1

How well did the data structures perform for the assigned application?

The data structures used in this program comprise of a linked list of arrays and circular linked list. The circular linked list is used to keep a track of the history used and the stack keeps a list of operations. In hindsight the circular linked list may have been computationally expensive when considered that it requires both traversal for displaying and minor operations for dequeue and enqueue. Further memory use would have better if it was a circular linked list of arrays. This is because the memory allocated for each node on the heap is littered through out the run time stack frames for when they're called. The data structure used for each ADT is mostly seperated by walls, however there is cases where one variable is altered by two different classes, this may cause some side effects during run time. A few functions like dequeue and display for the stack ADT may not have been required, so not quite sure what the reason for the implementation. The problem with the implementation, is in the undo functions where if the match is not found the entire stack is popped and nothing is left. The precedence for the computation is done by popping the stack into an array and using a look ahead method to do the computation. This may cause stack overflow with a big stack.

2

Would a different data structure work better? If so, which one and why.

A better data structure for this programming assignment would have to be a binary parse tree from converting from infix to prefix notation for determining precidence and keeping a history of the calculations. The display function would still be iterative and on the order of $O(n \times log(n))$. Then converting each of the operations and numbers from characters to actual computation would be done in the same class. Thus, only one ADT would be required. The recovery of the mathematical expression can be done using the binary search since it is a binary parse tree. This implementation would not be more than a few functions and one ADT with a client interface. Parse trees would also be more efficient in doing complicated computation for the client interface. Also, algebraic operations become easier with sorting to solve symbolically.

3

What was efficient about this program design?

The efficiency about this program design is the abstraction between classes. The user only sees the calculator interface and not the ADT. The calculator ADT can only call the stack and queue ADT. The abstraction reduces the amount of bugs that can break the program during run time, thus making it an efficient design implementation. The efficiency also played a part of using a circular linked list to avoid traversal, further implementing the queue and stack ADT as a linked structure avoided traversals with the FIFO and LIFO methodology.

4

What was not efficient about this program design?

The efficiency in this design lacked in the area of using two separate ADT's to accomplish one thing. Further storing the history as a linked list may not be a great idea for large computation. This would be very memory intensive especially with the pointer arithmetic for displaying large queues, and popping for large stacks would still be a linear operation. The memory may over flow when performing computation, as described above.

5

What would you do differently?

If I could restructure the program I would, there would be more functions, each small and accomplishing different things. It would be more cohesive for a lack of vocabulary. The program also doesn't read in negative numbers so it's a calculator only over the natural numbers. Also see 1 for more design flaws encountered in this program design. The memory overflow may have been solved using another instance of the stack object. There may also be unchecked redundancies in the code.