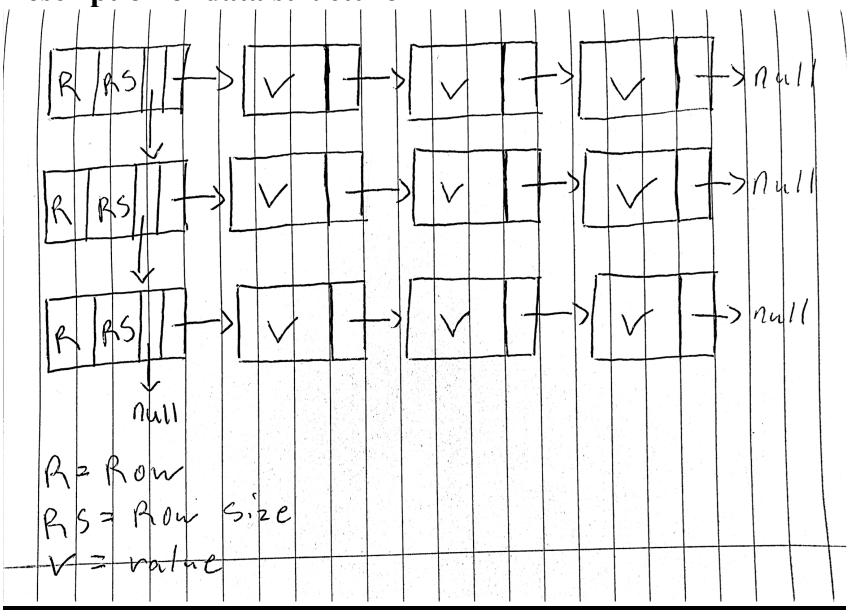
Kevin (Feng Yong) Chen
JHU Email: fchen29@jhu.edu
Student ID: FE4A8B

Lab 3: Analysis

Data Structure and Implementation

Description of data structure



All of the matrices in this lab were stored using the linked list data structure depicted above. The data structure consist of 2 kinds of nodes:

1) A header node marks the beginning of a row and it has 2 data fields, row and row size, as depicted above. Head nodes are multi-linked. Each header node has a reference field that points to the first value node in the row and another reference field that points to the header node in the next row. Header nodes act as indices for a matrix so that the program knows which row it is currently located. They do not contain actual values of a matrix and are not considered to be a part of the actual matrix.
2) Value nodes contains the actual values that make up a matrix. Value nodes are singly linked. Each value node has a data field that stores data from the matrix and a reference field that points to the next value node in the row.

The linked list structure I used in this lab is essentially a number of singly linked lists with header nodes that are linked to each other through their header nodes. I believe this is a simple yet effective implementation for this lab. Justification for this implementation will be discuss later in this analysis.

Justification of data structure

Although a 2D array may be a very convenient structure to use for storing matrices as shown in the previous lab, it may not be the most practical in the real world due to its static allocative property. The input format of lab 2 allows the program to know the order of a matrix before constructing it. This allows the program to allocate exactly the amount of space that is needed for a matrix. However, in reality the order of a matrix may not always be known prior to construction. Hence, inaccurate allocation of space may lead to overflow or waste of space if the space allocated is too small or too big, respectively. Thus, using a linked list structure may be beneficial when real world constraints are taken into consideration.

Using a linked list, the order of a matrix does not have to be known before the matrix is constructed. An application could simply construct a matrix as values are being inputted without having to know how big the matrix needs to be and the matrix could be of any size. There is no risk of overflow. Further, its dynamic allocative property ensures that the amount of space allocated is exactly as needed so no space will be wasted. Lastly, using linked list gives the application the option to construct both valid matrices (i.e. a square matrix with only numeric values) and invalid matrices first and postpone the handling of invalid matrices after all the matrices are constructed. The program could decide which one to process after everything is stored. This would be harder to do using an array since the space of an array is fixed and if there is an invalid matrix that has an extra row or column, it would cause errors in the program.

Justification of implementation

I believe my implementation of the linked list structure is suitable to this lab as it is simplistic yet it still meets all the demands of this lab. Besides from accessibility and the capability to store values, there are no other things that are mandatory for the matrices in this lab. Thus I believe any linked structure that possesses the following 3 properties would be useful for this assignment:

1) Ease of access: The data in a matrix need to be easy to access. Given a pair of coordinates, the application needs to be able to efficiently retrieve the value at that location in the matrix. This is important when constructing minors of a matrix as the program needs to retrieve values from the original matrix with ease and paste that value onto the new sub-matrix.
2) Easy to build: the matrices need to be easy to construct. This means that, to maximize efficiency, there should be as little unnecessary operations as possible when inserting a new value into the matrix. This is important when the program takes in data from the input text file to construct matrices.
3) Storage efficiency: The matrix needs to be able to store data efficiently without allocating space that are not utilized by the program.

Since the linked lists do not need to perform that many functions in this lab, I believe it is best to keep the data structure as simple as possible to avoid wasting computational resources. Thus I believe a structure that maximizes the efficiency of the 3 functions noted above while minimizing other unnecessary features would be the most suitable for this lab.

Since linked list structures all have sequential access to its elements, the runtime efficiency to access elements in a matrix should be similar for all types of linked list structures. Assuming coordinates are given and every element is equally likely to be accessed, then the average runtime to access an element in a linked list is equal to the average runtime to access the correct row, n/2, plus the average runtime to access the correct column, n/2. Thus the average runtime to access an element in a matrix should be n/2+n/2= n for all linked list structures.

Since a singly linked list is the simplest form of linked list structures, it is the easiest to build. This is because an addition of a new node only requires the program to create one link. So in general the insertion of a singly linked node involves the least amount of constant time operation.

As mentioned earlier, the value nodes in my linked list structure only has 1 data field and 1 reference field. Thus it does not take up as much space compared to other nodes that may have multiple data fields or reference fields. In other words, the value nodes are spatially efficient.

Efficiency

Since the algorithm for determining the determinant of a matrix is the same as the last lab, the best case runtime complexity for computing the determinant is also $\Omega(n!)$. However, since linked list structures have sequential access rather than random access, this adds additional processing time when the program is executed.

Specifically, the extra processing time will affect the minor method in the Determinant class. Given the coordinates, the runtime complexity to access a particular value in the matrix is O(n) – it takes O(n) time to access the correct row and O(n) time to access the correct column so the total runtime equals O(n) + O(n)=O(2n)=O(n). In contrast, accessing a value in a 2D array would only have a runtime complexity of O(1) since the data structure allows for random access. The amount of work it takes to build the minor of a n*n matrix is (n-1)(n-1). In other words, when computing the minor of a n*n matrix, the n*n matrix will be accessed (n-1)(n-1) times by the minor method. Since each access to the n*n matrix takes O(n) time, the runtime complexity of the minor method is O(n)*O((n-1)(n-1))=O(n^3). In contrast, constructing a minor using a 2D array will only have a runtime complexity of O(n^2). This extra processing time from using a linked list will in turn lead to longer processing time for the determinant algorithm compared to using a 2D array because at each recursive level, the algorithm calls the minor method (n-i) times with i being the level of recursion starting from 0.

Spatial complexity:
Since the same algorithm is used, the spatial complexity for the determinant algorithm is the same as lab 2 which is

$$\sum_{i=0}^{n}(n-i)(n-i)$$

However, if we are simply considering the upper bound of the algorithm we could simplify (n-i) to n and the worst case runtime then becomes O(n^3) – we know that in the worst case, the algorithm given in this lab will not exceed n^3.

Although the spatial complexity is theoretically the same as what we saw in lab 2, in reality the recursive determinant algorithm may take up more space when it is implemented using a linked list structure. This is because the nodes that make up the linked list tend to require more space than the slots in an array. A slot in an array only needs to store a numeric value but a node needs to store references to other nodes in addition to storing that numeric value.

Appropriateness of Approach

A recap on the iterative approach

As described in lab 2, an iterative approach could also be used to implement the determinant algorithm. This could be done by using a while loop and a linked implementation of the stack data structure. The nodes used in the stack needs to be structured so that it has a data field that stores a matrix and a multiplier field to represent the ***"Math.pow(-1, ((i + 1) + (j + 1))) * matrix[i][j]"*** part of the algorithm. At each iteration, the while loop will create a minor of the matrix that is at the top of the stack and push that minor onto the stack so that it becomes the new top. The while loop will keep doing this until it detects a minor with an order of n=1. In such case the matrix at the top will return a calculated value back to the matrix underneath it. The runtime and spatial complexity of an iterative approach should be similar to that of the recursive approach.

Justification of the recursive approach

Given choices between a recursive approach and an iterative approach. I believe a recursive approach is a better way to implement the determinant algorithm. My reasons are as follows:

1) As shown in my lab 2 analysis, an iterative approach is likely to require longer and more complicated codes. It may also require the coder to create more classes to support the determinant algorithm. For example, to actually implement the suggested iterative approach, one would have to create a node class declaring class variables such as a matrix variable to store the matrix, a data field to store the multiplier, an integer variable to keep count of the

number of times the minor method has been called on the matrix if n>1 and also create a stack class that uses such nodes. This requires more coding and is more time consuming.
2) On the other hand, the codes for a recursive approach are much shorter and more simplistic. When a coder can write simpler, more elegant codes, I believe that reduces the risk for making careless mistakes. It also reduces the amount of time needed to develop the program, making coding more efficient.
3) The algorithm is naturally more recursive than iterative as it involves expressing a higher order, more complex matrix in terms of its simpler form. It also provides a base case (i.e. n=1) which allows the system to eventually solve the problem using that base case.

Description and Justification of Design

There are 5 classes that were developed in this program:

1) Main class: contains the main method that is responsible for processing data from an input text file and print the processed results onto an output text file. The main method does the majority of the error handling of the program.
2) Determinant class: the determinant class contains the algorithm that is used to calculate the determinant of a matrix and it also contains the algorithm that computes the minor of an element in a matrix.
3) Matrix class: the matrix class is a linked list structure class that is used to store a matrix.
4) Header Node class: this class supports the matrix class. It is one of the 2 types of nodes used to construct a linked list matrix.
5) Node class: this class supports the matrix class. It is one of the 2 types of nodes used to construct a linked list matrix.

I chose to do most of my error handling in the main class because of the following reasons:

- Checking and filtering data early on ensures that all the data that are passed down to the matrix class and determinant class are valid data. If I could be sure that all of the data that are fed to the matrix and determinant classes are in the correct format, then that avoids the necessity to implement error handling measures within the 2 classes. This allows me to simplify coding when designing the matrix and it allows me to keep the design of the matrix and the determinant class as simple as possible.
- As mentioned earlier, the structure of the matrix was design to be as simple as possible. The determinant class also has a very simple structure. Prior to writing the main method, both classes were develop separately and tested extensively to ensure that each works properly before I moved on to develop the main class which handles the I/O. Thus, I could be fairly sure that if there is an error, it probably stems from the input data. The main method will be the first method to process these data so it made sense to me to report an error early on so that it is easier to pin point what the problem is and where it originated.
- The main method is also better able to handle errors because it is a higher level method and it is easier for a higher level method to determine the appropriate response to an error compared to a lower level method such as the append function in the matrix class.

The matrix class was designed to be as simple and efficient as possible. The purpose of each method in the class were noted in the introductory comment block of the matrix class in the source code. Thus, they will not be repeated again in this section.

There are 2 methods written in the determinant class – a minor method, which calculates the minor of an element, and a determinant method, det, which calculates the determinant of a matrix. Writing minor as a separate method from the det method allows for 2 things:

1) It breaks the codes down so that they are easier for debugging.
    - The minor method was developed and tested first to ensure that it functions properly

- The det method was subsequently developed and so if there was an error, I could be sure that the problem stems from the det method itself and not from the minor method.

2) It declutters the codes in the det method and makes it easier for others to read. Decluttering the codes also reduces the risk for me to make coding mistake since the codes are simplified

Lessons From This Lab

- I gained a better understanding of how sequential access affects the runtime complexity of a program and how a program that uses a structure with random access could run faster than one that uses a linked list structure.
- I also learned about the benefit of using a linked list structure. For example, I now have a more solid understanding of how a linked list structure could give a program more flexibility and how that could be useful in the real world
- This lab pushed me to further analyze the advantages and disadvantages of an array versus a linked structure. The experience I gained from lab 2 and 3 gave me better insights on the pros and cons of each structure. The results from lab 2 and 3 taught me that there are tradeoffs to using each data structure and which one you use will depend on the context in which the structure is used and what it is used for.

Enhancement

- The program ignores extra spaces between values. For example, if there is a row in the input file that reads "1 2 3", the program will interpret it as "1 2 3" and will not report it as error. I think this allows the input format to be more flexibility and thus allows the user to make more minor mistakes while still be able to get the correct results.
- Although the input data in this lab are presumed to be all integers, I enhanced the program so that it can also compute matrices with values that are doubles. Test cases have also been created to test this function.
- The program recognizes when the number of elements in a row does not match with the order of the matrix, and tells the user that the number of columns does not match with n.
- The program recognizes when the number of rows does not match with the order of the matrix, and tells the user that the number of rows does not match with n.
- I added in various labels to make the output format more user friendly.
- I created a "Replace" method in the matrix class even though this method was not required for the assignment. This "Replace" method was useful during the development stage when I was testing the determinant algorithm with different sets of data. I could simply alter a few of the elements in the matrix to test for a different scenario rather than having to create a whole new matrix.