9
Hash Tables

9.1 Calculating instead of Comparing

In what we saw so far, the search time for an element could be reduced from $O(n)$ to $O(\log n)$, which is a significant improvement. However, even logarithmic time might be, for certain applications, too high a price to pay. Consider, for instance, a very large computer program, with thousands of variables. We sometimes write quite complicated arithmetic expressions, involving many of these variables, and having to search for the values of each of them may impair the execution time.

We can get faster access by changing the approach altogether. In previous chapters, an element x was sought for by comparing its value to that of some elements stored in a list or a tree, until x was found or could be declared as missing. The new approach is to access the data structure at an address which is not the result of a comparison, but which can be calculated by means of x alone.

The basic idea is not new and is familiar to everybody. Many institutions, like schools, stores, hospitals, or clubs, used to assign *membership numbers* to their students, clients, patients, or members. If a group had n adherents, these numbers were 1 to n, and information about member i was stored in a table T at entry $T[i]$. With the growing number of such organizations, it became unpractical to manage different indices for a single person, and it was preferred to use some official, unique, identification number, which is provided in many countries to each of their inhabitants as ID or Social Security number. These numbers need to be larger and consist generally of 8 to 10 digits.

It is obviously not reasonable to allocate a table with a number of entries of the order of 1 billion, only to get direct access to some n elements by using their ID numbers as index, when n may be a few hundred. The 8- to 10-digit ID number should thus be shortened. This has been realized by bank tellers long

ago, when they had to compile, by hand, a list of the check numbers for a batch of checks: they do not use the full check numbers, but rather only their, say, three rightmost digits, hoping that there are no clashes.

This simple idea has then been generalized as follows. We assume that the identifiers of the elements, large numbers or character strings, are drawn from some very large, or even infinite universe $\mathcal{U}$; we further consider a table T with M entries, into which the information about the elements has to be stored. The problem is to find an appropriate function

$$h : \mathcal{U} \longrightarrow \{0, 1, \ldots, M-1\}$$

so that the element identified by X should be stored in T at $h(X)$. Additional constraints on the function h are as follows:

(i) It should be easy to evaluate $h(X)$ for any X.
(ii) The values $h(X)$ should be evenly distributed over the possible table entries.

If the first condition is not met, the evaluation of $h(X)$ might take longer than the sequence of comparisons it was designed to replace. The second condition is needed to assure that there are not too many collisions. Hoping that there will be no collisions at all is not realistic: the universe $\mathcal{U}$ is often much larger than the set $\{0, 1, \ldots, M-1\}$, so the function h cannot be injective, and since we have generally no control of which subset of $\mathcal{U}$ will be chosen, we shall have to cope with pairs of different identifiers X and Y, such that $h(X) = h(Y)$.

The challenge is therefore to come up with a good compromise between the two demands. If the choice of the function h is too simplistic, like in the example of the bank teller, for which $h(X) = X \bmod 1000$, the distribution of the $h(X)$ values can be strongly biased for certain applications. The problem with considering only the last few digits is that all the other digits of X have no influence on $h(X)$. This could give biased results for subsets of $\mathcal{U}$ in which mostly these other digits vary. We would prefer a function for which every bit in X is significant, and the resulting value should be obtained by reshuffling all the input bits in some sophisticated way, that is nevertheless fast to implement. Such functions are therefore known under the name of *hashing* or *hash* functions, and the tables storing elements at indices obtained by applying hash functions are called *hash tables*.

Our study of hashing will first concentrate on possible functions, and then deal with various strategies to handle the unavoidable collisions. We also assume for the moment that elements may be added or searched for, but not deleted from the table. Deletion is problematic for hashing, and we shall deal with it later.

9.2 Hash Functions

The following is an example of a simple hash function, known as *Middle square method* and attributed to John von Neumann. Starting with a natural number $n > 0$, its hash value should fit into a table of size 2^m, for some given integer m, so that the index should consist of m bits. One can, of course, represent n in its standard binary form in $k = 1 + \lfloor \log_2 n \rfloor$ bits, and take a subset of m of these bits. Taking the rightmost $m = 3$ bits was the check number example, and any other proper subset would have the same flaw of not letting all the input bits influence the hash value. To overcome this, take first the square of n. If the binary form of n is $n = b_{k-1} \cdots b_2 b_1 b_0$, then the rightmost bit of n^2 is b_0^2. The next to rightmost bit is $b_1 b_0 \oplus b_0 b_1$, where $\oplus$ stands for addition modulo 2, and will always be 0. The closer the bit position i is getting to the middle of the $2k$ bits representing n^2, the more bits of the representation of n are involved in the determination of the bit at position i. The hash function will therefore be defined as the middle m bits of the $2k$ bits of n^2. Formally, let $\ell = k - \frac{m}{2}$ be the number of bits to be ignored at the right end of n^2, then

$$h(n) = \left(\lfloor n^2 / 2^\ell \rfloor \right) \bmod 2^m.$$

For example, consider $n = 1234567$ and $m = 16$ bits. The number of necessary bits to represent n is $k = 21$. Squaring n, one gets $n^2 = 1524155677489$, which is

$$010110001011011111011000001111 1101100110001$$

in binary. The decimal value of the emphasized middle 16 bits is 62991, so we conclude that $h(1234567) = 62991$. The middle square method has been suggested as a pseudo random number generator, but it fails many randomness tests and is not recommended. It is brought here only as example for the construction of hash functions.

The next example, taken from the data compression component of some operating system, will illustrate the choices in the design of a hashing function. The universe $\mathcal{U}$ consists of all the possible character pairs, and they are to be hashed into a table T with 2048 entries. An index into T is accordingly of length 11 bits, but, using the standard 8-bit ASCII representation for a character, the input consists of 16 bits. Figure 9.1 shows the evaluation of the hash function for the pair (Y, V).

The characters are first converted into their binary ASCII equivalents and then moved together, so that the overlapping part is of length 5 bits. Then some Boolean operation should be applied: an OR operation would produce too many 1-bits, and an AND operation would favor 0-bits, so a better choice seems to be

Y	0	1	0	1	1	0	0	1			
V				0	1	0	1	0	1	1	0
h(YV)	0	1	0	1	0	0	1	1	1	1	0

Figure 9.1. Example of hashing two characters to 11 bits.

to apply XOR. The mathematical definition is thus

$$h(x, y) = (8 \times x) \text{ XOR } y,$$

where the multiplication by 8 simulates a left shift of 3 bits.

While at first sight this function seems to fulfill all the constraints, applying it to an input file consisting mainly of textual data gives disastrous results: practically all the items are then squeezed into just one quarter of the table, between indices 512 and 1023. The reason for this strange behavior is that the ASCII representations of most printable characters, including all lower and upper case letters, start with 01 in their two leftmost bits. Since the overlapping part of the two bit-strings in the definition of the hash function h covers only the five rightmost bits of the first parameter, most of the resulting 11-bit values will remain with this 01 bit-pattern in their two leftmost bits, as seen in the example of Figure 9.1.

To rectify this deficiency, the bit-strings are cyclically shifted so as to move the leading 01 bits to the overlapping part; they then get a chance to be modified by the XOR operation. Figure 9.2 shows the result of this amended function, in which the first parameter is cyclically shifted left by 4 bits, and the second parameter by 5 bits. The previously leading 01 bits are emphasized to show their new positions.

The mathematical definition of the modified function is

$$h(x, y) = \Big(\big((x \bmod 16) + \lfloor x/16 \rfloor\big) \times 8\Big) \text{ XOR } \big((y \bmod 32) + \lfloor y/32 \rfloor\big).$$

The resulting function takes only slightly more time to be evaluated, as a cyclic shift is a basic operation in assembly languages, but the distribution of the values over the possible table entries will be much more uniform, even for purely textual input.

The simplest way of adapting the range of a hash function to fit a given table size M is to use the remainder function $h(X) = X \bmod M$, returning

Y	1	0	0	1	**0**	**1**	0	1			
V				1	1	0	**0**	**1**	0	1	0
h(YV)	1	0	0	0	1	1	0	0	0	1	0

Figure 9.2. Example of improved hashing of two characters to 11 bits.

values in the range $[0, 1, \ldots, M-1]$. We saw already that the usefulness of such a function may depend on the value of M. For instance, as explained earlier, $M = 1000$ is not a good choice. For similar reasons, $M = 1024$ is not much better: even if there is no obvious connection between the digits of X and $h(X) = X \bmod 1024$, notice that $1024 = 2^{10}$ is a power of 2, so if X is given in binary, $h(X)$ is again just the extraction of the 10 rightmost bits. Returning to the previous example, the binary representation of $X = 1234567$ is 10010110101010000111, and that of $X \bmod 1024 = 647$ is 010000111.

Actually, M does not need to be a power of any number to yield a skewed distribution for certain input sequences. If M is even, then every even input value X would imply that $X \bmod M$ is even. So if all the elements to be hashed happen to be even, half of the hash table, the entries with odd indices, would be empty! If M is a multiple of 3, a similar bias can be observed, and in fact, M should not be chosen as a multiple of 2, or 3, or 5, or any other smaller constant. This restricts the potential choices for M only to numbers that are not multiples of any others – that is, M should be a prime number.

Background Concept: Prime Numbers

A *prime number* is an integer larger than 1 that is divisible only by 1 and by itself. Many properties of prime numbers were known already to the Ancient Greek, but only in the last 100 years or so has their usefulness been discovered for many applications, in particular in cryptography. A natural number that is not prime is called *composite*.

There are infinitely many prime numbers, and the number of prime numbers up to n is approximately $\frac{n}{\ln n}$. The difference between consecutive prime numbers can become arbitrarily large, but can be as small as 2, for example, for 99989 and 99991. Such pairs are called *twin primes*, and it is not known whether there are infinitely many of them.

The simplest way to check if a number p is prime is by trying to divide it by smaller primes 2, 3, 5, up to $\sqrt{p}$. If no divisor has been found, then p must be prime. This procedure is, however, not feasible for the large primes, spanning hundreds of bits, used in cryptographic applications. To check whether $n = 2^{400} - 3$ is prime (it is not!), one would need about 2^{200} divisions, which requires billions of CPU-years even on the most powerful computers. Fortunately, there are *probabilistic algorithms* allowing to test the primality of a number in logarithmic time.

The suggested algorithm is therefore to choose a prime number M which is close to the size we intended to allocate to the hash table, and to use as

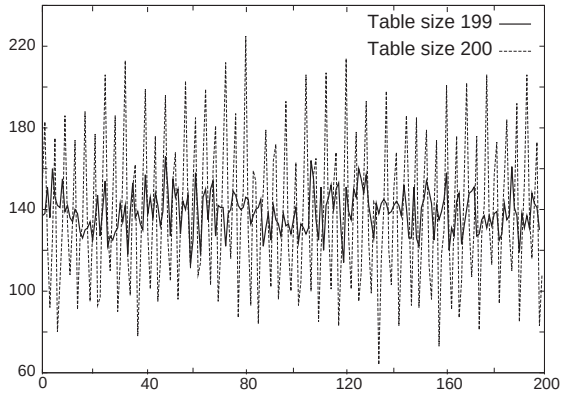

Figure 9.3. Distribution of hash values for King James Bible.

hash function

$$h(X) = X \bmod M.$$

This is an interesting case of a data structure for which it might be worth not to exploit all the available memory to get a better performance. For example, if there is enough space for 1000 entries, it may be better to prefer $M = 997$.

The following test shows the superiority of a prime sized hash table. The King James Version of the English Bible is used as input, viewing it as a sequence of blocks of length 4 characters each. There are 813352 such blocks B, of which 27590 are different ones, and using the ASCII representation of each character, each block is considered representing a 4-byte = 32 bit integer. The hash functions are B mod 200 and B mod 199. Figure 9.3 plots the number of hashes to entry i, for $0 \leq i < 200$, for both functions.

The solid line corresponds to the prime value $M = 199$, and can be seen fluctuating around the expected value $27590/199 = 139$. The lighter, broken, line corresponds to the nonprime size $M = 200$: the fluctuations are much more accentuated, and there are peaks at regular intervals of size 8, both above and below the average middle line. Such regular patterns are contradicting the expected uniformity.

And what if the element to be hashed is not a number, but a character string? This may happen, as in the example of variable names in a large computer program. There is an obvious way to convert a character string into an integer, using a standard encoding of the characters, like ASCII. The C Programming language even identifies characters with small integers from 0 to 255. To get the number N corresponding to a character string S, just concatenate the ASCII encodings of the individual characters of S and consider the resulting bit-string

as the binary representation of N. For example, for $S = \mathsf{AB}$, we have $\text{ASCII}(\mathsf{A}) = 65 = 01000001$ and $\text{ASCII}(\mathsf{B}) = 66 = 01000010$; the binary representation of N is thus 0100000101000010, so $N = 16706$.

This works for input strings of any length, just that the numbers to be manipulated may get out of control. Taking, for instance, the string `A-LONG-EXAMPLE-STRING` as input, the corresponding integer N would be 168 bits long and correspond, in decimal notation, to

$$95\,256\,212\,776\,338\,074\,265\,619\,361\,588\,642\,035\,558\,695\,416\,974\,919.$$

Most programming languages are not able to process such numbers with their standard tools. What comes to our rescue are the handy properties of modular arithmetic.

Background Concept: Modular Arithmetic

Let p be a fixed integer larger than 1 we shall call the *modulus*. For any natural number n, $n \bmod p$ is defined as the remainder of the division of n by p, that is $n \bmod p = r$ if $0 \leq r < n$ and n can be written as $n = kp + r$ for some integer k. As the modulus is assumed to be fixed, we may abbreviate the notation by setting $\overline{n} = n \bmod p$, and call it the *modulo* function. The following properties follow directly from this definition,

First note that $\overline{n} = \overline{\overline{n}}$, implying that the modulo function can be applied once or more times without changing the result. It follows that for all integers n and m,

$$\overline{n+m} = \overline{\overline{n} + \overline{m}} \qquad \text{and} \qquad \overline{n \times m} = \overline{\overline{n} \times \overline{m}},$$

which enables the decomposition of one operation on large arguments into several operations on smaller ones. For example, for $p = 13$, let us calculate $\overline{29 \times 72} = \overline{2088}$. Most of us will find it difficult to multiply these number in their heads, and even more so to divide the result by 13 to find the remainder. Applying the modulo repeatedly simplifies this task to

$$\overline{\overline{29} \times \overline{72}} = \overline{3 \times 7} = \overline{21} = 8,$$

for the evaluation of which we need no paper or pencil.

For the evaluation of a large integer, originating from a character string, modulo p, suppose the string is $S = c_t c_{t-1} \cdots c_2 c_1 c_0$, where the c_i are characters we shall identify with numbers between 0 and 255, like in C. The numerical value N of S can be written as

$$N = c_t 256^t + c_{t-1} 256^{t-1} + \cdots + c_2 256^2 + c_1 256 + c_0.$$

By *Horner's rule*, this can be rewritten as

$$N = \big(\cdots\big((c_t 256 + c_{t-1})256 + c_{t-2}\big)256 + \cdots\big)256 + c_0.$$

When evaluating $\overline{N}$, we may apply the modulo function on each of the nested parentheses, yielding

$$\overline{N} = \overline{\big(\cdots\overline{\big(\overline{(c_t 256 + c_{t-1})}256 + c_{t-2}\big)}256 + \cdots\big)256 + c_0},$$

so that at no stage of the calculation does the current value exceed p^2. Returning to the string `A-LONG-EXAMPLE-STRING` and choosing the modulus as the prime number $p = 2^{16} - 39 = 65497$, we get that

$$\overline{N} = \overline{\big(\cdots\overline{\big(\overline{(65 \cdot 256 + 45)}256 + 76\big)}256 + \cdots\big)256 + 71} = 20450,$$

where all the operations could be performed with 32-bit integers. The general algorithm is given by

N modulo p
    $a \longleftarrow 0$
    for $i \leftarrow t$ to 0 step -1
        $a \longleftarrow \big(a \times 256 + c_i\big) \bmod p$
    return a

9.3 Handling Collisions

For a small enough set of arguments, it may be possible to model a custom tailored hash function that is injective and thus avoids any collision. Refer, for example, to the first 80 digits of π following the decimal point, and consider them as a sequence of 20 four-digit numbers that we wish to hash into a table of size 33, filling the table only up to about 60%, so there is ample space. The numbers are

1415, 9265, 3589, 7932, 3846, 2643, 3832, 7950, 2884, 1971,

6939, 9375, 1058, 2097, 4944, 5923, 0781, 6406, 2862, 0899.

It may take a while and require several trial and error iterations, but an appropriate injective function can certainly be found. Here is a possible one: denote the four digits of a given number by m, c, x and i, from left to right,

then

$$h(mcxi) = \left(3m + \left\lfloor \frac{c}{3} \right\rfloor + \left\lfloor \frac{7\,x\,i}{4} \right\rfloor + \Big(x + i = c\Big)\right) \bmod 33.$$

The sequence of the hash values for the preceding numbers is then

12, 13, 4, 1, 20, 29, 21, 24, 31, 18,
2, 23, 7, 17, 10, 28, 16, 19, 30, 11.

Suppose, however, that we would like to extend the sequence to include also the following four-digit number of the digits of π, which is 8628. Applying h gives $h(8628) = 21$, which collides with $h(3832)$. To repair this, the whole work of the hash function design has to be restarted from scratch. Just choosing the function at random will most probably not help: the number of functions from a set of size 21 to a set of size 33 is 33^{21}, and the number of one-to-one functions for such sets is $\frac{33!}{12!}$. The probability of choosing an injective function, by selecting arbitrarily one of the possible functions, is the ratio of these numbers, which is 0.0002.

We have generally no a priori knowledge about the distribution of the elements to be stored in a hash table, but even if they are chosen at random from a much larger set, there are high chances that some of the numbers may cause a conflict, as illustrated by the following famous fact.

Background Concept: The Birthday Paradox

What is the probability that in a random group of 23 people, there are at least two which share the same birthday? Most of us would estimate it to be very low, yet it can be shown to be larger than $\frac{1}{2}$, that is, it is more likely to occur than not. This seems to be so surprising that many are reluctant to accept it, even after having seen the mathematical proof, which explains it being named a paradox.

Consider the complementing event of all the people in a group having different birthdays. More generally, a set of n elements is given, from which an element should be drawn at random k times, and we ask what the probability would be that all k elements will be different.

The probability for the second element not to be equal to the first is $\frac{n-1}{n}$. The probability for the third element to be different from both the first and the second is $\frac{n-2}{n}$, and in general, the probability of the event E_i, defined as the ith element being different from the preceding ones, is $\frac{n-i+1}{n}$. All the events $E_2, \ldots, E_k$ should occur simultaneously, and since they are

mutually independent, the probability of all k elements being different is

$$\left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right)\cdots\left(\frac{n-k+1}{n}\right)=\prod_{i=1}^{k-1}\left(1-\frac{i}{n}\right). \tag{9.1}$$

This is a product of factors that all seem close to 1, and if there are just a few factors, the product would still be close to 1. But in fact, the factors are slowly decreasing, while their number grows. There will thus be an index for which the product will not be so close to 1 any more. More formally, let us just take the last $\frac{k}{2}$ factors. Each of the left out factors being smaller than 1, the product will increase:

$$\prod_{i=1}^{k-1}\left(1-\frac{i}{n}\right)<\prod_{i=k/2}^{k-1}\left(1-\frac{i}{n}\right)<\prod_{i=k/2}^{k-1}\left(1-\frac{k/2}{n}\right)=\left(1-\frac{k/2}{n}\right)^{k/2}.$$

Substituting $k=2\sqrt{n}$, we get

$$\left(1-\frac{1}{\sqrt{n}}\right)^{\sqrt{n}}\xrightarrow[n\to\infty]{}\frac{1}{e}=0.368.$$

This shows that it suffices to choose randomly just of the order of $\sqrt{n}$ elements out on n, for large enough n, to get already collisions with probability larger than $\frac{1}{2}$. In particular, for $n=365$ and $k=23$, the exact value of the product in (9.1) is 0.493, so the probability of two people in a group of 23 having the same birthday is 0.507. For a group of 40 people, it would already be 0.891.

As example, consider the 25 first Nobel laureates, all those who got the price in the years 1901–1904. Among them, Élie Ducommun (Peace) and Svante Arrhenius (Chemistry) were both born on February 19. Moreover, Henri Becquerel (Physics) and Niels Ryberg Finsen (Medicine) shared a birthday on December 15.

The conclusion is that some strategy is needed resolving the case $h(X)=h(Y)$ for $X\neq Y$. One of the possibilities is to use the hash table as an array of size M for header elements, each pointing to a linked list. An element Y with $h(Y)=k$ will be appended at the end of the list headed by the kth element, as shown in Figure 9.4. To search for an element X, we first evaluate $\ell=h(X)$ and then search in the list headed by ℓ. In the worst case, all the elements might hash to the same location, but if the assumption of the even spread of the hash values holds, the average search time will just be $\frac{n}{M}$, where n is the number

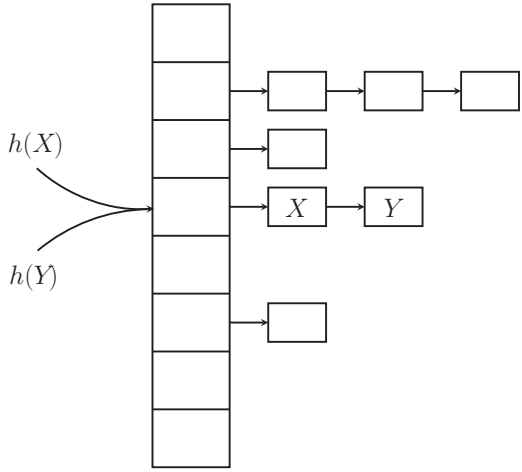

Figure 9.4. Collision resolution by chaining.

of elements to be inserted. If n and M are of the same order of magnitude, the average search time will be $O(1)$.

This approach is called *chaining* and has the advantage that the number of treated elements n might even exceed the size M of the table, so there is no need to have an exact estimate of n prior to the initialization of the hash table. However, for certain applications, one may wish to save the wasted space of the pointers, and prefer to store all the elements directly in the table itself. This can be achieved by the following methods, called *open addressing*.

The function $h(X)$ returns a value between 0 and $M-1$ that can be used as an index into the hash table T. If the spot is empty, X will be stored in T at index $h(X)$. Refer to Figure 9.5 to see what to do in case another element Z hashes to the same value $h(Z) = h(X)$, and should therefore be stored in the same place. A similar situation arises when we arrive to our assigned seat in the cinema, but find it occupied. Our reaction then probably depends on the size of the occupant; if he seems considerably stronger than us, we rather find an alternative place to sit. Here, too, Z will just move, for example, forward, to the closest free table entry, leaping over a block of occupied ones, shown in gray in the figure. In the example of Figure 9.5, the next spot is also occupied, by Y, so Z can only be stored in the following one. If the bottom of the table is reached, consider the table as cyclic and continue from the top. This is easily implemented by indexing the table entries from 0 to $M-1$ as shown, and incrementing the index by 1 modulo M during the search for an empty spot.

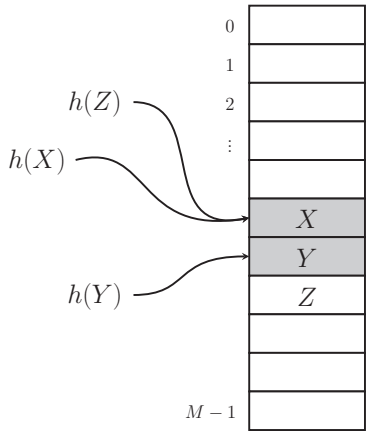

Figure 9.5. Open addressing using following empty location.

This works fine as long as only a small part of the table entries is used. When the table starts to fill up, the occupied cells will tend to form clusters. The problem is that the larger the cluster, the higher the probability of a new element being hashed into one of its locations, so the higher the probability of this specific cluster getting even larger – this contradicts the requested uniform spread of the values.

A better idea thus seems to be looking for an empty cell in jumps of size k, for some fixed number $k > 1$ of table entries. If k is chosen prime to M, that is, if the greatest common divisor of k and M, $\text{GCD}(k, M)$, is 1, then this jump policy will get us back to the starting point only after having visited all the entries in the table. This is yet another incentive to choose M as a prime number, since then any jump size k is prime to M, so every k is appropriate. In the example of Figure 9.6, $M = 13$ and $k = 5$. Since $h(Z) = 5$ is occupied by X, the current index is repeatedly increased by k. The sequence of the checked indices, called the *probe sequence*, is 5, 10, 15 mod $M = 2$, and finally 7. This is the first empty location in the order imposed by the probe sequence, so Z will be stored there.

The truth is, that this last change is only a cosmetic one: there will still be clusters just as before, they might only be harder to detect. To see this, consider a probe sequence visiting the entire table. For the example with $M = 13$ and $k = 5$, one such sequence could be 5, 10, 2, 7, 12, 4, 9, 1, 6, 11, 3, 8, 0. This is in fact a permutation of the table indices 0 to $M - 1$, and if we would rearrange the table entries in the order given by this permutation, the same clustering effect we observed earlier would also be evident here.

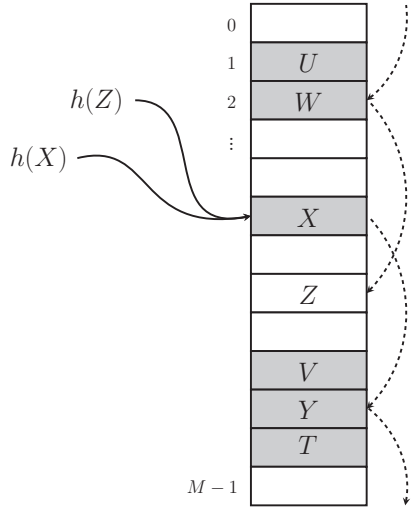

Figure 9.6. Open addressing using a probe sequence in jumps of size k.

To avoid these clusters, one can replace the fixed jump size k used in the case of a collision, by one which also depends on the element to be inserted. In fact, this suggests using two independent hash functions $h_1(X)$ and $h_2(X)$, which is why the method is called *double hashing*. The first attempt is to store X at the address

$$\ell \longleftarrow h_1(X).$$

If this entry is occupied, a jump size is defined by

$$k \longleftarrow h_2(X),$$

such that $1 \leq k < M$. The following attempts to store X are then, in order, at addresses

$$(\ell + k) \bmod M, \quad (\ell + 2k) \bmod M, \quad (\ell + 3k) \bmod M, \ldots,$$

until an empty location $(\ell + rk) \bmod M$ is found at the $(r + 1)$st index of the probe sequence. The search for an element X is done accordingly. One first checks whether X is stored at $h_1(X)$, and if not, one defines k as $h_2(X)$ and checks at addresses $h_1(X) + ik$, for $i = 1, 2, , \ldots$, until X is found or an empty location is encountered. In the latter case, the conclusion is that X is not in the table.

What would be a good choice for a second hash function $h_2(X)$? The requested range of values is between 1 and $M - 1$ (a jump size of 0 or M would

results in a probe sequence of identical indices), so

$$h_2(X) \longleftarrow 1 + X \bmod (M-1)$$

seems to be appropriate. But M has been chosen to be prime, thus $M-1$ will be even, and we saw already that the function $h(X) = X \bmod Q$, for a composite number Q, is not recommended. This leads to the suggestion of using

$$h_2(X) \longleftarrow 1 + X \bmod M',$$

where M' is the largest prime smaller than M. The largest possible range for $h_2(X)$ will then be obtained for $M' = M-2$. Indeed, such pairs of *twin* prime numbers, differing only by 2, do exist, as mentioned earlier.

Summarizing, to apply double hashing, we first choose a pair of twin primes M and $M-2$, close to the size we wish to allocate for the hash table. The functions can then be defined as

$$h_1(X) \longleftarrow X \bmod M$$
$$h_2(X) \longleftarrow 1 + X \bmod (M-2).$$

9.4 Analysis of Uniform Hashing

The idea of double hashing can be extended to the use of three, four, or more hashing functions. Ultimately, this leads to a theoretical method, called *uniform hashing*, in which the number of independent different functions used, $f_1, f_2, f_3, \ldots$, is not bounded. In practice, the two functions used in double hashing give already satisfying results, and the additional overhead of using more hash functions is not worth their marginal expected improvement. Nevertheless, we shall use uniform hashing as a theoretical model to evaluate the performance of double hashing, which has been shown to give very similar results, though using a more involved analysis.

Consider then an infinite sequence of hash functions $f_i(X)$, for $i \geq 1$. We assume that they all behave well, that is,

$$\forall i \geq 1 \quad \forall j \in \{0, \ldots, M-1\} \qquad \mathsf{Prob}\big(f_i(X) = j\big) = \frac{1}{M},$$

and that they are mutually independent. Deviating from the notation used earlier, for which $h_1(X)$ was an address in a table of size M, and $h_2(X)$ was a leap size, we shall now adopt a more symmetric definition, in which all the $f_i(X)$ are possible addresses in the table, so that in fact $f_1(X)$, $f_2(X)$, $f_3(X)$, ... is the

probe sequence for X. The insertion algorithm is accordingly to try the locations $f_i(X)$ in order, until the first empty space is found. This also explains why infinitely many functions are needed, and not just M: there is a (low) probability that $f_1(X) = f_2(X) = \cdots$, so one cannot assure that all the table entries have been checked, for any finite sequence of tests.

The quantities to be evaluated are the numbers of trials needed to insert a new element or to search for an element in the table. The analysis will be based on the following observations.

(i) A first observation is that the number of trials needed to insert some element X is equal to the number of trials to find X in subsequent searches. Moreover, even the sequences of comparisons will be identical, and they are a prefix of the probe sequence of X.

(ii) The number of collisions during an insertion depends on the number of elements already stored in the table. When the table is almost empty, there are good chances to find an empty spot already at the first trial. However, when the table is almost full, many more comparisons might be needed to get to an entry that is not yet occupied. If there are already n elements stored in the hash table, define

$$\alpha = \frac{n}{M}$$

as the *load factor* of the table. We expect the number of trials for insertion to be a function of α, where $0 \leq \alpha < 1$.

(iii) There is a difference between a successful and an unsuccessful search. An unsuccessful search is equivalent to the insertion of a new element. The number of trials is thus equal to the number of trials for insertion at the current load factor α. For a successful search, however, the number depends on when the given element has been inserted, and does *not* depend on the number of elements stored in the table at the time of the search. This is quite different from what we saw for search trees in Chapters 4, 5, and 6!

9.4.1 Insertion

Define C_n, for $0 \leq n \leq M$, as a random variable representing the number of comparisons needed to insert an element into the hash table in which n elements are already stored. Theoretically, C_n can be 1, or 2, or in fact any number, since the number of collisions is not necessarily bounded. We are interested in the mean value, or *expectation*, of C_n, usually denoted by $E(C_n)$. By definition, this expected value is the weighted sum of the possible values the random variable

C_n can assume, where the weights are the probabilities, i.e.,

$$E(C_n) = \sum_{i=1}^{\infty} i \, \mathsf{Prob}(C_n = i).$$

Abbreviating $\mathsf{Prob}(C_n = i)$ by p_i, and repeating the summation technique, used in the analysis of Heapify in Chapter 7, of writing a summation in rows and summing then by columns, we get

$$\begin{aligned} E(C_n) = \sum_{i=1}^{\infty} i\, p_i = \quad & p_1 \\ & + p_2 + p_2 \\ & + p_3 + p_3 + p_3 \\ & + \qquad \cdots . \end{aligned}$$

The first column of this infinite triangle is just $\sum_{i=1}^{\infty} p_i = 1$, the second column is $\sum_{i=2}^{\infty} p_i = \mathsf{Prob}(C_n \geq 2)$, etc., yielding

$$E(C_n) = \sum_{i=1}^{\infty} \mathsf{Prob}(C_n \geq i).$$

But the event $C_n \geq 2$ means that there has been a collision in the first trial, and this event occurs with probability α. The event $C_n \geq 3$ means that there were collisions both in the first and the second locations, so the probability of the event is α^2, where we have used the assumption that f_1 and f_2 are independent. In general, we get that the ith term will be α^i, so that

$$E(C_n) = 1 + \alpha + \alpha^2 + \alpha^3 + \cdots .$$

Background Concept: Infinite and Finite Summations

In the analysis of algorithms, one often encounters summations that are not given explicitly, but rather using an ellipsis ($\cdots$) from which a general term is supposed to be inferred. The following technique might be useful for getting a closed form of such summations. The example we shall follow here is a finite *geometric progression*,

$$A_n = 1 + \alpha + \alpha^2 + \cdots + \alpha^n. \tag{9.2}$$

We try to create another identity that should be similar to, but different from, (9.2). If we have similar equations, it might be possible to subtract them side by side and get thereby some useful information, as because of the similarity, many of the terms may cancel out. This leaves us with the problem of how to generate the other equation.

The use of ellipses is generally restricted to cases in which the reader may derive some regularity from the context. This is not unlike the number sequences given often in IQ tests, in which you have to guess the next element. So the next term in $6 + 7 + 8 + \cdots$ should apparently be 9, and $3 + 6 + 9 + \ldots$ is probably followed by 12. In our case, it seems that each term is obtained from the preceding one by multiplying by α. This suggests to try to multiply the whole identity by α, to get

$$\alpha A_n = \alpha + \alpha^2 + \alpha^3 + \cdots + \alpha^{n+1}. \tag{9.3}$$

Subtracting (9.3) from (9.2), each side of the equation separately, yields

$$A_n - \alpha A_n = 1 - \alpha^{n+1},$$

in which there are no more ellipses. The left-hand side is $(1 - \alpha)A_n$, so two cases have to be considered. If $\alpha = 1$, then $A_n = 1 + 1 + \cdots + 1 = n + 1$. Otherwise, we can divide by $(1 - \alpha)$ to get

$$A_n = \frac{1 - \alpha^{n+1}}{1 - \alpha}.$$

Returning to the expectation of C_n, the number of comparisons to insert the $(n + 1)$st element into a hash table,

$$E(C_n) = \lim_{n\to\infty} A_n = \frac{1}{1 - \alpha} = \frac{M}{M - n},$$

as $\alpha < 1$ because we assume that the table is not full. For example, if the table is filled to about half, that is $\alpha = \frac{1}{2}$, then we expect about 2 comparisons only. Note in particular, that this number does not depend on M, the size of the table! Even if both the table size and the number of inserted elements grow linearly, but keeping a constant ratio, the insertion time will be $O(1)$.

9.4.2 Search

As mentioned, the number of comparisons for an unsuccessful search is also given by C_n. For a successful search, we have to decide which of the stored elements will be sought. As measure for the search performance, we shall use C'_n, for $1 \le n \le M$, defined as the average of the number of searches, assuming that each of the n elements currently stored in the table is equally likely to be the target,

$$C'_n = \frac{C_0 + C_1 + \cdots + C_{n-1}}{n}.$$

Using the fact that the expectation is a linear function, this implies

$$E(C'_n) = \frac{1}{n}\sum_{j=0}^{n-1} E(C_j) = \frac{1}{n}\left(1 + \frac{M}{M-1} + \frac{M}{M-2} + \cdots + \frac{M}{M-n+1}\right)$$
$$= \frac{M}{n}\sum_{j=0}^{n-1}\frac{1}{M-j} = \frac{M}{n}\sum_{i=M-n+1}^{M}\frac{1}{i},$$

where the last equality follows from a change of parameters $i = M - j$ and from reversing the order of summation. This is a part of the *harmonic series* $\sum_{i=1}^{\infty}\frac{1}{i}$ which is known to be divergent. The question is, at what rate it goes to infinity. This is a good opportunity to recall a useful tool for the approximation of similar discrete sums.

Background Concept: Approximating a Sum by an Integral

Let f be a monotonic function defined on the integers, and let a and b be two given integers, with $a \le b$. A sum of the form

$$S = f(a) + f(a+1) + \cdots + f(b) = \sum_{i=a}^{b} f(i)$$

does frequently occur in the evaluation of the complexity of an algorithm. To approximate S, we assume that f can be extended to be defined on all the reals in the range $(a-1, b+1)$, not just on the integers, which is often the case for functions typically appearing in complexity expressions. We also assume that the function f is integrable on the given range.

A graphical interpretation of the sum S can be seen in Figure 9.7. The function is monotonically decreasing in our example, and the quantities $f(a), f(a+1), \ldots, f(b)$ are shown as bars connecting the x-axis with a point of the graph of the function. The problem is that we have not been taught to evaluate sums of lengths. But we know how to evaluate areas. This leads to the following idea, suggested in Figure 9.8.

Extend the bar of length $f(i)$ to its left, forming a rectangle of width 1 and height $f(i)$. The area of this rectangle is thus $f(i) \times 1 = f(i)$, so it is equal to the length of the bar. Of course, the purist will object that lengths and areas cannot be compared, since they use different units, nevertheless, the number measuring the length is equal to the number quantifying the area. Repeating for all i from a to b, we get that the sum $S = \sum_{i=a}^{b} f(i)$ we wish to estimate is equal to the area in form of a staircase, delimited

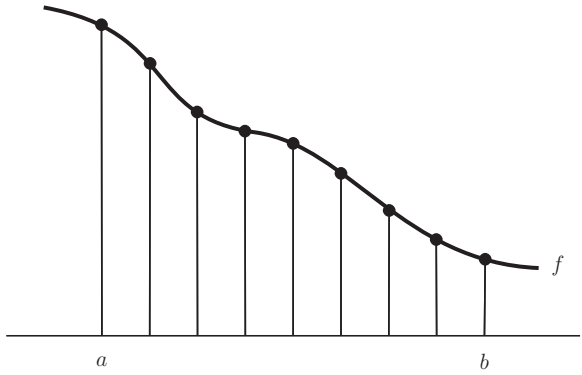

Figure 9.7. Graphical interpretation of $S = \sum_{i=a}^{b} f(i) =$ sum of the lengths of the bars.

by the broken line in Figure 9.8. But this staircase is completely below the graph of the function f, so the area of the staircase is smaller than the area below f, with appropriate limits, which is given by a definite integral:

$$S = \begin{matrix}\text{area of broken line} \\ \text{staircase}\end{matrix} \leq \int_{a-1}^{b} f(x)dx. \tag{9.4}$$

This gives an upper limit for the sum S. To get a double sided approximation, apply the same procedure again, extending this time the bars to the right, yielding the dotted line staircase of Figure 9.8. Note that the areas of both staircases are the same: just imagine the dotted line

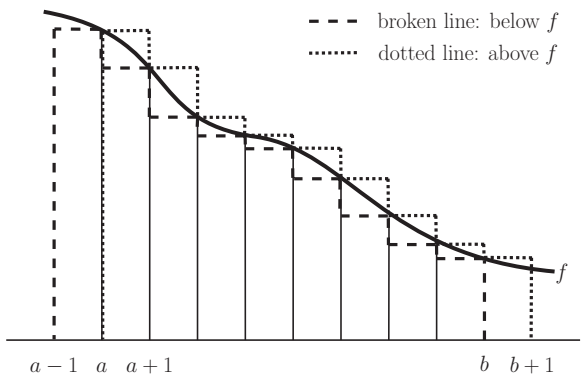

Figure 9.8. Replacing a sum of lengths by sums of areas.

staircase to be pushed to the left by one unit – it will then overlap with the broken line staircase. Now it is the area below the function that is bounded by the area below the dotted line staircase, but the limits have slightly changed. We get that

$$\int_a^{b+1} f(x)dx \;\leq\; S \;=\; \frac{\text{area of dotted line}}{\text{staircase}}. \tag{9.5}$$

A nice feature of this approximation is that it uses the integral of the same function twice, so only one primitive function has to be calculated. If $F(x)$ is the primitive function of $f(x)$, we get

$$F(b+1) - F(a) \;\leq\; S \;\leq\; F(b) - F(a-1).$$

If $f(x)$ is increasing instead of decreasing, the inequalities in (9.4) and (9.5) are reversed. If the function is not monotonic, we may try to approximate it separately on subranges on which it is monotonic, but care has to be taken to add the correct quantities when collecting the terms.

For the part of the harmonic series, we use the fact that the primitive function of $f(x) = \frac{1}{x}$ is $F(x) = \ln x$, the natural logarithm, and we get

$$\ln(M+1) - \ln(M-n+1) \;\leq\; \sum_{i=M-n+1}^{M} \frac{1}{i} \;\leq\; \ln(M) - \ln(M-n).$$

The sum is therefore approximated quite precisely as belonging to

$$\left[\ln \frac{M+1}{M-n+1}, \ln \frac{M}{M-n}\right],$$

which is a very small interval for large enough M and n. Substituting in the expression for $E(C'_n)$, we conclude that

$$E(C'_n) = \frac{M}{n} \sum_{i=M-n+1}^{M} \frac{1}{i} \simeq \frac{M}{n} \ln\left(\frac{M}{M-n}\right) = \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right).$$

For example, if the load factor is 0.9, the expected number of comparisons is only 2.56. Even for a table filled up to 99% of its capacity, the average

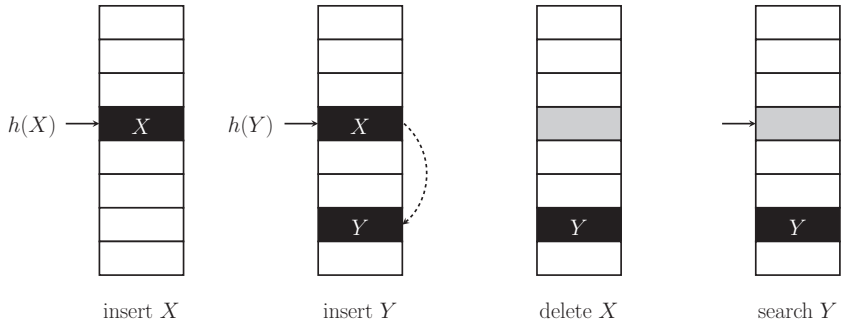

Figure 9.9. The problem with deletion in open addressing.

successful search does not take more than 4.65 trials, whereas a new insertion would be expected to require about 100 attempts.

9.5 Deletions from Hash Tables

So far, we have considered only insertions and searches. If an element should be removed, and the hash table is managed by means of chaining, the deletion of an element is done just like for the linked lists studied in Chapter 2. For open addressing. deletion is more problematic.

Figure 9.9 shows what might happen if an element to be deleted is just erased from the table. Proceeding from left to right, suppose an element X is inserted at $h(X)$, and that later, a second element Y is added, hashing to the same address, so that it will be stored elsewhere, say at the second address in its probe sequence. Suppose now that we remove X. A subsequent search for Y will fail, because it will try to locate Y at its first potential location $h(Y) = h(X)$, and since the spot is empty, it will erroneously conclude that Y is not in the table.

The problem can be overcome, if we define each entry of the table to be in one of three possible *states*: it may be either free (white), or occupied (black) or erased (gray). An erased cell will act like a free one for insertions, but like an occupied one for searches. That is, a new element may be inserted in the first free or erased entry encountered when scanning the element's probe sequence, but a search for X has to continue as long as the inspected entries are occupied or erased, until either X is found, or a free cell is reached for an unsuccessful search.

If the number of deletion is not large, this is actually a good solution. However, in a highly dynamic application, with many insertions and deletions, there

will be no free cells left in the long run, only occupied or erased ones, and the average search time may significantly increase.

9.6 Concluding Remarks

Summarizing this chapter, hash tables may be very efficient in many applications, though not in all of them. Their use is widespread, not only as a stand alone technique, but also as a standard building block in many others. If they are managed with care, they allow the reduction of insertion and access times from logarithmic to constant time. Here are just a few of the interesting applications of hashing.

(i) Deduplication: Consider a large backup system, in which the entire available electronic storage of some corporation has to be saved at regular time intervals to prevent the loss of data. The special feature of such backup data is that only a small fraction of it differs from the previously stored backup. This calls for a special special form of data compression, known as *deduplication*: trying to store duplicates only once. The challenge is, of course, to locate as much of the duplicated data as possible.

A standard deduplication system achieves its goal in the following way. Partition the input database into blocks, apply a hash function on each of them, and store the different hash values, along with the address of the corresponding block, in a hash table. For an update, we assume that a new copy of the data is given, which is also partitioned into similar blocks. The hash value of each of these new blocks is searched for in the table, and if it is found, there is a good chance that the new block is an exact copy of a previous one, so all one needs to store is a pointer to the earlier occurrence.

(ii) String matching: The basic problem of finding a string S of length m in a text T of length n has been treated in Chapter 1. The following *probabilistic* algorithm is due to R. M. Karp and M. O. Rabin. Let T_i denote the substring of T of length m starting at the ith character of T, that is,

$$T_i = T[i]T[i+1]\cdots T[i+m-1] \qquad \text{for } 1 \le i \le m-n+1.$$

Instead of comparing S to T_i for all possible values of i, which would require $O(m)$ comparisons for each i, apply a hash function h and compare $h(S)$ with $h(T_i)$, which can be done in $O(1)$ if the hash values are small enough, say up to $k \le 64$ bits. The overall complexity is then only $O(n)$, because $h(S)$ is evaluated only once, and, for a good choice of h, $h(T_i)$ can be derived from $h(T_{i-1})$ in constant time. A good hash function

could be $h(X) = X \bmod P$, where P is a randomly chosen prime number with k bits. For large enough k, it can be shown that the probability of an error is negligible.

(iii) Bloom filters: Another efficient way to check the membership of an element in a set, a problem we dealt with in Chapter 8. It uses a bit-vector A of size n bits and k independent hash functions $h_1, \ldots, h_k$, each returning integers between 1 and n. For each element x added to the set, the bits in positions $h_1(x), \ldots, h_k(x)$ of A are set to 1. Membership tests are then executed similarly: given an item y, one checks whether all the bits in positions $h_1(y), \ldots, h_k(y)$ are set. If not, then y cannot be in the set; if yes, then y might be in the set, but there is no evidence, as the bits may have been set by a combination of other items. If m items are to be stored, one can adapt the parameters n and k to the required efficiency and error probability.

(iv) Signature files: Two approaches for the processing of information retrieval systems have been mentioned in Section 8.1.1: the one, for smaller texts, based on pattern matching, as in Chapter 1, the other, for larger corpora, by means of inverted files. Actually, there is a third alternative, adapted to intermediate text sizes, known as *signature files,* and they are an extension of Bloom filters.

The text is partitioned into logical blocks A_i, like, say, one or more paragraphs. Each block is assigned a bit-vector $B(A_i)$ of length n bits and Bloom filters are used for each different significant word in A_i to set a part of the bits in $B(A_i)$ to 1. The parameters n and k are chosen as a function of the average number of terms in the blocks, so that the number of 1-bits in each vector is about $\frac{n}{2}$. Typical values could be $n = 128$ and $k = 2$ or 3. To process a query, each of its terms is hashed with the same functions, yielding a query-vector Q of length n. This helps to filter out a (hopefully large) part of the potential blocks to be retrieved. For each i, if

$$Q \text{ AND } B(A_i) \neq Q, \tag{9.6}$$

that is, if the 1-bits of Q are not a subset of those of $B(A_i)$, then the block A_i cannot contain all the terms of the query. If there is equality in (9.6), this does not assure that the block A_i does contain all the terms, but if the number of such blocks is low, each of them can be individually checked for the occurrence of the terms of Q.

On the other hand, the following situations are not handled well by hashing techniques, so if one of them occurs, it might be a better idea to choose some alternative.

(i) The worst case for hashing occurs when all the items hash to the same location. If this is not unreasonable, hashing should be avoided, as the corresponding insertion and search times may be linear in the number of stored items.

(ii) Open addressing requires that the number of elements to be inserted in the table does not exceed its size M. If it does, the table cannot be extended, a new table has to be allocated and all the elements need to be rehashed. This problem does not apply to chaining.

(iii) The only information gained from an unsuccessful search is that the item is not in the table. Consider the case of a list L of numbers and the problem of finding the element in L which is closest to some given number x. This is easily solved with an ordered list or a search tree, but a hash table may only locate the element itself, if it is in L. Thus hashing is not suitable for *approximate* searches.

(iv) In a *range query*, we want to retrieve all the elements in a list L between some constants a and b. When using a sorted list, we may locate the limits and return all the elements between them. For a search tree, the procedure is more involved, but two searches in the tree suffice to delimit the sought elements. In a hash table, locating a and b alone does not help.

Exercises

9.1 Every French child learns the following test, which helps detecting multiplication errors, in school. The test is based on a hashing function h. Given are two numbers X and Y we wish to multiply. To verify whether the calculated product $Z = X \times Y$ is correct, we apply h to the three numbers, and check if

$$h\big(h(X) \times h(Y)\big) = h(Z).$$

If the numbers differ, we must have made a mistake. If there is equality, the result is (most probably) correct. The function h is simply defined as taking repeatedly the sum of the digits, until only one digit is left, where 9 counts as 0.

For example, if $X = 123456$ and $Y = 98765432$, then $Z = 12193185172992$, and

$$h(X) = h(21) = 3, \qquad h(Y) = h(44) = 8, \qquad h(Z) = h(60) = 6,$$

and indeed, $h(3 \times 8) = h(24) = 6 = h(Z)$. To understand why the test is working, note that the digit 9 has no influence on the calculated result of the

function h, for example $h(49) = h(13) = 4$. The test is accordingly known under the name *la règle par neuf*, the rule by nine.

Give a simple mathematical expression for $h(X)$ and deduce from it the correctness of the test.

9.2 Consider the $n = 21$ numbers extracted from the expansion of π and listed at the beginning of Section 9.3: 1415, 9265, ..., 0899, 8628. Store them in a hash table of size 31 by means of double hashing using the functions

$$h_1(X) = X \bmod 31 \qquad h_2(X) = 1 + X \bmod 29.$$

Evaluate the actual value of C'_n and compare it to the theoretical expected value for this number of items if uniform hashing had been used. **Hint:** To check your results, the ratio of the two numbers should be 1.131.

9.3 Give an estimate for the following sums:

$$\sum_{i=2}^{n} i \log i, \qquad \sum_{i=1}^{n} i^3 \sqrt{i}, \qquad \sum_{i=1}^{n} \arctan i.$$

9.4 The following hash function h has been suggested to store a set of real numbers x such that $0 \leq x \leq 1$ in a table of size N: $h(x) = \lfloor Nx^2 \rfloor$. If the input values are uniformly distributed on the interval [0, 1], is it also true that the hash values $h(X)$ are uniformly distributed on the range $[0, N]$? What is the conclusion?

9.5 Using uniform hashing, you would like to store 1000 elements in a hash table of size M.

(a) What should be the size M of the table if you wish to get an average of 2.5 comparisons for the last inserted element?

(b) If you choose $M = 1500$, what is the average number of trials to find one of the stored elements, if they all have equal chance to be the element to be searched for?