**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

**“JnanaSangama”, Belgaum -590014, Karnataka.**

****

**LAB REPORT**

**On**

**DATA STRUCTURES (23CS3PCDST)**

**Submitted by**

**NAME (USN)**

**in partial fulfillment for the award of the degree of**

**BACHELOR OF ENGINEERING**

**in**

**COMPUTER SCIENCE AND ENGINEERING**



**B.M.S. COLLEGE OF ENGINEERING**

**(Autonomous Institution under VTU)**

**BENGALURU-560019**

**September 2024-January 2025**

**B. M. S. College of Engineering,**

**Bull Temple Road, Bangalore 560019**

**(Affiliated To Visvesvaraya Technological University, Belgaum)**

**Department of Computer Science and Engineering**

****

This is to certify that the Lab work entitled **“DATA STRUCTURES”** carried out by NAME **(USN)**, who is bonafide student of **B. M. S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2024-25. The Lab report has been approved as it satisfies the academic requirements in respect of Data structures Lab - **(23CS3PCDST)**work prescribed for the said degree.

**Prof. Namratha M** **Dr. Kavitha Sooda**

Assistant Professor Professor and Head

Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengaluru

**Index Sheet**

|  |  |  |
| --- | --- | --- |
| **Sl.**  **No.** | **Experiment Title** | **Page No.** |
| 1 | **WORKING OF STACK USING AN ARRAY** | 4 |
| 2 | **INFIX TO POSTFIX** | 6 |
| 3 | **PROGRAM TO IMPLEMENT LINEAR QUEUE:** | 9 |
| 4 | **CIRCULAR QUEUE** | 13 |
| 5 | **FACTORIAL** | 16 |
| 6 | **FIBONACCI** | 17 |
| 7 | **TOWER OF HANOI** | 18 |
| 8 | **CREATE LINKED LIST, INSERT ELEMENTS, DISPLAY LINKED LIST** | 20 |
| 9 | **SORTING A LINKED LIST ,REVERSING AND CONCATENATE** | 26 |
| 10 | **STACK AND QUEUE USING LINKED LIST** | 30 |
| 11 | **PROGRAM TO IMPLEMENT DOUBLY LINKED LIST INSERTION AND DELETION** | 34 |
| 12 | **BINARY SEARCH TREE** | 44 |
| 13 | **DFS TRAVERSAL** | 48 |
| 14 | **BFS TRAVERSAL** | 50 |
| 15 | **HASHING (LINEAR PROBING)** | 53 |

**Lab program 1:**

**Write a program to simulate the working of stack using an array with the following:**

**a) Push**

**b) Pop**

**c) Display**

**The program should print appropriate messages for stack overflow, stack underflow.**

#include <stdio.h>

#include<stdlib.h>

#define STACK\_SIZE 5

void push(int st[],int \*top)

{

int item;

if(\*top==STACK\_SIZE-1)

printf("Stack overflow\n");

else

{

printf("\nEnter an item :");

scanf("%d",&item);

(\*top)++;

st[\*top]=item;

}

}

void pop(int st[],int \*top)

{

if(\*top==-1)

printf("Stack underflow\n");

else

{

printf("\n%d item was deleted",st[(\*top)--]);

}

}

void display(int st[],int \*top)

{

int i;

if(\*top==-1)

printf("Stack is empty\n");

for(i=0;i<=\*top;i++)

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

}

void main()

{

int st[10],top=-1, c,val\_del;

while(1)

{

printf("\n1. Push\n2. Pop\n3. Display\n");

printf("\nEnter your choice :");

scanf("%d",&c);

switch(c)

{

case 1: push(st,&top);

break;

case 2: pop(st,&top);

break;

case 3: display(st,&top);

break;

default: printf("\nInvalid choice!!!");

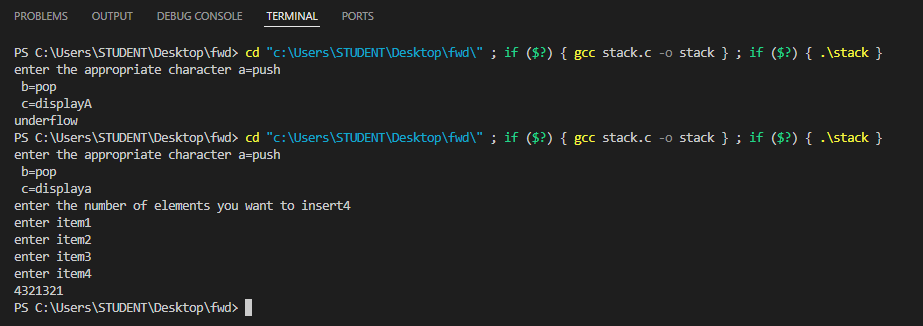
exit(0);

}

}

}

**Output:**



**WEEK 2: INFIX TO POSTFIX Z**:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX 100

// Stack implementation

char stack[MAX];

int top = -1;

// Function to push an element to the stack

void push(char x) {

    if (top == MAX - 1) {

        printf("Stack overflow\n");

        return;

    }

    stack[++top] = x;

}

// Function to pop an element from the stack

char pop() {

    if (top == -1) {

        printf("Stack underflow\n");

        return -1;

    }

    return stack[top--];

}

// Function to get precedence of operators

int precedence(char op) {

    if (op == '+' || op == '-')

        return 1;

    if (op == '\*' || op == '/')

        return 2;

    return 0;

}

// Function to convert infix expression to postfix expression

void infixToPostfix(char\* infix, char\* postfix) {

    int i = 0, j = 0;

    char ch;

    while (infix[i] != '\0') {

        ch = infix[i];

  // If the character is an operand, add it to the postfix expression

        if (isalnum(ch)) {

            postfix[j++] = ch;

        }

        // If the character is '(', push it to the stack

        else if (ch == '(') {

            push(ch);

        }

        // If the character is ')', pop and add to postfix until '(' is found

        else if (ch == ')') {

            while (top != -1 && stack[top] != '(') {

                postfix[j++] = pop();

            }

            pop(); // Remove '(' from stack

        }

        // If an operator is encountered

        else {

            while (top != -1 && precedence(stack[top]) >= precedence(ch)) {

                postfix[j++] = pop();

            }

            push(ch);

        }

        i++;

    }

    // Pop all remaining operators from the stack

    while (top != -1) {

        postfix[j++] = pop();

    }

    postfix[j] = '\0'; // Null-terminate the postfix expression

}

int main() {

    char infix[MAX], postfix[MAX];

    printf("Enter infix expression: ");

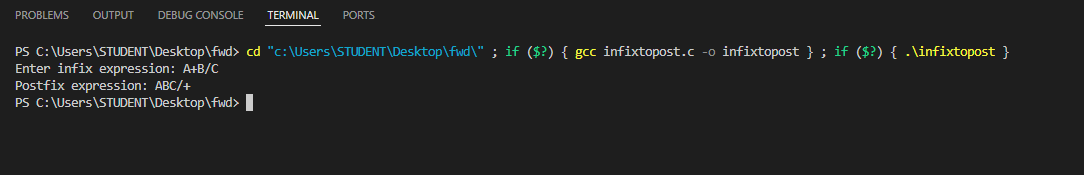
    scanf("%s", infix);

    infixToPostfix(infix, postfix);

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

    return 0;

}

 OUTPUT:

//**PROGRAM TO IMPLEMENT LINEAR QUEUE:**

**WEEK 3**

#include<stdio.h>

#include<stdlib.h>

#define max 300

int a[max];

int front = -1, rear = -1;

int isFull();

int isEmpty();

void enque(int x);

void deque();

void display();

int isFull() {

if (rear == max - 1) {

return 1;

}

return 0;

}

int isEmpty() {

if (front == -1 && rear == -1) {

return 1;

}

return 0;

}

void enque(int x) {

if (isFull()) {

printf("Queue is full\n");

} else if (isEmpty()) {

front = 0;

rear = 0;

a[rear] = x;

} else {

rear++;

a[rear] = x;

}

}

void deque() {

if (isEmpty()) {

printf("Queue is empty\n");

} else {

printf("Dequeued: %d\n", a[front]);

front++;

if (front > rear) {

front = rear = -1;

}

}

}

void display() {

printf("Shreya J G 1BM23IC061 \n");

if (isEmpty()) {

printf("Queue is empty\n");

return;

}

printf("The queue elements are: ");

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

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

}

printf("\n");

}

int main() {

int ch, n;

while (1) {

printf("Enter 1.Enqueue 2.Dequeue 3.Display 4.Exit\n");

scanf("%d", &ch);

switch (ch) {

case 1:

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

scanf("%d", &n);

enque(n);

break;

case 2:

deque();

break;

case 3:

display();

break;

case 4:

exit(0);

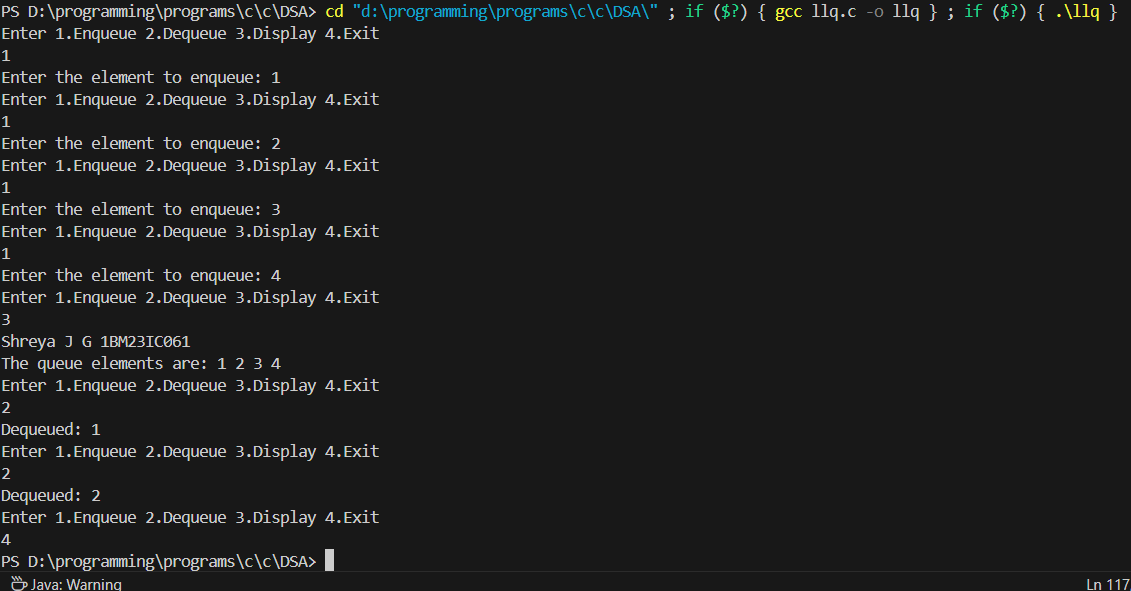
default:

printf("Wrong choice entered!\n");

}

}

}

OUTPUT:

**WEEK 4**

**// CIRCULAR QUEUE:**

#include <stdio.h>

#include <stdlib.h>

#define N 5

int cq[N];

int front = -1, rear = -1;

void enque(int x);

void deque();

void display();

void enque(int x) {

if ((rear + 1) % N == front) {

printf("Queue is full\n");

return;

}

if (front == -1 && rear == -1) {

front = 0;

rear = 0;

cq[rear] = x;

} else {

rear = (rear + 1) % N;

cq[rear] = x;

}

}

void deque() {

if (front == -1 && rear == -1) {

printf("Queue is empty\n");

return;

} else if (front == rear) {

printf("Dequeued: %d\n", cq[front]);

front = -1;

rear = -1;

} else {

printf("Dequeued: %d\n", cq[front]);

front = (front + 1) % N;

}

}

void display() {

printf("Shreya J G 1B23IC061 \n");

if (front == -1 && rear == -1) {

printf("Empty queue\n");

return;

} else {

printf("Queue is: \n");

int i = front;

while (i != rear) {

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

i = (i + 1) % N;

}

printf("%d\n", cq[rear]);

}

}

int main() {

enque(44);

enque(2);

enque(3);

enque(1);

enque(4);

deque();

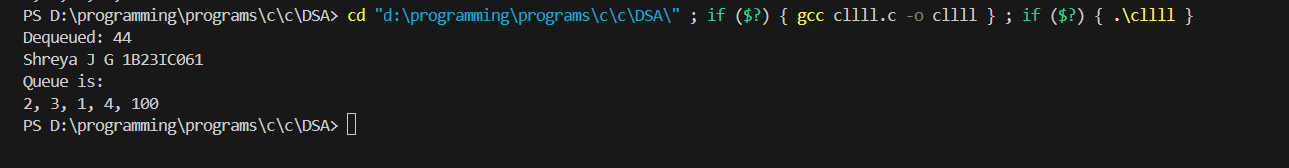
enque(100);

display();

return 0;

}

**OUTPUT:**

****

**WEEK 5**

**//factorial**

#include<stdio.h>

#include<stdlib.h>

int fact(int num);

void main()

{

int num,c;

printf("enter the number");

scanf("%d",&num);

c=fact(num);

printf("%d",c);

}

int fact(int n)

{

int f=1,m;

if(n>0)

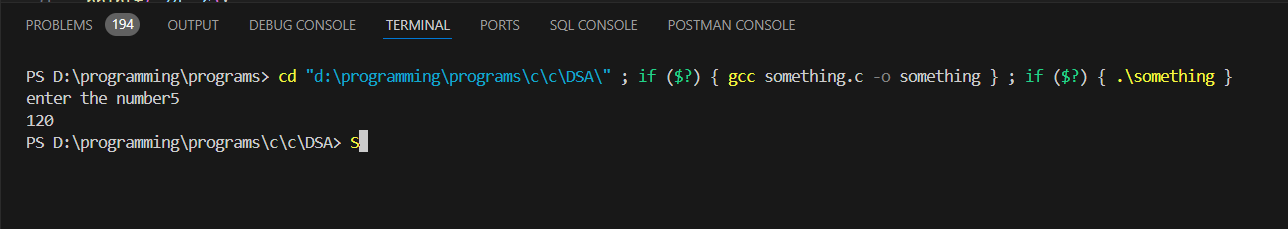
{

f=n\*fact(n-1);

}

return f;

}

****OUTPUT:

**//FIBONACCI SERIES:**

#include<stdio.h>

#include<stdlib.h>

int fibonacci(int n);

void main()

{

int n,c,i;

printf("enter the number of fibonacci series");

scanf("%d",&n);

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

{

printf("%d",fibonacci(i));

}

}

int fibonacci(int n)

{

if(n==0){

return 0;

}

else if(n==1){

return 1;

}

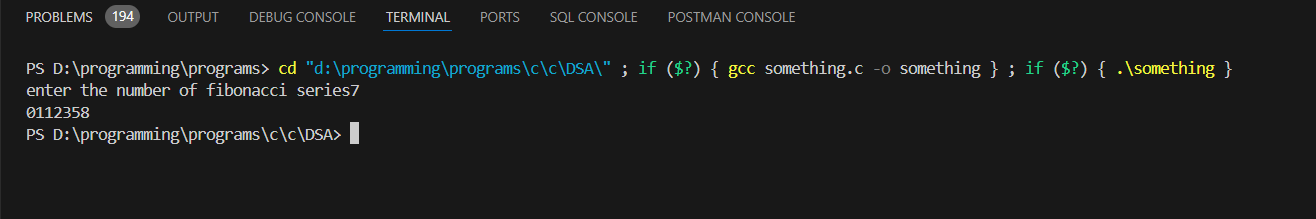
else{

return fibonacci(n-1)+fibonacci(n-2);

}

}

OUTPUT:

****

**3.Tower of Hanoi:**

#include <stdio.h>

void hanoi(int n, char from, char to, char via) {

if(n == 1){

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

}

else{

hanoi(n-1, from, via, to);

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

hanoi(n-1, via, to, from);

}

}

int main() {

int n = 3;

char from = 'A';

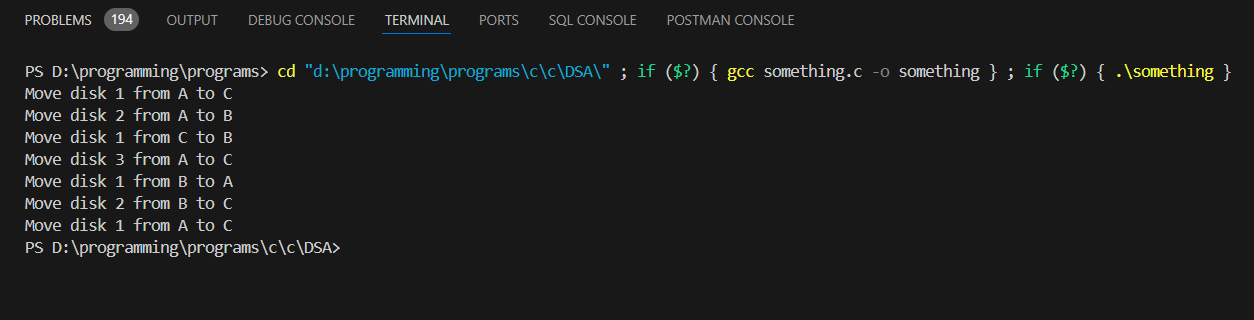
char to = 'B';

char via = 'C';

hanoi(n, from, via, to);

}

OUTPUT:

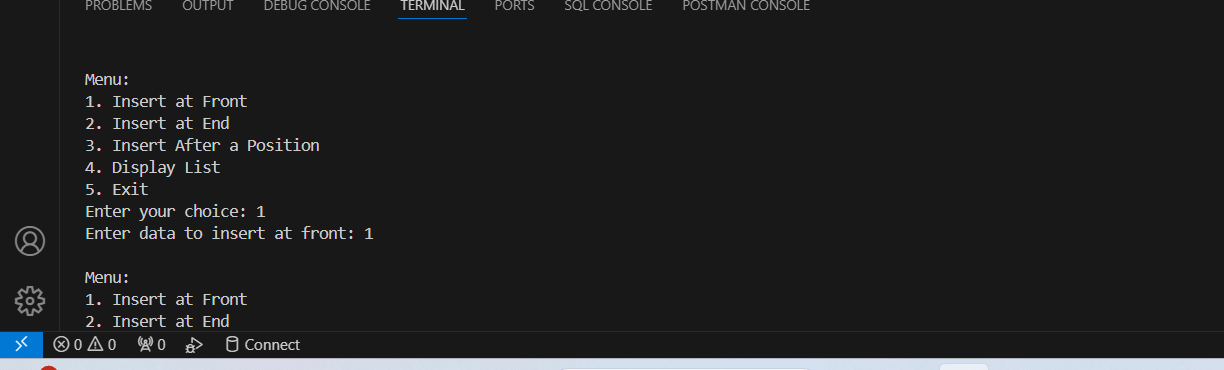


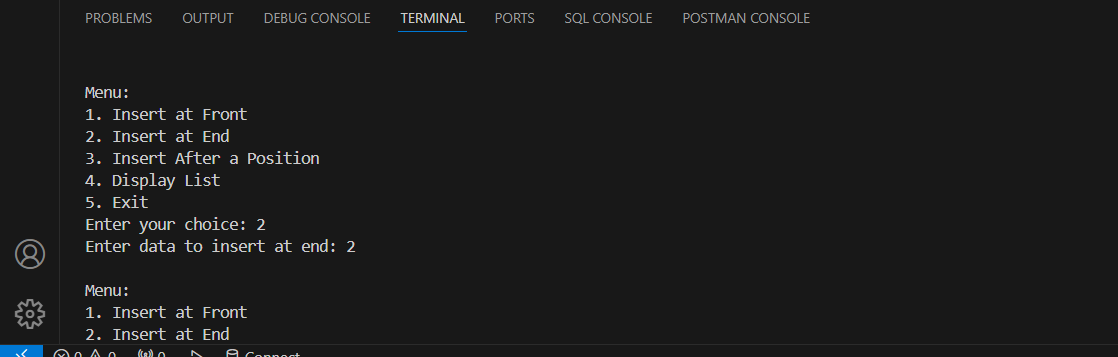
**WEEK 6**

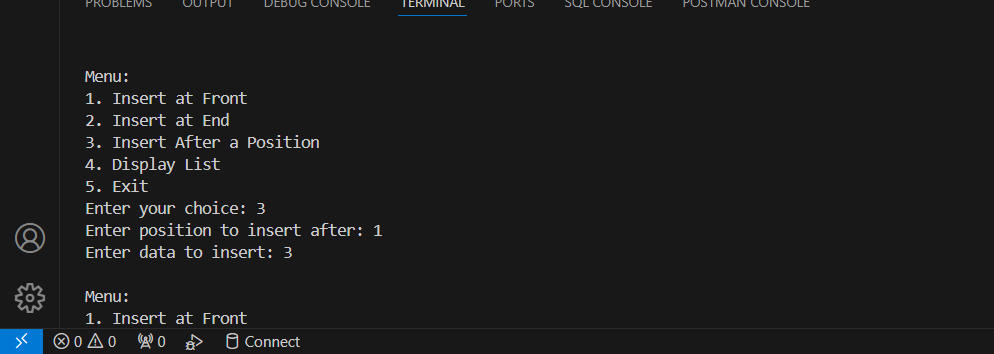
**CREATE LINKED LIST, INSERT ELEMENTS, DISPLAY LINKED LIST:**

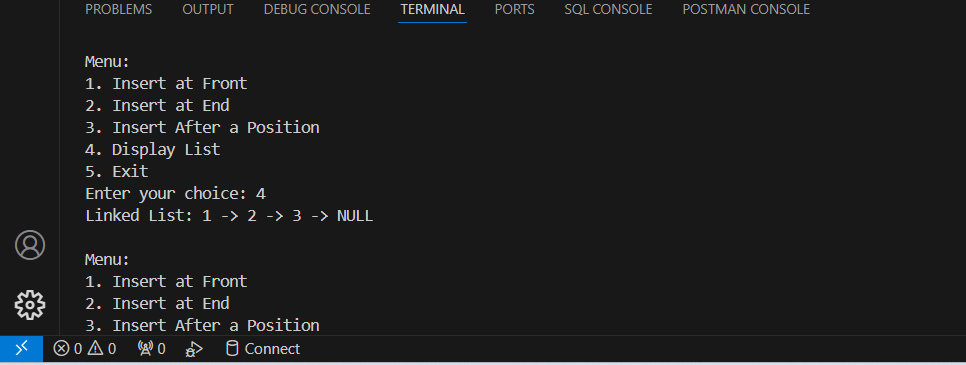
#include <stdio.h>  
#include <stdlib.h>  
  
// Define the structure for a singly linked list node  
struct node {  
    int data;  
    struct node\* next;  
};  
  
// Function to create a new node with the given data  
struct node\* createNode(int data) {  
    struct node\* newNode = (struct node\*)malloc(sizeof(struct node));  
    newNode->data = data;  
    newNode->next = NULL;  
    return newNode;  
}  
  
// Function to insert a node at the beginning of the list  
void insertFirst(struct node\*\* head, int data) {  
    struct node\* newNode = createNode(data);  
    newNode->next = \*head;  
    \*head = newNode;  
}  
  
// Function to insert a node at the end of the list  
void insertEnd(struct node\*\* head, int data) {  
    struct node\* newNode = createNode(data);  
    if (\*head == NULL) {  
        \*head = newNode;  
    } else {  
        struct node\* temp = \*head;  
        while (temp->next != NULL) {  
            temp = temp->next;  
        }  
        temp->next = newNode;  
    }  
}  
  
// Function to insert a node at a specific position in the list  
void insertAtPosition(struct node\*\* head, int data, int position) {  
    if (position < 1) {  
        printf("Position should be greater than or equal to 1\n");  
        return;  
    }  
  
    struct node\* newNode = createNode(data);  
    if (position == 1) {  
        newNode->next = \*head;  
        \*head = newNode;  
    } else {  
        struct node\* temp = \*head;  
        for (int i = 1; temp != NULL && i < position - 1; i++) {  
            temp = temp->next;  
        }  
        if (temp == NULL) {  
            printf("Position is greater than the length of the list\n");  
            return;  
        }  
        newNode->next = temp->next;  
        temp->next = newNode;  
    }  
}  
  
// Function to delete a node from the beginning of the list  
void deleteFirst(struct node\*\* head) {  
    if (\*head == NULL) {  
        printf("List is empty\n");  
        return;  
    }  
    struct node\* temp = \*head;  
    \*head = (\*head)->next;  
    free(temp);  
}  
  
// Function to delete a node from the end of the list  
void deleteEnd(struct node\*\* head) {  
    if (\*head == NULL) {  
        printf("List is empty\n");  
        return;  
    }  
  
    if ((\*head)->next == NULL) {  
        free(\*head);  
        \*head = NULL;  
        return;  
    }  
  
    struct node\* temp = \*head;  
    while (temp->next != NULL && temp->next->next != NULL) {  
        temp = temp->next;  
    }  
    free(temp->next);  
    temp->next = NULL;  
}  
  
// Function to delete a node at a specific position  
void deleteAtPosition(struct node\*\* head, int position) {  
    if (position < 1) {  
        printf("Position should be greater than or equal to 1\n");  
        return;  
    }  
  
    if (\*head == NULL) {  
        printf("List is empty\n");  
        return;  
    }  
  
    struct node\* temp = \*head;  
    if (position == 1) {  
        \*head = (\*head)->next;  
        free(temp);  
    } else {  
        for (int i = 1; temp != NULL && i < position - 1; i++) {  
            temp = temp->next;  
        }  
        if (temp == NULL || temp->next == NULL) {  
            printf("Position is greater than the length of the list\n");  
            return;  
        }  
        struct node\* toDelete = temp->next;  
        temp->next = temp->next->next;  
        free(toDelete);  
    }  
}  
  
// Function to display the list  
void displayList(struct node\* head) {  
    if (head == NULL) {  
        printf("List is empty\n");  
        return;  
    }  
  
    struct node\* temp = head;  
    while (temp != NULL) {  
        printf("%d -> ", temp->data);  
        temp = temp->next;  
    }  
    printf("NULL\n");  
}  
  
int main() {  
    struct node\* head = NULL;  
    int choice, data, position;  
  
    while (1) {  
        printf("\nMenu:\n");  
        printf("1. Insert at the beginning\n");  
        printf("2. Insert at the end\n");  
        printf("3. Insert at a specific position\n");  
        printf("4. Delete from the beginning\n");  
        printf("5. Delete from the end\n");  
        printf("6. Delete from a specific position\n");  
        printf("7. Display the list\n");  
        printf("8. Exit\n");  
        printf("Enter your choice: ");  
        scanf("%d", &choice);  
  
        switch (choice) {  
            case 1:  
                printf("Enter the value to insert at the beginning: ");  
                scanf("%d", &data);  
                insertFirst(&head, data);  
                break;  
  
            case 2:  
                printf("Enter the value to insert at the end: ");  
                scanf("%d", &data);  
                insertEnd(&head, data);  
                break;  
  
            case 3:  
                printf("Enter the value to insert: ");  
                scanf("%d", &data);  
                printf("Enter the position to insert: ");  
                scanf("%d", &position);  
                insertAtPosition(&head, data, position);  
                break;  
  
            case 4:  
                deleteFirst(&head);  
                break;  
  
            case 5:  
                deleteEnd(&head);  
                break;  
  
            case 6:  
                printf("Enter the position to delete: ");  
                scanf("%d", &position);  
                deleteAtPosition(&head, position);  
                break;  
  
            case 7:  
                displayList(head);  
                break;  
  
            case 8:  
                printf("Exiting...\n");  
                exit(0);  
  
            default:  
                printf("Invalid choice! Please try again.\n");  
        }  
    }  
  
    return 0;  
}

OUTPUT:











**WEEK 7**

**SORTING A LINKED LIST ,REVERSING AND CONCATENATE:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

    int data;

    struct Node\* next;

};

struct Node\* createNode(int data) {

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

    newNode->data = data;

    newNode->next = NULL;

    return newNode;

}

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

    struct Node\* newNode = createNode(data);

    if (\*head == NULL) {

        \*head = newNode;

        return;

    }

    struct Node\* temp = \*head;

    while (temp->next != NULL) {

        temp = temp->next;

    }

    temp->next = newNode;

}

void display(struct Node\* head) {

    if (head == NULL) {

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

        return;

    }

    struct Node\* temp = head;

    while (temp != NULL) {

        printf("%d -> ", temp->data);

        temp = temp->next;

    }

    printf("NULL\n");

}

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

    struct Node\* prev = NULL;

    struct Node\* current = \*head;

    struct Node\* next = NULL;

    while (current != NULL) {

        next = current->next;

        current->next = prev;

        prev = current;

        current = next;

    }

    \*head = prev;

}

void bubbleSort(struct Node\* head) {

    if (head == NULL) return;

    struct Node\* current;

    struct Node\* lastSorted = NULL;

    int swapped;

    do {

        swapped = 0;

        current = head;

        while (current->next != lastSorted) {

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

                int temp = current->data;

                current->data = current->next->data;

                current->next->data = temp;

                swapped = 1;

            }

            current = current->next;

        }

        lastSorted = current;

    } while (swapped);

}

void concatenate(struct Node\*\* head1, struct Node\*\* head2) {

    if (\*head1 == NULL) {

        \*head1 = \*head2;

        return;

    }

    struct Node\* temp = \*head1;

    while (temp->next != NULL) {

        temp = temp->next;

    }

    temp->next = \*head2;

}

int main() {

    struct Node\* list1 = NULL;

    struct Node\* list2 = NULL;

    printf("Appending elements to the first linked list:\n");

    append(&list1, 33);

    append(&list1, 10);

    append(&list1, 56);

    display(list1);

    printf("\nSorting the first linked list:\n");

    bubbleSort(list1);

    display(list1);

    printf("\nReversing the first linked list:\n");

    reverse(&list1);

    display(list1);

    printf("\nAppending elements to the second linked list:\n");

    append(&list2, 42);

    append(&list2, 5);

    append(&list2, 69);

    display(list2);

    printf("\nConcatenating the two linked lists:\n");

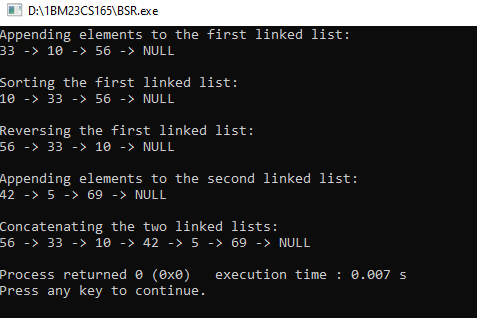
    concatenate(&list1, &list2);

    display(list1);

    return 0;

}

OUTPUT:



**2.STACK AND QUEUE USING LINKED LIST:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

void push(struct Node\*\* top, int data) {

struct Node\* newNode = createNode(data);

newNode->next = \*top;

\*top = newNode;

printf("%d pushed onto the stack.\n", data);

}

int pop(struct Node\*\* top) {

if (\*top == NULL) {

printf("Stack underflow.\n");

return -1;

}

struct Node\* temp = \*top;

int poppedData = temp->data;

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

free(temp);

return poppedData;

}

void displayStack(struct Node\* top) {

if (top == NULL) {

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

return;

}

printf("Stack: ");

struct Node\* temp = top;

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

void enqueue(struct Node\*\* front, struct Node\*\* rear, int data) {

struct Node\* newNode = createNode(data);

if (\*rear == NULL) {

\*front = \*rear = newNode;

printf("%d enqueued into the queue.\n", data);

return;

}

(\*rear)->next = newNode;

\*rear = newNode;

printf("%d enqueued into the queue.\n", data);

}

int dequeue(struct Node\*\* front, struct Node\*\* rear) {

if (\*front == NULL) {

printf("Queue underflow.\n");

return -1;

}

struct Node\* temp = \*front;

int dequeuedData = temp->data;

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

if (\*front == NULL) {

\*rear = NULL;

}

free(temp);

return dequeuedData;

}

void displayQueue(struct Node\* front) {

if (front == NULL) {

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

return;

}

printf("Queue: ");

struct Node\* temp = front;

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

int main() {

struct Node\* stack = NULL;

struct Node\* queueFront = NULL;

struct Node\* queueRear = NULL;

printf("STACK OPERATIONS:\n");

push(&stack, 11);

push(&stack, 52);

push(&stack, 30);

displayStack(stack);

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

displayStack(stack);

printf("\nQUEUE OPERATIONS:\n");

enqueue(&queueFront, &queueRear, 76);

enqueue(&queueFront, &queueRear, 43);

enqueue(&queueFront, &queueRear, 72);

displayQueue(queueFront);

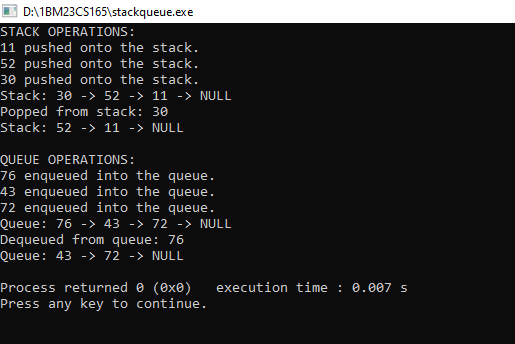
printf("Dequeued from queue: %d\n", dequeue(&queueFront, &queueRear));

displayQueue(queueFront);

return 0;

}

**OUTPUT:**

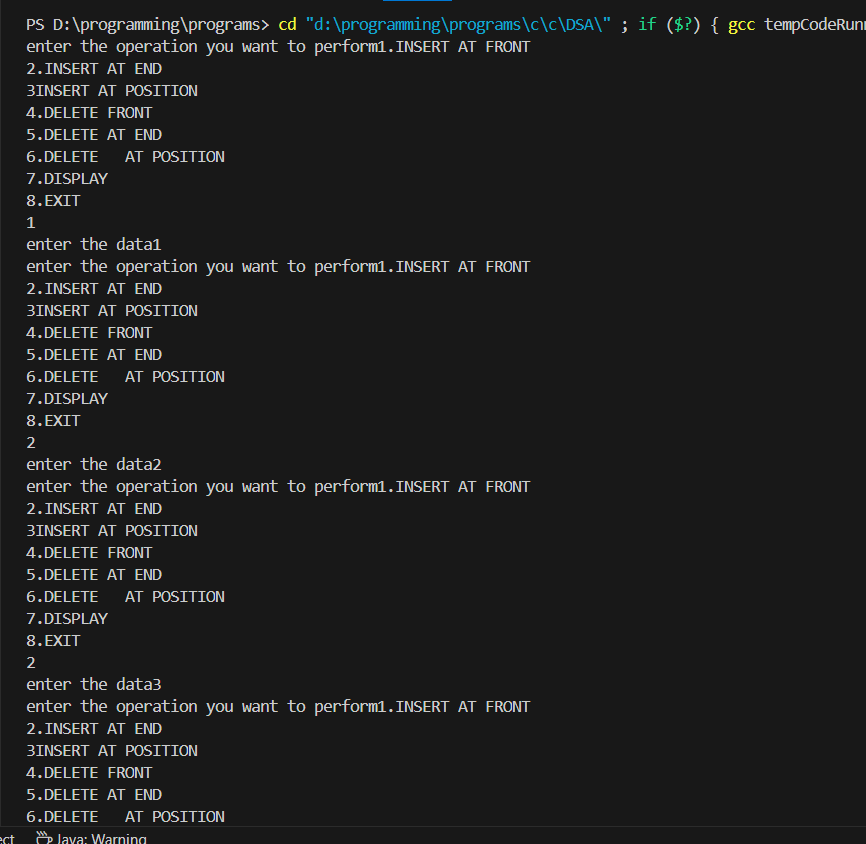


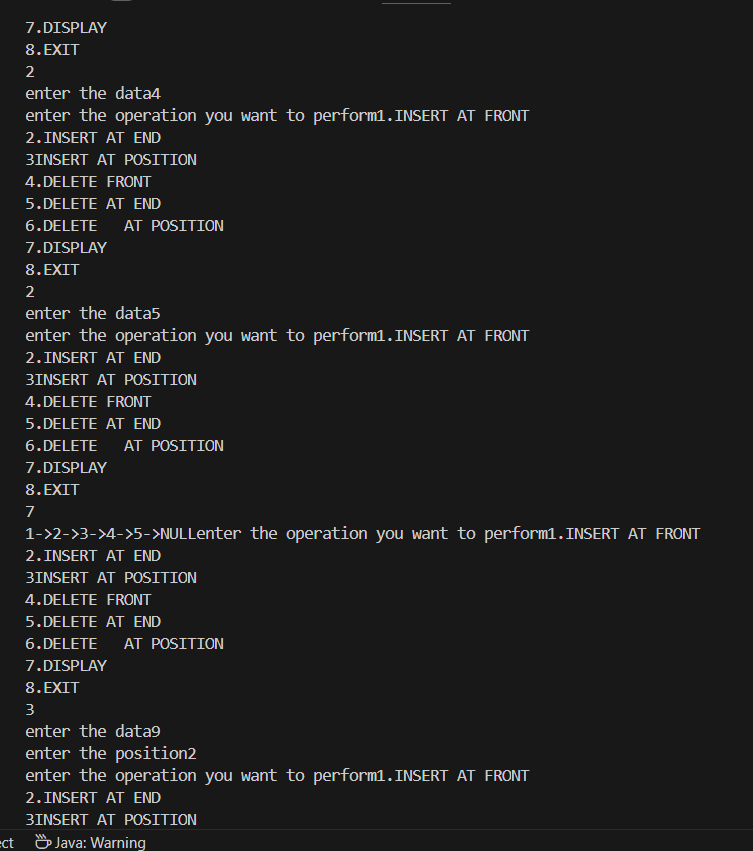
**WEEK 8**

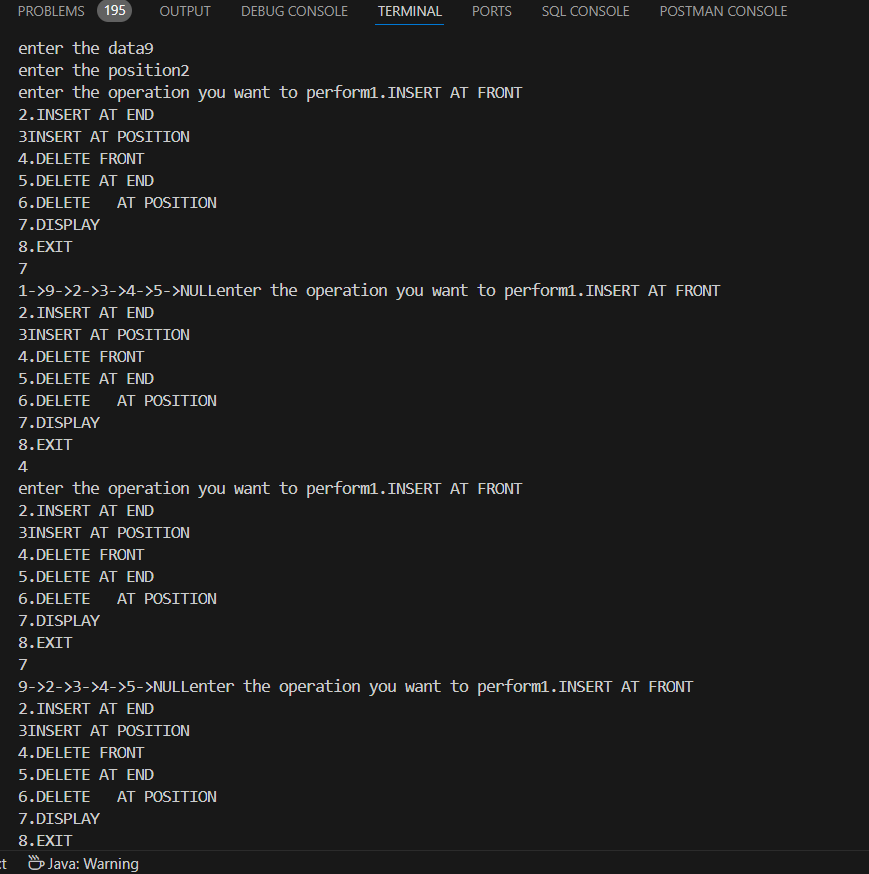
**Q.Program to implement Doubly linked list Insertion and deletion:**

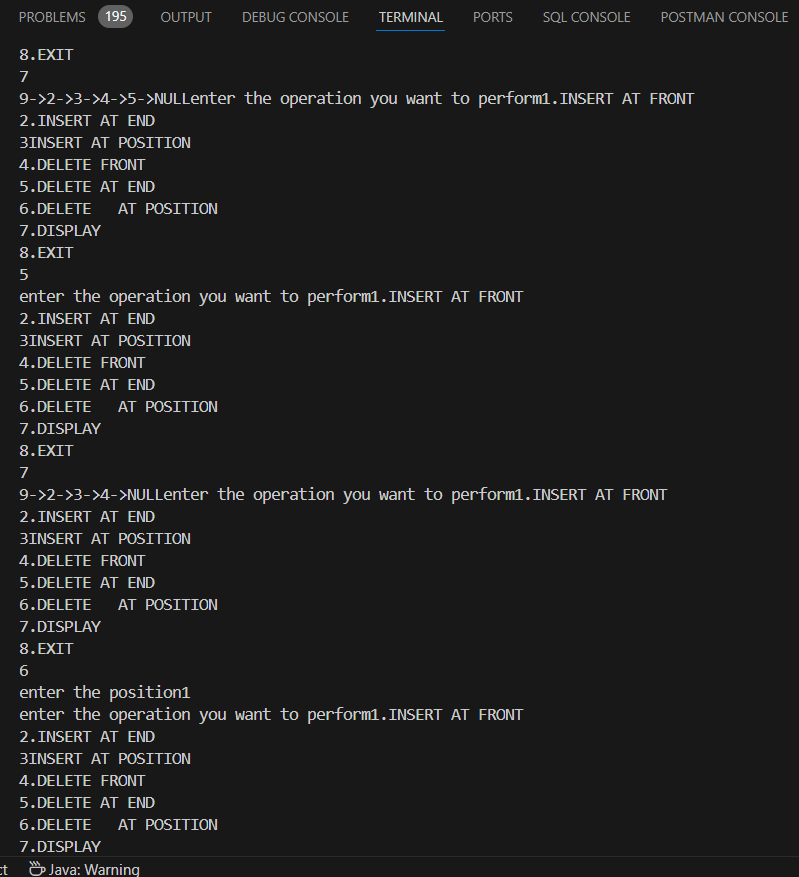
#include<stdio.h>  
#include<stdlib.h>  
  
struct node  
{  
    struct node\*prev;  
    int data;  
    struct  node\*next;  
};  
struct node\* create(int d)  
{  
    struct node\* newnode=(struct node\*)malloc(sizeof(struct node));  
    newnode->data=d;  
};  
void insert\_first(struct node\*\* head,int d)  
{  
    struct node\* newnode=create(d);  
    if(\*head==NULL)  
    {  
        newnode->next=NULL;  
        newnode->prev=NULL;  
        \*head=newnode;  
    }  
    else{  
        newnode->next=\*head;  
        newnode->prev=NULL;  
        \*head=newnode;  
    }  
}  
void insert\_end(struct node\*\* head,int d)  
{  
    struct node\* newnode=create(d);  
    if(\*head==NULL)  
    {  
        \*head=newnode;  
        newnode->next=NULL;  
        newnode->prev=NULL;  
        \*head=newnode;  
    }  
    else{  
        struct node\* temp=\*head;  
        while(temp->next!=NULL)  
        {  
  
            temp=temp->next;  
        }  
        temp->next=newnode;  
        newnode->prev=temp;  
        newnode->next=NULL;  
    }  
}  
  
void del\_first(struct node\*\* head)  
{  
  
    if(\*head==NULL)  
    {  
  
        printf("empty\n");  
        return;  
    }  
    else{  
        struct node\* temp=\*head;  
        \*head=temp->next;  
        free(temp);  
    }  
}  
  
void del\_end(struct node\*\* head)  
{struct node\* temp=\*head;  
    if(\*head==NULL)  
    {  
        printf("empty!");  
    }  
    else if(temp->next==NULL)  
    {  
  
        \*head=NULL;  
    }  
    else{  
  
        while(temp->next->next!=NULL)  
        {  
  
            temp=temp->next;  
        }  
        temp->next=NULL;  
  
    }  
}  
  
void display(struct node\*\* head)  
{  
    struct node\* temp=\*head;  
    if(\*head==NULL)  
    {  
        printf("empty!!");  
    }  
    else{  
        while(temp->next!=NULL)  
        {  
            printf("%d->",temp->data);  
            temp=temp->next;  
        }  
        printf("%d->NULL",temp->data);  
    }  
}  
void insert\_at\_pos(struct node\*\* head,int d,int pos)  
{  
    int count;  
    struct node\* t=\*head;  
    struct node\* temp=\*head;  
    struct node\* newnode=create(d);  
    while(t->next!=NULL)  
    {  
        count++;  
        t=t->next;  
    }  
    count++;  
    if(count<pos)  
    {  
        printf("invalid position");  
    }  
    else if(pos==0)  
    {  
        insert\_first(head,d);  
    }  
    else if(pos==count)  
    {  
        insert\_end(head,d);  
    }  
    else{  
        for(int i=0;i<pos-2;i++)  
        {  
            temp=temp->next;  
        }  
        newnode->next=temp->next;  
        temp->next=newnode;  
        newnode->prev=temp;  
    }  
  
}  
void del\_at\_pos(struct node\*\* head,int pos)  
{  
    int count;  
    struct node\* t=\*head;  
    struct node\* temp=\*head;  
    while(t->next!=NULL)  
    {  
        count++;  
        t=t->next;  
    }  
    count++;  
    if(count<pos)  
    {  
        printf("invalid position");  
    }  
    else if(pos==0)  
    {  
        del\_first(head);  
    }  
    else if(pos==count)  
    {  
        del\_end(head);  
    }  
    else{  
        for(int i=0;i<pos-1;i++)  
        {  
            temp=temp->next;  
        }  
        temp->next=temp->next->next;  
        temp->next->prev=temp;  
    }  
}  
int main()  
{  
    struct node\* head=NULL;  
    int d,pos,ele,c;  
    while(1)  
    {  
        printf("enter the operation you want to perform");  
        printf("1.INSERT AT FRONT\n");  
        printf("2.INSERT AT END\n");  
        printf("3INSERT AT POSITION\n");  
        printf("4.DELETE FRONT\n");  
        printf("5.DELETE AT END\n");  
        printf("6.DELETE   AT POSITION\n");  
        printf("7.DISPLAY\n");  
        printf("8.EXIT\n");  
        scanf("%d",&c);  
        switch(c)  
        {  
        case 1:  
            printf("enter the data");  
            scanf("%d",&d);  
            insert\_first(&head,d);  
            break;  
        case 2:  
            printf("enter the data");  
            scanf("%d",&d);  
            insert\_end(&head,d);  
            break;  
        case 3:  
            printf("enter the data");  
            scanf("%d",&d);  
            printf("enter the position");  
            scanf("%d",&pos);  
            insert\_at\_pos(&head,d,pos);  
            break;  
        case 4:  
            del\_first(&head);  
            break;  
        case 5:  
            del\_end(&head);  
            break;  
        case 6:  
            printf("enter the position");  
            scanf("%d",&pos);  
            del\_at\_pos(&head,pos);  
            break;  
        case 7:  
            display(&head);  
            break;  
        case 8:  
            printf("exiting---\n");  
            return 0;  
        }  
  
    }  
    return 0;  
}

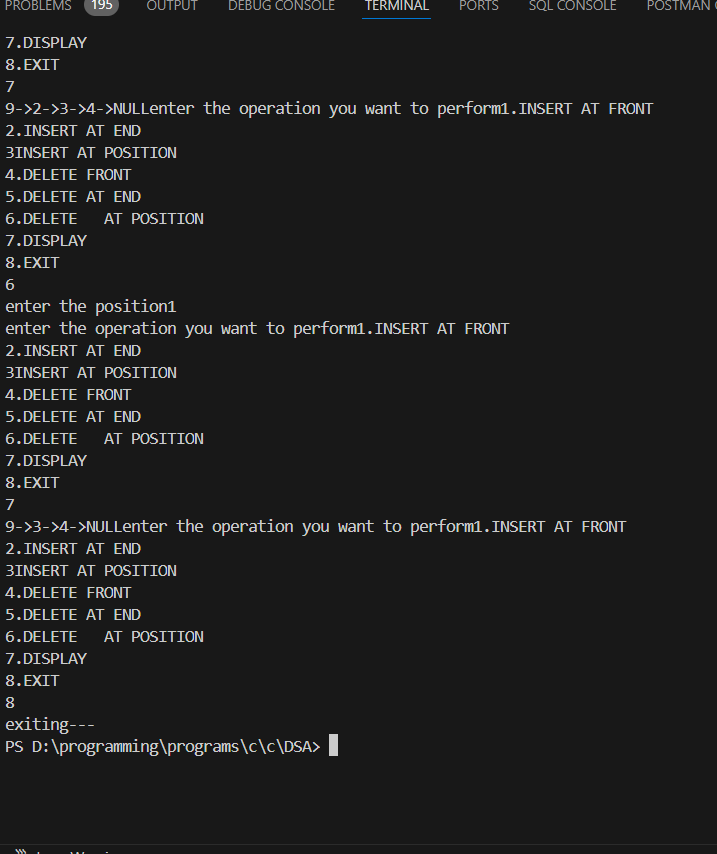
**OUTPUT:**











**2.BINARY SEARCH TREE:**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node\* left;

struct Node\* right;

} Node;

Node\* createNode(int data) {

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

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

Node\* insert(Node\* root, int data) {

if (root == NULL) {

return createNode(data);

}

if (data < root->data) {

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

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

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

}

return root;

}

void inorder(Node\* root) {

if (root != NULL) {

inorder(root->left);

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

inorder(root->right);

}

}

void preorder(Node\* root) {

if (root != NULL) {

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

preorder(root->left);

preorder(root->right);

}

}

void postorder(Node\* root) {

if (root != NULL) {

postorder(root->left);

postorder(root->right);

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

}

}

void display(Node\* root) {

printf("Tree elements (In-order): ");

inorder(root);

printf("\n");

}

int main() {

Node\* root = NULL;

int choice, data;

while (1) {

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

printf("1. Insert\n");

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

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

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

printf("5. Display Tree Elements\n");

printf("6. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter data to insert: ");

scanf("%d", &data);

root = insert(root, data);

break;

case 2:

printf("In-order Traversal: ");

inorder(root);

printf("\n");

break;

case 3:

printf("Pre-order Traversal: ");

preorder(root);

printf("\n");

break;

case 4:

printf("Post-order Traversal: ");

postorder(root);

printf("\n");

break;

case 5:

display(root);

break;

case 6:

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

exit(0);

default:

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

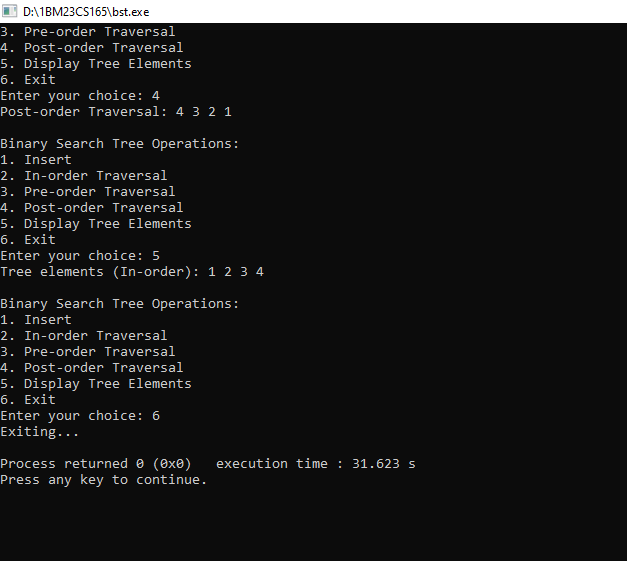
}

}

return 0;

}

OUTPUT:



WEEK 9:

**//DFS Traversal:**

#include<stdio.h>

#include<stdlib.h>

int arr[10][10],vis[10];

int n;

void dfs(int a);

void main()

{

    printf("enter the number of vertices");

    scanf("%d",&n);

    printf("n");

    printf("enter adjacency matrix");

    for(int i=0;i<n;i++)

    {

        for(int j=0;j<n;j++)

        {

            scanf("%d",&arr[i][j]);

        }

    }

    for(int i=0;i<n;i++)

    {

        vis[i]=0;

    }

    printf("dfs traversal");

    for(int i=0;i<n;i++)

    {

        if(vis[i]==0)

        {

            dfs(i);

        }

  }

}

void dfs(int a)

{

    printf("%d",a);

    vis[a]=1;

    for(int i=0;i<n;i++)

    {

        if(arr[a][a]==1&&arr[i]==0)

        {

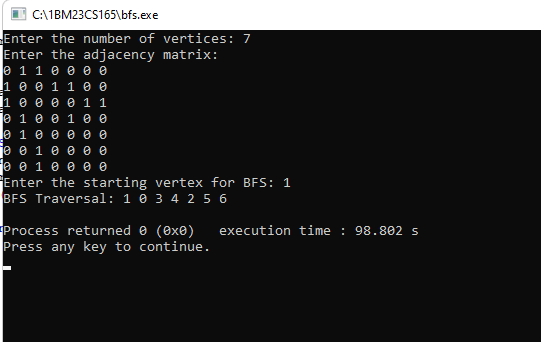
            dfs(i);

        }

    }

}

OUTPUT:



**//BFS Traversal**:

#include <stdio.h>

#include <stdlib.h>

#define MAX 100 // Maximum number of vertices

int queue[MAX], front = -1, rear = -1; // Queue for BFS

int visited[MAX]; // To track visited nodes

// Function to add an element to the queue

void enqueue(int vertex) {

    if (rear == MAX - 1) return; // Queue overflow

    if (front == -1) front = 0;

    queue[++rear] = vertex;

}

// Function to remove an element from the queue

int dequeue() {

    if (front == -1 || front > rear) return -1; // Queue underflow

    return queue[front++];

}

// Function to perform BFS

void bfs(int graph[MAX][MAX], int n, int start) {

    for (int i = 0; i < n; i++) visited[i] = 0; // Reset visited array

  enqueue(start);

    visited[start] = 1;

    printf("BFS Traversal: ");

    while (front <= rear) {

        int current = dequeue();

        printf("%d ", current);

        for (int i = 0; i < n; i++) {

            if (graph[current][i] == 1 && !visited[i]) {

                enqueue(i);

                visited[i] = 1;

            }

        }

    }

    printf("\n");

}

int main() {

int graph[MAX][MAX], n, start;

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

    scanf("%d", &n);

    printf("Enter the adjacency matrix:\n");

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < n; j++) {

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

        }

    }

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

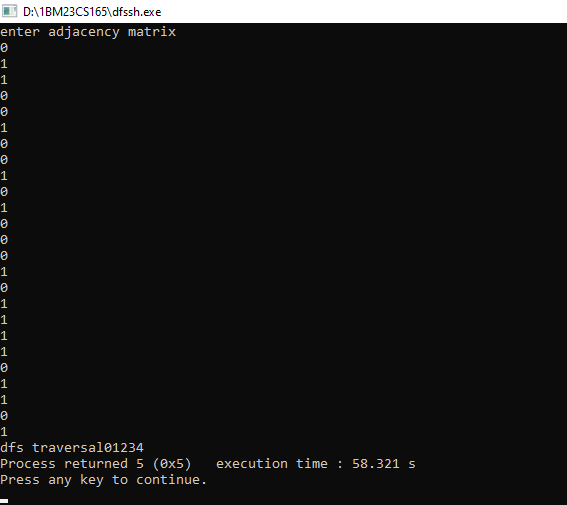
    scanf("%d", &start);

    bfs(graph, n, start);

    return 0;

}

2.Output for DFS:



10)Given a File of N employee records with a set K of Keys(4-digit) which uniquely determine the records in file F. Assume that file F is maintained in memory by a Hash Table (HT) of m memory locations with L as the set of memory addresses (2-digit) of locations in HT. Let the keys in K and addresses in L are integers. Design and develop a Program in C that uses Hash function H: K -> L as H(K)=K mod m (remainder method), and implement hashing technique to map a given key K to the address space L. Resolve the collision (if any) using linear probing

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_KEYS 100

#define TABLE\_SIZE 10

typedef struct {

int key;

char name[50];

} Employee;

typedef struct {

int key;

int isOccupied;

} HashTableEntry;

int hashFunction(int key, int m) {

return key % m;

}

int linearProbing(HashTableEntry hashTable[], int key, int m) {

int index = hashFunction(key, m);

while (hashTable[index].isOccupied) {

index = (index + 1) % m;

}

return index;

}

void insertEmployee(HashTableEntry hashTable[], int key, int m) {

int index = linearProbing(hashTable, key, m);

hashTable[index].key = key;

hashTable[index].isOccupied = 1;

}

void printHashTable(HashTableEntry hashTable[], int m) {

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

if (hashTable[i].isOccupied)

printf("Index %d: Key %d\n", i, hashTable[i].key);

else

printf("Index %d: Empty\n", i);

}

}

int main() {

int m = TABLE\_SIZE, N;

scanf("%d", &N);

HashTableEntry hashTable[m];

for (int i = 0; i < m; i++) hashTable[i].isOccupied = 0;

Employee employees[MAX\_KEYS];

int key;

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

scanf("%d", &employees[i].key);

getchar();

fgets(employees[i].name, sizeof(employees[i].name), stdin);

employees[i].name[strcspn(employees[i].name, "\n")] = '\0';

insertEmployee(hashTable, employees[i].key, m);

}

printHashTable(hashTable, m);

return 0;

}

**OUTPUT:**

