Implementing a dictionary

- easy approach: an array
  - * one spot for each key - its own designated spot
    - => all ops $\Theta(1)$

Problem? May have lots of wasted space

1. Keys are often not integers
   - eg strings, VIN # for cars
2. Storage if $n \ll M$ ← size of array/table
   - ← keys in use

eg student #s: 20 855 555

- 20: common prefix
- 855 555: 1,000,000 possible keys but UW does not have this many students

- often # keys used << # possible

Direct-Addressing is okay if we can allocate the space for one position for each key.

We want to reduce amount of space. lots of it empty

So instead of using  space

reduce to

Idea: Let M be # of buckets in hash table T.
Let $n = |K|$ = actual # KVPs stored in table

Goal: Reduce storage to $\Theta(n)$
but still get $O(1)$ time ops
at least in average case

Don't store key k in bucket k.
- use hash function h and store in bucket $h(k)$

h: Universe of keys $\rightarrow \{0, 1, \ldots, M-1\}$
- all possible keys
- U
- set of buckets
- size of hash table

- so $h(k)$ is a valid bucket / index of T
- key k "hashes" to index $h(k)$

If $M < |U|$ then ... problem?
- two or more keys hash to same index
  => collision!!!

How likely?
Birthday Paradox

Birthday Paradox

What is the probability 2 people have a birthday on the same day?

1 - Prob that no 2 people share a birthday

$365 \times 364 \times 363 \times \ldots (365-(N+1))$

Something like: $1 - \frac{\#\text{ subsets of } \{1,2,\ldots,365\} \text{ of size } n}{365^n}$

$n=2 \Rightarrow 0.2\%$

$n=10 \Rightarrow 12\%$

$n=23 \Rightarrow 50.73\%$ ~ more likely to happen than not

$n=50 \Rightarrow 97\%$

** Collisions are very likely!

=> we need a method to handle them!

Chaining

- Average case: M buckets, n keys
  - on average $\frac{n}{M}$ keys per bucket

Search:

Best Case: 1 - hash to bucket, first item in chain

Worst Case: check all items in bucket (chain)

average $O(\frac{n}{M})$

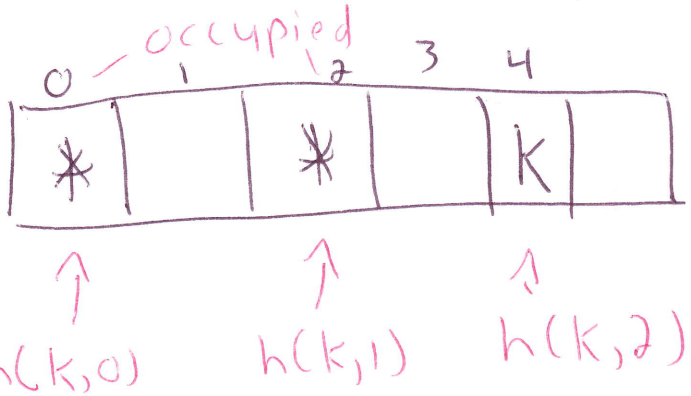
- average ~ not first, not last but half way.

Open Addressing

- can only fit 1 per cell of table.

If this spot is occupied, try another
=> probing sequence

Careful with Delete:



- insert here
- 3 probes to find an open spot

What if items at index 0 and 2 are deleted?
How do you find K?

- lazy delete = mark as deleted
- cell is empty & item is deleted are different

Linear Probing

Problem: primary clustering

- a sequence of cells gets full
- any key hashing to anywhere in this block goes to end & extends block
- bigger the cluster, larger area it spans
  => larger chance get a collision & expand
  vicious cycle

- longer { search times / insert

α - load factor $\frac{n}{m}$

α	$\frac{1}{2}$	$\frac{2}{3}$	$\frac{3}{4}$	$\frac{9}{10}$
hit	1.5	2.0	3.0	5.5
miss	2.5	5.0	8.0	55.5

Why so big?

- As the table fills up, instead of having multiple clusters they start merging into a super cluster, ...

Double Hashing

Idea: stop formation of clusters by using 2 independent hash functions.

Independent! if $h_1(k_1) = h_1(k_2)$ collision

then $h_2(k_1) \neq h_2(k_2)$ • unlikely but possible

Suppose $M = 2^{10} = 1024$

$h_1(k) = 10$ and $h_2(k) = 256$ for some k

probes: 10, 266, 522, 778, 10, 266, ... • cycles over spots

=> choose prime # for M and $1 < h_2(k) < M$

• no common factors

α	$\frac{1}{2}$	$\frac{2}{3}$	$\frac{3}{4}$	$\frac{9}{10}$
hit	1.4	1.6	1.8	2.6
miss	1.5	2.0	3.0	5.5

10x faster than linear probing

Analysis: Ave # probes

• successful search: $\frac{1}{2} \ln\left(\frac{1}{1-\alpha}\right)$

unsuccessful $\frac{1}{1-\alpha}$

Cuckoo Hashing

- 1 KVP per slot
- 2 independent hash functions
- always insert k to T[h0(k)]
  - may "kick out" another item
    - insert ↑ at alt location, etc

"Good" Hash function

- efficient to compute
- be unrelated to possible patterns in the data
- depend on all parts of the key