Hash tables

- key: part of an element used for comparisons

- a big array
  \( m \): length of array

hash function \( h \): keys \( \rightarrow \{0, 1, \ldots, m-1\} \)

store element \( x \) in array position \( h(x) \)
Problem: hash collisions:

\[ h(x) = h(y) \text{ for } x \neq y \]

We cannot just put them in the same array cell.

We want a good hash function that is:
1) fast
2) avoids collisions
but we will see collisions, so what to do?

if we do well, containing, remove & insert all take \( O(1) \) time

but-
you will lose ordering into no fast min Element
no \( O(n) \) time list of elements in sorted order
Hash functions say keys are integers.

Idea 1: \( h(x) = x \mod m \)

But if \( m = 2 \),
we only use low order
6-bits for hash — bad if
keys are big & use few sig figs

Idea 16: make sure \( m \) is prime
ASSUME AN EVIL USER!!

Just hash several values differing by multiples of m to make things slow.

Have to use randomness (let std writers take care of this for you).

Pick a random hash function h from a family
of hash functions $\mathcal{H}$

(Monday we will see a good family)

For now, assume we're given a good one, that evals in old time.

universal hash family $\mathcal{H}$

For any two keys $x \neq y$ 

+ randomly selected $h \in \mathcal{H}$

$\Pr[h(x) = h(y)] \leq \frac{1}{m}$
Collisions:

1) Chaining: Each array cell is a linked list holding its elements.

Analysis: Let \( l(x) \) be elements in cell \( h(x) \). Time for \( \text{contains}(x) \) is...
\[0(1 + l(x)) \text{ time} \]

\[C_{x,y} := \left[ h(x) = h(y) \right] \]

\[\text{indicator variable} \]

\[1 \text{ if } h(x) = h(y) \quad 0 \text{ o.w.} \]

\[E\left[l(x)\right] = E\left[\exists y \in C_{x,y}\right] \]

\[= \exists y \in E\left[C_{x,y}\right] \]

\[= \exists y \in \Pr\left[h(x) = h(y)\right] \]

\[= \frac{1}{m} \]

\[= \frac{n}{m} \]
\[ \lambda = \frac{n}{m} \text{ load factor} \]

expected time for an operation is \( \Theta(1+\lambda) \)

chain with red-black trees

operations take \( \Theta(1+\log t) \) in expectation

or even another hash table...
Probing:
Each array cell still stores $\leq 1$ element, so we look for a different cell during collision.

You have a sequence of hash functions $h = h_0, h_1, \ldots$

contains/insert: check position $h_i(x)$ for each
If until you find x or an empty position. Then report answer or insert x at that position.

Remove: don't null out x's cell. Instead mark it so we know other probes should move past it.
```java
public boolean contains(E element) {
    int currentPos = findPos(element);
    return isActive(currentPos); // returns false if we marked element as removed
}

// uses linear probing
private int findPos(E element) {
    int currentPos = myhash(element);

    while (table[currentPos] != null && !array[currentPos].element.equal(element)) {
        currentPos++;
        if (currentPos >= table.length) {
            currentPos = 0;
        }
    }

    return currentPos;
}
```

**Linear probing**

\[ h_i(x) = (h(x) + i) \mod m \]
If $\lambda < 1$, expected # probes is $O(1)$

As $\lambda$ approaches 1, you get chunks of full cells: primary clustering.

$\lambda$ cannot be $\geq 1$!

When table is too full, do rehashing with a bigger table and new hash function.
Quadratic probing

$h_\omega(x) = (h(x) + \omega^2) \mod m$

If \( \lambda \leq \frac{1}{2} + m \text{ is prime} \), you can always find a spot to insert a new element.

Note $\omega^2 = (\omega-1)^2 + 2\omega - 1$

so you can find positions with only addition.
Double hashing:

Pick a second hash function \( h' \).

\[
\begin{align*}
h'(x) &= (h(x) + i \cdot h'(x)) \\ & \quad \mod m
\end{align*}
\]

Expect \( \Theta(1) \) probes for quadratic probing & double hashing.
Perfect Hashing: Chaining but secondary structures are hash tables.

Want a small worst-case time for an operation.

Intuition: balls & bins

We throw n balls into n bins uniformly at random. How full is fullest bin?

Answer: $\Theta\left(\frac{\log n}{\log \log n}\right)$
But what about more bins?

Expected # collisions is

\[ \sum_{x \neq y} \Pr(h(x) = h(y)) \leq \binom{n}{2} \frac{1}{m} \leq \frac{n(n-1)}{2m} \]

If \( m = n^2 \), this # is \( < \frac{1}{2} \).

\( \Rightarrow \) with prob \( > \frac{1}{2} \) we have no collisions.
Perfect hashing:

Assume we know $n$ in advance.

Make a table of length $N$.

$n_{\text{in}} = \# \text{elements in cell } i$

Put them in a hash table of length $m_{\text{in}} = n_{\text{in}}^2$.

(each gets its own hash function)

Rebuild each secondary table until it has no
collisions. (Expected one rebuild each)

Worst-case operation time of $\Theta(1)$.

Thm: $E[\sum_{i=1}^{n} n^2] = 2n$

So structure is expected to use $\Theta(n \log n)$ space.