CSE 373 -- Autumn 1999 -- Programming Assignment #4 -- Sparse Array ADT

(*due Wednesday, Dec 8*)

The Goal: This assignment is designed to give you experience implementing a realistic ADT -- the *sparse array*. In doing so, you will need to decide between different implementation choices, each with its own advantages and disadvantages. You will have the option of using the ADTs we've studied in class or creating your own, just as real-world programmers have to. As with the previous project, spending a fair amount of time planning out your approach before diving into the coding will save you quite a bit of effort, so spend some time thinking about the assignment before you start typing.

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 a 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 zero. 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 example, if a 100x100 array only has 5 nonzero values, we ought to be able to just store those 5 interesting values and not the other 9,995 zeroes.

For this project, we'll only be considering 2D sparse arrays, though sparse arrays can generally have arbitrary dimension. We'll use three variables to specify the problem size of a sparse array implementation: n_r, the number of rows in the array, n_c, the number of columns in the array, and r, the number of *represented* values in the array -- that is, the number of values that are different than the unrepresented value. For this assignment, we will assume that $r = O(n_r + n_c)$.

As an example, the array on the left is a normal 2D dense array of doubles, whereas the array on the right might be stored as a 2D sparse array due to all of the zeroes 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 of the arrays above have $n_r = n_c = 6$ (so we'll simply call this n for this example). 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 three main diagonals (at locations where *row* = *col* +/- 1).

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 one on the right 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, which is incredibly 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 requirement for 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 above arrays would be 1.0 and 0.44 (16/36), respectively.

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 throughout 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 no more than 10 classes or so, we can therefore treat it as a constant).

Many large scientific simulations also make heavy use of sparse arrays. Boeing uses them in airplane design to perform simulations over aircraft hulls. Oceanographers use them in simulations to store coastline-specific data (since they take up a relatively small fraction of the earth's surface).

Implementation: As with the other collection ADTs that we've studied, the type that is stored in the sparse array is not important to its implementation. Thus, your implementation of the sparse array ADT will be as a C++ template class. We will provide you with a header file that describes the interface that you must implement. You will need to come up with a sparse array implementation that requires no more than $O(n_r + n_c)$ space. As an added challenge, you might try to implement the sparse array so that it only requires $O(r)$ space.

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

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

Each of these will be described in broad terms below. The header file will describe the operations in more detail. If you have any additional questions about their behavior, email the staff.

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.

Storage Operations: There are two main storage operations: one will allow users to store a value into the array, the other will allow them to read a value from the array. Each of these operator has an interesting case to deal with:

- If the storage operator is used to store the nonrepresented value, this should be implemented by removing the value from your data structure (i.e., it should no longer be stored explicitly, but rather implcitly using the URV). For example, if I store the value 1.0 at location (0,0) and then later replace it with 0.0 (and 0.0 happens to be my URV), you should no longer store the value explicitly, but rather remove your storage of location (0,0) and treat it as the URV once more.

- When reading a value from the array you should return a value whether the location is explicitly represented or not. Thus, if the value is explicitly represented, you would return the value being stored; if it is not, you should return the URV.

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 would like to be able to do this in two formats. One is a "dense" format which prints out the array as though it was a normal 2D array (a dense printout would look like the 6x6 sample arrays shown above). The other format is a more compact "sparse" format which prints out the size of the array, the URV, and then the represented values and their locations. For example, the sparsely formatted representation of the tridiagonal array above would be printed 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).

Any array that you write in the sparse format should be readable with your sparse read function. Similarly, any array that you write in the dense format should be readable in with your dense read function (though it will also need to take the URV as a parameter to be sure that it does not store that value).

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 = numcols-1; col >= 0; col--) {
    if (myarray.Read(row,col) != URV) {
      /* 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 represented values is only O(n). Thus, the sparse array ADT supports a way to iterate over the data in O(n_r + n_c) time.

Our sparse array iteration will use an implementation in which the ADT keeps track of what the "current" position is in a private variable known as a *cursor*. To begin using the cursor, the user specifies the order in which they want to iterate over the array's dimensions (for example the loop above iterates from the lowest row to the highest row, and in each row it starts from the highest column and goes to the lowest column). Once the cursor has been reset, its position can be advanced which will move it to the next represented value. In addition, the current position of the cursor can be queried. In this way, iteration over the array in $O(n_r + n_c)$ time is possible. For example, the loop above could be written more efficiently as follows:

```
int row, col;

myarray.SetupIteration(1,-2);
myarray.GetCursorPos(row,col);
while (row != -1) {
  /* do something to the element here... */
  myarray.Advance();
  myarray.GetCursorPos(row,col);
}
```

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 $O(n_r + n_c)$ space, and supports all operations in $O(n_r + n_c)$ time. As a personal challenge, you should try and make all the operations as fast as possible, but correctness is more important than efficiency for your final grade (just make sure to adhere to the given limits).

As a starting point, I would recommend going through each ADT that we've talked about in class to judge 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 upon it.

The most naive approach to this ADT 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 completely reasonable approach for this assignment, and it should be fairly straightforward if you don't mind playing with pointers a bit (if you do mind, another approach may be easier and more efficient...). Other approaches can be worked out that are based on hash tables, binary search trees, heaps, or some other combination of the data structures that we've studied this quarter.

Turn-in: In addition to your code, you will need to turn in a written description of your ADT's design. This will be a significant part of your grade, so don't put it off until the last minute (or expect that if you have the coding done by the deadline that you can turn in the written part sometime afterwards). This description should:

- Describe the approach you took -- what data structures did you use, how did you use them, how did you implement your cursor, what advantages and challenges did your approach have?
- Include a picture of your ADT storing a sample sparse array (you may use the example from above, for example).
- Give the running time for each of your operations and a brief (but accurate) argument of why it has that running time.

After reading this description, we should understand your approach completely, so make sure it's clear.

Grading:

The most important thing that you'll be graded on is the correctness of your write-up: does your approach meet the goals of the ADT and is your analysis of your operations' running times correct. The next most important thing will be 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.

Competition (*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 similarly-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).