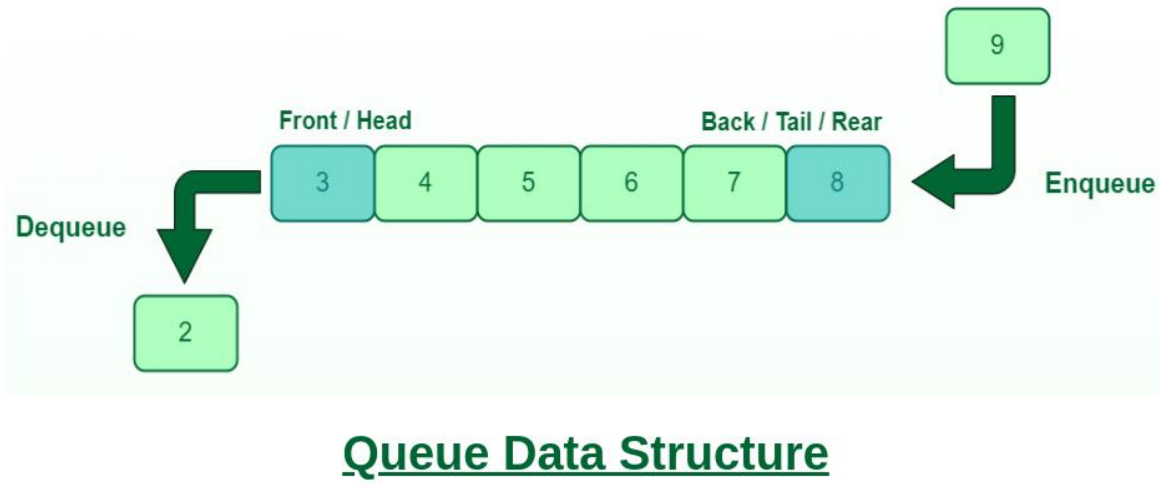
What is Queue Data Structure?

A **Queue** is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.



- **Queue:** the name of the array storing queue elements.
- **Front**: the index where the first element is stored in the array representing the queue.
- **Rear:** the index where the last element is stored in an array representing the queue.

Queue Implementation Using Array:

Declare a Class Queue:

A class named Queue is declared to encapsulate the circular queue implementation.

Class Data Members:

The Queue class contains the following data members:

front: An integer representing the front position of the queue.

rear: An integer representing the rear position of the queue.

size: An integer representing the maximum size of the queue.

arr: A pointer to an integer array for storing queue elements.

noofelements: An integer tracking the number of elements in the queue.

Constructor:

The constructor Queue(int size) initializes the queue with a given size. It sets front to 0, rear to -1, and noofelements to 0. It also dynamically allocates memory for the integer array arr of the specified size.

Destructor:

The destructor ~Queue() is responsible for releasing the dynamically allocated memory for the arr array.

isEmpty() Function:

bool isEmpty() checks whether the queue is empty by comparing noofelements with 0. It returns true if the queue is empty and false otherwise.

isFull() Function:

bool isFull() checks whether the queue is full by comparing noofelements with the maximum size. It returns true if the queue is full and false otherwise.

enqueue(int val) Function:

void enqueue(int val) adds an element val to the rear of the queue, if it is not full.

If the rear index reaches the end (size - 1), it wraps around to the beginning. It then increments rear, stores val at the new rear position, and increments noofelements.

dequeue() Function:

void dequeue() removes the front element from the queue, if it is not empty.

Similar to enqueue(), if the front index reaches the end of the array, it wraps around to the beginning. It then increments front and decrements noofelements.

getFront() Function:

void getFront() retrieves and displays the front element of the queue if the queue is not empty. Otherwise, it indicates that the queue is empty.

display() Function:

void display() prints the elements of the queue in their order if the queue is not empty.

It uses a loop that starts at the front index and wraps around in a circular manner to print each element.

```
#include<iostream>
using namespace std;
class Queue {
	public:
		int front;
		int rear;
		int size;
		int *arr;
		int noofelements;
		Queue(int size) {
			front=0;
			rear=-1;
			noofelements=0;
			this->size=size;
			arr=new int[size];
		}
		~Queue()
		{
			delete []arr;
		}
		bool isEmpty() {
			if(noofelements==0) {
				return true;
			} else {
				return false;
			}
		}
```

```
bool isFull() {
    if(noofelements==size) {
        return true;
    } else {
        return false;
    }
}
void enqueue(int val) {
    if(isFull()) {
        cout<<"Queue is full"<<endl;
        return;
    }
    else
    {
        if(rear==size-1)
        {
            rear=0;
        }
        else
        {
            rear++;
        }
        arr[rear]=val;
        noofelements++;
    }
}
```

```
void dequeue() {
    if(isEmpty()) {
        return;
    }
    else
    {
        if(front==size-1)
        {
            front=0;
        }
        else
        {
            front++;
        }
        noofelements--;
    }
}
void getFront() {
    if(!isEmpty()) {
        cout<<"Front is: "<<arr[front]<<endl;
    } else {
        cout<<"Queue is empty"<<endl;
    }
}
```

```
                void display() {
                        if(!isEmpty()) {
                                int index=front;
                                for(int i=1; i<=noofelements; i++)
                                {
                                        cout<<arr[index]<<"  ";
                                        index=(index+1)%size;
                                }
                                cout<<endl;
                        }
                        else
                        {
                                cout<<"queue is empty"<<endl;
                        }
                }
};
int main() {
        Queue q(5);
        q.enqueue(1);
        q.enqueue(2);
        q.enqueue(3);
        q.enqueue(4);
        q.enqueue(5);
        q.display();
        q.dequeue();
        q.dequeue();
        q.dequeue();
        q.display();
        q.enqueue(6);
        q.enqueue(7);
        q.display();
}
```

Output:

```
1 2 3 4 5

4 5

4 5 6 7
```

Implement the queue using linked list your self.

Lab Task:

Necklace

- Implement using Linked List

Your best friend has a very interesting necklace with nn pearls. On each of the pearls of the necklace there is an integer. However, your friend wants to modify the necklace a bit and asks you for help. She wants to move the first pearl kk spots to the left (and do so with all other pearls).

For example: if the necklace was originally 1,5,3,4,21,5,3,4,2 and k=2k=2, now it becomes 3,4,2,1,53,4,2,1,5.

Help your best friend determine how the necklace will look after the modification.

Input Format

- First line will contain TT, the number of test cases. Then the test cases follow.
- Each test case contains two lines of input, the first containing two integers n,kn,k.
- The second line of each test case contains nn integers a1,a2,...,an$a$1,a2,...,an representing the integers on the pearls starting from the first one.

Output Format

For each testcase, output in a single line nn integers representing the necklace after modification.

Constraints

- 1≤T≤1001≤T≤100
- 1≤n≤1051≤n≤105
- The sum of nn over all test cases does not exceed 3·1053·105
- 0≤k≤n0≤k≤n
- −109≤ai≤109−109≤ai≤109

Subtasks

- 30 points : The sum of nn over all test cases does not exceed 50005000
- 70 points : original constraints

Sample 1:

Input

2

5 3

1 5 3 4 2

6 5

10 1 2 9 8 2

Output

4 2 1 5 3

2 10 1 2 9 8

Explanation:

The first test case is the example from the statement. In the second test case, when we move every element 5 to the left we get the answer.