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>

// Employee structure

struct Employee {

char name[50];

char designation[20];

float salary;

struct Employee\* next;

};

// Function to insert a new employee into the linked list

struct Employee\* 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 = head;

return newEmployee;

}

// Function to search and display details of an employee by name

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

struct Employee\* current = head;

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);

return;

}

current = current->next;

}

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

}

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

void averageSalaryOfManagers(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) {

float averageSalary = totalSalary / count;

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

} else {

printf("No Managers found!\n");

}

}

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

void freeLinkedList(struct Employee\* head) {

struct Employee\* current = head;

while (current != NULL) {

struct Employee\* temp = current;

current = current->next;

free(temp);

}

}

// Driver program

int main() {

struct Employee\* head = NULL;

int N;

// Input: Number of employees

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

scanf("%d", &N);

// Input: Employee details

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

char name[50];

char designation[20];

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);

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

}

// Search for an employee

char searchName[50];

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

scanf("%s", searchName);

searchAndDisplayEmployee(head, searchName);

// Display average salary of Managers

averageSalaryOfManagers(head);

// Free memory

freeLinkedList(head);

return 0;

}

1. 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 node at the end of a linked list

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

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

newNode->data = data;

newNode->next = NULL;

if (head == NULL) {

return newNode;

}

struct Node\* current = head;

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

return head;

}

// Function to merge two sorted linked lists into a new sorted linked list

struct Node\* mergeSortedLists(struct Node\* listA, struct Node\* listB) {

struct Node\* mergedList = NULL;

while (listA != NULL && listB != NULL) {

if (listA->data <= listB->data) {

mergedList = insertNode(mergedList, listA->data);

listA = listA->next;

} else {

mergedList = insertNode(mergedList, listB->data);

listB = listB->next;

}

}

// If there are remaining elements in listA or listB

while (listA != NULL) {

mergedList = insertNode(mergedList, listA->data);

listA = listA->next;

}

while (listB != NULL) {

mergedList = insertNode(mergedList, listB->data);

listB = listB->next;

}

return mergedList;

}

// Function to display a linked list

void displayList(struct Node\* head) {

struct Node\* current = head;

while (current != NULL) {

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

current = current->next;

}

printf("\n");

}

// Driver program

int main() {

struct Node\* listA = NULL;

struct Node\* listB = NULL;

// Insert elements into linked list A (sorted)

listA = insertNode(listA, 1);

listA = insertNode(listA, 3);

listA = insertNode(listA, 5);

// Insert elements into linked list B (sorted)

listB = insertNode(listB, 2);

listB = insertNode(listB, 4);

listB = insertNode(listB, 6);

// Merge the two sorted linked lists into a new sorted linked list

struct Node\* mergedList = mergeSortedLists(listA, listB);

// Display the merged sorted linked list

printf("Merged Sorted List: ");

1. 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 node at the end of a linked list

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

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

newNode->data = data;

newNode->next = NULL;

if (head == NULL) {

return newNode;

}

struct Node\* current = head;

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

return head;

}

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

void splitLinkedList(struct Node\* source, struct Node\*\* frontRef, struct Node\*\* backRef) {

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

\*frontRef = source;

\*backRef = NULL;

return;

}

struct Node\* slow = source;

struct Node\* fast = source->next;

// Advance 'fast' by two nodes and 'slow' by one node

while (fast != NULL) {

fast = fast->next;

if (fast != NULL) {

slow = slow->next;

fast = fast->next;

}

}

// 'slow' is before the midpoint in the list

\*frontRef = source;

\*backRef = slow->next;

slow->next = NULL;

}

// Function to display a linked list

void displayList(struct Node\* head) {

struct Node\* current = head;

while (current != NULL) {

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

current = current->next;

}

printf("\n");

}

// Driver program

int main() {

struct Node\* listA = NULL;

struct Node\* listB = NULL;

struct Node\* listC = NULL;

// Insert elements into linked list A

listA = insertNode(listA, 1);

listA = insertNode(listA, 2);

listA = insertNode(listA, 3);

listA = insertNode(listA, 4);

listA = insertNode(listA, 5);

// Split linked list A into two parts (B and C)

splitLinkedList(listA, &listB, &listC);

// Display the original linked list A

printf("Original List A: ");

displayList(listA);

// Display the two split linked lists B and C

printf("List B (first half): ");

displayList(listB);

printf("List C (second half): ");

displayList(listC);

// Free memory

free(listA);

free(listB);

free(listC);

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 pointer to the head node and the count of elements

struct Head {

struct Node\* head\_ptr;

int num;

};

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

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

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

newNode->data = data;

newNode->next = NULL;

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

// If the list is empty, set the new node as the head

listHead->head\_ptr = newNode;

} else {

// Traverse to the end and insert the new node

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

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

}

// Increment the count of elements

listHead->num++;

}

// Function to delete a node from the linked list

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

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

printf("List is empty. Cannot delete.\n");

return;

}

struct Node\* current = listHead->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) {

printf("Element %d not found in the list. Cannot delete.\n", data);

return;

}

// Adjust the pointers to remove the node

if (prev == NULL) {

// If the node to be deleted is the head

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

} else {

prev->next = current->next;

}

// Free the memory of the deleted node

free(current);

// Decrement the count of elements

listHead->num--;

}

// Function to display the linked list

void displayList(struct Head\* listHead) {

printf("Linked List: ");

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

while (current != NULL) {

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

current = current->next;

}

printf("\n");

}

// Driver program

int main() {

struct Head listHead = {NULL, 0}; // Initialize the head structure

// Insert elements into the linked list

insertNode(&listHead, 1);

insertNode(&listHead, 2);

insertNode(&listHead, 3);

// Display the linked list

displayList(&listHead);

// Delete a node from the linked list

deleteNode(&listHead, 2);

// Display the updated linked list

displayList(&listHead);

// Free memory (optional)

struct Node\* current = listHead.head\_ptr;

while (current != NULL) {

struct Node\* next = current->next;

free(current);

current = next;

}

return 0;

}

1. 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>

#define N 10 // Size of the array

// Structure to represent two stacks

struct TwoStacks {

int\* array;

int top1;

int top2;

};

// Function to initialize two stacks in one array

struct TwoStacks\* initializeTwoStacks() {

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

if (ts == NULL) {

printf("Memory allocation failed.\n");

exit(EXIT\_FAILURE);

}

ts->array = (int\*)malloc(sizeof(int) \* N);

if (ts->array == NULL) {

printf("Memory allocation failed.\n");

exit(EXIT\_FAILURE);

}

ts->top1 = -1; // Initialize top of the first stack

ts->top2 = N; // Initialize top of the second stack

return ts;

}

// Function to push an element onto the first stack

void push1(struct TwoStacks\* ts, int data) {

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

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

} else {

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

}

}

// Function to push an element onto the second stack

void push2(struct TwoStacks\* ts, int data) {

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

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

} else {

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

}

}

// Function to pop an element from the first stack

int pop1(struct TwoStacks\* ts) {

if (ts->top1 >= 0) {

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

} else {

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

exit(EXIT\_FAILURE);

}

}

// Function to pop an element from the second stack

int pop2(struct TwoStacks\* ts) {

if (ts->top2 < N) {

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

} else {

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

exit(EXIT\_FAILURE);

}

}

int main() {

struct TwoStacks\* ts = initializeTwoStacks();

push1(ts, 1);

push1(ts, 2);

push1(ts, 3);

push2(ts, 4);

push2(ts, 5);

push2(ts, 6);

printf("Popped element from stack 1: %d\n", pop1(ts));

printf("Popped element from stack 2: %d\n", pop2(ts));

free(ts->array);

free(ts);

return 0;

}

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

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a queue node

struct QueueNode {

int data;

struct QueueNode\* next;

};

// Structure to represent a queue

struct Queue {

struct QueueNode\* front, \* rear;

};

// Function to create a new queue node

struct QueueNode\* createQueueNode(int data) {

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

if (newNode == NULL) {

printf("Memory allocation failed.\n");

exit(EXIT\_FAILURE);

}

newNode->data = data;

newNode->next = NULL;

return newNode;

}

// Function to initialize a queue

struct Queue\* createQueue() {

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

if (queue == NULL) {

printf("Memory allocation failed.\n");

exit(EXIT\_FAILURE);

}

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

return queue;

}

// Function to check if a queue is empty

int isQueueEmpty(struct Queue\* queue) {

return (queue->front == NULL);

}

// Function to enqueue a data into a queue

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

struct QueueNode\* newNode = createQueueNode(data);

if (isQueueEmpty(queue)) {

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

} else {

queue->rear->next = newNode;

queue->rear = newNode;

}

}

// Function to dequeue a data from a queue

int dequeue(struct Queue\* queue) {

if (isQueueEmpty(queue)) {

printf("Queue is empty.\n");

exit(EXIT\_FAILURE);

}

struct QueueNode\* temp = queue->front;

int data = temp->data;

queue->front = temp->next;

// If front becomes NULL, update rear to NULL as well

if (queue->front == NULL) {

queue->rear = NULL;

}

free(temp);

return data;

}

// Structure to represent a stack using two queues

struct StackUsingQueues {

struct Queue\* q1;

struct Queue\* q2;

};

// Function to initialize a stack using two queues

struct StackUsingQueues\* createStack() {

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

if (stack == NULL) {

printf("Memory allocation failed.\n");

exit(EXIT\_FAILURE);

}

stack->q1 = createQueue();

stack->q2 = createQueue();

return stack;

}

// Function to push an element onto the stack

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

// Enqueue the element to q2

enqueue(stack->q2, data);

// Transfer all elements from q1 to q2

while (!isQueueEmpty(stack->q1)) {

enqueue(stack->q2, dequeue(stack->q1));

}

// Swap the names of q1 and q2

struct Queue\* temp = stack->q1;

stack->q1 = stack->q2;

stack->q2 = temp;

}

// Function to pop an element from the stack

int pop(struct StackUsingQueues\* stack) {

if (isQueueEmpty(stack->q1)) {

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

exit(EXIT\_FAILURE);

}

// Dequeue the front element from q1

return dequeue(stack->q1);

}

int main() {

struct StackUsingQueues\* stack = createStack();

push(stack, 1);

push(stack, 2);

push(stack, 3);

printf("Popped element: %d\n", pop(stack));

push(stack, 4);

push(stack, 5);

printf("Popped element: %d\n", pop(stack));

printf("Popped element: %d\n", pop(stack));

free(stack->q1);

free(stack->q2);

free(stack);

return 0;

}

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

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a stack node

struct StackNode {

int data;

struct StackNode\* next;

};

// Structure to represent a stack

struct Stack {

struct StackNode\* top;

};

// Function to create a new stack node

struct StackNode\* createStackNode(int data) {

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

if (newNode == NULL) {

printf("Memory allocation failed.\n");

exit(EXIT\_FAILURE);

}

newNode->data = data;

newNode->next = NULL;

return newNode;

}

// Function to initialize a stack

struct Stack\* createStack() {

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

if (stack == NULL) {

printf("Memory allocation failed.\n");

exit(EXIT\_FAILURE);

}

stack->top = NULL;

return stack;

}

// Function to check if a stack is empty

int isStackEmpty(struct Stack\* stack) {

return (stack->top == NULL);

}

// Function to push an element onto the stack

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

struct StackNode\* newNode = createStackNode(data);

newNode->next = stack->top;

stack->top = newNode;

}

// Function to pop an element from the stack

int pop(struct Stack\* stack) {

if (isStackEmpty(stack)) {

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

exit(EXIT\_FAILURE);

}

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 QueueUsingStacks {

struct Stack\* stack1;

struct Stack\* stack2;

};

// Function to initialize a queue using two stacks

struct QueueUsingStacks\* createQueue() {

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

if (queue == NULL) {

printf("Memory allocation failed.\n");

exit(EXIT\_FAILURE);

}

queue->stack1 = createStack();

queue->stack2 = createStack();

return queue;

}

// Function to enqueue an element into the queue

void insert(struct QueueUsingStacks\* queue, int data) {

// Push all elements from stack1 to stack2

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

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

}

// Push the new element onto stack1

push(queue->stack1, data);

// Push back all elements from stack2 to stack1

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

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

}

}

// Function to dequeue an element from the queue

int delete(struct QueueUsingStacks\* queue) {

if (isStackEmpty(queue->stack1)) {

printf("Queue is empty.\n");

exit(EXIT\_FAILURE);

}

// Pop the front element from stack1

return pop(queue->stack1);

}

int main() {

struct QueueUsingStacks\* queue = createQueue();

insert(queue, 1);

insert(queue, 2);

insert(queue, 3);

printf("Deleted element: %d\n", delete(queue));

insert(queue, 4);

insert(queue, 5);

printf("Deleted element: %d\n", delete(queue));

printf("Deleted element: %d\n", delete(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 {

char name[50];

int rollNumber;

float mathScore;

float scienceScore;

struct Student\* next;

struct Student\* prev;

};

// Function to create a new student node

struct Student\* createStudent(char name[], int rollNumber, float mathScore, float scienceScore) {

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

if (newStudent == NULL) {

printf("Memory allocation failed.\n");

exit(EXIT\_FAILURE);

}

strcpy(newStudent->name, name);

newStudent->rollNumber = rollNumber;

newStudent->mathScore = mathScore;

newStudent->scienceScore = scienceScore;

newStudent->next = NULL;

newStudent->prev = NULL;

return newStudent;

}

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

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

struct Student\* newStudent = createStudent(name, rollNumber, mathScore, scienceScore);

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 and display details

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

struct Student\* temp = head;

while (temp != NULL) {

if (temp->rollNumber == rollNumber) {

printf("Student found!\n");

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

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

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

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

return;

}

temp = temp->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\* temp = head;

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

while (temp != NULL) {

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

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

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

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

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

printf("\n");

}

temp = temp->next;

}

}

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

void displayStudents(struct Student\* head) {

struct Student\* temp = head;

while (temp != NULL) {

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

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

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

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

printf("\n");

temp = temp->next;

}

}

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

void freeList(struct Student\* head) {

struct Student\* temp;

while (head != NULL) {

temp = head;

head = head->next;

free(temp);

}

}

int main() {

struct Student\* studentsList = NULL;

// Insert N students into the doubly linked list

insertStudent(&studentsList, "John", 101, 95.5, 92.0);

insertStudent(&studentsList, "Alice", 102, 88.0, 94.5);

insertStudent(&studentsList, "Bob", 103, 92.5, 89.0);

// Add more students as needed

// Display all students

printf("All Students:\n");

displayStudents(studentsList);

// Search for a student

int searchRollNumber = 102;

searchStudent(studentsList, searchRollNumber);

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

displayHighScorers(studentsList);

// Free the memory allocated for the doubly linked list

freeList(studentsList);

return 0;

}

1. 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 binary tree node

struct TreeNode {

int data;

struct TreeNode\* left;

struct TreeNode\* right;

};

// Function to create a new node in the binary tree

struct TreeNode\* createNode(int data) {

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

if (newNode == NULL) {

printf("Memory allocation failed.\n");

exit(EXIT\_FAILURE);

}

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Function to perform Inorder traversal of the binary tree

void inorderTraversal(struct TreeNode\* root) {

if (root != NULL) {

inorderTraversal(root->left);

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

inorderTraversal(root->right);

}

}

// Function to perform Preorder traversal of the binary tree

void preorderTraversal(struct TreeNode\* root) {

if (root != NULL) {

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

preorderTraversal(root->left);

preorderTraversal(root->right);

}

}

// Function to perform 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 sample binary tree

struct TreeNode\* root = createNode(1);

root->left = createNode(2);

root->right = createNode(3);

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

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

root->right->left = createNode(6);

root->right->right = createNode(7);

// Traverse the binary tree using different traversal 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 nodes

free(root->left->left);

free(root->left->right);

free(root->right->left);

free(root->right->right);

free(root->left);

free(root->right);

free(root);

return 0;

}

1. 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 binary search tree node

struct TreeNode {

int data;

struct TreeNode\* left;

struct TreeNode\* right;

};

// Function to create a new node in the BST

struct TreeNode\* createNode(int data) {

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

if (newNode == NULL) {

printf("Memory allocation failed.\n");

exit(EXIT\_FAILURE);

}

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Function to insert a node into the BST

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 minimum value node in a BST

struct TreeNode\* findMin(struct TreeNode\* root) {

while (root->left != NULL) {

root = root->left;

}

return root;

}

// Function to delete a node from the BST

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

if (root == NULL) {

return root;

}

if (data < root->data) {

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

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

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

} 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 = findMin(root->right);

// Copy the inorder successor's content 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 the BST

void levelOrderTraversal(struct TreeNode\* root) {

if (root == NULL) {

return;

}

struct TreeNode\* queue[100];

int front = 0, rear = 0;

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 the memory allocated for the BST

void freeBST(struct TreeNode\* root) {

if (root != NULL) {

freeBST(root->left);

freeBST(root->right);

free(root);

}

}

int main() {

struct TreeNode\* root = NULL;

// Insert nodes into the BST

root = insertNode(root, 50);

root = insertNode(root, 30);

root = insertNode(root, 20);

root = insertNode(root, 40);

root = insertNode(root, 70);

root = insertNode(root, 60);

root = insertNode(root, 80);

// Display Level Order Traversal of the BST

printf("Level Order Traversal: ");

levelOrderTraversal(root);

printf("\n");

// Delete a node (e.g., node with value 30) from the BST

int nodeToDelete = 30;

root = deleteNode(root, nodeToDelete);

// Display Level Order Traversal after deletion

printf("Level Order Traversal after deletion of %d: ", nodeToDelete);

levelOrderTraversal(root);

printf("\n");

// Free the memory allocated for the BST

freeBST(root);

return 0;

}

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

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Node structure for the adjacency list

struct Node {

int data;

struct Node\* next;

};

// Graph structure

struct Graph {

int V; // Number of vertices

struct Node\*\* adjList; // Array of adjacency lists

};

// Queue structure for BFS

struct Queue {

int front, rear;

int\* array;

};

// Function to create a new node

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

// Function to create a graph

struct Graph\* createGraph(int V) {

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

graph->V = V;

// Create an array of adjacency lists

graph->adjList = (struct Node\*\*)malloc(V \* sizeof(struct Node\*));

// Initialize each adjacency list as empty

for (int i = 0; i < V; ++i)

graph->adjList[i] = NULL;

return graph;

}

// Function to add an edge to the 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->adjList[src];

graph->adjList[src] = newNode;

// Add an edge from dest to src (since the graph is undirected)

newNode = createNode(src);

newNode->next = graph->adjList[dest];

graph->adjList[dest] = newNode;

}

// Function to create a new queue

struct Queue\* createQueue(int capacity) {

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

queue->front = queue->rear = -1;

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

return queue;

}

// Function to check if the queue is empty

bool isEmpty(struct Queue\* queue) {

return queue->front == -1;

}

// Function to enqueue an element to the queue

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

queue->array[++queue->rear] = item;

if (queue->front == -1)

++queue->front;

}

// Function to dequeue an element from the queue

int dequeue(struct Queue\* queue) {

int item = queue->array[queue->front];

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

queue->front = queue->rear = -1;

else

++queue->front;

return item;

}

// Function to perform BFS

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

struct Queue\* queue = createQueue(graph->V);

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

// Mark all vertices as not visited

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

visited[i] = false;

visited[startVertex] = true;

enqueue(queue, startVertex);

while (!isEmpty(queue)) {

int currentVertex = dequeue(queue);

printf("%d ", currentVertex);

struct Node\* temp = graph->adjList[currentVertex];

while (temp) {

int adjVertex = temp->data;

if (!visited[adjVertex]) {

visited[adjVertex] = true;

enqueue(queue, adjVertex);

}

temp = temp->next;

}

}

free(queue);

free(visited);

}

// Driver program

int main() {

int V = 5; // Number of vertices

struct Graph\* graph = createGraph(V);

// Adding edges

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 1, 3);

addEdge(graph, 2, 4);

int startVertex = 0; // Starting vertex for BFS

printf("BFS starting from vertex %d: ", startVertex);

BFS(graph, startVertex);

return 0;

}

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

#include <stdio.h>

#include <stdlib.h>

// Graph structure

struct Graph {

int V; // Number of vertices

int\*\* adjMatrix; // Adjacency matrix

int\* visited; // Array to track visited vertices

};

// Function to create a graph

struct Graph\* createGraph(int V) {

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

graph->V = V;

// Create an adjacency matrix

graph->adjMatrix = (int\*\*)malloc(V \* sizeof(int\*));

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

graph->adjMatrix[i] = (int\*)malloc(V \* sizeof(int));

for (int j = 0; j < V; ++j)

graph->adjMatrix[i][j] = 0;

}

// Initialize visited array

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

for (int i = 0; i < V; ++i)

graph->visited[i] = 0;

return graph;

}

// Function to add an edge to the graph

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

// Add an edge from src to dest

graph->adjMatrix[src][dest] = 1;

// Add an edge from dest to src (since the graph is undirected)

graph->adjMatrix[dest][src] = 1;

}

// Function to perform DFS

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

printf("%d ", startVertex);

graph->visited[startVertex] = 1;

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

if (graph->adjMatrix[startVertex][i] == 1 && !graph->visited[i]) {

DFS(graph, i);

}

}

}

// Driver program

int main() {

int V = 5; // Number of vertices

struct Graph\* graph = createGraph(V);

// Adding edges

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 1, 3);

addEdge(graph, 2, 4);

int startVertex = 0; // Starting vertex for DFS

printf("DFS starting from vertex %d: ", startVertex);

DFS(graph, startVertex);

// Free allocated memory

for (int i = 0; i < V; ++i)

free(graph->adjMatrix[i]);

free(graph->adjMatrix);

free(graph->visited);

free(graph);

return 0;

}

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

#include <stdio.h>

#include <stdlib.h>

// Node structure for the adjacency list

struct Node {

int data;

struct Node\* next;

};

// Graph structure

struct Graph {

int V; // Number of vertices

struct Node\*\* adjList; // Array of adjacency lists

int\* visited; // Array to track visited vertices

};

// Function to create a new node

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

// Function to create a graph

struct Graph\* createGraph(int V) {

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

graph->V = V;

// Create an array of adjacency lists

graph->adjList = (struct Node\*\*)malloc(V \* sizeof(struct Node\*));

// Initialize each adjacency list as empty

for (int i = 0; i < V; ++i)

graph->adjList[i] = NULL;

// Initialize visited array

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

for (int i = 0; i < V; ++i)

graph->visited[i] = 0;

return graph;

}

// Function to add an edge to the 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->adjList[src];

graph->adjList[src] = newNode;

// Add an edge from dest to src (since the graph is undirected)

newNode = createNode(src);

newNode->next = graph->adjList[dest];

graph->adjList[dest] = newNode;

}

// Function to perform DFS for finding connected components

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

printf("%d ", vertex);

graph->visited[vertex] = 1;

struct Node\* temp = graph->adjList[vertex];

while (temp) {

int adjVertex = temp->data;

if (!graph->visited[adjVertex]) {

DFS(graph, adjVertex);

}

temp = temp->next;

}

}

// Function to find connected components using DFS

void findConnectedComponents(struct Graph\* graph) {

printf("Connected Components:\n");

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

if (!graph->visited[i]) {

printf("Component: ");

DFS(graph, i);

printf("\n");

}

}

}

// Driver program

int main() {

int V = 7; // Number of vertices

struct Graph\* graph = createGraph(V);

// Adding edges

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 3);

addEdge(graph, 2, 4);

addEdge(graph, 5, 6);

printf("Graph:\n");

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

struct Node\* temp = graph->adjList[i];

printf("Vertex %d: ", i);

while (temp) {

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

temp = temp->next;

}

printf("\n");

}

findConnectedComponents(graph);

// Free allocated memory

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

struct Node\* temp = graph->adjList[i];

while (temp) {

struct Node\* next = temp->next;

free(temp);

temp = next;

}

}

free(graph->adjList);

free(graph->visited);

free(graph);

return 0;

}

14. 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 <unistd.h>

#include <sys/wait.h>

int main() {

pid\_t pid = fork();

if (pid == -1) {

perror("Fork failed");

return 1;

}

if (pid == 0) {

// This code is executed by the child process

char\* args[] = { "ls", "-l", NULL };

execvp("ls", args);

perror("Exec failed");

return 1;

} else {

// This code is executed by the parent process

// With wait()

int status;

wait(&status);

printf("Child process has finished.\n");

// Without wait()

// sleep(2); // Uncomment this line if you want to observe without wait()

printf("Parent process continues without waiting.\n");

}

return 0;

}

15. 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 for the first thread

void \*threadFunction1(void \*arg) {

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

return NULL;

}

// Function for the second thread

void \*threadFunction2(void \*arg) {

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

return NULL;

}

int main() {

pthread\_t t1, t2;

// Create two threads and start them

pthread\_create(&t1, NULL, threadFunction1, NULL);

pthread\_create(&t2, NULL, threadFunction2, NULL);

// Wait for both threads to finish

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

return 0;

}