Homework 4

MPCS 51100 due: Nov. 12

In this assignment we consider a special lookup problem in the case that we want to find the same key in many "similar" arrays. It is assumed that the arrays are sorted and that the sorting cost is amortized over many lookups (rather than a single lookup we aim to do very many successive ones). "Similar" can be considered qualitatively to mean that the keys are all sorted and that the arrays have lengths and key ranges that do not differ drastically. This problem is a good illustration of tradeoffs between memory and time complexity, and is a good C programming exercise.

Let $\{L_1, L_2, \ldots, L_k\}$ denote a set of k similar, sorted arrays with average length n. Suppose we wish to locate the bounding indices of some key q in each of the arrays. In other words, a binary search for a key should return either the index of the key if the key is present in the array or the index of the next largest value in the array otherwise. The most obvious solution is to perform a separate binary search on each array, as shown in Figure 1.

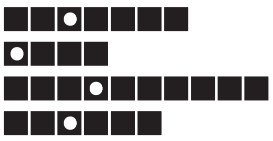

Figure 1: Original arrays where key is located using series of binary searches.

Suppose that instead of locating a single key in the arrays we want to successively locate many different key values. We can then use more intricate search techniques which pre-compute some information in order to save time later during the searches, improving the query time of the naive solution at the cost of a larger memory footprint. One such approach approach requires only a single binary search followed by k lookups using some pre-calculated pointer values to identify the location of q in each array (here "pointer" is used generally, not necessarily a C pointer). A new master *unionized array* is constructed as the ordered union of the values in each of the original arrays. Associated with each value in this unionized array is a series of k pointers, one for each of the original arrays, which identify the index in that array that would be returned in a search for that value. Figure 2 depicts the unionized structure and lookup process. Let U denote the unionized array and $\{P_1, P_2, \ldots, P_k\}$ denote the k pointer arrays associated with the original k arrays $\{L_1, L_2, \ldots, L_k\}$. If a binary search for q in U returns index j, the value of the j^{th} element in the pointer array P_i is the index of q in L_i.

Though this method speeds up the index search process, these pointer arrays can be quite large, and a third solution presents a compromise between memory requirement and execution time. This new data structure consists of the original k arrays, a new set of k augmented arrays, and a pointer pair for each value in the augmented arrays. The structure is designed to maintain a correspondence both between

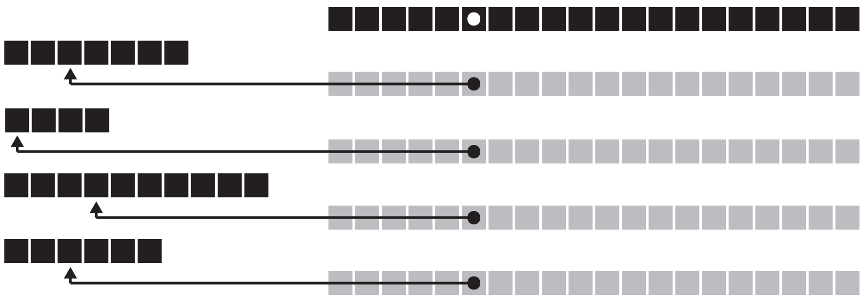

Figure 2: Unionized array structure where key is located using single binary search followed by pointer lookups.

the adjacent pairs of augmented arrays and between each augmented array and the original array it is associated with.

Let $\{M_1, M_2, \ldots, M_k\}$ denote the set of augmented arrays built from the original arrays $\{L_1, L_2, \ldots, L_k\}$. Furthermore, let each value $x \in M_i$ have two pointers, p_1 and p_2, associated with it. This augmented data structure is implemented in the following manner:

1. The final augmented array M_k is the same as the final array L_k
2. The augmented array M_i is a sorted array containing every element in L_i and every second element in M_{i+1}
3. The pointer p_1 associated with value x in M_i is the index of x in L_i
4. The pointer p_2 associated with value x in M_i is either the index of x in M_{i+1} or one more than the index of x in M_{i+1}

To locate q in each of the original arrays, a binary search is performed on only the first augmented grid M_1. The pointer p_1 associated with the returned index is the index of q in L_1, and the pointer p_2 is the approximate index of q in the following augmented array M_2. To determine the true position of q in M_2, a single comparison is made between the value at p_2 and the value at $p_2 - 1$. This process of following the pointers is repeated down to the last augmented array M_k. Figure 3 depicts the augmented array structure and lookup process.

As a simple example, consider the problem where you have the following four arrays and where you wish to perform a query for $q = 4.0$ in each array:

1.3	2.2	$\boxed{\mathbf{4.1}}$	7.3	8.8
$\boxed{\mathbf{4.8}}$	5.1	7.4		
2.6	2.8	$\boxed{\mathbf{4.7}}$	8.2	
1.1	$\boxed{\mathbf{5.2}}$	6.1	6.7	9.2

Using the naive method, you would simply perform a binary search on each array (returning 4.0 if it exists or the next largest value otherwise). To solve this problem using the second method, the following unionized array structure would be created:

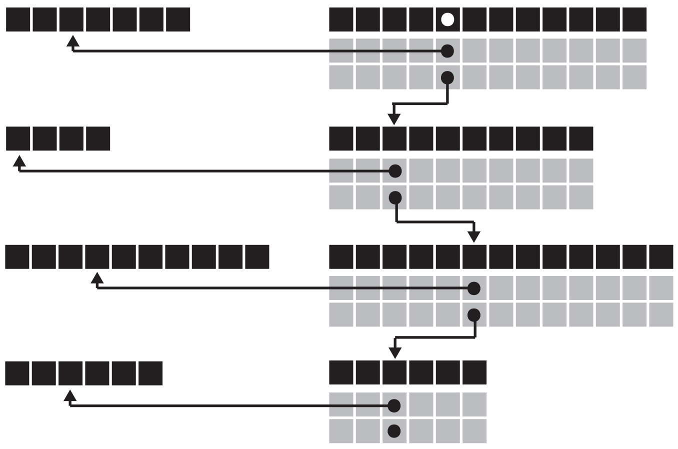

Figure 3: Augmented array structure where key is located using single binary search followed by pointer lookups.

1.1	1.3	2.2	2.6	2.8	$\boxed{\textbf{4.1}}$	4.7	4.8	5.1	5.2	6.1	6.7	7.3	7.4	8.2	8.8	9.2
0	*0*	*1*	*2*	*2*	***2***	*3*	*3*	*3*	*3*	*3*	*3*	*3*	*4*	*4*	*4*	*5*
0	*0*	*0*	*0*	*0*	***0***	*0*	*0*	*1*	*2*	*2*	*2*	*2*	*2*	*3*	*3*	*3*
0	*0*	*0*	*0*	*1*	***2***	*2*	*3*	*3*	*3*	*3*	*3*	*3*	*3*	*3*	*4*	*4*
0	*1*	*1*	*1*	*1*	***1***	*1*	*1*	*1*	*1*	*2*	*3*	*4*	*4*	*4*	*4*	*4*

The unionized array is shown on top with the pointer arrays below in italics. In this case, you would perform a single binary search for q on U which would return the index of 4.1. You would then use the corresponding pointers to locate the key in the original arrays, finding its position to be index 2 in L_1, index 0 in L_2, index 2 in L_3, and index 1 in L_4.

Finally, the third solution involves creating the following augmented array structure:

1.3	2.2	$\boxed{\textbf{4.1}}$	4.8	5.2	7.3	8.2	8.8
0	*1*	***2***	*3*	*3*	*3*	*4*	*4*
1	*1*	***1***	*1*	*3*	*5*	*5*	*7*

2.8	**4.8**	5.1	5.2	7.4	8.2
0	***0***	*1*	*2*	*2*	*3*
1	***3***	*3*	*3*	*5*	*5*

2.6	2.8	**4.7**	5.2	6.7	8.2
0	*1*	***2***	*3*	*3*	*3*
1	*1*	***1***	*1*	*3*	*5*

1.1	**5.2**	6.1	6.7	9.2
0	***1***	*2*	*3*	*4*
0	***0***	*0*	*0*	*0*

The four augmented arrays are shown with their pointers in italics below them. An initial binary search for q is done on the first augmented array M_1, returning the index of 4.1. The first pointer, 2, tells us the index of q in L_1. The second pointer, 1, tells us the approximate index of q in M_2, i.e. a binary search for q in M_2 would return either 1 or one position before, 0. We perform a single comparison of q to the element at index 0 in M_2 and find that q is greater, so the correct index is 1. Now we know q is located at position 1 in M_2, and we look at the corresponding pointers to locate q at index 0 in L_2 and at either index 3 or 2 (one less that 3) in M_3. After a single comparison we find that q is at position 2 in M_3. Continuing in this manner we can locate q in the remaining arrays.

For this problem, you will be provided with some data, the arrays $\{L_1, L_2, \ldots, L_k\}$. The file $arrays.txt$ contains $k = 68$ arrays of doubles of varying length. Each line of the file contains a single array. The first value of each line is the length of the array; the remaining values are the elements of the array. Read in the arrays and implement each of the three data structures described above. Then, write a lookup kernel which for some large number of iterations at each step generates a random double between the minimum and maximum values of the arrays and locates it in each array. Conduct a performance analysis comparing the three different search techniques.