CS5008-HW07-BST

1. What does it mean if a binary search tree is a balanced tree?

   A balanced binary search tree means that height difference between left and right subtrees for every node is not more than 1. This ensures tree not too skewed and searching, insertion, and deletion operations perform efficiently.

2. What is the big-Oh search time for a balanced binary tree? Give a logical argument for your response. You may assume that the binary tree is perfectly balanced and full.

   For balanced binary tree, big-Oh search time is O(log n). In balanced tree, each level has twice nodes as previous level. So, total nodes n can be represented as n = 1 + 2 + 4 + ... + 2^(h-1), where h is height of tree. This sum is 2^h - 1, so n = 2^h - 1. Solving for h, we get h = log(n + 1). In terms of big-Oh notation, search time is O(log n) because search eliminates half of remaining nodes at each step.

3. Now think about a binary search tree in general, and that it is basically a linked list with up to two paths from each node. Could a binary tree ever exhibit an O(n) worst-case search time? Explain why or why not. It may be helpful to think of an example of operations that could exhibit worst-case behavior if you believe it is so.

   Yes, a binary search tree can exhibit O(n) worst-case search time. This happens when tree becomes very skewed and looks like a linked list. For example, inserting elements in sorted order creates a tree with only right children (or only left children if reverse sorted order). In this case, searching for element at end of list takes O(n) time as we traverse entire tree, which is same as linear search in a linked list.

4. What is the recurrence relation for a binary search? Your answer should be in the form of T(n) = aT(n/b) + f(n). Clearly state the values for a, b and f(n).

   For binary search, the recurrence relation is T(n) = T(n/2) + O(1). Here, a = 1, b = 2, and f(n) = O(1).

5. Solve the recurrence for binary search algorithm using the substitution method. For full credit, show your work.

   T(n) = T(n/2) + O(1)
   Let n = 2^k, then T(2^k) = T(2^(k-1)) + O(1)

   Apply substitution repeatedly:
   T(2^k) = T(2^(k-1)) + O(1)
   = T(2^(k-2)) + O(1) + O(1)
   = T(2^(k-3)) + O(1) + O(1) + O(1)
   ...
   = T(2^0) + O(1) + O(1) + ... + O(1) [k times]

   Total O(1) terms are k, so T(2^k) = T(1) + k * O(1). Since n = 2^k, k = log(n). So, T(n) = T(1) + log(n) * O(1), which simplifies to T(n) = O(log n).

6. Confirm that your solution to #5 is correct by solving the recurrence for binary search using the master theorem. For full credit, clearly define the values of a, b, and d.

   Using master theorem, for recurrence relation T(n) = aT(n/b) + O(n^d), we have a = 1, b = 2, and d = 0.

   According to master theorem:
   If a > b^d, T(n) = O(n^(log_b(a))).
   If a = b^d, T(n) = O(n^d * log(n)).
   If a < b^d, T(n) = O(n^d).
   In our case, a = 1 and b^d = 2^0 = 1. Since a = b^d, we use second case of master theorem: T(n) = O(n^d * log(n)). As d = 0, n^d = n^0 = 1, so T(n) = O(log(n)). This confirms solution from #5 is correct, as both methods give same result: T(n) = O(log n) for binary search.