**TASK 4: Queue ADT**

**PROBLEM 1:**

To implement a [queue](http://www.geeksforgeeks.org/queue-data-structure/) using array, create an array arr of size n and take two variables front and rear both of which will be initialized to 0 which means the queue is currently empty. Element rear is the index upto which the elements are stored in the array and front is the index of the first element of the array. Now, some of the implementation of queue operations are as follows:

* Enqueue:
* Dequeue
* Front
* Display

**Testcase 1:**

**Sample Input and Output:**

Queue is Empty

10 <-- 20 <-- 30 <-- 40 <--

Queue is full

10 <-- 20 <-- 30 <-- 40 <--

after two node deletion

30 <-- 40 <--

Front Element is: 30

**Program 4a:**

#include<stdio.h>

#include<stdlib.h>

#define MAXSIZE 100

struct Queue {

int front, rear, capacity;

int\* queue;

Queue(int c)

{

front = rear = 0;

capacity = c;

queue = new int;

}

// function to insert an element

// at the rear of the queue

int queueEnqueue()

{

int data;

// check queue is full or not

if (capacity == rear) {

printf("\nQueue is full\n");

return 0;

}

// insert element at the rear

else {

printf ("\nEnter element to be enqueued: ");

scanf ("%d", &data);

queue[rear] = data;

rear++;

}

return 0;

}

// function to delete an element

// from the front of the queue

int queueDequeue()

{

int num;

if (front == -1) {

printf("\nQueue is empty\n");

}

// shift all the elements from index 2 till rear

// to the left by one

else {

num = queue[front];

printf("Dequeued Element : %d \n",num);

front++;

}

return 0;

}

// print queue elements

void queueDisplay()

{

int i;

if (front == rear) {

printf("\nQueue is Empty\n");

return;

}

// traverse front to rear and print elements

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

printf(" %d <-- ", queue[i]);

}

return;

}

// print front of queue

void queueFront()

{

if (front == rear) {

printf("\nQueue is Empty\n");

return;

}

printf("\nFront Element is: %d", queue[front]);

return;

}

};

// Driver code

int main()

{

int ch;

Queue q(5);

printf ("\t Queue OPERATIONS\n");

printf("----------------------------\n");

printf(" 1. Enqueue\n");

printf(" 2. Dequeue\n");

printf(" 3. Display\n");

printf(" 4. Front\n");

printf(" 5. Exit\n");

//printf("----------------------------\n");

while(1)

{

printf("\nChoose operation : ");

scanf("%d", &ch);

switch (ch)

{

case 1:

q.queueEnqueue();

break;

case 2:

q.queueDequeue();

break;

case 3:

q.queueDisplay();

break;

case 4:

q.queueFront();

break;

case 5:

exit(0);

default:

printf("Invalid operation \n");

return 0; }

}

return 0;

}

**PROBLEM 2:**

A queue is an abstract data type that maintains the order in which elements were added to it, allowing the oldest elements to be removed from the front and new elements to be added to the rear. This is called a First-In-First-Out (FIFO) data structure because the first element added to the queue (i.e., the one that has been waiting the longest) is always the first one to be removed.

A basic queue has the following operations:

**Enqueue:** add a new element to the end of the queue.

**Dequeue:** remove the element from the front of the queue and return it.

In this challenge, you must first implement a queue using two stacks. Then process q queries, where each query is one of the following 3 types:

1 x: Enqueue element into the end of the queue.

2: Dequeue the element at the front of the queue.

3: Print the element at the front of the queue.

**Input Format**

The first line contains a single integer, q, the number of queries.

Each of the next q lines contains a single query in the form described in the problem statement above. All queries start with an integer denoting the query type, but only query 1 is followed by an additional space-separated value, x, denoting the value to be enqueued.

**Constraints**

1 <= q <= 10^5

1 <= type <= 3

1 <= | x | <= 10^9

It is guaranteed that a valid answer always exists for each query of types 2 and 3.

**Output Format**

For each query of type 3, return the value of the element at the front of the fifo queue on a new line.

**Testcase 1:**

**Sample Input**

STDIN Function

----- --------

10 q = 10 (number of queries)

1 42 1st query, enqueue 42

2 dequeue front element

1 14 enqueue 42

3 print the front element

1 28 enqueue 28

3 print the front element

1 60 enqueue 60

1 78 enqueue 78

2 dequeue front element

2 dequeue front element

**Sample Output**

14

14

**Program 4b:**

#include<stdio.h>

#include<malloc.h>

struct node{

int data;

struct node\* next;

};

struct queue{

struct node\* head1;

struct node\* head2;

};

struct node\* push(struct node\* temp,int data){

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

newnode->data = data;

newnode->next = temp;

temp = newnode;

return temp;

}

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

q->head1 = push(q->head1,data);

}

int pop(struct node\*\* temp){

int data = ( \*temp )->data;

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

return data;

}

void dequeue(struct queue\*q){

int x;

if(q->head2 == NULL){

while(q->head1){

x = pop( &(q->head1) );

q->head2 = push(q->head2,x);

}

x = pop(&(q->head2));

}else{

x = pop(&(q->head2));

}

}

void print(struct queue\* q){

int x;

if(q->head2){

}else{

while(q->head1){

x = pop((&q->head1));

q->head2 = push(q->head2,x);

}

}

x = q->head2->data;

printf("%d\n",x);

}

int main(void){

int data,n,choice;

struct queue\* q = (struct queue\*)malloc(sizeof(struct queue));

q->head1 = q->head2 = NULL;

printf("Enter n=");

scanf("%d",&n);

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

printf("\n Enter Your choice");

scanf("%d",&choice);

if(choice == 1){

scanf("%d",&data);

enqueue(q,data);

}else if(choice == 2){

dequeue(q);

}else if(choice == 3){

print(q);

}

}

}

**PROBLEM 3:**

Design your implementation of the circular queue. The circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called "Ring Buffer".

One of the benefits of the circular queue is that we can make use of the spaces in front of the queue. In a normal queue, once the queue becomes full, we cannot insert the next element even if there is a space in front of the queue. But using the circular queue, we can use the space to store new values.

Implementation the MyCircularQueue class:

* MyCircularQueue(k) Initializes the object with the size of the queue to be k.
* int Front() Gets the front item from the queue. If the queue is empty, return -1.
* int Rear() Gets the last item from the queue. If the queue is empty, return -1.
* boolean enQueue(int value) Inserts an element into the circular queue. Return true if the operation is successful.
* boolean deQueue() Deletes an element from the circular queue. Return true if the operation is successful.
* boolean isEmpty() Checks whether the circular queue is empty or not.
* boolean isFull() Checks whether the circular queue is full or not.

**Testcase1:**

**Sample Input**

["MyCircularQueue", "enQueue", "enQueue", "enQueue", "enQueue", "Rear", "isFull", "deQueue", "enQueue", "Rear"]

[[3], [1], [2], [3], [4], [], [], [], [4], []]

**Sample Output**

[null, true, true, true, false, 3, true, true, true, 4]

**Explanation**

MyCircularQueue myCircularQueue = new MyCircularQueue(3);

myCircularQueue.enQueue(1); // return True

myCircularQueue.enQueue(2); // return True

myCircularQueue.enQueue(3); // return True

myCircularQueue.enQueue(4); // return False

myCircularQueue.Rear(); // return 3

myCircularQueue.isFull(); // return True

myCircularQueue.deQueue(); // return True

myCircularQueue.enQueue(4); // return True

myCircularQueue.Rear(); // return 4

**Constraints:**

* 1 <= k <= 1000
* 0 <= value <= 1000
* At most 3000 calls will be made to enQueue, deQueue, Front, Rear, isEmpty, and isFull.

**Program 4c:**

#include <stdio.h>

#include<conio.h>

#include<stdlib.h>

# define max 6

int queue[max]; // array declaration

int front=-1;

int rear=-1;

// function to insert an element in a circular queue

void enqueue(int element)

{

if(front==-1 && rear==-1) // condition to check queue is empty

{

front=0;

rear=0;

queue[rear]=element;

}

else if((rear+1)%max==front) // condition to check queue is full

{

printf("Queue is overflow..");

}

else

{

rear=(rear+1)%max; // rear is incremented

queue[rear]=element; // assigning a value to the queue at the rear position.

}

}

// function to delete the element from the queue

void dequeue()

{

if((front==-1) && (rear==-1)) // condition to check queue is empty

{

printf("\nQueue is underflow..");

}

else if(front==rear)

{

printf("\nThe dequeued element is %d", queue[front]);

front=-1;

rear=-1;

}

else

{

printf("\nThe dequeued element is %d", queue[front]);

front=(front+1)%max;

}

}

// function to display the elements of a queue

void display()

{

int i=front;

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

{

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

}

else

{

printf("\nElements in a Queue are :");

while(i<=rear)

{

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

i=(i+1)%max;

}

}

}

int main()

{

int choice,x; // variables declaration

printf("\*\*\*\*Circular Queue\*\*\*\*\*");

printf("\n Press 1: Enqueue an element");

printf("\nPress 2: Dequeue an element");

printf("\nPress 3: Display the element");

printf("\n Press 4: Exit");

while(1) // while loop

{

printf("\nEnter your choice");

scanf("%d", &choice);

switch(choice)

{

case 1:

printf("Enter the element which is to be inserted");

scanf("%d", &x);

enqueue(x);

break;

case 2:

dequeue();

break;

case 3:

display();

break;

case 4:

exit(0);

default:

printf("Invalid operation \n");

}

}

return 0;

}