Analysis of the Hash Table vs. AVL Tree

When comparing two data structures, it is important to retain a sense of context. For the purposes of our search engine design, the context is the need for vast amounts of quick data insertions and retrievals, where the ordering of the data is not as important as the speed at which it can be inserted or retrieved. The Hash Table implemented in our search engine design is the superior of the two structures, as it is able to perform both quick insertions and retrievals.

The basic functionality of our AVL Tree is that of a binary search tree with the ability to restructure its branches as they become unbalanced, thereby minimizing the overall insertion time of each consecutive element. The AVL Tree maintains a structure that allows its average insertion and retrieval time to be a O( log(n) ) operation. This means that for an AVL Tree will take more time for insertions the taller the tree is, though not nearly as much time as a linear structure would require. One general advantage of the AVL Tree is that its ordered structure means that the elements can be retrieved in a specific way (in the case of our implementation, alphabetical order). However, this advantage does not have much use in our design, where retrievals are generally made one at a time.

The basic functionality of our Hash Table is an array of a set size, each index containing the AVL Tree described above. Elements are placed into this array according to a hashing function (in our design, the standard string hash function modulus the number of indexes in the array), which allows elements to be inserted and retrieved in a O( 1 ) operation. Our Hash Table design does lose some speed as it fills up, as collisions in the array require a separate structure to handle (in our case, an AVL Tree), meaning that as the array fills up, these collisions become more and more common. However, this still means a significant decrease in insertion/retrieval time over an AVL Tree alone. Another thing to note is that the hashing function is a relatively slow operation, meaning that for a small number of elements, Hash Tables do not hold a clear advantage.

As stated above, our Hash Table is the superior data structure for the purposes of the search engine design. This is because the Hash Table preforms better for both insertions and retrievals, especially for large numbers of elements. The graphs titled “Hash Table Inserts” and “AVL Tree Inserts” demonstrate the differences between the two structures for insertion time. The Hash Table maintains a fairly consistent linear pattern between the number of elements and the time it takes to insert all of them. The AVL Tree graph has a slight curve to it, which is in fact quite pronounced, but is flattened by the scale. Speaking of the scale, while at low insertion counts the two structures have similar times, by the largest count the Hash Table is ahead by more than an order of magnitude, and will likely only become quicker. A more telling trend is shown in the second pair of graphs given below, showing the average insertion time for a single element (with the horizontal axis in logarithmic scale for emphasis). The Hash Table starts relatively slow, but rapidly reaches a relatively constant average time for large data sets. The AVL Tree shows the opposite trend, starting relatively low at first but increasing at an accelerating rate. This means that for large data sets the Hash Table will reach a “constant” insert time, while the AVL Tree will only continue to slow. It should be noted that, because of the initial expense of the hashing function, the AVL Tree is faster for very small data sets (under 100 elements). However, for the purpose of our search engine (requiring upwards of 74 million inserts for the wikibooks data set), this is not a relevant characteristic. When considered in the context of our search engine design, the clear superior structure for insertion and retrieval is the Hash Table.