LinkedListQueue Inputs:

```
  LinkedListQueue ll;
   cout << "ll.peek(): " << ll.peek() << endl;
   cout << "ll.size(): " << ll.size() << endl;
   ll.dequeue();
   ll.enqueue(1);
   ll.enqueue(2);
   ll.enqueue(3);
   ll.enqueue(4);
   ll.enqueue(5);
   ll.display();
   ll.dequeue();
   ll.display();
   ll.enqueue(6);
   ll.display();
   cout << "ll.peek(): " << ll.peek() << endl;
   cout << "ll.size(): " << ll.size() << endl;
```

LinkedListQueue Outputs:

```
ll.peek(): -1
ll.size(): 0
Queue is Empty
1 2 3 4 5
2 3 4 5
2 3 4 5 6
ll.peek(): 2
ll.size(): 5
```

CircArrayQueue Inputs:

```
    CircArrayQueue ca;
    cout << "ca.peek(): " << ca.peek() << endl;
    cout << "ca.size(): " << ca.size() << endl;
    ca.dequeue();
    ca.enqueue(1);
    ca.enqueue(2);
    ca.enqueue(3);
    ca.enqueue(4);
    ca.enqueue(5);
    ca.display();
    ca.dequeue();
    ca.display();
    ca.enqueue(6);
    ca.display();
    cout << "ca.peek(): " << ca.peek() << endl;
    cout << "ca.size(): " << ca.size() << endl;
```

CircArrayQueue Outputs:

```
ca.peek(): -1
ca.size(): 0
Queue is Empty
1 2 3 4 5
2 3 4 5
2 3 4 5 6
ca.peek(): 2
ca.size(): 5
```

I chose the above inputs for my code to check edge cases such as peeking and dequeuing and empty queue. The rest of the inputs were numbered in increasing order so that I could see if the order was maintained throughout the following enqueue and dequeue operations, which proved robustness of those two functions. I also called the size and peek functions in all possible scenarios in which they could have an error or incorrect value. The outputs of the test cases I created matched perfectly between the linked list and circular array implementations of a queue, showing that they were both implemented correctly.