1)

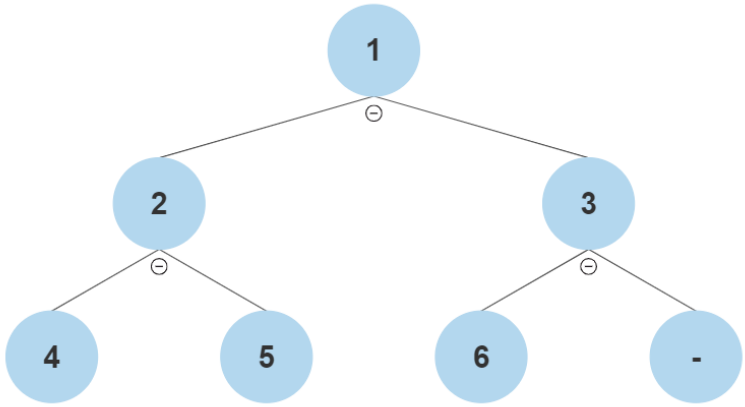
Yes, this is true. We know that a min heap binary tree is a tree where the root node has a minimum value in the tree, and this holds true also for all sub-trees in the tree. When the array that is has an ascending order arr[i] < arr[i+1]. The array implementation of a binary heap tree has a rule, that children of the node at index i (=arr[i]) can be found at arr[2i+1] and arr[2i+2]. We know that arr is sorted, so arr[i] < arr[2i+1] and arr[i] < arr[2i+2]. So, a sorted array is a min heap. However, keep in mind, that it does not work vice versa – an array based binary heap will not store a sorted list, and sorted array in descending order is a max-heap.

Example:

arr= [1, 2, 3, 4, 5, 6]

(arr is sorted in ascending order: arr[i] < arr[i+1], and is a complete binary tree)

Representation of min-heap:



In this case, every parent node < its children. The smallest element (=1) is at the root.

Therefore, a sorted array is a valid min-heap.

2)

The runtime of the quick-sort algorithm depends on the pivot selected. If we select the last element of already sorted array as a pivot, then we will end up with all partitions having only one element, so we end up with a recursion tree with the depth n (n = number of elements in the array). This results in running time of $O(n^2)$. If we select the element at index $\lfloor n / 2 \rfloor$ as a pivot, the partitions will be divided into equal parts. Both parts will have $n/2$ elements (this depends if the size of an array is even or odd, if odd -> $arr1.size = n/2$; $arr2.size = (n/2) + 1$). So in this case the depth of the recursion tree will be $\log(n)$. This will result in complexity of **$O(n*\log n)$.**

3)

The third line handles the case when the two strings being compared are identical. (== refers to the same memory location). When two string objects are identical the output of the method compareTo() is always 0. When we check if String s and String t are the same object by "==" operator, we avoid iterating through the characters of the strings and we can return 0 immediately. Without this line of code, the method would have to iterate through all the characters of both strings and compare them, even if they are the same. Note that, in Java when a string is created as String <name> = "<string>"; the compiler checks if an existing string with the same content already exists. If such a string exists,

the new string reference is assigned to the existing object, and the memory location is used for both string references (pointers of the two same string objects point to the same memory location). So, the third line saves time and reduces the number of characters comparisons needed.

Example:

String s = “hello”;

s.compareTo(“hello”) will return 0 from the 3rd line -> if(s==t) is true; without this line we would have to loop trough every character of both strings and compare them

However s.compareTo(new String(“hello”)) will go to a for loop although the return output is still 0. This is because s==t is now false (new String(“hello”) has a pointer to a different object/memory location then s).

4)

R,R,P,O,T,Y,I,I,U,Q,E,U

or

(18,R) (18,R) (16,P) (15,O) (20,T) (25,Y) (9,I) (9,I) (21,U) (17,Q) (5,E) (21,U)

or

Method	**Return Value**	**Priority Ques Contents**
Insert(16,P)		{(16,P)}
Insert(18,R)		{(16,P), (18,R)}
Insert(9,I)		{(9,I), (16,P), (18,R)}
Insert(15,O)		{(9,I), (15,O), (16,P), (18,R)}
removeMax()	(18,R)	{(9,I), (15,O), (16,P)}
Insert(18,R)		{(9,I), (15,O), (16,P), (18,R)}
removeMax()	(18,R)	{(9,I), (15,O), (16,P)}
removeMax()	(16,P)	{(9,I), (15,O)}
Insert(9,I)		{(9,I), (9,I), (15,O)}
removeMax()	(15,O)	{(9,I), (9,I)}
Insert(20,T)		{(9,I), (9,I), (20,T)}
removeMax()	(20,T)	{(9,I), (9,I)}
Insert(25,Y)		{(9,I), (9,I), (25,Y)}
removeMax()	(25,Y)	{(9,I), (9,I)}
removeMax()	(9,I)	{(9,I)}
removeMax()	(9,I)	{}
Insert(17,Q)		{(17,Q)}
Insert(21,U)		{(17,Q), (21,U)}
Insert(5,E)		{(5,E), (17,Q), (21,U)}
removeMax()	(21,U)	{(5,E), (17,Q)}
removeMax()	(17,Q)	{(5,E)}
removeMax()	(5,E)	{}
Insert(21,U)		{(21,U)}
removeMax()	(21,U)	{}
Insert(5,E)		{(5,E)}