Programming Assignment 3:

Augmenting and Balancing Binary Search Trees

Due Saturday Nov. 4 11:59PM

In this assignment you will modify the binary search tree code studied in class to:

1. Support several new features (some with runtime requirements) and
2. Enforce a balancing property ("size-balancing") which results in amortized logarithmic runtime for insertion and deletion (and all operations that take time proportional to the tree height are $O(\log n)$ in the worst case.

NOTE: Although described as two parts, they may be implemented in either order: part 2 does not really depend on part-1 (although they will both probably rely on the bookkeeping information you).

(1) Additional Features

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).

Bookkeeping Info Hint: keeping track of the number of nodes in each subtree might come in handy!

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 *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 *t, int i);

/* returns the value in the tree closest to x -- in other
   words, some y in the tree 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 *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 *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 *t, int x);

/* returns the number of elements in t which are between min
       and max (inclusive).

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

*/
extern int bst_num_range(BST *t, int min, int max);
```

Pre-existing functions needing modification:

Three pre-existing functions either modify an existing tree or build one from scratch You will need to change them so that they also make sure that the bookkeeping information is correct. The relevant functions are:

```
int bst_remove(BST * t, int x);
int bst_insert(BST * t, int x);
BST * bst_from_sorted_arr(int *a, int n);
```

The runtime of these bst_remove and bst_insert must still be O(h); the runtime of bst_from_sorted_arr must still be O(n).

Comment: once you have completed part-2 (size-balancing), these runtime bounds will become $O \log n)$ because in a size-balanced tree, the height is guaranteed to be $O(\log n)$

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.

HINT: some of the logic employed in the previously studied QuickSelect algorithm may be handy (not the entire algorithm per-se, but its underlying logic

(2) "Size-Balancing"

In this part of the assignment you will implement the size-balanced strategy described below.

As we know, "vanilla" BSTs do not in general guarantee logarithmic height and as a result, basic operations like lookup, insertion and deletion are linear time in the worst case. There are ways to fix this weakness -- e.g., AVL trees and Red-Black trees. We will not be doing either of those; instead, you will implement the "size-balanced" strategy described below.

The *"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 delete operations can result in a violation.

- When an operation which modifies the tree (an insert or delete) results in a violation, you must rebalance **the violating node/subtree <u>closest to the root</u>**
- 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).

Amortized Claim: 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.

Strategy/Suggestions:

A straightforward approach to rebalancing a subtree is as follows:

- Populate a temporary array with the elements/nodes in the subtree in sorted order.
- From this array, re-construct a perfectly balanced (as perfectly as possible) tree to replace the original tree.
- The details are up to you, but observe that the number of tree nodes before and after the re-balancing is unchanged, you should be able to re-use the already existing nodes.

Statistics function:

You will also implement a function which reports the total "work" done by re-balancing. Every time you do a re-balance, the cost or work performed is the size of the sub-tree that was rebalanced.

You will keep track of the total amount of rebalancing work performed and report it via the following function:

```
/**
* function:  bst_sb_work
* description:  returns the total amount of work performed by re-balancing
*              since the creation of the tree.
*                Every time a rebalance  operation happens, the work
*                is equal to the size of the subtree that was rebalanced.
*              The total work is simply taken over all re-balancing ops.
*/
extern int bst_sb_work(BST *t);
```

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).

Checklist/Points

TASK	**POINTS**	**DONE?**
extern int * bst_to_array(BST *t);	20	
extern int bst_get_ith(BST *t, int i);	20	
extern int bst_get_nearest(BST *t, int x);	20	
extern int bst_num_geq(BST *t, int x);	15	
extern int bst_num_leq(BST *t, int x);	15	
extern int bst_num_range(BST *t, int min, int max);	15	
Implementation of Size-Balancing Strategy	60	
extern int bst_sb_work(BST *t);	15	
Readme File: **You have been given a template Readme file; answer the questions in the file to the best of your ability.**	15	
(total points)	195	

Submission: you will submit bst.c and the Readme file in a *single* archive file.