CS2121B - Assignment 4 - 2020

Binary Search Trees: Efficient Insertion and Search

Published: March 11th, 2020

Due date: March 29th, 2020 at 11:00 pm via OWL

Note Well:

You must take note of the following:

- You are given two (2) .py files that you are to complete. For your work with these files take note of the following:
  - Your function names, along with the parameters they take **must** remain the same; **you cannot alter these**. If you alter these, you will lose significant marks and risk getting a mark of zero.
  - A file called `test.py` will be used by the TA to mark your assignment – I have provided you with the file we will actually use to test your code. You are free to use this on your own code.
  - Please **do not** prompt the user for input; all functions will be called by just calling the functions.
- Some of the descriptions are deliberately left somewhat vague. This gives you practice interpreting design instructions, but things will become clear as you get further on with your implementation.
- If there is something you are not sure how to do, you should first Google it or look it up in the Python documentation online: `https://docs.python.org/3/index.html`

Purpose

The purpose of this assignment is to continue to familiarize yourself with writing your own classes, using them, and learning about data structures specialized for search. In particular, in this assignment you will familiarize yourself with binary search trees and gain further experience of how to interpret functional requirements (specification of the behaviour of software) and how to develop your implementation to meet them.

Summary

You will be writing a binary tree class that is specialized for efficient search. This class will be called `BinarySearchTree`.

Provided:

You are provided with a testing script (`test.py`) and two incomplete python scripts (`BinaryTreeNode.py` and `BinarySearchTree.py`) that you are to finish.

BinaryTreeNode class

The `BinaryTreeNode` class specifies the content and behaviour of a single node in a binary search tree. This class will have three (3) attributes: `data`, `left` and `right`. You will need to complete two (2) functions:

1. `__init__(self, data=None)`
    - This is the "constructor", which will set the three attributes of the class.
    - Implement the initializer so that we can optionally pass some `data`, as indicated
2. `__str__(self)`
    - This function will return the string version of the data.

There is nothing else to the `BinaryTreeNode` class.

BinarySearchTree class

Unlike the `BinaryTreeNode` class, the `BinarySearchTree` class will manage an actual tree and provide the functionality for it that we want. What the various functions need to do should be clear from their names, their descriptions and what we covered in lecture.

Note well: Some things to watch out for:

- Your search trees can have multiple items with the same data. Smaller values should appear to the left, and greater or equal values to the right.
- Except for two functions, no loops are allowed. You need to use recursion.

This class has two (2) attributes: `root` and `count`, which should be self-explanatory. You will need to complete 15 functions:

1. `__init__(self, head)`
    - The initializer will define the two attributes of the class.
    - It is possible that we can pass a `BinaryTreeNode` to the initializer which has links to children. If we do this, we need to make sure we count how many things are in the tree.
2. `__str__(self)`
    - Return a string representation of the inorder traversal of the tree (so just call the `inOrder` traversal).

3. `insert(self, data)` **and** `recursiveInsert(self, data, node)`
    - The `insert` function is already complete. It just calls the recursive version of insert that you will write.
    - For `recursiveInsert`, **recursively** insert the new piece of data into the binary search tree.
4. `search(self, data)` **and** `recursiveSearch(self, data, node)`
    - The `search` function is already complete. It just calls the binary search on the root.
    - For `recursiveSearch`, **recursively** implement a binary search for the data and, if it exists, return a reference to the node containing the thing we are looking for, otherwise return `None`.
5. `count(self, data)` **and** `recursiveCount(self, data, node)`
    - The `count` function is already complete.
    - For `recursiveCount`, **recursively** count the number of items in the tree.
6. `depth(self)` **and** `recursiveDepth(self, node)`
    - The `depth` function is already complete.
    - For `recursiveDepth`, **recursively** find the depth of the tree.
    - **Note well:** an empty tree is defined to have a depth of -1 (negative 1).
7. `findMax(self)` **and** `recursivefindMax(self, node)`
    - The `findMax` function is already complete.
    - For `recursiveFindMax`, **recursively** find and return a reference to the node with the largest value in the tree.
8. `findMin(self)` **and** `recursivefindMin(self, node)`
    - The `findMin` function is already complete.
    - For `recursiveFindMin`, **recursively** find and return a reference to the node with the smallest value in the tree.
9. `validate(self)`
    - This function returns a Boolean telling us if the tree is in fact a binary search tree (`True` if it is, `False` if it is not).
    - **Note well** you may use a loop for this.

10. `inOrder(self)` **and** `recursiveInOrder(self, node, dataStringList)`
    - The `inOrder` function is already complete.
    - For `recursiveInOrder`, **recursively** recursively go through and add the string version of the nodes to the list `dataStringList`.
11. `preOrder(self)` **and** `recursivePreOrder(self, node, dataStringList)`
    - This is the same as `inOrder` but for a preorder traversal of the tree.
    - Make sure that `preOrder` returns the list.
12. `postOrder(self)` **and** `recursivePostOrder(self, node, dataStringList)`
    - This is the same as `inOrder` but for a postorder traversal of the tree.
    - Make sure that `postOrder` returns the list.
13. `levelOrder(self)`
    - This function returns a list of the elements in the level order traversal of the tree.
    - **Note well:** this method can use a loop, which is strongly recommended.

Questions

You need to answer a couple of questions as part of this assignment. Make sure you answer the following questions as **numbered** comments in your `BinarySearchTree.py` file.

1. If your binary tree is an expression tree, what types of expressions do the inorder, preorder and postorder traversals produce? Hint: think back to the lecture on stacks and consider a simple example (such as the expression tree for $A + (B * C)$).
2. What is the (order of the) time complexity of the search and insertion operations for your binary search tree?

Submitting the Assignment

You will submit the files via OWL, under the assignments tab. **Only submit your complete python files as text files** (`BinaryTreeNode.txt` and `BinarySearchTree.txt`) via OWL. Please follow the instructions carefully and do what is asked, but not more, to ensure that you do not unnecessarily lose marks. Make sure not to modify the function names.