**RUSTAMJI INSTITUTE OF TECHNOLOGY**

**BSF ACADEMY, TEKANPUR**

**Practical File for**

**CS303 (Data Structure)**



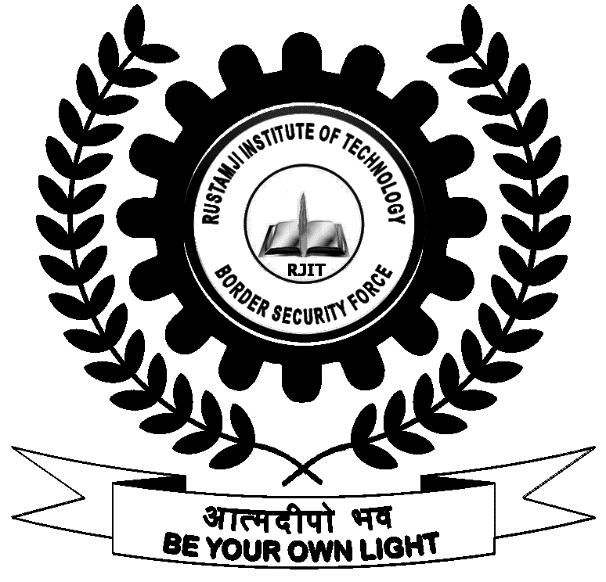
**Submitted by**

*Richa Jadon (0902CS231080)*

B.Tech. Computer Science & Engineering 3rd Semester

(2023-2027 batch)

|  |  |
| --- | --- |
| **Subject Teacher**  Dr. Jagdish Makhijani | **File Checked by**  Mr. Yashwant Pathak |



**Self-Declaration Certificate**

I, **Richa Jadon**, hereby declare that I have completed the lab work of CS303 (Data Structure) at my own effort and understanding.

I affirm that the work submitted is my own, and I take full responsibility for its authenticity and originality.

Date:

Richa Jadon

0902CS231080

**ENVIRONMENT USED**

**Hardware Configuration :** APTOP-VQ422Q29

Processor AMD Ryzen 5 5500U with Radeon Graphics 2.10 GHz

Installed RAM 8.00 GB (5.85 GB usable)

Device ID 4461EF90-C0A4-41DC-ADAF-313702329DD1

Product ID 00356-24701-39296-AAOEM

System type 64-bit operating system, x64-based processor

**C Compiler :** GCC Compiler

**User Interface :** VS code

**GROUP MEMBERS**

**Member-1 :** Sakshi Bhadoriya 0902cs231091

https://github.com/sakshyy14/DSA-

**Member-2 :** Sanjana 0902cs231091

https://github.com/SANJANA-cse/DSA

**Member-3 :** Richa Jadon 0902cs231080

https://github.com/richaaa14/Dsa

**TABLE OF CONTENTS**

**Section-A (Linked List)**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Practical Description** | **Page Nos.** | **COs** |
| 1 | Implementation of Linked List using array. | 1-3 | CO-1 |
| 2 | Implementation of Linked List using Pointers. | 4-7 | CO-1 |
| 3 | Implementation of Doubly Linked List using Pointers. | 8-12 | CO-1 |
| 4 | Implementation of Circular Single Linked List using Pointers. | 13-16 | CO-1 |
| 5 | Implementation of Circular Doubly Linked List using Pointers. | 17-21 | CO-1 |

**Section-B (Stack)**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Practical Description** | **Page Nos.** | **COs** |
| 1 | Implementation of Stack using Array. | 22-24 | CO-2 |
| 2 | Implementation of Stack using Pointers. | 25-27 | CO-2 |
| 3 | Program for Tower of Hanoi using recursion. | 28-29 | CO-2 |
| 4 | Program to find out factorial of given number using recursion. Also show the various states of stack using in this program. | 30-31 | CO-2 |

**Section-C (Queue)**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Practical Description** | **Page Nos.** | **COs** |
| 1 | Implementation of Queue using Array. | 32-35 | CO-2 |
| 2 | Implementation of Queue using Pointers. | 36-39 | CO-2 |
| 3 | Implementation of Circular Queue using Array. | 40-43 | CO-2 |

**Section-D (Trees & Graphs)**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Practical Description** | **Page Nos.** | **COs** |
| 1 | Implementation of Binary Search Tree. | 44-45 | CO-3 |
| 2 | Conversion of BST PreOrder/PostOrder/InOrder. | 46-47 | CO-3 |
| 3 | Implementation of Kruskal Algorithm | 48-50 | CO-4 |
| 4 | Implementation of Prim Algorithm | 51-52 | CO-4 |
| 5 | Implementation of Dijkstra Algorithm | 53-55 | CO-4 |

**Section-E (Sorting & Searching)**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Practical Description** | **Page Nos.** | **COs** |
| 1 | Implementation of Sorting  a. Bubble  b. Selection  c. Insertion  d. Quick  e. Merge | 56-60 | CO-5 |
| 2 | Implementation of Binary Search on a list of numbers stored in an Array | 61-63 | CO-5 |
| 3 | Implementation of Binary Search on a list of strings stored in an Array | 64-65 | CO-5 |
| 4 | Implementation of Linear Search on a list of strings stored in an Array  OR  Implementation of Binary Search on a list of strings stored in a Single Linked List | 66-67 | CO-5 |

**Section-A (Linked List)**

**Experiment No.: 1**

**Program Description:**

Implementation of Linked List using Arrays.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_NODES 10

typedef struct {

int data;

int next;

} Node;

typedef struct {

Node nodes[MAX\_NODES];

int head;

int free;

} LinkedList;

void initList(LinkedList\* list) {

list->head = -1;

list->free = 0;

for (int i = 0; i < MAX\_NODES - 1; i++) {

list->nodes[i].next = i + 1;

}

list->nodes[MAX\_NODES - 1].next = -1;

}

void insertNode(LinkedList\* list, int value) {

if (list->free == -1) {

printf("No space left in the list.\n");

return;

}

int newNodeIndex = list->free;

list->free = list->nodes[newNodeIndex].next;

list->nodes[newNodeIndex].data = value;

list->nodes[newNodeIndex].next = -1;

if (list->head == -1) {

list->head = newNodeIndex;

}

else

{

int lastNodeIndex = list->head;

while (list->nodes[lastNodeIndex].next != -1) {

lastNodeIndex = list->nodes[lastNodeIndex].next;

}

list->nodes[lastNodeIndex].next = newNodeIndex;

}

}

void printList(LinkedList\* list) {

int current = list->head;

while (current != -1) {

printf("%d -> ", list->nodes[current].data);

current = list->nodes[current].next;

}

printf("NULL\n");

}

int main() {

LinkedList list;

initList(&list);

insertNode(&list, 10);

insertNode(&list, 20);

insertNode(&list, 30);

insertNode(&list, 40);

printList(&list);

return 0;

}

**Output:**

10 -> 20 -> 30 -> 40 -> NULL

**Experiment No.: 2**

**Program Description:**

Implementation of Linked List using Pointers.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node

{

int data;

struct Node\* next;

} Node;

Node\* createNode(int value)

{

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

if (newNode == NULL)

{

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

return NULL;

}

newNode->data = value;

newNode->next = NULL;

return newNode;

}

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

{

Node\* newNode = createNode(value);

if (\*head == NULL)

{

\*head = newNode;

return;

}

Node\* temp = \*head;

while (temp->next != NULL)

{

temp = temp->next;

}

temp->next = newNode;

}

void printList(Node\* head)

{

Node\* temp = head;

while (temp != NULL)

{

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

temp = temp->next;

}

printf("NULL\n");

}

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

{

if (\*head == NULL)

{

printf("List is empty!\n");

return;

}

Node\* temp = \*head;

if (temp->data == value)

{

\*head = temp->next;

free(temp);

return;

}

Node\* prev = NULL;

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

{

prev = temp;

temp = temp->next;

}

if (temp == NULL)

{

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

return;

}

prev->next = temp->next;

free(temp);

}

void freeList(Node\* head)

{

Node\* temp;

while (head != NULL)

{

temp = head;

head = head->next;

free(temp);

}

}

int main()

{

Node\* head = NULL;

insertNode(&head, 10);

insertNode(&head, 20);

insertNode(&head, 30);

insertNode(&head, 40);

printf("Linked List: ");

printList(head);

deleteNode(&head, 20);

printf("After deleting 20: ");

printList(head);

deleteNode(&head, 10);

printf("After deleting 10: ");

printList(head);

freeList(head);

return 0;

}

**Output:**

Linked List: 10 -> 20 -> 30 -> 40 -> NULL

After deleting 20: 10 -> 30 -> 40 -> NULL

After deleting 10: 30 -> 40 -> NULL

**Experiment No.: 3**

**Program Description:**

Implementation of Doubly Linked List using Pointers.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node

{

int data;

struct Node\* next;

struct Node\* prev;

} Node;

Node\* createNode(int value)

{

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

if (newNode == NULL)

{

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

return NULL;

}

newNode->data = value;

newNode->next = NULL;

newNode->prev = NULL;

return newNode;

}

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

{

Node\* newNode = createNode(value);

if (\*head == NULL)

{

\*head = newNode;

return;

}

Node\* temp = \*head;

while (temp->next != NULL)

{

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

void printListFromHead(Node\* head)

{

Node\* temp = head;

while (temp != NULL)

{

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

temp = temp->next;

}

printf("NULL\n");

}

void printListFromTail(Node\* head)

{

Node\* temp = head;

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

{

temp = temp->next;

}

while (temp != NULL)

{

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

temp = temp->prev;

}

printf("NULL\n");

}

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

{

if (\*head == NULL)

{

printf("List is empty!\n");

return;

}

Node\* temp = \*head;

if (temp->data == value)

{

\*head = temp->next;

if (\*head != NULL)

{

(\*head)->prev = NULL;

}

free(temp);

return;

}

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

{

temp = temp->next;

}

if (temp == NULL)

{

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

return;

}

if (temp->next != NULL)

{

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

}

if (temp->prev != NULL)

{

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

}

free(temp);

}

void freeList(Node\* head)

{

Node\* temp;

while (head != NULL)

{

temp = head;

head = head->next;

free(temp);

}

}

int main()

{

Node\* head = NULL;

insertNode(&head, 10);

insertNode(&head, 20);

insertNode(&head, 30);

insertNode(&head, 40);

printf("List from head to tail: ");

printListFromHead(head);

printf("List from tail to head: ");

printListFromTail(head);

deleteNode(&head, 20);

printf("After deleting 20:\n");

printListFromHead(head);

deleteNode(&head, 40);

printf("After deleting 40:\n");

printListFromHead(head);

freeList(head);

return 0;

}

**Output:**

List from head to tail: 10 <-> 20 <-> 30 <-> 40 <-> NULL

List from tail to head: 40 <-> 30 <-> 20 <-> 10 <-> NULL

After deleting 20:

List from head to tail: 10 <-> 30 <-> 40 <-> NULL

After deleting 40:

List from head to tail: 10 <-> 30 <-> NULL

**Experiment No.: 4**

**Program Description:**

Implementation of Circular Single Linked List using Pointers.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node\* next;

} Node;

Node\* createNode(int value) {

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

if (newNode == NULL) {

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

return NULL;

}

newNode->data = value;

newNode->next = NULL;

return newNode;

}

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

Node\* newNode = createNode(value);

if (\*head == NULL) {

\*head = newNode;

newNode->next = \*head;

} else {

Node\* temp = \*head;

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

temp = temp->next;

}

temp->next = newNode;

newNode->next = \*head;

}

}

void printList(Node\* head) {

if (head == NULL) {

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

return;

}

Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

printf("(head)\n");

}

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

if (\*head == NULL) {

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

return;

}

Node\* temp = \*head;

Node\* prev = NULL

if (temp->data == value) {

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

free(temp);

\*head = NULL;

return;

}

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

temp = temp->next;

}

Node\* toDelete = \*head;

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

temp->next = \*head;

free(toDelete);

return;

}

prev = temp;

temp = temp->next;

while (temp != \*head && temp->data != value) {

prev = temp;

temp = temp->next;

}

if (temp == \*head) {

printf("Value not found in the list.\n");

return;

}

prev->next = temp->next;

free(temp);

}

void freeList(Node\* head) {

if (head == NULL) return;

Node\* temp = head;

do {

Node\* toDelete = temp;

temp = temp->next;

free(toDelete);

} while (temp != head);

head = NULL;

}

int main() {

Node\* head = NULL;

insertNode(&head, 10);

insertNode(&head, 20);

insertNode(&head, 30);

insertNode(&head, 40);

printf("Circular Linked List: ");

printList(head);

deleteNode(&head, 20);

printf("After deleting 20: ");

printList(head);

deleteNode(&head, 10);

printf("After deleting 10: ");

printList(head);

freeList(head);

return 0;

}

**Output:**

Circular Linked List: 10 -> 20 -> 30 -> 40 -> (head)

After deleting 20: 10 -> 30 -> 40 -> (head)

After deleting 10: 30 -> 40 -> (head)

**Experiment No.: 5**

**Program Description:**

Implementation of Circular Doubly Linked List using Pointers.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node\* next;

struct Node\* prev;

} Node;

Node\* createNode(int value) {

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

if (newNode == NULL) {

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

return NULL;

}

newNode->data = value;

newNode->next = newNode;

newNode->prev = newNode;

return newNode;

}

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

Node\* newNode = createNode(value);

if (\*head == NULL) {

\*head = newNode;

} else {

Node\* tail = (\*head)->prev;

tail->next = newNode;

newNode->prev = tail;

newNode->next = \*head;

(\*head)->prev = newNode;

}

}

void printListForward(Node\* head) {

if (head == NULL) {

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

return;

}

Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

printf("(head)\n");

}

void printListBackward(Node\* head) {

if (head == NULL) {

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

return;

}

Node\* tail = head->prev;

Node\* temp = tail;

do {

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

temp = temp->prev;

} while (temp != tail);

printf("(tail)\n");

}

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

if (\*head == NULL) {

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

return;

}

Node\* temp = \*head;

if (temp->data == value) {

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

free(temp);

\*head = NULL;

} else {

Node\* tail = temp->prev;

\*head = temp->next;

(\*head)->prev = tail;

tail->next = \*head;

free(temp);

}

return;

}

temp = temp->next;

while (temp != \*head && temp->data != value) {

temp = temp->next;

}

if (temp == \*head) {

printf("Value not found in the list.\n");

return;

}

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

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

free(temp);

}

void freeList(Node\* head) {

if (head == NULL) return;

Node\* temp = head;

Node\* nextNode;

do {

nextNode = temp->next;

free(temp);

temp = nextNode;

} while (temp != head);

head = NULL;

}

int main() {

Node\* head = NULL;

insertNode(&head, 10);

insertNode(&head, 20);

insertNode(&head, 30);

insertNode(&head, 40);

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

printListForward(head);

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

printListBackward(head);

deleteNode(&head, 20);

printf("After deleting 20 (Forward): ");

printListForward(head);

deleteNode(&head, 10);

printf("After deleting 10 (Forward): ");

printListForward(head);

freeList(head);

return 0;

}

**Output:**

Circular Doubly Linked List (Forward): 10 <-> 20 <-> 30 <-> 40 <-> (head)

Circular Doubly Linked List (Backward): 40 <-> 30 <-> 20 <-> 10 <-> (tail)

After deleting 20 (Forward): 10 <-> 30 <-> 40 <-> (head)

After deleting 10 (Forward): 30 <-> 40 <-> (head)

**Section-B (Stack)**

**Experiment No.: 1**

**Program Description:**

Implementation of Stack using Array.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

#define MAX 5

typedef struct {

int arr[MAX];

int top;

} Stack;

void initStack(Stack\* stack) {

stack->top = -1;

}

int isFull(Stack\* stack) {

return stack->top == MAX - 1;

}

int isEmpty(Stack\* stack) {

return stack->top == -1;

}

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

if (isFull(stack)) {

printf("Stack Overflow! Unable to push %d\n", value);

return;

}

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

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

}

int pop(Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack Underflow! The stack is empty\n");

return -1;

}

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

}

int peek(Stack\* stack) {

if (isEmpty(stack)) {

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

return -1;

}

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

}

void display(Stack\* stack) {

if (isEmpty(stack)) {

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

return;

}

printf("Stack contents: ");

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

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

}

printf("\n");

}

int main() {

Stack stack;

initStack(&stack);

push(&stack, 10);

push(&stack, 20);

push(&stack, 30);

push(&stack, 40);

push(&stack, 50);

push(&stack, 60);

display(&stack);

printf("Top element is: %d\n", peek(&stack));

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

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

display(&stack);

return 0;

}

**Output:**

**10 pushed onto stack**

**20 pushed onto stack**

**30 pushed onto stack**

**40 pushed onto stack**

**50 pushed onto stack**

**Stack Overflow! Unable to push 60**

**Stack contents: 10 20 30 40 50**

**Top element is: 50**

**Popped element: 50**

**Popped element: 40**

**Stack contents: 10 20 30**

**Experiment No.: 2**

**Program Description:**

Implementation of Stack using Pointers.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node\* next;

} Node;

typedef struct {

Node\* top;

} Stack;

void initStack(Stack\* stack) {

stack->top = NULL;

}

int isEmpty(Stack\* stack) {

return stack->top == NULL;

}

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

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

if (newNode == NULL) {

printf("Memory allocation failed! Unable to push %d\n", value);

return;

}

newNode->data = value;

newNode->next = stack->top;

stack->top = newNode;

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

}

int pop(Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack Underflow! The stack is empty\n");

return -1;

}

Node\* temp = stack->top;

int poppedValue = temp->data;

stack->top = stack->top->next;

free(temp);

return poppedValue;

}

int peek(Stack\* stack) {

if (isEmpty(stack)) {

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

return -1;

}

return stack->top->data;

}

void display(Stack\* stack) {

if (isEmpty(stack)) {

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

return;

}

Node\* temp = stack->top;

printf("Stack contents: ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("\n");

}

int main() {

Stack stack;

initStack(&stack);

push(&stack, 10);

push(&stack, 20);

push(&stack, 30);

push(&stack, 40);

display(&stack);

printf("Top element is: %d\n", peek(&stack));

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

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

display(&stack);

return 0;

}

**Output:**

10 pushed onto stack

20 pushed onto stack

30 pushed onto stack

40 pushed onto stack

Stack contents: 40 30 20 10

Top element is: 40

Popped element: 40

Popped element: 30

Stack contents: 20 10

**Experiment No.: 3**

**Program Description:**

Program for Tower of Hanoi using recursion.

**Solution:**

#include <stdio.h>

void towerOfHanoi(int n, char source, char destination, char auxiliary) {

if (n == 1) {

printf("Move disk 1 from %c to %c\n", source, destination);

return;

}

towerOfHanoi(n - 1, source, auxiliary, destination);

printf("Move disk %d from %c to %c\n", n, source, destination);

towerOfHanoi(n - 1, auxiliary, destination, source);

}

int main() {

int n;

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

scanf("%d", &n);

printf("\nThe sequence of moves involved in the Tower of Hanoi are:\n");

towerOfHanoi(n, 'A', 'C', 'B'); // 'A' is source, 'C' is destination, 'B' is auxiliary

return 0;

}

**Output:**

Enter the number of disks: 3

The sequence of moves involved in the Tower of Hanoi are:

Move disk 1 from A to C

Move disk 2 from A to B

Move disk 1 from C to B

Move disk 3 from A to C

Move disk 1 from B to A

Move disk 2 from B to C

Move disk 1 from A to C

**Experiment No.: 4**

**Program Description:**

Program to find out factorial of given number using recursion. Also show the various states of stack using in this program.

**Solution:**

#include <stdio.h>

int factorial(int n) {

printf("Calling factorial(%d)\n", n);

if (n == 0) {

printf("Returning 1 from factorial(0)\n");

return 1;

}

int result = n \* factorial(n - 1);

printf("Returning %d from factorial(%d)\n", result, n);

return result;

}

int main() {

int number;

printf("Enter a number: ");

scanf("%d", &number);

int result = factorial(number);

printf("Factorial of %d is %d\n", number, result);

return 0;

}

**Output:**

Enter a number: 3

Calling factorial(3)

Calling factorial(2)

Calling factorial(1)

Calling factorial(0)

Returning 1 from factorial(0)

Returning 1 from factorial(1)

Returning 2 from factorial(2)

Returning 6 from factorial(3)

Factorial of 3 is 6.

**Section-C (Queue)**

**Experiment No.: 1**

**Program Description:**

Implementation of Queue using Array.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

#define MAX 5

typedef struct Queue {

int arr[MAX];

int front;

int rear;

} Queue;

void initQueue(Queue\* q) {

q->front = -1;

q->rear = -1;

}

int isEmpty(Queue\* q) {

return (q->front == -1);

}

int isFull(Queue\* q) {

return (q->rear == MAX - 1);

}

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

if (isFull(q)) {

printf("Queue is full, cannot enqueue %d\n", value);

return;

}

if (q->front == -1) {

q->front = 0;

}

q->rear++;

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

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

}

int dequeue(Queue\* q) {

if (isEmpty(q)) {

printf("Queue is empty, cannot dequeue\n");

return -1;

}

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

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

q->front = -1;

q->rear = -1;

} else {

q->front++;

}

return value;

}

int peek(Queue\* q) {

if (isEmpty(q)) {

printf("Queue is empty, cannot peek\n");

return -1;

}

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

}

void display(Queue\* q) {

if (isEmpty(q)) {

printf("Queue is empty\n");

return;

}

printf("Queue elements: ");

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

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

}

printf("\n");

}

int main() {

Queue q;

initQueue(&q);

enqueue(&q, 10);

enqueue(&q, 20);

enqueue(&q, 30);

enqueue(&q, 40);

enqueue(&q, 50);

enqueue(&q, 60);

display(&q);

printf("Dequeued: %d\n", dequeue(&q));

printf("Dequeued: %d\n", dequeue(&q));

display(&q);

enqueue(&q, 60);

display(&q);

printf("Front element is %d\n", peek(&q));

return 0;

}

**Output:**

10 enqueued to the queue

20 enqueued to the queue

30 enqueued to the queue

40 enqueued to the queue

50 enqueued to the queue

Queue is full, cannot enqueue 60

Queue elements: 10 20 30 40 50

Dequeued: 10

Dequeued: 20

Queue elements: 30 40 50

60 enqueued to the queue

Queue elements: 30 40 50 60

Front element is 30.

**Experiment No.: 2**

**Program Description:**

Implementation of Queue using Pointers.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node\* next;

} Node;

typedef struct Queue {

Node\* front;

Node\* rear;

} Queue;

void initQueue(Queue\* q) {

q->front = NULL;

q->rear = NULL;

}

int isEmpty(Queue\* q) {

return (q->front == NULL);

}

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

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

if (newNode == NULL) {

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

return;

}

newNode->data = value;

newNode->next = NULL;

if (isEmpty(q)) {

q->front = newNode;

q->rear = newNode;

} else {

q->rear->next = newNode;

q->rear = newNode;

}

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

}

int dequeue(Queue\* q) {

if (isEmpty(q)) {

printf("Queue is empty, cannot dequeue\n");

return -1;

}

Node\* temp = q->front;

int value = temp->data;

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

if (q->front == NULL) {

q->rear = NULL;

}

free(temp);

return value;

}

int peek(Queue\* q) {

if (isEmpty(q)) {

printf("Queue is empty, cannot peek\n");

return -1;

}

return q->front->data;

}

void display(Queue\* q) {

if (isEmpty(q)) {

printf("Queue is empty\n");

return;

}

Node\* temp = q->front;

printf("Queue elements: ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("\n");

}

int main() {

Queue q;

initQueue(&q);

enqueue(&q, 10);

enqueue(&q, 20);

enqueue(&q, 30);

enqueue(&q, 40);

display(&q);

printf("Dequeued: %d\n", dequeue(&q));

display(&q);

printf("Peek front element: %d\n", peek(&q));

dequeue(&q);

dequeue(&q);

display(&q);

return 0;

}

**Output:**

10 enqueued to the queue

20 enqueued to the queue

30 enqueued to the queue

40 enqueued to the queue

Queue elements: 10 20 30 40

Dequeued: 10

Queue elements: 20 30 40

Peek front element: 20

Dequeued: 20

Dequeued: 30

Queue elements: 40

**Experiment No.: 3**

**Program Description:**

Implementation of Circular Queue using Array.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 5

typedef struct {

int arr[MAX\_SIZE];

int front;

int rear;

} CircularQueue;

void initQueue(CircularQueue\* queue) {

queue->front = -1;

queue->rear = -1;

}

int isEmpty(CircularQueue\* queue) {

return (queue->front == -1);

}

int isFull(CircularQueue\* queue) {

return ((queue->rear + 1) % MAX\_SIZE == queue->front);

}

void enqueue(CircularQueue\* queue, int value) {

if (isFull(queue)) {

printf("Queue is full. Cannot enqueue %d\n", value);

return;

}

if (queue->front == -1) {

queue->front = 0;

}

queue->rear = (queue->rear + 1) % MAX\_SIZE;

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

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

}

int dequeue(CircularQueue\* queue) {

if (isEmpty(queue)) {

printf("Queue is empty. Cannot dequeue\n");

return -1;

}

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

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

queue->front = -1;

queue->rear = -1;

} else {

queue->front = (queue->front + 1) % MAX\_SIZE;

}

return value;

}

int front(CircularQueue\* queue) {

if (isEmpty(queue)) {

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

return -1;

}

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

}

int rear(CircularQueue\* queue) {

if (isEmpty(queue)) {

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

return -1;

}

return queue->arr[queue->rear];

}

void display(CircularQueue\* queue) {

if (isEmpty(queue)) {

printf("Queue is empty\n");

return;

}

printf("Queue elements: ");

int i = queue->front;

while (i != queue->rear) {

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

i = (i + 1) % MAX\_SIZE;

}

printf("%d ", queue->arr[queue->rear]);

printf("\n");

}

int main() {

CircularQueue queue;

initQueue(&queue);

enqueue(&queue, 10);

enqueue(&queue, 20);

enqueue(&queue, 30);

enqueue(&queue, 40);

enqueue(&queue, 50);

display(&queue);

printf("Dequeued: %d\n", dequeue(&queue));

display(&queue);

enqueue(&queue, 60);

display(&queue);

printf("Front element: %d\n", front(&queue));

printf("Rear element: %d\n", rear(&queue));

enqueue(&queue, 70);

display(&queue);

return 0;

}

**Output:**

10 enqueued to the queue

20 enqueued to the queue

30 enqueued to the queue

40 enqueued to the queue

50 enqueued to the queue

Queue elements: 10 20 30 40 50

Dequeued: 10

Queue elements: 20 30 40 50

60 enqueued to the queue

Queue elements: 20 30 40 50 60

Front element: 20

Rear element: 60

70 enqueued to the queue

Queue elements: 20 30 40 50 60 70

**Section-D (Trees & Graphs)**

**Experiment No.: 1**

**Program Description:**

Implementation of Binary Search Tree.

**Solution:**

#include <stdio.h> #include <stdlib.h> struct Node { int data; struct Node\* left; struct Node\* right;

} ;

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node)); newNode->data = data; newNode->left = NULL; newNode->right = NULL; return newNode;

}

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;

}

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;

printf("Enter the number of values to insert into the BST: "); scanf("%d", &n); for (int i = 0; i < n; i++) { printf("Enter value %d: ", i + 1); scanf("%d", &value); root = insert(root, value);

}

printf("In-order traversal of the BST: "); inOrderTraversal(root); printf("\n"); return 0;

}

**Output:**

Enter the number of values to insert into the BST: 5

Enter value 1: 68

Enter value 2: 257

Enter value 3: 54

Enter value 4: 36

Enter value 5: 14

In-order traversal of the BST: 14 25 36 54 68

**Experiment No.:2**

**Program Description:**

Conversion of BST PreOrder/PostOrder/InOrder.

**Solution:**

#include <stdio.h> #include <stdlib.h> struct Node { int data; struct Node\* left; struct Node\* right;

} ;

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node)); newNode->data = data; newNode->left = NULL; newNode->right = NULL; return newNode;

}

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;

}

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);

}

}

int main() { struct Node\* root = NULL; int n, value;

printf("Enter the number of values to insert into the BST: "); scanf("%d", &n); for (int i = 0; i < n; i++) { printf("Enter value %d: ", i + 1); scanf("%d", &value); root = insert(root, value);

}

printf("In-order traversal of the BST: "); inOrderTraversal(root); printf("\n");

printf("Pre-order traversal of the BST: "); preOrderTraversal(root);

printf("\n");

printf("Post-order traversal of the BST: "); postOrderTraversal(root); printf("\n"); return 0;

}

**Output:**

Enter the number of values to insert into the BST: 5

Enter value 1: 89

Enter value 2: 785

Enter value 3: 46

Enter value 4: 5

Enter value 5: 25

In-order traversal of the BST: 4 5 25 78 89

Pre-order traversal of the BST: 89 78 4 5 25

Post-order traversal of the BST: 25 5 4 78 89

**Experiment No.: 3**

**Program Description:**  Implementation of Kruskal Algorithm  **Solution:**

#include <stdio.h>

#include <stdlib.h>

#de ine MAX 100 #de ine INF 99999 struct Edge {

int src, dest, weight;

} ;

struct Subset {

int parent; int rank;

};

int compare(const void\* a, const void\* b) { struct Edge\* edgeA = (struct Edge\*)a; struct Edge\* edgeB = (struct Edge\*)b; return edgeA->weight - edgeB->weight;

}

int ind(struct Subset subsets[], int i) {

if (subsets[i].parent != i) {

subsets[i].parent = ind(subsets, subsets[i].parent);

}

return subsets[i].parent;

}

void unionSubsets(struct Subset subsets[], int x, int y) {

int xroot = ind(subsets, x); int yroot = ind(subsets, y); if (subsets[xroot].rank < subsets[yroot].rank) {

subsets[xroot].parent = yroot;

} else if (subsets[xroot].rank > subsets[yroot].rank) { subsets[yroot].parent = xroot;

} else {

subsets[yroot].parent = xroot; subsets[xroot].rank++;

}

}

void kruskal(struct Edge edges[], int V, int E) {

struct Edge result[MAX]; int e = 0; int i = 0;

qsort(edges, E, sizeof(edges[0]), compare);

struct Subset\* subsets = (struct Subset\*)malloc(V \* sizeof(struct Subset)); for (int v = 0; v < V; v++) {

subsets[v].parent = v; subsets[v].rank = 0;

}

while (e < V - 1 && i < E) {

struct Edge next\_edge = edges[i++]; int x = ind(subsets, next\_edge.src); int y = ind(subsets, next\_edge.dest);

if (x != y) {

result[e++] = next\_edge; unionSubsets(subsets, x, y);

}

}

printf("Edges in the Minimum Spanning Tree:\n");

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

printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);

}

free(subsets);

}

int main() { int V, E;

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

scanf("%d", &V);

printf("Enter the number of edges: "); scanf("%d", &E); struct Edge edges[E];

printf("Enter the edges (src dest weight):\n"); for (int i = 0; i < E; i++) {

scanf("%d %d %d", &edges[i].src, &edges[i].dest, &edges[i].weight);

}

kruskal(edges, V, E); return 0;

}

**Output:**

Enter the number of vertices: 3

Enter the number of edges: 5

Enter the edges (src dest weight):

25

45

78

36

25

12

35

68

14

78

45

10

233

83

2

Edges in the Minimum Spanning Tree:

**Experiment No.: 4**

**Program Description:**  Implementation of Prim Algorithm  **Solution:**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h> #de ine V 5

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

int min = INT\_MAX, min\_index; for (int v = 0; v < V; v++) {

if (mstSet[v] == 0 && key[v] < min) {

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

}

}

return min\_index;

}

void printMST(int parent[], int graph[V][V]) {

printf("Edge \tWeight\n"); for (int i = 1; i < V; i++) {

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

}

}

void primMST(int graph[V][V]) {

int parent[V]; int key[V]; int mstSet[V]; for (int i = 0; i < V; i++) {

key[i] = INT\_MAX; mstSet[i] = 0;

}

key[0] = 0;

parent[0] = -1;

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

int u = minKey(key, mstSet); mstSet[u] = 1; for (int v = 0; v < V; v++) {

if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {

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

}

}

}

printMST(parent, graph);

}

int main() {

int graph[V][V] = {

{0, 2, 0, 6, 0},

{2, 0, 3, 8, 5},

{0, 3, 0, 0, 7},

{6, 8, 0, 0, 9},

{0, 5, 7, 9, 0}

};

primMST(graph); return 0;

}

**Output:**

Edge Weight

1. - 1 2
2. - 2 3
3. - 3 6
4. - 4 5

**Experiment No.: 5**

**Program Description:**

Implementation of Dijkstra Algorithm

**Solution:**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#de ine MAX\_VERTICES 20

int minDistance(int dist[], int sptSet[], int V) {

int min = INT\_MAX, min\_index; for (int v = 0; v < V; v++) {

if (sptSet[v] == 0 && dist[v] < min) {

min = dist[v]; min\_index = v;

}

}

return min\_index;

}

void printSolution(int dist[], int n) {

printf("Vertex \t Distance from Source\n"); for (int i = 0; i < n; i++) {

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

}

}

void dijkstra(int graph[MAX\_VERTICES][MAX\_VERTICES], int src, int V) {

int dist[MAX\_VERTICES]; int sptSet[MAX\_VERTICES];

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

dist[i] = INT\_MAX; sptSet[i] = 0;

}

dist[src] = 0;

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

int u = minDistance(dist, sptSet, V); sptSet[u] = 1;

for (int v = 0; v < V; v++) { if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] <

dist[v]) {

dist[v] = dist[u] + graph[u][v];

}

}

}

printSolution(dist, V);

}

int main() {

int graph[MAX\_VERTICES][MAX\_VERTICES]; int V, src;

printf("Enter the number of vertices (max %d): ", MAX\_VERTICES); scanf("%d", &V);

printf("Enter the adjacency matrix (use 0 for no edge):\n"); for (int i = 0; i < V; i++) { for (int j = 0; j < V; j++) { scanf("%d", &graph[i][j]);

}

}

printf("Enter the source vertex (0 to %d): ", V - 1); scanf("%d", &src); if (src < 0 || src >= V) { printf("Invalid source vertex.\n"); return 1;

}

dijkstra(graph, src, V); return 0;

}

**Output:**

Enter the number of vertices (max 20): 2

Enter the adjacency matrix (use 0 for no edge):

1

1

0

0

Enter the source vertex (0 to 1): 1

Vertex Distance from Source

1. 2147483647
2. 0

**Section-E (Sorting & Searching)**

**Experiment No.: 1**

**Program Description:**

Implementation of Sorting

a. Bubble

1. Selection
2. Insertion
3. Quick
4. Merge

**Solution:**

**a.Bubble Sort**

#include <stdio.h>

void bubbleSort(int arr[], int n) { int i, j, temp;

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

swapped = 0;

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

if (arr[j] > arr[j + 1]) { temp = arr[j]; arr[j] = arr[j + 1]; arr[j + 1] = temp; swapped = 1;

}

}

if (swapped == 0) {

break;

}

}

}

void printArray(int arr[], int size) { int i;

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

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

}

printf("\n");

}

int main() {

int arr[] = {64, 34, 25, 12, 22, 11, 90}; int n = sizeof(arr) / sizeof(arr[0]); printf("Unsorted array: \n"); printArray(arr, n); bubbleSort(arr, n); printf("Sorted array: \n"); printArray(arr, n); return 0;

}

**Output:**

Unsorted array:

64 34 25 12 22 11 90

Sorted array:

11 12 22 25 34 64 90

**b. Selection Sort**

#include <stdio.h>

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

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

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

minIndex = j;

}

}

if (minIndex != i) {

temp = arr[i]; arr[i] = arr[minIndex]; arr[minIndex] = temp;

}

}

}

void printArray(int arr[], int size) { int i;

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

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

}

printf("\n");

}

int main() {

int arr[] = {64, 25, 12, 22, 11}; int n = sizeof(arr) / sizeof(arr[0]); printf("Unsorted array: \n"); printArray(arr, n); selectionSort(arr, n); printf("Sorted array: \n"); printArray(arr, n); return 0;

}

**Output:**

Unsorted array:

64 25 12 22 11

Sorted array:

11 12 22 25 64

**c.Insertion Sort:**

#include <stdio.h>

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

int i, key, j;

for (i = 1; i < n; 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 printArray(int arr[], int size) { int i;

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

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

}

printf("\n");

}

int main() {

int arr[] = {25,12,87,63,89}; int n = sizeof(arr) / sizeof(arr[0]); printf("Unsorted array: \n"); printArray(arr, n); insertionSort(arr, n); printf("Sorted array: \n"); printArray(arr, n); return 0;

}

**Output:**

Unsorted array:

25 12 87 63 89 Sorted array:

12 25 63 87 89

**d.Quick Sort**

#include <stdio.h>

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

int temp = \*a; \*a = \*b;

\*b = temp;

}

int partition(int arr[], int low, int high) {

int pivot = arr[high]; int i = (low - 1); for (int j = low; j < high; j++) {

if (arr[j] <= pivot) {

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]); return (i + 1);

}

void quickSort(int arr[], int low, int high) {

if (low < high) {

int pi = partition(arr, low, high); quickSort(arr, low, pi - 1); quickSort(arr, pi + 1, high);

}

}

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

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

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

}

printf("\n");

}

int main() {

int arr[] = {75,68,42,30,20,}; int n = sizeof(arr) / sizeof(arr[0]); printf("Unsorted array: \n"); printArray(arr, n); quickSort(arr, 0, n - 1); printf("Sorted array: \n"); printArray(arr, n); return 0;

}

**Output:**

Unsorted array:

75 68 42 30 20 Sorted array: 20 30 42 68 75

**e.Merge Sort:**  #include <stdio.h>

void merge(int arr[], int left, int mid, int right) { int i, j, k;

int n1 = mid - left + 1; int n2 = right - mid; int L[n1], R[n2]; for (i = 0; i < n1; i++)

L[i] = arr[left + i]; for (j = 0; j < n2; j++)

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

i = 0; j = 0; k = left; while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i]; i++;

} else { arr[k] = R[j]; j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i]; i++; k++;

}

while (j < n2) {

arr[k] = R[j]; j++; k++;

}

}

void mergeSort(int arr[], int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2; mergeSort(arr, left, mid); mergeSort(arr, mid + 1, right); merge(arr, left, mid, right);

}

}

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

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

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

}

printf("\n");

}

int main() {

int arr[] = {98,25,87,02,12};

int n = sizeof(arr) / sizeof(arr[0]); printf("Unsorted array: \n"); printArray(arr, n); mergeSort(arr, 0, n - 1); printf("Sorted array: \n"); printArray(arr, n); return 0;

}

**Output:**

Unsorted array:

98 25 87 2 12

Sorted array:

2 12 25 87 98

**Experiment No.: 2**

**Program Description:**

Implementation of Binary Search on a list of numbers stored in an Array

**Solution:**

#include <stdio.h>

int binary\_search(int arr[], int size, int target) { int left = 0; int right = size - 1; while (left <= right) { int mid = left + (right - left) / 2; if (arr[mid] == target) {

return mid;

}

else if (arr[mid] < target) { left = mid + 1;

}

else { right = mid - 1;

}

}

return -1;

}

int main() {

int numbers[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; int size = sizeof(numbers) / sizeof(numbers[0]); int target = 7;

int result = binary\_search(numbers, size, target); if (result != -1) {

printf("Element found at index: %d\n", result);

} else {

printf("Element not found in the array.\n");

}

return 0;

}

**Output:**

Element found at index: 6

**Experiment No.: 3**

**Program Description:**

Implementation of Binary Search on a list of strings stored in an Array

**Solution:**

#include <stdio.h>

#include <string.h>

int binary\_search(char \*arr[], int size, const char \*target) { int left = 0; int right = size - 1; while (left <= right) {

int mid = left + (right - left) / 2; int comparison = strcmp(arr[mid], target); if (comparison == 0) {

return mid;

}

else if (comparison < 0) {

left = mid + 1;

}

else {

right = mid - 1;

}

}

return -1;

}

int main() { char \*strings[] = {

"apple",

"banana",

"cherry",

"date",

" ig",

"grape",

"kiwi",

"mango",

"orange",

"pear"

} ;

int size = sizeof(strings) / sizeof(strings[0]); const char \*target = "kiwi"; int result = binary\_search(strings, size, target); if (result != -1) {

printf("Element found at index: %d\n", result);

} else {

printf("Element not found in the array.\n");

}

return 0;

}

**Output:**

Element found at index: 6

**Experiment No.:4**

**Program Description:**

Implementation of Linear Search on a list of strings stored in an Array

**Solution:**

#include <stdio.h>

#include <string.h>

int linear\_search(char \*arr[], int size, const char \*target) { for (int i = 0; i < size; i++) {

if (strcmp(arr[i], target) == 0) { return i;

}

}

return -1;

}

int main() { char \*strings[] = {

"apple",

"banana",

"cherry",

"date",

" fig",

"grape",

"kiwi",

"mango",

"orange",

"pear"

};

int size = sizeof(strings) / sizeof(strings[0]); const char \*target = "grape";

int result = linear\_search(strings, size, target); if (result != -1) {

printf("Element found at index: %d\n", result);

} else {

printf("Element not found in the array.\n");

}

return 0;

}

**Output:**

Element found at index: 5