COMPX201 & COMPX241 Assignment Three
Binary Search Tree

Due: Friday 20th May 11:59pm.

In this assignment you will gain experience maintaining a Binary Search Tree (BST) of words. Your program is to process the contents of a file as a stream of words according to the following specification:

- A word is any contiguous sequence of alpha-numeric characters, specifically, the letters A to Z in either upper or lower case, and digits.
- Any other characters are ignored (i.e. treated as a blank space between words).
- Any word found on input must be converted entirely to lowercase letters.
- Given the above, it follows that the contents of the input file can be treated as a stream of words, composed entirely from lowercase letters and digits.

See here for reading the contents of a file: https://www.w3schools.com/java/java_files_read.asp

Part One:

You must write your own BST class and all its required operations. Your program should follow the specifications given below.

1. **BST:** define a class called `BinarySearchTree` in a file called `BinarySearchTree.java`. This class is to implement an unbalanced BST using self-referential nodes. Your solution should support the following public methods using **recursion** where applicable:
    - `search(String value)` - search the tree to find the specified value and return a boolean true if value is found, false otherwise.
    - `insert(String value)` - add the value to the BST, maintain ordering alphabetically by using the `compareTo` method (see https://www.w3schools.com/java/ref_string_compareto.asp for more details) and assume that duplicates are not allowed.
    - `remove(String value)` - remove the specified value from the BST.
    - `dump()` - a method which prints out the tree following an in-order traversal with each value on a separate line.
    - `height()` - returns as an int the height of the tree.
2. **The Node:** define a class called `BSTNode` for the nodes in your `BinarySearchTree`. It can be either an external class in a separate file called `BSTNode.java` or an inner class of `BinarySearchTree`. It should have the following:
    - A public member variable to hold the value of node as a string.
    - A public member variable to hold a link to the left subtree.

- A public member variable to hold a link to the right subtree.
- A constructor that takes a value as a string argument and copies that value into the node's member variable. The constructor should also initialize the left and right subtree links to null.

You may have additional member variables and methods if they are useful to you, but they should be private, except for methods that are used to support part two.

Part Two:

Create a class called `OddWords` in a file called `OddWords.java`. Using your `BinarySearchTree` from part one, your program must adhere to the following specifications:

- Each word found on input must be added to the BST if it is not already there, but deleted if it is found.
- Each time your program searches for a word (either for insertion or deletion) all the words along the search path down the BST must be printed on a single line, separated by spaces, with the target word appearing last.
- If the word is not found, it is added to the tree and the label "`INSERTED`" is to appear after the target word at the end of the output line.
- If the word is found, then it is removed from the tree and the label "`DELETED`" is to appear after the target word at the end of the output line.
- After the file has been processed, have your program output the label "`LEXICON:`" on a line by itself.
- Following the "`LEXICON:`" label, print out all the words still in the BST, one word per line and in dictionary order - that is, you must do an in-order traversal of the BST and print out all the words still in it.

Part Three:

Using JUnit as described in class create two test files, one for each solution to the above parts. You must write four different tests for each of the `BinarySearchTree` operations (at minimum), making sure you think about testing for "edge" cases, like an empty BST. You also need to write four different tests for your `OddWords` program class, two for the case where a word would be added to the tree and two for the case when the word is already in the tree.

Assessment:

Completing Part One can earn up to a C+ grade. You must also complete Part Two to earn up to a B+ grade and to be eligible for an A+ you must also complete Part Three. Your solution will be marked on the basis of how well it satisfies the specification, how well-formatted and easy to read your code is and whether each class and public method has at least some comment explaining what it does, what it's for and what any of its arguments are (i.e. documentation).

Your code should compile and run as a console program from the Linux command-line (i.e. no GUI). Students are encouraged to test their code in the lab prior to submitting their solutions.

Submission:

Create an empty directory (i.e. folder) using your student ID number as the directory name. Place copies of your source code in this directory. If you wish to communicate with the marker any additional information then you may include a plain text README file, but nothing else (e.g. no compiled code). Upload this directory through the Moodle submission page for this assignment.