Hashing and Hash Tables

1 Introduction

A hash table is a look-up table that, when designed well, has nearly $O(1)$ average running time for a find or insert operation. More precisely, a ***hash table*** is an array of fixed size containing data items with unique keys, together with a function called a ***hash function*** that maps keys to indices in the table. For example, if the keys are integers and the hash table is an array of size 127, then the function $hash(x)$, defined by

$$\text{hash}(x) = x\%127$$

maps numbers to their modulus in the finite field of size 127.

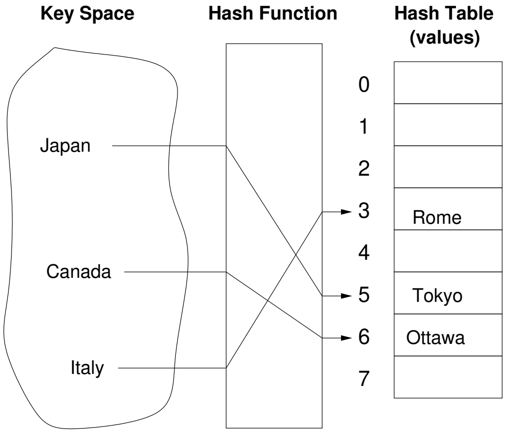


Figure 1: A hash table for world capitals.

Conceptually, a hash table is a much more general structure. It can be thought of as a table H containing a collection of *(key, value)* pairs with the property that H may be indexed by the key itself. In other words, whereas you usually reference an element of an array `A` by writing something like `A[i]`, using an integer index value `i`, with a hash table, you replace the index value "`i`" by the key contained in location `i` in order to get the value associated with it. For example, we could conceptualize a hash table containing world capitals as a table named $Capitals$ that contains the set of pairs

```
(“Italy”, “Rome”), (“Japan”, “Tokyo”), (“Canada”, “Ottawa”)
```

and so on, and conceptually (not actually) you could write a statement such as

```
print Capitals[“Italy”]
```

and `Rome` would be printed, or

```
print Capitals[“Japan”]
```

and `Tokyo` would be printed. In practice, hash tables are implemented as two-stage lookups. In the first stage, the hash function is applied to the key to get an index, and in the second stage, the index is used to get the value associated with the key. Figure 1 illustrates the idea. So the expression that provides the capital of "`Japan`" would be `Capitals[hash(“Japan”)].` (If the programming language supports it, the subscript operator could be overloaded so that you could just write `Capitals[“Japan”]`.)

A table that can be addressed in this way, where the index is content rather than a subscript, is called a ***content-addressable-table*** (***CAT***). Hash tables are content-addressable tables. Some programming languages provide containers called ***maps***, which are essentially hash tables.

Obviously, if you can look up a value in $O(1)$ running time, you've got a good thing going, especially if inserting the value also takes $O(1)$ time. This would be a much better search table than a binary search tree of any kind, balanced or not.

2 Properties of a Good Hash Function

To *hash* means to chop up or make a mess of things, liked hashed potatoes. A hash function is supposed to chop up its argument and reconstruct a value out of the chopped up little pieces. Good hash functions

- make the original value intractably hard to reconstruct from the computed hash value,
- are easy to compute (for speed), and
- randomly disperse keys evenly throughout the table, making sure that no two keys map to the same index.

Hash functions are used in cryptography to encrypt messages. In this case, three other properties are required:

- They are deterministic.
- Small changes in the key result in big changes in the computed value.
- It is almost impossible to find two different keys that hash to the same value.

Easy to compute generally means that the function is an $O(1)$ operation, practically independent of the input size and hash table size. For example, if the function tried to find all of the prime factors of a given number in order to compute the hash function, this would not be easy to compute. Being easy to compute is a fuzzy concept. Dispersing the keys evenly means that there is as much distance between successive pairs of keys as possible. For example, if the hash table is of size 1000 and there are 200 keys in it, they should each be about five addresses apart from their neighbors, because $1000/200 = 5$.

In principle, if the set of keys is finite and known in advance, we can construct a ***perfect hash function***, one that maps each key to a unique index, so no two keys hash to the same value. Much research has been done on how to find perfect hash functions efficiently. For example, if we have the integer keys

112, 46, 75, 515

we would want a function that maps them to the numbers 0,1,2, and 3 uniquely. Coincidentally, although I picked these numbers randomly, my first guess at a perfect hash function for them was a good guess. Suppose that $g(x)$ is a function that returns the sum of the decimal digits in x. $g(x)$ could be defined recursively by

$$g(x) = \begin{cases} x & \text{if } x \leq 9 \\ x\%10 + g(x/10) & \text{otherwise} \end{cases}$$

For example, $g(122) = 5$ and $g(75) = 12$. Let $h(x)$ be defined by

$$h(x) = \begin{cases} g(x) & \text{if } g(x) \leq 9 \\ h(g(x)) & \text{otherwise} \end{cases}$$

The function $h(x)$ is the sum of the digits in x, but added recursively until it falls into the range $[0, 9]$. For example, $h(112) = 1{+}1{+}2 = 4$, $h(46) = h(4{+}6) = h(10) = 1$, and $h(75) = h(7{+}5) = h(12) = 3$. It was just dumb luck that these numbers mapped to unique indices; this particular hash function is, in fact, a very poor hash function. It is poor because it does not use much of the information content in the key such as the order of the digits; to wit, $h(112) = h(121) = h(211) = 4$. There are tools that construct perfect hash functions. One such tool is GNU's *gperf*, which can be downloaded from ftp://ftp.gnu.org/pub/gnu/.

3 Collisions

Since almost all practical hash functions are not perfect, they will map one or more keys to the same indices, the way that $h()$ defined above mapped 112, 121, and 211 to the same index value 4. When two or more keys are mapped to the same location by the hash function, it is called a ***collision***. When a collision occurs, a new location must be found for the key that caused it, i.e., the second key to be hashed. The strategy for relocating keys for which a collision occurred is called the ***collision resolution strategy*** or the ***collision resolution algorithm***. Since the relocation itself may cause further collisions, the goal of a collision resolution algorithm is to minimize the total number of collisions in a hash table.

Notation. Throughout the remainder of these notes on hash tables, M will denote the length of the hash table, and the table itself will be denoted by H, so that $H[0]$ through $H[M-1]$ are the table entries. Hash functions will be denoted by function symbols such as $h(x)$, $h_1(x)$, $h_2(x)$ and so on, and x will always denote a key.

4 Hash Functions

While there are many different classes of integer functions to server as candidate hash functions, we will focus on the three most common types:

division-based hash functions: primary operation is division of the key

bit-shift-based hash functions: bit-shifting is the primary operation

multiplicative hash functions: multiplication is the primary operation

4.1 Division Based Hash Functions

The division method uses the modulus operation (%), which is actually a form of division both conceptually and operationally[1]. In the division method, the hash function is of the form

$$h(x) = x \,\%\, M$$

Certain choices of M are obviously worse than others. For example, if x is an n-bit number and $M = 2^m$, where $m < n$, then $x \,\%\, M = x \,\%\, 2^m$ is nothing more than the m least significant bits of x, which is not particularly good because it fails to use the upper $m - n$ bits, throwing away a lot of information. Although this looks like a division operation, when $M = 2^m$ it is implemented easily using the bitwise-and operation. Let `mask=000...0111...1` where there are $n - m$ 0's followed by m 1's. Then

$x \,\%\, M = x \,\&\, mask$

Other poor choices follow. If M is any even number, $h(x)$ will be even for even x and odd for odd x, introducing bias into the table. It is a bad idea for M to be a multiple of 3 also, because then two numbers that differ only by a permutation of their digits will be hashed to locations that differ by a multiple of 3. For example, if we let $M = 6$, then $52 \,\%\, 6 = 4$ and $25 \,\%\, 6 = 1$, a distance of 3 apart. Similarly, $1157 \,\%\, 6 = 5$ and if we permute 1157 to 7511, $7511 \,\%\, 6 = 5$.

There are other subtle problems that arise for various values of M. Making the table size M a prime number tends to avoid these problems. In fact, there are even certain types of primes that work better than others. (See *The Art of Computer Programming, Vol. 3, Sorting and Searching* by Knuth for more details.) The hash functions

$$h_1(x) = x \,\%\, 127$$
$$h_2(x) = x \,\%\, 511$$
$$h_3(x) = x \,\%\, 2311$$

are examples of division-based hash functions with M being prime. Compared to multiplication, addition, and subtraction, division is a slow operation; this is one reason to investigate other types of hash functions. It is easy to write such functions, but they do not have many of the desirable properties of hash functions.

[1] $k = x \,\%\, M$ stores into k the remainder of x divided by M for positive x and M. Unless M is a power of *2*, it will most likely require a hardware division.

4.2 Bit-Shifting Hash Functions Using the Middle Square Method[2]

Bit-shifting refers to the machine operation in which data is shifted to the left or right by some fixed number of bits. A bit shift of k bits to the right, filling the upper k bits with zeros, is equivalent to integer division by 2^k. For example, shifting the bit string $10100_2 = 20_{10}$ to the right two bits results in $00101_2 = 5_{10}$, which is 20/4. Let w be the word size in bits of a processor. Most computers on the market today have a word size of 64 bits. Older models have a 32-bit word size. Let $W = 2^w$. Then W is the total number of integers that can be represented in a machine word, regardless of the representation. There are $W = 2^w$ integers from 0 to $2^w - 1$, so W is one more than the largest `unsigned int`, i.e., W =`UINT_MAX+1`, where `UINT_MAX` is the largest representable unsigned integer as defined in the header file `limits.h`. The 32-bit number shown below is `UINT_MAX` on a 32-bit architecture; W is the number that could be represented with a 1 in the 33^{rd} bit and 0's in the remaining 32 bits.

```
Bit#        32 31 30                                                       2  1  0
UINT_MAX =    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
W        = 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

It is not hard to see that the expression $(b \,\%\, W)$ where b is any integer, is nothing more than the low-order 32 bits of b. If the machine word is 32 bits, this is just a way to ignore the overflow if $b \geq W$ and b is stored in a machine word. If you instruct the processor to ignore integer overflow during integer computations, you are essentially computing $(b \,\%\, W)$. Now consider the expression:

$$h(x) = \frac{M}{W}(x^2 \,\%\, W)$$

First of all, for this function to be efficient, M must be a power of 2. Assume that $M = 2^m$ for some positive integer $m < w$. Then $M/W = 2^m/2^w = 2^{m-w}$. Then $h(x)$ is equivalent to

$$h(x) = 2^{m-w}(x^2 \,\%\, 2^w)$$

Notice that, because W is larger than M, $0 < M/W < 1$ and $(m - w) < 0$. Thus, multiplying by M/W is the same as dividing by $W/M = 2^{w-m}$ which is the same as shifting right $(w - m)$ bits. Also, if x^2 overflows the machine word, then $x^2 \,\%\, 2^w$ is just the low order w bits of x^2. This hash function basically ignores the overflow caused by squaring x, and then shifts x^2 to the right by $(w - m)$ bits. Since x^2 is shifted $(w - m)$ bits to the right in a word with w bits, the leading $(w - m)$ bits are zero filled and the final value lies in the low-order m bits. This proves that $0 \leq h(x) < 2^m = M$. Hence this hash function generates numbers between 0 and $M - 1$, as it should.

Notice that this function does not require a division operation, since it is equivalent to the following *C/C++* expression:

```
h(x) = ( x * x ) >> ( w - m)
```

where `>>` is the shift-right operator in *C/C++*. This function uses one multiplication and one bit shift and is therefore much faster than the division method. This assumes that integer overflow is ignored.

[2] This method was proposed in 1947 by John von Neumann in a letter written to Robert Richtmyer.

Example

Suppose that $w = 8$, $m = 5$. Then $W = 2^8 = 256$ and $M = 2^5 = 32$. Then

$$h(x) = 2^{5-8}(x^2\% \, 256) = 2^{-3}(x^2\% \, 256)$$

Let $x = 22$. Then $x^2 = 484 = 111100100_2$ and $x^2\% \, 256 = 11100100_2$. When this is shifted to the right by 3 bits, it becomes $00011100_2 = 28$.

4.3 Multiplicative Hash Functions

A multiplicative hash function is of the form

$$h(x) = \frac{M}{W}(Ax \,\%\, W)$$

where A is a carefully chosen constant. Although it is not necessary for M to be a power of 2, assume for simplicity that $M = 2^m$. When M is so defined, multiplying by M/W has the exact same effect as described in Section 4.2 above: it right-shifts the bitstring produced by $Ax \,\%\, W$ by $w - m$ bits. If $Ax < W$, this hash function does nothing more than bit-shift the product Ax to the right by $(w - m)$ bits. If Ax is larger than W, it removes all but the low-order w bits of Ax and then shifts this value to the right $(w - m)$ bits:

```
h(x) = A*x >> (w-m);
```

The resulting number is a value between 0 and $M - 1$, for the same reason that the *Middle Squares* method result lies between 0 and $M - 1$. We now explain more about how to choose the constant A and why its value is so critical to the behavior of this hash function.

The way to think about this hash function is to remind yourself that a binary integer has a decimal point (a binary point?) to the right of the least significant bit, and that the effect of division by W is to move that binary "decimal point" to the left side of the most significant bit. Suppose that the rectangle labeled W in line 1 in Figure 2 is the length of a machine word with w bits. Suppose that the product Ax is much larger than W, as depicted in line 2. Then $Ax \,\%\, W$ is the portion of Ax shown in line 3, shaded lightly - the part of Ax that fits into the low-order w bits. Multiplying by M/W is the same as dividing by W and then multiplying by M. As we just stated, dividing by W is the same as shifting the "binary decimal point" from the extreme right of the word to the point just left of W, as shown in line 4. Multiplication by M is the same as shifting the quantity $Ax \,\%\, W$ to the left by m bits. If the double-ended arrow labeled M represents the length of an m-bit word, then line 5 shows the result of this multiplication - the shaded reactangle shifts to the left by m bits and m zeros are filled to the right.

You have always assumed that the "binary decimal point" is to the right of a w-bit integer. Suppose instead that all numbers z stored in machine words are actually the numerators of fractions of the form (z/W) and therefore that the binary decimal point is actually to the left of the word, as it is in line 4. The picture in line 4 shows you that we can think of $Ax \,\%\, W$ as just the fractional part of Ax. The part to the left of the decimal point is the whole number and the part to the right is the fractional part. Let us denote the fractional part of any real number z by $\{z\}$ i.e.,

$$\{z\} = z - \lfloor z \rfloor$$

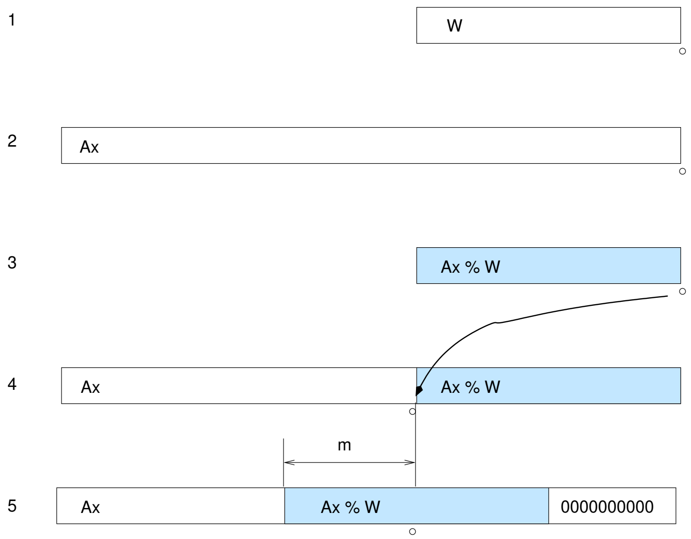


Figure 2: Multiplicative hash function.

For example $\{2.324\} = 0.324$ and $\{62.075\} = 0.075$. Then $Ax \ \% \ W$ is just $\{Ax\}$, and the hash function $h(x)$ can be written as

$$h(x) = \lfloor M \cdot \{Ax\} \rfloor$$

Now since $0 \leq \{Ax\} < 1$, it follows that $0 \leq M \cdot \{Ax\} < M$. In other words, $h(x)$ hashes values x into indices in the range of the hash table.

In the multiplicative method, the multiplier A must be carefully chosen to scatter the numbers in the table. A very good choice for the constant A is $\phi^{-1}W$, where ϕ, known as the ***golden ratio***, is the positive root of the polynomial

$$x^2 - x - 1$$

This is because x is a solution to the polynomial if and only if

$$\begin{aligned} & x^2 - x - 1 &&= 0 \\ iff \ & x^2 - x &&= 1 \\ iff \ & x - 1 &&= \frac{1}{x} \end{aligned}$$

In other words, a solution x has the property that its inverse is $x - 1$. The solution, ϕ, is called the golden ratio because it arises in nature in an astonishing number of places and because the ancient Greeks used this ratio in the design of many of their buildings, considering it a ***divine proportion***. Thus, ϕ and $\phi^{-1} = \phi - 1$ are both roots of $x^2–x–1$. ϕ is also the value to which f_n/f_{n-1} converges as $n \to \infty$, where f_n is the n^{th} Fibonacci number. Since $\phi^{-1} = \phi - 1$, it is approximately 0.6180339887498948482045868343656. (Well *approximately* depends on your notion of approximation, doesn't it?)

When we let $A = \phi^{-1}W$ as the multiplicative constant, the multiplicative method is called ***Fibonacci hashing***. The constant has many remarkable and astonishing mathematical properties, but the property that makes it a good factor in the above hash function is the following fact.

First observe that the sequence of values $\left\{\phi^{-1}\right\}, \left\{2\phi^{-1}\right\}, \left\{3\phi^{-1}\right\}, \ldots$ lies entirely in the interval $(0, 1)$. Remember that the curly braces mean, "the fractional part of", so for example, $2\phi^{-1} \approx$

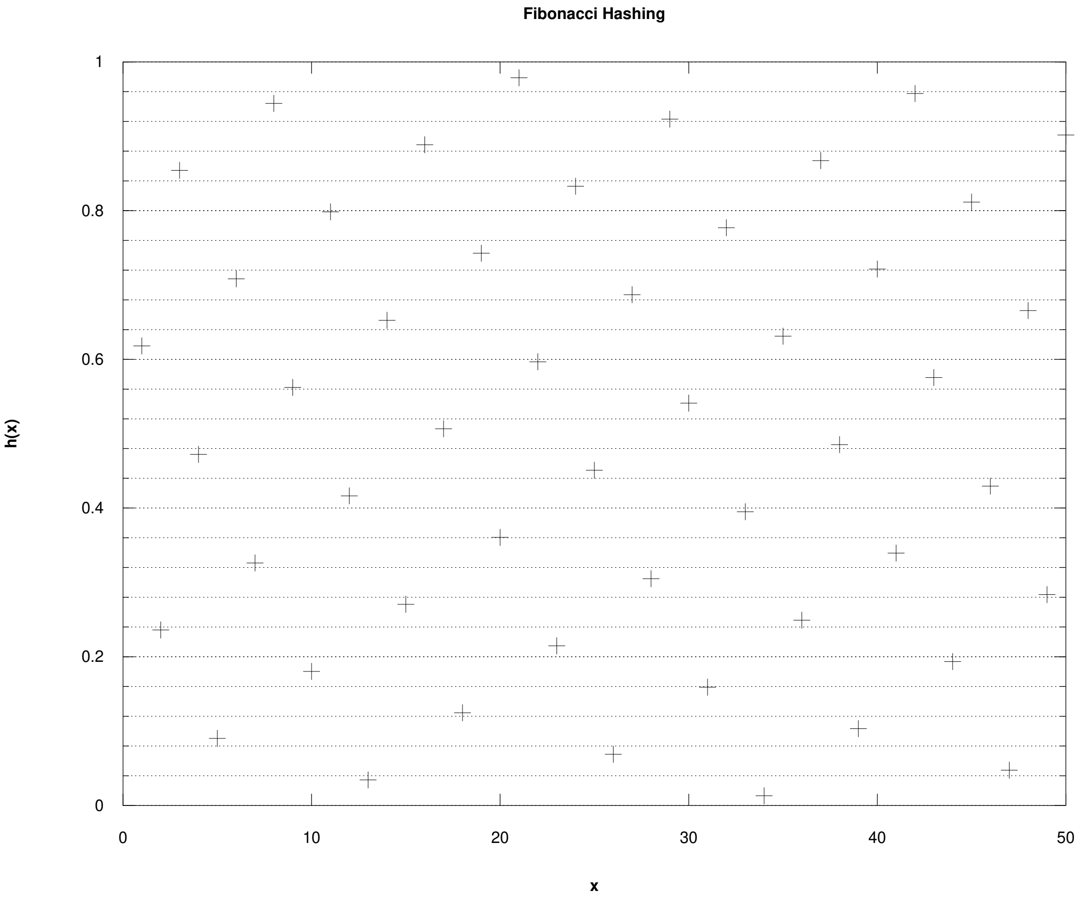


Figure 3: Spread of hashed values using Fibonacci hashing.

1.236067977 and $\left\{2\phi^{-1}\right\} \approx 0.236067977$. The first value, $\left\{\phi^{-1}\right\}$, divides the interval $(0,1)$ into two segments whose lengths are in the golden ratio.

Proof. The segment $(0,1)$ is divided into two segments, one of length ϕ^{-1} and the other of length $1-\phi^{-1}$. We claim that

$$\frac{1-\phi^{-1}}{\phi^{-1}} = \phi^{-1}$$

Multiplying numerator and denominator by ϕ we get

$$\frac{\phi-1}{1} = \frac{1}{\phi}$$

Since ϕ is a solution of $x^2-x-1=0$, $\phi^2-\phi-1=0$ and $\phi^2-\phi=1$. This is just the cross-product of the above equation. □

In fact, every value $\{k\phi^{-1}\}$ divides the segment into which it is placed into two segments whose lengths are in the golden ratio. Moreover, each successive value is placed into one of the largest segments in the interval $(0, 1)$. This implies that the successive values are spread out across the interval uniformly, no matter how many points there are[3]. See Figure 3.

When used in the hash function above, they spread successive keys into the hash table in the same way. This tends to reduce the possibility of collisions. A table showing the values of the multiplier for typical machine word sizes is shown in Table 1.

w	$\phi^{-1}W$
16	40,503
32	2,654,435,769
64	11,400,714,819,323,198,485

Table 1: Fibonacci hashing multipliers.

5 String Encodings

Keys are often strings. A hash function is usually a function from integers to integers, so how do we map strings to integers? Let an arbitrary string be denoted s, consisting of the $k + 1$ symbols over some fixed alphabet:

$s = s_0 s_1 s_2 ... s_k$

Let S denote the set of all possible strings over this alphabet. A ***string encoding*** is a function from the set of strings S to non-negative integers

encode : $S \rightarrow Z$

The `encode` function does not have to be invertible, i.e., one-to-one and onto; it may map different strings to the same number. Once we have an encoding function, it can be composed with a hash function h as follows:

```
table_index = h(encode(s));
```

Encodings need to be chosen carefully. There are good encodings and bad ones. Here is a bad one:

`encode1`$(s) =$ `int`$(s_0) +$ `int`(s_1) `+ ... + int`(s_k)

This is bad because it ignores letter order, so that `stop`, `spot`, and `tops` hash to the same location. It is better to find an encoding that maps words that are permutations of each other to unique numbers. One way to do that is to reinterpret a text string as a number over a very large alphabet.

In our customary decimal number system, a numeral such as 3276 is a representation of the number that we express in English as "three thousand two hundred seventy six," which is equal to

$$3 \cdot 10^3 + 2 \cdot 10^2 + 7 \cdot 10^1 + 6 \cdot 10^0$$

When you see a hexadecimal numeral such as $BADCAB2$, you know that this is a representation of the number

$$B \cdot 16^6 + A \cdot 16^5 + D \cdot 16^4 + c \cdot 16^3 + A \cdot 16^2 + b \cdot 16^1 + 2 \cdot 16^0$$

[3] This property of the golden ratio is actually a special case of a more general theorem proved in 1957 by Vera Turán Sós, which states that for any irrational number θ, when the points $\{\theta\}$, $\{2\theta\}$, $\{3\theta), \ldots \{n\theta\}$are placed into the line segment $[0, 1]$, the $n + 1$ line segments between them are of at most three different lengths and the next point $\{(n + 1)\theta\}$ will fall into one of the largest intervals!

where A represents the number 10, B represents 11, C represents 12, and so on. This numeration system is surjective (onto) in that, for any integer n, there is a unique hexadecimal string that represents it. We can generalize this idea. Suppose all strings are over an alphabet of, say 26 characters, such as the symbols, `a,b,c,d,...,z`. Suppose that `a` represents the value 0, `b`, the value 1, `c`, 2, up to `z` representing 25. Then the string `hashing` represents the number

$$\begin{aligned}
&h \cdot 26^6 + a \cdot 26^5 + s \cdot 26^4 + h \cdot 26^3 + i \cdot 26^2 + n \cdot 26^1 + g \cdot 26^0 \\
=&7 \cdot 26^6 + 0 \cdot 26^5 + 18 \cdot 26^4 + 7 \cdot 26^3 + 8 \cdot 26^2 + 13 \cdot 26^1 + 6 \cdot 26^0 \\
=&2162410432 + 8225568 + 123032 + 5408 + 338 + 156 \\
=&2170764934
\end{aligned}$$

This method of encoding strings maps any string s to a unique integer. The problem is that the integer is usually larger than can be represented in even a 64-bit number. In the example above, the value of `hashing` will not fit into a signed 32-bit integer. Nonetheless, this encoding can be used by selecting some of the letters from a given word instead of all of them. The encoding can use the even numbered letters or the first m letters, or every third letter, or the odd letters, and so on.

Summarizing, let $s = s_0 s_1 s_2 ... s_k$ be a string over an alphabet containing B distinct symbols. As it does not matter which side of s we start from, we can define an encoding of s to a unique integer by

$$\text{encode }(s) = \sum_{j=0}^{k} s_j B^j = s_0 B^0 + s_1 B^1 + s^2 B^2 + \cdots + s_k B^k$$

As mentioned, unless the string is sampled, the encoded value for most strings will be too large. Another problem with this encoding is that it does not satisfy the condition that it is easy to compute. The naive way to compute it would require $k + (k-1) + (k-2) + ... + 2 + 1 = k(k+1)/2$ multiplications. A much more efficient way to compute this is to use ***Horner's Rule***, turning off integer overflow. Horner's Rule is a way to compute polynomials efficiently. It is based on applying the following definition of a polynomial recursively:

$$\begin{aligned}
a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n &= a_0 + x(a_1 + a_2 x + \cdots + a_n x^{n-1}) \\
&= a_0 + x(a_1 + x(a_2 + a_3 x + \cdots + a_n x^{n-2})
\end{aligned}$$

and so on. In other words, a polynomial $p(x)$ of degree n can be written $p(x) = a_0 + x(q(x))$ where $q(x)$ is of degree $n-1$. This definition uses the customary notation for polynomials, with the highest index coefficient multiplying the highest power of x. The following function uses Horner's Rule for computing a code value for a string over an alphabet with `RADIX` many symbols:

```
long long encode ( const int RADIX, const string & s)
{
    long long hashval = 0;
    for (int i = 0; i < s.length(); i++)
        hashval = s[i] + RADIX * hashval; // p(x) = s_i + x(q(x))
    return hashval;
}
```

Notice that this function performs one multiplication and one addition per iteration, for a total of n multiplications and n additions for a string of length n.

6 Collision Resolution

There are two basic methods of collision resolution:

- separate chaining
- open addressing

Open addressing itself has many different versions, as you will see. ***Separate chaining*** is the easiest to understand, and perhaps implement, but its performance is not as good as open addressing. There are also methods that are hybrids of the two:

- cuckoo hashing
- universal hashing

Some perfect hash function generators use a form of universal hashing to generate the perfect hash functions.

6.1 Separate Chaining

In separate chaining, the hash table is an array of linked lists, with all keys that hash to the same location in the same list. New keys are inserted in the front of the list. In other words, each hash table entry is a pointer to a list of keys and their associated data items. See Figure 4.

To insert a key into the table, the hash table index is computed, and then the list is searched to see if the key is already in the table. If it is not, it is inserted at the head of the list. In the worst case, this requires a search of the entire list. On average, half of the list is searched on each insertion. It is not worth keeping the list in sorted order if it is short.

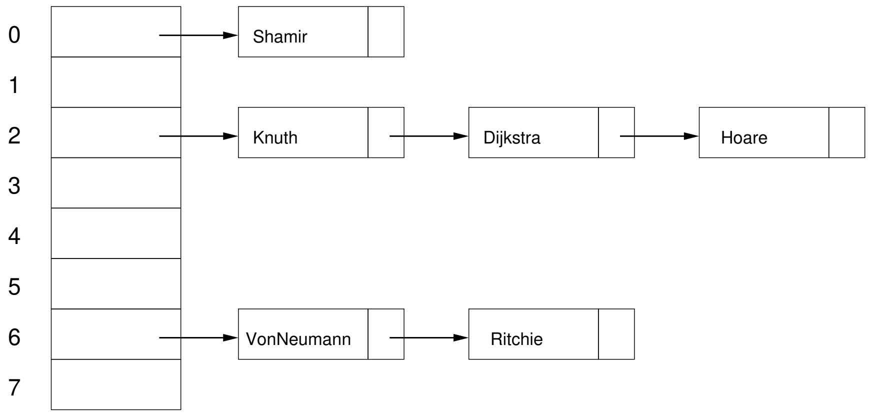


Figure 4: Separate chaining method of collision resolution.

The ***load factor***,λ, is defined as the ratio of the number of items in the table to the table size. If M is the table size and N is the number of items in the table, then $\lambda = N/M$. The table in Figure 4 has a load factor of $6/8 = 0.75$. In separate chaining, the load factor is the average length of the linked lists, if we count the empty ones as well as the non-empty ones. It is possible for the load factor to exceed 1.0 when separate chaining is used.

The following listing shows an example of a template class interface for a hash table. From the private part we can surmise that this table uses separate chaining. This interface provides functions to insert, remove, and find keys, as well as a copy constructor and an ordinary constructor. Once again the weakness of C++ surfaces here because the interface, which should hide the fact that the hash table uses separate chaining, does not. We could overcome this by declaring an abstract base class and deriving the specific hash table from it.

```
template <class HashedObj>
class HashTable
{
    public:
        explicit HashTable     ( const HashedObj & notFound, int size =
           101 );
        HashTable              ( const HashTable & rhs ) :
                                 ITEM_NOT_FOUND(rhs.ITEM_NOT_FOUND ),
                                    Lists(rhs.theLists ) { }
        const HashedObj & find ( const HashedObj & x ) const;
        void makeEmpty         ( );
        void insert            ( const HashedObj & x );
        void remove            ( const HashedObj & x );

    private:
        vector<List<HashedObj> > Lists;   // array of linked lists
        const HashedObj ITEM_NOT_FOUND;
};

int hash( const string & key, int tableSize );
int hash( int key, int tableSize );
```

Let us now analyze the performance of this method of implementation. The most expensive operation in using a hash table is the operation of accessing a key in a hash table entry and inspecting its value. We call such an operation a ***probe***. Consider the find operation. If a search is successful, the key is found in exactly one of the nodes in some list. If that node is the k^{th} node in the list, then k probes are required. If all nodes are equally likely to be the one searched for, then there are, on average,

$$\frac{1}{\lambda}\sum_{k=1}^{\lambda} k = \frac{\lambda+1}{2}$$

probes in a list of length λ. As λ is the average list length, $(\lambda+1)/2$ is the expected time to find a key that is in the table. An unsuccessful search requires traversing the entire list, which is λ links. Inserting into a separately chained hash table takes the same amount of time as an unsuccessful search, roughly. An insertion has to search to see if the key is in the table, which requires traversing the entire list in which it is supposed to be located. Only after the list is checked can it be inserted at the front of that list.

Separate chaining makes deletion from a hash table quite easy because it is amounts to nothing more than a linked list deletion. On the other hand, as we will soon see, it is slower than deletion in open addressing because of the time it takes to traverse the linked lists.

6.2 Open Addressing

In open addressing, there are no separate lists attached to the table. All values are in the table itself. When a collision occurs, the cells of the hash table itself are searched until an empty one is found. Which cells are searched depends upon the specific method of open addressing. All variations can be described generically by a sequence of functions

$$\begin{aligned} h_0(x) &= h(x) + f(0,x) \; \% \; M \\ h_1(x) &= h(x) + f(1,x) \; \% \; M \\ &\ldots \\ h_k(x) &= h(x) + f(k,x) \; \% \; M \end{aligned}$$

where $h_i(x)$ is the i^{th} location tested and $f(i,x)$ is a function that returns some integer value based on the values of i and x. The idea is that the hash function $h(x)$ is first used to find a location in the hash table for x. If we are trying to insert x into the table, and the index $h(x)$ is empty, we insert it there. Otherwise we need to search for another place in the table into which we can store x. The function $f(i,x)$, called the ***collision resolution function***, serves that purpose. We search the locations

$$\begin{aligned} h(x) + f(0,x) \; \% \; M \\ h(x) + f(1,x) \; \% \; M \\ h(x) + f(2,x) \; \% \; M \\ \ldots \\ h(x) + f(k,x) \% \; M \end{aligned}$$

until either an empty cell is found or the search returns to a cell previously visited in the sequence. The function $f(i,x)$ need not depend on both i and x. Soon, we will look at a few different collision resolution functions.

To search for an item, the same collision resolution function is used. The hash function is applied to find the first index. If the key is there, the search stops. Otherwise, the table is searched until either the item is found or an empty cell is reached. If an empty cell is reached, it implies that the item is not in the table. This raises a question about how to delete items. If an item is deleted, then there will be no way to tell whether the search should stop when it reaches an empty cell, or just "jump" over the hole. The way around this problem is to ***lazy deletion***. In lazy deletion, the cell is marked DELETED. Only when it is needed again is it re-used. Every cell is marked as either

ACTIVE: it has an item in it

EMPTY: it has no item in it

DELETED: it had an item that was deleted – it can be re-used

These three constants are supposed to be defined for any hash table implementation in the ANSI standard.

There are several different methods of collision resolution using open addressing: ***linear probing***, ***quadratic probing***, ***double hashing***, and ***hopscotch hashing***.

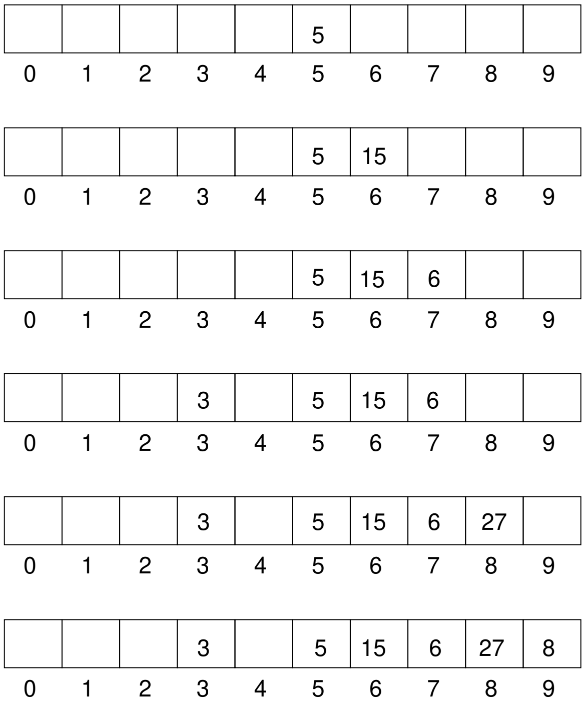
6.3 Linear Probing

In ***linear probing***, the collision resolution function, $f(i, x)$, is a linear function that ignores the value of x, i.e.., $f(i) = a \cdot i + b$. In the simplest case, $a = 1$ and $b = 0$, so that $f(i, x) = i$ and $h_i(x) = (h(x) + i) \,\%\, M$. In other words, consecutive locations in the hash table are probed, treating the table like a circular list.

Example 1. Consider a hash table of size 10 with the simple division hash function $h(x) = x \,\%\, 10$ and suppose we insert the sequence of keys,

$5, 15, 6, 3, 27, 8$

In principle, only one collision should occur: 5 and 15 because they both map to the location 5. But linear probing causes many more collisions. After inserting 5, 15 causes a collision. It is placed in `H[6]`. Then 6 has a collision at `H[6]` and is placed in `H[7]`. 3 gets placed without a collision, but 27 collides with 6 and is placed in `H[8]`. This causes 8 to collide, and it is placed in $H[9]$. Figure 5 shows the state of the hash table after each insertion.



					5				
0	1	2	3	4	5	6	7	8	9

					5	15			
0	1	2	3	4	5	6	7	8	9

					5	15	6		
0	1	2	3	4	5	6	7	8	9

			3		5	15	6		
0	1	2	3	4	5	6	7	8	9

			3		5	15	6	27	
0	1	2	3	4	5	6	7	8	9

			3		5	15	6	27	8
0	1	2	3	4	5	6	7	8	9

Figure 5: Linear probing example. This shows the successive states of the table when the keys $5, 15, 6, 3, 27, 8$ are inserted and the hash function is $h(x) = x \,\%\, 10$.

The problem with linear probing is that it tends to form ***clusters***. This phenomenon is called ***primary clustering***. As more cells that are adjacent to the cluster are filled, a snowball effect takes place, and the cluster grows ever more faster. It can be proved that the expected number of probes for insertions and unsuccessful searches is approximately

$$\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

and that the expected number of probes for successful searches is

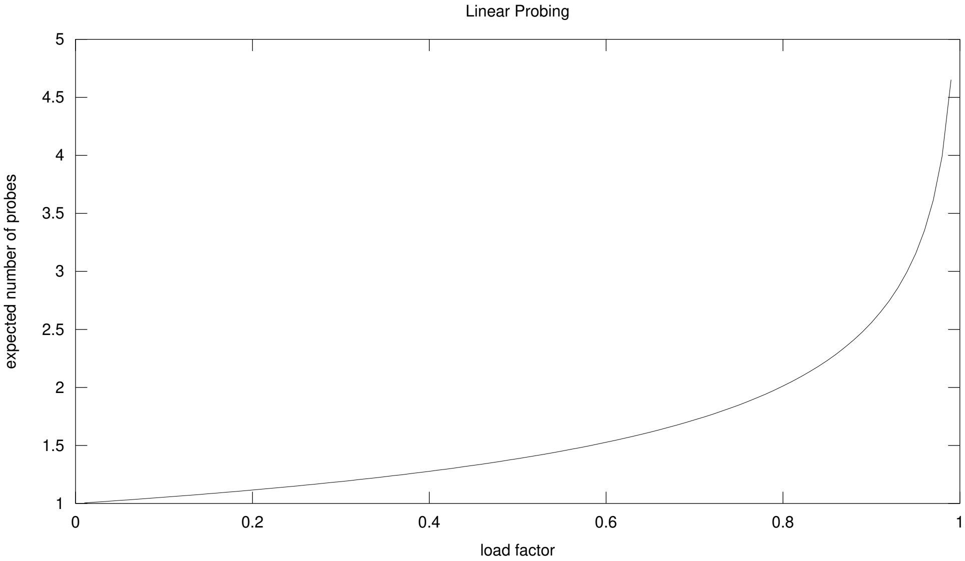


Figure 6: Graph of number of expected probes for a successful search as a function of load factor in linear probing.

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$$

Linear probing performs very poorly as load factor increases.

6.4 Random Probing

We can compare the performance of linear probing with a theoretical strategy in which clusters do not form and the probability that the next probe will succeed is independent of the probability that the previous probes succeeded. This is a kind of random collision resolution strategy. In this strategy, the number of probes in an unsuccessful search when the table has a load factor of λ is $1/(1–\lambda)$. This is because, if the collision resolution strategy is random, keys are placed uniformly throughout the table. Since λ is the fraction of the table that is occupied, the fraction of the table that is free is $1–\lambda$. This means that there is a λ probability that a random probe will hit an unoccupied cell, and a $(1–\lambda)$ probability that it hits an empty cell. The expected number of probes in an unsuccessful search is the expected number of cells that we visit before we find an empty cell, given that the item is not in the table. This is the mean of a geometric distribution with parameter $(1-\lambda)$, which is $1/(1–\lambda)$.

As the table is filled, the load factor increases, from 0 to the current load factor,λ. The average number of probes is therefore approximated by the definite integral

$$I(\lambda)=\frac{1}{\lambda}\int\limits_{0}^{\lambda}\frac{1}{1-x}dx=\frac{1}{\lambda}\ln\left(\frac{1}{1-\lambda}\right)$$

6.5 Quadratic Probing

Quadratic probing eliminates clustering. In quadratic probing the collision resolution function is a quadratic function of i and does not depend on x, namely $f(i,x) = i^2$. In other words, when a collision occurs, the successive locations to be probed are at a distance (modulo table size) of $1, 4, 9, 16, 25, 26, 49$, and so on. The sequence of successive locations is defined by the equations

$$\begin{aligned} h_0 &= h(x) \\ h_i &= (h_0 + i^2)\ \%\ M \end{aligned}$$

You should wonder why this is efficient, because it adds another multiplication for each probe. If it were implemented this way, it would not be efficient. Instead, it can be computed without multiplications by taking advantage of the recurrence relation

$$\begin{aligned} d_0 &= 1 \\ h_0 &= 0 \\ h_{i+1} &= h_i + d_i \\ d_{i+1} &= d_i + 2 \end{aligned}$$

The values of h_i are the successive squares. This relies on the observation that the odd numbers separate successive squares:

$$\begin{array}{lccccccccccccc} h_i & 0 & & 1 & & 4 & & 9 & & 16 & & 25 & & 36\ \dots \\ d_i & & 1 & & 3 & & 5 & & 7 & & 9 & & 11 & \dots \\ & & & 2 & & 2 & & 2 & & 2 & & 2 & \dots & \end{array}$$

To find an item x in the hash table when quadratic probing is used, the table must be probed until either an empty cell is found or until x is found. But this raises the question, is it possible that neither case will arise? The answer is yes, if M is not a prime number.

Example 2. Let $M = 24$, and suppose that we use the hash function $h(x) = x\ \%\ 24$ and quadratic probing. Further, suppose that the current state of the table is that $H[k] = k$ for $k = 0, 1, 4, 9, 12$, and 16. Only 6 out of 24 cells are in use, less than half. Suppose that we try to insert 24 into the table. The probe sequence will be

h_0	h_1	h_2	h_3	h_4	h_5
$(24+0)\%24$	$(24+1)\%24$	$(24+4)\%24$	$(24+9)\%24$	$(24+16)\%24$	$(24+25)\%24$
0	1	4	9	16	1

h_6	h_7	h_8	h_9	h_{10}	h_{11}
$(24+36)\%24$	$(24+49)\%24$	$(24+64)\%24$	$(24+81)\%24$	$(24+100)\%24$	$(24+121)\%24$
12	1	16	9	4	1

h_{12}	h_{13}	h_{14}	h_{15}	h_{16}	h_{17}
$(24+144)\%24$	$(24+169)\%24$	$(24+196)\%24$	$(24+225)\%24$	$(24+256)\%24$	$(24+289)\%24$
0	1	4	9	16	1

h_{18}	h_{19}	h_{20}	h_{21}	h_{22}	h_{23}
$(24+324)\%24$	$(24+361)\%24$	$(24+400)\%24$	$(24+441)\%24$	$(24+484)\%24$	$(24+529)\%24$
12	1	16	9	4	1

and the sequence $(0, 1, 4, 9, 16, 1, 12, 1, 16, 9, 4, 1, ...)$ will repeat *ad infinitum.*

Example 3. Suppose $M = 24$ and the table currently has the following contents:

0	1			4					9			12				16							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Suppose we try to insert 24. $h(24) = 0$. The locations searched are the successive squares, modulo 24, which are: $1, 4, 9, 16$,$25\%24 = 1$, $36\%24 = 12$, $49\%24 = 1$, $64\%24 = 16$, $125\%24 = 1$, and so on. In fact, the only locations that will be probed are the ones that are currently filled, and no others.

M must be a prime number or else the search canl go on forever, but this is a necessary but not sufficient condition to prevent an infinite search. The table must also be less than half full. But if M is a prime number and the table is less than half full we are guaranteed to find a cell. This is stated as a theorem:

Theorem 4. *If quadratic probing is used, and table size is prime, a new element can always be inserted if the table is at least half empty.*

Proof. Suppose that M is prime and the table is at least half empty. Assume that M is larger than 2, because if $M = 2$, it is trivial to see that it is true: there is 1 empty cell and adding 1 to the first location finds it.

Since $M > 2$ and prime, it is an odd number. Let $M = 2m + 1$ for some m. Since the table is at least half empty and M is odd, at least $m + 1$ cells are empty. (If only m cells were empty, it would be less than half of $2m + 1$.) Therefore at most m cells are full. Now suppose that the theorem is false. To say it is false means that indefinite, repeated probing fails to find an empty cell. Consider the first $m+1$ locations probed by quadratic probing, including the first location where the collision occurred. Let these $m + 1$ locations be labeled h_0, $h_1, h_2, \ldots, h_m$, where

$h_k = (h(x) + k^2)\ \%\ M$

Since probing did not find an empty cell in these first $m+1$ locations, all of these $m+1$ locations are full. But there are at most m full cells. This implies that two of these $m+1$ probes must have been at the same location. (Pigeon-hole principle: if there are $m + 1$ pigeons and only m pigeon holes, then two pigeons share a hole.) Suppose that h_i and h_j are the two probes at the same location, where $0 \leq j < i \leq m$. Then

$$\begin{aligned}
h_i &= h_j && iff \\
(h(x) + i^2)\ \%\ M &= (h(x) + j^2)\ \%\ M && iff \\
i^2\ \%\ M &= j^2\ \%\ M && iff \\
(i^2 - j^2)\ \%\ M &= 0\ \%\ M && iff \\
(i + j)(i - j)\ \%\ M &= 0\ \%\ M
\end{aligned}$$

Since M is a prime number it has no factors other than 1 and itself. This implies that either $(i - j)$ is zero or a multiple of M, or $(i + j)$ is a nonzero multiple of M. But since $i > j$, $(i - j) > 0$. Also, since $(i - j) < M$, their difference cannot be a multiple of M greater than 0. Therefore, the only possibility is that $(i + j)$ is a multiple of M. But $0 \leq j < i \leq m$, which implies that $j < m$ and

$i \leq m$, so the sum of i and j cannot be equal to or greater than $2m$. Since $2m < 2m + 1 = M$, this implies that $(i + j) < M$, so it certainly is not a positive multiple of M. This is a contradiction, which implies that the hypothesis that probing failed to find an empty location is false. The theorem must be true. □

7 Algorithms

The probing algorithm can be used for both insertion and searching. The quadratic probing function is called findPos, and is below. It returns a hash table location where the key is located, or the first empty cell found. The hash table must have a tag with each cell to indicate whether it is empty or not. This member is called info. The calling function can check if the return value is an empty cell using code such as

```
if ( H[findPos(x)].info == EMPTY)
```

where findPos can be defined as follows:

```
template <class HashedObj>
int HashTable<HashedObj>::findPos( const HashedObj & x ) const
{
int collisionNum = 0;
int currentPos = hash( x, array.size( ) );
    while( array[ currentPos ].info != EMPTY
        &&  array[ currentPos ].element != x ) {
        currentPos += 2 * ++collisionNum - 1;  // Compute ith probe
        if( currentPos >= array.size( ) )
            currentPos -= array.size( );
    }
    return currentPos;
}
```

The insertion algorithm:

```
template <class HashedObj>
void HashTable<HashedObj>::insert( const HashedObj & x )
{
    // Insert x as active
    int currentPos = findPos( x );
    if ( isActive( currentPos ) )
        return;
    array[ currentPos ] = HashEntry( x, ACTIVE );
    // Rehash
    // if the insertion made the table get half full,
    // increase table size
    if( ++currentSize > array.size( ) / 2 )
        rehash( );
}
```

The algorithm to find an item is

```
template <class HashedObj>
const HashedObj & HashTable<HashedObj>::find( const HashedObj & x ) const
{
   int currentPos = findPos( x );
   return
    isActive( currentPos ) ?
       array[ currentPos ].element : ITEM_NOT_FOUND;
}
```

8 Double Hashing

In ***double hashing***, the sequence of probes is a linear sequence with an increment obtained by applying a second hash function to the key:

```
f(i) = i * hash2(x);
```

We search locations `hash(x) + i*hash2(x)` for `i` = 1,2,3,...

The choice of the second hash function can be disastrous – it should never evaluate to a factor of the table size, obviously. It should be relatively prime to table size. It should never evaluate to 0 either. Choosing

```
hash2(x) = R- (x % R)
```

will work well if `R` is a small prime number.

NOTE: Add a worked example here.

Example

A hash table H is of size 23 and the primary hash function is

```
h(x) = x % 23
```

The second hash function is chosen to be

```
hash2(x) = 13 - (x % 13)
```

The sequence of keys is 12, 35, 6, 29, 5, 28, ...

9 Rehashing

If the hash table gets too full it should be resized. The best way to resize it is to create a new hash table about twice as large and hash all of the elements of the hash table into the new table using its hash function.

Rehashing is expensive, so it should only be done when necessary:

1. When an insertion fails, or
2. When the table gets half full, or
3. When the table load factor reaches some predefined value.