7.3 Hashing

Hashing **is based on the idea of distributing keys among a one-dimensional array** $H[0..m-1]$ **called a** ***hash table*****. The distribution is done by computing, for each of the keys, the value of some predefined function** h **called the** ***hash function*****. This function assigns an integer between 0 and** $m-1$**, called the** ***hash address*****, to a key.**

For example, if keys are nonnegative integers, a hash function can be of the form $h(K) = K \bmod m$**; obviously, the remainder of division by** m **is always between 0 and** $m-1$**. If keys are letters of some alphabet, we can first assign a letter its position in the alphabet, denoted here** $ord(K)$**, and then apply the same kind of a function used for integers. Finally, if** K **is a character string** $c_0c_1 \ldots c_{s-1}$**, we can use, as a very unsophisticated option,** $(\sum_{i=0}^{s-1} ord(c_i)) \bmod m$**. A better option is to compute** $h(K)$ **as follows:**[2]

$$h \leftarrow 0; \quad \text{for } i \leftarrow 0 \text{ to } s-1 \text{ do } h \leftarrow (h * C + ord(c_i)) \bmod m,$$

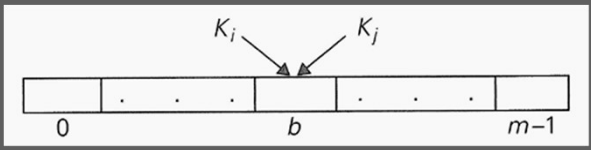
where C **is a constant larger than every** $ord(c_i)$**.**

In general, a hash function needs to satisfy somewhat conflicting requirements:

- Ahash table's size should not be excessively large compared to the number of keys, but it should be sufficient to not jeopardize the implementation's time efficiency (see below).
- A hash function needs to distribute keys among the cells of the hash table as evenly as possible. (This requirement makes it desirable, for most applications, to have a hash function dependent on all bits of a key, not just some of them.)
- A hash function has to be easy to compute.

Obviously, if we choose a hash table's size *m* to be smaller than the number of keys *n*, we will get *collisions*—a phenomenon of two (or more) keys being hashed into the same cell of the hash table (Figure 7.4). But collisions should be expected even if *m* is considerably larger than *n* (see Problem 5 in this section's exercises). In fact, in the worst case, all the keys could be hashed to the same cell of the hash table. Fortunately, with an appropriately chosen hash table size and a good hash function, this situation happens very rarely. Still, every hashing scheme must have a collision resolution mechanism.

This mechanism is different in the two principal versions of hashing: *open hashing* (also called *separate chaining*) and *closed hashing* (also called *open addressing*).



Open Hashing (Separate Chaining)

In open hashing, keys are stored in linked lists attached to cells of a hash table. Each list contains all the keys hashed to its cell. Consider, as an example, the following list of words:

A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED.

keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
hash addresses	1	9	6	10	7	11	11	12

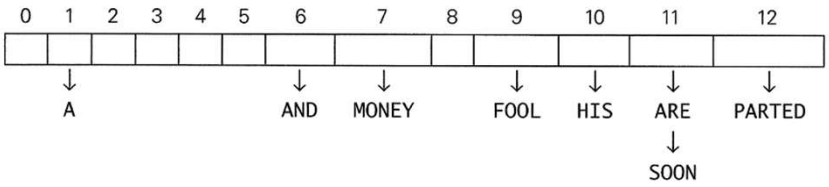


FIGURE 7.5 Example of a hash table construction with separate chaining.

In general, the efficiency of searching depends on the lengths of the linked lists, which, in turn, depend on the dictionary and table sizes, as well as the quality of the hash function. If the hash function distributes *n* keys among *m* cells of the hash table about evenly, each list will be about *n/m* keys long. The ratio $\alpha = n/m$, called the *load factor* of the hash table, plays a crucial role in the efficiency of hashing. In particular, the average number of pointers (chain links) inspected in successful searches, *S*, and unsuccessful searches, *U*, turns out to be

$$S \approx 1 + \frac{\alpha}{2} \quad \text{and} \quad U = \alpha, \tag{7.4}$$

Closed Hashing (Open Addressing)

In closed hashing, all keys are stored in the hash table itself without the use of linked lists. (Of course, this implies that the table size *m* must be at least as large as the number of keys *n*.) Different strategies can be employed for collision resolution. The simplest one—called *linear probing*—checks the cell following the one where the collision occurs. If that cell is empty, the new key is installed there; if the next cell is already occupied, the availability of that cell's immediate successor is checked, and so on. Note that if the end of the hash table is reached, the search is wrapped to the beginning of the table; i.e., it is treated as a circular array.

This method is illustrated in Figure 7.6 with the same word list and hash function used above to illustrate separate chaining.

keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
hash addresses	1	9	6	10	7	11	11	12

0	1	2	3	4	5	6	7	8	9	10	11	12
	A											
	A								FOOL			
	A					AND			FOOL			
	A					AND			FOOL	HIS		
	A					AND	MONEY		FOOL	HIS		
	A					AND	MONEY		FOOL	HIS	ARE	
	A					AND	MONEY		FOOL	HIS	ARE	SOON
PARTED	A					AND	MONEY		FOOL	HIS	ARE	SOON

FIGURE 7.6 Example of a hash table construction with linear probing.

The mathematical analysis of linear probing is a much more difficult problem than that of separate chaining.[3] The simplified versions of these results state that the average number of times the algorithm must access the hash table with the load factor α in successful and unsuccessful searches is, respectively,

$$S \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) \quad \text{and} \quad U \approx \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right) \qquad (7.5)$$

(and the accuracy of these approximations increases with larger sizes of the hash table). These numbers are surprisingly small even for densely populated tables, i.e., for large percentage values of α:

α	$\frac{1}{2}(1+\frac{1}{1-\alpha})$	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$
50%	1.5	2.5
75%	2.5	8.5
90%	5.5	50.5

Several other collision resolution strategies have been suggested to alleviate this problem. One of the most important is *double hashing*. Under this scheme, we use another hash function, $s(K)$, to determine a fixed increment for the probing sequence to be used after a collision at location $l = h(K)$:

$$(l + s(K)) \bmod m, \qquad (l + 2s(K)) \bmod m, \qquad \ldots . \qquad (7.6)$$

Mathematical analysis of double hashing has proved to be quite difficult. Some partial results and considerable practical experience with the method suggest that with good hashing functions—both primary and secondary—double hashing is superior to linear probing. But its performance also deteriorates when the table gets close to being full. A natural solution in such a situation is *rehashing*: the current table is scanned, and all its keys are relocated into a larger table.

- *Asymptotic time efficiency* With hashing, searching, insertion, and deletion can be implemented to take $\Theta(1)$ time on the average but $\Theta(n)$ time in the very unlikely worst case. For balanced search trees, the average time efficiencies are $\Theta(\log n)$ for both the average and worst cases.
- *Ordering preservation* Unlike balanced search trees, hashing does not assume existence of key ordering and usually does not preserve it. This makes hashing less suitable for applications that need to iterate over the keys in order or require range queries such as counting the number of keys between some lower and upper bounds.

Since its discovery in the 1950s by IBM researchers, hashing has found many important applications. In particular, it has become a standard technique for storing a symbol table—a table of a computer program's symbols generated during compilation. Hashing is quite handy for such AI applications as checking whether positions generated by a chess-playing computer program have already been considered. With some modifications, it has also proved to be useful for storing very large dictionaries on disks; this variation of hashing is called *extendible hashing*.

Since disk access is expensive compared with probes performed in the main memory, it is preferable to make many more probes than disk accesses. Accordingly, a location computed by a hash function in extendible hashing indicates a disk address of a *bucket* that can hold up to *b* keys. When a key's bucket is identified, all its keys are read into main memory and then searched for the key in question. In the next section, we discuss *B*-trees, a principal alternative for storing large dictionaries.