EECS 2011: Assignment 2

July 6, 2017
8 % of the course grade
Due: Thursday, July 20, 2017, 23:59 EDT

Motivation

The purpose of this assignment is to evaluate two implementations of binary search trees in terms of their performance for different insertion and deletion operations. The trees will then be tested to implement a TreeSort sorting algorithm.

Introduction

In computer science, binary search trees (BST) are a particular type of container: data structures that store "items" (such as numbers, names etc.) in memory. They allow fast lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its key.
The `Tree` interface provides three methods to **add** and **remove** elements to and from the tree. It also provides an iterator that visits the elements in-order, as well as a function `height` that simply returns the height of the tree.
Note that the speed of these operations may strongly depend on the implementation.

Description

In this assignment, you will have to write two implementations of `Tree` interface (provided), one that uses regular (possibly unbalanced) **Binary Search Trees**[1], and one that uses **balanced AVL Trees**[2]. After that, you will have to test the performance of TreeSort[3] when using your implementations. For your convenience, some starting code is provided. Note that it either does not implement some features or implements them improperly

Part 1

Implement the necessary public methods in the two implementations (called `A2BSTree` and `A2AVLTree`) of `Tree` interface:

```
public void add(E e);
public void addAll (Collection<? extends E> c);
public boolean remove(Object o);
public Iterator<E> iterator();
public int height();
public int size();
```

[1] https://en.wikipedia.org/wiki/Binary_search_tree
[2] https://en.wikipedia.org/wiki/AVL_tree
[3] https://en.wikipedia.org/wiki/Tree_sort

The classes should use generics. The AVL *add* and *remove* operation should keep the tree balanced. It should also be possible to have duplicate items in the trees (something that binary search trees normally do not allow); think about how you can work around it.

Of course, you are free to implement any private or protected methods and classes as you see fit. However, you may not have any public methods other than the ones mentioned (or the ones present in the interface or its superclasses).

Part 2

Name your class `TreeSort`.
The class should have the following methods:

```
public static <E> void sort( E[] a);
public static <E> void sort(Tree <E> tree, E[] a);
```

See the comments in the code provided to determine their behaviour.

Part 3

Name your class `SortTester`
Take both of your tree implementations and compare them when used to implement a TreeSort sorting algorithm.

For numbers $N = \{10, 100, 1000, 10000, 100000, 1000000\}$
a) Starting with **shuffled** *arrays* of N `Number`-s, measure how long it takes to sort such an array using regular BSTs and balanced AVL trees.

b) Starting with **reverse-sorted** *arrays* of N `Number`-s, measure how long it takes to sort such an array using regular BSTs and balanced AVL trees

At the end, produce the following table (the timing values below are just placeholders and do not relate to any real measurements):

```
N = 10:
BST                 123 ms
AVL                 456 ms
BST(rev-sorted)     567 ms
AVL(rev-sorted)     742 ms

N = 100:
 …
N = 1000:
…
<repeat for all values of N>
```

Save the result of your program execution in a file `testrun.txt` and submit it together with your other files.

NOTES:
1. Make sure you reset the timer (or save the intermediate time before the next measurement); i.e., make sure you measured time contains only the time to perform one set of operations that was supposed to be timed.

2. In case the operations for larger N numbers take too long (e.g., more than 30 s) you may reduce the number to a smaller one or eliminate it (so that you will have a range from, say, 1 to 100000).

3. Do not use *package*-s in your project (put your classes in a `default` package). Using packages will cost you a 20 % deduction from the assignment mark.

4. Name your classes as specified. Using incorrect names will cost you a 20 % deduction from the assignment mark.

5. Some aspects of your code will be marked automatically (e.g., how it handles boundary cases and error conditions). It is also imperative you test your classes. If any of the java files that you submit do not compile, the whole submission will be given a grade of zero, regardless of how trivial the compiler error is.

6. Your code should include Javadoc comments. Also, part of your mark will be based on coding style.

Submission

Submit your work using the `submit` command. Remember that you first need to find your workspace directory, then you need to find your project directory.
`submit 2011 a2 <list of your files>`
(The directory will be created soon).
You can check the usage examples by executing `man submit`.

Alternatively, you may use the web form at
`https://webapp.eecs.yorku.ca/submit/index.php`

You only need to submit 6 files (the interface, two implementations or trees, the sorting class, the tester, and the test run); optionally, you may also submit a file `readme.txt` containing comments for the marker. Make sure you submit Java source code files, and not the compiled classes.

Late penalty is 20 % per day. Submission 5 days or more after deadline will be given a mark of zero (0). Contact the instructor *in advance* if you cannot meet the deadline explaining your circumstances.

Academic Honesty

Direct collaboration (e.g., sharing code or answers) is not allowed (plagiarism detection software[4] will be employed). However, you're allowed to discuss the questions, ideas, approaches you take, etc.

State all sources you use (online sources, books, etc.). Using textbook examples is allowed (still needs to be cited).

[4] http://theory.stanford.edu/~aiken/moss/