CIS 233 – Assignment 1

(125 points)

Due: Mon. 11/13 @ 11:59p (code and documents uploaded)

In this assignment you will obtain experience in implementing a balanced tree algorithm and modifying its basic functionality. Be sure to read and follow the specifications carefully, especially the method signatures.

Implement an AVL tree that builds on the Weiss version in these ways:

1) Accommodates (generic) objects with duplicate keys (as determined by that object's `compareTo()` method). All objects that are added must be maintained as individual instances (i.e. a counter is insufficient – multiple objects can have the same key but not be completely identical.)
2) All removals are successfully implemented using lazy deletion.
3) Implements a `printBalTree()` method(s). This method has the same signature as `printTree()`, with the addition of a boolean value for the last parameter. This method outputs the entire tree (including lazily deleted nodes) as follows:
   a. Print each node's (including duplicates!) element(s), height, and current balance condition (right – left) on a line by itself. -1 represents a (sub)tree that is heavy to the left, 1 is a (sub)tree heavy to the right, and 0 is a balanced (sub)tree.
   b. Have your output show the node's status (Active/Inactive)
   c. In addition – print out the data (only) of the node's children, or "null" if that subtree is null. Clearly identify each child (L or R) and indent. Each child must be on its own line.
   d. An empty tree must be depicted as "Tree is currently empty."

   Sample output (e.g. wrapped `Integer`s):

```
Data: 1      Height: 0     Balance: 0   Status: Active
      Left:  null
      Right: null

Data: 2      Height: 1     Balance: 0   Status: Inactive
      Left:  Data: 1       Height: 0    Balance: 0
      Right: Data: 3       Height: 0    Balance: 0
Etc….
```

   e. If the boolean parameter is `true`, the values are output in ascending order. `False`, descending. Note that you build only one tree, and it must be maintained in ascending order.

4) Implement a `writeBalTree()` method that works identically to `printBalTree()`, but writes the data (with indention) to a text file. The file must be named: `A1233XXxxAVLout.txt` (`XXxx` = first initial and first 3 letters of your last name) with no path – i.e., it writes to the default directory.

(continued on next page)

5) `printTree()` only displays active items.
6) Change the `contains()` method to private visibility and replace it by implementing a public `findAll()` method that takes an `AnyType` as its only parameter and returns all active items that match the parameter as a Collection (your choice, choose well) of `AnyType`.
7) Change the `remove()` method to private visibility and replace it by implementing a public `removeAll()` method that takes an `AnyType` as its only parameter and a `void` return type that will (lazily) delete all items matching the parameter from the tree.
8) Implement a `findMode()` method similar to previous assignments. The return type is `Result<AnyType>`, using the `Result` interface provided with this assignment.
9) In service of 8), create a custom static nested class with private visibility that implements the `Result` interface made famous in 232. This object is merely a container for the results generated by `findMode()`. The object itself is not to have any mode computation logic as that is the role of `findMode()`.
10) Implement a method named `author()` that returns a string consisting of your name (only).
11) All output does needs to incorporate all duplicate values. You have some flexibility as to how duplicate values are displayed as long as the output is clear about how many times the value occurs. It may be helpful to have your output suggest what your duplicate strategy is. All instances of duplicates must be retained.

These alterations must be implemented so that all operations are as optimal as possible given the use of lazy deletion.

Your tree needs to work with any `Comparable` data, so make sure you do the following while you are testing (but don't include in your submitted assignment):

- Provide a data object type for my testing that implements `Comparable` (generically) and has an overridden `toString()` method.
- Write an application class which will do the following (write your own test class or temporary `main()` to do same – don't include that when you turn the assignment in):
  - Create an instance of your tree
  - Create the data objects
  - Invoke operations

(continued on next page)

Turn in the following documentation *(b and c - separate pro quality docs)*:

a. Reasonable comments as needed. Avoid obvious comments.
b. Brief (but complete) Executive Summary describing what the tree does, targeted at a non-technical audience. This needs to be properly labeled and professionally formatted.
c. A Design Outline that describes how you chose to implement points 1-6 above, and why you made those choices (include what the alternatives were and why you didn't choose them.) Be sure to indicate why your new/altered methods meet the complexity standards. Assume that the reader's technical knowledge is at least equal to your own and the tone should reflect that. This document also must be properly labeled/formatted.
d. Be sure that the wording used is appropriate to each of the target audiences in the documents. Strive to be clear and specific while remaining aware of the reader's point of view and knowledge.

Additional notes:

1. Upload your one and only source code file (as an attachment) to Canvas by the deadline. Send only the tree class and any nested/inner class(es) that it will be utilizing. Send the file as a zip that includes your documentation files (PDFs).
2. All of your classes must be prefaced with `A1233XXxx`. Name your tree class `A1233XXxxAVL`.
3. All of your classes must belong to this package: `cis233.a1;` - using another package will result in a noticeable deduction. Note that the package name is the same for everyone.
4. You can make use of any code available in the book or the other (Data Structures and Algorithm Analysis) Weiss book (whose code is on the Q drive) in implementing this solution. You can also use basic API data structures (e.g., arrays and `ArrayList`s) as desired. Everything else must come from you.
5. `UnderflowException` and `Result` will be available in the package when I test it. Don't send them. Use the `Result` and modified `UnderflowException` that provided on Canvas.
6. This is an individual assignment. The code, strategies, and approach must come solely from you with the exceptions granted in point 4 above. This stipulation also includes other people (directly or via any manner of online forum) or any other resources. There is to be no discussion/exchange of this assignment with other students (present, past, or future.)
7. It is expected that everything about this assignment will be done at high level of precision and efficiency. Anything substandard about the submission will incur a noticeable penalty. Aggressively test your code to make sure that it meets all specifications. Make sure that you are using the correct method signatures and catching any exceptions to avoid significant penalties.