# Hashing

**Reading:**

- Chapter 9 on searching, including hashing



## Hashing Issues

Must devise an appropriate hash function. Because the hash function maps a large space to a small space, we will have "collisions".

We can make each location a "bucket" that can store lots of records.

- But buckets must have fixed size, thus they can still overflow.

We will need a scheme to handle this.

Must decide on the # and size of buckets.

When file gets very full, collisions can be too numerous. May be worthwhile re-organizing the file layout to have more buckets



## Hashing Performance

If everything is well designed, retrieval can be very fast — just a few file accesses.

One operation is really slow:



## Hash functions

A hash function is a mathematical function that maps from
keys ⇒ locations.

There are some standard types of hash function, including

- **mid-square:** square the number and then take some digits from the middle.
- **folding:** Divide the number in half and combine the two halves, *e.g.*, add them together.
- **modular division:** Mod by some number, preferably a prime.

See the text for more about hash functions. Note that there is a lot of interesting theory about hash functions and their properties. (csc 378 covers this.)

## Examples of hash functions

Say our key is a string. Before we hash it, we need to turn it into an integer.

One solution: Concatenate together the alphabetic position of the 1st and the second character. **E.g.** "$\boxed{\text{T}}\boxed{\text{o}}$yota" ⇒ $\boxed{20}\boxed{15}$.

Now we need a hash function to hash up the integer. Examples of the three general types:

- Mid-square, taking the middle 2 digits. **E.g.** $2015^2 = 406\boxed{02}25$.
- Folding: adding the two halves.
  **E.g.** 2015 ⇒ 20 + 15 = 35.
- Mod by 97.
  **E.g.** 2015 mod 97 = 75.



| key | | h(key) | | |
|---|---|---|---|---|
| as string | converted | mid-square | folding | mod 97 |
| Toyota | 2015 | $406\boxed{02}25$ | 35 | 75 |
| Chev | 0308 | $94\boxed{86}4$ | 11 | 17 |
| Ford | 0615 | $37\boxed{82}25$ | 21 | 33 |
| Chrysler | 0308 | $94\boxed{86}4$ | 11 | 17 |
| Volkswagen | 2215 | $490\boxed{62}25$ | 37 | 81 |
| Nissan | 1409 | $198\boxed{52}81$ | 23 | 51 |
| Plymouth | 1612 | $259\boxed{85}44$ | 28 | 60 |
| Dodge | 0415 | $17\boxed{22}5$ | 19 | 27 |
| Renault | 1805 | $325\boxed{80}25$ | 23 | 59 |
| Saab | 1901 | $361\boxed{38}01$ | 20 | 58 |
| Isuzu | 0919 | $84\boxed{45}61$ | 28 | 46 |
| Pontiac | 1615 | $260\boxed{82}25$ | 31 | 63 |
| Fiat | 0609 | $37\boxed{08}81$ | 15 | 27 |

Two kinds of collisions:

- collisions that occur because we begin with the same 2 integers. Are collisions for every hash function we might choose.
- collisions that occur even though we begin with 2 different integers. Not necessarily collisions with a different hash function.



Couldn't we use the 2015 as the hashed value?

This would be a bad idea. With 4 digits, there are 10,000 possible values (0...9999), yet only a few will be used.

- Some are unlikely to crop up
  **E.g.** "aa" ⇒ 0101.
- Some *cannot* crop up
  **E.g.** ?? ⇒ 2701; 27 is out of range.

Yet we need our hash table to be continuous, and therefore to have all 10,000 slots.
So our hash table will be largely empty.

With each of the 3 hash functions we looked at, the range of h(key) is 0...99 (or less with mod 97).
So our hash table only needs 100 slots.

Of course we might overflow it, but we have to deal with this anyway.



## Avoiding collisions?

Upon doing a new insertion, how likely is a collision?

It's certainly more likely when many items have already been inserted.

**Loading factor:** $= \frac{\#records\ currently\ in\ the\ file}{\#records\ that\ the\ file\ can\ hold}$

In our cars example, the loading factor is only $\frac{13}{100} = 0.13$, yet we already have collisions!

We could reduce collisions by making the capacity of the file bigger (and hence the loading factor smaller). But ...

## Exactly how likely are collisions?

For a given file capacity, how likely are collisions as file gets more loaded?

Example: A file of 365 buckets. Let $Q(n)$ be the probability that NO collisions occur during $n$ insertions.

$Q(1) =$

$Q(2) =$

$Q(3) =$

In general,
$Q(n) =$

Q(1) =



Solution to the recurrence relation:

$Q(n) = \frac{365!}{365^n(365-n)!}$

The probability that collisions DO occur is $1 - Q(n)$.

| $n$ | $1 - Q(n)$ |
|---|---|
| 10 | 0.1169 |
| 20 | 0.4114 |
| 23 | 0.5073 |
| 30 | 0.7063 |
| 40 | 0.8912 |
| 50 | 0.9704 |
| 60 | 0.9941 |

If the loading factor is only $\frac{23}{365}$, 50% chance.

If the loading factor is only $\frac{47}{365}$, $>$ 95% chance!

So yes, collisions are a problem!



## Buckets

The hash function $h(k)$ tells us where to store (or retrieve) a record with key $k$.

This could be a slot in an array in memory, or a slot in a file. (For this course, a file.) Either way, we often use the term "hash table".

The slots are called "buckets", because they have capacity for $> 1$ record. We choose the capacity based on the number of records that can be read or written in one file access.

Analogy: your address book.
**key:**

**hash function:**

**bucket:**



## Handling collisions

What do we do when a collision occurs?

Easy case: the bucket has room for the record.

Hard case: the bucket doesn't have room for the record. We call this "overflow".
We need to figure out two things:

- A place to put the record that won't fit it its home bucket.
- A way to find that record later!

How do you handle overflow in your address book?

## Handling overflow

Two kinds of approach: either compute or store where to try next. (Gee, where have we heard that before?)

**Open addressing**

Compute another bucket to try, based on some rule.

**Closed addressing (or chaining)**

Store the location of another bucket to try, using some sort of pointer.

The “overflow” records may be kept in a separate overflow area, perhaps in a separate file.



## Open Addressing

Compute where to look, based on the key.

General method for insertion:
(search is analogous; why?)

Let $A_i$ be where to look on the ith try.

- Use the hash function ($h$) to find the first bucket where the record might go, $A_0$.
  $A_0 = h(key)$
- If that bucket is full,
  use a new function ($f$) to find the next bucket to try. Repeat as necessary.
  $A_i = f(i, key)$
- Stop when we hit a bucket with room, or the sequence of $A_i$'s starts to repeat (*i.e.*, there is no room anywhere).

Many ways to design the function $f$ ...



## I. Linear Probing

Step through a sequence of buckets always using the same step size.

Use mod to wrap around when we hit the end of the hash table.

$$\boxed{A_i = (A_{i-1} + stepSize) \bmod n}$$

(We can also express $A_i$ in terms of $A_0$.)

Example:

$$\begin{aligned} A_i &= (A_{i-1} + 2) \bmod n \\ &= (A_0 + 2i) \bmod n \end{aligned}$$

The sequence of buckets considered is called the “probe sequence”.



**Linear probing example**

n (# buckets) = 13     h(key) = (sum 1st 3) mod 13
bucket size = 1
step size = 2     So $A_i = (A_{i-1} + 2)$ mod 13

| key | h(key) | probe sequence |
|---|---|---|
| Chevrolet | 16 mod 13 = 3 | |
| Chrysler | 29 mod 13 = 3 | |
| Jaguar | 18 mod 13 = 5 | |
| Nissan | 42 mod 13 = 3 | |
| | | |
| | | |
| Karman Ghia | 30 mod 13 = 4 | |

| Bucket # | Bucket Contents |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | |
| 9 | |
| 10 | |
| 11 | |
| 12 | |

## II. Non-linear Probing

Problem: **Primary clustering**.

If several records hash to the same spot, or even *any* spot along the probe sequence, they will all follow that same probe sequence.

Example: n = 100; step size = 2.
hash to: 5 probe seq:
hash to: 9 probe seq:

Solution: Make the step size depend on the step number, $i$.
This is called **non-linear probing**.

*E.g.*, $A_i = (A_{i-1} + 2i^2)$ mod $n$

Example: n = 100; step size = $2i^2$.
hash to: 5 probe seq:
hash to: 9 probe seq:



## III. Double Hashing

Problem: **Secondary clustering**.

All records that hash to the same spot still have the same probe sequence.

Example: n = 100; step size = $2i^2$.
key 1; hash to: 5 probe seq:
key 2; hash to: 5 probe seq:

Solution: Make step size depend on the *key* (but differently than in original hash function).
This is called **double hashing**.

*E.g.*, $A_i = (A_{i-1} + h_2(key))$ mod $n$



## Closed Addressing

Instead of computing where to look next, store it, using a "pointer".

Each full bucket has a pointer to an overflow area,

- in the same file (for example at the end)
- or in another file

There are many ways to organize this, since pointers are so flexible.

Cost vs open addressing:
Savings:

Do deletions introduce problems?