CSE 413 19wi Homework 6

Assigned: Friday, Feb. 22, 2019
Due: Wednesday, Feb. 27, 2019 by 11 pm via Gradescope

This assignment contains two short Ruby programming exercises. The first part involves simple text manipulation and use of Ruby containers, especially hashes. The second part involves creating a small set of classes with methods to manipulate a simulated file system consisting of Document and Directory objects.

When you are done, you should submit the two source files containing your Ruby code using Gradescope. You should submit the files for both parts as a single Gradescope assignment (hw6). They will be graded independently of each other, of course.

Notes: You are free to use any or the standard Ruby language and library classes and documentation, and you should do so. You do not need to give detailed attribution to information found in that documentation. (There are links on the course web site's Ruby resources page to the main Ruby documentation sites.)

You may be able to find solutions to these or similar problems with a web search or by looking in various Ruby repositories. Please try to refrain from doing that, since the real value of this assignment is solving the problems and learning your way around Ruby. If you do base your code on any online resources, be sure to credit them properly.

Part I

In a file named `wordfreq.rb`, write a Ruby program that has a single command-line argument that is the name of a text file. The program should open that text file, read the words in it, and then print out a list of all of the words found in the file and the number of times each word appears in the file. The output should be sorted using the standard Ruby `<` ordering for strings. Each word should appear at the beginning of an output line followed by a space and then followed by the number of times the word appears in the file.

For example, suppose the input file `quotes.txt` contains the following:

```
to be or not to be
to do is to be
to be is to do
do be do be do
```

Then the output of the program when it reads `quotes.txt` should consist of the following lines:

```
be 6
do 5
is 2
```

```
not 1
or 1
to 6
```

You should make the following simplifying assumptions (i.e., this is meant to be a simple exercise in text processing, not a complicated natural language processing program):

- The file contains basic alphabetic and possibly numeric (ASCII) characters. You do not need to verify this.
- You do not need to deal with words that contain embedded punctuation like `can't` or with any other punctuation marks. You can either discard any punctuation marks encountered or just assume that the source file does not contain any punctuation at all.
- A "word" is a sequence of characters separated by spaces or newlines.
- Words must match exactly to be considered the same. For example, `word`, `Word`, and `woRD` are three different words because they consist of different combinations of lower- and upper-case letters. Each of these should be counted separately.

Hints:

Ruby's string `split` method is useful for this problem, as are Ruby hashes.

When a Ruby program is run as a script or executable program, its command line arguments can be accessed using the variable named `ARGV`. To run this program and pass it the command-line argument `story.txt`, the following ruby command can be used:

```
> ruby wordfreq.rb story.txt
```

Ruby's `File` class contains methods to access files, and you can iterate through (i.e., read) a file much like any other sequence by using an appropriate method call with a block argument.

CSE 413 19wi Homework 6

Part II

This section contains a sequence of three problems to create two related classes with a set of specified methods. The code for these two classes should be contained in a Ruby source file named `classes.rb`.

1. For this problem, create a Ruby class called `Document` that represents a simple text document in a hypothetical file system. Each `Document` object should keep track of the time it was last modified (stored as a Ruby `Time` object), its contents (stored as a string), and a history of its contents (which you should implement as a Ruby array).

   Your `Document` class should implement the following methods:

`initialize`	Creates an empty `Document` (whose contents are an empty string) and records that it was modified at the time the method was called.
`contents=(new_contents)`	Sets the `Document's` contents to be equal to `new_contents`.
`contents`	Returns the `Document's` contents as a string.
`modified`	Returns a Ruby `Time` object representing the last time this `Document` was modified. Actions that modify the `Document` are creating it, settings its contents, or undoing.
`size`	Returns the size of the `Document`, which is equal to the number of characters in its contents.
`undo(n=1)`	Takes one optional parameter, `n`, with default value 1 and reverts the contents of the `Document` to the way they appeared `n` versions ago, removing the current version and any skipped-over versions entirely from the history. If there are not `n` versions of history available, returns `nil` and does not modify the `Document`. Otherwise, returns the new contents of the `Document`. Example usage: If a document had historical versions A, B, C, D (from oldest to newest) and current version E, calling `undo(3)` would revert to version B, and then calling `undo(1)` would revert to version A.

2. Create a Ruby class called Directory that stores Documents or other Directory objects, each specified by a name (stored as a string). You should store the children of your Directory object using a Ruby hash.

   Your Directory class should implement the following methods:

`initialize`	Creates an empty `Directory` with no children.
`store(name, child)`	Stores the given `child` object under the name `name` in this `Directory`. If a child is already stored under that name, replaces it. You may assume the child will be a `Document` or another `Directory`.
`get(name)`	Returns the `Document` or `Directory` stored under the given name.
`delete(name)`	Deletes the `Document` or `Directory` stored under the given name, removing it from the `Directory` object. Returns the deleted child object, or `nil` if there is no object by that name.
`size`	Returns the total size of all the `Documents` in this `Directory`, including those stored recursively in child `Directory` objects.
`undo(n)`	Attempts to undo every `Document` stored anywhere in this `Directory` by `n` versions, including those stored recursively in child `Directory` objects. If any `Document` does not have enough history to be reverted `n` versions, it should not be modified.

3. Now, add a method to your `Directory` class called `get_by_path` that builds on the functionality of `get` by accepting a file path representing the location of a `Document` within a `Directory` and recursively examining child `Directory` objects until reaching that `Document`. A well-formed file path should be a string consisting of 0 or more directory names and then a document name, with all components separated by forward slashes (you may assume directory and document names will not contain forward slashes). If the file path is malformed in any way, such as any of the components of the path not existing or the final name referring to a `Directory` instead of a `Document`, your method should return `nil` (this may require some fairly thorough error checking).

   Hint: Since you will need to split the input string on forward slashes, you may find it easiest to write a `protected` helper method taking an array of file path components for recursion (`protected` so that instances of the same class can call it but not instances of any other class).

   For example, given the following file tree:

```
Directory
├ "hw": Directory
|       └ "hw6": Directory
|                 └ "hw6-solution.rkt": Document A
└ "notes.txt": Document B
```

   Then these are the results of calls to `get_by_path` on the top-level `Directory` object:

```
get_by_path('notes.txt') => Document B
get_by_path('hw/hw6/hw6-solution.rkt') => Document A

get_by_path('lectures/lec1.zip') => nil
get_by_path('notes.txt/a.pptx') => nil
get_by_path('hw') => nil
```

Ruby Hints

To get the current time as a Ruby Time object, you can call the `now` method:

```
Time.now
```

Remember that in Ruby, defining a method called `contents=` allows clients to use a convenient syntax to "set" that field, while really calling the method (and allowing the object to run whatever code is needed to update its state appropriately):

```
doc = Document.new
doc.contents = 'I do not like green eggs and ham.'
doc.contents  # => 'I do not like green eggs and ham.'
```

Also, when a Ruby argument is given a default value, it becomes optional and does not need to be specified. These three calls would therefore be equivalent for `undo(n=1)`:

```
doc.undo
doc.undo()
doc.undo(1)
```