**OSDS FINAL LAB QUESTIONS**

1. Write a program to insert N employee information into the Linked List.

a. Write a function to Search an Employee and display his details.

b. Display the Average salary of all the Employees with designation Manager.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Define the Employee structure

struct Employee {

char name[50];

char designation[50];

float salary;

struct Employee\* next;

};

// Function to insert an employee into the linked list

void insertEmployee(struct Employee\*\* head, char name[], char designation[], float salary) {

struct Employee\* newEmployee = (struct Employee\*)malloc(sizeof(struct Employee));

strcpy(newEmployee->name, name);

strcpy(newEmployee->designation, designation);

newEmployee->salary = salary;

newEmployee->next = NULL;

if (\*head == NULL) {

\*head = newEmployee;

} else {

struct Employee\* current = \*head;

while (current->next != NULL) {

current = current->next;

}

current->next = newEmployee;

}

}

// Function to search for an employee by name and display their details

void searchEmployee(struct Employee\* head, char name[]) {

struct Employee\* current = head;

int found = 0;

while (current != NULL) {

if (strcmp(current->name, name) == 0) {

printf("Employee found:\n");

printf("Name: %s\n", current->name);

printf("Designation: %s\n", current->designation);

printf("Salary: %.2f\n", current->salary);

found = 1;

break;

}

current = current->next;

}

if (!found) {

printf("Employee with name %s not found.\n", name);

}

}

// Function to calculate the average salary of employees with the designation "Manager"

float calculateAverageManagerSalary(struct Employee\* head) {

int count = 0;

float totalSalary = 0.0;

struct Employee\* current = head;

while (current != NULL) {

if (strcmp(current->designation, "Manager") == 0) {

totalSalary += current->salary;

count++;

}

current = current->next;

}

if (count > 0) {

return totalSalary / count;

} else {

return 0.0;

}

}

int main() {

struct Employee\* head = NULL;

int N;

printf("Enter the number of employees (N): ");

scanf("%d", &N);

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

char name[50], designation[50];

float salary;

printf("\nEnter details for Employee %d:\n", i + 1);

printf("Name: ");

scanf("%s", name);

printf("Designation: ");

scanf("%s", designation);

printf("Salary: ");

scanf("%f", &salary);

insertEmployee(&head, name, designation, salary);

}

char searchName[50];

printf("\nEnter the name of the employee to search: ");

scanf("%s", searchName);

searchEmployee(head, searchName);

float averageSalary = calculateAverageManagerSalary(head);

printf("\nAverage salary of Managers: %.2f\n", averageSalary);

return 0;

}

2. Consider two liked list A and B in sorted order. Write a program to merge the linked list A and B such that the final list C is in sorted order.

#include <stdio.h>

#include <stdlib.h>

// Node structure for a linked list

struct Node {

int data;

struct Node\* next;

};

// Function to insert a new node at the end of a linked list

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

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

newNode->data = data;

newNode->next = NULL;

if (\*head == NULL) {

\*head = newNode;

} else {

struct Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

}

// Function to merge two sorted linked lists A and B into a new sorted list C

struct Node\* mergeSortedLists(struct Node\* A, struct Node\* B) {

struct Node\* result = NULL;

// Base cases

if (A == NULL) return B;

if (B == NULL) return A;

// Choose the smaller value and recur

if (A->data <= B->data) {

result = A;

result->next = mergeSortedLists(A->next, B);

} else {

result = B;

result->next = mergeSortedLists(A, B->next);

}

return result;

}

// Function to print a linked list

void printList(struct Node\* head) {

while (head != NULL) {

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

head = head->next;

}

printf("\n");

}

int main() {

// Sample linked lists A and B

struct Node\* A = NULL;

struct Node\* B = NULL;

insertNode(&A, 1);

insertNode(&A, 3);

insertNode(&A, 5);

insertNode(&B, 2);

insertNode(&B, 4);

insertNode(&B, 6);

// Merge the linked lists A and B into a new sorted list C

struct Node\* C = mergeSortedLists(A, B);

// Print the merged list C

printf("Merged Sorted List: ");

printList(C);

return 0;

}

3. Consider two liked list A of size N. Write a program to Split the linked list two parts i.e. B and C each of size N/2.

#include <stdio.h>

#include <stdlib.h>

// Node structure for a linked list

struct Node {

int data;

struct Node\* next;

};

// Function to insert a new node at the end of a linked list

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

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

newNode->data = data;

newNode->next = NULL;

if (\*head == NULL) {

\*head = newNode;

} else {

struct Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

}

// Function to split a linked list into two parts, B and C

void splitLinkedList(struct Node\* A, struct Node\*\* B, struct Node\*\* C) {

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

// A has fewer than two nodes, cannot split

\*B = A;

\*C = NULL;

return;

}

struct Node\* slow = A;

struct Node\* fast = A->next;

// Move 'fast' two steps and 'slow' one step at a time

while (fast != NULL) {

fast = fast->next;

if (fast != NULL) {

slow = slow->next;

fast = fast->next;

}

}

// 'slow' is at the middle of the list, split the list into B and C

\*B = A;

\*C = slow->next;

slow->next = NULL;

}

// Function to print a linked list

void printList(struct Node\* head) {

while (head != NULL) {

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

head = head->next;

}

printf("\n");

}

int main() {

// Sample linked list A

struct Node\* A = NULL;

// Insert nodes into the linked list

for (int i = 1; i <= 6; ++i) {

insertNode(&A, i);

}

// Initialize linked lists B and C

struct Node\* B = NULL;

struct Node\* C = NULL;

// Split the linked list A into B and C

splitLinkedList(A, &B, &C);

// Print the original and split lists

printf("Original List: ");

printList(A);

printf("List B: ");

printList(B);

printf("List C: ");

printList(C);

return 0;

}

4. Write a program to create a single linked list. The address of the head is to be stored in a separate structure which has two fields

struct head {

struct node \*head\_ptr;

int num;

};

The head\_ptr will store the pointer to the head node num: Is the count of the number of elements present in the linked list. Write implementations for insertion and deletion which will update the head structure appropriately

#include <stdio.h>

#include <stdlib.h>

// Node structure for a linked list

struct Node {

int data;

struct Node\* next;

};

// Head structure to store the head pointer and the count of elements

struct Head {

struct Node\* head\_ptr;

int num;

};

// Function to initialize the head structure

struct Head\* initializeHead() {

struct Head\* head = (struct Head\*)malloc(sizeof(struct Head));

head->head\_ptr = NULL;

head->num = 0;

return head;

}

// Function to insert a new node at the end of the linked list

void insertNode(struct Head\* head, int data) {

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

newNode->data = data;

newNode->next = NULL;

if (head->head\_ptr == NULL) {

// If the list is empty, update head pointer

head->head\_ptr = newNode;

} else {

// Traverse the list to find the last node

struct Node\* temp = head->head\_ptr;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

// Update the count of elements

head->num++;

}

// Function to delete a node with a given value from the linked list

void deleteNode(struct Head\* head, int data) {

if (head->head\_ptr == NULL) {

// List is empty, nothing to delete

return;

}

struct Node\* current = head->head\_ptr;

struct Node\* prev = NULL;

// Search for the node with the given data

while (current != NULL && current->data != data) {

prev = current;

current = current->next;

}

if (current == NULL) {

// Node with given data not found

return;

}

// Update the links to skip the node to be deleted

if (prev == NULL) {

// If the node to be deleted is the first node

head->head\_ptr = current->next;

} else {

prev->next = current->next;

}

// Free the memory of the deleted node

free(current);

// Update the count of elements

head->num--;

}

// Function to print the linked list

void printList(struct Head\* head) {

struct Node\* temp = head->head\_ptr;

while (temp != NULL) {

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

temp = temp->next;

}

printf("\n");

}

int main() {

// Initialize the head structure

struct Head\* head = initializeHead();

// Insert elements into the linked list

insertNode(head, 1);

insertNode(head, 2);

insertNode(head, 3);

// Print the initial linked list

printf("Initial List: ");

printList(head);

// Delete an element from the linked list

deleteNode(head, 2);

// Print the modified linked list after deletion

printf("List after deletion: ");

printList(head);

// Free the memory allocated for the head structure

free(head);

return 0;

}

5. WAP to implement two stacks in one array A[1 .. N] in such a way that neither stack overflows unless the total number of elements in both stacks together is N.

#include <stdio.h>

#include <stdlib.h>

// Structure to represent the two stacks

struct TwoStacks {

int\* array;

int capacity;

int top1; // Top index for Stack 1

int top2; // Top index for Stack 2

};

// Function to initialize the two stacks

struct TwoStacks\* initializeTwoStacks(int capacity) {

struct TwoStacks\* twoStacks = (struct TwoStacks\*)malloc(sizeof(struct TwoStacks));

twoStacks->capacity = capacity;

twoStacks->array = (int\*)malloc(capacity \* sizeof(int));

twoStacks->top1 = -1; // Initialize top index for Stack 1

twoStacks->top2 = capacity; // Initialize top index for Stack 2

return twoStacks;

}

// Function to push an element onto Stack 1

void pushStack1(struct TwoStacks\* twoStacks, int data) {

if (twoStacks->top1 < twoStacks->top2 - 1) {

twoStacks->array[++twoStacks->top1] = data;

} else {

printf("Stack Overflow: Cannot push element onto Stack 1\n");

}

}

// Function to push an element onto Stack 2

void pushStack2(struct TwoStacks\* twoStacks, int data) {

if (twoStacks->top1 < twoStacks->top2 - 1) {

twoStacks->array[--twoStacks->top2] = data;

} else {

printf("Stack Overflow: Cannot push element onto Stack 2\n");

}

}

// Function to pop an element from Stack 1

int popStack1(struct TwoStacks\* twoStacks) {

if (twoStacks->top1 >= 0) {

return twoStacks->array[twoStacks->top1--];

} else {

printf("Stack 1 is empty\n");

return -1; // Invalid value to indicate an empty stack

}

}

// Function to pop an element from Stack 2

int popStack2(struct TwoStacks\* twoStacks) {

if (twoStacks->top2 < twoStacks->capacity) {

return twoStacks->array[twoStacks->top2++];

} else {

printf("Stack 2 is empty\n");

return -1; // Invalid value to indicate an empty stack

}

}

// Function to print the elements of Stack 1

void printStack1(struct TwoStacks\* twoStacks) {

printf("Stack 1: ");

for (int i = 0; i <= twoStacks->top1; ++i) {

printf("%d ", twoStacks->array[i]);

}

printf("\n");

}

// Function to print the elements of Stack 2

void printStack2(struct TwoStacks\* twoStacks) {

printf("Stack 2: ");

for (int i = twoStacks->capacity - 1; i >= twoStacks->top2; --i) {

printf("%d ", twoStacks->array[i]);

}

printf("\n");

}

int main() {

// Initialize two stacks in one array with a capacity of 6

struct TwoStacks\* twoStacks = initializeTwoStacks(6);

// Push elements onto Stack 1

pushStack1(twoStacks, 1);

pushStack1(twoStacks, 2);

pushStack1(twoStacks, 3);

// Push elements onto Stack 2

pushStack2(twoStacks, 4);

pushStack2(twoStacks, 5);

pushStack2(twoStacks, 6);

// Print the elements of both stacks

printStack1(twoStacks);

printStack2(twoStacks);

// Pop elements from both stacks

int popped1 = popStack1(twoStacks);

int popped2 = popStack2(twoStacks);

// Print the elements of both stacks after popping

printf("Popped from Stack 1: %d\n", popped1);

printf("Popped from Stack 2: %d\n", popped2);

printStack1(twoStacks);

printStack2(twoStacks);

// Free the memory allocated for the two stacks

free(twoStacks->array);

free(twoStacks);

return 0;

}

6. WAP to implement stack operations PUSH and POP using 2 Queues.

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in a queue

struct QueueNode {

int data;

struct QueueNode\* next;

};

// Structure to represent a queue

struct Queue {

struct QueueNode\* front;

struct QueueNode\* rear;

};

// Function to initialize a queue

struct Queue\* initializeQueue() {

struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

queue->front = queue->rear = NULL;

return queue;

}

// Function to check if a queue is empty

int isEmpty(struct Queue\* queue) {

return (queue->front == NULL);

}

// Function to enqueue an element into a queue

void enqueue(struct Queue\* queue, int data) {

struct QueueNode\* newNode = (struct QueueNode\*)malloc(sizeof(struct QueueNode));

newNode->data = data;

newNode->next = NULL;

if (isEmpty(queue)) {

queue->front = queue->rear = newNode;

} else {

queue->rear->next = newNode;

queue->rear = newNode;

}

}

// Function to dequeue an element from a queue

int dequeue(struct Queue\* queue) {

if (isEmpty(queue)) {

printf("Queue is empty\n");

return -1; // Invalid value to indicate an empty queue

}

struct QueueNode\* temp = queue->front;

int data = temp->data;

if (queue->front == queue->rear) {

// Only one element in the queue

queue->front = queue->rear = NULL;

} else {

queue->front = temp->next;

}

free(temp);

return data;

}

// Function to implement stack using two queues

struct Stack {

struct Queue\* q1;

struct Queue\* q2;

};

// Function to initialize a stack

struct Stack\* initializeStack() {

struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

stack->q1 = initializeQueue();

stack->q2 = initializeQueue();

return stack;

}

// Function to push an element onto the stack

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

// Enqueue the new element to the non-empty queue

if (!isEmpty(stack->q1)) {

enqueue(stack->q1, data);

} else {

enqueue(stack->q2, data);

}

}

// Function to pop an element from the stack

int pop(struct Stack\* stack) {

// Dequeue all elements from the non-empty queue except the last one

struct Queue\* nonEmptyQueue = isEmpty(stack->q1) ? stack->q2 : stack->q1;

struct Queue\* emptyQueue = isEmpty(stack->q1) ? stack->q1 : stack->q2;

if (isEmpty(nonEmptyQueue)) {

printf("Stack is empty\n");

return -1; // Invalid value to indicate an empty stack

}

while (nonEmptyQueue->front->next != NULL) {

enqueue(emptyQueue, dequeue(nonEmptyQueue));

}

// Dequeue the last element from the non-empty queue

int poppedData = dequeue(nonEmptyQueue);

return poppedData;

}

int main() {

// Initialize a stack using two queues

struct Stack\* stack = initializeStack();

// Push elements onto the stack

push(stack, 1);

push(stack, 2);

push(stack, 3);

// Pop elements from the stack

int popped1 = pop(stack);

int popped2 = pop(stack);

int popped3 = pop(stack);

// Print the popped elements

printf("Popped from stack: %d, %d, %d\n", popped1, popped2, popped3);

// Free the memory allocated for the stack

free(stack->q1);

free(stack->q2);

free(stack);

return 0;

}

7. WAP to implement Queue operations INSERT and DELETE using 2 stacks.

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in a stack

struct StackNode {

int data;

struct StackNode\* next;

};

// Structure to represent a stack

struct Stack {

struct StackNode\* top;

};

// Function to initialize a stack

struct Stack\* initializeStack() {

struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

stack->top = NULL;

return stack;

}

// Function to check if a stack is empty

int isEmpty(struct Stack\* stack) {

return (stack->top == NULL);

}

// Function to push an element onto a stack

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

struct StackNode\* newNode = (struct StackNode\*)malloc(sizeof(struct StackNode));

newNode->data = data;

newNode->next = stack->top;

stack->top = newNode;

}

// Function to pop an element from a stack

int pop(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack is empty\n");

return -1; // Invalid value to indicate an empty stack

}

struct StackNode\* temp = stack->top;

int data = temp->data;

stack->top = temp->next;

free(temp);

return data;

}

// Structure to represent a queue using two stacks

struct Queue {

struct Stack\* stack1; // For enqueue (insert) operation

struct Stack\* stack2; // For dequeue (delete) operation

};

// Function to initialize a queue

struct Queue\* initializeQueue() {

struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

queue->stack1 = initializeStack();

queue->stack2 = initializeStack();

return queue;

}

// Function to enqueue (insert) an element into a queue

void enqueue(struct Queue\* queue, int data) {

// Push all elements from stack2 to stack1

while (!isEmpty(queue->stack2)) {

push(queue->stack1, pop(queue->stack2));

}

// Push the new element onto stack1

push(queue->stack1, data);

}

// Function to dequeue (delete) an element from a queue

int dequeue(struct Queue\* queue) {

// If both stacks are empty, the queue is empty

if (isEmpty(queue->stack1) && isEmpty(queue->stack2)) {

printf("Queue is empty\n");

return -1; // Invalid value to indicate an empty queue

}

// Push all elements from stack1 to stack2

while (!isEmpty(queue->stack1)) {

push(queue->stack2, pop(queue->stack1));

}

// Pop the element from stack2 (this was the front of the queue)

return pop(queue->stack2);

}

int main() {

// Initialize a queue using two stacks

struct Queue\* queue = initializeQueue();

// Enqueue elements into the queue

enqueue(queue, 1);

enqueue(queue, 2);

enqueue(queue, 3);

// Dequeue elements from the queue

int dequeued1 = dequeue(queue);

int dequeued2 = dequeue(queue);

int dequeued3 = dequeue(queue);

// Print the dequeued elements

printf("Dequeued from queue: %d, %d, %d\n", dequeued1, dequeued2, dequeued3);

// Free the memory allocated for the queue

free(queue->stack1);

free(queue->stack2);

free(queue);

return 0;

}

8. Write a program to insert N Student information into the Doubly Linked List. Write a function to

a. Search a Student and display his details.

b. Display the details of the students who have scored above 90 in Math’s and Science.

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a student

struct Student {

int rollNumber;

char name[50];

float mathScore;

float scienceScore;

struct Student\* prev;

struct Student\* next;

};

// Function to insert a student into the doubly linked list

void insertStudent(struct Student\*\* head, int rollNumber, const char\* name, float mathScore, float scienceScore) {

struct Student\* newStudent = (struct Student\*)malloc(sizeof(struct Student));

newStudent->rollNumber = rollNumber;

strcpy(newStudent->name, name);

newStudent->mathScore = mathScore;

newStudent->scienceScore = scienceScore;

newStudent->prev = NULL;

newStudent->next = NULL;

if (\*head == NULL) {

\*head = newStudent;

} else {

struct Student\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newStudent;

newStudent->prev = temp;

}

}

// Function to search for a student by roll number and display details

void searchStudent(struct Student\* head, int rollNumber) {

struct Student\* current = head;

while (current != NULL) {

if (current->rollNumber == rollNumber) {

printf("Student Found!\n");

printf("Roll Number: %d\n", current->rollNumber);

printf("Name: %s\n", current->name);

printf("Math Score: %.2f\n", current->mathScore);

printf("Science Score: %.2f\n", current->scienceScore);

return;

}

current = current->next;

}

printf("Student with Roll Number %d not found.\n", rollNumber);

}

// Function to display details of students who scored above 90 in Math and Science

void displayHighScorers(struct Student\* head) {

struct Student\* current = head;

printf("Students who scored above 90 in Math and Science:\n");

while (current != NULL) {

if (current->mathScore > 90 && current->scienceScore > 90) {

printf("Roll Number: %d\n", current->rollNumber);

printf("Name: %s\n", current->name);

printf("Math Score: %.2f\n", current->mathScore);

printf("Science Score: %.2f\n", current->scienceScore);

printf("\n");

}

current = current->next;

}

}

// Function to display details of all students in the doubly linked list

void displayAllStudents(struct Student\* head) {

struct Student\* current = head;

while (current != NULL) {

printf("Roll Number: %d\n", current->rollNumber);

printf("Name: %s\n", current->name);

printf("Math Score: %.2f\n", current->mathScore);

printf("Science Score: %.2f\n", current->scienceScore);

printf("\n");

current = current->next;

}

}

// Function to free memory allocated for the doubly linked list

void freeStudents(struct Student\* head) {

struct Student\* current = head;

while (current != NULL) {

struct Student\* next = current->next;

free(current);

current = next;

}

}

int main() {

struct Student\* students = NULL;

// Insert N student information into the doubly linked list

insertStudent(&students, 1, "John", 95.5, 92.0);

insertStudent(&students, 2, "Alice", 88.0, 94.5);

insertStudent(&students, 3, "Bob", 90.0, 91.5);

// Add more students as needed

// Search for a student by roll number and display details

searchStudent(students, 2);

// Display details of students who scored above 90 in Math and Science

displayHighScorers(students);

// Display details of all students

displayAllStudents(students);

// Free memory allocated for the doubly linked list

freeStudents(students);

return 0;

}

9. WAP program to create binary tree and traverse the tree using Inorder, PreOrder, Post Order Traversing Technique.

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in the binary tree

struct TreeNode {

int data;

struct TreeNode\* left;

struct TreeNode\* right;

};

// Function to create a new node with the given data

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Function to insert a node into the binary tree

struct TreeNode\* insertNode(struct TreeNode\* root, int data) {

if (root == NULL) {

return createNode(data);

}

if (data < root->data) {

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

} else if (data > root->data) {

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

}

return root;

}

// Inorder traversal of the binary tree

void inorderTraversal(struct TreeNode\* root) {

if (root != NULL) {

inorderTraversal(root->left);

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

inorderTraversal(root->right);

}

}

// Preorder traversal of the binary tree

void preorderTraversal(struct TreeNode\* root) {

if (root != NULL) {

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

preorderTraversal(root->left);

preorderTraversal(root->right);

}

}

// Postorder traversal of the binary tree

void postorderTraversal(struct TreeNode\* root) {

if (root != NULL) {

postorderTraversal(root->left);

postorderTraversal(root->right);

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

}

}

int main() {

// Create a binary tree

struct TreeNode\* root = NULL;

root = insertNode(root, 50);

insertNode(root, 30);

insertNode(root, 70);

insertNode(root, 20);

insertNode(root, 40);

insertNode(root, 60);

insertNode(root, 80);

// Traverse the binary tree using different techniques

printf("Inorder Traversal: ");

inorderTraversal(root);

printf("\n");

printf("Preorder Traversal: ");

preorderTraversal(root);

printf("\n");

printf("Postorder Traversal: ");

postorderTraversal(root);

printf("\n");

// Free the memory allocated for the binary tree (optional)

// You may want to implement a proper tree deletion function if needed

return 0;

}

10. WAP to create and delete a node from a binary Search tree. Traverse the BST using Level Order Traversing Technique.

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in the binary search tree

struct TreeNode {

int data;

struct TreeNode\* left;

struct TreeNode\* right;

};

// Function to create a new node with the given data

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Function to insert a node into the binary search tree

struct TreeNode\* insertNode(struct TreeNode\* root, int data) {

if (root == NULL) {

return createNode(data);

}

if (data < root->data) {

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

} else if (data > root->data) {

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

}

return root;

}

// Function to find the node with the minimum value in a BST

struct TreeNode\* findMinNode(struct TreeNode\* node) {

while (node->left != NULL) {

node = node->left;

}

return node;

}

// Function to delete a node with a given key from the binary search tree

struct TreeNode\* deleteNode(struct TreeNode\* root, int key) {

if (root == NULL) {

return root;

}

if (key < root->data) {

root->left = deleteNode(root->left, key);

} else if (key > root->data) {

root->right = deleteNode(root->right, key);

} else {

// Node with only one child or no child

if (root->left == NULL) {

struct TreeNode\* temp = root->right;

free(root);

return temp;

} else if (root->right == NULL) {

struct TreeNode\* temp = root->left;

free(root);

return temp;

}

// Node with two children, get the inorder successor (smallest in the right subtree)

struct TreeNode\* temp = findMinNode(root->right);

// Copy the inorder successor's data to this node

root->data = temp->data;

// Delete the inorder successor

root->right = deleteNode(root->right, temp->data);

}

return root;

}

// Function to perform Level Order Traversal of a binary search tree

void levelOrderTraversal(struct TreeNode\* root) {

if (root == NULL) {

return;

}

// Use a queue for Level Order Traversal

struct TreeNode\* queue[100];

int front = -1, rear = -1;

queue[++rear] = root;

while (front != rear) {

struct TreeNode\* current = queue[++front];

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

if (current->left != NULL) {

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

}

if (current->right != NULL) {

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

}

}

}

// Function to free memory allocated for the binary search tree

void freeBST(struct TreeNode\* root) {

if (root == NULL) {

return;

}

freeBST(root->left);

freeBST(root->right);

free(root);

}

int main() {

// Create a binary search tree

struct TreeNode\* root = NULL;

root = insertNode(root, 50);

insertNode(root, 30);

insertNode(root, 70);

insertNode(root, 20);

insertNode(root, 40);

insertNode(root, 60);

insertNode(root, 80);

// Display Level Order Traversal of the binary search tree

printf("Level Order Traversal: ");

levelOrderTraversal(root);

printf("\n");

// Delete a node with a given key (e.g., delete node with key 30)

int keyToDelete = 30;

root = deleteNode(root, keyToDelete);

// Display Level Order Traversal after deletion

printf("Level Order Traversal after deleting %d: ", keyToDelete);

levelOrderTraversal(root);

printf("\n");

// Free memory allocated for the binary search tree

freeBST(root);

return 0;

}

4. Create an adjacency list for graph and implement BFS.

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Structure to represent a node in the adjacency list

struct Node {

int vertex;

struct Node\* next;

};

// Structure to represent the adjacency list

struct AdjList {

struct Node\* head;

};

// Structure to represent a graph

struct Graph {

int numVertices;

struct AdjList\* array;

};

// Function to create a new node with the given vertex

struct Node\* createNode(int vertex) {

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

newNode->vertex = vertex;

newNode->next = NULL;

return newNode;

}

// Function to create a graph with a given number of vertices

struct Graph\* createGraph(int numVertices) {

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

graph->numVertices = numVertices;

graph->array = (struct AdjList\*)malloc(numVertices \* sizeof(struct AdjList));

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

graph->array[i].head = NULL;

}

return graph;

}

// Function to add an edge to an undirected graph

void addEdge(struct Graph\* graph, int src, int dest) {

// Add an edge from src to dest

struct Node\* newNode = createNode(dest);

newNode->next = graph->array[src].head;

graph->array[src].head = newNode;

// Add an edge from dest to src

newNode = createNode(src);

newNode->next = graph->array[dest].head;

graph->array[dest].head = newNode;

}

// Function to perform Breadth-First Search (BFS) traversal

void BFS(struct Graph\* graph, int startVertex) {

// Create a boolean array to keep track of visited vertices

bool\* visited = (bool\*)malloc(graph->numVertices \* sizeof(bool));

for (int i = 0; i < graph->numVertices; ++i) {

visited[i] = false;

}

// Create a queue for BFS

int\* queue = (int\*)malloc(graph->numVertices \* sizeof(int));

int front = -1, rear = -1;

// Enqueue the start vertex and mark it as visited

queue[++rear] = startVertex;

visited[startVertex] = true;

// Perform BFS

while (front != rear) {

int currentVertex = queue[++front];

printf("%d ", currentVertex);

// Traverse the adjacent vertices of the current vertex

struct Node\* temp = graph->array[currentVertex].head;

while (temp != NULL) {

int adjacentVertex = temp->vertex;

if (!visited[adjacentVertex]) {

// Enqueue the adjacent vertex and mark it as visited

queue[++rear] = adjacentVertex;

visited[adjacentVertex] = true;

}

temp = temp->next;

}

}

// Free the memory allocated for the boolean array and queue

free(visited);

free(queue);

}

// Function to free memory allocated for the graph

void freeGraph(struct Graph\* graph) {

for (int i = 0; i < graph->numVertices; ++i) {

struct Node\* current = graph->array[i].head;

while (current != NULL) {

struct Node\* next = current->next;

free(current);

current = next;

}

}

free(graph->array);

free(graph);

}

int main() {

// Create a graph with 5 vertices

struct Graph\* graph = createGraph(5);

// Add edges to the graph

addEdge(graph, 0, 1);

addEdge(graph, 0, 4);

addEdge(graph, 1, 2);

addEdge(graph, 1, 3);

addEdge(graph, 1, 4);

addEdge(graph, 2, 3);

addEdge(graph, 3, 4);

// Perform BFS traversal starting from vertex 0

printf("BFS Traversal starting from vertex 0: ");

BFS(graph, 0);

printf("\n");

// Free memory allocated for the graph

freeGraph(graph);

return 0;

}

5. Create an adjacency matrix for graph and implement DFS.

#include <stdio.h>

#include <stdlib.h>

// Function to perform Depth-First Search (DFS) traversal

void DFS(int\*\* adjMatrix, int\* visited, int numVertices, int currentVertex) {

printf("%d ", currentVertex);

visited[currentVertex] = 1; // Mark the current vertex as visited

// Traverse the adjacent vertices of the current vertex

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

if (adjMatrix[currentVertex][i] == 1 && !visited[i]) {

DFS(adjMatrix, visited, numVertices, i);

}

}

}

int main() {

int numVertices;

// Input the number of vertices in the graph

printf("Enter the number of vertices in the graph: ");

scanf("%d", &numVertices);

// Create an adjacency matrix

int\*\* adjMatrix = (int\*\*)malloc(numVertices \* sizeof(int\*));

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

adjMatrix[i] = (int\*)malloc(numVertices \* sizeof(int));

}

// Input the adjacency matrix

printf("Enter the adjacency matrix:\n");

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

for (int j = 0; j < numVertices; ++j) {

scanf("%d", &adjMatrix[i][j]);

}

}

// Create an array to keep track of visited vertices

int\* visited = (int\*)malloc(numVertices \* sizeof(int));

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

visited[i] = 0; // Initialize all vertices as not visited

}

// Perform DFS traversal starting from vertex 0

printf("DFS Traversal starting from vertex 0: ");

DFS(adjMatrix, visited, numVertices, 0);

printf("\n");

// Free memory allocated for the adjacency matrix and visited array

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

free(adjMatrix[i]);

}

free(adjMatrix);

free(visited);

return 0;

}

6. Create an adjacency list for graph and find its connected components using DFS.

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in the adjacency list

struct Node {

int vertex;

struct Node\* next;

};

// Structure to represent the adjacency list

struct AdjList {

struct Node\* head;

};

// Structure to represent a graph

struct Graph {

int numVertices;

struct AdjList\* array;

};

// Function to create a new node with the given vertex

struct Node\* createNode(int vertex) {

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

newNode->vertex = vertex;

newNode->next = NULL;

return newNode;

}

// Function to create a graph with a given number of vertices

struct Graph\* createGraph(int numVertices) {

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

graph->numVertices = numVertices;

graph->array = (struct AdjList\*)malloc(numVertices \* sizeof(struct AdjList));

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

graph->array[i].head = NULL;

}

return graph;

}

// Function to add an edge to an undirected graph

void addEdge(struct Graph\* graph, int src, int dest) {

// Add an edge from src to dest

struct Node\* newNode = createNode(dest);

newNode->next = graph->array[src].head;

graph->array[src].head = newNode;

// Add an edge from dest to src

newNode = createNode(src);

newNode->next = graph->array[dest].head;

graph->array[dest].head = newNode;

}

// Function to perform Depth-First Search (DFS) traversal

void DFS(struct Graph\* graph, int\* visited, int currentVertex) {

visited[currentVertex] = 1; // Mark the current vertex as visited

printf("%d ", currentVertex);

// Traverse the adjacent vertices of the current vertex

struct Node\* temp = graph->array[currentVertex].head;

while (temp != NULL) {

if (!visited[temp->vertex]) {

DFS(graph, visited, temp->vertex);

}

temp = temp->next;

}

}

// Function to find connected components in a graph

void findConnectedComponents(struct Graph\* graph) {

int\* visited = (int\*)malloc(graph->numVertices \* sizeof(int));

for (int i = 0; i < graph->numVertices; ++i) {

visited[i] = 0; // Initialize all vertices as not visited

}

printf("Connected Components:\n");

for (int i = 0; i < graph->numVertices; ++i) {

if (!visited[i]) {

DFS(graph, visited, i);

printf("\n");

}

}

free(visited);

}

// Function to free memory allocated for the graph

void freeGraph(struct Graph\* graph) {

for (int i = 0; i < graph->numVertices; ++i) {

struct Node\* current = graph->array[i].head;

while (current != NULL) {

struct Node\* next = current->next;

free(current);

current = next;

}

}

free(graph->array);

free(graph);

}

int main() {

// Create a graph with 7 vertices

struct Graph\* graph = createGraph(7);

// Add edges to the graph

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 3);

addEdge(graph, 2, 4);

addEdge(graph, 5, 6);

// Find and print connected components in the graph

findConnectedComponents(graph);

// Free memory allocated for the graph

freeGraph(graph);

return 0;

}

8. Let the parent fork and let the child execute ls command with exec. Observe the result with and without having wait() system call in the parent.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

int main() {

pid\_t pid = fork(); // Create a child process

if (pid < 0) {

// Fork failed

perror("Fork failed");

exit(EXIT\_FAILURE);

} else if (pid == 0) {

// Child process

printf("Child process executing ls command:\n");

execl("/bin/ls", "ls", (char \*)NULL); // Execute ls command

perror("execl failed"); // This line will be reached only if execl fails

exit(EXIT\_FAILURE);

} else {

// Parent process

printf("Parent process waiting for the child to finish (without wait system call)\n");

// Note: Uncommenting the line below will make the parent wait for the child

// wait(NULL);

printf("Parent process continuing without waiting\n");

}

return 0;

}

9. Create two threads in a main program, let the first thread execute a function to display a message namely ”this is thread one” , similarly let the second thread displays ”this is thread two”.

#include <stdio.h>

#include <pthread.h>

// Function to be executed by the first thread

void \*threadOneFunction(void \*arg) {

// Display a message from thread one

printf("This is thread one\n");

// Exit the thread

pthread\_exit(NULL);

}

// Function to be executed by the second thread

void \*threadTwoFunction(void \*arg) {

// Display a message from thread two

printf("This is thread two\n");

// Exit the thread

pthread\_exit(NULL);

}

int main() {

pthread\_t threadOne, threadTwo; // Thread identifiers

// Create the first thread

if (pthread\_create(&threadOne, NULL, threadOneFunction, NULL) != 0) {

fprintf(stderr, "Error creating thread one\n");

return 1;

}

// Create the second thread

if (pthread\_create(&threadTwo, NULL, threadTwoFunction, NULL) != 0) {

fprintf(stderr, "Error creating thread two\n");

return 1;

}

// Wait for both threads to finish

pthread\_join(threadOne, NULL);

pthread\_join(threadTwo, NULL);

return 0;

}