# Lecture 7: Hashing III: Open Addressing

## Lecture Overview

- Open Addressing, Probing Strategies
- Uniform Hashing, Analysis
- Cryptographic Hashing

## Readings

CLRS Chapter 11.4 (and 11.3.3 and 11.5 if interested)

## Open Addressing

*Another approach to collisions:*

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

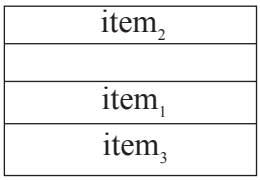


Figure 1: Open Addressing Table

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

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

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

universe of keys trial count slot in table

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

is a permutation of $0, 1, \ldots, m-1$. i.e. if I keep trying $h(k,i)$ for increasing $i$, I will eventually hit all slots of the table.

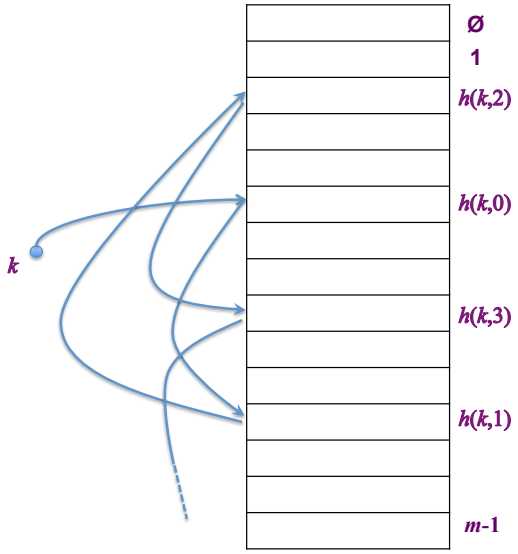


Figure 2: Order of Probes

**Insert(k,v)** : Keep probing until an empty slot is found. Insert item into that slot.

```
for i in xrange(m):
    if T[h(k,i)] is None:          # empty slot
        T[h(k,i)] = (k,v)          # store item
        return
raise 'full'
```

**Example:** Insert $k = 496$

**Search(k)**: As long as the slots you encounter by probing are occupied by keys $\neq k$, keep probing until you either encounter $k$ or find an empty slot—return *success* or *failure* respectively.

```
for i in xrange(m):
    if   T[h(k,i)] is None:        # empty slot?
         return None               # end of "chain"
    elif T[h(k,i)][Ø] == k:        # matching key
         return T[h(k,i)]          # return item
return None                        # exhausted table
```

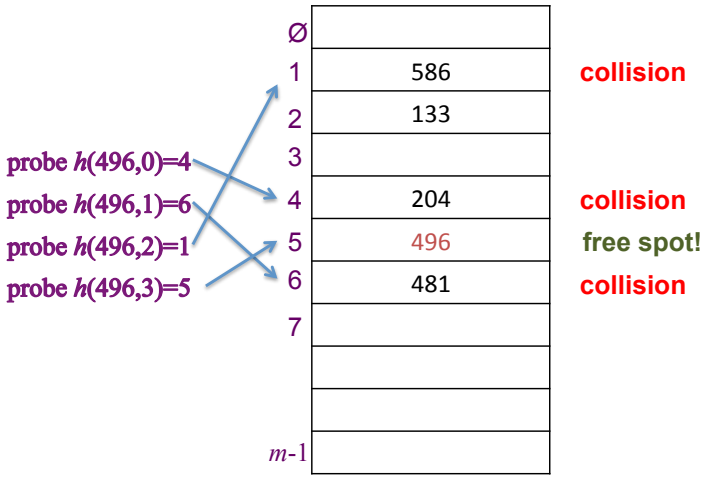


Figure 3: Insert Example

### Deleting Items?

- can't just find item and remove it from its slot (i.e. set $T[h(k,i)] = \text{None}$)
- *example*: delete(586) $\implies$ search(496) fails
- replace item with special flag: "DeleteMe", which Insert treats as None but Search doesn't

## Probing Strategies

### Linear Probing

$h(k,i) = (h'(k) + i) \bmod m$ where $h'(k)$ is ordinary hash function

- like street parking
- **problem?** *clustering*—cluster: consecutive group of occupied slots
  as clusters become longer, it gets *more* likely to grow further (see Fig. 4)
- can be shown that for $0.01 < \alpha < 0.99$ say, clusters of size $\Theta(\log n)$.

### Double Hashing

$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m$ where $h_1(k)$ and $h_2(k)$ are two ordinary hash functions.

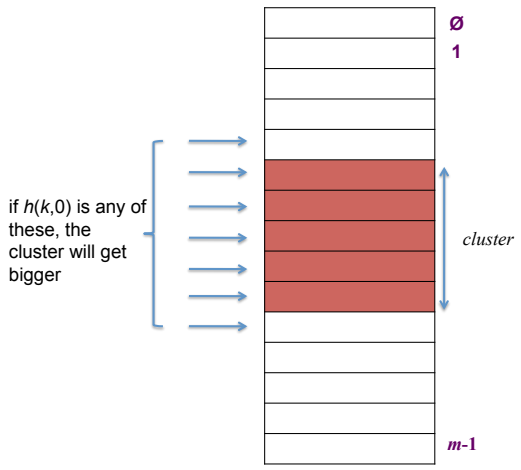


Figure 4: Primary Clustering

- actually hit all slots (permutation) if $h_2(k)$ is relatively prime to $m$ for all $k$ **why?**

$$h_1(k) + i \cdot h_2(k) \bmod m = h_1(k) + j \cdot h_2(k) \bmod m \Rightarrow m \textit{ divides } (i - j)$$

- e.g. $m = 2^r$, make $h_2(k)$ always odd

## Uniform Hashing Assumption (cf. Simple Uniform Hashing Assumption)

Each key is equally likely to have any one of the $m!$ permutations as its probe sequence

- not really true
- but double hashing can come close

## Analysis

Suppose we have used open addressing to insert $n$ items into table of size $m$. Under the uniform hashing assumption the next operation has expected cost of $\leq \frac{1}{1-\alpha}$, where $\alpha = n/m (< 1)$.
**Example**: $\alpha = 90\% \implies$ 10 expected probes

**Proof:**

Suppose we want to insert an item with key $k$. Suppose that the item is not in the table.

- probability first probe successful: $\frac{m-n}{m} =: p$
  ($n$ bad slots, $m$ total slots, and first probe is uniformly random)
- if first probe fails, probability second probe successful: $\frac{m-n}{m-1} \geq \frac{m-n}{m} = p$
  (one bad slot already found, $m-n$ good slots remain and the second probe is uniformly random over the $m-1$ total slots left)
- if 1st & 2nd probe fail, probability 3rd probe successful: $\frac{m-n}{m-2} \geq \frac{m-n}{m} = p$
  (since two bad slots already found, $m-n$ good slots remain and the third probe is uniformly random over the $m-2$ total slots left)
- ...

$\Rightarrow$ Every trial, success with probability at least $p$.
**Expected Number of trials for success?**

$$\frac{1}{p} = \frac{1}{1-\alpha}.$$

With a little thought it follows that search, delete take time $O(1/(1-\alpha))$. Ditto if we attempt to insert an item that is already there.■

## Open Addressing vs. Chaining

Open Addressing: better cache performance (better memory usage, no pointers needed)

Chaining: less sensitive to hash functions (OA requires extra care to avoid clustering) and the load factor $\alpha$ (OA degrades past 70% or so and in any event cannot support values larger than 1)

## Cryptographic Hashing

A cryptographic hash function is a deterministic procedure that takes an arbitrary block of data and returns a fixed-size bit string, the (cryptographic) hash value, such that an accidental or intentional change to the data will change the hash value. The data to be encoded is often called the *message*, and the hash value is sometimes called the *message digest* or simply digest.

The ideal cryptographic hash function has the properties listed below. $d$ is the number of bits in the output of the hash function. You can think of $m$ as being $2^d$. $d$ is typically 160 or more. These hash functions can be used to index hash tables, but they are typically used in computer security applications.

## Desirable Properties

1. **One-Way (OW)**: Infeasible, given $y \in_R \{0,1\}^d$ to find any $x$ s.t. $h(x) = y$. This means that if you choose a random $d$-bit vector, it is hard to find an input to the hash that produces that vector. This involves "inverting" the hash function.

2. **Collision-resistance (CR)**: Infeasible to find $x$, $x'$, s.t. $x \neq x'$ and $h(x) = h(x')$. This is a collision, two input values have the same hash.

3. **Target collision-resistance (TCR)**: Infeasible given $x$ to find $x' = x$ s.t. $h(x) = h(x')$.

TCR is weaker than CR. If a hash function satisfies CR, it automatically satisfies TCR. There is no implication relationship between OW and CR/TCR.

## Applications

1. **Password storage**: Store $h(PW)$, not $PW$ on computer. When user inputs $PW'$, compute $h(PW')$ and compare against $h(PW)$. The property required of the hash function is OW. The adversary does not know $PW$ or $PW'$ so TCR or CR is not really required. Of course, if many, many passwords have the same hash, it is a problem, but a small number of collisions doesn't really affect security.

2. **File modification detector**: For each file $F$, store $h(F)$ securely. Check if $F$ is modified by recomputing $h(F)$. The property that is required is TCR, since the adversary wins if he/she is able to modify $F$ without changing $h(F)$.

3. **Digital signatures**: In public-key cryptography, Alice has a public key $PK_A$ and a private key $SK_A$. Alice can sign a message $M$ using her private key to produce $\sigma = sign(SK_A, M)$. Anyone who knows Alice's public key $PK_A$ and verify Alice's signature by checking that $verify(M, \sigma, PK_A)$ is true. The adversary wants to forge a signature that verifies. For large $M$ it is easier to sign $h(M)$ rather than $M$, i.e., $\sigma = sign(SK_A, h(M))$. The property that we

require is CR. We don't want an adversary to ask Alice to sign $x$ and then claim that she signed $x'$, where $h(x) = h(x')$.

## Implementations

There have been many proposals for hash functions which are OW, CR and TCR. Some of these have been broken. MD-5, for example, has been shown to not be CR. There is a competition underway to determine SHA-3, which would be a Secure Hash Algorithm certified by NIST. Cryptographic hash functions are significantly more complex than those used in hash tables. You can think of a cryptographic hash as running a regular hash function many, many times with pseudo-random permutations interspersed.