COSC2430: Programming and Data Structures
Sparse Matrix Addition
2×2: Non-recursive and recursive; arrays and linked lists

1 Introduction

You will create a C++ program to add two sparse matrices. Sparse matrices have many zero entries, but such entries are omitted in the input file and corresponding data structures to save space and reduce processing time. Your algorithm should process matrices efficiently.

Addition: Two matrices to be added must match in the number of rows and columns. That is they must have the same size, but they do not need to be square. Given two matrices A and B of sizes $m \times n$ its addition $C = A + B$ is defined as the addition of their respective entries:

$$c_{ij} = a_{ij} + b_{ij}.$$

2 Input

The input are two text files, with one matrix per file. There will be ONE matrix entry per line in the file and each line will have a triplet of numbers i, j, v where i, j indicate the entry and $v \neq 0$ is a real number. Subscripts i, j are positive integers and the value v is a real number (possibly negative). Comments have a # at the beginning of the line. Examples of input and result file are below.

```
#example 1
#Matrix A, no zero entries
#i j value
1 1 1
1 2 2
2 1 3
2 2 4

#Matrix B
1 1 4
1 2 3
2 1 2
2 2 1

#Matrix C=A+B
1 1 5.00
1 2 5.00
2 1 5.00
2 2 5.00
```

```
# example 2
#Matrix A, I matrix, notice one zero entry included
#i j value
1 1 1.00
1 2 0.0
2 2 1

#Matrix B
1 1 -1.0
2 2 -1.0

#Matrix C=A+B
2 2 0.00

#example 3
# A
1 1 1.00
100000 100000 0.00

#B
100000 100000 1.00

#C=A+B, huge matrix
1 1 1.00
100000 100000 1.00


#example 4: incompatible for addition
# A
10 10 1.0

#B
100 100 1.00

#example 5
#Matrix A, not sorted
#i j value
2 2 4
2 1 3
1 1 1
1 2 2

#Matrix B, not sorted
2 1 2
1 1 4
2 2 1
1 2 3

#Matrix C=A+B
```

```
1 1 5.00
1 2 5.00
2 1 5.00
2 2 5.00
```

3 Program and input specification

The main program should be called "sparse". The result matrix should be stored on the (sorted) output file (another text file), provided on the command line. Call syntax:

```
sparse "operation=<add|multiply>;storage=<array|linkedlist>;recursive=<Y/N>;A=<file>;B=<file>;result=<file>".
```

Notice the brackets $< >$ indicate the parameter value and they are not part of the input string. Assumptions: If parameter "storage" is not specified arrays are the default. If parameter "recursive" is not specified then you can assume non-recursive is the default. Matrices are given in sparse form, in list form (not necessarily sorted). However, output matrix is written sorted by row, column (this is required for testing). In general, the list contains only non-zero entries, but you can assume the lower right corner of the matrix will be given to determine size even if such entry is zero. That is, if an entry does not appear it is assumed to be zero. The input file has 3 columns, where numbers are separated by spaces. Subscripts are assumed positive integers, but they should be validated, the entry value will be a real number (it may be negative). The output matrix will be written in sparse form.

```
# you can use array or list: non-recursive
sparse "operation=add;A=a.txt;B=b.txt;result=C.txt"

# recursive array
sparse "operation=add;A=01.txt;B=02.txt;storage=array;recursive=Y;result=C.txt"


# recursive linked list
sparse "operation=add;A=X.txt;B=Y.txt;storage=linkedlist;recursive=Y;result=sumXY.txt"
```

4 Requirements

- You need to program lists with arrays and doubly linked lists in 2 phases.

  Phase 1: Arrays: recursive and non-recursive functions.

  Phase 2: linked lists: recursive and non-recursive functions. Notice that your program is expected to still process arrays fixing any errors from Phase 1.

  In other words, you need *four* versions of addition.

- Operations required (in this HW):

  matrix addition: YES

  matrix multiplication: NO (but you will do it a future homework).

- Storage: You will develop two sparse matrix versions: one with arrays and one with doubly linked lists.

  Arrays: lists stored as arrays of non-zero elements. You can store the matrix as a single 1-dimensional array or or as a list of lists (rows or columns) or as an array of lists (with one row/column per list). You cannot store zero entries, except the lower-right corner of the matrix.

Doubly linked list: the list stores only non-zero entries, except for the lower right corner. You can store the matrix as a single list (sorted by row, column), or as a list of lists. You can store the matrix as an array of linked lists, but it is discouraged. Memory for each node should be allocated as input matrices are read, entry by entry, or row by row. It is preferable the input files are read once. Memory should be freed at the end of the program.

- Processing: recursive Y/N

  You should develop both non-recursive and recursive versions of addition for either data structure.

  Non-recursive: while or for loops allowed.

  recursive: you **cannot use loops** to process matrices (while/for). That is, you must define recursive C++ functions to do the sum For each row/column and then recursively add each entry pair. Note: You do not need to program recursion for reading or printing matrices,

- C++ programming requirements:

  Sparse storage: you cannot store zeroes in your matrix data structure, other than the lower right corner a_{mn}. You cannot use simple 2-dimensional arrays to store the matrix (e.g. double A[10][10]). It is preferable your storage takes space $O(q)$, where q is the number of non-zero entries.

  Your C++ code must include at least two ".h" files: arraylist.h, linkedlist.h, whose source code will be verified by the TAs. Your main list functions must be in these files. You can have more C++ files for parameters, base classes, text file manipulation, etc.

  Subscripts in input and output matrices are based on a standard mathematical definition, indexed from 1. Matrices of size $m \times n$ start on entry [1][1] and go up to [max(i),max(j)] $\forall a_{ij}, 1 \leq i \leq m, 1 \leq j \leq n$. However, in your C++ code you can use arrays indexed from [0], from [1] or indexed by your own functions or pointers making a subscript conversion, but the input/output matrices always have subscripts starting at 1,1.

- Input:

  One file per matrix. In general (80% of test cases), you can assume matrices are sorted by row/column and therefore ready for fast addition. However, prepare your program to detect and handle sparse matrices (20% of test cases) who entries are not sorted. In such case you need to program a slower algorithm that searches for each entry to add in either matrix. Prepare your program for slight changes in format (e.g. an extra carriage return, empty lines, an extra space separating numbers, etc). Your program should not get stuck due to a small format issue.

  Notice there may be 1, 2 or more comment lines at the top of the file and there may be comments elsewhere in the file.

  Numbers: can be positive or negative, where negatives start with - and one digit (we will not provide numbers like -.2, or -0.31416e+10). In general zeroes are omitted. But it is acceptable an input matrix has a few zero entries. Input numbers can have up to 5 decimals. You should write output numbers only with 2 decimals (for testing purposes).

  Your program must reject matrices whose sizes are incompatible for addition and produce an empty file as result. Incompatible matrices is the only case where your program should stop. Your program must be prepared to skip lines with bad input (bad subscripts,errors, strange symbols, etc). That is, it should attempt to process the matrices anyway.

  Size: Notice matrices are not necessarily squared ane they can even have only one entry: 1×1. Notice the lower right corner will always be provided in the input matrix and it should also be written in the output matrix: This entry determines matrix size. Always determine matrix theoretical size based on the lower right corner matrix entry.

- Output:

  The output matrix should be written in sorted order by row and column in sparse format (one entry per line) as well. That is, all non-zero entries for row 1, all entries for row 2, and so on. Depending on how you program addition sorting may not be required. For sorting you can user any algorithm, but you will need one for arrays, and another one for linked lists.

- Testing:

  Your program will be tested for dynamic memory allocation (i.e. enforcing linked lists instead of arrays) and memory leaks (new, delete operators correctly programmed). In other words, the TAs will verify you do not exploit 2-dimensional arrays instead of doubly linked lists for Phase 2 and will make sure your program does not crash.

  Notice the program will be tested with sparse matrices whose theoretical size goes up to 1000000×1000000 (i.e. they will have large subscripts in some entries). In other words, it will be impossible to use C++ 2-dimensional arrays. Additional clarifications: matrix subscripts can be large, matrix files may have incorrect entries (skip them), matrix files may be empty or non-existent (report error).

- Grades:

  Phase 1 grade: you will receive a final grade for Phase 1 and a preliminary grade for Phase 2. It is highly recommended you make an effort to attempt Phase 2 on the 1st deadline so that you get feedback and a better grade on Phase 2.

  Phase 2 grade: you can fix errors from Phase 1 and make Phase 1 and Phase 2 requirements fully functional. Phase 2 is an opportunity for resubmission of a fixed program. Resubmissions or late submissions will not be allowed (30% penalty for each day).