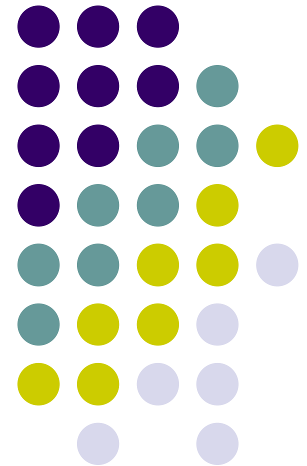
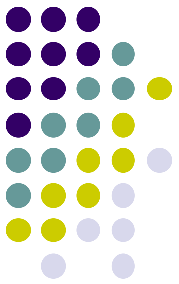
Κατακερματισμός (Hashing)



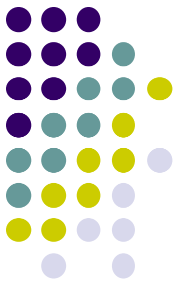
Γιάννης Κωτίδης

What is hashing?

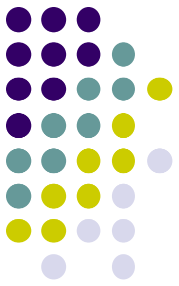


- Hashing generally takes records whose key values come from a large range and maps those records in a "hash table" with a relatively smaller number of slots called buckets
- Collisions occur when two records with different keys hash to the same bucket

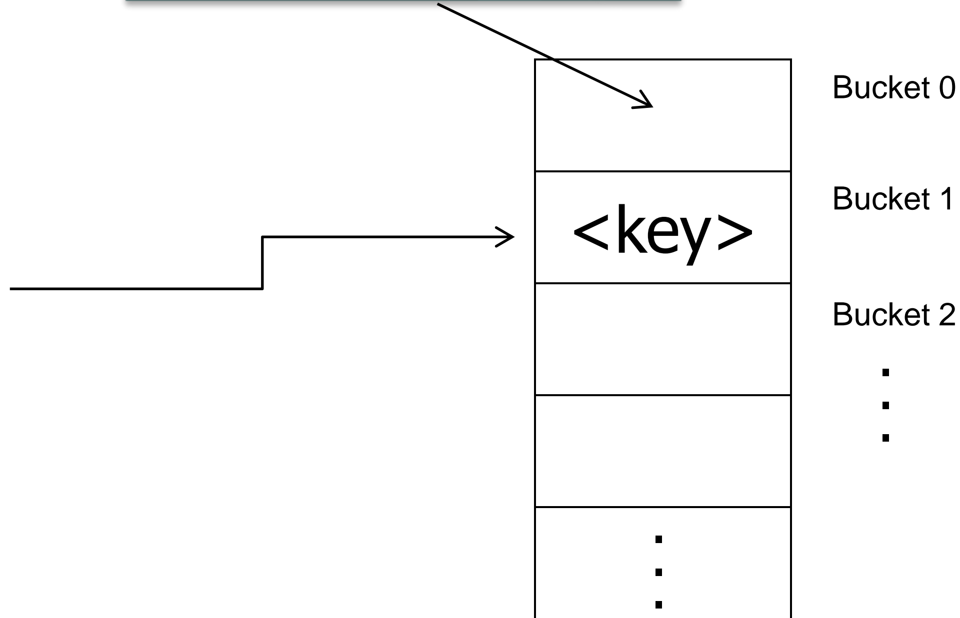
Hash function h()

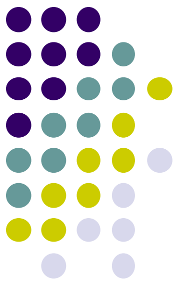


- Maps arbitrary items (keys) into integers
  - We can limit the number of buckets by using modulo arithmetic: bucket= h() % N
- A good hash function should:
  - be easy (fast) to compute
  - provide a uniform distribution across the hash table and should not result in clustering of keys (unless this is desirable for the applications)
  - avoid collisions (to the extend possible)



Hashing for indexes



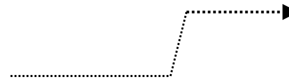


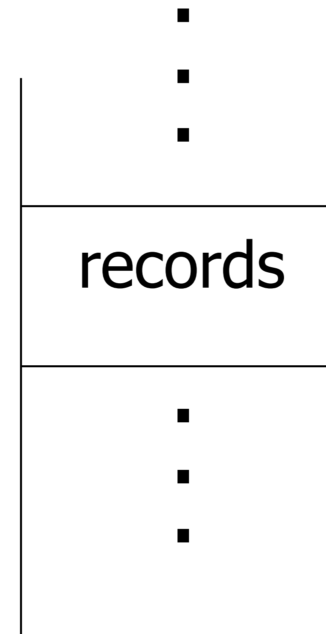
$1^{η}$ Εναλλακτική:

Η τιμή h(key) μας οδηγεί στο bucket όπου βρίσκεται η εγγραφή*

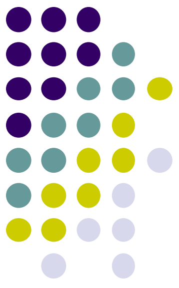
*σε κάποια ερωτήματα ενδέχεται η τιμή που ψάχνουμε να μην υπάρχει

key → h(key)





Σε αυτή την περίπτωση η **θέση** (κάδος) της εγγραφής στο δίσκο καθορίζεται από την τιμή της συνάρτησης h(). Ο κατακερματισμός λειτουργεί όπως ένα **ευρετήριο συστάδων** (clustering index).

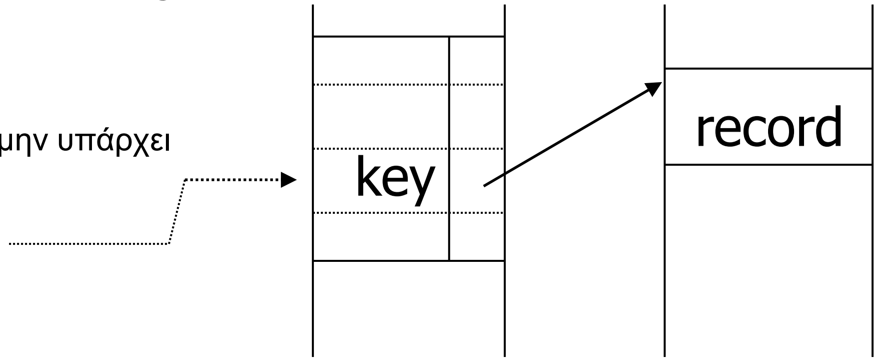


$2^{η}$ Περίπτωση:

Η τιμή της συνάρτησης μας οδηγεί σε κάδο στο δίσκο στον οποίο υπάρχει η τιμή του κλειδιού και αναφορά προς τη θέση της εγγραφής*

*σε κάποια ερωτήματα ενδέχεται η τιμή που ψάχνουμε να μην υπάρχει

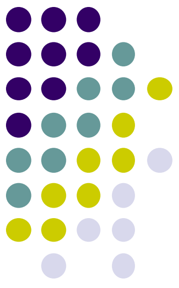
$$key \rightarrow h(key)$$



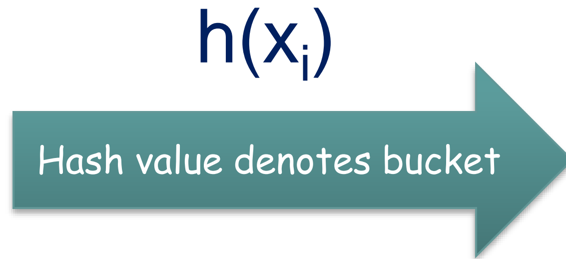
Index

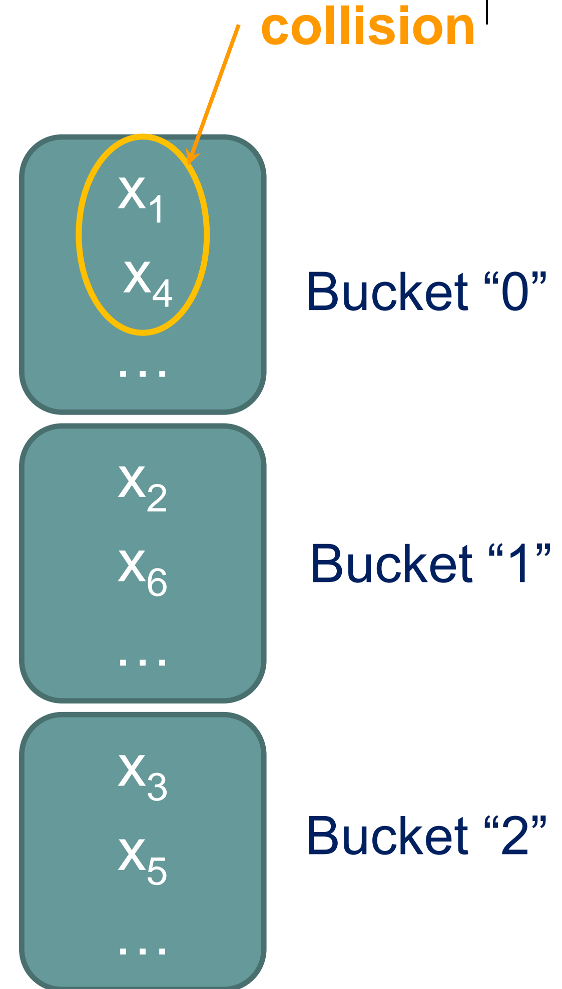
Ο κατακερματισμός λειτουργεί ως **δευτερεύον ευρετήριο** (secondary index)

Example: h(x)=(2x+1) % 3

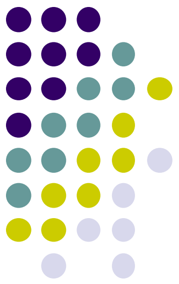


$x_1=1$
$x_2=3$
$x_3=2$
$x_4=7$
$x_5=5$
$x_6=9$



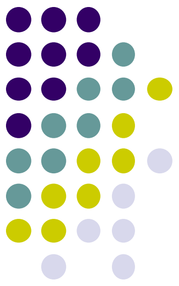


Careful



- Often key values exhibit skew
  - *Age* of my customers used as index key. But most of my customers are young
- Prefer hash functions that distribute records uniformly among the buckets
- Example: want to hash strings extracted from a document

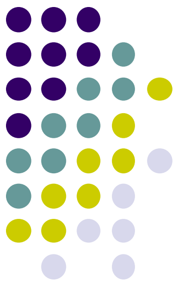
Όχι καλή συνάρτηση κατακερματισμού: h(s) = length of string s



- English Word Length Distribution:

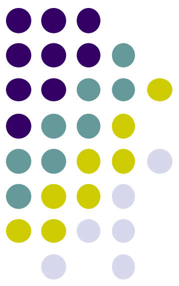
A better function (input string s)

ΕΚΤΟΣ ΥΛΗΣ



- s[i] = i^{th} character in string
- K=some prime number
- Recursively compute
  - h(1) = s[1] /*** first character in string ***/
  - h(i) = h(i-1)*K + s[i] , for i>1
- **Return h[length(s)]**

ΕΚΤΟΣ ΥΛΗΣ



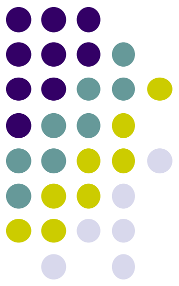
Example for s='abc'

- h(1)=a
- h(2)=h(1)K+b=aK+b
- h(3)=h(2)K+c=aK^2+bK+c
- Thus:

h('abc')=aK^2+bK+c

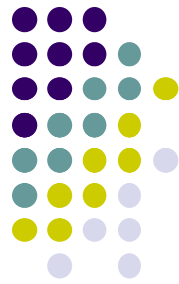
Example continued (assume K=31)

ΕΚΤΟΣ ΥΛΗΣ



- Ascii codes of ‘a’,’b’ and ’c’ are 97, 98 and 99, respectively
- h(‘abc’) = (a*K+b)*K+c = aK^2+bK+c =97*31^2+98*31+99=96354
- Another example: h(‘acb’) = … = 96384

ΕΚΤΟΣ ΥΛΗΣ

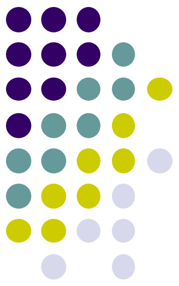


Think

- Previous function may return arbitrary large numbers
  - h('supercalifragilisticexpialidocious')= 3892360994585874516170035123354428844321330295603161825327689504791395104502384955 (for K=257)
- Quite often you want to restrict the range of buckets in an implementation
  - For instance assume you want to create N=1024 buckets
  - How to modify the hashing function?

ΕΚΤΟΣ ΥΛΗΣ

Universal Hashing

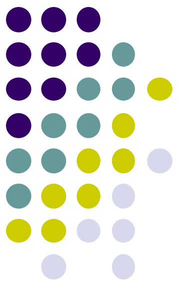


- Informally: derive a family of hash functions H with low probability of collisions
- Assume keys (data) are drawn from a universe U and there are m slots in the hash table.
- For every hash function h∈H, the following property should hold:

$$\forall x, y \in U,\ x \neq y: \quad \Pr_{h \in H}[h(x) = h(y)] \leq \frac{1}{m}$$

ΕΚΤΟΣ ΥΛΗΣ

Universal Hashing Example

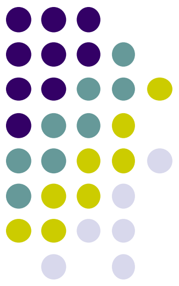


- Assume a, b are randomly chosen integers and $a \neq 0$
- Given a prime number p, with $p \geq m$
- Then, the following family of hash functions is universal:

$$h_{a,b}(x) = ((ax+b) \% p) \% m$$

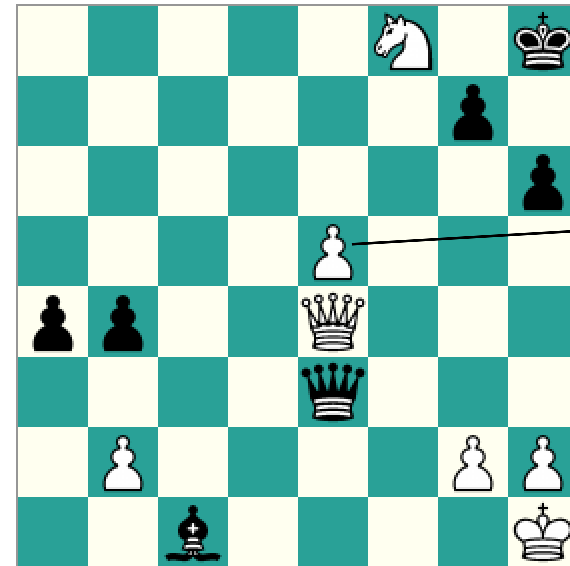
- Note: commonly used families of hash function use bit-arithmetic instead of modulo operations for efficiency

ΕΚΤΟΣ ΥΛΗΣ



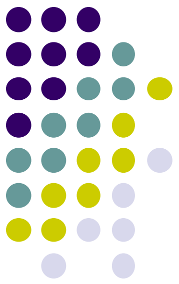
Hashing for indexing Chess Games

- Used in Chess engines
- Zorbist hashing:
  - Generate an array of 781 64 bit random numbers
  - One number for each piece at a position (2*6*64 total)
  - 13 additional numbers encoding side to move, castling rights, etc
  - A position is hashed to a bucket by XORing appropriate random numbers
    - Need 64bits to describe a board
    - Very small rate of collisions



$X = x_1$ XOR x_2 XOR … x_{14}

More on x XOR y



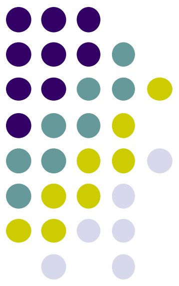
- Result is 1 if input bits differ, 0 otherwise
- 0101 XOR 0110 = 0011
- 0011 XOR 0110 = 0101

and

- 0011 XOR 0101 = 0110

ΕΚΤΟΣ ΥΛΗΣ

Hashing as an index



- Organize your data so as to quick locate records based on attribute's x value
  - Data may be stored in memory or on disk

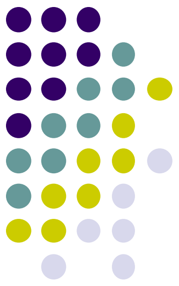
retrieve info on orderNo=00174 → ? → h(00174)=1 →

Data container (e.g. table or memory array)

Bucket	
0	
1	00174,John, Smith, 555-2047162, Smartphone, Delivered
2	82011, Nick, Taylor, 555-4014154, PC, Pending
3	
4	

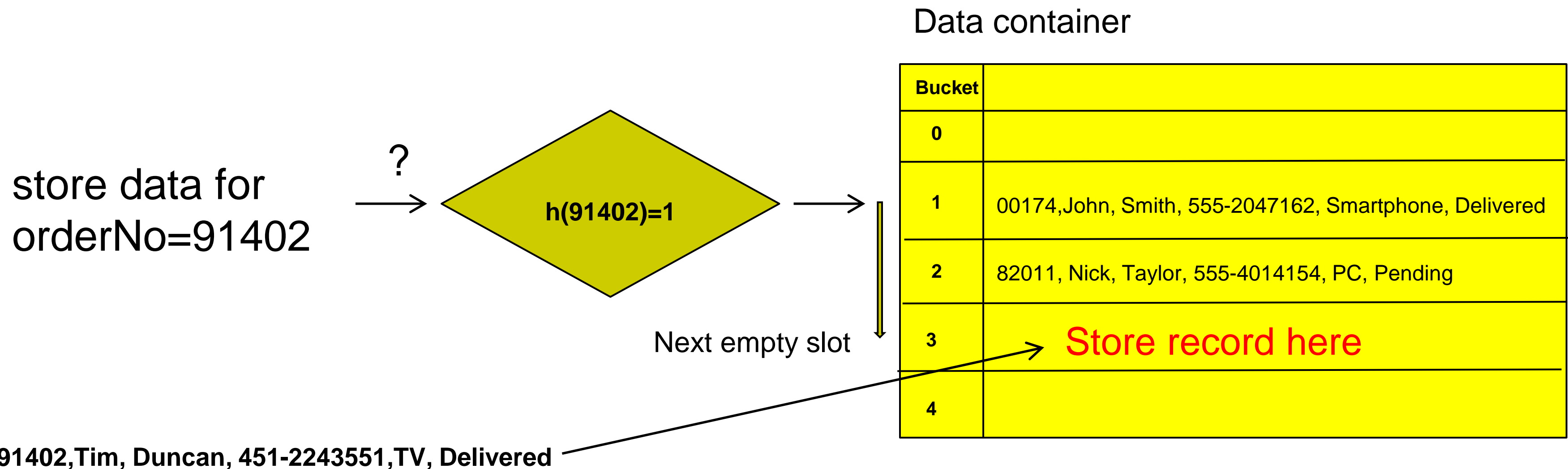
Index may store records, or refs (pointers) to these records

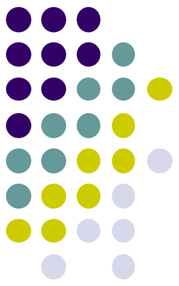
ΕΚΤΟΣ ΥΛΗΣ



Handling *collisions*

- Another key hashes to the same position
  - Idea 1: rehash
    - simplest implementation: linearly scan for next available slot





Chaining (Main Memory)

buckets

Pointer to record with key=32

key

g (key)

g(key) = key % 8

0
1
2
3
4
5
6
7

32 → 48 → (null)

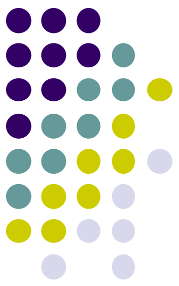
10 → (null)

27 → 75 → (null)

21 → (null)

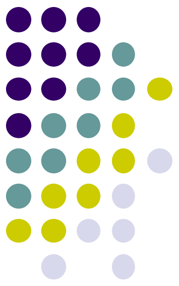
55 → (null)

Roadmap

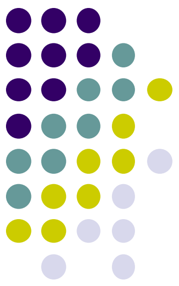


- Στο μάθημα θα μας απασχολήσουν τεχνικές κατακερματισμού για την οργάνωση εγγραφών στο δίσκο
- Στη συζήτηση αυτή ένα bucket θα αποθηκεύεται σε μία σελίδα στο δίσκο
  - Σε περίπτωση που αυτή η σελίδα γεμίσει, σε κάποιες από τις τεχνικές, ενδέχεται να δημιουργούνται σελίδες υπερχείλισης

Πως οργανώνουμε τις εγγραφές σε ένα bucket;



- Τις κρατάμε ταξινομημένες ως προς την τιμή του κλειδιού X;
  - Ναι, εφόσον θέλουμε να μειώσουμε το CPU κόστος της αναζήτησης.
    - κάνοντας δυαδική αναζήτηση μέσα στον κάδο όταν αυτός έρθει στη μνήμη.
    - εισαγωγές και διαγραφές γίνονται ποιο περίπλοκες.



Στατικός Κατακερματισμός

Έστω ότι κάθε bucket αρχικοποιείται ως μία σελίδα με χωρητικότητα 2 εγγραφές

INSERT:

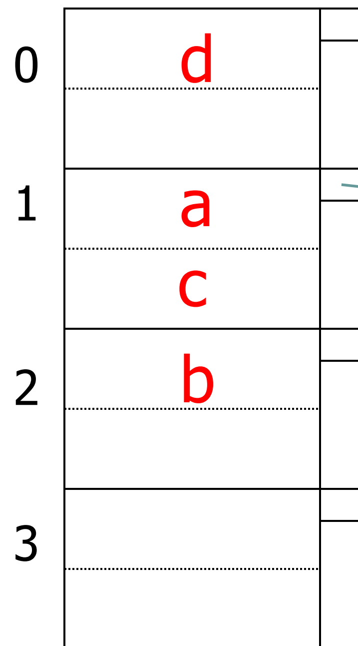
h(a) = 1

h(b) = 2

h(c) = 1

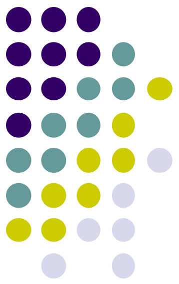
h(d) = 0

h(e) = 1



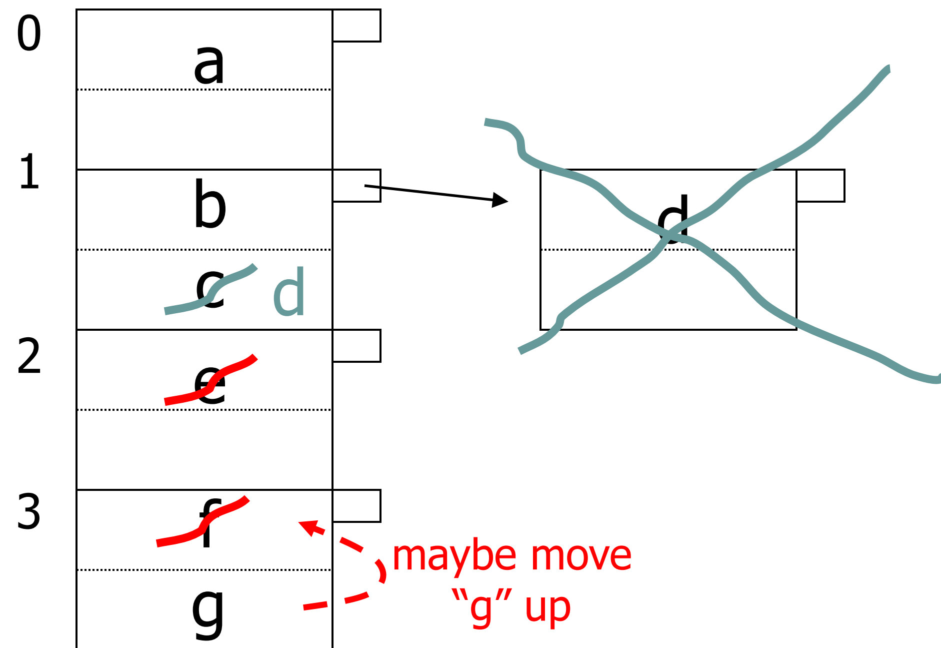
Υπερχείλιση (overflow)

Οι σελίδες του ίδιου bucket είναι συνδεδεμένες σε μία λίστα (pointer = αριθμός σελίδας)

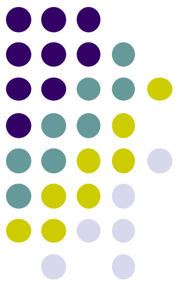


Διαγραφές

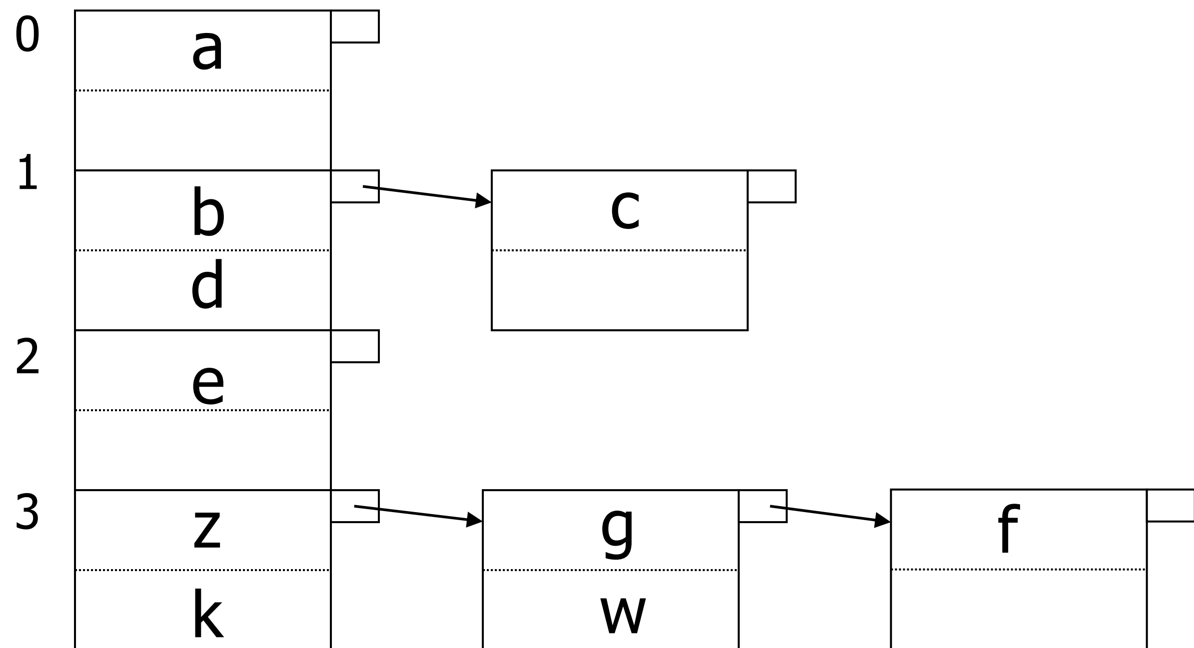
Delete:
- e
- f
- c

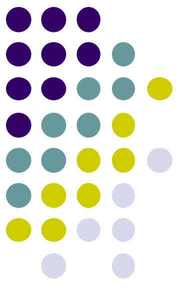


Κόστος αναζήτησης?



- Clustering index
  - έστω ότι οι εγγραφές τοποθετούνται χωρίς διάταξη μέσα σε κάθε bucket
- Ποιο είναι το μέσο κόστος αναζήτησης μιας τιμής x?
  - Αν το x δεν υπάρχει στο dataset?
  - Αν το x υπάρχει στο dataset?





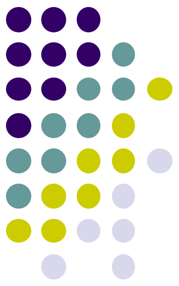
Rule of thumb:

- Θέλουμε τα buckets να είναι γεμάτα από 50% έως 80%

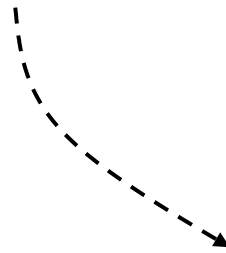
Space Utilization U= # keys used / total # keys that fit

- If U< 50%, σπατάλη χώρου
- If U> 80%, συχνές αναδιατάξεις λόγω υπερχείλισης (overflow)

Πως αντιμετωπίζουμε εισαγωγές/διαγραφές?

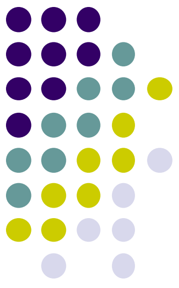


- Overflows and reorganizations
  - (βλ. προηγούμενα παραδείγματα)
- Δυναμικός Κατακερματισμός (Dynamic hashing)



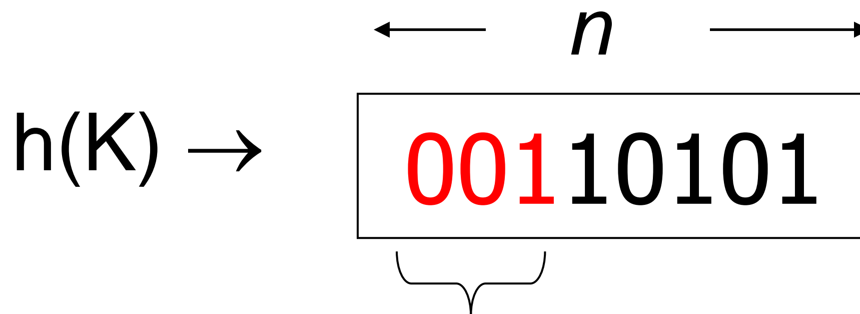
- Extendible (επεκτατός)
- Linear (γραμμικός)

Extendible hashing: Δύο κεντρικές ιδέες

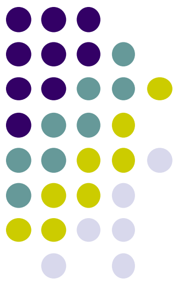


(a) Για τιμή κλειδιού K, βλέπω το αποτέλεσμα της αποτίμησης της συνάρτησης h(K) στη δυαδική του αναπαράσταση.

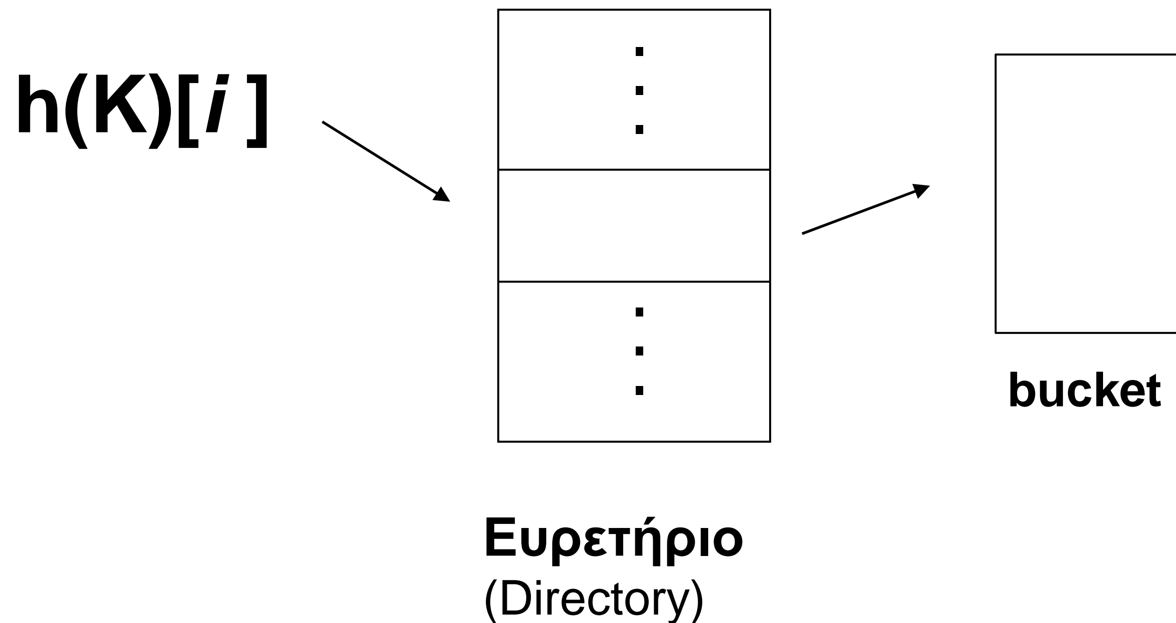
Χρησιμοποιώ τα πρώτα *i* από τα *n* bits



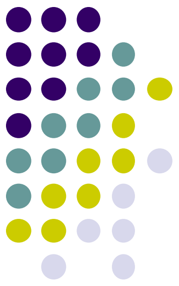
i → **(ολικό βάθος)** η τιμή του *i* αυξάνει ελεγχόμενα όσο αυξάνει ο αριθμός των εγγραφών



(b) Τα **i (=ολικό βάθος) πρώτα bits** της συνάρτησης με οδηγούν σε **ευρετήριο** μέσω του οποίου βρίσκω το κατάλληλο bucket.

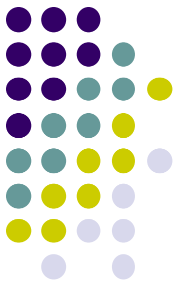


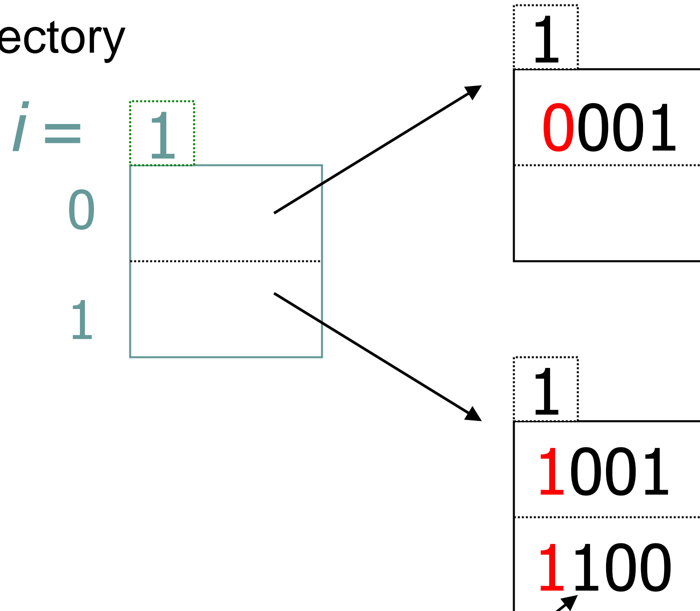
Επίσης



- Δεν υπάρχουν σελίδες υπερχείλισης

Παράδειγμα: h(k) με n=4 bits 2 εγγραφές/bucket



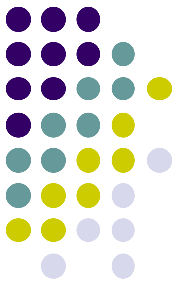


Δύο αρχικά buckets. Το πρώτο για hash codes τα οποία αρχίζουν από 0 και το δεύτερο για όσα αρχίζουν από 1

Σημ: στα παραδείγματα αναφέρεται η τιμή της h(k) αντί για το κλειδί k μέσα στα buckets

Παράδειγμα: h(k) με n=4 bits 2 εγγραφές/bucket

Insert 1010



Νέα κατάσταση

Ολικό βάθος= 2

Τοπικό βάθος κάδου

1
0001

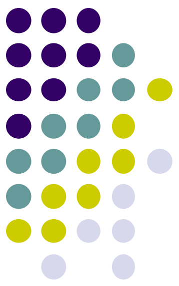
2
1001
1010

2
1100

00
01
10
11

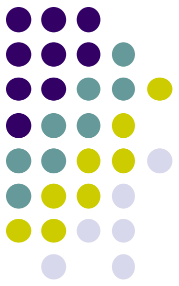
ευρετήριο

Νέα κατάσταση



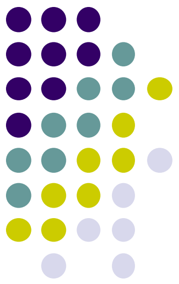
Η σελίδα αυτή χρησιμοποιείται για τα buckets 00 και 01

Σύνοψη



- Όταν χρειαστεί να κάνω εισαγωγή σε bucket που είναι γεμάτο εγγραφές
  - Αυξάνω το τοπικό βάθος του κάδου κατά 1
    - Το συγκεκριμένο bucket σπάει σε δύο
  - Το ευρετήριο διπλασιάζεται όταν το νέο τοπικό βάθος ξεπεράσει το τρέχον ολικό βάθος
    - Αυξάνει το ολικό βάθος κατά 1

Συνεχίζω



i = 2

00	→ 0000, 0001 (2)
01	→ 0001, 0111 (~~1~~ 2)
10	→ 1001, 1010 (2)
11	→ 1100 (2)

2: 0000, 0001

~~1~~ 2: ~~0001~~, ~~0111~~ 0111

2: 1001, 1010

2: 1100

Insert:

0111

0000

Ακόμη μία εισαγωγή

i = 2

00
01
10
11

0000
0001
2

0111
2

1001
1001
3

1010
~~1001~~
~~1010~~
2 3

1100
2

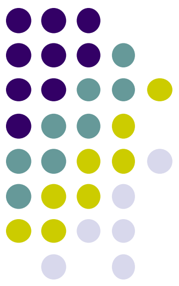
i = 3

000
001
010
011
100
101
110
111

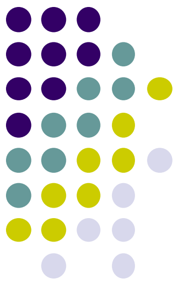
Insert:

1001

Extendible hashing: Διαγραφές

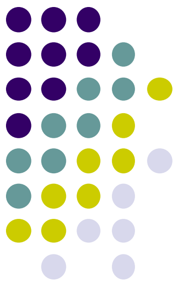


- Όχι αναδιάταξη: απλώς διαγράφω από τα buckets
  - Μπορεί να καταλήξω με buckets που έχουν πολύ λίγες εγγραφές (U<50%)
- Αναδιάταξη
  - Ένωσε buckets, συρρίκνωσε το ευρετήριο (αν γίνεται)



Παράδειγμα διαγραφής:

- Δείτε το παράδειγμα της εισαγωγής με την ανάποδη σειρά…



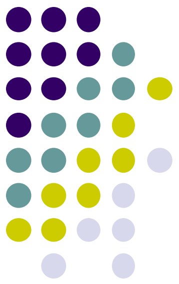
Σύνοψη

Extendible hashing

+ Διαχειρίζεται σχέσεις με αυξανόμενο/μεταβλητό αριθμό εγγραφών
  - Χωρίς μεγάλη σπατάλη χώρου
  - Χωρίς να απαιτείται κάθε φορά πλήρης αναδιοργάνωση

- Έμμεση πρόσβαση στις εγγραφές μέσω του ευρετηρίου
  - Δεν είναι πρόβλημα εφόσον το ευρετήριο χωράει στη μνήμη

- Με κάθε αύξηση του i το μέγεθος του ευρετηρίου **διπλασιάζεται**

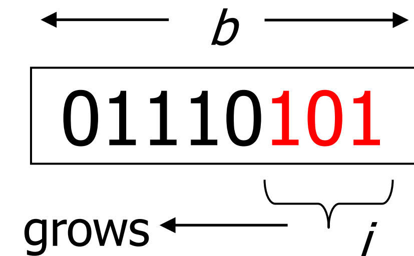


Linear hashing

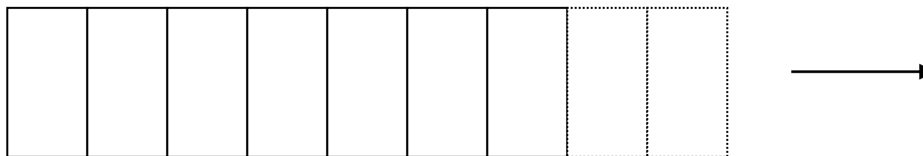
- Μία διαφορετική τεχνική δυναμικού κατακερματισμού

2 Ιδέες:

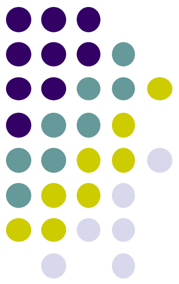
(a) Χρησιμοποιώ τα *i* λιγότερο σημαντικά bits



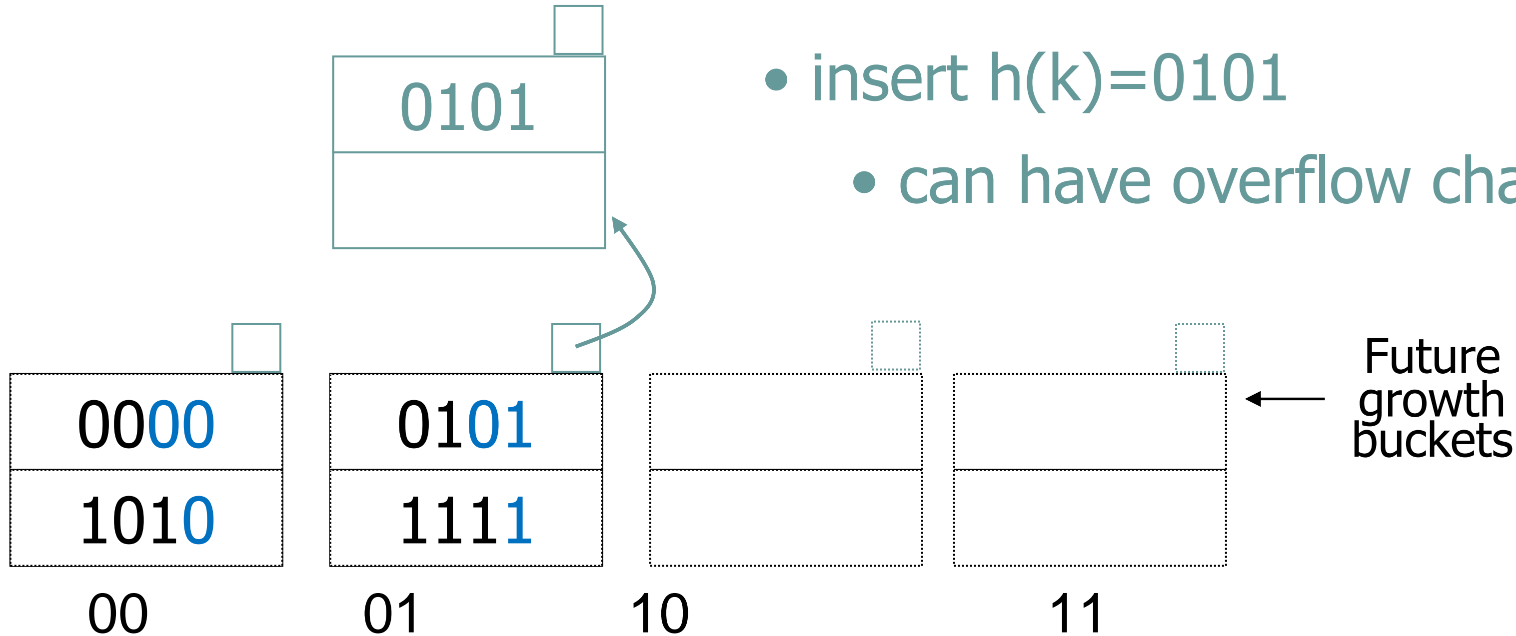
(b) Το αρχείο με τα buckets αυξάνει γραμμικά ανάλογα με τις ανάγκες



Example b=4 bits, i=2, 2 keys/bucket



- insert h(k)=0101
  - can have overflow chains!

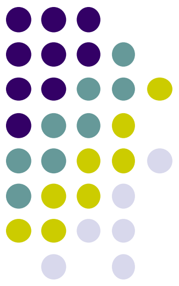


m = 01 (max used block)

Rule

If h(k)[i] ≤ m, then
look at bucket h(k)[i] → Κοίτα τα τελευταία i bits
else, look at bucket h(k)[i] - 2^{i-1} → Κοίτα τα τελευταία i-1 bits

<u>Example</u> b=4 bits, i =2, 2 keys/bucket



- insert 0101

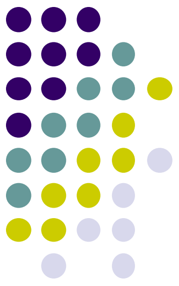
0000	0101	1010	1111
1010	0101 1111		
00	01	10	11

0101

Future growth buckets

m = 01 (max used block)
10
11

Example Continued: How to grow beyond this?



i = ~~2~~ 3

0000	~~0101~~	1010	1111		0101
	~~0101~~				0101
0 00	0 01	0 10	0 11	100	101
~~100~~	~~101~~	110	111		. . .

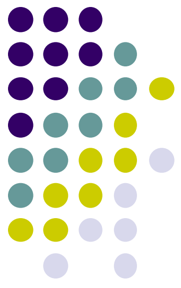
m = ~~11~~ (max used block)

~~100~~

101

⌧ Πώς μεγαλώνει το αρχείο?

Απάντηση: Αυξάνοντας το m όταν το space utilization U ξεπεράσει ένα όριο

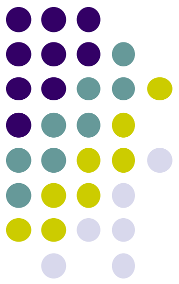


- Θυμήσου $U = \frac{\text{\# used slots}}{\text{total \# of slots}}$

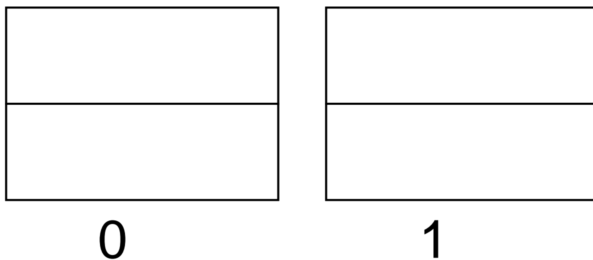
- If U > threshold (πχ 80%) αυξάνω το *m*

(και πιθανών το *i αν το m ξεπεράσει το όριο* 2^i-1)

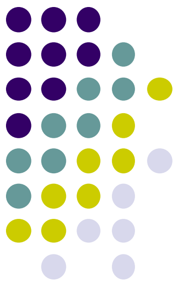
Παράδειγμα 2

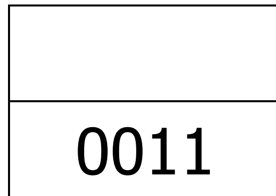


- Έστω h(x) = x
- Αρχική κατάσταση: m=1, i=1
- Split εφόσον U>= 80%



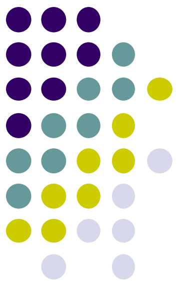
Εισαγωγή 3





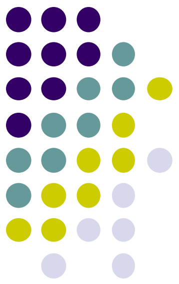
0 1

Εισαγωγή 2, 0



0000	
0010	0011
0	1

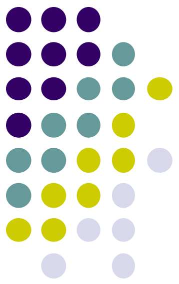
Εισαγωγή 5



- U = 100%

0000	0101
0010	0011
0	1

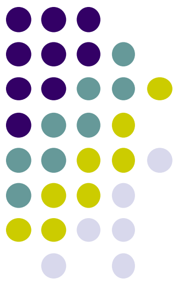
Εισαγωγή 5

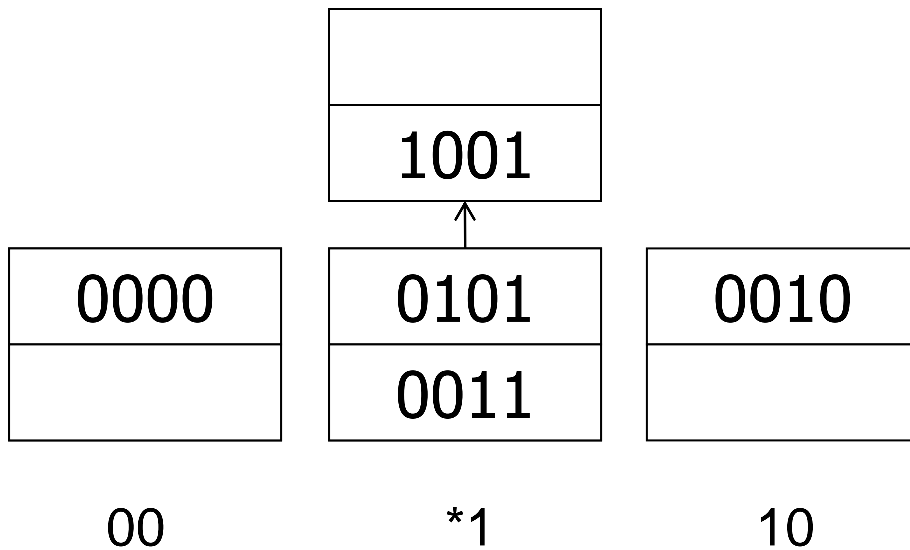


- Increase m→10, i→2. U=4/6<80%

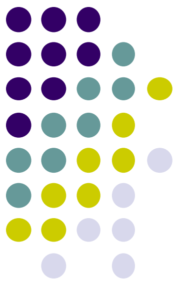
0000	0101	0010
~~0010~~	0011	
00	*1	10

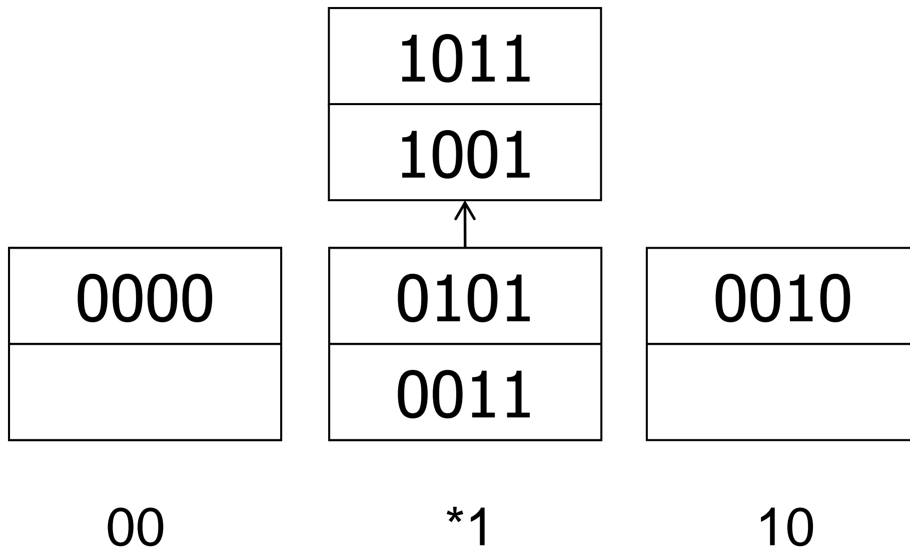
Εισαγωγή 9



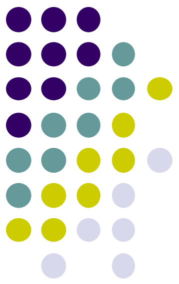


Εισαγωγή 11

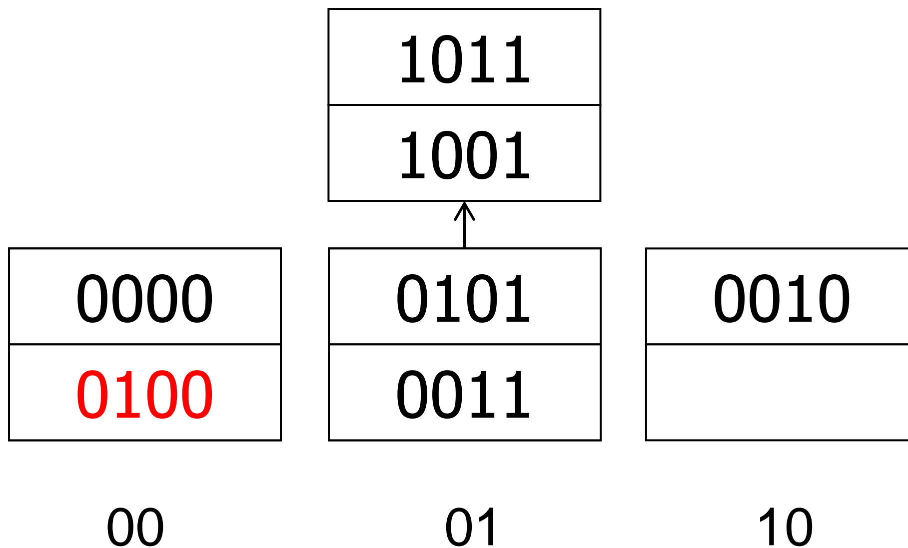




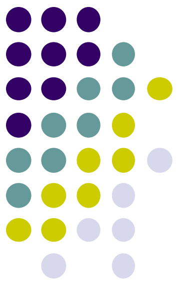
Εισαγωγή 4



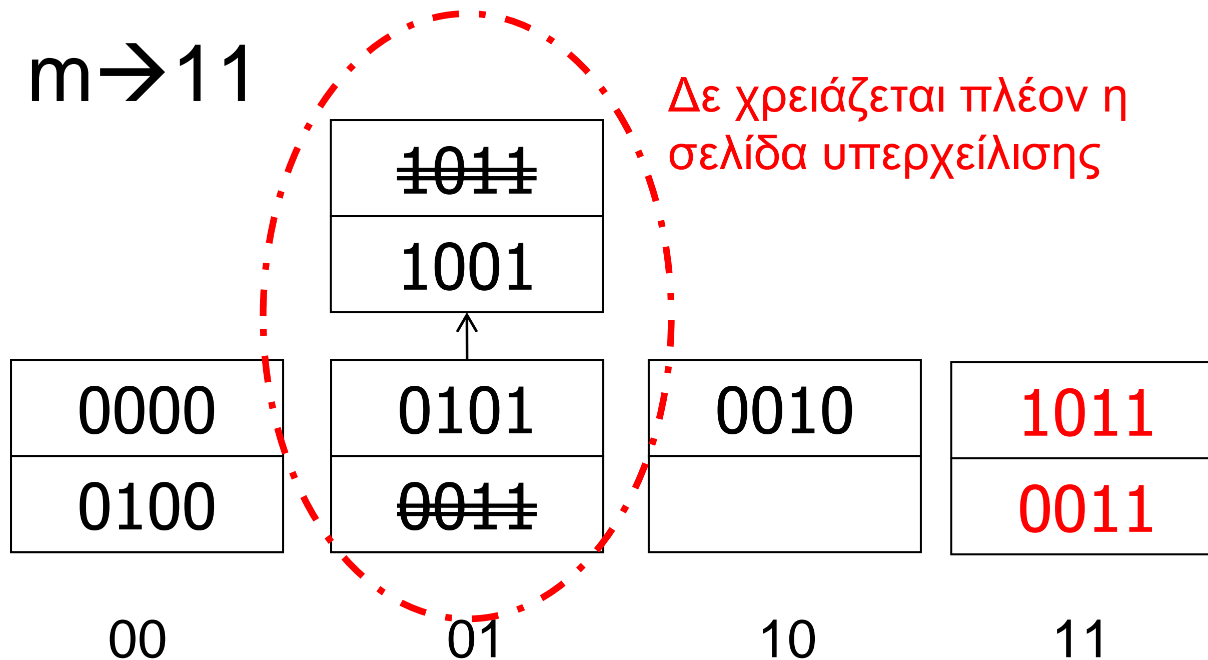
- U=7/8>80%



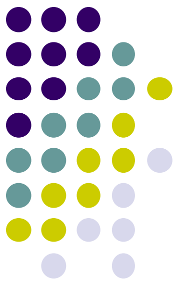
Εισαγωγή 4



- m→11



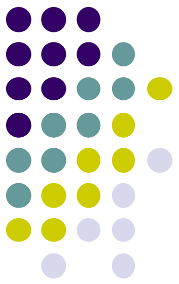
Still U>80%



00	01	10	11
0000	0101	0010	1011
0100	1001		0011

- m→100 , i→3, U=70%

000	*01	*10	*11	100
0000	0101	0010	1011	
~~0100~~	1001		0011	0100



Summary

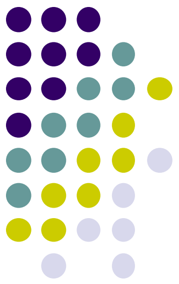
Linear Hashing

(+) Can handle growing files
- with less wasted space
- with no full reorganizations

(+) No indirection like extendible hashing

(-) Can still have overflow chains

Example: BAD CASE



Very full

Very empty

Need to move

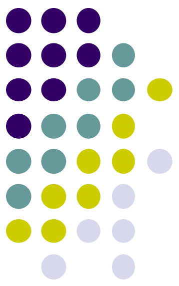
m here (111)

Would waste

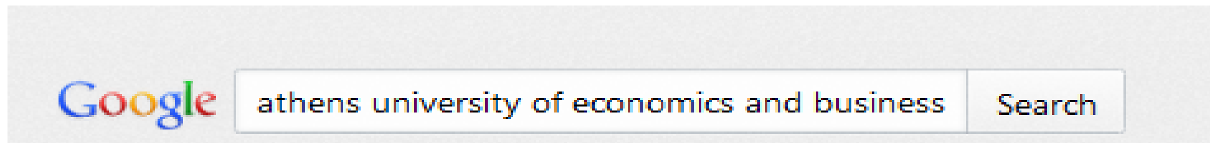
space...

*11

Τι άλλο μπορώ να κάνω με τη χρήση κατακερματισμού?

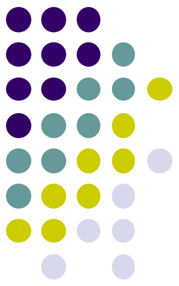


- ΣΔΒΔ: Compute Joins/Group bys/Duplicate Elimination
- Big Data Management and Analysis
  - Membership test & Distributed Joins (Bloom Filters)
  - (Distinct) Counting/Heavy Hitters
    - E.g. Google wants to know the most frequent search terms



  - Compute Norms/Dot Products/Cosine Similarity, Document/Set Similarity, Nearest Neighbor Search…

Ευρετήρια (πχ B-tree) ή κατακερματισμός;



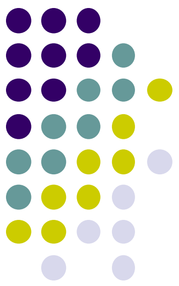
- Κατακερματισμός χρήσιμος σε “point queries» (αναζήτηση μίας συγκεκριμένης τιμής)

πχ

```
SELECT …
FROM R
WHERE R.A = 5
```

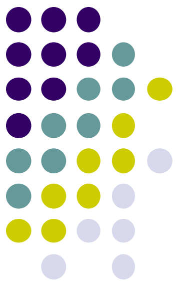
Σημ.: Η ερώτηση είναι καταχρηστική.
Ο κατακερματισμός είναι είδος ευρετηρίου

Παρατήρηση



- Κόστος αναζήτησης με κατακερματισμό
  - 1 I/O (εκτός και αν έχω overflow pages)
- Κόστος αναζήτησης με B^+-tree
  - # I/Os = ύψος δέντρου
- (+1 στα παραπάνω σε secondary index)

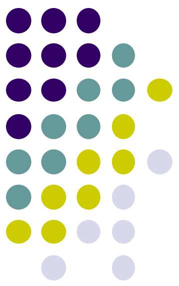
Ευρετήρια (πχ B-tree) ή κατακερματισμός;



- Ευρετήρια (πχ B-trees) καλά και για “Range Queries”

πχ

```
SELECT
FROM R
WHERE R.A > 5
```



Index definition in SQL

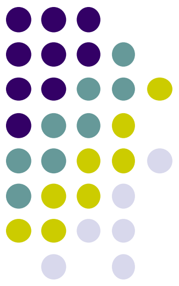
- Create index name on rel (attr)
- Create unique index name on rel (attr)

Δεν επιτρέπονται εγγραφές με την ίδια τιμή στο γνώρισμα attr. Τιμές NULL επιτρέπονται

- Drop index name

- Σε μερικά ΣΔΒΔ μπορείς να επιλέξεις και τον τύπο του ευρετηρίου (B-tree,HASH)

Τα σύγχρονα ΣΔΒΔ υποστηρίζουν και άλλα είδη ευρετηρίων



- Ευρετήριο Σύζευξης: Join index

  CREATE BITMAP INDEX loc_sales_bit

  ON sales(location.state)

  FROM sales, location

  WHERE sales.loc_location_key = location.location_key;

- Η παρακάτω επερώρηση χρησιμοποιεί το join index

  SELECT SUM(sales.amount)

  FROM sales,location

  WHERE sales.location_key=location.location_key

  AND location.state="Αττική"