RUSTAMJI INSTITUTE OF TECHNOLOGY

BSF ACADEMY, TEKANPUR

Lab File for

CS303 (Data Structure)

Submitted by

VARUN JAIN (0902CS231128)

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, VARUN JAIN, 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: 16/ 12/2024

Student's Signature

VARUN JAIN

0902CS231128

ENVORIONMENT USED

Hardware Configuration	**:**	Intel Core i5 12th Gen.
C Compiler	**:**	GCC Compiler.
User Interface	**:**	Visual Studio Code.

GROUP MEMBERS

MEMBER 1: PREMVIR SINGH CHAUDHARY (0902CS231073)

MEMBER 2: SANGAM KUMAR (0902CS231094)

MEMBER3: VARUN JAIN (0902CS231128)

MEMBER 4: YASH SAHU (0902CS231135)

TABLE OF CONTENTS

Section-A (Linked List)

S. No.	Practical Description	Page Nos.	COs
1	Implementation of Linked List using array.	1-4	CO-1
2	Implementation of Linked List using Pointers.	5-8	CO-1
3	Implementation of Doubly Linked List using Pointers.	9-11	CO-1
4	Implementation of Circular Single Linked List using Pointers.	12-14	CO-1
5	Implementation of Circular Doubly Linked List using Pointers.	15-19	CO-1

Section-B (Stack)

S. No.	Practical Description	Page Nos.	COs
1	Implementation of Stack using Array.	20-21	CO-2
2	Implementation of Stack using Pointers.	22-23	CO-2
3	Program for Tower of Hanoi using recursion.	24-25	CO-2
4	Program to find out factorial of given number using recursion. Also show the various states of stack using in this program.	26-27	CO-2

Section-C (Queue)

S. No.	Practical Description	Page Nos.	COs
1	Implementation of Queue using Array.	28-29	CO-2
2	Implementation of Queue using Pointers.	30-31	CO-2
3	Implementation of Circular Queue using Array.	32-33	CO-2

Section-D (Trees)

S. No.	Practical Description	Page Nos.	COs
1	Implementation of Binary Search Tree.	34-36	CO-3
2	Conversion of BST PreOrder/PostOrder/InOrder.	37-38	CO-3
3	Implementation of Kruskal Algorithm	39-42	CO-3
4	Implementation of Prim Algorithm	43-45	CO-3
5	Implementation of Dijkstra Algorithm	46-48	CO-3

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	49-56	CO-5
2	Implementation of Binary Search on a list of numbers stored in an Array	57	CO-5
3	Implementation of Binary Search on a list of strings stored in an Array	58	CO-5
4	Implementation of Linear Search on a list of strings stored in an Array	59	CO-5
5	Implementation of Binary Search on a list of strings stored in a Single Linked List	60-61	CO-5

Experiment No.: A1

Program Description:

Implementation of Linked List using array.

Solution:

Implementation of Linked List using array.

CODE

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100

struct node{
    int data;
    int next;
};

struct node arr[MAX_SIZE];
int free_slot=0;
int head=-1;

int createnode(int data){
    if(free_slot==MAX_SIZE){
        printf("ERROR: OVERFLOW.\n");
        return -1;
    }
    int new_node=free_slot;
    free_slot=arr[free_slot].next;
    arr[new_node].data= data;
    arr[new_node].next =-1;

    return new_node;
}
void insertathead(int data){
    int new_node=createnode(data);
    if(new_node==-1){
        return;
    }
    arr[new_node].next =head;
    head=new_node;
    printf("Successfully done.\n");
}
void insertatend(int data){
    int new_node=createnode(data);
    if(new_node==-1){
        return ;
    }
    if(head==-1){
        head=new_node;
```

```
    }
    else{
        int last = head;
        while(arr[last].next!=-1){
            last=arr[last].next;
        }
        arr[last].next=new_node;
    }
    printf("Successfully done.\n");
}
void insertatposition(int data,int key){
    if (key < 1){
        printf("Invalid key.\n");
        return;
    }
    int new_node= createnode(data);
    if(new_node==-1){
        return;
    }
    if(key==1){
        arr[new_node].next=head;
        head = new_node;
    }
    else{
        int prev= head;
        int count = 1;
        while (count < key -1 && prev!=-1){
            prev = arr[prev].next;
            count++;
        }
        if (prev == -1){
            printf("Invalid position.\n");
            return;
        }
        arr[new_node].next=arr[prev].next;
        arr[prev].next=new_node;
    }
    printf("Succesfully Done.\n");
}
int deletenode(int i){
    if(i<0 || i >= MAX_SIZE){
        printf("Invalid index.\n");
        return -1;
    }
    int data = arr[i].data;
    arr[i].next= free_slot;
    free_slot= i;
    return data;
}
int deleteathead(){
    if ( head== -1){
        printf("ERROR:Underflow.\n");
        return -1;
    }
    int del_node = head;
    head = arr[head].next;
    int data = deletenode(del_node);
```

```
    printf("Successfully Done.\n");

}
int deleteatend(){
    if(head==-1){
        printf("ERROR: Underflow.\n");
        return -1;
    }

    int prev =-1;
    int last = head;
    while(arr[last].next!=-1){
        prev = last;
        last = arr[last].next;
    }
    if(prev==-1){
        head =-1;
    }
    else{
        arr[prev].next=-1;
    }
    int info= deletenode(last);
    printf("Successfully Done.\n");
}
void display(){
    if(head==-1){
        printf("ERROR:Underflow.\n");
        return;
    }
    printf("THE LINKED LIST IS:\n");
    int temp = head;
    while(temp!=-1){
        printf("%d->",arr[temp].data);
        temp=arr[temp].next;
    }
    printf("NULL\n");

}
void init(){
    for(int j=0;j<MAX_SIZE; j++){
        arr[j].data = 0;
        arr[j].next =j+1;
    }
    arr[MAX_SIZE-1].next=-1;
}

int main(){
    init();
    int ch,data,key;

    printf("MENU:\n");
    printf("1.Insert at head\n");
    printf("2.Insert at end.\n");
    printf("3.Insert at position.\n");
    printf("4.Delete at head.\n");
    printf("5.Delete at end.\n");
    printf("6.Display\n");
```

```
    printf("7.Exit\n");
    while( 1){
        printf("Enter your choice:");
        scanf("%d", &ch);

        switch(ch){
            case 1:
            printf("Enter data to insert at head:");
            scanf("%d",&data);
            insertathead(data);
            break;
            case 2:
            printf("Enter data to insert at end:");
            scanf("%d",&data);
            insertatend(data);
            break;
            case 3:
            printf("Enter the data to enter and position:");
            scanf("%d%d",&data,&key);
            insertatposition(data,key);
            break;
            case 4:
            printf("Deleting data from head\n");
            deleteathead();
            break;
            case 5:
            printf("Deleting element from end\n ");
            deleteatend();
            break;
            case 6:
            printf("here is your output:");
            display();
            break;
            case 7:
            printf("Thankyou\n");
            return 0;
            default:
            printf("Invalid choice.Please try again./n");
        }
    }
}
```

Output:

```
output,MENU:
1.Insert at head
2.Insert at end.
3.Insert at position.
4.Delete at head.
5.Delete at end.
6.Display
7.Exit
Enter your choice:
```

Experiment No.: A2

Program Description:

Implementation of Linked List using Pointers.

Solution:

Implementation of Linked List using Pointers.

CODE

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *next;
};
struct node *head = NULL;
struct node *createnode(int data) {
    struct node *newnode = (struct node *)malloc(sizeof(struct node));
    if (newnode == NULL) {
        printf("Memory allocation failed. Cannot create the node.\n");
        return NULL;
    }
    newnode->data = data;
    newnode->next = NULL;
    return newnode;
}
void insertathead(int data) {
    struct node *newnode = createnode(data);
    if (newnode == NULL) {
        return;
    }
    newnode->next = head;
    head = newnode;
    printf("Successfully Done.\n");
}
void insertatend(int data) {
    struct node *newnode = createnode(data);
    if (newnode == NULL) {
        return;
    }
    if (head == NULL) {
        head = newnode;
    } else {
        struct node *last = head;
        while (last->next != NULL) {
            last = last->next;
        }
        last->next = newnode;
    }
```

```
    printf("Successfully Done.\n");
}
void insertatposition(int data, int key) {
    if (key < 1) {
        printf("Invalid key.\n");
        return;
    }
    struct node *newnode = createnode(data);
    if (newnode == NULL) {
        return;
    }
    if (key == 1) {
        newnode->next = head;
        head = newnode;
    } else {
        struct node *prev = head;
        int count = 1;
        while (count < key - 1 && prev != NULL) {
            prev = prev->next;
            count++;
        }
        if (prev == NULL) {
            printf("Invalid key.\n");
            return;
        }
        newnode->next = prev->next;
        prev->next = newnode;
    }
    printf("Successfully Done.\n");
}
int deleteathead() {
    if (head == NULL) {
        printf("ERROR:Underflow.\n");
        return -1;
    }
    struct node *delnode = head;
    head = head->next;
    int data = delnode->data;
    free(delnode);
    printf("Successfully Done.\n");
}
int deleteatend() {
    if (head == NULL) {
        printf("ERROR:Underflow.\n");
        return -1;
    }
    struct node *prev = NULL;
    struct node *last = head;
    while (last->next != NULL) {
        prev = last;
        last = last->next;
    }
    int data = last->data;
    if (prev == NULL) {
        head = NULL;
    } else {
        prev->next = NULL;
```

```
    }
    free(last);
    printf("Successfully Done.\n");
}
void display() {
    if (head == NULL) {
        printf("ERROR:Underflow");
        return;
    }
    printf("The linked list is:\n");
    struct node *temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
int main() {
    int ch, data, key;
    printf("Menu:\n");
    printf("1. Insert at head\n");
    printf("2. Insert at end\n");
    printf("3. Insert at position\n");
    printf("4. Delete at head\n");
    printf("5. Delete at end\n");
    printf("6. Display\n");
    printf("7. Exit\n");
    while(1){
        printf("Enter your choice: ");
        scanf("%d", &ch);
        switch (ch) {
        case 1:
            printf("Enter the data to insert at the head: ");
            scanf("%d", &data);
            insertathead(data);
            break;
        case 2:
            printf("Enter the data to insert at the end: ");
            scanf("%d", &data);
            insertatend(data);
            break;
        case 3:
            printf("Enter the data and position to insert: ");
            scanf("%d%d", &data, &key);
            insertatposition(data,key);
            break;
        case 4:
            deleteathead();
            break;
        case 5:
            deleteatend();
            break;
        case 6:
            display();
            break;
        case 7:
            printf("Thank you\n");
```

```
            return 0;
        default:
            printf("Invalid choice. Please try again.\n");
        }
    }
}
```

OUTPUT

1. Insert at head

2. Insert at end

3. Insert at position

4. Delete at head

5. Delete at end

6. Display

7. Exit

Enter your choice:

Experiment No.: A3

Program Description:

Implementation of Doubly Linked List using Pointers.

Solution:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node{
    int data;
    struct Node* prev;
    struct Node* next;
} Node;
Node* createNode(int data) {
    Node* newNode = (Node*) malloc(sizeof(Node));
    if (!newNode) {
        printf("Memory error\n");
        exit(1);
    }
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}
void insertAtBeginning(Node** head, int data) {
    Node* newNode = createNode(data);
    if (*head != NULL) {
        (*head)->prev = newNode;
    }
    newNode->next = *head;
    *head = newNode;
}
void insertAtEnd(Node** head, int data) {
    Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
}
```

```
void traverse(Node* head) {
    Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
Node* search(Node* head, int key) {
    Node* temp = head;
    while (temp != NULL) {
        if (temp->data == key) {
            return temp;
        }
        temp = temp->next;
    }
    return NULL;
}
void deleteAtBeginning(Node** head) {
    if (*head == NULL)
    return;
    Node* temp = *head;
    *head = (*head)->next;
    if (*head != NULL) {
    (*head)->prev = NULL;
    }
    free(temp);
}
void deleteAtEnd(Node** head) {
    if (*head == NULL)
    return;
    Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    if (temp->prev != NULL) {
        temp->prev->next = NULL;
    }
    else{
        *head = NULL;
    }
    free(temp);
}
int main(){
    Node* head = NULL;
    int choice, data;
    while(1){
        printf("1. Insert element at the beginning \n");
        printf("2. Insert element at the end \n");
        printf("3. Display all elements \n");
        printf("4. Search for an element \n");
        printf("5. Delete element from the beginning \n");
        printf("6. Delete element from the end \n");
        printf("7. Exit \n");
        printf("Enter your choice : ");
        scanf("%d",&choice);
        switch(choice){
```

```
            case 1: printf("Enter a value to insert: ");
                    scanf("%d", &data);
                    insertAtBeginning(&head, data);
                    break;
            case 2: printf("Enter a value to insert: ");
                    scanf("%d", &data);
                    insertAtEnd(&head, data);
                    break;
            case 3: traverse(head);
                    break;
            case 4: printf("Enter a value to search: ");
                    scanf("%d", &data);
                    Node* result = search(head, data);
                    if(result != NULL) {
                        printf("Element found: %d\n", result->data);
                    } else {
                        printf("Element not found\n");
                    }
                    break;
            case 5: deleteAtBeginning(&head);
                    break;
            case 6: deleteAtEnd(&head);
                    break;
            case 7: exit(0);
            default: printf("Invalid choice! Try again!!!\n");
        }
    }
    return 0;
}
```

OUTPUT

1. Insert element at the beginning

2. Insert element at the end

3. Display all elements

4. Search for an element

5. Delete element from the beginning

6. Delete element from the end

7. Exit

Enter your choice :

Experiment No.: A4

Program Description:

Implementation of Doubly Linked List using Pointers.

Solution:

```
// Implementation of Circular Single Linked List using Pointers.
#include <stdio.h>
#include <stdlib.h>

// Node structure
struct Node
{
    int data;
    struct Node *next;
};

// Function to create a new node
struct Node *createNode(int value)
{
    struct Node *newNode = (struct Node *)malloc(sizeof(struct
Node));
    if (newNode == NULL)
    {
        printf("Memory allocation failed.\n");
        exit(1);
    }

    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the beginning of the circular linked
list
struct Node *insertAtBeginning(struct Node *head, int value)
{
    struct Node *newNode = createNode(value);

    if (head == NULL)
```

```
    {
        // If the list is empty, make the new node the head and
point to itself
        head = newNode;
        head->next = head;
    }
    else
    {
        // Otherwise, insert the new node at the beginning and
update pointers
        newNode->next = head->next;
        head->next = newNode;
    }

    return head;
}

// Function to display the circular linked list
void display(struct Node *head)
{
    if (head == NULL)
    {
        printf("List is empty.\n");
        return;
    }

    struct Node *current = head;
    do
    {
        printf("%d -> ", current->data);
        current = current->next;
    } while (current != head);
    printf("(head)\n");
}

// Function to free the memory allocated for the circular linked
list
void freeList(struct Node *head)
{
    if (head == NULL)
    {
        return;
    }

    struct Node *current = head;
    struct Node *temp;

    do
    {
        temp = current;
        current = current->next;
        free(temp);
    } while (current != head);
}
```

```c
int main()
{
    struct Node *head = NULL;

    // Inserting elements into the circular linked list
    head = insertAtBeginning(head, 3);
    head = insertAtBeginning(head, 2);
    head = insertAtBeginning(head, 1);

    // Displaying the circular linked list
    printf("Circular Linked List: ");
    display(head);

    // Freeing the memory allocated for the circular linked list
    freeList(head);

    return 0;
}
```

```
Output
Circular Linked List: 3 -> 1 -> 2 -> (head)
```

Experiment No.: A5

Program Description:

Implementation of Circular Doubly Linked List using Pointers.

Solution:

```
#include <stdio.h>
#include <stdio.h>

// Node structure for a doubly linked list
struct Node {
   int data;
   struct Node* next;
   struct Node* prev;
};

// Function to create a new node
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 = NULL;
    newNode->prev = NULL;
    return newNode;
}

// Function to insert a node at the beginning of the circular doubly linked list
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = createNode(data);

    if (*head == NULL) {
        // If the list is empty, make the new node the only node in the list
        *head = newNode;
        (*head)->next = *head;
        (*head)->prev = *head;
    } else {
        // If the list is not empty, insert the new node at the beginning
        newNode->next = *head;
        newNode->prev = (*head)->prev;
        (*head)->prev->next = newNode;
        (*head)->prev = newNode;
        *head = newNode; // Update the head pointer
    }
}

// Function to display the circular doubly linked list
void displayList(struct Node* head) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
```

```
    struct Node* current = head;

    do {
        printf("%d <-> ", current->data);
        current = current->next;
    } while (current != head);

    printf("(head)\n");
}

// Function to free the memory allocated for the circular doubly linked list
void freeList(struct Node** head) {
    if (*head == NULL) {
        return;
    }

    struct Node* current = *head;
    struct Node* nextNode;

    do {
        nextNode = current->next;
        free(current);
        current = nextNode;
    } while (current != *head);

    *head = NULL;
}

int main() {
    struct Node* head = NULL;

    // Insert nodes at the beginning of the list
```

```
    insertAtBeginning(&head, 3);
    insertAtBeginning(&head, 2);
    insertAtBeginning(&head, 1);

    // Display the list
    printf("Circular Doubly Linked List:\n");
    displayList(head);

    // Free the memory allocated for the list
    freeList(&head);

    return 0;
}
```

Implementation of Circular Doubly Linked List using Pointers.

```
// Function to insert a node at the beginning of the list
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        newNode->next = *head;
        newNode->prev = *head;
    } else {
        newNode->next = *head;
        newNode->prev = (*head)->prev;
        (*head)->prev->next = newNode;
        (*head)->prev = newNode;
        *head = newNode;
    }
}

// Function to display the Circular Doubly Linked List
```

```
void displayList(struct Node* head) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }

    struct Node* current = head;
    do {
        printf("%d <-> ", current->data);
        current = current->next;
    } while (current != head);

    printf("(head)\n");
}

// Main function to test the Circular Doubly Linked List implementation
int main() {
    struct Node* head = NULL;

    // Inserting nodes at the beginning
    insertAtBeginning(&head, 3);
    insertAtBeginning(&head, 2);
    insertAtBeginning(&head, 1);

    // Displaying the Circular Doubly Linked List
    printf("Circular Doubly Linked List:\n");
    displayList(head);

    // Adding more nodes
    insertAtBeginning(&head, 4);
    insertAtBeginning(&head, 5);

    // Displaying the Circular Doubly Linked List again
    printf("Updated Circular Doubly Linked List:\n");
```

```
    displayList(head);

    return 0;
}
```

Experiment No.: B1

Program Description:

Implementation of Stack using Array

Solution:

```
//Implementation of Stack using Array.
#include <stdio.h>
int stack[100], choice, n, top, x, i;
void push(void);
void pop(void);
void display(void);
int main()
{
    top = -1;
    printf("\n Enter the size of STACK[MAX=100]:");
    scanf("%d", &n);
    printf("\n\t STACK OPERATIONS USING ARRAY");
    printf("\n\t--------------------------------");
    printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
    do
    {
        printf("\n Enter the Choice:");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
        {
            push();
            break;
        }
```

```
        case 2:
        {
            pop();
            break;
        }
        case 3:
        {
            display();
            break;
        }
        case 4:
        {
            printf("\n\t EXIT POINT ");
            break;
        }
        default:
        {
            printf("\n\t Please Enter a Valid Choice(1/2/3/4)");
        }
        }
    } while (choice != 4);
    return 0;
}
void push()
{
    if (top >= n - 1)
    {
        printf("\n\tSTACK is over flow");
    }
    else
    {
        printf(" Enter a value to be pushed:");
        scanf("%d", &x);
        top++;
        stack[top] = x;
    }
}
void pop()
{
    if (top <= -1)
    {
        printf("\n\t Stack is under flow");
    }
    else
    {
        printf("\n\t The popped elements is %d", stack[top]);
        top--;
    }
}
void display()
{
    if (top >= 0)
    {
        printf("\n The elements in STACK \n");
```

```
        for (i = top; i >= 0; i--)
            printf("\n%d", stack[i]);
        printf("\n Press Next Choice");
    }
    else
    {
        printf("\n The STACK is empty");
    }
}
```

```
Output
Enter the size of STACK[MAX=100]:10
STACK OPERATIONS USING ARRAY
     --------------------------------
      1.PUSH
      2.POP
      3.DISPLAY
      4.EXIT
 Enter the Choice:2
 Stack is under flow
```

Experiment No.: B2

Program Description:

Implementation of Stack using Pointers.

Solution:

CODE

```
#include <stdio.h>
#include <stdlib.h>

// Define a structure for a stack node
struct Node
{
    int data;
    struct Node *next;
};

// Function to create a new node
struct Node *createNode(int data)
{
    struct Node *newNode = (struct Node *)malloc(sizeof(struct
Node));
    if (newNode == NULL)
    {
        printf("Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
```

```
    newNode->next = NULL;
    return newNode;
}

// Function to check if the stack is empty
int isEmpty(struct Node *root)
{
    return (root == NULL);
}

// Function to push a new element onto the stack
void push(struct Node **root, int data)
{
    struct Node *newNode = createNode(data);
    newNode->next = *root;
    *root = newNode;
    printf("%d pushed to stack\n", data);
}

// Function to pop an element from the stack
int pop(struct Node **root)
{
    if (isEmpty(*root))
    {
        printf("Stack is empty\n");
        exit(EXIT_FAILURE);
    }
    struct Node *temp = *root;
    *root = (*root)->next;
    int popped = temp->data;
    free(temp);
    return popped;
}

// Function to peek at the top element of the stack
int peek(struct Node *root)
{
    if (isEmpty(root))
    {
        printf("Stack is empty\n");
        exit(EXIT_FAILURE);
    }
    return root->data;
}

// Example usage of the stack
int main()
{
    struct Node *root = NULL;

    push(&root, 10);
    push(&root, 20);
    push(&root, 30);
```

```
    printf("Top element is %d\n", peek(root));

    printf("%d popped from stack\n", pop(&root));
    printf("%d popped from stack\n", pop(&root));

    printf("Top element is %d\n", peek(root));

    return 0;
}
```

Output

10 pushed to stack

20 pushed to stack

30 pushed to stack

Top element is 30

30 popped from stack

20 popped from stack

Top element is 10

Experiment No.: B3

Program Description:

Program for Tower of Hanoi using recursion.

Solution:

```
#include <stdio.h>

// Function to move a disk from source pole to destination pole
void towerOfHanoi(int n, char source, char destination, char
auxiliary)
{
    if (n == 1)
    {
        // Base case: If there's only one disk, move it from source
to destination
        printf("Move disk 1 from %c to %c\n", source, destination);
        return;
    }

    // Move (n-1) disks from source to auxiliary pole using
destination pole
    towerOfHanoi(n - 1, source, auxiliary, destination);

    // Move the nth disk from source to destination pole
    printf("Move disk %d from %c to %c\n", n, source, destination);
```

```
    // Move the (n-1) disks from auxiliary pole to destination pole
using source pole
    towerOfHanoi(n - 1, auxiliary, destination, source);
}

int main()
{
    int n;

    // Input: Number of disks
    printf("Enter the number of disks: ");
    scanf("%d", &n);

    // Function call to solve Tower of Hanoi
    towerOfHanoi(n, 'A', 'C', 'B');

    return 0;
}
```

```
Output
Enter the number of disks: 3
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.: B4

Program Description:

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

Solution:

```
// During each recursive call, it prints the current state of the
stack, helping you visualize how the function calls are stacked up.
#include <stdio.h>

// Function to calculate factorial using recursion
int factorial(int n)
{
    // Display the state of the stack
    printf("Calculating factorial(%d)\n", n);

    // Base case: factorial of 0 is 1
    if (n == 0 || n == 1)
    {
        printf("Base case reached: factorial(%d) = 1\n", n);
        return 1;
    }
    else
    {
```

```
        // Recursive case: factorial(n) = n * factorial(n-1)
        int result = n * factorial(n - 1);
        printf("factorial(%d) = %d * factorial(%d) = %d\n", n, n, n
- 1, result);
        return result;
    }
}

int main()
{
    int num;

    // Input: Number for which factorial needs to be calculated
    printf("Enter a number: ");
    scanf("%d", &num);

    // Calculate and display the factorial
    int result = factorial(num);
    printf("Factorial of %d = %d\n", num, result);

    return 0;
}
```

```
Output
Enter a number: 3
Calculating factorial(3)
Calculating factorial(2)
Calculating factorial(1)
Base case reached: factorial(1) = 1
factorial(2) = 2 * factorial(1) = 2
factorial(3) = 3 * factorial(2) = 6
Factorial of 3 = 6
```

Experiment No.: C1

Program Description:

Implementation of Queue using Array.

Solution:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100
typedef struct{
    int arr[MAX_SIZE];
    int front;
    int rear;
} Queue;
void initialize(Queue *q) {
    q->front = -1;
    q->rear = -1;
}
int isEmpty(Queue *q) {
    return q->front == -1;
}
int isFull(Queue *q) {
    return q->rear == MAX_SIZE - 1;
}
```

```
void enqueue(Queue *q, int item) {
    if (isFull(q)) {
        printf("Queue is full!\n");
    return;
    }
    if (isEmpty(q)) {
        q->front = 0;
    }
    q->arr[++(q->rear)] = item;
}
int dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        exit(1);
    }
    int dequeuedItem = q->arr[q->front];
    if (q->front == q->rear) {
        q->front = -1;
        q->rear = -1;
    }
    else {
        q->front = (q->front + 1) % MAX_SIZE;
    }
    return dequeuedItem;
}
void display(Queue *q) {
    if(isEmpty(q)) {
        printf("Queue is empty!\n");
        return;
    }
    int i;
    printf("Queue: ");
    for(i = q->front; i != q->rear; i = (i + 1) % MAX_SIZE) {
        printf("%d ", q->arr[i]);
    }
    printf("%d\n", q->arr[i]);
}
int main() {
    Queue q;
    initialize(&q);
    int choice, item;
    while(1){
        printf("\n\n***** MENU *****\n");
        printf("1. Insert an element\n2. Delete an element\n3.
Display.\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d",&choice);
        switch(choice){
            case 1: printf("Enter the value to be inserted: ");
                    scanf("%d",&item);
                    enqueue(&q,item);
                    break;
            case 2: dequeue(&q);
                    break;
            case 3: display(&q);
                    break;
            case 4:exit(0);
```

```
            default: printf("\nInvalid Choice.\n");
        }
    }
}
```

Output:

********MENU*******

1. Insert an element

2. Delete an element

3. Display.

4. Exit

Enter your choice:

Invalid Choice.

Experiment No.: C2

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{
    Node* front;
    Node* rear;
} Queue;
void initialize(Queue* q) {
    q->front = NULL;
    q->rear = NULL;
}
int isEmpty(Queue* q) {
```

```
    return q->front == NULL;
}
void enqueue(Queue* q, int item) {
    Node* newNode = (Node*) malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Queue overflow!\n");
        return;
    }
    newNode->data = item;
    newNode->next = NULL;
    if (isEmpty(q)) {
        q->front = newNode;
    }
    else{
        q->rear->next = newNode;
    }
    q->rear = newNode;
}
int dequeue(Queue* q) {
    if (isEmpty(q)) {
        printf("Queue underflow!\n");
        exit(1);
    }
    Node* temp = q->front;
    int dequeuedItem = temp->data;
    q->front = q->front->next;
    if (q->front == NULL) {
        q->rear = NULL;
    }
    free(temp);
    return dequeuedItem;
}
void display(Queue *q) {
    Node* temp = q->front;
    while(temp!= NULL){
        printf("%d",temp->data);
        temp=temp->next;
    }


}
int main(){
    Queue q;
    initialize(&q);
    int choice, item;
    while(1){
        printf("1. Insert element to queue \n");
        printf("2. Delete element from queue \n");
        printf("3. Display all elements of queue \n");
        printf("4. Exit \n");
        printf("Enter your choice : ");
        scanf("%d",&choice);
        switch(choice){
            case 1: printf("Enter a value to insert: ");
                    scanf("%d", &item);
                    enqueue(&q,item);
                    break;
```

```
            case 2: dequeue(&q);
                    break;
            case 3: display(&q);
                    break;
            case 4: exit(0);
            default: printf("Invalid Choice!\n");
        }
    }
    return 0;
}
```

output,

1. Insert element to queue

2. Delete element from queue

3. Display all elements of queue

4. Exit

Enter your choice :

Experiment No.: C3

Program Description:

Implementation of Circular Queue using Array.

Solution:

```
#include <stdio.h>
#include<stdlib.h>
#define MAX_SIZE 100
typedef struct{
    int arr[MAX_SIZE];
    int front;
    int rear;
} CircularQueue;
void initialize(CircularQueue *q) {
    q->front = -1;
    q->rear = -1;
}
int IsEmpty(CircularQueue *q){
    return q->front == -1;
```

```
}
int IsFull(CircularQueue *q){
    return q->rear == MAX_SIZE-1;
}
void enqueue(CircularQueue *q, int item) {
    if (isFull(q)) {
        printf("Queue is full!\n");
        return;
    }
    if (isEmpty(q)) {
        q->front = 0;
        q->rear = 0;
    }
    else{
        q->rear = (q->rear + 1) % MAX_SIZE;
    }
    q->arr[q->rear] = item;
}
int dequeue(CircularQueue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        exit(1);
    }
    int dequeuedItem = q->arr[q->front];
    if (q->front == q->rear) {
        q->front = -1;
        q->rear = -1;
    }
    else{
        q->front = (q->front + 1) % MAX_SIZE;
    }
    return dequeuedItem;
}
void display(CircularQueue *q) {
    if(IsEmpty(q)) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue: ");
    int i;
    for(i = q->front; i != q->rear; i = (i + 1) % MAX_SIZE) {
        printf("%d ", q->arr[i]);
    }
    printf("%d\n", q->arr[i]);
}
int main(){
    CircularQueue q;
    initialize(&q);
    int choice, item;
    while(1){
        printf("1. Insert element to queue \n");
        printf("2. Delete element from queue \n");
        printf("3. Display all elements of queue \n");
        printf("4. Exit \n");
        printf("Enter your choice : ");
        scanf("%d",&choice);
        switch(choice){
```

```
            case 1: printf("Enter a value to insert: ");
                    scanf("%d", &item);
                    enqueue(&q, item);
                    break;
            case 2: dequeue(&q);
                    break;
            case 3: display(&q);
                    break;
            case 4: exit(0);
            default: printf("Invalid \n");
        }
    }
    return 0;
}
```

```
output,
1. Insert element to queue
2. Delete element from queue
3. Display all elements of queue
4. Exit
Enter your choice : 1
Enter a value to insert: 6
```

Experiment No.: D1

Program Description:

Implementation of Binary Search Tree.

Solution:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node
{
int data;
struct Node* left;
struct Node* right;
} Node;
Node* createNode(int data) {
    Node* newNode = (Node*) malloc(sizeof(Node));
    if (!newNode) {
```

```
        printf("Memory error\n");
        exit(1);

    }
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;

}
Node* searchBST(Node* root, int data) {
    if (root == NULL || root->data == data)
    {
        return root;
    }
    if (data < root->data)
    {
        return searchBST(root->left, data);

    }
    return searchBST(root->right, data);
}
void insertBST(Node** root, int data)
{
    if (*root == NULL)
    {
        *root = createNode(data);
        return;

    }
    if (data < (*root)->data)
    {
        insertBST(&((*root)->left), data);

    }
    else if (data > (*root)->data)
    {
        insertBST(&((*root)->right), data);

    }
    else
    {
        printf("Successfully Done.\n");

    }
}
Node* findMinValueNode(Node* Node) {
    struct Node* current = Node;

    while (current && current->left != NULL)
        current = current->left;

    return current;
}
Node* deleteBST(Node* root, int data)
{
    if (!root)
```

```
        return root;
    if (data < root->data){
        root->left = deleteBST(root->left, data);

    }
    else if (data > root->data) {
        root->right = deleteBST(root->right, data);

    }
    else{
        if (!root->left)
        {
            Node* temp = root->right;
            free(root);
            return temp;

        }
        else if (!root->right) {
            Node* temp = root->left;
            free(root);
            return temp;

        }
        Node* temp = findMinValueNode(root->right);
        root->data = temp->data;
        root->right = deleteBST(root->right, temp->data);
    }
    return root;
}
void inorder(Node* temp) {
    if (temp == NULL)
        return;

    inorder(temp->left);
    printf("%d ", temp->data);
    inorder(temp->right);
}

int main() {
    Node* root = NULL;
    int choice, data;

    while(1) {
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                insertBST(&root, data);
                break;
            case 2:
```

```
                printf("Enter data to delete: ");
                scanf("%d", &data);
                deleteBST(root, data);
                break;
            case 3:
                printf("Displaying the tree in Inorder: ");
                inorder(root);
                printf("\n");
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }

    return 0;
}
```

Output ,1. Insert

2. Delete

3. Display

4. Exit

Enter your choice:

Experiment No.: D2

Program Description:

Conversion of BST PreOrder/PostOrder/InOrder.

Solution:

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

```
} Node;
Node* createNode(int data) {
    Node* newNode = (Node*) malloc(sizeof(Node));
    if (!newNode) {
        printf("Memory error\n");
        exit(1);

    }
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;

}

void insertBST(Node** root, int data)
{
    if (*root == NULL)
    {
        *root = createNode(data);
        return;

    }
    if (data < (*root)->data)
    {
        insertBST(&((*root)->left), data);

    }
    else if (data > (*root)->data)
    {
        insertBST(&((*root)->right), data);

    }
    else
    {
        printf("Successfully Done.\n");

    }
}
void inorder(Node* root)
{
    if (root == NULL)
    return;
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}
void preorder(Node* root)
{
    if (root == NULL)
    return;
    printf("%d ", root->data);
    preorder(root->left);
    preorder(root->right);
}
void postorder(Node* root)
{
```

```
    if (root == NULL)
        return;
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
}

int main() {
    Node* root = NULL;
    insertBST(&root, 50);
    insertBST(&root, 30);
    insertBST(&root, 20);
    insertBST(&root, 40);
    insertBST(&root, 70);
    insertBST(&root, 60);
    insertBST(&root, 80);

    printf("Preorder traversal: ");
    preorder(root);
    printf("\n");

    printf("Inorder traversal: ");
    inorder(root);
    printf("\n");

    printf("Postorder traversal: ");
    postorder(root);
    printf("\n");

    return 0;
}
```

Output:

Preorder traversal: 50 30 20 40 70 60 80

Inorder traversal: 20 30 40 50 60 70 80

Postorder traversal: 20 40 30 60 80 70 50

Experiment No.: D3

Program Description:

Implementation of Kruskal Algorithm

Solution:

```
#include <stdio.h>

#include <stdlib.h>
```

```
// Structure to represent an edge in the graph
struct Edge {
    int src, dest, weight;
};

// Structure to represent a subset for union-find
struct Subset {
    int parent;
    int rank;
};

// Function prototypes
int find(struct Subset subsets[], int i);
void Union(struct Subset subsets[], int x, int y);
int compare(const void* a, const void* b);
void Kruskal(struct Edge graph[], int V, int E);

int main() {
    // Example graph represented by edges and weights
    int V = 4; // Number of vertices
    int E = 5; // Number of edges
    struct Edge graph[] = {
        {0, 1, 10},
        {0, 2, 6},
        {0, 3, 5},
        {1, 3, 15},
        {2, 3, 4}
    };

    Kruskal(graph, V, E);

    return 0;
```

```
}

// Find set of an element i (uses path compression technique)
int find(struct Subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// Union of two sets x and y (uses union by rank)
void Union(struct Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(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++;
    }
}

// Compare function for qsort() to sort edges based on their weight
int compare(const void* a, const void* b) {
    return ((struct Edge*)a)->weight - ((struct Edge*)b)->weight;
}

// Kruskal's algorithm to find Minimum Spanning Tree
void Kruskal(struct Edge graph[], int V, int E) {
```

```
// Allocate memory for subsets
struct Subset* subsets = (struct Subset*)malloc(V * sizeof(struct Subset));

// Initialize subsets with single elements
for (int i = 0; i < V; i++) {
    subsets[i].parent = i;
    subsets[i].rank = 0;
}

// Sort the graph edges in non-decreasing order by weight
qsort(graph, E, sizeof(graph[0]), compare);

// Initialize result
struct Edge result[V];
int e = 0; // Index for result[]

// Iterate through all sorted edges
for (int i = 0; e < V - 1 && i < E; i++) {
    // Get the smallest edge and increment the index for next iteration
    struct Edge next_edge = graph[i];

    int x = find(subsets, next_edge.src);
    int y = find(subsets, next_edge.dest);

    // If including this edge doesn't cause a cycle, add it to the result
    if (x != y) {
        result[e++] = next_edge;
        Union(subsets, x, y);
    }
}

// Print the result
```

```
    printf("Edges in the Minimum Spanning Tree:\n");
    for (int i = 0; i < e; i++)
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);

    // Clean up memory
    free(subsets);
}
```

output

Edges in the Minimum Spanning Tree:

2 -- 3 == 4

0 -- 3 == 5

0 -- 1 == 10

Experiment No.: D4

Program Description:

Implementation of Prim Algorithm.

Solution:

```c
#include <stdio.h>
#include <limits.h>

#define 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]; // Array to store constructed MST
    int key[V];    // Key values used to pick minimum weight edge in cut
    int mstSet[V]; // To represent set of vertices not yet included in MST

    // Initialize all keys as INFINITE and mstSet[] as false
    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
```

```
        mstSet[i] = 0;
    }

    // Always include the first vertex in MST.
    key[0] = 0;     // Make key 0 so that this vertex is picked as the first vertex
    parent[0] = -1;  // First node is always the root of MST

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum key vertex from the set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST set
        mstSet[u] = 1;

        // Update key value and parent index of the adjacent vertices of the picked vertex
        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];
            }
        }
    }

    // Print the constructed MST
    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
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5
```

Experiment No.: D5

Program Description:

Implementation of Dijkstra Algorithm

Solution:

```
#include <stdio.h>
#include <limits.h>

#define V 6 // Number of vertices in the graph

// Function to find the vertex with the minimum distance value
// from the set of vertices not yet included in the shortest path tree
int minDistance(int dist[], int sptSet[]) {
    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;
}

// Function to print the constructed distance array
void printSolution(int dist[]) {
    printf("Vertex \tDistance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t%d\n", i, dist[i]);
}

// Function that implements Dijkstra's single-source shortest path algorithm
void dijkstra(int graph[V][V], int src) {
```

```
int dist[V];     // The output array dist[i] holds the shortest distance from src to i
int sptSet[V];   // sptSet[i] will be true if vertex i is included in the shortest
                 // path tree or the shortest distance from src to i is finalized

// Initialize all distances as INFINITE and sptSet[] as false
for (int i = 0; i < V; i++) {
    dist[i] = INT_MAX;
    sptSet[i] = 0;
}

// Distance from the source vertex to itself is always 0
dist[src] = 0;

// Find the shortest path for all vertices
for (int count = 0; count < V - 1; count++) {
    // Pick the minimum distance vertex from the set of vertices not yet processed
    int u = minDistance(dist, sptSet);

    // Mark the picked vertex as processed
    sptSet[u] = 1;

    // Update the distance value of the adjacent vertices of the picked vertex
    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];
        }
    }
}

// Print the constructed distance array
printSolution(dist);
```

```
}

int main() {
   int graph[V][V] = {
      {0, 1, 4, 0, 0, 0},
      {1, 0, 4, 2, 7, 0},
      {4, 4, 0, 3, 5, 0},
      {0, 2, 3, 0, 4, 6},
      {0, 7, 5, 4, 0, 7},
      {0, 0, 0, 6, 7, 0}
   };

   int source = 0; // Source vertex

   dijkstra(graph, source);

   return 0;
}
```

output

```
Vertex  Distance from Source
0       0
1       1
2       4
3       3
4       7
5       9
```

Experiment No.: E1A

Program Description:

Implementation of sorting.

Solution:

```c
#include <stdio.h>


void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

void printArray(int arr[], int size) {
    for (int 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("Original array: ");
    printArray(arr, n);

    bubbleSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Output

Original array: 64 25 1

<u>Experiment No.: E1B</u>

Program Description:

Implementation of sorting.

Solution:

```
#include <stdio.h>

// Function to perform selection sort on an array
void selectionSort(int arr[], int n) {
    int i;
    int j;
    int minIndex;
    int temp;
      for (i = 0; i < n - 1; i++) {

        minIndex = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
      }
}


void printArray(int arr[], int size) {
    for (int 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("Original array: ");
    printArray(arr, n);


    selectionSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);
```

```
    return 0;
}
```

Output:

Original array: 64 25 12 22 11

Sorted array: 11 12 22 25 64

Experiment No.: E1C

Program Description:

Implementation of sorting.

Solution:

```
#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) {
    for (int 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("Original array: ");
    printArray(arr, n);

    insertionSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Output :

Original array: 64 34 25 12 22 11 90

Experiment No.: E1D

Program Description:

Implementation of sorting.

Solution:

```
#include <stdio.h>

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++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return i + 1;
}
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 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[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);
```

```
    quickSort(arr, 0, n - 1);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Output :

Original array: 64 34 25 12 22 11 90

Sorted array: 11 12 22 25 34 64 90

Experiment No.: E1E

Program Description:

Implementation of sorting.

Solution:

```
#include <stdio.h>
#include <stdlib.h>

void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;
    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 l, int r) {
    if (l < r) {

        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
void printArray(int A[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

int main() {
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("Sorted array: ");
    printArray(arr, arr_size);

    return 0;
}
```

Output :

Original array: 38 27 43 3 9 82 10

Sorted array: 3 9 10 27 38 43 82

Experiment No.: E2

Program Description:

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

Solution:

```
#include <stdio.h>

int binarySearch(int arr[], int low, int high, int key) {
    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == key)
            return mid;
        else if (arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

int main() {
    int arr[] = {2, 5, 8, 12, 16, 23, 38, 42, 50};
    int n = sizeof(arr) / sizeof(arr[0]);

    int key = 23;
    int result = binarySearch(arr, 0, n - 1, key);

    if (result == -1)
        printf("Element %d is not present in the array.\n", key);
    else
        printf("Element %d is present at index %d.\n", key, result);

    return 0;
}
```

```
Output:

Element 6 found at index 5
```

Experiment No.: E3

Program Description:

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

Solution:

```
#include <stdio.h>
#include <string.h>

int stringBinarySearch(char arr[][50], int left, int right, const char *key) {
    while (left <= right) {
        int mid = left + (right - left) / 2;

        int compareResult = strcmp(arr[mid], key);

        if (compareResult == 0)
            return mid;

        if (compareResult < 0)
            left = mid + 1;

        else
            right = mid - 1;
    }
    return -1;
}

int main() {
    char strArray[][50] = {"anita", "aradhi", "kamlya", "anit", "anu",
"naman", "sonu", "shruti", "nimi"};
    int n = sizeof(strArray) / sizeof(strArray[0]);
    const char *key = "sonu";
    int result = stringBinarySearch(strArray, 0, n - 1, key);

    if (result == -1)
        printf("Element '%s' is not present in the array\n", key);
    else
        printf("Element '%s' is present at index %d\n", key, result);

    return 0;
}
```

Output:

Element 'sonu' is present at index 6

Experiment No.: E4

Program Description:

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

Solution:

```
#include <stdio.h>
#include <string.h>

int stringLinearSearch(char arr[][50], int n, const char *key) {
    for (int i = 0; i < n; i++) {

        if (strcmp(arr[i], key) == 0) {

            return i;
        }
    }


    return -1;
}

int main() {
    char strArray[][50] = {"anita", "aradhi", "kamlya", "anit", "Anu",
"Naman", "sonu", "shruti", "nimi"};
    int n = sizeof(strArray) / sizeof(strArray[0]);
    const char *key = "sonu";


    int result = stringLinearSearch(strArray, n, key);

    if (result == -1)
        printf("Element '%s' is not present in the array\n", key);
    else
        printf("Element '%s' is present at index %d\n", key, result);

    return 0;
}
```

Output:

Element 'sonu' is present at index 6

Experiment No.: E5

Program Description:

Implementation of Binary Search on a list of strings stored in a Single Linked List

Solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


struct Node {
    char data[50];
    struct Node* next;
};


struct Node* insertNode(struct Node* head, const char* data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    strncpy(newNode->data, data, sizeof(newNode->data));
    newNode->next = NULL;

    if (head == NULL) {
        return newNode;
    }

    struct Node* current = head;
    while (current->next != NULL) {
        current = current->next;
    }

    current->next = newNode;
    return head;
}


int stringLinearSearchLinkedList(struct Node* head, const char* target) {
    int index = 0;
    struct Node* current = head;

    while (current != NULL) {
```

```
        if (strcmp(current->data, target) == 0) {

            return index;
        }

        current = current->next;
        index++;
    }


    return -1;
}

int main() {

    struct Node* head = NULL;
    head = insertNode(head, "apple");
    head = insertNode(head, "banana");
    head = insertNode(head, "cherry");
    head = insertNode(head, "date");
    head = insertNode(head, "fig");
    head = insertNode(head, "grape");
    head = insertNode(head, "kiwi");
    head = insertNode(head, "orange");
    head = insertNode(head, "pear");

    const char* target = "kiwi";


    int result = stringLinearSearchLinkedList(head, target);

    if (result == -1)
        printf("Element '%s' is not present in the linked list.\n", target);
    else
        printf("Element '%s' is present at index %d in the linked list.\n",
target, result);


    struct Node* current = head;
    while (current != NULL) {
        struct Node* temp = current;
        current = current->next;
        free(temp);
    }

    return 0;
}
```

Output :

Element 'kiwi' is present at index 6 in the linked list.