Bharatiya Vidya Bhavan's
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Autonomous Institute Affiliated to University of Mumbai)
Munshi Nagar, Andheri (W), Mumbai – 400 058.
COMPS Department

Experiment	8
Aim	**Implement Breadth Frst Search and Depth First Search Traversal for given Graph. Graph should be dynamic , means it should accept number of vertices and edges , dynamically. (It should not be fixed in program)**
Name	David Daniels
UID	2023300038
Class	Div -A
Batch	C
Date of Submission	17-10-24

Theory	**Graph Data Structure is a non-linear data structure consisting of vertices and edges.** It's used to represent relationships between different entities. **Graph algorithms** are methods used to manipulate and analyze graphs, solving various problems like **finding the shortest path** **a Graph is composed of a set of vertices (V) and a set of edges (E).** The graph is denoted by **G(V, E).** **<u>Components of Graph Data Structure</u>** • **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled. • **Edges:** Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabelled. **Representations of Graph** There are two common methods of representing a graph. These are, 1. Adjacency Matrix 2. Adjacency List When the graph contains a large number of edges then it is good to store it as a matrix because only some entries in the matrix will be empty.

1) Adjacency Matrix

An adjacency matrix is a way of representing a graph as a matrix of boolean (0's and 1's)
Let's assume there are **n** vertices in the graph So, create a 2D matrix **adjMat[n][n]** having dimension n x n.

- If there is an edge from vertex **i** to **j**, mark **adjMat[i][j]** as **1**.
- If there is no edge from vertex **i** to **j**, mark **adjMat[i][j]** as **0**
-

When the graph contains a large number of edges then it is good to store it as a matrix because only some entries in the matrix will be empty.

2) Adjacency List

An array of Lists is used to store edges between two vertices. The size of array is equal to the number of **vertices (i.e, n)**. Each index in this array represents a specific vertex in the graph. The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex **i**.
Let's assume there are **n** vertices in the graph So, create an **array of list** of size **n** as **adjList[n].**

- adjList[0] will have all the nodes which are connected (neighbour) to vertex **0**.
- adjList[1] will have all the nodes which are connected (neighbour) to vertex **1** and so on.

<u>Advantages of Graph Data Structure:</u>

- Graph Data Structure used to represent a wide range of relationships as we do not have any restrictions like previous data structures (Tree cannot have loops and have to be hierarchical. Arrays, Linked List, etc are linear)
- They can be used to model and solve a wide range of problems, including pathfinding, data clustering, network analysis, and machine learning.
- Any real world problem where we certain set of items and relations between them can be easily modeled as a graph and a lot of standard graph algorithms like BFS, DFS, Spanning Tree, Shortest Path, Topological Sorting and Strongly Connected
- Graph Data Structure can be used to represent complex data structures in a simple and intuitive way, making them easier to understand and analyze.

<u>Disadvantages of Graph Data Structure:</u>

- Graph Data Structure can be complex and difficult to understand, especially for people who are not familiar with graph theory or related algorithms.

- Creating and manipulating graphs can be computationally expensive, especially for very large or complex graphs.
- Graph algorithms can be difficult to design and implement correctly, and can be prone to bugs and errors.
- Graph Data Structure can be difficult to visualize and analyze, especially for very large or complex graphs, which can make it challenging to extract meaningful insights from the data.

1) **Breadth First Search (BFS)** is a fundamental **graph traversal algorithm.** It begins with a node, then first traverses all its adjacent. Once all adjacent are visited, then their adjacent are traversed. This is different from DFS in a way that closest vertices are visited before others. We mainly traverse vertices level by level. A lot of popular graph algorithms like Dijkstra's shortest path, Kahn's Algorithm, and Prim's algorithm are based on BFS. BFS itself can be used to detect cycle in a directed and undirected graph, find shortest path in an unweighted graph and many more problems.

Applications of BFS in Graphs:
BFS has various applications in graph theory and computer science, including:

- **Shortest Path Finding:** BFS can be used to find the shortest path between two nodes in an unweighted graph. By keeping track of the parent of each node during the traversal, the shortest path can be reconstructed.
- **Cycle Detection:** BFS can be used to detect cycles in a graph. If a node is visited twice during the traversal, it indicates the presence of a cycle.
- **Connected Components:** BFS can be used to identify connected components in a graph. Each connected component is a set of nodes that can be reached from each other.
- **Topological Sorting:** BFS can be used to perform topological sorting on a directed acyclic graph (DAG). Topological sorting arranges the nodes in a linear order such that for any edge (u, v), u appears before v in the order.
- **Level Order Traversal of Binary Trees:** BFS can be used to perform a level order traversal of a binary tree. This traversal visits all nodes at the same level before moving to the next level.
- **Network Routing:** BFS can be used to find the shortest path between two nodes in a network, making it useful for routing data packets in network protocols.

Advantages of Breadth First Search:

- BFS will never get trapped exploring the useful path forever.
- If there is a solution, BFS will definitely find it.
- If there is more than one solution then BFS can find the minimal one that requires less number of steps.
- Low storage requirement – linear with depth.
- Easily programmable.

Disadvantages of Breadth First Search:
The main drawback of BFS is its memory requirement. Since each level of the graph must be saved in order to generate the next level and the amount of memory is proportional to the number of nodes stored the space complexity of BFS is **O(b^d)**, where **b** is the branching factor(the number of children at each node, the outdegree) and **d** is the depth. As a result, BFS is severely space-bound in practice so will exhaust the memory available on typical computers in a matter of minutes.

2) **Depth First Traversal (**or DFS) for a graph is similar to Depth First Traversal of a tree. Like trees, we traverse all adjacent vertices one by one. When we traverse an adjacent vertex, we completely finish the traversal of all vertices reachable through that adjacent vertex. After we finish traversing one adjacent vertex and its reachable vertices, we move to the next adjacent vertex and repeat the process. This is similar to a tree, where we first completely traverse the left subtree and then move to the right subtree. The key difference is that, unlike trees, graphs may contain cycles (a node may be visited more than once). To avoid processing a node multiple times, we use a boolean visited array.

Applications of Depth First Search:
1. Detecting cycle in a graph: A graph has a cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.
2. Path Finding: We can specialize the DFS algorithm to find a path between two given vertices u and z.

- Call DFS(G, u) with u as the start vertex.
- Use a stack S to keep track of the path between the start vertex and the current vertex.
- As soon as destination vertex z is encountered, return the path as the contents of the stack

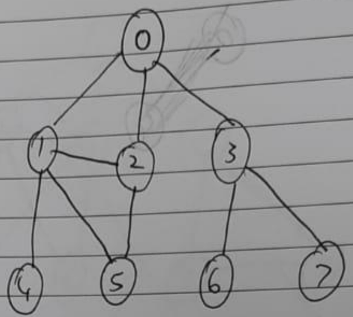
3. Topological Sorting: Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers.
4. To test if a graph is bipartite: We can augment either BFS or DFS when we first discover a new vertex, color it opposite its parents, and for each other edge, check it doesn’t link two vertices of the same color. The first vertex in any connected component can be red or black.
5. Finding Strongly Connected Components of a graph: A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.

	6. Solving puzzles with only one solution: such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.). **7. Web crawlers:** Depth-first search can be used in the implementation of web crawlers to explore the links on a website. **8. Maze generation:** Depth-first search can be used to generate random mazes. **9. Model checking:** Depth-first search can be used in model checking, which is the process of checking that a model of a system meets a certain set of properties. **10. Backtracking:** Depth-first search can be used in backtracking algorithms. **Advantages of Depth First Search:** • Memory requirement is only linear with respect to the search graph. This is in contrast with breadth-first search which requires more space. The reason is that the algorithm only needs to store a stack of nodes on the path from the root to the current node. • The time complexity of a depth-first Search to depth d and branching factor b (the number of children at each node, the outdegree) is O(bd) since it generates the same set of nodes as breadth-first search, but simply in a different order. Thus practically depth-first search is time-limited rather than space-limited. • If depth-first search finds solution without exploring much in a path then the time and space it takes will be very less. • DFS requires less memory since only the nodes on the current path are stored. By chance DFS may find a solution without examining much of the search space at all. **Disadvantages of Depth First Search:** • The disadvantage of Depth-First Search is that there is a possibility that it may down the left-most path forever. Even a finite graph can generate an infinite solution to this problem is to impose a cutoff depth on the search. Although ideal cutoff is the solution depth d and this value is rarely known in advance of actually solving the problem. If the chosen cutoff depth is less than d, the algorithm will fail to find a solution, whereas if the cutoff depth is greater than d, a large price is paid in execution time, and the first solution found may not be an optimal one. • Depth-First Search is not guaranteed to find the solution. • And there is no guarantee to find a minimal solution, if more than one solution.
Algorithm	1) Expand_graph **Memory Reallocation for Vertices**: • The graph's adjacency list (array of pointers) is resized to accommodate the new number of vertices using realloc. **Reallocate Inner Arrays**: For each vertex, the inner arrays (which represent edges to other vertices) are also resized to match the new number of vertices.

	Initialize New Vertices: • A nested loop is used to set the newly added vertices' connections to zero, indicating no edges exist between them and any vertices (both existing and new). 2) Add_edge • **Update Adjacency Matrix**: The adjacency matrix is updated: • The entry corresponding to the connection from e1 to e2 is set to 1, indicating an edge exists from e1 to e2. • The entry for the reverse connection from e2 to e1 is also set to 1, ensuring the graph remains undirected. 3) BFS **Initialization:** Enqueue the given source vertex into a queue and mark it as visited. 1. **Exploration:** While the queue is not empty: o Dequeue a node from the queue and visit it (e.g., print its value). o For each unvisited neighbor of the dequeued node: ▪ Enqueue the neighbor into the queue. ▪ Mark the neighbor as visited. 2. **Termination:** Repeat step 2 until the queue is empty. 4) DFS 1. Start by putting any one of the graph's vertices on top of a stack. 2. Take the top item of the stack and add it to the visited list. 3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack. 4. Keep repeating steps 2 and 3 until the stack is empty.

Problem Solving

David Daniels 2023300038



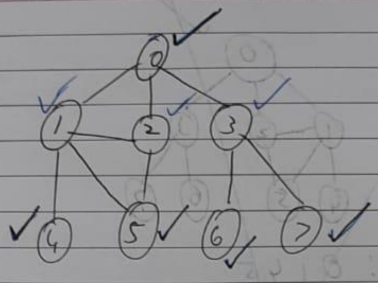
Policy for BFS, DFS: ascending order

We will use matrix representation for graph

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

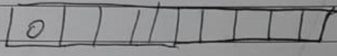
1) BFS

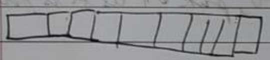
Take starting ~~node~~ vertex 0



Output : 0 1 2 3 4 5 6 7

Queue :

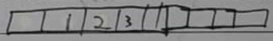




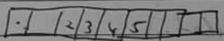
node = 0

check in arry[0][i] for all i if it is in adjacency matrix and not visited and enque them and set visited

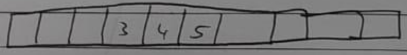
i.e



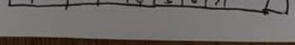
check all the adjacent nodes for 1, remove 1



Checking nodes for 2 gives no new vertices
simply dequeue, ~~similarly for 4, 5~~

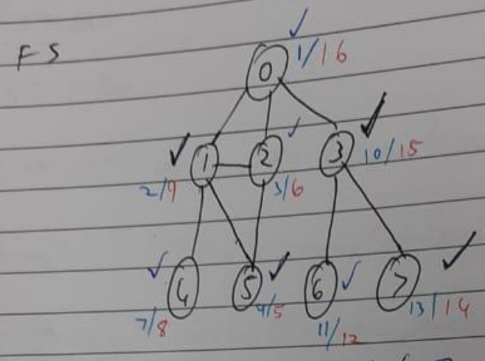


for ~~nodes~~ adj vertices of 3, we get 6, 7 and remove 3



Continuing for 4, 5, 6, 7
gives no new vertices

DFS



Output: 0 1 2 5 4 3 6 7

Stack:

0

Look at all adjacent vertices of 0. Pick 1 as per policy and push into stack

1
0

As per policy from vertex 1, pick vertex 2 push it in stack, print

2
1
0

From 2, unvisited node is 5 push it and print

5
2
1
0

5 has no unvisited adjacent nodes. Pop it similarly for 2

$\rightarrow$

2
1
0

$\rightarrow$

1
0

Pick vertex 4 and push

4
1
0

No adjacent nodes for 4 and 1. Pop the both

0

$\rightarrow$ Only adjacent non visited node for 0 is 3. Push the vertex

3
0

$\rightarrow$ Push 6

6
3
0

$\xrightarrow{Pop}$ 6

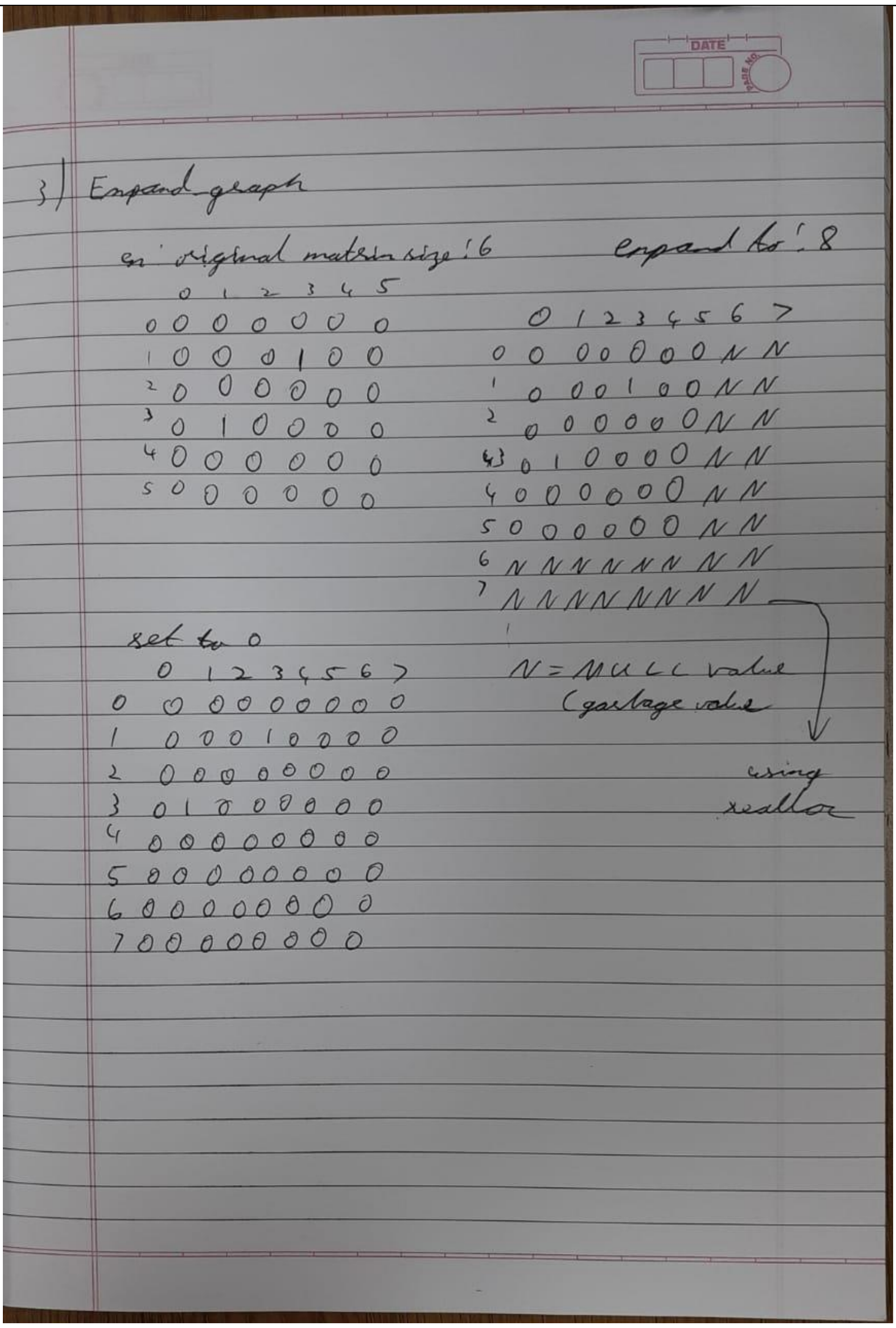
3
0

$\xrightarrow{Push}$ 7

7
3
0

Pop all nodes $\longrightarrow$

(all nodes are visited



3) Expand graph

ex: original matrix size: 6

$$\begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 1 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

expand to: 8

$$\begin{array}{c|cccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & N & N \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & N & N \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 & N & N \\ 3 & 0 & 1 & 0 & 0 & 0 & 0 & N & N \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 & N & N \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 & N & N \\ 6 & N & N & N & N & N & N & N & N \\ 7 & N & N & N & N & N & N & N & N \end{array}$$

N = NULL value
(garbage value

using realloc

set to 0

$$\begin{array}{c|cccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Program(Code)

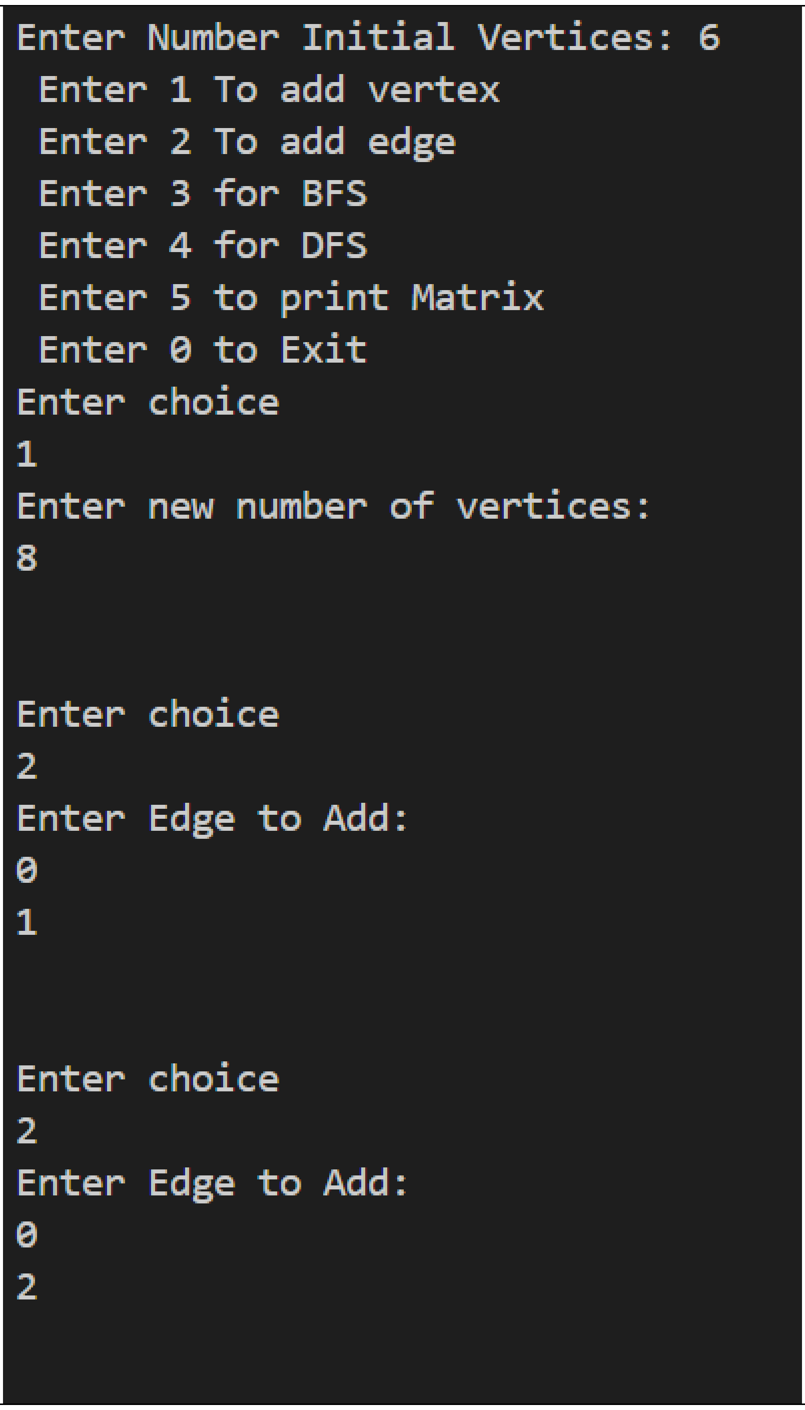
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct graph
{
    int no_of_edges;
    int no_of_vertices;
    //array of adjacency graphs
    int ** arr;
}graph;
```

```
graph* expand_graph(graph * g ,int vertices) {
  int old_vertices_no= g->no_of_vertices;
  g->no_of_vertices=vertices;

    g->arr= realloc(g->arr,g->no_of_vertices * sizeof(int*));
    for(int i = 0; i < g->no_of_vertices; i++)
    {
        g->arr[i] = realloc(g->arr[i],g->no_of_vertices * sizeof(int));
    }

  //set new nodes to null
  for (int i = old_vertices_no; i <vertices ; i++)
  {
    for (int j = 0; j < vertices; j++)
    {
        g->arr[i][j]=0;
        g->arr[j][i]=0;
    }
  }

  return g;
}

graph * add_edge (graph * g ,int e1,int e2)
{
    //printf("ADDING %d , %d \n",e1,e2);
    g->no_of_edges++;
    g->arr[e1][e2]=1;
    g->arr[e2][e1]=1;
    return g;
}

typedef struct queue{
    int* items;
    int front;
    int rear;
    int size;
} queue;

bool isEmpty(queue* q) { return (q->front == q->rear - 1); }

bool isFull(queue* q) { return (q->rear == q->size); }

void enqueue(queue* q, int value)
{
    if (isFull(q)) {
        printf("queue is full\n");
        return;
    }
    q->items[q->rear] = value;
```

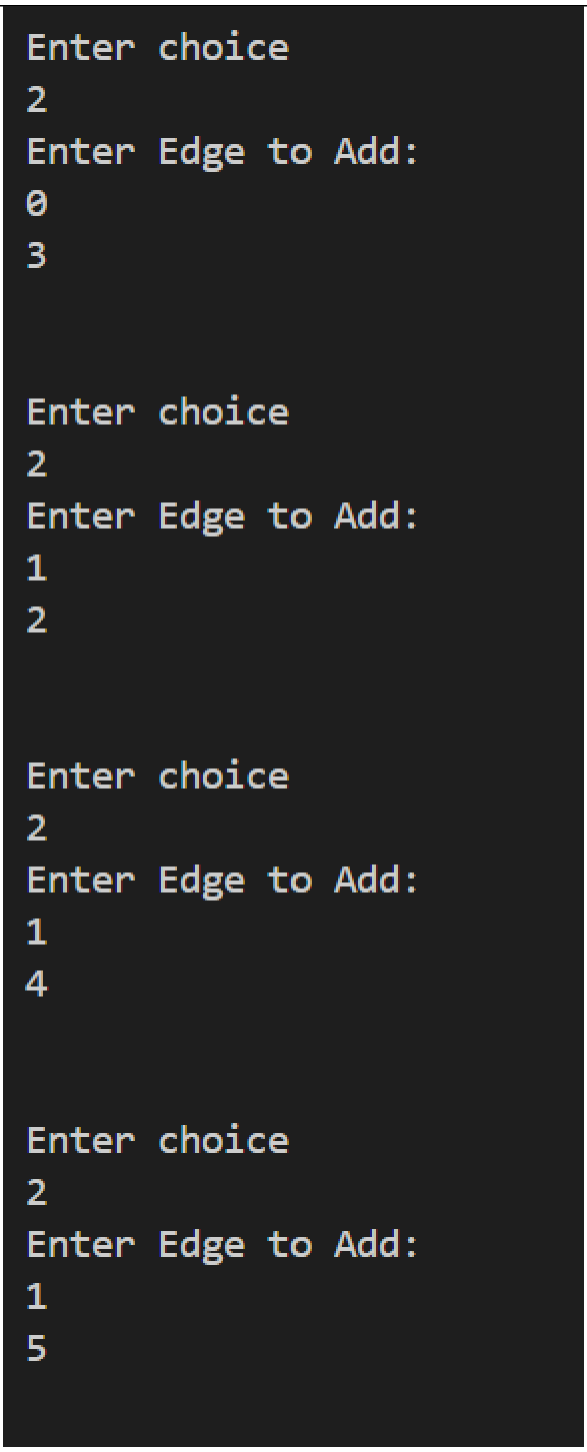
```
   q->rear++;
}

int dequeue(queue* q)
{
   if (isEmpty(q)) {
      printf("queue is empty\n");
      return -1;
   }
   q->front++;
   return q->items[q->front];
}

int * init_visited(graph  * g)
{
int * visited = (int *) malloc(g->no_of_vertices * sizeof(int));
   for (int i = 0; i < g->no_of_vertices; i++)
   {
      visited[i]=0;
   }
return visited;
}

void bfs_with_level(graph * g,int start)
{
   int * visited = init_visited(g);
   queue * q = (queue * )malloc(sizeof(queue));
   int size_o_q=10;
   q->size=size_o_q;
   q->items=(int *)malloc(size_o_q*sizeof(int));
   q->front = -1;
   q->rear = 0;
   int level=0;

   //i IS SAME AS STARTING VERTEX
   int i=start;
   printf("%d ,Level= %d\n",i,level);

   enqueue(q,i);
   printf("Enqueuing %d \n",i);
   visited[i]=1;

   while (!isEmpty(q))
   {
      int node = dequeue(q);
      printf("Dequeuing %d \n",node);
      level++;

      for (int j = 0; j < g->no_of_vertices; j++)
      {
            //if edge is 1 and node not visited
            if (g->arr[node][j]==1 && visited[j]==0)
            {
                //printf("%d \n",j);
                printf("%d ,Level= %d\n",j,level);
                visited[j]=1;
                enqueue(q,j);
                printf("Enqueuing %d \n",j);
            }

        }
        level++;
        i++;

    }
    free(q->items);
    free(q);
}

void bfs(graph * g,int start)
{
    int * visited = init_visited(g);
    queue * q = (queue * )malloc(sizeof(queue));
    int size_o_q=10;
    q->size=size_o_q;
    q->items=(int *)malloc(size_o_q*sizeof(int));
    q->front = -1;
    q->rear = 0;

    //i IS SAME AS STARTING VERTEX
    int i=start;
    printf("%d ",i);
    enqueue(q,i);
    visited[i]=1;

    while (!isEmpty(q))
    {
        int node = dequeue(q);


        for (int j = 0; j < g->no_of_vertices; j++)
        {
            //if edge is 1 and node not visited
            if (g->arr[node][j]==1 && visited[j]==0)
            {
                printf("%d ",j);
                visited[j]=1;
                enqueue(q,j);
            }
        }
        i++;


    }
    free(q->items);
    free(q);
}
int time = 0;
int *start_time;
int *end_time;

void dfs_with_time(graph *g, int start, int *visited) {
    if (start_time == NULL) {
        start_time = (int *)malloc(sizeof(int) * g->no_of_vertices);
        end_time = (int *)malloc(sizeof(int) * g->no_of_vertices);
        for (int i = 0; i < g->no_of_vertices; i++) {
            start_time[i] = 0;
            end_time[i] = 0;
        }
    }

    time++;
    visited[start] = 1;
    start_time[start] = time;

    printf("Node %d: Entering Time = %d\n", start, start_time[start]);

    for (int j = 0; j < g->no_of_vertices; j++) {
        if (g->arr[start][j] == 1 && !visited[j]) {
            dfs_with_time(g, j, visited);
        }
    }

    time++;
    end_time[start] = time;
    printf("Node %d: Exiting Time = %d\n", start, end_time[start]);
}

void show_table(graph *g) {
    printf("\nNode\tStart Time\tEnd Time\n");
    for (int i = 0; i < g->no_of_vertices; i++) {
        printf("%d\t%d\t\t%d\n", i, start_time[i], end_time[i]);
    }
}

void dfs(graph *g, int start, int * visited) {
    int i=start;
    printf("%d ",i);
    visited[i]=1;
    for (int j = 0; j < g->no_of_vertices; j++)
    {
        if (g->arr[i][j]==1 && !visited[j])
        {
            dfs(g,j,visited);
        }

    }

}

void display_graph(graph * g)
{
    for (int i = 0; i < g->no_of_vertices; i++)
    {
        for (int j = 0; j < g->no_of_vertices; j++)
        {
            printf("%d ",g->arr[i][j]);
        }
        printf("\n");
    }

}

void choice(graph * g)
{
    int choice=0,num=0;
    printf(" Enter 1 To add vertex \n Enter 2 To add edge \n Enter 3 for
BFS \n Enter 4 for DFS \n Enter 5 to print Matrix \n Enter 0 to Exit \n");

    while (true)
    {

    printf("Enter choice \n");
        scanf("%d",&choice);
        if (choice==0)
        {
            printf("\nGoodbye\n");
            break;
        }
    switch (choice)
    {
    case 1:
        {
            printf("Enter new number of vertices: \n");
            scanf("%d", &num);
            expand_graph(g,num);
            printf("\n\n");
            break;
        }
    case 2:
        {
            printf("Enter Edge to Add: \n");
            scanf("%d", &num);
            int num1=num;
            scanf("%d", &num);
            int num2=num;
            add_edge(g,num1,num2);
            printf("\n\n");
            break;
        }
    case 3:
        {
            printf("Enter Start Vertice: \n");
            scanf("%d", &num);
            printf("BFS is: \n");
            bfs(g,num);
            bfs_with_level(g,num);
            printf("\n\n");
            break;
        }
    case 4:
        {
            printf("Enter Start Vertice: \n");
            scanf("%d", &num);
            int * visited=init_visited(g);
            printf("DFS is: \n");
            dfs_with_time(g,num,visited);
            show_table(g);
            printf("\n\n\n");
            visited=init_visited(g);
            dfs(g,num,visited);
            printf("\n\n\n");
            break;
        }
    case 5:
        {
            printf("Graph Matrix is: \n");
            display_graph(g);
            break;
        }
    default:
        {
            printf("Error");
            break;
        }
    }
}
}
```

```
int main(int argc, char const *argv[])
{
   graph * g = (graph *) malloc(sizeof(graph));
   printf("Enter Number Initial Vertices: ");
   int num=0;
   scanf("%d",&num);
   g->no_of_vertices=num;
   g->no_of_edges=0;

   // make 2d array
   g->arr= malloc(g->no_of_vertices * sizeof(int*));
   for(int i = 0; i < g->no_of_vertices; i++)
   {
      g->arr[i] = malloc(g->no_of_vertices * sizeof(int));
   }

   for (int i = 0; i < g->no_of_vertices; i++)
   {
      for (int j = 0; j < g->no_of_vertices; j++)
      {
         g->arr[i][j]=0;
      }
   }

   choice(g);

   for(int i = 0; i < g->no_of_vertices; i++)
   {
   free(g->arr[i]);
   }

   free(g->arr);
   free(g);

   return 0;
}
```

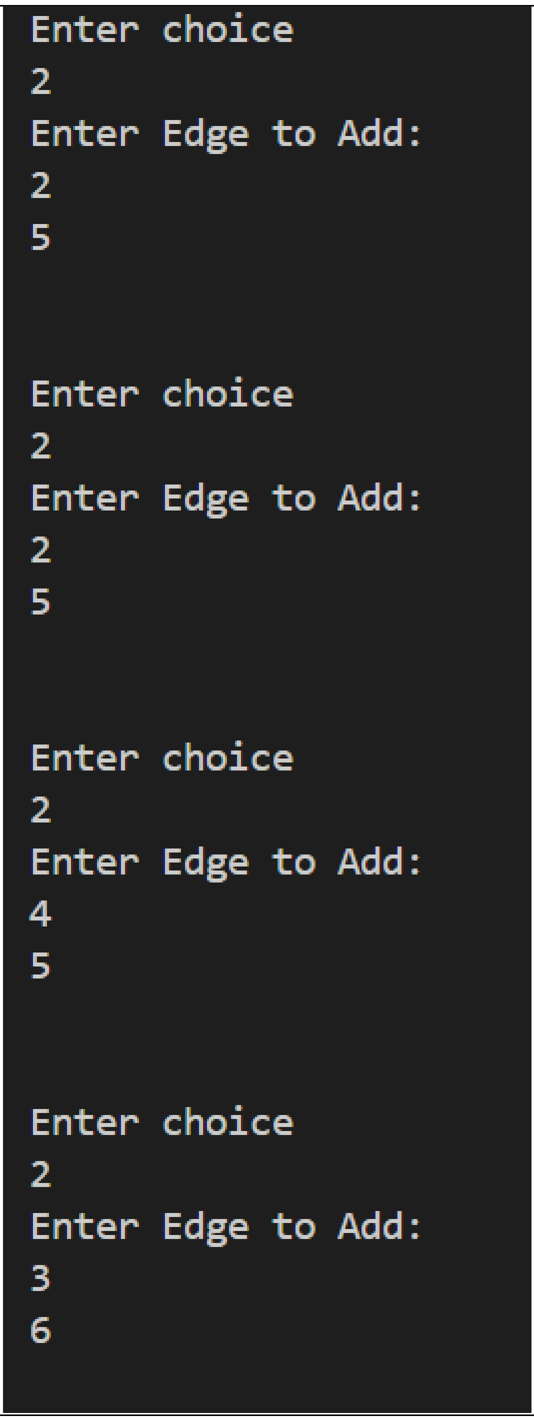
Output



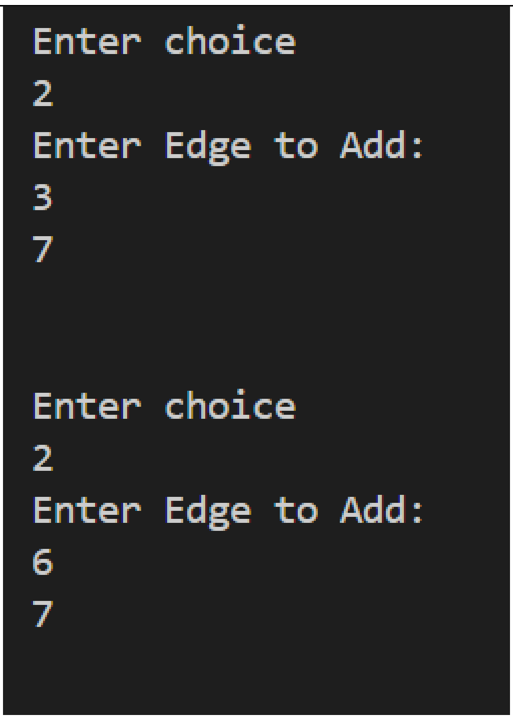
```
Enter Number Initial Vertices: 6
 Enter 1 To add vertex
 Enter 2 To add edge
 Enter 3 for BFS
 Enter 4 for DFS
 Enter 5 to print Matrix
 Enter 0 to Exit
Enter choice
1
Enter new number of vertices:
8


Enter choice
2
Enter Edge to Add:
0
1


Enter choice
2
Enter Edge to Add:
0
2
```



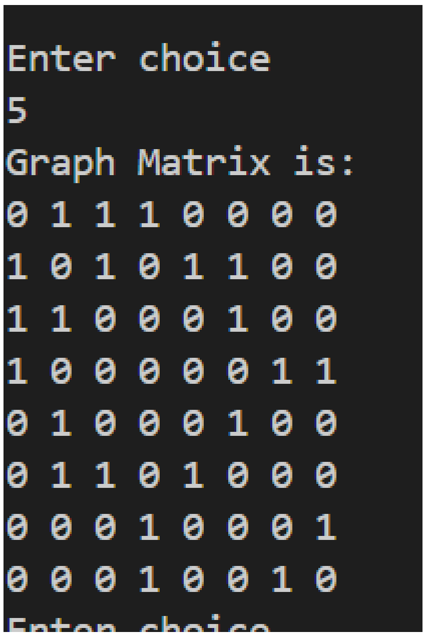
```
Enter choice
2
Enter Edge to Add:
0
3


Enter choice
2
Enter Edge to Add:
1
2


Enter choice
2
Enter Edge to Add:
1
4


Enter choice
2
Enter Edge to Add:
1
5
```



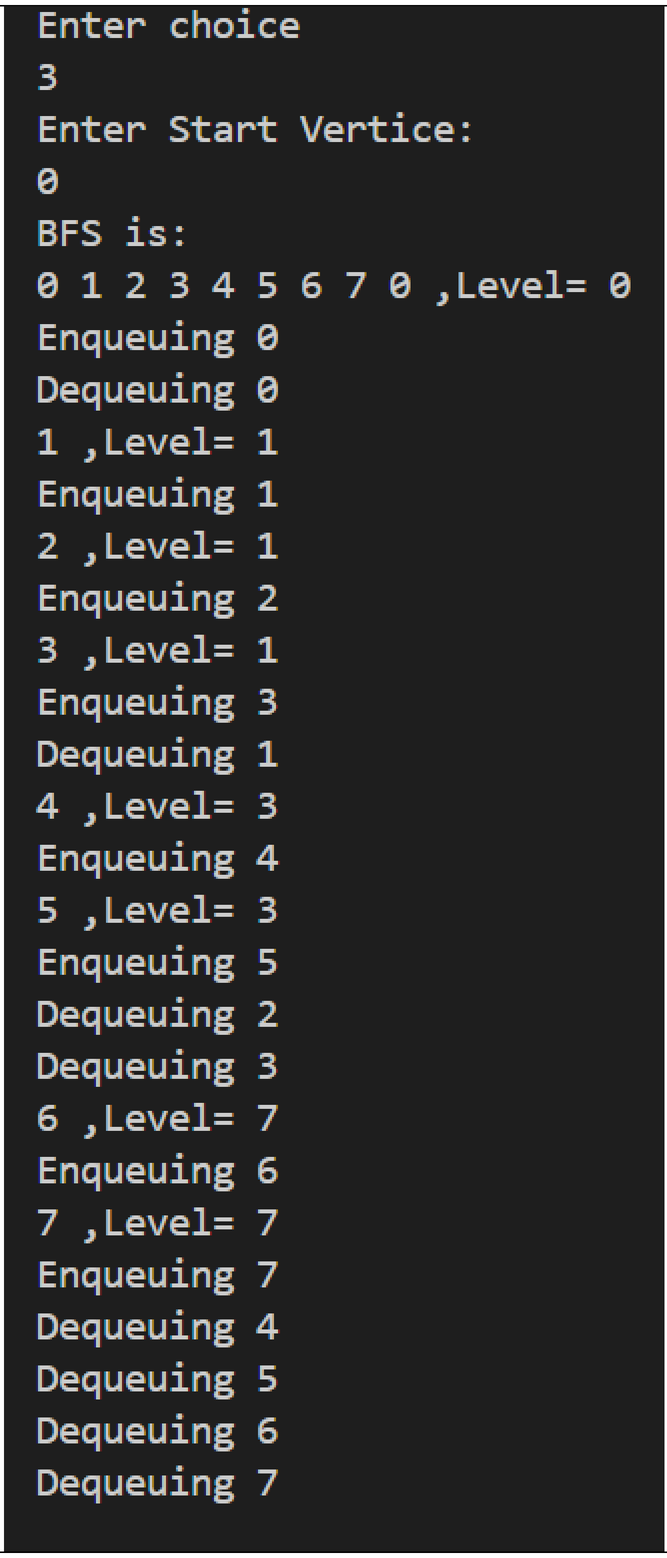
```
Enter choice
2
Enter Edge to Add:
2
5


Enter choice
2
Enter Edge to Add:
2
5


Enter choice
2
Enter Edge to Add:
4
5


Enter choice
2
Enter Edge to Add:
3
6
```



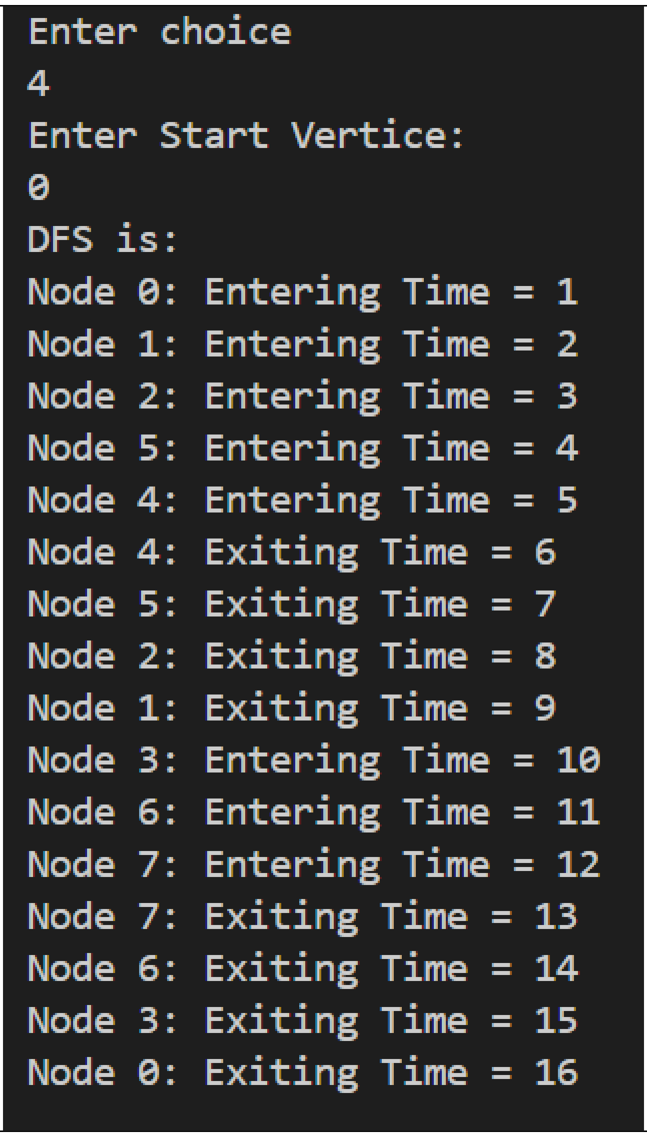
```
Enter choice
2
Enter Edge to Add:
3
7


Enter choice
2
Enter Edge to Add:
6
7
```



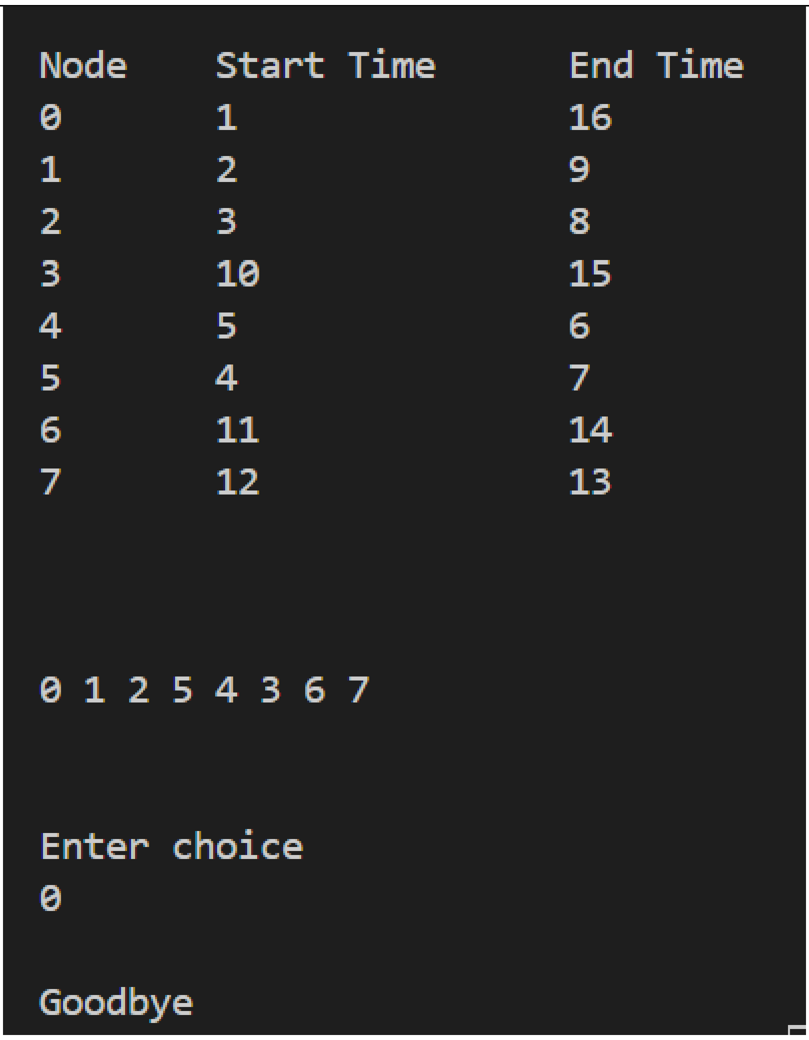
```
Enter choice
5
Graph Matrix is:
0 1 1 1 0 0 0 0
1 0 1 0 1 1 0 0
1 1 0 0 0 1 0 0
1 0 0 0 0 0 1 1
0 1 0 0 0 1 0 0
0 1 1 0 1 0 0 0
0 0 0 1 0 0 0 1
0 0 0 1 0 0 1 0
Enter choice
```

```
Enter choice
3
Enter Start Vertice:
0
BFS is:
0 1 2 3 4 5 6 7 0 ,Level= 0
Enqueuing 0
Dequeuing 0
1 ,Level= 1
Enqueuing 1
2 ,Level= 1
Enqueuing 2
3 ,Level= 1
Enqueuing 3
Dequeuing 1
4 ,Level= 3
Enqueuing 4
5 ,Level= 3
Enqueuing 5
Dequeuing 2
Dequeuing 3
6 ,Level= 7
Enqueuing 6
7 ,Level= 7
Enqueuing 7
Dequeuing 4
Dequeuing 5
Dequeuing 6
Dequeuing 7
```

```
Enter choice
4
Enter Start Vertice:
0
DFS is:
Node 0: Entering Time = 1
Node 1: Entering Time = 2
Node 2: Entering Time = 3
Node 5: Entering Time = 4
Node 4: Entering Time = 5
Node 4: Exiting Time = 6
Node 5: Exiting Time = 7
Node 2: Exiting Time = 8
Node 1: Exiting Time = 9
Node 3: Entering Time = 10
Node 6: Entering Time = 11
Node 7: Entering Time = 12
Node 7: Exiting Time = 13
Node 6: Exiting Time = 14
Node 3: Exiting Time = 15
Node 0: Exiting Time = 16
```



```
Node    Start Time      End Time
0       1               16
1       2               9
2       3               8
3       10              15
4       5               6
5       4               7
6       11              14
7       12              13



0 1 2 5 4 3 6 7


Enter choice
0

Goodbye
```

Conclusion	Thus, we implemented dynamic Breadth-First Search (BFS) and Depth-First Search (DFS) traversals for a graph that allows the user to specify the number of vertices and edges at runtime. This flexibility enhances our understanding of graph structures and algorithms, showcasing their adaptability to various scenarios. By experimenting with different graph configurations, we observed how BFS and DFS yield distinct traversal paths, highlighting their unique characteristics and applications in graph theory.