CS29003 Algorithms Laboratory

Assignment No: 5

Last date of submission: 24–August–2017

The AlgoTooth Company produces smart tooth brushes that contain embedded processors. When one uses the tooth brush, sensors collect a lot of data, and the sensor co-processor sends these data to a hard disk in the brush stand, for future processing. It is often necessary to sort the saved data. The tooth brush has a computing processor for doing that. But that processor is not very powerful. More importantly, it does not have enough RAM (main memory) to copy the entire data from the hard disk. The AlgoTooth Research Team wants your help to implement quick sort tailored to their tooth brushes.

Let us be more specific. Let M be an array of n data items in the hard disk. We want to sort M with respect to some field in the data items. We cannot load the entire array (or only the desired fields) in the RAM. So we will keep M in the hard disk itself. We assume that the RAM has sufficient space to store four *index* arrays of size s (in addition to the recursion stack and a constant number of local variables). You use these index arrays to build a min-max priority queue for the partitioning step needed for quick sort. Suppose finally that it is allowed to make direct copies $B[j] = A[i]$ from one hard-disk array A to another hard-disk array B without routing the item through the RAM.

The AlgoTooth computing system could have been presented to you in the form of a black box. For simplicity, this is not done. You do not even need to work with hard-disk arrays (files). Assume instead that M is just an array of integers. You will use some other integer arrays (like L and R to be explained later) pretending that these are hard-disk files.

An Implementation of Min-Max Priority Queues

A min-max priority queue supports retrieval, insertion, and deletion of both the minimum and the maximum elements. There are numerous ways of implementing such a queue. Here, we follow an indexing strategy suited to the AlgoTooth application. Let A be an array of s integers. We build both a min-heap and a max-heap from A. We can permute the elements of A so that it becomes a min-heap. However, this permutation will not correspond to a max-heap structure in general. We instead keep A as it is, and use two index arrays Q_1 and Q_2 to store the min-heap and the max-heap permutations. The quick-sort variant detailed below requires locating *arbitrary* elements of A in the two heaps. So we store the inverse permutations of Q_1 and Q_2 in two other index arrays I_1 and I_2. We have $Q_1[i] = j \iff I_1[j] = i$, and $Q_2[i] = j \iff I_2[j] = i$.

$Q_1[i]$	The index in $A[\,]$ of the i-th element in the min-heap
$Q_2[i]$	The index in $A[\,]$ of the i-th element in the max-heap
$I_1[i]$	The index in the min-heap, which contains $A[i]$
$I_2[i]$	The index in the max-heap, which contains $A[i]$

These arrays are illustrated below for a queue of $s = 10$ elements.

i	0	1	2	3	4	5	6	7	8	9
$A[i]$	588	609	596	620	525	555	537	618	622	564
$Q_1[i]$	4	6	9	1	5	2	8	7	3	0
$A[Q_1[i]]$	525	537	564	609	555	596	622	618	620	588
$I_1[i]$	9	3	5	8	0	4	1	7	6	2
$Q_2[i]$	8	7	3	1	5	2	6	9	0	4
$A[Q_2[i]]$	622	618	620	609	555	596	537	564	588	525
$I_2[i]$	8	3	5	2	9	4	6	1	0	7

As an example, the minimum 525 in the queue is stored at the index $Q_1[0] = 4$ in $A[\,]$. If we want to locate the position of this element in the max-heap, we read $I_2[4] = 9$. Likewise, the maximum 622 in the queue is at the index $Q_2[0] = 8$ in $A[\,]$, and this element is in the position $I_1[8] = 6$ in the min-heap.

Write the functions *getmin*, *minheapify* and *makeminheap* for the min-heap (which perform the standard min-heap-related tasks). These functions use the arrays Q_1 and I_1 only. Likewise, write the functions *getmax*, *maxheapify* and *makemaxheap* for the max-heap. Now, you use only Q_2 and I_2.

The application you are dealing with requires the minimum to be replaced by a larger value and the maximum to be replaced by a smaller value. Write two functions *minreplace* and *maxreplace* for this purpose. These functions need access to all the four arrays Q_1, Q_2, I_1, I_2. To see why, let the minimum m be replaced by a larger value v. You can locate m at index $i = Q_1[0]$ in A. Setting $A[i] = v$ simulates a hard-disk copy. Since $v > m$, there is a need to call *minheapify* at index 0. Moreover, this replacement has the potential to change the max-heap structure too. You locate the (previous) minimum m at index $j = I_2[i]$ in the max-heap. Since the value at this node has increased, there may be a necessity to move this value up the max-heap starting from index j (this process is akin to *maxinsert*).

You may also write a function *maxdelete* that would be needed during heap sorting of $A[\,]$. However, since this is a loop of data-copy-and-*maxheapify*, you can do it inline also. In this assignment, we do not use *mininsert*, *maxinsert*, and *mindelete*, so you do not have to implement these functions.

Memory-Constrained Quick Sort

Let M be an array of n integers, which we want to sort. The following variant of quick sort uses four arrays A, L, R, H. These five arrays simulate hard-disk files. A and H are of sizes s (the size of the queue), whereas L and R are of sizes $\leqslant n - s$. In the following description, we assume $n \geqslant s$. Handling the cases $n < s$ to terminate the recursion is left to you. The min-max queue Q consisting of four index arrays Q_1, Q_2, I_1, I_2 simulates what happens inside the RAM of AlgoTooth tooth brushes. It is easy to verify that the partitioning procedure runs in $O(n)$ time so long as $s = O(n/\log n)$.

1. Copy the first s elements of M to A, and initialize Q_1, Q_2, I_1, I_2. Also, initialize L, R to empty.
2. Build a min-heap and a max-heap of indices on A (update Q_1, Q_2, I_1, I_2).
3. For each of the remaining elements m_i in M, repeat:
   (a) If $m_i < \min(Q)$, append m_i to L,
   (b) else if $m_i > \max(Q)$, append m_i to R,
   (c) else we have $\min(Q) < m_i < \max(Q)$, so we want to insert m_i in Q. But the queue is already full. There are now two possibilities, namely, replace either $\min(Q)$ or $\max(Q)$ by m_i. The element to be replaced should be copied to L or R (as appropriate) before being overwritten by m_i. Both the min-heap and the max-heap structures should be restored after the replacement. During multiple invocations of this case, alternately delete $\min(Q)$ and $\max(Q)$.
4. Sort A to the array H. Any sorting algorithm can be used for that. However, since you already have a max-heap on A, implement heap sort.
5. Copy L, H, R (in that order) back to M.
6. Erase L, R and the queue Q. These are not needed after partitioning is done. Freeing the memory associated with Q is very much required, since the AlgoTooth RAM has only limited space.
7. Recursively call quick sort on the copies of L and R in M.

The *main*() function

- Read n and s from the user.
- Read n integers, and store them in the array M.
- Run the memory-constrained quick-sort algorithm on M.
- Print the sorted array M.

Submit a single C/C++ source file. Do not use global/static variables.

Sample output

```
      100 10
      738 362 783 984 269 792 616 838 276 836 681 671 823 787 128 390 648 983 480
      321 480  11 829 257 160  44  35 941 999 428 704 738 790 840  74 411 632 690
      602 261 878 283 932 701  70 413 444 718 396 276  40 876 287 221 485 800 265
      872 741 265 300 797 355 442 637 429 206 622 472 808 883 350  91 815 404 161
      580 200 231 976 476 623 852 764 845 689 564 110 562 657 375 214 454  82   9
      444 512 215  66 984

+++ Array after sorting
        9  11  35  40  44  66  70  74  82  91 110 128 160 161 200 206 214 215 221
      231 257 261 265 265 269 276 276 283 287 300 321 350 355 362 375 390 396 404
      411 413 428 429 442 444 444 454 472 476 480 480 485 512 562 564 580 602 616
      622 623 632 637 648 657 671 681 689 690 701 704 718 738 738 741 764 783 787
      790 792 797 800 808 815 823 829 836 838 840 845 852 872 876 878 883 932 941
      976 983 984 984 999
```