1)
input: 4 2 7 9 12 1 3 0 10 11
Assume first index is 0. Left child in array is A[2k+1], right child is A[2k+2].
insert 4: [4]
insert 2: [4, 2]
insert 7: [7, 2, 4]
insert 9: [9, 7, 4, 2]
insert 12: [12, 9, 4, 2, 7]
insert 1: [12, 9, 4, 2, 7, 1]
insert 3: [12, 9, 4, 2, 7, 1, 3]
insert 0: [12, 9, 4, 2, 7, 1, 3, 0]
insert 10: [12, 10, 4, 9, 7, 1, 3, 0, 2]
insert 11: [12, 11, 4, 9, 10, 1, 3, 0, 2, 7]
The binary max heap array after the input sequence is [12, 11, 4, 9, 10, 1, 3, 0, 2, 7].

2)
To run a single delMax, first swap 7 and 12, and the 12 is removed.
The 7 must be moved to its correct place.
Swap 7 and 11.
Swap 7 and 10.
The binary max heap array after single delMax is [11, 10, 4, 9, 7, 1, 3, 0, 2]

3)

main.py

```python
def sort_using_heap(li):
    for i in range(len(li)):
        if (li[i] > li[(i-1) // 2]):
            j = i
            while (li[j] > li[(j-1) // 2]):
                li[j], li[(j-1) // 2] = li[(j-1) // 2], li[j]
                j = (j - 1) // 2

    for i in range(len(li) - 1, 0, -1):
        li[0], li[i] = li[i], li[0]
        j = 0
        idx = 0
        while (True):
            idx = 2*j + 1
            if ((idx < (i-1) and li[idx] < li[idx+1])):
                idx += 1
            if (idx < i and li[j] < li[idx]):
                li[j], li[idx] = li[idx], li[j]
            j = idx
            if (i <= idx):
                break

```

4)
The height of a complete binary tree is $O(logn)$. Worst case is the numbers sink to the leaf. We do this n times. So total $O(nlogn)$.
Make heap is also $O(nlogn)$.
Therefore worst case complexity is $O(nlogn)$.

5)
We use 2 Priority Queues, a min heap and max heap. The min heap saves bigger half of the elements with the smallest at the root. The max heap saves the smaller half of the elements with biggest at the root. We rebalance by changing the root element to the min heap or max heap if differ by more than 1.

Main.java

```java
class runningLmedian
{
    PriorityQueue<Integer> minHeap = new PriorityQueue<>();
    PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Comparator.reverseOrder());

    public void insertNum(int num)
    {
        minHeap.offer(num);
        maxHeap.offer(minHeap.poll());
        if(minHeap.size() < maxHeap.size())
            minHeap.offer(maxHeap.poll());
    }

    public double getLmedian() {
        if(minHeap.size() > maxHeap.size())
            return minHeap.peek();
        return (minHeap.peek() + maxHeap.peek()) / 2.0;
    }
}

```

Inserting a number takes $O(logn)$. Getting a number takes $O(1)$. So overall it's $O(logn)$ complexity.