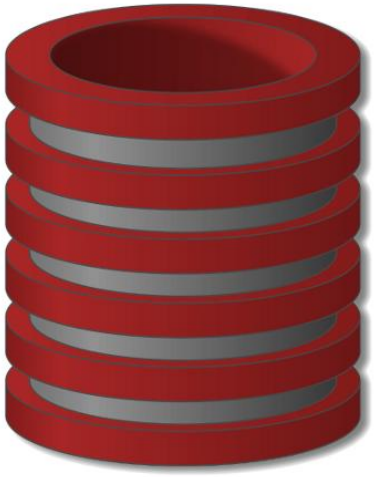


BAZE DE DATE

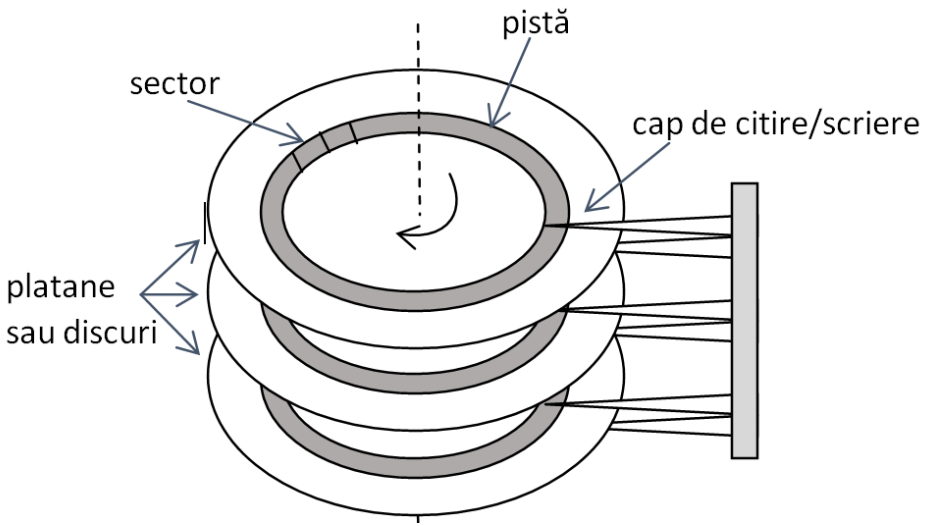
Indecși

Mihaela Elena Breabăn

Indecşi
Cuprins

- Stocarea fizica a datelor
- Indexare - motivatie
- Indecși ordonați
  - Indecși secvențiali
  - B^+-Arbori
- Hashing
  - Hashing static
  - Hashing dinamic
- Acces multi-cheie și Indecși Bitmap
- Definirea indecșilor în standardul SQL
- Indecși în Oracle

Stocarea fizica a datelor



- Viteza de citire a unui bloc este data de:
  - Viteza de miscare a bratului (pozitionare pe cilindru)
  - Viteza de rotatie a platanelor
  - Timpul de transfer

Indexare - 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

Indexare – Motivație

- Căutarea binară
  - Complexitate: $\log_2(N)$
    ($\log_2(100\ 000)=17$)

SELECT * FROM Student

WHERE sNume='Ioana';

Cum putem rezolva si aceasta interogare eficient?

- Solutia: 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

- Fișier de date – secventa de blocuri de memorie ce contine inregistrarile unui tabel
- Sortare:
  - a indexului pe baza cheii de căutare
  - a fișierului/tabelei/relației stocate -> cheie de sortare = atribut care da ordonarea fisierului de date

- Un fișier de date 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: valorile cheii de căutare sunt stocate într-o anumită ordine
- Indecși hash: valorile cheii 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, codifica distribuția valorilor sub formă de matrice binară

Indecsi ordonati: fisiere secventiale

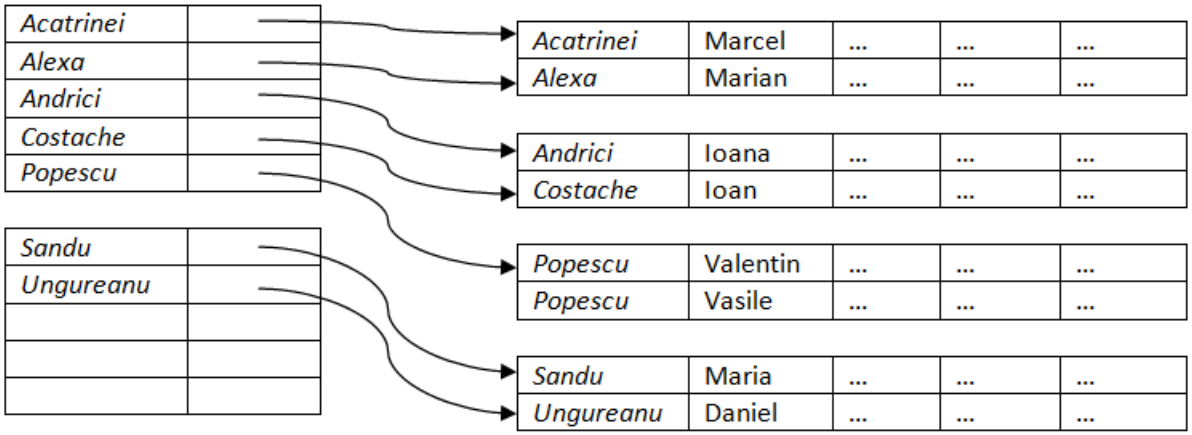
Indecşi secvenţiali (fisiere secventiale)

- 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/clustered 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/nonclustered): 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

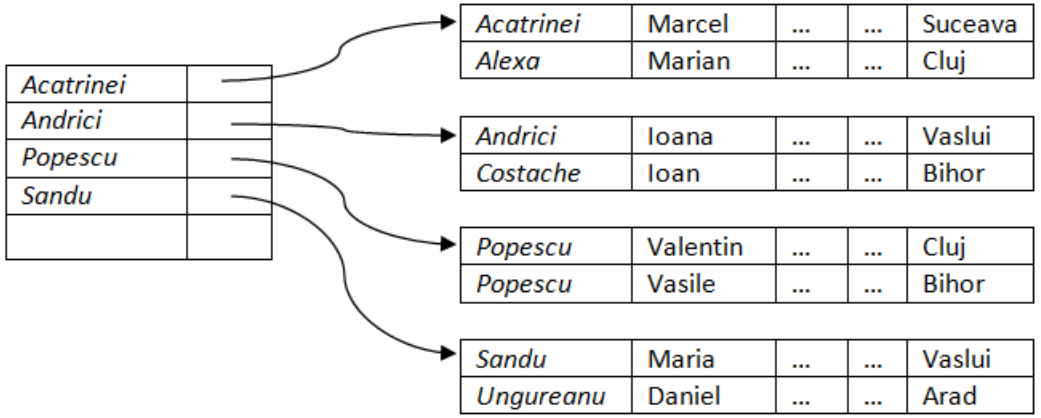
- Index dens: există înregistrări index pentru fiecare valoare a cheii de căutare în fișier/tabel/relație
- Dacă indexul e primar va păstra câte un singur pointer-doar la prima intrare cu valoarea respectivă
- Dacă indexul e secundar vor fi necesari mai mulți pointeri la o singură valoare a cheii de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



Index dens primar: cheia de cautare coincide cu cheia de sortare a fisierului de date

Fişiere index rare

- Index rar: conține intrări doar pentru unele valori a cheii de căutare
  - Aplicabil doar când înregistrările sunt ordonate secvențial după cheia de căutare
  - Balansul timp-spațiu
  - De obicei o intrare in index corespunde unui bloc din fisierul de date
- 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 fisier începând cu înregistrarea spre care indică înregistrarea index



Index rar: cheia de cautare coincide **obligatoriu** cu cheia de sortare a fisierului de date

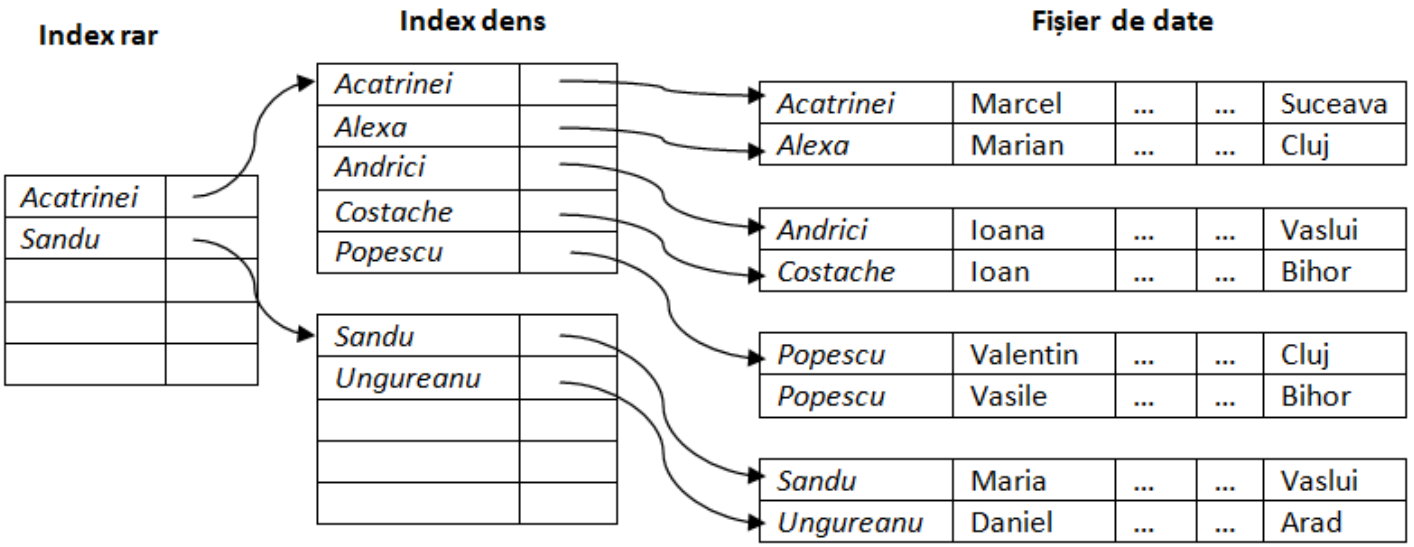
Indecşi multi-nivel

- 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

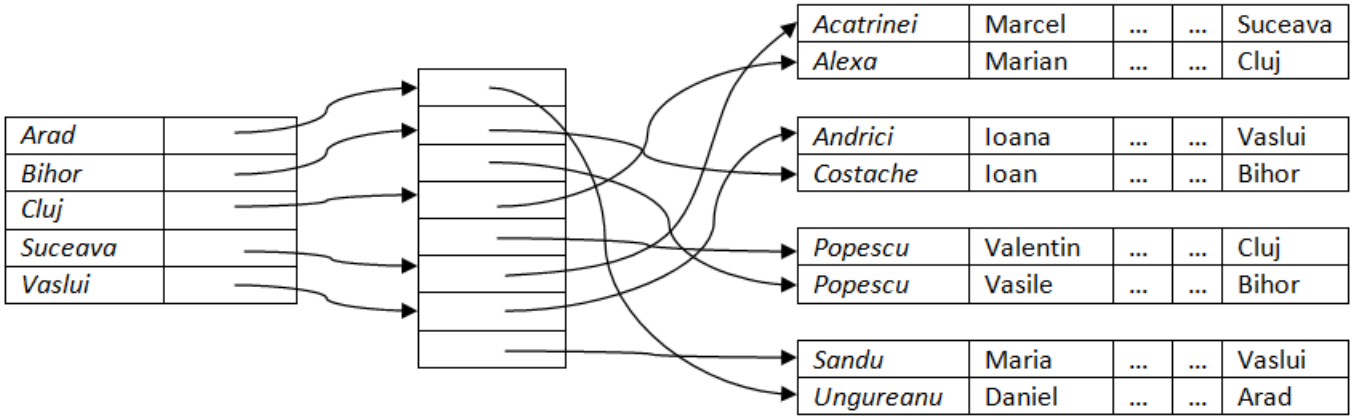
- Dacă și indexul extern este prea mare pentru a încăpea în memorie, se creează un index pe un nou nivel, etc…

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

Indecşi multi-nivel



Indecşi secundari



- *În relația Studenti sortată după nume, care sunt studentii ce locuiesc in Cluj?*
- Soluția: index secundar (dens!)
- Pentru a implementa relația de tip unu-la-multi dintre index și datele destinație se utilizează referințe la bucket-uri de pointeri

Actualizarea indecşilor secvențiali
Ştergere

- Stergerea inregistrarii din fisierul cu date atrage modificari asupra indexului
- 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 secvențiali
Inserare

- Este necesară localizarea valorii cheii de căutare ce apare în tuplul inserat
- Indecși denși: dacă valoarea nu apare în index se va insera
- Indecși rari: dacă indexul păstrează o intrare pentru fiecare bloc al fișierului 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 in fisierul cu date ordonat si in indexul secvential poate necesita crearea unor blocuri de exces -> degenerarea structurii secventiale

- Inserarea (și ștergerea) pentru indecși multi-nivel sunt extensii simple a algoritmilor uni-nivel prezentați

Indecsi ordonati: B+-arbori

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

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

  - 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 m care specifică numărul maxim de valori dintr-un nod (numarul maxim de pointeri /descendenti este m+1)
  - De obicei dimensiunea unui nod e cea a unui bloc
- Valorile cheii de căutare într-un nod sunt ordonate

$$K_1 < K_2 < K_3 < \ldots < K_m$$

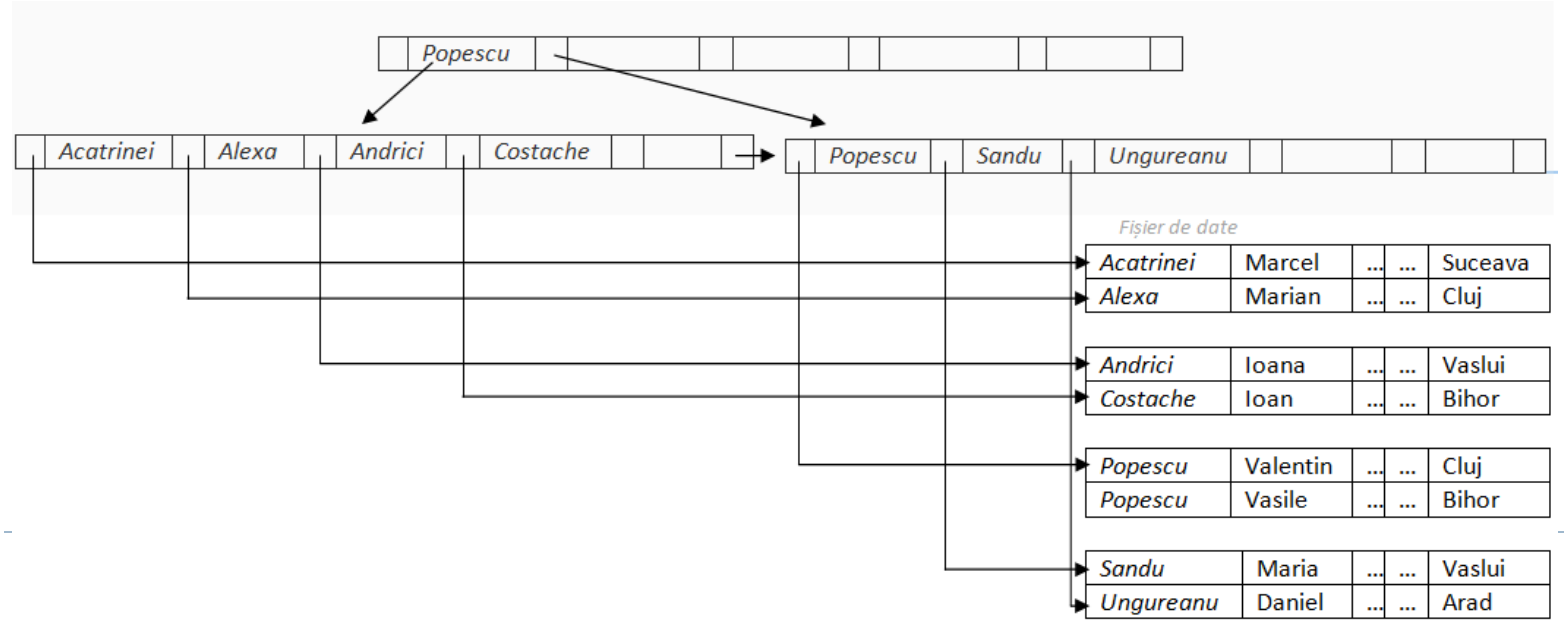
Noduri care nu sunt frunze în B^+-arbore

- 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}

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

Structura B+-arbore (2)

- Nodurile nu sunt complet ocupate pt. a evita reorganizari frecvente:
  - nodul radacina are cel putin doi si cel mult *m + 1 pointeri/descendenti* (corespunzator, cel putin una si cel mult *m valori*, ordonate crescator)
  - fiecare nod de pe un nivel intermediar are cel putin *⌈(m+1)/2⌉* si cel mult *m + 1* pointeri/descendenti (corespunzator, cel putin *⌈m/2⌉* si cel mult *m valori* ordonate crescator)
  - fiecare nod frunza are cel putin *⌈m/2⌉* si cel mult *m* valori; toti pointerii fac trimitere catre fisierul de date, cu exceptia ultimului pointer care face trimitere catre urmatorul nod frunza (cu valori mai mari).



B^+-arbore
Exemplu

- 1 bloc memorie = 1024 octeti
- Cheia de cautare = sir de max. 20 caractere (1 caracter=1octet)
- 1pointer = 8 octeti

- Numarul maxim de intrari in nod?
  - Cea mai mare valoare m cu proprietatea $20m + 8(m + 1) <= 1024$.
  - $m=36$
- Radacina: cel putin una, cel mult 36 valori
- Nod intermediar: cel putin 19, cel mult 37 pointeri
- Nod frunza: cel putin 18, cel mult 36 valori = pointeri catre fisierul de date

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
- Ultimul nivel = index secvential dens
- B^+-arborele conține un număr relativ mic de nivele
  - Cel mult $\lceil \log_{\lceil (m+1)/2 \rceil}(K) \rceil$ pentru k valori a cheii de căutare
    - Nivelul imediat următor rădăcinii: cel puțin 2 noduri
    - Urmatorul: $2*\lceil (m+1)/2 \rceil$ noduri
    - Următorul: cel puțin $2*\lceil (m+1)/2 \rceil * \lceil (m+1)/2 \rceil$
    - etc…
- Inserările și ștergerile se fac eficient, restructurarea indexului necesitând timp logaritmic

Interogări pe B^+-arbori
Algoritm

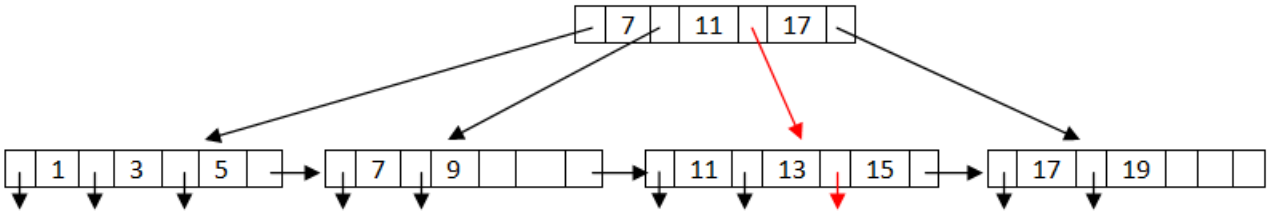
Determinarea tuturor înregistrărilor cu valoarea *k* a cheii de căutare

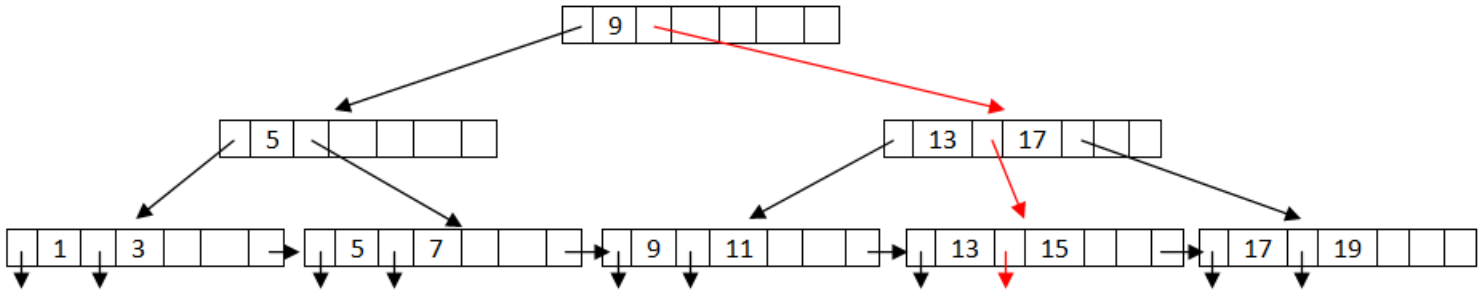
1. N=rădăcina
2. Repetă
   - Caută în N cea mai mică valoare a cheii de căutare >*k*
   - Dacă aceasta există și e egală cu K_i, atunci $N = P_i$
     - Altfel $N = P_n$ $(k \geq K_{n-1})$

   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*

Interogari

Cheia de cauare - stocheaza numerele impare intre 1-19
Intrarea 15?





Interogări pe B^+-arbori
Observații

- Ex:
  - Un nod este în general de aceeași dimensiune ca a unui bloc pe disc - 4Kbytes
  - Pp. m=100
  - Fiecare acces al unui nod poate necesita o localizare pe disc (<20 milisecunde)

- *Pentru 1 milion valori a cheii de căutare, câte noduri (blocuri pe disc) sunt accesate la o căutare în B^+-arbore? (4)*
- *Dar dacă se utilizează un index secvențial? (20)*

Actualizări în B^+-arbori
Inserarea

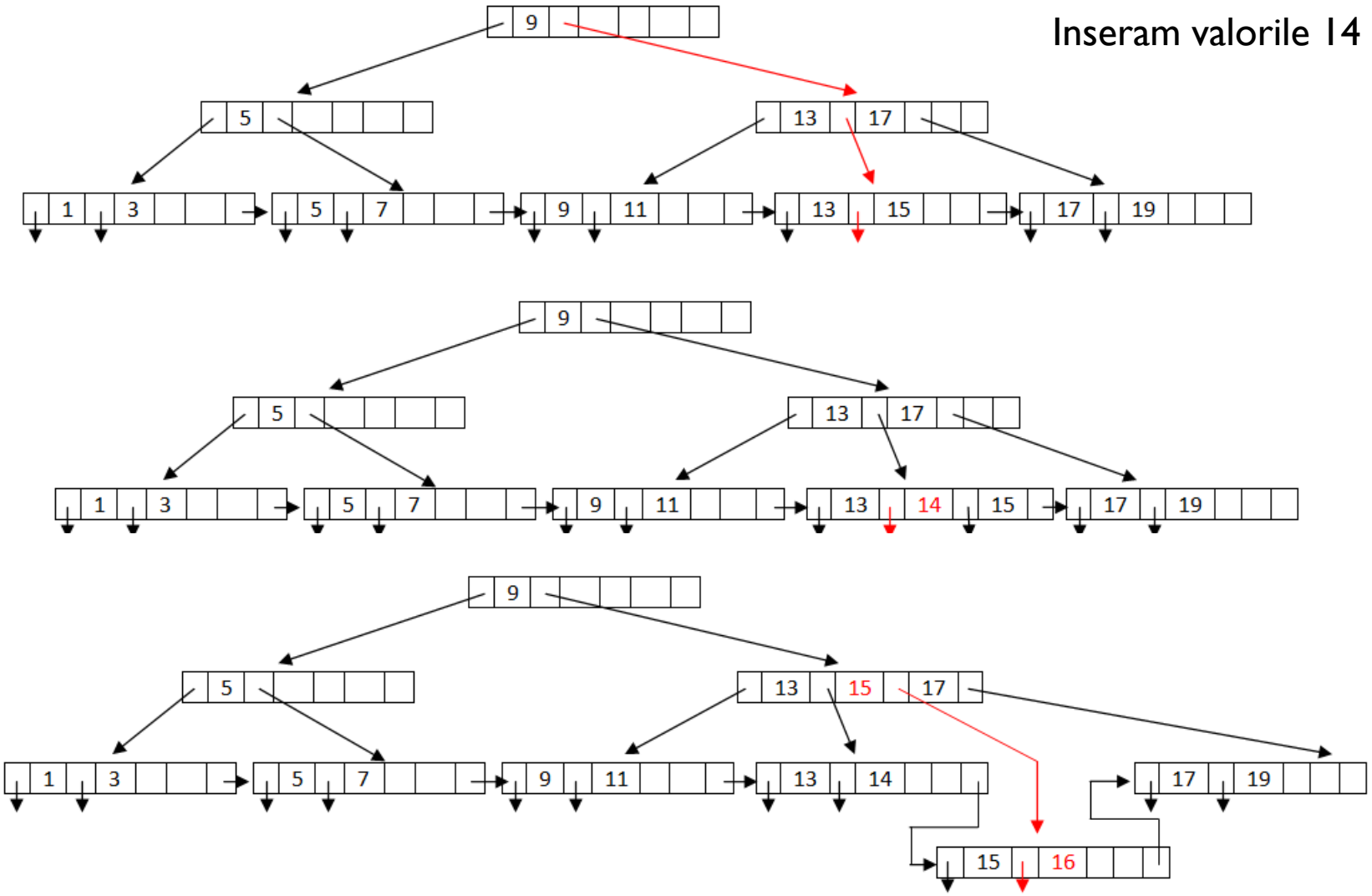
1. Determină nodul frunză în care va apărea valoarea cheii de căutare
2. Dacă valoarea e deja prezentă într-un nod frunză
    1. Adaugă înregistrarea în fișier/tabel/relație
    2. Adaugă un pointer în bucket
3. Dacă nu e prezentă valoarea
    1. Adaugă înregistrarea în fișier/tabel/relație
    2. Dacă 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 creează 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

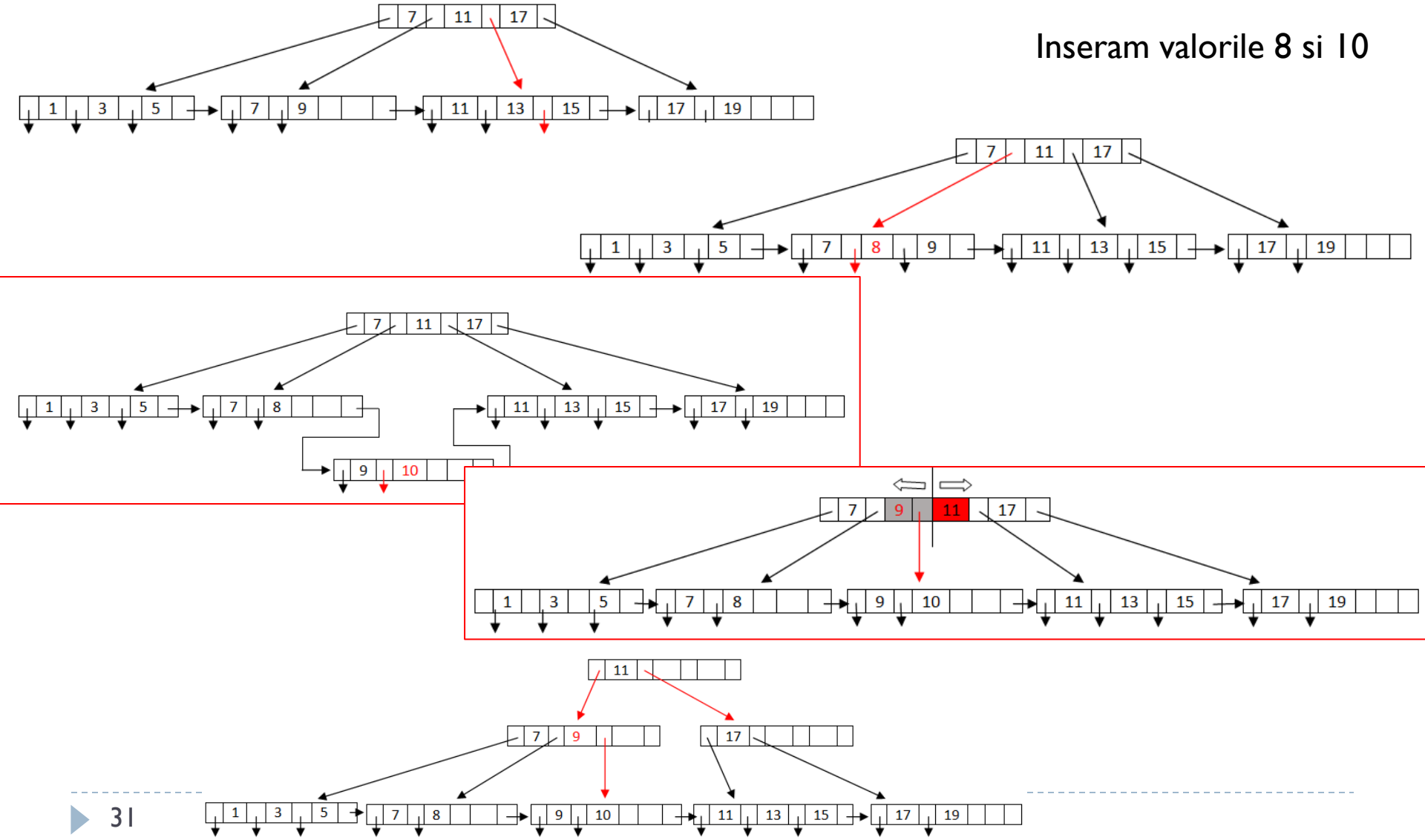
Actualizări în B$^+$-arbori
Inserarea: Exemplu

Inseram valorile 14 si 16



Actualizări în B+-arbori
Inserarea: Exemplu(2)

Inseram valorile 8 si 10

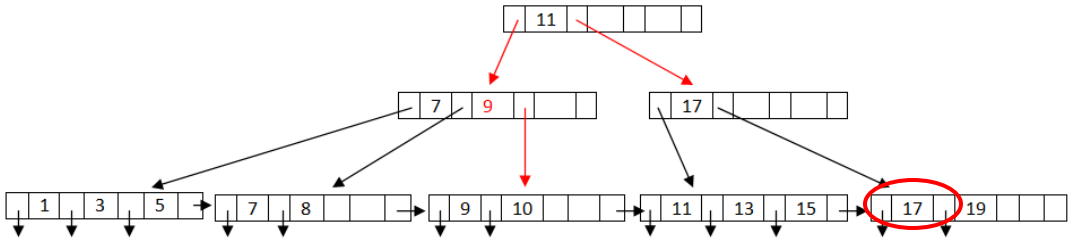


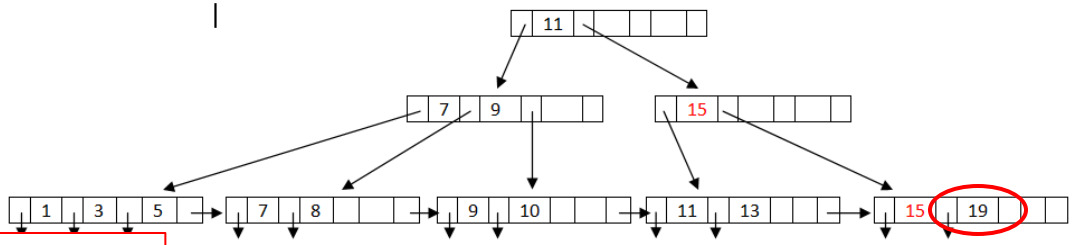
Actualizări în B^+-arbori
Ştergerea

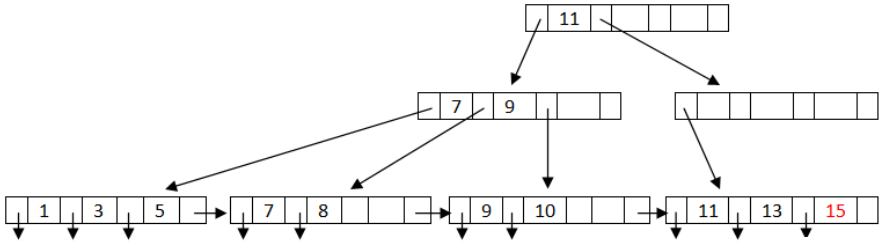
1. Şterge înregistrarea din fişier/tabel/relaţie
2. Dacă înregistrarea face parte dintr-un bucket, e ştearsă din acesta. Altfel (sau dacă bucketul devine gol) şterge din nodul frunză perechea (pointer, valoare cheie)
3. Dacă nodul va avea prea puţine intrări în urma ştergerii şi ele încap într-un nod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

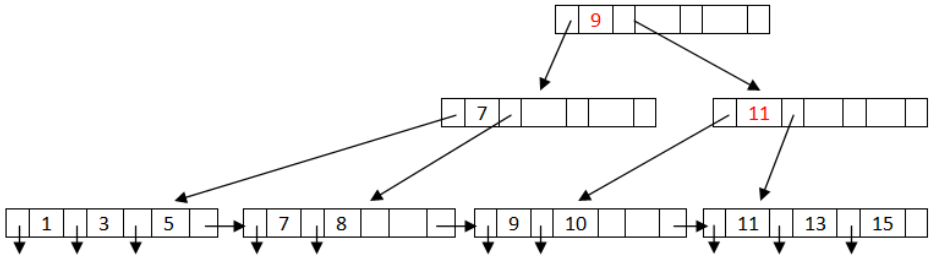
Actualizări în B⁺-arbori
Ştergerea: Exemplu (1)

Stergem valorile 17 si 19



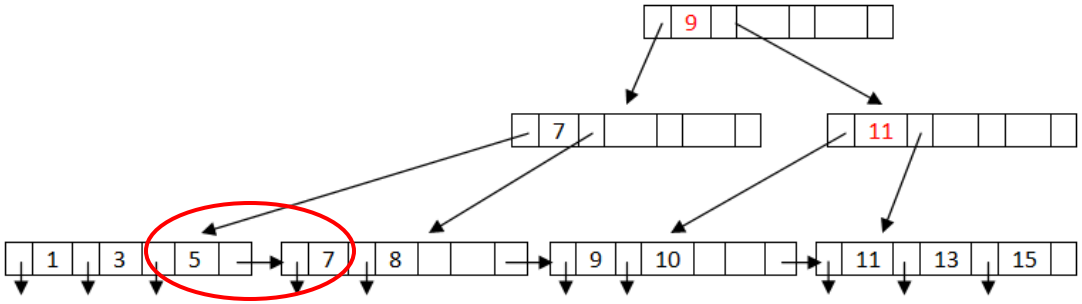


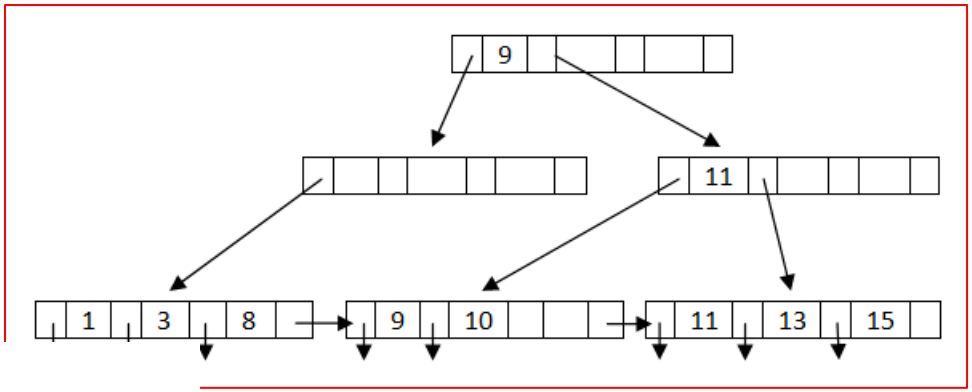


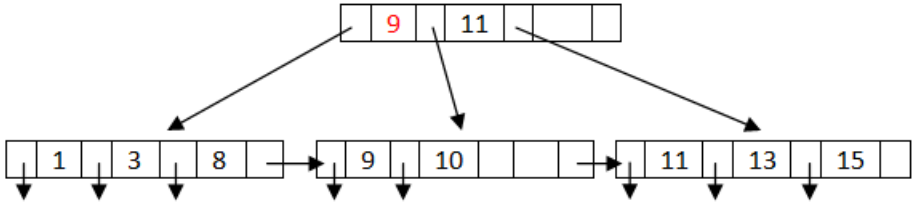


Actualizări în B$^+$-arbori
Ştergerea: Exemplu (2)

Stergem valorile 5 si 7







B^+-arbori
Eficiența

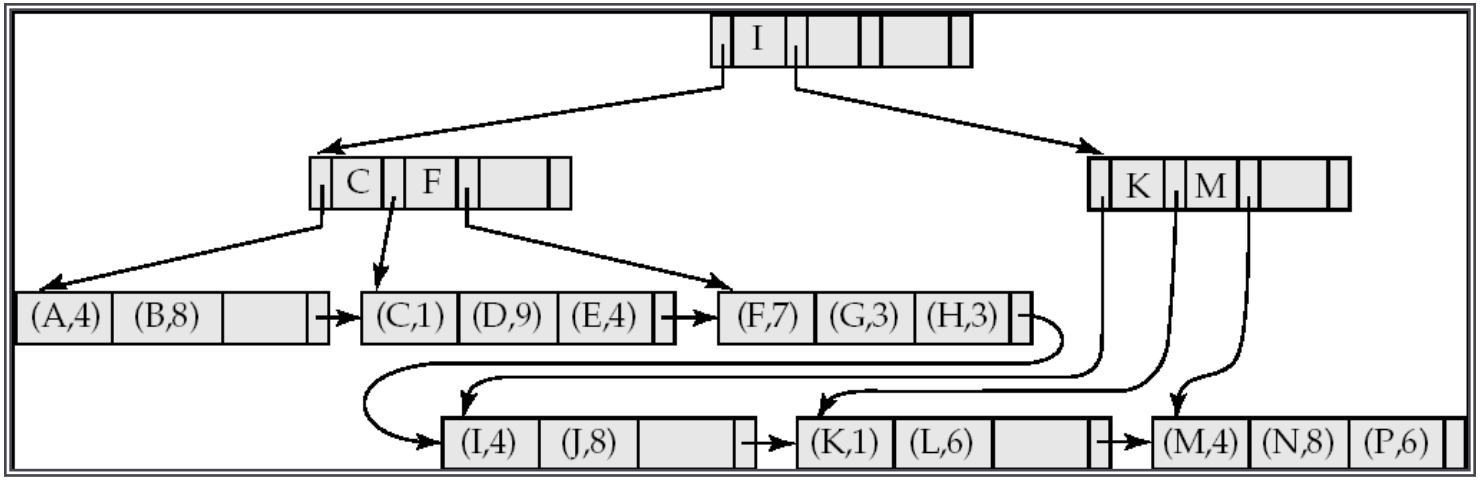
- Căutare: cel mult $\lceil \log_{\lceil (m+1)/2 \rceil}(K) \rceil$ blocuri transferate
  - Datorită înlănțuirii nodurilor frunză sunt eficienți și pt. Căutări în interval

- Inserare, stergere: cel mult $2 \lceil \log_{\lceil (m+1)/2 \rceil}(K) \rceil$ blocuri transferate

Organizarea fişierelor B+-arbore

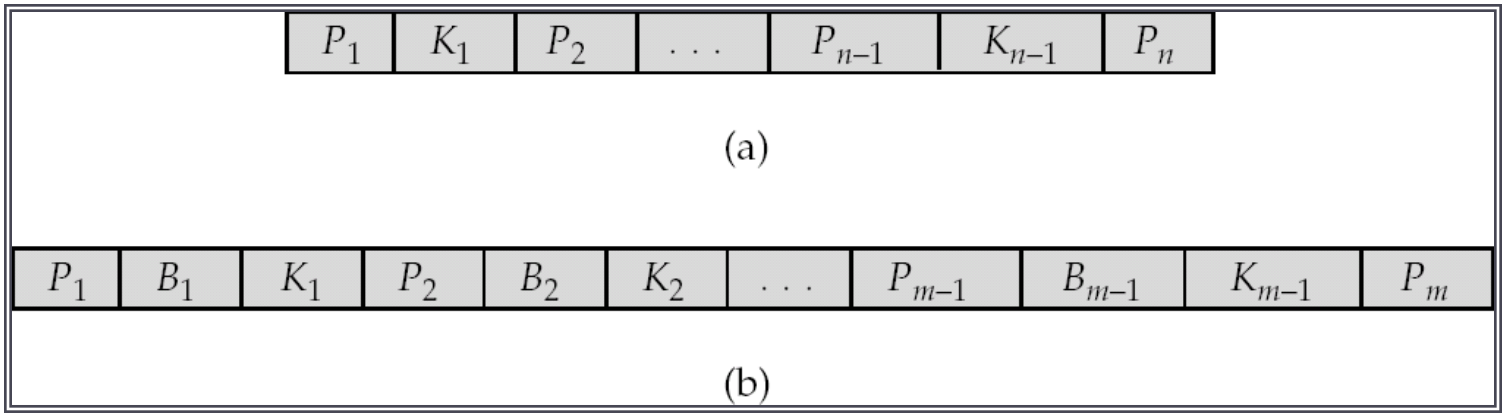
- B+-arborii pot fi utilizați direct pentru organizarea fișierului și nu doar pentru indexare
  - Nodurile frunză stochează înregistrări și nu pointeri
  - Pentru a îmbunătăți utilizarea spațiului sunt implicați mai mulți vecini în redistribuire pentru a evita divizarea sau unirea

$$\lfloor 2n/3 \rfloor$$



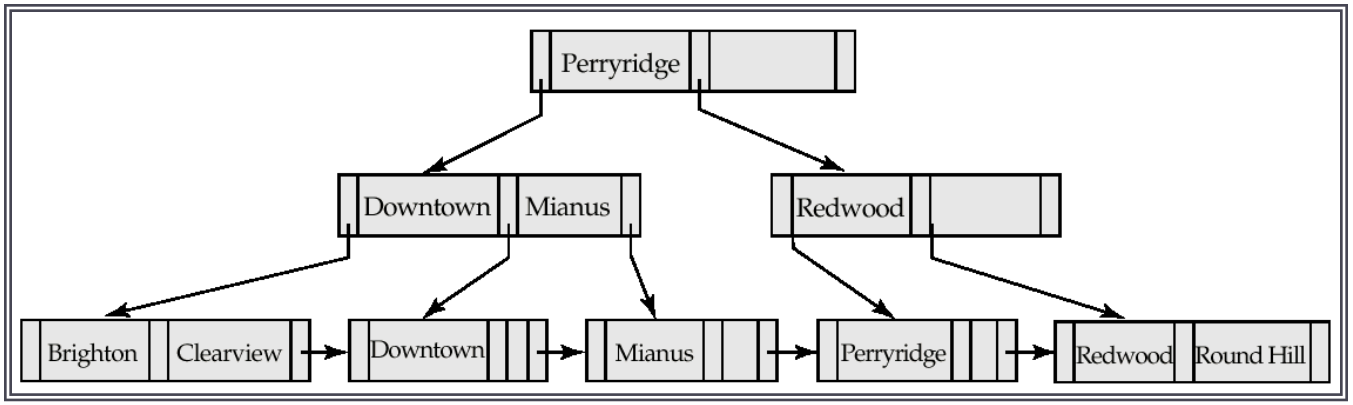
Indecşi B-arbore

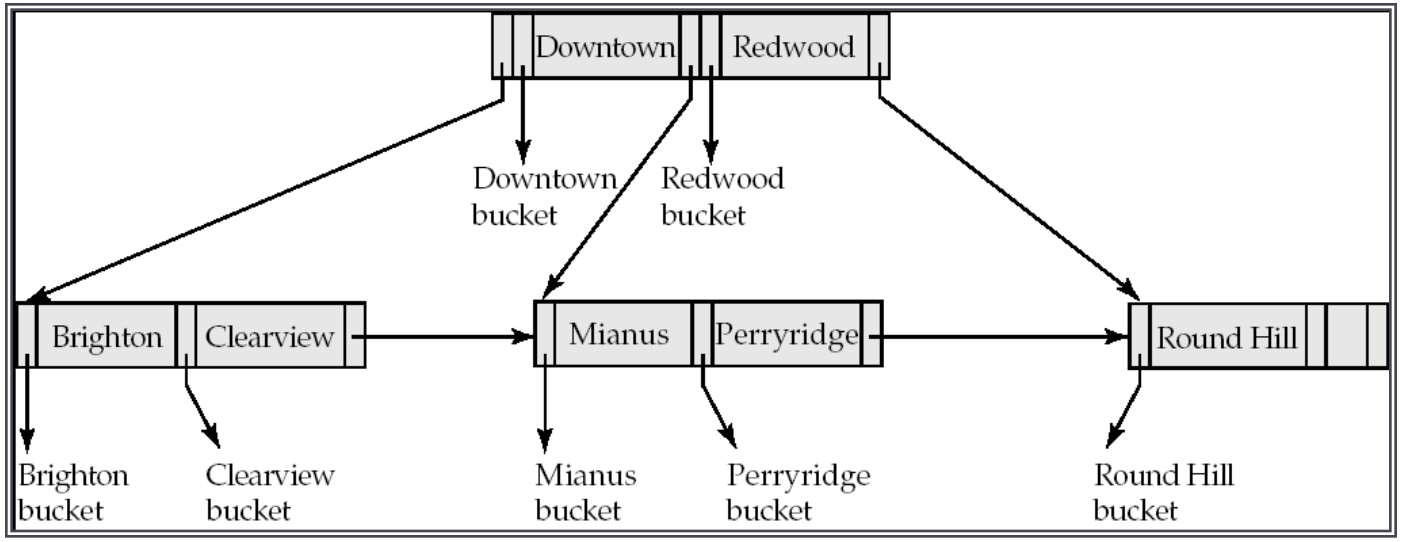
- Asemănători B^+-arborilor însă permit o singură apariție a valorilor cheilor de căutare
- Cheile de căutare în nodurile care nu sunt frunză nu mai apar nicăieri în arbore ceea ce necesită introducerea unui pointer adițional



- 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

Indecsi ordonati: indecsi multi-cheie

Acces multi-cheie

- Pot fi utilizați mai mulți indecși la o interogare

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

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

Indecşi multi-cheie

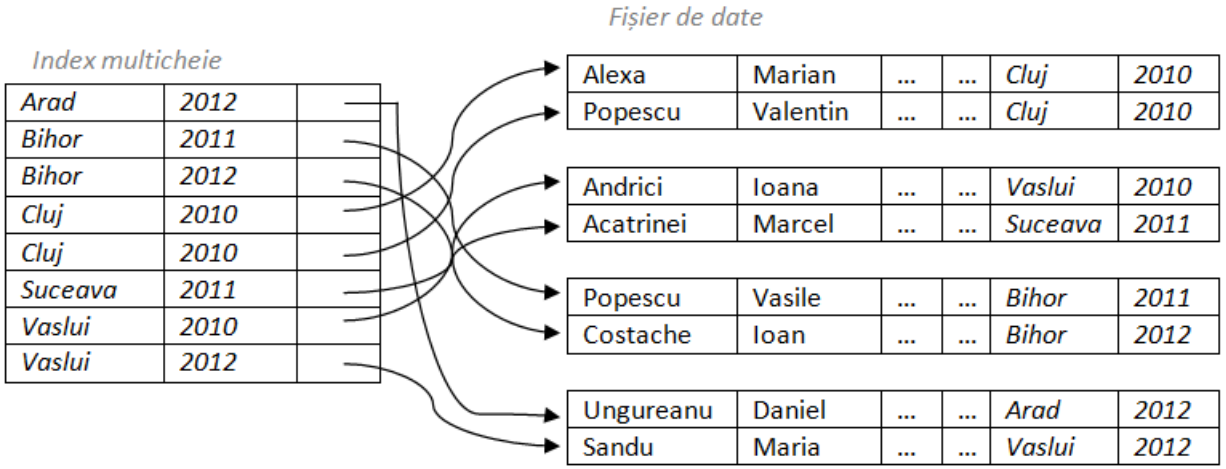
- Cheile de căutare compuse sunt chei ce conțin mai mult de un atribut
- Ordinea lexicografică: $(a_1, a_2) < (b_1, b_2)$ dacă
  - $a_1 < b_1$ sau
  - $a_1 = b_1$ și $a_2 < b_2$

Ex. *(judet, an)*

Pot fi rezolvate eficient condițiile de mai jos?

a) where judet = 'Bihor' AND an > 2010

b) where judet > 'Bihor' AND an = 2010



Eficiența

- Ordinea atributelor într-un index multi-cheie este foarte importantă
- Eficiența depinde de selectivitatea atributelor

k-d arbori

- Generalizare a arborelui binar de cautare:
  - k= nr. de atribute ce formeaza cheia de cautare
  - Fiecare nivel corespunde unui atribut din cele k
  - Succesiunea nivelelor reprezintă iterări peste mulțimea celor k atribute
  - Pentru a obține arbori echilibrați, la nivelul fiecărui nivel în noduri este pusă valoarea mediană

Organizarea hash/Indecsi hash

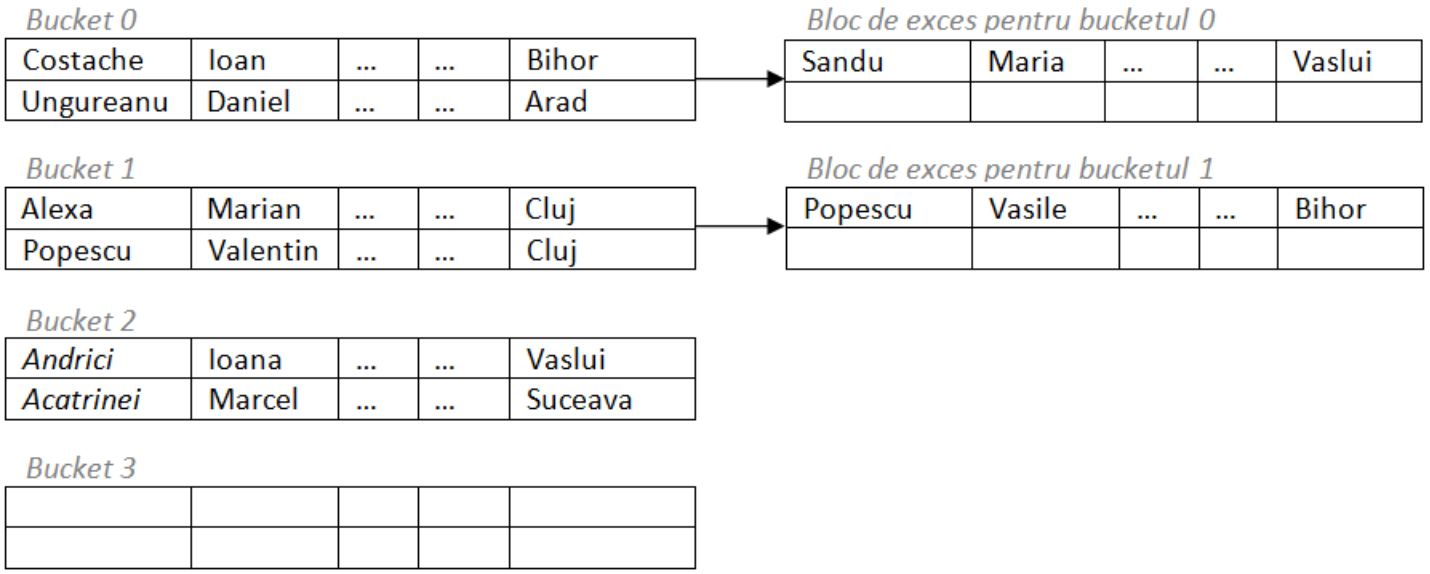
Hashing

- În organizarea de tip hash a fișierului/tabelului/relației înregistrările sunt grupate în bucketuri care pot fi localizate pe baza valorilor cheii de căutare (un bucket ia dimensiunea unui bloc)
- Funcția hash (de dispersie) *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 ale cheii de căutare pot fi mapate la același bucket
  - Căutare secvențială în bucket

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

Organizarea de tip hash
Exemplu



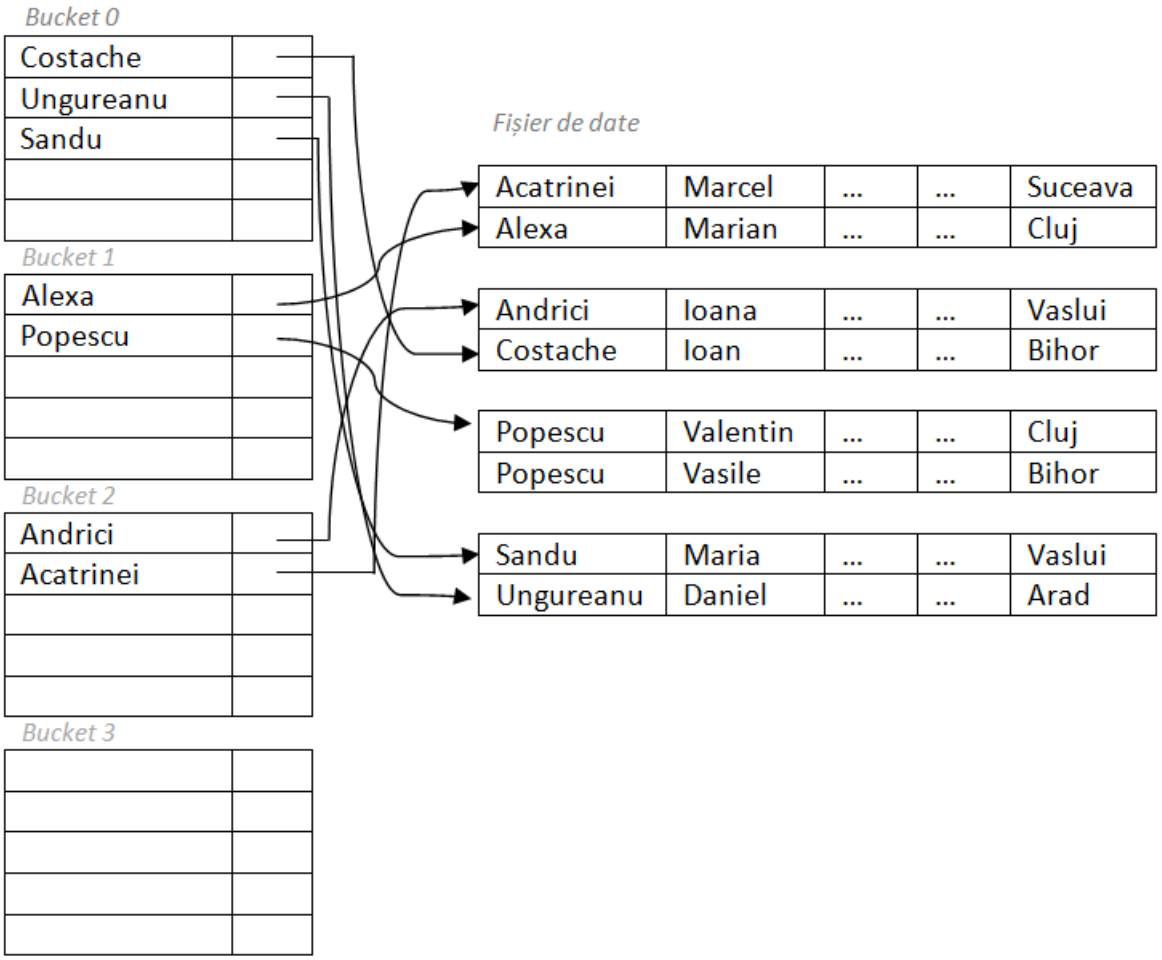
Organizarea de tip hash utilizând *nume* drept cheie:

Funcția hash: h:Dom(nume)->{*0,1,2,3*} - suma reprezentărilor binare modulo 4

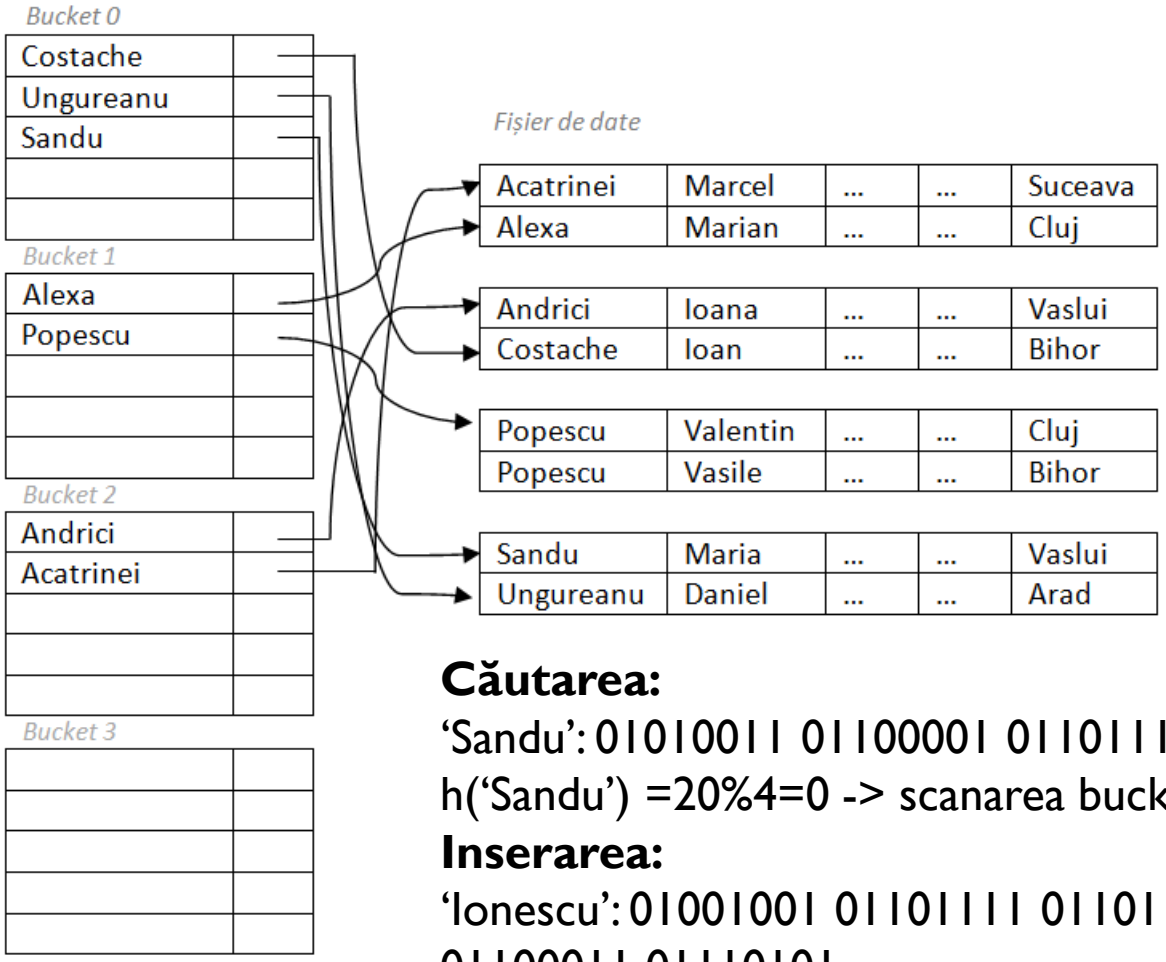
'Acatrinei':
'1000001 1100011 1100001 1110100 1110010 1101001 1101110 1100101 1101001'
h('Acatrinei') = 34%4 = 2.

Indecşi hash

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



Indecși hash
Operații



Căutarea:

'Sandu': 01010011 01100001 01101110 01100100 01110101

h('Sandu') =20%4=0 -> scanarea bucketului 0

Inserarea:

'Ionescu': 01001001 01101111 01101110 01100101 01110011 01100011 01110101

H(Ionescu)=32%4=0 -> inseram in fisierul de date si apoi in bucketul 0 in index

Ștergerea: calcul hash, scanare bucket, stergere

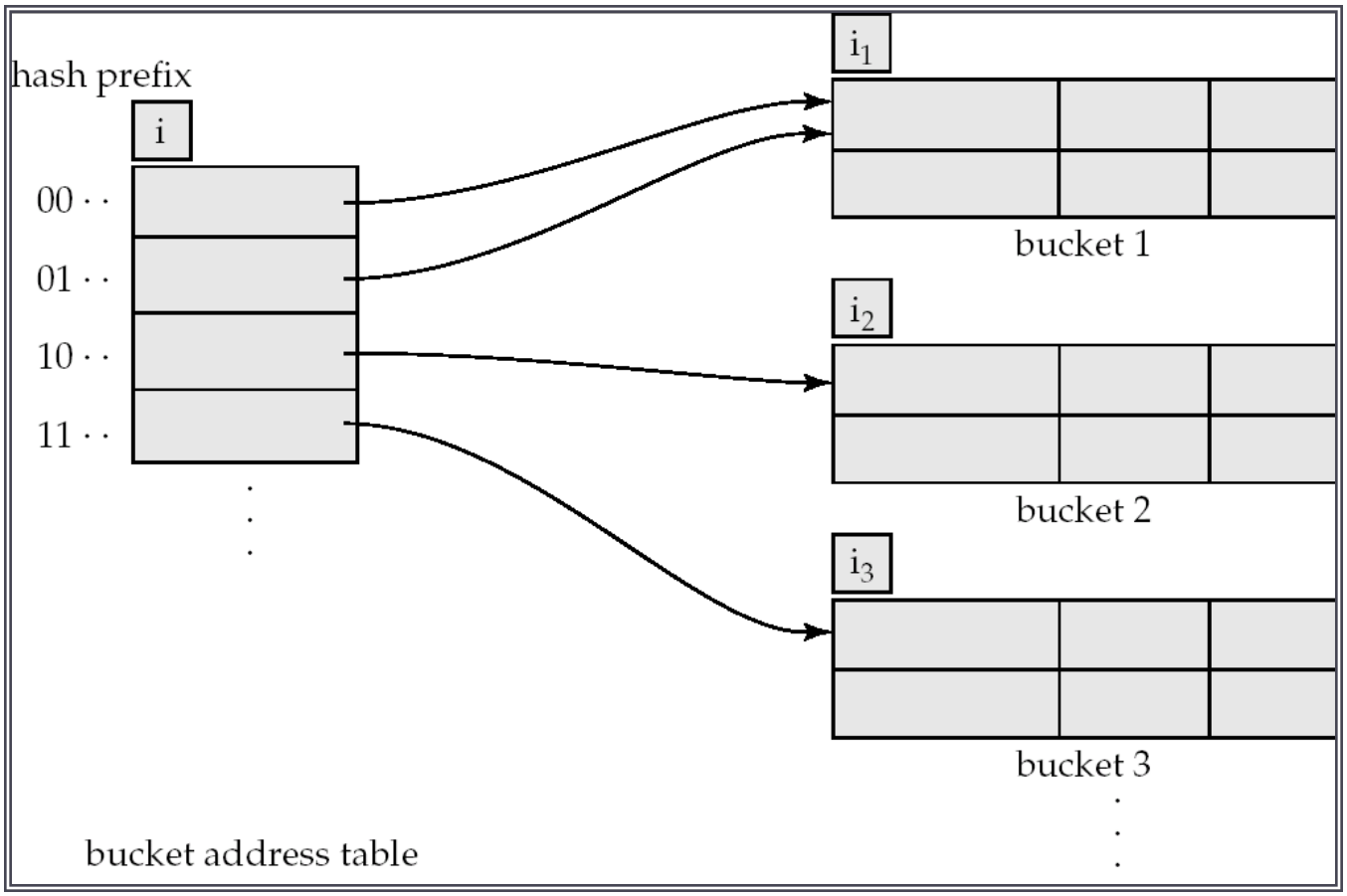
Indecși hash
Performanța

- Pentru căutări punctuale, în absența coliziunilor, localizarea unei valori necesită citirea unui singur bloc
- Pentru căutări în interval indecșii hash nu sunt eficienti, deoarece trebuie calculat hashul fiecărei valori posibile

Hash dinamic

- Funcția h mapează valori ale 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) al 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:

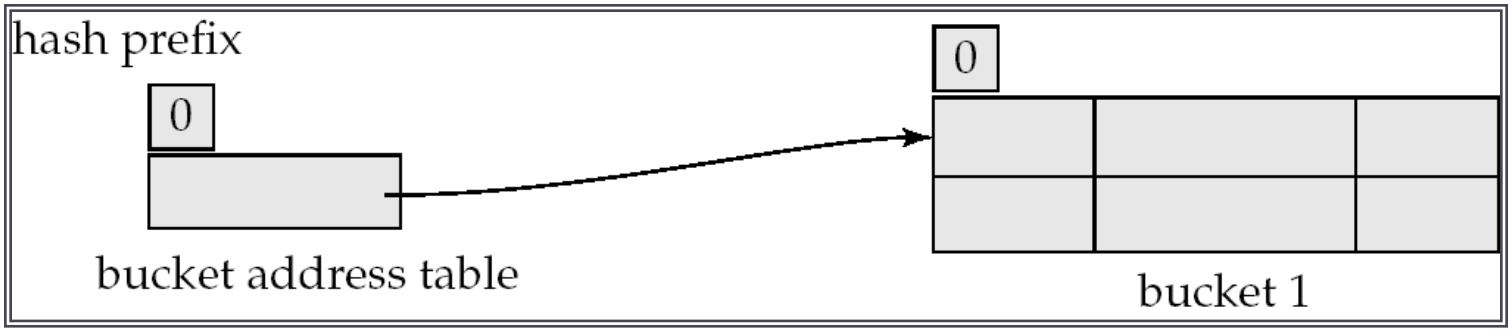
- Dacă $i > i_j$
  1. Se alocă un nou bucket z și $i_j = i_z = (i_j + 1)$
  2. Se actualizează a doua jumătate a tabelei de adrese a bucketurilor pentru a indica spre z
  3. Se scot înregistrările din j și sunt reinserate în j sau z
  4. Se recalculează adresa bucketului pentru K_j și se inserează
- 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

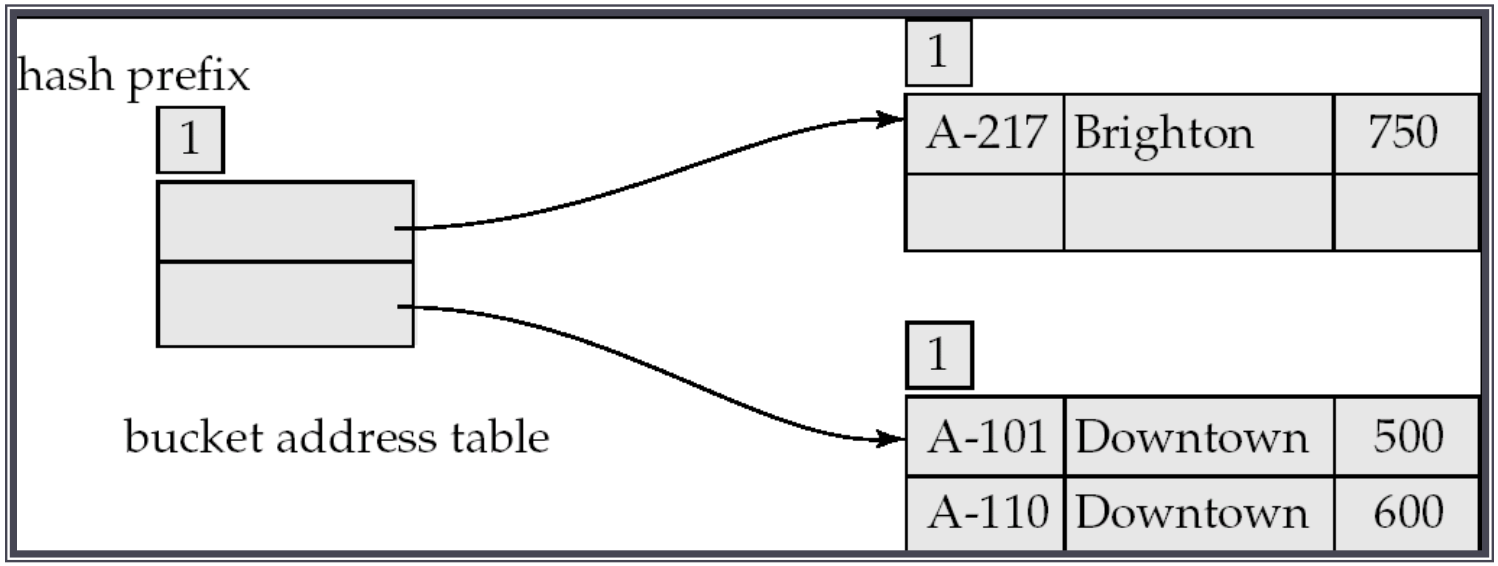
branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111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



structura hash inițială, dimensiune bucket = 2

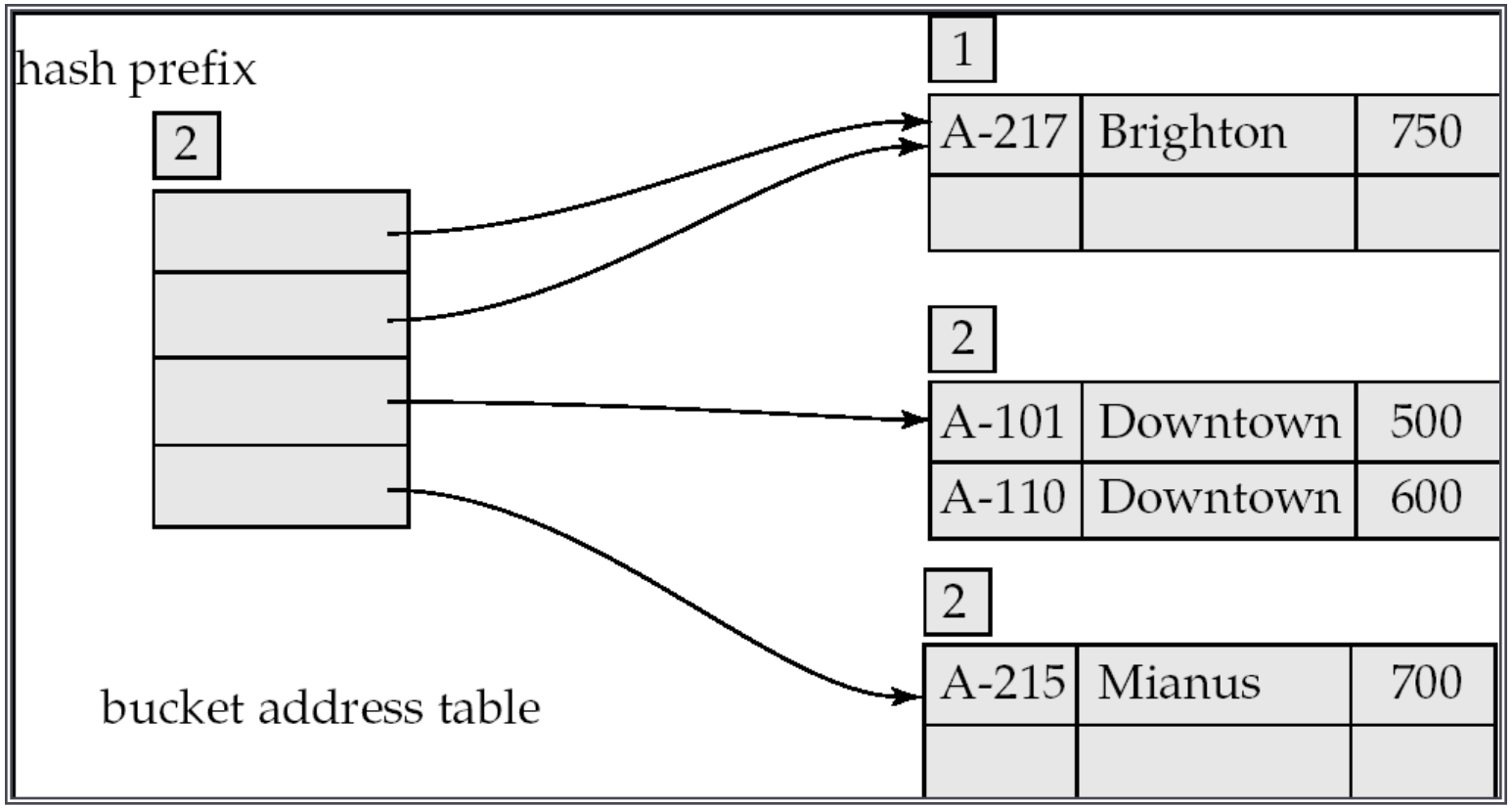
Hash extensibil
Exemplu

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



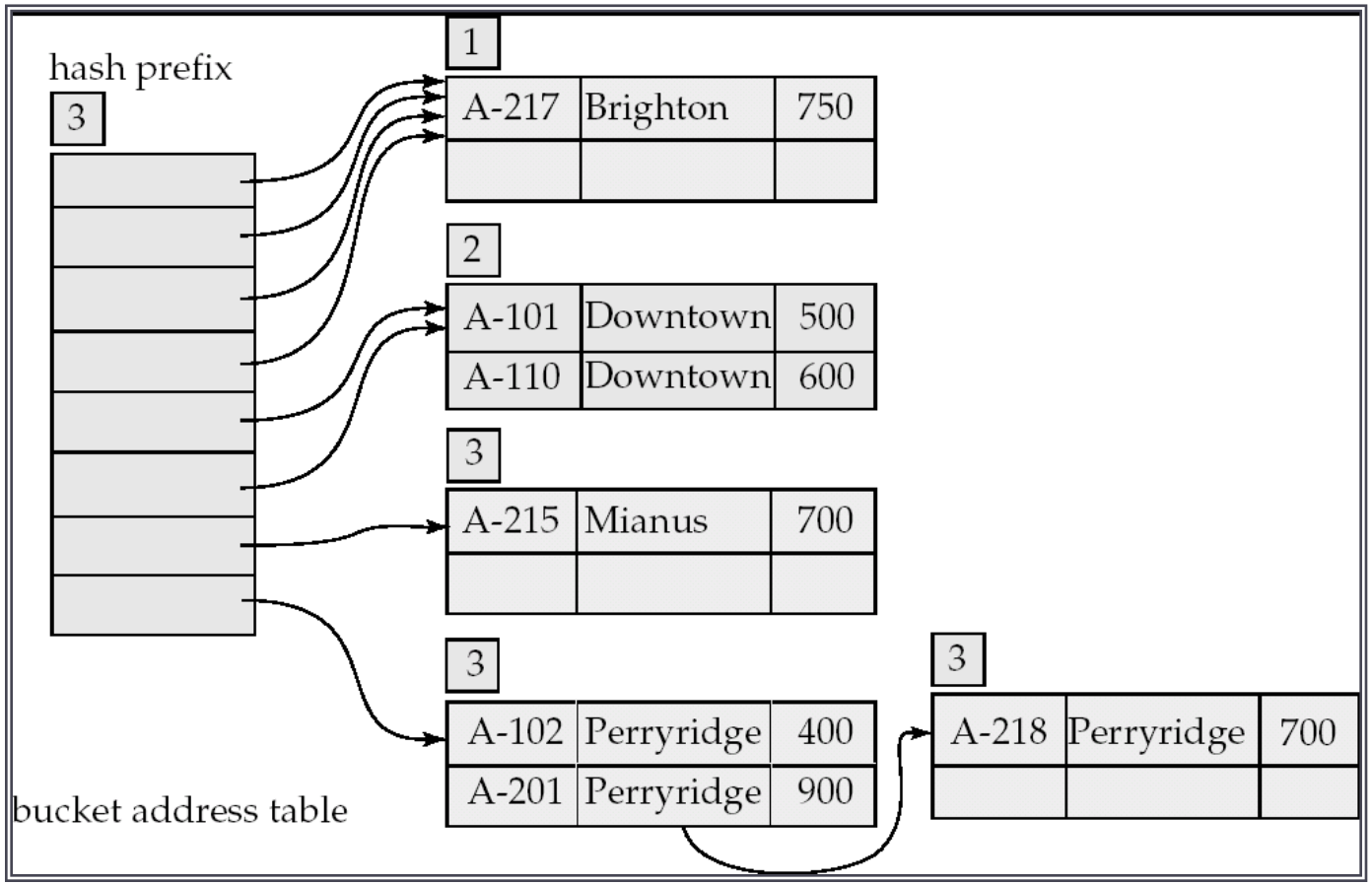
Hash extensibil
Exemplu

- După inserarea înregistrării Mianus



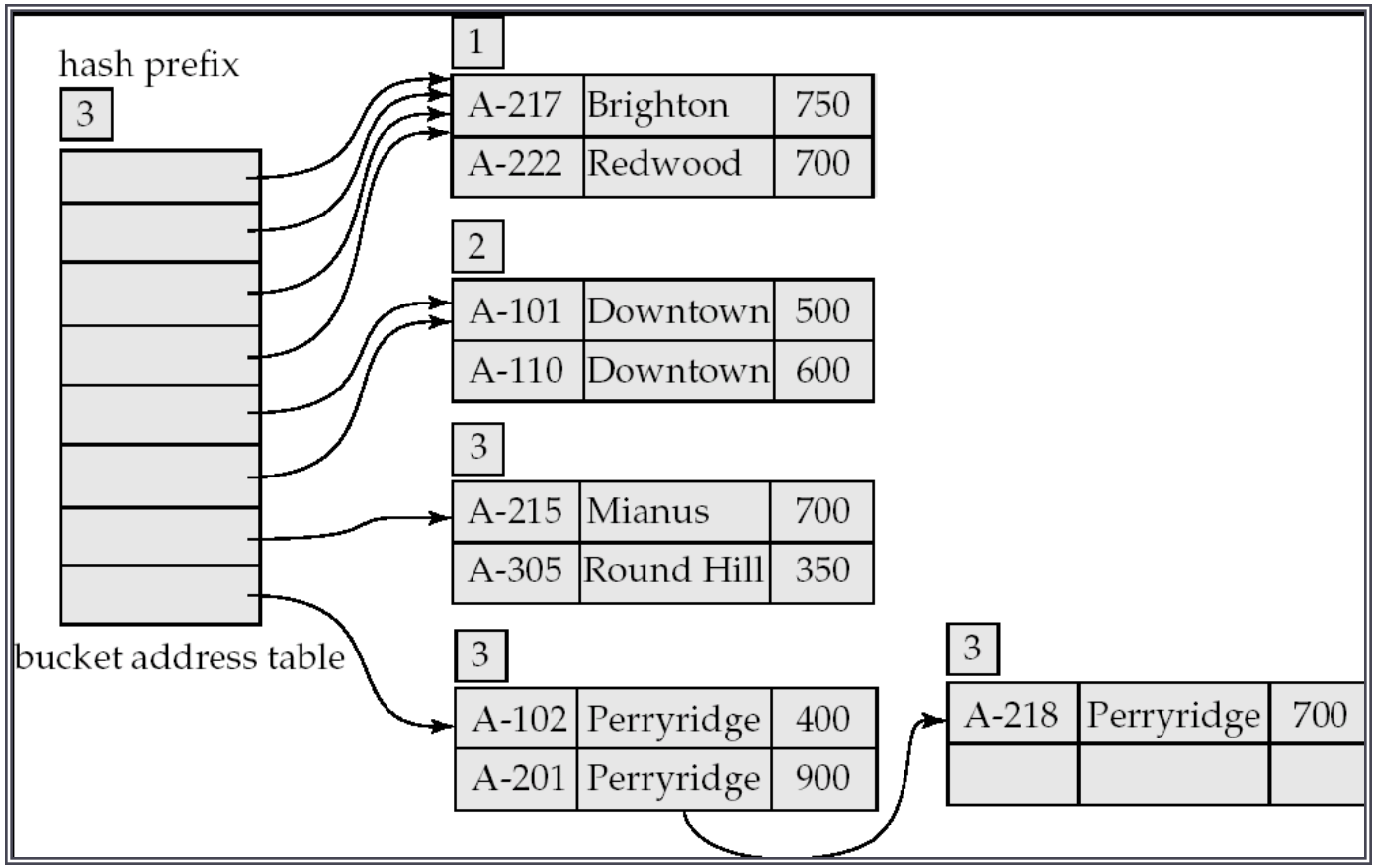
Hash extensibil
Exemplu

- După inserarea a trei înregistrări Perryridge



Hash extensibil
Exemplu

- După inserarea înregistrărilor Redwood și Round Hill



Hash 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

Indecsi bitmap

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
- Tuplele 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

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

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

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

Indecşi bitmap
Observații

- Interogările sunt rezolvate utilizând 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)

  - 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)

Definirea indecsilor in SQL

Definirea indecşilor în SQL

- Standardul nu reglementeaza, dar in practica SGBD-urile adera la aceeasi sintaxa
- 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
- Majoritatea SGBD-urilor creează implicit indecsi la specificarea constrangerii *unique*
- Uneori sunt generati indecsi la specificarea constrangerilor referentiale

Indexarea în Oracle

- Oracle suportă B^+-arbori implicit la crearea indexului cu comanda SQL (in documentatia Oracle apar ca B-arbori dar corespund in teorie B^+-arborilor)
- 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

Indexarea în Oracle
Considerente

- Crearea indecsilor e recomandată dupa incarcarea datelor – oricand e posibil
- Indexarea coloanelor potrivite:
  - Coloanele au valori (in proportie mare) unice
  - Dacă selecția filtrează un număr mic de tuple dintr-un tabel mare (selectivitate ridicata ~ 15%)
  - Coloanele sunt utilizate in join
  - Coloane cu valori distincte putine -> structura bitmap
  - Range query (filtrare in interval) -> B+trees
  - Interogari punctuale frecvente: organizare hash a fisierului
  - Selctie cu conditii petse functii -> indecsi definiti peste functii

Bibliografie

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