CSE 373 -- Spring 1999 -- Assignment #6 -- Sparse Array ADT

(due Wednesday, May 19 -- proposal
Wednesday, June 2 -- implementation)

The Goal: This assignment is designed to give you experience implementing a realistic ADT -- the sparse array. The project will require you to decide between multiple implementation strategies, each of which will have advantages and disadvantages. You will need to investigate and explain the advantages and disadvantages of two implementations, and then implement one of them. You may work in teams of two for this project or by yourself. The amount of work involved shouldn't be significantly more than on previous assignments, but having a partner should be useful for brainstorming (and making your end-of-quarter workload slightly lighter).

Definition of Sparse Arrays: A sparse array is a data structure that is used to represent an array of values. The characteristic that differentiates sparse arrays from normal arrays is that the majority of its elements are identical in value. We'll call this value that is highly repeated the *unrepresented value* (URV for short; in most real-world applications of sparse arrays, the URV tends to be "0"). Sparse array implementations take advantage of this repetition in order to reduce the amount of memory and time required to store and operate on the array.

For this project, we'll only be considering 2D sparse arrays of `doubles`, though in general sparse arrays can have arbitrary type and dimension. We'll use two variables to specify the problem size of a sparse array implementation: n, the size of the sparse array in one dimension (assuming for simplicity that it's square; we would use n_1 and n_2 to characterize its size when it's nonsquare). The other variable is r, the number of *represented* values in the array -- that is, the number of values that are different than the unrepresented value.

As an example, the array on the left would be a normal 2D dense array, whereas the array on the right is a sparse 2D array due to all of the 0's that it contains:

```
1.1  1.2  1.3  1.4  1.5  1.6      1.1  1.2  0.0  0.0  0.0  0.0
2.1  2.2  2.3  2.4  2.5  2.6      2.1  2.2  2.3  0.0  0.0  0.0
3.1  3.2  3.3  3.4  3.5  3.6      0.0  3.2  3.3  3.4  0.0  0.0
4.1  4.2  4.3  4.4  4.5  4.6      0.0  0.0  4.3  4.4  4.5  0.0
5.1  5.2  5.3  5.4  5.5  5.6      0.0  0.0  0.0  5.4  5.5  5.6
6.1  6.2  6.3  6.4  6.5  6.6      0.0  0.0  0.0  0.0  6.5  6.6
```

Both arrays would have $n = 6$. However, the array on the left has $r = 36$, whereas the one on the right has $r = 16$. The array on the right is a special kind of sparse array known as a *tridiagonal* array, because it only contains values along its main diagonal (where *row* = *col*), and in the positions directly surrounding the main diagonal.

Obviously, we could store either of these arrays using the C declaration: `double myarray[6][6]`. However, whereas the array on the left contains $O(n^2)$ data, the second contains $O(3n) = O(n)$ elements (since there are never more than three values in each of the n rows). Thus, if we used C arrays to store larger and larger versions of this sparse array, we'd be using $O(n^2)$ space to store its $O(n)$ represented values. This is highly wasteful. The goal of our sparse array ADT therefore, is to store a sparse array's values in $O(n)$ space rather than $O(n^2)$. An even better representation would

use O(*r*) space, though this is not a strict requirement of this assignment.

One last definition: The *density* of a sparse array is the number of represented values divided by the total number of values that it represents. For example, the density of the array on the left above would be 1.0, whereas the density of the array on the right would be 16/36 = 0.44.

Applications of Sparse Arrays: An intuitive example of a sparse array is the UW registry that we talked about during the first week of class. At that point, we said that it could be implemented using a large array of size *#students* x *#classes* in which a value of "1" would represent that the student was taking the class in question and a value of "0" would represent that the student was not taking the class. We quickly decided that this would be a wasteful implementation since most students only take a few classes. Therefore, the bulk of the array would be 0's, with only a few 1's scattered around in it. In contrast, if we were to use a sparse array representation, only the 1's would be represented, and thus our representation would use space proportional to the number of 1's (which would be O(*#students*), since each student would take a maximum of 10 classes or so...).

Many large scientific simulations also make heavy use of sparse arrays. Boeing uses them to model airplanes and run simulations over their designs. Oceanographers use them to store data at coastlines (since coastlines take up a relatively small fraction of the earth's surface). Etc. etc. I'm not going to describe lots of applications here, but if you're interested in talking about them, feel free to ask me about it in office hours or on email.

Sparse Array Operations: The operations for our sparse array ADT will fall into five general classes:

- *creation/deletion* -- create new sparse arrays; delete old ones
- *storage* -- store, read, and access a sparse array's values
- *I/O* -- print sparse arrays out; read sparse arrays in
- *iteration* -- iterate efficiently over a sparse array
- *miscellaneous* -- query a sparse array's density or # of represented values

Each of these will be described at a high level below. The header file giving the prototypes for all the operations are given on the course web page, with comments describing how they should work.

> **Creation/Deletion Operations:** These are what you'd expect: ways to create a new sparse array and free up all the memory used by one. Creating an array involves specifying a number of rows and columns as well as a value to be used as the unrepresented value. Deleting an array should free up all memory used by the array representation. Sparse arrays can also be created by reading them in from a file/stream -- these will be described in the I/O section below. In C++ these are implemented as overloaded constructors.
>
> **Storage Operations:** These are the basic ways of accessing data in the array and storing new data. There are three main operators which allow users to read a value from the array, store a value into the array, and get a pointer to an array value if it exists (this will return NULL otherwise). All of these routines take a row and column index as arguments. We will assume a 0-based indexing scheme to be consistent with C. Other storage operations allow the user to query and change the unrepresented value. Note that there is no way to delete data from a sparse array in our representation.

One key detail is that when reading a sparse array, it should give the appearance of a normal dense array: represented values should be returned as expected; however, if an unrepresented value is read, the unrepresented value should be returned.

I/O Operations: With most data structures, it's useful to be able to read and write them to a file or the console. With sparse arrays we provide two formats. One is a "dense" format which prints out the array as though it was a normal 2D array (a dense printout might look like the 6x6 sample arrays shown above). When reading in the dense format, any values that are equal to the unrepresented value should obviously not be stored.

The other format is a more compact "sparse" format which prints out the size of the array, the unrepresented values, and then the represented values and their locations. For example, the sparsely formatted representation of the tridiagonal array above would be stored as:

```
6 6
0.0
(0,0) 1.1
(0,1) 1.2
(1,0) 2.1
(1,1) 2.2
(1,2) 2.3
(2,1) 3.2
... etc. ...
(5,5) 6.6
(-1,-1) -1
```

(Note that the list of represented values is terminated by the sentinel position (-1,-1) to mark the end of the list). These routines take in a FILE* (in C) or an ifstream/ofstream (in C++). You should assume that the file/stream has been opened outside of the routine (or that stdin/stdout/cin/cout are passed in) and that the file/stream will contain data in the correct format. Any array written with your `WriteDense()` operation should be readable with your `ReadDense()` operations; similarly, any array written with your `WriteSparse()` operation should be readable with your `ReadSparse()` operation.

Iteration Operations: The user could obviously iterate over the represented elements of a sparse array by using a nested for loop:

```
for (row = 0; row < numrows; row++) {
  for (col = 0; col < numcols; col++) {
    if (Access(myarray,row,col) != NULL) {
      /* do something to the element here... */
    }
  }
}
```

However, this will be an $O(n^2)$ operation which is a waste of time if the number of elements is only $O(n)$. Thus, the sparse array ADT supports a way to iterate over the data in $O(n)$ time. These routines use a *cursor* implementation in which the data structure keeps track of what the "current" position is in a variable known as a "cursor". Different routines allow the user to set the cursor or move it to an adjacent represented value in the current row or column. In this way, iteration over the array in $O(n)$ time is possible (assuming it holds $O(n)$ data). For example, the loop above would be written more efficiently as follows:

```
for (row = 0; row < numrows; row++) {
  col = SetCursorRowStart(myarray,row);
  while (col != -1) {
    /* do something to the element here... */
    col = NextInRow(myarray);
  }
}
```

Miscellaneous Operations: In addition to the operations above, operations to query the number of represented values in an array or its density are supported.

Implementation: How you implement your sparse array is completely up to you. My only requirements are that your implementation conforms to my header file, uses only O(n) or O(r) space (*not* O(n^2)), and that you use something more interesting than a 1-dimensional List (which would work, but is extremely inefficient and doesn't show much creativity). As a starting point, I would recommend going through all the data structures that we've talked about in class in a manner similar to what we did in lecture on Monday for the mode question on the midterm. Judge whether the datatype would be appropriate, which operations it would support well, which operations it wouldn't support well, and how you might modify it from its "pure" form to improve it (this will be the topic of Wednesday's lecture, "Breaking Out").

The most naive implementation would be to use a variation on the List ADT called a multilist, which is described briefly in Chapter 2 of the textbook. This is a fine solution, and may be one of the easiest solutions, depending on how easy you find working with linked lists (if you don't find it easy, another data structure may be easier and more efficient...).

Thinking briefly about other possibilities, Binary Search Trees seem similar to what we want to do, since they support fairly efficient storage, searching, and traversal. How could they be used to implement this ADT? Hash Tables support efficient storage, but in their most basic form don't efficiently support the iteration that we need. How would the implementation have to be modified in order to support iteration?

Note that you can implement several of the operations using the other operations that are available to you. This is fine with me, and generally a good idea when implementing ADTs. In some cases, you may be able to implement them more efficiently by referring directly to your data structure. For the most part efficiency won't matter to me as long as you can correctly describe the running time of the operations.

You're welcome to use any implementations of any ADTs that we've been using (Strings, Vectors) or that we've studied in this class (Lists, Stacks, Queues, Trees, Binary Search Trees, Hash Tables, Heaps) if it makes your life easier.

Grading: The most important thing that you'll be graded on is whether or not your implementation works correctly. I'll be coding up several tests that use the interfaces I've provided you, and will expect to be able to compile my program using your ADT and run it without any problems. For this reason, you should test all of your operations thoroughly.

The second most important thing is that you can judge how expensive your operations are. Obviously, the faster they are (asymptotically) the nicer your implementation is, but this will be a

miniscule fraction of the program's worth (around 5%). If you write an O(n) operation where an O(1) operation would have sufficed, the most important thing to me is that you know it's an O(n) operation. That said, none of your operations (other than I/O and deletion) should be as expensive as O(r) or O(n^2), since that would imply that your data structure requires blindly searching for values rather than doing something logical and direct.

I'm hoping to do grading interactively so that you can describe to one of us how your implementation works and we can compile and run it right there. I haven't figured out whether or not this is going to be logistically possible yet, though.

Racing (*optional*): Those of you who enjoy competition are welcome to make your implementation as efficient as possible to see whether you can beat out your equally-competitive classmates. No-one is required to participate in this race, and it will not affect your grade in any way. My intent here is that you work at the algorithmic and data structure level, making your code as asymptotically fast as possible, and then making the secondary effects as small as possible. I *do not* want anyone doing crazy stuff like coding in assembly or doing stuff that's really hacky. Your implementation should still be reasonably clean and understandable, just fast. And obviously, you shouldn't let your competitive streak get in the way of completing a correct implementation (since that's what will be graded).

The Proposal: The first thing that you need to turn in is a proposal that considers two (2) possible implementations that you came up with while brainstorming (even though you will probably come up with several while brainstorming, do not submit more than two). The proposal should detail the benefits and drawbacks of each. You should think of this as being similar to the mode midterm question: I want you to describe to me in English and in a reasonable amount of detail (so that I understand what you're doing) how the data structure works, how much space it would take, and how much time the interesting operations would take (the ones that are hardest to make fast):

- `Read()`
- `Write()`
- `Access()`
- `SetCursorRowStart()` (*assuming the other 3 variations are the same -- if not, say why and what they would cost*)
- `NextInRow()` (*assuming the other 3 variations are the same -- if not, say why and what they would cost*).

Your proposal should conclude by stating which implementation you will be doing.

The proposal has three purposes: the first is to make sure that you get working on this before the last minute. The second is to see whether you can invent and evaluate different approaches to a problem like this. The third is to give me a chance to warn you away from a bad or extremely tricky implementation.

Do not make the two data structures you consider in your proposal ridiculously similar: For example, a singly-linked multilist and a doubly-linked multilist are not different enough to make an interesting proposal. A doubly-linked multilist and a modified hash table would be much more interesting. Similarly, you shouldn't propose any solutions that wouldn't meet the project requirements to begin with; both should be valid implementations if you were to carry them out. If you have any doubts as to whether or not your proposed implementations meet these requirements, feel free to ask the course staff.