CSCI 2270 LAB: BINARY SEARCH TREES

The Bag Class with a Binary Search Tree, adapted from Michael Main's version

The Assignment:

Implement the BSTreeBag template class, using a binary search tree to store the items.

Purposes:

Ensure that you understand and can use binary search trees and recursive algorithms for processing them.

Before Starting:

Read this handout and skim the Trees chapter (pp. 507-535) in Carrano.

Due Dates:

Part 1: Insert, count functions, due by moodle this Saturday.

Part 2: Removal functions, due by moodle next Saturday.

Files that you must write:

BSTTreeBag.h: Header file for this version of the BSTreeBag class. You don't have to write much of this file unless you add helper functions. Just copy our version and add your name and other information at the top.

BSTreeBag.cxx: The implementation file for the new BSTreeBag class. You have stubs for the functions you need to write.

Other files that you may find helpful:

bintree.h and bintree.cxx. These define the binary tree node template class. You don't have to write anything for these. NOTE: This version of the binary tree node template has the return values from the non-const versions of the **`left()`** and **`right()`** functions return a reference to the pointer in the node. This is indicated by the & symbol for them in bintree.h and bintree.cxx:

```
binary_tree_node*& left( )
```

The use of a "reference return type" (indicated by the ampersand) in the return value has two advantages that simplify the material. It now allows a direct assignment such as: **`p->left() = nullptr`**. This is not a huge advantage, since the same thing can be accomplished by using the **`set_left`** function.

The expression **`p->left()`** can be passed as the argument to a function such as: **`tree_clear(p->left())`**; The parameter of **`tree_clear`** is a reference parameter, so that any changes that **`tree_clear`** makes to **`p->left()`** will now affect the actual left pointer in the **`binary_tree_node<ItemType>* p`**. In this example, the **`tree_clear`** function does set its parameter to **`nullptr`**, so that the total effect of **`tree_clear(p->left())`** is to clear the left subtree of **`p`** and to set **`p`** 's left pointer to **`nullptr`**.

In the case of `tree_clear`, this is not a huge advantage because we could have just set `p`'s left pointer to `nullptr` ourselves. But, in this assignment, there are two functions, `bst_remove` and `bst_remove_max`, which are easier to write if we can use `root_ptr->left()` and `root_ptr->right()` as the parameters of recursive calls. See my implementations in BSTreeBag.h for details.

BSTreeBagTest.cxx: A simple interactive test program.
BSTreeBagExam.cxx: A non-interactive test program that will be used to grade the correctness of your bag class.
Makefile: for compiling the assignment.

The Bag Class Using a Binary Search Tree
Discussion of the Assignment

This assignment is another template class. I am giving you the code for bintree.h and bintree.cxx, which define a template class for the `binary_tree_node`. Use these files, but do not change them. You'll see the specification for the whole basic bintree class in bintree.h: We begin by making a namespace for your class (the test code has to say

```
using bst2270;
```

to use `binary_tree_nodes` without having to say `bst2270::binary_tree_node` all the time.)

```
namespace bst2270
{

      template <class ItemType>     // bag holds any ItemType
      class binary_tree_node        // class name
      {
            public:

            // TYPEDEF
            typedef ItemType value_type;  // comparable ItemTypes

            // CONSTRUCTOR          // if data was not supplied,
                                    // it gets initialized here.
            binary_tree_node(
                  const ItemType& init_data = ItemType( ),
                  binary_tree_node* init_left = nullptr,
                  binary_tree_node* init_right = nullptr
            )
            {
                  data_field = init_data;
                  left_field = init_left;
                  right_field = init_right;
            }

            // MODIFICATION MEMBER FUNCTIONS
            ItemType& data( ) { return data_field; }
            binary_tree_node*& left( ) { return left_field; }
            binary_tree_node*& right( ) { return right_field; }

            // CONST MEMBER FUNCTIONS
            const ItemType& data( ) const { return data_field; }
```

```
            const binary_tree_node* left( ) const
                  { return left_field; }
            const binary_tree_node* right( ) const
                  { return right_field; }
            bool is_leaf( ) const
                  { return (left_field == nullptr) &&
                  (right_field == nullptr); }
```

Next come your private member variables for your state as a node in a binary tree:

```
      private:
            ItemType data_field;
            binary_tree_node *left_field;
            binary_tree_node *right_field;
};
```

Last come the nonmember functions for making trees from these binary tree nodes.

The first 3 do tree traversals (remember there are 3 orderings) and pass in a Process as a function to be done on every element on the tree (now we are passing functions like they're data, mind you). I didn't actually use these functions for this homework, but they're here if you want them.

```
// NON-MEMBER FUNCTIONS for the binary_tree_node<ItemType>:
template <class Process, class BTNode>
void inorder(Process f, BTNode* node_ptr);

template <class Process, class BTNode>
void preorder(Process f, BTNode* node_ptr);

template <class Process, class BTNode>
void postorder(Process f, BTNode* node_ptr);
```

The next one, print, is very useful for displaying your tree:

```
template <class ItemType, class SizeType>
void print(binary_tree_node<ItemType>* node_ptr, SizeType depth);
```

We have a tree_clear function to destroy a tree (which has to happen recursively without losing the links):

```
template <class ItemType>
void tree_clear(binary_tree_node<ItemType>*& root_ptr);
```

We have a tree_copy function to make a deep copy of a full tree (also recursive):

```
      template <class ItemType>
      binary_tree_node<ItemType>* tree_copy(const binary_tree_node<ItemType>*
root_ptr);
```

Finally, the code has a recursive method to count up all the nodes in the tree and return that number:

```
template <class ItemType>
```

```
	typename binary_tree_node<ItemType>::size_type tree_size(const
binary_tree_node<ItemType>* node_ptr);
```

We'll discuss the bintree functions in class, but you can trust that they already take care of a lot of the binary search tree memory functions. I used print for debugging a lot. (Start print's depth at 0, and depth will count up in recursions and terminate properly.)

Your bintree.cxx and bintree.h files give you the basic functions of a binary tree, but you are writing a class that extends this binary tree into a *binary search tree*. In the textbook, Carrano's binary search tree has the rule that a binary search tree node has data that is greater than all the data in its left subtree, and its data is also less than all the data in its right subtree. But this doesn't let us add duplicates to a tree, which is overly restrictive. We'll relax this rule to say that the data in a binary search tree node is >= the data in its left subtree, and < the data in its right subtree. Now we can put duplicate entries in and not lose them.

In BSTreeBag.h, you will see the functions you need to create in the BSTreeBag.cxx file. These are template functions, like the bintree.cxx and bintree.h ones. You can use the functions in bintree to get this done; it will help you keep your code short and safe.

```
	#ifndef BAG6_H
	#define BAG6_H
	#include <cstdlib>
	#include "bintree.h"   // Provides binary_tree_node and related
functions

	namespace bst2270
	{
		template <class ItemType>
		class BSTreeBag
		{
			public:

				// TYPEDEFS
				typedef unsigned int size_type;
				typedef ItemType value_type;
				// CONSTRUCTORS and DESTRUCTOR
				BSTreeBag( );
				BSTreeBag(const BSTreeBag& source);
				~BSTreeBag( );

				// MODIFICATION functions
				size_type erase(const ItemType& target);
				bool erase_one(const ItemType& target);
				void insert(const ItemType& entry);
				void operator +=(const BSTreeBag& addend);
				void operator =(const BSTreeBag& source);
				// CONSTANT functions
				size_type size( ) const;
				size_type count(const ItemType& target) const;

			private:

				// Root pointer of binary search tree
				binary_tree_node<ItemType> *root_ptr;
```

```
                        void insert_all(binary_tree_node<ItemType>*
addroot_ptr);
            };

            // NONMEMBER functions for the BSTreeBag<ItemType> template class
            template <class ItemType>
            BSTreeBag<ItemType> operator +(const BSTreeBag<ItemType>& b1,
      const BSTreeBag<ItemType>& b2);
}

#include "BSTreeBag.cxx" // Include the implementation.
#endif
```

1. BASIC BINARY SEARCH TREE FUNCTIONS

Begin (as we always must) by considering a constructor. Don't forget the template line for each of these functions. Like this:

```
template <class ItemType>
BSTreeBag<ItemType>::BSTreeBag()
{
      root_ptr = nullptr;
}
```

Your Bag should set its `root_ptr` to `nullptr` here. Since no numbers have yet been added, we aren't storing any nodes yet. For us, an empty tree means the `root_ptr` is `nullptr`, always. That's all this constructor needs to do.

Your copy constructor, next, uses the `tree_copy` function in bintree.h.

Your destructor, likewise, uses `tree_clear`.

Your assignment operator makes a call to `tree_clear` if needed, and it also uses `tree_copy`.

Your size() function uses `tree_size`.
Part 1 tasks: count and insert

```
unsigned int BSTreeBag<ItemType>::size( ) const
```

The next steps are to `insert` an `entry`, so you have a tree to play with. Remember that you always know the tree's `root_ptr` in this code, since that is a member of the class.

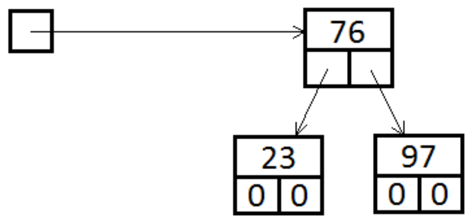
```
void BSTreeBag<ItemType>::insert(const ItemType& entry)
```

This involves handling one special case, where you're inserting into an empty tree like the one that your default constructor made. You can tell if that's true, provided that you set `root_ptr` correctly in that constructor. If so, say

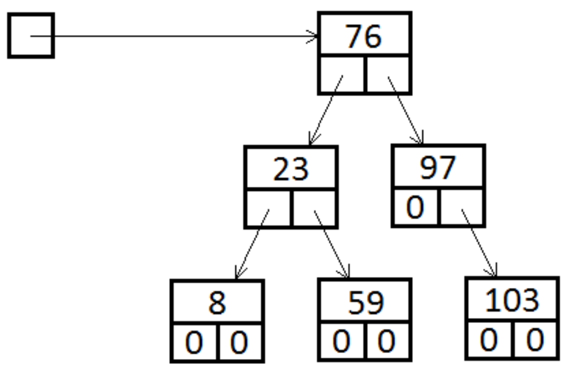
```
root_ptr = new binary_tree_node<ItemType>(entry);
```

to make a new node on the heap (with left and right child pointers `== nullptr`, and data containing the `entry`). The address of this node will now be stored in your `root_ptr`. In that case, your `insert` is done. In the example below, we've inserted 76 into an empty binary search tree of integers.

If the tree already has data in it, though, the code has to put the new `entry` in the right place. This works a lot like the method for searching for an item in a tree (as discussed quite ably in the trees chapter of Carrano's book). In the example below, we have added 23 and 97 to the tree above:



Later, if we add more numbers (59, 103, 8), we'll see:



One way to find the right place to insert a new `entry` is is to define a boolean variable called `done`, which starts as `false` (we have done nothing yet!) and a node pointer (`binary_tree_node<ItemType>* cursor`), which starts as `root_ptr`. Then, while `done` remains `false`,

If the data at the `cursor` is greater than or equal to the `entry`,

If the `cursor` has no left child (its `left()` pointer is `nullptr`), you can add the new `entry` there.

In this case, add the new binary_tree_node with this `entry` as the left child `cursor->left() = new binary_tree_node<ItemType>(entry);`

and note that you are now `done`.

else, keep looking in the left subtree

`cursor = cursor->left();`

Else, similar logic applies to the right subtree.

Notice that `cursor->left()` is now returning a `binary_tree_node<ItemType>*&`, and the reference return type allows us to assign to it. This change sticks around after the code finishes, so you have really added the node. Slick.

Counting the copies of a `target` item in the tree:

```
unsigned int BSTreeBag<ItemType>::count(const ItemType& target) const
```

This requires us to walk a `binary_tree_node<ItemType>* cursor` from the `root_ptr` down through the tree, looking for more copies of `target` and counting our `answer` up with each one we find. Initialize your `answer` to 0. Start `cursor` out as `root_ptr`.

While the `cursor` is not `nullptr`,   // edited for better description

if the data at `cursor` is equal to the `target`, increment the `answer` by 1.

If the data at the `cursor` is greater than or equal to the `target`, set `cursor` to look for the `target` in the left child.

If the data at the `cursor` is less than the target, set `cursor` to look for the `target` in the right child.

When your cursor becomes nullptr, your count is correct--at least so we hope. That's all you need for part 1, this week. Note that now you

Once this code is working, you should see `operator+=` working too. We use the pre-supplied helper function `insert_all`:

```
void BSTreeBag<ItemType>::insert_all(binary_tree_node<ItemType>* addroot_ptr)
```

Here, if the `addroot_ptr` is not `nullptr`, we insert its data into ourselves, and call

```
insert_all(addroot_ptr->left());
// and one other thing you can deduce from the line above.
```

Now, `operator+=` can check for self assignment (very important). If the assignment's NOT something like `b += b`, calling `insert_all` directly will work to do the job. If it is like `b += b`, though, we need to double `b`. For this, make a copy of the `addroot_ptr`, use that as the root to `insert_all`, and then clear out the copy. Else, what bad thing can happen? Ponder that.

Having `operator+=` working lets us make `operator+`.

PART 2. ERASING AN ITEM FROM A BINARY SEARCH TREE

To erase an item from a binary search tree is harder. We usually have to replace the erased item with something else to keep the tree an honest binary search tree. Erasing is complicated enough that we need 3 helper functions for all of the cases. One is `bst_remove`. This corresponds to erase_one; your public function erase_one will call your helper function `bst_remove` to do its whole job.

2a. `bst_remove`

The code I used has a second helper function called `bst_remove_max` that `bst_remove` might need to use. But first, let's talk about `bst_remove`. `bst_remove` removes a single copy of a `target` from a tree, and updates the `root_ptr` (and the others) as needed. It returns `true` if it found the `target` to remove, and `false` if it did not.

```
	bool bst_remove(binary_tree_node<ItemType>*& root_ptr, const ItemType&
target)
```

The first case for `bst_remove` to handle is when the `root_ptr == nullptr`. In this case, we can return `false` right away; there is no search tree left to remove an item from. That's a base case for the recursion.

Else, if the `root_ptr`'s data is greater than the `target`, we must look in the left subtree for the item to remove. Make a recursive call using the left child of the `root_ptr`:

```
	return bst_remove(root_ptr->left(), target);
```

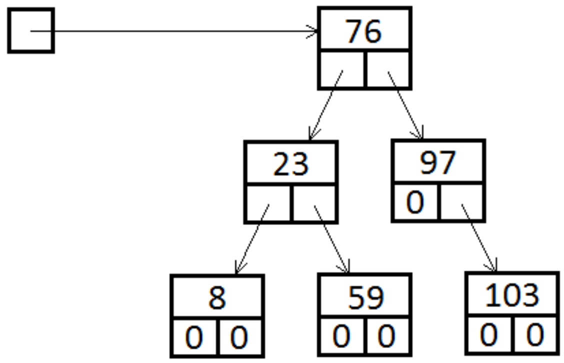
Else if the `root_ptr`'s data is less than the `target`, look in the right subtree; do something like the line above;

Else, you've found the `target` at this `root_ptr`. Congratulations. Things now get complicated:

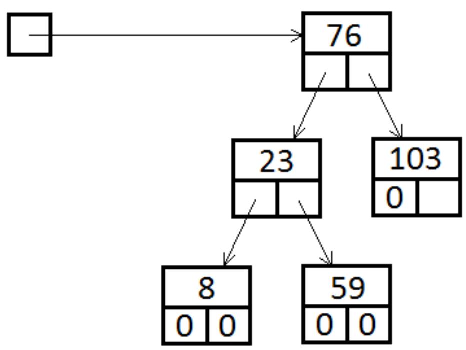
If the `root_ptr` has no left child, then you can delete `root_ptr`, but you want to replace it with its right child (even though that might be `nullptr`, it's ok).

```
		binary_tree_node<ItemType>* old_root_ptr = root_ptr;
		root_ptr = root_ptr->right();       // replace with right child
		delete old_root_ptr;                // no leaks
```

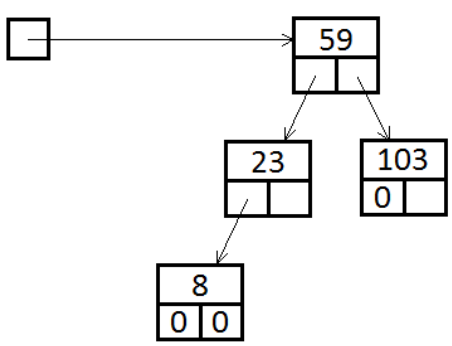
Consider our tree again.



To reinforce this example, suppose we wish to delete the 97. Here is what should happen:



Else, you are removing a **target** in the middle of the tree with children, which is harder. You'll probably have to replace this removed **target** with another entry in the tree to keep it a valid binary search tree. In this case, the best thing to do is to replace the **target** we're deleting with the largest item in our left subtree. Suppose we want to delete the 76 from our tree and illustrate this example. Here is what should happen:



This function's handled by the **bst_remove_max** function. Done correctly, this lets us replace our own data with the item we find there:

```
bst_remove_max(root_ptr->left(), root_ptr->data());
```

Now all your `bst_remove` cases should be done.

2b. `bst_remove_max`

Back to `bst_remove_max`, which removes the largest item from a tree and writes this item back by reference to `removed` using a reference input parameter):

```
void bst_remove_max(binary_tree_node<ItemType>*& root_ptr, ItemType&
removed)
```

This means that when you find the `binary_tree_node<ItemType>*` with the maximum value (call that node `frodo` here), you say

```
removed = frodo->data();       // function remembers removed item
```

`bst_remove_max` must consider 2 recursive cases.

The first is that it has been given a `root_ptr` with no right child. In this case, the data in `root_ptr` is what needs to be removed, as above. We want to delete this `root_ptr` node, too, but carefully.

```
binary_tree_node<ItemType>* old_root_ptr = root_ptr;
```

If `root_ptr`'s right child is missing, we can replace the node we are deleting with its left child without breaking the rules of a binary search tree:

```
root_ptr = root_ptr->left();  // copy left child to root's address
```

Now we can delete the `old_root_ptr`.

The second case is when the `root_ptr` has a right child, and we should look there for even bigger items. The code below will update the right child (lvalue!) and the data (lvalue!) automatically.

```
bst_remove_max(root_ptr->right(), root_ptr->data());
```

2c. `bst_remove_all`

Erasing all the items in a tree requires you to write `bst_remove_all`. This works like `bst_remove`, but if `bst_remove_all` finds and removes the target, it must still keep looking in case more copies of the target exist. When you write this, have `erase` call `bst_remove_all` and you are done.

SOME HINTS:

Work piece by piece. Comment out the tests you can't pass in the functions.

Since this is a template class, debugging can be more difficult (some debuggers don't permit breakpoints in a template function.) To help in debugging, you can call `print(root_ptr, 0)` in a program to print the binary search tree for the bag you're changing. But clean those out before you submit the code.

If you use our Makefile, you might notice that the BSTreeBag and bintree templates in the .cxx files are never compiled on their own, but in order to create BSTreeBagTest and BSTreeBagExam, all the template files must be present in the current directory.

When you have the code working, show your TA for a fast grade.