CSC220 Lab10
Heaps

The goal of this week's assignment is:

1. Practice using heaps
2. Learn about the importance of debugging

Part 0

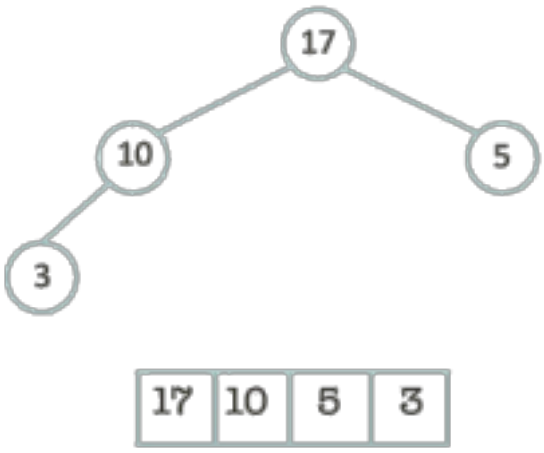
- Create a new Java project and call it **Lab10.**
- Create a package inside your project and call it **lab10**.
- Download the Lab10-Assignment10 ZIP folder from Blackboard and copy in the following files:
  - **MaxHeap.java**. You will be working on developing methods for this class. This class represents a max heap. This class has been partially implemented for you.
  - **HeapTester.java**. A set of test cases your code should handle.

Part 1 – Problem description

For this lab, you are asked to write methods to complete the implementation of a binary max heap class. In a binary max heap, each node can have at most two children, and the data in the **parent node must be greater than or equal to all of its descendants**. Remember that, like min heaps, there is no relationship between the data values of the siblings of a specific parent **(i.e., the right child does not have to be greater or smaller than the left child)**. The max heap in this case is nothing but an array. Recall from the lectures, for implementing a binary tree (or in this case a heap) using an array, we need the following rule: for any node at index **i**, its two children are at index **(i*2)+1** and **(i*2)+2**. The functions to access the parent, left child or right child of a node have been implemented for you.

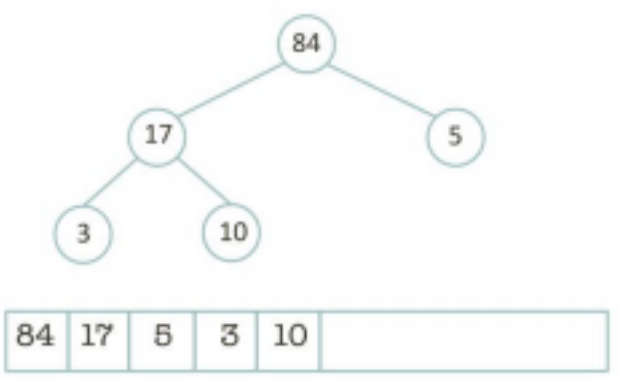
Given an array of N values, a heap containing those values can be built, in-place, by simply "sifting" each internal node down to its proper location. You have seen the

opposite behavior during the lecture for a min heap. For example, imagine that we want to add 84 to the following heap that is represented using the following array:



In order to offer 84 in this max heap, we start by putting the value 84 at the last internal node possible. Which one is it?!

Now, we need to make sure whether it is in its current position, meaning if the value of its parent is smaller than 84, we need to swap the current node (84) with its parent till the max heap property is satisfied. Before you continue, take a moment and think about how the array (representing the max heap) should look like after this offer. Here is how it should look like after the offer:



You need to convince yourself why the internal nodes have been moved around before you continue.

Similar to lab 9, We provided a simple variation of in-order traversal of a tree (suggested in the book – Chapter 17.2 – page 1031): “instead of printing values all on a line, we will print them one per line and use indentation to indicate the level of the tree”. For example, the previous heap can be printed using a call to **printSideways**():

```
                0
        0
                0
    5
                0
        0
                0
84
                0
        10
                0
    17
                0
        3
                0
```

The extra zero values are due to the fact that the max heap has been initialized so that it can contain 15 elements. You need to imagine rotating this output 90 degrees in a clockwise fashion (or tilting your head to the left) to see the tree structure. This function can come handy. There is also a **toString**() method that you can use to return a String representation of the heap.

Before you move on to the next part, take a couple of minutes and look at the definition of the methods provided for you and the fields of this class. **Consult the lab TAs if you have questions!**

Remember: you are **NOT** allowed to change the signature of the methods. However, you are more than welcome to create and use helper methods.

Finally, please note that you do not need to worry about the array containing the heap running out of space (keep this in mind, when you are testing your code – not to add more than the array can contain).

This week you are required to implement the following methods. We will only do the **first three** during the lab:

1. `public MaxHeap(int maxsize)`
2. `public void offer(int element)`

3. `public int poll()`
4. `public void sort()`
5. `public MaxHeap(int[] arr)`

Part 2 – MaxHeap **constructor**

By the end of this week's assignment you are going to implement two constructors for this class. We start with the easy constructor for the lab. The signature of this constructor MUST be the following (only modify where it says "FILL IN" – DO NOT change the signature):

```
public MaxHeap(int maxsize)
```

As the signature of this method suggests, this is a constructor for this class. This method will initialize the status of the member variables (i.e., size) of this class based on the input parameter. So, this function will create the array (heap) with the specified size and initialize all of its elements to be zero. You need to be careful about whether any other field needs to be initialized at this stage (hint: think about what the value of size should be).

Part 3 – offer **method**

The signature of this method should be (only modify where it says "FILL IN" – DO NOT change the signature):

```
public void offer(int element)
```

This function will offer the given value (element) into the heap and position it into the correct location. Go back to the problem description part and make sure you are following the steps given EXACTLY. The element is supposed to be added to the latest position available in the heap array and then you must make proper changes to the heap structure to maintain the max heap property. Don’t forget to update the size value.

Part 4 – poll **method**

The signature of this method should be:

```
public int poll()
```

This function will remove and return the **maximum** value stored in the max heap. Recall that, by definition, this value would be stored in the root. The logic here is

similar to removing min from the min heap covered during the class. To implement this function, start by swapping the root with the last leaf, sift the new root down to restore heap property. If the heap is empty this method should return -1.

Part 5 – sort method

The signature of this method should be:

```
public void sort(int[] arr)
```

This function receives an input array arr and provides an in-place heap sort. That means you are not allowed to have **any auxiliary** array. This method invokes two routines, heapify() and siftDown(). We have provided a starting point for you by implementing sort() and heapify(). **To complete the implementation, you must write** siftDown().

Since the data is initially unordered and most likely violates the rules for a valid max heap, it must first be heapified by calling heapify(). This method starts at the ***last parent node*** and it continually sifts down the smallest descendant at each level by invoking siftDown(), where this method is passed the index to sift down from. If a child (left or right child of index) has a larger value than the parent (index), the largest child value is swapped with the parent. That node is then sifted down again until no sift occurs, or there are no more children. heapify() calls siftDown() on each parent node, from the last parent until the root, at which point the heap structure is valid. This means the largest value will be at the root of the tree, and every parent will have a value greater than or equal to its descendants.

Now that the heap structure has been built, the largest element is the root. This is swapped with the last leaf node so that it is now in the last position in the array. The trick

here is that once the largest element is stored in its final position, size is decremented so that it will remain untouched for the remainder of the sorting.

The heap from the root is then sifted down (by calling siftDown(0); why is 0 passed in as the parameter?), excluding the last position (how?). This process is repeated until all of the nodes except the root have been excluded. At this point, the data is sorted.

You should work out with pen and paper how the algorithm operates, running through the code that is already provided for you. Also think about the task of siftDown() and how it fits into the code (after all, you will be implementing it :-).

Part 6 – advanced MaxHeap constructor

The signature of this method should be:

```
public MaxHeap(int[] arr)
```

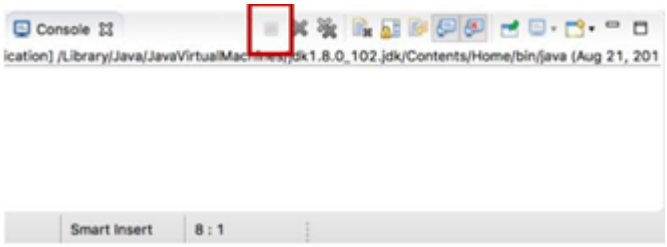
This is a constructor for the max heap class that initializes a heap and puts the values of the input array in it in such a way that the constructed max heap is valid. Hint: think about how you can reuse the functions you have implemented so far.

Part 7 – Test your code

As usual you need to test the functionality of the methods you have implemented. A set of test cases has been provided for you as part of **MaxHeap.java**. Uncomment the assignment portion of the tests and run the main function. If you see any red text that says “TEST FAILED”, you need to debug your code.

How to debug your code?

1. Use the Eclipse debugger you learned about during the first lab
2. If you see JavaStackOverflow, that means that you have an infinite recursive call and your recursive call is filling up the “call stack” (we talked about this concept in class). Go back and debug the method that is causing the problem.
3. Infinite loops! How would you know you have an infinite loop? As you should know from CSC120, if you have an infinite loop in your code, your code will not stop running. An easy way to inspect that in Eclipse is to look at your console window, if your code is done running the console should look like the following



If the little square marked above is red and continues to stay red, that means your code has an infinite loop!