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

Recap: So far, ordered dicts.
- insert, delete, find
- Comparison-based: \(<, =, >\)
- getMin, getMax, getK, findUp...
- Query/Update time: \( O(\log n) \)
  \( \rightarrow \) Worst-case, amortized, random.
  \( \nabla \) 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 \( \text{Prob}(h(x) = h(y)) = \frac{1}{m} \)
  Eg. Let \( p \) - large prime, \( a \in [0..p-1] \) all random
  \[ h_{a,b}(x) = ((ax+b) \mod p) \mod m \]

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

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

Good Hash Function:
- Efficient to compute
- Produce few collisions
- Use every bit in key
- Break up natural clusters
  Eg. Java variable names:
  \( \text{temp1, temp2, temp3} \)

Common Examples:
- Division hash:
  \[ h(x) = \frac{x}{m} \mod m \]
- Multiplicative hash:
  \[ h(x) = (ax + b) \mod p \mod m \]
  \( a, p \) - large prime numbers
- Linear hash:
  \[ h(x) = (ax + b) \mod p \mod m \]
  \( a, b, p \) - large primes
Overview:
- Separate Chaining
- Open Addressing:
  - Linear probing
  - Quadratic probing
  - Double hashing

Separate Chaining:
- Table[i] is head of linked list of keys that hash to i.

Example:
- Keys: d, e, p, w, t, f
- Hash function: h(x) = x % 7
- Table:

<table>
<thead>
<tr>
<th>Index</th>
<th>h(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
</tr>
<tr>
<td>1</td>
<td>e</td>
</tr>
<tr>
<td>2</td>
<td>p</td>
</tr>
<tr>
<td>3</td>
<td>w</td>
</tr>
<tr>
<td>4</td>
<td>t</td>
</tr>
<tr>
<td>5</td>
<td>f</td>
</tr>
</tbody>
</table>

Collision Resolution:
- If there were no collisions, hashing would be trivial!
- Hash function: h = hash(func)
- Insert(x, v) → table[h(x)] = v
- Find(x) → return table[h(x)]
- Delete(x) → table[h(x)] = null

If λ < λmin or λ > λmax? Rehash!
- Allocate new table size = n/λ
- Compute new hash fn h
- Copy each x, v from old to new using h
- Delete old table

Thm: Amortized time for rehashing is
\[ 1 + \frac{2λ_{max}}{(λ_{max} - λ_{min})} \]

How to control λ?
- Rehashing: If table is too dense/too sparse, reallocate to new table of ideal size = \( \frac{n}{\lambda_0} \)

Designer: \( \lambda_{min}, \lambda_{max} \) - allowed λ values
\[ \frac{1}{2} \leq \lambda_0 = \frac{\lambda_{min} + \lambda_{max}}{2} \leq \frac{3}{4} \]

Analysis: Recall load factor \( \lambda = \frac{n}{m} \)
- n = # of keys
- m = table size

Proof: On avg, each list has \( \frac{n}{m} = \lambda \)
- Success: 1 for head + half the list
- Unsuccess: 1 " " + all the list

Threshold: \( \lambda_{min}, \lambda_{max} \) - allowed λ values
If \( \lambda < \lambda_{min} \) or \( \lambda > \lambda_{max} \) ...
Open Addressing:
- Special entry ("empty") means this slot is unoccupied.
- Assume $\lambda \leq 1$
- To insert key $x$:
  - check $h(x)$ if not empty try $h(x)+i_1$
  - $h(x)+i_2$
  - $h(x)+i_3$
- What's the best probe sequence?

Collision Resolution (cont.):
- Separate chaining is efficient, but uses extra space (nodes, pointers...)
- Can we just use the table itself?

Analysis:
- Improves primary clustering
- May fail to find empty entry
  - Try $m=4$, $j^2 \mod 4 = 0 \rightarrow 1$ but not $2 \rightarrow 3$
- How bad is it? It will succeed if $\lambda < \frac{1}{2}$

Thm: If quad probing used $m \rightarrow \infty$
- prime, the the first $Lm/2L$ probe locations are distinct.

Pf: See latex notes.

Linear Probing:
- $h(x), h(x)+1, h(x)+2, ...$
- $h(x)$
- Simple, but is it good?
- $x: 0, 1, 2, p, w, t$
- $h(x): 0, 2, 2, 0, 1$

Hashing III

Primary Clustering
- Clusters form when keys are hashed to nearby locations
- Spread them out!

Quadracl Probing:
- $h(x), h(x)+1, h(x)+4, h(x)+9, ...$
- $h(x)$
- Primary clustering
- Obs: As $\lambda \rightarrow 1$ times increase rapidly

Obs: As $\lambda \rightarrow 1$ times increase rapidly.

Analysis: Improves primary clustering

Thm: $S_{ULP} = \frac{1}{2} (1 + \frac{1}{1 - \lambda})$

Primary Clustering
- Clusters form when keys are hashed to nearby locations
- Spread them out!
Double Hashing:
(See best of the open-addressing methods)
- Probe sequence det'd by second hash fn. - g(x)
  \( h(x) + \{0, g(x), 2g(x), 3g(x)\} \mod m \)

Recap:
- Separate Chaining:
  Fastest but uses extra space (linked list)

Open Addressing:
- Linear probing:
  - Primary clustering
- Quadratic probing:
  - Secondary clustering

Delete c(x):
- Apply find(x)
  → Not found ⇒ error
  → Found ⇒ set to "empty"

Problem:
- insert(a): \( h(a) \)
- delete(c): \( h(c) \)
- find(a): \( h(a) \) "a" not found

Find(x):
- Visit entries on probe sequence until:
  - found x ⇒ return v
  - hit empty ⇒ return null

Thm: \( S_{DH} = \frac{1}{2} \ln \left( \frac{1}{1-\lambda} \right) \)
\( U_{DH} = \frac{1}{1-\lambda} \)

Expected search time of double hash, if successful:
\( E_{DH} = \frac{1}{2} \ln \left( \frac{1}{1-\lambda} \right) \)
Expected if unsuccessful:
\( U_{DH} = \frac{1}{1-\lambda} \)
Recall: Load factor \( \lambda = \frac{n}{m} \)

Proof is nontrivial (skip)

Dictionary Operations:
- Insert (x, v):
  Apply probe sequence until finding first empty slot.
  - Insert (x, v) here.
  (If x found along the way ⇒ duplicate key error!)

- Why does bust up clusters?
  - Even if \( h(x) = h(y) \) [collision] it is very unlikely that \( g(x) = g(y) \)
  ⇒ Probe sequences are entirely different!
Programming Assignment 2