A purple text on a black background

AI-generated content may be incorrect.

**LAB 13 - DFS and BFS**

Name: Omar Afef Mohammed

Roll number: SU92-BSSEM-S24-086

Section: 3A

Teacher: Mr. Rasikh Ali

Subject: DSA LAB

Date: May 4, 2025

Tasks:

1. Insert and Traverse for DFS in tree

**ANSWER**

**CODE**

#include <iostream>

using namespace std;

class Node{

public:

int data;

Node\* right;

Node\* left;

Node(int d){

data=d;

right=NULL;

left=NULL;

}

};

class DSA{

public:

Node\* head;

DSA(){

head=NULL;

}

void insert(int d){

if(head==NULL){

head=new Node(d);

return;

}

Node\* temp=head;

while(true){

if(d>temp->data){

if(temp->right==NULL){

temp->right=new Node(d);

return;

}

temp=temp->right;

}

else if(d<temp->data){

if(temp->left==NULL){

temp->left=new Node(d);

return;

}

temp=temp->left;

}

else{

cout<<temp->data<<"="<<d<<endl;

return;

}

}

}

void display(Node\* temp){

cout<<temp->data<<" ";

}

void Travers() {

TraversHelper(head);

}

void TraversHelper(Node\* temp) {

if (temp == NULL) return;

display(temp);

TraversHelper(temp->left);

TraversHelper(temp->right);

}

};

int main(){

DSA me;

me.insert(1);

me.insert(2);

me.insert(3);

me.insert(4);

me.insert(4);

me.insert(5);

me.insert(6);

me.insert(6);

me.Travers();

}

**How it works (step by step):**

Node class:

Same as before — stores data, left, and right.

insert(d):

Same Binary Search Tree (BST) insertion —

* Goes left if smaller
* Goes right if larger
* If equal, prints "x = x" and skips insertion (no duplicates)

display():

Prints the data value of one node only (used inside traversal).

Travers():

Starts preorder traversal from the root:

* Visit current node
* Visit left subtree
* Visit right subtree

TraversHelper(Node\* temp):

Recursive function that performs the preorder traversal.

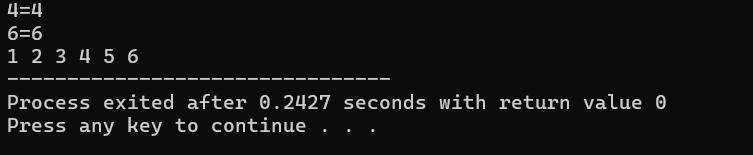
main():

* Creates the tree
* Inserts values (some duplicates, which get skipped)
* Calls Travers() to print the tree nodes in preorder

**Why it works:**

* BST Rules: Maintains correct order — left < root < right
* No balancing: It's just a regular BST, no AVL rotations
* Preorder Traversal: Displays data in root-left-right order
* Duplicate handling: Skips adding duplicates and notifies you

**OUTPUT**

****

2. Insert and Traverse for DFS in graph

**ANSWER**

**CODE**

#include <iostream>

using namespace std;

class GraphNode;

// Node in the adjacency list

class ListNode {

public:

GraphNode\* neighbor;

ListNode\* next;

ListNode(GraphNode\* n) {

neighbor = n;

next = NULL;

}

};

// Graph node

class GraphNode {

public:

int data;

ListNode\* adjList;

bool visited;

GraphNode(int d) {

data = d;

adjList = NULL;

visited = false;

}

void addNeighbor(GraphNode\* n) {

ListNode\* newNode = new ListNode(n);

newNode->next = adjList;

adjList = newNode;

}

void printNeighbors() {

cout << "Node " << data << " connected to: ";

ListNode\* temp = adjList;

while (temp != NULL) {

cout << temp->neighbor->data << " ";

temp = temp->next;

}

cout << endl;

}

};

// DFS function

void dfs(GraphNode\* node) {

if (node == NULL || node->visited) return;

node->visited = true;

cout << node->data << " ";

ListNode\* temp = node->adjList;

while (temp != NULL) {

dfs(temp->neighbor);

temp = temp->next;

}

}

int main() {

// Create nodes

GraphNode\* A = new GraphNode(1);

GraphNode\* B = new GraphNode(2);

GraphNode\* C = new GraphNode(3);

GraphNode\* D = new GraphNode(4);

GraphNode\* G = new GraphNode(5);

GraphNode\* F = new GraphNode(6);

GraphNode\* E = new GraphNode(7);

// Connect nodes (undirected graph example)

A->addNeighbor(B);

A->addNeighbor(C);

B->addNeighbor(A);

B->addNeighbor(D);

C->addNeighbor(A);

D->addNeighbor(B);

D->addNeighbor(G);

C->addNeighbor(F);

F->addNeighbor(E);

// Print graph

A->printNeighbors();

B->printNeighbors();

C->printNeighbors();

D->printNeighbors();

G->printNeighbors();

F->printNeighbors();

E->printNeighbors();

// Perform DFS from node A

cout << "\nDFS traversal: ";

dfs(A);

cout << endl;

return 0;

}

**How it works (step by step):**

ListNode class (adjacency list node):

* Stores neighbor (pointer to another GraphNode)
* Stores next (next neighbor in the list)
* Acts like a linked list node.

GraphNode class (graph vertex):

* data: The value of the node
* adjList: Head of the linked list of neighbors (type ListNode\*)
* visited: For DFS (to avoid visiting twice)

addNeighbor(GraphNode\* n)

* Adds a new neighbor to the front of the adjacency list.

printNeighbors()

* Prints the current node and its connected nodes.

dfs(GraphNode\* node):

* Classic Depth-First Search:
  + Mark node as visited
  + Print node's data
  + Recursively visit unvisited neighbors
* Uses recursion ⇒ DFS
* Uses visited flag to avoid cycles/infinite loops

main():

* Creates 7 graph nodes: A to E, F, G
* Adds edges manually (some are undirected)
  + Example: A-B, A-C, B-D, C-F, D-G, F-E
* Prints the adjacency list of all nodes
* Starts DFS traversal from node A

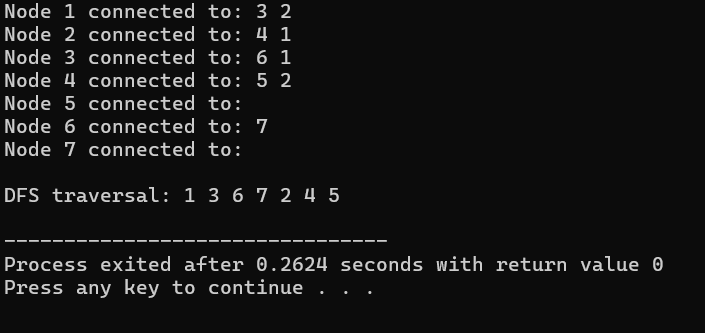
Output behavior:

* Based on how addNeighbor works (adds to front), DFS order depends on insertion order.
* Example DFS path:  
  A → C → F → E → B → D → G (depends on how neighbors were added)

**Why it works:**

* Graph is built with adjacency lists (efficient for sparse graphs).
* DFS uses recursion and a visited flag to explore deeply.
* Graph traversal shows connections and avoids re-visiting nodes.

**OUTPUT**

****

3. Insert and Traverse for BFS in tree

**ANSWER**

**CODE**

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* right;

Node\* left;

Node(int d) {

data = d;

right = NULL;

left = NULL;

}

};

class DSA {

public:

Node\* head;

DSA() {

head = NULL;

}

void insert(int d) {

if (head == NULL) {

head = new Node(d);

return;

}

Node\* temp = head;

while (true) {

if (d > temp->data) {

if (temp->right == NULL) {

temp->right = new Node(d);

return;

}

temp = temp->right;

} else if (d < temp->data) {

if (temp->left == NULL) {

temp->left = new Node(d);

return;

}

temp = temp->left;

} else {

cout << temp->data << " = " << d << endl;

return;

}

}

}

void display(Node\* temp) {

cout << temp->data << " ";

}

void Travers() {

bfsTraversal(head);

}

void bfsTraversal(Node\* root) {

if (root == NULL) return;

const int MAX\_SIZE = 100;

Node\* queue[MAX\_SIZE];

int front = 0, rear = 0;

queue[rear++] = root;

while (front < rear) {

Node\* current = queue[front++];

display(current);

if (current->left != NULL)

queue[rear++] = current->left;

if (current->right != NULL)

queue[rear++] = current->right;

}

}

};

int main() {

DSA me;

me.insert(6);

me.insert(1);

me.insert(2);

me.insert(3);

me.insert(4);

me.insert(4); // duplicate

me.insert(5);

me.insert(6); // duplicate

me.Travers(); // BFS Traversal

}

**How it works (step by step):**

ListNode class:

* Holds a pointer to a neighbor (another GraphNode)
* Holds a pointer to the next node in the adjacency list

GraphNode class:

* data: the value of the node
* adjList: points to the head of the linked list of neighbors
* visited: flag used to avoid revisiting during DFS

addNeighbor(GraphNode\* n):

* Creates a new ListNode pointing to n
* Inserts it at the beginning of the adjacency list

printNeighbors():

* Iterates through the adjacency list
* Prints all connected neighbors for this node

dfs(GraphNode\* node):

* Base case: if node is null or already visited, return
* Marks node as visited
* Prints node data
* Recursively explores all connected, unvisited neighbors

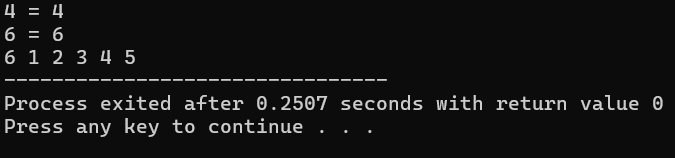
main():

* Creates 7 graph nodes (A to G)
* Connects nodes by adding neighbors (undirected style — both directions)
* Prints adjacency lists of each node
* Performs DFS traversal starting from node A

**Why it works:**

* DFS (Depth-First Search) uses recursion to go as deep as possible before backtracking.
* visited ensures no node is visited twice (avoids infinite loops in cycles).
* The adjacency list structure models connections efficiently.
* DFS correctly explores connected components by visiting all reachable neighbors from a starting node.

**OUTPUT**

****

4. Insert and Traverse for BFS in graph

**ANSWER**

**CODE**

#include <iostream>

using namespace std;

class GraphNode;

// Node in the adjacency list

class ListNode {

public:

GraphNode\* neighbor;

ListNode\* next;

ListNode(GraphNode\* n) {

neighbor = n;

next = NULL;

}

};

// Graph node

class GraphNode {

public:

int data;

ListNode\* adjList;

bool visited;

GraphNode(int d) {

data = d;

adjList = NULL;

visited = false;

}

void addNeighbor(GraphNode\* n) {

ListNode\* newNode = new ListNode(n);

newNode->next = adjList;

adjList = newNode;

}

void printNeighbors() {

cout << "Node " << data << " connected to: ";

ListNode\* temp = adjList;

while (temp != NULL) {

cout << temp->neighbor->data << " ";

temp = temp->next;

}

cout << endl;

}

};

// Simple array-based queue

class NodeQueue {

GraphNode\*\* arr;

int front, rear, size;

public:

NodeQueue(int capacity) {

arr = new GraphNode\*[capacity];

front = 0;

rear = 0;

size = capacity;

}

bool isEmpty() {

return front == rear;

}

void enqueue(GraphNode\* node) {

if (rear < size) {

arr[rear++] = node;

}

}

GraphNode\* dequeue() {

if (!isEmpty()) {

return arr[front++];

}

return NULL;

}

~NodeQueue() {

delete[] arr;

}

};

// BFS function

void bfs(GraphNode\* start) {

if (start == NULL) return;

const int MAX\_NODES = 100; // You can adjust based on expected graph size

NodeQueue q(MAX\_NODES);

start->visited = true;

q.enqueue(start);

while (!q.isEmpty()) {

GraphNode\* current = q.dequeue();

cout << current->data << " ";

ListNode\* temp = current->adjList;

while (temp != NULL) {

if (!temp->neighbor->visited) {

temp->neighbor->visited = true;

q.enqueue(temp->neighbor);

}

temp = temp->next;

}

}

}

int main() {

// Create nodes

GraphNode\* A = new GraphNode(1);

GraphNode\* B = new GraphNode(2);

GraphNode\* C = new GraphNode(3);

GraphNode\* D = new GraphNode(4);

GraphNode\* G = new GraphNode(5);

GraphNode\* F = new GraphNode(6);

GraphNode\* E = new GraphNode(7);

// Connect nodes (undirected graph example)

A->addNeighbor(B);

A->addNeighbor(C);

B->addNeighbor(A);

B->addNeighbor(D);

C->addNeighbor(A);

D->addNeighbor(B);

D->addNeighbor(G);

C->addNeighbor(F);

F->addNeighbor(E);

// Print graph

A->printNeighbors();

B->printNeighbors();

C->printNeighbors();

D->printNeighbors();

G->printNeighbors();

F->printNeighbors();

E->printNeighbors();

// Perform BFS from node A

cout << "\nBFS traversal: ";

bfs(A);

cout << endl;

return 0;

}

**How it works (step by step):**

ListNode class:

* Holds a pointer to a neighbor (another GraphNode)
* Holds a pointer to the next node in the adjacency list

GraphNode class:

* data: the value of the node
* adjList: points to the head of the linked list of neighbors
* visited: flag used to avoid revisiting during BFS

addNeighbor(GraphNode\* n):

* Creates a new ListNode pointing to n
* Inserts it at the beginning of the adjacency list

printNeighbors():

* Iterates through the adjacency list
* Prints all connected neighbors for this node

NodeQueue class (simple queue for BFS):

* Holds an array arr of pointers to GraphNode objects
* Manages queue operations (enqueue, dequeue, check if empty)

enqueue(GraphNode\* node):

* Adds the node to the queue

dequeue():

* Removes and returns the node at the front of the queue

isEmpty():

* Checks if the queue is empty

bfs(GraphNode\* start):

* Base case: if start is null, return
* Marks the starting node as visited and enqueues it
* While the queue is not empty:
  + Dequeues a node and prints its data
  + Visits all unvisited neighbors and enqueues them

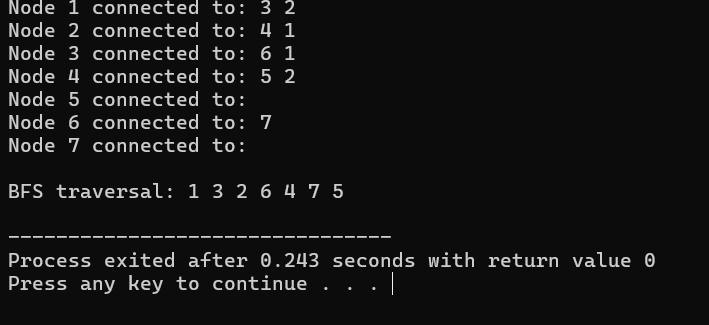
main():

* Creates 7 graph nodes (A to G)
* Connects nodes by adding neighbors (undirected style — both directions)
* Prints adjacency lists of each node
* Performs BFS traversal starting from node A

**Why it works:**

* BFS (Breadth-First Search) uses a queue to explore all nodes level by level.
* Each node is visited once, ensuring no cycles or re-visits.
* The visited flag prevents revisiting nodes.
* BFS is ideal for finding the shortest path in an unweighted graph, as it explores all nodes at the current level before moving to the next level.

**OUTPUT**

****