Program #1 Review

Written by Alexander DuPree

Data Structure Performance

Program #2 utilized two data structures. First, a circular linked list was used for a queue ADT; and second, a linear linked list of Arrays (Flex Array) was used to implement a stack ADT. For this program both structures worked sufficiently. As we were not performing any search or remove operations, we could add items to top of the stack, or push items to the back of the queue in constant time. The largest benefits of these data structures was the ability to grow and shrink our containers as needed. Whether the client pushed 1 item on the stack, or 20, the Flex Array could grow to meet the demand.

Data Structure Recommendation

For this assignment, both the circular linked list and Flex arrays worked great. However, if we decided to expand the application to sort products based on a criteria, or to search for a specific feature, then these structures would fall apart. As the circular linked list searches items sequentially, and sorting a stack would change the order of the stack. If these features were required, a binary search tree would have been an appropriate structure.

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 queue and stack ADT's. Instead, they relied on a public interface to accomplish the goals of the program.

Inefficiences

For the linear linked list of arrays I reused a specialized version of my linear linked list class from my previous assignment. I stripped the class down and refactored it for use in the product_stack ADT, and it performed reasonably well. However, in the clear_stack() method I had to loop through the list twice to properly clear the stack.

I had originally written the linear linked list class, to be agnostic of the data it contains. Thus, the LLL only ensures that nodes are properly deleted. But if the node contains a dynamically allocated, as was the case in our stack implementation, there would be a memory leak because the node would not deallocate the data properly. Two options presented itself, rewrite the LLL class to have knowledge that it was storing a dynamic array, or loop through each node deallocating the array manually and then tell the LLL to clear the list. I opted for the second method even though it incurs a second iteration. I felt this cost was small to keep the LLL
data agnostic, and simple in its implementation.

Feature Additions

Another solution to the described inefficiencies would have of course have been to write a specialized Flex Array class for this program; And this was my original plan. In the beginning phases of this project I had opted to reuse the linear linked class from the previous assignment until the program was functional. Once the program was running, I would specialize a Flexible Array class and plug it into the stack ADT and perform other optimizations. However, time and other priorities prevented me from doing this.