CSC 226 FALL 2014
ALGORITHMS AND DATA STRUCTURES II
ASSIGNMENT 2 - PROGRAMMING
UNIVERSITY OF VICTORIA

1 Programming Assignment

A Java template containing an incomplete binary search tree implementation has been provided. Your assignment is to complete the `insert()` and `remove()` methods to implement the insertion and removal procedures of an AVL tree. The `find()` method is already implemented, and you are not permitted to change it or any other details of the `AVLTree` class besides the `insert()` and `remove()` methods. You may, however, add functions (or classes) as needed. You should read through the comments in the template carefully before starting, since you will need to use methods and data types the template provides, such as the `TreeNode` class which is used to contain each element of the tree.

1.1 Input Format

The `main()` function in the template contains testing code to perform a series of `insert`, `find` and `remove` operations based on test data entered by hand or read from a file. Input data consists of a set of strings to insert, one per line, followed by the token `"###"`, followed by a set of strings to search for in the tree. After searching for each element, the testing code in `main()` deletes the element if it is found in the tree. The following input file constructs a binary search tree containing the strings `"Victoria"`, `"Vancouver"`, `"Saanich"` and `"Sidney"`, then searches for the strings `"Duncan"`, `"Nanaimo"` and `"Vancouver"`:

```
Victoria
Vancouver
Saanich
Sidney
###
Duncan
Nanaimo
Vancouver
```

1.2 Testing and Verification

Besides code to test the `insert`, `find` and `remove` methods, the template also contains functions to automatically verify the correctness of the constructed tree at each step. The testing code in `main()` will verify the binary search tree property and AVL property after each insertion and removal. These tests are time consuming (verifying the global tree properties is a $O(n)$ operation), but can be helpful for debugging. The comments in the template explain how to enable each test. Do not use the tree verification functions directly in your code (let the code in `main()` handle everything for you). The tests will all be disabled automatically before marking (you do not have to disable them yourself before handing in your code).

2 Test Datasets

Three datasets have been uploaded to the Data tab on conneX. All three insert a list of place names (cities, towns, geographical features, etc.) into the tree. The place name data is based on the freely available geographical data repository at `http://www.geonames.org`. The 'search' section of each dataset contains 1865 strings, based on the names of streets around Victoria. Some of these values will be present in the tree (for example, 'Waterloo' is the name of a road in Victoria and a city in Ontario), but some searches will be unsuccessful. The table below gives the statistics for each dataset.

File	**Insertions**	**Searches**		
		Total	Present	Not Present
`place_names_small.txt`	22107	1865	362	1503
`place_names_med.txt`	43229	1865	527	1338
`place_names_large.txt`	122359	1865	757	1108

3 Sample Run

The output of a model solution on the sample input data in section 1.1 is given in the listing below. Console input is shown in blue. Since the number of inserted values is less than the value of the `PrintTreeSizeThreshold` constant in the template, the contents of the tree are printed out after the insertion phase. The contents are printed as an in-order traversal, with lower levels indented and the left child of each node visited first. For example, in the output below, `Vancouver` is the root, `Saanich` is its left child and `Sidney` is the right child of `Saanich`.

```
Enter a list of strings to store in the tree, one per line.
To end the list, enter '###'.
Victoria
Vancouver
Saanich
Sidney
###
Read 4 strings.
Enter a list of strings to search for in the tree, one per line.
To end the list, enter '###'.
Duncan
Nanaimo
Vancouver
###
Read 3 strings.
Inserted 4 elements.
 Total Time (seconds): 0.00

Tree contents:
----------
         |-- (no left child)
    |--Saanich
    |    |
    |    |    |-- (no left child)
    |    |--Sidney
    |         |-- (no right child)
    |
Vancouver
    |
    |    |-- (no left child)
    |--Victoria
         |-- (no right child)
----------
Tree contains 4 nodes and has height 2.
Searched for 3 items (1 found and deleted, 2 not found).
Total Time (seconds): 0.00
```

4 Evaluation Criteria

The programming assignment will be marked out of 100, with the `insert` and `remove` methods each worth 50 marks, based on a combination of automated testing and human inspection. The implementation of the `insert` method will be marked using the following criteria:

Score (/50)	Description
$0-5$	Submission does not compile or does not conform to the provided template.
$5-25$	The implemented algorithm correctly inserts nodes into the tree but does not perform AVL rebalancing. If the insertion algorithm does not preserve the binary search tree properties, no more than 25 marks will be given.
$26-40$	The implemented algorithm correctly inserts nodes into the tree and correctly performs AVL rebalancing, but the insertion step does not have a $O(\log n)$ running time.
$41-50$	The implemented algorithm correctly inserts nodes and correctly performs AVL rebalancing, and has a $O(\log n)$ running time.

The implementation of the `remove` method will be marked using the following criteria:

Score (/50)	Description
$0-5$	Submission does not compile or does not conform to the provided template.
$5-35$	The implemented algorithm correctly removes nodes from the tree but does not perform AVL rebalancing. If the removal algorithm does not preserve the binary search tree properties, no more than 25 marks will be given.
$36-45$	The implemented algorithm correctly removes nodes from the tree and correctly performs AVL rebalancing, but the does not have a $O(\log n)$ running time.
$46-50$	The implemented algorithm correctly removes nodes and correctly performs AVL rebalancing, and has a $O(\log n)$ running time.

To be properly tested, every submission must compile correctly as submitted, and must be based on the provided template. You may only submit one source file. **If your submission does not compile for any reason (even trivial mistakes like typos), or was not based on the template, it will receive at most 10 out of 100**. The best way to make sure your submission is correct is to download it from conneX after submitting and test it. You are not permitted to revise your submission after the due date, and late submissions will not be accepted, so you should ensure that you have submitted the correct version of your code before the due date. conneX will allow you to change your submission before the due date if you notice a mistake. After submitting your assignment, conneX will automatically send you a confirmation email. If you do not receive such an email, your submission was not received. If you have problems with the submission process, send an email to the instructor **before** the due date.