
ADT Λεξικού

Hash tables

ADT Λεξικού

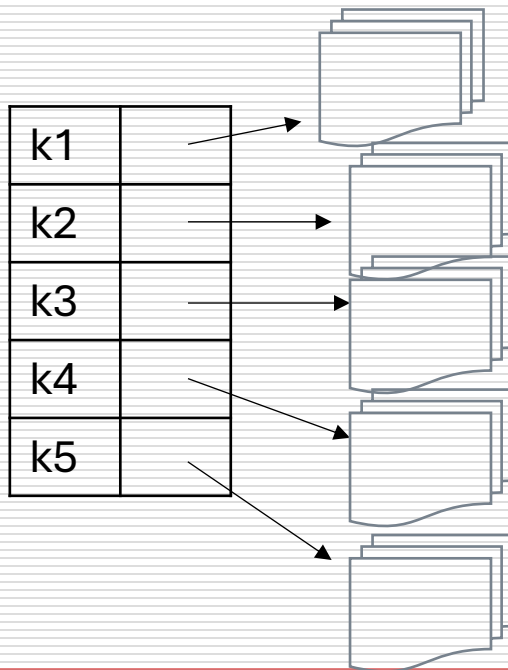
- Δυναμικά μεταβαλλόμενη συλλογή αντικειμένων που αναγνωρίζονται με «κλειδί» (π.χ. κατάλογοι, πίνακες ΒΔ).
- Λεξικό : συλλογή αντικειμένων με μοναδικό «κλειδί».
 - «Κλειδί»: αριθμός ή τύπος δεδομένων (με ολική διάταξη).
- Υποστηριζόμενες Λειτουργίες:
 - Αναζήτηση στοιχείου με κλειδί k
 - $member(k)$: ελέγχει ύπαρξη στοιχείου με κλειδί k
 - $search(k)$ ή $lookup(k)$: επιστρέφει δείκτη στο στοιχείο με κλειδί k
 - Εισαγωγή στοιχείου με κλειδί k
 - $insert(k, object)$
 - Διαγραφή στοιχείου με κλειδί k
 - $delete(k)$

ADT Λεξικού – Επιπλέον Λειτουργίες

- ❑ Μπορεί ακόμη να υποστηρίζει λειτουργίες που σχετίζονται με ταξινόμηση κλειδιών:
 - `outputSorted()`: Εκτύπωση στοιχείων σε αύξουσα / φθίνουσα σειρά κλειδιών
 - `min()`, `max()`: Ελάχιστο και μέγιστο στοιχείο με βάση το κλειδί.
 - `predecessor(k)`, `successor(k)`: Προηγούμενο και επόμενο στην ολική διάταξη από το στοιχείο που δίνεται ως είσοδος.
 - `select(i)`: Επιστρέφει δείκτη στο στοιχείο με το i -οστό μικρότερο κλειδί
 - `rank(k)`: Επιστρέφει πλήθος στοιχείων με τιμή κλειδιών **το πολύ** k
- ❑ Δεν υποστηρίζονται από όλες τις δομές δεδομένων λεξικού...

Περί Κλειδιών...

- Ασχολούμαστε αποκλειστικά με τα **κλειδιά**
- Όμως αυτά μπορεί να είναι ή να περιέχουν **δείκτες** σε άλλα αντικείμενα



Υλοποιήσεις Λεξικού

- Μη-ταξινομημένη διασυνδεδεμένη λίστα:
 - Εισαγωγή: $O(1)$
 - Μοναδικό κλειδί
 - Αναζήτηση / τυχαία διαγραφή: $O(n)$
 - Κατάλληλη όταν:
 - συχνές εισαγωγές
 - σπάνιες αναζητήσεις
 - διαγραφές μεμονωμένες ή στο άκρο (π.χ. ουρά).
- Ταξινομημένος πίνακας:
 - (Δυαδική) αναζήτηση: $O(\log n)$
 - min, max: $O(1)$
 - predecessor, successor: $O(\log n)$
 - outputSorted(): $O(n)$
 - select(): $O(1)$
 - rank(): $O(\log n)$
 - Στατική συλλογή: «εισαγωγή»/ «διαγραφή» $O(n)$ / στοιχείο Χρόνος ταξινόμησης: $O(n \log n)$
 - **Κατάλληλη** για συχνές αναζητήσεις και δεδομένα μεταβάλλονται σπάνια (π.χ. Άγγλο-ελληνικό λεξικό).

Υλοποιήσεις Λεξικού

□ (Δυαδικό) Δέντρο Αναζήτησης*

- Αναζήτηση / εισαγωγή / διαγραφή: $O(\log n)$
- Μέγιστο / ελάχιστο / προηγούμενο / επόμενο / k -οστό: $O(\log n)$
- `outputSorted`: $O(n)$
- **Range queries** σε γραμμικό χρόνο.
- Πλήρως δυναμική – επιπλέον χώρος για δείκτες!
- ***Balanced**

□ Hash Table

- Αναζήτηση / διαγραφή : $O(1)^*$
- Εισαγωγή:
 - $O(1)^*$
 - Καλή υλοποίηση
 - Μη παθολογικά δεδομένα!
- **Δεν υποστηρίζει αποδοτικά** άλλες λειτουργίες.
- Δυναμική – επιπλέον χώρος στον πίνακα ($util \approx 50\%$)

Εφαρμογές Λεξικού

- ❑ Motivating Application:
 - Symbol tables σε πρώτους compilers
- ❑ Network filters:
 - Μπλοκάρισμα πακέτων από συγκεκριμένες διευθύνσεις IP (πχ. spammers, DDoS)
- ❑ Deduplication
 - Εύρεση & διαγραφή διπλοτύπων
 - Διατήρηση μοναδικού αντιγράφου με πολλές εισαγωγές
 - Πώς βοηθάει το λεξικό;

Κοινό χαρακτηριστικό: Πολλές αναζητήσεις σε **μεγάλες** ποσότητες δεδομένων

2-SUM Problem

□ Είσοδος:

Δίνεται μη-ταξινομημένος πίνακας n ακεραίων A και ένας ακέραιος t .

□ Έξοδος:

Υπάρχουν ακέραιοι $x, y \in A: x + y = t$?

□ Λύσεις:

- $\Theta(n^2)$

- $O(n \cdot \log n)$

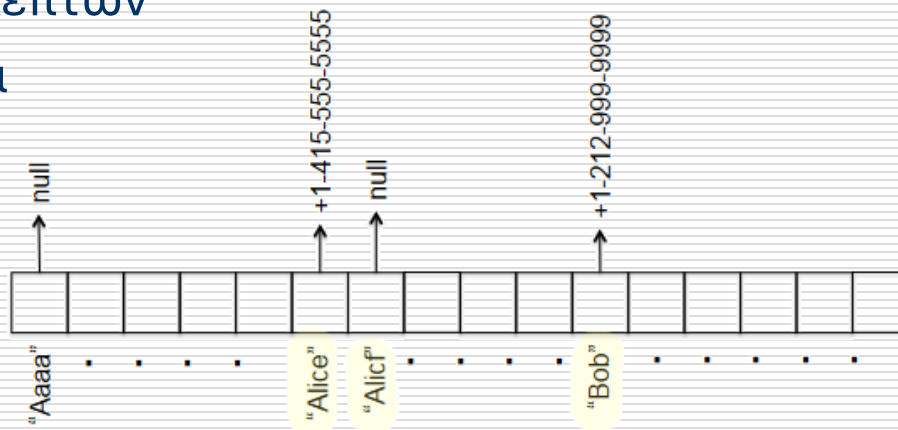
- $O(n)$

Hash Tables

- Γενίκευση των arrays
 - Random access σε $O(1)$
 - Χωρίς ακέραιους δείκτες
 - Με δυναμική αλλαγή μεγέθους
- Υποστηριζόμενες λειτουργίες
 - $\text{Lookup}(k): O(1)$ *
 - $\text{Insert}(k): O(1)$
 - $\text{Delete}(k): O(1)$ *
- Όχι πληροφορίες διάταξης
- Κάνουν λίγα πράγματα αλλά τα κάνουν εξαιρετικά καλά!

Κλειδιά

- Δίνεται ένα σύνολο U από **πιθανά** κλειδιά
 - Όλες οι πιθανές διευθύνσεις IP: 2^{128} .
 - Όλα τα πιθανά ονόματα μήκους 25 (26^{25}).
- Χρησιμοποιείται ένα σύνολο S από **πραγματικά** κλειδιά
 - Διευθύνσεις IP επισκεπτών
 - Πραγματικά ονόματα
- $|S| \ll |U|$

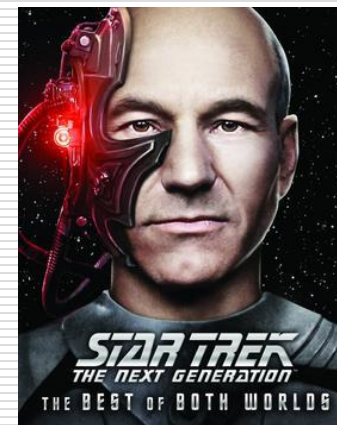


Διαφορές Υλοποίησης

Δομή	Χώρος	Χρόνος Lookup(k)
Πίνακας	$\Theta(U)$	$O(1)$
Γραμμική Λίστα	$\Theta(S)$	$\Theta(S)$
Hash-Table	$\Theta(S)$	$\Theta(1)$

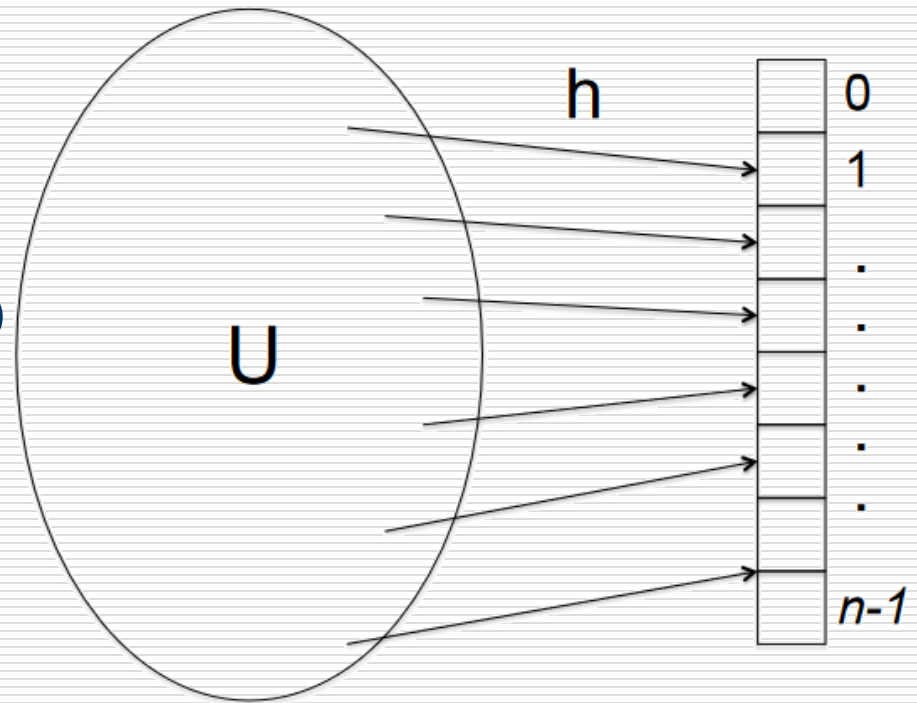
Hash Tables: **The best of both worlds**

Βασική ιδέα: Απεικόνιση στοιχείων U σε στοιχεία του S μέσω μιας συνάρτησης



Hash Functions

- Μετάφραση στοιχείων του U σε ακέραιους
- $H: U \rightarrow [n]$ με (π.χ. $n = 2|S|$)
- Αναζήτηση κλειδιού
 - Υπολογισμός hash
 - Προσπέλαση πίνακα
- Collision
 - $k_1 \neq k_2$ και $H(k_1) = H(k_2)$
- Σίγουρες λόγω:
 - $|S| \ll |U|$
 - Αρχή Περιστεριώνα
- Συχνές:
 - Παράδοξο Γενεθλίων



Παραδείγματα hash functions

□ Συναρτήσεις υπολοίπου

- $h(k) = k \bmod n$

- $h(k) = (ak + b) \bmod n$ με $\gcd(a, n) = 1$

- Ομοιόμορφη κατανομή για ομοιόμορφα δεδομένα

- Δεν ισχύει στον πραγματικό κόσμο π.χ. τιμές προϊόντων σε λεπτά $\bmod 100 \in \{0, 50, 99\}$

- $h(k) = (ak + b) \bmod 2^w \div 2^{w-r}$ με a τυχαίο περιττό

- Κρατάμε τα r MSB από τα w LSB του $ak + b$

□ Παραλλαγές

- $h(k) = (ak + b) \bmod p \bmod n, p > |U|$ πρώτος, τυχαία a, b

Collision Resolution - Chaining

- ❑ Χειρισμός σύγκρουσης:
- ❑ Κάθε θέση δείχνει σε συνδεδεμένη λίστα - bucket
- ❑ **Lookup(k):**
 - Υπολογισμός hash $h(k)$
 - Σειριακή Αναζήτηση στη λίστα $A[h(k)]$
- ❑ Για λειτουργίες $O(1)$
 - Σταθερό μέγεθος bucket
 - Υπόθεση: H αποτιμάται σε σταθερό χρόνο.

Παράδειγμα:

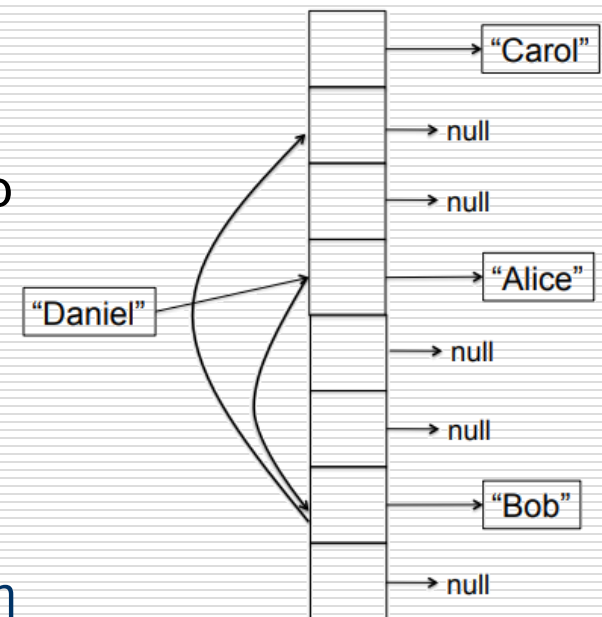
$$H(k) = k \bmod 4, U = [15]$$

0	→	4	8	12	
1	→	1	5	9	13
2	→	2	14	6	10
3	→	3	7	11	15

Collision Resolution

Probing (Open Addressing)

- Σε κάθε θέση του πίνακα αποθηκεύεται το πολύ 1 κλειδί
 - Όχι λίστα
- Κάθε κλειδί αντιστοιχεί σε ένα σύνολο πιθανών θέσεων (probe positions)
 - Το $h(k)$ είναι ένα σύνολο
- Εισαγωγή:
 - Υπολογισμός $h(k)$
 - Αναζήτηση μέχρι να βρεθεί κενή θέση σε κάποιες από τις πιθανές θέσεις του συνόλου $\{A[h(k)]\}$



Εναλλακτικά Σχήματα Probing

□ Linear Probing

- Υπολογισμός $h(k)$
- $\{h(k), h(k) + 1, h(k) + 2, \dots\} \bmod n$
- Χρήση probe function $\text{pos}(k, i) = (h(k) + f(i)) \bmod n$

□ Double Hashing

- $\text{pos}(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod n, i \geq 0$
- $h_1(k)$: Αρχική θέση
- $h_2(k)$: Βήμα Μετατόπισης (σε περίπτωση κατειλημμένης θέσης)
- Παράδειγμα:
 - $h_1(k) \rightarrow 20$
 - $h_2(k) \rightarrow 43$
 - Probing sequence: $\{20, 63, 106, \dots\} \bmod n$

Εκτίμηση Ποιότητας

- Φόρτος hash table
 - $\alpha = \text{πλήθος_αποθηκευμένων_κλειδιών} / \text{πλήθος_θέσεων}$
- Εκτίμηση **αναμενόμενης** απόδοσης lookup
 - Chaining: $O(1 + \alpha)$
 - α - αναμενόμενο μέγεθος bucket
 - Open Addressing: $O(\frac{1}{1-\alpha})$
 - α – πιθανότητα hash σε κατειλημμένη θέση $\alpha < 1$
 - $1 + \alpha + \alpha^2 + \dots = \frac{1}{1-\alpha}$ (για ανεξάρτητα probes)

Ποιότητα hash functions

- Θα διαλέγατε ποτέ την συνάρτηση $h(k) = 0, \forall k \in U$;

Ποιότητα hash functions

- Θα διαλέγατε ποτέ την συνάρτηση $h(k) = 0, \forall k \in U$;
- Όμως λόγω γενίκευσης αρχής περιστεριώνα:
 - $\forall h: U \rightarrow [n], \exists S \subset U \text{ με } |S| = \frac{|U|}{n} : h(k_1) = h(k_2), \forall k_1, k_2 \in S$

Ποιότητα hash functions

- ❑ Θα διαλέγατε ποτέ την συνάρτηση $h(k) = 0, \forall k \in U$;
- ❑ Όμως λόγω γενίκευσης αρχής περιστεριώνα:
 - $\forall h: U \rightarrow [n], \exists S \subset U \text{ με } |S| = \frac{|U|}{n} : h(k_1) = h(k_2), \forall k_1, k_2 \in S$
- ❑ Όλες οι λειτουργίες του Hash Table είναι $O(1)$ υπό την υπόθεση μιας καλής hash function.

Ποιότητα hash functions

- ❑ Θα διαλέγατε ποτέ την συνάρτηση $h(k) = 0, \forall k \in U$;
- ❑ Όμως λόγω γενίκευσης αρχής περιστεριώνα:
 - $\forall h: U \rightarrow [n], \exists S \subset U \text{ με } |S| = \frac{|U|}{n} : h(k_1) = h(k_2), \forall k_1, k_2 \in S$
- ❑ Όλες οι λειτουργίες του Hash Table είναι $O(1)$ υπό την υπόθεση μιας καλής hash function.
- ❑ Μήπως να διαλέξουμε μια τυχαία συνάρτηση;

Ποιότητα hash functions

- ❑ Θα διαλέγατε ποτέ την συνάρτηση $h(k) = 0, \forall k \in U$;
- ❑ Όμως λόγω γενίκευσης αρχής περιστεριώνα:
 - $\forall h: U \rightarrow [n], \exists S \subset U \text{ με } |S| = \frac{|U|}{n} : h(k_1) = h(k_2), \forall k_1, k_2 \in S$
- ❑ Όλες οι λειτουργίες του Hash Table είναι $O(1)$ υπό την υπόθεση μιας καλής hash function.
- ❑ Μήπως να διαλέξουμε μια τυχαία συνάρτηση;
 - $\Pr[h(k_1) = h(k_2)] = \frac{1}{n}$ για $k_1 \neq k_2$ (universal hashing)
 - Δύσκολο στην αποθήκευση και αποτίμηση

Ποιότητα hash functions

- ❑ Θα διαλέγατε ποτέ την συνάρτηση $h(k) = 0, \forall k \in U$;
- ❑ Όμως λόγω γενίκευσης αρχής περιστεριών:
 - $\forall h: U \rightarrow [n], \exists S \subset U \text{ με } |S| = \frac{|U|}{n}: h(k_1) = h(k_2), \forall k_1, k_2 \in S$
- ❑ Όλες οι λειτουργίες του Hash Table είναι $O(1)$ υπό την υπόθεση μιας καλής hash function.
- ❑ Μήπως να διαλέξουμε μια τυχαία συνάρτηση;
 - $\Pr[h(k_1) = h(k_2)] = \frac{1}{n}$ για $k_1 \neq k_2$ (universal hashing)
 - Δύσκολο στην αποθήκευση και αποτίμηση

Ποιότητα hash functions

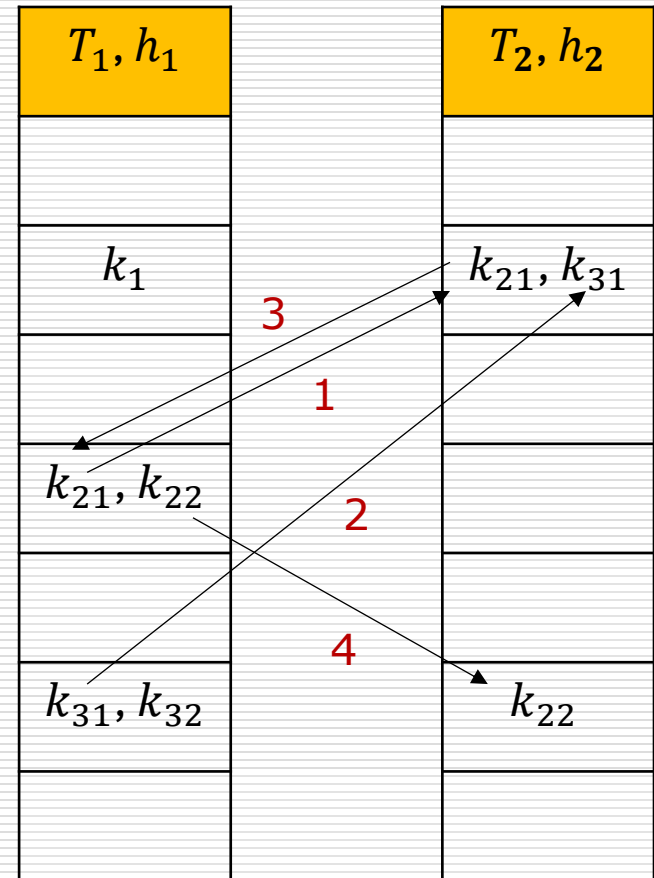
- ❑ Θα διαλέγατε ποτέ την συνάρτηση $h(k) = 0, \forall k \in U$;
- ❑ Όμως λόγω γενίκευσης αρχής περιστεριώνα:
 - $\forall h: U \rightarrow [n], \exists S \subset U \text{ με } |S| = \frac{|U|}{n} : h(k_1) = h(k_2), \forall k_1, k_2 \in S$
- ❑ Όλες οι λειτουργίες του Hash Table είναι $O(1)$ υπό την υπόθεση μιας καλής hash function.
- ❑ Μήπως να διαλέξουμε μια τυχαία συνάρτηση;
 - $\Pr[h(k_1) = h(k_2)] = \frac{1}{n}$ για $k_1 \neq k_2$ (universal hashing)
 - Δύσκολο στην αποθήκευση και αποτίμηση
- ❑ Απαιτούμενα χαρακτηριστικά h:
 - Αποτίμηση σε $O(1)$
 - ‘Μοιάζει’ με τυχαία συνάρτηση (*~ομοιόμορφη κατανομή κλειδιών*)

Πρακτικά θέματα

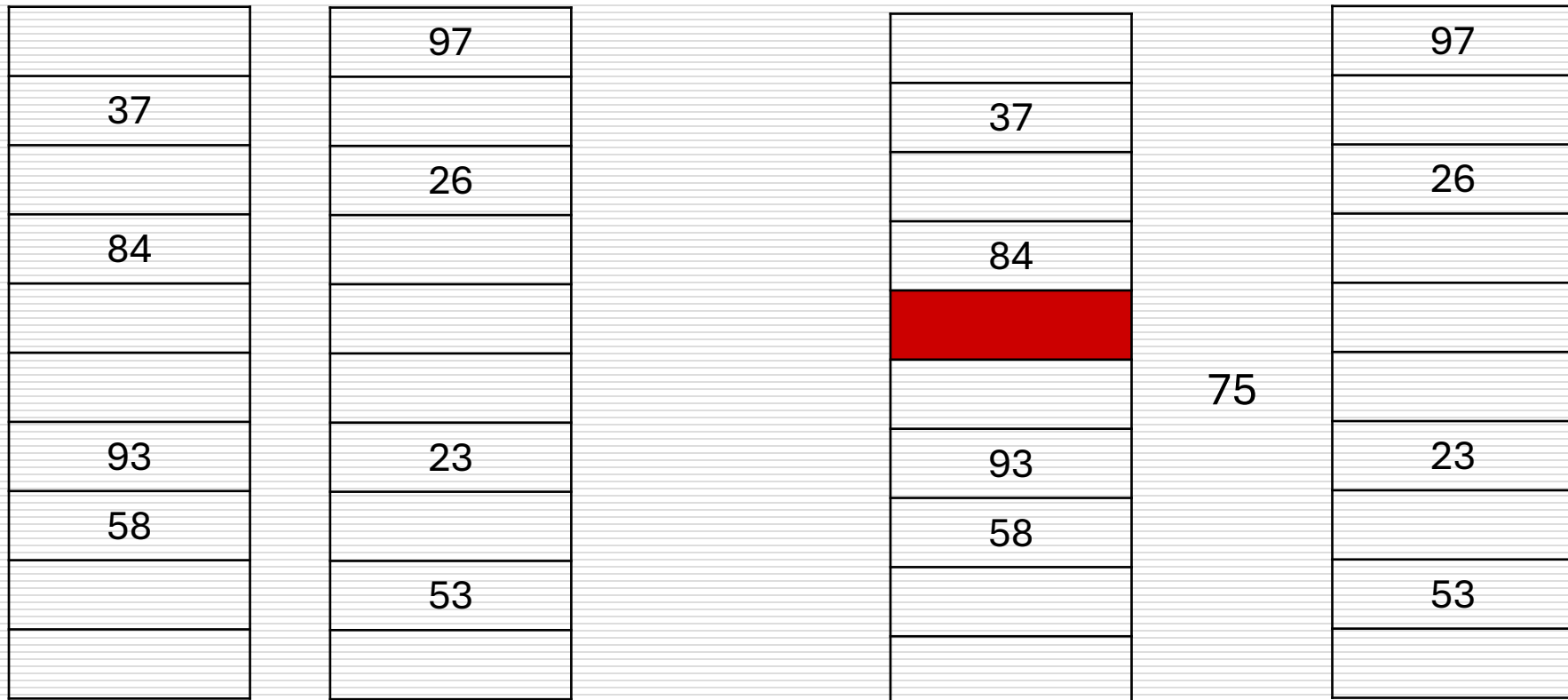
- ❑ Table – Resizing (Rehashing)
 - Όταν $\alpha > \text{threshold}$: αλλαγή μεγέθους πίνακα
 - Συνήθως διπλασιάζουμε το μέγεθος
 - Επιλογή νέας συνάρτησης hash
- ❑ Cryptographic hash functions
 - (Υπολογιστική) Δυσκολία εύρεσης συγκρούσεων
 - Δεν είναι $O(1)$ η αποτίμηση
 - ❑ Όμως είναι πολύ γρήγορη (υποστήριξη και από υλικό)

Cuckoo Hashing (Pagh, Rodler 2001)

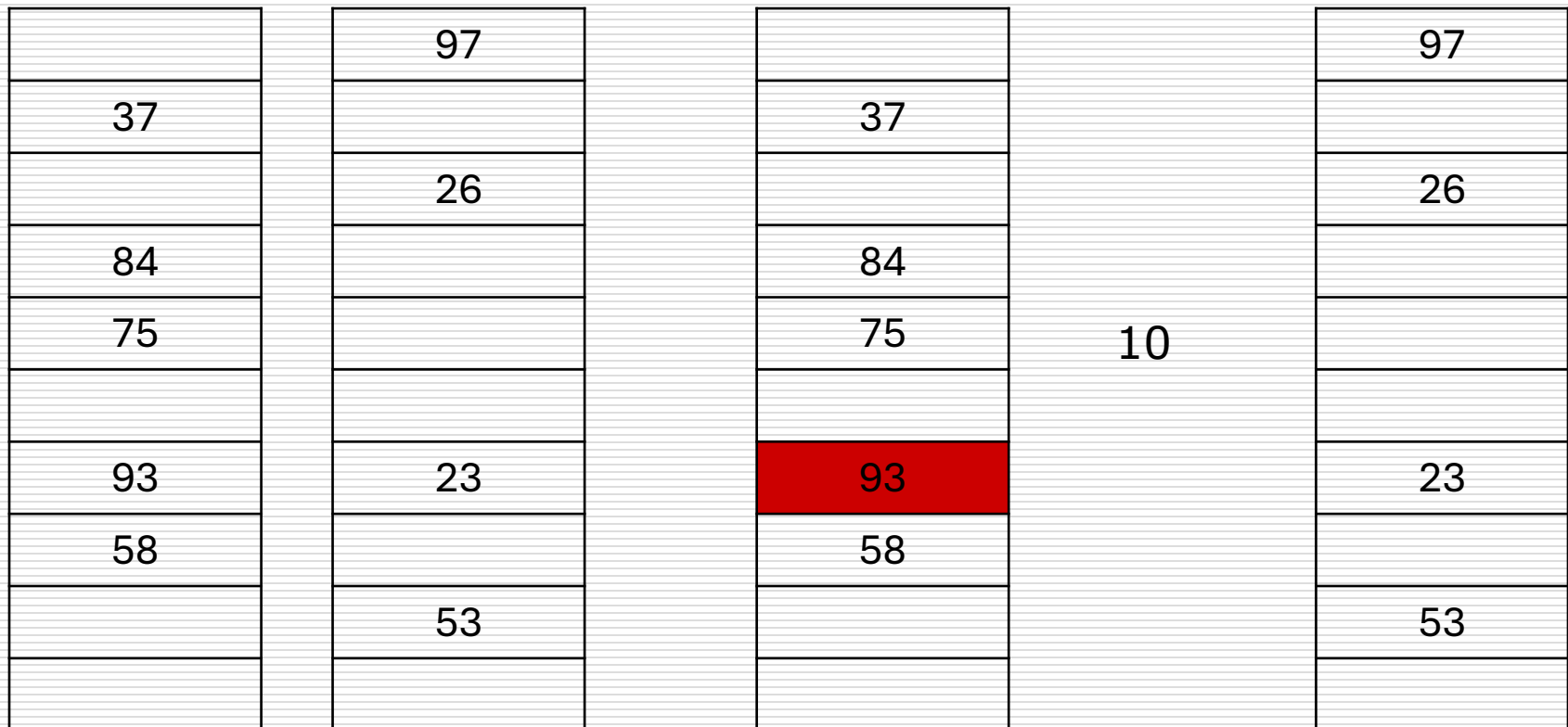
- Χρήση δύο συναρτήσεων h_1, h_2
- ... και δύο πινάκων T_1, T_2
- Χρήση $h_1 \rightarrow$ τοποθέτηση σε T_1
- Χρήση $h_2 \rightarrow$ τοποθέτηση σε T_2
- Σύγκρουση
 - Νέο κλειδί $k: T_1[h_1(k)]$
 - Παλιό κλειδί: $k': T_2[h_2(k')]$
- Σε νέα σύγκρουση στον T_2
 - Νέο κλειδί $k': T_2[h_2(k')]$
 - Παλιό κλειδί $k'': T_1[h_1(k'')]$
 - Νέα σύγκρουση στον T_1
 - Τη χειριζόμαστε ομοίως (διώχνουμε προηγούμενο)
- Rehash όταν έχω υπέρβαση ορίου ταξιδιών - κύκλο



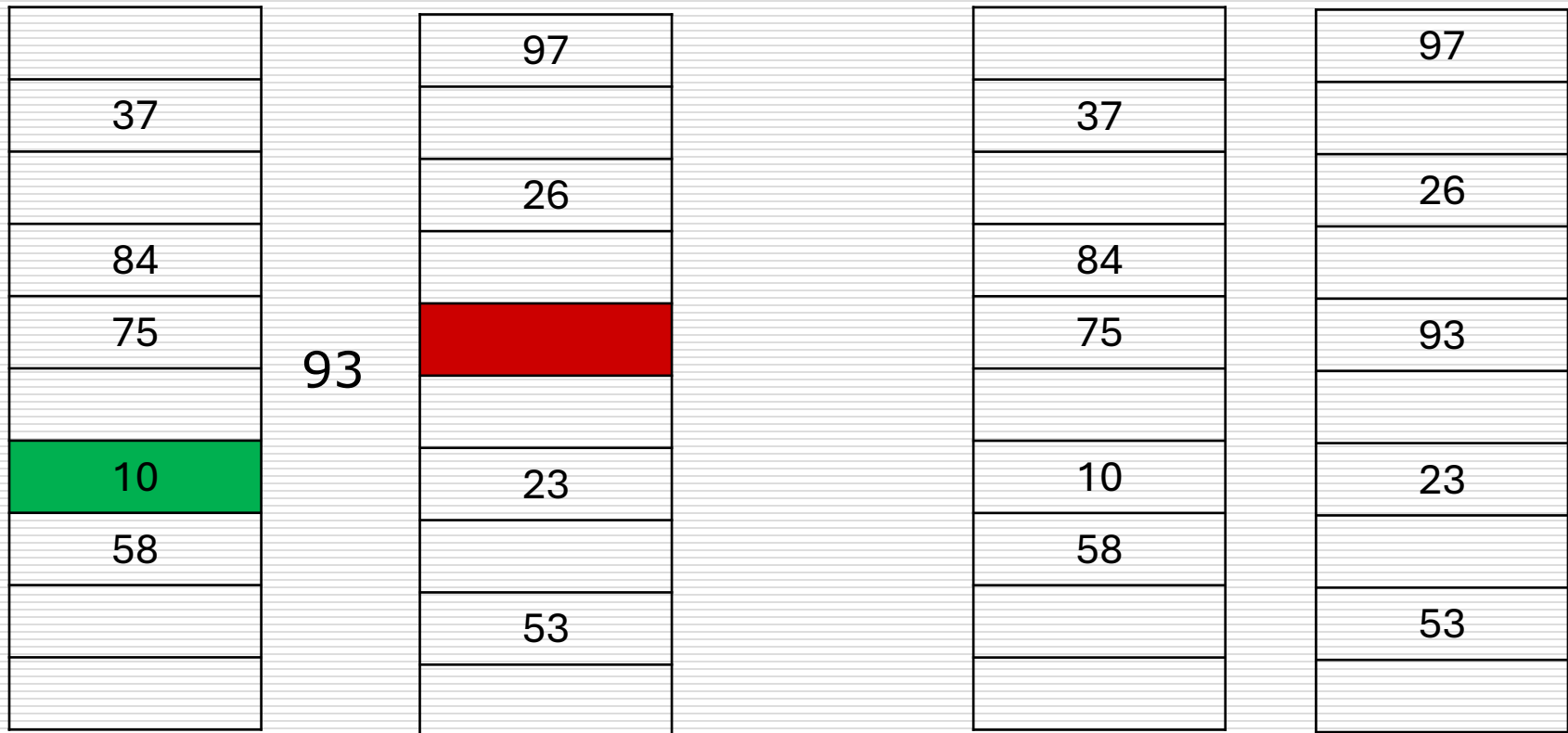
Cuckoo Hashing - Παράδειγμα



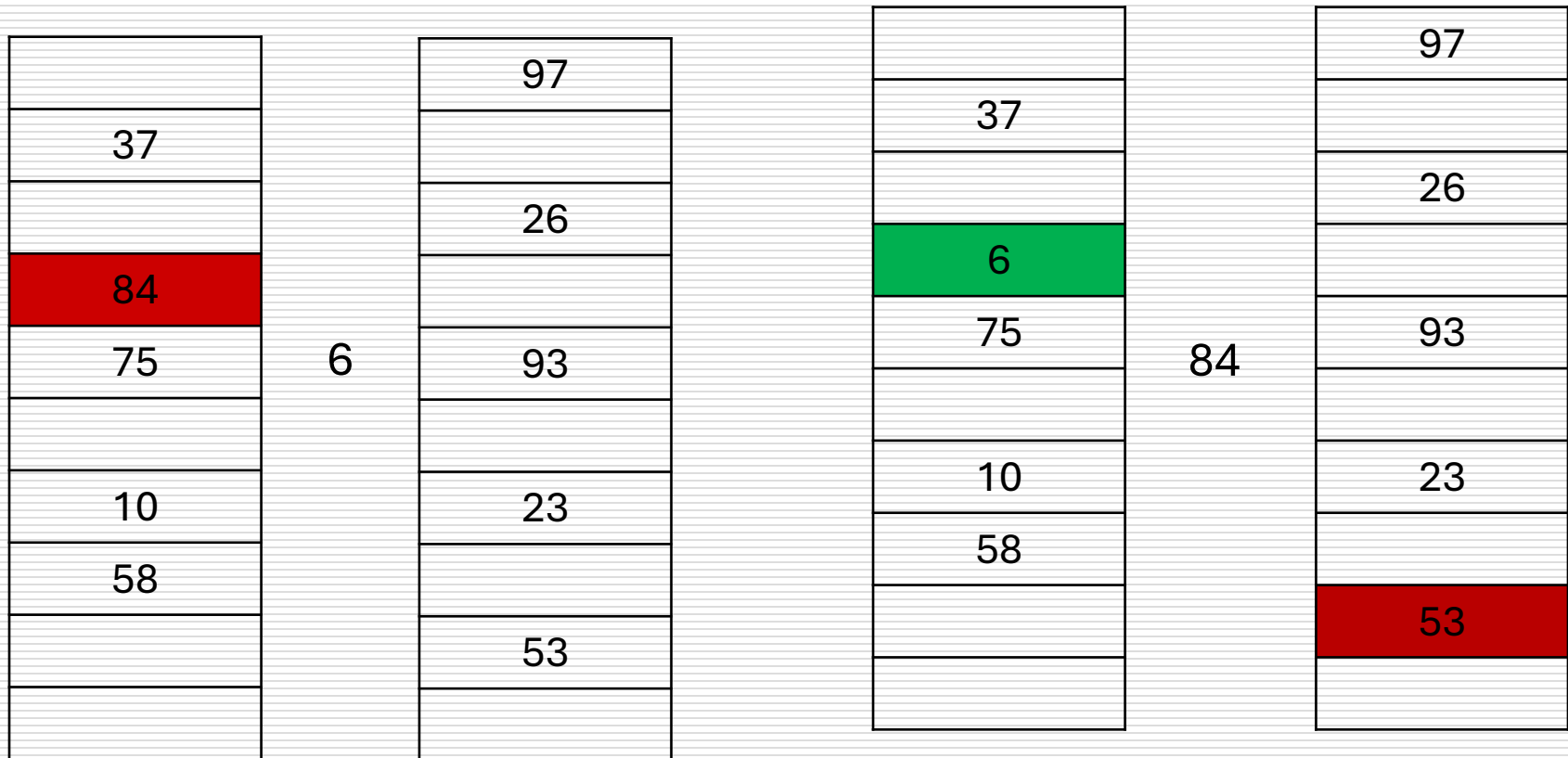
Cuckoo Hashing - Παράδειγμα



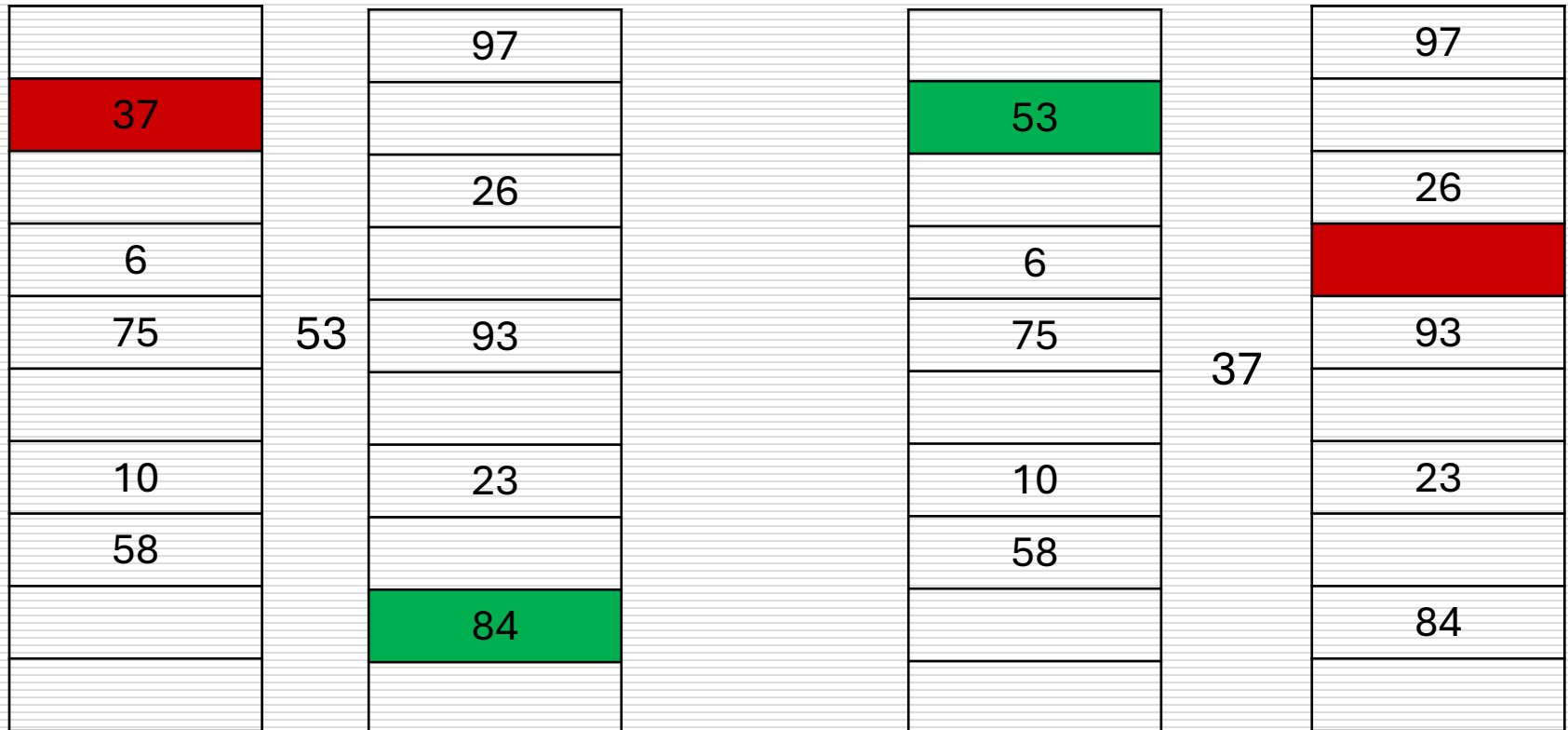
Cuckoo Hashing - Παράδειγμα



Cuckoo Hashing - Παράδειγμα



Cuckoo Hashing - Παράδειγμα

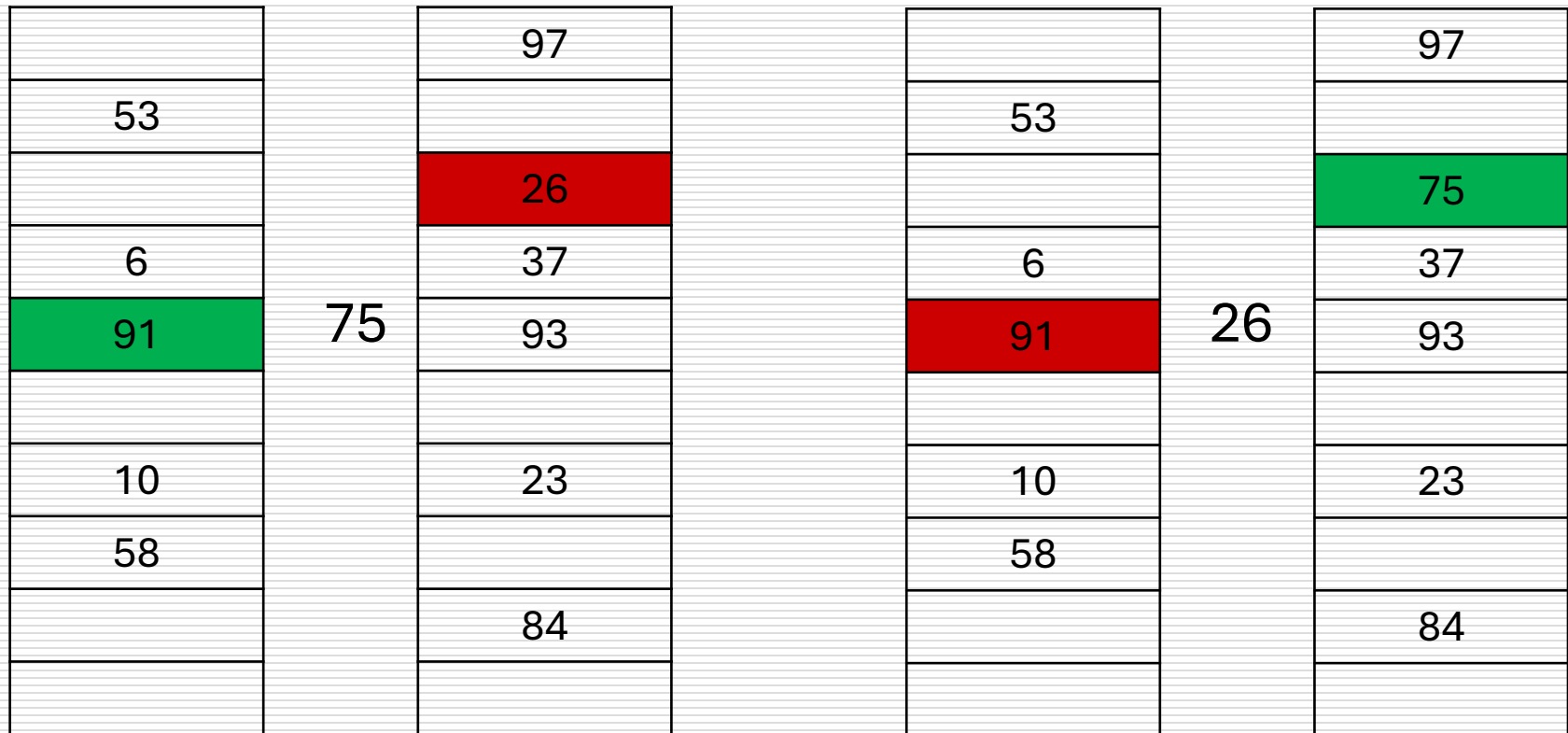


Cuckoo Hashing - Παράδειγμα

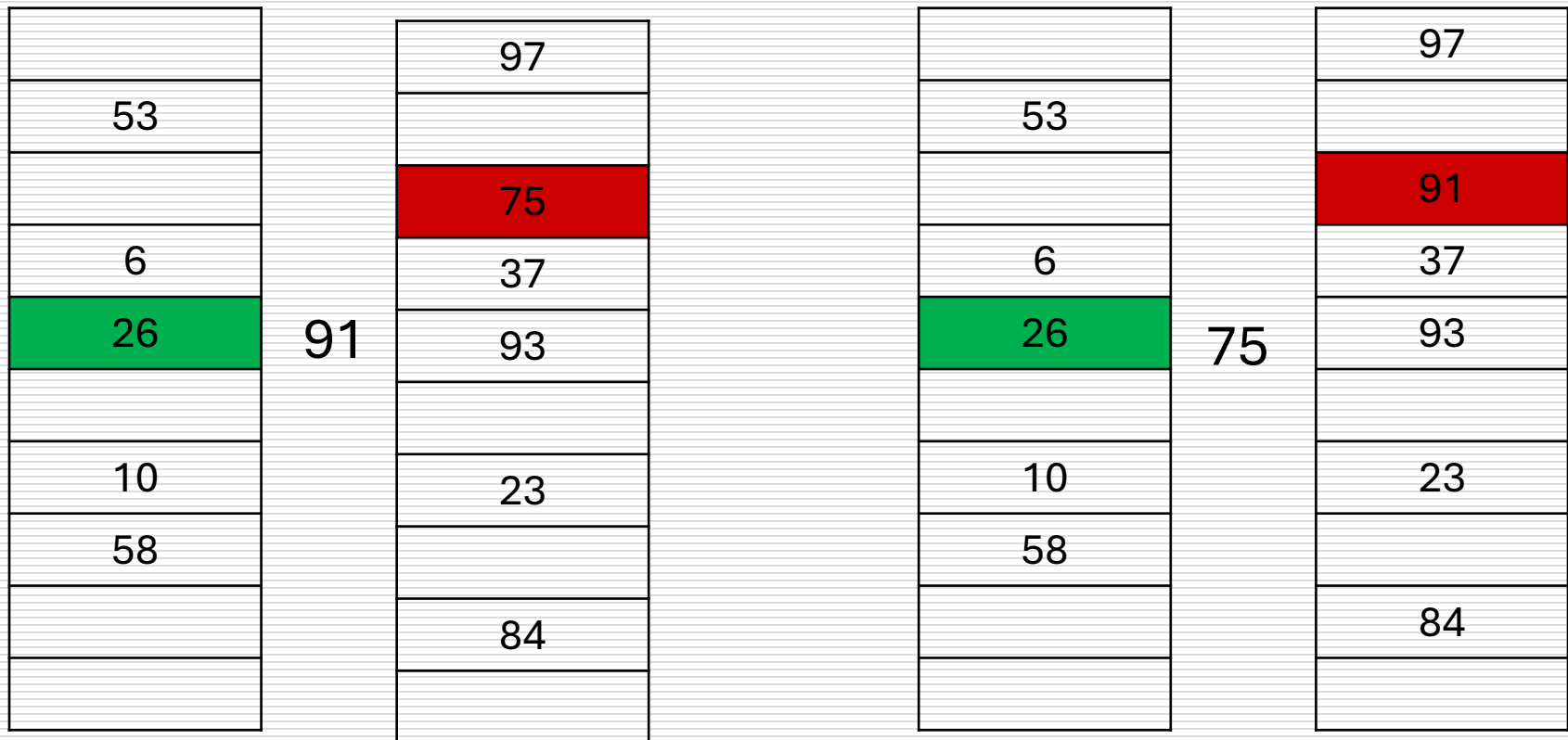
	97		97
53		53	
	26		26
6	37	6	37
75	93	75	93
10	23	10	23
58		58	
	84		84

91

Cuckoo Hashing - Παράδειγμα



Cuckoo Hashing - Παράδειγμα



Cuckoo Hashing (υλοποίηση)

lookup(k):

```
i=h1(k)
if T1[i]==k:
    return 1,i
else:
    i=h2(k)
    if T2[i] == k:
        return 2,i
    else:
        return null
```

**$O(1)$ – ελέγχος 2 θέσεων
χειρότερης περίπτωσης!**

delete(k):

```
i=h1(k)
if T1[i]==k:
    T1[i] = null
else:
    i=h2(k)
    if T2[i] == k:
        T2[i] = null
```

insert(k) :

```
if T1[h1(k)]==k or T2[h2(k)] == k:
    return
for i in range(max):
    swap(k, T1[h1(k)])
    if k==null return
    swap(k, T2[h2(k)])
    if k==null return
rehash()
insert(k)
```

**expected $O(n)$ για η εισαγωγές
υπό προϋποθέσεις**

Πιθανοτικά Hash Tables: Bloom Filters

Burton Bloom: Space/time trade-offs in hash coding with allowable errors. CACM 13(7), 1970.

- ❑ $\text{Lookup}(k) \in \{False, True\}$
 - Επιστρέφει ύπαρξη και όχι θέση.
- ❑ **False Positive:** Υπάρχουν σε μικρό βαθμό.
 - Αν δεις True, δεν είναι βέβαιο ότι υπάρχει αντικείμενο με κλειδί k .
 - Πιθανότητα λάθους - μικρή
- ❑ **False Negative:** Δεν υπάρχουν.
 - Αν δεις False ξέρεις σίγουρα ότι δεν υπάρχει αντικείμενο με κλειδί k .
- ❑ Χαρακτηριστικά
 - Εγγυημένο Constant Time
 - Space efficient
 - Υλοποίηση – επηρεάζει error rate
 - Όχι (εύκολες) διαγραφές

Εφαρμογές

- ❑ Μη αποδεκτά passwords
 - Εισάγουμε τα μη αποδεκτά passwords σε ένα bloom filter
 - Lookup για κάθε password
 - False positive: απόρριψη καλού password
- ❑ Έλεγχος ορθογραφίας
 - Εισάγουμε τις λέξεις σε ένα bloom filter
 - Lookup για κάθε λέξη
 - False positive: δεκτή ανορθόγραφη λέξη
 - Tradeoff με χώρο αποθήκευσης
- ❑ Internet Routing:
 - Blocked / Cached Διευθύνσεις IP

Bloom Filters – Βασική υλοποίηση

□ Πίνακας A n -bit (n θέσεων)

■ Αρχικά όλα τα στοιχεία 0

□ m hash functions
(σταθερός αριθμός)

■ $h_i: U \rightarrow [n]$

■ Στοιχεία του πίνακα

■ $0 \rightarrow 1$ (ποτέ αντίστροφα)

insert(k):

for i **in** $\text{range}(m)$:

$A[h_i(k)] = 1$

bool lookup(k):

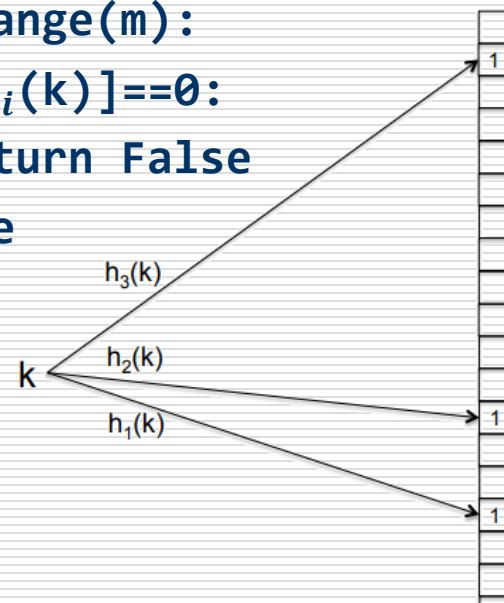
for i **in** $\text{range}(m)$:

if $A[h_i(k)] == 0$:

return False

return True

Key	Value of h_1	Value of h_2	Value of h_3
k_1	23	17	5
k_2	5	48	12
k_3	37	8	17
k_4	32	23	2



Υπολογισμός n – heuristic analysis

- Διαισθητικά:
 - Όσο μεγαλώνει το n – μικραίνει η πιθανότητα false positive
 - Δεν θέλουμε σπατάλη χώρου
- Υποθέσεις: $h_i(k)$
 - ανεξάρτητες τυχαίες μεταβλητές
 - ομοιόμορφα κατανεμημένες στο $[n]$
- False positive: $k \notin S, \forall i \in [m]: A[h_i(k)] = 1$
 - Δεν έχει κληθεί η insert για το συγκεκριμένο κλειδί, ενώ η hash λέει πως έχει

Υπολογισμός n – heuristic analysis

- Πιθανότητα συγκεκριμένο bit από τα n του A να έχει τεθεί 1 κατά την εισαγωγή των κλειδιών του S

- $\left(\frac{1}{n}\right)^{|S|}$

- $\left(1 - \frac{1}{n}\right)^{|S|}$

- $\left(1 - \frac{1}{n}\right)^{m|S|}$

- $1 - \left(1 - \frac{1}{n}\right)^{m|S|}$

Υπολογισμός n – heuristic analysis

- Πιθανότητα συγκεκριμένο bit από τα n του A να έχει τεθεί 1 κατά την εισαγωγή των κλειδιών του S

$$\blacksquare \left(\frac{1}{n} \right)^{|S|}$$

$$\blacksquare \left(1 - \frac{1}{n} \right)^{|S|}$$

$$\blacksquare \left(1 - \frac{1}{n} \right)^{m|S|}$$

$$\blacksquare 1 - \left(1 - \frac{1}{n} \right)^{m|S|}$$

Υπολογισμός n – heuristic analysis

- Είναι γνωστό ότι: $1 + x \approx e^x$ για $x \approx 0$
- Άρα: $1 - \left(1 - \frac{1}{n}\right)^{m|S|} \approx 1 - e^{-\frac{m|S|}{n}}$ αφού $\frac{1}{n} \approx 0$
- $\frac{n}{|S|} = b$: πλήθος bits του A που αντιστοιχούν σε κάθε κλειδί του $|S|$
- Πιθανότητα false positive $\approx \left(1 - e^{-\frac{m}{b}}\right)^m = f(m)$
- Βελτιστη τιμή για m για ελαχιστοποίηση false positives:
 - $f'(m^*) = 0$
 - $m^* = \ln 2 \cdot b$
- Πιθανότητα false positive $\approx \left(1 - e^{-\ln 2}\right)^{\ln 2 \cdot b} = \left(\frac{1}{2}\right)^{\ln 2 \cdot b}$
- Με βάση αυτή μπορούμε να ρυθμίσουμε το n .