QueueDriver:

```
import java.util.Scanner;

public class QueueDriver {

	public static void main(String[] args) {
		Scanner scan = new Scanner(System.in);
		CustomQueue list = new CustomQueue();
		System.out.println("My Queue");

		while(true) {
			System.out.println("\nQueue Options\n");
			System.out.println("1. Add to queue");
			System.out.println("2. Take from queue");
			System.out.println("3. Peek at queue");
			System.out.println("4. check if empty");
			System.out.println("5. check if full");
			System.out.println("6. get size");
			System.out.println("7. exit");
			int choice = scan.nextInt();
			switch (choice) {
			case 1:
				System.out.println("Enter integer element to insert");
				list.enQueue(scan.nextInt());
				break;
			case 2:
				if (list.getSize() == 0) {
					System.out.println("Nothing to remove from queue!");
				} else {
					System.out.println("Data just removed from queue: " +
list.deQueue());
				}
				break;
			case 3:
				if (list.getSize() == 0) {
					System.out.println("Nothing to see here!");
				} else {
```

```
                        System.out.println("Data currently at start of queue: " +
list.peek());
                        }
                        break;
                    case 4:
                        System.out.println("Empty status = "+ list.isEmpty());
                        break;
                    case 5:
                        System.out.println("Full status = " + list.isFull());
                        break;
                    case 6:
                        System.out.println("Size = " + list.getSize() + "\n");
                        break;
                    case 7:
                        scan.close();
                        System.exit(0);
                    default:
                        System.out.println("Wrong Entry \n");
                        break;
                    }
                    list.display();
                }
        }
}
```

Node:

```
public class Node {
        private int data;
        private Node next;

        public Node() {
                next = null;
                data = 0;
        }

        public Node(int d, Node n) {
                data = d;
```

```
		next = n;
	}

	public void setNext(Node n) {
		next = n;
	}

	public void setData(int d) {
		data = d;
	}

	public Node getNext() {
		return next;
	}

	public int getData() {
		return data;
	}
}
```

CustomLinkedList:

```
public class CustomLinkedList {
	private Node start;
	private Node end;
	private int size;

	public CustomLinkedList() {
		start = null;
		end = null;
		size = 0;
	}

	public boolean isEmpty() {
		if (start == null) {
			return true;
		} else {
```

```
                return false;
            }
    }

    public int getSize() {
            return size;
    }

    public void insertAtStart(int val) {
            Node node = new Node(val, null);
            size++;
            if (start == null) {
                    start = node;
                    end = start;
            } else {
                    node.setNext(start);
            }
    }

    public void insertAtEnd(int val) {
            Node node = new Node(val, null);
            size++;
            if (start == null) {
                    start = node;
                    end = start;
            } else {
                    end.setNext(node);
                    end = node;
            }
    }

    public void insertAtPos(int val, int pos) {
            Node node = new Node (val, null);
            Node aNode = start;
            pos = pos - 1;
            for (int i = 1; i < size; i++) {
                    if (i == pos) {
                            Node tmp = aNode.getNext();
                            aNode.setNext(node);
                    node.setNext(tmp);
                    break;
                }
                aNode = aNode.getNext();
            }
            size++;
        }

        public void deleteAtPos(int pos) {
            if (pos == 1) {
                start = start.getNext();
                size--;
                return;
            }
            if (pos == size) {
                Node s = start;
                Node t = start;
                while (s!= end) {
                    t = s;
                    s = s.getNext();
                }
                end = t;
                end.setNext(null);
                size--;
                return;
            }
            Node aNode = start;
            pos = pos - 1;
            for (int i = 1; i < size - 1; i++) {
                if (i == pos) {
                    Node tmp = aNode.getNext();
                    tmp = tmp.getNext();
                    aNode.setNext(tmp);
                    break;
                }
                aNode = aNode.getNext();
            }
            size--;
        }
```

```
	public void display() {
		System.out.print(/*"\n Linked List = "*/"\n Queue = "); //cosmetic change for
consistency with the program.
		if (size == 0) {
			System.out.print("empty\n");
			return;
		}
		if (start.getNext() == null) {
			System.out.println(start.getData());
			return;
		}
		Node aNode = start;
		System.out.print(start.getData()+"->");
		aNode = aNode.getNext();
		while (aNode.getNext() != null) {
			System.out.print(aNode.getData()+"->");
			aNode = aNode.getNext();
		}
		System.out.print(aNode.getData()+"\n");
	}

	public int getStartData() {
		/*
		 * This method returns the data at the start position of the custom linked list.
		 *
		 * This method was added to the CustomLinkedList class because
		 * otherwise, outside classes couldn't access the data hidden inside
		 * without using the display method to display the full list.
		 */
		if (size == 0 || start == null) {
			return 0;
		} else {
			return start.getData();
		}
	}
}
```

CustomQueue:

```
public class CustomQueue {
	private CustomLinkedList list;

	public CustomQueue() {
		list = new CustomLinkedList();
	}

	public void enQueue(int val) {
		list.insertAtEnd(val);
	}

	public int deQueue() {
		if (list.getSize() > 0) {
			int value = list.getStartData();
			list.deleteAtPos(1);
			return value;
		} else {
			return Integer.MIN_VALUE;
		}
	}

	public int peek() {
		return list.getStartData();
	}


	public boolean isEmpty() {
		return list.isEmpty();
	}

	public boolean isFull() {
		return false;
	}

	public int getSize() {
		return list.getSize();
	}
	public void display() {
		list.display();
	}
}
```

Console's Output:

```
My Queue

Queue Options

1. Add to queue
2. Take from queue
3. Peek at queue
4. check if empty
5. check if full
6. get size
7. exit
1
Enter integer element to insert
10

 Queue = 10

Queue Options

1. Add to queue
2. Take from queue
3. Peek at queue
4. check if empty
5. check if full
6. get size
7. exit
1
Enter integer element to insert
20
```

```
 Queue = 10->20

Queue Options

1. Add to queue
2. Take from queue
3. Peek at queue
4. check if empty
5. check if full
6. get size
7. exit
1
Enter integer element to insert
30

 Queue = 10->20->30

Queue Options

1. Add to queue
2. Take from queue
3. Peek at queue
4. check if empty
5. check if full
6. get size
7. exit
2
Data just removed from queue: 10

 Queue = 20->30

Queue Options

1. Add to queue
2. Take from queue
3. Peek at queue
4. check if empty
5. check if full
6. get size
7. exit
2
Data just removed from queue: 20
```

```
 Queue = 30

Queue Options

1. Add to queue
2. Take from queue
3. Peek at queue
4. check if empty
5. check if full
6. get size
7. exit
2
Data just removed from queue: 30

 Queue = empty

Queue Options

1. Add to queue
2. Take from queue
3. Peek at queue
4. check if empty
5. check if full
6. get size
7. exit
2
Nothing to remove from queue!

 Queue = empty

Queue Options

1. Add to queue
2. Take from queue
3. Peek at queue
4. check if empty
5. check if full
6. get size
7. exit
7
```

Our assignment was to design a Queue, using previously given code for custom Linked Lists and Nodes. The point of a queue is to have a first in, first out system where values are handled in the order in which they are inserted. The queue needed a method for adding items to the end of the queue, viewing or removing an item at the front of the queue, and a way to check whether or not the queue was empty or full to decide if those operations would be able to execute without errors. We also needed a method to display the queue in full.

The custom Linked List object already provided means for many of these functions, so my Queue class holds one of its own. The queue then just uses the linked list's isEmpty, getSize, and display functions as its own. I changed the text output by CustomLinkedList.display() to better match its use with a queue. The queue theoretically has no size cap, so the getFull() method just returns false. Adding to the queue only required adding the value to the end of the list, so enqueue(int val) merely takes its value and puts it through the CustomLinkedList's insertAtEnd() method. Removing (dequeue) or looking at (peek) an item from the queue, however, required a change in the given program. CustomLinkedList had no method that returned the value of just one of the nodes, so I added one to solve the issue, and now other classes can read the starting node's value. From there, peek() only required one call to this new function, and dequeue() only needed to save and return that function's value before it used CustomLinkedList.deleteAtPos(1) to remove it from the queue. With that taken care of, the queue performs as requested!