CMPS 12B
Introduction to Data Structures
Winter 2009

Programming Assignment 4

Due Sunday March 15, 10:00 pm

In this project you will again recreate the Dictionary ADT from pa2 and lab5, again in C, but now based on a hash table instead of a linked list. Start by reading the section on hash tables in chapter 13 of the book (pages 689 through 719.)

So far we have seen two data structures which 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) are 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 that for a linked list, namely $\Theta(n)$.

Hash Tables

A hash table is simply an array which is used to store the 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 non-negative integers. 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 pair (k,v) in array index k. This arrangement is called a *direct-address table*. The difficulty with direct addressing is obvious: if N too is large, storing 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 between 10 and 15 decimal digits, which makes the set of all possible keys very large indeed. Furthermore the set S of keys actually stored may be so small relative to the universe U of all possible keys, that most of the space allocated for the array would be wasted.

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) 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. Thus a suitable hash function must map the universe U of all possible keys to the set {0, 1, …, $m-1$} 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 $T[0\ldots(m-1)]$. We also say 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 need to be handled. Instead of $|U|$ indices, we need to handle only m indices. Storage requirements are correspondingly 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 or at least minimizing their number. 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* should always produce 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 which 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 which stores all the pairs which 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 `UNDEF` 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*. In this project *U* will be the set of non-negative integers {0, 1, 2, …}, and we will assume that keys are uniformly distributed over this universe. Under these conditions the function

$$h(k) = k \,\mathbf{mod}\, m$$

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*. Observe that this quantity necessarily lies in the range from 0 to $m-1$, as required. This is the function you will use to compute hash values in this project. Other possible hash functions and their pros and cons, will be discussed in class. It is suggested that your `Dictionary.c` file contain a private constant integer

```
const int tableSize=100;
```

establishing the length of the hash table array. The choice here of length 100 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. In C the remainder operation is denoted by the percent symbol: `%`. Thus `Dictionary.c` should also contain a private function which implements the hash function above:

```
int hash(int k){ return( k%tableSize ); }
```

This function will be used to compute the array index into which a pair with key `k` is to be stored. The header file for this project is identical to that found in lab5, and also to the header file for the Binary Search Tree based Dictionary posted on the website. There is however one ADT operation whose requirements are relaxed form those in lab5, namely the `printDictionary` function. Recall that in pa2, and therefore also in lab5, a text representation of a Dictionary was required to present (`key`, `value`) pairs in the order in which they were originally inserted into the Dictionary. This was easy and natural for the linked list representation since the list itself maintained the insertion order. Your `printDictionary` function in this project should just print out pairs in the order that they appear in the table, i.e. print out list $T[0]$ in order, then list $T[1]$, …, then end by printing list $T[m-1]$ in order.

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 `DictionaryTest.c`. The webpage includes the files `Dictionary.h` and `DictionaryClient.c`, which should be submitted unaltered with your project. Notice that the version of `DictionaryClient.c` provided here is more complex than the one in lab5. This version takes two command line arguments giving the names of an input file and an output file respectively. You can easily infer the input and output file formats by examining the code in `DictionaryClient.c`. Three correctly formatted input files together with their corresponding output files are also provided on the webpage. *Do not* turn in these files with your project. They are provided only so that you can test your project on relatively large dictionaries. If you set the `tableSize` parameter in your `Dictionary.c` file to be 50, your output files should look identical to the ones provided. If you change the `tableSize` parameter to something other than 50, the Dictionary will most likely look different when printed, but you should get identical results for lookups and deletes. Again, study the code in `DictionaryClient.c` to see what is going on.

A makefile is also provided on the webpage which includes clean and check utilities, and should work for your project, almost unchanged. The check command has been set to run `DictionaryClient` on the input file `in3` and write to the output file `junk3`. This output file name was chosen so as not to overwrite `out3` if it already exists in your pa4 directory. You can alter the check command as you see fit to test your project. To receive full credit, your project must produce no memory leaks when run under `bcheck`. Note also that this makefile should pose no problems with respect to the particular version of make being used, since it uses no wildcards to generate lists. Also included on the webpage is a Perl script which you can use to create correctly formatted random input files of any size for this project. Submit the files:

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

to the assignment name `pa4`. As always start early and ask plenty of questions.