Data Structures and Algorithms in Java™

Sixth Edition

Michael T. Goodrich
Department of Computer Science
University of California, Irvine

Roberto Tamassia
Department of Computer Science
Brown University

Michael H. Goldwasser
Department of Mathematics and Computer Science
Saint Louis University

Instructor's Solutions Manual

WILEY

Chapter 9 Priority Queues

Hints and Solutions

Reinforcement

R-9.1) Hint Each removeMin operation takes $O(\log n)$ time.

R-9.2) Hint Observe that the keys are guaranteed to satisfy the heap-order property. What other property does T need to satisfy in order to be a heap?

R-9.3) Hint Use a simple list-based priority queue and a pencil with a good eraser.

R-9.3) Solution $(1,D)$, $(3,J)$, $(4,B)$, $(5,A)$, $(2,H)$, $(6,L)$.

R-9.4) Hint There is a very good reason this exercise appears in this chapter.

R-9.4) Solution The best data structure for this air-traffic control simulation is a priority queue. The priority queue will enable the handling of the time stamps and keep the events in order so that the event with the smallest time stamp is extracted easily.

R-9.5) Hint Be prepared!

R-9.5) Solution Keep an additional variable that references the current minimum entry. This allows min to run in $O(1)$ time. Note that removeMin will still require $O(n)$ time; although the current min can be easily found and removed, that method must look through all remaining elements to identify the new minimum.

R-9.6) Hint Sounds too good to be true.

R-9.6) Solution See previous solution.

R-9.7) Hint Mimic the illustration style used in the book.

R-9.7) Solution

22	15	36	44	10	3	9	13	29	25
3	15	36	44	10	22	9	13	29	25
3	9	36	44	10	22	15	13	29	25
3	9	10	44	36	22	15	13	29	25
3	9	10	13	36	22	15	44	29	25
3	9	10	13	15	22	36	44	29	25
3	9	10	13	15	22	36	44	29	25
3	9	10	13	15	22	25	44	29	36
3	9	10	13	15	22	25	29	44	36
3	9	10	13	15	22	25	29	36	44

R-9.8) Hint Mimic the illustration style used in the book.

R-9.8) Solution

22	15	36	44	10	3	9	13	29	25
15	22	36	44	10	3	9	13	29	25
15	22	36	44	10	3	9	13	29	25
10	15	22	36	44	3	9	13	29	25
3	10	15	22	36	44	9	13	29	25
3	9	10	15	22	36	44	13	29	25
3	9	10	13	15	22	36	44	29	25
3	9	10	13	15	22	29	36	44	25
3	9	10	13	15	22	25	29	36	44

R-9.9) Hint Think about where insertion-sort has to put each added element and design your sequence so that insertion-sort has to put each next element as far as possible.

R-9.9) Solution A worst-case sequence for insertion-sort would be one that is in descending order of keys, e.g., 44 36 29 25 22 15 13 10 9 3. With this sequence, each element will first be moved to the front and then moved back in the sequence incrementally, as every remaining is processed. Thus, each element will be moved n times. For n elements, this means at a total of n^2 times, which implies $\Omega(n^2)$ time overall.

R-9.10) Hint Where might the second smallest key be?

R-9.11) Hint If the smallest is at the top of the heap...

R-9.11) Solution The largest key in a heap may be stored at any external node.

R-9.12) Hint The answer can be found in Section 9.2.2.

R-9.13) Hint Mimic the illustration style used in the book for insertion-sort and selection-sort.

R-9.14) Hint Consider the heap-order property and the definition of the level number of a node in a tree.

R-9.14) Solution Yes, tree T is a heap. It is a complete binary tree and each node stores a key value greater than the key of its parent, except for the root.

R-9.15) Hint Recall the definition of a complete binary tree.

R-9.15) Solution Since a heap is a complete binary tree, the levels of the heap are filled left to right. Thus, a node with the left child nay not have the right child. However, if a node has the right child, it must also have the left child.

R-9.16) Hint The answers are "yes,no,yes." Now all you have to do is to give examples for the yeses and a reason for the no.

R-9.16) Solution With a preorder traversal, a heap that produces its entries in increasing order is that which is represented by the array list $[1,2,5,3,4,6,7]$. There does not exist a heap for which an inorder traversal produces the keys in order. This is because in a heap the parent is always less than all of its children or greater than all of its children. The heap represented by $[1,5,2,7,6,4,3]$ is an example of one which produces its keys in decreasing order during a postorder traversal.

R-9.17) Hint The preorder sequence starts out 0, 1, 3, 7, ...

R-9.18) Hint Consider the last $n/2$ terms in this sum.

R-9.18) Solution Consider the last $n/2$ terms in this sum. Each one is at least $\log n/2 = \log n - 1$. Thus, this sum is at least $(n/2)\log n - n/2$, which is $O(n \log n)$.

R-9.19) Hint Try to construct a heap that has larger elements in left subtrees.

R-9.19) Solution Imagine the heap which is represented by the array list $[1,5,2,8,9,7,6]$. This heap will not produce keys in nondecreasing order when a preorder traversal is used.

R-9.20) Hint Try to construct a heap that has larger elements in right subtrees.

R-9.20) Solution Imagine the heap which is represented by the array list $[1,5,2,8,9,7,6]$. This heap will not produce keys in nonincreasing order when a postorder traversal is used.

R-9.21) Hint Mimic the illustration style used in the book.

R-9.22) Hint Mimic the illustration style used in the book.

R-9.23) Hint You need to be very careful about how you partition the keys between the subtrees rooted at the children of the root.

R-9.24) Hint Structure the insertions so that each requires lots of down-heap bubbling.

Creativity

C-9.25) Hint Figure out a way to time stamp the entries in the priority queue.

C-9.25) Solution Maintain a variable m initialized to 0. On a push operation for element e, call insert(m, e) and decrement m. On a pop operation, call remove and increment m.

C-9.26) Hint Figure out a way to time stamp the entries in the priority queue.

C-9.26) Solution Maintain a `maxKey` variable initialized to 0. On an enqueue operation for element e, call insert(`maxKey`, e) and increment `maxKey`. On a dequeue operation, call removeMin.

C-9.27) Hint Is it ever possible that a new element gets a key that is strictly smaller than a previously inserted element?

C-9.28) Hint Manage the array circularly.

C-9.29) Hint Do a single upward swap and recur (if necessary).

C-9.30) Hint Do a single downward swap and recur (if necessary).

C-9.31) Hint Do simple up-and-down searches in the tree to locate the last node each time.

C-9.31) Solution

```
/**
 *  Utility called just after insert has been called. It updates 'last'
 *  reference to be an external node of a proper binary tree to expand.
 */
private Position<E> findInsertionPosition( ) {
  Position<E> z;      // desired insertion position
  if (isEmpty( )) {
    z = root( );
  } else {
    z = last;         // assumed reference to current last position
    while (!isRoot(z) && z == right(parent(z)))
      z = parent(z);              // walk upward
    if (!isRoot(z))
      z = right(parent(z));       // then go to right sibling
    while (!isExternal(z))        // and finally
      z = left(z);                // find leftmost internal node in subtree
  }
  return z;
}
```

C-9.32) Hint Think about what changes need to be made when leaves are created or destroyed.

C-9.33) Hint Consider the binary expansion of $n-1$, n, and $n+1$.

C-9.33) Solution The path to the last node in the heap is given by the path represented by the binary expansion of n with the highest-order bit removed.

C-9.34) Hint Note that the entries do not need to be reported in sorted order. Use binary recursion on the subtrees of the heap and think about where the keys smaller than k are stored in the heap H.

C-9.34) Solution If the root of the tree has a key value less than k, record that value and then recursively search both the left and right subtrees. This algorithm takes $O(k)$ time, because there is no node in H storing a key larger than k that has a descendant storing a key less than k.

C-9.35) Hint Think carefully about how location-aware entries can be implemented efficiently.

C-9.36) Hint Use calculus (or review the hint for Exercise C-4.39).

C-9.36) Solution This summation is relevant because the time for each upheap call is bounded by $O(i)$ for a node at depth i, and there are at most 2^i nodes at depth i. For an analysis of this series, see solution to Exercise C-4.39.

C-9.37) Hint Study the combine step in the bottom-up heap construction algorithm.

C-9.37) Solution Create a new node to serve as rot of T, linked to the roots of T_1 and T_2 as its children, and then remove a leaf from one of the trees and place its item at the new root of T. Then call downheap at that root to reestablish the heap property.

C-9.38) Hint Use a suitably constructed heap.

C-9.38) Solution Build a heap storing the frequent flyers and their mileage, using bottom-up heap construction. This takes $O(n)$ time. Next, call removeMin $\log n$ times, which takes $O(\log n \cdot \log n)$ time, to determine the top $\log n$ flyers. Thus, the total time is $O(n)$.

C-9.39) Hint Start by using the bottom-up construction.

C-9.40) Hint Process elements one at a time, always storing the largest k that you have seen.

C-9.40) Solution Maintain a minimum-oriented heap with maximum size k. Start by inserting the first k numbers, and from that point on, if the next number is greater than the smallest number in the heap, then remove the smallest number and then insert the new number. There will be at most $2n$ heap operations, each of which takes $O(\log k)$ time since the heap has at most k entries.

C-9.41) Hint Write a short method that computes the number of 1's in the binary expansion of an integer by using the bitwise "and" operation.

C-9.41) Solution

```
public class integerComparator implements Comparator<Integer> {
  private int countBits(int k) {
    int numOnes = 0;
    int tmp = k;
    while (tmp != 0) {
      int bit = tmp & 1;
      if (bit == 1)
        numOnes++;
      tmp = tmp >> 1;
    }
    return numOnes;
  }

  public int compare(Integer a, Integer b) {
    int valA = countBits(a);
    int valB = countBits(b);
    return (valA - valB);
  }
}
```

C-9.42) Hint Replace all use of operators $>$, $<$, and $==$, with the appropriate call to the comparator's compare method.

C-9.43) Hint Create a new key type internally that wraps the provided keys to invert comparisons.

C-9.44) Hint Partition the array into a sorted part and an unsorted part and use swaps to move elements around.

C-9.45) Hint Partition the array into a sorted part and an unsorted part and use swaps to move elements around.

C-9.45) Solution Note well that the insertion sort implementation given in Code Fragment 3.6 suffices (although not written for a generic type).

C-9.46) Hint Use the right portion of the array to store the heap.

C-9.47) Hint You will need two data structures that somehow keep "links" between each other.

C-9.47) Solution Unmonopoly can be played efficiently using two adaptable priority queues (with location-aware entries), one keeping track of the player with minimal amount of money and the other keeping track of the player with the most. Each turn involves pairing up the minimal and maximal elements, redistributing their wealth, and then updating their keys

(in both priority queues). These structures allow constant-time pairing of the minimal and maximal elements and fast logarithmic-time updating of keys.

C-9.48) Hint You will need two priority queues.

C-9.49) Hint Use adaptable priority queues.

Projects

P-9.50) Hint Use as large of inputs as you can for experimentation.

P-9.51) Hint Experiment with for which values of k your new implementation outperforms the original.

P-9.52) Hint Decide early how you are going to implement the "pointers" for your location-aware entries.

P-9.53) Hint Mimic the animation style provided in the book.

P-9.54) Hint Use a heap for the priority queue.

P-9.55) Hint Use a heap for the CPU job priority queue.

P-9.56) Hint Study again the bottom-up heap construction algorithm.