Thadomal Shahani Engineering College

Bandra (W.), Mumbai - 400050

Division : C2

Batch : 3

Roll Number : 1902112

Certify that Mr. / ~~Miss~~ TUSHAR NANKANI

of COMPUTER Department, Semester 3 with

Roll No. 1902112 has completed a course of the necessary experiments in the subject **Data Structures Lab (CSL301)** under my supervision in the Thadomal Shahani Engineering College Laboratory in the **year 2020 - 2021.**

Prof. Shilpa Ingoley
Teacher In-Charge

Prof. Tanuja Sarode
Head of the Department

Dr. G.T. Thampi
Principal

Date 30/11/2020

List of Experiments
S.E. (Comp.) Sem III
Subject: **Data Structures Lab (CSL301)**

Title of Experiments

STACK

Linear Data Structure

THEORY: Stack is a linear data structure in which addition of new element or deletion of an existing element always take place at the same end which is known as top of the stack.

For eg: stack of books, coins, etc.

As the elements in the stack can be added or removed from the top, it means the last element to be added to stack is the first one to be removed.

Therefore stack are also called as Last In First Out (LIFO) or FILO.

Stacks are useful, when data needs to be stored and then retrieved in reverse order.

APPLICATION OF STACK:

- Page-visited history in a Web browser
- Undo sequence in a text editor
- Saving local variables when one function calls another, and this one calls another, and so on.
- Checking balanced expressions
- Evaluating algebraic expressions
- Converting a number to any base system

OPERATIONS:

create stack,
push stack,
Pop stack,
peek stack ,
empty stack,
full stack,
stack count,
destroy stack

Implementation of stack using Array:

Stack contains an ordered collection of elements and array is used to store ordered list of elements.
Hence it would be very easy to manage stack using an array.
In array we can push elements one by one from 0th position till (n-1)th position.

A stack of elements of any particular type is finite sequence of elements of that type together with following operations:

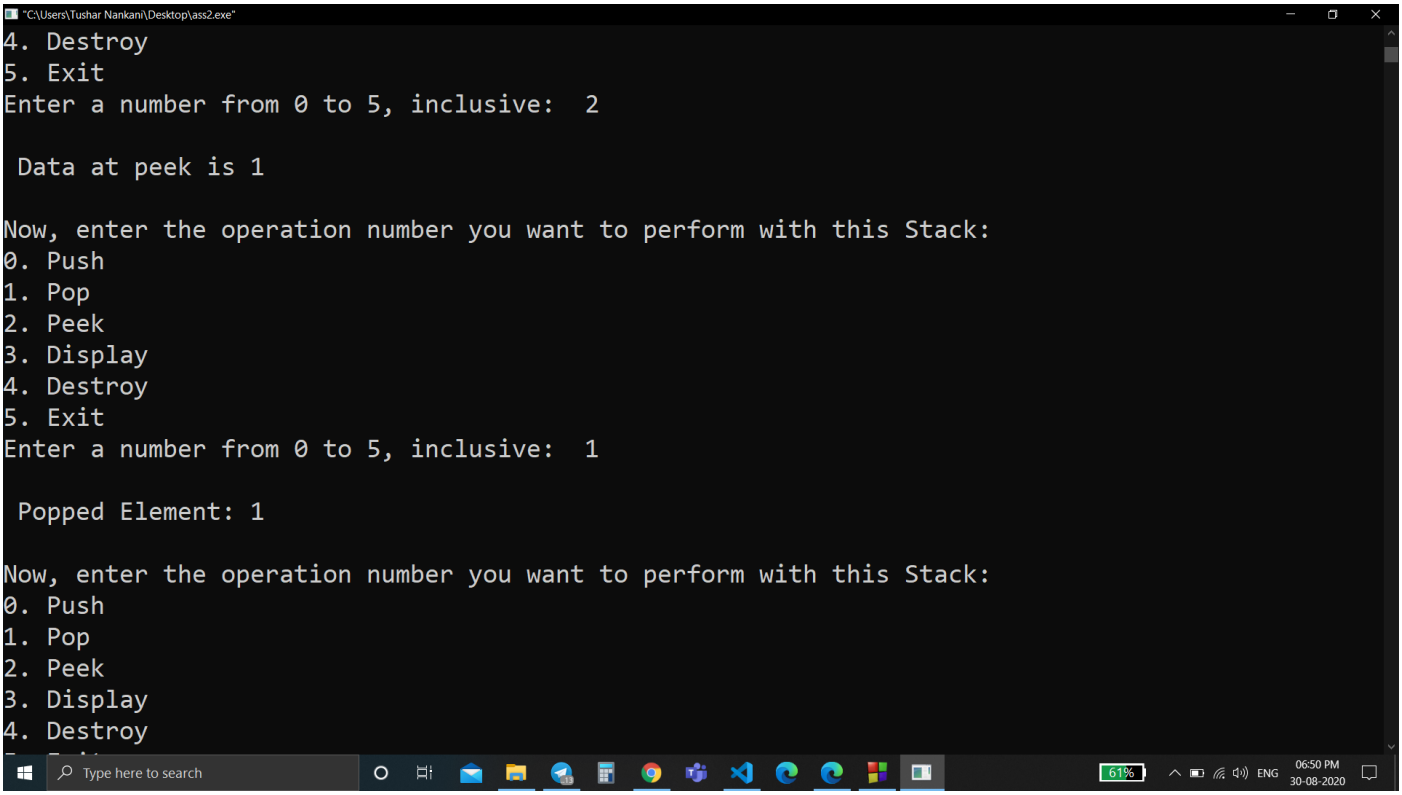
- Initialize stack to be empty
- Determine whether stack is empty or not
- Determine if stack is full or not.
- If stack is not full, then insert new element at one end of stack called top. This operation is called as push.
- If stack is not empty then retrieve the elements from its top.
- If stack is not empty then delete the elements from its top. This operation is called as pop.

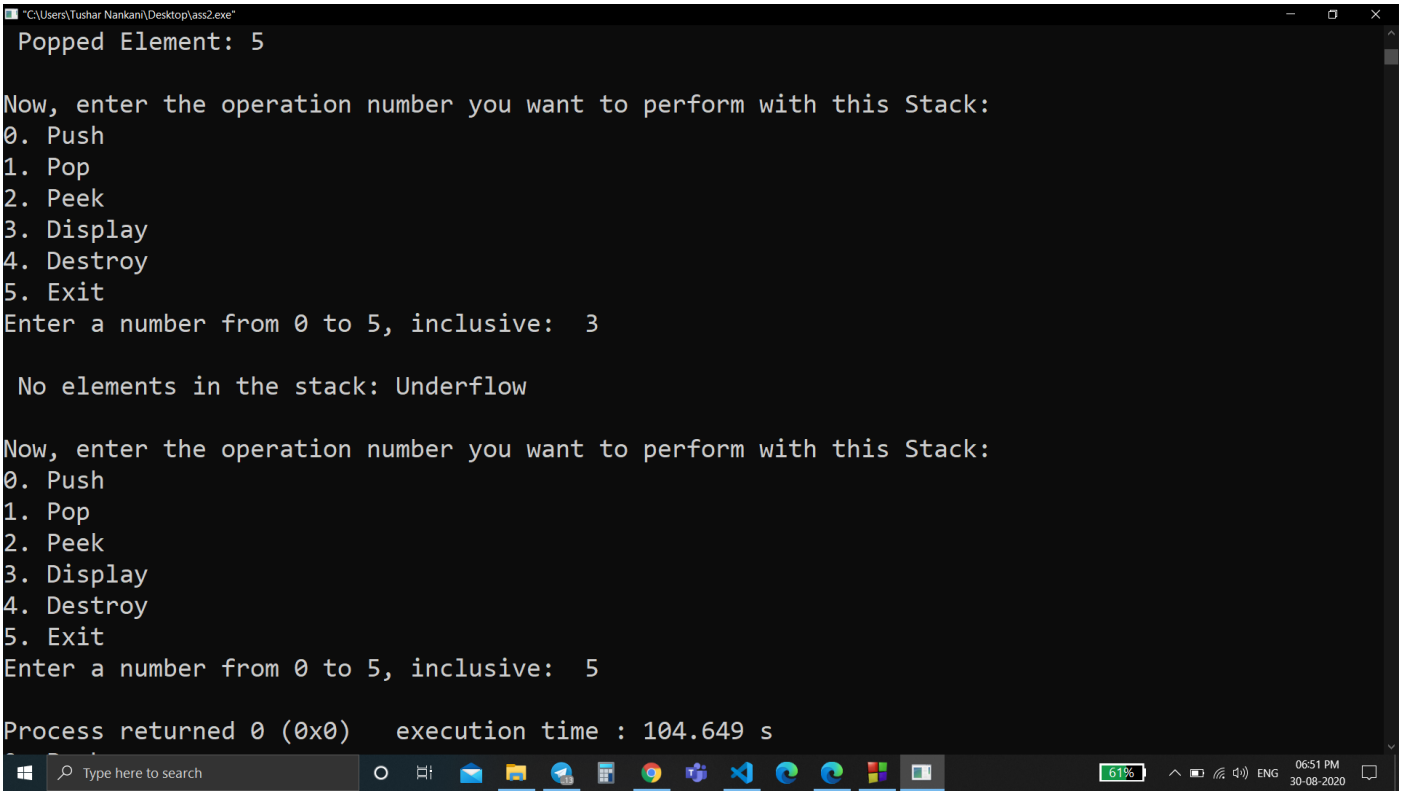
```c
#include<stdio.h>
#include<unistd.h>

#define MAX 50
int choice, element;

struct stack
{
    int data[MAX];
    int top;
};

void welcome()
{
    // welcome message;
    printf("\nWELCOME TO THE STACK MANIPULATION CENTER!\n");
}

void options()
{
    // Operations;
    printf("\n\nNow, enter the operation number you want to perform
with this Stack: \n");
    printf("0. Push\n");
    printf("1. Pop\n");
    printf("2. Peek\n");
    printf("3. Display\n");
    printf("4. Destroy\n");
    printf("5. Exit\n");
    printf("Enter a number from 0 to 5, inclusive: \t");
    scanf("%d", &choice);
}

void initialize(struct stack *s) {
    s->top = -1;
}

int isEmpty(struct stack *s) {
    return (s->top == -1);
}

int isFull(struct stack *s) {
```

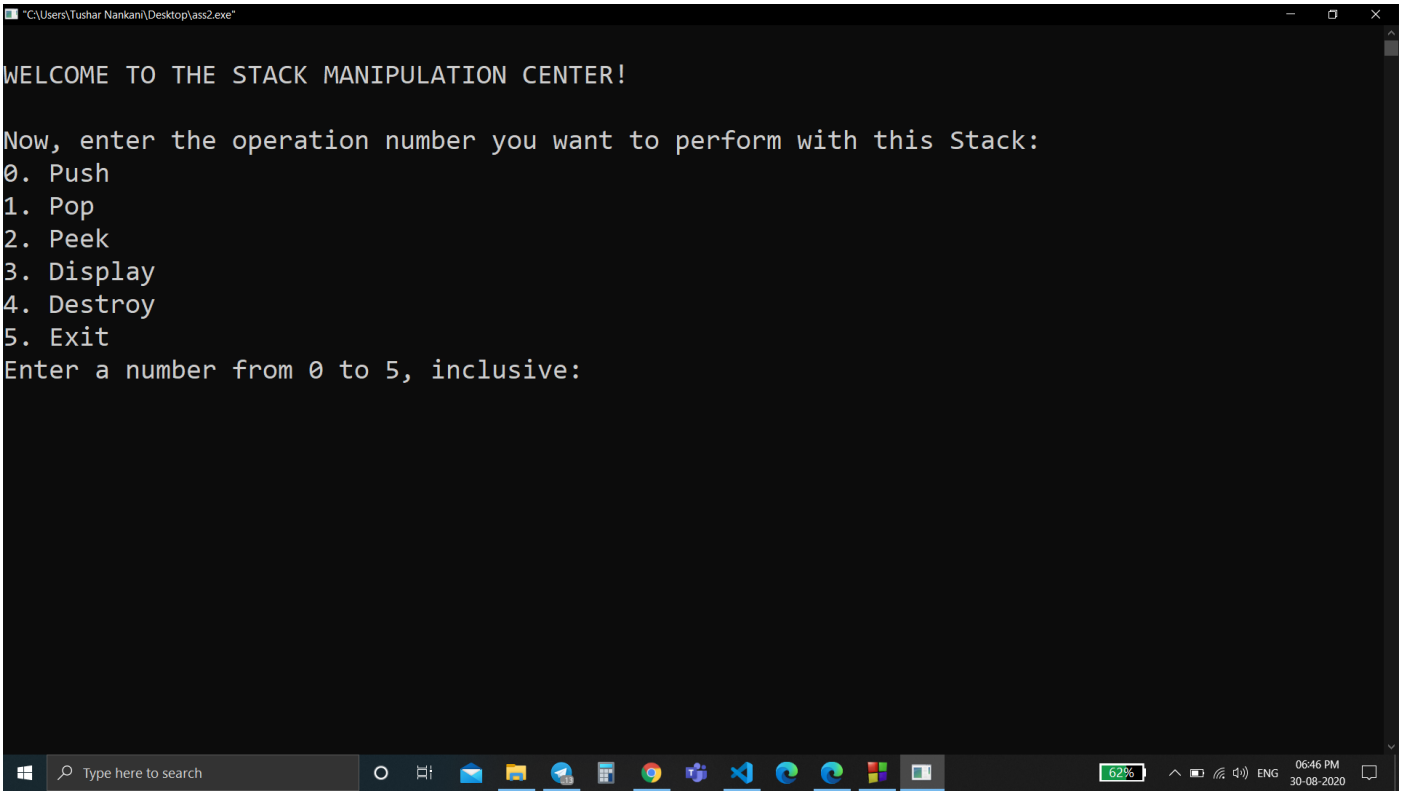
```
    return (s->top == MAX - 1);
}

void push(struct stack *s, int e)
{
    if(isFull(s))
        printf("\n The stack is full-Overflow Condition");
    else
    {
        s->data[++s->top] = e;
        printf("\n Pushed Successfully.");
    }
}

void pop(struct stack *s)
{
    int popped;
    if(isEmpty(s))
    printf("\n The stack in empty-Underflow Condition");
    else
    {
        popped = s->data[s->top--];
        printf("\n Popped Element: %d", popped);
    }
}

void peek(struct stack *s)
{
    if (isEmpty(s))
        printf("\n No elements in the stack");
    else
        printf("\n Data at peek is %d", s->data[s->top]);
}

void display(struct stack *s)
{
    int i;
    if (isEmpty(s))
        printf("\n No elements in the stack: Underflow");
    else
    {
        printf("\n The elements in the stack are: ");
```

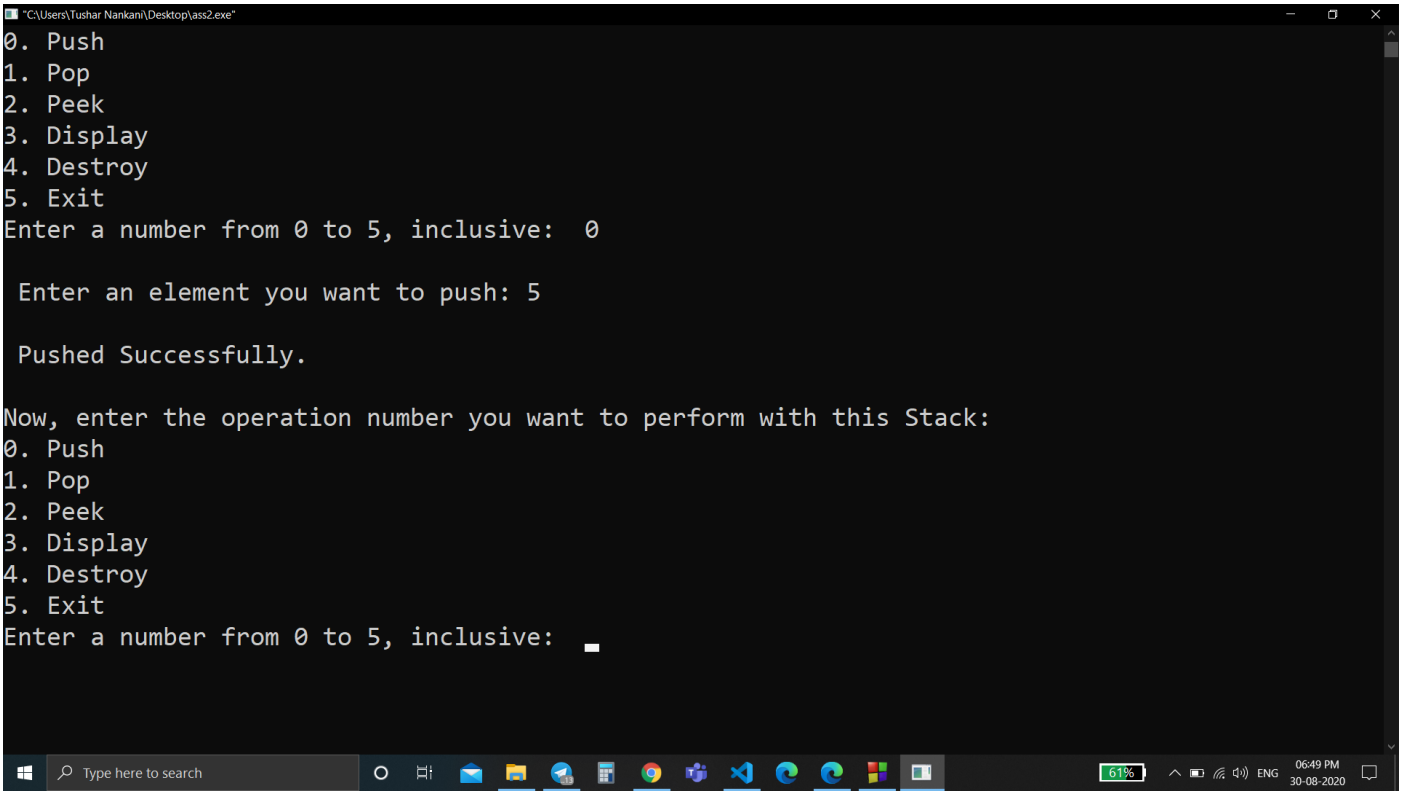
```
        for(i = s->top; i >= 0; i--)
            printf("\n %d", s->data[i]);
    }
}

void destroy(struct stack *s)
{
    // s->top = -1;

    while(s->top != -1)
        pop(s);
}

void end()
{
    printf("\nThank you for using our services!\nSee you again!\n");
}

void main()
{
    struct stack st;
    initialize(&st);

    welcome();
    options();

    while(1)
    {
        options();
        switch(choice)
        {
            case 0:
                {
                    printf("\n Enter an element you want to push: ")
;
                    scanf("%d", &element);
                    push(&st, element);
                    break;
                }
            case 1:
                pop(&st);
                break;
```

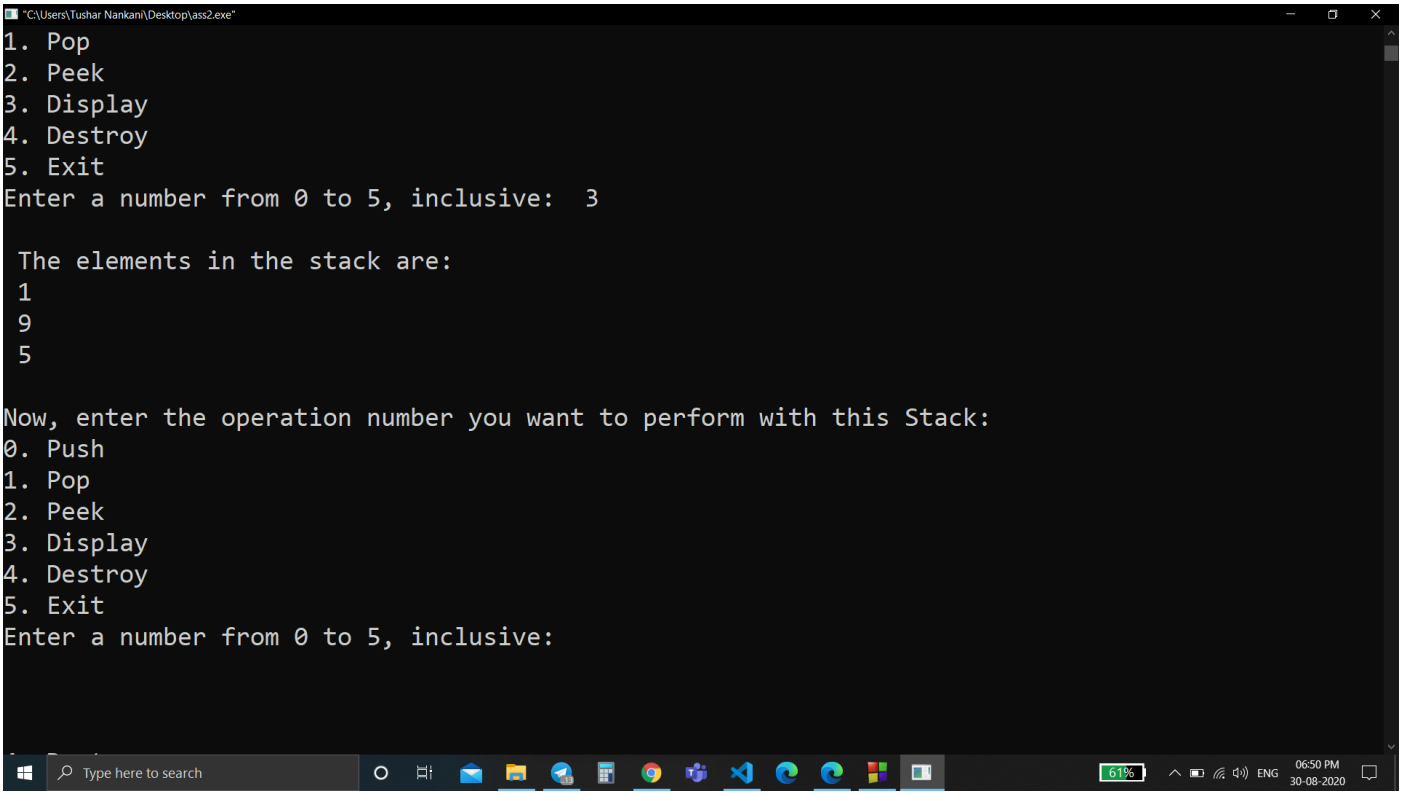
```
            case 2:
                peek(&st);
                break;
            case 3:
                display(&st);
                break;
            case 4:
                destroy(&st);
                break;
            case 5:
                {
                    end();
                    exit(0);
                }
            default:
                printf("\n* Option not valid! *");
        }
    }
}
```


"C:\Users\Tushar Nankani\Desktop\ass2.exe"
4. Destroy
5. Exit
Enter a number from 0 to 5, inclusive: 2
Data at peek is 1
Now, enter the operation number you want to perform with this Stack:
0. Push
1. Pop
2. Peek
3. Display
4. Destroy
5. Exit
Enter a number from 0 to 5, inclusive: 1
Popped Element: 1
Now, enter the operation number you want to perform with this Stack:
0. Push
1. Pop
2. Peek
3. Display
4. Destroy
Type here to search
61%
ENG
06:50 PM
30-08-2020


"C:\Users\Tushar Nankani\Desktop\ass2.exe"
Popped Element: 5
Now, enter the operation number you want to perform with this Stack:
0. Push
1. Pop
2. Peek
3. Display
4. Destroy
5. Exit
Enter a number from 0 to 5, inclusive: 3
No elements in the stack: Underflow
Now, enter the operation number you want to perform with this Stack:
0. Push
1. Pop
2. Peek
3. Display
4. Destroy
5. Exit
Enter a number from 0 to 5, inclusive: 5
Process returned 0 (0x0) execution time : 104.649 s
Type here to search
61%
ENG
06:51 PM
30-08-2020


"C:\Users\Tushar Nankani\Desktop\ass2.exe"
WELCOME TO THE STACK MANIPULATION CENTER!
Now, enter the operation number you want to perform with this Stack:
0. Push
1. Pop
2. Peek
3. Display
4. Destroy
5. Exit
Enter a number from 0 to 5, inclusive:
Type here to search
62%
ENG
06:46 PM
30-08-2020


"C:\Users\Tushar Nankani\Desktop\ass2.exe"
0. Push
1. Pop
2. Peek
3. Display
4. Destroy
5. Exit
Enter a number from 0 to 5, inclusive: 0
Enter an element you want to push: 5
Pushed Successfully.
Now, enter the operation number you want to perform with this Stack:
0. Push
1. Pop
2. Peek
3. Display
4. Destroy
5. Exit
Enter a number from 0 to 5, inclusive:
Type here to search
61%
ENG
06:49 PM
30-08-2020


"C:\Users\Tushar Nankani\Desktop\ass2.exe"
1. Pop
2. Peek
3. Display
4. Destroy
5. Exit
Enter a number from 0 to 5, inclusive: 3
The elements in the stack are:
1
9
5
Now, enter the operation number you want to perform with this Stack:
0. Push
1. Pop
2. Peek
3. Display
4. Destroy
5. Exit
Enter a number from 0 to 5, inclusive:
Type here to search
61%
ENG
06:50 PM
30-08-2020

INFIX TO POSTFIX

- Basically, there are 3 types of notations for expressions:

1. Infix: operator is between operands

A + B

2. Postfix: operator follows operands

A B +

3. Prefix: operator comes before operands

+ A B

A, B --> OPERANDS

+ --> OPERATOR

POSTFIX NOTATION

- Reverse Polish notation (RPN), also known as Polish postfix notation or simply postfix or Suffix notation, is a mathematical notation in which operators follow their operands
- The notation is used because the format that the equation is in is easier for machines to interpret rather than the notation we are used to, infix notation(A + B)
- The syntax does not require brackets or parenthesis

- To convert infix expression to postfix expression, we will use the stack data structure.

- By scanning the infix expression from left to right, when we will get any operand simply add them to the postfix-form(string), and for the operator and parenthesis add them in the stack maintaining the precedence of them

OPERATOR PRECEDENCE:

OPERATOR	SYMBOL	PRECEDENCE
1. Exponential	$ or ^	Highest
2. Multiplication /Division	*, /, %	Next Highest
3. Addition /Subtraction	+, -	Lowest

Evaluation Order in infix

- Brackets or parenthesis
- Exponentiation
- Multiplication or Division
- Addition or Subtraction

Note: Operators with the same priority are evaluated from left to right

Using this rule with the above expression, we would take the following actions:

1. Copy operand A to output expression.
2. Push operator + into the stack.
3. Copy operand B to output expression.
4. Push operator * into the stack. (Priority of * is higher than +.)
5. Copy operand C to output expression.
6. Pop operator * and copy to output expression.
7. Pop operator + and copy to output expression.

When a current operator with a lower or equal priority forces the top operator to be popped from the stack, we must check the new top operator. If it is also greater than the current operator, it is popped to the output expression. Consequently, we may pop several operators to the output expression before pushing the new operator into the stack.

X + Y

READ CHAR	STACK	POSTFIX
X		X
+	+	X
Y	+	XY
		XY+

Name: **Tushar Nankani** Roll No: **1902112** Batch: **C23**

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

#define MAX 50

char ch, infix[MAX], postfix[MAX], stack[MAX];

int priorityCheck(char c)
{
    if(c == '$' || c == '^')
        return 3;

    if(c == '*' || c == '/' || c == '%')
        return 2;

    if(c == '+' || c == '-')
        return 1;

    return 0;
}

// debugging stack;
void display(int top)
{
    int i;

    printf("\n %c\t\t\t", ch);

    if (top == -1)
        printf("Empty");

    else
    {
        for(i = top; i >= 0; i--)
            printf("%c", stack[i]);
    }
    printf("\t\t\t%s", postfix);
}
```

```
int main()
{
    int top = -1, i = -1, j = 0;

    printf("Enter the Infix Expression:\t");
    scanf("%s", infix);

    printf("\n\nSYMBOL\t\t\tSTACK\t\t\tPOSTFIX\n");

    while(infix[++i] != '\0')
    {
        ch = infix[i];

        // if `ch` is any of the following: + - * / % ^ $;
        if(priorityCheck(ch))
        {
            if(priorityCheck(ch) <= priorityCheck(stack[top]))
            {
                while(priorityCheck(ch) <= priorityCheck(stack[top])
)
                    postfix[j++] = stack[top--];
            }

            stack[++top] = ch;
        }

        else if(ch == '(')
        {
            stack[++top] = ch;
        }

        else if(ch == ')')
        {
            while(stack[top] != '(')
            {
                postfix[j++] = stack[top--];
            }
            --top;
        }

        // when it is not an operator, it is a number;
        else
```

```
        {
            postfix[j++] = ch;
        }

        // debugging stack;
        display(top);


    }

    while(top != -1)
    {
        postfix[j++] = stack[top--];
    }


    // very important step, tends to be missed;
    postfix[j] = '\0';

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

    return 0;
}
```

```
"C:\Users\Tushar Nankani\Desktop\Git\sem3-assignments\assignment3.exe"
Enter the Infix Expression:     (a+b*c-d)/(e*f)


SYMBOL                  STACK                   POSTFIX

 (                      (
 a                      (                       a
 +                      +(                      a
 b                      +(                      ab
 *                      *+(                     ab
 c                      *+(                     abc
 -                      -(                      abc*+
 d                      -(                      abc*+d
 )                      Empty                   abc*+d-
 /                      /                       abc*+d-
 (                      (/                      abc*+d-
 e                      (/                      abc*+d-e
 *                      *(/                     abc*+d-e
 f                      *(/                     abc*+d-ef
 )                      /                       abc*+d-ef*


The  Postfix expression is:      abc*+d-ef*/

Process returned 0 (0x0)   execution time : 31.974 s
Press any key to continue.
```

```
"C:\Users\Tushar Nankani\Desktop\Git\sem3-assignments\assignment3.exe"
Enter the Infix Expression:     a+b*(c^d-e)^(f+g*h)-i


SYMBOL                  STACK                   POSTFIX

 a                      Empty                   a
 +                      +                       a
 b                      +                       ab
 *                      *+                      ab
 (                      (*+                     ab
 c                      (*+                     abc
 ^                      ^(*+                    abc
 d                      ^(*+                    abcd
 -                      -(*+                    abcd^
 e                      -(*+                    abcd^e
 )                      *+                      abcd^e-
 ^                      ^*+                     abcd^e-
 (                      (^*+                    abcd^e-
 f                      (^*+                    abcd^e-f
 +                      +(^*+                   abcd^e-f
 g                      +(^*+                   abcd^e-fg
 *                      *+(^*+                  abcd^e-fg
 h                      *+(^*+                  abcd^e-fgh
 )                      ^*+                     abcd^e-fgh*+
 -                      -                       abcd^e-fgh*+^*+
 i                      -                       abcd^e-fgh*+^*+i


The  Postfix expression is:      abcd^e-fgh*+^*+i-

Process returned 0 (0x0)   execution time : 30.681 s
Press any key to continue.
```

POSTFIX EVALUATION

THEORY:

To evaluate a postfix expression, execute the following steps until the end of the expression.

1. If recognize an operand, push it on the stack.
2. If recognizing an operator, pop its operands, apply the operator and push the value on the stack.

Upon conclusion, the value of the postfix expression is on the top of the stack.

At each step in the postfix evaluation algorithm, the state of the stack allows us to identify whether an error occurs and the cause of the error.

The concept of associativity refers to the order of execution for operators at the same precedence level. If more than one operator has the same precedence, the leftmost operator executes first in the case of left associativity (+, -, *, /, %) and the rightmost operator executes first in the case of right associativity (^).

The concept of associativity refers to the order of execution for operators at the same precedence level. If more than one operator has the same precedence, the leftmost operator executes first in the case of left associativity (+, -, *, /, %) and the rightmost operator executes first in the case of right associativity (^).

There are various advantages to this as well, since, it easy for calculators to evaluate such answers, it becomes easy to evaluate, as we do not have to take care of the precedence of the operators, hence it is widely used.

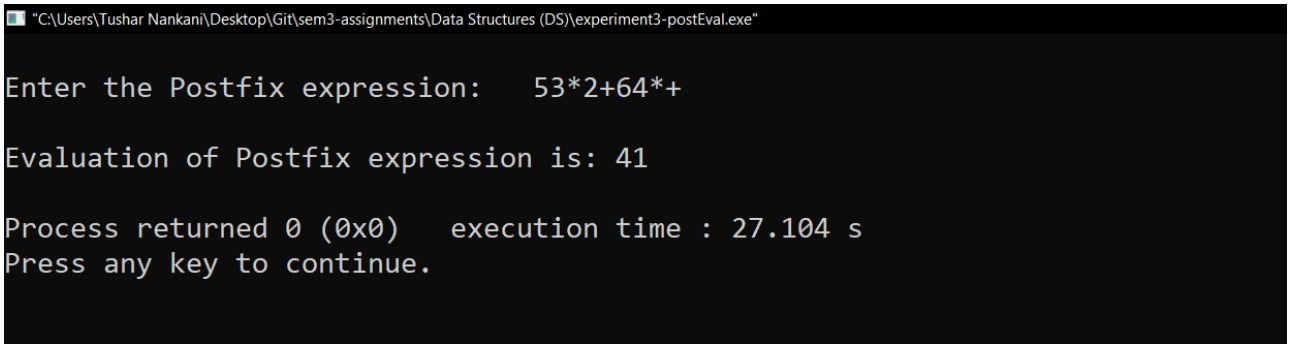
Name: **Tushar Nankani** Roll No: **1902112** Batch: **C23**

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

#define MAX 50

char ch, postfix[MAX];
int stack[MAX], top, ans;

int priorityCheck(char c)
{
    return (c == '$' || c == '^' || c == '*' || c == '/' || c == '%'
 || c == '+' || c == '-');
}

void push(int temp)
{

    if(top == MAX-1)
        printf("\nStack full! Increase Stack Size.");
    else
        stack[++top] = temp;
}

void eval(char op,int num1, int num2)
{
    int result;

    switch(op)
    {
        case '+' :
            result = num1 + num2;
            break;
        case '-' :
            result=num1 - num2;
            break;
        case '*' :
            result=num1 * num2;
            break;
        case '/' :
```

```
            result=num1 / num2;
            break;
        case '%' :
            result=num1 % num2;
            break;
        case '^':
        case '$':
            result=pow(num1,num2);
    }
    push(result);
}

int pop()
{
    int x;

    if(top == -1)
        printf("Stack-underflow condition!");
    else
        x = stack[top--];

    return x;
}

int main()
{
    int n1, n2, i = -1;
    top = -1, ans = 0;

    printf("\nEnter the Postfix expression: \t");
    scanf("%s", postfix);

    while(postfix[++i] != '\0')
    {
        if(priorityCheck(postfix[i]))
        {
            n2=pop();
            n1=pop();
            eval(postfix[i],n1,n2);
        }
        else
        {
```

```
            ch = postfix[i] - '0';
            push(ch);
        }
    }

    printf("\nEvaluation of Postfix expression is: %d\n", pop());
    return 0;
}
```

OUTPUT:



QUEUE - DATA STRUCTURE

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO).
Eg- queue is any queue of consumers for a resource where the consumer that came first is served first.

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Operations on Queue:
Mainly the following four basic operations are performed on queue:

- Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
- Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflo condition.
- Front: the front item from queue.
- Rear: the last item from queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are -

peek() - Gets the element at the front of the queue without removing it.

isfull() - Checks if the queue is full.

isempty() - Checks if the queue is empty.

APPLICATION OF QUEUE:

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

- Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
- In real life scenario, Call Center phone systems uses -
- Queues to hold people calling them in an order, until a service representative is free.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, i.e First come first served.

Name: **Tushar Nankani** Roll No: **1902112** Batch: **C23**

IMPLEMENTATION OF **QUEUE**: BRUTE FORCE

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

#define MAX 5
int queue[MAX];
int front, rear;

int options()
{
    int choice = -1;
    printf("\n\nEnter the operation number you want to perform with
this Queue: \n");
    printf("0. Enqueue (Insert)\n");
    printf("1. Dequeue (Delete)\n");
    printf("2. Traverse (Display)\n");
    printf("3. Peek\n");
    printf("4. Destroy (Reinitialize)\n");
    printf("5. Exit\n");
    printf("Enter a number from [0] to [5], inclusive: \t");
    scanf("%d", &choice);
    return choice;
}

void initialize()
{
    front = rear = 0;
}

int isEmpty()
{
    if(front == rear) {
        printf("\n** The Queue is empty - Underflow Condition. **");
        return 1;
    }
    return 0;
}
```

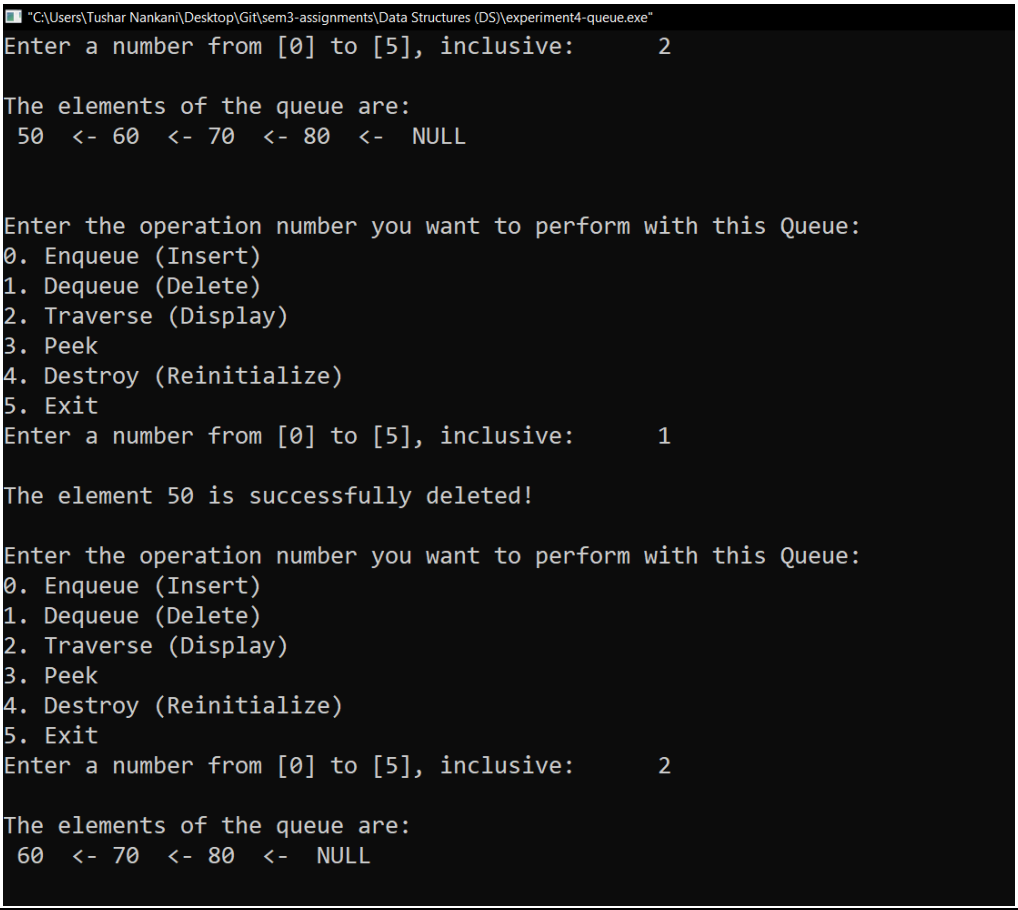
```
int isFull()
{
    if(rear == MAX){
        printf("\n** The Queue is full - Overflow Condition. **");
        return 1;
    }
    return 0;
}

void enqueue()
{
    if(!isFull())
    {
        int n;
        printf("\nEnter a element to enqueue:   ");
        scanf("%d", &n);
        queue[rear++] = n;
        printf("\nThe element %d is successfully inserted!", n);
    }
}

void dequeue()
{
    if(!isEmpty())
    {
        // BRUTE FORCE
        printf("\nThe element %d is successfully deleted!", queue[fr
ont]);
        for(int i = 1; i < rear; i++)
            queue[i - 1] = queue[i];
    }
    rear -= 1;
}

void traverse()
{
    if(!isEmpty())
    {
        printf("\nThe elements of the queue are: \n");
        for(int i = front ; i < rear; i++)
            printf(" %d  <-", queue[i]);
        printf("  NULL\n");
```

```
    }
}

void peek()
{
    if(!isEmpty())
        printf("\nQueue's front element is: %d", queue[front]);
}

void destroy()
{
    initialize();
    printf("\nThe queue is successfully reinitialized!");
}

int main()
{
    // welcome message;
    printf("\nWELCOME TO THE QUEUE MANIPULATION CENTER!\n");
    initialize();
    while(1)
    {
        switch(options())
        {
        case 0:
            enqueue();
            break;
        case 1:
            dequeue();
            break;
        case 2:
            traverse();
            break;
        case 3:
            peek();
            break;
        case 4:
            destroy();
            break;
        case 5:
            exit(0);
        default:
```

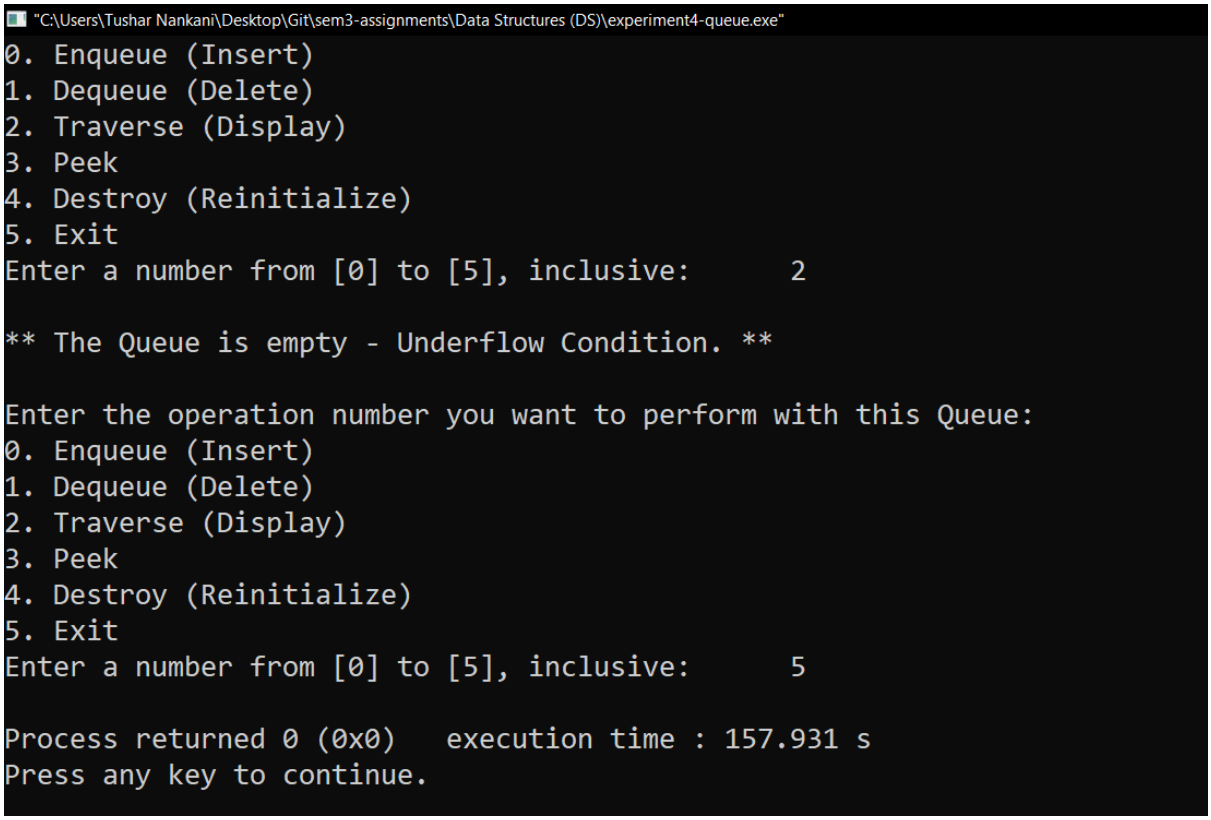
```
            printf("\nINVALID OPTION.");
        }
    }

    return 0;
}
```

```
WELCOME TO THE QUEUE MANIPULATION CENTER!


Enter the operation number you want to perform with this Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive:      0

Enter a element to enqueue:   50

The element 50 is successfully inserted!

Enter the operation number you want to perform with this Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive:      0

Enter a element to enqueue:   60

The element 60 is successfully inserted!
```

```
"C:\Users\Tushar Nankani\Desktop\Git\sem3-assignments\Data Structures (DS)\experiment4-queue.exe"
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive:      3
Queue's front element is: 50

Enter the operation number you want to perform with this Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive:      2

The elements of the queue are:
 50  <- 60  <- 70  <-  NULL


Enter the operation number you want to perform with this Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive:      0

Enter a element to enqueue:   80
```




```
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive:      2

The elements of the queue are:
 60  <- 70  <- 80  <-  NULL


Enter the operation number you want to perform with this Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive:      3
Queue's front element is: 60

Enter the operation number you want to perform with this Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive:
```


"C:\Users\Tushar Nankani\Desktop\Git\sem3-assignments\Data Structures (DS)\experiment4-queue.exe"
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive: 2
** The Queue is empty - Underflow Condition. **
Enter the operation number you want to perform with this Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive: 5
Process returned 0 (0x0) execution time : 157.931 s
Press any key to continue.

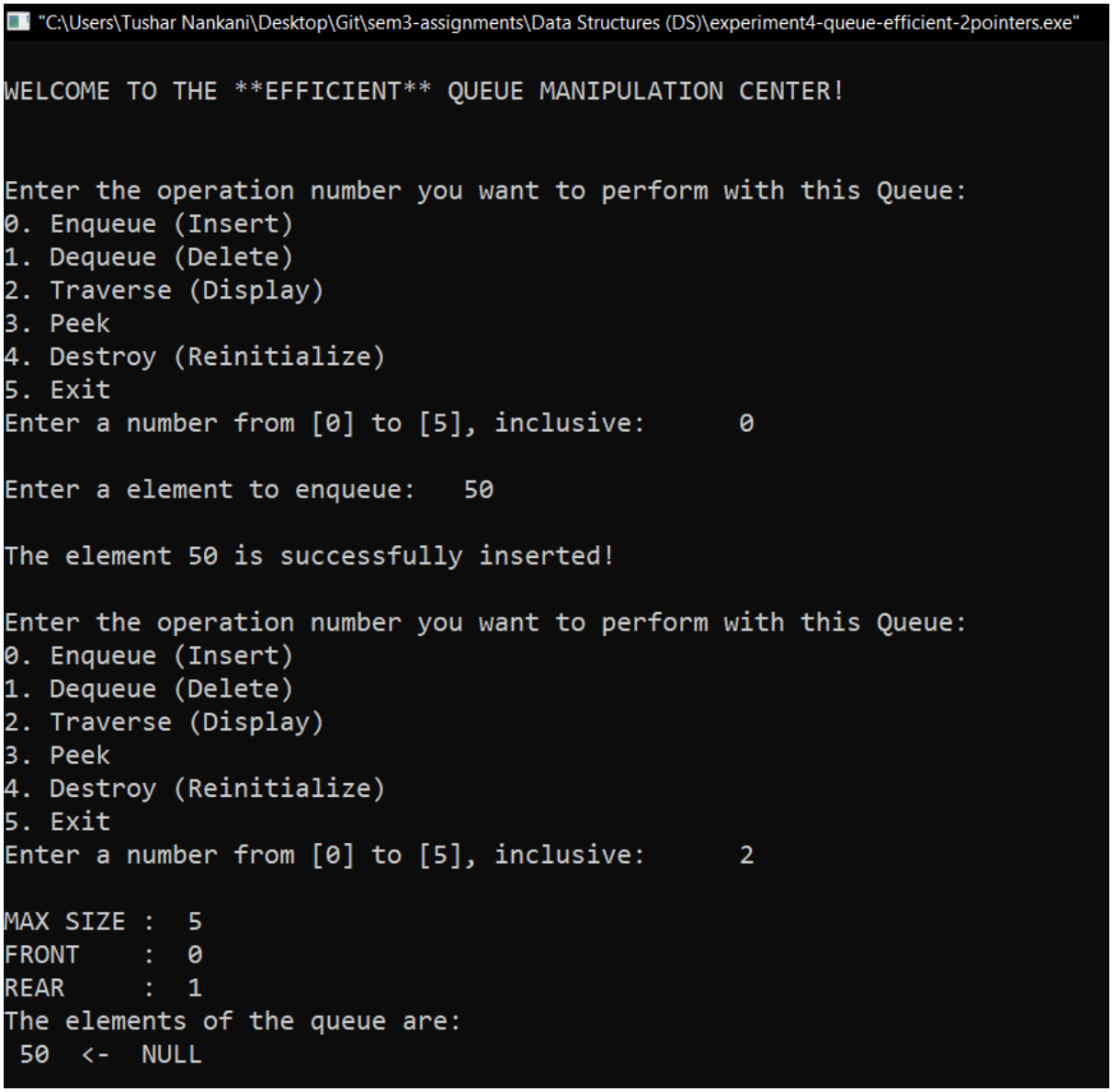
IMPLEMENTATION OF QUEUE: EFFICIENT 2 POINTER METHOD

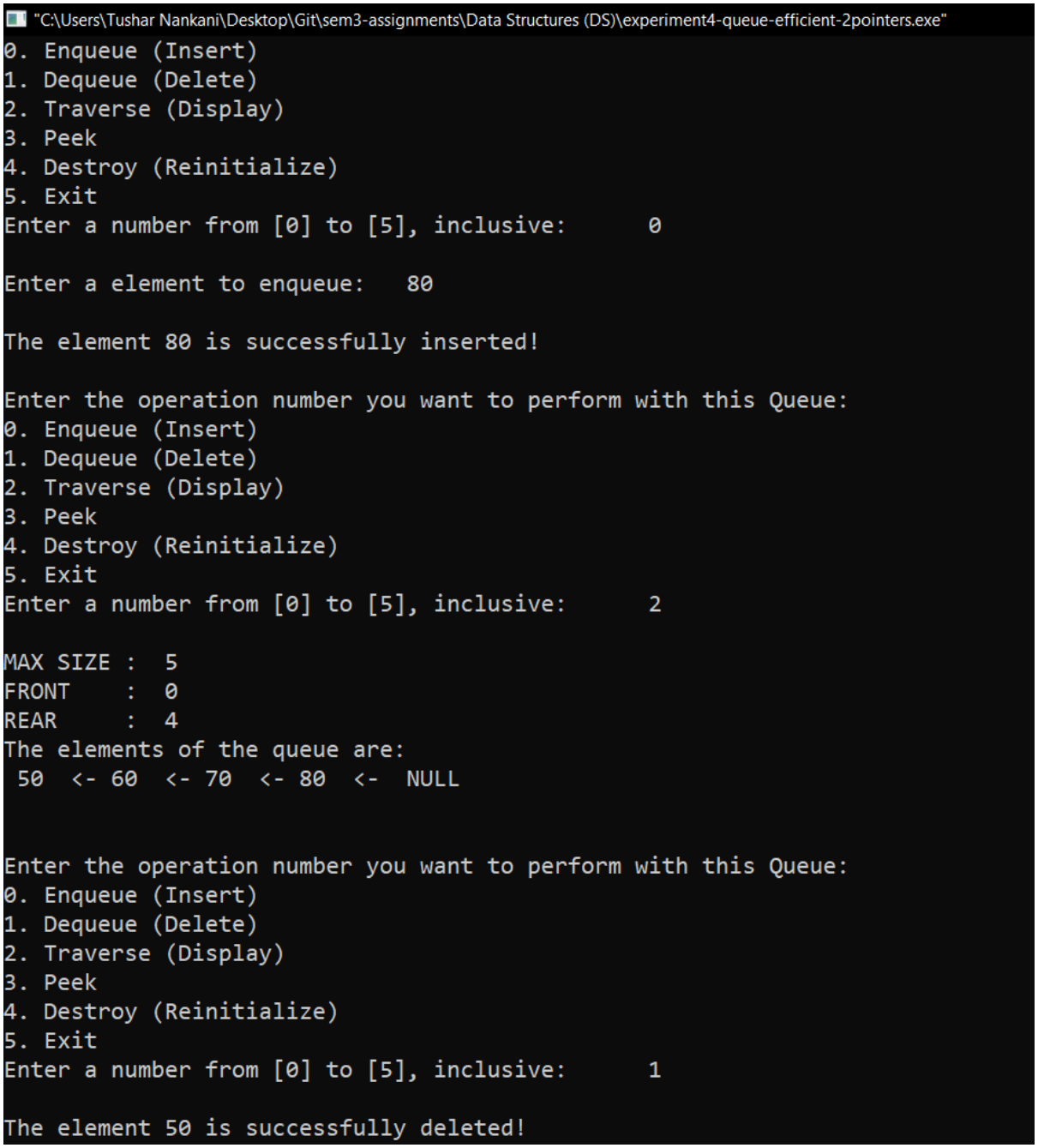
```c
#include<stdio.h>
#include<unistd.h>

#define MAX 5
int queue[MAX];
int front, rear;

int options()
{
    int choice = -1;
    printf("\n\nEnter the operation number you want to perform with
this Queue: \n");
    printf("0. Enqueue (Insert)\n");
    printf("1. Dequeue (Delete)\n");
    printf("2. Traverse (Display)\n");
    printf("3. Peek (Front)\n");
    printf("4. Destroy (Reinitialize)\n");
    printf("5. Exit\n");
    printf("Enter a number from [0] to [5], inclusive: \t");
    scanf("%d", &choice);
    return choice;
}

void initialize()
{
    front = rear = 0;
}

int isEmpty()
{
    if(front == rear) {
        printf("\n** The Queue is empty - Underflow Condition. **");
        return 1;
    }
    return 0;
}

int isFull()
```

```
{
    if(rear == MAX){
        printf("\n** The Queue is full - Overflow Condition. **");
        return 1;
    }
    return 0;
}

void enqueue()
{
    if(!isFull())
    {
        int n;
        printf("\nEnter a element to enqueue:   ");
        scanf("%d", &n);
        queue[rear++] = n;
        printf("\nThe element %d is successfully inserted!", n);
    }
}

void dequeue()
{
    if(!isEmpty())
    {
        // EFFICIENT METHOD;
        printf("\nThe element %d is successfully deleted!", queue[fr
ont]);
        if(front == MAX && rear == MAX)
                initialize();
        else
            front++;
    }
}

void traverse()
{
    if(!isEmpty())
    {
        printf("\nMAX SIZE :  %d", MAX);
        printf("\nFRONT    :  %d", front);
        printf("\nREAR     :  %d", rear);
```

```
        printf("\nThe elements of the queue are: \n");
        for(int i = front ; i < rear; i++)
            printf(" %d  <-", queue[i]);
        printf("  NULL\n");
    }
}

void peek()
{
    if(!isEmpty())
        printf("\nQueue's front element is: %d", queue[front]);
}

void destroy()
{
    initialize();
    printf("\nThe queue is successfully reinitialized!");
}

int main()
{
    // welcome message;
    printf("\nWELCOME TO THE **EFFICIENT** QUEUE MANIPULATION CENTER
!\n");
    initialize();
    while(1)
    {
        switch(options())
        {
        case 0:
            enqueue();
            break;
        case 1:
            dequeue();
            break;
        case 2:
            traverse();
            break;
        case 3:
            peek();
            break;
        case 4:
```

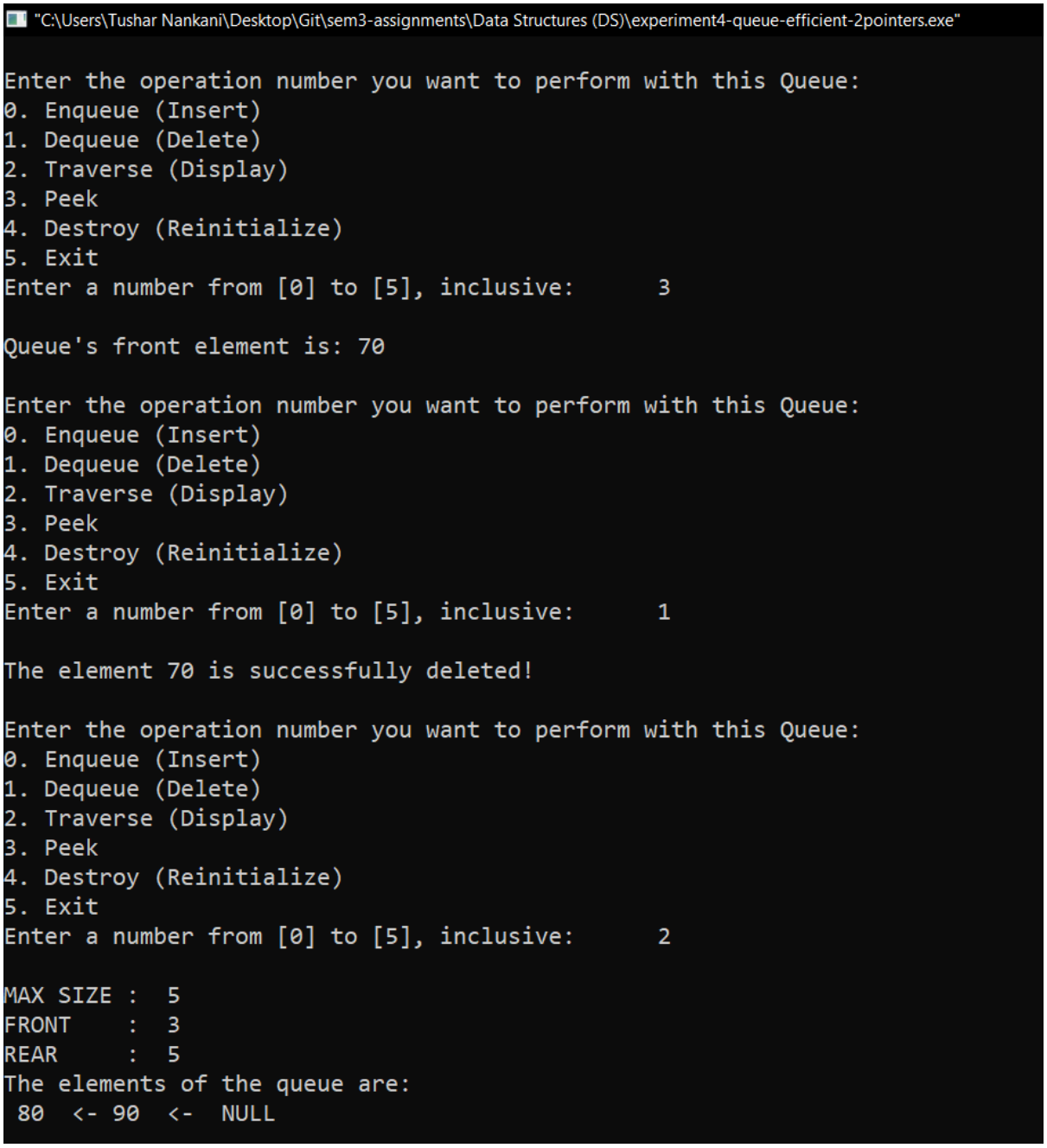
```
            destroy();
            break;
        case 5:
            exit(0);
        default:
            printf("\nINVALID OPTION.");
        }
    }

    return 0;
}
```


"C:\Users\Tushar Nankani\Desktop\Git\sem3-assignments\Data Structures (DS)\experiment4-queue-efficient-2pointers.exe"
WELCOME TO THE **EFFICIENT** QUEUE MANIPULATION CENTER!
Enter the operation number you want to perform with this Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive: 0
Enter a element to enqueue: 50
The element 50 is successfully inserted!
Enter the operation number you want to perform with this Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive: 2
MAX SIZE : 5
FRONT : 0
REAR : 1
The elements of the queue are:
50 <- NULL


"C:\Users\Tushar Nankani\Desktop\Git\sem3-assignments\Data Structures (DS)\experiment4-queue-efficient-2pointers.exe"
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive: 0
Enter a element to enqueue: 80
The element 80 is successfully inserted!
Enter the operation number you want to perform with this Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive: 2
MAX SIZE : 5
FRONT : 0
REAR : 4
The elements of the queue are:
50 <- 60 <- 70 <- 80 <- NULL
Enter the operation number you want to perform with this Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive: 1
The element 50 is successfully deleted!

```
Enter the operation number you want to perform with this Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive:      0

** The Queue is full - Overflow Condition. **

Enter the operation number you want to perform with this Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive:      1

The element 80 is successfully deleted!
```

```
Enter the operation number you want to perform with this Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive:      2

MAX SIZE :  5
FRONT    :  2
REAR     :  4
The elements of the queue are:
 70  <- 80  <-  NULL


Enter the operation number you want to perform with this Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive:      0

Enter a element to enqueue:   90
```


"C:\Users\Tushar Nankani\Desktop\Git\sem3-assignments\Data Structures (DS)\experiment4-queue-efficient-2pointers.exe"
Enter the operation number you want to perform with this Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive: 3
Queue's front element is: 70
Enter the operation number you want to perform with this Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive: 1
The element 70 is successfully deleted!
Enter the operation number you want to perform with this Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive: 2
MAX SIZE : 5
FRONT : 3
REAR : 5
The elements of the queue are:
80 <- 90 <- NULL

Singly LinkedList

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers.

head
A [next]--> B [next]--> C [NULL]

A linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

- Linked List is a very commonly used linear data structure that consists of a group of nodes in a sequence.
- Each node holds its own data and the address of the next node hence forming a chain-like structure.
- Linked Lists are used to create trees and graphs.

Advantages of Linked Lists

- They are dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces access time.

Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in the linked list.

Applications of Linked Lists

- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists, we don't need to know the size in advance.

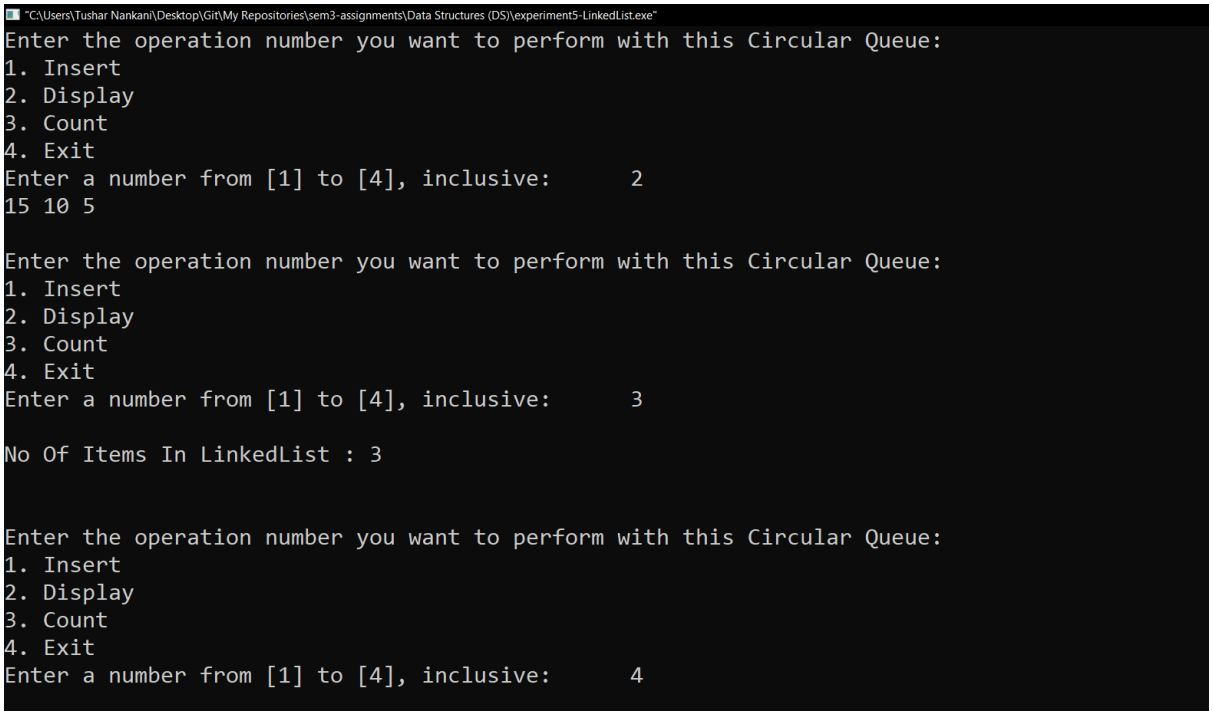
Name: **Tushar Nankani** Roll No: **1902112** Batch: **C23**

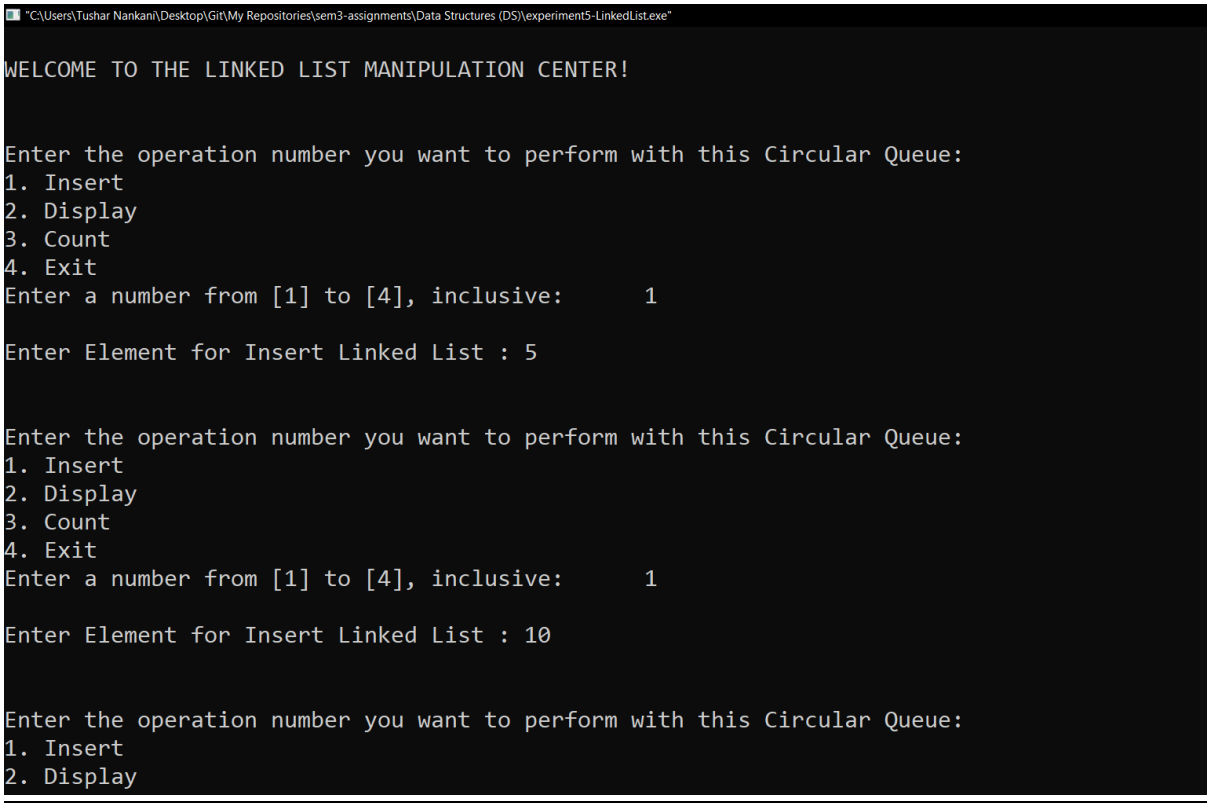
IMPLEMENTATION OF **SINGLY LINKED LIST**

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

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

struct Node* head = NULL, *temp_node = 0;
int data;

int options()
{
    int choice = -1;
    printf("\n\nEnter the operation number you want to perform with
this Circular Queue: \n");
    printf("1. Insert\n");
    // printf("1. Delete\n");
    printf("2. Display\n");
    printf("3. Count\n");
    printf("4. Exit\n");
    printf("Enter a number from [1] to [4], inclusive: \t");
    scanf("%d", &choice);
    return choice;
}

void insert() {
    int new_data;
    printf("\nEnter Element for Insert Linked List : ");
    scanf("%d", &new_data);
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node
));
    new_node->data = new_data;
    new_node->next = head;
    head = new_node;
}
```

```c
void display()
{
    temp_node = head;
    while (temp_node != 0) {
        printf("%d ", temp_node->data);
        temp_node = temp_node->next;
    }
}

void count() {
    int count = 0;
    temp_node = head;
    while (temp_node != 0) {
        count++;
        temp_node = temp_node -> next;
    }
    printf("\nNo Of Items In LinkedList : %d\n", count);
}


int main()
{
    // welcome message;
    printf("\nWELCOME TO THE LINKED LIST MANIPULATION CENTER!\n");
    while(1)
    {
        switch(options())
        {
        case 1:
            insert();
            break;
        case 2:
            display();
            break;
        case 3:
            count();
            break;
        case 4:
            exit(0);
        default:
            printf("\nINVALID OPTION.");
```

```
        }
    }
    return 0;
}
```


"C:\Users\Tushar Nankani\Desktop\Git\My Repositories\sem3-assignments\Data Structures (DS)\experiment5-LinkedList.exe"
Enter the operation number you want to perform with this Circular Queue:
1. Insert
2. Display
3. Count
4. Exit
Enter a number from [1] to [4], inclusive: 2
15 10 5
Enter the operation number you want to perform with this Circular Queue:
1. Insert
2. Display
3. Count
4. Exit
Enter a number from [1] to [4], inclusive: 3
No Of Items In LinkedList : 3
Enter the operation number you want to perform with this Circular Queue:
1. Insert
2. Display
3. Count
4. Exit
Enter a number from [1] to [4], inclusive: 4


"C:\Users\Tushar Nankani\Desktop\Git\My Repositories\sem3-assignments\Data Structures (DS)\experiment5-LinkedList.exe"
WELCOME TO THE LINKED LIST MANIPULATION CENTER!
Enter the operation number you want to perform with this Circular Queue:
1. Insert
2. Display
3. Count
4. Exit
Enter a number from [1] to [4], inclusive: 1
Enter Element for Insert Linked List : 5
Enter the operation number you want to perform with this Circular Queue:
1. Insert
2. Display
3. Count
4. Exit
Enter a number from [1] to [4], inclusive: 1
Enter Element for Insert Linked List : 10
Enter the operation number you want to perform with this Circular Queue:
1. Insert
2. Display

CIRCULAR QUEUE

- In a circular queue, all nodes are treated as circular. The last node is connected back to the first node.

It is an abstract data type.

A circular queue contains a collection of data that allows the insertion of data at the end of the queue and the deletion of data at the beginning of the queue.

This is the main problem with the linear queue, although we have space available in the array, we can not insert any more elements in the queue. This is simply the memory wastage and we need to overcome this problem.

Algorithm to insert an element in the circular queue

```
IF FRONT = -1 and REAR = -1
    SET FRONT = REAR = 0
ELSE IF REAR = MAX - 1 and FRONT ! = 0
    SET REAR = 0
ELSE
    SET REAR = (REAR + 1) % MAX
```

Name: **Tushar Nankani** Roll No: **1902112** Batch: **C23**

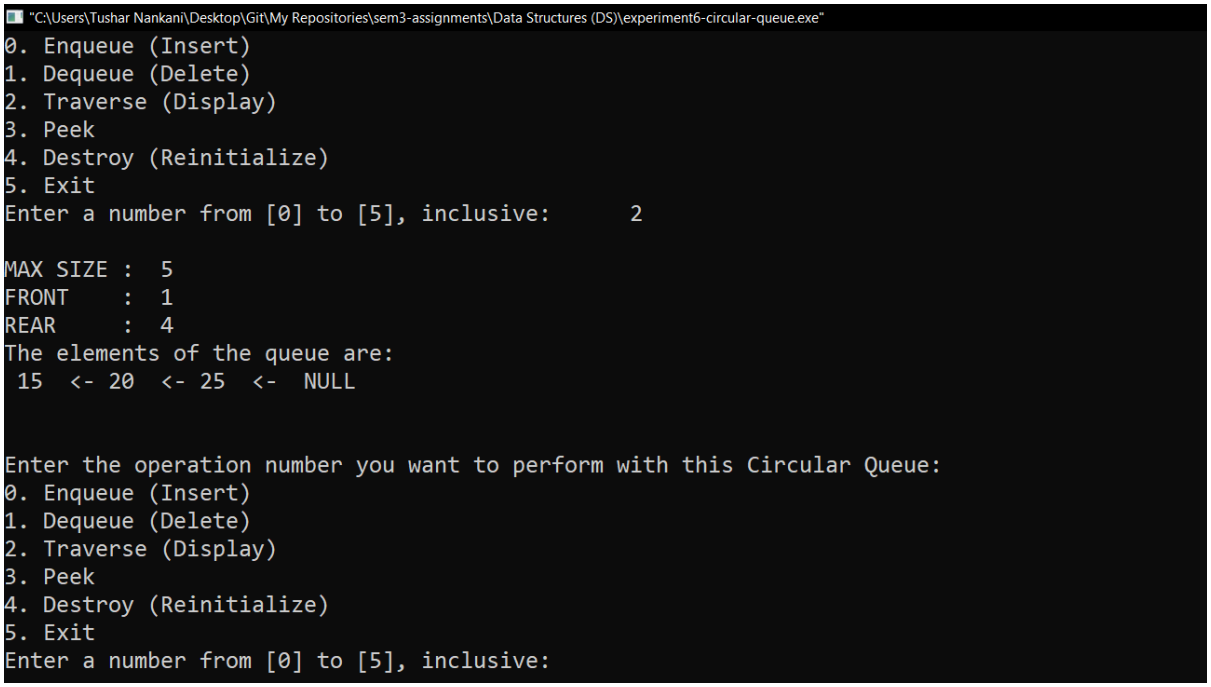
IMPLEMENTATION OF **CIRCULAR QUEUE**

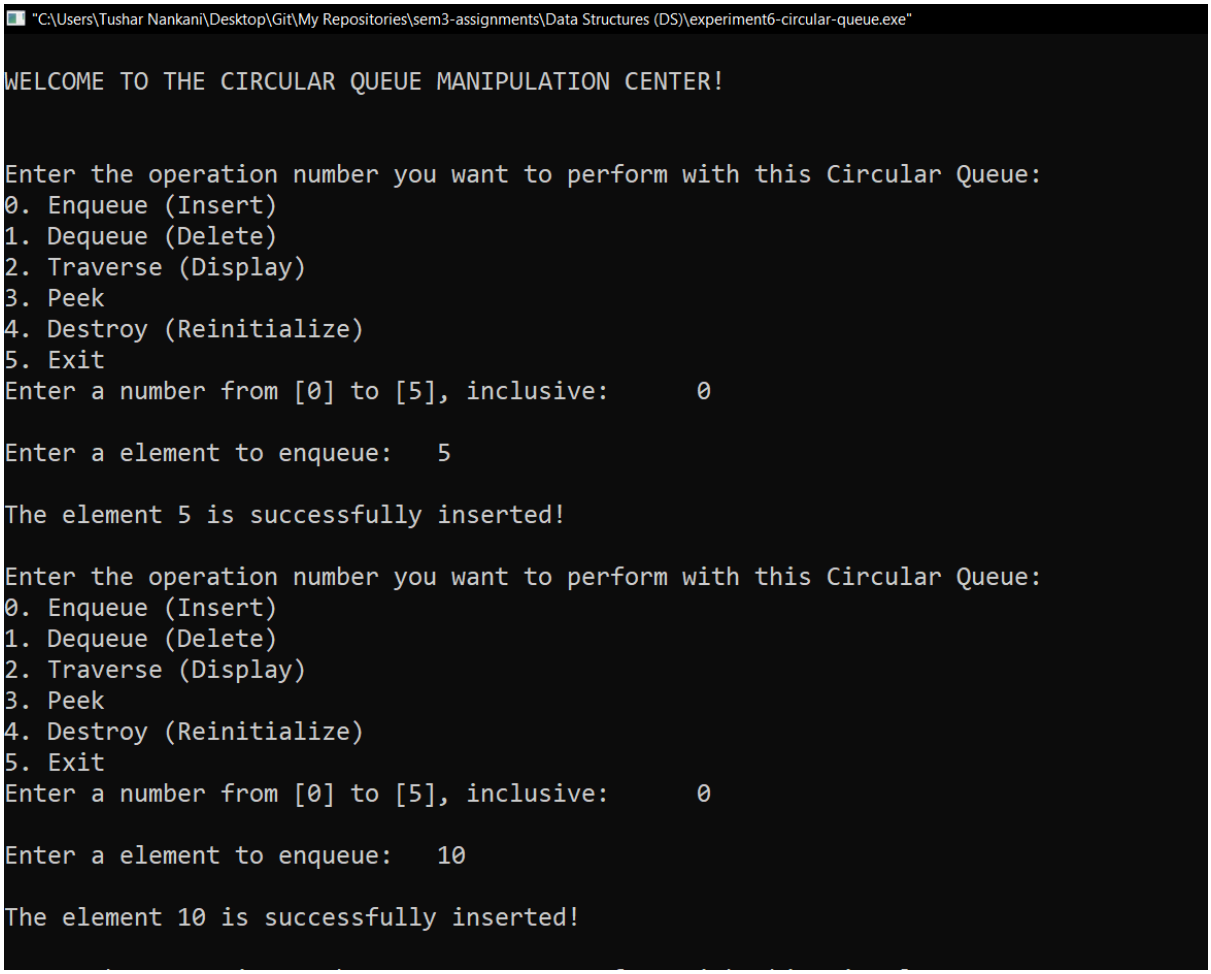
```
#include<stdio.h>
#include<unistd.h>

#define MAX 5
int queue[MAX];
int front, rear;

int options()
{
    int choice = -1;
    printf("\n\nEnter the operation number you want to perform with
this Circular Queue: \n");
    printf("0. Enqueue (Insert)\n");
    printf("1. Dequeue (Delete)\n");
    printf("2. Traverse (Display)\n");
    printf("3. Peek\n");
    printf("4. Destroy (Reinitialize)\n");
    printf("5. Exit\n");
    printf("Enter a number from [0] to [5], inclusive: \t");
    scanf("%d", &choice);
    return choice;
}

void initialize()
{
    front = rear = -1;
}

int isEmpty()
{
    if(front == rear) {
        front = rear = 0;
        printf("\n** The Queue is empty - Underflow Condition. **");
        return 1;
    }
    return 0;
```

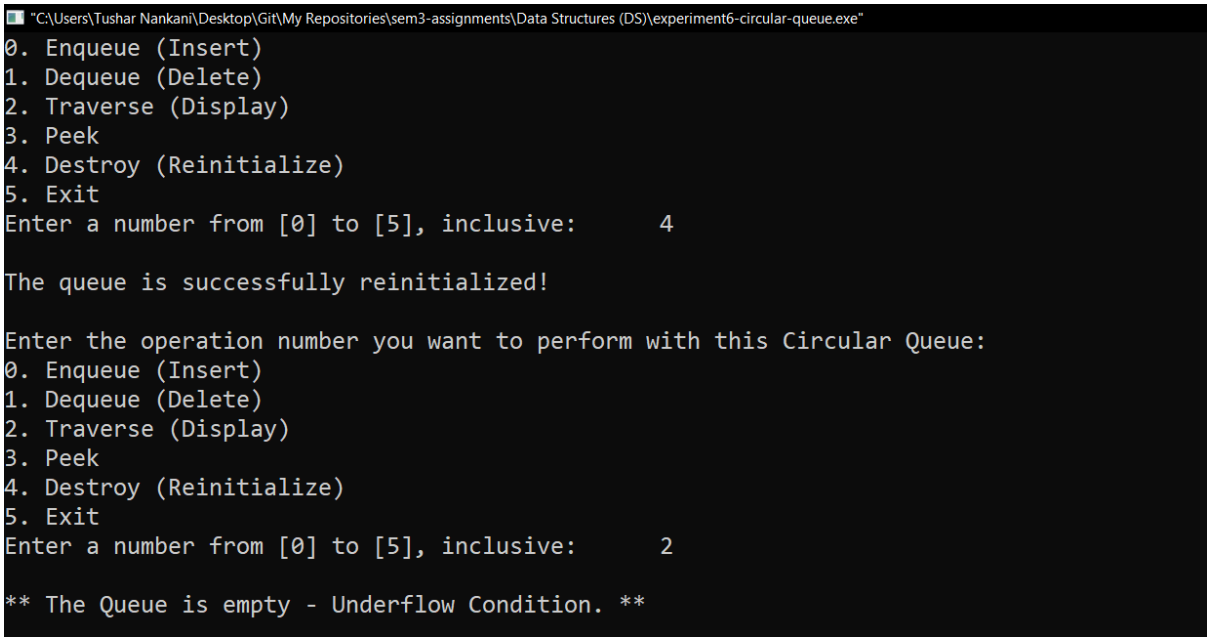
```
}

int isFull()
{
    if((rear + 1) % MAX == front){
        printf("\n** The Queue is full - Overflow Condition. **");
        return 1;
    }
    return 0;
}

void enqueue()
{
    if(!isFull())
    {
        int n;
        printf("\nEnter a element to enqueue:   ");
        scanf("%d", &n);
        queue[(rear++) % MAX] = n;
        printf("\nThe element %d is successfully inserted!", n);
    }
}

void dequeue()
{
    if(!isEmpty())
    {
        printf("\nThe element %d is successfully deleted!", queue[fr
ont]);
        if(front == rear)
            initialize();
        else
            front = (front + 1) % MAX;
    }
}

void traverse()
{
    if(!isEmpty())
    {
        printf("\nMAX SIZE :  %d", MAX);
        printf("\nFRONT    :  %d", front);
```

```
        printf("\nREAR     :  %d", rear);

        printf("\nThe elements of the queue are: \n");
        for(int i = front ; i != rear; i = (i + 1) % MAX) {
            printf(" %d  <-", queue[i]);
        }
        printf("  NULL\n");
    }
}

void peek()
{
    if(!isEmpty())
        printf("\nQueue's front element is: %d", queue[front]);
}

void destroy()
{
    initialize();
    printf("\nThe queue is successfully reinitialized!");
}

int main()
{
    // welcome message;
    printf("\nWELCOME TO THE CIRCULAR QUEUE MANIPULATION CENTER!\n")
;
    initialize();
    while(1)
    {
        switch(options())
        {
        case 0:
            enqueue();
            break;
        case 1:
            dequeue();
            break;
        case 2:
            traverse();
            break;
        case 3:
```

```
            peek();
            break;
        case 4:
            destroy();
            break;
        case 5:
            exit(0);
        default:
            printf("\nINVALID OPTION.");
        }
    }
    return 0;
}
```


"C:\Users\Tushar Nankani\Desktop\Git\My Repositories\sem3-assignments\Data Structures (DS)\experiment6-circular-queue.exe"
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive: 2
MAX SIZE : 5
FRONT : 1
REAR : 4
The elements of the queue are:
15 <- 20 <- 25 <- NULL
Enter the operation number you want to perform with this Circular Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive:


"C:\Users\Tushar Nankani\Desktop\Git\My Repositories\sem3-assignments\Data Structures (DS)\experiment6-circular-queue.exe"
WELCOME TO THE CIRCULAR QUEUE MANIPULATION CENTER!
Enter the operation number you want to perform with this Circular Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive: 0
Enter a element to enqueue: 5
The element 5 is successfully inserted!
Enter the operation number you want to perform with this Circular Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive: 0
Enter a element to enqueue: 10
The element 10 is successfully inserted!


"C:\Users\Tushar Nankani\Desktop\Git\My Repositories\sem3-assignments\Data Structures (DS)\experiment6-circular-queue.exe"
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive: 4
The queue is successfully reinitialized!
Enter the operation number you want to perform with this Circular Queue:
0. Enqueue (Insert)
1. Dequeue (Delete)
2. Traverse (Display)
3. Peek
4. Destroy (Reinitialize)
5. Exit
Enter a number from [0] to [5], inclusive: 2
** The Queue is empty - Underflow Condition. **

DOUBLYLinkedList

A Doubly Linked List (DLL) contains an extra pointer, typically called the previous pointer, together with the next pointer and data which are there in the singly linked list.

Basic Operations

Following are the basic operations supported by a list:

Insertion - Adds an element at the beginning of the list.

Deletion - Deletes an element at the beginning of the list.

Insert Last - Adds an element at the end of the list.

Delete Last - Deletes an element from the end of the list.

Insert After - Adds an element after an item of the list.

Delete - Deletes an element from the list using the key.

Display - Displays the complete list

Advantages over singly linked list

1) A DLL can be traversed in both forward and backward directions.

2) The delete operation in DLL is more efficient if a pointer to the node to be deleted is given.

3) We can quickly insert a new node before a given node.

Disadvantages over singly linked list

1) Every node of DLL Require extra space for an previous pointer. It is possible to implement DLL with single pointer though (See this and this).

2) All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with next pointers. For example in following functions for insertions at different positions, we need 1 or 2 extra steps to set previous pointer.

Name: **Tushar Nankani** Roll No: **1902112** Batch: **C23**

IMPLEMENTATION OF **DOUBLY LINKED LIST**

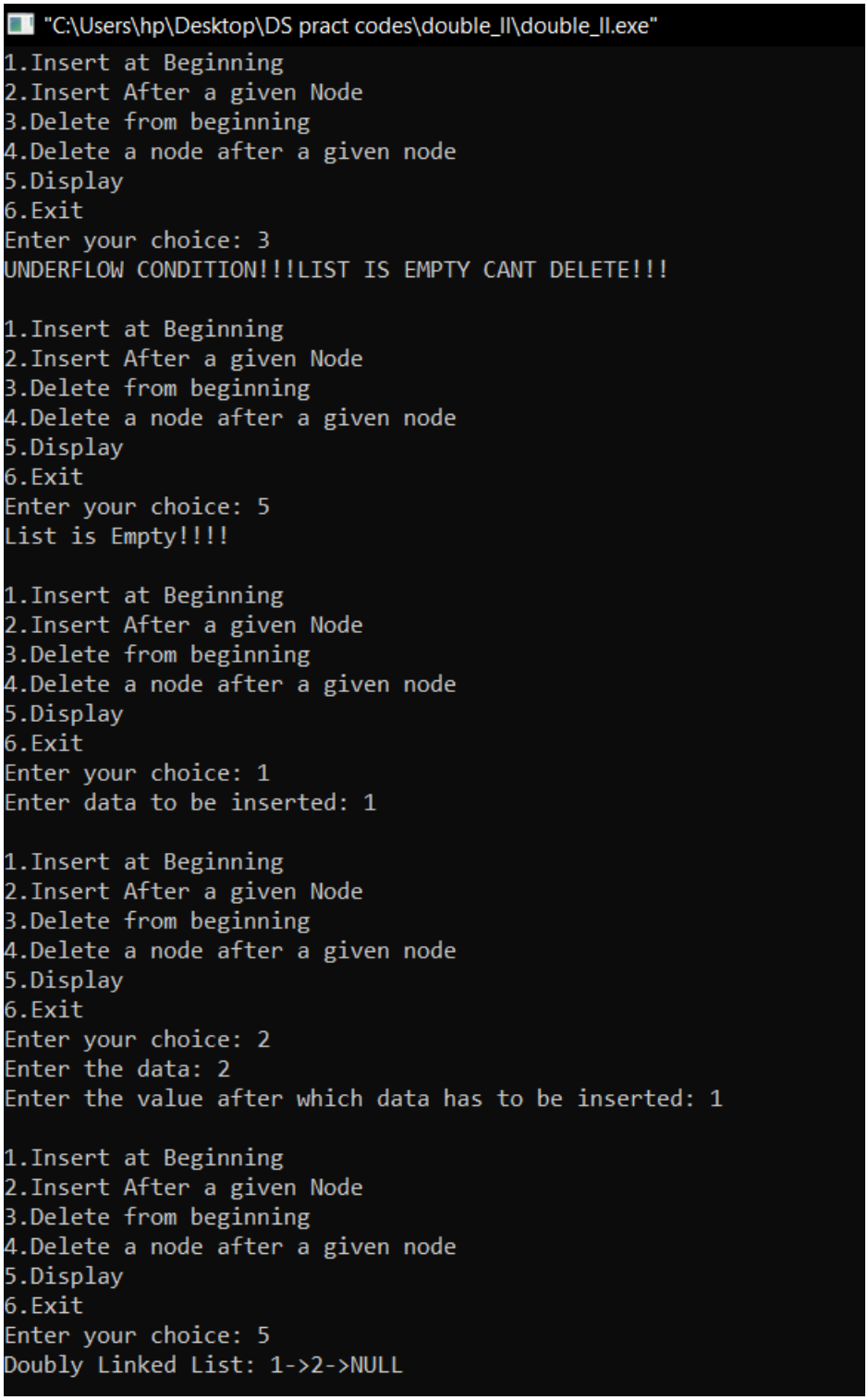
```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    struct node* next;
    struct node* prv;
    int data;
};

struct node* newNode(int x) //function to allocate memory
to our node
{
    struct node* temp=(struct node*)malloc(sizeof(struct n
ode));
    temp->next=NULL;
    temp->prv=NULL;
    temp->data=x;
    return temp;
}

struct node* insert_beg(struct node* head)
{
    int data;
    printf("Enter data to be inserted: ");
    scanf("%d",&data);
    struct node* temp=newNode(data);
    if(head)
    {
        head->prv=temp;
        temp->next=head;
        return temp;
    }
    else
    {
        return temp;
    }
```

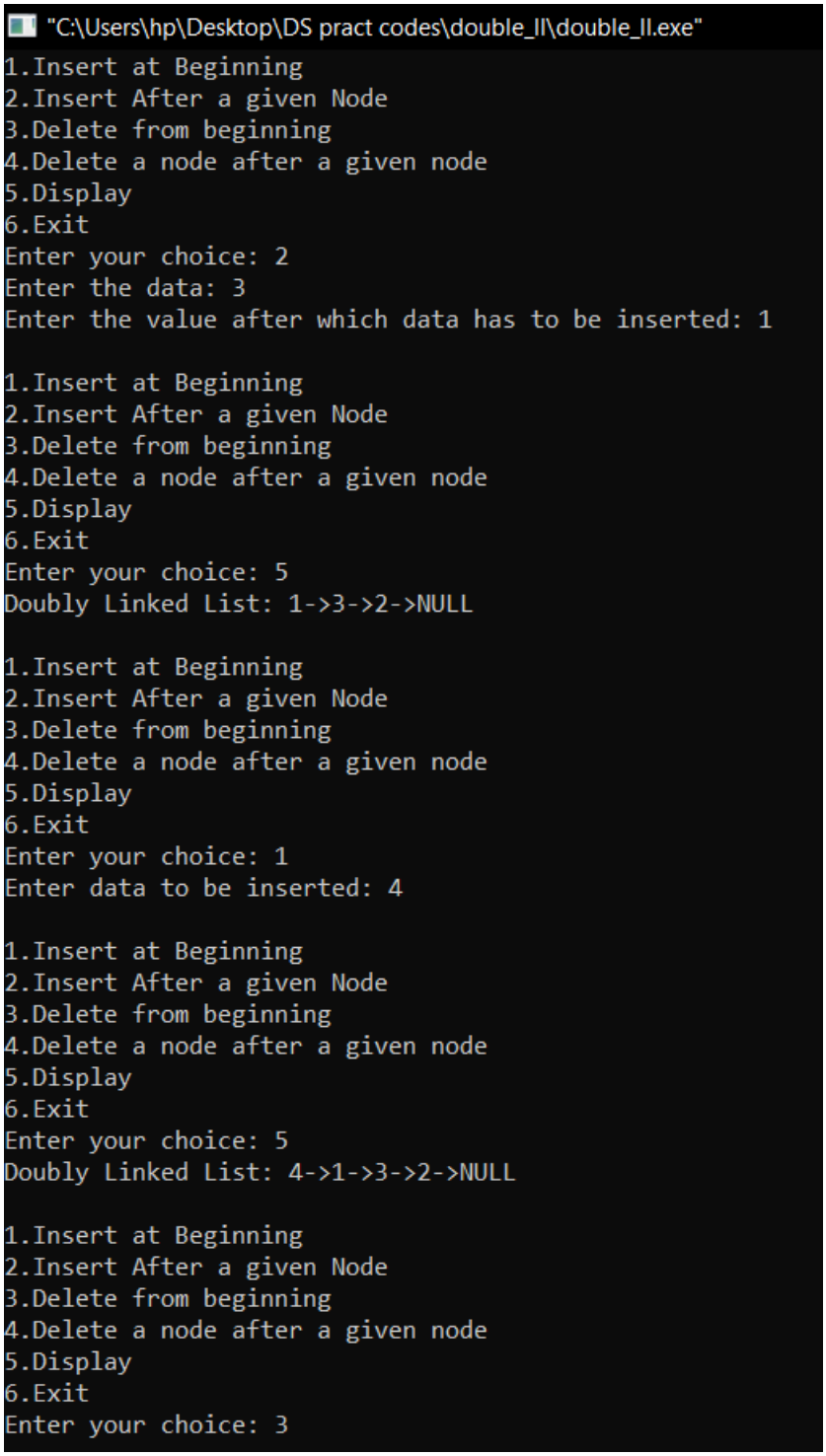
```
}
struct node* insert_after(struct node* head)
{
    int data,val;
    printf("Enter the data: ");
    scanf("%d",&data);
    printf("Enter the value after which data has to be ins
erted: ");
    scanf("%d",&val);
    struct node* temp=newNode(data);
    struct node* ptr=head;
    while(ptr->data!=val)
    {
        ptr=ptr->next;
    }
    if(ptr->next)
    {
        temp->prv=ptr;
        temp->next=ptr->next;
        ptr->next->prv=temp;
        ptr->next=temp;
    }
    else
    {
        temp->prv=ptr;
        ptr->next=temp;
    }
    return head;
}
struct node* delete_beg(struct node* head)
{
    if(head)
    {
        struct node* ptr=head;
        head=head->next;
        head->prv=NULL;
        free(ptr);
        return head;
    }
    else
    {
```

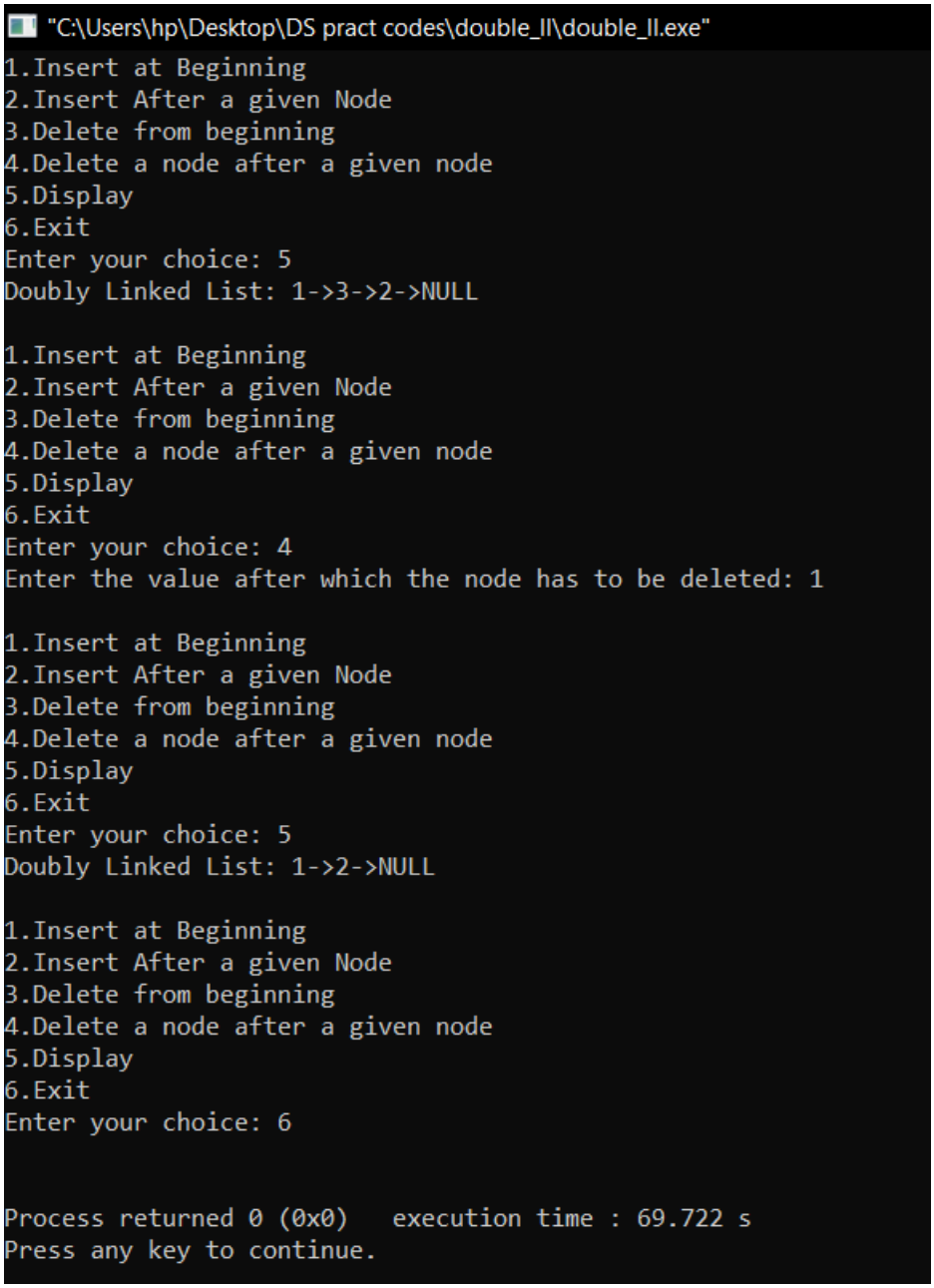
```
        printf("UNDERFLOW CONDITION! LIST IS EMPTY CANT DE
LETE!\n");
        return NULL;
    }
}

struct node* delete_after(struct node* head)
{
    struct node *ptr,*temp;
    int val;
    printf("Enter the value after which the node has to be
 deleted: ");
    scanf("%d",&val);
    ptr=head;
    while(ptr->data!=val)
        ptr=ptr->next;
    temp=ptr->next;
    if(temp->next)
    {
        ptr->next=temp->next;
        temp->next->prv=ptr;
        free(temp);
        return head;
    }
    else
    {
        ptr->next=NULL;
        free(temp);
        return head;
    }
}
void display(struct node* head)
{
    if(!head)
    {
        printf("List is Empty!\n");
        return;
    }
    printf("Doubly Linked List: ");
    struct node* ptr=head;
    while(ptr)
    {
```

```
        printf("%d->",ptr->data);
        ptr=ptr->next;
    }
    printf("NULL\n");
}

int main()
{
    struct node* head=NULL;
    int c=1,choice;
    while(c)
    {
        printf("1.Insert at Beginning\n2.Insert After a gi
ven Node\n3.Deletefrom beginning\n4.Delete a node after a
given node\n5.Display\n6.Exit\nEnter your choice: ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            {
                head=insert_beg(head);
                break;
            }
            case 2:
            {
                head=insert_after(head);
                break;
            }
            case 3:
            {
                head=delete_beg(head);
                break;
            }
            case 4:
            {
                head=delete_after(head);
                break;
            }
            case 5:
            {
                display(head);
                break;
```

```
            }
            case 6:
            {
                c=0;
                break;
            }
        }
        printf("\n");
    }
    return 0;
}
```


"C:\Users\hp\Desktop\DS pract codes\double_ll\double_ll.exe"
1.Insert at Beginning
2.Insert After a given Node
3.Delete from beginning
4.Delete a node after a given node
5.Display
6.Exit
Enter your choice: 3
UNDERFLOW CONDITION!!!LIST IS EMPTY CANT DELETE!!!
1.Insert at Beginning
2.Insert After a given Node
3.Delete from beginning
4.Delete a node after a given node
5.Display
6.Exit
Enter your choice: 5
List is Empty!!!!
1.Insert at Beginning
2.Insert After a given Node
3.Delete from beginning
4.Delete a node after a given node
5.Display
6.Exit
Enter your choice: 1
Enter data to be inserted: 1
1.Insert at Beginning
2.Insert After a given Node
3.Delete from beginning
4.Delete a node after a given node
5.Display
6.Exit
Enter your choice: 2
Enter the data: 2
Enter the value after which data has to be inserted: 1
1.Insert at Beginning
2.Insert After a given Node
3.Delete from beginning
4.Delete a node after a given node
5.Display
6.Exit
Enter your choice: 5
Doubly Linked List: 1->2->NULL


"C:\Users\hp\Desktop\DS pract codes\double_ll\double_ll.exe"
1.Insert at Beginning
2.Insert After a given Node
3.Delete from beginning
4.Delete a node after a given node
5.Display
6.Exit
Enter your choice: 2
Enter the data: 3
Enter the value after which data has to be inserted: 1
1.Insert at Beginning
2.Insert After a given Node
3.Delete from beginning
4.Delete a node after a given node
5.Display
6.Exit
Enter your choice: 5
Doubly Linked List: 1->3->2->NULL
1.Insert at Beginning
2.Insert After a given Node
3.Delete from beginning
4.Delete a node after a given node
5.Display
6.Exit
Enter your choice: 1
Enter data to be inserted: 4
1.Insert at Beginning
2.Insert After a given Node
3.Delete from beginning
4.Delete a node after a given node
5.Display
6.Exit
Enter your choice: 5
Doubly Linked List: 4->1->3->2->NULL
1.Insert at Beginning
2.Insert After a given Node
3.Delete from beginning
4.Delete a node after a given node
5.Display
6.Exit
Enter your choice: 3


"C:\Users\hp\Desktop\DS pract codes\double_ll\double_ll.exe"
1.Insert at Beginning
2.Insert After a given Node
3.Delete from beginning
4.Delete a node after a given node
5.Display
6.Exit
Enter your choice: 5
Doubly Linked List: 1->3->2->NULL
1.Insert at Beginning
2.Insert After a given Node
3.Delete from beginning
4.Delete a node after a given node
5.Display
6.Exit
Enter your choice: 4
Enter the value after which the node has to be deleted: 1
1.Insert at Beginning
2.Insert After a given Node
3.Delete from beginning
4.Delete a node after a given node
5.Display
6.Exit
Enter your choice: 5
Doubly Linked List: 1->2->NULL
1.Insert at Beginning
2.Insert After a given Node
3.Delete from beginning
4.Delete a node after a given node
5.Display
6.Exit
Enter your choice: 6
Process returned 0 (0x0) execution time : 69.722 s
Press any key to continue.

IMPLEMENTATION OF STACK & QUEUE USING LINKEDLIST

A stack can be easily implemented through the linked list. In stack Implementation, a stack contains a top pointer. which is "head" of the stack where pushing and popping items happens at the head of the list. first node have null in link field and second node link have first node address in link field and so on and last node address in "top" pointer.

The main advantage of using linked list over an arrays is that it is possible to implements a stack that can shrink or grow as much as needed. In using array will put a restriction to the maximum capacity of the array which can lead to stack overflow. Here each new node will be dynamically allocate. so overflow is not possible.

Stack Implementation of Linked list

Algorithm

1. Declare and initialize necessary variables such as struct node *top, *p, top = NULL

2. For push operation,

- check for memory full

if((p=(nodetype*) malloc (sizeof(nodetype))) == NULL) print "Memory Exhausted"

Else

-take data to be inserted say x

-Create an empty node p and assign data x to its info field

i.e. p->info = x;

p->next = NULL

if(top!=NULL)

p->next = top

3. For next push operation, goto step 2.

4. For pop operation,

if top = NULL

print "stack is empty"

else

-temp = top, top = top->next

-display popped item as top->info, delete temp

(B) Queue Implementation of Linked List

Algorithm

1. Declare and initialize necessary variables such as struct node *front, *rear etc
2. For enqueue operation,
-take input data to be inserted
-create an empty node and assign data to its into field
i.e. p->info=data, p->next = NULL
if front = NULL front = p
else
rear->next = p, rear = p
3. For next enqueue operation, goto step 2
4. For dequeue operation,
if front = NULL
print "Queue is empty"
else
-create a node pointer temp
-temp = front
-front = front->next
-display dequeued item as temp->data
-delete temp
5. For dequeue of next data item, goto step 4

Name: **Tushar Nankani** Roll No: **1902112** Batch: **C23**

IMPLEMENTATION OF **STACK & QUEUE** USING **LINKEDLIST**

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    struct node* next;
    int data;
};
struct node* newNode(int x) //function to allocate memory to our nod
e
{
    struct node* temp=(struct node*)malloc(sizeof(struct node));
    temp->next=NULL;
    temp->data=x;
    return temp;
}
void display(struct node* head)
{
    struct node*temp=head;
    while(temp)
    {
        printf("%d->",temp->data);
        temp=temp->next;
    }
    printf("NULL\n");
}
int main()
{
int c1=1,choice1,data;
while(c1)
{
printf("1.Stack\n2.Queue\n3.Exit\nEnter your choice: ");
scanf("%d",&choice1);
switch(choice1)
{
case 1:
{
```

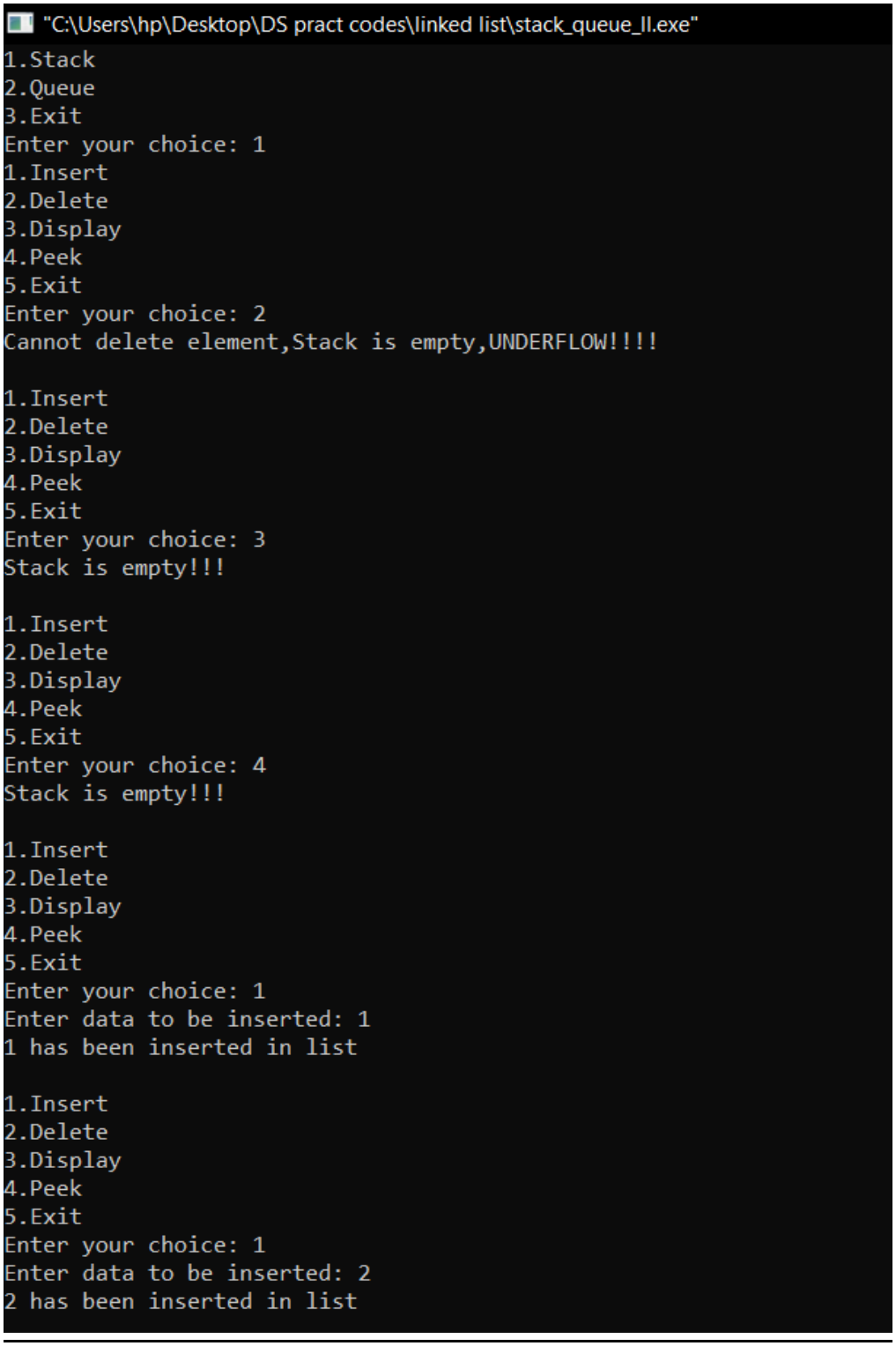
```
struct node* head=NULL;
int c2=1,choice2;
while(c2)
{

printf("1.Insert\n2.Delete\n3.Display\n4.Peek\n5.Exit\nEnter your
choice: ");
scanf("%d",&choice2);
switch(choice2)
{
case 1:
{
printf("Enter data to be inserted: ");
scanf("%d",&data);
if(!head) //if stack is empty create a
new head
{
head=newNode(data);
}
else //traverse stack till the end and
add a new node
{
struct node* temp=head;
while(temp->next)
{
temp=temp->next;
}
temp->next=newNode(data);
temp=temp->next;
}
printf("%d has been inserted in
list\n",data);
break;
}
case 2:
{
if(!head)//UNDERFLOW condition
{
printf("Cannot delete
element,Stack is empty,UNDERFLOW!!!!\n");
}
else if(!head->next)//if only 1 element
```

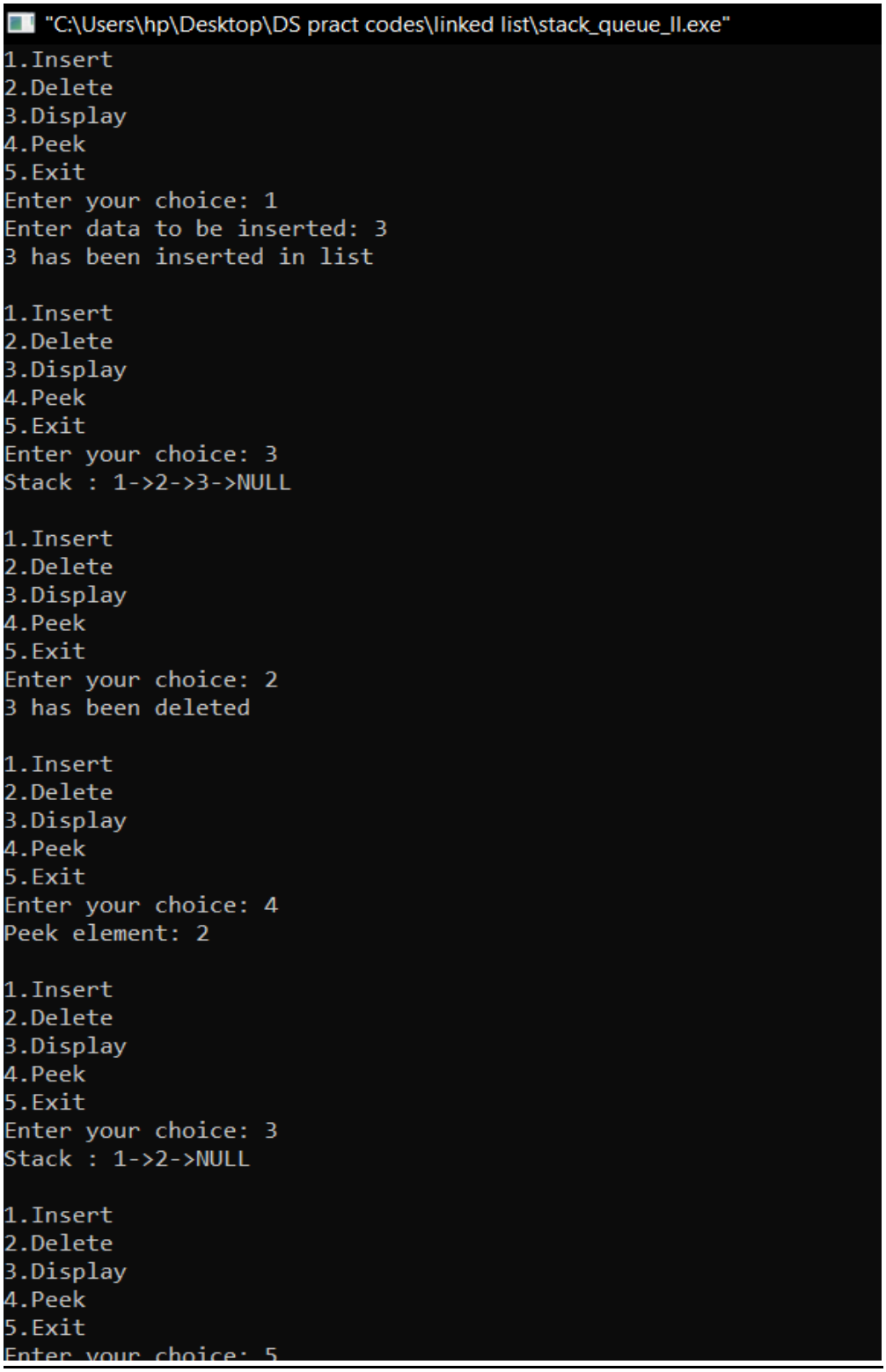
```
is present make head as NULL,otherwise make list empty
{
printf("%d has been
deleted\n",head->data);
struct node* temp=head;
head=NULL;
free(temp);
}
else//traverse till the second last
element ans set its next is NULL,and free the last node
{
struct node* cur=head;
struct node* n=cur->next;
while(n->next)
{
cur=n;
n=n->next;
}
cur->next=NULL;
printf("%d has been
deleted\n",n->data);
free(n);
}
break;
}
case 3:
{
if(head)
{
printf("Stack : ");
display(head);
}
else
printf("Stack is empty!!!\n");
break;
}
case 4:
{
if(head)
{
struct node* temp=head;
while(temp->next)
```

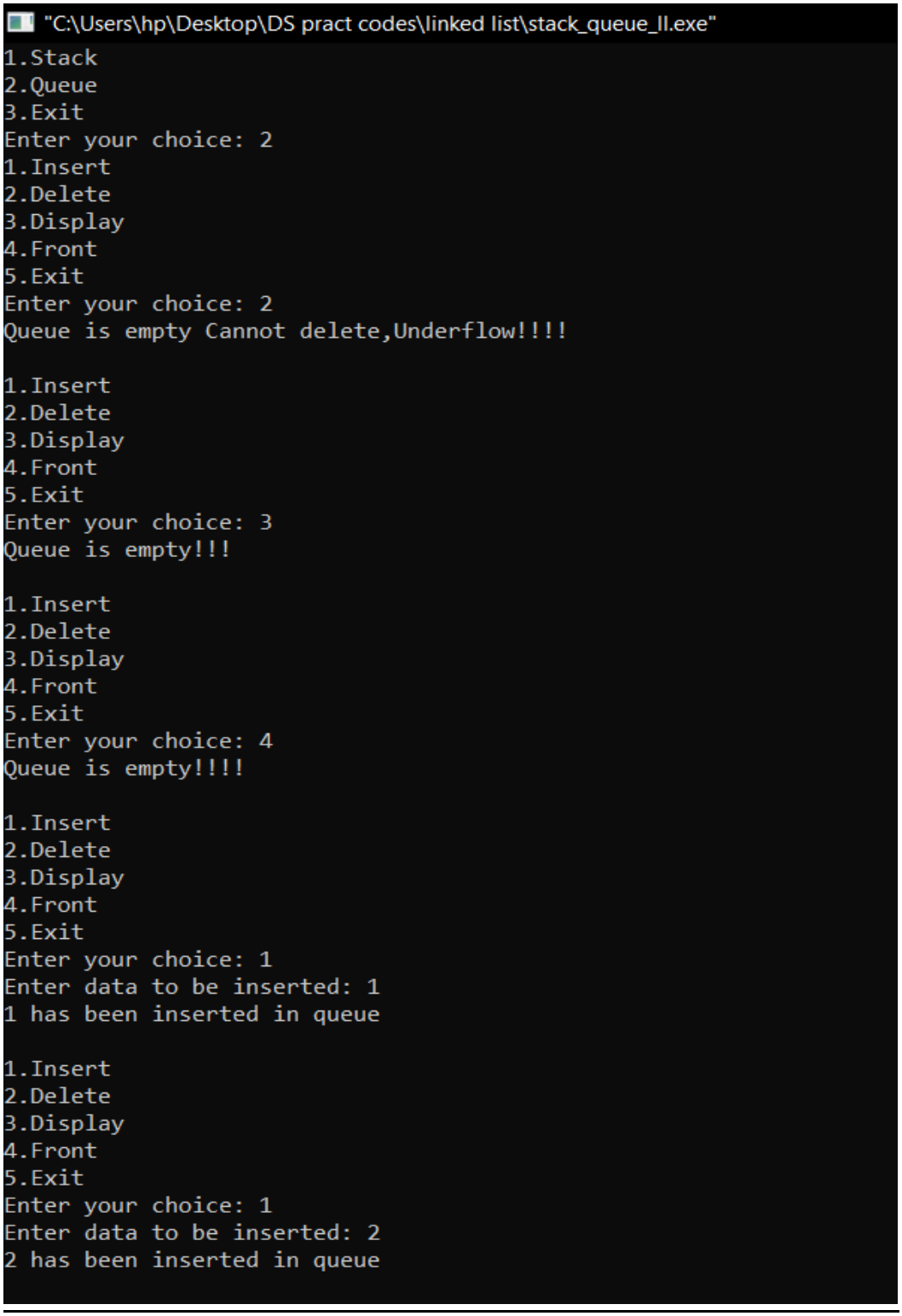
```
{
temp=temp->next;
}
printf("Peek
element: %d\n",temp->data);
}
else
printf("Stack is empty!!!\n");
break;
}
case 5:
{
c2=0;
break;
}
default:
{
printf("Invalid choice,Plz try
again!!!!\n");
}
}
printf("\n");
}
break;
}
case 2:
{
struct node* head=NULL;
int c2=1,choice2;
while(c2)
{

printf("1.Insert\n2.Delete\n3.Display\n4.Front\n5.Exit\nEnter your
choice: ");
scanf("%d",&choice2);
switch(choice2)
{
case 1:
{
printf("Enter data to be inserted: ");
scanf("%d",&data);
if(!head) //if queue is empty create a
```

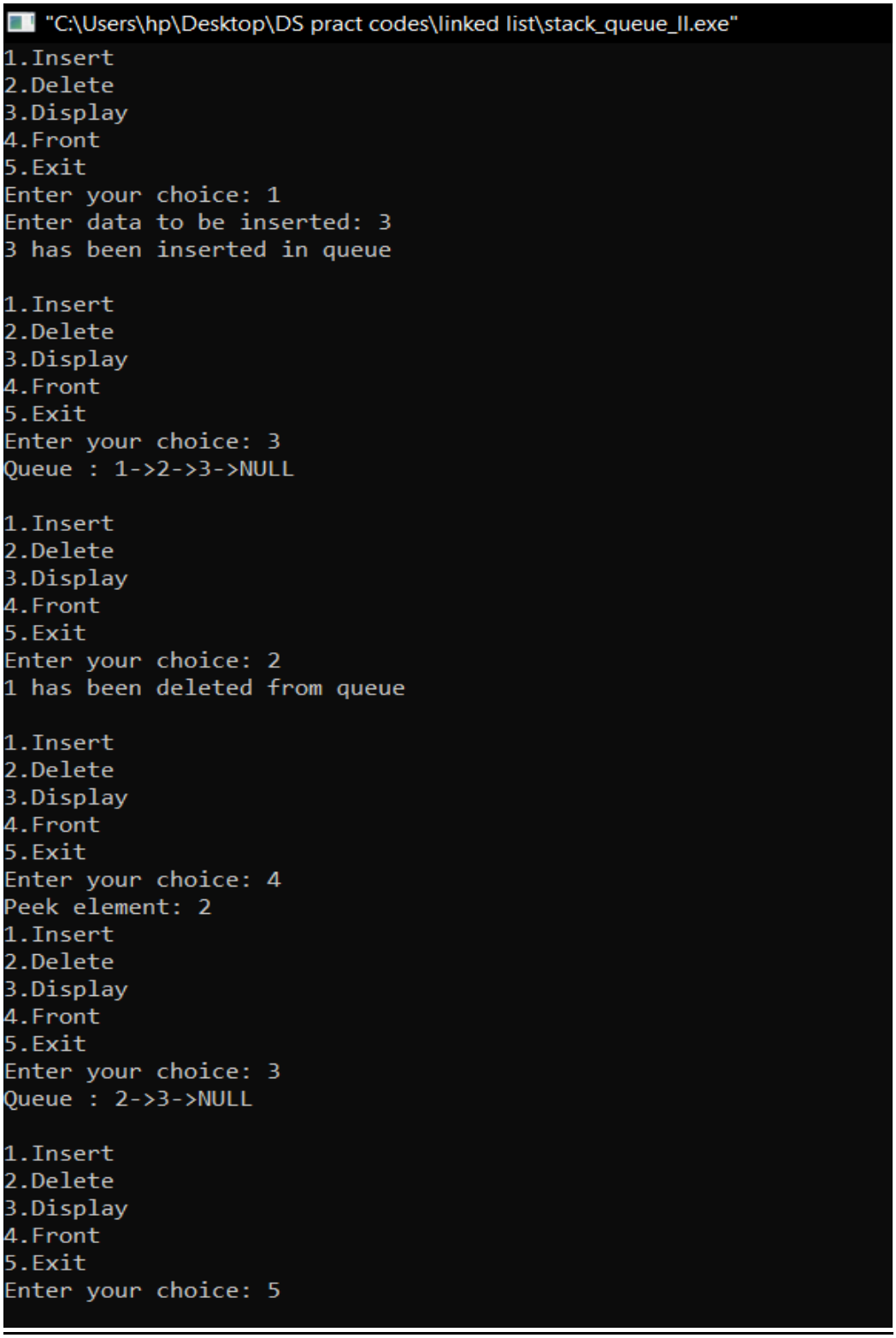
```
new head
{
head=newNode(data);
}
else //traverse queue till the end and
add a new node
{
struct node* temp=head;
while(temp->next)
{
temp=temp->next;
}
temp->next=newNode(data);
temp=temp->next;
}
printf("%d has been inserted in
queue\n",data);
break;
}
case 2:
{
if(head)
{
printf("%d has been deleted from
queue\n",head->data);
head=head->next;
}
else
printf("Queue is empty Cannot
delete,Underflow!!!!\n");
break;
}
case 3:
{
if(head)
{
printf("Queue : ");
display(head);
}
else
printf("Queue is empty!!!\n");
break;
```

```
}
case 4:
{
if(head)
{
printf("Peek
element: %d",head->data);
}
else
printf("Queue is empty!!!!\n");
break;
}
case 5:
{
c2=0;
break;
}
default:
{
printf("Invalid choice!!!\n");
}
}
printf("\n");
}
break;
}
case 3:
{
c1=0;
break;
}
default:
{
printf("Invalid choice!!!!\n");
}
}
}
 return 0;
}
```


"C:\Users\hp\Desktop\DS pract codes\linked list\stack_queue_II.exe"
1.Stack
2.Queue
3.Exit
Enter your choice: 1
1.Insert
2.Delete
3.Display
4.Peek
5.Exit
Enter your choice: 2
Cannot delete element,Stack is empty,UNDERFLOW!!!!
1.Insert
2.Delete
3.Display
4.Peek
5.Exit
Enter your choice: 3
Stack is empty!!!
1.Insert
2.Delete
3.Display
4.Peek
5.Exit
Enter your choice: 4
Stack is empty!!!
1.Insert
2.Delete
3.Display
4.Peek
5.Exit
Enter your choice: 1
Enter data to be inserted: 1
1 has been inserted in list
1.Insert
2.Delete
3.Display
4.Peek
5.Exit
Enter your choice: 1
Enter data to be inserted: 2
2 has been inserted in list


"C:\Users\hp\Desktop\DS pract codes\linked list\stack_queue_II.exe"
1.Insert
2.Delete
3.Display
4.Peek
5.Exit
Enter your choice: 1
Enter data to be inserted: 3
3 has been inserted in list
1.Insert
2.Delete
3.Display
4.Peek
5.Exit
Enter your choice: 3
Stack : 1->2->3->NULL
1.Insert
2.Delete
3.Display
4.Peek
5.Exit
Enter your choice: 2
3 has been deleted
1.Insert
2.Delete
3.Display
4.Peek
5.Exit
Enter your choice: 4
Peek element: 2
1.Insert
2.Delete
3.Display
4.Peek
5.Exit
Enter your choice: 3
Stack : 1->2->NULL
1.Insert
2.Delete
3.Display
4.Peek
5.Exit
Enter your choice: 5


"C:\Users\hp\Desktop\DS pract codes\linked list\stack_queue_ll.exe"
1.Stack
2.Queue
3.Exit
Enter your choice: 2
1.Insert
2.Delete
3.Display
4.Front
5.Exit
Enter your choice: 2
Queue is empty Cannot delete,Underflow!!!!
1.Insert
2.Delete
3.Display
4.Front
5.Exit
Enter your choice: 3
Queue is empty!!!
1.Insert
2.Delete
3.Display
4.Front
5.Exit
Enter your choice: 4
Queue is empty!!!!
1.Insert
2.Delete
3.Display
4.Front
5.Exit
Enter your choice: 1
Enter data to be inserted: 1
1 has been inserted in queue
1.Insert
2.Delete
3.Display
4.Front
5.Exit
Enter your choice: 1
Enter data to be inserted: 2
2 has been inserted in queue


"C:\Users\hp\Desktop\DS pract codes\linked list\stack_queue_II.exe"
1.Insert
2.Delete
3.Display
4.Front
5.Exit
Enter your choice: 1
Enter data to be inserted: 3
3 has been inserted in queue
1.Insert
2.Delete
3.Display
4.Front
5.Exit
Enter your choice: 3
Queue : 1->2->3->NULL
1.Insert
2.Delete
3.Display
4.Front
5.Exit
Enter your choice: 2
1 has been deleted from queue
1.Insert
2.Delete
3.Display
4.Front
5.Exit
Enter your choice: 4
Peek element: 2
1.Insert
2.Delete
3.Display
4.Front
5.Exit
Enter your choice: 3
Queue : 2->3->NULL
1.Insert
2.Delete
3.Display
4.Front
5.Exit
Enter your choice: 5

Graph Traversal - BFS and DFS

Graph traversal is a technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

DFS (Depth First Search)

BFS (Breadth First Search)

BFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal.

We use the following steps to implement BFS traversal...

Step 1 - Define a Queue of size total number of vertices in the graph.

Step 2 - Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.

Step 3 - Visit all the non-visited adjacent vertices of the vertex which is at front of the Queue and insert them into the Queue.

Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

Step 5 - Repeat steps 3 and 4 until queue becomes empty.

Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

DFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal.

Step 1 - Define a Stack of size total number of vertices in the graph.

Step 2 - Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

Step 3 - Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on to the stack.

Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

Step 5 - When there is no new vertex to visit then use back tracking and pop one vertex.

Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

Name: **Tushar Nankani** Roll No: **1902112** Batch: **C23**

IMPLEMENTATION OF GRAPH TRVERSAL TECHNIQUES - **BFS + DFS**

```
#include<stdio.h>
#include<unistd.h>
#define MAX 20

struct stack
{
    int data[MAX];
    int top;
};

struct Queue
{
    int data[MAX];
    int front,rear;
};


void push(struct stack *s,int d)
{
    if(s->top==MAX-1)
        printf("\n\tStack Overflows->");
    else
    {
        s->top++;
        s->data[s->top]=d;
    }
}

int pop(struct stack *s)
{
    if(s->top==-1)
        printf("\n\tStack Underflows->");
    else
        return s->data[s->top--];
}

void initialize(struct stack *s)
```

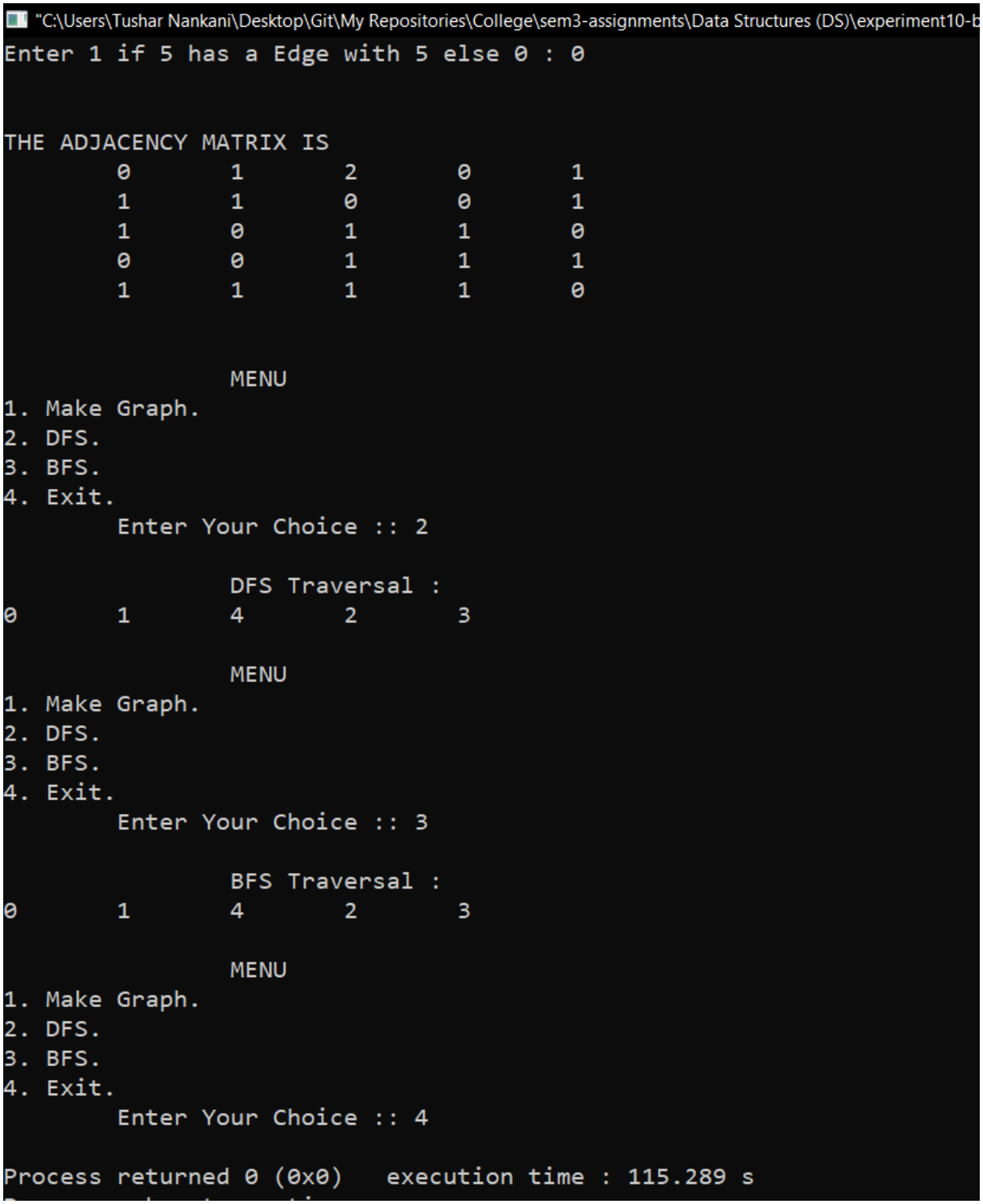
```
{
    s->top=-1;
}


void insert(struct Queue *q,int d)
{
    if(q->rear==MAX-1)
        printf("\n\tQueue is Full->");
    else
    {
        q->rear++;
        q->data[q->rear]=d;
        if(q->front==-1)
            q->front=0;
    }
}

void initializeQ(struct Queue *q)
{
    q->front=q->rear=-1;
}

int delete(struct Queue *q)
{
    if(q->rear==-1)
        printf("\nQueue is empty..");
    else
    {
        int d;
        d=q->data[q->front];
        q->front++;
        if(q->front>q->rear)
        q->front=q->rear=-1;
        return d;
    }
}

void dfs(int a[][10],int n)
{
    struct stack s;
    int visited[10];
```

```
    int i,j,v;
    initialize(&s);
    for(i=0;i<n;i++)
    visited[i]=0; //setting all unvisited.


    visited[0]=1; //visit first vertex

    printf("\n\t\tDFS Traversal :\n0\t");
    push(&s,0);
    while(s.top!=-1)      // until stack empty,
    {
        v=-1;
            // get an unvisited vertex adjacent to stack top
        for(j=0;j<n;j++)
            if(a[s.data[s.top]][j]==1 && visited[j]==0)
            {
                v=j;
                break;
            }
        if(v==-1)                    // if no such vertex,
            v=pop(&s);
        else                           // if it exists,
        {
            visited[v]=1;  // mark it
            printf("%d\t",v);                 // display it
            push(&s,v);                 // push it
        }
    }  // end while
}


void bfs(int a[][10],int n)
{
    struct Queue q;
    int visited[10];
    int i,j,v;
    initializeQ(&q);
    for(i=0;i<n;i++)
    visited[i]=0;
     //setting all unvisited.
    visited[0]=1; //visit first vertex
```

```
    printf("\n\t\tBFS Traversal :\n0\t");
    insert(&q,0);
    while(q.front!=-1)      // until queue empty,
    {
        v=delete(&q);
            // get an unvisited vertex adjacent to stack top
    for(j=0;j<n;j++)
        if(a[v][j]==1 && visited[j]==0)
        {
            visited[j]=1;
            printf("%d\t",j);
            insert(&q,j);
        }
    }  // end while
}


int main()
{
    int n,i,s,ch,j,a[10][10];
    char c,dummy;
    while(1)
    {
        printf("\n\n\t\tMENU\n1. Make Graph.\n2. DFS.\n3. BFS. \n4.
Exit.");
        printf("\n\tEnter Your Choice :: ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                printf("\nENTER THE NUMBER OF VERTICES :: " );
                scanf("%d",&n);
                for (i=0;i<n;i++)
                {
                 for (j=0;j<n;j++)
                 {
                   printf("\nEnter 1 if %d has a Edge with %d else 0
 : ", i+1, j+1);
                   scanf( "%d",&a[i][j]);
                 }
                }
                printf("\n\nTHE ADJACENCY MATRIX IS\n" );
```

```
                for(i=0;i<n;i++)
                {
                  for(j=0;j<n;j++)
                   {
                    printf("\t%d",a[i][j]);
                   }
                  printf("\n");
                 }
            break;
            case 2:
                dfs(a,n);
                break;
            case 3:
                bfs(a,n);
                break;
            case 4:
                exit(0);
                break;
        }
    }
}
```

```
                MENU
1. Make Graph.
2. DFS.
3. BFS.
4. Exit.
        Enter Your Choice :: 1

ENTER THE NUMBER OF VERTICES :: 5

Enter 1 if 1 has a Edge with 1 else 0 : 0

Enter 1 if 1 has a Edge with 2 else 0 : 1

Enter 1 if 1 has a Edge with 3 else 0 : 2

Enter 1 if 1 has a Edge with 4 else 0 : 0

Enter 1 if 1 has a Edge with 5 else 0 : 1

Enter 1 if 2 has a Edge with 1 else 0 : 1

Enter 1 if 2 has a Edge with 2 else 0 : 1

Enter 1 if 2 has a Edge with 3 else 0 : 0

Enter 1 if 2 has a Edge with 4 else 0 : 0

Enter 1 if 2 has a Edge with 5 else 0 : 1

Enter 1 if 3 has a Edge with 1 else 0 : 1

Enter 1 if 3 has a Edge with 2 else 0 : 0

Enter 1 if 3 has a Edge with 3 else 0 : 1

Enter 1 if 3 has a Edge with 4 else 0 : 1

Enter 1 if 3 has a Edge with 5 else 0 : 0
```


"C:\Users\Tushar Nankani\Desktop\Git\My Repositories\College\sem3-assignments\Data Structures (DS)\experiment10-b
Enter 1 if 5 has a Edge with 5 else 0 : 0
THE ADJACENCY MATRIX IS
0 1 2 0 1
1 1 0 0 1
1 0 1 1 0
0 0 1 1 1
1 1 1 1 0
MENU
1. Make Graph.
2. DFS.
3. BFS.
4. Exit.
Enter Your Choice :: 2
DFS Traversal :
0 1 4 2 3
MENU
1. Make Graph.
2. DFS.
3. BFS.
4. Exit.
Enter Your Choice :: 3
BFS Traversal :
0 1 4 2 3
MENU
1. Make Graph.
2. DFS.
3. BFS.
4. Exit.
Enter Your Choice :: 4
Process returned 0 (0x0) execution time : 115.289 s

Implementation of Binary Search Tree Using LinkedList

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value.

While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following are the basic operations of a tree –

Search – Searches an element in a tree.

Insert – Inserts an element in a tree.

Pre-order Traversal – Traverses a tree in a pre-order manner.

In-order Traversal – Traverses a tree in an in-order manner.

Post-order Traversal – Traverses a tree in a post-order manner.

Name: **Tushar Nankani** Roll No: **1902112** Batch: **C23**

IMPLEMENTATION OF BINARY SEARCH TREE ADT USING LINKEDLIST

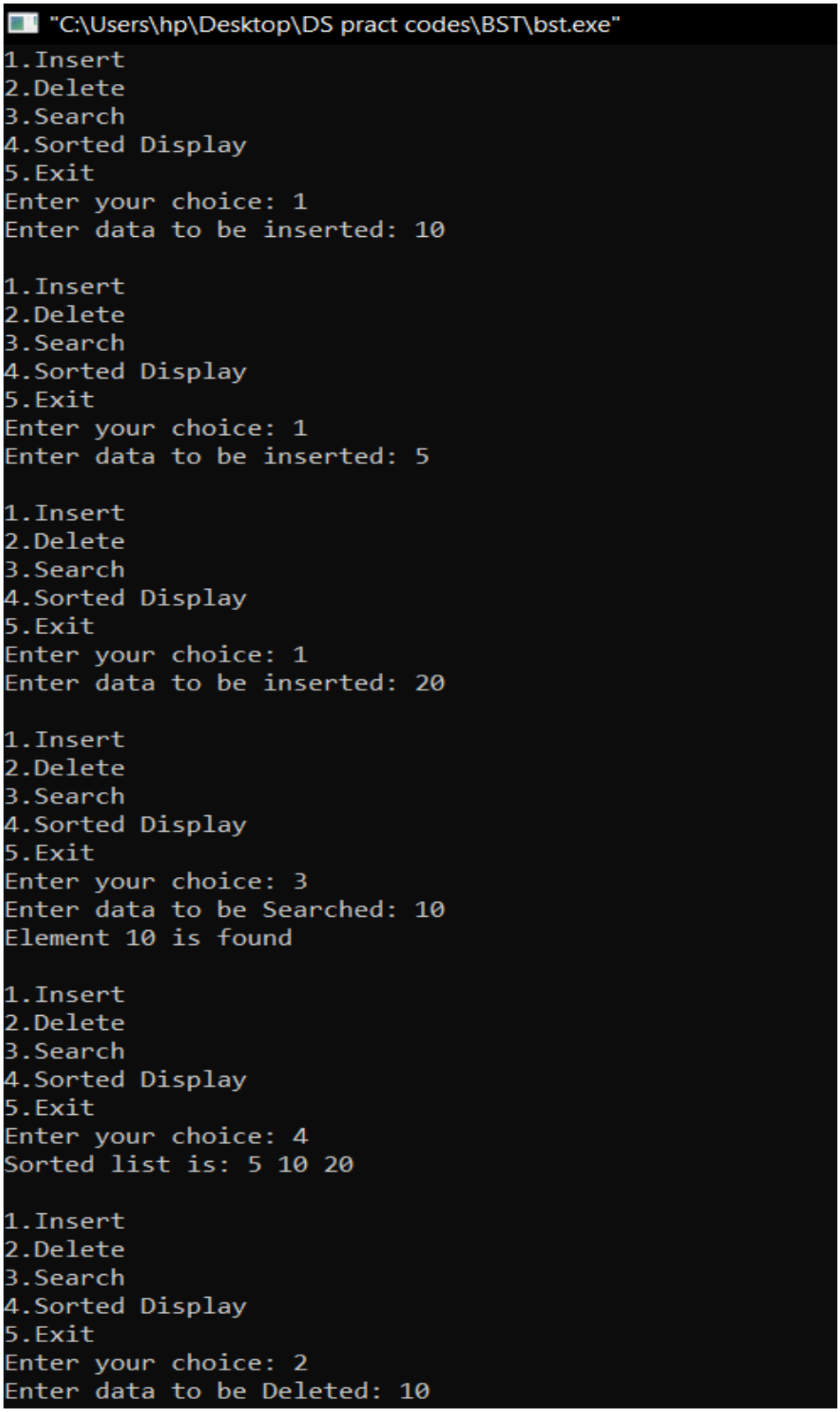
```
#include <stdio.h>
#include <stdlib.h>
struct Node
{
    struct Node *lchild;
    int data;
    struct Node *rchild;
}*root=NULL;

struct Node* newNode(int data)
{
    struct Node* temp=(struct Node *)malloc(sizeof(struct Node));
    temp->data=data;
    temp->lchild=temp->rchild=NULL;
    return temp;
}

void Inorder(struct Node *p)
{
    if(p)
    {
        Inorder(p->lchild);
        printf("%d ",p->data);
        Inorder(p->rchild);
    }
}

struct Node * Search(int key)
{
    struct Node *t=root;
    while(t!=NULL)
    {
        if(key==t->data)
            return t;
        else if(key<t->data)
            t=t->lchild;
```

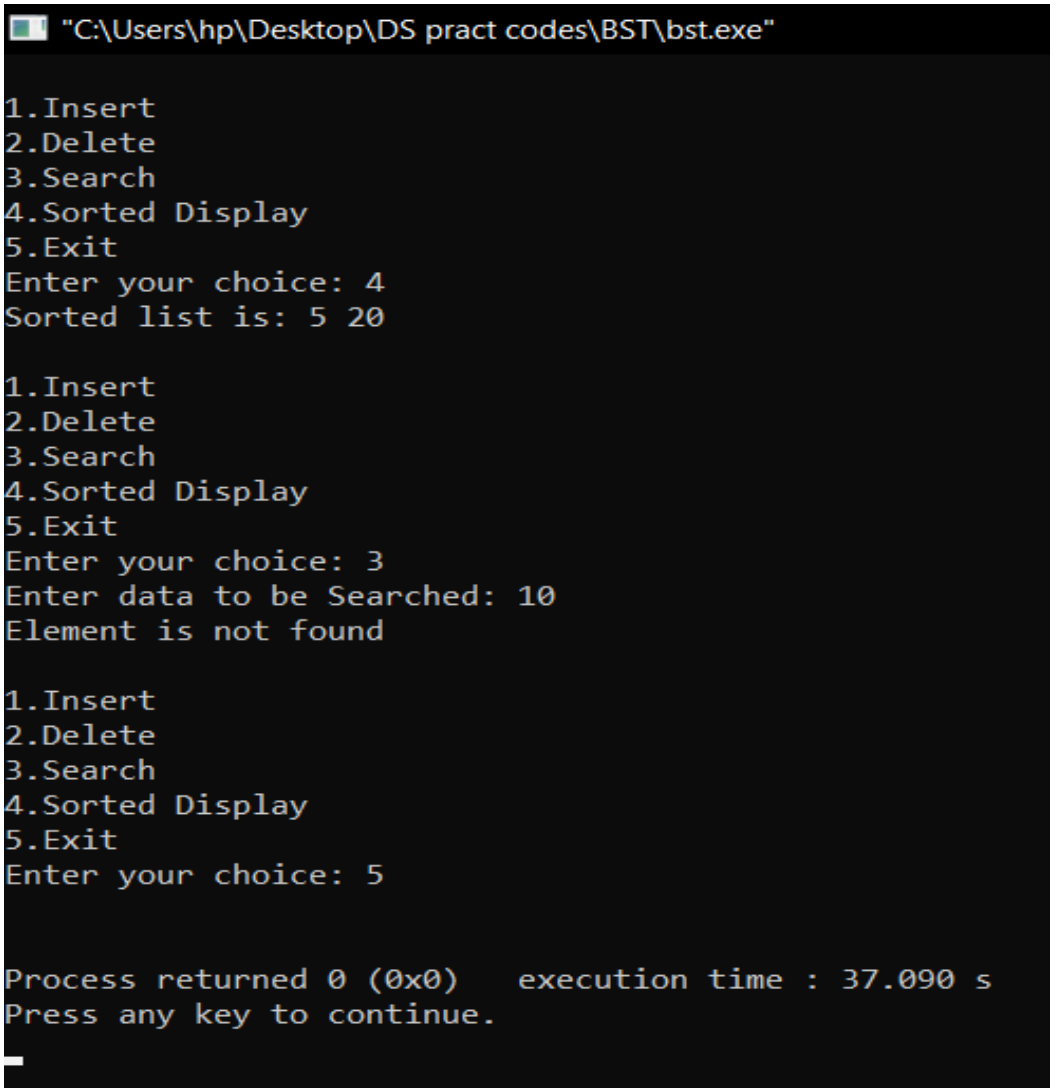
```
        else
            t=t->rchild;
    }
    return NULL;
}

struct Node *RInsert(struct Node *p,int key)
{
    struct Node *t=NULL;
    if(p==NULL)
    {
        t=newNode(key);
        return t;
    }
    if(key < p->data)
        p->lchild=RInsert(p->lchild,key);
    else if(key > p->data)
        p->rchild=RInsert(p->rchild,key);
    return p;
}

int Height(struct Node *p)
{
    int x,y;
    if(p==NULL)
        return 0;
    x=Height(p->lchild);
    y=Height(p->rchild);
        return x>y?x+1:y+1;
}

struct Node *InPre(struct Node *p)
{
    while(p && p->rchild!=NULL)
        p=p->rchild;
    return p;
}

struct Node *InSucc(struct Node *p)
{
    while(p && p->lchild!=NULL)
        p=p->lchild;
```

```
    return p;
}

struct Node *Delete(struct Node *p,int key)
    {
    struct Node *q;
    if(p==NULL)
        return NULL;
    if(p->lchild==NULL && p->rchild==NULL)
        {
        if(p==root)
        root=NULL;
        free(p);
        return NULL;
        }
    if(key < p->data)
        p->lchild=Delete(p->lchild,key);
    else if(key > p->data)
        p->rchild=Delete(p->rchild,key);
    else
        {
        if(Height(p->lchild)>Height(p->rchild))
        {
            q=InPre(p->lchild);
            p->data=q->data;
            p->lchild=Delete(p->lchild,q->data);
        }
        else
        {
            q=InSucc(p->rchild);
            p->data=q->data;
            p->rchild=Delete(p->rchild,q->data);
        }
    }
    return p;
}

int main()
{
    int c=1,choice,data;
    while(c)
    {
```

```
        printf("1. Insert\n2. Delete\n3. Search\n4. Sorted Display\n
5. Exit\nEnter your choice: ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            {
                printf("Enter data to be inserted: ");
                scanf("%d",&data);
                root=RInsert(root,data);
                break;
            }
            case 2:
            {
                printf("Enter data to be Deleted: ");
                scanf("%d",&data);
                Delete(root,data);
                break;
            }
            case 3:
            {
                printf("Enter data to be Searched: ");
                scanf("%d",&data);
                struct Node* temp=Search(data);
                if(temp!=NULL)
                    printf("Element %d is found\n",temp->data);
                else
                    printf("Element is not found\n");
                break;
            }
            case 4:
            {
                printf("Sorted list is: ");
                Inorder(root);
                printf("\n");
                break;
            }
            case 5:
            {
                c=0;
                break;
            }
```

```
            default:
            {
                printf("Invalid choice!!!!");
            }
        }
        printf("\n");
    }
    return 0;
}
```


"C:\Users\hp\Desktop\DS pract codes\BST\bst.exe"
1.Insert
2.Delete
3.Search
4.Sorted Display
5.Exit
Enter your choice: 1
Enter data to be inserted: 10
1.Insert
2.Delete
3.Search
4.Sorted Display
5.Exit
Enter your choice: 1
Enter data to be inserted: 5
1.Insert
2.Delete
3.Search
4.Sorted Display
5.Exit
Enter your choice: 1
Enter data to be inserted: 20
1.Insert
2.Delete
3.Search
4.Sorted Display
5.Exit
Enter your choice: 3
Enter data to be Searched: 10
Element 10 is found
1.Insert
2.Delete
3.Search
4.Sorted Display
5.Exit
Enter your choice: 4
Sorted list is: 5 10 20
1.Insert
2.Delete
3.Search
4.Sorted Display
5.Exit
Enter your choice: 2
Enter data to be Deleted: 10


"C:\Users\hp\Desktop\DS pract codes\BST\bst.exe"
1.Insert
2.Delete
3.Search
4.Sorted Display
5.Exit
Enter your choice: 4
Sorted list is: 5 20
1.Insert
2.Delete
3.Search
4.Sorted Display
5.Exit
Enter your choice: 3
Enter data to be Searched: 10
Element is not found
1.Insert
2.Delete
3.Search
4.Sorted Display
5.Exit
Enter your choice: 5
Process returned 0 (0x0) execution time : 37.090 s
Press any key to continue.

Binary Search

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer.

Binary search looks for a particular item by comparing the middlemost item of the collection. If a match occurs, then the index of the item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

ALGGORITHM

```
Set lowerBound = 1
Set upperBound = n

while x not found:
    if upperBound < lowerBound
        EXIT: x does not exists.

set midPoint = lowerBound + ( upperBound -
lowerBound ) / 2

if A[midPoint] < x:
    set lowerBound = midPoint + 1

if A[midPoint] > x:
    set upperBound = midPoint - 1

if A[midPoint] = x
    EXIT: x found at location midPoint
end while
```

Application of Binary Search

- We can use binary search to compute the square root of a number because we know that any given two
- Applied in Binary Search Trees (BSTs)

Binary search trees are collections that can efficiently maintain a dynamically changing dataset in sorted order.

The binary search tree is a different way of structuring data so that it can still be binary searched (or a very similar procedure can be used), but it's easier to add and remove elements. Instead of just storing the elements contiguously from least to greatest, the data is maintained in many separate chunks, making adding an element a matter of adding a new chunk of memory and linking it to existing chunks.

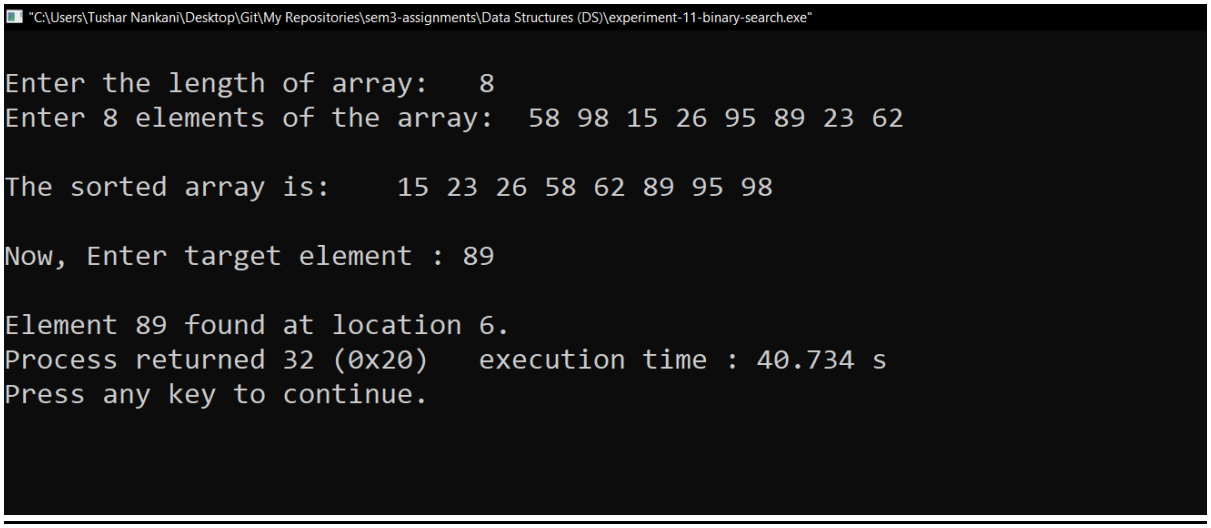
Name: **Tushar Nankani** Roll No: **1902112** Batch: **C23**

IMPLEMENTATION OF **BINARY SEARCH**

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

void binarysearch(int *a, int start, int end, int target)
{
    if(end < start)
    {
        printf("\nArray does not contain the target element.");
        return;
    }
    int mid = (start + end) / 2;
    if(a[mid]==target)
    {
        printf("\nElement %d found at location %d.", target, mid + 1
);
        return;
    }
    else if(a[mid] > target)
        //new end would be mid-1, because target in the lower half
        binarysearch(a, start, mid - 1, target);
    else
        //new start would be mid+1, because target in upper half
        binarysearch(a, mid + 1, end, target);
}

void sort(int *a, int n)
{
    for(int i = 0; i < n; i++) {
        for(int j = i + 1; j < n; j++) {
            if(a[i] > a[j]) {
                int temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
```

```
    printf("\nThe sorted array is: \t");
    for(int i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
}

void main()
{
    int length, i, target;
    printf("\nEnter the length of array:   ");
    scanf("%d", &length);

    int a[length];
    printf("Enter %d elements of the array:  ", length);
    for(i = 0; i < length; i++)
        scanf("%d", &a[i]);

    // sorting if the array is not sorted;
    sort(&a, length);

    printf("\nNow, Enter target element : ");
    scanf("%d",&target);

    // Recursive Approach;
    binarysearch(a, 0 , length, target);
}
```

OUTPUT

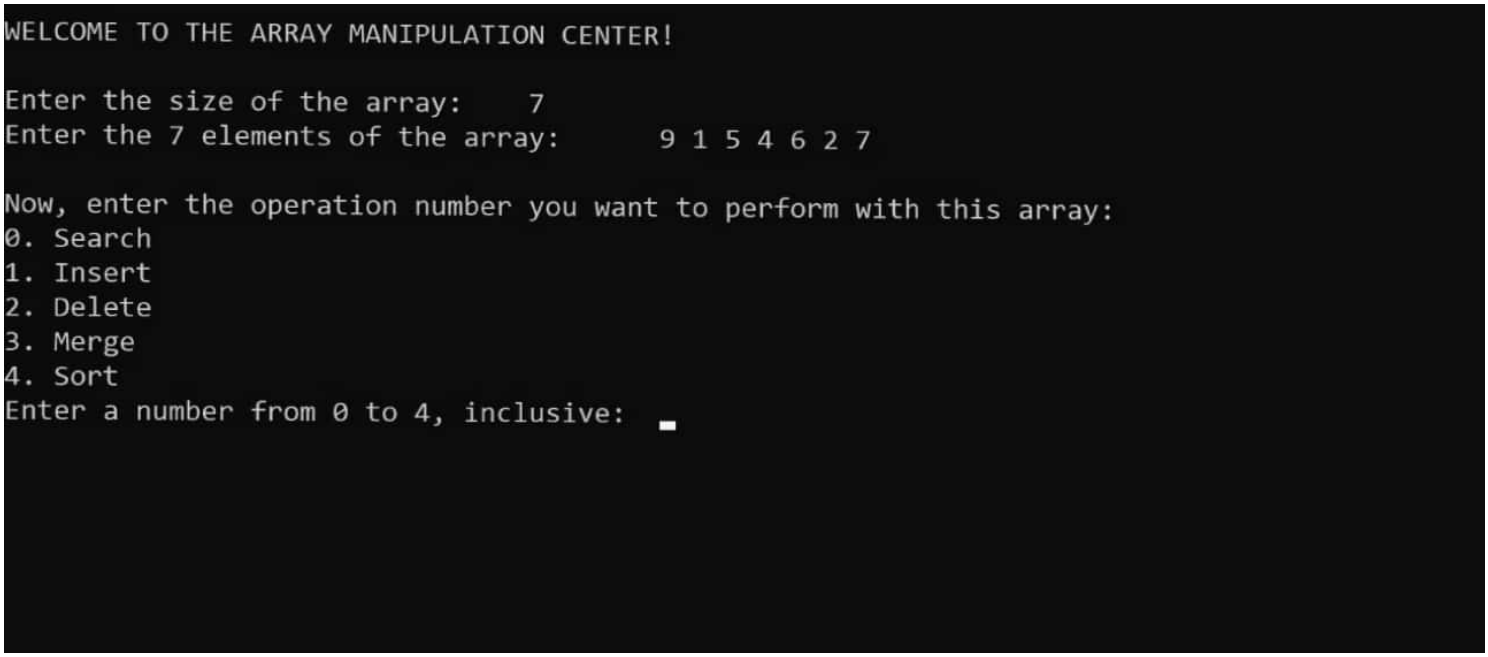

"C:\Users\Tushar Nankani\Desktop\Git\My Repositories\sem3-assignments\Data Structures (DS)\experiment-11-binary-search.exe"
Enter the length of array: 8
Enter 8 elements of the array: 58 98 15 26 95 89 23 62
The sorted array is: 15 23 26 58 62 89 95 98
Now, Enter target element : 89
Element 89 found at location 6.
Process returned 32 (0x20) execution time : 40.734 s
Press any key to continue.

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

int i, j, n, a[50], num;

void welcome()
{
    // welcome message;
    printf("\nWELCOME TO THE ARRAY MANIPULATION CENTER!\n");

    // array to be manipulated;
    printf("\nEnter the size of the array:\t");
    scanf("%d", &n);
    printf("Enter the %d elements of the array:\t", n);
    for(i = 0; i < n; i++)
        scanf("%d", &a[i]);
}

void options()
{
    // Operations;
    printf("\nNow, enter the operation number you want to perform wi
th this array: \n");
    printf("0. Search\n");
    printf("1. Insert\n");
    printf("2. Delete\n");
    printf("3. Merge\n");
    printf("4. Sort\n");
    printf("Enter a number from 0 to 4, inclusive: \t");
    scanf("%d", &num);
    printf("\n");
}

void search()
{
    int b, index = -1;
    printf("Enter the element to be found:\t");
    scanf("%d", &b);
    for(i = 0; i < n; i++)
    {
        if(a[i] == b)
        {
```

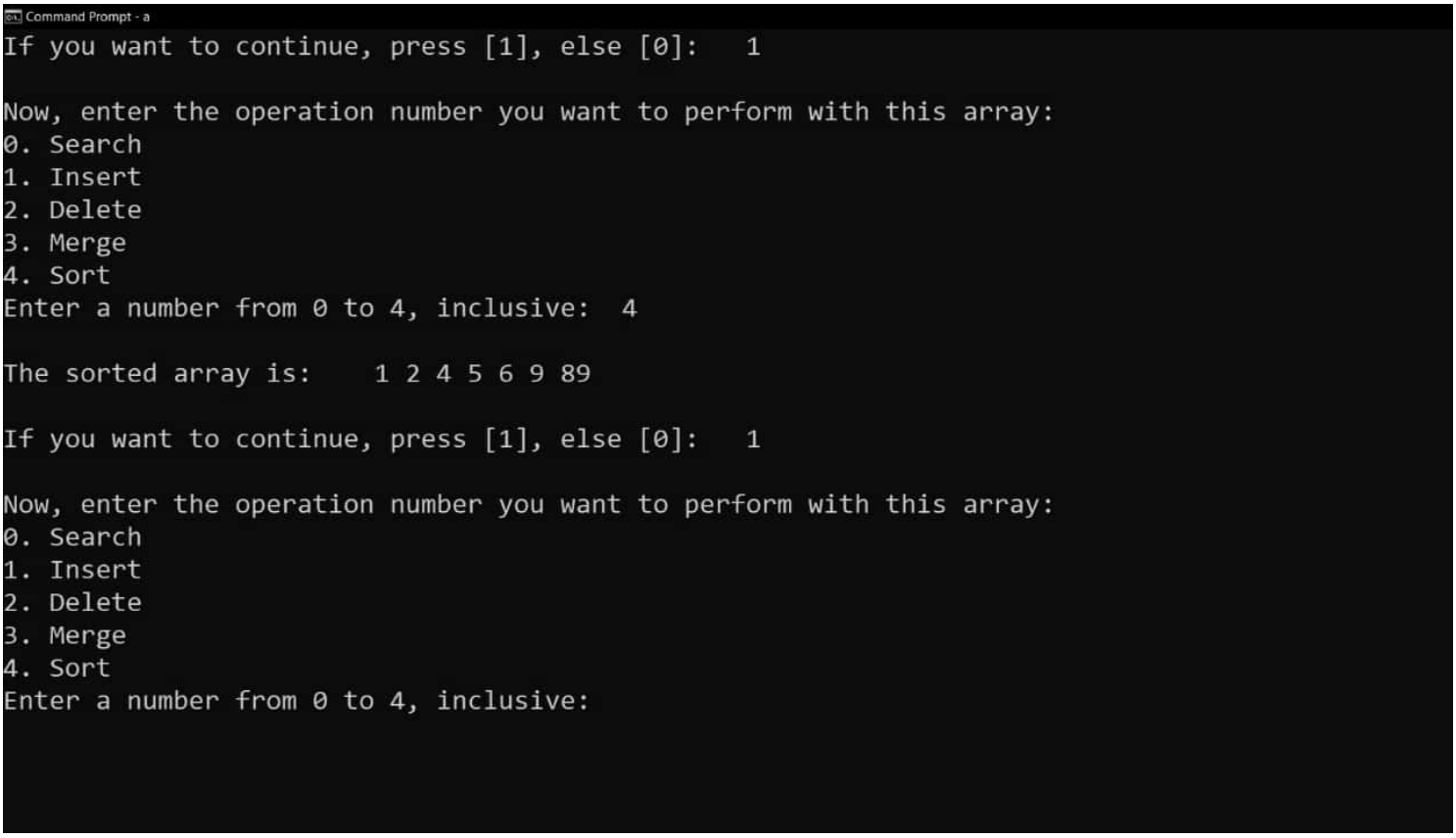
```
            index = i;
            break;
        }
    }
    if(index == -1)
        printf("Element not found!");
    else
        printf("The number %d was found at the position %d, consider
ing a 0-indexed array.\n", n, index);
}

void insert()
{
    int val, pos;
    printf("Enter the element to be inserted:\t");
    scanf("%d", &val);
    printf("\nEnter the index position (according to 0-
indexed; between 0 and %d, inclusive) to be inserted:\t", n);
    scanf("%d", &pos);
    for(i = n - 1; i >= pos; i--)
        a[i + 1] = a[i];
    a[pos] = val;
    printf("The modified array is: \t");
    // NOTE: the array size is changed to n + 1;
    for(i = 0; i < n + 1; i++)
        printf("%d ", a[i]);
    printf("\n");
}

void delete()
{
    int pos;
    printf("\nEnter the index position (according to 0-
indexed; between 0 and %d, inclusive) to be deleted:\t", n - 1);
    scanf("%d", &pos);
    for(i = pos; i < n; i++)
        a[i] = a[i + 1];
    printf("The modified array is: \t");
    // NOTE: the array size is changed to n - 1;
    for(i = 0; i < n - 1; i++)
        printf("%d ", a[i]);
    printf("\n");
```

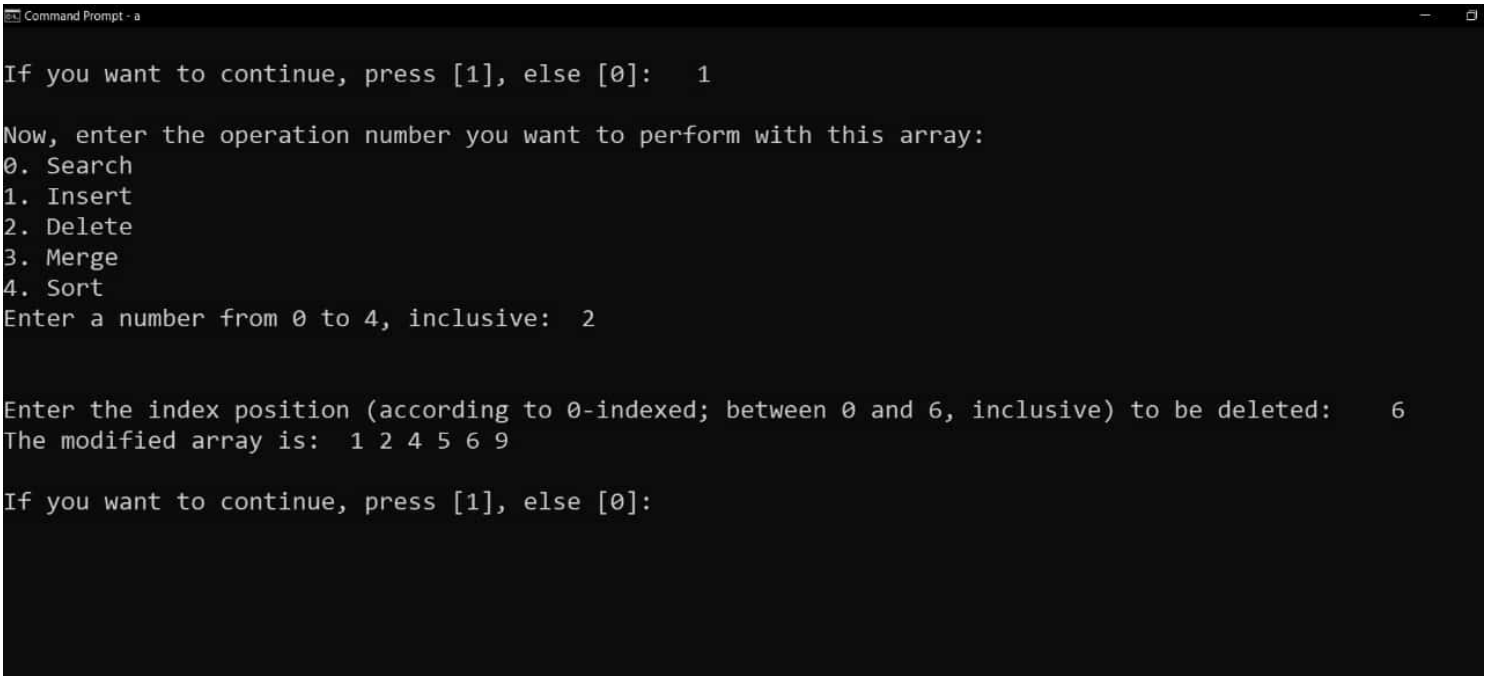
```
}

void merge()
{
    int m, b[20];
    printf("Enter the size of the array to be merged: \t");
    scanf("%d", &m);
    printf("Enter the %d elements of the array to be merged:\t", m);
    for(i = 0; i < m; i++)
        scanf("%d", &b[i]);
    for(i = n; i < n + m; i++)
        a[i] = b[i - n];

    printf("The size of the merged array is %d.\n", n + m);
    for(i = 0; i < n + m; i++)
        printf("%d ", a[i]);
    printf("\n");
}

void sort()
{
    for(i = 0; i < n; i++) {
        for(j = i + 1; j < n; j++) {
            if(a[i] > a[j]) {
                int temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
    printf("The sorted array is: \t");
    for(i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
}

int cont()
{
    int t;
    printf("\nIf you want to continue, press [1], else [0]:\t");
    scanf("%d", &t);
    return t;
```

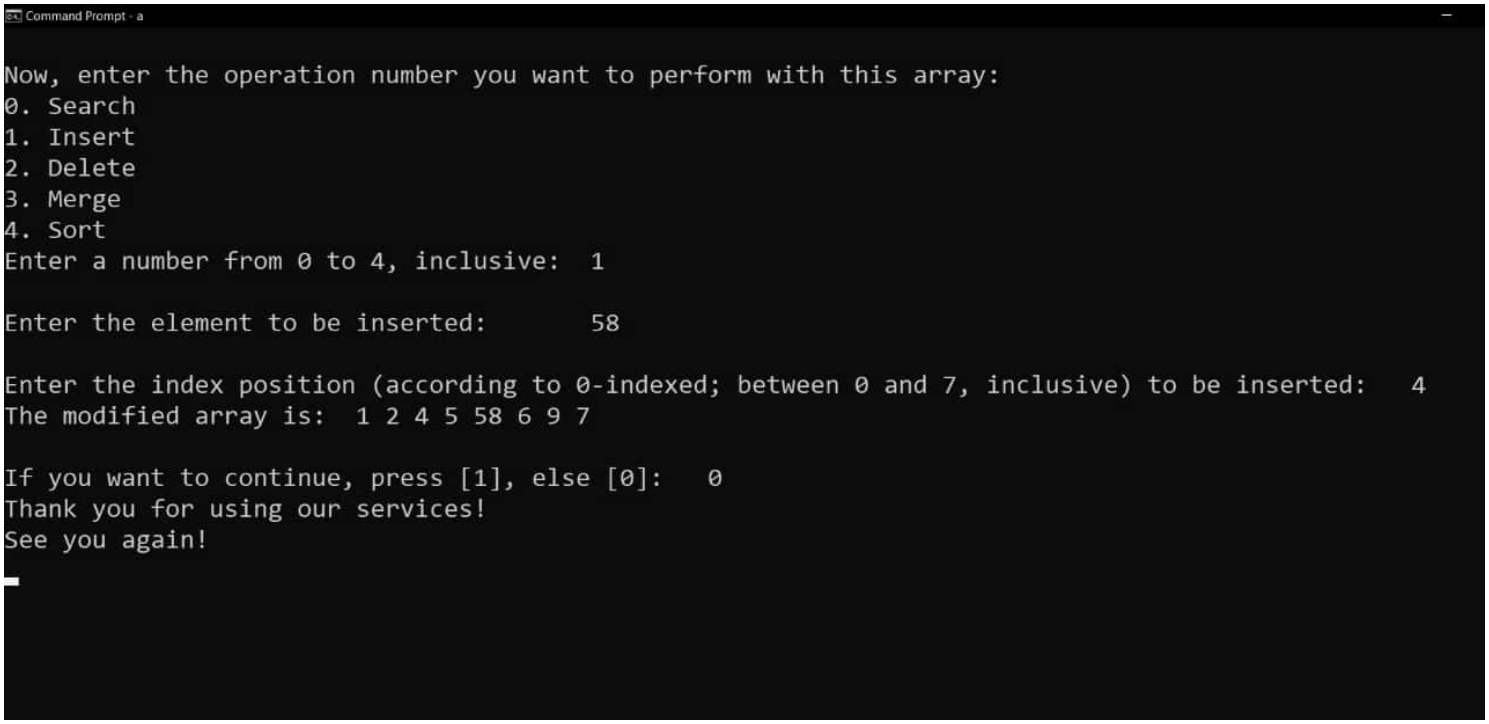
```
}

void end()
{
    printf("\nThank you for using our services!\nSee you again!\n");
}

void main()
{
    welcome();

    int t = 1;
    // loop until the user stops;
    while(t)
    {
        options();
        switch (num)
        {
            case 0:
                search();
                break;
            case 1:
                insert();
                break;
            case 2:
                delete();
                break;
            case 3:
                merge();
                break;
            case 4:
                sort();
                break;

            default:
                printf("Option not valid!");
        }
        t = cont();
    }
    end();
    getch();
}
```



```
WELCOME TO THE ARRAY MANIPULATION CENTER!

Enter the size of the array:    7
Enter the 7 elements of the array:      9 1 5 4 6 2 7

Now, enter the operation number you want to perform with this array:
0. Search
1. Insert
2. Delete
3. Merge
4. Sort
Enter a number from 0 to 4, inclusive:  1

Enter the element to be inserted:       89

Enter the index position (according to 0-indexed; between 0 and 7, inclusive) to be inserted:   5
The modified array is:  9 1 5 4 6 89 2 7

If you want to continue, press [1], else [0]:
```







Q2. Compare Arrays and Linked List with examples.

Ans: ARRAYS:

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).

The base value is index 0 and the difference between the two indexes is the offset.

EXAMPLE:

array - A = [1, 2, 3, 4, 5]

indexing : 0 1 2 3 4

Advantages of using arrays:

Arrays allow random access of elements. This makes accessing elements by position faster.

Arrays have better cache locality that can make a pretty big difference in performance.

(contd..)

Disadvantages of using arrays:

You can't change the size i.e. once you have declared the array you can't change its size because of static memory allocated to it.

LINKED-LIST:

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers.

HEAD

[DATA][NEXT] ---> [DATA][NEXT] --> [DATA][X]

- NEXT represents the next memory location.
- X represents end of a Linked List.

Advantages of using Linked List:

- Manipulation with LinkedList is faster than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
- LinkedList is better for manipulating data.

Disadvantages of using Linked List:

- Pointer Requires extra memory for storage.
- Time consuming
- More time Complexity

Q3. Difference Between Linear Saerch and Binary Search.

A linear search scans one item at a time, without jumping to any item .
- The worst case complexity is O(n), sometimes known an O(n) search
- Time taken to search elements keep increasing as the number of elements are increased.

A binary search however, cut down your search to half as soon as you find middle of a sorted list.
- The middle element is looked to check if it is greater than or less than the value to be searched.
- Accordingly, search is done to either half of the given list

DIFFERENCES:
- Input data needs to be sorted in Binary Search and not in Linear Search
- Linear search does the sequential access whereas Binary search access data randomly.
- Time complexity of linear search -O(n) , Binary search has time complexity O(log n).
- Linear search performs equality comparisons and Binary search performs ordering comparisons

Q4. Explain different types of recursion.

Any recursive function can be characterized based on:
- whether the function calls itself directly or indirectly (direct or indirect recursion).
- whether any operation is pending at each recursive call (tail-recursive or not).
- the structure of the calling pattern (linear or tree-recursive/Excessive recursion)

1. Direct Recursion
A function is said to be directly recursive if it explicitly calls itself.

```
long factorial(int n)
{
  if(x == 0)
      return 1;
  else
      return x * factorial(x - 1);
}
```

- Tail recursive functions are highly desirable because they are much more efficient to use as in their case, the amount of information that has to be stored on the system stack is independent of the number of recursive calls.

```
void fun(int n)
{
  if(n==0)
     return;
  else
     printf(" %d ",n);
     return fun(n-1);
}

int main()
{
  fun(3);
  return 0;
}
```

INDIRECT RECURSION

A function is said to be indirectly recursive if it contains a call to another function which ultimately calls it.

```
int func1(int n)
{
   if(n == 0)
      return n;
    return func2(n);
}

int func2(int x)
{
    return func1(x - 1);
}
```

3. TAIL RECURSION

A recursive function is said to be tail recursive if no operations are pending to be performed when the recursive function returns to its caller, when the called function returns, the returned value is immediately returned from the calling function.

4. Linear recursion:

A linear recursive function is a function that only makes a single call to itself each time the function runs (as opposed to one that would call itself multiple times during its execution). The factorial function is a good example of linear recursion.

For example, the factorial function is linearly recursive as the pending operation involves only multiplication to be performed and does not involve another call to fact.

5. Tree Recursion:

Recursive functions can also be characterized depending on the way in which the recursion grows- in a linear fashion or forming a tree structure.

On the contrary, a recursive function is said to be tree recursive (or non-linearly recursive) if the pending operation makes another recursive call to the function.

For example, the Fibonacci function Fib in which the pending operations recursively calls the Fib function.

Example of tree recursion:

```
int Fibonacci(int num)
{
   if(num ==0 || num == 1)
       return num;
return ( Fibonacci (num - 1) + Fibonacci(num - 2));
}
```

DS - ASSIGNMENT NO 2

Explain in detail with example :

1. B tree
2. B+ tree

1. B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in main memory.

To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in the main memory. When the number of keys is high, the data is read from the disk in the form of blocks. Disk access time is very high compared to the main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree.

2. B+ tree:

A B+ Tree is primarily utilized for implementing dynamic indexing on multiple levels. Compared to the B- Tree, the B+ Tree stores the data pointers only at the leaf nodes of the Tree, which makes the search more process more accurate and faster.
Leaves are used to store data records.
It stored in the internal nodes of the Tree.
If a target key value is less than the internal node, then the point just to its left side is followed.

If a target key value is greater than or equal to the internal node, then the point just to its right side is followed. The root has a minimum of two children.

3. Explain the different types of queues along with an example.

Queue: Queue is a linear data structure in which the removal of elements is done in the same order they were inserted i.e., the element will be removed first which is inserted first.

A queue has two ends which are the front end and rear end. Insertion takes place at the rear end and deletion is taking place at the front end. The queue is also known as the first in first out (FIFO) data structure. The queue can be classified into the following types:

- Circular Queue:
  - In circular queues last element next address shows first element address (front)
  - In circular queues first element previous address always shows last element address (rear).

• Priority Queue

In priority queues, while inserting an element we assign priority to that element.

- While deleting we check two conditions.

1. Highest priority element deleted first.
2. While two elements have the same priority the element entered first into the queue is deleted first then next.

Application - Patient Priority queue in a hospital, Task Scheduling in computers.

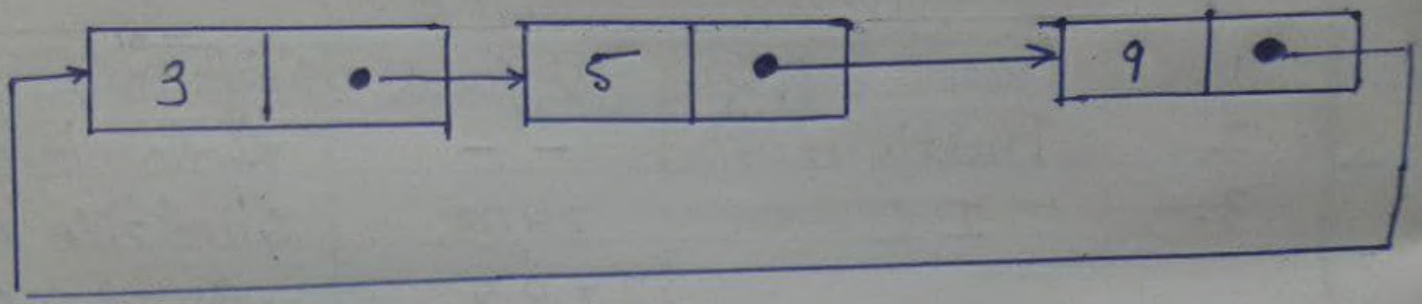
4. Describe the applications of graphs.

- In Computer science graphs are used to represent the flow of computation.
- Google maps use graphs for building transportation systems, where the intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
- In Facebook, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of an undirected graph.
- In World Wide Web, web pages are considered to be the vertices. There is an edge from page u to other page v if there is a link of page v on page u. This is an example of Directed graph. It was the basic idea behind Google Page Ranking Algorithm.

- In Operating System, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur.

5. Write a short note on circular singly LinkedList.

In a singly linked list, for accessing any node of linked list, we start traversing from the first node. If we are at any node in the middle of the list, then it is not possible to access nodes that precede the given node. This problem can be solved by slightly altering the structure of a singly linked list. In a singly linked list, the next part (pointer to next node) is NULL, if we utilize this link to point to the first node then we can reach the preceding nodes.To implement a circular singly linked list, we take an external pointer that points to the last node of the list. If we have a pointer last pointing to the last node, then last -> next will point to the first node.



Circular singly Linkedlist

Mini Project-Binary Tree Traversal

Code:

```
#include<stdio.h>
#include<stdlib.h>
//queue functions
/*
            1
      2           3
4           5 6         7
*/
struct node
{
      struct node* next;
      struct t_node* data;
};

struct node* new_qNode(struct t_node* x)
{
      struct node* temp=(struct node*)malloc(sizeof(struct node));
      temp->next=NULL;
      temp->data=x;
      return temp;
}
//queue functions



//trees function
```

```
struct t_node
{
	struct t_node* left;
	struct t_node* right;
	int data;
};

struct t_node* new_tNode(int x)
{
	struct t_node* temp=(struct t_node*)malloc(sizeof(struct t_node));
	temp->left=temp->right=NULL;
	temp->data=x;
	return temp;
}

void Inorder(struct t_node* root)
{//LNR
	if(!root)
		return;
	Inorder(root->left);
	printf("%d ",root->data);
	Inorder((root->right));
}

void Preorder(struct t_node* root)
{
	//NLR
	if(!root)
```

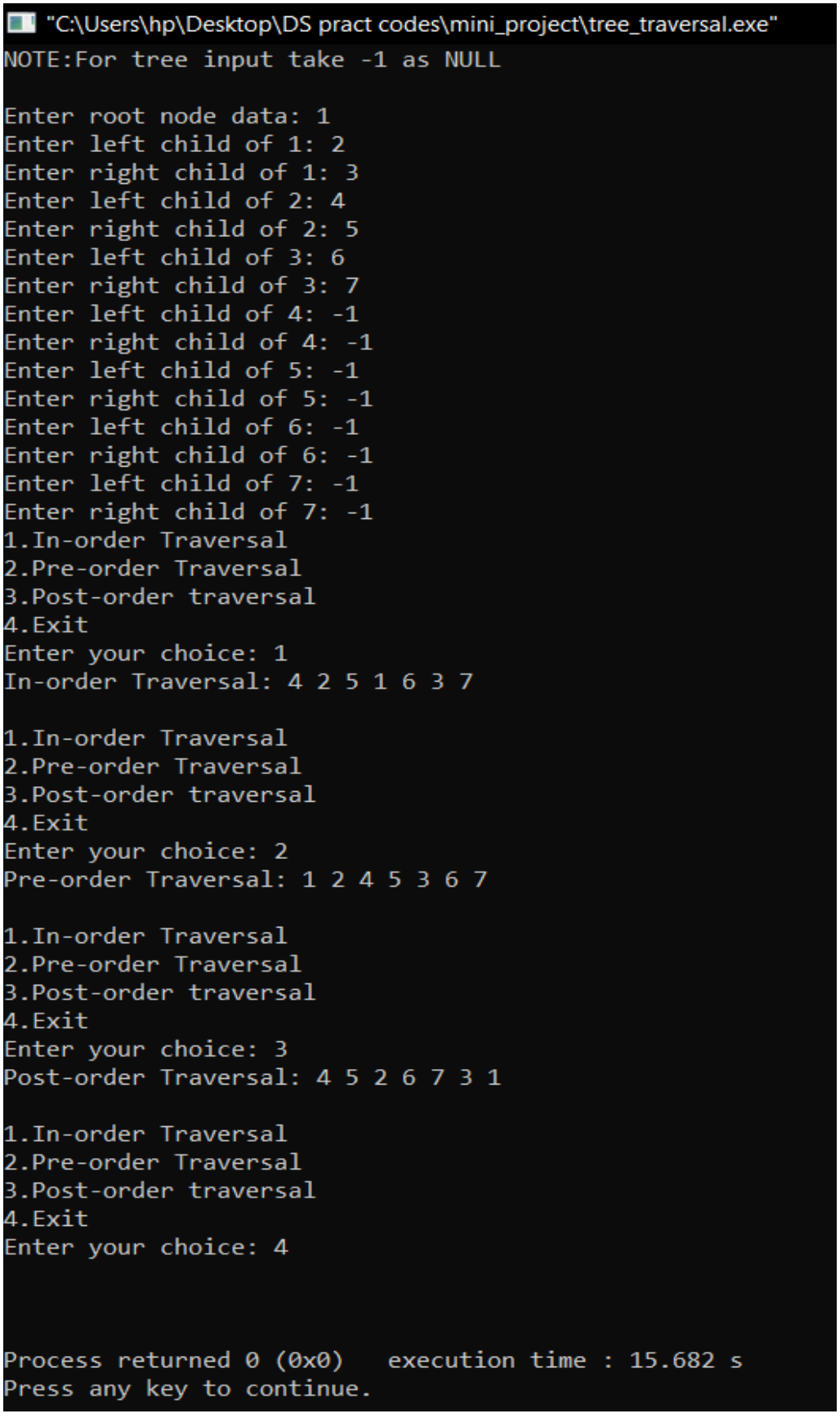
```
            return;
        printf("%d ",root->data);
        Preorder(root->left);
        Preorder(root->right);
}

void Postorder(struct t_node* root)
{
        if(!root)
            return;
        Postorder(root->left);
        Postorder((root->right));
        printf("%d ",root->data);
}



/*
1.          1
2.    2           3
3.4         5
Inorder:4 2 5 1 3
*/
//trees function
int main()
{
        printf("NOTE:For tree input take -1 as NULL\n\n");
        struct t_node* root=NULL;
        int x,lc,rc;
```

```
printf("Enter root node data: ");
scanf("%d",&x);
root=new_tNode(x);
//queue
struct node* head=new_qNode(root);
struct node* tail=head;//rear pointer
while(head)
{
	struct t_node* temp=head->data;
	printf("Enter left child of %d: ",temp->data);
	scanf("%d",&lc);
	if(lc!=-1)
	{
		//tree Linking Nodes
		temp->left=new_tNode(lc);
		//tree
		//queue
		tail->next=new_qNode(temp->left);
		tail=tail->next;
		//queue
	}
	printf("Enter right child of %d: ",temp->data);
	scanf("%d",&rc);
	if(rc!=-1)
	{
		//tree linking
		temp->right=new_tNode(rc);
		//tree
```

```
            //queue
            tail->next=new_qNode(temp->right);
            tail=tail->next;
            //queue
        }
        struct node* t=head;
        head=head->next;
        free(t);
    }
    int c=1,choice;
    while(c)
    {
        printf("1.In-order Traversal\n2.Pre-order Traversal\n3.Post-order
traversal\n4.Exit\nEnter your choice: ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            {
                printf("In-order Traversal: ");
                Inorder(root);
                break;
            }
            case 2:
            {
                printf("Pre-order Traversal: ");
                Preorder(root);
                break;
```

```
                }
                case 3:
                {
                        printf("Post-order Traversal: ");
                        Postorder(root);
                        break;
                }
                case 4:
                {
                        c=0;
                        break;
                }
                default:
                {
                        printf("Invalid choice!!!!Try again!!!!\n");
                }
            }
            printf("\n");
        }
        return 0;
}
```

Output:


"C:\Users\hp\Desktop\DS pract codes\mini_project\tree_traversal.exe"
NOTE:For tree input take -1 as NULL
Enter root node data: 1
Enter left child of 1: 2
Enter right child of 1: 3
Enter left child of 2: 4
Enter right child of 2: 5
Enter left child of 3: 6
Enter right child of 3: 7
Enter left child of 4: -1
Enter right child of 4: -1
Enter left child of 5: -1
Enter right child of 5: -1
Enter left child of 6: -1
Enter right child of 6: -1
Enter left child of 7: -1
Enter right child of 7: -1
1.In-order Traversal
2.Pre-order Traversal
3.Post-order traversal
4.Exit
Enter your choice: 1
In-order Traversal: 4 2 5 1 6 3 7
1.In-order Traversal
2.Pre-order Traversal
3.Post-order traversal
4.Exit
Enter your choice: 2
Pre-order Traversal: 1 2 4 5 3 6 7
1.In-order Traversal
2.Pre-order Traversal
3.Post-order traversal
4.Exit
Enter your choice: 3
Post-order Traversal: 4 5 2 6 7 3 1
1.In-order Traversal
2.Pre-order Traversal
3.Post-order traversal
4.Exit
Enter your choice: 4
Process returned 0 (0x0) execution time : 15.682 s
Press any key to continue.