## BFS (Matrix)

```
#include <iostream>
#include <cstring> // For memset

using namespace std;

#define MAX_NODES 100 // Maximum number of nodes

// Graph class for BFS traversal
class Graph {
    int vertices;                // Number of vertices
    int adjMatrix[MAX_NODES][MAX_NODES]; // Adjacency matrix

public:
    // Constructor
    Graph(int V) {
        vertices = V;
        memset(adjMatrix, 0, sizeof(adjMatrix)); // Initialize matrix with 0
    }

    // Function to add an edge to the graph
    void addEdge(int u, int v) {
        adjMatrix[u][v] = 1; // Set the connection in the adjacency matrix
        adjMatrix[v][u] = 1; // Undirected graph
    }
```

```
    // BFS function
    void BFS(int start) {
        int visited[MAX_NODES] = {0}; // Visited array to mark visited nodes
        int queue[MAX_NODES];        // Manual queue
        int front = 0, rear = 0;     // Front and rear pointers

        // Start BFS
        visited[start] = 1; // Mark the starting node as visited
        queue[rear++] = start; // Enqueue the starting node

        cout << "BFS starting from vertex " << start << ": ";

        while (front != rear) { // While queue is not empty
            int current = queue[front++]; // Dequeue the front node
            cout << current << " ";

            // Check all neighbors
            for (int i = 0; i < vertices; i++) {
                if (adjMatrix[current][i] == 1 && !visited[i]) { // If there's a connection and not visited
                    visited[i] = 1;      // Mark as visited
                    queue[rear++] = i;   // Enqueue the neighbor
                }
            }
        }
        cout << endl;
    }
};
```

```
int main() {
    Graph g(6); // Create a graph with 6 vertices (0 to 5)

    // Add edges to the graph
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
    g.addEdge(2, 4);
    g.addEdge(3, 5);

    // Perform BFS starting from vertex 0
    g.BFS(0);

    return 0;
}
```

**BFS (Linked List)**

```
#include <iostream>
#include <cstring> // For memset

using namespace std;

#define MAX_NODES 100 // Maximum number of nodes

class Graph {
```

```
    int vertices;                   // Number of vertices
    int adjList[MAX_NODES][MAX_NODES]; // Adjacency list
    int listSize[MAX_NODES];          // Tracks number of neighbors for each node

public:
    // Constructor
    Graph(int V) {
        vertices = V;
        memset(adjList, -1, sizeof(adjList)); // Initialize adjacency list to -1
        memset(listSize, 0, sizeof(listSize));
    }

    // Add an edge to the graph
    void addEdge(int u, int v) {
        adjList[u][listSize[u]++] = v; // Add v to u's adjacency list
        adjList[v][listSize[v]++] = u; // For undirected graph, add u to v's list
    }

    // BFS function
    void BFS(int start) {
        int visited[MAX_NODES] = {0}; // Visited array
        int queue[MAX_NODES];        // Manual queue implementation
        int front = 0, rear = 0;     // Front and rear for queue

        // Start BFS from the starting node
        visited[start] = 1;      // Mark start as visited
        queue[rear++] = start;   // Enqueue start
```

```
        cout << "BFS starting from vertex " << start << ": ";

        while (front != rear) {   // While queue is not empty
            int node = queue[front++]; // Dequeue node
            cout << node << " ";

            // Traverse all neighbors of the current node
            for (int i = 0; i < listSize[node]; i++) {
                int neighbor = adjList[node][i];
                if (!visited[neighbor]) { // If neighbor is unvisited
                    visited[neighbor] = 1;   // Mark as visited
                    queue[rear++] = neighbor; // Enqueue neighbor
                }
            }
        }
        cout << endl;
    }
};

int main() {
    Graph g(6); // Create a graph with 6 vertices (0 to 5)

    // Add edges to the graph
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
```

```
    g.addEdge(1, 4);
    g.addEdge(2, 4);
    g.addEdge(3, 5);

    // Perform BFS starting from vertex 0
    g.BFS(0);

    return 0;
}
```

**DFS (Matrix)**

```
#include <iostream>
#include <cstring> // For memset

using namespace std;

#define MAX_NODES 100 // Maximum number of nodes

class Graph {
    int vertices;               // Number of vertices
    int adjMatrix[MAX_NODES][MAX_NODES]; // Adjacency matrix

public:
    // Constructor
    Graph(int V) {
        vertices = V;
        memset(adjMatrix, 0, sizeof(adjMatrix)); // Initialize adjacency matrix with 0
```

```
}

// Function to add an edge to the graph
void addEdge(int u, int v) {
    adjMatrix[u][v] = 1; // Add edge u -> v
    adjMatrix[v][u] = 1; // For undirected graph, add edge v -> u
}

// DFS function
void DFSUtil(int node, int visited[]) {
    visited[node] = 1; // Mark current node as visited
    cout << node << " ";

    // Traverse all neighbors
    for (int i = 0; i < vertices; i++) {
        if (adjMatrix[node][i] == 1 && !visited[i]) { // If there's an edge and not visited
            DFSUtil(i, visited); // Recur for neighbor
        }
    }
}

// Function to perform DFS traversal
void DFS(int start) {
    int visited[MAX_NODES] = {0}; // Visited array to track visited nodes
    cout << "DFS starting from vertex " << start << ": ";
    DFSUtil(start, visited); // Call the utility function
    cout << endl;
```

```
    }
};

int main() {
    Graph g(6); // Create a graph with 6 vertices (0 to 5)

    // Add edges to the graph
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
    g.addEdge(2, 4);
    g.addEdge(3, 5);

    // Perform DFS starting from vertex 0
    g.DFS(0);

    return 0;
}
```

**DFS (Linked List)**

```
#include <iostream>
#include <cstring> // For memset

using namespace std;

#define MAX_NODES 100
```

```
// Node structure for adjacency list
struct Node {
    int vertex;
    Node* next;
};

// Graph class
class Graph {
    int vertices;          // Number of vertices
    Node* adjList[MAX_NODES]; // Array of pointers to adjacency lists

public:
    // Constructor
    Graph(int V) {
        vertices = V;
        memset(adjList, 0, sizeof(adjList)); // Initialize adjacency list
    }

    // Function to create a new node
    Node* createNode(int v) {
        Node* newNode = new Node;
        newNode->vertex = v;
        newNode->next = nullptr;
        return newNode;
    }
```

```
// Function to add an edge to the graph
void addEdge(int u, int v) {
    // Add v to u's adjacency list
    Node* newNode = createNode(v);
    newNode->next = adjList[u];
    adjList[u] = newNode;

    // Add u to v's adjacency list (undirected graph)
    newNode = createNode(u);
    newNode->next = adjList[v];
    adjList[v] = newNode;
}

// DFS function using a manual stack
void DFS(int start) {
    int visited[MAX_NODES] = {0}; // Visited array
    int stack[MAX_NODES];        // Manual stack
    int top = -1;                // Stack top pointer

    // Push the starting vertex onto the stack
    stack[++top] = start;

    cout << "DFS starting from vertex " << start << ": ";

    while (top != -1) { // While stack is not empty
        int node = stack[top--]; // Pop a node
```

```
            // If the node is not visited, visit it
            if (!visited[node]) {
                cout << node << " ";
                visited[node] = 1;
            }

            // Traverse all neighbors (linked list)
            Node* temp = adjList[node];
            while (temp != nullptr) {
                if (!visited[temp->vertex]) {
                    stack[++top] = temp->vertex; // Push unvisited neighbors onto the stack
                }
                temp = temp->next;
            }
        }
        cout << endl;
    }
};

int main() {
    Graph g(6); // Create a graph with 6 vertices (0 to 5)

    // Add edges
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
```

```
    g.addEdge(2, 4);
    g.addEdge(3, 5);

    // Perform DFS starting from vertex 0
    g.DFS(0);

    return 0;
}
```