ADT Λεξικού
Binary Search Trees, Hash Tables

Δημήτρης Φωτάκης, Αριστείδης Παγουρτζής,

Δώρα Σούλιου, Παναγιώτης Γροντάς

Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών

Εθνικό Μετσόβιο Πολυτεχνείο

ADT Λεξικού

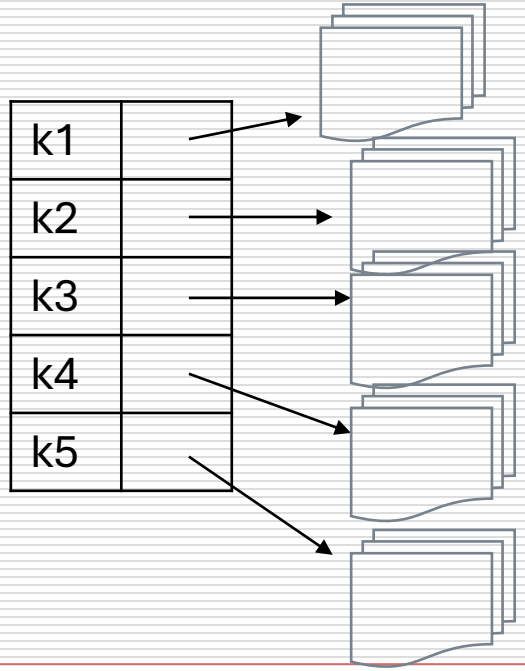
- Δυναμικά μεταβαλλόμενη συλλογή αντικειμένων που αναγνωρίζονται με «κλειδί» (π.χ. κατάλογοι, πίνακες ΒΔ).
- **Λεξικό** : συλλογή αντικειμένων με μοναδικό «κλειδί».
  - «Κλειδί»: αριθμός ή τύπος δεδομένων με ολική διάταξη.
- **Υποστηριζόμενες Λειτουργίες:**
  - Αναζήτηση στοιχείου με κλειδί k
    - member(k): 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: $O(1)$
  - 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(): $O(logn)$
  - 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) expected amortized
    - Καλή υλοποίηση
    - Μη παθολογικά δεδομένα!
  - Δεν υποστηρίζει αποδοτικά άλλες λειτουργίες.
  - Δυναμική – επιπλέον χώρος στον πίνακα (util ≈ 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 τεράστια ποσά δεδομένων

2-SUM Problem

- Είσοδος:

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

- Έξοδος:

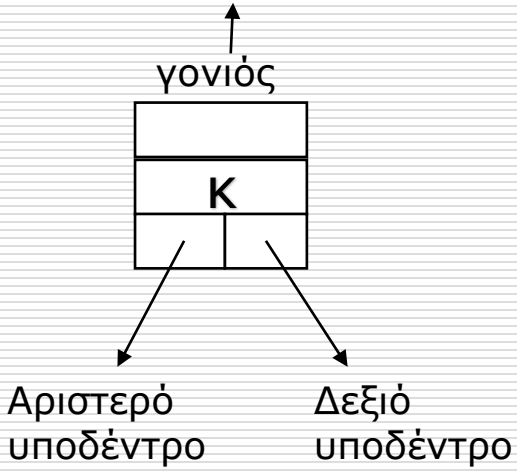
Υπάρχουν ακέραιοι $x, y \in A: x + y = t$?

- Λύσεις:
  - $\boldsymbol{\Theta(n^2)}$
  - $\boldsymbol{O(n \cdot logn)}$
  - $\boldsymbol{O(n)}$

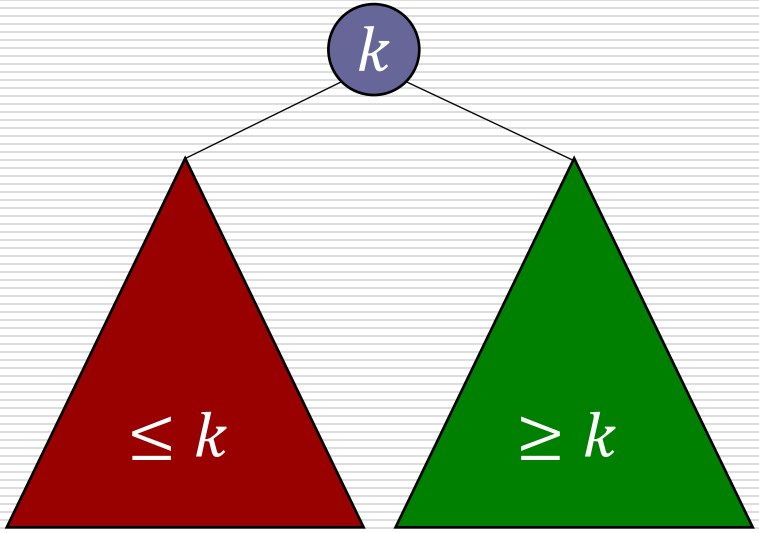
Binary Search Trees

Υλοποίηση - Ιδιότητα BST

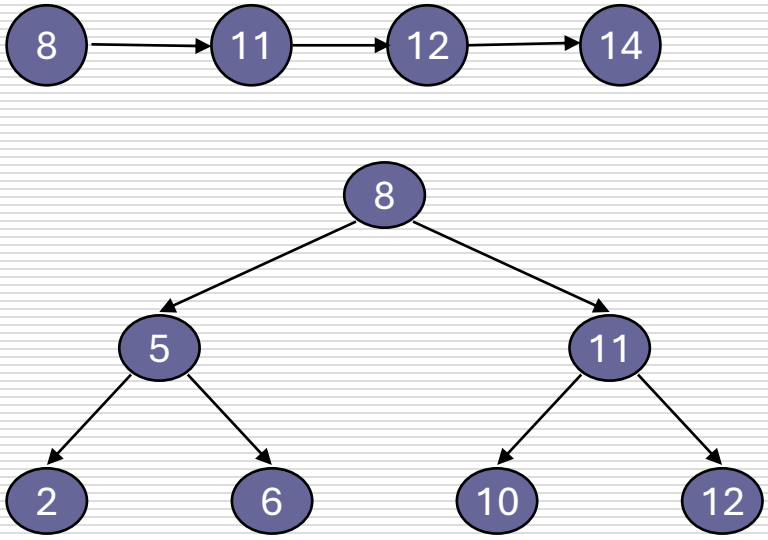
- Κόμβος
  - Κλειδί
  - *Γονικός Δείκτης*
  - Αριστερός Δείκτης
  - Δεξιός Δείκτης
- Δεν είναι κατ' ανάγκη πλήρη
  - Υλοποίηση με πίνακα - κενά

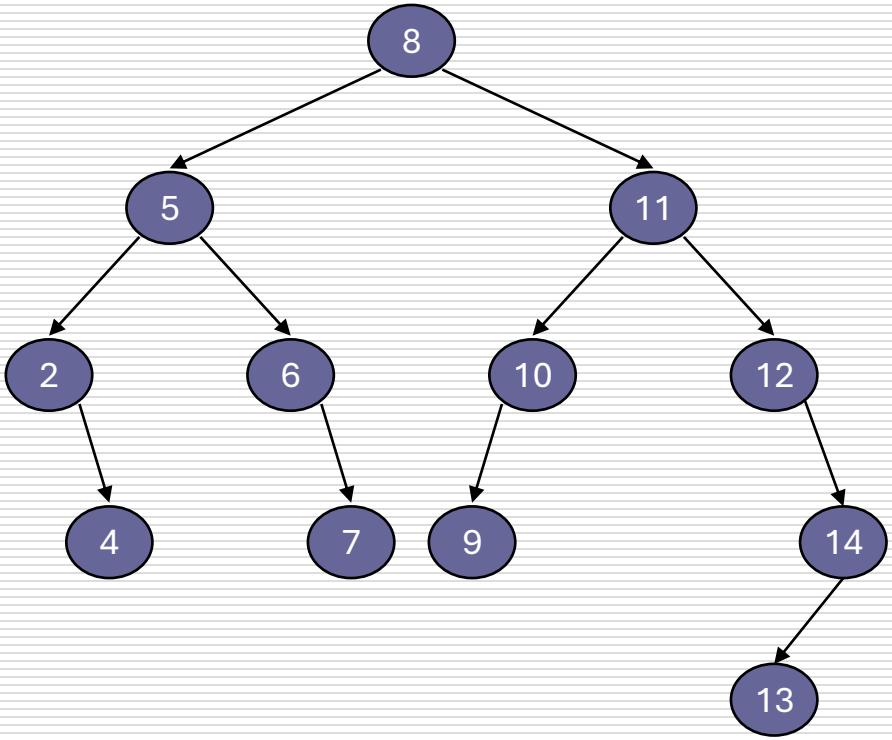


- $\forall k: x \in Left(k) \Rightarrow x \leq k$
- $\forall k: x \in Right(k) \Rightarrow x > k$



Παραδείγματα BST





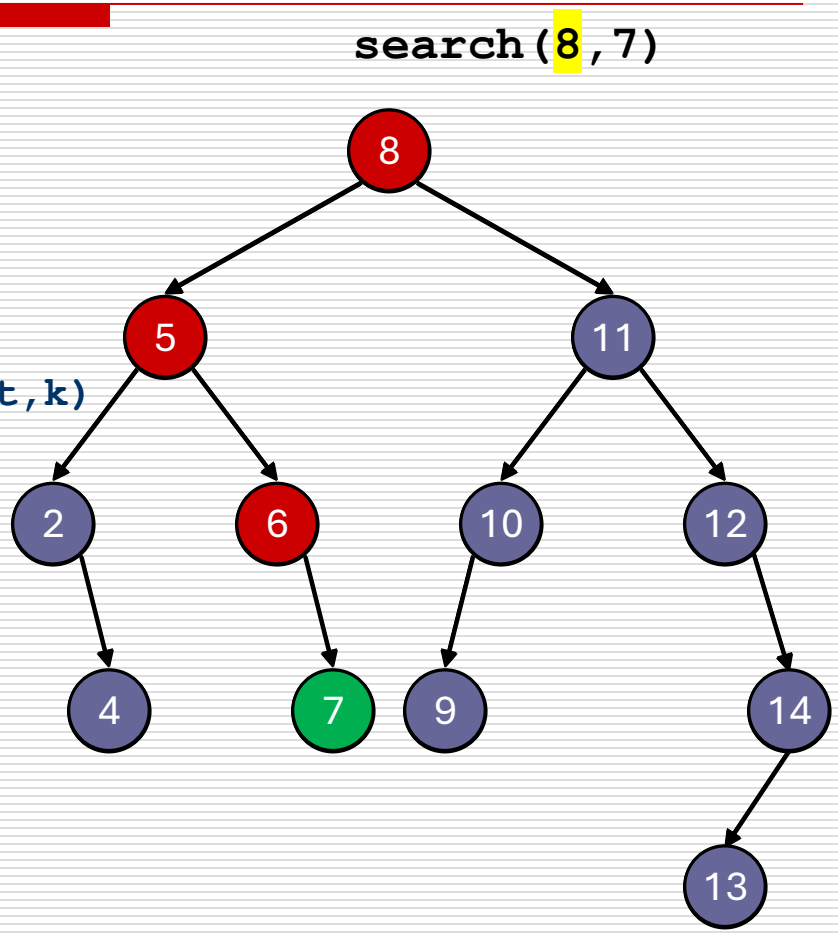
$\log_2(n+1) - 1 \leq$ ύψος $\leq n - 1$

Όλες οι λειτουργίες υλοποιούνται σε: $O(\text{ύψος})$
Αν balanced τότε: $O(logn)$

Υλοποίηση λειτουργιών – search

```
node search(node r, int k):
        if (r == null):
                return null
        if (r.key == k):
                return k
        if (r.key < k):
                return search(r.left,k)
        return search(r.right,k)
```

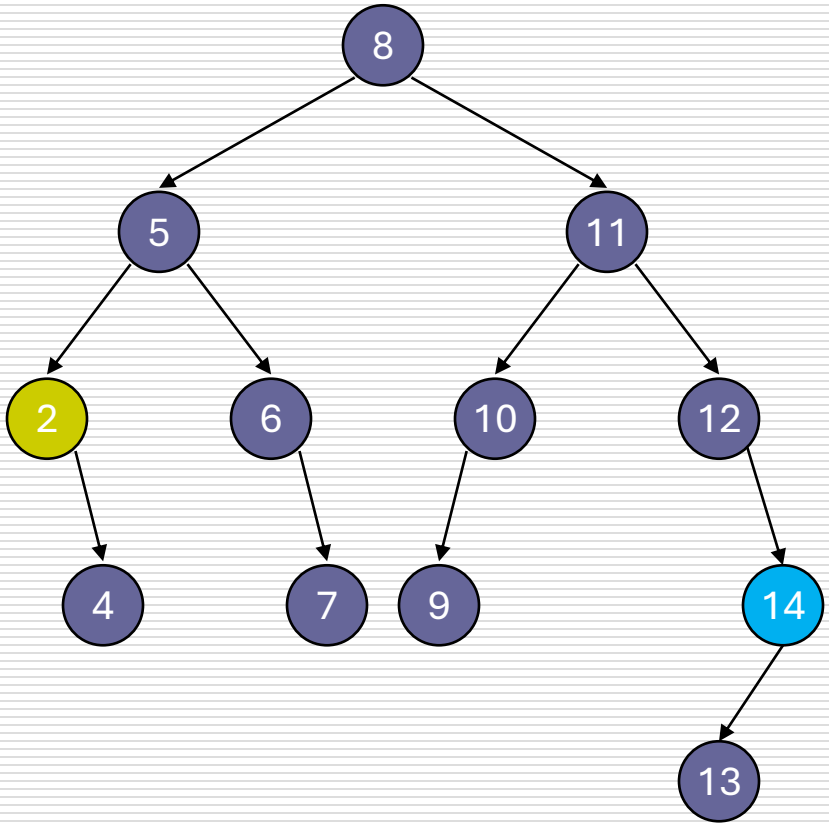
```
node search(node r, int k):
    x = r
    while x!=null and x.key!=k:
         if k < x.key:
           x = x.left
         else:
           x = x.right
    return null if x==null else x
```



Υλοποίηση λειτουργιών – min, max

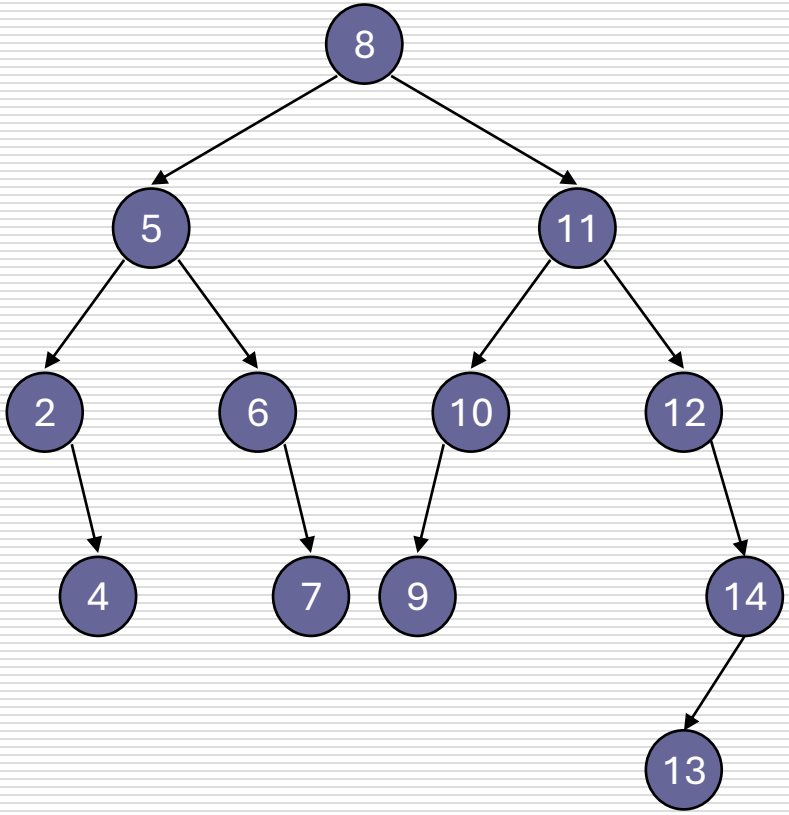
```
node min(node r):
        while (r.left != null):
              r=r.left
        return r.key
```

```
node max(node r):
        while (n.right != null):
              r=r.right
        return r.key
```



Υλοποίηση Λειτουργιών - outputSorted

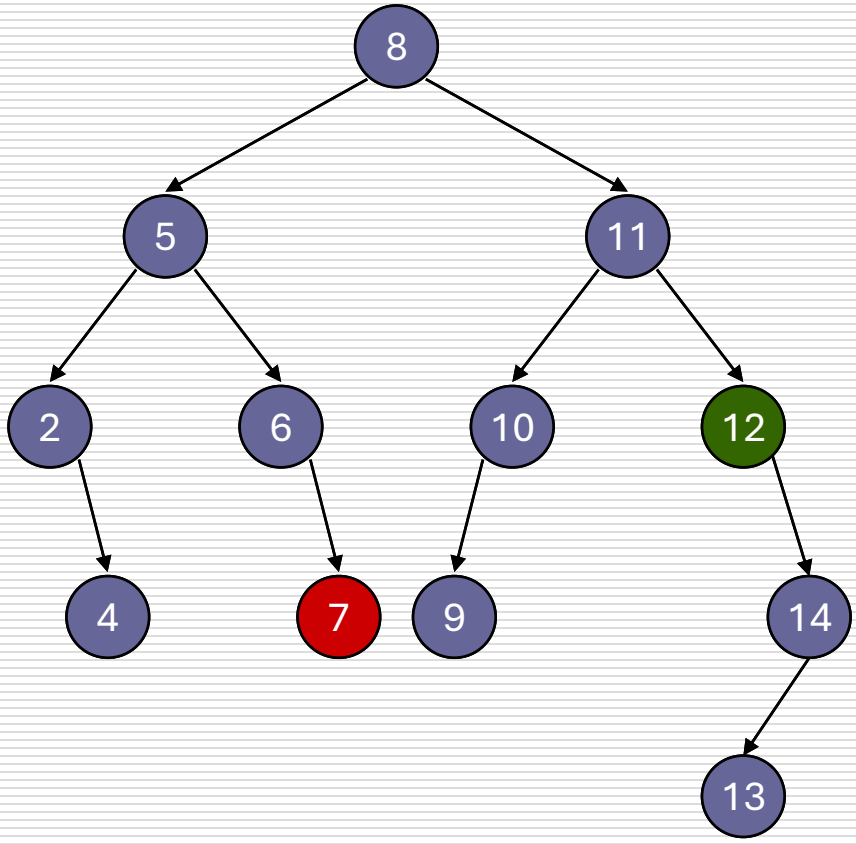
```
outputSorted(node r):
        outputSorted(r.left)
        return r.key
        outputSorted(r.right)
```



outputsorted(8) = [2,4,5,6,7,8,9,10,11,12,13,14]

Υλοποίηση λειτουργιών – predecessor

- Αποκλείεται το δεξί υποδέντρο
- Δύο περιπτώσεις:
  - Μη-κενό αριστερό υποδέντρο:
    - Το μέγιστο στοιχείο του
    - pred(8) → 7
  - Κενό αριστερό υποδέντρο:
    - Ο κοντινότερος πρόγονος με μικρότερο κλειδί
    - pred(13) → 12



Υλοποίηση λειτουργιών – predecessor

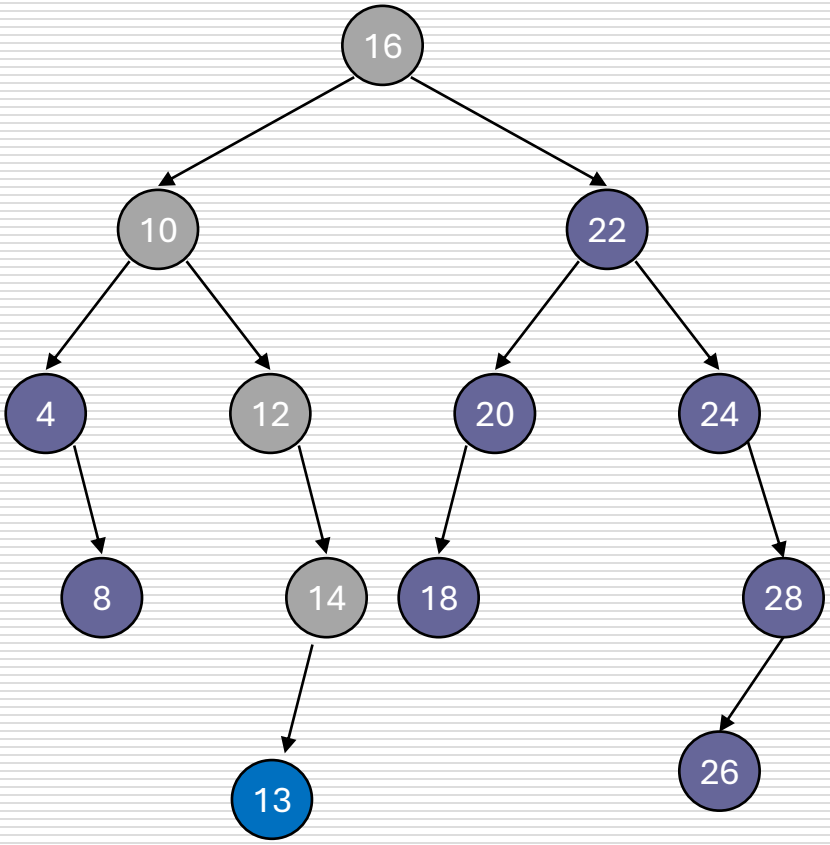
```
node pred(node k):
   if (k.left != null):     #μη-κενό αριστερό υποδέντρο
       return max(k.left)   #το μέγιστό του στοιχείο

   y = k.parent                  #κενό αριστερό υποδέντρο
   #ανεβαίνουμε από 'δεξιά'
   while (y!=null and k == y.left):
       k=y
       y=y.parent
   return y   #το τελευταίο ανέβασμα – από 'αριστερά' κλαδί
```

Εντελώς συμμετρικά για το successor (Άσκηση)

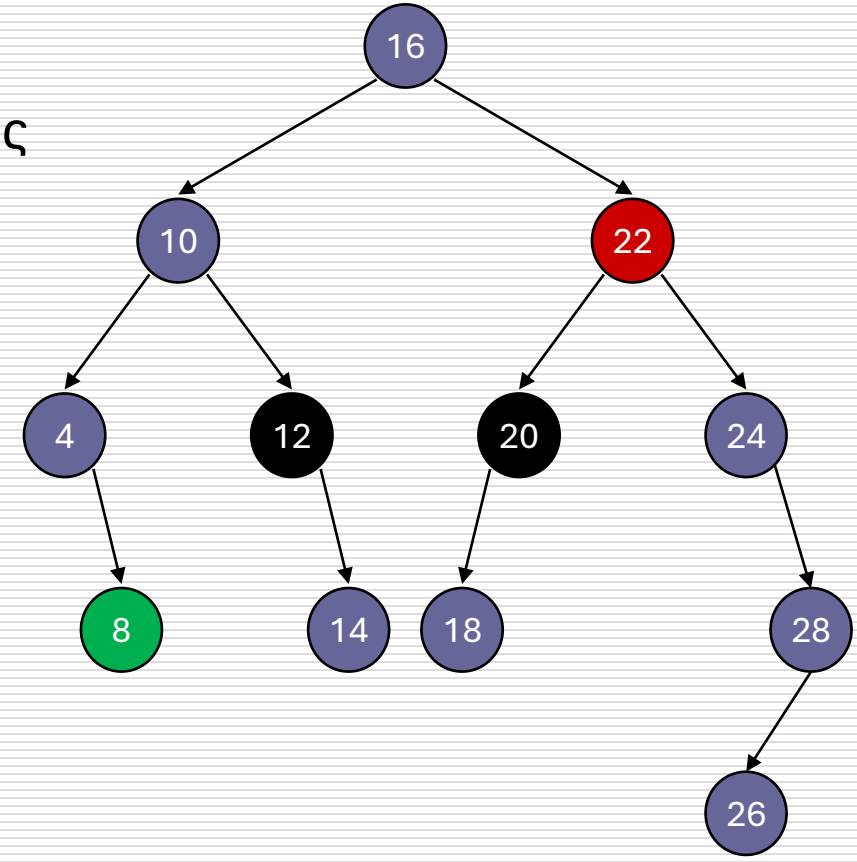
Υλοποίηση Λειτουργιών - insert

```
insert(node r, node k):
      x=r #διάσχιση του δέντρου
      y=null #γονιός του νέου
      while (x!=null):
          y=x
          if k.key<x.key:
             x=x.left
          else:
             x=x.right
      k.parent = y
      if y != null:
          if k.key < y.key:
             y.left = k
          elif k.key > y.key:
             y.right = k
```



Υλοποίηση Λειτουργιών - delete

- Αναζήτηση κόμβου
- Επιδιόρθωση ζημιάς διαγραφής
- Περιπτώσεις
  - 0 παιδιά
    - Απλή αφαίρεση
  - 1 παιδί
    - Αντικατάσταση με παιδί
  - 2 παιδιά
    - Αντικατάσταση με predecessor ή successor

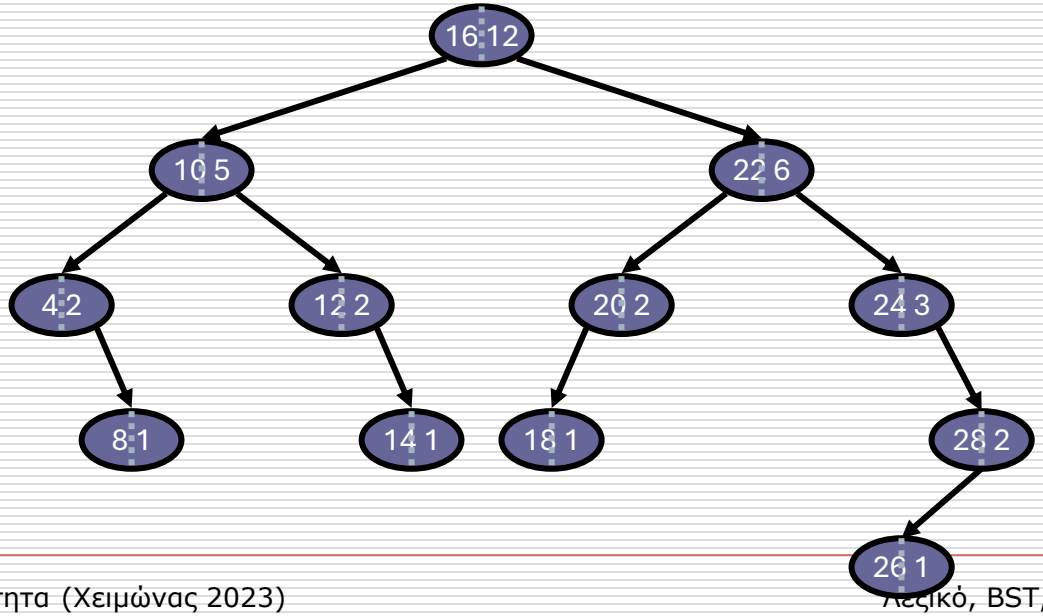


Υλοποίηση Λειτουργιών - delete

```
node delete(node r, int k):
  if (r == null): #locate node
     return null
  elif k < r.key:
     r.left = delete(r.left, k)
  elif k > r.key:
     r.right = delete(r.right,k)
  else: #delete node
     if r.left == null: #leaf / right
       t = r.right
       return t #connect right
     elif r.right == null:
       t = r.left
       return t #connect left
  else: #two children
     t=pred(r)
     r.key=t.key  #swap
     r.left = delete(r.left,t.key)
     return r
```

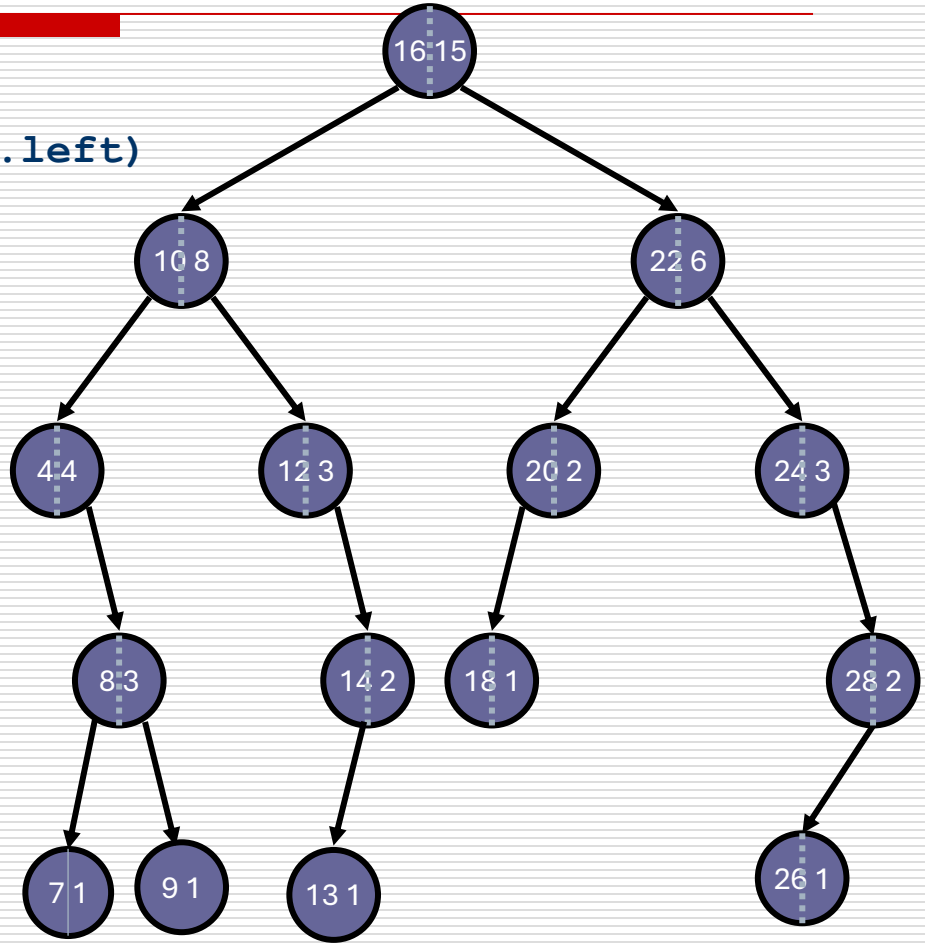
Order statistics BST

- Για υλοποίηση select / rank
- Νέο πεδίο $size$: περιέχει μέγεθος υποδέντρου κάθε κόμβου
- $size(p) = size(l) + size(r) + 1$
- Τροποποίηση λειτουργιών insert, delete (άσκηση)



Επιλογή (i-οστού μικρότερου) στοιχείου

```
node select(node r, int i):
  l = 0 if r.left == null else size(r.left)
  if i==l+1:
     return r.key
  elif i<l+1:
     return select(r.left,i)
  else:
     return select(r.right,i-(l+1))
```



Άσκηση:

`rank(node r, int k)`

Πόσα στοιχεία είναι το πολύ k

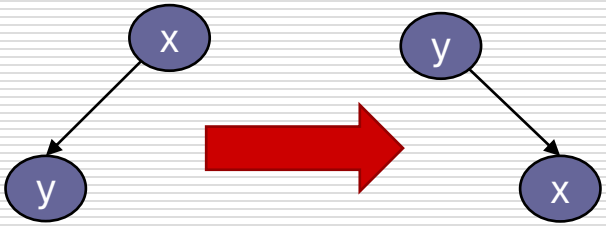
πχ. rank(16,21)→11

Balancing

- Επιπλέον δουλειά κατά την εισαγωγή / διαγραφή ώστε το BST να παραμένει ισοζυγισμένο
- … με αποτέλεσμα: $ύψος = O(logn)$
- Βασικό εργαλείο – Rotations: μετασχηματισμοί που διατηρούν τη βασική ιδιότητα BST
- 2-3 Trees, AVL Trees, B+ Trees

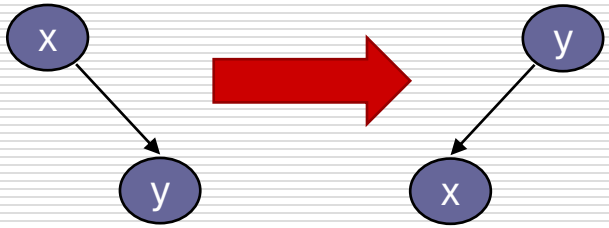
Balancing (2)

Right Rotation



$y < x$

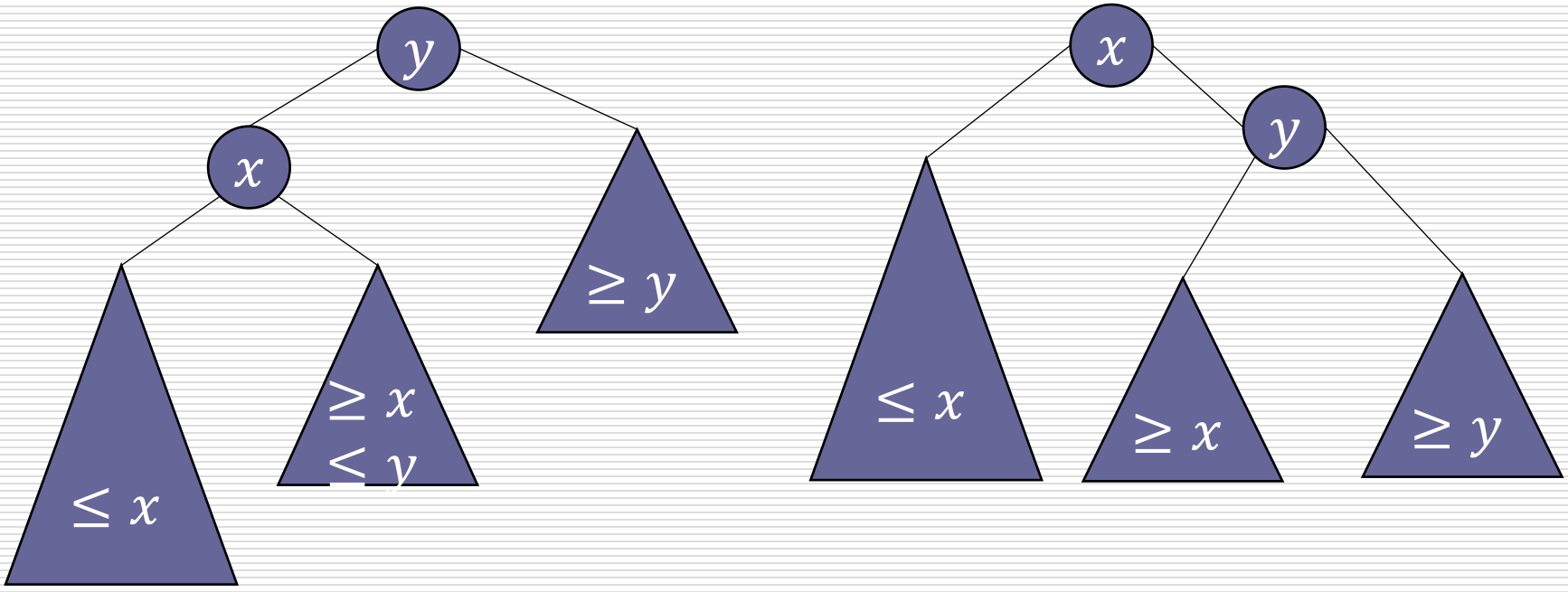
Left Rotation



$x < y$

Διατήρηση ιδιότητας BST

Balancing (3)



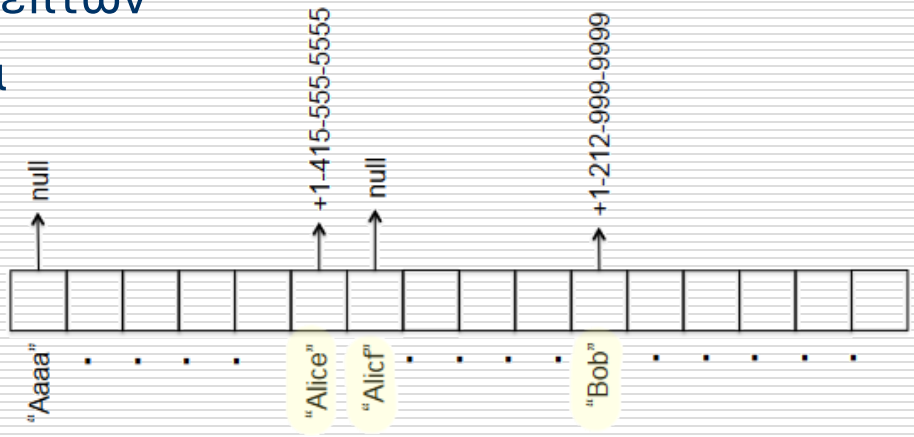
Hash Tables

Hash Tables

- Γενίκευση των arrays
  - Random access O(1)
  - Χωρίς ακέραιους δείκτες
  - Με δυναμική αλλαγή μεγέθους
- Υποστηριζόμενες λειτουργίες
  - Lookup(k): O(1) *
  - Insert(k): O(1)
  - 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 25 χαρακτήρων (26^{25}).
- Χρησιμοποιείται ένα σύνολο S από **πραγματικά** κλειδιά
  - Διευθύνσεις IP επισκεπτών
  - Πραγματικά ονόματα
- $|S| \ll |U|$



Διαφορές Υλοποίησης

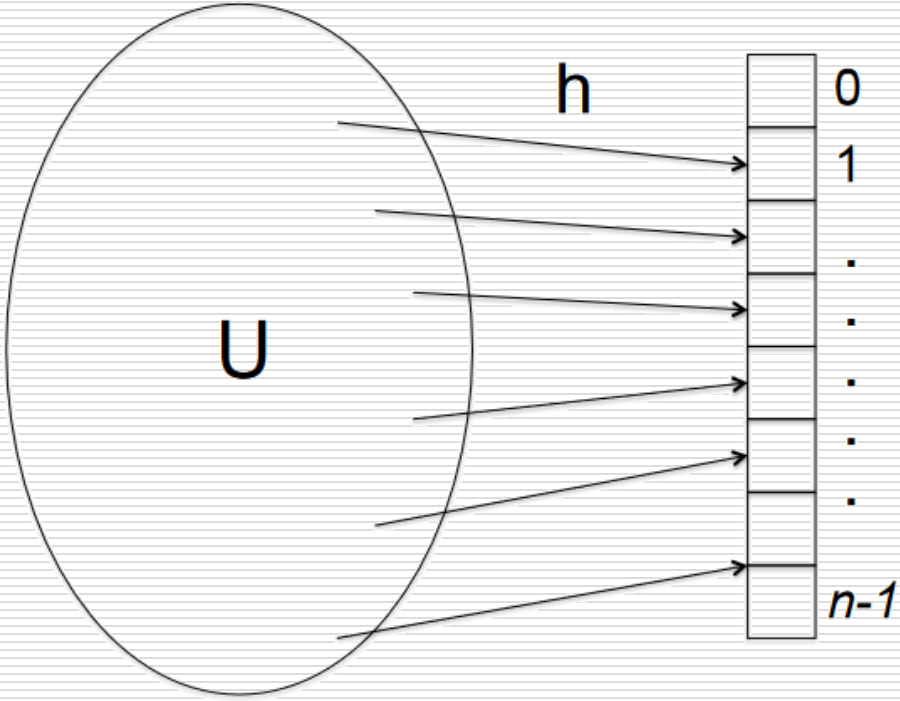
Δομή	Χώρος	Χρόνος Lookup(k)
Πίνακας	$\Theta(\|U\|)$	$O(1)$
Γραμμική Λίστα	$\Theta(\|S\|)$	$\Theta(\|S\|)$
Hash-Table	$\Theta(\|S\|)$	$\Theta(1)$

Hash Tables: The best of both worlds

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

Hash Functions

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

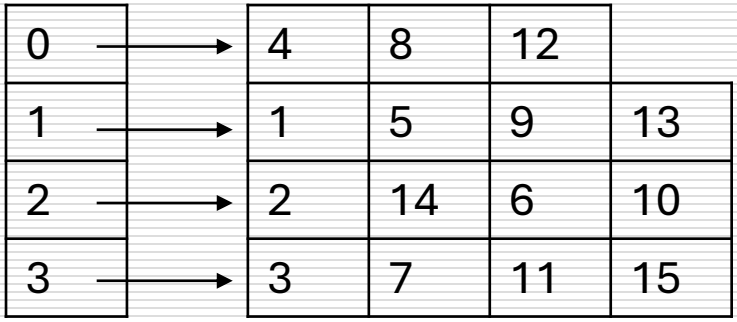


Collision Resolution - Chaining

- 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

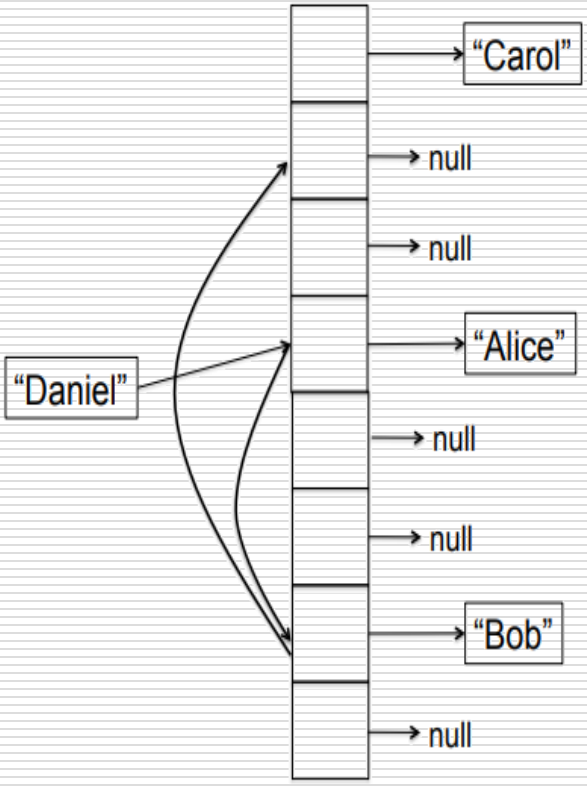
Παράδειγμα:

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



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 $h(k)$
  - $\{h(k), h(k)+1, h(k)+2, \cdots\} \bmod n$
  - Χρήση probe function $\text{pos}(k,i) = h(k) + f(i) \bmod n$
- Double Hashing
  - $\text{pos}(k,i) = h_1(k) + i \cdot h_2(k), i \geq 0$
  - $h_1(k)$: Αρχική θέση
  - $h_2(k)$: Βήμα Μετατόπισης (σε περίπτωση κατειλημμένης θέσης)
  - Παράδειγμα:
    - $h_1(k) \rightarrow 20$
    - $h_2(k) \rightarrow 43$
    - Probing sequence: $\{20,63,106, \cdots\}$

Ποιότητα hash functions

- Θα διαλέγατε ποτέ την συνάρτηση $h(k) = 0, \forall\, k \in U$;
- Όμως λόγω γενίκευσης αρχής περιστεριώνα:
  - $\forall h: U \rightarrow [n], \exists\, S \subset U: h(k_1) = h(k_2), \forall k_1, k_2 \in S$
- Όλες οι λειτουργίες του Hash Table είναι O(1) υπό την υπόθεση μιας καλής hash function.
- Μήπως να διαλέξουμε μια τυχαία συνάρτηση;
  - $\Pr[h(k) = h_0] = \frac{1}{n}\ \forall k \in |U|$
  - Random oracle - Independent Uniform Hashing
  - Δύσκολο στην αποθήκευση και αποτίμηση
- Απαιτούμενα χαρακτηριστικά h:
  - Αποτίμηση σε O(1)
  - Αποθήκευση O(1)
  - 'Μοιάζει' με τυχαία συνάρτηση *(~ομοιόμορφη κατανομή κλειδιών)*

Εκτίμηση Ποιότητας

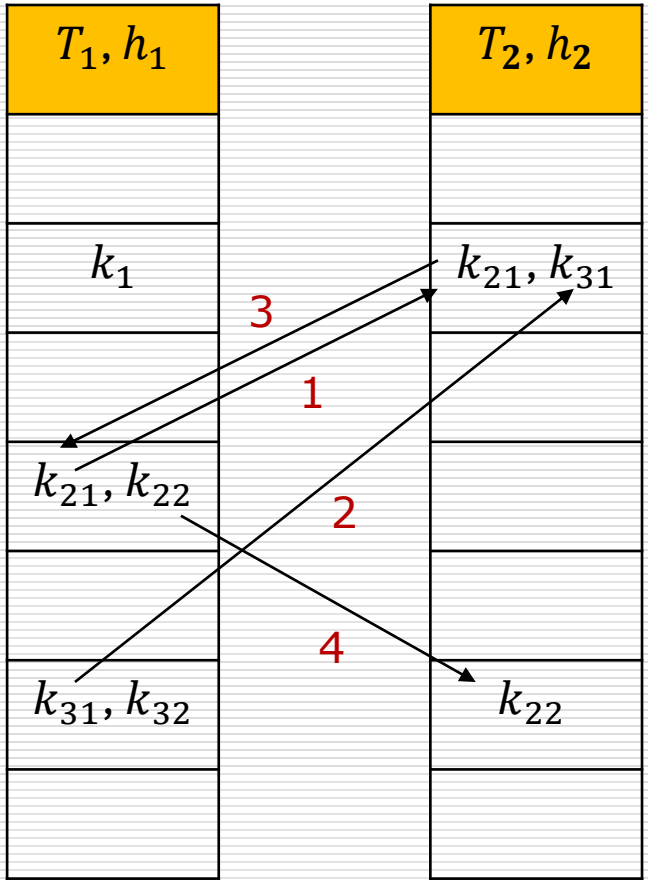
- Φόρτος hash table
  - α = πλήθος_αποθηκευμένων_κλειδιών / πλήθος_θεσεων
- Εκτίμηση χειρότερης περίπτωσης lookup
  - Chaining: $O(1+\alpha)$
    - α - αναμενόμενο μέγεθος bucket $\alpha \geq 1$
  - Open Addressing: $O(\frac{1}{1-\alpha})$
    - α – πιθανότητα hash σε κατειλημμένη θέση $\alpha \leq 1$
    - $1+\alpha+\alpha^2+\cdots=\frac{1}{1-\alpha}$

Πρακτικά θέματα

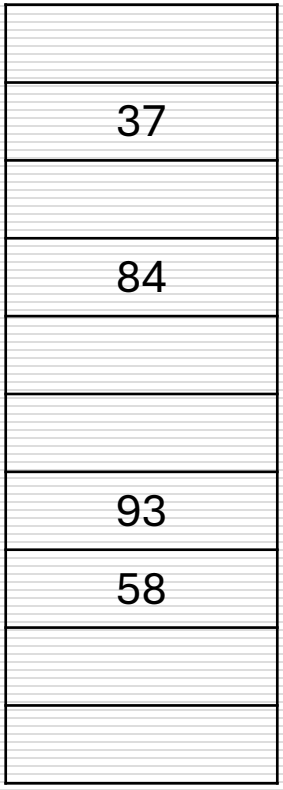
- Table – Resizing (Rehashing)
  - Όταν α > threshold: αλλαγή μεγέθους πίνακα
  - Συνήθως διπλασιάζουμε το μέγεθος
  - Επιλογή νέας συνάρτησης hash
- Cryptographic hash functions
  - (Υπολογιστική) Δυσκολία εύρεσης συγκρούσεων
  - Δεν είναι O(1) η αποτίμηση
    - Όμως είναι πολύ γρήγορη (υποστήριξη και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

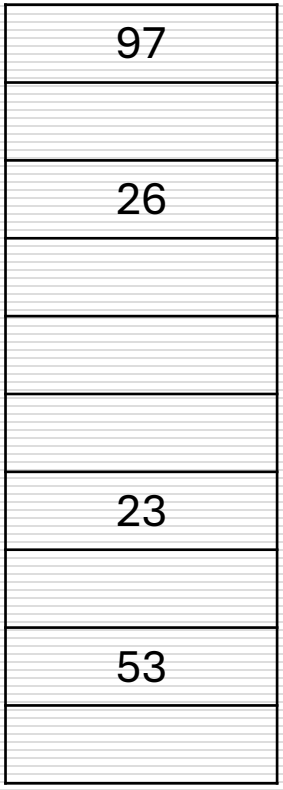
Cuckoo Hashing (Pagh, Rodle 2001)

- Χρήση δύο συναρτήσεων h_1, h_2
- ... και δύο πινάκων T_1, T_2
- Χρήση h_1 → τοποθέτηση σε T_1
- 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 κανονικά (διώχνουμε προηγούμενο)
- Rehash όταν έχω υπέρβαση ορίου ταξιδιών - κύκλο

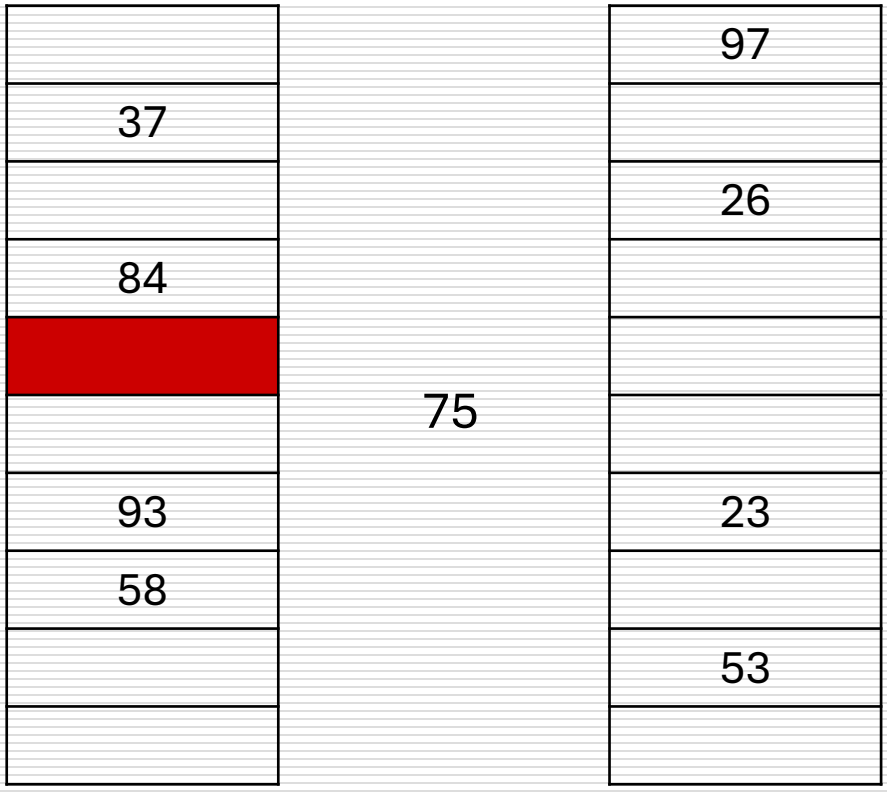


Cuckoo Hashing - Παράδειγμα





Cuckoo Hashing - Παράδειγμα

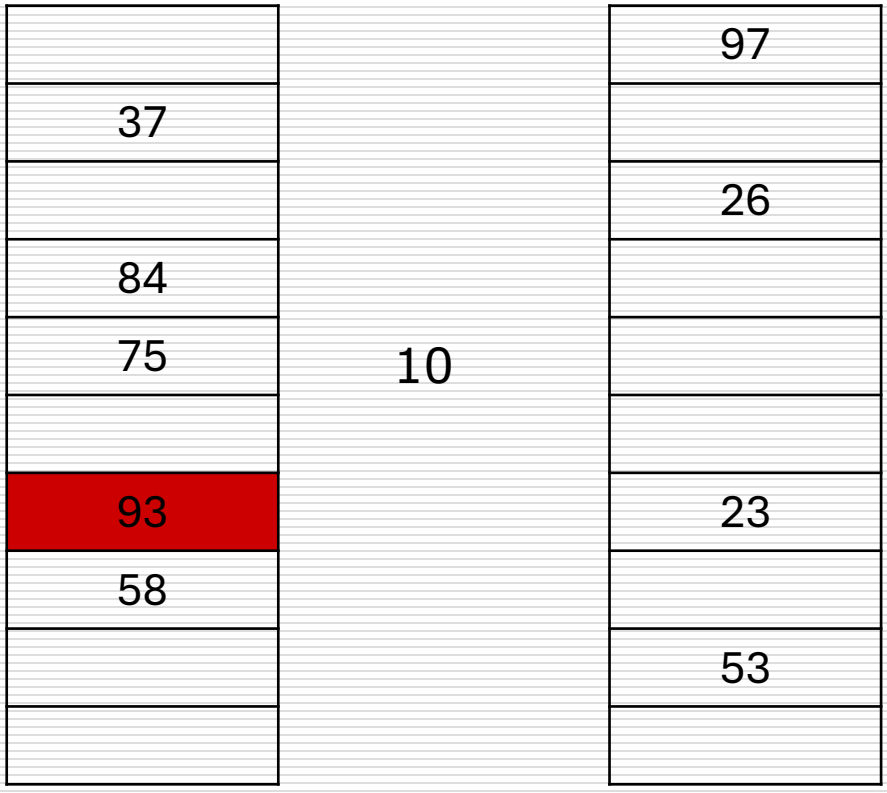


Cuckoo Hashing - Παράδειγμα

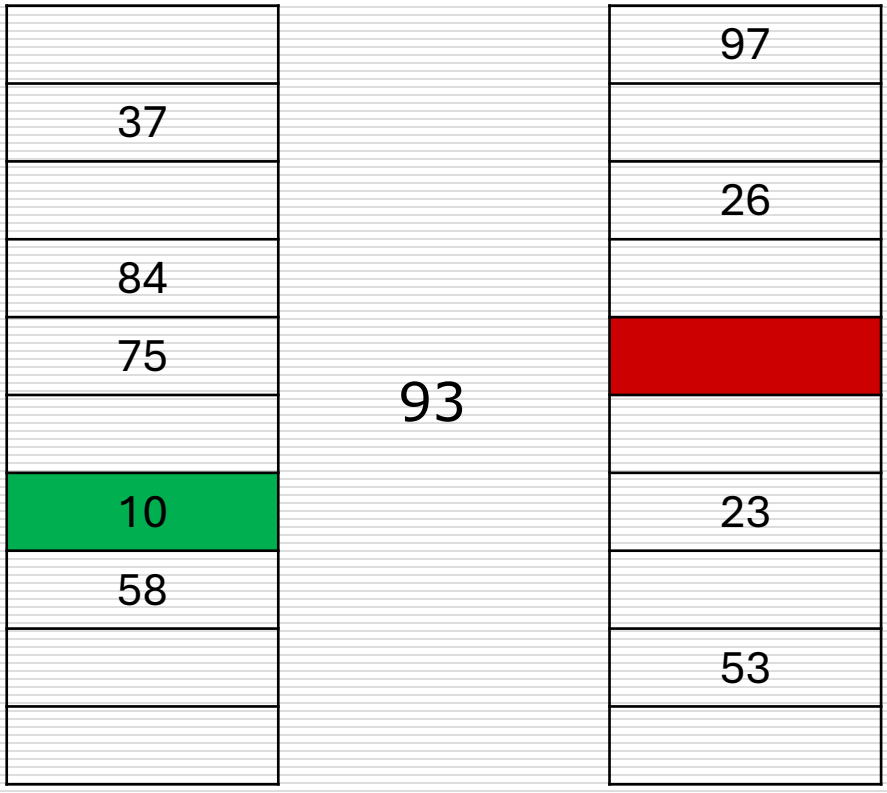
37
84
75
93
58

97
26
23
53

Cuckoo Hashing - Παράδειγμα



Cuckoo Hashing - Παράδειγμα

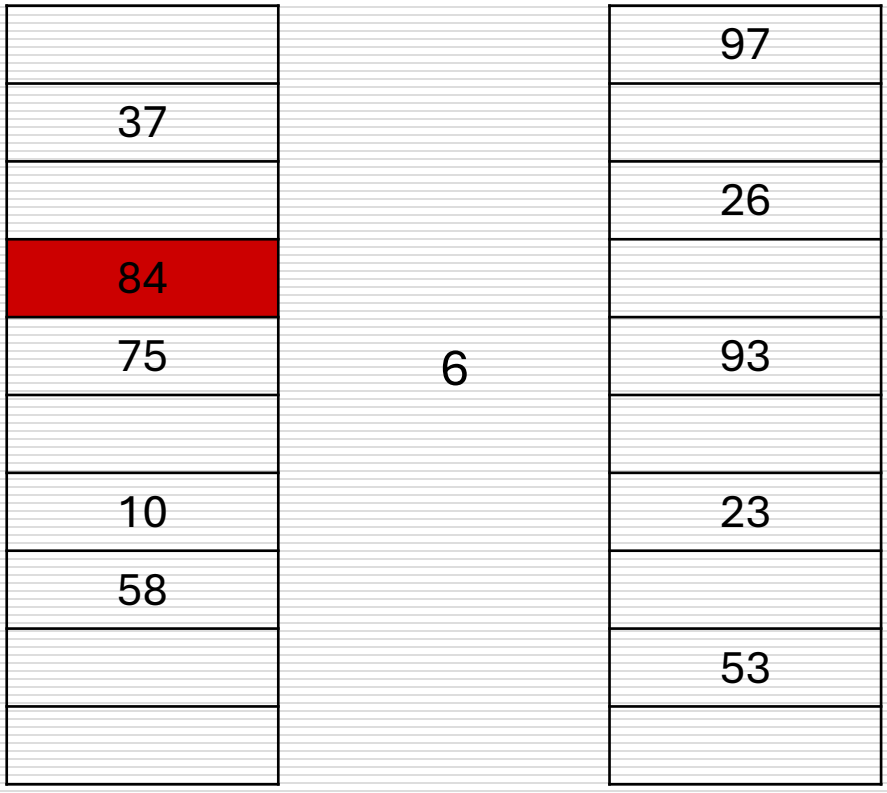


Cuckoo Hashing - Παράδειγμα

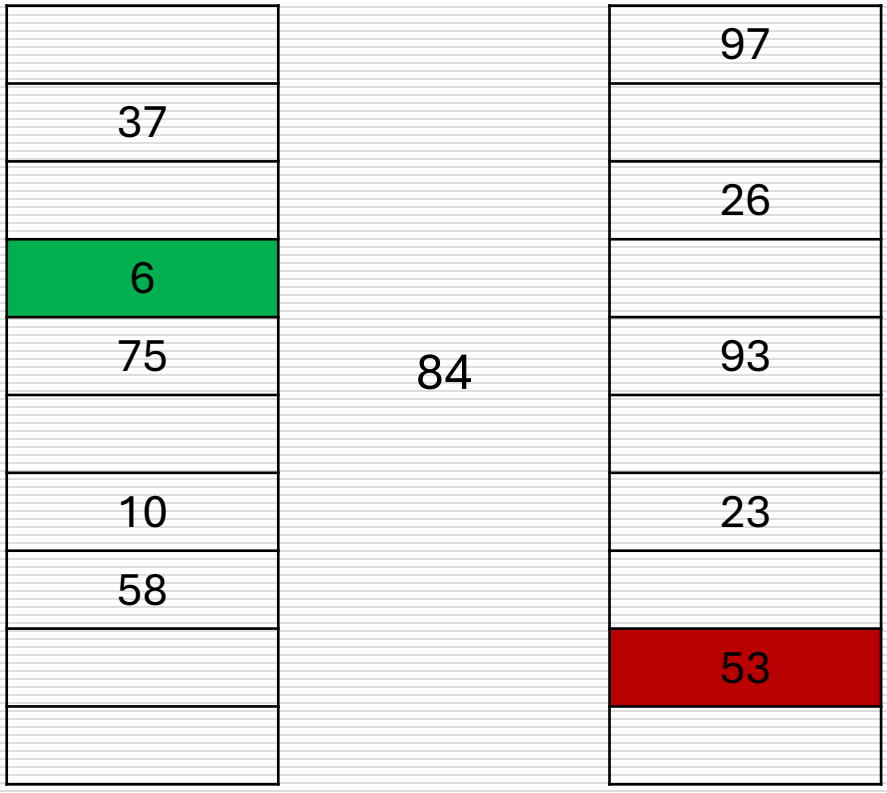
37
84
75
10
58

97
26
93
23
53

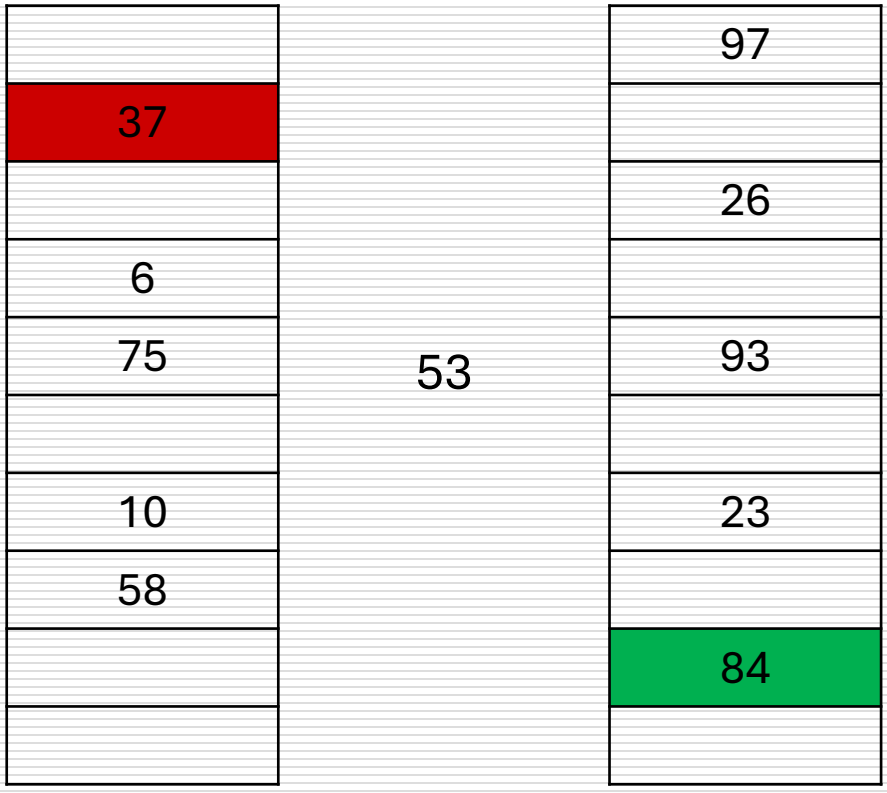
Cuckoo Hashing - Παράδειγμα



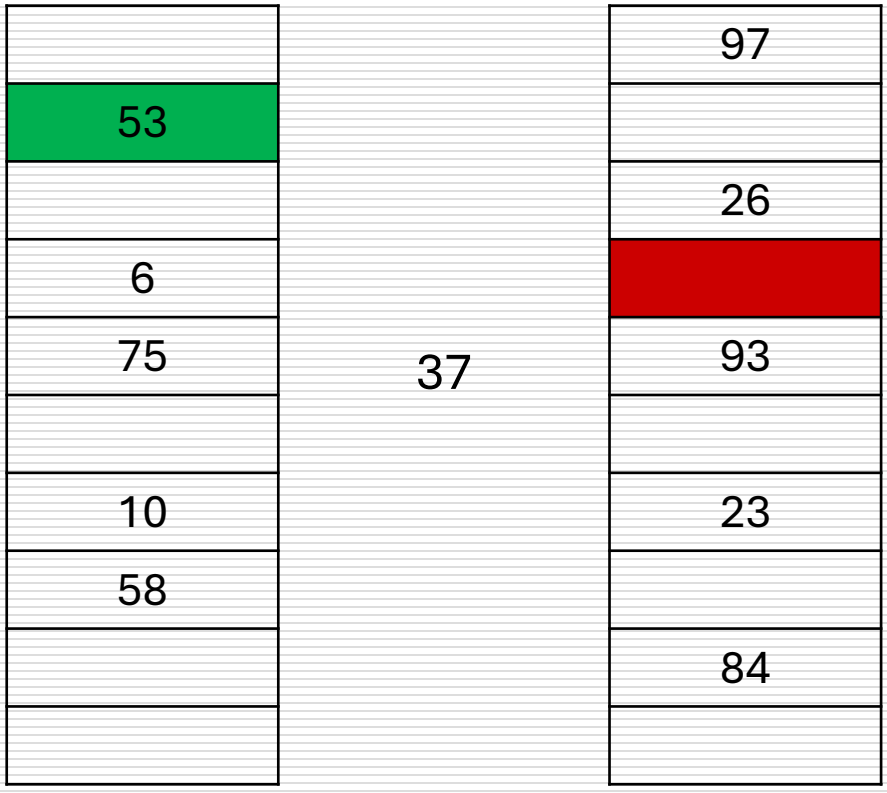
Cuckoo Hashing - Παράδειγμα



Cuckoo Hashing - Παράδειγμα



Cuckoo Hashing - Παράδειγμα

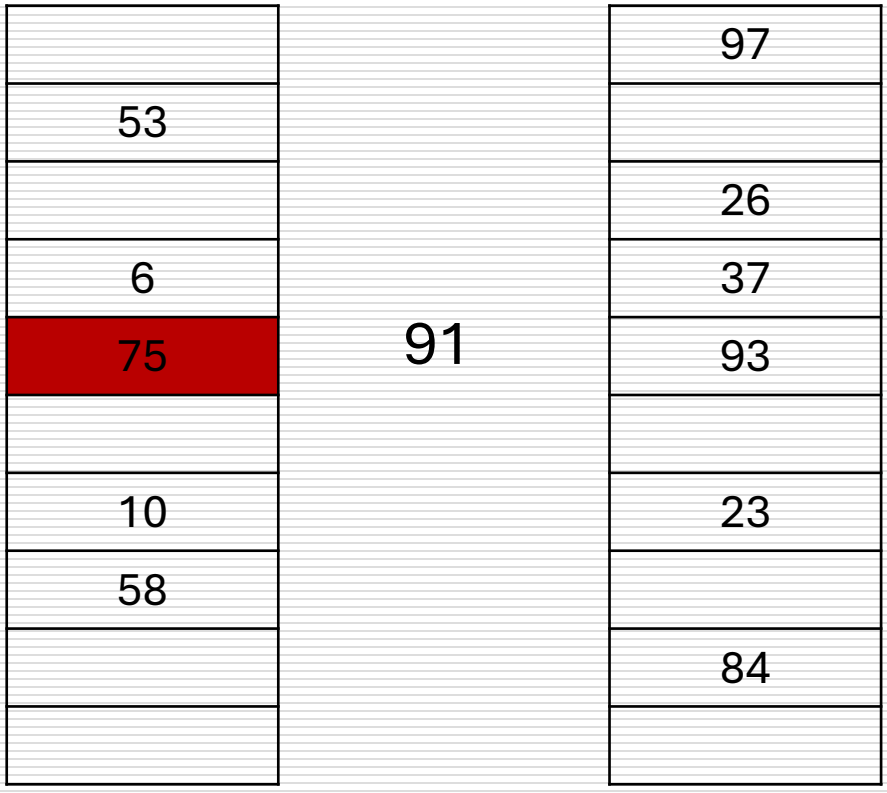


Cuckoo Hashing - Παράδειγμα

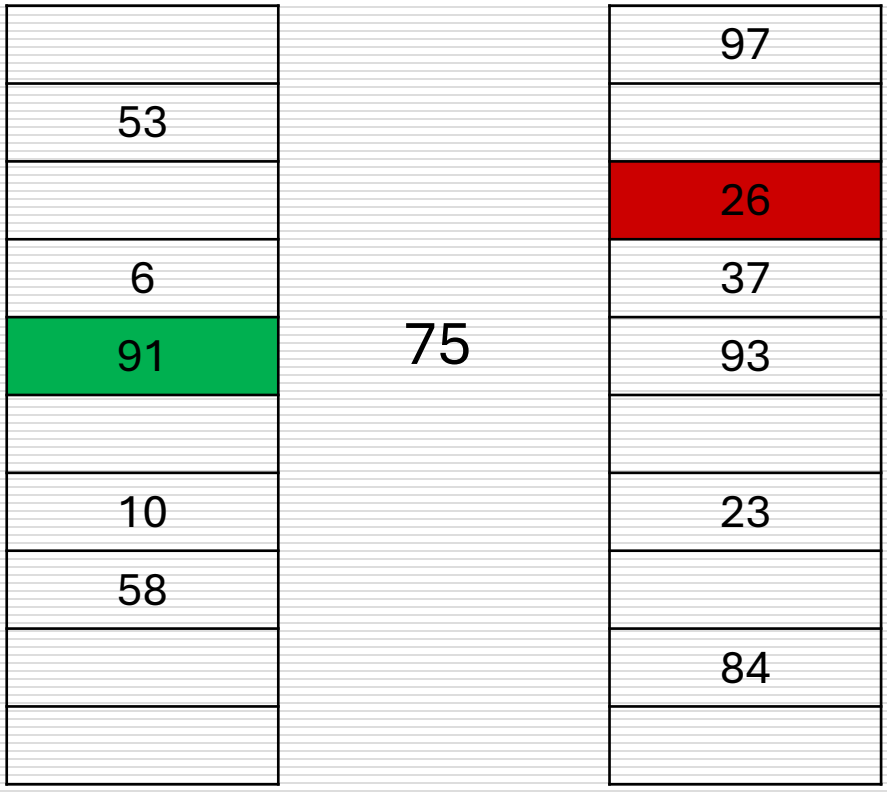
53
6
75
10
58

97
26
37
93
23
84

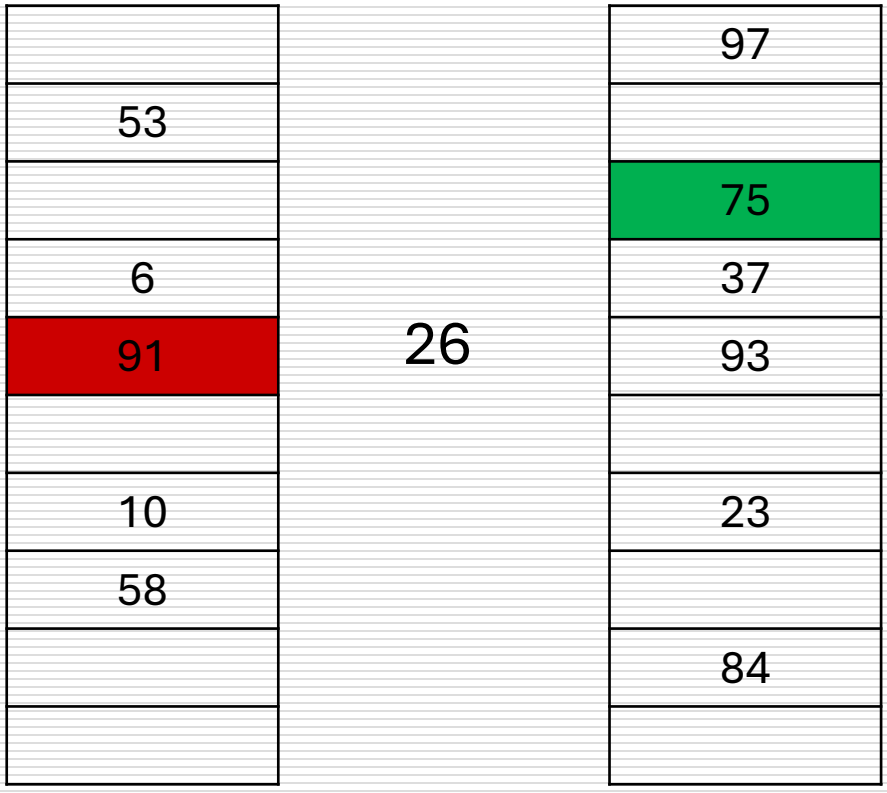
Cuckoo Hashing - Παράδειγμα



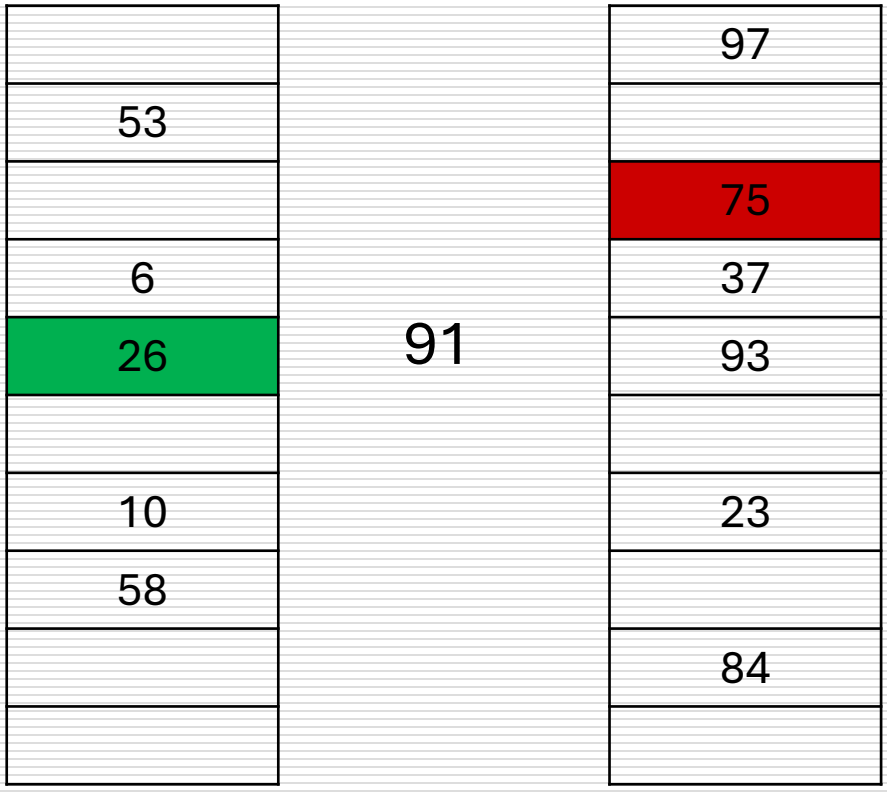
Cuckoo Hashing - Παράδειγμα



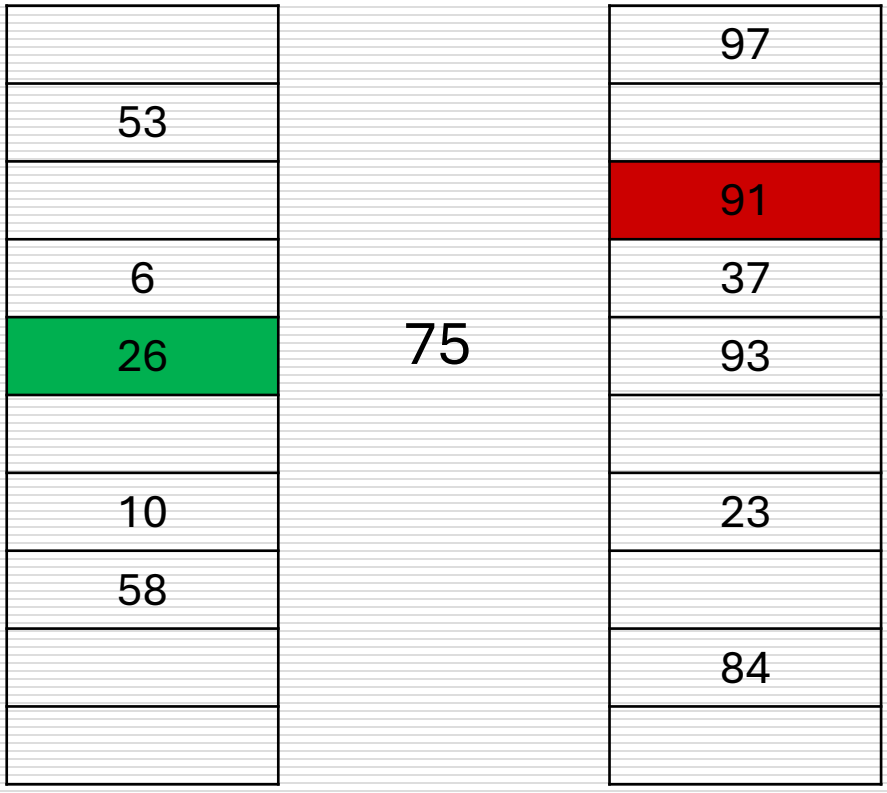
Cuckoo Hashing - Παράδειγμα



Cuckoo Hashing - Παράδειγμα



Cuckoo Hashing - Παράδειγμα



Cuckoo Hashing (υλοποίηση)

lookup(k) :

```
i=h_1(k)
if T_1[i]==k:
    return 1,i
else:
    i=h_2(k)
    if T_2[i] == k:
      return 2,I
    else
      return null
```

O(1) – ελέγχος 2 θέσεων

delete(k) :

```
i=h_1(k)
if T_1[i]==k:
   T_1(i) = null
else:
    i=T_2[i]
    if T_2[i] == k:
      T_2[i] = null
```

O(1)

insert(k) :

```
if T_1[i]==k or T_2[i] == k:
        return
for i in range(max):
     swap(k, T_1[h_1(k)])
    if k==null return
    swap(k, T_2[h_2(k)])
    if k==null return
rehash()
insert(k)
```

expected O(n) για n εισαγωγές

Πιθανοτικά Hash Tables: Bloom Filters

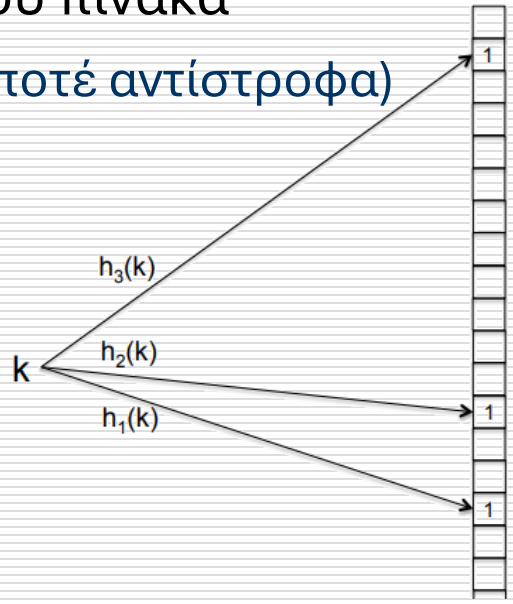
- Lookup(k) ∈ {False, True}
  - Επιστρέφει ύπαρξη και όχι θέση.
- False Positive: Υπάρχουν σε μικρό βαθμό.
  - Αν δεις True, δεν είναι βέβαιο ότι υπάρχει αντικείμενο με κλειδί k.
  - Πιθανότητα λάθους - μικρή
- False Negative: Δεν υπάρχουν.
  - Αν δεις False ξέρεις σίγουρα ότι δεν υπάρχει αντικείμενο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

Bloom Filters – Βασική υλοποίηση

```
insert(k):
   for i in range(m):
         A[h_i(k)]=1

bool lookup(k):
    for i in range(m):
         if A[h_i(k)]=0:
              return False
    return True
```

- Πίνακας A n-bit (n θεσεων)
- m hash functions (σταθερός αριθμός)
- Στοιχεία του πίνακα
  - 0 → 1 (ποτέ αντίστροφα)



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