1. public Decreaser<T> insert(T thing){}
   The insert method worked by taking the specified thing of type T and putting it into the Decreaser¡T¿ heap. The decrease method was then called to sort the newly inserted item to make sure its value was less than the parent node. The method used all three instance variable in the method. First was the array, to store "thing" as the last element in the array, specified by the second instance variable, size. Lastly, the ticker was used to count the ticks in the specific method. The decrease method was used as a helper method to reorder the array into descending order.

2. void decrease(int loc){}
   The decrease method evaluates the parent and child node for a specified location in the binary tree. If the parent node was larger than the child node then they would swap positions in the tree. The ticker instance variable was used to count the amount of operations done in the method while the array instance variable was used to switch the locations of the parent and child node. A helper method called swap was used to switch the values of the parent and child node.

3. public T extractMin(){}
   The extractMin method would take the minimum value of the array, in this case the root element of the array since it is in descending order. The method used all three instance variable. The ticker was used to count the amount of commands done in the method. The array method was used to change the locations of the first and last locations and the size instance variable allowed the last element in the array to become null. The size was then decreased by one because the extracMin method takes the minimum value out of the array. The swap method was used again as a helper method to swap the first and last values in the array and then the heapify method was called to reorder the array into the correct order for a binary tree.

4. private void heapify(int where){}
   The heapify method compares the parent node to its children at a specified location, where. It then sees if the two need to be swapped or not. If it does the method is then called again to check the next parent and child to see if the node needs to go further down in the tree. The array instance variable was used again here to help check the nodes in the array and change their locations when needed. The ticker instance variable was used to keep track of the ticks used in the method. Lastsly, the swap method was used as a helper method to change the elements values.

5. Include the tick-count graph obtained from the HeapTimer experiment (as described in the lab instructions) as a figure in your writeup. Does the curve have the shape suggested in the instructions?

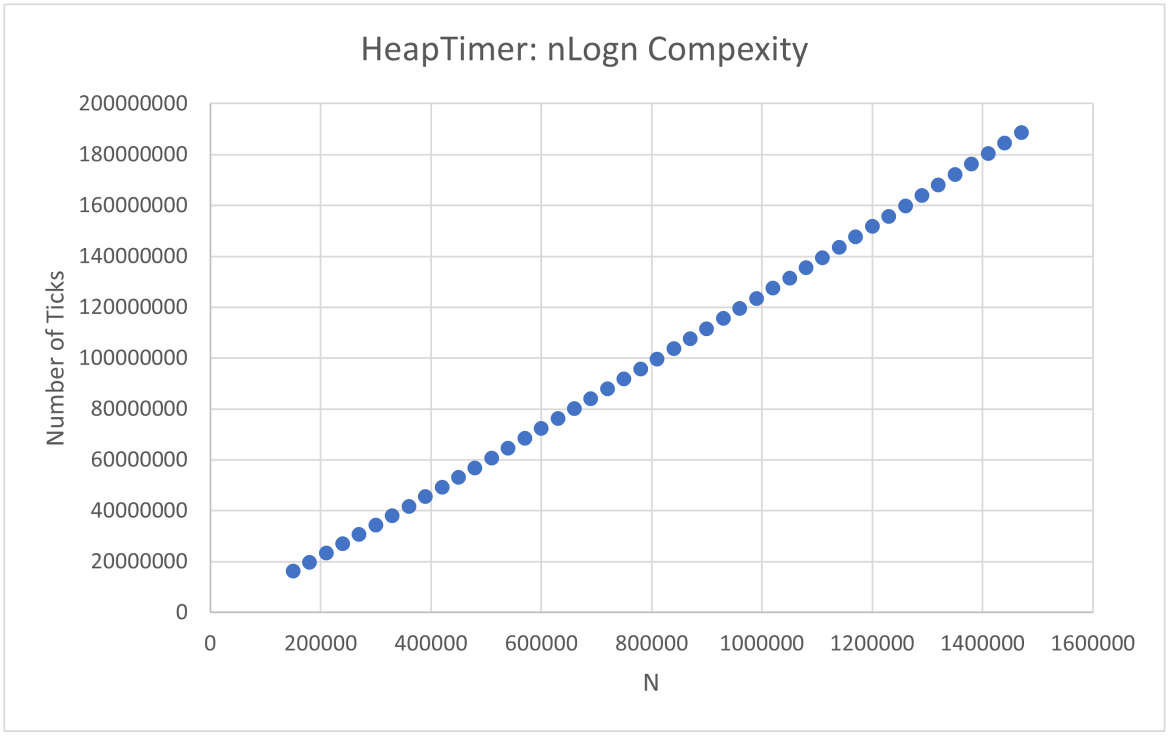

Figure 1: Heapify: nLogn Complexity

Based when plotting a linear line from the first two points, the curve was observed to be above the linear line as n increased. Also, when taking the measured time in ticks and dividing by nLogn the number stayed almost constant, around 20 with a small change less than 0.2. From this, the data does show a curve with Θ(nLogn).

6. What is the running time of calling extractMin n times on a heap of size n? (Note that size refers here to the number of elements in the heap rather than the total capacity of the heap.) Give an asymptotic upper bound (O) on the running time as a function of n, and justify this bound.
   In the extractMin method I have a tick when it gets the value of the array, which I assume constant time for. Then I also say that their is also constant time in the changing of the locations of the elements in the array. I then used a helper method called swap, which has three ticks per execution. This makes it also constant time, because if called once then there will be three ticks. However I also call the heapify method, starting it at the root. So, if their are n elements the worst case scenario is that it has to run through all the levels of the tree to resort the values. So I believe that the method extractMin doesn't have any for loops, the minimum is at the root of the tree so it takes constant time as well. There for I think the whole method has constant time, O(1).

7. Briefly describe what the heap property is and how it is maintained in the code during the insert and extractMin methods.
   The heapify property is how a binary tree keeps a pyramid like shape and always has the parent node less than or equal to its children. The pyramid like shape means that their will always be a left child before a right child and that for a given height, that height will be filled before going to height + 1. It is maintained in the insert method by the helper method decrease. This method will compare the node inserted with its parent node and the child is smaller they switch. This ensures that the smaller values are higher in the tree as well as the fact that the node could be inserted as a left or right child because both can be divided by two to get the parent node for both. The extractMin method maintains the heap property by starting with the parent node specified and working its way down the tree by

comparing the parent node to the smaller of the two children, if it has two children. This allows the switch between the smaller child, preserving the decrease in values character and the pyramid shape.

8. Could one side of a min-heap ever become very tall compared to the other side? To be precise, could the maximum height of any node in the left subtree of a min-heap ever differ from the max height of any node in the right subtree by more than (say) 5? If so, give a sequence of insertions that would produce such a min-heap; if not, explain why.
   If each node is being filled one after another to preserve the pyramid shape then no, otherwise you could have a root with two children and then only add values to the left child, causing the height to go up and up on the left but not the right. This shape would then end up not preserving the pyramid shape.

9. Name one advantage and one disadvantage of storing data in a min-first heap compared to storing it in an unordered list.
   One advantage of sorting data in a min-first heap is that for large data sets the exicution to getting the minimum value would be quick compared to an undordered list. However, one disadvantage is that inserting a value into the heap list would take a long time because it may need to be reordered to preserve the minimum value. This would not be the case for the unordered list because the value could just be added to the end.