# Hash Tables

## Dictionary

- **Dictionary:**
  Dynamic-set data structure for storing items indexed using *keys*.
  Supports operations Insert, Search, and Delete.
  Applications:
  - Symbol table of a compiler.
  - Memory-management tables in operating systems.

- **Hash Tables:**
  Effective way of implementing dictionaries.
  Generalization of ordinary arrays.

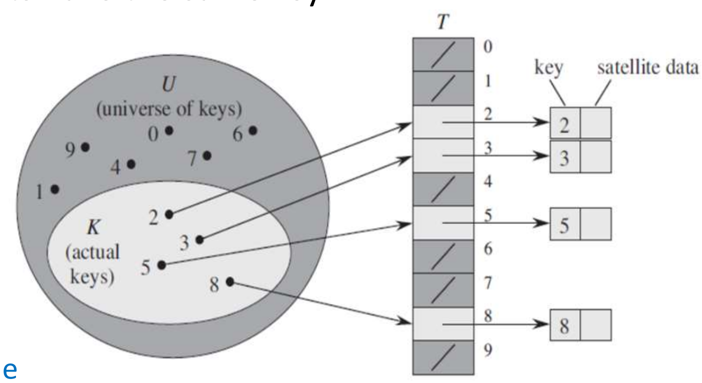
# Direct-address Tables

- Keys are drawn from the universe $U = \{1,2, \dots, m-1\}$, where $m$ is not too large.
- Assumption: No two elements have the same key.

DIRECT-ADDRESS-SEARCH$(T, k)$
    **return** $T[k]$

DIRECT-ADDRESS-INSERT$(T, x)$
    $T[x.key] = x$

DIRECT-ADDRESS-DELETE$(T, x)$
    $T[x.key] =$ NIL




- Dictionary operations take $O(1)$ time

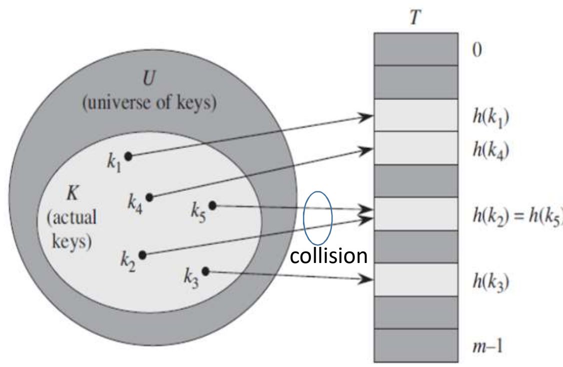
$U = \{1,2, \dots, 9\}$   $K = \{2,3,5,8\}$

# Hash Tables

- **Notation:**
  $U$ – Universe of all possible keys.
  $K$ – Set of keys actually stored in the dictionary.
  $|\ K\ | = n$.
- When $U$ is very large,
  Arrays are not practical.
  $|\ K\ | << |\ U\ |$.
- Use a table of size proportional to $|\ K\ |$ – The hash tables.
  However, we lose the direct-addressing ability.
  Define functions that map keys to slots of the hash table.

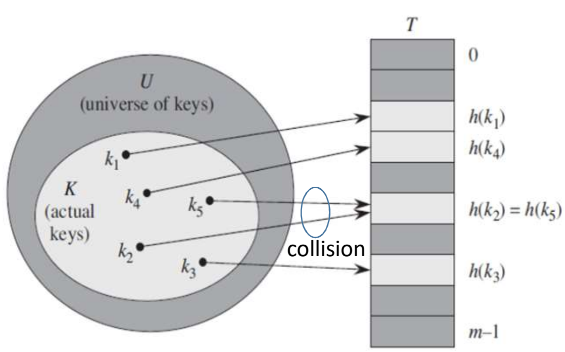
# Hashing

- Hash function $h$ : Mapping from $U$ to the slots of a hash table $T[0 \dots m-1]$
  $h: U \to \{0,1,\dots,m-1\}$
- With direct-address tables, key $x$ maps to slot $T[x]$.
- With hash tables, key $x$ maps or "hashes" to slot $T[h(x)]$.
- $h(x)$ is the *hash value* of key $x$.
- **Example:** Map a key $x$ into one of the $m$ slots by taking the remainder of $x$ divided by $m$. That is,
  $h(x) = x \bmod m$




# Issues With Hashing

- Multiple keys can hash to the same slot – collisions are possible.
  - Design hash functions such that collisions are minimized.
  - But avoiding collisions is impossible.
    - Design collision-resolution techniques.
- Search will cost $\Theta(n)$ time in the worst case.
  - However, all operations can be made to have an expected complexity of $\Theta(1)$



# Hash Function

**Desirata:**

- A good hash function should distribute the keys uniformly into the slots of the table.
- Regularity in the key distribution should not affect this uniformity.

**The standard convention for hash functions is to view keys in one of two ways**

- Each key is a tuple of integers, $x = \langle x_0, x_1, \cdots, x_d \rangle$ with each $x_i$ being an integer in the range $[0, m-1]$, for some $m$ and $d$
- Each key, $x$, is a nonnegative integer, which could possibly be very large

# Hash Functions

**Summing Components:**

Keys, $x$, is a $d$-tuple, of the form $x = \langle x_0, x_1, \cdots, x_d \rangle$, where each $x_i$ is an integer

$$h(x) = \sum_{i=1}^{d} x_i \bmod p$$

**Variation:** $h(x) = \oplus_{i=1}^{d} x_i$

**Problem:** Symmetry Can Cause Collisions e.g., “stop”, “tops”, “pots”, and “spot” collide

**Polynomial Evaluation Function:**

Keys, $x$, is a $d$-tuple, of the form $x = \langle x_0, x_1, \cdots, x_d \rangle$, where each $x_i$ is an integer, choose $a \neq 1$ and compute $h(x) = x_1 a^{d-1} + x_2 a^{d-2} + \cdots + x_{d-1} a^1 + x_d$

**How to compute $h(x)$ efficiently?** $h(x) = x_d + a(x_{d-1} + a(x_{d-2} + \cdots + a(x_3 + a(x_2 + a x_1)) \ldots))$

**Note:** we can evaluate such a hash function in a simple for-loop, which has *d* – 1 iterations; hence, this function can be evaluated using *d* – 1 additions and *d* – 1 multiplications.

**Note:** $h(x) = \sum_{i=1}^{d} x_i p^{d-i} mod\ m, p$ is not a multiple of $m$ (better to choose $p$ as a prime)

# Hash Functions

**Tabulation-based Hashing:**

- Each key is a tuple of integers, $x = \langle x_1, x_2, \cdots, x_d \rangle$ with each $x_i$ being an integer in the range $[0, m-1]$
- Initialize $d$ tables $T_1, T_2, \cdots, T_d$, of size $m$ each, so that each $T_i(j)$ is uniformly chosen independent random number in the range $[0, n-1]$ ($n$ is the size of hash table and $n \le m^d$)
- $h(x) = T_1[x_1] \oplus T_2[x_2] \oplus \cdots \oplus T_d[x_d]$

**Note:** it can be shown that such a function will cause two distinct keys to collide at the same hash value with probability $1/n$, which is what we would get from a perfectly random function

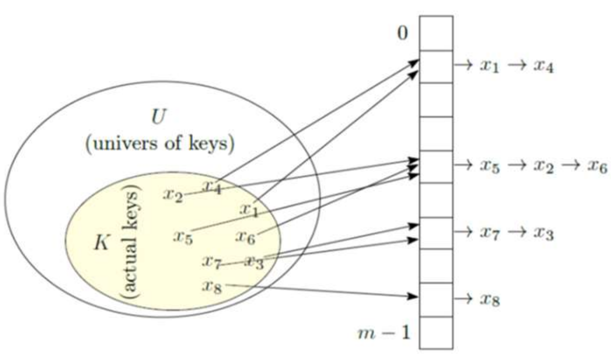
**Modular Division:**

For general integer keys and a table of size $p$, a prime number: a good fast general purpose hash function is $h(x) = x \; mod \; p$

**Random Linear Hash Function:**

For an integer $x$, $h(x) = (ax + b) \; mod \; p$, where $p$ is a prime number, and $0 < a < p$ and $0 \le b < p$ are independent uniform random integers.

# Hashing With Chaining




**Dictionary Operations:**

- **Chained-Hash-Insert ($T, x$)**

  Insert $x$ at the head of list $T[h(x)]$, assuming $x \notin T$

  Worst-case complexity -- O(1)

- **Chained-Hash-Delete ($T, x$)**

  Delete $x$ from the list $T[h(x)]$, assuming $x \in T$

  Worst-case complexity – proportional to length of list

- **Chained-Hash-Search ($T, x$)**

  Search $x$ in $T[h(x)]$

  Worst-case complexity – proportional to length of list

# Analysis of Chained-Hash-Search

- **Load factor** $\alpha = \frac{n}{m} =$ 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 $h(x)$
- **Average-case complexity:** depends on how $h$ distributes keys among $m$ slots

**Assumptions:**

- Simple uniform hashing: any key is equally likely to hash any of the $m$ slots, independent of where any other key hashes to i.e., $P\{h(x_i) = h(x_j)\} = 1/m$
- O(1) time to compute $h(x)$

- Time to search for an element with key $x$ is $\Theta(|T[h(x)]|)$
- Expected length of a linked list $=$ load factor $=\ \alpha = \frac{n}{m}$

# Expected Cost of an Unsuccessful Search

**Theorem:** 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 (simple uniform hashing).
- To search unsuccessfully for any key $x$, need to search to the end of the list $T[h(x)]$, whose expected length is $\alpha$.
- Adding the time to compute the hash function, the total time required is $\Theta(\underbrace{1}_{\text{Time to compute hash function}} + \underbrace{\alpha}_{\text{Search the list}})$

# Expected Cost of an Successful Search

**Theorem:** A successful search takes expected time $\Theta(1+\alpha)$

**Proof:**

- Assumption: each element is equally likely to be searched.
- Let $x_i$ be the $i$-th element inserted to the table. Note that, new elements are placed at the front of the list $T[h(x_i)]$
- Therefore, elements before $x_i$ in $T[h(x_i)]$ (a subset of $\{x_{i+1}, x_{i+2}, \dots, x_n\}$) were all inserted after $x_i$ was inserted
- **Expected number of elements inserted before $x_i$ in $T[h(x_i)]$**
  - Under the assumption of simple uniform hashing $P\{h(x_i) = h(x_j)\} = 1/m$ (for $j = i+1, i+2, \dots, n$)
  - Average number of elements inserted before $x_i$ in $T[h(x_i)]$ is $(n-i)/m$
- As each element is equally likely to be searched, the average cost of an successful search is $\frac{1}{n}\sum_{i=1}^{n}(1+\frac{n-i}{m})$ (1 is added as we need to search $x_i$ also)

# Expected Cost of an Successful Search

**Theorem:** A successful search takes expected time $\Theta(1+\alpha)$

**Proof(continued):**

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

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

Thus, the total time required for a successful search (including the time for computing the hash function) is $\Theta(1+1+\frac{\alpha}{2}-\frac{1}{2m}) = \Theta(1+\alpha)$

- Expected search time = $\Theta(1)$ if $\alpha = O(1)$, or equivalently, if $n = O(m)$

# Resolving Collisions by Open Addressing

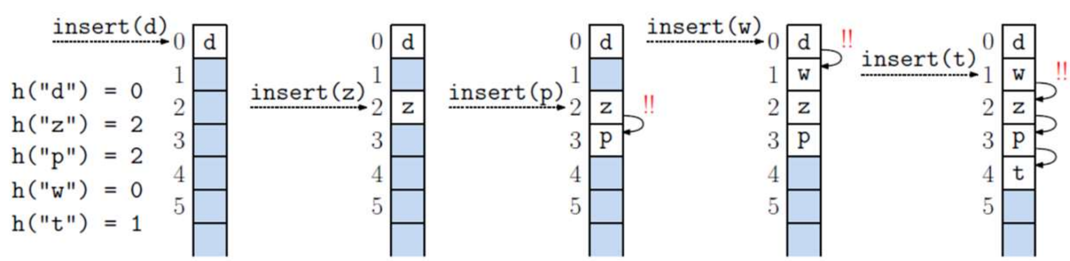
No storage is used outside of the hash table itself. The hash table stores a special value NIL in the empty table entries.

- Insertion systematically probes the table until an empty slot is found.
- The load factor $\alpha$ can never exceed 1 i.e., $\alpha \leq 1$.
- The hash function depends on both the key and probe number:

  $$h: U \times \{0,1,\ldots,m-1\} \rightarrow \{0,1,\ldots,m-1\}$$
- The probe sequence $\langle h(x,0), h(x,1), \ldots, h(x,m-1)\rangle$ should be a permutation of $\{0,1,\ldots,m-1\}$

# Probing Strategies: Linear Probing

Given an ordinary hash function $h(x)$, linear probing uses the hash function $H(x,i) = (h(x) + i)\ mod\ m$




- Given a key $x$, the probe sequence is $(h(x), h(x)+1, \cdots, 0, 1, \cdots, h(x)-1)$. As the initial probe determines the entire probe sequence, there are only $m$ distinct probe sequences.

- Primary clustering happens when multiple keys hash to the same location (ex: (d,w), (z,p)).
- Secondary clustering happens when keys hash to different locations, but the collision-resolution has resulted in new collisions (ex: t).

# Performance of Linear Probing

Expected cost for successful search $= \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$

Expected cost for unsuccessful search $= \frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right)$

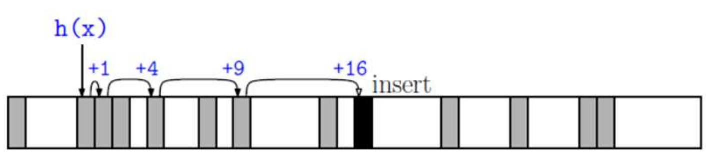
- In open addressing, secondary clustering is a significant phenomenon. As the $\alpha \rightarrow 1$, probe sequences may become unacceptably long due to secondary clustering.
- As long as the table remains less than 75% full, linear probing performs fairly well.

# Probing Strategies: Quadratic Probing

Given an ordinary hash function $h(x)$, quadratic probing uses the hash function

$$H(x, i) = (h(x) + i^2) \bmod m$$

- Given a key $x$, the probe sequence is
  $$(h(x) mod\ m, (h(x) + 1) mod\ m, (h(x) + 2^2) mod\ m, \cdots, (h(x) + (m-1)^2) mod\ m)$$
- As the initial probe determines the entire probe sequence, there are only $m$ distinct probe sequences.




Shaded squares are occupied and the black square indicates where the key is inserted

- Quadratic probing might miss some slots in the table. For example, consider $m = 4$. Suppose $h(x) = 0$ and T[1] and T[3] has empty slot. The quadratic probe sequence will inspect the following indices:
  $1^2$ mod 4 = 1; $2^2$ mod 4 = 0; $3^2$ mod 4 = 1; $4^2$ mod 4 = 0 ...
  Note: We can't find an empty place although table is half full!!
  Try with $m = 105$ for more realistic example.

# Probing Strategies: Quadratic Probing

**Theorem:** If quadratic probing is used, and the table size $p$ is prime, the first $\left\lfloor \frac{p}{2} \right\rfloor$ probe sequences are distinct

**Proof:** For contradiction, suppose for $0 \leq i < j \leq \left\lfloor \frac{p}{2} \right\rfloor$, both $(h(x) + i^2)$ and $(h(x) + j^2)$ are equivalent modulo $m$ i.e., probe- $i$ and probe-$j$ look into same slot.

$$i^2 \equiv j^2 \Leftrightarrow i^2 - j^2 \equiv 0 \Leftrightarrow (i-j)(i+j) \equiv 0 \ (mod\ p)$$

This implies, $(i-j)(i+j)$ is a multiple of $p$. But this can't be, since $p$ is prime and both $i-j$ and $i+j$ are nonzero and strictly smaller than $p$ ($i < j \leq \left\lfloor \frac{m}{2} \right\rfloor$). Thus we have the desired contradiction.

**Implication:**
- If we choose table size $p$ to be a prime number, then quadratic probing is guaranteed to visit at least half of the table entries before repeating.
- This means that it will succeed in finding an empty slot, provided that $p$ is prime and load factor is smaller than 1/2.

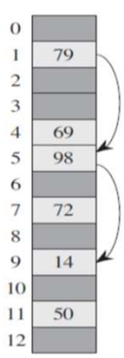
# Probing Strategies: Double Hashing

Given two ordinary hash functions $h_1(x)$ and $h_2(x)$, double hashing uses the hash function

$$H(x,i) = (h_1(x)+i.h_2(x))mod\ m$$

**Requirement:** $h_2(x)$ must be relatively prime to $m$.

- One way is to make $m$ a power of 2 and design $h_2(x)$ to produce only odd numbers.
- Another way is to let $m$ be prime and to design $h_2$ so that it always returns a positive integer less than $m$.




$m = 13$
$h_1(x) = x\ mod\ 13$
$h_2(x) = 1 + (x\ mod\ 11)$

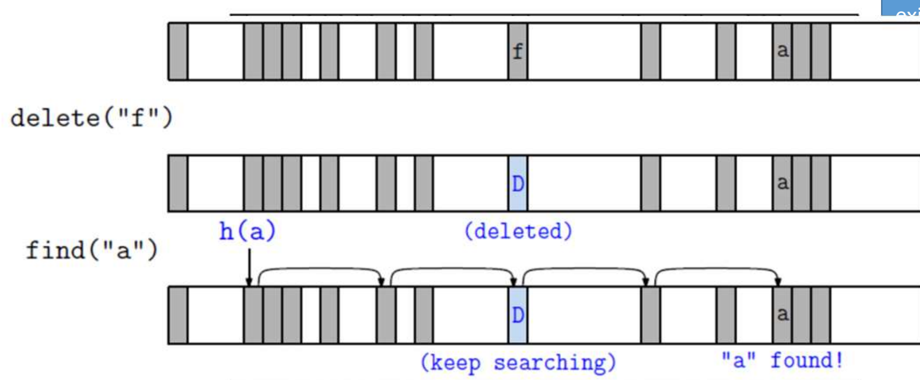
$x = 14$
$h_1(x) = x\ mod\ 13 = 1$
$h_2(x) = 1 + (x\ mod\ 11) = 4$
$H(x,i) = (1 + 4i)\ mod\ 13$

Probe sequence:
1, 5, 9, 0, 4, 8, 12, 3, …

- When $m$ is prime or a power of 2, double hashing has $\Theta(m^2)$ probe sequences, since each possible $(h_1(x), h_2(x))$, pair yields a distinct probe sequence.
  - Expected cost for successful search $= \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$
  - Expected cost for unsuccessful search $= \frac{1}{1-\alpha}$

# Deletion in Open Addressing

When we insert a, an existing key f lies on the probe path of a




When a is searched an empty slot is hit

Deleting in open-addressing by using special empty entry.

The problem with deletion in open addressing system

# Analysis of Open Addressing

**Assumption:**

- **Uniform hashing:** the probe sequence $\langle h(x,0), h(x,1), \cdots, h(x,m-1)\rangle$ of search/insert for each key $x$ is equally likely to be any permutation of $\langle 0,1,\ldots,m-1\rangle$

**Theorem:** Given an open-address hash table with load factor $\alpha = \frac{n}{m} < 1$, the expected number of probes in an unsuccessful search is at most $\frac{1}{1-\alpha}$, assuming uniform hashing.

**Proof:** In an unsuccessful search, every probe (except the last) accesses an occupied slot that does not contain the desired key, and the last slot probed is empty.

Define $p_i = P\{\text{exactly } i \text{ probes access occupied slots}\}$ for $i = 0, 1, 2, \ldots$
Note: for $i > n, p_i = 0$ (we can find at most $n$ occupied slots)

Expected number of probes is $1 + \sum_{i=0}^{\infty} ip_i$

Define $q_i = P\{\text{at least } i \text{ probes access occupied slots}\}$ for $i = 0, 1, 2, \ldots$

Note that, $1 + \sum_{i=0}^{\infty} ip_i = \sum_{i=1}^{\infty} q_i$

Suppose a random variable $X$ takes on values from the $N = \{0,1,2,\ldots\}$

$$E[X] = \sum_{i=0}^{\infty} iP\{X = i\}$$

$$= \sum_{i=0}^{\infty} i(P\{X \geq i\} - P\{X \geq i+1\})$$

$$= \sum_{i=1}^{\infty} P\{X \geq i\}$$

# Analysis of Open Addressing

**Theorem:** Given an open-address hash table with load factor $\alpha = \frac{n}{m} < 1$, the expected number of probes in an unsuccessful search is at most $\frac{1}{1-\alpha}$, assuming uniform hashing.

**Proof(contd.):** what is the value of $q_i$, for $i \geq 1$

The probability that the first probe accesses an occupied slot is $\frac{n}{m}$ thus, $q_1 = \frac{n}{m}$

With uniform hashing, a second probe (if necessary) is one of the remaining $m-1$ unprobed slots, $n-1$ are occupied. We make a second probe only if the first probe accesses an occupied slot. Thus $q_2 = \frac{n}{m}\frac{n-1}{m-1}$

In general, $i$-th probe is made only if the first $i-1$ probes access occupied slots.
The $i$-th slot probed is equally likely (assuming uniform hashing) to be any of the remaining $m-i+1$ unprobed slots, $n-i+1$ are occupied. Thus
$q_i = \frac{n}{m}\frac{n-1}{m-1}\cdots\frac{n-i+1}{m-i+1} \leq \left(\frac{n}{m}\right)^i = \alpha^i$ for $i = 1,2,\cdots,n$ (since, $\frac{n-j}{m-j} \leq \frac{n}{m}$ if $n \leq m$ and $j \geq 0$)

$q_i = 0, i > n$ (after $n$ probes, all $n$ occupied slots have been seen and will not be probed again)

$$1 + \sum_{i=0}^{\infty} i p_i = \sum_{i=1}^{\infty} q_i \leq 1 + \alpha + \alpha^2 + \cdots = \frac{1}{1-\alpha}$$

# Analysis of Open Addressing

**Theorem:** Given an open-address hash table with load factor $\alpha = \frac{n}{m} < 1$, the expected number of probes in an unsuccessful search is at most $\frac{1}{1-\alpha}$assuming uniform hashing.

**Corollary:** Inserting an element into an open-address hash table with load factor $\alpha$ requires at most $\frac{1}{1-\alpha}$ probes on average, assuming uniform hashing.

**Proof:**

- An element is inserted only if there is room in the table, and thus $\alpha < 1$.
- Inserting a key requires an unsuccessful search followed by placing the key into the first empty slot found.
- Thus, the expected number of probes is at most $\frac{1}{1-\alpha}$

# Analysis of Open Addressing

**Theorem:** Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$, assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

**Proof:**

- A search for a key $x$ reproduces the same probe sequence as when the element with key $x$ was inserted.
- if $x$ was the $(i+1)$-st key inserted into the hash table, the expected number of probes made in a search for $x$ is at most $1/\left(1-\frac{i}{m}\right) = \frac{m}{m-i}$
- Thus, the expected number of probes in successful search (avg. over all keys) is at most

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} = \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \le \frac{1}{\alpha}\int_{m-n}^{m}\frac{1}{x}dx = \frac{1}{\alpha}\ln\frac{m}{m-n} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

# Theory of Hashing

**Theorem:** For any hash function $h: U \to \{0,1,\cdots,m-1\}$, there exists a set $U$ of $n$ keys that all map to the same location, assuming $|U| > nm$

**Proof:**

- Take any hash function $h$ and map all the keys of $U$ using $h$ to the table of size $m$.
- By the pigeon-hole principle, at least one table entry will have $n$ keys.
- Choose those $n$ keys as input set $S$.
- Now $h$ will maps the entire set $S$ to a single location.

- The negative result says that given a fixed hash function $h$, one can always construct a set $S$ that is bad for $h$.
- However, what we desire is something different:
  - We are not choosing $S$; it is our (given) input.
  - Can we find a good $h$ for this particular $S$?
  - Theory shows that a random choice of $h$ works.

# Theory of Hashing: Universal Hash Function

- Let $H$ be a finite collection of hash functions that map a given universe $U$ of keys into the range $\{0,1,\cdots,m-1\}$. Such a collection is said to be ***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 $\frac{|H|}{m}$.
- A set of hash functions $H$ is called universal if for any hash function $h$ chosen randomly from it $P[h(\boldsymbol{x}) = h(\boldsymbol{y})] = \frac{1}{m}$, for any distinct $\boldsymbol{x}, \boldsymbol{y} \in U$.

**Theorem.** Suppose $H$ is universal, $S$ is an $n$-element subset of $U$, and $h$ a random hash function from $H$. The expected number of collisions is $(n-1)/m$ for any $x \in S$.

**Proof:**

- For each pair $x, y \in S$ of distinct keys let $c_{xy}$ be an indicator random variable i.e., $c_{xy}$ = 1 if $h(x) = h(y)$ and 0 otherwise.
- From the definition of universal hashing, $P[h(x) = h(y)] = \frac{1}{m}$. Hence E$[c_{xy}] = \frac{1}{m}$.
- Let $C_x$ be the total number of collision involving key $x$ in a hash table $T$ of size $m$ containing $n$ keys. $E[C_x] = \sum_{y \in T \text{ and } y \neq x} E\left[c_{xy}\right] = (n-1)/m$ (By linearity of expectation)

**Corollary:** By using a random hash function (from a universal family), we get expected search time $O(1 + \frac{n}{m})$.

# Constructing Universal Hash Function

- Table size is $m$ (prime)
- Decompose a key $x$ into $r+1$ digits, each with value in the set $\{0,1,\cdots,m-1\}$ i.e., $x = \langle x_0, x_1, \cdots, x_r \rangle$, where $x_i \in \{0,1,\cdots,m-1\}$

**Hash function:**

- Pick $a = \langle a_0, a_1, \cdots, a_r \rangle$, where $a_i$ is chosen randomly from $\{0,1,\cdots,m-1\}$
- Define $h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m$ (dot product modulo $m$)
- $H = \{h_a\}$
- $|H| = m^{r+1}$

# Constructing Universal Hash Function

**Theorem:** The set $H = \{h_a\}$ is universal

**Proof:**

- Suppose $x = \langle x_0, x_1, \cdots, x_r \rangle, y = \langle y_0, y_1, \cdots, y_r \rangle$ be distinct keys. Without loss of generality keys $x$ and $y$ differs in position 0 i.e., $x_0 \neq y_0$
- For how many $h_a \in H$, $x$ and $y$ collide?
- For collision, we must have $h_a(x) = h_a(y)$, which implies $\sum_{i=0}^{r} a_i x_i \equiv \sum_{i=0}^{r} a_i y_i \; mod \; m$
- Equivalently, we have
  $\sum_{i=0}^{r} a_i(x_i - y_i) \equiv 0 \; mod \; m$
  or $a_0(x_0 - y_0) + \sum_{i=1}^{r} a_i(x_i - y_i) \equiv 0 \; mod \; m$
  which implies that,
  $a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r} a_i(x_i - y_i) \; mod \; m$

# Constructing Universal Hash Function

**Theorem:** The set $H = \{h_a\}$ is universal

**Proof (Contd.):**

- Since $x_0 \neq y_0$, an inverse $(x_0 - y_0)^{-1}$ must exist, which implies

$$a_0 \equiv (-\sum_{i=1}^{r} a_i(x_i - y_i))(x_0 - y_0)^{-1} mod \; m$$

- Thus for any choices of $a_1, a_2, \cdots, a_r$, exactly one choice of $a_0$ causes $x$ and $y$ to collide
- How many $h_a \in H$ causes $x$ and $y$ to collide?
- There are $m$ choices for each of $a_1, a_2, \cdots, a_r$, but once these are chosen, exactly one choice of $a_0$ causes $x$ and $y$ to collide, namely $a_0 = (-\sum_{i=1}^{r} a_i(x_i - y_i))(x_0 - y_0)^{-1} mod \; m$
- Thus the number of $h_a$'s that causes $x$ and $y$ to collide is $m^r = \frac{|H|}{m}$

**Theorem**: Let $m$ be a prime. For any $z \in Z_m$ such that $z \neq 0$, there exist a unique $z^{-1} \in Z_m$ such that $zz^{-1} \equiv 1 \; mod \; m$

**Example:** $n = 7$

| $z$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $z^{-1}$ | 1 | 4 | 5 | 2 | 3 | 6 |

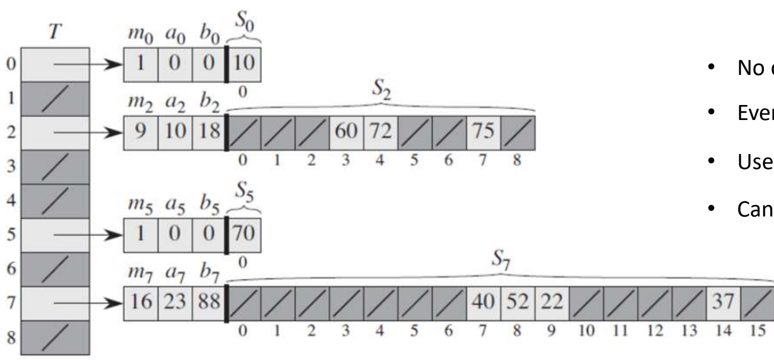
# Universal Hash Function

- Choose a large prime number $p$ so that any key $x \in \{0,1,\dots,p-1\}$
- $h_{a,b}(x) = \big((ax+b) mod\ p\big) mod\ m$, where $a \in \{1,2,\dots,p-1\}$ and $b \in \{0,1,\dots,p-1\}$
- $H_{p,m} = \{h_{a,b} \mid a \in \{1,2,\dots,p-1\}$ and $b \in \{0,1,\dots,p-1\}\}$
- $|H_{p,m}| = p(p-1)$

**Theorem:** The class $H_{p,m}$ of hash functions is universal

# Static Perfect Hashing

A static dictionary stores (key, value) pairs and supports:
➢ Lookup(key) (which returns value) – no inserts and deletes are allowed




- No collision
- Every lookup takes $O(1)$ worst-case time
- Uses expected $O(n)$ spaces
- Can be built in expected $O(n)$ time

$K = \{10, 22, 37, 40, 52, 60, 70, 72, 75\}$

Outer level hash function $h_{a,b}(x) = \big((ax+b) mod\ p\big) mod\ m$, where $a = 3, b = 42, p = 101, m = 9$

Secondary hash function $h_j(x) = \big((a_j x + b_j) mod\ p\big)\ mod\ m_j$

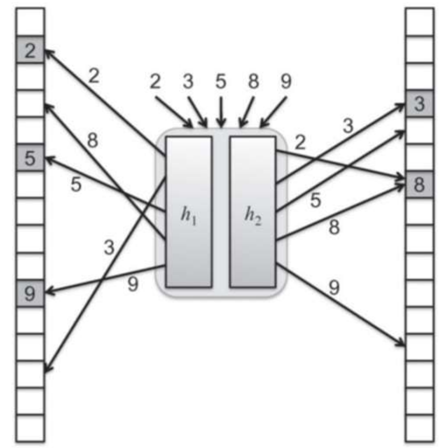
# Dynamic Perfect Hashing

A dynamic dictionary stores (key, value) pairs and supports:

- add(key, value), lookup(key) (which returns value) and delete(key)

In the Cuckoo hashing scheme:

- Two lookup tables $T_0, T_1$ of size $N$ are used ($N \geq cn$)
- Two different hash functions $h_0$(for $T_0$) and $h_1$(for $T_1$) are used
- Any key $x$ can be stored either in $T_0[h_0(x)]$ or $T_1[h_1(x)]$




- Lookup and delete take $O(1)$ worst-case time
- The space is $O(n)$, where $n$ is the number of keys stored
- An insert takes amortized expected $O(1)$ time

# Cuckoo Hashing

get($k$):
    **if** $T_0[h_0(k)] \neq$ NULL **and** $T_0[h_0(k)]$.key $= k$ **then**
        **return** $T_0[h_0(k)]$
    **if** $T_1[h_1(k)] \neq$ NULL **and** $T_1[h_1(k)]$.key $= k$ **then**
        **return** $T_1[h_1(k)]$
    **return** NULL

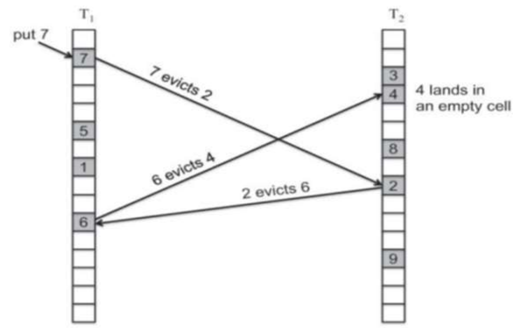
remove($k$):
    **if** $T_0[h_0(k)] \neq$ NULL **and** $T_0[h_0(k)]$.key $= k$ **then**
        $temp \leftarrow T_0[h_0(k)]$
        $T_0[h_0(k)] \leftarrow$ NULL
        **return** $temp$
    **if** $T_1[h_1(k)] \neq$ NULL **and** $T_1[h_1(k)]$.key $= k$ **then**
        $temp \leftarrow T_1[h_1(k)]$
        $T_1[h_1(k)] \leftarrow$ NULL
        **return** $temp$
    **return** NULL

put($k, v$):
    **if** $T_0[h_0(k)] \neq$ NULL **and** $T_0[h_0(k)]$.key $= k$ **then**
        $T_0[h_0(k)] \leftarrow (k, v)$
        **return**
    **if** $T_1[h_1(k)] \neq$ NULL **and** $T_1[h_1(k)]$.key $= k$ **then**
        $T_1[h_1(k)] \leftarrow (k, v)$
        **return**
    $i \leftarrow 0$
    **repeat**
        **if** $T_i[h_i(k)] =$ NULL **then**
            $T_i[h_i(k)] \leftarrow (k, v)$
            **return**
        $temp \leftarrow T_i[h_i(k)]$
        $T_i[h_i(k)] \leftarrow (k, v)$ // cuckoo eviction
        $(k, v) \leftarrow temp$
        $i \leftarrow (i + 1) \bmod 2$
    **until** a cycle occurs
    Rehash all the items, plus $(k, v)$, using new hash functions, $h_0$ and $h_1$

How to detect?

# Cuckoo Hashing

put$(k, v)$:
  **if** $T_0[h_0(k)] \neq$ NULL **and** $T_0[h_0(k)]$.key $= k$ **then**
    $T_0[h_0(k)] \leftarrow (k, v)$
    **return**
  **if** $T_1[h_1(k)] \neq$ NULL **and** $T_1[h_1(k)]$.key $= k$ **then**
    $T_1[h_1(k)] \leftarrow (k, v)$
    **return**
  $i \leftarrow 0$
  **repeat**
    **if** $T_i[h_i(k)] =$ NULL **then**
      $T_i[h_i(k)] \leftarrow (k, v)$
      **return**
    $temp \leftarrow T_i[h_i(k)]$
    $T_i[h_i(k)] \leftarrow (k, v)$   // cuckoo eviction
    $(k, v) \leftarrow temp$
    $i \leftarrow (i + 1) \bmod 2$
  **until** a cycle occurs
  Rehash all the items, plus $(k, v)$, using new hash functions, $h_0$ and $h_1$




An eviction sequence of length 3

**Lemma:** The probability that there is a possible sequence of evictions of length $L$ between a cell $x$ and a cell $y$ in $T_0 \cup T_1$ is at most $\frac{1}{2^L N}$