Kristian Snyder  
Data Structures CS2021 Section 001  
Prof. Charles Zimmer, Prof. Karthik Vadambacheri  
Lab 10

Simplicity is the goal of this lab. I would like to say that I engineered a very efficient solution that makes as few operations as possible and approaches STL implementation for trees, but I didn't. However, readability was very important when creating a b-tree. The algorithm's complexity is such that the number of operations to insert one item is far too many for me to hold in my mind at one moment.

Therefore, creating repeatable operations and even breaking out sections of code that are only used once into separate functions helps add a layer of abstraction. When debugging, I can then consider whether the set of operations that splits a node is incorrect, instead of the set of operations that takes a node and an item, compiles them into two separate arrays, creates two nodes to store each side in, and replaces the original node with those.

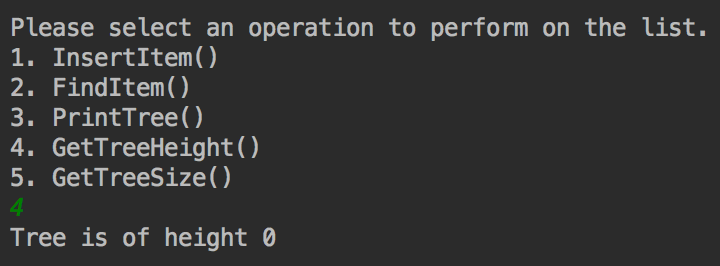
This prevents stepping through a monolith of code; if I'm reasonably sure that finding an item in the node is working correctly, I can pull that out into a separate function and step over it there. More than ever, good code design and clear commenting is what determines success on this lab. When working in industry, almost every problem will be this complex. Perhaps it will not have more difficult algorithms, but the processes of applying various business rules and sending data between systems looks starkly different when reading a 500 line function compared to ten fifty-line functions.

While implementing a b-tree is a small project compared to those completed in industry, many of the same principles apply and a good design initially can result in enormous benefits later. Planning, then, is extremely necessary. The amount of time debugging can be mitigated by a robust understanding of the problem, which allows for more time to be spent on other things, such as additional features of the product.

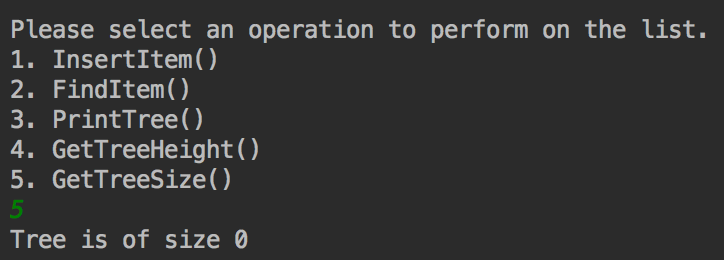
## Task 2

Other than reusing my previous code from Lab 8 that accepted input from the user to interact with a sorted list, I had to make very few changes. The most important was making sure that the printing, size, and level functions worked correctly when the list was empty, since my tree always has a root node, even if no items are in the list. All other functionality worked as-is, once the user input was retooled.

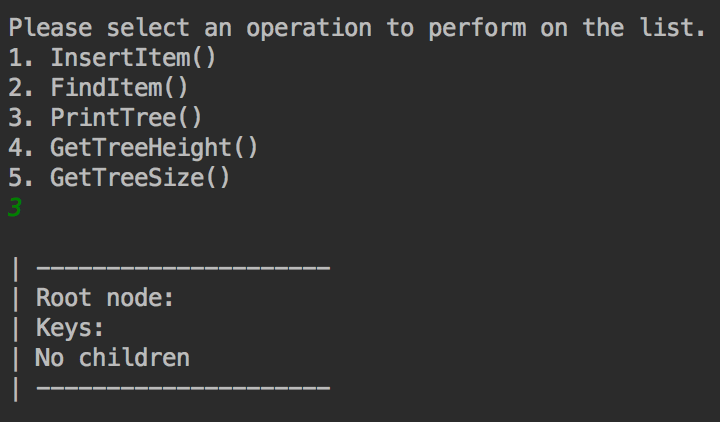
### Tests



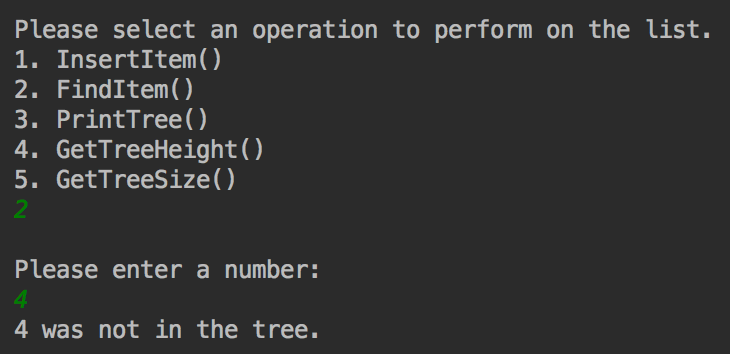
Height of empty tree



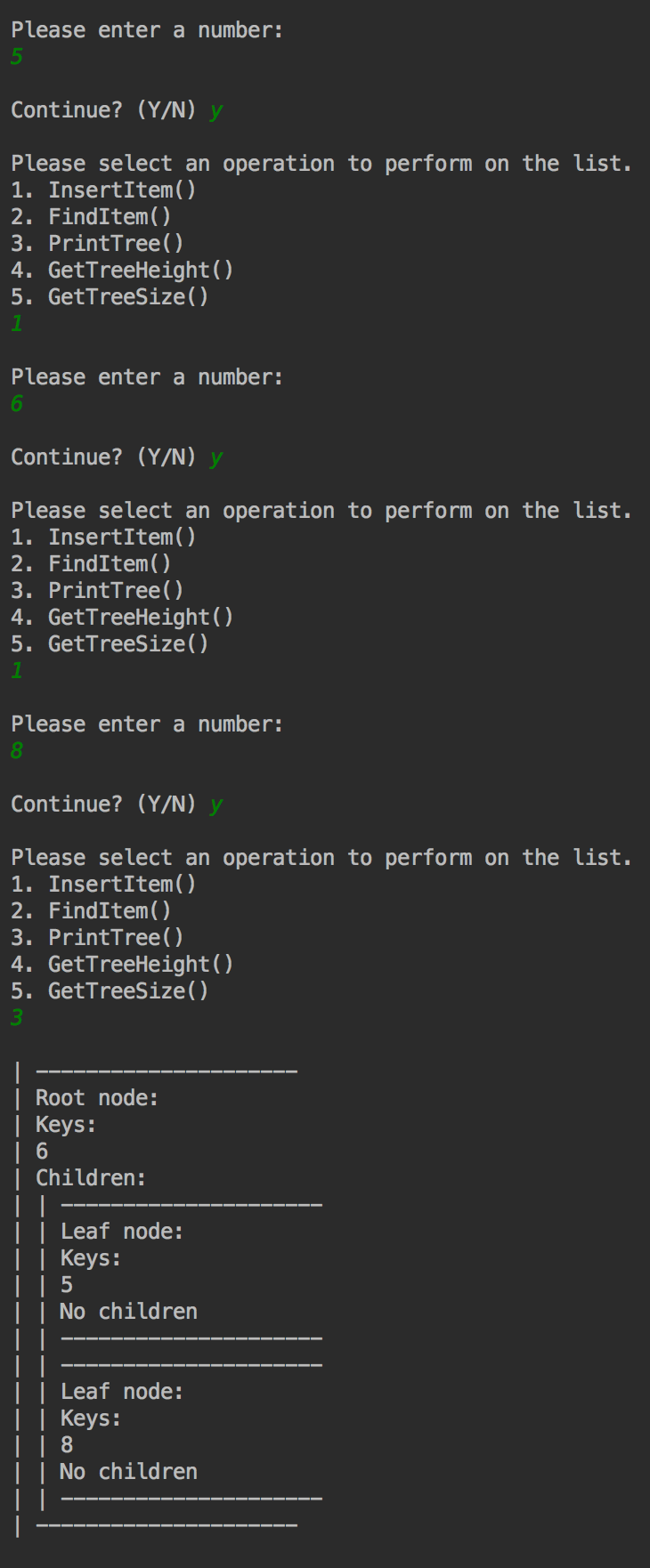
Size of empty tree



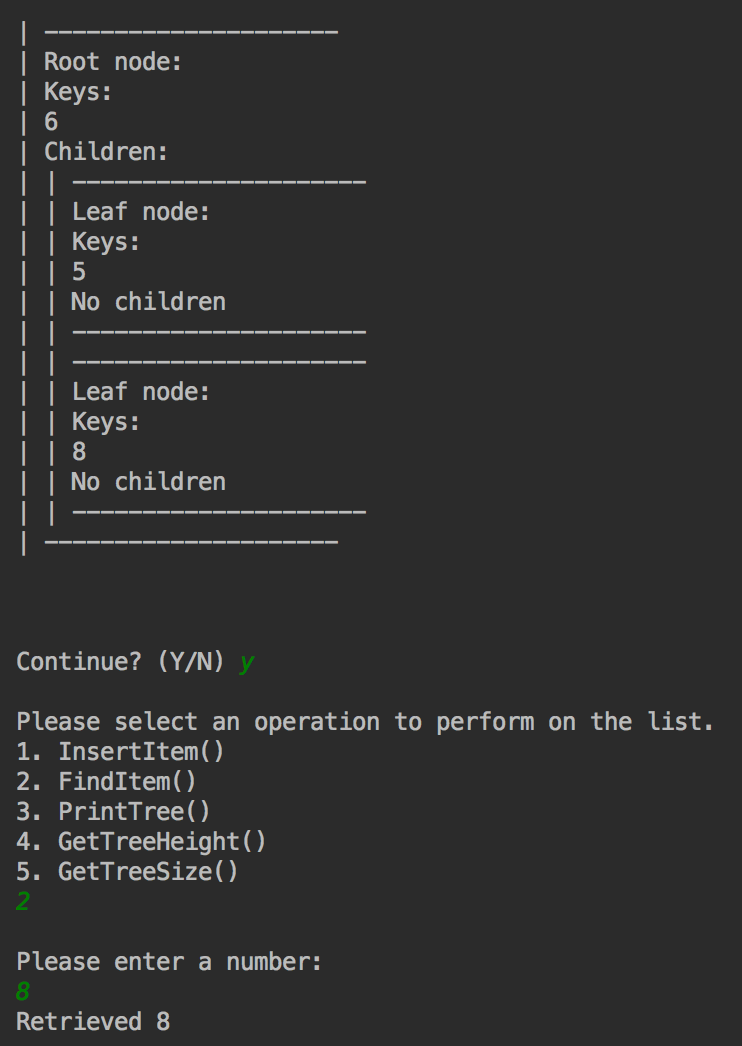
Print empty tree



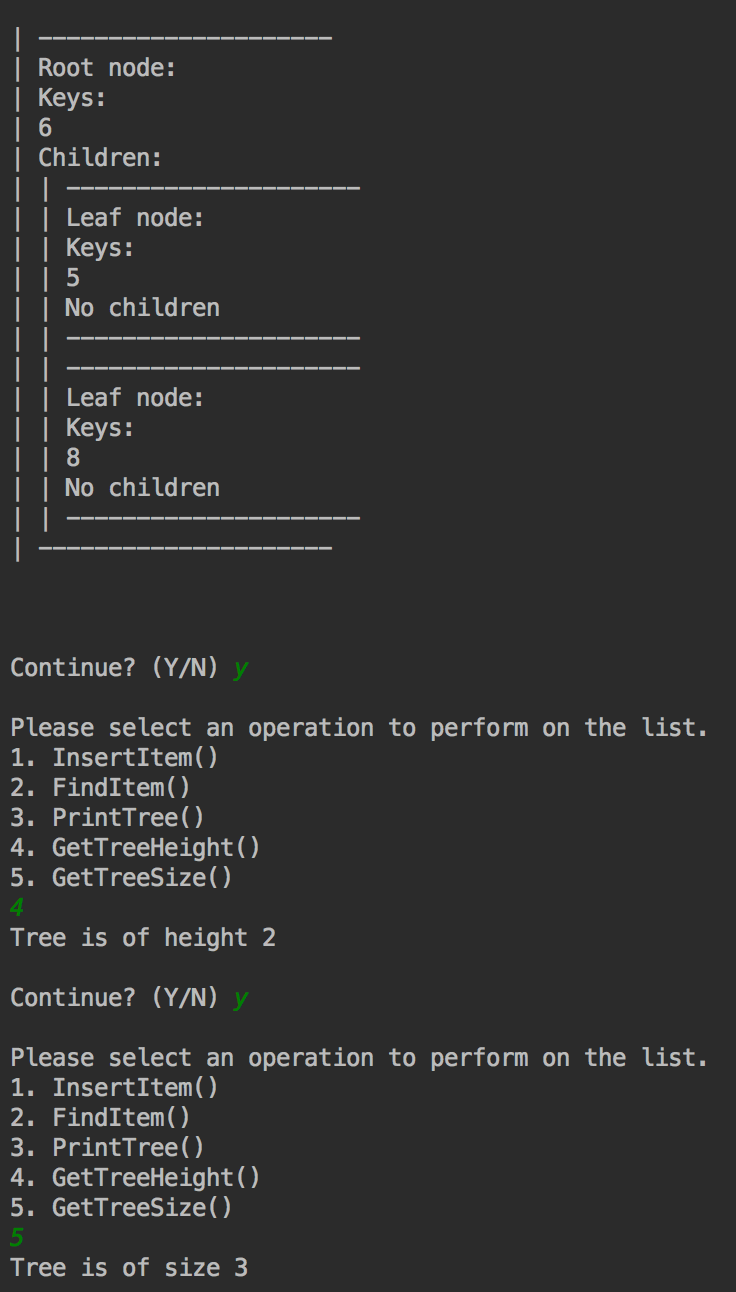
Attempting to find item not in tree



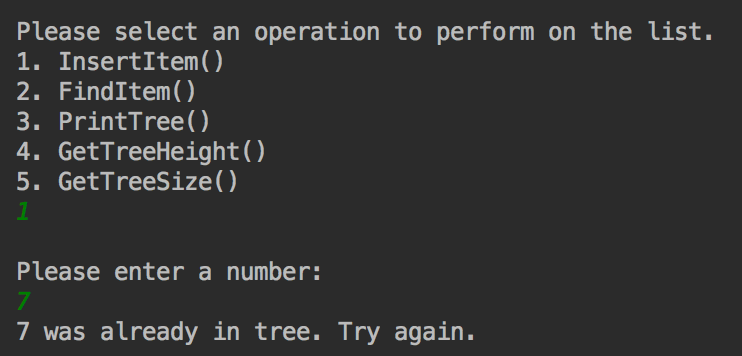
Adding items to the tree and printing them



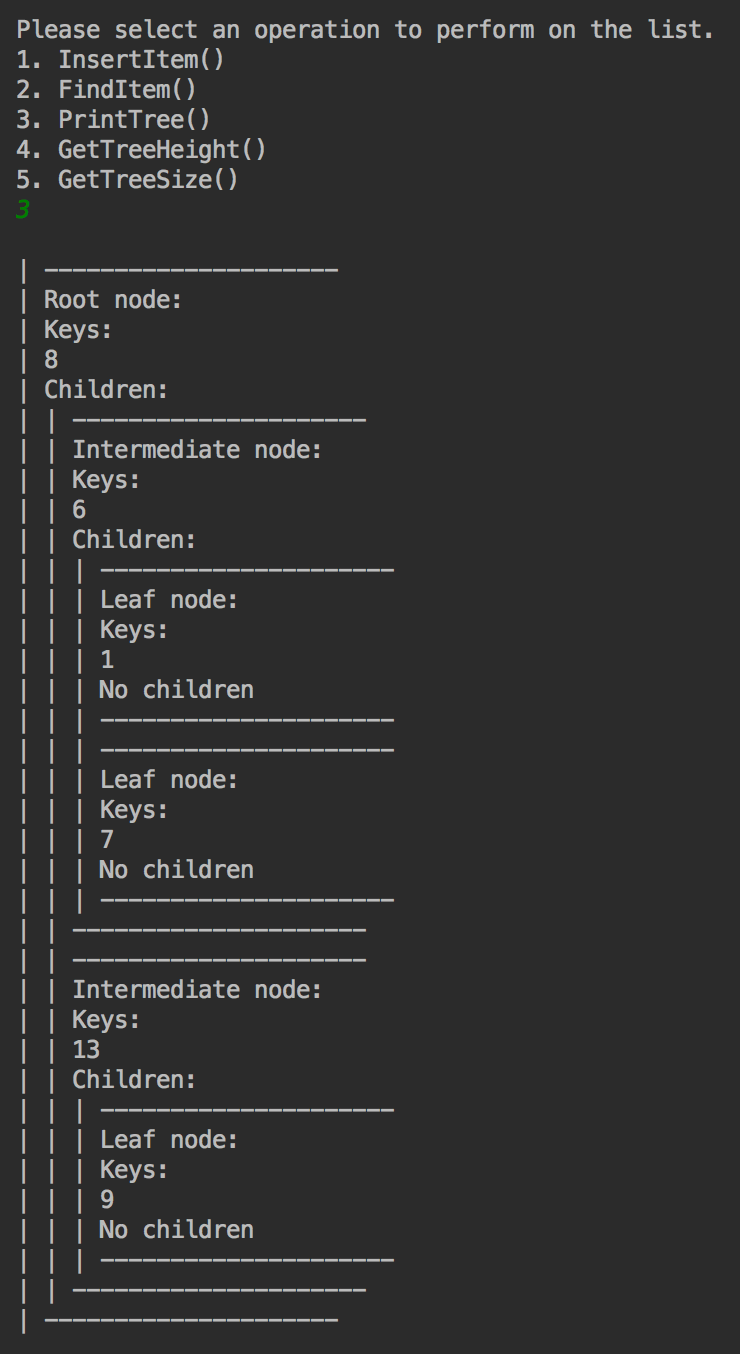
Retrieving item in tree



Getting size and height of tree



Attempting to add item already in tree



Printing a complex tree

## Task 3

Task 3 required a small change to the b-tree class to track moving between nodes and also necessitated reusing the code from the previous lab about hash tables. Adding a function to retrieve the counter function and reset it was trivial, while adding another object to monitor with the loop of adding and retrieving students was a little more complicated since it also required reformatting the printing at the end to capture the b-tree of degree 5.

I did have to change the Student class to overload both the <= and >= operators, since those are both utilized in the b-tree class and so were resulting in incorrect insertion and data loss due to pointer comparison for both of those operators.

### Test results

For each of the four trials of adding the same 50 random students to three b-trees of different degrees, I was not surprised to see these results. Increasing the degree reduces the number of moves, which results in fewer nodes being traversed. However, not shown here are the additional comparisons necessary to find the correct child to move to as the function walks down the tree, which can result in an increased number of operations as degree increases. While I do believe that this conclusion is correct, further investigation would be a way to continue evaluation of different b-trees.

While the logic above is certainly valid, a b-tree should not be considered as a data structure of its own merit, especially in comparison to a binary search tree. B-trees commonly have large advantages over other tree types due to larger node size in certain cases. One of the most significant is with respect to external storage; most disks require reading data in large blocks, with the same time required to read a block no matter how much of the data is actually required. Filling the block as much as possible, then, can increase performance from fewer disk accesses to retrieve the same amount of data. Investigating disk performance with different degrees of b-trees would be, I imagine, a more accurate pursuit to examine their real-world performance.

|  |  |  |  |
| --- | --- | --- | --- |
| Trial | Degree 3 Moves | Degree 4 Moves | Degree 5 Moves |
| 1 | 274 | 228 | 198 |
| 2 | 266 | 232 | 178 |
| 3 | 251 | 225 | 178 |
| 4 | 250 | 233 | 176 |