Candace Holcombe-Volke
Lab 3 Reflection

Lab 3, LLDeterminants, required development of a program that uses linked list structures to read in input data from a file, calculate the determinant through recursive or iterative methods, and write the results to an output file. In this reflection, I will discuss my choice of linked list structures used to hold the data, alternative options, the cost of my method to calculate the determinant compared to the cost when using array-based structures, alternative methods, future program enhancements, and lessons learned.

main() initiates the program from within the ReadFile class. It accepts command line arguments for the output file location, args[0], and any number of input file locations, args[>0]. main() loops through all input files, calling readFile() while passing input file location arguments. readFile() interprets the input data by reading in one line at a time into an int array, tempArr. If the next input line does not contain a " " character, readFile() will interpret the input as the order of the next matrix. Otherwise, readFile() will initiate a loop to pull in the next matrix, continuing while the next input does not contain a space character, and add the data into a grounded header multi- linked list. For each element parsed into the int array, the loop will call addEnd(), passing the value of the data from the array. (addEnd() will be discussed further along with the details of the lists and nodes used.) All values in the input line are added to tempMatrixHorizontal list. Once all elements of the input array are added to tempMatrixHorizontal, the loop will add tempMatrixHorizontal to another list of LinkedClass lists. Once the loop completes, some error checking is performed, and valid matrices are then passed to getDeterminant().

getDeterminant() executes recursively to calculate the determinant of the matrix it is given. Using the order, current "*a"* value (of the Laplace formula), and the matrix, getDeterminant() will call itself with the same arguments, where the order value is decremented by one. In each recursive call to getDeterminant(), the matrix passed is the return value of getMatrix(), which is the minor of the calling matrix. getMatrix() creates the minors by evaluating the up nodes of the addNode pointer to determine if it is in the column of the *a* value. Nodes that should be added into the minor matrix will be added via the addEnd() and addLinkedNodeList() methods.

Finally, after the determinant has been calculated, it is passed to the writeFile() method with the output file location and string results to be appended to the output file. This continues through all inputs, for each input file.

The main data structure used in this program is the grounded header multi-linked list. Both the LinkedClass "horizontal" list of LinkedNodes and the LinkedNodeList "vertical" list of LinkedClass lists employ this structure. LinkedNode nodes hold next, prev, up, and down pointers, while the LinkedClass adds head and tail and holds up and down pointers as well. The LinkedNodeList adds head and tail to the LinkedClass list. I chose this structure for a number of reasons, mainly focusing on the most efficient way to store the data, without wasting extra space, but while adding capability to run the program as efficiently as possible.

Calculating the determinant of a matrix using the Laplace technique requires extracting data from nodes "previous" and "next" to the current node. This leads to a decision by the program designer to choose between using lists that are: a.) singularly linked with header node that will "reset" to the first node and use counters and loops to traverse lists to arrive at the desired previous nodes; b.) doubly linked with additional "prev" pointers; and c.) circularly linked with moving header node.

Candace Holcombe-Volke
Lab 3 Reflection

A singularly linked list has the benefit that additional space is not required to maintain a pointer to the previous node, however in order to arrive at a previous node, a "currentNode" pointer would need to be created, moved to the header node, and then iterated to the desired location. If moving backwards across the lists happened very rarely and not very "far" into the list, this might have been an efficient choice, particularly if the matrix is very big, requiring a lot of space for previous pointers. However, as moving backwards happens very often in the recursive calls, this was not an appropriate solution. Moving a current pointer back one node in a doubly linked list requires only one action, where moving to the previous node in the singly linked list described would require iterating m-1 times (where the current node is the m^{th} in the list) plus moving the current pointer to the head, and managing a loop with counter.

Alternatively, a circular singly linked list with header could be used. This would avoid needing to allocate space for previous pointers and manage for loops to arrive at the previous node, however in some cases could require a lot of extra traversal. In a list of n nodes, arriving at the previous node would require n-1 traversals, compared with the single movement in the doubly linked list. Further, it is common in the calculations to need the node that is previous to current, and above previous. If both horizontal and vertical lists utilized singly linked structures, this would require (n-1)+(n-1) node traversals rather than just 2 for the doubly linked structures. In both situations, the multi-linked aspect allows traversal to the previous node with only one movement – potentially much more efficient.

For the same reasons stated above, I opted to use a grounded header doubly linked list for the tempMatrixVertical list. Each LinkedNode is then attached to the top and bottom through the addNodeLinkedList() method, building the multi-linked element. This allows the up and down movement that is frequently utilized in the getDeterminant() calculations.

While using the multi- doubly-linked list is more efficient that the linked alternatives, it is still less efficient than using an array-based structure. In this type of problem, exploiting random access is very beneficial and allows the program to avoid traversing nodes that it does not need to access. This is especially noticeable when utilizing multi-linked lists. Arrays, due to the fact that they hold data in specific order, are automatically built with multi-linked capability. Simply by utilizing a 2-dimensional array, the "left", "right", "top", and "bottom" nodes can be derived and directly reached by simply calling for their data at specific indexes. For example the node above and to the left of a[m][n] is a[m-1][n-1] – no additional linking required after the nodes' values are assigned. In linked structures, an additional traversal must occur after initialization for each "horizontal" list added, to specifically connect "top" and "bottom" nodes. Additionally, array based lists do not require the space allocation for previous and next pointers, as the array structure holds that information inherently. In smaller matrices, the space is not a driving issue, but in large matrices, it would become more important.

This program, like my previous determinant calculator, uses a recursive method to evaluate the matrix. I did a lot of consideration about how to implement the calculation iteratively, and it became a very complicated problem. I initially intended to create nested for loops that would cross the top row of the original matrix, returning sub matrices for each *a* value along the way. I realized early on that this was much more difficult than I expected because the number of nesting would depend on the order of the matrix. For order two matrices, two loops would suffice. However, when the order increased to three, it would require an additional loop to account for the *a* values of the new minor matrix. I then explored was in which I could store the smallest minors, which would allow just one additional multiplication when the matrix grows rather than recalculating the minor all over again. But this also

became very difficult to manage. Ultimately, I was not able to find a suitable iterative method using the Laplace expansion method to complete this task that would be more efficient than the O(n!) cost of the recursion.

In trying to understand the iterative alternatives, I was able to find alternative methods to calculate determinants that are simpler than the Laplace method. While I'm not very adept at manipulating matrices, I found that there are alternative ways of calculating matrix determinants which can significantly reduce the complexity of the problem. One enhancement I was able to introduce in this calculator is the recognition that an *a* value of 0 will result in a determinant of 0 for its minor. This can significantly reduce the number of operations required to find the matrix, particularly if the *a* value is a multiplier of the starting matrix, or one of the largest sub matrices.

In the future, I would enhance this program by having some initial tests to determine if the matrix has certain features, such as being sparse, upper, lower, etc., that I could exploit to more efficiently calculate the determinant. One negative to these type of tests is that, in a linked list structure, they would require additional traversal, which could be costly if a majority of the matrices would require the full recursive calculations in addition. I would implement these tests with "exit" points, so that if, for example, a test was determining whether the matrix had a certain feature, as soon as it was shown to be false, it would discontinue the traversal, and return to the starting position.

Additionally, I would like to implement a "zero counter" and store that information in the header node. This would count the number of 0s in each row, and once the full matrix is read in, make a quick comparison for which row contains the most 0 elements. This would then be the row selected to traverse, allowing for the fewest number of calculations.

Finally, most of the efficiency measures that would be implemented would require additional functions that could be applied to the lists, such as insert (which would add nodes to the beginning of a list and would be useful in the alternative implementation that stores minors rather than recalculating). This implementation did not require the use of many additional optional methods.

In conclusion, I learned a lot about implementing linked list structures while undertaking this assignment. I initially struggled with finding a way to implement the mulit-linked aspect of the lists, but established an additional traversal and creation of a "list of lists" as the best solution. I was not able to implement the calculator more efficiently, and in fact added to the cost because of the additional traversals that were not needed when using the array-based structure. For the purposes of this particular challenge, where I did not require the ability to shift nodes frequently and also know the size of the input matrix (as given in the input), the array-based linked list allowed for exploiting random access and inherent multi-linking capability, and would be the preferred method when implementing the Laplace formula for matrix determinant calculation.