Program #4 Review

Written by Alexander DuPree

Data Structure Performance

Program #4 utilized a binary search tree data strucute to implement a Table ADT. The binary search tree performed admirably when the data was preprocessed for a balanced insertion. However, when a data set was partially sorted, or contained repeated entries, the tree would become unbalanced and reduce runtime efficiency.

To combat this problem, I loaded the data into an array first, sorted it, and inserted the median element into the tree until the each element was inserted. This is an expensive operation, but as it only occured on initial program startup I felt the extra runtime performance was justified. After the preprocessing I was able to achieve a tree height of 10 with 490 leaves given a thousand item data set. Without preprocessing the height of the tree grew to 23, with only 133 leaves.

Data Structure Recommendation

The largest drawback of the binary search tree is it's tendency to become unbalanced. This was partially remedied by preprocessing the data; however, if the user made a number of insertions or deletions the tree would again take on an unbalanced state. Therefore, a self balancing tree like a red-black tree or AVL tree would have been a better candidate for this assignment. A red-black tree would have preserved the simplicity of a binary search tree while staying balanced at the cost of mininmal memory overhead.

Design

As far as application design, I feel that the data structures were sufficiently abstracted. A client using the application would have difficulty determining exactly which data structures were being used. The main, and other classes had no knowledge of the specifics of the Table ADT's implementation. Instead, they relied on a public interface to accomplish the goals of the program. I also abstracted a lot of the binary search tree's algorithms into higher order functions. For example, clearing the tree, counting the number of nodes, finding an element, all of these algorithms follow the same recipe: Traverse the tree and conduct an action. To prevent a lot of redundant code I created generic traversal functions that accept a function as a parameter. The traversal function then traverses each node in the specified manner and executes the provided function on each node. This helped reduce a lot of redundant code and, in my opinion, made the code more readable.

Inefficiences and Feature Additions

I wanted the extra challenge of providing iterator support for the BST. However, I think my implementation of a BST iterator could have been improved. As traversing the tree correctly required recursion, the only solution I could think of

was to create a queue of the traversal state and load pointers to the correct nodes onto the queue. And when the iterator is instantiated it would pop pointers from the front of the queue. This implementation has a few drawbacks. First, in order to create a begin iterator, the system has to traverse the entire tree first and load the correct pointers into the queue. Second, this method has a lot of memory overhead for a queue and the extra pointers associated with it. I would be intersted in developing a more performant solution for tree iterators in the future.