7.8 Exercises

Reinforcement

R-7.1 Draw a representation, akin to Example 7.1, of an initially empty list L after performing the following sequence of operations: add(0, 4), add(0, 3), add(0, 2), add(2, 1), add(1, 5), add(1, 6), add(3, 7), add(0, 8).

R-7.2 Give an implementation of the stack ADT using an array list for storage.

R-7.3 Give an implementation of the deque ADT using an array list for storage.

R-7.4 Give a justification of the running times shown in Table 7.1 for the methods of an array list implemented with a (nonexpanding) array.

R-7.5 The java.util.ArrayList includes a method, trimToSize( ), that replaces the underlying array with one whose capacity precisely equals the number of elements currently in the list. Implement such a method for our dynamic version of the ArrayList class from Section 7.2.

R-7.6 Redo the justification of Proposition 7.2 assuming that the the cost of growing the array from size k to size $2k$ is $3k$ cyber-dollars. How much should each push operation be charged to make the amortization work?

R-7.7 Consider an implementation of the array list ADT using a dynamic array, but instead of copying the elements into an array of double the size (that is, from N to $2N$) when its capacity is reached, we copy the elements into an array with $\lceil N/4 \rceil$ additional cells, going from capacity N to $N + \lceil N/4 \rceil$. Show that performing a sequence of n push operations (that is, insertions at the end) still runs in $O(n)$ time in this case.

R-7.8 Suppose we are maintaining a collection C of elements such that, each time we add a new element to the collection, we copy the contents of C into a new array list of just the right size. What is the running time of adding n elements to an initially empty collection C in this case?

R-7.9 The add method for a dynamic array, as described in Code Fragment 7.5, has the following inefficiency. In the case when a resize occurs, the resize operation takes time to copy all the elements from the old array to a new array, and then the subsequent loop in the body of add shifts some of them to make room for a new element. Give an improved implementation of the add method, so that, in the case of a resize, the elements are copied into their final place in the new array (that is, no shifting is done).

R-7.10 Reimplement the ArrayStack class, from Section 6.1.2, using dynamic arrays to support unlimited capacity.

R-7.11 Describe an implementation of the positional list methods addLast and addBefore realized by using only methods in the set {isEmpty, first, last, before, after, addAfter, addFirst}.

R-7.12 Suppose we want to extend the PositionalList abstract data type with a method, indexOf(p), that returns the current index of the element stored at position p. Show how to implement this method using only other methods of the PositionalList interface (not details of our LinkedPositionalList implementation).

R-7.13 Suppose we want to extend the PositionalList abstract data type with a method, findPosition(e), that returns the first position containing an element equal to e (or null if no such position exists). Show how to implement this method using only existing methods of the PositionalList interface (not details of our LinkedPositionalList implementation).

R-7.14 The LinkedPositionalList implementation of Code Fragments 7.9–7.12 does not do any error checking to test if a given position p is actually a member of the relevant list. Give a detailed explanation of the effect of a call L.addAfter(p, e) on a list L, yet with a position p that belongs to some other list M.

R-7.15 To better model a FIFO queue in which entries may be deleted before reaching the front, design a LinkedPositionalQueue class that supports the complete queue ADT, yet with enqueue returning a position instance and support for a new method, remove(p), that removes the element associated with position p from the queue. You may use the adapter design pattern (Section 6.1.3), using a LinkedPositionalList as your storage.

R-7.16 Describe how to implement a method, alternateIterator( ), for a positional list that returns an iterator that reports only those elements having even index in the list.

R-7.17 Redesign the Progression class, from Section 2.2.3, so that it formally implements the Iterator<long> interface.

R-7.18 The java.util.Collection interface includes a method, contains(o), that returns true if the collection contains any object that equals Object o. Implement such a method in the ArrayList class of Section 7.2.

R-7.19 The java.util.Collection interface includes a method, clear( ), that removes all elements from a collection. Implement such a method in the ArrayList class of Section 7.2.

R-7.20 Demonstrate how to use the java.util.Colletions.reverse method to reverse an array of objects.

R-7.21 Given the set of element $\{a,b,c,d,e,f\}$ stored in a list, show the final state of the list, assuming we use the move-to-front heuristic and access the elements according to the following sequence: $(a,b,c,d,e,f,a,c,f,b,d,e)$.

R-7.22 Suppose that we have made kn total accesses to the elements in a list L of n elements, for some integer $k \geq 1$. What are the minimum and maximum number of elements that have been accessed fewer than k times?

R-7.23 Let L be a list of n items maintained according to the move-to-front heuristic. Describe a series of $O(n)$ accesses that will reverse L.

R-7.24 Implement a resetCounts( ) method for the FavoritesList class that resets all elements' access counts to zero (while leaving the order of the list unchanged).

Creativity

C-7.25 Give an array-based list implementation, with fixed capacity, treating the array circularly so that it achieves $O(1)$ time for insertions and removals at index 0, as well as insertions and removals at the end of the array list. Your implementation should also provide for a constant-time get method.

C-7.26 Complete the previous exercise, except using a dynamic array to provide unbounded capacity.

C-7.27 Modify our ArrayList implementation to support the Cloneable interface, as described in Section 3.6.

C-7.28 In Section 7.5.3, we demonstrated how the Collections.shuffle method can be adapted to shuffle a reference-type array. Give a direct implementation of a shuffle method for an array of **int** values. You may use the method, nextInt(n) of the Random class, which returns a random number between 0 and $n-1$, inclusive. Your method should guarantee that every possible ordering is equally likely. What is the running time of your method?

C-7.29 Revise the array list implementation given in Section 7.2.1 so that when the actual number of elements, n, in the array goes below $N/4$, where N is the array capacity, the array shrinks to half its size.

C-7.30 Prove that when using a dynamic array that grows and shrinks as in the previous exercise, the following series of $2n$ operations takes $O(n)$ time: n insertions at the end of an initially empty list, followed by n deletions, each from the end of the list.

C-7.31 Give a formal proof that any sequence of n push or pop operations (that is, insertions or deletions at the end) on an initially empty dynamic array takes $O(n)$ time, if using the strategy described in Exercise C-7.29.

C-7.32 Consider a variant of Exercise C-7.29, in which an array of capacity N is resized to capacity precisely that of the number of elements, any time the number of elements in the array goes strictly below $N/4$. Give a formal proof that any sequence of n push or pop operations on an initially empty dynamic array takes $O(n)$ time.

C-7.33 Consider a variant of Exercise C-7.29, in which an array of capacity N, is resized to capacity precisely that of the number of elements, any time the number of elements in the array goes strictly below $N/2$. Show that there exists a sequence of n push and pop operations that requires $\Omega(n^2)$ time to execute.

C-7.34 Describe how to implement the queue ADT using two stacks as instance variables, such that all queue operations execute in amortized $O(1)$ time. Give a formal proof of the amortized bound.

C-7.35 Reimplement the ArrayQueue class, from Section 6.2.2, using dynamic arrays to support unlimited capacity. Be especially careful about the treatment of a circular array when resizing.

C-7.36 Suppose we want to extend the PositionalList interface to include a method, positionAtIndex(i), that returns the position of the element having index i (or throws an IndexOutOfBoundsException, if warranted). Show how to implement this method, using only existing methods of the PositionalList interface, by traversing the appropriate number of steps from the front of the list.

C-7.37 Repeat the previous problem, but use knowledge of the size of the list to traverse from the end of the list that is closest to the desired index.

C-7.38 Explain how any implementation of the PositionalList ADT can be made to support all methods of the List ADT, described in Section 7.1, assuming an implementation is given for the positionAtIndex(i) method, proposed in Exercise C-7.36.

C-7.39 Suppose we want to extend the PositionalList abstract data type with a method, moveToFront(p), that moves the element at position p to the front of a list (if not already there), while keeping the relative order of the remaining elements unchanged. Show how to implement this method using only existing methods of the PositionalList interface (not details of our LinkedPositionalList implementation).

C-7.40 Redo the previous problem, but providing an implementation within the class LinkedPositionalList that does not create or destroy any nodes.

C-7.41 Modify our LinkedPositionalList implementation to support the Cloneable interface, as described in Section 3.6.

C-7.42 Describe a nonrecursive method for reversing a positional list represented with a doubly linked list using a single pass through the list.

C-7.43 Page 281 describes an *array-based* representation for implementing the positional list ADT. Give a pseudocode description of the addBefore method for that representation.

C-7.44 Describe a method for performing a ***card shuffle*** of a list of $2n$ elements, by converting it into two lists. A card shuffle is a permutation where a list L is cut into two lists, L_1 and L_2, where L_1 is the first half of L and L_2 is the second half of L, and then these two lists are merged into one by taking the first element in L_1, then the first element in L_2, followed by the second element in L_1, the second element in L_2, and so on.

C-7.45 How might the LinkedPositionalList class be redesigned to detect the error described in Exercise R-7.14.

C-7.46 Modify the LinkedPositionalList class to support a method swap(p, q) that causes the underlying nodes referenced by positions p and q to be exchanged for each other. Relink the existing nodes; do not create any new nodes.

C-7.47 An array is ***sparse*** if most of its entries are **null**. A list L can be used to implement such an array, A, efficiently. In particular, for each nonnull cell $A[i]$, we can store a pair (i, e) in L, where e is the element stored at $A[i]$. This approach allows us to represent A using $O(m)$ storage, where m is the number of nonnull entries in A. Describe and analyze efficient ways of performing the methods of the array list ADT on such a representation.

C-7.48 Design a circular positional list ADT that abstracts a circularly linked list in the same way that the positional list ADT abstracts a doubly linked list.

C-7.49 Provide an implementation of the listiterator( ) method, in the context of the class LinkedPositionalList, that returns an object that supports the java.util.ListIterator interface described in Section 7.5.1.

C-7.50 Describe a scheme for creating list iterators that ***fail fast***, that is, they all become invalid as soon as the underlying list changes.

C-7.51 There is a simple algorithm, called ***bubble-sort***, for sorting a list L of n comparable elements. This algorithm scans the list $n-1$ times, where, in each scan, the algorithm compares the current element with the next one and swaps them if they are out of order. Give a pseudocode description of bubble-sort that is as efficient as possible assuming L is implemented with a doubly linked list. What is the running time of this algorithm?

C-7.52 Redo Exercise C-7.51 assuming L is implemented with an array list.

C-7.53 Describe an efficient method for maintaining a favorites list L, with the move-to-front heuristic, such that elements that have not been accessed in the most recent n accesses are automatically purged from the list.

C-7.54 Suppose we have an n-element list L maintained according to the move-to-front heuristic. Describe a sequence of n^2 accesses that is guaranteed to take $\Omega(n^3)$ time to perform on L.

C-7.55 A useful operation in databases is the ***natural join***. If we view a database as a list of *ordered* pairs of objects, then the natural join of databases A and B is the list of all ordered triples (x,y,z) such that the pair (x,y) is in A and the pair (y,z) is in B. Describe and analyze an efficient algorithm for computing the natural join of a list A of n pairs and a list B of m pairs.

C-7.56 When Bob wants to send Alice a message M on the Internet, he breaks M into n ***data packets***, numbers the packets consecutively, and injects them into the network. When the packets arrive at Alice's computer, they may be out of order, so Alice must assemble the sequence of n packets in order before she can be sure she has the entire message. Describe an efficient scheme for Alice to do this. What is the running time of this algorithm?

C-7.57 Implement the FavoritesList class using an array list.

Projects

P-7.58 Develop an experiment, using techniques similar to those in Section 4.1, to test the efficiency of n successive calls to the add method of an ArrayList, for various n, under each of the following three scenarios:

a. Each add takes place at index 0.
b. Each add takes place at index size( )/2.
c. Each add takes place at index size( ).

Analyze your empirical results.

P-7.59 Reimplement the LinkedPositionalList class so that an invalid position is reported in a scenario such as the one described in Exercise R-7.14.

P-7.60 Implement a CardHand class that supports a person arranging a group of cards in his or her hand. The simulator should represent the sequence of cards using a single positional list ADT so that cards of the same suit are kept together. Implement this strategy by means of four "fingers" into the hand, one for each of the suits of hearts, clubs, spades, and diamonds, so that adding a new card to the person's hand or playing a correct card from the hand can be done in constant time. The class should support the following methods:

- addCard(r, s): Add a new card with rank r and suit s to the hand.
- play(s): Remove and return a card of suit s from the player's hand; if there is no card of suit s, then remove and return an arbitrary card from the hand.
- iterator( ): Return an iterator for all cards currently in the hand.
- suitIterator(s): Return an iterator for all cards of suit s that are currently in the hand.

P-7.61 Write a simple text editor, which stores and displays a string of characters using the positional list ADT, together with a cursor object that highlights a position in the string. The editor must support the following operations:

- left: Move cursor left one character (do nothing if at beginning).
- right: Move cursor right one character (do nothing if at end).
- insert c: Insert the character c just after the cursor.
- delete: Delete the character just after the cursor (if not at end).

Chapter Notes

The treatment of data structures as collections (and other principles of object-oriented design) can be found in object-oriented design books by Booch [16], Budd [19], and Liskov and Guttag [67]. Lists and iterators are pervasive concepts in the Java Collections Framework. Our positional list ADT is derived from the "position" abstraction introduced by Aho, Hopcroft, and Ullman [6], and the list ADT of Wood [96]. Implementations of lists via arrays and linked lists are discussed by Knuth [60].