COSC 1030 Introduction to Computer Science II
Assignment 2: Sparse Matrices
Due: Wednesday, February 12, 1997, 10:30am

1. Main Points

1.1 Learning objectives

- Pointers
- Abstract Data Types, Classes and Objects
- Design, Implementation, and Documentation
- Testing -- use of assertions and debugging statements

1.2 What is given

The following files are on ariel in `/cs/course/1030/a2/`.

- cover2.ps – the cover page for the assignment (postscript)
- a2spec.ps – this document (postscript)
- `matelem.tu` – implementation of the element class for sparse matrices
- `spmat.dem` – a demonstration program that you use to test your program
- `sample.in` – an example input file that you use with `spmat.dem` to test your program
- `sample.out` – an example script created with `spmat.dem` using `sample.in`
- `debug.tu` – module containing a set of debug flags.
- `spmat.tu` – implementation part 1 of the sparse matrix class.
- `spmat2.tu` – partial implementation of part 2 of the sparse matrix class, **this is the only file you may modify for the assignment.**

1.3 To hand in

A professional report describing the work you did to fulfil the specification in this document. In addition to your written report, the report includes the following items.

1. Cover page `cover2.ps` – **must** be the first page of your report ***
2. Report on your solution to the problem posed in section 3.1
3. Completed, `spmat2.tu` for as many of the tasks in section 4 as you can do
4. `Script from running smat.dem on spmat.tu using sample.in`
5. Script from running `spmat.dem` on `spmat.tu` using `your own test file`

***If the cover page does not appear as the **very first page of** document you will lose one full letter grade (e.g. A becomes B, C+ becomes D+, and D or lower becomes F). This means **no binders or envelopes**.

You should make use of assertions and debugging statements and flags (so you can turn debugging on and off) to help you test and verify the correctness of your implementation.

1.4 Readings

- Chapter 3 – design, loop invariants
- Chapter 4 – abstract data types, linked list implementation of address book
- On-line notes on design and verification – ariel:/cs/course/1030/notes/onDesign
- On-line examples from the textbook – ariel:/cs/course/1030/text-code/ch04

2. Assignment Overview

You are learning about the design and implementation of abstract data types by studying an example for two-dimensional sparse matrices, see Figures 1. The idea can be extended, with more programming effort, to matrices with any number of dimensions.

A sparse matrix is a matrix with predominately 0 entries. Rather than taking up memory storing the zeros, only nonzero entries are stored. For example in Figure 1 there are only four matrix elements stored (represented with squares containing the values 3, -1, 6 and 10) whereas the 8x10 matrix contains 80 elements (see Figure 3 for a standard two-dimensional matrix). In addition to the user data (the data values for each matrix element) there is a need to store **metadata** – data about the user data. For the sparse matrix, metadata consists of the row and column header lists, and the row and column maximum values. Each matrix element appears in a singly linked list of elements in the same row, and a singly linked list of elements in the same column.

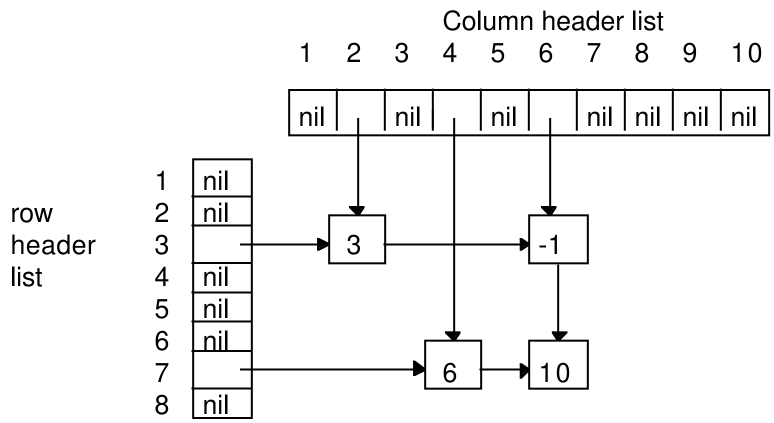

Figure 1: Example of an 8x10 sparse matrix with 4 nonzero elements at <3,2>, <3,6>, <7,4>, and <7,6>. All other elements are assumed to be zero. Compare with Figure 2.

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	6	0	0	0	-1	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	6	0	10	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0

Figure 2: The standard way of storing the 8x10 matrix shown in Figure 1. All the zero elements take up space.

Each matrix element (see Figure 3) contains, in addition to the user data, metadata consisting of the row and column position for the element (must be explicitly stored as physical position cannot be used; for example, the -1 is physically the second element in row 3 but logically it is in column 6), and pointers to the next row element and the next column element. The pointers are required to link elements in the same row and column so they can be easily found.

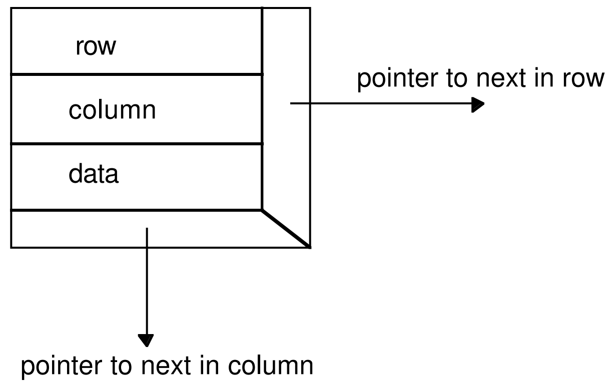

Figure 2: Shows the fields in a typical sparse matrix element.

You will study and use an implementation of a sparse matrix element class, `matelem.tu`, to learn about abstract data types, hiding implementation details and classes. You will extend the implementation of a sparse matrix class, `spmat.tu`, to do the following.

- study the design of abstract data types
- use classes to organize program structures
- use pointers to organize and process collections of data.

Since programs must satisfy their specifications and work correctly, you will continue to practice the use of debugging statements, assertions, and test drivers. At the design level you will practice top down design on various procedures and functions. In particular, documenting your designs with pre and post conditions, assertions and loop invariants.

3. Study the Problem Context

3.1 Determine space saved with sparse matrices

Assume that in a sparse matrix implementation, the data values, the row numbers, the column numbers and the pointers are all 4 bytes in length. Consider a square matrix, RxC where R = C. At what ratio of nonzero elements to the total number of matrix elements (=RxC) does a sparse matrix use less space than the standard method of storing matrices. In computing space requirements you need to include not only the user data but also the space required for the metadata. For the standard storage method, metadata space can be considered to be zero. What is the asymptotic amount of overhead for square matrices with all nonzero elements? What is the big O value for overhead space?

Now consider the three questions in the preceeding paragrph for vectors (one-dimensional matrices). Do the answers change? Explain why or why not.

3.2 Study the file matelem.tu.

Compare the items in Figure 2 with the data declarations in the file. Note the correspondence. For every item in the figure there is a corresponding data object in the class. There are get and set operation for each field in a matrix element to hide their implementation – the user sees only the interface representation. Using the get and set operations means the implementation can change without changing other parts of a program that uses sparse matrix elements. This is a standard OO (object oriented) design technique. For matrix elements it makes sense to be able to access all of the fields.

3.3 Study the file spmat.tu

Compare the items in Figure 1 with the data declarations in the file – note the correspondence. While the SparseMatrix class has get operations, these are only for a subset of metadata objects – not all of the information in the class needs to be know by the user. There are no set operations for individual objects. All objects are maintained indirectly through utility operations such as, for example, `Insert`, `Put`, and `Add`.

Study the loop invariants. They are written in a mixture of mathematics and English. You have to write loop invariants for the operations you design. Try be as mathematical as possible, but use natural language, if you are unable to give a mathematical description or the mathematics becomes too cumbersome.

To help you understand the use of pointers and classes study the given operations such as the iterator, `InitIterationByRow` and `NextIterRowElem`, and operations such as `PutElemByRow` and `GetCountByRow`.

3.4 Study the demonstration program

Examine the given demonstration program `spmat.dem` to see the specifications for the input commands. Also examine the files `sample.in` to see how the commands are used and `sample.out` to see their effect.

4. To Program

4.1 Initialization

Complete the implementation of `Initialize`.

4.2 Assertion testing

Implement the GetData and ExistsElement operations.

Start writing a no input demonstration program `massert.dem`. To test GetData and ExistsElement and the interface for the demonstration program, you can experiment with inserting elements into a matrix with explicit, hardwired pointer changes. The following is an example that inserts two matrix elements in row 4 of an empty matrix.

```
var elem1, elem2 : ^MatrixElement
new elem1 ; elem1->Initialize(4,2,5)
new elem2 ; elem2->Initialize(4,6,7)
rowList(4) := elem1
elem1->SetNextRowElem(elem2)
colList(2) := elem1
colList(6) := elem2
```

You cannot do the above in the demonstration program – it does not have access to the metadata. But you can do it in the Insert operation which you then call to create a test matrix. After calling Insert in the demonstration program, you could do the following (assuming the matrix is called a).

```
assert(not a->ExistsElement(4,5,'r')
assert(a->GetData(4,5,'r') = 0)
assert(a->ElementExists(4,6,'c'))
assert(a->GetData(4,6,'c') = 7)
assert(a->GetData(4,6,'r') = 7)
```

The practice will help you understand pointers before you attempt to implement the other operations in the SparseMatrix class.

Insert assertions into `massert.dem` to test your implementation of Insert (section 4.4) and Sum (section 4.7) as you develop them. The demonstration program could contain statements like the following. Suppose an earlier part of your demonstration program created matricies A and B with location <4,4> being +5 in A and -5 in B, then the following could be done.

```
s->Sum(a,b)
assert(s->GetData(4,4,'r') = 0 )
assert(not s->ExistsElement(4,4,'c'))
```

4.3 Use of debug flags

The module `debug.tu` contains a set of flags that you can use for debugging statements. The module also contains operations Set, Reset , and Toggle so you can change value of the flags. Use the debug flags in a similar way as in Task 5 in Assignment 1 to show what is taking place in your program. There is more than one flag so you can turn different debugging features on and off independently. The following is an example use of the debug flags. Depending upon the combination of values of `flag0`, `flag1`, `flag2`, `flag3` and `flag4`, different combinations of Block 1 .. Block 4 are executed.

```
if Debug.flag0 then
    Block 1
end if
if Debug.flag1 then
    Block 2
end if
if Debug.flag3 then
    Block 3
    if Debug.flag4 then
        Block 4
    end if
end if
```

The demonstration program `spmat.dem` contains commands to set, reset and toggle the flags. This enables you to create test files that can produce different debugging output that you can script, print and examine. Make use of debug flags as you develop the algorithms in the following sections.

4.4 Insert operation.

Complete the implementation of `Insert`. This is an exercise in top down design; also known as stepwise refinement.

You must manipulate the pointers directly, you cannot make use of the iterator operations. It is strongly recommended you use procedures or functions for subgoals.

Use the no input demonstration program (see section 4.2) and debug flags (see section 4.3) to test your incremental development of the algorithm.

4.5 Finalize operation

Complete the implementation of `Finalize`. Do not use the iterator operations but make direct use of your knowledge of the implementation.

To test `Finalize` you should reduce the count as you free each matrix element. During testing this is better than just setting the count to zero as it is verification that every matrix element was freed. This is an example of a good place to use debugging statements and flags.

4.6 Put operation

Complete the implementation of `Put` using the given iterator operations, `InitIterationByRow` and `NextIterRowElem`. For a matrix with no rows, or columns, print the message “The matrix is empty”.

The following is an example output of Put for a 3x4 matrix. This version of Put will produce, according to the postcondition, good looking output (on standard letter sized paper) for matrices with less than 16 columns.

```
         1    2    3    4
       ____ ____ ____ ____
    1|   **   **   -1   **
    2|   **    1    2   **
    3|   **   **   **    6
```

Use debug flags (see section 4.3).

4.7 Sum operation

Complete the implementation of Sum using the iterator operations. Consider what to do if a matrix is being added to itself – explain what problem exists, why it occurs, and how you solved it.

The basis of the Sum algorithm is the merging of elements from both matricies. The iterator retrieves the operand elements, matElem1 from matrix1 and matElem2 from matrix2, in row order and the sum is produced in row order. The operand elements matElem1 and matElem2 are compared with respect to their row and column location. If their locations differ, then a copy of the earlier element is inserted into the sum matrix. The earlier element is replaced by the next element from the same matrix. If their location is the same, then the data values are summed and that value stored as an element in the sum matrix at that row and column position and both operand elements are replaced by the next element from their respective matricies.

The most obvious algorithm is to use Insert to insert the sum elements into the sum matrix. This algorithm takes more than linear big O time. A linear big O time algorithm can be written by directly linking the new element at the tail end of its row list and the tail end of its column list. Since the sum is being computed in the environment of the sum metadata all the information you need is available to do direct linking.

As a first pass you may want to write the simpler algorithm using Insert. Higher marks will be given for a correct algorithm that directly links the new element into its proper position.

Use the no input demonstration program (see section 4.2) and debug flags (see section 4.3) to test your incremental development of the algorithm.

4.8 Test script

Design your test script, `spmat.in`, so it shows that the operations you implemented work. One run should show everything. You can wrote descriptive comments in the input file and also hand annotate the script to highlight interesting features of the script.