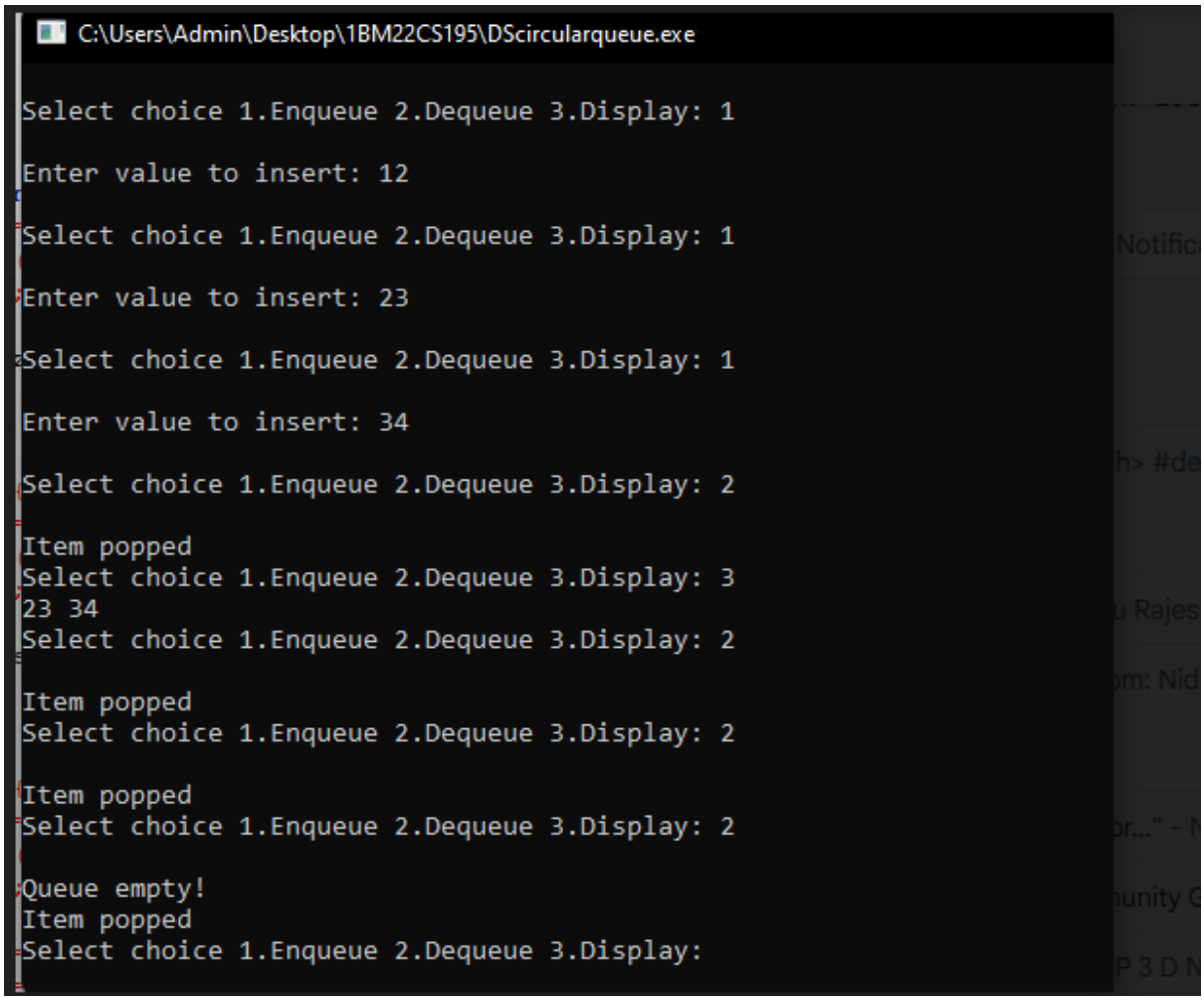
3b circular queue

Code:-

```
#include <stdio.h>
#include <stdlib.h>
#define size 5

int q[size], f = 0, r = -1;
int count = 0;

void enqueue(int item){
    if(count == size){
        printf("\nQueue full!");
        return;
    }
    q[(++r)%size] = item;
    count++;
}

void dequeue(){
    if(count == 0){
        printf("\nQueue empty!");
        return;
    }
    f = (f+1)%size;
    count--;
}

void display(){
    if(count == 0){
        printf("\nQueue empty!");
        return;
    }
    int front = f;
    for(int i = 0; i<count; i++){
        printf("%d ",q[front]);
        front = (front+1)%size;
    }
}

int main(){
    int ch, item;
    while(1){
        printf("\nSelect choice 1.Enqueue 2.Dequeue 3.Display: ");
        scanf("%d",&ch);

        switch(ch){
        case 1:
            printf("\nEnter value to insert: ");
```

```
            scanf("%d",&item);
            enqueue(item);
            break;
        case 2:
            dequeue();
            printf("\nItem popped");
            break;
        case 3:
            display();
            break;
            default:
        exit(0);
    }
    }
}
```

Output:-



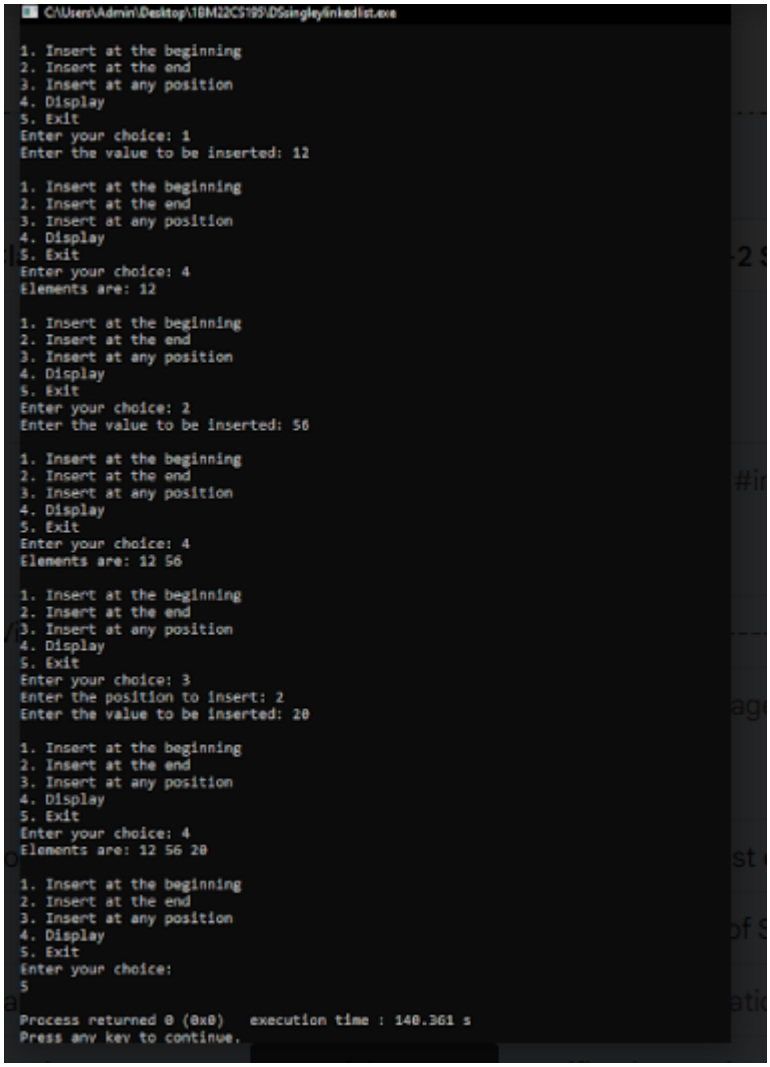

4

Code:-

```
#include<stdio.h>
#include<stdlib.h>
struct node {
    int data;
    struct node *next;
};
struct node *head = NULL;
void display() {
    struct node *ptr = head;
    if (ptr == NULL) {
        printf("List is empty\n");
        return;
    }
    printf("Elements are: ");
    while (ptr != NULL) {
        printf("%d ", ptr->data);
        ptr = ptr->next;
    }
    printf("\n");
}
void insert_begin() {
    struct node *temp;
    temp = (struct node *)malloc(sizeof(struct node));
    printf("Enter the value to be inserted: ");
    scanf("%d", &temp->data);
    temp->next = head;
    head = temp;
}
void insert_end() {
    struct node *temp, *ptr;
    temp = (struct node *)malloc(sizeof(struct node));
    printf("Enter the value to be inserted: ");
    scanf("%d", &temp->data);
    temp->next = NULL;
    if (head == NULL) {
        head = temp;
    } else {
        ptr = head;
        while (ptr->next != NULL) {
            ptr = ptr->next;
        }
        ptr->next = temp;
    }
}
void insert_pos() {
    int pos, i;
    struct node *temp, *ptr;
    temp = (struct node *)malloc(sizeof(struct node));
    printf("Enter the position to insert: ");
    scanf("%d", &pos);
    printf("Enter the value to be inserted: ");
    scanf("%d", &temp->data);
    temp->next = NULL;
```

```
    if (pos == 0) {
        temp->next = head;
        head = temp;
    } else {
        ptr = head;
        for (i = 0; i < pos - 1; i++) {
            ptr = ptr->next;
            if (ptr == NULL) {
                printf("Position not found\n");
                return;
            }
        }
        temp->next = ptr->next;
        ptr->next = temp;
    }
}
int main() {
    int choice;
    while(1) {
        printf("\n1. Insert at the beginning\n2. Insert at the end\n3. Insert at any position\n4.
Display\n5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                insert_begin();
                break;
            case 2:
                insert_end();
                break;
            case 3:
                insert_pos();
                break;
            case 4:
                display();
                break;
            case 5:
                exit(0);
                break;
            default:
                printf("Enter the correct choice\n");
        }
    }
    return 0;
}
```

Output:-




Leet code:-1

Code:-

```c
#include <stdlib.h>


typedef struct {
int *stack;
int *minStack;
int top ;
} MinStack;


MinStack* minStackCreate() {
MinStack* stack = (MinStack*)malloc(sizeof(MinStack));
stack->stack = (int*)malloc(sizeof(int) * 50);
stack->minStack = (int*)malloc(sizeof(int) * 50);
stack->top = -1;
return stack;
}
```

```
void minStackPush(MinStack* obj, int val) {
obj->top++;
obj->stack[obj->top] = val;
if (obj->top == 0 || val <= obj->minStack[obj->top - 1]) {
obj->minStack[obj->top] = val;
} else {
obj->minStack[obj->top] = obj->minStack[obj->top - 1];
}
}

void minStackPop(MinStack* obj) {
obj->top--;
}

int minStackTop(MinStack* obj) {
return obj->stack[obj->top];
}

int minStackGetMin(MinStack* obj) {
return obj->minStack[obj->top];
}

void minStackFree(MinStack* obj) {
free(obj->stack);
free(obj->minStack);
free(obj);
}
```

Output:-

Testcase | Test Result

**Accepted** Runtime: 0 ms

• Case 1

Input

```
["MinStack","push","push","push","getMin","pop","top","getMin"]
```

```
[[],[-2],[0],[-3],[],[],[],[]]
```

Output

```
[null,null,null,null,-3,null,0,-2]
```

Expected

```
[null,null,null,null,-3,null,0,-2]
```