CSC2001F 2015 Practical 2

1. Introduction

You should read this document completely before starting the practical tasks. In this section the tasks are outlined, in section 2, we describe the relevant algorithms, and in section 3, we document the resources that we expect you to use. Section 4 summarises with a precise to-do list.

1.1 Task One

Write a program that may be used to show the state of an AVL tree following a series of operations.

The program will provide a command driven user interface that enables the user to:

- Create a new AVL Tree (replacing any previous one).
- Insert a key into the current AVL Tree.
- Determine whether the tree contains an given key.
- Print a textual representation of the current AVL Tree on the screen.
- Write a textual representation of the current AVL Tree to a file.
- See a list of the available commands.
- Quit the program.

Sample I/O:

```
Commands: new, help, print, contains <key value>, insert <key value>, quit,
write <file name>.
>new
>insert 3
>insert 2
>insert 1
>insert 4
>contains 3
Yes
>insert 5
>print
               2(-1)
      1(0)                4(0)
                     3(0)      5(0)
>write AVL0.out
>quit
```

The `print` command displays the key value at each node followed by the balance factor in brackets.

The textual representation output to file by the `write` command should be the same as that written to the screen by `print` i.e. the same algorithm is used in both cases.

You should provide evidence of program function by carrying out the following steps:

A. Create an AVL tree, insert the keys 100, 150, 175, 125, 90, 99, 95 (in that order), and write a textual representation to a file called '`AVL0.out`'.

B. Starting with the tree constructed in A, insert the keys 87, 200, 110, 130, 135 (in that order), and write a textual representation to a file called ‘`AVL1.out`’.

You are expected to construct your program by completing the framework detailed in section 3.

1.2 Task Two

Write a program ***based on your solution to task one*** that uses an AVL tree to store a list of String items in alphabetical order.

Sample I/O:

```
Commands: new, help, print, contains <value>, insert <value>, quit,
write <file name>.
>new
>insert Alpha
>insert Beta
>insert Delta
>contains Beta
Yes
>print
            2(0)(Beta)
1(0)(Alpha)           4(0)(Delta)
>quit
```

The `insert` command will accept a String item as an argument. It will calculate a key consisting of the position in the alphabet of the first letter of the item. (Hence, as illustrated, ‘Alpha’ gains a key of 1, ‘Beta’ a key of 2, and ‘Delta’ a key of 4.)

The `print` command displays the key value at each node followed by the balance factor in brackets, followed by the item value in brackets.

The textual representation output to file by the `write` command should be the same as that written to the screen i.e. the same algorithm is used in both cases.

You should provide evidence of program function by carrying out the following steps:

A. Create an AVL tree.
B. Insert these items:
   nu, xi, omicron, pi, rho, sigma, tau, upsilon, phi, chi, psi, omega
C. Write a textual representation to a file called ‘AVL3.out’.

1.3 Task 3

Extend your solution to Task Two (Part I) so that items can be deleted from the tree.

Sample I/O:

```
Commands: new, help, print, contains <value>, insert <value>, delete
<value>, write <file name>, quit.
>new
>insert Alpha
```

```
>insert Beta
>insert Delta
>contains Beta
Yes
>print
            2(0)(Beta)
1(0)(Alpha)           4(0)(Delta)
>delete Delta
>print
            2(0)(Beta)
1(0)(Alpha)
```

The `delete` command will accept a String item as an argument. It will calculate a key consisting of the position in the alphabet of the first letter of the item. (Hence, as illustrated, 'Alpha' gains a key of 1, 'Beta' a key of 2, and 'Delta' a key of 4.)

If the item cannot be found in the tree, an error message should be printed.

You should provide evidence of program function by carrying out the following steps:

A. Create an AVL tree.
B. Insert these items:
   nu, xi, omicron, pi, rho, sigma, tau, upsilon, phi, chi, psi, omega
C. Write a textual representation to a file called 'AVL3.out'.
D. Delete nu and delete omicron.
E. Write a textual representation to a file called 'AVL4.out'.

1.4 Task Four

Based on your solution to Tasks 2 and 3, develop a new program that accommodates the following.

A. Modify the AVL tree classes to implement a dictionary, where items are grouped and stored by letter heading, and the number of items under a heading is also stored.
B. Implement a search command called 'find' that may be used to retrieve the data about any letter key.
C. Implement a delete command that removes specific words and updates a node's attributes. If all the words at a node are deleted then the node will be deleted and the tree structure appropriately updated.

Sample I/O:

```
>new
>insert Alpha
>insert Alpha
>insert Beta
>insert Bill
>insert Ball
>insert Delta
>insert Epsilon
```

```
>print
                                    (B)(3)
                                    Beta
                                    Bill
                                    Ball
            (A)(2)                                   (D)(1)
            Alpha                                    Delta
                                                              (E)(1)
                                                              Epsilon
>find B
(3)(Beta, Bill, Ball)
>find C
No entry found
>delete Bill
>find B
(2)(Beta, Ball)
>delete Gamma
```

To accommodate these changes you will need to begin by modifying the AVLTreeNode class:

- The first value stored in a node is the key (‘A’, …, ‘Z’).
- The second is the number of times a word with this key has been entered into the dictionary (‘Alpha’ has two entries in the example).
- The third part is a collection of word entries.

NOTE: With regard to the print and write commands, if you wish, you may modify your tree printing code, however, a version will be provided for you in due course.

Provide evidence of your program’s function:

A. Start with an empty AVL tree and insert the following:
   `alpha, beta, gamma, delta, epsilon, zeta, eta, theta, iota, kappa, lambda, mu, ka, sa, ta, na, ha, ma, ra, wa`
B. Write the resulting tree to file called ‘`AVL5.out`’.
C. Delete the following:
   `eta, theta, iota, kappa, lambda`
D. Write the resulting tree to a file called ‘`AVL6.out`’

2. Algorithms

We recommend first reading section 19.4 of the course text.

The key algorithms for this assignment is those required for '`insert`', and '`delete`'. We provide the outline of a recursive procedure for insertion (the simplest approach). You will need to do your own research for deletion.

2.1 Insert

When an item is inserted into an AVL tree, the balance of nodes can be destroyed, requiring that it be restored. Particularly, this restoration may involve selecting a new root node for the tree. Insertion then is an operation that accepts as a parameter, a root node and an item, and that returns a root node that may or may not the one that was passed in!

A pseudo code description of recursive insertion goes as follows:

```
def insert(node, key):
      if node is null:
            node = new node containing item.
      elsif key equals key at node:
            # Duplicate key, do nothing.
      elsif key less than key at node:
            # Insert left.
            node = insert(left child of node, key)
            # check for and repair imbalance
            if node imbalanced:
                  node = rebalanceLeft(node, key)
      else:
            # (key greater key at node.)
            # Insert right.
            node = insert(right child of node, key)
            # check for and repair imbalance
            if node imbalanced:
                  node = rebalanceRight(node, key)
      # Recalculate the node's height.
      set node height to max of children +1
      return node
```

We assume that an AVL tree node contains a key, a left sub tree (that may or may not be empty), a right sub tree (that may or may not be empty), and a value for the node height.

Checking for an imbalance is a case of comparing the heights of a node's children.

Repairing an imbalance is abstracted into two sub procedures: rebalanceLeft, and rebalanceRight.

Note: this is not the most efficient procedural abstraction. We've done it this way to manage complexity at each stage of the narration. Your implementation probably won't need rebalanceXXX sub routines.

2.1 Rebalancing

If an imbalance is detected, at a node N there are four possible causes (Weiss, section 19.4.1):

1. An insertion in the left sub tree of the left child of N.
2. An insertion in the right sub tree of the left child of N.
3. An insertion in the left sub tree of the right child of N.
4. An insertion in the right sub tree of the right child of N.

We must determine which case applies and then perform the appropriate single or double rotation.

Case 4 is a mirror image of case 1, as is case 2 of case 3. In our pseudo code, `rebalanceLeft` handles cases 1 and 2, `rebalanceRight` handles cases 3 and 4.

Here's pseudo code for `rebalanceLeft`:

```
def rebalanceLeft(node, key):
      if key < key of node’s left child:
            # Case 1 applies.
            return rotateWithLeftChild(node)

      else:
            # Case 2 applies.
            return doubleRotateWithLeftChild(node)
```

On detecting an imbalance, the procedure determines which rotation to perform based on whether the left child's left sub tree was altered (single rotation), or whether the left child's right sub tree was altered (double rotation).

2.2 Rotation

Single rotation is documented in Weiss section 19.4.2, and double rotation is documented in section 19.4.3. Java code is presented.

Here are pseudo code versions of two of the four methods:

```
def rotateWithLeftChild(node_k2):
      node_k1 = left child of node_k2
      left child of node_k2 = right child of node_k1
      right child of node_k1 = node_k2
      return node_k1

def doubleRotateWithLeftChild(node_k3):
      left child of node_k3 = rotateWithRightChild(left child of 
node_k3)
      return rotateWithLeftChild(node_k3)
```

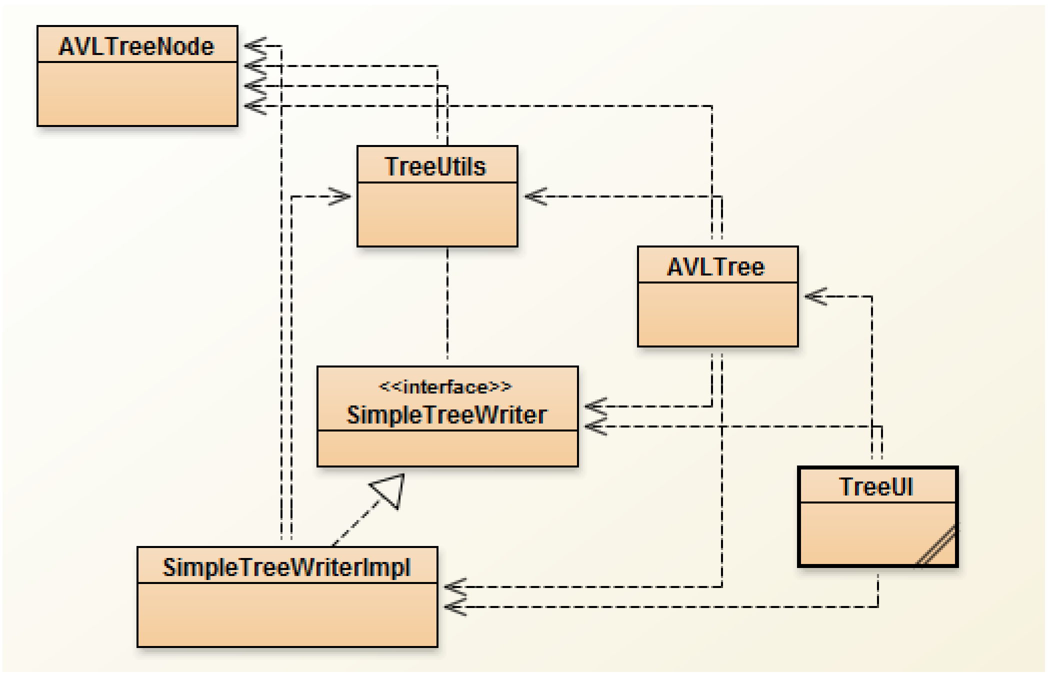
Something that is missing from this sketch (and the Java code of sections 19.4.2 and 19.4.3) is that the height values stored in the nodes have to be adjusted in light of the manipulations.

Since double rotate methods delegate to single rotate methods, only the code for the latter needs to be augmented.

Adjustment is not complicated, and what is required can be easily deduced from the code - perhaps with further assistance from the examples in section 19.4.2.

3. Resources

We have designed (but not completely implemented) a package of Java components that you must use to structure your work (you will find the materials on the Vula assignment page):



- The AVLTree and AVLTreeTreeNode classes implement an ordered AVL Tree ADT.
- SimpleTreeWriter describes a type of object that may be used to print an AVLTreeNode structure using a java.io.PrintStream object. (System.out is a PrintStream object and PrintStream objects can be created that print to Files.
- SimpleTreeWriterImpl is an implemention of SimpleTreeWriter. It provides a method that implements the printing algorithm sketched in section 2.2.
- TreeUtils provides methods supporting breadth first search (levelZero, nextLevel, isPlaceHolder), for searching an AVLTreeNode structure, and for insertion into an AVLTreeNode structure.
- TreeUI implements the command line interface (and contains the main method) for the program described in section 1 task 1.

3.1 TreeUI

The TreeUI class implements a text-based command interface and provides a main method for the program.

- It uses a scanner to break a command entered by the user down into keyword and optional argument.
- It maintains a Map of keywords to types of Command object.
- A Command object implements the actions associated with a command. It provides an 'execute' method and a 'help' method that returns information on the form of the associated command.

Here, for example, is the class declaration for the '`contains <key value>`' command:

```
private class Contains extends Command {
    public String help() { return "contains <key value>"; }

    public void execute(String argument) throws IllegalArgumentException {
        try {
            String response = target.contains(Integer.parseInt(argument)) ? "Yes" :
"No";
            System.out.println(response);
        }
        catch (NumberFormatException numFormE) {
            throw new IllegalArgumentException("Insert "+argument+" : argument not
an integer.");
        }
    }
}
```

If the user enters an instance of the contains command such as 'contains 4', the TreeUI slices the text into keyword 'contains' and argument '4', it uses the keyword to locate the instance of the Contains class in the Map of keywords to command objects, and then calls the object with 'execute('4').

Class declarations are provided for 'contains', 'help' and 'quit'. You are required to provide class declarations for 'new', 'insert', 'print', and 'write'.

4. The To-do List

To complete Task 1 (as outlined in section 1.1):

- Implement the TreeUtils ‘contains’, insert, rotate, and double rotate methods as per the algorithms in section 2. (The insert method must use the rotate methods.)
- Implement the Command classes in TreeUI that are required for the ‘insert’, ‘print’, ‘write’ and ‘new’ commands.
- Construct SimpleTreeWriterImpl by adapting your solution to assignment one. (Use AVLTreeNode toString() to obtain the label that you output for a node.)
- Provide evidence of program function as described in section 1.1.

To complete Task 2 (as outlined in section 1.2):

- Create a new version of the program constructed for task 1.
- Modify the AVLTreeNode, AVLTRee, TreeUtils, SimpleTreeWriterImpl, and TreeUI classes so that you can construct AVL trees that store an integer key and a string value at each node.
- Provide evidence of program function as described in section 1.2.

To complete Task 3 (as outlined in section 1.3):

- Following the design pattern for insert, create a delete method in AVLTree that uses a delete method in TreeUtils. The latter should have this signature:

```
public static AVLTreeNode delete(AVLTreeNode node, Integer key)
```

- Add a delete command to TreeUI.
- Provide evidence of program function as described in section 1.3.

To complete Task 4 (as outlined in section 1.4):

- Modify the AVLTreeNode class to incorporate the new fields.
- Following the design pattern for insert and delete, create a find method in AVLTree that uses one in AVLTreeUtils.
- Revise the TreeUtils delete method you created for Task 3.
- Add a find command to TreeUI.
- Provide evidence of program function as described in section 1.4.

If you complete these tasks you will gain 90% of the marks for the assignment. To obtain the final 10%, develop a version of the task 3 program that:

I. uses ITERATIVE insert and delete methods, and that
II. uses an AVLTreeNode class that stores the balance factor and NOT the height of a node.