Project 6: Scapegoat Tree

A simple form of a self-balancing binary search tree.

Overview

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

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:

- Demonstrate an 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.
- Experience debugging and testing code.

Project Specification:

Background.

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 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.)

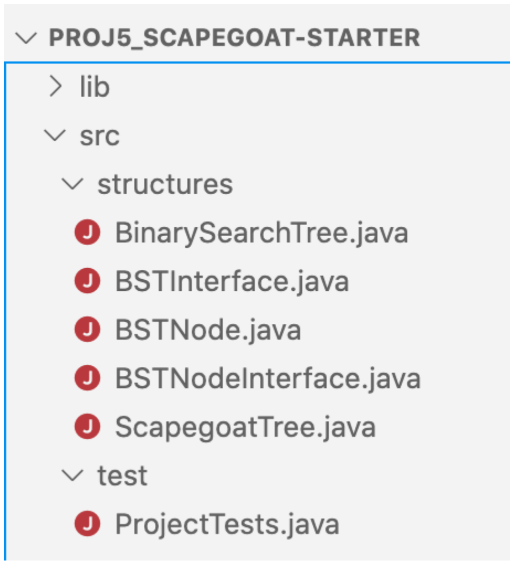
Code Structure.

This project contains the following important folders:

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.

src/test: This is the package folder where you can find all of the public unit tests.

lib: This is where you can find libraries that are included with the project. At the very least you will find two jar files that are used to run the JUnit test framework.



```
PROJ5_SCAPEGOAT-STARTER
  lib
  src
    structures
      BinarySearchTree.java
      BSTInterface.java
      BSTNode.java
      BSTNodeInterface.java
      ScapegoatTree.java
    test
      ProjectTests.java
```

Fig 1: Code structure for the project.

You will first focus on finishing the implementation of the `BinarySearchTree` class, along with its supporting classes, then you will implement the balancing algorithm in the `ScapegoatTree` class, a subclass of `BinarySearchTree`. Note that in the starter code:

- some methods return default or random values.
- the main methods and some tests cannot run as there are incomplete TODOs.

Tasks and TODOs

The project TODOs denote where you need to make changes in the source code.
There are two main “problems” in this project, each problem containing a number of TODOs.

PROBLEM 1: Complete **`BinarySearchTree.java`**

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. You may also add private helper methods as required.

Review the code in `BinarySearchTree` and `BSTInterface`. The `BSTInterface` is very similar to the `ListInterface`, and as in the book, 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 in problem 2 i.e. the ordering property that any node's left subtree keys ≤ the node's key, and the right subtree's keys > the node's key. Check the `addToSubtree` method in the `BinarySearchTree` class.

Tasks
Your first task is to implement the methods that are not yet implemented (that is, whose method bodies are marked with TODOs). In doing so, you may find that you need or want to change other methods. Note these restrictions:

1. Your changes must not break the semantics of the methods.
2. You may not change the signatures of public methods, or add or remove such methods in the interface.

Here are some specific hints for the methods we’ve left for you to complete:
`get:` Returns the item t of type T if it is in the tree or null if it doesn’t exist. The structure for this method is nearly identical to the **`contains`** method. Throws a `NullPointerException` if the parameter `t` is null.
`contains`*:* Determines if the given item `t` of type `T` is in the tree. You should implement this method in terms of the `get` method. Throws a `NullPointerException` if the parameter t is null.
`getMinimum and getMaximum`*:* Follow 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 useful. Note: the height of a one-node tree is 0.

preorderIterator 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, return the result of calling `iterator()` on the queue. The **inorderTraverse** method is given to you.

equals*:* This method is best expressed with a recursive helper method. It returns true if and only if the trees are structure- and value-equivalent and throws a **NullPointerException** if other is null.

sameValues: returns true if and only if the trees are value-equivalent and throws a **NullPointerException** if other is null. Hint: Think about re-using the code from the equals method.

isBalanced: Read the comments in the interface. 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 is its size.

balance: Review the course material, including the links and additional material on the Scapegoat algorithm.

We include the following pseudocode for the balance method and its helper method.

```
Balance method pseudocode:
let:
inorderIter be an inorder iterator
size = size of tree
values be an array to store the data elements

for(int index =0; index<size;index++){
   values[index] = inorderIter.next()
}
reset root to null
Balance helper(values, 0, size-1)

Balance helper(values, low, high) pseudocode:

if(low==high)
   tree.add(values[low])
else if ((low + 1) == high)
   tree.add(values[low])
   tree.add(values[high])
else
   mid = (low + high)/2
   tree.add(values[mid])
   Balance helper(values, low, mid-1)
```

```
                Balance helper(values, mid + 1, high)
```

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.

`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 you to do so in your browser. You may find this helpful in debugging.

Finally, note that there 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()];
```

Problem 2: Implement a Scapegoat Tree

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), it 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:

$$q/2 \le n \le q$$

In addition, a `ScapegoatTree` has logarithmic height (recall that $\log_x(y) = \log_z(y) / \log_z(x)$ for any real x, y, z); at all times, the height of the scapegoat tree does not exceed:

$$\text{height} \le \log_{3/2}(q) \le \log_{3/2}(2n) < \log_{3/2}(n) + 2$$

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 (we gratefully acknowledge Pat Morin and the other contributors to the latter, from which we drew this problem). Please check out the other additional material we provide on Scapegoat Trees.

Tasks

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.

`add:` 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 log3/2 upperBound. If so, then leave well enough alone and don't do anything else. You can check the java `Math.log` method for the logarithm part.

Unfortunately, it will sometimes happen that, by adding this node, you've increased the tree's height to greater than log3/2 upperBound. In this case, you need to reduce the height to maintain the invariant required by the second half of the scapegoat rule. There is only one node (let's call it u), whose height exceeds log3/2 upperBound. u must be the node you just added - make sure 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`) > `2/3`, where `w.child` is the child of w on the path from the root to u.

We'll omit the proof that the scapegoat exists - you can take it for granted (That is, if your code doesn't find it, it's due to an error in your code, not the algorithm. The scapegoat's existence is guaranteed.) 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. NEED MORE CLEAR EXPLANATION OF THIS PART TO CUT DOWN QUESTIONS FROM STUDENTS.

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).

`remove`: Search for the 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 × 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` to `size`.

Some Hints

You'll probably want to modify `BSTNode` to maintain a reference to the node's parent. You might add a get and set method for this reference, 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`!

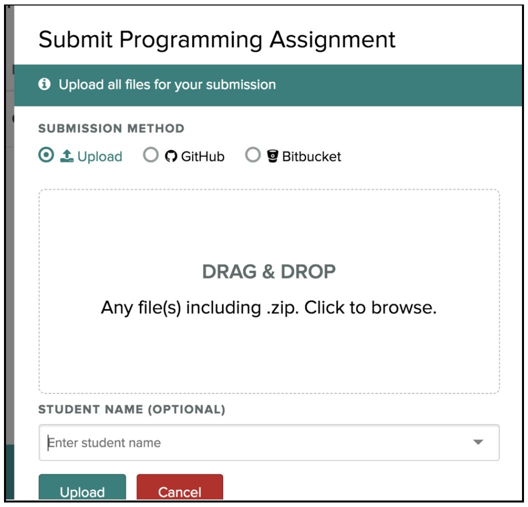
Note on Testing

You'll note that for problems 1 and 2, we have provided some tests. These are an absolutely minimal set of tests, and definitely do not cover all possible cases! Take the time to create 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 each of them.

Export and Submit

Step 1: Export your project
Use the Archive extension or other means to create the correct zip file. If your zip file is not in the correct form, the autograder will not process your code and you will not receive any credit. The autograder will not run (often stating an error occurred) if your code does not compile or if your zip file is not correctly made.

Step 2: Submit the zip file to Gradescope
Log into Gradescope, select the assignment, and submit the zip file for grading.


Submit Programming Assignment

Upload all files for your submission

SUBMISSION METHOD

[✓ Upload] GitHub Bitbucket

DRAG & DROP

Any file(s) including .zip. Click to browse.

STUDENT NAME (OPTIONAL)

Enter student name

The autograder will not run successfully if you do submit a **correctly formatted zip file**- it has to have the same **names and directory structure as described earlier**. The autograder will also not run if your code does not compile, or if you import libraries that were not specifically allowed in the instructions.

Remember, you can re-submit the assignment to gradescope as many times as you want, until the deadline. If it turns out you missed something and your code doesn't pass 100% of the tests, you can keep working until it does. Attend office hours for help or post your questions in Piazza.

Academic Honesty Policy 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. Posting solutions to the project on public sites is not allowed. If you are confused about what constitutes academic dishonesty, you should re-read the course policies.