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

Study Guide: Hints to Exercises

WILEY

Chapter 7 List and Iterator ADTs

Hints

Reinforcement

R-7.1) Draw the state of the array after each operation using a pencil with a good eraser.

R-7.2) Use the size method to help keep track of the top of stack.

R-7.3) Use the obvious correspondance.

R-7.4) Explain, in particular, why add and remove both run in linear time.

R-7.5) Notice that the existing resize method can be used to shrink the array.

R-7.6) Now we are charging more for the growing, so we need to store more cyber-dollars with each element. Calculate the number needed for growing and this will help you determine the number you need to save, which in turn tells you one less than you need to charge each push.

R-7.7) Consider how much cyber "money" is saved up from one expansion to the next.

R-7.8) Consider the running time for adding the ith element, and characterize the total running time using a summation.

R-7.9) You cannot use the existing resize method, as written.

R-7.10) Change the push method so that it resizes the array when needed.

R-7.11) Exploit the ability of going before or after positions you have access to through the positional methods, but be sure to check for the empty-list case.

R-7.12) Count the steps while traversing the list until encountering position p.

R-7.13) Remember to use the equals method to test equality.

R-7.14) Try executing such a command.

R-7.15) Carefully map the public methods of the queue interface to the concrete behaviors of the LinkedPositionalList class.

R-7.16) You may do this as a snapshot or as a cursor in the current list. In either case, describe the pointer hops needed to implement the hasNext and next methods.

R-7.17) For an infinite progression, you may have hasNext( ) return **true**.

R-7.18) Remember to use the equals method to test equality.

R-7.19) For good measure, set all references to null.

R-7.20) Model your solution off the book's example of shuffling an array.

R-7.21) Implement the move-to-front using a pencil and eraser.

R-7.22) Consider the two extreme cases of how we could distribute m accesses across n elements.

R-7.23) The first should be last, both physically and in terms of how long ago it has been accessed.

R-7.24) You might wish to add functionality to the nested Item class.

Creativity

C-7.25) Think about how to extend the circular array implementation of the queue ADT given in the previous chapter.

C-7.26) We suggest realigning the front of the list with index 0 during a resize operation.

C-7.27) Make sure that the clone has its own array.

C-7.28) Consider randomly shuffling the deck one card at a time.

C-7.29) The existing resize method can be used to shrink the array.

C-7.30) You can predict precisely when a resize occurs during this process, and take the sum of those costs.

C-7.31) Apply the amortized analysis accounting technique using a monetary accounting scheme with extra funds for both insertions and removals.

C-7.32) Consider, after any resize takes place, how many subsequent operations are needed to force another resize.

C-7.33) Try to oscillate between growing and shrinking.

C-7.34) Consider using one stack to collect incoming elements, and another as a buffer for elements to be delivered.

C-7.35) We suggest realigning the front of the queue with index 0 during a resize operation.

C-7.36) Code Fragment 7.6 demonstrates a traversal of a positional list.

C-7.37) Compare to size( )/2.

C-7.38) Be careful of boundary conditions.

C-7.39) Find the syntax to remove and reinsert the element.

C-7.40) Draw pictures of the linked list operations needed to move p to the front.

C-7.41) Compare to the clone method provided for the SinglyLinkedList in Section 3.6.2.

C-7.42) You obviously need to switch next and prev pointers, but make sure you do it in the right order.

C-7.43) It is okay to be inefficient in this case.

C-7.44) Convert the two parts to two separate lists as sublists.

C-7.45) Have each position keep track of what list it is in.

C-7.46) Watch out for the special case when p and q are neighbors.

C-7.47) There is a trade-off between insertion and searching depending on whether the entries in L are sorted.

C-7.48) You should replace the first() and last() methods with a method abstracting the cursor.

C-7.49) Reread Section 7.5.1 carefully.

C-7.50) Have each iterator keep an instance variable that can be compared with a similar variable for the underlying list to determine if the list has changed since the iterator was created.

C-7.51) Avoid index-based operations.

C-7.52) You can either use index-based methods or positional ones, but the index-based ones would be more intuitive.

C-7.53) Think about maintaining the sequence of the last n accesses in addition to the list L itself.

C-7.54) For this lower bound, assume that when an element is accessed we search for it by traversing the list starting at the front.

C-7.55) Be sure to handle the case where every pair (x,y) in A and every pair (y,z) in B have the same y value.

C-7.56) You should be able to achieve $O(n)$ time.

C-7.57) You may wish to consider the coverage of insertion-sort on an array from an earlier chapter.

Projects

P-7.58) Get coding!

P-7.59) Review Exercise C-7.45 and its hint.

P-7.60) Keep all cards in a single list, and four positions to demark the beginning of the respective suits.

P-7.61) Use a position instance variable to keep track of the cursor location.