Kapitola 11: Indexování a hešování

- Základní představa
- Řazené indexy (ordered indices)
- B+-strom indexový soubor
- B-strom indexový soubor
- Hešování
- Porovnání řazených indexů a hešování
- Definice indexů v SQL

Základní představa

- Indexové mechanizmy se používají pro zrychlení přístupu k požadovaným datům
  - Např. katalog autorů v knihovně
- **Vyhledávací klíč** (search key) – atribut nebo množina atributů používaný pro vyhledávání záznamů v souboru
- **Indexový soubor** se skládá ze záznamů (index entries) ve tvaru

Vyhledávací klíč	ukazatel

- Indexový soubor je typicky mnohem menší než původní soubor
- Dva základní typy indexů:
  - **Řazené indexy** – vyhledávací klíče jsou uspořádané
  - **Hešovací indexy** – vyhledávací klíče jsou rovnoměrně rozprostřeny po adresovacím prostoru hešovací funkce

Metriky pro porovnávání indexů

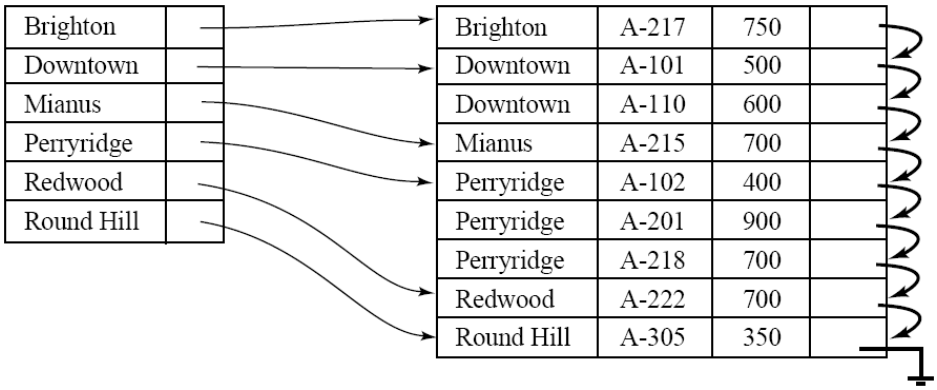
- Typy efektivně (rychle) podporovaných přístupů:
  - Dotazy na přesnou shodu ve vyhledávacím klíči
  - Dotazy na rozsah vyhledávacích klíčů
- Přístupová doba
- Doba pro vložení záznamu
- Doba pro smazání záznamu
- Prostorová režie (kolik místa je potřeba pro index)

Řazené indexy

- Indexové záznamy jsou uloženy uspořádané podle vyhledávacího klíče, např. katalog autorů v knihovně
- **Primární index** – seřazený sekvenční soubor; index, jehož vyhledávací klíč určuje pořadí záznamů v souboru
  - často nazývaný jako **shlukující soubor**
  - vyhledávací klíč primárního indexu nemusí být ve všech případech primárním klíčem
- **Sekundární index** – index, jehož vyhledávací klíč určuje jiné pořadí než v seřazeném sekvenčním souboru; nazývaný **neshlukující index**
- **Index-sekvenční soubor** – seřazený sekvenční soubor s primárním indexem

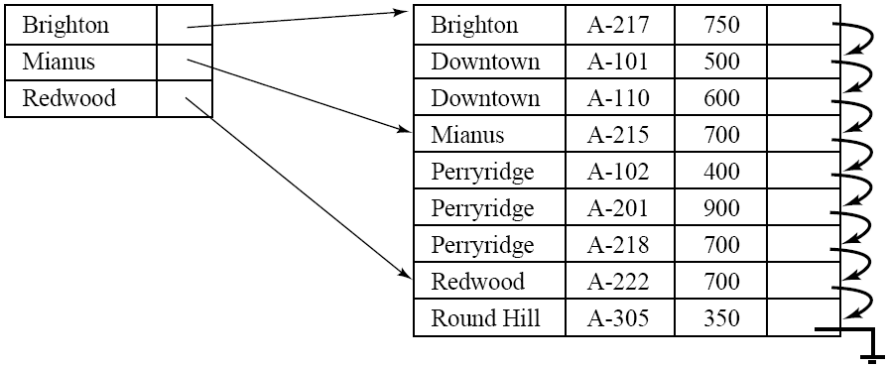
Hustý indexový soubor

- Hustý index – indexový záznam je uložený pouze pro každou hodnotu vyhledávacího klíče (ale stejné hodnoty klíče se v indexu neopakují)



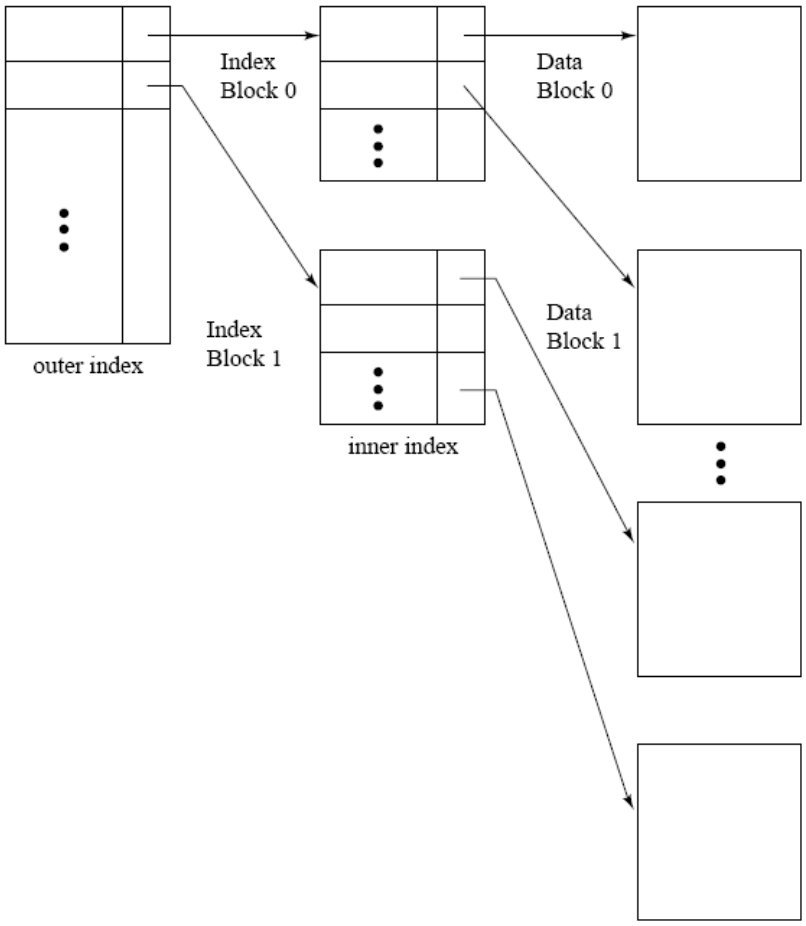
Řídký indexový soubor

- Indexové záznamy jsou pouze pro některé hodnoty vyhledávacího klíče
- Pro nalezení záznamu s vyhledávacím klíčem *K* musíme:
  - Nalézt indexový záznam s největším vyhledávacím klíčem menším než *K*
  - Prohledat sekvenční soubor od tohoto záznamu
- Méně prostoru pro uložení indexu a méně udržujících operací při vkládání a mazání záznamu
- Obecně je ale při vyhledávání pomalejší než hustý index
- Vhodné řešení je řídký indexový soubor pro každý blok v souboru



Víceúrovňový index

- Pokud se primární index nevejde do operační paměti mohou být přístupy pomalé (tím pádem drahé)
- Pro snížení počtu diskových přístupů k indexovému souboru se primární index považuje za sekvenční soubor na disku a vytvoří se pro něj řídký index.
  - Vnější (outer) index je řídký index na primárním indexu
  - Vnitřní (inner) index je primární index na souboru
- Pokud se i vnější index nevejde do paměti, můžeme vytvořit další úroveň
- Indexy na všech úrovních musí být aktualizovány při vkládání nebo mazání



Aktualizace indexu – mazání

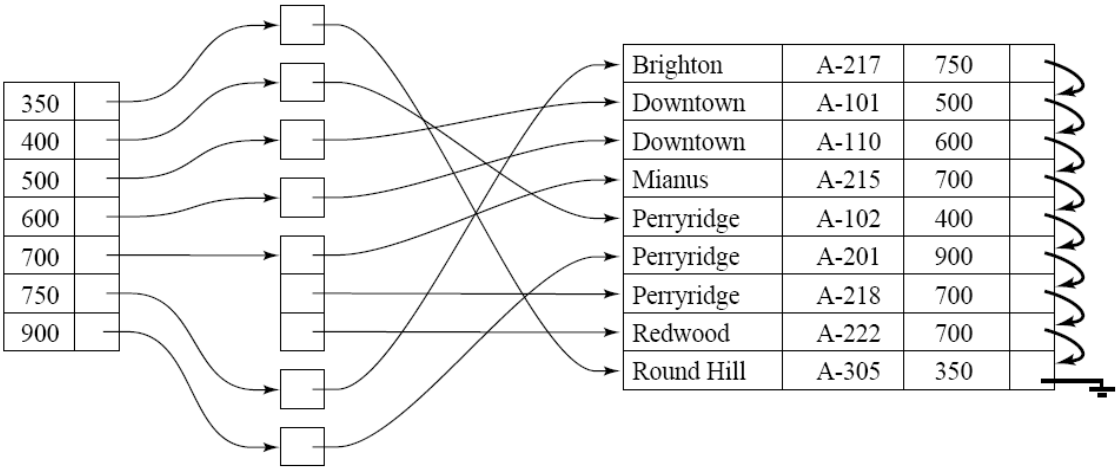
- Pokud je mazaný záznam jediným záznamem s daným vyhledávacím klíčem, vyhledávací klíč je také vymazán z indexu
- Jednoúrovňové mazání:
  - Hustý index – mazání vyhledávacího klíče je podobné jako mazání záznamu
  - Řídký index – pokud existuje indexový záznam pro hodnotu vyhledávacího klíče, je tento záznam nahrazen vyhledávacím klíčem, který bezprostředně následuje mazaný klíč v pořadí řazení podle vyhledávacího klíče. Pokud takový klíč již v indexu existuje, je hodnota pro mazaný záznam v indexu smazána.

Aktualizace indexu – vkládání

- Jednoúrovňové mazání:
  - Proveď vyhledávání s hodnotou vyhledávacího klíče vkládaného záznamu
  - Hustý index – pokud není klíč v indexu nalezen, je tam vložen
  - Řídký index – pokud je v indexu uložen klíč pro každý blok souboru, žádné změny není nutno provádět tehdy, když není vytvořen nový blok. Pokud je nový blok vytvořen je vložen klíč, který ho bude zastupovat.

Sekundární index

- Často jsou kladeny vyhledávací dotazy, které hledají všechny záznamy, které na určitém atributu (který není součástí primárního indexu) nabývají požadované hodnoty.
  - Např.: v relaci *účet* jsou záznamy uloženy v sekvenčním souboru a upořádány podle čísla účtu, ale my vyhledáváme všechny účty vedené danou pobočkou.
  - Např.: stejný dotaz, ale navíc chceme vypsat pouze účty, které mají určitý zůstatek nebo mající zůstatek v nějakém intervalu.
- Můžeme definovat sekundární index se záznamy pro každou hodnotu vyhledávacího klíče; takový indexový záznam určuje blok, který obsahuje ukazatele na všechny záznamy mají danou hodnotu vyhledávacího klíče.
- Sekundární index na relaci *účet* s atributem *zůstatek*



Primární a sekundární indexy

- Sekundární indexy musí být husté.
- Indexy obecně nabízejí podstatné výhody při vyhledávání záznamů.
- Pokud je soubor změněn, každý index na tomto souboru musí být změněn také. Aktualizace indexů ale přináší další režii.
- Sekvenční prohledávání s použitím primárního indexu je výkonné, ale sekvenční hledání s použitím sekundárního indexu je drahé (přístup ke každému záznamu může vést ke čtení nového bloku, protože pořadí záznamů v sekvenčním souboru se může významně lišit od pořadí podle sekundárního indexu).

B^+-stromy

B^+-stromy jsou jinou možností k index-sekvenčním souborům

- Nevýhody index-sekvenčních souborů: výkonnost klesá s rostoucím souborem, protože mnoho bloků přeteče a je nutné provádět opakované reorganizace.

- Výhoda indexů s B^+-stromy: při vkládání/mazání se provádí automatická reorganizace pouze s malými, lokálními změnami. Reorganizace celého souboru nejsou nutné pro opětovné zvýšení výkonnosti.
- Nevýhoda B^+-stromů: režie při vkládání/mazání záznamů, zvýšené prostorové nároky.
- Ovšem výhody B^+-stromů převažují nevýhody a jsou nejčastěji používanou indexovou organizací v databázích.

B^+-strom je strom s jedním kořenem splňující následující podmínky:

- Všechny cesty od kořene k listům mají stejnou délku
- Každý uzel, který není kořenem nebo listem, má potomků $\lceil n/2 \rceil$ až n
- Listový uzel má $\lceil (n-1)/2 \rceil$ až $n-1$ hodnot (klíčů)
- Zvláštní případy:
  - Pokud kořen stromu není zároveň list, potom má nejméně 2 potomky.
  - Pokud je kořen současně listem (strom má 1 uzel), počet uložených hodnot je mezi 0 a $(n-1)$
- Typický uzel

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

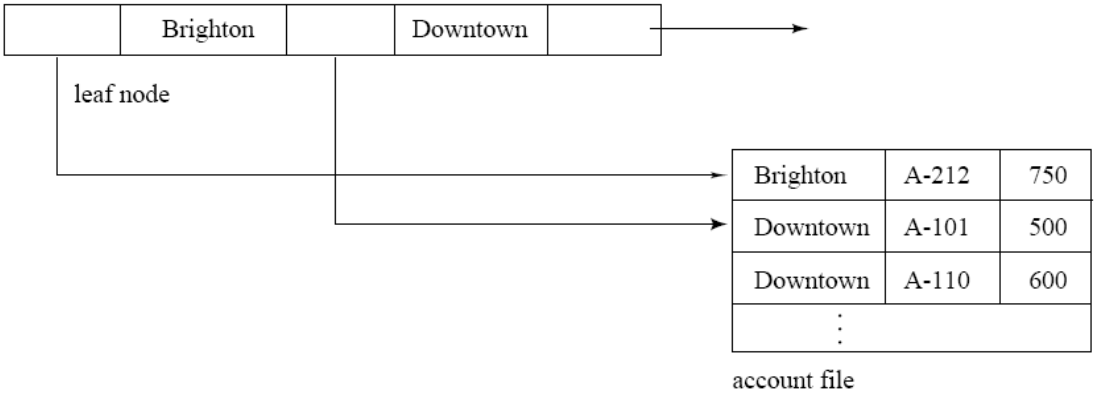
  - K_i jsou hodnoty vyhledávacího klíče
  - P_i jsou ukazatele na potomky (platí pro vnitřní uzly) nebo ukazatele na záznamy nebo bloky záznamů (pro listové uzly)
- Vyhledávací klíče jsou uspořádány

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

Listy v B^+-stromech

Vlastnosti každého listu:

- Pro každé $i=1,2,\ldots,n-1$ ukazatel P_i odkazuje buď na záznam s klíčem K_i nebo na blok ukazatelů na záznamy, kde každý záznam má klíč K_i. Pokud index není primární, potom jsou uloženy odkazy na bloky.
- Pokud L_i, L_j jsou listy a $i < j$, potom klíče v L_i jsou menší než klíče v L_j.
- P_n ukazuje na další list



Vnitřní uzly v B^+-stromech

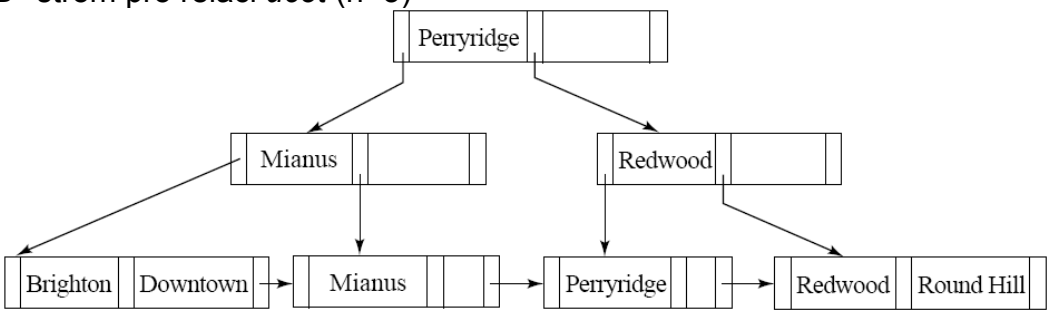
- Nelistové uzly vytvářejí víceúrovňový řídký index na listových uzlech. Pro každý nelistový uzel s m ukazateli platí:

- Všechny vyhledávací klíče v podstromu, na který P_1 ukazuje, jsou menší než K_1
- Pro $2 \le i \le n-1$, všechny vyhledávací klíče v podstromu, na který ukazuje P_i, mají hodnoty větší nebo rovny než K_{i-1} a menší než K_i
- Všechny vyhledávací klíče v podstromu, na který ukazuje P_m, jsou větší nebo rovny než K_{m-1}

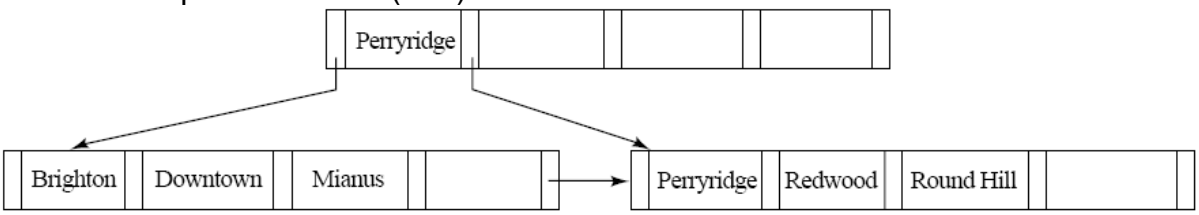
P_1	K_1	P_2	…	P_{m-1}	K_{m-1}	P_m

Příklad B$^+$-stromu

- B$^+$-strom pro relaci *účet* (n=3)



- B$^+$-strom pro relaci *účet* (n=5)



- Listy musí mít 2 až 4 hodnoty (pro n=5)
- Nelistový uzel kromě kořene musí mít 3 až 5 potomků (pro n=5)
- Kořen musí mít alespoň 2 potomky

B$^+$-strom - pozorování

- Protože spojení vnitřních uzlů je pomocí ukazatelů, nemůžeme předpokládat, že bloky „logicky“ blízké v B$^+$-stromu jsou také blízké „fyzicky“
- Nelistové úrovně stromu tvoří hierarchii řídkých indexů
- B$^+$-strom obsahuje relativně malý počet úrovní (logaritmický k velikosti souboru), tedy vyhledávání jsou prováděny efektivně.
- Vkládání a mazání v souboru lze řešit také efektivně, protože index je aktualizovaný v logaritmickém čase.

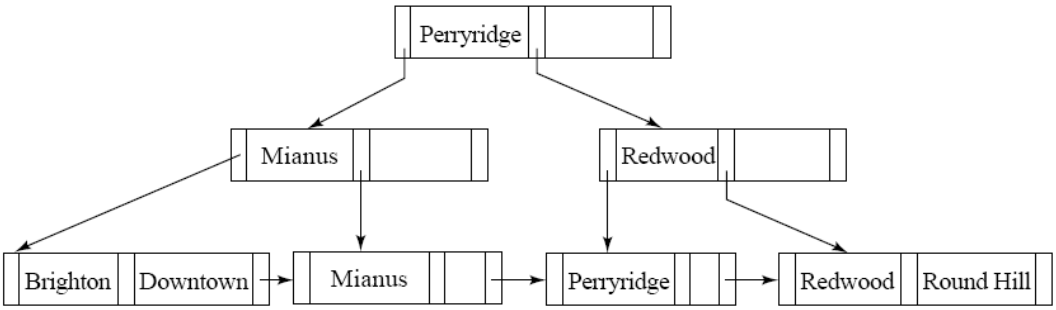
B$^+$-strom - dotazy

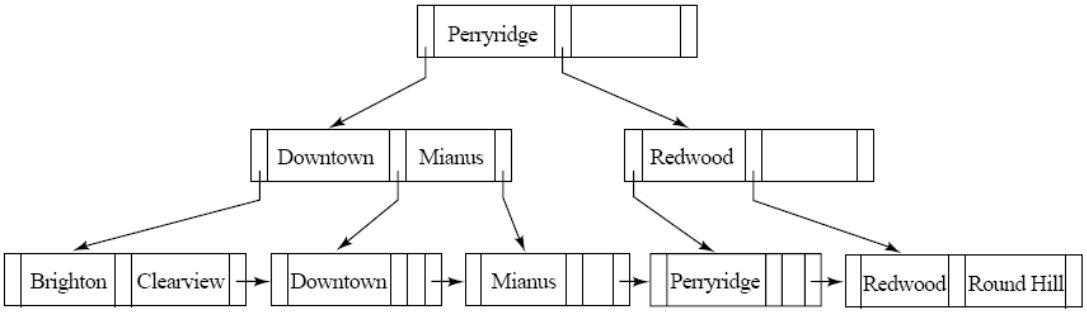
- Najdi všechny záznamy s vyhledávacím klíčem *k*
  - Začni v kořenu stromu
    * V uzlu najdi klíč K_i, který je nejmenší větší než *k*
    * Pokud takové K_i existuje, jdi do potomka P_i
    * Jinak $k \ge K_{m-1}$, jdi do potomka P_m
  - Pokud není aktuální uzel list, pokračuj předchozím bodem

  - Pokud jsme v listu, proveď následující: když $K_i=k$, následuj ukazatel P_i a přečti uložený záznam nebo blok; pokud není žádné K_i rovné k, žádný takový záznam neexistuje.
- Během zpracování dotazu je strom procházen od kořene k nějakému listu
- Pokud je v souboru K vyhledávacích klíčů, potom cesta není delší než $\lceil log_{\lceil n/2 \rceil}(K) \rceil$
- Uzel má obvykle stejnou velikost jako blok na disku, tj. 4 KB, a n bývá kolem 100 (což znamená 40 B na jednu položku indexu)
- Při 1 milionu vyhledávacích klíčů a $n=100$, nejvýše $log_{50}(1000000)=4$ uzlů je přistoupeno během vyhledávání
- Pro porovnání s vyváženým binárním stromem máme asi 20 uzlů přistoupených během hledání ($log_2(1000000) \sim 20$)
  - Tento rozdíl je významný, protože každý přistoupený uzel vyžaduje čtení jednoho bloku z disku, což odpovídá asi 30 milisekundám!

Aktualizace B^+-stromu - vkládání

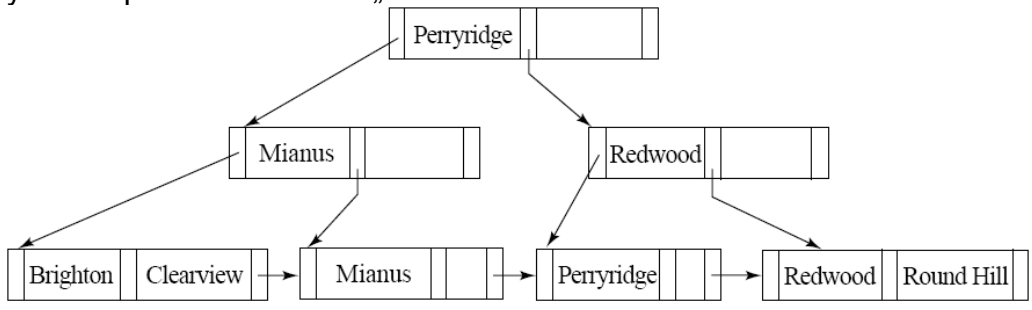
- Najdi list, do kterého má být nový klíč vložen
- Pokud klíč již existuje, nový záznam je přidán do souboru a je vytvořen ukazatel na blok souboru, pokud je třeba.
- Pokud není nový klíč v indexu, opět ulož nový záznam do souboru a potom:
  - Pokud je v listu dostatek místa, vlož nový klíč a ukazatel na záznam na správnou pozici
  - Pokud v listu není volné místo, rozděl uzel a ulož nový klíč a ukazatel na záznam
- Rozdělení uzlu (node split)
  - Vezmi všech n dvojic (klíč, ukazatel na záznam) včetně nového klíče a setřiď je. Prvních $\lceil n/2 \rceil$ dvojic ulož do původního uzlu a zbývající ulož do nového uzlu.
  - Předpokládejme, že nový uzel je p a nechť k je nejmenší klíč v p. Vlož dvojici (k,p) do rodičovského uzlu právě rozděleného uzlu. Pokud je rodičovský uzel přeplněný, rozděl ho a propaguj rozdělení o úroveň výše.
- Rozdělování uzlů postupuje postupně do vyšší a vyšší úrovně stromu, dokud není nalezený nepřeplněný uzel. V nejhorším případě je rozdělený kořen stromu a je vytvořen nový, výsledkem celý strom vyroste o jednu úroveň.



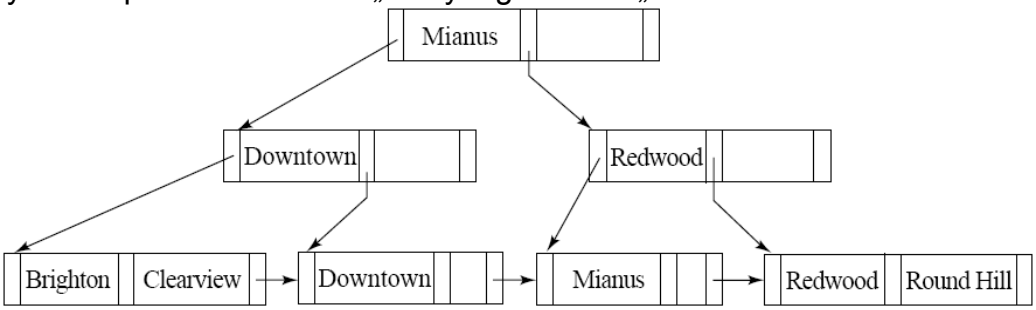


Aktualizace B$^+$-stromu - mazání

- Najdi záznam, který má být smazán, a vymaž ho ze souboru.
- Zruš dvojici (klíč, ukazatel na záznam) z listu, pokud ukazatel není na blok nebo je blok již prázdný.
- Pokud má uzel příliš málo položek kvůli mazání záznamů a pokud se zbývající položky vejdou do sousedního uzlu, potom
  - Vlož všechny klíče z obou uzlů do jednoho (do uzlu vlevo) a smaž prázdný uzel.
  - Z rodičovského uzlu smaž dvojici (K_{i-1},P_i), pokud P_i je ukazatel na právě smazaný uzel. Když je třeba aplikuj rekurzivně na rodiče.
- Pokud má uzel příliš málo položek kvůli mazání záznamů a pokud se zbývající položky nevejdou do sousedního uzlu, potom
  - Rozděl položky z obou uzlů tak, že oba uzly mají alespoň minimální počet položek.
  - Aktualizuj odpovídající položky v rodičovském uzlu
- Mazání uzlu se může kaskádovitě přesunovat do vyšších úrovní, dokud není nalezen uzel alespoň $\lceil n/2 \rceil$dvojicemi. Pokud po mazaní má kořen pouze jednu položku, je zrušen a jeho jediný potomek se stane novým kořenem.
- Výsledek po smazání klíče „Downtown“ z indexu na relaci *účet*



- Smazání klíče „Downtown“ vede ke smazání listového uzlu, které ale nevede k nedostatečnému počtu klíčů v rodičovském uzlu. Tedy kaskádové mazání je ukončeno.
- Výsledek po smazání klíče „Perryridge“ místo „Downtown“



- Po smazání uzlu s klíčem „Perryridge“ dojde ke snížení počtu klíčů v rodičovském uzlu pod minimální hodnotu, ale sousední uzel rodiče (sibling) je již plný a nemůže přijmout ukazatel na další list, proto dojde k rozdělení ukazatelů na listy mezi dva uzly. Všimněme si, že se změnil klíč v kořeni stromu.

Souborová organizace B$^+$-strom

- Problém s degradací indexového souboru (a následné reorganizace) jsou vyřešeny s použitím B$^+$-stromu. Degradace souboru je řešena pomocí souborové organizace B$^+$-strom.
- Listové uzly B$^+$-stromu obsahují přímo celé záznamy místo ukazatelů na ně.
- Protože záznamy jsou větší než samotné ukazatele, sníží se kapacita listu.
- Podmínka na alespoň poloviční naplnění listu je stále platná.
- Vkládání a mazání probíhá totožně jako v indexu B$^+$-strom.
- Dobré využití místa je zde důležitější kvůli větší prostorové náročnosti záznamů. Pro zvýšení využívání volného místa se používá metoda přeskupování záznamů mezi více sousedními uzly než při použití indexu B$^+$-strom, to se děje při rozdělování a slučování uzlů.

Index B-strom

- Podobný s B$^+$-stromy, ale B-stromy neopakují klíče ve vnitřních uzlech, každý klíč se ve stromu vyskytuje nejvýše jednou. To snižuje prostorové nároky indexu.
- Vyhledávací klíče použité ve vnitřních uzle pro navigaci ve stromu se již v listech neopakují, proto je struktura vnitřního uzlu rozšířena o ukazatel na záznam.
- List v B-stromu:

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

- Vnitřní uzel v B-stromu: (ukazatelé B_i ukazují buď na záznamy nebo na bloky)

P_1	B_1	K_1	P_2	B_2	K_2	…	P_{m-1}	B_{m-1}	K_{m-1}	P_m

- Výhody indexu B-strom:
  - Má méně uzlů než odpovídající B$^+$-strom
  - Někdy je možné nalézt klíč dříve něž v listu
- Nevýhody B-stromů:
  - Pouze velmi malý počet klíčů je nalezený dříve než v listu

  - Nelistové uzly jsou větší, tedy počet potomků je nižší (protože uzel je uložený v jednom bloku). Důsledkem je vyšší hloubka stromu než v případě B^+-stromů.
  - Vkládání a mazání je složitější než u B^+-stromů
  - Implementace je také obtížnější
- Typicky výhody použití B-stromu nepřevažují nad jeho nevýhodami.

Statické hešování

- **Kyblík** (bucket) je základní úložnou jednotkou obsahující jeden nebo více záznamů (kyblík je obvykle tvořen jedním diskovým blokem). V **hešovací souborové organizaci** získáme kyblík, kde je hledaný záznam uložen, přímo z jeho vyhledávacího klíče pomocí **hešovací funkce**.
- Hešovací funkce *h* je funkce převádějící množinu vyhledávacích klíčů *K* na množinu adres jednotlivých kyblíků *B*.
- Hešovací funkce se používá jak při vyhledávání záznamu, tak při vkládání a mazání.
- Záznamy s rozdílnými klíči mohou být uloženy ve stejném kyblíku, tedy celý obsah kyblíku musí být sekvenčně prohledán.

Hešovací funkce

- Nejhorší hešovací funkce mapuje všechny vyhledávací klíče do jednoho kyblíku, což vede k vyhledávacímu času závislému na počtu klíčů v souboru.
- Ideální hešovací funkce je *rovnoměrná*, tj. každý kyblík obsahuje stejné množství klíčů z množiny všech možných hodnot vyhledávacího klíče.
- Ideální hešovací funkce je *náhodná* a každý kyblík má přeřazen stejný počet záznamů bez ohledu na aktuální rozložení vyhledávacích klíčů v souboru.
- Typicky hešovací funkce pracují s interní binární podobou vyhledávacího klíče. Např. pro řetězcový klíč binární zápis všech znaků v řetězci může být sečten a adresa kyblíku je výsledkem operace modulo celkový počet kyblíků.

Příklad hešovací souborové organizace

bucket 0

bucket 1

bucket 2

bucket 3

Brighton	A-217	750
Round Hill	A-305	350

bucket 4

Redwood	A-222	700

bucket 5

Perryridge	A-102	400
Perryridge	A-201	900
Perryridge	A-218	700

bucket 6

bucket 7

Mianus	A-215	700

bucket 8

Downtown	A-101	500
Downtown	A-110	600

bucket 9

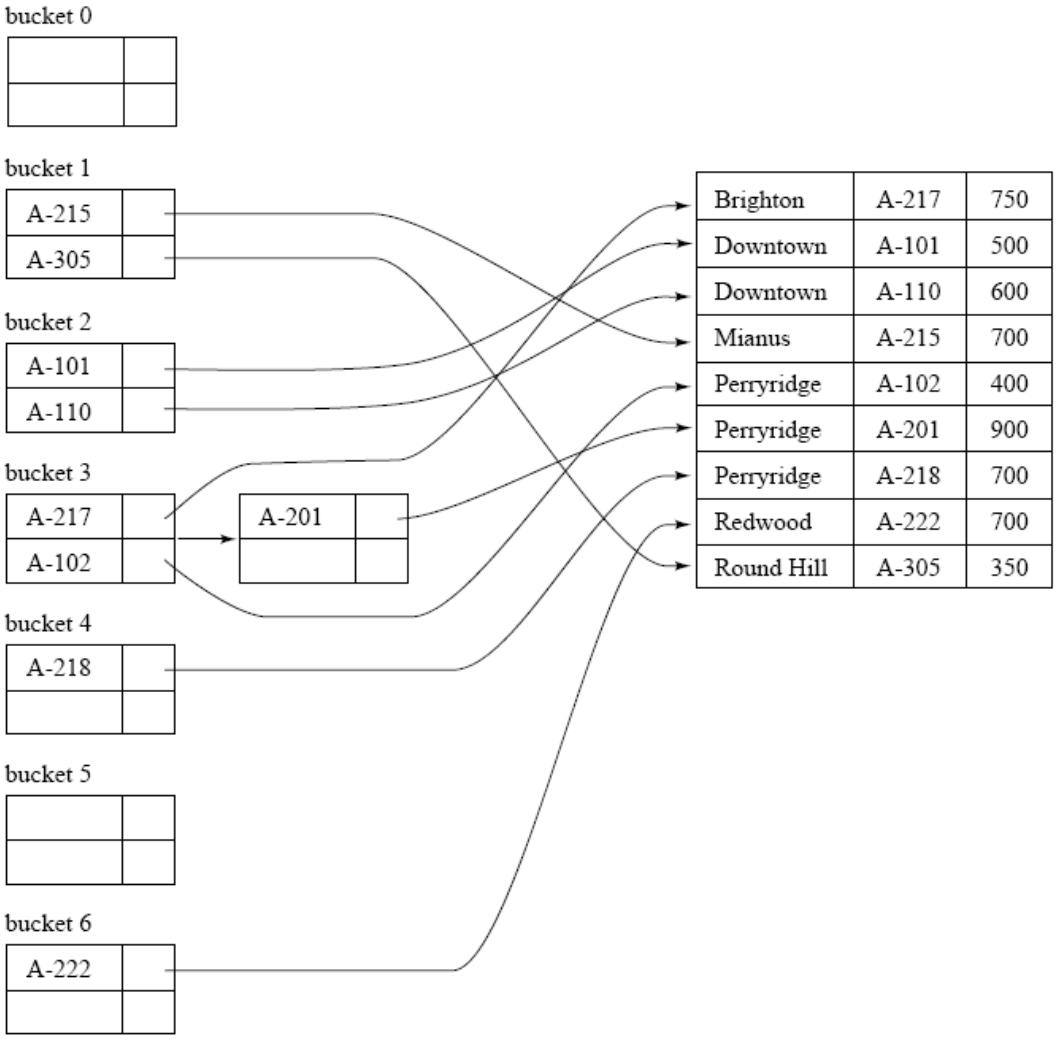
- 10 kyblíků
- binární zápis *i-tého* znaku je chápán jako integer *i* (celé číslo)
- hešovací funkce vrací součet všech integerů modulo 10

Řešení přetečení kyblíku

- K přetečení kyblíku dochází
  - Nedostatečný počet kyblíků
  - Asymetrické rozložení klíčů, které může mít dva důvody:
    * Více záznamů má stejný vyhledávací klíč
    * Zvolená hešovací funkce není rovnoměrná
- Ačkoli pravděpodobnost přetečení kyblíku může být snížena, nelze se jí zbavit úplně a je řešena pomocí **přetokových kyblíků.**
- **Řetězení přetoků** – přetokové kyblíky některého kyblíku jsou řetězeny do jednoho seznamu
- Výše popsané schéma je nazýváno jako **uzavřené hešováni.** Jinou možností je **otevřené hešování**, které není vhodné pro databázové aplikace.

Hešovací indexy

- Hešování může být použito nejenom jako souborová organizace, ale i jako indexová struktura. **Hešovací index** organizuje vyhledávací klíče spolu s ukazateli na záznamy v hešovací souborové organizaci.
- Hešovací indexy jsou vždy používány jako sekundární indexy – pokud i soubor používá hešování, není potřeba vytvářet zvláštní primární index nad stejným vyhledávacím klíčem. Často se používá výraz hešovací index (nebo jen hešování) pro obě techniky sekundární hešovací index i hešovací soubor.



Nedostatky statického hešování

- Ve statickém hešování mapuje hešovací funkce vyhledávací klíče na pevnou množinu adres kyblíků.
  - Ovšem databáze se v čase zvětšují. Pokud je počáteční velikost adresového prostoru příliš malá, výkonnost této techniky je snížena kvůli příliš častému přetečení.
  - Pokud je zvolena nějaká předpokládaná velikost souboru a je vytvořen odpovídající počet kyblíků, dochází ze začátku k významnému plýtvání místem.
  - Je-li velikost databáze snížena, opět se plýtvá místem.

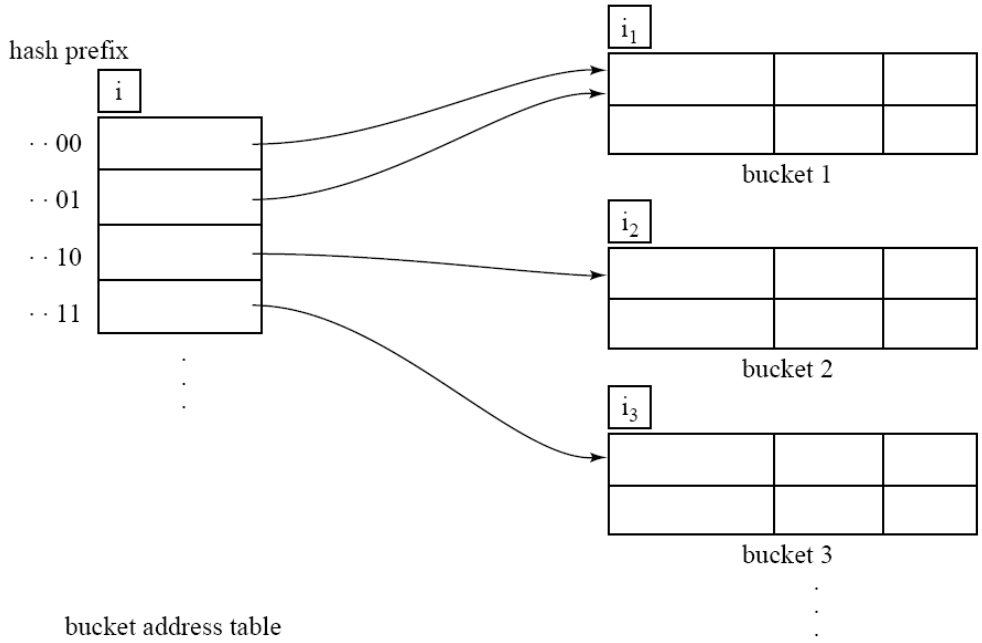
  - Jednou z možností je pravidelná reorganizace souboru s novou hešovací funkcí, ale to je velmi nákladné.
- Těmto problémům se můžeme vyhnout použitím techniky, která umožňuje dynamicky alokovat a rušit kyblíky.

Dynamické hešování

- Vhodné pro databáze, které mění svoji velikost
- Umožňuje modifikovat hešovací funkci
- **Rozšiřitelné hešování** – jednou z forem dynamického hešování
  - Hešovací funkce generuje velká čísla, typicky 32-bitová.
  - Vždy se používá pouze prefix pro získání adresy kyblíku. Délka prefixu je *i* bitů, $0 \leq i \leq 32$
  - Na počátku je $i = 0$
  - Hodnota *i* se zvyšuje nebo snižuje podle velikosti souboru
  - Aktuální počet kyblíků je menší než 2^i, to se také dynamicky mění kvůli slívání a rozdělování kyblíků.

Obecná struktura rozšiřitelného hešování

- Příklad struktury s $i_2 = i_3 = i$, ale $i_1 = i - 1$



Použití rozšiřitelného hešování

- Více adres kyblíků může ukazovat na stejný kyblík. Každý kyblík *j* ukládá hodnotu i_j. Všechny položky ukazující na stejný kyblík musí mít stejnou hodnotu na prvních i_j bitech.
- Pro nalezení kyblíku obsahující klíč K_j:
  1. vypočítej $h(K_j) = X$
  2. použij prvních *i* bitů hodnoty *X* pro vyhledání v tabulce adres kyblíků a následuj ukazatel do příslušného kyblíku

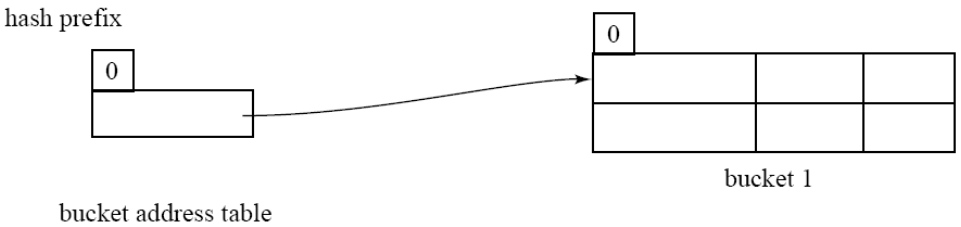
- Při vkládání záznamu s hodnotou klíče K_l proveď kroky uvedené výše a najdi kyblík *j*. Pokud je v kyblíku místo vlož nový záznam. Jinak musí být kyblík rozdělen a vkládání se opakuje.

Aktualizace rozšiřitelného hešování

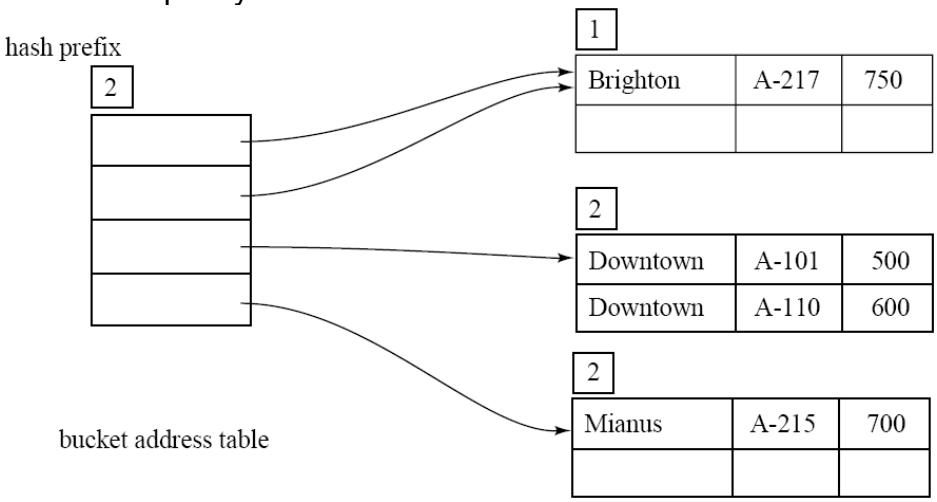
Pro rozdělení kyblíku *j* při vkládání nového záznamu s hodnotou vyhledávacího klíče K_l:

- Pokud $i > i_j$ (více než jeden ukazatel na kyblík *j*)
  - Vytvoř nový kyblík *z* a nastav i_j a i_z na hodnotu $i_j + 1$ (zde i_j je původní hodnota)
  - Změň polovinu ukazatelů ukazujících na *j*, aby ukazovaly na *z*.
  - Smaž a znovu vlož každý záznam z kyblíku *j*
  - Znovu vypočítej adresu kyblíku pro K_l a vlož záznam do kyblíku (další dělení je možné, pokud je kyblík stále plný)
- Pokud $i = i_j$ (pouze jeden ukazatel na kyblík *j*)
  - Zvyš hodnotu *i* a zdvojnásob velikost tabulky adres kyblíků
  - Nahraď každou položku v této tabulce dvěma novými položkami, které ukazují na stejný kyblík
  - Znovu vypočítej adresu kyblíku pro K_l. Nyní je $i > i_j$, tak použij předchozí postup.
- Při vkládání hodnoty může dojít k vícenásobnému dělení kyblíku, pokud se dělí stále stejný a počet dělení překročí nastavenou mez, vytvoř přetokový kyblík místo dalšího dělení.
- Pro smazání hodnoty najdi příslušný kyblík a smaž hodnotu. Samotný kyblík může být uvolněn až je úplně prázdný (ovšem s odpovídající změnou v tabulce adres kyblíků). Slévání kyblíků do jednoho a snižování tabulky adres kyblíků jsou možné.
- Příklad s klíčem *jméno-pobočky (brach-name)* a velikostí kyblíku 2:

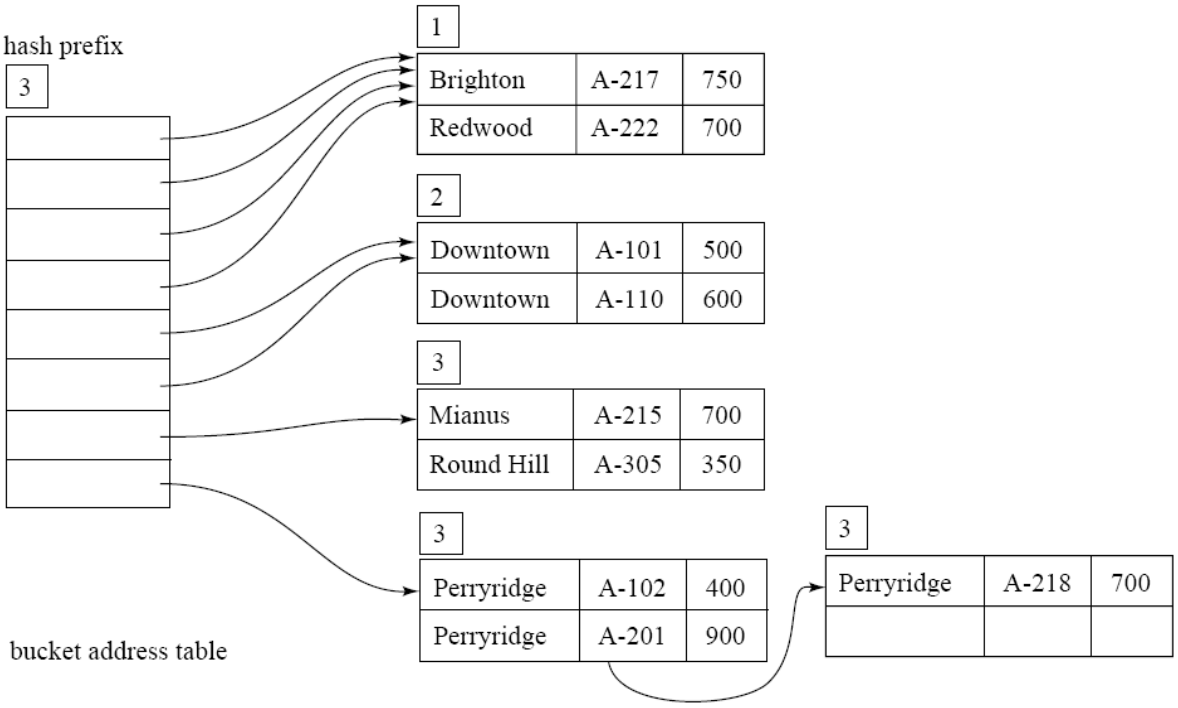
branch-name	h(*branch-name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111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



- Hešovací index po čtyřech vloženích:



- Výsledek po vložení všech klíčů:



Srovnání řazených indexů a hešování

Zvažované otázky:

- Náklady na pravidelnou reorganizaci
- Četnost vkládání a mazání
- Je žádoucí optimalizovat průměrnou dobu přístup vůči nejhoršímu případu doby přístupu?
- Očekávané typy dotazů:
  - Hešování je obecně lepší při vyhledávání záznamů mající danou hodnotu klíče.
  - Pokud jsou rozsahové dotazy běžné, je lepší použít řazené indexy

Definice indexů v SQL

- Vytvoření indexu
  create index <jméno_indexu> **on** <jméno_relace> (<seznam_atributů>)
  Např. **create index** *b-index* **on** *pobočka (jméno-pobočky)*
- Použitím **create unique index** nepřímo zajistíme podmínku, že vyhledávací klíč je kandidátní klíč.
- Pro zrušení indexu
  drop index <jméno_indexu>