Quiz 1 Wed 10/16 in 32-124, 32-144, 34-101 7:30 pm
May bring one 8½" x 11" cheat sheet, o.w. closed book
Coverage through L10 (Hashing III)

Pset #3 out today; pset #2 due.

Readings: Chap 10; Chap 11 (sections 1-3)

- Dictionaries & Sets (math & python)
- Applications
- Key Ideas
- Collision resolution by chaining
- Computing hash fn's; universal hash fn's; analysis
Dictionaries & Sets

We often need sets & mappings in an algorithm.
These are closely related, and commonly implemented using hash tables, a very efficient data structure.

Abstract Data Types (Sets)

We begin by describing the interface ("API") we would for an implementation of (dynamic) sets. These are dynamic because the set can be modified.

Operations:

1. **Create an empty set:**
   
   \[ S \leftarrow \emptyset \]

2. **Insert an element \( x \) into set \( S \):**
   
   \[ S \leftarrow S \cup \{ x \} \]

3. **Delete an element \( x \) from set \( S \):**
   
   \[ S \leftarrow S - \{ x \} \]

4. **Membership: Determine if \( x \) is in \( S \):**
   
   \[ x \in S \]
Note: not necessarily in sorted order; output order is arbitrary...

5. **Enumerate**: list the elements in $S$

   $S \rightarrow$ list of its elements

Other operations, such as union & intersection, can be implemented from these primitives.

**Python syntax**

Python provides language support for sets:

1. $S = \text{set}()$
2. $S.add(x)$
3. $S.remove(x)$
4. $x \in S$ (also: $x$ not in $S$)
5. list($S$) (also: for $x$ in $S$: ...)

(Also: $S.union(T)$

   $S.difference(T)$

   ...

)}
Abstract Data Types (Dictionaries)

A dictionary (as an ADT, not a Python type) is very similar: it maintains a set of keys; for each such key it also has a value. *

Operations:

1. Create an empty dictionary:
   \[ D \leftarrow \emptyset \]

2. Insert a key-value pair:
   \[ D \leftarrow D \cup \{ (\text{key}, \text{value}) \} \]
   [removes any earlier pair with same key]

3. Delete any values for given key:
   remove any pair having given key from \( D \)

4. Find value for given key (search):
   return \( v \) s.t. \( (\text{key}, v) \in D \)

5. Enumerate keys in \( D \) (or values in \( D \), or pairs)
   \[ D \Rightarrow \text{list of keys in } D \]

* Think of dictionary as a set of (key, value) pairs...
**Python syntax:**

1. \( D = \{ \} \) (or \( D = \text{dict}() \))
2. \( D[\text{key}] = \text{value} \)
3. \( \text{del} \ D[\text{key}] \)
4. \( D[\text{key}] \) (or \( D.\text{get}(\text{key}, \text{default value}) \))
5. \( D.\text{keys}() \)
   - \( D.\text{values}() \)
   - \( D.\text{items}() \) [returns list of (key, value) pairs]

→ A good implementation of sets can be extended to provide a good implementation of dictionaries: just carry along value with each key, as "satellite data". So we'll focus on sets...

**How to implement dynamic sets efficiently?**

We'd like time \( O(1) \) for each basic operation (1) ... (4).

**BSTs only give us** \( \Theta(\text{log}(n)) \), because they (unnecessarily) maintain set in sorted order.
Applications:

- our “document distance” problem (map words to counts)
- English database (map words to defns)
- OS (map usernames to account into)
- Compiler (map variable names to into e.g. type)
- built in to most programming languages now!

Also:
- substring search (e.g. grep)
- file system synchronization (rsynch)
- crypto (but need stronger properties for “crypto hash”)
- Amazon’s “chaotic storage” system in a warehouse
Key ideas:

1. Treat keys as integers
2. Use key as index into table (direct lookup!)
   (or use hash function to map keys into smaller
   range if otherwise table would be too big)
3. Use some "collision resolution" method if
   two or more keys want same table position (a
   "collision").

1. Treat key as integer ("pre-hash")

   e.g. for strings

   "cat" \(\iff\) "c" \(\cdot\) 256 \(^2\) + "a" \(\cdot\) 256 + "t"

   (base 256) \(\iff\) 99 \(\cdot\) 65536 + 97 \(\cdot\) 256 + 116

   \(\iff\) 6503312

   Python "hash" function provides integer value
   for any (immutable) type. Really a "pre-hash"...

In theory:  \(x = y \iff\) hash\( (x) =\) hash\( (y)\)
2. Use table (Python list)

notation:

\[ m = \text{size of table} \]

\[ T[0..m-1] = \text{table} \]

\[ n = \# \text{ of keys (set elements)} \]

\[ \mathcal{U} = \text{universe of possible keys} \]

\[ \mathcal{K} = \text{set of actual keys stored} \]

\[ \mathcal{K} \subseteq \mathcal{U} \]

\[ |\mathcal{K}| = n \]

Let \( h = \text{“hash function” mapping } \mathcal{U} \rightarrow \{0, 1, \ldots, m-1\} \)

Store key \( k \) in table position \( T[h(k)] \)

Two key questions:

- How to compute \( h \)? (good hash fns?)
- How to resolve collisions? (What if \( h(x) = h(y) \)?)
Quick answers:

- Think of $h$ as (pseudo)-random; $h(x)$ is equally likely to be any of 0, 1, ..., $m-1$, independent of $h$ for other values. Of course, $h$ is deterministic. More later...

- Let $T[i]$ be list of keys whose hash value $h(k) = i$.

```
\[ T = \{ k_1, k_2, ..., k_6 \} \quad n = 6 \]
\[ h(k_1) = h(k_2) = 1 \quad \text{collision} \]
\[ h(k_3) = 4 \]
\[ h(k_5) = h(k_6) = h(k_7) = 6 \quad \text{3-way collision} \]
```

"list" ≡ "chain"

- "collision resolution by chaining" (We'll see other collision resolution methods later..."
**Operations:**

**Insert** \((T, k)\):
Insert \(k\) on list \(T[h(k)]\)

**Delete** \((T, k)\):
Delete \(k\) from list \(T[h(k)]\)

\(k\) in \(T_i\):
Search to see if \(k\) is in list \(T[h(k)]\)

Let \(n_i = \text{length of list } T[i]\)

\((\text{so } \sum_{i=0}^{m} n_i = n)\)

Then running time is \(\mathcal{O}(\max_{i} n_i)\) (worst case)

(length of longest list) [could be \(\Theta(n)\) \(\square\)]

We'll switch to average cost of operation.
We'll keep \(n\) and \(m\) comparable \(m = \Theta(n)\)
(with small constants).
We'll let \(\alpha = n/m = "\text{load factor}"\)

\(= \text{average length of list}\)

\(\Rightarrow\) running times are \(\Theta(1+\alpha)\) on average...

\(= \Theta(1)\) if \(m = \Theta(n)\)
How to compute a hash function?

Assume $k$ is a non-negative integer (pre-hashing done).

1. Division method:
   $$h(k) = k \mod m$$

2. Multiplication method $(m = 2^r)$
   $$h(k) = (A \cdot k) \mod 2^w \gg (w-r)$$
   \(\gg\) = right shift

   $w =$ word size

   $\gg$ = high order $r$ bits of low-order $w$-bit word of product $A \cdot k$

   $A =$ magic constant

3. Universal hashing (example)
   $$h(k) = [(a \cdot k + b) \mod p] \mod m$$

   $a$, $b$ random (chosen at setup time)
   $0 \leq a, b < p$ $a \neq 0$

   $p$ large prime

   $$\text{Prob}_{a,b} (h(k_1) = h(k_2)) = \frac{1}{m} \quad k_1 \neq k_2$$
Expected time to do search with universal hash fn

= expected time to look up a given key k

= \( O(1 + \text{expected # keys in k's list}) \)

= \( O(1 + \text{expected # keys that collide with k}) \)

= \( O\left(1 + \frac{m-1}{m}\right) \)

= \( O(1 + \alpha) \) \hspace{1cm} \alpha = \frac{n}{m} = \text{load factor}

= \( O(1) \) if \( m = \Theta(n) \)