LAB 2

Experiment 1

OBJECTIVE: To perform various operations on a stack using array implementation.

THEORY:

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. The basic operations performed on a stack are:

1. Push – Adds an element to the top of the stack.
2. Pop – Removes the top element from the stack.
3. Display – Shows all elements currently in the stack.

Stacks can be implemented using arrays or linked lists. In this lab, we use an array to implement the stack.

<u>Algorithm:</u>

1. Push Operation:
   - Check if the stack is full (Overflow condition).
   - If not full, increment the top pointer and insert the new element.

2. Pop Operation:
   - Check if the stack is empty (Underflow condition).
   - If not empty, remove the top element and decrement the top pointer.

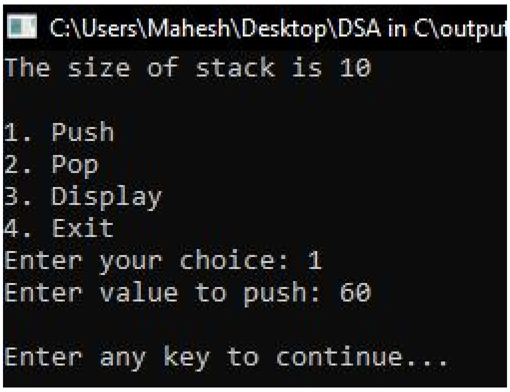
3. Display Operation:
   - If the stack is empty, display an appropriate message.
   - Otherwise, traverse the stack from top to bottom and print all elements.

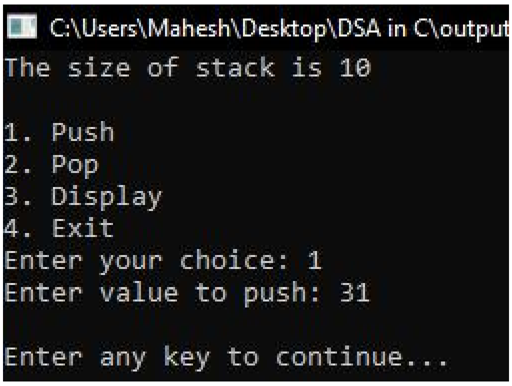
PROGRAMS

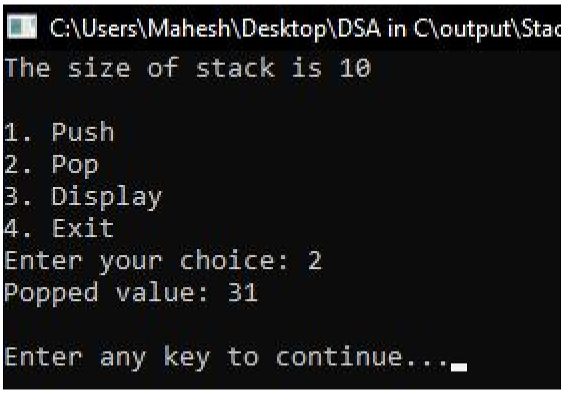
```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

int main()
{
    int stack[100], top = -1, n;
    int i, choice, value;
    printf("Enter the size of stack: ");
    scanf("%d", &n);
    do
    {
        system("cls");
        printf("The size of stack is %d \n", n);
        printf("\n1. Push\n2. Pop\n3. Display\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
            if (top == n - 1)
                printf("Stack Overflow\n");
            else
            {
                printf("Enter value to push: ");
                scanf("%d", &value);
                stack[++top] = value;
            }
            break;
        case 2:
            if (top == -1)
                printf("Stack Underflow\n");
            else
            {
                value = stack[top--];
                printf("Popped value: %d\n", value);
            }
            break;
        case 3:
            if (top == -1)
                printf("Stack is empty\n");
            else
```

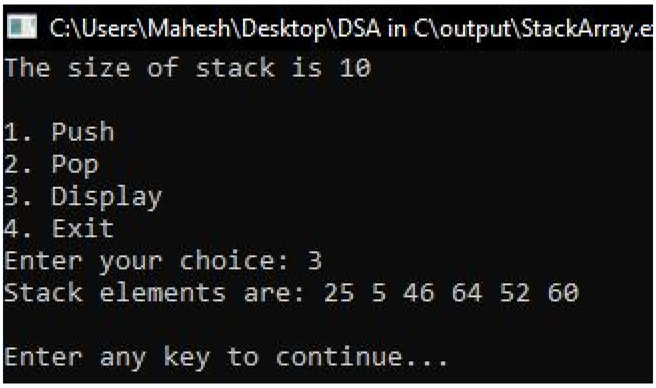
```
        {
            printf("Stack elements are: ");
            for (i = top; i >= 0; i--)
                printf("%d ", stack[i]);
            printf("\n");
        }
        break;
    case 4:
        printf("Exiting...\n");
        break;
            default:
                printf("Invalid choice\n");
            }
            printf("\nEnter any key to continue...");
            getch();
        } while (choice != 4);
        return 0;
    }
```

Output:









CONCLUSION:

In this lab, we successfully implemented stack operations using an array. We performed push, pop, and display operations and handled stack overflow and underflow conditions effectively. This implementation helped in understanding the fundamental working of the stack data structure.

Experiment 2

OBJECTIVE: To perform various operations on a stack with Linked List operation.

THEORY:

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. The main operations of a stack are:

- Push: Inserts an element onto the stack.
- Pop: Removes and returns the top element of the stack.
- Display: Shows all the elements in the stack.

A linked list-based stack dynamically allocates memory and does not require a predefined size. It consists of nodes where each node contains:

- data: The value stored in the node.
- next: A pointer to the next node.

Algorithm

Push Operation
1. Create a new node.
2. Assign the given value to the node.
3. Set the next pointer of the new node to the current top.
4. Update the top pointer to the new node.
5. Increment the stack size.

Pop Operation
1. Check if the stack is empty.
2. If not, store the top node's data.
3. Update the top pointer to the next node.
4. Free memory allocated to the previous top node.
5. Decrement the stack size.
6. Return the popped value.

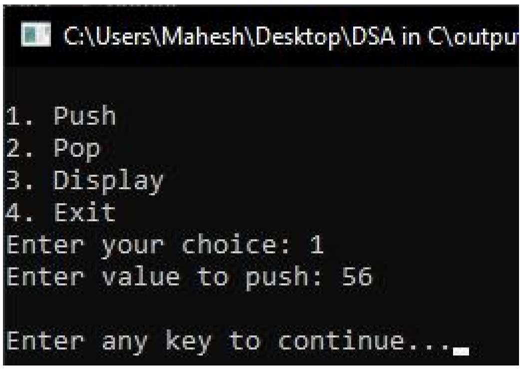
Display Operation
1. Traverse the stack from the top to the bottom.
2. Print each node's data.

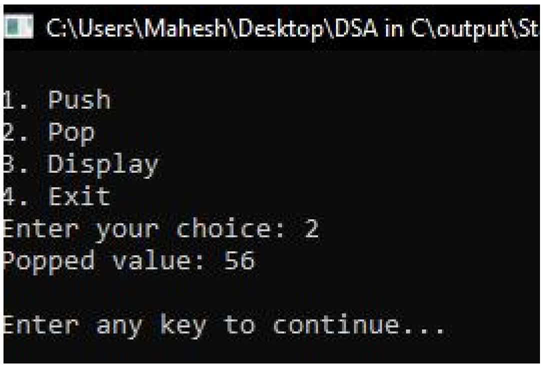
PROGRAMS

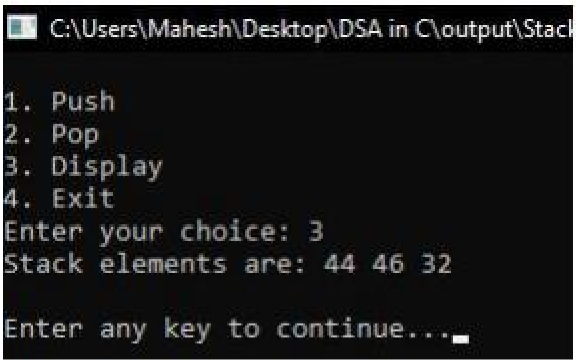
```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

struct Node
{
    int data;
    struct Node *next;
};
typedef struct Node Node;

Node *createNode(int data)
{
    Node *newNode = (Node
*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

struct LinkedListStack
{
    Node *top;
    int size;
};
typedef struct LinkedListStack LinkedListStack;

void push(LinkedListStack *stack, int data)
{
    Node *newNode = createNode(data);
    newNode->next = stack->top;
    stack->top = newNode;
    stack->size++;
}
int pop(LinkedListStack *stack)
{
    if (stack->top == NULL)
    {
        printf("Stack Underflow\n");
        return -1; // Indicate stack is empty
    }
    Node *temp = stack->top;
    int poppedValue = temp->data;
    stack->top = stack->top->next;
    free(temp);
    stack->size--;
    return poppedValue;
}

void display(LinkedListStack *stack)
```

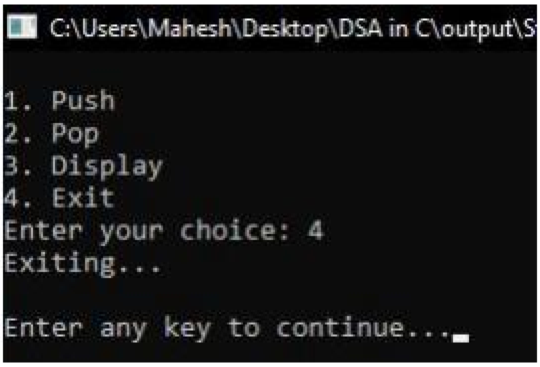
```
{
    if (stack->top == NULL)
    {
        printf("Stack is empty\n");
        return;
    }
    Node *temp = stack->top;
    printf("Stack elements are: ");
    while (temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
int main()
{
    LinkedListStack stack;
    stack.size = 0;
    stack.top = NULL;

    int choice, value;
    int poppedValue;
    do
    {
        system("cls");
        printf("\n1. Push\n2. Pop\n3. Display\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
            printf("Enter value to push: ");
            scanf("%d", &value);
            push(&stack, value);
            break;
        case 2:
            poppedValue = pop(&stack); // Use the declared variable
            if (poppedValue != -1)
                printf("Popped value: %d\n", poppedValue);
            break;
        case 3:
            display(&stack);
            break;
        case 4:
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice\n");
        }
        printf("\nEnter any key to continue...");
        getch();
    } while (choice != 4);
    return 0;
}
```

Output:









CONCLUSION:

In this lab, we successfully implemented a stack using a linked list. We performed operations such as push, pop, and display, demonstrating the Last In, First Out (LIFO) principle. This approach allows dynamic memory allocation, eliminating the limitations of a fixed-size stack array.

Experiment 3

OBJECTIVE: To convert an infix expression into a postfix expression using stack.

THEORY:

An infix expression is a mathematical notation where operators are placed between operands (e.g., A + B). A postfix expression places operators after the operands (e.g., AB+). Postfix notation eliminates the need for parentheses, making evaluation easier.

To convert an infix expression to a postfix expression, we use a stack to manage operator precedence and parentheses. The main steps involve:

- Scanning the expression from left to right.
- Pushing operators onto a stack while maintaining precedence order.
- Popping operators when encountering lower precedence operators or closing parentheses.
- Appending operands directly to the postfix expression.

Algorithm

Infix to Postfix Conversion

1. Read the infix expression character by character.
2. If the character is an operand (A-Z, a-z), append it to the postfix expression.
3. If the character is an opening parenthesis (, push it onto the stack.
4. If the character is a closing parenthesis), pop from the stack and append to postfix until (is encountered.
5. If the character is an operator:
   - Pop operators from the stack to postfix if they have higher or equal precedence.
   - Push the current operator onto the stack.
6. After scanning, pop any remaining operators from the stack to the postfix expression.

PROGRAMS

```
#include<stdio.h>
#include<conio.h>
#include<stdbool.h>
#define MAX 50

char stack[MAX];
int tos = -1;

bool isempty(){
    return tos == -1;
}
void push(char ch){
    stack[++tos] = ch;
}
char pop(){
    if(!isempty())
        return stack[tos--];
    return ' ';
}
char peek(){
    return stack[tos];
}
```

```
int precedende(char op){
    if(op=='!') return 1; //highest precedence
    if(op=='^' || op=='$') return 2;
    if(op=='*' || op=='/' || op=='%') return 3;
    if(op=='+' || op=='-') return 4;
    else return 5;
}

bool CheckPrecedende(char op1, char op2){
    return precedende(op1) < precedende(op2);
}

int whatCharacterIs(char ch){
    if((ch>=65 && ch<=90) || (ch>=97 &&
ch<=122))
        return 1; // if the character is Letter.

    if(ch=='(') return 2;
    if(ch==')') return 3;
```

```
    if(ch=='+' || ch=='-' || ch=='*' || ch=='/' ||
ch=='$' || ch=='^' || ch=='%')
        return 4;

    return 0;
}

void TableRow(char infix[], char postfix[], int i,
int j){
    if(infix[i]=='\0') printf("\n\t...");
    else printf("\n\t%c",infix[i]);

    printf("\t\t");
        if(isempty()) printf("...");
        else
            for(int a=0; a<=tos;a++)
                printf("%c",stack[a]);

    printf("\t\t");
    if(j==0) printf("...");
    else
        for(int a=0; a<j; a++)
            printf("%c",postfix[a]);
}

int main(){
    char infix[MAX],postfix[MAX];
    int i=0;
    char ch;

    printf("Enter the Infix expression:\n");
    do{
        ch = getch();
        printf("%c",ch);

        if(ch != 13)
            infix[i++] = ch;

    }while(ch != 13);
    infix[i] = '\0';

    //print table
    printf("\n\nProcess Table\n");
    printf("\n\tInput\t\tStack\t\tPostfix\n");

    int j = 0;
    for(i=0; infix[i] != '\0'; i++){
        switch(whatCharacterIs(infix[i])){
            case 1:
                postfix[j++] = infix[i];
                break;
            case 2:
                push(infix[i]);
                break;
            case 3:
                while(peek()!='('
&& !isempty()){
                    postfix[j++] = pop();
                }
                if(!isempty()) pop();
                break;
            case 4:
                if(isempty() || peek()=='(')
                    push(infix[i]);
                else{

                    if(CheckPrecedende(peek(),i
nfix[i])){
                        postfix[j++] = pop();
                        push(infix[i]);
                    }else{
                        push(infix[i]);
                    }
                }

                break;
        }

        TableRow(infix,postfix,i,j);
    }
    while(!isempty()){
        if(peek()!='(')
            postfix[j++] = pop();
        else
            pop();
        TableRow(infix,postfix,i,j);
    }
    TableRow(infix,postfix,i,j);

    postfix[j] = '\0';

    printf("\n\nPostfix Expression is :\n");
    for(i=0; postfix[i]!='\0'; i++)
        printf("%c",postfix[i]);

    getch();
    return 0;
}
```

Output:

```
C:\Users\Mahesh\Desktop\DSA in C\output\InfixToPostfix.exe
((A-(B+C))*D)$(E+F)

Process Table

        Input           Stack           Postfix

        (               (               ...
        (               ((              ...
        A               ((              A
        -               ((-             A
        (               ((-(            A
        B               ((-(            AB
        +               ((-(+           AB
        C               ((-(+           ABC
        )               ((-             ABC+
        )               (               ABC+-
        *               (*              ABC+-
        D               (*              ABC+-D
        )               ...             ABC+-D*
        $               $               ABC+-D*
        (               $(              ABC+-D*
        E               $(              ABC+-D*E
        +               $(+             ABC+-D*E
        F               $(+             ABC+-D*EF
        )               $               ABC+-D*EF+
        ...             ...             ABC+-D*EF+$
        ...             ...             ABC+-D*EF+$

Postfix Expression is :
ABC+-D*EF+$
```

CONCLUSION:

In this lab, we successfully converted an infix expression to a postfix expression using a stack. This conversion helps eliminate parentheses and simplifies expression evaluation in computer systems.

Experiment 4

OBJECTIVE: To evaluate a postfix expression using a stack.

THEORY:

A postfix expression is evaluated using a stack by processing the expression from left to right:

1. If the character is an operand, push it onto the stack.
2. If the character is an operator, pop the top two elements, perform the operation, and push the result back onto the stack.
3. The final result is the only value left in the stack.

Algorithm

1. Read the postfix expression character by character.
2. If the character is an operand, push it onto the stack.
3. If the character is an operator:
   - Pop the top two elements.
   - Perform the operation.
   - Push the result back onto the stack.
4. The final result is the only value left in the stack.

PROGRAMS

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#define MAX 50

int main()
{
    int vstack[MAX];
    int tos = -1;

    char postfix[MAX];
    int i = 0;

    char ch;
    printf("Enter the postfix expression:\n");
    while (1)
    {
        ch = getch();
        printf(" %c", ch);
        if (ch == 13)
            break;

        postfix[i++] = ch;
    }
    postfix[i] = '\0';
    printf("\n");

    printf("\nvalue\top1\top2\tresult\tV-
Stack\n");
printf("---------------------------------------\n");

int op1, op2, result;
for (i = 0; postfix[i] != '\0'; i++)
{
    op1 = -1;
    if (postfix[i] >= '0' && postfix[i] <= '9')
        vstack[++tos] = postfix[i] - '0';
    else
    {
        op1 = vstack[tos--];
        op2 = vstack[tos--];
        switch (postfix[i])
        {
        case '+':
            result = op2 + op1;
            break;
        case '-':
            result = op2 - op1;
            break;
        case '*':
            result = op2 * op1;
            break;
        case '/':
            result = op2 / op1;
            break;
        case '^':
            result = pow(op2, op1);
            break;
```

```
        }
        vstack[++tos] = result;
    }

    if (op1 != -1)
        printf("%c\t%d\t%d\t%d\t",
postfix[i], op1, op2, result);
        else
        printf("%c\t-\t-\t-\t", postfix[i]);

        for (int j = 0; j <= tos; j++)
        {
            printf("%d ", vstack[j]);
        }
        printf("\n");
    }

    printf("\nThe value is: %d", vstack[tos]);
    getch();
}
```

Output:

```
C:\Users\Mahesh\Desktop\DSA in C\output\PostfixEvaluation.exe
Enter the postfix expression:
 1 2 3 + * 3 2 1 - + *

value   op1     op2     result  V-Stack
----------------------------------------
1       -       -       -       1
2       -       -       -       1 2
3       -       -       -       1 2 3
+       3       2       5       1 5
*       5       1       5       5
3       -       -       -       5 3
2       -       -       -       5 3 2
1       -       -       -       5 3 2 1
-       1       2       1       5 3 1
+       1       3       4       5 4
*       4       5       20      20

The value is: 20
```

CONCLUSION:

In this lab, we successfully evaluated a postfix expression using a stack. These techniques are fundamental in expression evaluation in computer systems, providing efficiency and clarity in computation.

LAB 3

Experiment 1

OBJECTIVE: To perform various operations in a queue using array implementation.

THEORY:

A queue is a linear data structure that follows the First In, First Out (FIFO) principle. The element inserted first is removed first. The main operations on a queue are:

1. Enqueue (Insertion): Adds an element to the rear of the queue.
2. Dequeue (Deletion): Removes an element from the front of the queue.
3. Display: Shows all elements in the queue.
4. isEmpty: Checks if the queue is empty.
5. isFull: Checks if the queue is full.

Queue Representation in Array

- front points to the first element.
- rear points to the last inserted element.
- If rear == MAX - 1, the queue is full.
- If front > rear or front == -1, the queue is empty.

ALGORITHM:

Enqueue Operation:

1. Check if the queue is full.
2. If not full, increment rear and insert the element.
3. If inserting the first element, set front = 0.

Dequeue Operation:

1. Check if the queue is empty.
2. If not empty, print and remove the front element.
3. Increment front.

Display Operation:

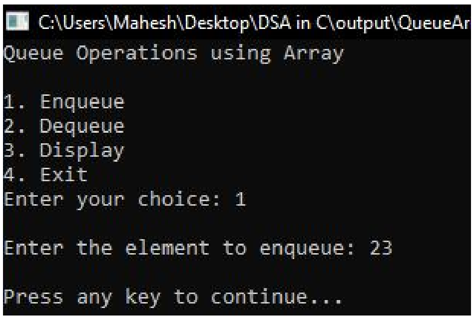
1. If empty, print "Queue is empty".
2. Otherwise, print elements from front to rear.

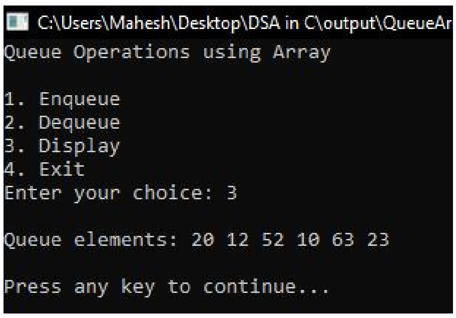
PROGRAMS

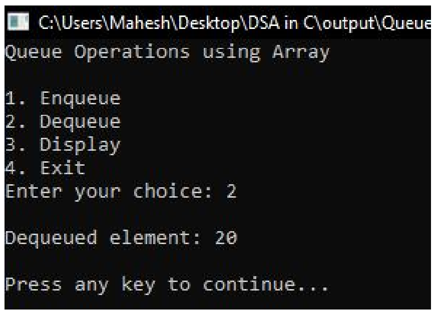
```
#include<stdio.h>
#include<conio.h>
#define MAX 10

int queue[MAX], front = -1, rear = -1;

int isFull() {
    return rear == MAX - 1;
}
int isEmpty() {
    return front == -1 || front > rear;
}

void enqueue(int item) {
    if (isFull()) {
        printf("Queue is full\n");
    }
    else {
        if (front == -1) front = 0;
        rear++;
        queue[rear] = item;
    }
}
void dequeue() {
```

```
    if (isEmpty()) {
        printf("Queue is empty\n");
    }
    else {
        printf("Dequeued element: %d\n",
queue[front]);
        front++;
    }
}
void display() {
    if (isEmpty()) {
        printf("Queue is empty\n");
    }
    else {
        printf("Queue elements: ");
        for (int i = front; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
        printf("\n");
    }
}

int main() {
    int choice, item;

    while (1) {
        system("cls");
        printf("Queue Operations using
Array\n");
        printf("\n1. Enqueue\n2. Dequeue\n3.
Display\n4. Exit\nEnter your choice: ");
        scanf("%d", &choice);
        printf("\n");
        switch (choice) {
        case 1:
            printf("Enter the element to enqueue:
");

            scanf("%d", &item);
            enqueue(item);
            break;
        case 2:
            dequeue();
            break;
        case 3:
            display();
            break;
        case 4:
            return 0;
        default:
            printf("Invalid choice\n");
        }
        printf("\nPress any key to continue...\n");
        getch();
    }
    return 0;
    getch();
}
```

Output:







CONCLUSION:

In this lab, we implemented queue operations using an array. The enqueue and dequeue functions worked as expected, following FIFO order. This demonstrates how queues can be efficiently managed using arrays.

Experiment 2

OBJECTIVE:

To perform various operations on a Queue with Linked List implementation.

THEORY:

A queue is a linear data structure that follows the First In, First Out (FIFO) principle. The element inserted first is removed first. The main operations on a queue are:

1. Enqueue (Insertion): Adds an element to the rear of the queue.
2. Dequeue (Deletion): Removes an element from the front of the queue.
3. Display: Shows all elements in the queue.
4. isEmpty: Checks if the queue is empty.

Queue Representation using Linked List

- The queue is implemented using a linked list where each node contains data and a pointer to the next node.
- front points to the first node in the queue.
- rear points to the last node in the queue.
- If front == NULL, the queue is empty.

Algorithm

Enqueue Operation:

1. Create a new node.
2. If the queue is empty, set front = rear = new node.
3. Else, set rear->next = new node and update rear.

Dequeue Operation:

1. Check if the queue is empty.
2. If not empty, print and remove the front node.
3. Update front to the next node.

Display Operation:

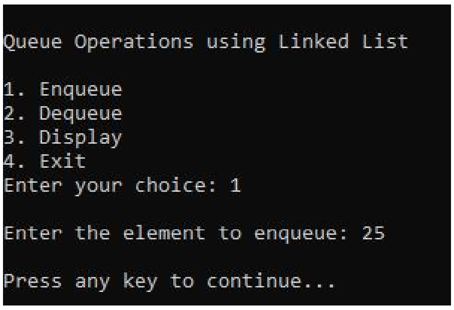
1. If empty, print "Queue is empty".
2. Otherwise, traverse from front to rear and print elements.

PROGRAMS

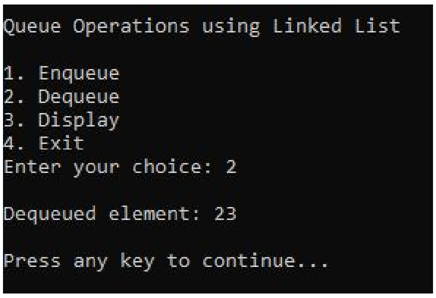
```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

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

typedef struct Node Node;
Node* front = NULL;
Node* rear = NULL;

int isEmpty() {
    return front == NULL;
}

void enqueue(int item) {
    Node* newNode =
(Node*)malloc(sizeof(Node));
    newNode->data = item;
    newNode->next = NULL;
    if (rear == NULL) {
        front = rear = newNode;
    } else {
        rear->next = newNode;
```

```
        rear = newNode;
    }
}

void dequeue() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }
    Node* temp = front;
    printf("Dequeued element: %d\n", front-
>data);
    front = front->next;
    if (front == NULL) rear = NULL;
    free(temp);
}

void display() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }
    Node* temp = front;
    printf("Queue elements: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    int choice, item;
    while (1) {
        system("cls");
        printf("\nQueue Operations using
Linked List\n\n");
        printf("1. Enqueue\n2. Dequeue\n3.
Display\n4. Exit\nEnter your choice: ");
        scanf("%d", &choice);
        printf("\n");
        switch (choice) {
        case 1:
            printf("Enter the element to enqueue:
");

            scanf("%d", &item);
            enqueue(item);
            break;
        case 2:
            dequeue();
            break;
        case 3:
            display();
            break;
        case 4:
            return 0;
        default:
            printf("Invalid choice\n");
        }
        printf("\nPress any key to continue...\n");
        getch();
    }
    return 0;
}
```

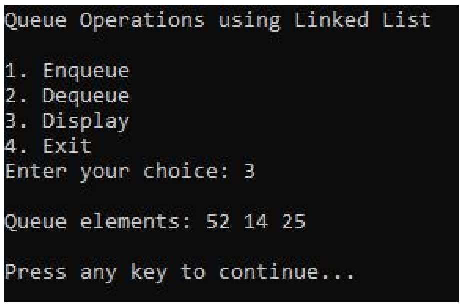
Output:



```
Queue Operations using Linked List

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1

Enter the element to enqueue: 25

Press any key to continue...
```



```
Queue Operations using Linked List

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2

Dequeued element: 23

Press any key to continue...
```



```
Queue Operations using Linked List

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3

Queue elements: 52 14 25

Press any key to continue...
```

CONCLUSION:

In this lab, we implemented queue operations using a linked list. The enqueue and dequeue functions worked as expected, following FIFO order. This demonstrates how queues can be efficiently managed using linked lists.

Experiment 3

OBJECTIVE:

To perform various operations in a circular queue using array implementation.

THEORY:

A circular queue is a linear data structure that follows the First In, First Out (FIFO) principle but connects the end of the queue back to the front to utilize unused space efficiently.

Features of Circular Queue:

1. Efficient Space Utilization: Unlike a linear queue, it does not waste space after elements are dequeued.
2. Circular Structure: The rear wraps around when it reaches the last index.
3. Key Operations:
   - Enqueue: Adds an element at the rear.
   - Dequeue: Removes an element from the front.
   - Display: Shows all elements in the queue.
   - isFull: Checks if the queue is full.
   - isEmpty: Checks if the queue is empty.

<u>Algorithm</u>

Enqueue Operation:

1. Check if the queue is full.
2. If empty, set front = 0.
3. Increment rear circularly using (rear + 1) % MAX.
4. Insert the new element at rear.

Dequeue Operation:

1. Check if the queue is empty.
2. Print and remove the front element.
3. If only one element was left, reset front and rear to -1.
4. Otherwise, update front using (front + 1) % MAX.

Display Operation:

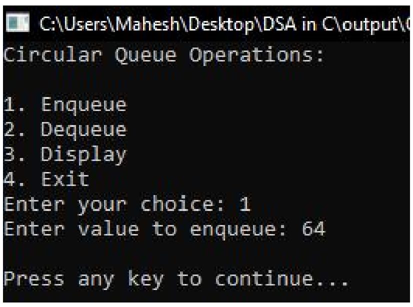
1. If empty, print "Queue is empty".
2. Traverse from front to rear circularly and print elements.

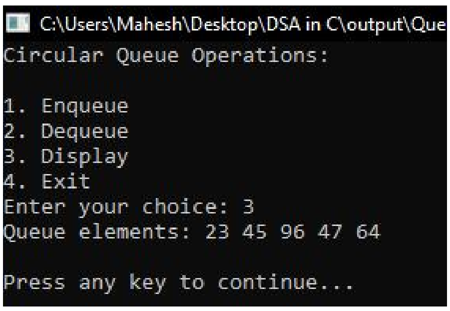
PROGRAMS

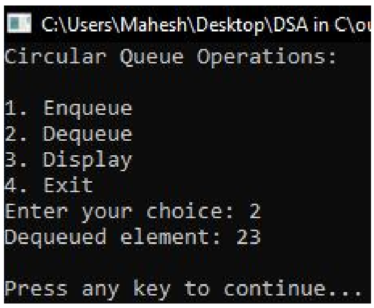
```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#define MAX 10

int queue[MAX], front = -1, rear = -1;

int isFull() {
    return (front == (rear + 1) % MAX);
}

int isEmpty() {
    return (front == -1);
}

void enqueue(int value) {
    if (isFull()) {
        printf("Queue is full\n");
        return;
    }
    if (isEmpty()) {
        front = 0;
```

```
    }
    rear = (rear + 1) % MAX;
    queue[rear] = value;
}

void dequeue() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }
    printf("Dequeued element: %d\n", queue[front]);
    if (front == rear) {
        front = -1;
        rear = -1;
    } else {
        front = (front + 1) % MAX;
    }
}

void display() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = front; i != rear; i = (i + 1) % MAX) {
        printf("%d ", queue[i]);
    }
    printf("%d\n", queue[rear]); // Print the last element
}

int main() {
    int choice, value;
    while (1) {
        system("cls");
        printf("Circular Queue Operations:\n\n");
        printf("1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                enqueue(value);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
        printf("\nPress any key to continue...\n");
        getch();
    }
    return 0;
}
```

Output:







CONCLUSION:

In this lab, we implemented a circular queue using an array. The enqueue and dequeue operations worked efficiently, demonstrating the advantage of circular queues in avoiding wasted space compared to linear queues.

Experiment 4

OBJECTIVE: To perform various operations in a priority queue, implementing both Min and Max Priority Queues using arrays.

THEORY:

A Priority Queue is a special type of queue in which elements are inserted based on their priority. The two types of priority queues are:

1. Min Priority Queue: The element with the lowest value has the highest priority and is dequeued first.
2. Max Priority Queue: The element with the highest value has the highest priority and is dequeued first.

Key Operations:

- Enqueue: Inserts an element in the correct position based on priority.
- Dequeue: Removes the highest (or lowest) priority element.
- Display: Shows elements in priority order.
- isEmpty: Checks if the queue is empty.

Algorithm

Enqueue (Insertion):

- Insert the element at the rear.
- Sort the queue based on priority (ascending for Min Queue, descending for Max Queue).

Dequeue (Deletion):

- Remove the first element (highest priority).

Display:

- Print elements from front to rear.

PROGRAMS

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define MAX 10

int minQueue[MAX], maxQueue[MAX];
int minFront = -1, minRear = -1;
int maxFront = -1, maxRear = -1;

int isEmpty(int front) {
    return front == -1;
}

void minEnqueue(int value) {
    if (minRear == MAX - 1) {
        printf("Min Priority Queue is full\n");
        return;
    }
    if (isEmpty(minFront)) {
        minFront = 0;
    }
    minRear++;
    minQueue[minRear] = value;
    for (int i = minRear; i > minFront &&
minQueue[i] < minQueue[i - 1]; i--) {
        int temp = minQueue[i];
        minQueue[i] = minQueue[i - 1];
        minQueue[i - 1] = temp;
    }
}

void minDequeue() {
    if (isEmpty(minFront)) {
        printf("Min Priority Queue is empty\n");
        return;
    }
    printf("Dequeued Min: %d\n",
minQueue[minFront]);
    if (minFront == minRear) minFront =
minRear = -1;
```

```c
    else minFront++;
}

void maxEnqueue(int value) {
    if (maxRear == MAX - 1) {
        printf("Max Priority Queue is full\n");
        return;
    }
    if (isEmpty(maxFront)) {
        maxFront = 0;
    }
    maxRear++;
    maxQueue[maxRear] = value;
    for (int i = maxRear; i > maxFront &&
maxQueue[i] > maxQueue[i - 1]; i--) {
        int temp = maxQueue[i];
        maxQueue[i] = maxQueue[i - 1];
        maxQueue[i - 1] = temp;
    }
}

void maxDequeue() {
    if (isEmpty(maxFront)) {
        printf("Max Priority Queue is empty\n");
        return;
    }
    printf("Dequeued Max: %d\n",
maxQueue[maxFront]);
    if (maxFront == maxRear) maxFront =
maxRear = -1;
    else maxFront++;
}

void display(int queue[], int front, int rear) {
    if (isEmpty(front)) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = front; i <= rear; i++) {
        printf("%d ", queue[i]);
    }
    printf("\n");
}

int main() {
    int choice, value;
    while (1) {
        system("cls");
        printf("\nPriority Queue
Operations:\n\n");
        printf("1. Min Enqueue\n2. Min
Dequeue\n3. Display Min Queue\n");
        printf("4. Max Enqueue\n5. Max
Dequeue\n6. Display Max Queue\n7. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        printf("\n");
        switch (choice) {
            case 1:
                printf("Enter value to enqueue
(Min Priority): ");
                scanf("%d", &value);
                minEnqueue(value);
                break;
            case 2:
                minDequeue();
                break;
            case 3:
                display(minQueue, minFront,
minRear);
                break;
            case 4:
                printf("Enter value to enqueue
(Max Priority): ");
                scanf("%d", &value);
                maxEnqueue(value);
                break;
            case 5:
                maxDequeue();
                break;
            case 6:
                display(maxQueue, maxFront,
maxRear);
                break;
            case 7:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
        printf("\nPress any key to continue...\n");
        getch();
    }
    return 0;
}
```

<u>Output:</u>

```
Priority Queue Operations:

1. Min Enqueue
2. Min Dequeue
3. Display Min Queue
4. Max Enqueue
5. Max Dequeue
6. Display Max Queue
7. Exit
Enter your choice: 1

Enter value to enqueue (Min Priority): 45

Press any key to continue...
```

```
Priority Queue Operations:

1. Min Enqueue
2. Min Dequeue
3. Display Min Queue
4. Max Enqueue
5. Max Dequeue
6. Display Max Queue
7. Exit
Enter your choice: 3

Queue elements: 12 45 52 63

Press any key to continue...
```

```
Priority Queue Operations:

1. Min Enqueue
2. Min Dequeue
3. Display Min Queue
4. Max Enqueue
5. Max Dequeue
6. Display Max Queue
7. Exit
Enter your choice: 2

Dequeued Min: 12

Press any key to continue...
```

```
Priority Queue Operations:

1. Min Enqueue
2. Min Dequeue
3. Display Min Queue
4. Max Enqueue
5. Max Dequeue
6. Display Max Queue
7. Exit
Enter your choice: 4

Enter value to enqueue (Max Priority): 16
```

```
Priority Queue Operations:

1. Min Enqueue
2. Min Dequeue
3. Display Min Queue
4. Max Enqueue
5. Max Dequeue
6. Display Max Queue
7. Exit
Enter your choice: 6

Queue elements: 97 63 54 52 16

Press any key to continue...
```

```
Priority Queue Operations:

1. Min Enqueue
2. Min Dequeue
3. Display Min Queue
4. Max Enqueue
5. Max Dequeue
6. Display Max Queue
7. Exit
Enter your choice: 5

Dequeued Max: 97

Press any key to continue...
```

CONCLUSION:

In this lab, we implemented a Priority Queue using arrays. The Min Priority Queue removes the smallest element first, while the Max Priority Queue removes the largest element first. This approach is useful in scheduling and real-time processing applications.

LAB 4

Experiment 1

OBJECTIVE: To perform and implement various iterative sorting algorithms:

- Bubble Sort
- Insertion Sort
- Selection Sort

THEORY:

Sorting is a fundamental operation in computer science used to arrange data in a particular order (ascending or descending). This lab demonstrates three common iterative sorting techniques which are simple and easy to understand.

ALGORITHM:

1. Bubble Sort

Bubble Sort compares adjacent elements and swaps them if they are in the wrong order. This process continues until the array is sorted.

<u>Key Points:</u>

- Time Complexity: $O(n^2)$
- Best Case (already sorted): $O(n)$
- Worst Case: $O(n^2)$

2. Insertion Sort

Insertion Sort builds the final sorted array one element at a time. It removes one element from the input data, finds the location it belongs to, and inserts it there.

<u>Key Points:</u>

- Time Complexity: $O(n^2)$
- Best Case (already sorted): $O(n)$
- Worst Case: $O(n^2)$

3. Selection Sort

Selection Sort divides the input into a sorted and unsorted region. It repeatedly selects the smallest (or largest) element from the unsorted region and moves it to the end of the sorted region.

Key Points:

- Time Complexity: $O(n^2)$
- Best Case: $O(n^2)$
- Worst Case: $O(n^2)$

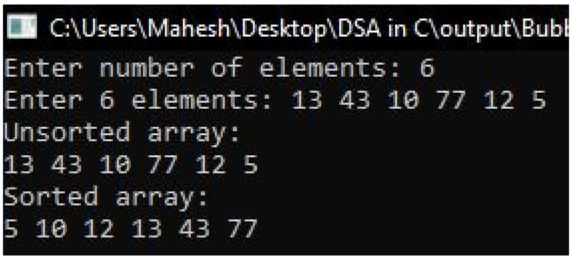
PROGRAMS

1. **Bubble Sort :**

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

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

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
```

```
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[100], n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    printf("\nUnsorted array:\n");
    printArray(arr, n);
    BubbleSort(arr, n);
    printf("\nSorted array:\n");
    printArray(arr, n);
    getch();
    return 0;
}
```

Output:

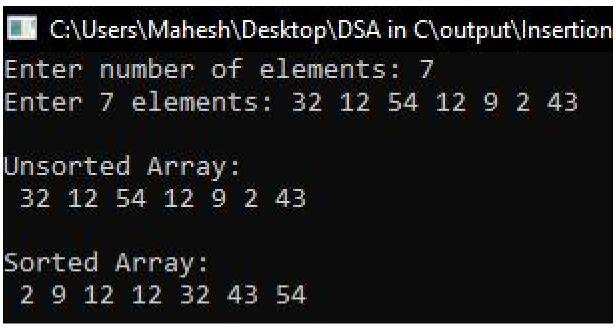


```
C:\Users\Mahesh\Desktop\DSA in C\output\Bubb
Enter number of elements: 6
Enter 6 elements: 13 43 10 77 12 5
Unsorted array:
13 43 10 77 12 5
Sorted array:
5 10 12 13 43 77
```

2. Insertion Sort:

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

void InsertionSort(int arr[], int n) {
    int temp, i, j;
    for (i = 1; i < n; i++) {
        temp = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > temp) {
            arr[j + 1] = arr[j];
            j--;
        }
        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[100], n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    printf("\nUnsorted Array:\n");
    printArray(arr, n);
    InsertionSort(arr, n);
    printf("\nSorted Array:\n");
    printArray(arr, n);
    getch();
    return 0;
}
```

Output:

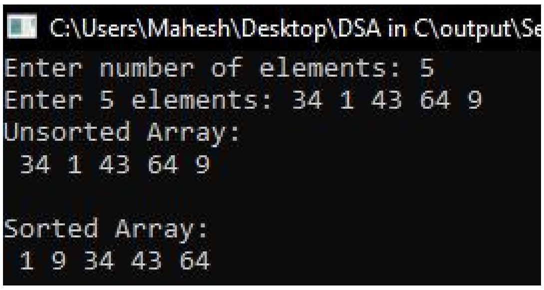


```
C:\Users\Mahesh\Desktop\DSA in C\output\Insertion
Enter number of elements: 7
Enter 7 elements: 32 12 54 12 9 2 43

Unsorted Array:
 32 12 54 12 9 2 43

Sorted Array:
 2 9 12 12 32 43 54
```

3. Selection Sort :

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

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

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

int main() {
	int arr[100], n;
	printf("Enter number of elements: ");
	scanf("%d", &n);
	printf("Enter %d elements: ", n);
	for (int i = 0; i < n; i++)
		scanf("%d", &arr[i]);

	printf("\nUnsorted Array:\n");
	printArray(arr, n);
	SelectionSort(arr, n);
	printf("\nSorted Array:\n");
	printArray(arr, n);
	getch();
	return 0;
}
```

Output:



CONCLUSION:

The lab helped understand the working of iterative sorting algorithms. All three methods successfully sorted the input data. Among these, Insertion Sort performs better on smaller or partially sorted arrays, while Bubble Sort and Selection Sort are less efficient for larger datasets.

Experiment 2

OBJECTIVE: To perform and implement various recursive sorting algorithms:

- Merge Sort
- Quick Sort

THEORY:

Sorting is a key operation in computer science used to organize data in a particular sequence. Recursive sorting algorithms use the principle of divide and conquer to sort data by breaking the problem into smaller sub-problems and solving them recursively.

ALGORITHM:

1. Merge Sort

Merge Sort divides the array into two halves, recursively sorts them, and then merges the sorted halves.

Steps:

i. Divide the array into two halves.
ii. Recursively sort the sub-arrays.
iii. Merge the sorted halves.

Time Complexity:

- Best Case: O(n log n)
- Average Case: O(n log n)
- Worst Case: O(n log n

2. Quick Sort

Quick Sort picks a pivot element, partitions the array into elements less than and greater than the pivot, and recursively sorts the sub-arrays.

Steps:

i. Select a pivot.
ii. Partition the array around the pivot.
iii. Recursively apply the same steps to the sub-arrays.

Time Complexity:

- Best Case: O(n log n)
- Average Case: O(n log n)
- Worst Case: $O(n^2)$

PROGRAMS

1. Merge Sort

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

void Merge(int arr[], int start, int mid, int end) {
	int i = start, j = mid + 1, k = start;
	int temp[end - start + 1];

	while (i <= mid && j <= end) {
		if (arr[i] < arr[j]) temp[k++] =
arr[i++];
		else temp[k++] = arr[j++];
	}

	while (i <= mid) temp[k++] = arr[i++];
	while (j <= end) temp[k++] = arr[j++];

	for (int i = start; i <= end; i++)
		arr[i] = temp[i];
}

void MergeSort(int arr[], int start, int end) {
	if (start < end) {
		int mid = (start + end) / 2;
		MergeSort(arr, start, mid);
		MergeSort(arr, mid + 1, end);
		Merge(arr, start, mid, end);
	}
}

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

int main() {
	int arr[] = {5, 3, 76, 1, 53};
	int n = sizeof(arr) / sizeof(arr[0]);

	printf("Before Sort:\n");
```

```
    printArray(arr, n);

    MergeSort(arr, 0, n - 1);

    printf("After Sort:\n");
    printArray(arr, n);

    return 0;
}
```

Output

```
Before Sort:
 5 3 76 1 53
After Sort:
 1 3 5 53 76
```

2. Quick Sort

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

int partition(int arr[], int start, int end){
    int l = start, r = end;
    int p = arr[start];

    while (l < r){
        while(arr[l] <= p) l++;
        while(arr[r] >= p) r--;
        if(l < r){
            int temp = arr[l];
            arr[l] = arr[r];
            arr[r] = temp;
        }
    }
    arr[start] = arr[r];
    arr[r] = p;
    return r;
}

void QuickSort(int arr[], int start, int end) {
    if (start < end) {
        int pi = partition(arr, start, end);
        QuickSort(arr, start, pi - 1);
        QuickSort(arr, pi + 1, end);
    }
}

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

int main() {
    int arr[] = {4, 1, 5, 7, 2};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Before Sort:\n");
    printArray(arr, n);

    QuickSort(arr, 0, n - 1);

    printf("After Sort:\n");
    printArray(arr, n);

    return 0;
}
```

Output:

```
Before Sort:
 4 1 5 7 2
After Sort:
 1 2 4 5 7
```

CONCLUSION:

Recursive sorting algorithms such as Merge Sort and Quick Sort are efficient and widely used. Merge Sort is stable and guarantees O(n log n) time, while Quick Sort is faster in practice but can degrade to $O(n^2)$ in the worst case. This lab enhanced understanding of recursion and divide-and-conquer strategies.

Experiment 3

OBJECTIVE: To perform binary search on a sorted array using C programming.

THEORY:

Binary Search is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. It works on the principle of divide and conquer.

Features of Binary Search:

1. Fast Search Time: O(log n) time complexity.
2. Works Only on Sorted Arrays.
3. Efficient than Linear Search for large datasets.

ALGORITHM:

1. Initialize low = 0 and high = size - 1.
2. Repeat while low <= high:
   - Find mid = (low + high) / 2
   - If arr[mid] == target, return mid
   - Else if arr[mid] > target, set high = mid - 1
   - Else set low = mid + 1
3. If not found, return -1

PROGRAMS

```c
#include <stdio.h>
int binarySearch(int arr[], int size, int key) {
	int low = 0, high = size - 1, mid;
	while (low <= high) {
		mid = (low + high) / 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[100], n, key, result;
	printf("Enter number of elements: ");
	scanf("%d", &n);
	printf("Enter %d sorted elements: ", n);
	for (int i = 0; i < n; i++)
		scanf("%d", &arr[i]);
	printf("Enter element to search: ");
	scanf("%d", &key);
	result = binarySearch(arr, n, key);
	if (result != -1)
		printf("Element found at index %d\n",
result);
	else
		printf("Element not found\n");

	return 0;
}
```

Output:

```
Enter number of elements: 5
Enter 5 sorted elements: 1 3 5 7 9
Enter element to search: 7
Element found at index 3
```

CONCLUSION:

In this lab, we implemented binary search in C and verified its efficiency in searching sorted arrays. The time complexity was observed to be logarithmic, making it ideal for large datasets.