Úvod do databázových systémov

http://www.dcs.fmph.uniba.sk/~plachetk

/TEACHING/DB1

Tomáš Plachetka

Fakulta matematiky, fyziky a informatiky,
Univerzita Komenského, Bratislava

Zima 2022–2023

Trvácne médiá

- Dáta sú uložené na **trvácnych (externých) médiách**, ktoré možno rozdeliť do dvoch kategórií:
  - médiá **so sekvenčným prístupom** (napr. magnetické pásky), práca s nimi pripomína prácu s linkovaným zoznamom
  - médiá **s ľubovoľným prístupom** (napr. magnetické disky), práca s nimi pripomína prácu s poľom
- Dáta v tabuľkách sú organizované po riadkoch (records), dáta na externých médiách sú organizované **v blokoch**
  - predpokladá sa, že **jeden blok obsahuje viacero záznamov**
  - pre jednoduchosť predpokladáme, že dáta v operačnej pamäti (RAM) sú organizované po rovnakých blokoch
  - drivery médií abstrahujú od konkrétnych fyzických detailov (napr. cylindre, sektory, stopy), ponúkajú adresáciu blokov a čítanie/písanie bloku z/do operačnej pamäte (operácie input/output v Cache Manager)

Fyzická optimalizácia dotazov: fyzická algebra

Literatúra:

• H. Garcia-Molina, J.D. Ullman, J. Widom: Database System Implementation, Prentice Hall, 2000 (Chapters 6–7)

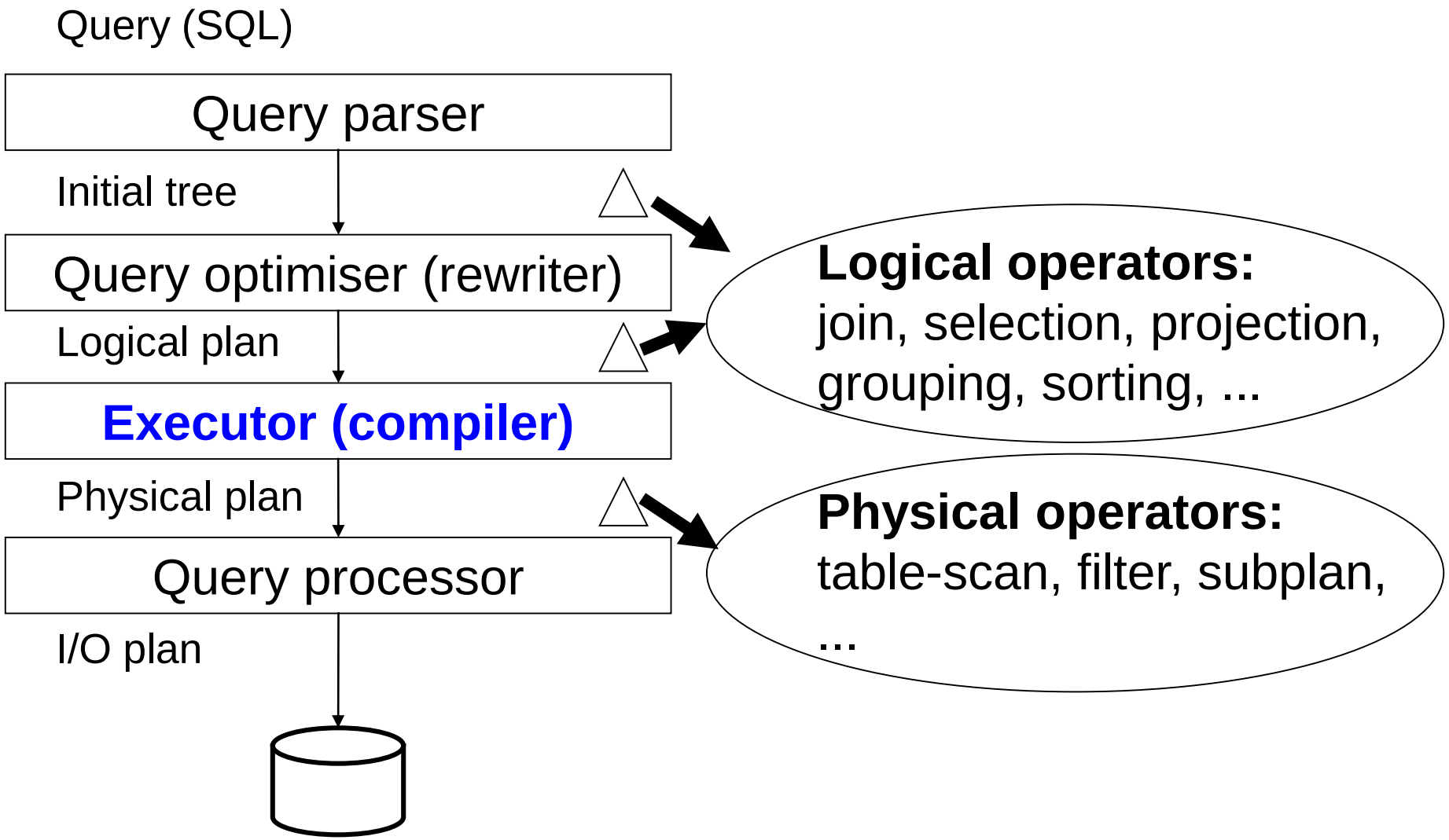
• R. Treat: Explaining Explain

• Postgres manual, http://www.postgresql.org/docs/current/static/performance-tips.html

• Y.E. Ioannidis: Query Optimization, http://www-db.stanford.edu/~widom/cs346/ioannidis.pdf

• S. Chaudhuri: An Overview of Query Optimization in Relational Systems, http://www-db.stanford.edu/~widom/cs346/chaudhuri.pdf

Fyzická optimalizácia dotazov: fyzická algebra



Fyzická optimalizácia dotazov: fyzická algebra

Možnosti implementácie fyzických operátorov:

- Materializácia: medzivýsledky operácií sa reprezentujú tabuľkou. Nevýhody: medzivýsledky môžu byť obrovské, medzivýsledky treba ukladať a čítať viackrát ako raz, ...

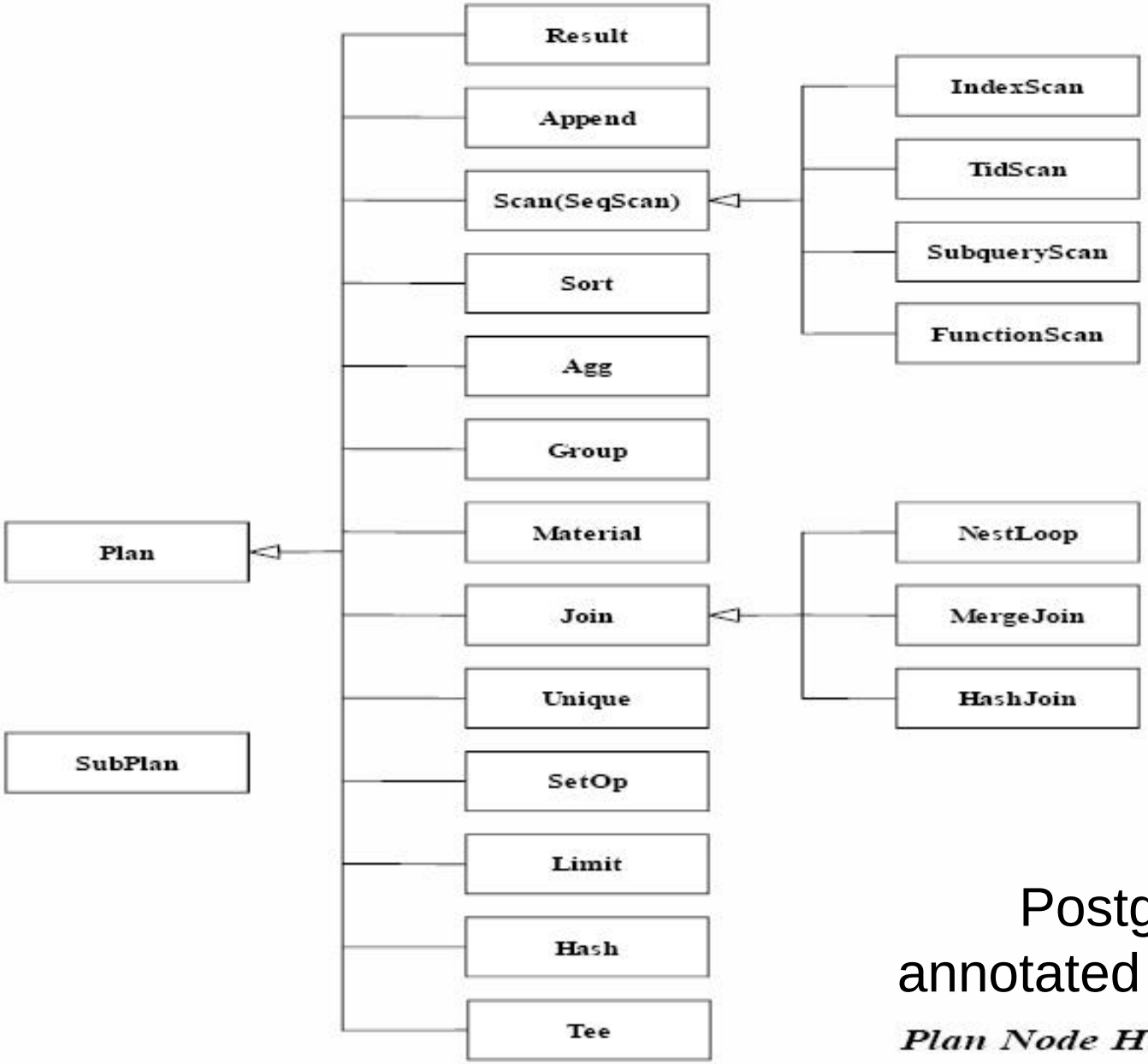
- **Iterátory (pipelining):** medzivýsledky sa reprezentujú vhodným previazaním pointerov na záznamy (riadky relácií). V jednej iterácii (next) sa vypočíta jeden riadok výslednej relácie, výsledok iterátora môže byť vstupom do ďalšieho iterátora

Metódy iterátorov: **open**, **next**, **close** (objektová filozofia)

Postgres, ako aj prakticky všetky súčasné DB systémy, používa iterátory na implementáciu fyzických operátorov. Materializácia je implementovaná ako špeciálny iterátor

Cieľom návrhu a implementácie fyzických operátorov je minimalizovať počet diskových prenosov, t.j. počet input/output operácií

Fyzická optimalizácia dotazov: fyzická algebra



PostgreSQL,
annotated source code

Plan Node Hierarchy Chart

Fyzická optimalizácia dotazov: zložitosť fyz. operátorov

Miery zložitosti relácií (relácia je „bag“, nie set)
B(R): počet blokov R
T(R): počet tuples (záznamov) R, vrátane duplikátov
V(R, a_1, a_2, ..., a_N): počet rôznych tuples $\pi_{a_1, a_2, ..., a_N}(R)$

Miery zložitosti fyzických operátorov
počet I/O prenosov, **čas**

POSTGRESQL Cost Parameters

Parameter	Description	Default	vs. page read
page read	Cost to read fetch one page of data, by definition	1.00	-
cpu_tuple_cost	Cost of typical CPU time to process a tuple	0.01	100x quicker
cpu_index_tuple_cost	Cost of typical CPU time to process an index tuple	0.001	1000x quicker
cpu_operator_cost	Cost of CPU time to process a typical WHERE operator	0.0025	400x quicker
random_page_cost	Cost of a non-sequential page fetch	4	4x slower
effective_cache_size	Amount of cache = likelihood of finding a page in cache	1000	N/A

Fyzická optimalizácia dotazov: zložitosť fyz. operátorov

Iterator **SeqScan(R)**

```
open(R) {
  R.open();
}
close(R) {
  R.close();
}
next(R) {
  return R.next();
}
```

Seq Scan Operator : Example

```
oscon=# explain select oid from pg_proc;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on pg_proc  (cost=0.00..87.47
rows=1747 width=4)
```

- Most basic, simply scans from start to end
- Checks each row for any conditionals as it goes
- Large tables prefer index scans
- Cost (no startup cost), rows (tuples), width (oid)

Miera zložitosti fyzických operátorov: čas vykonávania

V konfigurácii Postgres je konštanta 0.01, ktorá hovorí že RAM je 100x rýchlejšia ako disk. Postgres meria zložitosť v počte I/O prenosov, ale zahŕňa do nej aj čas výpočtov v RAM. V tomto príklade B(R)=70 a T(R)=1747, tak odhad zložitosti seq_scan(R) je 70+1747*0.01=87.47 I/O prenosov. (To 0.00 je odhad „startup cost“, t.j. zložitosť „open“. V tomto prípade nestojí startup nič.)

Fyzická optimalizácia dotazov: zložitosť fyz. operátorov

Iterator **SeqScan(R)**

```
open(R) {
  R.open();
}
close(R) {
  R.close();
}
next(R) {
  return R.next().filter();
}
```

```
oscon=# explain select oid from pg_proc where oid<>0;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on pg_proc  (cost=0.00..91.84 rows=1746 width=4)
   Filter: (oid <> 0::oid)
```

IOcost+CPUcost=B(R)+T(R)*(0.01+0.0025)=91.835
(0.0025 je cena filtrovacej podmienky)

```
oscon=# explain select oid from pg_proc where oid<>0
and oid<100;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on pg_proc  (cost=0.00..96.20 rows=582
width=4)
   Filter: ((oid <> 0::oid) AND (oid < 100::oid))
```

IOcost+CPUcost=B(R)+T(R)*(0.01+2*0.0025)=96.205
(cenu za testovanie podmienky platíme dvakrát)

Vybrané fyzické operátory: merge-sort

Algoritmus sort (externý merge-sort)

Triedenie dlhého súboru, ktorý sa nezmestí celý do operačnej pamäte. Nech je v operačnej pamäti miesto pre M diskových blokov

1.fáza, vytvorenie utriedených behov:

i=0;

Opakuj pre celú reláciu:

- Prečítaj M blokov do operačnej pamäte
- Utrieď bloky v operačnej pamäti (heapsort, resp. quicksort)
- Zapíš utriedené bloky do behu R_i; i=i+1;

Nech N je počet behov

Vybrané fyzické operátory: merge-sort

Algoritmus sort (externý merge-sort)

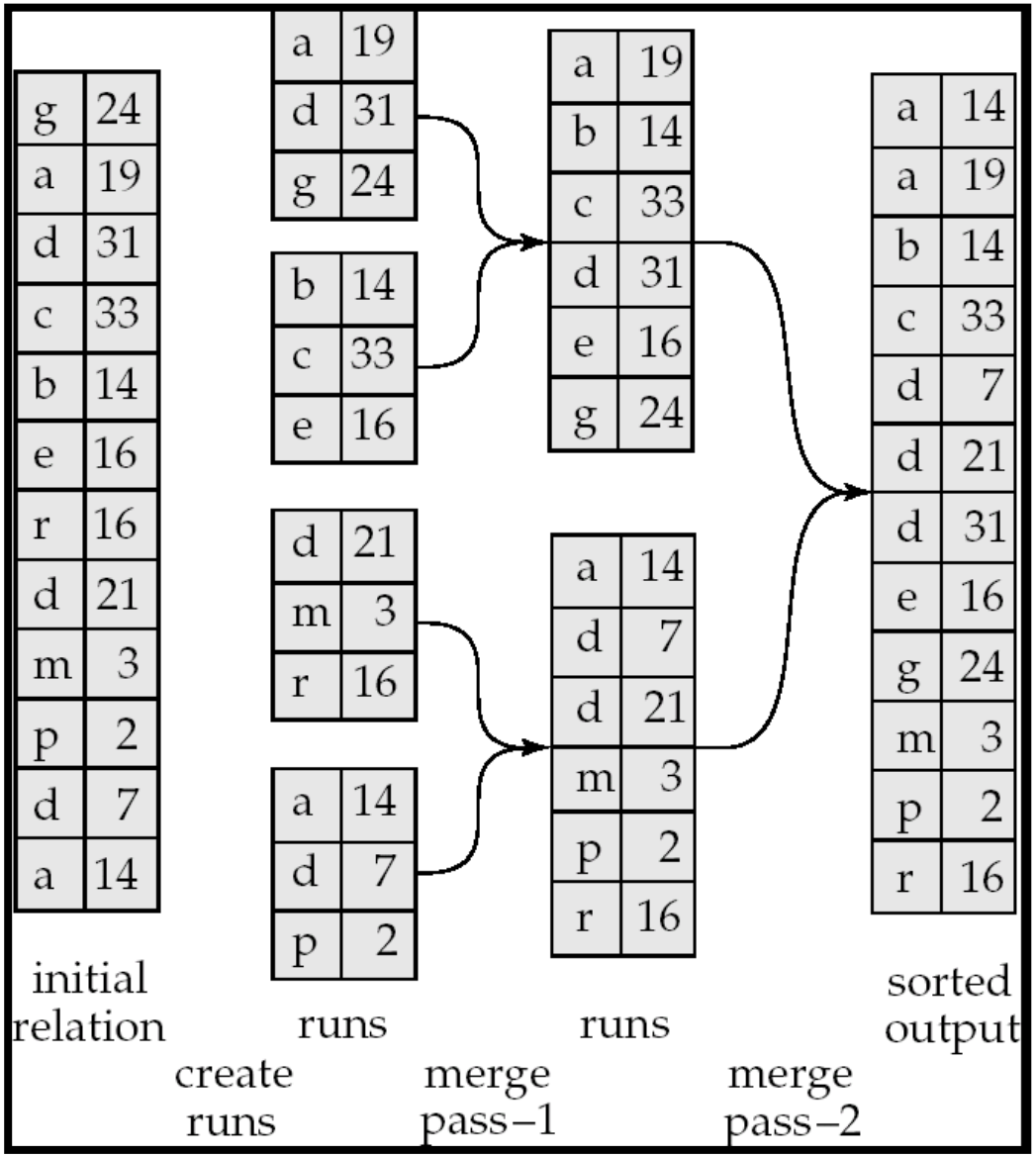
2.fáza, spájanie behov do väčších behov (pre jednoduchosť predpokladajme, že N<M). Použi M-1 blokov v pamäti pre vstup, 1 blok pre výstup:
Načítaj 1 blok *z každého* behu.
Opakuj kým sú vstupné bloky neprázdne:

- Vyber najmenší záznam spomedzi všetkých M-1 behov v pamäti
- Zapíš ten záznam do výstupného bloku. Ak je výstupný blok plný, zapíš ho na disk
- Presuň sa cez ten najmenší záznam. Ak v tom vstupnom bloku nie je ďalší záznam, načítaj ďalší blok z toho behu

Ak N≥M, druhá fáza sa musí niekoľko krát opakovať, až kým nevznikne ***jeden*** utriedený beh

Vybrané fyzické operátory: merge-sort

Príklad (Silberschatz et al.), M=3:



Vybrané fyzické operátory: merge-sort

Zložitosť:

- Počet opakovaní druhej fázy: $\lceil \log_{M-1} (B(R) / M) \rceil$
- Počet diskových operácií v 1. fáze aj v 2. fáze je 2 B(R), ak počítame aj výstup na disk
- Celkový počet diskových operácií je (približne)
  $cost = 2\ B(R)\ (1 + \lceil \log_{M-1} (B(R) / M) \rceil)$

Vybrané fyzické operátory: merge-sort

Iterator **Sort(R)**

```
private table oldR, newR;
open(R) {
  R.open();
  mergesort(R, newR);
  R.close();
  oldR=R;
  R=newR;
  R.open();
}
next(R) {
  return R.next();
}
close(R) {
  R.close();
  R.drop();
  R=oldR;
}
```

```
oscon=# explain select oid from pg_proc order by oid;
                        QUERY PLAN
-----------------------------------------------------------------
 Sort  (cost=181.55..185.92 rows=1747 width=4)
   Sort Key: oid
   ->  Seq Scan on pg_proc  (cost=0.00..87.47 rows=1747
width=4)
```

- Explicit: ORDER BY clause
- Implicit: Handling Unique, Sort-Merge Joins, and other operators
- Has startup cost: cannot return right away

Takmer celú prácu iterátora sort treba urobiť počas open(). V tomto príklade je odhad startup cost 185.92, čo zodpovedá internému volaniu mergesort. Ako už vieme, mergesort je 2-fázový (resp. multifázový, ak je málo RAM):

1. Číta postupne všetky bloky, utriedi každý z nich v RAM a zapíše na disk.
2. Spojí utriedené bloky do jednej tabuľky.

Vybrané fyzické operátory: nested-loop join

Algoritmus join (nested-loop join)

$R \bowtie_c S$

Predpokladajme, že R má menej blokov než S.

Prečítaj M-2 blokov **menšej** relácie R do RAM. 1 blok rezervuj pre vstup S a 1 blok pre výstup. Postupne prečítaj všetky bloky S, a pre každý blok aplikuj join záznamov R v RAM na záznamy v práve načítanom bloku S. Ak joinovacia podmienka *c* platí, zapíš výsledný záznam do výstupného bloku. Ak je výsledný blok plný, zapíš ho na disk

Ak R má viac než M-2 blokov, prečítaj nasledujúcich M-2 blokov R do RAM a zopakuj postupné čítanie celej S. Atď. (počet opakovaných čítaní S je $\lceil B(R) / (M-2) \rceil$).

cost = $B(R) + \lceil B(R) / (M-2) \rceil * B(S)$

Toto nezahŕňa počet zápisov na disk, veľkosť joinu závisí od joinovacej podmienky a obsahu relácií. Špeciálnym prípadom je kartézsky súčin, kde joinovacia podmienka splnená pre každú dvojicu záznamov

Vybrané fyzické operátory: nested-loop join

Iterator **NestedLoopJoin(R,S)**

```
private tuple r,s;
open(R,S) {
  R.open();
  S.open();
  s=S.next();
}
close(R,S) {
  R.close();
  S.close();
}
```

```
next(R,S) {
  if (s==EOF)
    return EOF;
  do {
    r=R.next();
    if (r=EOF)
      R.close();
      s=S.next();
      if (s==EOF)
        return EOF;
      R.open();
  } while not (r and s join);
  return join of r and s;
}
```

```
oscon-# select * from pg_foo join pg_namespace on
(pg_foo.pronamespace=pg_namespace.oid);
                        QUERY PLAN
-------------------------------------------------------------------------
 Nested Loop  (cost=1.05..39920.17 rows=5867 width=68)
   Join Filter: ("outer".pronamespace = "inner".oid)
   -> Seq Scan on pg_foo  (cost=0.00..13520.54 rows=234654 width=68)
   -> Materialize  (cost=1.05..1.10 rows=5 width=4)
        -> Seq Scan on pg_namespace  (cost=0.00..1.05 rows=5 width=4)
```

- Joins two tables (two input sets)
- USED with INNER JOIN and LEFT OUTER JOIN
- Scans 'outer' table, finds matches in 'inner' table
- No startup cost
- Can lead to slow queries when not indexed, especially when functions are in the select clause

Vybrané fyzické operátory: merge join

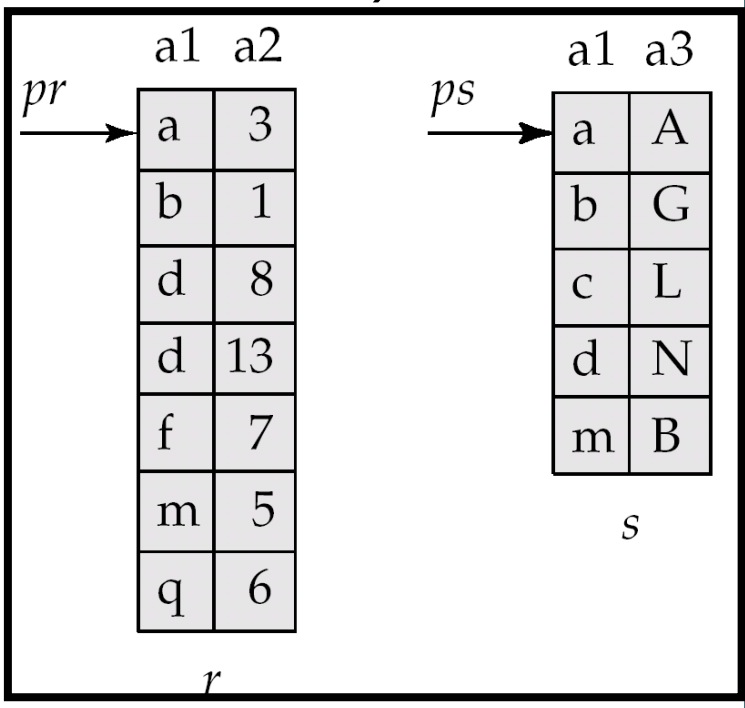
Algoritmus join (merge join)

Tento algoritmus dovoľuje len joinovaciu podmienku na rovnosť (equijoin, resp. natural join) a predpokladá, že obe relácie sú utriedené

cost = B(R) + B(S)

(Cena písania na disk tu nie je zahrnutá, veľkosť joinu závisí od joinovacej podmienky a obsahu relácií.)

Podobný 2. fáze merge-sort algoritmu. Ale nie celkom rovnaký: rozdiel je v spracovaní duplikátov!



Fyzická optimalizácia dotazov: zložitosť fyz. operátorov

Iterator **IndexScan(R)**

```
open(R) {
  R.open();
}
close(R) {
  R.close();
}
next(R) {
  return R.index().next();
}
```

```
oscon=# explain select oid from pg_proc where oid=1;
                              QUERY PLAN
------------------------------------------------------------------------
 Index Scan using pg_proc_oid_index on pg_proc  (cost=0.00..5.99
rows=1 width=4)
   Index Cond: (oid = 1::oid)
```

IndexScan je pri veľkých reláciách oveľa rýchlejší ako SeqScan

Indexy

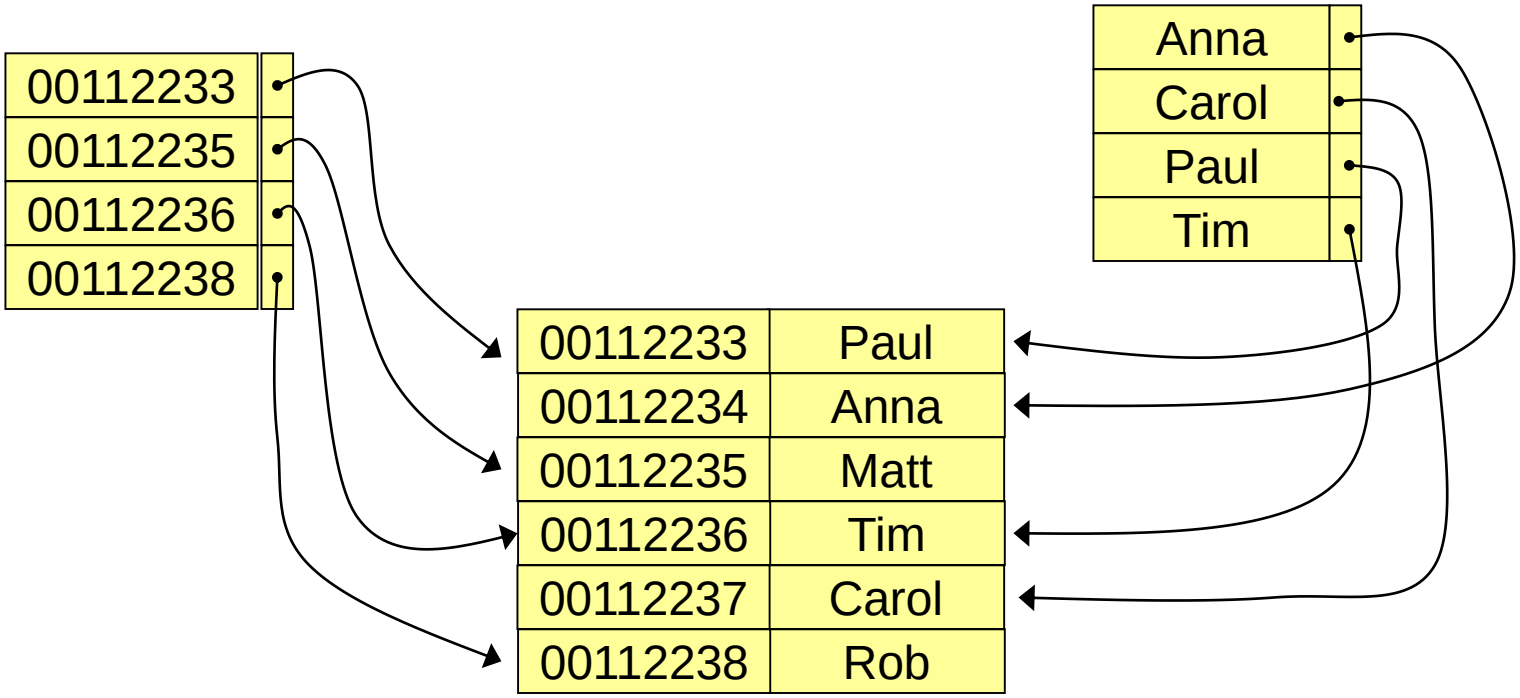
Motivácia: pri hľadaní záznamu napr. podľa kľúča sa chceme vyhnúť sekvenčnému prehľadávaniu (sequential scan). Pozor, „vyhľadávací kľúč“ je nový pojem!

Jednou z možností je udržiavať tabuľku na disku utriedenú podľa toho kľúča. Toto síce redukuje čas vyhľadávania, ale spôsobuje **problémy pri vkladaní a vynechávaní**, podobné problémom pri vkladaní a vynechávaní do poľa v RAM. Navyše, čo ak chceme vyhľadávať podľa viacerých rôznych kľúčov?

Idea: index v nezávislom súbore (ISAM, Index Sequential Access Method). Záznamy v indexovom súbore sú dvojice [kľúč, pointer]. Tabuľka je neusporiadaná, ale **index je usporiadaný podľa nejakého kľúča**. K jednej tabuľke môže byť priradených viacero indexov, ktoré sú usporiadané podľa rôznych kľúčov

Primárny sekvenčný index (clustered index)

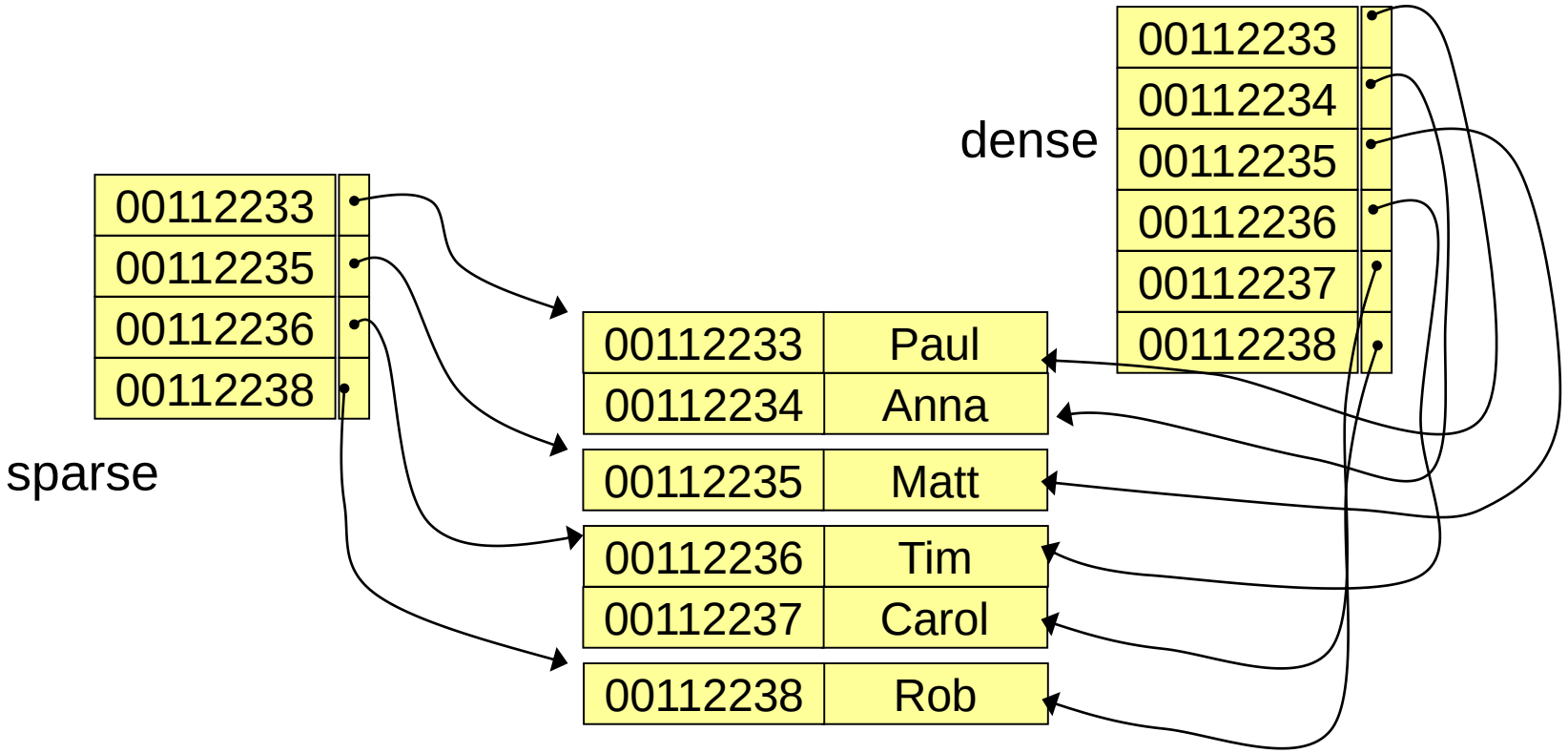
Primárny sekvenčný index (clustered index) definuje fyzické usporiadanie relácie, spravidla podľa primary key. Je možné pridať k relácii ďalšie indexy podľa iných atribútov (dokonca nie nutne kľúčových—vyhľadávací kľúč nemusí byť kľúč, resp. nadkľúč relácie)



Typy sekvenčných indexov: dense, sparse

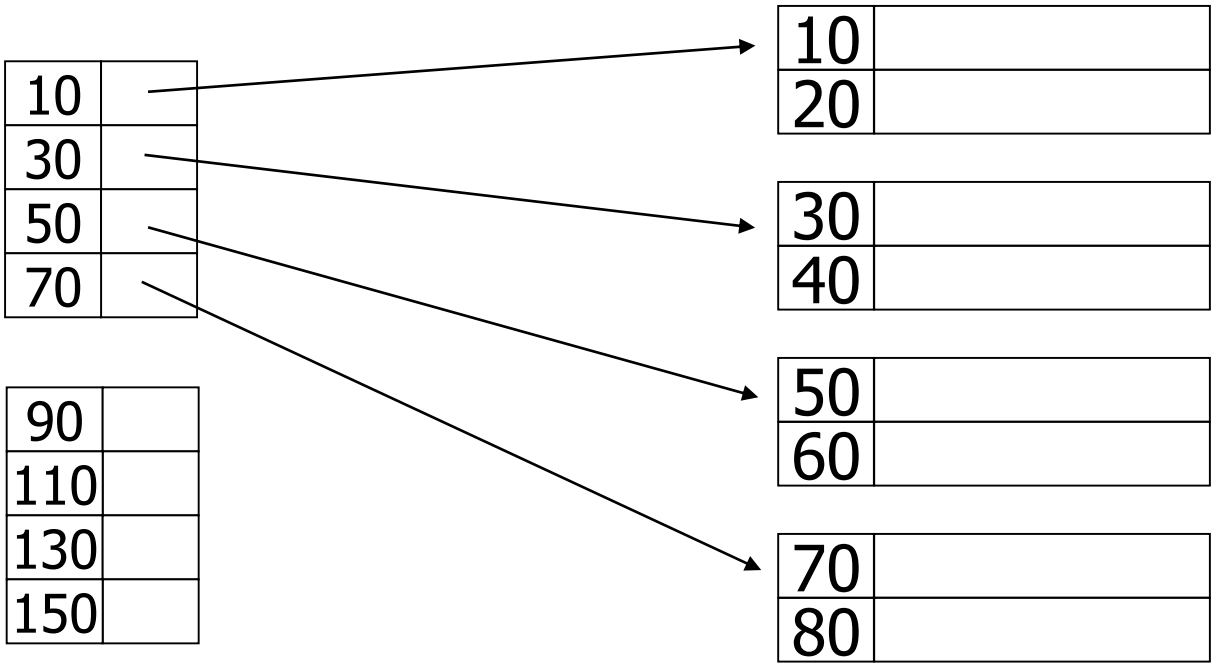
Dense (hustý) index obsahuje adresy všetkých (existujúcich) záznamov

Sparse (riedky) index obsahuje len adresy niektorých záznamoch, napr. adresy začiatkov blokov



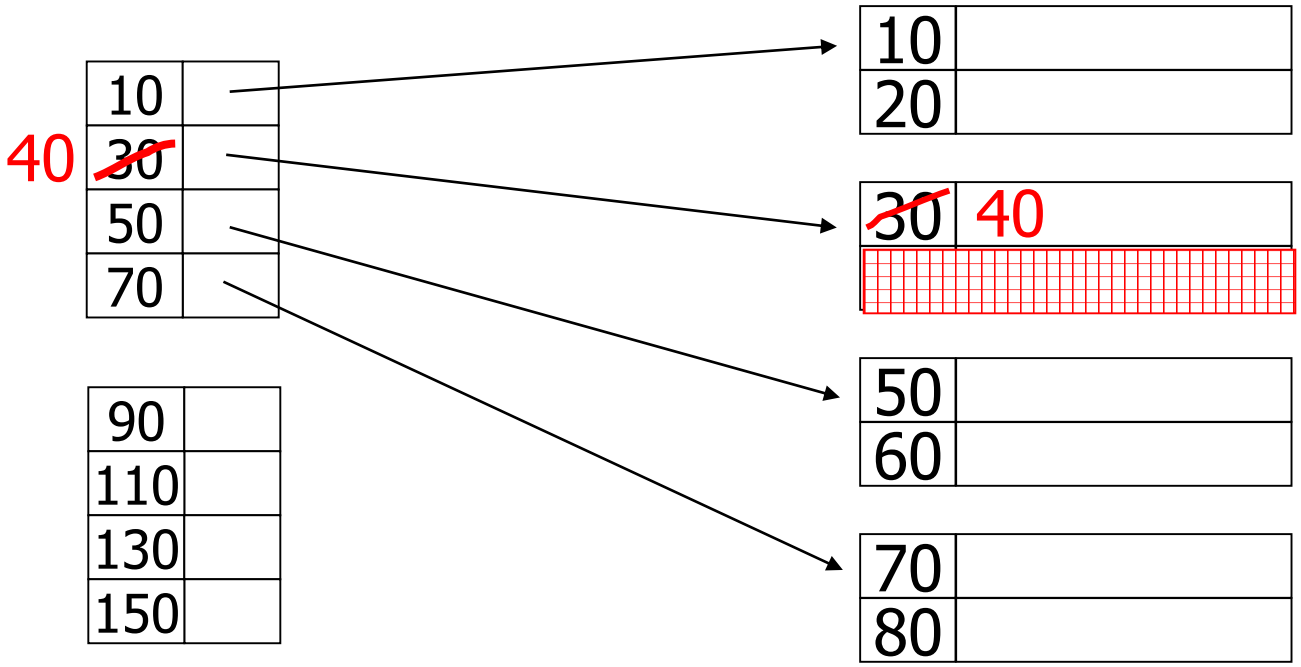
Vynechávanie z riedkeho sekvenčného indexu

Príklad (Gupta)



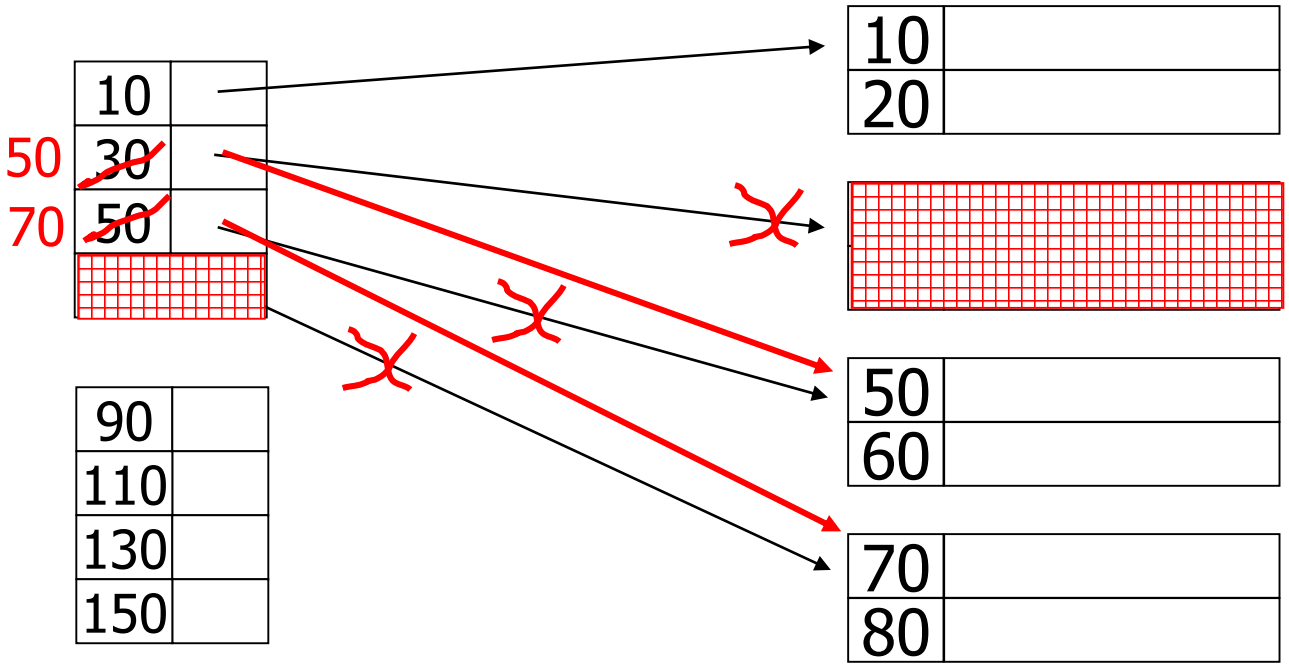
Vynechávanie z riedkeho sekvenčného indexu

Príklad (Gupta): vynechanie záznamu 30



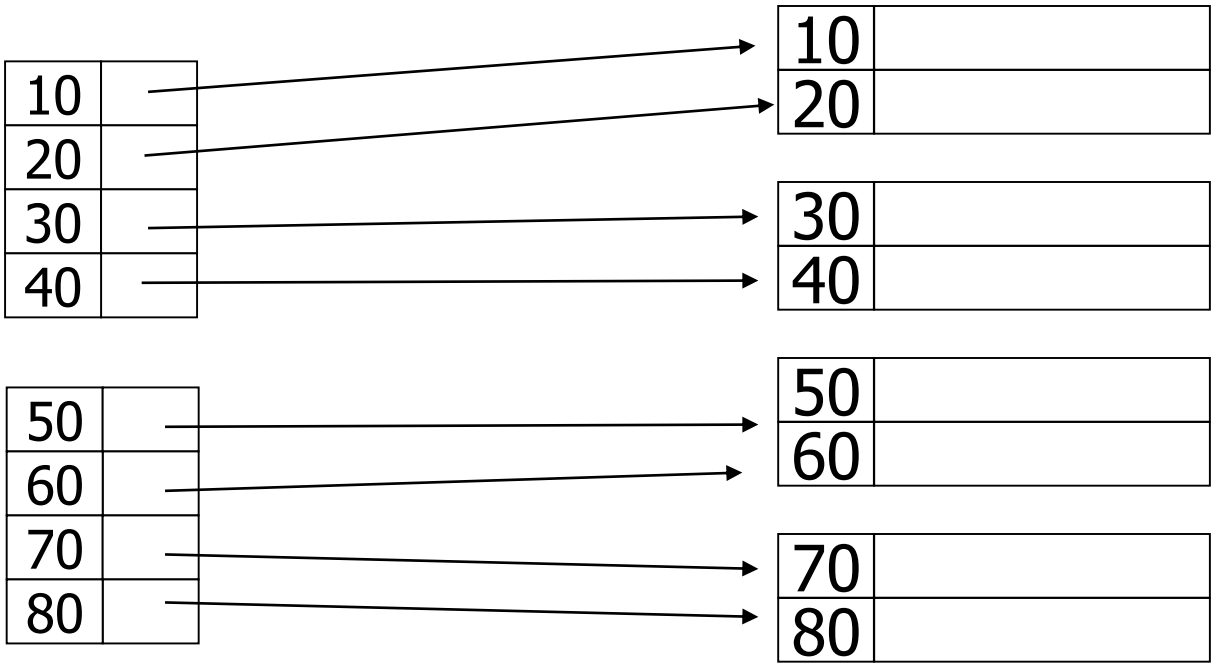
Vynechávanie z riedkeho sekvenčného indexu

Príklad (Gupta): následné vynechanie záznamu 40. Vzniká voľný blok



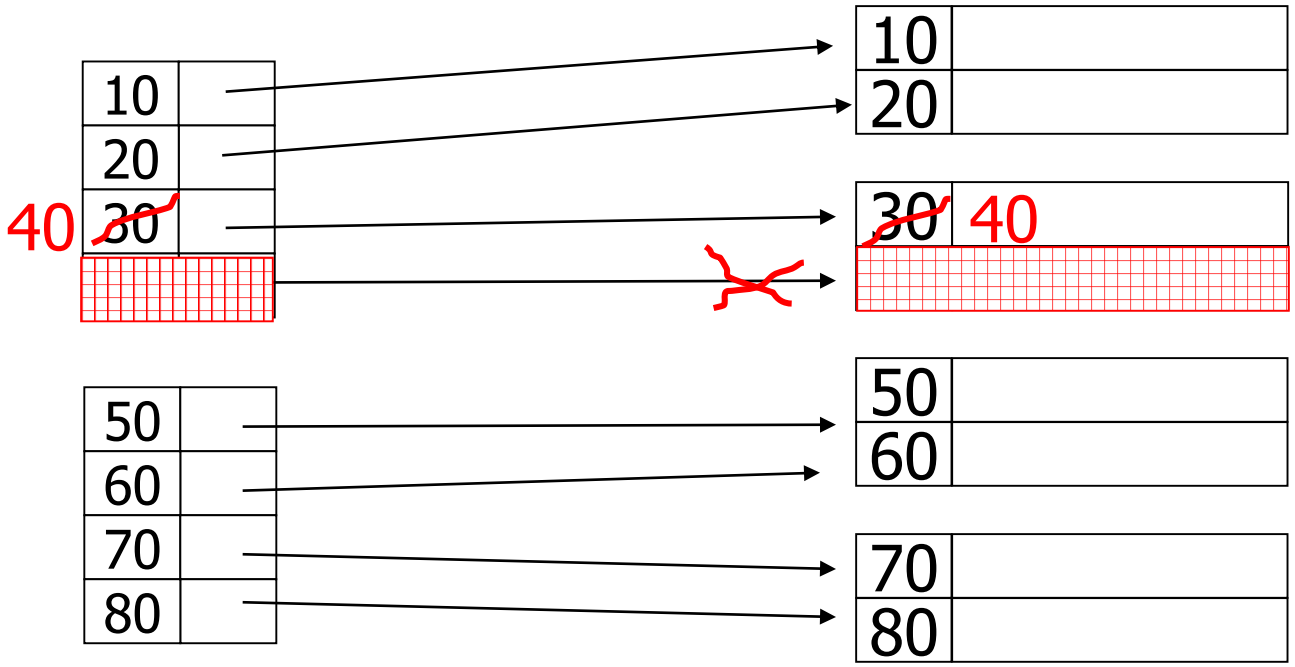
Vynechávanie z hustého sekvenčného indexu

Príklad (Gupta)



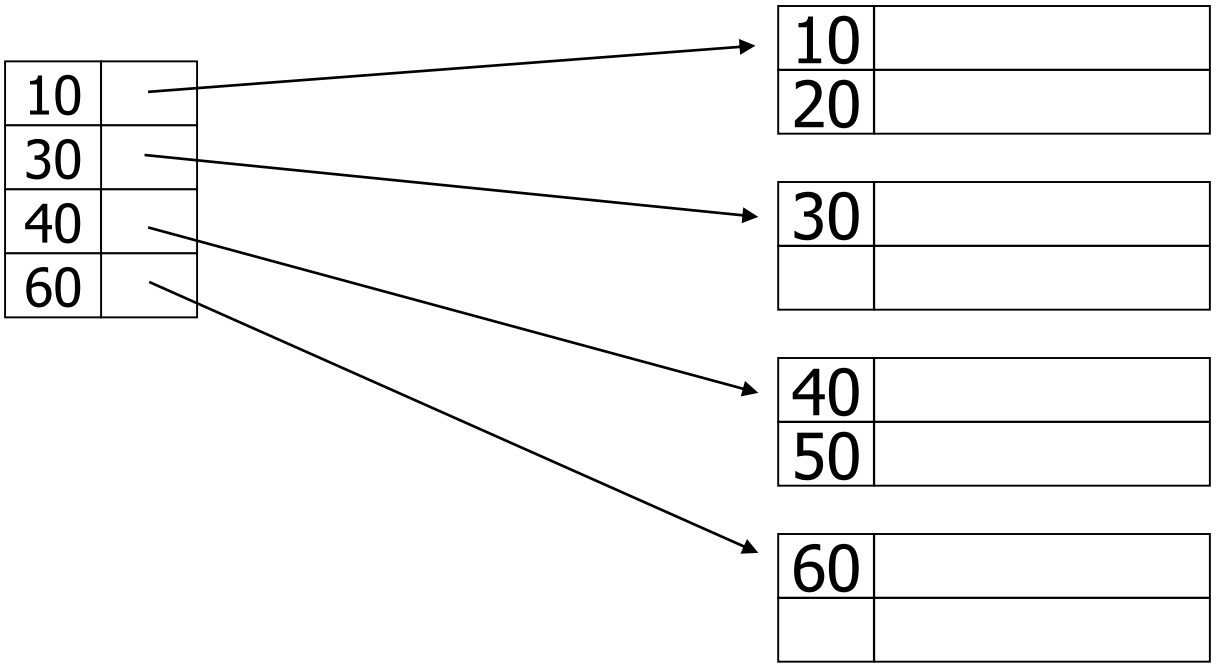
Vynechávanie z hustého sekvenčného indexu

Príklad (Gupta): vynechanie záznamu 30



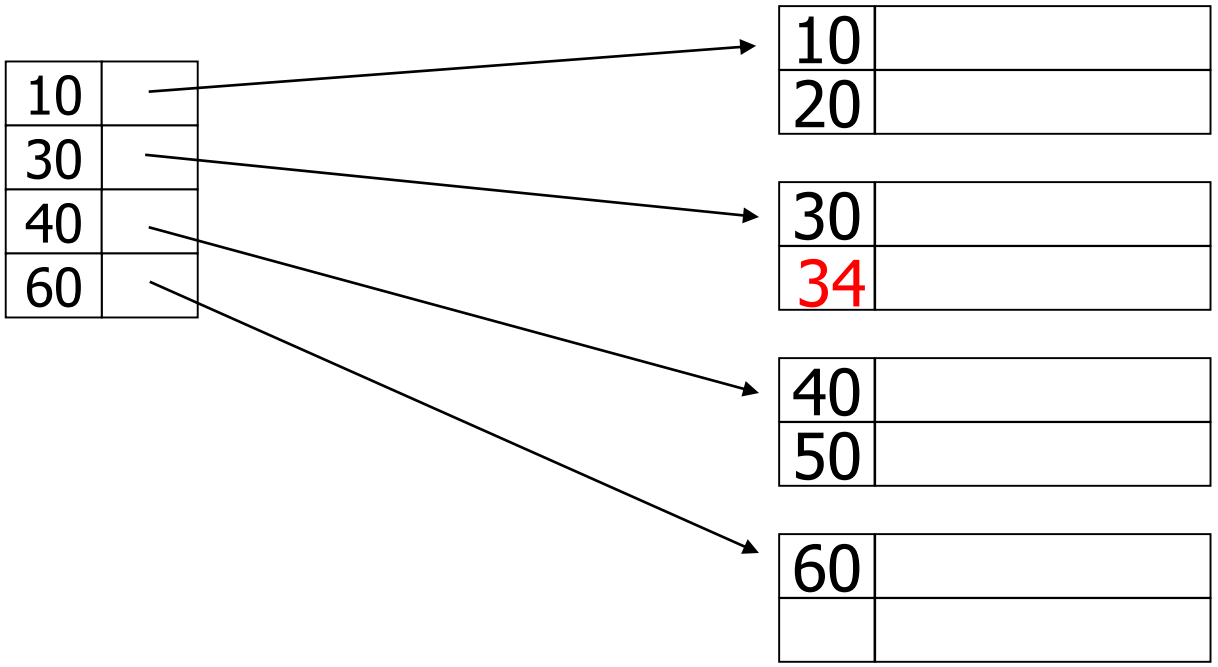
Vkladanie do riedkeho sekvenčného indexu

Príklad (Gupta): vloženie záznamu 34



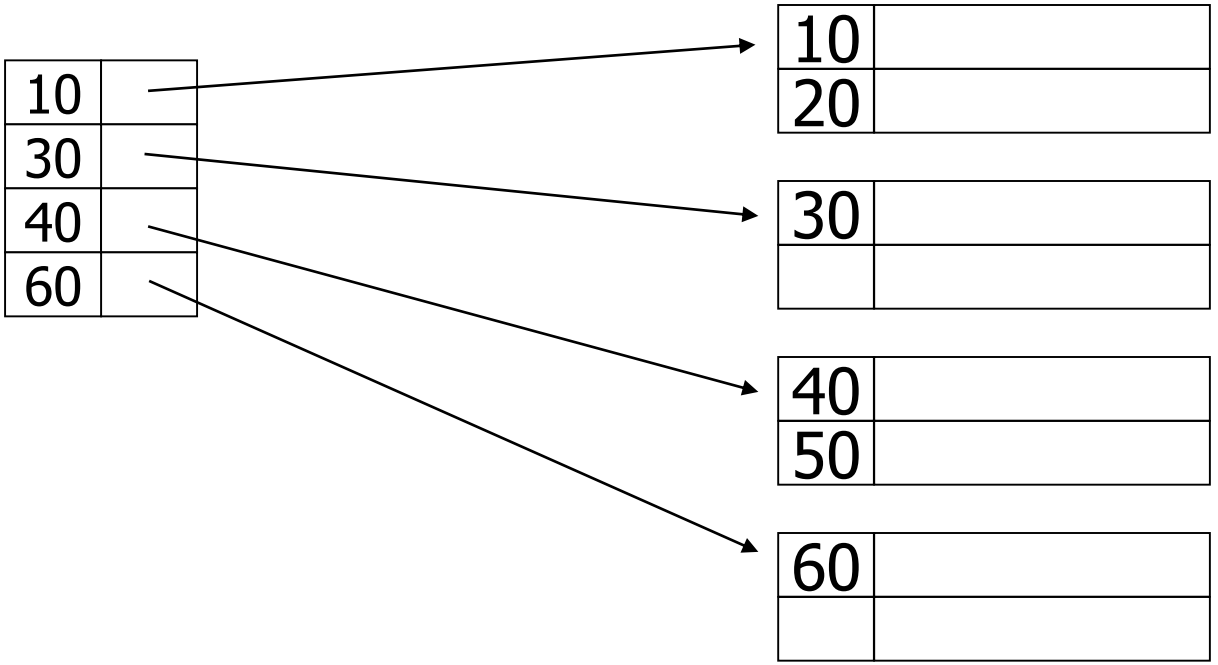
Vkladanie do riedkeho sekvenčného indexu

Príklad (Gupta): vloženie záznamu 34. Ak je v bloku miesto, jednoducho vložíme záznam



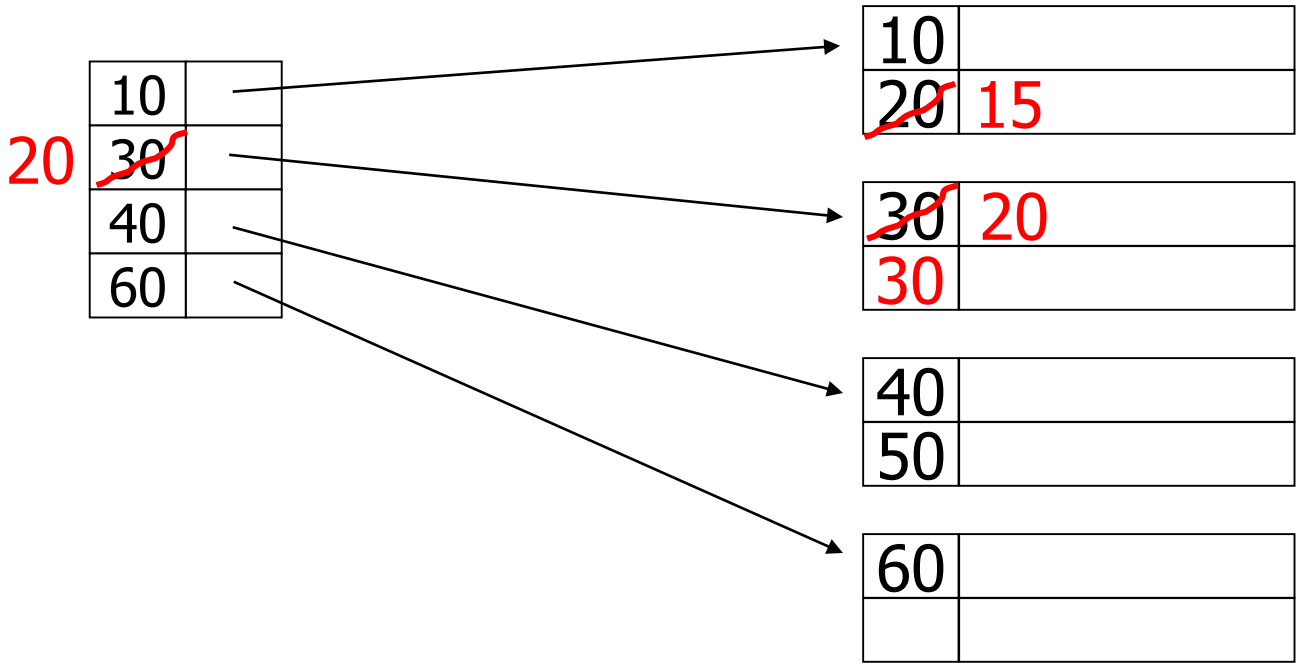
Vkladanie do riedkeho sekvenčného indexu

Príklad (Gupta): vloženie záznamu 15. Ak nie je v bloku miesto, máme dve alternatívy: 1.buď okamžite reorganizujeme index alebo 2.použijeme blok preplnenia



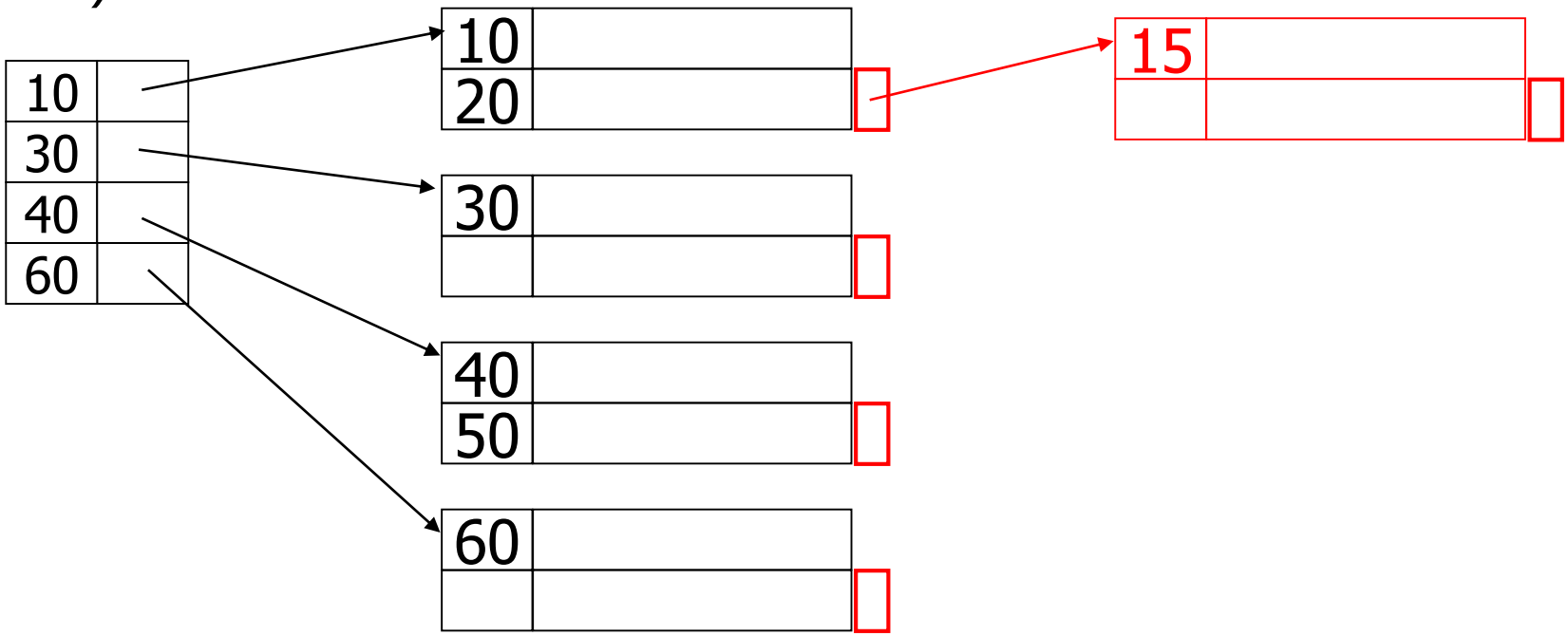
Vkladanie do riedkeho sekvenčného indexu

Príklad (Gupta): vloženie záznamu 15, reorganizácia indexu

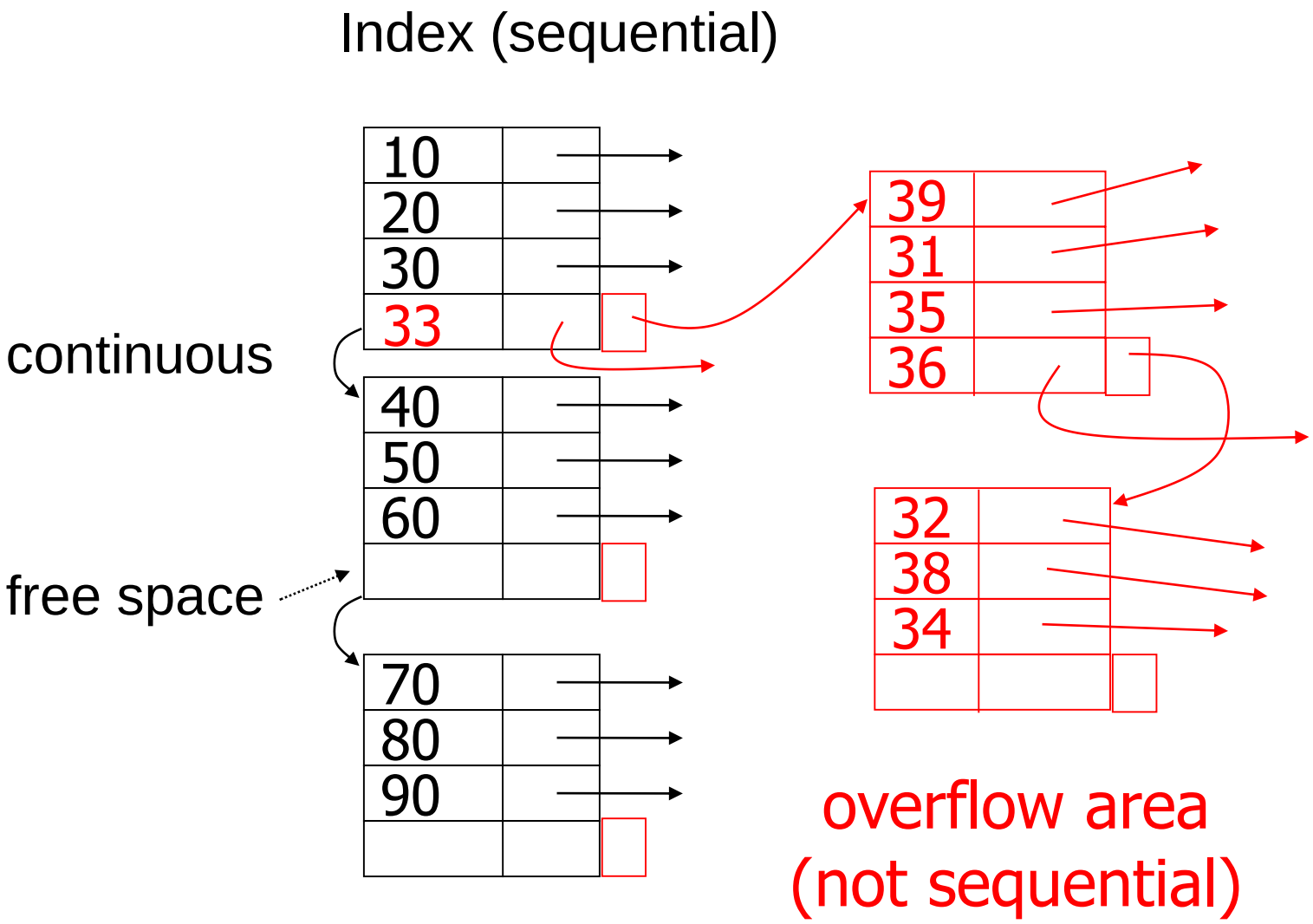


Vkladanie do riedkeho sekvenčného indexu

Príklad (Gupta): vloženie záznamu 15 s použitím bloku preplnenia. (Ak v budúcnosti vznikne príliš veľa blokov preplnenia, bude treba index preorganizovať alebo vytvoriť nanovo.)

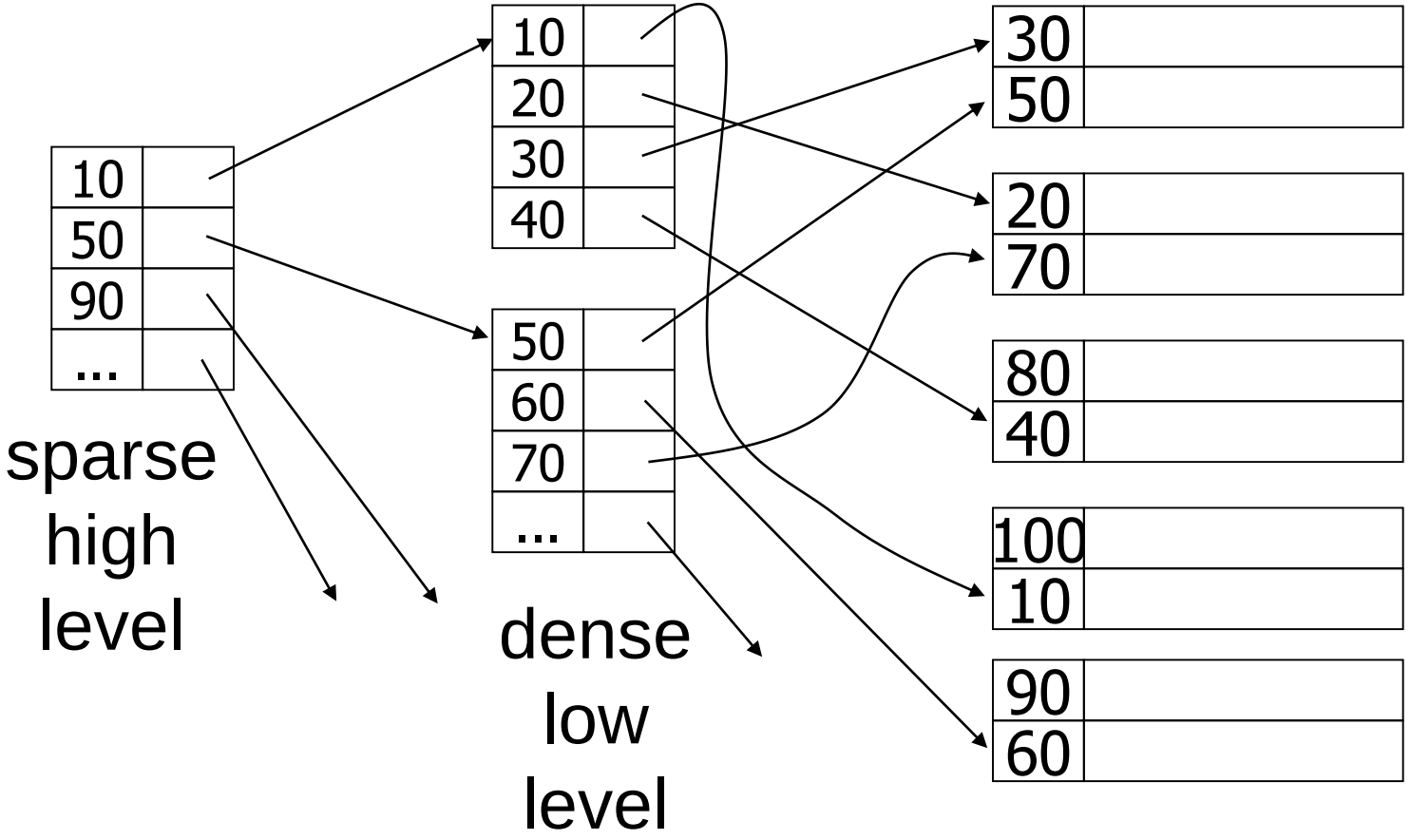


Vynechávanie a vkladanie: všeobecný stav indexu



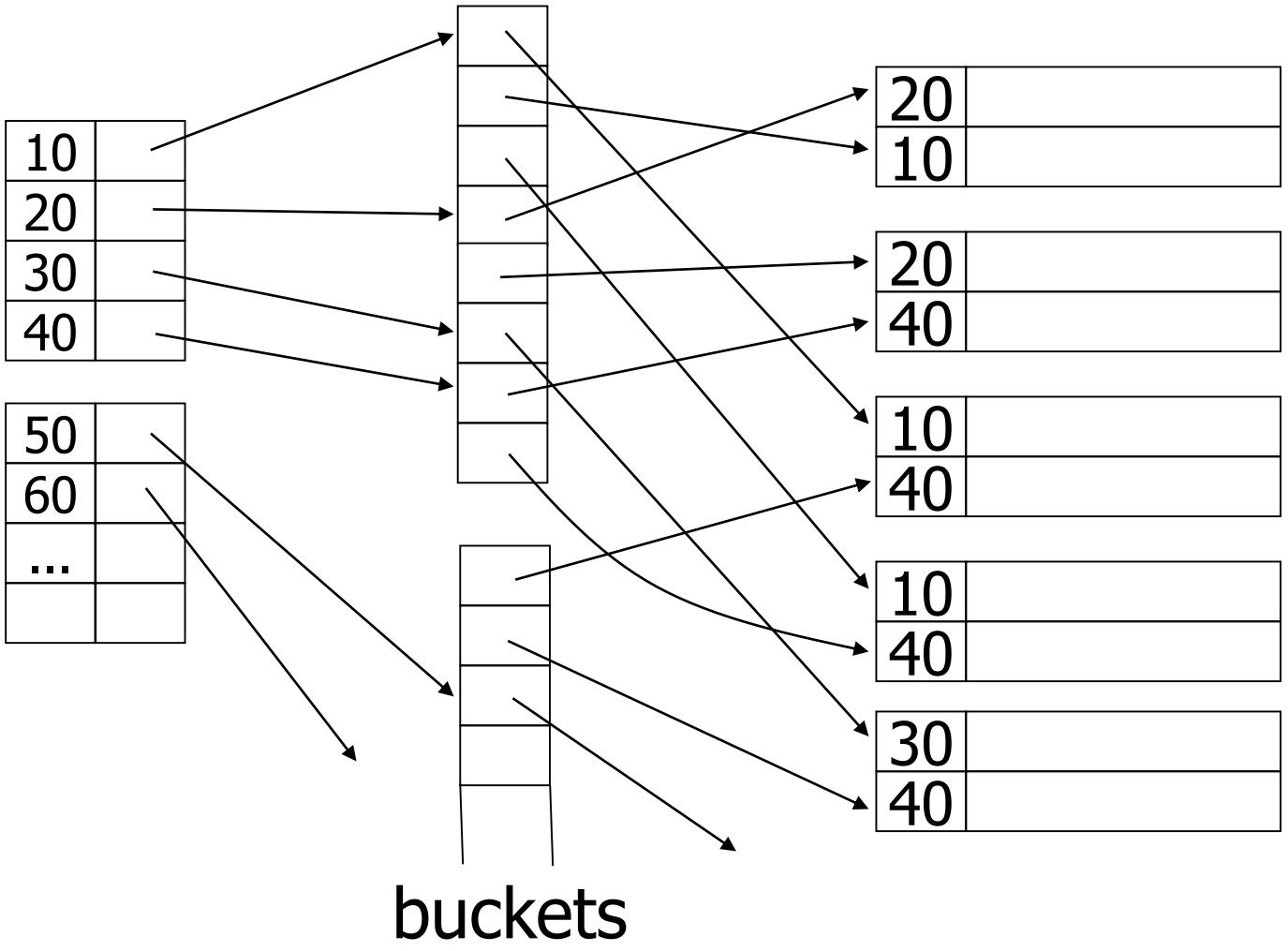
Multilevel indexy (index indexu)

Pri multilevel indexoch musí byť
najvnútornejší index vždy hustý!



Multilevel indexy (index indexu)

Pri multilevel indexoch musí byť **najvnútornejší index vždy hustý!** (Toto platí aj pre hashovanie.)



Sekvenčné indexy: prehodnotenie

Keďže pre každú reláciu treba implementovať prinajmenšom SeqScan (treba vedieť enumerovať všetky záznamy relácie)**, ku každej relácii je priradený nejaký sekvenčný index**

Výhody:

• jednoduchá implementácia

• vhodné pre sekvenčné prehľadávanie: v prípade hustého indexu vieme rozhodnúť o existencii záznamu **bez toho, aby sme pristupovali k relácii (dátovému súboru)**

Nevýhody:

• drahé vkladanie

• **drahé vyhľadávanie**

Riešenie nevýhod: vyhľadávacie stromy (B^+ stromy)

B stromy a B^+ stromy

Vyhľadávacie AVL stromy nie sú pre blokovú organizáciu vhodné, lebo každý uzol AVL obsahuje len 1 záznam

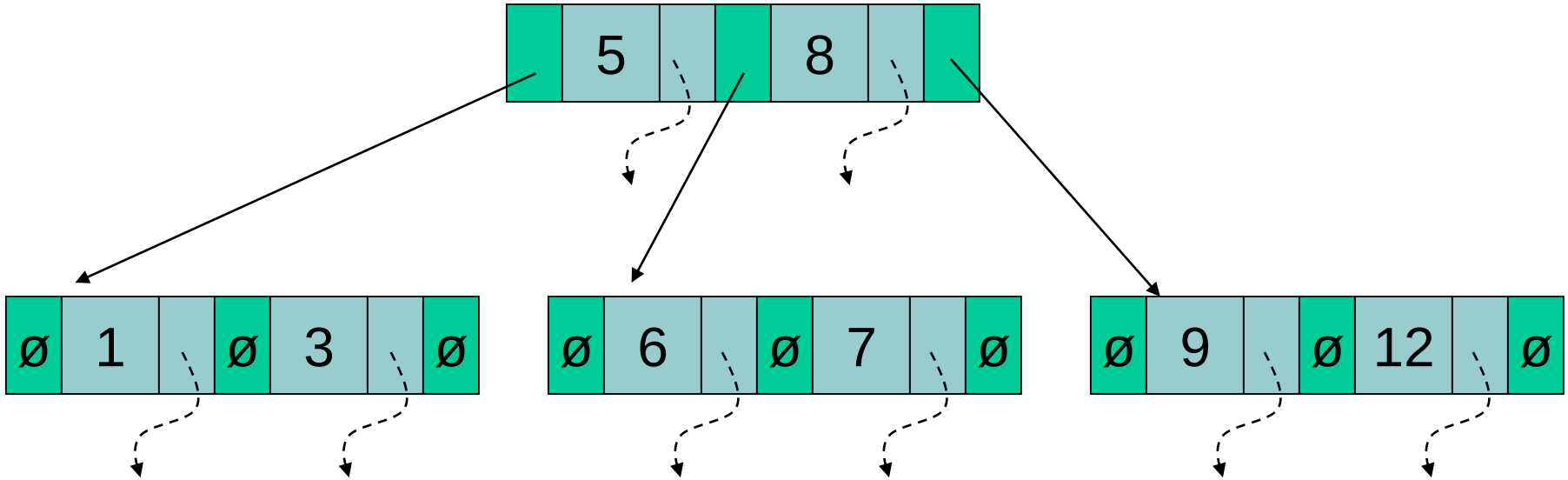
Definícia: **B(m)-strom** je strom, pre ktorý platí:

- Každý uzol okrem koreňa obsahuje *k* záznamov, kde $\lfloor m/2 \rfloor \leq k \leq m$. (Dá sa vyžadovať aj naplnenie $\lfloor 2m/3 \rfloor \leq k \leq m$.)
- Koreň, ak nie je listom, obsahuje aspoň *1* a najviac *m* záznamov
- Vnútorný uzol s *k* záznamami má *k+1* synov
- Všetky listy sú na tej istej úrovni
- **B^+ stromy** majú vo vnútorných uzloch iba kľúče (index) a dátové záznamy sú iba v listoch

2/3 -stromy sú teoreticky výhodnejšie ako 1/2-stromy, lebo majú menšiu hĺbku, ale 2/3 sa ťažšie implementujú

B stromy

Príklad B(2) stromu



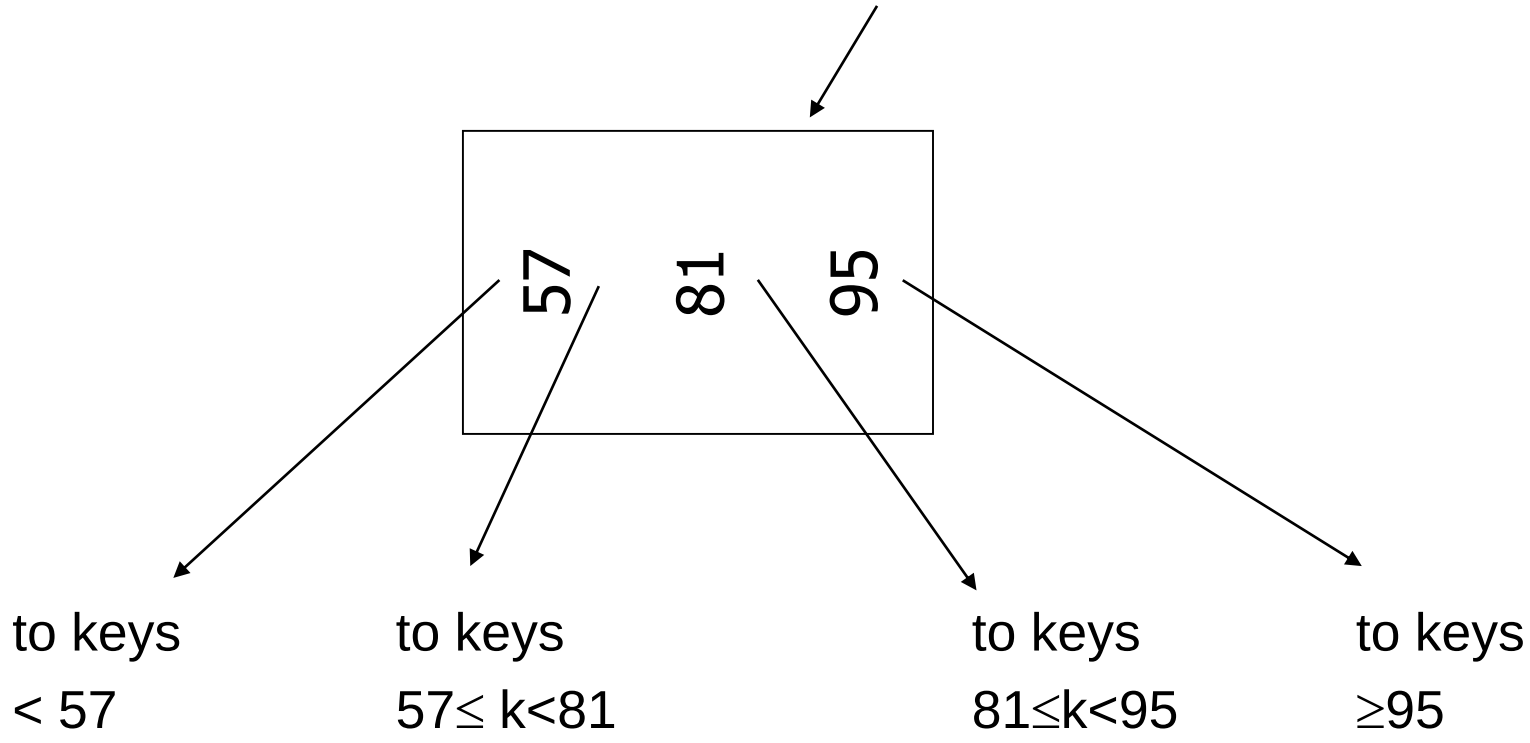
tree pointer

data pointer

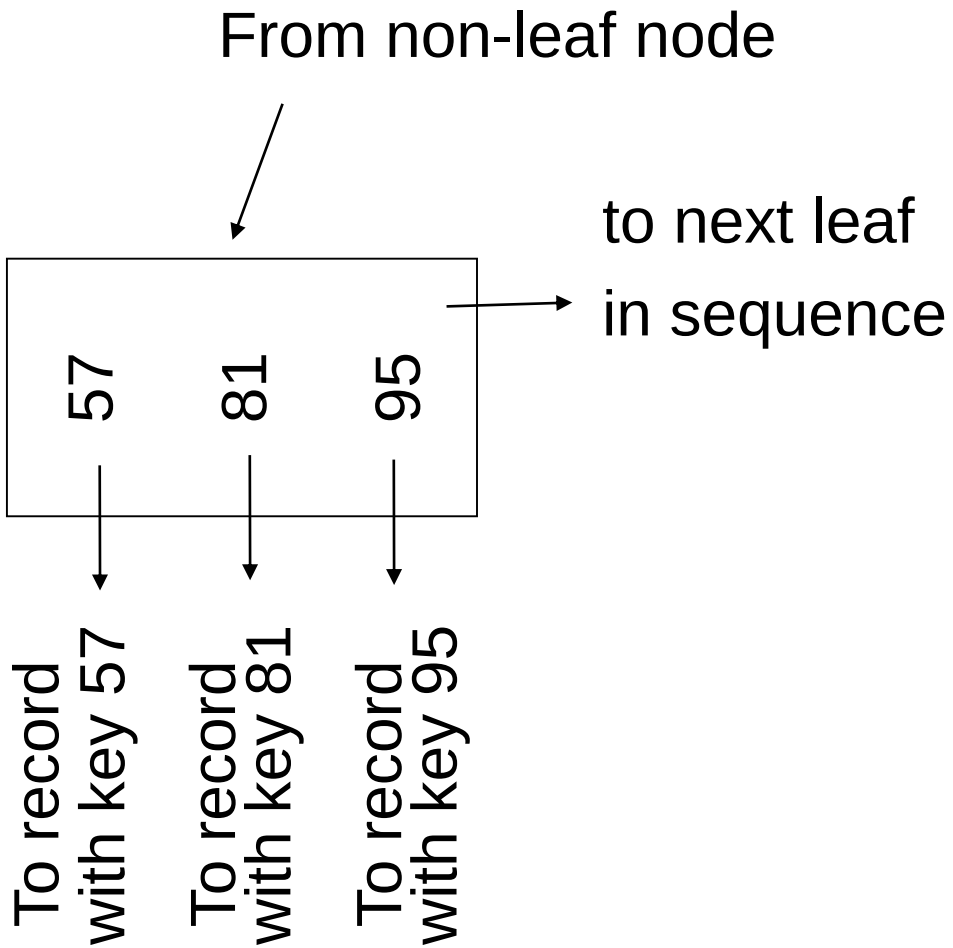
ø null pointer

B^+ stromy

Príklad nelistového (vnútorného) uzla (Gupta)



Príklad listového uzla (Gupta)



B^+ stromy: vkladanie

1. Nájdi list do ktorého záznam patrí a vlož
2. Ak sa list nepreplnil, skonči (A)
3. Ak je list preplnený (m+1 záznamov), tak skontroluj či sa nedajú redistribuovať záznamy medzi susednými listami. Ak áno, urob to a uprav index (dve možnosti: spoločný otec, rôzni otcovia). Potom rozdeľ list na dva a index pre druhú časť vlož do nadradeného uzla (otca) (B)

Vloženie do vnútorného uzla je podobné vloženiu do listu, ale vkladá sa aj príslušný smerník. Pri redistribúcii sa redistribujú aj príslušné smerníky. (C)

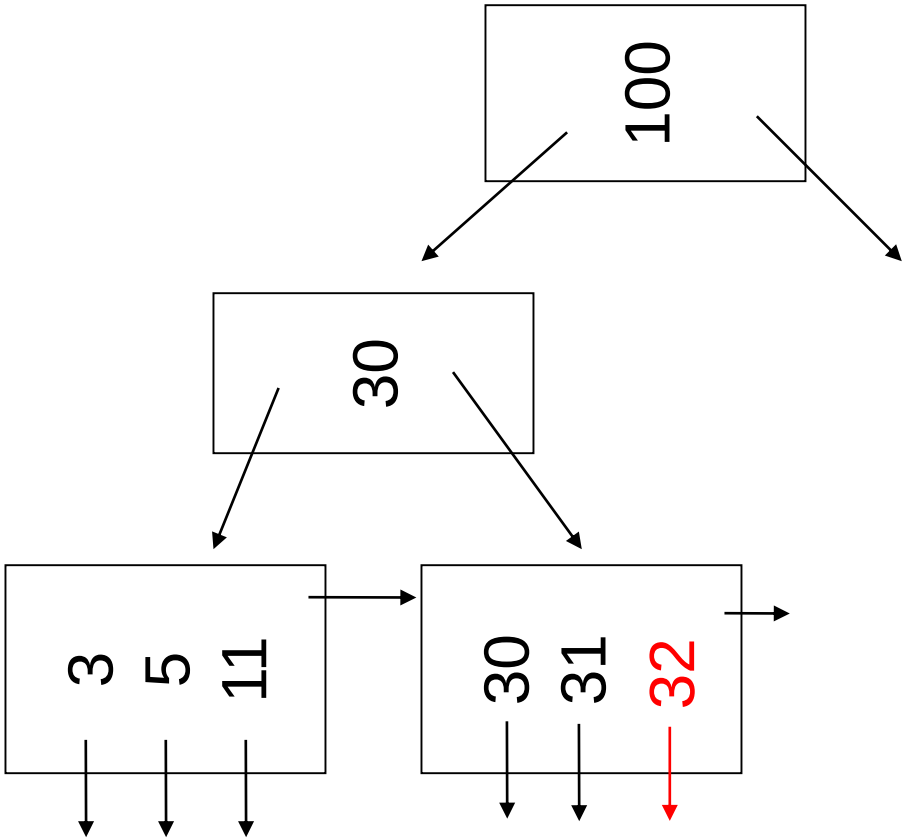
Ak sa preplnil koreň, vznikne nový koreň s jednou hodnotou a smerníkmi na ľavý a pravý podstrom (D)

Nasledujúce obrázky zodpovedajú definícii $B^+(m)$ stromu s naplnením $\lfloor m/2 \rfloor \leq k \leq m$ *(niektoré ukazujú len podstatné fragmenty stromu)*

B+ stromy: vkladanie (A) simple case

Insert 32

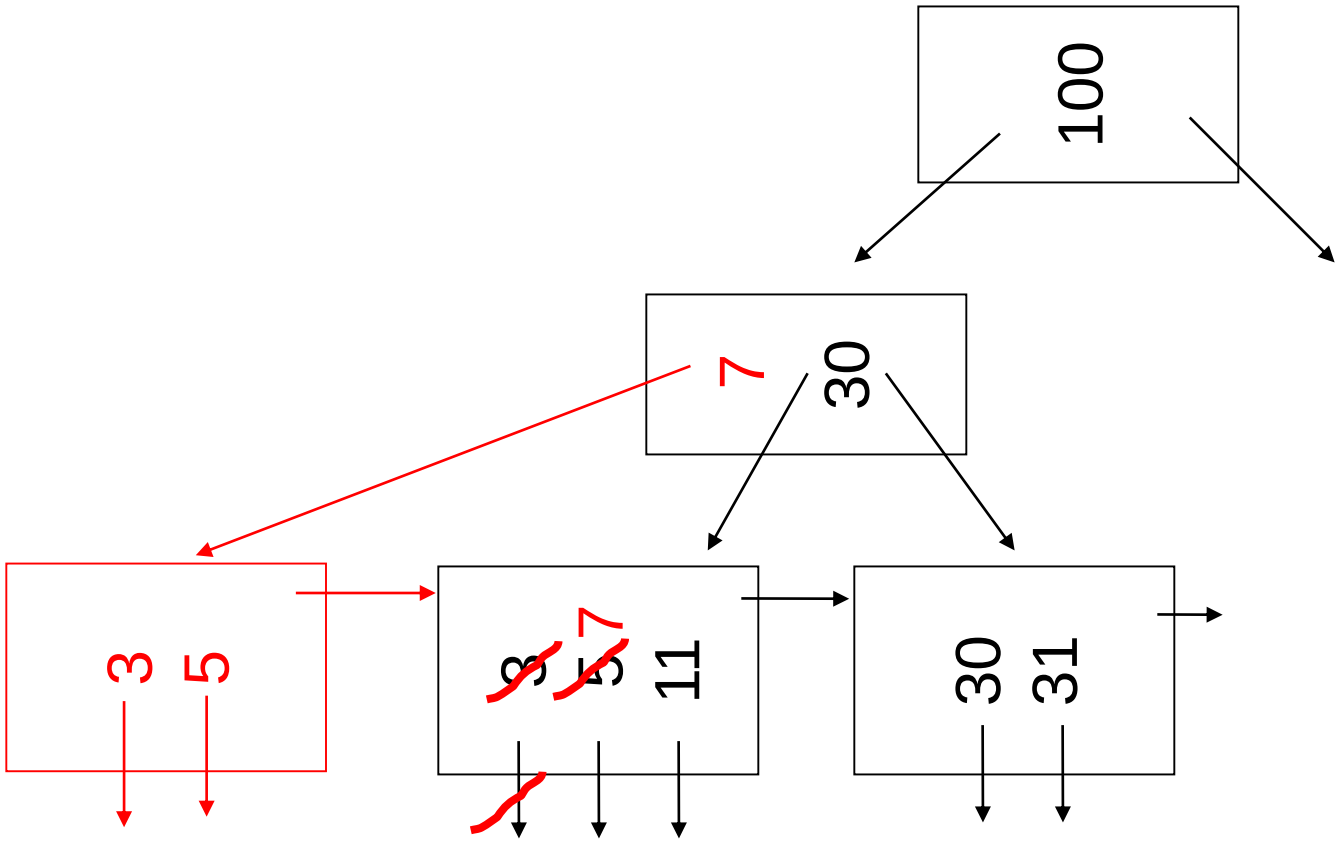
n=3



B+ stromy: vkladanie (B) leaf overflow

Insert 7

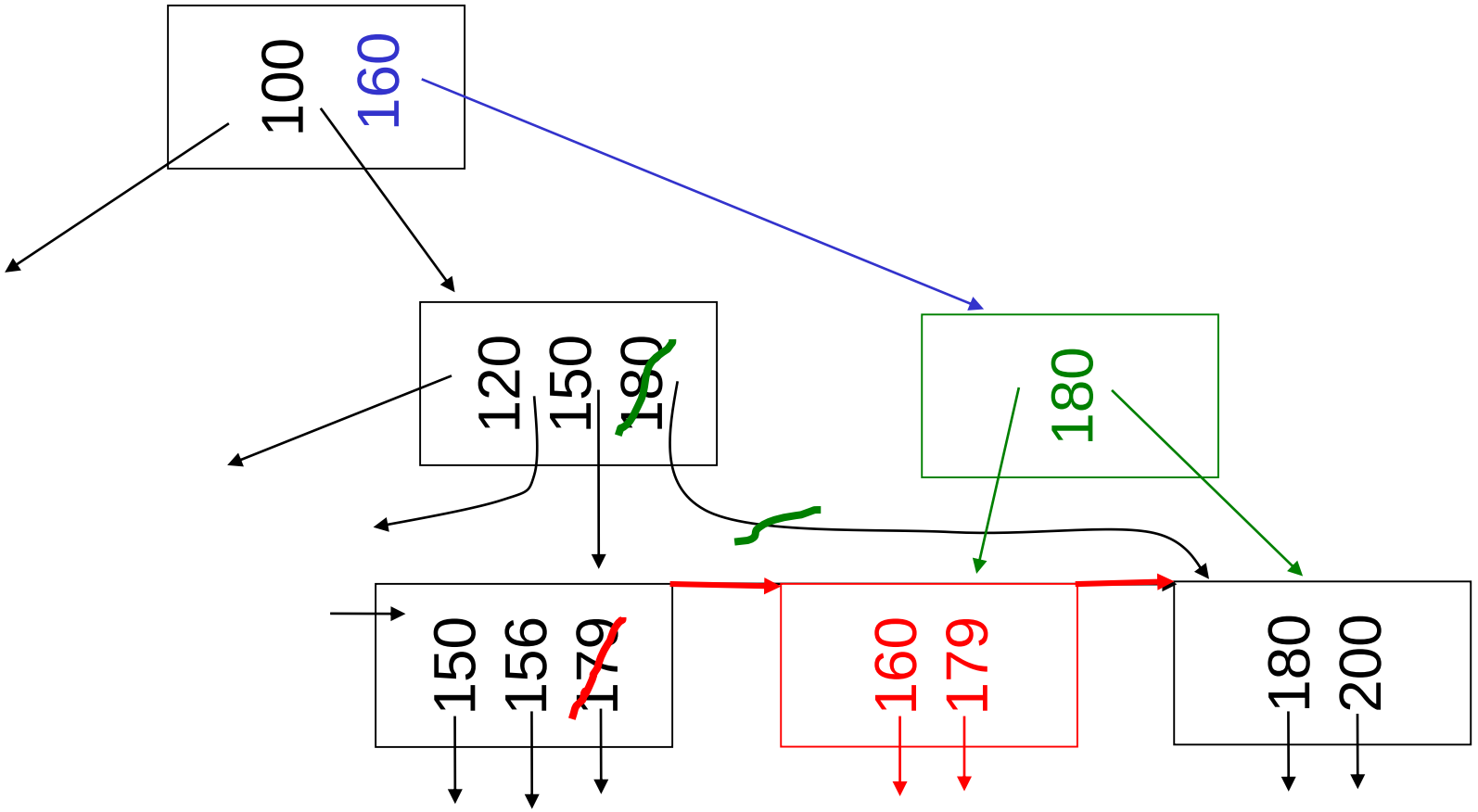
n=3



B+ stromy: vkladanie (C) non-leaf overflow

Insert 160

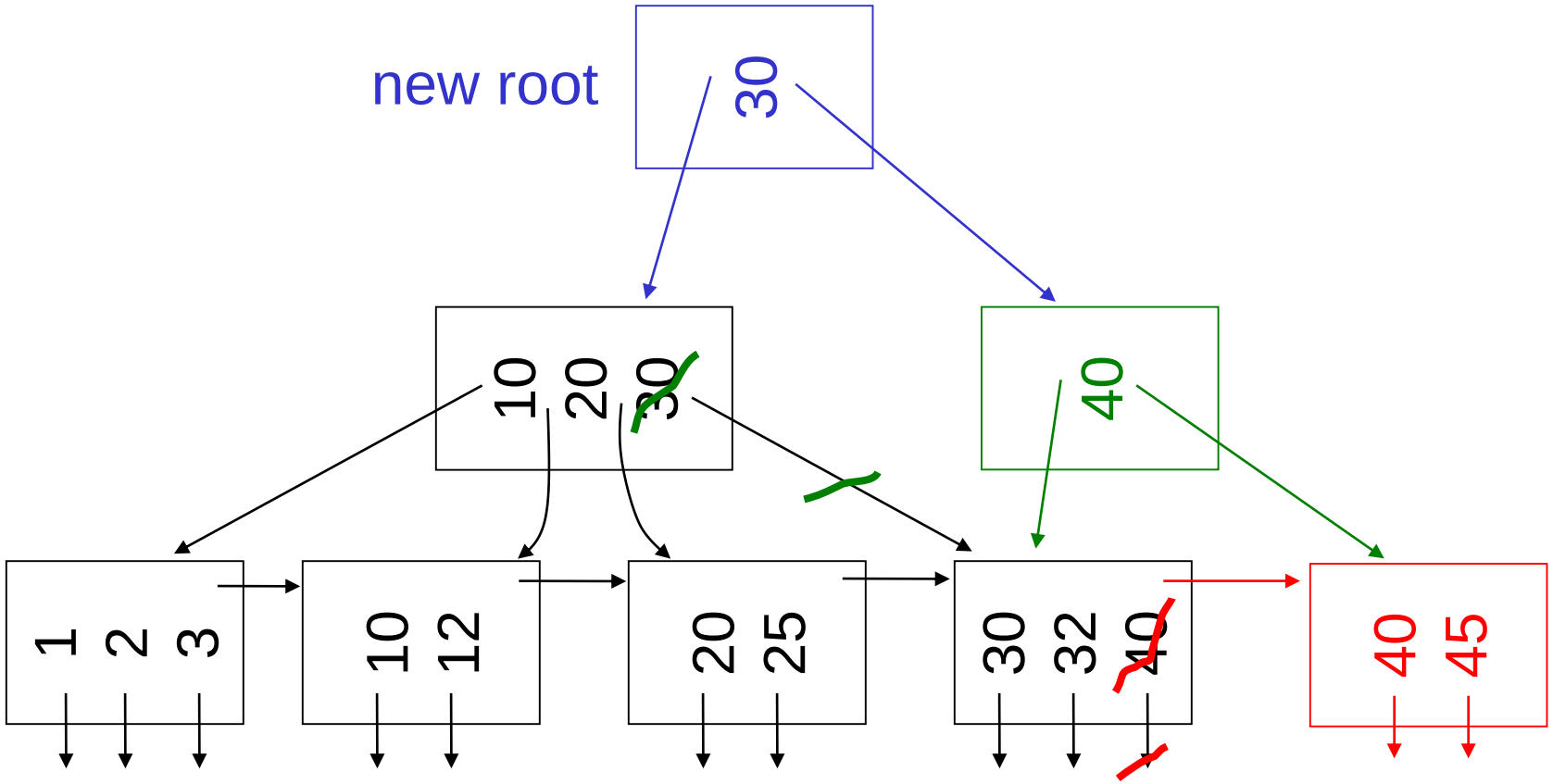
n=3



B+ stromy: vkladanie (D) new root

Insert 45

n=3



B^+ stromy: vynechanie

1. Nájdi a vynechaj záznam
2. Ak list nie je podplnený, skonči (A)
3. Ak je list podplnený a ak je brat dostatočne plný, tak redistribuuj záznamy medzi listom a bratom a aktualizuj otca. (C) Inak zlúč list s bratom a vynechaj smerník a index z otca (B)

Vynechanie z vnútorného uzla je podobné ako vynechanie z listu. Môže sa propagovať až ku koreňu.

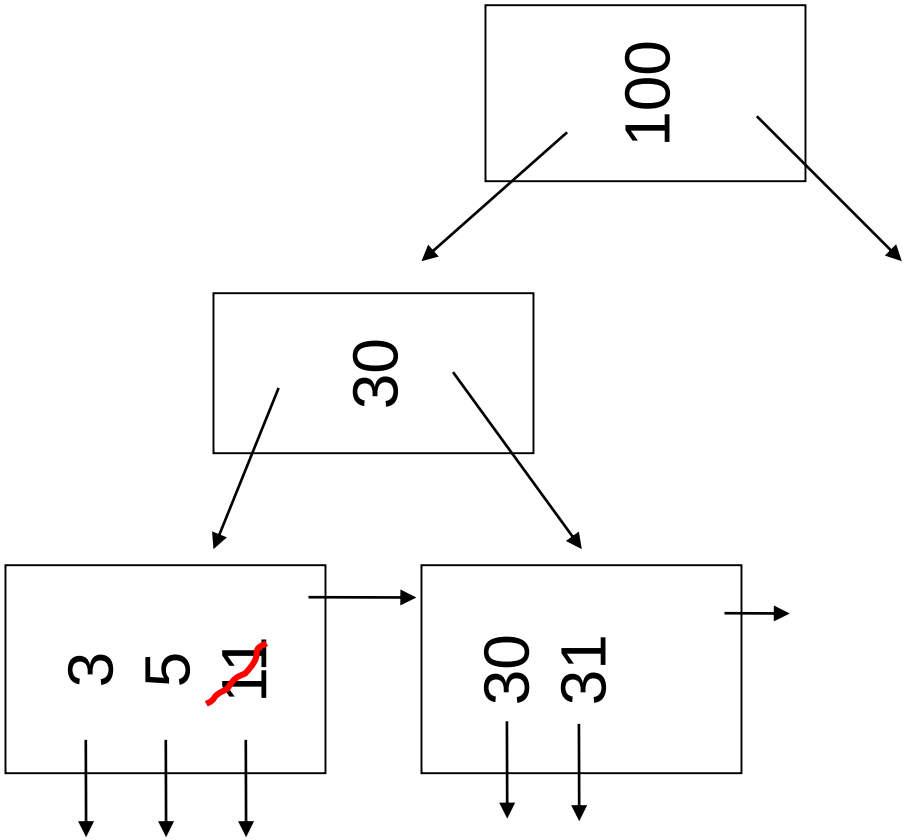
Ak vynecháme poslednú hodnotu z koreňa, tento zanikne. Novým koreňom sa stane jeho jediný syn. (D)

Zlučovanie uzlov je náročné na implementáciu, takže sa často neimplementuje. Preto indexové súbory majú po opakovanom vkladaní a vynechávaní tendenciu rásť.

B+ stromy: vynechanie (A) simple case

Delete 11

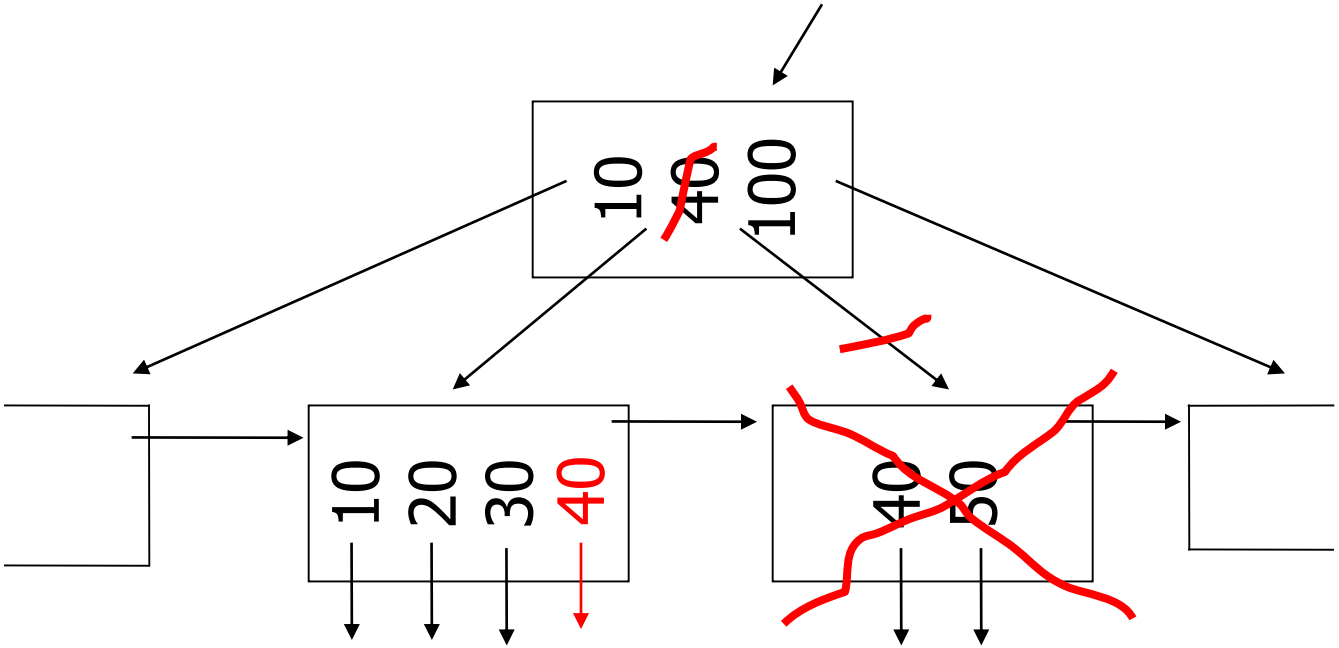
n=3



B+ stromy: vynechanie (B) coalesce with sibling

Delete 50

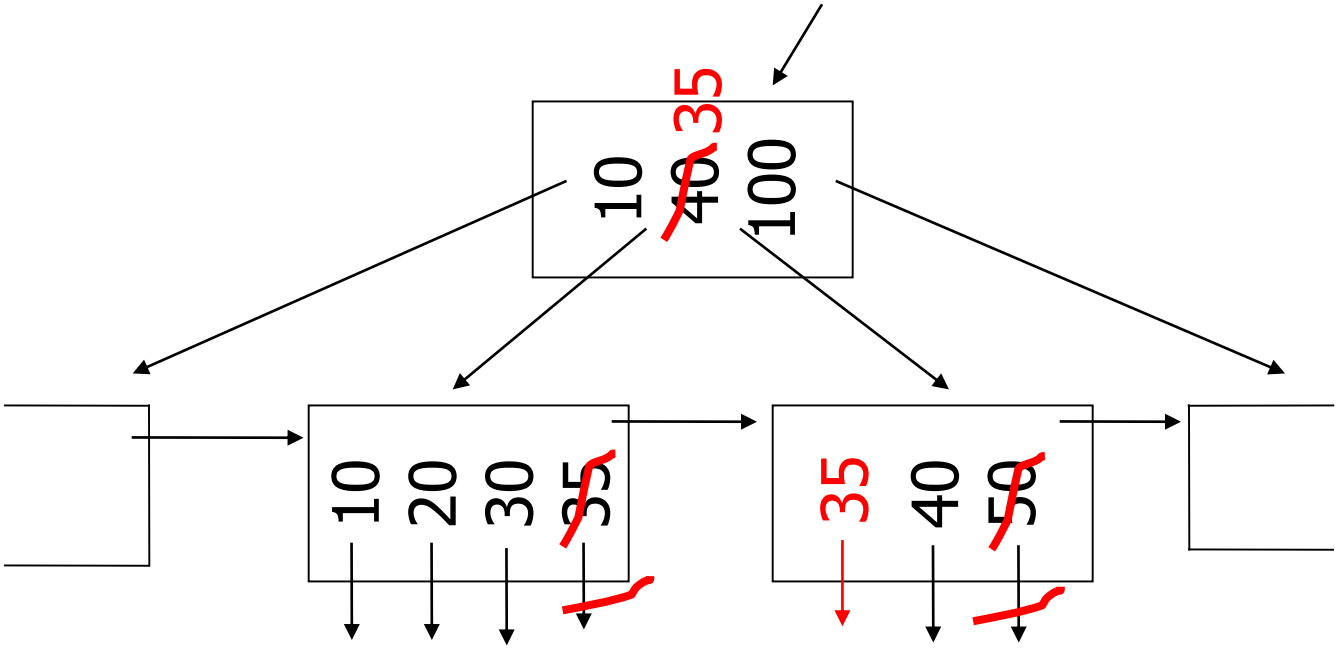
n=4



B+ stromy: vynechanie (C) redistribute keys

Delete 50

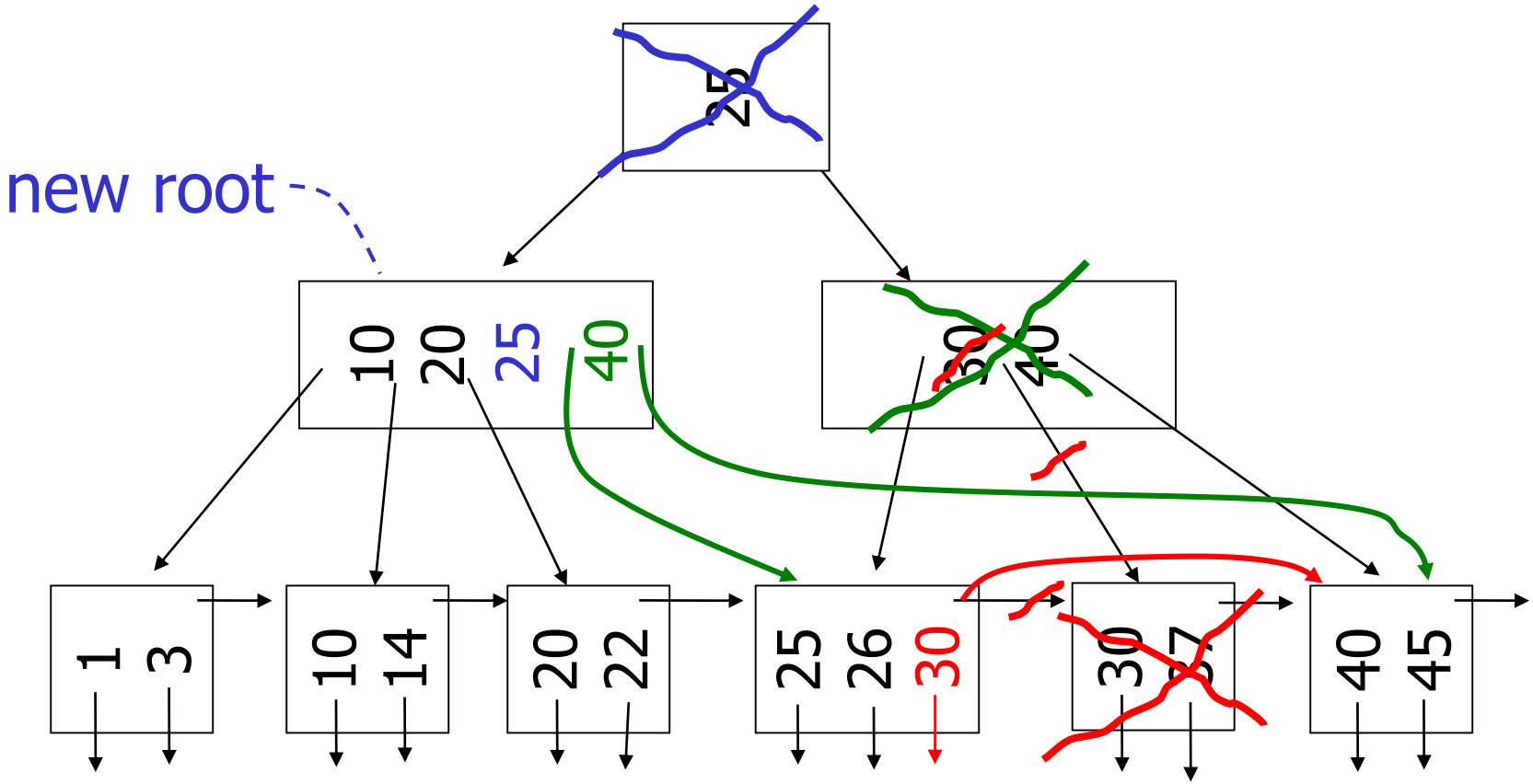
n=4



B+ stromy: vynechanie (D) non-leaf coalesce

Delete 37

n=3



B^+ stromy: prehodnotenie

Výhody:

• vhodné pre sekvenčné prehľadávanie aj pre prehľadávanie podľa kľúča (3-4 diskové operácie)

• rýchle vkladanie a vynechávanie (zriedka vzniká potreba reorganizácie stromu)

Nevýhody:

• relatívne náročná implementácia (hoci pravdepodobne jednoduchšia ako implementácia B stromov)

• pamäťový overhead (oproti sekvenčným indexom treba pridať dodatočné stupne)

Hashovanie

Idea: transformácia kľúča **key→h(key)**

Príklad hashovacej funkcie

- Key = ‘x_1 x_2 … x_n’ môžeme považovať za pole bytov dĺžky *n*
- Všetky možné stringy rozdelíme do *B* bucketov hashovacou funkciou (x_1 + x_2 + ….. x_n) modulo *B*

Iný príklad (hashovacia funkcia zachovávajúca usporiadanie):

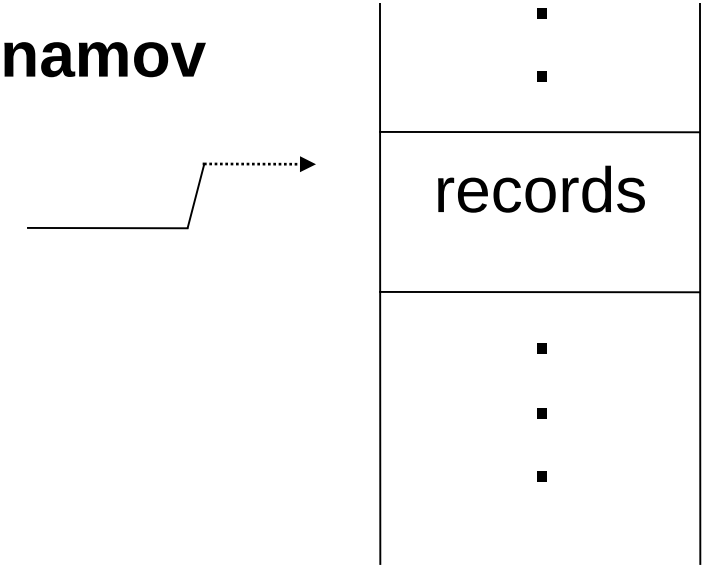
$h(k)= \lfloor B*(k-x)/(b-a) \rfloor$, kde $a \leq k \leq b$

Definícia (inžinierska). **Ideálna hashovacia funkcia** je taká, pre ktorú očakávaný počet záznamov v každom bucket e je rovnaký (cez všetky okamžité stavy databázy)

Hashovanie: priama vs. nepriama adresácia záznamov

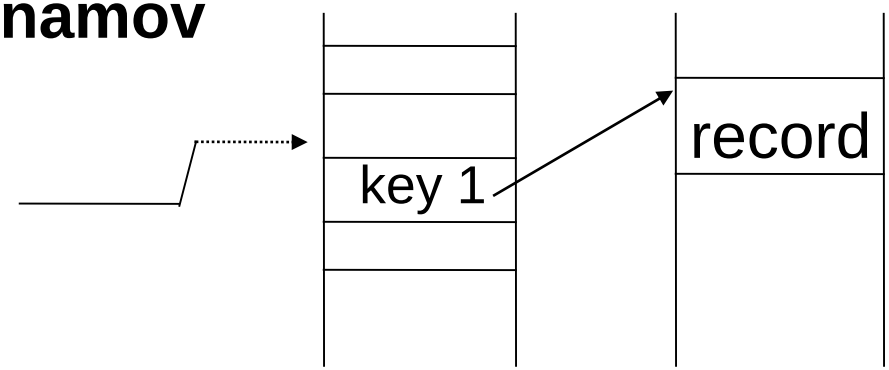
Priama adresácia záznamov

$key \rightarrow h(key)$



Nepriama adresácia záznamov

$key \rightarrow h(key)$



Index

Hashovaný súbor: directory, preplnenie

INSERT:

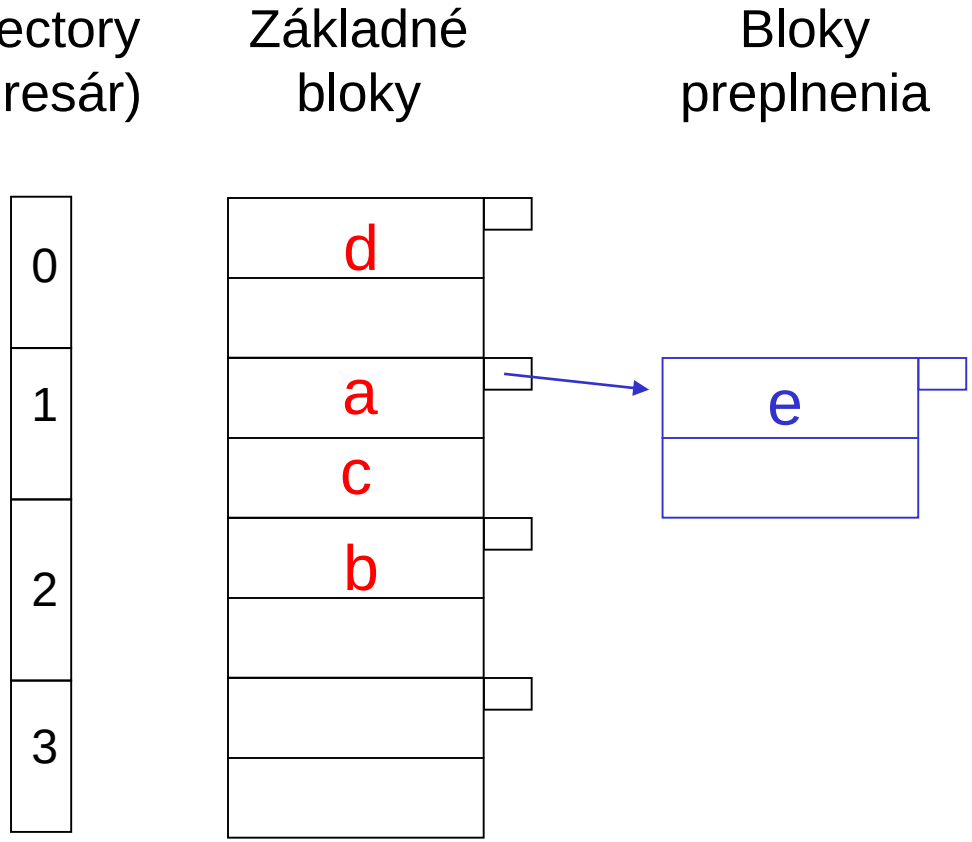
h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0

h(e) = 1



Dynamické hashovanie: využitie priestoru

Využitie priestoru v percentách:

U = # použitých blokov / # všetkých blokov

U by malo byť medzi 50% a 80%

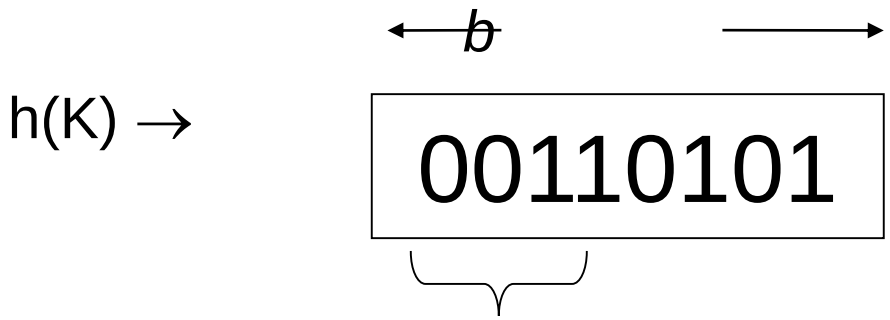
- U < 50% znamená plytvanie pamäťou, a plytvanie časom (pri diskových prenosoch sa prenášajú skoro prázdne bloky)
  → reorganizácia (spájanie blokov)
- U > 80% znamená dlhé sekvencie blokov preplnenia. Efektivita hashovania je vtedy porovnateľná s efektivitou sekvenčného indexu
  → reorganizácia (zdvojnásobenie pamäťovej štruktúry)

Dva spôsoby reorganizácie:

- Rozšíriteľné hashovanie
- Lineárne hashovanie

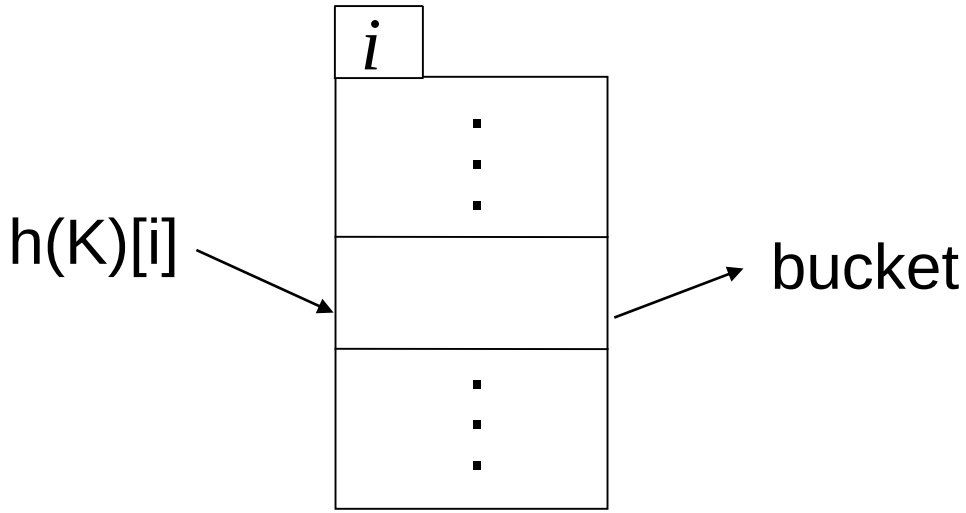
Rozšíriteľné hashovanie

(a) Použi len prvých *i* bitov z *b* bitov, ktoré vracia hashovacia funkcia



použi prvých *i* bitov → a pri preplnení zvýš i

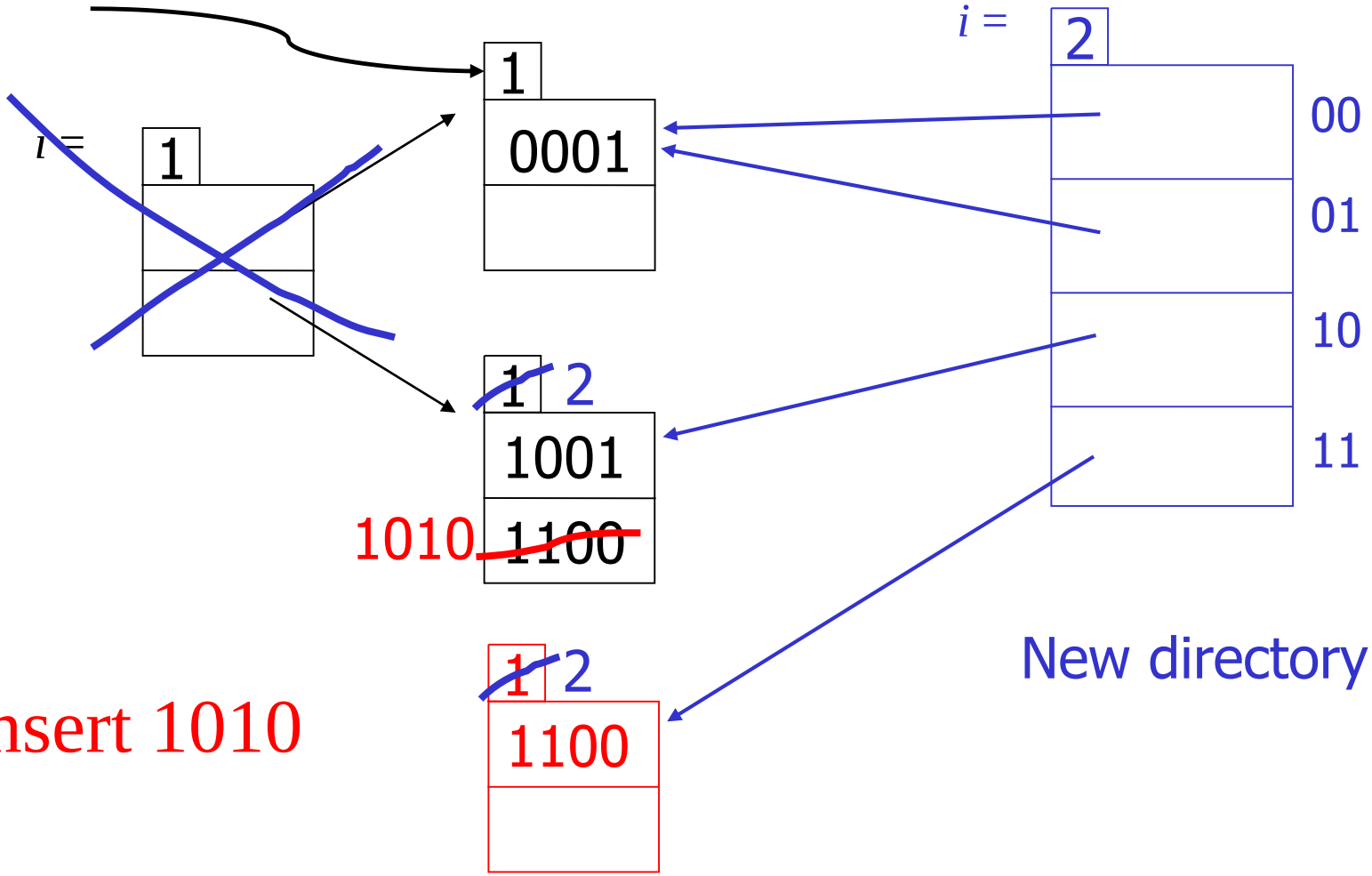
(b) Použi adresár



Rozšíriteľné hashovanie: vkladanie

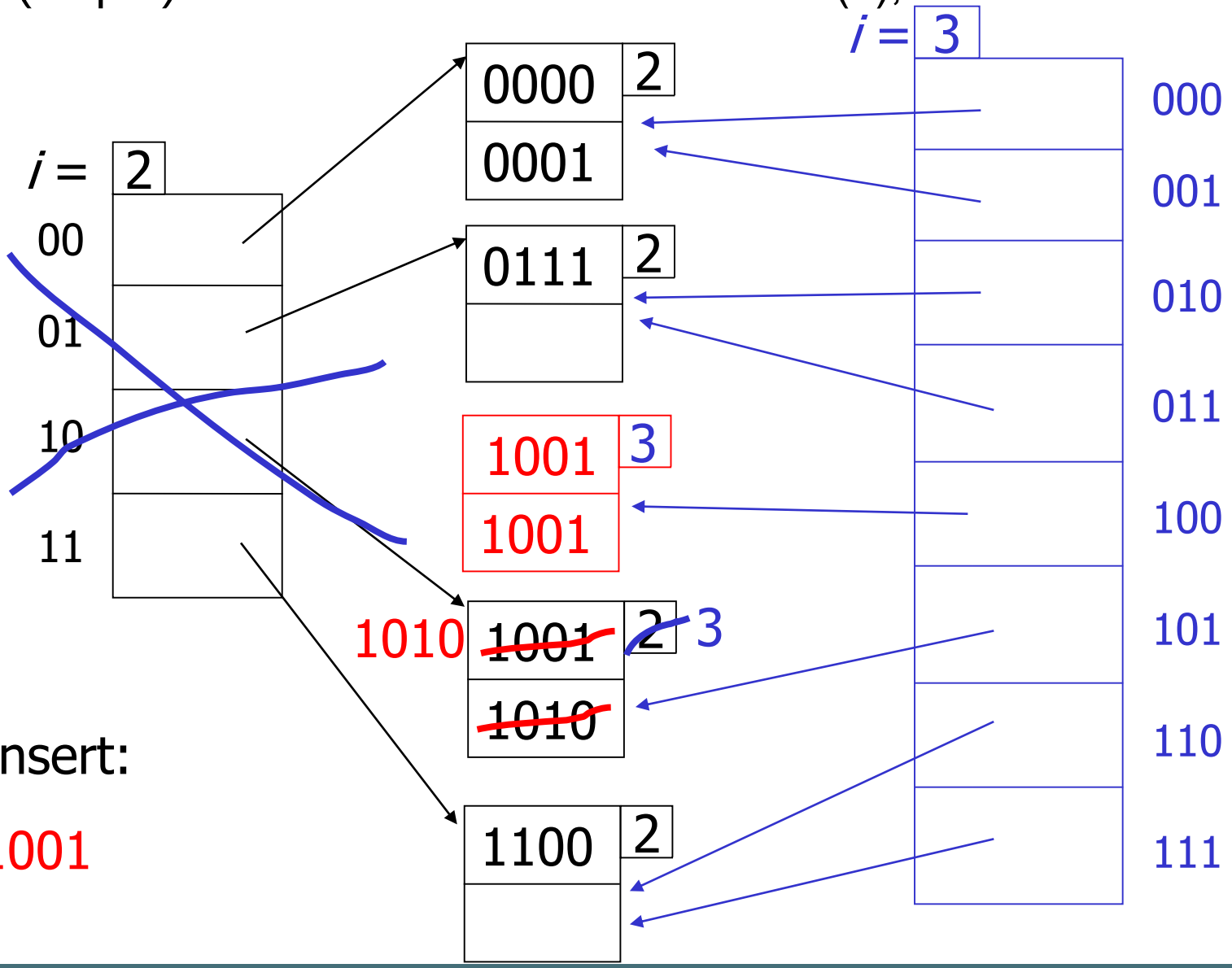
Príklad (Gupta): 4-bitová hashovacia funkcia h(k), 2 kľúče / bucket

Nub' -> Number of bits použitých
na určenie bloku



Rozšíriteľné hashovanie: vkladanie

Príklad (Gupta): 4-bitová hashovacia funkcia h(k), 2 kľúče / bucket



Lineárne hashovanie

Základná myšlienka: vyhnúť sa réžii s adresárom:

- na adresár stačí jeden bit (existuje/neexistuje blok)
- adresár si netreba pamätať, stačí si pamätať adresu posledného použitého bloku (t.j. momentálnu maximálnu adresu)

Nekonečná trieda hashovacích funkcií: $h_{p,q}(K)=K \bmod (2^q * p)$

Nech $N = 2^q * p$. Potom $h_{p,q}(K)=h_N(K)=K \bmod (N)$

Adresár: A = array[1..N] of bits

Platí (dôležité pre vyhľadávanie a vkladanie):

K mod 2N = buď K mod N, alebo (K mod N) + N

Lineárne hashovanie

Vyhľadanie bloku (resp. kľúča):

```
n = N;
k = K mod N; /* hashovacia funkcia */
while ((k > p) and (not A[k]))
{
  n = n / 2; /* celočíselné delenie */
  k = k – n;
}
```

Lineárne hashovanie

Vloženie záznamu:

Ak blok k nie je plný:
vlož záznam do bloku k;

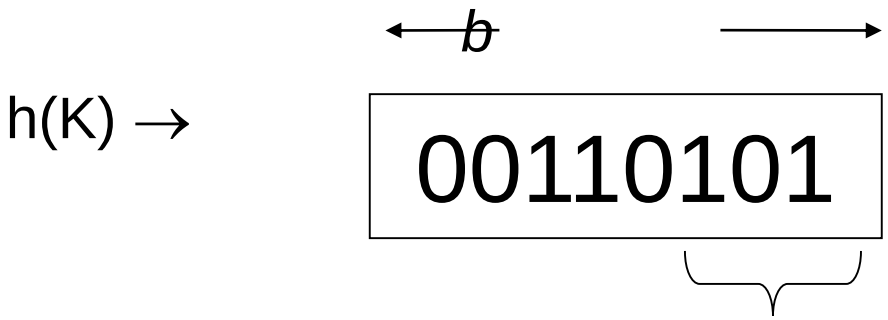
Inak, ak n < N:
vlož záznam do bloku k + n;

Inak zdvojnásob adresár A:

```
for i = N downto 0
{
  A[n + i] = FALSE;
}
A[k + n]=TRUE;
vlož záznam do bloku k + n;
N = 2 * N; /* modifikácia hashovacej funkcie */
```

Lineárne hashovanie

(a) Použi len posledných *m* bitov z *b* bitov, ktoré vracia hashovacia funkcia

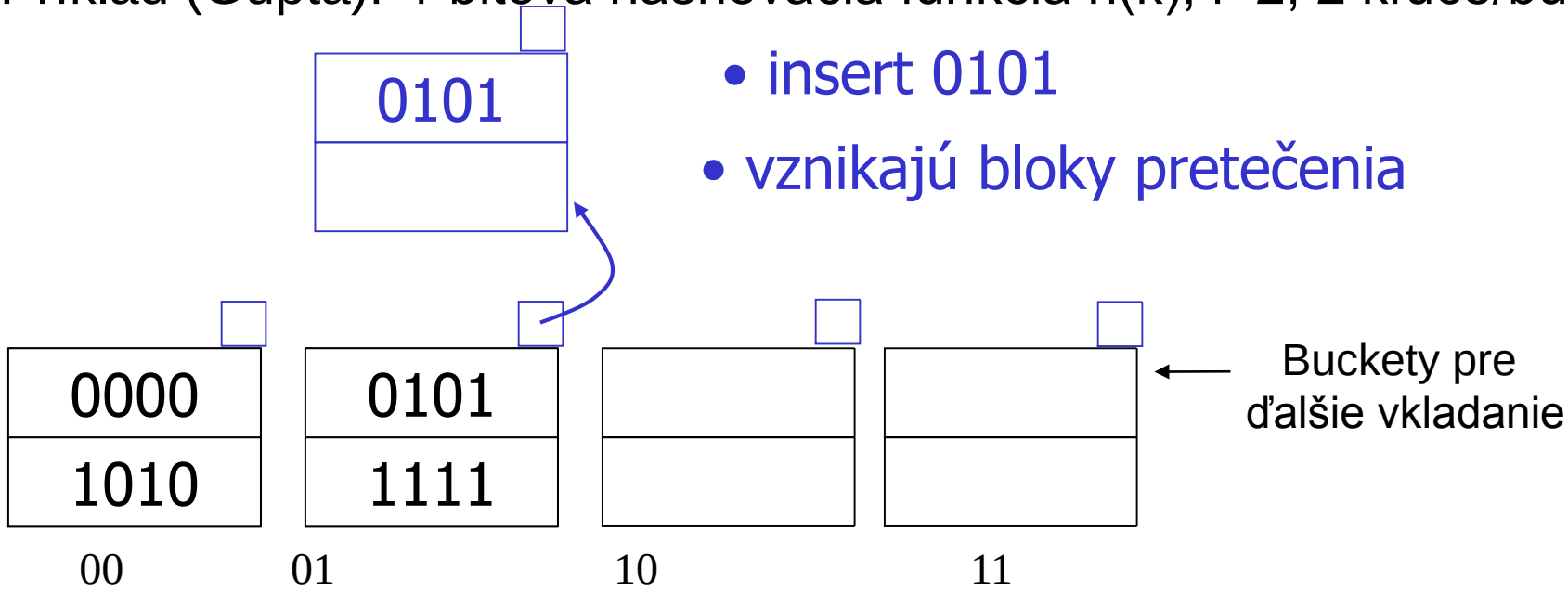


použi prvých *m* bitov → a pri preplnení zvýš *m*

(b) Pamätaj si len adresu posledného bloku

Lineárne hashovanie

Príklad (Gupta): 4-bitová hashovacia funkcia h(k), i=2, 2 kľúče/bucket



m = 01 (max used block)

if h(k)[i] ≤ m, then

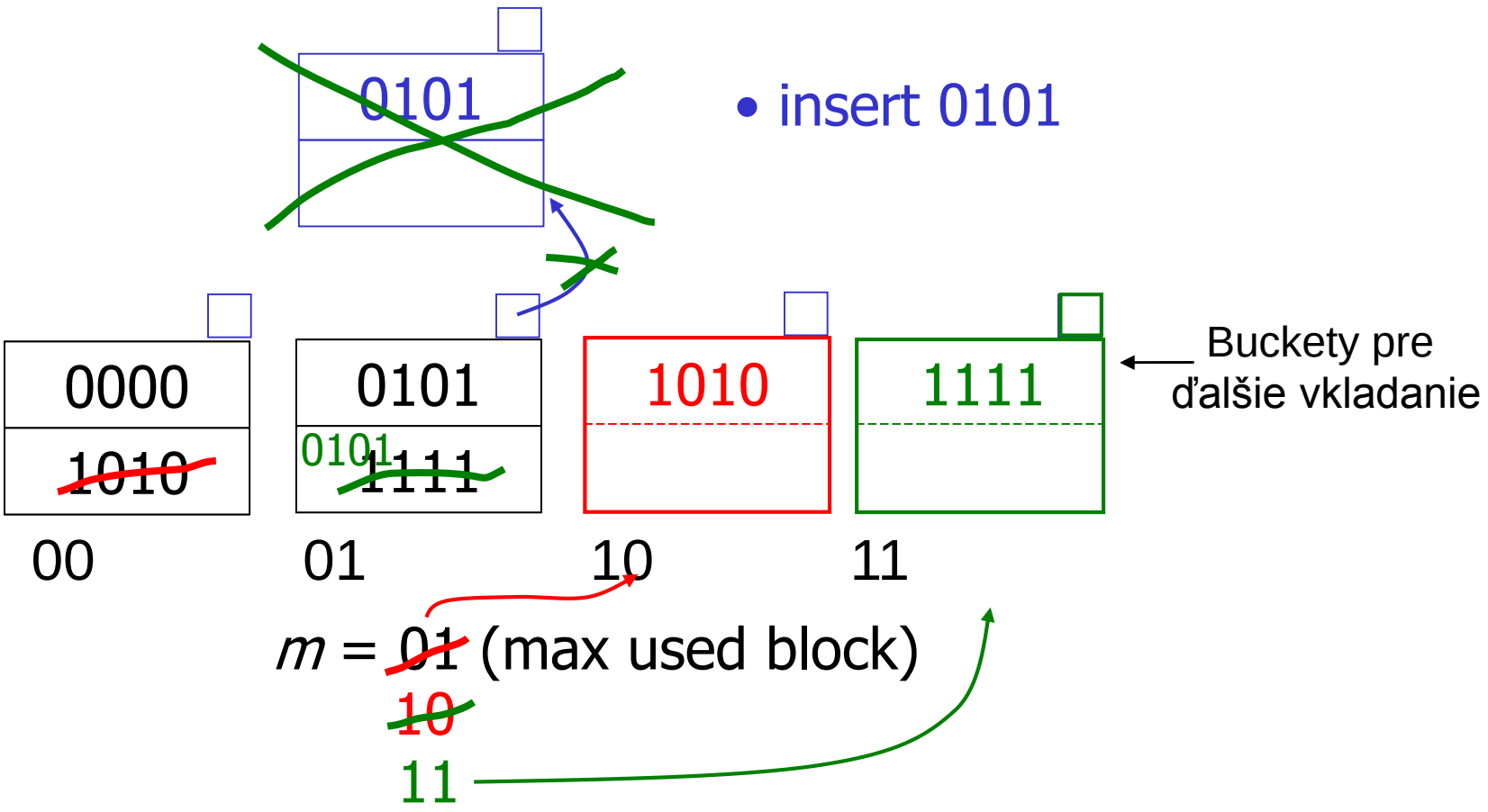
hľadaj v buckete h(k)[i]

else

hľadaj v buckete h(k)[i] - 2^{i-1}

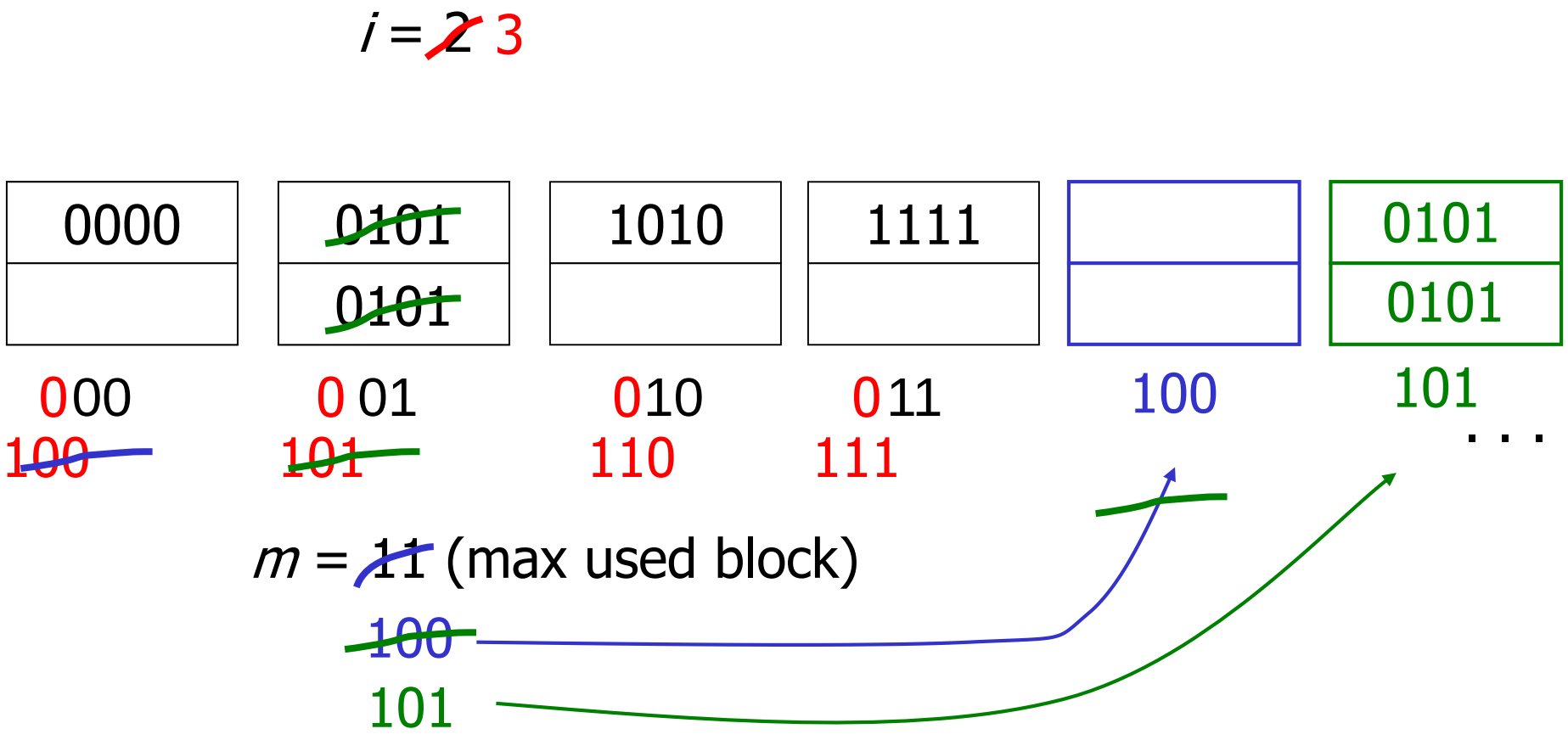
Lineárne hashovanie

Príklad (Gupta): 4-bitová hashovacia funkcia h(k), i=2, 2 kľúče/bucket



Lineárne hashovanie

Príklad (Gupta): 4-bitová hashovacia funkcia h(k), i=2, 2 kľúče/bucket



Cena reorganizácie

Predpokladajme, že dátová štruktúra veľkosti n sa dá vytvoriť v čase O(n log(n)). Počas existencie štruktúry (bez reorganizácie) sa vykoná O(n) operácií, pričom každá operácia trvá O(log(n)). Potom treba štruktúru reorganizovať. Reorganizácia zväčší veľkosť štruktúry na qn (kde q>1, napr. q=2).

Vtedy sa tá reorganizácia dokázateľne oplatí, lebo amortizovaná cena operácií ostane konštantná!

$$\lim_{n\to\infty}\frac{\sum_{i=0}^{n}\left(q\log(q)\right)^i}{nq^n}=\lim_{n\to\infty}\frac{\log(q)}{nq^n}\sum_{i=0}^{n}iq^i=const$$

/* Amortizovaná cena operácií */

$$\sum_{i=0}^{n}i\,q^i=q\sum_{i=0}^{n}i\,q^{i-1}=q\frac{\partial}{\partial q}\left(\frac{q^n-1}{q-1}\right)=\frac{(n-1)\left(q^{n+1}-q^n\right)}{(q-1)^2}$$

/* Výpočet tej sumy */

$$\lim_{n\to\infty}\frac{\log(q)}{nq^n}\frac{(n-1)\left(q^{n+1}-q^n\right)}{(q-1)^2}=\lim_{n\to\infty}\frac{(n-1)\log q}{n(q-1)}=\frac{\log q}{q-1}$$

/* Výpočet tej limity */

QED

Hashovanie vs. B^+ stromy

Hashovanie je výhodnejšie, ak sú časté dotazy na rovnosť kľúča

SELECT …

FROM R, S

WHERE R.A = 42

Teoreticky je očakávaný čas vyhľadávania hashovaného záznamu konštantný! (Bohužiaľ, iba teoreticky, lebo nájsť dobrú a zároveň rýchlu hashovaciu funkciu je problém.)

B^+ stromy sú výhodnejšie, ak sú časté intervalové dotazy na kľúč, (ale pomáhajú aj pri dotazoch na rovnosť a poskytujú veľmi dobré garancie aj pre najhorší prípad)

SELECT …

FROM R, S

WHERE R.A > S.B

Vybrané fyzické operátory: index scan

Algoritmus index scan (vyžaduje existenciu index-file podľa testovaného kľúča)

Nájdi prvý vyhovujúci záznam, ďalšie stačí hľadať v nasledujúcich blokoch.

cost = height + b + 1, kde height je hĺbka indexového stromu a b je počet blokov obsahujúcich vyhovujúce záznamy

Vybrané fyzické operátory: index scan

Iterator **IndexScan(R)**

```
open(R) {
  R.open();
}
close(R) {
  R.close();
}
next(R) {
  return R.index().next();
}
```

```
oscon=# explain select oid from pg_proc where oid=1;
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using pg_proc_oid_index on pg_proc  (cost=0.00..5.99
rows=1 width=4)
   Index Cond: (oid = 1::oid)
```

Vybrané fyzické operátory: hash join

Algoritmus join (hash join)

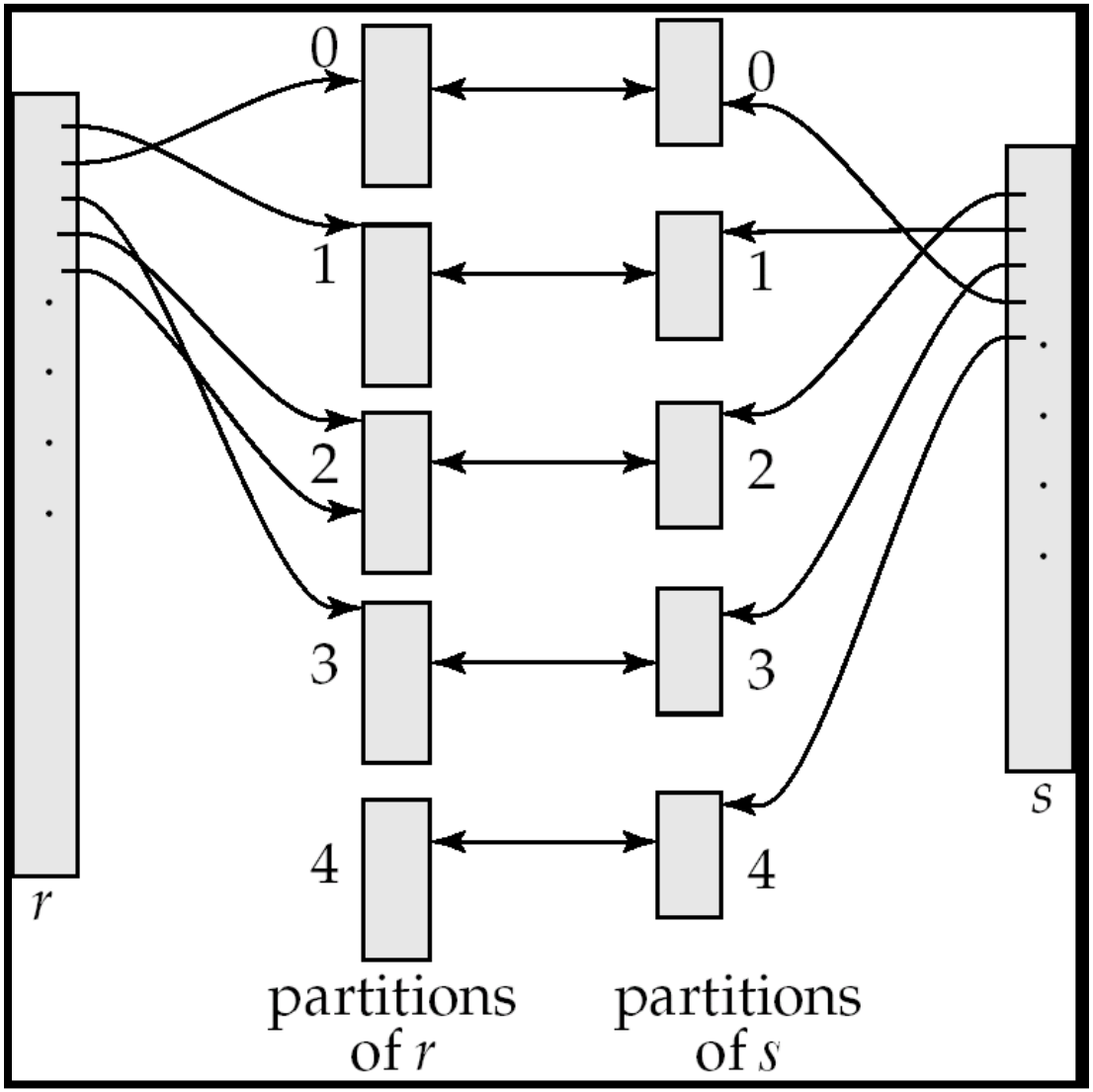
Tento algoritmus dovoľuje len joinovaciu podmienku na rovnosť (equijoin, resp. natural join)

Základná myšlienka: predpokladajme, že každý hashovaný bucket sa zmestí do pamäte. (V hashovanom buckete sú záznamy s rovnakou hodnotou hashovacej funkcie.) V takom prípade stačí opakovane ukladať do pamäte hashované buckety R a S, lebo *len* pre záznamy s rovnakým hashovacím kľúčom *môže byť* joinovacia podmienka splnená

(Ak nie je pravda, že každý hashovaný bucket zmestí do pamäte, treba toto opakovať.)

Vybrané fyzické operátory: hash join

Príklad (Silberschatz et al.):



Vybrané fyzické operátory: hash join

Algoritmus join (hash join):

1. Vytvor hashované buckety R_1, R_2, ..., R_p z relácie R, a podobne vytvor hashované buckety S_1, S_2, ..., S_p
2. Pre každý bucket i, i=1,...,p:

- načítaj jeden bucket R_i a vytvor preň nový hash (index)
- pre každý záznam bucketu S_i otestuj na základe novej hashovacej funkcie, či sa môže nachádzať v R_i. Ak áno, nájdi záznamy, ktoré joinujú a zapíš do výstupného bloku. Ak je výstupný blok plný, zapíš ho na disk

cost=3*(B(R)+B(S))+4*p

(Cena písania na disk tu nie je zahrnutá, lebo veľkosť joinu závisí od joinovacej podmienky.)

Dualita triedenia (stromy) a hashovania

- Paradigma rozdeľuj a panuj
  (“rozdeľovanie = neporiadok, usporiadavanie = poriadok”)
  - Stromy: fyzické rozdeľovanie, logické usporiadavanie
  - Hashovanie: logické rozdeľovanie, fyzické usporiadavanie
- Spracovanie dlhých vstupov (väčších ako operačná pamäť)
  - Stromy: viacfázové spájanie
  - Hashovanie: viacfázové rozdeľovanie
- Vstupno-výstupné vzory (I/O patterns)
  - Stromy: sekvenčný zápis, náhodné čítanie (merge)
  - Hashovanie: náhodný zápis, sekvenčné čítanie (buckety)