Worksheet 38: Hash Tables using Buckets

In Preparation: Read Chapter 12 to learn more about hash tables. If you have not done so already, complete Worksheet 37 on open address hashing.

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, 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:

```
void addHashTable (struct hashTable * ht, TYPE newValue) {

        // compute hash value to find the correct bucket
    int hashIndex = HASH(newValue) % ht->tablesize;
        if (hashIndex < 0) hashIndex += ht->tablesize;

        struct link * newLink = (struct link *) malloc(sizeof(struct
link));
        assert(newLink);

        newLink->value = newValue;
        newLink->next = ht->table[hashIndex];
        ht->table[hashIndex] = newLink; // add to bucket
        ht->count++; // 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 (l) 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(l).
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 vector 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.

```
struct hlink {
        TYPE value;
        struct hlink *next;
};

struct hashTable {
        struct hlink **table;
        int tableSize;
        int count;
};

void initHashTable (struct hashTable * ht, int size) {
   int i;
   assert (size > 0);
```

```
   assert (ht != NULL);

   //allocate table array of hash links
   ht->table = (hlink **) malloc(size * sizeof(hlink *));
   assert(ht->table != 0);

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

   ht->tablesize = size;
   ht->count = 0;
}

int hashTableSize (struct hashTable * ht) { return ht->count; }

void hashTableAdd (struct hashTable *ht, TYPE newValue) {

        int hashIndex = HASH(newValue) % ht->tablesize; //compute hash
value
        if (hashIndex < 0) hashIndex += ht->tablesize;

        struct link * newLink = (struct hlink *) malloc(sizeof(struct
hlink));
                assert(newLink);

        newLink->value = newValue;
        newLink->next = ht->table[hashIndex];
        ht->table[hashIndex] = newLink;                            //
add to bucket
        ht->count++;

        if ((ht->count / (double) ht->tablesize) > 8.0)
                _resizeHashTable(ht);
}

int hashTableContains (struct hashTable * ht, TYPE testElement) {
        // compute hash value to find the correct bucket
        int hashIndex = HASH(testElement) % ht->tablesize;
        struct hlink *curLink;

        if (hashIndex < 0)
                hashIndex += ht->tablesize;

        //traverse list and seek testE
        curLink = ht->table[hashIndex];
        while (curLink != 0) {
                if (curLink->value == testElement)
                        return 1;
                curLink = curLink->next;
```

```
        }
        return 0;
}


void hashTableRemove (struct hashTable * ht, TYPE testElement) {
        int hashIndex = HASH(testElement) % ht->tablesize;
        struct hlink *curLink;
        struct hlink *lastLink;

        if (hashIndex < 0)
                hashIndex += ht->tablesize;

        if (hashTableContains (ht, testElement) {
                curLink = ht->table[hashIndex]
                lastLink = ht->table[hashIndex]
                while (curLink != 0) {
                        if (cur->value == testElement) {
                                lastLink->next = curLink->next; //
unlink
                                free(curLink;
                //free
                                curLink = 0;
                //exitLoop
                        }
                        else {
                                lastLink = curLink;
                                curLink = curLink->next;
                        }
                }
                ht->count--;
        }
}

void resizeTable (struct hashTable *ht) {
        assert (ht != NULL)
        int i, idx;
        int newSize = ht->tablesize * 2;

        //malloc new dynamic array of pointers to links
        hlink **newTable = (hlink **) malloc((newSize) * sizeof(hlink
*));
                assert(newTable != 0);

        for (i = 0; i < newSize; i++)
                newTable[i] = 0;
        // or replace with: initHashTable(newTable, newSize);

        //copy and rehash elements
        hlink *currentLink,
```

```
                *tempLink;
        for (i = 0; i < ht->tableSize; i++) {
                currentLink = ht->table[i];
                while (current != 0) {
                        //get new index
                        idx = HASH(current) % newSize;
                        if (idx < 0)
                                idx += newSize;
                        // add the value to new hash
                        tempLink = currentLink;
                        currentLink->next = newTable[idx];
                        newTable[idx] = currentLink;

                        currentLink = tempLink->next;
                }
        }
        //free old table, set new table, set size
        free(ht->table)
        ht->table = newTable
        ht->tablesize = newSize
}
```