Worksheet 37: Hash Tables (Open Address Hashing)

In preparation: Read Chapter 12 on dictionaries and hash tables.

In this chapter we will explore yet another technique that can be used to provide a very fast Bag abstraction. We have seen how containers such as the skip list and the AVL tree can reduce the time to perform operations from O(n) to O(log n). But can we do better? Would it be possible to create a container in which the average time to perform an operation was O(1)? The answer is both yes and no.

In chapter 12 you learned about the idea of hashing. To hash a value means to convert it into an integer index. This index is then used to access an array. Unfortunately, two different values may result in the same index. This is termed a collision. There are several different techniques used to deal with the problem of collisions. These will be explored in this and subsequent lessons.

The first technique you will explore is termed *open-address hashing*. Here all elements are stored in a single large table. Positions that are not yet filled are given a **null** value. An eight-element table using Amy's algorithm would look like the following:

0-aiqy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gow	7-hpx
Amina			Andy	Alessia	Alfred		Aspen

Notice that the table size is different, and so the index values are also different. The letters at the top show characters that hash into the indicated locations. If Anne now joins the club, we will find that the hash value (namely, 5) is the same as for Alfred. So to find a location to store the value Anne we *probe* for the next free location. This means to simply move forward, position by position, until an empty location is found. In this example the next free location is at position 6.

0-aiqy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gow	7-hpx
Amina			Andy	Alessia	Alfred		Aspen

Now suppose Agnes wishes to join the club. Her hash value, 6, is already filled. The probe moves forward to the next position, and when the end of the array is reached it continues with the first element, eventually finding position 1:

0-aiqy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gow	7-hpx
Amina			Andy	Alessia	Alfred		Aspen

Finally, suppose Alan wishes to join the club. He finds that his hash location, 0, is filled by Amina. The next free location is not until position 2:

0-aiqy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gow	7-hpx
Amina			Andy	Alessia	Alfred		Aspen

To see if a value is contained in a hash table the test value is first hashed. But just because the value is not found at the given location doesn't mean that it is not in the table. Think about searching the table above for the value Alan, for example. Instead, an unsuccessful test must continue to probe, moving forward until either the value is found or an empty location is encountered. (We will assume that our hash table contains pointers to elements, so that an empty position is indicated by an empty pointer).

Removing an element from an open hash table is problematic. We cannot simply replace the location with a null entry, as this might interfere with subsequent search operations. Imagine that we replaced Agnes with a null value in the table given above, and then once more performed a search for Alan. What would happen?

One solution to this problem is to not allow removals. This is the technique we will use. The second solution is to create a special type of marker termed a *tombstone*. A tombstone replaces a deleted value, can be replaced by another newly inserted value, but does not halt the search.

Complete the implementation of the HashTableBag based on these ideas. The initial size of the table is here fixed at 17. The data field tablesize holds the size of the table. The table should be resized if the load factor becomes larger than 0.75. Because the table contains pointers, the field in the struct is declared as a pointer to a pointer, using the double star notation. The variable count should represent the number of elements in the table. The macro HASH is used to compute the hash value. The calculation of the hash index is performed using long integers. The reason for this is explored in Chapter 12.

```
  struct openHashTable {
 TYPE ** table;
 int tablesize;
 int count;
 };
 void initOpenHashTable (struct openHashTable * ht, int size) {
        int i;
        assert (size > 0);
        ht->table = (TYPE **) malloc(size * sizeof(TYPE *));
        assert(ht->table != 0);
        for (i = 0; i < size; i++){
                ht->table[i] = 0; /* initialize empty */
                ht->tablesize = size;
                ht->count = 0;
        }
 }
 int openHashTableSize (struct openHashTable *ht){
 return ht->count;
 }
 void openHashTableAdd (struct openHashTable * ht, TYPE *newValue) {
        int idx;
        /* Make sure we have space and under the load factor threshold*/
        if ((ht->count / (double) ht->tablesize) >0.75)   _resizeOpenHashTable(ht);
        ht->count++;
        idx = HASH(newValue) % ht->tablesize;
        /* To be safe, use only positive arithmetic.   % may behave very differently on diff implementations or diff
languages . However, you can do the following to deal with a negative result from HASH */
        if (idx < 0) idx += ht->tablesize;
        int i = 0;
	while (i != idx){
		if(ht->table[(i+idx)%ht->size] == NULL || ht->table[(i+idx)%ht->size] == "_TB_")
			ht->table[(i+idx)%ht->size] = newValue;
		i++;
	}
}
int openHashTableBagContains (struct openHashTable *ht, TYPE testValue){
	int idx;
	idx = HASH(newValue) % ht->tablesize;
	if (idx < 0) idx += ht->tablesize;
	int i = 0;
	while (i != idx){
		if(ht->table[(i+idx)%ht->size] == NULL)
			return 0;
		if(ht->table[(i+idx)%ht->size] == testValue)
			return 1;
		i++;
	}
	return 0;
}
void _resizeOpenHashTable (struct openHashTable *ht) {
	struct * newTable = (struct * newTable) malloc(size * sizeof(TYPE *));
	initOpenHashTable (newTable, ht->size * 2);
	for (int i = 0; i < ht->size; i++){
		openHashTableAdd(newTable, ht->table[i]);
	}
}
```