Arrayed Implementation, Reading

Wait -- there's a problem. When we had a numeric index it was easy to associate the value with a position in the "values" array. Now that the index can be *anything*, how to we store that? *Where* do we store that? Do we even have a "values" array anymore? Let's rethink how we store values.

Conceptually, we're storing data in pairs -- *key-value* pairs. If we were to create a phone book object "on paper", we'd probably write a 2-column table, and label the first column as "name" (or key) and the second column as "phone number" (or value). Something like this:

key	value
Bill	9252341230
Jane	4155551212
Fred	6501110000
Alice	9257681234

If we came across a statement like **phoneBook["Rob"]=9259692416;**, we would (1) see if "Rob" is already in the table as a key, and if so, overwrite the value (that is, give Rob a new phone number), *or* (2) add another row if the key is not already in the list.

And maybe there should be a way to *remove* a row. Maybe something like **phoneBook.deleteKey("Bill");**, and a way to just see if a key exists already in the table without returning its value -- something like **phoneBook.containsKey("Jill")**, returning true or false.

Why couldn't we do these same things in an associative array template? Surely we can!

A Way To Store Key-Value Pairs

We can represent the table in an array, but each index position has to store *two* things -- the key *and* the value. And we'll need loops to traverse the array to find a matching key. (We may not like the big oh implications of that, but we'll worry about that in another module.) It may look something like this (with capacity 10):

Bill 9252341230	Jane 4155551212	Fred 6501110000	Alice 9257681234	? ?	? ?	? ?	? ?	? ?	? ?
0	1	2	3	4	5	6	7	8	9

Since it's possible for key-value pairs to be removed, there may be "holes" in the array:

~~Bill~~ ~~9252341230~~	Jane 4155551212	Fred 6501110000	Alice 9257681234	? ?	? ?	? ?	? ?	? ?	? ?
0	1	2	3	4	5	6	7	8	9

Unless we want to shift key-value pairs to fill holes, we need some way to mark indexes as "not in use", or else our traversal loops may find phantom matches (against uninitialized or removed and forgotten keys). A solution is to add an "inUse" Boolean to each position in the array:

Bill 9252341230 false	Jane 4155551212 true	Fred 6501110000 true	Alice 9257681234 true	? ? false	? ? false	? ? false	? ? false	? ? false	? ? false
0	1	2	3	4	5	6	7	8	9

A New Private Data Member

The array can no longer be **V* values;**, because a **typename K** and a **bool** are stored there, too. We could just make three arrays, and there's nothing wrong with that, but a more organized and manageable solution would be to create a struct (like the **struct Node** in the stack and queue linked list implementations) with three attributes, like this:

```
template <typename K, typename V>
class AssociativeArray
{
  struct Node
  {
    K key;
    V value;
    bool inUse;
  };

  Node* data;
  ...
```

V* values; gets replaced by **Node* data;**. It's an *array* of Nodes, not a linked list of Nodes -- note that very important difference.

Unique Keys Vs. Multiple Keys

We need to decide how we're going to deal with multiple phone numbers for the same person. Do we allow multiple rows for the same key, so that a key can appear more than once? Or do we change the **V Value;** attribute to an array or linked list of values? Or do we just not allow more then one value for any specific key?

Key-value data structures that limit one value per key have what are called "unique keys". That's what we'll be developing in this module, and they are the default for the language examples shown in the previous reading. Key-value data structures allow multiple values for a key are called "multi-key" structures. We will not be developing those in this course, although they are a rather straightforward extension to the logic we will develop for unique keys.