University of Illinois @ Urbana-Champaign

CI 487: Data Structures for CS Teachers

Application 2: Tree Dictionary

Objectives and Overview

Application assignments are intended to be an opportunity to take a step back use some of the data structures you have been using. It is often easy to get so bogged down in the details of creating and implementing data structures and not have the opportunity to use them in performing useful tasks. In this exercise you will be using an AVL tree similar to the implementation of the one you implemented last week. You will be using this data structure to implement a simple dictionary lookup engine. In doing so it has three objectives:

1. To introduce you to a situation where storing data in the form of a tree is useful and to use that situation as a method of practicing using trees.
2. To give you the opportunity to write several classes largely from scratch.

With respect to the last point, the starter files you will be presented for this assignment are *much* more spare than those you have been given for past assignments. The code needed to implement these methods will require far less code than previous assignments and will primarily focus on translating the more abstract problem definition into a working program. As such, you are granted a great deal of freedom in how you complete this assignment and it's core functionality.

1 Structures and Specifications

The structure of this assignment will be spread over five separate file:

1. AVL.java - The provided AVL tree that you will use to implement your dictionary. This class should be read through but remain unmodified.
2. DictEntry.java - This will be fundamental unit in our dictionary. Each of these classes will contain some attributes for holding words and definitions, a compare to method for comparing instances of DictEntry, and a toString methods for printing them.
3. Dictionary.java - This class will hold the AVL tree of DictEntry objects. It will contain a constructor that populates the dictionary with an initial set of works from a file in addition to two methods that allow for searching and insertion into the tree.
4. Main.java - This will be the class that contains the main method where you will implement the user interactivity with the dictionary.
5. dict.csv - This is the file of initial comma separated files containing an initial set of words and definitions. Your Main class will use this to create a Dictionary with an initial set of words.

1.1 Step 0: Reading through AVL.java

This file contains a correct implementation of the AVL class. You are encouraged to read through the code in the file prior to proceeding. Specifically, the following methods:

1. `public void insert(T data)`: This method inserts some data of generic type into the AVL tree.
2. `public TreeNode<T> search(T data)`: This method finds and returns a node containing some data of generic type T.

Each of these methods will be useful when you implement the `Dictionary`.

1.2 Step 1: Implementing DictEntry

The first thing we will need for our dictionary tree is a class that stores a word and it's definitions. This is what the `DictEntry` class will be used for. Use the following specifications to implement the class such that it contains the attributes and methods we will require in future portions of the assignment.

Attributes

The class should have two `String` attributes: 1) `word` and 2) `definition` each of these attributes should be `private final` as we don't want the user of this class to have unmoderated access to the attributes and we don't want them being changed after they are initially set.

Methods

The following methods should be implemented for this class:

1. Create a constructor for `DictEntry` that is used to set the `word` and `definition` attributes when a new instance of the class is created.
2. Create a getter for each of the attributes
3. `compareTo`: This methods should override the compareTo method in order to allow for two `DictEntry` classes to be compared.
4. `toString`: This method should return a string containing the word and it's definition.

You will be implementing these methods from scratch in this assignment so be sure to look back on your previous assignments for examples.

1.3 Step 2: Implementing Dictionary

This class has two core components:

- It should have a constructor that takes a String representing a file path as a parameter, read that file, and use its contents to populate the AVL tree.
- It should have methods that allow for adding `DictEntry`s to the tree and searching the tree.

As such, this section will be organized such that the first part will be a walkthrough on how to read a CSV file with clues on how you can use the data from that file to construct the initial dictionary. The second part will be details on the methods you will implement but, in keeping with the theme of the assignment, you will be implementing these methods yourself with reduced skeleton code.

1.3.1 Implementing the Constructor

Provided to you is a constructor which takes a single parameter, a string representing a file path. That file path should be to a file containing comma separated values. The structure of the CSV format that this method should support is a series of lines of the format: `word,definition`. This walkthrough will cover how to: 1) open and iterate over the file and 2) split each line to extract the word and the definition.

Creating a File Object: Reading the file begins with creating a new instance of a `File` object, as shown below.

```
File infile = new File(fp);
```

As you can see this file object should take

Using File + Scanner to Read File: The `Scanner` class is fundamentally a class that is used to read from input streams. In the past, and in this assignment, you primarily use it to read from the `System.in` (i.e., your keyboard). However, you can also provide it other input streams, such as the file object you just created, as seen below:

```
try{
    fileReader = new Scanner(infile)
} catch (FileNotFoundException e){
    System.out.println("File not found");
    return;
}
```

Reading the file with a Scanner can produce a `FileNotFoundException` which Java requires us to catch.

Reading a File: The following loop structure can be used to read the file:

```
while(fileReader.hasNextLine()){
    String line = fileReader.nextLine();
    String[] elements = line.split(",");
}
```

Each of these methods performs the following:

- `fileReader.hasNextLine()`: This returns true if the file has a next line and false otherwise.
- `fileReader.nextLine()`: This gets the next line from the file.
- `line.split(",")`: This splits a string into an array of strings on the comma (e.g., `"word, this is a def."` → `{"word", "this is a def."}`).

Your Task: Take the above code segments and integrate them into the constructor. You will use the result of the line split to create an instance of `DictEntry` each iteration and add it to the dictionary.

Methods

Each of these methods will be very short (1-3 lines) as they will effectively serve as wrappers for the AVL tree methods since the `dictionary` attribute in the class is private.

1. `lookupWord`: This method should do three things: 1) take a single string word as a parameter, 2) create an instance of `DictEntry` with a `null` entry for the definition, and 3) use that `DictEntry` to call the search method for the AVL tree and return the result.
2. `addWord`: This method should take two string parameters: A word and a definition. It should then create a new `DictEntry` using those parameters and add it to the dictionary.

Each of these methods should have `public` access.

1.4 Step 3: Implementing Main

This will be the main class for the user to interact with the dictionary. Provided is a line of code that creates a new instance of a `Dictionary` and assigns it to the variable `dict` along with a `while(true)` loop in which you will implement the following interactive structure.

1. Ask the user for a word
2. Use the `lookupWord` method from the `dict` to lookup that word and store the result
3. If the results are `null` indicating that the word was not found you should...
   (a) Ask the user if they want to add the word to the dictionary.
   (b) If they do ask them for the definition and use the `addWord` method to add that word and associated definition to the dictionary.
4. If the results weren't `null` (i.e., the word was found) then you should print the word entry.