Hashing

- Usual Problem: find record for given key
- Best time far: O(logn) – AVL/B-Tree/Binary Search; O(m) (Trie)
- can we do it in O(1)?
- Yes – go "directly" from key to record in table (Random Access assumed)
  - If keys are integers, key can be index to array
- Problems?
  - wasted space
  - for SID (9 digits) need table (array) sized 1,000,000,000
- Solution: Convert key into integer in desired range
- Hash Function: function h(k) which transforms key, k, into an index (slot #)
- Collision: happens when $h(k_1) = h(k_2)$, i.e. 2 keys transformed into same slot
- Collision Resolution: procedure to follow after collision to find(empty) slot for this key

Uses in Computer Science:

Often more efficient than search trees Thus, widely used in CS. Some common uses:

Associative arrays
Hash tables are commonly used to implement many types of in-memory tables.
They are used to implement associative arrays (arrays whose indices are arbitrary strings or other complicated objects), especially in interpreted programming languages like AWK, Perl, and PHP.

Database indexing
Hash tables may also be used for disk-based persistent data structures and database indices (such as dbm) although balanced trees are more popular in these applications.

Caches
Hash tables can be used to implement caches, auxiliary data tables that are used to speed up the access to data that is primarily stored in slower media. In this application, hash collisions can be handled by discarding one of the two colliding entries—usually the one that is currently stored in the table.

-e.g., Operating System memory and disk caches
-e.g., Memcached (a distributed memory caching system in C)

- speeds up dynamic database-driven websites by
- caching data/objects in RAM to reduce number of database reads
- API: giant hash table distributed across multiple machines.
- table full → subsequent inserts purge older data (least recently used order)
- Applications using Memcached typically layer requests and additions into core before falling back on a slower backing store, such as a database.
- used by YouTube, Facebook , Twitter, Google Apps, etc

Sets

Besides recovering the entry which has a given key, many hash table implementations can also tell whether such an entry exists or not.
Those structures can therefore be used to implement a set data structure, which merely records whether a given key belongs to a specified set of keys. In this case, the structure can be simplified by eliminating all parts which have to do with the entry values.
Hashing can be used to implement both static and dynamic sets.

Object representation

Several dynamic languages, such as Python, JavaScript, and Ruby, use hash tables to mplement objects.
In this representation, the keys are the names of the members and methods of the object, and the values are pointers to the corresponding member or method.

Unique data representation

Hash tables can be used by some programs to avoid creating multiple character strings with the same contents. For that purpose, all strings in use by the program are stored in a single hash table, which is checked whenever a new string has to be created.
This technique was introduced in Lisp interpreters under the name hash consing, and can be used with many other kinds of data (expression trees in a symbolic algebra system, records in a database, files in a file system, binary decision diagrams, etc.)

[Reference: http://en.wikipedia.org/wiki/Hash_table]

Hashing Functions

suppose table size = 1000
need slots 0-999

1) Truncation
   e.g. h(2647983) = 983 – some fixed # of digits from fixed spot
   problem? May throw away Unique part of key. e.g. all auto parts end in "83"

2) Middle Square
   e.g. k = 4263 4263^2 = 18173169 or 18173169 that's 173 or 731
   reduce patterns (collisions) somewhat

3) Folding
   partition k int sections and recombine
   e.g. k = 782146

782	782
+ 146	+ 641
928	1423 *truncate if necessary

   could subtract, add, multiply, divide, etc.

   NOTE: above-table size must be power of 10 (10, 100, 1000, …)

4) Division
   h(k) = mod (k, m) m=table size

   Best result when m is prime (less collisions and cover and covers table well – Standish proves)

   e.g.

key	M=13 (prime)	M=12
558	12	6
723	8	3
692	3	8
876	5	0
574	2	10
945	9	9
716	1	8
201	6	9
946	10	10

Will tend to get multiples of factors of 12.

e.g.
4: 2x4=8
3: 3x3 = 9 etc.

Examples

- given table size m = 100 slots
- hash following keys into table: 382650, 125

Use Truncation to the last 2 digits

h(382650) →

h(125) →

Use Truncation to the first 2 digits

h(382650) →

h(125) →

Use Middle-Square, keeping 2 digits (go left if odd)

h(382650) →

h(125) →

Use Folding in pairs (truncate to first if necessary)

h(382650) →

h(125) →

Use Folding in triplets (truncate to first if necessary)

h(382650) →

h(125) →

Use Division

h(382650) →

h(125) →

Collision Resolution

Open Addressing: put key int in some other (empty) slot in table. Standish does 2 kinds: Linear Probe, double Hashing

Linear Probe

Insert: if k gets hashed into full slot, s, put in first empty slot in s – 1, s – 2, … (with wrapping) until: find k or hit empty slot (not found)

e.g. m = 10 truncation to last digit

0		Insert		0 300
1				1
2				2 492
3			355 →5	3
4			300 →0	4
5			492 →2	5 355
6				6
7				7
8				8
9				9

	Insert		0 300
			1 682
			2 492
		982 →2 →1	3
			4
			5 355
			6
			7
			8
			9

	Insert		0 300
			1 682
			2 492
		101 →1 →0 →9	3
			4
			5 355
			6
			7
			8
			9 101

Retrieve 861 →1^x→ 0^x →9^x →8 Not found

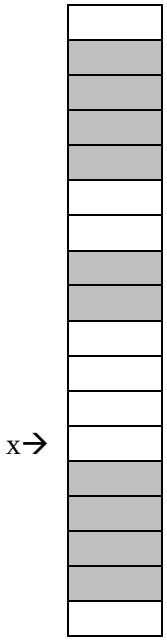
101 →1^x→ 0^x →9 YES Retrieve record

Table "full" when m – 1 slots occupied.

Leave 1 empty so unsuccessful search can terminate.

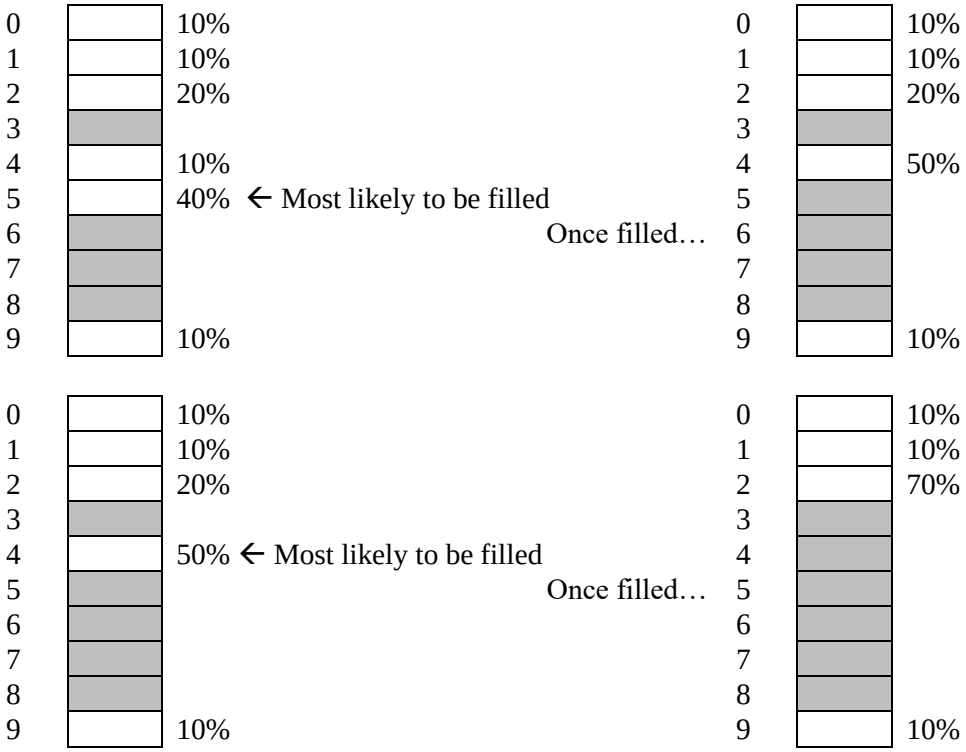
Problems: Primary Clustering

- A few keys randomly near each other tend to collect into clusters
- Clusters combine into bigger clusters



- Any key hashing into this band gets put into slot x clusters grow this way
- grows more and combine with above cluster

Assuming good random hash, probability slots get filled



More clustering problems

Both tables are 50% full:

Table 1

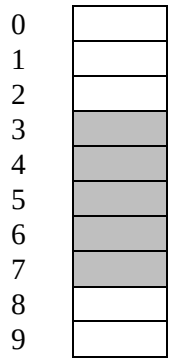
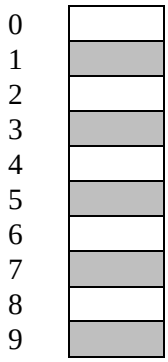


Table 2



- Suppose h(k) = mod(k, m)
- Search for key 364597 which is NOT in table
  Worst case number of problems
  table 1: 6 probes
  table 2: 2 probes
- insert key 364597
  Worst case number of problems
  table 1: 6 probes
  table 2: 2 probes
- search/insert closer to O(n) than O(1)

Double Hashing

- No primary clustering (only random ones)
- Instead of going down by 1 each time, go down by some other amount, called the probe decrement, p(k)

e.g. Suppose h(k) is division h(k) = mod(k,m), p (k) = max(1, k/m) (quotient integer division), m = 101

h(207) = mod(207,101) = 5
h(914) = mod(914,101) = 5 occupied p(914) = 914/101 = 9
try slot 5 – 9 = 97 (wrap) if full,
try slot 97-9 = 88 if full,
try slot 88-9=79 etc.

NOTE: don't have to use division and quotient, could use other hash functions for h(k) and p(k)

When $h(k_1) = h(k_2)$, k_1 follows a different probe sequence than k_2

e.g. h(712) = mod(712, 101) = 5. p(712) = 712/101 = 7

if 5 full try 5 – 7 = 99
if 99 full try 99 – 7 = 92
if 92 full try 92 – 7 = 85 …

key	h(key)	D.H. Probes Sequence	L.P. Probes Sequence
914	5	97, 88, 79 …	4, 3, 2, …
712	5	99, 92, 89 …	4, 3, 2, …

Retrieving:
- search for 712
- try 5, 99,92, 85 ... until empty slot (not found) or found

Deleting:
- delete key/record but set flag in slot to indicate "keep searching", why?
- suppose h(207) = 5 and h(914) = 5^X → 97
  if no flag … delete 207 then try to retrieve 914 but slot 5 empty and return not found WRONG
- note flag ignored for inserting

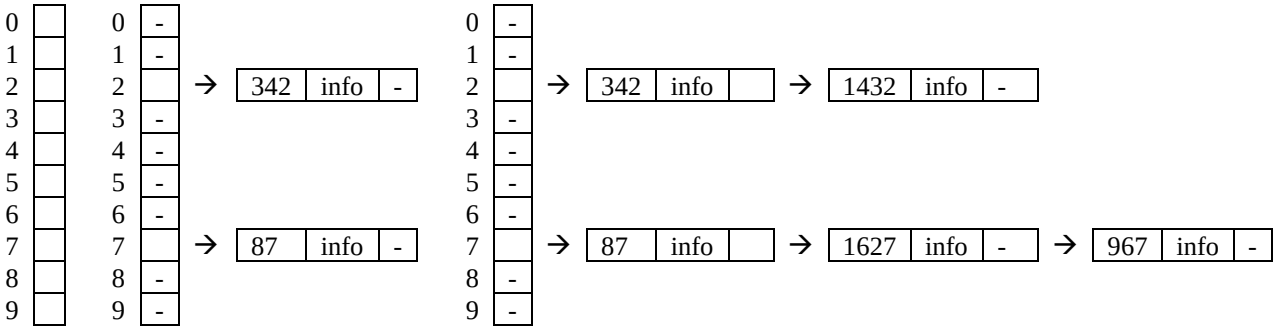
Problems with Open Addressing

1) Fixed m. If # records grow > m: locate larger table and re-hash all records
2) The "keep searching" flags might cause us to search through most of the table in order to conclude NOT FOUND. (If lots of set flags)

Separate Chaining

- Table entry contains pointer to linked list
- h(k) = slot => add record to linked list for slot

e.g. m = 10, truncation to last digit
insert 87, 342, 1627, 1432, 967



Advantages

- deletion has no effect on subsequent retrievals
- table size < in Open Address and less need to re-locate larger one

Disadvantages

- can take more space (links)
- if lists too long, efficiency drops

General Rule of Thumb: if record size large compared to pointer size advantages win

Improvements:

- insert at front of list, or
- keep list ordered (faster search) or tree

Collision Probability

Are Collisions all that likely?

YES! For table m = 365 that's 13% full (n=47), there's a 95% chance of at least 1 collision

Probably of at least 1 collision when n keys and table size m

$$p(n) = 1 - \frac{m!}{m^n(m-n)!}$$

This is a generalization of Von Mises probability argument – the birthday paradox.

if >= 23 people in room, there's a >50% chance 2 or more of them will have same birthday (m=365)

Can we prove it?

q(n) = probability when randomly put n keys in table size m no collisions

p(n) = probability at least 1 collision

p(n) = 1 - q(n)

Notice q(1) put 1 key in empty table, it is certain there are no collisions (so q(1) = m/m)

- when put 2nd key in table, chance of hitting empty slot is now m-1/m so

$$q(1) = q(1) \cdot \frac{m}{m} = \frac{m}{m} \cdot \frac{m-1}{m}$$

- when put 3rd key in table, chance of hitting empty slot is now m-2/m so

$$q(2) = q(2) \cdot \frac{m-2}{m} = \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m}$$

- when put nth key in table, chance of hitting empty slot is now (m-n+1)/m
  q(n) = q(n-1) * (m-n+1)/m = m/m * (m-1)/m * (m-2)/m … (m-n+1)/m

$$q(n) = q(n-1) \cdot \frac{m-n+1}{m} = \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \ldots \frac{m-n+1}{m}$$

$$q(n) = \frac{\left(m \cdot (m-1) \cdot (m-2) \cdot \ldots \cdot (m-n+1)\right)}{m^n} = \frac{m!}{m^n(m-n)!}$$

$$\therefore p(n) = 1 - \frac{m!}{m^n(m-n)!}$$

Analysis of Hashing

n = # entries in Table
m = Table size

Load Factor: $\alpha = \frac{n}{m}$

Linear Probe:

successful search/retrieval $\frac{1}{2} \cdot (1 + \frac{1}{1-\alpha})$

unsuccessful search/retrieval $\frac{1}{2} \cdot (1 + (\frac{1}{1-\alpha})^2)$

e.g. $\alpha = 50\%$ (= ½):
1.5 probes for successful search
2.5 probes for unsuccessful search

Double Hashing:

successful search/retrieval $\frac{1}{2} \cdot ln(\frac{1}{1-\alpha})$

unsuccessful search/retrieval $\frac{1}{1-\alpha}$

Separate Chaining:

successful search/retrieval $1 + \frac{1}{2}\alpha$

unsuccessful search/retrieval α

See Standish for proof

Standish Tables 11.21 and 11. 22 show # probes for various methods for various Load Factors

Successful	0.10	0.25	0.50	0.75	0.90	0.99
Chain	1.05	1.12	1.25	1.37	1.45	1.49
L. Probe	1.06	1.17	1.5	2.5	5.50	50.5 (16)
Double H.	1.05	1.15	1.39	1.85	2.56	4.65
Unsuccessful						
Chain	0.10	0.25	0.50	0.75	0.9	0.99
L. Probe	1.12	1.39	2.50	8.50	50.5	5000 (360)
Double H.	1.11	1.33	2.00	4.00	10.0	100.0 (99)

Note: Performance depends on Load Factor (how full table is), not # keys in table. See what happens if we keep Load Factor < 50%

Table Resizing

How to choose m?

m = size of array where n = # keys

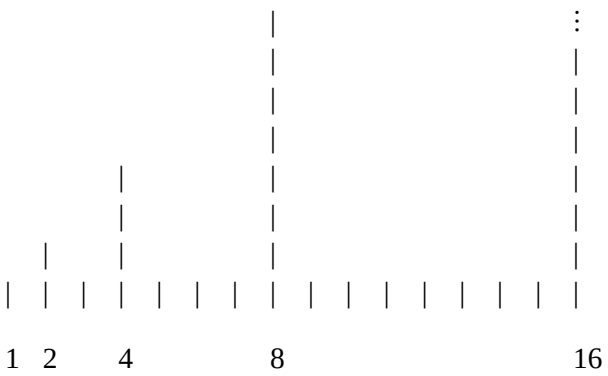
If m > n : grow table

- make (allocate) table of size m'
- build new hash (rehash)
  for each item i
    insert(item)
- run time is $n + m + m'$

- if $m' = m + 1$
  run time is $1 + 2 + 3 + \cdots + n = O(n^2)$

- if $m' = 2m$
  run time is $1 + 2 + 4 + 8 + \cdots + n = O(n)$
  Table Doubling

Amortization



1 2 4 8 16

k inserts takes k times

Amortized time for insertion is O(1)

Deletion:

- if $m = \frac{n}{2}$ then shrinks $\frac{m}{2}$
  slow when insert/delete at $2^k \leftrightarrow 2^k + 1$

- if $m = \frac{n}{4}$ then shrinks $\frac{m}{2}$
  amortized time is $O(1)$