ROLL NO:2019103033

NAME:MANOJ KUMAR M

1)DFS(for disconnected also)

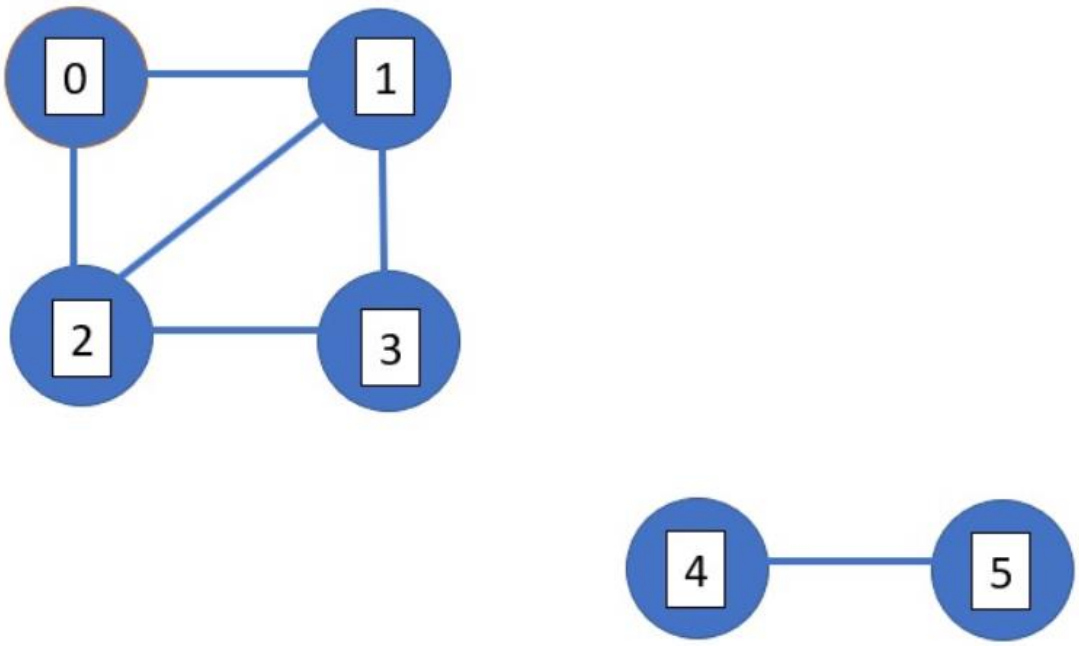
```c
#include <stdio.h>
#define MAX 100

int matrix[MAX][MAX], visited[MAX];

//STACK
int top = -1;
int stack[MAX];

void push(int item) //O(1)
{
    stack[++top] = item;
}

int pop() //O(1)
{
    return stack[top--];
}

int peek() //O(1)
{
    return stack[top];
}

int isStackEmpty() //O(1)
{
    return top == -1;
}

void init(int vertices) //O(V^2)
{
    int i, j;

    for (i = 0; i < vertices; i++)
    {
        for (j = 0; j < vertices; j++)
        {
            matrix[i][j] = 0;
        }
    }
}
void addEdge(int vertices) //O(V^2)
{
```

```
    int i, j, v;

    for (i = 0; i < vertices; i++)
    {
        visited[i] = -1;
        for (j = 0; j < vertices - 1; j++)
        {
            printf("Enter the adjacent node of %d(-1 to exit): ", i);
            scanf("%d", &v);
            if (v == -1)
            {
                break;
            }
            matrix[i][v] = 1;
        }
    }

    return;
}
int getAdjNode(int vertex, int vertices) //O(V)
{
    int i;
    for (i = 0; i < vertices; i++)
    {
        if (matrix[vertex][i] == 1 && visited[i] == -1)
        {
            return i;
        }
    }
    return -1;
}

void DFS(int vertices, int vertex) //O(V^2)
{
    int i;

    visited[vertex] = 1;
    printf("%d\t", vertex);
    push(vertex);

    while (!isStackEmpty())
    {
        int adjunviNode = getAdjNode(peek(), vertices);
        if (adjunviNode == -1)
        {
            pop();
        }
        else
```

```
        {
            visited[adjunviNode] = 1;
            printf("%d\t", adjunviNode);
            push(adjunviNode);
        }
    }
}

void DFSdis(int v)
{ //O(V^2)  //It works for a disonnected graph also
    for (int i = 0; i < v; i++)
    {
        if (visited[i] == -1)
        {
            DFS(v, i);
            printf("\n");
        }
    }
}

void printMatrix(int vertices) //O(V^2)
{
    for (int i = 0; i < vertices; i++)
    {
        for (int j = 0; j < vertices; j++)
        {
            printf("%d\t", matrix[i][j]);
        }
        printf("\n");
    }
}
int main()
{
    int vertices;

    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);
    printf("\n");

    init(vertices);

    addEdge(vertices);

    printf("The adjacency matrix is:\n");
    printMatrix(vertices);

    printf("\n");
    printf("DFS TRAVERSAL:\n");
```

```
    DFSdis(vertices);
}
```

GRAPH:



OUTPUT:

```
Enter the number of vertices: 6

Enter the adjacent node of 0(-1 to exit): 1
Enter the adjacent node of 0(-1 to exit): 2
Enter the adjacent node of 0(-1 to exit): -1
Enter the adjacent node of 1(-1 to exit): 0
Enter the adjacent node of 1(-1 to exit): 2
Enter the adjacent node of 1(-1 to exit): 3
Enter the adjacent node of 1(-1 to exit): -1
Enter the adjacent node of 2(-1 to exit): 0
Enter the adjacent node of 2(-1 to exit): 1
Enter the adjacent node of 2(-1 to exit): 3
Enter the adjacent node of 2(-1 to exit): -1
Enter the adjacent node of 3(-1 to exit): 1
Enter the adjacent node of 3(-1 to exit): 2
Enter the adjacent node of 3(-1 to exit): -1
Enter the adjacent node of 4(-1 to exit): 5
Enter the adjacent node of 4(-1 to exit): -1
Enter the adjacent node of 5(-1 to exit): 4
Enter the adjacent node of 5(-1 to exit): -1
```

```
The adjacency matrix is:
0       1       1       0       0       0
1       0       1       1       0       0
1       1       0       1       0       0
0       1       1       0       0       0
0       0       0       0       0       1
0       0       0       0       1       0

DFS TRAVERSAL:
0       1       2       3
4       5
```

TIME COMPLEXITY ANALYSIS:

Let V be the number of vertices and E be the number of edges.

We are using adjacency matrix here.During dfs traversal,each vertex v is called atleast once.From each vertex, all the vertices are checked until a negihbouring vertex is found.The complexity of this proces is O(V).Since this process is done for all the vertices,The Overall complexity is $O(VxV)=O(V^2)$.

2)BFS(including disconnected graph)

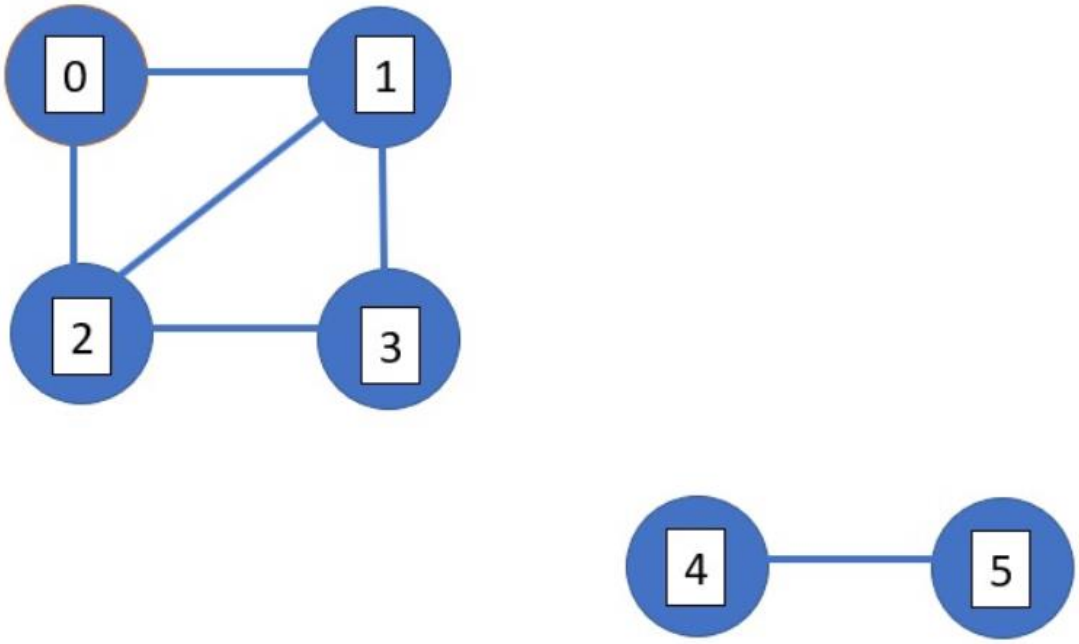
```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 100

typedef struct Node
{
    int vertex;
    struct Node *next;
} node;

typedef struct Graph
{
    int visited;
    struct Node *adjList;
} graph;

//QUEUE
int front = -1;
int rear = -1;
int queue[MAX];

void enqueue(int v) //O(1)
{
    if (front == -1 && rear == -1)
    {
        queue[++front] = queue[++rear] = v;
        return;
    }

    queue[++rear] = v;
    return;
}
int dequeue() //O(1)
{
```

```
    return queue[front++];
}
int getFront() //O(1)
{
    return queue[front];
}
int isQueueEmpty() //O(1)
{
    if (front > rear)
    {
        return 1;
    }
    return 0;
}
node *createNode(int v) //O(1)
{
    node *newnode = (node *)malloc(sizeof(node));
    newnode->vertex = v;
    newnode->next = NULL;

    return newnode;
}
void createGraph(graph *p, int v) //O(v)
{
    int i;
    for (i = 0; i < v; i++)
    {
        p[i].adjList = (node *)malloc(sizeof(node));
        p[i].visited = -1;
        p[i].adjList = NULL;
    }

    return;
}
void addEdge(graph *graph, int src, int dest) //O(1)
{
    node *newNode = createNode(dest);
    newNode->next = graph[src].adjList;
    graph[src].adjList = newNode;

    return;
}
int getAdjunvisited(graph *graph, int vertex) //O(V)--
>v is the number of vertices
{
    node *temp = graph[vertex].adjList;

    while (temp)
```

```
    {
        if (graph[temp->vertex].visited == -1)
        {
            return temp->vertex;
        }
        temp = temp->next;
    }

    return -1;
}
void BFS(graph *graph, int v, int vertex) //O(V+E)
{
    int i;

    graph[vertex].visited = 1;
    enqueue(vertex);

    while (!isQueueEmpty())
    {
        node *temp = graph[vertex].adjList;
        while (temp)
        {
            if (graph[temp->vertex].visited == -1)
            {
                enqueue(temp->vertex);
                graph[temp->vertex].visited = 1;
            }
            temp = temp->next;
        }
        printf("%d\t", getFront());
        dequeue();

        vertex = getFront();
    }

    printf("\n");
}
void BFSdis(graph *g, int v)
{ //O(V+E)  //It traverses the disconnected components also
    for (int i = 0; i < v; i++)
    {
        if (g[i].visited == -1)
            BFS(g, v, i);
    }
}
void printAdjNodes(graph *g, int v) //O(E)
{
    for (int i = 0; i < v; i++)
```

```
    {
        node *temp = g[i].adjList;
        printf("%d->", i);
        while (temp != NULL)
        {
            printf("%d,", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }
}
int main()
{
    int vertices, i, adj, j;
    printf("\nEnter the number of vertices: ");
    scanf("%d", &vertices);
    printf("\n");

    graph g[vertices];
    createGraph(g, vertices);

    for (i = 0; i < vertices; i++)
    {
        for (j = 0; j < vertices - 1; j++)
        {
            printf("Enter the adjacent node of %d(-1 to exit): ", i);
            scanf("%d", &adj);
            if (adj == -1)
            {
                break;
            }
            addEdge(g, i, adj);
        }
    }

    printAdjNodes(g, vertices);
    printf("\n");
    printf("BFS TRAVERSAL:\n");
    BFSdis(g, vertices);
    return 0;
}
```

GRAPH:



OUTPUT:

```
Enter the number of vertices: 6

Enter the adjacent node of 0(-1 to exit): 1
Enter the adjacent node of 0(-1 to exit): 2
Enter the adjacent node of 0(-1 to exit): -1
Enter the adjacent node of 1(-1 to exit): 0
Enter the adjacent node of 1(-1 to exit): 2
Enter the adjacent node of 1(-1 to exit): 3
Enter the adjacent node of 1(-1 to exit): -1
Enter the adjacent node of 2(-1 to exit): 0
Enter the adjacent node of 2(-1 to exit): 1
Enter the adjacent node of 2(-1 to exit): 3
Enter the adjacent node of 2(-1 to exit): -1
Enter the adjacent node of 3(-1 to exit): 1
Enter the adjacent node of 3(-1 to exit): 2
Enter the adjacent node of 3(-1 to exit): -1
Enter the adjacent node of 4(-1 to exit): 5
Enter the adjacent node of 4(-1 to exit): -1
Enter the adjacent node of 5(-1 to exit): 4
Enter the adjacent node of 5(-1 to exit): -1
0->2,1,
1->3,2,0,
2->3,1,0,
3->2,1,
4->5,
5->4,
```

```
BFS TRAVERSAL:
0       2       1       3
4       5
```

TIME COMPLEXITY ANALYSIS:

Let V be the number of vertices and E be the number of edges.

We are using adjacency list here.During bfs traversal,each vertex v is called atleast once.From,each vertex all of its neighbouring vertices are checked,it's complexity (summing over all vertices) is O(E).Constant operation of changing the visited flag to 1 happens for every vertex.It's complexity is O(V).The overall complexity is O(V)+O(E)=O(V+E).

3)THREADED BINARY TREE

```c
#include <stdio.h>
#include <stdlib.h>
//I have inserted data in level order using a queue

#define used 1
#define unused 0

typedef struct node
{
    int data;
    struct node *left;
    struct node *right;
    int lt;
    int rt;
}node;

//Queue
struct queue
{
    struct node *data;
    struct queue *next;
} typedef queue;

queue *head = NULL, *rear = NULL;

void enqueue(node *val)
{ //O(1)
    queue *n = (queue *)malloc(sizeof(queue));
    if (!n)
    {
        printf("\nOverflow\n");
        return;
    }
```

```
    n->data = val;
    n->next = NULL;
    if (!head && !rear)
    {
        head = n;
        rear = n;
    }
    else
    {
        rear->next = n;
        rear = n;
    }
}

node *dequeue()
{ //O(1)
    if (!head)
    {
        return NULL;
    }
    queue *temp = head;
    head = head->next;
    node *del = temp->data;

    free(temp);
    if (!head)
    {
        rear = NULL;
    }
    return del;
}

int isEmpty()
{ //O(1)
    if (head == NULL && rear == NULL)
        return 1;
    return 0;
}

//THREADED BINARY TREE

node *createNode(int data)
{ //O(1)
    node *new = (node *)malloc(sizeof(node));

    new->data = data;
    new->left = NULL;
    new->right = NULL;
```

```
    new->lt = unused;
    new->rt = unused;

    return new;
}

//LEVEL ORDER INSERT
node *insert(node *root, int data)
{ //O(height of the tree)
    if (!root)
    {
        return createNode(data);
    }

    enqueue(root);
    int leftinsert = 1;
    node *ptr = NULL, *parent = NULL;

    while (!isEmpty())
    {
        ptr = dequeue();
        parent = ptr;

        if (ptr->lt == unused && ptr->left)
        {
            enqueue(ptr->left);
        }
        else
        {
            leftinsert = 1;
            break;
        }

        if (ptr->rt == unused && ptr->right)
        {
            enqueue(ptr->right);
        }
        else
        {
            leftinsert = 0;
            break;
        }
    }

    //THREADING--O(1)
    node *tmp = createNode(data);
    tmp->lt = used;
    tmp->rt = used;
```

```
    //when to be inserted as left child
    if (leftinsert)
    {
        tmp->left = parent->left;
        tmp->right = parent;

        parent->lt = unused;
        parent->left = tmp;
    }
    //when inserted as the right child
    else
    {
        tmp->right = parent->right;
        tmp->left = parent;

        parent->rt = unused;
        parent->right = tmp;
    }

    return root;
}

node *inorderSuccessor(node *ptr)
{ //O(height of the tree)
    if (ptr->rt == used)
        return ptr->right;

    ptr = ptr->right;
    while (ptr->lt == unused)
        ptr = ptr->left;

    return ptr;
}

void inorder(node *root)
{ //O(n),where n is the number of nodes
    if (root == NULL)
    {
        printf("Tree is empty\n");
    }

    node *ptr = root;
    while (ptr->lt == unused)
        ptr = ptr->left;
    printf("\nTree: ");
    while (ptr != NULL)
    {
```

```
        printf("%d ", ptr->data);
        ptr = inorderSuccessor(ptr);
    }
}

int main()
{
    int choice, data;
    node *root = NULL;
    while (1)
    {
        printf("\n1. INSERT \n2. DISPLAY\n0. EXIT");
        printf("\nEnter your choice(0-2): ");
        scanf("%d", &choice);

        if (choice == 0)
        {
            break;
        }

        switch (choice)
        {
        case 1:
            printf("Data: ");
            scanf("%d", &data);
            root = insert(root, data);
            break;

        case 2:
            inorder(root);
            break;

        default:
            break;
        }
    }

    return 0;
}
```

OUTPUT:

```
1. INSERT
2. DISPLAY
0. EXIT
Enter your choice(0-2): 1
Data: 1

1. INSERT
2. DISPLAY
0. EXIT
Enter your choice(0-2): 1
Data: 2

1. INSERT
2. DISPLAY
0. EXIT
Enter your choice(0-2): 1
Data: 3

1. INSERT
2. DISPLAY
0. EXIT
Enter your choice(0-2): 1
Data: 4
```

```
1. INSERT
2. DISPLAY
0. EXIT
Enter your choice(0-2): 1
Data: 5

1. INSERT
2. DISPLAY
0. EXIT
Enter your choice(0-2): 2

Tree: 4 2 5 1 3
```

TIME COMPLEXITY ANALYSIS:

Let n be the number of nodes of the tree
Finding the inorder successor takes O(height) or O(logn) amount of time.Insertion in level order takes O(n) amount of time.Displaying in level order takes O(n) amount of time.

The overall Time complexity is O(n).