**Programming Assignment**

**Binary Tree**

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 end/beginning 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 class definitions are below:

|  |  |
| --- | --- |
| **template** <**typename** T>  **class** BSTree  {  **public:**  **struct** BinTreeNode  {  BinTreeNode \*left;  BinTreeNode \*right;  T data;  **int** balance\_factor; // optional  };  **typedef** BinTreeNode\* BinTree;  BSTree(ObjectAllocator \*OA = 0, **bool** ShareOA =  **false**);  **virtual** ~BSTree();  BSTree(**const** BSTree& rhs);  BSTree& **operator**=(**const** BSTree& rhs); | **template** <**typename** T>  **class** AVLTree : **public** BSTree<T>  {  **public**:  AVLTree(**void**);  **virtual** ~AVLTree();  **virtual void** insert(**int** value)  **throw**(BSTException);  **virtual void** remove(**int** value);  **private**:  // Private method  }; |
| **virtual** **void** insert(const T& value) **throw**(BSTException);  **virtual** **void** remove(const T& value);  **const** BinTreeNode\* **operator**[](**int** index) **const**;  **protected**:  // Protected methods  **private**:  // private members and methods  }; | |

**Notes**

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 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.
4. 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.**
5. Like all of our templated classes, you will include the implementation files in the header files.
6. 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.)
7. This first part of this assignment (implementing the *BSTree* class) is trivial since I have already shown you almost all of the code and explained it. That code is available to you from slides. You just need add the code to store the number of nodes in each subtree, put it in a class, and then templatize it.
8. 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.
9. 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.

**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 you are provided with all of the code for this.
2. Add the code to support the *select* operation, which is the subscript operator in C++. You have to write the code to store the size of each subtree in the nodes as this is the only code for the BST that wasn't given to you.
3. Derive a non-templated AVLTree from BSTree and run some tests on it.
   1. 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.**
   2. Depending on how you factored out the select code in BSTree, you may have to add this to the AVLTree class.
   3. Optionally, implement a more efficient method for balancing that uses balance factors. (The efficient implementation has no extra calls to *height*).
4. Convert the classes into templatized classes and run your same tests on it. (You'll need to modify your driver)
5. Try your completed code with the sample templated driver that I provided.

**Extra Credit - Efficient Balancing**

The pseudocode I demonstrated was not optimal, but it works fine and gives you an understanding of how the algorithm works. In order for this data structure to be usable in a real application, the balancing algorithm needs to be more efficient. This method will require the use of the *balance\_factor* field in the node structure. Note that if you don't implement this technique, you can safely ignore this extra field. If you choose to implement this balancing technique during item insertion and deletion, you can earn an extra 25% on the assignment. If you choose to do so, you must be sure to return **true** from the static method called *ImplementedBalanceFactor* so the tests will include this functionality and you will receive credit.

**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 program for this assignment are available. You should use the 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) and the compiled help file (index.chm) 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 5 files specified. Details about what to submit and how are described in the syllabus.

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