Programming Assignment 4: Augmenting Binary Search Trees

Due Monday Nov. 16 by 11:59PM

In this assignment you will modify the binary search tree code used in lab to support several new features (some with runtime requirements).

These features will require *augmentation* of the existing data structures with additional book-keeping information. This bookkeeping info must be kept up to date incrementally; as a result you will have to modify some existing functions (insert, delete, build-from-array).

Now to the new functions/features:

```
/* allocates an integer array, populates it with the
   elements of t (in-order) and returns the array as an
   int pointer */
extern int * bst_to_array(BST_PTR t);

/* returns the ith smallest element in t.  i ranges
   from 1..n where n is the number of elements in
   the tree.

   If i is outside this range, an error message is printed
   to stderr and the return value is arbitrary (you may return
   whatever you like, but it has no meaning.

   Runtime:  O(h) where h is the tree height
*/
extern int bst_get_ith(BST_PTR t, int i);

/* returns the value in the tree closest to x -- in other
   words, some y in the three where |x-y| is minimum.

   If the tree is empty, a message is sent to stderr and
   the return value is of your choosing.

   Runtime:  O(h) where h is the tree height.
*/
extern int bst_get_nearest(BST_PTR t, int x);
```

```
/* returns the number of elements in t which are greater
   than or equal to x.

   Runtime:  O(h) where h is the tree height

*/
extern int bst_num_geq(BST_PTR t, int x);

/* returns the number of elements in t which are less
   than or equal to x.

   Runtime:  O(h) where h is the tree height

*/
extern int bst_num_leq(BST_PTR t, int x);
```

Additional runtime requirement:

Modify the size function to have O(1) runtime.

Functions needing modification:

The insert and remove functions modify the tree. You will need to change them so that they also make sure that the bookkeeping information is correct.

The runtime of these functions must still be O(h)

Submission: you will submit bst.c and a README file (see below) in a *single* archive file.

Comments/Suggestions:

You will need to augment the NODE struct. This is perfectly fine since it is in the .c file and not the .h file.

I recommend you write a sanity-checker function which, by brute force, tests whether the bookkeeping information you've maintained is indeed correct.

Of course, you should write extensive test cases. You are free to share test cases with classmates. You might try valgrind to also

look for memory errors.

Extra Credit 1:

You can receive up to 10% extra credit by modifying your implementation so that the bst_min and bst_max functions to operate in O(1) time.

Note that you will almost certainly need to modify more than just those functions to make this work.

Extra-Credit 2: Size-Balancing

You can receive up to an additional 30% if you correctly implement the size-balanced strategy described below (and include a proof as specified below).

Your tree will maintain logarithmic height by enforcing a *"size-balanced"* property.

Definition: size-balance property for a node. Consider a node *v* in a binary tree with n_l nodes in its left subtree and n_r nodes in its right subtree; we say that *v* is *size-balanced* if and only if:

$$max(n_l, n_r) \leq 2 \times min(n_l, n_r) + 1$$

(so roughly, an imbalance of up to ⅓ - ⅔ is allowed)

Definition: size-balance property for a tree. We say that a binary tree *t* **is size-balanced** if and only if all nodes *v* in *t* are size-balanced

Your implementation must ensure that the tree is *always size-balanced*. Only the insert and remove operations can result in a violation. When an operation (and insert, update or delete) results in a violation, you must rebalance **the violating subtree closest to the root.** You **do not** in general want to rebalance at the root each time there is a violation (only when there is a violation at the root).

As it turns out, while every now and then we may have to do an expensive rebalancing operation, a sequence of m operations will still take O(mlog n) time -- or O(log n) on average for each of the m operations. Thus, it gives us performance as good as AVL trees (and similar data structures) in an amortized sense.

Your data structure will keep track of some statistics to help us corroborate this claim.

You should have a subroutine for building a perfectly balanced tree from a sorted array to aid in this process and make it as simple and fast as possible.

You are also required to submit a proof that the height of any size-balanced tree is $O(\log n)$.

Readme File:

> To make grading more straightforward (and to make you explain how you achieved the assignment goals), you must also submit a Readme file.
>
> The directory containing the source files and this handout also contains a template Readme file which you should complete (it is organized as a sequence of questions for you to answer).