CSCI 204 – Introduction to Computer Science II

Project 4 – Search and Sorting

Assigned Wednesday, November 15th, 2017
Due Tuesday, December 5th, 2017

1 Objective

The purpose of this assignment is to practice search, sorting, reading and writing files, user interaction, and graphical user interface (GUI).

2 Introduction

Reading classic literature can enrich one's life! In this project, you will develop a system that helps users to search and update a database of classic literature. You are given an HTML file named "classicliterature.html" which contains a list of mostly 19th century novels plus a few other entries. This is a modified list from https://www.thoughtco.com/19th-century-novels-reading-list-737909. The list is organized by the author's names. For each author, a collection of his or her representative works is listed. Your task is to read this list, build two search trees, one using the author's name and the other using the title of the book as search keys, create a user interface that allows the user to search the records either by book's title or by author's name. The system should also allow the user to add more authors and titles to the system. When a user decides to leave the system, your program should store the updated list in two formats, the HTML format, and the Python's pickle format so the system is ready for next use.

3 Technical Details

A project of this type can be implemented in many different ways. In this section, we describe some technical details and specific requirements.

3.1 Raw data and its representation in the program

The data used by the program is stored as a complete HTML file. Your program needs to read the file and parse the file into proper records that can be put into a search tree node. An HTML file is a plain text file that contains formatting commands (tags) in addition to the contents. When reading an HTML file, a web browser such as FireFox formats the output according to the tags in the HTML file and displays the content on the screen. An HTML file contains two main sections, a header section and a body section. Your program can skip the header section (the part that precedes `<body>` tag) and take the information directly from the body section. Here is a sample raw HTML file with a head and body section and two entries in the body section in the same format as you'd see in our input file "classicliterature.html".

Note that the head section contains a title and some comments. Your program should save the header section when reading the data. Your program should also save anything starting from the closing body segment indicated by the tag `</body>` (trailer). Both parts will be used when saving results to files when the program finishes.

```
<html>
<head>
<title>List of classic novels</title>
<!--
<p>The following list is a copy from
<a href =
"http://classiclit.about.com/od/19thcenturyadwriters/a/19thnovels_wr.htm">http://classiclit.
about.com/od/19thcenturyadwriters/a/19thnovels_wr.htm</a>
with some additions.</p>
-->
</head>

<body>
<p>Alcott, Louisa May</p>
<ul>
<li>Little Women </li>
</ul>

<p>Austen, Jane </p>
<ul>
<li>Emma </li>
<li>Mansfield Park</li>
<li>Persuasion </li>
<li>Pride and Prejudice </li>
</ul>
</body>
</html>
```

Inside the body section, contents are listed by author's last name. Each author's name is enclosed in a pair of paragraph tags (`<p>` and `</p>`). The family name and the given name are separated by a comma. Each title that belongs to that author is listed in an unordered list between `<ul>` and `</ul>`. The number of titles listed for each author is not known ahead of the time. But you may assume at least one entry is included in the unordered list. Your program should create and store the relevant information for each author in a Python class with last name, first name, and a list of titles as the data members. Let's call this class `Node`. The program should first store this collection of nodes in a Python list. Then the program can build two search trees from this collection of nodes.

3.2 Building the search trees

Once you have a collection (a Python list) of nodes, you should build two search trees, one using last name of the author as the search key, the other using the book title as the search key. Searching the name-based tree should return a list of titles the author published, while searching the title based tree should return the author's full name and the complete list of titles this author published. You are given an implementation of binary search tree. You should first use this binary search tree class to implement the project to get a sense how things should work, though you are not required to do so. The project asks you to implement an `AVLTree` class and use the `AVLTree` to replace the binary search tree (`BST`).

In particular, you are asked to create the `AVLTree` class as an inherited class from the `BST` class. The `AVLTree` should reuse the functionality provided by BST class as much as possible, for instance, use `self._root, self._size, self.__len__(), self.__contains__()` and other possible methods and data members of the parent class. Refer to the course website for a complete implementation of the `BST` class and a partially implemented `AVLTree` class.

3.3 Operations that should be supported by your program

After building the search trees, your program should support the following operations.

1. Search by author's name: A user can type in an author's last name and first name to search for associated publications. If the name is found in the database, your program should print a complete list of publications that associated with this author. If the name is not found, your program should display a message indicating so. The search should be case insensitive (i.e., upper case and lower case should generate the same result.)
2. Search by the title of publication: A user can type in a title, the system should return the author's full name and a list of other publications by this author. If the title is not found in your database, the program should display a message indicating so.
3. Add an author and books: A user can request the addition of an author that is not in the database. When doing so, the user needs to provide a full name of the author and a list of book titles associated with this author. You may assume that the author names and the titles of publications are unique that wouldn't cause problems in search tree.
4. Remove an author from the database: A user can request the removal of an author by supplying the author's last name. If an author is removed, the entire publication record is removed (i.e., other publications from the publication search tree should all be removed.)

These operations should be managed through a menu-driven program which is demonstrated in the given program `menu-demo.py`. Your program should provide a menu of operations and the user can pick which operation she wants to perform. Once the operation is completed, the menu should be displayed on the screen so that next operation can be chosen. (See the transcript of a sample run for an example.) For some extra credits, you can implement a graphical user interface (GUI).

3.4 Store the results back to files

One of the menu operations should be "Exit." Upon executing this option, the program terminates. Before terminating the program, the data needs to be stored back to two files. One is a text file in HTML format. The result should look very similar to the original input file with possible additions and removals of the entries. The other is in Python's *pickle* format.

Writing the data back to a text file in HTML format should be a familiar task. You have done similar writing to text files in previous work in this course. What you need to do is to go through either the search tree or the Python list where you kept the data and generate the body segment of the HTML file by adding tags to each of the element. You need to add the paragraph tags to each author and the unordered list tags for each of the publication of a particular author.

3.4.1 Store the results in Python *pickle* format

Writing the result back to an HTML file helps the user to read the information using a web browser. However if you start your program again, you'd have to rebuild the search trees or any other data structures you might have created for your program. Python allows programmers to store the data that a program created in a structured manner and reuse the data next time as needed without having to recreate them. Assume you have created a binary search tree filled in with data. If you store the data in Python pickle format, your program will be able to simply read this data back into a binary search tree next time you need to use it. The following is a segment of an example of using Python pickle to create a file and to read a file back after having created it. The complete program is a part of the files you are given for the project. The program is named `testpickle.py`.

To use methods in the pickle package, one needs to import the package. The pickle package uses a `dump()` method to store the information into a disk file in binary format (note the file is open for writing in *'wb'* which means writing in binary.) If you have multiple pieces of data need to be stored, call the `dump()` method multiple times in a chosen order. Use `load()` method to read back from a pickle file, in the same order as the data were written. In the following example, *b* is written before *c* when saving the file. So *b* should be read back before *c* when loading.

```
import pickle
# assume a Book class has been defined
b = Book( 'hello', 'author 1' )
c = Book( 'world', 'author 2' )
print( 'before pickle ' )
print( b )
print( c )
## To create a pickle file, uncomment the following three lines.
#f = open( 'data.pickle', 'wb' )
#pickle.dump( b, f )
#pickle.dump( c, f )

## The following three lines are used after the pickle file has been
## created.
f = open( 'data.pickle', 'rb' )
b = pickle.load( f )
c = pickle.load( f )
print( 'after pickle ' )
print( b )
print( c )

f.close()
```

To make full use of a pickle file, your program should check to see if a data pickle file exists (you can choose a default file name, or you can ask the user to enter a file name). If the pickle file exists, use the pickle file as your data. If the specified file doesn't exist, create the data (e.g., the search trees) from the source file (the HTML file for the book records).

4 The Files You Are Given

You are given a set of files to help you start the project. These files are given at the Linux directory at

~csci204/2017-fall/student/project/p4/

You need to copy the entire directory as there is a sub-directory within that directory. You can use the command

cp -r ~csci204/2017-fall/student/project/p4/ .

Don't forget the last dot '.' which indicates you copy everything recursively to your current directory.

In that directory, you should see the following files.

```
avltree_student.py,  bst.py,   bstiter.py,  gui/, classicliterature.html,
menu_demo.py, read_html.py, testavl.py,  testbst.py, test.html,
testpickle.py, transcript
```

A complete implementation of binary search tree class is given in `bst.py`. The `avltree_student.py` contains a partial implementation of an AVL tree class. **Note that while you can consult, copy, and paste code segment from this file, you are asked to implement the AVL tree class as an inherited class from binary search tree.** The required implementation is different from the file given. You should consult your lecture notes, lab work, the textbook and the code example on the course website for the syntax needed to implement an inherited class. The two test files (`testbst.py` and `testavl.py`) allow you to test various classes as needed. The program `testpickle.py` shows how to create and use Python pickle files.

The *classicliterature.html* is a list of literature works and their authors, to be used as the input for your program. The directory `gui` contains an example and necessary package for you to create GUI in the second phase. Study the example `gui.py` in that directory to learn how a GUI can be written.

In addition, the file `read_html.py` is a program that you can use directly in your project to read an HTML file for a given format and store the information in a list. This segment of code is extracted directly from the solution set. Giving you this piece of code allows you to concentrate on the core task of creating the AVL tree for search. Try to run the program (make sure both *test.hml* and *classicliterature.html* are in the same directory)

```
python read_html.py
```

The program `menu_demo.py` demonstrates how to write and use a menu-driven program.

5 Implement the Project

It will be a good strategy to implement the project in steps, though you could jump into the final implementation. Only the final implementation will be graded. You should first complete the tasks of reading the HTML file, building two search trees using the given `BST` class, providing a text-based menu driven user interface that allows search, insertion, and deletion of records, and writing the result to a text file in HTML format.

When the above part works, you can them complete the `AVLTree` class (given partially). Specifically, you need to design and implement the AVLTree class as an inherited class from BST and implement the `remove()` method that is missing from the partial implementation. You then replace the `BST` in the first phase with the completed `AVLTree` class. In addition, you need to complete the writing of data to files in Python's pickle format.

You are also asked to implement a graphical user interface (GUI). Please study and run the example `gui.py` in the directory `gui`. In that example, the GUI is created and used twice. We suggest you use it in a similar pattern. When displaying the user choice menu, use one GUI; when allowing the user to enter author and book information using another GUI. In short, you may create and use as many as GUI as necessary, but only one at a time.

6 Test Your Program and Demonstrate It Works

Test each segment of your program as you develop it. Don't wait until all code has been written before doing any testing. You are asked to create a separate `testdb.py` program that demonstrates the programs you write for this project works well. In particular, the test program should show all functionality required by the project description, searching by author's name, searching by a title, adding an author and her collection, removing an author (thus removing all related publications).

Since the list of books in the input file *classicliterature.html* is relatively long (133 titles at this point), it may be better to create a small subset of these titles for testing purpose (e.g., pick a few authors and their books and call the test data *test.html*). Test your program using the *test.html* as the input until everything is working properly. Then you can test your program using the original data. It is possible that the large test data may reveal some problems you didn't see with the smaller test set, but the strategy should better help you find many initial problems.

Create a `readme.txt` file that record the test runs for each of the two phases. Make sure you put your name and the assignment information in this file.

7 Submission

Submit one zip file containing all necessary files so that we can run the test directly from your folder after unzipping the files.

8 Grading Criteria

When grading, we will look for the quality of the program from various aspects. Here are the grading criteria.

1. Functionality: 60%

   User-friendly and well-formatted input and output via GUI (10%)
   Correctness of computation (25%)
   Program satisfies the problem specifications (25%)

2. Organization and style: 30%
   a. Program follows object-oriented design approach, contains well designed classes and uses inheritance of AVL tree from BST tree (10%)
   b. Variables and functions have descriptive names following Python convention (5%), program contains sufficient amount of docstrings explaining the functionality of its major parts and comments explaining specific actions that blocks of code are performing, each code file has comments on the top including programmer's (your) name, date and file description (5%)
   c. Code is structured, clean, organized and efficient. Each python code file that doesn't include the "main" function contains description of only one class (10%). *To earn full grade in this category, avoid leaving commented-out legacy code in your submission.*

3. Extraordinary elements, creativity and innovation: 10%

   This includes anything beyond the provided specification that makes your program stand out.

4. **Extra credits**: up to **10%** will be given as an extra credit work if you implement a working GUI based on the example given.