LLab 13 List Implementation

Goal

In this lab you will explore two implementations of the ADT list. The first implementation will use an array and the second will use a linked structure. In both cases you will create new methods that work directly with the implementations. 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

In `javadoc` directory

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

Java Files

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

Introduction

As was seen in the last lab, a list is an ordered collection of elements supporting basic operations such as add and remove. 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.

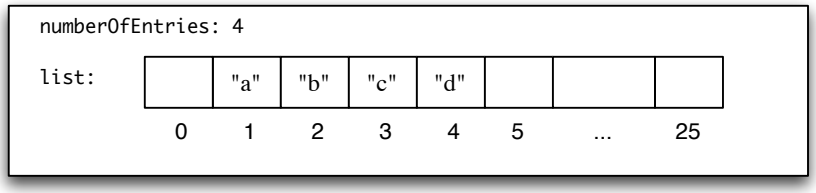
```
public void add(int newPosition, T newEntry) {
    checkInitialization();

    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



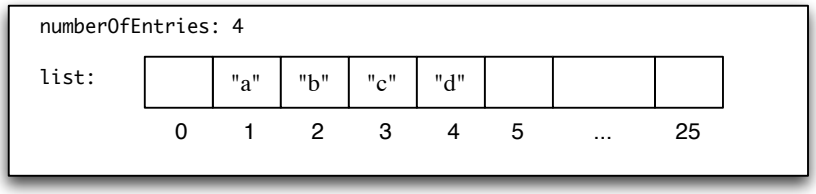
The object has been initialized, so `checkInitialization()` returns.

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)`.

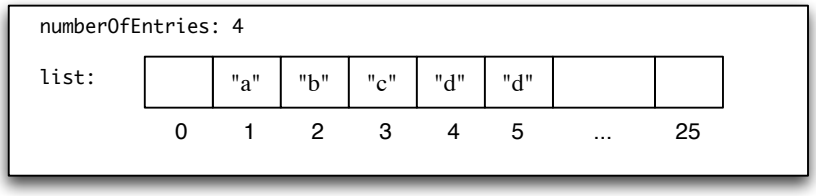
We set the local variables `newIndex` to 2 and `lastIndex` to 4 resulting in the following state.

```
newPosition: 2
newIndex: 2
lastIndex: 4
```



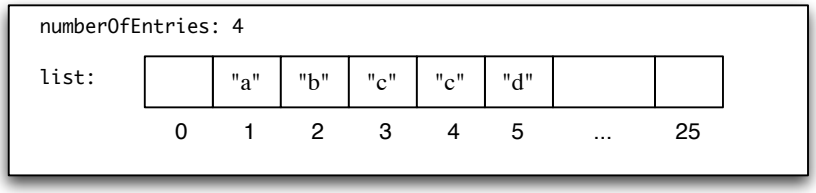
Now we do the for loop in `makeRoom`, we start `index` at lastIndex and then execute `list[index + 1] = list[index]`. We are in the following state.

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



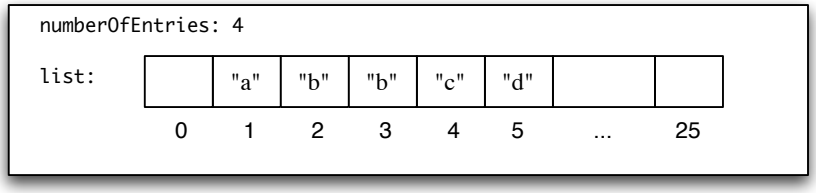
We update the loop index by sutracting 1 and then compare the result with `newIndex`. Since 3 is >= 2, we continue with a second execution of the body of the loop. After it is done, we are in the following state.

```
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.

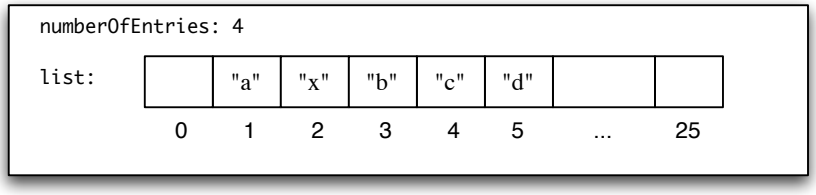
```
newPosition: 2
newIndex: 2
lastIndex: 4
index: 2
```



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.

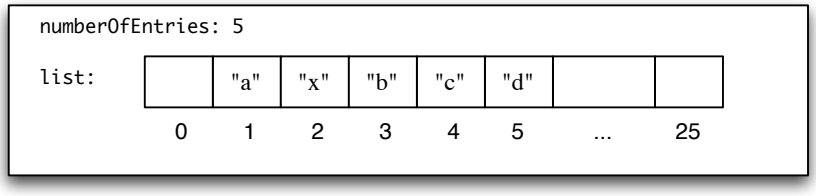
The next line in the add method puts `newEntry` into the array at `newPosition`.

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

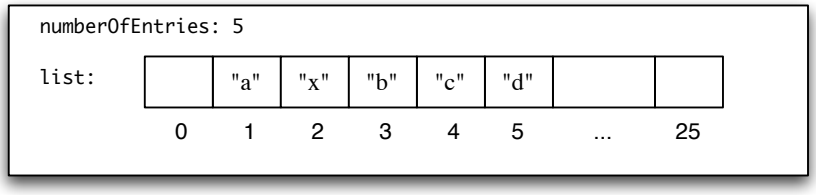


The next line in the add method adjusts the number of entries.

`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



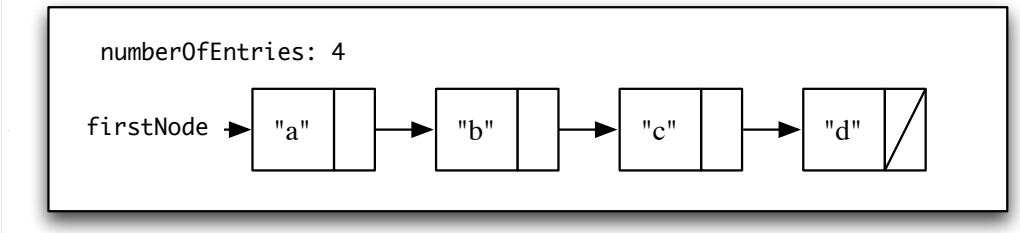
Lets contrast this with the add method for the linked implementation of the list.

```
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
```

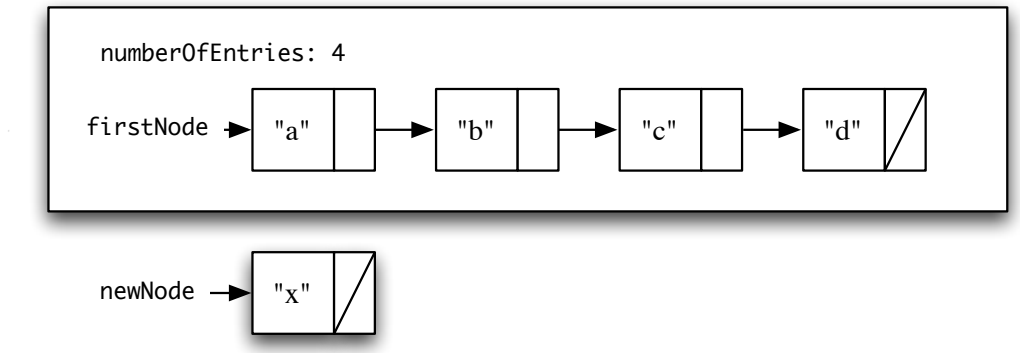
The initial state of the object is



[4]

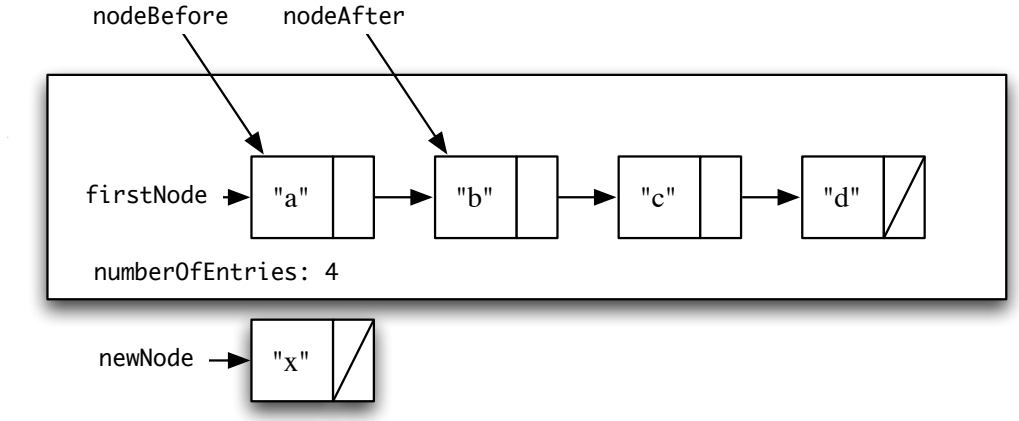
The condition `((newPosition >= 1) && (newPosition <= numberOfEntries + 1))` is true. A new node is created.

```
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.

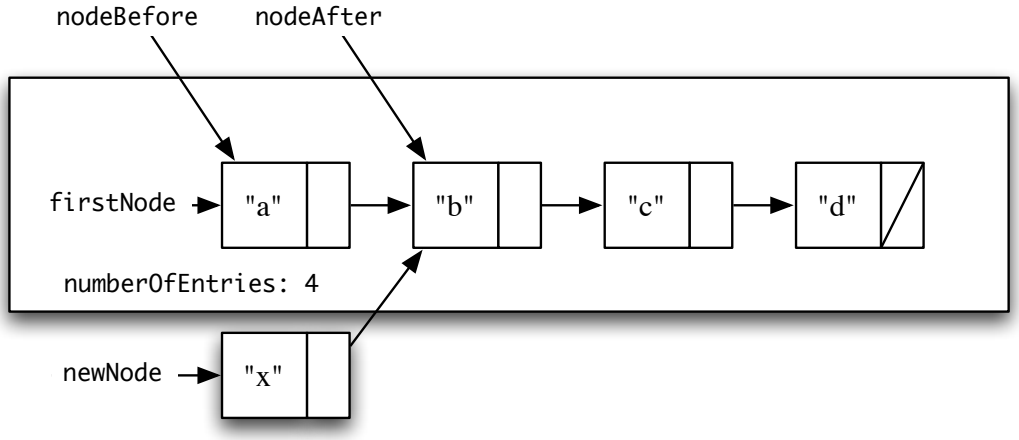
```
newPosition: 2
newEntry: "x"
isSuccessful: true
```



[6]

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

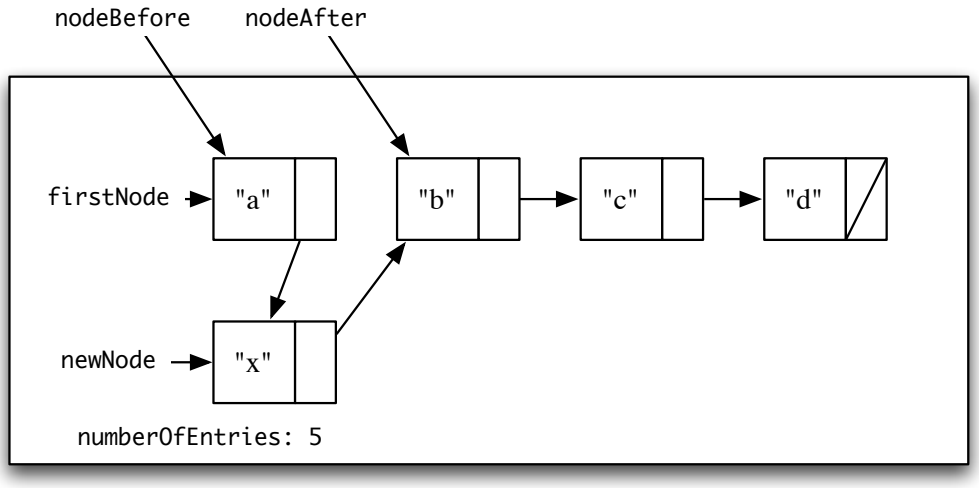
```
newPosition: 2
newEntry: "x"
isSuccessful: true
```



[7]

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
```



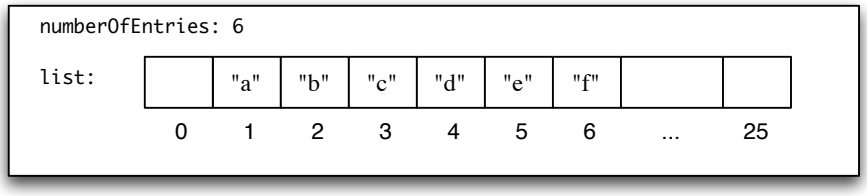
The reverse operation is often supplied as a basic operation of a list. One example that we saw in the Advanced sorting lab was used to convert an array that was sorted from smallest to largest into an array that was sorted in the other direction. The cycle operation is less common, but can be used in 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.

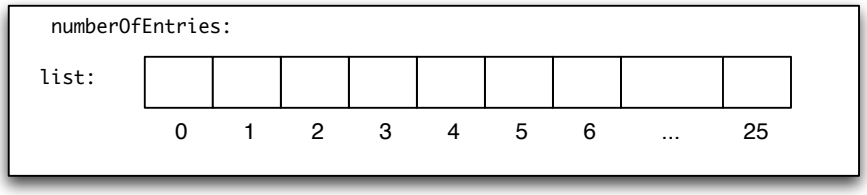
Pre-Lab Visualization

Array Reverse

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

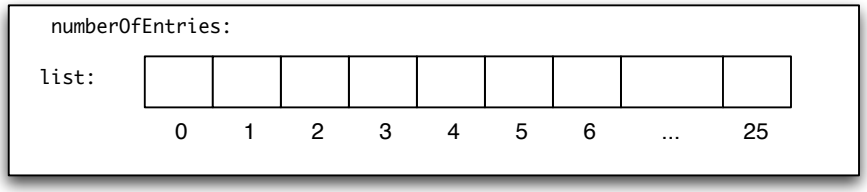


What will the final state be after `reverse()`?

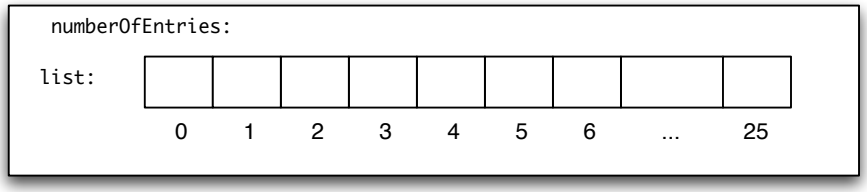


To reach the final state, follow these steps. Show the state of the list after each step.

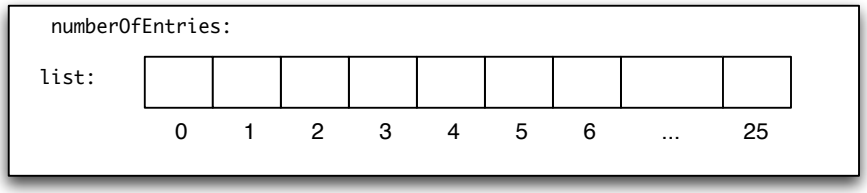
a. Swap the items in the first and last positions.



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.

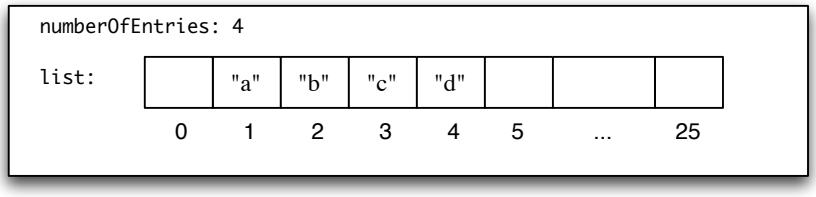
If there are *n* items in the list, what will be the indices of the first items in the swaps?

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

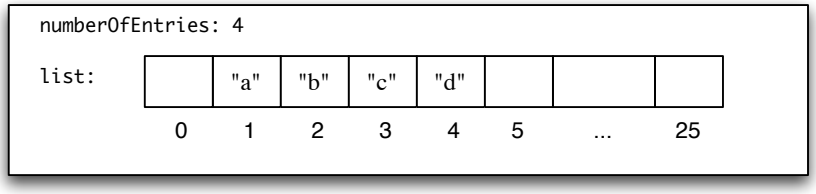
Write an algorithm to implement `reverse`.

Array Cycle

Suppose there is a list with the following state:

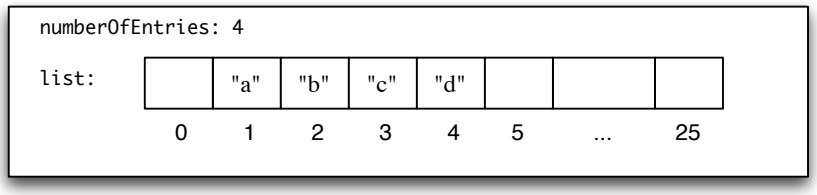


What will the final state be after `cycle()`?

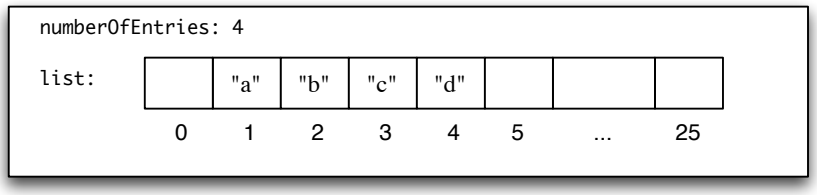


To reach the final state, follow these steps. Show the state of the list after each step.

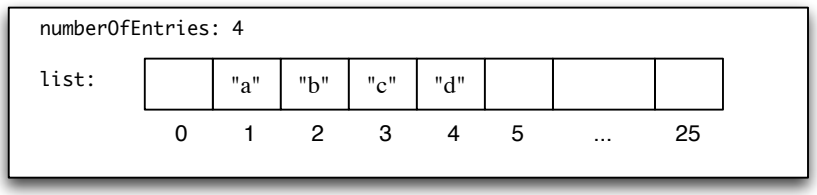
a. Remember the first item.



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?

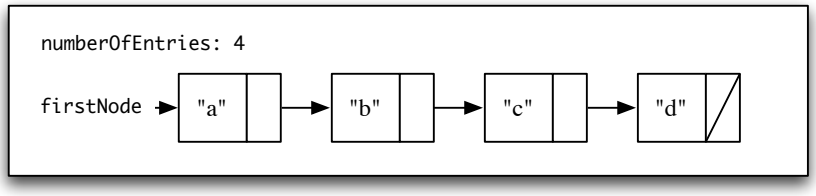
What are the indices of the locations that the items are moved into?

Write an algorithm to implement `cycle`.

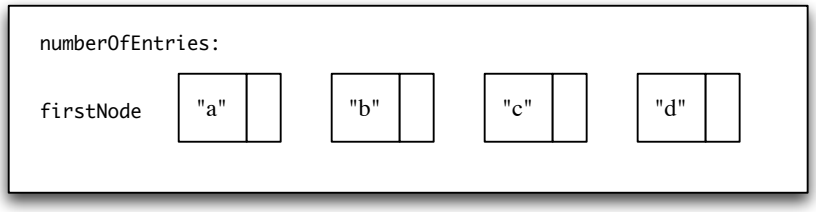
Linked Reverse

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:

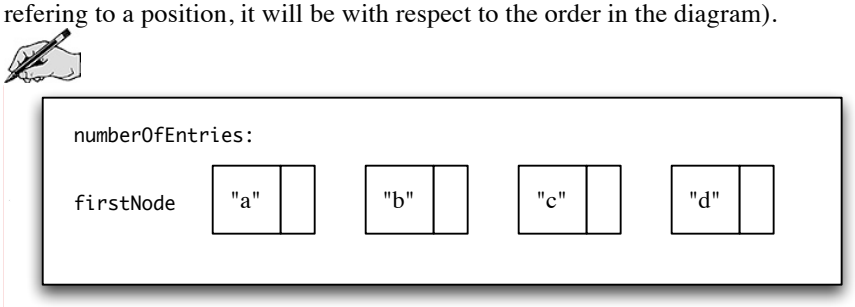


What will the final state be after `reverse()`?



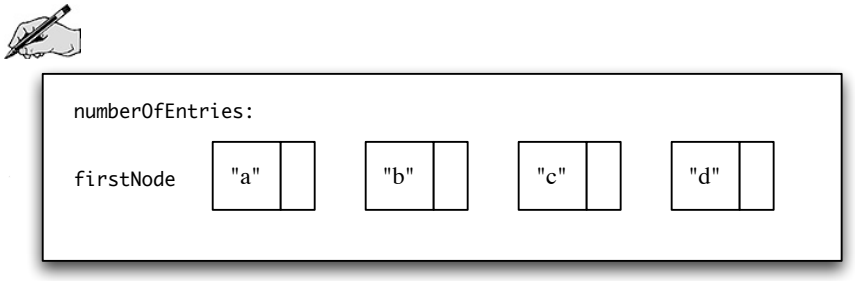
To reach the final state, follow these steps. Show the state of the list after each step.

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).

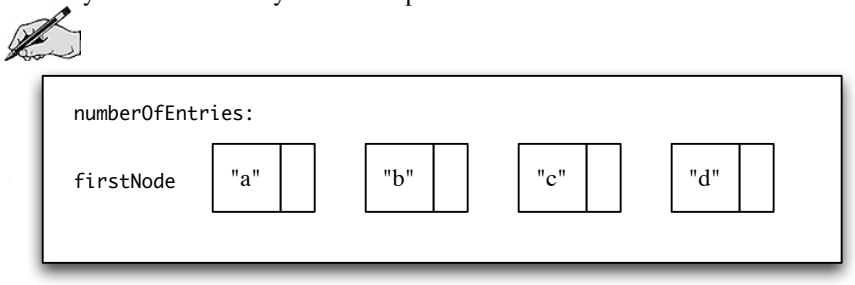


[20]

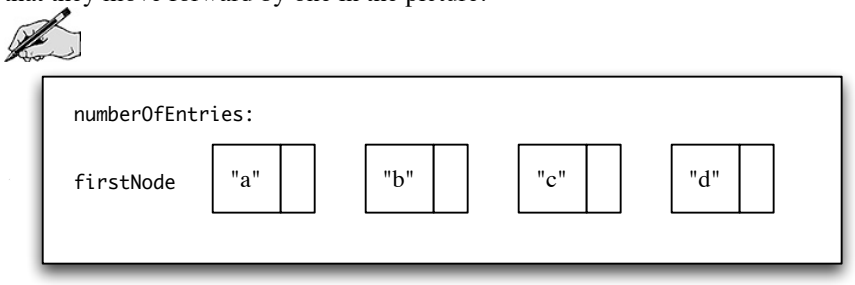
b. Make the first node's next reference be null.



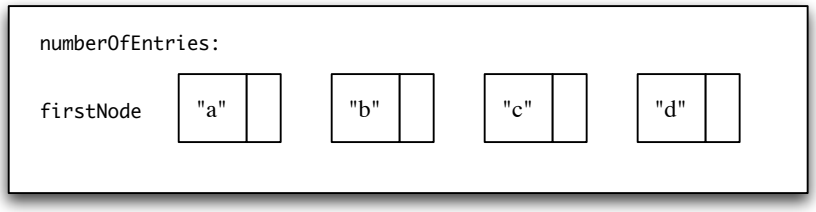
c. Make the second node's next reference be the first node. Change all three reference variables so that they move forward by one in the picture.



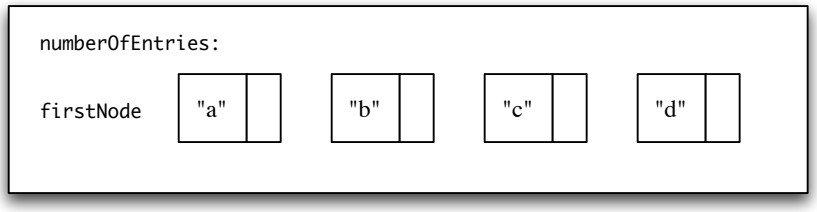
d. Make the third node's next reference be the second node. Change all three reference variables so that they move forward by one in the picture.



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



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



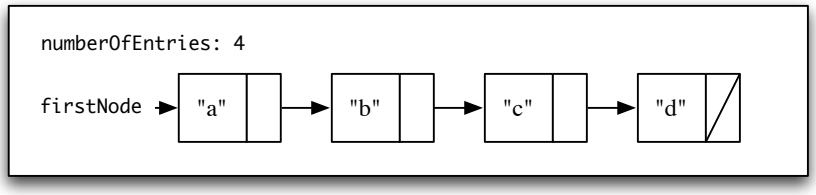
To be general, a loop will be needed. Write an algorithm for the reverse operation.

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

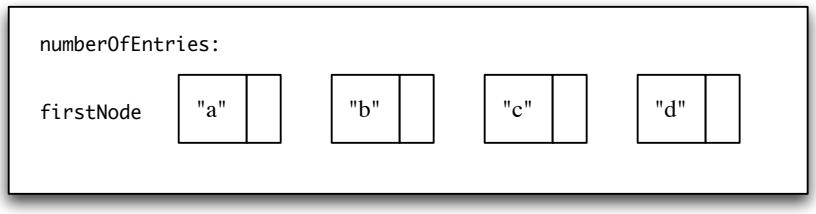
i. A list with no elements
ii. A list with one element
iii. A list with two elements

Linked Cycle

Suppose there is a list with the following initial state:

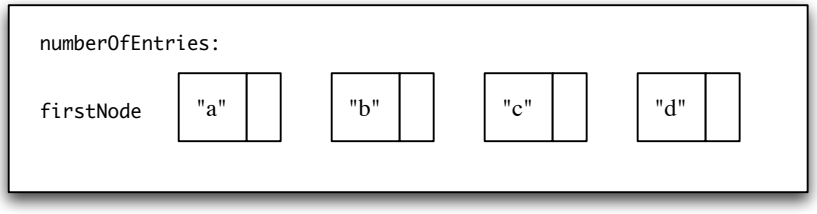


What will the final state be after `cycle()`? (The first node is moved to the back of the list.)

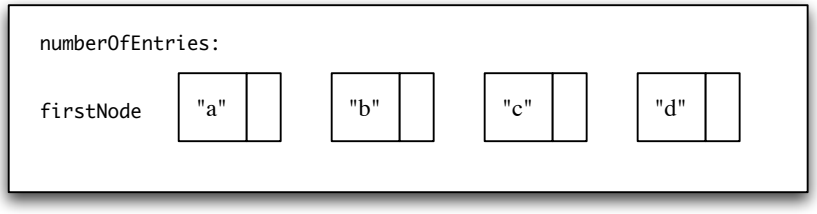


To reach the final state, we can follow these steps. Show the state of the list after each step.

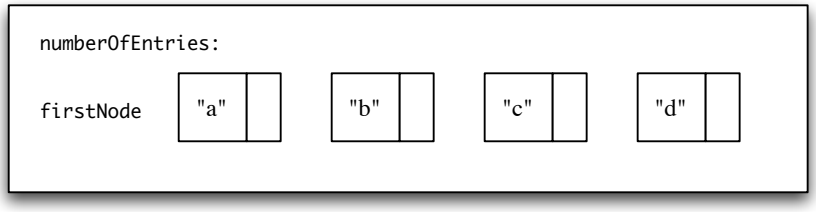
a. Start in the initial state. Create a variable that refers to the second node in the list. Find and then create a variable that refers to the last node in the list.



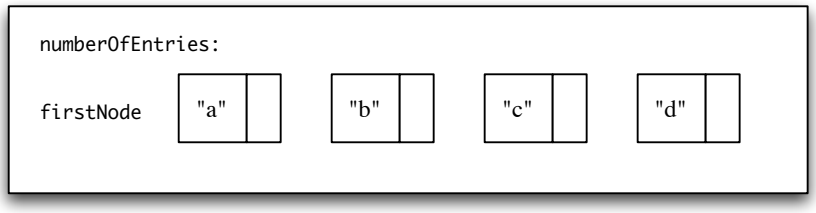
b. Make the last node's next reference be the first node.



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

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

2. 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`.)

3. 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`.)

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

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

6. 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.

7. 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.

8. 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.