EECS 214/395 Data Structures Fall 2014 Program 2

Posted: Saturday November 8th
Due: 11:59pm Tuesday November 25th
% of Final Grade: 12.5%

From Program 1: "Programming projects in this course will emphasize both fundamentals of data structures as discussed in lecture; and add a design component whereby students should be able to implement multiple data structures (including understanding inheritance where applicable) as building blocks towards a larger, integrated purpose." *To this we now add*: the design component might also requiring augmenting a data structure in our existing tool set.

In this project, students will build a binary search tree, with augmented functionality to include rank statistics as described in Section 14.1 of the textbook.

The tree does not have to be balanced so methods requiring walking from the root down to the leaves should run in O(h), not necessarily O(log n). It does need the structure we have previously described in lecture to allow for "binary" searches. You may assume values will always be integers.

Your Tasks

- Download the python file template "Pboth_Project2_acj123.py", which is attached to this assignment.

- Test code appears within a "__main__" block at the bottom; there are also some print statements that appear within the code that you should not remove; these are required to facilitate grading.

- Where my netID appears, change it to be yours.

- When you submit your file, it should be named as "P##_Project2_[netid]" where ## replaces "both" in the template, and is either 27 or 34 *to indicate which version of Python you are using*.

- Write the Insert( ) method in Tree_acj123 (TREE) class. Each time Insert has finished running, every single node in the tree should have accurate information regarding: parent, children, predecessor, successor, and 'rankCounter.' Some of these will need to be "initialized again" and others will need to be updated as Insert runs. It should run in O(h); it does not have a return value. (20 points)

- Write the Search( ) method in TREE class. It should run in O(h); it should return a Boolean. (10 points)

- Write the BFS_Print( ) method in TREE class. This should print the values of the tree in BFS order. A correct Queue class from Program 1 has been provided to assist you in writing the BFS method. You will need to add the print statement in the correct location yourself (it will not be provided as in Program 1). It should run in O(n); it does not have a return value. (10 points)

- Write the Find_Value_Of_Rank method in TREE class. This method takes an integer as input (you may assume the input is always an integer); it returns the value of the element with that rank in the tree. Some sanity checking is provided in the code to guarantee that the rank is valid. It should run in O(h); it returns an integer for valid ranks, otherwise None. (20 points)

- Write the Get_Rank_Of_Value method in TREE class. This does the reverse look-up as the last one; given a value, it returns the rank of the value within the tree, if the value is present (otherwise return 'None.') Be careful with how you design this method- it is difficult (and maybe impossible) to try to first directly call or copy the code of Search, as there is probably a calculation we want to run within the search process such that when we find a value, we know how many values are smaller than it. It should run in O(h); it returns an integer for existing values, otherwise None. (20 points)

- Write the Node_Count method in TREE class. It should return the number of elements in the tree. It should run in O(1); it should return an integer. (10 points)

- Write the Set_Predecessor( ) method in the Tree_Node class. (The Set_Successor function is given, these two will have a good amount of symmetry.) It should run in O(h); it does not have a return value. (5 points)

- Set the initial value for 'self.RankCounter' field in the Tree_Node class constructor. (5 points)

- **CODING RESTRICTIONS.** Part of the objective of this assignment is to build fundamental data structures, not to use language built-ins. You may not use the built-in functions list, tuple, dict or similar; you may not use the built-in string functions split, append, or similar. You may not use the built in list or dict constructors as [ ] or { }. In fact, you should not use characters [ ] or { } anywhere. Submissions might be uniformly searched by program or manual check for these prohibited uses of Python. If you're not sure if you can use something or not, please ask. See the course syllabus for further explanation on restrictions.

- You can check your answer by running the test case given in __main__ block against the output below.

- Submit your project solutions via Blackboard.

Method-Writing "Check List"
Always, when you write methods, be mindful of:

- Invariants; if you are changing the data structure, be sure to fix up its pointers and fields so that future methods can be called correctly
- Boundary conditions; are you removing something from an empty data structure, or do you need more memory?
- Pointer order of operations; if you have to give new assignments to two pointers, does the order in which you do it matter? One might be your only pointer to a part of your data structure.

Python Tips
Formatting. Remember, Python does not requires brackets as { }, or end-of-line signifiers as ;. Instead, it uses strict indenting to delineate blocks within classes, methods, conditionals and loops. By default, lines end where the text ends; this can be subverted by enclosing a multiple-line expression in parentheses ( ) such that it is "open" when the first line ends.

Object Types. You should not expect any problems with Types in this project, but you should be aware of them. Python is an "un-typed" language. This means that you do not have to declare the types of variables before assigning them, and you will not see Type declarations in the template. This has both benefits and drawbacks. The benefit is that a variable's type can be dynamically changed by simply assigning it to anything else as needed. (We used this in Program 1 when we let the value of a List_Node be either a string or a pointer to an embedded list!) The drawback is that if you do this by accident, Python will not throw you an error as a red flag. Casting is sometimes necessary (usually between string and "number" types).

Classes. You do not need to design any classes from scratch for this assignment but do need to know how to navigate them, and how to implement methods. When the class is declared, inheritance is designated in parentheses immediately following the class' name. Constructors are located within "__init__" methods. All class methods, when they are declared, are required to explicitly identify a pointer back to the current object in the first place of its input list. By convention this variable is named "self". When class methods are called, they do not pass a value into "self". So if a method is declared as MethodA(self, v1), it is called with just MethodA(v2_into_v1). Within methods, class field FieldB should then be accessed using self.FieldB. For examples of these, see the template.

Testing

The template comes with test code provided within the __main__ function. This is to help you test your code, but it will not be the input used for final grading. It might not even be all of the functions used for final grading (I might write a DFS traversal

as an additional way to check the arrangement of your tree). So just because your output matches does not guarantee that your code is flawless.

(This is just a sample of tests you might run. I repeat: I cannot guarantee that because your code reproduces the given test output, that it does not have other bugs.)

Please remember, do not change any of the print statements or __str__ methods (which tell objects how to "print" themselves") within classes. Then it should result in the following output to the screen.

Found all inserted values
No bad values found

run 1
1063
744
2121
240
897
1613
9020
194
401
1442
1961
9015
9997
349
1444
1634
2093
6802
9640
1465
1805
6171
8068
9498
9990
4780
6799
7617
8986
9112
2816

5772
6500
7159
8020
8609
2726
3504
5639
5884
6224
6625
6884
7588
7758
8359
8735
2571
3016
4215
4805
5734
6047
6599
6713
6833
6886
7548
7673
7759
8083
8611
8776
2302
2663
3134
3979
4458
5250
6552
6612
6642
6779
6906
7487
7813
8267

8777
2322
2612
3091
3306
4074
4313
4743
5032
5417
7817
8240
3366
4016
4681
5092
5338
5578
8117
3336
4569
5085

Min:
194

[[[First set of searches should all succeed, second should all fail. Run 1 was BFS. Min is 194. Then: Runs 2 and 3 should be the same (Run 2 uses Python to sort the list and prints); and both should be the same as the beginning of Run 5. Run 5 calls the min value of the tree and then prints successors in order. Run 4 searches for v and returns the rank of v if it exists in the tree. For example, 240 is the 2nd smallest key, so its rank is 2; 4780 is the 40th smallest key, so its rank is 40.]]]

run 4
0; None
20; None
40; None
60; None
80; None
100; None
120; None
140; None
160; None
180; None
200; None

220; None
240; 2
260; None
280; None
4720; None
4740; None
4760; None
4780; 40
4800; None
4820; None
4840; None
4860; None
4880; None
4900; None
4920; None
4940; None
4960; None
4980; None