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:

```
 Graph
       [0]    [1]    [2]    [3]    [4]
[0]     0      0      0      0      0
[1]     0      0      0      0      0
[2]     0      0      0      0      0
[3]     0      0      0      0      0
[4]     0      0      0      0      0


 Graph
       [0]    [1]    [2]    [3]    [4]
[0]     0      1      1      0      0
[1]     0      0      0      1      0
[2]     0      0      0      0      1
[3]     0      0      0      0      0
[4]     0      0      0      0      0

> BFS Distances:
>> Vertex 0: Distance = 0
>> Vertex 1: Distance = 1
>> Vertex 2: Distance = 1
>> Vertex 3: Distance = 2
>> Vertex 4: Distance = 2

> BFS Path Display
>> Source: 0
>> Destination: 4
> Shortest path from source to destination (BFS):
4 <- 2 <- 0 | Distance: 2
```

```
> DFS Times:
>> Vertex 0: Discovery Time =  1 | Finish Time = 10
>> Vertex 1: Discovery Time =  2 | Finish Time =  5
>> Vertex 2: Discovery Time =  6 | Finish Time =  9
>> Vertex 3: Discovery Time =  3 | Finish Time =  4
>> Vertex 4: Discovery Time =  7 | Finish Time =  8

> DFS Path Display
>> Source: 0
>> Destination: 4
> Fastest path from source to destination (BFS):
4 <- 2 <- 0 | Discovery Time:  7 | Finish Time:  8
```

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).

2. **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.

3. **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.

4. **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.

5. **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.

6. **Displaying Paths:**

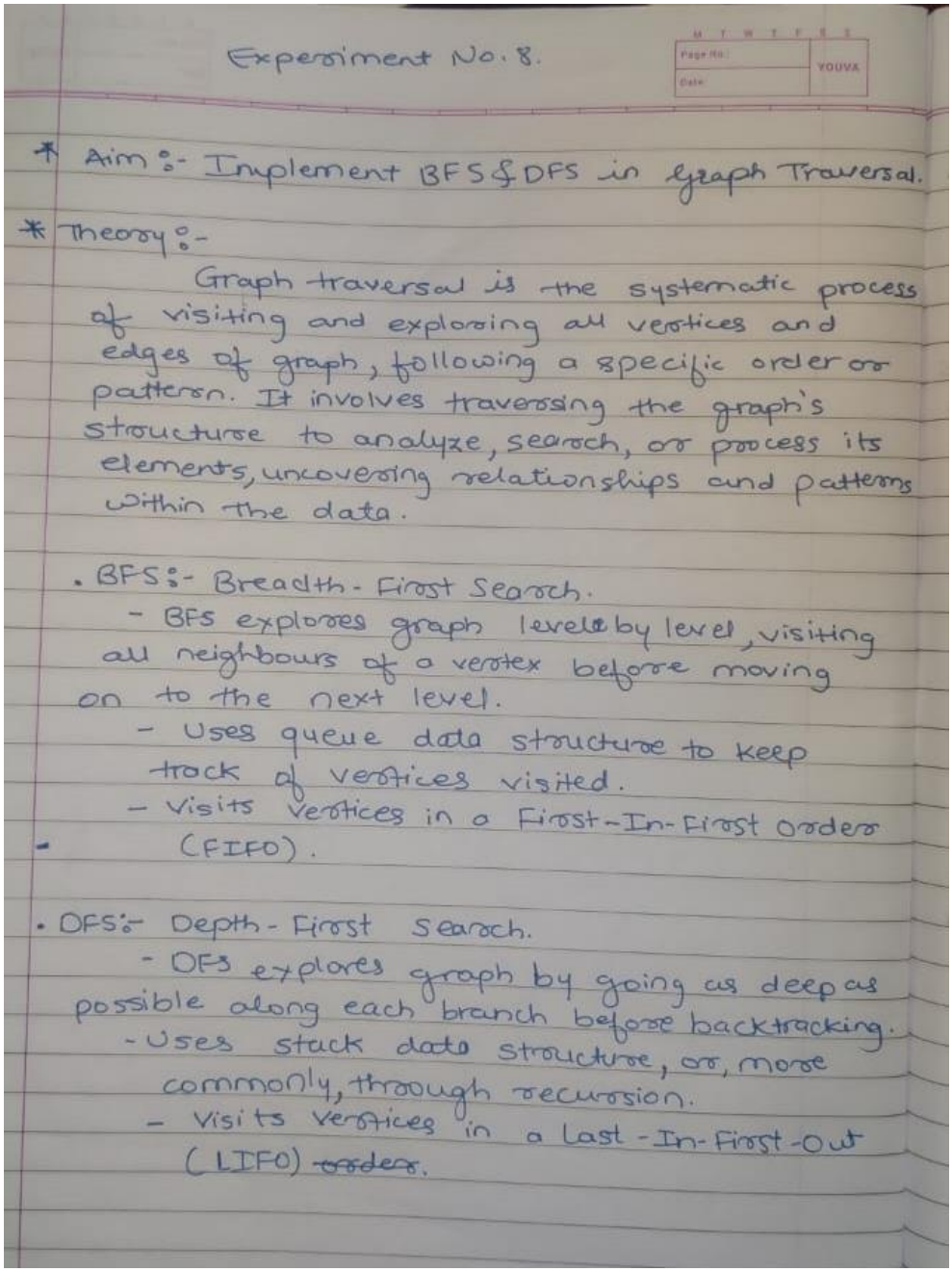
The display_path function displays the path from the source vertex to a specified destination vertex. It considers whether BFS or DFS was performed and displays the path accordingly.

7. **Displaying the Graph:**

The display_graph function displays the adjacency matrix representation of the graph.

8. **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.



Experiment No. 8.

* Aim :- Implement BFS & DFS in Graph Traversal.

* Theory :-

Graph traversal is the systematic process of visiting and exploring all vertices and edges of graph, following a specific order or pattern. It involves traversing the graph's structure to analyze, search, or process its elements, uncovering relationships and patterns within the data.

- BFS :- Breadth - First Search.
  - BFS explores graph levels by level, visiting all neighbours of a vertex before moving on to the next level.
  - Uses queue data structure to keep track of vertices visited.
  - Visits vertices in a First-In-First order (FIFO).

- DFS :- Depth - First Search.
  - DFS explores graph by going as deep as possible along each branch before backtracking.
  - Uses stack data structure, or, more commonly, through recursion.
  - Visits vertices in a Last-In-First-Out (LIFO) ~~order~~.

Conclusion:

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