Username for 'https://github.com': 2201512
Password for 'https://2201512@github.com':
remote: Enumerating objects: 17, done.
remote: Counting objects: 100% (17/17), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 9 (delta 8), reused 9 (delta 8), pack-reused 0
Unpacking objects: 100% (9/9), 16.82 KiB | 17.00 KiB/s, done.
From https://github.com/sit-dsa/dsa-practical-test-2023-2201512
   3f6b5b8..d9e59a2 main   -> origin/main
Updating 3c7e28a..d9e59a2
error: Your local changes to the following files would be overwritten by merge:
	AVL.h
	output2.txt
	output4.txt
	output5.txt
	output6.txt
Please commit your changes or stash them before you merge.
Aborting

can you give me the command so that i can overwrite my local changes as i dont care for them

If you want to completely overwrite your local changes and do not care about them, you can use the following command to forcefully discard your local changes and make your local branch exactly match the remote branch:

shell

```
git reset --hard origin/main
```

This command will reset your local branch to match the `origin/main` branch (assuming you are on the `main` branch). It will discard all your local changes, and your working directory will be in sync with the remote repository.

Please be very cautious when using the `--hard` option, as it will permanently discard any uncommitted changes, and they cannot be recovered. Make sure you have backups or copies of any important changes before using this command.

echo "Compiling..."
Compiling...
g++ -o out SimpleAllocator.cpp prng.cpp test.cpp -std=c++17 -Wall
In file included from BST.h:309,
                 from AVL.h:27,
                 from test.cpp:16:
BST.cpp: In member function 'BST<T>& BST<T>::operator=(const BST<T>&)':
BST.cpp:40:8: warning: unnecessary parentheses in declaration of 'rhs' [-Wparentheses]
   40 | BST(rhs); // Copy the rhs tree to this tree (using copy constructor
      | ^
BST.cpp:40:12: error: declaration of 'BST<T> rhs' shadows a parameter
   40 | BST(rhs); // Copy the rhs tree to this tree (using copy constructor
      | ^
make: *** [Makefile:15: compile] Error 1

The error message you're seeing is related to a shadowing issue in your C++ code. Shadowing occurs when a variable in an inner scope has the same name as a variable in an outer scope, causing confusion for the compiler.

In your case, the error message points to the line in `BST.cpp` where you have this code:

cpp

```
BST(rhs); // Copy the rhs tree to this tree (using copy constructor)
```

The problem here is that you are trying to create a new `BST` object named `rhs` within the current scope, but there's already a parameter named `rhs` for the `operator=` function. This causes a shadowing issue, as the variable `rhs` in the current scope shadows the parameter `rhs`.

To fix this issue, you should use the member initialization list to initialize the object instead of trying to create a new one. Here's how you can do it:

cpp

```
BST(const BST<T>& rhs) : /* Initialize your data members here */ {
    // Copy the contents of the rhs tree to this tree
    // You should perform the actual copying here
}
```

Replace `/* Initialize your data members here */` with the actual initialization of your data members in the `BST` class. Additionally, inside the constructor, you should perform the copying of the tree contents from `rhs` to the current object, which is the main purpose of the copy constructor.

Make sure you update your code accordingly in both the declaration and definition of your copy constructor to avoid shadowing the parameter `rhs`.

template <typename T>
BST<T>::BST(const BST& rhs) {
  allocator_ = rhs.allocator_;// Copy the 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();
  BST(rhs); // Copy the rhs tree to this tree (using copy constructor)
  return *this;
}

It looks like you've provided the code for your copy constructor and assignment operator for a binary search tree (BST). However, there are a couple of issues in your code:

1. In your assignment operator (`operator=`), the line `BST(rhs);` does not actually perform the copy operation you intend. It creates a temporary `BST` object but doesn't update the current object. You should instead use the copy constructor and then swap the contents of the current object with the temporary one. Here's the corrected code:

cpp

```
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();

    // Use the copy constructor to create a temporary copy of rhs
    BST<T> temp(rhs);

    // Swap the contents of the current object with the temporary one
    std::swap(root_, temp.root_);
    std::swap(allocator_, temp.allocator_);
    std::swap(isOwnAllocator_, temp.isOwnAllocator_);

    return *this;
}
```

1. The `copy_` function you mentioned should be responsible for copying the tree structure and elements, but you didn't provide its implementation. Make sure that the `copy_` function properly handles the copying of the tree structure and elements from the source tree (`rhs`) to the destination tree (the current object).
2. It's a good practice to define a destructor to clean up any dynamically allocated resources in your binary search tree. Make sure you have a destructor to release memory associated with nodes in the tree when the BST is destroyed.
3. If you have a custom allocator, make sure it is properly managed and shared between the trees as needed.

With these changes, your assignment operator should correctly perform a deep copy of the binary search tree from `rhs` to the current object.

template <typename T>
BST<T>::BST(const BST& rhs) {
  allocator_ = rhs.allocator_;// Copy the 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();
  BST(rhs); // Copy the rhs tree to this tree (using copy constructor)
  allocator_ = rhs.allocator_;// Copy the 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_);
  return *this;
}

how do i use the copy contructor in the operator overload

To use the copy constructor in the copy assignment operator, you should create a temporary object using the copy constructor and then swap the contents of the temporary object with the current object. Here's how you can modify your `operator=` function to use the copy constructor:

cpp

```
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();

    // Use the copy constructor to create a temporary copy of rhs
    BST<T> temp(rhs);

    // Swap the contents of the current object with the temporary one
    std::swap(root_, temp.root_);
    std::swap(allocator_, temp.allocator_);
    std::swap(isOwnAllocator_, temp.isOwnAllocator_);

    return *this;
}
```

This code first creates a temporary `BST` object `temp` using the copy constructor, effectively making a copy of the `rhs` tree. Then, it uses `std::swap` to exchange the contents of the current object with the temporary object. This way, you ensure that you have a deep copy of the `rhs` tree in your current object.