Lesson 25: Hash Tables using Buckets

In the previous lesson you learned about the concept of hashing, and how it was used in an open address hash table. In this lesson you will explore a different approach to dealing with collisions, the idea of hash tables using buckets.

A hash table that uses buckets is really a combination of an array and a linked list. Each element in the array (the hash table) is a header for a linked list. All elements that hash into the same location will be stored in the list.

Each operation on the hash table divides into two steps. First, the element is hashed and the remainder taken after dividing by the table size. This yields a table index. Next, the linked list indicated by the table index is examined. The algorithms for the latter are very similar to those used in the linked list. For example, to add a new element is simply the following:

```c
void addHashTable(struct HashTable *ht, TYPE val) {  
    /* Compute hash value to find the correct bucket. */
    int idx = HASH(val) % ht->size;

    if (idx < 0) idx += ht->size;

    addList(ht->table[idx], val);   /* Add to bucket. */
    ht->cnt++;
    /* Note: later might want to add resizing the table (below). */
}
```

The `contains` test is performed as a loop, but only on the linked list stored at the table index. The removal operation is the most complicated, since like the linked list it must modify the previous element. The easiest way to do this is to maintain a pointer to both the current element and to the previous element, as you did in Lesson 32. When the current element is found, the next pointer for the previous is modified.

As with open address hash tables, the load factor ($\lambda$) is defined as the number of elements divided by the table size. In this structure the load factor can be larger than one, and represents the average number of elements stored in each list, assuming that the hash function distributes elements uniformly over all positions. Since the running time of the `contains` test and removal is proportional to the length of the list, they are $O(\lambda)$. Therefore the execution time for hash tables is fast only if the load factor remains small. A typical technique is to resize the table (doubling the size, as with the dynamic array and the open address hash table) if the load factor becomes larger than 10.

Complete the implementation of the `HashTable` class based on these ideas.

```c
struct HashTable {
    struct List **table;    /* Hash table → array of lists. */
    int cnt;                 /* Count of elements in the hash table. */
    int size;                /* Hash table size. */
};
```
void initHashTable(struct HashTable *ht, int size) {
    int i;

    ht->size  = size;
    ht->cnt   = 0;
    ht->table = (struct List **)malloc(size * sizeof(struct List *));
    assert(ht->table != 0);

    for (i = 0; i < size; i++) ht->table[i] = newList();
}

int sizeHashTable(struct HashTable *ht) { return ht->cnt; }

void _resizeTable(struct HashTable *ht) {
}

void addHashTable(struct HashTable *ht, TYPE val) {
    /* Compute hash value to find the correct bucket. */
    int idx = HASH(val) % ht->size;
    if (idx < 0) idx += ht->size;

    addList(ht->table[idx], val);  /* Add to bucket. */
    ht->cnt++;
}

int containsHashTable(struct HashTable *ht, TYPE val) { 
}
void removeHashTable(struct HashTable *ht, TYPE val) {
}

On Your Own

1. What is a bucket?

2. How are these hash tables similar to those used in open address hashing? How are they different?

3. What is the definition of the load factor for a hash table? Assuming that the hash function in use distributes the elements evenly over all buckets, what is another interpretation of the load factor?

4. Explain how the hash table combines features of an array and a linked list.

5. Suppose you wanted to test the hash table abstraction. What would be good boundary test cases? Write a test harness to feed these test values into the hash table methods and verify the result.

6. Would it make sense to use a different data structure, such as an AVL tree, for the buckets? What would be the advantage of this approach? What would be the disadvantage?

7. An iterator for the hash table class must produce all elements from every bucket. This combines features of both the dynamic array and the list iterator. Provide an implementation of this class. As with most iterators, the remove operation is the most complicated.

The bucket approach to hash tables is the most common form of this data structure. It is this technique that is used in the hash tables found in the Java standard library. There are three of these. The first, a HashSet, is similar to the data structure shown here. The HashMap and the Hashtable use the same technique, but provide a map-like interface.