# Information Systems
# (Informationssysteme)
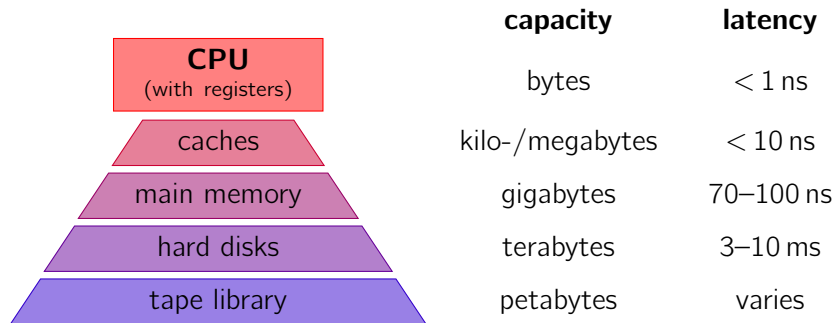
Jens Teubner, TU Dortmund
`jens.teubner@cs.tu-dortmund.de`

Summer 2018

# Part IX

## B-Trees

# Memory Hierarchy

| | capacity | latency |
|---|---|---|
| **CPU** (with registers) | bytes | $< 1\,\text{ns}$ |
| caches | kilo-/megabytes | $< 10\,\text{ns}$ |
| main memory | gigabytes | 70–100 ns |
| hard disks | terabytes | 3–10 ms |
| tape library | petabytes | varies |

- fast, but expensive and small, memory close to CPU
- larger, slower memory at the periphery
- Try to **hide latency** by using the fast memory as a **cache**.

## Latency vs. Bandwidth

"Slow" memory typically means **high latency**.

Example: Samsung HD642JJ Hard Drive (640 GB, SATA 3)
- rotational speed: 7200 rpm
- sequential read bandwidth: $\approx 106\,\text{MB/s}$ ($\nearrow$ `hdparm -t`)
- random access time: 15.2 ms (measured)

---

✎ **Time it takes to read 1,024 random 4 kB blocks?**

---

# Ways to Improve I/O Performance

The latency penalty is hard to avoid.

**However:**

- Throughput can be increased rather easily by exploiting **parallelism**.
- **Idea:** Use multiple disks and access them in parallel.

---

**⌃ TPC-C: An industry benchmark for OLTP**

The current number one system (Oracle 11g RAC on SPARC) uses

- 11,040 flash drives (24 GB each) and 720 hard drives **(!)**
  (plus drives for OS, etc.),
- connected with 8 Gbit Fibre Channel,
- yielding 30 tpmC ($\approx$ 60 M transactions per minute).

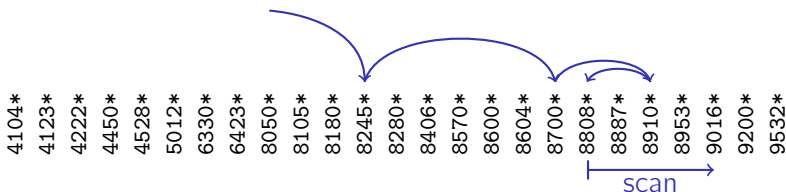---

To combat the latency problem:

1. Databases access and organize the disk with a **page granularity**.
   - Read larger chunks to amortize high latency.
   - Page size: at least 4 kB, better more; up to $\approx 64$ kB.

2. Use **sequential access** and/or aggressive **prefetching** (read-ahead).
   - But must read **many** pages ahead to actually avoid penalty.

# Finding a Needle in a Haystack

```
SELECT *
  FROM CUSTOMERS
 WHERE ZIPCODE BETWEEN 8800 AND 8999
```
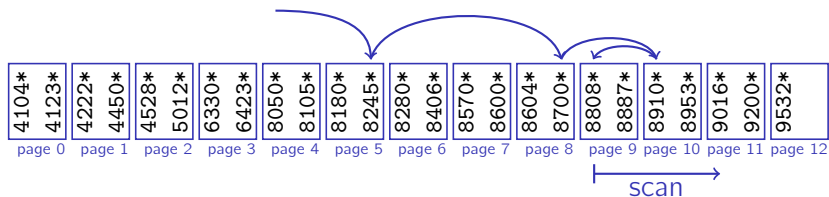
To answer this query, we could

1. **sort** the table on disk (in ZIPCODE order).
2. To answer queries, then use **binary search** to find first qualifying tuple, and **scan** as long as ZIPCODE < 8999.



4104* 4123* 4222* 4450* 4528* 5012* 6330* 6423* 8050* 8105* 8180* 8245* 8280* 8406* 8570* 8600* 8604* 8700* 8808* 8887* 8910* 8953* 9016* 9200* 9532*

scan

---

$k*$ denotes the full data record with search key $k$.

4104* 4123* | 4222* 4450* | 4528* 5012* | 6330* 6423* | 8050* 8105* | 8180* 8245* | 8280* 8406* | 8570* 8600* | 8604* 8700* | 8808* 8887* | 8910* 8953* | 9016* 9200* | 9532*

page 0  page 1  page 2  page 3  page 4  page 5  page 6  page 7  page 8  page 9  page 10  page 11  page 12

scan →

✔ Need to read only $\log_2(\#\ tuples)$ to find the first match.

✘ Need to read about as many **pages** for this.
(The whole point of binary search is that we make far, unpredictable jumps. This largely defeats prefetching.)
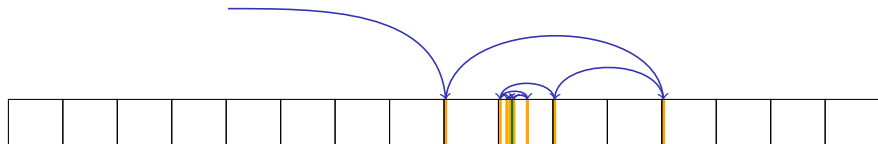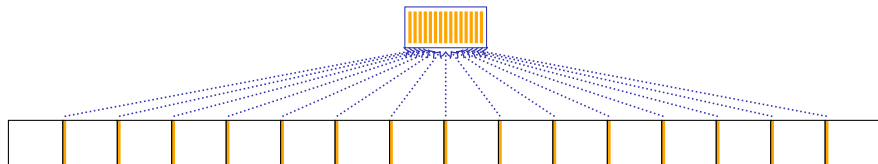
**Observations:**

- Make rather **far jumps initially**.
  - $\rightarrow$ For each step read **full page**, but inspect only **one record**.
- Last $\mathcal{O}\left(\log_2 pagesize\right)$ steps stay **within one page**.
  - $\rightarrow$ I/O cost is used much more efficiently here.

**Idea:** "Cache" those records that might be needed for the first phase.



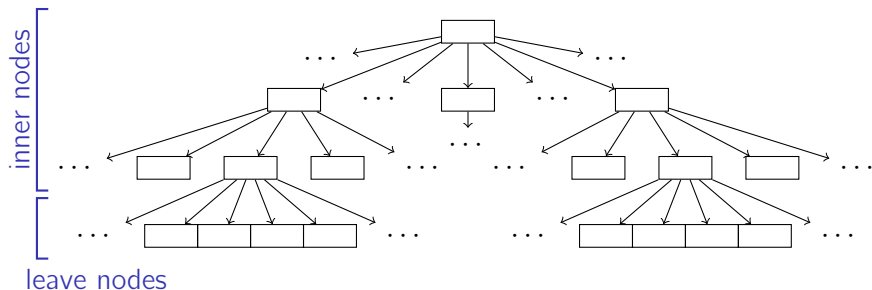$\rightarrow$ If we can keep the cache **in memory**, we can find **any** record with just a **single I/O**.

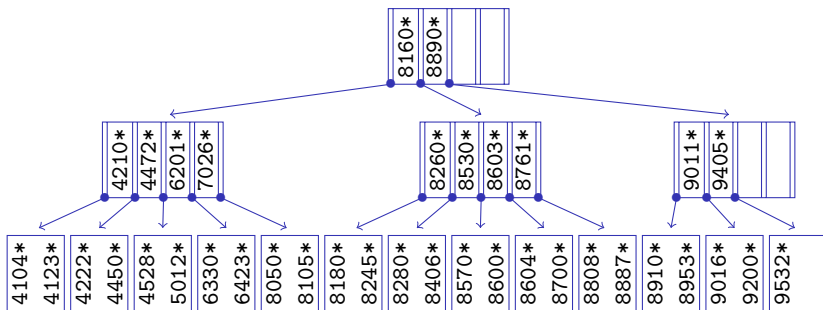✎ **Is this assumption reasonable?**

# Large Data

**What if my data set is really large?**

- "Cache" will span many pages, too.
  (In practice, we'll organize the cache just like any other database object.)
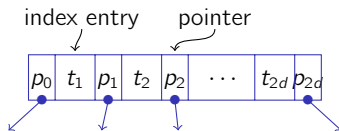- Thus: **"cache the cache"** → hierarchical "cache"



inner nodes

leave nodes

- **B-trees** are essentially such a "hierarchical cache."

# B-Trees



- All nodes are the size of a page
  - $\rightarrow$ hundreds of entries per page
  - $\rightarrow$ large fanout, low depth
- Search effort: $\log_{fanout}(\# \, tuples)$

$\nearrow$ Rudolf Bayer and Edward McCreight. *Organization and Maintenance of Large Ordered Indexes*, Acta Informatica 1(3), 1972.

# B-Trees

Each B-tree node contains

- A set of **index entries**, which include
  - the value of a **search key** (*e.g.*, 4711) and
  - **"associated information"** (indicated by $*$)
    (either a full data tuple or a reference to a tuple).
- A set of **child pointers**, pointing to a child page of the B-tree.

Each tree node (except the root) contains **at least** $d$ and **at most** $2d$ index entries ("minimum 50 % full"; on previous slide: $d = 2$).

$\rightarrow$ We call $d$ the **order** of the B-tree.

$\rightarrow$ In practice, $d$ is **large** (few hundreds).

B-trees are **balanced** at all times.

# Searching a B-Tree

1 **Function:**
   tree_search($k$, *node*)
2 **if** matching $*_i$ found on *node* **then**
3    | **return** $*_i$;
4 **if** *node* is a leaf node **then**
5    | **return** not found;
6 **switch** $k$ **do**
7    **case** $k < k_0$ **do**
8       **return**
      tree_search($k$, $p_0$);
9    **case** $k_i < k < k_{i+1}$ **do**
10      **return**
      tree_search($k$, $p_i$);
11    **case** $k_{2d} < k$ **do**
12      **return**
      tree_search($k$, $p_{2d}$);

- Invoke with
  *node* = root node.
- Note that B-trees are an
  **ordered** index structure.
  - $\rightarrow$ Support equality and
    range predicates

# B-Tree Modifications

**Goal:** Keep B-tree **balanced** at all times.[14]

> ✎ **Why is this desirable?**

**Thus:** Define routines for **insertion** and **deletion** that leave the B-tree properties intact.

---

[14] *I.e.*, every root-to-leaf path must have the same length.

## Inserting into a B-Tree

Sketch of the **insertion procedure** for entry $k*$:

1. **Find leaf page** $n$ where we would expect the entry for $k$.

2. If $n$ has **enough space** to hold the new entry (*i.e.*, at most $2d - 1$ entries in $n$), **simply insert** $k*$ into $n$.

3. Otherwise node $n$ must be **split** into $n$ and $n'$ and a new **separator** has to be inserted into the parent of $n$.

   Splitting happens recursively and may eventually lead to a split of the root node (increasing the tree height).

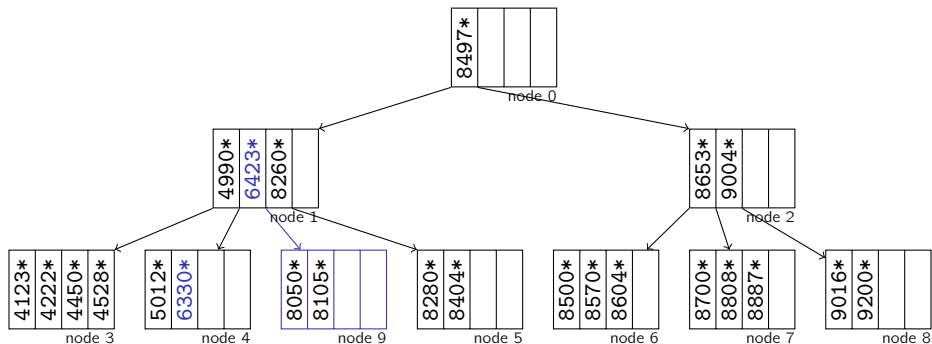   $\rightarrow$ B-trees grow at the root, not at the leaves!

# B-Tree Insert: Examples (Insert without Split)
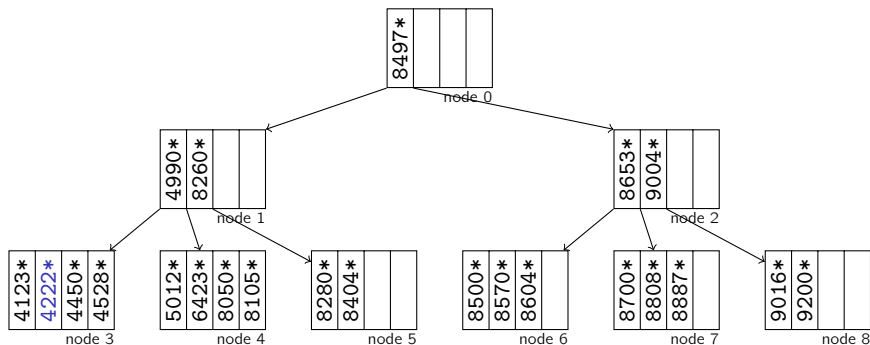

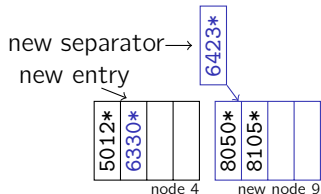
Insert new entry with key 4222.

$\rightarrow$ Enough space in node 3, simply insert.

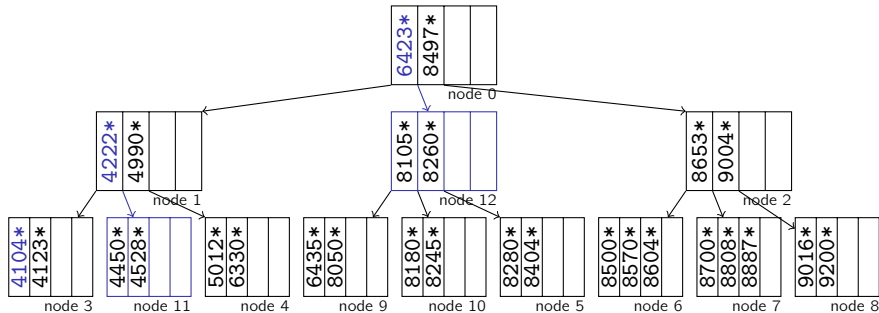# B-Tree Insert: Examples (Insert with Leaf Split)



Insert key 6330.

→ Must **split** node 4.

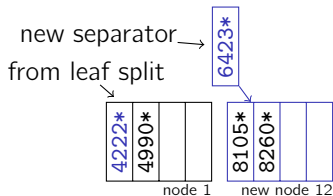→ **Middle entry** goes into node 1.

# B-Tree Insert: Examples (Insert with Inner Node Split)



After 8180, 8245, 6435 insert key 4104.

→ Must **split** node 3.
→ Node 1 overflows → split it
→ **New separator** goes into root

# Insert: Root Node Split

- Splitting starts at the leaf level and continues upward as long as index nodes are fully occupied.
- Eventually, this can lead to a split of the **root node**:
  - Split like any other inner node.
  - Use the separator to create a **new root**.
- The root node is the **only** node that may have an occupancy of less than 50 %.
- This is the **only** situation where the tree height increases.

✎ **How often do you expect a root split to happen?**

# Keys and Tuples

A B-tree maintains **key values** together with **"associated information"**.

The "associated information" ∗ can be

## Full Data Tuples

The B-tree becomes the mechanism to organize the table data

- → The table is **physically ordered** according to the key attribute.
- → We call this a **clustered index** or an **index-organized table**.
- → There can be at most one clustered index per table.

## Pointers to Actual Tuples

These pointers are also called **record identifiers** or **RIDs**.

- → Most systems use ⟨*pageno*, *pos. within page*⟩ to encode RIDs.
- → Such indexes are also called **secondary indexes**.
- → There can be arbitrarily many secondary indexes.

Many systems (*e.g.*, DB2) only support the latter index type.

# B-Trees $\rightarrow$ B$^+$-trees

Key to the efficiency of B-trees is their **high fanout**.

high fanout $\rightarrow$ low tree depth $\rightarrow$ fast root-to-leaf navigation
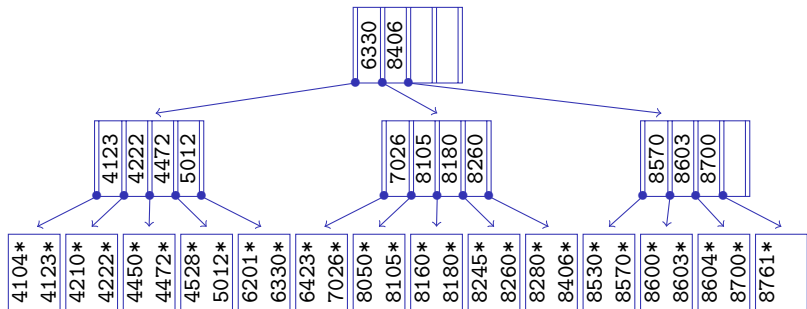
This gives incentive to **maximize fanout**:

- $\rightarrow$ Do **not** store * in **inner nodes**
  (Rather use that space to increase $d$ / store more keys.)
- $\rightarrow$ Inner nodes are then used for root-to-leaf navigation **only**.
- $\rightarrow$ For every data tuple, there is on leaf-level index entry.
- $\rightarrow$ The resulting index structure is then called **B$^+$-tree**.

**Real systems** today always use B$^+$-trees.

- $\rightarrow$ When database people say "B-tree," they typically mean "B$^+$-tree."

# $B^+$-trees



The root node contains keys 6330, 8406.

Inner nodes contain:
- 4123, 4222, 4472, 5012
- 7026, 8105, 8180, 8260
- 8570, 8603, 8700

Leaf nodes contain:
4104*, 4123*, 4210*, 4222*, 4450*, 4472*, 4528*, 5012*, 6201*, 6330*, 6423*, 7026*, 8050*, 8105*, 8160*, 8180*, 8245*, 8260*, 8280*, 8406*, 8530*, 8570*, 8600*, 8603*, 8604*, 8700*, 8761*

- Inner nodes do **not** store tuples or RIDs
  - $\rightarrow$ only used to navigate to leaves
  - $\rightarrow$ higher fanout, lower depth
- Only leaves contain (references to) tuple data (indicated here with *)

key value    pointer

| $p_0$ | $k_1$ | $p_1$ | $k_2$ | $p_2$ | $\cdots$ | $k_{2d}$ | $p_{2d}$ |
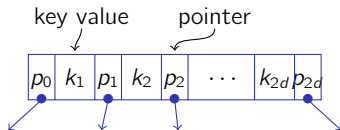
# Searching a B$^+$-tree

1 **Function:** search ($k$)

2 **return** tree_search ($k$, *root*);

---

1 **Function:**
  tree_search ($k$, *node*)

2 **if** *node* is a leaf **then**

3     **return** *node*;

4 **switch** $k$ **do**

5     **case** $k \leq k_0$ **do**

6        **return**
         tree_search ($k$, $p_0$);

7     **case** $k_i < k \leq k_{i+1}$ **do**

8        **return**
         tree_search ($k$, $p_i$);

9     **case** $k_{2d} < k$ **do**

10       **return**
         tree_search ($k$, $p_{2d}$);

- All searches now navigate to a leaf node.
  - $\rightarrow$ Makes search effort also more predictable.
- Function search ($k$) returns a pointer to the leaf node that contains potential hits for search key $k$.
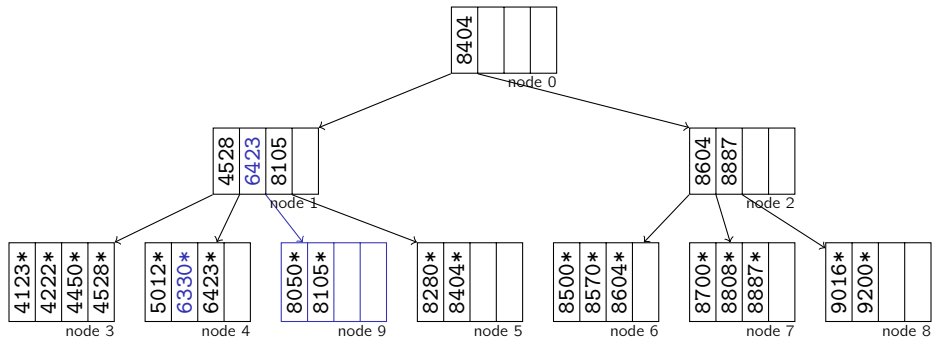
# B$^+$-tree Insert: Examples (Insert without Split)
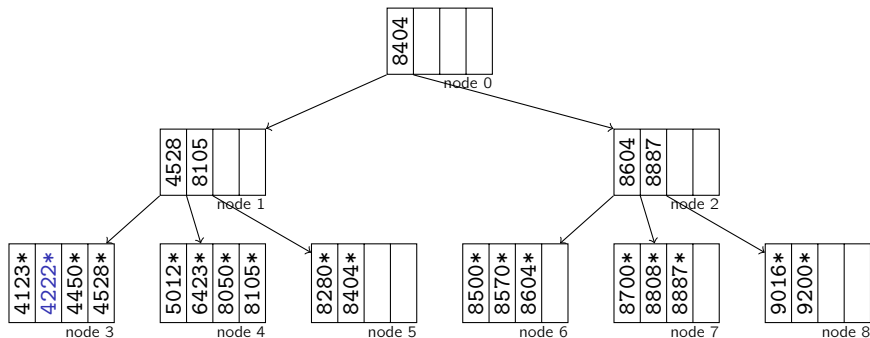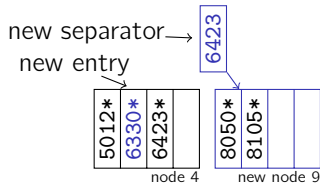


Insert new entry with key 4222.

→ Enough space in node 3, simply insert.
(Same as in B-tree)
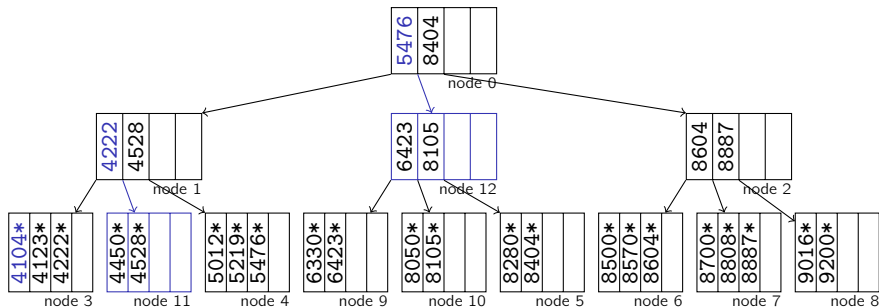
# B⁺-tree Insert: Examples (Insert with Leaf Split)



Insert key 6330.

→ Must **split** node 4.

→ **New separator** goes into node 1.
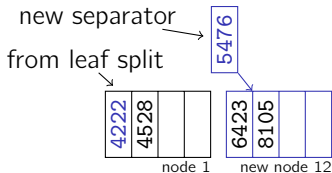But **keep** entry in node 4**!**

# B⁺-tree Insert: Examples (Insert with Inner Node Split)



After 5219, 5476, insert key 4104.

→ Must **split** leaf node 3.
→ Inner node 1 overflows → split it
→ **New separator** goes into root

Splitting the **inner node** works analogously to B-tree.

# B$^+$-tree Insertion Algorithm

1 **Function:** `tree_insert` ($k$, *rid*, *node*)

2 **if** *node* is a leaf **then**

3      **return** `leaf_insert` ($k$, *rid*, *node*);

4 **else**

5      **switch** $k$ **do**

6          **case** $k \leq k_0$ **do**

7             $\langle sep, ptr \rangle \leftarrow$ `tree_insert` ($k$, *rid*, $p_0$);

8          **case** $k_i < k \leq k_{i+1}$ **do**

9             $\langle sep, ptr \rangle \leftarrow$ `tree_insert` ($k$, *rid*, $p_i$);

10          **case** $k_{2d} < k$ **do**

11             $\langle sep, ptr \rangle \leftarrow$ `tree_insert` ($k$, *rid*, $p_{2d}$);

                                                       see `tree_search` ()

12      **if** *sep* is **null then**

13          **return** $\langle$**null**, **null**$\rangle$;

14      **else**

15          **return** `split` (*sep*, *ptr*, *node*);

```
1  Function: leaf_insert (k, rid, node)
2  if another entry fits into node then
3  │   insert ⟨k, rid⟩ into node ;
4  │   return ⟨null, null⟩;
5  else
6  │   allocate new leaf page p ;
7  │   take {⟨k₁⁺, p₁⁺⟩, ..., ⟨k₂d₊₁⁺, p₂d₊₁⁺⟩} := entries from node ∪ {⟨k, ptr⟩}
8  │   │   leave entries ⟨k₁⁺, p₁⁺⟩, ..., ⟨kd₊₁⁺, pd₊₁⁺⟩ in node ;
9  │   │   move entries ⟨kd₊₂⁺, pd₊₂⁺⟩, ..., ⟨k₂d₊₁⁺, p₂d₊₁⁺⟩ to p ;
10 │   return ⟨kd₊₁⁺, p⟩;
```

```
1  Function: split (k, ptr, node)
2  if another entry fits into node then
3  │   insert ⟨k, ptr⟩ into node ;
4  │   return ⟨null, null⟩;
5  else
6  │   allocate new leaf page p ;
7  │   take {⟨k₁⁺, p₁⁺⟩, ..., ⟨k₂d₊₁⁺, p₂d₊₁⁺⟩} := entries from node ∪ {⟨k, ptr⟩}
8  │   │   leave entries ⟨k₁⁺, p₁⁺⟩, ..., ⟨kd⁺, pd⁺⟩ in node ;
9  │   │   move entries ⟨kd₊₂⁺, pd₊₂⁺⟩, ..., ⟨k₂d₊₁⁺, p₂d₊₁⁺⟩ to p ;
10 │   │   set p₀ ← pd₊₁⁺ in node;
11 │   return ⟨kd₊₁⁺, p⟩;
```

# B$^+$-tree Insertion Algorithm

1 **Function:** `insert` ($k$, $rid$)
2 $\langle key, ptr \rangle \leftarrow$ `tree_insert` ($k$, $rid$, $root$);
3 **if** $key$ is not **null then**
4      allocate new root page $r$;
5      **populate** $n$ **with**
6          $p_0 \leftarrow root$;
7          $k_1 \leftarrow key$;
8          $p_1 \leftarrow ptr$;
9      $root \leftarrow r$ ;

- `insert` ($k$, $rid$) is called from outside.
- Note how leaf node entries point to RIDs, while inner nodes contain pointers to other B$^+$-tree nodes.

## Example

**Example:** Webserver access log (`people.inf.ethz.ch`)

- table cardinality: 11 million tuples (710K data pages)
- distinct IP addresses: 181,628 (stored as `CHAR(15)`)
- database: IBM DB2 9.7

### B$^+$-tree on IP addresses:

- 25,151 index pages total:
    - 1 root node
    - 110 second-level nodes; average fanout: 230
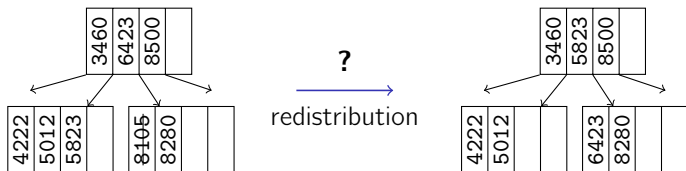    - 25,040 leaf-level nodes: 1–77 keys per node

## Deletion

- If a node is sufficiently full (*i.e.*, contains at least $d + 1$ entries, we may simply remove the entry from the node.
    - Note: Afterward, **inner nodes** may contain keys that no longer exist in the database. This is perfectly legal.

- **Merge** nodes in case of an **underflow** ("undo a split"):
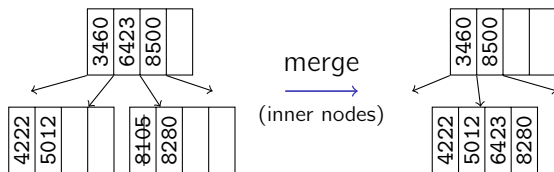


- "Pull" separator into merged node.

**⚠ It's not quite that easy...**



- Merging only works if **two** neighboring nodes were 50 % full.
- Otherwise, we have to **re-distribute**:
    - "rotate" entry through parent
- Redistribution is **complex** and **expensive**.
    - → Real systems usually do not implement deletion "by the book."

# 🎔 B$^+$-trees in Real Systems

- Actual systems often avoid the cost of merging and/or redistribution, but relax the minimum occupancy rule.

- *E.g.*, **IBM DB2 UDB**:
    - The MINPCTUSED parameter controls when the system should try a leaf node merge ("on-line index reorg").
    - Inner nodes are never merged
      ($\rightarrow$ need to do full table reorg for that).

- To improve **concurrency**, systems sometimes only **mark** index entries as deleted and physically remove them later (*e.g.*, IBM DB2 UDB "type-2 indexes")
    - $\rightarrow$ Resulting index entries are also called **ghost records**.

# B$^+$-trees and Sorting

A typical situation (for a secondary B$^+$-tree) looks like this:
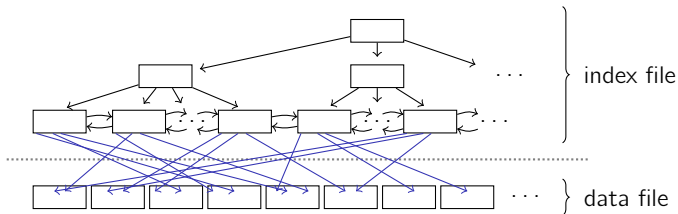


✎ **What are the implications when we want to execute**
   `SELECT * FROM CUSTOMERS ORDER BY ZIPCODE    ?`

# Composite Keys

$B^+$-trees can (in theory[15]) be used to index everything with a defined **total order**, *e.g.*:

- integers, strings, dates, . . . , and
- **concatenations** thereof (based on **lexicographical order**).

*E.g.*, in most SQL dialects:

```
CREATE INDEX ON TABLE CUSTOMERS (LASTNAME, FIRSTNAME);
```

A useful application are, *e.g.*, **partitioned B-trees**:

- Leading index attributes effectively **partition** the resulting $B^+$-tree.

↗ G. Graefe: Sorting And Indexing With Partitioned B-Trees. *CIDR 2003.*

---

[15]Some implementations won't allow you to index, *e.g.*, large character fields.

# Partitioned B-trees

```
CREATE INDEX ON TABLE STUDENTS(SEMESTER, ZIPCODE);
```

✎ **What types of queries could this index support?**