Chaining, Reading

Chaining works by allowing multiple Nodes at the same index. But the **Node data[CAP];** data member only allows for *one* Node at any index. What we need is a solution similar to our duplicate-checking solution to the DVC schedule history, where we had a data structure of data structure objects!

If we stored a *linked list* of Nodes at each index instead of a single Node, we could search the list of Nodes one-by-one at a given index for a matching key.

A good option for a object that represents a linked list of Nodes is the **STL list** **(http://www.cplusplus.com/reference/list/list/)** .

The C++ STL List

Its underlying data structure is a doubly-linked list, but that's transparent to the programmer. The programmer has this interface, as a little tutorial:

```
#include <list>
using namespace std;
...
struct Node
{
  K key;
  V value;
};
...
int main()
{
  list<Node> x; // an empty list
  ...
  // add a Node
  Node n = {key, value}; // a temporary Node...
  x.push_back(n); // ...copied to a more permanent location
  ...x.back().value // the V value attribute of the added Node

  // find a matching node
  typename list<Node>::iterator it; // use typename if this in inside a templated function
  for (it = x.begin(); it != it.end(); it++)
   if (it->key == key)
     break;

  if (it == x.end()) // not found
    ...
  if (it != x.end()) // found
  {
    it->value // is the value
    x.erase(it) // removes the key-value pair
```

There's no need for an "inUse" or "status" attribute, because if a Node is in the list all, it's in-use.

The Private Data Members

The data array used to store **Node**'s -- now it stores **list<Node>**'s. The list at any index is named **data[index]** instead of "x" as it is in the tutorial above.

So in all the searching and inserting and deleting, replace "x" with **data[index]**. The private data members now look like this:

```
template <typename K, typename V, int CAP>
class HashTable
{
  struct Node
  {
    K key;
    V value;
  };

  list<Node> data[CAP];
  int(*hashCode)(const K&); // alias for hash code function
  int siz;
  ...
```

The Member Function Templates With Key As Parameter

Here's how the templates change. Once the wrapped index is calculated, *that's* the list that the match will be in, if it's there at all. It's the list that will receive inserted Nodes, if any.

```
bool HashTable<K,V,CAP>::containsKey(const K& key) const
{
  int index = hashCode(key) % CAP;
  if (index < 0) index += CAP;
  typename list<Node>::const_iterator it; // getters need to use const_iterator
  for (it = data[index].begin(); it != data[index].end(); it++)
   if (it->key == key)
     break;
  if (it == data[index].end())
    ...not found...
```

It's left as an exercise for the student to write all the function coding. Note that getters should use **::const_iterator**, which is like a read-only pointer, and setters should use **::iterator**.

The Member Function Templates Without A Parameter

The clear and keys functions have no parameter and no way to compute an index. They are supposed to "touch" *all* lists in the data array, not just the one at "index".

So in a for-loop that spans all lists in the data array, the clear function clears each. It's left to the student to find the STL list documentation online and find out how to remove *all* values from a list.

The keys function should build *one* STL queue to return, using a for-loop to find and traverse each list, adding the keys to the queue.