Hashing: (Unordered) dictionary
- stores key-value pairs in array table [0..m-1]
- supports basic dict. ops. (insert, delete, find) in $O(1)$ expected time
- does not support ordered ops (getMin, findUp, ...)
- simple, practical, widely used $h(x)$

Overview:
- To store $n$ keys, our table should (ideally) be a bit larger (e.g., $m > cn$, $c=1.25$)
- Load factor: $\lambda = n/m$ - How congested?
- Running times increase as $\lambda \rightarrow 1$
- Hash function: $h : \text{Keys} \rightarrow [0..m-1]$
  - Should scatter keys random.
  - Need to handle collisions $x \neq y \text{ but } h(x) = h(y)$

Recap: So far, ordered dicts.
- insert, delete, find
  - Comparison-based: $<, ==, >$
  - getMin, getMax, getK, findUp...
- Query/Update time: $O(\log n)$
  - Worst-case, amortized, random. Eg. Let $p$ large prime $a, b \in [0..p-1]$ all random
  - Can we do better? $O(1)$?

Universal Hashing:
- Even better $\rightarrow$ randomize!
- Let $H$ be a family of hash fns
  - Select $h \in H$ randomly
  - If $x \neq y$ then $\Pr[h(x) = h(y)] = \frac{1}{m}$

Why $\mod p \mod m$?
- modding by a large prime scatters keys
- $m$ may not be prime (e.g., power of 2)

Good Hash Function:
- Efficient to compute
- Produce few collisions
- Use every bit in key
- Break up natural clusters

Common Examples:
- Division hash: $h(x) = x \mod m$
- Multiplicative hash: $h(x) = (ax \mod p) \mod m$
- Linear hash: $h(x) = ((ax + b) \mod p) \mod m$
Overview:
- Separate Chaining
- Open Addressing:
  - Linear probing
  - Quadratic probing
  - Double hashing
- Simple/slow
- Complex/fast

Collision Resolution:
- If there were no collisions, hashing would be trivial!
- Rehash!
  - If $\lambda < \lambda_{min}$ or $\lambda > \lambda_{max}$?
  - Alloc. new table size = $n/\lambda_0$
  - Compute new hash fn $h$
  - Copy each $x,v$ from old to new using $h$
  - Delete old table

Separate Chaining:
- `table[i]` is head of linked list of keys that hash to $i$.
- Worst-case: `log n`

Example:
```
Keys (x) | h(x) | table
---|---|---
 0  | 2  | 0
 1  | 4  | 1 0 4
 2  | 7  | 2 1
 3  | 0  | 3
 4  | 7  | 4
 5  | 3  | 5
 6  | 0  | 6
 7  | 4  | 7
```

Hashing II

Thm: Amortized time for rehashing is $1 + \left(2\lambda_{max}/(\lambda_{max} - \lambda_{min})\right)$

How to control $\lambda$?

- Rehashing: If table is too dense/too sparse, realloc to new table of ideal size
- $\frac{1}{2} \leq \lambda \leq \frac{3}{2}$
- $\frac{4}{5} \leq \lambda \leq \frac{5}{4}$

Designer: $\lambda_{min}, \lambda_{max}$ - allowed $\lambda$ values

```
$\lambda_0 = \frac{\lambda_{min} + \lambda_{max}}{2}$ "ideal"

Proof:
- On avg. each list has $n/m = \lambda$
- Success: 1 for head + half the list
- Unsuccess: 1 """" + all the list

Analysis:
- Recall load factor $\lambda = n/m$ $n$ = # of keys
- $m$ = table size
Announcements: 4/18

- Prog Assign 2, Part 2 -
  Extended deadline - Fri, Apr 21

- Midterm 2 - solutions posted
  (still grading)

- Prog Assign 3 - Soon