Lab 11 The List ADT

Goal

In this lab you will complete one application that uses the Abstract Data Type (ADT) list and will create new methods that work directly with the array and linked implementations of the list ADT. You will implement a reverse method that will reverse the order of the items in the list. You will also implement a cycle method that will move the first item in the list to the last position.

Resources

- Appendix A: Creating Classes from Other Classes
- Chapter 12: Lists
- Chapter 13: List Implementations That Use Arrays
- Chapter 14: A List Implementation That Links Data
- *docs.oracle.com/javase/8/docs/api/* —API documentation for the Java List interface

In `javadoc` directory

- *ListInterface.html*—Interface documentation for the interface ListInterface

Java Files

- *AList.java*
- *ListInterface.java*
- *Primes.java*
- *ArrayListExtensionsTest.java*
- *LList.java*
- *LinkedListExtensionsTest.java*

Introduction

The ADT list is one of the basic tools for use in developing software applications. It is an ordered collection of objects that can be accessed based on their position. Before continuing the lab you should review the material in Chapter 12. In particular, review the documentation of the interface *ListInterface.java*. While not all of the methods will be used in our applications, most of them will.

The application that you will implement is one that computes prime numbers using the Sieve of Eratosthenes.

One way to implement a list is to use an array. The other standard implementation is a linked structure. In this lab, you will take a working implementation of each basic type and create two new methods. If you have not done so already, take a moment to examine the code in *AList.java*.

Consider the code that implements the add method.

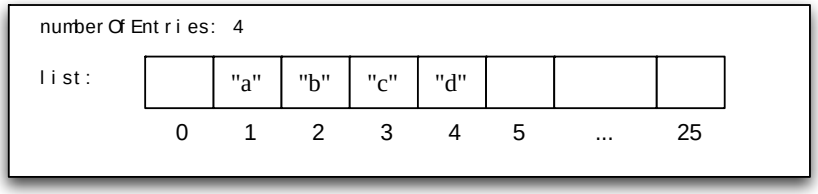
```
public void add(int newPosition, T newEntry) {
   checkInitialization();

   if ((newPosition >= 1) && (newPosition <= numberOfEntries + 1)) {
      if (newPosition <= numberOfEntries) {
         makeRoom(newPosition);
      }

      list[newPosition] = newEntry;
```

```
            numberOfEntries++;
            ensureCapacity(); // Ensure enough room for next add
        } else {
            throw new IndexOutOfBoundsException(
                                "Illegal position given to add operation.");
        }
    } // end add

    // Doubles the size of the array list if it is full.
    private void ensureCapacity() {
        int capacity = list.length - 1;

        if (numberOfEntries >= capacity) {
            int newCapacity = 2 * capacity;
            checkCapacity(newCapacity);  // Is capacity too big?
            list = Arrays.copyOf(list, newCapacity + 1);
        }
    } // end ensureCapacity


    /** Makes room for a new entry at newPosition.
     * Precondition: 1 <= newPosition <= numberOfEntries+1;
     * numberOfEntries is list's length before addition.
     * checkInitialization has been called.
     */
    private void makeRoom(int newPosition) {
        assert (newPosition >= 1) && (newPosition <= numberOfEntries + 1);
        int newIndex = newPosition;
        int lastIndex = numberOfEntries;
        // Move each entry to next higher index, starting at end of
        // array and continuing until the entry at newIndex is moved
        for (int index = lastIndex; index >= newIndex; index--) {
            list[index + 1] = list[index];
        }
    } // end makeRoom
```

Let's trace the last statement in the following code fragment.

```
AList<String> x = new AList<String>(5);
x.add("a");
x.add("b");
x.add("c");
x.add("d");
x.add(2, "x");        // trace this one
```

The initial state of the object is



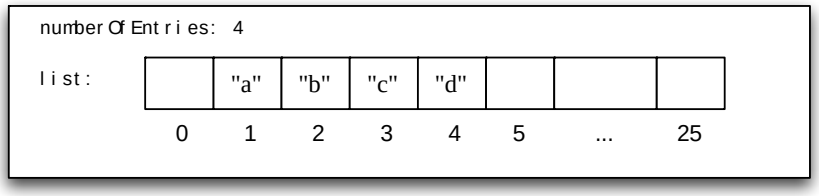
The object has been initialized, so checkInitialization() returns.

Since newPosition=2, the condition ((newPosition >= 1) && (newPosition <= numberOfEntries + 1)) is true.

The condition newPosition <= numberOfEntries is also true, so we invoke the method makeRoom(2).

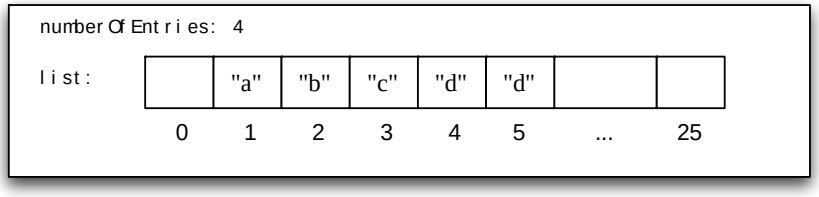
We set the local variables newIndex to 2 and lastIndex to 4 resulting in the following state.

newPosition: 2
newIndex: 2
lastIndex: 4



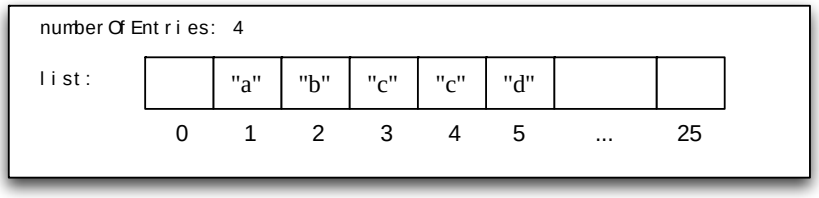
Now we do the for loop in makeRoom, we start index at lastIndex and then execute list[index + 1] = list[index]. We are in the following state.

newPosition: 2
newIndex: 2
lastIndex: 4
index: 4



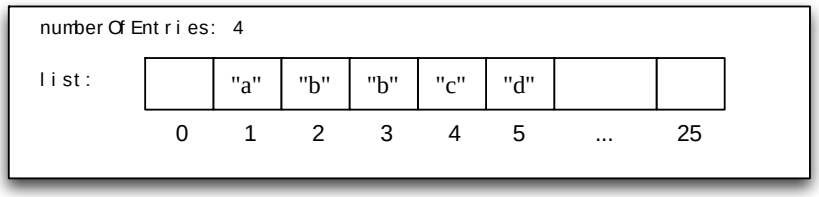
We update the loop index by sutracting 1 and then compare the result with newIndex. Since 3 is >= 2, we continue with a second execution of the body of the loop. After it is done, we are in the following state.

newPosition: 2
newIndex: 2
lastIndex: 4
index: 3



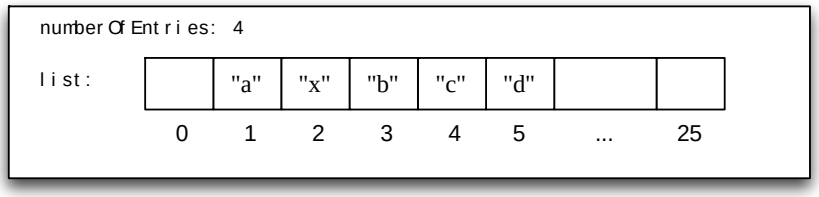
Again,we update the loop index by sutracting 1 and then compare the result with newIndex. Since 2 is >= 2, we continue with a third execution of the body of the loop. After it is done, we are in the following state.

newPosition: 2
newIndex: 2
lastIndex: 4
index: 2

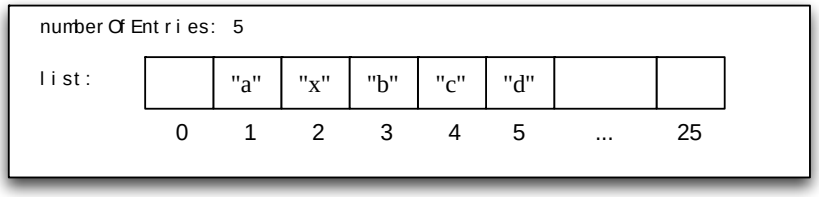


Again,we update the loop index by sutracting 1 and then compare the result with newIndex. Since 1 is not >= 2, we exit the loop and then return to add from the makeRoom method.

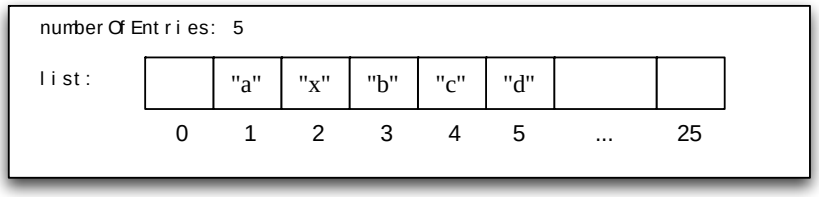
The next line in the add method puts newEntry into the array at newPosition.
newPosition: 2
newEntry: "x"



The next line in the add method adjusts the number of entries.
newPosition: 2
newEntry: "x"



Finally we invoke the ensureCapacity() method.
The list.length is 26, so we set capacity to 25.
The numberOfEnties is 5 and capacity is 26, so the condition numberOfEntries >= capacity is false. We return from the ensureCapacity() method.
We return from the add method and the final state of our object is



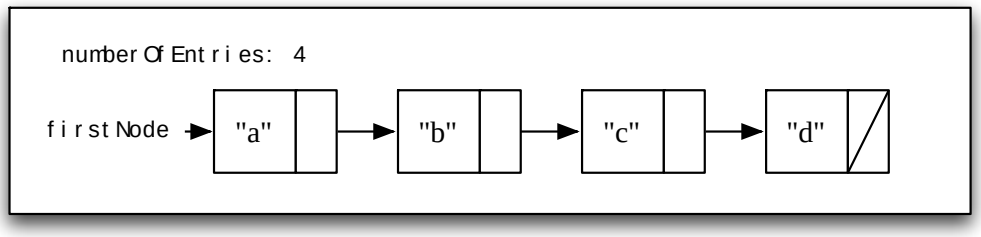
Lets contrast this with the add method for the linked implementation of the list.

```
public void add(int newPosition, T newEntry) {
   if ((newPosition >= 1) && (newPosition <= numberOfEntries + 1)) {
      Node newNode = new Node(newEntry);
      if (newPosition == 1) // Case 1
      {
         newNode.setNextNode(firstNode);
         firstNode = newNode;
      } else // Case 2: List is not empty
      { // and newPosition > 1
         Node nodeBefore = getNodeAt(newPosition - 1);
         Node nodeAfter = nodeBefore.getNextNode();
         newNode.setNextNode(nodeAfter);
         nodeBefore.setNextNode(newNode);
      } // end if
      numberOfEntries++;
   } else
       throw new IndexOutOfBoundsException(
             "Illegal position given to add operation.");
} // end add
```

Again we trace the last statement in the code fragment.

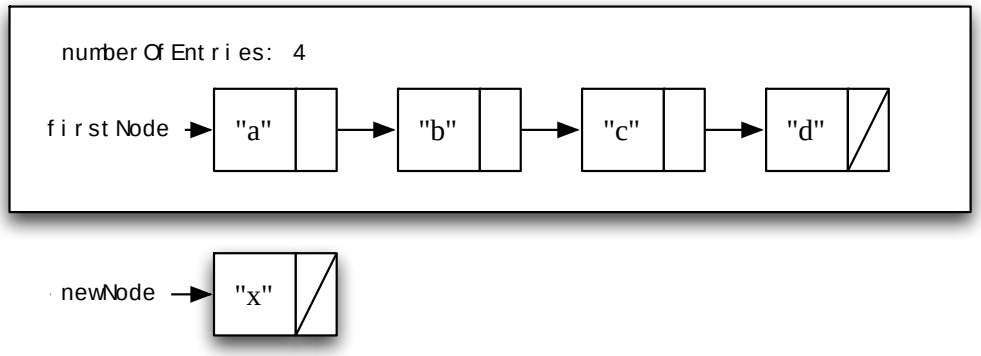
```
LList<String> x = new LList()<String>;
x.add("a");
x.add("b");
x.add("c");
x.add("d");
x.add(2, "x");        // trace this one
```

The initial state of the object is



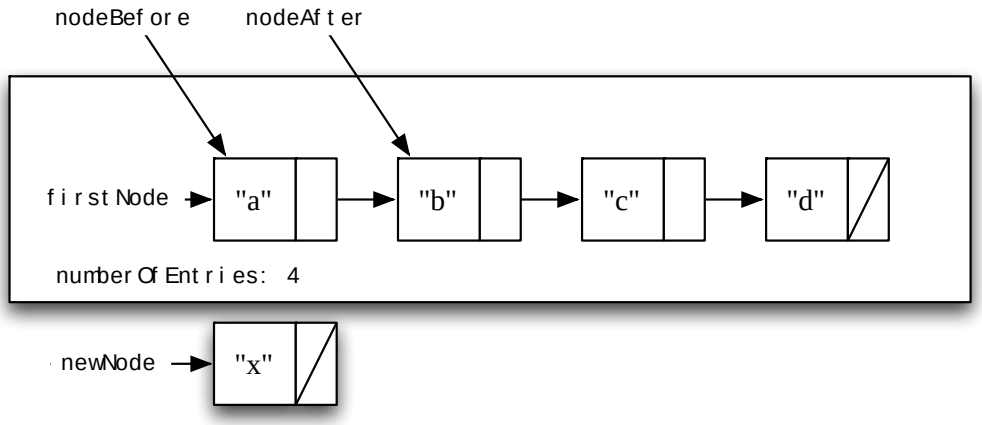
The condition ((newPosition >= 1) && (newPosition <= numberOfEntries + 1)) is true.
A new node is created.

```
newPosition: 2
newEntry: "x"
```



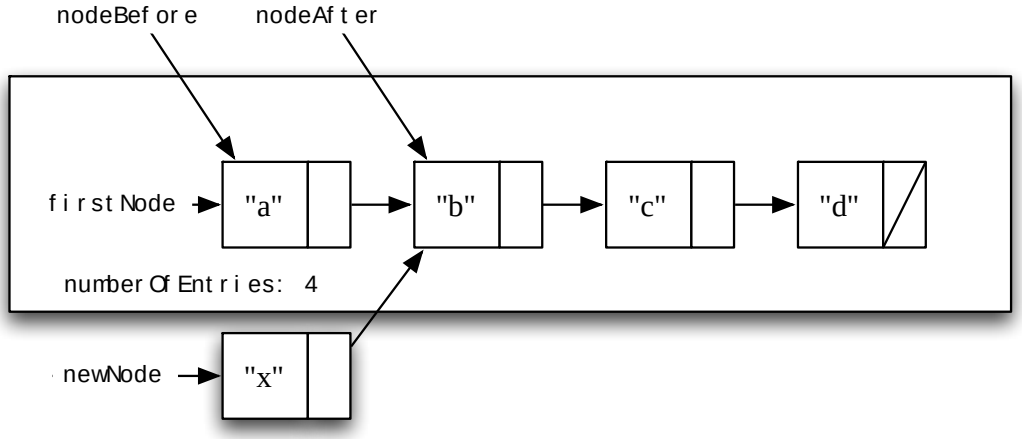
The condition (newPosition == 1) is false since the insertion is not at the front of the list. The else branch is chosen. The local variable nodeBefore is set to getNodeAt(1). Then the variable nodeAfter is set.

```
newPosition: 2
newEntry: "x"
isSuccessful: true
```



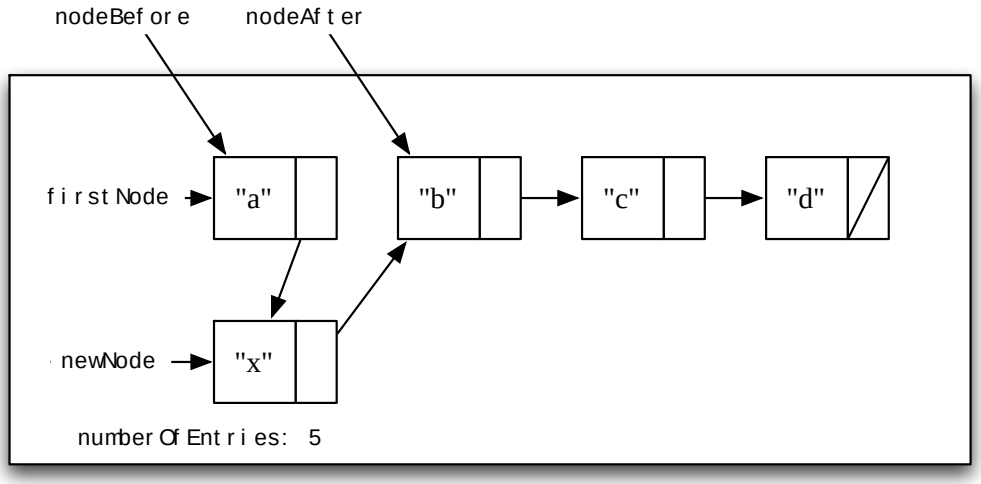
The next reference for the new node is set to be the node after the insertion point.

newPosition: 2
newEntry: "x"
isSuccessful: true



Finally, the new node is linked into the list by setting the next reference of the node before and the number of entries is updated.

newPosition: 2
newEntry: "x"
isSuccessful: true



The reverse operation is often supplied as a basic operation of a list. One example is used to convert an array that was sorted from smallest to largest into an array that was sorted in the other direction. The cycle operation is less common, but can be used in a number of different applications. One example is a list of days in the week. At the start of the week, the list is (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday). As each day passes, the list will be cycled by moving the first item to the end of the list. At Sunday midnight, for example, the list will be cycled to (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday).

To implement both the reverse and cycle methods, we will be using list surgery. As we saw with the linked representatin of the bag, the order that operations is done can be criticial. If you are not careful, the list will not survive the surgery. To avoid problems with surgery, it is always a good idea to trace carefully the intended operation of methods before implementing the code.

Pre-Lab Visualization

Primes

Suppose you are interested in finding the primes between 2 and 15 inclusive. The list of candidates is:

Candidates: 2 3 4 5 6 7 8 9 10 11 12 13 14 15

The algorithm proceeds in rounds. In each round a single prime is discovered and numbers that have that prime as a factor are eliminated.

Round 1:

Cross the first value off of the Candidates list and add it to the Primes list. Cross out any candidate that is divisible by the prime you have just discovered and add it to the composites list.

 *Primes*:

 *Composites*:

Round 2:

To make the operation of the algorithm clearer, copy the contents of the lists as they appear at the end of the previous round.

Cross the first value off of the Candidates list and add it to the Primes list. Cross out any candidate that is divisible by the prime you have just discovered and add it to the composites list.

 *Primes*:

 *Candidates*:

 *Primes*:

 *Composites*:

Round 3:

Again, copy the contents of the lists as they appear at the end of the previous round.

Cross the first value off of the Candidates list and add it to the Primes list. Cross out any candidate that is divisible by the prime you have just discovered and add it to the composites list.

 *Primes*:

 *Candidates*:

 *Primes*:

 *Composites*:

You can complete the other rounds if you wish, but most of the interesting work has been completed.

Finding Composites

The heart of this algorithm is removing the composite values from the candidates list. Let's examine this process more closely. In the first round, after removing the 2, the list of candidates is

Candidates: 3 4 5 6 7 8 9 10 11 12 13 14 15

How many values are in the list?

The first value to be examined is the 3. What is its index?

The second value to be examined is the 4. What is its index?

Since 4 is divisible by 2, we need to remove it from the list of candidates. What is the new list of candidates after the 4 is removed?

 *Candidates*:

The third value to be examined is the 5. What is its index?

The fourth value to be examined is the 6. What is its index?

Since 6 is divisible by 2, we need to remove it from the list of candidates. What is the new list of candidates after the 6 is removed?

 *Candidates*:

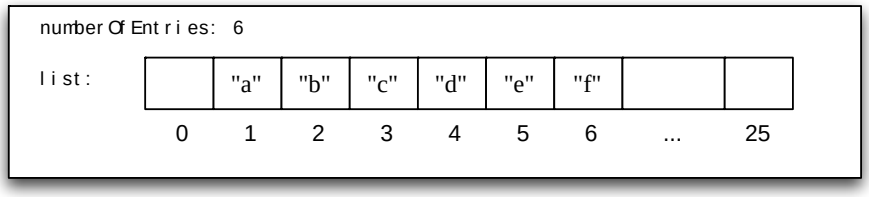
The fifth value to be examined is the 7. What is its index?

Can you simply loop over the indices from 1 to 13 to examine all the candidates?

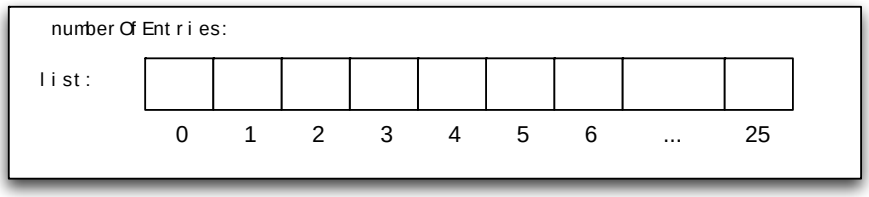
Develop an algorithm to examine all the values in the candidates list and remove them if they are divisible by the given prime.

Array Reverse

Suppose there is an array based list with the following state:

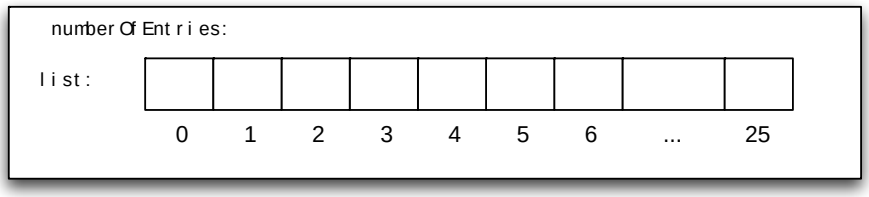


What will the final state be after reverse()?

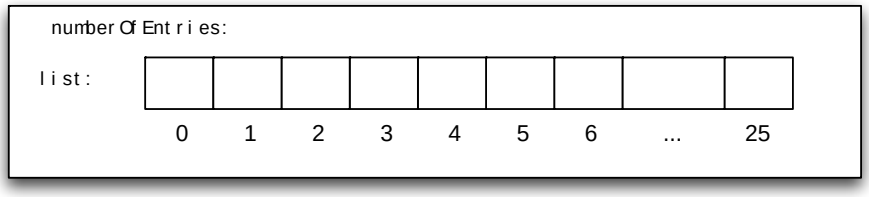


To reach the final state, follow these steps. Show the state of the list after each step.

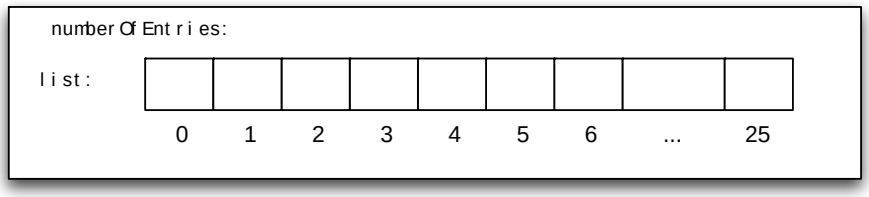
a. Swap the items in the first and last positions.



b. Swap the items in the second and second to last positions.



c. Swap the items in the third and third to last positions.



To be general, a loop will be needed.

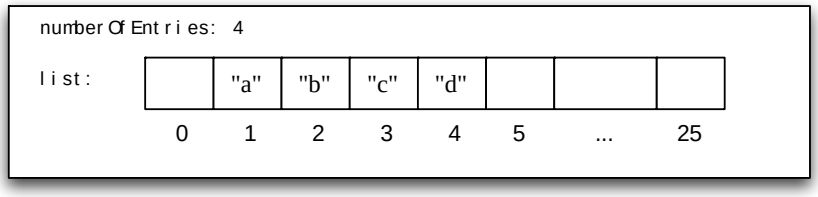
If there are *n* items in the list, what will be the indices of the first items in the swaps?

What will be the indices of the second items in the swaps?

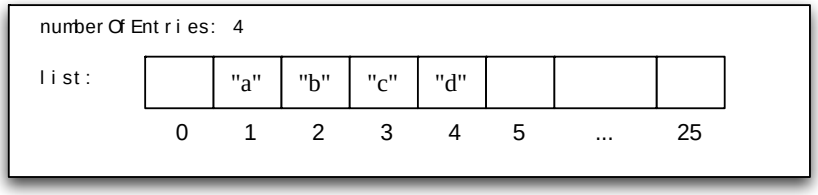
Write an algorithm to implement reverse.

Array Cycle

Suppose there is a list with the following state:

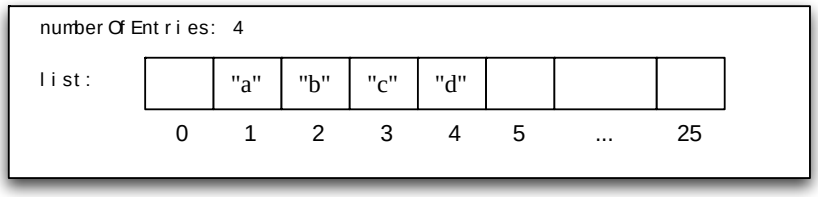


What will the final state be after cycle()?

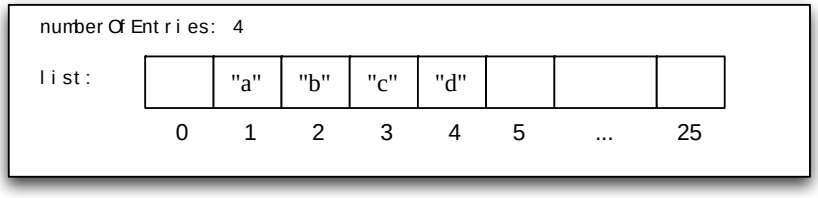


To reach the final state, follow these steps. Show the state of the list after each step.

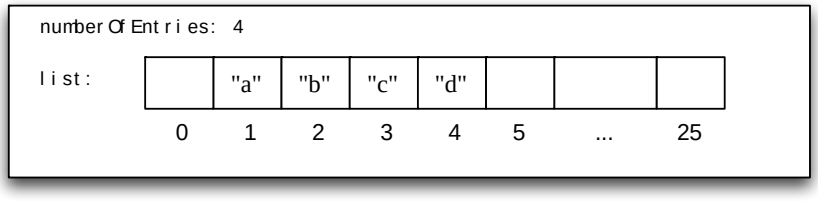
a. Remember the first item.



b. Copy each item down one position.



c. Copy the remembered item to the last position.



Again, a loop will be needed.

If there are *n* items in the list, what are the indices of the items that are moved?

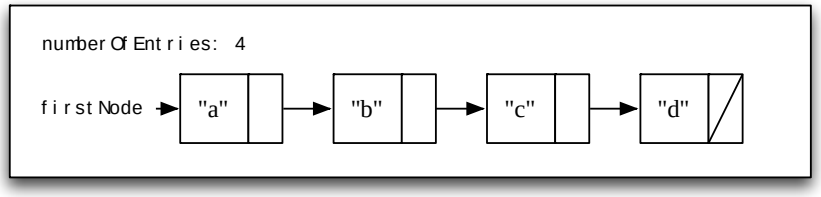
What are the indices of the locations that the items are moved into?

Write an algorithm to implement cycle.

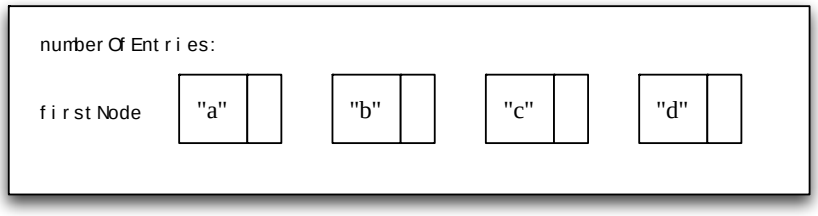
Linked Reverse

The algorithm that will be used to reverse the linked chain is very different from the algorithm used with the array implementation. No swaps will be done, but instead the links will be altered in a single pass over the list.

Suppose there is a list with the following state:

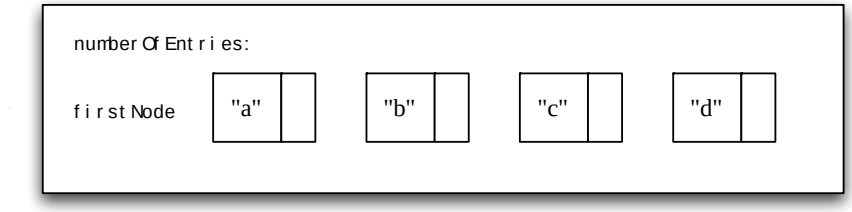


What will the final state be after reverse()?

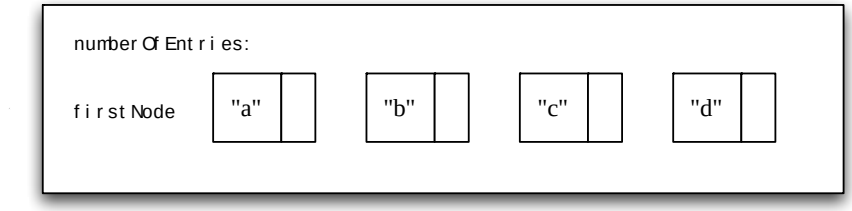


To reach the final state, follow these steps. Show the state of the list after each step.

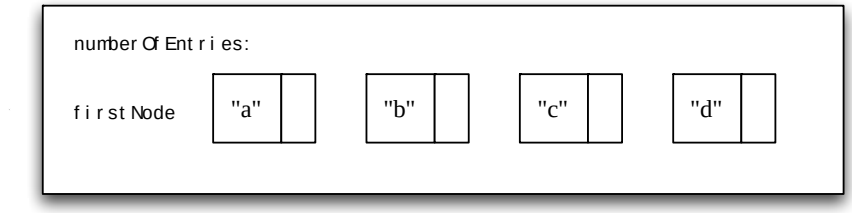
a. Start in the initial state. Use three variables to reference the first three nodes. (From now on, when refering to a position, it will be with respect to the order in the diagram).



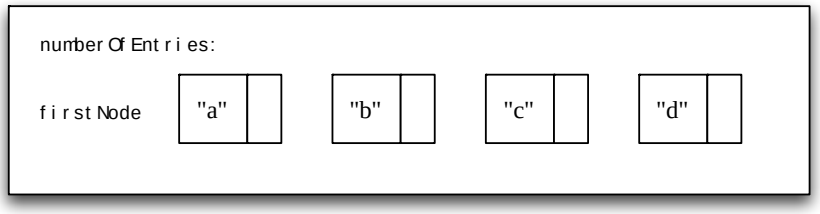
b. Make the first node's next reference be null.



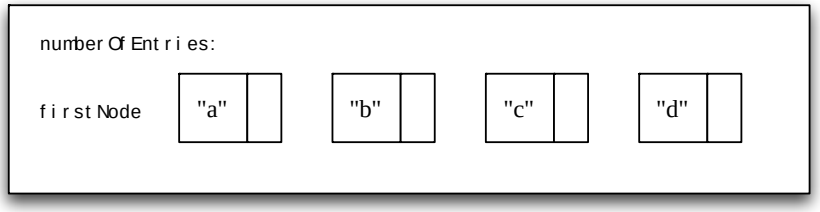
c. Make the second node's next reference be the first node. Change all three reference variables so that they move forward by one in the picture.



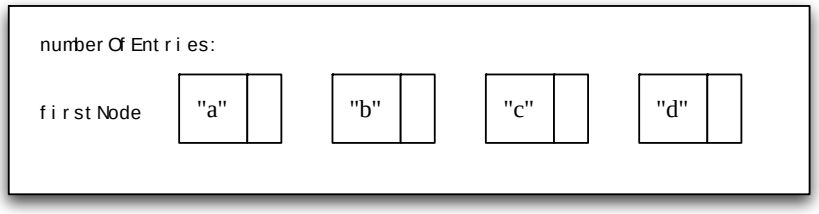
d. Make the third node's next reference be the second node. Change all three reference variables so that they move forward by one in the picture.



Make the fourth node's next reference be the third node.



f. Change the variable firstNode so that it references the fourth node.



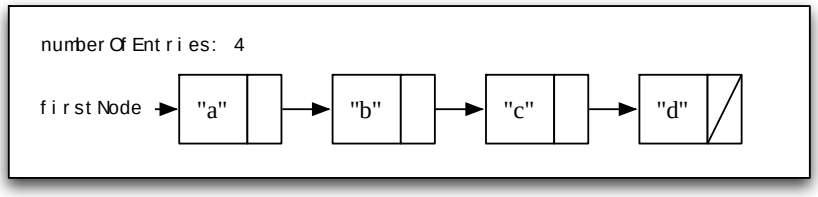
To be general, a loop will be needed. Write an algorithm for the reverse operation.

Consider each of the following cases and decide if the algorithm handles it correctly.

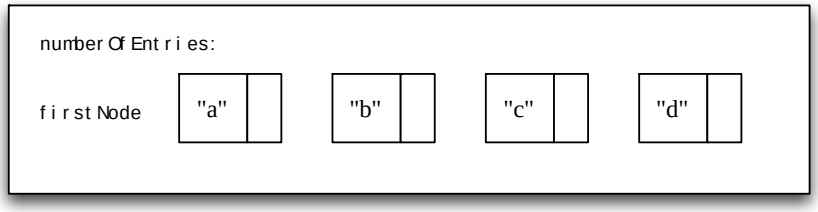
i. A list with no elements
ii. A list with one element
iii. A list with two elements

Linked Cycle

Suppose there is a list with the following initial state:

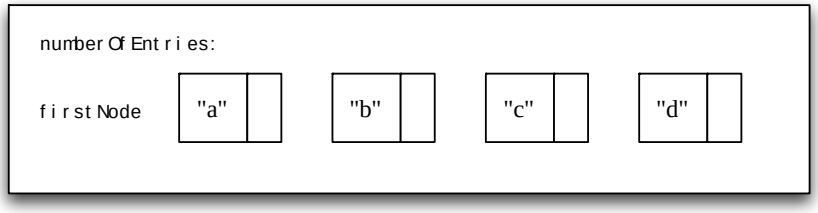


What will the final state be after cycle()? (The first node is moved to the back of the list.)

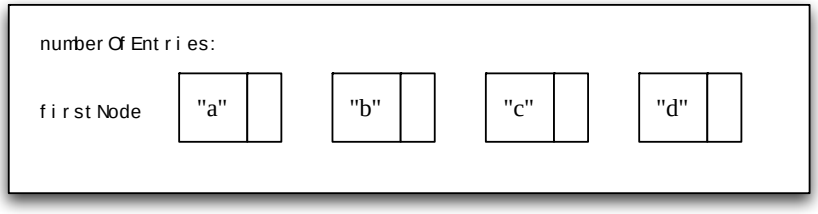


To reach the final state, we can follow these steps. Show the state of the list after each step.

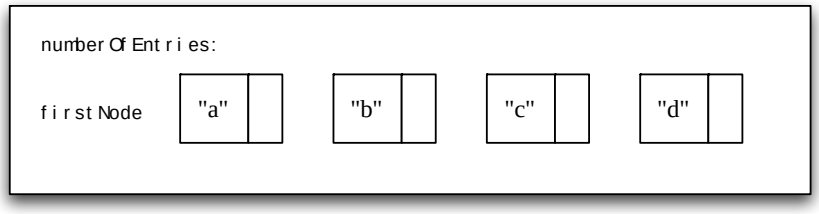
a. Start in the initial state. Create a variable that refers to the second node in the list. Find and then create a variable that refers to the last node in the list.



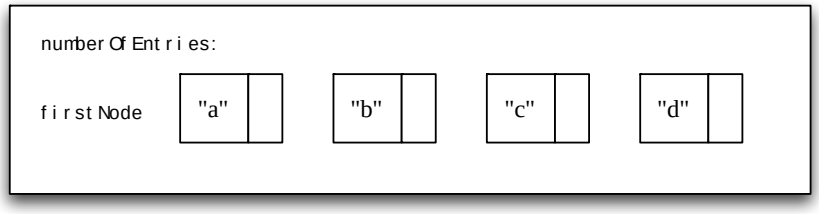
b. Make the last node's next reference be the first node.



c. Change the variable firstNode so that it references the second node.



d. Make the former first node's next reference be null.



Write an algorithm for the cycle operation.

Consider each of the following cases and decide if the algorithm handles it correctly.

i. A list with no elements
ii. A list with one element
iii. A list with two elements

Directed Lab Work

Primes

The skeleton of the Primes class already exists and is in *Primes.java*.

Step 1. Look at the skeleton in *Primes.java*. Compile Primes. Run the main method in Primes.

Checkpoint: If all has gone well, the program will run and accept input. It will then end. The goal now is to create the list of candidates.

Step 2. In main declare and create the candidates list. Add in the values.

Step 3. Print out the candidates list.

Checkpoint: Compile and run the program. Enter 7 for the maximum value. You should see the list { <2> <3> <4> <5> <6> <7> }. The next goal is to do a single round finding a prime in the candidates list.

Step 4. In main declare and create the primes and composites lists.

Step 5. Remove the first value from the primes list and remember it in a variable.

Step 6. Print out the prime that was discovered.

Step 7. Add it to the primes list.

Step 8. Print out all three lists.

Checkpoint: Compile and run the program. Enter 7 for the maximum value. The value 2 should be removed from the candidates list and added to the primes. Now all values that are divisible by the prime should be removed from the candidates list and added to the composites list. Next, this procedure will be encapsulated in the method getComposites().

Step 9. Refer to the pre-lab exercises and complete the getComposites() method. To determine if one integer value is divisible by another, you can use the modulus operator (% in Java).

Step 10. Between the code from steps 7 and 8, call getComposites().

Checkpoint: Compile and run the program. Enter 15 for the maximum value. Compare the results with the pre-lab exercises. Reconcile any differences.

Just as in the counting game, a loop will be used to do the rounds.

Step 11. Wrap the code from steps 5 through 8 in a while loop that continues as long as the candidates list is not empty.

Final checkpoint: Compile and run the program. Enter 15 for the maximum value. Compare the results with the pre-lab exercises. Reconcile any differences.

Run the program with 100 as the maximum value. Carefully verify your results.

The classes AList and LList are working implementations of the *ListInterface.java*. The methods you will be working on already exist but do not function yet. Take a look at that code now if you have not done so already.

Array Reverse

Step 1. In the reverse method of AList, implement your algorithm from the pre-lab exercises. Iteration is needed.

Checkpoint: Compile and run ArrayListExtensionsTest. The checkReverse tests should pass. If not, debug and retest.

Array Cycle

Step 2. In the cycle method of AList, implement your algorithm from the pre-lab exercises. This method needs some form of iteration, but it may not be explicit. It can use the private methods of the AList class to avoid an explicit loop in the cycle method. Either way is acceptable.

Checkpoint: : Compile and run ArrayListExtensionsTest. All tests should pass. If not, debug and retest.

Linked Reverse

Step 3. In the reverse method of LList, implement your algorithm from the pre-lab exercises. Iteration is needed.

Checkpoint: Compile and run LinkedListExtensionsTest. The checkReverse tests should pass. If not, debug and retest.

Linked Cycle

Step 4. In the cycle method of LList, implement your algorithm from the pre-lab exercises.

Final checkpoint: Compile and run LinkedListExtensionsTest. All tests should pass. If not, debug and retest.

Post-Lab Follow-Ups

1. After a certain point in our iteration, all the remaining values in the candidates list are prime. Modify the program to just copy values directly when that point has been reached.

2. Write a program that will get a list of words and then remove any duplicates.

3. Write a program that will get two lists of words and will create a third list that consists of any words in common between the two input lists.

4. Write a program that will read in a list of words and will create and display two lists. The first list will be the words in odd positions of the original list. The second list will be all the remaining words.

5. Write a program that will get a list of integer values each of which is greater than one and determines if all the posible pairs of values from the list are relatively prime. Two values are relatively prime if they share no prime factors. For example, the values 10 and 77 are relatively prime, but 10 and 55 are not since the share 5 as a factor. If we look at the list {6, 25, 77} each of the pairs (6, 25), (6, 77), and (25,77) are relatively prime and it passes our test. On the other hand the list {6, 25, 14} will fail the test. While the pairs (6, 25) and (25, 14) are relatively prime, the pair (6,14) is not.

6. Create test cases for the other methods in ListInterface.

7. In lists of size 10, 20 and 30, how many assignments are made using the list array by the reverse method in AList? How many times is the getNextNode method called by the reverse method in LList? (Include calls made directly by reverse or by helper methods like getNodeAt.)

8. In lists of size 10, 20 and 30, how many assignments are made using the list array by the cycle method? How many times is the getNextNode method called by the cycle method in LList? (Include calls made directly by reverse or by helper methods like getNodeAt.)

9. Implement the reverse method using only the public methods in ListInterface. Compare the performance with what you found in Questions 2 and 3.

10. Implement cycle method using only the public methods in ListInterface. Compare the performance with what you found in Questions 2 and 3.

11. Consider a method
    ```
    void randomPermutation()
    ```
    that will randomly reorder the contents of the list. Create three versions of the method and compare the performance as you did in Question 4. In the first version, only use the methods from ListInterface. In the second version, always work directly with the array list in the AList implementation. In the third version, always work directly with the linked chain in the LList implementation.

12. Consider a method
    ```
    void moveToBack(int from)
    ```
    that will move the item at position from to the end of the list. Create three versions of the method and compare the performance as you did in Question 4. In the first version, only use the methods from ListInterface. In the second version, always work directly with the array list in the AList implementation. In the third version, always work directly with the linked chain in the LList implementation.

13. Consider a method

```
void interleave()
```

that will do a perfect shuffle. Conceptually, you split the lists into halves and then alternate taking items from the two lists. For example, if the original list is [a b c d e f g h i], the splits would be [a b c d e] and [f g h i]. (If the length is odd, the first list gets the extra item.) The result of the interleave is [a f b g c g d i e]. Create three versions of the method and compare the performance as you did in Question 4. In the first version, only use the methods from ListInterface. In the second version, always work directly with the array list in the AList implementation. In the third version, always work directly with the linked chain in the LList implementation.