Q1 (Page 9): The height h doesn’t include the root node?

Answer: The root is put at depth/level 0 and we count the height from the children of root. So for the heap in this slide, the height h = 3.

Q2 (Page 18): How does Heapsort select the Element of the Root node? What if the letter “A” is the Root?

Answer: The root node will be selected in the heap construction phase. If the construction is correct, the letter “A” will not be the root if it is a maximizing heap. The heap construction function will identify the maximum value and put it at the root.

Q3 (Page 18): What do we need to do before creating the heap?

Answer: Given an input array, we will first call the function for heap construction. The output of this function will be a maximizing heap. The main idea of heap construction is to first fix the heap structure (shape), and then check the content constraint (partial order tree property). If the content constraint is not satisfied, we will fix the heap in some way.

Q4 (Page 22): Isn't the Y at index = 1? When deleting Y, R will be the max, then why are we using for (i = n;. ....)?

Answer: Yes, Y is at index 1 at the beginning as it is the current max. After deleting Y, we will store Y at index n. This is because in deleteMax, the size of the heap H will be reduced by 1, and index n will move out of the heap. The space at index n will be used to store Y because we are sorting in increasing order. After Y is deleted, the heap will be repaired and R will become the new max and be put at the root (i.e., index 1). Then this process continues and R will be deleted from the heap and stored at index n-1. That is why we have this “for loop” with a running index i. It represents the position to store the current max extracted.

Q5 (Page 24): If k is the last element removed, and the smallest value in the Heap, wouldn't the fixHeap() always enter the else statement and never enter the if statement?

Answer: Yes, you are correct. If the value of k is the smallest in the heap, fixHeap will never enter the if statement. What it will do is always entering the else statement, putting the winner of the two children in the current root, and moving one level down each time with another fixHeap at the corresponding sub-heap. Then it will stop at the base case when it reaches a leaf node and puts k there.

Q6 (Page 24): When H is leaf mean that n = 1?

Answer: Yes, if n here refers to the current size of the input heap.

Q7 (Page 27): What is currentSize?

Answer: currentSize refers to the size of the input heap H, i.e., the number of elements in H.

Q8 (Page 27): Is heap an array?

Answer: Yes, the heap here is implemented as an array. It is an effective and handy implementation as there is a one-to-one correspondence between the tree representation and the array representation. We could easily traverse the tree up and down by manipulating the array indices (the rules are given in Page 15).

Q9 (Page 52): Why Quicksort performs better than Radix, despite that the time complexity of Quicksort is O(nlogn) while that for Radix is O(n)? Can it be that it has many more terms added to it which add complexity?

Answer: Yes, one possible reason is that, the constant embedded in the big notation of Radix is much larger than that in Quicksort, which results in a worse practical performance of Radix, when compared with Quicksort. Another possible reason may be, the data size n tested is not big enough to

show the asymptotic trend. So you may enlarge the value of n and see if the conclusion would be different.

Q10 (Page 52): What condition does the last column of the table represent?

Answer: The algorithms were tested on an input array with 100,000 elements. The input array is in random order.

Q11 (Page 52): In terms of the performance of Quicksort vs Radix, does that mean the constant cannot be ignored here?

Answer: The asymptotic order gives us a sense when the input size n goes extremely large, what the complexity is. That is why it ignores constants. However, in practical applications, the constant may matter as well because it may also affect the performance significantly, especially when the input size n is not big enough to compensate the cost incurred by the constant. So in real applications, we should consider the actual scenario and shouldn't blindly choose an algorithm purely based on its asymptotic complexity.

Q12 (Page 52): So you are saying that either the n is too small or constant is affecting, which result in quicksort being faster than Radix?

Answer: Yes, that is correct. Please refer to the answer to Q9 for details.