**Module1**

**Water Connection Problem**

#include <stdio.h>

#include <limits.h>

#define MAX 1000

int start[MAX], end[MAX], diameter[MAX];

int visited[MAX];

void initialize(int n) {

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

start[i] = -1;

end[i] = -1;

diameter[i] = INT\_MAX;

visited[i] = 0;

}

}

void solve(int n) {

int count = 0;

int result[MAX][3];

for (int i = 1; i <= n; i++) {

if (end[i] == -1 && start[i] != -1) {

int curr = i;

int minDiameter = INT\_MAX;

while (start[curr] != -1) {

minDiameter = (minDiameter < diameter[curr]) ? minDiameter : diameter[curr];

curr = start[curr];

}

result[count][0] = i;

result[count][1] = curr;

result[count][2] = minDiameter;

count++;

}

}

printf("%d\n", count);

for (int i = 0; i < count; i++) {

printf("%d %d %d\n", result[i][0], result[i][1], result[i][2]);

}

}

int main() {

int n = 9, p = 6;

int a[] = {7, 5, 4, 2, 9, 3};

int b[] = {4, 9, 6, 8, 7, 1};

int d[] = {98, 72, 10, 22, 17, 66};

initialize(n);

for (int i = 0; i < p; i++) {

start[a[i]] = b[i];

diameter[a[i]] = d[i];

end[b[i]] = a[i];

}

printf("\nOutput:\n");

solve(n);

return 0;

}

**Gold Mine Problem**

#include <stdio.h>

#include <string.h>

#define MAX 100

// Function to get maximum of three integers

int max(int a, int b, int c) {

if (a > b && a > c) return a;

if (b > c) return b;

return c;

}

// Function to find the maximum gold collected

int getMaxGold(int gold[MAX][MAX], int n, int m) {

int dp[MAX][MAX];

memset(dp, 0, sizeof(dp));

// Fill the DP array starting from the last column

for (int col = m - 1; col >= 0; col--) {

for (int row = 0; row < n; row++) {

// Possible moves

int right = (col == m - 1) ? 0 : dp[row][col + 1];

int right\_up = (row == 0 || col == m - 1) ? 0 : dp[row - 1][col + 1];

int right\_down = (row == n - 1 || col == m - 1) ? 0 : dp[row + 1][col + 1];

// DP state transition

dp[row][col] = gold[row][col] + max(right, right\_up, right\_down);

}

}

// Find the maximum collected gold in the first column

int maxGold = dp[0][0];

for (int i = 1; i < n; i++) {

if (dp[i][0] > maxGold) {

maxGold = dp[i][0];

}

}

return maxGold;

}

// Driver Code

int main() {

int gold[MAX][MAX] = {

{1, 3, 1, 5},

{2, 2, 4, 1},

{5, 0, 2, 3},

{0, 6, 1, 2}

};

int n = 4, m = 4; // Grid size

printf("Maximum gold collected: %d\n", getMaxGold(gold, n, m));

return 0;

}

**Module2**

**Java Regex 2 - Duplicate Words**

import java.util.Scanner;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class DuplicateWords {

public static void main(String[] args) {

String regex = "\\b(\\w+)(?:\\W+\\1\\b)+";

Pattern p = Pattern.compile(regex, Pattern.CASE\_INSENSITIVE);

Scanner in = new Scanner(System.in);

int numSentences = Integer.parseInt(in.nextLine());

while (numSentences-- > 0) {

String input = in.nextLine();

Matcher m = p.matcher(input);

// Check for subsequences of input that match the compiled pattern

while (m.find()) {

input = input.replaceAll(m.group(), m.group(1));

}

// Prints the modified sentence.

System.out.println(input);

}

in.close();

}

}

**Sample Input**

5

Goodbye bye bye world world world

Sam went went to to to his business

Reya is is the the best player in eye eye game

in inthe

Hello hello Ab aB

**Sample Output**

Goodbye bye world

Sam went to his business

Reya is the best player in eye game

in inthe

Hello Ab

**Java Reflection - Attributes**

import java.lang.reflect.\*;

import java.util.\*;

class Student {

private String name;

private String id;

private String email;

public String getName() { return name; }

public String getId() { return id; }

public String getEmail() { return email; }

public void setName(String name) { this.name = name; }

public void setId(String id) { this.id = id; }

public void setEmail(String email) { this.email = email; }

}

public class Solution {

public static void main(String[] args) {

Class student = Student.class;

Field[] fields = student.getDeclaredFields();

List<String> fieldNames = new ArrayList<>();

for (Field field : fields) {

fieldNames.add(field.getName());

}

Collections.sort(fieldNames);

for (String name : fieldNames) {

System.out.println(name);

}

// Display methods

Method[] methods = student.getDeclaredMethods();

List<String> methodNames = **new** ArrayList<>();

**for** (Method method : methods) {

methodNames.add(method.getName());

}

Collections.*sort*(methodNames);

**for** (String name : methodNames) {

System.***out***.println(name);

}

}

}

**Module3**

**2. Merge Sort for Linked List**

#include <stdio.h>

#include <stdlib.h>

// Definition of the linked list node

struct Node {

int data;

struct Node\* next;

};

// Function to split the linked list into two halves

void splitList(struct Node\* head, struct Node\*\* front, struct Node\*\* back) {

struct Node\* slow = head;

struct Node\* fast = head->next;

// Move 'slow' one step and 'fast' two steps until 'fast' reaches the end

while (fast != NULL) {

fast = fast->next;

if (fast != NULL) {

slow = slow->next;

fast = fast->next;

}

}

// Split the list into two halves

\*front = head;

\*back = slow->next;

slow->next = NULL;

}

// Function to merge two sorted linked lists

struct Node\* mergeLists(struct Node\* list1, struct Node\* list2) {

// If one list is empty, return the other list

if (list1 == NULL) return list2;

if (list2 == NULL) return list1;

struct Node\* result = NULL;

// Merge the two lists

if (list1->data <= list2->data) {

result = list1;

result->next = mergeLists(list1->next, list2);

} else {

result = list2;

result->next = mergeLists(list1, list2->next);

}

return result;

}

// Merge Sort function

struct Node\* mergeSort(struct Node\* head) {

// Base case: If the list is empty or has only one element, return it

if (head == NULL || head->next == NULL) {

return head;

}

struct Node\* front = NULL;

struct Node\* back = NULL;

// Split the list into two halves

splitList(head, &front, &back);

// Recursively sort the two halves

front = mergeSort(front);

back = mergeSort(back);

// Merge the sorted halves

return mergeLists(front, back);

}

// Function to insert a node at the beginning of the list

void push(struct Node\*\* head\_ref, int new\_data) {

struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));

new\_node->data = new\_data;

new\_node->next = \*head\_ref;

\*head\_ref = new\_node;

}

// Function to print the linked list

void printList(struct Node\* head) {

struct Node\* temp = head;

while (temp != NULL) {

printf("%d ", temp->data);

temp = temp->next;

}

printf("\n");

}

// Function to read input from the user

void readInput(struct Node\*\* head\_ref) {

int n, data;

printf("Enter the number of elements in the linked list: ");

scanf("%d", &n);

printf("Enter the elements of the linked list:\n");

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

scanf("%d", &data);

push(head\_ref, data); // Insert the element into the linked list

}

}

int main() {

struct Node\* head = NULL;

// Read input from the user to create the linked list

readInput(&head);

// Sorting the linked list using Merge Sort

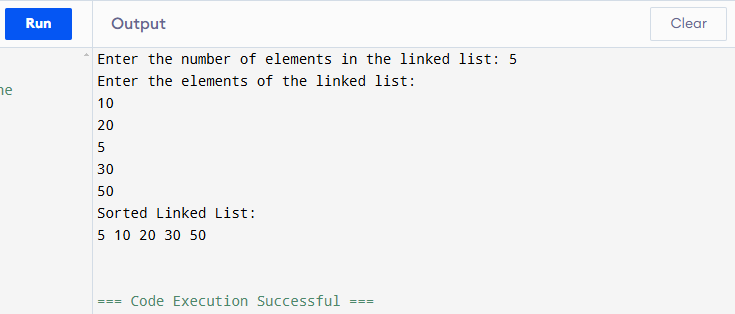
head = mergeSort(head);

printf("Sorted Linked List:\n");

printList(head);

return 0;

}



**2. Split a Circular linked list into two halves**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

void splitList(struct Node\* head, struct Node\*\* head1, struct Node\*\* head2) {

if (!head || head->next == head) {

\*head1 = head;

\*head2 = NULL;

return;

}

struct Node \*slow = head, \*fast = head;

while (fast->next != head && fast->next->next != head) {

slow = slow->next;

fast = fast->next->next;

}

if (fast->next->next == head) fast = fast->next;

\*head1 = head;

\*head2 = slow->next;

slow->next = \*head1;

fast->next = \*head2;

}

void push(struct Node\*\* head, int data) {

struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));

new\_node->data = data;

new\_node->next = \*head ? \*head : new\_node;

if (\*head) {

struct Node\* temp = \*head;

while (temp->next != \*head) temp = temp->next;

temp->next = new\_node;

}

\*head = new\_node;

}

void printList(struct Node\* head) {

if (!head) return;

struct Node\* temp = head;

do {

printf("%d ", temp->data);

temp = temp->next;

} while (temp != head);

printf("\n");

}

int main() {

struct Node \*head = NULL, \*head1 = NULL, \*head2 = NULL;

for (int i = 6; i > 0; i--) push(&head, i);

printf("Original List:\n");

printList(head);

splitList(head, &head1, &head2);

printf("\nFirst Half:\n");

printList(head1);

printf("\nSecond Half:\n");

printList(head2);

return 0;

}

**Module4**

**1.Find the middle element of a stack**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100 // Define maximum size of stack

// Stack structure

typedef struct {

int arr[MAX\_SIZE];

int top;

} Stack;

// Function to initialize stack

void init(Stack\* stack) {

stack->top = -1;

}

// Push operation

void push(Stack\* stack, int data) {

if (stack->top == MAX\_SIZE - 1) {

printf("Stack overflow\n");

return;

}

stack->arr[++stack->top] = data;

}

// Pop operation

int pop(Stack\* stack) {

if (stack->top == -1) {

printf("Stack underflow\n");

return -1;

}

return stack->arr[stack->top--];

}

// Function to get the middle element

int getMiddle(Stack\* stack) {

if (stack->top == -1) {

printf("Stack is empty\n");

return -1;

}

return stack->arr[stack->top / 2]; // Middle index calculation

}

// Main function

int main() {

Stack stack;

init(&stack);

push(&stack, 10);

push(&stack, 20);

push(&stack, 30);

push(&stack, 40);

push(&stack, 50);

printf("Middle Element: %d\n", getMiddle(&stack));

pop(&stack);

printf("Middle Element after pop: %d\n", getMiddle(&stack));

return 0;

}

**2.The celebrity Problem**

#include <stdio.h>

#include <stdbool.h>

#define N 4 // Number of people

// Mock knows function (should be given or implemented based on input)

int MATRIX[N][N] = {

{0, 1, 1, 1},

{0, 0, 0, 1},

{0, 1, 0, 1},

{0, 0, 0, 0} // Person 3 is the celebrity

};

// Function that returns whether A knows B

int knows(int a, int b) {

return MATRIX[a][b];

}

// Function to find the celebrity

int findCelebrity(int n) {

int candidate = 0;

// Step 1: Find the potential celebrity

for (int i = 1; i < n; i++) {

if (knows(candidate, i)) {

candidate = i; // Candidate cannot be a celebrity

}

}

// Step 2: Verify if the candidate is a real celebrity

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

if (i != candidate) {

if (knows(candidate, i) || !knows(i, candidate)) {

return -1; // No celebrity exists

}

}

}

return candidate; // Found a celebrity

}

// Main function

int main() {

int celebrity = findCelebrity(N);

if (celebrity == -1)

printf("No Celebrity found\n");

else

printf("Celebrity is Person %d\n", celebrity);

return 0;

}

**Module5**

**1. Check if two trees are mirror of each other**

#include <stdio.h>

#include <stdlib.h>

// Definition of a binary tree node

typedef struct Node {

int data;

struct Node \*left, \*right;

} Node;

// Function to create a new node

Node\* createNode(int data) {

Node\* newNode = (Node\*)malloc(sizeof(Node));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

// Function to check if two trees are mirror of each other

int areMirror(Node\* tree1, Node\* tree2) {

if (tree1 == NULL && tree2 == NULL)

return 1;

if (tree1 == NULL || tree2 == NULL)

return 0;

return (tree1->data == tree2->data) &&

areMirror(tree1->left, tree2->right) &&

areMirror(tree1->right, tree2->left);

}

// Main function to test the above implementation

int main() {

Node \*tree1 = createNode(1);

tree1->left = createNode(2);

tree1->right = createNode(3);

tree1->left->left = createNode(4);

tree1->left->right = createNode(5);

Node \*tree2 = createNode(1);

tree2->left = createNode(3);

tree2->right = createNode(2);

tree2->right->left = createNode(5);

tree2->right->right = createNode(4);

if (areMirror(tree1, tree2))

printf("The trees are mirror images of each other.\n");

else

printf("The trees are not mirror images of each other.\n");

return 0;

}

**2. Check whether BST contains Dead end**

#include <stdio.h>

#include <stdlib.h>

// Definition of a binary tree node

typedef struct Node {

int data;

struct Node \*left, \*right;

} Node;

// Function to create a new node

Node\* createNode(int data) {

Node\* newNode = (Node\*)malloc(sizeof(Node));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

// Function to insert a node in BST

Node\* insert(Node\* root, int data) {

if (root == NULL)

return createNode(data);

if (data < root->data)

root->left = insert(root->left, data);

else

root->right = insert(root->right, data);

return root;

}

// Function to check for dead end

int checkDeadEnd(Node\* root, int min, int max) {

if (root == NULL)

return 0;

// If leaf node and min == max, dead end found

if (min == max)

return 1;

// Recursively check for dead end in left and right subtrees

return checkDeadEnd(root->left, min, root->data - 1) ||

checkDeadEnd(root->right, root->data + 1, max);

}

// Function to check if BST contains a dead end

int hasDeadEnd(Node\* root) {

return checkDeadEnd(root, 1, \_\_INT\_MAX\_\_);

}

// Main function to test the implementation

int main() {

Node\* root = NULL;

root = insert(root, 8);

root = insert(root, 5);

root = insert(root, 2);

root = insert(root, 3);

root = insert(root, 7);

root = insert(root, 11);

root = insert(root, 4);

if (hasDeadEnd(root))

printf("The BST contains a dead end.\n");

else

printf("The BST does not contain a dead end.\n");

return 0;

}