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 7 List and Iterator ADTs

Hints and Solutions

Reinforcement

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

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

R-7.3) Hint Use the obvious correspondance.

R-7.3) Solution

Deque Method	*Realization with Array List Methods*
size()	size()
isEmpty()	isEmpty()
first()	get(0)
last()	get(size() − 1)
addFirst(e)	add(0, e)
addLast(e)	add(size(), e)
removeFirst()	remove(0)
removeLast()	remove(size() − 1)

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

R-7.4) Solution The methods *size*() and *isEmpty*() run in constant time because of the instance variable n. This variable can easily be returned or compared to give the results for these two methods. The method $elemAtRank(i)$ is also constant because of the underlying array implementation of the array list. Since we are using an array, an access at a particular index is done with a simple $A[i]$ reference. The $replace(i,e)$ method can also be done with a quick access to an index and a change of element, thus, it is also constant. However, the methods *add* and *remove* take linear time due to the nature of an array. When inserting at certain index, we must first move

over (increase the index of) every element that will follow (has a higher index than) this element. This can take up to n moves for an array of size n. Likewise, for a removal, all of the elements that followed the removed element in the array list previous to the removal now need to be shifted down (lowered in index). Again, this can take up to n moves.

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

R-7.5) Solution

```
public void trimToSize( ) {
  if (data.length != size)
    resize(size);
}
```

R-7.6) Hint 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-7.6) Solution Let us assume that one cyber-dollar is enough to pay for the execution of each push operation in S, excluding the time spent for growing the array. Now, however, growing the array from size k to size $2k$ requires $3k$ cyber-dollars. Once again, we will need to account for this cost with our "bank account" in the elements of the last half of the array list. To grow from 2^i to 2^{i+1}, we need $3 * 2^i$ cyber dollars. Thus, from the second half of the array list—the last 2^{i-1} elements we need to have 6 cyber-dollars apiece stored away. So, overall, we need to charge 7 cyber-dollars for each push operation: 6 for future growth and 1 for insertion.

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

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

R-7.8) Solution The total running time for adding n elements in this way is $O(n^2)$.

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

R-7.9) Solution

```
public void add(int i, E e) {
  checkIndex(i, size + 1);
  if (size == data.length) {
    E[ ] temp = (E[ ]) new Object[2*data.length];
    for (int k=0; k < i; k++)
        temp[k] = data[k];
    temp[i] = e;
    for (int k=i+1; k < size+1; k++)
      temp[k] = data[k1];
    data=temp;
  } else {
    for (int k=size−1; k >= i; k==)
      data[k+1] = data[k];
    data[i] = i;
  }
  size++;
}
```

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

R-7.11) Hint 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-7.11) Solution

```
public Position<E> addLast(E e) {
  if (isEmpty( ))
    return addFirst(e);
  else
    return addAfter(last( ), e);
}

public Position<E> addBefore(Position<E> p, E e) {
  if (p == first( ))
    return addFirst(e);
  else
    return addAfter(before(p), e);
}
```

(We don't actually need to use the after(p) method.)

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

R-7.12) Solution

```
public int indexOf(Position<E> p) {    // assuming p is valid
  int count;
  Position<E> walk = first( );
  while (walk != p) {
    count++;
    walk = after(walk);
  }
  return count;
}
```

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

R-7.13) Solution

```
public Position<E> findPosition(E e) {
  Position<E> walk = first( );
  while (walk != null && !e.equals(walk.getElement( )))
    walk = after(walk);
  return walk;
}
```

R-7.14) Hint Try executing such a command.

R-7.14) Solution This causes an internal inconsistency for both the list upon which it is invoked and the list to which p belongs. A node containing the new element will be linked into the list to which p belongs, but the size variable will be updated for the list upon which the method is invoked.

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

R-7.16) Hint 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) Hint For an infinite progression, you may have hasNext( ) return **true**.

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

R-7.18) Solution

```
public boolean contains(Object o) {
  for (int k=0; k < size; k++)
    if (data[k].equals(o))
      return true;
  return false;
}
```

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

R-7.19) Solution

```
public void clear( ) {
  for (int k=0; k < size; k++)
    data[k] = null;
  size = 0;
}
```

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

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

R-7.21) Solution $\{e,d,b,f,c,a\}$

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

R-7.22) Solution The minimum is 0, since we could have accessed each element exactly k times. the maximum is $n-1$, since we could have accessed just one element kn times.

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

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

R-7.24) Solution If we do not provide a setCount method for the Item class, an alternate approach is to create new Item instances for the reset.

```
public void resetCounts( ) {
  Position<Item<E>> walk = list.first( );
  while (walk != null) {
    list.set(walk, new Item<E>(walk.getElement( ).getValue( )));
    walk = list.after(walk);
  }
}
```

Creativity

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

C-7.25) Solution As we did with the ArrayQueue class, we recommend maintaining index f to the front of the list and size, which is the current number of elements. The element at index k in the list will be found at index (f+k) % data.length in the underlying data array. Insertions and removals can be processed at either end in constant time.

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

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

C-7.27) Solution

```
public ArrayList<E> clone( ) throws CloneNotSupportedException {
  ArrayList<E> other = (ArrayList<E>) super.clone( );
  other.data = data.clone( );
  return other;
}
```

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

C-7.28) Solution For each i from 0 to $n-1$, swap the element at index i with a randomly chosen element from the array list.

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

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

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

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

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

C-7.33) Solution This problem is different from the previous, because the threshold of N/2 is unfortunately aligned with the doubling rule used for an increase. Assume that an underlying array has capacity k for large k. When we push a $(k+1)$st element, the array will be doubled to capacity $2k$, at a cost of $\Omega(k)$. If we then pop two elements, the $k-1$ remaining elements is less than half of the array's capacity, so a resize to shrink the array will be performed with cost of $\Omega(k)$. By repeatedly pushing two elements and popping two elements, there will be a linear amount of work for every two operations.

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

C-7.34) Solution Consider the following implementation.

```
public <E> class SolnQueue {
  private Stack<E> incoming = new Stack<>();
  private Stack<E> outgoing = new Stack<>();

  private void transfer() {
    while (!incoming.isEmpty())
      outgoing.push(incoming.pop());
  }

  public int size() {
    return incoming.size() + outgoing.size();
  }

  public boolean isEmpty() {
    return (incoming.isEmpty() && outgoing.isEmpty());
  }

  public void enqueue(E e) {
    incoming.push(e);
  }

  public E front() {
    if (isEmpty()) return null
    if (outgoing.isEmpty())
      transfer();
    return outgoing.top();
  }

  public E dequeue() {
    if (isEmpty()) return null
    if (outgoing.isEmpty())
      transfer();
    return outgoing.pop();
  }
}
```

The correctness can be proven by arguing that elements in the outgoing stack are ordered with the older elements toward the top, and that all elements in the outgoing stack are older than any elements in the incoming stack.

Although the worst-case running time for front and dequeue are $O(n)$ due to a transfer, the $O(1)$ amortized bound can be seen from the fact that

any given element participates in at most one transfer in its lifetime, and thus the total time spent in the transfer loop during a sequence of k queue operations is k.

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

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

C-7.36) Solution

```
public Position<E> positionAtIndex(index i) {
  if (i < 0 || i >= size()) throw new IndexOutOfBoundsExcpetion("Invalid index");
  Position<E> walk = first();
  for (int k=0; k < i; k++)
    walk = after(walk);
  return walk;
}
```

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

C-7.38) Hint Be careful of boundary conditions.

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

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

C-7.40) Solution

```
public void moveToFront(Position<E> p) {
  Node<E> node = validate(p);
  if (node != header.getNext()) {
    // remove node from its current location
    node.getPrev().setNext(node.getNext());
    node.getNext().setPrev(node.getPrev());
    // insert node at front of list (just after header)
    node.setPrev(header);
    node.setNext(header.getNext());
    header.getNext().setPrev(node);
    header.setNext(node);
  }
}
```

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

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

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

C-7.43) Solution

Algorithm addBefore(p, e):
  Let i be the index associated with position p
  for j from size-1 down to i **do**
    Move the entry at index j to index $j+1$
    Update that position to note its new index $j+1$
  Create a new entry for element e and insert at index i

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

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

C-7.45) Solution Since we would not want to walk an entire list to verify that a position belongs to it, a better approach, if such robustness is desired, is to add a new field to each node which is a reference to the list to which it belongs. That can be initialized when a node is added to the list, and should be set back to null when a node is removed from the list (to denote it as deprecated).

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

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

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

C-7.49) Hint Reread Section 7.5.1 carefully.

C-7.50) Hint 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-7.50) Solution Each list can maintain an integer value that is incremented each time a method is called that modifies the state of the list in a way that should cause iterators to fail. Each iterator should keep a variable which equals that value the list had when the iterator was created. Then, whenever the iterator is used in an operation, if the current value stored by the list is not the same as that stored by the iterator, the iterator should fail.

C-7.51) Hint Avoid index-based operations.

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

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

C-7.53) Solution We will maintain a global count of the number of accesses in the sequence, and then with each element in the list, we will

keep track of the time stamp for its most recent access. After each operation is performed, we examine the last item in the list and if its most recent access is no longer within the most recent n operations, that element is deleted. (Note that only one element can be purged after a single operation, as it can be the only one who's most recent access was precisely $n+1$ operations ago.)

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

C-7.54) Solution For this lower bound, we assume that when an element is accessed, we must search for it by traversing the list starting at the front. We consider n elements, and access them each once in the order $1, 2, \ldots n$, and then access them each again in the original order, repeating n such phases of access. Other than when each element is originally inserted, each of the remaining accesses is to the element at the end of the list, and thus we have $n(n-1)$ accesses which take n steps, thus $\Omega(n^3)$ time overall.

C-7.55) Hint 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) Hint You should be able to achieve $O(n)$ time.

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

Projects

P-7.58) Hint Get coding!

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

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

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