lab_avl

Awful AVL Trees

Due: Mar 10, 23:59 PM

Doxygen | Lab handout

Assignment Description

In this lab we'll practice AVL tree rotations and insertions, and see some silly test cases.

Lab Insight

AVL trees can be used to build data structures like sorted maps and sets. To be quite honest, the Red-Black tree, a counter part to the AVL tree, is used more in these application because the work to insert a node for a Red-Black tree is less than that of an AVL trees. However, the general search performance of an AVL tree is better than a Red-Black tree. However, the general principle of a self-balancing tree is very powerful, so you can avoid situations where you end up with an O(n) search because you built a BST that looks like a linked list. A practical application of AVLs could include building the lookup datastructure for things like Yellowpages, a phone directory service. One could do an alphabetical search through the AVL tree to find the right number, if you may not know the persons full name, etc.

Checking Out The Code

From your CS 225 git directory, run the following on EWS:

```
git fetch release
git merge release/lab_avl -m "Merging initial lab_avl files"
```

If you're on your own machine, you may need to run:

```
git fetch release
git merge --allow-unrelated-histories release/lab_avl -m "Merging initial lab_avl files"
```

Upon a successful merge, your lab_avl files are now in your `lab_avl` directory.

As usual, don't forget to take a look at Doxygen for lab_avl.

Implement Rotation Functions

You must implement `rotateLeft()`, `rotateRight()`, and `rotateRightLeft()`. We have implemented `rotateLeftRight()` for you as an example for implementing `rotateRightLeft()`.

Implement the `rebalance()` Function

You must implement `rebalance()` function. `rebalance()` should, given a subtree, rotate the subtree so that it is balanced. You should assume that the subtree's `left` and `right` children are both already balanced trees. The node's `height` should always be updated, even if no rotations are required.

Implement the `insert()` Function

You must implement the `insert()` function. `insert()` should add a node with a key and value at the correct location in the tree, then rebalance appropriately (while returning from each recursive function) to fix the tree's balance.

Implement the `remove()` Function

You must implement the `remove()` function. `remove()` should remove the node with the specified key from the tree, then rebalance appropriately (while returning from each recursive function) to fix the tree's balance. You can assume that the key exists in the tree. You may want to use the `swap()` method.

> To match the provided output (and grading scripts), you should use IOP (in order predecessor) for removing a node with 2 children.

Testing Your Code

To test your code, compile using `make`:

```
make
```

Then run it with:

```
./testavl color
```

You will see that the output is colored — green means correct output, red means incorrect output, and underlined red means expected output that was not present. This mode is a bit experimental, and it might cause problems with your own debugging output (or other problems in general). To turn it off, simply leave off the "color" argument:

```
./testavl
```

You may also `diff` your solution with our expected output:

```
./testavl | diff -u - soln_testavl.out
```

Type `[Escape] [:] [q] [a] [ENTER]` to exit `vimdiff`.

To make the `catch` test suite, run:

```
make test
```

After compiling the test suite, run the tests using:

```
./test
```

Submitting Your Work:

The following files are used in grading:

- `avltree.cpp`
- `avltree.h`

All other files including any testing files you have added will not be used for grading.

Guide: How to submit CS 225 work using git