Optimalizace

přednáška č. 5

EVROPSKÁ UNIE

Indexy

- Jedná se o pomocnou datovou strukturu, která slouží k urychlení základních operací nad záznamy
- Otázky:
  - Jak jsou uložena data v indexu, aby zrychlovala získávání dat?
  - Co vše je třeba uložit?

EVROPSKÁ UNIE esf MŠMT

Jedna možnost

1. Uložit celý soubor setříděný podle klíče, nad kterým budeme vyhledávat
2. Uložit pomocnou datovou strukturu tak, aby naváděli k datům, která hledáme

EVROPSKÁ UNIE

Vlastnosti indexů

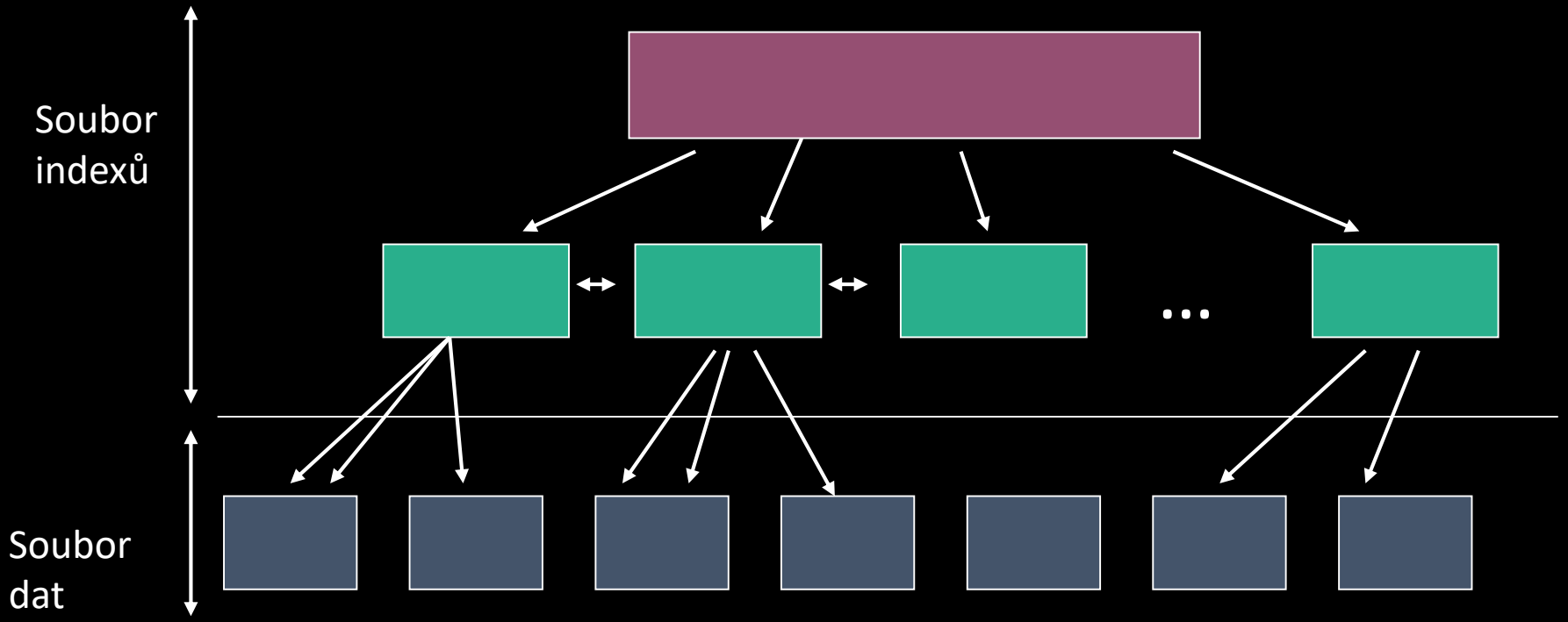
- Jak jsou data uložena?

- Clustered x Unclustered indexy
- Klastrované x neklastrované indexy

EVROPSKÁ UNIE

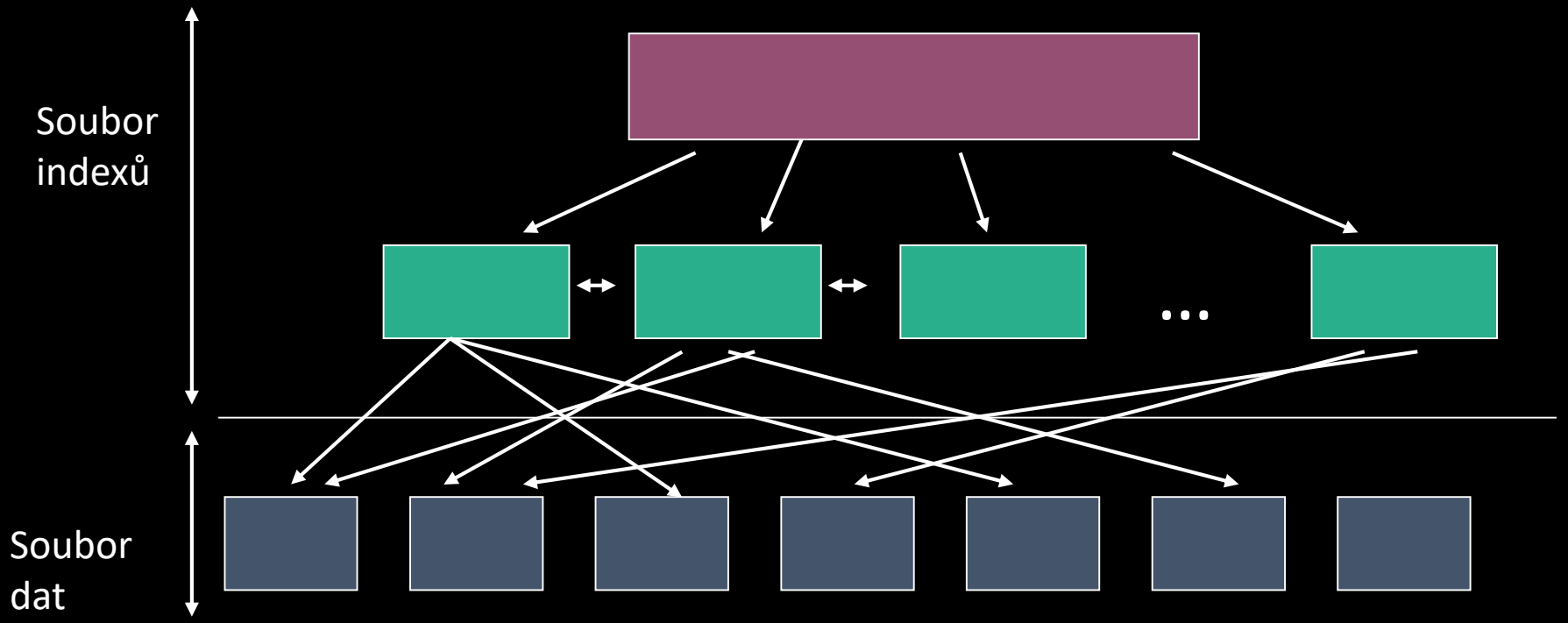
Clustered indexy

- Pokud jsou data uložena stejně nebo skoro stejně, jako jsou uloženy záznamy v indexu



Unclustered indexy

- Záznamy nejsou setříděny podle žádného klíče



Kdy použít který index?

- Co je třeba k udržování setříděného souboru?
- Jaké jsou rychlosti při vlastním vyhledávání?

EVROPSKÁ UNIE esf MŠMT

Clustred index

- Bude vyžadovat častou reorganizaci = je možné rezervovat „více“ místa a pak do něj vkládat než dojde a je nutné celý index reorganizovat
- Clustered indexy jsou náročné na údržbu pokud se často vkládá

EVROPSKÁ UNIE esf MŠMT

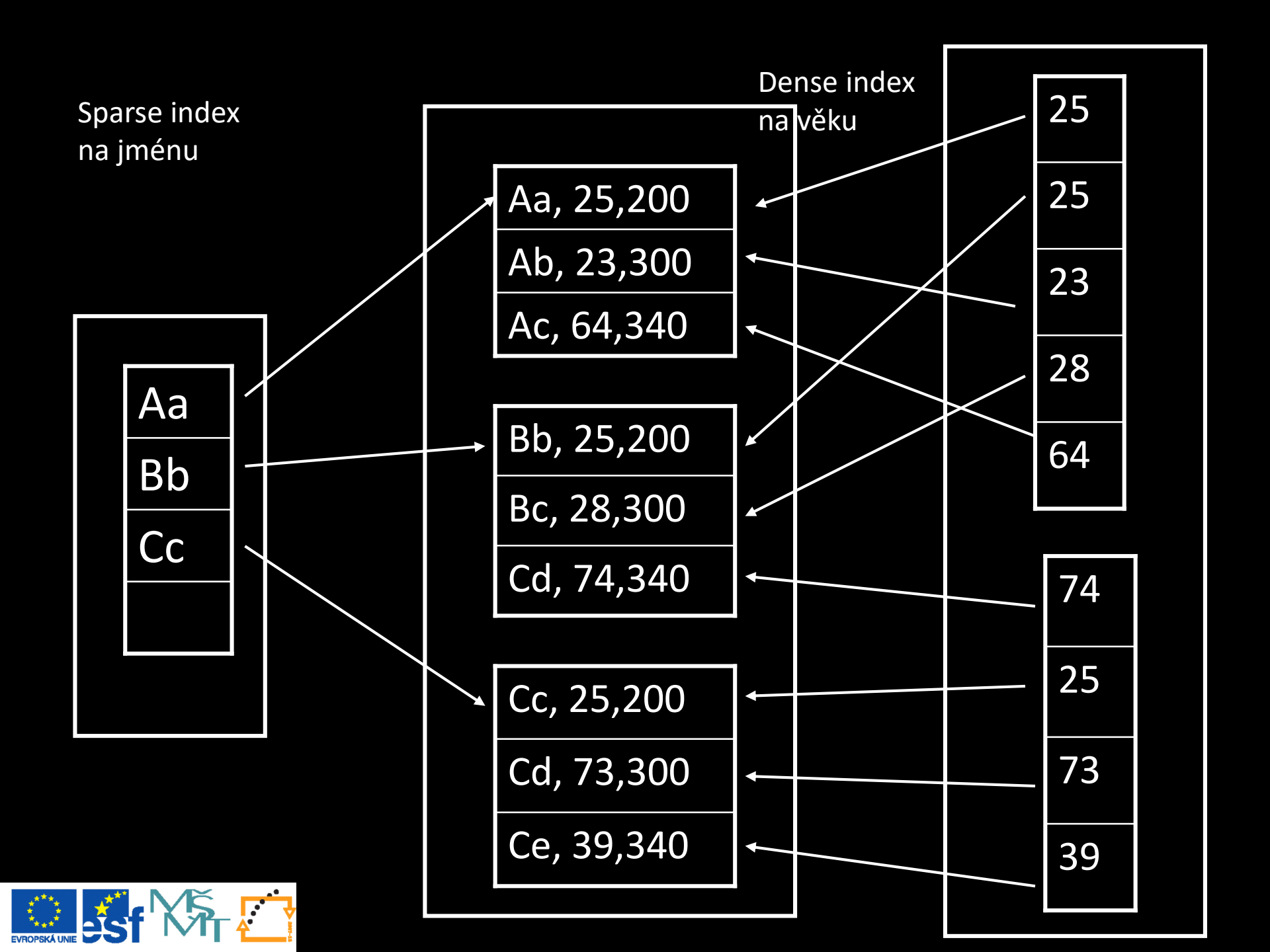
Unclustered indexy

- Můžeme mít více unclustred indexů na jednou souboru
- Jednoduchá údržba
- Jak to dopadne při tzv. dotazu na rozsah?

Dense x Sparse indexy

- Dense = pokud index obsahuje alespoň jeden zápis pro záznam v datovém souboru
- Sparse = Index obsahuje záznam pouze pro jednu stránku záznamů

EVROPSKÁ UNIE


Sparse index
na jménu
Aa
Bb
Cc
Aa, 25,200
Ab, 23,300
Ac, 64,340
Bb, 25,200
Bc, 28,300
Cd, 74,340
Cc, 25,200
Cd, 73,300
Ce, 39,340
Dense index
na věku
25
25
23
28
64
74
25
73
39

Primární a sekundární index

- Primární index = je vytvořen na atributech, které obsahují primární klíč
- Sekundární index = index, který není primárním indexem

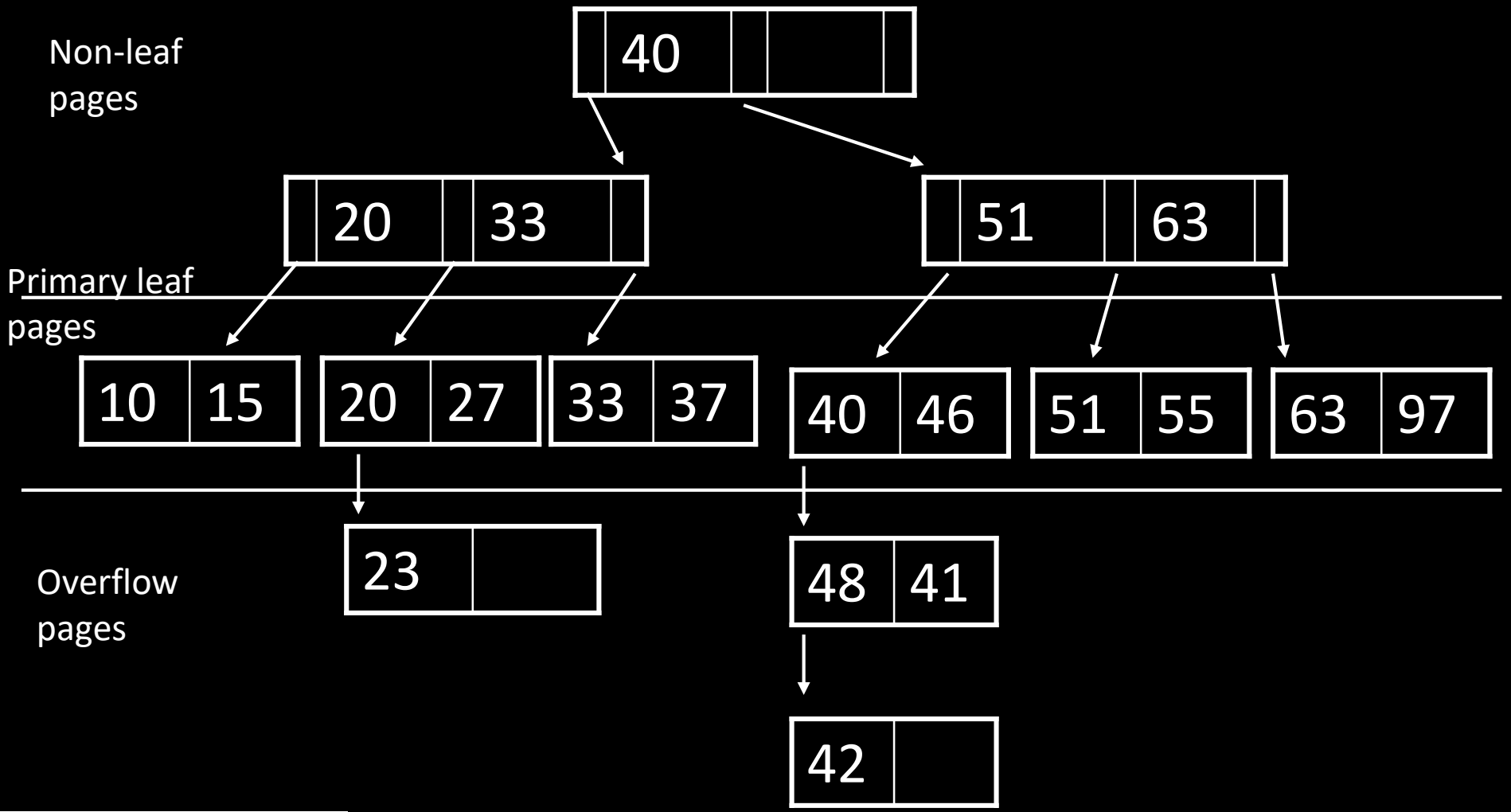
EVROPSKÁ UNIE

Typy indexů

- Stromové struktury
  - ISAM – statická struktura
  - B+ - dynamické stromy
- Hash indexy

EVROPSKÁ UNIE

ISAM


Non-leaf pages
40
20
33
51
63
Primary leaf pages
10
15
20
27
33
37
40
46
51
55
63
97
Overflow pages
23
48
41
42

ISAM

- Počet diskových operací = hloubce stromu
- Hlavním problémem je přetékání při vkládání nových záznamů

EVROPSKÁ UNIE

B+ stromy

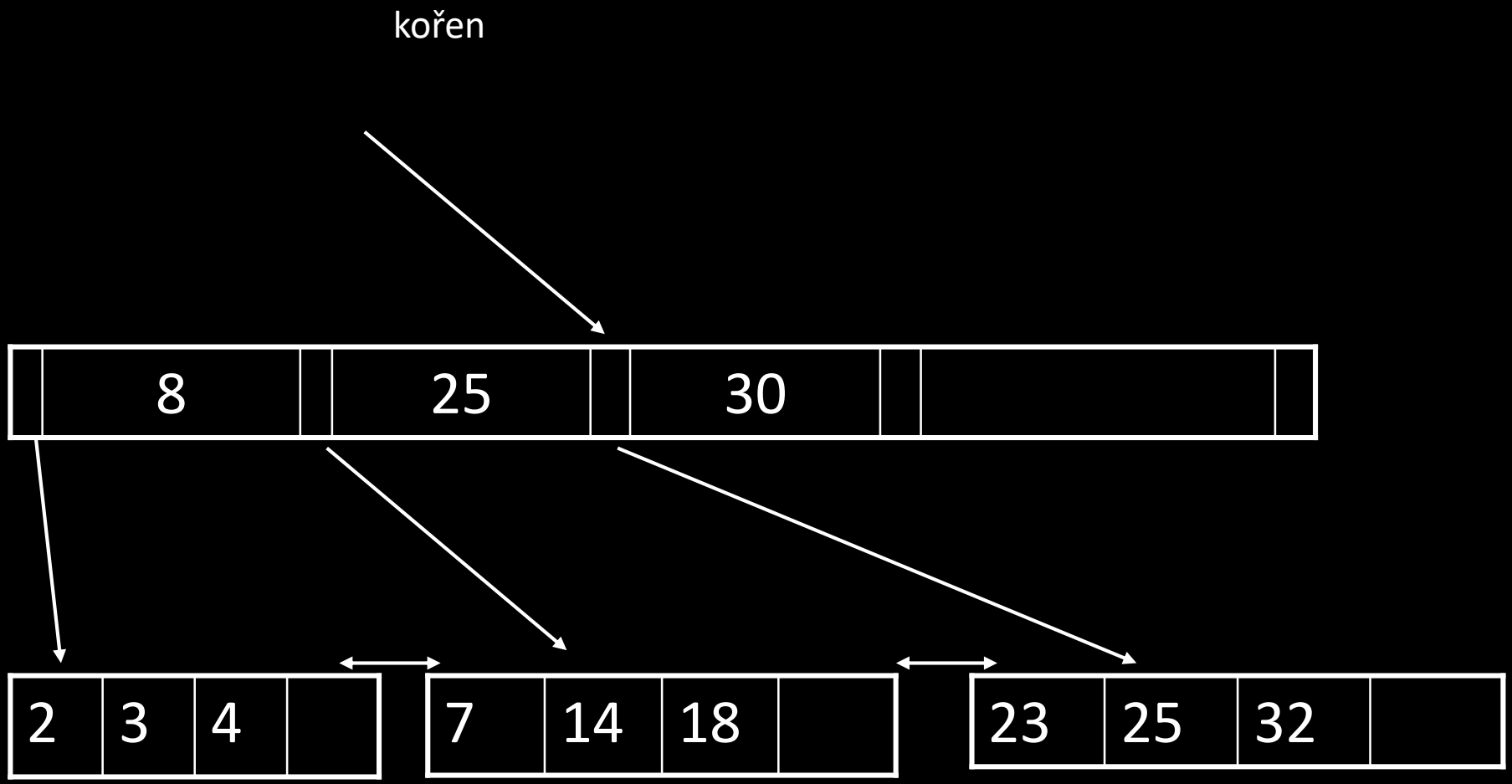
- ISAM trpí problémem přetečení listových prvků
- Proto byla navržena nová dynamické struktura B+ stromu

EVROPSKÁ UNIE

B+ stromy - vlastnosti

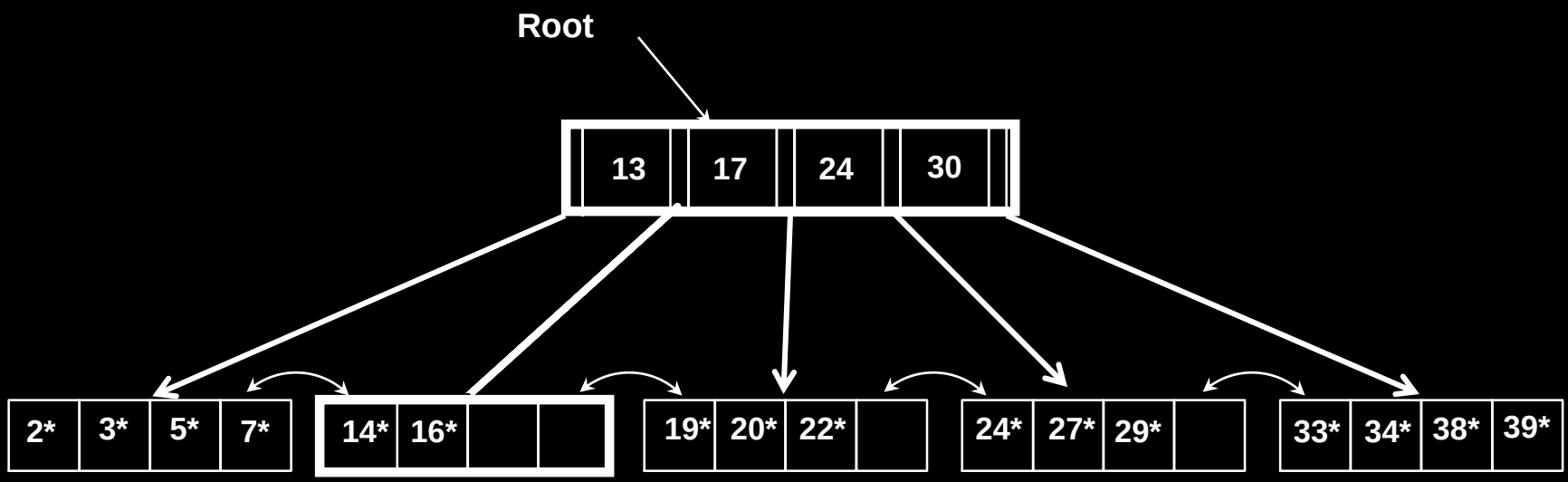
- Operace vložení a smazání jsou vážené
- Minimální obsazenost 50% pro každý uzel mimo root (mazání může toto pravidlo někdy porušovat, protože soubory častěji rostou)
- Vyhledání prosté sekvenční projití od kořene k listu

EVROPSKÁ UNIE esf MŠMT


kořen
8
25
30
2
3
4
7
14
18
23
25
32

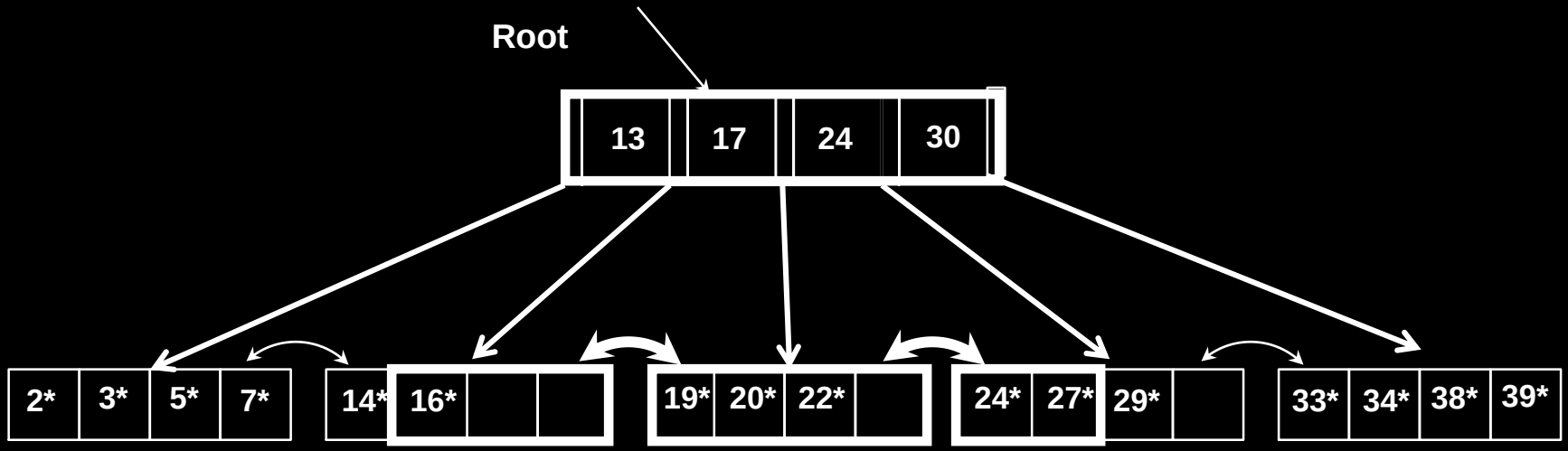
B+ Tree Vyhledávání

- Začneme v root a jdeme podle směrníků
- např. 15*…



B+ Tree Rozsahové dotazy

- pro záznamy v rozsahu [15,28].
  - hledáme 15*.
  - následujeme pomocné ukazatele.



Operace na B+ stromech

- Je nutné udržovat strom dle definice
- Vyvažování
- ….

EVROPSKÁ UNIE esf MŠMT

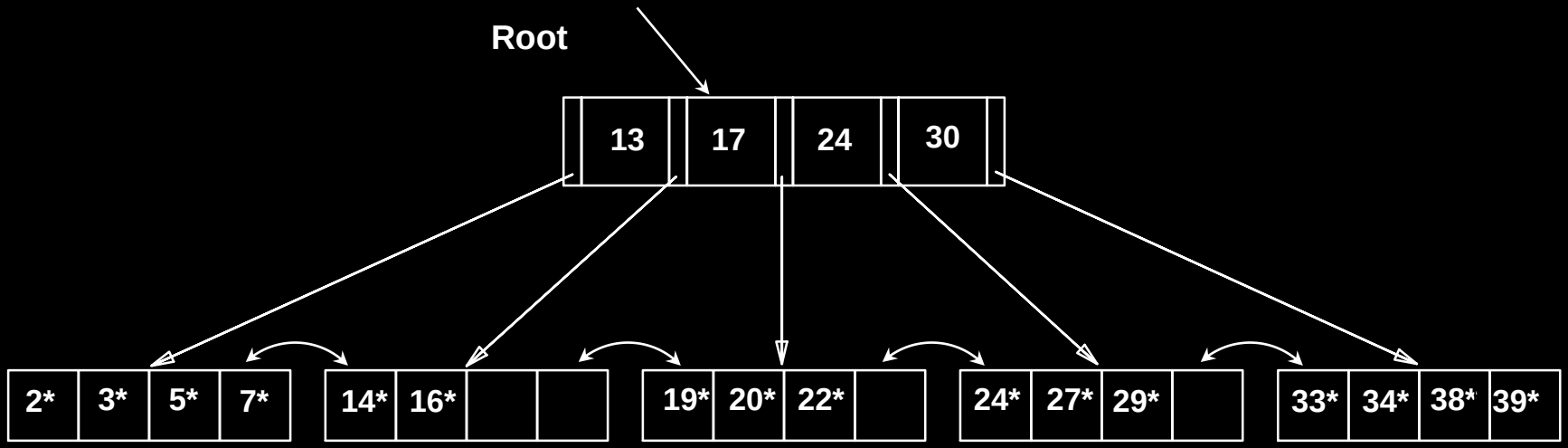
Vložení dat do B+ Stromu

- Najít požadovaný list *L.*
- Vložit data do *L*.
  - Pokud *L* je volný/je v něm místo, hotovo
  - jinak, musíme rozdělit *L (na L a nový uzel L2)*
    - pokud je třeba redistribuovat záznamy, nakopírovat do nadřazeného uzlu prostřední klíč (stane se rozcestníkem.
    - vložit index ukazující na L2 do otce L.
- Může se to rekurzivně propagovat do celé větve stromu
- Strom tímto roste

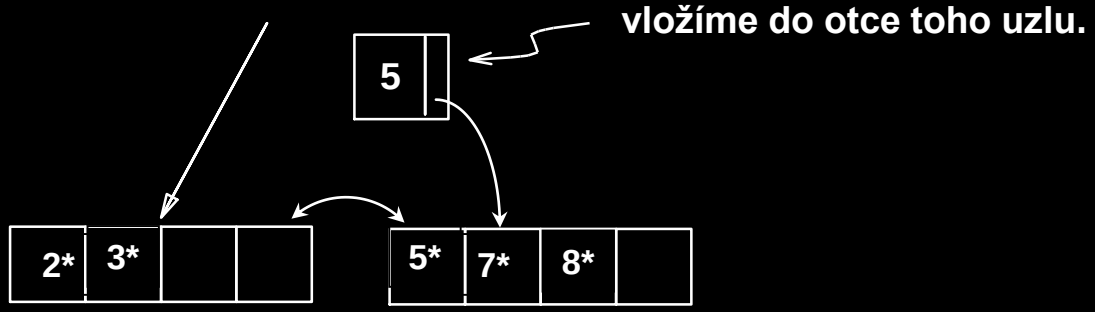
EVROPSKÁ UNIE esf MŠMT

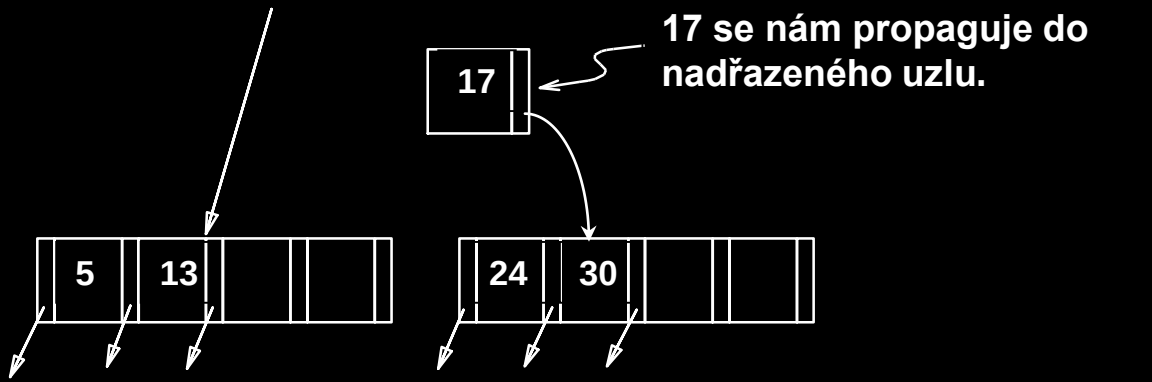
Vložení 8*

- Dojde k rozdělní.

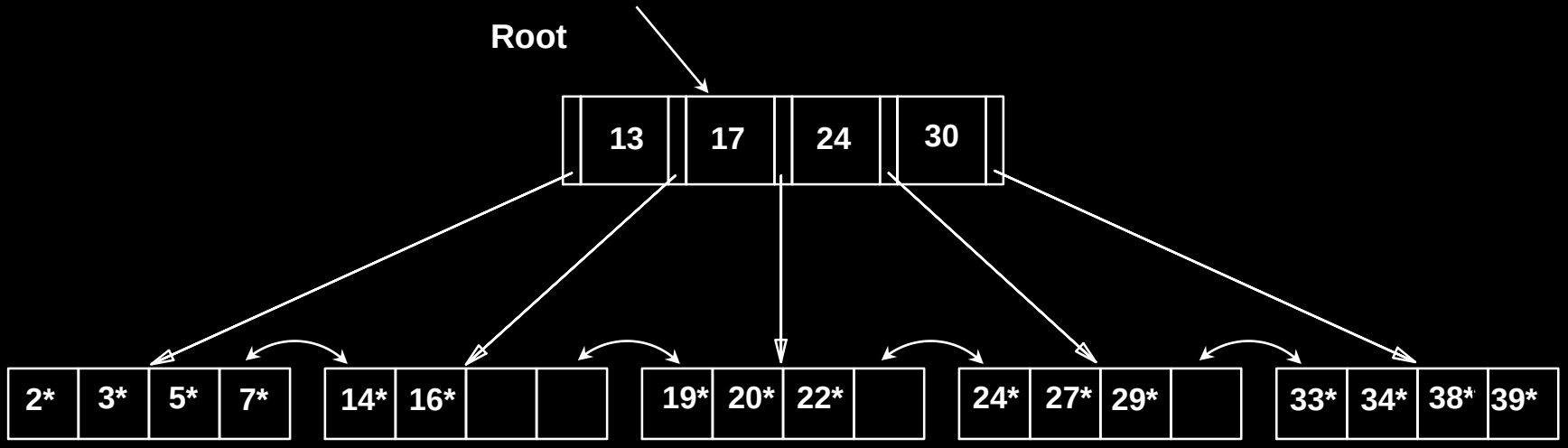


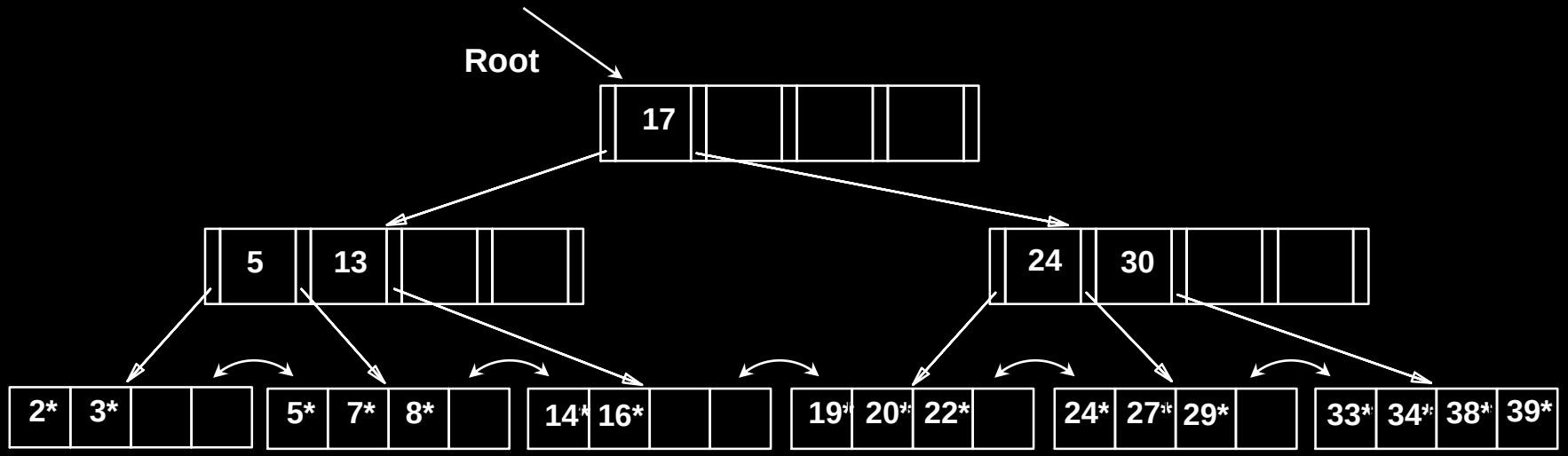
Pokračování, rozdělení





Po vložení 8*

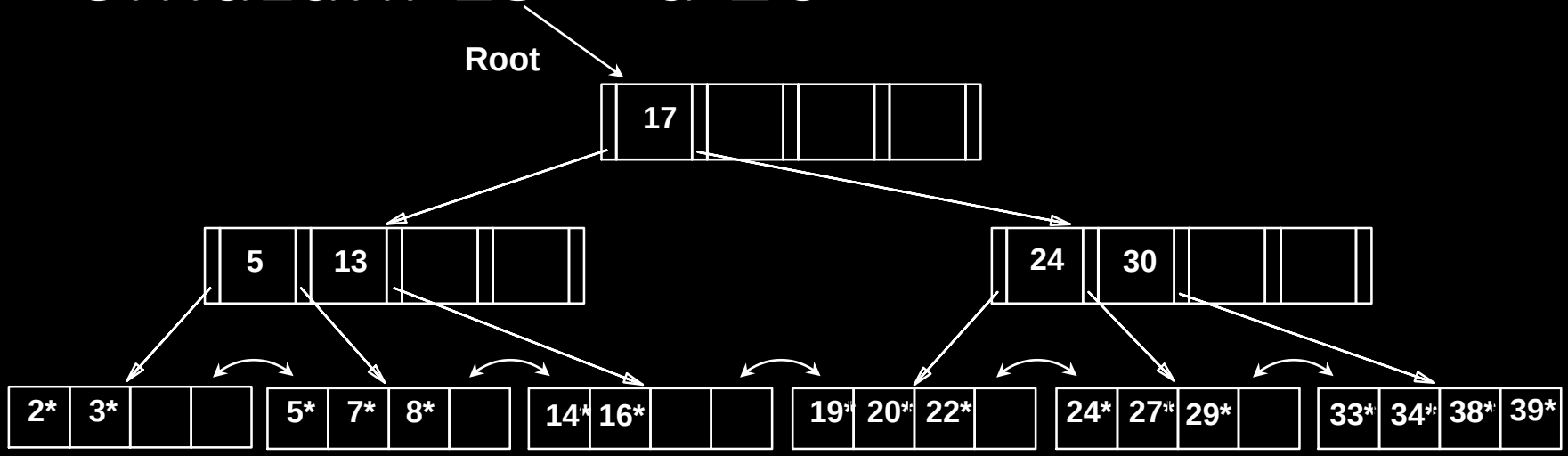

Root
13
17
24
30
2*
3*
5*
7*
14*
16*
19*
20*
22*
24*
27*
29*
33*
34*
38*
39*


Root
17
5
13
24
30
2*
3*
5*
7*
8*
14*
16*
19*
20*
22*
24*
27*
29*
33*
34*
38*
39*

Mazání z B++ stromu

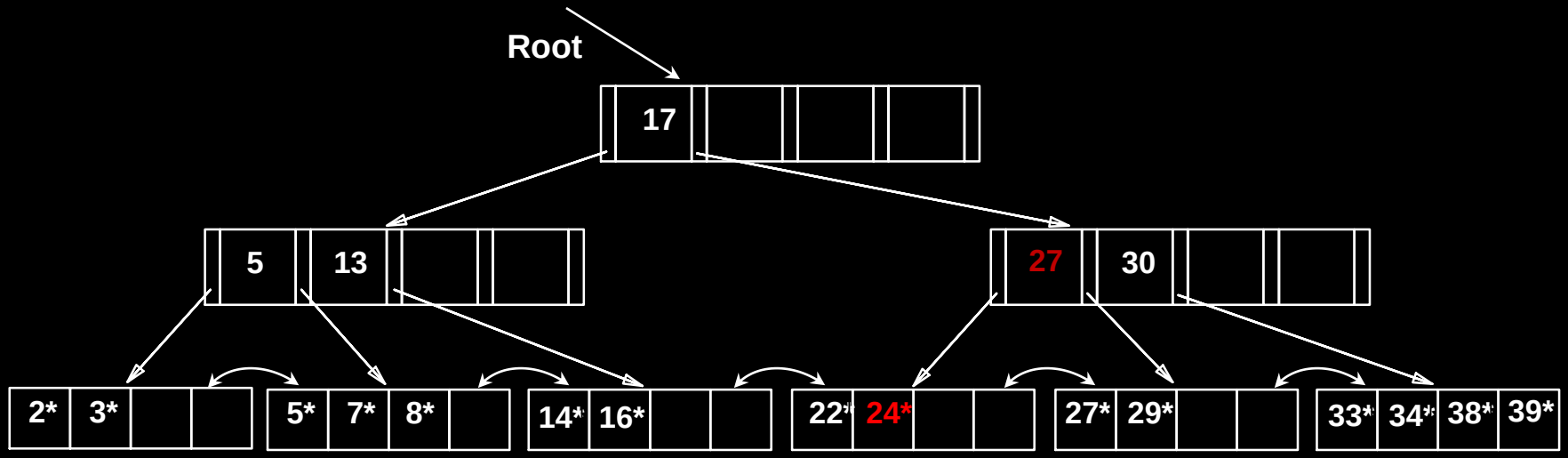
- Najít požadovaný list *L*.
- Odstranit záznam.
  - pokud je L nejméně z poloviny plný, hotovo
  - pokud má L pouze d-1 záznamů,
    - pokusit se o redistribuci, půjčit si záznamy od sousedních uzlů.
    - jinak spojit L a souseda.
- V případě spojování L a souseda je třeba smazat i záznamy ukazující na tyto dva uzly.
- Spojování mlže propagovat až do root a tím snížit hloubku stromu.

Smazání 19* a 20*



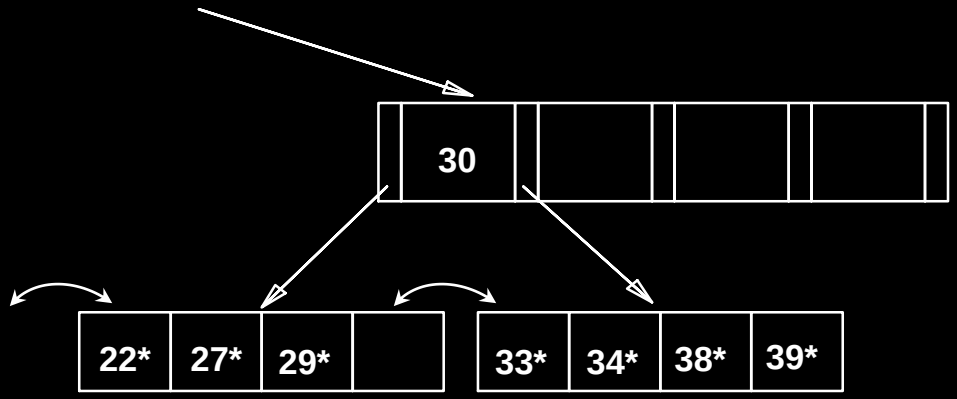
- 19* je snadné.
- 20* lze pomocí redistribuce.

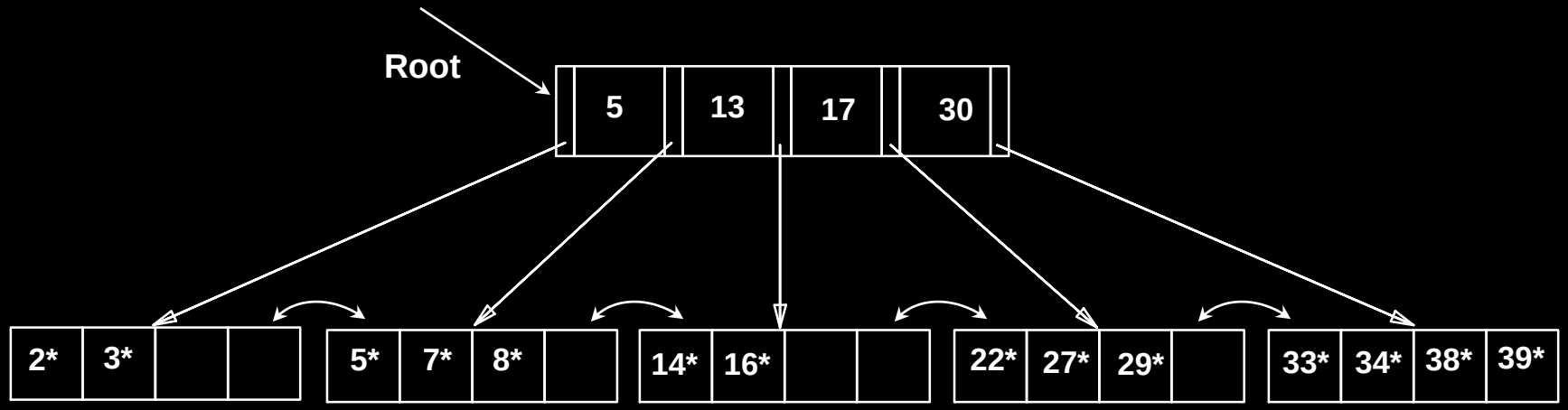
Po smazání 19* a 20*



- 27* byl zkopírován do nadřazeného uzlu
- pokud mažeme 24*…

mažeme 24*


30
22*
27*
29*
33*
34*
38*
39*


Root
5
13
17
30
2*
3*
5*
7*
8*
14*
16*
22*
27*
29*
33*
34*
38*
39*

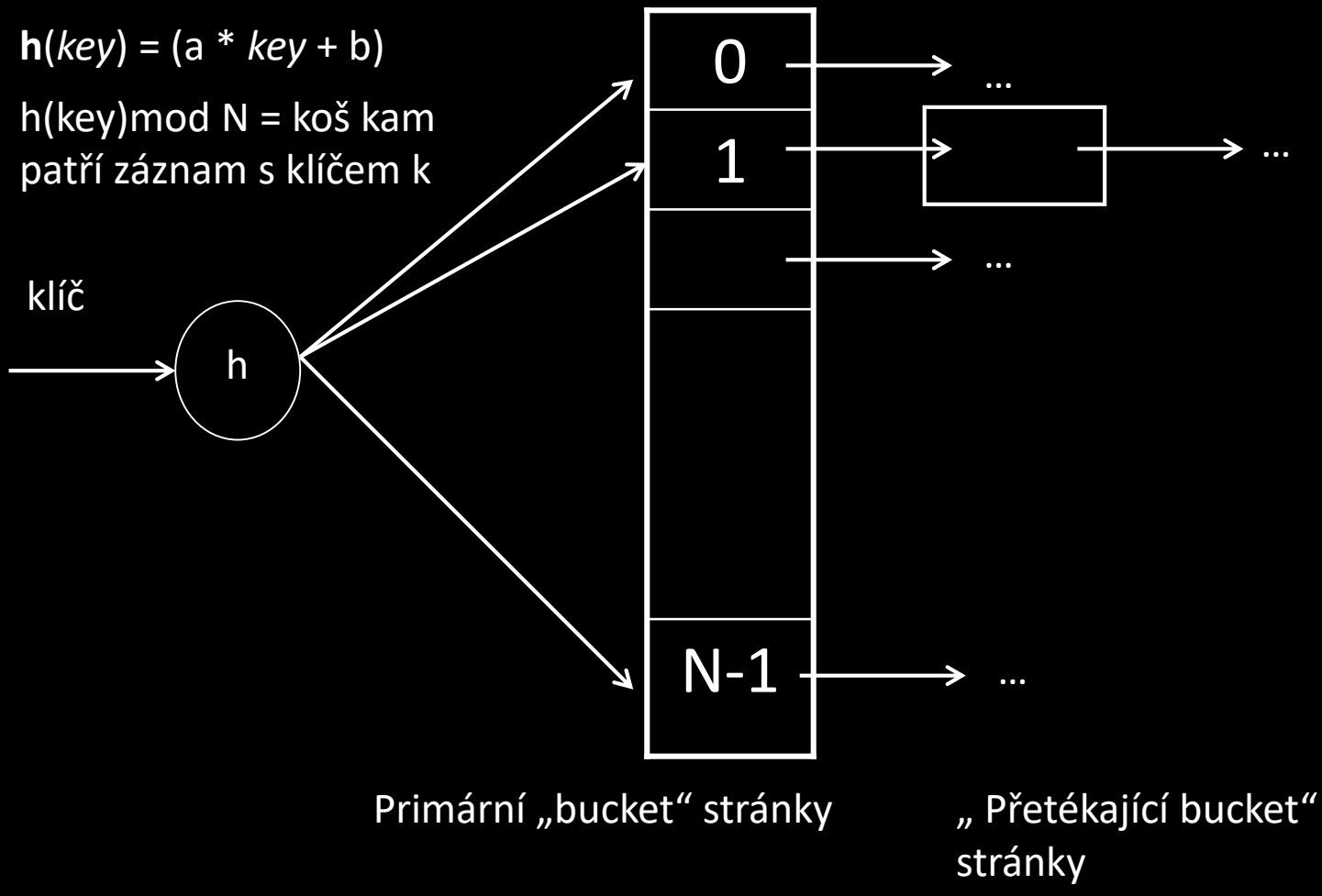
Hash indexování

- Principem je využití hash funkce pro mapování hledané do hodnoty do místa, kde se bude nacházet (tedy například stránky na disku, či skupinu záznamů)

Typy Hash indexů

- Static hashing (stejné problémy jako ISAM při dynamice)
- Extendible hashing (pro dynamické DB)
- Linear Hashing (sofistikovanější metody pro indexování)

Static Hashing



- Máme tedy seznam „buckets“ (budeme říkat koše)
- Jeden primární a případně více další (přetékajících) košů
- Soubor obsahuje N-1 košů s tím, že v každém je na počátku jedna stránka

Koše

- Každý koš může obsahovat záznamy uložené jako:
  - Hromada
  - Třídění soubor
  - Hash soubor

Vyhledávání

- Při hledání záznamu, nejprve aplikuje hash funkci h, která nám identifikuje koš, ve kterém se bude daná položka nalézat

Vkládání

- Použijeme hash h funkci, abychom identifikovali správný koš, do kterého budeme vkládat záznam
- Pokud zde již není místo, musíme přidat tzv. *overflow bucket*

Mazání

- Opět využijme hash funkci h pro identifikaci koše, ze kterého budeme mazat
- Pokud se jednalo o poslední záznam v overflow bucket, pak se tento označí jako prázdný

Hash funkce

- Jedná se o základní prvek ovlivňující efektivitu.
- Musí distribuovat záznamy do košů rovnoměrně

Cena operací

- Vzhledem k tomu, že N (počet košů) je dopředu známý, je možné uložit primární koše na za sebou jsoucí stránky
- Proto počet operací I/O je dán:
  - Jedna pro vyhledání
  - Dvě pro insert, delete (čtení a zápis=1+1)

Overflow koše

- Způsobují problém, neboť je pomocí hash funkce nelze přímo adresovat
- Tedy je nutné po výpočtu hash a identifikaci koše projít všechny ostatní overflow

Problém

- Hlavním problémem je právě zmíněný pevný počet košů, které při vládání mohou začít přetékat
- Případně pokud se bude hodně mazat, pak budeme plýtvat místem

Řešení

1. Čas od času, re-hash soubor
2. Nebo navrhnout hash indexování pro dynamické DB
    1. rozšiřitelné
    2. lineární

Dynamické hash funkce

- Rozšiřitelné
- Lineární

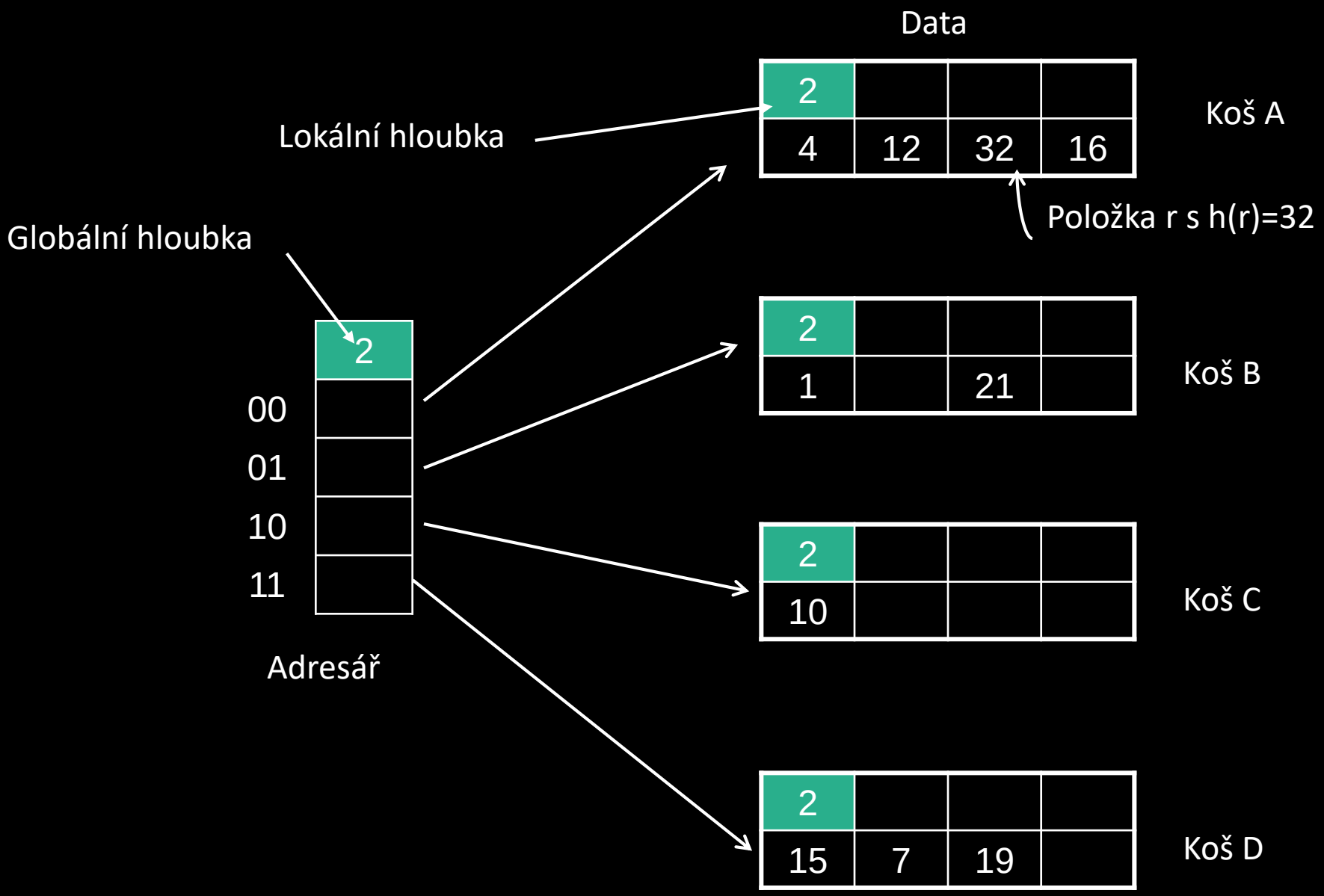
- Jako h(k) označme operaci použití hash funkce pro nalezení koše se záznamem k

Extendible Hashing

- Hledáme řešení pro problém s přetékajícími koši
- U statického se to muselo přeorganizovat, pokud jsme nechtěli mít *overflow buckets*
- K přeorganizování je třeba celý soubor načíst a pak ho znovu zapsat == drahé/časově i paměťově náročné

Řešení

- Použít adresář ukazatelů na koše
- Pro zdvojnásobení počtu košů stačí pracovat k adresářem ukazatelů

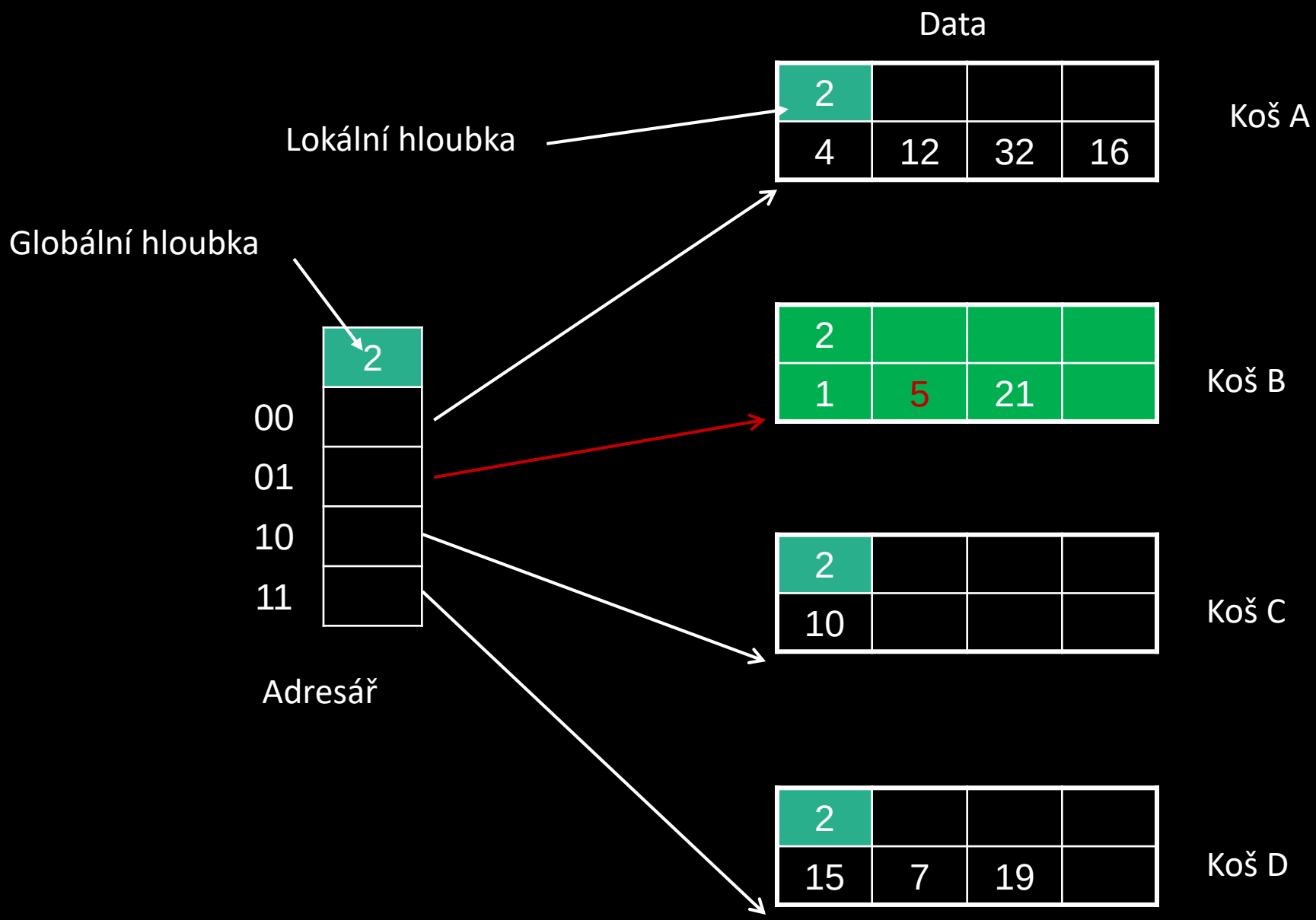

Data
Lokální hloubka
Globální hloubka
2
00
01
10
11
Adresář
2
4
12
32
16
Koš A
Položka r s h(r)=32
2
1
21
Koš B
2
10
Koš C
2
15
7
19
Koš D

Jak to funguje?

- Adresář je pole ukazatelů o velikosti 4, kde každý je ukazatel na koš
- Pro nalezení dat je použita hash funkce (v tuto chvíli bude vracet čísla 00 – 11)
- Tím dostáváme koš

Příklad: hledáme 5

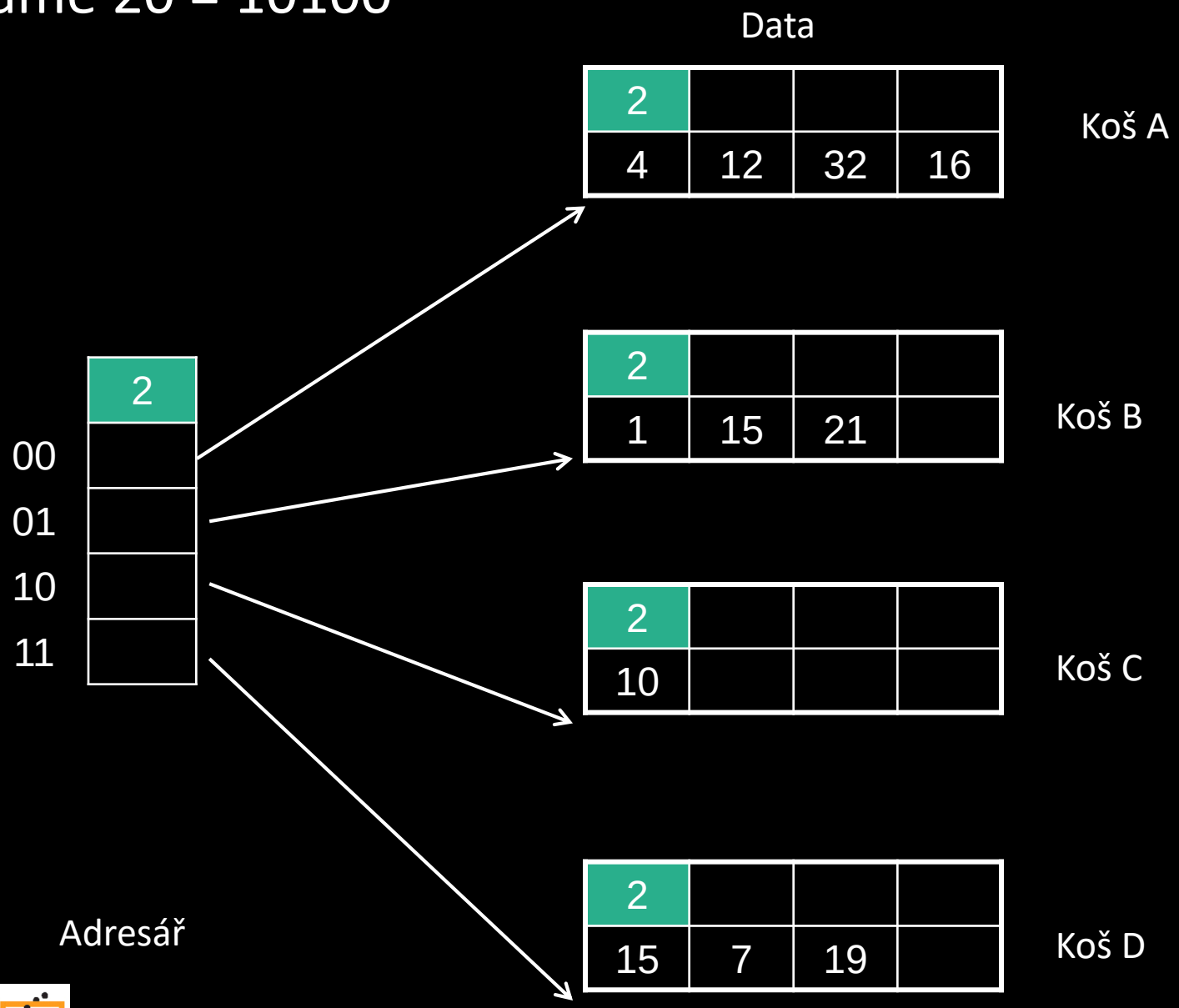
- Tedy hledáme data s hash hodnotou 5 = 101


Data
Lokální hloubka
Globální hloubka
2
00
01
10
11
Adresář
2
4
12
32
16
Koš A
2
1
5
21
Koš B
2
10
Koš C
2
15
7
19
Koš D

Vkládání

- Stejným způsobem najdeme v adresáři ukazatel na koš kam máme vložit
- Pokud je v koši místo přidáme
- Pokud ne, pak to musíme řešit, uvidíme dále …

- Vkládáme 20 = 10100

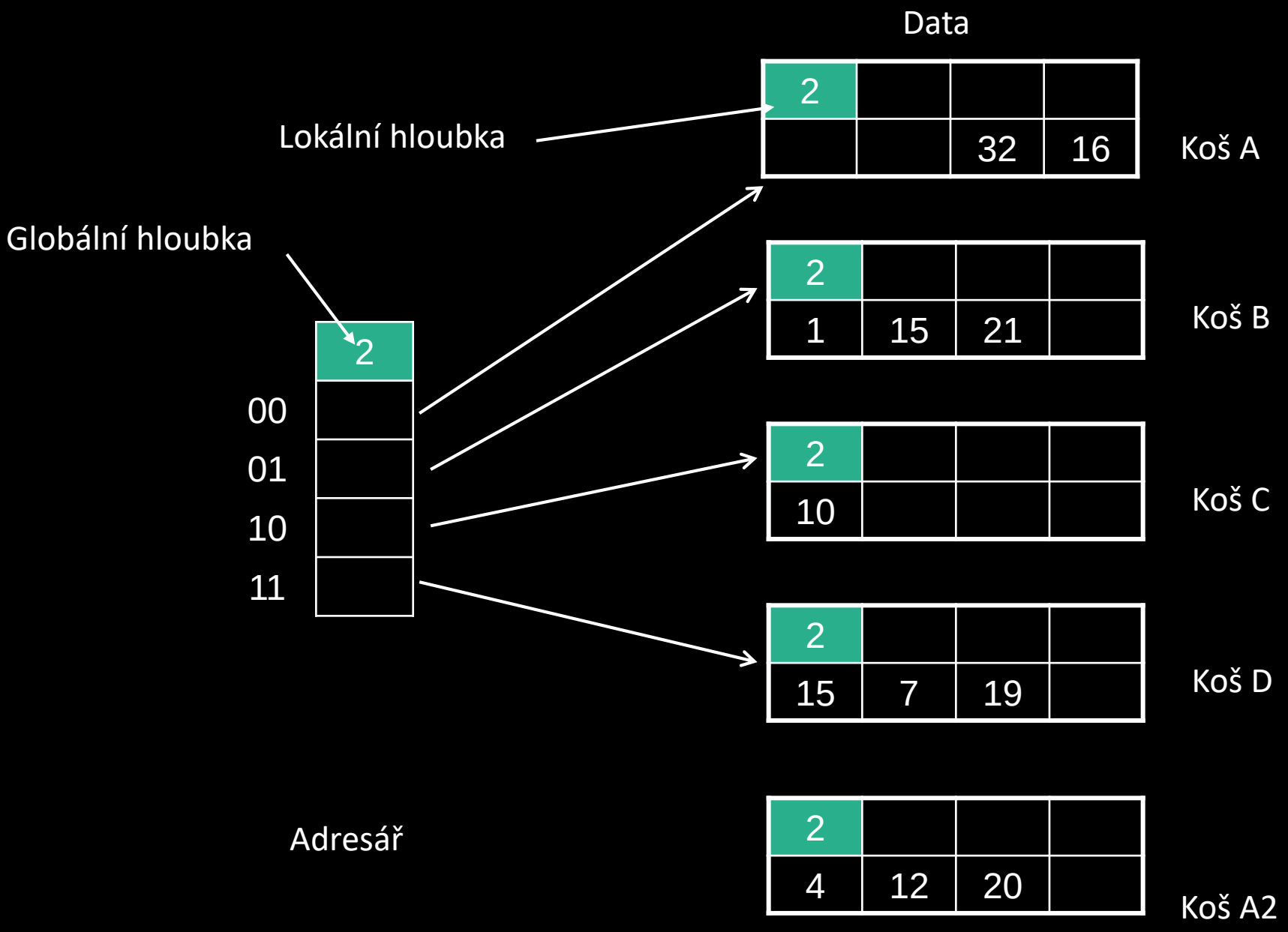


20= 10100

- 00 odpovídá koši A v adresáři
- Ten je ovšem plný

Postup

1. Vytvoříme nový koš
2. Rozdělíme data (stará) i nově vkládaný záznam 20 mezi nový a starý koš


Data
Lokální hloubka
Globální hloubka
2
00
01
10
11
2
32
16
Koš A
2
1
15
21
Koš B
2
10
Koš C
2
15
7
19
Koš D
Adresář
2
4
12
20
Koš A2

Rozdělení dat

- Pro rozdělení použijeme poslední 3 bity hash hodnoty
- První 2 použijeme pro adresaci v adresáři
- A třetí pro rozdělení mezi koši (novým a starým)

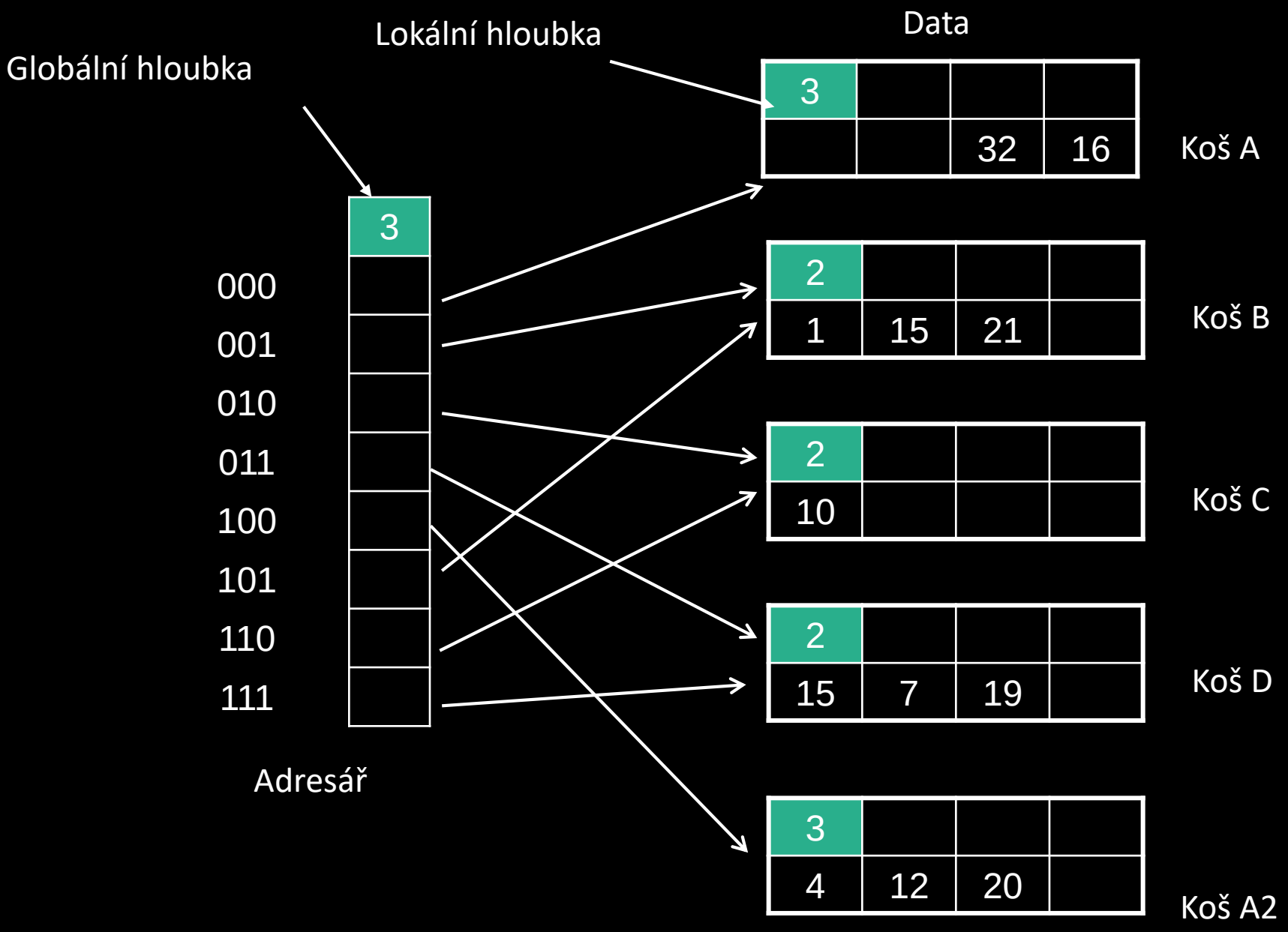
32	100000
16	10000
4	100
12	1100
20	10100

Jak s adresářem

- Nyní máme problém s velikostí adresáře
- Není jak adresovat starý koš
- Jak to budeme řešit?

Řešení

- Klidně zdvojnásobíme adresář (jsou to jen ukazatele)


Globální hloubka
Lokální hloubka
Data
3
000
001
010
011
100
101
110
111
Adresář
3
32
16
Koš A
2
1
15
21
Koš B
2
10
Koš C
2
15
7
19
Koš D
3
4
12
20
Koš A2

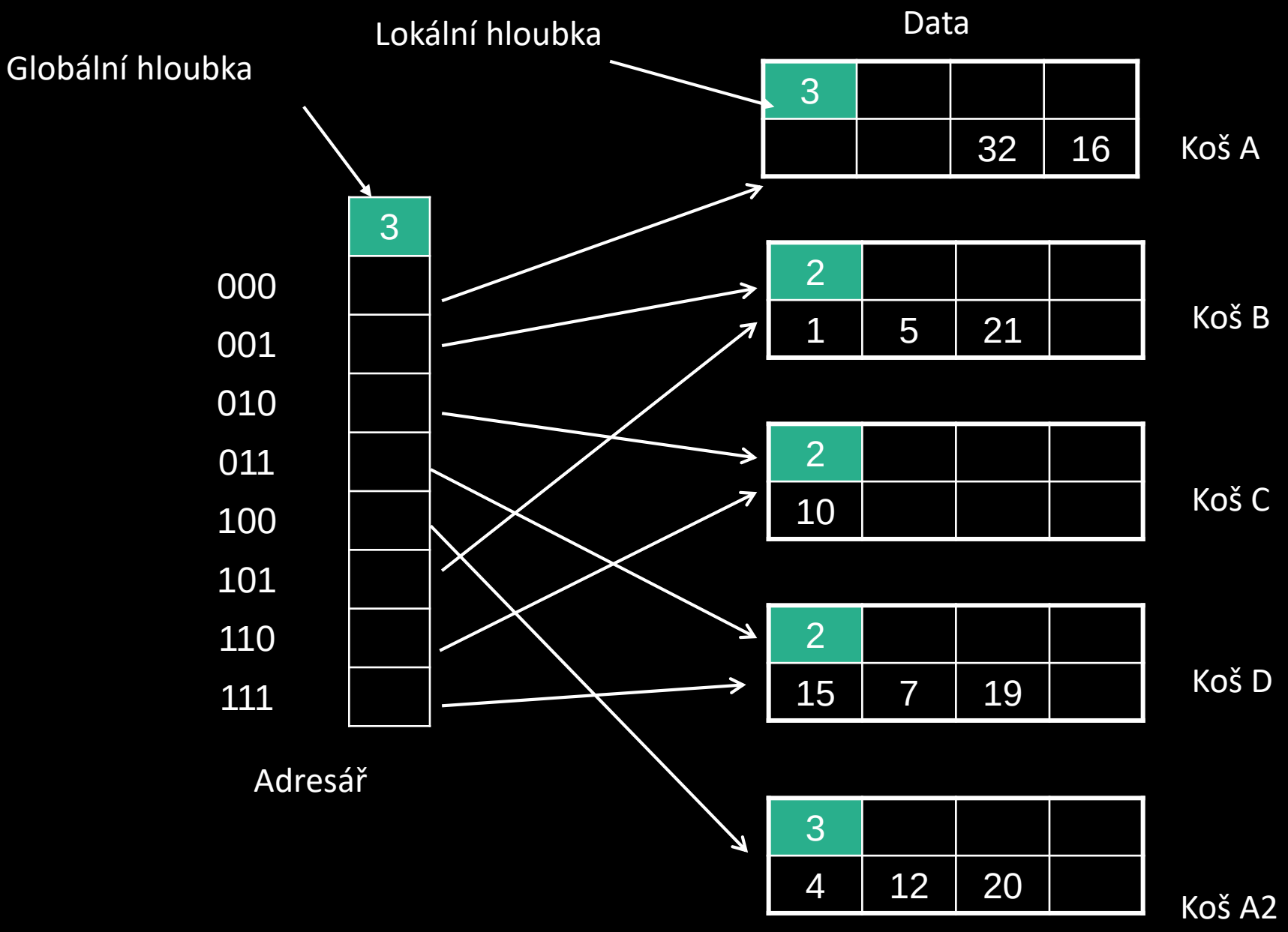
- Všimněme si, že všechny koše, krom nového a starého, jsou odkazovány dvakrát.
- Také si ukažme, že lokální hloubka se změnila. Proč?

Globální hloubka

- Určuje počet bitů, které se použijí k adresování v rámci adresáře (počet bitů na konci hash hodnoty)

Lokální hloubka

- Pokud budeme vkládat tak, že se budou dále vytvářet nové koše, může se stát, že bude třeba opět zdvojnásobit adresář
- Např. pokud chceme vložit 9 (1001)

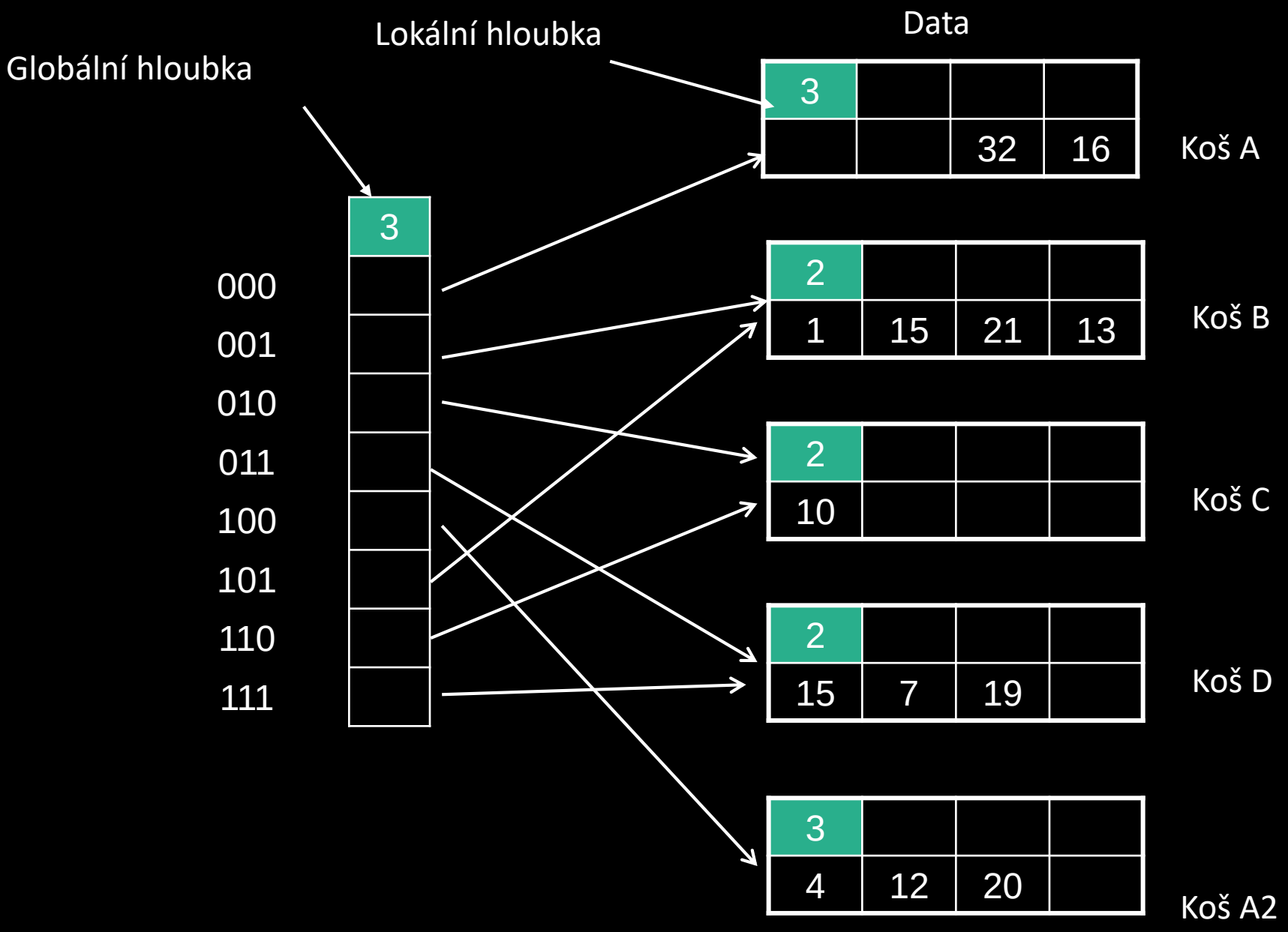

Globální hloubka
Lokální hloubka
Data
3
000
001
010
011
100
101
110
111
Adresář
3
32
16
Koš A
2
1
5
21
Koš B
2
10
Koš C
2
15
7
19
Koš D
3
4
12
20
Koš A2

- Tedy koš B
- Ten není plný, takže žádný problém

- Co pokud budeme vkládat hodnoty tak, že bude třeba rozdělit již dělený koš (A, A2)?

Lokální hloubka

- Proto udržujeme lokální hloubku, abychom věděli jestli je třeba zdvojnásobit adresář, či nikoli
- Pokud lokální hloubka < globální, pak není třeba nic dělat
- Pokud ovšem lokální hloubka == globální, pak je nutné zdvojnásobit adresář
- Příklad?


Globální hloubka
Lokální hloubka
Data
3
000
001
010
011
100
101
110
111
3
32
16
Koš A
2
1
15
21
13
Koš B
2
10
Koš C
2
15
7
19
Koš D
3
4
12
20
Koš A2

Vložení 9

- Patří do koše B, ale ten má lokální hloubku 2 zatím co globální je 3, tím pádem máme v adresáři adresu, kterou lze použít bez dvojnásobení

Lineární hash

- Dynamický hash, navržený pro operace vložení a mazání
- Máme zde více hash funkcí h_0, h_1, h_2, …
- S tím že rozsah funkce je vždy dvakrát větší než byl u jejího předchůdce (tedy h_1 má dvakrát větší rozsah než h_0)
- Pokud h_0 adresovalo do M košů, pak h_1 do 2M košů

Princip

- Pro zjednodušení operací vkládání a mazaní je snadné zdvojnásobit adresář.
- Pro zdvojnásobení se používají funkce h_{LEVEL}, $h_{LEVEL+1}$
- LEVEL znamená krok, ve kterém provádíme zdvojnásobení (porovnej s globální hloubkou)

Hledání

- Pro hledání se použije funkce h_{LEVEL}, pokud ukazuje na koš, který již obsahuje data (tedy nebyl dělen) pak jsme našli, jinak použij funkci $h_{LEVEL+1}$, atd. dokud není nalezen záznam

Indexy s bitovou mapou

- Oracle
- Vhodné pro:
  - Pro atributy s malou kardinalitou (poměr mezi celkovým počtem záznamů/řádků a počtem různým hodnot v těchto řádcích)
  - Jako ideální poměr se ukazuje být 1% (doporučuje Oracle)
  - Pro statické tabulky (málo nebo žádné insert/update dotazy), např. Datové sklady

- Jako příklad je možné uvést atribut pohlaví, kde je pouze muž nebo žena
- Jde je však využít i pro atributy s větším množstvím různých hodnot
- Např. tabulka s milionem záznamů a atribut s 10000 různých hodnot je vhodným kandidátem mil/10000=1%
- V ostatních případech je vhodnější použít jiný typ indexu

- Plné indexování velké tabulky v DB s tradičními indexy (B+ stromy) může být náročné na uložení (může nastat situace, kdy indexy mohou být několikráte větší než samotná data)
- Bitmap indexy jsou typicky pouze zlomky z velikosti celé tabulky

Proč malá kardinalita a proč vhodné pro WHERE?

- Pro atributy s malou kardinalitou (poměr mezi celkovým počtem záznamů/řádků a počtem různým hodnot v těchto řádcích)
- Bitmapové indexy se jeví být jako nejvhodnější pro dotazy s více podmínkami ve WHERE klausuli

Protože …

- Jsou uloženy jako bity, každá různá hodnota (záznam) v tabulce jeden bit

Příklad 1.

id	gender	status	cust_income_level
70	F		D: 70,000 - 89,999
80	F	married	C: 50,000 - 69,999
90	M	single	I: 170,000 - 189,999
100	F		C: 50,000 - 69,999
110	F	married	I: 170,000 - 189,999
120	M		J: 190,000 - 249,999
130	M	married	C: 50,000 - 69,999

- Protože atribut gender obsahuje málo různých hodnot, je vhodný pro bitmapové indexy

0=ne, 1=ano

	gender	gender='M'	gender=,F'
70	F	0	1
80	F	0	1
90	M	1	0
100	F	0	1
110	F	0	1
120	M	1	0
130	M	1	0

Dotaz

- "WHERE gender='Female' OR gender='Unknown';
- Což je zřejmě velmi snadná operace
- Co pokud by tam bylo AND?

Female	Unknown	
1	0	Match
1	0	Match
0	0	
0	0	
1	0	Match
0	0	
0	0	
1	0	Match
1	0	Match
0	0	

Příklad

- create bitmap index person_region on person (region);

Row	Region	Nort	East	West	South
1	North	1	0	0	0
2	East	0	1	0	0
3	West	0	0	1	0
4	West	0	0	1	0
5	South	0	0	0	1
6	North	1	0	0	0

- WHERE Region=„NORT“ OR „EAST“ OR „WEST“

Row	Region	Nort	East	West	South	
1	North	1	0	0	0	match
2	East	0	1	0	0	match
3	West	0	0	1	0	match
4	West	0	0	1	0	match
5	South	0	0	0	1	
6	North	1	0	0	0	match

Datové struktury pro indexy

- ISAM
- B++ stromy
- Hash indexy
- Indexy s bitovou mapou