

Brief guide to hash tables

What they are

How they are implemented

Hash tables

Data structures for fast **search**, **insertion** & **deletion**

Introduced by **H. P. Luhn** at IBM (1953) 

Complexity (for n data items)

In typical conditions, hash tables **ops** are **fast**:

Avg. case: **search**, **insert**, **delete** $\rightarrow O(1)$ key benefit

Worst case: **search**, **insert**, **delete** $\rightarrow \Theta(n)$

Storage complexity of hash tables: $\Theta(n)$

Suited for **Abstract Data Types**

Set

Map

Abstract Data Type: Set

Collection of **objects** with operations: each **object** x has a key: $x.key$

insert(x) insert object x into the Set

delete(x) remove object x from the Set

search(key) get x if $x.key = key$, else raise exception

Abstract Data Type: Map

Collection of **(key, value)** pairs with operations:

insert(key, value) add (key, value) pair to Map

delete(key) remove (key, value) pair from Map

search(key) get value if key present, else raise exception

Keys must be **unique**
for Sets and Maps

We focus on **Sets** (can achieve Map behaviour with $x.value$ attribute for each object)

References/Notes/Image credits:

H. Stevens, "Hans Peter Luhn and the birth of the hashing algorithm", IEEE spectrum (2018)

(Luhn photo) https://researcher.watson.ibm.com/researcher/view_page.php?id=6990

"Associative Array"/"Dictionary" can replace "Map" https://en.wikipedia.org/wiki/Associative_array

("keys") D. E. Knuth, "The art of computer programming, vol. 3: sorting and searching", p392 (1998)

Idea: search by index

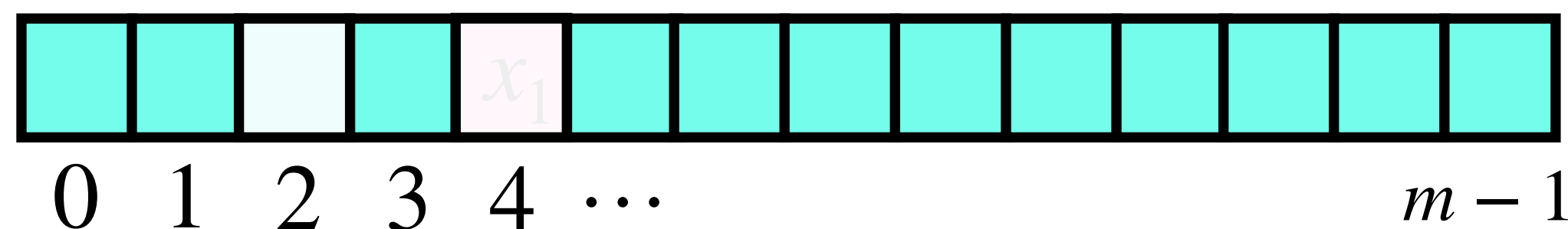
Idea: replace search with **array indexing** $O(1)$

Direct-address table

Suppose the n objects x_0, \dots, x_{n-1} we'll store have **unique integer keys** k_0, \dots, k_{n-1}

$k_i \in \{0, \dots, m-1\}$ Universe, U , is set of possible keys

Build a **big array** with m slots: $U = \{0, \dots, m-1\}$



Unused slots have a value of **None**:

Example operations on x_0 ($k_0 = 2$) x_1 ($k_1 = 4$)

Insert x_0, x_1

delete x_0

search $k = 4$

We **search**, **insert**, **delete** using array in $O(1)$

What happens if $|U| \gg n$? **Lots of wasted space!**

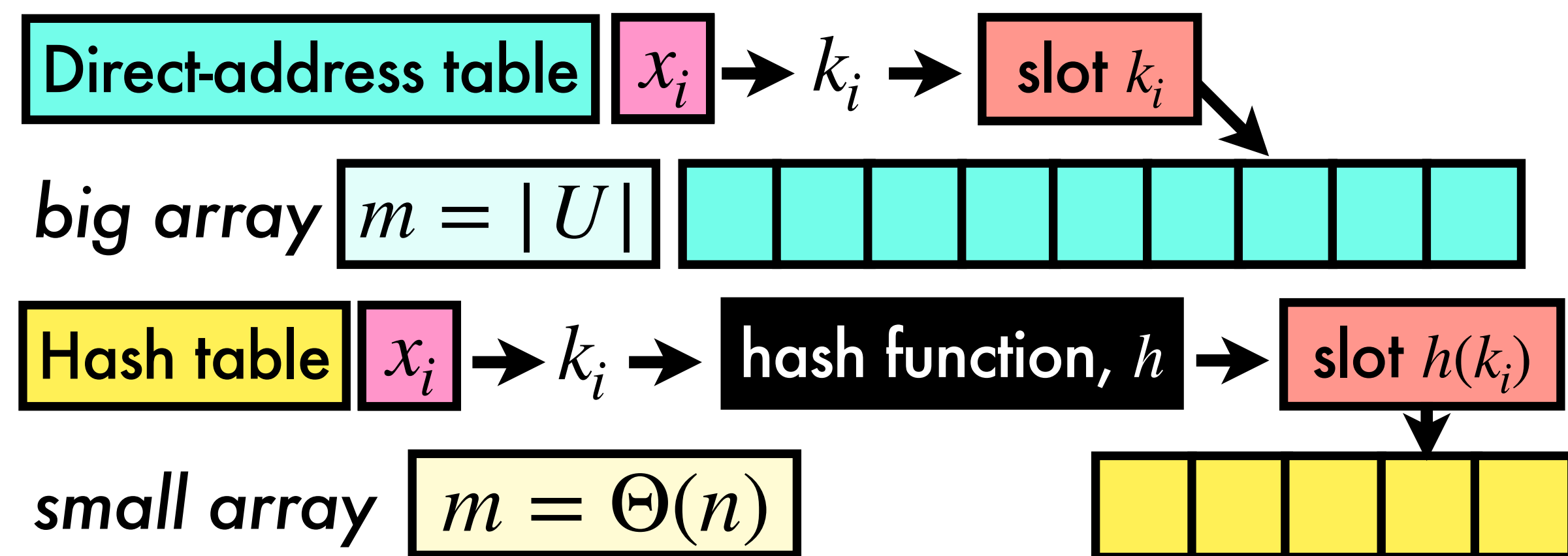
Suppose we want to store 5 **IPv6 addresses**

Our **universe size** is $|U| = 2^{128}$

$> 1\text{K trillion trillion}$ 1TB hard drives! $> 28 \cdot 10^{27}$ GBP

A **hash table** uses a function, h , to compute slots
 $h : U \rightarrow \{0, \dots, m-1\}$ is a **hash function**

Goal: design h to shrink array size ($m = \Theta(n)$)

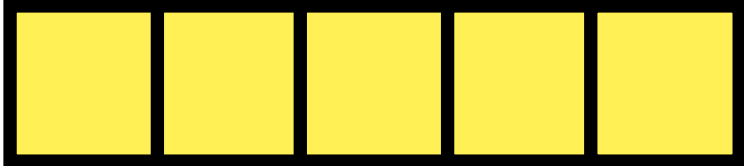


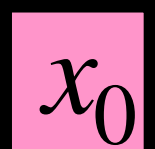
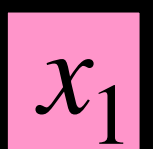
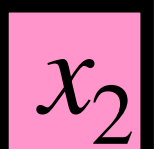
Hash functions

"Input data is not random! So good hash functions must be random!" J. Erickson

Suppose $U \subset \mathbb{Z}$ and our hash table has m slots

A basic **hash function**: $h(k) = k \bmod m$

$m = 5$  **collision!**
0 1 2 3 4

 $(k_0 = 2)$  $(k_1 = 8)$  $(k_2 = 23)$

Two key **requirements** for our hash function:

1. Fast to **compute**

2. Minimise **collisions** $h(k_i) = h(k_j)$ with $k_i \neq k_j$

Ideal $h(k)$ rolls a fair m -sided die for each k :
an **independent uniform random hash function**

How to get **randomness** from **nonrandom data**?

Division method

Static

$h(k) = k \bmod m$
helps (a bit) if m is prime

Multiplication method

Choose $A \in (0,1)$
 $h(k) = \lfloor m \cdot (Ak \bmod 1) \rfloor$

vulnerable to unfavourable key distributions (many collisions)

Cryptographic

Pre-image resistance

Collision resistance

(typically **slower**)

Universal family H :

Random

$$P_{h \in H}[h(k_i) = h(k_j)] \leq \frac{1}{m} \quad \forall i \neq j$$

A universal family

Pick a **prime number** $p > |U|$

$$h_{a,b}(x) \triangleq ((ax + b) \bmod p) \bmod m$$

$$H_{p,m} = \{h_{a,b} \mid a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$$

(a, b are "**salts**") **less vulnerable**

still vulnerable to interactive attacks

Applications

Hash tables

String search

Passwords

Signatures

Digests

Proof-of-work



CPython: **SipHash** (str/byte) **prevent DoS**

References/Notes/Image credits:

(Requirements/randomness) D. E. Knuth, "The art of computer programming, vol. 3: sorting and searching", Chap 6.4 (1998)

(Hash functions) T. Cormen et al., "Introduction to algorithms", Chap 11.3, MIT press, (2022)

J. Erickson, "Algorithms" <http://algorithms.wtf/> "Lecture 5: Hash Tables" (2019)

J. L. Carter et al., "Universal classes of hash functions", ACM STOC (1977)

Following T. Cormen et al., "Introduction to algorithms", Chap 11.3, MIT press, (2022), we use the notation that $\mathbb{Z}_p^* = \{1, \dots, p-1\}$

(Bitcoin logo) https://commons.wikimedia.org/wiki/File:Bitcoin_logo.svg#/media/File:Bitcoin.svg

J-P Aumasson et al., "SipHash: a fast short-input PRF", ICC (2012)

<https://tenthousandmeters.com/blog/python-behind-the-scenes-10-how-python-dictionaries-work/>

C. Heimes, "PEP 456 - Secure and interchangeable hash algorithm", <https://peps.python.org/pep-0456/> (2013)

Chaining

Chaining: a simple way to handle collisions

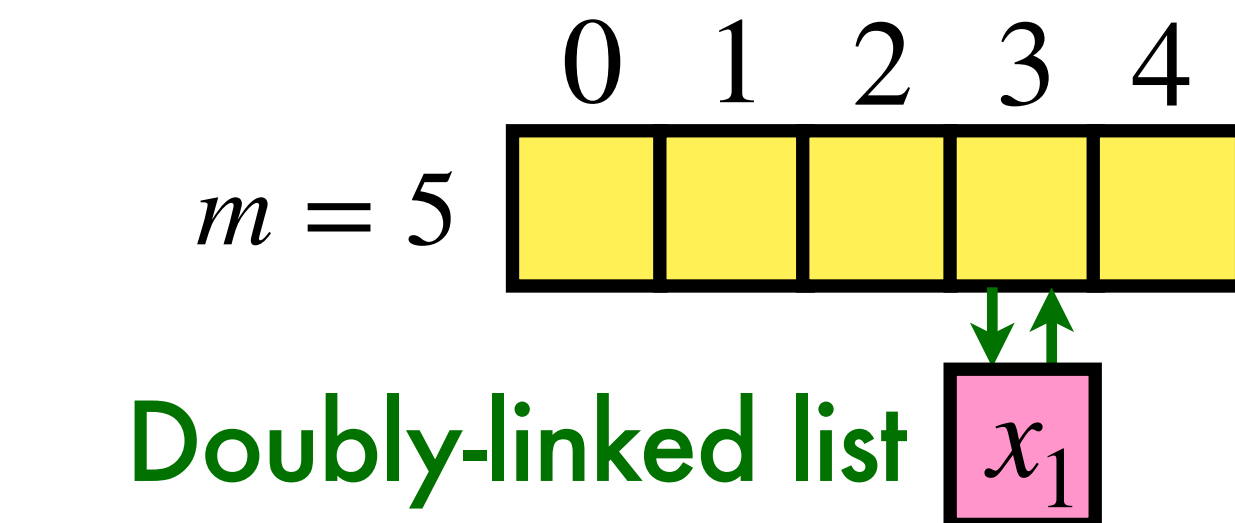
Insert

x_0 ($k_0 = 2$)

x_1 ($k_1 = 8$)

x_2 ($k_2 = 23$)

x_3 ($k_3 = 98$)



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

Search for ($k = 8$)

Delete

x_2

Worst case scenario (for search)

All n keys **collide** \implies all objects in same slot

Search is then $\Theta(n)$ with **linked lists**

Average scenario (cost of unsuccessful search)

Define the **load factor** of table: $\alpha \triangleq \frac{n}{m}$ items / slots

Assume our hash function is **universal**

Collision probability $\leq 1/m$

$$\mathbb{E}(\text{chain length}) = n/m = \alpha$$

Average cost: $\Theta(1 + \alpha)$ (hashing + chain search)

Average cost of successful search

Similarly to **unsuccessful search**: $\Theta(1 + \alpha)$

References/Notes:

See J. Erickson, "Algorithms" <http://algorithms.wtf/> "Lecture 5: Hash Tables" (2019) for a more detailed proof or T. Cormen et al., "Introduction to algorithms" MIT press, Chap 11.2 (2022) for an extended analysis

Open addressing

Open addressing: chain-free collision handling

Coined by William W. Peterson in 1957 

The simplest variant is linear probing:

Insert

x_0 ($k_0 = 2$)

x_1 ($k_1 = 8$)

x_2 ($k_2 = 23$)

x_3 ($k_3 = 98$)

$m = 5$

0	1	2	3	4
x_3				D

primary clustering

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

Search for ($k = 98$)

Delete x_2

Probe sequences

Open addressing schemes

produce permutation of $(0, 1, \dots, m - 1)$

Double hashing: $h(k, i) = (h_1(k) + i h_2(k)) \bmod m$
position in probe sequence

For a permutation, $h_2(k)$ and m must be coprime

Analysis: number of probes in unsuccessful search ($\alpha < 1$)

Assume independent uniform permutation hashing

$$\text{Max probes: } \frac{1}{1 - \alpha} = 1 + \alpha + \alpha^2 + \alpha^3 + \dots$$

at least one more than 1 more than 2 more than 3

Re-ordering schemes

Brent's method

reduce average

Robin Hood

reduce variance

Linear probing is not so bad in practice (caching)

CPython uses pseudorandom probing + heuristics

Maximum load factor of $2/3$ (before resize)

Optimised for: object attribute/method lookups

References/Notes/Image credits:

D. E. Knuth, "The art of computer programming, vol. 3: sorting and searching", Chap 6.4, (1974)

W. W. Peterson, "Addressing for random-access storage." IBM journal of Research and Development (1957)

https://en.wikipedia.org/wiki/W._Wesley_Peterson#/media/File:W._Wesley_Peterson.jpg

(Open addressing) T. Cormen et al., "Introduction to algorithms", Chap 11.4, MIT press, (2022)

R. P. Brent, "Reducing the retrieval time of scatter storage techniques", Communications of the ACM (1973)

(Robin Hood) P. Celis et al., "Robin hood hashing" Symposium on Foundations of Computer Science (1985)

(CPython hash tables) <https://tenthousandmeters.com/blog/python-behind-the-scenes-10-how-python-dictionaries-work/>

Appendix

Hard drives and direct-address tables

There are 2^{128} possible IPv6 addresses

A 1TB hard drive $\approx 2^{40}$ bytes

The size of slots depends on the implementation

Supposing 8 bytes per slot, then we require

$2^3 \cdot 2^{128} / 2^{40} = 2^{91} \approx 10^{27}$ 1TB hard drives

or one thousand trillion trillion 1TB hard drives