**B-Trees B a good solution, by Andrew Taylor**

**Since this prompt says it’s based on Module 9, I looked to the data structures presented there for my options. Out of 2-3-4 Trees, AVL Trees, Binary Trees and B-Trees, actually any of them seems like they would work, because a tree can only be sorted according to one of these attributes at a time. By consequence, this means we will need 5 different trees in order for our library to be searchable by all 5 of these attributes. We can do this with Adelson-Velsky and Landis Trees, known as AVL Trees.**

**We should use AVL trees when frequent insertions and deletions are expected, because the main advantage of AVL trees is their capability to rebalance themselves after every insertion or deletion. They are best suited for scenarios like this one with frequent changes to the structure. However, we should avoid excessive rebalancing by setting the rebalance threshold at an optimal value.**

**AVL trees are binary search trees with a height balance property, and specific operations to go along with the AVL tree to rebalance the tree and perform deletions or insertions. Height balanced means that for any node the heights of the left and right subtrees differ by only 0 or 1. This is calculated via a balance factor, which is the height of the left subtree minus the height of the right subtree. To maintain balance, the left mins the right can’t exceed 1, although this may be increased to avoid constant rebalancing, in anticipation of the frequent insertions and deletions mentioned in the prompt. Presumably these 5 trees would have similar node values, because we are looking for the book or books in question per node. Maintaining separate trees that automatically rebalance is the way to the fastest search performance. The value of the each node would be a list of book objects. That is each tree has a key and the values are the other associated attributes.**

**AVL trees still must follow binary search tree rules, so the left subtree is lesser and the right subtree is greater. A key, a value, and a height value will be stored in each node, as well as the child pointers. Insertions and deletions are maintained will special operations such as rotation and removal.**

**Handling duplicate keys is going to be an important factor in all of these attributes other than the ISBN number. That is the only attribute that can be reasonably expected to be unique in all cases. Approach to this problem seem to have three major flavors, storing duplicates in separate nodes, add a counter to the node, or using a map or multiset. In the first case, every duplicate node has the same key value and is linked to the same parent node. If we choose to add a counter instead, this requires additional memory to store the counter but doesn’t affect the balancing property or search efficiency. Using a map or multiset means the value associated with the key is a collection of all duplicates. This approach provides the optimal search performance, but requires more memory than the other approaches.**

**AVL trees are best implemented iteratively, because recursive algorithms would consume too much memory. The time complexity for AVL trees in insertion, deletion, search and traversal is O(lg n) in the best, average and worst cases. However, search can reach O(1) in the best case if the target is at the top of the tree. The space complexity in O(n).**