Universal Hashing

- A malicious adversary who has learned the hash function chooses keys that all map to the same slot, giving worst-case behavior.
- Defeat the adversary using Universal Hashing
  - Use a different random hash function each time.
  - Ensure that the random hash function is independent of the keys that are actually going to be stored.
  - Ensure that the random hash function is "good" by carefully designing a class of functions to choose from.
    - Design a **universal** class of functions.

Universal Set of Hash Functions

- A finite collection of hash functions ***H*** that map
  a universe ***U*** of keys into the range $\{0, 1, \ldots, m-1\}$
  is "***universal***" if, for each pair of distinct keys, $k, l \in U$,
  the number of hash functions $h \in H$
  for which $h(k)=h(l)$ is no more than $|H|/m$.

- The chance of a collision between two keys is the $1/m$ chance of choosing two slots randomly & independently.

- Universal hash functions give good hashing behavior.

Cost of Universal Hashing

Theorem:

Using chaining and universal hashing on key *k*:

- If *k* is not in the table T, the expected length of the list that *k* hashes to is $\leq \alpha$.
- If *k* is in the table T, the expected length of the list that *k* hashes to is $\leq 1+\alpha$.

Proof:

$X_{kl} = I\{h(k)=h(l)\}$. $E[X_{kl}] = Pr\{h(k)=h(l)\} \leq 1/m$.

RV Y_k = *no. of keys other than k that hash to the same slot as k. Then,*

$$Y_k = \sum_{l \in T \wedge l \neq k} X_{kl}, \text{ and } E[Y_k] = \mathrm{E}\left[\sum_{l \in T \wedge l \neq k} X_{kl}\right] = \sum_{l \in T \wedge l \neq k} E[X_{kl}] \leq \sum_{l \in T \wedge l \neq k} \frac{1}{m}$$

$$\text{If } k \notin T, \text{exp. length of list} = E[Y_k] \leq n/m = \alpha.$$

$$\text{If } k \in T, \text{exp. length of list} = E[Y_k] + 1 \leq (n-1)/m + 1 = 1 + \alpha - 1/m < 1 + \alpha.$$

Example of Universal Hashing

When the table size m is a prime,

key x is decomposed into bytes s.t. $x = <x_0, \ldots, x_r>$,

and $a = <a_0, \ldots, a_r>$ denotes a sequence of $r+1$

elements randomly chosen from $\{0, 1, \ldots, m-1\}$,

The class H defined by

$$H = \bigcup_a \{h_a\} \text{ with } h_a(x) = \sum_{i=0 \text{ to } r} a_i x_i \bmod m$$

is a universal function,

(but if some a_i is zero, h does not depend on all bytes of x and if all a_i are zero the behavior is terrible. See text for better method of universal hashing.)

Analysis on Chained-Hash-Search

- Load factor $\alpha = n/m$ = average keys per slot.
  - m – number of slots.
  - n – number of elements stored in the hash table.
- Worst-case complexity: $\Theta(n)$ + time to compute $h(k)$.

- Average depends on how h distributes keys among m slots.

- **Assume**
  - *Simple uniform hashing.*
    - o Any key is equally likely to hash into any of the m slots, independent of where any other key hashes to.
  - $O(1)$ time to compute $h(k)$.
- Time to search for an element with key k is $\Theta(|T[h(k)]|)$.
- Expected length of a linked list = load factor = $\alpha = n/m$.

Expected Cost of an Unsuccessful Search

Theorem 11.1:
An unsuccessful search takes expected time $\Theta(1+\alpha)$.

Proof:

- Any key not already in the table is equally likely to hash to any of the m slots.
- To search unsuccessfully for any key k, need to search to the end of the list $T[h(k)]$, whose expected length is α.
- Adding the time to compute the hash function, the total time required is $\Theta(1+\alpha)$.

Expected Cost of a Successful Search

Theorem 11.2:
A successful search takes expected time Θ(1+α) under simple Uniform hashing.

Proof:

- The probability that a list is searched is proportional to the number of elements it contains.
- Assume that the element being searched for is equally likely to be any of the n elements in the table.
- The number of elements examined during a successful search for an element x is 1 more than the number of elements that appear before x in x's list.
  - These are the elements inserted ***after x*** was inserted.
- Goal:
  - Find the average, over the n elements x in the table, of how many elements were inserted into x's list after x was inserted.

Expected Cost of a Successful Search

Theorem 11.2:
A successful search takes expected time Θ(1+α) under simple Uniform hashing.

Proof (contd):

- Let x_i be the i^{th} element inserted into the table, and let $k_i = key[x_i]$.

- Define indicator random variables $X_{ij} = I\{h(k_i) = h(k_j)\}$, for all i, j.
- Simple uniform hashing $\Rightarrow \Pr\{h(k_i) = h(k_j)\} = 1/m$

  $\Rightarrow E[X_{ij}] = 1/m$. (Lemma 5.1)
- Expected number of elements examined in a successful search is:

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]$$

No. of elements inserted after x_i into the same slot as x_i.

Proof – Contd.

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]$$

$$=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right)$$

(linearity of expectation)

$$=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)$$

$$=1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$$=1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right)$$

$$=1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right)$$

$$=1+\frac{n-1}{2m}$$

$$=1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

Expected total time for a successful search = *Time to compute hash function + Time to search*
= O(2+α/2 – α/2n) = O(1+ α).

Open Addressing

- An alternative to chaining for handling collisions.
- **Idea:**
  - Store all keys in the hash table itself. What can you say about α?
  - Each slot contains either a key or NIL.
  - To ***search*** for key k:
    - Examine slot $h(k)$. Examining a slot is known as a **probe**.
    - If slot $h(k)$ contains key k, the search is successful. If the slot contains NIL, the search is unsuccessful.
    - There's a third possibility: **slot $h(k)$ contains a key that is not k.**
      - Compute the index of some other slot, based on k and which probe we are on.
      - Keep probing until we either find key k or we find a slot holding NIL.
- **Advantages:** Avoids pointers; so can use a larger table.

Probe Sequence

- Sequence of slots examined during a key search constitutes a ***probe sequence***.
- Probe sequence must be a permutation of the slot numbers.

  - We examine every slot in the table, if we have to.

  - We don't examine any slot more than once.
- The hash function is extended to:

  - $h : U \times \underbrace{\{0, 1, \ldots, m-1\}}_{\text{probe number}} \rightarrow \underbrace{\{0, 1, \ldots, m-1\}}_{\text{slot number}}$
- $\langle h(k,0), h(k,1),\ldots,h(k,m-1)\rangle$ should be a permutation of $\langle 0, 1,\ldots, m-1\rangle$.

Ex: Linear Probing

- Example:
  - $h(x) = x \bmod 13$
  - $h(x,i) = (h(x) + i) \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Operation Insert

- Act as though we were searching, and insert at the first NIL slot found.
- Pseudo-code for Insert:


```
Hash-Insert(T, k)
i ← 0
repeat j ← h(k, i)
        if T[j] = NIL
           then T[j] ← k
                return j
           else i ← i + 1
until i = m
error "hash table overflow"
```

Pseudocode for Search

<u>Hash-Search (*T*, *k*)</u>

```
i ← 0
repeat  j ← h(k, i)
            if T[j] = k
                then return j
            i ← i + 1
until T[j] = NIL or i = m
return NIL
```

Deletion

- Cannot just turn the slot containing the key we want to delete to contain NIL. Why?
  - We might be unable to retrieve any key *k* during whose insertion we had probed slot *i* and found it occupied.

- Use a special value DELETED instead of NIL when marking a slot as empty during deletion.

  - ***Search*** should treat DELETED as though the slot holds a key that does not match the one being searched for.
  - ***Insert*** should treat DELETED as though the slot were empty, so that it can be reused. (So, the Hash-Insert need to be modified.)
- **Disadvantage:** Search time is no longer dependent on α.
  - Hence, chaining is more common when keys have to be deleted.

Computing Probe Sequences

- The ideal situation is ***uniform hashing***:
  - Generalization of simple uniform hashing.
  - Each key is equally likely to have any of the $\boldsymbol{m!}$ permutations of $\langle 0, 1, \ldots, m-1 \rangle$ as its probe sequence.

- It is hard to implement true uniform hashing.
  - Approximate with techniques that at least guarantee that the probe sequence is a permutation of $\langle 0, 1, \ldots, m-1 \rangle$.

- Some techniques:
  - Use ***auxiliary hash functions***.
    - Linear Probing.
    - Quadratic Probing.
    - Double Hashing.
  - Can't produce all $m!$ probe sequences. (None of these can fulfill the assumption of uniform hashing.)

Linear Probing

- $h(k, i) = (h'(k)+i) \bmod m.$

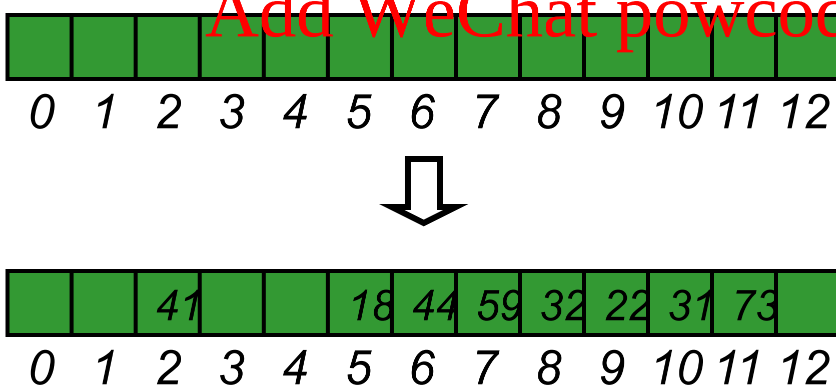
  key *Probe number* *Auxiliary hash function*

- The initial probe determines the entire probe sequence.
  - $T[h'(k)], T[h'(k)+1], \ldots, T[m-1], T[0], T[1], \ldots, T[h'(k)-1]$
  - Hence, only m distinct probe sequences are possible.
- Easy to implement, but suffers from ***primary clustering***:
  - Long runs of occupied sequences build up.
  - Long runs tend to get longer, since an empty slot preceded by i full slots gets filled next with probability $(i+1)/m$.
  - Hence, average search and insertion times increase.

Ex: Linear Probing

- Example:
  - $h'(x) = x \bmod 13$
  - $h(x)=(h'(x)+i) \bmod 13$

  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Quadratic Probing

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

  key *Probe number* *Auxiliary hash function*

- The initial probe position is $T[h'(k)]$, later probe positions are offset by amounts that depend on a quadratic function of the probe number i.

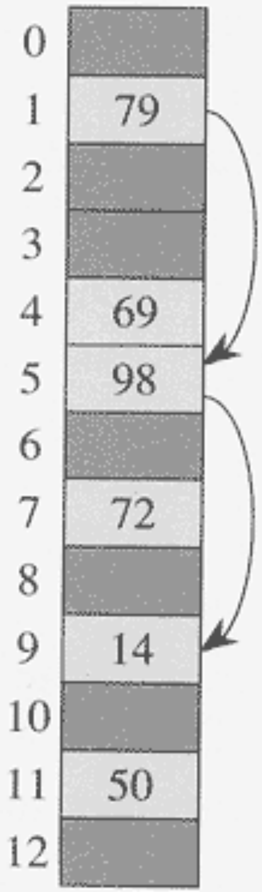
- Must constrain c_1, c_2, and m to ensure that we get a full permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
- Can suffer from ***secondary clustering***:
  - If two keys have the same initial probe position, then their probe sequences are the same. $h(k_1,0) = h(k_2,0)$

Double Hashing

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

key Probe number Auxiliary hash functions

- Two auxiliary hash functions.
  - h_1 gives the initial probe. h_2 gives the remaining probes.
- Must have $h_2(k)$ relatively prime to m, so that the probe sequence is a full permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
  - Choose m to be a power of 2 and have $h_2(k)$ always return an odd number. Or,
  - Let m be prime, and have $1 < h_2(k) < m$.
- $\Theta(m^2)$ different probe sequences.
  - One for each possible combination of $h_1(k)$ and $h_2(k)$.
  - Close to the ideal uniform hashing.



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

$m = 13$
$h_1(k) = k \bmod 13$
$h_2(k) = 1 + (k \bmod 11)$

$14 \equiv 1 \pmod{13}$
$14 \equiv 3 \pmod{11}$

Hashing with open addressing tool

Figure 11.5 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 $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, the key 14 is inserted into empty slot 9, after slots 1 and 5 are examined and found to be occupied.

Analysis of Open-address Hashing

- Analysis is in terms of load factor $\alpha = n/m$.
- **Assumptions:**
  - Assume that the table never completely fills, so $n < m$ and $\alpha < 1$.
  - Assume uniform hashing.
  - No deletion.
  - In a successful search, each key is equally likely to be searched for.

Expected cost of an unsuccessful

Theorem:
Given an open-address hash table with $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search in an open-address hash table is at most $1/(1-\alpha)$ assuming uniform hashing.

Proof:

Every probe except the last is to an occupied slot.

Let RV X = # of probes in an unsuccessful search.

$X \geq i$ iff probes 1, 2, …, $i-1$ are made to occupied slots

Let A_i = event that there is an ith probe, to an occupied slot.

$$\Pr\{X \geq i\}$$
$$= \Pr\{A_1 \cap A_2 \cap \ldots \cap A_{i-1}\}.$$
$$= \Pr\{A_1\}\Pr\{A_2 | A_1\} \Pr\{A_3 | A_2 \cap A_1\} \ldots \Pr\{A_{i-1} | A_1 \cap \ldots \cap A_{i-2}\}$$

Proof – Contd.

$X \geq i$ iff probes 1, 2, …, $i - 1$ are made to occupied slots

Let A_i = event that there is an ith probe, to an occupied slot.

$\Pr\{X \geq i\}$

$= \Pr\{A_1 \cap A_2 \cap \dots \cap A_{i-1}\}$

$= \Pr\{A_1\}\Pr\{A_2 | A_1\} \Pr\{A_3 | A_2 \cap A_1\} \dots \Pr\{A_{i-1} | A_1 \cap \dots \cap A_{i-2}\}$

- $\Pr\{A_j \mid A_1 \cap A_2 \cap \dots \cap A_{j-1}\} = (n-j+1)/(m-j+1).$

$$\Pr\{X \geq i\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2}$$

$$\leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}.$$

Proof – Contd.

$$
\begin{aligned}
E[X] &= \sum_{i=0}^{\infty} i \Pr\{X = i\} \\
&= \sum_{i=0}^{\infty} i(\Pr\{X \geq i\} - \Pr\{X \geq i+1\}) \\
&= 1 \cdot \Pr\{X \geq 1\} - 1 \cdot \Pr\{X \geq 2\} + 2 \cdot \Pr\{X \geq 2\} - 2 \cdot \Pr\{X \geq 3\} + \cdots \\
&= 1 \cdot \Pr\{X \geq 1\} + \Pr\{X \geq 2\} + \Pr\{X \geq 3\} + \cdots \qquad \textit{(C.25)} \\
&= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \\
&\leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha} \qquad \textit{(A.6)}
\end{aligned}
$$

- If α is a constant, search takes $O(1)$ time.
- Corollary: Inserting an element into an open-address table takes $\leq 1/(1-\alpha)$ probes on average.

Expected cost of a successful

> ***Theorem:***
> *The expected number of probes in a successful search in an open-address hash table is at most $(1/\alpha) \ln (1/(1-\alpha))$.*

Proof:

- A successful search for a key k follows the same probe sequence as when k was inserted.
- If k was the $(i+1)$st key inserted, then α equaled i/m at that time.
- By the previous corollary, the expected number of probes made in a search for k is at most $1/(1-i/m) = m/(m-i)$.
- This is assuming that k is the $(i+1)$st key. We need to average over all n keys.

Proof – Contd.

Averaging over all n keys, average # of probes is given by

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i}$$

$$= \frac{1}{\alpha}(H_m - H_{m-n})$$

$$\leq \frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

Perfect Hashing

- If you know the n keys in advance, make a hash table with $O(n)$ size, and worst-case $O(1)$ lookup time!

- Start with $O(n^2)$ size… no collisions

Thm 11.9: For a table of size $m = n^2$,

if we choose h from a universal class of hash functions, we have no collisions with probability $>½$.

Pf: Expected number of collisions among pairs:

$E[X] = (n \textit{ choose } 2) / n^2 < ½$,

& Markov inequality says $\Pr\{X \geq t\} \leq E[X]/t$. ($t=1$)

Perfect Hashing

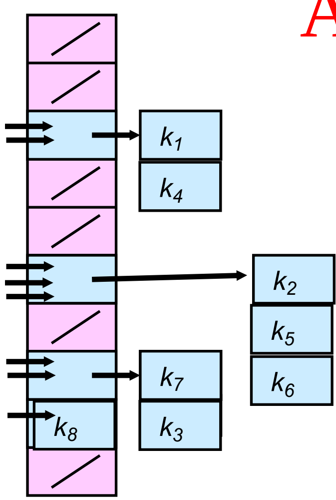
- If you know the n keys in advance, make a hash table with $O(n)$ size, and worst-case $O(1)$ lookup time!
- With table size n, few $(\text{collisions})^2$…

Thm 11.10: For a table of size $m = n$,
if we choose h from a universal class of hash functions,
$E[\sum_j n_j^2] < 2n$, where n_j is number of keys hashing to j.

Pf: essentially the total number of collisions.

Perfect Hashing

- If you know the n keys in advance (static), make a hash table with $O(n)$ size, and worst-case $O(1)$ lookup time!
- Just use two levels of hashing: A table of size n, then tables of size n_j^2.



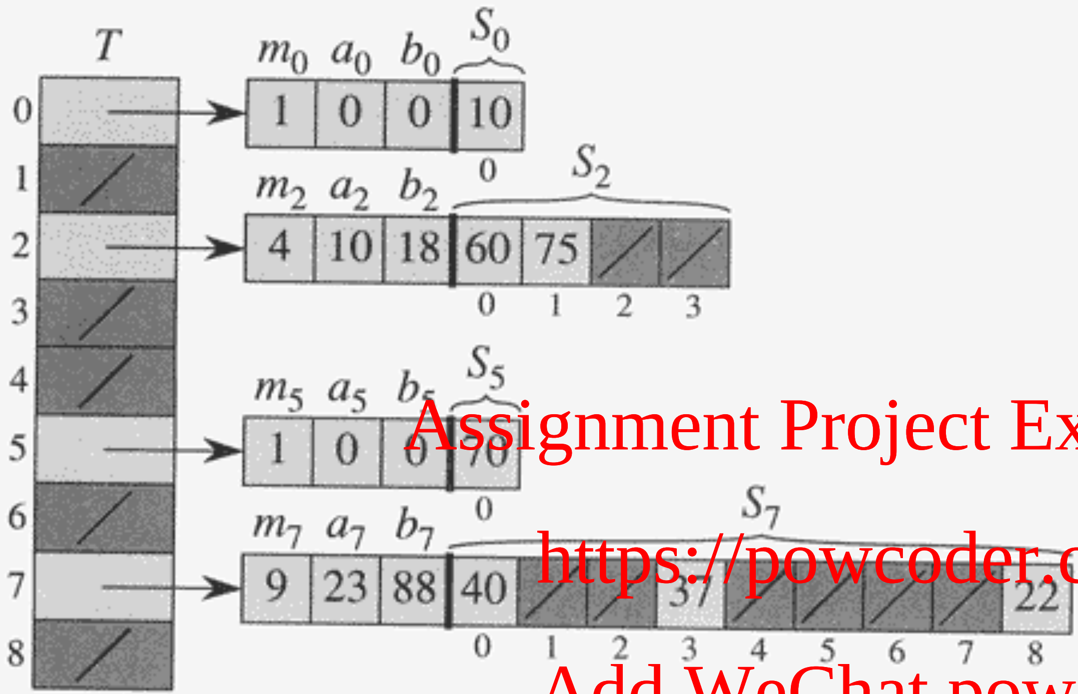


Figure 11.6 Using perfect hashing to store the set $K = \{10, 22, 37, 40, 60, 70, 75\}$. The outer hash function is $h(k) = ((ak + b) \bmod p) \bmod m$, where $a = 3$, $b = 42$, $p = 101$, and $m = 9$. For example, $h(75) = 2$, so key 75 hashes to slot 2 of table T. A secondary hash table S_j stores all keys hashing to slot j. The size of hash table S_j is m_j, and the associated hash function is $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$. Since $h_2(75) = 1$, key 75 is stored in slot 1 of secondary hash table S_2. There are no collisions in any of the secondary hash tables, and so searching takes constant time in the worst case.

End of Chapter 11