Homework #3

Question 1 (2 pt.)

A *deque*, or *double-ended queue*, is a circular data structure that allows for efficient insertions and deletions of objects at both its head and its tail. This is in contrast to the dynamically growing array, where an insertion or deletion at its head happens in linear time. Internally, a deque is implemented with an array and an index to its current head and tail. When an element is deleted from its head, the head pointer is just incremented by one in a circular manner, as opposed to having to shift all array elements on position to the left.

Implement a new class called `Deque` which provides the exact same public interface as class `Array`, presented in class. Write a main program with a behavior of your choice that calls all new functions at least once in order to verify their correct behavior, and show the program's output.

Upload a file on Blackboard called `q1.zip`, including files `Deque.h`, `Deque.cc`, and `main.cc`. The program should compile and run without errors on the CoE Linux machines.

Question 2 (2 pt.)

The heap's `MaxHeapify` function has been presented as a divide-and-conquer design, where the size of the sub-tree being considered becomes smaller for every recursive call. Looking at the implementation of this function, we can observe that the only recursive call occurs at the end of the function body, in a similar structure as the recursive implementation of `BinarySearch`. Since the current call context does not need to be saved before the recursive call, this function lends itself to a simple redesign using an iterative approach, and thus reducing its spatial cost.

Rewrite the code for function `Heap::MaxHeapify` avoiding recursiveness. The function must have the same interface and must provide the same behavior. Test your code with a `main()` function that sorts an array of integers using `Heap::HeapSort()`.

Upload your code in a single file named `q2.cc` containing all necessary functions for the program to compile and run correctly on the CoE machines.

Question 3 (2 pt.)

Extend class `Shape` with an implementation of function `Compare()`, as defined in the new version of parent class `Object`. The key to compare shape objects will be their area, as returned by the `GetArea()` virtual function, extended by each specific subclass of `Shape`. Write a new `main()` function that creates a heap, inserts 10 different kinds of shapes with different values for their attributes (rectangles, circles, and triangles), and sorts them using the *heapsort* algorithm. The program should print the array of shapes before and after it is sorted.

Submit a file named `q3.zip` that contains all header and source files required to compile and run this program on the CoE machines. This includes both files extended or implemented for this assignment, as well as files reused from other programs.

Question 4 (2 pt.)

Add a function in class `Heap` that deletes the object placed at position 0 (the object with the maximum value), and leaving the heap in a state where the heap property is respected for all remaining objects. The function should have the following prototype, returning the object that was extracted:

```
Object *Heap::ExtractMax();
```

Upload the implementation of this function in a file named `q4.cc`. The file does not need to contain any code other than the `ExtractMax()` function itself.

Question 5 (2 pt.)

Add a function in class `Heap` to insert an object. The operation should make sure that the heap property is restored after the insertion. The insertion strategy consists in initially placing the element at the end of the heap and then “floating” it up until it reaches its appropriate position, in an inverse process from what function `MaxHeapify()` does. The prototype of the function should be the following:

```
Heap::HeapInsert(Object *object);
```

Upload the implementation of this function in a file named `q5.cc`. The file does not need to contain any code other than the `HeapInsert()` function itself.