Codekata Report:

Name: Deepu Pandey

Email: deepu.24scse1011405@galgotiasuniversity.ac.in

Specialization: School of Computer Science & Engineering

Completion Year: 2028-3rd Sem

Section: Section-7

1. Problem Title: Escaping the MazeProblem StatementDescription:

You are given a maze represented by a 2D grid where cells contain integers. The goal is to find the minimum number of steps required to reach the exit of the maze starting from a given entry point. The maze has the following properties:

Cells that basically have the value 0 are walls and cannot be traversed.Cells with any positive integer n represent open paths. You can move from a cell to any other cell that is exactly n steps away either vertically or horizontally.

Function Signature:int minimumStepsToExit(int n, int m, int maze[n][m], int startX, int startY, int endX, int endY);

Input:An integer n representing the number of rows in the maze.An integer m is representing the number of columns in the maze.A 2D array maze of size n x m representing the maze.Two integers startX and startY are representing the starting cell coordinates (0-based index).Two integers endX and endY representing the exit cell coordinates (0-based index).

Output:You have to basically return the minimum number of steps which are required to reach the exit cell from the start cell. If it is not possible to reach the exit, return -1.

Constraints:1 <= n, m <= 1000 <= maze[i][j] <= 1000 <= startX, startY, endX, endY < n

ExampleInput:int n = 4;int m = 4;int maze[4][4] = {{2, 0, 0, 1},{1, 1, 0, 1},{1, 0, 1, 0},{1, 1, 1, 0}};int startX = 0, startY = 0;int endX = 3, endY = 3;

Output:5

Explanation:From the start cell (0, 0) with value 2, you can move to cells (2, 0) and (0, 2). From (2, 0) with value 1, you can move to cells (1, 0) and (2, 1). Continue this process until you reach (3, 3). The shortest path requires 5 moves.

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem DSA Post-Midterm

Language Used: C

Source Code:

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 105
typedef struct {
int x, y, dist;
} Node;
int minimumStepsToExit(int n, int m, int maze[][m],
int startX, int startY,
int endX, int endY) {
if (!(startX == endX && startY == endY) && maze[startX][startY] == 0)
return -1;
if (startX == endX && startY == endY)
return 0;
bool visited[MAX][MAX] = { false };
int dirX[4] = { 1, -1, 0, 0 };
int dirY[4] = { 0, 0, 1, -1 };
Node queue[MAX * MAX];
int front = 0, rear = 0;
queue[rear++] = (Node){ startX, startY, 0 };
visited[startX][startY] = true;
while (front < rear) {
Node cur = queue[front++];
int x = cur.x, y = cur.y, dist = cur.dist;
if (x == endX && y == endY)
return dist;
int jump = maze[x][y];
if (jump == 0) continue;
for (int i = 0; i < 4; i++) {
int nx = x + dirX[i] * jump;
int ny = y + dirY[i] * jump;
if (nx < 0 || ny < 0 || nx >= n || ny >= m)
continue;
if (visited[nx][ny])
continue;
if (maze[nx][ny] == 0 && !(nx == endX && ny == endY))
continue;
visited[nx][ny] = true;
queue[rear++] = (Node){ nx, ny, dist + 1 };
}
}
return -1;
}
int main() {
int n, m;
if (scanf("%d %d", &n, &m) != 2) return 0;
int maze[n][m];
for (int i = 0; i < n; i++)for (int j = 0; j < m; j++)
```

```
scanf("%d", &maze[i][j]);
int startX, startY, endX, endY;
scanf("%d %d %d %d", &startX, &startY, &endX, &endY);
int result = minimumStepsToExit(n, m, maze, startX, startY, endX, endY);
printf("%d\n", result);
return 0;
}
```

Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

5

Compilation Status: Passed

Execution Time:

0.001s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

3

Compilation Status: Passed

Execution Time:

0.002s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

3

Compilation Status: Passed

Execution Time:

0.001s

2. Problem: File Merge Cost CalculationYou are given multiple files with varying sizes, and you need to merge these files into one single file. Each merge operation takes a certain cost, equal to the sum of the file sizes being merged. Your task is to calculate the minimum total cost of merging all the files into one file.

This is a real-world scenario encountered in systems dealing with external file sorting and data storage management. Efficient file merging can minimize computation time and resources.

Input:The first input is an integer whose name is n which means number of files.The second input is a list of n integers which represents the size of each file.

Output:Print a single integer representing the minimum cost to merge all files into one file.

Problem Statement:Given n files, each with a certain size, you need to merge all the files into one. The cost of each merge is the sum of the sizes of the two files being merged. After a merge, the two files are replaced by a single file of their combined size. The goal is to find the minimum cost required to merge all the files into one file.

ExampleInput:410 20 30 40Output:190

Explanation:Merge file sizes 10 and 20 → Cost = 30, remaining files = [30, 30, 40]Merge file sizes 30 and 30 → Cost = 60, remaining files = [60, 40]Merge file sizes 60 and 40 → Cost = 100, remaining files = [100] Total cost = 30 + 60 + 100 = 190Constraints:2 <= n <= 10001 <= size of each file <= 10^6Detailed Explanation:You can think of this problem as similar to the Huffman coding problem. The most optimal strategy is to always merge the two smallest files together first. By following this greedy approach, we can ensure that the overall cost is minimized.

We use a min-heap (priority queue) to always merge the two smallest files efficiently. Each time two files are merged, the cost is added to a running total, and the resulting file is added back to the heap. This process continues until only one file remains.

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem DSA Post-Midterm

Language Used: C

Source Code:

```
#include <stdio.h>
#define MAX 2000
int heap[MAX];
int heapSize = 0;
void swap(int *a, int *b) {
int temp = *a;
*a = *b;
*b = temp;
}
void heapifyUp(int idx) {
while (idx > 1 && heap[idx] < heap[idx / 2]) {
swap(&heap[idx], &heap[idx / 2]);
idx /= 2;
}
}
void heapifyDown(int idx) {
while (1) {
int left = idx * 2;
int right = left + 1;
int smallest = idx;
if (left <= heapSize && heap[left] < heap[smallest])
smallest = left;
if (right <= heapSize && heap[right] < heap[smallest])
smallest = right;
if (smallest == idx)
break;
swap(&heap[idx], &heap[smallest]);
idx = smallest;
}
}
void pushHeap(int val) {
heap[++heapSize] = val;
heapifyUp(heapSize);
}
int popHeap() {
int top = heap[1];
heap[1] = heap[heapSize--];
heapifyDown(1);
return top;
}
int main() {
int n;
scanf("%d", &n);
heapSize = 0;
for (int i = 0; i < n; i++) {int x;
scanf("%d", &x);
pushHeap(x);
```

```
}
long long totalCost = 0;
while (heapSize > 1) {
int a = popHeap();
int b = popHeap();
int merged = a + b;
totalCost += merged;
pushHeap(merged);
}
printf("%lld\n", totalCost);
return 0;
}
```

Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

190

Compilation Status: Passed

Execution Time:

0.001s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

9

Compilation Status: Passed

Execution Time:

0.001s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

165

Compilation Status: Passed

Execution Time:

0.001s

3. Balanced Parentheses CheckerProblem Statement

Write a program to check if a given expression has balanced parentheses (), {}, and [] using a stack.

Description

Push opening brackets onto a stack.

Pop when encountering closing brackets and check for matching pairs.

If any mismatch or leftover brackets remain, the expression is not balanced.

Input Format

Single string expression

Output Format

Print "Balanced" if parentheses are correctly matched

Print "Not Balanced" otherwise

Sample Input{[()]}

Sample OutputBalanced

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem DSA Post-Midterm

Language Used: C

Source Code:

```
#include <stdio.h>
#include <string.h>
#define MAX 10005
char stackArr[MAX];
int top = -1;
void push(char c) {
stackArr[++top] = c;
}
char pop() {
if (top == -1) return '\0';
return stackArr[top--];
}
int isMatching(char open, char close) {
if (open == '(' && close == ')') return 1;
if (open == '{' && close == '}') return 1;
if (open == '[' && close == ']') return 1;
return 0;
}
int main() {
char expr[MAX];
scanf("%s", expr);
int n = strlen(expr);
top = -1;
for (int i = 0; i < n; i++) {
char c = expr[i];
if (c == '(' || c == '{' || c == '[') {
push(c);
}
else if (c == ')' || c == '}' || c == ']') {
if (top == -1) {
printf("Not Balanced\n");
return 0;
}
char open = pop();
if (!isMatching(open, c)) {
printf("Not Balanced\n");
return 0;
}
}
}
if (top == -1)
printf("Balanced\n");
else
printf("Not Balanced\n");
return 0;
}
```


Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

Balanced

Compilation Status: Passed

Execution Time:

0.001s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

Not Balanced

Compilation Status: Passed

Execution Time:

0.001s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

Not Balanced

Compilation Status: Passed

Execution Time:

0.001s

4. Evaluate Postfix ExpressionProblem Statement

Given a postfix expression, evaluate its value using a stack.

Description

Each token can be an integer or operator (+, -, *, /).

Use a stack to push operands and pop them when an operator is encountered.

Compute and push the result back on the stack.

Input Format

Single string postfix expression (tokens separated by spaces)

Output Format

Result as an integer

Sample Input2 3 1 * + 9 -

Sample Output-4

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem DSA Post-Midterm

Language Used: C

Source Code:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX 10005
int stackArr[MAX];
int top = -1;
void push(int x) {
stackArr[++top] = x;
}
int pop() {
return stackArr[top--];
}
int main() {
char expr[MAX];
fgets(expr, sizeof(expr), stdin);
char *token = strtok(expr, " ");
while (token != NULL) {if (strcmp(token, "+") == 0 ||
strcmp(token, "-") == 0 ||
strcmp(token, "*") == 0 ||
strcmp(token, "/") == 0) {
```

```
int b = pop();
int a = pop();
int result;
if (strcmp(token, "+") == 0)
result = a + b;
else if (strcmp(token, "-") == 0)
result = a - b;
else if (strcmp(token, "*") == 0)
result = a * b;
else
result = a / b;
push(result);
}
else {
push(atoi(token));
}
token = strtok(NULL, " ");
}
printf("%d\n", pop());
return 0;
}
```

Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

5

Compilation Status: Passed

Execution Time:

0.001s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

40

Compilation Status: Passed

Execution Time:

0.001s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

1

Compilation Status: Passed

Execution Time:

0.001s

5. Implement Circular QueueProblem Statement

Write a program to implement a circular queue with enqueue, dequeue, and display operations.

Description

Create a circular queue of size n.

Perform operations:

enqueue(x) – Add an element to the queue.

dequeue() – Remove an element from the queue.

display() – Print all elements in queue order.

Make sure to handle overflow and underflow conditions.

Input Format

First line: Integer n (size of the queue)

Subsequent lines: Operations in the format:

"enqueue x"

"dequeue"

"display"

Output Format

Print the queue after each display operation.

Print "Queue Overflow" or "Queue Underflow" for invalid operations.

Sample Input5enqueue 10enqueue 20enqueue 30displaydequeuedisplay

Sample Output10 20 3020 30

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem DSA Post-Midterm

Language Used: C

Source Code:

```
#include <stdio.h>
#include <string.h>
#define MAX 10005
int queueArr[MAX];
int front = -1, rear = -1, n;
int isFull() {
return (front == (rear + 1) % n);
}
int isEmpty() {
return (front == -1);
}
void enqueue(int x) {
if (isFull()) {
printf("Queue Overflow\n");
return;
}
if (isEmpty()) {
front = rear = 0;
}
else {
rear = (rear + 1) % n;
}
queueArr[rear] = x;
}
void dequeue() {
if (isEmpty()) {
printf("Queue Underflow\n");
return; }
if (front == rear) {
front = rear = -1;
}
else {
```

```
front = (front + 1) % n;
}
}
void display() {
if (isEmpty()) {
return;
}
int i = front;
while (1) {
printf("%d", queueArr[i]);
if (i == rear) break;
printf(" ");
i = (i + 1) % n;
}
printf("\n");
}
int main() {
scanf("%d", &n);
char op[20];
while (scanf("%s", op) != EOF) {
if (strcmp(op, "enqueue") == 0) {
int x;
scanf("%d", &x);
enqueue(x);
}
else if (strcmp(op, "dequeue") == 0) {
dequeue();
}
else if (strcmp(op, "display") == 0) {
display();
}
}
return 0;
}
```


Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

10 20 30
20 30

Compilation Status: Passed

Execution Time:

0.001s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

Queue Overflow
5 10 15

Compilation Status: Passed

Execution Time:

0.001s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

Queue Underflow

Compilation Status: Passed

Execution Time:

0.001s

6. Reverse First k Elements of QueueProblem Statement

Given a queue and integer k, reverse the first k elements while keeping the rest in the same order.

Description

Use a stack to reverse the first k elements.

The remaining elements should stay in their original order.

Input Format

First line: Integer n (size of queue)

Second line: n integers representing queue elements

Third line: Integer k

Output Format

Print the modified queue after reversing the first k elements.

Sample Input510 20 30 40 503

Sample Output30 20 10 40 50

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem DSA Post-Midterm

Language Used: C

Source Code:

```
#include <stdio.h>
#define MAX 10005
int queue[MAX];
int front = 0, rear = -1;
int main() {
int n, k;
scanf("%d", &n);
for (int i = 0; i < n; i++) {
scanf("%d", &queue[i]);
rear++;
}
scanf("%d", &k);
int stack[MAX];
int top = -1;
for (int i = 0; i < k; i++) {
stack[++top] = queue[front++];
}
int temp[MAX], idx = 0;
while (top != -1) {
temp[idx++] = stack[top--];
}
while (front <= rear) {
temp[idx++] = queue[front++];
}for (int i = 0; i < idx; i++) {
printf("%d", temp[i]);
if (i < idx - 1) printf(" ");
}
```

```
return 0;
}
```

Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

4 3 2 1

Compilation Status: Passed

Execution Time:

0.001s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

15 10 5 20 25 30

Compilation Status: Passed

Execution Time:

0.001s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

8 6 4 2 10 12

Compilation Status: Passed

Execution Time:

0.001s

7. Network Coverage Using Binary TreesTitle: Designing an Optimal Network Coverage System Using a Binary Tree

Problem Description:You have been hired to design a network coverage system for a telecommunications company. The network consists of multiple towers, and the range of each tower is represented as a node in a binary tree. The range of each tower is stored in such a way that each parent node has a greater range than its left child and a smaller range than its right child. This structure mimics the properties of a Binary Search Tree (BST).

Your task is to perform the following operations on this tree:

Insert a tower with a specific range into the tree.Delete a tower with a specific range from the tree.Find the minimum range tower in the tree.Find the maximum range tower in the tree.Find if a specific range exists in the tree.Print the towers' ranges in in-order traversal (ascending order).

Input:The first line contains an integer N, the number of operations to be performed (1 ≤ N ≤ 1000).Each of the next N lines contains an operation in one of the following formats:INSERT : Insert a tower with the given range into the binary tree.DELETE : Delete a tower with the given range from the binary tree.MIN: Find the tower with the minimum range.MAX: Find the tower with the maximum range.FIND : Check if a tower with the specified range exists.PRINT: Print the ranges of the towers in the binary tree in ascending order.

Output:For MIN, print the minimum tower range.For MAX, print the maximum tower range.For FIND, print "Found" if the tower range exists, otherwise print "Not Found".For PRINT, print the ranges in ascending order separated by spaces.For DELETE, if the tower is not found, print "Tower not found".

Constraints:The range of each tower will be an integer value between 1 and 10^5.No duplicate tower ranges will be inserted.All operations must be performed efficiently to ensure that the program runs within time limits.

Example Input and Output:Input:10INSERT 50INSERT 30INSERT 70INSERT 20INSERT 40INSERT 60MINMAXFIND 40PRINT

Output:2070Found20 30 40 50 60 70

Detailed Explanation:Test Case 1:

Insert 50, 30, 70, 20, 40, 60 into the binary tree. Resulting Tree:50/ \30 70/ \ /

20 40 60

2. The minimum range is `20`, and the maximum range is `70`.

3. Searching for a tower with range 40 will return `"Found"`.

4. The `PRINT` operation will output the ranges in ascending order:

20 30 40 50 60 70

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem DSA Post-Midterm

Language Used: C

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct Node {
int data;
struct Node *left, *right;
} Node;
Node* createNode(int val) {
Node* t = (Node*)malloc(sizeof(Node));
t->data = val;
t->left = t->right = NULL;
return t;
}
Node* insert(Node* root, int val) {
if (!root) return createNode(val);
if (val < root->data) root->left = insert(root->left, val);
else if (val > root->data) root->right = insert(root->right, val);
return root;
}
int find(Node* root, int val) {
while (root) {
if (val == root->data) return 1;
if (val < root->data) root = root->left;
else root = root->right;
}
return 0;
}
Node* findMinNode(Node* root) {
while (root && root->left) root = root->left;
return root;
}
Node* deleteNode(Node* root, int val, int *deleted) {if (!root) return NULL;
if (val < root->data)
root->left = deleteNode(root->left, val, deleted);
else if (val > root->data)
root->right = deleteNode(root->right, val, deleted);
else {
```

```
*deleted = 1;
if (!root->left && !root->right) {
free(root);
return NULL;
}
else if (!root->left) {
Node* temp = root->right;
free(root);
return temp;
}
else if (!root->right) {
Node* temp = root->left;
free(root);
return temp;
}
else {
Node* temp = findMinNode(root->right);
root->data = temp->data;
root->right = deleteNode(root->right, temp->data, deleted);
}
}
return root;
}
void inorder(Node* root, int *firstPrinted) {
if (!root) return;
inorder(root->left, firstPrinted);
if (*firstPrinted)
printf(" ");
printf("%d", root->data);
*firstPrinted = 1;
inorder(root->right, firstPrinted);
}
int main() {
int N;
scanf("%d", &N);
Node* root = NULL;
while (N--) {
char op[20];
int x;
scanf("%s", op);
if (strcmp(op, "INSERT") == 0) {
scanf("%d", &x);
root = insert(root, x);
}
else if (strcmp(op, "DELETE") == 0) {
scanf("%d", &x);
int deleted = 0;
root = deleteNode(root, x, &deleted);if (!deleted)
printf("Tower not found\n");
}
else if (strcmp(op, "MIN") == 0) {
if (root) {
Node* t = findMinNode(root);
printf("%d\n", t->data);
```

```
}
}
else if (strcmp(op, "MAX") == 0) {
Node* t = root;
while (t && t->right) t = t->right;
if (t) printf("%d\n", t->data);
}
else if (strcmp(op, "FIND") == 0) {
scanf("%d", &x);
if (find(root, x)) printf("Found\n");
else printf("Not Found\n");
}
else if (strcmp(op, "PRINT") == 0) {
int firstPrinted = 0;
inorder(root, &firstPrinted);
printf("\n");
}
}
return 0;
}
```

Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

20
70
Found
20 30 40 50 60 70

Compilation Status: Passed

Execution Time:

0.001s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

40
60
Not Found
40 60

Compilation Status: Passed

Execution Time:

0.002s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

100
120
100 120

Compilation Status: Passed

Execution Time:

0.001s

8. Robotic Vacuum Cleaner Path PlanningProblem StatementYou are tasked with programming a robotic vacuum cleaner that navigates a room divided into a grid of cells. The vacuum cleaner starts at the top-left corner of the grid and must determine its path based on sensor input that detects obstacles in the adjacent cells.

The vacuum can move in four directions: up, down, left, and right. The vacuum cleaner should prioritize movements based on the following rules:

If there is no obstacle to the right and moving right brings the vacuum closer to the bottom-right corner, move right.If there is an obstacle to the right or moving right would not bring the vacuum closer, check if the vacuum can move down. If it can, move down.If neither moving right nor down is possible, check if the vacuum can move left. If it can, move left.If none of the above moves are possible, move up.The program should decide and print the vacuum cleaner's next move based on these rules.

Input:

The first line contains two integers, n and m, representing the dimensions of the gri (rows and columns).The next n lines contain m characters each, representing the grid. A . represents an open cell, and a # represents an obstacle.The last line contai two integers, r and c, representing the current position of the vacuum cleaner in the grid.

Output:

Print the direction the vacuum cleaner should move next: "RIGHT", "DOWN", "LEFT", or "UP".

Constraints:

1 <= n, m <= 100The grid is guaranteed to have an open cell at the top-left (0, 0) and the bottom-right corner (n-1, m-1).The initial position (r, c) will always be an open cell.

ExplanationLet's consider a sample input to understand how the vacuum cleaner will determine its next move:

Example Input:4 4..#..#....#...#.2 1

Here, n = 4 and m = 4, so the grid is a 4x4 matrix. The vacuum cleaner is at position (2, 1) (third row, second column).

The grid looks like this:..#..#....#...#.

Rule 1: Moving right would place the vacuum at (2, 2), but there is an obstacle (#) there, so it can't move right.Rule 2: Moving down to (3, 1) is possible and would bring the vacuum closer to the bottom-right corner.Result: The vacuum should move "DOWN".

Expected Output:DOWN

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem DSA Post-Midterm

Language Used: C

Source Code:

```
#include <stdio.h>
#include <string.h>

int main() {
int n, m, r, c;
scanf("%d %d", &n, &m);

char grid[100][101];  // 100 rows, 100 columns + null terminator

for (int i = 0; i < n; i++) {
```

```
scanf("%s", grid[i]);
}

scanf("%d %d", &r, &c);

// Get the bottom-right corner position
int bottom_r = n - 1;
int bottom_c = m - 1;

// Check all four directions in priority order

// Rule 1: Try moving RIGHT
if (c + 1 < m && grid[r][c + 1] == '.' &&
((r + c + 1) < (bottom_r + bottom_c))) {
printf("RIGHT\n");
return 0;
}

// Rule 2: Try moving DOWN
if (r + 1 < n && grid[r + 1][c] == '.' &&
((r + 1 + c) < (bottom_r + bottom_c))) {
printf("DOWN\n");
return 0;
}

// Rule 3: Try moving LEFT
if (c - 1 >= 0 && grid[r][c - 1] == '.') {
printf("LEFT\n");
return 0;
}

// Rule 4: Move UP (always possible according to constraints)
printf("UP\n");
return 0;
}
```


Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

DOWN

Compilation Status: Passed

Execution Time:

0.001s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

RIGHT

Compilation Status: Passed

Execution Time:

0.001s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

LEFT

Compilation Status: Passed

Execution Time:

0.001s

9. Post-order Traversal of Binary Tree

Problem Statement:Write a program to perform post-order traversal of a binary tree. The traversal should follow Left Right Root order.

Description:The program will read the binary tree in level order. Use -1 to represent a NULL node. Build the binary tree from the input and then print its post-order traversal sequence.

Input Format:

First line: Integer n — number of nodes (including NULLs as -1 in level order).

Second line: n space-separated integers representing the node values in level order (-1 for NULL).

Output Format:

Single line: Post-order traversal sequence (space-separated).

Sample Input:

71 2 3 4 5 -1 6

Sample Output:

4 5 2 6 3 1

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem DSA Post-Midterm

Language Used: C

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
int data;
struct Node* left;
struct Node* right;
} Node;
typedef struct Queue {
Node* arr[1000];
int front, rear;
} Queue;
void enqueue(Queue* q, Node* node) {
q->arr[q->rear++] = node;
}Node* dequeue(Queue* q) {
return q->arr[q->front++];
}
Node* createNode(int val) {
if (val == -1) return NULL;
Node* node = (Node*)malloc(sizeof(Node));
node->data = val;
node->left = node->right = NULL;
return node;
}
void postorder(Node* root, int* first) {
if (!root) return;
postorder(root->left, first);
```

```
postorder(root->right, first);
if (*first) printf(" ");
printf("%d", root->data);
*first = 1;
}
int main() {
int n;
scanf("%d", &n);
if (n == 0) return 0;
int values[1000];
for (int i = 0; i < n; i++) scanf("%d", &values[i]);
if (values[0] == -1) return 0;
Node* root = createNode(values[0]);
Queue q;
q.front = q.rear = 0;
enqueue(&q, root);
int idx = 1;
while (idx < n) {
Node* curr = dequeue(&q);
if (!curr) continue;
if (idx < n) {
curr->left = createNode(values[idx++]);
if (curr->left) enqueue(&q, curr->left);
}
if (idx < n) {
curr->right = createNode(values[idx++]);
if (curr->right) enqueue(&q, curr->right);
}
}
int first = 0;
postorder(root, &first);
printf("\n");
return 0;
}
```


Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

4 5 2 6 3 1

Compilation Status: Passed

Execution Time:

0.001s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

25 20 40 50 30 10

Compilation Status: Passed

Execution Time:

0.001s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

1 4 3 7 9 8 5

Compilation Status: Passed

Execution Time:

0.001s

10. Construct Binary Tree from Post-order and In-order Traversal

Problem Statement:Given the in-order and post-order traversal sequences of a binary tree, construct the tree and print its pre-order traversal.

Description:

The program should reconstruct the binary tree from the provided in-order and post-order arrays.

After constructing the tree, print the pre-order traversal (Root Left Right).

Input Format:

First line: Integer n — number of nodes.

Second line: n integers — in-order traversal sequence.

Third line: n integers — post-order traversal sequence.

Output Format:

Single line: Pre-order traversal sequence (space-separated).

Sample Input:

64 2 5 1 6 34 5 2 6 3 1

Sample Output:

1 2 4 5 3 6

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem DSA Post-Midterm

Language Used: C

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
int data;
struct Node* left;
struct Node* right;
} Node;
Node* createNode(int val) {
Node* node = (Node*)malloc(sizeof(Node));
node->data = val;
node->left = node->right = NULL;return node;
}
Node* build(int in[], int post[], int inStart, int inEnd, int postStart, int postEnd) {
if (inStart > inEnd || postStart > postEnd) return NULL;
int rootVal = post[postEnd];
Node* root = createNode(rootVal);
int inIndex = inStart;
while (inIndex <= inEnd && in[inIndex] != rootVal) inIndex++;
int leftCount = inIndex - inStart;
root->left = build(in, post, inStart, inIndex - 1, postStart, postStart + leftCount - 1);
root->right = build(in, post, inIndex + 1, inEnd, postStart + leftCount, postEnd - 1);
return root;
}
void preorder(Node* root, int* first) {
```

```
if (!root) return;
if (*first) printf(" ");
printf("%d", root->data);
*first = 1;
preorder(root->left, first);
preorder(root->right, first);
}
int main() {
int n;
scanf("%d", &n);
int in[1000], post[1000];
for (int i = 0; i < n; i++) scanf("%d", &in[i]);
for (int i = 0; i < n; i++) scanf("%d", &post[i]);
Node* root = build(in, post, 0, n - 1, 0, n - 1);
int first = 0;
preorder(root, &first);
printf("\n");
return 0;
}
```

Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

1 2 3

Compilation Status: Passed

Execution Time:

0.001s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

4 2 1 3 5

Compilation Status: Passed

Execution Time:

0.001s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

2 1 4 3

Compilation Status: Passed

Execution Time:

0.001s

11. Problem Statement:Title: Memory Efficient Graph Operations using Dynamic Memory Allocation

You are tasked with creating a memory-efficient program to process and manage a graph in C using dynamic memory allocation. The program should allow you to store and manipulate an undirected graph represented by n nodes and m edges, and later release the allocated memory using free() when operations are complete.

Given a list of edges, your task is to:

Store the edges in an adjacency list.Counting the number of the connected components that are in the graph.

Input Format:The first line contains two integers n (1 ≤ n ≤ 10^5) and m (1 ≤ m ≤ 2 * 10^5), representing the number of nodes and the number of edges.The next m lines contain two integers u and v (1 ≤ u, v ≤ n), indicating that there is an edge between node u and node v.

Output Format:Output a single integer, the number of connected components in the graph.

Example:Input:5 31 22 34 5

Output:2

Explanation:The graph has two connected components:

Component 1: Nodes {1, 2, 3}Component 2: Nodes {4, 5}

Constraints:Time Complexity: Your program should run in O(n + m) time due to the

constraints.Memory Constraints: Your program should efficiently use dynamic memory allocation, and ensure that all allocated memory is properly freed using free() after processing is completed.

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem DSA Post-Midterm

Language Used: C

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
void dfs(int u, int* visited, int** adj, int* size) {
visited[u] = 1;
for (int i = 0; i < size[u]; i++) {
int v = adj[u][i];
if (!visited[v])
dfs(v, visited, adj, size);
}
}
int main() {
int n, m;
scanf("%d %d", &n, &m);
int** adj = (int**)malloc((n + 1) * sizeof(int*));
int* size = (int*)calloc(n + 1, sizeof(int));
int* capacity = (int*)malloc((n + 1) * sizeof(int));
for (int i = 1; i <= n; i++) {
adj[i] = (int*)malloc(2 * sizeof(int));
capacity[i] = 2;
}
for (int i = 0; i < m; i++) {
int u, v;
scanf("%d %d", &u, &v);if (size[u] == capacity[u]) {
capacity[u] *= 2;
adj[u] = (int*)realloc(adj[u], capacity[u] * sizeof(int));
}
adj[u][size[u]++] = v;
if (size[v] == capacity[v]) {
capacity[v] *= 2;
adj[v] = (int*)realloc(adj[v], capacity[v] * sizeof(int));
}
adj[v][size[v]++] = u;
}
int* visited = (int*)calloc(n + 1, sizeof(int));
int components = 0;
for (int i = 1; i <= n; i++) {
if (!visited[i]) {
```

```
dfs(i, visited, adj, size);
components++;
}
}
printf("%d\n", components);
for (int i = 1; i <= n; i++)
free(adj[i]);
free(adj);
free(size);
free(capacity);
free(visited);
return 0;
}
```

Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

2

Compilation Status: Passed

Execution Time:

0.001s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

3

Compilation Status: Passed

Execution Time:

0.001s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

2

Compilation Status: Passed

Execution Time:

0.001s

12. File System Simulation Using Advanced Data Structures

Title: File System Simulation with Hierarchical Structure and Memory Management

Problem Description:You are tasked with developing a simplified file system simulation for a large enterprise environment. The file system needs to support multiple operations like creating directories, creating files within directories, updating files, deleting files, and querying directories. Each directory can contain multiple files and subdirectories. Files and directories should be organized hierarchically, and the memory used by files and directories needs to be managed efficiently.

In this problem, you'll use dynamic data structures like trees and linked lists to simulate the file system. The operations include adding, deleting, updating files, and directories, as well as querying the system to return directory contents.

Supported Operations:Create a directory: Create a new directory under an existing directory.Create a file: Add a new file with given content in a specific directory.Update a file: Modify the content of an existing file in a directory.Delete a file: Remove a file from a directory.Query directory: List all files and subdirectories in a given directory, sorted lexicographically.Get directory size: Calculate the total size of all files in a given directory (including subdirectories).Your task is to implement this file system simulation in C using appropriate data structures (e.g., tree for directories and linked list for files).

Input:The first line contains an integer n, the number of operations (1 ≤ n ≤ 10^5).The next n lines describe operations in one of the following formats:1 parent_dir new_dir: Create a new directory new_dir under parent_dir.2 dir_name file_name file_size: Create a new file with file_name and size file_size in dir_name.3 dir_name file_name new_size: Update the content of the file file_name in dir_name with new size new_size.4 dir_name file_name: Delete the file file_name from dir_name.5 dir_name: List all files and subdirectories in dir_name in lexicographical order.6 dir_name: Print the total size of all files (including those in subdirectories) in dir_name.

Output:For operation 5, output the list of files and directories in lexicographical order.For operation 6, output the total size of all files in the specified directory (including all subdirectories).

Constraints:File and directory names will only consist of lowercase English letters and will not exceed 100 characters.File size will be between 1 and 10^6.No two files or directories in the same directory will have the same name.If a directory or file to operated on does not exist, print an appropriate error message.

Example Input and Output:Input 1:81 root dir11 root dir22 dir1 file1 1002 dir1 file2 2005 dir13 dir1 file1 3005 dir16 rootOutput1:file1file2file1file2Total size: 500

Explanation:Initially, two directories are created under the root directory: dir1 and dir2.Two files are added to dir1, and then the contents of dir1 are displayed.The content of file1 in dir1 is updated, and the directory contents are shown again.Finally, the total size of all files in root is calculated.

Input 2:71 root dir11 root dir22 dir1 file1 5004 dir1 file15 dir16 rootOutput 2:file1Total size: 0

Explanation:The directory dir1 is created under the root and a file is added to it.The file is then deleted, and the directory contents are shown.The total size of all files in the root directory is calculated and shown as 0 since the file was deleted.

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem DSA Post-Midterm

Language Used: C

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct FileNode {
char name[101];
int size;struct FileNode* next;
} FileNode;
typedef struct DirNode {
char name[101];
struct DirNode* subdirs;
struct DirNode* next;
FileNode* files;
} DirNode;
DirNode* findDir(DirNode* root, const char* name) {
if (!root) return NULL;
if (strcmp(root->name, name) == 0) return root;
DirNode* child = root->subdirs;
while (child) {
DirNode* res = findDir(child, name);
if (res) return res;
child = child->next;
}
```

```
return NULL;
}
void addDir(DirNode* parent, const char* name) {
DirNode* newDir = (DirNode*)malloc(sizeof(DirNode));
strcpy(newDir->name, name);
newDir->subdirs = NULL;
newDir->files = NULL;
newDir->next = NULL;
if (!parent->subdirs) parent->subdirs = newDir;
else {
DirNode* temp = parent->subdirs;
while (temp->next) temp = temp->next;
temp->next = newDir;
}
}
void addFile(DirNode* dir, const char* fname, int size) {
FileNode* newFile = (FileNode*)malloc(sizeof(FileNode));
strcpy(newFile->name, fname);
newFile->size = size;
newFile->next = NULL;
if (!dir->files) dir->files = newFile;
else {
FileNode* temp = dir->files;
while (temp->next) temp = temp->next;
temp->next = newFile;
}
}
void updateFile(DirNode* dir, const char* fname, int newSize) {
FileNode* temp = dir->files;
while (temp) {
if (strcmp(temp->name, fname) == 0) {
temp->size = newSize;
return;
}
temp = temp->next;
}
printf("File not found\n");}
void deleteFile(DirNode* dir, const char* fname) {
FileNode* temp = dir->files;
FileNode* prev = NULL;
while (temp) {
if (strcmp(temp->name, fname) == 0) {
if (prev) prev->next = temp->next;
else dir->files = temp->next;
free(temp);
return;
}
prev = temp;
temp = temp->next;
}
printf("File not found\n");
}
void listDir(DirNode* dir) {
int count = 0;
```

```
FileNode* f = dir->files;
while (f) { count++; f = f->next; }
DirNode* d = dir->subdirs;
while (d) { count++; d = d->next; }
if (count == 0) return;
char** names = (char**)malloc(count * sizeof(char*));
f = dir->files;
int idx = 0;
while (f) { names[idx++] = f->name; f = f->next; }
d = dir->subdirs;
while (d) { names[idx++] = d->name; d = d->next; }
for (int i = 0; i < count - 1; i++)
for (int j = i + 1; j < count; j++)
if (strcmp(names[i], names[j]) > 0) {
char* tmp = names[i];
names[i] = names[j];
names[j] = tmp;
}
for (int i = 0; i < count; i++) printf("%s\n", names[i]);
free(names);
}
int getDirSize(DirNode* dir) {
int total = 0;
FileNode* f = dir->files;
while (f) { total += f->size; f = f->next; }
DirNode* d = dir->subdirs;
while (d) { total += getDirSize(d); d = d->next; }
return total;
}
void freeFiles(FileNode* f) {
while (f) {
FileNode* tmp = f;
f = f->next;
free(tmp);
}
}void freeDir(DirNode* dir) {
if (!dir) return;
DirNode* d = dir->subdirs;
while (d) {
DirNode* nextD = d->next;
freeDir(d);
d = nextD;
}
freeFiles(dir->files);
free(dir);
}
int main() {
int n;
scanf("%d", &n);
DirNode* root = (DirNode*)malloc(sizeof(DirNode));
strcpy(root->name, "root");
root->subdirs = NULL;
root->files = NULL;
root->next = NULL;
```

```
for (int i = 0; i < n; i++) {
int op;
scanf("%d", &op);
if (op == 1) {
char parent[101], newDir[101];
scanf("%s %s", parent, newDir);
DirNode* p = findDir(root, parent);
if (!p) { printf("Directory not found\n"); continue; }
addDir(p, newDir);
}
else if (op == 2) {
char dir[101], fname[101];
int size;
scanf("%s %s %d", dir, fname, &size);
DirNode* d = findDir(root, dir);
if (!d) { printf("Directory not found\n"); continue; }
addFile(d, fname, size);
}
else if (op == 3) {
char dir[101], fname[101];
int size;
scanf("%s %s %d", dir, fname, &size);
DirNode* d = findDir(root, dir);
if (!d) { printf("Directory not found\n"); continue; }
updateFile(d, fname, size);
}
else if (op == 4) {
char dir[101], fname[101];
scanf("%s %s", dir, fname);
DirNode* d = findDir(root, dir);
if (!d) { printf("Directory not found\n"); continue; }
deleteFile(d, fname);
}
else if (op == 5) {
char dir[101];
scanf("%s", dir);
DirNode* d = findDir(root, dir);
if (!d) { printf("Directory not found\n"); continue; }
listDir(d); }
else if (op == 6) {
char dir[101];
scanf("%s", dir);
DirNode* d = findDir(root, dir);
if (!d) { printf("Directory not found\n"); continue; }
printf("Total size: %d\n", getDirSize(d));
}
}
freeDir(root);
return 0;
}
```


Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

android
ios
Total size: 400
ui
Total size: 450

Compilation Status: Passed

Execution Time:

0.001s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

a
c
Total size: 150
Total size: 600

Compilation Status: Passed

Execution Time:

0.001s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

docs
images
Total size: 3500
Total size: 1500
pic1
report

Compilation Status: Passed

Execution Time:

0.001s

13. BFS Traversal of Graph

Problem Statement:Write a program to perform Breadth-First Search (BFS) traversal of a graph starting from a given source vertex.

Description:

The program will read a graph represented as an edge list.

Use BFS to traverse the graph from the given source vertex.

Print the BFS traversal order of the vertices.

Input Format:

First line: Two integers n and m — number of vertices and number of edges.

Next m lines: Two integers u and v representing an edge between vertices u and v.

Last line: Source vertex s.

Output Format:

A single line with BFS traversal order (space-separated).

Sample Input:

5 40 10 21 32 40

Sample Output:

0 1 2 3 4

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem DSA Post-Midterm

Language Used: C

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
int vertex;
struct Node* next;
} Node;
typedef struct Queue {
int *data;
int front, rear, size, capacity;
} Queue;
Queue* createQueue(int capacity) {
Queue* q = (Queue*)malloc(sizeof(Queue));
q->data = (int*)malloc(capacity * sizeof(int));
q->front = 0;
q->rear = -1;
q->size = 0;
q->capacity = capacity;
return q;
}
void enqueue(Queue* q, int val) {
q->rear = (q->rear + 1) % q->capacity;
q->data[q->rear] = val;
q->size++;
}
int dequeue(Queue* q) {
int val = q->data[q->front];
q->front = (q->front + 1) % q->capacity;
q->size--;
return val;}
int isEmpty(Queue* q) {
return q->size == 0;
}
void addEdge(Node** adj, int u, int v) {
Node* newNode = (Node*)malloc(sizeof(Node));
newNode->vertex = v;
newNode->next = adj[u];
adj[u] = newNode;
}
int main() {
int n, m;
scanf("%d %d", &n, &m);
Node** adj = (Node**)malloc(n * sizeof(Node*));
for (int i = 0; i < n; i++) adj[i] = NULL;
for (int i = 0; i < m; i++) {
int u, v;
scanf("%d %d", &u, &v);
addEdge(adj, u, v);
addEdge(adj, v, u);
}
```

```
int s;
scanf("%d", &s);
int* visited = (int*)calloc(n, sizeof(int));
Queue* q = createQueue(n);
// Start BFS from source
enqueue(q, s);
visited[s] = 1;
while (!isEmpty(q)) {
int curr = dequeue(q);
printf("%d", curr);
Node* temp = adj[curr];
while (temp) {
if (!visited[temp->vertex]) {
enqueue(q, temp->vertex);
visited[temp->vertex] = 1;
}
temp = temp->next;
}
if (!isEmpty(q)) printf(" ");
}
printf("\n");
for (int i = 0; i < n; i++) {
Node* temp = adj[i];
while (temp) {Node* next = temp->next;
free(temp);
temp = next;
}
}
free(adj);
free(visited);
free(q->data);
free(q);
return 0;
}
```


Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

0 1 2 3

Compilation Status: Passed

Execution Time:

0.001s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

0 2 1 5 4 3

Compilation Status: Passed

Execution Time:

0.002s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

0 2 1 4 3 6 5

Compilation Status: Passed

Execution Time:

0.001s

14. Shortest Path in Unweighted Graph (BFS)

Problem Statement:Find the shortest path between two nodes in an unweighted graph using BFS.

Description:

The graph is represented as an edge list.

The program should compute the shortest path in terms of number of edges from source to destination.

If no path exists, print "No Path".

Input Format:

First line: n m — number of vertices and edges.

Next m lines: u v — edge between u and v.

Last line: source destination.

Output Format:

A single line containing the sequence of vertices in the shortest path from source to destination.

If no path exists, print "No Path".

Sample Input:

5 40 10 21 32 40 4

Sample Output:

0 2 4

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem DSA Post-Midterm

Language Used: C

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
int vertex;
struct Node* next;
} Node;
typedef struct Queue {
int* data;
int front, rear, size, capacity;
} Queue;
Queue* createQueue(int capacity) {
Queue* q = (Queue*)malloc(sizeof(Queue));
q->data = (int*)malloc(capacity * sizeof(int));
q->front = 0;
q->rear = -1;
q->size = 0;
q->capacity = capacity;
return q;
}
void enqueue(Queue* q, int val) {
q->rear = (q->rear + 1) % q->capacity;
```

```
q->data[q->rear] = val;
q->size++;
}
int dequeue(Queue* q) {
int val = q->data[q->front];
q->front = (q->front + 1) % q->capacity;
q->size--;
return val;
}
int isEmpty(Queue* q) {
return q->size == 0;
}
void addEdge(Node** adj, int u, int v) {
Node* newNode = (Node*)malloc(sizeof(Node));
newNode->vertex = v;
newNode->next = NULL;
if (!adj[u]) {
adj[u] = newNode;
}
else {
Node* temp = adj[u];
while (temp->next) temp = temp->next;
temp->next = newNode;
}
}
int main() {
int n, m;
scanf("%d %d", &n, &m);
Node** adj = (Node**)malloc(n * sizeof(Node*));
for (int i = 0; i < n; i++) adj[i] = NULL;
for (int i = 0; i < m; i++) {
int u, v;
scanf("%d %d", &u, &v);
addEdge(adj, u, v);
addEdge(adj, v, u);
}
int src, dest;
scanf("%d %d", &src, &dest);
int* visited = (int*)calloc(n, sizeof(int));
int* parent = (int*)malloc(n * sizeof(int));
for (int i = 0; i < n; i++) parent[i] = -1;
Queue* q = createQueue(n);
enqueue(q, src);
visited[src] = 1;
int found = 0;
while (!isEmpty(q)) {
int curr = dequeue(q);
if (curr == dest) {
found = 1;
break;
}
Node* temp = adj[curr];
while (temp) {
int v = temp->vertex;
```

```
if (!visited[v]) {
enqueue(q, v);
visited[v] = 1;
parent[v] = curr;
}
temp = temp->next;
}
}
if (!found) {
printf("No Path\n");
}
else {
int path[n];
int idx = 0;
int curr = dest;
while (curr != -1) {
path[idx++] = curr;
curr = parent[curr];
}
for (int i = idx - 1; i >= 0; i--) {
if (i != idx - 1) printf(" ");
printf("%d", path[i]);
}printf("\n");
}
for (int i = 0; i < n; i++) {
Node* temp = adj[i];
while (temp) {
Node* next = temp->next;
free(temp);
temp = next;
}
}
free(adj);
free(visited);
free(parent);
free(q->data);
free(q);
return 0;
}
```


Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

No Path

Compilation Status: Passed

Execution Time:

0.001s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

0 2 5

Compilation Status: Passed

Execution Time:

0.001s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

0 1 2 3 4

Compilation Status: Passed

Execution Time:

0.001s

15. Detect Cycle in Undirected Graph Using DFS

Problem Statement:Check if an undirected graph contains a cycle using DFS.

Description:

Use DFS traversal to detect cycles in the graph.

If a cycle exists, print "Cycle Found", else print "No Cycle".

Input Format:

First line: n m — number of vertices and edges.

Next m lines: u v — edge between u and v.

Output Format:

Single line: "Cycle Found" or "No Cycle".

Sample Input:

4 40 11 22 33 0

Sample Output:

Cycle Found

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem DSA Post-Midterm

Language Used: C

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
int vertex;
struct Node* next;
} Node;
void addEdge(Node** adj, int u, int v) {
Node* newNode = (Node*)malloc(sizeof(Node));
newNode->vertex = v;
newNode->next = adj[u];
adj[u] = newNode;
}
int dfs(Node** adj, int v, int visited[], int parent) {
visited[v] = 1;
Node* temp = adj[v];
while (temp) {
int u = temp->vertex;
if (!visited[u]) {
if (dfs(adj, u, visited, v))
return 1;
}
else if (u != parent) {
```

```
return 1;
}
temp = temp->next;
}
return 0;
}
int main() {
int n, m;
scanf("%d %d", &n, &m);
Node** adj = (Node**)malloc(n * sizeof(Node*));
for (int i = 0; i < n; i++) adj[i] = NULL;
for (int i = 0; i < m; i++) {
int u, v;
scanf("%d %d", &u, &v);
addEdge(adj, u, v);
addEdge(adj, v, u);
}
int* visited = (int*)calloc(n, sizeof(int));
int cycleFound = 0;
for (int i = 0; i < n; i++) {
if (!visited[i]) {
if (dfs(adj, i, visited, -1)) {
cycleFound = 1;
break;   }
}
}
if (cycleFound)
printf("Cycle Found\n");
else
printf("No Cycle\n");
for (int i = 0; i < n; i++) {
Node* temp = adj[i];
while (temp) {
Node* next = temp->next;
free(temp);
temp = next;
}
}
free(adj);
free(visited);
return 0;
}
```


Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

No Cycle

Compilation Status: Passed

Execution Time:

0.001s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

Cycle Found

Compilation Status: Passed

Execution Time:

0.001s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

No Cycle

Compilation Status: Passed

Execution Time:

0.001s

16. Topological Sort Using DFS

Problem Statement:Perform topological sort of a Directed Acyclic Graph (DAG) using DFS.

Description:

The graph is directed and acyclic.

Use DFS to compute a valid topological ordering of the vertices.

Input Format:

First line: n m — number of vertices and edges.

Next m lines: u v — directed edge from u to v.

Output Format:

Single line: vertices in topological order (space-separated).

Sample Input:

6 65 25 04 04 12 33 1

Sample Output:

5 4 2 3 1 0

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem DSA Post-Midterm

Language Used: C

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
int vertex;
struct Node* next;
} Node;
void addEdge(Node** adj, int u, int v) {
Node* newNode = (Node*)malloc(sizeof(Node));
newNode->vertex = v;
newNode->next = adj[u];
adj[u] = newNode;
}
void dfs(Node** adj, int v, int visited[], int* stack, int* top) {
visited[v] = 1;
Node* temp = adj[v];
while (temp != NULL) {
if (!visited[temp->vertex])
dfs(adj, temp->vertex, visited, stack, top);
temp = temp->next;
}
```

```
stack[(*top)++] = v;
}
int main() {
int n, m;
scanf("%d %d", &n, &m);
Node** adj = (Node**)malloc(n * sizeof(Node*));
for (int i = 0; i < n; i++) adj[i] = NULL;
for (int i = 0; i < m; i++) {
int u, v;
scanf("%d %d", &u, &v);
addEdge(adj, u, v);
}
int* visited = (int*)calloc(n, sizeof(int));
int* stack = (int*)malloc(n * sizeof(int));
int top = 0;
for (int i = 0; i < n; i++) {
if (!visited[i])
dfs(adj, i, visited, stack, &top);
}
for (int i = top - 1; i >= 0; i--)
printf("%d%c", stack[i], i ? ' ' : '\n');
for (int i = 0; i < n; i++) {
Node* temp = adj[i];
while (temp != NULL) {
Node* next = temp->next;free(temp);
temp = next;
}
}
free(adj);
free(visited);
free(stack);
return 0;
}
```


Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

0 1 2 3

Compilation Status: Passed

Execution Time:

0.001s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

1 4 0 2 3

Compilation Status: Passed

Execution Time:

0.001s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

0 1 2

Compilation Status: Passed

Execution Time:

0.001s