Chapter 45 - Hashing

Thus far, the fastest means we have explored for inserting an item into a data structure is the insertion into an unordered list, where insertions of one object can be accomplished in $O(1)$ time and $n$ objects in $O(n)$ time. The penalty for the using an unordered list is the difficulty in locating or extracting items from the list, which is $O(n)$ in time for one item and $O(n^2)$ for $n$ items. We have looked at binary trees where the insertion and location time is potentially $O(n \log_2 n)$, but only if the actual number of items in the tree is close to the same as the maximum possible number of items in a tree of the given depth. Keeping a tree balanced is not a topic that has been covered. Balancing a tree is not a simple idea and has its own cost in time beyond that of insertion and location.

Hashing is a set of techniques that allows the placement of an item into or its location in or removal from a data structure in potentially $O(1)$ time and $n$ objects in $O(n)$ time.

1. Hash Tables and Hash functions

A hash table is a set of elements referred to as buckets, each bucket having a unique address. Each bucket can hold at least one item. In its simplest form, a hash table can be implemented as an array where each element in the array is a bucket. At its most complex, hash tables can be made up of buckets that are themselves complex data structures and are capable of holding many items.

A hash function is a function that, given a key value of an item, produces the address of the appropriate bucket in a hash table for that item. The most efficient hash functions produce a unique address for each key.

In practice, hash tables are seldom large enough to accommodate all possible keys. In most cases, hash tables that would be large enough to accommodate every possible key would have considerable empty buckets - a waste of system resources. In practice, the number of buckets in a hash table is usually reduced to avoid a large number of empty buckets and hash functions must produce the same address for multiple keys.

2. Collisions

A collision occurs when an item is hashed into the same bucket as another item. How this is handled depends upon the bucket. If the bucket is the same type as an item, a bucket can handle only one item and in the event of a collision, an alternative bucket for the new item must be found. If the bucket is more complex, for example the bucket is a pointer to another data structure, both or even many more items can be stored in the bucket.

In the case of a bucket being the same type as the item, one solution is to find the nearest empty bucket to the bucket with the correct address. If there are many empty buckets in the hash table, this may not be a bad strategy. If there are few empty buckets in the hash table, this could lead to a long linear search for many items, either to find a bucket to store the item in or to locate it later.
Another strategy that can be used in the case of a bucket being the same type as the item is to re-hash with another hash function. This can be an effective strategy if there are few collisions with adequate empty buckets.

However collisions are handled, each time there is a collision, hashing begins to degrade from its potential perfection in time. This is not necessarily bad. For example, hashing into an array of sorted linked lists can produce many linked lists. If the hash function is chosen well, the longest list will not be much longer than the number of items to hash divided by the number of buckets. While inserting an item into or locating an item in a sorted linked list can be an O(n) operation, if the number of buckets is large enough and the hash function distributes the items fairly evenly among the buckets, the size of the maximum number of items in the list of any given bucket may be substantially closer to 1 than to the number of items to be hashed. Thus the expected case of inserting, locating or removing an item from a well designed hash table with an efficient hash function that produces a smooth distribution of addresses for the data given is closer to O(1) than to O(n). Of course, if the hash function does not evenly distribute items among the available buckets, the expected case for inserting, locating or removing an item from the list may approach the worst case of O(n).

3. **Choosing**

Choosing hash functions, hash table size and how collisions are handled cannot be separated. All three of these elements must be considered and balanced in order to make hashing work. Because of this, hashing is the most abstract of the ideas that we have thus far examined. In particular, hash functions consist of whatever will act on the data at hand to return usable addresses or indices. This means that the programmer is responsible for creating the hash function, possibly from whole cloth. While this may sound daunting, in practice the hash function is usually suggested by the choice of key values on which to hash.

4. **A Hashing Example**

In this example, an array will be used as a hash table and a simple hash function will be used. The hash function that is suggested from the data will be a simple one: The ASCII values of the characters of the key will be summed and the modulus operator will be employed to reduce the sum to within the bounds of the indices of the array.

The shipping department of the MegaBig Company keeps track of the orders that clients place from the time that an order is placed till delivery is confirmed at the client's address. (After delivery is confirmed, orders move on to billing and are no longer the concern of the shipping department.) An order is represented by a struct that contains the account number of the client and a pointer to a linked list of items that are on the list.
Here is the struct that represents items purchased:

```c
struct Items {
    string itemCode;
    int    quantity;
    Items *next;
};
```

Here is the struct that represents an order:

```c
struct Order {
    string clientAccount;
    string orderNumber;
    items *list;
};
```

Client account numbers are 14 upper case alphabetic characters assigned in sequential order the first time that a client places an order. Client account numbers are never reused.

Since orders are placed and filled at a heavy rate, MegaBig needs a data structure with associated algorithms that will be efficient in inserting and locating order records. For this task, they have chosen hashing on the key field clientAccount.

An array of pointers to structs of type Order has been chosen for the hash table. There are 100 elements in the hash table array. For example:

```c
typedef Order *Orderptr;
Orderptr table[100];
```

Since there are only 100 elements in the array and there can be $26^{14}$ client numbers, a simple hash function can be devised by first adding the ASCII values of each character in a client account number, then using the modulus operator on the total to insure that the output of the hash function is within the limits of the indices of the hash table array. This could be implemented as follows:

```c
int numBuckets = 100;
int sum = 0, index, x;
for (x=0; x<14; x++)
    sum += oneOrder.clientAccount[x];
index = sum % numBuckets;
```

If all characters in the client account number field have no special meanings, this simple hash function may work well. If the distribution is not even, it may be necessary to "fiddle" with the size of the hash table, increasing it or decreasing it to produce a better result. (Odd but true tip: Making a hash table an odd number in length or a prime number in length can overcome strange clumping of data.)

However, if some characters in the client account number have a special meaning, it may skew the indices produced by the hash function in favor of certain buckets. For
example, if the first two characters represent the country of the client and the second two
represent the city of the client, clients from large countries and large cities may be
disproportionately represented. In such a case, it may be a good idea to drop the first four
characters from the hash functions consideration. This could be implemented as follows:

```c
int numBuckets = 100;
int sum = 0, index, x;
for (x=4; x<14; x++)
    sum += oneOrder.clientAccount[x];
index = sum % numBuckets;
```

A worse case example would be where client account numbers were produced from
company names. How many account numbers would end in "Co" or "Inc"? How many
would begin with "The" as in "The Martian Group". Worse, the number of characters
might differ. In such a case, it would be best to hash on the middle most numbers. For
example, here is the above hash function rewritten to hash only on the middle 7 characters
of the client account number.

```c
int numBuckets = 100;
int sum = 0, index, x;
int len = oneOrder.clientAccount.length( );
if (len >=7)
    for (x=len/2-3; len/2+3; x++)
        sum += oneOrder.clientAccount[x];
else
    for (x=0; x<len; x++)
        sum += oneOrder.clientAccount[x];
index = sum % numBuckets;
```
Programming Exercise 45.1

a. Create a file of 10,000 random alphabetic characters using the following statement to generate the characters:

```
// c is a variable of type char
c = (char) (rand() % 26 + 65);
```

b. Create a program that will read the file one character at a time, subjecting each character to the following hash function.

```
// i is an integer variable
i = (int) c % 26;
```

This program should also contain an integer array of 100 elements. Each element of the array will act as a counter, counting the number of times the hash function returns the address (index) of the element. When the file has been completely read and all characters have been hashed, output the contents of the array.

If this hash function were to be used with a hash table and data items with single upper case letter keys, would the buckets be evenly used? If so, why? If not, why?

Programming Exercise 45.2

Go to the Internet Public Library (http://www.ipl.org) or some other source and download a book (one in the public domain) in text form. Alter the program of 45.2 so that the program reads from the text file. Do not hash any non-alphabetic characters. Use the function

```
char toupper(char)
```

to convert any lower case letters to upper case (first including <stdlib.h> and <ctype.h>).

If this hash function were to be used with a hash table and data items with single upper case letter keys, would the buckets be evenly used? If so, why? If not, why?
Programming Exercise 45.3
Alter the program of 45.2 so that the key the hash function receives is one word. Read the entire file until each word in the file is hashed. (See the example hash functions in section 4.) This time, include all characters in the word, alphabetic, and disregarding case.

- Run the program with a counter array of size 26. If this hash function were to be used with a hash table and data items, would the buckets be evenly used? If so, why? If not, why?
- Run the program with a counter array of size 51. If this hash function were to be used with a hash table and data items, would the buckets be evenly used? If so, why? If not, why?
- Run the program with a counter array of size 99. If this hash function were to be used with a hash table and data items, would the buckets be evenly used? If so, why? If not, why?
- Run the program with a counter array of size 251. If this hash function were to be used with a hash table and data items, would the buckets be evenly used? If so, why? If not, why?

Programming Exercise 45.4
Try the programming exercises in 45.3 again with other text files.