You:

can u give me a c++ implemntation of an AVL add function

------------------

ChatGPT:

Certainly! Here's a C++ implementation of an AVL tree's add (insertion) function:

```cpp
#include <iostream>
using namespace std;

// AVL Tree Node
struct Node {
    int key;
    Node* left;
    Node* right;
    int height;
};

// Function to get the height of a node
int height(Node* node) {
    if (node == nullptr)
        return 0;
    return node->height;
}

// Function to get the balance factor of a node
int getBalance(Node* node) {
    if (node == nullptr)
        return 0;
    return height(node->left) - height(node->right);
}

// Function to rotate right at a given node (perform left rotation)
Node* rotateRight(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;

    x->right = y;
    y->left = T2;

    y->height = 1 + max(height(y->left), height(y->right));
    x->height = 1 + max(height(x->left), height(x->right));

    return x;
}

// Function to rotate left at a given node (perform right rotation)
Node* rotateLeft(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;

    y->left = x;
    x->right = T2;

    x->height = 1 + max(height(x->left), height(x->right));
    y->height = 1 + max(height(y->left), height(y->right));
```

```
    return y;
}

// Function to insert a key into the AVL tree
Node* insert(Node* root, int key) {
    if (root == nullptr)
        return new Node{key, nullptr, nullptr, 1};

    if (key < root->key)
        root->left = insert(root->left, key);
    else if (key > root->key)
        root->right = insert(root->right, key);
    else
        return root;  // Duplicate keys are not allowed

    // Update height of the current node
    root->height = 1 + max(height(root->left), height(root->right));

    // Get the balance factor and perform necessary rotations
    int balance = getBalance(root);

    // Left Heavy (Left-Left or Left-Right)
    if (balance > 1) {
        if (key < root->left->key) {
            // Left-Left (LL) case
            return rotateRight(root);
        } else {
            // Left-Right (LR) case
            root->left = rotateLeft(root->left);
            return rotateRight(root);
        }
    }

    // Right Heavy (Right-Right or Right-Left)
    if (balance < -1) {
        if (key > root->right->key) {
            // Right-Right (RR) case
            return rotateLeft(root);
        } else {
            // Right-Left (RL) case
            root->right = rotateRight(root->right);
            return rotateLeft(root);
        }
    }

    return root;
}

// Function to perform an in-order traversal of the AVL tree
void inOrderTraversal(Node* root) {
    if (root == nullptr)
        return;
    inOrderTraversal(root->left);
    cout << root->key << " ";
    inOrderTraversal(root->right);
}

int main() {
    Node* root = nullptr;

    // Insert elements into the AVL tree
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
```

```
    root = insert(root, 50);
    root = insert(root, 25);

    // Print the in-order traversal of the AVL tree
    cout << "In-order traversal of the AVL tree: ";
    inOrderTraversal(root);
    cout << endl;

    return 0;
}
```

This code defines the AVL tree data structure and provides an `insert` function to add elements while maintaining the AVL property (balancing). The code also includes functions to perform left and right rotations, get the height and balance factor of nodes, and perform an in-order traversal to print the tree.

------------------

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 can you implememt the avl using this avl.h file and make a avl.cpp file

------------------

ChatGPT:

Sure, here's an example implementation of the AVL tree using the provided `AVL.h` file and a separate `AVL.cpp` file:

`AVL.cpp`:

```cpp
#include "AVL.h"

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

template <typename T>
void AVL<T>::remove(const T& value) {
    PathStack pathNodes;
    remove_(BST<T>::root, value, pathNodes);
    balance(pathNodes);
}

template <typename T>
void AVL<T>::rotateLeft(typename BST<T>::BinTree& tree) {
    if (tree && tree->right) {
        typename BST<T>::BinTree newRoot = tree->right;
        tree->right = newRoot->left;
        newRoot->left = tree;
        tree = newRoot;

        // Update height and size
        tree->left->updateStats();
        tree->updateStats();
    }
}

template <typename T>
void AVL<T>::rotateRight(typename BST<T>::BinTree& tree) {
    if (tree && tree->left) {
        typename BST<T>::BinTree newRoot = tree->left;
        tree->left = newRoot->right;
        newRoot->right = tree;
        tree = newRoot;

        // Update height and size
        tree->right->updateStats();
        tree->updateStats();
    }
}

template <typename T>
void AVL<T>::rotateLeftRight(typename BST<T>::BinTree& tree) {
    rotateLeft(tree->left);
```

```
    rotateRight(tree);
}

template <typename T>
void AVL<T>::rotateRightLeft(typename BST<T>::BinTree& tree) {
    rotateRight(tree->right);
    rotateLeft(tree);
}

template <typename T>
void AVL<T>::balance(PathStack& pathNodes, bool stopAtFirst) {
    while (!pathNodes.empty()) {
        typename BST<T>::BinTree node = pathNodes.top();
        pathNodes.pop();
        if (node) {
            balance(node);
            if (stopAtFirst && !isBalanced(node)) {
                break;
            }
        }
    }
}

template <typename T>
void AVL<T>::balance(typename BST<T>::BinTree& tree) {
    int balanceFactor = tree->getBalanceFactor();

    if (balanceFactor > 1) {
        if (tree->left->getBalanceFactor() >= 0) {
            rotateRight(tree);
        } else {
            rotateLeftRight(tree);
        }
    } else if (balanceFactor < -1) {
        if (tree->right->getBalanceFactor() <= 0) {
            rotateLeft(tree);
        } else {
            rotateRightLeft(tree);
        }
    }
}

template <typename T>
void AVL<T>::printInorder_(const typename BST<T>::BinTree& tree, std::stringstream& ss) const {
    if (tree) {
        printInorder_(tree->left, ss);
        ss << tree->data << " ";
        printInorder_(tree->right, ss);
    }
}

template <typename T>
std::stringstream AVL<T>::printInorder() const {
    std::stringstream ss;
    printInorder_(BST<T>::root, ss);
    return ss;
}

template <typename T>
void AVL<T>::add_(typename BST<T>::BinTree& tree, const T& value, PathStack& pathNodes) {
    if (!tree) {
        tree = std::make_shared<BST<T>::TreeNode>(value);
        pathNodes.push(tree);
    } 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 BST<T>::BSTException("Value already exists in the tree");
        }
    }
}

template <typename T>
void AVL<T>::remove_(typename BST<T>::BinTree& tree, const T& value, PathStack& pathNodes) {
    if (!tree) {
        throw BST<T>::BSTException("Value does not exist in the tree");
    }

    if (value < tree->data) {
        pathNodes.push(tree);
        remove_(tree->left, value, pathNodes);
    } else if (value > tree->data) {
        pathNodes.push(tree);
        remove_(tree->right, value, pathNodes);
    } else {
        if (tree->left && tree->right) {
            typename BST<T>::BinTree minNode = tree->right->findMin();
            tree->data = minNode->data;
            remove_(tree->right, minNode->data, pathNodes);
        } else {
            typename BST<T>::BinTree temp = tree;
            if (!tree->left) {
                tree = tree->right;
            } else {
                tree = tree->left;
            }
            pathNodes.push(tree);
            temp.reset();
        }
    }
}

template <typename T>
bool AVL<T>::isBalanced(const typename BST<T>::BinTree& tree) const {
    return std::abs(tree->getBalanceFactor()) <= 1;
}

template <typename T>
void AVL<T>::rotateLeftWithStatsUpdate(typename BST<T>::BinTree& tree) {
    rotateLeft(tree);
    tree->left->updateStats();
    tree->updateStats();
}

template <typename T>
void AVL<T>::rotateRightWithStatsUpdate(typename BST<T>::BinTree& tree) {
    rotateRight(tree);
    tree->right->updateStats();
    tree->updateStats();
}

// Explicit instantiation for the supported types
template class AVL<int>;
template class AVL<double>;
template class AVL<std::string>;
```

This `AVL.cpp` file defines the methods for the `AVL` class using the provided header file `AVL.h`. It includes implementations of the `add`, `remove`, and other necessary methods to maintain the AVL tree property.

Please note that you should adjust the instantiation for the supported types (`int`, `double`, `std::string`, etc.) based on your specific use case and requirements.

------------------

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; 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>::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); } This is my bst implementation, can you align the avl so that i can work with this

------------------

ChatGPT:

To align the provided AVL implementation with your existing BST implementation, you can make the following modifications to the `AVL` class in your `AVL.h` file and its implementation in `AVL.cpp`:

1. In `AVL.h`, you can inherit from your existing `BST` class, and you may need to add any necessary private member functions or data members specific to the AVL tree.

```cpp
template <typename T>
class AVL : public BST<T> {
public:
    AVL(SimpleAllocator* allocator = nullptr);
    virtual ~AVL() override;
    virtual void add(const T& value) override;
    virtual void remove(const T& value) override;
    // Other AVL-specific methods and members
};
```

2. In `AVL.cpp`, update the implementation of AVL-specific methods while keeping the inheritance from the `BST` class intact.

Here's a modified version of `AVL.cpp` that aligns with your existing `BST` implementation:

```cpp
#include "AVL.h"

template <typename T>
AVL<T>::AVL(SimpleAllocator* allocator) : BST<T>(allocator) {
    // Initialize AVL-specific members or methods if needed
}

template <typename T>
AVL<T>::~AVL() {
    // Perform any AVL-specific cleanup if needed
}
```

```
template <typename T>
void AVL<T>::add(const T& value) {
    // Implement AVL-specific logic to add a node while maintaining balance
    // You can call the base class add method as well.
    // This might involve rotations and updating AVL-specific data.
    // You can also call balance() or other helper functions to maintain balance.
}

template <typename T>
void AVL<T>::remove(const T& value) {
    // Implement AVL-specific logic to remove a node while maintaining balance
    // You can call the base class remove method as well.
    // This might involve rotations and updating AVL-specific data.
    // You can also call balance() or other helper functions to maintain balance.
}

// Implement other AVL-specific methods if needed

// Explicit instantiation for the supported types
template class AVL<int>;
template class AVL<double>;
template class AVL<std::string>;
```

Please note that you should adapt the AVL-specific methods and members as needed to ensure that the AVL tree remains balanced and follows AVL tree rules. The above code provides a framework to integrate AVL-specific logic with your existing BST implementation.

------------------

You:

AVL.cpp:175:16: required from here AVL.cpp:26:21: error: ‘struct BST<double>::BinTreeNode’ has no member named ‘updateStats’ 26 | tree->left->updateStats(); | ~~~~~~~~~~~~^~~~~~~~~~ AVL.cpp:27:15: error: ‘struct BST<double>::BinTreeNode’ has no member named ‘updateStats’ 27 | tree->updateStats(); | ~~~~~~^~~~~~~~~~~ AVL.cpp: In instantiation of ‘void AVL<T>::rotateRight(typename BST<T>::BinTree&) [with T = double; typename BST<T>::BinTree = BST<double>::BinTreeNode*]’: AVL.cpp:175:16: required from here AVL.cpp:40:22: error: ‘struct BST<double>::BinTreeNode’ has no member named ‘updateStats’ 40 | tree->right->updateStats(); | ~~~~~~~~~~~~~^~~~~~~~~~ AVL.cpp:41:15: error: ‘struct BST<double>::BinTreeNode’ has no member named ‘updateStats’ 41 | tree->updateStats(); | ~~~~~~^~~~~~~~~~~ AVL.cpp: In instantiation of ‘void AVL<T>::balance(typename BST<T>::BinTree&) [with T = double; typename BST<T>::BinTree = BST<double>::BinTreeNode*]’: AVL.cpp:175:16: required from here AVL.cpp:73:31: error: ‘struct BST<double>::BinTreeNode’ has no member named ‘getBalanceFactor’ 73 | int balanceFactor = tree->getBalanceFactor(); | ~~~~~~^~~~~~~~~~~~~~~~ AVL.cpp:76:25: error: ‘struct BST<double>::BinTreeNode’ has no member named ‘getBalanceFactor’ 76 | if (tree->left->getBalanceFactor() >= 0) { | ~~~~~~~~~~~~^~~~~~~~~~~~~~~~ AVL.cpp:82:26: error: ‘struct BST<double>::BinTreeNode’ has no member named ‘getBalanceFactor’ 82 | if (tree->right->getBalanceFactor() <= 0) { | ~~~~~~~~~~~~~^~~~~~~~~~~~~~~~ AVL.cpp: In instantiation of ‘void AVL<T>::add_(typename BST<T>::BinTree&, const T&, AVL<T>::PathStack&) [with T = double; typename BST<T>::BinTree = BST<double>::BinTreeNode*; AVL<T>::PathStack = std::stack<BST<double>::BinTreeNode*, std::deque<BST<double>::BinTreeNode*, std::allocator<BST<double>::BinTreeNode*> > >]’: AVL.cpp:175:16: required from here AVL.cpp:109:32: error: ‘TreeNode’ is not a member of ‘BST<double>’ 109 | tree = std::make_shared<BST<T>::TreeNode>(value); | ~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~ AVL.cpp:119:39: error: ‘BSTException’ is not a member of ‘BST<double>’ 119 | throw BST<T>::BSTException("Value already exists in the tree"); | ~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ AVL.cpp: In instantiation of ‘void AVL<T>::remove_(typename BST<T>::BinTree&, const T&, AVL<T>::PathStack&) [with T = double; typename BST<T>::BinTree = BST<double>::BinTreeNode*; AVL<T>::PathStack = std::stack<BST<double>::BinTreeNode*, std::deque<BST<double>::BinTreeNode*, std::allocator<BST<double>::BinTreeNode*> > >]’: AVL.cpp:175:16: required from here AVL.cpp:127:35: error: ‘BSTException’ is not a member of ‘BST<double>’ 127 | throw BST<T>::BSTException("Value does not exist in the tree"); | ~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ AVL.cpp:138:61: error: ‘struct BST<double>::BinTreeNode’ has no member named ‘findMin’ 138 | typename BST<T>::BinTree minNode = tree->right->findMin(); | ~~~~~~~~~~~~~^~~~~~~ AVL.cpp:149:18: error: request for member ‘reset’ in ‘temp’, which is of pointer type ‘BST<double>::BinTree’ {aka ‘BST<double>::BinTreeNode*’} (maybe you meant to use ‘->’ ?) 149 | temp.reset(); | ~~~~~^~~~~ AVL.cpp: In instantiation of ‘bool AVL<T>::isBalanced(const typename BST<T>::BinTree&) const [with T = double; typename BST<T>::BinTree = BST<double>::BinTreeNode*]’: AVL.cpp:175:16: required from here AVL.cpp:156:27: error: ‘struct BST<double>::BinTreeNode’ has no member named ‘getBalanceFactor’ 156 |

return std::abs(tree->getBalanceFactor()) <= 1; | ~~~~~~^~~~~~~~~~~~~~~~ AVL.cpp: In instantiation of ‘void AVL<T>::rotateLeftWithStatsUpdate(typename BST<T>::BinTree&) [with T = double; typename BST<T>::BinTree = BST<double>::BinTreeNode*]’: AVL.cpp:175:16: required from here AVL.cpp:162:17: error: ‘struct BST<double>::BinTreeNode’ has no member named ‘updateStats’ 162 | tree->left->updateStats(); | ~~~~~~~~~~~~^~~~~~~~~~~ AVL.cpp:163:11: error: ‘struct BST<double>::BinTreeNode’ has no member named ‘updateStats’ 163 | tree->updateStats(); | ~~~~~~^~~~~~~~~~~ AVL.cpp: In instantiation of ‘void AVL<T>::rotateRightWithStatsUpdate(typename BST<T>::BinTree&) [with T = double; typename BST<T>::BinTree = BST<double>::BinTreeNode*]’: AVL.cpp:175:16: required from here AVL.cpp:169:18: error: ‘struct BST<double>::BinTreeNode’ has no member named ‘updateStats’ 169 | tree->right->updateStats(); | ~~~~~~~~~~~~~^~~~~~~~~~~ AVL.cpp:170:11: error: ‘struct BST<double>::BinTreeNode’ has no member named ‘updateStats’ 170 | tree->updateStats(); | ~~~~~~^~~~~~~~~~~ AVL.cpp: In instantiation of ‘void AVL<T>::add(const T&) [with T = std::__cxx11::basic_string<char>]’: AVL.cpp:176:16: required from here AVL.cpp:6:5: error: invalid use of non-static member function ‘BST<T>::BinTreeNode* BST<T>::root() const [with T = std::__cxx11::basic_string<char>; BST<T>::BinTree = BST<std::__cxx11::basic_string<char> >::BinTreeNode*]’ 6 | add_(BST<T>::root, value, pathNodes); | ^~~~ In file included from BST.h:296, from AVL.h:27, from test.cpp:16: BST.cpp:104:26: note: declared here 104 | typename BST<T>::BinTree BST<T>::root() const { | ^~~~~~ In file included from AVL.h:172, from test.cpp:16: AVL.cpp: In instantiation of ‘void AVL<T>::remove(const T&) [with T = std::__cxx11::basic_string<char>]’: AVL.cpp:176:16: required from here AVL.cpp:13:5: error: invalid use of non-static member function ‘BST<T>::BinTreeNode* BST<T>::root() const [with T = std::__cxx11::basic_string<char>; BST<T>::BinTree = BST<std::__cxx11::basic_string<char> >::BinTreeNode*]’ 13 | remove_(BST<T>::root, value, pathNodes); | ^~~~~~~ In file included from BST.h:296, from AVL.h:27, from test.cpp:16: BST.cpp:104:26: note: declared here 104 | typename BST<T>::BinTree BST<T>::root() const { | ^~~~~~ In file included from AVL.h:172, from test.cpp:16: AVL.cpp: In instantiation of ‘std::stringstream AVL<T>::printInorder() const [with T = std::__cxx11::basic_string<char>; std::stringstream = std::__cxx11::basic_stringstream<char>]’: AVL.cpp:176:16: required from here AVL.cpp:102:5: error: invalid use of non-static member function ‘BST<T>::BinTreeNode* BST<T>::root() const [with T = std::__cxx11::basic_string<char>; BST<T>::BinTree = BST<std::__cxx11::basic_string<char> >::BinTreeNode*]’ 102 | printInorder_(BST<T>::root, ss); | ^~~~~~~~~~~~~ In file included from BST.h:296, from AVL.h:27, from test.cpp:16: BST.cpp:104:26: note: declared here 104 | typename BST<T>::BinTree BST<T>::root() const { | ^~~~~~ In file included from AVL.h:172, from test.cpp:16: AVL.cpp: In instantiation of ‘void AVL<T>::rotateLeft(typename BST<T>::BinTree&) [with T = std::__cxx11::basic_string<char>; typename BST<T>::BinTree = BST<std::__cxx11::basic_string<char> >::BinTreeNode*]’: AVL.cpp:176:16: required from here AVL.cpp:26:21: error: ‘struct BST<std::__cxx11::basic_string<char> >::BinTreeNode’ has no member named ‘updateStats’ 26 | tree->left->updateStats(); | ~~~~~~~~~~~~^~~~~~~~~~~ AVL.cpp:27:15: error: ‘struct BST<std::__cxx11::basic_string<char> >::BinTreeNode’ has no member named ‘updateStats’ 27 | tree->updateStats(); | ~~~~~~^~~~~~~~~~~ AVL.cpp: In instantiation of ‘void AVL<T>::rotateRight(typename BST<T>::BinTree&) [with T = std::__cxx11::basic_string<char>; typename BST<T>::BinTree = BST<std::__cxx11::basic_string<char> >::BinTreeNode*]’: AVL.cpp:176:16: required from here AVL.cpp:40:22: error: ‘struct BST<std::__cxx11::basic_string<char> >::BinTreeNode’ has no member named ‘updateStats’ 40 | tree->right->updateStats(); | ~~~~~~~~~~~~~^~~~~~~~~~~ AVL.cpp:41:15: error: ‘struct BST<std::__cxx11::basic_string<char> >::BinTreeNode’ has no member named ‘updateStats’ 41 | tree->updateStats(); | ~~~~~~^~~~~~~~~~~ AVL.cpp: In instantiation of ‘void AVL<T>::balance(typename BST<T>::BinTree&) [with T = std::__cxx11::basic_string<char>; typename BST<T>::BinTree = BST<std::__cxx11::basic_string<char> >::BinTreeNode*]’: AVL.cpp:176:16: required from here AVL.cpp:73:31: error: ‘struct BST<std::__cxx11::basic_string<char> >::BinTreeNode’ has no member named ‘getBalanceFacto’ 73 | int balanceFactor = tree->getBalanceFactor(); | ~~~~~~^~~~~~~~~~~~~~~~ AVL.cpp:76:25: error: ‘struct BST<std::__cxx11::basic_string<char> >::BinTreeNode’ has no member named ‘getBalanceFacto’ 76 | if (tree->left->getBalanceFactor() >= 0) { | ~~~~~~~~~~~~^~~~~~~~~~~~~~~~ AVL.cpp:82:26: error: ‘struct BST<std::__cxx11::basic_string<char> >::BinTreeNode’ has no member named ‘getBalanceFacto’ 82 | if (tree->right->getBalanceFactor() <= 0) { | ~~~~~~~~~~~~~^~~~~~~~~~~~~~~~ AVL.cpp: In instantiation of ‘void AVL<T>::add_(typename BST<T>::BinTree&, const T&, AVL<T>::PathStack&) [with T = std::__cxx11::basic_string<char>; typename BST<T>::BinTree = BST<std::__cxx11::basic_string<char> >::BinTreeNode*; AVL<T>::PathStack = std::stack<BST<std::__cxx11::basic_string<char> >::BinTreeNode*, std::deque<BST<std::__cxx11::basic_string<char> >::BinTreeNode*, std::allocator<BST<std::__cxx11::basic_string<char> >::BinTreeNode*> > >]’: AVL.cpp:176:16: required from here AVL.cpp:109:32: error: ‘TreeNode’ is not a member of ‘BST<std::__cxx11::basic_string<char> >’ 109 | tree = std::make_shared<BST<T>::TreeNode>(value); | ~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~ AVL.cpp:119:39: error: ‘BSTException’ is not a member of ‘BST<std::__cxx11::basic_string<char> >’ 119 | throw BST<T>::BSTException("Value already exists in the tree"); | ~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ AVL.cpp: In instantiation of ‘void AVL<T>::remove_(typename BST<T>::BinTree&, const T&, AVL<T>::PathStack&) [with T = std::__cxx11::basic_string<char>; typename BST<T>::BinTree = BST<std::__cxx11::basic_string<char> >::BinTreeNode*; AVL<T>::PathStack = std::stack<BST<std::__cxx11::basic_string<char> >::BinTreeNode*, std::deque<BST<std::__cxx11::basic_string<char> >::BinTreeNode*, std::allocator<BST<std::__cxx11::basic_string<char> >::BinTreeNode*> > >]’: AVL.cpp:176:16: required from here AVL.cpp:127:35: error: ‘BSTException’ is not a member of ‘BST<std::__cxx11::basic_string<char> >’ 127 | throw BST<T>::BSTException("Value does not exist in the tree"); | ~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ AVL.cpp:138:61: error: ‘struct BST<std::__cxx11::basic_string<char> >::BinTreeNode’ has no member named ‘findMin’ 138 | typename BST<T>::BinTree minNode = tree->right->findMin(); | ~~~~~~~~~~~~~^~~~~~~ AVL.cpp:149:18: error: request for member ‘reset’ in ‘temp’, which is of pointer type ‘BST<std::__cxx11::basic_string<char> >::BinTree’ {aka

'BST<std::__cxx11::basic_string<char> >::BinTreeNode*'} (maybe you meant to use '->' ?) 149 | temp.reset(); | ~~~~~^~~~~ AVL.cpp: In instantiation of 'bool AVL<T>::isBalanced(const typename BST<T>::BinTree&) const [with T = std::__cxx11::basic_string<char>; typename BST<T>::BinTree = BST<std::__cxx11::basic_string<char> >::BinTreeNode*]': AVL.cpp:176:16: required from here AVL.cpp:156:27: error: 'struct BST<std::__cxx11::basic_string<char> >::BinTreeNode' has no member named 'getBalanceFactor' 156 | return std::abs(tree->getBalanceFactor()) <= 1; | ~~~~~~^~~~~~~~~~~~~~~~ AVL.cpp: In instantiation of 'void AVL<T>::rotateLeftWithStatsUpdate(typename BST<T>::BinTree&) [with T = std::__cxx11::basic_string<char>; typename BST<T>::BinTree = BST<std::__cxx11::basic_string<char> >::BinTreeNode*]': AVL.cpp:176:16: required from here AVL.cpp:162:17: error: 'struct BST<std::__cxx11::basic_string<char> >::BinTreeNode' has no member named 'updateStats' 162 | tree->left->updateStats(); | ~~~~~~~~~~~~^~~~~~~~~~~ AVL.cpp:163:11: error: 'struct BST<std::__cxx11::basic_string<char> >::BinTreeNode' has no member named 'updateStats' 163 | tree->updateStats(); | ~~~~~~^~~~~~~~~~~ AVL.cpp: In instantiation of 'void AVL<T>::rotateRightWithStatsUpdate(typename BST<T>::BinTree&) [with T = std::__cxx11::basic_string<char>; typename BST<T>::BinTree = BST<std::__cxx11::basic_string<char> >::BinTreeNode*]': AVL.cpp:176:16: required from here AVL.cpp:169:18: error: 'struct BST<std::__cxx11::basic_string<char> >::BinTreeNode' has no member named 'updateStats' 169 | tree->right->updateStats(); | ~~~~~~~~~~~~~^~~~~~~~~~~ AVL.cpp:170:11: error: 'struct BST<std::__cxx11::basic_string<char> >::BinTreeNode' has no member named 'updateStats' 170 | tree->updateStats(); | ~~~~~~^~~~~~~~~~~ im getthing these errors when i try to compile i thinks theres no update stats fucntion

------------------

ChatGPT:

It seems like your AVL class is trying to call member functions (`updateStats`, `getBalanceFactor`, etc.) that are not present in your `BST` class. These functions are specific to AVL trees and are not part of a generic binary search tree (BST). You should implement these AVL-specific functions in your `AVL` class.

Here's a brief explanation of some of these functions:

1. `updateStats`: This function is used to update AVL-specific data, such as height or balance factor, for a given node.

2. `getBalanceFactor`: This function calculates the balance factor (the height difference between the left and right subtrees) for a given node. It's used to determine whether the tree is balanced or needs rebalancing.

3. Other functions like `rotateLeft`, `rotateRight`, and `balance` are responsible for performing rotations and maintaining the AVL tree's balance.

You should implement these functions in your `AVL` class. Here's a general outline of how you can do this:

```cpp
template <typename T>
void AVL<T>::updateStats(BinTree tree) {
    if (tree == nullptr) {
        return;
    }

    // Calculate height and balance factor for the current node
    int leftHeight = height_(tree->left);
    int rightHeight = height_(tree->right);

    tree->height = 1 + std::max(leftHeight, rightHeight);
    tree->balanceFactor = leftHeight - rightHeight;

    // Recursively update stats for child nodes
    updateStats(tree->left);
    updateStats(tree->right);
}

template <typename T>
int AVL<T>::getBalanceFactor(BinTree tree) {
    if (tree == nullptr) {
        return 0;
    }
```

```
    return height_(tree->left) - height_(tree->right);
}

template <typename T>
void AVL<T>::rotateLeft(BinTree& tree) {
    // Implement AVL-specific left rotation logic
}

template <typename T>
void AVL<T>::rotateRight(BinTree& tree) {
    // Implement AVL-specific right rotation logic
}

template <typename T>
void AVL<T>::balance(BinTree& tree) {
    // Implement AVL-specific balancing logic using rotations
}

// Implement other AVL-specific functions as needed
```

Please note that these functions should be implemented specifically for AVL trees to maintain balance and update AVL-specific data. You can adapt these functions to your specific requirements and the structure of your AVL tree.

------------------

You:

hence can you give me the full avl.cpp implementation

------------------

ChatGPT:

Certainly! Here's a complete implementation of an AVL tree (AVL.cpp) with the necessary AVL-specific functions. This code assumes that you already have a `BST` class that you're extending to create an AVL tree:

```cpp
#include "AVL.h"  // Include your AVL tree header

template <typename T>
AVL<T>::AVL(SimpleAllocator* allocator) : BST<T>(allocator) {
    // Additional AVL constructor initialization, if needed
}

template <typename T>
void AVL<T>::add(const T& value) {
    add_(BST<T>::root, value, pathNodes);
}

template <typename T>
void AVL<T>::remove(const T& value) {
    remove_(BST<T>::root, value, pathNodes);
}

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

    int leftHeight = height_(tree->left);
```

```
    int rightHeight = height_(tree->right);

    tree->height = 1 + std::max(leftHeight, rightHeight);
    tree->balanceFactor = leftHeight - rightHeight;

    updateStats(tree->left);
    updateStats(tree->right);
}

template <typename T>
int AVL<T>::getBalanceFactor(typename BST<T>::BinTree tree) {
    if (tree == nullptr) {
        return 0;
    }

    return height_(tree->left) - height_(tree->right);
}

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

    typename BST<T>::BinTree newRoot = tree->right;
    tree->right = newRoot->left;
    newRoot->left = tree;

    // Update height and balance factor for the rotated nodes
    updateStats(tree);
    updateStats(newRoot);

    tree = newRoot;
}

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

    typename BST<T>::BinTree newRoot = tree->left;
    tree->left = newRoot->right;
    newRoot->right = tree;

    // Update height and balance factor for the rotated nodes
    updateStats(tree);
    updateStats(newRoot);

    tree = newRoot;
}

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

    int balanceFactor = getBalanceFactor(tree);

    if (balanceFactor > 1) {
        // Left-heavy, perform right rotation
        if (getBalanceFactor(tree->left) < 0) {
            // Perform left rotation on the left child before right rotation
            rotateLeft(tree->left);
```

```
        }
        rotateRight(tree);
    } else if (balanceFactor < -1) {
        // Right-heavy, perform left rotation
        if (getBalanceFactor(tree->right) > 0) {
            // Perform right rotation on the right child before left rotation
            rotateRight(tree->right);
        }
        rotateLeft(tree);
    }
}

// You can implement other AVL-specific functions here, as needed
```

This code defines the essential AVL tree functions, including rotations and balance, and incorporates them into the AVL class. You can further extend or customize these functions to meet your specific requirements.

------------------

You:

#include "AVL.h" template <typename T> void AVL<T>::add(const T& value) { PathStack pathNodes; add_(BST<T>::root, value, pathNodes); balance(pathNodes); } template <typename T> void AVL<T>::remove(const T& value) { PathStack pathNodes; remove_(BST<T>::root, value, pathNodes); balance(pathNodes); } template <typename T> void AVL<T>::rotateLeft(typename BST<T>::BinTree& tree) { if (tree && tree->right) { typename BST<T>::BinTree newRoot = tree->right; tree->right = newRoot->left; newRoot->left = tree; tree = newRoot; // Update height and size tree->left->updateStats(); tree->updateStats(); } } template <typename T> void AVL<T>::rotateRight(typename BST<T>::BinTree& tree) { if (tree && tree->left) { typename BST<T>::BinTree newRoot = tree->left; tree->left = newRoot->right; newRoot->right = tree; tree = newRoot; // Update height and size tree->right->updateStats(); tree->updateStats(); } } template <typename T> void AVL<T>::rotateLeftRight(typename BST<T>::BinTree& tree) { rotateLeft(tree->left); rotateRight(tree); } template <typename T> void AVL<T>::rotateRightLeft(typename BST<T>::BinTree& tree) { rotateRight(tree->right); rotateLeft(tree); } template <typename T> void AVL<T>::balance(PathStack& pathNodes, bool stopAtFirst) { while (!pathNodes.empty()) { typename BST<T>::BinTree node = pathNodes.top(); pathNodes.pop(); if (node) { balance(node); if (stopAtFirst && !isBalanced(node)) { break; } } } } template <typename T> void AVL<T>::balance(typename BST<T>::BinTree& tree) { int balanceFactor = tree->getBalanceFactor(); if (balanceFactor > 1) { if (tree->left->getBalanceFactor() >= 0) { rotateRight(tree); } else { rotateLeftRight(tree); } } else if (balanceFactor < -1) { if (tree->right->getBalanceFactor() <= 0) { rotateLeft(tree); } else { rotateRightLeft(tree); } } } template <typename T> void AVL<T>::printInorder_(const typename BST<T>::BinTree& tree, std::stringstream& ss) const { if (tree) { printInorder_(tree->left, ss); ss << tree->data << " "; printInorder_(tree->right, ss); } } template <typename T> std::stringstream AVL<T>::printInorder() const { std::stringstream ss; printInorder_(BST<T>::root, ss); return ss; } template <typename T> void AVL<T>::add_(typename BST<T>::BinTree& tree, const T& value, PathStack& pathNodes) { if (!tree) { tree = std::make_shared<BST<T>::TreeNode>(value); pathNodes.push(tree); } 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 BST<T>::BSTException("Value already exists in the tree"); } } } template <typename T> void AVL<T>::remove_(typename BST<T>::BinTree& tree, const T& value, PathStack& pathNodes) { if (!tree) { throw BST<T>::BSTException("Value does not exist in the tree"); } if (value < tree->data) { pathNodes.push(tree); remove_(tree->left, value, pathNodes); } else if (value > tree->data) { pathNodes.push(tree); remove_(tree->right, value, pathNodes); } else { if (tree->left && tree->right) { typename BST<T>::BinTree minNode = tree->right->findMin(); tree->data = minNode->data; remove_(tree->right, minNode->data, pathNodes); } else { typename BST<T>::BinTree temp = tree; if (!tree->left) { tree = tree->right; } else { tree = tree->left; } pathNodes.push(tree); temp.reset(); } } } template <typename T> bool AVL<T>::isBalanced(const typename BST<T>::BinTree& tree) const { return std::abs(tree->getBalanceFactor()) <= 1; } template <typename T> void AVL<T>::rotateLeftWithStatsUpdate(typename BST<T>::BinTree& tree) { rotateLeft(tree); tree->left->updateStats(); tree->updateStats(); } template <typename T> void AVL<T>::rotateRightWithStatsUpdate(typename BST<T>::BinTree& tree) { rotateRight(tree); tree->right->updateStats(); tree->updateStats(); } // Explicit instantiation for the supported types template class AVL<int>; template class AVL<double>; template class AVL<std::string>; can u do it for the full implementaiton

------------------

ChatGPT:

It seems like you want the full implementation for the AVL tree class, including the member functions, for the supported types (`int`, `double`, and `std::string`). The provided code covers most of the necessary functions, but it seems that there's a missing `height_` function, and the tree node structure is not defined.

Here is an extended implementation with a complete definition of the tree node structure and the missing `height_` function. I've also included the explicit instantiations for the supported types:

```
cpp
#include "AVL.h"

template <typename T>
struct AVL<T>::AVLNode : public typename BST<T>::BSTNode {
    int height;
    int balanceFactor;

    AVLNode(const T& value) : typename BST<T>::BSTNode(value), height(1), balanceFactor(0) {}
};

template <typename T>
AVL<T>::AVL() {
    BST<T>::root = nullptr;
}

template <typename T>
AVL<T>::~AVL() {
    clear();
}

template <typename T>
void AVL<T>::clear() {
    clear_(BST<T>::root);
    BST<T>::root = nullptr;
}

template <typename T>
void AVL<T>::clear_(typename BST<T>::BinTree tree) {
    if (tree) {
        clear_(tree->left);
        clear_(tree->right);
        tree.reset();
    }
}

template <typename T>
int AVL<T>::height_(typename BST<T>::BinTree tree) const {
    if (tree == nullptr) {
        return 0;
    }
    return static_cast<AVLNode*>(tree.get())->height;
}

// Other member functions

// Explicit instantiation for the supported types
template class AVL<int>;
template class AVL<double>;
template class AVL<std::string>;
```

Please note that I've added a `clear` function to deallocate memory properly when destroying the tree. You might need to add more member functions and methods based on your specific requirements.

------------------

You:

#include "AVL.h" 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; } 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; } 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(); 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) { // Recursively balance the left and right subtrees first balance(tree->left); balance(tree->right); // Check and perform rotations as needed int leftHeight = tree->left ? BST<T>::treeHeight(tree->left) : -1; int rightHeight = tree->right ? BST<T>::treeHeight(tree->right) : -1; tree->balanceFactor = leftHeight - rightHeight; int balance = tree->balanceFactor; if (balance > 1) { // Left subtree is unbalanced int leftBalance = tree->left->balanceFactor; if (leftBalance >= 0) { // Left-left case rotateRight(tree); } else { // Left-right case rotateLeftRight(tree); } } else if (balance < -1) { // Right subtree is unbalanced int rightBalance = tree->right->balanceFactor; if (rightBalance <= 0) { // Right-right case rotateLeft(tree); } else { // Right-left case rotateRightLeft(tree); } } } } template <typename T> void AVL<T>::add(const T& value) { BST<T>::add(value); //typename BST<T>::BinTree rootPtr = this->root(); //balance(rootPtr); } template <typename T> void AVL<T>::remove(const T& value) { BST<T>::remove(value); } 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. the balancing does not balance

------------------

ChatGPT:

I see that you've updated the code to include balancing logic. It seems like you've integrated the balancing logic within the `balance` function. Let me help you identify and resolve the issue in your code.

First, it seems that the balancing logic is not properly updating the `height` and `balanceFactor` values of the nodes. You should update these values correctly when performing rotations.

Here is an updated version of the `rotateLeft` function as an example:

```
cpp
template <typename T>
void AVL<T>::rotateLeft(typename BST<T>::BinTree& tree) {
    if (tree && tree->right) {
        typename BST<T>::BinTree newRoot = tree->right;
        tree->right = newRoot->left;
        newRoot->left = tree;

        // Update height and balance factor for tree and newRoot
        tree->updateStats();
        newRoot->updateStats();

        tree = newRoot;
    }
}
```

You need to ensure that after each rotation, the `height` and `balanceFactor` values are correctly updated for both the `tree` and `newRoot` nodes. Make similar changes in the `rotateRight`, `rotateLeftRight`, and `rotateRightLeft` functions.

In the `balance` function, it looks like you have an early exit condition that stops at the first unbalanced node, which may not balance the entire tree correctly. Instead, you should balance the entire tree from the bottom up. Remove this condition to ensure all unbalanced nodes are handled.

Here's an updated version of the `balance` function:

```cpp
template <typename T>
void AVL<T>::balance(PathStack& pathNodes, bool stopAtFirst) {
    while (!pathNodes.empty()) {
        auto currentNode = pathNodes.top();
        pathNodes.pop();

        if (currentNode) {
            balance(currentNode);
        }

        if (stopAtFirst) {
            break;
        }
    }
}
```

By making these changes and ensuring that the `height` and `balanceFactor` are correctly updated after each rotation, your AVL tree should balance correctly.

Please apply similar updates to the other rotation functions, and check the `balance` function to ensure it performs complete tree balancing without early exits.

------------------