Programming Assignment 3:

Augmenting and Balancing Binary Search Trees

Due Thursday, July 29 @ 11:59PM
No late submissions

In this assignment you will modify the binary search tree code studied in class and lab (i.e., the source file **bst**.h you will find in the src directory). Your modified and improved version will:

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 will almost certainly devise).

(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: position_of
     * Description:  this is like the inverse of
     *       get_ith:  given a value x, determine the
     *       position ("i") where x would appear in a
     *       sorted list of the elements and return
     *       the position as an integer (in {1..n}).
     *       If x is not in the tree, -1 is returned.
     *       Examples:
     *           if x happens to be the minimum, 1 is returned
     *           if x happens to be the maximum, n is returned where
     *               n is the tree size.
     *       Notice the following property:  for a bst t with n nodes,
     *           pick an integer pos:1 <= pos <= n.
     *           Now consider the following code segment:
     *
                 T x;
                 int pos, pos2;
                 // set pos to a value in {1..n}
                 t.get_ith(pos, x);  // must return true since pos is in {1..n}

                 // now let's find the position of x (just retrieved)
                 pos2 = t.position_of(x);
                 if(pos != pos2) {
                     std::cout << "THERE MUST BE A BUG!\n";
                 }

                 See how position_of performs the inverse operation of get_ith?
     *
     * Return:  -1 if x is not in the tree; otherwise, returns the position where
x
     *          would appear in the sorted sequence of the elements of the tree
(a
     *          value in {1..n}
     *
     * Runtime:  O(h) where h is the tree height
     */
     int position_of(const 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(const T & 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)

   /*
    * function:     extract_range
    * Description:  allocates a vector of element type T
    *               and populates it with the tree elements
    *               between min and max (inclusive) in order.
    *               A pointer to the allocated and populated
    *               is returned.
    *
    * notes/comments:  even if the specified range is empty, a
    *                  vector is still allocated and returned;
    *                  that vector just happens to be empty.
    *                  (The function NEVER returns nullptr).
    *
    * runtime:  the runtime requirement is "output dependent".
    *           Let k be the number of elements in the specified range
    *           (and so the length of the resulting vector) and let h
    *           be the height of the tree.  The runtime must be:
    *
    *                  O(h + k)
    *
    *           So...while k can be as large as n, it can be as small
    *           as zero.
    *
    */
   std::vector<T> * extract_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 from_sorted_vec must still be O(n).

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

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: We 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 what we will call the "size-balanced: strategy described below.

> But first, a WARNING:
>
> You MUST implement the size-balanced strategy described below (or simply not implement any balancing strategy -- you can still get points for the functions in (1)).
>
> If, for example, you decide that you don't have to follow the instructions and "implement" AVL trees instead you will **automatically receive a ZERO for the ENTIRE assignment.**
>
> Why? Because there is too much AVL code readily available and one of the goals of this assignment is to have you work through a balancing policy without easy reference to pre-existing solutions.
>
> Now...back to size-balanced trees.

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.

> Big picture:
>
> size-balanced trees are not quite as "strong" as AVL or Red-Black trees in the sense that insert and delete will have *amortized* logarithmic runtime instead of (instead of guaranteed logarithmic runtime for *individual* inserts/deletes as with AVL and Red-Black trees). (The amortized property is formalized a bit below.)
>
> On the other hand, size-balanced trees are probably easier to implement than AVL or Red-Black trees. Furthermore, as far as I know, a simple google search will not find complete C++ implementations of size-balanced trees :)

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 $\frac{1}{3}$ - $\frac{2}{3}$ 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.

PLEASE, PLEASE, PLEASE: Before posting to Piazza *"what does size-balanced mean?"*, Re-read the above definitions several times. and create some examples that do and do not obey the property.

Maintaining the Size-Balanced Property:

Your implementation must ensure that the tree is *always size-balanced*. Only insert and remove operations can result in a violation.

The following rebalancing rules are super **IMPORTANT**:

- 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).

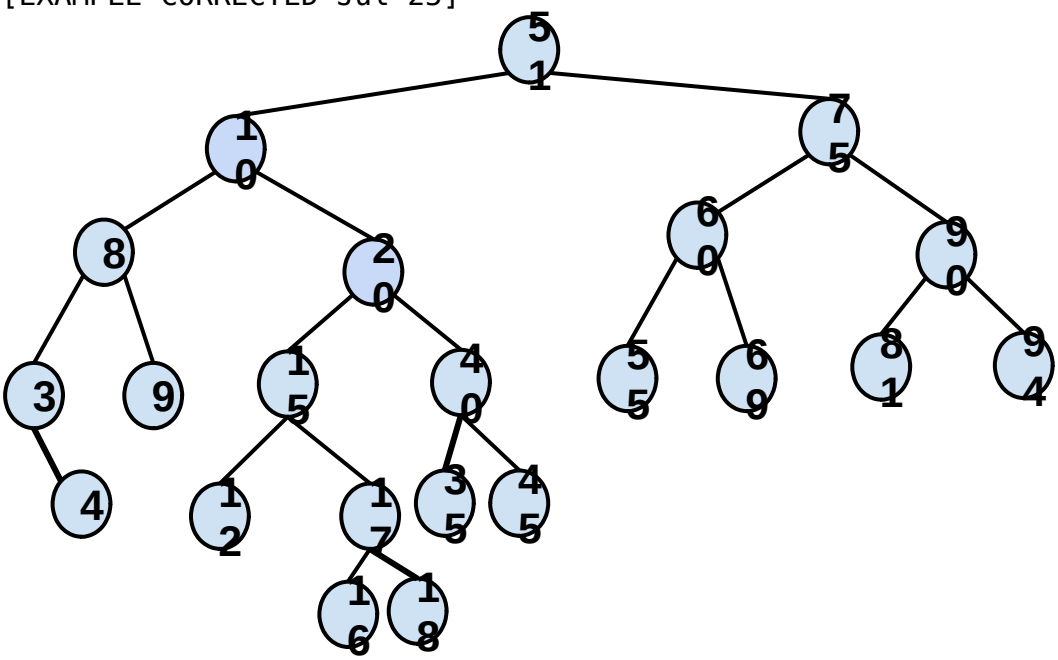
REMEMBER: when you rebalance a sub-tree, the result will be a subtree with exactly the same elements, but restructured to be **perfectly balanced**. The idea is like this:

> *"ok, this subtree is getting pretty far out of whack; let's just restructure the entire thing to be perfect, so we won't have to worry about it for a while"*

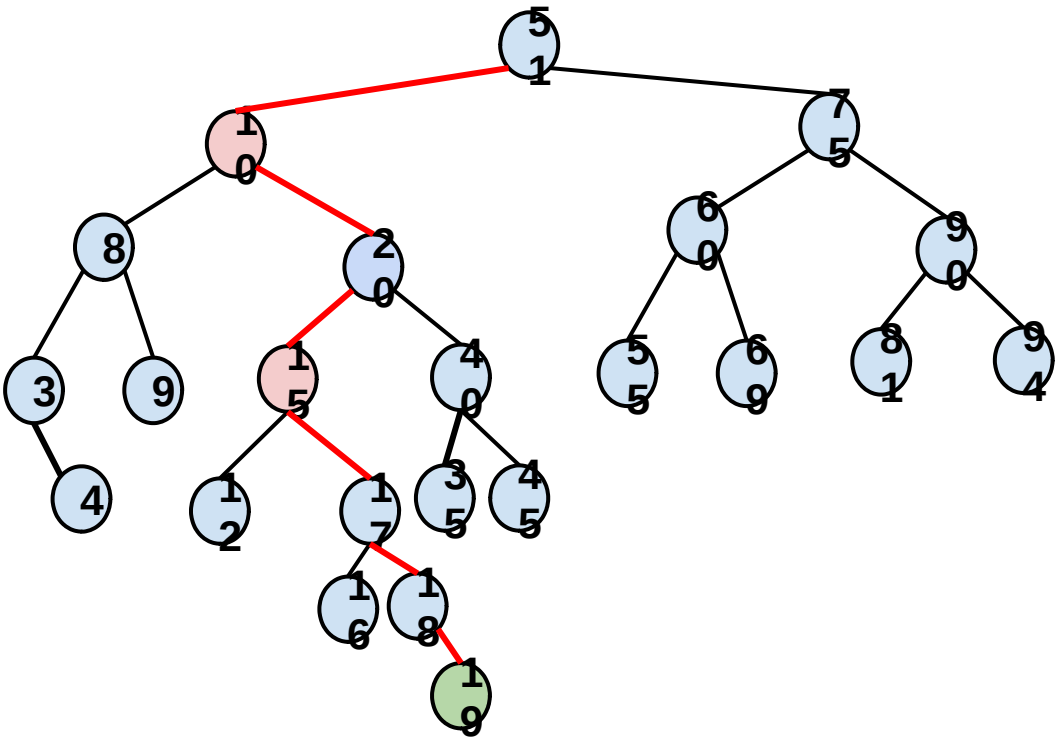
Example:

The BST below obeys the size-balanced property. Remember that *every* node/subtree (not just the "global" root) must obey the rule.

It is left as an exercise to verify this claim at each node. [EXAMPLE CORRECTED Jul 25]

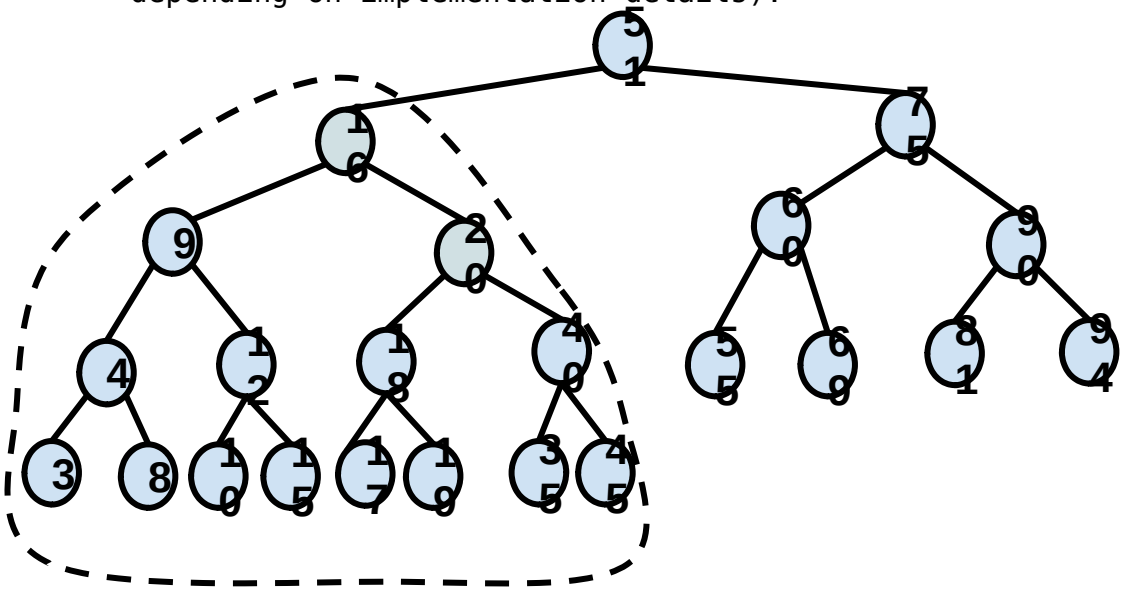


Notice that the subtree rooted at 10 just barely satisfies the rule. **Now suppose 19 is inserted.** We proceed as usual and (at least temporarily) end up with the configuration below.



The red lines indicate the insertion path. Some claims and observations for this "snapshot" configuration:

- There are now *two* violating subtrees (verification left as an exercise):
  - The subtree rooted at 15 and
  - The subtree rooted at 10.
- All other nodes/subtrees do *not* violate the property (verification left as an exercise).
- **Observation**: The violating nodes are ***on the insertion path*** traversed when inserting 19. In general, if a violation occurs, it must be on the insertion/deletion path (because the size of all other subtrees is unchanged).
- **Observation**: the node containing 10 is the violating node closest to the global root (15 is a descendant of 10).
- Now remember the rule: *perfectly rebalance the violating node closest to the root.* The resulting configuration will look like this (or an equally balanced configuration depending on implementation details):



- The circled subtree is now as balanced as possible. Notice that at each node, the left and right subtrees differ by at most one in size.

Some more observations and tips:

- To restructure a subtree with k nodes is achievable in $\Theta(k)$ time. Right?
- Since violating nodes must be on the insertion (or deletion) path, you should be able to detect a violation during the conventional insertion/deletion process -- assuming you have figured out an appropriate "augmentation".
- (**Do NOT do this)** Here is a naive idea that is kind of pointless:
  - perform insertion or deletion as usual
  - then walk the *entire tree* looking for a violation.
  - If a violation is found rebalance the appropriate subtree.

- Here is **another bad idea**:
  - if an insertion or deletion results in a violation, just rebalance the *entire* tree (even if the global root is not a violating node).
- (it is left as an exercise to see why these strategies are a disaster). If you find yourself wanting to "walk" the entire tree, think again!
- Bottom Line: an insertion or deletion should take:
  - case 1 (no violation results): $O(\log n)$
  - case 2 (violation results): $O(\log n+k)$ where k is the size (number of nodes) of the subtree that is rebalanced. **Remember**: the subtree to rebalance is rooted at the violating node closest to the global root.

Amortized Claim: If we follow this strategy, it turns out that although 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 (or vector) 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 *subtree*.
- 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.

ASIDE (you can get away with rebalancing all violators):

Remember that upon a violation, you must rebalance the violating subtree *closest to the root* (e.g. the subtree rooted at 10 in the preceding example).

However, suppose you rebalanced *all* violating subtrees as you "work your way back" toward the root (eventually arriving at the violator closest to the root and rebalancing that subtree).

Seems like some wasted work right? However, it turns out that the overall asymptotic runtime will still be the same as if you only rebalanced the violator closest to the root. (Can you figure out why?).

As a result, an implementation uses this approach (rebalancing all violating subtrees as you work back to the root) still meets the requirements of the assignment.

You may find this a little easier to implement (you can see how the logic is simpler -- you just need to determine if the subtree violates and if it does, rebalance; on the other hand, if you only rebalance the violator closest to the root, you have to also be able to determine if a violating node has no violating ancestors before deciding to rebalance.)

Deliverables and Scoring

Readme File:

To make grading more straightforward (and to force you to explain how you achieved the assignment goals), you must also submit a Readme file.

Template for your readme file: The directory containing the source files (subdirectory src) and this handout also contains a template readme file (name: Readme.txt) which you should complete (it is organized as a sequence of questions for you to answer).

Checklist/Point

TASK	POINTS	DONE?
std::vector <T> * to_vector();	20	
bool get_ith(int i, T &x);	20	
int position_of(const T & x);	15	
int num_geq(const T & x);	10	
int num_leq(const T & x);	10	
int num_range(const T & min, const T & max);	10	
std::vector<T> * extract_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	
"Honest Effort" points (not as many as on previous assignments!)	25	
(total points)	200	

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.

ADDITIONAL RESOURCES:

TEST-SUITE: A subset of the test programs that will be used to score your submission will be released roughly 3-4 days prior to the due date. In the meantime, you should be developing your own testing strategies (it builds character!).

NOTES: you will also notice a directory called NOTES which contains:

A collection of slides covering the fundamental BST concepts needed for this assignment. The 2nd slide is sort of a table-of-contents with links to relevant groups of slides.

Handwritten notes on the size-balanced property:

- understanding when the size-balanced property is and is not satisfied.
- What should happen when an insert or remove operation results in a violation of the property.

warmup_lab: in this directory, you will find a "lab" in which you work through a sequence of exercises relating to the size-balanced property.