CS1027: Foundations of Computer Science II
Assignment 4
Due: April 9, 11:55pm.

Purpose

To gain experience with

- Developing and using a binary tree
- Using different data structures (arrays, binary trees, linked lists) in one application

1. Introduction

The Ontario Scientific London Library (LL) requires a new program where users can search for words in a large collection *C* of documents stored in text files. More specifically, a user would like to know the files, and the location in these files, where a particular word exists. For this assignment, you will design and implement a program in Java to be able to efficiently search for words in a collection *C* of text documents. The collection of words contained in the text documents is called a lexicon.

The program receives as input a file *I* containing a list of words to search in the lexicon. You will be provided with code that will read the files in collection C and it will store the words contained in these files in a lexicon data structure that holds location information about the different words. This data structure is an array in which every entry stores a binary search tree (binary search trees are described below; also, read the lecture notes on binary search trees). Every node of a binary search tree stores a word *w* and a list of files where the word appears; for each file the data structure also stores a list of positions within this file where the word *w* appears. More details of the data structure are given below.

We consider that the same word may be contained in more than one file. Once the program has stored the words in the lexicon data structure, the input file *I* will be read. Each word in *I* will be searched in the lexicon and information about the files that contain the word and the positions of the word in the files will be printed. Some sample output is given below.

2. The Lexicon Data Structure

Figure 1 below depicts the lexicon data structure. The components of the lexicon data structure are described below.

- An array called *table* of length 1031 is used to store a set of binary search trees. Provided class *HashTable.java* contains the declaration of this array and methods to add and read information from it. A method that you will need to use from this class is *computeIndex*(*String word*) that returns the index of *table* where the given parameter *word* needs to be stored. So, for example, if word *word_x* needs to be stored in *table*, it will be stored in the binary search tree referenced by *table*[*computeIndex*(*word_x*)].
- A *binary tree* is a tree that is either
  - empty, or
  - it is a tree with only one node, or
  - each node of the tree has at most two children (a left child, or a right child, or both).
- A *binary search tree* is a binary tree that has the following properties:
  - The root of the tree stores a word that is lexicographically larger than the word stored in its left child (if any) and it is lexicographically smaller than or equal to the word stored in its right child (if any), and

- the left and right subtrees are binary search trees

Lexicographic order is the usual alphabetical order so, for example, the word "car" is smaller than "computer" and "computer" is smaller than "tree".

- The binary search trees in the lexicon data structure are implemented by the *BinSearchTree* class, that you must write. A binary search tree is composed of binary search tree nodes implemented in the provided *BinSearchTreeNode* class. Each node in the tree has several instance variables:
  - *BinSearchTreeNode left* is a reference to the left child of this node
  - *BinSearchTreeNode right* is a reference to the right child of this node
  - *String word* is the word stored in this node, and
  - *LinkedList files* is a reference to a singly linked list containing the names of the files where *word* is found.
- Each *LinkedList* in a *BinSearchTreeNode* object is a singly linked list composed of nodes of type *FileNode*. Class *LinkedList.java* is provided to you.
- Each *FileNode* object has the following instance variables:
  - *String filename* stores the name of a file where a word is found
  - *FileNode next* is a reference to the next node in the list (i.e. the next file containing a given word)
  - *ArrayList<Integer> positions* is a reference to a java *ArrayList* storing integers that denote the positions where a word is found in the corresponding file. Class *ArrayList.java* is part of the standard java libraries and you can use it in your program by adding the import statement: *import java.util.ArrayList*. Read the java documentation to learn about the methods provided by this class.

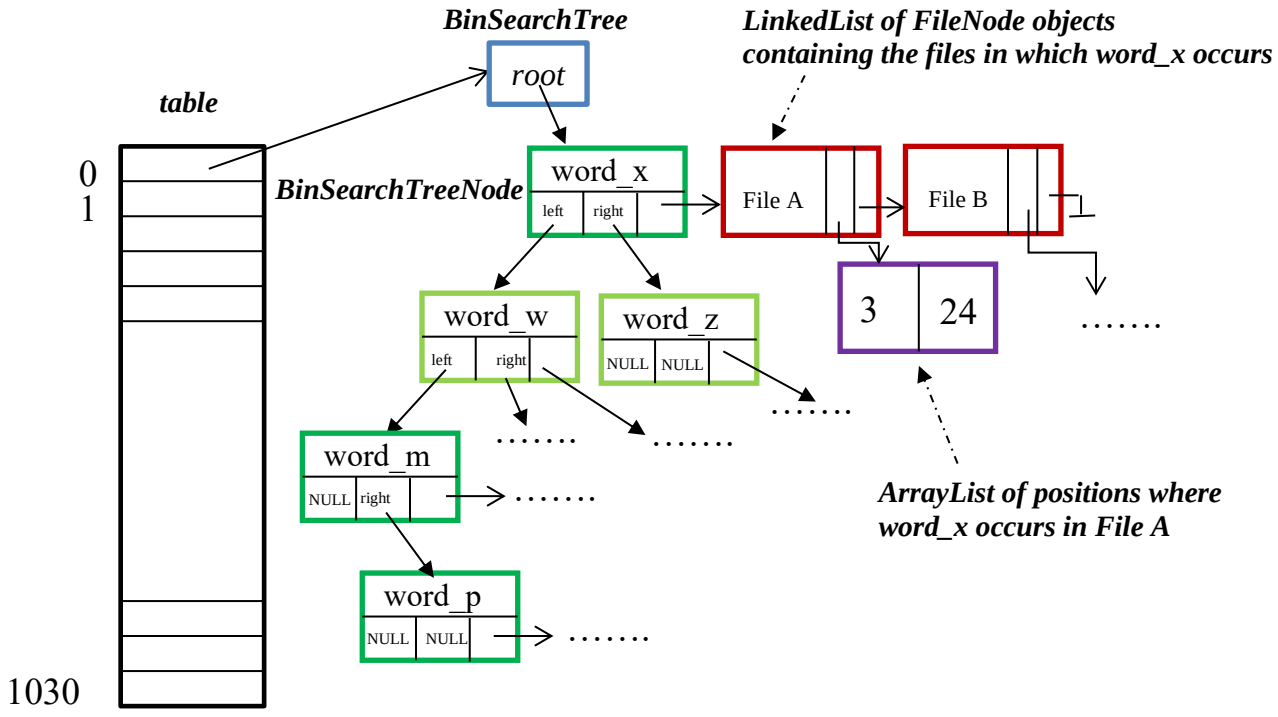

Figure 1. Schematic view of the lexicon data structure

Note in Figure 1 that:

- We assume that *computeIndex*(*word_x*) = *computeIndex*(*word_w*) = *computeIndex*(*word_z*) = *computeIndex*(*word_m*) = *computeIndex*(*word_p*) = *0*
- *word_x* is found in *File A* and in *File B*
- *word_x* is found in *File A* in positions *3* and *24*
- *word_x* is lexicographically larger then *word_w*, and smaller than *word_z*
- *word_w* is lexicographically larger than *word_m*, and *word_m* is lexicographically smaller than *word_p*

3. Classes to Implement

A description of the classes that you need to implement in this assignment is given below. You can implement more classes, if you want. You **cannot** use any static instance variables.

3.1 BinSearchTree.java

This class uses a linked structure to implement a binary search tree. The nodes of this tree are of the provided class *BinSearchTreeNode*. This class will have an instance variable

private BinSearchTreeNode root;

that is a reference to the root node of the binary search tree. You need to implement these methods:

- public *BinSearchTreeNode getWord*(String *searchWord*). Returns the node of the binary search tree storing *searchWord*, or null if no node stores it. We give below pseudocode for an algorithm for looking for a given word in a binary search tree.

  Algorithm search (*r*, *searchWord*)
  Input: Root *r* of a binary search tree and a word
  Output: node storing *searchWord* or null if no node stores *searchWord*

  if *r* is null **then return** null //It is an empty tree.
  else if word stored in *r* is equal to *searchWord* **then return** *r*
  else if *searchWord* < word stored in *r* **then** //*searchWord* should be in the left subtree
  return search(left child of *r*, *searchWord*)
  else return search (right child of *r*, *searchWord*) //*searchWord* should be in right subtree

  If you implement this algorithm you will need to write a private auxiliary method and invoke it from your implementation of method *getWord*.

- public void *insertWord*(String *theWord*, String *theFileName*, int *thePosition*). This method first looks for *theWord* in the binary search tree:
  - If some node *p* stores *theWord*, then *theFileName* and *thePosition* are added to the information stored in *p* as follows:
    - use method *getFiles* from class *BinTreeSearchNode* to get the list *Lp* of *FileNode* objects stored in node *p*
    - use method *insertWord* from class *LinkedList* to add *theFileName* and *thePosition* to *Lp*.
  - If no node in the binary search tree stores *theWord* then a new node of type *BinSearchTreeNode* storing *theWord*, *theFileName* and *thePosition* is added to the binary search tree.

We give below pseudocode for an algorithm to add to a binary search tree a new node storing a given word, file name and position of the word in the file.

Algorithm insert (*r*, *theWord*, *theFile*, *thePosition*)
Input: Root *r* of a binary search tree, word, file name, and position
Output: Add a new node to the binary search tree storing *theWord*, *theFile*, and *thePosition*

if *r* is null **then** //It is an empty tree. Add a new node as the only node of the tree.
    Create a new node storing *theWord*, *theFile*, and *thePosition* and set it as the root of the binary search tree
else if *theWord* < word stored in *r* **then** // Add new node on the left subtree of *r*
    if the left child of *r* is null **then**
        Create a new node storing *theWord*, *theFile*, and *thePosition* and set it as the left child of *r*
    else insert (left child of *r*, *theWord*, *theFile*, *thePosition*)
else if the right child of *r* is null **then**
        Create a new node storing *theWord*, *theFile*, and *thePosition* and set it as the right child of *r*
    else insert (right child of *r*, *theWord*, *theFile*, *thePosition*)

You will probably need to add a private auxiliary method that implements the above algorithm and invoke it from your implementation of method *insertWord*.

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

3.2 Searcher.java

This class has two instance variables:

- private *HashTable table*, this is a reference to the table of binary search trees that implements the lexicon data structure
- private String *inputFile*, this is the name of the input file storing all words that will be searched for in the lexicon data structure

You need to implement these methods in this class:

- public void *findAllWords*()
  This method reads the input file and for each word *w* in it invokes below method *findWord* to look for the word *w* in the lexicon and print information about where the word appears in the collection of text files that compose the lexicon.
- public void *findWord*(*String searchWord*)
  This method looks for *searchWord* in the lexicon in the following manner:
  - First invoke method *computeIndex* from class *HashTable* to determine the index *j* of the entry of *table* that might contain *searchWord*.
  - If *searchWord* is not in the binary search tree stored in *table*[*j*] then invoke method *CustomPrinter.wordNotFound*(*searchWord*, *inputFile*) that will print on the screen a message indicating that the word is not in the lexicon; this method will also write in an output file a message indicating that the word was not found in the lexicon. Note that if *searchWord* is not in the binary search tree referenced by *table*[*j*], it cannot be stored in any other binary search tree, so you should not look for it in the other binary search trees stored in *table*.

- Otherwise, if *searchWord* was found in some node *p* of the binary search tree referenced by *table*[*j*] then do the following
  - First, invoke method *CustomPrinter.wordFound*(*searchWord, inputFile*) that will print on the screen a message indicating that searchWord is in the lexicon; this method will also write in an output file a message indicating that the word was found in the lexicon.
  - Get the *LinkedList Lp* of *FileNode* objects stored in node *p* using method *getFiles*. Traverse linked list *Lp* and for each node *q* in the list get the *ArrayList ℓq* of positions stored in it using method *getPositions* and then invoke method *CustomPrinter.printPositionsPerFileFound*(*filename, ℓq, inputFile*) to print the list of positions where *searchWord* was found, where *filename* is the name of the file stored in node *q*.

You can write more methods in this class if you want to, but they must be declared as private.

4. Input File, Collection of Text Files, and Sample Output

You are given several text files that will be used to populate the lexicon; these files are contained in the *input* folder of the zip file that you can download from the course's webpage. The input file for your program is a text file containing a list of words separated by spaces. Six sample input files (*test1.txt – test5.txt*, and *wordsToBeFound.txt*) are included in the files provided for this. The expected output for each input file is also given (e.g. *output_test1.txt, output_wordsToBeFound.txt*).

The output that your program must produce for the provided input file *wordsToBeFound.txt* and the provided text documents in folder *input* is the following:

```
Word john does not exist.

Word to be found: highly
        Filename: The_Souls_Of Black_Folk.txt
                [43692]
        Filename: Pride_and_Prejudice.txt
                [1918, 5368, 14414, 19117, 20004, 22855, 24309, 25116, 25484, 26021, 28871, 34519,
                 34541, 36809, 39066, 39179, 40012, 41310, 66961, 68739, 74766, 77738, 83214,
                 83538, 85349, 98797, 113956]
        Filename: Frankenstein.txt
                [6543, 26184, 72421]
        Filename: A_Modest_Proposal.txt
                [1604]

Word bye does not exist.

Word to be found: short
        Filename: The_Souls_Of Black_Folk.txt
                [62324, 68700]
        Filename: Pride_and_Prejudice.txt
                [4643, 10606, 10884, 12479, 13404, 19735, 23988, 24610, 32462, 32793, 33393,
                 37239, 38852, 49228, 56042, 56681, 59996, 68133, 75488, 80128, 83898, 85977,
                 94570, 96800, 101303, 106500, 107707, 110468, 112001, 112256, 115314, 116597]
        Filename: Frankenstein.txt
                [11763, 16499, 17305, 19502, 21854, 29228, 29766, 34606, 36669, 46174, 53659,
                 56947, 58399, 59666, 65252, 66168, 70352]
```

5. Hints and How to Run the Program

To read a text file, you can use class BufferedReader:

```
BufferedReader in = new BufferedReader(new FileReader(inputFile));
String line = in.readLine();
```

The above code fragment opens *inputFile* and reads the first line into String variable line. When method readLine() returns null the end of the file has been reached. To get the words stored in a line of text that was read from the file, you can either use method *split* from the java class String or you can use java class *StringTokenizer*. Please read the java documentation to learn how to use method *split* or class *StringTokenizer*.

The main method for the program is in the provided class Lexicon.java. Hence, to run the program from the console you need to compile your java classes and then run

```
java Lexicon inputFile
```

where *inputFile* is the name of the input file. To run the program form Eclipse you need to specify the input file in Run->Run Configurations…

The collection of text files that will be used to construct the lexicon must be stored inside a folder called *input*. Hence, if you use Eclipse, you will put the input file in the same folder where your .java files are and you will create the *input* folder inside the folder that contains your *src* or *(default package)* folder. If the files are not found by Eclipse, you need to move them around until it finds them, as not all versions of Eclipse behave in the same manner.

6. Classes Provided

For this assignment you can download from the course's webpage several java classes. You are encouraged to study the given code to you learn how it works. Below is a description of these classes.

6.1 Class *BinSearchTreeNode*.java

This class represents a node of the binary search tree. Each node has the following four instance variables:

- `left`: points to the left child of this node (also a *BinSearchTreeNode* object)
- `right`: points to the right child of this node (also a *BinSearchTreeNode* object)
- `word`: the word stored in the node
- `files`: points to the *LinkedList* object storing a list of nodes of type *FileNode*

This class also has a constructor and getter, and setter methods.

6.2 Class *CustomPrinter*.java

This class contains methods to print custom messages for the output. More specifically:

- `wordNotFound(String word)` prints a custom message when a word is not in any of the input files.
- `wordFound(String word)` prints a custom message when a word is found.
- `printPositionsPerFileFound(String filename, List<Integer> positions)` prints the file name and positions of a word in that file. The parameter `filename` is the name of the file containing the word, and the parameter `positions` is the list of positions where the word is found in the file.

6.3 Class *FileNode*.java

This class represents a node in the singly linked list containing the files and the positions in each file where a word is found. Each node has three instance variables:

- `fileName`: the name of the file where the word is found
- `next`: points to the next node in the linked list containing this node
- `positions`: it is a pointer to a java *ArrayList<Integer>* which contains the list of positions in the file where the word is found

This class has a constructor and getter and setter methods.

6.4 Class *LinkedList*.java

This class implements a singly linked list of *FileNode* objects. Note that the class implements the *Iterable<FileNode>* interface, so you can use an iterator (see *LinkedIterator.java*) to easily traverse the linked list. Note method `iterator`() which returns a new iterator, and the use of the iterator for traversing the linked list in method `insertWord(String filename, int position)`.
This class has two instance variables `head` and `tail` that point to the first and last nodes of the linked list, respectively. The class also provides methods `getHead` and `getTail` to read the values of these instance variables.

6.5 Class *HashTable*.java

This class represents the array that stores the binary search trees. The class has one instance variable, `table`, which is an array of length 1031 whose entries are references to *BinSearchTree* objects.
The class has these methods:

- `computeHash(String word)` determines the index of `table` storing the binary search tree where `word` needs to be stored.
- `addWord(String word, String filename, int position)` adds a word, file name, and position to the appropriate binary search tree in `table`.

6.6 Class *InputFileReader*.java

The methods in this class read all text files in the *input* folder and add all their words to the lexicon.
This class has the following method:

- `readAllFiles(HashTable table)` reads all files in the *input* folder and adds all of their words to the binary search trees in `table`.

6.7 Class *Lexicon*.java

This class contains the main method. It initializes the table of binary search trees, reads the text files and adds their words to the table. Then, the words in the input file are searched and the results of the searches are printed in the console and saved in an output file. The program takes one argument in its main method: the name of the file that contains the words to be searched (e.g. *wordsToBeFound*.txt*).*

6.8 Class *LinkedIterator*.java

This class provides a glimpse of Java's utilities to traverse collections. Iterators are abstract data types in which a programmer can "hide" and encapsulate custom logic for traversing a collection of objects through methods *hasNext*() and *next*(). See file *LinkedIterator.java* for the definition of an iterator to

traverse the linked list of the *FileNode* objects. An example of an iterator being used is also given in class *LinkedList.java* (see the method `insertWord(String filename, int position)`).

6.9 Class *Comparator*.java

This class has a main method that receives as parameter the name of an input file. This method will compare the output produced by your program (for example, output_test1.txt) with the expected correct file (for example, correct_output_test1.txt, provided to you).

7. Submission

Submit all your .java files to OWL. **Do not** put the code inline in the textbox. **Do not** submit your *.class* files. If you do this and do not submit your *.java* files your program cannot be marked. **Do not** submit a compressed file with your java classes (.zip, .rar, .gzip, …). Do not put a "package" command at the top of your java classes.

8. Non-functional Specifications

1. **Assignments are to be done individually and must be your own work.** Software will be used to detect cheating.
2. Include comments in your code in **javadoc** format. Add javadoc comments at the beginning of your classes indicating who the author of the code is and a giving a brief description of the class. Add javadoc comments to the methods and instance variables. Read information about javadoc in the second lab for this course.
3. Add comments to explain the meaning of potentially confusing and/or major parts of your code.
4. Use Java coding conventions and good programming techniques. Read the notes about comments, coding conventions and good programming techniques in the first assignment.

9. What You Will Be Marked On

1. Functional specifications:
    - Does the program behave according to specifications? Does it run with the test input files provided? Are your classes created properly? Are you using appropriate data structures? Is the output according to specifications?
2. Non-functional specifications: as described above