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: Items are (key, value) pairs
(aka dict)

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

By default: indices but can be anything!

Example: D = { '1': 5, 'love': 42, '6.006': 63 }

D[ 'love' ] -> 42
D[ 42 ] -> KeyError
'6.006' in D -> True
D.items() -> [ ( '1', 5 ), ( 'love', 42 ), ( '6.006', 63 ) ]
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 → route
→ 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) [79]
→ String commonalities (DNA)
→ Cryptography: file transfer & identification[710]
How to implement a dictionary?

Decent solution: Use a **balanced** BST

\[ \Rightarrow \text{All operations in } O(\log n) \text{ time} \]

\[ n = \# \text{ of items in the set} \]

\[ \Rightarrow \text{Bonus: can do inexact searches too (e.g., find next-largest)} \]

\[ \Rightarrow \text{But: } \Omega(\log n) \text{ worst-case is } \]

\[ \not= \text{good enough in many scenarios (networking, databases)} \]

---

**Our goal:** \( O(1) \) time per operation!

Need "secret sauce": **Randomness**!

Resulting caveat: Only "in expectation"/

"with high probability"

\( O(1) \) time guarantee

\[ \Rightarrow \text{In Python: Can assume that all basic dictionary operations are } O(1) \text{ time} \]
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 $\Rightarrow$ 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 $\rightarrow$ (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(obj) -> (non-neg.) integer`
- `defined for numbers, strings, tuples, etc. or objects implementing `--hash--`
- `(default = id = memory address)`

Ideally, we want:

\[ x = y \iff \text{hash}(x) = \text{hash}(y) \]

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

\[ \implies \text{can happen} \quad \text{hash}(x) = \text{hash}(y) \land x \neq y \]

E.g., `hash(\'\phi\beta\') = 64 = \text{hash(\'\phi\phi\epsilon\')}`

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

\[ \implies \text{no mutable objects (like lists)} \]
Solution to ②: Hashing (verb from French 'hache' = hatched)

Key idea: Reduce the universe \( U \) of all keys down to reasonable size \( m \) for table

\[ m \approx n = \# \text{ of items stored} \]

Hash function: \[ h: U \to \{0, 1, ..., m-1\} \]

Big (and unavoidable!) problem:

\[ \text{COLLISIONS} \]

\[ \Rightarrow \text{two keys } k_i, k_j \text{ collide if } h(k_i) = h(k_j) \]
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: Simple but horrible for search (aka chain) ⇒ 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

\[ m = \# \text{ of cells in the table} \]

→ Under SUHA:

\( \implies \) Probability a newly inserted key lands in a particular cell = \( \frac{1}{m} \)

\[ \implies \] Expected # of keys per cell is equal to \[ \alpha = \frac{m}{m} \]

\[ \implies \] Expected length of a chain is equal to \( \alpha \) 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

\[ \Rightarrow \text{If we keep } m = \Omega(n) \]

\[ \Rightarrow \text{Load factor } \alpha = \frac{m}{m} = 0(1) \]

\[ \Rightarrow \text{We can execute all dictionary operations in } O(1) \text{ time!} \]

**Important:**

\[ \Rightarrow \text{This } O(1) \text{ performance is only in expectation (can show "high probability" too)} \]

\[ \Rightarrow \text{The } \Omega(n) \text{ per operation nightmare can still emerge, it is just unlikely that it does} \]

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

\[ \Rightarrow \text{Real hash functions do NOT satisfy SUHA} \]

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...
Some "good" hash functions:

- **Division method:**
  \[ h(k) = k \mod m \]

  - Practical when \( m \) is a prime
    but very bad when \( m \) close to power of 2 or 10
    (then just dependin 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:**
  \[ h(k) = \left\lfloor \frac{(a \cdot k) \mod 2^w}{2^r} \right\rfloor \]

  - Practical when \( a \) is odd &
    \( 2^{w-4} < a < 2^w \)
    & not too close
  - Fast (uses bit shifts)

- But (again):
  There exist sets of keys that "break" it
- Universal hashing: \[ 6.046, \text{CLRS 11.3.3} \]

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

Canonical example:

\[ h(k) = \left( (a \cdot k + b) \mod p \right) \mod m \]

\( a, b \) \text{ random seeds} \in \{0, \ldots, p-1\} \quad (>100) \]

\( p \) \text{ large prime} \quad (>1001) \]

- Can show: for any two keys \( k_1 \neq k_2 \)

\[ P_r[h(k_1) = h(k_2)] = \frac{1}{m} \quad (*) \]

(pair-wise independence)

Note: This is a strictly weaker condition than SUHA

E.g., we could have

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

- Still, it is all we need for \( 0(1) \) in expectation bound

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

\[ = \sum_{k_2} P_r[X_{k_1,k_2} = 1] \text{ by (x)} \]

\[ = \sum_{k_2} \frac{1}{m} = \frac{m}{m} = 1 \quad \leq O(1) \text{ for } m = \Theta(n) \]