



```

echo "Compiling..." Compiling... g++ -o out SimpleAllocator.cpp prng.cpp test.cpp -std=c++17 -Wall In file included
from AVL.h:187, from test.cpp:16: AVL.cpp: In instantiation of 'void AVL<T>::add(const T&) [with T = int]':
test.cpp:335:20: required from 'void addInts(AVL<T>&, int, bool, bool, bool) [with T = int]' test.cpp:501:21: required
from here AVL.cpp:12:20: error: cannot bind non-const lvalue reference of type 'BST<int>::BinTreeNode*&' to an
rvalue of type 'BST<int>::BinTree' {aka 'BST<int>::BinTreeNode*'} 12 | add_(this->root(), value, pathNodes); |
~~~~~^~ AVL.cpp:35:45: note: initializing argument 1 of 'void AVL<T>::add_(typename BST<T>::BinTree&,
const T&, AVL<T>::PathStack&) [with T = int; typename BST<T>::BinTree = BST<int>::BinTreeNode*];
AVL<T>::PathStack = std::stack<BST<int>::BinTreeNode*, std::deque<BST<int>::BinTreeNode*,
std::allocator<BST<int>::BinTreeNode*> > >]' 35 | void AVL<T>::add_(typename BST<T>::BinTree& tree, const T&
value, PathStack& pathNodes) { | ~~~~~^~ AVL.cpp: In instantiation of 'void
AVL<T>::remove(const T&) [with T = int]': test.cpp:414:27: required from 'void removeInts(AVL<T>&, bool, int,
bool, bool) [with T = int]' test.cpp:506:24: required from here AVL.cpp:20:23: error: cannot bind non-const lvalue
reference of type 'BST<int>::BinTreeNode*&' to an rvalue of type 'BST<int>::BinTree' {aka
'BST<int>::BinTreeNode*'} 20 | remove_(this->root(), value, pathNodes); | ~~~~~^~ AVL.cpp:58:48: note:
initializing argument 1 of 'void AVL<T>::remove_(typename BST<T>::BinTree&, const T&, AVL<T>::PathStack&)
[with T = int; typename BST<T>::BinTree = BST<int>::BinTreeNode*; AVL<T>::PathStack =
std::stack<BST<int>::BinTreeNode*, std::deque<BST<int>::BinTreeNode*,
std::allocator<BST<int>::BinTreeNode*> > >]' 58 | void AVL<T>::remove_(typename BST<T>::BinTree& tree, const
T& value, PathStack& pathNodes) { | ~~~~~^~ make: *** [Makefile:15: compile]
Error 1 // Chatlog: https://chat.openai.com/share/e441bfa7-82fa-4931-b40a-5cb9cbde1c9f #include "AVL.h" using
namespace std; // ----- Public Methods ----- // Inserts a value into the AVL tree. template
<typename T> void AVL<T>::add(const T& value) { PathStack pathNodes; add_(this->root(), value, pathNodes); //
Note: It's already balanced inside the add_ function. } // Removes a value from the AVL tree. template <typename
T> void AVL<T>::remove(const T& value) { PathStack pathNodes; remove_(this->root(), value, pathNodes); } //
Returns an inorder traversal of the AVL tree. template <typename T> std::stringstream AVL<T>::printInorder()
const { std::stringstream ss; printInorder_(this->root(), ss); return ss; } // ----- Private Helper Methods -
----- // Recursive helper function to insert a value into the AVL tree. template <typename T> void
AVL<T>::add_(typename BST<T>::BinTree& tree, const T& value, PathStack& pathNodes) { if (!tree) { tree = this-
>makeNode(value); } else if (value < tree->data) { pathNodes.push(tree); add_(tree->left, value, pathNodes); } else
if (value > tree->data) { pathNodes.push(tree); add_(tree->right, value, pathNodes); } else { throw
BSTException(BSTException::E_DUPLICATE, "Duplicate value"); } // Update node count and balance factor. tree-
>count = 1 + this->size_(tree->left) + this->size_(tree->right); tree->balanceFactor = this->height_(tree->left) - this-
>height_(tree->right); // Ensure the subtree remains balanced. balance(tree); } // Recursive helper function to
remove a value from the AVL tree. template <typename T> void AVL<T>::remove_(typename BST<T>::BinTree&
tree, const T& value, PathStack& pathNodes) { if (!tree) { throw BSTException(BSTException::E_NOT_FOUND,
"Value not found"); } pathNodes.push(tree); if (value < tree->data) { remove_(tree->left, value, pathNodes); } else if
(value > tree->data) { remove_(tree->right, value, pathNodes); } else { if (!tree->left || !tree->right) { typename
BST<T>::BinTree temp = tree; tree = tree->left ? tree->left : tree->right; delete temp; } else { // If both sides are
equal. typename BST<T>::BinTree temp = tree->left; // Use predecessor. while (temp->right) { temp = temp->right;
} tree->data = temp->data; remove_(tree->left, temp->data, pathNodes); } } if (!tree) return; // If the tree is now
empty, just return. // Update node count and balance factor. tree->count = 1 + this->size_(tree->left) + this-
>size_(tree->right); tree->balanceFactor = this->height_(tree->left) - this->height_(tree->right); // Ensure the
subtree remains balanced. balance(tree); } // Left rotation to maintain AVL tree balance. template <typename T>
void AVL<T>::rotateLeft(typename BST<T>::BinTree& tree) { typename BST<T>::BinTree newRoot = tree->right;
tree->right = newRoot->left; newRoot->left = tree; tree = newRoot; // Update counts and balance factors for the
nodes after rotation. tree->left->count = 1 + this->size_(tree->left->left) + this->size_(tree->left->right); tree->left-
>balanceFactor = this->height_(tree->left->left) - this->height_(tree->left->right); tree->count = 1 + this->size_(tree-
>left) + this->size_(tree->right); tree->balanceFactor = this->height_(tree->left) - this->height_(tree->right); } // Right
rotation to maintain AVL tree balance. template <typename T> void AVL<T>::rotateRight(typename
BST<T>::BinTree& tree) { typename BST<T>::BinTree newRoot = tree->left; tree->left = newRoot->right; newRoot-
>right = tree; tree = newRoot; // Update counts and balance factors for the nodes after rotation. tree->right->count
= 1 + this->size_(tree->right->left) + this->size_(tree->right->right); tree->right->balanceFactor = this->height_(tree-
>right->left) - this->height_(tree->right->right); tree->count = 1 + this->size_(tree->left) + this->size_(tree->right);
tree->balanceFactor = this->height_(tree->left) - this->height_(tree->right); } // Left-Right rotation to maintain AVL
tree balance. template <typename T> void AVL<T>::rotateLeftRight(typename BST<T>::BinTree& tree) {
rotateLeft(tree->left); rotateRight(tree); } // Right-Left rotation to maintain AVL tree balance. template <typename T>
void AVL<T>::rotateRightLeft(typename BST<T>::BinTree& tree) { rotateRight(tree->right); rotateLeft(tree); } //
Balance the given subtree. template <typename T> void AVL<T>::balance(typename BST<T>::BinTree& tree) { if
(tree->balanceFactor > 1) { if (tree->left->balanceFactor >= 0) { rotateRight(tree); } else { rotateLeftRight(tree); } }
else if (tree->balanceFactor < -1) { if (tree->right->balanceFactor <= 0) { rotateLeft(tree); } else {
rotateRightLeft(tree); } } } // Recursive helper for inorder traversal. template <typename T> void
AVL<T>::printInorder_(const typename BST<T>::BinTree& tree, std::stringstream& ss) const { if (!tree) return; //

```

Base case: If tree is NULL, just return. `printlnorder_(tree->left, ss); // Process left subtree. ss << tree->data << " ";`
`// Process current node. printlnorder_(tree->right, ss); // Process right subtree. } // comment`



The error messages you provided indicate that there are issues with binding non-const lvalue references to rvalues in your AVL tree implementation. Specifically, the errors are related to your `add_` and `remove_` functions, where you are trying to modify the `tree` argument, which is a reference to a typename `BST<T>::BinTree`.

The error message is essentially saying that you cannot bind a non-const lvalue reference to an rvalue. This is because an rvalue is a temporary value that cannot be directly modified.

To fix these issues, you should update your `add_` and `remove_` functions to take a non-const reference to a pointer to `BinTreeNode`, which can be modified.

Here's how you can modify these functions:

```
cpp
template <typename T>
void AVL<T>::add_(typename BST<T>::BinTree& tree, const T& value, PathStack& pathNodes) {
    if (!tree) {
        tree = this->makeNode(value);
    } else if (value < tree->data) {
        pathNodes.push(tree);
        add_(tree->left, value, pathNodes);
    } else if (value > tree->data) {
        pathNodes.push(tree);
        add_(tree->right, value, pathNodes);
    } else {
        throw BSTException(BSTException::E_DUPLICATE, "Duplicate value");
    }

    // Update node count and balance factor.
    tree->count = 1 + this->size_(tree->left) + this->size_(tree->right);
    tree->balanceFactor = this->height_(tree->left) - this->height_(tree->right);

    // Ensure the subtree remains balanced.
    balance(tree);
}

template <typename T>
void AVL<T>::remove_(typename BST<T>::BinTree& tree, const T& value, PathStack& pathNodes) {
    if (!tree) {
        throw BSTException(BSTException::E_NOT_FOUND, "Value not found");
    }
    pathNodes.push(tree);

    if (value < tree->data) {
        remove_(tree->left, value, pathNodes);
    } else if (value > tree->data) {
        remove_(tree->right, value, pathNodes);
    } else {
        if (!tree->left || !tree->right) {
            typename BST<T>::BinTree temp = tree;
            tree = tree->left ? tree->left : tree->right;
            delete temp;
        } else { // If both sides are equal.
            typename BST<T>::BinTree temp = tree->left; // Use predecessor.
            while (temp->right) {
                temp = temp->right;
            }
        }
    }
}
```

```

    }
    tree->data = temp->data;
    remove_(tree->left, temp->data, pathNodes);
}

if (!tree) return; // If the tree is now empty, just return.

// Update node count and balance factor.
tree->count = 1 + this->size_(tree->left) + this->size_(tree->right);
tree->balanceFactor = this->height_(tree->left) - this->height_(tree->right);

// Ensure the subtree remains balanced.
balance(tree);
}

```

By taking a non-const reference to `BinTreeNode`, you can modify the tree as needed without encountering the binding error.



```

echo "Compiling..." Compiling... g++ -o out SimpleAllocator.cpp prng.cpp test.cpp -std=c++17 -Wall In file included
from AVL.h:187, from test.cpp:16: AVL.cpp: In instantiation of 'void AVL<T>::add(const T&) [with T = int]':
test.cpp:335:20: required from 'void addInts(AVL<T>&, int, bool, bool, bool) [with T = int]' test.cpp:501:21: required
from here AVL.cpp:12:16: error: 'BST<int>::BinTreeNode* BST<int>::root_' is private within this context 12 |
add_(this->root_, value, pathNodes); | ~~~~~^~~~~ In file included from AVL.h:27, from test.cpp:16:
BST.h:249:13: note: declared private here 249 | BinTree root_; | ^~~~~ In file included from AVL.h:187, from
test.cpp:16: AVL.cpp: In instantiation of 'void AVL<T>::remove(const T&) [with T = int]': test.cpp:414:27: required
from 'void removeInts(AVL<T>&, bool, int, bool, bool) [with T = int]' test.cpp:506:24: required from here
AVL.cpp:20:19: error: 'BST<int>::BinTreeNode* BST<int>::root_' is private within this context 20 | remove_(this-
>root_, value, pathNodes); | ~~~~~^~~~~ In file included from AVL.h:27, from test.cpp:16: BST.h:249:13: note:
declared private here 249 | BinTree root_; | ^~~~~ make: *** [Makefile:15: compile] Error 1

```



The error messages you're seeing indicate that your code is trying to access the private `root_` member of the `BST` class from the `AVL` class. This is a violation of encapsulation principles, as `root_` should be private and not directly accessible from outside the `BST` class.

To fix this issue, you should ensure that any access to the `root_` member of the `BST` class is done through appropriate public methods or functions provided by the `BST` class, rather than directly accessing it. Typically, the `BST` class should provide methods for accessing or modifying its internal structure, such as inserting and removing nodes. This way, the `AVL` tree implementation should use those public methods to interact with the underlying binary search tree, rather than directly accessing its private members.

Here's a general outline of how you can modify your `AVL` class to adhere to encapsulation principles:

1. Update the `BST` class to provide public methods for adding and removing nodes or accessing the root node.
2. Modify the `AVL` class to use these public methods to interact with the `BST` instead of directly accessing private members.

For example, you can modify your `BST` class like this:

```

cpp
// Inside the BST class
public:

```

```
    BinTree root() {
        return root_;
    }

    void insert(const T& value) {
        // Implement your insert logic here.
    }

    void remove(const T& value) {
        // Implement your remove logic here.
    }

private:
    BinTree root_;
```

And then, update your AVL class to use these methods:

```
cpp
// Inside the AVL class
void add(const T& value) {
    PathStack pathNodes;
    this->insert(value); // Use the public insert method of the BST class.
    // Continue with the rest of your AVL logic.
}

void remove(const T& value) {
    PathStack pathNodes;
    this->remove(value); // Use the public remove method of the BST class.
    // Continue with the rest of your AVL logic.
}
```

By following this approach, you ensure that the encapsulation of the BST class is respected, and the AVL class interacts with it through well-defined public interfaces.
