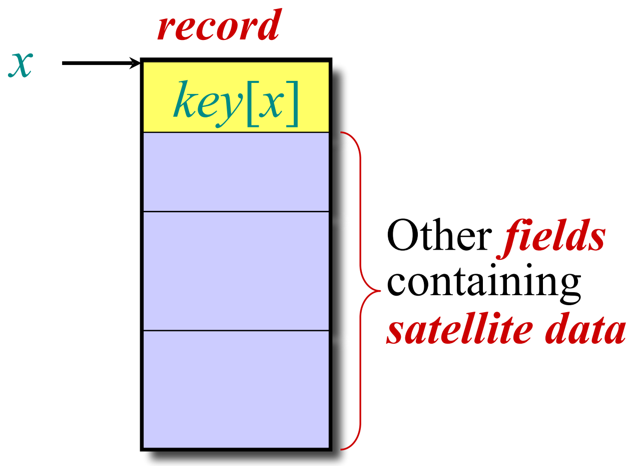
Hashing

Thanks to
Prof. Charles E. Leiserson

Symbol-table problem

Symbol table T holding n ***records***:



Operations on ***T***:

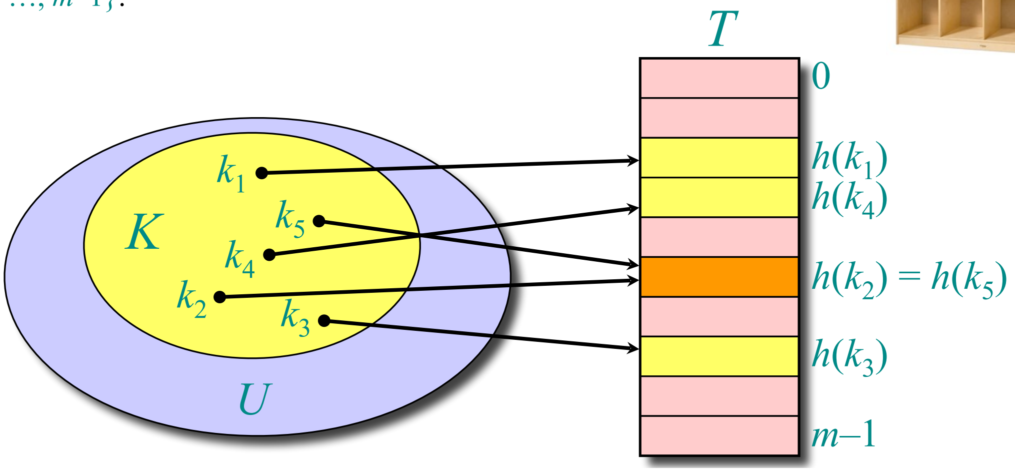
- INSERT(T, x)
- DELETE(T, x)
- SEARCH(T, k)

How should the data structure T be organized?

Hash tables and hash functions

We always have a table (cubby). Each cell has an index. The index is a number between *0..m-1*.

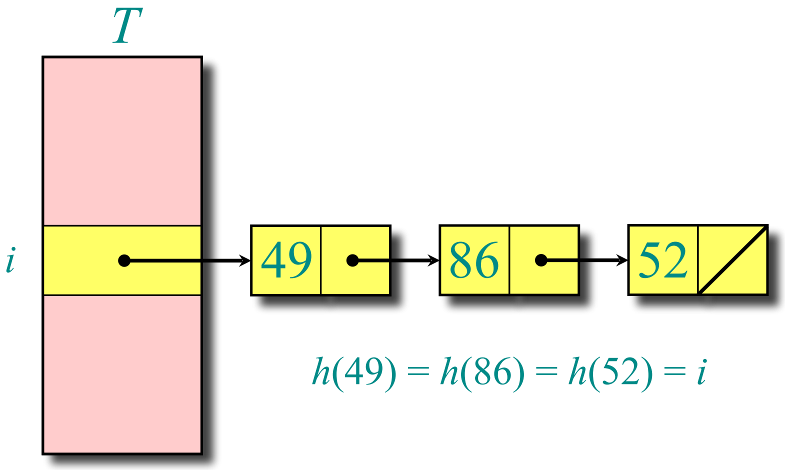
A ***hash function*** h computes for every **possible** key an index in a hash table $\{0, 1, \ldots, m-1\}$:



When a record to be inserted maps to an already occupied slot in T, a ***collision*** occurs.

Resolving collisions by chaining

- Records in the same slot are linked into a list.



$h(49) = h(86) = h(52) = i$

Analysis of chaining

Let n be the number of keys in the table, and let m be the number of slots.

Define the ***load factor*** of T to be

$$\begin{aligned}\alpha &= n/m \\ &= \text{average number of keys per slot.}\end{aligned}$$

We will try to keep the this value no larger than 1 (same number of keys and slots)

Search cost

Expected time to search for a record with a given key $= \Theta(1 + \alpha)$.

apply hash function and access slot

search the list

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

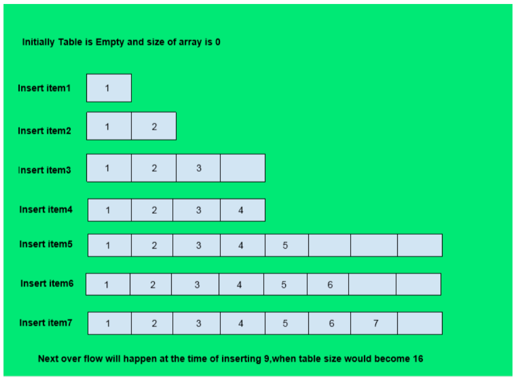
What to do if table too dense

Once α is too large

It does not effect corrections, but effects performances.
Once we have a chance, re-double the table (and compute a new hash function)
Example (credit GeeksforGeeks)

Start with a table of a small size
(Fig does not show the pointers to the linked list
When table too dense, double its size
sizes: 2,4,8,….

Total time: if it takes O(1) to re-insert a key, the total time for inserting **n** keys is $n\left(1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}\ldots\right) \le 2n$


Initially Table is Empty and size of array is 0

Next over flow will happen at the time of inserting 9,when table size would become 16

Choosing a 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.
- Hope: if $k_1 \neq k_2$ in **any** bit, then there is a good chance $h(k_1) \neq h(k_2)$
- Functions that ignore some bits (e.g. $h(k)=k \bmod 100$, $h(k)=k \bmod 1024$) should be used only if know enough about the data distribution to think that this is not an issue.

Division method

Assume all keys are integers, and define

$h(k) = k \bmod m$.

Deficiency: Don't pick an m that has a small divisor d. A preponderance of keys that are congruent modulo d can adversely affect uniformity.

Extreme deficiency: If $m = 2^r$, then the hash doesn't even depend on all the bits of k:

- If $k = 1011000111\underbrace{011010_2}_{h(k)}$ and $r = 6$, then $h(k) = 011010_2$.

Division method (continued)

$$h(k) = k \bmod m.$$

Pick m to be a prime not too close to a power of 2 or 10 and not otherwise used prominently in the computing environment.

Annoyance:

- Sometimes, making the table size a prime is inconvenient.

But, this method is popular, although the next method we'll see is usually superior.

Multiplication method

Assume that all keys are integers. Pick a constant integer *A,* and set

$$h(k) = (A \cdot k) \bmod m$$

A is an odd integer

Other variant of the multiplication method:

Pick A as a non-integer number

A=2.71828182846, or A=$\sqrt{2}$ = 1.41421356237

$$h(k) = \left\lfloor m\left((A \cdot k) - \lfloor A \cdot k \rfloor\right) \right\rfloor$$

Note - the part in the red parenthesis is a float in (0,1). Multiply my ***m*** gives a float in (0,m). The second floor just makes it a legit index in the table T[0…m-1].

Multiplication method example

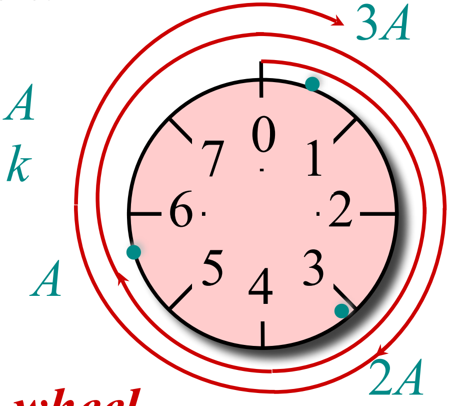
Variant 3: A is a large integer, but the value of *h(k)* is the number that several digits in the 'middle' of the (Ak).

$$\begin{array}{rr}
 & 1\,0\,1\,1\,0\,0\,1 = A \\
\times & 1\,1\,0\,1\,0\,1\,1 = k \\
\hline
 & 1\,0\,0\,1\,0\,1\,0\,\underbrace{0\,1\,1}_{h(k)}\,0\,0\,1\,1
\end{array}$$

Lets understand why all the variants of the multiplication method works nicely

- Think about a series of keys
- $k_1 = 1,\ k_2 = 2,\ k_3 = 3\ldots,$
- we hope that $h(k_1), h(k_2), h(3)\ldots$ fall in different and pairwise remote locations in the table.

$$\begin{array}{rr} & 1\,0\,1\,1\,0\,0\,1 = A \\ \times & 1\,1\,0\,1\,0\,1\,1 = k \\ \hline & 1\,0\,0\,1\,0\,1\,0\,\underbrace{0\,1\,1}_{h(k)}\,0\,0\,1\,1 \end{array}$$

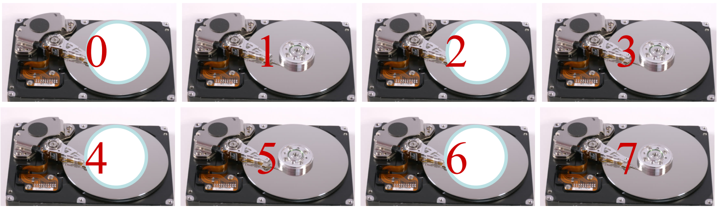


Modular wheel

Multiple hash functions:

Applications to distributed database. We need to store a large number n of records.

Lets think about a system with 8 disks.
Or 8 users, each in a different locations.



- RAID of disks is a device that contains multiple disks (sometimes sharing parts)
- Each Individual disk is prone to failures. Want to maintain **robustness** and **load fairness.**
    - **Robustness** to disk failures. We should still be able to access all our data, even if two disks crashed. So each record needs to be stored on multiple disks.
    - **Load fairness** Each disk should store a small portion of the database, and these portions should be split fairly. So each disk should contain approximately **3n/8** records.
    - **Efficiency:** Search time should be small. When searching for a key, should not have to check each individual disk
- Need to support: Insert(k)/delete(k) /find(k) .
- We don't know the data in advance - changing dynamically.
- So once a new record appears, we need to decide which 3 diskS will store it. Once a query find(k) appear, need to be able to find these 3 disks.

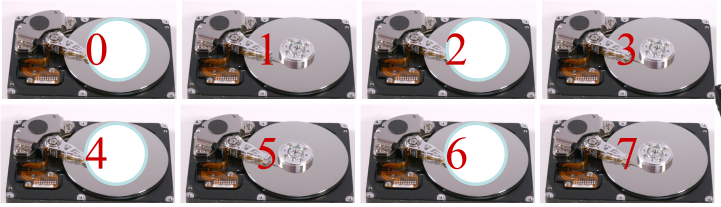
Multiple hash functions:

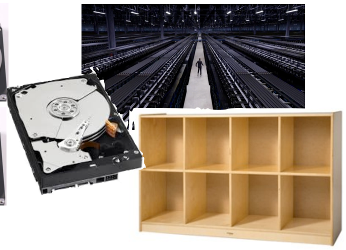
It is convenient sometimes to have multiple hash functions $\{h_1(k),\ h_2(k),\ h_3(k)\}$

We can generate them by picking 3 constants $A_1 \ldots A_3$ for example

$(3k) \mod 8,\ (5k) \mod 8,\ (7k) \mod 8$

We don't discuss here where in the disk each record is stored - orthogonal discussion.





- We don't know the data in advance - changing dynamically.
- Insert(k). A new record with key k appeared. Compute $h_1(k)$. Insert k into disk whose index is $h_1(k)$
  (example $k = 15, h_1(k) = (3k) \mod 8$, so we store this record in disk $(45 \mod 8) =$ disk 5).
  Similarly insert copies of k into disks $h_2(k),\ h_3(k)$.
- *Search(k):* Compute $h_1(k), h_2(k), h_3(k)$ and check these disks. If don't find, it is either because was never inserted or due to disk failures.

Dot-product method. Hashing large files.

In many applications, the key is too long to be considered a single number.

E.g. k="BDCZ".

In general, we need a hash functions that could be used on very long keys, as text documents, books, images, DNA, geometric structures, malware, viruses…

Expressing the key as a single number is not useful.

Idea: Remember that if the key k is a small number, we could use the multiplication method, and set *h(k)=(Ak)*mod *m*

Now if the key is very large, lets break it into several small pieces, so instead of tread the key as a single number, lets think about it as a **vector** (or a list) consisting of several numbers.

Instead of a single constant ***A*,** we compute multiple and different constants $a_1, a_2, a_3, a_4 \ldots$

We decompose the key into **characters**, multiply each by a different constant and sum (modulo m). Example:

$$\mathbf{h}(\mathbf{BDCZ}) = \left(a_1 \cdot 66 + a_2 \cdot 68 + a_3 \cdot 67 + a_4 \cdot 90\right) \mod m$$

(m is the size of the hash table. The ascii value of 'B' is 66 and of 'Z' is 90).

- Computing all constants a_i is very simple. Pick random integers between *1* and *m-1*.
- Excellent in practice, and in theory
- Involve one pass of the file, in the case of very long keys.

Dot product method-cont.

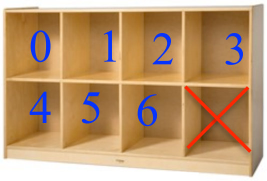
Before any data item arrives, decide about the size m of the hash table.
m should be prime, and >2n.
Let m $\approx 2^{20} = 1M$
Pick at random constants $\vec{\mathbf{a}} = (a_0, a_2, \ldots a_r)$.
Each a$_i$ is picked individually at random uniformly $1 < a_i < m - 1$
Now the first key k arrive. Lets break it into pairs of characters
k="According to section 1223(b) a nonprofit organization…"
Break into pairs of characters, and for each pair, compute its numeric value using base 256 (ASCII).

k=Ac|co|rd|in|g |to| s|ct

$\overbrace{Ac}^{k_0}\ \overbrace{co}^{k_1}\ \overbrace{rd}^{k_2}\ \overbrace{in}^{k_3}\ \overbrace{g\_}^{k_4}\ \overbrace{to}^{k_5}\ \overbrace{\_s}^{k_6}\ \overbrace{ec}^{k_7}\ \overbrace{ti}^{k_8}$ where $k_0 = \text{'A'} \cdot 256 + 'c' = 65 * 256 + 99$.

Finally $h_{\vec{\mathbf{a}}}(k) = \left(\sum_{i=0}^{r} a_i \cdot k_i \right) \mod m$

A deeper look at the dot product method



- Obviously, our aim is to minimizes collisions
- From now on, assume m (the table size) is a **prime** number.
- Assume all our keys $K = \{k_1 \ldots k_n\}$ are numbers, between 0..m-1. No key appears twice.
- Pick any constant integer $\alpha \in [1..m-1]$. Lets consider the hash function $h(x) = (\alpha x) \mod m$.

Lemma 1 : for every $Y \in [0..m-1]$, there is a **unique** $t \in [0..m-1]$ such that $(\alpha t) = Y \mod m$.

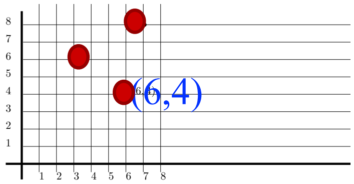
- Good news: The lemma guarantees that the hash function $h(x) = (\alpha x) \mod m$ will map the keys of K to different cells of the hash table. No collisions at all.
- Bad news: This guarantee is waved if we don't require that all keys of K are $< m$. For example, lets play with $h(x) = (3x) \mod 5$. Then $h(3) = h(8)$. So by itself, this is not very helpful. We will see next how to use it more efficiently.

- Before continuing, lets rewrite Lemma 1:

Lemma 2

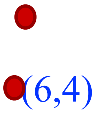
- For every fixed $Y \in [0..m-1]$, and every fixed $x_0 \in [1..m-1]$,
  …There is exactly one value $\alpha \in [0..m-1]$ such that $(x_0 \alpha) \mod m = Y$.

More on dot-product method



- Now think about a set of keys $K=\{p_1,\ldots p_n\}$ where each key is a point $p_i=(x_i,y_i)$ (for every i). These are points that we need to store in a hash table.
- Lets pick the table size m so $m \geq 2n$ and a m prime. Example: $n=10$, so we pick $m=23$.
- We want to choose a hash function that would map these points to the hash table.
- If we know which keys are in K, then we could create a **perfect** hash function that would create no collisions. But usually we don't know K, and even if we do, it does not worth the trouble.
- **Idea**: Pick **at random** two constants α, β, both in the range *[1..m-1]*. When we need to decide at which cell to store the point $p_i=(x_i,y_i)$, we use hash function is $h(p_i)=\big((\alpha x_i+\beta y_i) \mod m\big)$

More on dot-product method



- **Idea**: Pick **at random** two constants α, β, both in the range *[1..m-1]*. When we need to decide at which cell to store the point $p_i=(x_i,y_i)$, we use hash function is $h(p_i)=\big((\alpha x_i+\beta y_i) \mod m\big)$
- **Lemma 3.** The probability that $h(p_i)=h(p_j)$ is $\leq 1/\mathrm{m}$. That is, for any two points, the probability of a collision is really small.
- **Proof**: Assume $p_i=(x_i,y_i)$ and $p_j=(x_j,y_j)$. Since they are not the same point, assume $x_i \neq x_j$ (the case $y_i \neq y_j$ is symmetric)
  - If $h(p_i)=h(p_j)$ then $\big((\alpha x_i+\beta y_i) \mod m\big) = \big((\alpha x_j+\beta y_j) \mod m\big)$ which implies

$$\underbrace{\alpha\,(x_i-x_j)}_{=x_0} = \underbrace{\beta(y_j-y_i)}_{=Y} \mod m$$

  - Think about it this way: The values of x_i, x_j, y_i, y_j are given, and we have no control about them. We first picked β, so the value of $\beta(y_j-y_i)$ is fixed. The value (x_i-x_j) is also fixed. Now we (as a mental experiment) check the cases $\alpha=1,\ \alpha=2,\ldots\ \alpha=m-1$. Only for a single value of α the right box is equal to the left box.
  - In practice, instead of checking these values directly, we just pick α, β at random. **QED**

Dot-product method - cont

So we pick α, β at random from the range $[1..m]$. Lets pick two keys
$p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$
Conclusion from Lemma 3: The probability that
$h(p_1) = h(p_2)$ (that is, a collision occurs) is $\le \frac{1}{m} \le \frac{1}{2n}$.

Now consider a set $K = \{p_1, p_2 \dots p_n\}$ of n keys. lets ask what is the **expected number of collisions** between p_1 and the other keys of K. Using the same idea that we used for the hight of SkipList Analysis, this number is smaller that the some of each individual probability. That is,

$$\le \underbrace{\frac{1}{m}}_{\text{collisions } p_1,p_2} + \underbrace{\frac{1}{m}}_{\text{collisions between } p_1,p_3} + \dots + \underbrace{\frac{1}{m}}_{\text{collisions } p_1,p_n} = \frac{n}{m} \le \frac{\mathbf{1}}{\mathbf{2}}$$

- Next, assume that K is a set of n keys $K = \{k_1 \dots k_n\}$, each is a number in $[0\dots\mathbf{n}^2]$. Which hash function should we use ?
- As usual, we pick m as a prime $\ge 2n$.
- **Attempt 1**: Pick $\alpha \in [0..m-1]$ at random. Let $h(x) = (\alpha x) \mod m$. Possibly it works well, but no guaranties. A vicious adversary could pick the keys of K which are bad for almost every choice of α
-

Dot-product method - cont

- **Attempt 1**: Pick $\alpha \in [0..m-1]$ at random. Let $h(x) = (\alpha x) \mod m$. Possibly it works well, but no guaranties. A vicious adversary could pick the keys of K which are bad for almost every choice of α

- **Better approach.** For every key k_i , express k_i it in **base m**. $k_i = x_i m + y_i$. Now we are back to the case of 2D points.

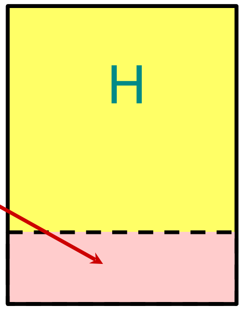
- Example for $m = 10$, $k_i = 35$. Then $x_i = 3$ and $y_i = 5$.
- Another example: $m = 11$, $k_i = 35$. Then $x_i = 3$ and $y_i = 2$. (since $3 \cdot 11 + 2 = 35$)
- Instead of expressing k_i in base m, we could use any other way to express k_i as two numbers (x_i, y_i), both $\le m-1$. For example, if $m \le 2^{16}$, and $k_i < 2^{32}$ then k_i has 4 bytes. We will use the first 2 bytes for x_i and the last two for y_i.

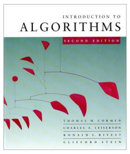
- Similarly, if each k_i is a number between 0 and m^3 we will pick at random 3 values $\alpha, \beta, \gamma \in [1..m-1]$. We express each k_i using 3 `digits' x_i, y_i, z_i, all in $[0..m-1]$. So $k_i = z_i m^2 + y_i m + x_i$.
- $h(k_i) = \left((\alpha x_i + \beta y_i + \gamma z_i) \mod m\right)$

- If the length of the key is unlimited (e.g. documents), we use round robin $\alpha_1, \alpha_2, \alpha_3, \alpha_4, \quad \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \quad \alpha_1, \alpha_2, \alpha_3, \alpha_4$
-

Universal family of hash functions

- We have a set of hash functions $H = \{h_1(k)\ldots h_L(k)\}$
- We say that it is **universal family** for every two keys $k_i, k_j \in U$, if we pick at random $h_i(k) \in H$, then the probability of a collision $h(k_i) = h(k_j)$ is $\leq 1/m$.
- That is, it is not worth than the probability of picking random cells for k_i, k_j.
- Only $\leq \frac{L}{m}$ of the functions of H cause collisions of k_i, k_j.
- If we think about all the possible hash functions $h((x_i, y_i)) = (\alpha x_i + \beta y_i) \mod m.$
- When we change $\alpha, \beta,$ (both in [0..m-1]), we create all different members of the family.
- We just saw that this family is universal.
- It guaranties that the probability of collusion between k_i, k_j is $\leq \frac{1}{m} \leq \frac{1}{2n}$, and that the average number of collisions between k_i and any other member of K is $\leq 1/2.$ That is, most cases, k_i has no collisions.





Universality is good

Theorem. Let h be a hash function chosen (uniformly) at random from a universal set H of hash functions. Suppose h is used to hash n arbitrary keys into the m slots of a table T. Then, for a given key x, we have

$$E[\#\text{collisions with } x] < n/m.$$

Proof of theorem (without random vars)

Proof. Let x, y be two keys, let $h_i \in H$ be a hash function. We define

$$c_{xy}\,(h_i) = \begin{cases} 1 & \text{if } h_i(x) = h_i(y) \\ & \text{otherwise.} \end{cases}$$

Remember: K is fixed - a set of n keys. x is one of them. We are worried about collisions between x and other members of K. Each different hash function causes other collision. So we ask what does the ‘average has function causes. .

$$\frac{1}{|H|}\sum_{h_i\in H}\sum_{y\in K} c_{xy}(h_i) = \frac{1}{|H|}\sum_{y\in K}\sum_{h_i\in H} c_{xy}(h_i) \le \frac{1}{|H|}\sum_{y\in K}\frac{|H|}{m} \le \frac{1}{|H|}n\frac{|H|}{m} = \frac{n}{m}$$

BTW - if $n \le m/2$, then this number $\le 1/2$

Proof of theorem (using random vars)

Proof. Let C_x be the random variable denoting the total number of collisions of keys in T with x, and let

$$c_{xy} = \begin{cases} 1 & \text{if } h(x) = h(y), \\ 0 & \text{otherwise.} \end{cases}$$

Note: $E[c_{xy}] = 1/m$ and $C_x = \sum_{y\in T-\{x\}} c_{xy}$.

$$E(C_x) = E\Big[\sum_{y\in K} c_{xy}\Big] = \sum_{y\in K} E[c_{cy}] = n/m$$

Constructing a set of universal hash functions

Let m be prime. Decompose key k into $r + 1$ digits, each with value in the set $\{0, 1, \ldots, m-1\}$. That is, let $k = \langle k_0, k_1, \ldots, k_r \rangle$, where $0 \le k_i < m$.

Randomized strategy:

Pick $a = \langle a_0, a_1, \ldots, a_r \rangle$ where each a_i is chosen randomly from $\{0, 1, \ldots, m-1\}$.

Define $h_a(k) = \sum_{i=0}^{r} a_i k_i \bmod m$.

Dot product, modulo m

How big is $\mathsf{H} = \{h_a\}$? $|\mathsf{H}| = m^{r+1}$. ← **REMEMBER THIS!**

Perfect hashing

A hash function is perfect (for a set K of n keys) if $h(x) \neq h(y)$ for every $x, y \in K$.

How could we find such a function?

Deterministic algorithm - hard.

Randomize algorithm. Let’s assume (unrealistically) that we could pick a really large table $m = n^2$. Pick $h \in H$ from from a universal family. The probability of no collision is

$$E[\sum_{x,y \in X} c_{xy}(h)] = \binom{n}{2}\frac{1}{m} = \frac{n(n-1)}{2}\frac{1}{m} = \frac{1}{2}\frac{n(n-1)}{n^2} \le \frac{1}{2}$$

Markov’s inequality says that for any nonnegative random variable X, we have

$$\Pr\{X \ge t\} \le E[X]/t.$$

So in this case, (large m), if we pick h at random, we have %50 chance to hit a perfect function.

Algorithm: Pick h at random. If perfect - great. If not - repeat

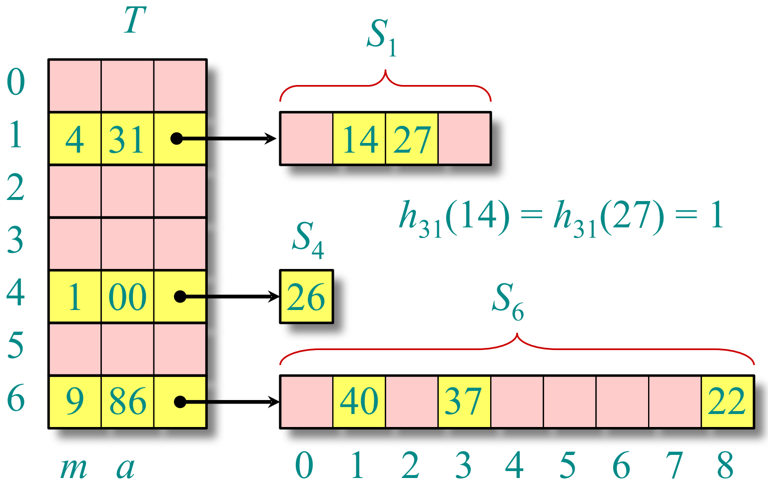
Expected number of trails = $1\frac{1}{2} + 2\frac{1}{4} + 3\frac{1}{8} + \ldots i\frac{1}{2^i} + \ldots = 2$.

Toward a Perfect hashing with linear storage.

Use one hash function h to partitions K into sets $S_1, S_2 \ldots S_m$. The set S_i set contains all the keys that are mapped to cell i in the table. $S_i = \{x \in K \mid h(x) = i\}$

Note: h does not guarantie no collisions. But the number of collisions is small

For each S_i build another hash table (only for this set) with table size $m_i = |S_i|^2$, and no collisions at all. (see previous slide)



$h_{31}(14) = h_{31}(27) = 1$

Analysis of storage

For the level-1 hash table T, choose $m = n$, and let n_i be random variable for the number of keys that hash to slot i in T. By using n_i^2 slots for the level-2 hash table S_i, the expected total storage required for the two-level scheme is therefore

$$E\left[\sum_{i=0}^{m-1} \Theta\left(n_i^2\right)\right] = \Theta(n),$$

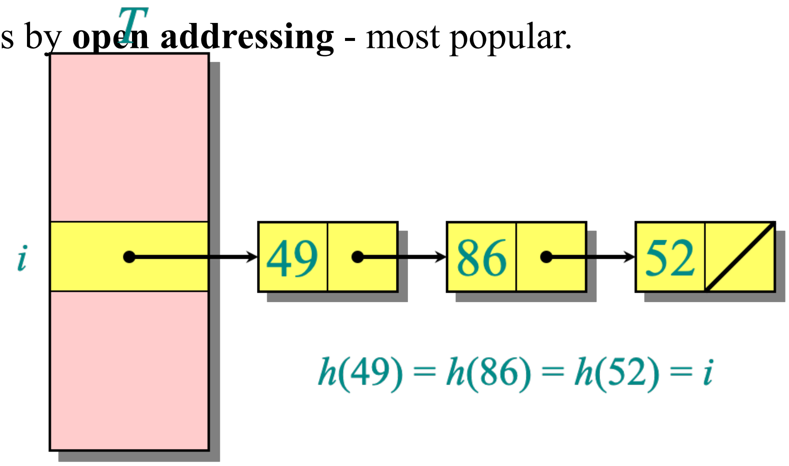
since the analysis is identical to the analysis from recitation of the expected running time of bucket sort. (For a probability bound, apply Markov.)

Resolving collisions

Several approaches

1. **Chain hashing** - all keys mapped to the same cell are stored in a linked list. (Less popular in practice - dynamic memory allocation is slow, multiple vulnerabilities, less friendly to compiler-optimization, GPU unfriendly…
2. **Cuckoo hashing** - will discuss later

3. Resolving collisions by **open addressing** - most popular.



Resolving collisions by open addressing

No storage is used outside of the hash table itself.

Each cell could contain at most one key.

The same key ***k*** might be mapped by ***h(k)*** to different locations in the table, depending on availability.

When either searching k or searching for a place for k, we will check

The **first** index that we search k. If fail
The **second** index that we search k. If fail
The **third** index that we search k. If fail etc

When should we give up? (will see in next slides)

How should we find these indexes ?

h(k, i)- a hash function that takes two parameters:

Key ***k***

Trial number ***i*** (first trail has index 0)

Resolving collisions by open addressing

No storage is used outside of the hash table itself..

- The hash function depends on both the **key** and **probe number**:

$h(k,i)$

input is a pair: a key and a trial number. 0,1,2,…m-1

Output: Always a legit index in the table T[ ]. a number in the range 0,1...m-1

E.g.

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

  here $h_2(k)$ is some other hash function
- $f(k,i) = (k+i^2) \bmod m$

Inserting a key k:

we check $T[h(k,0)]$. If empty we insert k, there. Otherwise,

we check $T[h(k,1)]$. If empty we insert k, there. Otherwise,…

otherwise etc for $h(k,2), h(k,3), \ldots, h(k,m-1)$.

Finding a key k:

we check whether $T[h(k,0)] == k$. If not, if empty, stop. otherwise

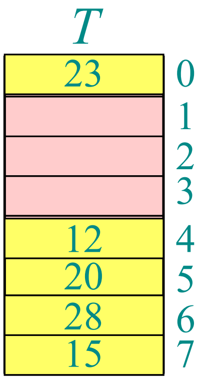
we check whether $T[h(k,1)] == k$. If not, if empty, stop. otherwise

otherwise etc for $h(k,2), h(k,3), \ldots, h(k,m-1)$.

Example of Insertion

Hash function: $h(k,i)=(k+i) \bmod 8$

k-key. i is the attempt number (start at 0)



- insert(12). h(12,0)=4

  Read: The first attempt (i=0) checks T[h(12,0)]. It is free
- insert(15). h(15,0)=7
- insert(20). h(20,0)=4 (collision)

  h(20,1)=(20+1)mod 8=5

  T[5] is empty. Place 20 at T[5]
- insert(23). h(23,0)=7 (collision)

  h(23,1)=0
- insert(28). h(28,0)=4 (collision) ;

  h(28,1)=5(collision);

  h(28,2)=6

Inserting a key k:

we check $T[h(k,0)]$. If empty we insert k, there. Otherwise,

we check $T[h(k,1)]$. If empty we insert k, there. Otherwise,…

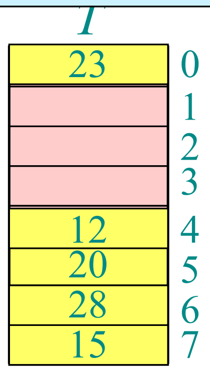
etc for $h(k,2), h(k,3), \ldots, h(k,m-1)$.

Searching a key. Example on the same table

Hash function: *h(k,i)=(k+i) mod 8*

Finding a key *k:*
we check if T[*h*(*k*,0)] =*k*. If not, if empty, stop. otherwise
we check if T[*h*(*k*,1)] =*k*. If not, if empty, stop. other etc

k-key. i is the attempt number (start at 0)



23	0
	1
	2
	3
12	4
20	5
28	6
15	7

\`Search' uses the same probing sequence. The Search stops once it hits an empty cell, or i=n-1.

Example. Search 28. First check h(28,0)=4, but T[4]≠28. Next check h(28,1)=5 but T[5]≠28. Next T[6]=28 - success.

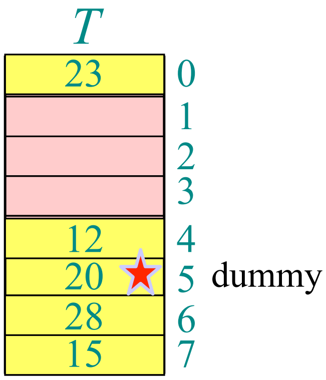
Search(16). h(16,0)=0. T[0]≠16. Next check h(16,1)=5, but T[5]-empty. Search terminates - 16 not in table.

Searching a key. Example on the same table

Hash function: *h(k,i)=(k+i) mod 8*

k-key. i is the attempt number (start at 0)

- Next, delete 20, and then lets again search 28.
- The search wrongly stops at the empty cell that used to contain 28. Error
- Solution: Place a **dummy** to indicate that this cell used to contain a key, but this key was deleted. The 'search' treats this cell as 'nonempty' and continues the probing sequence. The search stops only when reaching a cell that is "really" empty.
- When inserting a new key, we can replace the dummy with a real key. Example - inserting 13 will override the dummy



23	0
	1
	2
	3
12	4
20 ★	5
28	6
15	7

\`Search' uses the same probing sequence. The Search stops once it hits an empty cell, or i=n-1.

Example. Search 28. First check h(28,0)=4, but T[4]≠28. Next check h(28,1)=5 but T[5]≠28. Next T[6]=28 - success.

Search(16). h(16,0)=0. T[0]≠16. Next check h(16,1)=5, but T[5]-empty. Search terminates - 16 not in table.

Maintenance

Scan the table from time to time, and get rid of all of all dummies.

Re-insert each key,

If the table needs to be expanded - good opportunity to use the dynamic table technique and re-hash.

Probing strategies

Linear probing:

Given an ordinary hash function $h'(k)$, linear probing uses the hash function

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

This method, though simple, suffers from ***primary clustering***, where long runs of occupied slots build up, increasing the average search time. Moreover, the long runs of occupied slots tend to get longer.

Theoretically, inferior method.

In practice, is the fastest method. Why ? In the memory hierarchy, locality is a winner. If we accessed T[i], then it is likely that T[i+1] is awaiting in cache.

Probing strategies

Double hashing

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

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

This method generally produces excellent results, but $h_2(k)$ must be relatively prime to m. One way is to make m a power of 2 and design $h_2(k)$ to produce only odd numbers.

The expected number of probs, until an empty slot is found

Recall $\alpha = n/m$ - load factor

Assumption: At every i, the probability of hitting cell j is 1/m (uniformly)
Lets call a prob a ``success'' if we hit an empty cell, and ``fail'' if hit an occupied cell.
The sequence probs ends with a successful prob.

The probability that exactly 0 fail probs are needed is $1-\alpha$
(success on first try)

The probability that exactly 1 fail probs is needed is $\alpha(1-\alpha)$
(fail, then success)

The probability that exactly 2 fail probs are needed is $\alpha^2(1-\alpha)$
(fail, fail then success)

The probability that exactly 3 fail probs are needed is $\alpha^3(1-\alpha)$
(fail, fail, fail then success)

The probability that exactly **j** fail probs are needed is $\alpha^j(1-\alpha)$
(j fails, then success)

So the expected number of probs is

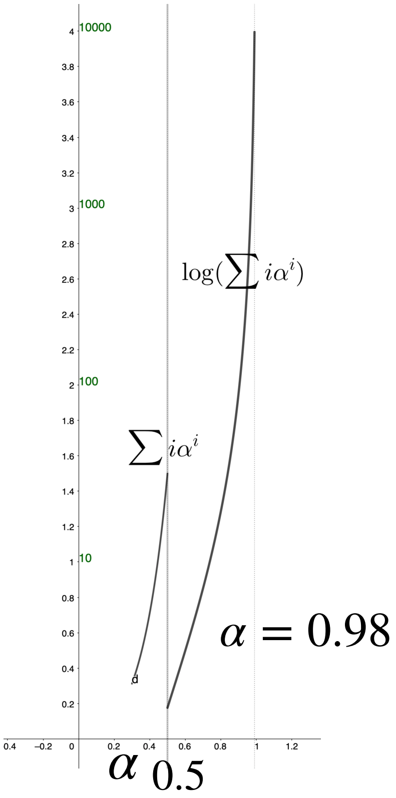
$$1_{\text{sucessful prob}} + (1-\alpha)\sum_{j=1}^{\infty} \alpha^j \cdot j_{fails}$$

Expected number of probs (cont)

Conclusions:
If $\alpha = 0.5$, the expected number of probs is 2
If $\alpha = 0.95$, the expected number of probs is 300
If $\alpha = 0.98$, the expected number of probs is 10^4

Conclusion: Keep $m \geq 2n$.

Rehash twice a day: first thing every morning, and before bed time.

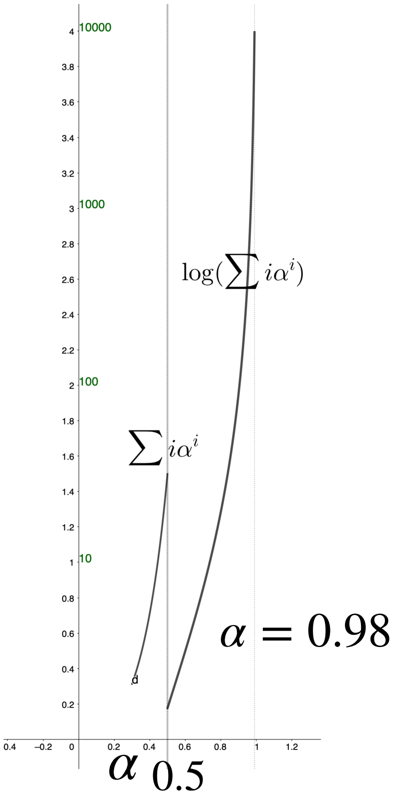


Expected number of probs (cont)

Conclusions:
If $\alpha = 0.5$, the expected number of probs is 2
If $\alpha = 0.95$, the expected number of probs is 300
If $\alpha = 0.98$, the expected number of probs is 10^4

Conclusion: Keep $m \geq 2n$.

Rehash twice a day: first thing every morning, and before bed time.



The third method to resolve collisions
Cuckoo Hashing

Hash table that supports (as usual):

- insert(k) expected $O(1)$ time
- search(k) ~~expected~~ worst-case $O(1)$ time
- delete(k) ~~expected~~ worst-case $O(1)$ time

Use two tables T, T', each with $\approx n$ cells, and two hash functions $h(k), h'(k)$.

We use $h(k)$ to search in T, and we use $h'(k)$ to search in T'.

```
Search(k){
    If $T[h(k)] = k$ OR $T'[h'(k)] = k$ return FOUND
    else return NOT FOUND
    }
```

Cuckoo Hashing - Deletion

Noting new for deletion:

Delete(k) -search for it. If found, replace by a gravestone/flag.

Next insert could write over the flag (similar t open addressing)

Rehashing removes this flag.