Lecture Summary

In this lecture, we learned in more detail about the abstract data type Dictionary (which is called "map" in STL and other places). We saw its definition and uses, some simple implementations, and then got started on implementing it using balanced search trees.

Several times throughout the semester, including in our very first lecture, we have alluded to the data type Dictionary. This is a data type that allows us to store maps between a key and a value, and then look up for a given key its corresponding value. The example that we used in our first lecture is looking up the definition of a word: the word is the key (which we use for looking things up), and the associated value is the definition for the word.

1 Definitions and Uses

Formally, the ADT Dictionary is defined as follows:

```cpp
template<class KeyType, class ValueType>
class Dictionary {
    void add (const KeyType & key, const ValueType & value);
    void remove (const KeyType & key);
    ValueType get (const KeyType & key) const;
};
```

The desired functionality is as follows:

1. If a pair (key, value) has been added, and key has not been removed since then, and we call get(key), then value should be returned.

   If multiple pairs with the same key have been added, then there are different semantics. One might choose this to mean that multiple copies should be added. (This is sometimes called a multimap.) In that case, one has to carefully specify which value gets returned when get is called, and which copy gets removed when remove is called. (Or perhaps, all of them get removed?) If one only allows one copy of each key, then the typical semantics is that adding another pair (key, value) with the same key will overwrite the old value.

2. If no pair (key, value) has been added for a particular key, or there has been a remove(key) since the most recent such addition, then get(key) should throw an exception, or otherwise signal that there was no such key present.

   Notice that this is a definition with two template types, but there's nothing particularly different about it. If we wanted to define a Dictionary whose keys were integers, and whose values were strings, we would do so via Dictionary<int,string> myDict.

   While the textbook calls this ADT "Dictionary," that is not a particularly standard name. In STL, as we discussed in that lecture, this type of data structure is called a map. It combines keys and values into pairs in the add function, i.e., the add function only has one parameter, which is a Pair<KeyType,ValueType>. 

1
Dictionaries/Maps are called *associative data structures*: they create “associations” between keys and values. It’s similar to your brain creating an association between the word “flower” and the image of a flower, or between the sensation of pain and the word “pain.”

There are many many applications of Dictionaries all through computer science and programming projects, some of which we’ve seen before. Here are some examples.

1. An actual dictionary for a language, our first example from class. Here, the key is the word, and the value is the definition.

2. A lists of users of a web site or service, like your social networking application. Here, the key is the login ID (such as a username), and the value is the user’s full record.

3. Databases for products, in which employees (or web site users) can look up information. Here the key may be the item’s ID number, and the value is the record of the product.

4. In implementing a search engine (e.g., Google), it is important to look up the web sites that are relevant for a particular query. Here, the key would be the user’s query (i.e., a string of words, or just one word), and the value would be the list of web sites matching that query (e.g., their URLs, or the index in an array where they are all stored).

Notice that Google of course doesn’t just work by lookups like that; there is a lot of post-processing involved, since there may be millions of matching pages, and the user would like to see the most relevant ones first, based on his/her own preferences.

## 2 Simple Implementations

The ADT Dictionary specifies what functionality we want, but not how to achieve it. In practice, we have to store the elements somewhere. Perhaps the simplest way is the way we have been storing user lists and similar structures so far: in an array or list-like structure. For each pair of a key and a value, we create a `Pair` (say, as a `struct`), and store that. When we call `get` to find the value associated with a key, we scan through all the pairs (linear search) until we find one whose key matches the given key.

There are slight differences based on what linear data structure we use to store the pairs. We could use an array/vector, or a linked list.

- For the array/vector, we could keep it sorted or unsorted. For an unsorted array or linked list, insertion takes \( O(1) \). But to get or remove an element, we need to find it, which now takes \( \Theta(n) \).

- If an array/vector is sorted, we can use binary search for `get`, speeding it up to \( O(\log n) \). But insertion and removal take \( \Theta(n) \), because we may need to shift much of the array to the right or left.

Of course, given that we don’t even know what the `KeyType` is, it’s not clear that we can sort by it. For instance, what if the keys are colors: \{red, blue, orange, green\}? In order to make this work, we would have to assume that `KeyType` implements a comparison function in some way, perhaps calling it `compare` (so you could call `key1.compare(key2)`), or perhaps by overloading the comparison operators \(<, <=, >, >=\). This means, of course, that you can only use your Dictionary implementations on `KeyType` classes that implement these desired operations.

In fact, we will see the same later as well: for some implementations, it’s necessary that the keys are *ordered* in this sense (they can be compared). The same, by the way, happens in STL, which distinguishes `map` from `unordered_map`, the former requiring the `KeyType` to implement a comparison operator.

In summary, we get the following running times:

<table>
<thead>
<tr>
<th></th>
<th>Unsorted array or list</th>
<th>Sorted array or vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>( \Theta(1) )</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>remove</td>
<td>( \Theta(n) )</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>get</td>
<td>( \Theta(n) )</td>
<td>( \Theta(\log n) )</td>
</tr>
</tbody>
</table>
None of these are particularly fast: looking up items in linear time is quite inefficient, since you want to think about data structures storing millions or billions of items. So for the next 4–5 lectures, we will learn about more advanced data structures that are much faster. The data structures are the following:

**Search Trees:** Search Trees generalize the ideas behind Binary Search, and support all three operations in time $O(\log n)$. They require that the keys be comparable, just like for sorted arrays.

**Hashtables:** Hashtables generalize the idea of a direct access array. The analysis doesn’t quite make worst-case guarantees, but typically, operations take time $O(1)$. They don’t require that keys be comparable, but instead need the keys to provide a hash function. They also are not as good as search trees at iterating through all elements.

### 3 Search Trees

Search Trees are one nice and standard way of implementing Dictionaries. Many search trees are binary, although the ones we will analyze in depth here (2-3 Trees) have nodes with degrees 2 or 3. But for the purpose of our introduction, let us focus only on Binary Search Trees for now. The key property that makes a (binary) tree a search tree is that the following holds at every node $v$:

Let $k$ be the key in node $v$, and $T_1$ and $T_2$ the left and right subtrees at $v$. Then, all keys in $T_1$ are less than $k$, and all keys in $T_2$ are greater than $k$.

An example tree to illustrate this concept is given in Figure 1:

![Figure 1: An illustration of the Search Tree property. The property is satisfied almost everywhere. The exception is the gray node labeled “12”. It is in the left subtree of the node labeled “10”, which violates the property that all keys in the left subtree of a given node are smaller than the key in the node itself.](image)

Contrast the search tree property with the Heap Property. In a (Max-)Heap, all keys in all subtrees below $v$ must be smaller than the key at $v$. In a search tree, the nodes in the left subtree must be smaller, while the ones in the right subtree must be larger.

Another thing to notice is that we state that all keys in $T_1$ are smaller than the key at $v$. Notice that we didn’t write “at most as large.” This restricts the search trees so that they cannot contain the same key twice. Sometimes, you may want to allow a dictionary to have multiple copies of the same key, but as a general rule, that tends to complicate implementations and usage, so at least for this introductory class, we will not consider that case.

Notice that we didn’t draw any values in this tree, just keys. That will be a common theme for our analysis of dictionaries. While the purpose of a dictionary is to store keys and values, all the interesting operations (comparisons, hash functions) will be performed on keys, and the values are just along for the ride. So to illustrate how the data structures actually works, we will only focus on the keys, with the understanding that the values are of course also there.
We will want to eventually implement three operations: insertion, removal, and search. Let’s start with search, since it doesn’t require changing the tree, and illustrates why search trees are defined the way they are. Suppose that we are looking for a key \( k \) in a tree rooted at \( v \). There are three cases:

1. If the tree is \texttt{null} (i.e., an empty tree), then \( k \) wasn’t in the tree, which we report (e.g., with an exception).

2. If \( v\.key == k \), then the key is at the root node, and we can just return the value at the root node, \( v\.value \).

3. If \( k < v\.key \), then the search tree property guarantees that \( k \) is either in the left subtree of \( v \), or not in the tree at all. So we recursively search in the left subtree.

4. In the final remaining case \( k > v\.key \), we recursively search for \( k \) in the right subtree \( T_2 \).

Notice the similarity between this search and Binary Search. We repeatedly query a “middle” value, then decide to recurse in one subpart or the other. Indeed, it is quite easy from a Search Tree to obtain a sorted array of all entries: perform an in-order traversal of the tree. The advantage of Search Trees over sorted arrays is that we can rearrange trees just a little bit at a time, rather than having to shift huge numbers of items around for every insertion. Thus, search in a Search Tree is as fast as Binary Search (which is very fast), but as we will see, insertions and removals are very fast, too.

So far, we have asserted that search in a Search tree is fast, and based on our example in Figure 1, it looks like it. As we do the analysis of the search algorithm above, we see that each step takes constant time, and increases the level we are searching by one. Thus, the time to search for an item is \( \Theta(\text{height}(T)) \). Whenever the height is small, say, \( \Theta(\log n) \), Search Trees will perform great. But will the height always be small? Consider the example in Figure 2.

![Figure 2: A degenerate search tree. Here, a search for the number “1” would take time linear in the number of elements, as it would just become a linear search.](image)

This tree doesn’t look like a tree at all — it’s basically a linked list. But of course, technically speaking, it is a tree. Its height is \( n \), the number of elements. While we haven’t really talked about how to insert
new elements into a tree, we can see how we would probably do it: search for a new key k, and if k isn’t in the tree yet, insert it at the place where the search terminated unsuccessfully. In that case, if the keys had arrived in the order 11, 10, 9, 8, 7, 5, 4, 2, we would in fact have built the tree from Figure 2. So the height could end up being \( \Theta(n) \) if we are not careful, in which case Search Trees would not perform any better than linked lists.

In order to keep the height small, ideally, our tree should be close balanced: for any node, the difference in heights of its subtrees should not be too big. That way, the tree will look almost like a complete binary tree, and we already proved earlier that a complete binary tree has height \( O(\log n) \). But how do we make sure a tree remains somewhat balanced?

We can perform an operation called a rotation, which shifts the elements in such a way that the search tree property is maintained, and the differences in subtree heights are lowered. For instance, in the example from Figure 2 after inserting 11, 10, 9, we should notice that we are already building a linked list, and move 10 to the root, with children 9 and 11. Then, after 8, 7 arrive (and are attached to the left of 9), we should rotate those around, and move 8 up, giving it subtrees 7 and 9. That way, the height stays at 3, and we would get the following tree instead.

![Figure 3: What happens if we keep rebalancing the tree as we insert the keys 11, 10, 9, 8, 7. The first figure shows the state after inserting 11, 10, 9. Then, we rotate the tree. Next, we insert 8, 7. Then, we rotate again the part that became unbalanced.](image)

Notice that a rotation does not rebuild the entire tree from scratch, as that would take \( \Theta(n) \). Instead, we are only moving pointers between a few nodes around, so each rotation just takes constant time. We will see a little more on Binary Trees and rotations later: the two standard examples are AVL trees and Red-Black Trees. However, an easier-to-understand way to obtain good Search Trees is based on a slightly different approach, and leads to B Trees (2-3 Trees). We will later see that 2-3 Trees can also be interpreted in terms of rotations, though perhaps not as easily.

## 4 B Trees (a.k.a., 2-3 Trees)

2-3 Trees are also search trees. They differ from the ones we have seen so far in that each node contains 1 or 2 keys. Unless it is a leaf, a node with 1 key has exactly 2 children, and a node with 2 keys has exactly 3 children. (There are also 2-3-4 trees, which we will talk about later. In those, each node has 1, 2, or 3 keys, and correspondingly 2, 3, or 4 children.) The defining properties of 2-3 Trees are the following:

- All leaves must be at the same level.
- For nodes with 1 key \( k \), the binary search tree property holds: all keys in the left subtree are smaller than \( k \), and all keys in the right subtree are greater than \( k \).
- For nodes with 2 keys \( k_1 < k_2 \), the three subtrees \( T_1, T_2, T_3 \) satisfy the following property:
  1. All keys in \( T_1 \) are smaller than \( k_1 \).
2. All keys in $T_2$ are strictly between $k_1$ and $k_2$.
3. All keys in $T_3$ are greater than $k_2$.

It will be our job in building these trees to ensure that the defining properties hold at all times. We will see later how to do that. We will also see how to do search, and why the depth of the tree is $O(\log n)$.

To implement a 2-3 Tree, we will need a slightly revised version of a node, to accommodate multiple keys and values. The class would look as follows:\footnote{This is slightly different from what I showed in lecture, where I used variables \texttt{leftKey} and \texttt{rightKey}. When you put them in an array, some of the operations we see later are actually easier to implement, hence the change.}

```cpp
template <class KeyType, class ValueType> class Node { 
    // variables could be public, or could add getters/setters
    int numberKeys;
    KeyType key[2]; // space for up to 2 keys
    ValueType value[2]; // space for the corresponding values
    Node* subtree[3]; // pointers to the up to 3 subtrees
}
```