Trees and `TreeMaps`

Chapter 16—which you should all start reading now—introduces trees, and binary search trees in particular. The chapter isn't very long, but it does spend some time talking about balancing strategies used by real binary search trees to ensure that no particular insertion order ever brings out the worst-case structure and performance of future insert and search operations. Today's lecture will exercise your understanding of binary search trees, discuss how various insertions orders of the same elements into a binary search tree can lead to different tree structures (some better than others), and how binary search trees can be used to back the **`Map`** we've been familiar with since week 2 of the course.

We'll also be careful to cover trees as data structures in general. They needn't always be binary search. ☺

`tree-map.h`

```
template <typename Key, typename Value>
class TreeMap {
public:
   TreeMap();
   ~TreeMap();

   bool isEmpty() const { return size() == 0; }
   int size() const { return count; }

   bool containsKey(const Key& key) const;
   void put(const Key& key, const Value& value);
   Value get(const Key& key) const;
   Value& operator[](const Key& key);

private:
   struct node {
      Key key;
      Value value;
      node *left, *right;
   };
   node *root;

   void disposeTree(node *root);
   const node *const& findNode(const Key& key) const;
   node *ensureNodeExists(const Key& key);
};

#include "tree-map-impl.h"
```

tree-map-impl.h

Once again, we make assumptions about what **Key** and **Value** support. In particular, I assume **Key** plays well with infix **<** and **==**, that both **Key** and **Value** can be copied, and each have zero-argument constructors.

There are complexities that come with being **const**-correct that I'll invest a little bit of time discussing, but I don't want to obscure the larger takeaway that binary search trees are commonly used to back dynamic data structures that need to provide fast insertion and search routines while simultaneously structuring everything so that **for** and iteration can surface all of the **Key**s in sorted order.

```
template <typename Key, typename Value>
TreeMap<Key, Value>::TreeMap() {
   root = NULL;
}

template <typename Key, typename Value>
TreeMap<Key, Value>::~TreeMap() {
   disposeTree(root);
}

template <typename Key, typename Value>
void TreeMap<Key, Value>::disposeTree(node *root) {
   if (root == NULL) return;
   disposeTree(root->left);
   disposeTree(root->right);
   delete root;
}

template <typename Key, typename Value>
bool TreeMap<Key, Value>::containsKey(const Key& key) const {
   return findNode(key) != NULL;
}

template <typename Key, typename Value>
Value TreeMap<Key, Value>::get(const Key& key) const {
   const node *found = findNode(key);
   return found != NULL ? found->value : Value();
}

template <typename Key, typename Value>
void TreeMap<Key, Value>::put(const Key& key, const Value& value) {
   ensureNodeExists(key)->value = value;
}

template <typename Key, typename Value>
Value& TreeMap<Key, Value>::operator[](const Key& key) {
   return ensureNodeExists(key)->value;
}
```

```
template <typename Key, typename Value>
const typename TreeMap<Key, Value>::node * const &
TreeMap<Key, Value>::findNode(const Key& key) const {
   const node * const *currp = &root;
   while (*currp != NULL && !((*currp)->key == key)) {
      if ((*currp)->key < key) currp = &(*currp)->right;
      else currp = &(*currp)->left;
   }
   return *currp;
}

template <typename Key, typename Value>
typename TreeMap<Key, Value>::node *
TreeMap<Key, Value>::ensureNodeExists(const Key& key) {
   node *& found = const_cast<node *&>(findNode(key));
   if (found == NULL) {
      node n = {key, Value(), NULL, NULL};
      found = new node(n);
   }
   return found;
}
```