Admin: -> Quiz 1 on Thursday!
    -> Practice on past quizzes + PSet 3 (Problems 1-4)
    -> Review recitations tomorrow
        ("Normal" recitations on Friday)
    -> NO LECTURE on Thursday

Today: -> Open addressing for collision resolution
    -> Double hashing
    -> Cryptographic hashing

Reading:
    CLRS ch 11.4
    Wikipedia: Cryptographic hash function

Last time: Hashing with chaining:

n "active" keys

h: U \rightarrow [0...m-1] (hash function)

Colliding items (chain)

Load factor:
\alpha = \frac{m}{m}

Expected cost of insert/search/delete:
\Theta(1+\alpha) = O(1)

(can maintain via table doubling) if m = \Theta(n)
Disadvantages of chaining:

- Hash table is used just for indexing, all storage in linked lists
- Have to use pointers (slow)
- Bad cache locality (memory performs better if we mostly access nearby memory locations)

Open addressing:

- No linked lists; all the data stored directly in the array
- Each cell is either NULL (empty) or an element

= Need \( n \leq m \), i.e. \( \alpha \leq 1 \)

= Table doubling is necessary (not so much when using chaining)

= Typically, want \( \alpha \leq \frac{1}{2} \)

How to resolve collisions?

- Hash function specifies a sequence of possible locations (probes):
  \[ h: 2^l \times [0..m-1] \rightarrow [0..m-1] \]

= Gives a probe sequence: \( h(k,0), h(k,1), \ldots, h(k,m-1) \)

Ideally, want this to be a permutation of \([0..m-1]\).
How to do search?

- Examine successful probes \( h(k,0) \) \( h(k,1) \) \( h(k,2) \) ... until either find the right element or \( NULL \) (\( \Rightarrow \) "Not in the set")

- Pseudocode:

  ```
  Search \( (T, k) \):
  
  \[
  \begin{align*}
  i &= 0 \\
  j &= h(k, i) \\
  \text{if } T[j] &= k : \text{return } j \\
  i &= i + 1 \\
  \text{until } T[j] &= NULL \text{ or } i = m \\
  \text{return } NULL
  \end{align*}
  \]
  ```

  Probes for \( k_3 \): 2, 0, 3 ...

  Key found!

- Running time:

  \( O(\# \text{ of probes used}) \)

How to do insertions?

- Similarly to search: Examine successive probes until \( NULL \) is found & insert there

- Pseudocode:

  ```
  Insert \( (T, k) \):
  
  \[
  \begin{align*}
  i &= 0 \\
  j &= h(k, i) \\
  \text{if } T[j] &= NULL: \\
  \quad T[j] &= k \\
  \quad \text{return } SUCCESS \\
  i &= i + 1 \\
  \text{until } i = m \\
  \text{return } FAIL
  \end{align*}
  \]
  ```

  Probes for \( k_4 \): 3, 0, 2, 4 ...

  Empty cell found!
How to do deletions?

→ More tricky: Finding the key and setting to NULL will NOT work as it can mislead future searches for other keys.

Example: Search \((T, k_4)\) after removing \(k_2\) \((T[k_2] = \text{null})\)

Probes for \(k_4\): \(3, 0, 2, 4, \ldots\)

Search concludes "Not found" even though \(k_4\) in \(T\)!

→ Need to be more careful (e.g., use DELETED marker)

→ Details: PSet 4

How to construct \(h(k, i)\)?

- Linear probing:
  \[
  h(k, i) = (h'(k) + i) \mod m
  \]

→ Think: street parking

→ Problem: Creates "clusters", i.e., sequences of full cells:
  → big clusters are hit by many new items
  → get put at the end of the cluster (growing it)
  → "rich get richer" phenomenon

→ Can show: for \(0.1 < \alpha < 0.99\) ⇒ cluster sizes \(\Theta(\log n)\)

→ Exp. # of probes needed \(\Theta(\log n) \leq \text{not bad!} \) 😞
Double hashing:

\[ h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m \]

- Here:
  - \( h_1(k) \) maps "randomly" to \([0...m-1]\)
  - \( h_2(k) \) maps "randomly" to \([1...m-1]\) (\(\neq 0\))

- \( m \) is prime (so \( \gcd(m, h_2(k)) = 1 \))
  - \( h(k,0), h(k,1), ..., h(k,m-1) \)
    - is a permutation of \([0...m-1]\)

How to analyze performance of double hashing?

**Uniform hashing assumption (UHA):** (≠ Simple Uniform hashing assumption)

- Probe sequence \( h(k,0), h(k,1), ..., h(k,m-1) \)
  - is a uniformly random permutation of \([0...m-1]\)

- A property of an idealized probe sequence

- In practice: double hashing gives good approx. of UHA
  - (this is a bit different to what uses the case for SUHA)
Analysis (under UHIA):

- Run time of insert (run time of an unsuccessful search) is equal to \# of probes needed until NULL cell is found.

- Let us fix a key \( k \) and define:

\[
p_i = \text{prob. that the probe } h(k,i) \text{ found NULL provided } h(k,0) \ldots h(k,i-1) \text{ did not}
\]

- By UHIA:

\[
p_0 = \frac{m-m}{m} = 1 - \alpha
\]

\# of NULL cells \# of all cells

\[
p_1 = \frac{m-m}{m-1} \geq p_0
\]

\# of NULL cells still the same \# of possible cells to probe after 1 unsuccessful probe

(we are using here the fact that, by UHIA, probe sequence is a permutation \( \Rightarrow h(k,0), \ldots, h(k,m-1) \) all distinct)

- In general:

\[
p_i = \frac{m-m}{m-i} \geq p_0 = 1 - \alpha
\]
Exp. # of probes until NULL found (exp. runtime of insert)

\[
\sum_{i=0}^{m-1} \frac{1}{i+1} \sum_{j=0}^{i-1} (1-p_i) \leq \sum_{i=1}^{m-1} \frac{i}{i+1} p_i (1-p_i)^i \approx \frac{1}{p_0} = \frac{1}{\alpha - 1}
\]

Think: Exp. # of Bernoulli trials until first “success” if prob. of a success is \( p_0 \) each time

Exp. runtime of insert/search is

\[ O \left( \frac{1}{1-\alpha} \right) \]

As long as \( \alpha \leq \frac{1}{2} \) this is \( O(1) \)!
Cryptographic hashing:

- A fundamental application of **hashing function** (but **not hash tables**)

- Think of hash function $h$ as a "random" mapping of strings to $\{0,1\}^d$ (e.g., $d=256$ for hash function SHA-256)

- "random" as in "unpredictable", not "load balancing"

Desirable properties:

- **Collision resistance**: (This does NOT mean that there is no collisions!)
  
  No one should ever be able to **find a collision**:

  a pair $x, x'$ of inputs s.t. $x \neq x'$ but $h(x) = h(x')$

  
  => Hash value of such function serves as a good "digest", "representative", or "proxy" for $x$

Applications:

- **Modification detection**:
  
  If $x$ is a file, save $h(x)$ securely.
  
  If $x$ is changed, then $h(x)$ will change.

  => Can detect file modification (e.g., due to malware)

- **Digital signatures**:
  
  Alice signs $h(x)$ rather than $x$.

  This is good enough, since Alice can't later claim to have signed $x' \neq x$. 
One-wayness:

It should be impossible to "invert" hash function $h$.
That is, given $h(x)$ for some randomly chosen $x$,
to find any $x'$ s.t. $h(x') = h(x)$

Application:

→ **Password storage**:

  Store $h(password)$ instead of password in plain text.

  → Hard for anyone to learn your password even if they hack the computer!

  → Can still log you in by computing $h(password)$ and comparing with what is stored

  → Of course, "inverting" $h$ easy if password short (or otherwise weak)

Important: All the hash functions we have seen are **NOT** good for cryptography

→ Constructing good cryptographic hash functions is difficult. (E.g. MD5 no longer collision resistant)