objects to be processed in sorted order, but in many ways that is not a handicap, since they can always be sorted directly if the presentation order matters.

If the keys being used to access the stored items are known to be small integers, an array can be directly indexed using the key. For example, if the keys are staff ID numbers, and all are in the range 1,000 to 9,999, then an array of 10,000 `staff_t` pointers can 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 there would also be other pointers that were left `NULL`. Even so, the apparent waste might be worthwhile, 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 character strings are being used as the keys?

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 enormously wasteful, and cannot be contemplated.

But an array of 10,000 pointers is not unreasonable. If the last four digits of the ID number – or any other four digit combination derived mathematically from it – were used to index such a table, there might only be a relatively small number of *collisions* occur, in which different objects seek to use the same slot in the table of pointers. So instead of an array of 10,000 pointers, each of which can only store a single record, an array of 10,000 linked lists is used. Each secondary list stores the set of objects that share the same four-digit extracted value; and each search operation performs a linear search in just one of the secondary lists. With 10,000 possible lists, and only 5,000 keys to be stored, the number of long lists should be relatively small, and hence most search operations should be fast.

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 92,184,007 "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.

Getting a good distribution of values from the hash function is critically important, even when the input values are correlated, or non-uniformly distributed. 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 the same as taking the remainder mod 10,000, it is better to treat the staff ID number as if it were a character string, and seek a more general approach to hashing that applies to any string. A good hashing technique for strings can also be applied to numeric data.

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

Figures 12.2 and 12.3 show one way of constructing a hash function for character strings. In Figure 12.2, the function `hash_func_create` adopts the usual pattern of allocating space for a structure, and then returning a pointer to it. It takes just one argument, the size of the table that the calling program will be using. Note that this code constructs a hash function; not a hash table. It is assumed that the calling program manages the hash table.

Each time `hash_func_create` is called a different set of prime number `values` is generated in the array associated with the `hash_t` structure. The consequence of this arrangement is that even if two strings happen to collide in one hash function of a given size, in another hash function, even if of the same table size, they are no more likely collide than any other pair of randomly chosen strings. That is, these functions describe a *family* of hash functions, since more than one hash function can

```
#define NVALUES 20

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

hashfunc_t
*hash_func_create(unsigned tabsize) {
    int i;
    hashfunc_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;
    /* then create a sequence of prime numbers from it */
    for (i=0; i<NVALUES; i++) {
        /* assumes that srand() has already been called */
        h->values[i] = nextprime(tabsize + rand()%tabsize);
    }
    h->tabsize = tabsize;
    return h;
}
```

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