Lab 4 Link Based Bag Implementation

Goal

In this lab you will explore the implementation of the ADT bag using a linked chain. To allow you to see the difference with the array implementation, the methods you will implement will be the same as in the previous lab (`equals`, `remove`, `duplicateAll`, and `removeDuplicates`).

Resources

- Appendix D: Creating Classes from Other Classes
- Chapter 1: Bags
- Chapter 3: A Bag Implementation That Links Data

In `javadoc` directory

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

Java Files

- *`BagExtensionsTest.java`*
- *`BagInterface.java`*
- *`LinkedBag.java`*

Introduction

In the last lab, you saw an array implementation of the ADT bag. In this lab, you will work with a singly linked chain. If you have not done so already, take a moment to examine the code in *`LinkedBag.java`*.

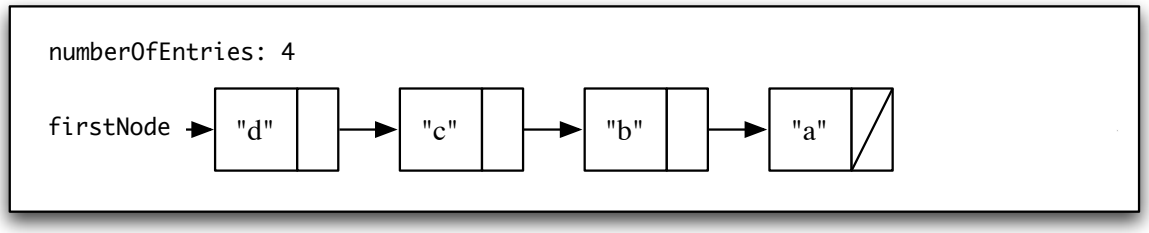
Consider the code that implements the `add` operation.

```
public boolean add(T newEntry) // OutOfMemoryError possible
{
   // Add to beginning of chain:
   Node newNode = new Node(newEntry);
   newNode.next = firstNode; // Make new node reference rest of chain

   // (firstNode is null if chain is empty)
   firstNode = newNode; // New node is at beginning of chain

   numberOfEntries++;
   return true;
   } // end add
```

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

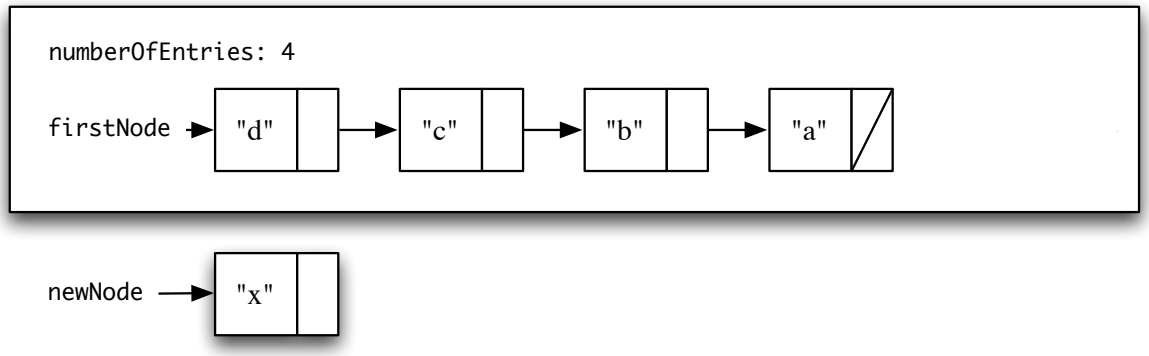
```
LinkedBag<String> x = new LinkedBag<String>();
x.add("a");
x.add("b");
x.add("c");
x.add("d");
x.add("x");      // trace this one
```

The initial state of the object is



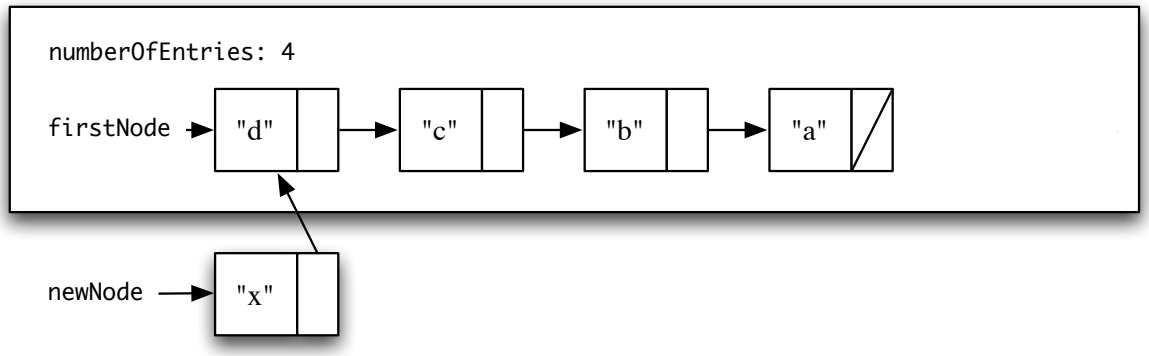
[1]

A new node is created.

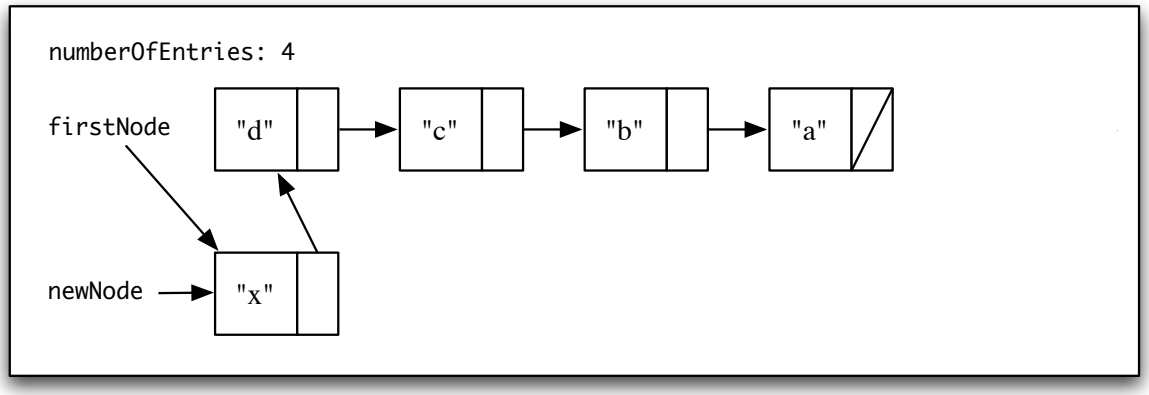


[2]

The next reference for the new node is set to be the first node.

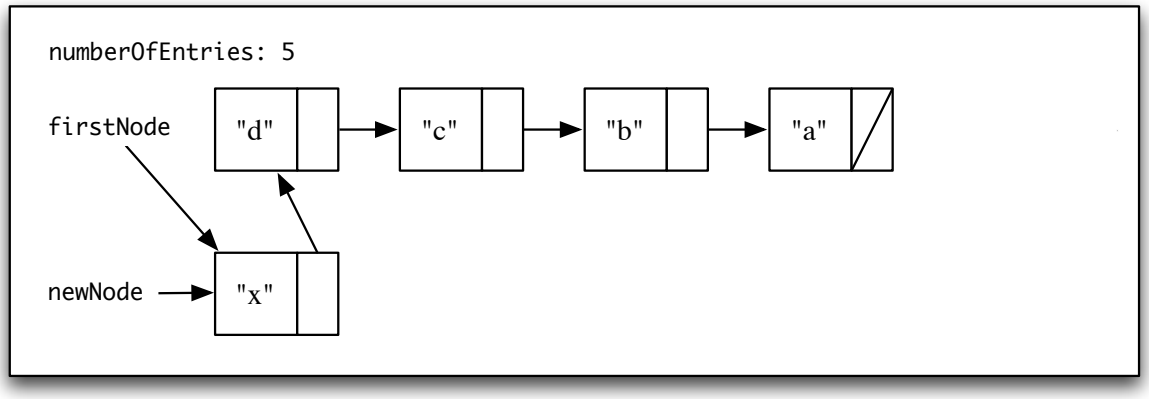


The first node reference is set to the new node.



[3]

Finally, the number of entries is updated.



[4]

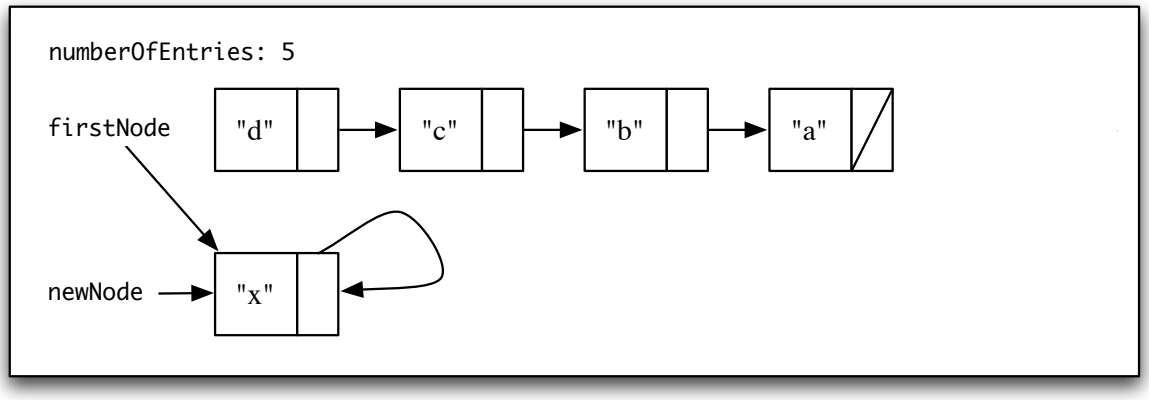
As in the last lab, the first task is to override the `equals` method. The same basic algorithm will be used as last time except that there is no direct access to the entries. Instead we will use a reference to scan across the entries.

The other methods all require that the state of the linked structure will change. Whenever possible, we want to do the minimal amount of work in the surgery or reuse existing code. Unfortunately, if you are not careful, the linked structure will not survive the surgery. Let's see an example of how things could go wrong.

Suppose we switch the order of the two lines that work with the new node in the add method that we just traced.

```
// (firstNode is null if chain is empty)
firstNode = newNode; // New node is at beginning of chain

newNode.next = firstNode; // Make new node reference rest of chain
```

With this change, the final result has problems. The orignal chain of the linked structure is lost and the part of the chain that isn't lost becomes circular. (While there are some interesting uses for a circularly linked structure, this is not the case here. Notice that any method that depends on a null check to find the end of the chain will potentially never finish.)



[5]

To avoid problems with surgery, it is always a good idea to carefully trace the intended operation of methods before implementing the code.

Pre-Lab Visualization

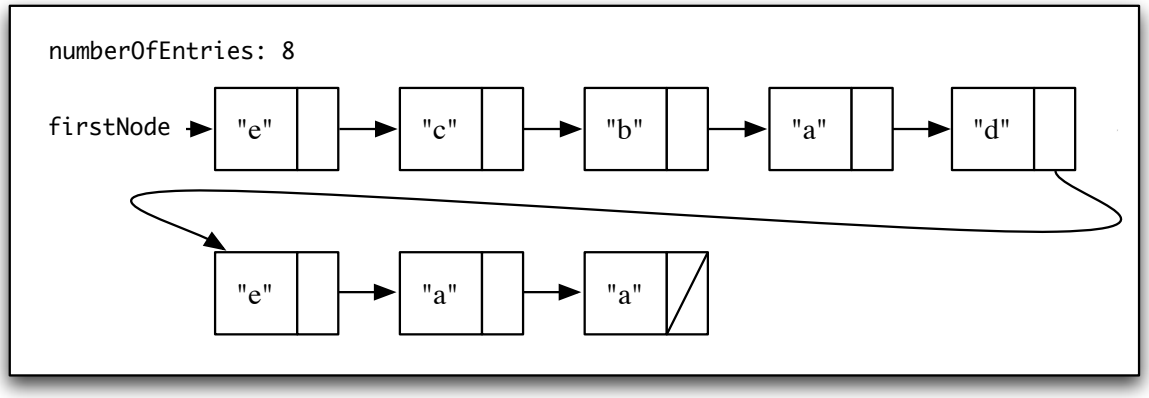
Equals

As with the previous lab, we will determine if two bags are equal by comparing the frequencies of the items in the bags. The main difference is that instead of using a for loop and directly accessing the entries in the bag by their index, we need to use a loop that will scan over one of the linked structures with a reference.

Consider the following example of two bags that we are going to compare for equality. The first bag (`this`) is the one that equals will be applied to and the second bag (`abag`) is the argument of the *equals()* method.

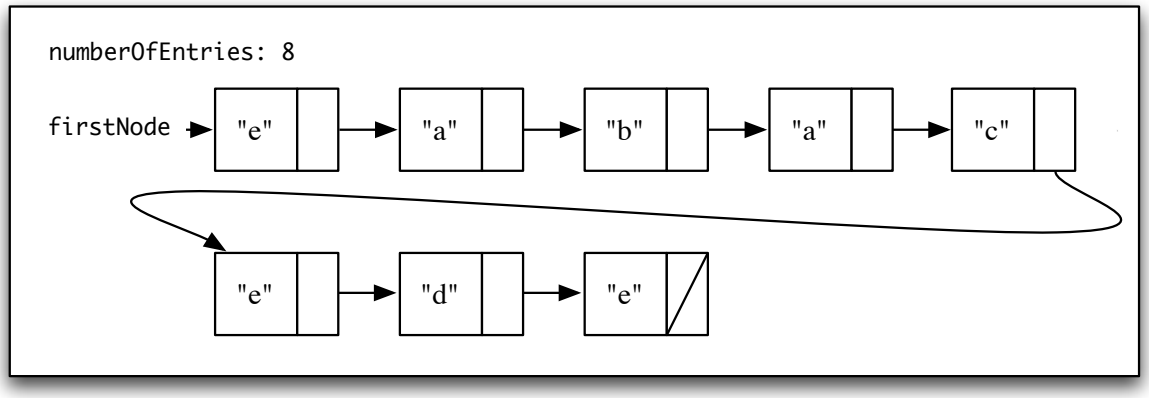
[6]

`this`



[7]

`aBag`



When scanning over the items in `this` bag, where should the reference (call it `scout`) start?

How do we move `scout` to the next node?

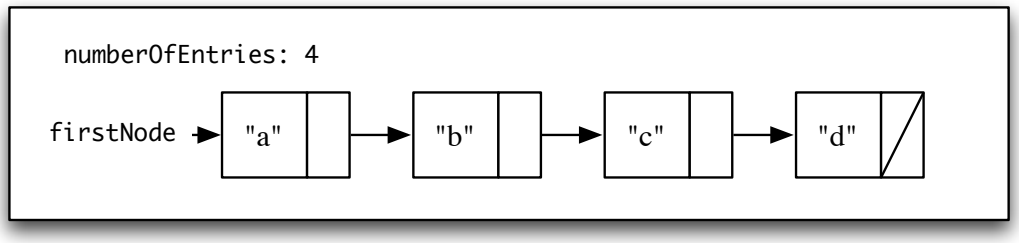
What condition should we check for to know when the `scout` has referenced every node in `this` bag?

Put these all together to create an algorithm that scans over each node in this bag. (We just want to scan the bag, so use a box inside the loop where you will add the code that uses the item referenced by `scout`.)

Finally, look at the algorithm you created for equals from the last lab. Replace the for loop with the loop you just created to implment the `equals()` method for the linked structure.

Remove

Suppose there is a bag with the following state:



[8]

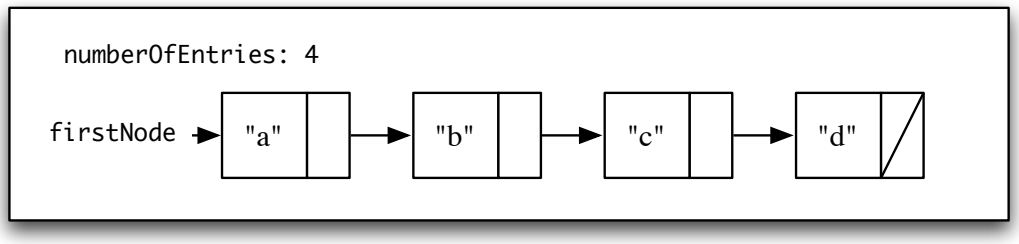
The existing code for the remove method is as follows.

```
public T remove() {
    T result = null;
    if (firstNode != null) {
        result = firstNode.data;
        firstNode = firstNode.next; // Remove first node from chain
        numberOfEntries--;
    } // end if
    return result;
} // end remove
```

What is the state of the bag after executing the remove method?

[9]

We want to modify the remove method so that it will remove a random item from the bag. We will use a similar strategy to what we did in the previous lab. We will generate a random position in the bag and record the item stored there to be returned. We then copy the item at the front of the bag into the position of the item that is to be returned. Finally, we can use the code from the original remove to get rid of the extra node. Lets examine the steps in the process. Suppose we start with a bag in the following state.



[10]

a. Look at the previous prelab exercise and record the expression for generating a random position in the bag.

b. Given that we have a random position, we still need to access the node at that position. We need to scan a reference over the linked structure. Suppose that the random position was 2 (item "c"). If the reference starts at the first node, how many times do we need to move it?

c. Write a chunk of code that will create a reference `scout` and then use a loop to move it to the appropriate position in the linked structure.

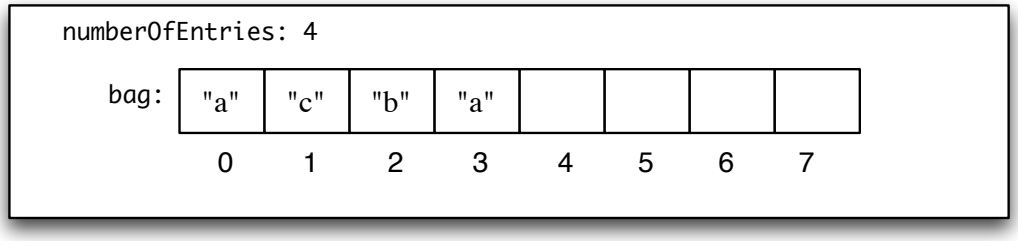
d. Write two lines of code to store the value of the item in the node referenced by `scout` into the variable `result`. Then copy the item at the first position into the node referenced by `scout`.

e. Finally, refer to the original code and write down the pieces of code needed to remove the first node from the chain.

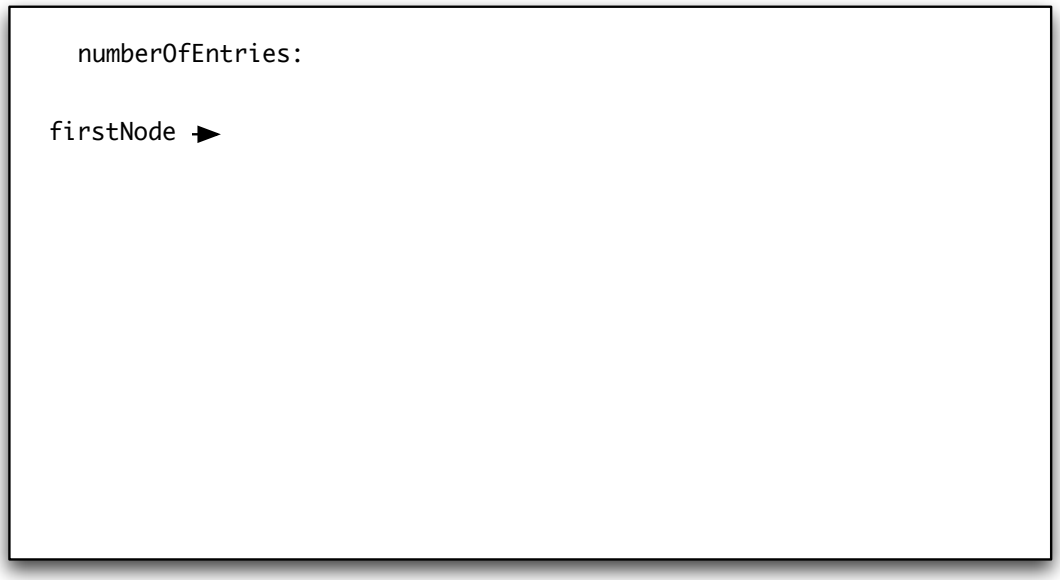
Put the pieces together to create code for the modified version of `remove`.

DuplicateAll

Suppose there is a bag with the following state:



[11]
What is a possible final state after `duplicateAll()`?



[12]
Let's think about what is needed to reach the final state.

a. We need to copy each of the items in the original bag. Write down a loop that will scan over each position of an item in the bag. (Remember to make it general.)

b. The body of the loop needs to create a node with the duplicate and add it to the bag. The easiest way to do this is to put the duplicate at the front of the bag. Write the body.

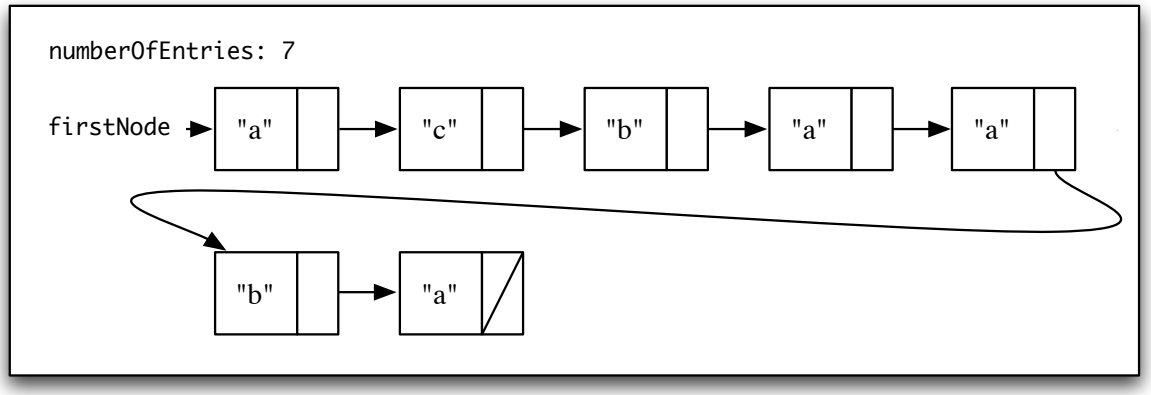
c. What is the increase in the number of items in the bag?

Using the above, write an algorithm to implement `duplicateAll`.

Remove Duplicates

Suppose there is a bag with the following state:

`this`



[13]

The approach we will use to remove the duplicates is different than what we did for the array implementation. Instead of removing the duplicates, we will create a second chain that holds the unique entries. We will still need to have nested loops. The outer loop will scan over the original chain. The inner loop will scan over the second chain to see if an item is already in the chain. If the item is not already in the chain of unique entries add it. Finally, after the outer loop is complete, change the `firstNode` reference and the `numberOfEntries`. (We could remove duplicate values from the chain as we scan over it, but modifying a chain as you scan it is inherently more complicated and liable to errors.)

What is a possible final state after `removeDuplicates()`?

[14]

a. Write an outer loop that will visit the position of each item in the chain of original items.

b. Write an inner loop that scans over the second chain of unique items and determines if a given item is already in the chain.

c. Write a chunk of code that will add the item at the front of the second chain and increments the count of the number of unique items

d. Write a couple lines of code that change the first node so that it references the chain of unique items and updates the number of entries.

Write an algorithm to implement `removeDuplicates`.

Directed Lab Work

The `LinkedBag` class is a working implementation of the *`BagInterface.java`*. The remove method already exists but needs to be modifed. The other three 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.

Equals

Step 1. Compile the classes `BagExtensionsTest` and `LinkedBag`. Run the `main` method in `BagExtensionsTest`.

Checkpoint: If all has gone well, the program will run and the test cases for the four methods will execute. Don't worry about the results of the tests yet. The goal now is to finish the implementation of each of our methods one at a time.

Step 2. In the `equals` method of `LinkedBag`, implement your algorithm from the pre-lab exercises. Some kind of iteration will be required in this method.

Checkpoint: Compile and run BagExtensionsTest. The tests for equals should all pass. If not, debug and retest.

Remove

Step 1. Look at the results from the test cases from the previous run of BagExtensionsTest. Since this is an existing method we want to make sure that our extension does not break the correct function of the method. All but the last two test cases should result in passes. The last two tests are intended to show the new behavior.

Step 2. In the `remove` method of `LinkedBag`, add the modifications from the pre-lab exercises.

Checkpoint: Compile and run BagExtensionsTest. All of the tests in the test remove section should now pass. If not, debug and retest.

Duplicate All

Step 3. In the `duplicateAll` method of `LinkedBag`, implement your algorithm from the pre-lab exercises. Iteration is needed.

Checkpoint: Compile and run BagExtensionsTest. All tests up through checkDuplicateAll should pass. If not, debug and retest.

Remove Duplicates

Step 4. In the `removeDuplicates` method of `LinkedBag`, implement your algorithm from the pre-lab exercises. This method will require some form of iteration. If you use the technique recommended in the pre-lab, you will use nested iteration.

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

Post-Lab Follow-Ups

1. We modified the `remove` method so that each of the items was equally likely to be removed. Which item we remove involves an interplay between add and remove. Instead of randomizing the item to remove, we could instead randomize where a new item is added into the chain. Modify the `add` method so that it adds the new item at a random location in the chain.

2. Implement a version of the `duplicateAll` method that will add the duplicates at the end of the linked structure. Note: You need to be careful with the loop termination condition.

3. Modify the `removeDuplicates` method so that it doesn't use a second chain. You could remove duplicates as you find them or swap items in the chain until all the duplicates are at the end where changing a single next reference to null will remove them all.

4. Implement and test a new method
```
boolean splitInto(BagInterface<T> first, BagInterface<T> second){
...
}
```
which will split and add the contents of the bag into two bags that are passed in as arguments. If there are an odd number of items, put the extra item into the first bag. This method will return a boolean value. If either bag overflows, return false. Otherwise, return true. Note that while you will directly access the chain of items for the bag that the method is applied to, you can only use the methods from `BagInterface` on the arguments. (Note that you don't even know if the bags in the arguments are implemented using arrays or chains of linked elements or some other representation.)

5. Implement and test a new method
```
boolean addAll(BagInterface<T> toAdd){
...
}
```
which will add all of the items from the argument into the bag. The method will return a boolean value indicating overflow (always false for this implementation.) Note that while you can directly access the chain of items of the bag that the method is applied to, you can only use the methods from `BagInterface` on the argument.

Another way of implementing a bag is not to have a reference to each duplicate, but instead only keep a single reference and a count of the duplicates. For each of the remaining questions, use this new implementation.

6. Change the inner class `Node` so that it has a count of the number of times an item is in the bag and then change the implementation of each of the methods in the bag interface. For example, when you add an item, first check to see if it is already in the bag. If so, just increment the count in the node. If it is not in the bag, add a new node with a starting count of one.

7. Redo the methods from the lab. Warning: The implementation of the randomized remove method is tricky. If you have a bag that contains eight "a"s and four "b"s, remove should be twice as likely to pick an "a" as it is to pick a "b".

8. Implement and test a new method
```
boolean isSet(){
...
}
```
which will return true if the bag is also a set (has no duplicates).

9. Implement and test a new method
```
T getMode(){
...
}
```
which will return the item with the greatest frequency. If there isn't a single item with the greatest frequency, return `null`.