Exp no 1

Write a C program to perform the following operations on a single-dimensional array: 1. Insert an element at a specific position. 2. Delete an element from a specific index. 3. Display all elements of the array

Program

#include <stdio.h>

#define MAX\_SIZE 100

void insertElement(int arr[], int \*size, int element, int position) {

if (\*size >= MAX\_SIZE) {

printf("Array is full. Cannot insert new element.\n");

return;

}

if (position < 0 || position > \*size) {

printf("Invalid position. Please enter a position between 0 and %d.\n", \*size);

return;

}

for (int i = \*size; i > position; i--) {

arr[i] = arr[i - 1];

}

arr[position] = element;

(\*size)++;

printf("Element inserted successfully.\n");

}

void deleteElement(int arr[], int \*size, int position) {

if (\*size <= 0) {

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

return;

}

if (position < 0 || position >= \*size) {

printf("Invalid position. Please enter a position between 0 and %d.\n", \*size - 1);

return;

}

for (int i = position; i < \*size - 1; i++) {

arr[i] = arr[i + 1];

}

(\*size)--;

printf("Element deleted successfully.\n");

}

void displayArray(int arr[], int size) {

if (size == 0) {

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

return;

}

printf("Array elements: ");

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

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

int arr[MAX\_SIZE];

int size = 0;

int choice, element, position;

while (1) {

printf("\nArray Operations Menu:\n");

printf("1. Insert Element\n");

printf("2. Delete Element\n");

printf("3. Display Array\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the element to insert: ");

scanf("%d", &element);

printf("Enter the position to insert (0 to %d): ", size);

scanf("%d", &position);

insertElement(arr, &size, element, position);

break;

case 2:

printf("Enter the position to delete (0 to %d): ", size - 1);

scanf("%d", &position);

deleteElement(arr, &size, position);

break;

case 3:

displayArray(arr, size);

break;

case 4:

printf("Exiting the program.\n");

return 0;

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

Que 2

Develop a C program to create and perform the following operations on a two-dimensional array (matrix): 1. Insert an element at a specific position. 2. Delete an element from a specific index. 3. Display all elements of the array.

Program

#include <stdio.h>

#define MAX\_SIZE 100

void insertElement(int arr[], int \*size, int element, int position) {

if (\*size >= MAX\_SIZE) {

printf("Array is full. Cannot insert new element.\n");

return;

}

if (position < 0 || position > \*size) {

printf("Invalid position. Please enter a position between 0 and %d.\n", \*size);

return;

}

for (int i = \*size; i > position; i--) {

arr[i] = arr[i - 1];

}

arr[position] = element;

(\*size)++;

printf("Element inserted successfully.\n");

}

void deleteElement(int arr[], int \*size, int position) {

if (\*size <= 0) {

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

return;

}

if (position < 0 || position >= \*size) {

printf("Invalid position. Please enter a position between 0 and %d.\n", \*size - 1);

return;

}

for (int i = position; i < \*size - 1; i++) {

arr[i] = arr[i + 1];

}

(\*size)--;

printf("Element deleted successfully.\n");

}

void displayArray(int arr[], int size) {

if (size == 0) {

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

return;

}

printf("Array elements: ");

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

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

int arr[MAX\_SIZE];

int size = 0;

int choice, element, position;

while (1) {

printf("\nArray Operations Menu:\n");

printf("1. Insert Element\n");

printf("2. Delete Element\n");

printf("3. Display Array\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the element to insert: ");

scanf("%d", &element);

printf("Enter the position to insert (0 to %d): ", size);

scanf("%d", &position);

insertElement(arr, &size, element, position);

break;

case 2:

printf("Enter the position to delete (0 to %d): ", size - 1);

scanf("%d", &position);

deleteElement(arr, &size, position);

break;

case 3:

displayArray(arr, size);

break;

case 4:

printf("Exiting the program.\n");

return 0;

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

Exp no 2

Basic Function Demonstration: Write a C program to create and use a function that: 1. Accepts two integers as parameters. 2. Returns their sum to the main function. 3. Displays the result in the main function.

Pro

#include <stdio.h>

int add(int a, int b) {

return a + b;

}

int main() {

int num1, num2, result;

printf("Enter the first number: ");

scanf("%d", &num1);

printf("Enter the second number: ");

scanf("%d", &num2);

result = add(num1, num2);

printf("The sum of %d and %d is: %d\n", num1, num2, result);

return 0;

}

Q2 Swapping Numbers Using Functions: Write a C program that swaps the values of two variables using a function. The program should: 1. Accept two numbers from the user. 2. Use a function to swap their values by passing them as pointers. 3. Display the values before and after the swap in the main function

#include <stdio.h>

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int main() {

int num1, num2;

printf("Enter the first number: ");

scanf("%d", &num1);

printf("Enter the second number: ");

scanf("%d", &num2);

printf("\nBefore swapping: num1 = %d, num2 = %d\n", num1, num2);

swap(&num1, &num2);

printf("After swapping: num1 = %d, num2 = %d\n", num1, num2);

return 0;

}

Q3 Recursive Function for Factorial: Develop a C program that calculates the factorial of a number using a recursive function. The program should: 1. Accept a number from the user. 2. Use recursion to compute the factorial. 3. Display the result in the main function

#include <stdio.h>

int factorial(int n) {

if (n == 0 || n == 1) {

return 1;

}

return n \* factorial(n - 1);

}

int main() {

int num;

printf("Enter a number to calculate its factorial: ");

scanf("%d", &num);

if (num < 0) {

printf("Factorial of a negative number is undefined.\n");

} else {

printf("The factorial of %d is: %d\n", num, factorial(num));

}

return 0;

}

Q4 Function with Array as Parameter: Write a C program that uses a function to find the largest element in an array. The program should: 1. Pass the array and its size as parameters to the function. 2. Find and return the largest element to the main function. 3. Display the result in the main function.

#include <stdio.h>

int findLargest(int arr[], int size) {

int largest = arr[0];

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

if (arr[i] > largest) {

largest = arr[i];

}

}

return largest;

}

int main() {

int size;

printf("Enter the size of the array: ");

scanf("%d", &size);

if (size <= 0) {

printf("Invalid size. The array must have at least one element.\n");

return 1;

}

int arr[size];

printf("Enter %d elements of the array:\n", size);

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

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

}

int largest = findLargest(arr, size);

printf("The largest element in the array is: %d\n", largest);

return 0;

}

Exp no 3

Basic Structure Usage: Write a C program to create a structure Student with the following fields: roll\_no, name, and marks. The program should: 1. Accept details of 5 students from the user. 2. Display the details of all students

#include <stdio.h>

#include <string.h>

struct Student {

int roll\_no;

char name[50];

float marks;

};

int main() {

struct Student students[5];

// Accept details of 5 students

for (int i = 0; i < 5; i++) {

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

printf("Roll Number: ");

scanf("%d", &students[i].roll\_no);

printf("Name: ");

getchar(); // Clear the input buffer

fgets(students[i].name, sizeof(students[i].name), stdin);

students[i].name[strcspn(students[i].name, "\n")] = 0; // Remove newline

printf("Marks: ");

scanf("%f", &students[i].marks);

}

// Display details of all students

printf("\nDetails of Students:\n");

for (int i = 0; i < 5; i++) {

printf("Student %d:\n", i + 1);

printf("Roll Number: %d\n", students[i].roll\_no);

printf("Name: %s\n", students[i].name);

printf("Marks: %.2f\n", students[i].marks);

}

return 0;

}

Q2 Structure and Function Integration: Write a C program that uses a structure Employee with fields id, name, and salary. The program should: 1. Accept details of employees from the user. 2. Pass the structure to a function to calculate and display the total salary of all employees

#include <stdio.h>

#include <string.h>

struct Employee {

int id;

char name[50];

float salary;

};

float calculateTotalSalary(struct Employee employees[], int size) {

float totalSalary = 0;

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

totalSalary += employees[i].salary;

}

return totalSalary;

}

int main() {

int numEmployees;

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

scanf("%d", &numEmployees);

if (numEmployees <= 0) {

printf("Invalid number of employees.\n");

return 1;

}

struct Employee employees[numEmployees];

// Accept details of employees

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

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

printf("ID: ");

scanf("%d", &employees[i].id);

printf("Name: ");

getchar(); // Clear the input buffer

fgets(employees[i].name, sizeof(employees[i].name), stdin);

employees[i].name[strcspn(employees[i].name, "\n")] = 0; // Remove newline

printf("Salary: ");

scanf("%f", &employees[i].salary);

}

// Calculate and display total salary

float totalSalary = calculateTotalSalary(employees, numEmployees);

printf("\nThe total salary of all employees is: %.2f\n", totalSalary);

return 0;

}

Q3 Array of Structures: Develop a C program to create an array of structures for Books, where each structure contains fields title, author, and price. The program should: 1. Accept details of N books from the user. 2. Search for a book by title and display its details if found.

#include <stdio.h>

#include <string.h>

struct Book {

char title[100];

char author[50];

float price;

};

void searchBookByTitle(struct Book books[], int size, char searchTitle[]) {

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

if (strcmp(books[i].title, searchTitle) == 0) {

printf("\nBook Found:\n");

printf("Title: %s\n", books[i].title);

printf("Author: %s\n", books[i].author);

printf("Price: %.2f\n", books[i].price);

return;

}

}

printf("\nBook with title '%s' not found.\n", searchTitle);

}

int main() {

int numBooks;

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

scanf("%d", &numBooks);

if (numBooks <= 0) {

printf("Invalid number of books.\n");

return 1;

}

struct Book books[numBooks];

// Accept details of books

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

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

printf("Title: ");

getchar(); // Clear the input buffer

fgets(books[i].title, sizeof(books[i].title), stdin);

books[i].title[strcspn(books[i].title, "\n")] = 0; // Remove newline

printf("Author: ");

fgets(books[i].author, sizeof(books[i].author), stdin);

books[i].author[strcspn(books[i].author, "\n")] = 0; // Remove newline

printf("Price: ");

scanf("%f", &books[i].price);

}

char searchTitle[100];

printf("\nEnter the title of the book to search: ");

getchar(); // Clear the input buffer

fgets(searchTitle, sizeof(searchTitle), stdin);

searchTitle[strcspn(searchTitle, "\n")] = 0; // Remove newline

// Search and display the book details

searchBookByTitle(books, numBooks, searchTitle);

return 0;

}

Q4 Nested Structures: Write a C program that defines a nested structure for a Company, where the Employee structure (with id and name) is a part of the Company structure. The program should: 1. Accept and display details of employees in a company. 2. Demonstrate accessing nested structure elements.

#include <stdio.h>

#include <string.h>

// Define the Employee structure

struct Employee {

int id;

char name[50];

};

// Define the Company structure which contains an array of Employees

struct Company {

char companyName[50];

int numEmployees;

struct Employee employees[10]; // Assuming a maximum of 10 employees

};

int main() {

struct Company company;

// Accepting company name and number of employees

printf("Enter company name: ");

fgets(company.companyName, sizeof(company.companyName), stdin);

company.companyName[strcspn(company.companyName, "\n")] = '\0'; // Remove trailing newline

printf("Enter number of employees: ");

scanf("%d", &company.numEmployees);

// Accepting employee details

for (int i = 0; i < company.numEmployees; i++) {

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

printf("ID: ");

scanf("%d", &company.employees[i].id);

getchar(); // To clear the buffer after reading an integer

printf("Name: ");

fgets(company.employees[i].name, sizeof(company.employees[i].name), stdin);

company.employees[i].name[strcspn(company.employees[i].name, "\n")] = '\0'; // Remove trailing newline

}

// Displaying company and employee details

printf("\nCompany Name: %s\n", company.companyName);

printf("Number of Employees: %d\n", company.numEmployees);

for (int i = 0; i < company.numEmployees; i++) {

printf("\nEmployee %d Details:\n", i + 1);

printf("ID: %d\n", company.employees[i].id);

printf("Name: %s\n", company.employees[i].name);

}

return 0;

}

Exp 4

Pointer Basics: Write a C program to demonstrate pointer operations: 1. Declare a pointer and assign it the address of an integer variable. 2. Perform read and write operations using the pointer. 3. Print the address and value of the variable using the pointer.`

#include <stdio.h>

int main() {

int num = 10; // Declare an integer variable

int \*ptr = &num; // Declare a pointer and assign it the address of 'num'

// Displaying the address of the variable 'num' using the pointer

printf("Address of num using pointer: %p\n", ptr);

// Displaying the value of the variable 'num' using the pointer

printf("Value of num using pointer: %d\n", \*ptr);

// Perform a write operation using the pointer

\*ptr = 20; // Changing the value of 'num' using the pointer

// Displaying the updated value of the variable 'num'

printf("\nAfter modifying the value using the pointer:\n");

printf("Address of num using pointer: %p\n", ptr);

printf("Updated value of num using pointer: %d\n", \*ptr);

printf("Updated value of num directly: %d\n", num);

return 0;

}

Q2 Swapping Numbers Using Pointers: Write a C program that swaps the values of two variables using pointers. The program should: 1. Accept two numbers from the user. 2. Use a function to swap their values by passing pointers as arguments. 3. Display the values before and after the swap.

#include <stdio.h>

// Function to swap two integers using pointers

void swap(int \*a, int \*b) {

int temp = \*a; // Store the value of \*a in temp

\*a = \*b; // Assign the value of \*b to \*a

\*b = temp; // Assign the stored value in temp to \*b

}

int main() {

int num1, num2;

// Accept two numbers from the user

printf("Enter the first number: ");

scanf("%d", &num1);

printf("Enter the second number: ");

scanf("%d", &num2);

// Display values before the swap

printf("\nBefore Swap:\n");

printf("num1 = %d, num2 = %d\n", num1, num2);

// Swap the values using the swap function

swap(&num1, &num2);

// Display values after the swap

printf("\nAfter Swap:\n");

printf("num1 = %d, num2 = %d\n", num1, num2);

return 0;

}

Q3 Pointer Arithmetic: Develop a C program that demonstrates pointer arithmetic. Perform the following: 1. Create an array and initialize it with values. 2. Modify array elements using pointer arithmetic and display the updated array.

#include <stdio.h>

int main() {

// Step 1: Create an array and initialize it with values

int arr[] = {5, 10, 15, 20, 25};

int size = sizeof(arr) / sizeof(arr[0]); // Calculate the number of elements in the array

// Display the original array

printf("Original array:\n");

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

printf("%d ", arr[i]);

}

printf("\n");

// Step 2: Modify array elements using pointer arithmetic

int \*ptr = arr; // Pointer to the first element of the array

// Modify each element by increasing it by 10 using pointer arithmetic

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

\*(ptr + i) += 10; // Increase each element by 10

}

// Display the updated array

printf("\nUpdated array using pointer arithmetic:\n");

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

printf("%d ", \*(ptr + i)); // Access and print the updated array elements using pointer arithmetic

}

printf("\n");

return 0;

}

Exp 5

Q1 Creating a Singly Linked List: Write a C program to create a singly linked list. The program should: 1. Accept values from the user to insert as nodes. 2. Display the linked list.

#include <stdio.h>

#include <stdlib.h>

// Define a node structure for the singly linked list

struct Node {

int data;

struct Node \*next;

};

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

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

struct Node \*newNode = (struct Node \*)malloc(sizeof(struct Node)); // Allocate memory for a new node

struct Node \*last = \*head;

newNode->data = value; // Assign value to the new node

newNode->next = NULL; // Set the next pointer to NULL

// If the list is empty, make the new node the head

if (\*head == NULL) {

\*head = newNode;

return;

}

// Otherwise, traverse to the last node and insert the new node

while (last->next != NULL) {

last = last->next;

}

last->next = newNode; // Make the last node point to the new node

}

// Function to display the linked list

void displayList(struct Node \*head) {

struct Node \*temp = head;

if (temp == NULL) {

printf("The list is empty.\n");

return;

}

printf("Linked List: ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

int main() {

struct Node \*head = NULL; // Initialize the linked list as empty

int n, value;

// Accept the number of nodes to be inserted

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

scanf("%d", &n);

// Accept values and insert them into the linked list

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

printf("Enter value for node %d: ", i + 1);

scanf("%d", &value);

insertNode(&head, value);

}

// Display the linked list

displayList(head);

return 0;

}

Q2 Insertion Operations: Write a C program to perform insertion operations in a singly linked list. The program should: 1. Insert a node at the beginning. 2. Insert a node at the end. 3. Display the updated linked list.

#include <stdio.h>

#include <stdlib.h>

// Define the structure for the node in the singly linked list

struct Node {

int data;

struct Node \*next;

};

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

void insertAtBeginning(struct Node \*\*head, int value) {

struct Node \*newNode = (struct Node \*)malloc(sizeof(struct Node)); // Create a new node

newNode->data = value; // Set the value for the new node

newNode->next = \*head; // Make the new node point to the current head

\*head = newNode; // Update the head to the new node

}

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

void insertAtEnd(struct Node \*\*head, int value) {

struct Node \*newNode = (struct Node \*)malloc(sizeof(struct Node)); // Create a new node

struct Node \*last = \*head;

newNode->data = value; // Set the value for the new node

newNode->next = NULL; // Make the new node point to NULL (end of the list)

// If the list is empty, the new node becomes the head

if (\*head == NULL) {

\*head = newNode;

return;

}

// Otherwise, traverse the list to find the last node

while (last->next != NULL) {

last = last->next;

}

// Make the last node's next point to the new node

last->next = newNode;

}

// Function to display the linked list

void displayList(struct Node \*head) {

struct Node \*temp = head;

if (temp == NULL) {

printf("The list is empty.\n");

return;

}

printf("Linked List: ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

int main() {

struct Node \*head = NULL; // Initialize the linked list as empty

int value, choice;

// Display the menu for insertion operations

do {

printf("\nMenu:\n");

printf("1. Insert a node at the beginning\n");

printf("2. Insert a node at the end\n");

printf("3. Display the linked list\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the value to insert at the beginning: ");

scanf("%d", &value);

insertAtBeginning(&head, value);

break;

case 2:

printf("Enter the value to insert at the end: ");

scanf("%d", &value);

insertAtEnd(&head, value);

break;

case 3:

displayList(head);

break;

case 4:

printf("Exiting program.\n");

break;

default:

printf("Invalid choice, please try again.\n");

}

} while (choice != 4);

return 0;

}

Q3 Deletion Operations: Write a C program to perform deletion operations in a singly linked list. The program should: 1. Delete a node from the beginning. 2. Delete a node from the end. 3. Display the updated linked list.

#include <stdio.h>

#include <stdlib.h>

// Define the structure for the node in the singly linked list

struct Node {

int data;

struct Node \*next;

};

// Function to delete a node from the beginning of the linked list

void deleteFromBeginning(struct Node \*\*head) {

// If the list is empty, nothing to delete

if (\*head == NULL) {

printf("The list is empty, cannot delete.\n");

return;

}

struct Node \*temp = \*head; // Temporary pointer to the current head

\*head = (\*head)->next; // Update the head to the next node

free(temp); // Free the memory of the old head

printf("Node deleted from the beginning.\n");

}

// Function to delete a node from the end of the linked list

void deleteFromEnd(struct Node \*\*head) {

// If the list is empty, nothing to delete

if (\*head == NULL) {

printf("The list is empty, cannot delete.\n");

return;

}

// If there is only one node in the list

if ((\*head)->next == NULL) {

free(\*head); // Free the only node

\*head = NULL; // Set the head to NULL as the list becomes empty

printf("Node deleted from the end.\n");

return;

}

struct Node \*temp = \*head;

// Traverse to the second-to-last node

while (temp->next && temp->next->next != NULL) {

temp = temp->next;

}

// Free the last node and set the second-to-last node's next to NULL

free(temp->next);

temp->next = NULL;

printf("Node deleted from the end.\n");

}

// Function to display the linked list

void displayList(struct Node \*head) {

if (head == NULL) {

printf("The list is empty.\n");

return;

}

struct Node \*temp = head;

printf("Linked List: ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

// Function to insert a node at the end (for testing purposes)

void insertAtEnd(struct Node \*\*head, int value) {

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

struct Node \*last = \*head;

newNode->data = value;

newNode->next = NULL;

// If the list is empty, the new node becomes the head

if (\*head == NULL) {

\*head = newNode;

return;

}

// Traverse to the last node

while (last->next != NULL) {

last = last->next;

}

// Insert the new node at the end

last->next = newNode;

}

int main() {

struct Node \*head = NULL; // Initialize the linked list as empty

int choice, value;

// Insert some nodes for testing

insertAtEnd(&head, 10);

insertAtEnd(&head, 20);

insertAtEnd(&head, 30);

insertAtEnd(&head, 40);

// Display the initial linked list

displayList(head);

// Menu-driven interface for deletion operations

do {

printf("\nMenu:\n");

printf("1. Delete a node from the beginning\n");

printf("2. Delete a node from the end\n");

printf("3. Display the linked list\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

deleteFromBeginning(&head);

break;

case 2:

deleteFromEnd(&head);

break;

case 3:

displayList(head);

break;

case 4:

printf("Exiting program.\n");

break;

default:

printf("Invalid choice, please try again.\n");

}

} while (choice != 4);

return 0;

}

Q4 Searching in a Linked List: Write a C program to search for a given value in a singly linked list. The program should: 1. Accept a value from the user. 2. Traverse the linked list to find the value. 3. Display whether the value was found and its position

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a node in the singly linked list

struct Node {

int data;

struct Node \*next;

};

// Function to insert a node at the end of the linked list (for testing purposes)

void insertAtEnd(struct Node \*\*head, int value) {

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

struct Node \*last = \*head;

newNode->data = value;

newNode->next = NULL;

// If the list is empty, the new node becomes the head

if (\*head == NULL) {

\*head = newNode;

return;

}

// Traverse to the last node

while (last->next != NULL) {

last = last->next;

}

// Insert the new node at the end

last->next = newNode;

}

// Function to search for a value in the linked list

int searchValue(struct Node \*head, int value) {

struct Node \*temp = head;

int position = 1; // Position starts from 1 (for human readability)

while (temp != NULL) {

if (temp->data == value) {

return position; // Return the position if the value is found

}

temp = temp->next;

position++;

}

return -1; // Return -1 if the value is not found

}

// Function to display the linked list

void displayList(struct Node \*head) {

if (head == NULL) {

printf("The list is empty.\n");

return;

}

struct Node \*temp = head;

printf("Linked List: ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

int main() {

struct Node \*head = NULL; // Initialize the linked list as empty

int value, searchVal, position;

// Insert some nodes for testing

insertAtEnd(&head, 10);

insertAtEnd(&head, 20);

insertAtEnd(&head, 30);

insertAtEnd(&head, 40);

// Display the initial linked list

displayList(head);

// Accept a value to search for

printf("Enter a value to search in the linked list: ");

scanf("%d", &searchVal);

// Search for the value and display the result

position = searchValue(head, searchVal);

if (position == -1) {

printf("Value %d not found in the linked list.\n", searchVal);

} else {

printf("Value %d found at position %d in the linked list.\n", searchVal, position);

}

return 0;

}

Exp 6

Q1 Creating a Doubly Linked List: Write a C program to create a doubly linked list. The program should: 1. Accept values from the user to insert as nodes. 2. Display the doubly linked list (both forward and backward).

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a node in the doubly linked list

struct Node {

int data;

struct Node \*next;

struct Node \*prev;

};

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

void insertAtEnd(struct Node \*\*head, int value) {

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

struct Node \*last = \*head;

newNode->data = value;

newNode->next = NULL;

// If the list is empty, the new node becomes the head

if (\*head == NULL) {

newNode->prev = NULL; // No previous node for the first node

\*head = newNode;

return;

}

// Traverse to the last node

while (last->next != NULL) {

last = last->next;

}

// Insert the new node at the end

last->next = newNode;

newNode->prev = last; // Link the new node's prev pointer to the last node

}

// Function to display the doubly linked list from head to tail (forward)

void displayForward(struct Node \*head) {

struct Node \*temp = head;

if (head == NULL) {

printf("The list is empty.\n");

return;

}

printf("Doubly Linked List (Forward): ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

// Function to display the doubly linked list from tail to head (backward)

void displayBackward(struct Node \*head) {

if (head == NULL) {

printf("The list is empty.\n");

return;

}

struct Node \*temp = head;

// Traverse to the last node

while (temp->next != NULL) {

temp = temp->next;

}

printf("Doubly Linked List (Backward): ");

// Traverse backward and print the list

while (temp != NULL) {

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

temp = temp->prev;

}

printf("NULL\n");

}

int main() {

struct Node \*head = NULL; // Initialize the doubly linked list as empty

int value, n;

// Accept the number of nodes to insert

printf("Enter the number of nodes to insert: ");

scanf("%d", &n);

// Accept values from the user to insert as nodes

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

printf("Enter value for node %d: ", i + 1);

scanf("%d", &value);

insertAtEnd(&head, value);

}

// Display the doubly linked list forward and backward

displayForward(head);

displayBackward(head);

return 0;

}

Q2 Insertion Operations in Doubly Linked List: Write a C program to perform insertion operations in a doubly linked list. The program should: 1. Insert a node at the beginning. 2. Insert a node at the end. 3. Insert a node after a given node.

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a node in the doubly linked list

struct Node {

int data;

struct Node \*next;

struct Node \*prev;

};

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

void insertAtBeginning(struct Node \*\*head, int value) {

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

newNode->data = value;

newNode->prev = NULL;

newNode->next = \*head;

if (\*head != NULL) {

(\*head)->prev = newNode; // Update the previous pointer of the current head node

}

\*head = newNode; // Make the new node the head of the list

}

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

void insertAtEnd(struct Node \*\*head, int value) {

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

struct Node \*last = \*head;

newNode->data = value;

newNode->next = NULL;

if (\*head == NULL) {

newNode->prev = NULL; // If the list is empty, the new node becomes the head

\*head = newNode;

return;

}

// Traverse to the last node

while (last->next != NULL) {

last = last->next;

}

last->next = newNode; // Set the next of the last node to the new node

newNode->prev = last; // Set the previous of the new node to the last node

}

// Function to insert a node after a given node

void insertAfter(struct Node \*\*head, int prevValue, int value) {

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

struct Node \*temp = \*head;

newNode->data = value;

// Traverse the list to find the node with the given value

while (temp != NULL && temp->data != prevValue) {

temp = temp->next;

}

// If the given value is not found, print an error message

if (temp == NULL) {

printf("Node with value %d not found.\n", prevValue);

free(newNode);

return;

}

// Insert the new node after the found node

newNode->next = temp->next; // Set the next of the new node

newNode->prev = temp; // Set the previous of the new node

if (temp->next != NULL) {

temp->next->prev = newNode; // If the new node is not the last node, update its next node's previous pointer

}

temp->next = newNode; // Set the next of the previous node to the new node

}

// Function to display the doubly linked list from head to tail (forward)

void displayForward(struct Node \*head) {

struct Node \*temp = head;

if (head == NULL) {

printf("The list is empty.\n");

return;

}

printf("Doubly Linked List (Forward): ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

// Function to display the doubly linked list from tail to head (backward)

void displayBackward(struct Node \*head) {

if (head == NULL) {

printf("The list is empty.\n");

return;

}

struct Node \*temp = head;

// Traverse to the last node

while (temp->next != NULL) {

temp = temp->next;

}

printf("Doubly Linked List (Backward): ");

// Traverse backward and print the list

while (temp != NULL) {

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

temp = temp->prev;

}

printf("NULL\n");

}

int main() {

struct Node \*head = NULL; // Initialize the doubly linked list as empty

int choice, value, prevValue;

// Menu-driven interface for insertion operations

do {

printf("\nMenu:\n");

printf("1. Insert a node at the beginning\n");

printf("2. Insert a node at the end\n");

printf("3. Insert a node after a given node\n");

printf("4. Display the doubly linked list (Forward)\n");

printf("5. Display the doubly linked list (Backward)\n");

printf("6. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the value to insert at the beginning: ");

scanf("%d", &value);

insertAtBeginning(&head, value);

break;

case 2:

printf("Enter the value to insert at the end: ");

scanf("%d", &value);

insertAtEnd(&head, value);

break;

case 3:

printf("Enter the value after which to insert: ");

scanf("%d", &prevValue);

printf("Enter the value to insert after %d: ", prevValue);

scanf("%d", &value);

insertAfter(&head, prevValue, value);

break;

case 4:

displayForward(head);

break;

case 5:

displayBackward(head);

break;

case 6:

printf("Exiting program.\n");

break;

default:

printf("Invalid choice, please try again.\n");

}

} while (choice != 6);

return 0;

}

Q3 Deletion Operations in Doubly Linked List: Write a C program to perform deletion operations in a doubly linked list. The program should: 1. Delete a node from the beginning. 2. Delete a node from the end. 3. Delete a given node from the list.

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a node in the doubly linked list

struct Node {

int data;

struct Node \*next;

struct Node \*prev;

};

// Function to delete a node from the beginning of the doubly linked list

void deleteFromBeginning(struct Node \*\*head) {

if (\*head == NULL) {

printf("The list is empty. Nothing to delete.\n");

return;

}

struct Node \*temp = \*head;

\*head = (\*head)->next; // Move the head pointer to the next node

if (\*head != NULL) {

(\*head)->prev = NULL; // Set the previous pointer of the new head to NULL

}

free(temp); // Free the memory of the deleted node

printf("Node deleted from the beginning.\n");

}

// Function to delete a node from the end of the doubly linked list

void deleteFromEnd(struct Node \*\*head) {

if (\*head == NULL) {

printf("The list is empty. Nothing to delete.\n");

return;

}

struct Node \*temp = \*head;

// Traverse to the last node

while (temp->next != NULL) {

temp = temp->next;

}

// If the list has only one node

if (temp->prev == NULL) {

\*head = NULL; // List becomes empty

} else {

temp->prev->next = NULL; // Set the previous node's next to NULL

}

free(temp); // Free the memory of the deleted node

printf("Node deleted from the end.\n");

}

// Function to delete a given node from the list by its value

void deleteNode(struct Node \*\*head, int value) {

if (\*head == NULL) {

printf("The list is empty. Nothing to delete.\n");

return;

}

struct Node \*temp = \*head;

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

if (temp->data == value) {

deleteFromBeginning(head);

return;

}

// Traverse the list to find the node with the given value

while (temp != NULL && temp->data != value) {

temp = temp->next;

}

// If the node with the given value was not found

if (temp == NULL) {

printf("Node with value %d not found.\n", value);

return;

}

// Update the previous node's next and the next node's prev

if (temp->next != NULL) {

temp->next->prev = temp->prev;

}

if (temp->prev != NULL) {

temp->prev->next = temp->next;

}

free(temp); // Free the memory of the deleted node

printf("Node with value %d deleted.\n", value);

}

// Function to display the doubly linked list from head to tail (forward)

void displayForward(struct Node \*head) {

struct Node \*temp = head;

if (head == NULL) {

printf("The list is empty.\n");

return;

}

printf("Doubly Linked List (Forward): ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

// Function to display the doubly linked list from tail to head (backward)

void displayBackward(struct Node \*head) {

if (head == NULL) {

printf("The list is empty.\n");

return;

}

struct Node \*temp = head;

// Traverse to the last node

while (temp->next != NULL) {

temp = temp->next;

}

printf("Doubly Linked List (Backward): ");

// Traverse backward and print the list

while (temp != NULL) {

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

temp = temp->prev;

}

printf("NULL\n");

}

int main() {

struct Node \*head = NULL; // Initialize the doubly linked list as empty

int choice, value;

// Menu-driven interface for deletion operations

do {

printf("\nMenu:\n");

printf("1. Delete a node from the beginning\n");

printf("2. Delete a node from the end\n");

printf("3. Delete a given node by value\n");

printf("4. Display the doubly linked list (Forward)\n");

printf("5. Display the doubly linked list (Backward)\n");

printf("6. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

deleteFromBeginning(&head);

break;

case 2:

deleteFromEnd(&head);

break;

case 3:

printf("Enter the value of the node to delete: ");

scanf("%d", &value);

deleteNode(&head, value);

break;

case 4:

displayForward(head);

break;

case 5:

displayBackward(head);

break;

case 6:

printf("Exiting program.\n");

break;

default:

printf("Invalid choice, please try again.\n");

}

} while (choice != 6);

return 0;

}

Q4 Traversing Doubly Linked List: Write a C program to traverse a doubly linked list in both forward and backward directions. The program should: 1. Accept values to create a doubly linked list. 2. Display the list forward and backward.

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a node in the doubly linked list

struct Node {

int data;

struct Node \*next;

struct Node \*prev;

};

// Function to create a new node with a given value

struct Node\* createNode(int value) {

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

if (newNode == NULL) {

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

exit(1);

}

newNode->data = value;

newNode->next = NULL;

newNode->prev = NULL;

return newNode;

}

// Function to add a node at the end of the doubly linked list

void appendNode(struct Node \*\*head, int value) {

struct Node \*newNode = createNode(value);

if (\*head == NULL) {

\*head = newNode; // If the list is empty, new node becomes the head

return;

}

struct Node \*temp = \*head;

while (temp->next != NULL) {

temp = temp->next; // Traverse to the last node

}

temp->next = newNode; // Link the last node to the new node

newNode->prev = temp; // Link the new node's previous pointer to the last node

}

// Function to display the doubly linked list from head to tail (forward)

void displayForward(struct Node \*head) {

if (head == NULL) {

printf("The list is empty.\n");

return;

}

struct Node \*temp = head;

printf("Doubly Linked List (Forward): ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

// Function to display the doubly linked list from tail to head (backward)

void displayBackward(struct Node \*head) {

if (head == NULL) {

printf("The list is empty.\n");

return;

}

struct Node \*temp = head;

// Traverse to the last node

while (temp->next != NULL) {

temp = temp->next;

}

printf("Doubly Linked List (Backward): ");

// Traverse backward and print the list

while (temp != NULL) {

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

temp = temp->prev;

}

printf("NULL\n");

}

int main() {

struct Node \*head = NULL; // Initialize the doubly linked list as empty

int n, value;

// Accept values from the user to create the doubly linked list

printf("Enter the number of nodes to create: ");

scanf("%d", &n);

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

printf("Enter value for node %d: ", i + 1);

scanf("%d", &value);

appendNode(&head, value);

}

// Display the list forward and backward

displayForward(head);

displayBackward(head);

return 0;

}

Exp 7

Q1 Creating a Circular Linked List: Write a C program to create a circular singly linked list. The program should: 1. Accept values from the user to insert as nodes. 2. Display the circular linked list.

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a node in the circular singly linked list

struct Node {

int data;

struct Node\* next;

};

// Function to create a new node with a given value

struct Node\* createNode(int value) {

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

if (newNode == NULL) {

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

exit(1);

}

newNode->data = value;

newNode->next = newNode; // Pointing to itself (circular reference)

return newNode;

}

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

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

struct Node\* newNode = createNode(value);

if (\*head == NULL) {

\*head = newNode; // If the list is empty, make the new node the head

} else {

struct Node\* temp = \*head;

// Traverse to the last node

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

temp = temp->next;

}

temp->next = newNode; // Make the last node point to the new node

newNode->next = \*head; // Make the new node point to the head (circular)

}

}

// Function to display the circular linked list

void displayCircularList(struct Node\* head) {

if (head == NULL) {

printf("The list is empty.\n");

return;

}

struct Node\* temp = head;

printf("Circular Singly Linked List: ");

// Traverse and print the list until we come back to the head

do {

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

temp = temp->next;

} while (temp != head);

printf("(back to head)\n");

}

int main() {

struct Node\* head = NULL; // Initialize the circular linked list as empty

int n, value;

// Accept values from the user to create the circular singly linked list

printf("Enter the number of nodes to create: ");

scanf("%d", &n);

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

printf("Enter value for node %d: ", i + 1);

scanf("%d", &value);

insertNode(&head, value);

}

// Display the circular linked list

displayCircularList(head);

return 0;

}

Q2 Insertion Operations in Circular Linked List: Write a C program to perform insertion operations in a circular singly linked list. The program should: 1. Insert a node at the beginning. 2. Insert a node at the end. 3. Insert a node after a given node

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a node in the circular singly linked list

struct Node {

int data;

struct Node\* next;

};

// Function to create a new node with a given value

struct Node\* createNode(int value) {

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

if (newNode == NULL) {

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

exit(1);

}

newNode->data = value;

newNode->next = newNode; // Point to itself, as it's a circular list

return newNode;

}

// Function to insert a node at the beginning of the circular singly linked list

void insertAtBeginning(struct Node\*\* head, int value) {

struct Node\* newNode = createNode(value);

if (\*head == NULL) {

\*head = newNode; // If the list is empty, make the new node the head

} else {

struct Node\* temp = \*head;

// Traverse to the last node

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

temp = temp->next;

}

temp->next = newNode; // Last node points to new node

newNode->next = \*head; // New node points to the head, maintaining circularity

\*head = newNode; // New node becomes the head

}

}

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

void insertAtEnd(struct Node\*\* head, int value) {

struct Node\* newNode = createNode(value);

if (\*head == NULL) {

\*head = newNode; // If the list is empty, make the new node the head

} else {

struct Node\* temp = \*head;

// Traverse to the last node

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

temp = temp->next;

}

temp->next = newNode; // Last node points to new node

newNode->next = \*head; // New node points to the head, maintaining circularity

}

}

// Function to insert a node after a given node in the circular singly linked list

void insertAfterNode(struct Node\* head, int prevValue, int value) {

struct Node\* temp = head;

// Traverse the list to find the node with the given value

while (temp != NULL && temp->data != prevValue) {

temp = temp->next;

if (temp == head) {

printf("Node with value %d not found.\n", prevValue);

return;

}

}

if (temp != NULL) {

struct Node\* newNode = createNode(value);

newNode->next = temp->next;

temp->next = newNode; // Link the new node after the node with prevValue

}

}

// Function to display the circular singly linked list

void displayCircularList(struct Node\* head) {

if (head == NULL) {

printf("The list is empty.\n");

return;

}

struct Node\* temp = head;

printf("Circular Singly Linked List: ");

// Traverse and print the list until we come back to the head

do {

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

temp = temp->next;

} while (temp != head);

printf("(back to head)\n");

}

int main() {

struct Node\* head = NULL; // Initialize the circular linked list as empty

int choice, value, prevValue;

while (1) {

printf("\nMenu:\n");

printf("1. Insert node at the beginning\n");

printf("2. Insert node at the end\n");

printf("3. Insert node after a given node\n");

printf("4. Display Circular Linked List\n");

printf("5. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to insert at the beginning: ");

scanf("%d", &value);

insertAtBeginning(&head, value);

break;

case 2:

printf("Enter value to insert at the end: ");

scanf("%d", &value);

insertAtEnd(&head, value);

break;

case 3:

printf("Enter value after which to insert: ");

scanf("%d", &prevValue);

printf("Enter value to insert after node %d: ", prevValue);

scanf("%d", &value);

insertAfterNode(head, prevValue, value);

break;

case 4:

displayCircularList(head);

break;

case 5:

printf("Exiting...\n");

exit(0);

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

Q3 Deletion Operations in Circular Linked List: Write a C program to perform deletion operations in a circular singly linked list. The program should: 1. Delete a node from the beginning. 2. Delete a node from the end. 3. Delete a given node from the list.

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a node in the circular singly linked list

struct Node {

int data;

struct Node\* next;

};

// Function to create a new node with a given value

struct Node\* createNode(int value) {

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

if (newNode == NULL) {

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

exit(1);

}

newNode->data = value;

newNode->next = newNode; // Point to itself, as it's a circular list

return newNode;

}

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

void insertAtEnd(struct Node\*\* head, int value) {

struct Node\* newNode = createNode(value);

if (\*head == NULL) {

\*head = newNode; // If the list is empty, make the new node the head

} else {

struct Node\* temp = \*head;

// Traverse to the last node

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

temp = temp->next;

}

temp->next = newNode; // Last node points to new node

newNode->next = \*head; // New node points to the head, maintaining circularity

}

}

// Function to delete a node from the beginning of the circular singly linked list

void deleteFromBeginning(struct Node\*\* head) {

if (\*head == NULL) {

printf("The list is empty. Nothing to delete.\n");

return;

}

struct Node\* temp = \*head;

if ((\*head)->next == \*head) { // Only one node in the list

free(\*head);

\*head = NULL;

} else {

struct Node\* last = \*head;

// Traverse to find the last node

while (last->next != \*head) {

last = last->next;

}

\*head = (\*head)->next; // Update head to the next node

last->next = \*head; // Last node should point to the new head

free(temp); // Free the old head

}

}

// Function to delete a node from the end of the circular singly linked list

void deleteFromEnd(struct Node\*\* head) {

if (\*head == NULL) {

printf("The list is empty. Nothing to delete.\n");

return;

}

struct Node\* temp = \*head;

if ((\*head)->next == \*head) { // Only one node in the list

free(\*head);

\*head = NULL;

} else {

struct Node\* secondLast = \*head;

// Traverse to find the second last node

while (secondLast->next->next != \*head) {

secondLast = secondLast->next;

}

free(secondLast->next); // Free the last node

secondLast->next = \*head; // Second last node points to the head

}

}

// Function to delete a given node from the circular singly linked list

void deleteGivenNode(struct Node\*\* head, int value) {

if (\*head == NULL) {

printf("The list is empty. Nothing to delete.\n");

return;

}

struct Node\* temp = \*head;

struct Node\* prev = NULL;

// Traverse the list to find the node to delete

do {

if (temp->data == value) {

// Node to be deleted is the head

if (temp == \*head) {

deleteFromBeginning(head);

return;

}

// If it's not the head, update previous node's next pointer

prev->next = temp->next;

free(temp);

return;

}

prev = temp;

temp = temp->next;

} while (temp != \*head); // Traverse the entire circular list

printf("Node with value %d not found in the list.\n", value);

}

// Function to display the circular singly linked list

void displayCircularList(struct Node\* head) {

if (head == NULL) {

printf("The list is empty.\n");

return;

}

struct Node\* temp = head;

printf("Circular Singly Linked List: ");

// Traverse and print the list until we come back to the head

do {

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

temp = temp->next;

} while (temp != head);

printf("(back to head)\n");

}

int main() {

struct Node\* head = NULL; // Initialize the circular linked list as empty

int choice, value;

while (1) {

printf("\nMenu:\n");

printf("1. Insert node at the end\n");

printf("2. Delete node from the beginning\n");

printf("3. Delete node from the end\n");

printf("4. Delete a given node\n");

printf("5. Display Circular Linked List\n");

printf("6. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to insert at the end: ");

scanf("%d", &value);

insertAtEnd(&head, value);

break;

case 2:

deleteFromBeginning(&head);

break;

case 3:

deleteFromEnd(&head);

break;

case 4:

printf("Enter value to delete: ");

scanf("%d", &value);

deleteGivenNode(&head, value);

break;

case 5:

displayCircularList(head);

break;

case 6:

printf("Exiting...\n");

exit(0);

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

Q4 Traversing a Circular Linked List: Write a C program to traverse a circular singly linked list. The program should: 1. Accept values to create a circular linked list. 2. Display the list once around the circle (traverse the list).

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a node in the circular singly linked list

struct Node {

int data;

struct Node\* next;

};

// Function to create a new node with a given value

struct Node\* createNode(int value) {

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

if (newNode == NULL) {

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

exit(1);

}

newNode->data = value;

newNode->next = newNode; // Point to itself as it's a circular list

return newNode;

}

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

void insertAtEnd(struct Node\*\* head, int value) {

struct Node\* newNode = createNode(value);

if (\*head == NULL) {

\*head = newNode; // If the list is empty, make the new node the head

} else {

struct Node\* temp = \*head;

// Traverse to the last node

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

temp = temp->next;

}

temp->next = newNode; // Last node points to new node

newNode->next = \*head; // New node points to the head, maintaining circularity

}

}

// Function to traverse the circular singly linked list

void traverseCircularList(struct Node\* head) {

if (head == NULL) {

printf("The list is empty.\n");

return;

}

struct Node\* temp = head;

printf("Circular Singly Linked List: ");

// Traverse and print the list until we come back to the head

do {

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

temp = temp->next;

} while (temp != head); // Stop when we reach the head again

printf("(back to head)\n");

}

int main() {

struct Node\* head = NULL; // Initialize the circular linked list as empty

int choice, value;

while (1) {

printf("\nMenu:\n");

printf("1. Insert node at the end\n");

printf("2. Traverse Circular Linked List\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to insert at the end: ");

scanf("%d", &value);

insertAtEnd(&head, value);

break;

case 2:

traverseCircularList(head);

break;

case 3:

printf("Exiting...\n");

exit(0);

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

Exp 8

Q1 Implementing Stack Using Array: Write a C program to implement a stack using an array. The program should: 1. Push elements onto the stack. 2. Pop elements from the stack. 3. Display the stack content.

#include <stdio.h>

#include <stdlib.h>

#define MAX 5 // Define the maximum size of the stack

// Define the stack structure

struct Stack {

int arr[MAX];

int top;

};

// Function to initialize the stack

void initStack(struct Stack\* stack) {

stack->top = -1; // Set top to -1 to indicate that the stack is empty

}

// Function to check if the stack is full

int isFull(struct Stack\* stack) {

return stack->top == MAX - 1;

}

// Function to check if the stack is empty

int isEmpty(struct Stack\* stack) {

return stack->top == -1;

}

// Function to push an element onto the stack

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

if (isFull(stack)) {

printf("Stack overflow! Cannot push %d\n", value);

} else {

stack->arr[++(stack->top)] = value; // Increment top and insert value

printf("%d pushed to stack\n", value);

}

}

// Function to pop an element from the stack

int pop(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack underflow! No elements to pop.\n");

return -1; // Return an invalid value to indicate underflow

} else {

return stack->arr[(stack->top)--]; // Return the top element and decrement top

}

}

// Function to display the contents of the stack

void displayStack(struct Stack\* stack) {

if (isEmpty(stack)) {

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

} else {

printf("Stack contents: ");

for (int i = stack->top; i >= 0; i--) {

printf("%d ", stack->arr[i]);

}

printf("\n");

}

}

int main() {

struct Stack stack; // Create a stack

int choice, value;

initStack(&stack); // Initialize the stack

while (1) {

printf("\nMenu:\n");

printf("1. Push\n");

printf("2. Pop\n");

printf("3. Display Stack\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to push onto the stack: ");

scanf("%d", &value);

push(&stack, value);

break;

case 2:

value = pop(&stack);

if (value != -1) {

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

}

break;

case 3:

displayStack(&stack);

break;

case 4:

printf("Exiting...\n");

exit(0);

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

Q2 Stack Operations: Write a C program to implement the basic stack operations (push, pop, peek) using an array. The program should: 1. Push an element onto the stack. 2. Pop an element from the stack. 3. Display the top element without popping.

#include <stdio.h>

#include <stdlib.h>

#define MAX 5 // Define the maximum size of the stack

// Define the stack structure

struct Stack {

int arr[MAX];

int top;

};

// Function to initialize the stack

void initStack(struct Stack\* stack) {

stack->top = -1; // Set top to -1 to indicate that the stack is empty

}

// Function to check if the stack is full

int isFull(struct Stack\* stack) {

return stack->top == MAX - 1;

}

// Function to check if the stack is empty

int isEmpty(struct Stack\* stack) {

return stack->top == -1;

}

// Function to push an element onto the stack

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

if (isFull(stack)) {

printf("Stack overflow! Cannot push %d\n", value);

} else {

stack->arr[++(stack->top)] = value; // Increment top and insert value

printf("%d pushed to stack\n", value);

}

}

// Function to pop an element from the stack

int pop(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack underflow! No elements to pop.\n");

return -1; // Return an invalid value to indicate underflow

} else {

return stack->arr[(stack->top)--]; // Return the top element and decrement top

}

}

// Function to peek the top element of the stack

int peek(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack is empty. No top element.\n");

return -1; // Return an invalid value to indicate empty stack

} else {

return stack->arr[stack->top]; // Return the top element

}

}

// Function to display the contents of the stack

void displayStack(struct Stack\* stack) {

if (isEmpty(stack)) {

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

} else {

printf("Stack contents: ");

for (int i = stack->top; i >= 0; i--) {

printf("%d ", stack->arr[i]);

}

printf("\n");

}

}

int main() {

struct Stack stack; // Create a stack

int choice, value;

initStack(&stack); // Initialize the stack

while (1) {

printf("\nMenu:\n");

printf("1. Push\n");

printf("2. Pop\n");

printf("3. Peek (Top element)\n");

printf("4. Display Stack\n");

printf("5. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to push onto the stack: ");

scanf("%d", &value);

push(&stack, value);

break;

case 2:

value = pop(&stack);

if (value != -1) {

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

}

break;

case 3:

value = peek(&stack);

if (value != -1) {

printf("Top element: %d\n", value);

}

break;

case 4:

displayStack(&stack);

break;

case 5:

printf("Exiting...\n");

exit(0);

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

Q3 Stack Using Linked List: Write a C program to implement a stack using a singly linked list. The program should: 1. Push elements onto the stack. 2. Pop elements from the stack. 3. Display the stack content.

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a stack node

struct Node {

int data;

struct Node\* next;

};

// Function to push an element onto the stack

void push(struct Node\*\* top, int value) {

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

if (newNode == NULL) {

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

return;

}

newNode->data = value;

newNode->next = \*top; // New node points to the current top node

\*top = newNode; // Update top to the new node

printf("%d pushed to stack\n", value);

}

// Function to pop an element from the stack

int pop(struct Node\*\* top) {

if (\*top == NULL) {

printf("Stack underflow! No elements to pop.\n");

return -1; // Return an invalid value to indicate underflow

}

struct Node\* temp = \*top;

int value = temp->data;

\*top = temp->next; // Update top to the next node

free(temp); // Free the memory of the popped node

return value;

}

// Function to display the stack content

void displayStack(struct Node\* top) {

if (top == NULL) {

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

return;

}

printf("Stack contents: ");

struct Node\* current = top;

while (current != NULL) {

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

current = current->next;

}

printf("\n");

}

int main() {

struct Node\* top = NULL; // Initialize the stack (top is NULL for an empty stack)

int choice, value;

while (1) {

printf("\nMenu:\n");

printf("1. Push\n");

printf("2. Pop\n");

printf("3. Display Stack\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to push onto the stack: ");

scanf("%d", &value);

push(&top, value);

break;

case 2:

value = pop(&top);

if (value != -1) {

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

}

break;

case 3:

displayStack(top);

break;

case 4:

printf("Exiting...\n");

exit(0);

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

Exp no 9

Q1 Implementing Queue Using Array: Write a C program to implement a queue using an array. The program should: 1. Enqueue elements into the queue. 2. Dequeue elements from the queue. 3. Display the queue content.

#include <stdio.h>

#include <stdlib.h>

#define MAX 5 // Define the maximum size of the queue

// Define the structure for the queue

struct Queue {

int arr[MAX];

int front, rear;

};

// Function to initialize the queue

void initQueue(struct Queue\* q) {

q->front = -1; // The queue is empty

q->rear = -1;

}

// Function to check if the queue is full

int isFull(struct Queue\* q) {

return q->rear == MAX - 1;

}

// Function to check if the queue is empty

int isEmpty(struct Queue\* q) {

return q->front == -1;

}

// Function to enqueue an element into the queue

void enqueue(struct Queue\* q, int value) {

if (isFull(q)) {

printf("Queue overflow! Cannot enqueue %d\n", value);

} else {

if (q->front == -1) // If the queue is empty, set front to 0

q->front = 0;

q->rear++;

q->arr[q->rear] = value;

printf("%d enqueued to the queue\n", value);

}

}

// Function to dequeue an element from the queue

int dequeue(struct Queue\* q) {

if (isEmpty(q)) {

printf("Queue underflow! No elements to dequeue.\n");

return -1; // Return an invalid value to indicate underflow

} else {

int value = q->arr[q->front];

if (q->front == q->rear) { // If there is only one element

q->front = q->rear = -1; // Reset the queue

} else {

q->front++;

}

return value;

}

}

// Function to display the queue content

void displayQueue(struct Queue\* q) {

if (isEmpty(q)) {

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

} else {

printf("Queue contents: ");

for (int i = q->front; i <= q->rear; i++) {

printf("%d ", q->arr[i]);

}

printf("\n");

}

}

int main() {

struct Queue q; // Create a queue

int choice, value;

initQueue(&q); // Initialize the queue

while (1) {

printf("\nMenu:\n");

printf("1. Enqueue\n");

printf("2. Dequeue\n");

printf("3. Display Queue\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to enqueue into the queue: ");

scanf("%d", &value);

enqueue(&q, value);

break;

case 2:

value = dequeue(&q);

if (value != -1) {

printf("Dequeued element: %d\n", value);

}

break;

case 3:

displayQueue(&q);

break;

case 4:

printf("Exiting...\n");

exit(0);

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

Q2 Queue Operations: Write a C program to implement basic queue operations (enqueue, dequeue, front) using an array. The program should: 1. Enqueue an element. 2. Dequeue an element. 3. Display the front element.

#include <stdio.h>

#include <stdlib.h>

#define MAX 5 // Define the maximum size of the queue

// Define the structure for the queue

struct Queue {

int arr[MAX];

int front, rear;

};

// Function to initialize the queue

void initQueue(struct Queue\* q) {

q->front = -1; // The queue is empty

q->rear = -1;

}

// Function to check if the queue is full

int isFull(struct Queue\* q) {

return q->rear == MAX - 1;

}

// Function to check if the queue is empty

int isEmpty(struct Queue\* q) {

return q->front == -1;

}

// Function to enqueue an element into the queue

void enqueue(struct Queue\* q, int value) {

if (isFull(q)) {

printf("Queue overflow! Cannot enqueue %d\n", value);

} else {

if (q->front == -1) // If the queue is empty, set front to 0

q->front = 0;

q->rear++;

q->arr[q->rear] = value;

printf("%d enqueued to the queue\n", value);

}

}

// Function to dequeue an element from the queue

int dequeue(struct Queue\* q) {

if (isEmpty(q)) {

printf("Queue underflow! No elements to dequeue.\n");

return -1; // Return an invalid value to indicate underflow

} else {

int value = q->arr[q->front];

if (q->front == q->rear) { // If there is only one element

q->front = q->rear = -1; // Reset the queue

} else {

q->front++;

}

return value;

}

}

// Function to display the front element of the queue

int front(struct Queue\* q) {

if (isEmpty(q)) {

printf("Queue is empty. No front element.\n");

return -1;

} else {

return q->arr[q->front];

}

}

int main() {

struct Queue q; // Create a queue

int choice, value;

initQueue(&q); // Initialize the queue

while (1) {

printf("\nMenu:\n");

printf("1. Enqueue\n");

printf("2. Dequeue\n");

printf("3. Display Front Element\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to enqueue into the queue: ");

scanf("%d", &value);

enqueue(&q, value);

break;

case 2:

value = dequeue(&q);

if (value != -1) {

printf("Dequeued element: %d\n", value);

}

break;

case 3:

value = front(&q);

if (value != -1) {

printf("Front element: %d\n", value);

}

break;

case 4:

printf("Exiting...\n");

exit(0);

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

Q3 Queue Using Linked List: Write a C program to implement a queue using a singly linked list. The program should: 1. Enqueue elements into the queue. 2. Dequeue elements from the queue. 3. Display the queue content.

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a queue node

struct Node {

int data;

struct Node\* next;

};

// Define the structure for the queue

struct Queue {

struct Node\* front;

struct Node\* rear;

};

// Function to initialize the queue

void initQueue(struct Queue\* q) {

q->front = q->rear = NULL;

}

// Function to check if the queue is empty

int isEmpty(struct Queue\* q) {

return q->front == NULL;

}

// Function to enqueue an element into the queue

void enqueue(struct Queue\* q, int value) {

// Create a new node

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

newNode->data = value;

newNode->next = NULL;

// If the queue is empty, both front and rear point to the new node

if (q->rear == NULL) {

q->front = q->rear = newNode;

printf("%d enqueued to the queue\n", value);

return;

}

// Otherwise, add the new node at the rear and move rear to the new node

q->rear->next = newNode;

q->rear = newNode;

printf("%d enqueued to the queue\n", value);

}

// Function to dequeue an element from the queue

int dequeue(struct Queue\* q) {

// If the queue is empty, return -1 (queue underflow)

if (isEmpty(q)) {

printf("Queue underflow! No elements to dequeue.\n");

return -1;

}

// Get the front node and its value

struct Node\* temp = q->front;

int value = temp->data;

// Move front to the next node

q->front = q->front->next;

// If the queue becomes empty, reset rear to NULL

if (q->front == NULL) {

q->rear = NULL;

}

// Free the dequeued node

free(temp);

return value;

}

// Function to display the queue content

void displayQueue(struct Queue\* q) {

if (isEmpty(q)) {

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

return;

}

struct Node\* temp = q->front;

printf("Queue contents: ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("\n");

}

int main() {

struct Queue q; // Create a queue

int choice, value;

initQueue(&q); // Initialize the queue

while (1) {

printf("\nMenu:\n");

printf("1. Enqueue\n");

printf("2. Dequeue\n");

printf("3. Display Queue\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to enqueue into the queue: ");

scanf("%d", &value);

enqueue(&q, value);

break;

case 2:

value = dequeue(&q);

if (value != -1) {

printf("Dequeued element: %d\n", value);

}

break;

case 3:

displayQueue(&q);

break;

case 4:

printf("Exiting...\n");

exit(0);

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

Q4 Circular Queue: Write a C program to implement a circular queue using an array. The program should: 1. Enqueue elements into the circular queue. 2. Dequeue elements from the circular queue. 3. Display the queue content.

#include <stdio.h>

#include <stdlib.h>

#define MAX 5 // Define the maximum size of the queue

// Define the structure for the circular queue

struct Queue {

int arr[MAX];

int front, rear;

};

// Function to initialize the queue

void initQueue(struct Queue\* q) {

q->front = q->rear = -1;

}

// Function to check if the queue is full

int isFull(struct Queue\* q) {

return (q->rear + 1) % MAX == q->front;

}

// Function to check if the queue is empty

int isEmpty(struct Queue\* q) {

return q->front == -1;

}

// Function to enqueue an element into the circular queue

void enqueue(struct Queue\* q, int value) {

if (isFull(q)) {

printf("Queue overflow! Cannot enqueue %d\n", value);

} else {

if (q->front == -1) { // If the queue is empty, set front to 0

q->front = 0;

}

q->rear = (q->rear + 1) % MAX; // Circular increment of rear

q->arr[q->rear] = value;

printf("%d enqueued to the queue\n", value);

}

}

// Function to dequeue an element from the circular queue

int dequeue(struct Queue\* q) {

if (isEmpty(q)) {

printf("Queue underflow! No elements to dequeue.\n");

return -1; // Return an invalid value to indicate underflow

} else {

int value = q->arr[q->front];

if (q->front == q->rear) { // Only one element left in the queue

q->front = q->rear = -1; // Reset the queue

} else {

q->front = (q->front + 1) % MAX; // Circular increment of front

}

return value;

}

}

// Function to display the queue content

void displayQueue(struct Queue\* q) {

if (isEmpty(q)) {

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

return;

}

printf("Queue contents: ");

int i = q->front;

while (i != q->rear) {

printf("%d ", q->arr[i]);

i = (i + 1) % MAX; // Circular increment of index

}

printf("%d\n", q->arr[q->rear]); // Print the rear element

}

int main() {

struct Queue q; // Create a queue

int choice, value;

initQueue(&q); // Initialize the queue

while (1) {

printf("\nMenu:\n");

printf("1. Enqueue\n");

printf("2. Dequeue\n");

printf("3. Display Queue\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to enqueue into the queue: ");

scanf("%d", &value);

enqueue(&q, value);

break;

case 2:

value = dequeue(&q);

if (value != -1) {

printf("Dequeued element: %d\n", value);

}

break;

case 3:

displayQueue(&q);

break;

case 4:

printf("Exiting...\n");

exit(0);

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

Exp 10

Q1 Infix to Postfix Conversion: Write a C program to convert an infix expression to a postfix expression using a stack. The program should: 1. Accept an infix expression with operators and operands. 2. Convert the expression to postfix notation. 3. Display the resulting postfix expression.

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#define MAX 50

// Define a structure for the stack

struct Stack {

int top;

char arr[MAX];

};

// Function to initialize the stack

void initStack(struct Stack\* s) {

s->top = -1;

}

// Function to check if the stack is empty

int isEmpty(struct Stack\* s) {

return s->top == -1;

}

// Function to check if the stack is full

int isFull(struct Stack\* s) {

return s->top == (MAX - 1);

}

// Function to push an element onto the stack

void push(struct Stack\* s, char value) {

if (isFull(s)) {

printf("Stack Overflow!\n");

return;

}

s->arr[++(s->top)] = value;

}

// Function to pop an element from the stack

char pop(struct Stack\* s) {

if (isEmpty(s)) {

printf("Stack Underflow!\n");

return -1;

}

return s->arr[(s->top)--];

}

// Function to peek at the top element of the stack

char peek(struct Stack\* s) {

if (isEmpty(s)) {

return -1;

}

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

}

// Function to check the precedence of operators

int precedence(char c) {

if (c == '+' || c == '-') {

return 1;

}

if (c == '\*' || c == '/') {

return 2;

}

return 0; // '(' has the lowest precedence

}

// Function to convert infix expression to postfix expression

void infixToPostfix(char\* infix, char\* postfix) {

struct Stack s;

initStack(&s);

int k = 0; // Index for postfix array

for (int i = 0; infix[i] != '\0'; i++) {

char current = infix[i];

// If the character is an operand, add it to the postfix expression

if (isalnum(current)) {

postfix[k++] = current;

}

// If the character is '(', push it to the stack

else if (current == '(') {

push(&s, current);

}

// If the character is ')', pop and append to postfix until '(' is found

else if (current == ')') {

while (!isEmpty(&s) && peek(&s) != '(') {

postfix[k++] = pop(&s);

}

pop(&s); // Pop the '(' from the stack

}

// If the character is an operator

else {

while (!isEmpty(&s) && precedence(peek(&s)) >= precedence(current)) {

postfix[k++] = pop(&s);

}

push(&s, current); // Push the current operator onto the stack

}

}

// Pop all remaining operators from the stack

while (!isEmpty(&s)) {

postfix[k++] = pop(&s);

}

postfix[k] = '\0'; // Null-terminate the postfix expression

}

int main() {

char infix[MAX], postfix[MAX];

// Accept an infix expression from the user

printf("Enter an infix expression: ");

scanf("%s", infix);

// Convert the infix expression to postfix

infixToPostfix(infix, postfix);

// Display the resulting postfix expression

printf("Postfix expression: %s\n", postfix);

return 0;

}

Q2 Postfix Evaluation: Write a C program to evaluate a postfix expression using a stack. The program should: 1. Accept a postfix expression with operands and operators. 2. Evaluate the expression using a stack. 3. Display the result of the evaluation.

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#define MAX 50

// Define a structure for the stack

struct Stack {

int top;

int arr[MAX];

};

// Function to initialize the stack

void initStack(struct Stack\* s) {

s->top = -1;

}

// Function to check if the stack is empty

int isEmpty(struct Stack\* s) {

return s->top == -1;

}

// Function to push an element onto the stack

void push(struct Stack\* s, int value) {

if (s->top == MAX - 1) {

printf("Stack Overflow!\n");

return;

}

s->arr[++(s->top)] = value;

}

// Function to pop an element from the stack

int pop(struct Stack\* s) {

if (isEmpty(s)) {

printf("Stack Underflow!\n");

return -1;

}

return s->arr[(s->top)--];

}

// Function to evaluate a postfix expression

int evaluatePostfix(char\* postfix) {

struct Stack s;

initStack(&s);

for (int i = 0; postfix[i] != '\0'; i++) {

char current = postfix[i];

// If the character is an operand (number), push it onto the stack

if (isdigit(current)) {

push(&s, current - '0'); // Convert char to int

}

// If the character is an operator

else if (current == '+' || current == '-' || current == '\*' || current == '/') {

int operand2 = pop(&s); // Pop the second operand

int operand1 = pop(&s); // Pop the first operand

int result;

// Perform the operation and push the result back onto the stack

switch (current) {

case '+': result = operand1 + operand2; break;

case '-': result = operand1 - operand2; break;

case '\*': result = operand1 \* operand2; break;

case '/': result = operand1 / operand2; break;

default: result = 0; break;

}

push(&s, result); // Push the result back onto the stack

}

}

// The final result will be the only element left in the stack

return pop(&s);

}

int main() {

char postfix[MAX];

// Accept a postfix expression from the user

printf("Enter a postfix expression: ");

scanf("%s", postfix);

// Evaluate the postfix expression

int result = evaluatePostfix(postfix);

// Display the result of the evaluation

printf("The result of the postfix expression is: %d\n", result);

return 0;

}

Exp no 11

Q1

Linear Search: Write a C program to implement linear search. The program should: 1. Accept a list of integers from the user. 2. Search for a target value in the list. 3. Display the index of the target if found or a message if not found.

#include <stdio.h>

#define MAX 100

// Function to perform linear search

int linearSearch(int arr[], int n, int target) {

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

if (arr[i] == target) {

return i; // Return the index if target is found

}

}

return -1; // Return -1 if target is not found

}

int main() {

int arr[MAX], n, target, result;

// Accept the number of elements in the array

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

scanf("%d", &n);

// Accept the elements of the array

printf("Enter %d integers:\n", n);

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

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

}

// Accept the target value to search for

printf("Enter the target value to search: ");

scanf("%d", &target);

// Perform linear search

result = linearSearch(arr, n, target);

// Display the result

if (result != -1) {

printf("Target %d found at index %d.\n", target, result);

} else {

printf("Target %d not found in the list.\n", target);

}

return 0;

}

Q2 Binary Search: Write a C program to implement binary search on a sorted array. The program should: 1. Accept a sorted list of integers. 2. Search for a target value in the list using binary search. 3. Display the index of the target if found or a message if not found.

#include <stdio.h>

#define MAX 100

// Function to perform binary search

int binarySearch(int arr[], int low, int high, int target) {

while (low <= high) {

int mid = low + (high - low) / 2;

// Check if the target is present at mid

if (arr[mid] == target) {

return mid; // Return the index if target is found

}

// If target is greater, ignore the left half

if (arr[mid] < target) {

low = mid + 1;

}

// If target is smaller, ignore the right half

else {

high = mid - 1;

}

}

return -1; // Return -1 if target is not found

}

int main() {

int arr[MAX], n, target, result;

// Accept the number of elements in the sorted array

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

scanf("%d", &n);

// Accept the elements of the array (must be sorted)

printf("Enter %d sorted integers:\n", n);

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

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

}

// Accept the target value to search for

printf("Enter the target value to search: ");

scanf("%d", &target);

// Perform binary search

result = binarySearch(arr, 0, n - 1, target);

// Display the result

if (result != -1) {

printf("Target %d found at index %d.\n", target, result);

} else {

printf("Target %d not found in the list.\n", target);

}

return 0;

}

Q3 Hashing (Chaining): Write a C program to implement hashing using chaining. The program should: 1. Create a hash table using an array of linked lists. 2. Implement the insert and search operations using hashing. 3. Display the hash table after insertion and search operations.

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a node in the linked list

struct Node {

int data;

struct Node\* next;

};

// Define the size of the hash table

#define SIZE 10

// 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;

}

// Hash function to compute the index for a given key

int hashFunction(int key) {

return key % SIZE; // Simple modulo operation for hashing

}

// Function to insert a key into the hash table

void insert(struct Node\* hashTable[], int key) {

int index = hashFunction(key); // Compute the hash index

struct Node\* newNode = createNode(key); // Create a new node

newNode->next = hashTable[index]; // Link the new node to the existing list

hashTable[index] = newNode; // Set the new node as the head of the list

}

// Function to search for a key in the hash table

int search(struct Node\* hashTable[], int key) {

int index = hashFunction(key); // Compute the hash index

struct Node\* current = hashTable[index]; // Get the list at the computed index

while (current != NULL) {

if (current->data == key) {

return 1; // Return 1 if the key is found

}

current = current->next;

}

return 0; // Return 0 if the key is not found

}

// Function to display the hash table

void display(struct Node\* hashTable[]) {

printf("Hash Table:\n");

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

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

struct Node\* current = hashTable[i];

while (current != NULL) {

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

current = current->next;

}

printf("NULL\n");

}

}

int main() {

struct Node\* hashTable[SIZE] = { NULL }; // Initialize the hash table

// Insert some elements into the hash table

insert(hashTable, 10);

insert(hashTable, 20);

insert(hashTable, 15);

insert(hashTable, 25);

insert(hashTable, 5);

// Display the hash table after insertions

display(hashTable);

// Search for some elements

int keyToSearch = 15;

if (search(hashTable, keyToSearch)) {

printf("Key %d found in the hash table.\n", keyToSearch);

} else {

printf("Key %d not found in the hash table.\n", keyToSearch);

}

keyToSearch = 30;

if (search(hashTable, keyToSearch)) {

printf("Key %d found in the hash table.\n", keyToSearch);

} else {

printf("Key %d not found in the hash table.\n", keyToSearch);

}

return 0;

}

Exp 12

Bubble Sort: Write a C program to implement the bubble sort algorithm. The program should: 1. Accept a list of integers from the user. 2. Sort the list using the bubble sort algorithm. 3. Display the sorted list.

#include <stdio.h>

// Function to perform bubble sort

void bubbleSort(int arr[], int n) {

int temp;

// Outer loop for each pass

for (int i = 0; i < n - 1; i++) {

// Inner loop for comparing and swapping adjacent elements

for (int j = 0; j < n - i - 1; j++) {

// If the element is greater than the next element, swap them

if (arr[j] > arr[j + 1]) {

temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

// Function to display the array

void display(int arr[], int n) {

printf("Sorted list: ");

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

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

int n;

// Accept the number of elements in the array

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

scanf("%d", &n);

int arr[n];

// Accept the list of integers from the user

printf("Enter %d integers:\n", n);

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

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

}

// Call the bubbleSort function to sort the array

bubbleSort(arr, n);

// Display the sorted array

display(arr, n);

return 0;

}

Q2 Selection Sort: Write a C program to implement the selection sort algorithm. The program should: 1. Accept a list of integers from the user. 2. Sort the list using the selection sort algorithm. 3. Display the sorted list.

#include <stdio.h>

// Function to perform selection sort

void selectionSort(int arr[], int n) {

int temp, minIndex;

// Outer loop for each pass

for (int i = 0; i < n - 1; i++) {

minIndex = i; // Assume the current index is the minimum

// Inner loop to find the minimum element in the unsorted part

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

if (arr[j] < arr[minIndex]) {

minIndex = j; // Update the index of the minimum element

}

}

// Swap the found minimum element with the first element of the unsorted part

if (minIndex != i) {

temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = temp;

}

}

}

// Function to display the array

void display(int arr[], int n) {

printf("Sorted list: ");

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

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

int n;

// Accept the number of elements in the array

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

scanf("%d", &n);

int arr[n];

// Accept the list of integers from the user

printf("Enter %d integers:\n", n);

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

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

}

// Call the selectionSort function to sort the array

selectionSort(arr, n);

// Display the sorted array

display(arr, n);

return 0;

}#include <stdio.h>

// Function to perform selection sort

void selectionSort(int arr[], int n) {

int temp, minIndex;

// Outer loop for each pass

for (int i = 0; i < n - 1; i++) {

minIndex = i; // Assume the current index is the minimum

// Inner loop to find the minimum element in the unsorted part

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

if (arr[j] < arr[minIndex]) {

minIndex = j; // Update the index of the minimum element

}

}

// Swap the found minimum element with the first element of the unsorted part

if (minIndex != i) {

temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = temp;

}

}

}

// Function to display the array

void display(int arr[], int n) {

printf("Sorted list: ");

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

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

int n;

// Accept the number of elements in the array

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

scanf("%d", &n);

int arr[n];

// Accept the list of integers from the user

printf("Enter %d integers:\n", n);

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

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

}

// Call the selectionSort function to sort the array

selectionSort(arr, n);

// Display the sorted array

display(arr, n);

return 0;

}

Q3 Insertion Sort: Write a C program to implement the insertion sort algorithm. The program should: 1. Accept a list of integers from the user. 2. Sort the list using the insertion sort algorithm. 3. Display the sorted list.

#include <stdio.h>

// Function to perform insertion sort

void insertionSort(int arr[], int n) {

int key, j;

// Outer loop for each element

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

key = arr[i]; // Store the current element to be inserted

j = i - 1;

// Shift elements of arr[0..i-1] that are greater than key

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j]; // Shift element to the right

j = j - 1;

}

arr[j + 1] = key; // Insert the key at its correct position

}

}

// Function to display the array

void display(int arr[], int n) {

printf("Sorted list: ");

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

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

int n;

// Accept the number of elements in the array

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

scanf("%d", &n);

int arr[n];

// Accept the list of integers from the user

printf("Enter %d integers:\n", n);

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

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

}

// Call the insertionSort function to sort the array

insertionSort(arr, n);

// Display the sorted array

display(arr, n);

return 0;

}

Exp no 13

Q1 Merge Sort: Write a C program to implement the merge sort algorithm. The program should: 1. Accept a list of integers from the user. 2. Sort the list using the merge sort algorithm. 3. Display the sorted list.

#include <stdio.h>

// Function to merge two subarrays

void merge(int arr[], int left, int right) {

if (left < right) {

int mid = (left + right) / 2;

// Recursively divide the array into two halves

merge(arr, left, mid);

merge(arr, mid + 1, right);

// Merge the sorted halves

int n1 = mid - left + 1;

int n2 = right - mid;

int leftArr[n1], rightArr[n2];

// Copy data to temporary arrays

for (int i = 0; i < n1; i++) {

leftArr[i] = arr[left + i];

}

for (int j = 0; j < n2; j++) {

rightArr[j] = arr[mid + 1 + j];

}

// Merge the temporary arrays back into the original array

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (leftArr[i] <= rightArr[j]) {

arr[k] = leftArr[i];

i++;

} else {

arr[k] = rightArr[j];

j++;

}

k++;

}

// Copy the remaining elements of leftArr[], if any

while (i < n1) {

arr[k] = leftArr[i];

i++;

k++;

}

// Copy the remaining elements of rightArr[], if any

while (j < n2) {

arr[k] = rightArr[j];

j++;

k++;

}

}

}

// Function to display the array

void display(int arr[], int n) {

printf("Sorted list: ");

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

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

int n;

// Accept the number of elements in the array

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

scanf("%d", &n);

int arr[n];

// Accept the list of integers from the user

printf("Enter %d integers:\n", n);

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

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

}

// Call the merge function to sort the array

merge(arr, 0, n - 1);

// Display the sorted array

display(arr, n);

return 0;

}

Q2 Quick Sort: Write a C program to implement the quick sort algorithm. The program should: 1. Accept a list of integers from the user. 2. Sort the list using the quick sort algorithm. 3. Display the sorted list.

#include <stdio.h>

// Function to perform the partitioning

int partition(int arr[], int low, int high) {

int pivot = arr[high]; // Taking the last element as pivot

int i = (low - 1); // Index of the smaller element

// Traverse through all elements, compare each with pivot

for (int j = low; j < high; j++) {

if (arr[j] <= pivot) {

i++;

// Swap arr[i] and arr[j]

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// Swap arr[i + 1] and arr[high] (pivot element)

int temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

return (i + 1); // Return the partition index

}

// Function to implement the quick sort algorithm

void quickSort(int arr[], int low, int high) {

if (low < high) {

// Find pivot element such that element smaller than the pivot are on the left

// and elements greater than the pivot are on the right

int pi = partition(arr, low, high);

// Recursively apply quick sort to the subarrays

quickSort(arr, low, pi - 1); // Before pivot

quickSort(arr, pi + 1, high); // After pivot

}

}

// Function to display the array

void display(int arr[], int n) {

printf("Sorted list: ");

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

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

int n;

// Accept the number of elements in the array

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

scanf("%d", &n);

int arr[n];

// Accept the list of integers from the user

printf("Enter %d integers:\n", n);

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

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

}

// Call the quickSort function to sort the array

quickSort(arr, 0, n - 1);

// Display the sorted array

display(arr, n);

return 0;

}

Q3 Heap Sort: Write a C program to implement the heap sort algorithm. The program should: 1. Accept a list of integers from the user. 2. Sort the list using the heap sort algorithm. 3. Display the sorted list.

#include <stdio.h>

// Function to heapify a subtree rooted at index i

void heapify(int arr[], int n, int i) {

int largest = i; // Initialize largest as root

int left = 2 \* i + 1; // Left child

int right = 2 \* i + 2; // Right child

// If left child is larger than root

if (left < n && arr[left] > arr[largest]) {

largest = left;

}

// If right child is larger than largest so far

if (right < n && arr[right] > arr[largest]) {

largest = right;

}

// If largest is not root

if (largest != i) {

// Swap arr[i] and arr[largest]

int temp = arr[i];

arr[i] = arr[largest];

arr[largest] = temp;

// Recursively heapify the affected subtree

heapify(arr, n, largest);

}

}

// Function to implement heap sort

void heapSort(int arr[], int n) {

// Build a max heap

for (int i = n / 2 - 1; i >= 0; i--) {

heapify(arr, n, i);

}

// One by one extract elements from heap

for (int i = n - 1; i > 0; i--) {

// Swap the root (maximum element) with the last element

int temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

// Call heapify on the reduced heap

heapify(arr, i, 0);

}

}

// Function to display the array

void display(int arr[], int n) {

printf("Sorted list: ");

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

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

int n;

// Accept the number of elements in the array

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

scanf("%d", &n);

int arr[n];

// Accept the list of integers from the user

printf("Enter %d integers:\n", n);

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

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

}

// Call the heapSort function to sort the array

heapSort(arr, n);

// Display the sorted array

display(arr, n);

return 0;

}

Exp 14

Binary Search Tree Formation: Write a C program to create a binary search tree (BST). The program should: 1. Accept a list of integers from the user. 2. Insert elements into the BST following BST properties. 3. Display the in-order traversal of the BST.

#include <stdio.h>

#include <stdlib.h>

// Definition of the node structure

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

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

struct Node\* createNode(int data) {

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

newNode->data = data;

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

return newNode;

}

// Function to insert a new node into the BST

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

// If the tree is empty, return a new node

if (root == NULL) {

return createNode(data);

}

// Otherwise, recur down the tree

if (data < root->data) {

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

} else {

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

}

// Return the (unchanged) node pointer

return root;

}

// Function for in-order traversal of the BST (left, root, right)

void inorderTraversal(struct Node\* root) {

if (root != NULL) {

inorderTraversal(root->left); // Visit left subtree

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

inorderTraversal(root->right); // Visit right subtree

}

}

int main() {

struct Node\* root = NULL;

int n, value;

// Accept the number of elements to be inserted into the BST

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

scanf("%d", &n);

// Accept the elements and insert them into the BST

printf("Enter the elements:\n");

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

scanf("%d", &value);

root = insert(root, value);

}

// Display the in-order traversal of the BST

printf("In-order Traversal of the BST: ");

inorderTraversal(root);

printf("\n");

return 0;

}

Q2 Binary Search Tree Traversal: Write a C program to implement different types of binary search tree (BST) traversal. The program should: 1. Perform In-order, Pre-order, and Post-order traversal. 2. Display the results of all three traversals.

#include <stdio.h>

#include <stdlib.h>

// Definition of the node structure

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

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

struct Node\* createNode(int data) {

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

newNode->data = data;

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

return newNode;

}

// Function to insert a new node into the BST

struct Node\* insert(struct 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;

}

// In-order traversal (left, root, right)

void inorderTraversal(struct Node\* root) {

if (root != NULL) {

inorderTraversal(root->left); // Visit left subtree

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

inorderTraversal(root->right); // Visit right subtree

}

}

// Pre-order traversal (root, left, right)

void preorderTraversal(struct Node\* root) {

if (root != NULL) {

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

preorderTraversal(root->left); // Visit left subtree

preorderTraversal(root->right);// Visit right subtree

}

}

// Post-order traversal (left, right, root)

void postorderTraversal(struct Node\* root) {

if (root != NULL) {

postorderTraversal(root->left); // Visit left subtree

postorderTraversal(root->right);// Visit right subtree

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

}

}

int main() {

struct Node\* root = NULL;

int n, value;

// Accept the number of elements to be inserted into the BST

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

scanf("%d", &n);

// Accept the elements and insert them into the BST

printf("Enter the elements:\n");

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

scanf("%d", &value);

root = insert(root, value);

}

// Display the different traversals

printf("\nIn-order Traversal: ");

inorderTraversal(root);

printf("\n");

printf("Pre-order Traversal: ");

preorderTraversal(root);

printf("\n");

printf("Post-order Traversal: ");

postorderTraversal(root);

printf("\n");

return 0;

}

Q3 Search in Binary Search Tree: Write a C program to search for an element in a binary search tree (BST). The program should: 1. Insert elements into the BST. 2. Search for a specific element in the tree. 3. Display whether the element is found or not.

#include <stdio.h>

#include <stdlib.h>

// Definition of the node structure

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

// Function to create a new node

struct Node\* createNode(int data) {

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

newNode->data = data;

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

return newNode;

}

// Function to insert a new node into the BST

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

if (root == NULL) {

return createNode(data);

}

if (data < root->data) {

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

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

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

}

return root;

}

// Function to search for an element in the BST

int search(struct Node\* root, int key) {

if (root == NULL) {

return 0; // Element not found

}

if (key == root->data) {

return 1; // Element found

}

if (key < root->data) {

return search(root->left, key); // Search in the left subtree

} else {

return search(root->right, key); // Search in the right subtree

}

}

// In-order traversal (to display the BST)

void inorderTraversal(struct Node\* root) {

if (root != NULL) {

inorderTraversal(root->left);

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

inorderTraversal(root->right);

}

}

int main() {

struct Node\* root = NULL;

int n, value, key;

// Accept the number of elements to insert into the BST

printf("Enter the number of elements to insert into the BST: ");

scanf("%d", &n);

// Accept the elements and insert them into the BST

printf("Enter the elements:\n");

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

scanf("%d", &value);

root = insert(root, value);

}

// Display the BST using in-order traversal

printf("\nIn-order Traversal of BST: ");

inorderTraversal(root);

printf("\n");

// Search for a specific element

printf("Enter the element to search for: ");

scanf("%d", &key);

// Perform search and display result

if (search(root, key)) {

printf("Element %d is found in the BST.\n", key);

} else {

printf("Element %d is not found in the BST.\n", key);

}

return 0;

}

Exp 15

Q1 Graph Representation (Adjacency Matrix): Write a C program to represent a graph using an adjacency matrix. Your program should: 1. Accept the number of vertices. 2. Accept edges and weights. 3. Display the adjacency matrix.

#include <stdio.h>

#include <stdlib.h>

#define INF 99999 // Represents no edge (infinite weight)

// Function to display the adjacency matrix

void displayMatrix(int \*\*matrix, int vertices) {

printf("\nAdjacency Matrix:\n");

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

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

if (matrix[i][j] == INF) {

printf("INF ");

} else {

printf("%3d ", matrix[i][j]);

}

}

printf("\n");

}

}

int main() {

int vertices, edges, u, v, weight;

// Accept the number of vertices

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

scanf("%d", &vertices);

// Dynamically allocate memory for the adjacency matrix

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

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

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

}

// Initialize the adjacency matrix with INF (no edge) and 0 (self-loops)

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

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

if (i == j) {

adjMatrix[i][j] = 0; // No cost for self-loop

} else {

adjMatrix[i][j] = INF; // No edge

}

}

}

// Accept the number of edges

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

scanf("%d", &edges);

// Accept edges and their weights

printf("Enter edges and weights (u v weight):\n");

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

scanf("%d %d %d", &u, &v, &weight);

adjMatrix[u][v] = weight;

adjMatrix[v][u] = weight; // Comment this line if the graph is directed

}

// Display the adjacency matrix

displayMatrix(adjMatrix, vertices);

// Free dynamically allocated memory

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

free(adjMatrix[i]);

}

free(adjMatrix);

return 0;

}

Q2 Graph Representation (Adjacency List): Write a C program to represent a graph using an adjacency list. The program should: 1. Accept the number of vertices. 2. Accept edges. 3. Display the adjacency list representation of the graph.

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in the adjacency list

typedef struct Node {

int vertex;

struct Node\* next;

} Node;

// Structure to represent the graph

typedef struct Graph {

int numVertices;

Node\*\* adjLists; // Array of pointers to Node

} Graph;

// Function to create a new node

Node\* createNode(int vertex) {

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

newNode->vertex = vertex;

newNode->next = NULL;

return newNode;

}

// Function to create a graph

Graph\* createGraph(int vertices) {

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

graph->numVertices = vertices;

// Allocate memory for adjacency list array

graph->adjLists = (Node\*\*)malloc(vertices \* sizeof(Node\*));

// Initialize all adjacency lists as empty

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

graph->adjLists[i] = NULL;

}

return graph;

}

// Function to add an edge to the graph

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

// Add edge from src to dest

Node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

// Add edge from dest to src (comment this line for directed graphs)

newNode = createNode(src);

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

graph->adjLists[dest] = newNode;

}

// Function to display the graph

void displayGraph(Graph\* graph) {

printf("\nAdjacency List Representation:\n");

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

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

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

while (temp) {

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

temp = temp->next;

}

printf("\n");

}

}

// Main function

int main() {

int vertices, edges, src, dest;

// Accept the number of vertices

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

scanf("%d", &vertices);

// Create the graph

Graph\* graph = createGraph(vertices);

// Accept the number of edges

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

scanf("%d", &edges);

// Accept edges

printf("Enter the edges (source destination):\n");

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

scanf("%d %d", &src, &dest);

addEdge(graph, src, dest);

}

// Display the adjacency list

displayGraph(graph);

// Free memory

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

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

while (temp) {

Node\* toFree = temp;

temp = temp->next;

free(toFree);

}

}

free(graph->adjLists);

free(graph);

return 0;

}

Q3 Graph Traversal (BFS): Write a C program to perform Breadth-First Search (BFS) on a graph. Your program should: 1. Accept a graph using adjacency list. 2. Perform BFS starting from a given vertex. 3. Display the traversal sequence.

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in the adjacency list

typedef struct Node {

int vertex;

struct Node\* next;

} Node;

// Structure to represent the graph

typedef struct Graph {

int numVertices;

Node\*\* adjLists;

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

} Graph;

// Queue structure for BFS

typedef struct Queue {

int\* items;

int front, rear, size;

int capacity;

} Queue;

// Function to create a new node

Node\* createNode(int vertex) {

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

newNode->vertex = vertex;

newNode->next = NULL;

return newNode;

}

// Function to create a graph

Graph\* createGraph(int vertices) {

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

graph->numVertices = vertices;

graph->adjLists = (Node\*\*)malloc(vertices \* sizeof(Node\*));

graph->visited = (int\*)calloc(vertices, sizeof(int));

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

graph->adjLists[i] = NULL;

}

return graph;

}

// Function to add an edge to the graph

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

// Add edge from src to dest

Node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

// Add edge from dest to src (comment this line for directed graphs)

newNode = createNode(src);

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

graph->adjLists[dest] = newNode;

}

// Function to create a queue

Queue\* createQueue(int capacity) {

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

queue->capacity = capacity;

queue->size = 0;

queue->front = 0;

queue->rear = -1;

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

return queue;

}

// Function to check if the queue is empty

int isEmpty(Queue\* queue) {

return queue->size == 0;

}

// Function to enqueue an element

void enqueue(Queue\* queue, int value) {

if (queue->size == queue->capacity) {

printf("Queue is full\n");

return;

}

queue->rear = (queue->rear + 1) % queue->capacity;

queue->items[queue->rear] = value;

queue->size++;

}

// Function to dequeue an element

int dequeue(Queue\* queue) {

if (isEmpty(queue)) {

printf("Queue is empty\n");

return -1;

}

int value = queue->items[queue->front];

queue->front = (queue->front + 1) % queue->capacity;

queue->size--;

return value;

}

// BFS function

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

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

graph->visited[startVertex] = 1;

enqueue(queue, startVertex);

printf("BFS Traversal: ");

while (!isEmpty(queue)) {

int currentVertex = dequeue(queue);

printf("%d ", currentVertex);

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

while (temp) {

int adjVertex = temp->vertex;

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

graph->visited[adjVertex] = 1;

enqueue(queue, adjVertex);

}

temp = temp->next;

}

}

printf("\n");

// Free the queue memory

free(queue->items);

free(queue);

}

// Main function

int main() {

int vertices, edges, src, dest, startVertex;

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

scanf("%d", &vertices);

Graph\* graph = createGraph(vertices);

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

scanf("%d", &edges);

printf("Enter the edges (source destination):\n");

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

scanf("%d %d", &src, &dest);

addEdge(graph, src, dest);

}

printf("Enter the starting vertex for BFS: ");

scanf("%d", &startVertex);

BFS(graph, startVertex);

// Free memory

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

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

while (temp) {

Node\* toFree = temp;

temp = temp->next;

free(toFree);

}

}

free(graph->adjLists);

free(graph->visited);

free(graph);

return 0;

}

Q4 Graph Traversal (DFS): Write a C program to perform Depth-First Search (DFS) on a graph. Your program should: 1. Accept a graph using adjacency list. 2. Perform DFS starting from a given vertex. 3. Display the traversal sequence.

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in the adjacency list

typedef struct Node {

int vertex;

struct Node\* next;

} Node;

// Structure to represent the graph

typedef struct Graph {

int numVertices;

Node\*\* adjLists;

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

} Graph;

// Function to create a new node

Node\* createNode(int vertex) {

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

newNode->vertex = vertex;

newNode->next = NULL;

return newNode;

}

// Function to create a graph

Graph\* createGraph(int vertices) {

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

graph->numVertices = vertices;

graph->adjLists = (Node\*\*)malloc(vertices \* sizeof(Node\*));

graph->visited = (int\*)calloc(vertices, sizeof(int));

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

graph->adjLists[i] = NULL;

}

return graph;

}

// Function to add an edge to the graph

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

// Add edge from src to dest

Node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

// Add edge from dest to src (comment this line for directed graphs)

newNode = createNode(src);

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

graph->adjLists[dest] = newNode;

}

// DFS function

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

graph->visited[vertex] = 1;

printf("%d ", vertex);

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

while (temp) {

int adjVertex = temp->vertex;

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

DFS(graph, adjVertex);

}

temp = temp->next;

}

}

// Main function

int main() {

int vertices, edges, src, dest, startVertex;

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

scanf("%d", &vertices);

Graph\* graph = createGraph(vertices);

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

scanf("%d", &edges);

printf("Enter the edges (source destination):\n");

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

scanf("%d %d", &src, &dest);

addEdge(graph, src, dest);

}

printf("Enter the starting vertex for DFS: ");

scanf("%d", &startVertex);

printf("DFS Traversal: ");

DFS(graph, startVertex);

printf("\n");

// Free memory

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

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

while (temp) {

Node\* toFree = temp;

temp = temp->next;

free(toFree);

}

}

free(graph->adjLists);

free(graph->visited);

free(graph);

return 0;

}