



```
#include "AVL.h" template <typename T> void AVL<T>::updateBalanceFactor(typename BST<T>::BinTree& tree) { if (tree == nullptr)
{ return; // Base case: tree is a null pointer } int leftHeight = BST<T>::treeHeight(tree->left); int rightHeight =
BST<T>::treeHeight(tree->right); // Update the balance factor for the current node tree->balanceFactor = leftHeight - rightHeight; //
Recursively update the balance factor for left and right subtrees // updateBalanceFactor(tree->left); // updateBalanceFactor(tree-
>right); } template <typename T> void AVL<T>::rotateLeft(typename BST<T>::BinTree& tree) { // Perform a left rotation. if (tree ==
nullptr || tree->right == nullptr) return; typename BST<T>::BinTree newRoot = tree->right; tree->right = newRoot->left; newRoot->left
= tree; tree = newRoot; // Update balance factors // updateBalanceFactor(tree); // updateBalanceFactor(newRoot); } template
<typename T> void AVL<T>::rotateRight(typename BST<T>::BinTree& tree) { // Perform a right rotation. if (tree == nullptr || tree->left
== nullptr) return; typename BST<T>::BinTree newRoot = tree->left; tree->left = newRoot->right; newRoot->right = tree; tree =
newRoot; // Update balance factors // updateBalanceFactor(tree); // updateBalanceFactor(newRoot); } template <typename T> void
AVL<T>::rotateLeftRight(typename BST<T>::BinTree& tree) { // Perform a left-right rotation. if (tree == nullptr || tree->left == nullptr)
return; rotateLeft(tree->left); rotateRight(tree); } template <typename T> void AVL<T>::rotateRightLeft(typename BST<T>::BinTree&
tree) { // Perform a right-left rotation. if (tree == nullptr || tree->right == nullptr) return; rotateRight(tree->right); rotateLeft(tree);
} template <typename T> void AVL<T>::balance(PathStack& pathNodes, bool stopAtFirst) { while (!pathNodes.empty()) { auto
currentNode = pathNodes.top(); pathNodes.pop(); // Update balance factor for the current node updateBalanceFactor(currentNode);
if (!isBalanced(currentNode)) { int balance = currentNode->balanceFactor; if (balance > 1) { if (currentNode->left->balanceFactor <
0) { rotateLeftRight(currentNode); } else { rotateRight(currentNode); } } else if (balance < -1) { if (currentNode->right->balanceFactor
> 0) { rotateRightLeft(currentNode); } else { rotateLeft(currentNode); } } } if (stopAtFirst) break; } } template <typename T> void
AVL<T>::balance(typename BST<T>::BinTree& tree) { if (tree == nullptr) { return; // Recursively balance the left and right subtrees
first // Update balance factor for the current node updateBalanceFactor(tree); if (tree->balanceFactor > 1) { // Left subtree is
unbalanced updateBalanceFactor(tree->left); int leftBalance = tree->left->balanceFactor; if (leftBalance >= 0) { // Left-left case
rotateRight(tree); } else { // Left-right case rotateLeftRight(tree); } } else if (tree->balanceFactor < -1) { // Right subtree is unbalanced
updateBalanceFactor(tree->right); int rightBalance = tree->right->balanceFactor; if (rightBalance <= 0) { // Right-right case
rotateLeft(tree); } else { // Right-left case rotateRightLeft(tree); } } balance(tree->left); balance(tree->right); } template <typename T>
void AVL<T>::add(const T& value) { BST<T>::add(value); // Perform the basic insertion typename BST<T>::BinTree rootPtr = this-
>root(); balance(rootPtr); // Rebalance the tree starting from the root } template <typename T> void AVL<T>::remove(const T&
value) { BST<T>::remove(value); typename BST<T>::BinTree rootPtr = this->root(); balance(rootPtr); // Rebalance the tree starting
from the root } template <typename T> bool AVL<T>::isBalanced(const typename BST<T>::BinTree& tree) const { if (tree == nullptr)
return true; int balanceFactor = tree->balanceFactor; return (balanceFactor >= -1 && balanceFactor <= 1) && isBalanced(tree->left)
&& isBalanced(tree->right); } template <typename T> std::stringstream AVL<T>::printInorder() const { std::stringstream ss;
printInorder_(this->root(), ss); return ss; } template <typename T> void AVL<T>::printInorder_(const typename BST<T>::BinTree&
tree, std::stringstream& ss) const { if (tree != nullptr) { printInorder_(tree->left, ss); ss << tree->data << " "; printInorder_(tree->right,
ss); } } // Add any other private methods or helper functions here. this is my avl.cpp function above. #include "BST.h" template
<typename T> BST<T>::BST(SimpleAllocator* allocator) : allocator_(allocator), root_(nullptr) { if (allocator_ == nullptr) { // If no
allocator is provided, create one with default configuration SimpleAllocatorConfig config; //additional changes if needed
//config.useC++MemManager = true; // Use C++ memory manager //config.objectsPerPage = 100; // Adjust as needed // Set other
config options here allocator_ = new SimpleAllocator(sizeof(BinTreeNode), config); isOwnAllocator_ = true; // The BST owns the
allocator } } template <typename T> BST<T>::BST(const BST& rhs) { allocator_ = rhs.allocator_; isOwnAllocator_ = false; // Not
owning the allocator (sharing with the rhs) root_ = nullptr; // Initialize the root // Copy the elements from rhs tree to this tree
copy_(root_, rhs.root_); } template <typename T> BST<T>& BST<T>::operator=(const BST& rhs) { if (this == &rhs) { return *this; //
Return itself if rhs is the same as this tree } // Clean up the current tree clear(); allocator_ = rhs.allocator_; isOwnAllocator_ = false; //
Not owning the allocator (sharing with rhs) root_ = nullptr; // Initialize the root // Copy the elements from rhs tree to this tree
copy_(root_, rhs.root_); return *this; } template <typename T> BST<T>::~BST() { // Destructor: Clear the tree clear(); if
(isOwnAllocator_) { // If this tree owns the allocator, release it delete allocator_; } } template <typename T> const typename
BST<T>::BinTreeNode* BST<T>::operator[](int index) const { return getNode_(root_, index); } template <typename T> void
BST<T>::add(const T& value) noexcept(false) { add_(root_, value); } template <typename T> void
BST<T>::updateBalanceFactor(typename BST<T>::BinTree& tree) { if (tree != nullptr) { return; // Base case: tree
is a null pointer } // Calculate the height of the left and right subtrees int leftHeight = treeHeight(tree->left); int rightHeight =
treeHeight(tree->right); // Update the balance factor for the current node tree->balanceFactor = leftHeight - rightHeight; //
Recursively update the balance factor for left and right subtrees updateBalanceFactor(tree->left); updateBalanceFactor(tree->right);
} } template <typename T> void BST<T>::remove(const T& value) { remove_(root_, value); } template <typename T> void
BST<T>::clear() { while (!isEmpty(root_)) { // Remove all elements in the tree remove_(root_, root_->data); } } template <typename
T> bool BST<T>::find(const T& value, unsigned& compares) const { // Search for a value in the tree and count the number of
comparisons return find_(root_, value, compares); } template <typename T> bool BST<T>::empty() const { // Check if the tree is
empty return isEmpty(root_); } template <typename T> unsigned int BST<T>::size() const { // Get the size of the tree return
size_(root_); } template <typename T> int BST<T>::height() const { // Get the height of the tree return height_(root_); } template
<typename T> typename BST<T>::BinTree BST<T>::root() const { // Get the root of the tree return root_; } template <typename T>
typename BST<T>::BinTree BST<T>::makeNode(const T& value) { if (allocator_ == nullptr) { throw
BSTException(BSTException::E_NO_MEMORY, "Allocator is not set"); } // Allocate memory for the node BinTree node =
static_cast<BinTree>(allocator_->allocate("NodeLabel")); if (node == nullptr) { throw
BSTException(BSTException::E_NO_MEMORY, "Failed to allocate memory for a new node"); } // Construct the node with the
provided value new (node) BinTreeNode(value); return node; } template <typename T> void BST<T>::freeNode(BinTree node) { if
(allocator_ == nullptr) { throw BSTException(BSTException::E_NO_MEMORY, "Allocator is not set"); } // Call the destructor explicitly
node->~BinTreeNode(); // Free the memory allocator_->free(node); } template <typename T> int BST<T>::treeHeight(BinTree tree)
const { if (root_ == nullptr) { return -1; // If the tree is empty, its height is -1 (no nodes) } if (tree == nullptr || (tree->left == nullptr &&
tree->right == nullptr)) { return 0; // If the current node is a leaf or the tree is empty, its height is 0 } // Calculate the height of the left
and right subtrees recursively int leftHeight = treeHeight(tree->left); int rightHeight = treeHeight(tree->right); // Return the maximum
height between the left and right subtrees, plus 1 for the current node return 1 + (leftHeight > rightHeight ? leftHeight : rightHeight);
} template <typename T> void BST<T>::findPredecessor(BinTree tree, BinTree& predecessor) const { if (tree == nullptr) { // If the
current node is nullptr, there is no predecessor. predecessor = nullptr; return; } if (tree->left == nullptr) { // If there is no left child,
there is no predecessor. predecessor = nullptr; return; } // Traverse to the rightmost node in the left subtree to find the predecessor.
tree = tree->left; while (tree->right != nullptr) { tree = tree->right; } predecessor = tree; } template <typename T> void
BST<T>::findSuccessor(BinTree tree, BinTree& successor) const { if (tree == nullptr) { // If the current node is nullptr, there is no
```

```

successor. successor = nullptr; return; } if (tree->right == nullptr) { // If there is no right child, there is no successor. successor =
nullptr; return; } // Traverse to the leftmost node in the right subtree to find the successor. tree = tree->right; while (tree->left !=
nullptr) { tree = tree->left; } successor = tree; } template <typename T> bool BST<T>::isEmpty(BinTree& tree) const { // Check if a
tree node is empty return tree == nullptr; } template <typename T> bool BST<T>::isEmpty(const BinTree& tree) const { // Check if a
tree node is empty (const version) return tree == nullptr; } template <typename T> bool BST<T>::isLeaf(const BinTree& tree) const {
// Check if a tree node is a leaf node return tree != nullptr && tree->left == nullptr && tree->right == nullptr; } template <typename T>
void BST<T>::add_(BinTree& tree, const T& value) { if (tree == nullptr) // Base case: Insert a new node { tree = makeNode(value);
tree->count = 1; // The current node has one element tree->balanceFactor = 0; // Set balance factor to 0 return; } // Recursive case:
Traverse the tree if (value < tree->data) { add_(tree->left, value); } else if (value > tree->data) { add_(tree->right, value); } else {
throw BSTException(BSTException::E_DUPLICATE, "Duplicate value not allowed"); } if (tree != nullptr) { // Update the count for the
current node tree->count = size_(tree->left) + size_(tree->right) + 1; // Update balance factor } } template <typename T> bool
BST<T>::find_(const BinTree& tree, const T& value, unsigned& compares) const { if (tree == nullptr) { // If the current node is nullptr,
the value is not found. return false; } compares++; if (value == tree->data) { // If the value matches the current node's data, it is
found. return true; } else if (value < tree->data) { // If the value is less than the current node's data, search in the left subtree. return
find_(tree->left, value, compares); } else { // If the value is greater than the current node's data, search in the right subtree. return
find_(tree->right, value, compares); } } template <typename T> const typename BST<T>::BinTree BST<T>::getNode_(const
BinTree& tree, int index) const { if (tree == nullptr) { // If the current node is nullptr, the index is out of bounds. throw
BSTException(BSTException::E_OUT_BOUNDS, "Index out of bounds"); } int leftSize = size_(tree->left); if (index < leftSize) { // If
the index is within the left subtree, recursively search in the left subtree. return getNode_(tree->left, index); } else if (index ==
leftSize) { // If the index matches the size of the left subtree, return the current node. return tree; } else { // If the index is in the right
subtree, adjust the index and search in the right subtree. return getNode_(tree->right, index - leftSize - 1); } } template <typename
T> unsigned BST<T>::size_(const BinTree& tree) const { if (tree == nullptr) { // If the current node is nullptr, it has no elements, so
the size is 0. return 0; } return tree->count; } template <typename T> void BST<T>::remove_(BinTree& tree, const T& value) { if (tree
== nullptr) { // If the current node is nullptr, the value to remove is not found in the tree. throw
BSTException(BSTException::E_NOT_FOUND, "Value to remove not found in the tree"); } if (value < tree->data) { // If the value is
less than the current node's data, continue the search in the left subtree. remove_(tree->left, value); } else if (value > tree->data) { //
If the value is greater than the current node's data, continue the search in the right subtree. remove_(tree->right, value); } else { if
(tree->left == nullptr) { // Case 1: No left child or both children are nullptr // Replace the current node with its right child. BinTree temp
= tree->right; freeNode(tree); tree = temp; } else if (tree->right == nullptr) { // Case 2: No right child // Replace the current node with
its left child. BinTree temp = tree->left; freeNode(tree); tree = temp; } else { // Case 3: Node has two children // Find the predecessor
node from the left subtree, replace the current node with it, // and remove the predecessor node. BinTree predecessor = nullptr;
findPredecessor(tree, predecessor); tree->data = predecessor->data; remove_(tree->left, predecessor->data); } } if (tree != nullptr) {
// Update the count to reflect the number of elements in the current node's subtree. tree->count = size_(tree->left) + size_(tree-
>right) + 1; // Update balance factor updateBalanceFactor(tree); } } template <typename T> int BST<T>::height_(const BinTree&
tree) const { return treeHeight(tree); } template <typename T> void BST<T>::copy_(BinTree& tree, const BinTree& rtree) { if (rtree
== nullptr) { // If the right tree is nullptr, there's nothing to copy. return; } // Create a new node in the current tree with the same data
as the right tree. tree = makeNode(rtree->data); tree->count = rtree->count; // Recursively copy the left and right subtrees.
copy_(tree->left, rtree->left); copy_(tree->right, rtree->right); } currently, im facing issues wit the balancing during addition and
removal

```



It seems like you are implementing an AVL tree in C++. AVL trees require balancing to maintain their properties, and from your code, it appears that you have some issues with balancing during addition and removal. I'll provide some guidance to help you identify and address these issues.

1. Balancing During Addition:

In the add function, you should update the balance factor of nodes while performing the addition. You are currently missing this step. You need to call `updateBalanceFactor` after adding a new node. Modify your add function as follows:

```

cpp
template <typename T>
void AVL<T>::add(const T& value) {
    BST<T>::add(value); // Perform the basic insertion
    typename BST<T>::BinTree rootPtr = this->root();
    balance(rootPtr); // Rebalance the tree starting from the root
    // Note: You should update balance factors while adding, not just at the end.
}

```

Make sure to update the balance factors for all nodes along the path from the inserted node to the root.

2. Balancing During Removal:

In the remove function, similar to addition, you should update the balance factors while removing a node. You are missing the step to update balance factors in your `remove_` function. Modify your `remove_` function to update balance factors as follows:

```

cpp
template <typename T>
void AVL<T>::remove_(typename BST<T>::BinTree& tree, const T& value) {

```

```

if (tree == nullptr) {
    // If the current node is nullptr, the value to remove is not found in the tree.
    throw BSTException(BSTException::E_NOT_FOUND, "Value to remove not found in the tree");
}

if (value < tree->data) {
    // If the value is less than the current node's data, continue the search in the left subtree.
    remove_(tree->left, value);
} else if (value > tree->data) {
    // If the value is greater than the current node's data, continue the search in the right subtree.
    remove_(tree->right, value);
} else {
    if (tree->left == nullptr) {
        // Case 1: No left child or both children are nullptr
        // Replace the current node with its right child.
        typename BST<T>::BinTree temp = tree->right;
        freeNode(tree);
        tree = temp;
    } else if (tree->right == nullptr) {
        // Case 2: No right child
        // Replace the current node with its left child.
        typename BST<T>::BinTree temp = tree->left;
        freeNode(tree);
        tree = temp;
    } else {
        // Case 3: Node has two children
        // Find the predecessor node from the left subtree, replace the current node with it,
        // and remove the predecessor node.
        typename BST<T>::BinTree predecessor = nullptr;
        findPredecessor(tree, predecessor);
        tree->data = predecessor->data;
        remove_(tree->left, predecessor->data);
    }
}

// Update the count to reflect the number of elements in the current node's subtree.
tree->count = size_(tree->left) + size_(tree->right) + 1;
// Update balance factor after removing a node
updateBalanceFactor(tree);
}

```

By updating balance factors during addition and removal, you should be able to maintain the balance of the AVL tree. Ensure that you are calling `updateBalanceFactor` correctly and that your rotations are working as expected to achieve a balanced AVL tree.



You:

```

#include "BST.h" template <typename T> BST<T>::BST(SimpleAllocator* allocator) : allocator_(allocator), root_(nullptr) { if
(allocator_ == nullptr) { // If no allocator is provided, create one with default configuration SimpleAllocatorConfig config; //additional
changes if needed //config.useCPPMemManager = true; // Use C++ memory manager //config.objectsPerPage = 100; // Adjust as
needed // Set other config options here allocator_ = new SimpleAllocator(sizeof(BinTreeNode), config); isOwnAllocator_ = true; //
The BST owns the allocator } } template <typename T> BST<T>::BST(const BST& rhs) { allocator_ = rhs.allocator_;
isOwnAllocator_ = false; // Not owning the allocator (sharing with the rhs) root_ = nullptr; // Initialize the root // Copy the elements
from rhs tree to this tree copy_(root_, rhs.root_); } template <typename T> BST<T>& BST<T>::operator=(const BST& rhs) { if (this
== &rhs) { return *this; // Return itself if rhs is the same as this tree } // Clean up the current tree clear(); allocator_ = rhs.allocator_;
isOwnAllocator_ = false; // Not owning the allocator (sharing with rhs) root_ = nullptr; // Initialize the root // Copy the elements from
rhs tree to this tree copy_(root_, rhs.root_); return *this; } template <typename T> BST<T>::~~BST() { // Destructor: Clear the tree
clear(); if (isOwnAllocator_) { // If this tree owns the allocator, release it delete allocator_; } } template <typename T> const typename
BST<T>::BinTreeNode* BST<T>::operator[](int index) const { return getNode_(root_, index); } template <typename T> void
BST<T>::add(const T& value) noexcept(false) { add_(root_, value); } template <typename T> void
BST<T>::updateBalanceFactor(typename BST<T>::BinTree& tree) { if (tree != nullptr) { return; // Base case: tree
is a null pointer } // Calculate the height of the left and right subtrees int leftHeight = treeHeight(tree->left); int rightHeight =
treeHeight(tree->right); // Update the balance factor for the current node tree->balanceFactor = leftHeight - rightHeight; //
Recursively update the balance factor for left and right subtrees updateBalanceFactor(tree->left); updateBalanceFactor(tree->right);
} } template <typename T> void BST<T>::remove(const T& value) { remove_(root_, value); } template <typename T> void
BST<T>::clear() { while (!isEmpty(root_)) { // Remove all elements in the tree remove_(root_, root_->data); } } template <typename
T> bool BST<T>::find(const T& value, unsigned& compares) const { // Search for a value in the tree and count the number of
comparisons return find_(root_, value, compares); } template <typename T> bool BST<T>::empty() const { // Check if the tree is
empty return isEmpty(root_); } template <typename T> unsigned int BST<T>::size() const { // Get the size of the tree return
size_(root_); } template <typename T> int BST<T>::height() const { // Get the height of the tree return height_(root_); } template
<typename T> typename BST<T>::BinTree BST<T>::root() const { // Get the root of the tree return root_; } template <typename T>

```

```

typename BST<T>::BinTree BST<T>::makeNode(const T& value) { if (allocator_ == nullptr) { throw
BSTException(BSTException::E_NO_MEMORY, "Allocator is not set"); } // Allocate memory for the node BinTree node =
static_cast<BinTree>(allocator_ ->allocate("NodeLabel")); if (node == nullptr) { throw
BSTException(BSTException::E_NO_MEMORY, "Failed to allocate memory for a new node"); } // Construct the node with the
provided value new (node) BinTreeNode(value); return node; } template <typename T> void BST<T>::freeNode(BinTree node) { if
(allocator_ == nullptr) { throw BSTException(BSTException::E_NO_MEMORY, "Allocator is not set"); } // Call the destructor explicitly
node->~BinTreeNode(); // Free the memory allocator_ ->free(node); } template <typename T> int BST<T>::treeHeight(BinTree tree)
const { if (root_ == nullptr) { return -1; // If the tree is empty, its height is -1 (no nodes) } if (tree == nullptr || (tree->left == nullptr &&
tree->right == nullptr)) { return 0; // If the current node is a leaf or the tree is empty, its height is 0 } // Calculate the height of the left
and right subtrees recursively int leftHeight = treeHeight(tree->left); int rightHeight = treeHeight(tree->right); // Return the maximum
height between the left and right subtrees, plus 1 for the current node return 1 + (leftHeight > rightHeight ? leftHeight : rightHeight); }
template <typename T> void BST<T>::findPredecessor(BinTree tree, BinTree& predecessor) const { if (tree == nullptr) { // If the
current node is nullptr, there is no predecessor. predecessor = nullptr; return; } if (tree->left == nullptr) { // If there is no left child,
there is no predecessor. predecessor = nullptr; return; } // Traverse to the rightmost node in the left subtree to find the predecessor.
tree = tree->left; while (tree->right != nullptr) { tree = tree->right; } predecessor = tree; } template <typename T> void
BST<T>::findSuccessor(BinTree tree, BinTree& successor) const { if (tree == nullptr) { // If the current node is nullptr, there is no
successor. successor = nullptr; return; } if (tree->right == nullptr) { // If there is no right child, there is no successor. successor =
nullptr; return; } // Traverse to the leftmost node in the right subtree to find the successor. tree = tree->right; while (tree->left !=
nullptr) { tree = tree->left; } successor = tree; } template <typename T> bool BST<T>::isEmpty(BinTree& tree) const { // Check if a
tree node is empty return tree == nullptr; } template <typename T> bool BST<T>::isEmpty(const BinTree& tree) const { // Check if a
tree node is empty (const version) return tree == nullptr; } template <typename T> bool BST<T>::isLeaf(const BinTree& tree) const {
// Check if a tree node is a leaf node return tree != nullptr && tree->left == nullptr && tree->right == nullptr; } template <typename T>
void BST<T>::add_(BinTree& tree, const T& value) { if (tree == nullptr) // Base case: Insert a new node { tree = makeNode(value);
tree->count = 1; // The current node has one element tree->balanceFactor = 0; // Set balance factor to 0 return; } // Recursive case:
Traverse the tree if (value < tree->data) { add_(tree->left, value); } else if (value > tree->data) { add_(tree->right, value); } else {
throw BSTException(BSTException::E_DUPLICATE, "Duplicate value not allowed"); } if (tree != nullptr) { // Update the count for the
current node tree->count = size_(tree->left) + size_(tree->right) + 1; // Update balance factor updateBalanceFactor(tree); } }
template <typename T> void BST<T>::remove_(BinTree& tree, const T& value) { if (tree == nullptr) { // If the current node is nullptr,
the value to remove is not found in the tree. throw BSTException(BSTException::E_NOT_FOUND, "Value to remove not found in
the tree"); } if (value < tree->data) { // If the value is less than the current node's data, continue the search in the left subtree.
remove_(tree->left, value); } else if (value > tree->data) { // If the value is greater than the current node's data, continue the search in
the right subtree. remove_(tree->right, value); } else { if (tree->left == nullptr) { // Case 1: No left child or both children are nullptr //
Replace the current node with its right child. BinTree temp = tree->right; freeNode(tree); tree = temp; } else if (tree->right == nullptr) {
// Case 2: No right child // Replace the current node with its left child. BinTree temp = tree->left; freeNode(tree); tree = temp; } else {
// Case 3: Node has two children // Find the predecessor node from the left subtree, replace the current node with it, // and remove
the predecessor node. BinTree predecessor = nullptr; findPredecessor(tree, predecessor); tree->data = predecessor->data;
remove_(tree->left, predecessor->data); } } if (tree != nullptr) { // Update the count to reflect the number of elements in the current
node's subtree. tree->count = size_(tree->left) + size_(tree->right) + 1; // Update balance factor updateBalanceFactor(tree); } }
template <typename T> bool BST<T>::find_(const BinTree& tree, const T& value, unsigned& compares) const { if (tree == nullptr) {
// If the current node is nullptr, the value is not found. return false; } compares++; if (value == tree->data) { // If the value matches the
current node's data, it is found. return true; } else if (value < tree->data) { // If the value is less than the current node's data, search in
the left subtree. return find_(tree->left, value, compares); } else { // If the value is greater than the current node's data, search in the
right subtree. return find_(tree->right, value, compares); } } template <typename T> const typename BST<T>::BinTree
BST<T>::getNode_(const BinTree& tree, int index) const { if (tree == nullptr) { // If the current node is nullptr, the index is out of
bounds. throw BSTException(BSTException::E_OUT_BOUNDS, "Index out of bounds"); } int leftSize = size_(tree->left); if (index <
leftSize) { // If the index is within the left subtree, recursively search in the left subtree. return getNode_(tree->left, index); } else if
(index == leftSize) { // If the index matches the size of the left subtree, return the current node. return tree; } else { // If the index is in
the right subtree, adjust the index and search in the right subtree. return getNode_(tree->right, index - leftSize - 1); } } template
<typename T> unsigned BST<T>::size_(const BinTree& tree) const { if (tree == nullptr) { // If the current node is nullptr, it has no
elements, so the size is 0. return 0; } return tree->count; } template <typename T> int BST<T>::height_(const BinTree& tree) const {
return treeHeight(tree); } template <typename T> void BST<T>::copy_(BinTree& tree, const BinTree& rtree) { if (rtree == nullptr) { //
If the right tree is nullptr, there's nothing to copy. return; } // Create a new node in the current tree with the same data as the right
tree. tree = makeNode(rtree->data); tree->count = rtree->count; // Recursively copy the left and right subtrees. copy_(tree->left,
rtree->left); copy_(tree->right, rtree->right); } Can you add the relevant function for balancing directly into bst.cpp

```



ChatGPT:

Certainly, you can add the relevant balancing function to your `bst.cpp` file. Here's an example of how you can integrate the AVL balancing logic into your `bst.cpp` file for the `BST` class:

```

cpp
#include "BST.h"

template <typename T>
void BST<T>::balance(BinTree& tree) {
    if (tree == nullptr) {
        return;
    }

    int balance = tree->balanceFactor;

```



```

    if (balance > 1) {
        if (tree->left->balanceFactor < 0) {
            // Left-right case
            rotateLeftRight(tree);
        } else {
            // Left-left case
            rotateRight(tree);
        }
    } else if (balance < -1) {
        if (tree->right->balanceFactor > 0) {
            // Right-left case
            rotateRightLeft(tree);
        } else {
            // Right-right case
            rotateLeft(tree);
        }
    }

    // Recursively balance the left and right subtrees
    balance(tree->left);
    balance(tree->right);
}

template <typename T>
void BST<T>::add(const T& value) noexcept(false) {
    add_(root_, value);
    updateBalanceFactor(root_); // Update balance factors after insertion
    balance(root_); // Rebalance the tree starting from the root
}

template <typename T>
void BST<T>::remove(const T& value) {
    remove_(root_, value);
    updateBalanceFactor(root_); // Update balance factors after removal
    balance(root_); // Rebalance the tree starting from the root
}

```

In this example, the balance function is added to the BST class in `bst.cpp`. After inserting or removing nodes, you call `updateBalanceFactor` to update balance factors, and then call `balance` to perform AVL tree balancing if necessary. This way, the balancing logic is directly integrated into the `bst.cpp` file.

Make sure to update your header file (`BST.h`) to declare the balance function as well.
