# ISTA 350 Linked Data Structure Worksheet Name:

Define a class called `Searcher`. `init` takes a string representing a regex with a default of the empty string. If the argument is the empty string, get a regex from the user. Either way, compile the regex and store it in an instance variable called `re`. Create an instance method called `search` that takes a filename and returns a list of all the strings in the file that match the regex. Recall that the `findall` method for compiled regexes returns a list of strings if the regex has no more than one group. If the regex has multiple groups, it returns a list of sequences (tuples) of strings. In this case, the first string in each tuple is what we want. Recall the `type` function.

Define a class called `Node`. `init` 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. Create an instance method called `get_child`. This instance method takes a key that defaults to `None`. The term "key" has multiple meanings in computer science. In this case, it's the target of a search, not a key in a dictionary. If there is a child that contains the key, return it; otherwise return `None`.

If you can't understand what a spec is asking for, you can't fulfill it. So we're going to practice that. Here is the specification for `init` from hw1:

```
class WatchList:
```

`init`: This magic method takes a filename that defaults to the empty string. Initialize an instance variable called `bills` to a dictionary that maps each of the five denominations of interest, represented as strings (i.e. `'5', '10'`, etc.), to an empty list. 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. Append the serial number for each bill in the file to the appropriate list in the dictionary. A Boolean instance variable called `is_sorted` indicates whether or not the lists in the dictionary are sorted. Assume that the bill files are not sorted. Finally, an instance variable called `validator` holds a compiled regular expression that will be used to check for valid serial numbers (see the **Introduction** above for the rules governing serial numbers).

Draw an empty `WatchList` instance. Draw a `WatchList` with two $10 bills in it.

Hw2 asks you to implement a parse tree in two different ways. The `WatchListLinked` class will build its parse tree using the `Node` class you implemented above. Here is its `init`:

```
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 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).

Draw an empty watch list. Draw a watch list that has had the following strings parsed and inserted: `"abc 5", "abef 5", "abcd 5", "abce 5", "abcde 5"`. The last character in each string will have as one of its children an empty node, i.e. its datum is `None` and it has no children. This is how we store the fact that we have reached the end of a string so that we can store, for instance, both the strings `"app"` and `"apple"`.