Binary Search Tree Functions

Homework #8
By
John Robertson
CS 303 Algorithms and Data Structures
October 18, 2019

1. **Problem Specification**

This assignment requires implementing a binary search tree and some functions associated with it, as well as testing it on a real-world application.

2. **Program Design**

This program has two files: a file with a node and binary search tree class (including), and a driver file.

The following steps were required to develop this program:

- Implement the Node and Tree classes
- Implement tree_insert(), inorder_tree_walk(), and tree_search()
- Debug the above until they worked, writing a mini-driver within the file
- Write the driver
- Debug the driver

I made it an option to build a tree on the large sorted file.

3. **Testing Plan**

I first had to test tree_insert(), then inorder_tree_walk(), then tree_search(). inorder_tree_walk() required the most debugging. I used small inputs for testing, knowing that I'd have bigger inputs to test on from input.dat and UPC.csv. Next, all I had to do was test my driver code itself in *driver.py* for correctness using input.dat.

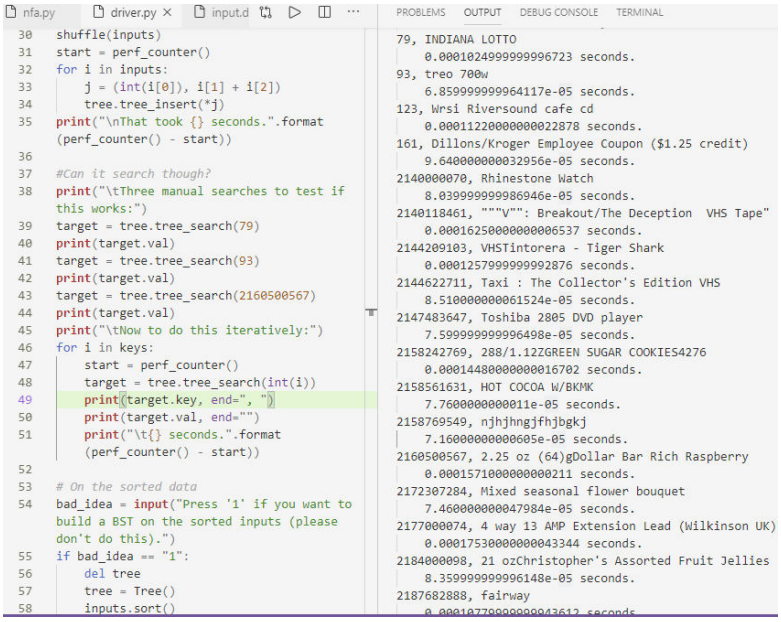
4. **Test Cases**

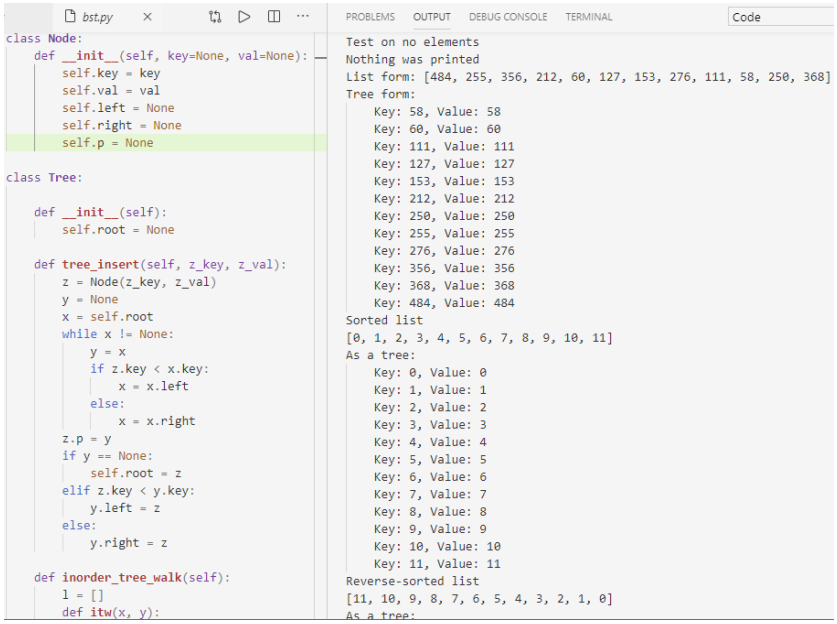
You should run *bst.py* and then *driver.py*. *bst.py* tests inorder_tree () on no elements, a random list, a sorted list, and a reverse sorted list: all the while, tree_insert() is by necessity tested. tree_search() is tested on a random item that is guaranteed to be inside it and an item that is guaranteed to not be inside it. *driver.py* tests on tree_insert() and inorder_tree_walk() on input.dat before testing the time that it takes to use tree_insert() on a shuffled UPC.csv. It then tests three manual searches on keys from input.dat on the tree with the items from UPC.csv before testing every key from input.dat. There is also an optional test case for inserting all the *sorted* data from UPC.csv in a tree, but this is obviously not recommended.

5. **Analysis and Conclusions**

Searched Key	Expected Outcome (s)	Time Took (s)	Done as expected?
79	0.001	0.0001	No (Faster)
93	0.001	0.00007	No (Faster)
123	0.001	0.0001	No (Faster)
161	0.001	0.0001	No (Faster)
2140000070	0.01	0.0001	No (Faster)

2140118461	0.01	0.0001	No (Faster)
2144209103	0.01	0.0001	No (Faster)
2144622711	0.01	0.00009	No (Faster)
2147483647	0.01	0.00008	No (Faster)
2158242769	0.01	0.0001	No (Faster)
2158561631	0.01	0.00008	No (Faster)
2158769549	0.01	0.00008	No (Faster)
2160500567	0.01	0.0002	No (Faster)
2172307284	0.01	0.00007	No (Faster)
2177000074	0.01	0.0002	No (Faster)
2184000098	0.01	0.00008	No (Faster)
2187682888	0.01	0.0001	No (Faster)





The code turned out faster than expected. Obviously, it is slow on the sorted list (I was unwilling to see how long this would take, given past experience with code that takes a long time to run), but the code looks up based on keys very quickly, because the search time is at worst only the log of the height of the tree. It is clear that binary search trees are one fast option for storing large amounts of data.

6. **References**

Dr. Unan's slides