CMPU203 Maya T. Hawkins

November 7, 2014

Finite Bags

A finite bag is a grouping of a certain type of items. This could hold numbers, strings, swords, cars. What makes this different than some other implements of storage, however is that it stores the number of times a certain item is in the finite bag. Say that a finite bag holds swords. It could have 5 long swords and 7 short swords. A finite bag only stores one type of thing though. It will not store candy and fruit in the same bag, as they belong to different classes.

In this project, the finite bag was implemented through the design of a search tree. The elements of the search tree that I looked into specifically were strings and numbers, though I designed the search trees to take in any type of element, as you will see later. The search trees would also be balanced, but unfortunately, that has been programmed into the finite bags. They would have been designed through the Red-Black Balanced Tree design. As you will see in the program, there are places for where the design would be implemented, but until then, do nothing.

First to mention is that FiniteBags extends Comparable, which is necessary to compare the items in FiniteBags, since not only numbers might be put into them (see line 14). FiniteBags also extends Sequence, which extends Sequenced. Sequence and Sequenced is necessary looking at the items within FiniteBags. Rather than looking through a left and right, the user will be able to use functions like here, to locate the first in FiniteBags, notEmpty to know if there is anything in the FiniteBag, and next to get the next item within FiniteBags (look at lines 7-10). It is similar to a list, except there is less functions. It is helpful that it can have anything inside of it as well.

The FiniteBags interface laid out the functions that would be necessary to create the rules for FiniteBags. For an empty FiniteBag, it is treated as similar to an empty tree. For the FiniteBags empty tree, it called is Things_MT, since anything can be in a finite bag. Things_MT has all the functions that a FiniteBags has, since it implements FiniteBags. Yet the only information that an empty search tree would have is its color, which would naturally be black, as leafs of a Red-Black Balanced Tree are black (see lines 38-44). Rather having the color being specifically labeled, I used a Boolean for whether the leaf is black or not. It doesn't really matter what other color the leaf or node is, and working with a Boolean is easier to wrap your mind around rather than changing colors. The functions created describe that any Things_MT is empty (see lines 57-63), has 0 for its size (see lines 50-52), is black, and has no counts of elements (see lines 46-47). In addition, any item can be added to it of any amount (see lines 65-71) and unionized to any other FiniteBags (see lines 85-87). However, when trying to remove an item, it will return itself, as it has nothing inside to remove (see lines 73-83). Things_MT only has an empty set in common with other FiniteBags as well (see lines 89-95). This also means that it is a subset of any FiniteBags, as an empty tree is the subset of any other tree (see lines 106-108). A Things_MT only is equal to another Things_MT, so for the function, I check if the other FiniteBags is empty, in which case, it is equal to a Things_MT (see lines 97-104). As for the function blacken, which would normally be change a node's isBlack state to true, would just return a new Things_MT (see lines 110-112).

One important place to look at is lines 114-117, which is used for the Sequence functions. Calling for the sequence seq() would return the Things_MT. The notEmpty function would call false, as Things_MT are empty. It will also return a -1 when here is called, as Things_MT don't

have anything inside, and so it has no here. It also returns itself for next since it has nothing after itself.

The Things_MT class is very important for when creating the class of FiniteBags that have items inside of it, called Things_ST. The Things_ST class has a lefty, a here, a righty, an amount, and an isBlack boolean. For its cardinality (see lines 137-139), it is important to remember that with cardinality, it counts all elements in something, including duplicates, which is why, instead of adding 1 for each type of item, it adds the amount of that item. The member function is the first place where items in the search tree are compared, and use the compareTo function (see lines 141-151). It will return a number depicting how different the two compared things are. If the number is 0, then the item is the same, which also means that the item is definitely a member of that Things_ST that is being compared to. If compareTo gives a larger number, then the right side of the FiniteBags is checked for the item, else the left is checked. If the function ever gets to a Things_MT, then it will return false, as that means that the item was not in the FiniteBags. The function counter (see lines 153-166) use the member function to see if an item is in the within a FiniteBags, and returns the number of that item is within the FiniteBags.

Another good place to look at is the add functions for FiniteBags (see lines 176-190). Had the Red-Black Balanced Tree system been implemented, the add function would be different, but as it is not implemented yet, it will just put the items in the FiniteBags according to their value. It will put a specific number of an item into the FiniteBags. If no number of items is provided, it is assumed that only 1 item is to be put into a FiniteBag. The remove functions are also very different from the previous project, as a specific number of items is indicated to be removed (see lines 192-215). Of course, all of an item can be removed with the removeAll function (see lines 212-215). Most of the other functions are similar to the previous project. The

Sequence functions are at the bottom of the FiniteBags class (see lines 255-260). The seq() function, like the Things_MT returns the entire Things_ST. All of the other parts of the Sequence functions are different, though. It will call true for notEmpty, return it's here since it has one, and when next is called, it will return a new class if things, which includes the sequence of the left side of the Things_ST and the right side. This new class if called a Things_Cat, which only has two Sequences, left and right. It puts the left most things in the list of things before the right, so that the Sequence is at least linear (see lines 264-295).

For testing, random numbers and strings were necessary for testing. For this, the Randomness interface the randString class and the randInt class were created (see lines 297-330). What is most important is the testers that check that the rules made by the previous functions hold. The FiniteBags randomTree was created so that it could be used in tester functions so they are unbiased (see lines 345-355). First, the rules of Things_MT need to be checked, as it is the backbone of Things_ST. The function mtChecker would throw an exception if the tree created within the function says it has something when it shouldn't, or vice-versa. This obviously shouldn't happen, since that section didn't change much from the previous function.

The addRemCounterTests puts a random number of an item into a bag and takes it out (see lines 378-389). It then throws an error of the FiniteBag does not come out the same as before. This also shouldn't throw an exception, though, because even if the item was in there previously, the same number was taken put in and out.

Lots more tests could be created for this program, so that the counter can be checked more thoroughly, and so that subset is checked. Subset is a bit more complicated to test, since subset should also include have the same amount of an item as well. If Red-Black Balance Tree

was implemented, then there could have also been functions that check if they really are balanced.