Time Complexity Analiysis

Part1 : Max-Heap:

Iterator of the heap:

```
public class MyIterator implements Iterator<E>{

    private int theData;
    private int lastReturnedItem;

    /**
     * Initializes the iterator.
     */
    public MyIterator() {
        theData = -1;
        lastReturnedItem = 0;
    }
    /**
     * Checks if the pointer is at the end of the heap.
     */
    @Override
    public boolean hasNext() {
        return (theData < heap.size() - 1);
    }
    /**
     * If exist, returns the next item.
     */
    @Override
    public E next() {
        if(hasNext()) {

            theData++;
            lastReturnedItem = theData;
            return heap.get(theData);
        }
        return null;
    }
```

Constructor just assigns some values to variables. So O(1)

Public Boolean hasNext() gets the values of theData and the heap(ArrayList)'s size and does a simple job. So O(1).

Public E next() if hasNext() is true (which is O(1)) calls the getter method (O(1)) . So O(1)

```java
/**
 *
 * @param x X is the value to set to the last returned item.
 * @return  Returns the item after insertion.
 */
public E setLast(E x) {
    heap.set(lastReturnedItem, x);
    return heap.get(lastReturnedItem);
}
```

Getters and setters are O(1) for an array. So O(1).

Max-Heap part:

```java
69    /**
70     *
71     * @param item Item is the element to add to the heap.
72     * @return
73     */
74    public boolean add(E item) {
75        int parent , current;
76
77        heap.add(item);
78        current = heap.size() - 1; // Smallest child.
79        parent = (current - 1) / 2; // Its parent.
80
81        while(heap.get(current).compareTo( heap.get(parent) ) > 0) { // If parent smaller than the child.
82            swap(current , parent); // swap.
83            current = parent; // And look for its parent.
84            parent = (current - 1) / 2;
85        }
86        return true;
87    }
```

At line 77: Adding to an ArrayList(heap) is O(1).

compareTo() compares a root of heap with a root of another heap. Datas are in the ArrayList so both getter and setter is O(1) so compareTo is O(1).

Swap() is just a couple of setters and getters of ArrayList. O(1)

```java
224    private void swap(int index1 , int index2) {
225
226        E temp = heap.get(index1);
227        heap.set(index1, heap.get(index2));
228        heap.set(index2, temp);
229    }
```

So inside the while loop is O(1).

While loop will continue until the value that added will be in the correct position which means lesser than its parent and greater than its childs.

So best case is adding an element which is smaller than every parent in the heap.

$T_{best}(n) = O(1)$ in that case because no need to swap.

Worst case is adding an element which is greater than all of the elements in the heap.

$T_{worst}(n) = O(\log n)$ because of the height of the tree. For example if the element number is 7, height will be 3 and I need to swap with the parent only 3 times. So add method $T(n) = O(\log n)$.

```
133    public E remove(int index) {
134        ArrayList<E> hold = new ArrayList<>();
135        int size = size();
136        E returnValue = heap.get(index);
137        for(int i = 0; i<size; i++) {
138            if(i != index) { // Keeping the values. Except the index that wanted to be deleted.
139                hold.add(heap.get(i));
140            }
141        }
142        heap = new ArrayList<>(); // Creating a new heap to replace the old one.
143        for(int i = 0; i<size - 1;i++) {    // That heap doesn't contain the item at the wanted index.
144            heap.add(hold.get(i));
145        }
146
147        return returnValue;
148    }
```

Inside for there is ArrayList.add() and ArrayList.get() . They both are O(1).

And first for loop will continue until i is the wanted index.

$T_{best}(n) = O(1)$ if the wanted index is 0.

$T_{worst}(n) = O(n)$ if the wanted index is the size – 1.

So $T(n) = O(n)$.

And second for loop will continue size -1 no matter what. So O(n).

The smallest element in the heap will be assign to the root of the heap. And then swapped with its child until it is in the correct position.

```
92      public E remove() {
93
94          int parent,
95              leftChild,
96              rightChild,
97              maxChild;
98
99          parent = 0;
100
101         E returnValue = heap.get(parent);
102         heap.set(parent , heap.remove(heap.size() - 1));    // Root is the last value in the heap array now.
103
104         while(true) { // Moving the root to its proper place.
105             leftChild = (2*parent) + 1;
106             rightChild = leftChild + 1;
107
108             if(leftChild >= heap.size()) { // It means there is no child of the current parent.
109                 break;
110             }
111             maxChild = leftChild; // Assume the leftChild is greater.
112
113             if(rightChild < heap.size() && heap.get(rightChild).compareTo(heap.get(maxChild)) > 0) {
114                 // If the rightChild is the greater one. Change the maxChild to rightChild.
115                 maxChild = rightChild;
116             }
117
118             if(heap.get(parent).compareTo( heap.get(maxChild)) < 0) { // If the parent is smaller than the maxchild
119                 swap(parent , maxChild); // Swap parent and the maxChild
120                 parent = maxChild;
121             }
122             else { // Structure is good now. Break.
123                 break;
124             }
125         }
126         return returnValue;
127     }
128     /**
```

Tworst (n) = $\Omega(n)$.

Tbest (n) = Θ(logn).

So T(n) is O(logn).

```java
/**
 * Removes the ith largest item.
 * @param wanted I'th largest item that wanted to be deleted.
 * @return Returns the removed item.
 */
public E remove_wanted_largest(int wanted) {

    int counter = 1;
    ArrayList<E> hold = new ArrayList<>();
    E returnValue = null;

    if(heap.size() == 0) {
        return null;
    }

    for(int i = 0; i<heap.size(); i++) {
        if(counter == wanted) {
            returnValue = remove();
            break;
        }
        else {
            hold.add(remove()); // remove() deletes the root of the heap which is the largest item.
                                // And keeping the values except the removed item.
            counter++;  // and counting.
        }
    }

    for(int i = 0; i<hold.size(); i++) { // Inserting the stored items to the heap.
        add(hold.get(i));
    }

    return returnValue;

}
```

Second for loop is O(n).
First for loop calls remove() several times which is O(logn).
Tbest(n) is the wanted largest element is the first largest element so it is same as remove() which is O(logn).
Tworst is the wanted largest element is the smallest. It means for loop will execute remove() n times. O(n logn).

Remove_wanted_largest() will be O(n . logn) . O(n) = O(n^2 logn).

```java
    /**
     * Merges two same type of heaps.
     * @param another Another is the heap to be added to the current heap.
     */
    public void merge(MyHeap<E> another) {

        if(another.size() == 0 ) {
            return;
        }

        for(int i = 0; i< another.size(); i++) {
            add(another.get(i)); // Adding all of the indexes of the another heap to the current heap.
        }
    }
    /**
     *
     * @param index
     * @return Returns the item at the wanted index.
     */
    public E get(int index) {
        return heap.get(index);
    }
    /**
     * Finds the index of the wanted item.
     * @param item  Wanted item.
     * @return   Returns the index of the wanted item.
     */
    public int search(E item) {
        for(int i = 0; i<heap.size(); i++) {
            if(heap.get(i).compareTo(item) == 0) { // checks if the "i" is the wanted index.
                return i;
            }
        }
        return -1;
    }
```

Merge() will execute add() (O(logn)) method the another heap's size times(m).
So Tmerge(n) = O(m . logn).

Search method will execute compareTo and size() n times which are O(1) so Tworst(n) = O(n). Tbest(n) = O(1) which is the wanted element is in the beginning of the heap. So Tamortized(n) = O(n).