1. **Array Operation (Insertion, Deletion, Sorting, Merging)**

**Program:**

#include <stdio.h>

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

printf("Array: ");

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

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

}

printf("\n");

}

void insert(int arr[], int \*n, int element, int position) {

if (\*n >= 100) {

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

return;

}

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

printf("Invalid position for insertion.\n");

return;

}

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

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

}

arr[position] = element;

(\*n)++;

}

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

if (\*n <= 0) {

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

return;

}

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

printf("Invalid position for deletion.\n");

return;

}

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

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

}

(\*n)--;

}

void merge(int arr1[], int n1, int arr2[], int n2, int result[]) {

int i = 0, j = 0, k = 0;

while (i < n1 && j < n2) {

if (arr1[i] < arr2[j]) {

result[k++] = arr1[i++];

} else {

result[k++] = arr2[j++];

}

}

while (i < n1) {

result[k++] = arr1[i++];

}

while (j < n2) {

result[k++] = arr2[j++];

}

}

int main() {

int arr1[100], arr2[100], merged[200];

int n1, n2;

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

scanf("%d", &n1);

printf("Enter elements for the first array:\n");

for (int i = 0; i < n1; i++) {

scanf("%d", &arr1[i]);

}

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

scanf("%d", &n2);

printf("Enter elements for the second array:\n");

for (int i = 0; i < n2; i++) {

scanf("%d", &arr2[i]);

}

merge(arr1, n1, arr2, n2, merged);

int mergedSize = n1 + n2;

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

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

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

int temp = merged[j];

merged[j] = merged[j + 1];

merged[j + 1] = temp;

}

}

}

printf("Merged and Sorted Array:\n");

display(merged, mergedSize);

int choice;

int element, position;

while (1) {

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

printf("1. Insertion\n");

printf("2. Deletion\n");

printf("3. Display Merged 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 for insertion: ");

scanf("%d", &position);

insert(merged, &mergedSize, element, position);

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

break;

case 2:

printf("Enter the position for deletion: ");

scanf("%d", &position);

deleteElement(merged, &mergedSize, position);

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

break;

case 3:

display(merged, mergedSize);

break;

case 4:

return 0;

default:

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

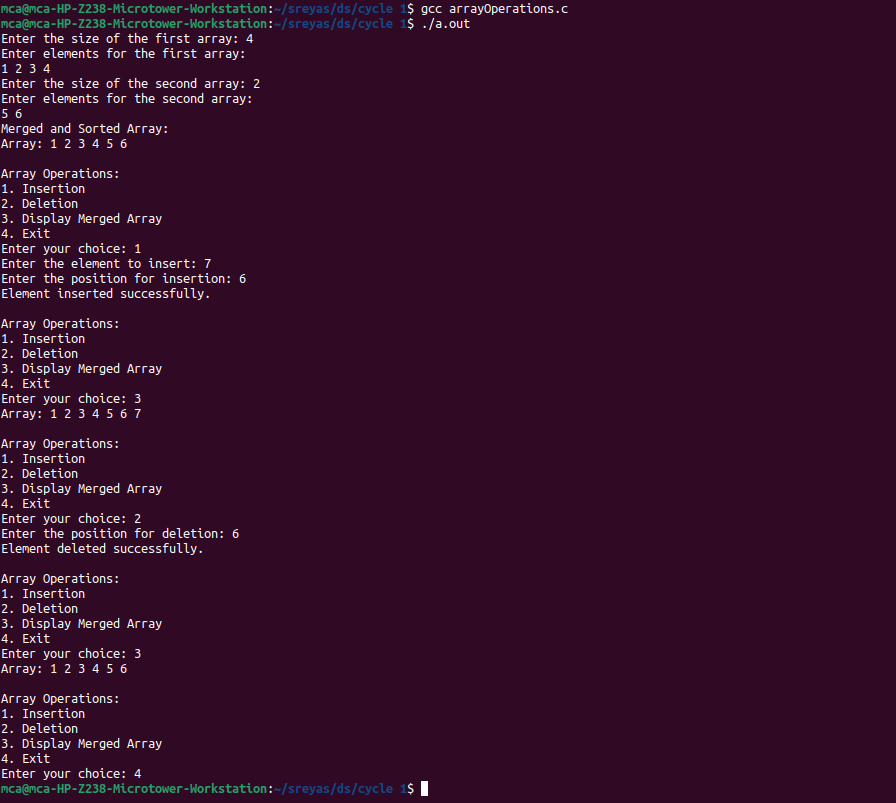
}

}

return 0;

}

**Output:**

****

1. **Searching an array element ( Linear Search, Binary Search)**

**Program:**

#include <stdio.h>

// This function is used for getting the search key

int getKey(){

int key;

printf("enter the key to search : ");

scanf("%d", &key);

return key;

}

// function to perform linear search

void linearSearch(int n) {

int arr[50], flag=0;

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

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

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

}

int key = getKey();

int i=0;

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

if (arr[i] == key) {

flag = 1;

break;

}

}

if(flag==1){

printf("element found at location %d.\n", i+1);

} else {

printf("element not found.\n");

}

}

// function to perform binary search

void binarySearch(int n) {

int left = 0, arr[50];

int right = n - 1;

printf("Enter elements of the array in ascending order:\n");

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

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

}

int key = getKey();

int flag = 0;

int mid = 0;

while (left <= right) {

mid = left + (right - left) / 2;

if (arr[mid] == key) {

flag = 1;

break;

} else if (arr[mid] < key) {

left = mid + 1;

} else {

right = mid - 1;

}

}

if(flag==1){

printf("element found at location %d.\n", mid+1);

} else {

printf("element not found.\n");

}

}

int main() {

int arr[100];

int n;

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

scanf("%d", &n);

int opt;

while(1){

printf("1.binary search\n2.linear search\n3.exit\nEnter the operation : ");

scanf("%d", &opt);

switch(opt){

case 1: binarySearch(n);

break;

case 2: linearSearch(n);

break;

case 3: return 1;

default: printf("invalid input.\n");

}

}

return 1;

}

**Output:**

****

1. **Matrix Operations ( Addition, Multiplication, Transpose)**

**Program:**

#include <stdio.h>

void displayMatrix(int mat[][100], int rows, int cols) {

printf("Matrix:\n");

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

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

printf("%d\t", mat[i][j]);

}

printf("\n");

}

}

void addMatrices(int mat1[][100], int mat2[][100], int result[][100], int rows, int cols) {

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

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

result[i][j] = mat1[i][j] + mat2[i][j];

}

}

}

void multiplyMatrices(int mat1[][100], int rows1, int cols1, int mat2[][100], int cols2, int result[][100]) {

for (int i = 0; i < rows1; i++) {

for (int j = 0; j < cols2; j++) {

result[i][j] = 0;

for (int k = 0; k < cols1; k++) {

result[i][j] += mat1[i][k] \* mat2[k][j];

}

}

}

}

void transposeMatrix(int mat[][100], int rows, int cols, int result[][100]) {

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

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

result[j][i] = mat[i][j];

}

}

}

void main() {

int mat1[100][100], mat2[100][100], result[100][100];

int rows1, cols1, rows2, cols2;

printf("Enter the number of rows for the first matrix: ");

scanf("%d", &rows1);

printf("Enter the number of columns for the first matrix: ");

scanf("%d", &cols1);

printf("Enter elements for the first matrix:\n");

for (int i = 0; i < rows1; i++) {

for (int j = 0; j < cols1; j++) {

scanf("%d", &mat1[i][j]);

}

}

printf("Enter the number of rows for the second matrix: ");

scanf("%d", &rows2);

printf("Enter the number of columns for the second matrix: ");

scanf("%d", &cols2);

printf("Enter elements for the second matrix:\n");

for (int i = 0; i < rows2; i++) {

for (int j = 0; j < cols2; j++) {

scanf("%d", &mat2[i][j]);

}

}

if (rows1 == rows2 && cols1 == cols2) {

addMatrices(mat1, mat2, result, rows1, cols1);

printf("Matrix Addition Result:\n");

displayMatrix(result, rows1, cols1);

} else {

printf("Matrix addition is not possible. Matrices must have the same dimensions for addition.\n");

}

if (cols1 != rows2) {

printf("Matrix multiplication is not possible. Number of columns in the first matrix must be equal to the number of rows in the second matrix.\n");

} else {

multiplyMatrices(mat1, rows1, cols1, mat2, cols2, result);

printf("Matrix Multiplication Result:\n");

displayMatrix(result, rows1, cols2);

}

transposeMatrix(mat1, rows1, cols1, result);

printf("Transpose of the First Matrix:\n");

displayMatrix(result, cols1, rows1);

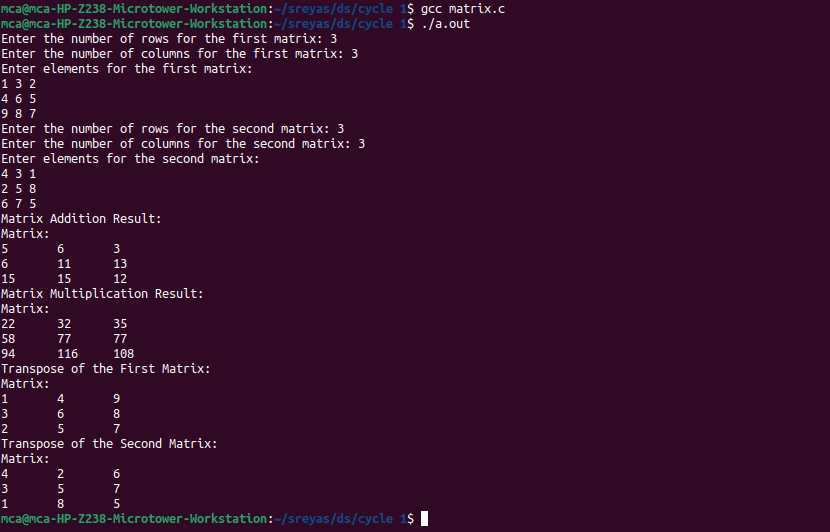
transposeMatrix(mat2, rows2, cols2, result);

printf("Transpose of the Second Matrix:\n");

displayMatrix(result, cols2, rows2);

}

**Output:**

****

1. **Using Structure, add two distances in the inch-feet system.**

**Program:**

#include <stdio.h>

struct Distance {

int feet;

int inches;

}d1, d2, result;

void addDistances() {

result.inches = d1.inches + d2.inches;

result.feet = result.inches >= 12 ? d1.feet + d2.feet + result.inches/12 : d1.feet + d2.feet;

result.inches %= 12;

}

void main() {

printf("Enter the first distance:\n");

printf("Feet: ");

scanf("%d", &d1.feet);

printf("Inches: ");

scanf("%d", &d1.inches);

printf("Enter the second distance:\n");

printf("Feet: ");

scanf("%d", &d2.feet);

printf("Inches: ");

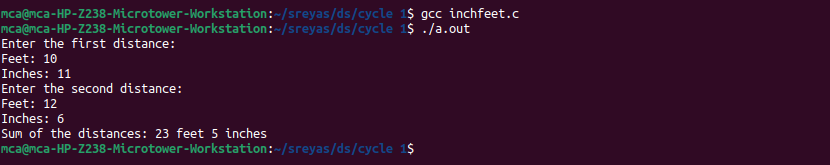
scanf("%d", &d2.inches);

addDistances();

printf("Sum of the distances: %d feet %d inches\n", result.feet, result.inches);

}

**Output:**

****

1. **Implement Stack Operations**

**Program:**

#include <stdio.h>

#include <stdlib.h>

int top=-1, stack[100], maxsize;

void caseCheck();

void push(){

int element;

printf("enter the element to read : ");

scanf("%d", &element);

if(top+1!=maxsize){

top++;

stack[top] = element;

} else {

printf("stack overflow\n");

}

caseCheck();

}

void pop(){

if(top == -1){

printf("stack is empty\n");

} else {

top--;

}

caseCheck();

}

void display(){

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

printf("%d \t", stack[i]);

}

caseCheck();

}

void peek(){

if(top == -1){

printf("stack is empty\n");

} else {

printf("last element is : %d\n", stack[top]);

}

caseCheck();

}

void isFull(){

if(top == maxsize-1){

printf("stack is full\n");

} else {

printf("stack is not full\n");

}

caseCheck();

}

void caseCheck(){

int option;

printf("\n1.push\n2.pop\n3.peek\n4.check if the stack is full\n5.display\n6.exit\nEnter the operation : ");

scanf("%d", &option);

switch(option){

case 1 : push();

break;

case 2 : pop();

break;

case 3 : peek();

break;

case 4 : isFull();

break;

case 5 : display();

break;

case 6 : exit(0);

break;

default : printf("enter a valid input");

}

}

void main(){

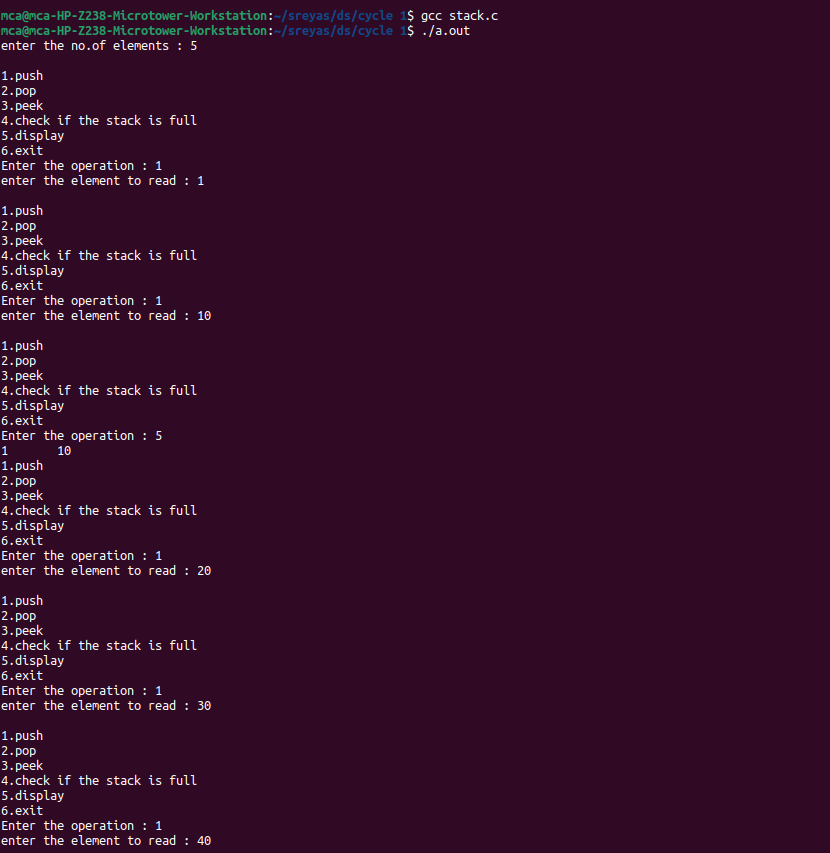
printf("enter the no.of elements : ");

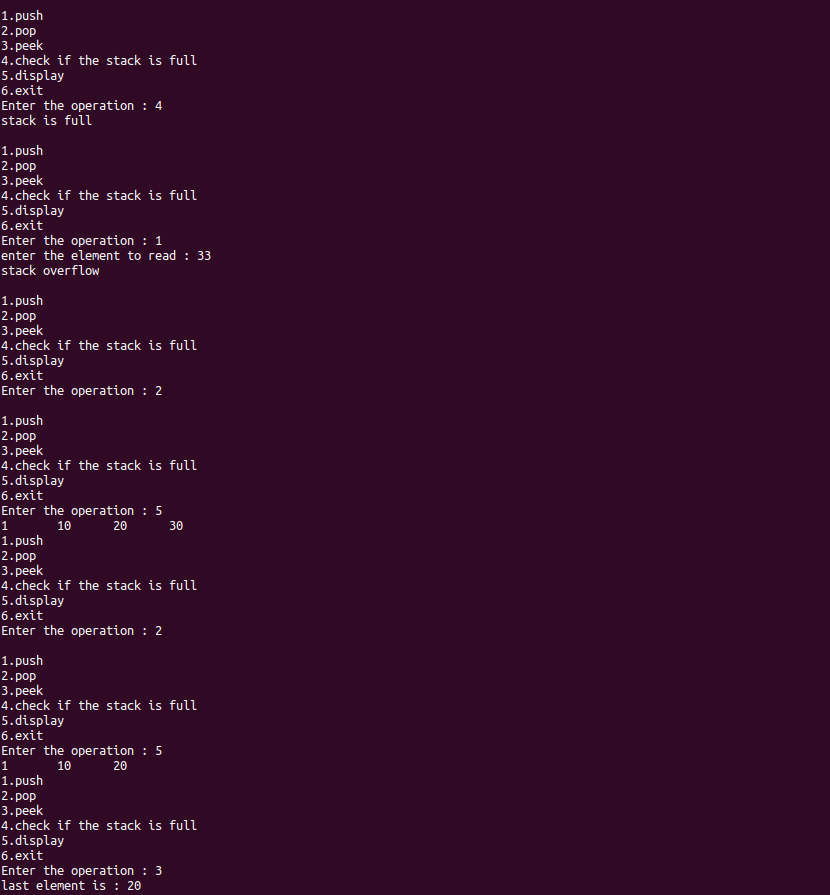
scanf("%d", &maxsize);

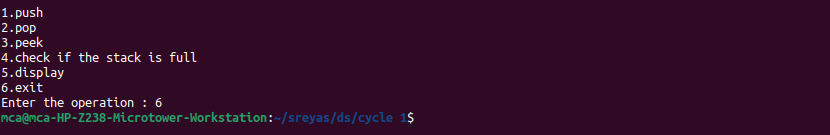
caseCheck();

}

**Output:**

****

****

****

1. **String Operations ( Searching, Concatenation, Substring)**

**Program:**

#include <stdio.h>

#include <string.h>

int searchSubstring(char \*str, char \*subStr) {

char \*p = strstr(str, subStr);

return (p) ? p - str : -1;

}

void concatenateStrings(char \*str1, char \*str2, char \*result) {

strcpy(result, str1);

strcat(result, str2);

}

void extractSubstring(char \*str, int start, int length, char \*result) {

strncpy(result, str + start, length);

result[length] = '\0';

}

int main() {

char str1[100], str2[100], subStr[100], result[200];

int start, length;

printf("Enter the first string: ");

scanf("%s", str1);

printf("Enter the second string: ");

scanf("%s", str2);

printf("Enter the substring to check : ");

scanf("%s", subStr);

concatenateStrings(str1, str2, result);

printf("Concatenated string: %s\n", result);

int index = searchSubstring(result, subStr);

if (index != -1) {

printf("Substring found at position %d in the string.\n", index+1);

} else {

printf("Substring not found in the first string.\n");

}

concatenateStrings(str1, str2, result);

printf("Concatenated string: %s\n", result);

printf("Enter the starting index for substring extraction: ");

scanf("%d", &start);

printf("Enter the length of the substring to extract: ");

scanf("%d", &length);

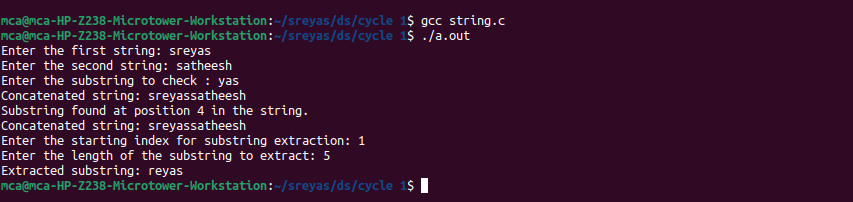
extractSubstring(str1, start, length, result);

printf("Extracted substring: %s\n", result);

return 0;

}

**Output:**

****

1. **Sorting an Array ( Bubble Sort, Selection Sort, Insertion Sort )**

**Program:**

#include <stdio.h>

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

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

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

}

printf("\n");

}

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

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

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

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

int temp = arr[j];

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

arr[j + 1] = temp;

}

}

}

}

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

int min, temp;

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

min = i;

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

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

min = j;

}

}

temp = arr[i];

arr[i] = arr[min];

arr[min] = temp;

}

}

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

int key, j;

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

key = arr[i];

j = i - 1;

while (j >= 0 && arr[j] > key) {

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

j = j - 1;

}

arr[j + 1] = key;

}

}

void main() {

int arr[100], size;

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

scanf("%d", &size);

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

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

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

}

int bubbleSortedArr[100];

for (int i = 0; i < size; i++) bubbleSortedArr[i] = arr[i];

bubbleSort(bubbleSortedArr, size);

printf("Bubble Sort Result: ");

displayArray(bubbleSortedArr, size);

int selectionSortedArr[100];

for (int i = 0; i < size; i++) selectionSortedArr[i] = arr[i];

selectionSort(selectionSortedArr, size);

printf("Selection Sort Result: ");

displayArray(selectionSortedArr, size);

int insertionSortedArr[100];

for (int i = 0; i < size; i++) insertionSortedArr[i] = arr[i];

insertionSort(insertionSortedArr, size);

printf("Insertion Sort Result: ");

displayArray(insertionSortedArr, size);

}

**Output:**

****

1. **Implement Queue operations (Insert, delete, display front & rear values)**

**Program:**

#include <stdio.h>

int front = -1;

int rear = -1;

int max = 10;

int q[10];

void enqueue();

void dequeue();

void displayRear();

void displayFront();

void display();

int main(){

int c;

while(1){

printf("1.enqueue\n2.dequeue\n3.display rear\n4.display front\n5.display\n6.Exit\nEnter the operation : ");

scanf("%d", &c);

switch(c){

case 1: enqueue();

break;

case 2: dequeue();

break;

case 3: displayRear();

break;

case 4: displayFront();

break;

case 5: display();

break;

case 6: return 0;

default: printf("invalid option! Please enter a valid option.\n");

}

}

}

void enqueue(){

int data;

if(rear+1 >= max){

printf("queue is full.\n");

} else {

printf("enter the data to insert : ");

scanf("%d", &data);

rear++;

q[rear] = data;

printf("data inserted to position %d\n", rear+1);

}

}

void dequeue(){

if(rear <= front){

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

} else {

front++;

printf("data successfully deleted from the position %d\n", front+1);

}

}

void display(){

printf("\narray elements are : ");

for(int i=front+1; i<=rear; i++){

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

}

printf("\n");

}

void displayFront(){

printf("first element is : %d\n", q[front+1]);

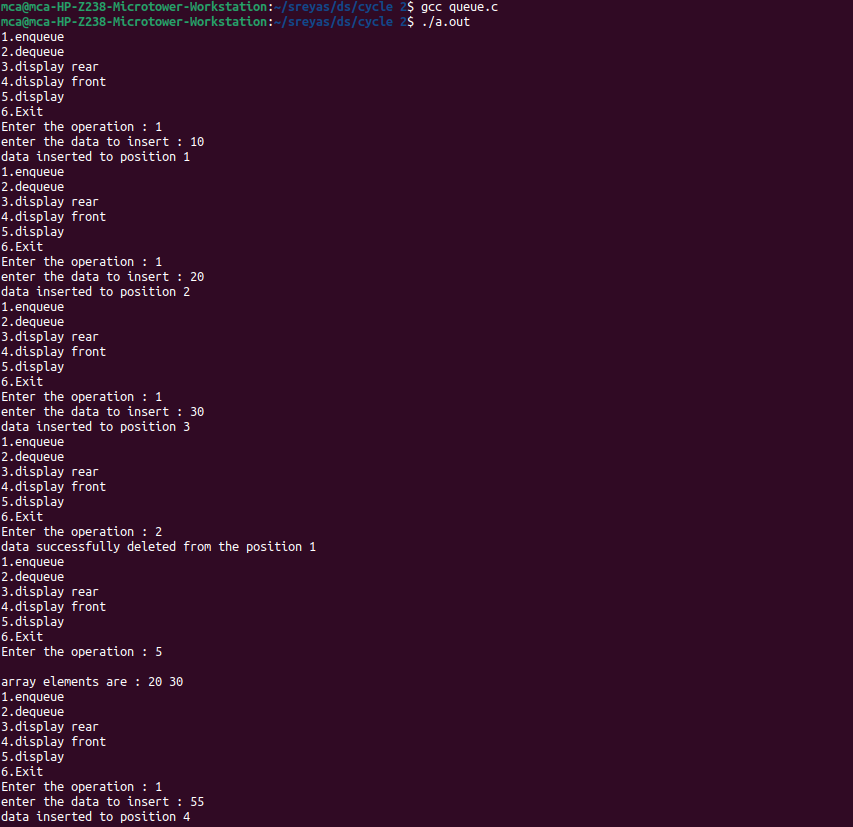
}

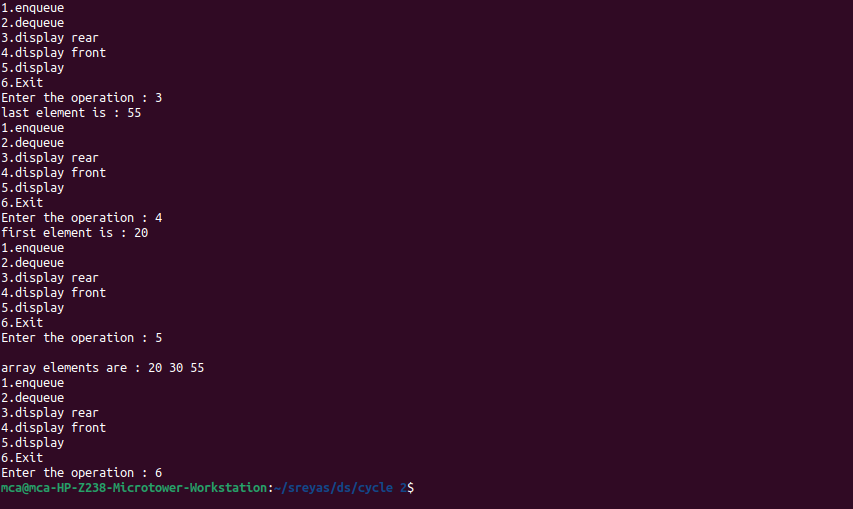
void displayRear(){

printf("last element is : %d\n", q[rear]);

}

**Output:**

****

****

1. **Implement Circular Queue operations (Insert, delete, display front & rear values)**

**Program:**

#include<stdio.h>

# define MAX 5

int Q[MAX];

int front = -1;

int rear = -1;

void insert(int item)

{

if((front == 0 && rear == MAX-1) || (front == rear+1)){

printf("Queue Overflow \n");

return;

}

if(front == -1){

front = 0;

rear = 0;

}

else{

if(rear == MAX-1) rear = 0;

else rear = rear+1;

}

Q[rear] = item;

}

void deletion(){

if(front == -1){

printf("Queue Underflown\n");

return ;

}

printf("Element deleted from queue is : %d\n",Q[front]);

if(front == rear){

front = -1;

rear=-1;

}

else {

if(front == MAX-1) front = 0;

else front = front+1;

}

}

void display(){

int front\_pos = front,rear\_pos = rear;

if(front == -1)

{

printf("Queue is empty\n");

return;

}

printf("Queue elements : ");

if( front\_pos <= rear\_pos )

while(front\_pos <= rear\_pos) {

printf("%d ",Q[front\_pos]);

front\_pos++;

}

else {

while(front\_pos <= MAX-1){

printf("%d ",Q[front\_pos]);

front\_pos++;

}

front\_pos = 0;

while(front\_pos <= rear\_pos){

printf("%d ",Q[front\_pos]);

front\_pos++;

}

}

printf("\n");

}

int main() {

int choice,item;

do {

printf("1.Insert\n");

printf("2.Delete\n");

printf("3.Display\n");

printf("4.Quit\n");

printf("Enter your choice : ");

scanf("%d",&choice);

switch(choice){

case 1 :

printf("Input the element for insertion in queue : ");

scanf("%d", &item);

insert(item);

break;

case 2 :

deletion();

break;

case 3:

display();

break;

case 4:

break;

default:

printf("Wrong choice\n");

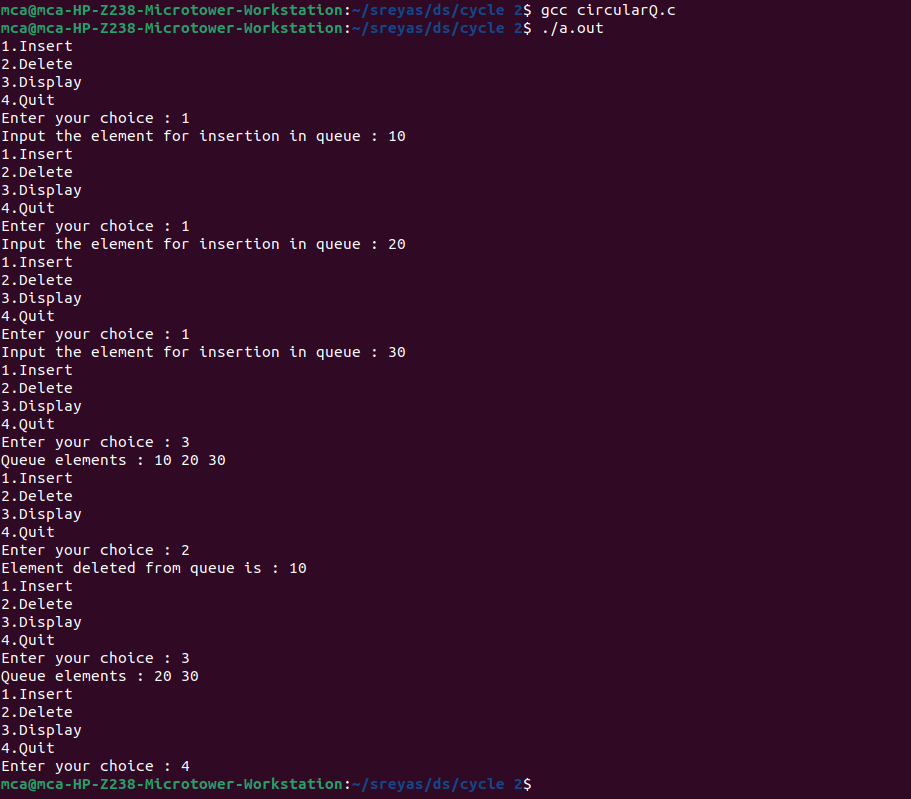
}

}while(choice!=4);

return 0;

}

**Output:**

****

1. **Implement singly linked list (Insert at the head, insert at tail, insert at a position, delete at the head, delete at tail, delete form a position, search an element).**

**Program:**

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

typedef struct node{

int data;

struct node \*link;

} node;

node \* head = NULL;

node \*createNewNode(){

int data;

printf("enter the data to be inserted : ");

scanf("%d", &data);

node \* newnode = (node \*)malloc((sizeof(node \*)));

newnode -> data = data;

newnode -> link = NULL;

return newnode;

}

void insertAtBeginning(){

node \*newnode = createNewNode();

if(head==NULL) head = newnode;

else {

(newnode)->link = head;

head = newnode;

}

}

void insertAtEnd(){

if(head==NULL) insertAtBeginning();

else {

node \*newnode = createNewNode();

node \* temp = head;

while(temp->link !=NULL) temp = temp -> link;

temp->link = newnode;

}

}

void insertAtPosition(){

int pos;

printf("enter the position to insert : ");

scanf("%d", &pos);

if(pos<=1 || head==NULL) insertAtBeginning();

else {

node \* newnode = createNewNode();

node \* temp = head;

int count = 2;

while(temp->link != NULL && count!=pos){

temp = temp->link;

count++;

}

newnode->link = temp->link;

temp->link = newnode;

}

}

void deleteAtBeginning(){

if(head==NULL) printf("linked list is empty.\n");

else head = head->link;

}

void deleteAtEnd(){

if(head==NULL) printf("linked list is empty.\n");

else {

node \* temp = head;

node \* dup = temp;

if(temp->link == NULL) head = NULL;

else {

while(temp->link!=NULL) {

dup = temp;

temp = temp-> link;

} dup-> link = NULL;

}

}

}

void deleteAtPosition(){

int pos;

if(head==NULL) printf("linked list is empty\n");

else {

printf("enter the position to insert : ");

scanf("%d", &pos);

if(pos<=1) deleteAtBeginning();

else {

node \* temp = head;

node \* dup = temp;

int count = 2;

while(temp->link != NULL && count!=pos){

dup = temp;

temp = temp->link;

count++;

}

if(temp->link == NULL) dup->link = NULL;

else dup->link = dup->link->link;

}

}

}

void display(){

if(head == NULL) printf("linked list is empty.\n");

else {

node \*temp = head;

while(temp != NULL){

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

temp = temp->link;

} printf("\n");

}

}

void search(){

int key, flag=0;

printf("enter the element to search : ");

scanf("%d", &key);

node \* temp = head;

while(temp != NULL){

if(temp->data == key){

printf("element found.\n");

return;

} flag = 1;

temp=temp->link;

}

printf("element not found.\n");

}

int main(){

int choice;

int data;

while(true){

printf("\n1.insert at beginning\n2.insert at end\n3.insert at position\n4.delete at beginning\n5.delete at end\n6.delete at position\n7.display\n8.search\n9.exit\nEnter the operation you want : ");

scanf("%d", &choice);

switch (choice){

case 1: insertAtBeginning();break;

case 2: insertAtEnd();break;

case 3: insertAtPosition();break;

case 4: deleteAtBeginning();break;

case 5: deleteAtEnd();break;

case 6: deleteAtPosition();break;

case 7: display();break;

case 8: search();break;

case 9: return 0;

default: printf("invalid input, please enter a valid input\n\n");

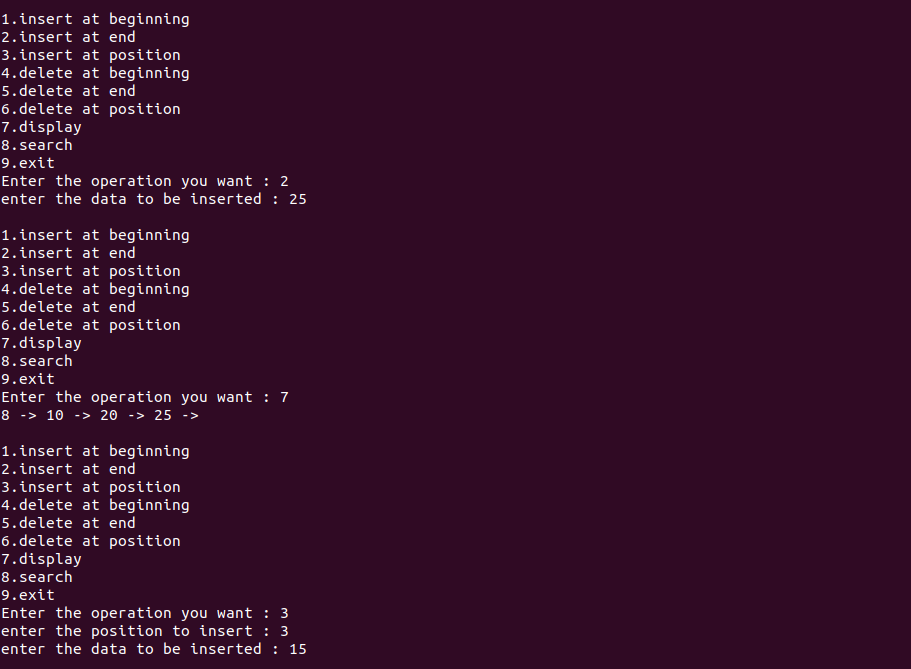
}

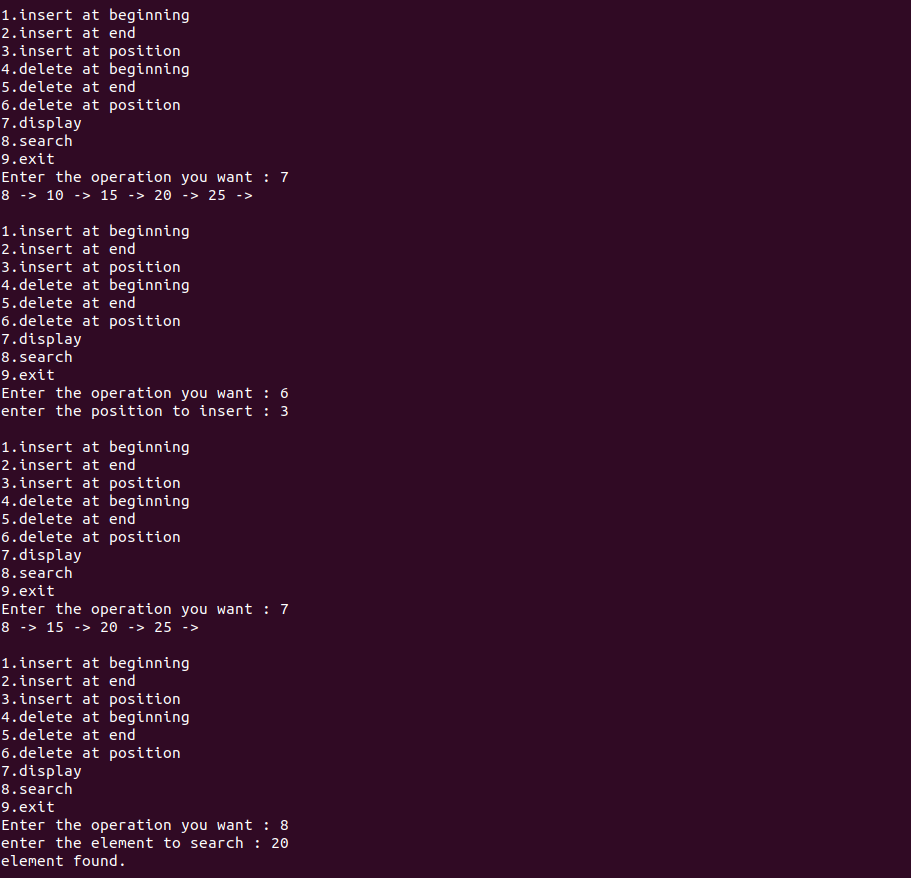
}

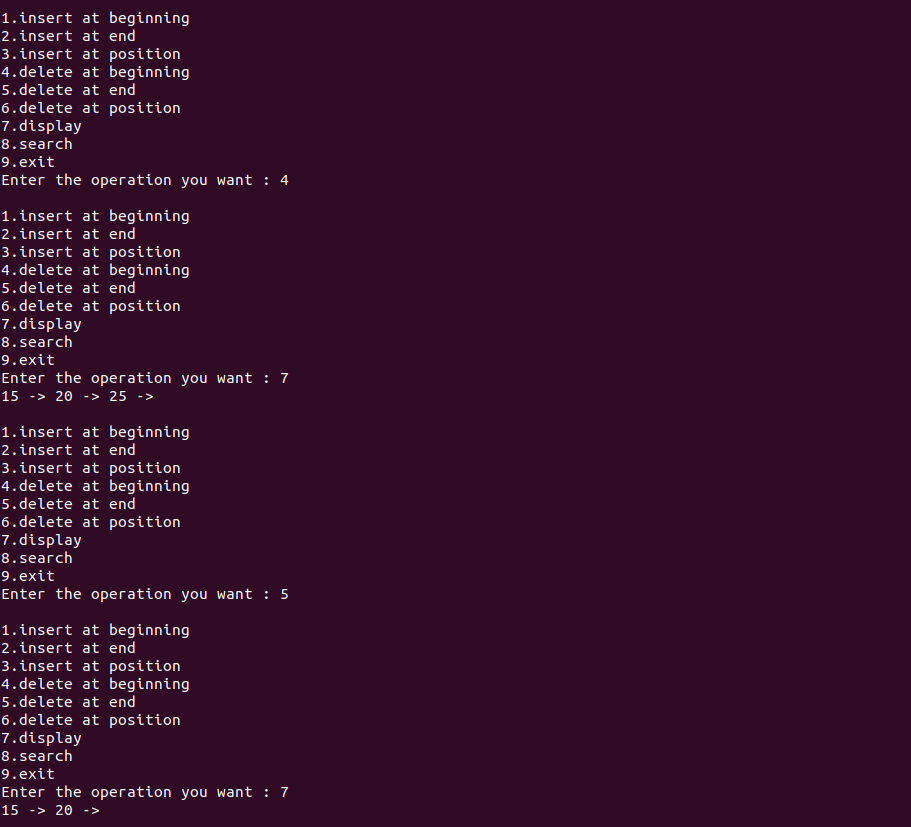
}

**Output:**

****









1. **Implement doubly linked list (Insert at the head, insert at tail, insert at a position, delete at the head, delete at tail, delete form a position, search an element).**

**Program:**

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in the doubly linked list

struct Node {

int data;

struct Node\* prev;

struct Node\* next;

};

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

struct Node\* createNode(int data) {

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

if (newNode == NULL) {

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

exit(1);

}

newNode->data = data;

newNode->prev = NULL;

newNode->next = NULL;

return newNode;

}

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

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

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

} else {

newNode->next = \*head;

(\*head)->prev = newNode;

\*head = newNode;

}

}

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

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

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

} else {

struct Node\* current = \*head;

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

newNode->prev = current;

}

}

// Function to insert a node at a specified position in the doubly linked list

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

if (position <= 0) {

printf("Invalid position for insertion.\n");

return;

}

if (position == 1) {

insertAtHead(head, data);

return;

}

struct Node\* newNode = createNode(data);

struct Node\* current = \*head;

int count = 1;

while (current != NULL && count < position - 1) {

current = current->next;

count++;

}

if (current == NULL) {

printf("Position out of bounds for insertion.\n");

free(newNode);

return;

}

newNode->next = current->next;

if (current->next != NULL) {

current->next->prev = newNode;

}

current->next = newNode;

newNode->prev = current;

}

// Function to delete a node at the head of the doubly linked list

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

if (\*head == NULL) {

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

return;

}

struct Node\* temp = \*head;

\*head = (\*head)->next;

if (\*head != NULL) {

(\*head)->prev = NULL;

}

free(temp);

}

// Function to delete a node at the tail of the doubly linked list

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

if (\*head == NULL) {

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

return;

}

struct Node\* current = \*head;

while (current->next != NULL) {

current = current->next;

}

if (current->prev != NULL) {

current->prev->next = NULL;

} else {

\*head = NULL;

}

free(current);

}

// Function to delete a node from a specified position in the doubly linked list

void deleteFromPosition(struct Node\*\* head, int position) {

if (position <= 0) {

printf("Invalid position for deletion.\n");

return;

}

if (position == 1) {

deleteAtHead(head);

return;

}

struct Node\* current = \*head;

int count = 1;

while (current != NULL && count < position) {

current = current->next;

count++;

}

if (current == NULL) {

printf("Position out of bounds for deletion.\n");

return;

}

if (current->prev != NULL) {

current->prev->next = current->next;

} else {

\*head = current->next;

}

if (current->next != NULL) {

current->next->prev = current->prev;

}

free(current);

}

// Function to search for a node with a specific data in the doubly linked list

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

struct Node\* current = head;

while (current != NULL) {

if (current->data == data) {

return current;

}

current = current->next;

}

return NULL;

}

// Function to display the elements of the doubly linked list

void displayList(struct Node\* head) {

printf("Doubly Linked List: ");

while (head != NULL) {

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

head = head->next;

}

printf("NULL\n");

}

int main() {

struct Node\* head = NULL;

int choice, data, position;

struct Node\* result;

do {

printf("Menu:\n");

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

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

printf("3. Insert at a position\n");

printf("4. Delete at the head\n");

printf("5. Delete at the tail\n");

printf("6. Delete from a position\n");

printf("7. Search for an element\n");

printf("8. Display the list\n");

printf("9. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter data to insert at the head: ");

scanf("%d", &data);

insertAtHead(&head, data);

break;

case 2:

printf("Enter data to insert at the tail: ");

scanf("%d", &data);

insertAtTail(&head, data);

break;

case 3:

printf("Enter data to insert: ");

scanf("%d", &data);

printf("Enter the position: ");

scanf("%d", &position);

insertAtPosition(&head, data, position);

break;

case 4:

deleteAtHead(&head);

break;

case 5:

deleteAtTail(&head);

break;

case 6:

printf("Enter the position to delete: ");

scanf("%d", &position);

deleteFromPosition(&head, position);

break;

case 7:

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

scanf("%d", &data);

result = searchElement(head, data);

if (result != NULL) {

printf("Element %d found in the list.\n", data);

} else {

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

}

break;

case 8:

displayList(head);

break;

case 9:

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

break;

default:

printf("Invalid choice. Please select a valid option.\n");

break;

}

} while (choice != 9);

// Clean up: Free memory

while (head != NULL) {

struct Node\* temp = head;

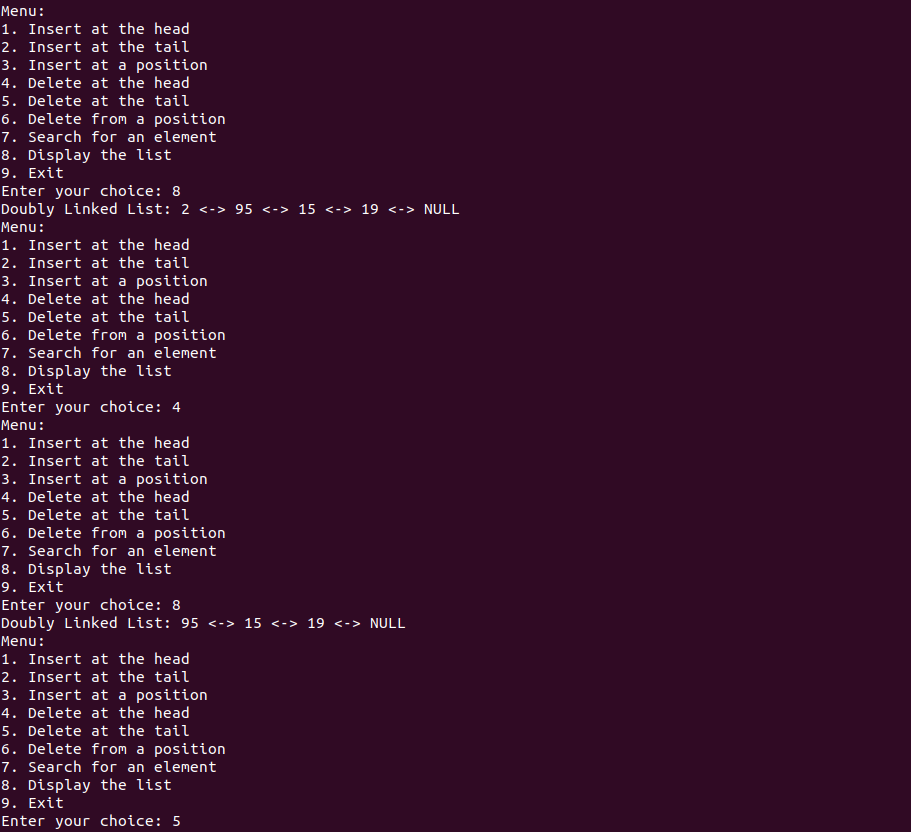
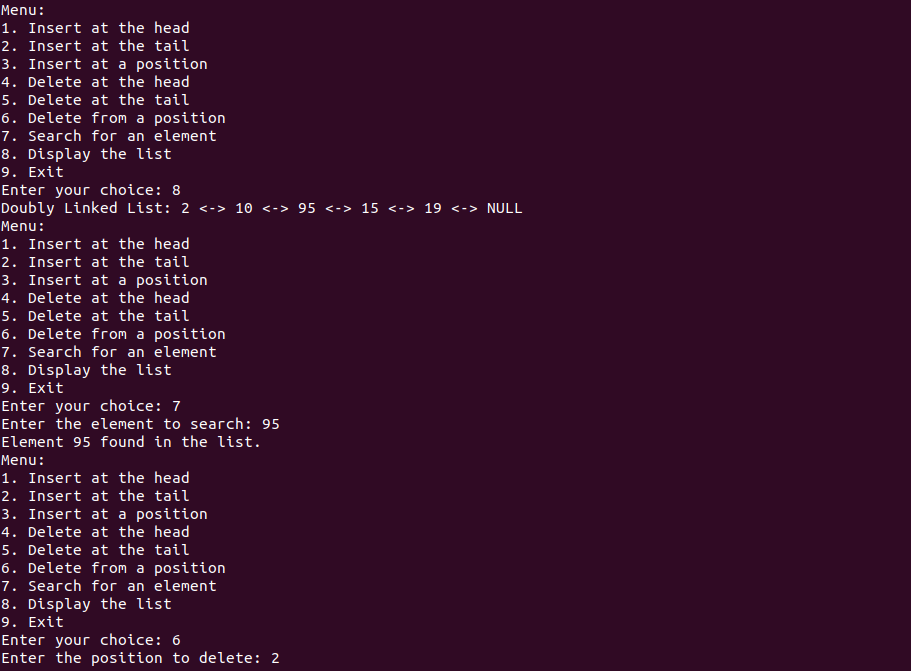
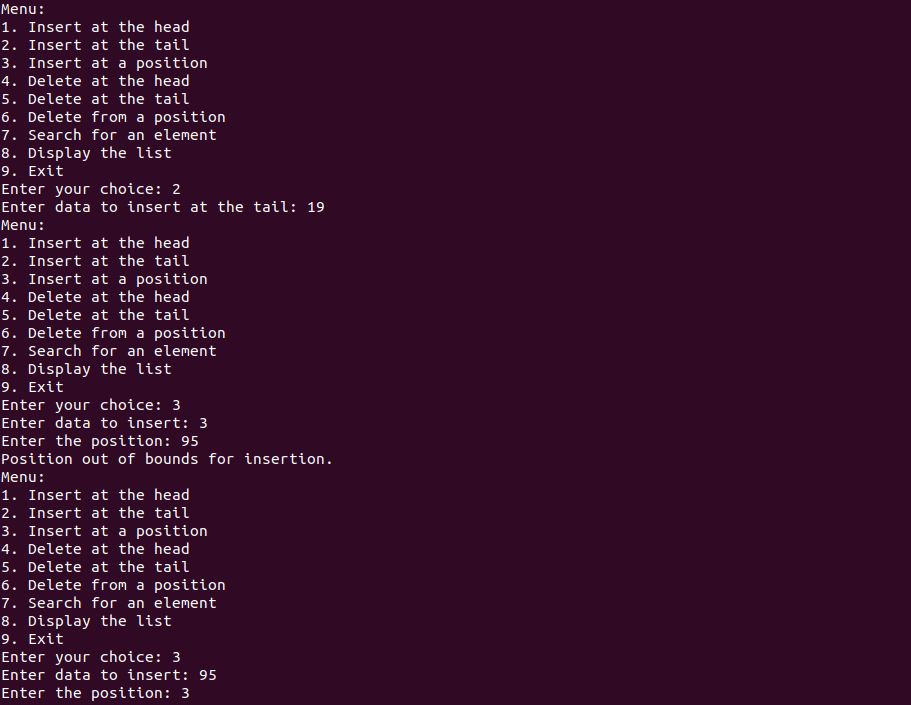
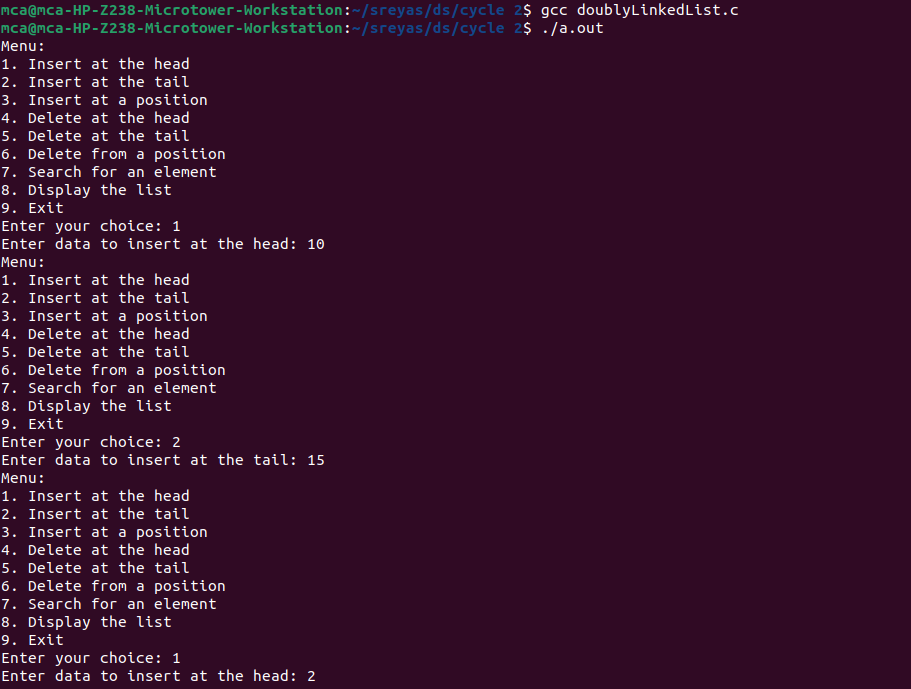
head = head->next;

free(temp);

}

return 0;

}

**Output:**

1. **Implement circular linked list (Insert at the head, insert at tail, insert at a position, delete at the head, delete at tail, delete form a position, search an element).**

**Program:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

struct Node\* createNode(int data) {

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

if (newNode == NULL) {

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

exit(1);

}

newNode->data = data;

newNode->next = newNode; // Points to itself in a circular list

return newNode;

}

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

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

} else {

struct Node\* tail = (\*head)->next;

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

tail = tail->next;

}

tail->next = newNode;

newNode->next = \*head;

\*head = newNode;

}

}

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

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

} else {

struct Node\* tail = (\*head)->next;

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

tail = tail->next;

}

tail->next = newNode;

newNode->next = \*head;

}

}

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

if (position <= 0) {

printf("Invalid position for insertion.\n");

return;

}

if (position == 1) {

insertAtHead(head, data);

return;

}

struct Node\* newNode = createNode(data);

struct Node\* current = \*head;

int count = 1;

while (current->next != \*head && count < position - 1) {

current = current->next;

count++;

}

newNode->next = current->next;

current->next = newNode;

}

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

if (\*head == NULL) {

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

return;

}

struct Node\* tail = (\*head)->next;

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

tail = tail->next;

}

if (\*head == tail) {

free(\*head);

\*head = NULL;

} else {

struct Node\* temp = \*head;

\*head = (\*head)->next;

tail->next = \*head;

free(temp);

}

}

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

if (\*head == NULL) {

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

return;

}

struct Node\* current = \*head;

struct Node\* prev = NULL;

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

prev = current;

current = current->next;

}

if (prev == NULL) {

free(current);

\*head = NULL;

} else {

prev->next = \*head;

free(current);

}

}

void deleteFromPosition(struct Node\*\* head, int position) {

if (position <= 0) {

printf("Invalid position for deletion.\n");

return;

}

if (position == 1) {

deleteAtHead(head);

return;

}

struct Node\* current = \*head;

struct Node\* prev = NULL;

int count = 1;

while (current->next != \*head && count < position) {

prev = current;

current = current->next;

count++;

}

if (current == \*head) {

printf("Position out of bounds for deletion.\n");

return;

}

prev->next = current->next;

free(current);

}

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

if (head == NULL) {

return NULL;

}

struct Node\* current = head;

do {

if (current->data == data) {

return current;

}

current = current->next;

} while (current != head);

return NULL;

}

void displayList(struct Node\* head) {

if (head == NULL) {

printf("Circular Linked List is empty.\n");

return;

}

struct Node\* current = head;

do {

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

current = current->next;

} while (current != head);

printf(" (Head)\n");

}

int main() {

struct Node\* head = NULL;

int choice, data, position;

struct Node\* result;

do {

printf("Menu:\n1. insert at head\n2. Insert at tail\n3. Insert at position\n4. Delete at head\n5. Delete at tail\n6. Delete at position\n7. search\n8. display\n9. Exit\nEnter your choice : ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter data to insert at the head: ");

scanf("%d", &data);

insertAtHead(&head, data);

break;

case 2:

printf("Enter data to insert at the tail: ");

scanf("%d", &data);

insertAtTail(&head, data);

break;

case 3:

printf("Enter data to insert: ");

scanf("%d", &data);

printf("Enter the position: ");

scanf("%d", &position);

insertAtPosition(&head, data, position);

break;

case 4:

deleteAtHead(&head);

break;

case 5:

deleteAtTail(&head);

break;

case 6:

printf("Enter the position to delete: ");

scanf("%d", &position);

deleteFromPosition(&head, position);

break;

case 7:

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

scanf("%d", &data);

result = searchElement(head, data);

if (result != NULL) {

printf("Element %d found in the list.\n", data);

} else {

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

}

break;

case 8:

displayList(head);

break;

case 9:

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

break;

default:

printf("Invalid choice. Please select a valid option.\n");

break;

}

} while (choice != 9);

// Clean up: Free memory

if (head != NULL) {

struct Node\* current = head;

struct Node\* temp = NULL;

do {

temp = current->next;

free(current);

current = temp;

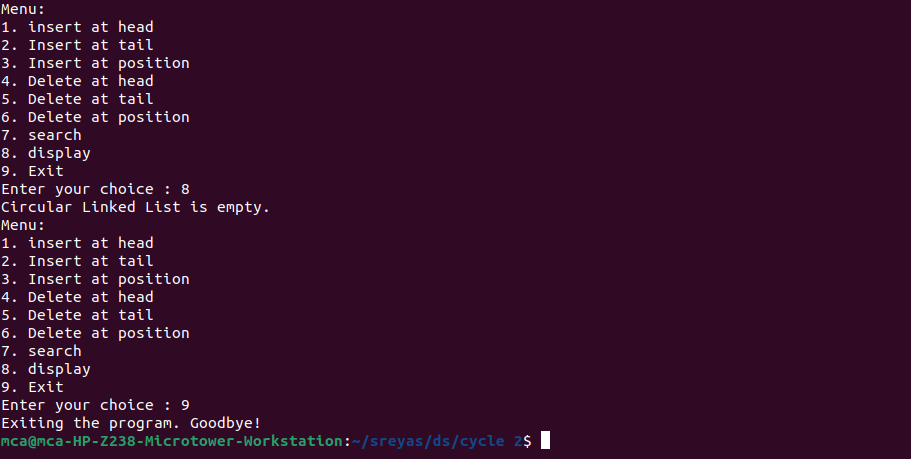
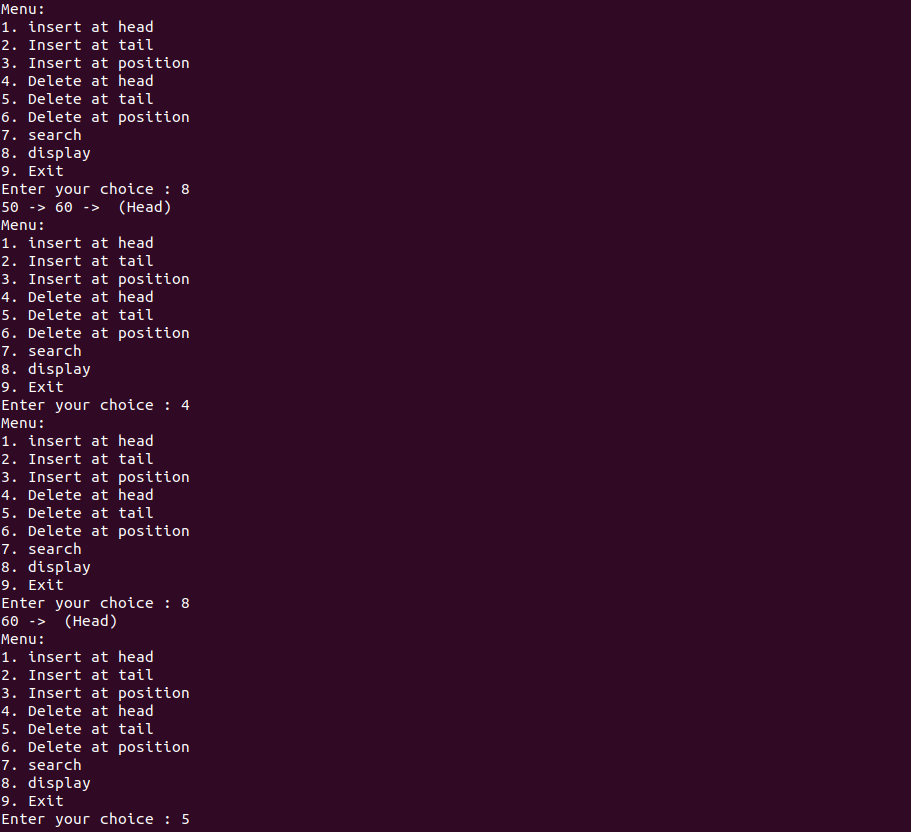
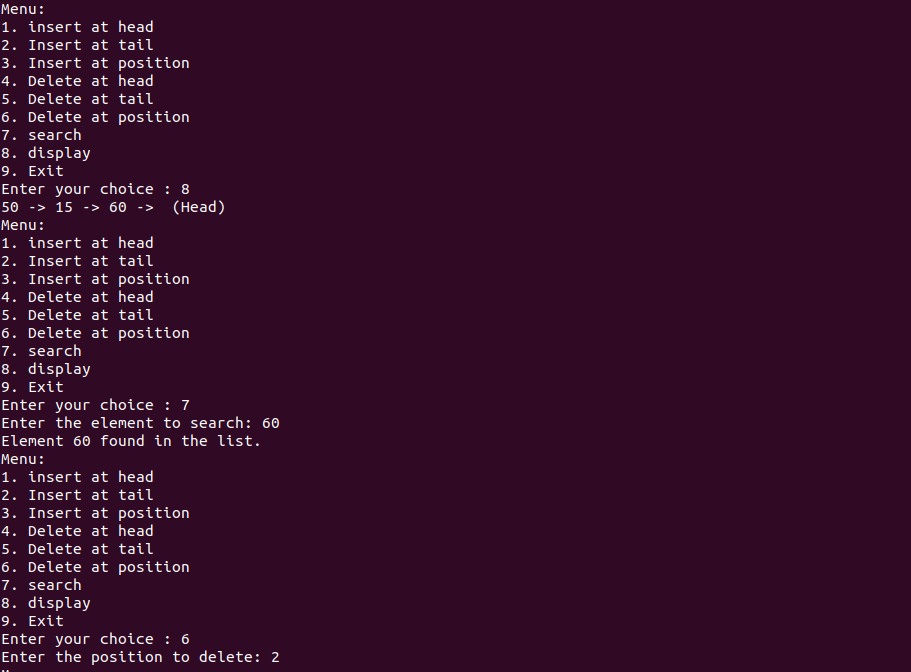
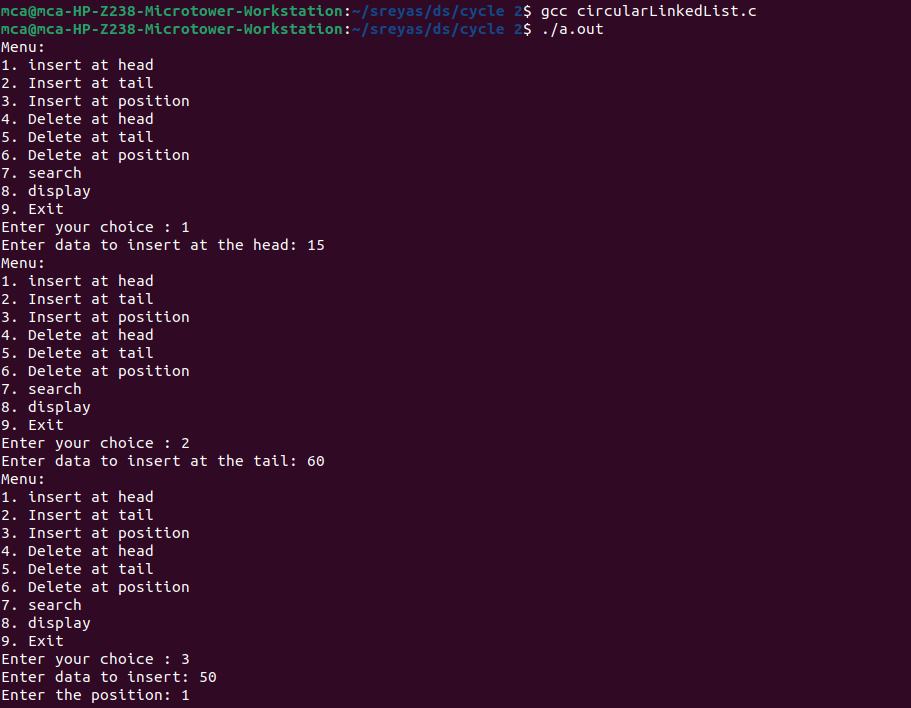
} while (current != head);

}

return 0;

}

**Output:**

****

1. **Implement binary search tree**

**Program:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* createNode(int value) {

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

newNode->data = value;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

struct Node\* insert(struct Node\* root, int value) {

if (root == NULL) {

return createNode(value);

}

if (value < root->data) {

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

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

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

}

return root;

}

struct Node\* findMin(struct Node\* node) {

while (node->left != NULL) {

node = node->left;

}

return node;

}

struct Node\* delete(struct Node\* root, int value) {

if (root == NULL) {

return root;

}

if (value < root->data) {

root->left = delete(root->left, value);

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

root->right = delete(root->right, value);

} else {

if (root->left == NULL) {

struct Node\* temp = root->right;

free(root);

return temp;

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

struct Node\* temp = root->left;

free(root);

return temp;

}

struct Node\* temp = findMin(root->right);

root->data = temp->data;

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

}

return root;

}

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, choice;

do{

printf("\n1.insert\n2.delete\n3.display(in-order)\n4.exit\nEnter your operation : ");

scanf("%d", &choice);

switch(choice){

case 1:

printf("enter the value to insert : ");

scanf("%d", &value);

root = insert(root, value);

break;

case 2:

printf("enter the value to delete : ");

scanf("%d", &value);

root = delete(root, value);

break;

case 3:

inorderTraversal(root);

break;

default:

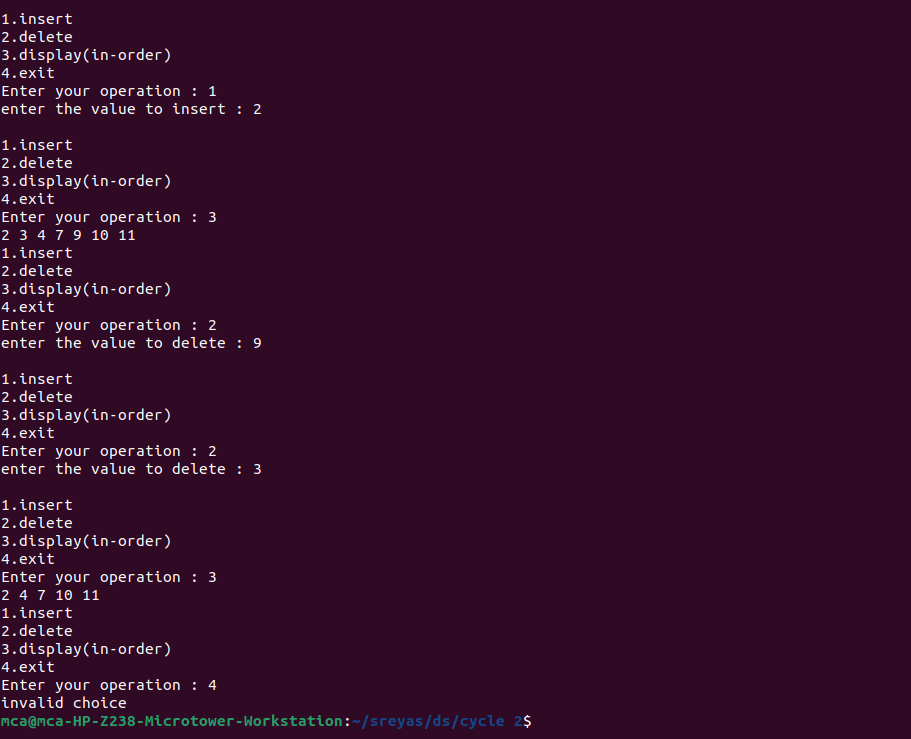
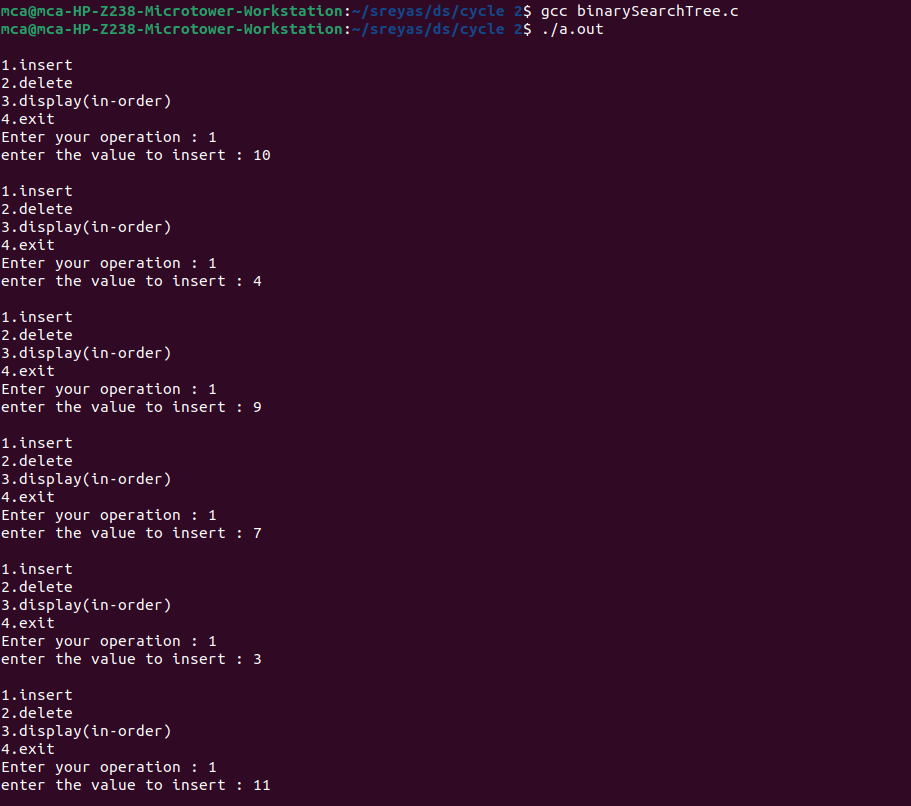
printf("invalid choice\n");

}

} while(choice != 4);

return 0;

}

**Output:**

1. **Implement balanced-binary-search tree**

**Program:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

int height; // Height of the node

};

// Function to calculate the height of a node

int height(struct Node\* node) {

if (node == NULL)

return 0;

return node->height;

}

// Function to get the maximum of two integers

int max(int a, int b) {

return (a > b) ? a : b;

}

// Function to create a new node

struct Node\* createNode(int value) {

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

newNode->data = value;

newNode->left = NULL;

newNode->right = NULL;

newNode->height = 1;

return newNode;

}

// Function to perform a right rotation

struct Node\* rightRotate(struct Node\* y) {

struct Node\* x = y->left;

struct Node\* T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Update heights

y->height = max(height(y->left), height(y->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

return x;

}

// Function to perform a left rotation

struct Node\* leftRotate(struct Node\* x) {

struct Node\* y = x->right;

struct Node\* T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

x->height = max(height(x->left), height(x->right)) + 1;

y->height = max(height(y->left), height(y->right)) + 1;

return y;

}

// Get the balance factor of a node

int getBalance(struct Node\* node) {

if (node == NULL)

return 0;

return height(node->left) - height(node->right);

}

// Function to insert a node into the AVL tree

struct Node\* insert(struct Node\* node, int value) {

// Perform the standard BST insert

if (node == NULL)

return createNode(value);

if (value < node->data)

node->left = insert(node->left, value);

else if (value > node->data)

node->right = insert(node->right, value);

else // Duplicate values are not allowed

return node;

// Update height of the current node

node->height = 1 + max(height(node->left), height(node->right));

// Get the balance factor to check if the node became unbalanced

int balance = getBalance(node);

// Perform rotations if necessary to restore balance

// Left Heavy (LL and LR cases)

if (balance > 1 && value < node->left->data)

return rightRotate(node);

// Right Heavy (RR and RL cases)

if (balance < -1 && value > node->right->data)

return leftRotate(node);

// Left Right Heavy (LR case)

if (balance > 1 && value > node->left->data) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

// Right Left Heavy (RL case)

if (balance < -1 && value < node->right->data) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

return node;

}

// Function to perform in-order traversal of the AVL tree

void inorderTraversal(struct Node\* node) {

if (node != NULL) {

inorderTraversal(node->left);

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

inorderTraversal(node->right);

}

}

int main() {

struct Node\* root = NULL;

int n, value;

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

scanf("%d", &n);

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

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

scanf("%d", &value);

root = insert(root, value);

}

printf("In-order traversal of the AVL tree: ");

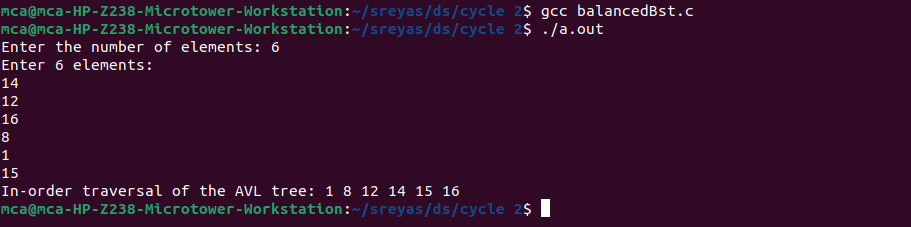
inorderTraversal(root);

printf("\n");

return 0;

}

**Output:**

****

1. **Implement set operations (union, intersection, difference)**

**Program:**

#include <stdio.h>

// Function to print a set

void printSet(int set[], int size) {

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

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

}

printf("\n");

}

// Function to perform the union of two sets

void setUnion(int set1[], int size1, int set2[], int size2, int result[], int \*resultSize) {

int i = 0, j = 0, k = 0;

while (i < size1 && j < size2) {

if (set1[i] < set2[j]) {

result[k++] = set1[i++];

} else if (set1[i] > set2[j]) {

result[k++] = set2[j++];

} else {

result[k++] = set1[i++];

j++;

}

}

while (i < size1) {

result[k++] = set1[i++];

}

while (j < size2) {

result[k++] = set2[j++];

}

\*resultSize = k;

}

// Function to perform the intersection of two sets

void setIntersection(int set1[], int size1, int set2[], int size2, int result[], int \*resultSize) {

int i = 0, j = 0, k = 0;

while (i < size1 && j < size2) {

if (set1[i] < set2[j]) {

i++;

} else if (set1[i] > set2[j]) {

j++;

} else {

result[k++] = set1[i++];

j++;

}

}

\*resultSize = k;

}

// Function to perform the set difference (set1 - set2)

void setDifference(int set1[], int size1, int set2[], int size2, int result[], int \*resultSize) {

int i = 0, j = 0, k = 0;

while (i < size1 && j < size2) {

if (set1[i] < set2[j]) {

result[k++] = set1[i++];

} else if (set1[i] > set2[j]) {

j++;

} else {

i++;

j++;

}

}

while (i < size1) {

result[k++] = set1[i++];

}

\*resultSize = k;

}

int main() {

int size1, size2;

printf("Enter the size of Set 1: ");

scanf("%d", &size1);

int set1[size1];

printf("Enter %d elements for Set 1:\n", size1);

for (int i = 0; i < size1; i++) {

scanf("%d", &set1[i]);

}

printf("Enter the size of Set 2: ");

scanf("%d", &size2);

int set2[size2];

printf("Enter %d elements for Set 2:\n", size2);

for (int i = 0; i < size2; i++) {

scanf("%d", &set2[i]);

}

int unionResult[size1 + size2];

int unionSize = 0;

int intersectionResult[size1 < size2 ? size1 : size2];

int intersectionSize = 0;

int differenceResult[size1];

int differenceSize = 0;

printf("Set 1: ");

printSet(set1, size1);

printf("Set 2: ");

printSet(set2, size2);

setUnion(set1, size1, set2, size2, unionResult, &unionSize);

printf("Union: ");

printSet(unionResult, unionSize);

setIntersection(set1, size1, set2, size2, intersectionResult, &intersectionSize);

printf("Intersection: ");

printSet(intersectionResult, intersectionSize);

setDifference(set1, size1, set2, size2, differenceResult, &differenceSize);

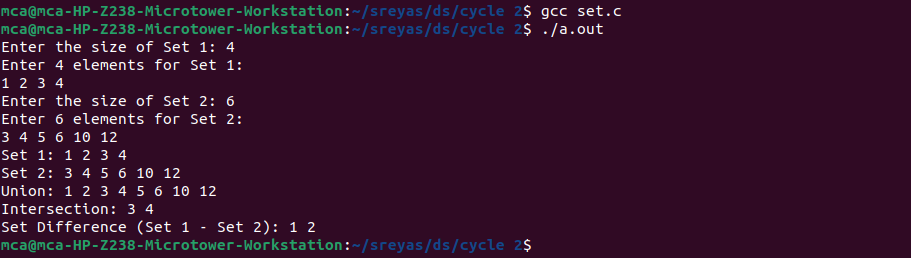
printf("Set Difference (Set 1 - Set 2): ");

printSet(differenceResult, differenceSize);

return 0;

}

**Output:**

****

1. **Implement disjoint set operations.**

**Program:**

#include <stdio.h>

#define MAX 100

int parent[MAX], rank[MAX], n;

// Function to find the representative of the set containing element x

int find(int x) {

if (x != parent[x])

parent[x] = find(parent[x]); // Path Compression

return parent[x];

}

int main() {

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

if (scanf("%d", &n) != 1 || n <= 0 || n > MAX) {

printf("Invalid input. Please enter a positive integer less than or equal to %d.\n", MAX);

return 1;

}

// Initialize sets

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

parent[i] = i;

rank[i] = 0;

}

int choice, x, y;

while (1) {

printf("\nOperations:\n1. Union\n2. Find\n3. Display Set Representatives\n4. Exit\nEnter your choice: ");

if (scanf("%d", &choice) != 1) {

printf("Invalid input. Please enter an integer.\n");

continue;

}

switch (choice) {

case 1:

// Union operation

printf("Enter elements to perform union: ");

if (scanf("%d %d", &x, &y) != 2 || x < 0 || x >= n || y < 0 || y >= n) {

printf("Invalid input. Please enter valid elements.\n");

} else {

int rootX = find(x);

int rootY = find(y);

if (rootX == rootY) {

printf("%d and %d are already in the same set.\n", x, y);

} else {

// Merge sets

if (rank[rootX] > rank[rootY]) {

parent[rootY] = rootX;

} else if (rank[rootX] < rank[rootY]) {

parent[rootX] = rootY;

} else {

parent[rootY] = rootX;

rank[rootX]++;

}

printf("Union of %d and %d is performed.\n", x, y);

}

}

break;

case 2:

// Find operation

printf("Enter element to find its set: ");

if (scanf("%d", &x) != 1 || x < 0 || x >= n) {

printf("Invalid input. Please enter a valid element.\n");

} else {

printf("Set representative of %d is %d\n", x, find(x));

}

break;

case 3:

// Display set representatives

printf("Set Representatives:\n");

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

printf("Element %d belongs to set with representative %d\n", i, find(i));

}

break;

case 4:

return 0;

default:

printf("Invalid choice. Please enter a valid option.\n");

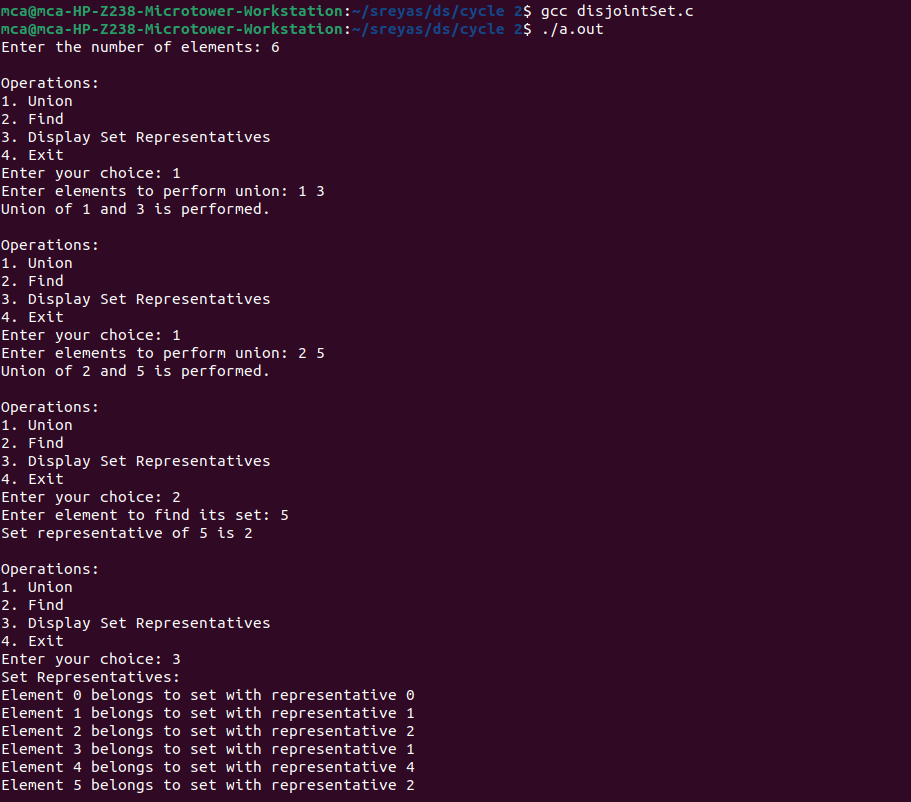
break;

}

}

}

**Output:**

****

1. **Implement tree traversal methods DFS ( In-order, Pre-Order, Post-Order), and BFS**

**Program:**

#include <stdio.h>

#include <stdlib.h>

struct Node

{

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* createNode(int value)

{

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

newNode->data = value;

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

return newNode;

}

struct Node\* insert(struct Node\* root, int value)

{

if (root == NULL)

return createNode(value);

if (value < root->data)

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

else if (value > root->data)

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

return root;

}

void inOrderTraversal(struct Node\* root)

{

if (root != NULL)

{

inOrderTraversal(root->left);

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

inOrderTraversal(root->right);

}

}

void preOrderTraversal(struct Node\* root)

{

if (root != NULL)

{

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

preOrderTraversal(root->left);

preOrderTraversal(root->right);

}

}

void postOrderTraversal(struct Node\* root)

{

if (root != NULL)

{

postOrderTraversal(root->left);

postOrderTraversal(root->right);

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

}

}

void breadthFirstSearch(struct Node\* root)

{

if (root == NULL)

return;

struct Node\* queue[100];

int front = -1, rear = -1;

queue[++rear] = root;

while (front < rear)

{

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

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

if (current->left != NULL)

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

if (current->right != NULL)

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

}

}

int main()

{

struct Node\* root = NULL;

int choice, value;

do

{

printf("\nBinary Tree Operations:\n");

printf("1. Insert\n");

printf("2. In-order Traversal\n");

printf("3. Pre-order Traversal\n");

printf("4. Post-order Traversal\n");

printf("5. Breadth-First Search (BFS)\n");

printf("6. Quit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice)

{

case 1:

printf("Enter value to insert: ");

scanf("%d", &value);

root = insert(root, value);

break;

case 2:

printf("In-order Traversal: ");

inOrderTraversal(root);

printf("\n");

break;

case 3:

printf("Pre-order Traversal: ");

preOrderTraversal(root);

printf("\n");

break;

case 4:

printf("Post-order Traversal: ");

postOrderTraversal(root);

printf("\n");

break;

case 5:

printf("Breadth-First Search (BFS): ");

breadthFirstSearch(root);

printf("\n");

break;

case 6:

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

break;

default:

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

break;

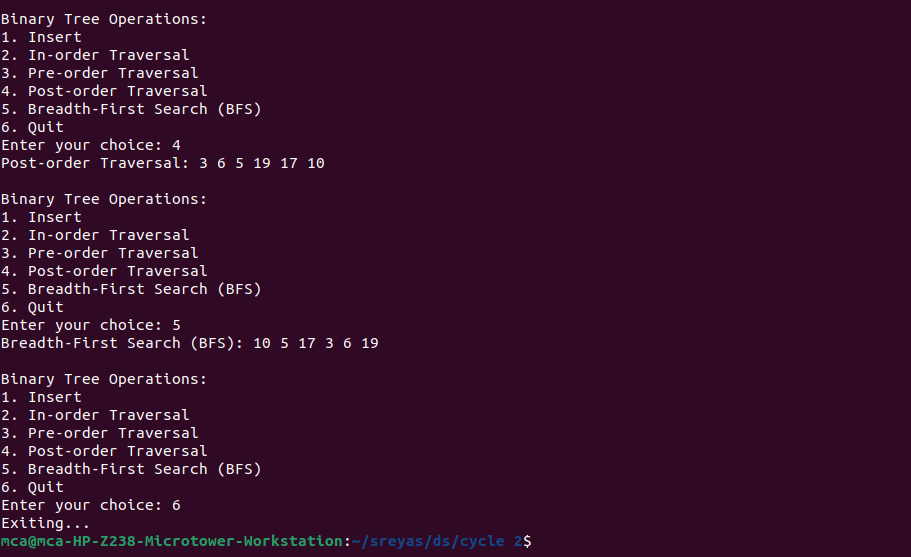
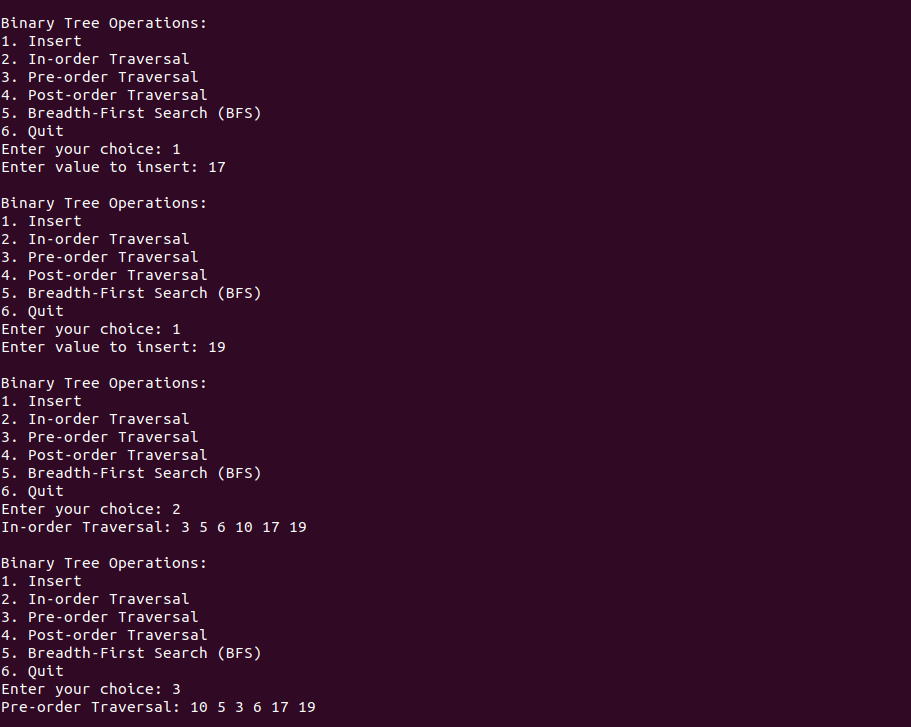
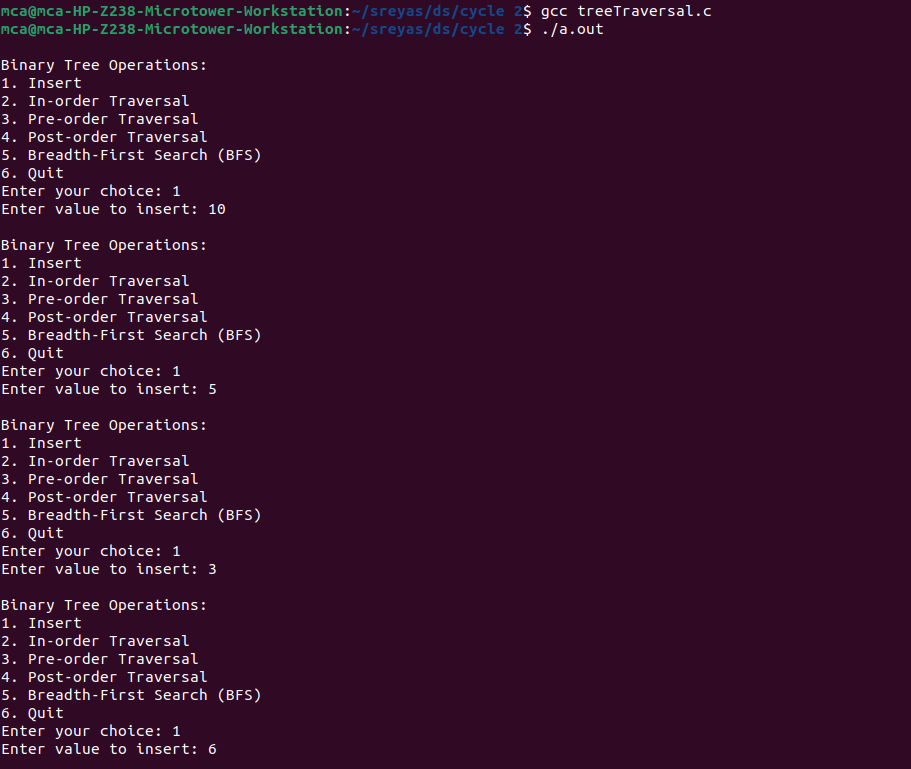
}

}

while (choice != 6);

return 0;

}

**Output:**

1. **Implement Binomial Heaps and operations (Create, Insert, Delete)**

**Program:**

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in the Binomial Heap

struct Node {

int data;

int degree; // Degree of the node

struct Node \*parent;

struct Node \*child;

struct Node \*sibling;

};

// Structure to represent a Binomial Heap

struct BinomialHeap {

struct Node \*head;

};

// Function to create a new node

struct Node \*createNode(int data) {

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

newNode->data = data;

newNode->degree = 0;

newNode->parent = NULL;

newNode->child = NULL;

newNode->sibling = NULL;

return newNode;

}

// Function to merge two Binomial Heaps

struct Node \*mergeHeaps(struct Node \*h1, struct Node \*h2) {

if (h1 == NULL)

return h2;

if (h2 == NULL)

return h1;

struct Node \*mergedHeap = NULL;

struct Node \*temp1 = h1;

struct Node \*temp2 = h2;

// Choose the node with smaller degree as the root of the merged heap

if (temp1->degree <= temp2->degree) {

mergedHeap = temp1;

temp1 = temp1->sibling;

} else {

mergedHeap = temp2;

temp2 = temp2->sibling;

}

struct Node \*current = mergedHeap;

// Merge the remaining nodes

while (temp1 != NULL && temp2 != NULL) {

if (temp1->degree <= temp2->degree) {

current->sibling = temp1;

temp1 = temp1->sibling;

} else {

current->sibling = temp2;

temp2 = temp2->sibling;

}

current = current->sibling;

}

// Attach the remaining nodes, if any

if (temp1 != NULL)

current->sibling = temp1;

else

current->sibling = temp2;

return mergedHeap;

}

// Function to link two Binomial Trees

void linkNodes(struct Node \*root1, struct Node \*root2) {

root1->parent = root2;

root1->sibling = root2->child;

root2->child = root1;

root2->degree++;

}

// Function to union two Binomial Heaps

struct Node \*unionHeaps(struct Node \*h1, struct Node \*h2) {

struct Node \*mergedHeap = mergeHeaps(h1, h2);

if (mergedHeap == NULL)

return NULL;

struct Node \*prev = NULL;

struct Node \*current = mergedHeap;

struct Node \*next = mergedHeap->sibling;

while (next != NULL) {

if ((current->degree != next->degree) || (next->sibling != NULL && next->sibling->degree == current->degree)) {

prev = current;

current = next;

} else {

if (current->data <= next->data) {

current->sibling = next->sibling;

linkNodes(next, current);

} else {

if (prev == NULL)

mergedHeap = next;

else

prev->sibling = next;

linkNodes(current, next);

current = next;

}

}

next = current->sibling;

}

return mergedHeap;

}

// Function to insert a new key into the Binomial Heap

struct Node \*insert(struct Node \*heap, int key) {

struct BinomialHeap \*newHeap = (struct BinomialHeap \*)malloc(sizeof(struct BinomialHeap));

newHeap->head = createNode(key);

return unionHeaps(heap, newHeap->head);

}

// Function to find the minimum key in the Binomial Heap

struct Node \*findMin(struct Node \*heap) {

if (heap == NULL)

return NULL;

struct Node \*minNode = heap;

struct Node \*current = heap;

while (current != NULL) {

if (current->data < minNode->data)

minNode = current;

current = current->sibling;

}

return minNode;

}

// Function to delete the minimum key from the Binomial Heap

struct Node \*deleteMin(struct Node \*heap) {

if (heap == NULL)

return NULL;

struct Node \*minNode = findMin(heap);

struct Node \*prev = NULL;

struct Node \*current = heap;

// Find the parent of the minNode

while (current != minNode) {

prev = current;

current = current->sibling;

}

// Remove minNode from the list

if (prev == NULL)

heap = minNode->sibling;

else

prev->sibling = minNode->sibling;

// Reverse the order of minNode's children to form a new Binomial Heap

struct Node \*newHeap = NULL;

struct Node \*child = minNode->child;

while (child != NULL) {

struct Node \*nextChild = child->sibling;

child->sibling = newHeap;

child->parent = NULL;

newHeap = child;

child = nextChild;

}

// Union the two heaps

heap = unionHeaps(heap, newHeap);

free(minNode);

return heap;

}

// Function to display the Binomial Heap

void displayHeap(struct Node \*heap) {

while (heap != NULL) {

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

heap = heap->sibling;

}

printf("\n");

}

int main() {

struct Node \*heapHead = NULL; // Use a separate pointer for the heap head

int choice, key;

do {

printf("\n------ Binomial Heap Operations ------\n");

printf("1. Insert\n");

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

printf("3. Display\n");

printf("4. Quit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter key to insert: ");

scanf("%d", &key);

heapHead = insert(heapHead, key);

printf("Key %d inserted successfully.\n", key);

break;

case 2:

heapHead = deleteMin(heapHead);

printf("Minimum key deleted.\n");

break;

case 3:

printf("Binomial Heap Root Nodes:\n");

displayHeap(heapHead); // Call the modified display function

printf("\n");

break;

case 4:

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

break;

default:

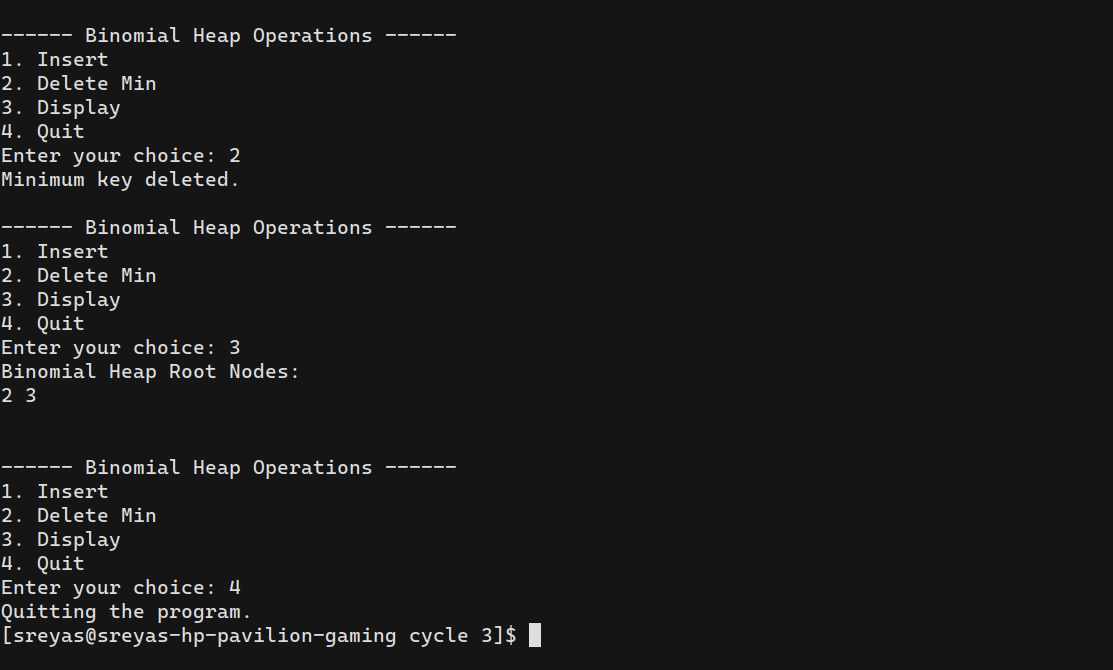
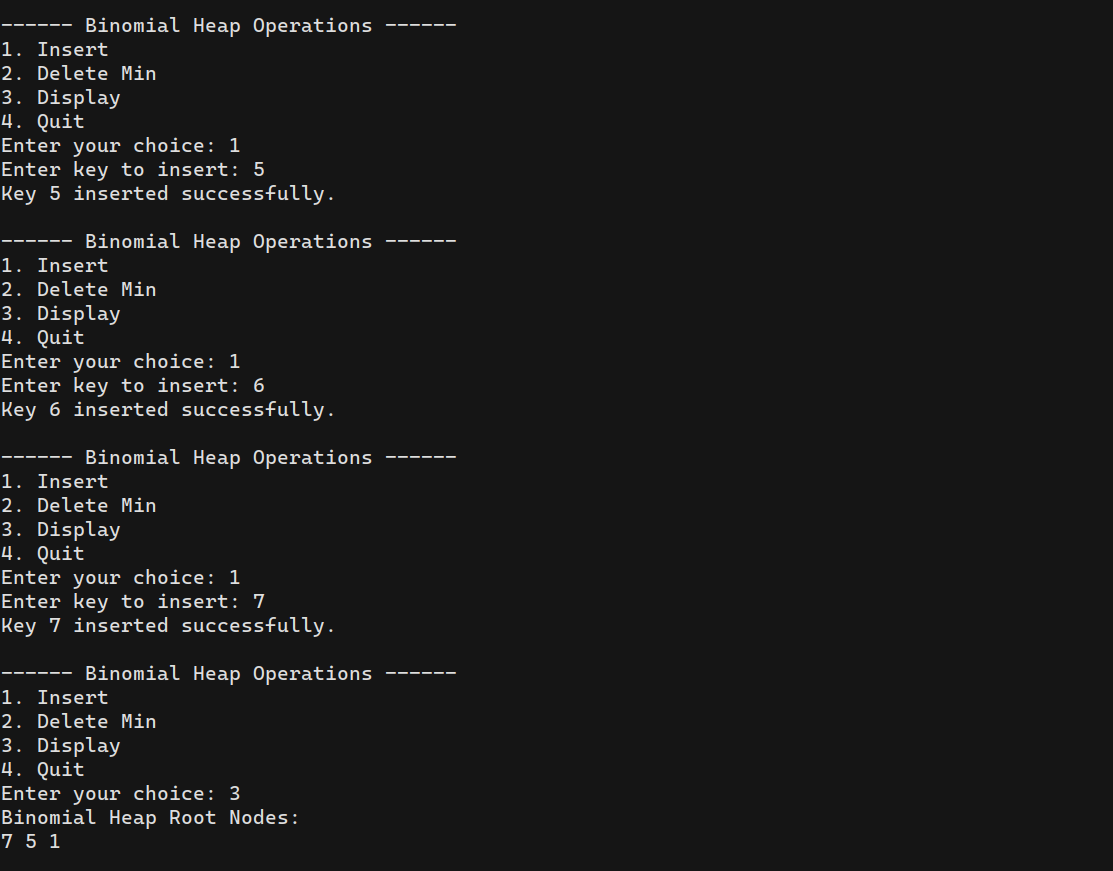
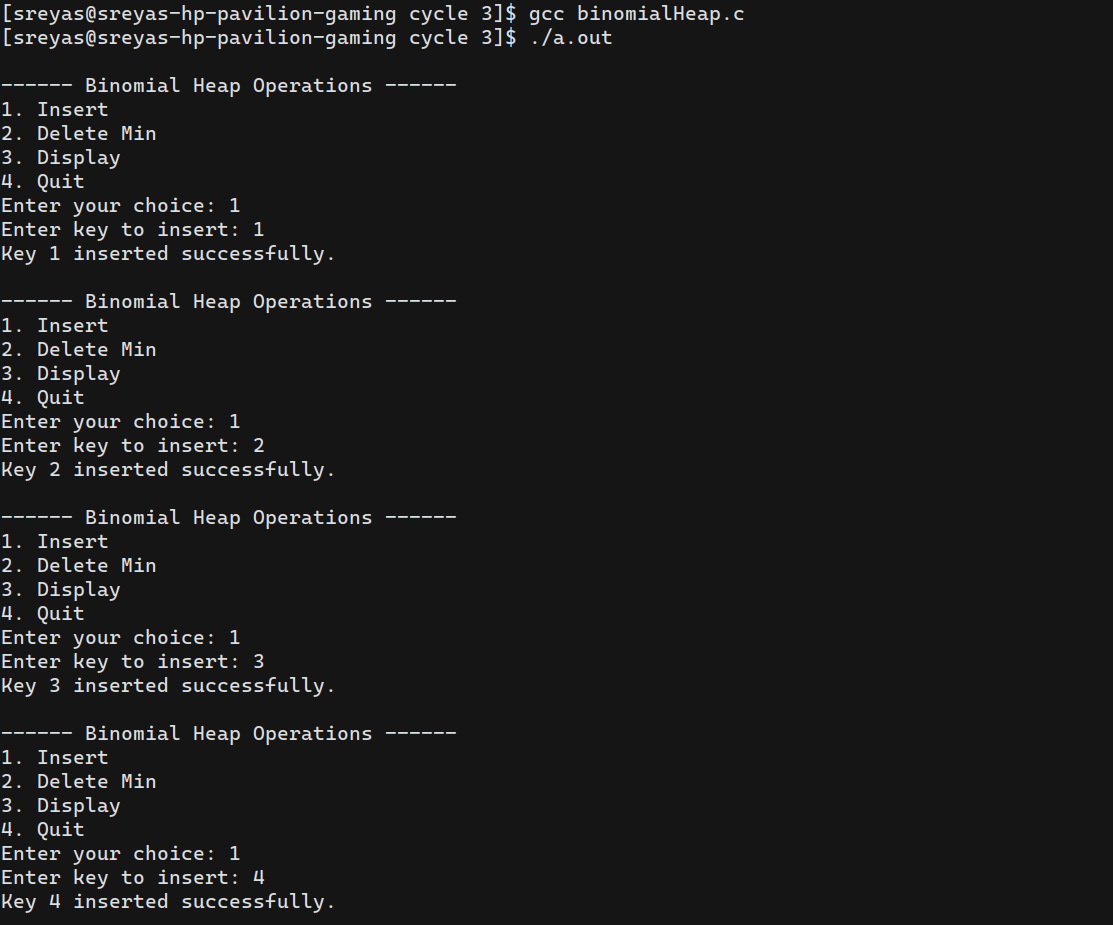
printf("Invalid choice. Please enter a valid option.\n");

}

} while (choice != 4);

return 0;

}

**Output:**

1. **Implement B Trees and its operations**

**Program:**

#include <stdio.h>

#include <stdlib.h>

#define MAX 4

#define MIN 2

struct btreeNode

{

int val[MAX + 1], count;

struct btreeNode \*link[MAX + 1];

};

struct btreeNode \*root;

struct btreeNode \* createNode(int val, struct btreeNode \*child)

{

struct btreeNode \*newNode;

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

newNode->val[1] = val;

newNode->count = 1;

newNode->link[0] = root;

newNode->link[1] = child;

return newNode;

}

void addValToNode(int val, int pos, struct btreeNode \*node,

struct btreeNode \*child)

{

int j = node->count;

while (j > pos)

{

node->val[j + 1] = node->val[j];

node->link[j + 1] = node->link[j];

j--;

}

node->val[j + 1] = val;

node->link[j + 1] = child;

node->count++;

}

void splitNode (int val, int \*pval, int pos, struct btreeNode \*node,

struct btreeNode \*child, struct btreeNode \*\*newNode)

{

int median, j;

if (pos > MIN)

median = MIN + 1;

else

median = MIN;

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

j = median + 1;

while (j <= MAX)

{

(\*newNode)->val[j - median] = node->val[j];

(\*newNode)->link[j - median] = node->link[j];

j++;

}

node->count = median;

(\*newNode)->count = MAX - median;

if (pos <= MIN)

{

addValToNode(val, pos, node, child);

}

else

{

addValToNode(val, pos - median, \*newNode, child);

}

\*pval = node->val[node->count];

(\*newNode)->link[0] = node->link[node->count];

node->count--;

}

int setValueInNode(int val, int \*pval,

struct btreeNode \*node, struct btreeNode \*\*child)

{

int pos;

if (!node)

{

\*pval = val;

\*child = NULL;

return 1;

}

if (val < node->val[1])

{

pos = 0;

}

else

{

for (pos = node->count;

(val < node->val[pos] && pos > 1); pos--);

if (val == node->val[pos])

{

printf("Duplicates not allowed\n");

return 0;

}

}

if (setValueInNode(val, pval, node->link[pos], child))

{

if (node->count < MAX)

{

addValToNode(\*pval, pos, node, \*child);

} else

{

splitNode(\*pval, pval, pos, node, \*child, child);

return 1;

}

}

return 0;

}

void insertion(int val)

{

int flag, i;

struct btreeNode \*child;

flag = setValueInNode(val, &i, root, &child);

if (flag)

root = createNode(i, child);

}

void copySuccessor(struct btreeNode \*myNode, int pos)

{

struct btreeNode \*dummy;

dummy = myNode->link[pos];

for (;dummy->link[0] != NULL;)

dummy = dummy->link[0];

myNode->val[pos] = dummy->val[1];

}

void removeVal(struct btreeNode \*myNode, int pos)

{

int i = pos + 1;

while (i <= myNode->count) {

myNode->val[i - 1] = myNode->val[i];

myNode->link[i - 1] = myNode->link[i];

i++;

}

myNode->count--;

}

void doRightShift(struct btreeNode \*myNode, int pos)

{

struct btreeNode \*x = myNode->link[pos];

int j = x->count;

while (j > 0) {

x->val[j + 1] = x->val[j];

x->link[j + 1] = x->link[j];

}

x->val[1] = myNode->val[pos];

x->link[1] = x->link[0];

x->count++;

x = myNode->link[pos - 1];

myNode->val[pos] = x->val[x->count];

myNode->link[pos] = x->link[x->count];

x->count--;

return;

}

void doLeftShift(struct btreeNode \*myNode, int pos)

{

int j = 1;

struct btreeNode \*x = myNode->link[pos - 1];

x->count++;

x->val[x->count] = myNode->val[pos];

x->link[x->count] = myNode->link[pos]->link[0];

x = myNode->link[pos];

myNode->val[pos] = x->val[1];

x->link[0] = x->link[1];

x->count--;

while (j <= x->count) {

x->val[j] = x->val[j + 1];

x->link[j] = x->link[j + 1];

j++;

}

return;

}

void mergeNodes(struct btreeNode \*myNode, int pos)

{

int j = 1;

struct btreeNode \*x1 = myNode->link[pos], \*x2 = myNode->link[pos - 1];

x2->count++;

x2->val[x2->count] = myNode->val[pos];

x2->link[x2->count] = myNode->link[0];

while (j <= x1->count)

{

x2->count++;

x2->val[x2->count] = x1->val[j];

x2->link[x2->count] = x1->link[j];

j++;

}

j = pos;

while (j < myNode->count)

{

myNode->val[j] = myNode->val[j + 1];

myNode->link[j] = myNode->link[j + 1];

j++;

}

myNode->count--;

free(x1);

}

void adjustNode(struct btreeNode \*myNode, int pos)

{

if (!pos) {

if (myNode->link[1]->count > MIN)

{

doLeftShift(myNode, 1);

} else

{

mergeNodes(myNode, 1);

}

} else

{

if (myNode->count != pos)

{

if(myNode->link[pos - 1]->count > MIN)

{

doRightShift(myNode, pos);

} else

{

if (myNode->link[pos + 1]->count > MIN)

{

doLeftShift(myNode, pos + 1);

} else

{

mergeNodes(myNode, pos);

}

}

} else

{

if (myNode->link[pos - 1]->count > MIN)

doRightShift(myNode, pos);

else

mergeNodes(myNode, pos);

}

}

}

int delValFromNode(int val, struct btreeNode \*myNode)

{

int pos, flag = 0;

if (myNode) {

if (val < myNode->val[1])

{

pos = 0;

flag = 0;

} else

{

for (pos = myNode->count;

(val < myNode->val[pos] && pos > 1); pos--);

if (val == myNode->val[pos])

{

flag = 1;

}

else

{

flag = 0;

}

}

if (flag)

{

if (myNode->link[pos - 1])

{

copySuccessor(myNode, pos);

flag = delValFromNode(myNode->val[pos], myNode->link[pos]);

if (flag == 0)

{

printf("Given data is not present in B-Tree\n");

}

} else

{

removeVal(myNode, pos);

}

} else

{

flag = delValFromNode(val, myNode->link[pos]);

}

if (myNode->link[pos])

{

if (myNode->link[pos]->count < MIN)

adjustNode(myNode, pos);

}

}

return flag;

}

void deletion(int val, struct btreeNode \*myNode)

{

struct btreeNode \*tmp;

if (!delValFromNode(val, myNode))

{

printf("Given value is not present in B-Tree\n");

return;

} else

{

if (myNode->count == 0)

{

tmp = myNode;

myNode = myNode->link[0];

free(tmp);

}

}

root = myNode;

return;

}

void searching(int val, int \*pos, struct btreeNode \*myNode)

{

if (!myNode)

{

return;

}

if (val < myNode->val[1])

{

\*pos = 0;

} else

{

for (\*pos = myNode->count;

(val < myNode->val[\*pos] && \*pos > 1); (\*pos)--);

if (val == myNode->val[\*pos]) {

printf("Given data %d is present in B-Tree", val);

return;

}

}

searching(val, pos, myNode->link[\*pos]);

return;

}

void traversal(struct btreeNode \*myNode)

{

int i;

if (myNode)

{

for (i = 0; i < myNode->count; i++)

{

traversal(myNode->link[i]);

printf("%d ", myNode->val[i + 1]);

}

traversal(myNode->link[i]);

}

}

int main()

{

int val, ch;

while (1)

{

printf("\n1. Insertion\n2. Deletion\n3. Searching\n4. Traversal\n5. Exit\nEnter your choice:\n");

scanf("%d", &ch);

switch (ch)

{

case 1:

printf("Enter your element:");

scanf("%d", &val);

insertion(val);

break;

case 2:

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

scanf("%d", &val);

deletion(val, root);

break;

case 3:

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

scanf("%d", &val);

searching(val, &ch, root);

break;

case 4:

traversal(root);

break;

case 5:

exit(0);

default:

printf("U have entered wrong option!!\n");

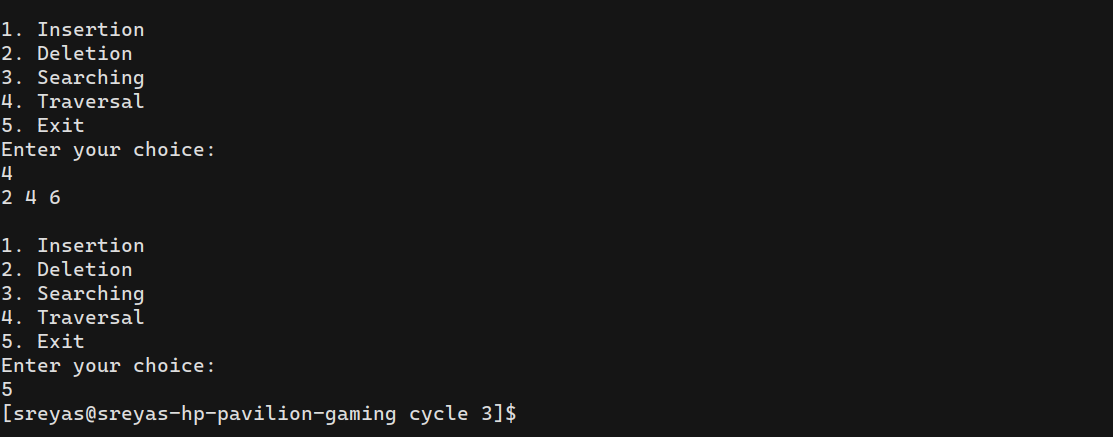
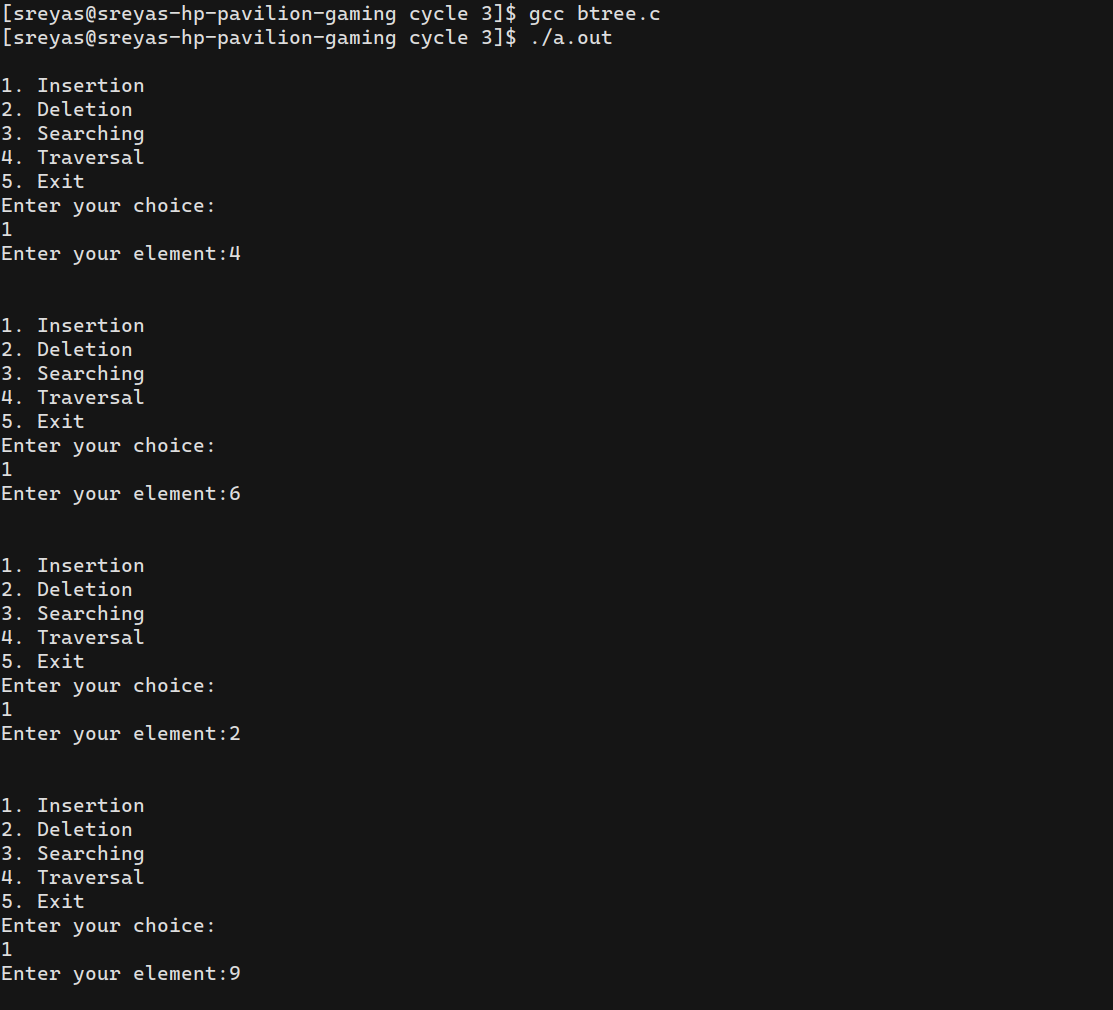
break;

}

printf("\n");

}

}

**Output:**

1. **Implement Red Black Trees and its operations**

**Program:**

#include <stdio.h>

#include <stdlib.h>

// Node structure for Red-Black Tree

struct Node {

int data;

char color; // 'R' for red, 'B' for black

struct Node \*left, \*right, \*parent;

};

struct Node \*createNode(int data) {

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

newNode->data = data;

newNode->color = 'R'; // New nodes are always red

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

return newNode;

}

// Function prototypes

void leftRotate(struct Node \*\*root, struct Node \*x);

void rightRotate(struct Node \*\*root, struct Node \*y);

void insertFixup(struct Node \*\*root, struct Node \*z);

void insertNode(struct Node \*\*root, int data);

void transplant(struct Node \*\*root, struct Node \*u, struct Node \*v);

struct Node \*treeMinimum(struct Node \*x);

void deleteFixup(struct Node \*\*root, struct Node \*x);

void deleteNode(struct Node \*\*root, int data);

void inOrderTraversal(struct Node \*root);

int main() {

struct Node \*root = NULL;

int choice, data;

do {

printf("\n------ Red-Black Tree Operations ------\n");

printf("1. Insert\n");

printf("2. Delete\n");

printf("3. Display (In-Order Traversal)\n");

printf("4. Quit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter data to insert: ");

scanf("%d", &data);

insertNode(&root, data);

printf("Data %d inserted successfully.\n", data);

break;

case 2:

printf("Enter data to delete: ");

scanf("%d", &data);

deleteNode(&root, data);

printf("Data %d deleted successfully.\n", data);

break;

case 3:

printf("In-Order Traversal:\n");

inOrderTraversal(root);

printf("\n");

break;

case 4:

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

break;

default:

printf("Invalid choice. Please enter a valid option.\n");

}

} while (choice != 4);

return 0;

}

// Left rotation in Red-Black Tree

void leftRotate(struct Node \*\*root, struct Node \*x) {

struct Node \*y = x->right;

x->right = y->left;

if (y->left != NULL)

y->left->parent = x;

y->parent = x->parent;

if (x->parent == NULL)

\*root = y;

else if (x == x->parent->left)

x->parent->left = y;

else

x->parent->right = y;

y->left = x;

x->parent = y;

}

// Right rotation in Red-Black Tree

void rightRotate(struct Node \*\*root, struct Node \*y) {

struct Node \*x = y->left;

y->left = x->right;

if (x->right != NULL)

x->right->parent = y;

x->parent = y->parent;

if (y->parent == NULL)

\*root = x;

else if (y == y->parent->left)

y->parent->left = x;

else

y->parent->right = x;

x->right = y;

y->parent = x;

}

// Fix the Red-Black Tree properties after insertion

void insertFixup(struct Node \*\*root, struct Node \*z) {

while (z->parent != NULL && z->parent->color == 'R') {

if (z->parent == z->parent->parent->left) {

struct Node \*y = z->parent->parent->right;

if (y != NULL && y->color == 'R') {

z->parent->color = 'B';

y->color = 'B';

z->parent->parent->color = 'R';

z = z->parent->parent;

} else {

if (z == z->parent->right) {

z = z->parent;

leftRotate(root, z);

}

z->parent->color = 'B';

z->parent->parent->color = 'R';

rightRotate(root, z->parent->parent);

}

} else {

struct Node \*y = z->parent->parent->left;

if (y != NULL && y->color == 'R') {

z->parent->color = 'B';

y->color = 'B';

z->parent->parent->color = 'R';

z = z->parent->parent;

} else {

if (z == z->parent->left) {

z = z->parent;

rightRotate(root, z);

}

z->parent->color = 'B';

z->parent->parent->color = 'R';

leftRotate(root, z->parent->parent);

}

}

}

(\*root)->color = 'B'; // Root should always be black

}

// Insert a node into the Red-Black Tree

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

struct Node \*z = createNode(data);

struct Node \*y = NULL;

struct Node \*x = \*root;

while (x != NULL) {

y = x;

if (z->data < x->data)

x = x->left;

else

x = x->right;

}

z->parent = y;

if (y == NULL)

\*root = z;

else if (z->data < y->data)

y->left = z;

else

y->right = z;

insertFixup(root, z);

}

// Transplant a subtree in the Red-Black Tree

void transplant(struct Node \*\*root, struct Node \*u, struct Node \*v) {

if (u->parent == NULL)

\*root = v;

else if (u == u->parent->left)

u->parent->left = v;

else

u->parent->right = v;

if (v != NULL)

v->parent = u->parent;

}

// Find the minimum node in a Red-Black Tree

struct Node \*treeMinimum(struct Node \*x) {

while (x->left != NULL)

x = x->left;

return x;

}

// Fix the Red-Black Tree properties after deletion

void deleteFixup(struct Node \*\*root, struct Node \*x) {

while (x != NULL && x != \*root && x->color == 'B') {

if (x == x->parent->left) {

struct Node \*w = x->parent->right;

if (w != NULL && w->color == 'R') {

w->color = 'B';

x->parent->color = 'R';

leftRotate(root, x->parent);

w = x->parent->right;

}

if (w != NULL && w->left != NULL && w->left->color == 'B' && w->right != NULL && w->right->color == 'B') {

w->color = 'R';

x = x->parent;

} else {

if (w != NULL && w->right != NULL && w->right->color == 'B') {

if (w->left != NULL)

w->left->color = 'B';

w->color = 'R';

rightRotate(root, w);

w = x->parent->right;

}

if (w != NULL)

w->color = x->parent->color;

x->parent->color = 'B';

if (w != NULL && w->right != NULL)

w->right->color = 'B';

leftRotate(root, x->parent);

x = \*root;

}

} else {

struct Node \*w = x->parent->left;

if (w != NULL && w->color == 'R') {

w->color = 'B';

x->parent->color = 'R';

rightRotate(root, x->parent);

w = x->parent->left;

}

if (w != NULL && w->right != NULL && w->right->color == 'B' && w->left != NULL && w->left->color == 'B') {

w->color = 'R';

x = x->parent;

} else {

if (w != NULL && w->left != NULL && w->left->color == 'B') {

if (w->right != NULL)

w->right->color = 'B';

w->color = 'R';

leftRotate(root, w);

w = x->parent->left;

}

if (w != NULL)

w->color = x->parent->color;

x->parent->color = 'B';

if (w != NULL && w->left != NULL)

w->left->color = 'B';

rightRotate(root, x->parent);

x = \*root;

}

}

}

if (x != NULL)

x->color = 'B';

}

// Delete a node from the Red-Black Tree

void deleteNode(struct Node \*\*root, int data) {

struct Node \*z = \*root;

while (z != NULL) {

if (data < z->data)

z = z->left;

else if (data > z->data)

z = z->right;

else

break; // Node with data found

}

if (z == NULL) {

printf("Data %d not found in the tree.\n", data);

return;

}

struct Node \*y = z;

char yOriginalColor = y->color;

struct Node \*x;

if (z->left == NULL) {

x = z->right;

transplant(root, z, z->right);

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

x = z->left;

transplant(root, z, z->left);

} else {

y = treeMinimum(z->right);

yOriginalColor = y->color;

x = y->right;

if (y->parent == z)

x->parent = y;

else {

transplant(root, y, y->right);

y->right = z->right;

y->right->parent = y;

}

transplant(root, z, y);

y->left = z->left;

y->left->parent = y;

y->color = z->color;

}

free(z);

if (yOriginalColor == 'B')

deleteFixup(root, x);

}

// Perform in-order traversal of the Red-Black Tree

void inOrderTraversal(struct Node \*root) {

if (root != NULL) {

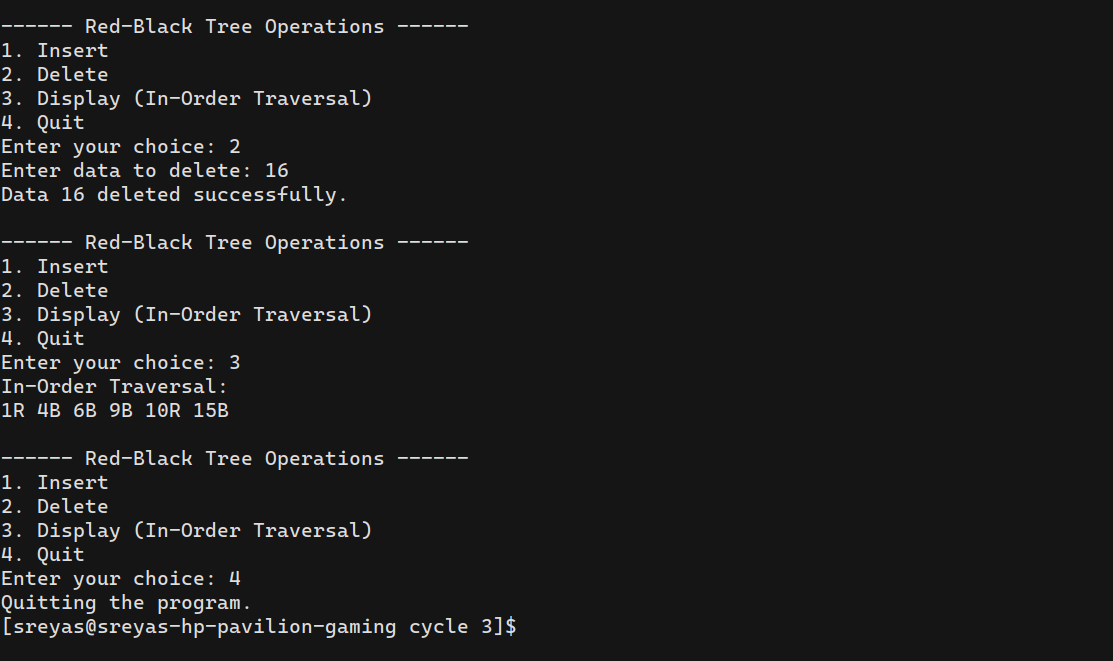
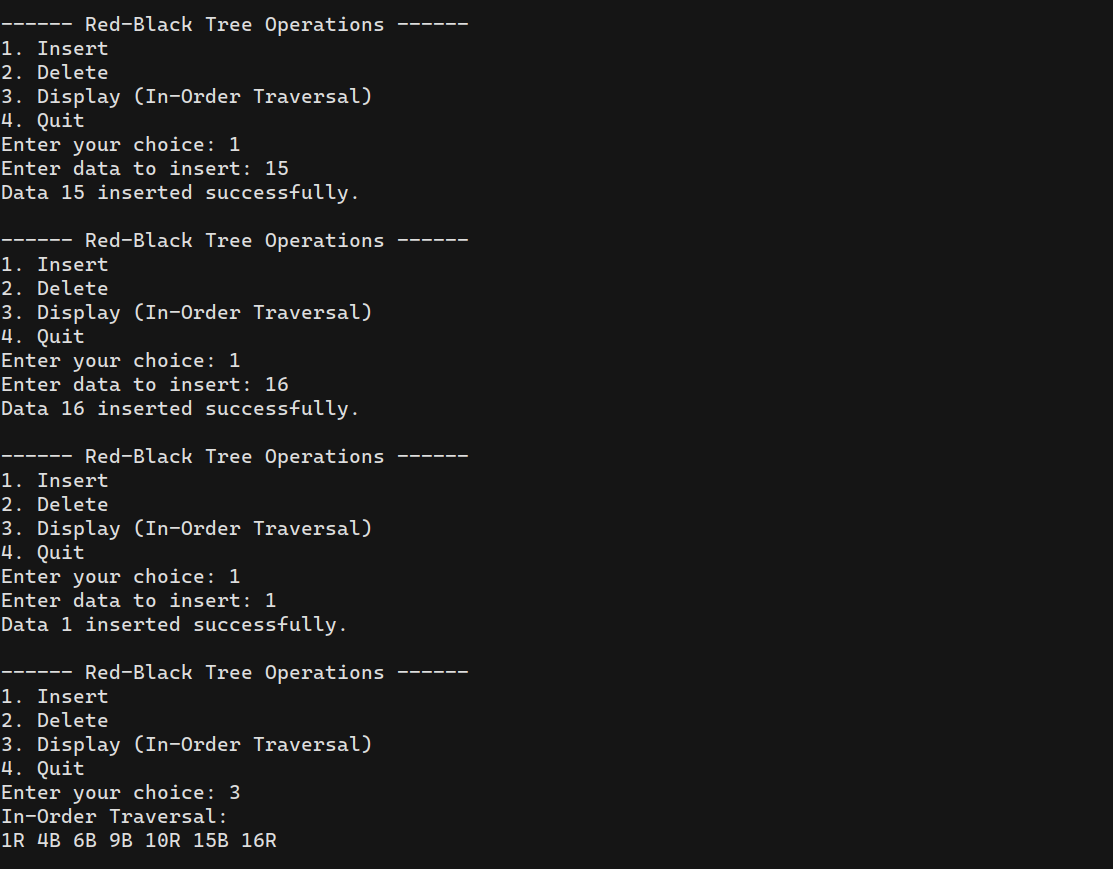
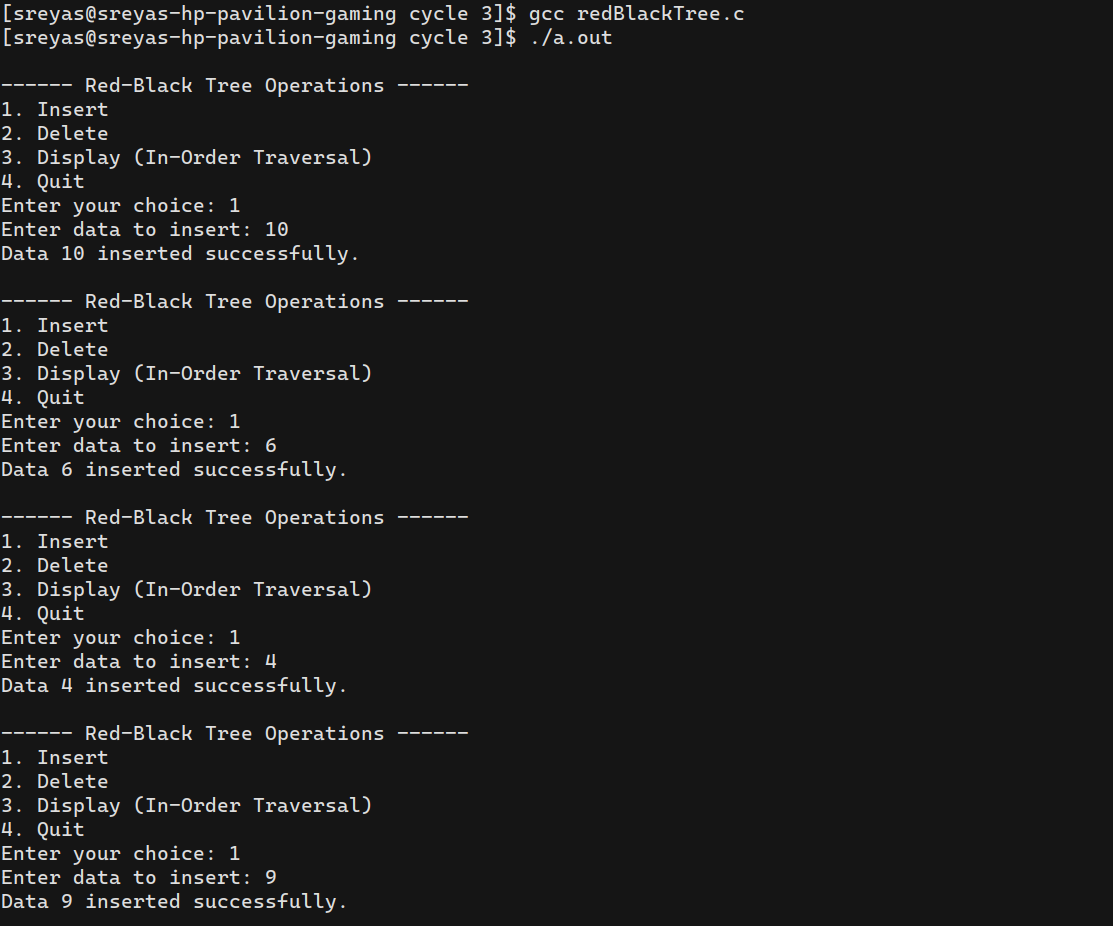
inOrderTraversal(root->left);

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

inOrderTraversal(root->right);

}

}

**Output:**

1. **Graph Traversal techniques (DFS and BFS) and Topological Sorting**

**Program:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_VERTICES 100

struct Queue {

int front, rear, capacity;

int\* array;

};

struct Stack {

int top;

int capacity;

int\* array;

};

struct Queue\* createQueue(int capacity) {

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

queue->capacity = capacity;

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

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

return queue;

}

struct Stack\* createStack(int capacity) {

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

stack->capacity = capacity;

stack->top = -1;

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

return stack;

}

int isEmpty(struct Queue\* queue) {

return queue->front == -1;

}

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

if (queue->rear == queue->capacity - 1) {

printf("Queue overflow\n");

return;

}

if (queue->front == -1)

queue->front = 0;

queue->rear++;

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

}

int dequeue(struct Queue\* queue) {

if (isEmpty(queue)) {

printf("Queue underflow\n");

return -1;

}

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

queue->front++;

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

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

return item;

}

int isEmptyStack(struct Stack\* stack) {

return stack->top == -1;

}

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

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

printf("Stack overflow\n");

return;

}

stack->array[++stack->top] = item;

}

int pop(struct Stack\* stack) {

if (isEmptyStack(stack)) {

printf("Stack underflow\n");

return -1;

}

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

}

struct Graph {

int vertices;

int\*\* adjacencyMatrix;

};

struct Graph\* createGraph(int vertices) {

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

graph->vertices = vertices;

// Allocating memory for the adjacency matrix

graph->adjacencyMatrix = (int\*\*)malloc(vertices \* sizeof(int\*));

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

graph->adjacencyMatrix[i] = (int\*)malloc(vertices \* sizeof(int));

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

graph->adjacencyMatrix[i][j] = 0; // Initializing with 0

}

}

return graph;

}

void addEdge(struct Graph\* graph, int source, int destination) {

graph->adjacencyMatrix[source][destination] = 1;

}

void displayMatrix(struct Graph\* graph) {

printf("Adjacency Matrix:\n");

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

for (int j = 0; j < graph->vertices; j++) {

printf("%d ", graph->adjacencyMatrix[i][j]);

}

printf("\n");

}

}

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

printf("%d ", vertex);

visited[vertex] = 1;

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

if (graph->adjacencyMatrix[vertex][i] == 1 && !visited[i]) {

DFSUtil(graph, i, visited);

}

}

}

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

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

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

visited[i] = 0;

}

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

DFSUtil(graph, startVertex, visited);

printf("\n");

free(visited);

}

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

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

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

visited[i] = 0;

}

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

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

visited[startVertex] = 1;

enqueue(queue, startVertex);

while (!isEmpty(queue)) {

int vertex = dequeue(queue);

printf("%d ", vertex);

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

if (graph->adjacencyMatrix[vertex][i] == 1 && !visited[i]) {

visited[i] = 1;

enqueue(queue, i);

}

}

}

printf("\n");

free(visited);

free(queue->array);

free(queue);

}

void topologicalSortUtil(struct Graph\* graph, int vertex, int\* visited, struct Stack\* stack) {

visited[vertex] = 1;

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

if (graph->adjacencyMatrix[vertex][i] == 1 && !visited[i]) {

topologicalSortUtil(graph, i, visited, stack);

}

}

push(stack, vertex);

}

void topologicalSort(struct Graph\* graph) {

struct Stack\* stack = createStack(graph->vertices);

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

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

visited[i] = 0;

}

printf("Topological Sorting: ");

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

if (!visited[i]) {

topologicalSortUtil(graph, i, visited, stack);

}

}

while (!isEmptyStack(stack)) {

printf("%d ", pop(stack));

}

printf("\n");

free(visited);

free(stack->array);

free(stack);

}

int main() {

int choice, vertices, source, destination, startVertex;

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

scanf("%d", &vertices);

struct Graph\* graph = createGraph(vertices);

do {

printf("\n------ Graph Traversal Techniques ------\n");

printf("1. Add Edge\n");

printf("2. Display Adjacency Matrix\n");

printf("3. DFS Traversal\n");

printf("4. BFS Traversal\n");

printf("5. Topological Sorting\n");

printf("6. Quit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter source and destination vertices for the edge: ");

scanf("%d %d", &source, &destination);

addEdge(graph, source, destination);

printf("Edge added between %d and %d.\n", source, destination);

break;

case 2:

displayMatrix(graph);

break;

case 3:

printf("Enter the starting vertex for DFS: ");

scanf("%d", &startVertex);

DFS(graph, startVertex);

break;

case 4:

printf("Enter the starting vertex for BFS: ");

scanf("%d", &startVertex);

BFS(graph, startVertex);

break;

case 5:

topologicalSort(graph);

break;

case 6:

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

break;

default:

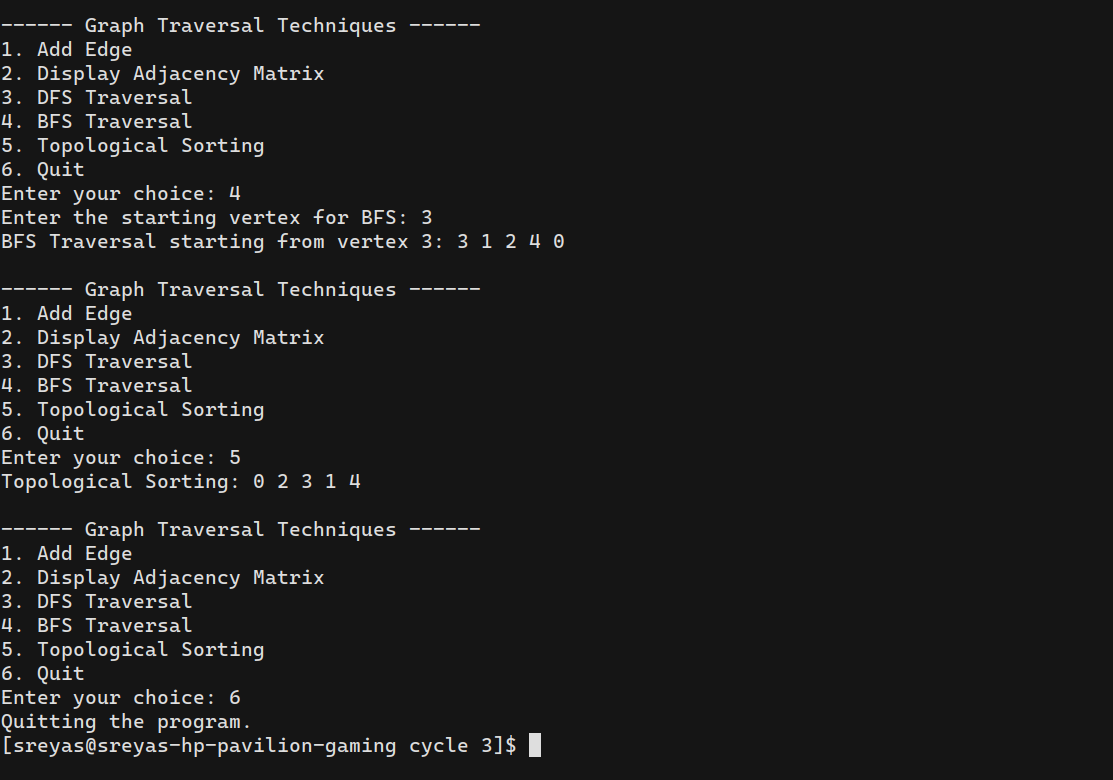
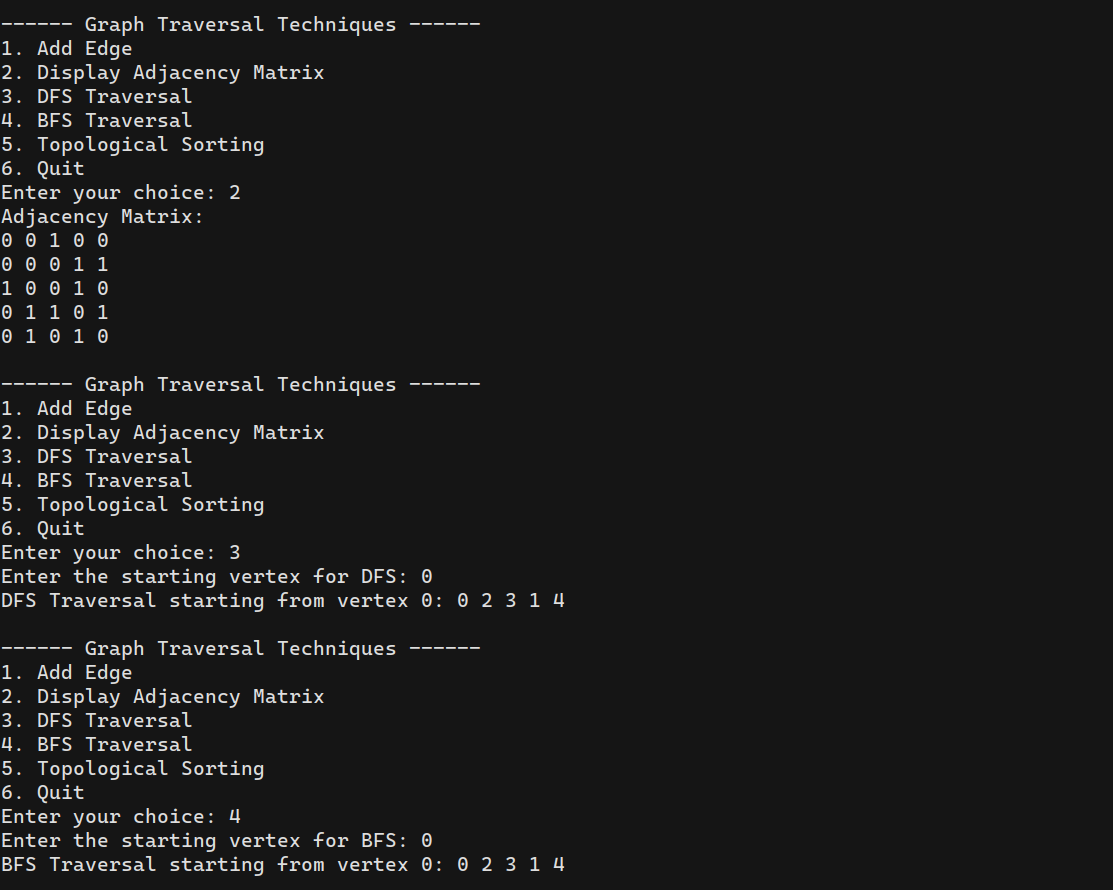
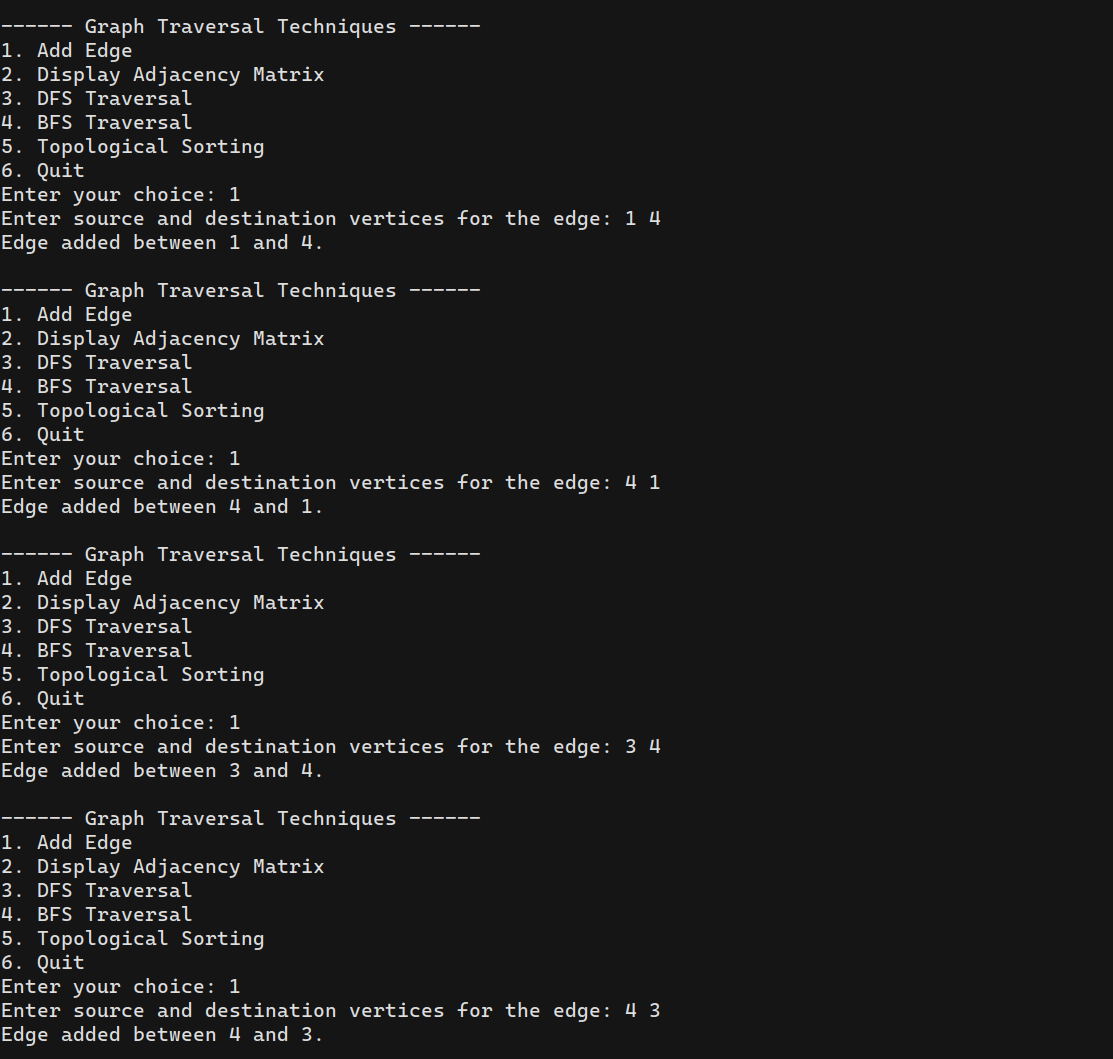
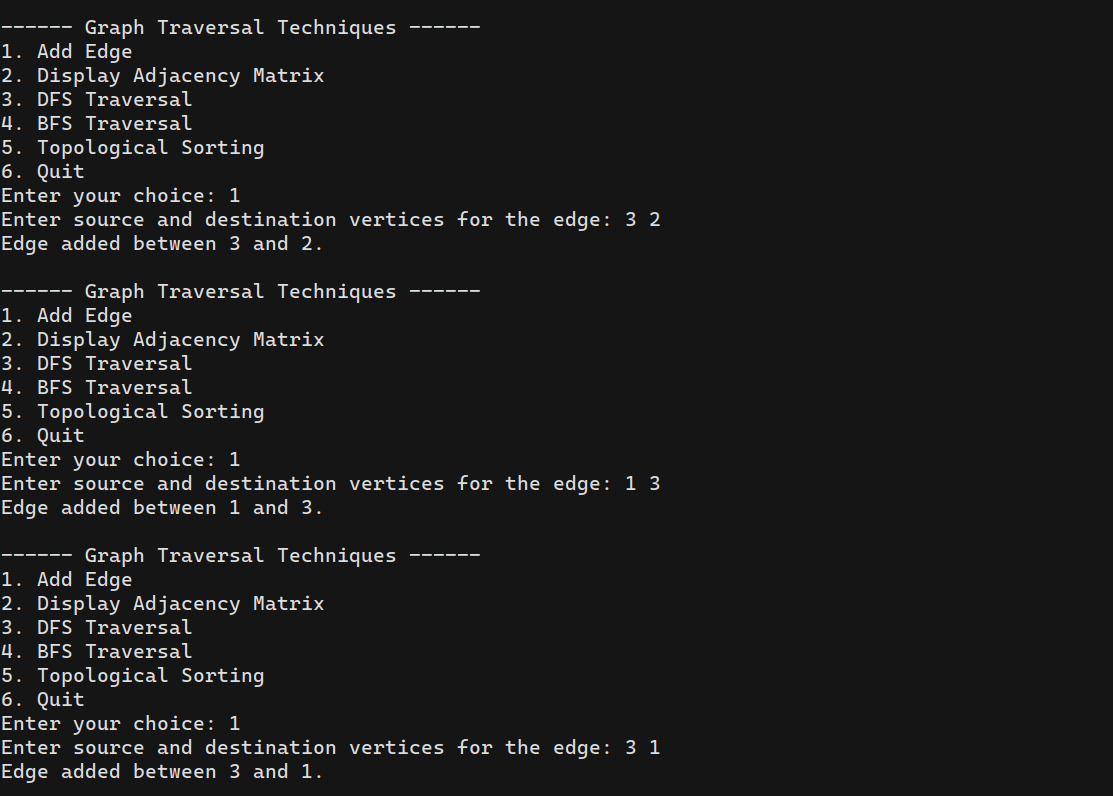
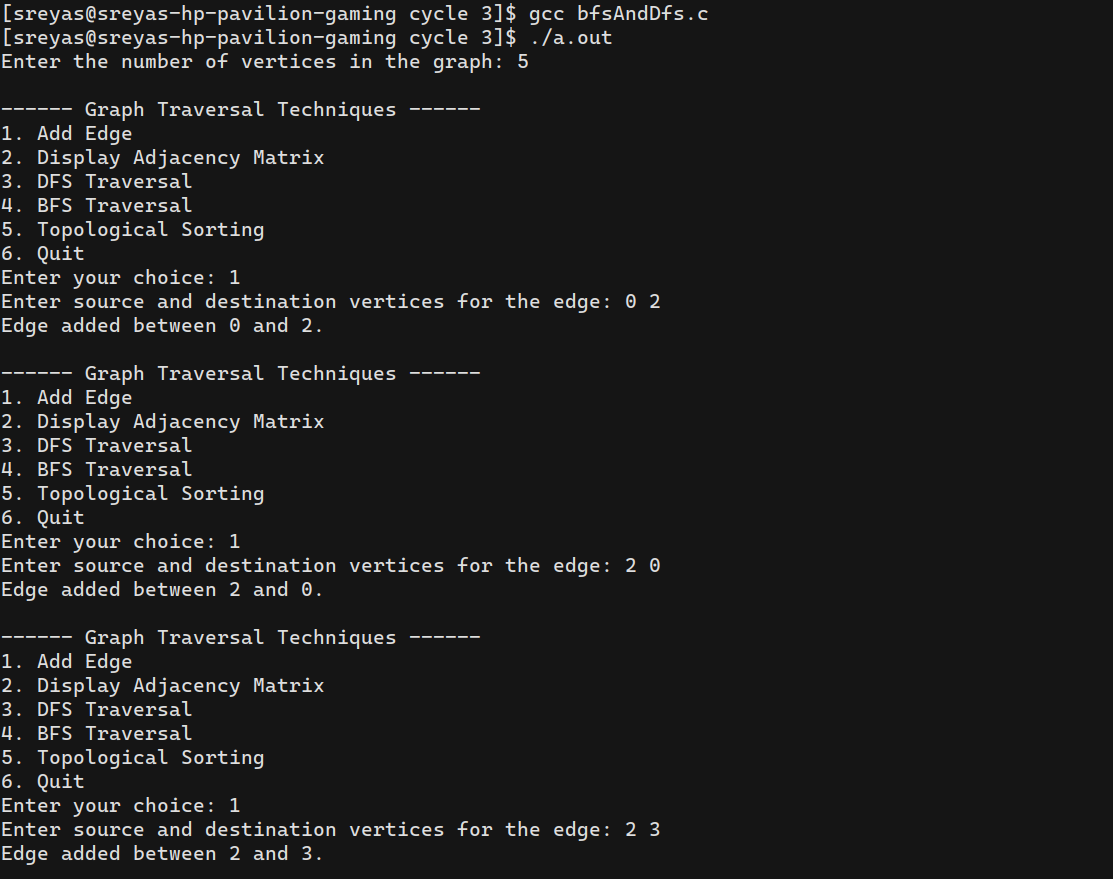
printf("Invalid choice. Please enter a valid option.\n");

}

} while (choice != 6);

return 0;

}

**Output:**

1. **Finding the Strongly connected Components in a directed graph**

**Program:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_VERTICES 100

struct Stack {

int items[MAX\_VERTICES];

int top;

};

struct Graph {

int vertices;

int\*\* adjacencyMatrix;

};

// Function prototypes

struct Stack\* createStack();

void push(struct Stack\* stack, int item);

int pop(struct Stack\* stack);

void dfs(struct Graph\* graph, int vertex, int\* visited, struct Stack\* stack);

struct Graph\* transposeGraph(struct Graph\* graph);

void printSCCs(struct Graph\* graph);

int main() {

struct Graph\* graph = NULL;

int choice, vertices, edges, i, j, src, dest;

do {

printf("\nMenu:\n");

printf("1. Create Graph\n");

printf("2. Find Strongly Connected Components\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

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

scanf("%d", &vertices);

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

graph->vertices = vertices;

graph->adjacencyMatrix = (int\*\*)malloc(vertices \* sizeof(int\*));

for (i = 0; i < vertices; i++) {

graph->adjacencyMatrix[i] = (int\*)malloc(vertices \* sizeof(int));

for (j = 0; j < vertices; j++) {

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

}

}

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

scanf("%d", &edges);

printf("Enter the edges (src dest):\n");

for (i = 0; i < edges; i++) {

scanf("%d %d", &src, &dest);

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

}

break;

case 2:

if (graph == NULL) {

printf("Graph not created. Please create a graph first.\n");

break;

}

printSCCs(graph);

break;

case 3:

printf("Exiting program.\n");

break;

default:

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

}

} while (choice != 3);

return 0;

}

struct Stack\* createStack() {

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

stack->top = -1;

return stack;

}

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

stack->items[++stack->top] = item;

}

int pop(struct Stack\* stack) {

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

return -1; // Empty stack

}

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

}

void dfs(struct Graph\* graph, int vertex, int\* visited, struct Stack\* stack) {

visited[vertex] = 1;

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

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

dfs(graph, i, visited, stack);

}

}

push(stack, vertex);

}

struct Graph\* transposeGraph(struct Graph\* graph) {

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

transposedGraph->vertices = graph->vertices;

transposedGraph->adjacencyMatrix = (int\*\*)malloc(graph->vertices \* sizeof(int\*));

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

transposedGraph->adjacencyMatrix[i] = (int\*)malloc(graph->vertices \* sizeof(int));

for (int j = 0; j < graph->vertices; j++) {

transposedGraph->adjacencyMatrix[i][j] = graph->adjacencyMatrix[j][i];

}

}

return transposedGraph;

}

void printSCCs(struct Graph\* graph) {

struct Stack\* stack = createStack();

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

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

visited[i] = 0;

}

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

if (!visited[i]) {

dfs(graph, i, visited, stack);

}

}

struct Graph\* transposedGraph = transposeGraph(graph);

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

visited[i] = 0;

}

printf("Strongly Connected Components:\n");

while (stack->top != -1) {

int vertex = pop(stack);

if (!visited[vertex]) {

struct Stack\* sccStack = createStack();

dfs(transposedGraph, vertex, visited, sccStack);

printf("{ ");

while (sccStack->top != -1) {

int sccVertex = pop(sccStack);

printf("%d ", sccVertex);

}

printf("}\n");

free(sccStack);

}

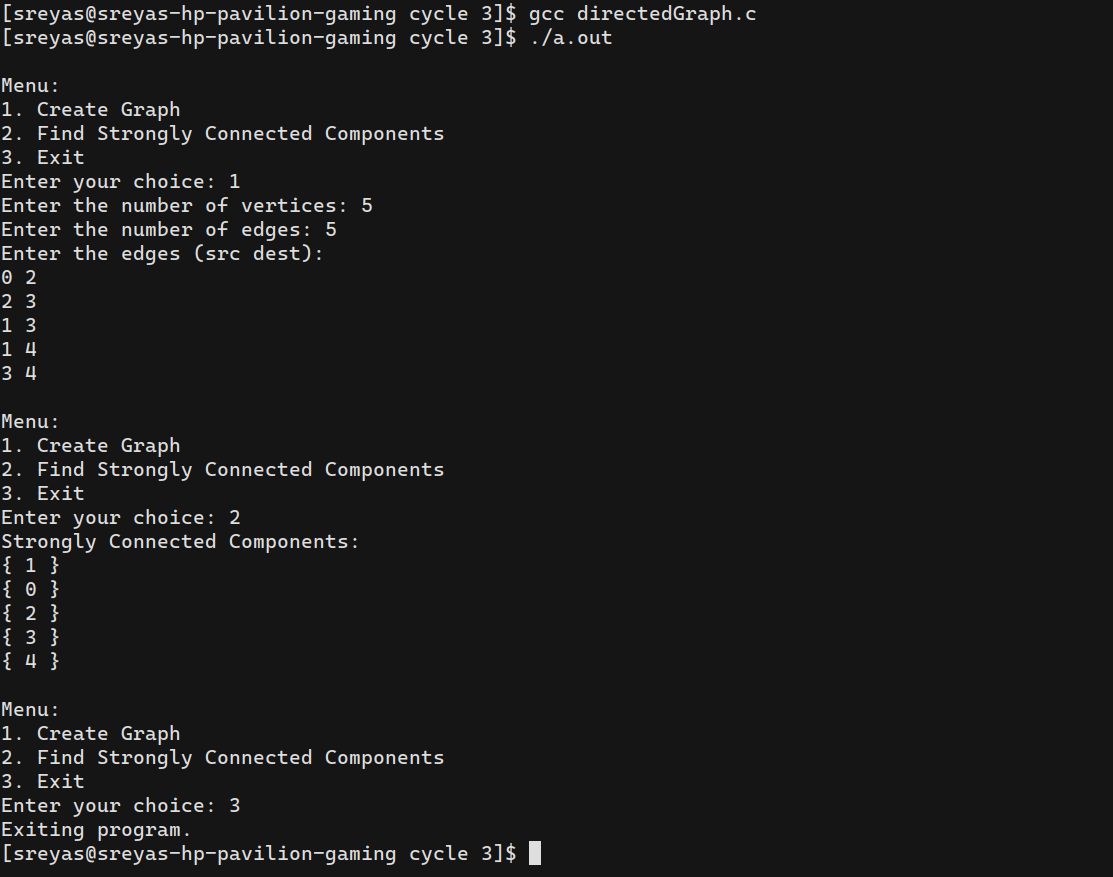
}

free(stack);

free(visited);

free(transposedGraph);

}

**Output:**

1. **Prim’s Algorithm for finding the minimum cost spanning tree**

**Program:**

#include <limits.h>

#include <stdbool.h>

#include <stdio.h>

#define MAX\_VERTICES 10

int minKey(int key[], bool mstSet[], int vertices) {

int min = INT\_MAX, min\_index;

for (int v = 0; v < vertices; v++)

if (mstSet[v] == false && key[v] < min)

min = key[v], min\_index = v;

return min\_index;

}

int printMST(int parent[], int graph[MAX\_VERTICES][MAX\_VERTICES], int vertices) {

printf("Edge \tWeight\n");

for (int i = 1; i < vertices; i++)

printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);

}

void primMST(int graph[MAX\_VERTICES][MAX\_VERTICES], int vertices) {

int parent[MAX\_VERTICES];

int key[MAX\_VERTICES];

bool mstSet[MAX\_VERTICES];

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

key[i] = INT\_MAX, mstSet[i] = false;

key[0] = 0;

parent[0] = -1;

for (int count = 0; count < vertices - 1; count++) {

int u = minKey(key, mstSet, vertices);

mstSet[u] = true;

for (int v = 0; v < vertices; v++)

if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])

parent[v] = u, key[v] = graph[u][v];

}

printMST(parent, graph, vertices);

}

int main() {

int vertices;

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

scanf("%d", &vertices);

if (vertices > MAX\_VERTICES || vertices <= 0) {

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

return 1;

}

int graph[MAX\_VERTICES][MAX\_VERTICES];

printf("Enter the adjacency matrix (size %dx%d, 0 for no edge):\n", vertices, vertices);

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

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

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

}

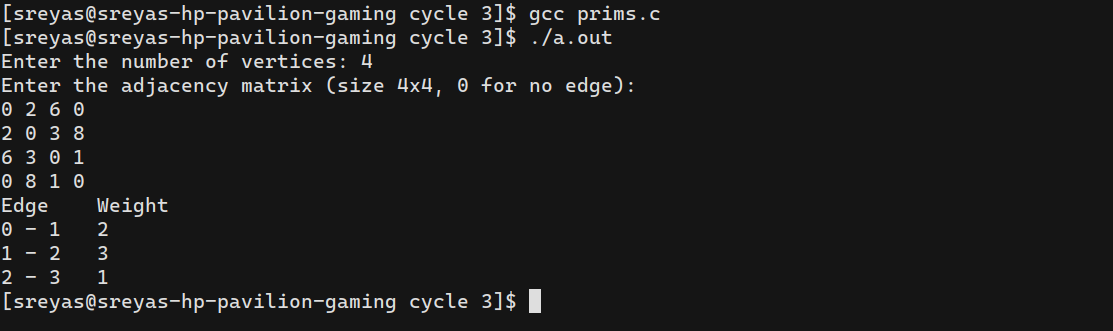
}

primMST(graph, vertices);

return 0;

}

**Output:**

****

1. **Kruskal’s algorithm using the Disjoint set data structure**

**Program:**

#include <stdio.h>

#include <stdlib.h>

int comparator(const void\* p1, const void\* p2) {

const int(\*x)[3] = p1;

const int(\*y)[3] = p2;

return (\*x)[2] - (\*y)[2];

}

void makeSet(int parent[], int rank[], int n) {

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

parent[i] = i;

rank[i] = 0;

}

}

int findParent(int parent[], int component) {

if (parent[component] == component)

return component;

return parent[component] = findParent(parent, parent[component]);

}

void unionSet(int u, int v, int parent[], int rank[], int n) {

u = findParent(parent, u);

v = findParent(parent, v);

if (rank[u] < rank[v]) {

parent[u] = v;

} else if (rank[u] > rank[v]) {

parent[v] = u;

} else {

parent[v] = u;

rank[u]++;

}

}

void kruskalAlgo(int n, int edge[][3]) {

qsort(edge, n, sizeof(edge[0]), comparator);

int parent[n];

int rank[n];

makeSet(parent, rank, n);

int minCost = 0;

printf("Following are the edges in the constructed MST\n");

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

int v1 = findParent(parent, edge[i][0]);

int v2 = findParent(parent, edge[i][1]);

int wt = edge[i][2];

if (v1 != v2) {

unionSet(v1, v2, parent, rank, n);

minCost += wt;

printf("%d -- %d == %d\n", edge[i][0], edge[i][1], wt);

}

}

printf("Minimum Cost Spanning Tree: %d\n", minCost);

}

int main() {

int vertices, edges;

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

scanf("%d", &vertices);

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

scanf("%d", &edges);

int edge[edges][3];

printf("Enter the edges (src dest weight):\n");

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

scanf("%d %d %d", &edge[i][0], &edge[i][1], &edge[i][2]);

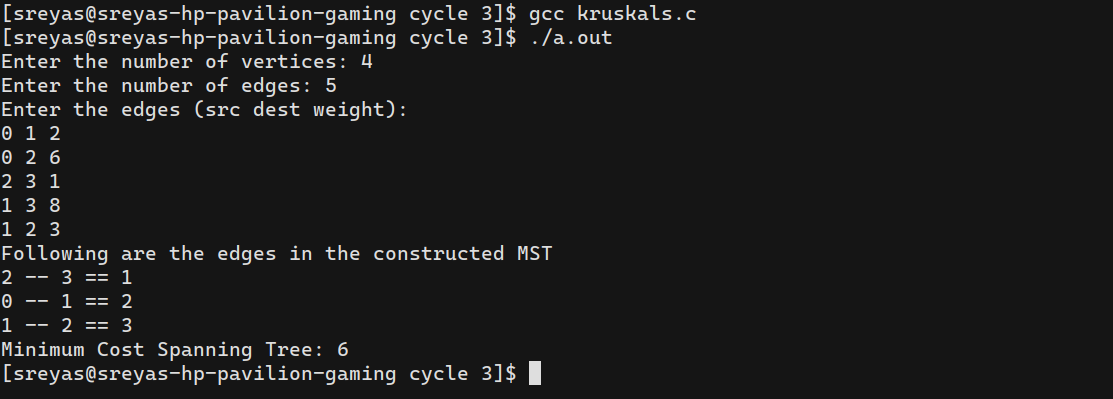
}

kruskalAlgo(edges, edge);

return 0;

}

**Output:**

****

1. **Single Source shortest path algorithm using any heap structure that supports mergeable heap operations**

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

// Structure to represent a node in the graph

struct Node {

int vertex;

int weight;

struct Node\* next;

};

// Structure to represent the graph

struct Graph {

int V; // Number of vertices

struct Node\*\* adjList; // Adjacency list

};

// Structure to represent a node in the heap

struct HeapNode {

int vertex;

int distance;

};

// Structure to represent the heap

struct BinaryHeap {

struct HeapNode\* array;

int capacity;

int size;

};

// Function to create a new graph

struct Graph\* createGraph(int V) {

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

graph->V = V;

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

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

graph->adjList[i] = NULL;

}

return graph;

}

// Function to add an edge to the graph

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

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

newNode->vertex = dest;

newNode->weight = weight;

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

graph->adjList[src] = newNode;

}

// Function to create a new heap node

struct HeapNode\* createHeapNode(int vertex, int distance) {

struct HeapNode\* node = (struct HeapNode\*)malloc(sizeof(struct HeapNode));

node->vertex = vertex;

node->distance = distance;

return node;

}

// Function to create a new binary heap

struct BinaryHeap\* createBinaryHeap(int capacity) {

struct BinaryHeap\* heap = (struct BinaryHeap\*)malloc(sizeof(struct BinaryHeap));

heap->array = (struct HeapNode\*)malloc(capacity \* sizeof(struct HeapNode));

heap->capacity = capacity;

heap->size = 0;

return heap;

}

// Function to swap two heap nodes

void swapHeapNodes(struct HeapNode\* a, struct HeapNode\* b) {

struct HeapNode temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to heapify a subtree rooted with the given index

void heapify(struct BinaryHeap\* heap, int index) {

int smallest = index;

int left = 2 \* index + 1;

int right = 2 \* index + 2;

if (left < heap->size && heap->array[left].distance < heap->array[smallest].distance) {

smallest = left;

}

if (right < heap->size && heap->array[right].distance < heap->array[smallest].distance) {

smallest = right;

}

if (smallest != index) {

swapHeapNodes(&heap->array[index], &heap->array[smallest]);

heapify(heap, smallest);

}

}

// Function to extract the minimum node from the heap

struct HeapNode extractMin(struct BinaryHeap\* heap) {

if (heap->size == 1) {

heap->size--;

return heap->array[0];

}

struct HeapNode root = heap->array[0];

heap->array[0] = heap->array[heap->size - 1];

heap->size--;

heapify(heap, 0);

return root;

}

// Function to decrease the distance value of a given vertex

void decreaseKey(struct BinaryHeap\* heap, int vertex, int newDistance) {

int i;

for (i = 0; i < heap->size; i++) {

if (heap->array[i].vertex == vertex) {

break;

}

}

heap->array[i].distance = newDistance;

// Fix the min heap property if it is violated

while (i > 0 && heap->array[i].distance < heap->array[(i - 1) / 2].distance) {

swapHeapNodes(&heap->array[i], &heap->array[(i - 1) / 2]);

i = (i - 1) / 2;

}

}

// Function to check if a vertex is in the heap

int isInHeap(struct BinaryHeap\* heap, int vertex) {

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

if (heap->array[i].vertex == vertex) {

return 1;

}

}

return 0;

}

// Function to print the distance values of vertices

void printDistances(int\* dist, int n) {

printf("Shortest distances from source:\n");

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

printf("To vertex %d: %d\n", i, dist[i]);

}

}

// Function to perform Dijkstra's algorithm using a binary heap

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

int V = graph->V;

int\* dist = (int\*)malloc(V \* sizeof(int));

struct BinaryHeap\* heap = createBinaryHeap(V);

// Initialize distances and heap

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

dist[i] = INT\_MAX;

heap->array[i] = \*createHeapNode(i, dist[i]);

heap->size++;

}

// Set the distance of the start vertex to 0

dist[startVertex] = 0;

decreaseKey(heap, startVertex, 0);

// Dijkstra's algorithm

while (heap->size > 0) {

struct HeapNode minNode = extractMin(heap);

int u = minNode.vertex;

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

while (temp != NULL) {

int v = temp->vertex;

int weight = temp->weight;

if (isInHeap(heap, v) && dist[u] != INT\_MAX && dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

decreaseKey(heap, v, dist[v]);

}

temp = temp->next;

}

}

// Print the shortest distances

printDistances(dist, V);

free(dist);

free(heap->array);

free(heap);

}

int main() {

int V, E;

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

scanf("%d", &V);

struct Graph\* graph = createGraph(V);

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

scanf("%d", &E);

printf("Enter edges in the format (src dest weight):\n");

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

int src, dest, weight;

scanf("%d %d %d", &src, &dest, &weight);

addEdge(graph, src, dest, weight);

}

int startVertex;

printf("Enter the starting vertex: ");

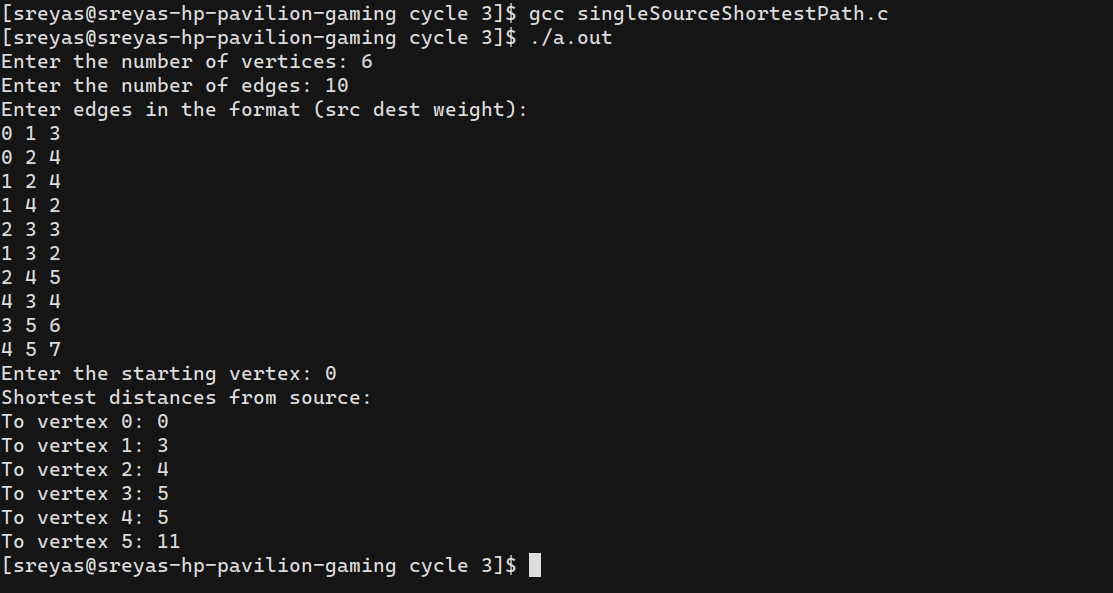
scanf("%d", &startVertex);

dijkstra(graph, startVertex);

return 0;

}

**Output:**

****