CS2230 Computer Science II: Data Structures

Homework 7

Implementing Sets with binary search trees

Due Nov 13, 2017

30 points

Goals for this assignment

- Learn about the implementation of Sets using binary search trees, both unbalanced and balanced
- Implement methods for a NavigableSet, including contains and remove
- Get more practice writing JUnit tests

Purpose

Binary search trees can be used to build efficient Sets that perform lookups and inserts in $O(log\ n)$ time, which is fairly efficient. As we've seen in class, Sets are useful for applications where you need to look things up by a "key" like checking airline tickets or looking up the existence of a word in a dictionary. In this homework, you will study the implementation of Sets using binary search trees. We have provided Java files that contain code for a Set implemented with an unbalanced binary search tree (BSTSet.java) and a Set implemented using an AVLTree (AVLTreeSet.java), and your job is to finish them.

Submission Checklist

By **November 9, 9:30am**: answers to the PROGRESS_REPORT.txt in your GitHub repository.

By **November 13, 11:59 pm** You should have changes in GitHub to the following files:

- BSTSet.java
- AVLTreeSet.java
- BSTSetTest.java
- AVLTreeTest.java (if you added optional tests)

Slip days: Your submission time is the date of the last commit we see in your GitHub repository.

You must submit to ICON: the link to your GitHub repository.

You will submit them via GitHub. Follow the directions in **getting_hw5.pdf** on "Setup your own private repository to push your commits to". Before you are done submitting, you must check the following.

- Do the tests pass?
- Did I write the required tests?
- Does my GitHub repository reflect the code I intend to turn in? (You must view this in your web browser, not in NetBeans).

Getting HW7

Follow **all the same** instructions for getting_hw5.pdf **with the following changes**:

GitHub Url: https://github.uiowa.edu/cs2230-assignments/sets-with-search-trees.git

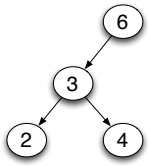
Project Name: SetsWithSearchTrees

adding collaborators: only add the username **talsulaiman.**

Part 0

Examine the TreeNode class. In particular, answer the following questions for yourself (you do not need to submit the answers, but this part will help you with the rest of the assignment).

- How do you test if *this* is a leaf?
- How does isBST work?
- What is the result of toString() when called on the root TreeNode of the following tree?



Part 1: Contains method

The Set.contains method returns true if and only if the element exists in the Set.

a) The BSTSetTest.java file has test cases for BSTSet. Devise three different tests for the contains method (put them in testContainsA,B, and C) so that you are confident that contains() works. Your tests for this part should be "black box", that is, they don't depend on the implementation: they only call public methods of BSTSet (in this case, the constructor, add(), and contains()). Your 3 tests need to be *different*: that is, your add methods should be such that they cause different underlying tree structures and there should be cases where contains() returns each true and false.
b) Implement the BSTSet.contains method.

Part 2: A method for NavigableSet

Take a look at the NavigableSet interface. It adds a new method keysInRange to the methods already provided by Set. The keysInRange method requires that keys stored in the NavigableSet have a total

order. Therefore, you'll see that the generic type is constrained to be a Comparable. The Comparable interface provides the compareTo method.

```
public interface NavigableSet<T extends Comparable<T>> extends Set<T>
```

Read the Java 8 API on Comparable
https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html to see what compareTo does.

a) BSTSet implements the NavigableSet interface, so you need to provide an implementation of keysInRange.

```
Iterator<T> keysInRange(T start, T end)
```

that returns an Iterator that produces all elements x in sorted order, such that $start \leq x < end$.

Your iterator must be efficient. In other words, the following code

```
Iterator<Integer> iter = t.keysInRange(start, end);
while (iter.hasNext()) System.out.println(iter.next());
```

should run in $O(s + logn)$ time, where s = the number of items returned by the iterator and n = the total number of nodes in the tree.

The following tests in BSTSetTest.java are relevant to this method: testKeyRange1,2,3.

b) You must write at least two more tests.
- testKeyRange4: It must test a different tree structure and range than the other 3 tests.
- testKeyRangeOutOfBounds: Test a situation where the range doesn't include any elements in the Set, although the Set does have elements.

Tips

- The subMap method in Goodrich chapter 11.3 is a similar method to keysInRange
- You'll probably need one or more private helper methods and/or inner classes.
- There are two basic approaches we can think of
  - Eagerly build the whole List<T> of keys that are in-range using recursion then return the List<T>'s iterator
  - Define an inner class that implements Iterator<T>. It keeps a frontier (e.g., a Stack) to do the traversal. Advance the traversal after each call to next(). HINT: a preorder traversal requires seeing each node twice (once before visiting left and once after) so may want to mark nodes as "visited" or not.

Remember, your implementation should run in $O(s + logn)$ time and ***not*** run in $O(n)$ time! For example, an implementation that visits *entire* subtrees that are not in range [start, end) is probably O(n).

An example of an O(n) algorithm would be: 1) perform an inorder traversal of the whole tree, inserting nodes into a List, then 2) iterate over the list to remove the elements not between start and end.

Part 3: remove elements from an unbalanced BST

The Set.remove method removes the element if it exists in the Set and returns true if the element was found.

Part 3a: deleteMin

The BSTSet.remove method will depend on the BSTSet.deleteMin method. This method takes a TreeNode n, and the parent of n, and removes the minimum element in the subtree rooted at n.

Tips:

- The first two lines of deleteMin are "pre conditions". Leave them there! They will help with debugging
- To perform the deletion you should use the method `updateParent`. It will greatly simplify your implementation.

We've provided tests in BSTSetTest (called testDeleteMin*) to help you ensure deleteMin is correct before proceeding to the remove method.

Part 3b: remove

Implement the BSTSet.remove method. Recall that there are four cases for removal in a BST:

a) removed node is a leaf
b) removed node has only a left child
c) removed node has only a right child
d) removed node has two children

Case d is the tricky one. Use the following algorithm, adapted from your textbook:

1) use deleteMin to delete the smallest element in right subtree
2) use the data of the node returned by deleteMin to overwrite the data in the "removed" node.

Take the time with some examples on paper (the remove test cases in BSTSetTest.java are one good source of examples) to convince yourself why the above algorithm works.

There are several tests called testRemove* in BSTSetTest to help you debug.

Part 4: Balanced tree

The BSTSet does not keep the tree balanced, and we know that an unbalanced tree can make contains/add/remove operations take O(N) in the worst case instead of O(log N). To improve your Set implementation, you will complete a second class that uses a balanced binary search tree, AVLTreeSet. Notice that this class extends BSTSet, borrowing most of the functionality.

We've provided implementations of add and remove that call a balancing method to fix the balance property after an insertion or removal. Your job will be to complete the balancing functionality by implementing two methods

- AVLTreeSet.rotateLeft
- AVLTreeSet.rotateRight

See the comments above these methods to see a detailed description of what each method should do. Notice that the AVLTreeSet.balance method uses `rotateLeft` and `rotateRight` in combination to do the double rotation cases, so you don't have to directly implement double rotations.

Tips:

- as in the remove implementation, you should use the `updateParent` method to change what node is at the top of the subtree. It will greatly simplify your code.
- Note that the rotation changes the height of the tree, so make sure to call `updateHeight` at the end. (***Do not* call updateHeightWholeTree**)

All of the tests in AVLTreeSetTest.java should pass when the rotate methods work.

Part 5: Stress test

You've put a lot of effort into these Set implementations, so try it out on bigger data than the tiny test cases! We've provided an example file StressTest.java. It times how long add() takes for 3 implementations of Set

- BSTSet
- AVLTreeSet
- java.util.TreeSet

The program tries a uniform random distribution of inputs, as well as an increasing order of inputs.

Answer the following in a file called hw7.pdf. Visuals such as bar graphs are encouraged. Just make sure to explain them.

a) What observations of heights and running times can you make from running StressTest?
b) Give an explanation of the heights and running times you observed, based on what you know about the 3 implementations. In other words, say "why" the heights and running times are what they are. (You can read about java.util.TreeSet at https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html)

This part will be graded on these 2 criteria:

- Did you answer the questions completely and specifically? Are your explanations accurate?
- Are your answers concise and clear?

Tips for Testing and Debugging

- We've provided a method `BSTSet.bulkInsert()` that is helpful for creating a tree.

- We have also provided an implementation of `TreeNode.equals()` so that trees can be compared properly in JUnit assert* methods.
- In your tests, do not build your test trees manually. Use `add()` or `bulkInsert()`.
- **IMPORTANT**: the methods checkIsBST (for BSTSet or AVLTreeSet) and checkIsBalanced (for AVLTreeSet) are **very helpful for debugging**. These methods throw an exception if the invariants of the data structure are violated. Some of the test cases call them. You should use them in your own test cases and even within your code at places where you know the invariants should hold (for example, at the end of the remove method).

Extra credit (up to 5 points)

Submission: put your work in the base of your GitHub repository in a file named extracredit.pdf.

You may attempt this extra credit no matter how much of the rest of the assignment you complete. However, the effort-to-points ratio may not be as high. The assignment is more open ended and allows for exploration and problem solving.

Read the code in StressTest.java. You'll notice that it prints the height of the trees inside of the BSTSet and the AVLTreeSet. The height is not part of the Set/NavigableSet interfaces, yet we exposed it for our testing and debugging purposes.

In contrast, we could not print out the height of the underlying tree in the java.util.TreeSet. The particulars of the tree implementation are not exposed to the user. Take a look at the public methods of https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html .

Discover interesting features of TreeSet's implementation. Here are some ideas:

- **Option 1a: What algorithm is it?** Look at the source code for TreeSet. Find out what algorithm/data structure TreeSet uses. Your explanation MUST include primary evidence of your findings (e.g., snippets of code that you refer to). We suggest starting in StressTest.java and right-clicking on "TreeSet" and choosing an option under Navigate. In general the options under Navigate will help you explore the source code.
- **Option 1b: Break the interface!** Devise a way to approximate the height of an instance of TreeSet with elements in it without calling any private methods. For example, you might run experiments to figure out the asymptotic run time of the public methods. You should attempt to isolate constant factors with control experiments. Your explanation MUST include plots, which you clearly describe and refer to.