CSCI 241 Data Structures

Project 5: AVL Trees

Team Palkia

Introduction

In project 4, we implemented a basic unbalanced binary search tree. This was done recursively and after the implementation, we tested our BST with an in-order, pre-order, and post-order traversal. In project 5, we build upon project 4 by improving our BST implementation to result in O(log n) insertions and removals. We do this by balancing the tree each time we insert and remove an object.

Balancing the Tree

A private method called __balance(t) was written to do this objective. __balance(t) takes a node t and checks if t has a balance factor indicating a rotation is necessary. We reasoned that there were four kinds of rotations to check for – a single left rotation where the subtree is right heavy, a single right rotation where the subtree is left heavy, a double left rotation (or a LR rotation), and a double right rotation (or a RL rotation). Because we call balance on our return in our recursively written insert and remove functions, we do not need to worry about the levels "deeper" than t – they should already be balanced. As a result, we simply check to see if the subtree is unbalanced, perform the appropriate rotation, and return the root of the newly balanced subtree. This is a constant time operation, O(1). We also update the height attribute of each node on the path where insert/remove occurred and nodes involved in rotations to ensure the heights are accurate. Then to finish our work in the Binary Search Tree class, we create a new method for constructing a Python list of values in the tree. This method functions as the recursive in-order traversal method does, with the only difference being its output type. That is to_list will return a python list instead of a string. We expected performance O(n^2) because our _to_list_helper (the private method companion to to_list) must recursively extend a list of n objects. Extending a list requires looking at each element n of the list. We do this n times for n elements, and therefore we expected O(n^2).

The Fraction Class

The Fraction class we implemented was mostly completed as a provided skeleton file. All we had to do here was implement the comparison operators (less than <, greater than >, and equal to ==). The in our main, we created an array of somewhat random fractions and inserted these into an empty AVL tree. We printed the array before insertion into the AVL tree, and afterwards, we used to_list to print the in-order representation of the tree in Python list format. This verifies that our AVL tree works and has sorted the fractions properly. Our fraction test used fractions to confirm this but we also included two simple tests using integers to further verify the correct sorting of the fractions.

Testing

We repeat the basics tests done in project 4 such as checking an empty tree, checking insertions, and checking removals. After confirming this, we then check to make sure that the three traversals (in-order, post-order, and pre-order) can identify a unique binary search tree. Our tests here were based on the in-class lecture examples of tree traversals to verify this. These new, more comprehensive tests also verify that the heights of our nodes are updating correctly.

Worst-Case Performance

Before we considered the performance of the insert_element method to be log(n). This is because of a log(n) look-up time, where we keep "splitting" the tree into a smaller and smaller structure until we find the value or reach a leaf. With the addition of invoking the balance method with each successful insertion, we can still expect log(n) performance in the worst case as the balance method itself is constant time.

This is also true for the remove method. That is, the remove_element method calls the private method_removehelper. In this private method, we look for the value we want to remove first. This look-up is a log(n) performance. If the value we want to remove is in a node that contains no child or one child, then the remove is constant time; if the value we want to remove is in a two-children node, then we need to find the smallest value in its right subtree, which again is log(n) performance. All considered, we still have a worst case of log(n) performance. As noted earlier, balancing the tree as we move up recursively after a removal does not increase this as the balance method is constant time.

In project 4, get_height method was constant time. It is still constant time here, since all we are doing in this method is extracting an attribute from the root. The in-order, pre-order, and post-order are traversal methods that visit every node of the tree by recursively breaking up the tree into subtrees and looking at the values in the order demanded. Because AVL trees require balance, the number of nodes visited will result in log n performance time. It seems rather obvious now that balancing with each recursive insertion and removal allows the AVL tree to be an efficient data structure with log n performance for traversal, insertion, and removal (as the subtree heights cannot differ by more than one node). In comparison, a simple, unbalanced BST could have a worst case of linear performance O(h), with h as the height of the tree.