Joe Canero
CSC260
Dr. Pulimood
Final Exam

Question 1:

A. While dynamic data structures are extraordinarily popular and versatile, they may not be the right variety of structure given a specific implementation; for example, Assignment 4 required a fixed-size array of Polynomials to be created. A fixed-size array saves time for both the programmer and computer. There is less work-related overhead when dealing with fixed-size data structures because additional code that manages a dynamic structure does not need to be generated. Also, memory leaks are less likely to happen as the program environment itself manages the allocated memory.

On the contrary, fixed-size arrays suffer the burden of their namesake: they can only store the amount they were instantiated with. If the need arises for the array to store more elements than it currently can, a large hassle ensues as processing time and memory resources need to be allocated to create a temporary array of a larger size and convert this into the main array; the fault with this method is lies in the fact that it is often difficult to accurately predict the size needed for an array; therefore, there will most likely be wasted space.

As stated earlier, when using fixed-size arrays, the program environment manages the memory, but dynamic arrays force the programmer to take an active role in the memory management process. Every dynamically allocated data item must be deleted properly before any references to those data items go out of scope or are changed to point to another data item. Even though this is intensive for the programmer, the memory resources on the computer are used more efficiently—when implemented correctly.

Considering Assignment 4, the program's robustness could have been improved by implementing a dynamic array in place of a fixed size array: exclusion of extra terms would be nonexistent. The array could resize easily to incorporate those extra terms. Simplification of terms was a significant hurdle in this assignment, but using dynamic arrays would not lessen that burden appreciably. When combining terms appropriately, empty elements are left behind, and these still need to be handled when dealing with a dynamic array.

As previously discussed, the incorporation of dynamic arrays would lead to an increase in the efficiency of memory usage: there would be no wasted space in any Polynomial array. Dynamic resizing of those arrays would occur based on the their contents after a simplification, addition, multiplication, etc. Multiplication of Polynomial arrays often led to the slicing off of extra terms, as multiplication of two Polynomials employs a general FOIL (first, inner, outer, last) method. And such a method can lead to the generation of large Polynomials in cases where there is very little or no simplification.

B. An important fundamental construct of dynamic data structures is the Linked List, which sees usability in a wide range of cases. Linked Lists could be implemented in any situation where a navigable chain of elements is needed. A benefit to the Linked List is its ability to store any number of elements, be it 1 or 1,000 or 100,000. In comparison with an array, this makes Linked Lists more robust, albeit time consuming to implement.

For a Linked List implementation to be efficient and secure, the code under the hood driving the Linked List must be error-free and carefully thought out. This is a real-time tradeoff, as the programmer needs to craft this code to a high degree of correctness. Another time trade off when analyzing Linked Lists is that random access to the data structure is not possible: nodes are accessed by traversing from the previous node—or from the last in certain implementations. Arrays permit random access; all that is needed is an array index.

In some situations, it is obvious that Linked Lists are the superior data type, and most of their memory trade offs are not exactly detrimental. A Linked List, as previously mentioned, is a dynamic data structure; in accordance with this status, it is capable of storing as much data as system memory allows. Linked Lists are beneficial in situations where the size of the data set is variable, for dynamic list addition and memory allocation occur automatically, whereas an array must be resized to add another element to a full array. Linked Lists also boast superiority in cases where elements must be removed. To remove an element from an array, that array index must be closed off; that is, the contents of the array must be shifted appropriately to accommodate the removal of that element. This array shifting is often not very efficient, and when dealing with massive data sets, is absolutely unacceptable. Linked Lists restructure themselves automatically as part of their remove protocol; therefore, it is reasonable to conclude that Linked Lists are generally just as memory efficient as arrays, if not more so.

A Linked List implementation for the Polynomial class would have been a good way to deal with the problem of adding, removing, multiplying, and simplifying terms. When using an array as the representation of a Polynomial, simplification leads to blank spaces in the array that must be closed off; contrarily, a Polynomial Linked List being simplified automatically corrects the loss of data and restructures effortlessly. Along the lines of simplification is the concept of sorting (simplifying the order?), and this operation would become less strenuous when using Linked Lists as the elements can be sorted as they are added to the Linked List, instead of sorting the Polynomial array only after all elements are added. Unfortunately, most other mathematical operations with our Polynomial class would not be improved significantly: addition and multiplication still require traversal of the Polynomial list.