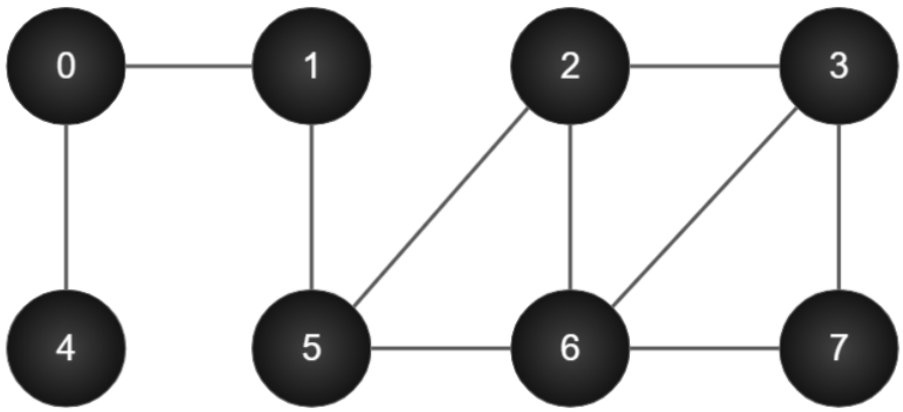
```
    int i=0;
    while (!isEmpty()) {
        print_stack();
        int current_vertex = pop();
        printf("\nVisited %d\n\n", current_vertex);
        node* temp = graph->adj_list[current_vertex];
        while (temp) {
        int adj_vertex = temp->dest;
        if(graph->visited[adj_vertex]==1){
            printf("\nVertex %d which is adjacent to %d is already
visited",adj_vertex,current_vertex);
        }
        else if(graph->visited[adj_vertex] == 0) {
            graph->visited[adj_vertex] = 1;
            printf("\nThe unvisited vertex %d is adjacent to
%d",adj_vertex,current_vertex);
            push(adj_vertex);
        }
        temp = temp->next;
        }
    }
}

//Function to print the graph
void print_graph(Graph* graph) {
    int v;
    for (v = 0; v < graph->Vertices; v++) {
        node* temp = graph->adj_list[v];
        printf("\n Adjacency list of vertex %d\n ", v);
        while (temp) {
        printf("%d -> ", temp->dest);
        temp = temp->next;
        }
        printf("\n");
    }
}

//driver function
int main() {
    system("cls");
    printf("\t\tDepth First Search in Graph");
    int vertices,ver1,ver2,vertex;
    char ch;
    printf("\n\nEnter the number of Vertices: ");
    scanf("%d",&vertices);
```

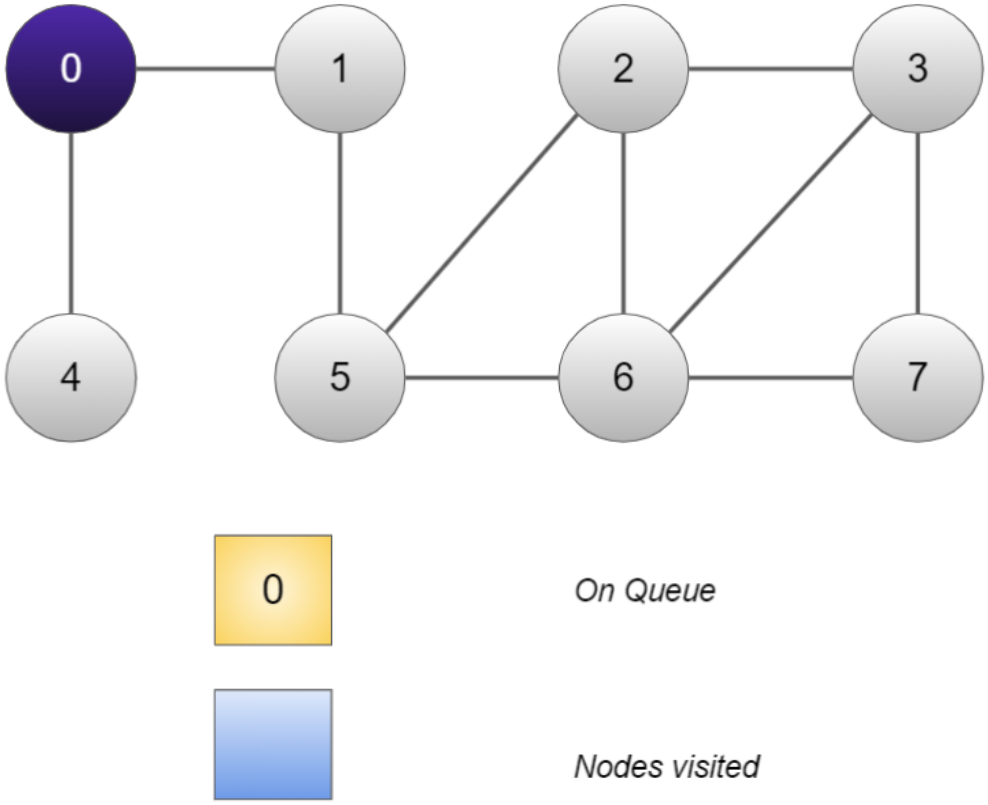
```
    Graph* graph = create_graph(vertices);
    do{
        printf("\nEnter the vertices you want to enter an edge in
between: ");
        scanf("%d %d",&ver1,&ver2);
        add_edge(graph,ver1,ver2);
        printf("Do you want to insert more edges->Y to continue and N
to Exit:");
        fflush(stdin);
        scanf("%c",&ch);
    }while(ch=='Y'||ch=='y');
    printf("\nDisplaying the Graph in adjacency List Format\n");
    print_graph(graph);
    printf("\n\n");
    printf("Enter the vertex you want to search the dfs of: ");
    scanf("%d",&vertex);
    printf("\n");
    DFS_visit(graph, vertex);
    printf("\n\n");
    return 0;
}
```

*Taking the same example as in the previous case*



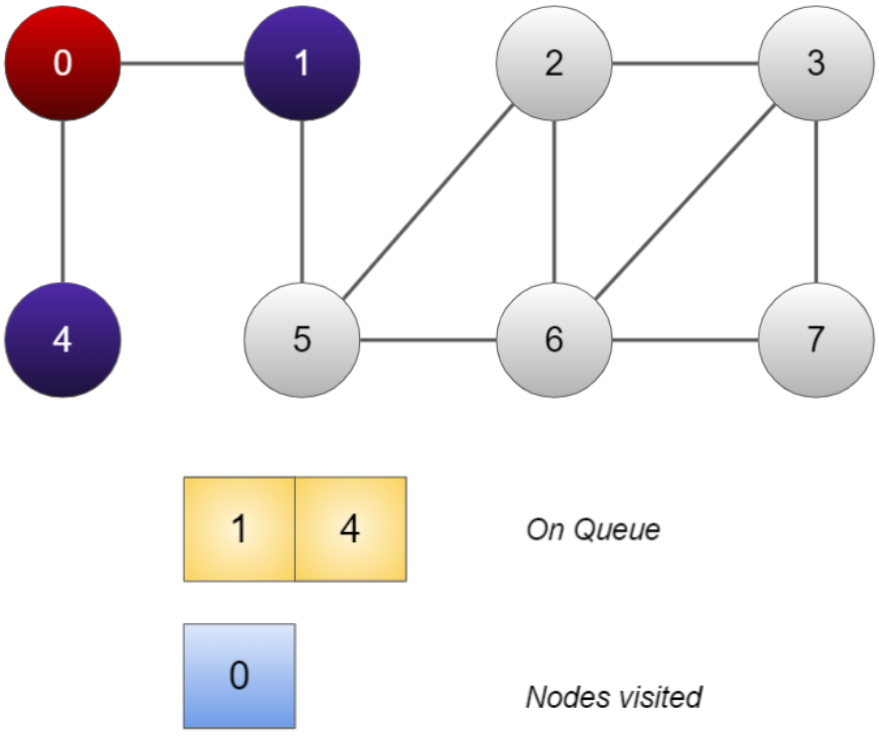

Let,
White nodes represent all the nodes that are not visited yet.
Purple nodes represent the nodes whose adjacency list is being checked and scanned
Red nodes represent the nodes which have been traversed.

Suppose we want to start from node 0



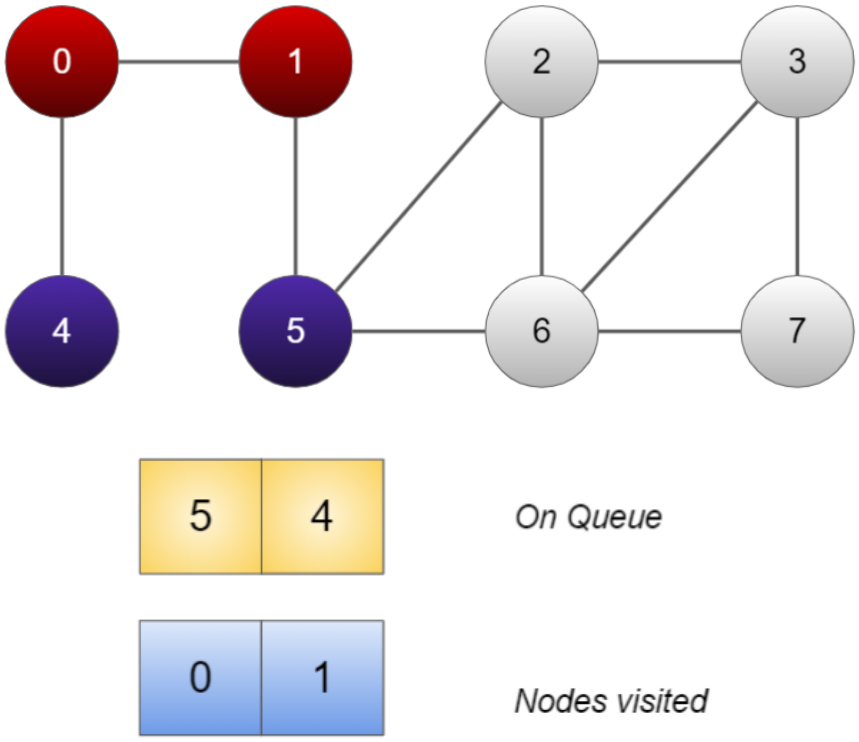

Adj[0] → 1,4

Traversing adjacent node of 0, i.e. 1



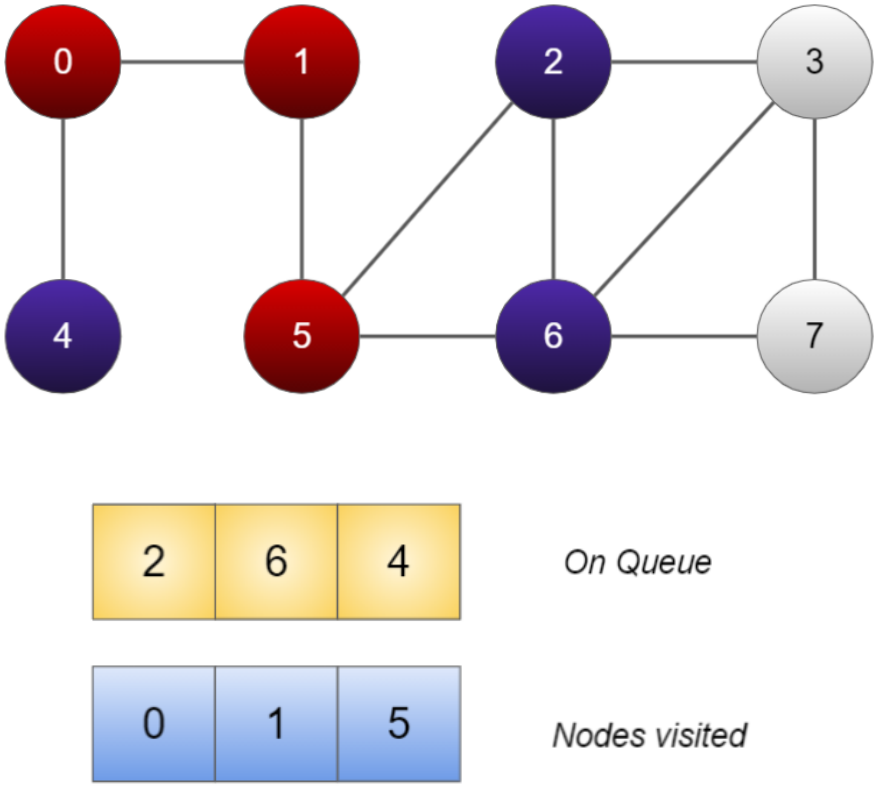

Adj[1] → 5

Traversing through 5




Adj[5] → 2,6

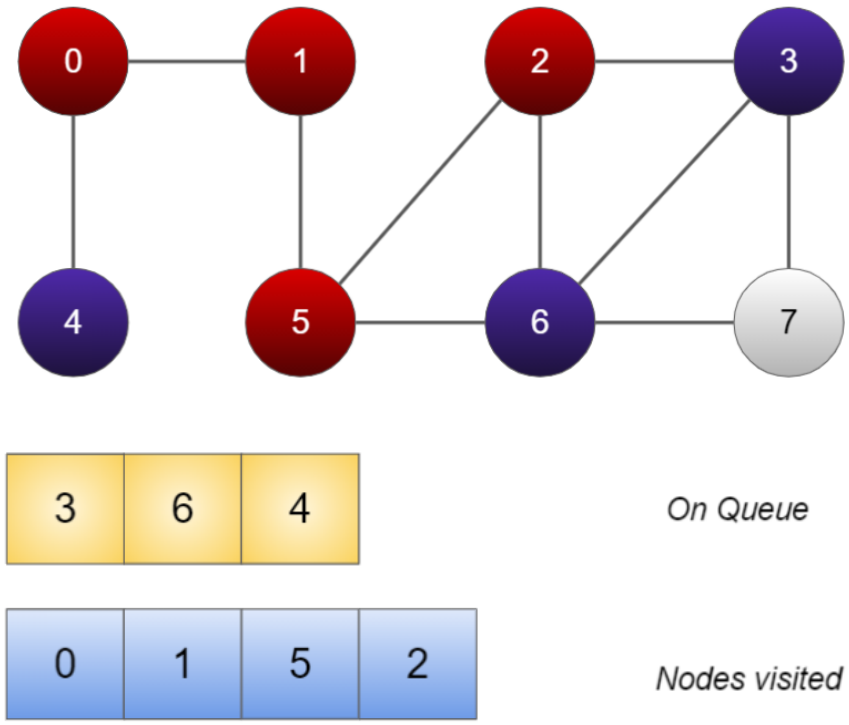
Traversing through 2 now




Adj[2] → 3,6,5

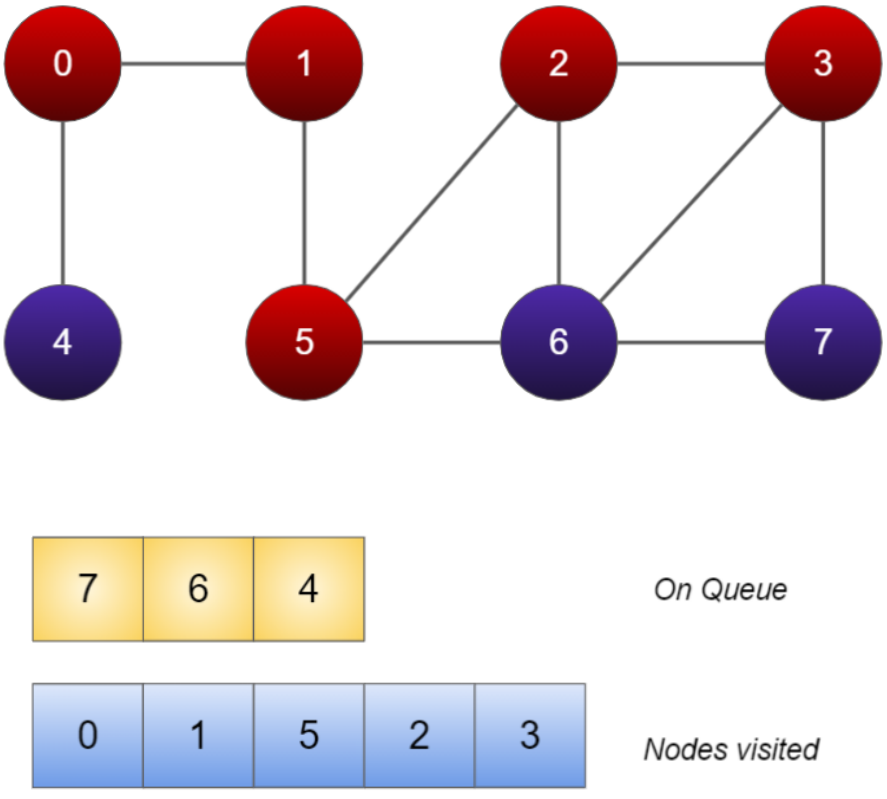
Out of which, the nodes 5 and 6 are already visited

Traversing node 3



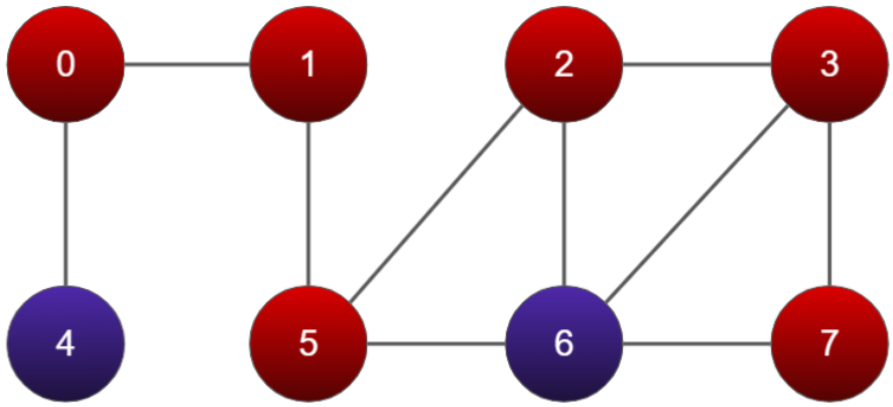

Adj[3] → 2,6,7
Out of which, nodes 2 and 6 are already visited

Traversing through node 7



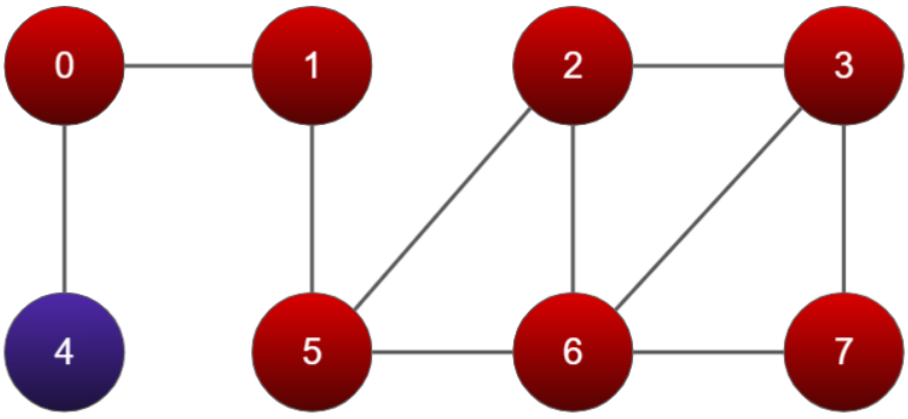
Adj[7] → 3,6
Out of which, node 3 is already visited

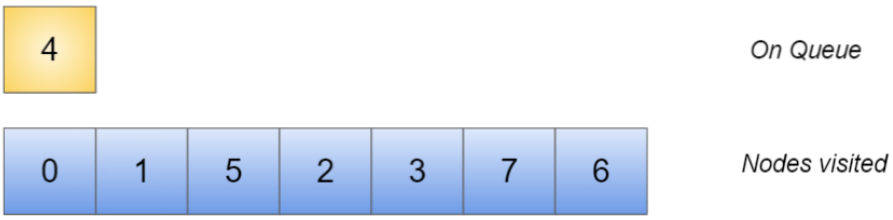
Traversing through 6 which is immediate on the queue



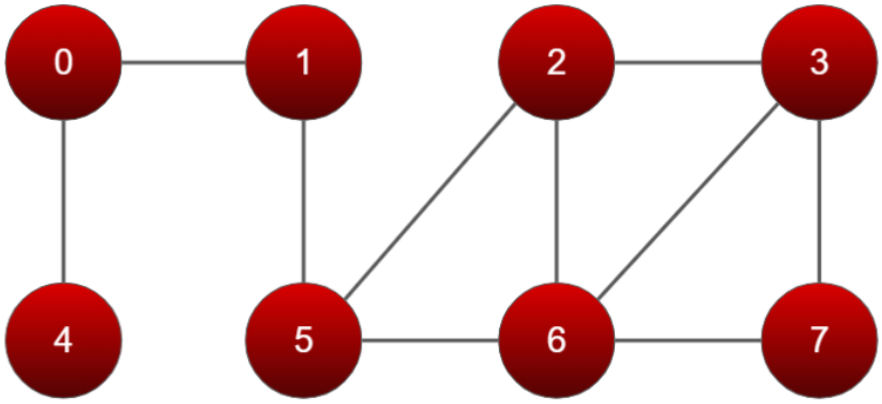


Traversing through 4 which is immediate on the queue

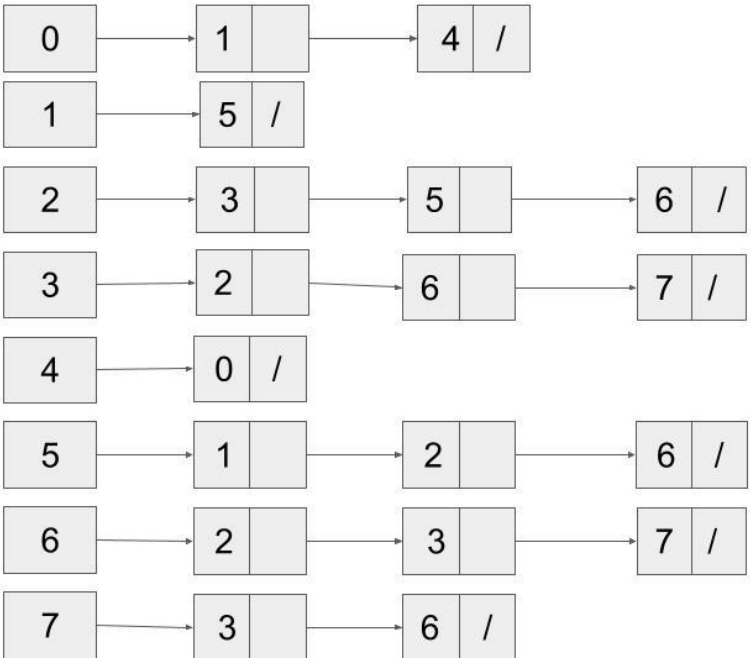


Finally all nodes are traversed




**ADJACENCY LIST:**



## OUTPUT CONSOLE

Edges and vertices are inserted according to the example schematic

```
                Depth First Search in Graph

Enter the number of Vertices: 8

Enter the vertices you want to enter an edge in between: 0 1
Do you want to insert more edges->Y to continue and N to Exit:Y

Enter the vertices you want to enter an edge in between: 0 4
Do you want to insert more edges->Y to continue and N to Exit:Y

Enter the vertices you want to enter an edge in between: 1 5
Do you want to insert more edges->Y to continue and N to Exit:Y

Enter the vertices you want to enter an edge in between: 5 2
Do you want to insert more edges->Y to continue and N to Exit:Y

Enter the vertices you want to enter an edge in between: 5 6
Do you want to insert more edges->Y to continue and N to Exit:Y

Enter the vertices you want to enter an edge in between: 6 2
Do you want to insert more edges->Y to continue and N to Exit:Y

Enter the vertices you want to enter an edge in between: 6 3
Do you want to insert more edges->Y to continue and N to Exit:Y

Enter the vertices you want to enter an edge in between: 6 7
Do you want to insert more edges->Y to continue and N to Exit:Y

Enter the vertices you want to enter an edge in between: 2 3
Do you want to insert more edges->Y to continue and N to Exit:Y

Enter the vertices you want to enter an edge in between: 3 7
Do you want to insert more edges->Y to continue and N to Exit:N
```

**Adjacency list:**

```
Displaying the Graph in adjacency List Format

 Adjacency list of vertex 0
 4 -> 1 ->

 Adjacency list of vertex 1
 5 -> 0 ->

 Adjacency list of vertex 2
 3 -> 6 -> 5 ->

 Adjacency list of vertex 3
 7 -> 2 -> 6 ->

 Adjacency list of vertex 4
 0 ->

 Adjacency list of vertex 5
 6 -> 2 -> 1 ->

 Adjacency list of vertex 6
 7 -> 3 -> 2 -> 5 ->

 Adjacency list of vertex 7
 3 -> 6 ->
```

**Suppose we want to start from 0**

```
Enter the vertex you want to search the dfs of: 0


The Stack contains: 0
The value poped is: 0
Visited 0


The unvisited vertex 4 is adjacent to 0
The unvisited vertex 1 is adjacent to 0
The Stack contains: 1 4
The value poped is: 1
Visited 1


The unvisited vertex 5 is adjacent to 1
Vertex 0 which is adjacent to 1 is already visited
The Stack contains: 5 4
The value poped is: 5
Visited 5


The unvisited vertex 6 is adjacent to 5
The unvisited vertex 2 is adjacent to 5
Vertex 1 which is adjacent to 5 is already visited
The Stack contains: 2 6 4
The value poped is: 2
Visited 2


The unvisited vertex 3 is adjacent to 2
Vertex 6 which is adjacent to 2 is already visited
Vertex 5 which is adjacent to 2 is already visited
The Stack contains: 3 6 4
The value poped is: 3
Visited 3


The unvisited vertex 7 is adjacent to 3
Vertex 2 which is adjacent to 3 is already visited
Vertex 6 which is adjacent to 3 is already visited
The Stack contains: 7 6 4
The value poped is: 7
Visited 7
```

```
Vertex 3 which is adjacent to 7 is already visited
Vertex 6 which is adjacent to 7 is already visited
The Stack contains: 6 4
The value poped is: 6
Visited 6


Vertex 7 which is adjacent to 6 is already visited
Vertex 3 which is adjacent to 6 is already visited
Vertex 2 which is adjacent to 6 is already visited
Vertex 5 which is adjacent to 6 is already visited
The Stack contains: 4
The value poped is: 4
Visited 4


Vertex 0 which is adjacent to 4 is already visited


Process returned 0 (0x0)   execution time : 54.662 s
Press any key to continue.
```

## CONCLUSION

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.

Mainly, the traversal of a graph can be done in two ways primarily:

1. Breadth first search which mainly prioritises all the nodes in the same breadth followed by the nodes in the other breadth/level.
2. Depth first search mainly prioritises recurrence of adjacent nodes to the highest depth and followed by backtracking and traversing the rest of the nodes.

Here in this lab, I explained the traversal process in both the traversal schemes step by step and showing the status of the queue after each step.