AVL Tree

This assignment involves implementing an AVL tree as a C++ template. Because this assignment will be auto-graded using a test harness I will provide, your implementation must conform to the public interfaces below, and include at least the private members that are shown:

```cpp
// AVLNodeT.h
// . . . includes omitted
enum BFactor {LEFTHI, RIGHTHI, EQUALHT, DBLRIGHTHI, DBLLEFTHI, BALUNKNOWN};

template<typename T> class AVLNodeT {
  friend class Monk;
public:
  T Element;
  AVLNodeT<T>* Left;
  AVLNodeT<T>* Right;
  BFactor Balance;

  AVLNodeT();
  AVLNodeT(T D, AVLNodeT<T>* L = NULL, AVLNodeT<T>* R = NULL);
  string bfToString() const; // return "LH", "RH", "EQ", "RR", // "LL" or "??", respectively
};

// AVL.h
// . . . includes omitted

template<typename T> class AVL {
private:
  AVLNodeT<T>* Root; // access to root node, if any
  // . . . the rest is up to you . . .

public:
  AVL(); // create empty AVL
  AVL(const T& D); // create one-node AVL with given data
  AVL(const BST<T>& Source); // deep copy support
  AVL<T>& operator=(const AVL<T>& Source);

  bool Insert(const T& D); // insert data into new node
  bool Delete(const T& D); // delete node with matching data
  T* const Find(const T& D); // return pointer to matching data, // or NULL if not found

  const T* const Find(const T& D) const; // same, but read-only access
  void Clear(); // reset tree to empty state
  ~AVL(); // destroy all tree contents

  friend class Monk; // give the test harness access
};
```

You may safely add features to the given interface, but if you omit or modify members of the given interface you will almost certainly face compilation errors when you submit your implementation for testing.

You must place the declaration of your AVL template in a header file named `AVL.h` because `Monk.h` (one of the test harness files) will explicitly `#include` that file. You may add iterator support, but it will not be tested. Note that if you do add an iterator, or six, you must still have `Find()`, `Insert()`, and `Delete()` functions that conform exactly to the interfaces given above.
Design and implementation requirements

There are some explicit requirements, in addition to those on the Programming Standards page of the course website:

- You must implement C++ templates for the AVL itself and also for the AVL nodes, conforming to the given interface.
- You must handle copy issues correctly. We will certainly test for this. Failures in your copy logic will almost certainly result in program crashes and scores of zero.
- You may assume that any data type stored in your template will handle its own copy issues correctly (as it must).
- You may assume that any data type stored in your template will provide `operator<<( )`.
- You may assume that any data type stored in your template will support all the relational operators.
- You must properly allocate and deallocate memory, as needed.
- The insertion logic must not allow duplicate records to be inserted.
- You must provide feedback to the client when an operation fails. The given prototypes of the member functions indicate where that is necessary. Under no circumstances should any of the template functions, other than `Display()` and its helper, write output.

The `friend` declaration provides a class that I will supply with my test harness privileged access to the internals of your AVL object.

Testing:

I will be testing your implementation with my own test driver. I may (or may not) release information about that driver before the assignment is due. In any case, it is your responsibility to design and carry out a sensible test of your implementation before submitting it. For that purpose, you may share test code (but absolutely no tree code!!) via the class Forum.

Be warned that your copy logic will be tested thoroughly, as will the operation of the destructor. Be sure you test all of the interface elements thoroughly, both in isolation and in interleaved fashion.

Also be sure you read and understand the test build instructions that will be posted on the website. The default build you would get if you're developing under Eclipse or KDevelop or .NET will often mask even serious errors in an implementation.

Evaluation:

You should document your implementation in accordance with the Programming Standards page on the course website. It is possible that your implementation will be evaluated for documentation and design, as well as for correctness of results. If so, your submission that achieved the highest score will be evaluated by one of the TAs, who will assess a deduction (ideally zero) against your score from the Curator.

Note that the evaluation of your project may depend substantially on the quality of your code and documentation.

What to turn in and how:

This assignment will be auto-graded using a test harness on the Curator system. The testing will be done under Windows XP using g++ 3.4.4.

Submit a single zip archive file containing the C++ header files for your template declarations and implementations to the Curator System. Submit only the template declarations and implementations. Submit nothing else. Be sure that your header file only contains `include` directives for Standard C++ header files; any other `include` directives will certainly result in compilation errors. Be sure you must use the specified names for your header files.

Your submitted source files will then be compiled with the test harness using the following command:
g++ -o <name we give the executable> *.cpp

Instructions, and the appropriate link, for submitting to the Curator are given in the Student Guide at the Curator website:

http://www.cs.vt.edu/curator/

You will be allowed to submit your solution up to ten times.

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you must include the pledge statement provided on the Programming Standards page in one of your submitted files.

Note on deletion:

The test harness will expect you to handle deletion in a precisely-specified manner. Deletion of a leaf simply requires deallocating the node and setting the pointer to that node to NULL. Deletion of a node with one non-empty subtree simply requires setting the pointer to that node to point to the node’s subtree, and then deallocating the node.

Deletion of a node with two non-empty subtrees must be handled in the following manner. First, locate the node rMin that holds the minimum value in the right subtree of the node to be deleted. Then, detach rMin from the right subtree using the appropriate logic for a leaf or a one-subtree node. Next, replace the node to be deleted with rMin, by resetting pointers as necessary. Finally, deallocate the node that was to be deleted.

This technique is a modification of the delete-by-merging described in Drozdek and addresses the issue of increasing the height of the tree that his algorithm would suffer. It should be cheaper, on average, than the delete-by-copy technique he describes, since the data objects stored in the tree are likely to be much larger than a few pointers.