Background

1.1 Linear Data Structures
There are a variety of linear structures that one may use to store data elements in. Some examples include vectors, linked lists, arrays, stacks, and queues. Some of these structures are specialized in terms of the order in which they process elements. The choice of which structure is the most appropriate to use depends on your application.
There are various trade-offs between linear structures. Dynamic arrays, for example, allow quick direct access to elements, but managing their sizes causes extra overhead. Linked lists address the size issues of arrays, but they are slower when accessing arbitrary elements.
Other linear structures do not allow for access to arbitrary elements, and they maintain a certain order amongst the elements themselves. In stacks and queues, for example, the next element to which access is allowed is determined by the order in which elements were inserted into these structures.

1.2 Implementing Linear Data Structures
Linear structures may be implemented in a number of ways. Linked lists, for example, can be implemented as singly linked or doubly linked, circular or non-circular. Stacks may be implemented using arrays or linked list-like structures. As discussed previously, each type of implementation comes with its own advantages, and which implementation is most suited depends on the situation.
One can easily cater for every type of implementation that one may possibly need, and do it in such a way that it is easy and convenient to add additional implementations at a later stage should it be required, without changing any code that you have already written. For example, you need not hard code two stack classes where one is implemented in terms of an array and one a linked list. Instead, you can code one stack class and instantiate it with a linked list or array implementation (provided these conform to the same interface). It might seem counterintuitive to code three classes (linked list, array, and stack) instead of just two (array stack and linked list stack). However, if you need a queue class later on, you could easily re-use your linked list or array class as an underlying structure. You can do the same if you decide to add a priority queue to your collection of structures. You then have 5 classes which you can mix and match with different implementations as opposed to 6 that you'll need to cater for all these combinations if they were hard-coded. If you add an additional underlying linear structure, such as a circular linked list, then you will be able to create 9 combinations of data stacks, queues and sorted queues with only 6 classes.

2 Your Task
Download the archive clean-files.tar.gz from the repo. This archive contains a number of classes that you will have to implement. The comments in the code describe how each function should be implemented. You are **not allowed** to modify the header files. You are provided with the following classes:

2.1 LinearStructure
This class is an interface to which linear structure implementations conform. DynamicArray and LinkedList inherit from this class.

2.2 Node
A doubly-linked node class to be used for the linked list implementation.

2.3 DynamicArray
This class encapsulates a dynamic array. The array grows, as more space is needed. See dynamicArray.h for details.

2.4 LinkedList
A **circular doubly-linked list** class. See linkedList.h for details.

2.5 OrderedContainer
This class provides an interface for all types of ordered containers. An ordered container may be implemented in a variety of ways. As an example, a stack may be implemented as either a linked list or an array. The ordered container is given an implementation object (either a linked list or a dynamic array), and data elements are stored in the specified structure. The subclasses (stack, queue, etc.) will have to manipulate their underlying implementations to produce the correct results.

2.6 Stack
A stack is a linear structure that does not allow access to arbitrary elements stored in it. The top of the stack is the only entry point of a stack structure, and when an element is added to a stack, it must be stored on top of the stack. When an element is read from the stack, only the top element can be accessed. Removal of elements can also only be applied to the top element.
Because of this property, stacks are known as LIFO (last-in-first-out) structures.

2.7 Queue

A queue is also a linear structure that does not allow access to arbitrary elements stored in it.
A queue has two entry points: front and rear. When an element is added, it is always added at the rear end. When an element is read or removed, it is always read/removed from the front of the queue. Queues are also known as FIFO (first-in-first-out) structures, which means that elements are removed and returned from a queue in the order in which they were inserted.

2.8 PriorityQueue

PriorityQueue inherits from Queue. Priority queues recognize the fact that some elements might be more important than others, and return elements with the highest importance first, regardless of the order in which the elements were inserted. For this assignment, you will have to implement your priority queue such that there is a choice between whether the smallest element or the largest element in the queue will be returned. See the comments in the code for more details.

3 Implementation Details

You are not allowed to change the header files. You will notice that the linear structures have a very limited interface. You will have to think carefully on how you can use this interface to manipulate the structures appropriately in your container classes.
A note on templates: You will notice that the source files (.cpp) are included in the provided header files. This is one way to make linking easier with templates. Your source files must not include the headers – remember that template classes are only compiled for concrete types. To test your template classes, all you need to include in your main is the corresponding header.