CMPSCI 187 / Spring 2015
Binary Search Trees and Scapegoat Trees

Due on Thursday, 09 April, 8:30 a.m.

Marc Liberatore and John Ridgway

Morrill I N375
Section 01 @ 10:00
Section 02 @ 08:30

Contents

Overview

In this assignment, you'll start with a codebase for binary search trees similar to that presented in lecture. You'll need to implement several additional methods for binary search trees. Then, you'll subclass the binary search tree to create a "scapegoat tree" — a simple form of a *self-balancing* binary search tree.

Learning Goals

- Show understanding of binary trees, sufficient to implement several non-trivial methods.
- Apply forethought to determine if iterative or recursive methods are more straightforward in various cases.
- Learn and implement a simple extension to binary search trees based on a textual description of the extension.
- Continue to develop unit-testing skills.

General Information

Read this entire document. If, after a careful reading, something seems ambiguous or unclear to you, then email cs187help@cs.umass.edu immediately.

Start this assignment as soon as possible. Do not wait until 5pm the night before the assignment is due to tell us you don't understand something, as our ability to help you will be minimal.

Reminder: Copying partial or whole solutions, obtained from other students or elsewhere, is academic dishonesty. Do not share your code with your classmates, and do not use your classmates' code.

You are responsible for submitting project assignments that compile and are configured correctly. If your project submission does not follow these policies exactly you may receive a grade of zero for this assignment.

Policies

- For some assignments, it will useful for you to write additional class files. Any class file you write that is used by your solution MUST be in the provided `src` directory you export.
- The TAs and instructors are here to help you figure out errors, but we won't do so for you after you submit your solution. When you submit your solution, **be sure to remove all compilation errors from your project**. Any compilation errors in your project will cause the autograder to fail, and you will receive a zero for your submission.

Test Files

In the `test` directory, we provide several JUnit test cases that will help you keep on track while completing the assignment. We recommend you run the tests often and use them to help create a checklist of things to do next. But you should be aware that we deliberately don't provide you the full test suite we use when grading.

We recommend that you think about possible cases and add new `@Test` cases to these files as part of your programming discipline. Simple tests to add will consider questions such as:

- Do your methods taking integers as arguments handle positives, negatives, and zeroes (when those values are valid as input)?
- Does your code handle unusual cases, such as empty or maximally-sized data structures?

More complex tests will be assignment-specific. To build good test cases, think about ways to exercise methods. Work out the correct result for a call of a method with a given set of parameters by hand, then add it as a test case. Note that we will not be looking at your test cases, they are just for your use.

Before submitting, make sure that your program compiles with and passes all of the original tests. If you have errors in these files, it means the structure of the files found in the `src` directory have been altered in a way that will cause your submission to lose some (or all) points.

Import Project into Eclipse

Begin by downloading the starter project and importing it into your workspace. It is very important that you **do not rename** this project as its name is used during the autograding process. If the project is renamed, your assignment will not be graded, and you will receive a zero.

The imported project may have some errors, but these should not prevent you from getting started. Specifically, we may provide JUnit tests for classes that do not yet exist in your code. You can still run the other JUnit tests.

The project should normally contain the following root items:

`src` This is the source folder where all code you are submitting must go. You can change anything you want in this folder (unless otherwise specified in the problem description and in the code we provide), you can add new files, etc.

`support` This folder contains support code that we encourage you to use (and must be used to pass certain tests). You must not change or add anything in this folder. To help ensure that, we suggest that you set the support folder to be read-only. You can do this by right-clicking on it in the package explorer, choosing Properties from the menu, choosing Resource from the list on the left of the pop-up Properties window, unchecking the Permissions check-box for Owner-Write, and clicking the OK button. A dialog box will show with the title "Confirm recursive changes", and you should click on the "Yes" button.

`test` The test folder where all of the public unit tests are available.

`JUnit 4` A library that is used to run the test programs.

`JRE System Library` This is what allows Java to run; it is the location of the Java System Libraries.

If you are missing any of the above or if errors are present in the project (other than as specifically described below), seek help immediately so you can get started on the project right away.

Problem 1

Note that for both parts of this assignment, you **are** allowed to use classes that implement the `Collection` class when needed. For example, you may use `java.util.Queue` in your implementations of the required iterators.

What to Do

In the imported project, take a look at `BinarySearchTree` and `BSTInterface`. As we discussed in class, the `BSTInterface` is very similar to the `ListInterface`, and as in DJW, we'll make the same assumptions for BSTs as ordered lists (no nulls, multiple copies of values allowed, etc.). We'll also mandate that BSTs must obey the BST rule in this and the following problem.

Your first task is to implement the methods that are not yet implemented (that is, whose method bodies are marked with `TODO`s). In doing so, you may find that you need or want to change other methods. This is generally OK, with the following restrictions: Your changes must not break the semantics of the methods. And you may not change the signatures of **`public`** methods, or add or remove such methods to the interface.

Here are some specific hints for the methods we've left for you to complete:

- `contains`: We implemented this method in lecture, and noted that it could easily be implemented in terms of `get`. Consider doing so here. Or copy the version we did in lecture – either is acceptable, but the former is simpler.
- `get`: The structure for this method is nearly identical to the `contains` method (and the associated recursive helper method) we covered in lecture. Use that as your starting point.
- `getMinimum` and `getMaximum`: These methods should be straightforward, given the BST Rule.
- `height`: The height of the tree is the height of the node that is furthest from the root. A recursive solution involving `Math.max` is probably easiest.
- `pre-` and `postorderIterator`: These methods are probably easiest to implement in terms of a `Queue` — see `inorderIterator` for an example. Recursively traverse the tree in the correct order, and insert each node into the queue as it is visited. Then, just return the result of calling `iterator()` on the queue.
- `equals`: This method is probably easiest to express with a recursive helper method.
- `sameValues`: Have you already implemented a method that imposes a deterministic order on the values in the tree? (Spoiler: yes.) If so, that will make writing this method straightforward.
- `isBalanced`: For this assignment, a tree is considered balanced when $2^h \leq n < 2^{h+1}$, where h is the tree's height, and n its size.
- `balance`: Review your notes from lecture, and the relevant section in the textbook: DJW fairly clearly outline one approach to the algorithm in pseudocode.

And here are some notes on some of the utility methods we've provided to you, that you should not (and in the case of `getRoot`, must not) change:

- `getRoot`: This method returns the root node of the tree. Normally, you wouldn't expose such a detail in your implementation, but we require it in order to run some of the autograder tests. You may want to use it in your own testing, or with the following method.

Problem 1 continued on next page...

- `toDotFormat`: This method will output a representation of the tree rooted at the given node in the DOT language, as described by its left and right child references. There are many programs that can read this language and display the results. For example, `http://sandbox.kidstrythisathome.com/erdos/index.html` allows to do so in your browser. You may find this helpful in debugging.

Finally, note that here may come a time when you want to allocate an array of generic (`T`) objects in your implementation of `BinarySearchTree`. So you'll do what you've always done: `T[] array = (T[]) new Object[size()];`

And to your dismay, you'll get a `ClassCastException`. Why? Because `T` is no longer an unconstrained generic type. Its full signature is `T extends Comparable<T>`. To satisfy the JVM's run-time type constraints on arrays, you'll need an array capable of holding objects compatible with that type: `T[] array = (T[]) new Comparable[size()];`

Tests

You'll note that for this problem (and the next one), we have provided some tests. These are an absolutely minimal set of tests, and definitely do not cover all possible cases!

You will need to come up with some tests of your own. Try to think of all of the cases that could occur in each method you write, and write tests to check that each of them. Look back at previous assignments' tests for a refresher if you're not sure how to do this. You will not be graded on your tests, but writing them and passing them is the only way that you can be reasonably sure that your code works.

Problem 2

Self-Balancing Binary Trees

As noted in lecture, just obeying the BST rule is not sufficient to achieve O($\log n$) performance when accessing a binary search tree. The tree must be balanced, or close to it. But, your implementation of `balance` was likely at least O(n). Therefore it does not make sense to call `balance` before or after each other method, as it will make all methods asymptotically slower than O($\log n$), negating the performance advantage that motivated binary search trees in the first place!

The solution to this paradox is to build a *self-balancing* binary tree, that is, a tree that maintains some invariant across calls to `add` or `remove`. This invariant enforces that the tree remains approximately balanced, and does so at an amortized low cost. (You can think of this as an analog to the technique of resizing the array that backs an array-based stack or queue: by doing so infrequently and in a specific way, we still achieved asymptotically good performance.)

In CMPSCI 311 you'll learn about one or more of the self-balancing BSTs that are used in practice, such as red-black trees or splay trees. Here, we'll focus on a simpler, but still reasonably effective, form of self-balancing tree: the *scapegoat tree*.

Problem 2 continued on next page...

Scapegoat Trees

The scapegoat tree is based on the common wisdom that, when something goes wrong, the first thing people tend to do is find someone to blame (the scapegoat). Once blame is firmly established, we can leave the scapegoat to fix the problem.

A `ScapegoatTree` keeps itself approximately balanced by partial rebuilding operations. During a partial rebuilding operation, an entire subtree is deconstructed and rebuilt into a perfectly balanced subtree.

A `ScapegoatTree` is a binary search tree that, in addition to keeping track of the number of nodes in the tree (its `size`), also keeps a counter, `upperBound`, that maintains an upper-bound on the number of nodes.

Suppose n = `size` and q = `upperBound`. Then, after each `add` or `remove`, we require that the tree obey a form of the following inequalities, known together as the *scapegoat rule*. First, that:

$$q/2 \leq n \leq q$$

In addition, a `ScapegoatTree` has logarithmic[1] height; at all times, the height of the scapegoat tree does not exceed

$$\text{height} \leq \log_{3/2} q \leq \log_{3/2} 2n < \log_{3/2} n + 2$$

By obeying the scapegoat rule, a scapegoat tree will retain asymptotically logarithmic (amortized) performance on the relevant operations: `add`, `remove`, etc. We'll leave aside the details of the theory of scapegoat trees and the proof of the rule's effect, though if you're curious we suggest taking a look at https://en.wikipedia.org/wiki/Scapegoat_tree and http://opendatastructures.org/ods-java/8_1_ScapegoatTree_Binary_Se.html.[2]

What to Do

A `ScapegoatTree` is a specialized version of a `BinarySearchTree`, and you will implement it as a subclass of `BinarySearchTree`. You will need to change the semantics of the `add` and `remove` methods in this subclass as follows.

To implement the `add` method, first increment `upperBound` and then use the usual algorithm for adding the element to a binary search tree. At this point, you may get lucky and the height of the tree might not exceed $\log_{3/2}$ `upperBound`. If so, then leave well enough alone and don't do anything else.

Unfortunately, it will sometimes happen that, by adding this node, you've increased the tree's height to greater than $\log_{3/2}$ `upperBound`. In this case, you need to reduce the height to maintain the invariant required by the second half of the scapegoat rule. This isn't too big of a job: There is only one node (let's call it `u`), whose height exceeds $\log_{3/2}$ `upperBound`. `u` must be the node you just added — do you understand why?

To fix `u`, follow the `parent` pointers from `u` back up toward the `root` looking for a *scapegoat*, `w`. The scapegoat, `w`, is a very unbalanced node. It has the property that `sizeOfSubtree(w.child) / sizeOfSubtree(w)` $> \frac{2}{3}$, where `w.child` is the child of `w` on the path from the root to `u`.

[1] Recall that $\log_x y = \frac{\log_z y}{\log_z x}$ for any real x, y, z.

[2] We gratefully acknowledge Pat Morin and the other contributors to the latter, from which we drew this problem.

Problem 2 continued on next page...

We'll omit the proof the the scapegoat exists – you can take it for granted.[3] Once you've found the scapegoat `w`, rebuild the subtree rooted at `w` into a balanced binary search tree. Be careful here! You must balance the subtree, then graft the subtree's root into the place `w` formerly occupied in the original tree. In particular, make sure you set the appropriate child reference in `w`'s original parent to point to the now-balanced subtree's root, and adjust the subtree's root node's parent pointer.

We know from the scapegoat rule that even before the addition of `u`, `w`'s subtree was not a complete binary tree. Therefore, when we rebuild the subtree rooted at `w`, the height decreases by at least 1 so that the height of the `ScapegoatTree` is once again at most $\log_{3/2}$`upperBound`.

The implementation of `remove(element)` in a `ScapegoatTree` is simpler than that of `add`. Search for `element` and remove it using the usual algorithm for removing a node from a `BinarySearchTree`. This removal can never increase the height of the tree, but it may reduce the size. So, check if `upperBound` $>$ 2 $\times$ `size`. If so, the first half of the scapegoat rule no longer holds! To fix it, *rebuild the entire tree* into a perfectly balanced binary search tree (by calling `balance`), and set `upperBound` equal to `size`.

Some Hints

You'll probably want to modify `BSTNode` to maintain a reference to the node's parent. You might add a getter and setter, but you could also do this update when calling `setLeft` and `setRight`.

There are at least two ways to do this problem without using `parent` references at all. You could write a method to find the path of nodes from the root to the highest node, store the path in a list, and use that list to search for the scapegoat. Or you could temporarily flip child pointers to point at the parent, as described in the Wikipedia article.

Any of these approaches are acceptable – we won't be testing that your `parent` pointers are correct, only that your tree obeys both the BST rule and the scapegoat rule (and does not just call `balance` after every `add` or `remove`, but rather behaves as described above).

You may want to change some of the helper methods in `BinarySearchTree` from **`private`** to **`protected`**, so that `ScapegoatTree` can access them. That's fine. You may also want to change some of their signatures, or add new **`protected`** or **`private`** methods to either class, or override more methods in `ScapegoatTree`. Any of that is fine too, but remember that you must not break `BinarySearchTree`!

Export and Submit

When you have completed the changes to your code, you should export an archive file containing the entire Java project. To do this, click on the `bst-scapegoat-student` project in the package explorer. Then choose "File → Export" from the menu. In the window that appears, under "General" choose "Archive File". Then choose "Next" and enter a destination for the output file. Be sure that the project is named **`bst-scapegoat-student`**. Save the exported file with the `zip` extension (any name is fine). Log into Moodle and submit the exported zip file.

[3] That is, if your code doesn't find it, it's due to an error in the code, not the algorithm. The scapegoat's existence is guaranteed.