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

Overview:
- To store n keys, our table should (ideally) be a bit larger (e.g., m = cn, c=1.25)
- Load factor: λ = n/m - How congested?
- Running times increase as λ → 1
- Hash function
  ₋ h: Keys → [0..m-1]
  → Should scatter keys random.
  → Need to handle collisions

Recap: So far, ordered dicts
- insert, delete, find
- Comparison-based: <, ==, >
- getMin, getMax, getK, findUp...
- Query/Update time: O(log n)
- Even better - randomize!

Universal Hashing:
- Let H be a family of hash fns
- Select h ∈ H randomly
- If x ≠ y then \( \text{Prob}(h(x) = h(y)) = \frac{1}{m} \)
- Can we do better? O(1)?
- Let \( P \) be a large prime, @ \([1..P-1]\) all random

Hashing I

Good Hash Function:
- Efficient to compute
- Produces 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:
  \( h(x) = (ax + b) \mod p \mod m \)
  \( a, b, p \) - large prime numbers
  \( x \) - key

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

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
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!
- Insert \( x, v \) → table \[ h(x) \] = v
- Find \( x \) → return table \[ h(x) \]
- Delete \( x \) → table \[ h(x) \] = null

If \( \lambda < \lambda_{\text{min}} \) or \( \lambda > \lambda_{\text{max}} \) ? Rehash!
- Alloc. new table size = \( \frac{n}{\lambda_0} \)
- 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 \( i \).

Example:
- Keys \( x \)
- \( h(x) \)
- Table

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

Expected search time:
- \( S_{\text{se}} \) = Expected search time
- If \( x \) found (successful)
- \( U_{\text{se}} \) = Expect. search time if \( x \) not found (unsuccessful)

Thm: Amortized time for rehashing is:
\[
1 + \frac{2\lambda_{\text{max}}}{\lambda_{\text{max}} - \lambda_{\text{min}}}
\]

How to control \( \lambda \)?
- Rehashing: If table is too dense/too sparse, realloc. to new table of ideal size
- Designer: \( \lambda_{\text{min}}, \lambda_{\text{max}} - \text{allowed values} \)
- \( \lambda_0 = \frac{\lambda_{\text{min}} + \lambda_{\text{max}}}{2} \) "ideal"
- If \( \lambda < \lambda_{\text{min}} \) or \( \lambda > \lambda_{\text{max}} \) ...
Open Addressing:
- Special entry ("empty") means this slot is unoccupied
- Assume \( \lambda \leq 1 \)
- To insert key:
  - check: \( h(x) \) if not empty try \( h(x) + i_j \) \( \mod m \)
- Probes sequence
- 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 secondary clustering
- May fail to find empty entry
- How bad is it? It will succeed if \( \lambda < \frac{1}{2} \)

Thm: If quad. probing used + mod success / failures, the the first \( \frac{m}{2} \) probe locations are distinct.

Pf: See latex notes.

Hashing III

Analysis:
Let \( S_{LP} \) = expected time for successful search
\[
U_{LP} = \begin{cases} \frac{1}{2} (1 + \frac{1}{1-\lambda}) & \text{if } \text{success} \\ \frac{1}{2} (1 + \frac{1}{1-\lambda})^2 & \text{if } \text{failure} \end{cases}
\]

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

Quadratic Probing:
- \( h(x), h(x)+1, h(x)+4, h(x)+9, \ldots \)
- \( h(x) \mod m \)

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

Obs: As \( \lambda \to 1 \) times increase rapidly

incremental
Double Hashing:
(Best of the open-addressing methods)
- Probe sequence det'd by 
  second hash fn. \(-g(x)\)
  \(h(x) + \{0, g(x), 2g(x), 3g(x), \ldots \}\)
- \(g(x) = 2g(x)\) \(mod \ m\)

Recap:
Separate Chaining:
Fastest but uses extra space (linked list)
Open Addressing:
Linear probing: Primary probe sequence
Quadratic probing: Secondary probe sequence
Probing sequence det'd by second hash fn. \(-g(x)\)

\(h(x)\) = \(h(y)\) \(\rightarrow\) Probes sequences are entirely different!

Analysis: 
Defns:
- \(S_{DH}\) = Expected search time of double hashing if successful
- \(U_{DH}\) = Exp. if unsuccessful
Recall: Load factor \(\lambda = n/m\)

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

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 \(\rightarrow\) duplicate key error!)

Delete \((x)\): Apply \(\text{find}(x)\)
- Not found \(\Rightarrow\) error
- Found \(\Rightarrow\) set to "empty"

Find \((x)\):
Visit entries on probe sequence until:
- Found \(\Rightarrow\) return \(v\)
- Hit empty \(\Rightarrow\) return null

Problem:
Insert \((a)\):
\(h(a)\)
\(\rightarrow\) delete \((a)\):
\(h(u)\)

Why does bust up clusters?
Even if \(h(x) = h(y)\) [collision], it is very unlikely that 
\(g(x) = g(y)\)

\(\rightarrow\) Probes sequences are entirely different!
Announcements: 4/20
- Proj Assign 2, Part 2 -
  Extended deadline - Fri, Apr 21
- Midterm 2 - solutions posted
  (still grading)
- Proj Assign 3 - Soon

Fixes:
\( (h(x) + j^2) \mod m \)

1. \( (h(x) \pm j^2) \mod m \)
2. \( (h(x) \pm j(j+2)) \mod m \)

This implies that if table size = 9
quad. probing will visit only 4 entries

\( \{0, 1, 4, 7\} \)

E.g. \( m = 9 \)
\( \{0^2 = 0, 1^2 = 1, 2^2 = 4, 3^2 = 9 = 0, 4^2 = 16 = 7, 5^2 = 25 = 7, 6^2 = 36 = 0, 7^2 = 49 = 4\} \)

Given int \( m \), its quad. residues are\( \{ q \mid q \equiv x^2 \mod m \} \)

This includes:
- 0
- 1
- 4
- 7

Nothing else
Number of quadratic residues of $n$

$H_i \leq \frac{m}{2}$ entries for any $m$

$y = \frac{n}{2}$
Brent's Algorithm

- Improves Double Hashing
- Even for $\lambda = 1$
  
  expected time $= O(1)$

$h(x)$

$+g(x)$

$+g(x)$

$+g(y)$

$h(y)$