CS202 -- Lab B

- **CS202 -- Data Structures and Algorithms I**
- **James S. Plank**
- **This file: http://web.eecs.utk.edu/~jplank/plank/classes/cs202/Labs/LabB**

Time to implement AVL trees.

To start this lab, please do the following:

```
mkdir obj
mkdir bin
cp -r /home/jplank/cs202/Labs/LabB/include .
cp -r /home/jplank/cs202/Labs/LabB/src .
cp -r /home/jplank/cs202/Labs/LabB/makefile .
```

You will be submitting one file -- your own **src/avltree_lab.cpp**.

The AVLTree API

And take a look at **include/avltree.hpp**:

```
#include <vector>
#include <string>

namespace CS202 {

/* AVL Tree nodes are just like binary search tree nodes, but we maintain the height
   in each node. */

class AVLNode {
  public:
    AVLNode *left;
    AVLNode *right;
    AVLNode *parent;
    std::string key;
    void *val;
    size_t height;
};

class AVLTree {
  public:

    /* Constructor, copy constructor, assignment overlead, destructor.
       I am only implementing the constructor and destructor here in the notes.
       You will implement the other two in your lab. */

    AVLTree();
    AVLTree(const AVLTree &t);
    AVLTree& operator= (const AVLTree &t);
    ~AVLTree();

    void Clear();                                 // Turns the tree into an empty tree.

    bool Insert(const std::string &key, void *val);    // Insert the key and val.  Returns success (duplicates are not allowed.
    void *Find(const std::string &key) const;          // Return the val associated with the key.  Returns NULL if key not found.
    bool Delete(const std::string &key);               // Delete the node with the key.  Returns whether there was such a node.

    void Print() const;                           // These are obvious.
    size_t Size() const;
    bool Empty() const;
    size_t Height() const;
    std::string Is_AVL() const;                   // This double-checks to make sure that the tree is an AVL tree.
                                                  // It returns an empty string if it is, and an error string if it is not.

    std::vector <std::string> Ordered_Keys() const;    // Return a vector of sorted keys
    std::vector <void *> Ordered_Vals() const;         // Return a vector of the vals, sorted by the keys.

  protected:
    AVLNode *sentinel;                            // Like the bstrees, there is a sentinel.  Its right points to the root.
    size_t size;                                  // Size of the tree

    void recursive_inorder_print(int level, const AVLNode *n) const;          // A helper for Print()
    void recursive_destroy(AVLNode *n);                                      // A helper for Clear()
    AVLNode *recursive_postorder_copy(const AVLNode *n) const;               // A helper for the assignment overload
    void make_key_vector(const AVLNode *n, std::vector<std::string> &v) const; // A helper for Ordered_Keys()
    void make_val_vector(const AVLNode *n, std::vector<void *> &v) const;      // A helper for Ordered_Vals()
    std::string recursive_is_avl(const AVLNode *n) const;                      // A helper for Is_AVL()
};

};
```

You'll note that it is very similar to the binary search tree header file. There are a few important differences:

- The **AVLNode** class now includes a height.
- The **AVLNode** class' members are public. Since you can't get at the **AVLNode** pointers from an **AVLTree**, that isn't unsafe. It also facilitates writing some of the helpter procedures below.
- The sentinel's height is zero. All other nodes will have heights as defined in the AVLTree lecture notes.
- There is no **Depth()** method.
- There is a procedure called **Is_AVL()** which returns a string. If the string is empty, then the tree is an AVL tree. Otherwise, the string tells why it's not an AVL tree.
- The assignment overload and copy constructor will work differently. The assignment overload will use a protected method called **recursive_postorder_copy()** to copy the tree held by its parameter. It will copy it so that it has the same structure as the tree held by its parameter. In other words, there's no need to balance an AVL tree -- it's already balanced!

Like the Binary Search Tree lab, I have split the implementation of the class into two files:

1. **src/avltree_fixed.cpp** -- this contains methods that I have implemented:
    - The constructor
    - The copy constructor
    - The destructor
    - **Clear()**
    - **Print()**
    - **Size()**
    - **Empty()**
    - **Is_AVL()**
    - **Ordered_Vals()**
    - **recursive_inorder_print()**
    - **recursive_destroy()**
    - **make_val_vector()**
    - **recursive_is_avl()**
2. **src/avltree_lab.cpp** -- this contains dummy implementations for the things that you need to write:
    - The assignment overload
    - **Height()**
    - **Ordered_Keys()**
    - **recursive_postorder_copy()**
    - **make_key_vector()**
    - **Insert()** -- this contains the implementation from **bstree_fixed.cpp**, which is a good starting point for you. It sets the new node's height to 1.
    - **Delete()** -- this also contains the implementation from **bstree_fixed.cpp**, which is a good starting point for you.

Your job in the lab is to implement these methods so that they work on AVL trees.

I have updated **src/avltree_tester.cpp** to work with AVL trees:

```
UNIX> echo '?' | bin/avltree_tester
usage: avltree_tester prompt(- for empty) -- commands on stdin.

commands:
  INSERT name phone ssn  - Insert the person into the tree.
  FIND name              - Find the person and print them out.
  DELETE person          - Delete the person.
  PRINT                  - Print the keys using the Print() method.
  EMPTY                  - Print whether the tree is empty.
  SIZE                   - Print the tree's size.
  HEIGHT                 - Print the tree's height.
  IS_AVL                 - Verifies that the tree is an avl tree.
  KEYS                   - Print the keys using the Ordered_Keys() method.
  VALS                   - Print the vals using the Ordered_Vals() method.
  PRINT_COPY             - Call the copy constructor and call its Print() method.
  ASSIGNMENT             - Test the assignment overload.
  CLEAR                  - Clear the tree back to an empty tree.
  DESTROY                - Call the destructor and remake an empty tree.
  QUIT                   - Quit.
  ?                      - Print commands.
UNIX>
```

I've denoted the commands which have changed significantly from the binary search tree lab in blue. To wit:

- **Insert()** has to result in a valid AVL tree, following the specifications in the AVLTree lecture notes.
- **Delete()** also has to result in a valid AVL tree.
- **IS_AVL** calls the **Is_AVL()** method to test whether the tree is indeed an AVL tree. If it is, it prints nothing. Otherwise, it prints the error string and exits.
- **PRINT_COPY** works like before -- calling the copy constructor and then calling the **Print()** method. This duplicates the tree -- it doesn't do balancling like before.
- **ASSIGNMENT** tests the assignment overload. It returns silently, but if you've messed up the assignment overload, you'll discover it in subsequent commands.

Getting Started - Hints

When you compile this out of the box, it works like the binary search tree lab, but with some things broken. One problem is that every node gets inserted with a height of one. Take a look:

```
UNIX> bin/avltree_tester '--->'
---> INSERT Kim-Kardashian 0 0
---> INSERT Khloe-Kardashian 0 0
---> INSERT Kourtney-Kardashian 0 0
---> PRINT
  (1) Kourtney-Kardashian
(1) Kim-Kardashian
  (1) Khloe-Kardashian
---> IS_AVL
Not an AVL tree:
Node Kim-Kardashian's height is less than or equal at least one child's height.
UNIX>
```

As I said above, **Insert()** creates with nodes with heights of one, and nothing updates them, so even though that tree is indeed an AVL tree, **Is_AVL()** returns, **false**. This is because **Is_AVL()** checks a node's height with relation to its children's heights.

As a first task, I suggest that you update **Insert()** so that when it's done, it traverses the path from the newly inserted node to the root, updating heights if necessary. I have done that, and now take a look at how it works:

```
UNIX> bin/avltree_tester '--->'
---> INSERT Kim-Kardashian 0 0
---> INSERT Khloe-Kardashian 0 0
---> INSERT Kourtney-Kardashian 0 0
---> PRINT
  (1) Kourtney-Kardashian
(2) Kim-Kardashian                        # Insert() has correctly updated the height.
  (1) Khloe-Kardashian
---> IS_AVL                               # Now it is correctly identified as an AVL tree.
---> INSERT Caitlyn-Jenner 0 0
---> INSERT Kris-Kardashian 0 0
---> PRINT                                # Adding these two nodes still results in an AVL tree.
    (1) Kris-Kardashian
  (2) Kourtney-Kardashian
(3) Kim-Kardashian
  (2) Khloe-Kardashian
    (1) Caitlyn-Jenner
---> IS_AVL
---> INSERT Rob-Kardashian 0 0
---> PRINT                                # However, now the "Kourtney" node is imbalanced.
      (1) Rob-Kardashian
    (2) Kris-Kardashian
  (3) Kourtney-Kardashian
(4) Kim-Kardashian
  (2) Khloe-Kardashian
    (1) Caitlyn-Jenner
---> IS_AVL
Not an AVL tree:
Node Kourtney-Kardashian's height is more than two greater than at least one child's height.
UNIX>
```

Here's how I suggest you proceed (you don't have to do it this way -- this is just my suggestion):

1. Write and test **Height()**

2. Do the assignment overload. I know it's a little confusing, but start by testing it with very small trees, and have it print out nodes as it creates them. You'll note that **recursive_postorder_copy()** does not have a parameter for the tree that it is copying -- just a node that roots

a subtree. So, how can you determine whether **n** is the sentinel node for its tree? Test to see if its height is zero!

3. Copy your **Ordered_Key()** and **make_key_vector()** from your binary search tree lab. They should work without any modification. Test.
4. Write the following procedure (not part of the class -- just a procedure):

```
bool imbalance(const AVLNode *n)
```

   This should simply check whether is an imbalance around a single node (by checking the heights of its children). Test it by calling it as you set the heights in **Insert()**: If you insert A, B and C, then there will be an imbalance at A after inserting C.

5. Write the following procedure:

```
void rotate(AVLNode *n)
```

   This should rotate about the given node. Since this procedure exists outside the class definition, you don't have access to the **sentinel** pointer. However, if a node's height is 0, then it is the sentinel node. Just as a hint, it's best to have four **(AVLNode *)'s**:

   - **n**: The node itself.
   - **parent**: The node's parent. Exit if the node's parent is the sentinel.
   - **grandparent**: The node's grandparent. It's ok if this is the sentinel.
   - **middle**: The "middle" node in the rotation -- it is the parent's other child. It's ok if this is the sentinel.

   That's all you need to have to implement **rotate()**. I tested it by having **Insert()** call **rotate()** on the parent of every newly inserted node, so long as the parent wasn't the root of the tree. That allowed me to test a lot of cases. Once I was confident that rotate was working, I took out this testing code.

6. Write the following procedure:

```
void fix_height(AVLNode *n)
```

   This should make sure that a node's height is correctly calculated from the heights of its children. Test it as you write the following code:

7. Write the following procedure:

```
void fix_imbalance(AVLNode *n)
```

   This is called on an imbalanced node, and what you do is identify whether it's a Zig-Zig or Zig-Zag imbalance, using the "Imbalanced Identification Picture" from the AVL Tree lecture notes. Then you do the correct rotations. Write this slowly -- first implement Zig-Zig in one direction, then in the other direction, then Zig-Zag in one direction and Zig-Zag in the other direction.

   Test is by calling it in **Insert()** when it detects an imbalance. You'll note that once you have this working, insertion is done!

8. Write the code to do deletion -- it will use **imbalance()** and **fix_imbalance()**, and will be a lot easier than you thought it would be when you started this lab.
9. Now you can try the gradescripts.

The Gradescript

The gradescripts work in the following way:

- Gradescripts 1-25 test **Insert()**
- Gradescripts 26-50 test **Insert()** and **Delete()**
- Gradescripts 51-75 make sure that the assignment overload works (either by calling ASSIGNMENT or PRINT_COPY).
- The remaining gradescripts test everything.