Kathryn Hodge

Professor McCarthy

CMPU 203: Software Development

September 19, 2014

Data 1: Finite Sets as BST's

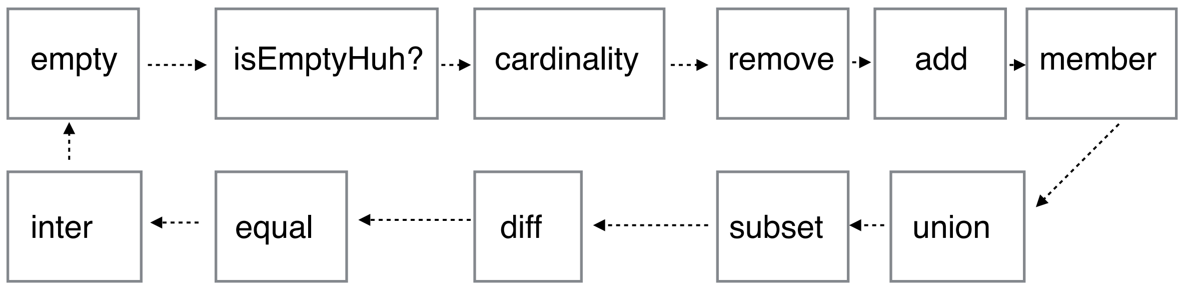
The finite integer set data structure can be represented as a binary search tree because BST implementation meets the mathematical specifications of a finite set and its operations are closed under the given requirements. A finite integer set is a finite set of only integers, without duplicates. A finite set 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 finite set. Some questions we can ask about sets are:

1. "Is an integer a member of the finite set?"
2. "Is the set empty?"
3. "Are the sets equal?"
4. "Is one finite set a subset of another finite set?"

In addition, we can join two sets together, create a new set with the common elements of two other sets, and find what's in one set and not the other through difference. In a binary search tree, all of these methods can be implemented, meeting all of the specifications above. These functions implemented in my code either return an int, boolean, or a new finite set so that does not modify the old set.

In my program, I refer to a non-empty finite set as a *FiniteSet,* an empty finite set as a *FiniteSet_Empty,* and a finite set (that could be empty or non-empty) as a *Tree*. *FiniteSet* and *FiniteSet_Empty* are both classes that implement the *Tree* interface because they are both finite integer sets that must implement the methods specified by the definition of a finite set.

To prove BST's implement finite sets, I created a connected graph of property testers that all depend on the co-relation of my 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.



Above, there is a diagram showing how all of my functions will be circularly connected (and also intertwined - not shown in graph) through tester properties. When you call them, they won't be connected, but in my testers, they all work coherently within 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. Thus, by appealing to the properties of finite sets, which combine different functions, I can show the accuracy of these operations and the implementation of a finite set as a binary search tree.

In each case **t** is a *Tree*, **r** is a *Tree*, **x** is an *int*, y is an *int*, and **nT** is a new *Tree* that I'm creating. In the program, the inputs to these tester functions come from a random number generator (*rndInt)* and a random tree generator *(rndTree)*, so the functions essentially test all cases of a certain property. Below are the following circular properties that I will use to prove my BST implementation meets the specifications of a finite integer set:

A. (isEmptyHuh **t**) = true <-> **t** = empty()
B. (cardinality **t**) = 0 <-> (isEmptyHuh **t**) = true
C. **nT** = cardinality (remove **t x**) <-> **nT** = (cardinality **t**) - 1 V **nT** = (cardinality **t**)
D. (member **t x**) = false ^ **nT** = remove (add **t x**) x <-> (equals **t nT**)
E. member (add **t x**) **y** = true <-> **x** = **y** V (member **t y**) = true
F. member (union **t r**) **x** = true <-> (member **t x**) = true V (member **r x**) = true
G. **nT** = (union **t r**) <-> (subset **t nT**) = true & (subset **r nT**) = true
H. **nT** = (diff **t r**) <-> (isEmptyHuh? **t**) = true V (subset **t nT**) = false
I. (equal (inter **t r**) empty()) <-> (equal (diff **r t**) **t**)
J. (equal **t r**) <-> (equal (union **t r**) (inter **t r**)
K. **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 Testers.java file (i.e., my Testers class).

———————————————————————————————

A. (isEmptyHuh t) = true <-> t = empty()

—> empty() & isEmptyHuh

In *checkTree_empty_isEmptyHuh,* my parameter is a random *int* **count** that is either 0 or 1 [22]. This allows us to get each case about an equal number of times -> rather than just one case every time. If **count** equals 0 [24], then the function creates a new empty tree (*FiniteSet_Empty)* by calling empty() [25] and checks if the tree is actually empty by calling isEmptyHuh [26]. If the tree is in fact empty, then our two functions work because isEmptyHuh is true iff the tree is empty [27]. If not, then our functions do not work because an empty set is said to have elements, which is not true according to the specification [29]. If **count** isn't equal to 0, then I create a random *FiniteSet* that is not empty [33]. If we negate both sides of the property, then isEmptyHuh is false iff the tree is not empty. To prove our functions work, we call isEmptyHuh on this randomly generated *FiniteSet* [34]. If isEmptyHuh returns false, then a nonempty finite set is not empty, the property holds, and it's success [35]. Otherwise, the nonempty finite set is empty and the implementation fails, meaning either isEmptyHuh or empty() does not work [37]. After calling *checkTree_empty_isEmptyHuh* 50 times with randomly generated inputs, all cases are shown and all succeed.

B. (cardinality t) = 0 <-> (isEmptyHuh t) = true

—> isEmptyHuh & cardinality

In *checkTree_isEmptyHuh_cardinality,* I input a random *Tree* **t** that is either empty or nonempty[42]. If the tree is empty and it's size equals 0 [43], then our functions work and it's a success, according to the specification above [45-45]. We can also negate both sides of the iff statement, saying a tree is not empty iff it's size is not equal to zero [47-50]. If neither of the above statements are true, then either the implemented cardinality or isEmptyHuh function has failed to meet the specifications [51-53]. After calling *checkTree_isEmptyHuh_cardinality* 50 times with randomly generated inputs, all cases are shown and all succeed.

C. nT = cardinality (remove t x) <-> nT = (cardinality t)-1 V nT =(cardinality t) —> cardinality & remove

In *checkTree_cardinality_remove,* my parameters are a random *Tree* **t** (empty or non-empty) and a random *int* **x** [58]. When we remove an item from a set, either the item was removed, in which case the size of the set would decrease by 1, or the item was not in the set in the first place and the size would stay the same. First, the test removes an item **x** from the set (*Tree*) **t** [59]. Then, it tests if the new tree's size is one less than before; in other words, if something was removed from the tree, does it's size match up [62]. If so, the test is successful before an item was removed and the set's size was adjusted accordingly. In the other case, an item was not removed because the item was never there so the tree's size stays the same [64]. If this happens to be true for an arbitrary input, then the functions succeed [65-66] because the second clause of the 'or' statement in the specification is true. If two statements above are not true for a given input, then the cardinality or remove function (or both) fails to meet the specifications as provided by the property [67]. As before, after calling *checkTree_cardinality_remove* 50 times with randomly generated inputs, all cases are shown and all succeed.

D. (member t x) = false ^ nT = remove (add t x) x <-> (equals t nT) —> remove & equal & add

In *checkTree_remove_equal_add,* I input a random *Tree* **t** (empty or non-empty) [72]. First, the test adds an element (**rand**) that is not in the original set (out of the range of the *rndInt* within *rndTree*) and copies it into a *Tree* variable called **nT** [74-75]. Then, it removes that said element from said *Tree* **nT** [76]. By testing if original tree equals the new tree **nT** [79], we can see if the tree stayed the same after adding and removing the same item that was not originally in the tree. If this is true, then the implemented functions work [80-81]! Otherwise, either remove or add does not work because the new tree does not equal the old tree, which fails the property stated above [83]. After calling *checkTree_remove_equal_add* 50 times with randomly generated inputs, all cases are shown and all succeed.

E. member (add t x) y = true <-> x = y V (member t y) = true

—> add & member

In *checkTree_add_member,* the parameters are a random *Tree* **t** (empty or non-empty) and two random numbers: *int* **x** and *int* ***y*** [88]. First, the test adds an input *int* (**x**) to the set (**t**) and asks whether a different *int* is a member of the newly created tree [89]. If this happens to be true, then either the two random numbers equal each other [90-91] or the added value was already a member of the initial tree **t** [92-94]. However, if **x** was not a member of the new tree, then **x** couldn't equal **y** and it couldn't be a member of the original tree [95-96], according to the equally valid negation of the specification above. If none of the three cases above are true, then either add or member has failed to follow the property [98]. After calling *checkTree_add_member* 50 times with randomly generated inputs, all cases are shown and all succeed.

F. member (union t r) x = true <-> (member t x) = true V (member r x) = true

—> member & union

In *checkTree_member_union,* the test takes input two random *Trees* **t** and **r** (empty or non-empty) and a random number *int* **x** [104]. First, the test creates a bool that keeps track of whether the number **x** is a member of the union of the two trees (two sets) **t** and **r** [105]. **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** [106], if so, the test is successful [107] because it proves the specification above. Next, we check if **x** is a member of the union and tree **r** [108]. 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 [109]. 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** [110]. 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 [113]. After calling *checkTree_member_union* 50 times with randomly generated inputs, all cases are shown and all succeed.

G. nT = (union t r) <-> (subset t nT) = true & (subset r nT) = true

—> union & subset

In *checkTree_union_subset*, the parameters are two random *Trees* **t** and **r** (empty or non-empty) [118]. First, the test unions the two given trees [119]. For the implementation to meet the specification, the tester must prove that both trees are a subset of their union. Thus, in the code, we ask if *Tree* **t** is a subset of the union [120] and if *Tree* **r** is a subset of the union as well [120]. Only if both of these are true, the tests succeed and the functions meet the properties' specifications [121-122]. Otherwise, there are problems with the program's implementation of the union or subset function [124]. After calling *checkTree_union_subset* 50 times with randomly generated inputs, all cases are shown and all succeed.y generated inputs, all cases are shown and all succeed.

H. nT = (diff t r) <-> (isEmptyHuh? t) = true V (subset t nT) = false

—> subset&isEmptyHuh&diff

In *checkTree_subset_diff,* the test's inputs are two random *Trees* **t** and **r** (empty or non-empty) [130]. First, the test takes the difference of the two given trees (**r** - **t**) and puts it in a variable called **nT**. According to the property, for the difference of the two trees, either *Tree* **t** is the empty set or **t** is not a subset of the difference **nT**, which makes sense because by definition of difference, all of the elements that were in **t** should be taken out of **r**. The tester tests this property by calling isEmptyHuh on *Tree* **t**, seeing if the tree is empty [134]. If this is true, then the *Tree* **t** is empty and the test is a success because the first part of the 'or' specification is met. In addition, this leaves the diff to be all of **r** [135-136]. Otherwise, the test checks if *Tree* **t** is a subset of the difference **nT** [137]. If it is a not a subset of the difference, then the test is a success because the tree is not a subset of the difference [138-139], which meets the specification above. If the two cases above are not true, then there is a problem with the program's implementation of the subset or difference function [142]. After calling *checkTree_subset_diff* 50 times with randomly generated inputs, all cases are shown and all succeed.y generated inputs, all cases are shown and all succeed.

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

—> diff & inter & empty & equal

In *checkTree_diff_inter_empty_equal* my parameters are two random *Trees* **t** and **r** (empty or non-empty) [148]. In the wrapper function in my FiniteSet.java file (where all my tester functions are actually called), I add in another variable called **randomNumber**

[301-302]. Approximately every fourth time the loop runs it course, another version of *checkTree_diff_inter_empty_equal* will be called where the two random trees are actually the same random tree [303]. This will allow for all cases within the tester to be hit several times, rather than just one case the entire time. This property states that if the intersection of two given trees is the empty set, then the difference of one tree (say **t**) and another tree (say **r**) still equals the original tree **t**. In the first case, the test asks if the intersection of the two trees is empty and the difference (**t** - **r**) equals **t** [150]. If this is true, then the specifications in the property are met and the test is successful [151]. Otherwise, the function tests if the intersection is not equal to the empty set and if the difference does not equal t [152]. If this is true, then the test is a success because the function negates both sides of the specification, still providing a true value for the property because of the rules of iff statements [153]. If the two above cases fail, then the functions fail to implement the property correctly [155]. After calling *checkTree_diff_inter_empty_equal* 50 times with randomly generated inputs, all cases are shown and all succeed.

J. (equal t r) <-> (equal (union t r) (inter t r)

—> equal & union & inter

In *checkTree_equal_union_inter,* my parameters are two random *Trees* **t** and **r** (empty or non-empty) [159]. In the wrapper function in my FiniteSet.java file (where all my tester functions are actually called), I add in another variable called **randomNumber** [319-320]. Approximately every fourth time the loop runs it course, another version of *checkTree_equal_union_inter* will be called where the two random trees are actually the same random tree [321]. This will allow for all cases within the tester to be hit several times, rather than just one case the entire time. This property says two sets are equal iff their union and intersection are the same. To test, first the function tests if the union of the two given trees is equal to the intersection of the trees and if the two trees are equal [161]. If both of these conditions are true, then the specifications within the property are met and the test is successful [162-163]. In the case where both sides of the property are negated, the function tests whether the intersection and union of the two trees are not equal and where hey do not equal each other [164]. If both of these are true, then the test is a success because the conditions of the property are met: false = false. If the input does not fall within one of the two cases above, then the program fails to implement equal, union, or diff (or a select combination of the three) [168]. After calling *checkTree_equal_union_inter* 50 times with randomly generated inputs, all cases are shown and all succeed.

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

—> inter & equal & isEmptyHuh? & empty()

In *checkTree_inter_empty* the test's input is a random *Tree* **t** [173]. First, the test creates a variable **bool** that stores whether a given tree's intersection with the empty set is in fact the empty set [174]. This should be true in all cases, whether the set is empty or nonempty [175-176]. In the first case, the test asks if in fact, the given tree's intersection with the empty set is the empty set, and if the set is nonempty [177]. If both conditions are true, then the test proves that the intersection of a nonempty set with the empty set is just the empty set [178-180]. The next case tests also if the given tree's intersection with the empty set is the empty set and if the set is empty() [181]. If both conditions are true, then the test proves that the intersection of an empty set with the empty set is also the empty set [182-183]. If none of the above cases are true, then the test fails and the implementation of the inter, equal, isEmptyHuh, and empty() (or a select combination) functions fail to meet the specifications of a finite integer set [185-186]. After calling *checkTree_inter_empty* 50 times with randomly generated inputs, all cases are shown and all succeed.

————————————————————————————

Thus, because my testers interconnect all of my functions, none of them can be hard coded. They all must really 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 each property because the functions within the interconnected testers keep their original behavior, adjust to changes within the instance of the data structure, and do not mess with the behavior of other methods.