CSCI 241 Data Structures

Project 4: Less Left, More Right

Team Palkia

Introduction

This project implements a basic unbalanced binary search tree. We did this recursively and after implementation, tested an in-order, pre-order, and post-order traversal of the tree.

Test Case Analysis

We first test an empty tree to check if it is nothing and especially if height is 0, which is a dangerous case. Then we test the basic function of inserting one element into the tree, to see if the element (and the node that contains it) is truly inserted and the height is increased by 1. Then we test if remove one element is working. We conduct some more comprehensive tests afterwards.

We build a three-element tree with height 3 and another with height 2 and check if in-order, pre-order and post-order work well. Then we remove leaf, which has no child, then root, which has two children, to check the remove function. And we check the two "raise value error" cases, which are insert a value that already exists and remove a value that does not exist. Then we conduct an even more comprehensive test with 8 values and height 4, in which there are nodes with two children, nodes with one child, and node with no child. After checking the whole tree works (get_height and the three _orders), we remove a node with no child, a node with one child, and a node with two children. Then another test remove several values in a row to check it works constantly. Finally we use the example shown in class to conduct a final check.

Worst-Case Performance

The insert_element method calls the private method _inserthelper. In the private method since we try to locate where we should put the value, the performance is log(n).

The remove_element method calls the private method_removehelper. In this private method, we look for the value we want to remove first, whose performance is log(n). Then 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, and since the remove must be the case of removing a node with one or no child, the remove is constant value. So in this case the performance is $(\log(n))^2$.

The get_height method is clearly constant time, because we are just extracting an attribute from the root.

In-order, pre-order, and post-order are traversal methods that visit every node of the tree, and therefore are O(n)