11. Hash Tables (Cont.)

11.3. Hash functions

We deal with issues on the design of good hash functions. A good hash function satisfies the assumption of **simple uniform hashing**:

Each key is equally likely to hash to any of the m slots in the hash table, independently of where any other key has hashed to.

Most hash functions assume that the universe of keys is the set $\mathbb{N} = \{0, 1, 2, \ldots\}$. We here consider the following schemes of hash function creation.

- Hashing by Division
- Hashing by Multiplication
- Universal Hashing

11.3.1 Division method

Map a key k into one of the m slots of a hash table by taking the remainder of k divided by m.
That is, the hash function is

$$h(k) = k \; mod \; m,$$

where $k \; mod \; m = k - \lfloor k/m \rfloor \cdot m$.

E.g., $m = 23$ and the key $k = 107$, then $h(k) = 15$.

- Certain m values, such as $m = 2^p$, should be avoided. Why?
- Good values for m are primes that are not too close to exact powers of 2. Why?

11.3.2 Multiplication method

The multiplication method operates in two steps.

- Step 1. Multiply the key k by a constant A in the range $0 < A < 1$, and **extract the fractional part of kA**.

- Step 2. Multiply this value by m and **take the floor of the result**. (Why?)

In short, the hash function is

$$h(k) = \lfloor m \cdot (kA \; mod \; 1) \rfloor ,$$

where $(kA \; mod \; 1)$ denotes the fractional part of kA, that is, $kA - \lfloor kA \rfloor$.

11.3.2 Multiplication method (cont.)

E.g., $m = 10000$, $k = 123456$, and $A = \frac{\sqrt{5}-1}{2} = 0.618033$,
then

$$\begin{aligned} h(k) &= \lfloor 10000 \cdot (123456 \cdot 0.61803\ldots mod\ 1) \rfloor \\ &= \lfloor 10000 \cdot (76300.0041151\ldots\ mod\ 1) \rfloor \\ &= \lfloor 10000 \cdot 0.0041151\ldots \rfloor \\ &= \lfloor 41.151\ldots \rfloor \\ &= 41. \end{aligned}$$

The advantage of this method is that the value choice of m is not critical.

11.3.3 Universal hashing

If a malicious adversary chooses keys to be hashed, then he can choose n keys that all hash to **the same slot**, yielding an average retrieval time of $\Theta(n)$. i.e., the simple uniform hashing assumption does not hold any more.

Any fixed hash function is vulnerable to this sort of worst case behavior.

The only effective way to improve the situation is to choose a hash function randomly in a way that is independent of the keys that are actually going to be stored.

This approach is referred to as **universal hashing**. Universal hashing is to select the hash function at random and at run time from a carefully designed collection of hash functions.

11.3.3 Universal hashing (cont.)

Let $\mathcal{H} = \{h_1, h_2, \ldots, h_l\}$ be a finite collection of hash functions that map a given universe U of keys into a range $\{0, 1, \ldots, m-1\}$.

Such a collection is said to be universal if for each pair of distinct keys $x, y \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(x) = h(y)$ is at most

$$\frac{|\mathcal{H}|}{m} = \frac{l}{m},$$

where l is the number of hash functions in $\mathcal{H}$.

11.4 Open Addressing

All elements in open addressing are stored in **the hash table itself**.

To perform insertion using open addressing, we successively probe the hash table until we find an empty slot in which to put the key.

The sequence of positions probed depends on the key being inserted. To determine which slots to probe, we extend the hash function to include the probe number as a second input. Thus, the hash function becomes

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

We require that for each k the probe sequence

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

is a permutation of $\{0, 1, \ldots, m-1\}$ so that every hash table position is eventually considered as a slot for a new key as the table fills up.

11.4 Open Addressing (cont.)

```
HASH_INSERT($T, k$)
1   $i \leftarrow 0$
2   repeat  $j \leftarrow h(k, i)$
3           if    $T[j] = NIL$
4           then  $T[j] \leftarrow k$
5                 return $j$
6           else  $i \leftarrow i + 1$
7   until   $i = m$
8   error “hash table overflow”
```

11.4 Open Addressing (cont.)

```
HASH_SEARCH($T,k$)
    1    $i \leftarrow 0$
    2    repeat  $j \leftarrow h(k,i)$
    3            if   $T[j] = k$
    4            then return $j$  /* the slot index $j$ */
    6            $i \leftarrow i+1$
    7    until   $T[j] = NIL$ or $i = m$
    8    return NIL
```

What about the DELETE operation?

11.4 Probing

11.4 Linear Probing

Given an ordinary hash function $h': U \to \{0, 1, \ldots, m-1\}$, linear probing uses the hash function

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

for $i = 0, 1 \ldots, m-1$.

Given key k, the 1st slot is $T[h'(k)]$, the 2nd slot is $T[h'(k)+1]$, and so on up to slot $T[m-1]$. Then, we wrap around to slots $T[0], T[1], \ldots$ until we finally probe slot $T[h'(k)-1]$.

Disadvantage: **the primary clustering problem**.

11.4 Quadratic Probing

Quadratic probing uses a hash function of the form

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

where h' is an auxiliary hash function, c_1 and c_2 are constants.

This method works much better than linear probing, but to make full use of the hash table, the values c_1 , c_2 and m are subject to certain constraints.

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 $h(k_1, i) = h(k_2, i)$.

Disadvantage: **the secondary clustering problem**.

11.4 Double hashing

Doubling 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

$$h(k,i) = [h_1(k) + i \cdot h_2(k)] \; mod \; m,$$

where h_1 and h_2 are auxiliary hash functions.

- The initial position is $T[h_1(k)]$
- A successive probe position is offset from its previous position by the amount $h_2(k)$ modulo m.

11.4 Double hashing

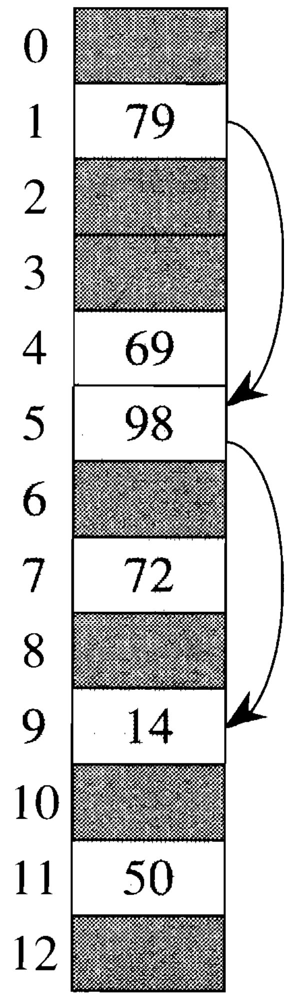
Example:

Insertion using double hashing.

$h_1(k) = k \; mod \; 13$

$h_2(k) = 1 + (k \; mod \; 11)$

Key 14 is to be inserted.



11.4 Double hashing (cont.)

The value $h_2(k)$ must be relatively prime to the hash table size m for the entire hash table to be searched. Otherwise, m and $h_2(k)$ have greatest common divisor $d > 1$ for some key k, and the search for key k only examines $(1/d) \cdot m$ slots of the hash table. This can for example be achieved by choosing m as a prime and by defining

$$h_1(k) = k \ mod \ m,$$

$$h_2(k) = 1 + (k \ mod \ m')$$

where m' is chosen to be slightly less than m (say $m - 1$ or $m - 2$).

For example, given the values $k = 123456$ and $m = 701$, we have $h_1(k) = 80$ and $h_2(k) = 257$, we first probe slot (position) 80 and then every 257th slot from this position is examined.

11.4 Double hashing (cont.)

Double hashing represents an improvement over linear or quadratic probings in that $\Theta(m^2)$ probe sequences, rather than $\Theta(m)$, are used, since each possible $(h_1(k), h_2(k))$ pair yields a distinct probe sequence with $0 \leq h_1(k), h_2(k) \leq m-1$.

As we vary the value of key k, the initial probe position and the offset may vary independently.