1.4 Open addressing

Another approach to collisions:

- no chaining; instead all items stored in table (see Fig. 1)

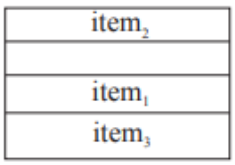

Figure 1: Open Addressing Table

- one item per slot $\implies m \geq n$
- hash function specifies *order* of slots to probe (try) for a key (for insert/search/delet not just one slot; **in math. notation**:

  We want to design a function h, with the property that for all $k \in \mathcal{U}$:

$$h : \mathcal{U} \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$$

universe of keys trial count slot in table

With open addressing, we require that for every key k, the probe sequence

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$$

be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$, so that every hash-table position is eventually considered as a slot for a new key as the table fills up. In the following pseudocode, we assume that the elements in the hash table T are keys with no information; the key k is identical to the element containing key k. Each slot contains either a key or NIL (if the slot is empty).

```
HASH-INSERT(T,k)
1   i ← 0
2    repeat j ← h(k,i)
3              if T[j] = NIL
4                  then T[j] ← k
5                       return j
6                  else i ← i + 1
7   until i = m
```

8 error "hash table overflow"

The algorithm for searching for key k probes the same sequence of slots that the insertion algorithm examined when key k was inserted. Therefore, the search can terminate (unsuccessfully) when it finds an empty slot, since k would have been inserted there and not later in its probe sequence.

```
HASH-SEARCH(T, k)
1   i ← 0
2    repeat j ← h(k, i)
3              if T[j]= j
4                  then return j
5               i ← i + 1
6     until T[j] = NIL or i = m
7   return NIL
```

Deletion from an open-address hash table is difficult. When we delete a key from slot i, we cannot simply mark that slot as empty by storing NIL in it. Doing so might make it impossible to retrieve any key k during whose insertion we had probed slot i and found it occupied. One solution is to mark the slot by storing in it the special value DELETED instead of NIL. We would then modify the procedure HASH-SEARCH so that it keeps on looking when it sees the value DELETED, while HASH-INSERT would treat such a slot as if it were empty so that a new key can be inserted. When we do this, though, the search times are no longer dependent on the load factor α, and for this reason chaining is more commonly selected as a collision resolution technique when keys must be deleted.

In our analysis, we make the assumption of uniform hashing: we assume that each key considered is equally likely to have any of the m! permutations of {0, 1, . . . , m - 1} as its probe sequence.

Three techniques are commonly used to compute the probe sequences required for open addressing: linear probing, quadratic probing, and double hashing. These techniques all guarantee that $\langle h(k, 1), h(k, 2), \ldots, h(k, m)\rangle$ is a permutation of $\langle 0, 1, \ldots, m - 1\rangle$ for each key k.

Linear probing

Given an ordinary hash function h': U → {0, 1, . . . , m - 1}, the method of linear probing uses the hash function

h(k,i) = (h'(k) + i) mod m

for i = 0,1,...,m - 1. Given key k, the first slot probed is T[h'(k)]. We next probe slot T[h'(k) + 1], and so on up to slot T[m - 1]. Then we wrap around to slots T[0], T[1], . . . , until we finally probe slot T[h'(k) - 1]. Since the initial probe position determines the entire probe sequence, only m distinct probe sequences are used with linear probing.

Linear probing is easy to implement, but it suffers from a problem known as primary clustering. Long runs of occupied slots build up, increasing the average search time. For example, if we have n = m/2 keys in the table, where every even-indexed slot is occupied and every odd-indexed slot is empty, then the average unsuccessful search takes 1.5 probes. If the first n = m/2 locations are the ones occupied, however, the average number of probes increases to about n/4 = m/8. Clusters are likely to arise, since if an empty slot is preceded by i full slots, then the probability that the empty slot is the next one filled is (i + 1)/m, compared with a probability of 1/m if the preceding slot was empty. Thus, runs of occupied slots tend to get longer, and linear probing is not a very good approximation to uniform hashing.

Quadratic probing

Quadratic probing uses a hash function of the form

$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m,$

where (as in linear probing) h' is an auxiliary hash function, c_1 and $c_2 \neq 0$ are auxiliary constants, and i = 0, 1, . . . , m - 1. The initial position probed is T[h'(k)]; later positions probed are offset by amounts that depend in a quadratic manner on the probe number i. This method works much better than linear probing, but to make full use of the hash table, the values of c_1, c_2, and m are constrained. Also, if two keys have the same initial probe position, then their probe sequences are the same, since $h(k_1, 0) = h(k_2, 0)$ implies $h(k_1, i) = h(k_2, i)$. This leads to a milder form of clustering, called secondary clustering. As in linear probing, the initial probe determines the entire sequence, so only m distinct probe sequences are used.

Double hashing

Double hashing is one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations. Double hashing uses a hash function of the form

$h(k, i) = (h_1(k) + i h_2(k)) \bmod m,$

where h_1 and h_2 are auxiliary hash functions. The initial position probed is $T[h_1(k)]$; successive probe positions are offset from previous positions by the amount $h_2(k)$, modulo m. Thus, unlike the case of linear or quadratic probing, the probe sequence here depends in two ways upon the key k, since the initial probe position, the offset, or both, may vary. Figure 1.5 gives an example of insertion by double hashing.

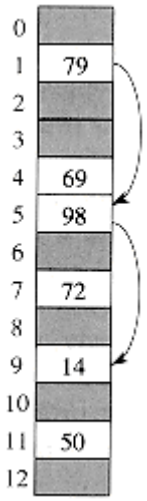

Figure 1.5 Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \bmod 13$ and $14 \equiv 3 \bmod 11$, the key 14 will be inserted into empty slot 9, after slots 1 and 5 have been examined and found to be already occupied.

We now analyze the expected number of probes for hashing with open addressing under the assumption of uniform hashing, beginning with an analysis of the number of probes made in an unsuccessful search.

See Theorem 11.6 and 11.8 from Book -T. H. Cormen, C. E. Leiserson , R. L. Rivest and C. Stein : Introduction to Algorithms, Third Edition ,The MIT Press Cambridge, Massachusetts London, England .

1.5 Perfect Hashing

We say a hash function is perfect for S if all look ups involve O(1) work. Here are now two methods for constructing perfect hash functions for a given set S.To create a perfect hashing scheme, we use two levels of hashing, with universal

hashing at each level. Figure given below illustrates the approach.

The first level is essentially the same as for hashing with chaining: we hash

the n keys into m slots using a hash function h carefully selected from a family of

universal hash functions.

Instead of making a linked list of the keys hashing to slot j , however, we use a

small secondary hash table S_j with an associated hash function h_j . By choosing

the hash functions h_j carefully, we can guarantee that there are no collisions at the

secondary level.

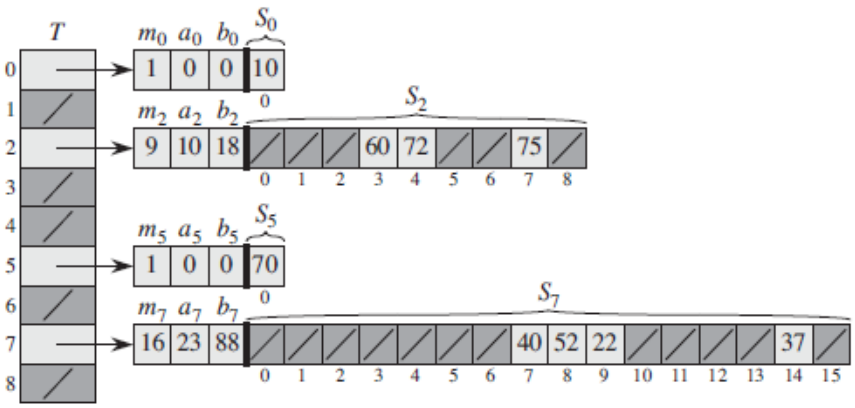

Figure 11.6 Using perfect hashing to store the set $K = \{10, 22, 37, 40, 52, 60, 70, 72, 75\}$. The outer hash function is $h(k) = ((ak + b) \bmod p) \bmod m$, where $a = 3$, $b = 42$, $p = 101$, and $m = 9$. For example, $h(75) = 2$, and so key 75 hashes to slot 2 of table T. A secondary hash table S_j stores all keys hashing to slot j. The size of hash table S_j is $m_j = n_j^2$, and the associated hash function is $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$. Since $h_2(75) = 7$, key 75 is stored in slot 7 of secondary hash table S_2. No collisions occur in any of the secondary hash tables, and so searching takes constant time in the worst case.

See Theorem 11.9 and 11.10 from Book -T. H. Cormen, C. E. Leiserson , R. L. Rivest and C. Stein : Introduction to Algorithms, Third Edition ,The MIT Press Cambridge, Massachusetts London, England .

Reference :

Text Book-T. H. Cormen, C. E. Leiserson , R. L. Rivest and C. Stein : Introduction to Algorithms, Third Edition ,The MIT Press Cambridge, Massachusetts London, England .