Western University

CS 2210B - Data Structures and Algorithms
Fall 2021-22, Instructor: Dr. Saad Bin Ahmed

Assignment 4 (Programming)

Total Marks: 100
Due Date: March 31, 2022

1 Overview

In this assignment, you will do an investigation in the area of Natural Language Processing (NLP). At the the same time, you will practice the AVL tree data structure. You will need to use AVL trees for two steps of the assignment. In NLP, one often needs to count how many times each particular word occurs in a text (or collection of texts). Let r be the number of times a particular word occurs in the text. For example, consider the following text:

"I would like to eat an apple, a banana, and a peach. I already ate a banana."

The word "a" occurs 3 times, the word "an" occurs 1 time, etc. For language modeling, one often needs to know how many distinct words occur exactly r times in the text. Let N (r) be the number of distinct words that occur exactly r times. For example, consider the text above again. The word "I" occurs 2 times, the word "banana" occurs 2 times, the word "a" occurs 3 times. The remaining 10 words occur exactly once. Therefore in this case,

- N (3) = 1, since one word ("a") occurs 3 times in the text;
- N (2) = 2, since two words ("I","banana") occur 2 times in the text;
- N (1) = 10, since ten words ("would", "like", "to", "eat", "an", "apple", "and", "peach", "already", "ate") occur once in the text.

In NLP, N (r)'s need to be computed to build language models. It is also quite interesting to look at N (r)'s. What we find is that a few words, such as "a", "the", "and", are used very often in English (that is for large r, N (r) is small). However, about half of all words in text occur only once or twice (that is for small r, N (r) is very large). To explain this phenomenon, a "Principle of Least Effort" has been developed. This principle roughly states that people act so as to minimize their probable average rate of work. Both speaker and listener are trying to minimize their effort. The speaker's effort is conserved by having a small vocabulary of common words, and the listener's effort is lessened by having a large vocabulary of rarer words (the more there are words, the less ambiguous are messages, and it's easier to understand them). The maximally economical compromise between these competing needs is to have a few very common words and a large amount of rare words. In this assignment, you are to compute efficiently N (r)'s from a given text. Your main program will be called 'calculateNr', and should read the text file specified by a command line argument: 'CalculateNr text.txt'. It should output N (r)s separately on each line. For our example text above, the program's output should list all N (r)'s sorted by the increasing value of r:

$N(1) = 10$
$N(2) = 2$

$N(3) = 1$

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 program very efficiently using the AVL tree data structure. First read all the words from the input file, using the code provided in `FileWordRead.java`. As you read the words, insert them in your AVL tree, let's call this tree T_1. The word itself will be the key. The value will be an integer, which specifies how often the word occurs in the input file. The first time you see a word, say word 'cat' you should create an entry ('cat',1) and insert it into your AVL tree. The second time you see word 'cat', you should update the count of the entry ('cat',1') to be ('cat',2'). In our example text: *'I would like to eat an apple, a banana, and a peach. I already ate a banana.",* after you finish reading all the words, the AVL tree T_1 should contain the following entries:

```
(I,2), (would,1), (like,1), (to,1), (eat,1), (an,3),
(apple,1), (banana,2), (and,1), (peach,1), (already,1),
(ate,1).
```

After this first step, you know how often each word occurs in the input file. Now you need to count how many words occur exactly 1 time, 2 times, etc. The efficient way of doing this is with another AVL tree. You construct a new AVL tree, let's call it T_2. The entries in T_2 are different from entries in T_1, since the purpose of T_2 is different. In T_2, the key is the word rate r. Notice that r is in the value field of the first tree T_1. The value field of an entry in T_2 is the count of how many words have the rate r.

Now we are ready to construct T_2 from T_1. Delete entries from T_1 until T_1 is empty. Recall that each entry in T_1 is of form ("word", r), where r is how often the "word" occurs in the text file. For each deleted entry ("word", r), if the r is not present in T_2 yet, insert a new $(r, 1)$ entry in T_2. If key r is already present in T_2, say we have an entry $(r, count)$ in T_2, update this entry to $(r, count + 1)$, since you found one more word that occurs exactly r times in the input file. After you are done inserting entries into T_2, for any entry (r, c) in T_2 c is the count of how many words occur in text exactly r times. For our example text above:

"I would like to eat an apple, a banana, and a peach. I already ate a banana."

after you are done reinserting entries from T1 into T2, T2 should contain the following entries:

```
(1,10), (2,2), (3,1).
```

When you are done inserting into T_2, simply go over the tree in inorder traversal and print out the key and value fields to the standard output. Notice that the keys in T_1 are of type String, but the keys in T_2 are of type Integer. We will use Java Generics for both the keys and the value fields of the Entry Class.

3 Classes Provided

3.1 FileWordRead

This is a program for reading words from file. I wanted to make sure you read exactly the same words from file as my test program, so you have to use this program. You need the following methods from this class:

- `FileWordRead(String name)`: This is a constructor which takes the name of the file to read tokens from.
- `public Iterator<String> getIterator()`: Returns iterator over words read from the input file. Here is how to use the iterator:

```
FileWordRead words = new FileWordRead(fileName);
Iterator<String> it = words.getIterator(); // grab the iterator into variable it
while (it.hasNext()) { // Check if anything is left in the iterator
String next = it.next(); // get the next item in the iterator
....
```

3.2 AVLTree

This class implements various AVL operations and is a main supporting file to start with. It has provided implementation details that would be helpful in presenting the solution. The description of the methods are found in section 4. The provided methods are mentioned as follows,

- /* Function to check if tree is empty */
  public boolean isEmpty()
- /* Make the tree logically empty */
  public void makeEmpty()
- /* Function to insert data */
  public void insert(int data)
- /* Function to get height of node */
  private int height(AVLNode t)
- /* Function to max of left/right node */
  private int max(int lhs, int rhs)
- /* Function to insert data recursively */
  private AVLNode insert(int x, AVLNode t)
- /* Rotate binary tree node with left child */
  private AVLNode rotateWithLeftChild(AVLNode k2)
- /* Rotate binary tree node with right child */
  private AVLNode rotateWithRightChild(AVLNode k1)

- /* Double rotate binary tree node: first left child * with its right child; then node k3 with new left child */

  private AVLNode doubleWithLeftChild(AVLNode k3)

- /* Double rotate binary tree node: first right child * with its left child; then node k1 with new right child */

  private AVLNode doubleWithRightChild(AVLNode k1)

- /* Functions to count number of nodes */

  public int countNodes()

- /* Functions to search for an element */

  public boolean search(int val)

- /* Function for inorder traversal */

  public void inorder()

- /* Function for preorder traversal */

  public void preorder()

- /* Function for postorder traversal */

  public void postorder()

3.3 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, it's most expensive since it checks if your tree is a valid AVL tree.

3.4 AVLNode

The AVL Node class implements the structure of node. You will also need it in the AVLTree class whose partial implementation is provided to you. The detail implementation of AVL Node is also provided in the file mentioned in section 3.2.

4 Classes to Implement

With reference to provided code, assembly of code is required to create a logical output as elicited in Test.java.

4.1 AVLTree

This class will extend the implementation of provided AVLTree class. For this class, you must implement all and only the following `public` methods in addition to the methods already implemented as provided in the AVLTree.java:

- `public isBST(t1):` This method will return true if the provided tree is BST.
- `public isBalanced(t1):` This method will return true if given BST tree is also AVL.

You can implement any other methods that you want to in this class, but they must be declared as private methods.

4.2 AVLTreeException

This is the class for AVL tree exception.

4.3 calculateNr

This class contains the main method, declared with the usual method header: `public static void main(String[] args)` You can implement any other methods that you want in this class, but they must be declared as `private` methods.

In your class `AVLTree`, you must have a `private` object of type

5 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.

6 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 loose 10 marks on Test 11 if you implement BST instead of AVL tree.

6.1 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
- `calculateNr` program implementation: 15 marks.

7 Important Note

- To be eligible for full marks, your programming assignment must run on the departmental computing equipment. You may develop assignments on your home computer, but you must allow for the amount of time it will take to get the final programs working on Computer Science's machines. All programming assignments must be submitted through OWL.
- The late penalty for programming assignment is [2.5i] (2.5 to the i-th power, rounded to the nearest integer), where i $>$ 0 is the number of days you are late. So if you hand in your assignment 1 day late, you will be penalized 3%, a delay of 2 days will decrease your grade by 6%, 3 days is penalized 16% and 4 days takes 39% off your grade. You cannot be more than 4 days late.