**Name-Barsha Routh**

**RegNo- 24MCA0164**

**Course title- Data structures and Algorithms Lab  
Course Code- PMCA501P**

**ASSIGNMENT 1**

**1.Stack**

#include<stdio.h>

#include<stdlib.h>

struct node

    {

        int data;

        struct node \*next;

    };

struct node \*head=NULL, \*tail=NULL;

int len = 0;

void insert(){

    struct node \*newnode=(struct node\*)malloc(sizeof(struct node));

    printf("Enter data: ");

    scanf("%d",&newnode->data);

    newnode->next=NULL;

    len++;

    if (head==NULL)

        head = newnode;

    else

        tail->next=newnode;

    tail = newnode;

    display();

}

void insertAtBegin(){

    struct node \*newnode=(struct node\*)malloc(sizeof(struct node));

    printf("Enter data: ");

    scanf("%d",&newnode->data);

    newnode->next=head;

    len++;

    if(head==NULL)

        tail=newnode;

    head=newnode;

    display();

}

void insertAtEnd(){

    struct node \*newnode=(struct node\*)malloc(sizeof(struct node));

    printf("Enter data: ");

    scanf("%d",&newnode->data);

    newnode->next=NULL;

    len++;

    if(head==NULL)

        head = newnode;

    else

        tail->next=newnode;

    tail = newnode;

    display();

}

void display(){

    struct node \*temp = head;

    if(head==NULL)

        printf("EMPTY\n");

    else

    {

        while(temp!=NULL){

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

            temp=temp->next;

        }

        printf("\n\n");

    }

}

void insertAtPos(){

    int pos,i;

    printf("Enter position: ");

    scanf("%d",&pos);

    if(pos==1){

        insertAtBegin();

        return;

    }

    if(pos==len+1){

        insert();

        return;

    }

    if(pos > len || pos<1){

        printf("Invalid position\n\n");

        return;

    }

    struct node \*newnode=(struct node\*)malloc(sizeof(struct node));

    printf("Enter data: ");

    scanf("%d",&newnode->data);

    len++;

    struct node \*temp = head;

    while(i < pos-2){

        temp = temp->next;

        i++;

    }

    newnode->next = temp->next;

    temp->next=newnode;

    display();

}

void deleteAtBeginning(){

    if(head==NULL){

        printf("List already EMPTY\n");return;

    }

    struct node \*temp=head;

    head = head->next;  //logical deletion

    free(temp);     //physical deletion

    display();

    len--;

}

void deleteAtEnd(){

    struct node \*temp;

    // int i;

    if(head==NULL){

        printf("List already EMPTY\n");return;

    }

    temp=head;

    while(temp<len){

        temp=temp->next;

    }

    temp->next=NULL;

    tail=temp;

    len--;

    display();

}

void deleteAtPos(){

    int pos,i=1;

    struct node \*temp;

    temp=head;

    printf("Enter position of element to be deleted: ");

    scanf("%d",&pos);

    if(pos==1){

        deleteAtBeginning();

        return;

    }

    if(pos==len){

        deleteAtEnd();

        return;

    }

    while(i<pos-1){

        temp=temp->next;

        i++;

    }

    struct node \*delNode=temp->next;

    temp->next=delNode->next;

    if(delNode->next==NULL)

        tail=temp;

    free(delNode);

    len--;

    display();

}

void main(){

    int choice;

    do

    {

        printf("1.INSERT\n2.INsert at begin\n3.insert at position\n4.display\n5 Delete Head\n6.DELETE AT END\n7.DELETE AT POSITION\nEnter your choice: ");

        scanf("%d",&choice);

        switch (choice)

        {

            case 1:insert();break;

            case 2:insertAtBegin();break;

            case 3:insertAtPos();break;

            case 4:display();break;

            case 5:deleteAtBeginning();break;

            case 6:deleteAtEnd();break;

            case 7:deleteAtPos();break;

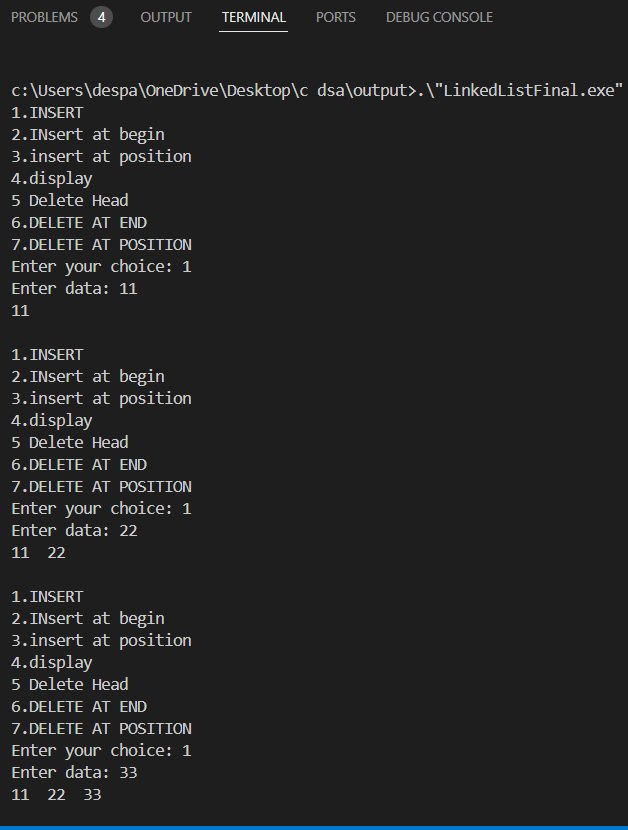
            default:printf("INVALID");break;

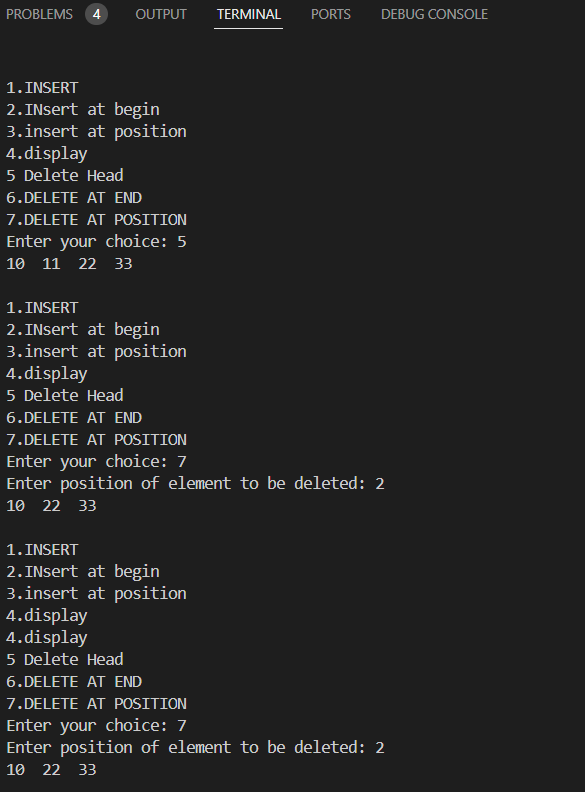
        }

    }while(choice!=0);

}

**Output**

****

****

**2.Infix to Postfix Conversion**

#include<stdio.h>

#include<conio.h>

char infix[20],postfix[20];

struct stack{

    int s[20];

    int top;

}st;

void push(int ele){

    st.top++;

    st.s[st.top]=ele;

}

char pop(){

    int el;

    char e;

    el=st.s[st.top];

    st.top--;

    switch (el)

    {

    case 1:

        e='+';

        break;

    case 2:

        e='-';

        break;

    case 3:

        e='\*';

        break;

    case 4:

        e='/';

        break;

    case 5:

        e='^';

        break;

    }

    return e;

}

void post(){

    int i,j=0;

    for ( i = 0; infix[i] != '\0'; i++)

    {

        switch (infix[i])

        {

        case '+':

        {

            while (st.s[st.top]>=1){

                postfix[j]=pop();

                j++;

            }

            push(1);

            break;

        }

        case '-':

        {

            while(st.s[st.top]>=1)

            {

                postfix[j]=pop();

                j++;

            }

            push(2);

            break;

        }

        case '\*':

        {

            while (st.s[st.top]>=3)

            {

                postfix[j]=pop();

                j++;

            }

            push(3);

            break;

        }

        case '/':

        {

            while (st.s[st.top]>=3)

            {

                postfix[j]=pop();

                j++;

            }

            push(4);

            break;

        }

        case '^':

        {

            while (st.s[st.top]>=4){

                postfix[j]=pop();

                j++;

            }

            push(5);

            break;

        }

        case '(':

        {

            push(0);

            break;

        }

        case ')':

        {

            while (st.s[st.top]!=0)

            {

                postfix[j]=pop();

                j++;

            }

            st.top--;

            break;

        }

        default:

        //when any operand comes...A or B or C....

        postfix[j++]=infix[i];

        }

    }

    while (st.top>=0)

    {

        postfix[j++]=pop();

        // printf("PostFix Expression is :",postfix);

    }

    printf("PostFix Expression is : %s",postfix);

}

void main(){

    st.top=-1;

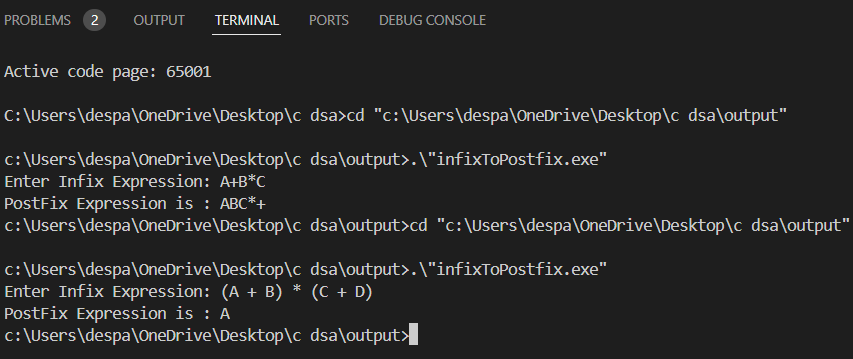
    printf("Enter Infix Expression: ");

    scanf("%s",&infix);

    post();

}

**OUTPUT**

****

**3.Infix to Prefix Conversion**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

char infix[20], prefix[20], temp[20];

struct stack {

    int s[20];

    int top;

} st;

void push(int ele) {

    st.top++;

    st.s[st.top] = ele;

}

char pop() {

    int el;

    char e;

    el = st.s[st.top];

    st.top--;

    switch (el) {

        case 1:

            e = '+';

            break;

        case 2:

            e = '-';

            break;

        case 3:

            e = '\*';

            break;

        case 4:

            e = '/';

            break;

        case 5:

            e = '^';

            break;

    }

    return e;

}

void infixToPostfix() {

    int i, j = 0;

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

        switch (infix[i]) {

            case '+':

                while (st.s[st.top] >= 1) {

                    prefix[j++] = pop();

                }

                push(1);

                break;

            case '-':

                while (st.s[st.top] >= 1) {

                    prefix[j++] = pop();

                }

                push(2);

                break;

            case '\*':

                while (st.s[st.top] >= 3) {

                    prefix[j++] = pop();

                }

                push(3);

                break;

            case '/':

                while (st.s[st.top] >= 3) {

                    prefix[j++] = pop();

                }

                push(4);

                break;

            case '^':

                while (st.s[st.top] >= 4) {

                    prefix[j++] = pop();

                }

                push(5);

                break;

            case '(':

                push(0);

                break;

            case ')':

                while (st.s[st.top] != 0) {

                    prefix[j++] = pop();

                }

                st.top--;

                break;

            default:

                // When any operand comes (A, B, C, etc.)

                prefix[j++] = infix[i];

        }

    }

    while (st.top >= 0) {

        prefix[j++] = pop();

    }

    prefix[j] = '\0';

}

void reverse(char \*exp) {

    int len = strlen(exp);

    for (int i = 0; i < len / 2; i++) {

        char temp = exp[i];

        exp[i] = exp[len - i - 1];

        exp[len - i - 1] = temp;

    }

}

void replaceBrackets(char \*exp) {

    for (int i = 0; exp[i] != '\0'; i++) {

        if (exp[i] == '(') {

            exp[i] = ')';

        } else if (exp[i] == ')') {

            exp[i] = '(';

        }

    }

}

void infixToPrefix() {

    // Reversing

    reverse(infix);

    //'(' -> ')' and ')' -> '('

    replaceBrackets(infix);

    // infix to postfix

    infixToPostfix();

    // Step 4: Reverseing to get the prefix

    reverse(prefix);

    printf("Prefix Expression is: %s\n", prefix);

}

int main() {

    st.top = -1;

    printf("Enter Infix Expression: ");

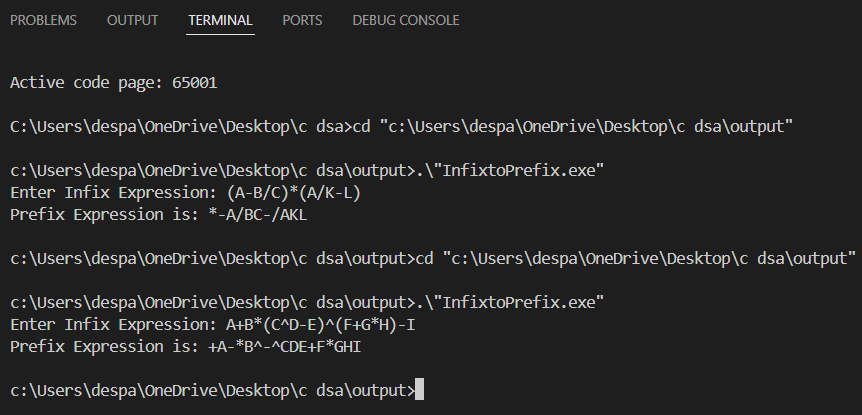
    scanf("%s", infix);

    infixToPrefix();

    return 0;

}

**OUTPUT**

****

**4. Postfix Evaluation**

#include <stdio.h>

char postfix[20];

struct stack {

    int s[20];

    int top;

} st;

void push(int ele) {

    st.top++;

    st.s[st.top] = ele;

}

int pop() {

    int el = st.s[st.top];

    st.top--;

    return el;

}

void eval() {

    int i, op1, op2, result;

    for (i = 0; postfix[i] != '\0'; i++) {

        switch (postfix[i]) {

            case '+':

                op2 = pop();

                op1 = pop();

                result = op1 + op2;

                push(result);

                break;

            case '-':

                op2 = pop();

                op1 = pop();

                result = op1 - op2;

                push(result);

                break;

            case '\*':

                op2 = pop();

                op1 = pop();

                result = op1 \* op2;

                push(result);

                break;

            case '/':

                op2 = pop();

                op1 = pop();

                if (op2 != 0) { // Check for division by zero

                    result = op1 / op2;

                } else {

                    printf("Error: Division by zero\n");

                    return;

                }

                push(result);

                break;

            default:

                // Converting char digit to integer and push onto stack

                push(postfix[i] - '0');

        }

    }

}

int main() {

    st.top = -1;

    printf("Enter postfix expression: ");

    scanf("%s", postfix);

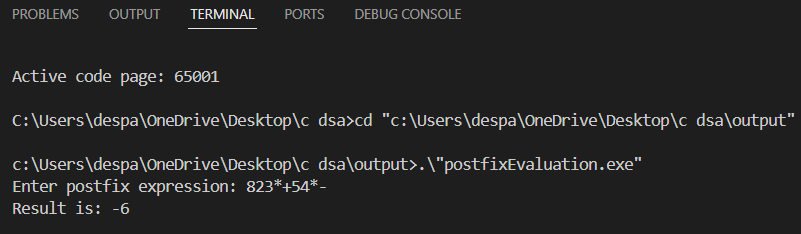
    eval();

    printf("Result is: %d\n", st.s[st.top]);

    return 0;

}

**Output**

****

**5. Prefix Evaluation**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define SIZE 20

void eval();

void push(int);

int pop();

void reverse(char\*);

void prefixToPostfix(char\*, char\*);

char prefix[SIZE];

char postfix[SIZE];

struct stack {

    int s[SIZE];

    int top;

} st;

void main() {

    st.top = -1;

    printf("Enter the prefix expression: ");

    scanf("%s", prefix);

    prefixToPostfix(prefix, postfix);

    eval();

    printf("Result: %d\n", st.s[st.top]);

}

void eval() {

    int i, op1, op2, result;

    for (i = 0; postfix[i] != '\0'; i++) {

        if (isdigit(postfix[i])) {

            push(postfix[i] - '0');

        } else {

            op1 = pop();

            op2 = pop();

            switch (postfix[i]) {

                case '+': result = op2 + op1; break;

                case '-': result = op2 - op1; break;

                case '\*': result = op2 \* op1; break;

                case '/':

                    if (op1 != 0) result = op2 / op1;

                    else {

                        printf("Error: Division by zero\n");

                        return;

                    }

                    break;

                default:

                    printf("Error: Invalid operator\n");

                    return;

            }

            push(result);

        }

    }

}

void push(int ele) {

    if (st.top < SIZE - 1) {

        st.s[++st.top] = ele;

    } else {

        printf("Stack Overflow\n");

    }

}

int pop() {

    if (st.top >= 0) {

        return st.s[st.top--];

    } else {

        printf("Stack Underflow\n");

        return -1;

    }

}

void reverse(char\* exp) {

    int len = strlen(exp);

    for (int i = 0; i < len / 2; i++) {

        char temp = exp[i];

        exp[i] = exp[len - i - 1];

        exp[len - i - 1] = temp;

    }

}

void prefixToPostfix(char\* prefix, char\* postfix) {

    reverse(prefix);

    int j = 0;

    for (int i = 0; prefix[i] != '\0'; i++) {

        if (isdigit(prefix[i])) {

            postfix[j++] = prefix[i];

        } else if (prefix[i] == '+' || prefix[i] == '-' || prefix[i] == '\*' || prefix[i] == '/') {

            while (st.top != -1 && (st.s[st.top] == '\*' || st.s[st.top] == '/' || st.s[st.top] == '+' || st.s[st.top] == '-')) {

                postfix[j++] = pop();

            }

            push(prefix[i]);

        }

    }

    while (st.top != -1) {

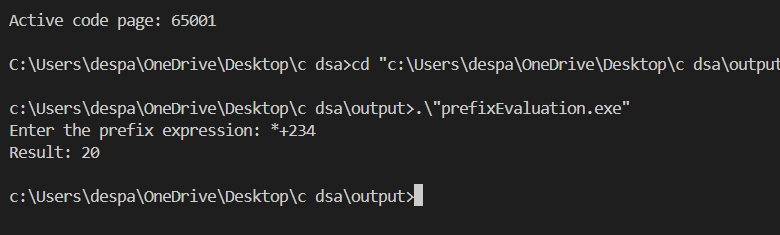
        postfix[j++] = pop();

    }

    postfix[j] = '\0';

}

**OUTPUT**

****

**6.LINEAR QUEUE OPERATIONS**

#include <stdio.h>

#include <stdlib.h>

struct queue {

    int size;

    int f; // Front index

    int r; // Rear index

    int \*arr;

};

void initQueue(struct queue \*q, int size) {

    q->size = size;

    q->f = -1;

    q->r = -1;

    q->arr = (int\*)malloc(size \* sizeof(int));

}

int isEmpty(struct queue \*q) {

    return q->f == -1; // Queue is empty if front is -1

}

int isFull(struct queue \*q) {

    return q->r == q->size - 1; // Queue is full if rear is at the last position

}

void enqueue(struct queue \*q, int val) {

    if (isFull(q)) {

        printf("Queue is full\n");

    } else {

        if (isEmpty(q)) {

            q->f = 0; // Queue was empty, so set front to 0

        }

        q->r++;

        q->arr[q->r] = val;

        printf("Enqueued element %d\n", val);

    }

}

int dequeue(struct queue \*q) {

    int a = -1;

    if (isEmpty(q)) {

        printf("Queue is empty\n");

    } else {

        a = q->arr[q->f];

        if (q->f >= q->r) {

            // Queue has become empty

            q->f = -1;

            q->r = -1;

        } else {

            q->f++;

        }

    }

    return a;

}

//view peek ele

int peek(struct queue \*q) {

    if (isEmpty(q)) {

        printf("Queue is empty\n");

        return -1;

    }

    return q->arr[q->f];

}

void display(struct queue \*q) {

    if (isEmpty(q)) {

        printf("Queue is empty\n");

        return;

    }

    printf("Queue elements: ");

    for (int i = q->f; i <= q->r; i++) {

        printf("%d ", q->arr[i]);

    }

    printf("\n");

}

int main() {

    struct queue q;

    int size = 10;

    initQueue(&q, size);

    enqueue(&q, 12);

    enqueue(&q, 13);

    enqueue(&q, 14);

    display(&q);

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

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

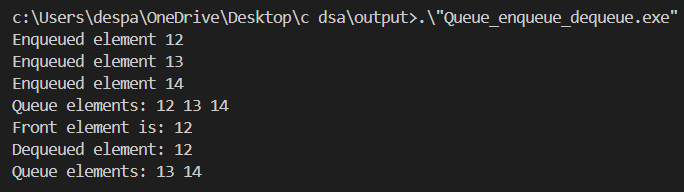
    display(&q);

    free(q.arr);

    return 0;

}

**OUTPUT**

****

**7.CIRCULAR QUEUE**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

typedef struct {

    int size;

    int f;

    int r;

    int \*arr;

} CircularQueue;

void initQueue(CircularQueue \*q, int size) {

    q->size = size;

    q->f = -1;

    q->r = -1;

    q->arr = (int\*)malloc(size \* sizeof(int));

}

int isEmpty(CircularQueue \*q) {

    return q->f == -1;

}

int isFull(CircularQueue \*q) {

    // Full when the next position of rear is the front

    return ((q->r + 1 == q->size && q->f == 0) || (q->r + 1 == q->f));

}

void enqueue(CircularQueue \*q, int val) {

    if (isFull(q)) {

        printf("Queue is full\n");

    } else {

        if (isEmpty(q)) {

            q->f = 0;

        }

        if (q->r == q->size - 1) {

            q->r = 0;

        } else {

            q->r++;

        }

        q->arr[q->r] = val;

        printf("Enqueued element %d\n", val);

    }

}

int dequeue(CircularQueue \*q) {

    int a = -1;

    if (isEmpty(q)) {

        printf("Queue is empty\n");

    } else {

        a = q->arr[q->f];

        if (q->f == q->r) {

            q->f = -1;

            q->r = -1;

        } else {

            if (q->f == q->size - 1) {

                q->f = 0;

            } else {

                q->f++;

            }

        }

    }

    return a;

}

int peek(CircularQueue \*q) {

    if (isEmpty(q)) {

        printf("Queue is empty\n");

        return -1;

    }

    return q->arr[q->f];

}

void display(CircularQueue \*q) {

    if (isEmpty(q)) {

        printf("Queue is empty\n");

        return;

    }

    printf("Queue elements: ");

    int i = q->f;

    while (1) {

        printf("%d ", q->arr[i]);

        if (i == q->r) break;

        if (i == q->size - 1) {

            i = 0;

        } else {

            i++;

        }

    }

    printf("\n");

}

int main() {

    CircularQueue q;

    int size = 5;

    initQueue(&q, size);

    enqueue(&q, 10);

    enqueue(&q, 20);

    enqueue(&q, 30);

    enqueue(&q, 40);

    enqueue(&q, 50);

    display(&q);

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

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

    display(&q);

    enqueue(&q, 60);

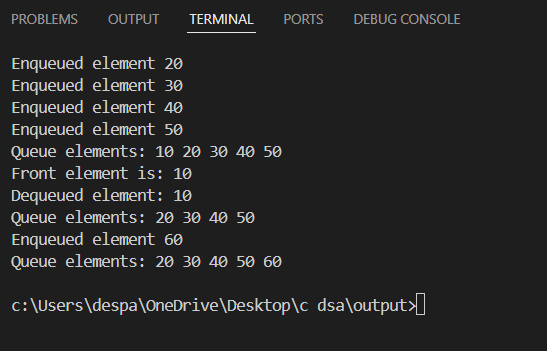
    display(&q);

    free(q.arr);

    return 0;

}

**OUTPUT**



**8.PRIORITY QUEUE**

#include<stdio.h>

#include<conio.h>

int pq\_full(void);

int pq\_empty(void);

void priority(void);

struct priorityQueue

{

    int a[5];

    int front,rear;

}dpq;

void main()

{

    int choice,item,temp;

    dpq.front=-1;

    dpq.rear=-1;

    do

    {

        printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

        printf("\n1.For Insertion\n2.For deletion\n3.For display\n4.exit");

        printf("\nEnter the choice : ");

        scanf("%d",&choice);

        switch(choice)

        {

        case 1:

        {

            if(pq\_full()==1)

            printf("\nQueue is full");

            else

            {

                printf("\nEnter the item to be inserted : ");

                scanf("%d",&item);

                if(dpq.rear==-1)

                {

                    dpq.front=0;

                    dpq.rear=0;

                    dpq.a[dpq.rear]=item;

                }

                else

                dpq.a[++dpq.rear]=item;

            }

            priority();

            break;

        }

        case 2:

        {

            if(pq\_empty()==1)

            printf("\nQueue is empty");

            else

            {

                if((dpq.rear==dpq.front)&&(dpq.rear!=-1))

                {

                    dpq.rear=-1;

                    dpq.front=-1;

                }

                else

                dpq.front++;

            }

            break;

        }

        case 3:

        {

            if(pq\_empty()==1)

            printf("\nqueue empty");

            else

            {

                //priority();

                temp=dpq.front;

                while(temp<=dpq.rear)

                {

                    printf("%d\t",dpq.a[temp]);

                    temp++;

                }

            }

        }

        break;

        }

    }while(choice<4);

}

int pq\_full()

{

    if(dpq.rear==4)

    return 1;

    else

    return 0;

}

int pq\_empty()

{

    if(dpq.rear==-1)

    return 1;

    else

    return 0;

}

void priority()

{

    int i,j,k,temp;

    for(i=1;i<5;i++)

    for(j=0;j<i;j++)

    {

        if(dpq.a[j]<dpq.a[i])

        {

            temp=dpq.a[j];

            dpq.a[j]=dpq.a[i];

            for(k=i;k>j;k--)//shifting to right

            dpq.a[k]=dpq.a[k-1];

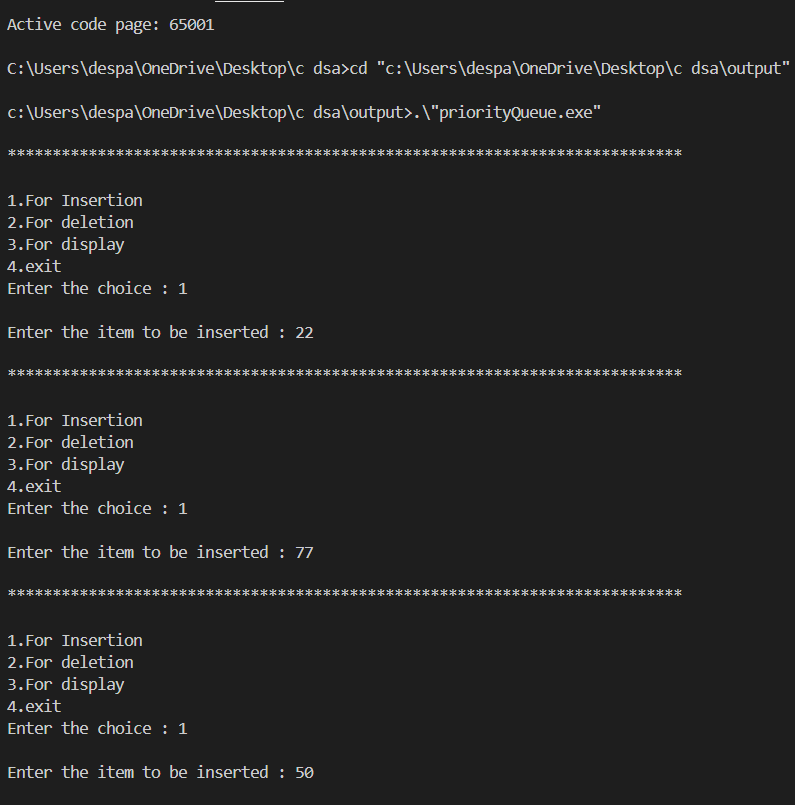
            dpq.a[k+1]=temp;

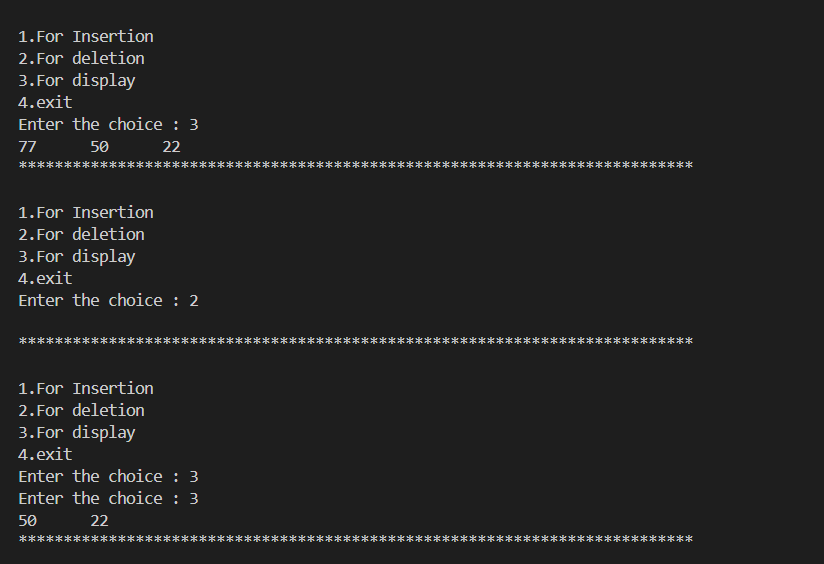
        }

    }

}

**OUTPUT**

****

****

**9.SINGLY LINKED LIST**

#include<stdio.h>

#include<stdlib.h>

struct node

    {

        int data;

        struct node \*next;

    };

struct node \*head=NULL, \*tail=NULL;

int len = 0;

void insert(){

    struct node \*newnode=(struct node\*)malloc(sizeof(struct node));

    printf("Enter data: ");

    scanf("%d",&newnode->data);

    newnode->next=NULL;

    len++;

    if (head==NULL)

        head = newnode;

    else

        tail->next=newnode;

    tail = newnode;

    display();

}

void insertAtBegin(){

    struct node \*newnode=(struct node\*)malloc(sizeof(struct node));

    printf("Enter data: ");

    scanf("%d",&newnode->data);

    newnode->next=head;

    len++;

    if(head==NULL)

        tail=newnode;

    head=newnode;

    display();

}

void insertAtEnd(){

    struct node \*newnode=(struct node\*)malloc(sizeof(struct node));

    printf("Enter data: ");

    scanf("%d",&newnode->data);

    newnode->next=NULL;

    len++;

    if(head==NULL)

        head = newnode;

    else

        tail->next=newnode;

    tail = newnode;

    display();

}

void display(){

    struct node \*temp = head;

    if(head==NULL)

        printf("EMPTY\n");

    else

    {

        while(temp!=NULL){

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

            temp=temp->next;

        }

        printf("\n\n");

    }

}

void insertAtPos(){

    int pos,i;

    printf("Enter position: ");

    scanf("%d",&pos);

    if(pos==1){

        insertAtBegin();

        return;

    }

    if(pos==len+1){

        insert();

        return;

    }

    if(pos > len || pos<1){

        printf("Invalid position\n\n");

        return;

    }

    struct node \*newnode=(struct node\*)malloc(sizeof(struct node));

    printf("Enter data: ");

    scanf("%d",&newnode->data);

    len++;

    struct node \*temp = head;

    while(i < pos-2){

        temp = temp->next;

        i++;

    }

    newnode->next = temp->next;

    temp->next=newnode;

    display();

}

void deleteAtBeginning(){

    if(head==NULL){

        printf("List already EMPTY\n");return;

    }

    struct node \*temp=head;

    head = head->next;  //logical deletion

    free(temp);     //physical deletion

    display();

    len--;

}

void deleteAtEnd(){

    struct node \*temp;

    // int i;

    if(head==NULL){

        printf("List already EMPTY\n");return;

    }

    temp=head;

    while(temp->next!=tail){

        temp=temp->next;

    }

    free(tail);

    tail=temp;

    tail->next=NULL;

    len--;

    display();

}

void deleteAtPos(){

    int pos,i=1;

    struct node \*temp;

    temp=head;

    printf("Enter position of element to be deleted: ");

    scanf("%d",&pos);

    if(pos==1){

        deleteAtBeginning();

        return;

    }

    if(pos==len){

        deleteAtEnd();

        return;

    }

    while(i<pos-1){

        temp=temp->next;

        i++;

    }

    struct node \*delNode=temp->next;

    temp->next=delNode->next;

    if(delNode->next==NULL)

        tail=temp;

    free(delNode);

    len--;

    display();

}

void main(){

    int choice;

    do

    {

        printf("1.INSERT\n2.INsert at begin\n3.insert at position\n4.display\n5 Delete Head\n6.DELETE AT END\n7.DELETE AT POSITION\nEnter your choice: ");

        scanf("%d",&choice);

        switch (choice)

        {

            case 1:insert();break;

            case 2:insertAtBegin();break;

            case 3:insertAtPos();break;

            case 4:display();break;

            case 5:deleteAtBeginning();break;

            case 6:deleteAtEnd();break;

            case 7:deleteAtPos();break;

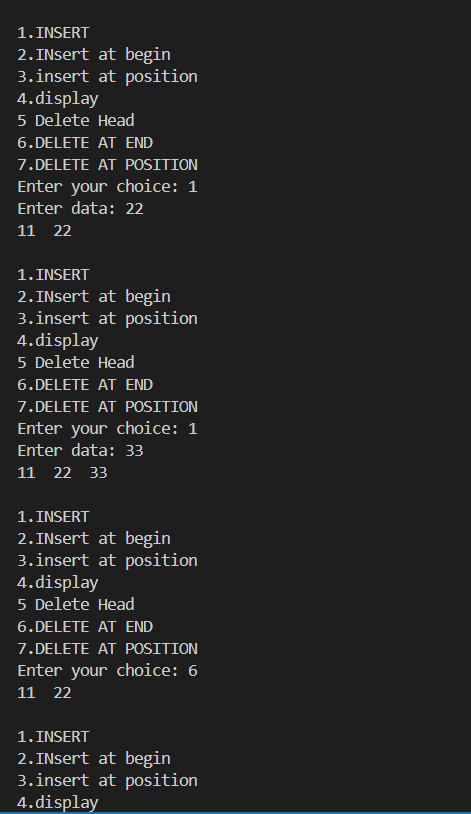
            default:printf("INVALID");break;

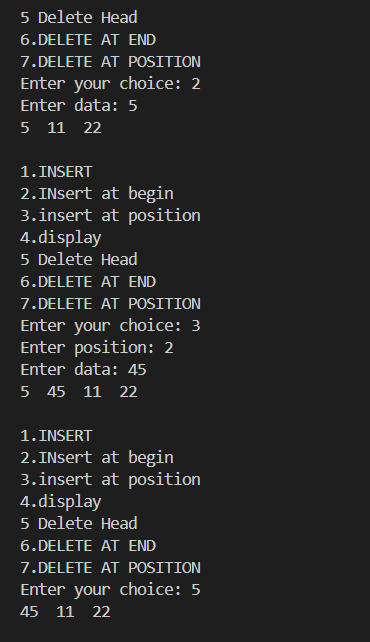
        }

    }while(choice!=0);

}

**OUTPUT**

****

****

**10.Doubly LINKED LIST all Operation**

#include <stdio.h>

#include <stdlib.h>

struct node {

    int data;

    struct node \*next;

    struct node \*prev;

};

struct node \*head = NULL, \*tail = NULL;

int len = 0;

void insert() {

    struct node \*newnode = (struct node \*)malloc(sizeof(struct node));

    printf("Enter data: ");

    scanf("%d", &newnode->data);

    newnode->next = NULL;

    newnode->prev = NULL;

    len++;

    if (head == NULL) {

        head = tail = newnode;

    } else {

        tail->next = newnode;

        newnode->prev = tail;

        tail = newnode;

    }

    display();

}

void insertAtBegin() {

    struct node \*newnode = (struct node \*)malloc(sizeof(struct node));

    printf("Enter data: ");

    scanf("%d", &newnode->data);

    newnode->next = head;

    newnode->prev = NULL;

    len++;

    if (head == NULL) {

        tail = newnode;

    } else {

        head->prev = newnode;

    }

    head = newnode;

    display();

}

void insertAtEnd() {

    insert();

}

void display() {

    struct node \*temp = head;

    if (head == NULL)

        printf("List is EMPTY\n");

    else {

        while (temp != NULL) {

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

            temp = temp->next;

        }

        printf("\n\n");

    }

}

void insertAtPos() {

    int pos, i = 1;

    printf("Enter position: ");

    scanf("%d", &pos);

    if (pos == 1) {

        insertAtBegin();

        return;

    }

    if (pos == len + 1) {

        insertAtEnd();

        return;

    }

    if (pos > len + 1 || pos < 1) {

        printf("Invalid position\n\n");

        return;

    }

    struct node \*newnode = (struct node \*)malloc(sizeof(struct node));

    printf("Enter data: ");

    scanf("%d", &newnode->data);

    len++;

    struct node \*temp = head;

    while (i < pos - 1) {  // Traverse to the node before the desired position

        temp = temp->next;

        i++;

    }

    newnode->next = temp->next;

    newnode->prev = temp;

    if (temp->next != NULL) {  // Update the previous pointer of the next node

        temp->next->prev = newnode;

    }

    temp->next = newnode;

    display();

}

void deleteAtBeginning() {

    if (head == NULL) {

        printf("List already EMPTY\n");

        return;

    }

    struct node \*temp = head;

    head = head->next;

    if (head != NULL) {

        head->prev = NULL;

    } else {

        tail = NULL;  // List becomes empty

    }

    free(temp);

    len--;

    display();

}

void deleteAtEnd() {

    if (head == NULL) {

        printf("List already EMPTY\n");

        return;

    }

    if (head->next == NULL) {

        free(head);

        head = tail = NULL;

        len--;

    } else {

        struct node \*temp = tail;

        tail = tail->prev;

        tail->next = NULL;

        free(temp);

        len--;

    }

    display();

}

void deleteAtPos() {

    int pos, i = 1;

    struct node \*temp = head;

    printf("Enter position of element to be deleted: ");

    scanf("%d", &pos);

    if (pos == 1) {

        deleteAtBeginning();

        return;

    }

    if (pos == len) {

        deleteAtEnd();

        return;

    }

    if (pos > len || pos < 1) {

        printf("Invalid position\n\n");

        return;

    }

    while (i < pos - 1) {  // Traverse to the node before the desired position

        temp = temp->next;

        i++;

    }

    struct node \*delNode = temp->next;

    temp->next = delNode->next;

    if (delNode->next != NULL) {  // Update the previous pointer of the next node

        delNode->next->prev = temp;

    }

    free(delNode);

    len--;

    display();

}

int main() {

    int choice;

    do {

        printf("1.INSERT\n2.Insert at begin\n3.Insert at position\n4.Display\n5.Delete Head\n6.DELETE AT END\n7.DELETE AT POSITION\nEnter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1: insert(); break;

            case 2: insertAtBegin(); break;

            case 3: insertAtPos(); break;

            case 4: display(); break;

            case 5: deleteAtBeginning(); break;

            case 6: deleteAtEnd(); break;

            case 7: deleteAtPos(); break;

            default: printf("INVALID\n"); break;

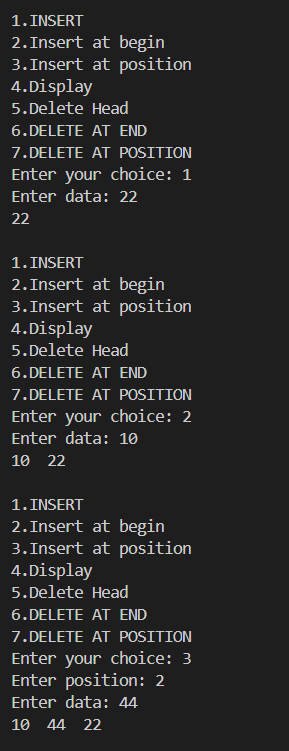
        }

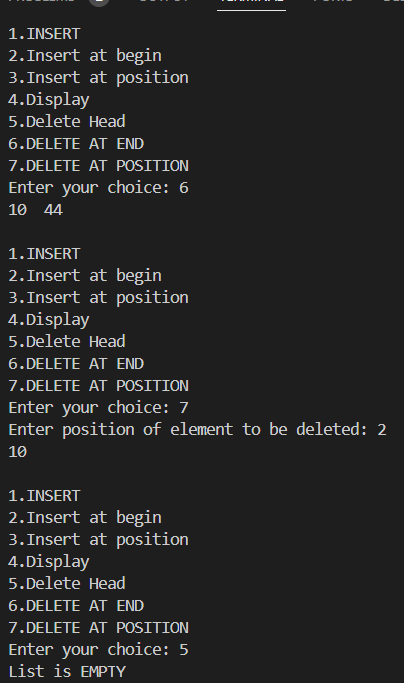
    } while (choice != 0);

    return 0;

}

**OUTPUT**

****

****