HW09 - BST

Prasun Surana
CS5008, Summer 1

1. In a regular binary search tree, the nodes are added in their appropriate place (smaller values to the left and larger values to the right) sequentially. This may result in very long chains of nodes which may adversely affect operational times such as lookup, insertion, deletion, etc. A balanced binary search tree is a BST where the depth of the subtrees of any node never differ by more than 1, so we would not find long chains of nodes on any one side of a balanced binary tree.

2. The Big-Oh complexity of a balanced binary search tree would be O(log(n)). This is similar to a binary search of an array, where once we determine which side the node we are looking for is on, we can 'eliminate' the other half the the tree, and this continues down the subtrees. At each level, half the (sub)tree is eliminated, so if we start with a tree with 16 nodes (let us assume they are roughly equally distributed, since the binary tree is balanced), on the next iteration, the problem would be halved to 8 nodes, then 4 nodes, then 2 nodes, then finally we find our final node which we are looking for. Hence the time complexity for a search in a balanced binary tree would be $O(\log_2 n)$. For the example above, this can be shown from the fact that it takes 4 steps to go from 16 nodes down to our 1 node that we are looking for (16 -> 8 -> 4 -> 2 -> 1). $\text{Log}_2 16$ is indeed 4, which illustrates that this is the correct average time complexity for a balanced binary tree.

3. A binary search tree can exhibit O(n) runtimes if the nodes that are added are progressively and consistently smaller or larger. For instance, if we add our first (root) node as the value 10, and then the next node to be 11, then 12, then 13, then 14, then 15, and so on, all these nodes would be added to the rightChild field of the previous node since they are larger in value (larger values in binary search trees get added to the right and smaller values get added to the left). Visually, this would create a long chain of nodes from the root to the last node, instead of a tree, which would end up resembling a linked list. If we then wanted to run operations on them, for instance bst_find(t, 15), the function would have to go through each node to get to 15, instead of it's usual method of halving the problem by comparing the value we want to the value stored in the root node. Hence, in this case, the binary tree would exhibit O(n) runtimes instead of $O(\log_2 n)$.

4. $T(n) = T(n/2) + O(1)$. In this recurrence relation, $a = 1$, $b = 2$ and $f(n) = O(1)$.

5.

$T(n) = T(n/2) + O(1)$

$T(n/2) = T(n/4) + O(1)$
$T(n) = T(n/4) + 2O(1)$

$T(n/4) = T(n/8) + O(1)$
$T(n) = T(n/8) + 3O(1)$

$T(n/8) = T(n/16) + O(1)$
$T(n) = T(n/16) + 4O(1)$

Therefore, on the k^{th} step, $T(n) = T(n/2^k) + kO(1)$.

$n/2^k = 1$
$n = 2^k$
Therefore, $T(n/2^k) = T(1)$

$\log_2(n) = k$
Substituting this into the original equation:

$T(n) = T(1) + \log_2(n)$

Since the $\log_2(n)$ term dominates in $T(n)$, we can say that the time complexity for a binary search is $O(\log(n))$.

6.

The Master Theorem is:
If $T(n) = aT(n/b) + O(n^d)$, where $a > 0$, $b > 1$, and $d >= 0$:

$T(n) = O(n^d)$ if $d > \log_b a$
$O(n^d \log n)$ if $d = \log_b a$
$O(n^{\log_b a})$ if $d < \log_b a$

For $T(n) = T(n/2) + O(1)$:
$a = 1$
$b = 2$
$d = 0$

$\log_b a = \log_2 1 = 0$

Since $d = 0$ and $\log_2 1 = 0$, by the Master Theorem, $T(n) = O(n^0 \log n) = O(\log(n))$.