Problem 5.3

Iterator, n slots

An iterator could traverse a hash table, only looking at n slots, by starting at index 0 of the array that stores the data. hasNext() would return true if current pointed did not point to null.

This invariant would be maintained in next(), which would increment a variable storing the index of the current location of the iterator in the array and incrementing the location, peering into the slot, and progressing like such until it found a slot with an undeleted OAPair, at which point it sets the current to point to that OAPair and the location variable remains at the current index. If next() was called with the current location at the final index, then current will point to null and the location variable will point to -1. If it begins elsewhere but increments all the way to the final index of the array and it is null, then current will point to null and the location to -1.

With this implementation of iterators, no change at all is required to the already written methods in the class.

Iterator, k slots

The solution here is basically to superimpose a doubly linked list structure onto the hashtable. That is, the hashtable implemented in class will be augmented with attributes pointing to the first and last item in the collection (call them "start" and "end"). Furthermore, each OAPair will be augmented with attributes pointing to the OAPair added before (call this previous) and the one added after. Then, the iterator's next() will advance it to the OAPair pointed to by the current one's after variable. As in a linked list, the "before" of "start" and the "after" of "end" will be null. This implementation ensures that only k slots will be traversed because each pair's "before" and "after" only point to other non-empty, non-deleted slots (except for start and end).

This implementation does not change the runtime of any of the hashtable methods. lookup() is unchanged. insert just needs to do the additional constant work of changing the changing the inserted pair's reference to "before" to the end, its "after" to null, changing end's after referent to the inserted pair, and updating the referent of end from to the inserted pair. Basically, it must perform a constant addLast() to the doubly linked list. update() does not need to construct a new pair, but only change the value, so any references to it will remain. Finally, delete() needs to change the "before" and "after" referents of the deleted pair's "before" and "after," specifically to each other, and if the deleted pair is the start or end, that reference must be change as well. This is constant work as well.

Advantages

The first implementation has the advantage of being a rather transparent and pure implementation of hashtables, whereas the second one is obscured with the superimposed linked list. And while the second implementation does not affect the runtime of any of the methods in the hashtable (as discussed above), it does require insert and delete to perform additional work which may be noticeable if the hashtable is calling these methods a very large number of times. Of course, the second implementation has the obvious advantage of iterating through the entire hashtable in linear time with respect to k, the number of items in the hashtable, while the first implementation is linear with respect to n, the size of the data array, and n is never smaller than k, and usually considerable larger.