Data Structure Assignment#3

German language and literature 2016130927
Park Jun Yeong

1. Development environment: Microsoft Visual Studio Professional 2013

2. Explanation and algorithm for the code

: There are 64 blank matrix indexes (0-63, total 64) initially as global variables so that you could handle these by putting its number (for example you can treat matrix of index 0 by putting 1, since the number decremented by 1 is the real index you would treat). Until you enter number 7 for exit, you could deal with any matrix operation between 1 and 6: read, write, erase, add, multiply, and transpose. All of these sparse matrix operations return an expected result if conditions for operation are sufficient: input data should be at least matrix with a certain rows and columns except mread operation. That is, no NULL (or blank) matrixPointer pointed by your index would be allowed. Then, the following is a more detailed description for these 6 operations:

(1). mread:

1). Check sufficient condition to construct a matrix.
When this function is called, you should enter the number of rows, columns and nonzero terms for your matrix: rows and columns should be at least more than or equal to 1 and at most MAX_SIZE number (in this case MAX_SIZE is set as 50) and no negative number for nonzero terms is allowed. So the nonzero terms should be at least more than or equal to 0.

2). Enter your entries and construct a matrix.
Then you would also enter which value is entered in which row and column by putting them at exactly 'numTerms' times in the while loop. Being checked whether your values are sufficient or not for this matrix, you cannot also enter values in the repeated place, i.e. in the same row and column. So an array of entryNodes, 'mark' is as many as 'numTerms' dynamically allocated to check which coordinate is dealt with: if there is no duplicated place, what you entered is inserted and so on. After these reading process is done, rows and columns are connected by putting them together. An array mark is also got freed.

3). Sort in row major order and return.
At this moment, however, it is not enough to return this matrix in this order itself: Since these entries have been stored in order you entered, not in order required by other operations – that is, in row major order (if in same row, in column major order). So these entries should be sorted by some sort algorithm, which is implemented as merge sort here. An array of entryNode 'data' is dynamically allocated as many as the numTerms of matrix and these coordinate values are pushed into 'data' from that matrix. Then, as is well-known, these are recursively sorted in row major order by merge sort and these sorted data also are re-pushed into the original matrix in the same way all of data are printed out in mwrite function.

4). Analysis of mread
This mread operation is almost same with that of textbook p.182-183, except a certain conditions for constructing a matrix is to be checked by loop statement and sorting process: Unless you enter the wrong (whether that number of row or col is not allowed in that matrix or that coordinate is duplicate), you cannot break it out. Right before your matrix is returned, it is also sorted in row major order so as to enable other matrix operations to function more easily. Thus, the total time is: O(numRows + numCols +

numTerms), exactly same as in the textbook.

(2). mwrite:

1). Check sufficient condition to print out a matrix.
At least your input matrix should be matrix – not just NULL matrixPointer so that this function could recognize it. The only part that has actually been modified is this NULL check. If not, just return NULL.

2). Print out in row major form and analysis.
If then (it is a matrix), with nested for loop – outer for each row(the numRows) and inner for these entries in that row (the numTerms), all of entries are printed out in row major form. So the total time of the this function mwrite, as in the textbook p.184, O(numRows + numTerms).

(3). merase:

1). Check sufficient condition to erase a matrix.

As in the case of mwrite function, a matrix should be entered first to be removed out: so check whether it is NULL or not, if then (NULL) just return. This checker is also the only parted that has been added as in the case of mwrite.

2). Free all of nodes and analysis.

If then (it is a matrix), all of entries and header nodes by row are freed and other remaining header nodes too with loop statements. The total time is: O(numRows + numCols + numTerms) as in the textbook p. 185.

(4). madd:

1). Check sufficient condition to get a sum of two matrices.

To get a sum of two matrices, you should make sure that they are in same size (rows and cols) as a matrix. If not, just return NULL.

2). Highest priority entries of each matrices being compared, enter all of entries.

If then (input matrices are in same size), a new node and the header nodes are also initialized as in mread. Then, until all of entries in both of matrices are handle, their highest priority entries are compared each: you should give higher priority to smaller row (if row is same, then to smaller column). Insert it, hold another entry and get back. Of course you should get the sum of that values when that entries are in the same position. While entries being taken, there could be no remaining entries in either of them. Then, just think entry of that matrix as minimum priority (that is MAX_SIZE) and compare with it. If there's no entries in both of them, this process finishes. All of header node for this sum matrix are connected and sum matrix is returned.

3). Analysis of madd

As in mread function, the while loop requires only O(numTerms) times (this numTerms is the larger number of terms between two matrices). Considering other set up time for initialization and connection, the total time is: O(max {numRows, numCols} + numTerms) = O(numRows + numCols + numTerms).

(5). mmulti:

1). Check sufficient condition to get a product of two matrices.

To multiply two matrices, you should make sure that the size of column in first matrix and the size of row in second matrix are the same. If not, just return NULL.

2). If there is a match, enter the total.

The value of a certain coordinate (i, j) in product matrix is the sum of products of two values in which the column in first matrix and the row in second matrix is the same: $d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$

In this matrix multiplication implement, after this coordinate (i, j) is fixed, then each entry in row i of first matrix is compared to each entry in column j of second matrix with triple nested loop – outer loop for a certain fixed entry in row i, inner loop for all of existing entries in col j. If there is no matching entry in each other, skip the entry with higher priority (=smaller row or smaller col size in this case) and go to the next entry in that skipped matrix, holding other one. Only if a matching between two entries occurs, the product of these values are summed to a certain variable 'value' (which is initialize as 0) and it would be inserted as an final value for entry in (i, j) after all of entries in row i of first matrix and col j of second matrix are searched. If all of entries in col j is searched, then j becomes j+1, j+2… until last column of second matrix. Then, row i becomes i+1, j is reset as 0 and this process goes on until last entry of last row of first matrix is handled: insertion process is done.

3). Analysis of mmulti

The number of iterations of the outermost for loop is numRows of first matrix. for any row i, the number of iterations of middle for loop is numCols of second matrix + the number of entries for col j in second matrix (as in write function). for any col j, the number of iterations of the innermost while loop is equal to the number of entries between first taken entry and last taken entry(which could be in any of these matrices). Since then, it is often unknown how many times these comparison occurs: in best case, there is no other entries in either of that line (for row i of first matrix or for col j of second matrix) and in worst case, there are as many entries as col (for first matrix) number. Then, let numRows of first matrix be n, numCols of second matrix m and numTerms l (it is assumed to be worst, so it is equal to numCols of first matrix and numRows of second matrix). So total time is: O(nml).

(6). mtranspose:

1). Check sufficient condition to transpose.

You should enter the matrix with row and col, not just NULL matrixPointer.

2). Transpose and analysis

Let the numRows of entered matrix be n, the numCols be m, the numTerms be l. transposed matrix of entered one is m by n matrix. numTerms is same. Then all of entries in entered one is inserted into transposed matrix one by one with its row and col swapped. After then, header nodes are connected and entries are sorted also in row major order as in mread. So total time is: O(max {numRows, numCols} + numTerms), which is O(numRows + numCols + numTerms).

*more detailed commentaries for C code in 2016130927_Sparse Matrix.c and exe file is also attached.