Array Based Bag Implementation

Goal

In this assignment you will explore the implementation of the ADT bag using arrays. 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 C: Creating Classes from Other Classes
- Chapter 1: Bags
- Chapter 2: Bag Implementations That Use Arrays

In `javadoc` directory

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

Java Files

- *`ArrayBag.java`*
- *`BagExtensionsTest.java`*
- *`BagInterface.java`*

Introduction

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 assignment, 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`*.

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

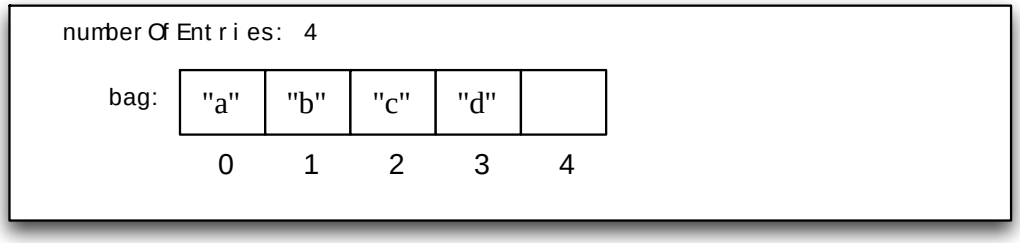
```
public boolean add(T newEntry) {
    checkIntegrity();
    boolean result = true;
    if (isArrayFull()) {
        result = false;
    } else { // Assertion: result is true here
        bag[numberOfEntries] = newEntry;
        numberOfEntries++;
    } // end if
    return result;

} // end add
```

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

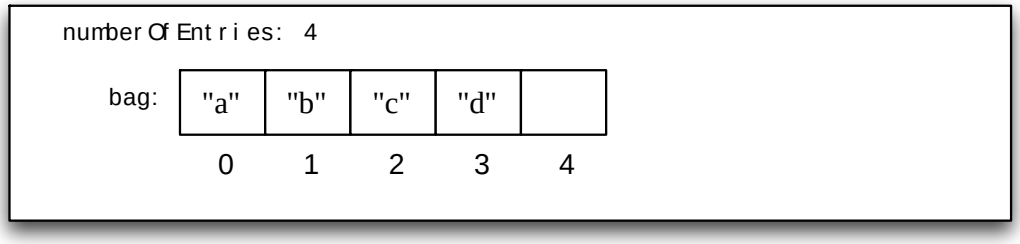
```
ArrayBag<String> x = new ArrayBag <String>(5);
x.add("a");
x.add("b");
x.add("c");
x.add("d");
x.add("e");      // trace this one
```

The initial state of the object is



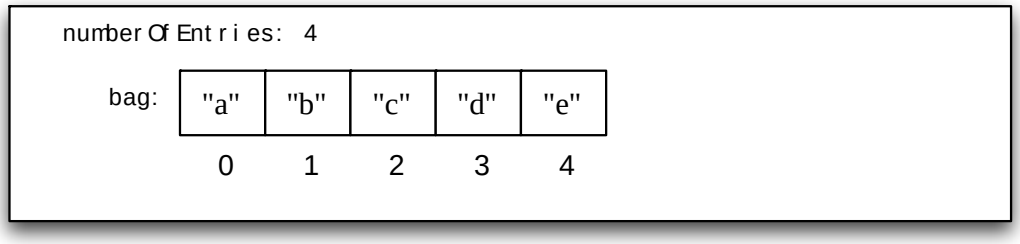
The variable `result` is initialized to `true`.

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



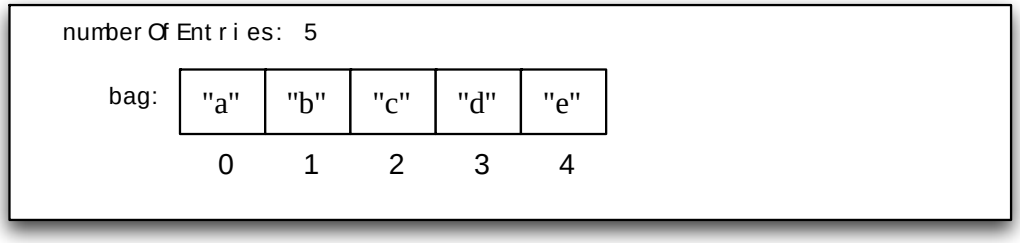
The bag is not full, so the else part will be executed. We the item in `bag` at `numberOfEntries` (`4`) to "e".

```
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.

The first thing that you will do in this assignment 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 last item in the bag array 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.

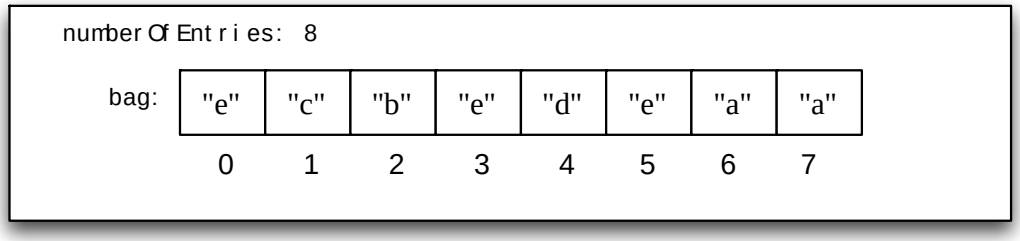
One operation that we might want to have available is to duplicate all the items in the bag. While we can do this using just the methods in the bag interface, we have a problem in that there is no convenient way to visit each item in the bag and add in a copy. For example, if we try an algorithm that 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.

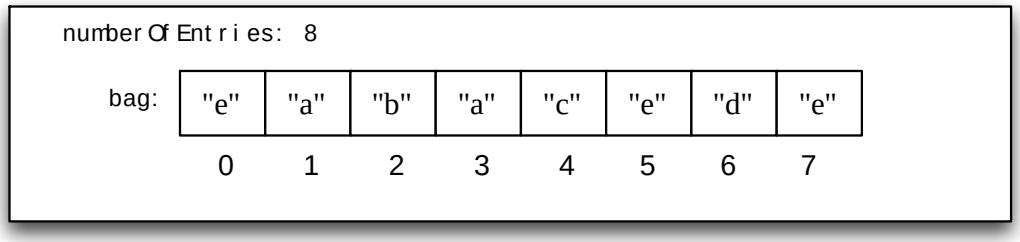
In mathematics, the concepts of bags and sets are closely related. The difference is that bags allow duplicate items, while a set does not. An operation that removes the duplicates from a bag would be helpful if one wished to implement a set. As with the duplicate all method, the inability to visit each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

Visualization

Equals

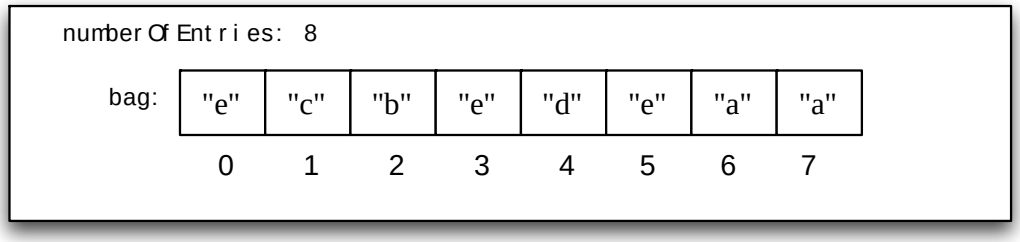
One way to determine that two bags are equal is by comparing the frequencies of the items in the bags. Consider the following two bags. Which items and frequencies need to be compared to determine that they are equal?

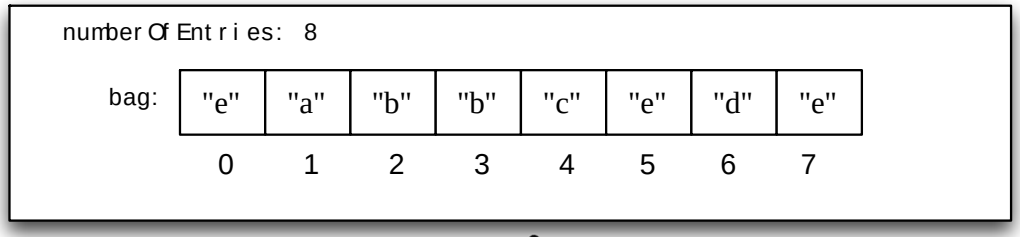




The frequencies of "a", "b", "c", "d", and "e" need to be compared. If all frequencies are equal, the bag is equal

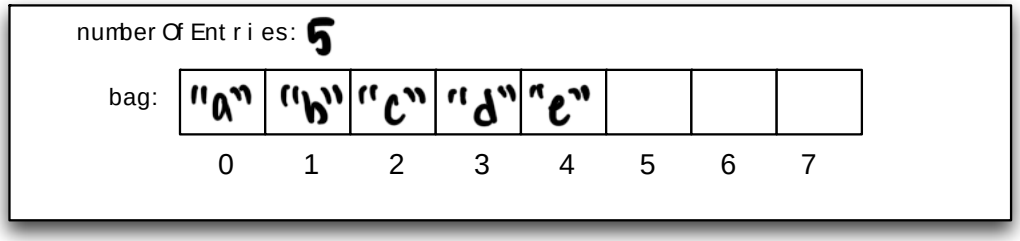
Consider the following two bags. Which items and frequencies need to be compared to determine that they are not equal?

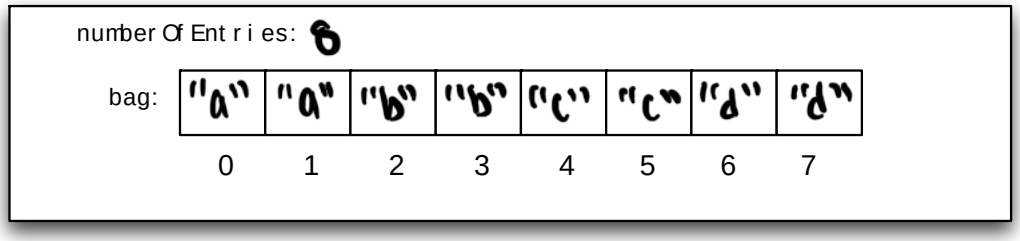




Comparing the frequencies of "b" will conclude the bags are unequal

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



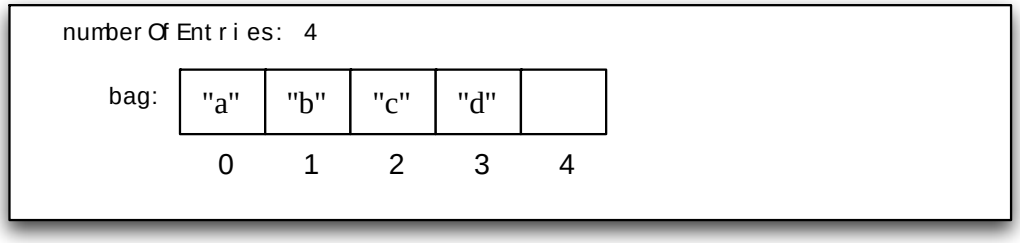


Write an algorithm that returns true if the items in two bags have the same frequencies. (Hint: Scan over the items in one bag and use the method `getFrequencyOf()` with both bags.)

```
if (b1.contains(anEntry)) {
   if (b1.getFrequencyOf(anEntry) && b2.getFrequencyOf(anEntry)
      return True;
   else
      return False;
}
```

Remove

Suppose there is a bag with the following state:

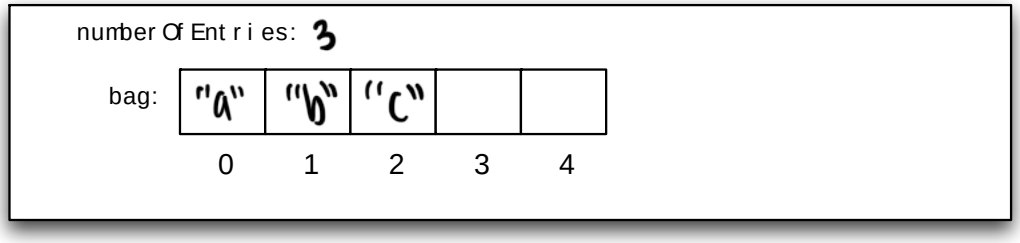


The existing code for the remove method is as follows.

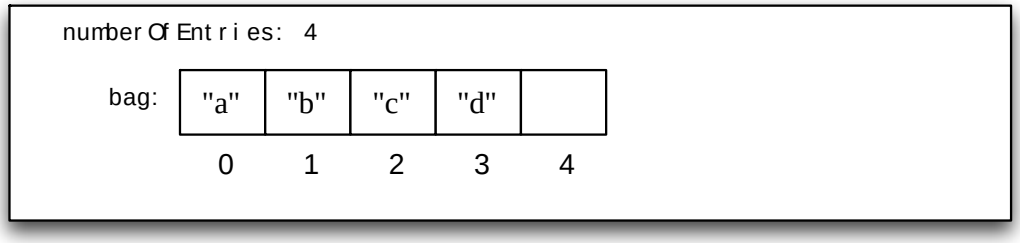
```
public T remove() {
    checkInitialization();
    T result = removeEntry(numberOfEntries - 1);
    return result;
} // end remove

private T removeEntry(int givenIndex) {
    T result = null;
    if (!isEmpty() && (givenIndex >= 0)) {
        result = bag[givenIndex];                    // entry to remove
        bag[givenIndex] = bag[numberOfEntries - 1]; // Replace entry with last
                                                     // entry
        bag[numberOfEntries - 1] = null;             // remove last entry
       numberOfEntries--;
     } // end if
    return result;
} // end removeEntry
```

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. Lets examine the steps in the process. Suppose we start with a bag in the following state.



a. First, we need to generate a random position in the array. What are the largest and smallest positions that are occupied by an item? (Give you answer in general terms that will work for any bag, not just this specific example.)

smallest ⇒ 0

largest ⇒ MAX_CAPACITY / getCurrentSize()

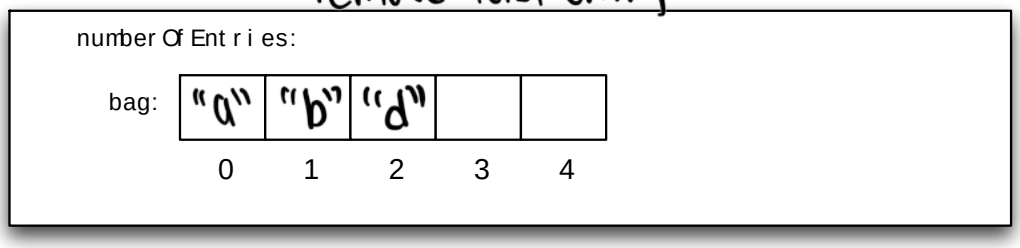
b. Look at the documentation for the class `java.util.Random`. Write an expression that uses an instance of this class to create a valid random position in the bag array.

position = random.nextInt(getCurrentSize())

c. Suppose that the random position generated was 2. Trace the operation of the removeEntry method and show the final state of the bag.

"c" and "d" swapped

`result` remove last entry



d. Suppose that the bag is empty. What should the remove method do in this case?

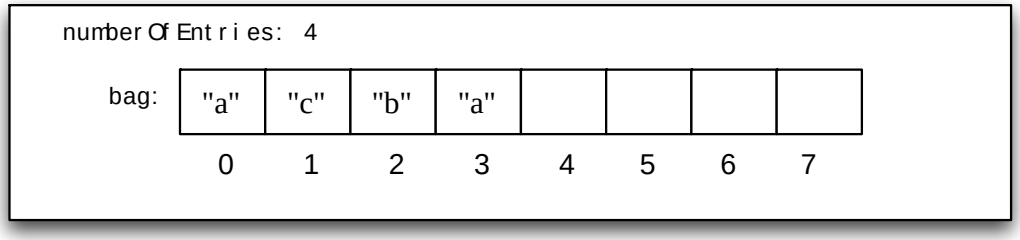
throw IllegalArgumentException

Write an algorithm to implement the modified version of `remove`.

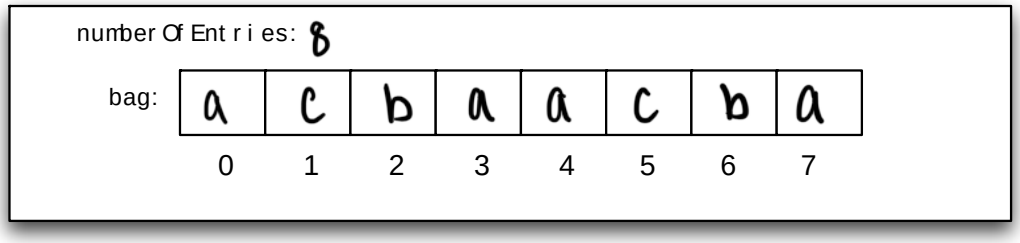
```
public T remove() {
    checkInitialization();
    Random rand = new Random();
    int position = rand.nextInt(getCurrentSize());
    T result = randomEntry(position);
    return result;
}
```

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.)

```
T[] arr = toArray();
for (int i = 0; i < getCurrentSize(); i++)
    // code
}
```

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.

The copy is getCurrentSize() away

```
add(arr[i]);
```

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

getCurrentSize() * 2

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.

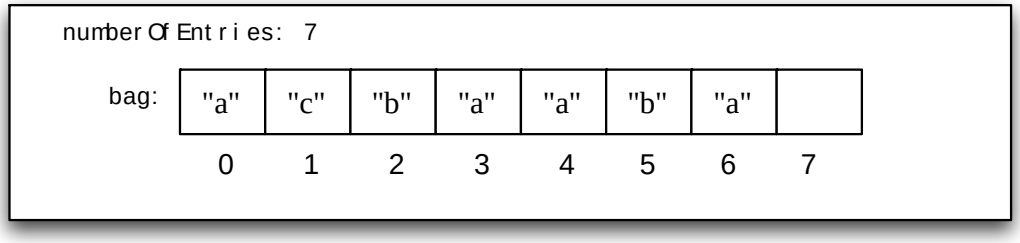
```
if( getCurrentSize() * 2 < MAX_CAPACITY)
```

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

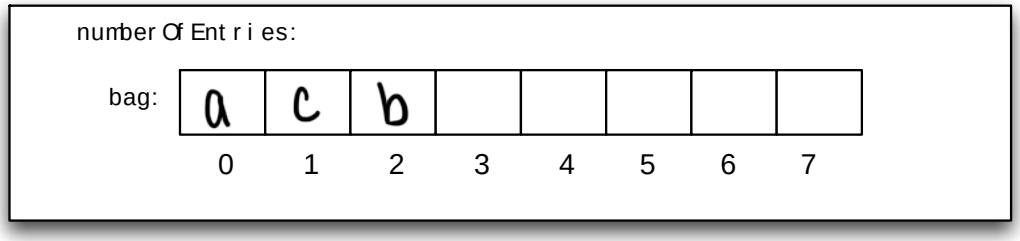
```
public boolean duplicateAll() {
  if (getCurrentSize() * 2 < MAX_CAPACITY) {
    T[] arr = toArray();
    for (int i = 0; i < getCurrentSize(); i++) {
        add(arr[i]);
    }
}
```

Remove Duplicates

Suppose there is a bag with the following state:



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



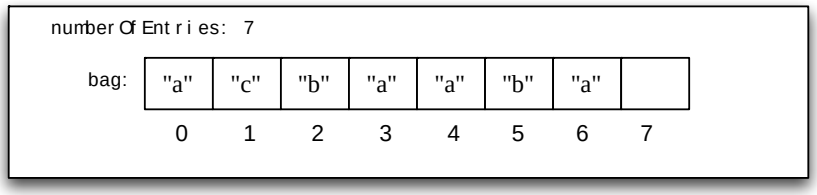
One way to remove the duplicates is to use a pair of nested loops. The outer loop will scan over unique items in the bag, while the inner loop will remove the duplicates that come afterwards.

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

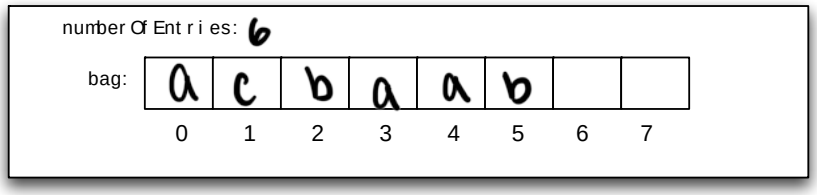
```
for (int i = 0; i < getCurrentSize(); i++) {
    // code
}
```

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.

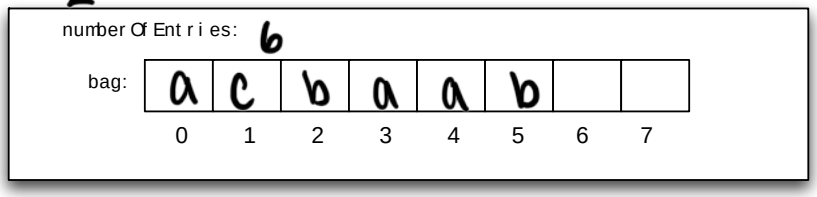
index: 0



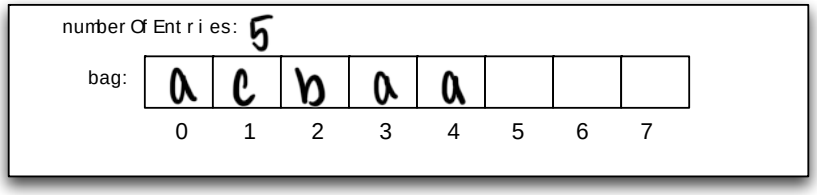
index: 1



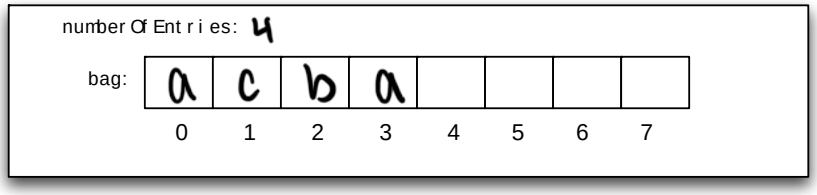
index: 2



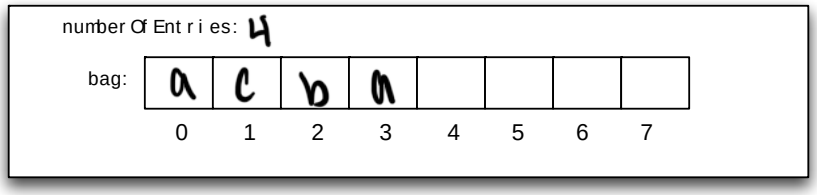
index: 3



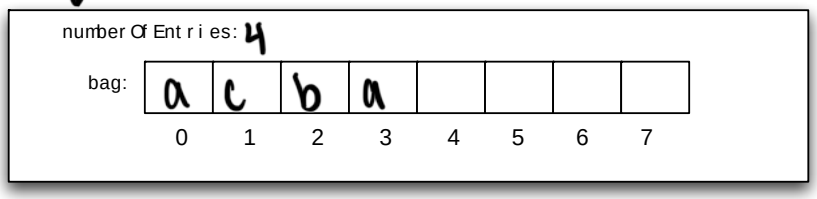
index: 4



index: 5



index: 6



Write an algorithm to implement `removeDuplicates`.

```
for (int i = 0; i < getCurrentSize(); i++) {
    check = bag[i];
    while ( i < getCurrentSize()) {
        i++;
        if (check == bag[i])
            removeEntry(i);
    }
}
```

Directed Work

The `ArrayBag` 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 `ArrayBag`. 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 `ArrayBag`, implement your algorithm from the 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 `ArrayBag`, add the modifications from the 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 `ArrayBag`, implement your algorithm from the 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 `ArrayBag`, implement your algorithm from the exercises. This method will require some form of iteration. If you use the technique recommended in the visualization section, you will use nested iteration.

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

Follow-Ups

1. Create test cases for the other methods in `ArrayBag`.

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

3. Use the following idea to implement the body of the outer loop of the `removeDuplicates` method. We can scan over each of the remaining items in the bag and "slide over" all the ones that are not duplicates of item from the outer loop. You will need two indices for this inner loop, one does the scan and the other marks the location of the slide. If an item is not a duplicate, copy it to the slide position and increment the slide index. If an item is a duplicate, do nothing. After this is done, we need to replace any items from after the final slide position with null.

4. Implement the `removeDuplicates` method using the private method `removeEntry`.

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

6. 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.

7. 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.

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`.