Pset 5

CPSC 223

Due Date: April 17, 2023

In this assignment, you will devise an implementation of an AVL Tree starting with an implementation of a naive binary search tree. A binary search tree has the invariant that all nodes to the left of a node are smaller in value, and all nodes to the right are larger in value. An AVL Tree is a "self-balancing" binary search tree.

Why is balancing a BST important? If a binary search tree is balanced (or close to being balanced), it has a logarithmic runtime for the following functions:

- `find_min()`
- `find_max()`
- `contains()`
- `insert(value)`
- `remove()`
- a few others

However, if the binary search tree is *not* balanced, the runtime for these operations could be linear in the number of nodes in the tree!

Makefiles

As with Pset 4, the makefile is given. Type `make` to compile the code you write. Run the program with `./avlt`.

The Driver

Similar to Pset 4, you have been given a driver for this program. You should not modify this driver. In addition to the driver, you are given an extra function that can be used to print out your trees (because it's kind of a pain to write such a thing from scratch)! Looking at the trees that print (after doing operations such as `insert`, `remove`, *etc.*) can be a great way to check your code.

The Assignment

For this assignment, we provide the AVLTree header (`AVLTree.h`) file and a structured outline of the AVLTree functions (`AVLTree.cpp`). You are responsible for writing many of the implementation functions, although we have provided you with some of the functions already (mainly the parts that are essentially identical between naive BSTs and AVL trees).

Remember, AVL trees are a subset of binary search trees. That means that your AVL Trees must maintain the binary search tree invariant that all values

in the left subtree of any node are less than the node and all values in the right subtree are greater than the node. We have thoroughly commented `AVLTree.h`, so please refer to that file for specific information.

You are not allowed to change the header or functionality of the public functions. What you *can* do is to add extra functions and attributes to your code to better modularize the program. Remember that these functions and attributes are to be added only under the `private` section of your code.

You are responsible for writing the following functions:

- `AVLTree()` (Constructor)
  - a simple function that initializes values properly for an empty tree
- `AVLTree(const AVLTree &source)` (Copy constructor)
  - creates a (deep) copy of `source`
- `~AVLTree()` (destructor)
  - Remove all nodes from this and free all occupied memory
- `AVLTree& operator=(const AVLTree& source)` (Assignment operator overload)
  - This function is invoked when you execute `someTree = anotherTree`. It has to work in the same way as integer assignment: original value of `someTree` should be deleted, and then you need to create a *copy* of `anotherTree` into `someTree`.
  - Here are the steps this function must take
    1. Check for self-assignment
    2. Only do something if not self assignment (so, if you do `thisTree = thisTree;` program should do nothing)
    3. Delete any memory used by `someTree`
    4. After `someTree` has been deleted you need to copy anotherTree into `someTree`
    5. Once you are finished you have to return `*this` (which effectively returns a pointer to `someTree`)
- `Node* find_min (Node* node)` (private function)
  - Recurse through tree until node with smallest value is found, and return that node
  - Remember the invariant of BSTs!
  - Assumes `node` is not `NULL` (its behavior is undefined if `node` is `NULL`)

    Note: there are two `find_min` functions (one with and one without parameters). One is already implemented, you have to implement the private one. This is a common pattern used throughout this pset.
- `Node* find_max (Node* node)` (private function)
  - This function is VERY similar to `find_min`
- `bool contains (Node* node, int value)` (private function)
  - Return `true` if tree rooted at `node` contains `value`, `false` otherwise

    Hint: you will have to recurse through the tree
  - Use the invariant property of BSTs to make your code as efficient as

possible

- `int tree_height(Node *node)` (private function)
  - Retun the height of the tree.
    Hint: for the most efficient solution, use the `height` field of the `Node` struct with which you are provided. But make sure to keep it up-to-date during all of the other operations!
    An alternative implementation could compute the height of a node on-the-fly every time it's queried. This is slower than using the stored height, but is just as correct.
- `int node_count (Node* node)` (private function)
  - Return the total number of nodes in the tree
    - An empty tree has 0 nodes
    - A tree with a single vertex has 1 node
    - and so on...
  - *Multiplicity is ignored for this function.* So a number that has been inserted 10 times still counts as a single node
- `int count_total (Node* node)` (private)
  - Similar function to `node_count`
  - Rather than simply counting the *number* of nodes, you have to sum all *values* that have been inserted into the tree
  - We care about multiplicity for this one: if we insert `3`, `4`, and `3` again in the tree, `count_total` should return `10`
- `Node* pre_order_copy(Node* node)`
  - this function receives a `Node*` (`node`) and must return a pointer to the root of a copy of the tree with root `node`.
    Note: the copy should be complete, so if `node` has children you should create children nodes to add to this, and so on.
  - This function is used in the copy constructor (as you see in the starter code) and may be used in other functions yet to be implemented
    Hint: as the function name suggests, you should use a pre-order traversal to copy details from old node to new node
- `void post_order_delete (Node* node)`
  - Very similar to (in fact, could be called by) the destructor, this function removes all nodes and frees all occupied memory.
    Hint: as the function name suggets, you should use a post-order traversal to remove all the nodes
- `Node* balance (Node* node)` (private function)
  - Called by the `insert` and `remove` functions
    - You do not need to—nor should you—modify these two functions
  - Make sure that you preserve the BST invariants
  - Ignore multiplicity for balancing
  - Use the AVL balancing algorithms discussed in class
- `Node* right_rotate(Node* node)` (private function)
  - Rotates the tree rooted at `node` to the right, returining a pointer to the root of the rotated tree
  - Make sure your rotation preserves the BST invariants and updates

the node heights!

- `Node* left_rotate(Node* node)` (private function)
  - Rotates the tree rooted at `node` to the left, returining a pointer to the root of the rotated tree
  - Make sure your rotation preserves the BST invariants and updates node heights!
- `int height_diff (Node* node)` (private function)
  - Returns the difference of the height of the left subtree and the height of the right subtree
  - Useful for detecting if the tree is balanced

Notes about the starter code

The starter code has several important functions implemented for you, which make up most of a rudimentary implementation of a naive binary search tree, similar to the one we discussed in class. As soon as you implement the constructors, destructor, and assignment overload, you'll have a working BST implementation! The rest of the assignment is adding functionality to this BST implementation and adding the self-balancing aspect of AVL trees to keep everything efficient.

Except for `AVLTree.cpp`, you should not modify any starter files, even to make their code conform to the style guide. Any style violations in starter code are intentional, either to exhibit alternate code styles or to keep the code true to its source. All code that you write must conform to the style guide unless you have a *very* good reason not to.

Notes about the Copy Constructor and Assignment Overload

There is a very fine line between copy constructor and assignment overload. This topic will be further discussed in class. For now, you need only know that copy constructor is invoked when a *new* object is constructed as a copy of another object. The assignment operator is invoked when an *existing* object gets the value of another object.

For example, if your code says:

```
AVLTree new_tree = old_tree;
```

or

```
AVLTree new_tree(old_tree);
```

You are invoking the *copy constructor* (essentially you are creating the variable and immediately making a copy of it).

If you separate this into two lines of code, then you will invoke assignment overload instead:

```
AVLTree new_tree; // default constructor called here
new_tree = old_tree // assignment overload called here
```

Further reading for interested students

When you try to make a copy of an object, those functions get called, depending on the situation. Why? The C++ compiler only makes shallow copies automatically—all the data is copied exactly from one object to the other. This is fine for fundamental types (`int`, `bool`, `float` *etc.*). But pointer types are also copied to the new object (so the new object points to the same location)! This leads to aliasing of values referred to by pointer members. We need to make a *deep* copy of the object (in this case, a new AVL tree that is identical to the original one is created at a different location in memory so any modification to one of the trees is not passed to the others).

There is nothing magical about either a copy constructor or an assignment operator overload, but it does take some time to learn how to write them. The structure is (almost) always the same, however, so you should be able to follow the code on the page below to write the ones for this assignment. Read the following article if you want more information on those two operations: http://www.cplusplus.com/articles/y8hv0pDG/.

Futher reading for *really* interested students

Deep copying can be expensive. The C++11 standard introduced *move semantics* to help with this. Move semantics transfer ownership of a value from one variable to another, which prevents the aliasing problem mentioned above and makes it cheap to assign a value to a new variable. A good introduction to move semantics is available at https://www.learncpp.com/cpp-tutorial/introduction-to-smart-pointers-move-semantics/. (As a bonus, that tutorial *also* provides an introduction to "smart pointers", a beautiful construct in C++ to help us manage memory.) If you're interested, we encourage you to implement a move constructor and overload the move assignment operator for your `AVLTree` class.

The Reference Implementation

We provide a reference implementation of the driver, named `the_avlt`, which was compiled and will run on the Zoo (but possibly not your personal computers). We encourage you to play around with it; it's quite an interesting end product that you'll end up with! Pay careful attention to the format of the output. A large fraction of your grade will be automated, so even small errors in spacing can have an impact on your grade. Fortunately, most of the output is performed by code that was written for you, so as long as you do not modify `main.cpp` or `pretty_print.cpp`, the output format should conform to expectations. You can also verify that your code is correct by executing `diff` on the output obtained by the reference implementation (an empty diff is a good diff).

Testing

This time around, you're basically on your own for testing! Except for the reference executable, we won't be releasing a testing script (there will be a `make` script on Gradescope that will allow you to confirm you submitted the appropriate files in the appropriate way). But we will drop hints here and there about what kinds of things you should look for in your testing.

In fact, we'll start those recommendations now!

1. Pay careful attention to your implementation's handling of duplicate entries.
2. Consider the worst-case scenario for a naive BST. How does your AVL tree handle it?
3. What does a newly-created tree look like? A newly-destroyed one?
4. What happens if you don't put anything at all into your tree and try to call the various functions?
5. How big can you make your tree before things start to noticeably slow down?

Remember `unittest.cpp` from the last pset? We strongly encourage you to create a version of that with your own unit tests for the AVLTree implementation. Let one of us know if you'd like some guidance on how to get started. Here are some resources to give you a better overview of unit testing in general:

- Wikipedia: Unit Testing
- Googletest
  - **Note**: Googletest (aka gtest) relies on toolchains we haven't discussed in class (and won't!) but it's an industry standard unit testing framework for C and C++ code and it's probably worth an hour or two of your time to get it set up
- Introduction to the theory of unit testing, from OSU

Submitting

Upload to Gradescope before the deadline. Do not upload object (`.o`) files or executables (such as `avlt` or `the_avlt`), but do include all source and header files needed to compile your code along with your `Makefile` and `LOG.md` file.
