Homework Sheet 6

Author	**Matriculation Number**	**Tutor**
Abdullah Oğuz Topçuoğlu	7063561	Maryna Dernovaia
Ahmed Waleed Ahmed Badawy Shora	7069708	Jan-Hendrik Gindorf
Yousef Mostafa Farouk Farag	7073030	Thorben Johr

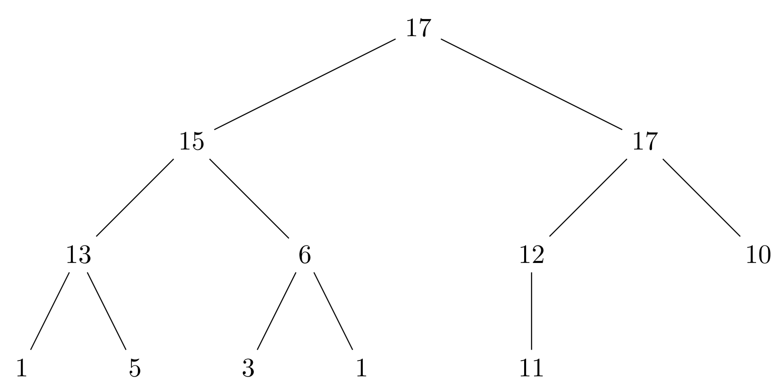
Exercise 1

(a)

We are given the array

$$A = [17, 15, 17, 13, 6, 12, 10, 1, 5, 3, 1, 11].$$

The heap tree would be



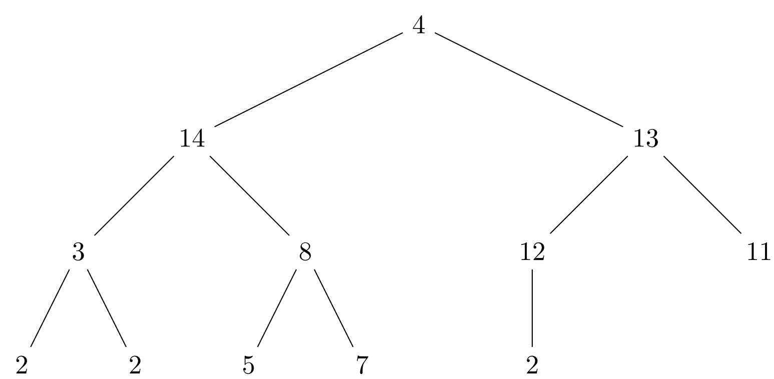
This is a heap since all nodes satisfy the max heap property.

(b)

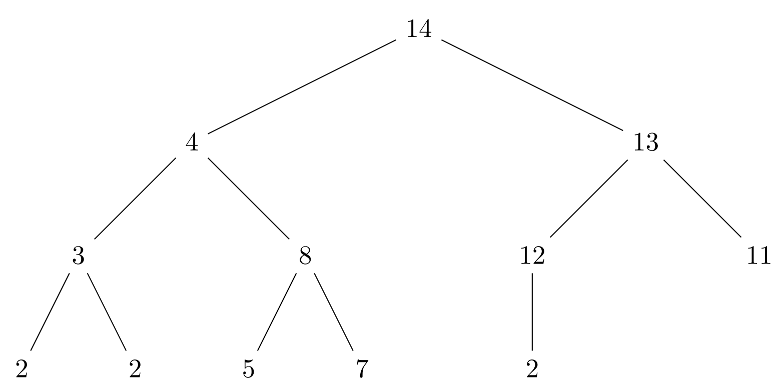
We are given the array

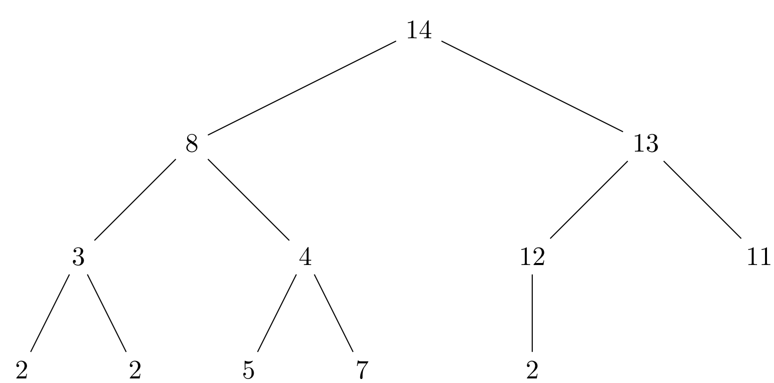
$$A = [4, 14, 13, 3, 8, 12, 11, 2, 2, 5, 7, 2].$$

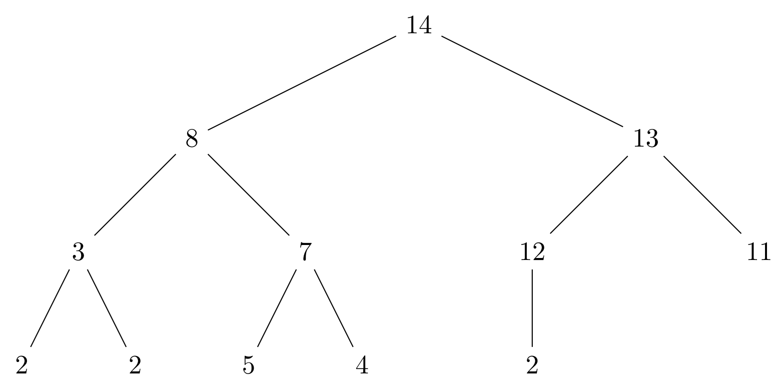
The heap tree would be



This is a near heap because only the root node (4) vioaltes the max heap property. Every other node satisfies it.
Steps of heapify operation:





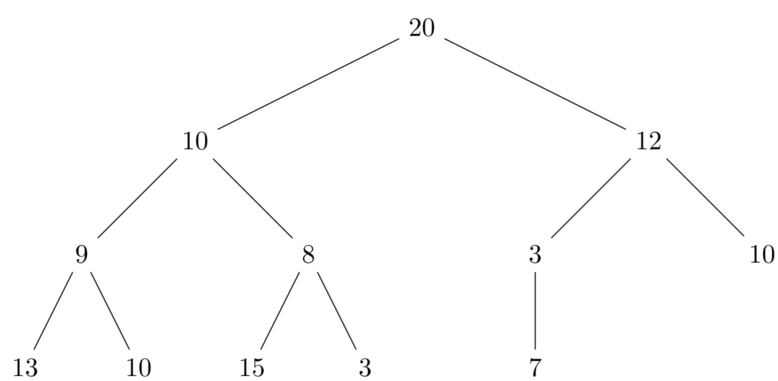


(c)

We are given the array

$$A = [20, 10, 12, 9, 8, 3, 10, 13, 10, 15, 3, 7].$$

The heap tree would be



This is not a heap also not a near heap because multiple nodes violate the max heap property. For example the node with the value 9 and the node with the value 8 both violates the max heap property and they are not descendants of each other.

Exercise 3

We are gonna use the same approach that we used in the lecture. We will build a max heap from the given array and then we will call DeleteMax() k many times.

```
int KthLargestElement(A[1..n], k)
  BuildMaxHeap(A)
```

```
  for i = 1 to k do
    result = DeleteMax(A)
  return result

void BuildMaxHeap(A[1..n]) // this is called makeheap() in the lecture slides
  for i = n/2 down to 1 do
    Heapify(A, i) // heapify function from the lecture
```

Running Time Analysis:
Building the max heap takes O(n) time as we saw in the lecture. Each call to DeleteMax() takes O(log n) time. Since we are calling DeleteMax() k many times, this part takes O(k log n) time. Therefore the total running time of the algorithm is

$$O(n) + O(k \log n) = O(n + k \log n).$$

Correctness Proof:
The BuildMaxHeap() function builds a valid max heap from the given array A. In a max heap the maximum element is always at the root node. The DeleteMax() function removes and returns the maximum element from the heap and then reestablishes the max heap property by calling Heapify(). So at the time we call DeleteMax() for the kth time we get the kth largest element from the original array A and then we return it.

Exercise 4

We will maintain two heaps: a max heap Hlow to store the lower half of the elements and a min heap Hhigh to store the upper half of the elements. The max heap Hlow will allow us to efficiently retrieve the maximum element of the lower half. The min heap Hhigh will allow us to retrieve the minimum element of the upper half.

```
int[] RunningMedian(A[1..n])
  Hlow = new MaxHeap()
  Hhigh = new MinHeap() // the difference is the comparison function
  R = new int[1..n]
  for i = 1..n do
    if Hlow.isEmpty() or A[i] <= Hlow.Max() then
      Hlow.Insert(A[i])
    else
      Hhigh.Insert(A[i])

    // Balance the heaps
    if Hlow.size() > Hhigh.size() + 1 then
      Hhigh.Insert(Hlow.DeleteMax())
    else if Hhigh.size() > Hlow.size() then
```

```
        Hlow.Insert(Hhigh.DeleteMin())

      // Calculate median
      // either they are equal size or Hlow has one more element
      if Hlow.size() == Hhigh.size() then
        R[i] = Hhigh.Min()
      else
        R[i] = Hlow.Max()
    return R
```

Running Time Analysis:
Each insertion into a heap takes O(log m) tim where m is the number of elements in the heap. Since we are inserting n elements the total time for insertions is O(n log n). Balancing the heaps involves at most one deletion and one insertion, which also takes O(log n) time. Since we do this for each of the n elements, the total time for balancing is also O(n log n). Calculating the median takes O(1) time for each element, resulting in O(n) time for all n elements. Therefore, the overall time complexity of the algorithm is O(n log n).

Correctness Proof:
The algorithm maintains two heaps to keep track of the lower and upper halves of the elements seen so far. By ensuring that the sizes of the heaps differ by at most one, we can determine the median after each insertion. If the heaps are of equal size, the median is the minimum of the upper half (Hhigh). If Hlow has one more element than Hhigh, the median is the maximum of the lower half (Hlow). This approach guarantees that we always have access to the median in O(1) time after each insertion, and thus correctly computes the running median for each prefix of the array A.