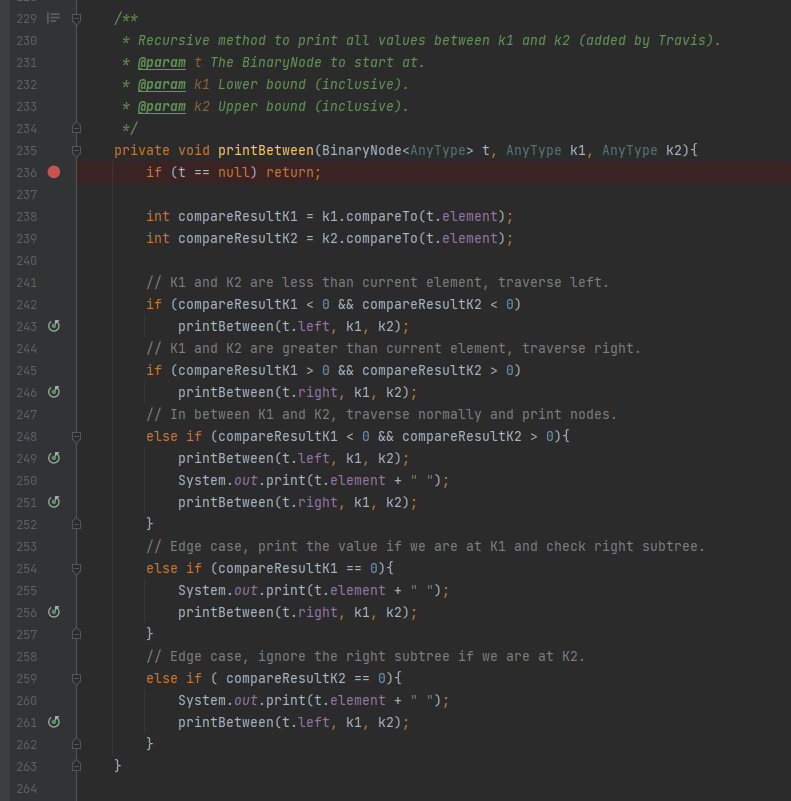
Travis Munyer

8/3/21

Programming Assignment #2 Complexity



This algorithm is the one used for the printBetween operation. Due to the difficulty of this recursive algorithm, computing a recurrence relation and providing an analysis using this method is difficult and out of the scope of this course. To provide an analysis for this algorithm, I will break the problem into cases and provide an intuitive-based analysis similar to the analysis’s provided for binary search trees in lecture. As discussed, we assume that the BST is balanced for this analysis. If the tree is unbalanced, the worst-case complexity would be O(n + k).

This algorithm handles three different cases where the location of k1 and k2 is different. As in the assignment doc, let k be the number of nodes in between k1 and k2 (both inclusive).

Case 1, k1 is to the left of the root and k2 is to the right of the root or either k1 or k2 is the root:

This is the best case for this algorithm (especially when either k1 or k2 is the root). This case occurs mainly in lines 248-251. The algorithm does not have to traverse the tree until it finds a value in between k1 and k2 as done in case 2 and 3. In this case, the algorithm will perform constant operation k times. This is because the traversal will stop (and check the right subtree since these values are larger and included in k1 through k2) when hitting k1 as shown in lines 254 to 256. Likewise, the traversal will stop and check the left subtree when hitting k2 as shown on lines 258-262. Therefore, this case is O(k) complexity.

Case 2 and Case 3, k1 and k2 are entirely on the left (case 2) or right (case 3) subtree of the root:

This is the worst-case for this algorithm. These cases are symmetric, so the analysis is provided together. Since we are assuming the tree is balanced, the height of this tree is log(n). We traverse down the tree where the maximum traversal is the height - 1 (log(n) – 1, since k1 cannot equal k2 in this algorithm.) as shown on lines 241-243 (case 2) or lines 244-246 (case 3). Once either k1 or k2 is found (based on the case) we perform the case 1 algorithm. Therefore, the algorithm performs at most log(n) + k - 1 operations, which is O(log(n) + k) complexity.