

Course Algorithms and Data Structures · Lecture 12

Hash Tables

Textbook: A. Levitin, Section 7.3

The idea: a cloakroom

You get ticket 37 for your coat. Later the attendant goes straight to hook 37: one step, however many coats there are.

A **hash function** h turns every key K into an index of an array H[0..m – 1]:
$h(K) \in \{0, 1, \ldots, m - 1\}$

Insert, search, delete: one computation, one array access, O(1) on average.

Hash functions

- **Fast** to compute
- Spread the keys **evenly** over the table
- Always the **same** index for the same key

Integers

h(K) = K mod m, with m prime
m = 13: h(27) = 1, h(15) = 2

Strings (Levitin)

sum of letter positions mod m
h(MONEY) = 72 mod 13 = 7

Collisions

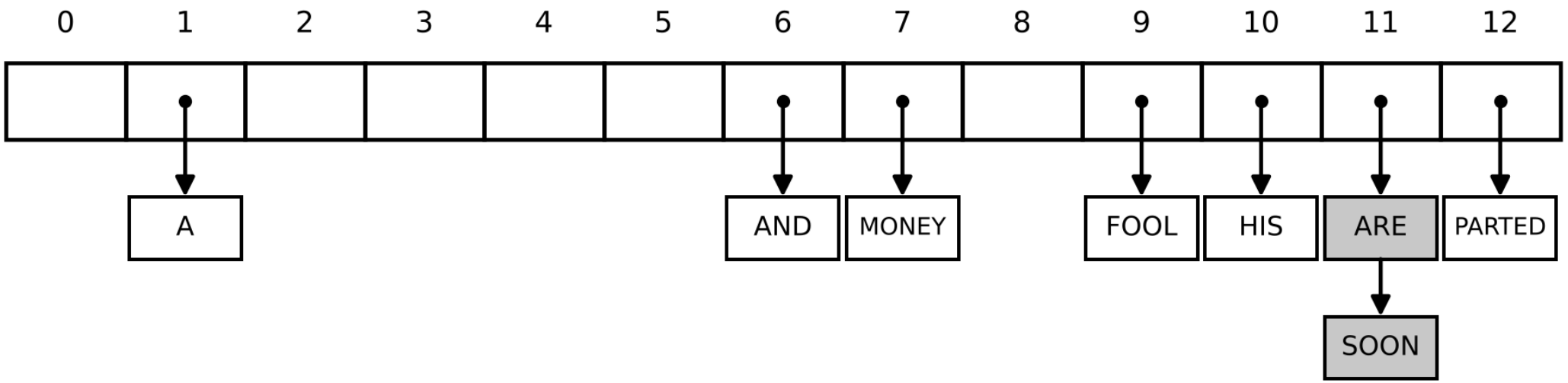
Insert A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED into a table of size 13:

Key	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
Sum	1	48	19	36	72	24	63	64
h(K)	1	9	6	10	7	11	11	12

ARE and SOON both get 11: a **collision**. Collisions can’t be avoided, so every hash table needs a strategy.

Strategy 1: separate chaining

Each cell holds a **linked list** of all keys that hash to it.

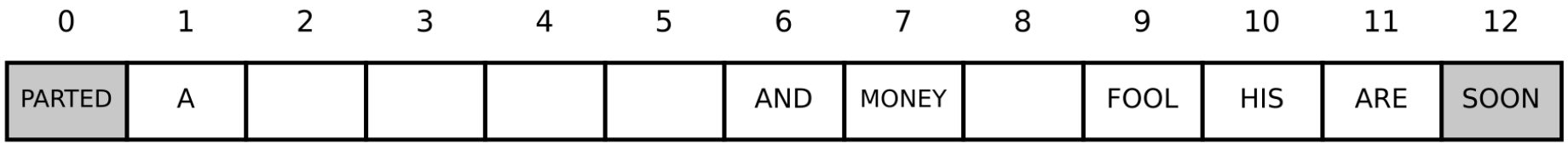


- Search SOON: cell 11, compare ARE, then SOON: found
- Search KID: h = 24 mod 13 = 11; ARE, SOON, end of list: not found

Strategy 2: linear probing

All keys stay in the array. If the cell is taken, try the next one, wrapping around after the last cell.

```
ALGORITHM InsertLinearProbing(H[0..m − 1], K)
    i ← h(K)
    while H[i] is not empty do
        i ← (i + 1) mod m
    H[i] ← K
```



Linear probing: search and deletion

- SOON wanted cell 11 (taken by ARE): went to 12
- PARTED wanted 12 (taken by SOON): went to 0
- Search KID: 11, 12, 0, 1, then cell 2 is empty: not found

Clustering

colliding keys form long blocks of occupied cells, and searches walk along them

Deletion

don’t simply empty a cell: mark it “deleted”, or later searches stop too early

The load factor

$$\alpha = n / m$$

- **Chaining**: α = average list length; search takes about 1 + α/2 steps: O(1) when α is small
- **Linear probing**: keep α below about 0.5–0.7
- Too full? **Rehash**: build a table twice as large. Rare, so still O(1) on average

How fast is a hash table?

Operation	Average	Worst
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

Worst case: all keys in one cell. With a good hash function and a sensible load factor it practically never happens.

Hash table or tree?

	Sorted array	Balanced BST	Hash table
Search	O(log n)	O(log n)	O(1) average
Insert / delete	O(n)	O(log n)	O(1) average
Keys in order	yes	yes, inorder	no

- Used in Python dict and set, counting words, removing duplicates, databases, caches

Summary

- A hash table stores each key in the cell given by a hash function: O(1) on average.
- Collisions are resolved by separate chaining or by linear probing.
- The load factor $\alpha = n/m$ controls the speed; a full table is rebuilt twice as large.
- Hash tables are the fastest dictionaries, but they keep no order.

Review questions

1. What is a hash function? What makes a hash function good?
2. Insert 10, 22, 31, 4, 15, 28 into a table of size 11 with h(K) = K mod 11, using linear probing. Draw the final table.
3. What is a collision? Name two ways to resolve collisions.

Reading: Levitin, Section 7.3; Bhargava, Chapter 5.