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

The depth of two subtrees of every node doesn't differ by more than 1.

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.

   O(logn), traverse the tree by either go left or right, eliminating half of the tree for each step. if unbalanced, it's O(N)

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, for totally skewed binary search tree, it needs to traverse the whole tree to search for certain node. for those kinds of unbalanced binary search tree the worst-case is O(n)

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)*.

T(n) = T(n/2) + 1, it prunes the given sequence into half each time and one comparison for each iteration. a=1, b=2, f(n) is constant.

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

$$T(n) = T\left(\frac{n}{2}\right) + 1, n = 2^k, T(1) = 1, then\ T(n) = T\left(\frac{n}{2^k}\right) + 1 + \cdots + 1 = T(1) + k$$
$$= \log_2 n$$

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*.

$$a = 1, b = 2, d = 0, since\ d = 0 = \log_2 1 = \log_2 1\,, therefore\ T(n) = O(logn)$$

TEAM EXERCISE! B-Trees:

compare and contrast the use of a Binary Search Tree versus a B-Tree for indexing blocks on a disk associated with a file.

Binary search Tree: mostly used when data is stored in the RAM(small & fast) instead of disk since accessing speed of RAM is faster and CPU is connected to cache with high bandwidth channels, used for code optimization, Huffman coding.

B-Tree: mostly used when data is stored on disk(large & slow) since the height(logmN, M is the order) of B-tree is flat and more branchy thus reducing the access time, CPU is connected to disk with low bandwidth channel, can used in database management systems All of its child nodes are on the same level and can have the number of order of subtrees maximally.

Therefore when there is large dataset that may cannot fit in main memory or the number of keys is high and data must be read from disk in blocks, in this case disk access time is significantly higher compared with main memory access time therefore B-tree is used in this case since it’s fat and its height is low by achieving max keys within a node(equal to disk block size) to reduce disk access.