CSCI 2270
Data Structures and Algorithms
Binary Trees and Binary Search Trees

Elizabeth White
elizabeth.white@colorado.edu
Office hours: ECCS 112/128
Wed 9:30am-11:30am
Thurs 10am-11am

Administrivia

Read Trees chapter, pp.507-535

In lab this week, you'll write insert and count for the binary search tree class. You'll have a lot of code supplied already.

Destroying a binary tree

```
void tree_clear(binary_tree_node<ItemType>*& root_ptr) {
        binary_tree_node<ItemType>* child;
        if (root_ptr != nullptr)
        {
                child = root_ptr->left( );
                tree_clear( child );
                child = root_ptr->right( );
                tree_clear( child );
                delete root_ptr;
                root_ptr = nullptr;
        }
  }
```

Copying a binary tree

```
binary_tree_node<ItemType>* tree_copy(const
        binary_tree_node<ItemType>* root_ptr) {
        binary_tree_node<ItemType> *l_ptr, *r_ptr;
        if (root_ptr == nullptr)  return nullptr;
        else {
                l_ptr = tree_copy( root_ptr->left( ) );
                r_ptr = tree_copy( root_ptr->right( ) );
                return new binary_tree_node<ItemType>
                        (root_ptr->data( ), l_ptr, r_ptr);
        }
}
```

Traversal order matters (*post order*)

Tree_clear:
```
child = root_ptr->left( ); tree_clear( child );
child = root_ptr->right( ); tree_clear( child );
delete root_ptr; root_ptr = nullptr;
```

Tree_copy:
```
l_ptr = tree_copy( root_ptr->left( ) );
r_ptr = tree_copy( root_ptr->right( ) );
return new binary_tree_node<ItemType>
        (root_ptr->data( ), l_ptr, r_ptr);
```

Binary tree node access

In bintree.h, you have methods to get at the data and the children:

```
ItemType& data( ) { return data_field; }
binary_tree_node*& left( ) { return left_field; }
binary_tree_node*& right( ) { return right_field; }
```

Notice, please, that each of these returns a reference. This means that we can assign to them. THIS IS WEIRD.

```
root_ptr->left() = new binary_tree_node(entry);
root_ptr->data() = 2;
```

Inserting an Item

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

2 cases:

empty tree (easy to find where the Item goes)

non-empty tree: walk a pointer from the root to a leaf

```
binary_tree_node<ItemType> *cursor;
cursor = root_ptr;
bool done = false;
while (!done)
{
    // find where this entry goes
}
```

Inserting an Item

```
void BSTreeBag<ItemType>::insert(const ItemType& entry)
        cursor = root_ptr; bool done = false;
        while (!done) {
                if (entry <= cursor->data())  {
                        // if cursor's left child is nullptr, add entry:
                                cursor->left() = new
binary_tree_node<ItemType> (entry);  done = true;
                        // else keep looking
                                cursor = cursor->left();
                // else look in the right subtree
                }
        }
```

Counting Items

```
unsigned int BSTreeBag<ItemType>::count(const ItemType&
target) const
{
        unsigned int answer = 0;
        binary_tree_node<ItemType> *cursor;
        while (cursor != nullptr)
        {
                // check if cursor has the target as its data
                // if so, count answer up
                // set cursor to the right child and look for more
        }
}
```

Removing Items

Inserting is easy, removing is hard

Reason: we need to keep the tree kosher after removals

2 cases:

- we're removing a node with no left subtree
    - we delete this node and move the right subtree up into its spot
- we're removing a node with a left subtree
    - we find the maximum node in our left subtree,
    - delete that node,
    - and copy its data to the node we wanted to delete

Trees offer the chance for log(n) performance if you're lucky

For a binary search tree of n nodes,

Search: O(log n) average, O(n) worst case

Insert: O(log n) average, O(n) worst case

Remove: O(log n) average, O(n) worst case

Traverse: O(n)

Another tree ADT, the B-tree, ensures that the tree is perfectly balanced (and even full); this guarantees the O(log n) performance, worst case