Wing Kuen Lee <cswkl@ust.hk>

6.1-3

Let i denote the index of the element at the root of a subtree of a heap. Then its left child is A[2i] and its right child is A[2i + 1], if available. Recall that the definition of the max-heap property is: For every node i other than root, A[Parent(i)] ≥ A[i] where Parent(i) = $\lfloor i/2 \rfloor$. Therefore,

For the left child 2i, A[Parent(2i)] = A[$\lfloor 2i/2 \rfloor$] = A[i] ≥ A[2i]
For the right child 2i+1, A[Parent(2i+1)] = A[$\lfloor (2i+1)/2 \rfloor$]=A[$\lfloor i+1/2 \rfloor$] = A[i] ≥ A[2i + 1]

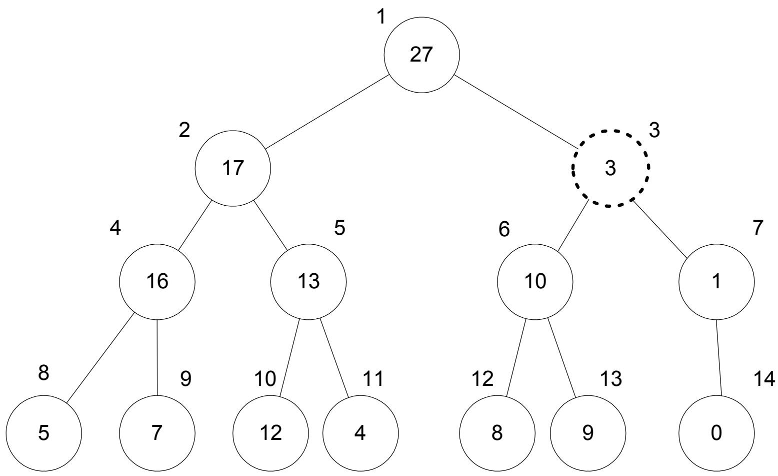
Since i is arbitrary, the largest element in a subtree of a heap is at the root of the subtree.

6.2-1

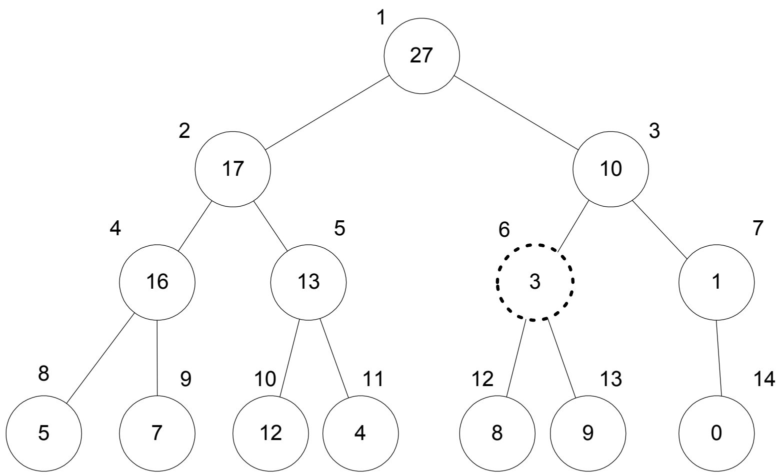
The array index begins at 1:

1	2	3	4	5	6	7	8	9	10	11	12	13	14
27	17	3	16	13	10	1	5	7	12	4	8	9	0

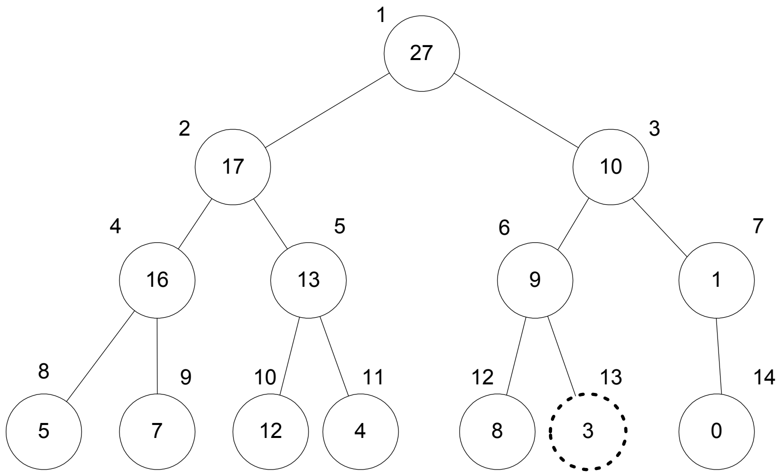
1)



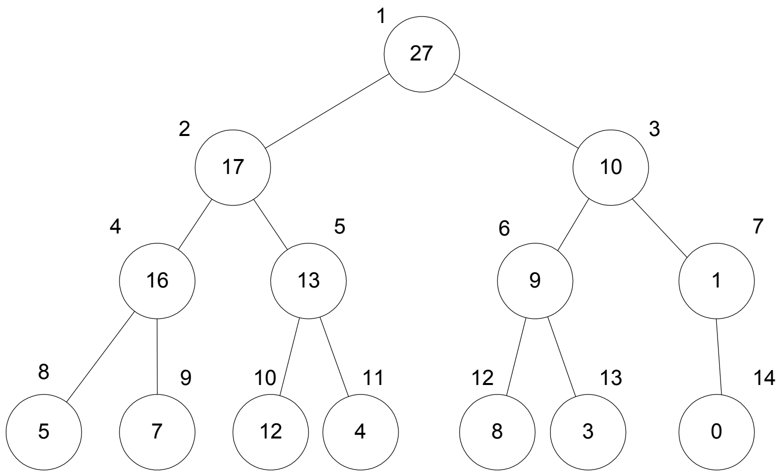
2)



3)



4)



6.3-2

Notice that the conversion of an array into a heap is in a bottom-up manner. When processing node *i*, we want the subtrees of node *i* already be heaps. In the other words, we want the node processing order to guarantee that the subtrees rooted at children of a node *i* are heaps before HEAPIFY is executed at that node. Because the elements at the bottom have larger index number than their parents', we want the loop index *i* in line 2 of BUILD-MAX_HEAP to decrease to 1.

[We may not obtain a heap if the process starts from the root to the bottom. Try apply the algorithm on <4,1,3,8,6>].

7.1-1

[This solution is done using the algorithm from the lecture, which also appears in the first edition of the textbook. It now appears as Q7-1 on page 159.]

pivot = A[1]=13

1)

i	*x*												*j*
	13	19	9	5	12	8	7	4	11	2	6	21	

2)

x, i										*j*	
13	19	9	5	12	8	7	4	11	2	6	21

(swap i,j)

3)

i										*x,j*	
6	19	9	5	12	8	7	4	11	2	13	21

4)

	i								*j*	*x*	
6	19	9	5	12	8	7	4	11	2	13	21

(swap i,j)

5)

	i								*j*	*x*	
6	2	9	5	12	8	7	4	11	19	13	21

6)

								j	*i*	*x*	
6	2	9	5	12	8	7	4	11	19	13	21

Return 9 (index begins at 1)

7.2-4

The performance of quicksort is at best $\Theta(n\lg n)$. However, the performance of insertion-sort is at best $\Theta(n)$. Now consider the algorithm of insertion-sort on page 24 of the textbook. For an almost-sorted input, the while-loop will be executed a few times more than in the best case. Since the extra steps are still constant, the performance of insertion-sort is still $\Theta(n)$. Indeed, using the analysis similar to the one on page 24, $T(n) = c1*n + c2*(n-1) + c4*(n-1) + c5*(n-1+k1) + c6*k2 + c7*k2 + c8*(n-1) = \Theta(n)$, where k1, k2 are some extra steps.

On the other hand, no matter if the input is already sorted or not, the best-case of quicksort is $\Theta(n\lg n)$. Thus, when the input is almost-sorted, insertion-sort beats quicksort.

8.3-3

Base case:
Let d=1. We only sort on the least significant digit. Obviously, the radix sort works on the least sig. digit.

Hypothesis:
Radix sort works on numbers with arbitrary digits, using an intermediate stable-sort.

We need to assume that radix sort uses a stable sort to sort array A on digit i so the intermediate sort is stable. Assume also that, for d = k, the least k-th sig. digits of the numbers are sorted properly.

Inductive case:
For d = k + 1, use a stable sort to sort array A on digit k+1. Since the least k-th digits are sorted properly by a stable sort (by assumption), after a stable sort is used on digit k+1, the order of the least k-th digits is preserved. Thus, the least k+1 th digits of n numbers are in order.

[Not all numbers in array need to have the same number d. The empty digit can be filled with '0']

By induction, radix sort works.

8.6

c)
Suppose there are *2n* elements: $a_1, a_2, a_3, \ldots, a_{2n}$ such that $a_1 \le a_2 \le \ldots \le a_{2n}$. Moreover, we have two sorted arrays A1 and A2, where a_i in A1 and a_{i+1} in A2. Assume that a_i will not be compared with a_{i+1}. Then,

Case 1: a_i is compared with an element $a \neq a_{i+1}$ in A2. However, a must be less than a_{i+1} because a_i and a_{i+1} are consecutive. Thus, all such elements in A2 will be merged, and a_i will finally be compared with a_{i+1}.

Case 2: a_{i+1} is compared with an element $b \neq a_i$ in A1. However, b must be less than a_i because a_i and a_{i+1} are consecutive. Thus, all such elements in A1 will be merged, and a_{i+1} will finally be compared with a_i.

By contradiction, if two elements are consecutive in the sorted order and from opposite lists, then they must be compared.

d)
Suppose there are *2n* elements: $a_1, a_2, a_3, \ldots, a_{2n}$ such that $a_1 \le a_2 \le a_3 \le \ldots \le a_{2n}$. The first sorted list A1=< $a_1, a_3, \ldots, a_{2n-1}$> and the second A2=<$a_2, a_4, \ldots, a_{2n}$>. By part c, we must have $a_1 \le? a_2 \le? a_3 \le? \ldots \le? a_{2n}$ where $\le?$ denotes comparison. Therefore, there are 2n-1 comparisons for n elements in each list.