Kathryn Hodge

203 Software Development

November 7, 2014

Polymorphic Finite Bags

The Polymorphic Finite Bags data structure can be purely implemented as a generic self-balancing binary search tree because generic BST implementation meets the mathematical specifications of a finite bag, and allows the finite set bags to hold any type of comparable data. In addition, these bags can be transformed into a sequence that can be iterated through. A finite bag is a multi-set, meaning each element may many occur times within the set, but because the bags are polymorphic, the bag should hold any kind of comparable data.

I. Binary Search Trees Can Represent Finite Set Bags

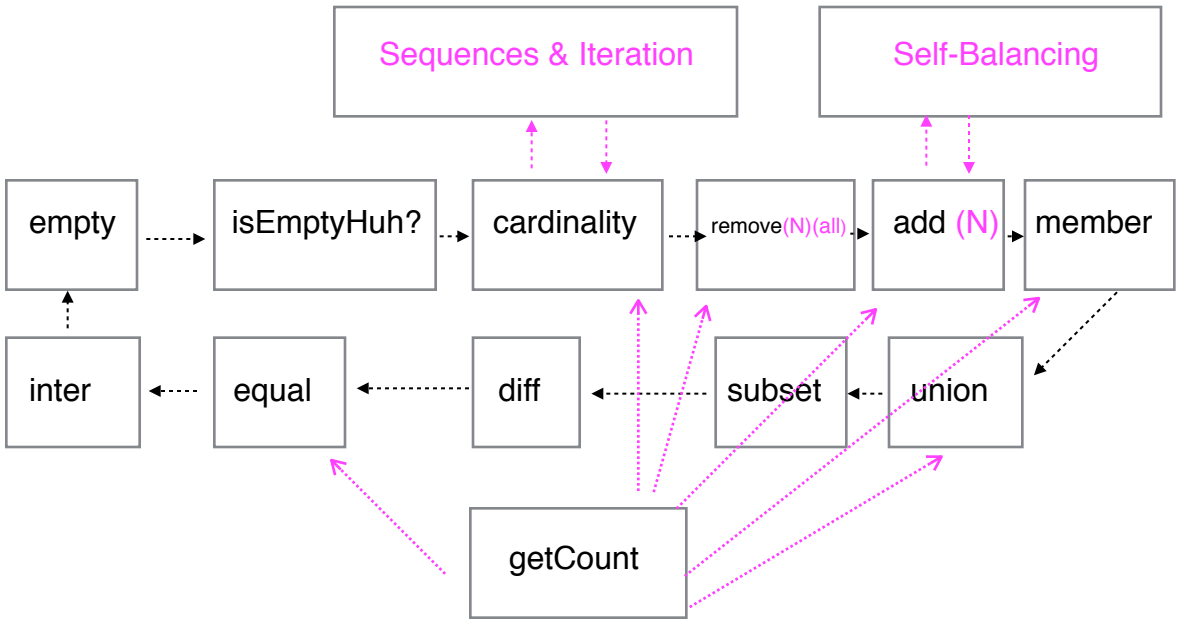
Like a finite set, a finite bag can be created with nothing in it or with many elements in it. It can also stay empty or integers can be added or removed from the bag. Some questions we can ask about bags are:

1. “Is an integer a member of the finite bag?”
2. “Is the bag empty?”
3. “Are two bags equal?”
4. “Is one bag a subset of another bag?”

In addition, we can join two bags together, combining their contents, or create a new bag with the common elements of two other bags. Through difference, we can also find what’s in one bag and not the other. In addition, through sequences, we can print out what’s in the bag or apply a function to every item in the bag. For example, if the bag is full of integers, sequences and iteration can be used to add 5 to each integer in the set. In a self-balancing binary search tree, all of these methods can be implemented efficiently, with log(N) time instead of linear time, and still meet all of the specifications above. The functions implementing the operations of finite set bags either return an int, boolean, or a new finite bag that does not modify the old bag.

In my program, I refer to a non-empty finite bag as a *SetBag_NonEmpty,* an empty finite set as a *SetBag_Empty,* and a finite bag (that could be empty or non-empty) as a *Bag*. *SetBag_NonEmpty* and *SetBag_Empty* are both classes that implement the *Bag* interface because they are both finite set bags that must implement the methods, defined by the operations of finite bags, specified within the *Bag* interface.

To prove self-balancing BST's implement finite bags, I created a connected graph of property testers that all depend on the co-relation of the functions. In other words, by testing all of the functions in relation to each other through properties, it can be easy to tell where one function fails because all the tests connected to that function will also fail. For more details on how each function works, see the API. The text in pink are new functions added for multi-set bags that were not in original finite-set assignment. However, the original functions were also to change to be compatible with the new count variable (for duplicate items within the set).



NOTE: The different add and remove functions all call addN, so in testing, if I prove one of the add/remove functions works, then all of the work.

Above, there is a diagram showing how all of my functions will be circularly connected (and also intertwined) through tester properties. When called individually, they won't be connected, but in the testing suite, they all work coherently through a given

mathematical specification or property for finite sets. Testing functions in relation to each other rather than independently proves the precision of the system because it demonstrates the overall functionality of finite sets. In addition, I use wrappers for all my property testers that randomly generate large numbers of test cases and search for witnesses of their negation. When the program is run, all of the tester will run 1000 times first for integers, then strings, and finally booleans. Testing on three completely different types of comparable data proves the finite bags are polymorphic and can hold any kind of comparable data (and not just a specific data type). Thus, by appealing to the properties of finite bags, I can show the accuracy of these operations and the implementation of a finite bag as a binary search tree.

In each case **t** is a *Bag*, **r** is a *Bag*, **x** is of type *D*, **y** is an *D*, **c** is an *int*, and **nT** is a new *Bag* that I'm creating. The types of x and y are unknown until the testing suite is run because the binary trees are generic. The *TesterClass.java* holds all the tests, but to test these functions, an instance of the TesterClass must be created with a generator. The generator (*gen*) will create random data of a certain type and determine what will actually be in the finite bags for that set of tests. In the code, *IntGen* and *StringGen* classes create random integers and strings that will be put into randomly generated bags (*rndBag)* and eventually be tested with the *TesterClass* functions. However, for booleans, a new class (*Hoolean*) had to be created because booleans are not comparable, meaning they do not have a compareTo function. In this program, true is arbitrarily greater than false and the BooleanGenerator creates random booleans that will be put into randomly generated bags (*rndBag)* and eventually be tested with the *TesterClass* functions. By testing on many types of data and with every value randomly generated, all cases of every property in the testing suite (and essentially for finite bags) is run. Below are the following circular properties that I will use to prove my BST implementation meets the specifications of a finite bag. Note any in color are properties added or changed in addition to finite set because the finite bag can hold duplicates.

A. (isEmptyHuh **t**) = true <-> **t** = empty()
B. (cardinality **t**) = 0 <-> (isEmptyHuh **t**) = true
C. (sumIt **t**) == (cardinality **t**)
D. [**c** = cardinality (remove **t** **x**)] <->

{ [**c** = (cardinality **t**) - 1 && (getCount **t x**) >= 1]
V [**c** = (cardinality **t**) && (getCount **t x**) = 0]
}

E. (member **t x**) = false && **nT** = remove (add **t x**) x <-> (equals **t nT**)
nT = (add **t x**) <-> (getCount **nT x**) - 1 == (getCount **t x**)
nT = (remove **nT x**) <-> (getCount **nT x**) == (getCount **t x**)

F. member (add **t x**) **y** = true <-> **x** = **y** ∨ (member **t y**) = true

G. member (union **t r**) **x** = true <-> (member **t x**) = true ∨ (member **r x**) = true
(member (union **t r**) **x**) && (member **t x**) <-> (getCount **t x**) > 0
(member (union **t r**) **x**) && (member **r x**) <-> (getCount **r x**) > 0
!(member (union **t r**) **x**) || !(member **t x**) || !(member **r x**) <->
(getCount **t x**) == 0 &&
(getCount (union **t r**) **x**) == 0 &&
(get Count **r x**) == 0

H. (getCount (union t r) x) == (getCount t x) +(getCount r x)

I. (equal (diff t r) t) && (equal (diff r t) r) <->
[!(isEmptyHuh r) && !(isEmptyHuh t) —>
subset r t) == (subset t r) == false]

J. (equal (inter **t r**) empty()) <-> (equal (diff **r t**) **t**)

K. (equal t r) <-> (cardinality (union t r) == (cardinality (inter t r)) * 2

L. **nT** = (inter **t** empty()) <-> (equal empty() **nT**)

Now, I will show how each of these specifications (properties) are met in my code through calls to my testers with randomly generated input of the correct type. To refer to lines of code, I will put [lineOfCode] after the statement. All of these lines of code, unless otherwise specified, will refer to my *TesterClass.java* file (i.e., my Testers class). Also, for the sake of space and time, I'm only going to talk about the ones the properties that changed since my previous game1 assignment discussed the un-changed properties

— —

C. (sumIt t) == (cardinality t)

CheckTree_seq_cardinality input a random Bag **t** with any type of comparable data [74]. Then, the tests checks if it the number of iterations through the sequence equals the number of elements in the bag [75]. In essence, this property checks if the

iteration abstraction goes through every element within the sequence, which represents the binary search trees, which in turn, represents a finite bag. If the property (i.e., the equality) is untrue, then an exception is thrown to stop the false program from running. After calling *checkTree_seq_cardinality* 1000 times with randomly generated inputs with several different types, all cases are shown and all succeed.

D. [c = cardinality (remove t x)] <->

{ [c=(cardinality t) - 1 && (getCount t x) >=1]

V [c = (cardinality t) && (getCount t x)=0] }

In *checkTree_cardinality_remove_getCount,* my parameters are a random *Bag* **t** (empty or non-empty) with randomly generated data and a random *D* **x** [82]. When we remove an item from a bag [83], either the item was removed, in which case the size of the bag would decrease by 1, or the item was not in the bag in the first place and the size would stay the same [86, 90]. In addition to affecting the size, the remove function should also affect count of that element in the set. In other words, if the item was actually removed from the bag, then the count of the bag before the item was removed should be greater than or equal to 1 [86]. If the item was not removed, then the element shouldn't have been there in the first place and the count of the item in **t** should be 0 [90]. However, because the testing suite compares the before and after of remove, the bags should not mutate. As before, after calling *checkTree_cardinality_remove* 1000 times with randomly generated inputs, all cases are shown and all succeed.

E. (member t x) = false && nT = remove (add t x) x <-> (equals t nT)

nT = (add t x) <-> (getCount nT x) - 1 == (getCount t x)

nT = (remove nT x) <-> (getCount nT x) == (getCount t x)

This test is very similar to the finite set test, but now, because a count variable is added, it can test even more things. In *checkTree_remove_equal_add,* the function inputs a random *Bag* **t** (empty or non-empty) [98]. First, the test adds an element (**rand**) and copies it into a *Bag* variable called **nT** [101-102]. Then, the test checks if the count of that variable in the tree was incremented via getCount [103]. Then, it removes that the element from the *Bag* **nT** [107]. Again, the test checks if the count of that variable in the tree was decremented via getCount [108]. Finally, the test checks if the original tree and equal to the new tree to see if the tree stayed the same after adding and removing

the same item [113]. If this is true, then the implemented functions work! Otherwise, either remove or add does not work because the new tree does not equal the old tree, which fails the property stated above. After calling *checkTree_remove_equal_add* 1000 times with randomly generated bags with several types of data, all cases are shown and all succeed.

G. (member (union t r) x) && (member t x) <-> (getCount t x) > 0

```
(member (union t r) x) && (member r x) <-> (getCount r x) > 0
!(member (union t r) x) || !(member t x) || !(member r x) <->
                                        (getCount t x) == 0 &&
                                        (getCount (union t r) x) == 0 &&
                                        (get Count r x) == 0
```

In *checkTree_member_union_getCount,* the test takes two random *Bags* **t** and **r** (empty or non-empty with random data) and a random datum *D* **x** [139]. First, the test creates a bool that keeps track of whether **x** is a member of the union of the two trees (two sets) **t** and **r** [140]. **X** can only be a member of the union iff **x** is a member of tree **t** or if **x** is a member of tree **r.** Thus, in testing, first we check if **x** is a member of the union and tree **t** [141], if so, the test is successful [107] because it proves the specification above. However, because the a count variable is added, we also must check if the count is greater than 0 [143]. Next, we check if **x** is a member of the union and tree **r** [148]. If this is true, the argument is still valid because it makes the second part of the 'or' statement true —> making the entire argument true and equating it with the left side [149]. However, as before, we must check that the count variable within **r** for x is greater than zero because for something to be a member of the set, the count for that variable within the tree must be greater than 0 [151]. For the last case, we test that the **x** is not a member of the union iff **x** is not a member of the *Tree* **t** and *Tree* **r** [156]. In addition, we also test that the count of **x** in **t**, **r**, and the union of **t** and **r** is zero because it is not a member of any of those bags[61]If this is true, the value meets the specifications of the property above and the functions are successful. If none of the three cases above are true, then there is a problem is with either the member or union function [162] and we throw an exception. After calling *checkTree_member_union_getCount* 1000 times with

randomly generated inputs and several types of data, all cases are shown and all succeed.

H. (getCount (union t r) x) == (getCount t x) +(getCount r x)

In In *checkTree_getCount_union,* the test takes two random *Bags* **t** and **r** (empty or non-empty with random data) and a random datum *D* **x** [178]. The test checks that the count of **x** in the union of the two trees equals the count of **x** in the **t** tree plus the count of **x** in the **r** tree. If this equality, which represents the property, is untrue, then an exception is thrown and the program crashes. After calling *checkTree_getCount_union* 1000 times with randomly generated inputs and several types of data, all cases are shown and all succeed.

L. (equal (diff t r) t) && (equal (diff r t) r) <->
!(isEmptyHuh r) && !(isEmptyHuh t) —>
(subset r t) == (subset t r) == false

In *CheckTree_subset_diff*, the inputs are two random *Bags* **r** and **t** [195]. First, the function diff's **t** and **r** and diff's **r** and **t**, creating two local variables [196-197]. If the two differences equal each other, meaning r - t = r and t - r = t, then the two bags must be completely disjoint and not be subsets of one another [200, 202]. However, because the empty bag is a subset of every bag, this property does not hold if either bag is empty. If the property is not held, then an exception is thrown and the program crashes. After calling *checkTree_subset_diff* 1000 times with randomly generated inputs and several types of data, all cases are shown and all succeed.

K. (equal t r) <-> (cardinality (union t r) == (cardinality (inter t r)) * 2

In *checkTree_equal_union_inter*, the inputs are two random *Bags* **r** and **t** [235]. This property changes quite a bit from finite sets (where the union and inter should be equal in data and size iff two sets are equal). In multi-sets (finite bags), if two bags have the same elements, then the intersection of the two bags should be twice as big as the union of the bags because the union should put everything from the two bags into one bag without taking out duplicates. Thus, the function checks this property [236] and the negation of this property [238] and if the property does not hold, the test throws an error and the program crashes. After calling *checkTree_equal_union_inter* 1000 times with

randomly generated inputs and several types of data, all cases are shown and all succeed.

————————————————————————————————

To make running all of these tests simpler, near the bottom of *TesterClass.java*, there is a function called runAll, which runs all of the tests a certain number of times according to the tests field.

Thus, because my testers interconnect all of my functions, none of them can be hard coded and they are forced to all fail or all succeed. In other words, the code must work with any given valid input — or else they wouldn't work in conjunction with each other. Moreover, the binary search tree implementation meets every specification of finite set bags because the functions within the interconnected testers keep their original behavior, adjust to changes within the instance of the data structure without mutating the original data, and do not mess with the behavior of other methods.

II. The Binary Search Trees Are Self-Balancing

In my code, I use self-balancing red-black trees so that the code can run more efficiently. To do this, I call addInner [312] as a helper to add in *SetBag_NonEmpty.java,* which finds the place to add the element, and then I call balance() [209] to re-balance the tree with the new element. To prove the trees are self-balancing, the programmer could create a test that could check periodically if the heights of every branch within the tree were within one of each other. Another test that could be run is that no two black nodes should be next to each other at anytime after the balancing is run. However, for the sake of this assignment, the binary trees have an implemented self-balancing system that works with the properties of finite set bags so we can assume that the tree is self-balancing. The reason for making these trees self-balanced is to make the trees run log(N) time versus linear time when the member function is called. This makes the code more efficient and faster, which is the goal of every programmer.

III. The Finite Set Bags Are Polymorphic

The bags have been proven to be balanced and meet the specifications of finite set bags, but now they must also be polymorphic, meaning they have to accept any kind

of data. For my tests, I test the bags on integers, strings, and booleans and the properties still hold, so the bags must hold any kind of data and cannot be fixed towards a certain type. Furthermore, the finite set bags are polymorphic!

IV. The Binary Search Trees as Sequences

The Binary Search Trees also have a built-in Iterable (*Sequence*) interface that allows the programmer to present the finite set bag in a sequence form to the client. Then, the client can say, for example, “sum up everything in the sequence”, and the programmer can iterate through the binary search tree via the sequence and apply this function. My sequences are proven to work because of the *CheckTree_seq_cardinality* tester function, which is discussed in part I. In addition to connecting sequencing to the property testing, I also have a stringIt function [28] in the *Bag* interface and when implemented, the function takes a Bag, turn it into a sequence, and turns all of the elements into a long catenated string. When the function is run, this string can then be printed to the client and the client can see what’s in the bag. This function doesn’t necessarily help prove that the sequences work, but it can help debugging because it is easy to see exactly what is in the bag and what has gone wrong. At the end of the runAll function in my *TesterClass.java* [417], I print out several random trees as sequences to prove that my randomly generated tests are in fact randomly generated and output the bags expected.

Hope you enjoyed the color coding