**Academic Year: 2022-2023**

**Name – Prerna Sunil Jadhav** **SAP ID - 60004220127**

**Experiment No – 09**

## AIM: Implementation of BFS DFS

## Breadth First Search (BFS)

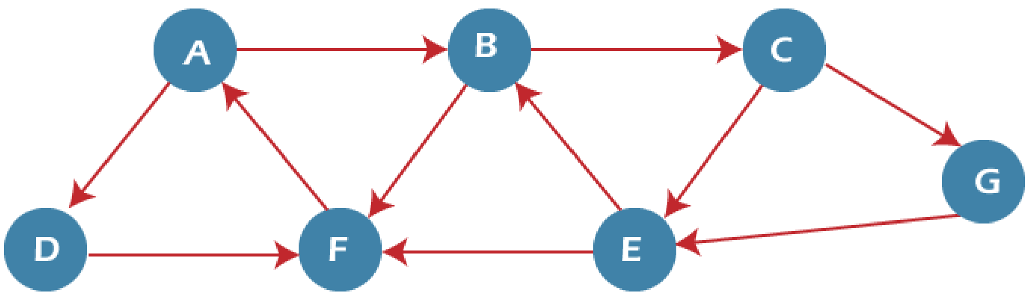
### Theory:

The breadth-first search (BFS) algorithm is used to search a tree or graph data structure for a node that meets a set of criteria. It starts at the tree's root or graph and searches/visits all nodes at the current depth level before moving on to the nodes at the next depth level. Breadth-first search can be used to solve many problems in graph theory.

### Algorithm:

```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until QUEUE is empty
Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).
Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and
set their STATUS = 2 (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

### Example:




**Adjacency Lists**

A : B, D
B : C, F
C : E, G
G : E
E : B, F
F : A
D : F

**BFS Traversal:** A, B, D, C, F, E

### Program:

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 40

struct queue {
  int items[SIZE];
```

```
  int front;
  int rear;
};

struct queue* createQueue();
void enqueue(struct queue* q, int);
int dequeue(struct queue* q);
void display(struct queue* q);
int isEmpty(struct queue* q);
void printQueue(struct queue* q);

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

struct node* createNode(int);

struct Graph {
  int numVertices;
  struct node** adjLists;
  int* visited;
};

// BFS algorithm
void bfs(struct Graph* graph, int startVertex) {
  struct queue* q = createQueue();

  graph->visited[startVertex] = 1;
  enqueue(q, startVertex);

  while (!isEmpty(q)) {
    printQueue(q);
    int currentVertex = dequeue(q);
    printf("Visited %d\n", currentVertex);

    struct node* temp = graph->adjLists[currentVertex];

    while (temp) {
      int adjVertex = temp->vertex;

      if (graph->visited[adjVertex] == 0) {
        graph->visited[adjVertex] = 1;
        enqueue(q, adjVertex);
      }
      temp = temp->next;
```

```
    }
  }
}

// Creating a node
struct node* createNode(int v) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
}

// Creating a graph
struct Graph* createGraph(int vertices) {
  struct Graph* graph = malloc(sizeof(struct Graph));
  graph->numVertices = vertices;

  graph->adjLists = malloc(vertices * sizeof(struct node*));
  graph->visited = malloc(vertices * sizeof(int));

  int i;
  for (i = 0; i < vertices; i++) {
    graph->adjLists[i] = NULL;
    graph->visited[i] = 0;
  }

  return graph;
}

// Add edge
void addEdge(struct Graph* graph, int src, int dest) {
  // Add edge from src to dest
  struct node* newNode = createNode(dest);
  newNode->next = graph->adjLists[src];
  graph->adjLists[src] = newNode;

  // Add edge from dest to src
  newNode = createNode(src);
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}

// Create a queue
struct queue* createQueue() {
  struct queue* q = malloc(sizeof(struct queue));
  q->front = -1;
```

```
  q->rear = -1;
  return q;
}

// Check if the queue is empty
int isEmpty(struct queue* q) {
  if (q->rear == -1)
    return 1;
  else
    return 0;
}

// Adding elements into queue
void enqueue(struct queue* q, int value) {
  if (q->rear == SIZE - 1)
    printf("\nQueue is Full!!");
  else {
    if (q->front == -1)
      q->front = 0;
    q->rear++;
    q->items[q->rear] = value;
  }
}

// Removing elements from queue
int dequeue(struct queue* q) {
  int item;
  if (isEmpty(q)) {
    printf("Queue is empty");
    item = -1;
  } else {

    item = q->items[q->front];
    q->front++;
    if (q->front > q->rear) {
      printf("Resetting queue ");
      q->front = q->rear = -1;
    }
  }
  return item;
}
// Print the queue
void printQueue(struct queue* q) {
  int i = q->front;

  if (isEmpty(q)) {
```

```
    printf("Queue is empty");
  } else {
    printf("\nQueue contains \n");
    for (i = q->front; i < q->rear + 1; i++) {
      printf("%d ", q->items[i]);
    }
  }
}
int main() {
  struct Graph* graph = createGraph(6);
  addEdge(graph, 0, 1);
  addEdge(graph, 0, 2);
  addEdge(graph, 1, 2);
  addEdge(graph, 1, 4);
  addEdge(graph, 1, 3);
  addEdge(graph, 2, 4);

  addEdge(graph, 3, 4);

  bfs(graph, 0);

  return 0;
}
```

**OUTPUT:**

```
Queue contains
0 Resetting queue Visited 0

Queue contains
2 1 Visited 2

Queue contains
1 4 Visited 1

Queue contains
4 3 Visited 4

Queue contains
3 Resetting queue Visited 3
```

**Conclusion**:

- Time complexity of BFS depends upon the data structure used to represent the graph. The time complexity of BFS algorithm is O(V+E), since in the worst case, BFS algorithm explores every node and edge. In a graph, the number of vertices is O(V), whereas the number of edges is O(E).
- The space complexity of BFS can be expressed as O(V), where V is the number of vertices.

## Depth-first search (DFS)

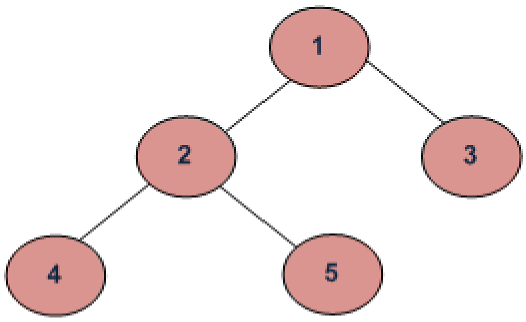
### Theory:

DFS (Depth-first search) is a technique used for traversing trees or graphs. Here backtracking is used for traversal. In this traversal first, the deepest node is visited and then backtracks to its parent node if no sibling of that node exists

### Algorithm:

```
A standard DFS implementation puts each vertex of the graph into one of two categories:
1. Visited
2. Not Visited
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.
The DFS algorithm works as follows:
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.
```

### Example:




Therefore, the Depth First Traversals of this Tree will be:

Inorder: 4 2 5 1 3
Preorder: 1 2 4 5 3
Postorder: 4 5 2 3 1

### Program:

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int vertex;
    struct node *next;
};
struct node *createNode(int v);
```

```
struct Graph
{
    int numVertices;
    int *visited;
    // We need int** to store a two dimensional array.
    // Similary, we need struct node** to store an array of Linked lists
    struct node **adjLists;
};

// DFS algo
void DFS(struct Graph *graph, int vertex)
{
    struct node *adjList = graph->adjLists[vertex];
    struct node *temp = adjList;

    graph->visited[vertex] = 1;
    printf("Visited %d \n", vertex);

    while (temp != NULL)
    {
        int connectedVertex = temp->vertex;

        if (graph->visited[connectedVertex] == 0)
        {
            DFS(graph, connectedVertex);
        }
        temp = temp->next;
    }
}
// Create a node
struct node *createNode(int v)
{
    struct node *newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}
// Create graph
struct Graph *createGraph(int vertices)
{
    struct Graph *graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;
    graph->adjLists = malloc(vertices * sizeof(struct node *));
    graph->visited = malloc(vertices * sizeof(int));
    int i;
    for (i = 0; i < vertices; i++)
```

```
    {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }
    return graph;
}
// Add edge
void addEdge(struct Graph *graph, int src, int dest)
{
    // Add edge from src to dest
    struct node *newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Add edge from dest to src
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Print the graph
void printGraph(struct Graph *graph)
{
    int v;
    for (v = 0; v < graph->numVertices; v++)
    {
        struct node *temp = graph->adjLists[v];
        printf("\n Adjacency list of vertex %d\n ", v);
        while (temp)
        {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }
}

int main()
{
    printf("Prerna Sunil Jadhav - 60004220127\n");

    struct Graph *graph = createGraph(4);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 2, 3);
```

```
    printGraph(graph);
    DFS(graph, 2);
    return 0;
}
```

**Output:**

```
Prerna Sunil Jadhav - 60004220127

 Adjacency list of vertex 0
 2 -> 1 ->

 Adjacency list of vertex 1
 2 -> 0 ->

 Adjacency list of vertex 2
 3 -> 1 -> 0 ->

 Adjacency list of vertex 3
 2 ->
Visited 2
Visited 3
Visited 1
Visited 0
```

**Conclusion:**

**Time Complexity: O(N)**

**Auxiliary Space: O(log N)**