**CMPS 12B**
**Introduction to Data Structures**
**Programming Assignment 5**
**Hash Table Implementation**

In this project you will create another version of a Dictionary ADT based on a hash table instead of a linked list as you created in assignments 1 & 2. A Dictionary ADT supports the operations INSERT, DELETE and MEMBER. You can read section 13.2 on hash tables in the textbook (starting at page 737) for background on hashing.

So far we have seen two data structures that can form the basis of a Dictionary ADT, namely linked lists and binary search trees. In the linked list implementation, the worst case run time of the essential Dictionary operations (insert, delete, and lookup) were all in $\Theta(n)$, where $n$ is the number of pairs in the dictionary. In the binary search tree implementation, the Dictionary operations all run in time $\Theta(\log(n))$, provided that the underlying binary search tree is "balanced". It is possible to do better still using a hash table as the underlying data structure. In this implementation the Dictionary operations will run in constant time $\Theta(1)$. The catch is that this is the *average case* run time. The worst case run time of the Dictionary operations in a hash table implementation can be as bad as $\Theta(n)$, the time for a linked list.

**Hash Tables**
A hash table is simply an array that is used to store data associated with a set of keys. In our Dictionary ADT we wish to store a set of (`key`, `value`) pairs where `key` and `value` are C strings (i.e. null `'\0'` terminated `char` arrays). To simplify our discussion we will suppose for the moment that the keys are integers rather than strings. Actually it is not difficult to turn a string into an integer.

If the keys all happen to be in the range 0 to $N-1$, and $N$ is not too large, one can simply allocate an array of length $N$, and store the value $v$ in array index $k$. This arrangement is called a *direct-address table*. Accessing a pair ($k$, $v$) then has constant cost. The difficulty with direct addressing is obvious: if $N$ too is large, allocating an array of length $N$ may be impractical, or even impossible. Think of an application in which `key` is an account number and `value` is an account balance. Such applications often have account numbers consisting of 15 or more decimal digits, which makes the universe $U$ of *all possible keys* very large. Furthermore the set $S$ of *keys actually used* may be so small relative to $U$ that most of the space allocated for the array would be wasted. In the case of account numbers of 15 decimal digits, that is a universe of possible keys on the order or one quadrillion. That's a very long array, and even if everyone on the planet has an account, over 99% of the space is unused.

A hash table $T$ requires much less storage than a direct address table. Specifically the storage requirements for a hash table can be reduced to $\Theta(|S|)$, while maintaining the benefit that the dictionary operations run in (average case) constant time $\Theta(1)$. To do this we use a *hash function h* to compute the index (or *slot*) $h(k)$ where a given pair (`k`, `v`) will be stored. A suitable hash function must map the universe $U$ of possible keys to the set $\{0, 1, \ldots, m-1\}$ of array indices:

$$h : U \to \{0, 1, \ldots, m-1\}$$

We say that the pair (`k`, `v`) *hashes to* the slot $h(k)$ in the hash table $T[0\ldots(m-1)]$, and that $h(k)$ is the *hash value* of key $k$. The point of the hash function is to reduce the range of array indices that must be handled. Instead of $|U|$ indices, we need handle only $m$ indices. Storage requirements are thereby reduced.

There is of course one problem: two keys may hash to the same slot. We call this situation a *collision*. Fortunately, there are effective techniques for resolving the conflict created by collisions. The ideal solution would be to avoid collisions altogether. We might try to achieve this goal through our choice of hash function $h$. One possibility is to make $h$ appear to be random, thus avoiding collisions by making them improbable. The very term "hash", which evokes images of random mixing and chopping, captures the spirit of this approach. Of course, a hash function $h$ must be deterministic in the sense that a given input $k$ always produces the same output $h(k)$. Since $|U| > m$, and in general $|U|$ is *much larger* than $m$, there must be at least two keys that have the same hash value, and therefore avoiding collisions altogether is impossible. Thus, while a well designed random looking hash function can minimize the number of collisions, we still need a method for resolving the collisions that do occur.

Several methods for resolving collisions will be discussed in class. The method that we will use in this project is called *chaining*, and is perhaps the simplest collision resolution technique. In chaining, we put all the pairs that hash to the same slot into a linked list. Thus the hash table $T[0\ldots(m-1)]$ is an array of linked lists. More precisely, $T[j]$ is a pointer to the head of a linked list storing all pairs that hash to slot $j$. If there are no such elements, $T[j]$ will be `NULL`. The Dictionary ADT operations on a hash table $T$ are easy to implement when collisions are resolved by chaining. To insert a pair (`k`, `v`) into the Dictionary, we create a new Node storing this pair, then insert that Node at the head of the linked list $T[h(k)]$. To lookup a given key `k`, we do a linear search of the list $T[h(k)]$, and return the corresponding `value` if found, or `NULL` if not found. To delete the pair with key `k`, simply splice the corresponding Node out of the list headed by $T[h(k)]$. The remaining Dictionary operations are equally simple and are left to the student to design.

At this point the only question left is what to choose as our hash function $h$. A good hash function should satisfy (at least approximately) the assumption of *simple uniform hashing*: each key is equally likely to hash to any of the $m$ slots, independently of the slot to which any other key has hashed. Unfortunately, it is often not possible to check this condition in practice, since one may not know the probability distribution on the universe $U$ from which the keys are drawn. Furthermore the keys may not be drawn independently from $U$, i.e. the next key used may depend (probabilistically) on the previous key.

In this project $U$ will be the set of non-negative integers up to the maximum value of the `unsigned int` data type in C (typically $2^{32}-1$ in most implementations.) We will assume that keys are uniformly distributed over this universe. (We will try to enforce this assumption, at least approximately, by the way we turn strings into `unsigned ints`.) Under these conditions the function

$$h(k) = k \bmod m$$

satisfies the simple uniform hashing condition. Here $m$ is the length of the hash table $T$, and **mod** denotes the remainder operation, i.e. $h(k)$ is simply the remainder of $k$ upon division by $m$. Note that this quantity necessarily lies in the range from 0 to $m-1$, as required. Other possible hash functions with their pros and cons, will be discussed in class. In our implementation we will build the has table from the size specification from the client that constructs the hash table. It is suggested that your `DictionaryClient.c` file contain a constant integer

```
const int tableSize=101; // or some prime other than 101
```

establishing the length of the hash table array. The choice here of length 101 is somewhat arbitrary, and should be changed during testing of your project. In particular, you should see how your Dictionary ADT performs with small table lengths, i.e. no more than half the number of pairs in the Dictionary, so that collisions are guaranteed. You will include the following functions unaltered in your Dictionary.c file.

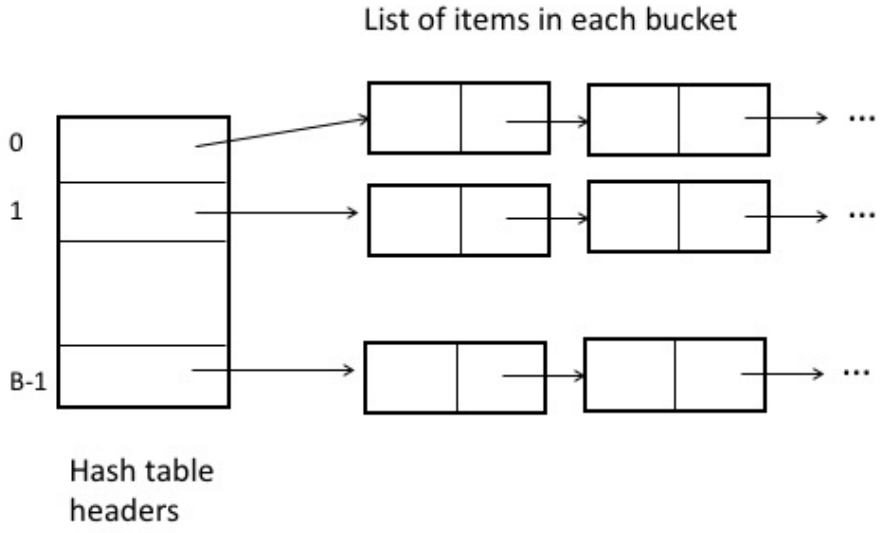
```
// rotate_left()
// rotate the bits in an unsigned int
unsigned int rotate_left(unsigned int value, int shift) {
   int sizeInBits = 8*sizeof(unsigned int);
   shift = shift & (sizeInBits - 1);
   if ( shift == 0 )
      return value;
   return (value << shift) | (value >> (sizeInBits - shift));
}

// pre_hash()
// turn a string into an unsigned int
unsigned int pre_hash(char* input) {
   unsigned int result = 0xBAE86554;
   while (*input) {
      result ^= *input++;
      result = rotate_left(result, 5);
   }
   return result;
}

// hash()
// turns a string into an int in the range 0 to tableSize-1
int hash(char* key, int tableSize)){
   return pre_hash(key)%tableSize;
}
```

Functions `rotate_left()` and `pre_hash()` turn a string into an `unsigned int`, and function `hash()` converts that number into an `int` in the range 0 to tableSize – 1. We will discuss the operational details of these functions, but try to figure out how they work on your own by consulting the C language and other sources such as Google and stack-overflow.

For this assignment we are using open hashing and the data structure is visualized as follows:



In the Dictionary.h file you'll notice that the declaration of the hash table is a variable length array which means that the number of elements is not established at compile time. You as the programmer must enforce array boundaries and allocate the correct amount of space. For example, use this code to allocate the hash table:

```
HashTableObj *
newHashTable(int size) {

// error checking code omitted
// we explicitly allocate space for the fixed size element in HashTableObj,
// the size element plus the space for the variable size array

H = (HashTableObj *) malloc(sizeof(H->size) + sizeof(bucketList)*size))
```

The size of the `HashTableObj` is determined at run time. We will do the similar malloc for the `BucketListObj` which has a fixed size element `next` and then a variable character array `item`.

**What to turn in**

Write the implementation file `Dictionary.c` as described above based on a hash table data structure. Also write your own set of test procedures in a file called `DictionaryTest.c`. You will be provided a Dictionary.h file and starter `DictionaryClient.c`, which you should submit unaltered with your project. Of course, you will write several DictionaryClient.c files to completely test your implementation, but will not submit them, only turn in the DictionaryClient.c file that was provided. These files will be available through Canvas.

A Makefile will also be provided which you may alter as needed. To receive full credit, your project must compile without errors or warnings, produce correct output, and cause no memory leaks when run under `valgrind`. Your assignment5 directory should contain the files:

```
README
Dictionary.c
Dictionary.h
DictionaryTest.c
DictionaryClient.c
Makefile
```

Submit a zip file named assignment5.zip containing a folder named assignment5 as above.