## Programming Assignment 3:

## Augmenting and Balancing Binary Search Trees

Due MONDAY April 16 6 @ 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 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 bookkeeping information. This bookkeeping info must be kept up to date incrementally; as a result you will have to modify some existing functions (insert, delete, from\_vector).

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

Now to the new functions/features:

|  |
| --- |
| /\* **Function: to\_vector**  **Description**: creates a vector and populates it with the  elements of the tree (in-order) and returns the vector  as a pointer    Runtime: O(n) where n is the number of elements in the tree.  \*/  **std::vector<T> \* to\_vector()**  /\* **Function: get\_ith**  **Description**: determines the ith smallest element in t and  "passes it back" to the caller via the reference parameter x.  i ranges from 1..n where n is the number of elements in the  tree.  Return value: If i is outside this range, false is returned.  Otherwise, true is returned (indicating "success").  Runtime: O(h) where h is the tree height  \*/  **bool get\_ith(int i, T &x)**  /\* Function: num\_geq  Description: returns the number of elements in tree which are  greater than or equal to x.  Runtime: O(h) where h is the tree height  \*/  **int num\_geq(const T & x)**  /\* Function: num\_leq  Description: returns the number of elements in tree which are less  than or equal to x.  Runtime: O(h) where h is the tree height  \*/  **int num\_leq(BST \*t, int x)**  /\* Function: num\_range  Description: returns the number of elements in tree which are  between min and max (inclusive).  Runtime: O(h) where h is the tree height  \*/  **int num\_range(const T & min, const T & 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:

bool remove(T & x)

bool insert(T & x)

static bst \* from\_sorted\_vec(const std::vector<T> &a, int n)

The runtime of these remove and 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), the runtime bounds for insert and remove will become because in a size-balanced tree, the height is guaranteed to be .

**Comments/Suggestions:**

**AUGMENTATION**: You will need to *augment* the bst\_node struct. What should it keep track of in addition to left/right subtrees and the value stored at the node? Once again: *How about keeping track of the number of nodes in the subtree rooted at the node?*

**SLOW VERSIONS OF VARIOUS FUNCTIONS:** You will notice that there are a pre-written "slow" versions of several of the functions that you are implementing. For example, get\_ith\_SLOW performs the same task as get\_ith (one of your TODOs) BUT does not meet the runtime requirements.

You may use these SLOW versions to help test your solutions.

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

**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 i*n a binary tree with nodes in its left subtree and nodes in its right subtree; we say that *v*is *size-balanced* if and only if:

+ 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 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).

**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.

**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?** |
| **vector <T> \* to\_vector();** | 20 |  |
| **bool get\_ith(int i, T &x);** | 20 |  |
| **int num\_geq(const T & x);** | 10 |  |
| **int num\_leq(const T & x);** | 10 |  |
| **int num\_range(const T & min, const T & max);** | 10 |  |
| **Correct Implementation of Size-Balancing Strategy** | 60 |  |
| **Readme File:**  **You have been given a template Readme file; answer the questions in the file to the best of your ability.** | 20 |  |
| **(total points)** | 150 |  |

**DELIVERABLES:**

Your only real deliverables are bst.h and Readme.txt

**COMPILATION:**

Any program utilizing (including) bst.h MUST compile using:

**g++ -std=c++11**

Any submission that fails to compile under this rule may simply be assigned a score of zero.