Lecture 8: Hashing I

Lecture Overview

- Dictionaries and Python
- Motivation
- Prehashing
- Hashing
- Chaining
- Simple uniform hashing
- "Good" hash functions

Dictionary Problem

Abstract Data Type (ADT) — maintain a 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

We 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()	$\rightarrow$	[('algorithms', 5),('cool',5)]
d['cool']	$\rightarrow$	42
d[42]	$\rightarrow$	KeyError
'cool' in d	$\rightarrow$	True
42 in d	$\rightarrow$	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 docdist code: word counts & inner product
- 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 or directory synchronization (rsync)
- cryptography: file transfer & identification [L10]

How do we solve the dictionary problem?

Simple Approach: Direct Access Table

This means items would need to be stored in an array, indexed by key (random access)

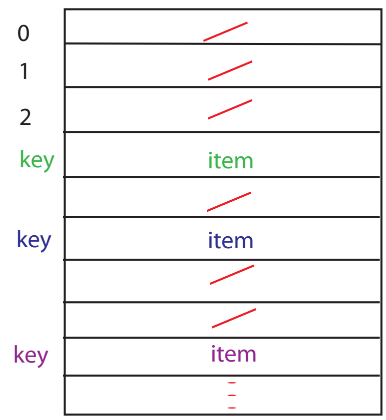

Figure 1: Direct-access table

Problems:

1. keys must be nonnegative integers (or using two arrays, integers)
2. large key range $\implies$ large space — e.g. one key of 2^{256} is bad news.

2 Solutions:

Solution to 1: "prehash" keys to integers.

- In theory, possible because keys are finite $\implies$ set of keys is countable
- In Python: hash(object) (actually hash is misnomer should be "prehash") where object is a number, string, tuple, etc. or object implementing __hash__ (default = id = memory address)
- In theory, $x = y \Leftrightarrow$ hash(x) = hash(y)
- Python applies some heuristics for practicality: for example, hash('$\backslash 0B$ ') = 64 = hash('$\backslash 0 \backslash 0C$')
- Object's key should not change while in table (else cannot find it anymore)
- No mutable objects like lists

Solution to 2: hashing (verb from French 'hache' = hatchet, & Old High German 'happja' = scythe)

- 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: $\mathcal{U} \rightarrow \{0, 1, \ldots, m-1\}$

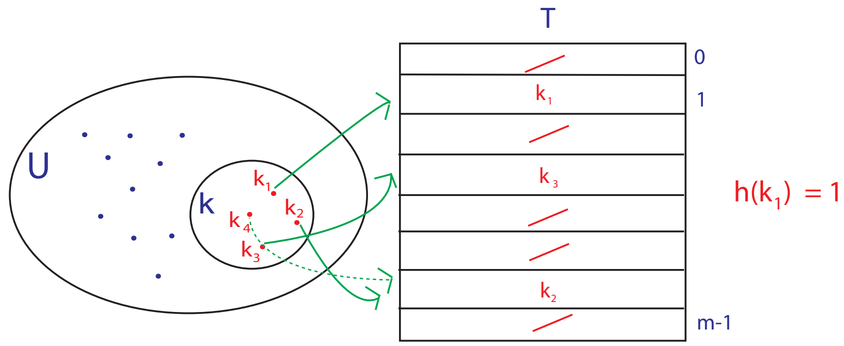

Figure 2: Mapping keys to a table

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

How do we deal with collisions?

We will see two ways

1. Chaining: TODAY
2. Open addressing: L10

Chaining

Linked list of colliding elements in each slot of table

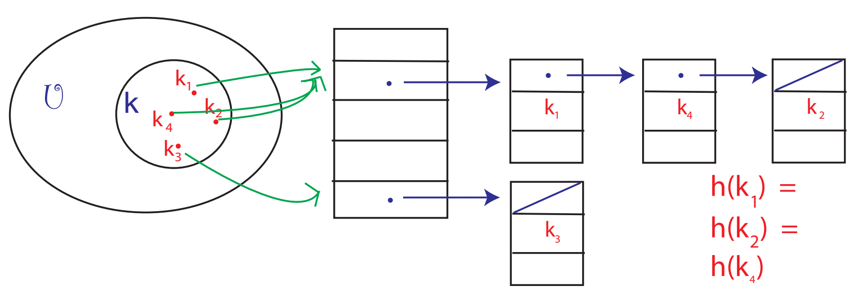

Figure 3: Chaining in a Hash Table

- Search must go through *whole* list T[h(key)]
- Worst case: all n keys hash to same slot $\implies \Theta(n)$ 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.

$$\begin{aligned} \text{let } n &= \#\text{ keys stored in table} \\ m &= \#\text{ slots in table} \\ \underline{\text{load factor}}\, \alpha &= n/m = \text{expected } \#\text{ keys per slot} = \text{expected length of a chain} \end{aligned}$$

Performance

This implies that expected running time for search is $\Theta(1+\alpha)$ — the 1 comes from applying the hash function and random access to the slot whereas the α comes from searching the list. This is equal to $O(1)$ if $\alpha = O(1)$, i.e., $m = \Omega(n)$.

Hash Functions

We cover three methods to achieve the above performance:

Division Method:

$$h(k) = k \bmod m$$

This is practical when m is prime but not too close to power of 2 or 10 (then just depending on low bits/digits).

But it is inconvenient to find a prime number, and division is slow.

Multiplication Method:

$$h(k) = [(a \cdot k) \text{ mod } 2^w] \gg (w - r)$$

where a is random, k is w bits, and $m = 2^r$.
This is practical when a is odd & $2^{w-1} < a < 2^w$ & a not too close to 2^{w-1} or 2^w.

Multiplication and bit extraction are faster than division.

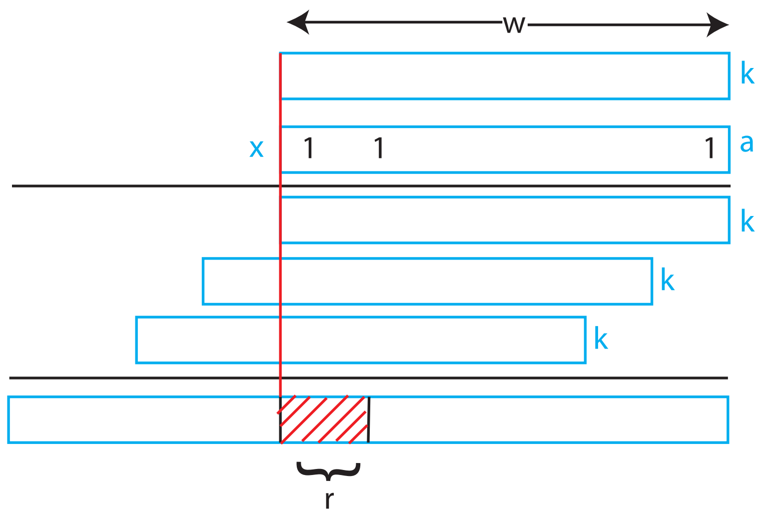

Figure 4: Multiplication Method

Universal Hashing

[6.046; CLRS 11.3.3]

For example: $h(k) = [(ak+b) \mod p] \mod m$ where a and b are random $\in \{0, 1, \ldots p-1\}$, and p is a large prime ($> |\mathcal{U}|$).
This implies that for *worst case* keys $k_1 \neq k_2$, (and for a, b choice of h):

$$Pr_{a,b}\{\text{event } X_{k_1k_2}\} = Pr_{a,b}\{h(k_1) = h(k_2)\} = \frac{1}{m}$$

This lemma not proved here
This implies that:

$$\begin{aligned}
E_{a,b}[\#\text{ collisions with } k_1] &= E[\sum_{k_2} X_{k_1k_2}] \\
&= \sum_{k_2} E[X_{k_1k_2}] \\
&= \sum_{k_2} \underbrace{Pr\{X_{k_1k_2} = 1\}}_{\frac{1}{m}} \\
&= \frac{n}{m} = \alpha
\end{aligned}$$

This is just as good as above!