Today: Hashing I
- Dictionaries & Python
- Motivation
- Prehashing
- Hashing
- Chaining
- Simple uniform hashing
- "Good" hash functions
Dictionary problem: Abstract Data Type (ADT) maintain set of items, each with a key, subject to
- `insert(item)`: add item to set
- `delete(item)`: remove item from set
- `search(key)`: return item with key if it exists

- Assume items have distinct keys (or that inserting new one clobbers old)
- Balanced BSTs solve in $O(\log n)$ time per op. (in addition to inexact searches like next-largest)
- Goal: $O(1)$ time per operation

**Python dictionaries:** Items are `(key, value)` pairs

- `{ 'algorithms': 5, 'cool': 42 }`
- `d.items()` -> `[('algorithms', 5), ('cool', 5)]`
- `d['cool']` -> 42
- `d[42]` -> `KeyError`
- `'cool' in d` -> `True`
- `42 in d` -> `False`

- Python set is really dict where items are keys (no values)
Motivation: dictionaries are perhaps the most popular data structure in CS
- built into most modern programming languages (Python, Perl, Ruby, JavaScript, Java, C++, C#, ...)
- e.g. best doc/dist cache: word counts & inner prod.
- implement databases: (DB-HASH in BerkeleyDB)
  - English word → definition (literal dict.)
  - English words: for spelling correction
  - word → all webpages containing that word
  - username → account object
- compilers & interpreters: names → variables
- network routers: IP address → wire
- network server: port number → socket/app.
- virtual memory: virtual address → physical
less obvious, using hashing techniques:
- substring search (grep, Google) [L9]
- string commonalities (DNA) [PS4]
- file/directory synchronization (rsync)
- cryptography: file transfer & identification [L10]
How do we solve the dictionary problem?

**Simple approach:** Direct-access table
- Store items in an array, indexed by key *(random access)*
- Problems:
  1. Keys must be nonnegative integers *(or, using two arrays, integers)*
  2. Large key range $\Rightarrow$ large space
     e.g. one key of $2^{256}$ is bad news

**Solution to 1:** “prehash” keys to integers
- In theory: possible because keys are finite $\Rightarrow$ set of keys is countable
- In Python: `hash(object)` where
  - Mismomer should be “prehash” object is a number, string, tuple, etc., or object implementing `__hash__`
    - Default = `id = memory address`
- In theory: $x = y \iff \text{hash}(x) = \text{hash}(y)$
- Python applies some heuristics for practicality
e.g. $\text{hash}('\phi B') = 64 = \text{hash}('\phi \phi C')$
- Object’s key should not change while in table
  (else can’t find it anymore)
- No mutable objects like lists
Solution to Q: hashing

- reduce universe \( U \) of all keys (say, integers) down to reasonable size \( m \) for table

- idea: \( m \approx n = \# \) keys stored in dictionary

- hash function \( h: U \rightarrow \{0, 1, \ldots, m-1\} \)

- two keys \( k_i, k_j \) collide if \( h(k_i) = h(k_j) \)

How do we deal with collisions? we'll see two ways

- chaining: TODAY
- open addressing: L10

Chaining: linked list of colliding elements in each slot of table

- search must go through whole list \( T[h(key)] \)
- worst case: all \( n \) keys hash to same slot

\( \Rightarrow \Theta(n) \) per operation
Simple uniform hashing: an assumption: (cheating)
each key is equally likely to be hashed
to any slot of table, independent of
where other keys are hashed
- let \( n = \# \text{keys stored in table} \)
  \( m = \# \text{slots in table} \)
- load factor \( \alpha = \frac{n}{m} \)
  = expected \# keys per slot
  = expected length of a chain
  \( \Rightarrow \) expected running time for search
  = \( \Theta(1+\alpha) \)
  \( \Rightarrow \) search the list
  \( \Rightarrow \) apply hash function
  \& random access to slot

= \( O(1) \) if \( \alpha = O(1) \) i.e. \( m = \Omega(n) \)
Hash functions to achieve this performance:

- **division method:** \( h(k) = k \mod m \)
  - practical when \( m \) is prime
  - but not close to power of 2 or 10
    (then just depending on low bits/digits)

- **multiplication method:**
  \[
  h(k) = [(a \cdot k) \mod 2^w] \gg (w-r)
  \]
  \( \text{random } \ll \text{ low } w \text{ bits} \)
  \( \Rightarrow m = 2^r \)
  - practical when
    \( a \) is odd &
  \( 2^{w-1} \leq a < 2^w \)
  - not too close
  - fast

- **universal hashing:** [6.046; CLRS 11.3.3]
  \[
  e.g. \quad h(k) = [(ak+b) \mod p] \mod m
  \]
  \( \Rightarrow \text{large prime (}>191) \)
  \( \Rightarrow \text{for worst-case keys } k_1 \neq k_2: \)

\[\Pr_{a,b} \{ h(k_1) = h(k_2) \} = \frac{1}{m} \]  
\( \text{lemma ~ not proved here} \)

\[ \Rightarrow E[\# \text{ collisions with } k_1] = E[ \sum_{k_2} X_{k_1k_2} ] \]
\[ = \sum_{k_2} E[X_{k_1k_2}] \]
\[ = \sum_{k_2} \Pr\{X_{k_1k_2} = 1\} \]
\[ = \frac{n}{m} = \alpha \]

just as good as above!