TCSS 342 - Data Structures
Assignment 7 - AVL Tree

Version 1.1.1 (May 10, 2022)

Time Estimate: 6-8 hours

Learning Outcomes

The purpose of this assignment is to:

- Build an AVL self-balancing binary search tree data structure.
- Apply the AVL self-balancing binary search tree to a problem well suited to it.
- Gain experience testing and debugging data structures.
- Build data structures exactly to an API's specifications.
- Read a text file and extract the unique words.

Overview

An AVL tree is one of two common self-balancing binary search trees. In this assignment we will build an AVL self-balancing binary search tree. Self-balancing binary search trees are an upgrade to naive binary search trees because in all cases they can maintain a logarithmic run time for all three primary functions: insert, delete, and search.

While you should test your data structure yourself, we will put the binary search tree to the task from Assignment 2 - Unique Words. So for this assignment we will upgrade the UniqueWords class.

To complete this you will upgrade two classes:

- MyBinarySearchTree<Type> (with a private Node)
- UniqueWords

Formal Specifications

MyBinarySearchTree<Type extends Comparable<Type>>

`…`
`+rotations : Integer`
`-balancing : boolean`

`…`
`+MyBinarySearchTree(balancing : boolean)`
`+add(item : Type)`
`-add(item : Type, subTree : Node) : Node`

```
+remove(item : Type)
-remove(item : Type, subTree : Node) : Node
-updateHeight(node : Node)
-rotateRight(node : Node) : Node
-rotateLeft(node : Node) : Node
-rebalance(node : Node) : Node
```

Field summary

- `rotations` - This value is used to count the number of rotations made. It should be incremented once in `rotateRight` and once in `rotateLeft`.
- `balancing` - If this flag is true the binary search tree will rebalance itself after adds and removes. Otherwise the binary search tree will not rebalance itself.

Method summary

- `MyBinarySearchTree` - Add a constructor that takes a boolean as an argument and sets the balancing flag. Also leave a default constructor that sets balancing to false so your prior code will still work without change (i.e. backwards compatibility).
- `add` - This method should be upgraded to rebalance the tree if necessary after the add. It should do this only if `balancing` is true. This method should run in O(d) time where d is the depth the `item` added.
- `remove` - This method should be upgraded to rebalance the tree if necessary after the remove. It should do this only if `balancing` is true. This method should run in O(d) time where d is the depth the `item` added.
- `updateHeight` - Updates the height and the balance factor of the `node`. This method should run in O(1) time.
- `rotateRight` - Performs a right rotation on `node`. This method should run in O(1) time.
- `rotateLeft` - Performs a left rotation on `node`. This method should run in O(1) time.
- `rebalance` - Checks `node` for imbalance and if found performs the appropriate rotations to correct it. This method should run in O(1) time.

Node
… `+balanceFactor : int`
`+toString() : String`

Method summary

- `balanceFactor` - The balance factor of the node. Defined in AVL trees to the height of the left subtree minus the height of the right subtree.

Method summary

- `toString` - Returns the contents of the node in this format:
  <item>:H<height>:B<balanceFactor>

UniqueWords
…
… +addUniqueWordsToAVL()

Method summary

- `addUniqueWordsToAVL` - Adds the unique words to a self-balancing MyBinarySearchTree.
  a. For each word in the list stored in book:
     - ■ Check to see if the word is stored in the binary search tree of unique words.
       - If it is not, add it to the binary search tree.
       - Otherwise, skip this word.

  b. Calls toString from the binary search tree to extract the words in order.

  This method should time both steps and output each runtime to the console as well as the number of unique words, height of the binary search tree and the number of comparisons made. This method should run in time O(nlogn) where n is the number of words in the book.

Testing

It is important that you test your code to ensure it works on all potential inputs. Please do this in a separate Main class, and without using additional tools (like JUnit). You will not submit your test code. Instead, your data structures will be tested by code developed by the instructor and/or grader. This code will not be provided to you, as you should test your code before submitting it. If your code fails our tests we will let you know which tests it fails so you can find and correct your errors.

Here is the output from my solution:

```
Reading input file "./WarAndPeace.txt"... 3291642 characters in 84 milliseconds.

Finding words and adding them to a linked list...  in 347 milliseconds.
```

```
The linked list has a length of 570240.

Adding unique words to an AVL binary search tree...  in 190 milliseconds.
20228 unique words
16 height
5946690 comparisons
14945 rotations
Traversing the AVL...  in 31 milliseconds.
```

Submission

You will submit a .zip file containing:

- MyBinarySeachTree.java
- MyLinkedList.java
- BookReader.java
- UniqueWords.java

Grading Rubric

In order to count as complete your submission must:

1. Implement the binary search tree efficiently and without errors.
2. Follow the API exactly so that your classes will work with any testing code.
3. Pass all tests in the testing code. If you fail any tests, you will be told which test failed.
4. Get the numbers right:
    a. Find all 570240 words in the book.
    b. Find the 20228 unique words in the book.
    c. The binary search tree should have depth 16.
    d. Make 5946690 comparisons to find the unique words in the binary search tree.
    e. Make 14945 rotations.

Reminder: Incomplete assignments can always be corrected and resubmitted. If they are completed within 7 days of the due date they will count as late and after that period they will count as missed. Please review the grading matrix for the number of permitted late and missed assignments.