Experiment No.10
Implementation of Graph traversal techniques - Depth First Search, Breadth First Search
Name: Sharvari Anand Bhondekar
Roll No: 06
Date of Performance:
Date of Submission:
Marks:
Sign:

Experiment No. 10: Depth First Search and Breath First Search

Aim : Implementation of DFS and BFS traversal of graph.

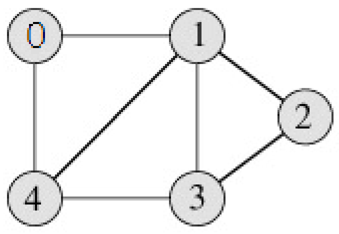
Objective:

1. Understand the Graph data structure and its basic operations.
2. Understand the method of representing a graph.
3. Understand the method of constructing the Graph ADT and defining its operations

Theory:

A graph is a collection of nodes or vertices, connected in pairs by lines referred to as edges. A graph can be directed or undirected.

One method of traversing through nodes is depth first search. Here we traverse from the starting node and proceed from top to bottom. At a moment we reach a dead end from where the further movement is not possible and we backtrack and then proceed according to left right order. A stack is used to keep track of a visited node which helps in backtracking.



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

DFS Traversal –0 1 2 3 4

Algorithm

Algorithm: DFS_LL(V)

Input: V is a starting vertex

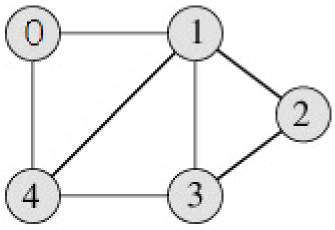
Output : A list VISIT giving order of visited vertices during traversal.

Description: linked structure of graph with gptr as pointer

```
1. if gptr = NULL then
       print "Graph is empty" exit
2. u=v
3. OPEN.PUSH(u)
4. while OPEN.TOP !=NULL do
       u=OPEN.POP()
```

```
            if search(VISIT,u) = FALSE then
                  INSERT_END(VISIT,u)
                  Ptr = gptr(u)
                  While ptr.LINK != NULL do
                        Vptr = ptr.LINK
                        OPEN.PUSH(vptr.LABEL)
                  End while
            End if
         End while
   5.  Return VISIT
   6.  Stop
```

BFS Traversal



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

BFS Traversal – 0 1 4 2 3

Algorithm

```
Algorithm: DFS()
i=0
count=1
visited[i]=1
print("Visited vertex  i")

repeat this till queue is empty or all nodes visited
repeat this for all nodes from first till last
if(g[i][j]!=0&&visited[j]!=1)
{
```

```
push(j)
}
i=pop()
print("Visited vertex  i")
visited[i]=1
count++
Algorithm: BFS()
i=0
count=1
visited[i]=1
print("Visited vertex  i")

repeat this till queue is empty or all nodes visited
repeat this for all nodes from first till last
if(g[i][j]!=0&&visited[j]!=1)
{
enqueue(j)
}

i=dequeue()
print("Visited vertex  i")
visited[i]=1
count++
```

Code:

1.BREADTH FIRST SEARCH

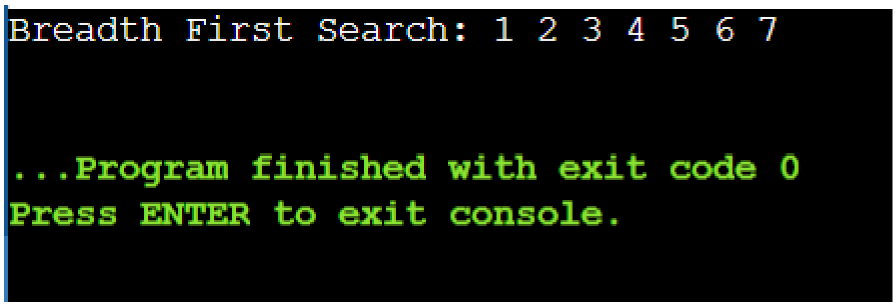
code:

```
#include <stdio.h>

#include <stdlib.h>
```

```
struct Node {
        int data;
        struct Node* left;
        struct Node* right;
};

struct Queue {
        struct Node** items;
        int front;
        int rear;
        int capacity;
};

struct Node* createNode(int data) {
        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->data = data;
        newNode->left = NULL;
        newNode->right = NULL;
        return newNode;
}

struct Queue* createQueue(int capacity) {
        struct Queue* q = (struct Queue*)malloc(sizeof(struct Queue));
        q->capacity = capacity;
        q->front = 0;
        q->rear = -1;

        q->items = (struct Node**)malloc(q->capacity * sizeof(struct Node*));

        return q;

}


int isFull(struct Queue* q) {

        return q->rear == q->capacity - 1;

}


int isEmpty(struct Queue* q) {

        return q->front > q->rear;

}


void enqueue(struct Queue* q, struct Node* node) {

        if (!isFull(q)) {

        q->items[++q->rear] = node;

        }

}


struct Node* dequeue(struct Queue* q) {

        if (!isEmpty(q)) {

        return q->items[q->front++];

        }

        return NULL;

}
```

```
void bfs(struct Node* root) {
        if (root == NULL) return;
        struct Queue* q = createQueue(100);
        enqueue(q, root);

        while (!isEmpty(q)) {
        struct Node* current = dequeue(q);
        printf("%d ", current->data);
        if (current->left != NULL) enqueue(q, current->left);
        if (current->right != NULL) enqueue(q, current->right);
        }
}

int main() {
        struct Node* root = createNode(1);
        root->left = createNode(2);
        root->right = createNode(3);
        root->left->left = createNode(4);
        root->left->right = createNode(5);
        root->right->left = createNode(6);
        root->right->right = createNode(7);

        printf("Breadth First Search: ");
        bfs(root);
        printf("\n");


        return 0;
}
```

Output:



```
Breadth First Search: 1 2 3 4 5 6 7


...Program finished with exit code 0
Press ENTER to exit console.
```

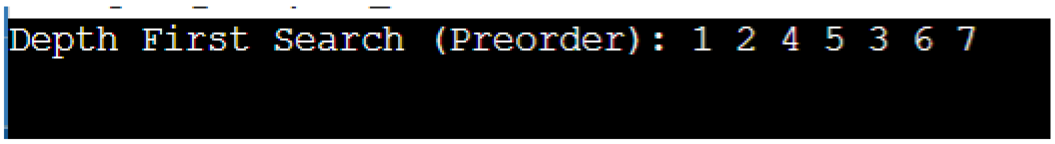
2.DEPTH FIRST SEARCH

Code:

```
#include <stdio.h>

#include <stdlib.h>



struct Node {

    int data;

    struct Node* left;

    struct Node* right;

};



// Create a new node

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

    newNode->data = data;

    newNode->left = NULL;

    newNode->right = NULL;

    return newNode;

}



// Stack structure

struct Stack {

    struct Node** items;

    int top;

    int capacity;

};



// Create a stack

struct Stack* createStack(int capacity) {

    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));

    stack->capacity = capacity;

    stack->top = -1;

    stack->items = (struct Node**)malloc(capacity * sizeof(struct Node*));

    return stack;

}
```

```
// Check if stack is full
int isFull(struct Stack* stack) {
    return stack->top == stack->capacity - 1;
}

// Check if stack is empty
int isEmpty(struct Stack* stack) {
    return stack->top == -1;
}

// Push a node onto the stack
void push(struct Stack* stack, struct Node* node) {
    if (!isFull(stack)) {
        stack->items[++stack->top] = node;
    }
}

// Pop a node from the stack
struct Node* pop(struct Stack* stack) {
    if (!isEmpty(stack)) {
        return stack->items[stack->top--];
    }
    return NULL;
}


// Preorder DFS traversal using a stack
void dfsPreorder(struct Node* root) {
    if (root == NULL) return;

    // Create a stack and push the root node
    struct Stack* stack = createStack(100);
    push(stack, root);

    // Traverse the tree
    while (!isEmpty(stack)) {
        // Pop the current node and process it
        struct Node* current = pop(stack);
        printf("%d ", current->data);

        // Push right child first so left child is processed next
        if (current->right != NULL) {
            push(stack, current->right);
        }
        // Push left child
        if (current->left != NULL) {
            push(stack, current->left);
        }
    }
}

int main() {
    // Creating the tree
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);
    root->right->left = createNode(6);
    root->right->right = createNode(7);

    // Perform Preorder DFS
    printf("Depth First Search (Preorder): ");
    dfsPreorder(root);
    printf("\n");
    return 0;
}
```

OUTPUT:



```
Depth First Search (Preorder): 1 2 4 5 3 6 7
```

Conclusion:

1.Write the graph representation used by your program and explain why you choose that.

➔ In the provided Breadth-First Search (BFS) program, the binary tree is represented using **adjacency relationships** via pointers. Each node in the tree has pointers to its left and right children, which allows for efficient traversal and representation of the tree structure.

Graph Representation

The graph representation used in the program can be depicted as follows:

Node Structure: Each node is defined by a structure that contains:

i. An integer data field to store the value of the node.

ii. Two pointers (left and right) to point to its child nodes.

Binary Tree:

```
 1

/   \

2    3

/ \    / \

4  5  6  7
```

Why This Representation Was Chosen:

Simplicity: Using pointers for child nodes is a straightforward and intuitive way to represent a binary tree. It allows for clear hierarchical relationships.

Dynamic Memory Usage: The use of dynamic memory allocation (malloc) allows for flexible tree sizes, accommodating any number of nodes without needing a predefined size, unlike static arrays.

Ease of Traversal: The pointer-based structure allows for easy traversal operations (like BFS and DFS) since you can directly access child nodes without needing to maintain additional data structures like arrays or lists for storing relationships.

Efficient Insertion and Deletion: Inserting and deleting nodes can be done efficiently without needing to shift elements, as would be necessary with an array-based implementation.

Space Efficiency: The representation minimizes wasted space. Nodes only use memory when they are needed, as opposed to a fixed-size array that might allocate more space than necessary.

2.Write the applications of BFS and DFS other than finding connected nodes and explain how it is attained?

➔ Applications of BFS

i. **Shortest Path in Unweighted Graphs**:

How It Works: BFS explores nodes layer by layer, meaning it visits all neighbors at the present depth prior to moving on to nodes at the next depth level. This ensures that when a node is reached for the first time, it has been reached via the shortest path. Thus, BFS can be used to find the shortest path between two nodes in an unweighted graph.

Example: In a social network, finding the shortest connection path between two users.

ii. Level Order Traversal of Trees:

How It Works: BFS can be used to traverse binary trees level by level. The nodes at each level are processed before moving on to the next level.

Example: Printing the nodes of a binary tree in level order.

iii.Finding All Nodes Within One Connected Component:

How It Works: When you need to find all nodes reachable from a starting node, BFS can explore all reachable nodes systematically and store them in a list.

Example: Finding all friends of a user in a social media platform starting from a single user.

Applications of DFS

i. **Topological Sorting**:

How It Works: DFS can be used to perform topological sorting on a directed acyclic graph (DAG). By performing a DFS and keeping track of the nodes in a stack, the nodes can be output in topological order by reversing the stack at the end.

Example: Scheduling tasks that have dependencies (e.g., course prerequisites).

ii. Cycle Detection:

How It Works: DFS can be utilized to detect cycles in both directed and undirected graphs. During traversal, if a node is visited that has already been encountered and is still in the current path (recursion stack), a cycle exists.

Example: Detecting deadlocks in resource allocation systems.

iii.Path Finding in Mazes:

How It Works: DFS can be used to find a path through a maze or labyrinth. By exploring as deep as possible before backtracking, DFS can help find one or more paths from the starting point to the exit.

Example: Solving puzzles like a maze or Sudoku.