Lab 3 Bag Implementations

Goal

In this lab you will explore the implementation of the ADT bag using arrays and linked lists. You will take an existing implementation and create new methods that work with that implementation. You will override the equals method so that it will determine if two bag are equal based on their contents. You will modify the remove method so that it will remove a random item from the bag. You will implement a duplicateAll method that will create a duplicate of every item in the bag. Finally, you will implement a removeDuplicates method that will guarantee that each item in the bag occurs only once by removing any extra copies.

Resources

- Appendix D: Creating Classes from Other Classes
- Chapter 1: Bags
- Chapter 2: Bag Implementations That Use Arrays
- Chapter 3: A Bag Implementation That Links Data

In `javadoc` directory

- *BagInterface.html*—Documentation for the interface BagInterface

In addition, there are also some online resources that you might find useful. The place to start for all things Java is www.oracle.com/technetwork/java/index.html. Here you can find links to free downloads like Java SE (The standard edition of the Java Platform). You can also find links to tutorials and documentation for Java. One useful starting point is the API documentation that documents all of the standard classes and their methods. At the time this was written, the most recent version of the standard edition was Java SE 8 and the API documentation was at docs.oracle.com/javase/8/docs/api. Please refer to the Java website for the most recent version.

Java Files

- *ArrayBag.java*
- *LinkedBag.java*
- *ArrayBagExtensionsTest.java*
- *LinkedBagExtensionsTest.java*
- *BagInterface.java*

Introduction

As was seen in the last lab, a bag is an unordered collection of items that may contain duplicates. It supports basic methods that allow one to add or remove items from the bag and query methods that allow one to determine if an item is contained in the bag. One basic way to implement a collection of items is to use an array. The other basic implementation is a linked structure. In this lab, you will take a working implementation and create some new methods. If you have not done so already, take a moment to examine the code in *ArrayBag.java* and *LinkedBag.java*.

To get a better feel for the implementation, lets consider the code that implements the add method.

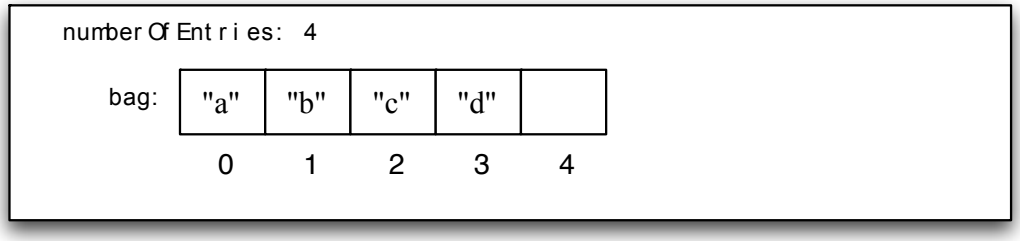
```
public boolean add(T newEntry) {
   checkInitialization();
   boolean result = true;
   if (isArrayFull()) {
      result = false;
   } else { // Assertion: result is true here
      bag[numberOfEntries] = newEntry;
```

```
        numberOfEntries++;
      } // end if
      return result;

   } // end add
```

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

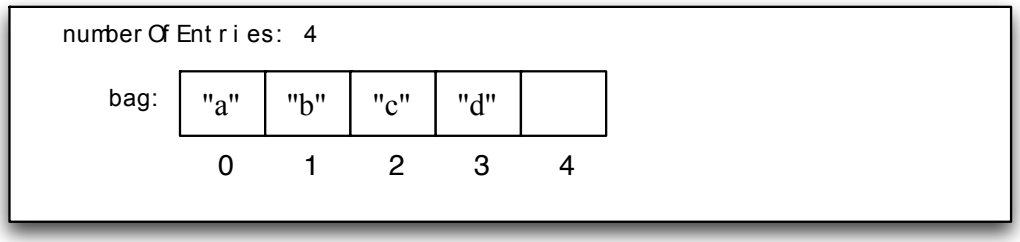
```
ArrayBag<String> x = new ArrayBag <String>(5);
x.add("a");
x.add("b");
x.add("c");
x.add("d");
x.add("e");          // trace this one
```

The initial state of the object is



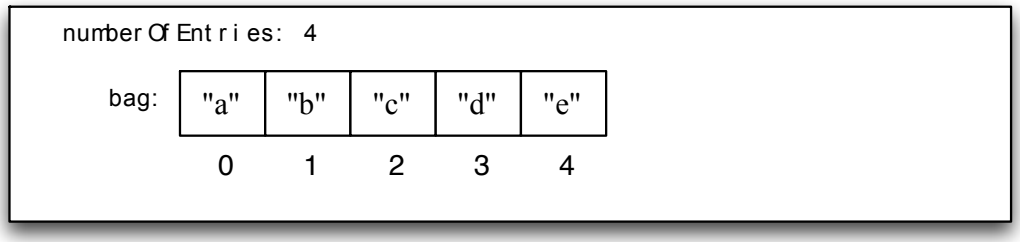
The variable result is initialized to true.

newEntry: "e"
result: true



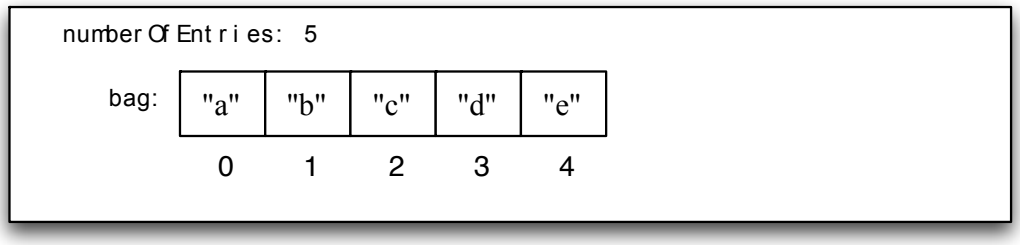
The bag is not full, so the else part will be executed. We the item in bag at numberOfEntries (4) to "e".

newEntry: "e"
result: true



Finally, we increment numberOfEntries by 1 and then return the value in result.

```
newEntry: "e"
result: true
```



We notice that entries are always added at the end of the collection and that the number of entries indexes the position that the next item will be added into.

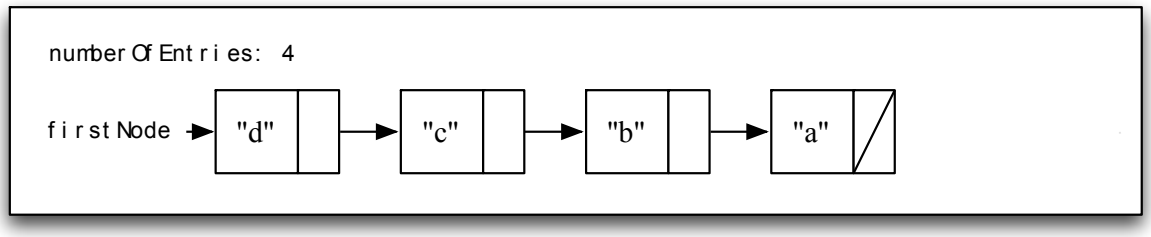
For the linked implementation, consider the code that implements the add operation.

```
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
```

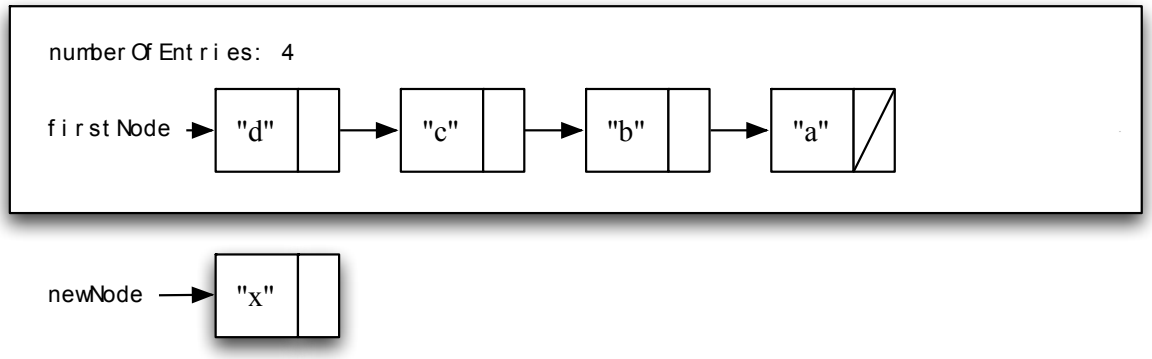
Let's trace the last statement in the following code fragment.

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

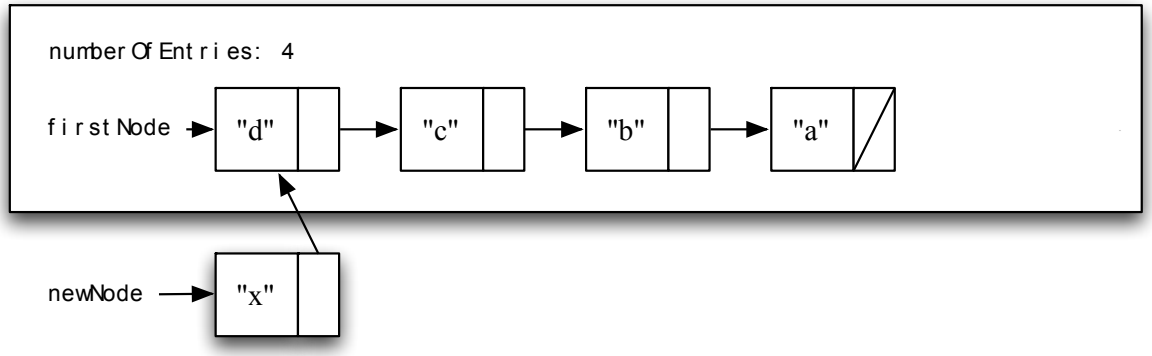
The initial state of the object is



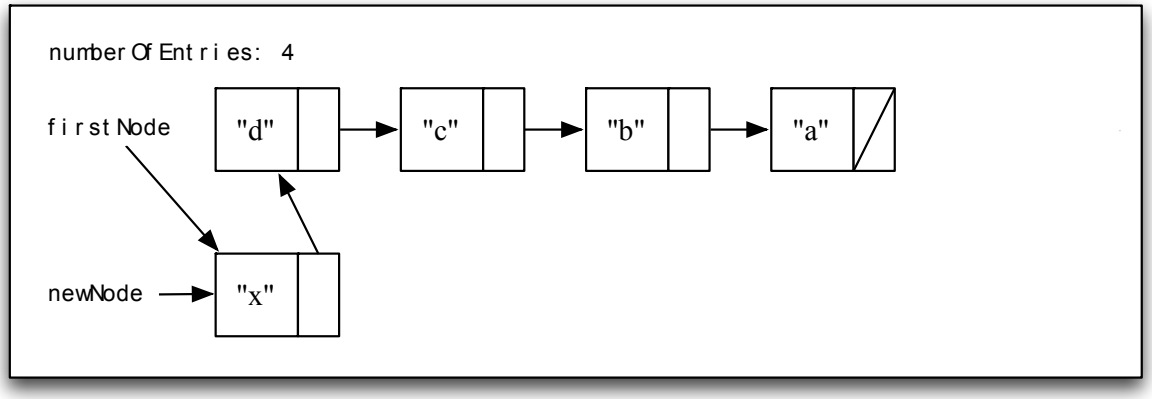
A new node is created.



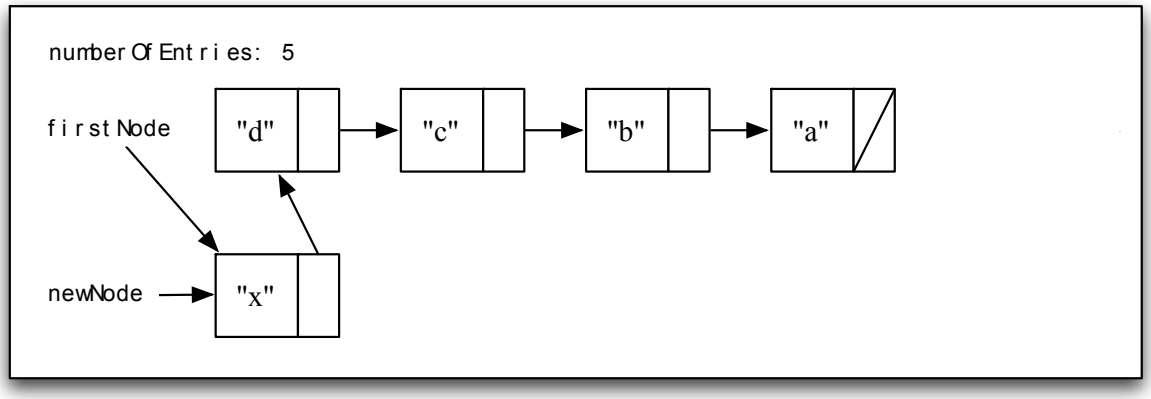
The next reference for the new node is set to be the first node.



The first node reference is set to the new node.



Finally, the number of entries is updated.



The first thing that you will do in this lab is to override the equals method. Every class inherits the equals method from the class Object. The inherited method determines if two objects are equal based on their identity. Only if two objects are located at the same memory location will the inherited equals method return true. Instead of this, we need to know if two bags are the same based on whether their contents are the same. The new version of equals will then be used to decide if the implementations of other methods you create are correct. Once the equals method has been completed correctly, the other three methods can be completed in any order.

In the current implementation of the remove method, the first item in the bag chain will be the item returned. While this is behavior is allowed under the postconditions of the method, it does not match the physical notion of a bag of items. For example, if we have a bag of marbles and reach in and remove one, we expect that each marble is equally likely to be selected. We will modify the remove method so that the item removed is randomly selected from all the items in the bag.

One operation that we might want to have available is to duplicate all the items in the bag. While we can do this using just the methods in the bag interface, we have a problem in that there is no convenient way to visit each item in the bag and add in a copy. For example, if we try an algorithm that repeatedly removes an item from the bag and then adds it back twice, we have no guarantee that all of the original items will be removed. It is likely that we will get duplicates of duplicates and that some items will not be duplicated at all. To deal with this issue, a second bag can be used. In this algorithm, remove all the items from the original bag and place them in the second bag. Now remove each item from the second bag and place two copies of it back into the original bag. While this algorithm will work, we can avoid the use of a second bag by working directly with the internal representation of the bag (which is what will be done in the lab).

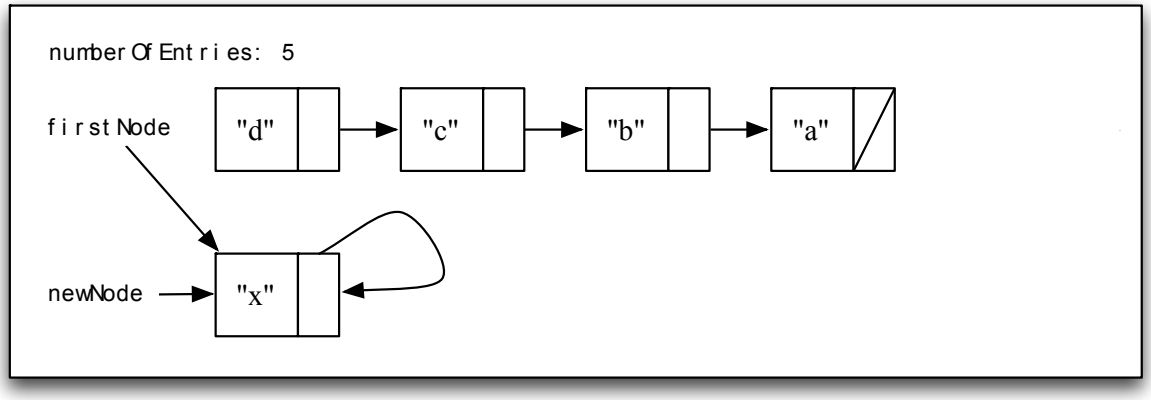
In mathematics, the concepts of bags and sets are closely related. The difference is that bags allow duplicate items, while a set does not. An operation that removes the duplicates from a bag would be helpful if one wished to implement a set. As with the duplicate all method, the inability to visit each item in the bag using the bag interface methods motivates a method that works directly with the internal representation of the bag.

For the linked implementation, 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.)

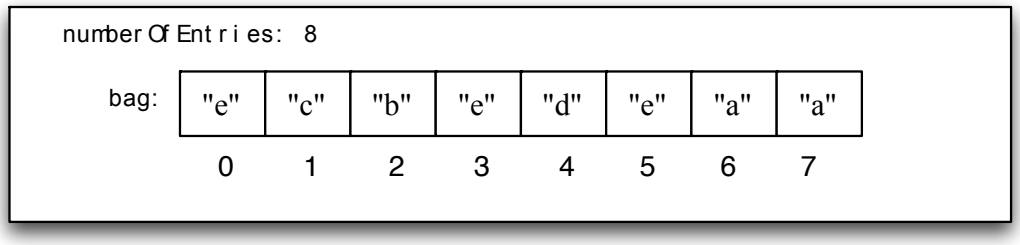


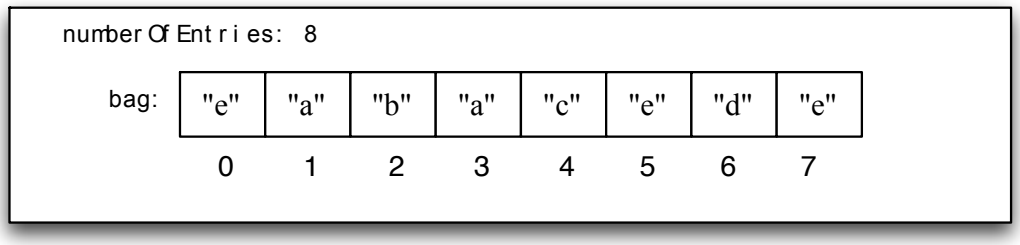
To avoid problems with surgery, it is always a good idea to carefully trace the intended operation of methods before implementing the code.

Pre-Lab Visualization

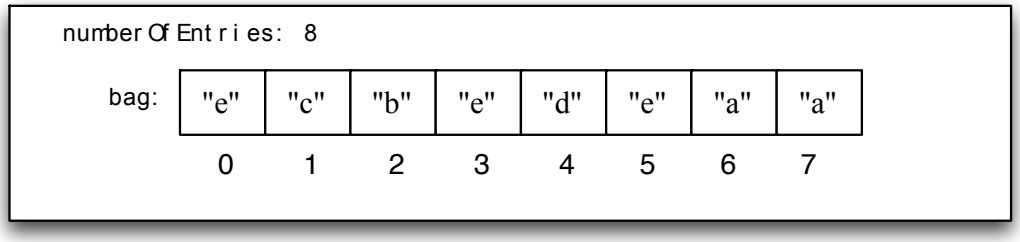
Equals

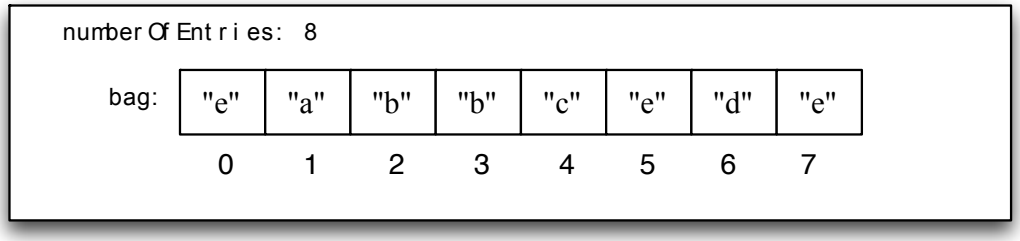
One way to determine that two bags are equal is by comparing the frequencies of the items in the bags. Consider the following two bags. Which items and frequencies need to be compared to determine that they are equal?



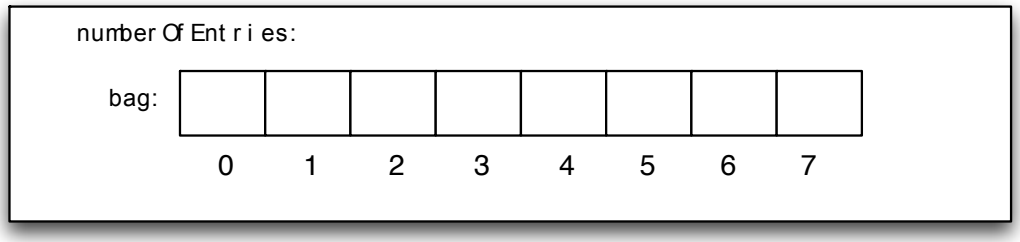


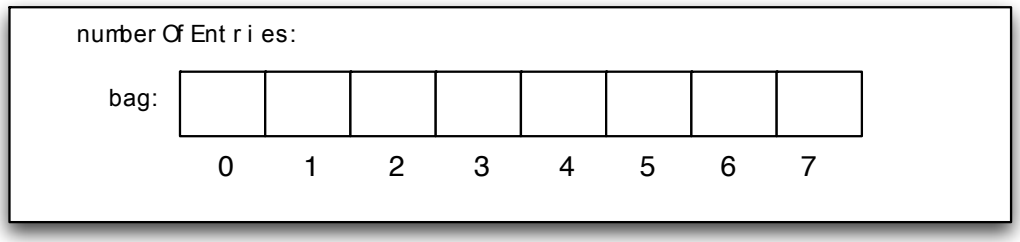
Consider the following two bags. Which items and frequencies need to be compared to determine that they are not equal?





Give an example of two bags that cannot be equal, yet no item comparisons are needed to make that determination.

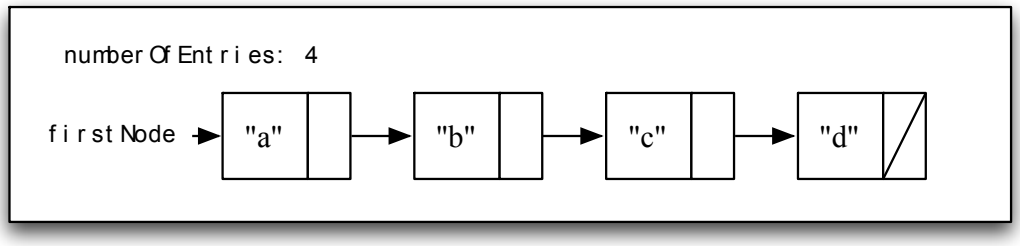




Write an algorithm that returns true if the items in two bags have the same frequencies. (Hint: Scan over the items in one bag and use the method getFrequencyOf() with both bags.)

Remove

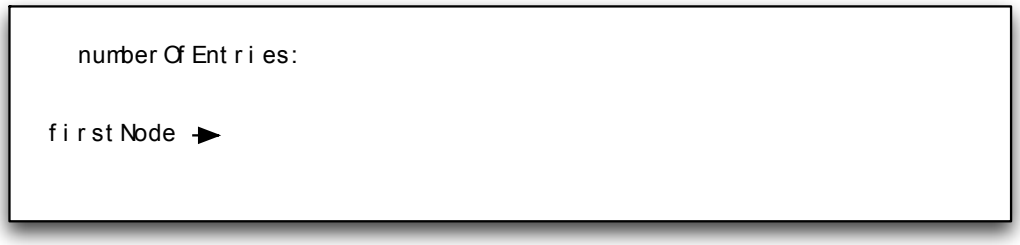
Suppose there is a bag with the following state:



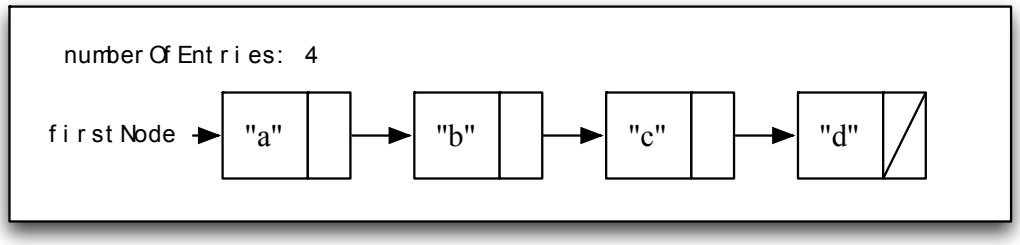
The existing code for the remove method is as follows.

```
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?



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 array-based implementation. 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.



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.

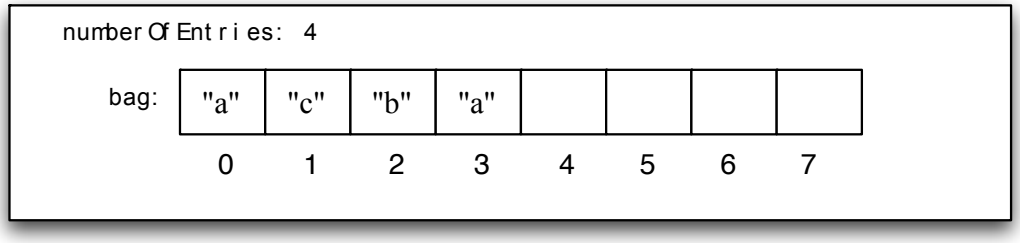
d. Write two lines of code to store the value of the item in the node referenced by scout into the variable result. Then copy the item at the first position into the node referenced by scout.

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

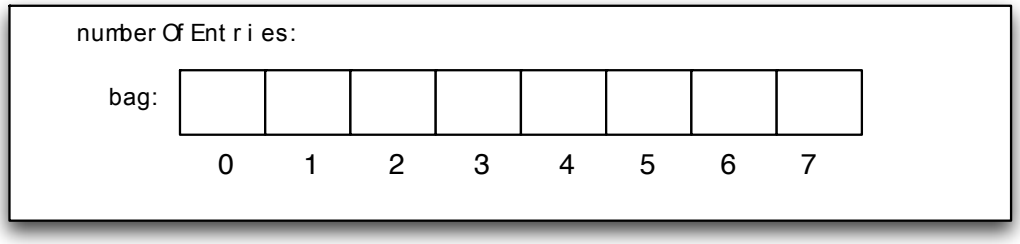
Put the pieces together to create code for the modified version of remove.

DuplicateAll

Suppose there is a bag with the following state:



Give a possible final state after duplicatAll()?



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 cover each position of an item in the bag. (Remember to make it general.)

b. The body of the loop needs to copy the items. How many positions away is the copy from the original? Write a statement to do that copy.

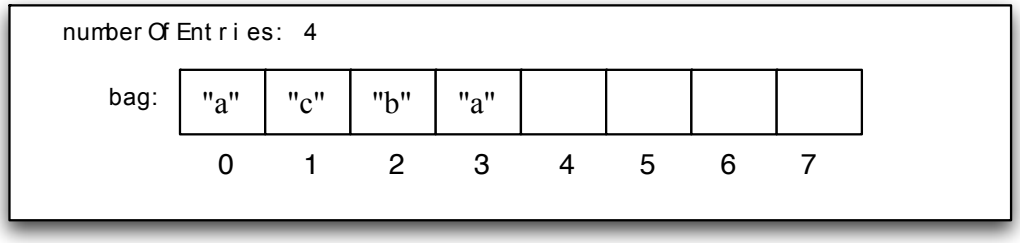
c. What is the increase in the number of items in the bag?

d. Since our bag has limited space, not all bags can be duplicated. Give a test condition to determine if the duplicate all method can succeed.

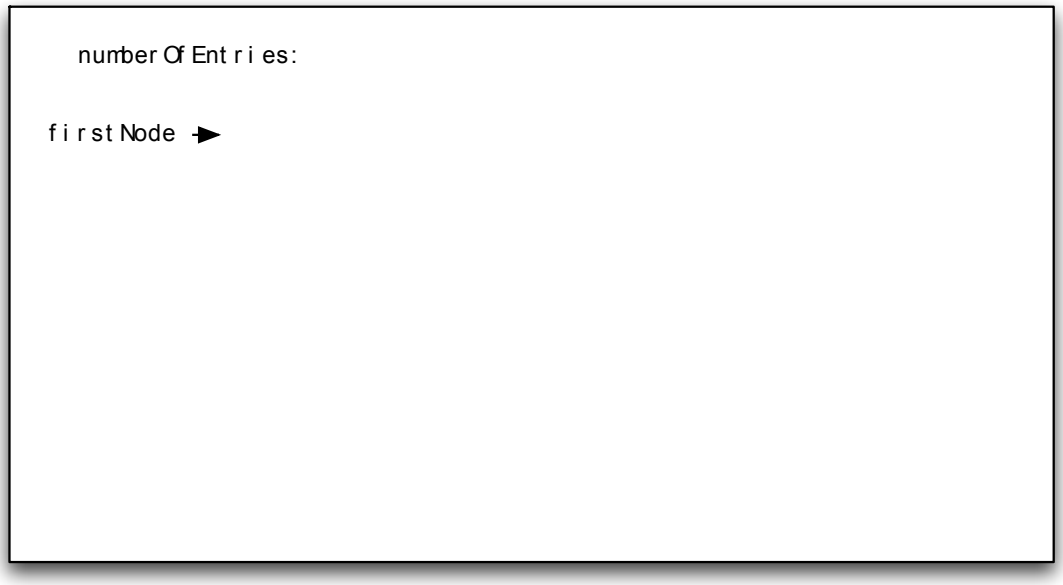
Using the above, write an algorithm to implement duplicateAll.

DuplicateAll (Linked Implementation)

Suppose there is a bag with the following state:



What is a possible final state after duplicateAll()?



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.)

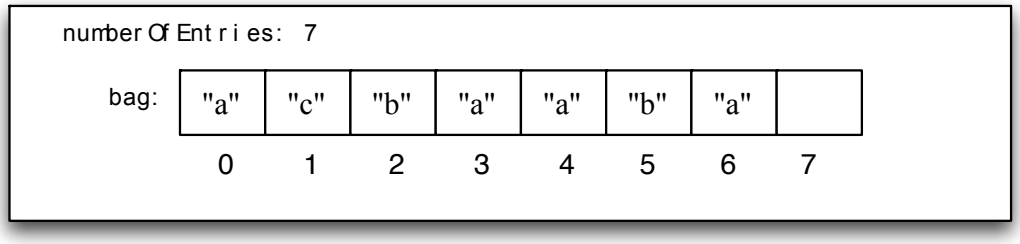
b. The body of the loop needs to create a node with the duplicate and add it to the bag. The easiest way to do this is to put the duplicate at the front of the bag. Write the body.

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

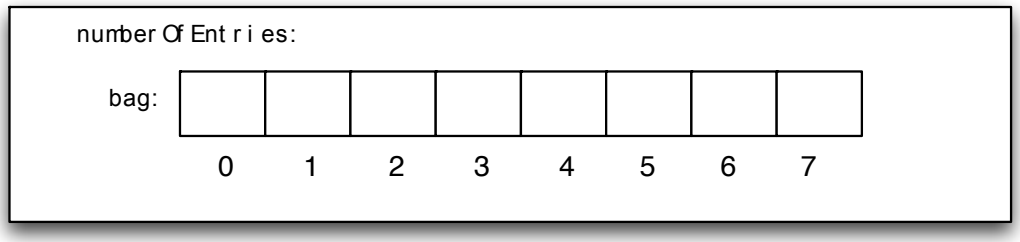
Using the above, write an algorithm to implement duplicateAll.

Remove Duplicates

Suppose there is a bag with the following state:



What is a possible final state after removeDuplicates()?



One way to remove the duplicates is to use a pair of nested loops. The outer loop will scan over unique items in the bag, while the inner loop will remove the duplicates that come afterwards.

a. Write an outer loop that will visit the position of each item in the array.

b. There are a couple ways to do the inner loop. One idea is to use a while loop to scan over the remaining items and remove each duplicate that we come across. (See the followup exercises for another technique.) To remove an item we will use the same technique that is used by the removeEntry method. Copy the last item over the item to be removed and then replace the last item with null. The first time

the inner loop runs on our sample bag, it will visit each of the last 6 items. Record the state of the bag and the loop index at the start and then after each iteration of the while loop.

index:

numberOfEntries: 7

bag:

"a"	"c"	"b"	"a"	"a"	"b"	"a"	
0	1	2	3	4	5	6	7

index:

numberOfEntries:

bag:

0	1	2	3	4	5	6	7

index:

numberOfEntries:

bag:

0	1	2	3	4	5	6	7

index:

numberOfEntries:

bag:

0	1	2	3	4	5	6	7

index:

numberOfEntries:

bag:

0	1	2	3	4	5	6	7

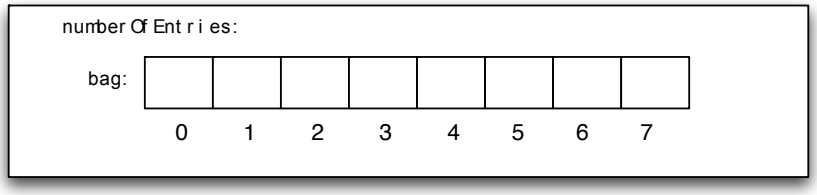
index:

numberOfEntries:

bag:

0	1	2	3	4	5	6	7

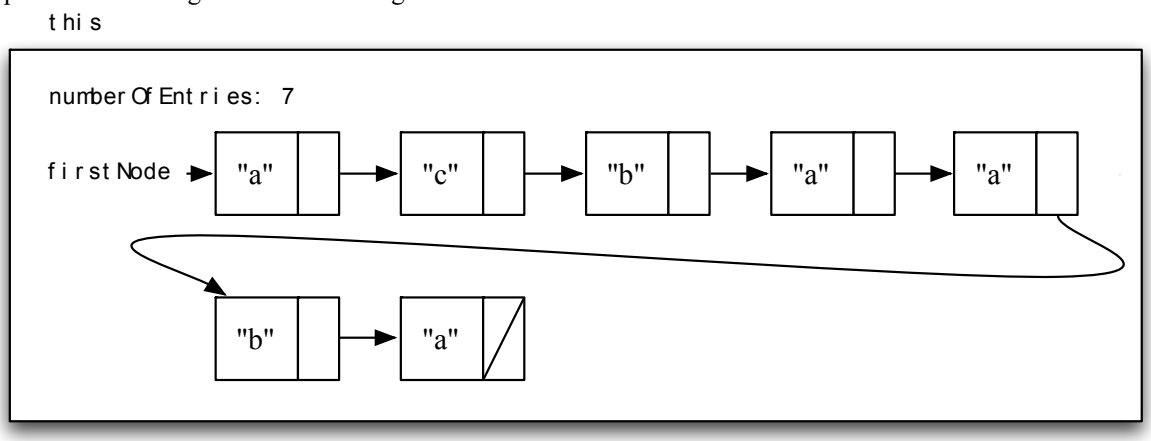
index:



Write an algorithm to implement removeDuplicates.

Remove Duplicates

Suppose there is a bag with the following state:



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()?

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 ArrayBag and LinkedBag classes are working implementations 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 ArrayBagExtensionsTest and ArrayBag. Run the main method in ArrayBagExtensionsTest.

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 ArrayBag, implement your algorithm from the pre-lab exercises. Some kind of iteration will be required in this method.

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

Duplicate All

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

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

Step 2. In the duplicateAll method of LinkedBag, implement your algorithm from the pre-lab exercises. Iteration is needed.

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

Remove Duplicates

Step 1. In the removeDuplicates method of ArrayBag, 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 ArrayBagExtensionsTest. All tests should pass. If not, debug and retest.

Step 2. 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 LinkedBagExtensionsTest. All tests should pass. If not, debug and retest.

Post-Lab Follow-Ups

1. Implement the duplicateAll method using just methods from the BagInterface along with a second bag.

2. Implement the removeDuplicates method using the private method removeEntry.

3. Implement the removeDuplicates method using just methods from the BagInterface along with a second bag.

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. The method will return a boolean value. If either bag overflows, return false. Otherwise, return true. Note that while you will directly access the array of the bag that the method is applied to, you can only use the methods from BagInterface on the arguments.

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 an overflow. If adding the items would cause the bag to overflow, do nothing and return false. Otherwise, add the items and return true. Note that while you will directly access the array of the bag that the method is applied to, you can only use the methods from BagInterface on the argument.

6. Implement and test a new method

   ```
   boolean isSet(){
   ...
   }
   ```

   which will return true if the bag is also a set (has no duplicates).

7. 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.

8. Modify the removeDuplicates method in LinkedBag 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.

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.

9. 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.

10. 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".