**NAME** :- Manish Shashikant Jadhav

**UID** :- 2023301005.

**BRANCH** :- Comps -B. **BRANCH:** B.

**EXPERIMENT 8:** Implement BFS & DFS traversal for graphs.

**SUBJECT** :- DS (DATA STRUCTURES).

**CODE** :-

#include "queue.c"

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

GraphRep \*init\_graph(int num\_vertices, bool is\_directed) {

    GraphRep \*graph = (GraphRep \*)malloc(sizeof(GraphRep));

    graph->nE = 0;

    graph->nV = num\_vertices;

    graph->is\_directed = is\_directed;

    graph->edges = (int \*\*)malloc(sizeof(int \*) \* num\_vertices \* num\_vertices);

    graph->distance = (int \*)malloc(sizeof(int) \* num\_vertices);

    graph->finish = (int \*)malloc(sizeof(int) \* num\_vertices);

    graph->predecessor = (Vertex \*)malloc(sizeof(Vertex) \* num\_vertices);

    graph->color = (Color \*)malloc(sizeof(Color) \* num\_vertices);

    for (int i = 0; i < num\_vertices; i++) {

        graph->edges[i] = (int \*)malloc(sizeof(int) \* num\_vertices);

    }

    for (int i = 0; i < num\_vertices; i++) {

        for (int j = 0; j < num\_vertices; j++) {

            graph->edges[i][j] = 0;

        }

    }

    return graph;

}

void insert\_edge(GraphRep \*graph, Edge e) {

    graph->nE++;

    graph->edges[e.u][e.v] = 1;

    if (!(graph->is\_directed)) {

        graph->edges[e.v][e.u] = 1;

        graph->nE++;

    }

}

void remove\_edge(GraphRep \*graph, Edge e) {

    graph->nE--;

    graph->edges[e.u][e.v] = 0;

    if (!(graph->is\_directed)) {

        graph->edges[e.v][e.u] = 0;

        graph->nE--;

    }

}

void traverse\_bfs(GraphRep \*graph, Vertex source) {

    graph->type = BFS;

    graph->source = source;

    for (int i = 0; i < graph->nV; i++) {

        graph->color[i] = WHITE;

        graph->distance[i] = -1;

        graph->predecessor[i] = -1;

    }

    graph->color[source] = GRAY;

    graph->distance[source] = 0;

    graph->predecessor[source] = -1;

    Queue \*q = initialize\_queue(graph->nV);

    enqueue(q, source);

    while (!isEmpty(q)) {

        Vertex u = dequeue(q);

        for (Vertex v = 0; v < graph->nV; v++) {

            if (graph->edges[u][v] == 1 && graph->color[v] == WHITE) {

                graph->color[v] = GRAY;

                graph->distance[v] = graph->distance[u] + 1;

                graph->predecessor[v] = u;

                enqueue(q, v);

            }

        }

        graph->color[u] = BLACK;

    }

    printf("\n> BFS Distances:");

    for (Vertex v = 0; v < graph->nV; v++) {

        printf("\n>> Vertex %d: Distance = %d", v, graph->distance[v]);

    }

}

void dfs(GraphRep \*graph, Vertex u, int \*time);

void traverse\_dfs(GraphRep \*graph, Vertex source) {

    graph->type = DFS;

    graph->source = source;

    for (Vertex v = 0; v < graph->nV; v++) {

        graph->color[v] = WHITE;

        graph->distance[v] = -1;

        graph->predecessor[v] = -1;

        graph->finish[v] = -1;

    }

    int time = 0;

    dfs(graph, source, &time);

    printf("\n> DFS Times:");

    for (Vertex v = 0; v < graph->nV; v++) {

        printf("\n>> Vertex %d: Discovery Time = %2d | Finish Time = %2d",

               v, graph->distance[v],

               graph->finish[v]);

    }

}

void dfs(GraphRep \*graph, Vertex u, int \*time) {

    graph->color[u] = GRAY;

    graph->distance[u] = ++(\*time);

    for (Vertex v = 0; v < graph->nV; v++) {

        if (graph->edges[u][v] == 1 && graph->color[v] == WHITE) {

            graph->predecessor[v] = u;

            dfs(graph, v, time);

        }

    }

    graph->color[u] = BLACK;

    graph->finish[u] = ++(\*time);

}

void display\_path(GraphRep \*graph, Vertex destination) {

    if (graph == NULL || destination < 0 || destination >= graph->nV) {

        printf("\n> Invalid input\n");

        return;

    }

    if (graph->type == BFS) {

        printf("\n\n> BFS Path Display\n>> Source: %d\n>> Destination: %d", graph->source, destination);

        if (graph->color[destination] != BLACK) {

            printf("\n> No path from source to destination.\n");

        } else {

            printf("\n> Shortest path from source to destination (BFS):\n");

            Vertex current = destination;

            while (current != -1) {

                printf("%d", current);

                current = graph->predecessor[current];

                if (current != -1) {

                    printf(" <- ");

                }

            }

            printf(" | Distance: %d\n", graph->distance[destination]);

        }

    } else {

        printf("\n\n> DFS Path Display\n>> Source: %d\n>> Destination: %d", graph->source, destination);

        if (graph->color[destination] != BLACK) {

            printf("\n> No path from source to destination.\n");

        } else {

            printf("\n> Fastest path from source to destination (BFS):\n");

            Vertex current = destination;

            while (current != -1) {

                printf("%d", current);

                current = graph->predecessor[current];

                if (current != -1) {

                    printf(" <- ");

                }

            }

            printf(" | Discovery Time: %2d | Finish Time: %2d\n",

                   graph->distance[destination],

                   graph->finish[destination]);

        }

    }

    printf("\n");

}

void display\_graph(GraphRep \*graph) {

    printf("\n\n Graph \n");

    for (int i = 0; i < graph->nV; i++) {

        printf("\t[%d]", i);

    }

    for (int i = 0; i < graph->nV; i++) {

        printf("\n[%d]", i);

        for (int j = 0; j < graph->nV; j++) {

            printf("\t %d", graph->edges[i][j]);

        }

    }

    printf("\n");

}

int main() {

    GraphRep \*g = init\_graph(5, true);

    display\_graph(g);

    Edge e;

    e.u = 0; e.v = 1; insert\_edge(g, e);

    e.u = 0; e.v = 2; insert\_edge(g, e);

    e.u = 1; e.v = 3; insert\_edge(g, e);

    e.u = 2; e.v = 4; insert\_edge(g, e);

    e.u = 3; e.v = 5; insert\_edge(g, e);

    display\_graph(g);

    traverse\_bfs(g, 0);

    display\_path(g, 4);

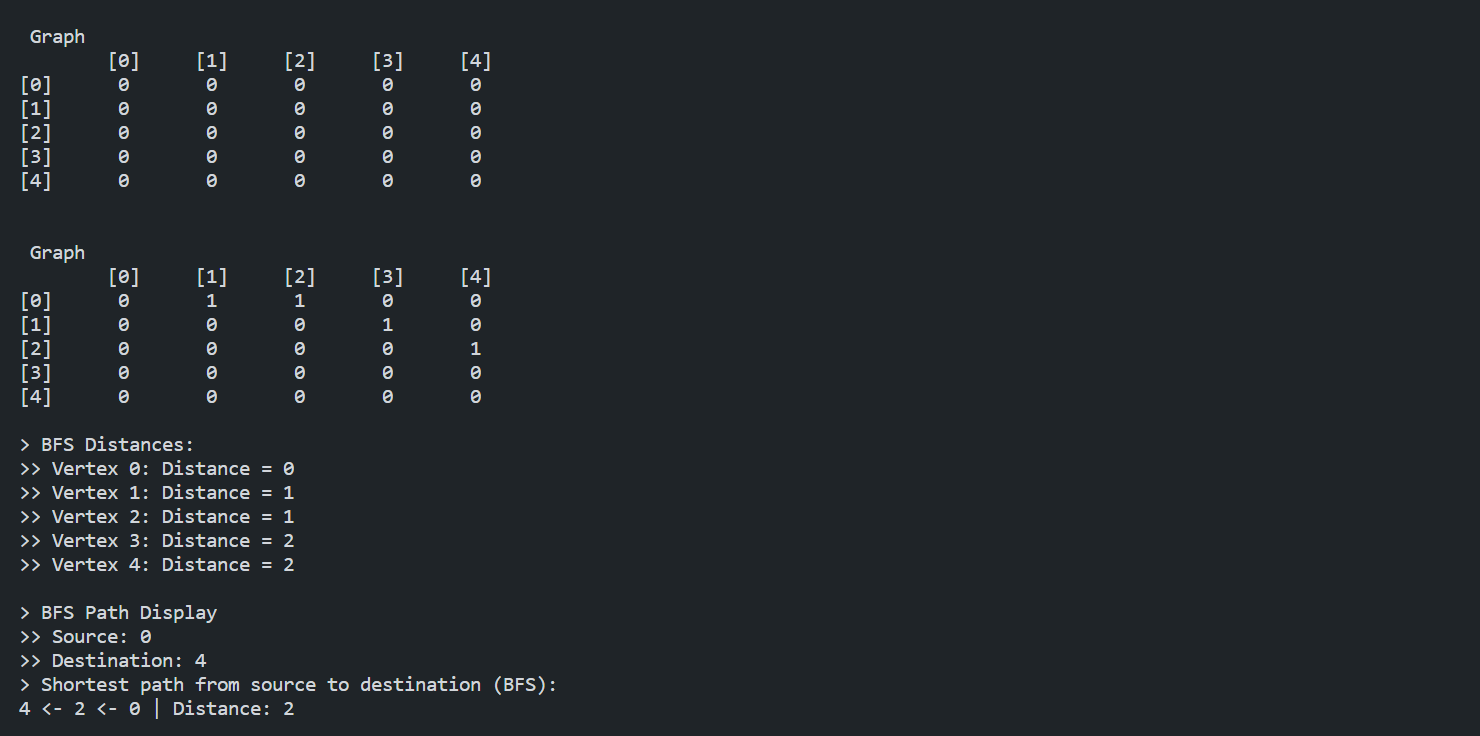
    traverse\_dfs(g, 0);

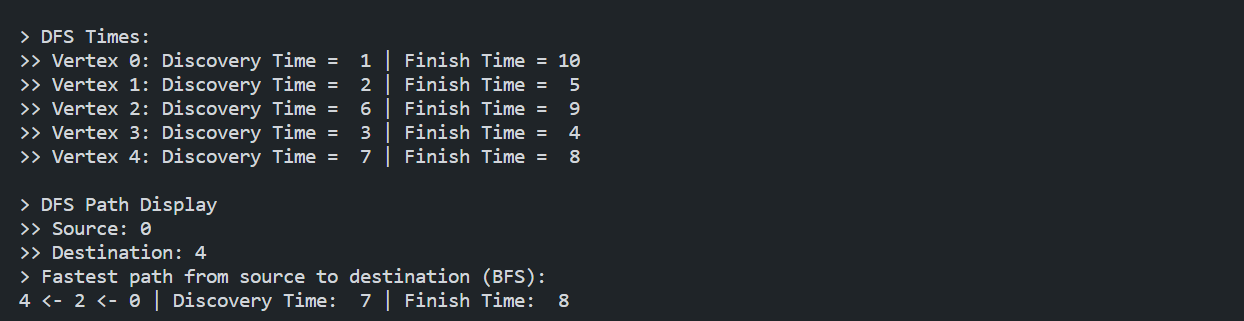
    display\_path(g, 4);

    return 0;

}

**Output:**

****

****

**Algorithm:**

1. **Graph Representation:**

The graph is represented using an adjacency matrix (edges), where edges[i][j] is 1 if there is an edge from vertex i to vertex j.Additional arrays (distance, predecessor, finish, color) are used to store traversal-related information.

The GraphRep structure contains information about the graph, such as the number of vertices (nV), whether it is directed (is\_directed), and the type of traversal being performed (type).

1. **Initialization:**

The init\_graph function initializes the graph with the specified number of vertices and whether it is directed or not. Memory is allocated for the necessary arrays.

1. **Edge Insertion and Removal:**

The insert\_edge and remove\_edge functions update the adjacency matrix to represent the presence or absence of edges between vertices.

1. **Breadth-First Search (BFS):**

The traverse\_bfs function performs BFS starting from a given source vertex. It uses a queue to keep track of vertices to visit.

During the traversal, the color of each vertex is updated to indicate whether it has been visited, and the distance from the source vertex is calculated.

After BFS, the program displays the distances from the source vertex to each other vertex.

1. **Depth-First Search (DFS):**

The traverse\_dfs function performs DFS starting from a given source vertex. It uses a recursive helper function (dfs).

During the traversal, the color of each vertex is updated to indicate whether it has been visited, and both discovery and finish times are recorded.

After DFS, the program displays the discovery and finish times for each vertex.

1. **Displaying Paths:**

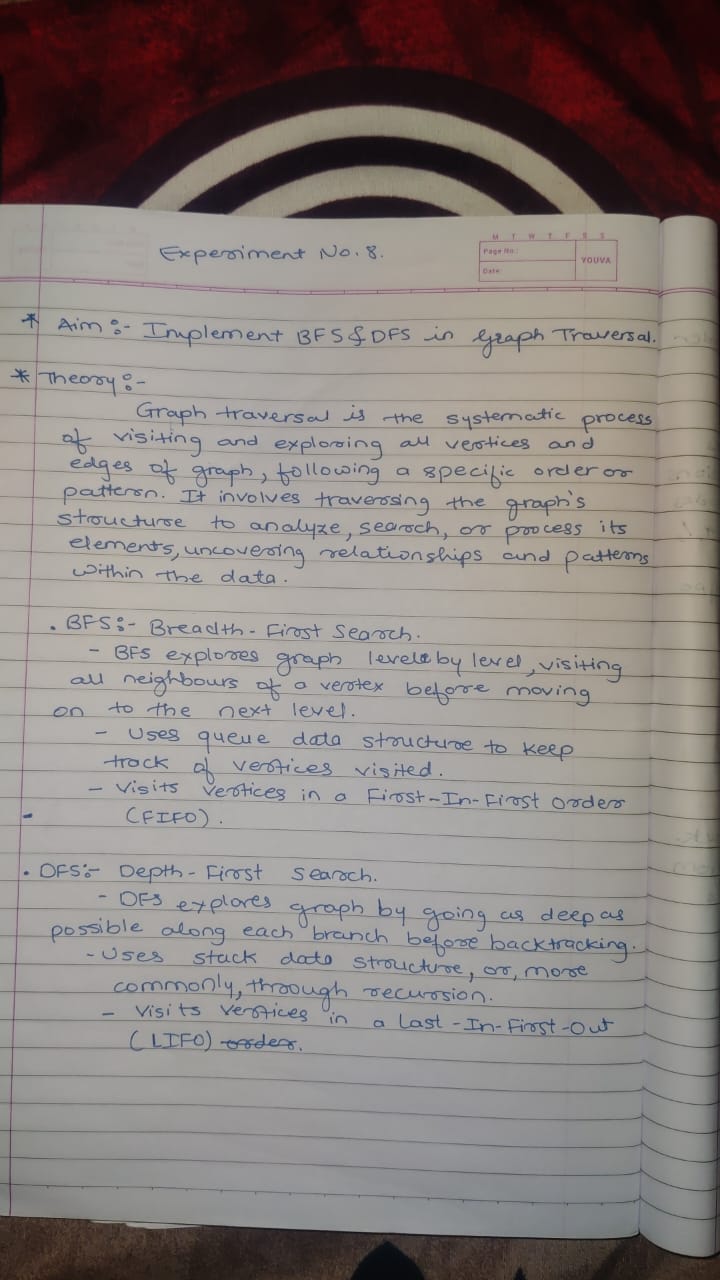
The display\_path function displays the path from the source vertex to a specified destination vertex. It considers whether BFS or DFS was performed and displays the path accordingly.

1. **Displaying the Graph:**

The display\_graph function displays the adjacency matrix representation of the graph.

1. **Main Function:**

The main function demonstrates the usage of the implemented functions. It initializes a graph, inserts edges, displays the graph, performs BFS and DFS traversals, and displays paths for a specific destination vertex.



**Conclusion:**

Hence, by completing this experiment I came to know about implement an expression tree.