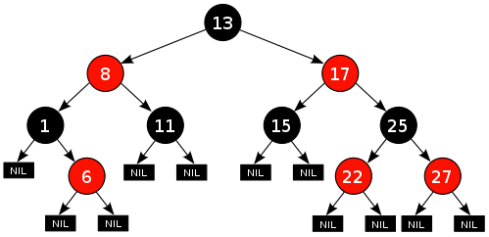
Laura Barreto
CMPU 203

Data2: The BEST Polymorphic Finite Bags Library

A polymorphic finite bag is a fancy way of saying multiset. A multiset allows for a set to contain many different instances of the same element. In order to create an efficient multiset, the set has to be a self-balancing binary tree. There are many ways of creating a self-balancing binary tree. Some methods include using Two-Brother, AVL, or Red-Black tree algorithms. Out of these mentions, Red-Black is the easiest algorithm to implement due to the fact that the balancing only occurs in one method called balance() rather than having to create Fake trees (in Two-Brothers).

I. Red-Black Trees



The balance( ) method is called by add( ), and remove( ) in order to balance the trees after adding or removing an element. Because all methods implement call each other, add( ) is in different methods, thus balance( ) plays a role in other methods such as union( ). Because all the testers work even with balance, I am assuming that the Red-Black implementation correctly balances the tree. Because I did not write software to draw the representation of unbalanced and balanced Red-Black trees, I am not able to observe whether the implementation of balance( ) works in accordance to the properties of Red-Black trees. In order to be able to test and know that my Red-Black tree was implemented correctly, I would have to test the properties of a red-black tree: [1]

[1] http://en.wikipedia.org/wiki/Red%E2%80%93black_tree
[2] http://www.geeksforgeeks.org/check-given-binary-tree-follows-height-property-red-black-tree/

1. A node is either red or black.
2. The root is black.
3. All empty nodes are black.
4. Every red node must have two black children.
5. Every path from a given node to any of its descendant leaves contain the same number of black nodes.
6. The maximum height of a node is at most twice the minimum height.[2]

Methods specific to the Red-Black trees are balance( ), blacken( ), and isBlackHuh( ).

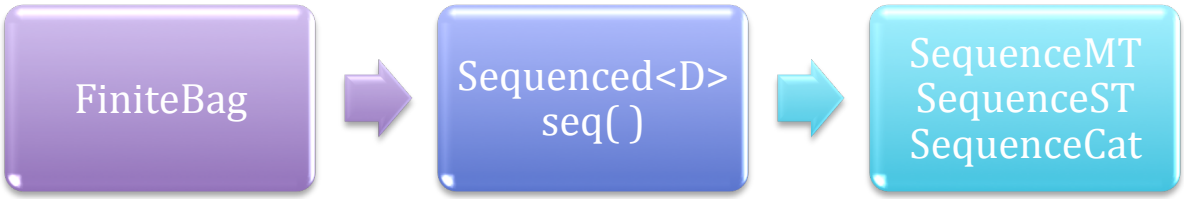
In data1, all methods having to search through, add, or remove elements would work in linear time. By implementing the Red-Black algorithm, the speed of these methods were reduced to O(log n) rather than O(n).

II. Iteration Abstraction

The iteration abstraction interface I used was Sequences. **Sequenced** creates an instance of a sequence out of the set. **Sequence** is the interface that allows each sequenced instance to be iterated through. One way to make sure that Sequence was working effectively was to make sure the size of a sequence created from a set should be the same as the cardinality of the set. This test occurs in the SumIt tester. Another feature of Iteration Abstraction that was tested was the SeqToString method to make sure that the result of the SeqToString successfully returns the String version of the Sequence. This is a method that is able to be

[2] http://www.geeksforgeeks.org/check-given-binary-tree-follows-height-property-red-black-tree/

applied to every element in the sequence. Each element in the Sequence is converted to a corresponding String type. Iteration abstraction allows us to find out what exactly is in a multiset as well as apply methods to each element in a multiset. SequenceMT creates a sequence out of the MT FiniteBag. In this case, the element here is null, hasNext is false, next returns this sequences (which is null), and seqToString, returns a String with nothing in it. SequenceST is the sequence for stuff it creates a sequence out of the element here in the finiteSet. Finally, SequenceCat works as a way to append sequences (concatenates sequences) in order to be able to return a sequence with all elements of the FiniteBag. The diagram below shows the process of how to make a FiniteBag be a sequence in order to be able to iterate through the Self-Balancing Tree.



III. Multisets

The multisets we had to create have to be able to be accurate regardless of the object type in the set. Because of this, it was important to have the trees use generics. In tests, both integers and strings were tested. The **FiniteBag** interface was extend by SBBT_ST (Self-Balancing Binary Tree with Stuff) and by SBBT_MT (Self-Balancing Binary Tree that is Empty). In the finite set data1 homework, the methods used to create the trees were:

Empty, Cardinality, isEmpty, member, add, remove, union, intersection, difference, equal, and subset.

In this **FiniteBag** polymorphic self-balancing set implementation, the methods used were:

Empty, Cardinality, isEmptyHuh, member, add, **add** (element, n), remove, **remove**(element, n), **removeAl**l, union, intersection, difference, equal, subset, and **getCount**(element). The bolded methods are introduced in multisets since in order for the set to be a multiset, it must be possible to add n instances of an element, remove n instances of an element, remove all instances of an element, and get the count of a certain element in the set. All preexisting methods were implemented differently in order to accommodate the slightly different features of a multiset. In order to effectively test the efficiency of my implementation of self-balancing binary search trees, I created an interface called **Randomness<D>** which takes a generic object type and uses createRand( ) to create either a random integer or a random string. Afterwards, I created a method called randTree( ) [TestTest 59-65] which creates a random tree with either random integers or random Strings. Within each test, a combination of random tree lengths, trees, and elements were created in order to create random cases to be tested. Each test runs 10,000 times: 5,000 times on a randInt tree and 5,000 times on a randString tree. All the testers throw exceptions so if a single tester fails, the code would break. Out of the 210,000 tests run, all 210,000 are successful for both integers and strings. For descriptions of the different properties and the definitions of union, intersection, difference, empty,

cardinality, and member, refer to the data1 essay that thoroughly outlines all the definitions. The properties tested and proved for multisets are:

1. Empty [TestTest 68]

   Empty test checks to make sure an empty finite set is empty and a non-empty finite set is non-empty.

2. Empty and Union [88]

   This test checks to make sure that the union of a random set and the empty set returns the random set. This satisfies the identity property of sets.

3. Empty and Intersection [99]

   This test checks to make sure that the intersection of a random set and the empty set returns the empty set. By definition of intersection, this test proves to be successful.

4. Empty and Difference [110]

   This test checks that the difference between a random set and the empty set should be the empty set because there will be no elements in the random set that are not in the empty set since there are no elements in the empty set.

5. Empty and Cardinality [126]

   If a set is empty, the cardinality should be zero.

6. Cardinality and Add [141]

   When an element is added, the cardinality should increase. When adding n instances of an element, the cardinality should increase by n.

7. Cardinality and Remove [156]

When removing an element from a multiset, the cardinality should either decrease if there are instances of the element found in the set, or stay the same in the case where there are no instances of an element to remove.

8. Member and Difference [173]

   If an element is a member of the difference of random sets A and B, then the element should not be an element of A but be an element of B.

9. Member and Union [193]

   If an element is in the union of random sets A and B, then the element should be included in either A or B or both.

10. Member and Intersection [213]

    If an element is in the intersection of random sets A and B, then the element should be included in both A and B.

11. Member and Add [230]

    After an element is added to set, the element should be a member of the set.

12. Union and Subsets [251]

    Random sets A and B should both be subsets of the union of A and B. The union of Sets A and B contains all elements in A and all elements in B, thus it is guaranteed that A and B are both subsets.

13. Intersection and Subsets [267]

    The intersection of random sets A and B should be a subset of sets A and B. The intersection returns a set that contains elements that are common to both A and B, thus the resulting set is a subset of A and a subset of B.

14. Difference and Subsets [280]

The difference set of random sets A and B, A-B should be a subset of set B since the set would contain the elements in B that are not in A. So since the elements are in B, it can be stated that the resulting set is a subset of B.

15. Equal and Intersection [296]

    For any random sets A and B, if A is a subset of the intersection of A and B, and B is a subset of the intersection of A and B, then sets A and B are equal.

16. Add and GetCount [310]

    After adding n number of elements to a set, getCount should return an int that is n larger than before adding the n elements.

17. Remove and GetCount [326]

    After removing n instances of an element from a set, getCount should either decrease or stay the same. In the case where the element is in the set n instances, the count should decrease, as there are elements present in the set. In the case where there are not n instances of an element, the count stays the same because no element would be removed.

18. Add, Remove, and GetCount [345]

    After adding an element, and then removing it, getCount should be the same before and after adding/removing. This indicates that the bag before is equal to the bag where a random element was added and them removed.

19. RemoveAll [367]

    Instances of an element after removing all instances should be zero. In a set where there are n elements, the count for that specific element would be n. Thus n – n would return a count of 0.

20. SumIt and Cardinality [381]

    Because sequence takes all the elements of a tree and creates a sequenced version, a sequence should have the same amount of elements as the set. SumIt checks the size of the sequence, and cardinality checks the size of the set. These amounts should be equal.

21. SeqToString [392]

    Checking to make sure that the output of making the sequence a string actually returns an instance of a String.

All methods implemented served as helper methods for each other and each method is connected with another. This allows many different combinations of methods in order to prove the rules of multisets. Because all tests and methods created satisfy and follow the properties of multisets, the implementation of a polymorphic finite bag is accurate, efficient, and usable. To efficiently test, mutation was not used when testing random Trees. In addition to that, random elements were generated. Testing used generics just like the rest of my code used generics, thus it proves that for any random combination of random elements and random trees, all tests succeed thus all properties are satisfied no matter the object type. For more information about the methods, refer to the API created for the interfaces used. Since the implementation of balance did not throw off methods such as add, union, intersection, remove, it has been shown that the self-balancing aspect of the FiniteBag was correctly implemented.