Notes on Bags & Dense Trees

T. Reinhardt

1 Overview

A Bag<T> is a Java data-structure that implements multisets. Recall that objects that implement the Set<T> interface are unordered collections of unique elements. Thus, repetition is not allowed; or, more correctly, repetition is ignored. Thus \( S = \{1, 2, 3,\} \) and \( T = \{1, 2, 1, 2, 1, 2, 2, 3\} \) are the same set (i.e., they are indistinguishable), and we write \( S = T \).

Multisets are defined to bridge the gap from abstract sets to lists or sequences, where repetition is allowed and two sequences differ by their elements and, perhaps, by the number of times that each element appears in one or the other. Thus, in addition to cardinality, these multisets give us the additional property of multiplicity, i.e., the number of times that an element appears within the multiset.

Revisiting sets \( S \) and \( T \), imagine the function \( m(e, S) \) that returns the number of times that \( e \) appears in the set \( S \): \( m(1, S) = 1 \) but \( m(1, T) = 3 \). Throughout these notes, I will use the term count to mean multiplicity . . . it’s easier all around. Finally, we will limit the codomain of \( m \) to the natural numbers, \( \mathbb{N} \), which we will say comprise the integers \( > 0 \), i.e., \( \{1, 2, \ldots\} \). In this way we don’t spend too much time musing about the one multiset that contains the universe, etc. Interestingly, this means that we don’t really explicitly address the question of the empty set.¹

1.1 Defining a Bag

Many of the observations and comments that we make about bags, it so happens, will also be relevant to the discussion of DenseTrees as well. These structures differ in that one is unordered (the Bag) while the other ordered, the tree.

As discussed in class, I could think of a bag simply as a multiset and represent it directly as a collection over which I provide the required methods. That being said, I prefer to think of a multiset as a collection of maps, \( K \rightarrow \mathbb{N} \), where \( K \) is a set of unique keys, and \( \mathbb{N} \) denotes the natural numbers (defined above).

Several important considerations should be kept in mind:

¹In other words: we avoid philosophical questions about the “universe” just being a multi set whose elements are \( \geq 0 \), so that even the missing elements exist in our set, it’s just that their multiplicity is 0. While this is a fun discussion, it doesn’t result in useful code . . .
1. Because we need to keep track of the number of times that an element appears in the set, we need to store a “count” as an integer value;

2. Because we need to return the contents of the bag as an iteration, we will need to provide \( n \) instances of the element on demand; and,

3. The “size” of a bag needs to take the multiplicities (counts) of each element as well.

1.2 Defining a Dense Tree

Everything that has been said about bags applies to Dense Trees except that trees are ordered. Because explicit duplicates are not allowed in our Dense Tree, the ordering should be left-branches contain elements that are < and the right branch contains elements that are < their root (parent) nodes, respectively.

2 Implementation details

First, you should review the Java documentation for the classes HashMap and TreeMap, as these are relevant to these kinds of structures. Naturally, I would expect you to implement your own Bag and DenseTree classes.

You may use a HashMap, for example, as the backing store for your Bags class.

You may not use TreeMap as the backing store for your DenseTree, however.
2.1 Common methods

It’s instructive to review the general (common) operations that these classes need to provide:

<table>
<thead>
<tr>
<th>Method</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(E e)</td>
<td>Adds an element to the Bag or Dense Tree. Note, if the element already resides in the structure, then an internal count is incremented, otherwise, a new entry for this element is created and is installed in a location appropriate to the structure.</td>
</tr>
<tr>
<td>contains(E e)</td>
<td>Returns true if e is found in the structure.</td>
</tr>
<tr>
<td>count(E e)</td>
<td>Returns an integer (≥ 0) indicating how many times e appears within the structure.</td>
</tr>
<tr>
<td>remove(E e)</td>
<td>Throws an exception if e is not contained within the structure. Otherwise, decrements the count associated with e. Should this count subsequently go to 0, then e is removed from the structure.</td>
</tr>
<tr>
<td>asSet()</td>
<td>Returns the objects contained within the structure as the appropriate kind of set—i.e., unordered or ordered for Bags and DenseTrees, respectively.</td>
</tr>
<tr>
<td>size()</td>
<td>Returns the number of items in the structure; the size must take multiplicities into account.</td>
</tr>
<tr>
<td>iterator()</td>
<td>Returns the appropriate Iterator for the structure. Note, all iterators must provide the proper number of objects taking into account their multiplicities. Iterators on Dense Trees, however, have the additional requirement of maintaining the natural ordering of elements in the tree, which is in-order.</td>
</tr>
</tbody>
</table>

Dense Trees have two additional methods: one to find the minimum or maximum elements.

2.2 Back to sets . . .

Notice that both classes require the asSet() method that takes their respective elements to sets. In the case of Bags, the mapping from keys (objects in the bag) to sets is pretty straightforward. But, in the case of Dense Trees, however, we have several options to consider: should the set be ordered? If so, which ordering? Binary search trees have three common orderings: pre-, in-, and post-ordering. Generally, human beings are expecting an in-order arrangement of the underlying elements, so your implementation should deliver an in-ordering listing of elements.

Our task is simplified by observing that the sets created and returned by the toSet() methods are immutable. Moreover, an examination of the Set interface, reveals a fair number of optional methods. All of this suggests a reasonable course of action might be to define an inner class that depends upon a well-known and trusted collection type that
implements just the methods from the Set interface that you require. Encapsulate that object and implement the interfacing, throwing UnsupportedOperationException where necessary.

By the way: the various tests for the Bag class make no assumptions about ordering—as far as I know that someone will ask this, so I will just say it: The only ordering that is tested is for the Iterator on the Dense Tree class.

2.3 The devilish details . . .

Of course, the nuts and bolts that you will require for successful implementation is found in the documentation that ships with the Project—i.e., in the refman.pdf file. Naturally, we will be discussing the nitty gritty details in class over the next week or so, and you should feel free to bring your questions to class. Finally, this project should be much easier for you having completed the BST Project and the Binary Trees Lab. You may concurrently be working on the Basic Sets Lab too (depending upon how the semester’s work load has played out). In any event, these assignments are intended to be related, mutually supportive, and generous in giving you an opportunity to succeed in several settings.

3 Grading, etc

The test suite comprises 14 Public and 12 Release tests. 90% of your grade will be determined by the results of these, with a slightly heavier emphasis on the Public tests—because there are more of them and their results or more accessible to you.

Please read carefully through the guidelines at the top of the document (right before the table), and within the manual that ships with the project. We don't expect you to implement a lot of Java’s Set interface, but you will be required to implement Iterator logic, as well as to use your own Tree representation as opposed to relying on Java’s library.

A final piece of advise here: try to keep the implementation of the Dense Tree straightforward; unlike the Polymorphic Binary Search Tree Project, you need only a consistent Binary Search Tree that does not allow duplicates—that information is stored as the count property on the tree’s Node!