Timing Analysis

	Function Tested			
Dictionary Class	Put		Lookup	
	Complexity	Time	Complexity	Time
Class 1	The complexity of the number of recursive calls in terms of number of entrees in dict (n), is $O(\log_2(n))$ -or in complexity class, O(log(n))	.0685 secs	The complexity of the number of recursive calls in terms of number of entrees in dict (n), is $O(\log_2(n))$ -or in complexity class, O(log(n))	.0201 secs
Class 2	The complexity of the number of recursive calls in terms of the depth of the tree (m) is O(m).	0.4687 secs	The complexity of the number of recursive calls in terms of the depth of the tree (m) is O(m).	1.3111 secs

All times are based on the following test done on my computer >>> testDictionarySpeeds(10000, 30000, 10100, 42) # like the above, but more lookups

For class 1, put and lookup both essentially use binary search. In put, the distinction is, if the element is not found, instead of returning "No entry", we insert a value- which takes O(n) (number of basic operations) time of n, the number of elements. However, these are not (as far as I know, recursive calls, so the number of recursive calls is still of class O(log(n)). In other words, the complexity of inserting into the list is not of recursive calls but rather of how many elements need to be shifted in memory- which may or may not be comparable to the time cost of the number of recursive calls.

Similarly, in class 2 we have an almost identical algorithm. To put, we search the tree (do a lookup) for the element to alter its value and if we cannot find it, we insert a new element where we expected to find it. At worst case, the spot is at the very bottom of the tree- which means we have m recursive calls for a tree of depth m. Since I do not have the mathematical understanding of how to relate the depth of a randomly generated tree to its number of elements, I can only say that if the tree were balanced, the complexity of the number of recursive calls in terms of the number of elements would be $O(\lceil \log_2(n+1) \rceil)$ (the function is 'ceiling-y' since a balanced tree having, say, 8 elements has a max depth of 4 even though $\log_2(9)$ is a little over 3 and rounds to 3.). I can also say, if we by some miracle of chance had a binary search tree generated in a sorted order, the complexity of recursive calls in terms of n items would be O(n).

In comparing times, the times seems reasonable. We would expect class 2 to be much slower than class 1 since class 2 is not represented with a balanced binary search tree and thus is not close to O(log(n)) of recursive calls in terms of n items. Within class 1, one would expect a faster search since 'put' is essentially a search plus an insert operation. Within class 2, I am not sure as to why lookup takes much longer. My guess is that since look ups are tested after 'puts' are tested, the 'puts' in making a binary search tree are very fast initially making the 'put' time average drop. On the other hand, I speculate, the lookups are all of a fully made binary search tree which starts out with a substantial depth. Nonetheless, the times seem reasonable.

As a last note, this timing analysis would suggest that of these two classes, if put and search are the most important operations a user needs, a user should pick the first implementation because it is faster for these two main methods.