Tree-Structured Indexes

Werner Nutt

Introduction to Database Systems

Free University of Bozen-Bolzano

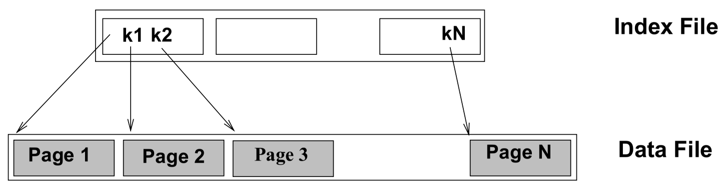
Introduction

- As for any index, three alternatives for data entries K^*:
  - Data record with key value K
  - $\langle K, r \rangle$, where r is rid of a record with search key value K
  - $\langle K, [r_1, \ldots, r_n] \rangle$, where $[r_1, \ldots, r_n]$ is a list or rid's of records . with search key value K
- Choice orthogonal to *indexing technique* used to locate entries K^*.
- Tree-structured indexing techniques support both *range searches* and *equality searches*.
- **ISAM:** static structure;
  B+-tree: dynamic, adjusts gracefully under inserts and deletes.

Range Searches

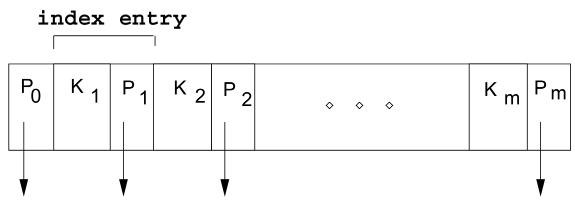
- *"Find all employees with sal > 1500"*
  - If data is in sorted file, do binary search to find first such employee, then scan to find others
  - Cost of binary search can be quite high

- Simple idea: create an "index" file



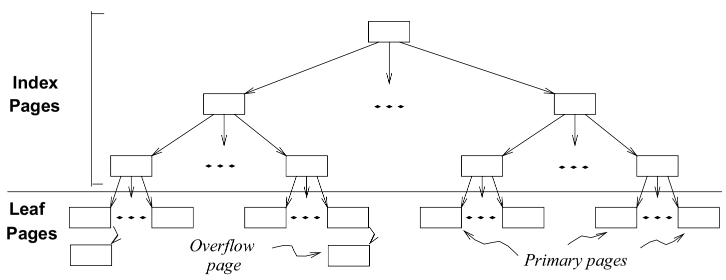
$\rightsquigarrow$ can do binary search on (smaller) index file!

ISAM (= Indexed Sequential Access Method)



Index file may still be quite large.

But we can apply the idea repeatedly!



$\rightsquigarrow$ Leaf pages contain data entries

Comments on ISAM

File creation: Leaf (data) pages allocated sequentially, sorted by search key; then index pages allocated, then space for overflow pages.

Index entries: $\langle$*search key value, page id*$\rangle$; 'direct' search for *data entries,* which are in leaf pages

Search: Start at root; use key comparisons to go to leaf.
Cost $\propto \log_F N$ where $F = \#$ entries/index page ('fanout') and $N = \#$ leaf pages

Insert: Find leaf data that entry belongs to, and put it there

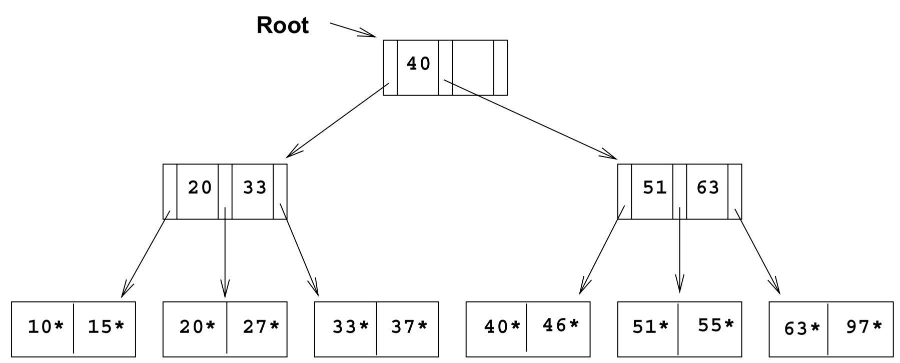
Delete: Find leaf and remove from leaf;
if empty overflow page, de-allocate

$\leadsto$ Static tree structure: *inserts/deletes affect only leaf pages*

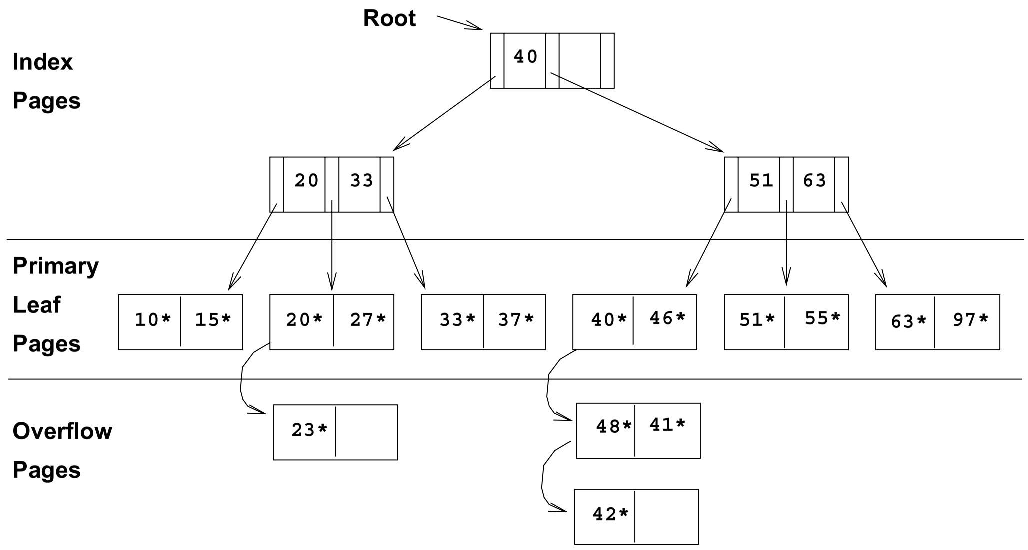
Example ISAM Tree

Each node can hold 2 entries

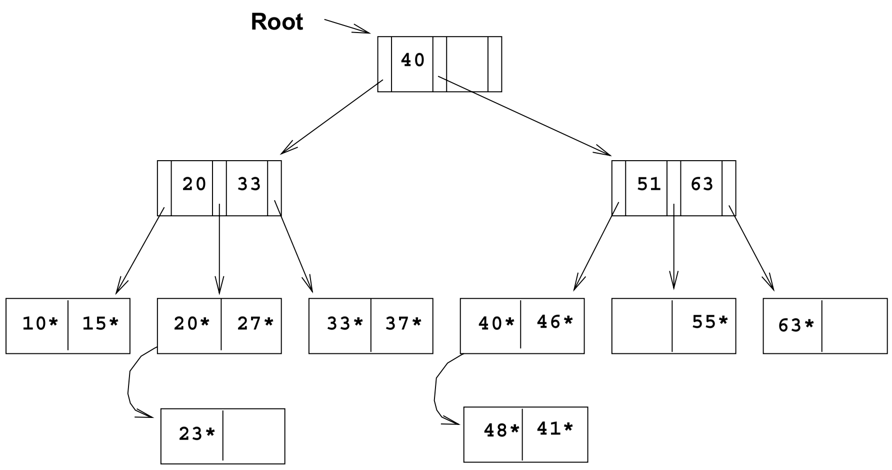
No need for 'next-leaf-page' pointers (Why?)



After Inserting 23^*, 48^*, 41^*, 42^*, ...



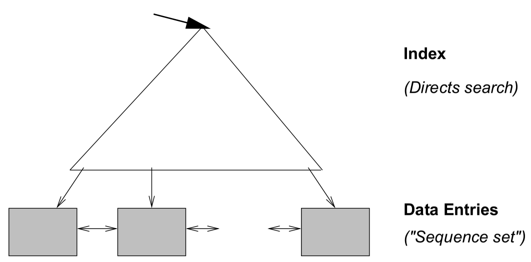
Then Deleting 42^*, 51^*, 97^*



Note that 51^* *appears in index levels, but not in leaf!*

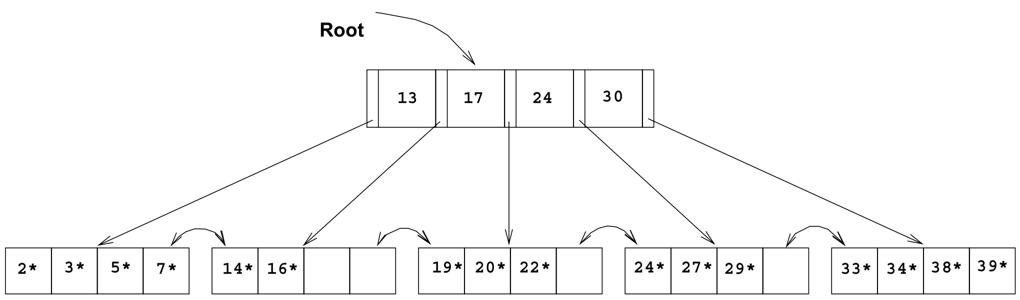
B+-Tree: The Most Widely Used Index

- Insert/delete at $\log_F N$ cost ($F =$ 'fan out' and $N = \#$ leaf pages); keep tree *height-balanced.*
- Minimum 50% occupancy (except for root).
- Each node contains $\boxed{d \leq m \leq 2d}$ entries (d is the *order* of the tree).
- Supports equality and range-searches efficiently.



Example B+-Tree

- Search begins at root, and key comparisons direct it to a leaf *(as in ISAM)*
- Search for 5*, 15*, all data entries with key $\geq 24^*$



$\leadsto$ *Based on the search for* 15^*, *we* **know** *it is not in the tree!*

B+-Trees in Numbers

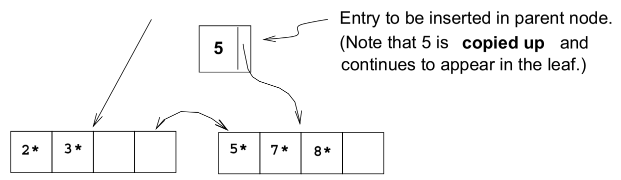
- Average fill-factor: 66% ($= \ln 2$)
- Typical order: 100
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 KBytes
  - Level 2 = 133 pages = 1 MByte
  - Level 3 = 17,689 pages = 133 MBytes

Inserting a Data Entry into a B+-Tree

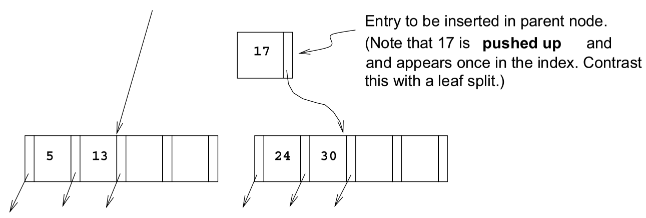
- Find correct leaf L
- Put data entry onto L
  - If L has enough space, *done!*
  - Else, must *split L* *(into L and a new node L')*
    - Redistribute entries evenly, **copy up** middle key
    - Insert index entry pointing to L' into parent of L
- This can happen recursively
  - To split index note, redistribute entries evenly,
    but **push up** middle key (contrast with leaf splits!)
- Splits "grow" three; root split increases height
  - Tree growth: gets *wider* or *one level taller at top*

Inserting 8^* into Example B+-Tree

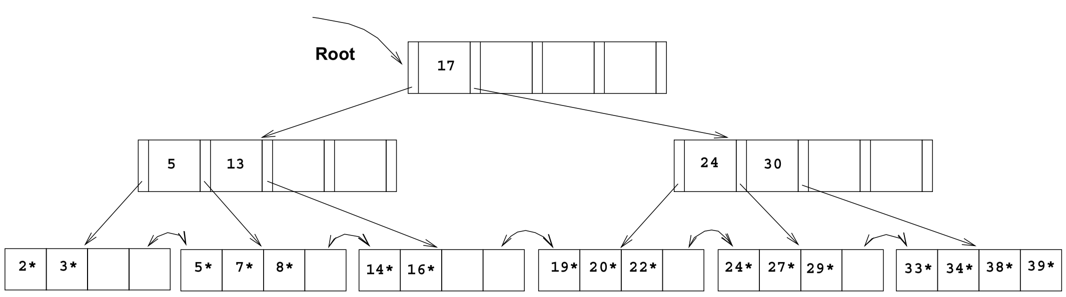
- Observe how minimum occupancy is guaranteed in both leaf and index page splits.



- Note difference between **copy up** and **push up**! What's the reason?



. . . After Inserting 8^*

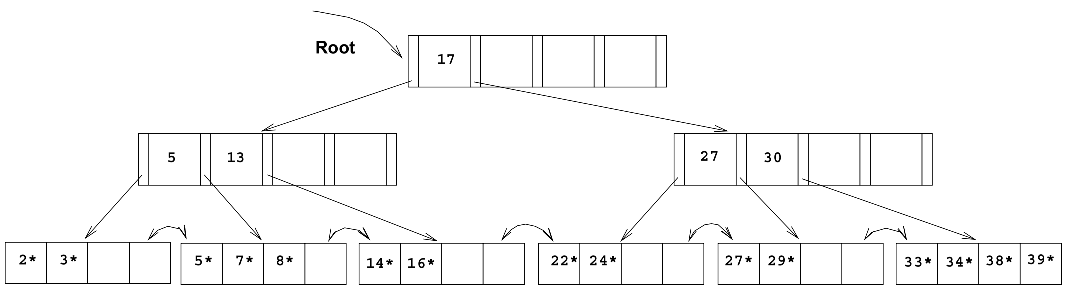


- Notice that root was split, leading to increase in height
- In this example, we can avoid split be re-distributing entries; however, this is usually not done in practice

Deleting a Data Entry from a B+-Tree

- Start at root, find leaf L where entry belongs
- Remove the entry
  - If L is at least half-full, *done!*
  - If L has only $d-1$ *entries*,
    - Try to *re-distribute*, borrowing from *sibling* *(adjacent node with same parent as L)*
    - If re-distribution fails, *merge* L and sibling
- If merge occurred, must delete entry (pointing to L or sibling) from parent of L
- Merge could propagate to root, decreasing height

... after (Inserting 8*, then) Deleting 19* and 20*

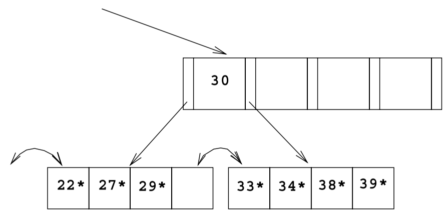


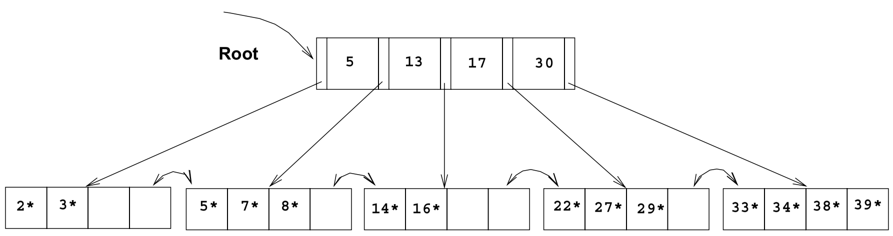
- Deleting *19** is easy
- Deleting *20** is done with re-distribution.

Notice how middle key is *copied up!*

. . . and then Deleting 24*

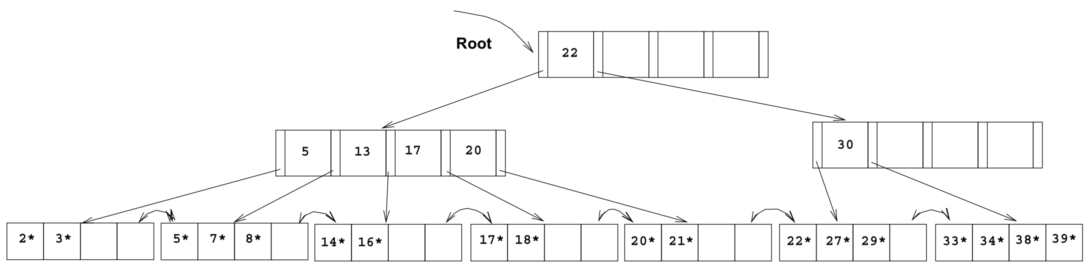
- Must merge
- observe "*toss*" of index entry (on right), and "*pull down*" of index entry (below)





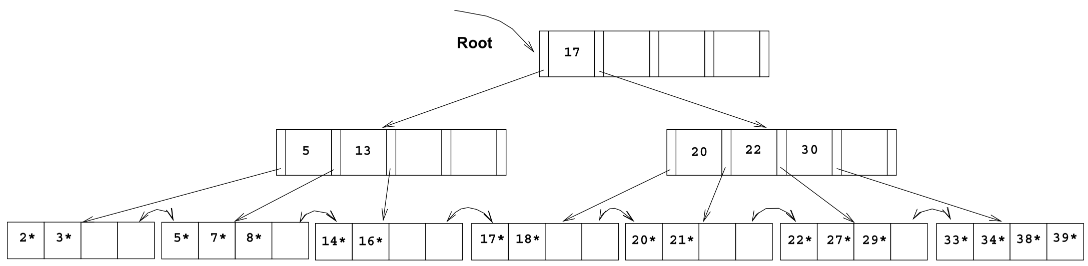
Example of Non-leaf Re-distribution

- Tree is shown below *during deletion* of *24**
  (What could be a possible tree?)
- In contrast to previous example, can re-distribute entry form left child of root to right child



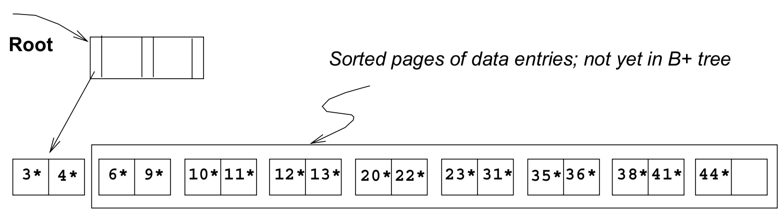
After Re-distribution

- Intuitively, entries are re-distributed by *"pushing through"* the splitting entry in the parent node
- It suffices to re-distribute index entry with key 20;
  (we have re-distributed 17 as well for illustration)

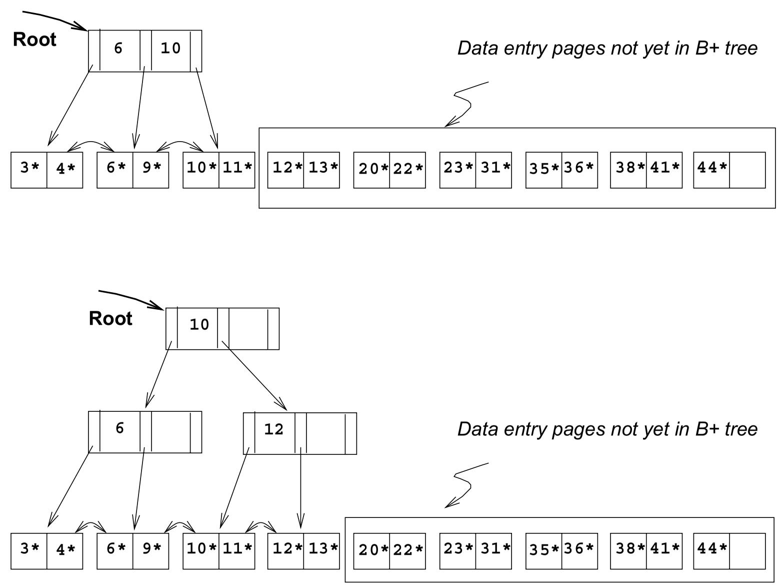


Bulk Loading of a B+-Tree

- If we have a large collection of records, and we want to create a B+-tree on some filed, doing so by repeatedly inserting records is very slow
- **Bulk loading** can be done much more efficiently
- *Initialisation:* Sort all data entries, insert pointer to first (leaf) page in a new (root) page



Bulk Loading of a B+-Tree (Cntd.)

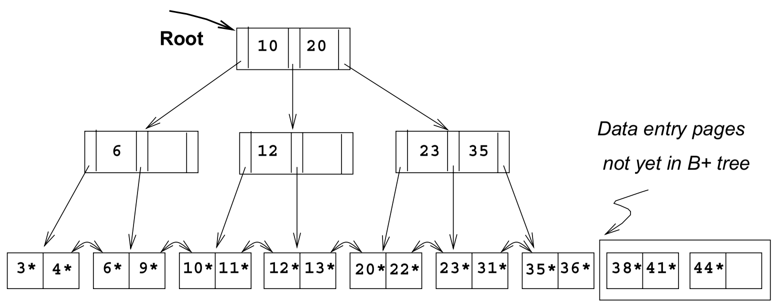


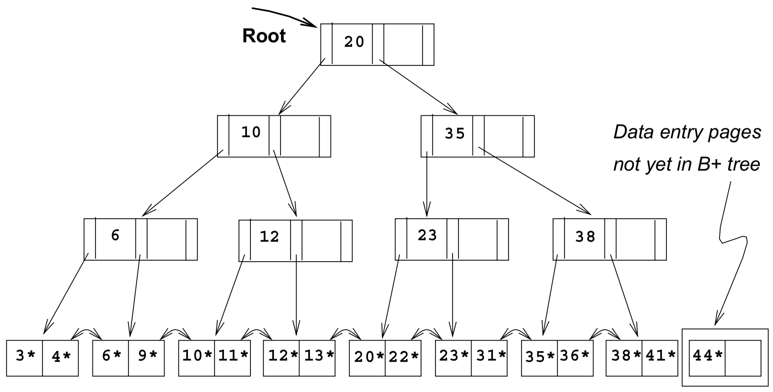
Bulk Loading of a B+-Tree (Cntd.)

- Index entries for leaf pages always entered into right-most index page just above leaf level.
  When this fills up, it splits.
  (Split may go up right-most path to the root)

- Much faster than repeated inserts, especially when one considers locking!

Bulk Loading of a B+-Tree (Cntd.)





Summary of Bulk Loading

- Option 1: multiple inserts
  - Slow
  - Does not give sequential storage of leaves
- Option 2: **Bulk Loading**
  - Has advantages for concurrency control
  - Fewer I/O's during build
  - Leaves will be stored sequentially (and linked, of course)
  - Can control "fill factor" on pages

Storage and Access Cost for an Average B+-tree

Example: Relation `Orders` with attribute `Orders.CustId`

Assumptions:

Page Size: 4KBytes (including 96 Bytes page header)

Occupancy of Page: 70 %

Number of records in `Orders`: 10,000,000

Number of distinct Customer ID's: 100,000
(for every customer, there is an equal number of orders)

Length of a Customer ID: 24 Bytes

Length of an rid: 6 Bytes

Length of a pointer in B+-tree: 6 Bytes

We Conclude:

Length of Rid List: $24 + 100 \times 6$ Bytes $= 624$ Bytes

Number of Rid Lists on an Index Page: $\lfloor .7 \times (4096 - 96)/624 \rfloor = 4$

Number of Index Pages: $\lceil 100,000/4 \rceil = 25,000$

Length of a "Signpost" to a Non-leaf Node: $24 + 6$ Bytes $= 30$ Bytes

Fanout: $\lfloor .7 \times (4096 - 96)/30 \rfloor = 93$

Height of Index: $\lceil \log_{93} 25,000 \rceil + 1 = 4$
(3 Levels for non-leaf nodes plus leaf level)

Number of Pages in Index: $25,000$ pages on Level 4,
$\lceil 25,000/93 \rceil = 269$ non-leaf nodes on Level 3
$\lceil 269/93 \rceil = 3$ non-leaf nodes on Level 2 plus
1 root node

Storage Space: $25,270 \times 4$ KBytes ≈ 100 MBytes

$\leadsto$ Reading all orders for a CustId requires 4 + 100 = 104 page accesses

Tree-structured Indexes: Summary

- Ideal for **range-searches,** also good for **equality searches**
- **ISAM** is a **static structure**
  - Only leaf pages modified; *overflow pages* needed
  - Overflow chains can degrade performance unless size of data set and data distribution stay constant
- **B+-tree** is a **dynamic structure**
  - Inserts/deletes leave tree *height-balanced* ($\log_F N$ cost)
  - *High fanout* F means depth rarely more than 3 or 4
  - Almost always better than maintaining a sorted file
  - Typically, *66% (=* $\ln 2$*) occupancy* on average
  - If data entries are data records, splits can change rids!

Tree-structured Indexes: Summary

- *Bulk loading* can be much faster than repeated inserts for creating a B+-tree on a large data set
- *Most widely used* index in database management systems because of its versatility. On of the most optimized components of a DBMS.

References

These slides are based on Chapter 10 of the book *Database Management Systems* by R. Ramakrishnan and J. Gehrke, and on slides by the authors published at

`www.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html`