COSC122, 2020 Semester 2
INTRODUCTION TO COMPUTER SCIENCE

Assignment 3—Trees are fun

7% of overall grade
Due 11:55pm on Fri 16 October 2020

1 Overview

1.1 Introduction

This assignment follows on from assignment two in that the basic goal is still to generate a list of test results (or lack thereof) for quarantined people. This time the main data structure will be a Binary Search Tree (BST). In the first task you will finish the implementation of the BST class and in the second task you will use a BST to help look up test results.

1.2 Due date

The official due date is 11:55pm on Fri 16 October 2020.

Students can submit to the normal submission quiz up two days after this (ie, Sun 18 Oct) with no penalty (ie, normal submissions close on Sun 18 Oct). After this point submissions can be made until the drop dead date, which is Fri 23 October, via the *late* submission quiz. But *late* submissions incur a 15% absolute penalty, ie, if you submit to the *late* quiz then `your_mark = raw_mark - 15`. Please ask if you are unsure.

1.3 Submission

Submit via the quiz server. The submission quiz won't open until about a week before the assignment is due—to emphasise the point that the quiz won't offer much help, your own testing and the provided unit tests are what you should be using to test your code. The submission quiz will not test your code properly until after submissions close, that is, it won't give you any more information than the provided tests so you can start work straight away! *This assignment is a step up from COSC121 so please make sure you start early so that you have time to understand the requirements and to debug your code.*

1.4 Implementation

All the files you need for this assignment can be found on the quiz server. Do not import any additional standard libraries unless explicitly given permission within the task outline. For each task there you will need to complete a/some function/s. Feel free to write other helper functions if you wish. All submitted code needs to pass *pylint* style checks—make sure you leave time for style adjustments when submitting.

1.5 Getting help

The work in this assignment is to be carried out individually, and what you submit needs to be your own work. You must not discuss code-level details with anyone other than the course tutors and lecturers. You are, however, permitted to discuss high-level details of the program with other students. If you get stuck on a general programming issue, you are encouraged to ask your tutor or the lecturer for help. You may not copy material from books, the internet, or other students. We will be checking carefully for copied work. If you have a general assignment question or need clarification on how something should work, please use the class forums on Learn, as this enables everyone to see responses to common issues, but **never** post your own program code to Learn (or make it available publicly on the internet via sites like GitHub)[1]. Remember that the main point of this assignment is for you to exercise what you have learned from lectures and labs, so make sure you do the relevant labs first, so don't cheat yourself of the learning by having someone else write material for you.

2 Looking up results

The overall goal in this assignment is similar to that of the previous two assignments. You will ultimately be required to write a function that takes a list of test results and a list of quarantined people, and returns a list containing the quarantined peoples' test results, or lack thereof. The result finder function in this assignment will use a Binary Search Tree (BST) to store the data from the list of tested people. As in previous assignments your finder function will need to count the number of times `Name` objects are compared and return this count along with the list of results—as a measure of the work done. Your BST functions will need to help out by also keeping track of the number of `Name` comparisons used.

The file `tests.py` provides you with a suite of tests to help you check your implemented code before submission. These tests use the Python unit-test framework and *you are not expected to fully understand how they work*; you just need to know that the tests will check that your code generates the correct list of output, and that the number of comparisons that your code makes is appropriate. If you think that you have an implementation that is very close to the test range, and works reliably, you can ask to have the expected number of comparisons range reconsidered.

The quiz server tests that use for marking will mainly be based on the provided unit-tests, but there will be a number of other test cases. Therefore passing the unit tests is not a guarantee that full marks will be given (however it's a good indication your code is working as required). Please make sure you test your code with edge cases such as empty lists, etc...

[1] Check out section 3 of UC's *Academic Integrity Guidance* document. Making your code available to the whole class is certainly not discouraging others from copying.

3 Tasks

Your main tasks are listed in his section. More details on the provided files and tests are provided in subsequent sections. Please make sure you read everything so you are aware of the changes since assignment 1 and 2.

3.1 Binary Search Tree (BST) Functionality [70 Marks]

This task requires you to complete the following functions in the `bst_module`. The functions carry out various BST operations. You should follow the specification in the function docstrings and use the provided tests to help check that your function work as intended. Please ask if you are not sure about something or you think the specification is missing something important. Please note that the following are all functions rather than methods. This means their first parameter is typically the root of a BST or None if the BST is empty.

bst_store_pair Basically stores a node with the given key and value in the appropriate place in the tree. If the key exists in the tree then the value in the node with that key should be updated to the given value. This function should return the number of key comparisons used when adding the given key,value pair.

get_value_from_tree If the key exists in the tree then this function returns the value associated with the given key and the number of key comparisons used. Otherwise this function returns `None` and the number of comparisons used.

min_key_in_bst Returns the minimum key in the BST starting at the given node.

max_key_in_bst Returns the maximum key in the BST starting at the given node.

num_nodes_in_tree Returns the number of nodes in a the tree starting at the given node.

bst_depth Returns the depth of the tree starting at the given node. Be sure to read the docstring for this one.

bst_in_order Returns a list of all the (key,value) tuples from the tree, in order by key. No key comparisons should be used here!

You should write the `bst_store_pair`, `get_value_from_tree`, `min_key_in_bst`, and `max_key_in_bst` functions using non-recursive code (a while loop will come in handy here). Recursive implementations will tend to crash Python when adding large lists of items that are already in order. Feel free to write recursive versions as and exercise as they will be more interesting and maybe even easier—they should pass the smaller test cases ok but don't be surprised if Python just gives up (or bogs down severely) on the larger test cases. Just make sure you submit robust non-recursive versions for marking.

The other functions are a bit more tricky to write in a non-recursive way (ie, you would need to implement a stack to keep track of everything) so you should write them recursively and test them only with small to medium lists/files of data. Our marking will only test them with suitably small test data.

Please ask us if you are not sure about non-recursive vs. recursive implementations.

3.2 bst_result_finder [30 Marks]

This task is similar to the result finders in the previous assignments. Your function should start with an empty list representing the results for quarantined people. It should then, if needed, create a BST and add in all the `name, (nhi, result)` pairs from the `tested` list. The items in `tested` should be added in the order they appear in the `tested` list. In terms of the BST, the keys will be names and the values will be `(nhi, result)` tuples. If your function sets up a BST then it will need to go through each `Name` in `quarantined` and get the associated value from in the BST. If a match is found, append a tuple containing `(name, nhi, result)` to the results list. If a match isn't found then add a tuple with `(name, None, None)` to the results list as you don't have a NHI number or test result for that person. The returned list should be given in the same order that the names appear in the `quarantined` list.

Your `bst_result_finder` should return the results list and the number of Name comparisons used as a tuple, that is, in the form `(your_results_list, comparisons_used)`.

3.2.1 Notes

- For this task you cannot assume anything about the order of the items in the `tested` and `quarantined` lists. That is, some tests might use sorted lists but others will not.
- However, you can assume that there are no duplicate names in either the *tested* list or the *quarantined* list, ie, within either list no name will appear more than once.
- We recommend testing your function with very small lists first, eg, start out with one item in each list, then try with one in the first list and two in the second, etc, ...
- Make sure your `bst_result_finder` function deals with edge cases, such as empty `tested` and/or `quarantined` lists, properly. For example, you don't need to make a BST if no people have been tested.
- The provided tests are really just a final check—they won't help you very much when debugging your code.
- Your function shouldn't mutate the lists it is given in any way, eg, don't pop anything off the `quarantined` or `tested` list and don't insert anything into them either. You will, of course, need to append things to the results and this is fine.

4 Provided classes

4.1 classes3.py

We use `classes3.py` to provide the basic classes you will need for assignment 3. This ensures you don't accidentally import something old from assignment 1 or 2. Please make sure you download/extract all the files for assignment 3 into a new folder so nothing gets mixed up with assignment 1 or 2 (eg, you could extract the files into a folder named *assignment3*).

4.2 Node class

Nodes in this assignment are used in Binary Search Trees and as such they have the following attributes: a `key`, a `value`, a `left` pointer (to the root of the left sub-tree, containing keys that are smaller than the root key), and a `right` pointer (to the root of the left sub-tree, containing keys that are greater than the root key).

4.3 bst_nested_repr

This function takes a node and returns the string of a nested list representation of the tree starting at that node. The representation looks like [root.key:root.value, representation of left subtree, representation of left subtree]. This function will be useful for small trees—it gets a bit cumbersome with bigger trees.

4.4 Name class

For this assignment the main class you will need to know about is the `Name` class. A `Name` object is basically a repackaged `str` object, that updates an internal counter every time it is compared with another `Name` object. You will be required to count each `Name` comparison made by your code and our tests will use the internal counter to check that you got it right.

Further information on `Name` objects is given below.

- `my_name = Name(name)` creates a new `Name` object with the given name.
- When you are processing `Name` objects you shouldn't access the `_name` attribute, eg, you shouldn't do things like `name1._name == name2._name`, you should just use `name1 == name2`!
- `print(my_name)` will print the Name object, in the form <name> so you don't confuse it with a simple `str` object.
- Every time two `Name` objects are compared the name comparison counter in the StatCounter class will be updated. For example, whenever a comparison such as `name1 < name2` is evaluated the internal counter is incremented. You will see that all the comparison operators work like this, ie, <, >, >=, <=, ==, !=.
- You shouldn't be using any of the double under-scored methods for `Name` objects directly. You should use the normal comparison operators, eg, name1 < name2 will be automatically translated into `name1.__lt__(name2)` in the background so you don't need to call `name1.__lt__(name2)` directly!
- `StatCounter.get_count(NAME_COMPS)` gives the actual number of `Name` comparisons that have been performed. If you read the definition for the `Name` class you will see that `Name` objects update the name comparisons counter each time they are compared. This is for your debugging only and using this in submitted code will cause your code to fail the submission tests. You must use the line `from stats import StatCounter, NAME_COMPS` if you want to check this counter.

The following example code should help you further understand the Name class:

```
>>> from classes2 import Name
>>> from stats import StatCounter, NAME_COMPS
>>> my_list = []
>>> name1 = Name('Paul')
>>> name1
Name('Paul')
>>> print(name1)
<Paul>
>>> my_list.append(name1)
>>> my_list.append(Name('Tim'))
>>> my_list
[Name('Paul'), Name('Tim')]
>>> name3 = Name('Zheng')
>>> name1 < name3
True
>>> my_list[0] == Name('Paul')
True
>>> StatCounter.get_count(NAME_COMPS) #This is allowed only for testing!
2
>>> my_list[1] >= my_list[0]
True
>>> StatCounter.get_count(NAME_COMPS) #This is allowed only for testing!
3
```

5 Provided tools and test data

Important Note: The contents of some test files are slightly different from those used in assignment 1 so make sure you extract all your assignment 2 files into a different folder to the one you used for assignment 1.

Test files are given to you in the folder `test_data` to make it easier to test your own code. The test data files are named in the form `test_data-{i}{x}-{j}{y}-{k}-{m}.txt` and contain three lists of data. The first list, referred to as *tested*, contains (NHI, name, result) tuples for people that have been tested for the virus. The second list, referred to as *quarantined*, contains the names of people who are quarantined. The third list, referred to as *results*, contains (name, NHI, result) tuples for all the quarantined people, in the same order as they appear in the *quarantined* list. If a quarantined person isn't in the *tested* list then their NHI and result will be None.

5.1 Decoding data file names

The *tested* and *quarantined* data sections start with a line containing the number or records in that section. The third data section doesn't have this number as it will have the same number of records as the *quarantined* list. Note that lines starting with # are commented out and are not read.

- i = number of records in *tested* list
- x = 'i' if *tested* is sorted by NHI or 'n' if *tested* is sorted by name.
- j = number of records in *quarantined*
- y = 'n' if *quarantined* is sorted by name or 'r' if it's unsorted/randomly ordered.
- k = number of *quarantined* people that have a result, ie, can be found in the *tested* list.
- m = the random seed used to generate the test file. We may generate different random files for the quiz server tests.

For example, a file named `test_data-5i-5n-2-a.txt` will contain: 5 records for tested people that are sorted by NHI, 5 names of quarantined people that are sorted by name, 5 results records for the quarantined people, and two of the quarantined people will have have results in the tested list (in the example below Cissiee and Jon are the two).

```
# Number tested and their details:
5
2120000,Zorina Latin,True
2120002,Selime Sziladi,False
2120003,Vallipuram Kortekaas,True
2120005,Jon Klaudt,False
2120007,Cissiee Bednar,True
# Number quarantined and their names:
5
Andrzej Challice
Cissiee Bednar
Jon Klaudt
Meris Dendi
Rheal Wolfenbarger
# Expected result details for quarantined people:
Andrzej Challice,None,None
Cissiee Bednar,2120007,True
Jon Klaudt,2120005,False
Meris Dendi,None,None
Rheal Wolfenbarger,None,None
```

When tested records are sorted by NHI the names will effectively be in random order (as seen in the example above).

5.2 Reading data files

The `tools.py` module contains functions for reading data from test files and for making simple lists of `Name` objects, etc...The most useful function in the `tools` module is `read_test_data(filename)`, which reads the contents of the test file and returns a tuple containing the *tested, quarantined* and expected *results* lists respectively.

Suppose `test_data-2n-2n-1-a.txt` contains the following data, then the shell code below should give you an idea of how the data is read.

```
# Number tested and their details:
2
2120001,Filippa Mau,False
2120000,Sabina Starkes,False
# Number quarantined and their names:
2
Albert Willison
Sabina Starkes
# Expected result details for quarantined people:
Albert Willison,None,None
Sabina Starkes,2120000,False
```

Example data loading using `read_test_data`:

```
>>> import tools
>>> filename = "test_data/test_data-2n-2n-1-a.txt"
>>> tested, quarantined, expected_results = tools.read_test_data(filename)
>>> tested
[(2120001, Name('Filippa Mau'), False), (2120000, Name('Sabina Starkes'), False)]
>>> quarantined
[Name('Albert Willison'), Name('Sabina Starkes')]
>>> expected_results
[(Name('Albert Willison'), None, None), (Name('Sabina Starkes'), 2120000, False)]
>>> print(quarantined[0])
```

<Albert Willison>

5.2.1 Making your own tested/quarantined lists

The `make_name_list` function in the `tools` module can be used to make lists of `Name` objects from simple lists of strings or using the letters in a string. The `make_tested_list` function will make a list of `(nhi, Name, result)` tuples from a simple list of strings, or using the letters in a string. `make_tested_list` will generate the NHI numbers and test results for you and there are options for list sorting and the result you want everyone to have. Check out the function definitions and doc-strings for more details. An example of each is given below.

```
>>> my_quarantined = tools.make_name_list(['Bob', 'Abba', 'Faba'], sort_order='name')
>>> my_quarantined
[Name('Abba'), Name('Bob'), Name('Faba')]
>>> my_tested = tools.make_tested_list(['Bingle', 'Zabba', 'Faba'], sort_order='name')
>>> my_tested
[(1, Name('Bingle'), True), (3, Name('Faba'), True), (2, Name('Zabba'), True)]
```

You will be able to use these functions to quickly generate simple lists that you can send to your functions for processing. This will help you verify and debug your code by allowing you to compare your hand-cranked results with the results returned by your function. For example, you could generate the lists above and then call your function to see if it returns the result you expect. If not then you need to work out whether your hand-cranking was wrong or your function was wrong...

6 Provided tests

The `tests.py` provides a number of tests to perform on your code. Running the file will cause all tests to be carried out. Each test has a name indicating what test data it is using, what it is testing for, and a contained class indicating which algorithm is being tested. In the case of a test case failing, the test case will print which assertion failed or what exception was thrown. The `all_tests_suite()` function has a number of lines commented out indicating that the commented out tests will be skipped; un-comment these lines out as you progress through the assignment tasks to run subsequent tests.

To get all the test results in a more manageable form in Wing101 you should make sure that *Enable debugging* is turned off in the options for the Python shell. You can get to the options by clicking on the Options drop down at the top right of the Shell window.

In addition to the provided tests you are expected to do your own testing of your code. This should include testing the trivial cases such as empty parameters and parameters of differing sizes. You will, of course, want to start by testing with very small lists, eg, with zero, one or two items in each list. This will allow you to check your answers by hand.

NOTE: *The tests that are provided in `tests.py` aren't good for debugging! You should be using your own simple tests to help you debug*