



```
CREATE BITMAP INDEX idx ON  
BAZE_DE_DATE (topic);
```

```
SELECT * FROM BAZE_DE_DATE WHERE topic = 'INDECȘI';
```

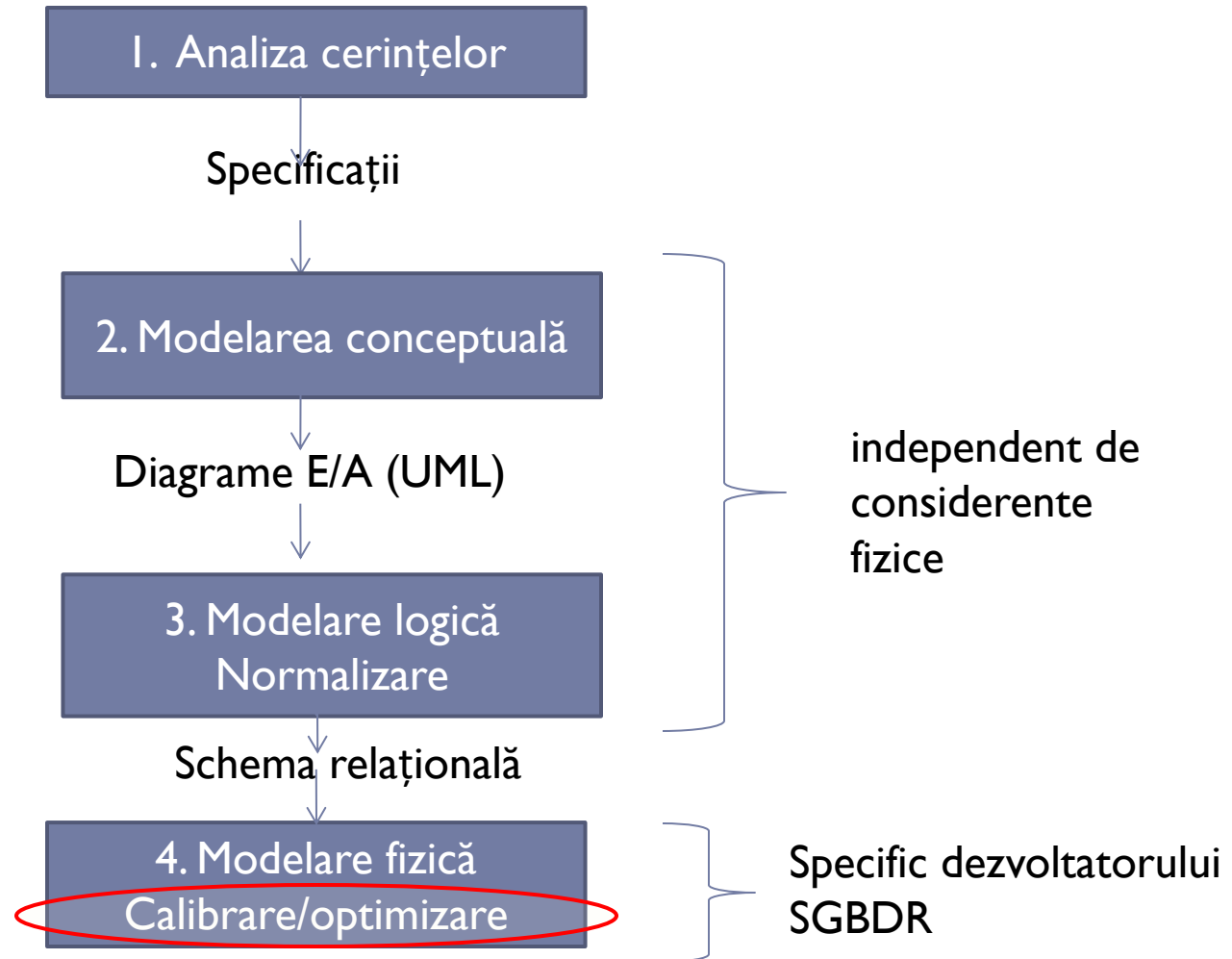
Mihaela Elena Breabăn

© FII 2024-2025

# Proiectarea Bazelor de date Relaționale

## Metodologie

---



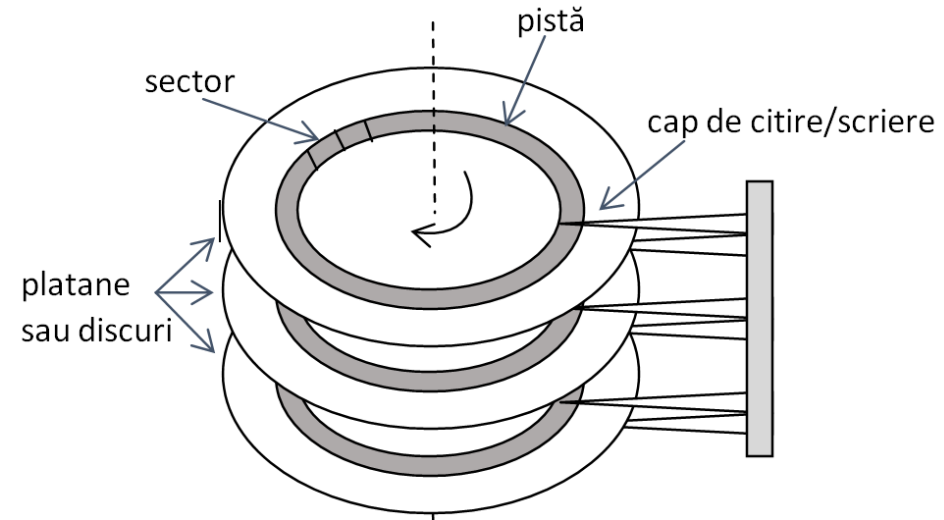
# Cuprins

---

- ▶ Stocare fizică și acces
- ▶ Indexare – motivație și concepte de bază
- ▶ Structuri ordonate
  - ▶ Indecși secvențiali
  - ▶ B<sup>+</sup>-arbori
  - ▶ Indecși multi-cheie
- ▶ Hashing
  - ▶ Hashing static
  - ▶ Hashing dinamic
- ▶ Acces multi-cheie și Indecși bitmap
- ▶ Suportul SQL pentru indexare
- ▶ Indexarea în Oracle

# Stocarea datelor și acces

ID	prenume	nota
20	Ioana	9.5
40	Andrei	8.66
10	Tudor	8.55
30	Maria	8.33
70	Alex	9.33



- ▶ Timpul necesar pentru a aduce un **bloc** de date în memorie este determinat de:
  - ▶ *Timpul de localizare* (timpul necesar poziționării capului de citire pe pistă)
  - ▶ *Latența rotațională* (timpul de rotație a pistei/discului sub capul de citire)
  - ▶ *Timpul de transfer* (timpul necesar transferului datelor către memoria de lucru)
- ▶ Pentru a optimiza timpul de acces, o bază de date relațională stochează datele în mod secvențial, înregistrare după înregistrare

# Indexare – Motivație (1)

- ▶ De obicei, SGBD-ul petrece majoritatea timpului rezolvând interogări (căutând)

```
SELECT * FROM Student  
WHERE ID=40;
```

Cheie de căutare

Cum găsim înregistrările dorite?

a) Ordonare aleatorie

ID	prenume	nota
20	Ioana	9.5
40	Andrei	8.66
10	Tudor	8.55
30	Maria	8.33
70	Alex	9.33

Cheie de sortare

b) Ordonare crescătoare după ID

ID	prenume	nota
10	Tudor	8.55
20	Ioana	9.5
30	Maria	8.33
40	Andrei	8.66
70	Alex	9.33

# Indexare – Motivație (2)

---

- ▶ Când datele sunt sortate după cheia de căutare devine posibilă căutarea binară
  - ▶ Complexitate timp:  $O(\log_2(N))$   
( $\log_2(100\ 000)=17$ )

SELECT \* FROM student

WHERE prenume='Ioana';

Cum putem rezolva interogarea de mai sus eficient (datele nu sunt sortate după prenume)?

Soluția: construim un fișier **index**

# Concepte de bază în indexare

---

- ▶ **Fișier de date** – secvența de blocuri ce conțin înregistrările unui tabel
- ▶ **Cheie de căutare** – un atribut (sau o mulțime de atribute) care constituie criteriu de selecție/căutare
- ▶ **Cheie de sortare** – un atribut care decide ordonarea înregistrărilor în fișierul cu date
- ▶ **Fișier index**– este asociat unei chei de căutare într-un fișier cu date și conține **înregistrări index** de forma

Valoare a cheii de căutare	pointer
----------------------------	---------

- ▶ **Index dens** – stochează câte o intrare pentru fiecare valoare existentă în fișierul cu date a cheii de căutare
- ▶ **Index rar** – nu stochează toate valorile cheii de căutare

## Observații:

- ▶ Un fișier cu date poate avea asociate mai multe fișiere index
- ▶ Fișierele index sunt de obicei de dimensiuni mai mici comparativ cu fișierul de date

# De reflectat asupra...

---

- ▶ CÂȚI indecși ar trebui să folosim în practică ?
- ▶ CÂND trebuie creați indecși și când nu trebuie creați indecși?
- ▶ CUM ar trebui să indexăm (ca structuri de date utilizate)?
  
- ▶ Considerente:
  - ▶ Spațiul de stocare necesar
  - ▶ Timpul de acces
  - ▶ Timpul de inserare
  - ▶ Timpul de ștergere
  - ▶ Tipul interogărilor adresate

# Tipuri de indecși

---

- ▶ **Indecși ordonați:** valorile cheii de căutare sunt ordonate
  - ▶ Indecși secvențiali
  - ▶ B<sup>+</sup>-arbori
- ▶ **Indecși hash:** valorile cheii de căutare sunt uniform distribuite în grupuri denumite *buckets* cu ajutorul unei funcții *hash*
  - ▶ **Bucket:** unitate de stocare ce poate conține una sau mai multe înregistrări
  - ▶ **Funcție hash** = funcție de *dispersie* – mapează date de dimensiune variabilă la o mulțime fixă de valori
- ▶ **Indecși bitmap:** asociați atributelor categoriale/discrete, codifică distribuția valorilor ca o matrice binară



## Indecși ordonați: fișiere secvențiale

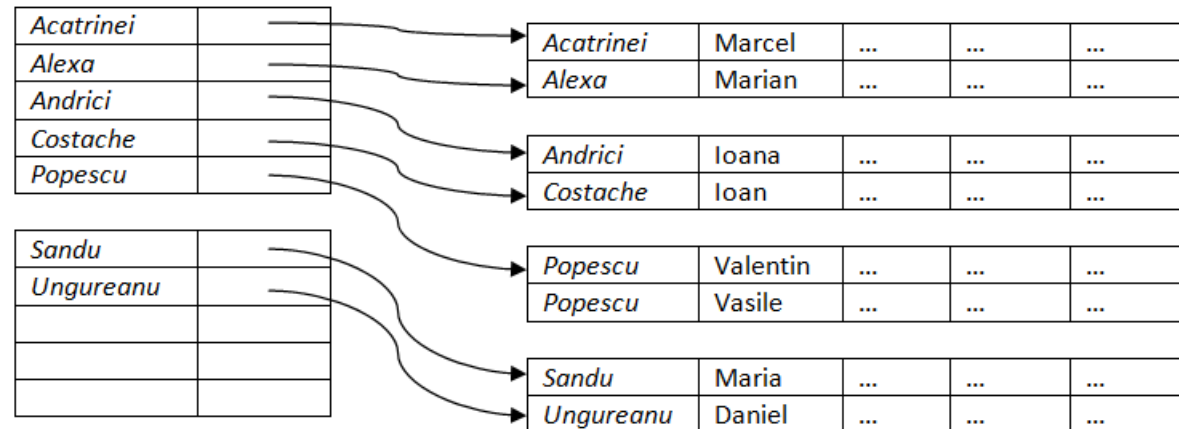
# Indecși secvențiali

---

- ▶ Intrările index sunt sortate pe baza cheii de căutare
  - ▶ Ex: catalogul cu autori dintr-o bibliotecă
- ▶ **Index primar**: cheia de căutare (cheia indexului) este și cheie de sortare a fișierului cu date
  - ▶ Cheia de căutare/sortare în acest caz constituie de obicei (nu obligatoriu) chiar cheia primară a tabelului
  - ▶ Un tabel poate avea cel mult un index primar. **DE CE?**
- ▶ **Index secundar**: cheia de căutare dă o altă ordonare a înregistrărilor decât cea din fișierul de date

# Indecși denși

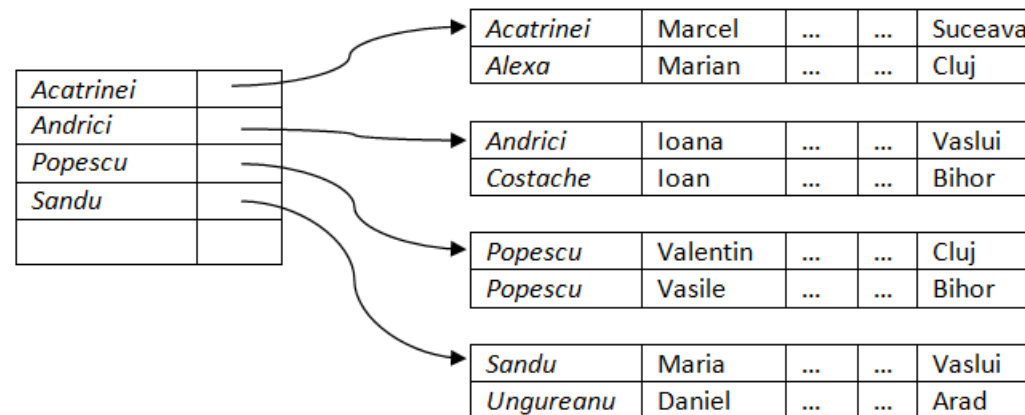
- ▶ **Index dens:** conține intrări pentru fiecare valoare a cheii de căutare existentă în fișierul cu date.
- ▶ Dacă indexul este primar, conține un singur pointer pentru toate înregistrările cu aceeași valoare a cheii de căutare (un pointer către prima înregistrare din serie). **DE CE?**
- ▶ Dacă indexul este secundar, mai multe intrări pot fi necesare pentru o aceeași valoare. **DE CE?**



Index dens primar – cheia de căutare este aceeași cu cheia de sortare a fișierului cu date

# Indecși rari

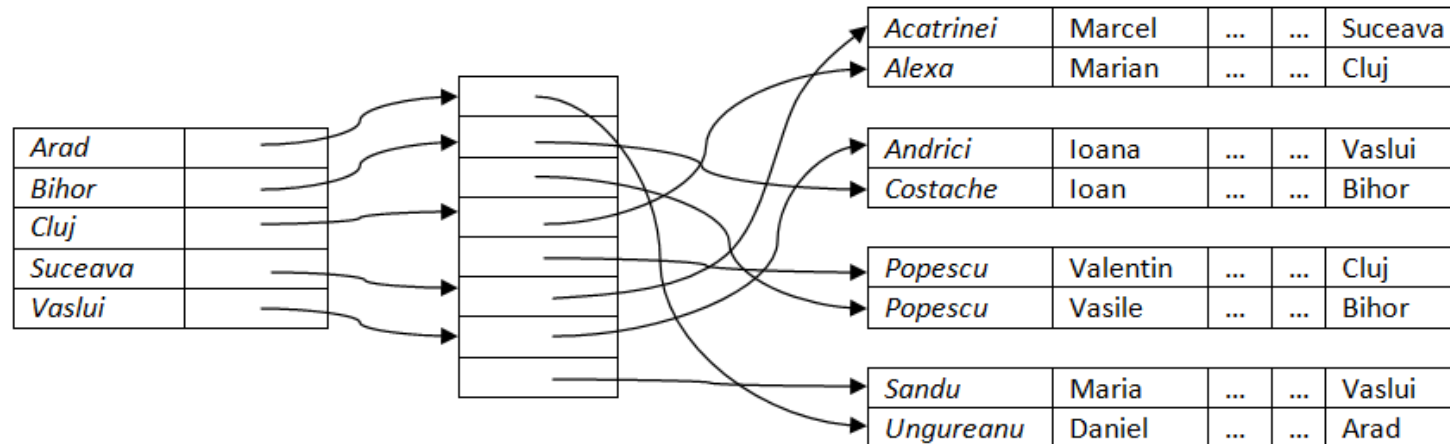
- ▶ **Indecșii rari NU** conțin ca intrări toate valorile cheii de căutare
  - ▶ Aplicabili doar pentru înregistrări care sunt ordonate pe baza cheii de căutare (când atributul cheie de căutare este și cheie de sortare a fișierului cu date)
  - ▶ De obicei o intrare în index trimite către un bloc din fișierul cu date
- ▶ Pentru a localiza o intrare cu valoarea  $k$  a cheii de căutare în fișierul cu date:
  - ▶ Găsim intrarea din index care corespunde la cea mai mare valoare mai mică decât  $k$
  - ▶ Ne uităm secvențial în fișierul cu date începând cu înregistrarea la care ne trimite indexul



Index rar: cheia de căutare este întotdeauna și cheie de sortare a fișierului cu date

# Index secundar

- ▶ *Interogare: Găsiți toți studenții care locuiesc în Cluj (fișierul cu date este sortat pe baza numelor studenților!)*
- ▶ Soluția: indexul secundar (dens!)
- ▶ Pentru a implementa eficient asocierea de tip unu-la-mulți dintre index și fișierul cu date, blocuri cu pointeri sunt utilizate



# Indecși multi-nivel

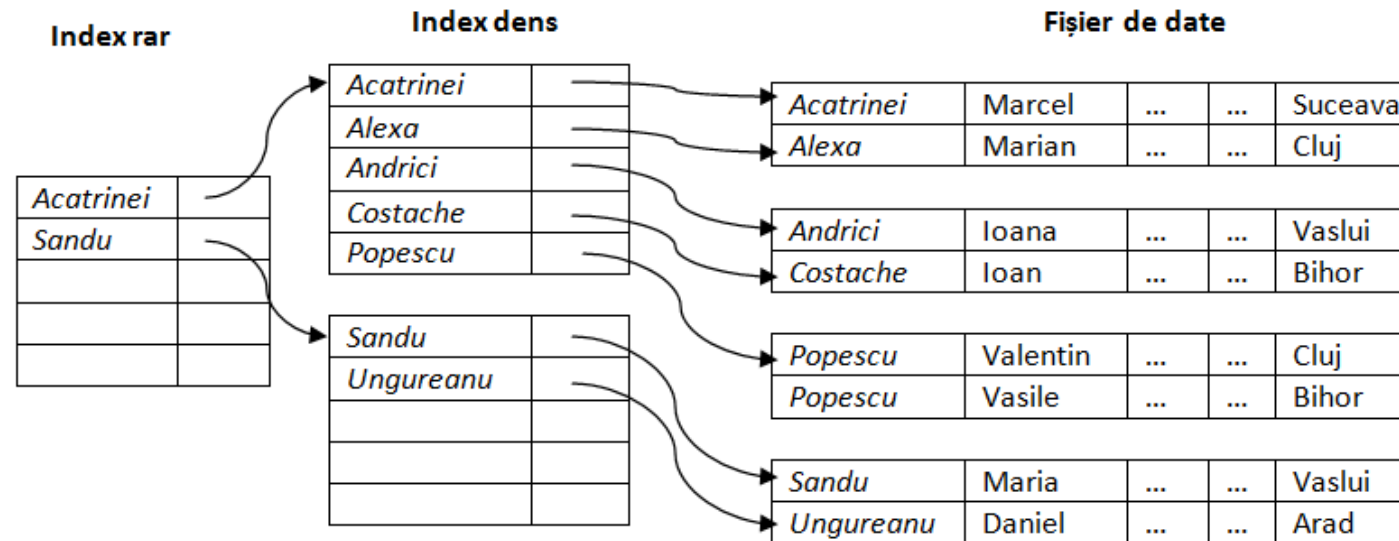
---

- ▶ **Index multi-nivel:** un index poate fi asociat altui index și nu direct fișierului de date; acesta este un index rar
  - ▶ Necesă cînd fișierul index asociat fișierului cu date are dimensiuni considerabile sau nu încapă în memoria de lucru
  - ▶ **Index intern** – indexul construit peste fișierul de date
  - ▶ **Index extern** – indexul rar construit peste indexul intern
- ▶ Cînd indexul extern este prea mare un alt index rar poate fi construit peste acesta, etc...
- ▶ Indecșii de pe toate nivelurile trebuie actualizați cînd fișierul cu date suferă modificări prin operații DML

# Indecși multi-nivel

## Exemplu

---



# Actualizarea indecșilor secvențiali

## Ștergeri în date

---

1. Se găsește înregistrarea ce trebuie ștearsă – se poate apela la index;
2. Se șterge înregistrarea din fișierul cu date;
3. Se actualizează indecșii asociați tabelului:
  1. Dacă mai există și alte înregistrări cu aceeași valoare a cheii de căutare, se șterge doar pointerul
  2. Dacă înregistrarea ștearsă este singura cu valoarea cheii  $k$ , aceasta trebuie ștearsă din index
    - ▶ Ștergerea din indexul dens: e similară ștergerii dintr-un fișier de date
    - ▶ Ștergerea dintr-un index rar:
      - Dacă există intrarea  $k$  în index, aceasta este înlocuită de următoarea valoare a cheii de căutare (din ordonarea valorilor cheii de căutare existente în fișierul cu date)
      - Dacă următoare valoare există deja în index, intrarea corespunzătoare lui  $k$  este ștearsă.

# Actualizarea indecșilor secvențiali

## Inserări

---

1. Se inserează tuplul în fișierul cu date
  2. Se caută în index valoarea cheii de căutare corespunzătoare noului tuplu și se actualizează indexul astfel:
    - ▶ Indexul dens: dacă valoarea nu apare în index, va fi inserată; altfel, dacă indexul este secundar se adaugă doar pointerul
    - ▶ Indexul rar: dacă indexul menține o intrare pentru fiecare bloc a fișierului cu date, doar când un nou bloc este creat în fișierul cu date, o nouă intrare va fi adăugată în index, trimitând la prima înregistrare a blocului.
- 
- ▶ Inserările în fișierul cu date și în fișierul index poate necesita crearea unor blocuri de exces => structura secvențială degenerază
  - ▶ Inserarea și ștergerea în indecșii multi-nivel sunt extensii simple ale cazurilor discutate

Indecși ordonați:  
B<sup>+</sup>-arbori

# Indecși bazați pe structuri B<sup>+</sup>-arbori

## Motivație

---

- ▶ Structurile secvențiale ordonate se degradează după multe operații DML
- ▶ Reconstruirea indecșilor este necesară dar costisitoare
- ▶ B<sup>+</sup>-arborii
  - ▶ Măresc viteza de găsim a datelor și elimină necesitatea de reorganizare continuă
  - ▶ Sunt utilizați extensiv pentru indexarea datelor în SGBD-urile relaționale

# Structura unui B<sup>+</sup>-arbore (1)

---

- ▶ Un arbore echilibrat a.i. toate frunzele sunt pe același nivel
- ▶ Structura unui nod:

P <sub>1</sub>	K <sub>1</sub>	P <sub>2</sub>	K <sub>2</sub>	...	P <sub>m</sub>	K <sub>m</sub>	P <sub>m+1</sub>
----------------	----------------	----------------	----------------	-----	----------------	----------------	------------------

- ▶ K<sub>i</sub> – valori ale cheii de căutare
- ▶ P<sub>i</sub> pointeri către
  - ▶ Noduri de pe nivelul imediat inferior
  - ▶ Dacă nodul este frunză, către o înregistrare din fișierul cu date sau către blocuri de pointeri către înregistrări
- ▶ Arborele este caracterizat de o constantă *m* ce specifică numărul maxim de valori ce pot fi stocate într-un nod (numărul maxim de pointeri sau descendeți ai nodului este *m+1*)
  - ▶ De regulă *m* este calculat a.i. dimensiunea unui nod să fie egală cu cea a unui bloc de date
- ▶ Valorile cheii de căutare sunt ordonate crescător în cadrul fiecărui nod

$$K_1 < K_2 < K_3 < \dots < K_m$$

# Structura unui B<sup>+</sup>-arbore (2)

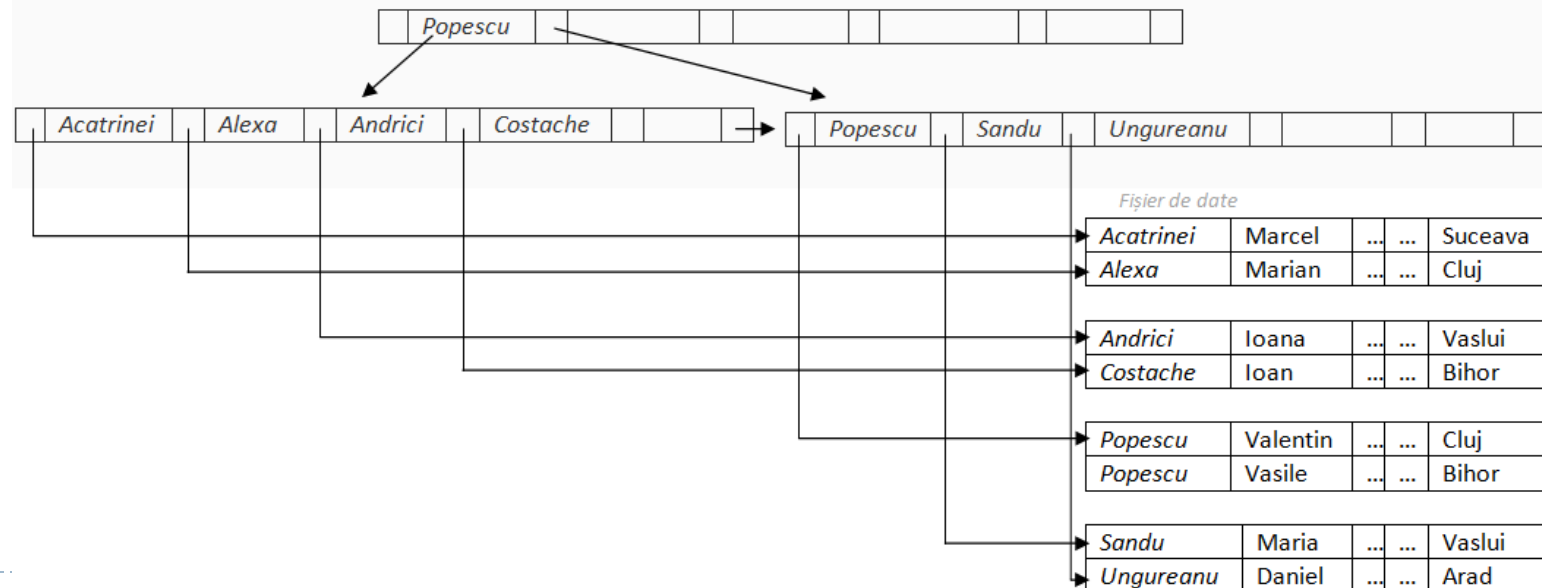
---

$P_1$	$K_1$	$P_2$	$K_2$	...	$P_m$	$K_m$	$P_{m+1}$
-------	-------	-------	-------	-----	-------	-------	-----------

- ▶ Pentru un nod cu  $m+1$  pointeri:
  - ▶ Toate valorile cheii de căutare ce apar în subarboarele către care trimite  $P_1$  sunt mai mici decât  $K_1$
  - ▶ Pentru pointerul  $P_i$ ,  $2 \leq i \leq m$ , toate valorile cheii de căutare din subarboarele spre care acesta trimite sunt mai mari decât sau egale cu  $K_{i-1}$  și mai mici decât  $K_i$
  - ▶ Toate valorile cheii de căutare din subarboarele spre care trimite  $P_{m+1}$  sunt mai mari decât sau egale cu  $K_m$

# Reguli pentru ocuparea nodurilor

- ▶ Nu e obligatoriu ca nodurile să fie complet ocupate:
  - ▶ **Rădăcina** are cel puțin **2** și cel mult  **$m + 1$**  pointeri/descendenți (respectiv, cel puțin **1** și cel mult  **$m$**  valori, ordonate crescător)
  - ▶ Fiecare nod de pe un nivel **intern** are cel puțin  **$\lceil (m+1)/2 \rceil$**  și cel mult  **$m + 1$**  pointeri/descendenți (echivalent, cel puțin  **$\lfloor m/2 \rfloor$**  și cel mult  **$m$**  valori ordonate crescător)
  - ▶ Fiecare nod **frunză** are cel puțin  **$\lfloor m/2 \rfloor$**  și cel mult  **$m$**  valori; toți pointerii trimit către fișierul de date, exceptând ultimul pointer care trimite către următorul nod frunză (cu valori mai mari).



# B<sup>+</sup>-arbori – parametrul *m*

## Exemplu

---

### ▶ Presupunând că

- ▶ 1 bloc de memorie = 1024 octeți
- ▶ Cheia de căutare = un șir de maxim 20 caractere (1 caracter = 1 octet)
- ▶ 1 pointer = 8 octeți

Care este constanta arborelui, adică numărul maxim de valori într-un nod?

### ▶ Răspuns

- ▶ Identificăm cea mai mare valoare  $m$  care satisface  $20m + 8(m + 1) \leq 1024$ .
- ▶  $m=36$

### ▶ Structura arborelui

- ▶ Rădăcina: cel puțin 2 pointeri, cel mult 37 pointeri (echivalent: între una și 36 valori)
- ▶ Nod intern: cel puțin 19, cel mult 37 pointeri
- ▶ Nod frunză: cel puțin 18, cel mult 36 valori (echivalent minim 18 și maxim 36 pointeri către fișierul cu date)

# B<sup>+</sup>-arbori

## Observații

---

- ▶ Pentru că nodurile sunt conectate prin pointeri, blocuri apropiate logic nu trebuie să fie neapărat și apropiate fizic
- ▶ Toate nivelele exceptând nivelul frunză formează o ierarhie de indecși rari externi peste nivelul frunză
- ▶ Nivelul frunză formează un index secvențial dens peste fișierul cu date
  
- ▶ B<sup>+</sup>-arboarele conține un număr relativ mic de niveluri, în cazul cel mai nefavorabil:
  - ▶ Cel mult  $\lceil \log_{\lceil (m+1)/2 \rceil}(K) \rceil$  pentru K valori a cheii de căutare
    - ▶ Nivelul 2: cel puțin 2 noduri
    - ▶ Nivelul 3: cel puțin  $2 * \lceil (m+1)/2 \rceil$  noduri
    - ▶ Nivelul 4: cel puțin  $2 * \lceil (m+1)/2 \rceil * \lceil (m+1)/2 \rceil$  noduri
    - ▶ etc...
- ▶ Inserările și ștergerile sunt procesate eficient: restructurarea indexului necesită timp logaritmic

# Interogări pe $B^+$ - arbori

---

Scop: determinați toate înregistrările din fișierul de date care corespund valorii  $k$  a cheii de căutare

1.  $N$  = rădăcina
2. Repetă
  - Identifică în  $N$  cea mai mică valoare a cheii de căutare care e mai mare decât  $k$
  - Dacă o astfel de valoare  $K_i$  există, atunci  $N = P_i$ 
    - altfel  $N = P_n$  ( $k \geq K_{n-1}$ )până când  $N$  este frunză
3. Dacă există  $K_i = k$  în frunza  $N$ , pointerul  $P_i$  trimite către înregistrarea dorită  
Altfel, nu există nici o înregistrare cu valoarea  $k$  a cheii de căutare

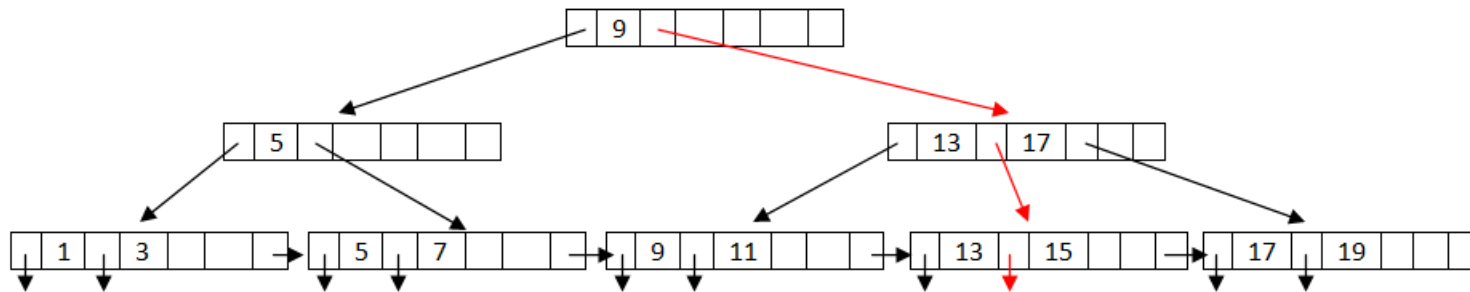
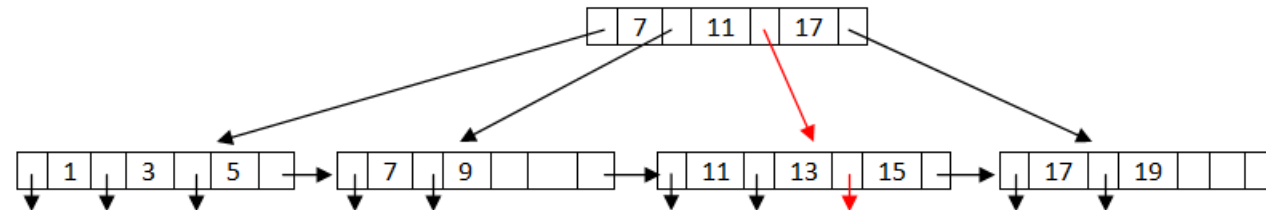
# Interogări

## Exemple

---

Cheia de căutare stochează toate valorile impare între 1-19

Reprezentați posibil arbori pentru  $m=3$  și efectuați o interogare pentru cheia 15



# Interogări pe $B^+$ -arbori

## Exercițiu

---

- ▶ Dat  $m=100$  (fiecare nod are dimensiunea unui bloc!)
- ▶ Pentru 1 milion de valori a cheii de căutare, cât de multe noduri (echivalent blocuri pe disc) sunt accesate la o căutare în  $B^+$ -arbore? (R: 4)
- ▶ Dar dacă e utilizat un index secvențial? (R: 20)

# Actualizări în B<sup>+</sup>-arbori

## Inserarea

---

După inserarea unei înregistrări în fișierul de date cu valoarea  $k$  a cheii de căutare, la care trimite pointerul  $p$ :

1. Găsește nodul frunză care ar trebui să conțină  $k$
2. Dacă valoarea există în nodul frunză:
  - ▶ Aduă pointerul  $p$  în bucketul corespunzător valorii  $k$  a cheii
3. Dacă valoare nu există
  - ▶ Dacă este loc în nodul frunză, inserează perechea  $(p, k)$Altfel, divide nodul frunză ->

# Actualizări în B<sup>+</sup>-arbori

## Inserare: divizarea unui nod

---

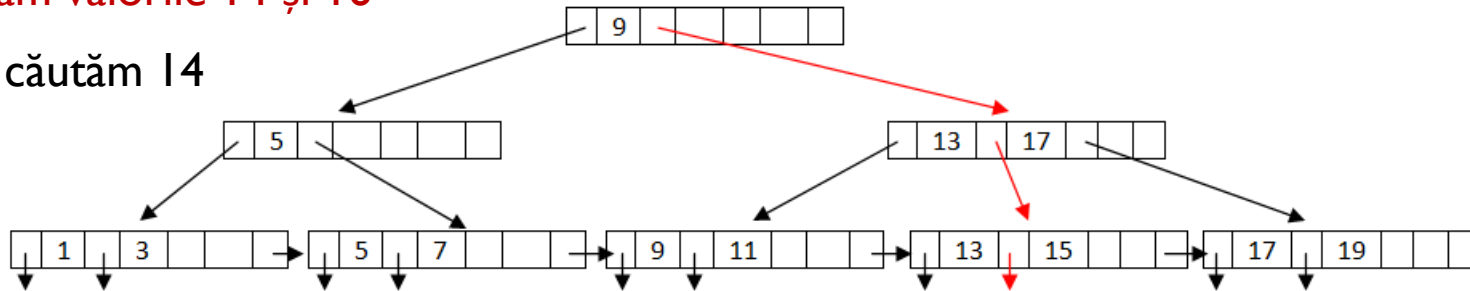
- ▶ Divizarea unui nod frunză la inserarea unei perechi noi (p<sub>i</sub>, k<sub>i</sub>):
  1. Se iau cele  $n$  perechi ordonate, inclusiv cea nou creată (p<sub>i</sub>, k<sub>i</sub>). Se păstrează primele  $\lceil n/2 \rceil$  perechi în nodul frunză existent și crează unul nou, P, care să conțină restul perechilor
  2. Fie  $k$  cea mai mică valoare din P. Inserează perechea (k,p) în nodul părinte a frunzei care s-a divizat – unde p este pointerul către noua frunză P.
  3. Dacă nodul părinte este plin, acesta trebuie să se dividă, propagând în sus divizarea până când se ajunge la un nod care nu e complet ocupat. În cel mai rău caz, nodul rădăcină este divizat, caz care crește adâncimea arborelui.
- ▶ Divizarea unui nod intern N la inserarea unei perechi (k,p)
  1. Se creează un nod temporar M care stochează  $m+2$  pointeri și  $m+1$  valori; se inserează perechile din N împreună cu perechea (k,p), ordonate
  2. Se copiază  $P_1, K_1, \dots, K_{\lceil (m+1)/2 \rceil - 1}, P_{\lceil (m+1)/2 \rceil}$  din M înapoi în N
  3. Se copiază  $P_{\lceil (m+1)/2 \rceil + 1}, K_{\lceil (m+1)/2 \rceil + 1}, \dots, K_{m+1}, P_{m+2}$  din M într-un nod nou N'
  4. Se inserează (K <sub>$\lceil (m+1)/2 \rceil$</sub> , pN') în părintele nodului N

# Actualizări în B<sup>+</sup>-arbori

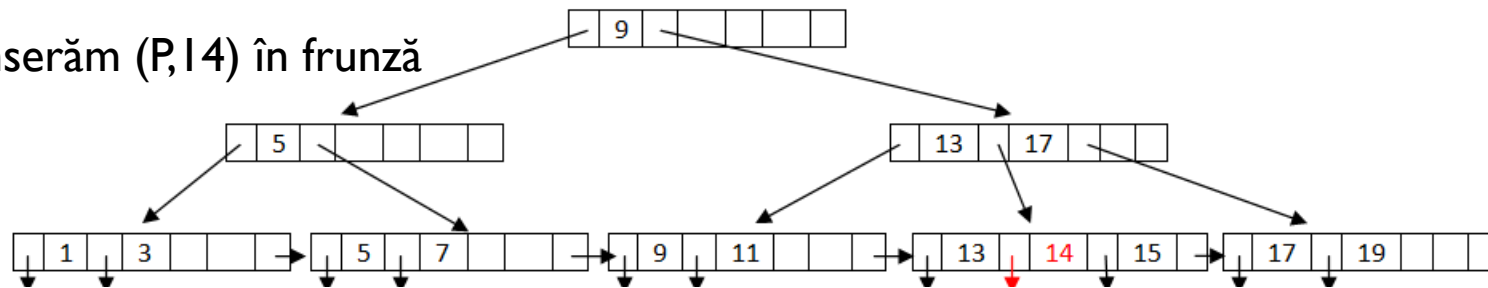
## Inserare: Exemplu (1)

Inserăm valorile 14 și 16

Pas 1: căutăm 14

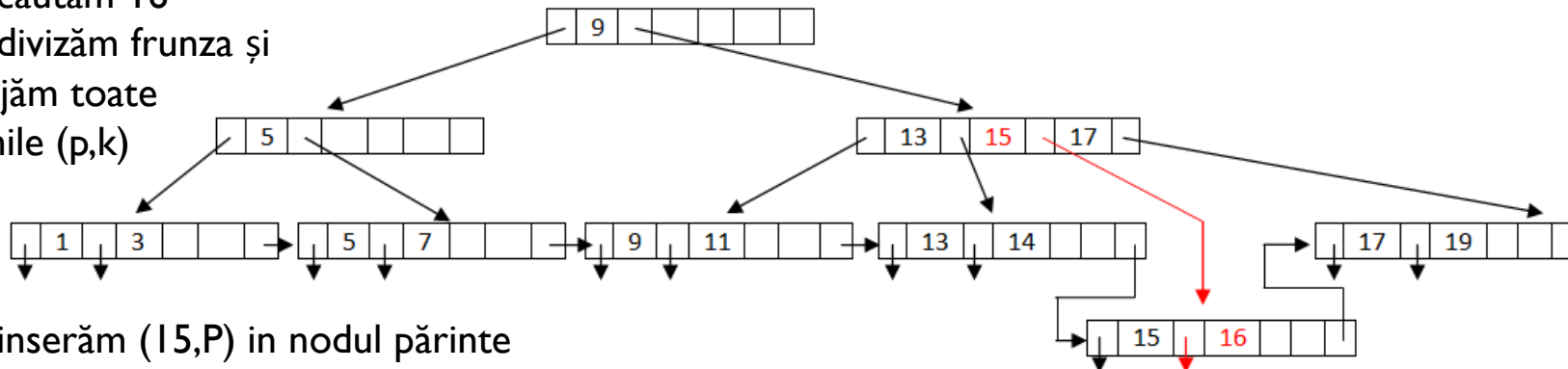


Pas 2: inserăm (P,14) în frunză



Pas 3: căutăm 16

Pas 4: divizăm frunza și  
rearanjăm toate  
Perechile (p,k)

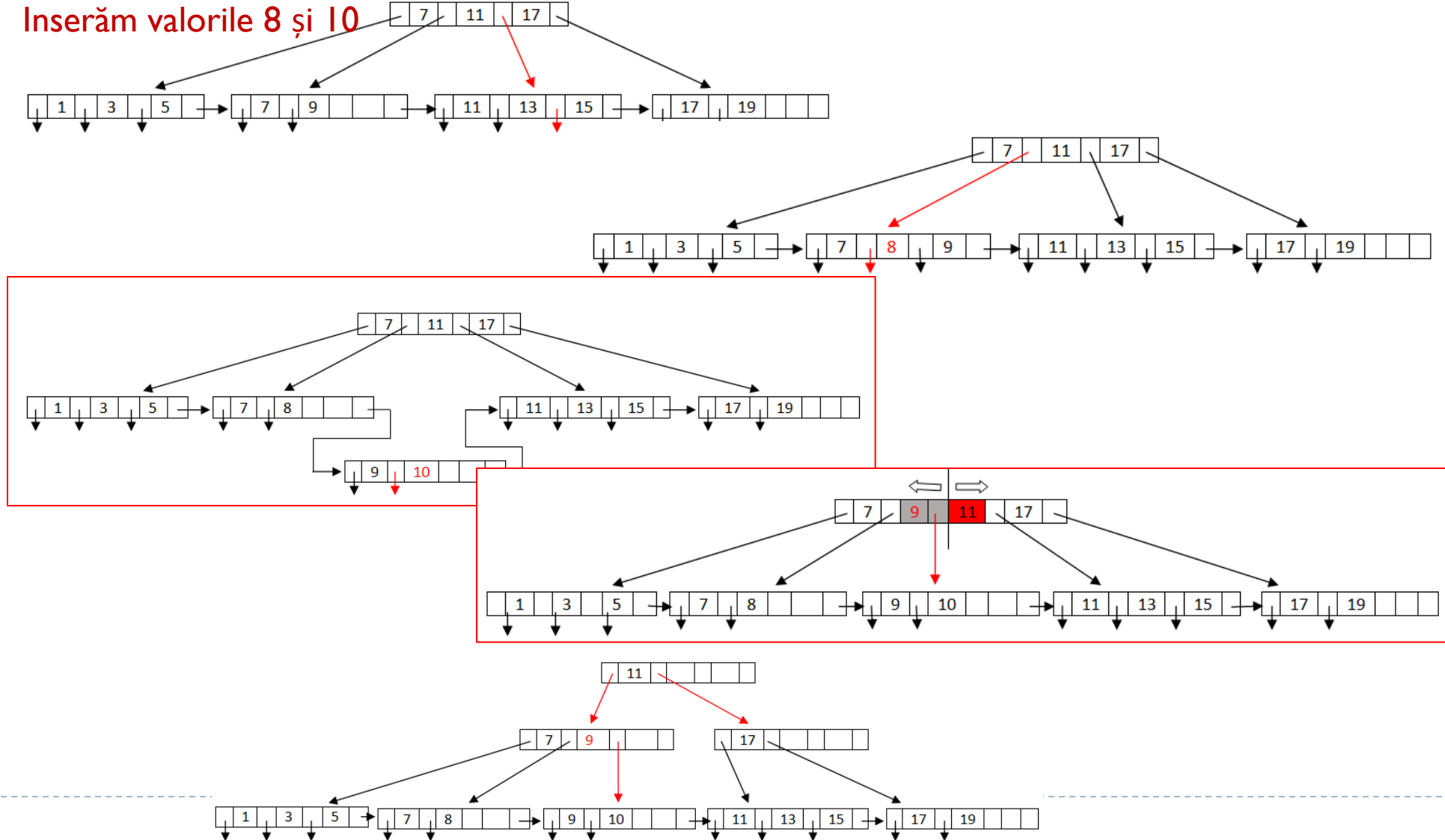


Pas 5: inserăm (15,P) in nodul părinte

# Actualizări în B<sup>+</sup>-arbori

## Inserare: Exemplu (2)

Inserăm valorile 8 și 10



# Actualizări în B<sup>+</sup>-arbori

## Ștergerea

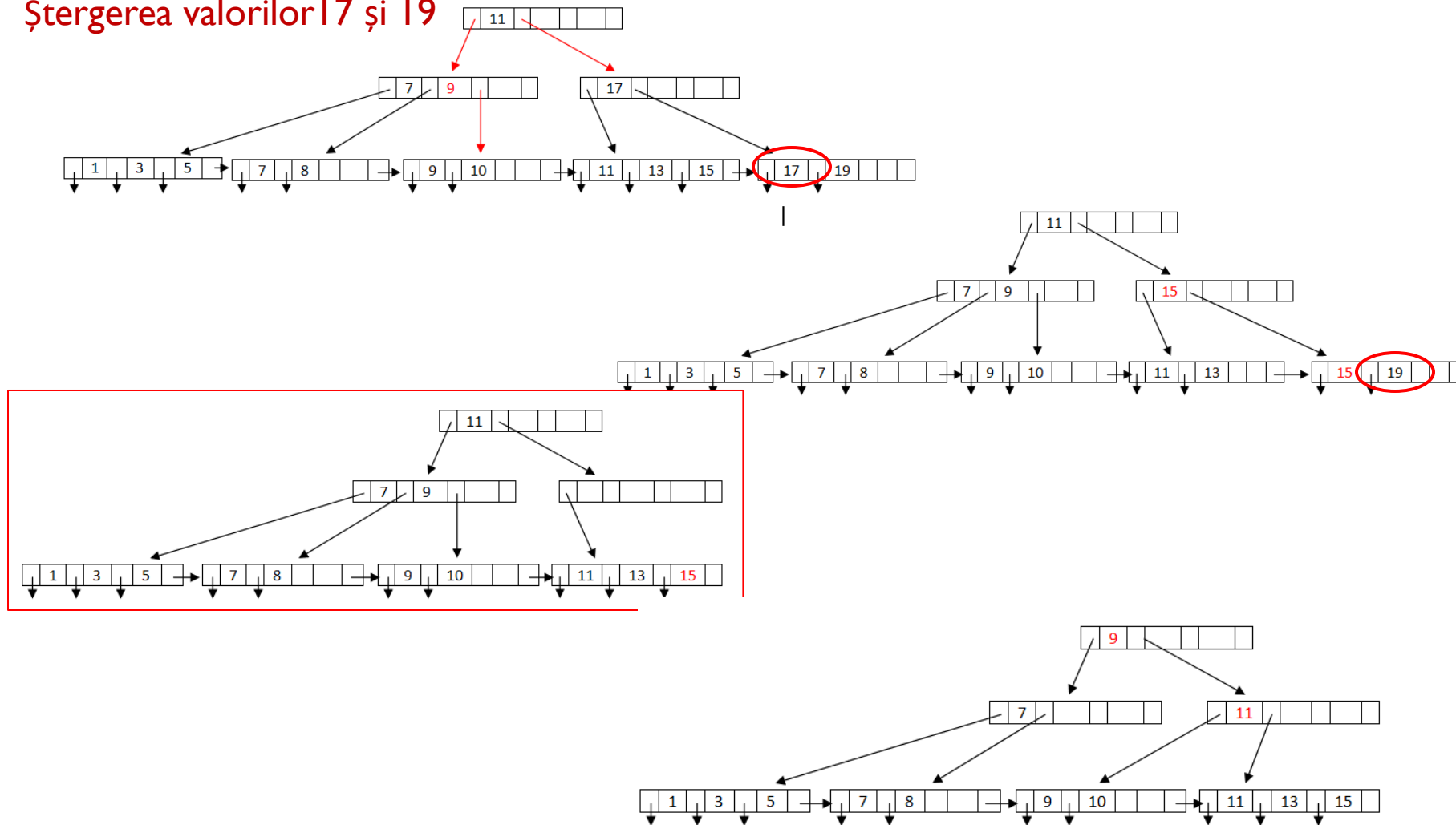
---

1. Se șterge înregistrarea din fișierul cu date; fie  $k$  valoarea cheii și  $p$  pointerul ce ne conduce către această înregistrare
2. Se identifică în index frunza care conține valoarea  $k$
3. Dacă pointerul  $p$  către înregistrarea ștearsă face parte dintr-un bucket în index,  $p$  este șters din bucket. Altfel (sau dacă bucketul devine gol) se șterge din nodul frunză perechea  $(p, k)$
4. Dacă nodul frunză rămâne cu prea puține intrări și dacă există loc pentru acestea într-o frunză alăturată, un nod frunză este șters:
  1. Inserează toate intrările în frunza stângă și șterge frunza dreaptă
  2. Șterge perechea  $(K_{i-1}, P_i)$ , unde  $P_i$  este pointerul către nodul frunză șters din nodul părinte. Dacă este necesar, se propagă în sus ștergerea. Dacă nodul rădăcină rămâne cu un singur pointer, acesta este șters și adâncimea arborelui scade.
5. Altfel, dacă nu este suficient spațiu într-o frunză vecină, perechile (pointer, key\_value) sunt redistribuite între nodul curent și o frunză vecină:
  1. Astfel încât minimul este satisfăcut în ambele
  2. Se actualizează o pereche (key\_value, pointer) în nodul părinte dacă este necesar

# Actualizări în B<sup>+</sup>-arbori

## Ștergere: Exemplu (1)

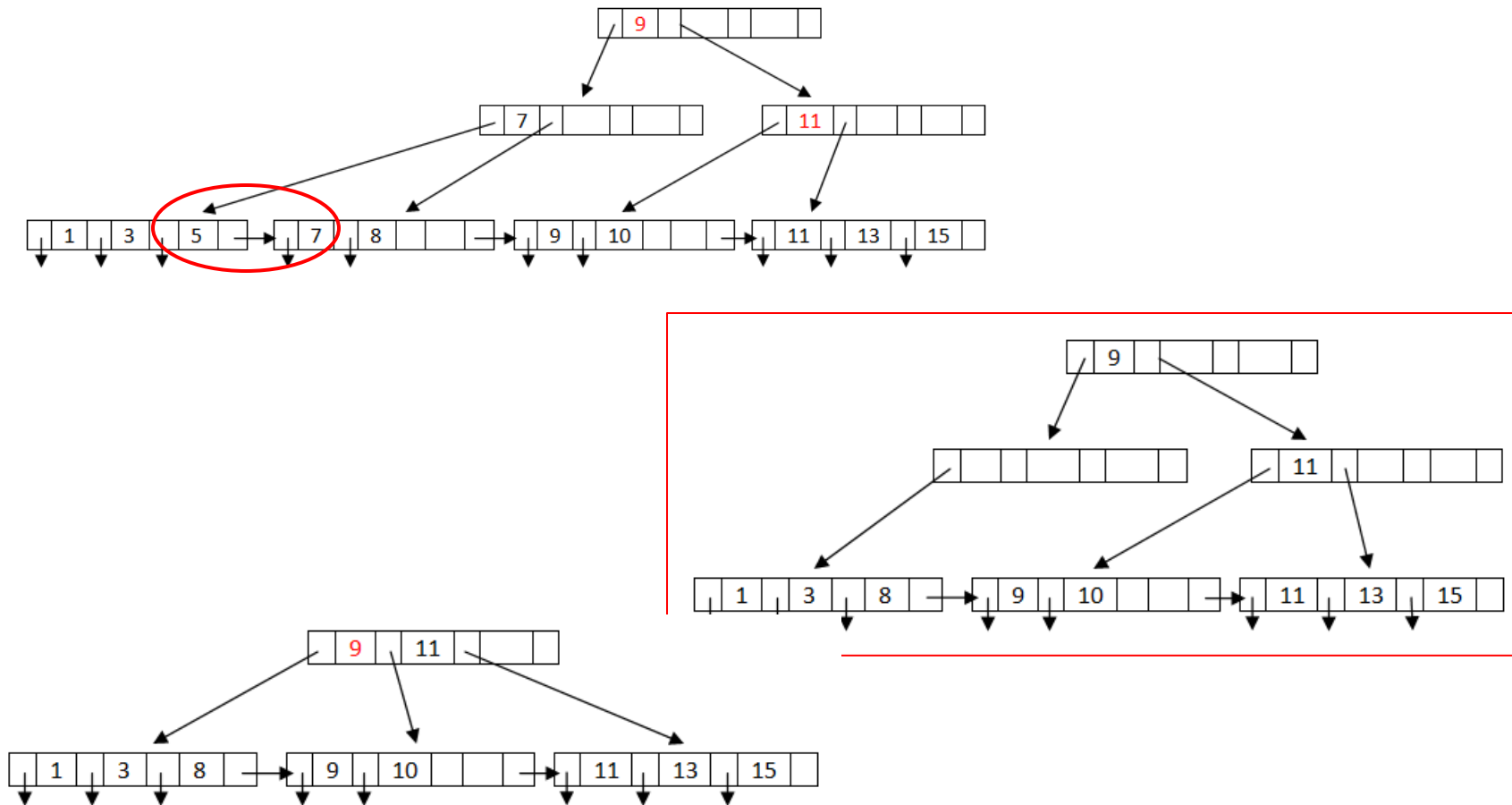
Ștergerea valorilor 17 și 19



# Actualizări în B<sup>+</sup>-arbori

## Ștergere: Exemplu (2)

Ștergerea valorilor 5 și 7



# B<sup>+</sup>-arbori

## Efficiența

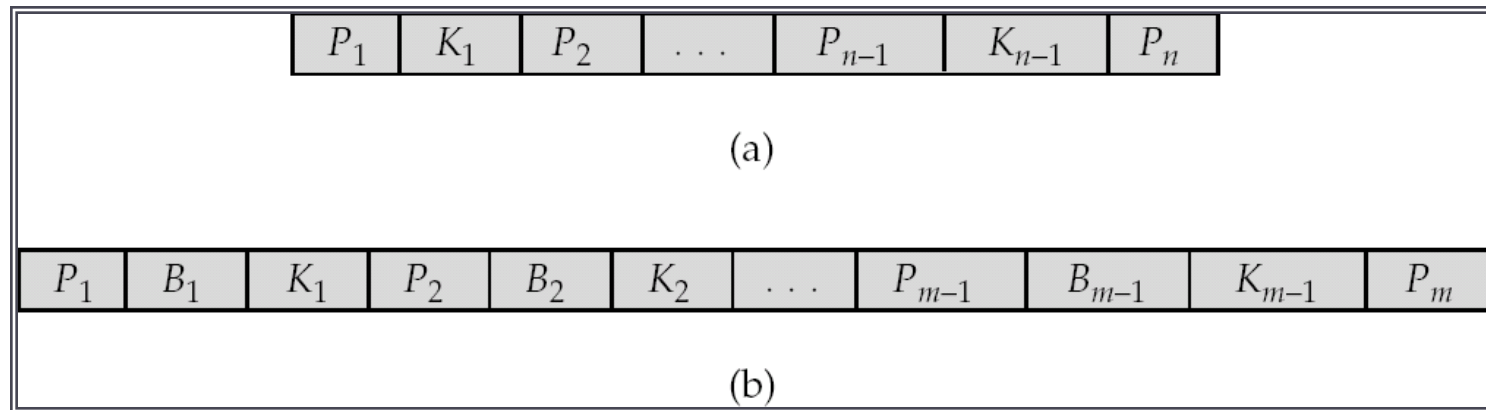
---

- ▶ **Căutare: cel mult  $\lceil \log_{(m+1)/2}(K) \rceil$  blocuri transferate**
  - ▶ Deoarece nivelul frunză este conectat formând un index secvențial dens, interogările de tip interval sunt de asemenea rezolvate eficient; nivelul frunză este de obicei o listă dublu înlănțuită pentru a permite parcurgerea în ordine crescătoare și descrescătoare a valorilor.
  - ▶ Ordonarea înregistrărilor unei interogări (order by) poate beneficia de existența unui index asociat atributului criteriu de ordonare
  
- ▶ **Inserarea, ștergerea: cel mult  $2 \lceil \log_{(m+1)/2}(K) \rceil$  blocuri transferate**

Indecși ordonați:  
B-arbori

# B-arbori

- ▶ Similari arborilor  $B^+$  dar permit o singură apariție a unei valori a cheii
- ▶ Nu toate valorile cheii apar astfel pe nivelul frunză
  - ▶ Fiecare valoare vine cu un pointer în plus

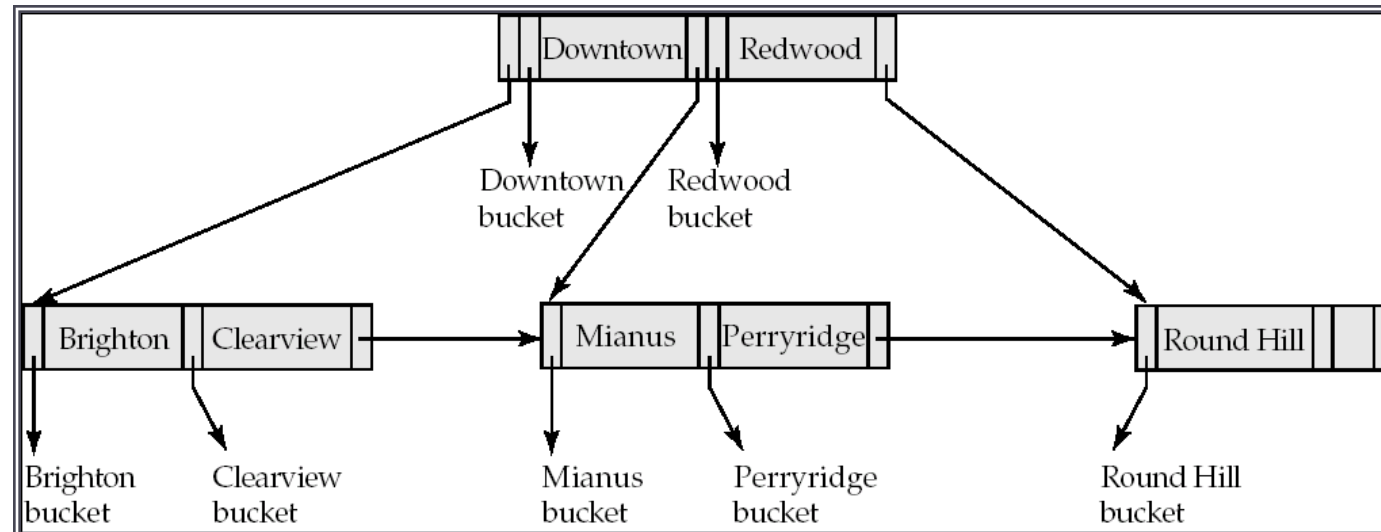
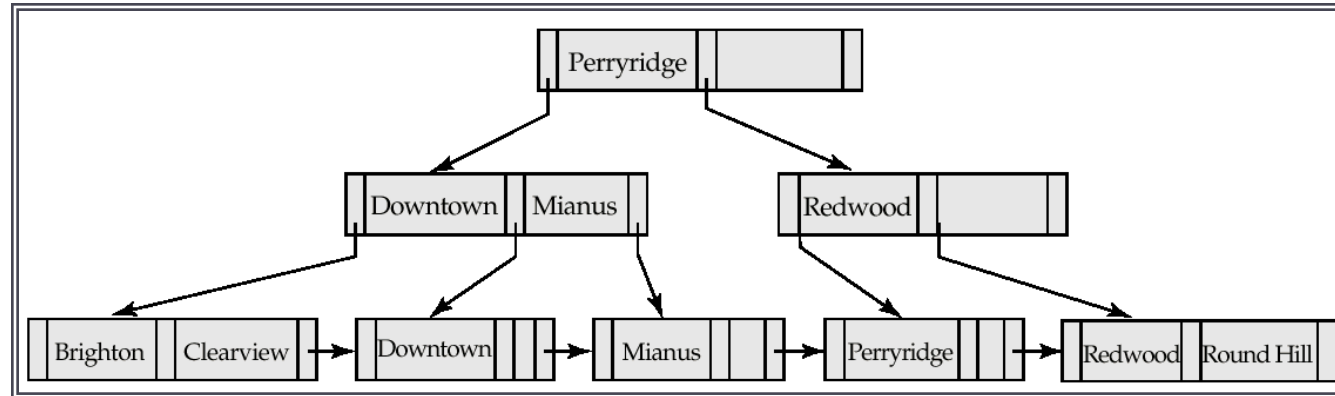


(a) Nod într-un  $B^+$ -arbore; (b) nod într-un B-arbore

- ▶ Pointerii  $B_i$  conduc către înregistrări din fișierul cu date sau *buckets* de pointeri către înregistrări

# Index B-arbore

## Exemplu\*



# Indecși B-arbore

## Observații

---

- ▶ **Avantaje**

- ▶ Devine posibil ca înregistrarea căutată să fie localizată înainte de a ajunge la nivelul frunză

- ▶ **Dezavantaje**

- ▶ Nodurile interne conțin mai multă informație (dar mai puține chei) rezultând în arbori cu adâncime mai mare
- ▶ Inserările și ștergerile sunt mai complicate -> implementarea este mai dificilă
- ▶ Nu este posibil să scanăm un tabel pe baza nivelului frunză

- ▶ **Avantajele nu cântăresc mai mult decât dezavantajele: B<sup>+</sup>-arborii sunt preferați în detrimentul B-arborilor de către dezvoltatorii SGBD-urilor relaționale**



## Indecși ordonați: Indecși multi-cheie

# Acces multi-cheie

---

```
SELECT *  
FROM student  
WHERE judet= 'Bihor' AND an> 2010;
```

- ▶ Sunt posibile mai multe strategii pentru a rezolva interogări cu mai multe chei de căutare:
  - ▶ Utilizarea unui index asociat atributului *judet*
  - ▶ Utilizarea unui index asociat atributului *an*
  - ▶ Utilizarea ambilor indecși de mai sus urmată de operația de intersecție a mulțimilor de pointeri  
Dar dacă doar una dintre condiții este satisfăcută de un număr mare de înregistrări?

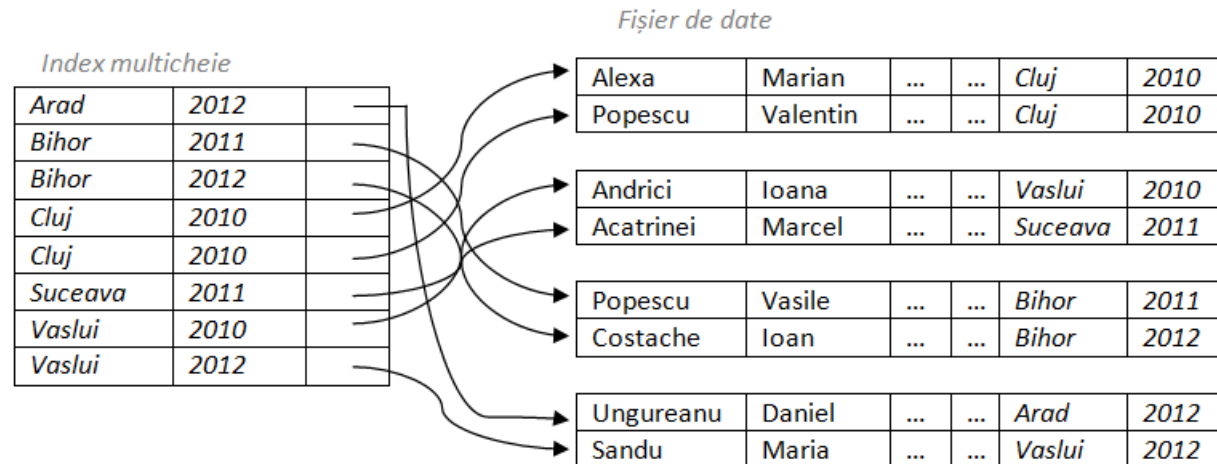
# Indecși multi-cheie

- ▶ Cheia de căutare este compusă din mai mult de un atribut
- ▶ Ordinea lexicografică este utilizată:  $(a_1, a_2) < (b_1, b_2)$  dacă
  - ▶  $a_1 < b_1$  sau
  - ▶  $a_1 = b_1$  și  $a_2 < b_2$

Ex. Considerăm indexul multi-cheie (judet, an)

Rezolvă acesta la fel de eficient ambele interogări de mai jos?

- where judet = 'Bihor' AND an > 2010
- where judet > 'Bihor' AND an = 2010



# Eficiența

---

- ▶ Ordinea atributelor într-un index multi-cheie contează!
- ▶ Eficiența depinde de selectivitatea atributelor



# Indecși ordonați multi-cheie: kd-arbori

# kd-arbori

---

- ▶ Nu sunt utilizați de regulă în bazele de date relaționale dar îi menționăm fiind o structură de căutare utilizată frecvent în bazele de date spațiale/geografice;
- ▶ Generalizare a arborelui binar de căutare:
  - ▶  $k$  = numărul de atribute a cheii de căutare multi-atribut
  - ▶ Fiecare nivel din arbore corespunde unuia dintre atribute
  - ▶ Succesiunea de nivele reprezintă iterații peste mulțimea de atribute
  - ▶ Pentru a obține arbori echilibrați, de obicei mediana este utilizată ca valoare în noduri interne

# Organizarea de tip hash și Indecși hash

# Organizarea hash a fișierului cu date

---

- ▶ În **organizarea de tip hash** a fișierului cu date, înregistrările nu sunt ordonate pe baza valorilor unui atribut, ci sunt grupate în bucketuri cu ajutorul unei funcții hash (dimensiunea unui bucket corespunde, de regulă, unui bloc de memorie)
  - ▶ **Bucket**: unitate de stocare ce poate conține una sau mai multe înregistrări
  - ▶ **Funcție hash** = funcție de *dispersie* – mapează valori dintr-o mulțime de dimensiune variabilă la o mulțime fixă de valori
- ▶ **Funcția hash**  $h:K \rightarrow B$  este în acest caz o funcție ce mapează valori ale cheii de căutare la o mulțime fixată de bucketuri
- ▶ Înregistrările cu valori diferite ale cheii de căutare pot fi mapate la același bucket
  - ▶ Căutarea implică parcurgerea scvențială a bucketului

# Funcții hash

---

- ▶ Cerințe
  - ▶ Distribuție uniformă
  - ▶ Asignări aleatorii (spre deosebire de *locality sensitive hashing*)
- ▶ Funcțiile hash tipice utilizează operații pe reprezentarea binară (internă) a cheii de căutare
- ▶ Pot să apară situații în care dimensiunea bucketului (a blocului de memorie) e prea mică pentru a stoca toate înregistrările mapate -> blocuri de exces sunt utilizate

# Organizarea de tip hash a fișierului cu date

## Exemplu

---



Atributul *prenume* este cheia de căutare iar numărul de bucketuri este fixat la 4:  
Funcția hash:  $h: \text{Dom}(\text{nume}) \rightarrow \{0, 1, 2, 3\}$  – calculează suma reprezentării binare modulo 4

Ex:

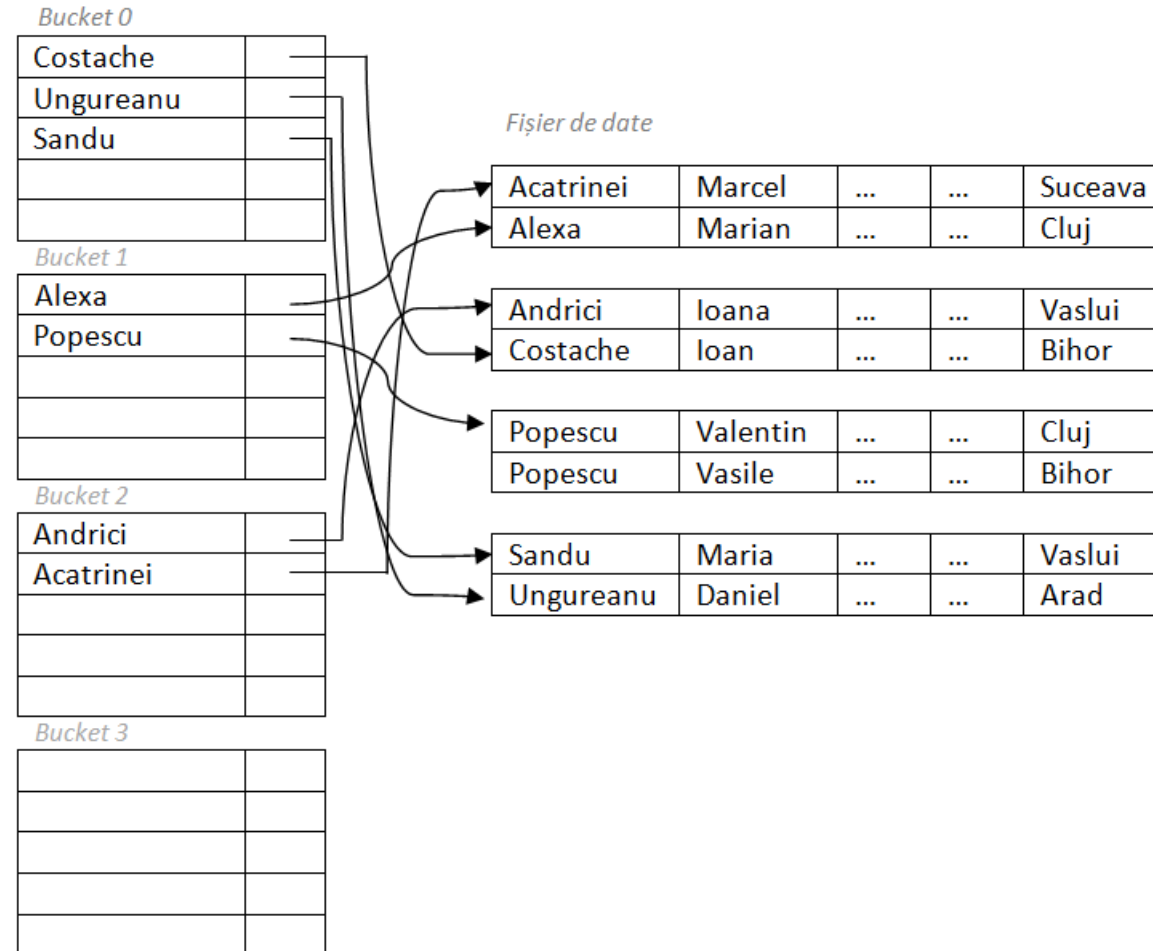
'Acatrinei':

'1000001 1100011 1100001 1110100 1110010 1101001 1101110 1100101 1101001'

$h(\text{'Acatrinei'}) = 34\%4 = 2.$

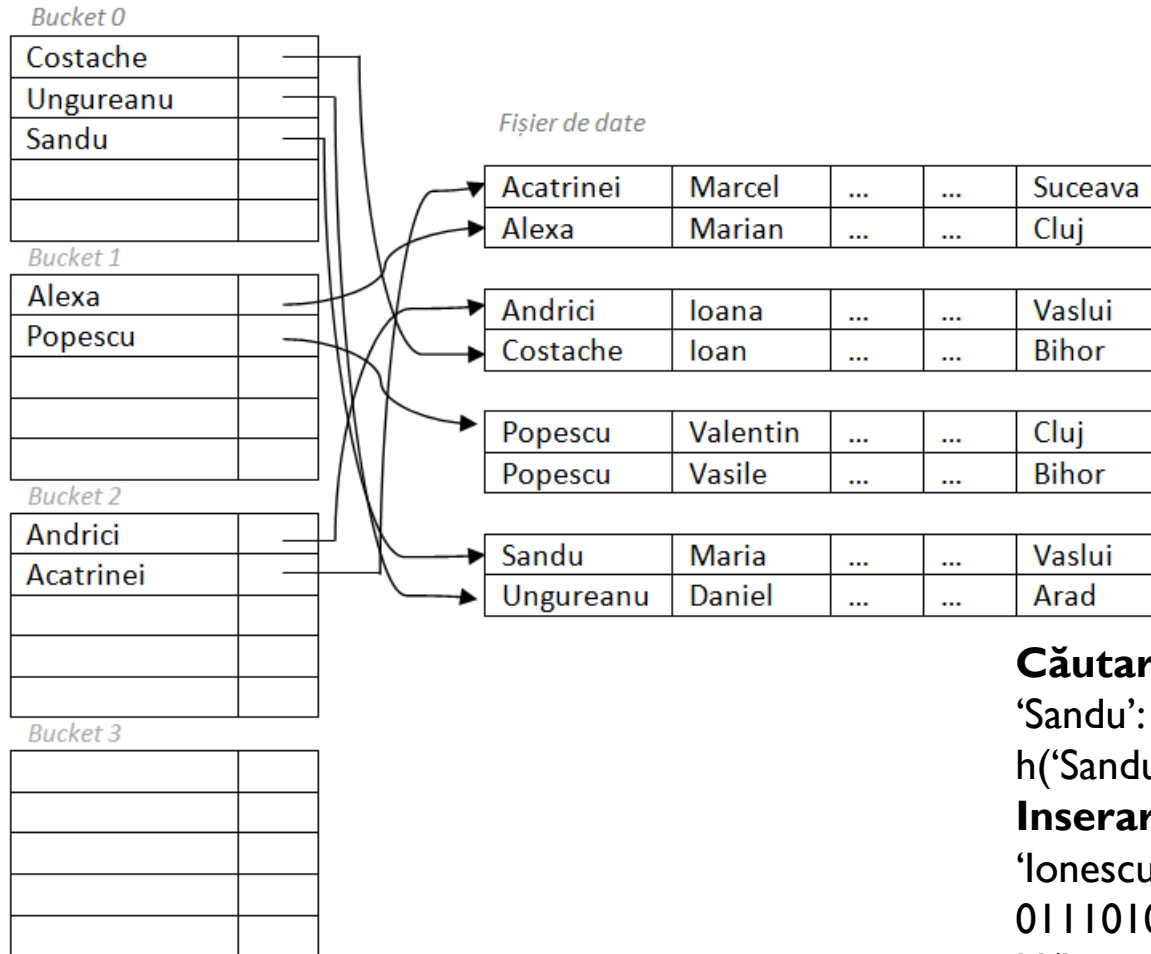
# Indecși de tip hash

- ▶ Organizează valorile cheii de căutare cu pointerii asociați într-o tabelă hash



# Indecși hash

## Operații



### Căutare:

'Sandu': 01010011 01100001 01101110 01100100 01110101

$h(\text{'Sandu'}) = 20\%4 = 0 \rightarrow$  scanează bucketul 0

### Inserare:

'Ionescu': 01001001 01101111 01101110 01100101 01110011 01100011  
01110101

$H(\text{Ionescu}) = 32\%4 = 0 \rightarrow$  inserează mai întâi înregistrarea în fișierul de date și apoi intrarea index în bucketul 0

**Ștergere:** calculează hashul, scanează bucketul, șterge



# Indecși hash

## Eficiență

---

- ▶ În căutarea unei singure valori, în absența coliziunilor, gasirea unei înregistrări necesită citirea unui singur bloc ( $O(1)$ )
- ▶ Pentru interogări de tip interval, indecșii hash nu sunt eficienți. **DE CE?**
- ▶ În practică:
  - ▶ Postgres și SQLServer implementează indecșii hash
  - ▶ Oracle implementează organizarea de tip hash a fișierului cu date dar nu și indecșii hash

# Hash dinamic

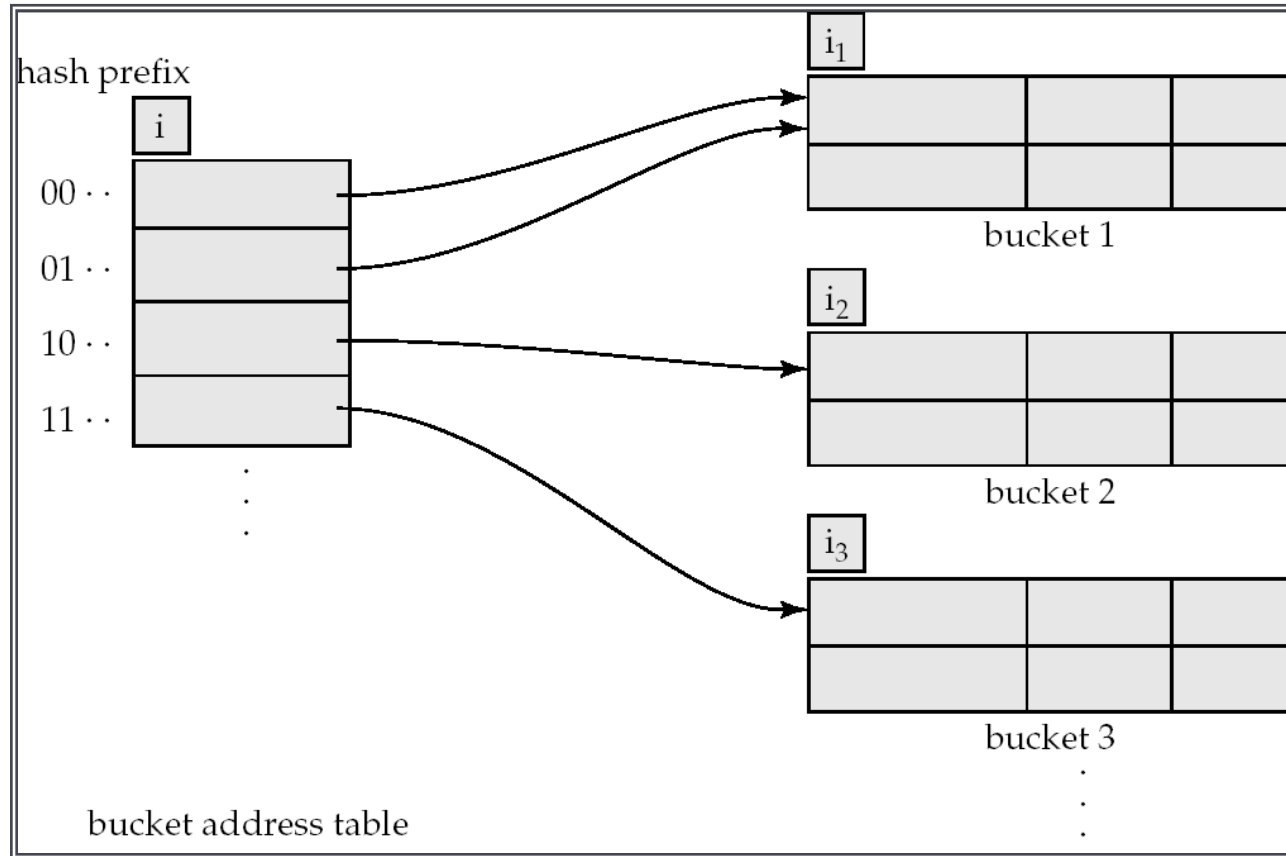
## Motivație

---

- ▶ Funcția  $h$  mapează valorile cheii de căutare la o mulțime fixă de blocuri.
  - ▶ Dacă dimensiunea fișierului cu date crește, blocuri de exces sunt generate
  - ▶ Dacă dimensiunea fișierului se micșorează, spațiu este alocat inutil
- ▶ Soluții:
  - ▶ Reorganizări periodice cu o nouă funcție hash (costisitor, necesită întreruperea operațiilor bazei de date)
  - ▶ Modificarea dinamică, după necesități, a numărului de bucketuri
- ▶ Din a doua categorie, Hashul extensibil modifică funcția hash astfel:
  - ▶ Generează valori într-o mulțime mare, de regulă întregi pe 32 biți
  - ▶ La un anumit moment utilizează doar un prefix (doar primii  $i$  biți) a cărui dimensiune crește sau descrește după necesități

# Organizarea de tip Hash extensibil

## Structura generală



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

# Hash extensibil

## Implementare

---

- ▶ Fiecare bucket  $j$  are asociată o valoare  $i_j$  care specifică lungimea prefixului
  - ▶ Toate intrările din bucketul  $j$  au aceleași valoare pe primii  $i_j$  biți
- ▶ Pentru a căuta o valoare cu cheia  $K_j$ :
  - ▶ Calculează  $h(K_j) = X$
  - ▶ Utilizează doar primii  $i$  biți ai  $X$ , scanează tabela de adrese și umărește pointerul către bucket
- ▶ Pentru a insera o înregistrare cu cheia de căutare  $K_j$ :
  - ▶ Găsește bucketul  $j$  ca mai sus
  - ▶ Dacă este loc în bucket inserează înregistrarea
  - ▶ Altfel, divide bucketul și reîncearcă inserarea ->

# Hash extensibil

## Divizarea bucketului la inserare

---

Pentru a diviza bucketul  $j$  la inserarea unei noi valori  $K_j$ :

- ▶ Dacă  $i > i_j$ 
  1. Alocă un nou bucket  $z$  și inițializează  $i_j = i_z = (i_j + 1)$
  2. Actualizează a doua jumătate a tabelului de adrese ca să trimită către bucketul  $z$
  3. Elimină înregistrările din  $j$  și reinserează-le în  $j$  sau  $z$  conform prefixului
  4. Recalculează adresa bucketului pentru  $K_j$  și execută inserarea
- 1. Dacă  $i = i_j$ 
  1. Dacă din anumite motive există o limită pentru  $i$  și aceasta este atinsă, se utilizează blocuri de exces
  2. Altfel
    1. Incrementează  $i$  și dublează dimensiunea tabelului de adrese
    2. Înlocuiește fiecare intrare din tabel cu alte două intrări, ambele trimițând la același bucket
    3. Recalculează adresa bucketului pentru  $K_j$  și realizează inserarea (acum  $i > i_j$ )

# Hash extensibil

## Ștergere

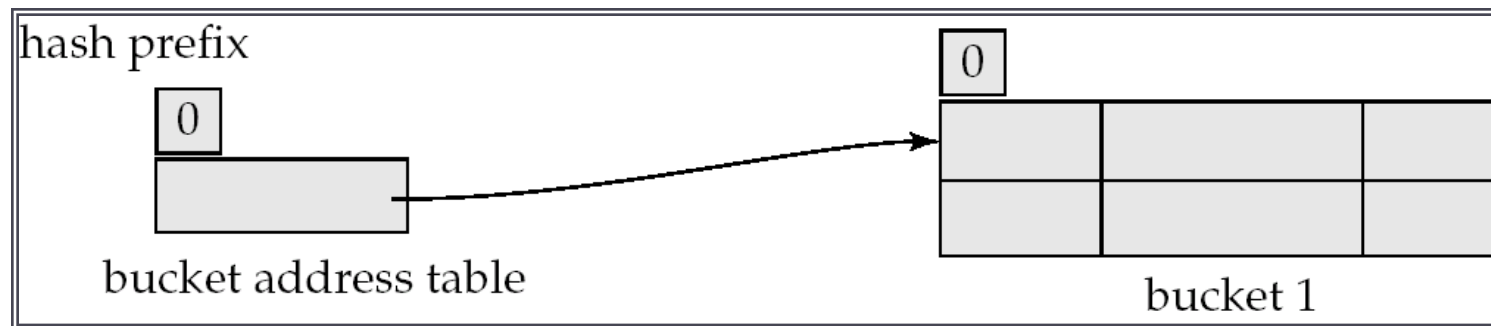
---

- ▶ Pentru a șterge o înregistrare
  - ▶ Găsește bucketul și șterge înregistrarea din acesta
  - ▶ Dacă bucketul devine gol se efectuează modificările necesare în tabela de adrese
  - ▶ Bucketurile care au aceeași valoare pentru  $i_j$  și același prefix  $i_j - 1$  sunt contopite
  - ▶ Descrește dimensiunea ( $i$ ) a tablei de adrese dacă este posibil

# Hash extensibil

## Exemplu\*

<i>branch_name</i>	<i>h(branch_name)</i>
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

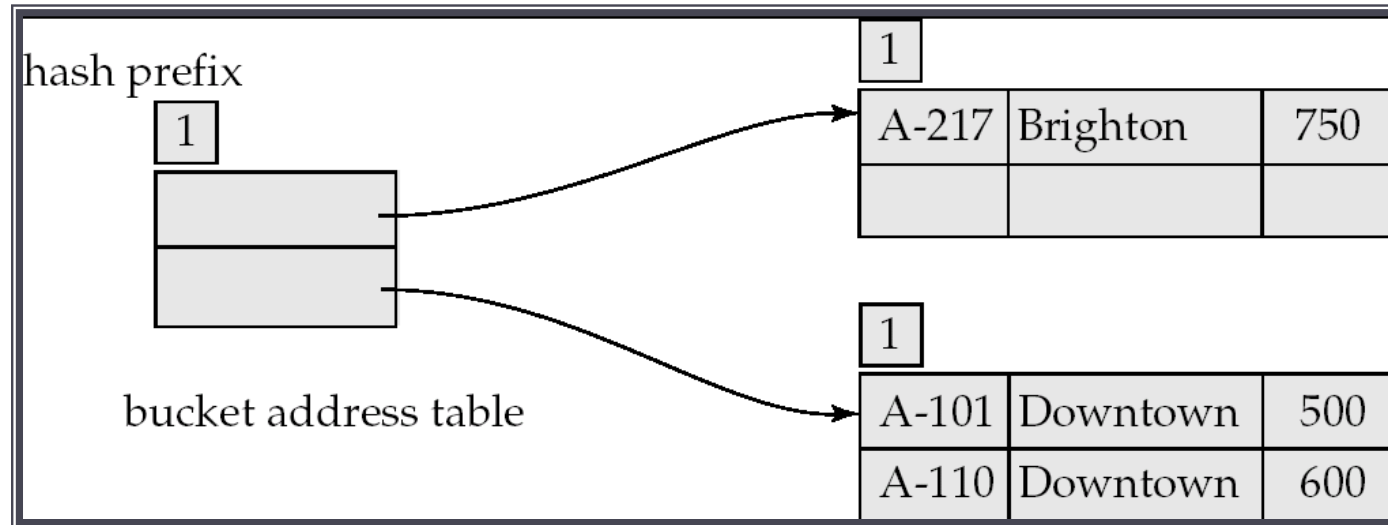


Initial hash structure, bucket size = 2

# Hash extensibil

## Exemplu\*

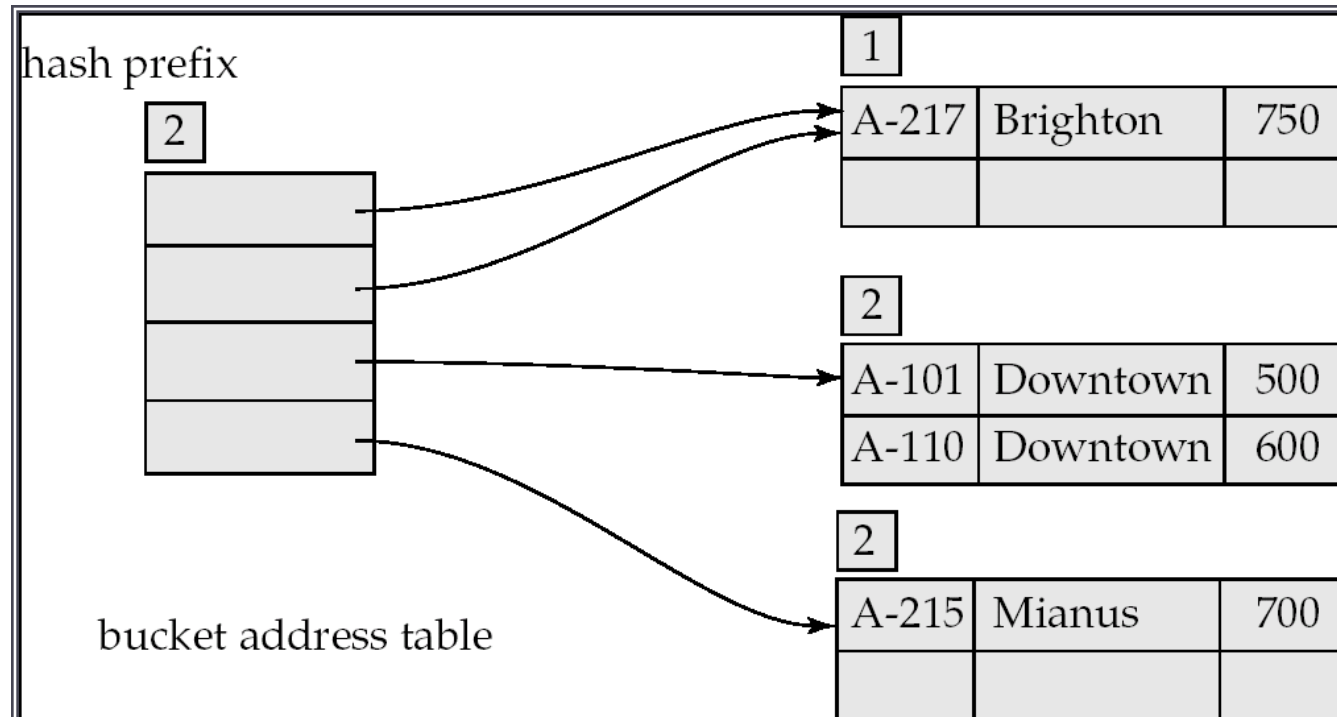
- ▶ După inserarea unei înregistrări Brighton și a două înregistrări Downtown



# Hash extensibil

## Exemplu\*

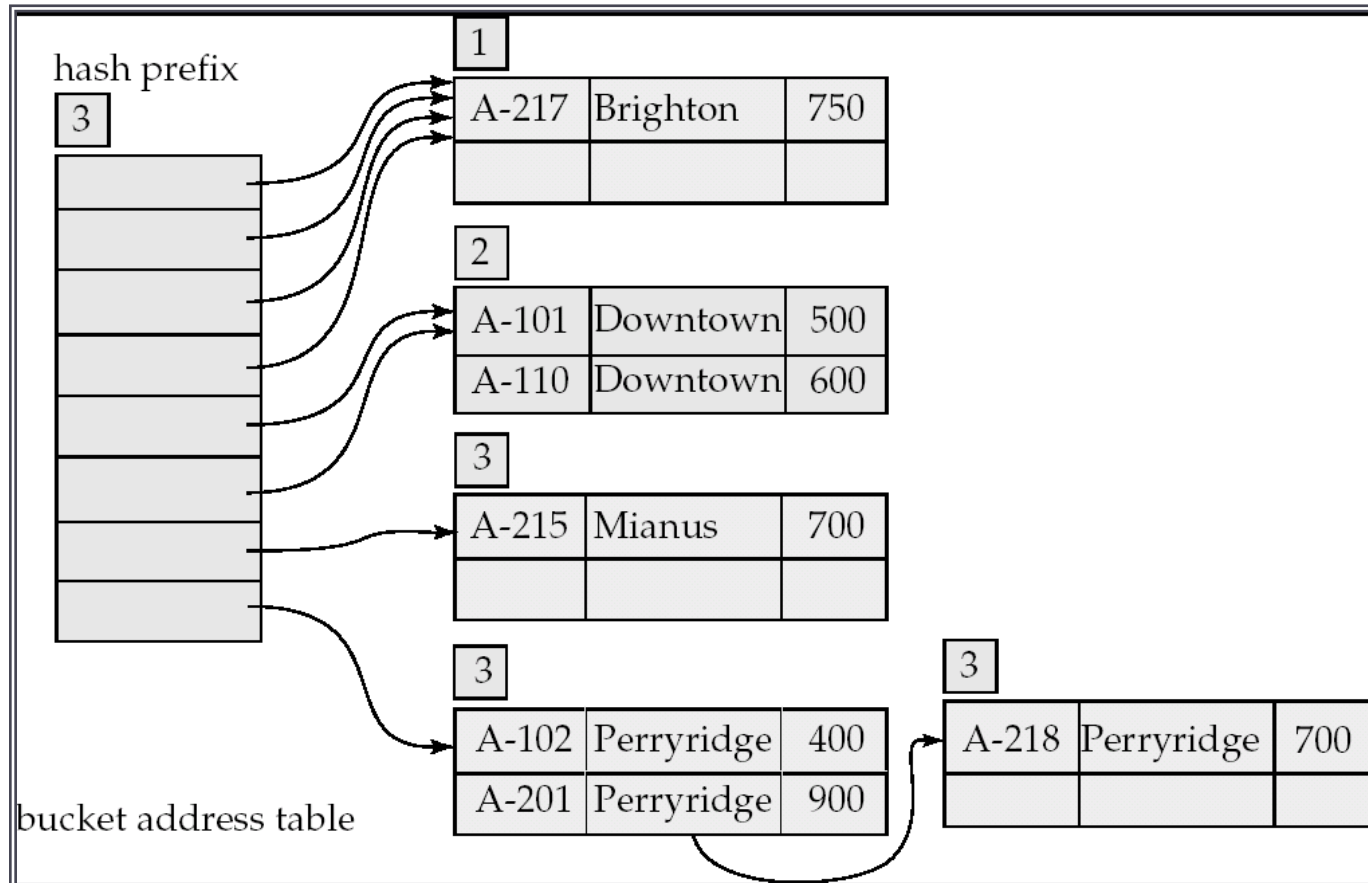
- ▶ După inserarea unei înregistrări Mianus



# Hash extensibil

## Exemplu\*

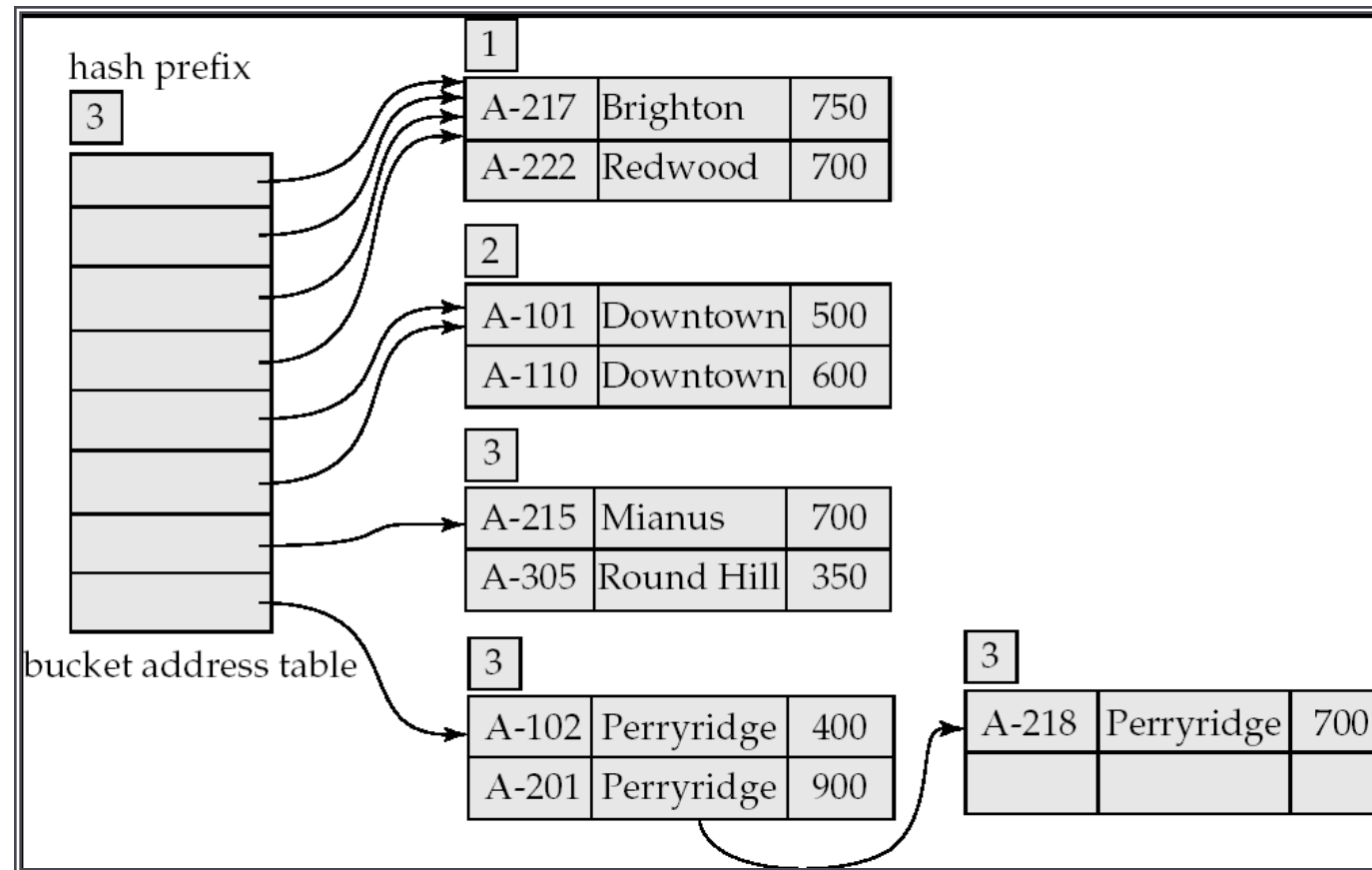
- ▶ Dupa inserarea a trei înregistrări Perryridge



# Hash extensibil

## Exemplu\*

- ▶ După inserarea înregistrărilor Redwood și Round Hill



# Hash extensibil

## Eficiență

---

- ▶ **Avantaje**
  - ▶ Performanța nu scade la creșterea dimensiunii fișierului
  - ▶ Minimizează alocarea de spațiu de stocare
- ▶ **Dezavantaje**
  - ▶ Tabela de adrese poate deveni foarte mare
    - ▶ Soluția: utilizarea unui  $B^+$ -arbore pentru o căutare eficientă în tabela cu adrese
  - ▶ Modificarea dimensiunii tabeli de adrese este costisitoare
- ▶ **În funcție de tipul interogării:**
  - ▶ Ca și hashingul static, este eficient pentru interogări punctuale dar nu și de tip interval

# Indecşi bitmap

# Indecși bitmap

- ▶ Proiectați pentru a trata eficient interogări cu mai multe chei de căutare
- ▶ Aplicabili pentru attribute ce iau un număr mic de valori distincte
- ▶ Tuplele relației sunt considerate a fi numerotate
- ▶ Structura:
  - ▶ Pentru fiecare valoare a cheii un  $>$  șir binar de lungime egală cu numărul de înregistrări
  - ▶ Valoarea 1 în șir indică faptul că înregistra de la poziția dată ia valoarea la care este atașat șirul

nume	prenume	str	loc	judet	
Alexa	Marian	Strada Florilor	Cluj Napoca	Cluj	Arad   0 0 0 0 0 0 1 0
Popescu	Valentin	Strada Unirii	Dej	Cluj	Bihor   0 0 0 0 1 1 0 0
Andrici	Ioana	Bulevardul Republicii	Vaslui	Vaslui	Cluj   1 1 0 0 0 0 0 0
Acatrinei	Marcel		Putna	Suceava	Suceava   0 0 0 1 0 0 0 0
Popescu	Vasile	Bulevardul Independentei	Oradea	Bihor	Vaslui   0 0 1 0 0 0 0 1
Costache	Ioan	Strada Teiului	Nucet	Bihor	
Ungureanu	Daniel	Aleea Amara	Arad	Arad	
Sandu	Maria	Strada Victoriei	Barlad	Vaslui	

# Indecși bitmap

## Observații

- ▶ Interogările cu mai multe selecții (chei de căutare) pot fi rezolvate cu operatori pe biți:

- ▶ Intersecția – AND
- ▶ Reuniunea – OR
- ▶ Complementarierea – NOT

**SELECT \***

**FROM studenti**

**WHERE judet IN ('Arad', 'Cluj') AND an <> 2010;**

- ▶ (Arad OR Cluj) AND NOT(2010)
- 
- ▶ Implementare eficientă:
    - ▶ La ștergere/inserare un șir binar de existență este actualizat
    - ▶ Structurile Bitmap sunt împachetate sub tipul word pe 32 sau 64 biți (operatorul AND necesită o instrucțiune CPU)

```
2010    |1 1 1 0 0 0 0 0
-----+-----
2011    |0 0 0 1 1 0 0 0
-----+-----
2012    |0 0 0 0 0 1 1 1

Arad    |0 0 0 0 0 0 1 0
-----+-----
Bihor   |0 0 0 0 1 1 0 0
-----+-----
Cluj    |1 1 0 0 0 0 0 0
-----+-----
Suceava |0 0 0 1 0 0 0 0
-----+-----
Vaslui  |0 0 1 0 0 0 0 1
```

# Definirea indecșilor în SQL

# Declararea indecșilor în SQL

---

- ▶ Standardul nu reglementează mecanismele de indexare (țin de nivelul fizic), dar în practică dezvoltatorii au căzut de acord asupra sintaxei:

- ▶ Creare:

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

E.g.: **create index** *c-index* **on** *student(judet)*

- ▶ Ștergere:

**drop index** <index-name>

- ▶ Cele mai multe SGBD-uri permit specificarea structurii pentru indexare
- ▶ Cele mai multe SGBD-uri creează implicit indecși la declararea constrângerii unique
- ▶ Uneori indecși sunt generați și la declararea constrângerilor referențiale

# Indexarea în Oracle

---

- ▶ Indecși pot fi creați pe:
  - ▶ Atribute și liste de atribute
  - ▶ Rezultatele unei funcții peste atribute
- ▶ Oracle oferă suport implicit pentru B<sup>+</sup>-arbori
- ▶ Indecșii bitmap sunt creați cu sintaxa  
`create bitmap index <index-name> on <relation-name> (<attribute-list>)`
- ▶ Oracle nu oferă suport pentru indecșii hash dar implementează organizarea fișierului de date de tip hash

# Indexarea în Oracle

## Când și cum?

---

- ▶ Este recomandat ca crearea indecșilor să aibă loc după inserarea datelor în tabel (dar e posibil să creăm indexul în orice moment)
- ▶ Se aleg coloanele potrivite:
  - ▶ Care iau (majoritar) valori distincte
  - ▶ Pentru care selecția filtrează un număr mic de tuple dintr-un tabel de dimensiuni mari (selectivitate ridicată ~ 15%)
  - ▶ Care sunt utilizate în join
- ▶ Se alege structura de date potrivită pentru indexare:
  - ▶ Coloanele au un număr redus de valori distincte și avem condiții compuse -> bitmap
  - ▶ Interogarea este de tip interval -> B+arbori
  - ▶ Interogările punctuale sunt frecvente -> organizarea hash
  - ▶ Selecție cu funcții -> indecși definiți peste funcții

# Bibliografie

---

- ▶ Capitolul 11 în *Avi Silberschatz Henry F. Korth S. Sudarshan. "Database System Concepts". McGraw-Hill Science/Engineering/Math; 6 edition (January 27, 2010)*
- ▶ Se execută scriptul *12.1create* și apoi comenzile din *12.2indexes* cu inspectarea planurilor de execuție