**Time Complexity Analysis for Adaptable and Flexible Priority Queue:**

**replaceValue():**

*replaceValue()* has a time complexity of constant time also known as **O(1).** Since *replaceValue()* takes in an Entry object, we can get the index where the Entry is stored inside the array by using *Entry.getValue()*. From there, instead of looping through every element inside of the contents array, we can immediately access the correct Entry using our *contents.get()* method to set the new value of the Entry object, thus only performing one operation which is constant time.

**replaceKey():**

*replaceKey()* has a time complexity of **O(logn)**. Similarly to *replaceValue()*, we can already get the index of the Entry that we want to replace the value so we don’t have to loop through each element of the contents array as we can directly access the element to replace its current key with the new key. However, we have to maintain the heap state priority as the tree is no longer respecting either MIN\_STATE or MAX\_STATE so we must perform *upheap()* and *downheap()* operation. The node can only do one of the following possibilities: either *upheap()* and *downheap()* and both operations run in **O(logn)** operations. This is because the heap has **O(logn)** levels and when we call *upheap()* or *downheap()*, we are accessing the nodes one level above (parent) or below (child), thus the time complexity is **O(logn).**

**remove():**

*remove()* has a time complexity of **O(logn)**. To perform *remove()*, we must first swap the Entry that we want to remove with the last Entry of the tree. This is done in constant time as we already have access to the indices of both entries to replace. However, once the swap is done, the heap state priority is no longer respected as the key of the last entry of the tree might not maintain the tree to be MIN\_HEAP or MAX\_HEAP. We must then perform *upheap()* and *downheap()* operations which we have discussed previously runs in **O(logn)** times, thus *remove()* has a time complexity of **O(logn).**

**toggle():**

*toggle()* has a time complexity of **O(n).** To perform *toggle(),* we must change the state of the heap from MIN\_HEAP to MAX\_HEAP and vice versa first. Then, we must loop through every internal node of the tree element and perform a *downheap()* operation as performing a *downheap()* operation on any internal node will assure that the subtree of that node will respect the heap state priority. Since a heap is a binary tree and every node will contain at most two children, the maximum amount of iterations that it can perform is 2log(n) where 2 is the number of children per node and log(n) is the number of operations *downheap()* can perform. We know 2log(n) follows a geometric series and the worst case time complexity of geometric series on **O(n),** thus *toggle()* is linear time: **O(n)**