Hashing, Reading

Now that we have a way to convert any key data type into an int, we can replace the for-loops in the member function templates, kind of like this:

```
V& AssociativeArray<K,V>::operator[ ](const K& key)
{
  for (int index = 0; index < cap; index++)
  {
    if (data[index].inUse)
    {
      if (data[index].key == key)
        ...found it!
```

```
V& HashTable<K,V,CAP>::operator[ ](const K& key)
{
  int index = hashCode(key);
  ...found it!
```

Remember that "hashCode" is a private data member -- an alias for a function that's defined in the main program's CPP. The hash code function is in the main program, and that's where the hash code is calculated. The desired index is computed in and owned by the templated class, where it is computed using the aliased hashCode function.

"Desired Index" Range Issues

Now we're going to have to deal with all those issues that we put off before. Remember how we converted "Joe" to a number, using the name's char's ASCII codes? We got 286. What if our Node array is not big enough to include that index? Or how about using Student objects as keys, and using their student IDs as their hash codes and "desired indexes"? DVC IDs are 7 digits long -- we certainly don't want to have our data array that size. And what if we ever calculate a *negative* index? In reality, the range for our desired index is the full range of int values: +/- 2 billion!

We need a way to make the computed desired index conform to the available capacity. One clever way to do that is to "wrap" it around the array. That is, if the capacity is 100 and the desired index is 99, fine -- it fits. But if the desired index is 123, subtract 100 and call it 23. If the desired index is 1234567, subtract 100, and *keep* subtracting 100 until it becomes 67. This new"wrapped index" is easily calculated like this:

```
  wrappedIndex = desiredIndex % CAP;
```

Just one more thing -- what about negative desired index values? The above would result in a *negative* wrapped index in the range -CAP to -1. So here's the solution:

```
V& HashTable<K,V,CAP>::operator[ ](const K& key)
{
  int index = hashCode(key) % CAP;
  if (index < 0) index += CAP;
  ...found it!
```

Problem solved!

But for this to really work well, it's *critical* that the range of wrapped index values span the available capacity, `CAP`. Because if the computed wrapped index for some application never yields values above 1000, for example, it does not do much good to have a capacity of 100,000.

"Wrapped Index" Collisions

The next issue is when different keys resolve to the same index -- like "Jane" and "Jean", using our adding of the char ASCII values idea. When different keys want to occupy the same index at the same time, that's called a "*collision*". If we store Jane's value at, let's say index 45, and then we go looking to see if there's a value stored for Jean, we're going to find Jane's value! The first thing we have to do is remember (1) whose key is associated with the value stored at an index, and (2) that there's even anything stored there at all. We need more checking:

```
V& HashTable<K,V,CAP>::operator[ ](const K& key)
{
  int index = hashCode(key) % CAP;
  if (index < 0) index += CAP;
  if (data[index].inUse && data[index].key == key)
    ...found it!
```

That would at least keep Jane's data secure, not to be confused with Jean's. But what about Jean? Jane got there first, and now there's no place for Jean's data. What now? ***How to we accommodate this collision*** and find a place for Jean?

There are ways to make such an accommodation. Some of them involve using an unused adjacent Node, and devising a search procedure for looking nearby when a matching key is not found at the *expected* index. Others involve stacking multiple Nodes at each index instead of just one, and searching the stack for a matching key. These fall into these general categories:

1. **Probing**, in which there is a formula for searching alternate indexed locations for a matching key when the key is not found at its calculated wrapped index.
2. **Chaining**, in which multiple Nodes are stored at each index, so that a matching key will always be located at its calculated wrapped index.

Of course, we're trying to maintain the O(1) that we were able to achieve with the hash code idea, and applying searching would seem to devalue that. But as long as the searches are kept short -- say, no more than 5 compares in a loop, all this does to the timing is to multiply by a constant multiplier. The scalability is not affected.