B0B36DBS, BD6B36DBS: **Database Systems**

http://www.ksi.mff.cuni.cz/~svoboda/courses/192-B0B36DBS/

Lecture 8

Physical Layer

Authors: **Tomáš Skopal**, **Irena Holubová**

Lecturer: **Martin Svoboda**, martin.svoboda@fel.cvut.cz

7. 4. 2020

Czech Technical University in Prague, Faculty of Electrical Engineering

Today's lecture outline

- disk management, paging, buffer manager
- database files organization
- indexing
  - B^+-tree
  - bitmaps
  - hashing

Three layers of database modeling

abstraction
implementation

- conceptual layer
  - models a part of the **“structured”** real world relevant for applications built on top of your database
    - real world part = **real-world entities and relationships between them**
  - different conceptual models (e.g. ER, UML)
- logical layer
  - specifies how conceptual components are represented in logical machine interpretable data structures
  - different logical models (e.g. object, relational, object-relational, XML, graph, etc.)
- **physical layer**
  - specifies how logical database structures are implemented in a specific technical environment
  - data files, index structures (e.g. B+ trees), etc.

Introduction

- relations/tables are stored in files on the disk
- we need to organize table records within a file
  - efficient storage, update and access

Example:
Employees (name char(20), age integer, salary integer)

Paging

- records are stored in **disk pages** of fixed size (a few kB)
  - reason: hardware
    - assuming a magnetic disk based on rotational plates and reading heads
  - the data organization must be adjusted w.r.t. this mechanism
- the HW firmware can only access entire pages
  - I/O operations – reads, writes
- **real time for I/O operations =**

  = seek time + rotational delay + data transfer time
- sequential access to pages is much faster than random access
  - the seek time and rotational delay not needed

Example: reading 4 KB could take 8 + 4 + 0,5 ms = 12,5 ms;
i.e., the reading itself takes only 0,5 ms = 4% of the real time!!!

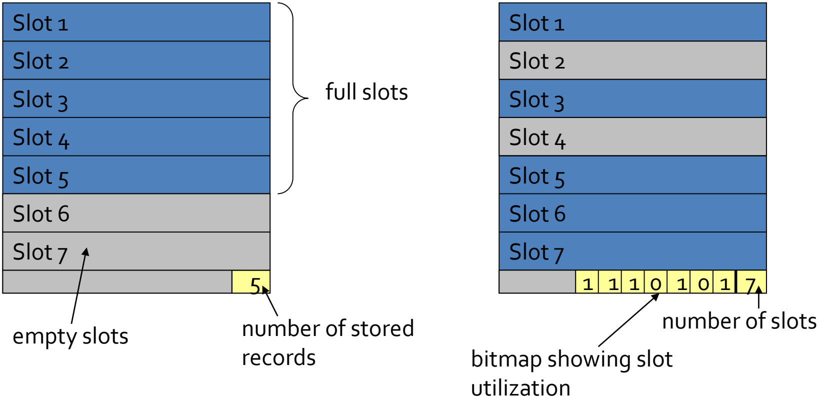
Paging

- I/O is a unit of time cost
- the page is divided into **slots**, that are used to store **records**
  - a record is identified by **rid** (record id) = **page id + slot id**
- a record can be stored
  - in multiple pages ⇒
    - better space utilization
    - need for more I/Os for record manipulation
  - in a single page (assuming it fits) ⇒
    - a part of page may not be used
    - less I/Os
  - ideally: records fit the entire page

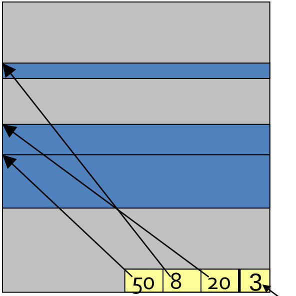
Paging

- only **fixed-size data types** are used in the record
  ⇒ fixed record size
- also **variable-size data types** are used in the record
  ⇒ variable size of the records,
  - e.g., types `varchar(X)`, `BLOB`, …
- fixed-size records = fixed-size slots
- variable-size records = need for slot directory in the page header

Fixed-size page organization, example



Variable-size page organization, example



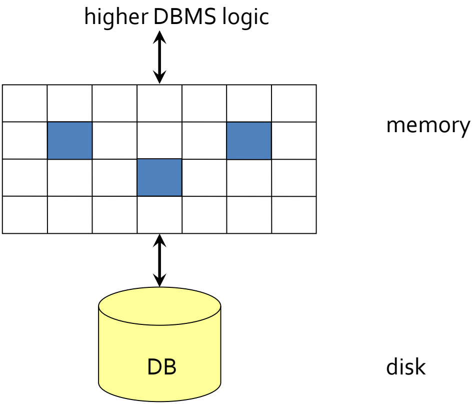
number of stored records

slot directory
(pairs offset + record size)

Buffer management

- **buffer** = a piece of main memory for temporary storage of disk pages
  - disk pages are mapped into **memory frames** 1:1
- every frame has 2 flags:
  - **pin_count** = number of references to the page in frame
  - **dirty** = indication of containing a modified record
- buffer manager
  - implements read and write operations for higher DBMS logic
- **read**: retrieves the page from buffer + increasing **pin_count**
  - if it is not there, it is first fetched from the disk
- **write**: puts the page into the buffer + setting **dirty**
- if the buffer is full (during read or write), some page must be replaced ⇒ various policies, e.g., LRU (least-recently-used), ↘ věci z OSY ☺
  - if the replaced page is **dirty**, it must be stored

Buffer management



Database storage

- **data files** – contain table data
- **index files** – speed up processing of queries
- **system catalogue** – contains metadata
  - table schemas
  - index names
  - integrity constraints, keys, etc.

Data files

1. heap → basically linear (linked list)
2. sorted file → logarithmic (trees, bisection).
3. hashed file → amortized constant (hash table).

} lookup

Observing average I/O cost of simple operations:

1) sequential access to records
2) searching records based on equality (w.r.t search key)
3) searching records based on range (w.r.t search key)
4) record insertion
5) record deletion

Cost model:

N = number of pages, R = records per page

Simple operations, SQL examples

- sequential reading of pages
  SELECT * FROM Employees
- searching on equality
  SELECT * FROM Employees WHERE age = 40
- searching on range
  SELECT * FROM Employees
  WHERE salary > 10000 AND salary < 20000
- record insertion
  INSERT INTO Employees VALUES (...)
- record deletion based on rid
  DELETE FROM Employees WHERE rid = 1234
- record deletion
  DELETE FROM Employees WHERE salary < 5000

Heap file

- records stored in pages are not ordered (e.g., according to key)
  - they are stored in the order of insertion
- page search can only be achieved by sequential scan (**`GetNext`** operation)
- quick record insertion (at the end of file)
- deletion problems: „holes“ (pieces of not utilized space)

Maintenance of empty heap pages

- double linked list
  - header + lists of full and non full pages

→ zase OSY

↳ to jsme

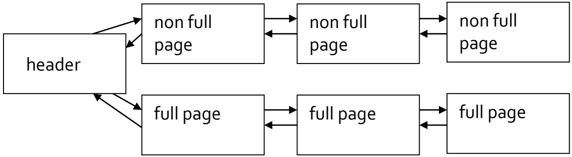
přímo implementovali

(MALLOC).

- page directory
  - linked list of directory pages
  - every item in the directory refers to a data page
    - flag = item utilization

Maintenance of empty heap pages

double linked list



header

non full page

non full page

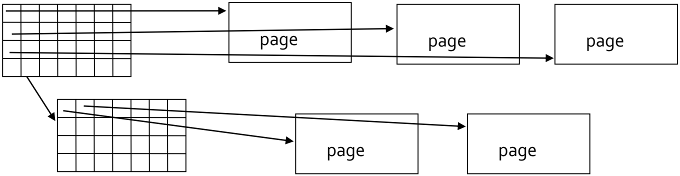
non full page

full page

full page

full page

page directory



page

page

page

page

page

Heap, cost of simple operations

- sequential access = N
- search on equality = N
- search on range = N
- record insertion = 1 → good for insertion
- record deletion
  - 2,
    assuming **rid** based search costs 1 I/O
  - N or 2*N,
    if deleted based on equality or range

Sorted file

- records stored in pages based on an ordering according to a search key
  - single or multiple attributes
- file pages maintained contiguous, i.e., no „holes“
- fast: search on equality and/or range
- slow: insertion and deletion
  - „moving“ the rest of pages
- in practice:
  - sorted file at the beginning
  - each page has an overhead space where to insert
  - if the overhead space is full, update pages are used (linked list)
  - a reorganization needed from time to time
    - i.e., sorting

= so basically combined with the previous one.

Sorted file, cost of simple operations

- sequential access = N
- search on equality = $\log_2 N$
- search on range = $\log_2 N + M$
  - where M is the number of relevant pages

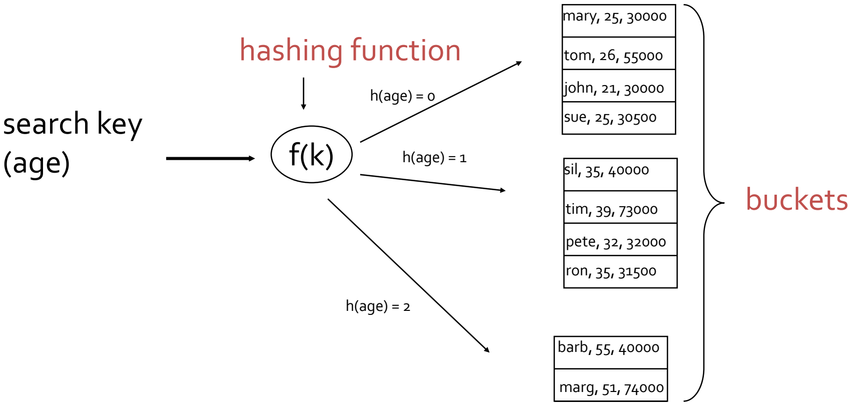
like in tree (binary search).

- record insertion = N
- record deletion = $\log_2 N + N$ (based on key)

Hashed file

- organized in K buckets
  - a bucket is extensible to multiple disk pages
- a record is inserted into/read from a bucket determined by hashing function *f* applied on search key
  - **bucket id = f(key)**
- if the bucket is full, new pages are allocated and linked to the bucket (linked list)
- fast search / deletion on equality
- higher space overhead, problems with chained pages (solved by dynamic hashed techniques)

Hashed file



search key (age)

hashing function

f(k)

h(age) = 0

h(age) = 1

h(age) = 2

mary, 25, 30000
tom, 26, 55000
john, 21, 30000
sue, 25, 30500

sil, 35, 40000
tim, 39, 73000
pete, 32, 32000
ron, 35, 31500

barb, 55, 40000
marg, 51, 74000

buckets

Hashed file, cost of simple operations

- sequential access = N
- search on equality = N/K (best case)
  - K = number of buckets ↳ OK, so number of buckets is important.
- search on range = N
- record insertion = N/K (best case)
- deletion on equality = N/K + 1 (best case)

Indexing

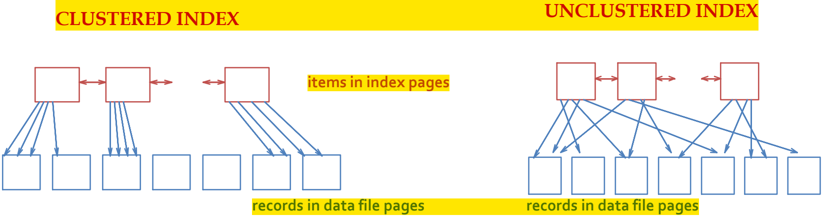
- **index** is a helper structure that provides fast search based on search key(s)
- organized into disk pages (like data files)
  - usually different file than data files
- usually contains only search keys and links to the respective records
  - i.e., rid
- need much less space than data files
  - e.g., 100x less

Indexing, principles

- index item can contain
  - the whole record (then index and data file are the same)
  - pair **`<key, rid>`**
  - pair **`<key, rid-list>`**, where **`rid-list`** is a list of links to records with the same search key value

1. **clustered:** ordering of index items is (almost) the same as ordering in the data file
   - tree-based index, index containing the entire records, hashed index, …
   - primary key = search key used in clustered index
2. **unclustered:** the order of search keys is not preserved

Indexing, principles



Clustered index:

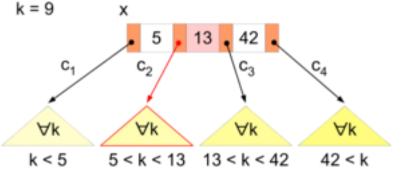
Pros: huge speedup when searching on range – result record pages are read sequentially from data file

Cons: large overhead for keeping the data file sorted

B$^+$-tree

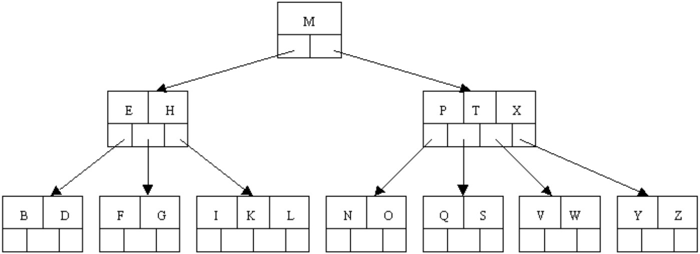
↳ here comes

ALG



- extends B-tree
  - balanced tree-based index
- provides logarithmic complexity for insertion, search on equality (no duplicates), deletion on equality (no duplicates)
- guarantees 50% node (page) utilization
- B$^+$-tree extends B-tree by
  - all **keys are in the leaves** – inner nodes contain indexed intervals
  - **linking leaf pages** for efficient range queries

Princip B-stromu

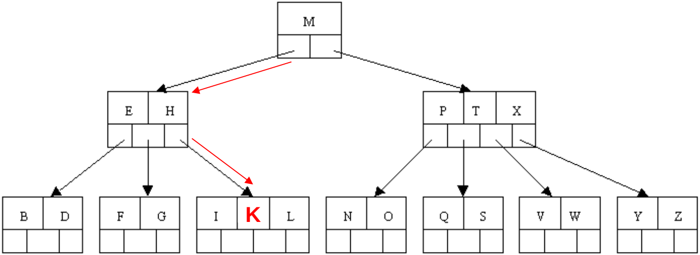


B-strom má definovánu:

- maximální kapacitu uzlu (max. počet záznamů v uzlu)
- minimální kapacitu uzlu (min. počet záznamů v uzlu)

Záznamy uvnitř uzlu jsou setříděné podle hodnoty klíče.

Princip B-stromu



- Hloubka B-stromu

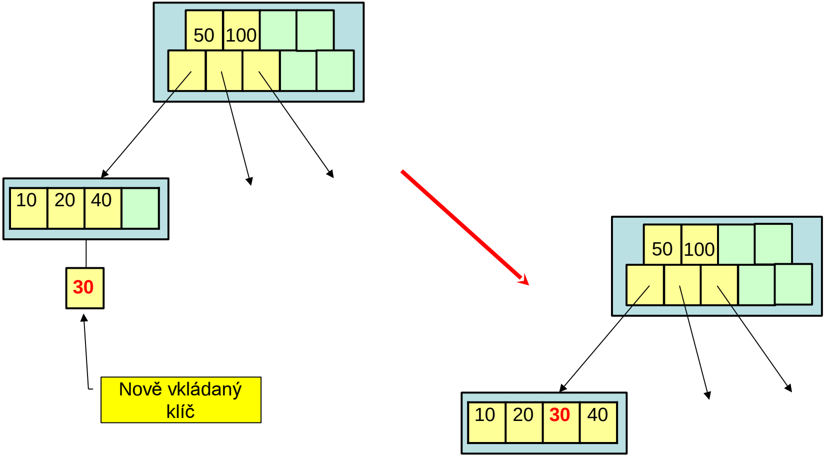
v nejlepším případě (všechny uzly naplněny na 100%) ... $\log_{\max} n$

v nejhorším případě (všechny uzly naplněny na *min*) ... $\log_{\min} n$

Max / min … maximální / minimální počet záznamů v uzlu
n ... počet záznamů v databázi

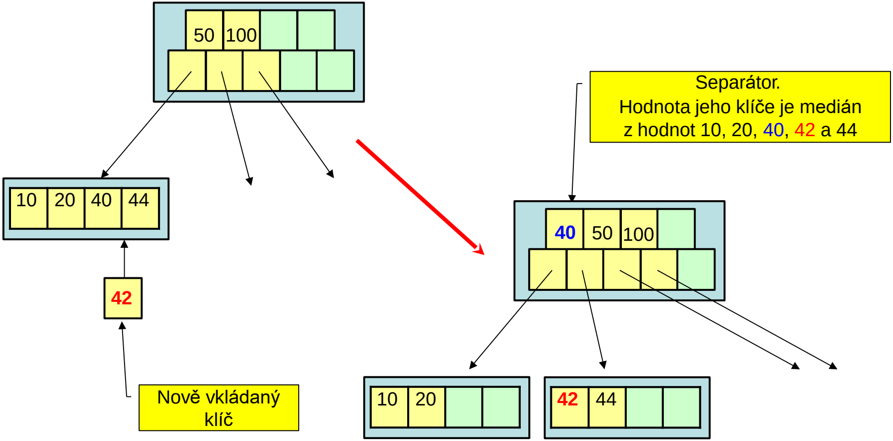
Vkládání záznamu do B-stromu

Triviální, není-li kapacita daného uzlu naplněna



Vkládání záznamu do B-stromu

Je-li kapacita daného uzlu naplněna, musí dojít k rozdělení uzlů:



Separátor.
Hodnota jeho klíče je medián z hodnot 10, 20, 40, 42 a 44

Nově vkládaný klíč

Vkládání záznamu do B-stromu

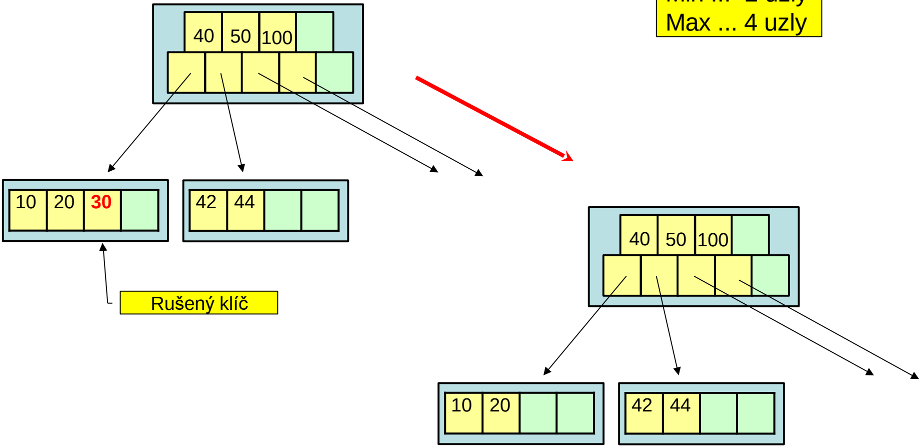
Algoritmus:

1. Pokud uzel, do něhož máme přidat nový záznam, obsahuje méně záznamů, než je jeho kapacita (maximální počet záznamů, který se „vejde“ do jednoho uzlu), vložíme nový záznam do tohoto uzlu
   při zachování uspořádání záznamů.
2. V opačném případě (uzel je naplněn na svou kapacitu), rozdělíme stávající uzel na dva uzly.
   a. Nalezneme separační záznam jako medián množiny klíčů, která je tvořena klíči záznamů existujících v děleném uzlu plus klíče vkládaného uzlu.
   b. Záznamy s klíči menšími než klíč separačního záznamu necháme v původním uzlu, záznamy s klíčem větším než klíč separačního záznamu uložíme do nového uzlu (zařazeného napravo od původního uzlu).
   c. Separační záznam vložíme do rodičovského uzlu, který se zase může rozlomit, pokud jeho kapacita již byla naplněna. Pokud uzel nemá rodiče (t.j. byl kořenem B-stromu), vytvoříme nový kořen nad tímto uzlem (dojde ke zvětšení hloubky stromu).

Rušení záznamu v listu B-stromu

Základní varianta

Min ... 2 uzly
Max ... 4 uzly

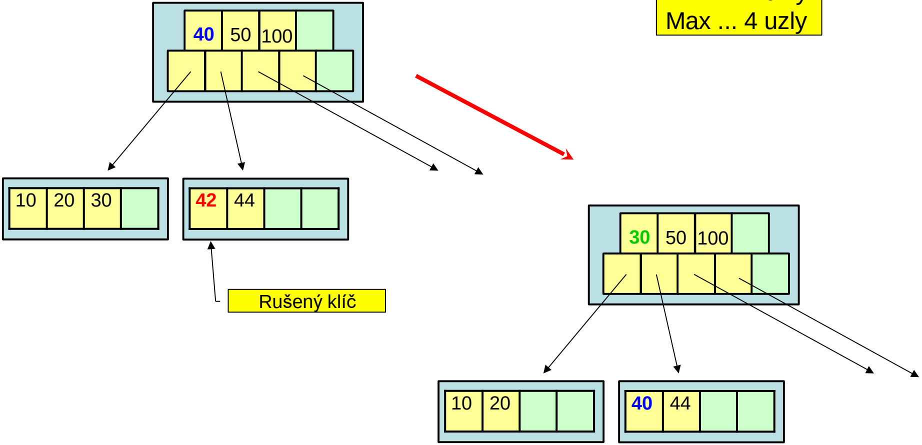


Rušený klíč

Rušení záznamu v listu B-stromu

Přesun hodnot

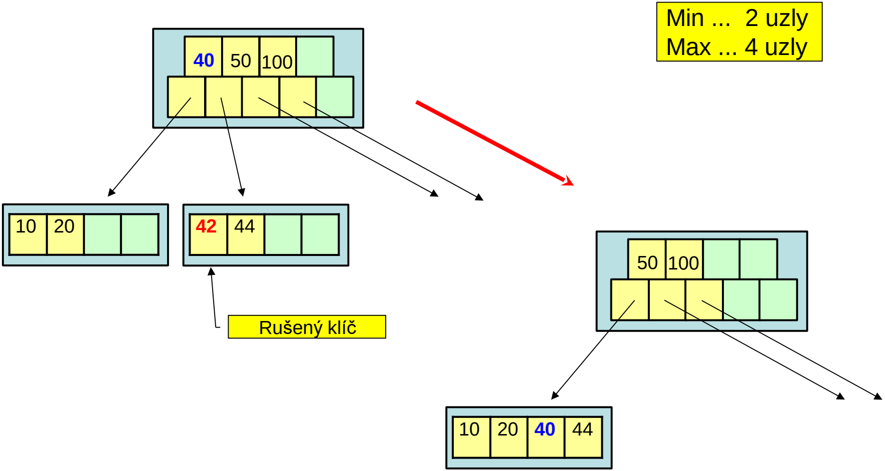
Min ... 2 uzly
Max ... 4 uzly



Rušený klíč

Rušení záznamu v listu B-stromu

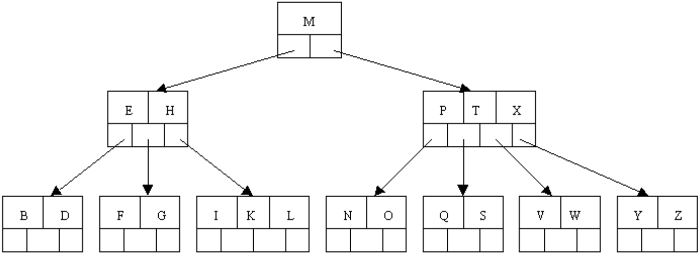
Spojení uzlů



Rušení záznamu v listu B-stromu

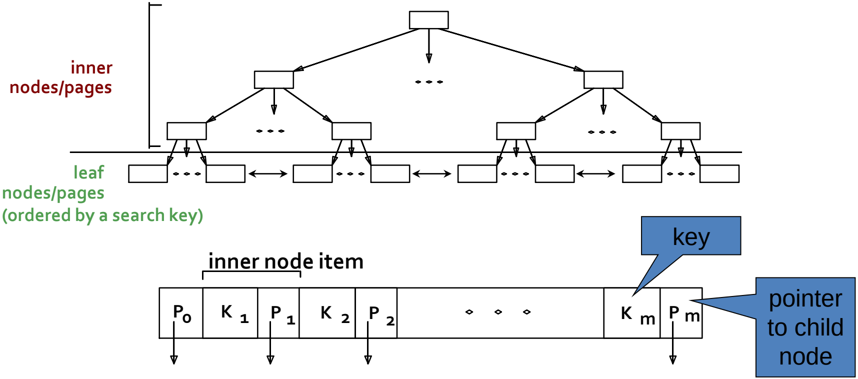
- Rušení záznamu v **listu** B-stromu je jednodušší než rušení záznamu v nelistovém uzlu.
- Pokud po zrušení záznamu klesne počet záznamů pod minimální kapacitu a sourozenecký uzel má více záznamů, než je minimální kapacita uzlu, **přesuneme** záznam ze sourozeneckého uzlu.
- Klesne-li počet záznamů v obou sourozeneckých uzlech pod minimální kapacitu uzlu, uzly **spojíme**.

Od B-stromu k B+-stromu



Hodnoty nemusí být unikátní
Hodnoty jsou jen v listech
Listy obsahují rid záznamů

B+-tree, schema



Hashed index

- similar to hashed data file
  - i.e., buckets + hashing function
- buckets contain only key values together with the **rid**s
- same pros/cons

Bitmaps

- suitable for indexing attributes of low-cardinality data types (like bool etc.)
  - e.g., attribute **FAMILY_STATUS** = {single, married, divorced, widow}
- for each value ***h*** of an indexed attribute ***a*** a bitmap (binary vector) is constructed, where 1 on ***i^{th}*** position means the value ***h*** appears in the ***i^{th}*** record (in the attribute ***a***), while it holds
  - bitwise OR = 1 (every attribute has a value)
  - bitwise AND = 0 (attribute values are deterministic)

Name	Address	Family status
John Smith	London	single
Rostislav Drobil	Prague	married
Franz Neumann	Munich	married
Fero Lakatoš	Malacky	single
Sergey Prokofjev	Moscow	divorced

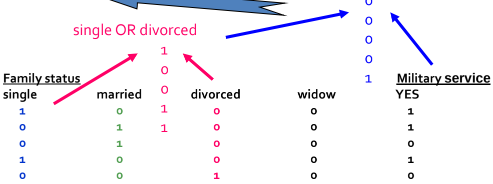
single	married	divorced	widow
1	0	0	0
0	1	0	0
0	1	0	0
1	0	0	0
0	0	1	0

Bitmaps

- query evaluation
  - bitwise operations with attribute bitmaps
  - resulting bitmap marks the queried records
- example
  - **Which single or divorced people did not complete the military service?**
    (bitmap(**single**) **OR** bitmap(**divorced**)) **AND not** bitmap(**YES**)

answer: Sergey Prokofjev, Moscow

Name	Address	Military service	Family status
John Smith	London	YES	single
Rostislav Drobil	Prague	YES	married
Franz Neumann	Munich	NO	married
Fero Lakatoš	Malacky	YES	single
Sergey Prokofjev	Moscow	NO	divorced



(single OR divorced) AND not YES

single OR divorced

Family status single	married	divorced	widow	Military service YES	single OR divorced	(single OR divorced) AND not YES
1	0	0	0	1	1	0
0	1	0	0	1	0	0
0	1	0	0	0	0	0
1	0	0	0	1	1	0
0	0	1	0	0	1	1

Bitmaps

- pros
    - efficient storage, could be also compressed
    - fast query processing, bitwise operations are fast
    - easy parallelization
- cons
    - suitable only for attributes with small cardinality domain
    - range queries get slow linearly with the number of values in the range (bitmaps for all the values must be processed)