Appendix A

Sparse Arrays Revisited

As mentioned at the end of chapter 3, there are a number of ways to represent sparse arrays. Which representation is best depends largely on how the arrays will be used.

Now that we have had a chance to see several other data structures, it makes sense to revisit the topic of sparse arrays and see how these new data structures can be used in this context.

A.1 Hash Tables

A hash table is a natural way to represent an array. (In fact, several interpreted languages implement all of their arrays as hash tables.)

The indices of the element being accessed are used as the argument to the hashing function. Once the hash bucket where the element resides has been determined, the bucket is searched for the element with the correct coordinates. If the element is missing, then it is assumed to have the default value.

The benefit of this approach is that if the hash function is effective (i.e. each hash bucket doesn't have too many elements in it, so that the search is fast), then accessing an element in the array is effectively O(1).

However, achieving this level of performance is not simple.

First, the hashing function must be chosen so that is relatively unlikely that a large number of the array elements will hash to the same bucket– for sparse matrices, this is not usually a problem (since there are relatively few interesting elements to begin with), but as the array becomes less sparse, the size of the hash table must grow. Eventually, it may become large enough so that there is no savings realized by using the hash table. In addition, if the pattern of the non-zero elements in the sparse array triggers degenerate behavior in the hash function, the hash table may become quite slow.

Second, the hashing function must be quick to compute. Even though a hash table can (at least in theory) implement array element access operations in $O(1)$ time, the constant overhead is much higher than that of ordinary array element access.

One large problem with the hash table representation is the lack of information about neighboring elements that can be easily derived from an element. Using the linked list method, it is easy to progress from non-zero element to non-zero element– just a matter of traversing along the linked list. In a hash table, the closest thing would be traversing through a bucket– but the elements in a particular bucket will typically represent elements scattered throughout the array, and there is generally no useful meaning to the order of the elements in the bucket.

A.2 Binary Search Trees

Instead of using linked lists to represent the rows and/or columns of sparse arrays, we can use binary search trees. If the trees are kept balanced, then this means that array element access can be performed in $O(\log x)$ time (where x is the number of non-zero elements in the given row or column). With some care, even a completely full n-by-n array can have its elements accessed in $O(\log n)$ time. While this is quite a bit more than the $O(1)$ required by the ordinary representation, it is also much less than the $O(n)$ required in the worst case by the linked-list approach.

Implementing binary trees instead of linked lists adds a degree of difficulty to the representation– keeping trees balanced is challenging, and traversing the tree inorder (to achieve the same effect as traversing the linked list) is somewhat more so. However, these problems have all been solved before, so they should present no overwhelming obstacles.

A.3 Exercises

1. Implement a library of functions to implement sparse arrays as hash tables. Make sure that you address the issue of finding a hash function that works well for arrays of different sizes.

2. Devise an efficient algorithm that finds the intersection of two binary search trees (i.e. the list of keys that appear in both trees) and prints it in order. The algorithm must visit each node in both trees *at most* once.

3. Using the algorithm from the previous exercise, implement the fast matrix multiplication algorithm for sparse arrays described at the end of chapter 3, modified to work with sparse arrays represented by binary trees.

4. Devise an efficient algorithm that finds the union of two binary search trees (i.e. the list of keys that appear in either trees) and prints it in order.