Linear Probing, Reading

"Probing" is what we call it when we start looking for alternate index locations for storing a key's data when it's calculated wrapped index is occupied. "Linear" probing is simply looking to the next higher index until an unused one is found. If none is found before reaching the capacity of the data array, start over at zero. If you loop back to the original, then the data structure is full, and we have bigger problems!

Other forms of probing have different schemes for how to proceed through the data array's indexes in search of a match.

The Linear Probing For-Loop

Here's how linear probing applies to the functions that previously used for-loops to find a matching key, using the **containsKey** getter in the example:

```
bool HashTable<K,V,CAP>::containsKey(const K& key) const
{
  int index = hashCode(key) % CAP;
  if (index < 0) index += CAP;
  for (int i = 0; i < CAP; i++, index++) // look through all Nodes
  {
    if (index == CAP) index = 0; // wraps to index zero
    if (!data[index].inUse)
      ...stop looking -- it's not here
    if (data[index].key == key)
      ...found it!
```

Yes, it looks like we're back to the AssociativeArray for-loops, but remember that we're **not supposed to have to go more than a few cycles** before we find a match *or* we stop looking. With the AssociativeArray, we cycled an average of n/2 times.

The reason we **stop looking** is that if we reach an unused Node, that's where the displaced key *should have been* placed.

The deleteKey Setter

This setter kind of upsets the whole idea of probing as we've considered it so far. Remember that we stop looking for a match when we find an unused Node. That's where the square bracket operators would return that Node's value (after the setter marks it as used and sets its key), the **containsKey** would return false, and the **deleteKey** would exit without doing anything further.

But what if **deleteKey** ever really did delete a key? That's a problem. What if we added Jane, then Jean. Jean, wanting to use the same index as Jane, would seek and find the next index after Jane's. **containsKey("Jean")** would check Jane's index, and not finding a key match at that in-use Node, it would look at the next higher index, finding Jean there.

But then we **deleteKey("Jane")**, marking that Node as not in-use. Now you see the problem -- **containsKey("Jean")** will check the index where Jean *should have been*, and finding the Node unused, concludes that Jean's not there, and stops looking!

Now, just because of this, we're back to searching the *whole* array just as we did with the AssociativeArray, O(n). Sigh...

A Status Indicator

Actually, if we were detectives, we would not give up so easily. A detective would check the Node to see if it *had ever been used* before deciding there's no need to keep looking, If Nodes were seats, the detective would be checking if the seat was warm or cold. What we could do is replace the **bool inUse;** attribute with an **int status;,** with different codes for in-use, no longer in-use, and never used. Whew -- back to O(1).

Rehashing

Just one remaining problem -- if there's a whole lot of deleteKey's going on, soon *no* Nodes will be "never used". To minimize that problem, the **deleteKey** function should periodically reschedule a "rehashing", where all in-use Nodes are copied to a temporary queue (or other data structure), the data array Nodes all set to "never used", and the queued nodes reinserted. It's kind of like the auto-resizing feature of DynamicArrays when they run out of capacity.