1 Introduction

In this assignment, you will write a program that efficiently finds anagrams of a given English word. An anagram of a word is another word that uses exactly the same letters but in different order. For example, the word top has an anagram word pot.

Your main program should be called FindAnagrams, and should be invoked as:

```
FindAnagrams dictionary.txt words.txt
```

File dictionary.txt stores valid English words, one per line, and words.txt stores the words for which to find anagrams, one per line. Words in both dictionary.txt and words.txt are capitalized. The output should be written to file out.txt and should contain all anagrams found, one per line in the alphabetic order. Anagrams for different words should be separated by an empty line. For example, if words.txt contains words POT and LISTEN, the output file should be:

```
OPT
POT
ENLIST
INLETS
SILENT
TINSEL
```

The list of anagrams should not contain the original word. In the example above, the list for TOP does not contain the word TOP itself, and the list for LISTEN does not contain LISTEN.

You have to write all the code yourself. You cannot use code from the textbook, Internet, and any other sources. You may use java build-in class for linked list java.util.LinkedList and the Iterator interface java.util.Iterator.

2 Implementation

You are to implement the program efficiently using AVL trees with repeated keys allowed. First read all the words from the dictionary file, and insert them into an AVL tree, let us call it dictTree. For the key field, use the word with letters arranged in lexicographical (alphabetical) order. For the value field, use the word itself. For example, word LISTEN should be inserted into dictTree with key = EILNST and value = LISTEN.

Now dictTree stores words that are anagrams of each other with the same key. Given some word w, to find all its anagrams, first sort letters of w in alphabetical order. Let the word with sorted letters be wSorted. To find all anagrams of w, use method findAll(wSorted) of the AVLTree. Note that the anagram words are in the value field of the dictionary entries returned by findAll(wSorted).

To sort letters of the word efficiently, you can use one more AVLTree, let us call it temp. Insert each letter of a word as a string into temp. Then use inOrder traversal of temp to output letters alphabetically and concatenate them into a new string.
Similarly, to sort the found anagrams efficiently, use one more AVL tree. Insert all found anagrams into this tree and use inOrder traversal to list them in alphabetical order.

3 Classes Provided

3.1 AVLnode \(< K, V >\) implements Position \(< K, V >\)

This is a class for the AVL tree node. \(K\) is the generic type for the key of an entry, \(V\) is the generic type for the value of an entry. You have to use this class for nodes of your AVL tree. Review the use of Position interface in Lecture 3, slides 31-39. This class comes completely implemented, you cannot modify it in any manner. There are 5 private fields:

- parent reference to the parent node
- left reference to the left child
- right reference to the right child
- entry the entry stored at the node
- height the height of the node

There are 12 public methods that you can use:

- AVLnode(inputEntry,inputParent,inputLeft,inputRight) This is a constructor which creates a node with given entry,parent,left and right children. To create an external node, use
  new AVLNode(null,parentNode,null,null).
  Recall that external nodes do not hold entries in AVL trees, so pass null value for the entry. Also an external node does not have children, so we pass null for its left and right child. An external node usually has a parent, that is passed as an input parameter. The only case when an external node does not have a parent is when an external node is the root of the tree, i.e. the tree consists of only one node (the root).
- getHeight(): returns height of the node.
- getEntry(): returns the entry stored at the node.
- parent(),left(),right() return the parent node, the left child, and the right child, respectively
- setParent(newParent),setLeft(newLeft),setRight(newRight), setEntry(newEntry) change the parent node, the left child, the right child, and entry, respectively, to new values given in the input
- element() Returns element stored at the node.
- resetHeight() resets the height of the node, assuming that the heights of its left and right children are correct. You will need this method for the triNodeRestructuring
3.2 Comparator
This is the interface that class StringComparator has to follow.

3.3 IntegerComparator
Comparator for Integer class. You are not using this class directly, but my program Test requires it.

3.4 AVLTreeInterface< K, V >
This is the interface that your AVLtree class must implement. The description of the methods are found in section 4.

3.5 Test
This is a test program for your tree implementation. Compile and run Test.java with your tree implementation. It has 11 tests, and will tell you whether your implementation passes/fails these tests. First 10 tests are worth 4 points each, test 11 is worth 10 points. Test 11 is the most expensive since it checks if your tree is a valid AVL tree.

3.6 Position< K, V >
This is the interface AVLNode< K, V > class implements. You will also need it in the AVLTree class that you have to implement.

4 Classes to Implement
4.1 DictEntry< K, V >
This class represents an entry in the dictionary. For this class, you must implement all and only the following public methods:

- public DictEntry(K key, V value): A constructor which takes a key of type K and value of type V.
- public K key(): Returns the key in the DictEntry.
- public V value(): Returns the value in the DictEntry.
- public void changeValue(V newVal): changes the value to newVal.

You can implement any other methods that you want, but they must be declared as private methods.
4.2 AVLTree< K, V >

This class is for an AVL tree, implementing the provided AVLTreeInterface< K, V >. For the main AVL tree algorithms (remove and insert), you have to follow the algorithms as described in class.

For this class, you must implement all and only the following public methods:

- public AVLTree(Comparator< K > inputComparator): A constructor which initialises an empty AVL tree and sets the comparator object for the tree to the one provided in the input. A good choice for the “empty” tree is to consist of 1 external node which is the root of the tree. This can be done with new AVLnode(null, null, null, null).
- public boolean external(Position< K, V > p): returns true if p is external, false otherwise.
- public Position< K, V > left(Position< K, V > p): returns the position of the left child of p. If p does not have a left child, returns null.
- public Position< K, V > right(Position< K, V > p): returns the position of the right child of p. If p does not have a right child, returns null.
- public DictEntry< K, V > find(K key): returns the dictionary entry corresponding to the input key, or null if there is no entry with the input key.
- public Iterator< DictEntry< K, V >> findAll(K key): returns iterator over all dictionary entry with the input key. You can use java built-in LinkedList to first insert all entries with the input key and then return listIterator(). To implement this method efficiently, first find some node n storing the given key. Then search for key in the left and right subtrees of n, using recursion.
- public void insert(K key, V value): creates a new (key, value) entry and inserts it into the dictionary.
- public DictEntry< K, V > remove(K key) throws TreeException: if entry with key is present in the tree, removes and returns it. If key is not present, throws AVLTreeException.
- public Iterator< DictEntry< K, V >> inorder(): returns iterator over entries in the tree. The order should be sorted by increasing key, that is inorder traversal order.
- public void modifyValue(K key, V newValue) throws AVLTreeException: if entry with given key is present in the tree, changes the value field of this entry to newValue. If key is not present, throws AVLTreeException.
- public Position< K, V > giveRoot(): gives the position of the root node.
- boolean isEmpty() returns true if the tree does not have any entries, false otherwise.
- public int size(): returns the size of the tree.
- public int treeHeight(): returns the height of the tree.

You can implement any other methods that you want to in this class, but they must be declared as private methods. For some of the above listed public methods, you may feel like you really want to change their input/output parameters. This is not allowed. However, you can create a private method that does most of the required work and has the input/output parameters that you like. Then have the required public method that call your private method.
4.3 **StringComparator**

Comparator for String objects. Should use lexicographical order to compare strings. You can use Java’s build in method for Strings compareTo(), which implements lexicographical order. The StringComparator must implement the provided Comparator interface. It must have only 1 public method,

```java
public int compare(String a, String b). Method compare should return a negative integer if the first String is less than the second String, 0 if the two Strings are equal, and a positive integer if the first String is larger than the second String. It may be useful to look at the provided IntegerComparator when implementing StringComparator.
```

4.4 **AVLTreeException**

This is the class for AVL tree exception.

4.5 **FindAnagrams**

This is the main class for the program to find Anagrams.

5 **Using Comparator**

In your class AVLTree, you must have a private object of type Comparator< K >. Suppose you named it private Comparator< K > treeComparator;

The constructor for AVLTree takes as the input a comparator: AVLTree(Comparator< K > inputComparator).

In the constructor, you assign the inputComparator to the private treeComparator:

```java
treeComparator = inputComparator;
```

Then to compare the keys in the AVLTree, you invoke the Comparator method compare:

```java
treeComparator.compare(key1,key2).
```

See also lecture 4, slides 21-24 which explain comparators.

6 **Coding Style**

Your mark will be based partly on your coding style.

- Variable and method names should be chosen to reflect their purpose in the program.
- Comments, indenting, and whitespace should be used to improve readability.
- No variable declarations should appear outside methods (“instance variables”) unless they contain data which is to be maintained in the object from call to call. In other words, variables which are needed only inside methods, whose value does not have to be remembered until the next method call, should be declared inside those methods.
- All variables declared outside methods (“instance variables”) should be declared private (not protected) to maximize information hiding. Any access to the variables should be done with accessor methods (like key() and value() for the DictEntry).
7 AVL Tree Implementation vs. Binary Search Tree

If you cannot get the AVL tree to re-balance correctly, you can implement a binary search tree (in which case you should still have all the classes/method names as specified). The only difference between the BST and the AVL tree is that BST does not have to be balanced, you do not have to implement triNodeRestructuring for BST. You will lose 10 marks on Test 11 if you implement BST instead of AVL tree.

7.1 Grading

Your grade will be computed as follows.

- Program compiles, produces a meaningful output: 15 marks
- Test pass: 50 marks. First 10 tests are worth 4 marks, Test 11 is worth 10 marks.
- Coding Style: 20 marks
- findAnagrams program implementation: 15 marks.

8 Handing In Your Program

Hand in your program through the OWL system. Make sure you hand in all your .java files. Each file should be handed in separately, please do not zip your files. Also please do not put your code into a package.