Searching Binary Search Trees, Reading

Let's start with the easy operations, because they teach us about the structure of binary search trees without getting into a lots of complicated details -- those will come later in this module!

In the traversals in the sample code, we use a **Node* p;** pointer to traverse through the tree, usually starting from the root. The pointer is *not* a data member. It's declared locally in each function template that needs it (which is most of them). Note that "p" *cannot* be a data member because it's used and set in getter functions. Those could not be getters if they changed the value of a data member (unless you know about the "mutable" keyword, which we will not be covering in this course).

The containsKey Getter

This is the O(log n) operation that attracted us to BSTs in the first place. It's basically the linked implementation of the binary search algorithm, without the middle and range index calculations. It's more like traversing a singly-linked list with **p=p->next** to traverse from the head to the end, except that we have to choose between **p=p->left** and **p=p->right** to direct us through the tree. If we reach a dead-end without having found a match, then the key is not there.

```
Node* p = rootNode;
while (p)
{
  if (p->key == key) break; // found it!
  if (p->key < key) p = p->right; else p = p->left;
}
return p != 0; // not found if the above loop exits because p gets to zero
```

Notice that we did not return out of the loop -- we *broke out* and used the value of p to decide what to do next. As we noted in the previous module, this is a good way to design the search logic because so many of the function templates will be using the same search logic, but doing something different with the result.

The Square Bracket Getter

The difference between this and **containsKey** is that **operator[ ]** returns a *copy* of the value instead of a Boolean. So if "p" is a valid pointer value (that is, not null) the **p->value** is what to return from the function. But if "p" is null (or zero), we still have to return *something* -- this is a value-returning function and is expected to return something, no matter where the logic may lead. Since it's just a copy, we just return the default **typename V** value.

```
Node* p = rootNode;
while (p)
{
  if (p->key == key) break; // found it!
  if (p->key < key) p = p->right; else p = p->left;
}
if (p) return p->value;
return V( );
```

There's no need for an "else" in the **if(p)** because the only way we get to the last two lines is if "p" is null.