Programming Assignment #4

Due: Monday, March 7, 2016 at 11:59 pm

This assignment gives you some experience implementing a simple API for binary trees. You will also have another chance to become comfortable with recursive algorithms while manipulating binary trees. Because trees are a form of sequence container, the interface will be simpler than the interfaces we've seen for lists. In other words, since the container must remain sorted, there are no methods for adding/removing to/from the beginning/end or inserting at any arbitrary location.

The task is to implement two classes named BSTree and AVLTree which clients can use to store data. The public interface for the BSTree class is straight-forward. The partial struct/class definitions are below:

```
template <typename T>
class BSTree
{
  public:
    struct BinTreeNode
    {
      BinTreeNode *left;
      BinTreeNode *right;
      T data;
      int balance_factor;  // unused (for future use)
      unsigned count;      // optional, extra credit
    };
    typedef BinTreeNode* BinTree;
    BSTree(ObjectAllocator *OA = 0, bool ShareOA = false);
    BSTree(const BSTree& rhs);
    virtual ~BSTree();
    BSTree& operator=(const BSTree& rhs);
    virtual void insert(const T& value);
    virtual void remove(const T& value);
    const BinTreeNode* operator[](int index) const;
  protected:
    // Protected methods
  private:
    // private members and methods
};
```

```
template <typename T>
class AVLTree : public BSTree<T>
{
  public:
    AVLTree(void);
    virtual ~AVLTree();
    virtual void insert(const T& value);
    virtual void remove(const T& value);

  private:
    // Private methods
};
```

Notes (more details will be discussed in class)

1. The constructor takes a pointer to an *ObjectAllocator*. You will use this for all allocations/deallocations in your class. You don't own this allocator, so do not destroy it. If this parameter is 0, you will instantiate your own allocator (setting *UseCPPMemManager* to true) and be responsible for it.
2. The only public method that throws an exception is the *insert* method. There is only one kind of exception that can be thrown: out of memory. The exception class is provided.
3. You will not be given duplicate values to insert. Sometimes, we would handle duplicates by throwing an exception. To keep the implementation simpler, you won't have to deal with that case and are guaranteed not to receive any duplicates.
4. You have to catch the exception from the ObjectAllocator and throw a `BSTException.` Failure to do so will cause your program to crash because the client (driver) is expecting a `BSTException` not an ObjectAllocator exception and won't catch it.
5. The *find* method returns a boolean, indicating whether or not the item was found. There is a second parameter which will contain the number of comparisons performed to determine the outcome of *find*. **Make sure your counts match the counts from the sample driver to receive credit.**
6. Like all of our templated classes, you will include the implementation files in the header files.
7. The public *root* method simply returns the root of the tree. This allows the user to walk the tree. (Normally, we wouldn't want that, but for learning purposes, we would like to be able to examine the tree outside of the class, such as within the text or GUI driver.)
8. This first part of this assignment (implementing the *BSTree* class) is trivial since I have already given you almost all of the code and explained it. That code is available to you from the web site. You just need put it in a class, and then templatize it.
9. You'll notice that AVL.h includes `<stack>`. For this assignment you can use `std::stack` from the STL instead of having to write your own. You'll need a stack to implement the simple balancing algorithm as it was demonstrated in class. If you'd like, you can balance the tree recursively, which doesn't require a separate `std::stack`. The choice is yours.
10. You can't change the public interface at all. That means you can't add, remove, or change any *public* method. You can add anything you like to the *private* section, and will likely need quite a few methods.
11. Additional notes are on the web page for this assignment.

Implementation Steps
If you break down the assignment into manageable pieces, it isn't that difficult. However, if you decide to start immediately with the sample templated driver, you will get nowhere fast. You should solve the problem in pieces. This is a smarter approach to implement this assignment (or any assignment, for that matter).

1. Implement a non-templated (using integers) BSTree class and run some tests on it to make sure it works. This should only take a short amount of time since I've provided all of the code for this on the web pages.
2. Derive a non-templated AVLTree from BSTree and run some tests on it.
   a) Implement the inefficient balancing algorithm from the pseudocode I demonstrated in class. **Copy and paste the pseudocode into your code and use that to guide your implementation.**
   b) Optionally, implement the indexing operation. You have to write the code to store the size of each subtree (count) in the nodes as this is the only code for the BST that wasn't given to you.
3. Convert the classes into templatized classes and run your same tests on it. (You'll need to modify your driver)
4. Try your completed code with the sample templated driver that I provided.

Extra Credit - Indexing
In order to allow the client to use the subscript operator, you need to implement the code for this. In generic data structures terms, this is called the *select* operation. The C++ community tends to just call it indexing via the subscript operator. If you choose to implement this operation, you can earn an extra 25 points on the assignment. If you do implement it, you must be sure to return **true** from the static method called *ImplementedIndexing* so the tests will include this functionality and you will receive credit. Also, realize that there are two things that need to be implemented. The first is storing the count in each node. The count indicates how many nodes are in the subtree, including the node itself. You need these counts so you can implement an **efficient** indexing operator. If you implement it poorly (inefficiently) you will not earn any extra credit. See the website for more details.

Testing
As always, testing represents the largest portion of work and insufficient testing is a big reason why a program receives a poor grade. (My driver programs take longer to create than the implementation file itself.) Sample drivers for this assignment are available. You should use the sample driver program as an example and create additional code to thoroughly test all functionality with a variety of cases. (Don't forget stress testing.)

What to submit
You must submit your program files (header and implementation files) in a .zip file to the appropriate submission page by the due date and time.

Files	Description
BSTree.h AVLTree.h	The header files. **No** implementation code is allowed. The public interface must be exactly as described above.
BSTree.cpp AVLTree.cpp	The implementation files. All implementation code goes here. You must document the functions using Doxygen syntax.

Usual stuff
Your code must compile using the compilers specified in this assignment to receive credit. Note that you must not submit any other files in the zip file other than the 4 files specified. Details about what to submit and how are posted on the course website and in the syllabus.

Make sure your name and other info is in all documents.