Hashing Assignment

As long as a low percentage of the elements of a hash table are occupied (that is, as long as the hash table has a low load factor), if the hash table is implemented appropriately then we can expect that on average storing and retrieving items in the hash table will take constant time, or time $O(1)$. However, when we modified the POGIL Activity's model code so that it returned None when an item not in the hash table was requested, this operation ran in time $O(n)$, where n represents the length of the hash table. In this assignment, you will implement a class FasterHashMap that on average runs in constant time even when asked to retrieve an item that is not contained in its hash table.

There are different ways to try to accomplish this task, but for this assignment you must take the following approach. First, notice that if there were no deletions, then when searching for an item we could stop the search as soon as we found an empty table element (an element with value `None`) rather than searching the entire table. This suggests that, when we delete an item, rather than leaving a "hole" where that item was we should look beyond it for an item that we might move into the hole. And, of course, after moving that item, we should look beyond it to see if there's an item that can be moved into the hole that the item has left by moving. If we do this sort of thing carefully, we guarantee that later, when searching for an item, the search can terminate if an empty element is reached before the item is found, just as would be the case if no deletions at all occurred.

Let's consider some examples of cases the deletion algorithm will need to handle. I'll assume a hash table of length 7 and use – to denote an empty element. I'll also assume integer keys and only show keys (no item values shown). I'll assume that we're using the `% 7` operation to map hashed keys into the table. And `h` will be the name of a FasterHashMap variable that has been initialized by appropriate calls to the setitem() method.

Example 1: Two adjacent items both hashing to the same index.

Initial table: `0 7 – – – – –`
`del h[0]`
Final table: `7 – – – – – –`

After key 0 is deleted, key 7 would not be found by a search ending at an empty element if there were an empty element at index 0. Therefore, key 7 must move into the "hole" left by removing key 0.

Example 2: Three adjacent items hashing to same index.

Initial table: `0 7 14 – – – –`
`del h[0]`
Final table: `7 14 – – – – –`
or `14 7 – – – – –`

After key 0 is deleted, keys 7 and 14 would not be found if there were an empty element at index 0. Therefore, this hole must be filled. Furthermore, the keys must remain adjacent. Although either of the options shown above would be correct, you'll probably find it easier to implement the first. The algorithm I suggest would in this example begin at one index past the "hole" at index 0, see that if the 7 moves into the "hole" it will be closer to (actually, at) its "home" location (the "home" location is index 0, where key 7 hashes to), move key 7 to the hole at index 1, then record that the hole is now at index 1 since key 7 moved out of this index, then check that 14 can move to this hole and be closer to its home, therefore move 14 into the hole at 1 and record that the hole is now at 2, then look at index 3 and see that there is nothing more to check because an empty element has been reached.

Example 3: Two items hashing to same index with an intervening item that is in its "home" position.

Initial table: `0 1 7 - - - -`
`del h[0]`
Final table: `7 1 - - - - -`

After key 0 is deleted, key 7 would not be found if there were an empty element at index 0. Therefore, this hole must be filled. But it must not be filled by moving key 1, since then key 1 would not be found. In the terms of the previous example, moving key 1 to index 0 would be moving it farther from its home position (1) rather than closer to it, so the algorithm will not make this move. So, key 7 must move into the hole left by removing key 0 and key 1 must remain at its position.

Example 4: Two items hashing to same index with two intervening items that should not move.

Initial table: `0 1 8 7 - - -`
`del h[0]`
Final table: `7 1 8 - - - -`

This is very similar to the previous example, but illustrates that the algorithm should handle the case of an item that does not move even though the key for that item is not in its home position (key 8 is at index 2 while its home is index 1). It would also be acceptable to reverse the 1 and 8 keys in the final table, but again, I think that that would result from a more complicated algorithm than necessary.

Example 5: Shifted version of previous example.

Initial table: `0 7 13 - - - 6`
`del h[6]`
Final table: `0 7 - - - - 13`

This is very similar to the previous example, but illustrates that the algorithm should handle the "wrapping" that is inherent when using modular arithmetic. In this case, the "next" index after 6 (next modulo 7) is 0.

In Example 4, it is relatively easy to see that the 1 and 8 keys should not move to index 0 because index 1 is the home index for keys 1 and 8 and therefore moving to index 0 would move these keys farther from this home (in some sense, move the key "behind" the home rather than at or in front of it). So the logical test for whether or not to move a key is relatively simple in the case when the item being removed hashes to index 0: if the hole index is greater than or equal to the home index, move the key. And although it might be surprising, the test is not all that much more complicated when the index of the item being removed is nonzero. The idea is that we can subtract the index value (mod 7) and "shift" the hole and home values in order to test as if the item being removed was at index 0.

Let's see how this works in Example 5. The item being removed is at index 6, so this is where the initial hole will be in the hash table. But, for purposes of deciding whether or not an item should move into this hole, we will use the value (6-6) mod 7 = 0 as the shifted location of the hole. The home index for key 0 is 0, but the shifted location of the home is (0-6) mod 7 = 1. Since it is not true that 0 >= 1, we can see that key 0 should not move into the hole. And since the shifted home for key 7 is also (0-6) mod 7 = 1, key 7 should also not move. But the shifted home for key 13 is (6-6) mod 7 = 0, and since 0 >=0 moving key 13 to the hole will move it closer to its home (actually, it moves exactly to its home in this case). We therefore move key 13.

Example 6: Extension of the previous example illustrating multiple moves and non-moves.

Initial table: `0 7 13 3 2 - 6`

```
del h[6]
```

Final table: `0 7 2 3 - - 13`

Now the key 13 moved as before, leaving a hole at index 2. Since key 3 is already at its home index, it should not move. Technically, using the shifting algorithm described above, key 3 does not move because the hole left by moving key 13 is at shifted position (2-6) mod 7 = 3, the shifted home position for key 3 is at (3-6) mod 7 = 4, and it is not true that 3 >= 4. However, by similar reasoning, the key 2 can and should move to the hole at index 2.

That covers the getitem() and delitem() operations. The setitem() operation should operate as modified for POGIL Activity question 15: It should add the item to the hash table if the key is not already in the table and should modify the value of the existing item otherwise.

Note: You can assume that the hash table will never be full, that there will always be at least one empty cell. So, there is no need to (and you should not) maintain a counter in order to avoid an infinite loop when searching for a key.