CS29003 ALGORITHMS LABORATORY
Assignment No: 3
Last Date of Submission: 14-Aug-2014

You manage a site *www.opionics.com* where people post photos and rate the posted photos. You need to maintain a good data structure for handling the photo ratings. Individual photos are given unique IDs in the sequence 0, 1, 2, … as they are posted to your site. Users can upvote (like) or downvote (dislike) a photo. The rating of a photo is an integer equal to the number of upvotes minus the number of downvotes that the photo receives. If the downvotes outnumber the upvotes for a photo, its rating is treated as zero, that is, the rating is not allowed to be negative. You need to access, alter the rating of, insert, and delete photos. A service like this is expected to deliver frequent top-photo suggestions (like the photo of the day).

Since your server is very loaded with billions of requests every second, you prefer to go for a very efficient data structure, namely, a priority queue (a max-heap) H in which insert, delete, upvoting and downvoting must be done with reasonable effort. You go for a contiguous (array) representation of H in which every cell stores two items: the ID of a photo, and its rating. The heap ordering is with respect to the ratings of the photos.

Read an initial size of the heap, say, *ninit*. Populate the first *ninit* entries of H with the pairs $(i,rinit_i)$, where i is the ID of the ith element and $rinit_i$ is the initial rating of the ith photo—an integer chosen randomly in the range 0–10. At this stage, the array is not necessarily a heap. Write a linear-time function *makeHeap*() to convert this array to a max-heap with respect to the rating vales.

Next, you enter a loop. In each iteration of the loop, you perform one of the following operations based on user inputs:

1. *Insert*: Add a new photo to the queue with an initial random rating in the range 0–10.
2. *Upvote*: Increment the rating (by one) of any randomly posted photo.
3. *Downvote*: Decrement the rating (by one) of any randomly posted photo (unless its current rating is zero).
4. *Delete*: Delete any randomly chosen photo.
5. *Quit*: Break the loop and exit.

Operations 2–4 in the loop call for accessing elements by their IDs, whereas the heap ordering is with respect to the rating values which have nothing to do with the IDs. In order to relieve your server from the burden of making linear searches in H for locating an ID, you maintain an index array *IDX*. The ith entry in *IDX* stores the index in H where the record (i,r_i) for the ith photo resides. Locating a specific ID i in H is then a constant-time effort. After you initially convert the *ninit* entries in H to a max-heap, write a function *initIndex*() that makes a pass through H and populates the index array *IDX* appropriately.

During an up- or down-voting inside the loop, you use *IDX* to locate a randomly chosen element (i,r_i) in H. You increment/decrement r_i as needed (do not make any change during downvoting if $r_i = 0$). A change in r_i may violate heap ordering in H. Handle this appropriately to restore heap ordering. Both up- and down-voting functions should run in O(log n) time, where n is the current size of the heap.

Efficient insertion and deletion require some care. When you delete the ith photo, set $IDX[i]$ to an invalid index like –1. Moreover, copy $H[n-1]$ to $H[i]$, and restore heap ordering. No photo inserted after this deletion will get the ID i. So you need to maintain two counts: the current size n of H, and the total number N of photos that you ever dealt with. An insert event assigns the ID N to the new photo (the earlier photos, present or deleted, have IDs 0, 1, 2, …, $N-1$), and adds this ID with an initial rating to H as in a priority queue. Notice that an insert event could reassign a deleted ID to the new photo, but that adds to the bookkeeping burden of your busy server, which you cannot afford. Moreover, your service may be thought to support an undelete operation (don't implement this in your program). There is a limit *NMAX* on the maximum number of photos your program can handle. An attempt to insert the (*NMAX* + 1)-st element should fail even if $n < NMAX$. Both insert and delete must run in O(log n) time.

Operations 1–4 mentioned above may perform swapping of elements of H. You must also swap the corresponding elements in *IDX* to reflect the change of positions in H.

Write a *main*() function like the following:

> Read *ninit* from the user.
> Populate *H* with *ninit* random entries $(i,rinit_i)$. Print *H*.
> Convert *H* to a max-heap with respect to the rating values. Print *H*.
> Prepare the index array *IDX* by making a pass through *H*. Print *IDX*.
> Run the Insert/Upvote/Downvote/Delete loop (until broken). Print *H* and *IDX* after every iteration.

Write the functions in the following sequence: *makeHeap*(), *initIndex*(), *insert*(), *upVote*(), *downVote*(), *delete*().

Sample Output

The following run starts with *ninit* = 10. Initially *H* is not a heap. After that, *H* is always printed as a max-heap, and *IDX* entries store indices in *H*. The heap is printed as a sequence of (i,r_i) values. The entry $i(j)$ in the printing of *IDX* indicates that the photo with ID i can be located at $H[j]$. If the photo with ID i is deleted, we have $IDX[i] = -1$. Here, upvote(2) means upvote the photo with ID 2, downvote(3) means downvote the photo with ID 3, delete(1) means delete the photo with ID 1, insert(6) means insert a new photo with initial rating 6, and so on. All indexing is zero-based.

```
Current heap (10 nodes)
        (0,3) (1,9) (2,1) (3,4) (4,6) (5,3) (6,5) (7,6) (8,6) (9,3)
--------------------------------------------------------------------------------
Current heap (10 nodes)
        (1,9) (7,6) (6,5) (8,6) (4,6) (5,3) (2,1) (3,4) (0,3) (9,3)
Current index array (10 cells used)
        0(8) 1(0) 2(6) 3(7) 4(4) 5(5) 6(2) 7(1) 8(3) 9(9)
--------------------------------------------------------------------------------
Operation: upVote(2)
Current heap (10 nodes)
        (1,9) (7,6) (6,5) (8,6) (4,6) (5,3) (2,2) (3,4) (0,3) (9,3)
Current index array (10 cells used)
        0(8) 1(0) 2(6) 3(7) 4(4) 5(5) 6(2) 7(1) 8(3) 9(9)
--------------------------------------------------------------------------------
Operation: downVote(3)
Current heap (10 nodes)
        (1,9) (7,6) (6,5) (8,6) (4,6) (5,3) (2,2) (3,3) (0,3) (9,3)
Current index array (10 cells used)
        0(8) 1(0) 2(6) 3(7) 4(4) 5(5) 6(2) 7(1) 8(3) 9(9)
--------------------------------------------------------------------------------
Operation: delete(1)
Current heap (9 nodes)
        (7,6) (8,6) (6,5) (9,3) (4,6) (5,3) (2,2) (3,3) (0,3)
Current index array (10 cells used)
        0(8) 1(-1) 2(6) 3(7) 4(4) 5(5) 6(2) 7(0) 8(1) 9(3)
--------------------------------------------------------------------------------
Operation: insert(6)
Current heap (10 nodes)
        (7,6) (8,6) (6,5) (9,3) (4,6) (5,3) (2,2) (3,3) (0,3) (10,6)
Current index array (11 cells used)
        0(8) 1(-1) 2(6) 3(7) 4(4) 5(5) 6(2) 7(0) 8(1) 9(3)
        10(9)
--------------------------------------------------------------------------------
Operation: upVote(8)
Current heap (10 nodes)
        (8,7) (7,6) (6,5) (9,3) (4,6) (5,3) (2,2) (3,3) (0,3) (10,6)
Current index array (11 cells used)
        0(8) 1(-1) 2(6) 3(7) 4(4) 5(5) 6(2) 7(1) 8(0) 9(3)
        10(9)
--------------------------------------------------------------------------------
```