cs280-a3-avl

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.

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 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.

Implementation Ideas

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 we've provided most of the code for this in class.
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.

Indexing (a.k.a subscript operator)

Note: Near the top of the driver file, there is this line:

```
#define SHOW_COUNTS
```

This is what turns on/off the counts when displaying the tree. Remove it and you won't see any counts displayed. If it's defined, you'll see them. There are sample outputs with counts in the **data** folder.

Since the subscript operator (operator[]) is not recursive, you'll need a helper function to do the recursion:

```
some_helper(node, index);
```

where node is the node (initially the root) and index is the position in the tree.

Suppose we're looking for the i'th node and we're at some node (the root initially). Assume that L is the number of nodes in the left subtree.

- If node is NULL, return NULL. (Base case)
- If the left subtree has more than i nodes (L > i), then it's in the left subtree.
  - Recursively call the method with the left node and the index.
- If the left subtree has less than i nodes (L < i), then it's in the right subtree.
  - Recursively call the method with the right node and i - L - 1.
- Else the left subtree has exactly i nodes so it's in the current node.

Looking at the example below, the values represent: Key[count]. Key is the value in the node and count is the number of nodes in the (sub)tree. Because I inserted Keys from 0 to 9, the Key is the index as well. This makes following the example below easy. On non-sequential data (random, strings, etc.) this will not necessarily be the case.

```
                        5[10]

          2[5]                          8[4]

0[2]            3[2]            6[2]            9[1]

     1[1]            4[1]            7[1]
```

Suppose we're looking for tree[3] (L is the number of nodes in the left subtree, i is the index): - Starting at the root: - L > i so it's in the left subtree. (Recurse left with node 2[5] and i, which is 3) - Now, L < i, so it's in the right subtree. (Recurse right with node 3[2] and i - L - 1, which is 0) - Now, L == i, so the value is in the current node, 3[2]. - Return the node.

Suppose we're looking for tree[8] (L is the number of nodes in the left subtree, i is the index): - Starting at the root: - L < i so it's in the right subtree. (Recurse right with node 8[4] and i - L - 1, which is 2) - Now, L == i, so it's in the current node, 8[4]. - Return the node.

Suppose we're looking for tree[7] (L is the number of nodes in the left subtree, i is the index): - Starting at the root: - L < i so it's in the right subtree. (Recurse right with node 8[4] and i - L - 1, which is 1.) - Now, L > i, so it's in the left subtree. (Recurse left with node 6[2] and i, which is 1.) - Now, L < i, so it's in the right subtree. (Recurse right with node 7[1] and i - L - 1, which is 0) - Now, L == i, so the value is in the current node, 7[1]. - Return the node.

Try a couple more on your own to convince yourself that this works.

Notice that you don't necessarily have to do any checking of the subscript. For example, if I ask for tree[12] (non-existent), it will eventually recurse on an empty child (NULL). Be sure to return NULL from the method, indicating that nothing was found. This is the non-recursive case (base case) in the algorithm. Of course, as an optimization, you could just check the value of the index and if it's out of range, just return NULL. We aren't throwing any exceptions for an out-of-bounds index.

Finally, realize that this doesn't have to be recursive as you are only walking down the tree.

FAQ

You said that we can't call new in our tree classes. How do we construct our own ObjectAllocator if we can't call new.

I meant that you can't use `new` to allocate any memory for nodes (or anything else). The only place that you need to call `new` is when you are constructing your own allocator. (You aren't allocating any nodes.)

when you call placement `new` to construct the node in the memory returned from the allocator. All nodes that you create must be created via the ObjectAllocator. If you call `new` for anything but the items above, your will lose points. Realize that this isn't a limitation for your program because you don't need to use new except where specified above. I would expect to see only 2 or 3 occurrences of the keyword `new` in your entire program. (You DID create a helper function to allocate a node instead of calling the allocator from several places in your code, didn't you?)

I'm getting weird errors from the compilers when I try to build my code. The errors say something about a bad token '<'. What's wrong?

You probably are including the implementation file twice. Remember, it's a templated class, so you have to include the implementation with the header file. Since it's already included in the header, adding it to the project or on the command line causes the code to be added twice. See the command line posted for an example of correct usage.

I'm making a function that returns a BinTreeNode* but the compiler is complaining that it doesn't know what a BinTreeNode is. What's wrong?

You probably forgot to use the keyword `typename` with the return value. Here's an example of how you would specify a function called make_node that creates a node:

```
template <typename T>
typename BStree<T>::BinTreeNode* BStree<T>::make_node(const T& Item) const
```

Since `BSTree` is a templated class, the compiler doesn't know the the type of `T` yet, so it can't look for `BinTreeNode`. Since it can't look it up, it assumes that `BinTreeNode` is a member variable of `BSTree`, which doesn't make sense when the compiler expects to find a type. (It is the return-type afterall). In order to tell the compiler that `BinTreeNode` is a type and not a member variable, use the `typename` keyword. Refer to your CS170 or CS225 notes for more information.

My ObjectAllocator isn't working correctly. How do I implement this assignment without a working allocator?

Use the dummy allocator that I provided, not the one that you created. It's just a wrapper around new/delete, but it contains everything you need to implement this assignment properly. Your stress tests won't be as fast, but don't worry about that. I will be using the dummy allocator when I grade your code.

I'm not getting the same number of compares that your output shows. For example, in an empty tree, I have 0 compares, but the output shows 1. Why is that?

Because compares is not actually counting comparisons. It's counting the number of times the recursive method was called. So, when you first check the tree and find it empty (NULL), you haven't actually compared the value in the node with the value you are looking for. However, you did have to check (compare with NULL) the pointer, so we count that. Essentially, if you just increment the counter upon entering the recursive find() method, you will get the correct results.

I'm confused about the two parameters in the constructor: `ObjectAllocator *OA = 0` and `bool ShareOA = false`. They seem to contradict each other.

The parameters are not related. The first one is used to provide an allocator to the `BSTree`/`AVLTree` object. There are two possible situations:

If the parameter `OA` is non-NULL, then a valid allocator has been provided. The `BSTree` object should use that allocator for all of its allocations. Also, the object does not own the allocator and should not destroy (`free`) it. The owner of the allocator (the driver) will free it when it's finished with it.

If the parameter `OA` is NULL, then the BSTree object must create its own allocator and use that for all of its allocations. The object now owns the allocator and must destroy (`free`) it when the object is finished with it (most likely in the destructor). Make sure to set `UseCPPMemManager` to `true` when you construct your own `ObjectAllocator`. This will make sure that the allocator will have enough memory (pages) for the client.

The second parameter tells the copy constructor and assignment operator how to construct the copy or do the assignment (i.e. whether or not to use the right-hand-side allocator).

If the object being copied/assigned from has its `ShareOA` set to true, then the copy will use the allocator from the copied (right-hand-side) object. The object does not own the allocator and should not destroy (`free`) it. The value of `ShareOA` should be propagated to other copies made from this object. This allows one `ObjectAllocator` to be used for all `BSTrees` in the program.

If the object being copied has its `ShareOA` set to false, then the copy will create its own allocator and use that for all of its allocations. The object now owns the allocator and must destroy (`free`) it when the object is finished with it (most likely in the destructor). This means that the copy will not share its allocator with other copies.

Some sample code should clarify this:

```
template <typename T>
BSTree<T>::BSTree(const BSTree &rhs)
{
    // If we are to use rhs' allocator
  if (rhs.shareAllocator_)
  {
    objAllocator_ = rhs.objAllocator_; // Use rhs' allocator
    freeAllocator_ = false;            // We don't own it (won't free it)
    shareAllocator_ = true;            // If a copy of 'this object' is made, share the allocator
  }
  else // No sharing, create our own personal allocator
  {
    OAConfig config(true);   // Set UseCPPMemManager to true, default the others
    objAllocator_ = new ObjectAllocator(sizeof(BinTreeNode), config);

    freeAllocator_ = true;   // We own the allocator, we will have to free it
    shareAllocator_ = false; // Do not share this allocator with any other list
  }
```

```
  // other stuff

}
```

Note: Don't copy the value of `shareAllocator_` or `objAllocator_` (or whatever you've named these private members) in the copy constructor or assignment operator. Also, DO NOT make a copy of the `ObjectAllocator` in your copy constructor!

I don't see a copy constructor or assignment operator in the AVLTree class. Is that a mistake?

No. Since there is no additional data in `AVLTree` that needs to be copied or assigned, the default copy constructor and assignment operator generated by the compiler are adequate. Remember that the default copy constructor and assignment operator in the derived class will call the base class methods, which is where all of the work is done.

If you do add additional fields to the `AVLTree`'s private section, you may need to define and implement these methods for them to work properly. Consult your CS170 notes regarding when the compiler-generated methods are inadequate.

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.)

Files Provided

The BST interface[1] and AVL interface[2] .

Sample driver[3] containing loads of test cases as usual. There is also a spell-checker implementation[4] to mimick a client using your lib for an actual application that uses words instead of numbers.

There are a number of sample **outputs** as usual in the **data** folder.

The spell-checker driver

As mentioned above, there is a spell-checker implementation[5].

The program will accept 3 command line arguments:

```
driver-spell [test_number] [dictionary] [show_tree]
```

Example:

```
driver-spell 2 allwords.txt 1
```

This will run the second test with the dictionary allwords.txt and turn on the displaying of the tree. A few sample dictionaries are provided in **data/dictionaries**. Here's an example running this:

```
driver-spell 2 lexicon.txt 1
```

[1] code/BSTree.h
[2] code/AVLTree.h
[3] code/driver-sample.cpp
[4] code/driver-spell.cpp
[5] code/driver-spell.cpp

Compilation:

These are some sample command lines for compilation. GNU should be the priority as this will be used for grading.

GNU g++: (Used for grading)

```
g++ -o vpl_execution driver-sample.cpp ObjectAllocator.cpp PRNG.cpp \
    -Werror -Wall -Wextra -Wconversion -std=c++14 -pedantic -Wno-deprecated
```

Microsoft: (Good to compile but executable not used in grading)

```
cl -Fems driver-sample.cpp PRNG.cpp ObjectAllocator.cpp \
   /WX /Zi /MT /EHsc /Oy- /Ob0 /Za /W4 /D_CRT_SECURE_NO_DEPRECATE
```

Deliverables

You must submit your program files (header and implementation file) by the due date and time to the appropriate submission page as described in the syllabus.

BSTree.h and AVLTree.h

The header files. No implementation is allowed by the student, although the exception class is implemented here. The public interface must be exactly as described above.

BSTree.cpp and AVLTree.cpp

The implementation file. All implementation goes here. You must document this file (file header comment) and functions (function header comments) using Doxygen tags as usual. Make sure you document all functions, even if some were provided for you.