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 3 Fundamental Data Structures

Hints and Solutions

Reinforcement

R-3.1) Hint Use a calculator to aid in the arithmetic.

R-3.2) Hint You have to have your random number select a random index in the array, so be sure to keep track of the number, n, of entries in the array and do not index past index $n-1$.

R-3.3) Hint The alphabets for most alphabet-based languages are included in the Unicode character encoding standard.

R-3.3) Solution The alpha array and the various uses of value 26 would have be changed to the new alphabet and its size. In addition, all the places that use the literal A to refer to the first letter would now have to be changed to the first letter in the new alphabet. In this case, it would be better to define a final static int FIRSTLETTER, set it to the first letter, and use it instead of A. This assumes the messages are still in upper-case, of course.

R-3.4) Hint You may want to add a new instance variable to track if the game has been completed.

R-3.5) Hint Make the modification in the code and test it.

R-3.5) Solution There are no negative consequences if removing those lines (in fact there would be an every so slight increase in efficiency without). While the internal state may seem inconsistent with tail referencing a defunct node, that tail reference is never accessed for an empty list, so the inconsistency has no effect.

R-3.6) Hint It is okay to have an algorithm running in linear time.

R-3.6) Solution

```
private Node<E> penultimate( ) {
  if (size < 2)
     throw new IllegalStateException("list must have 2 or more entries");
  Node<E> walk = head;
  while (walk->next->next != null)
    walk = walk->next;
  return walk;
}
```

R-3.7) Hint There exists a one-line solution.

R-3.7) Solution

```
tail.setNext(new Node<>(e, tail.getNext( )))
```

R-3.8) Hint Consider a combined search from both ends. Also, recall that a link hop is an assignment of the form "p = p.getNext( );" or "p = p.getPrev( );".

R-3.8) Solution The following method runs in $O(n)$ time.

```
private Node<E> middle( ) {
  if (size == 0)
     throw new IllegalStateException("list must be nonempty");
  Node<E> middle = header->next
  Node<E> partner = trailer->prev;
  while (middle != partner && middle->next != partner) {
    middle = middle.getNext( );
    partner = partner.getPrev( );
  }
  return middle;
}
```

R-3.9) Hint Use a loop to traverse the list while counting.

R-3.9) Solution

```
public int size( ) {
  int count=0;
  Node<E> walk = head;
  while (walk != null) {
    count++;
    walk = walk.getNext( );
  }
  return count;
}
```

R-3.10) Hint You need to keep track of where you start or your method will have an infinite loop.

R-3.10) Solution

```
public int size( ) {
  if (tail == null) return 0;
  Node<E> walk = tail->next;
  int count=1;
  while (walk != tail) {
    count++;
    walk = walk.getNext( );
  }
  return count;
}
```

R-3.11) Hint Do not include the sentinels in the count.

R-3.11) Solution

```
public int size( ) {
  int count=0;
  Node<E> walk = header->next;
  while (walk != trailer) {
    count++;
    walk = walk.getNext( );
  }
  return count;
}
```

R-3.12) Hint Carefully relink existing nodes.

R-3.13) Hint Recall that a two-dimensional array in Java is really a one-dimensional array such that each entry is itself a reference to a one-dimensional array.

R-3.14) Hint Recall the ways for doing array copying in the java.util.Arrays class.

R-3.14) Solution It seems a stretch to come up with three meaningful ways, but technically each of the following will suffice:

```
backup = original.clone( );
backup = java.util.Arrays.copyOf(original, backup.length);
backup = java.util.Arrays.copyOfRange(original, 0, backup.length);
```

R-3.15) Hint You can rely on the size variable to walk the correct number of steps when traversing the lists.

R-3.16) Hint The sentinels are irrelevant to the equivalence of two lists.

Creativity

C-3.17) Hint You don't need to sort A.

C-3.17) Solution

```
public int missing(int[ ] A) {
  boolean[ ] found = new boolean[A.length]; // false, by default
  for (int val : A)
    if (found[val])
      return val;
    else
      found[val] = true;
  return -1;  // shouldn't happen if input as expected
}
```

C-3.18) Hint It might help to sort B.

C-3.18) Solution Sort the array B then scan it from front to end looking for the repeated entries. Each pair of repeated integers will be consecutive in the sorted listing.

C-3.19) Hint Add items at the "end" of the contiguous run of objects in the array. For removing an object, consider first swapping it with the object at index $n-1$.

C-3.19) Solution

```
public void add(GameEntry e) {
  if (numEntries < board.length) {
    board[numEntries] = e;
    numEntries++;
  } // otherwise no room for new entry
}

public GameEntry remove(int i) throws IndexOutOfBoundsException {
  if (i < 0 || i >= numEntries)
    throw new IndexOutOfBoundsException("Invalid index: `` + i);
  GameEntry temp = board[i];
  if (i != numEntries - 1)
    board[i] = board[numEntries - 1];
  board[numEntries-1] = null;
  numEntries--;
  return temp;
}
```

C-3.20) Hint Imagine what would happen if a = 1.

C-3.21) Hint Recall the definition of the java.util.nextInt method and note that one of the values returned has a special property with respect to multiplication.

C-3.21) Solution The product will be 0 with probability $1-0.9^{100}$, which is much larger than 0.99, since the only way the product will not be zero is if no element of A is zero.

C-3.22) Hint Randomly choose the first element, then the second, and so on.

C-3.23) Hint You might want to consider using a two-dimensional array.

C-3.23) Solution Define an $(n+1) \times (n+1)$ two-dimensional boolean array M, which will record the meeting pairs. So that $M[i,j] =$ **true** if and only if player i and j have meet. In addition, define a one-dimensional integer array c of size $n+1$ so that $c[i]$ is a count of the number of other players that player i has met. When a player i and j meet, we check $M[i,j]$, and if it is **true**, then we are done—the players have already met. Otherwise, we set $M[i,j]$ and $M[j,i]$ to **true** and we increment $c[i]$ and $c[j]$. If either $c[i]$ or $c[j]$ equals $n-1$ after this, then we have a winner.

C-3.24) Hint The entries $A[i][j][k]$ and $B[i][j][k]$ are the ones that need to be added.

C-3.25) Hint This concatenation operation need not search all of L and M.

C-3.25) Solution Simply use a temporary node to walk to the end of list L. Then, make the last element of L point to the first element of M as its "next" node.

```
Concatenate(L,M)
    Create a new node v
    v = L.getHead( )
    while v.getNext( ) != null do
        v = v.getNext( )
    v.setNext(M.getHead( ))
    L' = L
```

The number of steps is proportional to the size of L.

C-3.26) Hint Splice the end of L into the beginning of M.

C-3.26) Solution Use two temporary Node elements, temp1 and temp2. Initialize temp1 to be the trailer node of L and temp2 to be the header node of M. Make the element of temp1 have its next field point to temp2 and set the element of temp2 to have its prev field point to temp1. Set L' to be L and then set the trailer node of L' to be the trailer node of M.

C-3.27) Hint Performing the swap for a singly linked list will take longer than for a doubly linked list.

C-3.27) Solution Implementing a precise solution takes great care, especially when x and y neighbor each other. However, the issue regarding efficiency is that for swapping x and y in a singly-linked, we must locate the nodes immediately preceding x and y, and we have no quick way to do so.

C-3.28) Hint Consider changing the orientation of links while making a single pass through the list.

C-3.28) Solution Such a method of the SinglyLinkedList class could be implemented as follows.

```
public void reverse( ) {
  Node<E> prev = null;
  Node<E> walk = head;
  while (walk != null) {
    Node<E> adv = walk.getNext( );
    walk.setNext(prev);
    prev = walk;
    walk = adv;
  }
  head = prev;
}
```

Note well that the above implementation works, even for trivial lists.

C-3.29) Hint Try to find a matching alignment for the first node of one list.

C-3.30) Hint You are going to have to keep two cursors and count around L.

C-3.31) Hint Adjust the constructor to properly initialize the sentinel.

C-3.32) Hint Blend techniques seen in the existing CircularlyLinkedList and DoublyLinkedList.

C-3.33) Hint You will need to add a prev reference to the node, and maintain it properly whenever the list changes.

C-3.34) Hint Make sure to properly link the new chain of nodes.

C-3.35) Hint Make sure to create new sentinel nodes.

Projects

P-3.36) Hint Matrix addition is defined so that if $C = A + B$, then $C[i, j] = A[i, j] + B[i, j]$. Matrix multiplication is defined so that if $C = AB$, where A is a $c \times d$ matrix and B is a $d \times e$ matrix, then $C[i, j] = \sum_{k=0}^{d} A[i, k]B[k, j]$. That is, C is a $c \times e$ matrix.

P-3.37) Hint You should keep track of the number of game entries explicitly.

P-3.38) Hint You should keep track of the number of game entries explicitly.

P-3.39) Hint You will probably need separate encrypt and decrypt arrays for the upper- and lower-case characters.

P-3.40) Hint The original CaesarCipher implementation was already effectively a substitution cipher, with a specifically chosen encoder pattern.

P-3.41) Hint If you get the constructor to use the correct encoder string, everything else should work.

P-3.42) Hint A good way to generate a random encryption array is to start with the alphabet array. Then for each letter in this array, randomly swap it with some other letter in the array.