Lab 3 – Data Structures

The code submitted for lab3 takes input polynomial equations in standard form (i.e. 1x1y1z1+...) and evaluates them against against stored values of x,y,z.

More specifically, the program reads the input file line by line, character by character to represent the input polynomial equation on each line in a linked list with each character being stored in a separate node of the linked list sequentially. For this implementation, the most simple form of a linked list was used, a singly-linked list. In my experience, more complex is not always more better. Frequently, the more complex the solution, the more difficult it can be to read/maintain/change. However when first writing a pen and paper solution, it was found that any advantages by a doubly-linked list could simply be achieved by either starting processing before the node of interest after examining the next node in the list or temporarily storing off a previous node.

The linked list representation of the polynomial is then passed to a calculator where starting with the list head, the calculator recursively steps through the polynomial to calculate the value of the equation based on the stored x,y,z values as required by the project description. The sum of all the given polynomials is then added together for the final evaluation. Recursion in this case offered a more elegant solution allowing the implicit compiler stack to be used in order to multiply all the parts of each polynomial before using a simple iterative process to step through the returned array to sum the total of the equation. Double digit numbers offered a particularly unique challenge for this approach as it was obvious how to group together two digits. It was eventually found that the program could always check the node for also being a number and simply add the value of the previous node multiplied by 10 to the next node if another digit was found.

The lab description indicated that a circular list was recommended, however only a single list implementation was written. There was no benefit found to being able to continuously step through the list, going from the tail back to the head. Actually, that type of list implementation would likely cause this program to get stuck and lead to a stack overflow exception eventually. After a circular list was determined to be unnecessary, a double linked list was then evaluated. As mentioned before, the benefits of a double linked list could easily be achieved in other ways. Thus, only a singly-linked list was utilized in order to reduce unnecessary overhead processing.

One of the main advantages of using a linked list is that you are not limited by a predetermined size like you would be when using an array. This feature was exploited extensively for taking inputs since the size could be variable. This is actually for efficient in comparison to the last lab where the size of the array being used was simply doubled every time it was close to overflowing. If this project were coded in C, this would also help to avoid potential inadvertent access of undefined memory resulting in any number of issues. However it is also sometimes beneficial to execute processing in a more constrained manner in order to also avoid inadvertent overwrite of memory.

A better approach next time may be to create a linked list and then use a separate polynomial object that leverages the linked list class in order to support the calculator or polynomial evaluator. Currently as implemented, the code is function albeit redundant at times because of the still relative rawness of the data. More data massaging would have allowed for a better interface to manipulate the polynomials likely making the use of a linked list nearly invisible.