Member Functions, Reading

Now that we have a specification, private data members to store values, and a search strategy using loops, we're ready to design templated member functions.

Main Constructor

Here's its prototype, allowing for a default capacity if the programmer does not specify it. We're choosing to start at 2:

```
AssociativeArray(int=2); // prototype
```

Here are the two possible ways to declare an associative array object, with and without overriding the default initial capacity (if a Student object could be a key).

```
AssociativeArray<string, unsigned long long> phoneBook; // accept default capacity
AssociativeArray<Student, int> finalExamScores(40); // override default capacity
```

The function itself needs to **allocate the dynamic array** of Node objects, set the **"inUse" flags to false**, and **initialize the capacity and size** tracking data member ints. The exact code for doing all this is left as an exercise for the student.

The Square Bracket Operator Getter

The prototype was shown in a previous reading:

```
V operator[ ](const K&) const; // getter
```

It should work like this -- a for-loop to traverse the whole "data" array, looking for a Node that is (1) in use and (2) has a matching key (using equals-equals, which the **typename K** is expected to support). If found, return the Node's value attribute. Otherwise we'll need to return... *something*.

What to return if there's no match... Since this is the getter and not the setter, it returns a *copy* instead of a reference to a data member. So we could just declare a local dummy and return it: **V dummy; return dummy;**. Note that this puts an additional restriction on **typename V** -- it has to be something that supports declaration without parameters! In object-oriented programming terms, that means it has to have either a "default constructor" *or* no constructors at all.

The exact code for doing all this is also left as an exercise for the student.

The Square Bracket Operator Setter

The prototype was shown in a previous reading:

```
V& operator[ ](const K&); // setter
```

It should work like this -- a for-loop to traverse the whole "data" array, looking for a Node that is (1) in use and (2) has a matching key. If found, return the Node's "value" attribute -- so far, exactly like the getter, except that it returns a mutable reference instead of a copy.

But if there is no match, we're going to *create an entry for the key*. Yes, the very act of "looking" for a key *creates* that key (if it's not there already).

We'll know that there is no match if we traverse the whole "data" array and find no in-use matching key. During the traversal we will either have skipped over Nodes that were not in use, or not. If not, we need more Nodes -- expand capacity. But if so, let's go back to the first found Node that was not in use, and use that one. For a parameter named **const K& key**, it could be something like this:

```
// find a matching key and return it's value
```

```
int indexOfFirstUnusedKey = cap; // =cap indicates no not in-use found (yet)
for (int index = 0; index < cap; index++) // where "cap" is a private data member
{
  if (data[index].inUse)
  {
    if (data[index].key == key) // K must support ==
      return data[index].value;
  }
  else if (indexOfFirstUnusedKey == cap) // no not in-use found yet
    indexOfFirstUnusedKey = index; // this is the first one
}

// if we get this far, no matching key was found
if (indexOfFirstUnusedKey == cap) capacity(2 * cap); // need more Nodes
data[indexOfFirstUnusedKey].key = key;
data[indexOfFirstUnusedKey].inUse = true;
++siz;
return data[indexOfFirstUnusedKey].value; // this is how the value gets set
```

containsKey Getter And deleteKey Setter

Here are the prototypes:

```
bool containsKey(const K&) const; // getter
void deleteKey(const K&); // setter
```

Both traverse the whole "data" array, skipping not in-use Nodes. For in-use Nodes, check to see if the Node's key attribute matches the parameter key. If not, continue the loop.

But if there is a match, using **typename K**'s equals-equals, then the getter should return true. The setter should (1) set the "inUse" attribute to false, (2) decrement the "siz" data member, and (3) either return using a **return;** statement, or break from the loop -- no point in looking for more matches because there won't be any.

If the end of the for-loop is reached without having returned using a **return** statement from inside the loop, then the getter should return false -- no matching key was found. The setter should just do nothing, as there's nothing to do.

size Getter And clear Setter

The getter is simply a return of the "siz" private data member -- it's short enough to be written inline. The setter should traverse the whole "data" array and set the "inUse" attributes to false, and then set "siz" to zero. No need to check if "inUse" is already false -- just set it to false no matter what, and avoid having to execute a compare and an if-statement.

The keys Getter

Here's the new function, made necessary because the main program has no way of traversing an associative array. Actually all the other languages that have associative arrays have something just like this. For example, PHP has a function named "array_keys". Python's is named "keys", just like ours!

Here's our prototype -- it will require a **#include** for the "queue" library in the H, and a **using namespace std;**:

```
queue<K> keys() const;
```

The function is a standard value-returning function. It should declare a queue object, traverse the "data" array and push the "key" attribute for in-use Nodes into the queue. After the loop ends, return that queue.

nodes into the queue. After the loop ends, return that queue.

The exact code for doing all this is also left as an exercise for the student -- it's an opportunity to use the **online documentation (http://www.cplusplus.com/reference/queue/queue/)** to figure out how to create a queue and push values into it.

Here's how to actually *use* the returned queue in the main program:

```
#include <iostream>
#include <queue>
using namespace std;

#include "AssociativeArray.h"

int main()
{
  AssociativeArray<string, int> phoneBook;
  ...

  // list all the keys
  for (queue<string> keys = phoneBook.keys(); !keys.empty(); keys.pop())
    cout << keys.front() << endl;
  ...

  // list all the keys and their values
 for (queue<string> keys = phoneBook.keys(); !keys.empty(); keys.pop())
   cout << "phoneBook[\"" << keys.front() << "\"] = "
     << phoneBook[keys.front()] << endl;
}
```

We call that for-loop a **"copy-pop" loop**. It's a clever way to traverse a stack or queue when those data structures don't allow looking through their contents. The first part of the for-loop copies the queue, the loop part "looks" at the front of the queue, and the last part of the for-loop "loses" the front. This continues until the copied queue is exhausted.

And yes, the main program should #include its own queue, and not depend on the included H to do that for it.

The Private capacity Setter

There's no need for a capacity *getter*, but the square bracket operator needs a private capacity *setter* function so that it can request doubling of the capacity when it needs to. The function should be exactly as it was for previous arrayed implementations, except that the array it replaces should be the new **Node* data;** instead of the old **V* values;**.