6.006	Lecture 8	Oct. 4. 2011

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 $O(\lg n)$ time per op. (in addition to inexact searches like next-largest)
- goal: $O(1)$ time per operation

Python dictionaries: items are (key, value) pairs

e.g. d = {'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)

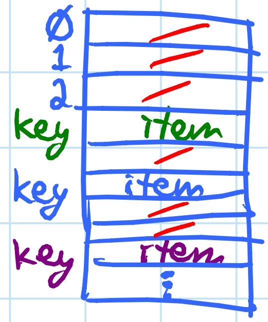
<u>Motivation:</u> dictionaries are perhaps the most popular data structure in CS

- built into most modern programming languages (Python, Perl, Ruby, JavaScript, Java, C++, C#, ...)
- e.g. best docdist code: word counts & inner prod.
- implement databases: (DB_HASH in Berkeley DB)
  - 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 ⇒ large space
     e.g. one key of 2^{256} is bad news

Solution to ①: "prehash" keys to integers

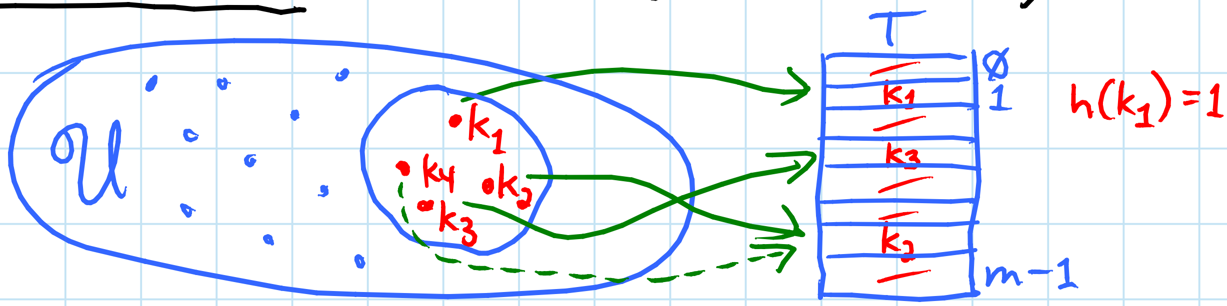
- in theory: possible because keys are finite
  ⇒ set of keys is countable
- in Python: hash(object) where
  ↳ misnomer ~ should be "prehash"

  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. hash('\ØB') = 64 = hash('\Ø\ØC')
- object's key should not change while in table
  (else can't find it anymore)
  - no mutable objects like lists

& Old High German 'happja' = scythe

Solution to ②: hashing (verb from French 'hache' = hatchet)

- reduce universe $\mathcal{U}$ of all keys (say, integers) down to reasonable size m for table
- idea: $m \approx n = \#$ keys stored in dictionary
- hash function $h: \mathcal{U} \to \{0, 1, \ldots, m-1\}$

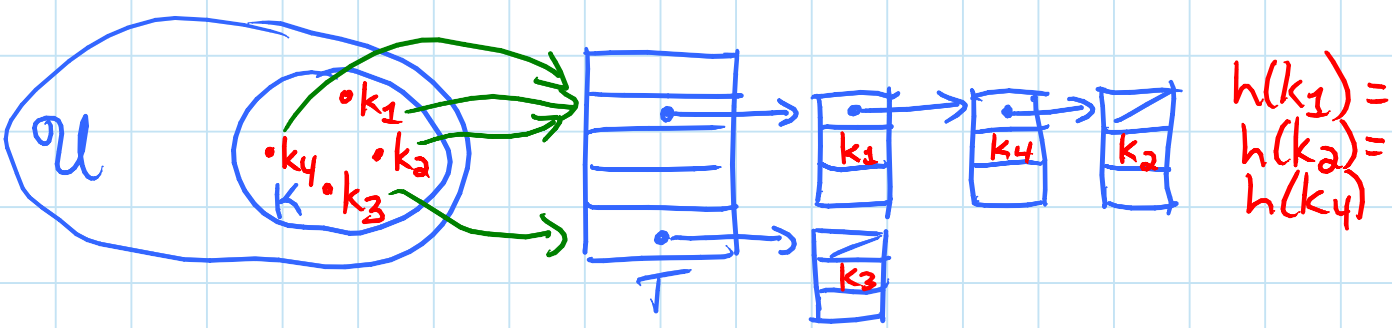


- two keys k_i, k_j collide if $h(k_i) = h(k_j)$

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 = # keys stored in table
  m = # slots in table
- load factor $\alpha = n/m$
  = expected # keys per slot
  = expected length of a chain

$\Rightarrow$ expected running time for search

$= \Theta(1+\alpha)$

↳ search the list

↳ apply hash function & random access to slot

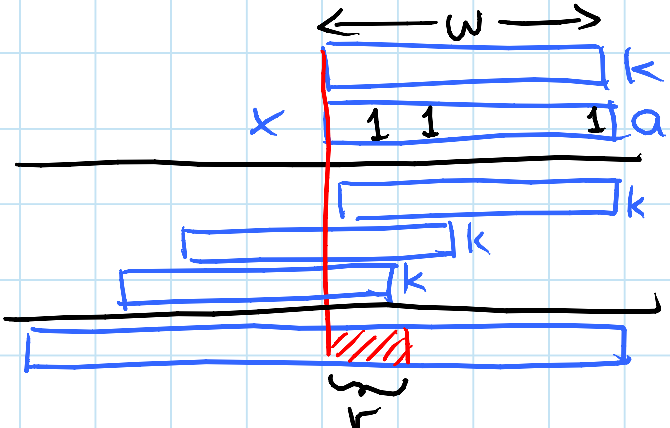
$= O(1)$ if $\alpha = O(1)$ i.e. $m = \Omega(n)$

Hash functions to achieve this performance:

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

$$h(k) = [(\underbrace{a}_{\text{random}} \cdot k) \bmod \underbrace{2^w}_{w \text{ bits}}] \gg (w - \underbrace{r}_{m = 2^r})$$

  - practical when a is odd & $2^{w-1} < a < 2^w$ & not too close
  - fast



- **universal hashing:** [6.046: CLRS 11.3.3]

e.g. $h(k) = [(ak + b) \bmod p] \bmod m$

random $a, b \in \{0, 1, \ldots, p-1\}$; p large prime ($> |\mathcal{U}|$)

$\Rightarrow$ for worst-case keys $k_1 \neq k_2$:

$$\Pr_{a,b}\{\underbrace{h(k_1) = h(k_2)}_{\text{event } X_{k_1 k_2}}\} = 1/m$$

lemma ~ not proved here

a, b: choice of h

$$\Rightarrow \mathop{E}_{a,b}[\#\text{ collisions with } k_1] = E\left[\sum_{k_2} X_{k_1 k_2}\right]$$
$$= \sum_{k_2} E[X_{k_1 k_2}]$$
$$= \sum_{k_2} \underbrace{\Pr\{X_{k_1 k_2} = 1\}}_{1/m}$$
$$= n/m = \alpha$$

just as good as above!