Thomas Lum
11/7/2014
McCarthy
CMPU 203

Implementation of Iterator and Self-Sorting Polymorphic Finite Bags

As a relative of the binary tree implementation of finite sets, the RB Tree implementation of Self-Sorting Polymorphic Finite Bags borrows many properties and methods from finite sets. The equal, subset, diff, inter, and union methods remain almost entirely unchanged, changed only if to accommodate the new count variable, which keeps track of how many of a particular element there are in the PFB. Add and remove are also altered to reflect the addition of count, accommodating the ability to increase the count if an element is already in the PFB for add. Remove now permits the user to remove a specific amount from that element's count, or, by calling removeAll, to reduce the count to 0, thereby removing it from the tree. This rather different definition of removal from the previous Set implementation is reflected in the new noFilledBags and eltCount methods. While isEmptyHuh will return whether or not the node or PFB is a PFBBranch or if it is a PFBLeaf, and cardinality will return the number of nodes that exist in the tree, noFilledBags will only return true if the current node has a count of 0, as well as its children (or if it is a leaf), and eltCount will only count bags within PFB with count greater than 0. Thus, if a PFB has nodes of only count 0, it is essentially 'empty' in one sense of the phrase, yet it still may have children that must be considered when adding or balancing, and so it is not true for isEmptyHuh.

This difference is tested in the testers method testcountmethods (1023). In the test, a random element is added to an empty PFB. The member and new countOf

function are tested with the number that was added, which should provide true and the count of the element respectively. The random element is then removed using the removeAll method with the element on the PFB. This final PFB should return true when calling noFilledBags and false when calling emptyHuh, else an error message is printed. Lastly the cardinality of the final PFB is checked to not equal zero, as there are still empty bags (not to be confused with leafs), and the eltCount is checked to equal zero, as no bags in the PFB have counts above 0.

It should be noted that despite phrasing like removing and adding, all functions for PFB, like those for FinSet, do not mutate any values, but simply return new PFBs or values.

A rather major difference from the FinSet interface is the generic type acceptance of PFB. So long as type T is comparable, it can be used as an element of the PFB. Because of this, any method for tree navigation that would normally use less than or greater, such as remove or add, or that used == to compare elements, such as member, now use the compareTo and equals methods, which apply to all classes that sufficiently extend comparable, thus permitting type-appropriate sorting and comparison.

The final major difference between FinSet and PFB is the new boolean instance variable red, as well as the other functions that help create and check a proper implementation of red black tree sorting. The redHuh function simply allows methods to check on the red instance variable of a PFB. Similarly, left() and right() allow the user to see the left and right branches of a tree, however, as a leaf has no such elements, the methods also throw NothingHere exceptions, which will be caught in the later balance method (672).

In order to properly insert into the red black PFB, the add function essentially acts as a mask for the helper function ad which should only ever be called from add. The masking via the add functions allows the PFB to color the top of the tree black after all ad calls are done. The ad method functions similarly to the add method of FinSet, using compareTo rather than less than or greater than to find which path to continue inserting (as well as offering the option of adding to a pre-existing node's count) however, after each insert call, a balance method is also called on that PFB. This balance function properly rearranges any instance where a red node has a red child by converting it into a rebalanced red node with two black children. By catching NothingHere exceptions, the method is able to try probing the PFBBranch's children's children, or lack there of, by using the left and right methods described earlier. Several try and catch blocks find and balance the four different combinations of black nodes with red children and red grandchildren.

The two representation invariants of red black trees are tested in the testers method testRBInvars (917). PFBs have two methods that help assert these invariants. The first is rbInvar1, which returns true so long as no red node in a PFB has any red children (551 & 814), the first rep invariant of red black trees. This is tested repeatedly on randomly generated PFBs (of both String and Integer types) in testRBInvars, and prints an error if this is ever not true. The second helper method ranPathBCount (558 & 818) counts the number of black nodes it encounters along a random path down PFB. testRBInvars calls this function multiple times on a randomly generated PFB and will print an error if any ranPathBCount is different from any other of the same PFB.

The Iterator class, an implementation of sequences or iterators, permits any class that extends it to be navigated iterably. Both FinSet and PFB extend Iterator, providing the appropriate here, anythingHere and next methods required of the extension, the first and last of which throw NothingHere exceptions, on the off chance that it is called on a PFBLeaf or Leaf. In order to properly and efficiently iterate through the PFB and FinSet, as both are realized through trees, the Trunk method acts to more evenly distribute the cost of each next call, making it pay as you go, by looking at the leftmost item whenever possible before looking at the right PFB at all.

The iterability of both FinSet and PFB are tested in the Data2 methods. The first, hasOdds, iterates through an Iterator and returns true if there is any odd element is found, which makes the most sense with Iterators that contain numbers. As such, testHasOdds (951) creates PFBs and FinSets with only either even or odd int elements before running the hasOdds method on them, and an error is printed if the test fails. The longest and testLongest(972) methods serve to iterate through an Iterator and find the value that, when converted to a string, is the longest (so “101” is longer than “10” but no longer than “999”). String PFBs, Integer PFBs, and FinSets were all tested on this method by creating random PFBs/FinSets before inserting a final String or number longer than any other element in the Iterator. This value was compared to the result of calling longest and printed an error if the two were not equal. Lastly, iterLength and testIterLength(1000) serve to tell and compare that length of the iterator to the call to cardinality on the PFB/FinSet, respectively. Again, the test was made over all three types of bags/sets, and errors were indicated by printing.