Indian Institute of Information Technology, Nagpur
Department of Computer Science & Engineering
4th Semester (**Batch – C2: Section C**)
Subject: Design and Analysis of Algorithms

Date of submission / evaluation: 24th to 28th April, 2023 in regular lab time.

Question 1:

In this assignment, you work with binary trees. We assume that each node in the tree contains only a positive integer value and two child pointers (left and right). No parent pointers or additional values can be stored in the nodes.

Part I

You are given an array *A* with *n* positive integer values. Your task is to convert the array to a binary tree *T* storing the values in *A*. The first element of *A* (that is, A[0]) is stored at the root. Now, we make a random choice. With probability 1/3, the left subtree of *T* is empty, so the right subtree contains all of the remaining $n - 1$ values. With probability 1/3, the right subtree is empty, that is, the left subtree contains the remaining $n - 1$ values. Finally, with probability 1/3, both the subtrees are non-empty. In this case, each subtree consists of a half of the remaining $n - 1$ values: the left subtree the first half, and the right subtree the second half. Recursively, construct the left and/or right subtrees, whichever is/are not empty. You may read the array *A* from a file, or generate it randomly.

Which traversal of the created tree *T* produces the same listing of the values as stored in the original array *A*? Implement that traversal in order to check the correctness of your construction of *T*.

Part II

Let *r* be the root of the tree, and *v* any node in the tree *T*. There is a unique path from *r* to *v* in *T*. The *weight* of *v* is defined as the sum of all the values stored on this unique *r*-to-*v* path. Your task is to locate the maximum of the weights of all the nodes in *T*. Since the values stored in the nodes are positive, a maximum-weight node must be a leaf node. Write a *linear-time* recursive function to compute the largest weight. Do not use any global or static variables. Pass a parameter to store the sum of weights of all nodes on the path from the root to the current node (or its parent). During recursive calls, increment this parameter appropriately. Alternatively, you may use the fact that the maximum weight of a tree is the value stored at the root plus the larger of the maximum weights of the two subtrees of the root.

Part III

Let *w* be the maximum weight returned by the function of Part II. Write a recursive function to locate a path from the root to a leaf such that the sum of the values encountered on the path is *w*. If you have parent pointers, then such a path can be located in linear time. Likewise, if you store some indicator (like which subtree has the largest-weight leaf), then also it is easy to locate the path in linear time. But in your case, you do not have these options. If the function of Part II reorganizes the tree links so as to let the left subtree at every node contain the largest-weight leaf under it, then the maximum-weight path is the leftmost path in the restructured tree. But this destroys the original tree. Do not do that too.

Nevertheless, you can manage to locate a maximum-weight path by an expected linear-time (but worst-case quadratic-time) algorithm. Write a function of this complexity.

Sample Output

```
+++ INPUT ARRAY
 941 999 153 295 461 734 175 738 235 405 966 151 815 391 504  12 537 688 985 390
 139 981 257 693 338  78 715 269 643 598 569 584 597 721 598 776 455 772 514 408
 177 199 278 991 590 781 722 127 188 426 235 327 408 491 738 464 288 173 732 930
 770  20 233  86 740 830 861 913 322  95  41 217 293 318 209 882  99 649 727 287
  75 680 332 482 890  70 664 178 242 396 826  12 134  59 815 592 608 677 225 929
 771 265 146 782 582  73 665 680 721 111 685 796 791  17 996 681 805 661 858  47
 775 403 777 628 461 592 220 787
+++ TREE TRAVERSAL LISTING
 941 999 153 295 461 734 175 738 235 405 966 151 815 391 504  12 537 688 985 390
 139 981 257 693 338  78 715 269 643 598 569 584 597 721 598 776 455 772 514 408
 177 199 278 991 590 781 722 127 188 426 235 327 408 491 738 464 288 173 732 930
 770  20 233  86 740 830 861 913 322  95  41 217 293 318 209 882  99 649 727 287
  75 680 332 482 890  70 664 178 242 396 826  12 134  59 815 592 608 677 225 929
 771 265 146 782 582  73 665 680 721 111 685 796 791  17 996 681 805 661 858  47
 775 403 777 628 461 592 220 787
+++ MAXIMUM WEIGHT = 10280
+++ VALUES ON THE MAX-WEIGHT PATH
941 + 999 + 830 + 861 + 677 + 225 + 929 + 771 + 265 + 146 + 782 + 582 + 685 + 796 + 791
```

Question 2:

In this assignment, you work with a type of balanced binary search trees. The balancing condition being very demanding (compared to what happens in height-balanced trees), the insert and delete functions can be inefficient. However, the height of the tree will always have the least possible value.

A node in the tree contains an integer value v, a counter c, and three pointers L, R and P. The value v is the key stored at the node. The count c is the size (number of nodes) of the subtree rooted at the node (including the node itself). Finally, the pointers point respectively to the left child, the right child, and the parent. For the root node, the parent pointer is assumed to be NULL.

Part I

Write a function *tryinsert*() in order to make an insertion attempt of a key x in a tree T. Use the standard BST insertion procedure for this insertion attempt. The function returns a pointer to the root of the modified tree T after the insertion. If x was already present in T, the returned tree is the same as the tree before insertion. The function also returns (via a proper argument) a pointer q to the parent of the inserted node. If x was already present in T, then no node is inserted, and q receives the value NULL. A new key is always inserted at a new leaf node, so its initial count is always 1.

After a successful insertion, we need to update the counts of subtrees. Only the nodes on the insertion path are affected. If q is not NULL (that is, a real insertion took place), then traverse from q to the root via the parent pointers, and increment the count at every node on the path by one. Write a function *incrcount*() which takes the pointer q as the only argument, and performs this count updating. If q is NULL, no insertion took place (or the root is inserted to a NULL tree), so no node counts are affected, and this function need not be called.

Part II

Write a function *trydelete*() in order to make a deletion attempt of a key x in a tree T. If x is not present in T, then a pointer to the root of the original tree T is returned, and a pointer argument q receives the value NULL. If x is present, then the standard BST deletion procedure is invoked, a pointer to the root of the new tree is returned, and q is assigned a pointer to the parent of the deleted node.

The purpose of q is to readjust (decrement by one) the count at every node on the path from q to the root. Write a function *decrcount*() that implements this readjustment. If q is NULL, no deletion took place (or the root is deleted), so no node counts are affected, and this function need not be called.

Part III

We now introduce the balancing condition. We call a BST T a *perfectly balanced binary search tree* (or *PB-BST*) if the absolute difference of the counts of the nodes of the two subtrees at every node is never more than 1. If we define the imbalance at a node v as count(right(v)) – count(left(v)), then the only allowed values of imbalance(v) are 0, +1, and –1.

A successful insertion/deletion may throw a PB-BST T out of balance. Since T was perfectly balanced before the insertion/deletion, some nodes can now have an imbalance of +2 or –2. All such nodes lie on the insertion/deletion path (from the pointer q returned in Part I or II to the root). This is detected after the count adjustments are made. Write a function *rebalance*() that repairs all imbalance cases, and restores perfect balance back to T.

The re-balancing algorithm is explained now. Suppose that at some node v, the imbalance is +2. This means that for some $k \geq 0$, the left subtree of v contains k nodes, and the right subtree $k + 2$ nodes. We insert the value at v in the left subtree of v. Next, we locate in the right subtree of v the immediate successor of the value stored at v. Since the size of the right subtree is $k + 2$, it is in particular non-empty, and the immediate successor can always be found. Copy this immediate successor value to the node v. Finally, delete this immediate successor value from the right subtree. After this re-balancing, both the subtrees of v contain exactly $k + 1$ nodes, and we proceed to the parent of v for possible re-balancing.

The case of imbalance(v) = –2 is symmetrically handled.

Append a call of this re-balancing function after the count-adjustment function during each insertion/deletion. Notice that each insertion or deletion in the re-balancing stage may additionally throw certain subtrees out of balance. These must be recursively handled.

Sample Output

You should first read the number of integers to insert, call it *nins*. For each of the *nins* integers read, call the functions *tryinsert*(), *incrcount*() and *rebalance*() (the last two if needed), and print the tree *before* you read and insert the next integer. The pre-order listing of the tree should be printed with each line specifying the following items at a node *v*:

value(*v*) (count(*v*)) → value(left(*v*)) (count(left(*v*))), value(right(*v*)) (count(right(*v*)))

For example, the line

```
53 (14) -> 21 (6), 82 (7)
```

means that a node contains the key 53 and count 14. Its left and right children store the keys 21 and 82 and have respective counts 6 and 7.

After the insertion of *nins* integers, read the number of integers to delete, call it *ndel*. For each of the *ndel* integers read, call the functions *trydelete*(), *decrcount*() and *rebalance*() (the last two if needed), and print the tree *before* you read and delete the next integer.

Here, we demonstrate insertion and deletion in a small PB-BST. Let, at some point of time, the tree contain eight nodes, and the preorder listing of these nodes is:

```
26 (8) -> 20 (3), 55 (4)
   20 (3) -> 4 (1), 24 (1)
      4 (1) -> -1 (0), -1 (0)
      24 (1) -> -1 (0), -1 (0)
   55 (4) -> 49 (1), 74 (2)
      49 (1) -> -1 (0), -1 (0)
      74 (2) -> 70 (1), -1 (0)
         70 (1) -> -1 (0), -1 (0)
```

Let us insert 37 in this tree. This makes one violation of the PB-BST property (at Node 26 where the imbalance becomes +2). So two recursive calls are made. First, 26 is inserted to the left subtree (rooted at Node 20). The immediate successor of 26 is 37. So 37 replaces 26 at the root, and 37 is deleted from the right subtree (rooted at Node 55).

The subtrees are printed after each call.

```
+++ Insertion of 26 under 20

20 (4) -> 4 (1), 24 (2)
   4 (1) -> -1 (0), -1 (0)
   24 (2) -> -1 (0), 26 (1)
      26 (1) -> -1 (0), -1 (0)

+++ Deletion of 37 under 55

55 (4) -> 49 (1), 74 (2)
   49 (1) -> -1 (0), -1 (0)
   74 (2) -> 70 (1), -1 (0)
      70 (1) -> -1 (0), -1 (0)
```

After these adjustments, the tree is perfectly balanced, and looks like the following:

```
37 (9) -> 20 (4), 55 (4)
   20 (4) -> 4 (1), 24 (2)
      4 (1) -> -1 (0), -1 (0)
      24 (2) -> -1 (0), 26 (1)
         26 (1) -> -1 (0), -1 (0)
   55 (4) -> 49 (1), 74 (2)
      49 (1) -> -1 (0), -1 (0)
      74 (2) -> 70 (1), -1 (0)
         70 (1) -> -1 (0), -1 (0)
```

Now, the BST deletion of 37 brings 49 to the root, and deletes the leaf node 49. At Node 55, the PB-BST property is violated. This requires the adjustments: insertion of 55 in the left subtree (here it is NULL), replacement of 55 by its immediate successor 70, and deletion of 70 from the right subtree (rooted at Node 74).

```
+++ Insertion of 55 under NULL

55 (1) -> -1 (0), -1 (0)

+++ Deletion of 70 under 74

74 (1) -> -1 (0), -1 (0)
```

One node up the tree, we encounter Node 37, where re-balancing is not necessary. Since Node 37 is the root, the deletion procedure stops, and the final balanced tree is:

```
49 (8) -> 20 (4), 70 (3)
   20 (4) -> 4 (1), 24 (2)
      4 (1) -> -1 (0), -1 (0)
      24 (2) -> -1 (0), 26 (1)
         26 (1) -> -1 (0), -1 (0)
   70 (3) -> 55 (1), 74 (1)
      55 (1) -> -1 (0), -1 (0)
      74 (1) -> -1 (0), -1 (0)
```

Look at sample output files linked from the laboratory page. The files demonstrate that recursion levels can be quite high (near the height of the tree).

Submit two separate files.

1. A program that implements Parts I and II alone. Here, no re-balancing attempts are made, so the imbalance at a node is allowed to be more than +2 or less than –2.

2. A program that additionally implements Part III. Every tree printed in this program should be a PB-BST. You may use the same Parts I and II as in your first file.

Question 3:

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 $0, 1, 2, \ldots, N-1$), 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)
-------------------------------------------------------------------------------
```