CSE326 – Data Structures and Algorithms
Winter 2004
Wet assignment #1

Due dates:

Team formation:	by 5pm on Monday, Jan 26, 2004
Full assignment:	by class time on Wed, Feb. 4, 2004

General.
Be sure to read this *entire* document as soon as possible, since this may affect how you approach the problem!

Logistics.
For this wet (programming) assignment, you need to work in a team with one other person in the class. It is up to you to decide who that person will be.

We require that one of the two students in each team email both TAs (and CC their teammate) *by 5pm on Monday, Jan 26* letting us know the following:

(a) who the team members will be – names and email addresses for both; and
(b) what implementation language (Java or C++) both of you are going to be using.

Note: If you fail to notify us on time, we will deduct 10% of your score on this assignment.

Problem area description.
The problem area for this assignment will be *sparse matrices*.
Let $A_{m,n}$ be a matrix of $m \times n$ integer elements organized in m rows and n columns. If many of the matrix elements are 0, we call the matrix sparse.
To store a sparse matrix in memory, one can do better than storing all the elements (since many of them are already known to be 0). Thus, the required space reduces from $m \times n$ (for a general matrix) to $n + m + k_A$ (where k_A is the actual number of non-zero elements in the sparse matrix A). For such efficient storage one can use a data structure based on linked lists. Figure 1 illustrates this.

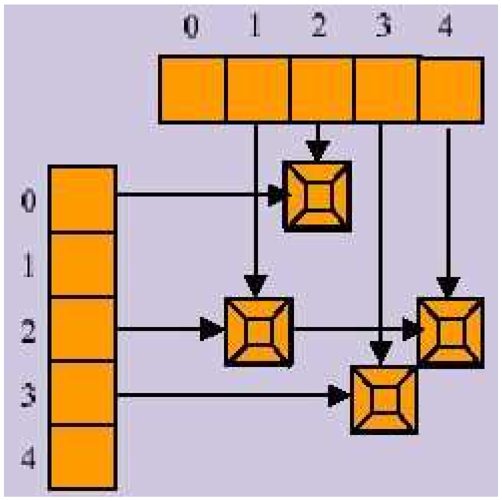

Figure 1. A 5 × 5 sparse matrix represented with linked lists.

The element in cell `(2,1)` is the first non-zero element on row `2`. It is pointed to by the header element for that row. The next non-zero element on row `2` is in cell `(2,4)`, to which there is a pointer from cell `(2,1)`. The element in cell `(2,4)` is also the first non-zero element in column `4`. It is pointed to by the header element for that column. Finally, row `1` and column `0` have no non-zero elements.

What you need to do.
For this assignment you need to implement a class for handling sparse matrices using the suggested linked list-based data structure.
A list element definition may look like this (in pseudo-code):

```
typedef struct node {
   row:    integer;
   column: integer;
   value:  integer;
   next:   node pointer;
   down:   node pointer;
} NODE;
```

1. *Implementation.* Specifically, we want to have the following operations:

1) `read(A,f)` – reads in a sparse matrix A from file f and creates, initializes, and fills in the above data structure with the elements of matrix A.
   This operation needs to have time complexity $O(n+m+k_A)$.
   The format of the input file f is described in the next section.
2) `mult(A,c)` – multiplies the elements of a sparse matrix A by a constant `c`.
   This operation needs to have time complexity $O(1)$.
3) `add(A,B)` – adds two sparse matrices A and B of the same size $m \times n$, and stores the result in matrix A.
   This operation needs to have time complexity $O(n+m+k_A+k_B)$.
4) `mult(A,B,C)` – multiplies two sparse matrices A and B of sizes $m \times n$ and $n \times p$, respectively, and stores the result in matrix C of size $m \times p$. As part of this operation your code will need to first create and initialize the new matrix C. (Hint: You may find it helpful to implement a separate helper operation for that.)
   The multiplication needs to have as good a time complexity as you can manage.
5) `write(A,f)` – writes a sparse matrix A out to file f, following the same format as in the input file (see the next section).
   This operation, similarly to `read(A,f)`, needs to have time complexity $O(n+m+k_A)$.

It is OK, and you may find it useful to implement other helper routines in your sparse matrix class, e.g., `create(A,m,n)`, `getNumRows(A)`, `getNumColumns(A)`, etc.

2. *Testing.* To test your sparse matrix implementation, you will need to write a simple test module (program) that uses all of the above operations to perform computations on matrices whose data come from test case files.

3. *Special cases.* Think about how you would deal with special cases, like a sparse matrix with some or all of its elements becoming 0 as a result of an operation. Does that affect, and if so – how, the way you store the matrix, or the time and space complexities of any further operations on such a matrix? Be sure to discuss these questions in your submission.

Test cases.
We will provide test case files having the following format:

- Rows starting with the pound sign (#) contain comments and should be disregarded by your program.
- The first row that does not start with a pound sign defines the size of the matrix (number of rows and columns) as well as the number of non-zero elements in it. For example,

  ```
  m = 3   n = 10   numItems = 18
  ```

  means that the matrix has 3 rows, 10 columns and 18 non-zero elements (of the 30 total elements).
  An exception to this is when there is a scaling (constant) factor like in the operation `mult(A,c)`. Then the test case file will indicate the value of that constant factor, so the corresponding row will look, for example, like this:

  ```
  m = 3   n = 10   numItems = 18   const = 376
  ```
- Each row after this first defining row contains information about exactly 1 non-zero element – the row and column coordinates of the cell where it is stored, and its value. For example,

  ```
  2,8 = 5708
  ```

  means that the element in row `2` and column `8` has value `5708`.
- The elements in the test files are ordered by column first, and then by row. That is, element `(4,1)` would precede element `(2,3)` (since column 1 comes before column 3), which itself would precede element `(5,3)` (since both have the same column 3 but row 2 precedes row 5). Look at the test case files for more examples.

The test case files will be available shortly at:
http://www.cs.washington.edu/education/courses/cse326/04wi/assignments/wet1-tests/

Development environment.
As an implementation language you are allowed to use either Java or C++. Be sure to avoid using features that are specific to a particular version of a compiler, since if you do so we likely will be unable to compile your code and you will lose points.

Submission.
The submission is electronic and part of it is also paper-based, as specified below.
We need *only one* electronic submission and *only one* printed submission per team.

What to submit. All of the following (1-5) need to be submitted *electronically* by class time on the due date. You should archive all files (winzip, tar, etc.) and submit only the single archive file.

1) Your code – the sparse matrix class `SparseMatrix.java` or `SparseMatrix.cpp` (depending on which language you use) with the code for all of the implemented operations. Also include any supporting headers, packages, and the test module(s) you wrote. Include only the source code---no .jar, .vcp or other project files. C++ programs should only use the standard C or C++ libraries, no MFC, GTK or other libraries. Don't use any fancy build options (precompiled headers, etc.).
2) A description of all modules your code consists of (including module names and a one-paragraph description of what each module does), and *clear* instructions on how to compile and run your code, as well as how to test it on different test case files (we might create). Put this in a plain text file, called `description.txt`.
3) Your brief argument about what the time and space complexities are for each operation. Put this in a plain text file, called `complexities.txt`.

4) A brief description of which test cases (from those we provided) you have tested your code on; name them explicitly. Did all of your tests give results as expected (as in the test results we provided)? Did you encounter any unusual (special) cases and if so how did you deal with them? How would you deal with the special case of a matrix of all zeros – what is a good way to store it and does that affect time and space complexities of future operations on such a matrix? Put this description in a plain text file, called `testing.txt`.
5) A brief description of who (of the two partners in the team) did what part of the assignment: the different parts of the code, the testing, the complexity arguments, and anything else you did. Be sure to include your names and emails here. You are encouraged to work on all parts together, and if so, indicate that this is indeed the case. If, however, you decided to split the work, we also need to know that, as well as how you did it. Put this description in a plain text file, called `contributions.txt`.

In addition, *print* and bring to class on the due date:
1) The code of the SparseMatrix class *only*. (Please, *do not* print header files, packages, or test modules/classes!)
2) The description text files 2-5 (described above). To save paper and trees, please merge them together into one file and then print.

Do not forget to write the names of both partners on the printed sheet you submit.

Printing guidelines. Double-sided printing and condensed code printing (i.e., double-sided at 2 pages of code per side) is highly encouraged! Modern printers, including many of those available at UW facilities, can do that.

Where to submit. Instructions on where to electronically submit your assignment will appear on the course web page and will be announced on the mailing list.

Quality criteria.
For full credit, you need to meet the following quality criteria:
1) Your submission needs to contain all parts described in the previous section.
2) Your code needs to:
   - Compile and run without errors or warnings. (We will follow the instructions you will have provided us with, so those need to be accurate.)
   - Give the same results on the test cases as the test results provided by us (unless you found an error in our test results).
   - Contain comments, making it easier for a human to understand what you have done.
3) Your argumentation about the time and space complexities of all operations needs to be sound. Brevity is encouraged but not at the expense of clarity or soundness.
4) Your printed materials need to look professional (i.e., be printed and stapled together rather than hand-written and loose).

Good luck!