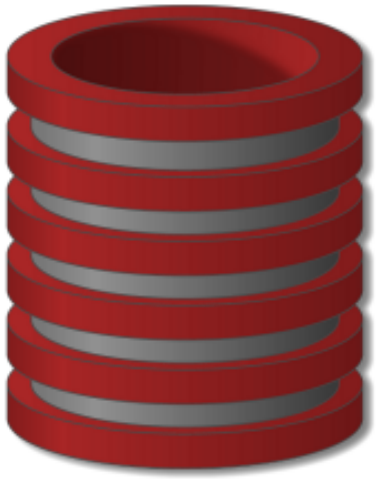


BAZE DE DATE

Indecși

@FII (2011-2012)

prezentat de Mihaela Elena Breabăn

Tematică curs

- Proiectarea bazelor de date relaționale
  - Normalizare și denormalizare
  - Modelul entitate-asociere, diagrame UML
  - Constrângeri și declanșatoare
  - View-uri
  - Indecși
- Procesarea interogărilor
- Managementul tranzacțiilor
- OLAP, Baze de date distribuite, NoSQL, Data Mining

Indecşi
Cuprins

Motivație

- Bazele de date consumă mult timp căutând

SELECT * FROM Student

WHERE sID=40

Cum putem regăsi rezultatul în următoarele situații:

a) Ordine aleatoare a datelor

sID	sNume	medie
20	Ioana	9.5
40	Andrei	8.66
10	Tudor	8.55
30	Maria	8.33
70	Alex	9.33

b) Date secvențiale/ordonate

sID	sNume	medie
10	Tudor	8.55
20	Ioana	9.5
30	Maria	8.33
40	Andrei	8.66
70	Alex	9.33

Ideea de bază

- Căutarea binară
  - Complexitate: $\log_2(N)$

    ($\log_2(100\ 000)=17$)

- Construirea unei structuri auxiliare care să ajute la localizarea unei înregistrări
  - Mecanismele de indexare sunt utilizate pentru a mări viteza de acces la datele dorite

Concepte de bază

- Un index este asociat cu o cheie de căutare – atribut sau set de atribute dintr-un fișier/tabel/relație utilizate pentru a căuta înregistrări în fișier
- Fișier index – constă din înregistrări index de forma

cheie de căutare	pointer

- Sortare:
  - a indexului pe baza cheii de căutare
  - a fișierului/tabelei/relației stocate

- Un fișier/tabel/relație poate avea asociați mai mulți indecși
- Fișierele index sunt de obicei de dimensiuni mult mai mici decât fișierul original cu date

Metrici de evaluare a indecşilor

- Timpul de acces
- Timpul de inserare
- Timpul de ștergere
- Spațiul necesar
- Tipurile de acces suportate eficient – influențează alegerea indexului
  - Înregistrări cu o valoare specificată a atributului
  - Înregistrări cu valoarea atributului inclusă într-un interval specificat

Tipuri de indecşi

- Indecși ordonați: cheile de căutare sunt stocate într-o anumită ordine
- Indecși hash: cheile de căutare sunt distribuite uniform în bucket-uri cu ajutorul unei funcții hash
  - Bucket: o unitate de stocare conținând una sau mai multe înregistrări
- Indecși bitmap: asociați cheilor de căutare de tip atribute discrete, reprezintă distribuția valorilor sub formă de matrice binară

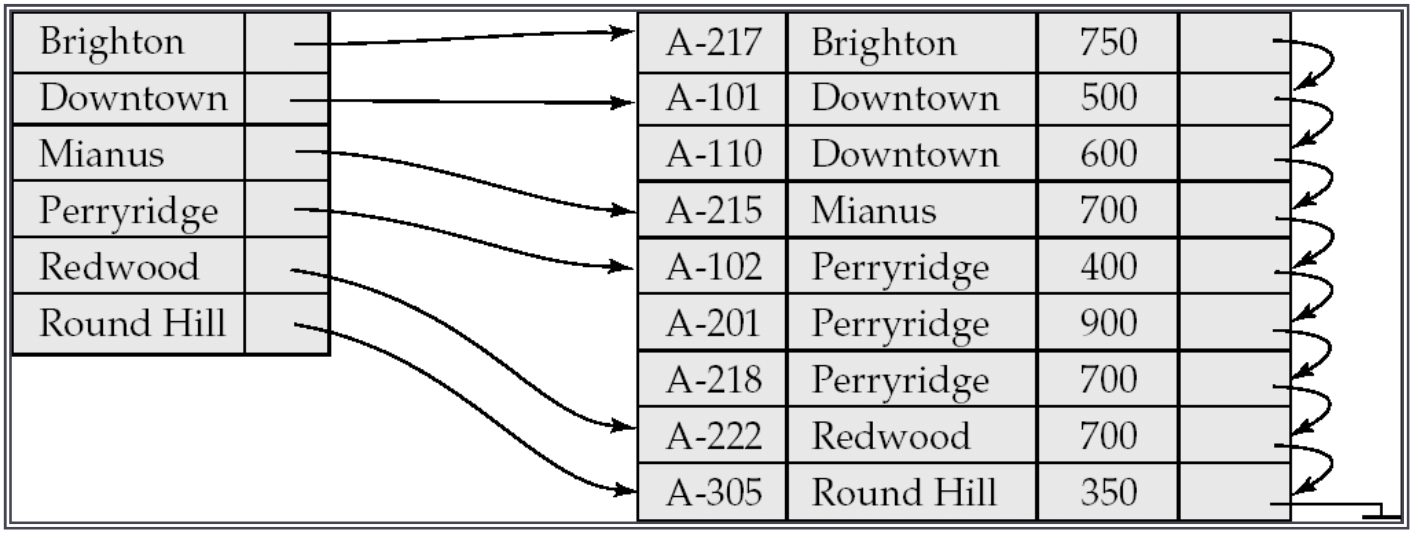
Indecşi secvenţiali

- Intrările index sunt sortate după cheia de căutare
  - Ex: catalogul cu autori într-o bibliotecă

- Index primar: indexul a cărui cheie de căutare definește și ordonarea secvențială a fișierului/tabelei/relației
  - denumit și index de grupare (clustering index)
  - cheia de căutare a unui index primar este de obicei cheia primară dar nu e obligatoriu!!!
  - o tabelă poate avea cel mult un index primar
- Index secundar (nonclustering): index a cărui cheie de căutare specifică o ordonare diferită de ordonarea secvențială a fișierului cu date
- Fișier index-secvențial: combinația fișier ordonat secvențial cu un index primar

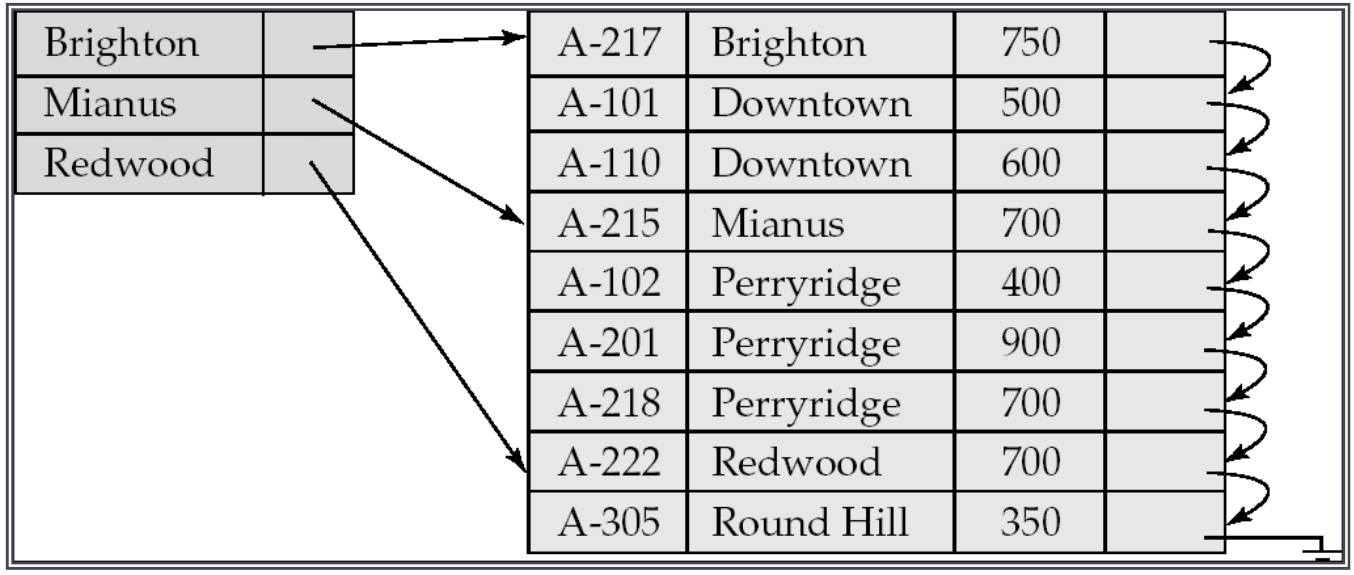
Fişiere index dense

- Index dens: există înregistrări index pentru fiecare valoare a cheii de căutare în fișier/tabel/relație
- Dacă indexul e primar va păstra câte un singur pointer-doar la prima intrare cu valoarea respectivă
- Dacă indexul e secundar vor fi necesari mai mulți pointeri la o singură valoare a cheii de căutare



Fişiere index rare

- Index rar: conține valori doar pentru unele valori a cheii de căutare
  - Aplicabil doar când înregistrările sunt ordonate secvențial după cheia de căutare
  - Balansul timp-spațiu
- Pentru a localiza o înregistrare cu valoarea k a cheii de căutare:
  - Se determină înregistrarea index cu cea mai mare valoare a cheii de căutare <k
  - Se caută secvențial în fișier începând cu înregistrarea spre care indică înregistrarea index



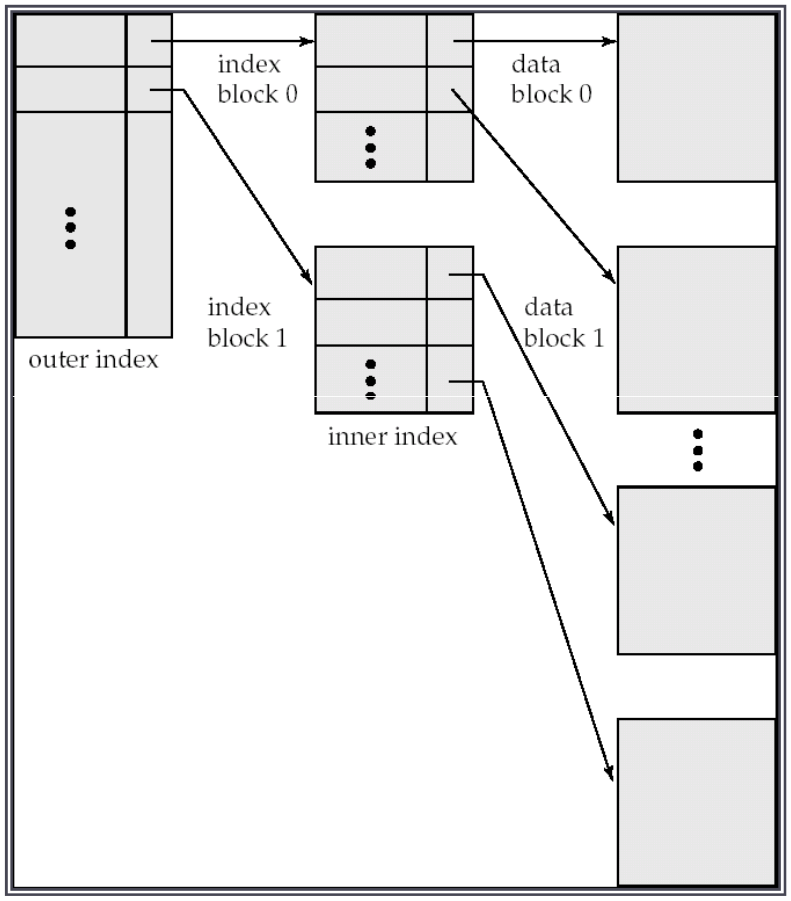
Indecşi multi-nivel

- Index multi-nivel: index asociat unui alt index
  - Dacă indexul primar nu încape în memorie accesul devine costisitor
  - Soluția: indexul primar păstrat pe disc este tratat ca un fișier secvențial și se construiește un index rar pentru el
- Indexul extern – un index rar al indexului primar
- Index intern – fișierul index primar

- Dacă și indexul extern este prea mare pentru a încăpea în memorie, se creează un index pe un nou nivel, etc…

- Indecșii de pe toate nivelele trebuiesc actualizați la inserare și ștergere în fișierul cu date

Indecşi multi-nivel



Actualizarea indecşilor secvenţiali
Ştergere

- Dacă înregistrarea ștearsă este singura care conține o valoare particulară a cheii de căutare, aceasta este ștearsă și din index

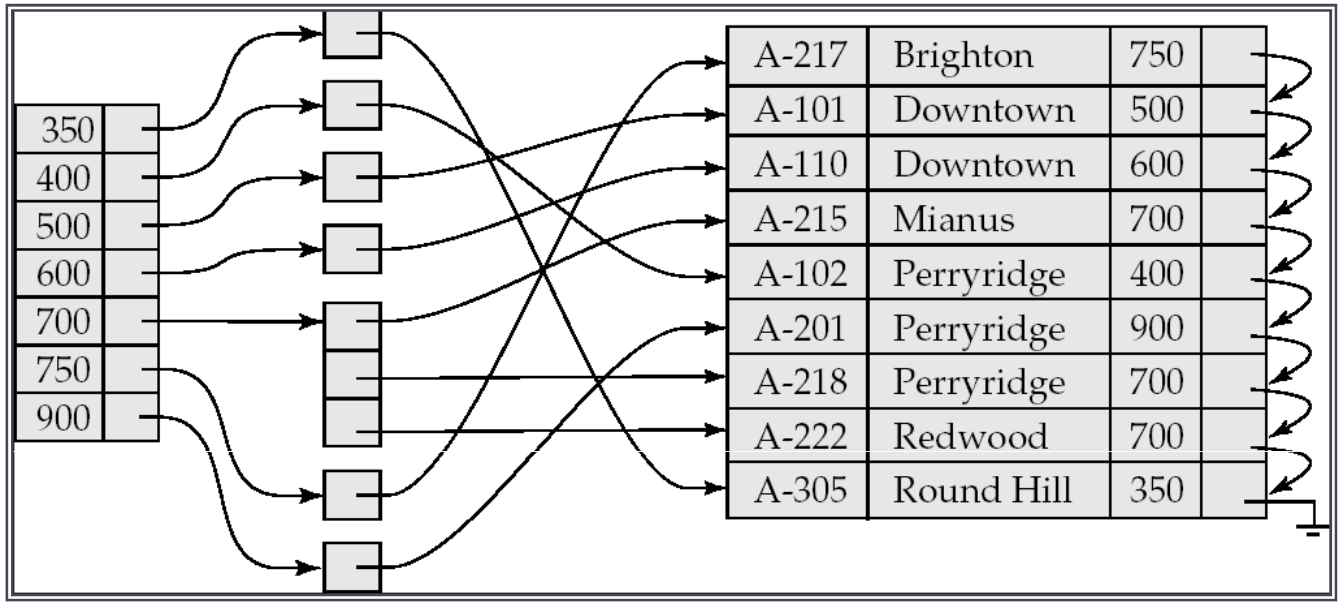
- Ștergerea în
  - Indecși denși: similară ștergerii din fișierul cu date
  - Indecși rari:
    - dacă există o intrare a cheii de căutare în index aceasta este înlocuită cu următoarea valoare a cheii de căutare din fișier (în ordinea cheii de căutare)
    - dacă următoarea valoare a cheii de căutare deja are o intrare în index, este efectuată ștergerea.

Actualizarea indecşilor secvenţiali
Inserare

- Este necesară localizarea valorii cheii de căutare ce apare în uplul inserat
- Indecşi denşi: dacă valoarea nu apare în index se va insera
- Indecşi rari: dacă indexul păstrează o intrare pentru fiecare bloc al fişierului nu sunt necesare modificări decât dacă un nou bloc este creat (prima valoare a cheii de căutare ce apare în noul bloc este inserată în index)

- Inserarea (şi ştergerea) pentru indecşi multi-nivel sunt extensii simple a algoritmilor uni-nivel prezentaţi

Indecşi secundari
Exemplu



- *În relația conturi sortată după filiale, care sunt conturile cu o balanță specificată?*
- Soluția: index secundar (dens!)
- Pentru a implementa relația de tip mulți-la-unu dintre index și datele destinație se utilizează referințe la bucket-uri de pointeri

Fişiere index de tip B^+-arbori
Motivație

- Organizarea secvențială a indecșilor se degradează pe măsură ce dimensiunea crește
- Reconstruirea indecșilor la intervale de timp e necesară însă costisitoare
- Indexul B^+-arbore
  - mărește viteza de localizare și elimină necesitatea constantă de reorganizare
  - utilizați extensiv

Structura B$^+$-arbore (1)

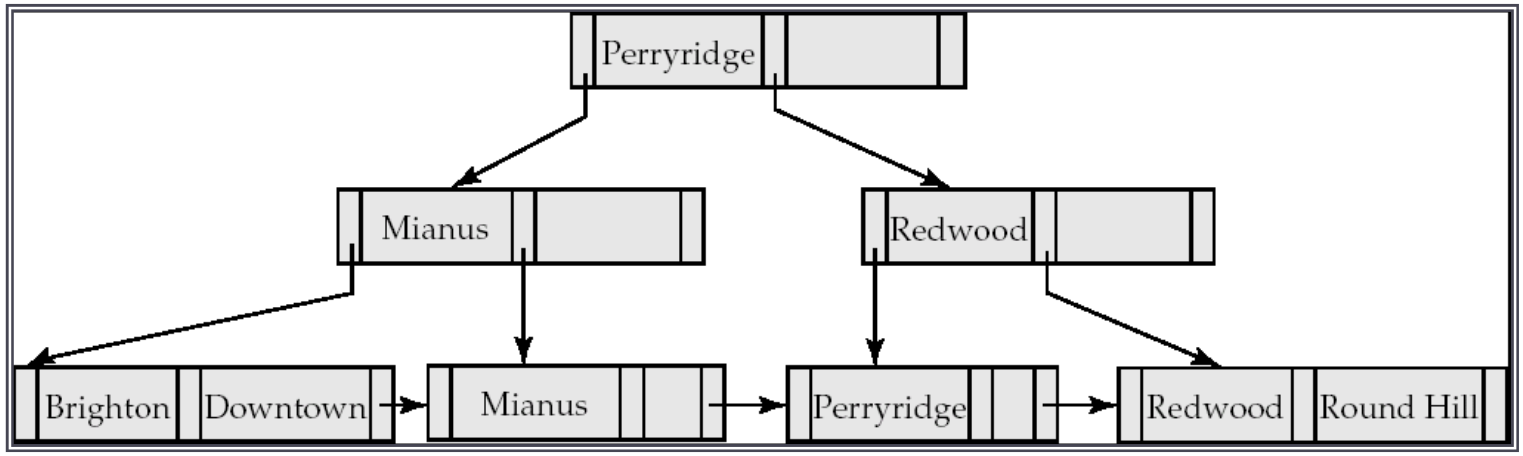
- Un arbore balansat astfel încât fiecare drum de la rădăcină la frunze are aceeași lungime
- Un nod tipic

  - K_i sunt valori a cheii de căutare
  - P_i sunt pointeri către
    - noduri copil/descendente (noduri care nu sunt frunze)
    - înregistrări sau bucket-uri de înregistrări (noduri frunză)
- Arborele este definit de o constantă n care specifică numărul maxim de valori dintr-un nod ca fiind (n-1) și numarul maxim de pointeri egal cu n
  - De obicei dimensiunea unui nod e cea a unui bloc
- Valorile cheii de căutare într-un nod sunt ordonate

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

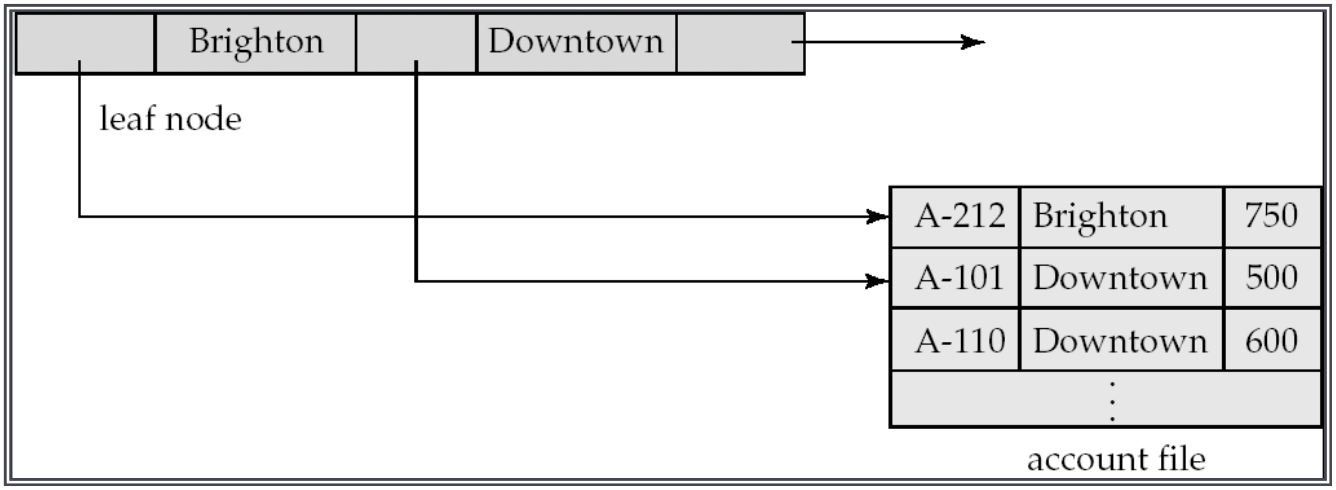
Structura B$^+$-arbore (2)

- Numărul de descendenți a unui nod este egal cu numărul de pointeri și e constrâns la o valoare între $\lceil n/2 \rceil$ și n
- Un nod frunză are între $\lceil (n-1)/2 \rceil$ și n-1 valori
- Cazuri speciale
  - Dacă rădăcina nu e frunză are măcar 2 descendenți
  - Dacă rădăcina e frunză poate avea între 0 și n-1 valori



Noduri frunză în B^+-arbore

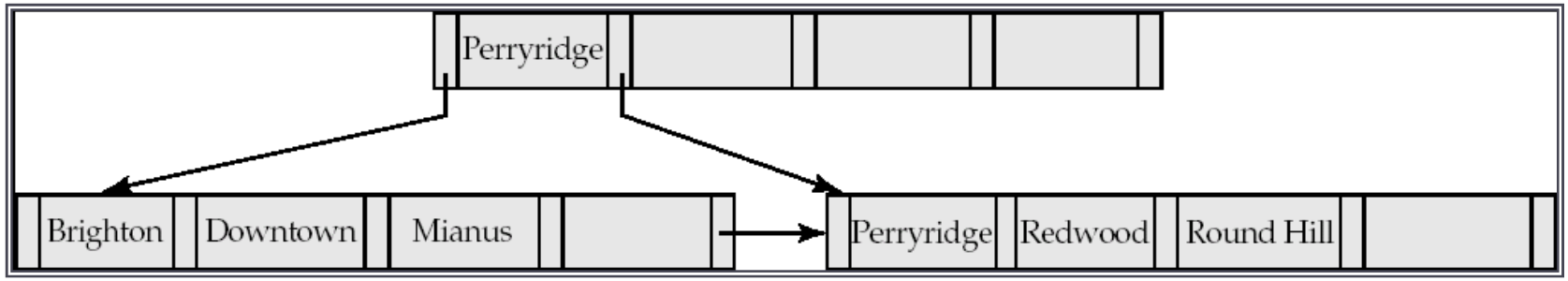
- Proprietăți a unui nod frunză
  - Fiecare pointer P_i dintre $P_1, \ldots, P_{n-1}$ indică spre o înregistrare în fișier/tabel/relație cu valoarea K_i, a cheii de căutare, sau spre un bucket de pointeri către înregistrări care au aceeași cheie de căutare K_i
  - Dacă nodurile N și M apar în această ordine de la stânga la dreapta, atunci valorile cheii de căutare din nodul N sunt mai mici decât cele din nodul M
  - P_n indică spre următorul nod în ordinea valorilor cheii de căutare



Noduri care nu sunt frunze în B^+-arbore

- formează un index rar multi-nivel pentru nodurile frunză
- Pentru un nod cu *n* pointeri:
  - Toate valorile cheii de căutare din subarborele spre care P_1 indică sunt mai mici decât K_1
  - Pentru P_i, $2 \leq i \leq n - 1$, toate valorile cheii de căutare din subarborele spre care indică sunt mai mari sau egale cu K_{i-1} și mai mici decât K_i
  - Toate valorile cheii de căutare din subarborele spre care indică P_n sunt mai mari sau egale cu K_{n-1}

B+-arbore
Exemplu



- n=5
  - Nodurile frunză au între 2 și 4 valori
  - Nodurile care nu sunt frunză au între 3 și 5 noduri copil
  - Rădăcina are măcar 2 noduri copil

B^+-arbori
Observații

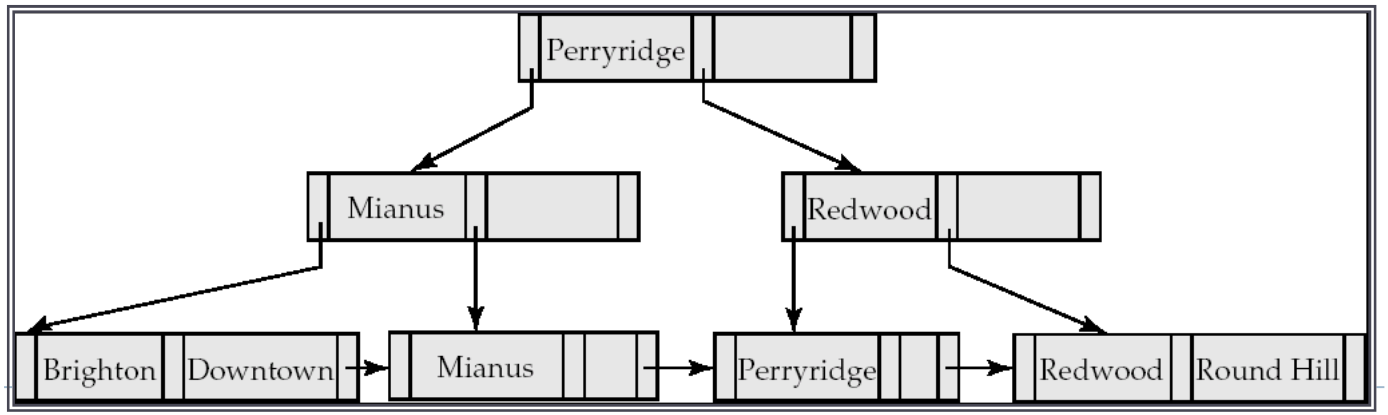
- Fiindcă conexiunile dintre noduri se realizează prin pointeri, blocuri apropiate logic nu trebuie să fie apropiate și fizic
- Nivelele diferite de nivelul frunză formează o ierarhie de indecși rari
- B^+-arborele conține un număr relativ mic de nivele
  - Cel mult $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$ pentru k valori a cheii de căutare
    - Nivelul imediat următor rădăcinii: cel puțin $2*\lceil n/2 \rceil$ valori
    - Următorul: cel puțin $2*\lceil n/2 \rceil * \lceil n/2 \rceil$
    - etc…
- Inserările și ștergerile se fac eficient, restructurarea indexului necesitând timp logaritmic

Interogări pe B^+-arbori
Algoritm

Determinarea tuturor înregistrărilor cu valoarea *k* a cheii de căutare

1. N=rădăcina
2. Repetă
   1. Caută în N cea mai mică valoare a cheii de căutare >*k*
   2. Dacă aceasta există și e egală cu K_i, atunci $N = P_i$
   3. Altfel $N = P_n$ $(k \geq K_{n-1})$

   Până N este nod frunză
3. Dacă există $K_i = k$, pointerul Pi indică înregistrarea dorită
4. Altfel nu există înregistrarea cu cheia de căutare *k*



Interogări pe B^+-arbori
Observații

- Un nod este în general de aceeași dimensiune ca a unui bloc pe disc - 4Kbytes
- Pt. fiecare intrare în index se utilizează 40 bytes (n=100)
- Fiecare acces a unui nod poate necesita o citire pe disc (<20 milisecunde)

- *Pentru 1 milion valori a cheii de căutare și n=100, câte noduri (blocuri pe disc) sunt accesate la o căutare în B^+-arbore?*
- *Dar dacă se utilizează un index secvențial?*

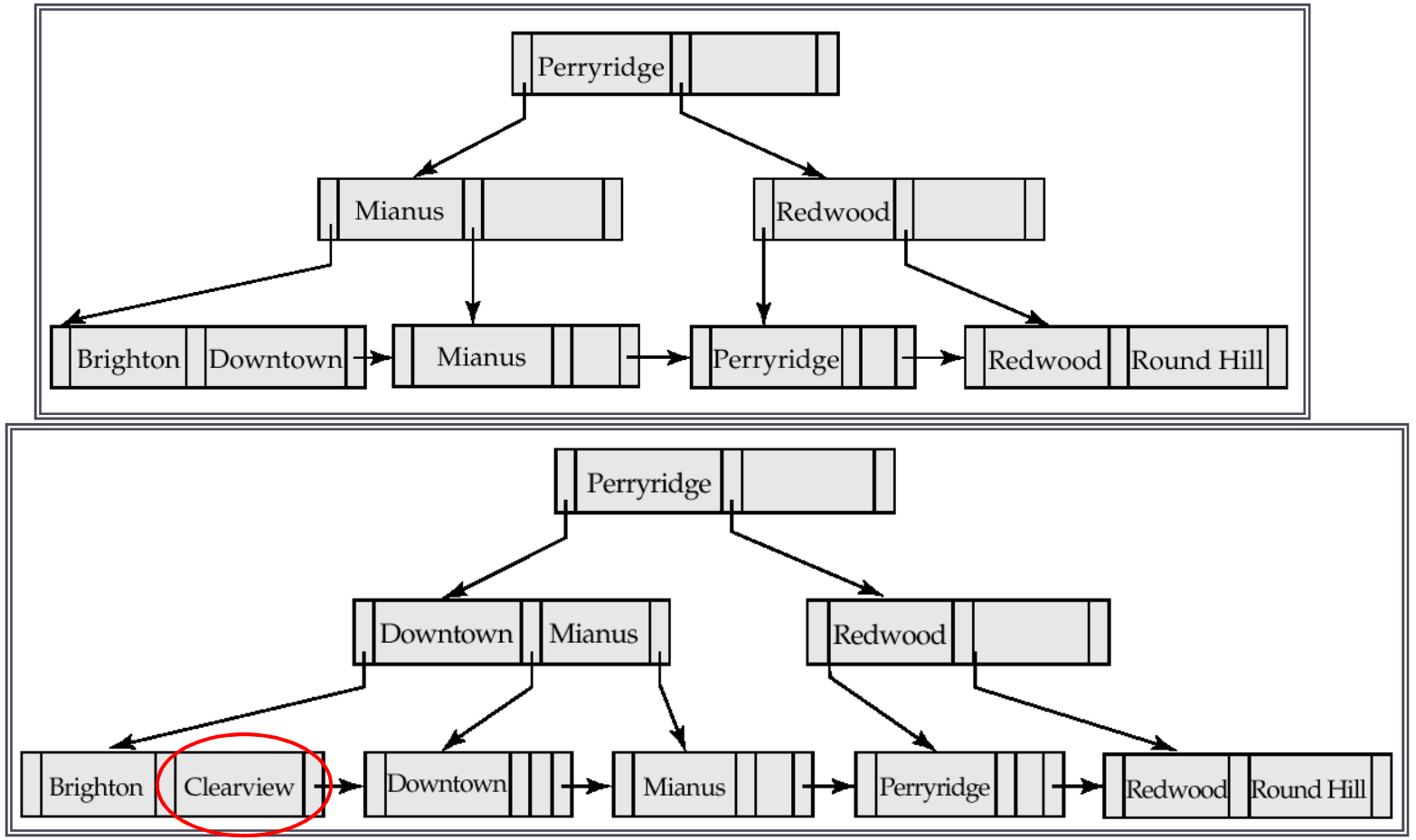
Actualizări în B^+-arbori
Inserarea

1. Determină nodul frunză în care va apărea valoarea cheii de căutare
2. Dacă valoarea e deja prezentă într-un nod frunză
    1. Adaugă înregistrarea în fișier/tabel/relație
    2. Dacă e necesar adaugă un pointer în bucket
3. Dacă nu e prezentă valoarea
    1. Adaugă înregistrarea în fișier/tabel/relație
    2. Dacă e loc în nodul frunză inserează perechea (valoare cheie, pointer)
    3. Altfel divide nodul

Actualizări în B^+-arbori
Inserarea: divizarea nodurilor

- Divizarea unui nod frunză
  1. Se iau cele n perechi (inclusiv cea care urmează a fi inserată) ordonate. În nodul original se pun primele $\lceil n/2 \rceil$ perechi iar restul într-un nod nou
  2. Fie p noul nod și k cea mai mică valoare din p. Inserează (k,p) în părintele nodului care se divide
  3. Dacă părintele este plin acesta se divide la rândul său și divizarea se propagă în sus până când un nod nu este plin. În cel mai rău caz nodul rădăcină este divizat ceea ce crește înălțimea arborelui cu 1.
- Divizarea unui nod plin intern N la inserarea unei perechi (k,p)
  1. Se crează un nod temporar M cu spațiu pentru n+1 pointeri și n valori în care se copie N și perechea (k,p)
  2. Se copie $P_1, K_1, \ldots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ din M înapoi în N
  3. Se copie $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \ldots, K_n, P_{n+1}$ din M într-un nou nod N'
  4. Inserează $(K_{\lceil n/2 \rceil}, N')$ în părintele lui N

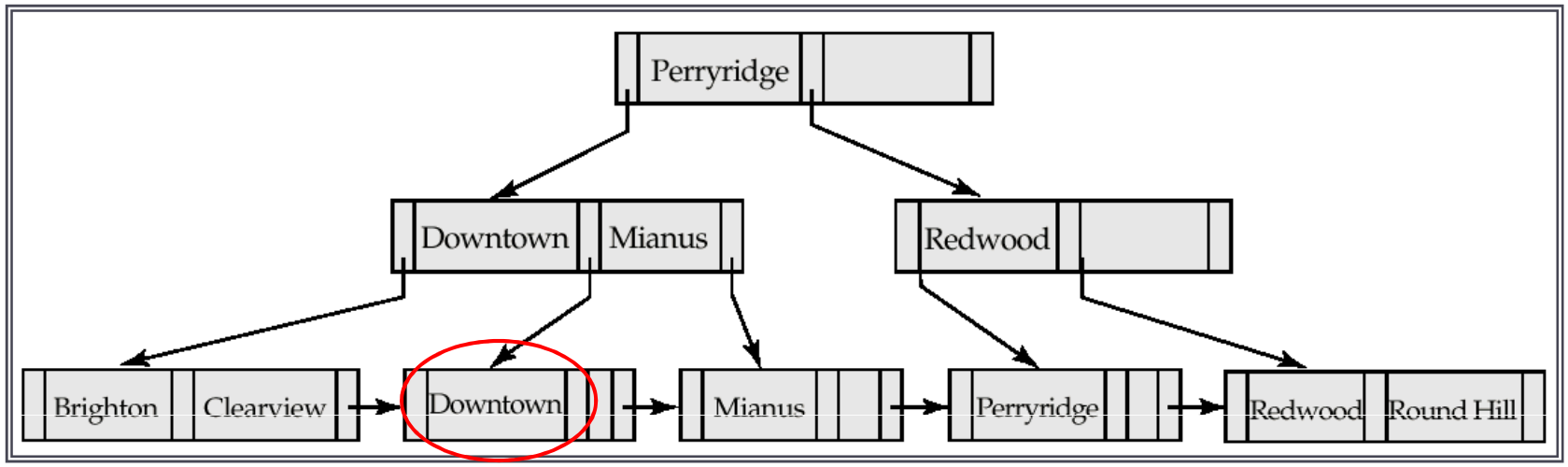
Actualizări în B$^+$-arbori
Inserarea: Exemplu

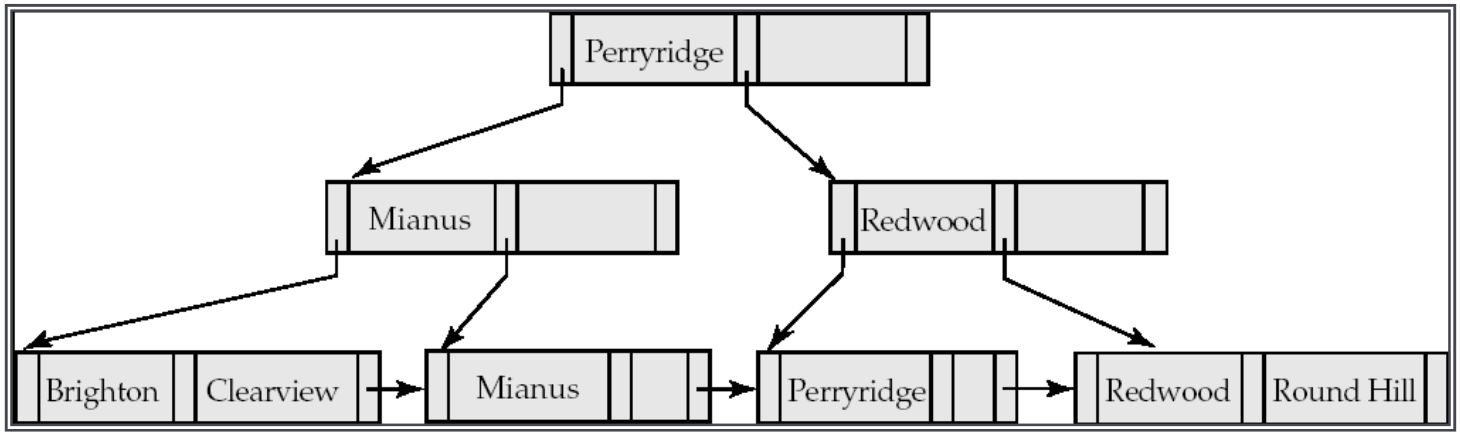


Actualizări în B^+-arbori
Ştergerea

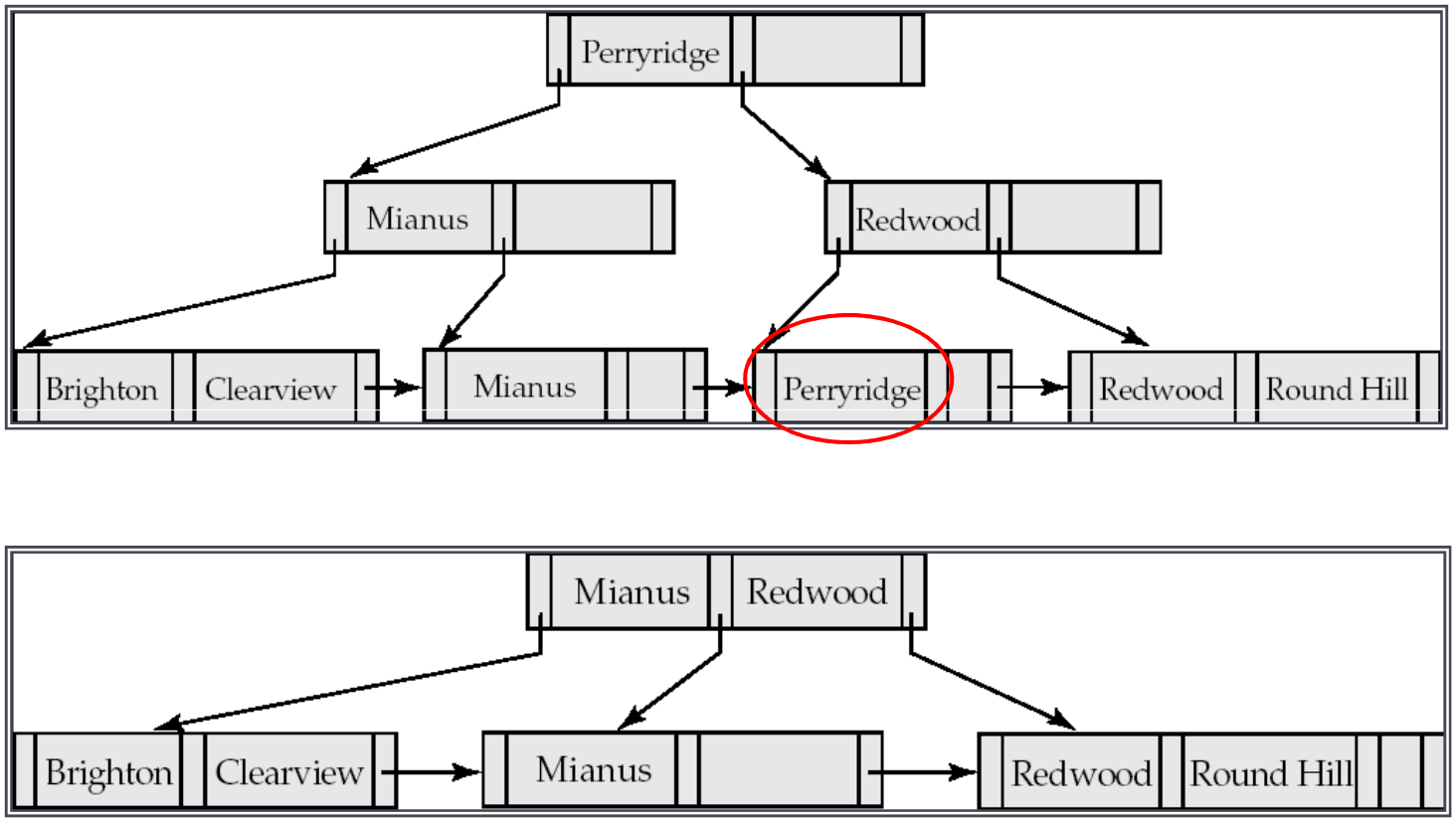
1. Șterge înregistrarea din fișier/tabel/relație
2. Dacă înregistrarea face parte dintr-un bucket, e ștearsă din acesta. Altfel (sau dacă bucketul devine gol) șterge din nodul frunză perechea (valoare cheie,pointer)
3. Dacă nodul va avea prea puține intrări în urma ștergerii și ele încap într-un nod vecin va avea loc unirea:
    1. Se inserează toate intrările în nodul stâng și se șterge celălalt
    2. Se șterge perechea (K_{i-1}, P_i), unde P_i este pointerul către nodul șters de la părinte. Dacă e necesar se propagă ștergerea recursiv în sus. Dacă nodul rădăcină rămâne cu un singur pointer va fi șters.
4. Altfel, dacă nodurile nu încap în vecin se redistribuie pointerii:
    1. Se redistribuie astfel încât avem satisfăcută condiția de minim în ambele noduri
    2. Se actualizează valoarea cheii de căutare corespunzătoare în părinte

Actualizări în B$^+$-arbori
Ştergerea: Exemplu (1)

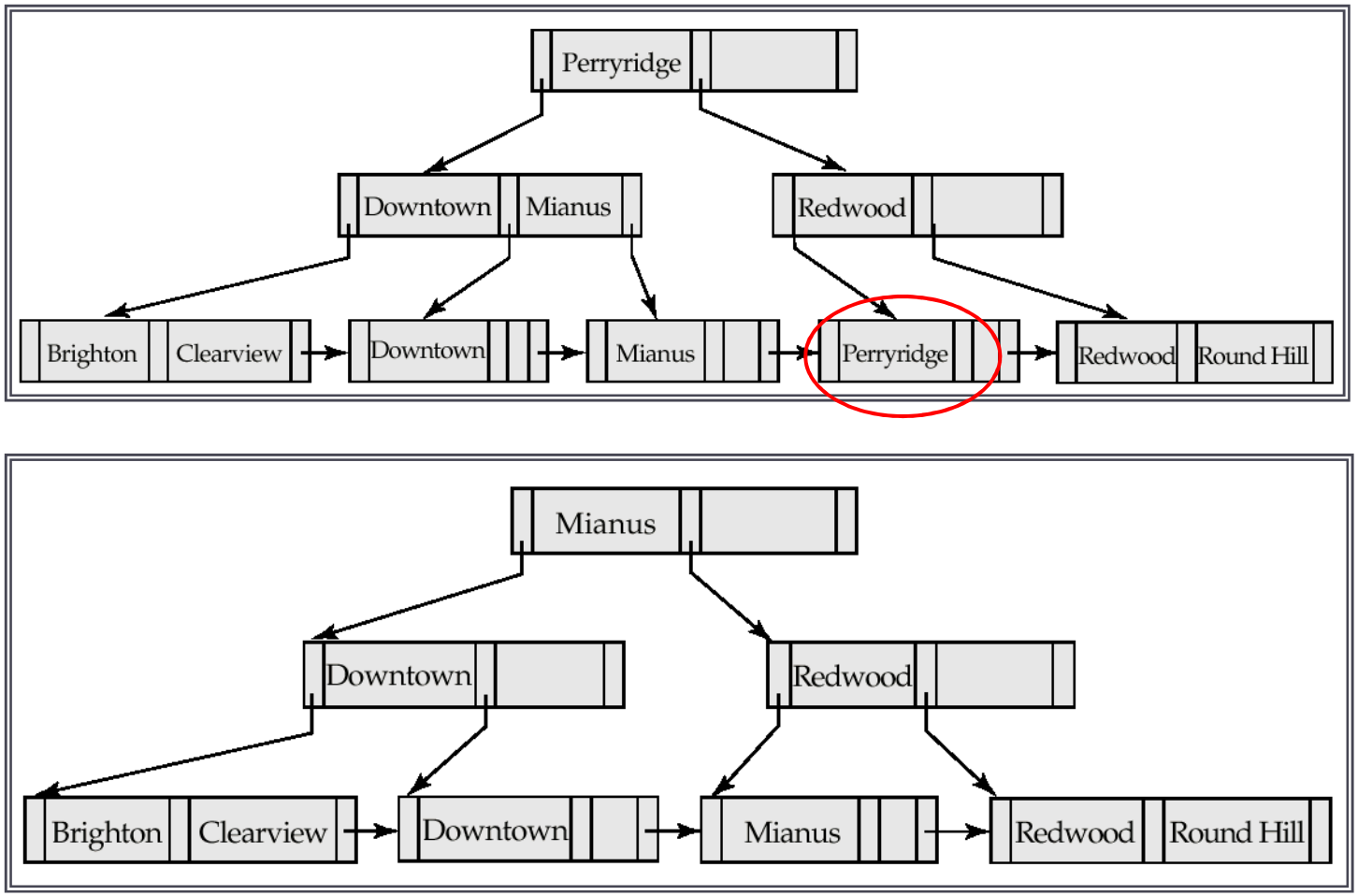




Actualizări în B$^+$-arbori
Ştergerea: Exemplu (2)

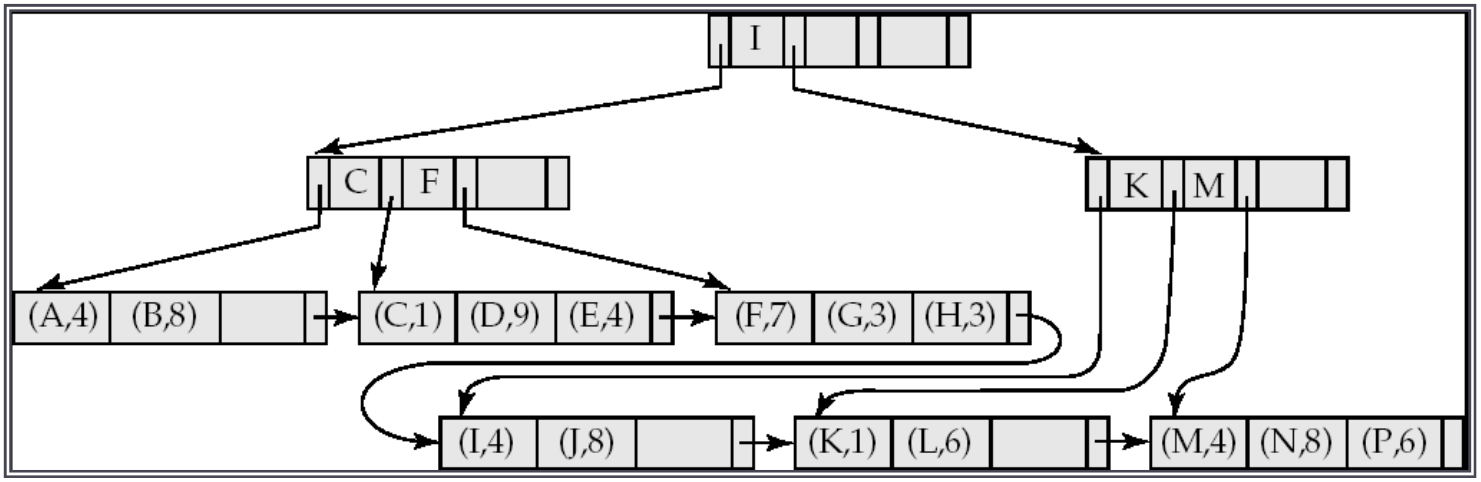


Actualizări în B$^+$-arbori
Ştergerea: Exemplu (3)



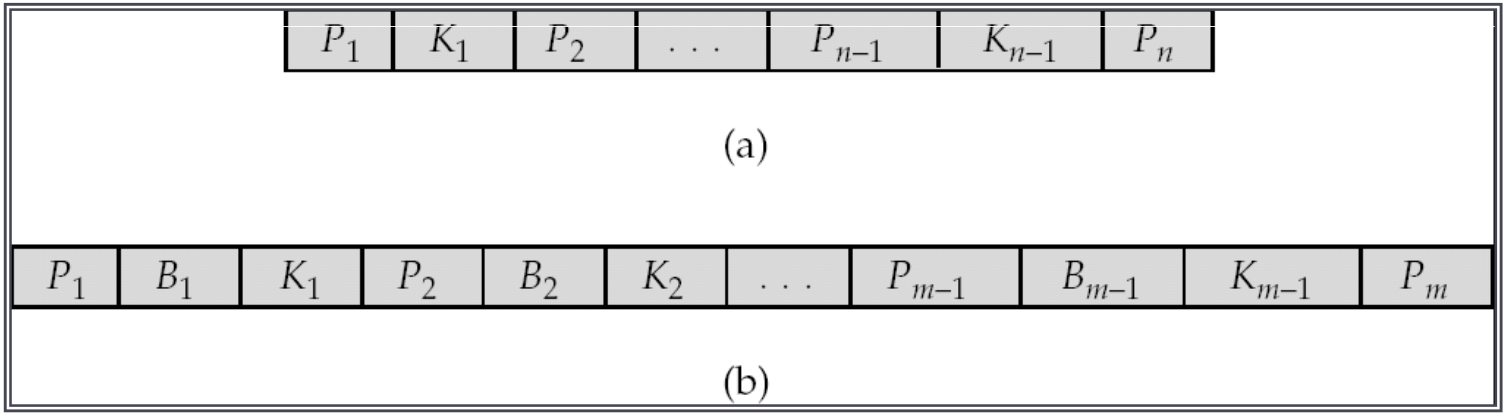
Organizarea fişierelor B+-arbore

- B+-arborii pot fi utilizaţi direct pentru organizarea fişierului şi nu doar pentru indexare
  - Nodurile frunză stochează înregistrări şi nu pointeri
  - Pentru a îmbunătăţi utilizarea spaţiului sunt implicaţi mai mulţi vecini în redistribuire pentru a evita divizarea sau unirea (utilizând doi vecini la redistribuire rezultă noduri având cel puţin $\lfloor 2n/3 \rfloor$ intrări)



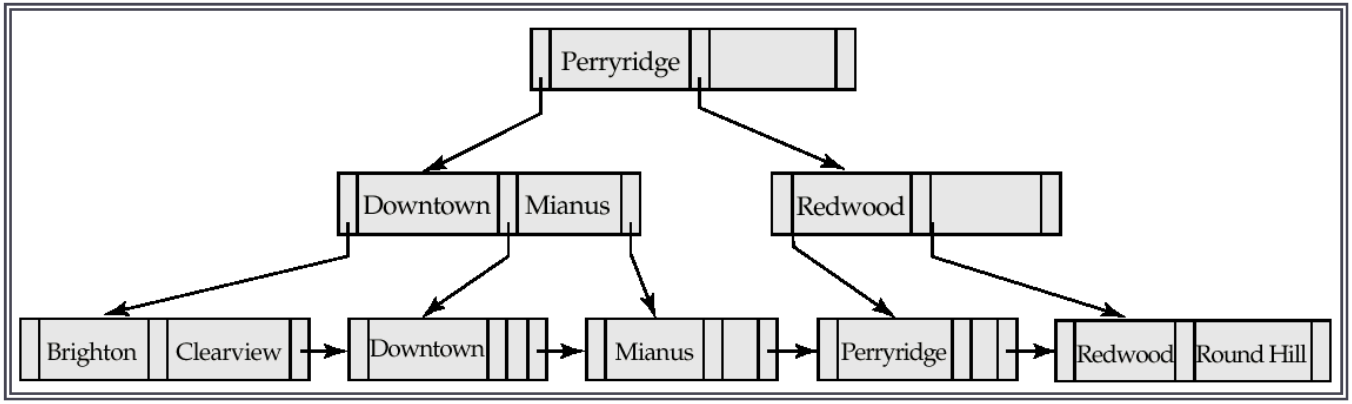
Indecşi B-arbore

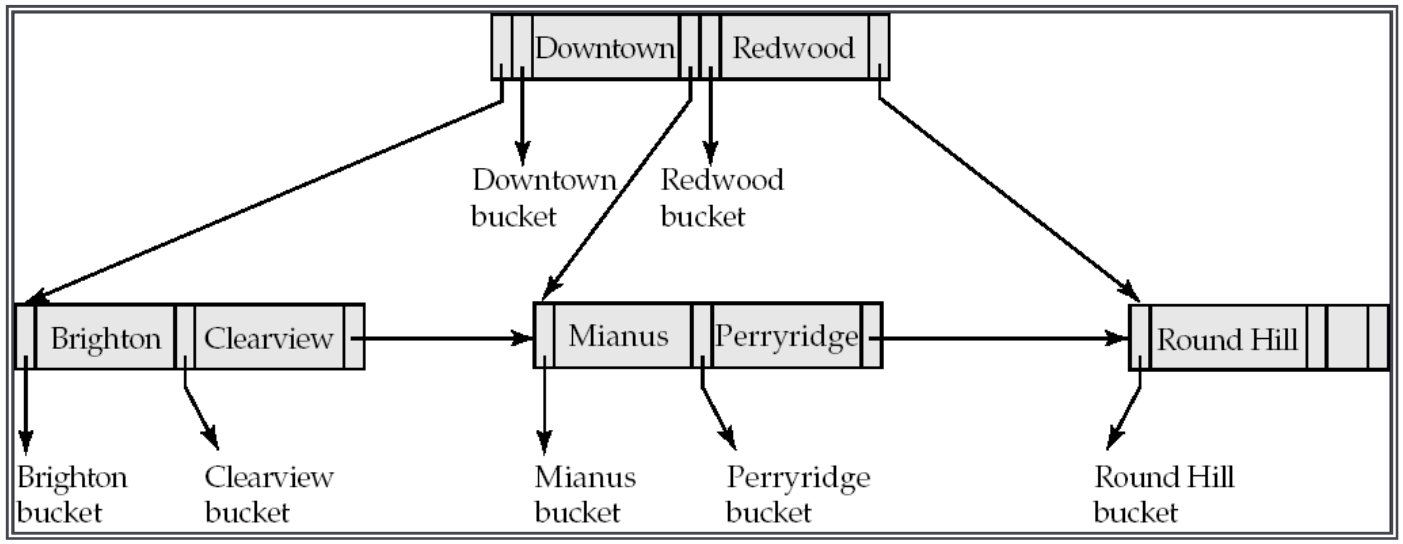
- Asemănători B^+-arborilor însă permit o singură apariție a valorilor cheilor de căutare
- Cheile de căutare în nodurile care nu sunt frunză nu mai apar nicăieri în arbore ceea ce necesită introducerea unui pointer adițional



- Pointerii B_i sunt pointeri către înregistrări sau bucketuri

Indecşi B-arbore
Exemplu





Indecşi B-arbore
Observații

- Avantaje
  - Pot utiliza mai puține noduri decât B^+-arborele corespunzător
  - E posibil a se localiza valoarea căutată înainte de a ajunge la frunze
- Dezavantaje
  - Nodurile care nu sunt frunze sunt mai mari ceea ce necesită reducerea numărului de valori stocate; înălțimea va fi mai mare
  - Inserările și ștergerile sunt mai complicate
  - Implementarea e mai dificilă
  - Nu e posibil a fi scanat un tabel doar cu ajutorul frunzelor
- Avantajele nu cântăresc mai mult decât dezavantajele, B^+-arborii fiind preferați de către SGBD-uri

Acces multi-cheie

- Pot fi utilizați mai mulți indecși la o interogare

  select *account_number*
  from *account*
  where *branch_name* = "Perryridge" **and** *balance* = 1000

- Strategii posibile pentru utilizarea indecșilor uni-atribut:
  - Utilizarea indexului cu cheia de căutare *branch_name*
  - Utilizarea indexului cu cheia de căutare *balance*
  - Utilizarea ambilor și efectuarea intersecției
- Dezavantaje:
  - Pot exista multe înregistrări ce satisfac numai una dintre condiții

Indecşi multi-cheie

- Cheile de căutare compuse sunt chei ce conțin mai mult de un atribut
- Ordinea lexicografică: $(a_1, a_2) < (b_1, b_2)$ dacă
  - $a_1 < b_1$ sau
  - $a_1 = b_1$ și $a_2 < b_2$

Ex. (*branch_name, balance*)

Pot fi rezolvate eficient condițiile de mai jos?

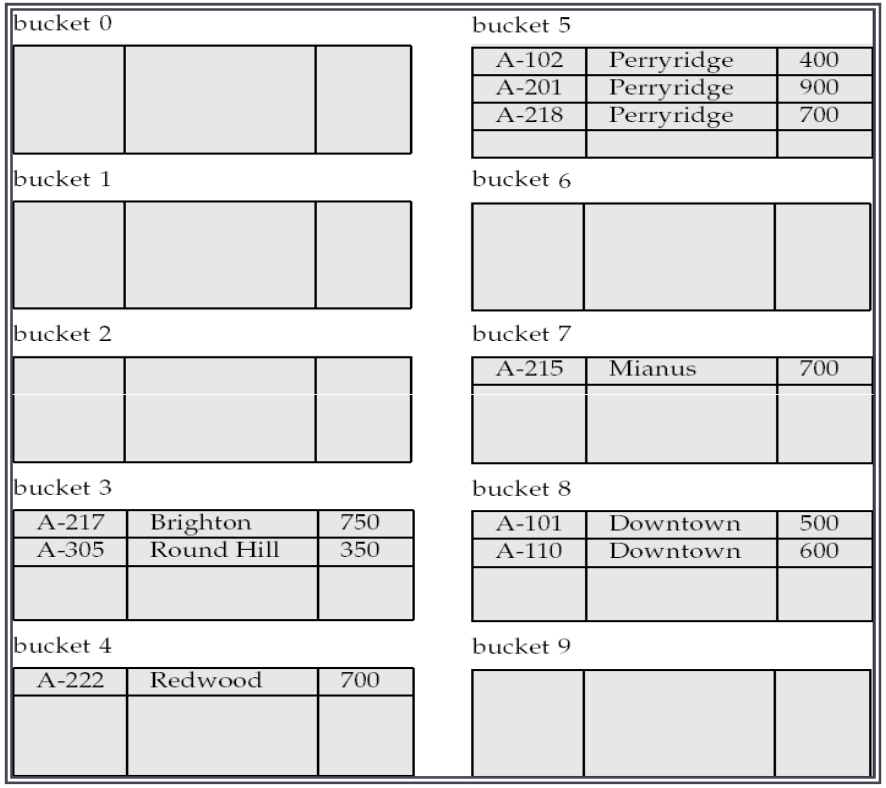
a) **where** *branch_name* = "Perryridge" **and** *balance* < 1000

b) **where** *branch_name* < "Perryridge" **and** *balance* = 1000

Hashing

- În organizarea de tip hash a fișierului/tabelului/relației înregistrările sunt grupate în bucketuri care pot fi localizate pe baza valorilor cheii de căutare
- Funcția hash *h:K->B* este o funcție de la mulțimea valorilor cheii de căutare la mulțimea adreselor tuturor bucketurilor
  - Localizează înregistrările pentru acces, inserare, ștergere
- Înregistrări cu valori diferite a cheii de căutare pot fi mapate la același bucket
  - Căutare secvențială în bucket

Organizarea de tip hash Exemplu

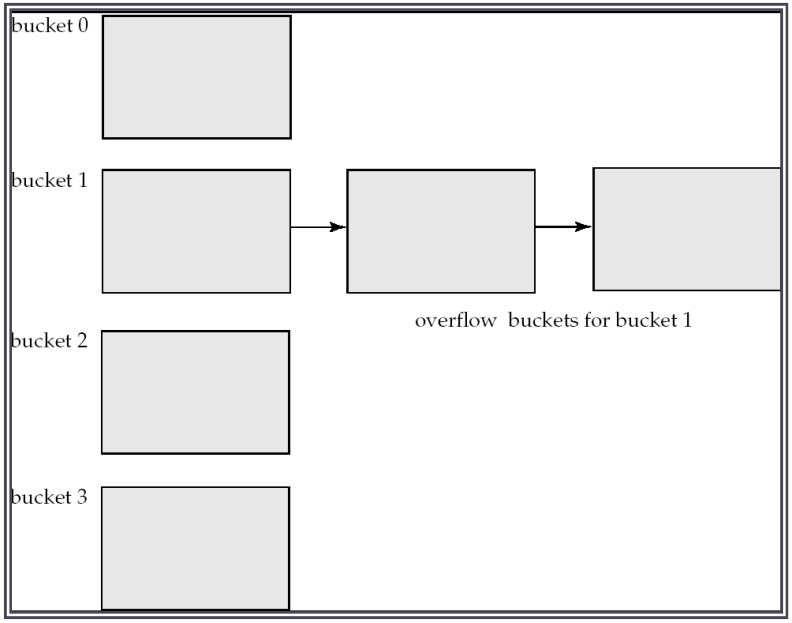


Organizarea de tip hash utilizând *branch_name* drept cheie:

Reprezentarea binară a caracterului i din alfabet este considerat a fi întregul i. Funcția hash: suma reprezentărilor binare modulo 10

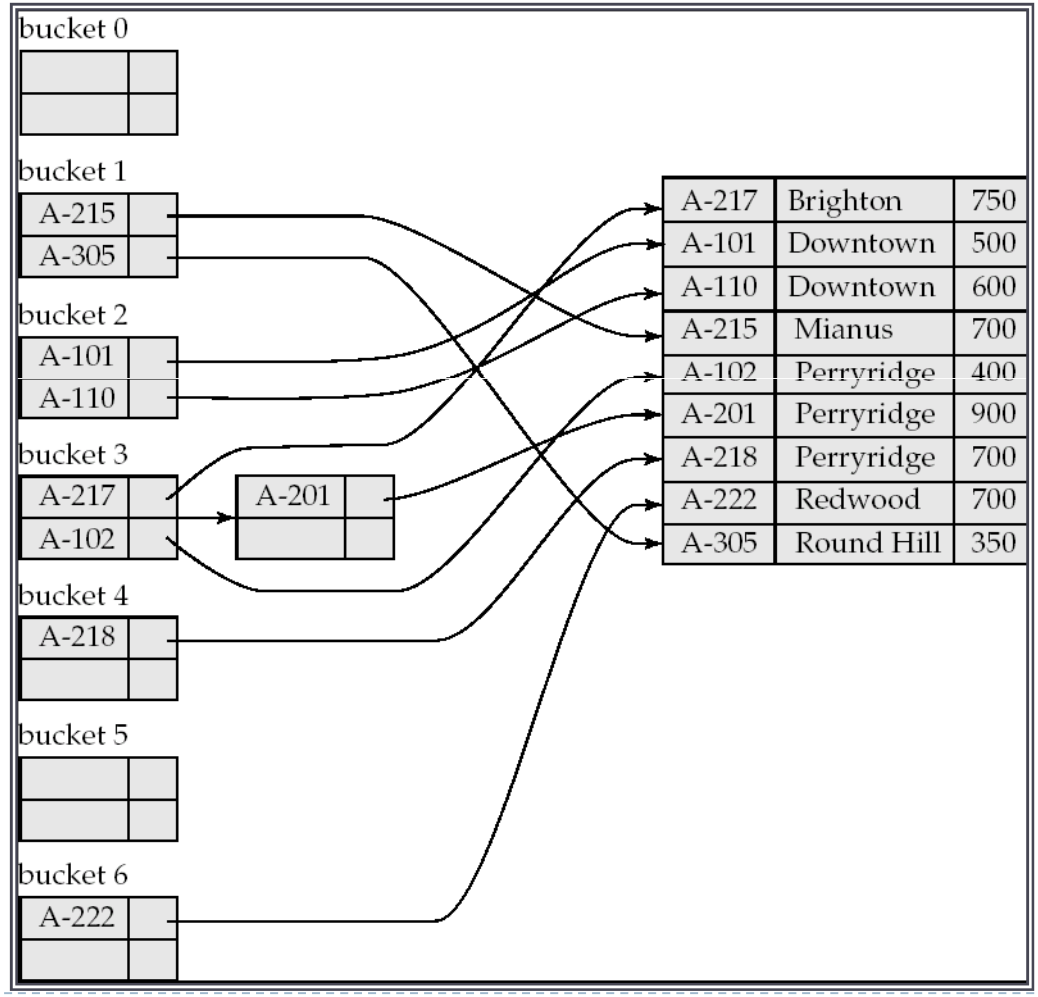
Funcții hash

- Cerințe
  - Uniformitate
  - Caracter aleatoriu
- Funcțiile hash tipice au la bază calcule pe reprezentarea binară internă a cheii de căutare
- Pot apărea situații de depășire a bucketului caz în care se utilizează bucketuri de exces



Indecşi hash

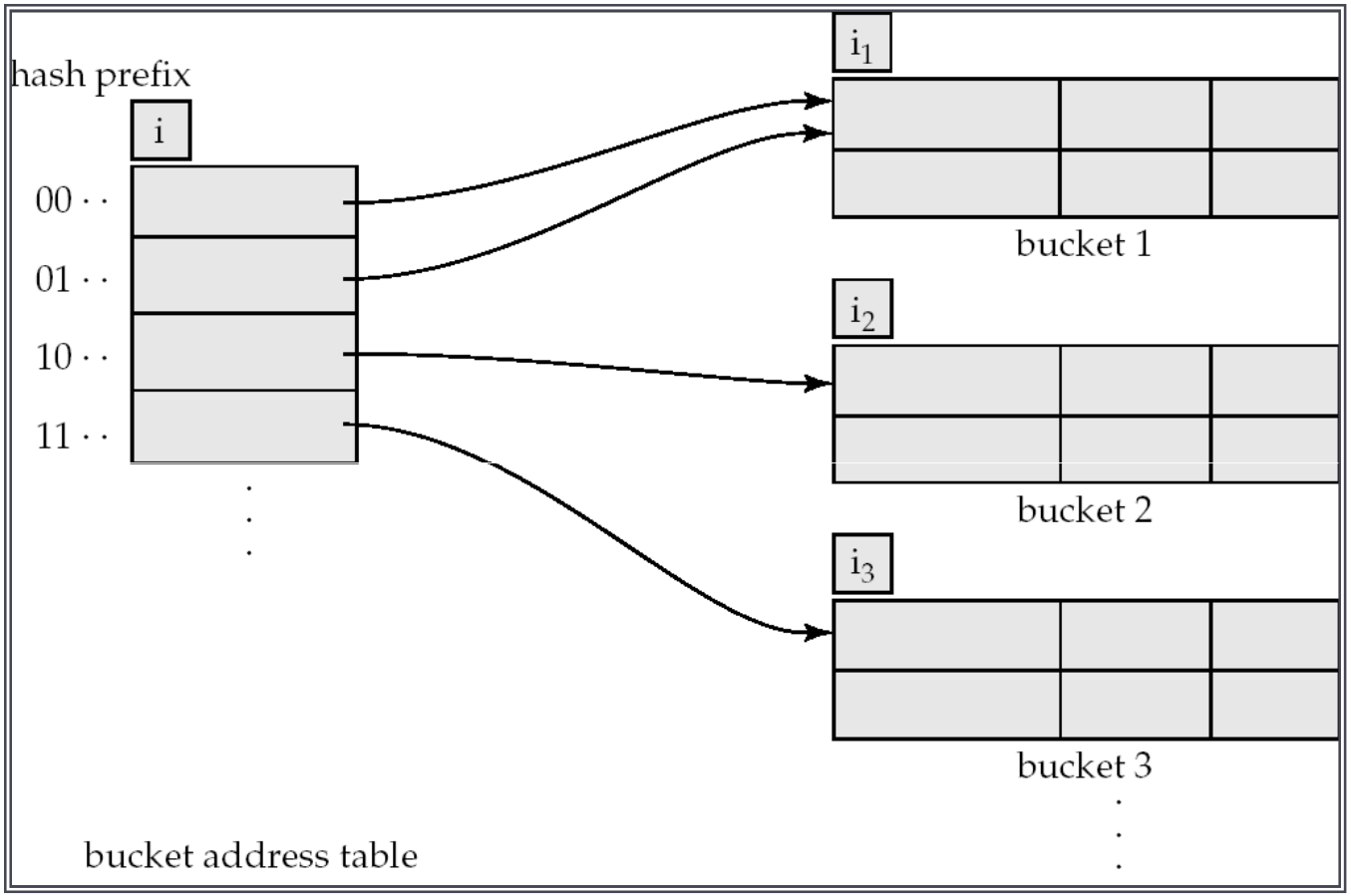
- Organizează cheile de căutare cu pointerii asociați într-o structură de tip hash



Hash dinamic

- Funcția h mapează valori a cheii de căutare la un set fix de adrese de bucketuri.
  - Dacă fișierul crește apar depășiri ale bucketurilor
  - Daca fișierul se micșorează spațiu este alocat inutil
- Soluții:
  - Reorganizări periodice cu o nouă funcție hash (costisitoare, necesită întreruperea operațiunilor)
  - Numărul de bucketuri este modificat dinamic
- Hash extensibil: funcția hash e modificată dinamic
  - Generează valori într-o mulțime mare, tipic întregi pe 32 biți
  - La un anumit moment se utilizează doar un prefix al funcției hash (doar primii i biți) a cărui lungime scade sau crește după caz

Hash extensibil
Structura generală



$i=2, i_2 = i_3 = i, i_1 = i - 1$

Hash extensibil
Utilizare

- Fiecare bucket j stochează o valoare i_j
  - toate intrările care indică spre bucketul j vor avea același valoare pe primii i_j biți

- Pentru a localiză bucketul ce conține cheia de căutare K_j:
  - Se calculează $h(K_j) = X$
  - Se utilizează primii i biți ai lui X și se urmează pointerul către bucketul potrivit

- Pentru a insera o înregistrare cu cheia de căutare K_j:
  - Se localizează bucketul j ca mai sus
  - Daca este spațiu în bucket se inserează înregistrarea
  - Altfel bucketul este divizat și inserarea este reîncercată

Hash extensibil
Divizare bucket la inserare

Pentru a diviza bucketul j la inserarea unei valori K_j:

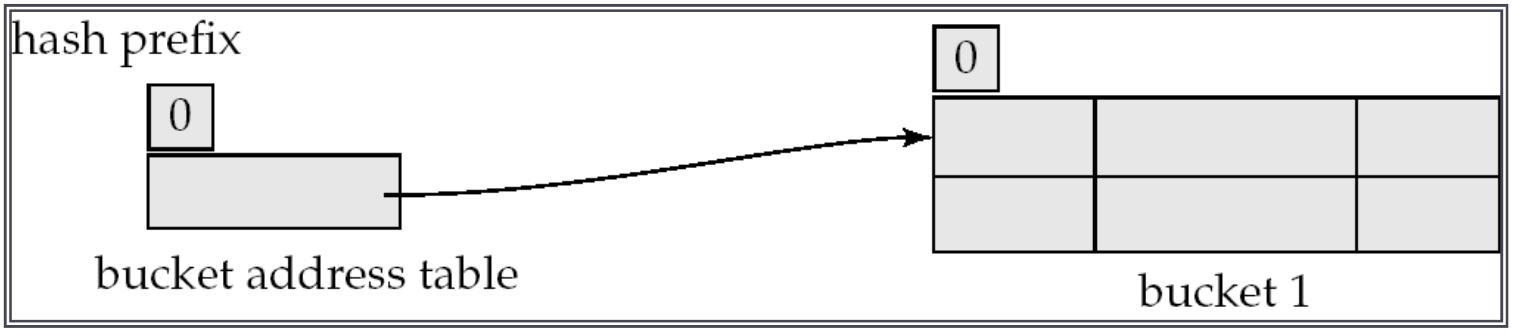
- Dacă $i > i_j$
  1. Se alocă un nou bucket z și $i_j = i_z = (i_j + 1)$
  2. Se actualizează a doua jumătate a tabelei de adrese a bucketurilor pentru a indica spre z
  3. Se scot înregistrările din j și sunt reinserate în j sau z
  4. Se recalculează adresa bucketului pentru K_j și se inserează
- Dacă $i = i_j$
  1. Dacă se atinge o limită a lui i se utilizează bucketuri de exces
  2. Altfel
     1. Se incrementează i și se dublează dimensiunea tabelei de adrese
     2. Se înlocuiește fiecare intrare în tabel cu două intrări care indică spre același bucket
     3. Se recalculează adresa bucketului pentru K_j și se inserează (acum $i > i_j$)

Hash extensibil
Ştergere

- Pentru a șterge o înregistrare
  - Se localizează bucketul și se șterge din el
  - Dacă bucketul devine gol acesta este șters cu modificările necesare în tabela de adrese
  - Pot fi contopite bucketuri care au aceeași valoare pentru i_j și același prefix $i_j - 1$
  - Descreșterea dimensiunii tabelei de adrese este posibilă

Hash extensibil
Exemplu

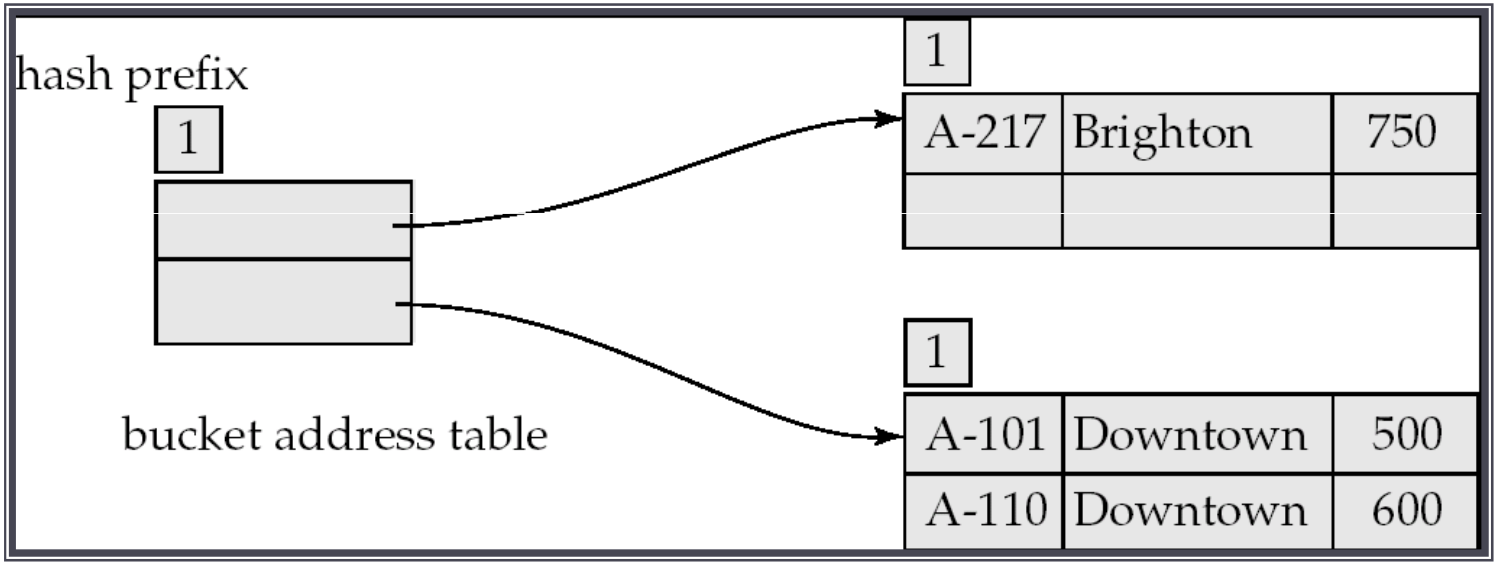
branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111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



structura hash inițială, dimensiune bucket = 2

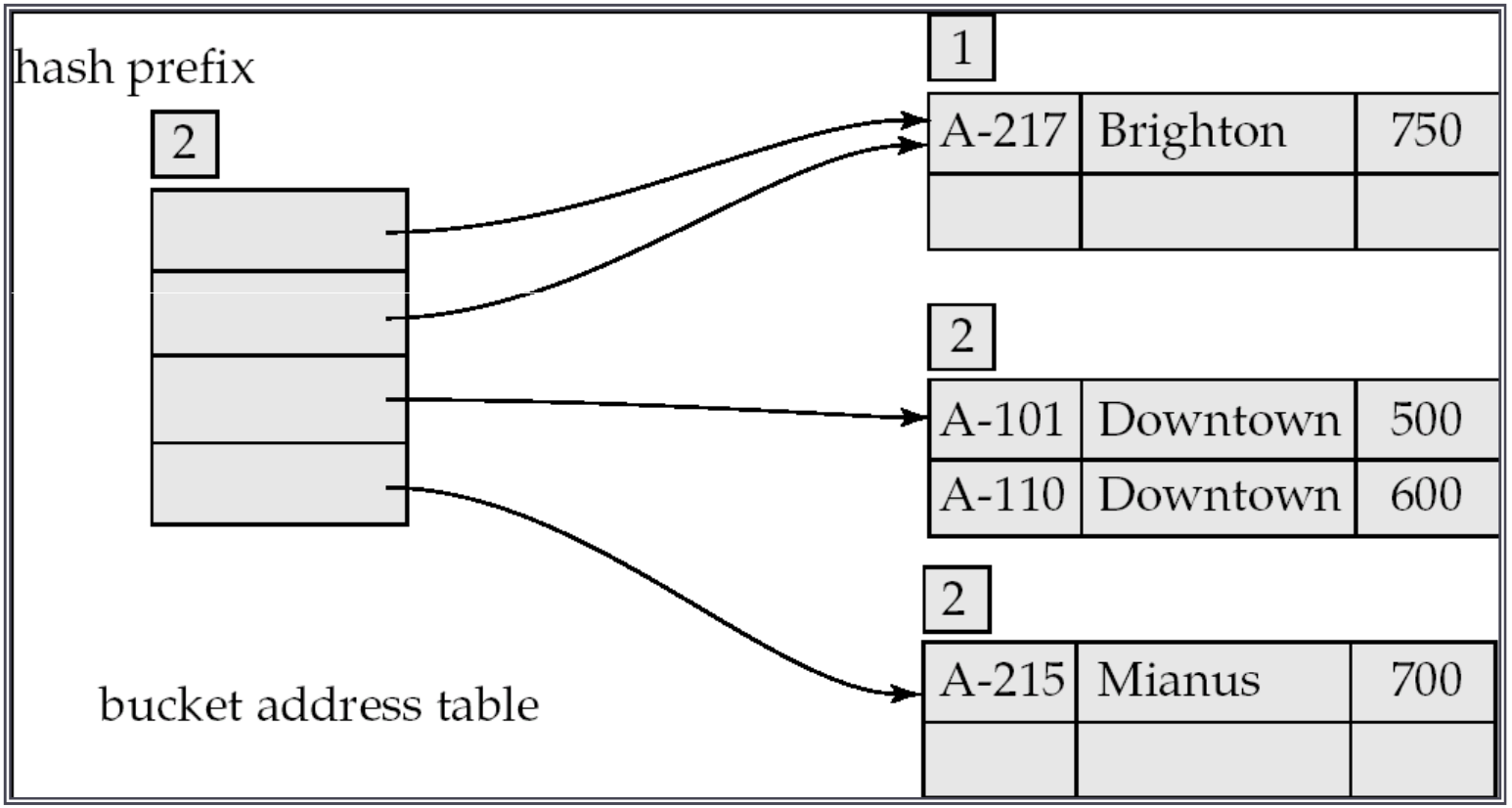
Hash extensibil
Exemplu

- Structura după inserarea unei înregistrări Brighton și a două înregistrări Downtown



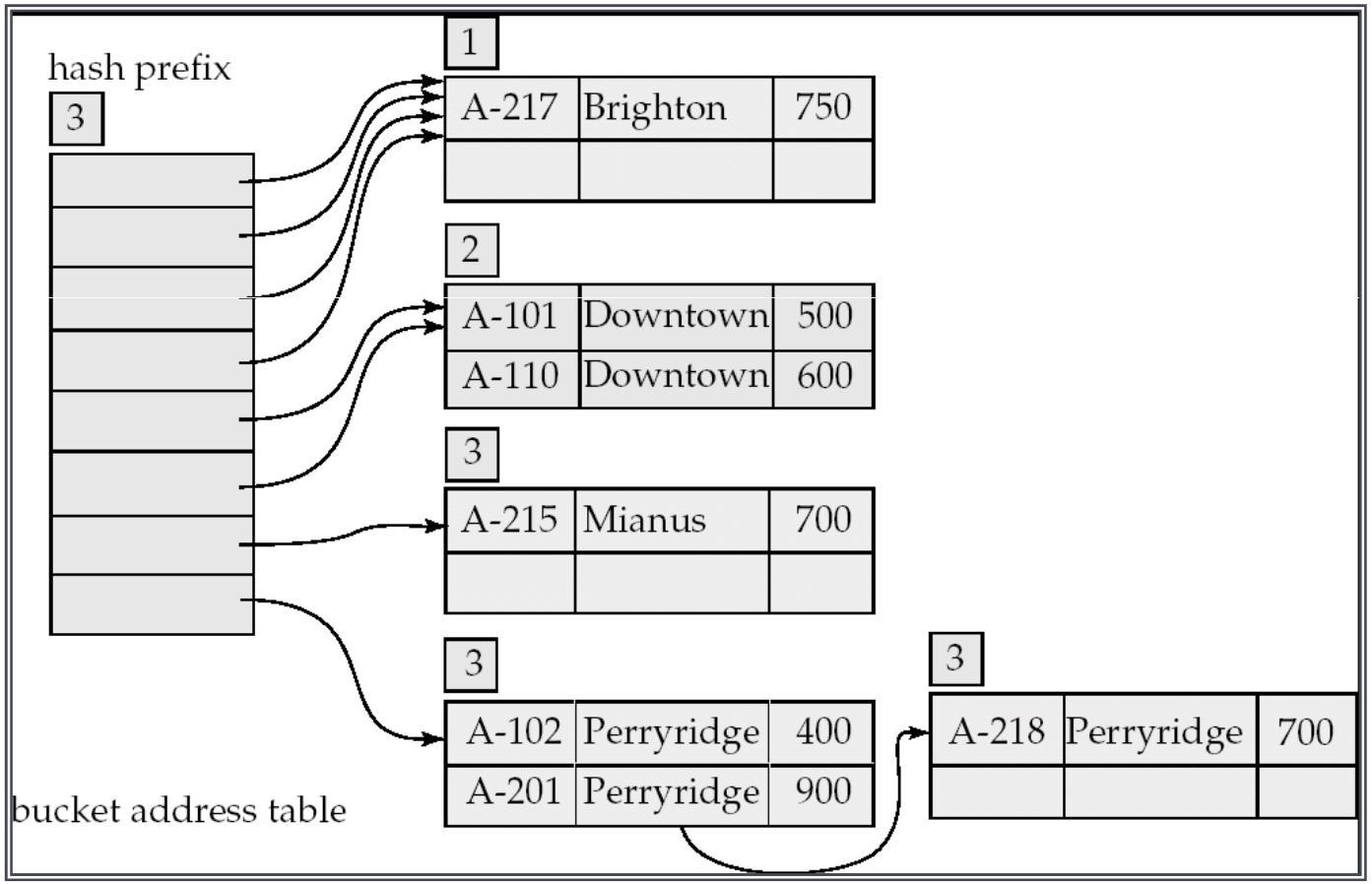
Hash extensibil
Exemplu

- După inserarea înregistrării Mianus



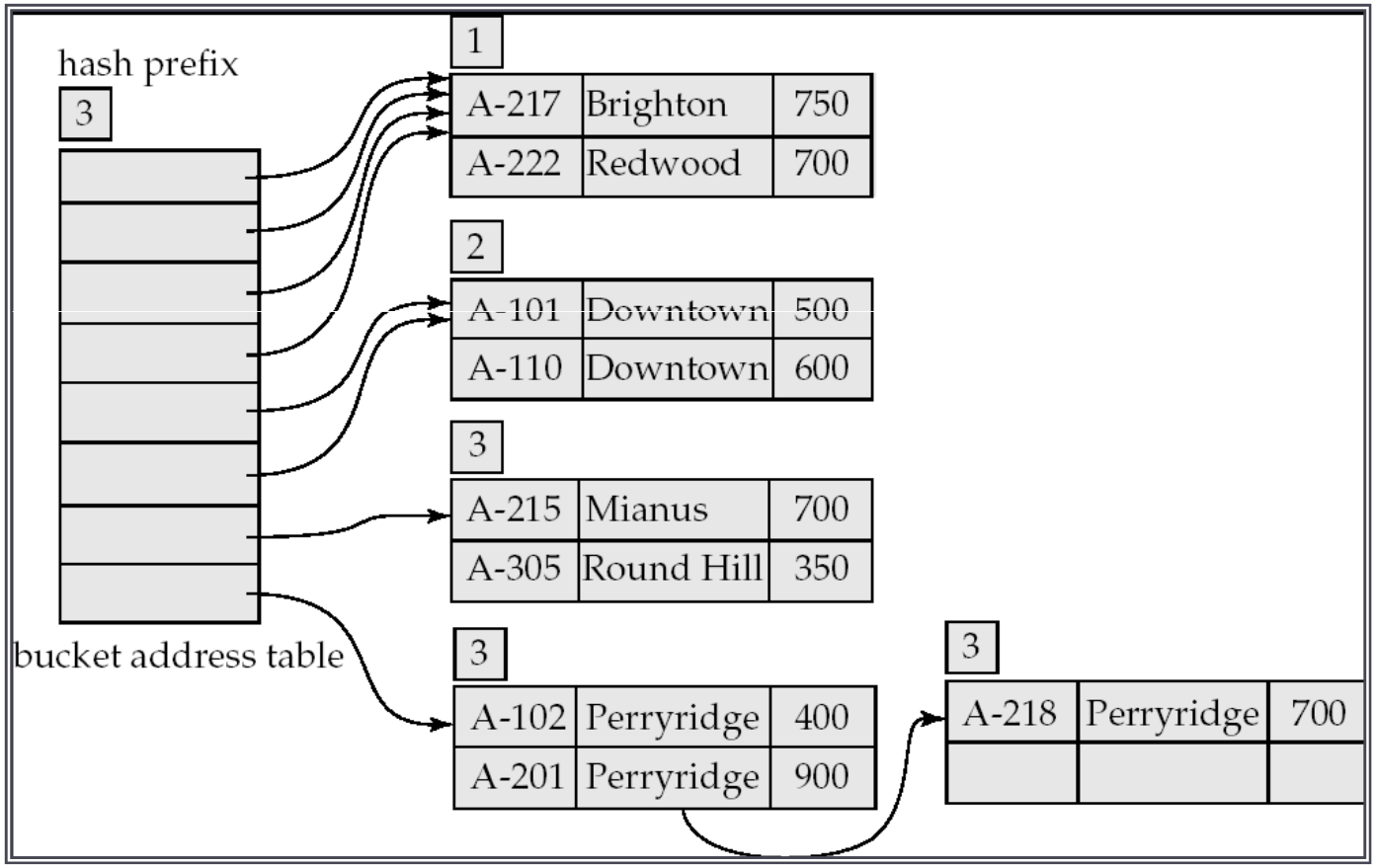
Hash extensibil
Exemplu

- După inserarea a trei înregistrări Perryridge



Hash extensibil
Exemplu

- După inserarea înregistrărilor Redwood și Round Hill



Hashing extensibil
Observații

- Beneficii
  - Performanța nu se degradează cu creșterea fișierului
  - Minimizează consumul de memorie
- Dezavantaje
  - Tabela de adrese a bucketurilor poate deveni foarte mare
    - Soluție: utilizarea unui B^+-arbore pentru a localiza înregistrarea dorită în tabela de adrese
  - Modificarea dimensiunii tabelei de adrese este costisitoare
- În funcție de tipul interogării:
  - Hashingul e indicat când se specifică o valoare a cheii de căutare
  - Dacă se lucrează cu intervale de valori e mai rapid indexul ordonat
- În practică:
  - Postgres suportă indecșii hash
  - Oracle suportă organizarea statică de tip hash, nu și indecși hash
  - SQLServer suportă numai B^+-arbori

Indecşi bitmap

- Proiectați pentru a trata eficient interogările cu mai multe chei de căutare
- Aplicabili pentru atribute care iau un set redus de valori distincte
- Uplele relației sunt considerate a fi numerotate
- Structura:
- Un șir de biți pentru fiecare valoare a atributului
  - Șirul are lungimea numărului de înregistrări
  - Valoarea 1 semnifică egalitate cu valoarea căreia îi este asociat bitmapul

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

Indecşi bitmap
Observații

- Interogările sunt rezolvate utilizând operatori pe biți:
  - Intersecția – AND
  - Reuniunea – OR
  - Complementarierea – NOT

  where *gender ='m'* ***and*** *income_level ='L1'*

  (10010 AND 10100) = 10000
  - Nu e necesar accesul fișierului
  - Utili când interogarea necesită numărare
- Implementare eficientă:
  - La ștergere se preferă utilizarea unui bitmap de existență
  - Bitmapurile sunt împachetate în cuvinte (tipul word) de 32 sau 64 biți (operatorul and pe un cuvânt – o singură instrucțiune CPU)
  - Bitmapuri pot fi utilizate pe nivelul frunză în B^+-arbori pentru valori ale cheii de căutare ce corespund unui număr mare de înregistrări

Definirea indecşilor în standardul SQL

- Creare:

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

E.g.: **create index** *b-index* **on** *branch(branch_name)*

- Ştergere:

drop index <index-name>

- Majoritatea SGBD-urilor permit specificarea tipului de index

Indexarea în Oracle

- Oracle suportă B^+-arbori implicit la crearea indexului cu comanda SQL
- Un nou atribut nenul row-id este adăugat tuturor indecșilor pentru a garanta că toate valorile cheii de căutare sunt unice
- Indecșii sunt suportați pe:
  - Atribute și liste de atribute
  - Rezultatul unei funcții peste atribute
- Indecși bitmap sunt suportați cu declararea

  create bitmap index <index-name> **on** <relation-name> (<attribute-list>)
- Indecși hash nu sunt suportați dar există suport pentru organizarea hash statică

Bibliografie

- Capitolul 11 în *Avi Silberschatz Henry F. Korth S. Sudarshan.* "*Database System Concepts*". McGraw-Hill Science/Engineering/Math; 6 edition (January 27, 2010)