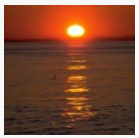
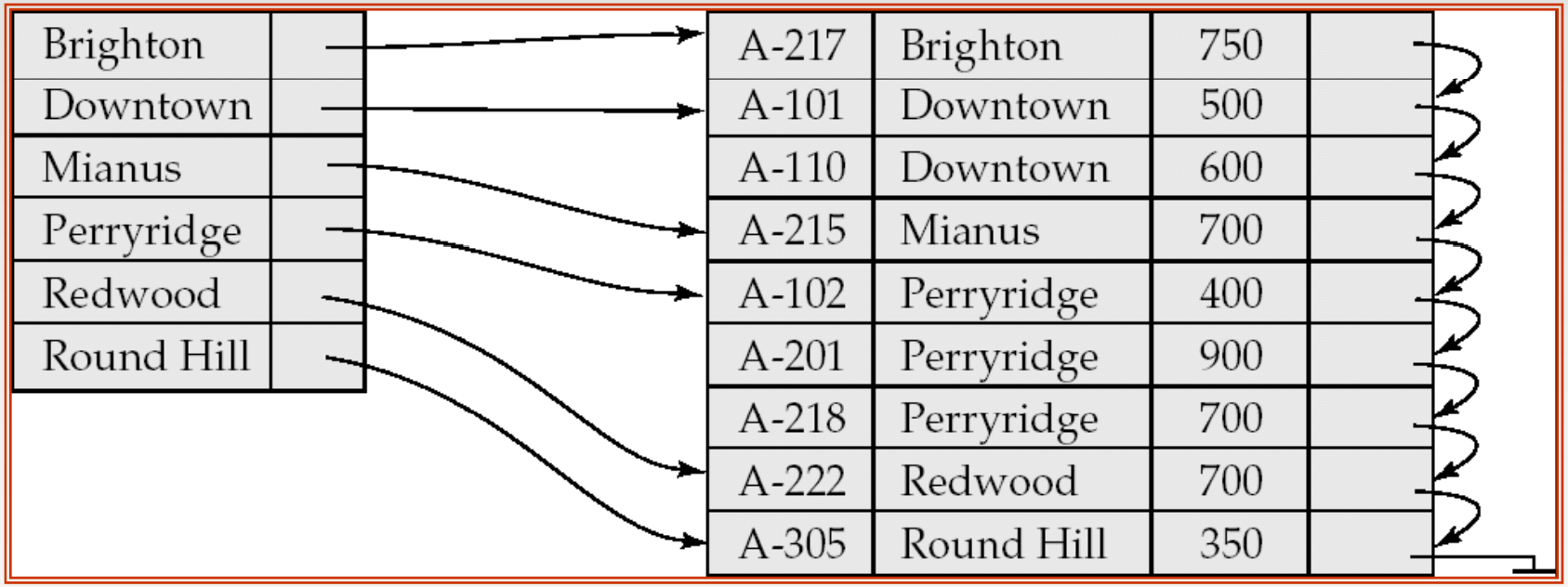
Πολυδιάστατα Ευρετήρια και Συναρτήσεις Κατακερματισμού

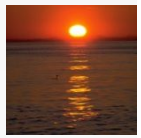
Database System Concepts, 5th Ed.



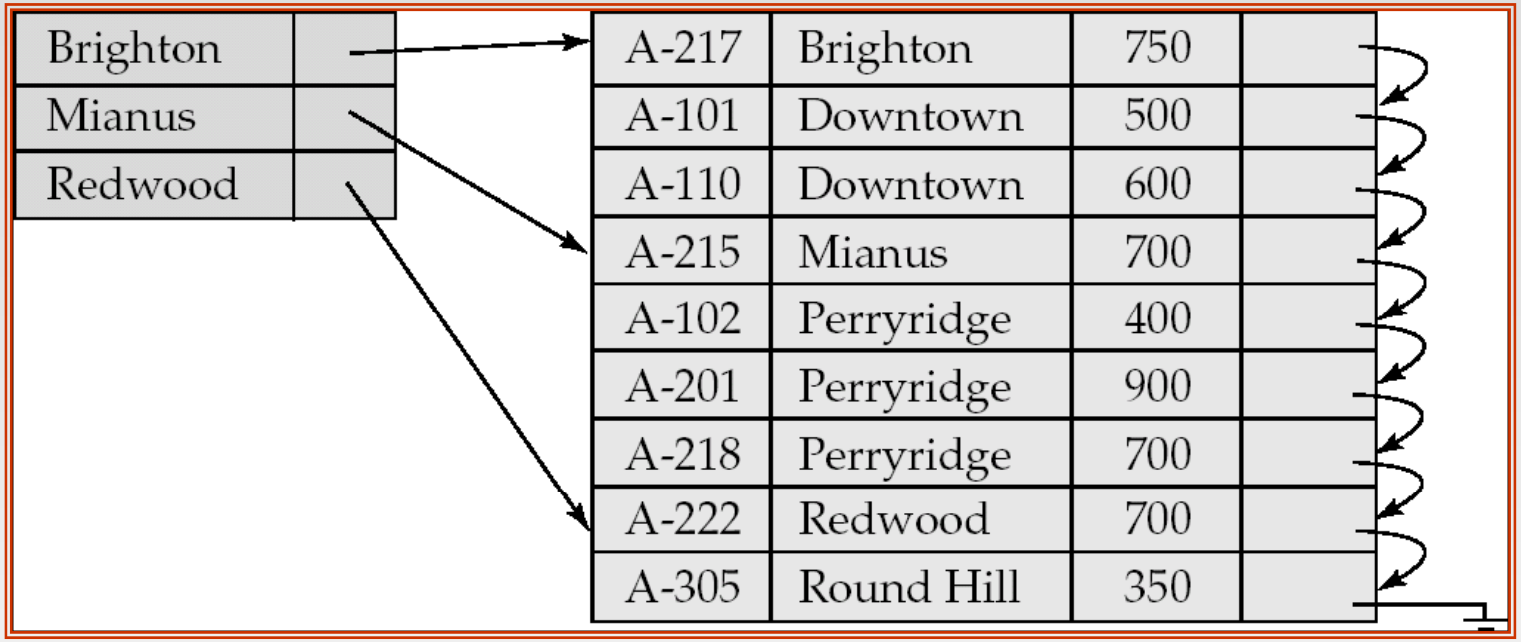
Πυκνά Ευρετήρια (Dense Index Files)

- Dense index — Index record εμφανίζεται για κάθε search-key value στο αρχείο (file).

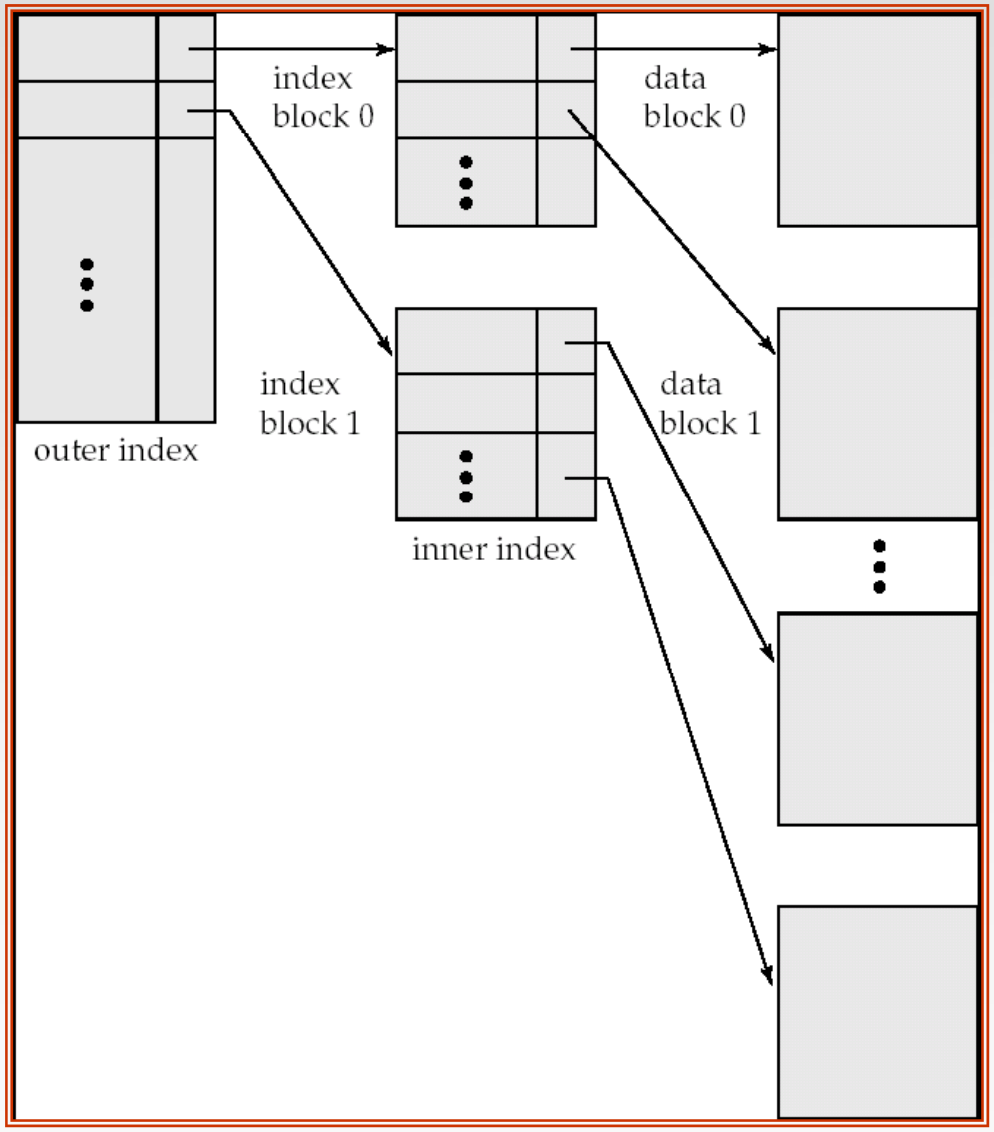




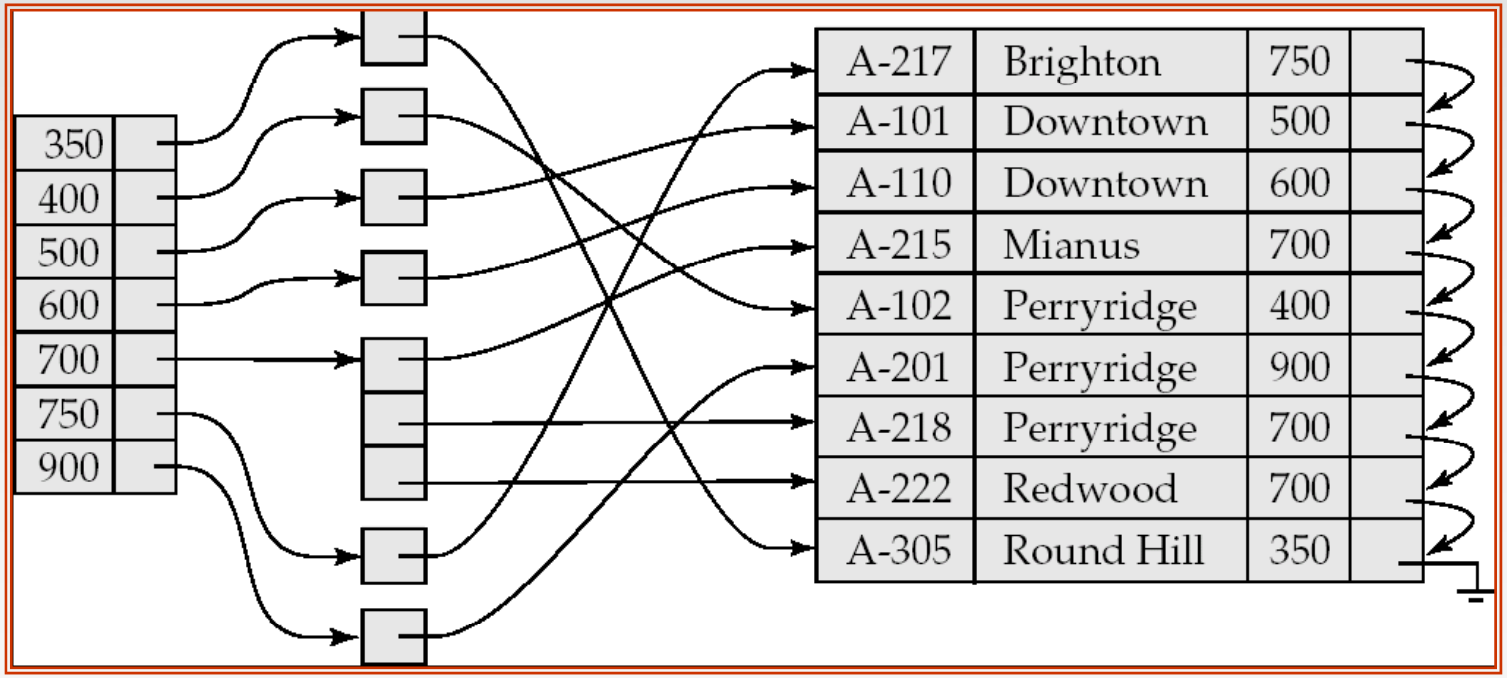
Αραιά Ευρετήρια (Sparse Index Files)



Πολυεπιπεδα Ευρετήρια (Multilevel Index)



Δευτερεύον Ευρετήριο (secondary index) στο πεδίο *balance* του πίνακα *account*



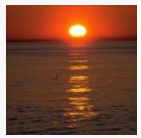
Η δομή ενός κόμβου στο B+-Tree

- Τυπικός κόμβος

P_1	K_1	P_2	. . .	P_{n-1}	K_{n-1}	P_n

  - K_i είναι τα search-key values
  - P_i είναι δείκτες στα παιδιά (για εσωτερικούς κόμβους) ή δείκτες στα records ή στα buckets των records (για κόμβους φύλλα).
- Τα search-keys σε κάθε κόμβο είναι διατεταγμένα

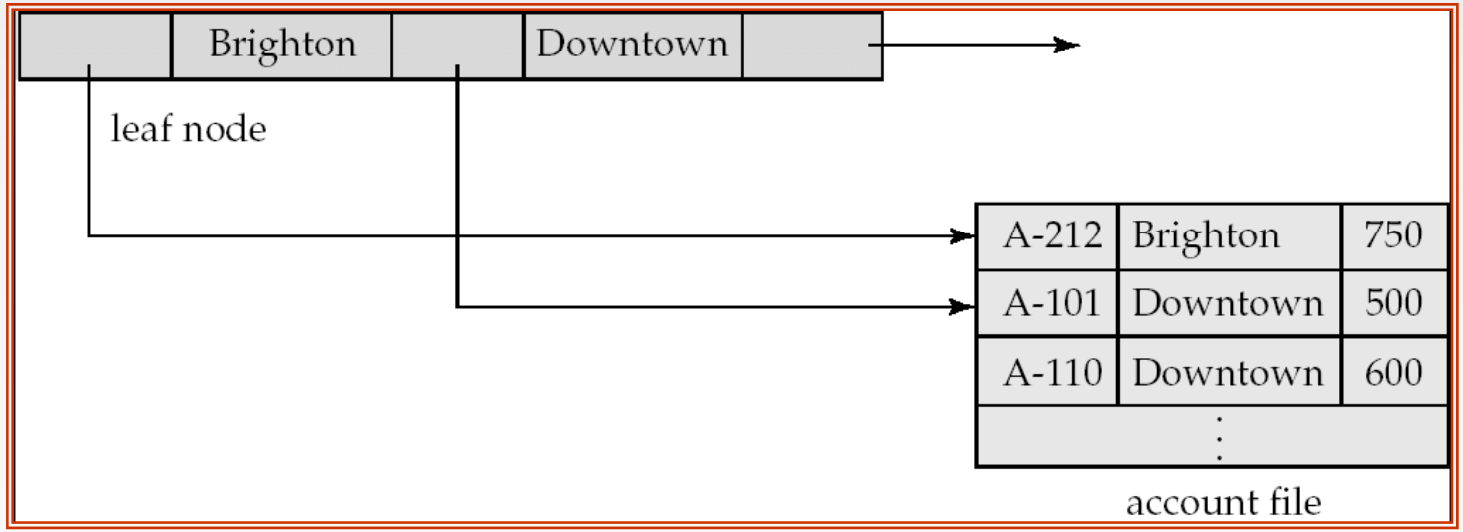
$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$



Κόμβοι – Φύλλα των B+-Trees

Ιδιότητες των κόμβων - φύλλων:

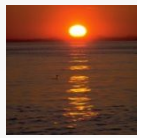
- Για i = 1, 2, . . ., n–1, ο δείκτης P_i δείχνει είτε στο file record με search-key την τιμή K_i, ή σε bucket με δείκτες στα file records, κάθε ένα από τα οποία έχει search-key την τιμή K_i. Η δομή των buckets χρειάζεται όταν το search-key δεν αποτελεί από μόνο του *πρωτεύον κλειδί*.
- Αν L_i, L_j είναι φύλλα και $i < j$, *τα* search-key values του L_i είναι μικρότερα από αυτά του L_j
- *Ο δείκτης* P_n δείχνει στο επόμενο φύλλο με βάση τη διάταξη των search-keys



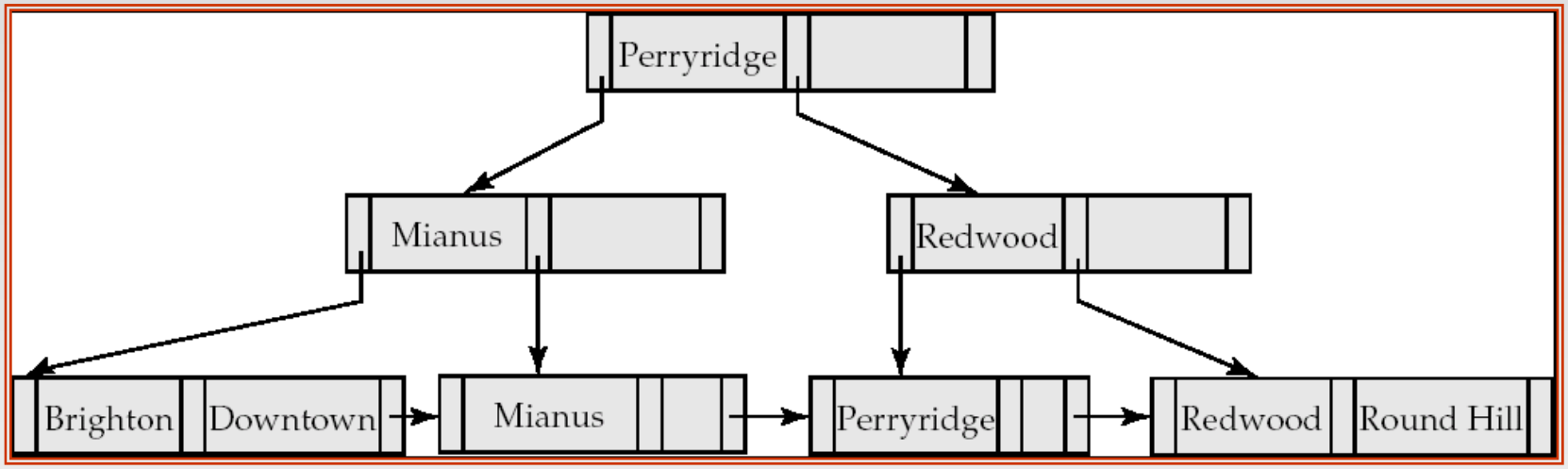
Εσωτερικοί Κόμβοι των B+-Trees

- Οι εσωτερικοί κόμβοι δημιουργούν ένα multi-level sparse index πάνω στους κόμβους - φύλλα. Για κάθε εσωτερικό κόμβο με *m* pointers:
  - Όλα τα search-keys του υποδέντρου (subtree) στο οποίο δείχνει ο δείκτης P_1 είναι μικρότερα από την τιμή K_1
  - Για $2 \leq i \leq n - 1$, όλα τα search-keys του υποδέντρου στο οποίο δείχνει ο P_i έχουν τιμές μεγαλύτερες ή ίσες από K_{i-1} και μικρότερα από K_i

P_1	K_1	P_2	. . .	P_{n-1}	K_{n-1}	P_n

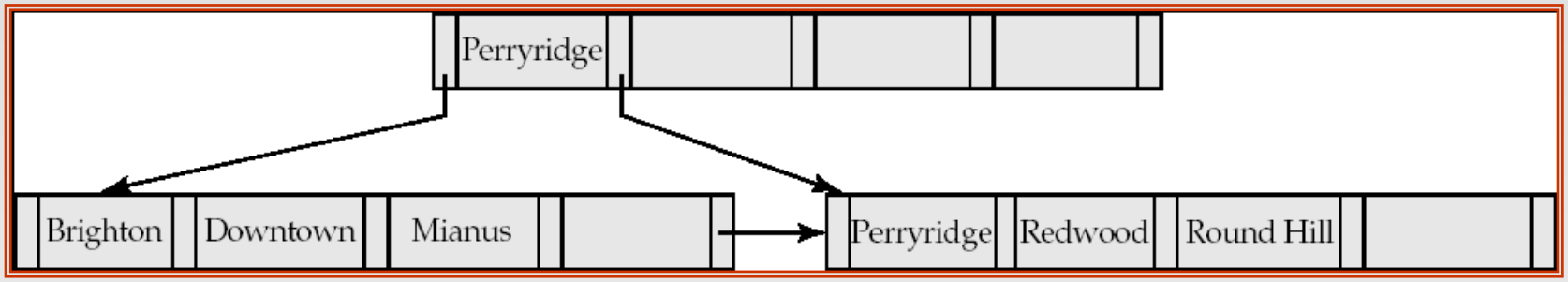


Example of a B+-tree



B+-tree για το *account* file (*n* = 3)

Example of B+-tree

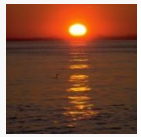


B+-tree για το *account* file (n = 5)

- Τα φύλλα έχουν από 2 έως 4 τιμές ($\lceil (n–1)/2 \rceil$ και n –1, όπου n = 5).
- Οι εσωτερικοί κόμβοι εκτός της ρίζας πρέπει να έχουν μεταξύ 3 και 5 παιδιών ($\lceil (n/2 \rceil$ και n για n =5).
- Η ρίζα πρέπει να έχει τουλάχιστον 2 παιδιά.

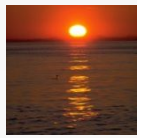
Ερωτήματα πάνω στα B^+-Trees

- Βρες όλα τα records με search-key την τιμή *k.*
  1. Ξεκινάμε από τη ρίζα
     1. Εξετάζουμε τη ρίζα για το μικρότερο search-key > *k.*
     2. Αν υπάρχει τέτοια τιμή, έστω K_i, ακολούθησε τον δείκτη P_i.
  2. Αν ο κόμβος στον οποίο πάμε ακολουθώντας τους προηγούμενους δείκτες δεν είναι κόμβος - φύλλο, επανέλαβε στον κόμβο αυτόν το βήμα 1.
  3. Διαφορετικά έχουμε μετακινηθεί σε κόμβο – φύλλο οπότε:
     1. Αν για κάποιο *i*, το κλειδί K_i = k τότε ακολούθησε τον δείκτη P_i ο οποίος μας πάει στο επιθυμητό record ή bucket.
     2. Διαφορετικά το record με το κλειδί k που ψάχνουμε δεν υπάρχει.



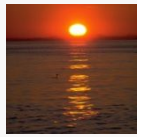
Ερωτήματα πάνω στα B+-Trees(Συν.)

- Για την επεξεργασία – απάντηση του ερωτήματος, ένα μονοπάτι από τη ρίζα στο κατάλληλο φύλλο διαπερνάται.
- Αν υπάρχουν *K* search-key τιμές στο αρχείο, το μονοπάτι δεν είναι μεγαλύτερο από $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- Ένας κόμβος έχει το ίδιο μέγεθος με ένα disk block, τυπικά 4 kilobytes, και το *n* κυμαίνεται περίπου στο 100 (40 bytes ανά index entry).
- Με 1 εκατομμύριο search key τιμές και *n* = 100, το πολύ $log_{50}(1{,}000{,}000) = 4$ κόμβοι προσπελαύνονται.
- Αντιθέτως αν χρησιμοποιούσαμε ένα ισοζυγισμένο δυαδικό δέντρο για να αποθηκεύσουμε 1 εκατομμύριο search key τιμές — θα έπρεπε να προσπελάσουμε περίπου 20 κόμβους
  - Η παραπάνω διαφορά είναι πολύ σημαντική, αφού κάθε προσπέλαση κόμβου απαιτεί ένα disk I/O, το οποίο με τη σειρά του απαιτεί χρόνο γύρω στα 20 milliseconds!



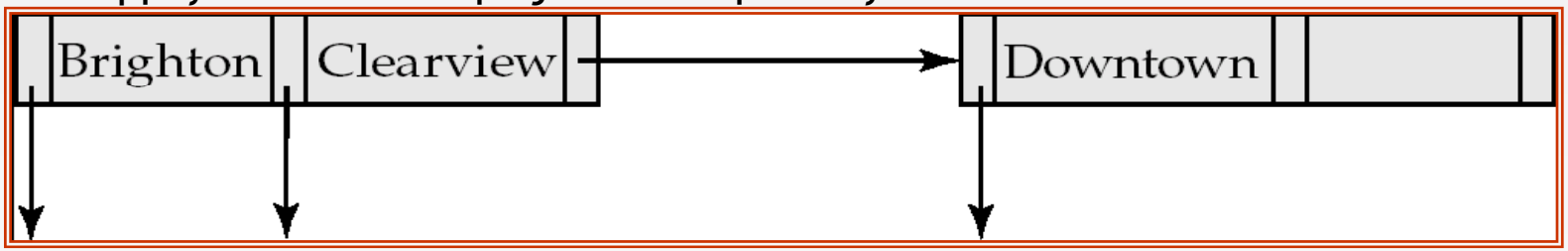
Ενημερώσεις σε B+-Trees: Ένθεση

- Βρες το φύλλο στο οποίο η τιμή του search-key θα έπρεπε να εμφανίζεται
- Αν η τιμή του search-key βρίσκεται ήδη στο φύλλο, τότε ένα record προστίθεται στο αρχείο.
- Αν η τιμή του search-key δεν βρίσκεται στο φύλλο, τότε πρόσθεσε το record στο κυρίως αρχείο. Τότε:
  - Αν υπάρχει χώρος στο φύλλο, τότε ένθεσε εκεί το ζεύγος (key-value, pointer).
  - Διαφορετικά, διέσπασε (split) τον κόμβο - φύλο (μαζί με τη νέα (key-value, pointer) καταχώρηση) ως εξής:



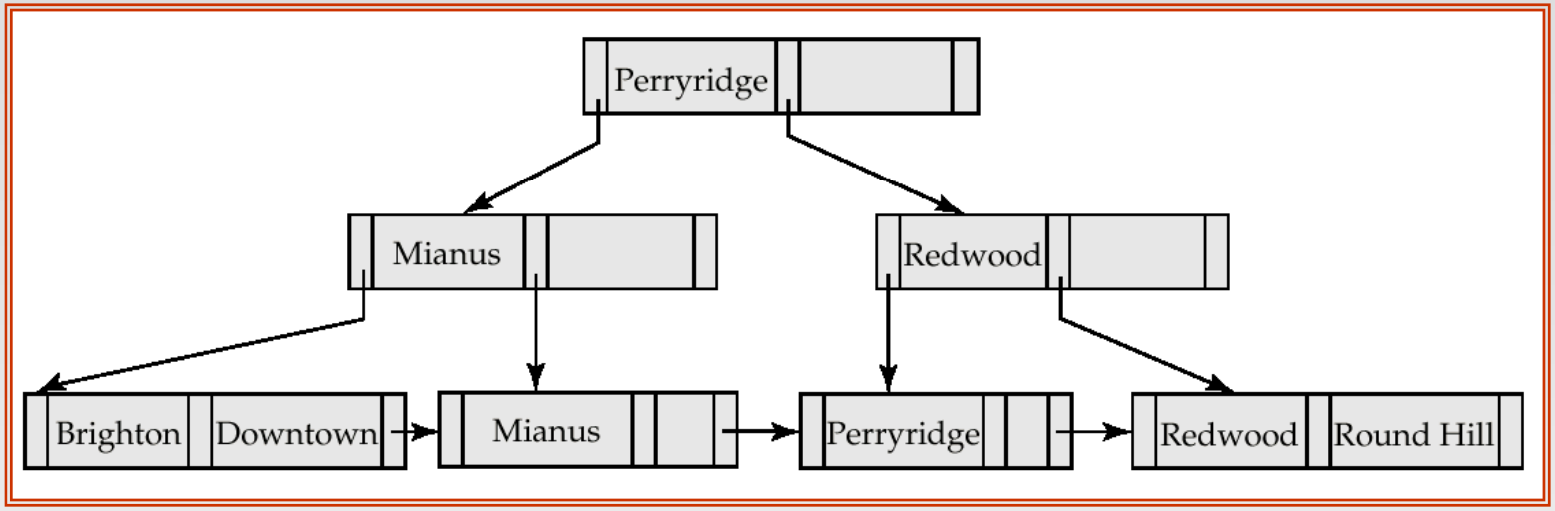
Ενημερώσεις σε B+-Trees: Ένθεση(Συν.)

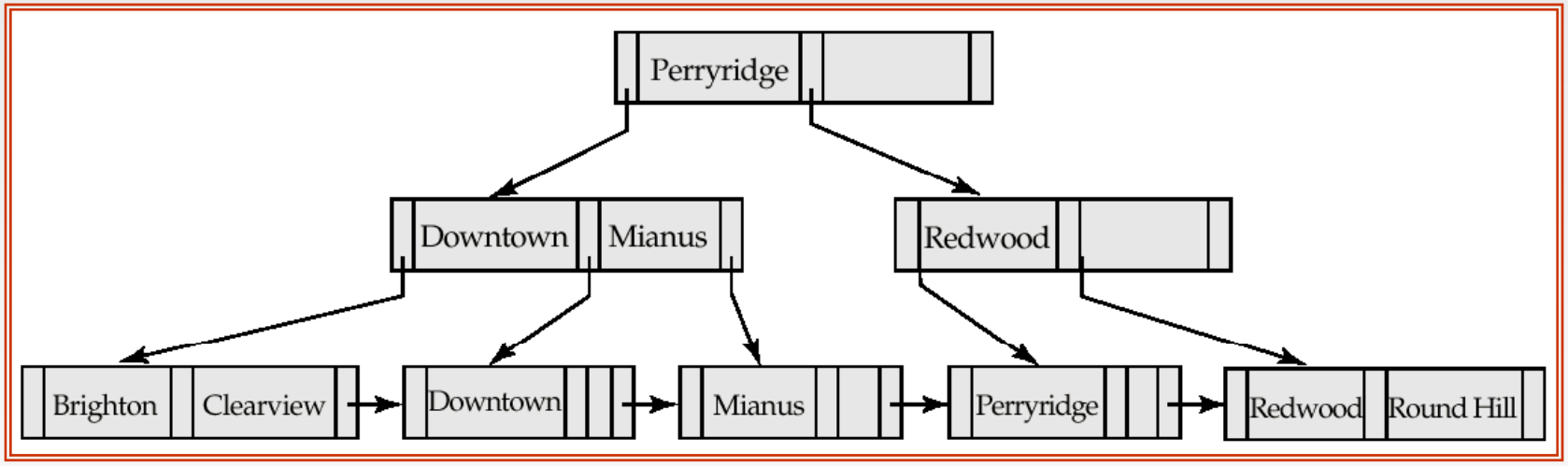
- Διάσπαση κόμβου:
  - Πάρε τα *n* (search-key value, pointer) ζεύγη (συμπεριλαμβανομένου και του νέου ζεύγους που εντέθηκε) σε διάταξη. Τοποθέτησε τα πρώτα $\lceil n/2 \rceil$ στον αυθεντικό κόμβο, και τα υπόλοιπα στον νέο κόμβο.
  - Έστω *p o νέος κόμβος,* και έστω *k* η ελάχιστη τιμή κλειδιού στον *p. Κάνε* Insert (*k,p*) στον πατέρα του κόμβου που διασπάστηκε. Αν ο πατέρας είναι πλήρης (full) , διέσπασέ τον και διέδωσε τη διάσπαση προς τα πάνω.
- Η διάσπαση των κόμβων διαδίδεται προς τα πάνω μέχρι ένας μη-πλήρης (not full) κόμβος βρεθεί. Στη χειρότερη περίπτωση διασπάται και η ρίζα οπότε το ύψος του δέντρο αυξάνεται κατά 1.



Το αποτέλεσμα της διάσπασης του κόμβου που περιέχει το Brighton και Downtown όταν ενθέσαμε το Clearview

Ενημερώσεις σε B+-Trees: Ένθεση(Συν.)

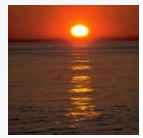




B+-Tree πριν και μετά την ένθεση του “Clearview”

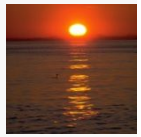
Ενημερώσεις σε B$^+$-Trees: Διαγραφή

- Βρες το record που πρέπει να διαγραφεί, και διέγραψέ το από το κυρίως αρχείο.
- Διέγραψε το ζεύγος (search-key value, pointer) από το φύλλο.
- Αν μετά τη διαγραφή ο κόμβος έχει ελάχιστα κλειδιά και αυτά μαζί με τα κλειδιά ενός αδελφού – κόμβου (sibling) χωράνε σε έναν απλό κόμβο, τότε:
  - Ένθεσε όλες τις παραπάνω search-key τιμές σε έναν απλό κόμβο (αυτόν από αριστερά), και διέγραψε τον άλλο κόμβο.
  - Διέγραψε το ζεύγος (K_{i-1}, P_i), όπου P_i ο δείκτης προς τον σβησμένο πλέον κόμβο, από τον πατέρα κόμβο, αναδρομικά χρησιμοποιώντας την παραπάνω διαδικασία.

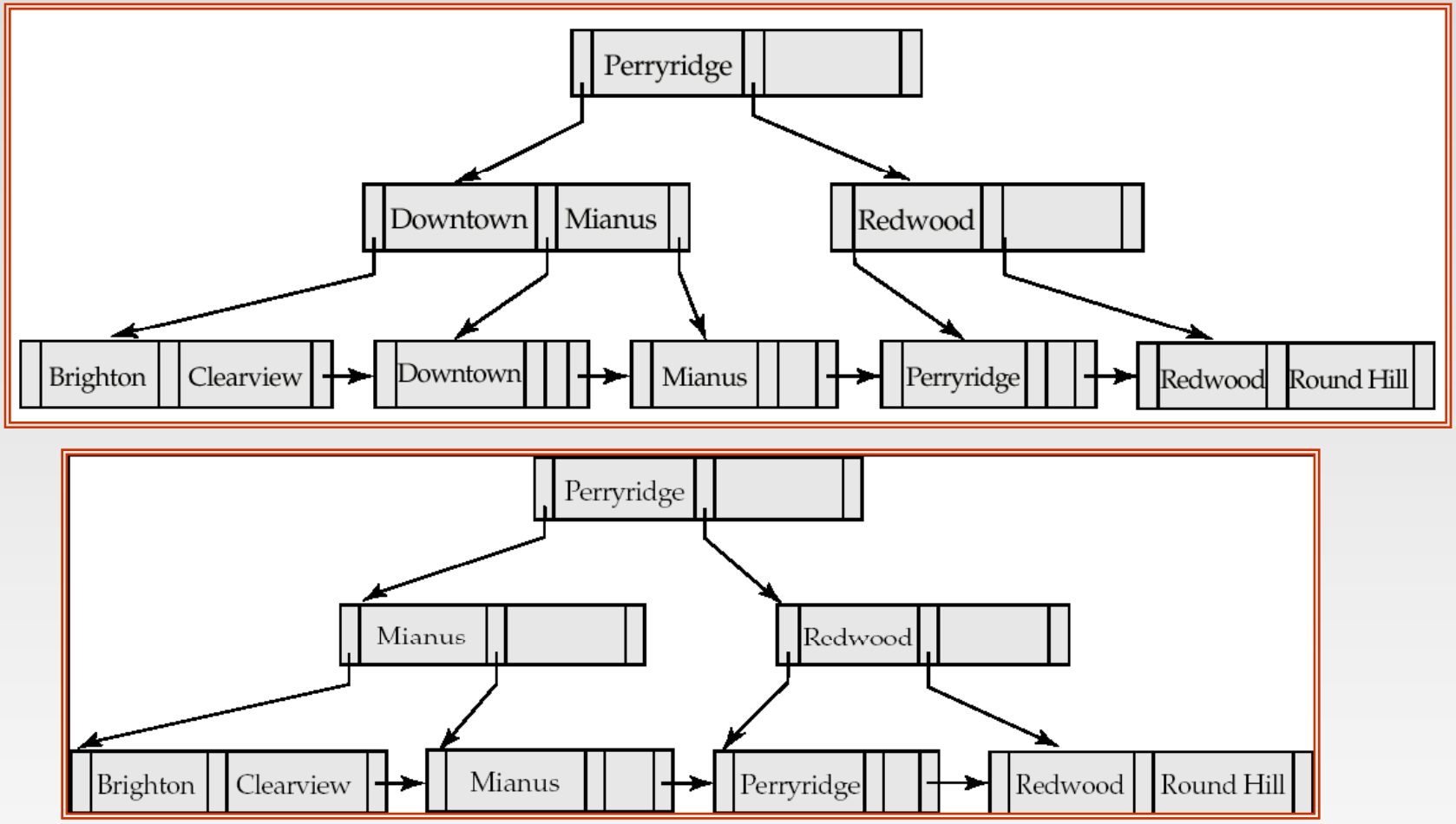


Ενημερώσεις σε B$^+$-Trees: Διαγραφή

- Διαφορετικά, αν μετά τη διαγραφή ο κόμβος έχει λίγα κλειδιά και αυτά μαζί με τα κλειδιά ενός αδελφού – κόμβου (sibling) δεν χωράνε σε έναν απλό κόμβο, τότε
  - Κάνε ανακατανομή των δεικτών μεταξύ του κόμβου και του αδελφού του έτσι ώστε να έχουν και οι δύο τον ελάχιστο αριθμό κλειδιών που απαιτείται.
  - Ενημέρωσε την τιμή του search-key στον πατέρα κόμβο.
- Οι διαγραφές των κόμβων διαδίδονται προς τα πάνω έως ότου βρεθεί κόμβος με $\lceil n/2 \rceil$ ή περισσότερους δείκτες. Αν μετά τη διαγραφή η ρίζα έχει μόνο έναν δείκτη ή αλλιώς μόνο ένα παιδί τότε η ρίζα διαγράφεται και το μοναδικό της παιδί γίνεται η νέα ρίζα του δέντρου.



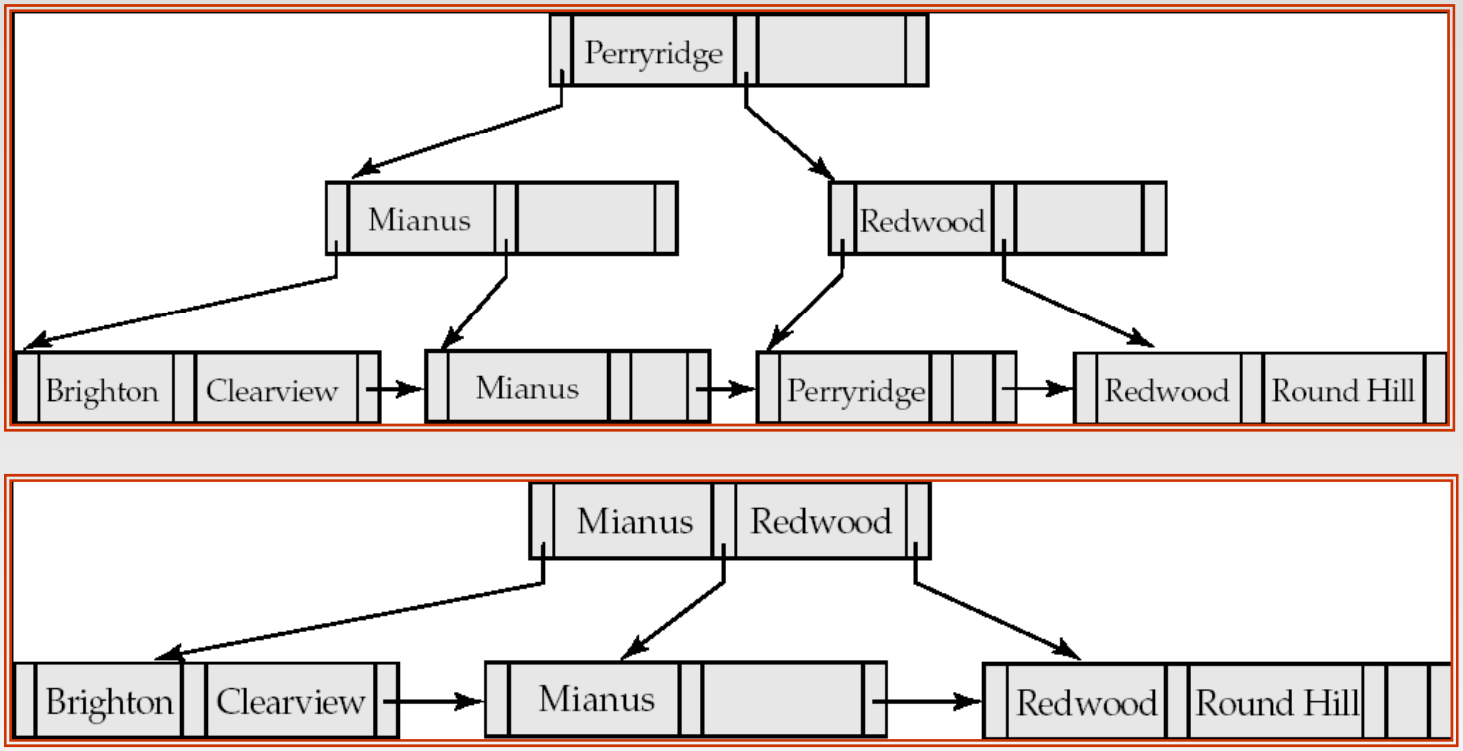
Ενημερώσεις σε B+-Trees: Διαγραφή



Πριν και μετά τη διαγραφή του "Downtown"

- Η διαγραφή του φύλλου που περιέχει το "Downtown" δεν έχει ως αποτέλεσμα ο κόμβος – πατέρας του να αποκτήσει υπερβολικά ελάχιστους δείκτες. Έτσι οι διαγραφές δεν διαδίδονται προς τα πάνω.

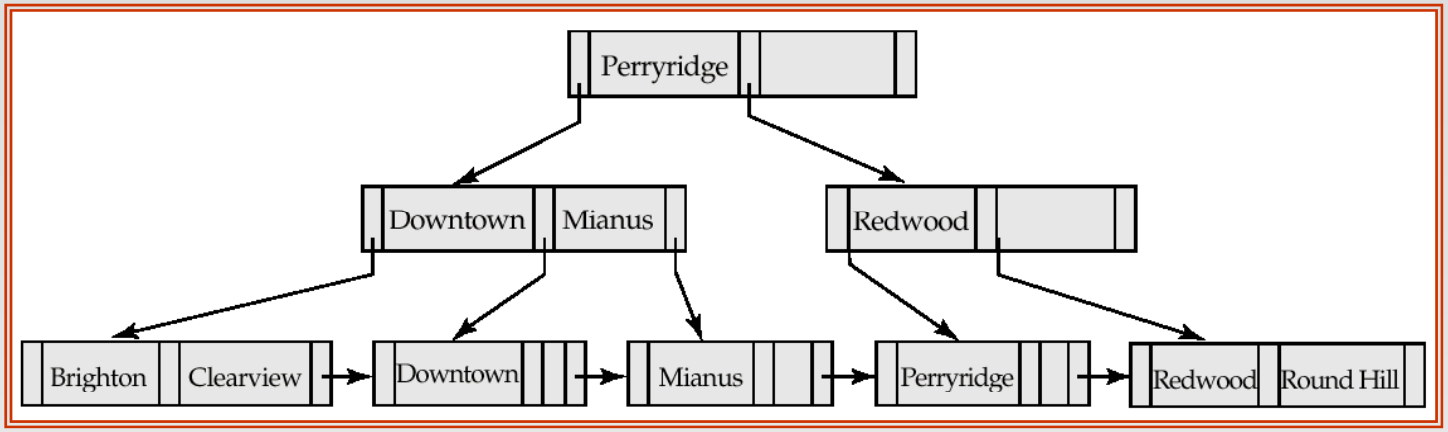
Ενημερώσεις σε B+-Trees: Διαγραφή

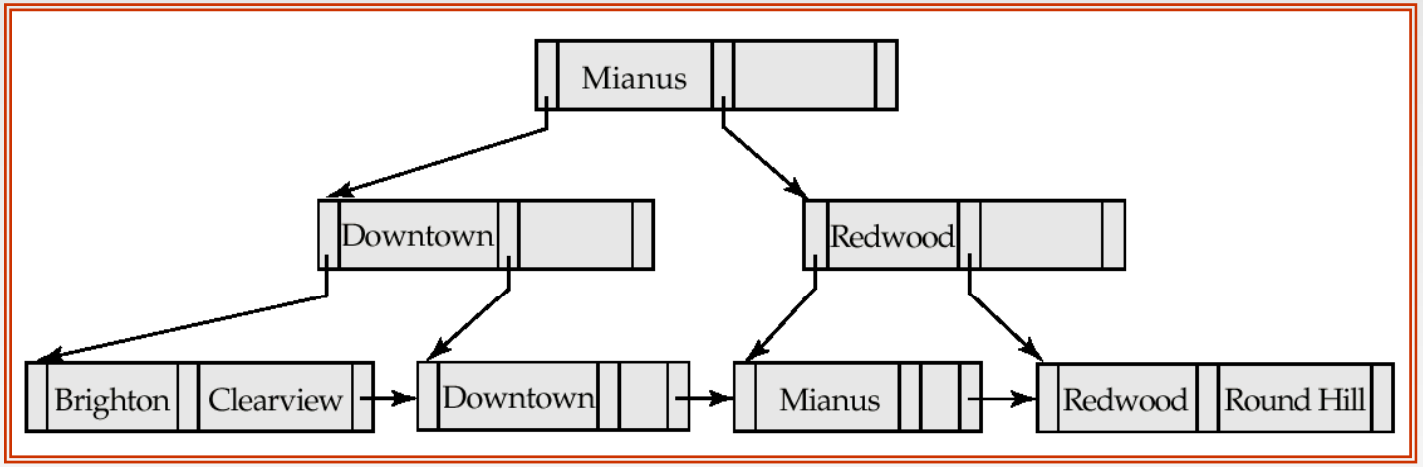


Διαγραφή του "Perryridge" στο αποτέλεσμα του προηγούμενου παραδείγματος

- Το φύλλο "Perryridge" γίνεται underfull (συγκεκριμένα άδειο) και συγχωνεύεται με το διπλανό (στα δεξιά του) φύλλο - αδελφό του.
- Έτσι ο κόμβος – πατέρας του "Perryridge" γίνεται και αυτός underfull, και συγχωνεύεται με τον διπλανό (στα αριστερά του) κόμβο - αδελφό του.
- Τώρα η ρίζα έχει ένα μόνο παιδί, έτσι διαγράφεται και το παιδί της γίνεται η νέα ρίζα του δέντρου.

Ενημερώσεις σε B+-Trees: Διαγραφή



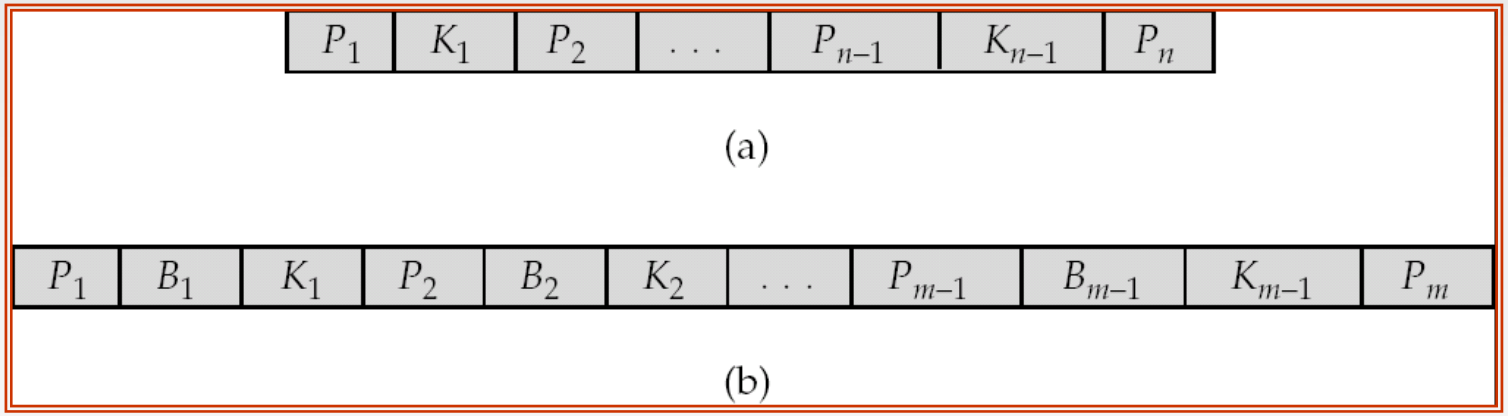


Πριν και μετά τη διαγραφή του “Perryridge” στο πρώτο παράδειγμα

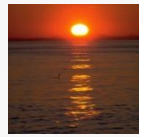
- Με τη διαγραφή του φύλλου Perryridge ο πατέρας του (Redwood) έχει ένα μόνο παιδί, οπότε γίνεται ανακατανομή και συγκεκριμένα transfer του φύλλου mianus δεξιά, ώστε και οι δύο κόμβοι (Downtown και Redwood αντίστοιχα), να έχουν τον ίδιο αριθμό παιδιών (2)
- Αλλάζει βέβαια η Search-key τιμή στη ρίζα

Ευρετήρια B-Tree

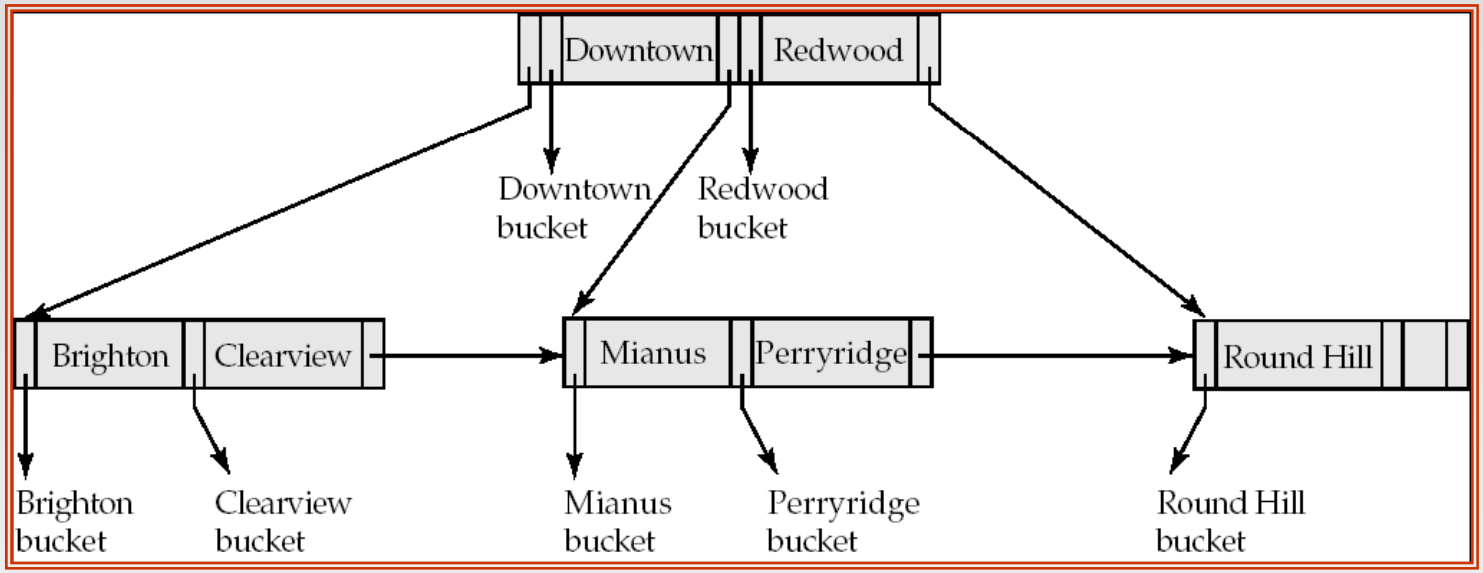
- Όμοιο με το B+-tree, μόνο που το B-tree επιτρέπει search-key τιμές να εμφανίζονται μόνο μία φορά. Εξαλείφει έτσι την πλεονάζουσα πληροφορία των κλειδιών αναζήτησης (search keys).
- Τα Search keys των εσωτερικών κόμβων δεν εμφανίζονται πουθενά αλλού μέσα στο B-tree; Ένας επιπλέον δείκτης για κάθε search key εσωτερικού κόμβου, πρέπει να προστεθεί.
- Γενικευμένο B-tree φύλλο



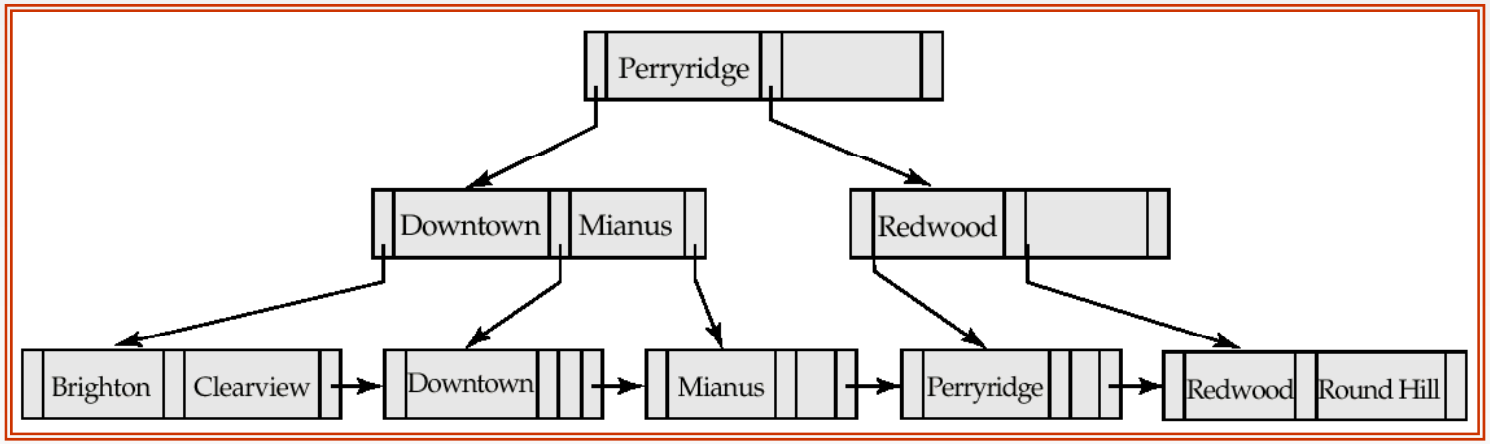
- Των εσωτερικών κόμβων οι pointers B_i είναι τα γνωστά bucket ή file record pointers.



Ευρετήρια B-Tree (παράδειγμα)

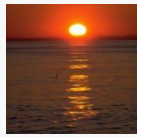


B-tree (πάνω) και B+-tree (κάτω) στα ίδια δεδομένα



Ευρετήρια B-Tree (Συν.)

- Πλεονεκτήματα των B-Tree ευρετηρίων:
  - Χρησιμοποιεί λιγότερους κόμβους από ότι το B^+-Tree.
  - Μερικές φορές βρίσκει το κλειδί αναζήτησης χωρίς να φτάσει μέχρι τα φύλλα.
- Μειονεκτήματα των B-Tree ευρετηρίων :
  - Μόνο ένα μικρό ποσοστό των κλειδιών αναζήτησης βρίσκεται σχετικά νωρίς
  - Οι ενθέσεις και οι διαγραφές είναι πιο περίπλοκες από ότι στα B^+-Trees
  - Πιο δύσκολη υλοποίηση από αυτήν των B^+-Trees.
- Επειδή στις ΒΔ το μεγάλο βάρος πέφτει στα update operations, τα B^+-Trees θεωρούνται πιο αποδοτικά.



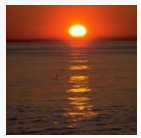
Προσπέλαση πολλαπλών κλειδιών

- Σε μερικούς τύπους ερωτημάτων πρέπει να χρησιμοποιήσουμε πολλαπλά ευρετήρια.
- Παράδειγμα:

  select *account_number*

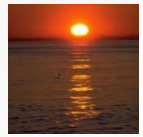
  from *account*

  where *branch_name* = “Perryridge” **and** *balance* = 1000
- Δυνατές στρατηγικές επεξεργασίας του ερωτήματος χρησιμοποιώντας ευρετήρια σε απλά πεδία του πίνακα account:
  1. Χρησιμοποίησε index on *branch_name* για να βρεις λογαριασμούς στο Perryridge. Κάνε τον έλεγχο balance = *1000*.
  2. Χρησιμοποίησε index on *balance* για να βρεις λογαριασμούς με υπόλοιπο $1000; Κάνε τον έλεγχο *branch_name* = “Perryridge”.
  3. Χρησιμοποίησε index on *branch_name* για να βρεις λογαριασμούς στο Perryridge. Ομοίως κάνε χρήση του index on *balance*. Πάρε την τομή τους.



Ευρετήρια σε πολλαπλά κλειδιά

- **Σύνθετα κλειδιά αναζήτησης** είναι κλειδιά αναζήτησης που περιέχουν περισσότερα από ένα πεδία (attributes).
  - π.χ. (*branch_name, balance*)
- Λεξικογραφική Διάταξη: $(a_1, a_2) < (b_1, b_2)$ αν είτε
  - $a_1 < b_1$, ή
  - $a_1 = b_1$ και $a_2 < b_2$

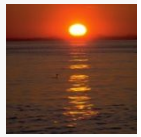


Ευρετήρια σε πολλαπλά κλειδιά

Υποθέστε ότι έχουμε ένα ευρετήριο στο σύνθετο κλειδί (*branch_name, balance*).

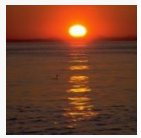
- με την **where** πρόταση **where** *branch_name* = “Perryridge” **and** *balance* = 1000
  το index on (*branch_name, balance*) θα επιστρέψει records τα οποία ικανοποιούν και τις 2 συνθήκες.
  - Είναι λιγότερο αποδοτικό αν χρησιμοποιήσουμε ξεχωριστά ευρετήρια — μπορεί να πάρουμε πολλά records που να ικανοποιούν τη μία μόνο συνθήκη.

- Έτσι, π.χ. μπορούμε αποδοτικά να χειριστούμε την πρόταση: **where** *branch_name* = “Perryridge” **and** *balance* < 1000



Στατικός κατακερματισμός

- Στο **στατικό κατακερματισμό** (static hashing) δεσμεύεται εξαρχής ένας αριθμός από κάδους (buckets).
- Η συνάρτηση κατακερματισμού **h(K)** είναι μία **συνάρτηση** από το σύνολο των τιμών του κλειδιού διάταξης K στο σύνολο των διευθύνσεων των κάδων B.
- Η συνάρτηση κατακερματισμού χρησιμοποιείται για την αναζήτηση, εισαγωγή αλλά και τη διαγραφή εγγραφών.
- Εγγραφές με διαφορετικές τιμές του κλειδιού διάταξης μπορεί να τοποθετηθούν στον ίδιο κάδο. Επομένως, όλος ο κάδος πρέπει να **σαρωθεί σειριακά** για την εύρεση μιας εγγραφής.



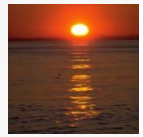
Παραδείγματα Κατακερματισμού

Κατακερματισμός στο αρχείο **account**, με κλειδί το **branch-name** (δες επόμενη διαφάνεια.)

- Υπάρχουν 10 κάδοι (buckets), 4 εγγραφές / κάδο
- Έστω, η δυαδική αναπαράσταση του i-οστού χαρακτήρα είναι ο ακέραιος j. Η συνάρτηση κατακερματισμού επιστρέφει το άθροισμα των δυαδικών αναπαραστάσεων των χαρακτήρων modulo 10

Παράδειγμα:

- h(Perryridge) = 5
- h(Round Hill) = 3



Παραδείγματα Κατακερματισμού

bucket 0

bucket 1

bucket 2

bucket 3

A-217	Brighton	750
A-305	Round Hill	350

bucket 4

A-222	Redwood	700

bucket 5

A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700

bucket 6

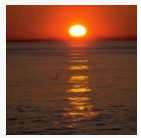
bucket 7

A-215	Mianus	700

bucket 8

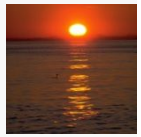
A-101	Downtown	500
A-110	Downtown	600

bucket 9



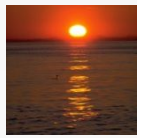
Συναρτήσεις κατακερματισμού

- Η **χειρότερη** συνάρτηση κατακερματισμού αντιστοιχεί όλες τις τιμές του κλειδιού διάταξης στον ίδιο κάδο.
- Μια **ιδανική** συνάρτηση κατακερματισμού είναι **ομοιόμορφη**, δηλαδή σε κάθε κάδο ανατίθεται ο ίδιος αριθμός από τιμές του κλειδιού διάταξης από το σύνολο όλων των πιθανών τιμών.
- Η ιδανική συνάρτηση κατακερματισμού είναι **τυχαία**, έτσι ώστε κάθε κάδος να έχει τον ίδιο αριθμό από εγγραφές ανεξάρτητα από την **πραγματική κατανομή** των τιμών του κλειδιού διάταξης στο αρχείο.
- Οι συνηθισμένες συναρτήσεις κατακερματισμού υπολογίζουν την εσωτερική δυαδική αναπαράσταση του κλειδιού διάταξης.

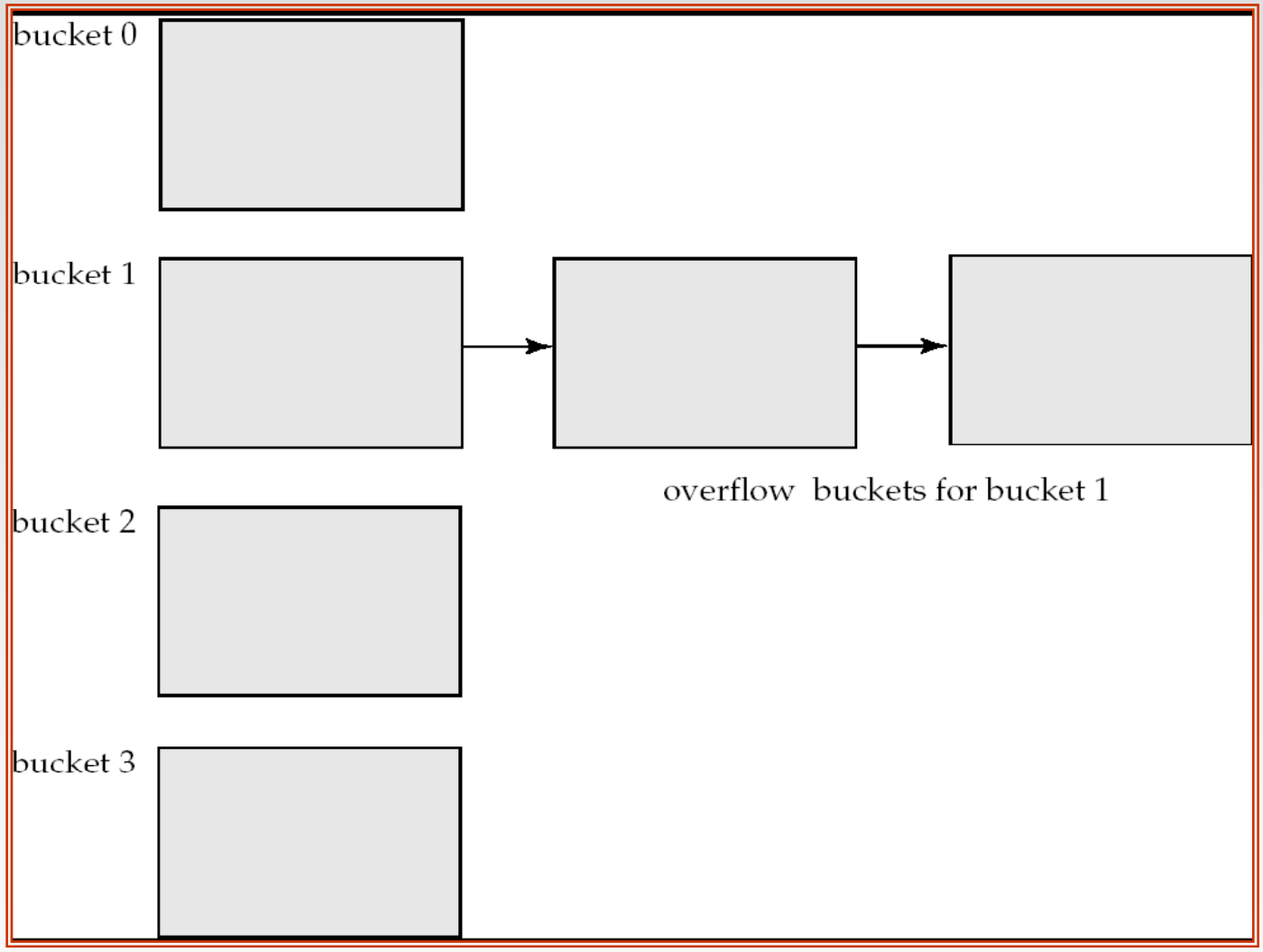


Χειρισμός των Bucket Overflows

- **Σύγκρουση** (collision) συμβαίνει όταν μια νέα εγγραφή κατακερματίζεται σε έναν ήδη γεμάτο κάδο λόγω:
  - ανεπάρκειας των κάδων (λίγοι σε σχέση με τον όγκο της ΒΔ ή
  - ασυμμετρίας στην κατανομή των εγγραφών.
- Επίλυση συγκρούσεων:
  - **Ανοιχτή Διευθυνσιοδότηση** (open addressing): τοποθέτηση στην επόμενη κενή θέση
  - **Αλυσιδωτή Σύνδεση** (chaining): συνδεδεμένη λίστα με κάδους υπερχείλισης (overflow buckets)
  - **Πολλαπλός Κατακερματισμός** (multiple hashing): εφαρμογή δεύτερης συνάρτησης κατακ/μού

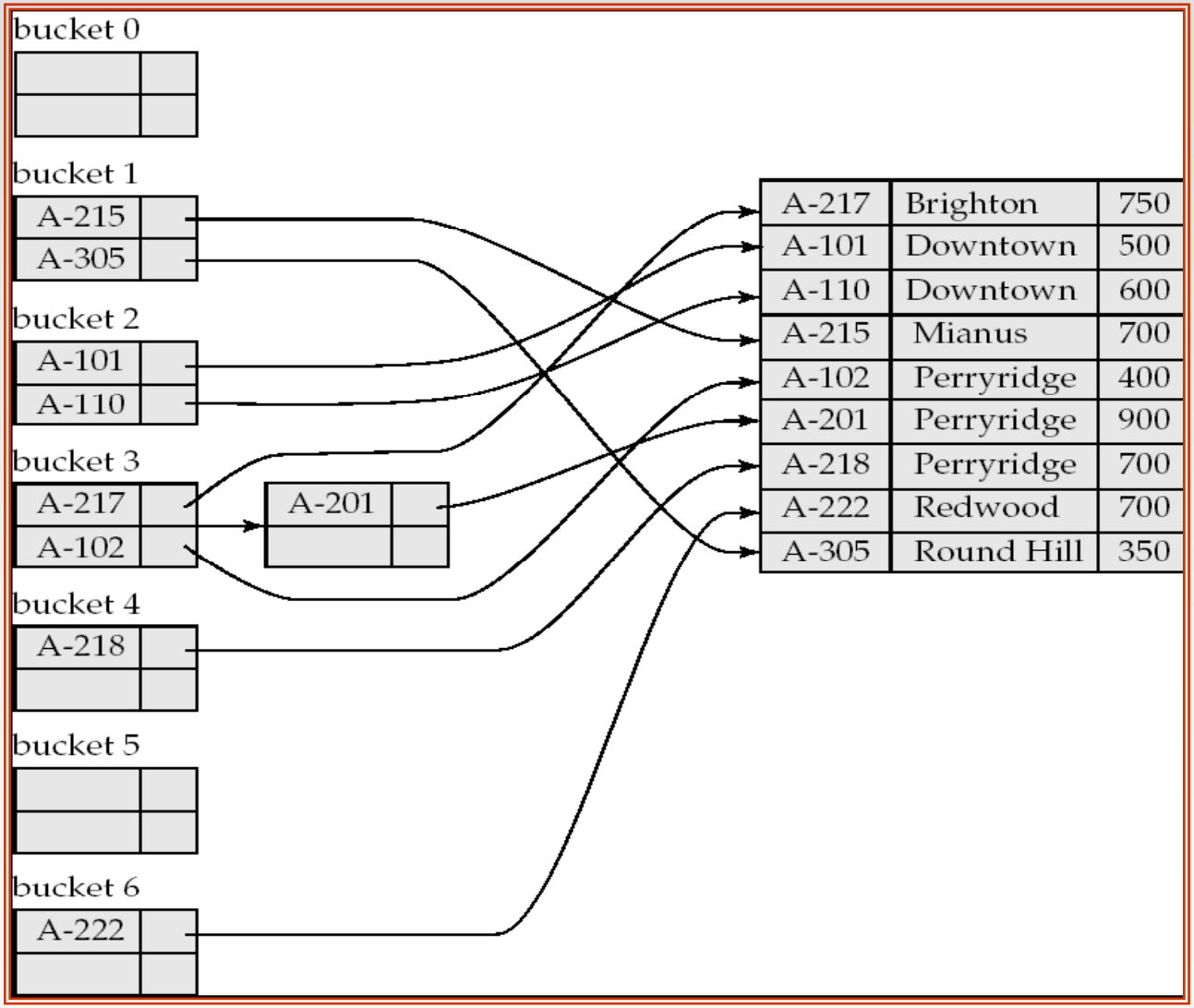


Χειρισμός των Bucket Overflows



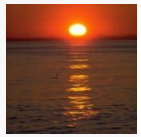
Hash Ευρετήρια

- Το Hashing χρησιμοποιείται όχι μόνο για οργάνωση αρχείων, αλλά και ως τεχνική δόμησης αποδοτικών ευρετηρίων.



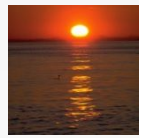
Μειονεκτήματα στατικού κατακ/μού

- Το σύνολο των διευθύνσεων κάδων είναι προκαθορισμένο.
  - Οι βάσεις δεδομένων μεγαλώνουν με το χρόνο. Αν ο αρχικός αριθμός κάδων είναι πολύ μικρός, η απόδοση θα μειωθεί λόγω των αλυσίδων υπερχείλισης.
  - Έστω και εάν είναι προβλέψιμο το μελλοντικό μέγεθος των αρχείων και προσδιοριστεί ανάλογα ο αριθμός των κάδων, στην αρχή θα σπαταλάται άσκοπα σημαντικό ποσό χώρου.
  - Όταν συρρικνώνεται η βάση δεδομένων, πάλι υπάρχει σπατάλη χώρου.
  - Μια λύση είναι η **περιοδική αναδιοργάνωση** του αρχείου με μια νέα συνάρτηση κατακερματισμού, αλλά αυτή η διαδικασία είναι πολύ ακριβή (σε χώρο και χρόνο).
- Αυτά τα προβλήματα αποφεύγονται με τεχνικές που επιτρέπουν να τροποποιείται δυναμικά ο αριθμός των κάδων
  - **Επεκτάσιμος κατακερματισμός** (extensible hashing).

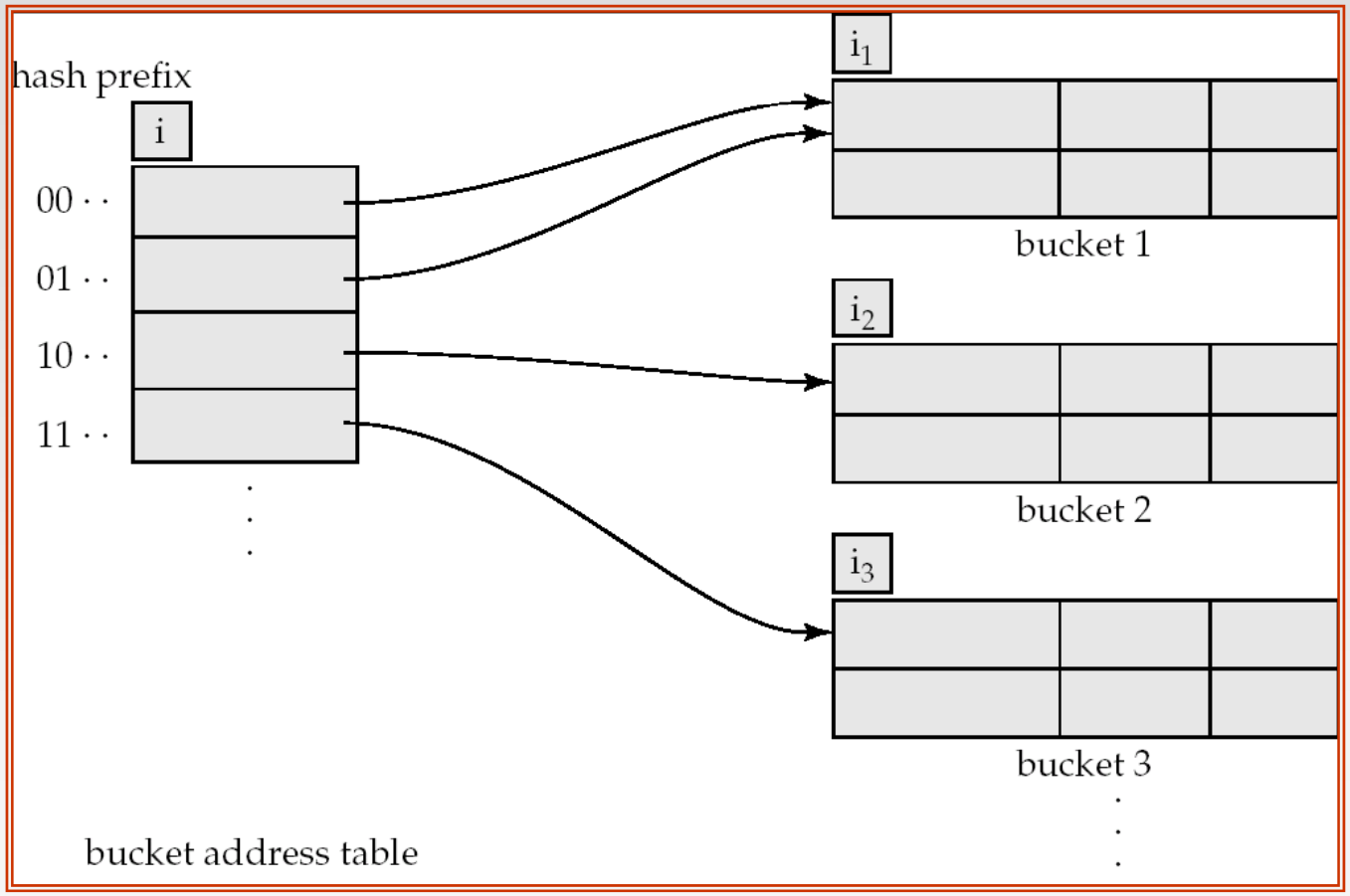


Δυναμικό Hashing

- Βολεύει για ΒΔ το μέγεθος των οποίων αυξομειώνεται
- Επιτρέπει στην hash συνάρτηση να τροποποιείται δυναμικά
- Μια μορφή δυναμικού hashing είναι το **Extendable hashing**
  - Η συνάρτηση Hash παράγει τιμές σε ένα μεγάλο εύρος — τυπικά ακεραίους των *b*-bits, όπου *b* = 32.
  - Ανά πάσα στιγμή μπορούμε να χρησιμοποιήσουμε το πρόθεμα (prefix) της συνάρτησης hash για να δεικτοδοτήσουμε τον πίνακα των διευθύνσεων των κάδων.
  - Έστω ότι το μήκος του προθέματος είναι *i* bits, $0 \leq i \leq 32$.
  - Bucket address table size = 2^i. Αρχικά $i = 0$
  - Το *i* αυξομειώνεται καθώς και η ΒΔ αυξομειώνεται.
  - Πολλαπλές καταχωρήσεις του bucket address table μπορεί να δείχνουν σε ένα bucket.
  - Άρα, ο ακριβής αριθμός των buckets < 2^i
    - Ο αριθμός των buckets επίσης αλλάζει δυναμικά εξαιτίας της συγχώνευσης ή της διάσπασης τους.



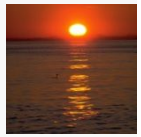
Γενική Extendable Hash Δομή



Σε αυτή τη δομή, $i_2 = i_3 = i$, όπου $i_1 = i - 1$ (για λεπτομέρειες δείτε επόμενη διαφάνεια)

Χρήση της Extendable Hash Δομής

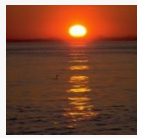
- Κάθε bucket *j* αποθηκεύει μία τιμή i_j . Όλα τα entries που δείχνουν στο ίδιο bucket έχουν τις ίδιες τιμές στα i_j bits.
- Για να βρούμε το bucket που περιέχει το κλειδί K_j:
  1. Υπολογίζουμε τη συνάρτηση $h(K_j) = X$
  2. Χρησιμοποιούμε τα πρώτα *i* bits της δυαδικής τιμής *X* ώστε να μετακινηθούμε στην κατάλληλη θέση του bucket address table, και ακολουθούμε τον κατάλληλο δείκτη προς τον κάδο
- Για να ενθέσουμε μία εγγραφή με τιμή K_j
  - Ακολουθούμε την ίδια διαδικασία εύρεσης με πριν για να εντοπίσουμε το κατάλληλο bucket, έστω *j*.
  - Αν υπάρχει χώρος στο bucket *j* ενθέτουμε εκεί μέσα την εγγραφή.
  - Διαφορετικά το bucket πρέπει να διασπαστεί και η ένθεση ξανά-επιχειρείται (δείτε επόμενη διαφάνεια).
    - Αντί αυτού, σε μερικές περιπτώσεις όπως θα δούμε χρησιμοποιούνται Overflow buckets



Ενημερώσεις της Extendable Hash Δομής

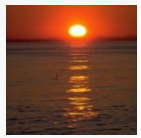
Για να κάνουμε split ένα bucket *j* όταν εντίθεται εγγραφή με κλειδί K_j:

- Αν $i > i_j$ (περισσότεροι από ένας δείκτες προς το bucket *j*)
  - Δεσμεύουμε ένα νέο bucket *z*.
  - Κάνουμε το δεύτερο μισό των entries του bucket address table που δείχνουν στο *j* να δείχνουν στο *z*
  - Διαγράφουμε και επανενθέτουμε κάθε εγγραφή στο bucket *j*.
  - Επαναϋπολογίζουμε ένα νέο bucket για το κλειδί K_j και ενθέτουμε την εγγραφή στο bucket (περαιτέρω splitting απαιτείται αν το bucket παραμένει ακόμη full)
- Αν $i = i_j$ (μόνο ένας δείκτης προς το bucket *j*)
  - Αυξάνουμε το *i* και διπλασιάζουμε το μέγεθος του bucket address table.
  - Αντικαθιστούμε κάθε 1 entry του table με 2 entries που δείχνουν στο ίδιο bucket.
  - Επαναϋπολογίζουμε ένα νέο bucket address table entry για το K_j Τώρα $i > i_j$ οπότε μπορούμε να χρησιμοποιήσιμε την παραπάνω (πρώτη) περίπτωση.



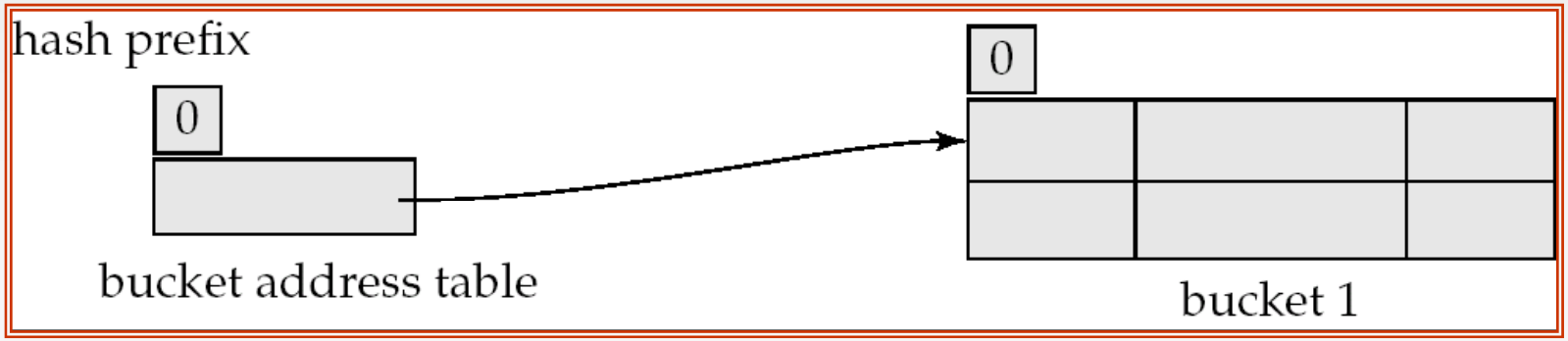
Ενημερώσεις της Extendable Hash Δομής (Συν.)

- Όταν ενθέτουμε μία τιμή, αν το bucket παραμένει full μετά από πολλά splits (που σημαίνει ότι το *i* αγγίζει το όριο *b*) δημιουργούμε ένα overflow bucket αντί να κάνουμε επαναλαμβανόμενα splittings στο bucket entry table.
- Για να διαγράψουμε ένα κλειδί,
  - Το εντοπίζουμε στο κατάλληλο bucket και το διαγράφουμε.
  - Αν το bucket γίνει empty, τότε θα πρέπει να διαγραφτεί (με τις κατάλληλες ενημερώσεις στο bucket address table).
  - Συγχώνευση από buckets μπορεί να γίνει (μπορεί να συγχωνευθεί μόνο με το "συνεταιρακι (buddy)" bucket που έχει την ίδια τιμή του i_j και το ίδιο i_j –1 πρόθεμα)
  - Μείωση στο μέγεθος του bucket address table μπορεί επίσης να γίνει
    - Σημ.: Η μείωση του μεγέθους του bucket address table είναι μία ακριβή πράξη και θα πρέπει να γίνεται μόνο στην περίπτωση που ο αριθμός των buckets γίνει πολύ μικρότερος από το μέγεθος του πίνακα.



Παράδειγμα

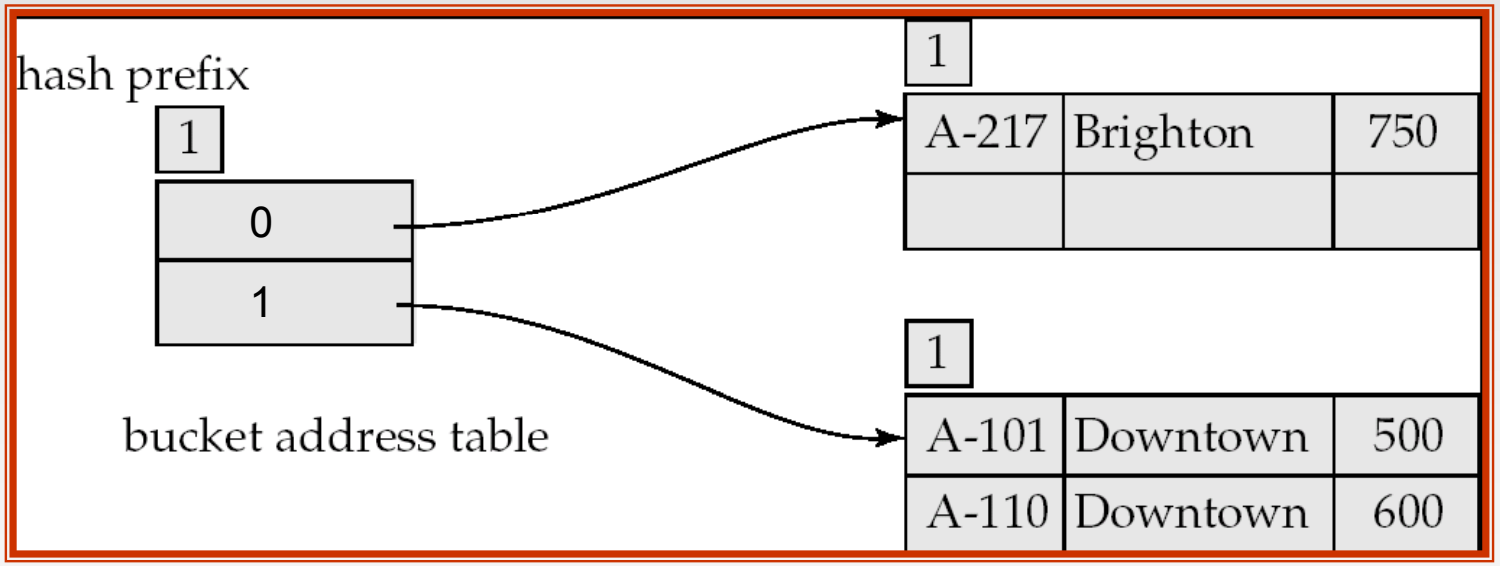
branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



Initial Hash structure, bucket size = 2

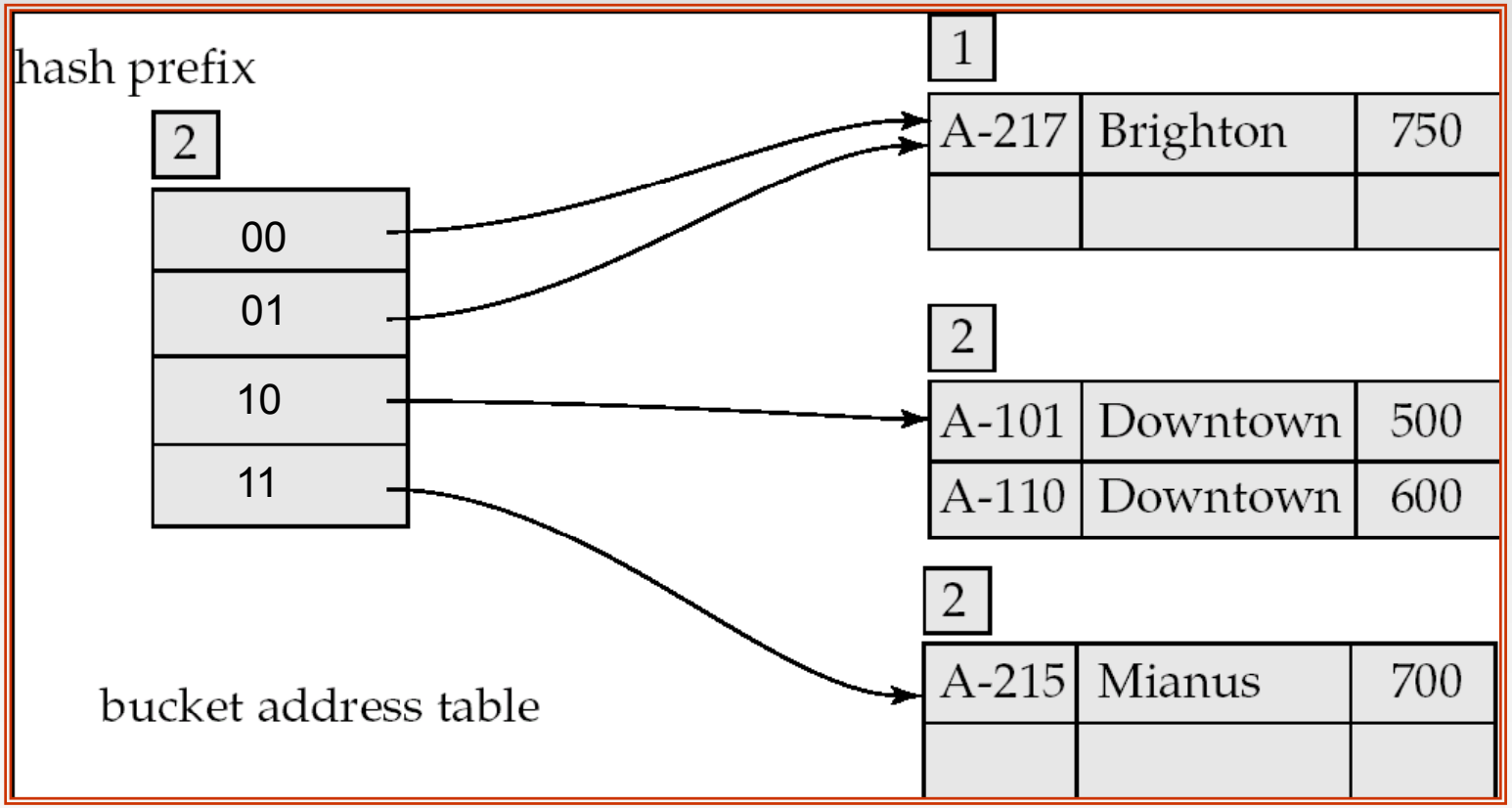
Παράδειγμα (Συν.)

- Η Hash δομή μετά την ένθεση ενός Brighton και δύο Downtown εγγραφών

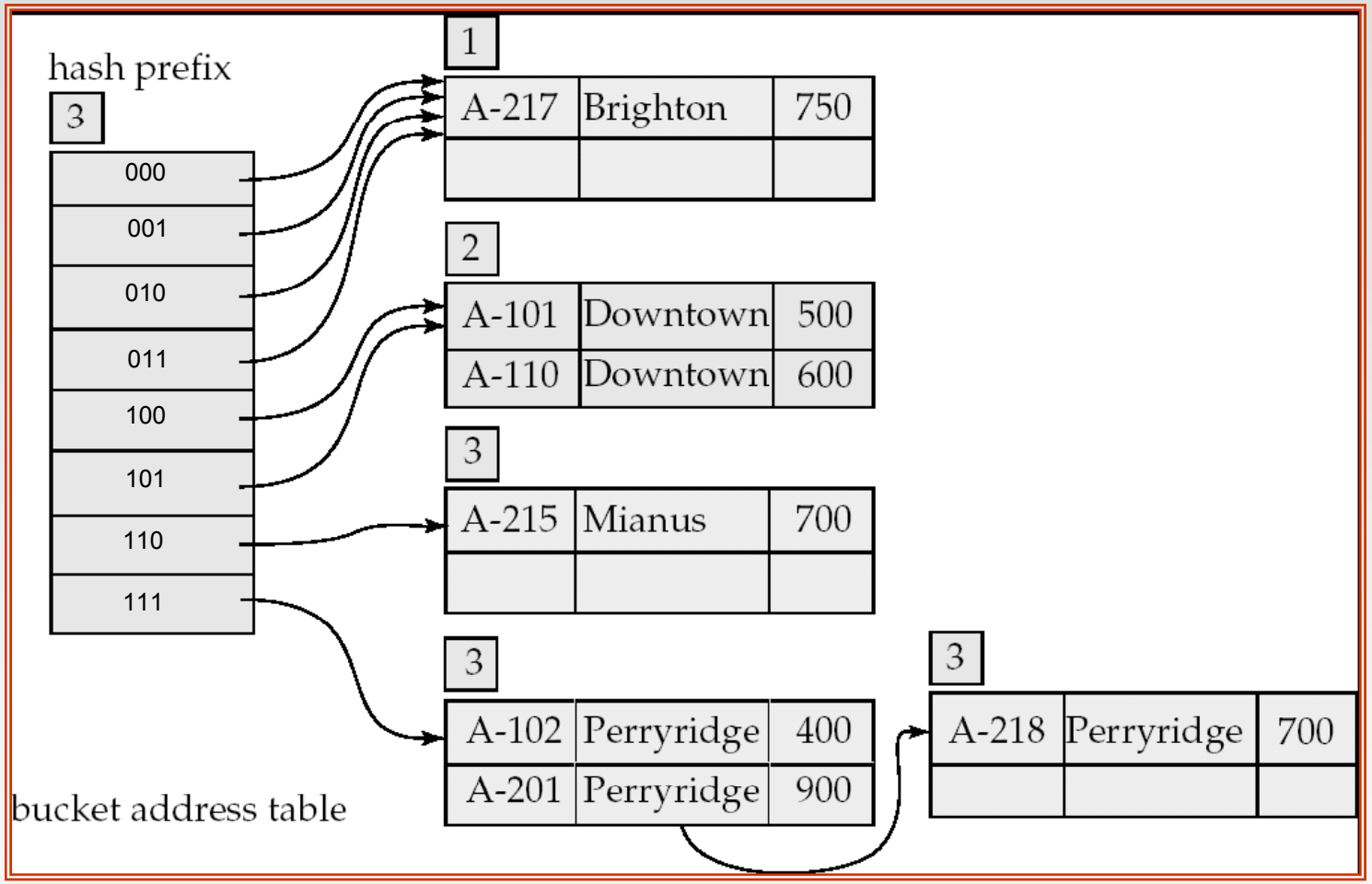


Παράδειγμα (Συν.)

Η Hash δομή μετά την ένθεση του Mianus record



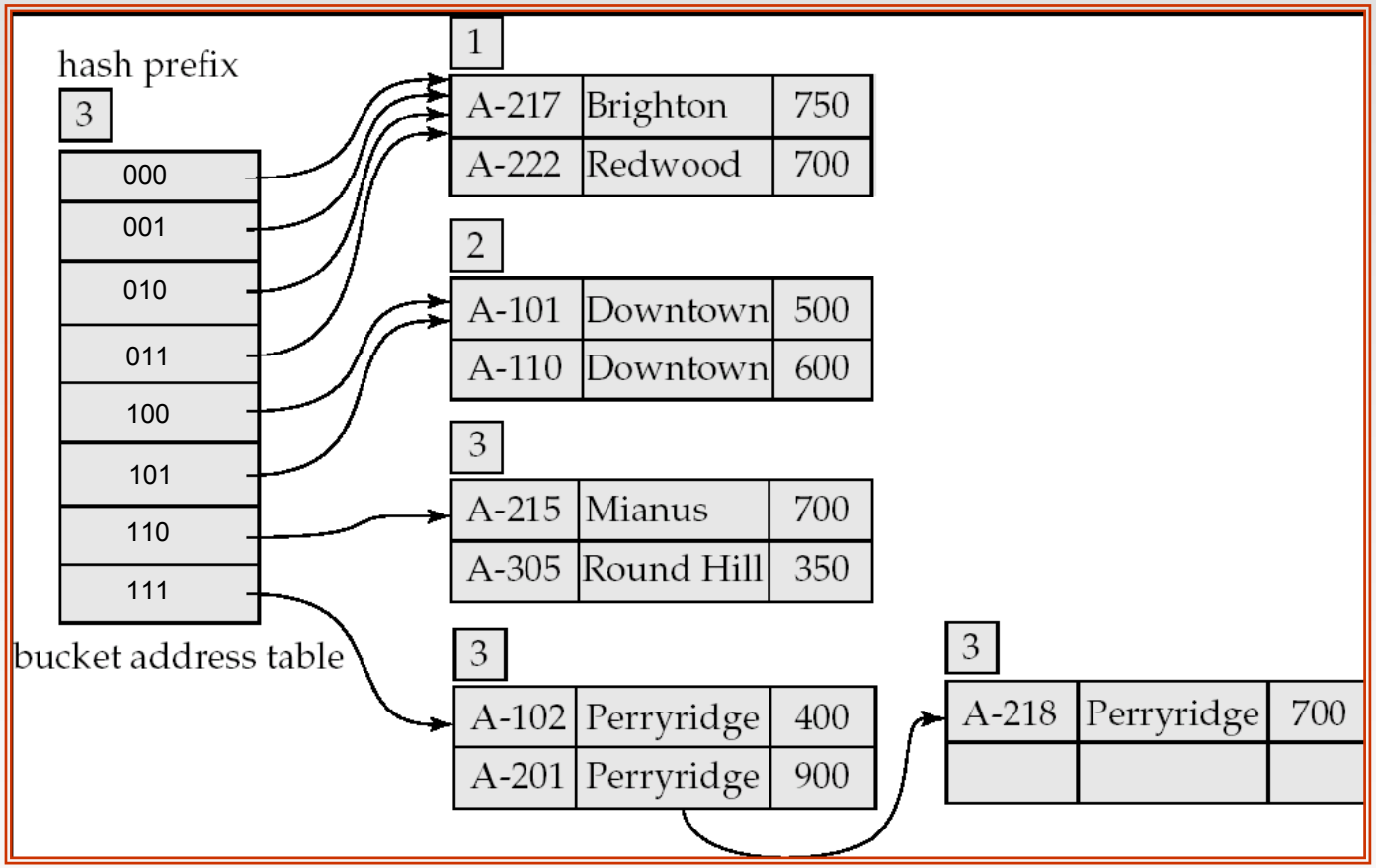
Παράδειγμα (Συν.)



Η Hash δομή μετά την ένθεση τριών Perryridge records

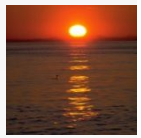
Παράδειγμα (Συν.)

- Η Hash δομή μετά την ένθεση των Redwood και Round Hill records



Διατεταγμένα ή κατακερματισμού ;

- Κόστος περιοδικής αναδιοργάνωσης.
- Σχετική συχνότητα εισαγωγών και διαγραφών.
- Αναμενόμενοι τύποι ερωτήσεων:
  - Ο κατακερματισμός είναι προτιμότερος για **ερωτήσεις ταυτότητας** πάνω στο κλειδί διάταξης.
  - Για **ερωτήσεις διαστήματος** πάνω στο κλειδί διάταξης επιλέγονται τα ταξινομημένα ευρετήρια.
    - Ο κατακερματισμός **δεν** μπορεί να υποστηρίξει ερωτήσεις διαστήματος



Ευρετήρια Bitmap

- Ένα απλό bitmap index σε ένα attribute έχει bitmap για κάθε τιμή του attribute
  - Το Bitmap έχει τόσα bits όσα είναι και τα records
  - Σε ένα bitmap για την τιμή v, το bit για το record είναι 1 αν το record για το attribute αυτό έχει την τιμή v, διαφορετικά είναι 0

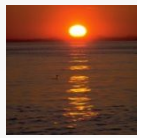
record number	*name*	*gender*	*address*	*income_level*
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

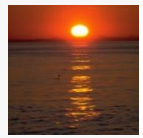


Ευρετήρια Bitmap (Συν.)

- Τα ευρετήρια Bitmap είναι χρήσιμα για queries σε multiple attributes
  - Όχι ιδιαίτερα χρήσιμα για single attribute queries
- Τα Queries απαντώνται χρησιμοποιώντας bitmap operations
  - Intersection (and)
  - Union (or)
  - Complementation (not)
- Κάθε operation παίρνει δύο bitmaps ιδίου μεγέθους και εφαρμόζει την operation στα αντίστοιχα bits για να πάρει το result bitmap
  - π.χ. 100110 AND 110011 = 100010

    100110 OR 110011 = 110111

    NOT 100110 = 011001
  - Males με income level L1: 10010 AND 10100 = 10000
    - Έτσι μπορούμε να ανακτήσουμε τα απαιτούμενα tuples.
    - Η αρίθμηση των matching tuples γίνεται επίσης πιο γρήγορα



Ορισμός Ευρετηρίων σε γλώσσα SQL

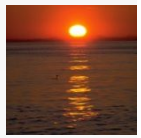
- Δημιουργία Ευρετηρίου

  create index <index-name> **on** <relation-name>
  (<attribute-list>)

  π.χ.: **create index** *b-index* **on** *branch(branch_name)*

- Χρησιμοποιούμε την πρόταση **create unique index** για να ορίσουμε έμμεσα ότι το search key είναι ένα candidate key.
  - Στην πραγματικότητα δεν απαιτείται αν υποστηρίζεται το λεγόμενο SQL **unique** integrity constraint.
- Για να διαγράψουμε ένα ευρετήριο γράφουμε

  drop index <index-name>



ΤΕΛΟΣ

Database System Concepts, 5th Ed.