Unique Keys, Reading

Let's list our specification as we've developed it so far, as there are bits and pieces of it throughout previous readings and modules.

```
AssociativeArray<name, int> testScores; // key is a student name, value is a test score
testScores["John"] = 96; // store John's score as 96
testScores["John"] = 98; // change John's test score to 98
cout << testScores.size(); // output 1 as the number of unique key-value pairs stored
cout << testScores["John"]; // look up and output 98
if (testScores.containsKey("John")) cout << "John is found (and he should be)\n";
if (testScores.containsKey("Jane")) cout << "Jane is found (and she should NOT be)\n";
queue<string> studentNames = testScores.keys(); // return a list of all stored keys
testScores.deleteKey("John"); // now the size is zero
testScores.clear(); // if it was not zero already, it would be now!
```

There's one of these that seems to come from way out in "left field". It's the AssociativeArray::keys getter. Without a numeric index, there's no way to traverse the array! So this function returns an STL queue of all the stored keys. Traverse the queue to traverse the array.

Here are some more details that should fill in some blanks:

1. Use a default capacity, but allow it to be set by the programmer as a defaulted parameter in the main constructor.
2. Do not initialize keys or values in the main constructor, but *do* initialize the "inUse" flags to false.
3. **testScores["Fred"]** calls a setter function call that returns a mutable reference to Fred's score. If there is no record for "Fred", then this function call *should create one* and return it without initializing its value.
4. When new nodes are added, put them in the first unused node. If there are no unused nodes, *double the capacity*.
5. If deleteKey finds no matching key, do nothing.
6. Track size as a private data member, and increment and decrement in the square bracket setter and in deleteKey.
7. Track capacity, but allow no public access to its value or its setter.

Perhaps the most surprising of these is #3. Just referring to a value by its key *creates* a place for that key. That's normal in most languages' implementations of associative arrays. It has to be, because the function has no idea of the context in which it's being called, so as a setter it can't return a dummy if there is no matching key -- it might be being used in an assignment statement.

Limitations

The templated member functions are going to have to use loops to locate matching keys. They should use the equals-equals operator to see if keys match. This means that the **typename K** data type *must* support equals-equals. Int and string do, so no problem there. But the Student objects we wrote in Module 3 do not. As we now know them, Student objects cannot be keys.