Today:
- → Dictionaries (ADT)
- → Intro to hashing
- → Collision resolution by chaining
- → Simple uniform hashing
- → "Good" hash functions

Reading:
CLRS Ch. 11.1-11.3

Dictionary [as an 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 exists)

(→ Create(): creates an empty dictionary)

→ Assume that items have distinct keys (or that inserting a new item with the same key overwrites the old one)

Python dictionaries: (aka dict) Items are (key, value) pairs
- → By default: indices but can be anything!

`D[key]` ~ search
`D[key]=val` ~ insert
`del D[key]` ~ delete

Example: `D = {'1': 5, 'love': 42, '6.006': 6}`

`D['love']` → 42
`D[42]` → KeyError
`'6.006' in D` → True
`D.items()` → `[('1',5), ('love',42), ('6.006',6)]`

Python set is really a 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#, ...)

→ implement databases:
- → English word → definition
- → English words: for spelling correction
- → word → all documents containing that word
- → username → account object

→ Compilers & interpreters: names → variables/mem. loc.

→ network routers: IP address → wire

→ network server: port # → socket / app.

→ virtual memory: virtual address → phys. address

→ PSet 1: best docdist code: word counts & inner prod.

Less direct (uses hashing techniques):

→ substring search (grep, Google) [L9]

→ string commonalities (DNA)

→ cryptography: file transfer & identification [L10]

How to implement a dictionary?

Decent solution: Use a balanced BST

-> All operations in $O(\log n)$ time
(n = # of items in the set)

-> Bonus: can do inexact searches too
(e.g., find next-largest)

-> But: $\Omega(\log n)$ worst-case is not good enough in many scenarios (networking, databases)

Note: This is unavoidable as BSTs keep the data (unnecessarily) sorted

Our goal: $O(1)$ time per operation!

Need "secret sauce": Randomness!

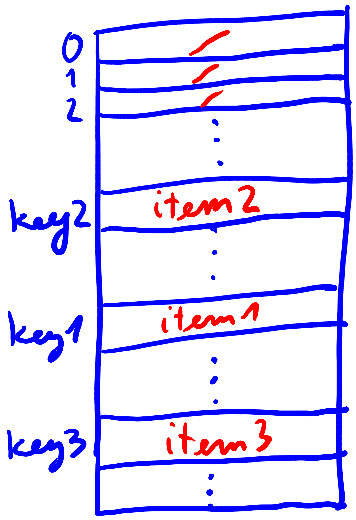
Resulting caveat: Only "in expectation" / "with high probability" $O(1)$ time guarantee

-> In Python: Can assume that all basic dictionary operations are $O(1)$ time

How to implement dictionaries in O(1) time per op.?

Simple approach: Direct-access table

→ Store items in an array indexed by key (similar in spirit to Countsort!)



→ Random access

⇒ All operations in O(1) time!

→ Problems:

① keys must be nonnegative integers

② Large key range ⇒ large space needed
(e.g. one key of 2^{256} is bad news)

Solution to ①: "Rehash" keys to integers, i.e., define a canonical mapping:

object → (non-neg.) integer

In theory: Always possible because the number of objects is countable

(everything can be stored as a string of bits anyway)

In Python: `hash(object)` → (non-neg.) integer (misnomer, should be "prehash")

→ defined for numbers, strings, tuples, etc. or objects implementing `__hash__`
(default = id = memory address)

Ideally, want: $x = y \iff hash(x) = hash(y)$

But: Python applies some heuristics to make hash(·) practical

⇒ Can happen $hash(x) = hash(y)$ & $x \neq y$

E.g., $hash('\phi B') = 64 = hash('\phi\phi C')$

Also: Object's prehash has to stay the same after inserting into table (else can't find it anymore)

⇒ no mutable objects (like lists)

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

Key idea: Reduce the universe $\mathcal{U}$ of all keys down to reasonable size m for table

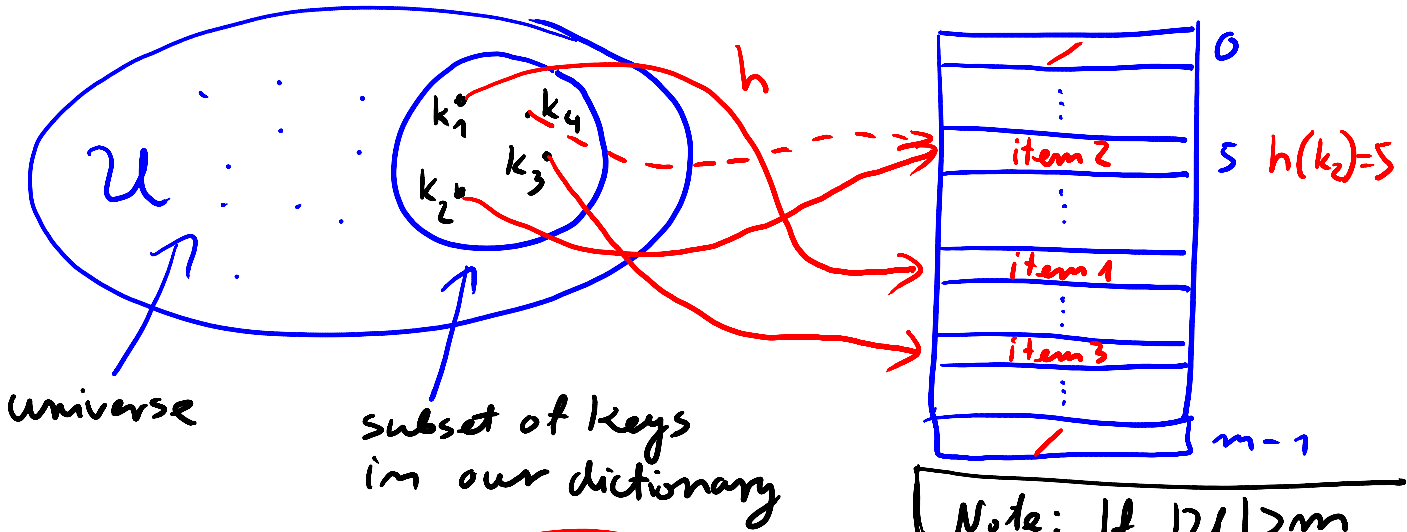
→ Usually, want:

$m \approx n$ = # of items stored

Hash function: $h : \mathcal{U} \to \{0, 1, \ldots, m-1\}$

universe of all keys

indices of cells in our table



Big (and unavoidable!) problem: COLLISIONS

Note: If $|\mathcal{U}| > m$, h has to have some collisions! (see "pigeonhole principle")

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

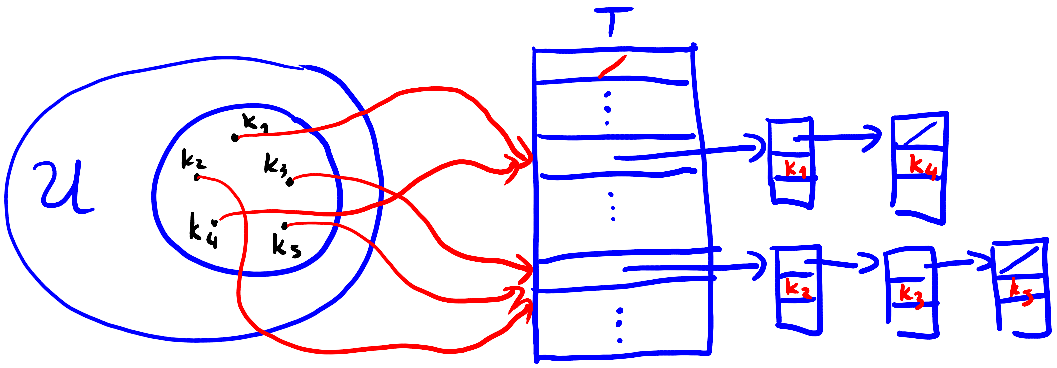
How to deal with collisions?

We will see two techniques:

→ Chaining [Today]

→ open addressing [L10]

Chaining: Keep in each cell of the table a linked list of colliding items



Linked list (aka chain): Simple but horrible for search

⇒ Search must go through the whole list linked at T[h(key)]

Worst case: All n keys hash to the same slot

⇒ $\Theta(n)$ time per operation!

In the worst case, hash tables are really bad!

Crucial idea: Use randomization to try to avoid worst-case

Simple uniform hashing assumption (SUHA):

Each new key is equally likely to be hashed to any slot of the table, independently of where other keys are hashed

Way to think about it:

h is a "black box" that returns a uniformly random cell when queried on a new key, but is consistent with past choices for previously seen keys

"Magic" of randomness:

→ Let n = # of keys stored in the table
m = # of cells in the table

→ Under SUHA:

⇒ Probability a newly inserted key lands in a particular cell = $\frac{1}{m}$

⇒ Expected # of keys per cell is equal to $\boxed{\alpha = \frac{n}{m}}$ ← load factor

⇒ Expected length of a chain is equal to α too!

⇒ Expected running time of search (and insert and delete too) under SUHA:

$$\Theta(1+\alpha)$$

apply the hash function + random access to the cell

searching/updating the chain/list

→ If we keep $m = \Omega(n)$

⇒ Load factor $\alpha = \frac{n}{m} = O(1)$

→ We can execute all dictionary operations in $O(1)$ time!

Important: → This $O(1)$ performance is only in expectation (can show "high probability" too)

⇒ The "$\Omega(n)$ per operation nightmare" can still emerge, it is just unlikely that it does

But: there is no "worst-case set of keys", only "unlucky" random choices

→ Real hash functions do NOT satisfy SUHA

SUHA is just a useful idealization of what we want hash functions to be like

(One can obtain $O(1)$ performance under much weaker assumptions than SUHA - see below)

Some "good" hash functions:

→ Division method:

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

→ Practical when m is a prime
but very bad when m close to power of 2 or 10
(then just depending on low bits/digits)

→ But: there exist set of keys that "break" it, e.g., each key a multiple of m

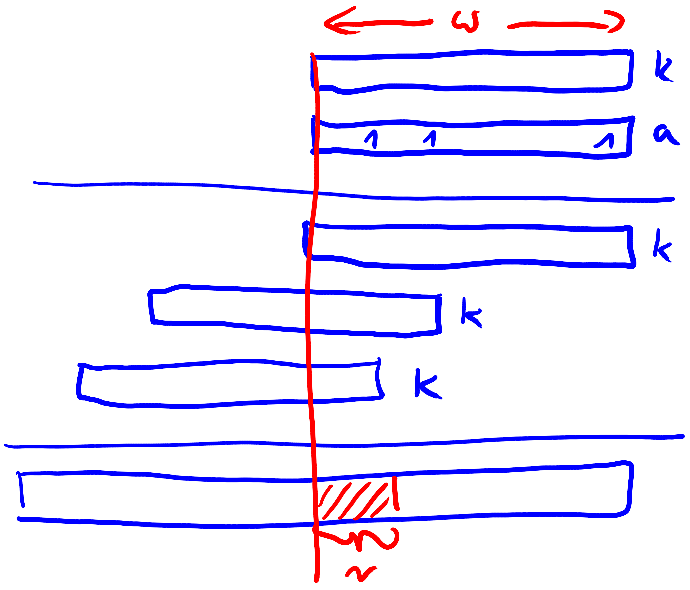
This makes this function "bad" in our "worst-case" sense

→ Multiplication method:

$$h(k) = \lfloor (a \cdot k) \bmod 2^w \rfloor >> (w-r)$$

a: random seed ← | w: w bits ↳ | r: $m = 2^r$ ↳

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



→ fast (uses bit shifts)

→ But (again):
There exist sets of keys that "break" it

→ Universal hashing: [6.046, CLRS 11.3.3]

→ Provably good hashing functions that are also decent in practice (NO set of keys that "breaks" it)

Canonical example:

$$h(k) = [(a \cdot k + b) \bmod p] \bmod m$$

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

→ Can show: for any two keys $k_1 \neq k_2$,

(pair-wise independence)

$$\Pr_{a,b}[h(k_1) = h(k_2)] = \frac{1}{m} \quad (*)$$

a,b: choice of h; $h(k_1) = h(k_2)$: event X_{k_1,k_2}

Note: This is a strictly weaker condition than SUHA

E.g., we could have

$$\Pr[h(k_1) = h(k_2) = h(k_3)] = \frac{1}{m} > \frac{1}{m^2}$$

(This is what SUHA gives)

→ Still, it is all we need for $O(1)$ in expectation bound (but no w/ high prob. bound)

$$\Rightarrow E[\#\text{ of collisions with } k_1] = E\left[\sum_{k_2} X_{k_1,k_2}\right] = \sum_{k_2} E[X_{k_1,k_2}] =$$

$$= \sum_{k_2} \Pr[X_{k_1,k_2} = 1] \overset{\text{by } (*)}{=} \sum_{k_2} \frac{1}{m} = \frac{n}{m} = \alpha \; (= O(1) \text{ for } m = \Omega(n))$$