Linked Data Structures and Parse Trees

ISTA 350 Hw2, Due Thursday 2/13/2020 at 11:59 pm

Introduction. In this homework, you will implement parse trees in two different ways: as a linked data structure and as nested dictionaries. We will use the hot bills context again, but these classes can be adapted to store any sequences with only minor modifications. **We will shortly explore in class why parse trees are so excellent for problems like this. They have a huge advantage over hw1's lists. What do you think it is?**

Instructions. Create a module named `hw2.py`. Below is the spec for three classes containing nine methods and one function. Implement them and upload your module to the appropriate D2L Assignments folder.

Testing. Download `hw2_test.py` and the auxiliary files and put them in the same folder as your `hw2.py` module. Run it from the command line to see your current correctness score. Eight methods and the function will be graded (the code for the ninth method is below). Each of the methods/function is worth 11% of your correctness score. You can examine the test module in a text editor to understand better what your code should do. The test module is part of the spec. The test file we will use to grade your program will be different and may uncover failings in your work not evident upon testing with the provided file. Add any necessary tests to make sure your code works in all cases.

Documentation. Your module must contain a header docstring containing your name, your section leader's name, the date, `ISTA 350 Hw2`, and a brief summary of the module. Each function/method must contain a docstring. Each docstring should include a description of the function's purpose, the name, type, and purpose of each parameter, and the type and meaning of the function's return value.

Grading. Your module will be graded on correctness, documentation, and coding style. Code should be clear and concise. You will only lose style points if your code is a real mess. Include inline comments to explain tricky lines and summarize sections of code.

Collaboration. Collaboration is allowed. You are responsible for your learning. Depending too much on others will hurt you on the tests. "Helping" others too much harms them in reality. Cite any sources/collaborators in your header docstring. Leaving this out is dishonest.

Resources.
https://en.wikipedia.org/wiki/Linked_data_structure

Method specifications for classes `Node`, `WatchListLinked`, and `WatchListDict`.

`class Node:`

`init`: This magic method takes an item to be stored in the `Node` instance that defaults to `None`. Initialize an instance variable called `datum` to the argument. Initialize an instance variable called `children` to the empty list.

`get_child`: This instance method takes a key (search target) that defaults to `None`. If there is a child that contains the key, return it; otherwise return `None`.

Insert this code into class `Node` so that the test will work:

```
    def __eq__(self, other):
        """
        The data comparison happens in get_child.
        This is strictly a helper method.
        """
        if len(self.children) != len(other.children):
            return False
        if not self.children:
            return True
        equal = True
        for child in self.children:
            other_child = other.get_child(child.datum)
            if not other_child:
                return False
            equal = equal and child == other_child
        return equal
```

`class WatchListLinked:`

`init`: This magic method takes a filename that defaults to the empty string. Initialize an instance variable called `root` to a node that has 5 children. The `datum` in the node should be the `None` object (let your default arg do the work, don't pass it in). The data in the 5 children are the 5 denomination strings (i.e. `'5', '10'`, etc.), respectively. The children have no children of their own. If a filename was passed in, each line of the file will represent a bill that we want to add to our watch list tree and will be in the format `'<serial_number> <denomination>\n'`. Look at one of the bill files in a text editor to see specific examples. Insert each of the bills into the watch list. Finally, an instance variable called `validator` holds a compiled regular expression that will be used to check for valid serial numbers (use the same regex as you did in hw1).

`insert`: This instance method takes a string representing a serial number and a string representing a denomination. Insert the bill into the watch list. We will do this one in class, but it will not be posted. If you miss that class and you can't figure it out yourself, you will need to get it from another student.

`search`: This instance method takes a string representing a serial number and a string representing a denomination. It returns `True` if the serial number is in the watch list, `False` otherwise. You should be able to implement this as a slight modification to `insert`.

`class WatchListDict:`

`init`: This magic method takes a filename that defaults to the empty string. Initialize an instance variable called `root` to a dictionary that maps each of the 5 denomination strings (i.e. `'5', '10'`, etc.) to an empty dictionary. If a filename was passed in, each line of the file will represent a bill that we want to add to our watch list dictionary and will be in the format `'<serial_number> <denomination>\n'`. Look at one of the bill files in a text editor to see specific examples. Insert each of the bills into the watch list. Finally, an instance variable called `validator` holds a compiled

regular expression that will be used to check for valid serial numbers (use the same regex as you did in hw1).

`insert`: This instance method takes a string representing a serial number and a string representing a denomination. Insert the bill into the watch list. We will do this one in class, but it will not be posted. If you miss that class and you can't figure it out yourself, you will need to get it from another student.

`search`: This instance method takes a string representing a serial number and a string representing a denomination. It returns `True` if the serial number is in the watch list, `False` otherwise. You should be able to implement this as a slight modification to `insert`.

Function specification:

`check_bills`: This function takes a watch list instance of either class and a filename. The file contains a list of bills in the format `'<serial_number> <denomination>\n'` that we wish to check against our watch list. The method returns a list of bills in the format `'<serial_number> <denomination>'`. Go through the file line-by-line. If a bill is on the watch list, append it to the list of bad bills that you are building. It is also a bad bill if it has an invalid serial number. Return the list of bad bills. **Why did we make this a function instead of instance methods in the classes?**