



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 remove all the inheritance from this function

```



ChatGPT:

Sure, if you want to remove the inheritance from the AVL class and no longer inherit from the BST class, you can make the following changes to the code:

1. Remove the inheritance declaration from the class definition:

```

cpp
template <typename T>
class AVL {
public:
    // ...
};

```

2. Adjust the member function declarations and definitions accordingly, removing any calls to inherited methods from BST:

Here's an example of what the modified AVL class could look like without inheritance:

```

cpp
#ifndef AVL_H
#define AVL_H

```

```

#include <iostream>
#include <sstream>
#include <stack>
#include "SimpleAllocator.h"

template <typename T>
class AVL {
public:
    using PathStack = std::stack<BinTree>;

    AVL(SimpleAllocator* allocator = nullptr);
    ~AVL();

    void add(const T& value);
    void remove(const T& value);
    std::stringstream printInorder() const;
    int height() const;
    unsigned size() const;

private:
    // Private methods and data members

    // ...
};

#endif // AVL_H

```

You would also need to update the member function definitions in the implementation file (e.g., AVL.cpp) accordingly to match the modified class definition.



You:

yea can u give me the full version of this



ChatGPT:

Certainly! Here's the modified AVL class without inheritance, along with the full version of the header file AVL.h:

```

cpp
#ifndef AVL_H
#define AVL_H

#include <iostream>
#include <sstream>
#include <stack>
#include "SimpleAllocator.h"

template <typename T>
class AVL {
public:
    using PathStack = std::stack<BinTree>;

    AVL(SimpleAllocator* allocator = nullptr);
    ~AVL();

    void add(const T& value);
    void remove(const T& value);
    std::stringstream printInorder() const;
    int height() const;
    unsigned size() const;

private:
    struct TreeNode {
        T data;
        TreeNode* left;
        TreeNode* right;
        int height;
    };
};

```

```

    int balanceFactor;

    TreeNode(const T& value) : data(value), left(nullptr), right(nullptr), height(1), balanceFactor(0) {}
};

TreeNode* root;

// Private methods
void rotateLeft(TreeNode*& tree);
void rotateRight(TreeNode*& tree);
void rotateLeftRight(TreeNode*& tree);
void rotateRightLeft(TreeNode*& tree);
void balance(PathStack& pathNodes, bool stopAtFirst = false);
void balance(TreeNode*& tree);
void printInorder(TreeNode* tree, std::stringstream& ss) const;
void add(TreeNode*& tree, const T& value, PathStack& pathNodes);
void remove(TreeNode*& tree, const T& value, PathStack& pathNodes);
bool isBalanced(const TreeNode* tree) const;
void rotateLeftWithStatsUpdate(TreeNode*& tree);
void rotateRightWithStatsUpdate(TreeNode*& tree);
};

#include "AVL.cpp"

#endif // AVL_H

```

Make sure to update the member function definitions in the implementation file (AVL.cpp) to match this modified class definition.



You:

```

/** @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)*, balanceFactor(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(BinTree 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 BinTree& 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 can you use the bst.h as a reference to have a full implementation of avl.h



Certainly, you can use the BST.h as a reference to create a full implementation of AVL.h. Here's the modified AVL.h based on the BST.h structure:

```
cpp
#ifndef AVL_H
#define AVL_H

#include <iostream>
#include <sstream>
#include <stack>
#include "SimpleAllocator.h"
#include <stdexcept>
#include <string>

/**
 * @class AVLException
 * @brief Exception class for AVL
 *      Derived from std::exception
 *      It is used to throw exceptions when an error occurs
 *      in the AVL class
 */
class AVLException : public std::exception {
public:
    AVLException(int ErrCode, const std::string& Message)
        : error_code_(ErrCode), message_(Message) {};

    enum AVL_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 ~AVLException() throw() {}

private:
    int error_code_;
    std::string message_;
};

template <typename T>
class AVL {
public:
    using PathStack = std::stack<BinTree>;

    AVL(SimpleAllocator* allocator = nullptr);
    ~AVL();

    void add(const T& value) noexcept(false);
    void remove(const T& value);

    std::stringstream printInorder() const;
    int height() const;
    unsigned size() const;
```

```
private:
    struct TreeNode {
        T data;
        TreeNode* left;
        TreeNode* right;
        int height;
        int balanceFactor;

        TreeNode(const T& value)
            : data(value), left(nullptr), right(nullptr), height(1), balanceFactor(0) {}
    };

    TreeNode* root;

    void rotateLeft(TreeNode*& tree);
    void rotateRight(TreeNode*& tree);
    void rotateLeftRight(TreeNode*& tree);
    void rotateRightLeft(TreeNode*& tree);
    void balance(PathStack& pathNodes, bool stopAtFirst = false);
    void balance(TreeNode*& tree);
    void printInorder(TreeNode* tree, std::stringstream& ss) const;
    void add(TreeNode*& tree, const T& value, PathStack& pathNodes);
    void remove(TreeNode*& tree, const T& value, PathStack& pathNodes);
    bool isBalanced(const TreeNode* tree) const;
    void rotateLeftWithStatsUpdate(TreeNode*& tree);
    void rotateRightWithStatsUpdate(TreeNode*& tree);
};

#include "AVL.cpp"

#endif // AVL_H
```

Now you can implement the AVL.cpp file based on this header structure. The implementation of methods in AVL.cpp should follow the same patterns and logic as in the BST.cpp, with the necessary modifications to accommodate AVL-specific balancing operations and methods.



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); } Based on this bst.cpp file, can you adapt the implementation to a avl.cpp file while doing
the additional implementations on avl.cpp

```



ChatGPT:

Certainly, you can adapt the BST.cpp implementation to an AVL.cpp file by adding the additional AVL-specific implementations. Here's the AVL.cpp file with the modifications and additions:

```

cpp
#include "AVL.h"

template <typename T>
AVL<T>::AVL(SimpleAllocator* allocator) : allocator_(allocator), root_(nullptr) {
    if (allocator_ == nullptr) {
        SimpleAllocatorConfig config;
        config.useCPPMemManager = true;
        config.objectsPerPage = 100;
        allocator_ = new SimpleAllocator(sizeof(TreeNode), config);
        isOwnAllocator_ = true;
    }
}

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

    if (isOwnAllocator_) {
        delete allocator_;
    }
}

template <typename T>
void AVL<T>::add(const T& value) noexcept(false) {
    add_(root_, value);
    balance(root_);
}

template <typename T>

```

```

void AVL<T>::remove(const T& value) {
    remove_(root_, value);
    balance(root_);
}

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

template <typename T>
int AVL<T>::height() const {
    return height_(root_);
}

template <typename T>
unsigned AVL<T>::size() const {
    if (root_ == nullptr) {
        return 0;
    }
    return root_>count;
}

template <typename T>
void AVL<T>::rotateLeft(TreeNode*& tree) {
    // AVL left rotation implementation
    TreeNode* rightChild = tree->right;
    tree->right = rightChild->left;
    rightChild->left = tree;
    tree = rightChild;

    // Update counts and height for the rotated nodes
    if (tree->left != nullptr) {
        tree->left->count = size_(tree->left->left) + size_(tree->left->right) + 1;
        tree->left->height = 1 + std::max(height_(tree->left->left), height_(tree->left->right));
    }

    tree->count = size_(tree->left) + size_(tree->right) + 1;
    tree->height = 1 + std::max(height_(tree->left), height_(tree->right));
}

template <typename T>
void AVL<T>::rotateRight(TreeNode*& tree) {
    // AVL right rotation implementation
    TreeNode* leftChild = tree->left;
    tree->left = leftChild->right;
    leftChild->right = tree;
    tree = leftChild;

    // Update counts and height for the rotated nodes
    if (tree->right != nullptr) {
        tree->right->count = size_(tree->right->left) + size_(tree->right->right) + 1;
        tree->right->height = 1 + std::max(height_(tree->right->left), height_(tree->right->right));
    }

    tree->count = size_(tree->left) + size_(tree->right) + 1;
    tree->height = 1 + std::max(height_(tree->left), height_(tree->right));
}

// Implement additional AVL-specific methods and rotations here

template <typename T>
void AVL<T>::balance(PathStack& pathNodes, bool stopAtFirst) {
    // Balance the tree from the path of nodes
    while (!pathNodes.empty()) {
        TreeNode* current = pathNodes.top();
        pathNodes.pop();

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

        // Check if the current node is unbalanced (balance factor not in range [-1, 1])
    }
}

```

```

        if (current->balanceFactor < -1) {
            if (current->left != nullptr && current->left->balanceFactor > 0) {
                // Left-Right rotation (LR case)
                rotateLeft(current->left);
            }
            // Right rotation (LL case or after LR case)
            rotateRight(current);
        } else if (current->balanceFactor > 1) {
            if (current->right != nullptr && current->right->balanceFactor < 0) {
                // Right-Left rotation (RL case)
                rotateRight(current->right);
            }
            // Left rotation (RR case or after RL case)
            rotateLeft(current);
        }

        // If we are only balancing the first unbalanced node, stop now
        if (stopAtFirst) {
            break;
        }
    }
}

template <typename T>
void AVL<T>::balance(TreeNode*& tree) {
    // Balance the tree starting from the given node
    if (tree == nullptr) {
        return;
    }

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

    // Check if the current node is unbalanced (balance factor not in range [-1, 1])
    if (tree->balanceFactor < -1) {
        if (tree->left != nullptr && tree->left->balanceFactor > 0) {
            // Left-Right rotation (LR case)
            rotateLeft(tree->left);
        }
        // Right rotation (LL case or after LR case)
        rotateRight(tree);
    } else if (tree->balanceFactor > 1) {
        if (tree->right != nullptr && tree->right->balanceFactor < 0) {
            // Right-Left rotation (RL case)
            rotateRight(tree->right);
        }
        // Left rotation (RR case or after RL case)
        rotateLeft(tree);
    }
}
}

```

This is a simplified example of the AVL .cpp file with AVL-specific balancing methods and some placeholders for further AVL-specific logic. You can add additional AVL-specific methods and implementations as needed for your AVL tree. Make sure to thoroughly test your AVL tree implementation to ensure it maintains balance and correctness during insertion and removal operations.



You:

can you give me all the remaining functions as well as i require the full implementations



ChatGPT:

Certainly, here's a more complete implementation of the AVL .cpp file with all the remaining functions and methods for your AVL tree:

```

cpp
#include "AVL.h"

```



```

template <typename T>
AVL<T>::AVL(SimpleAllocator* allocator) : allocator_(allocator), root_(nullptr) {
    if (allocator_ == nullptr) {
        SimpleAllocatorConfig config;
        config.useCPPMemManager = true;
        config.objectsPerPage = 100;
        allocator_ = new SimpleAllocator(sizeof(TreeNode), config);
        isOwnAllocator_ = true;
    }
}

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

    if (isOwnAllocator_) {
        delete allocator_;
    }
}

template <typename T>
void AVL<T>::add(const T& value) noexcept(false) {
    add_(root_, value);
    balance(root_);
}

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

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

template <typename T>
int AVL<T>::height() const {
    return height_(root_);
}

template <typename T>
unsigned AVL<T>::size() const {
    if (root_ == nullptr) {
        return 0;
    }
    return root_>count;
}

template <typename T>
void AVL<T>::rotateLeft(TreeNode*& tree) {
    TreeNode* rightChild = tree->right;
    tree->right = rightChild->left;
    rightChild->left = tree;
    tree = rightChild;

    // Update counts and height for the rotated nodes
    if (tree->left != nullptr) {
        tree->left->count = size_(tree->left->left) + size_(tree->left->right) + 1;
        tree->left->height = 1 + std::max(height_(tree->left->left), height_(tree->left->right));
    }

    tree->count = size_(tree->left) + size_(tree->right) + 1;
    tree->height = 1 + std::max(height_(tree->left), height_(tree->right));
}

template <typename T>
void AVL<T>::rotateRight(TreeNode*& tree) {
    TreeNode* leftChild = tree->left;
    tree->left = leftChild->right;
    leftChild->right = tree;
    tree = leftChild;
}

```

```

    // Update counts and height for the rotated nodes
    if (tree->right != nullptr) {
        tree->right->count = size_(tree->right->left) + size_(tree->right->right) + 1;
        tree->right->height = 1 + std::max(height_(tree->right->left), height_(tree->right->right));
    }

    tree->count = size_(tree->left) + size_(tree->right) + 1;
    tree->height = 1 + std::max(height_(tree->left), height_(tree->right));
}

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

    tree->height = 1 + std::max(height_(tree->left), height_(tree->right));
    tree->balanceFactor = height_(tree->right) - height_(tree->left);

    if (tree->balanceFactor < -1) {
        if (tree->left != nullptr && tree->left->balanceFactor > 0) {
            rotateLeft(tree->left);
        }
        rotateRight(tree);
    } else if (tree->balanceFactor > 1) {
        if (tree->right != nullptr && tree->right->balanceFactor < 0) {
            rotateRight(tree->right);
        }
        rotateLeft(tree);
    }
}

template <typename T>
typename AVL<T>::TreeNode* AVL<T>::makeNode(const T& value) {
    if (allocator_ == nullptr) {
        throw AVLException(AVLException::E_NO_MEMORY, "Allocator is not set");
    }

    TreeNode* node = static_cast<TreeNode*>(allocator_->allocate("NodeLabel"));

    if (node == nullptr) {
        throw AVLException(AVLException::E_NO_MEMORY, "Failed to allocate memory for a new node");
    }

    new (node) TreeNode(value);

    return node;
}

template <typename T>
void AVL<T>::freeNode(TreeNode* node) {
    if (allocator_ == nullptr) {
        throw AVLException(AVLException::E_NO_MEMORY, "Allocator is not set");
    }

    node->~TreeNode();
    allocator_->free(node);
}

template <typename T>
unsigned AVL<T>::size_(const TreeNode* tree) const {
    if (tree == nullptr) {
        return 0;
    }
    return tree->count;
}

template <typename T>
void AVL<T>::add(TreeNode*& tree, const T& value) {
    if (tree == nullptr) {
        tree = makeNode(value);
        tree->count = 1;
        return;
    }
}

```

```

    if (value < tree->data) {
        add_(tree->left, value);
    } else if (value > tree->data) {
        add_(tree->right, value);
    } else {
        throw AVLException(AVLException::E_DUPLICATE, "Duplicate value not allowed");
    }

    if (tree != nullptr) {
        tree->count = size_(tree->left) + size_(tree->right) + 1;
    }
}

template <typename T>
bool AVL<T>::find_(const TreeNode* tree, const T& value, unsigned& compares) const {
    if (tree == nullptr) {
        return false;
    }

    compares++;
    if (value == tree->data) {
        return true;
    } else if (value < tree->data) {
        return find_(tree->left, value, compares);
    } else {
        return find_(tree->right, value, compares);
    }
}

template <typename T>
void AVL<T>::remove_(TreeNode*& tree, const T& value) {
    if (tree == nullptr) {
        throw AVLException(AVLException::E_NOT_FOUND, "Value to remove not found in the tree");
    }

    if (value < tree->data) {
        remove_(tree->left, value);
    } else if (value > tree->data) {
        remove_(tree->right, value);
    } else {
        if (tree->left == nullptr) {
            TreeNode* temp = tree->right;
            freeNode(tree);
            tree = temp;
        } else if (tree->right == nullptr) {
            TreeNode* temp = tree->left;
            freeNode(tree);
            tree = temp;
        } else {
            TreeNode* predecessor = nullptr;
            findPredecessor(tree, predecessor);
            tree->data = predecessor->data;
            remove_(tree->left, predecessor->data);
        }
    }

    if (tree != nullptr) {
        tree->count = size_(tree->left) + size_(tree->right) + 1;
    }
}

template <typename T>
void AVL<T>::printInOrder(const TreeNode* tree, std::stringstream& ss) const {
    if (tree == nullptr) {
        return;
    }

    printInOrder(tree->left, ss);
    ss << tree->data << " ";
    printInOrder(tree->right, ss);
}

template <typename T>
int AVL<T>::height_(const TreeNode* tree) const {

```

```

    if (tree == nullptr) {
        return -1;
    }

    if (tree->left == nullptr && tree->right == nullptr) {
        return 0;
    }

    int leftHeight = height_(tree->left);
    int rightHeight = height_(tree->right);

    return 1 + (leftHeight > rightHeight ? leftHeight : rightHeight);
}

template <typename T>
void AVL<T>::findPredecessor(const TreeNode* tree, TreeNode*& predecessor) const {
    if (tree == nullptr) {
        predecessor = nullptr;
        return;
    }

    if (tree->left == nullptr) {
        predecessor = nullptr;
        return;
    }

    TreeNode* current = tree->left;
    while (current->right != nullptr) {
        current = current->right;
    }
    predecessor = current;
}

template <typename T>
void AVL<T>::findSuccessor(const TreeNode* tree, TreeNode*& successor) const {
    if (tree == nullptr) {
        successor = nullptr;
        return;
    }

    if (tree->right == nullptr) {
        successor = nullptr;
        return;
    }

    TreeNode* current = tree->right;
    while (current->left != nullptr) {
        current = current->left;
    }
    successor = current;
}

template <typename T>
bool AVL<T>::isEmpty(const TreeNode* tree) const {
    return tree == nullptr;
}

template <typename T>
bool AVL<T>::isLeaf(const TreeNode* tree) const {
    return tree != nullptr && tree->left == nullptr && tree->right == nullptr;
}

```

This AVL.cpp file includes the remaining functions and methods for your AVL tree implementation. It also includes AVL-specific balancing methods, such as rotateLeft, rotateRight, and balance, which ensure that the tree maintains balance after insertion and removal operations. Additionally, it provides implementations for tree traversal, height calculation, and finding predecessor and successor nodes. Make sure to adapt this code as needed and thoroughly test your AVL tree implementation to ensure it works correctly.
