be declared, and the ID number used to directly index the array. In this example, the first $1{,}000$ pointers in the array would definitely never be used, and perhaps many of the other pointers would also be left `NULL`. The actual number of pointers used would equal the number of staff being handled, so there might be thousands of unused pointers. Nevertheless, the memory space they consume is warranted, as the resultant search structure allows searching in $O(1)$ time – that is, searching in time that is constant and independent of the number n of items stored in the dictionary.

But what if the domain of key values is not so conveniently compact? What if staff ID numbers are eight-digits long, or if `staff_t` records must be searched by name, which is a character string? The fundamental idea in hashing is to create a compact set of integers from *any* set of keys, and then use those integers to directly index a table of pointers.

For example, suppose that staff ID numbers are eight-digit values in the range $10{,}000{,}000$ to $99{,}999{,}999$. If there are only $5{,}000$ different staff at any given time, then declaring an array of 100 million pointers is outrageously wasteful, and cannot be contemplated. Suppose instead that an array of $10{,}000$ pointers is declared, and the last four digits of the ID number is used to index the table.

Unless all staff ID numbers can be guaranteed to have a unique combination of the last four digits (which is a big ask), using those digits to index the table results in *collisions*, with different objects competing to use the same slot in the table of pointers. So instead of an array of $10{,}000$ pointers, an array of $10{,}000$ linked lists is created. Each list stores the set of objects that share the last four digits, and each search operation performs a linear search in one of the lists. If we are lucky, the number of long lists is relatively small, and most search operations are fast. After all, at least half of the lists must be empty, and if the original staff ID numbers are random, on average each non-empty list contains only one or two objects.

A hash function converts a value drawn from a large or indeterminate range into a seemingly random integer over a constrained range.

Taking the last four digits of each staff ID as the index is a simple and obvious transformation to reduce a large number into a more compact integer range, but has a serious drawback – it doesn't involve all of the digits of the original number, and should not be used in practice. For example, both $86{,}864{,}007$ and $87{,}864{,}007$ hash to the same location. If the staff ID values are genuinely random, this doesn't really matter. But real-life values are rarely random – for example, in a staff number like this, there might be a two-digit code to indicate year of commencement, then a three-digit code to indicate the cost center for salary payments, and then a three digit sequence number assigned starting at 000 for each cost center, each year. There will thus be far more ID codes ending with "$x000$" than with "$x999$", meaning that the randomness assumption is out the window, and with it the good "on average" behavior of the hash table.

The need for randomness should never be underestimated, and it is critically important that the hash function assigns seemingly random hash values, even when the input values are in some way similar. In particular, all components of the input value should affect the eventual constrained-range output value. So rather than extracting the last four digits, which is tantamount to taking the remainder mod $10{,}000$, a slightly different table size should be used. One way of ensuring that all digits contribute is to make the table size a prime number. For example, $9{,}973$ is the largest prime number less than $10{,}000$; and taking remainders mod $9{,}973$ transforms $86{,}864{,}007$ into $9{,}150$; the number $87{,}864{,}007$ into $1{,}877$; and $88{,}864{,}007$ into $4{,}577$. That is, it is probably preferable to design the hash function to use modulus $9{,}973$ and deliberately leave 27 of the $10{,}000$ possible hash values unused.

Hash functions should be constructed so that all parts of the key contribute to the calculated hash value.

Successful hashing of character strings – where "success" is measured according to the seeming randomness of the hash value for realistic sets of input data – is even more of a challenge. Figures 12.2 and 12.3 show one way of constructing a hash function for character strings. In order that the table can be any size (including round numbers such as $10{,}000$), great care is taken with the construction of the hash function. In Figure 12.2, the function `create_hash` adopts the usual pattern of allocating space for a structure, and then returning a pointer to it. It takes two arguments, a seed with which to start the random number generator, and a table size.

Each combination of `seed` and `tabsize` gives rise to a different set of `values` in the array associated with the `hash_t` structure. The consequence of this arrangement

```
#define NVALUES 10

typedef struct {
    unsigned nvalues;
    unsigned *values;
    unsigned tabsize;
} hash_t;

hash_t
*create_hash(unsigned seed, unsigned tabsize) {
    int i;
    hash_t *h;
    /* allocate the required memory space */
    h = malloc(sizeof(*h));
    assert(h != NULL);
    h->values = malloc(NVALUES*sizeof(*(h->values)));
    assert(h->values != NULL);
    h->nvalues = NVALUES;
    /* start the random number generator */
    srand(seed);
    /* then create a sequence of prime numbers from it */
    for (i=0; i<NVALUES; i++) {
        h->values[i] = nextprime(tabsize + rand()%tabsize);
    }
    h->tabsize = tabsize;
    return h;
}
```

Figure 12.2: Initializing a hash function for use on character strings.