



i will give u 4 files, bst.h, bst.cpp, avl.h, avl.cpp. the goal for now is to first implement the add function for the avl.cpp. the bst.h and bst.cpp and avl.h has all been implemented only the avl.cpp remains BST.h

```

/** * @file BST.h * @author Chek * @brief BST class definition
* Note that this file is lengthy * as it tries to contain as much information as possible * for students to understand the requirements
* @date 12 Sep 2023 */
#ifndef BST_H
#define BST_H
#include "SimpleAllocator.h" // to use your SimpleAllocator
#include <stdexcept>
#include <string>
/** * @class BSTException * @brief Exception class for BST
* Derived from std::exception
* It is used to throw exceptions when an error occurs * in the BST class */
class BSTException : public std::exception {
public:
    BSTException(int ErrCode, const std::string& Message) : error_code_(ErrCode), message_(Message) {}
    enum BST_EXCEPTION { E_OUT_BOUNDS, E_DUPLICATE, E_NO_MEMORY, E_NOT_FOUND };
    virtual int code() const { return error_code_; }
    virtual const char* what() const throw() { return message_.c_str(); }
    virtual ~BSTException() throw() {}
private:
    int error_code_;
    std::string message_;
};
/** * @class BST * @brief Binary Search Tree class
* It is a template class * It is implemented using a linked structure
* It is not a balanced tree */
template <typename T> class BST {
public:
    /** * @struct BinTreeNode * @brief A node in the Binary Tree
    * - didn't name it BSTNode as the definition here is generic */
    struct BinTreeNode {
        // left and right child pointers
        BinTreeNode* left;
        BinTreeNode* right;
        // the data stored in the node
        T data;
        // cache the number of nodes in the subtree rooted at this node
        unsigned count;
        // balance factor is used to determine if the tree is balanced
        // TODO: this is a suggestion when you want to implement some form // of balancing in order to improve the performance in the // last test case, e.g., implementing AVL trees
        // int balanceFactor;
        // default constructor
        BinTreeNode() : left(0), right(0), data(0), count(0) {}
        // constructor with data
        BinTreeNode(const T& value) : left(0), right(0), data(value), count(0) {}
        // balanceFactor(0) {}
    };
    typedef BinTreeNode* BinTree;
    // BinTree is a pointer to BinTreeNode
    /** * @brief Default constructor
    * @param allocator The allocator to be used
    * BST(SimpleAllocator* allocator = nullptr);
    /** * @brief Copy constructor
    * @param rhs The BST to be copied
    * BST(const BST& rhs);
    /** * @brief Assignment operator
    * @param rhs The BST to be copied
    * BST& operator=(const BST& rhs);
    /** * @brief Destructor
    * It calls clear() to free all nodes
    * It is virtual so that the destructor of the derived class * will be called when appropriate
    * virtual ~BST();
    /** * @brief Subscript operator that returns the node at the specified index.
    * It calls getNode_() to do the actual recursive traversal
    * @param index The index of the node to be returned
    * @return The node at the specified index
    * @throw BSTException if the index is out of range
    * TODO change to return BinTree&
    * const BinTreeNode* operator[](int index) const;
    /** * @brief Insert a value into the tree
    * It calls add_() to do the actual recursive insertion
    * It is virtual so that any derived class knows to override it
    * @param value The value to be added
    * @throw BSTException if the value already exists
    * virtual void add(const T& value) noexcept(false);
    /** * @brief Remove a value from the tree
    * It calls remove_() to do the actual recursive removal
    * It is virtual so that any derived class knows to override it
    * @param value The value to be removed
    * @throw BSTException if the value does not exist
    * virtual void remove(const T& value);
    /** * @brief Remove all nodes in the tree
    * void clear();
    /** * @brief Find a value in the tree
    * It calls find_() to do the actual recursive search
    * @param value The value to be found
    * @param compares The number of comparisons made * (a reference to provide as output)
    * @return true if the value is found * false otherwise
    * bool find(const T& value, unsigned& compares) const;
    /** * @brief Check if the tree is empty
    * @return true if the tree is empty * false otherwise
    * bool empty() const;
    /** * @brief Get the number of nodes in the tree
    * @return The number of nodes in the tree
    * unsigned int size() const;
    /** * @brief Get the height of the tree
    * It calls height_() to do the actual recursive calculation
    * @return The height of the tree
    * int height() const;
    /** * @brief Get the root of the tree
    * @return The root of the tree
    * BinTree root() const;
    // Method to get the root node protected:
    /** * @brief Allocate a new node
    * @param value The value to be stored in the new node
    * BinTree makeNode(const T& value);
    /** * @brief Free a node
    * @param node The node to be freed
    * void freeNode(BinTreeNode* node);
    /** * @brief Calculate the height of the tree
    * @param tree The tree to be calculated
    * int treeHeight(BinTree tree) const;
    /** * @brief Find the predecessor of a node
    * @param tree The tree to be searched
    * @param predecessor The predecessor of the node
    * void findPredecessor(BinTree tree, BinTree& predecessor) const;
    void findSuccessor(BinTree tree, BinTree& successor) const;
    /** * @brief Check if the tree is empty
    * @param tree The tree to be checked
    * @return true if the tree is empty * false otherwise
    * bool isEmpty(BinTree& tree) const;
    bool isEmpty(const BinTree& tree) const;
    /** * @brief Check if the tree is a leaf
    * @param tree The tree to be checked
    * @return true if the tree is a leaf * false otherwise
    * bool isLeaf(const BinTree& tree) const;
private:
    // the allocator to be used
    SimpleAllocator* allocator_;
    // whether the allocator is owned by the tree
    bool isOwnAllocator_ = false;
    // the root of the tree
    BinTree root_;
    /** * @brief A recursive step to add a value into the tree
    * @param tree The tree to be added
    * @param value The value to be added
    * void add_(BinTree& tree, const T& value);
    /** * @brief A recursive step to find the value in the tree
    * @param tree The tree to be searched
    * @param value The value to be found
    * @param compares The number of comparisons made * (a reference to provide as output)
    * Note that it is not const as it is used to provide output
    * @return true if the value is found
    * bool find_(const BinTree& tree, const T& value, unsigned& compares) const;
    /** * @brief A recursive step to get to the node at the specified index
    * This is used by operator[]
    * @param tree The tree to be traversed
    * @param index The index of the node to be returned
    * @return The node at the specified index
    * @throw BSTException if the index is out of range
    * const BinTreeNode* getNode_(const BinTree& tree, int index) const;
    /** * @brief A recursive step to calculate the size of the tree
    * @param tree The tree to be calculated
    * @return The size of the tree
    * unsigned size_(const BinTree& tree) const;
    /** * @brief A recursive step to remove a value from the tree
    * @param tree The tree to be removed
    * @param value The value to be removed
    * void remove_(BinTree& tree, const T& value);
    /** * @brief A recursive step to calculate the height of the tree
    * @param tree The tree to be calculated
    * @return The height of the tree
    * int height_(const BinTree& tree) const;
    /** * @brief A recursive step to copy the tree
    * @param tree The tree to be copied
    * @param rtree The tree to be copied to
    * void copy_(BinTree& tree, const BinTree& rtree);
};
// This is the header file but it is including the implementation cpp because // is a templated class. In C++, the compiler needs to know the implementation // of the class in order to compile it. The compiler does not know what type // the class will be used for. Therefore, the implementation is included in // the header file. This is not a good practice as it will increase the // compilation time. However, it is necessary for templated classes. // It will show an err if the cpp file does not exist yet.
#include "BST.cpp"
#endif
bst.cpp
#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;
        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, root_(nullptr) {
    // 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_);
}
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>::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
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; } } 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; } } 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); }

```



It looks like you've provided the implementation for the Binary Search Tree (BST) in the `bst.cpp` file, and you want to implement the add function for an AVL tree in the `avl.cpp` file. To implement the add function for the AVL tree, you'll need to modify the `avl.cpp` file.

Here's an example of how you can implement the add function for an AVL tree:

```
cpp
// avl.cpp
#include "AVL.h" // Include the AVL tree header file

// Implement the add function for the AVL tree
template <typename T>
void AVL<T>::add(const T& value) noexcept(false) {
    add_(root_, value); // Call the add_ function from the base class
    // After adding a node, check and perform any necessary rotations to rebalance the tree
    balance(root_);
}

// Implement the balance function to rebalance the AVL tree
template <typename T>
void AVL<T>::balance(BinTree& tree) {
    if (tree == nullptr) {
        return;
    }

    // Update the height of the current node
    tree->height = 1 + std::max(height_(tree->left), height_(tree->right));

    // Calculate the balance factor for the current node
    int balanceFactor = height_(tree->right) - height_(tree->left);

    // Perform AVL rotations to rebalance the tree
    if (balanceFactor > 1) {
        // Left-heavy subtree
        if (value > tree->right->data) {
            // Right rotation
            rotateRight(tree);
        } else {
            // Double rotation (left-right)
            rotateLeft(tree->right);
            rotateRight(tree);
        }
    } else if (balanceFactor < -1) {
        // Right-heavy subtree
        if (value < tree->left->data) {
            // Left rotation
            rotateLeft(tree);
        } else {
            // Double rotation (right-left)
            rotateRight(tree->left);
            rotateLeft(tree);
        }
    }

    // Update the height of the current node and its children after rotation
    tree->height = 1 + std::max(height_(tree->left), height_(tree->right));
    if (tree->left) {
        tree->left->height = 1 + std::max(height_(tree->left->left), height_(tree->left->right));
    }
    if (tree->right) {
        tree->right->height = 1 + std::max(height_(tree->right->left), height_(tree->right->right));
    }
}

// Recursively balance the parent nodes
balance(tree->left);
balance(tree->right);
}

// Implement the rotateLeft function to perform a left rotation
template <typename T>
void AVL<T>::rotateLeft(BinTree& tree) {
    BinTree newRoot = tree->right;
    tree->right = newRoot->left;
    newRoot->left = tree;
    tree = newRoot;
}
```

```

}

// Implement the rotateRight function to perform a right rotation
template <typename T>
void AVL<T>::rotateRight(BinTree& tree) {
    BinTree newRoot = tree->left;
    tree->left = newRoot->right;
    newRoot->right = tree;
    tree = newRoot;
}

```

This code adds the add function to the AVL tree, which inserts a new node into the tree and then checks and performs any necessary rotations to rebalance the tree according to AVL tree properties.

Please make sure that you have an AVL.h header file that defines the AVL class and includes the necessary declarations for the add function and other AVL-specific operations.



You:

```

/** @file AVL.h * @author Chek * @brief This file contains the AVL tree class declaration * @date 27 Oct 2023 */ #ifndef AVL_H
#define AVL_H #include <iostream> #include <sstream> #include <stack> #include "SimpleAllocator.h" /** * @brief AVL tree class
* Here, the suggested approach is to inherit from the BST class * you have already completed in the DSA-BST assignment. *
However, it is up to you how you implement this class, as long as * the interface works as expected in test.cpp * If you choose not
to inherit from BST, you can remove all the * inheritance code, e.g., `: public BST<T>` from the class definition, *
`virtual`/`override` modifiers from the inherited methods, etc. * @tparam T Type of data to be stored in the tree */ #include "BST.h"
template <typename T> class AVL : public BST<T> { public: // This is a convenience type definition for the stack of path nodes. //
It is used to trace back to the unbalanced node(s) after adding/removing, // as shown in class. Feel free to ignore this and use
other techniques if you wish. using PathStack = std::stack<typename BST<T>::BinTree>; /** * @brief Constructor. * The inline
implementation here calls the BST constructor. * However, you can implement this from scratch if you wish. * @param allocator
Pointer to the allocator to use for the tree. */ AVL(SimpleAllocator* allocator = nullptr) : BST<T>(allocator) {} /** * @brief
Destructor * The inline implementation here calls the BST destructor. * However, you can implement this from scratch if you wish.
*/ virtual ~AVL() override {} /** * @brief Add a new value to the tree and balance the tree. * Use the techniques discussed in class
to trace back to the * first unbalanced node and perform the appropriate rotations. * Feel free to use other techniques if you wish
but make sure your * outputs match the expected ones. * (The declaration assumes you will override the remove method from
BST. * However, you can remove the inheritance association if you wish.) * @param value to be added to the tree * @throw
BSTException if the value already exists in the tree */ virtual void add(const T& value) override; /** * @brief Remove a value from
the tree and balance the tree. * Use the techniques discussed in class to trace back to ALL * unbalanced nodes and perform the
appropriate rotations. * Feel free to use other techniques if you wish but make sure your * outputs match the expected ones. *
(The declaration assumes you will override the remove method from BST. * However, you can remove the inheritance association
if you wish.) * @param value to be removed from the tree * @throw BSTException if the value does not exist in the tree */ virtual
void remove(const T& value) override; /** * @brief Print the inorder traversal of the tree. * @return stringstream containing the
inorder traversal of the tree */ std::stringstream printInorder() const; /** * @brief Get the height of the tree. * This is mainly used to
update the cached balance factors in the nodes. * It is also called in test.cpp to verify correctness. * (The inline implementation
defaults to the BST height method. * However, you can implement this from scratch if you wish.) * @return height of the tree */ int
height() const { return BST<T>::height(); } /** * @brief Get the size of the tree. * This is mainly used to update the cached counts
in the nodes. * It is also called in test.cpp to verify correctness. * (The inline implementation defaults to the BST size method. *
However, you can implement this from scratch if you wish.) * @return size of the tree */ unsigned size() const { return
BST<T>::size(); } private: // TODO: Add any private methods or data members you need here. // For the public interface above,
as mentioned, the main requirement // is that the interface works as expected in test.cpp. /** * @brief Rotate the tree to the left. *
Note that you may need to update the counts and balance factors. * @param tree to be rotated */ void rotateLeft(typename
BST<T>::BinTree& tree); /** * @brief Rotate the tree to the right. * Note that you may need to update the counts and balance
factors. * @param tree to be rotated */ void rotateRight(typename BST<T>::BinTree& tree); /** * @brief Rotate the left subtree to
the left, then the whole tree to the right. * You may want to use the rotateLeft and rotateRight methods above. * @param tree to
be rotated */ void rotateLeftRight(typename BST<T>::BinTree& tree); /** * @brief Rotate the right subtree to the right, then the
whole tree to the left. * You may want to use the rotateLeft and rotateRight methods above. * @param tree to be rotated */ void
rotateRightLeft(typename BST<T>::BinTree& tree); /** * @brief Balance the tree from the stack of path nodes. * @param
pathNodes contain the stack of nodes from the root to the node just before * the added/removed node after adding/remove
operation * @param stopAtFirst if true, stop after fixing the first unbalanced node */ void balance(PathStack& pathNodes, bool
stopAtFirst = false); /** * @brief Balance the tree from the given node. * This checks the balance factors of the node and its'
children to * determine which of LL, LR, RR, RL case they are and call the * appropriate rotation methods above. * @param tree
to be balanced */ void balance(typename BST<T>::BinTree& tree); // TODO: Again, you do not need to stick to the private
methods above, // and likely you will need to add more of your own methods in order // to make your code more readable and
maintainable. void printInorder_(const typename BST<T>::BinTree& tree, std::stringstream& ss) const; void add_(typename
BST<T>::BinTree& tree, const T& value, PathStack& pathNodes); void remove_(typename BST<T>::BinTree& tree, const T&
value, PathStack& pathNodes); bool isBalanced(const typename BST<T>::BinTree& tree) const; void
rotateLeftWithStatsUpdate(typename BST<T>::BinTree& tree); void rotateRightWithStatsUpdate(typename BST<T>::BinTree&
tree); }; #include "AVL.cpp" #endif // AVL_H heres my avl.h and here my avl.cpp that i have drafted out #include "AVL.h" template
<typename T> void AVL<T>::rotateLeft(typename BST<T>::BinTree& tree) { // Perform a left rotation. } template <typename T>
void AVL<T>::rotateRight(typename BST<T>::BinTree& tree) { // Perform a right rotation. } template <typename T> void
AVL<T>::rotateLeftRight(typename BST<T>::BinTree& tree) { // Perform a left-right rotation. } template <typename T> void
AVL<T>::rotateRightLeft(typename BST<T>::BinTree& tree) { // Perform a right-left rotation. } template <typename T> void

```



```

AVL<T>::balance(PathStack& pathNodes, bool stopAtFirst) { while (!pathNodes.empty()) { auto currentNode = pathNodes.top();
pathNodes.pop(); if (!isBalanced(currentNode)) { if (currentNode->balanceFactor() > 1) { if (currentNode->left->balanceFactor() <
0) { rotateLeftRight(currentNode); } else { rotateRight(currentNode); } } else if (currentNode->balanceFactor() < -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) { // Balance the
current node. int balanceFactor = tree->balanceFactor(); if (balanceFactor > 1) { if (tree->left->balanceFactor() < 0) {
rotateLeftRight(tree); } else { rotateRight(tree); } } } else if (balanceFactor < -1) { if (tree->right->balanceFactor() > 0) {
rotateRightLeft(tree); } else { rotateLeft(tree); } } } } 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); } } template <typename T> void AVL<T>::add_(typename BST<T>::BinTree& tree, const T& value,
PathStack& pathNodes) { if (tree == nullptr) { tree = std::make_shared<typename BST<T>::BinTreeNode>(value);
pathNodes.push(tree); } else { if (value < tree->data) { add_(tree->left, value, pathNodes); } else if (value > tree->data) {
add_(tree->right, value, pathNodes); } } } template <typename T> void AVL<T>::add(const T& value) { PathStack pathNodes;
add_(this->root(), value, pathNodes); balance(pathNodes, true); } template <typename T> void AVL<T>::remove_(typename
BST<T>::BinTree& tree, const T& value, PathStack& pathNodes) { // if (tree == nullptr) { // // Value not found in the tree // throw
BSTException("Value does not exist in the tree"); // } // if (value < tree->data) { // // Recursively search in the left subtree //
remove_(tree->left, value, pathNodes); // } else if (value > tree->data) { // // Recursively search in the right subtree //
remove_(tree->right, value, pathNodes); // } else { // // Found the value to remove // pathNodes.push(tree); // if (tree->left ==
nullptr || tree->right == nullptr) { // // Node with one child or no child // typename BST<T>::BinTree temp = tree->left ? tree->left :
tree->right; // if (temp == nullptr) { // // No child case // temp = tree; // tree = nullptr; // } else { // // One child case // *tree = *temp; //
} // this->allocator->deallocate(temp); // } else { // // Node with two children: replace this node with the in-order successor //
typename BST<T>::BinTree successor = this->getInOrderSuccessor(tree); // tree->data = successor->data; // // Recursively
remove the in-order successor // remove_(tree->right, successor->data, pathNodes); // } // } // // Update balance factor and height
// if (tree != nullptr) { // tree->updateBalanceFactor(); // } } } 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> void
AVL<T>::remove(const T& value) { } template <typename T> std::stringstream AVL<T>::printInorder() const { std::stringstream ss;
printInorder_(this->root(), ss); return ss; } // Add any other private methods or helper functions here. can you, with this info give
me the implementation for add and add_ functions

```



Certainly! Here's the implementation for the add and add_ functions in your AVL tree class:

```

cpp
template <typename T>
void AVL<T>::add(const T& value) {
    PathStack pathNodes;
    add_(this->root(), value, pathNodes);
    balance(pathNodes, true);
}

template <typename T>
void AVL<T>::add_(typename BST<T>::BinTree& tree, const T& value, PathStack& pathNodes) {
    if (tree == nullptr) {
        tree = std::make_shared<typename BST<T>::BinTreeNode>(value);
        pathNodes.push(tree);
    } else if (value < tree->data) {
        add_(tree->left, value, pathNodes);
    } else if (value > tree->data) {
        add_(tree->right, value, pathNodes);
    } else {
        // Value already exists in the tree, so throw an exception.
        throw BSTException("Value already exists in the tree");
    }
    tree->updateBalanceFactor();
    balance(pathNodes);
}

```

In the add_ function, we traverse the tree to find the appropriate position for the new value, and when we find a null node, we create a new node with the given value. We then push the path nodes into the pathNodes stack for later balancing. If we encounter a value that already exists in the tree, we throw a BSTException to indicate that the value cannot be added again. Finally, after adding the value, we update the balance factors and balance the tree using the balance function.



You:

```
echo "Compiling..." Compiling... g++ -o out SimpleAllocator.cpp prng.cpp test.cpp -std=c++17 -Wall In file included from
AVL.h:171, from test.cpp:16: AVL.cpp: In member function 'void AVL<T>::add_(typename BST<T>::BinTree&, const T&,
AVL<T>::PathStack&): AVL.cpp:90:21: error: 'make_shared' is not a member of 'std' 90 | tree = std::make_shared<typename
BST<T>::BinTreeNode>(value); | ^~~~~~ AVL.cpp:2:1: note: 'std::make_shared' is defined in header '<memory>'; did you
forget to '#include <memory>'? 1 | #include "AVL.h" +++ |+#include <memory> 2 | AVL.cpp:90:61: error: expected '(' before '>'
token 90 | tree = std::make_shared<typename BST<T>::BinTreeNode>(value); | ^ | ( AVL.cpp: In instantiation of 'void
AVL<T>::add(const T&) [with T = int]': test.cpp:335:13: required from 'void addInts(AVL<T>&, int, bool, bool, bool) [with T = int]'
test.cpp:501:28: required from here AVL.cpp:83:5: error: cannot bind non-const lvalue reference of type
'BST<int>::BinTreeNode*&' to an rvalue of type 'BST<int>::BinTree' {aka 'BST<int>::BinTreeNode*'} 83 | add_(this->root(), value,
pathNodes); | ^~~~ AVL.cpp:88:45: 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*> > >]' 88 | void
AVL<T>::add_(typename BST<T>::BinTree& tree, const T& value, PathStack& pathNodes) { |
~~~~~ AVL.cpp: In instantiation 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*> > >]':
AVL.cpp:83:5: required from 'void AVL<T>::add(const T&) [with T = int]' test.cpp:335:13: required from 'void addInts(AVL<T>&, int,
bool, bool, bool) [with T = int]' test.cpp:501:28: required from here AVL.cpp:98:15: error: no matching function for call to
'BSTException::BSTException(const char [33])' 98 | throw BSTException("Value already exists in the tree"); |
~~~~~ In file included from AVL.h:27, from test.cpp:16: BST.h:25:5: note:
candidate: 'BSTException::BSTException(int, const string&)' 25 | BSTException(int ErrCode, const std::string& Message) |
~~~~~ BST.h:25:5: note: candidate expects 2 arguments, 1 provided BST.h:23:7: note: candidate:
'BSTException::BSTException(const BSTException&)' 23 | class BSTException : public std::exception { | ^~~~~~
BST.h:23:7: note: no known conversion for argument 1 from 'const char [33]' to 'const BSTException&' In file included from
AVL.h:171, from test.cpp:16: AVL.cpp:100:11: error: 'struct BST<int>::BinTreeNode' has no member named
'updateBalanceFactor' 100 | tree->updateBalanceFactor(); | ~~~~~~ make: *** [Makefile:15: compile]
Error 1 i have recieved these errors
```



ChatGPT:

The errors you're encountering are related to missing `#include` directives for certain headers and issues with the `BSTException` class. Let's address them one by one.

1. **std::make_shared Error:** The error message indicates that `std::make_shared` is not a member of the `std` namespace. This is because you are using C++11 or C++14, which doesn't provide `std::make_shared` directly. You can resolve this by including the `<memory>` header at the beginning of your `AVL.cpp` file:

```
cpp
#include <memory>
```

2. **BSTException Error:** It seems that the `BSTException` class you are trying to throw does not have a constructor that takes a single string as an argument. You need to update your `BSTException` class to accept a string as a message in its constructor. Make sure that `BSTException` is defined like this in `BST.h`:

```
cpp
class BSTException : public std::exception {
public:
    BSTException(const std::string& Message) : message(Message) {}

    virtual const char* what() const throw() {
        return message.c_str();
    }

private:
    std::string message;
};
```

3. **updateBalanceFactor Error:** The error suggests that there's no member function named `updateBalanceFactor` in your `BST` class's `BinTreeNode`. You should make sure that your `BinTreeNode` class has a method to update the balance factor based on the heights of its left and right children. The exact implementation would depend on your data structure and how you're tracking the balance factor.

After making these changes, you should be able to compile your code successfully. If you still encounter issues, please provide additional details, and I'll be happy to assist you further.
