ASSIGNMENT THREE: Binary Search Trees and Serialization

Description

You will be provided an ADT for a Binary Search Tree (BSTreeADT.java) to be implemented as a Binary Search Tree (BST). The implementation of the BST will be used in an application called WordTracker that reads text files and collects and stores all the unique words it finds in those files. The BST will be able to store information from multiple text files. It will also keep track of each occurrence of a word in a file and the line on which it was found in that file. This information will be stored in a list for each word object stored in the BST. The program will also produce output, specified by the user at command line, to generate reports using a variety of iterators built into the BST.

Notes

- This is your third assignment and is worth **5.7%** of your final grade in this course.
- You are required to complete this assignment with your assigned group.
- No late assignments will be accepted.
- Below you will find the specifications for your application, along with some hints on how to proceed.
- Read the specifications very carefully. If you are uncertain about any of the requirements, discuss it with your instructor.

Specifications

1. Write a reference-based implementation **(BSTree.java, BSTreeNode.java)** using the instructor-provided BSTreeADT.java and Iterator.java interfaces. JUnit should be used to test the implementation of the Binary Search Tree.

2. Write a cross-reference program **(WordTracker.java)**, which constructs a binary search tree with all words included from a text file (supplied at the command line) and records the line numbers on which these words were used. The line numbers should be stored with the file names, which in turn are associated with the nodes of the tree.

3. Using Java serialization techniques, store the tree in a binary file **(repository.ser)**. Make sure you insert the class version UID to ensure the backward compatibility with your repository should the class specification change with future enhancements.

4. Every time the program executes, it should check if the binary file **(repository.ser)** exists, and if so, restores the words tree. The results of the scanning the next file are to be inserted in the appropriate nodes of the tree. Therefore, **repository.ser** will contain all words occurred in all files scanned with the meta information about those word locations.

5. The user should be able to run the program via the command line as follows:

```
java -jar WordTracker.jar <input.txt> -pf/-pl/-po [-f <output.txt>]
```

a) <input.txt> is the path and filename of the text file to be processed by the WordTracker program.
b) 3 mutually exclusive options at command line:
   - **-pf** to print in alphabetic order all words along with the corresponding list of files in which the words occur.
   - **-pl** to print in alphabetic order all words along with the corresponding list of files and numbers of the lines in which the word occur.
   - **-po** to print in alphabetic order all words along with the corresponding list of files, numbers of the lines in which the word occur and the frequency of occurrence of the words.
c) Optional argument to redirect of the report in the previous step to the path and filename specified in <output.txt.
d) You can assume that the user will provide these arguments in the order shown.

Submission Deliverables

Assignment zip file will be uploaded into **D2L** by the specified due date and time, also, named as the assignment number and your group number – i.e. A1Group3.zip.

The assignment zip file will include the following:

a. An executable Java Archive file (.jar) for your sort application called **WordTracker.jar**.
b. A readMe.txt file with instructions on how to install and use the Word Tracker program.
c. The project should have completed javadoc using **"-private"** option when generated, with output placed in the doc directory of the project.
d. A folder containing the complete Eclipse project directory.
   - At the root of the project directory, include a readMe.txt to describe the completeness of the assignment (as a percentage) and a list of known deficiencies and/or missing functionalities.

Submission Criteria

Your program must execute. Code that doesn't run/compile will not be accepted.

No late Assignments will be accepted and will be assigned a mark of 0%.

You are expected not merely to solve the problems, but to make intelligent decisions about how to approach the problems. Remember, it is not enough to submit adequately coded programs that merely produce the expected output, but should be elegant, designed legibility, have maintainability, and other such important factors.

Please note that proficiency in using official Java API Doc can answer a lot of questions you may have about using some standard Java classes and methods, and many other types of programming which you will encounter, both at SAIT, and beyond.

Criteria for Marking Assignment

Binary Search Tree implementation		/	**4**
Proper implementation of Serialization and tree persisted		/	**2**
Proper reconstruction of BST from repository file		/	**2**
Proper reading of files and stripping of punctuation		/	**3**
-pf flag generates proper output		/	**1**
-pl flag generates proper output		/	**1**
-po flag generates proper output		/	**1**
-f flag to create output file works properly		/	**1**
Complete set of JUnit tests		/	**3**
Readmes/Instructions and javadoc documentation		/	**2**
Total:		**/**	**20**

Final Individual Grade:

Team Total (20)	*	Peer Evaluation Multiplier (as a percentage)	=	Subtotal (20)	Final Score
	*		=		
					+
Peer Evaluations Completed (3)					
					=
				Final Grade (23)	