

# Chapter 4

## Tree-Structured Indexing

ISAM and B<sup>+</sup>-trees

*Architecture and Implementation of Database Systems*  
Winter 2010/11

### Binary Search

#### ISAM

- Multi-Level ISAM
- Too Static?
- Search Efficiency

#### B<sup>+</sup>-trees

- Search
- Insert
- Redistribution
- Delete
- Duplicates
- Key Compression
- Bulk Loading
- Partitioned B<sup>+</sup>-trees



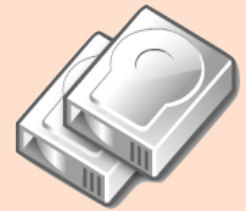
# Ordered Files and Binary Search

How could we prepare for such queries and evaluate them efficiently?

```
1 SELECT *  
2 FROM CUSTOMERS  
3 WHERE ZIPCODE BETWEEN 8880 AND 8999
```

We could

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



## Binary Search

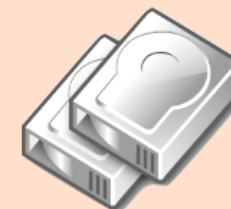
### ISAM

- Multi-Level ISAM
- Too Static?
- Search Efficiency

### B<sup>+</sup>-trees

- Search
- Insert
- Redistribution
- Delete
- Duplicates
- Key Compression
- Bulk Loading
- Partitioned B<sup>+</sup>-trees

# Ordered Files and Binary Search



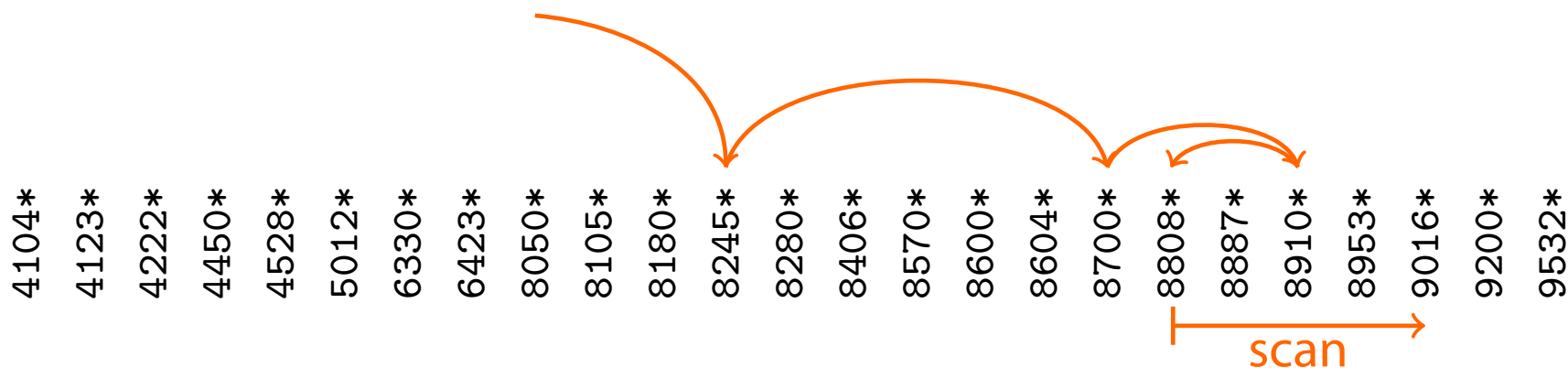
How could we prepare for such queries and evaluate them efficiently?

```
1 SELECT *
2 FROM CUSTOMERS
3 WHERE ZIPCODE BETWEEN 8880 AND 8999
```

We could

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

Here, let  $k^*$  denote the full record with key  $k$ :



## Binary Search

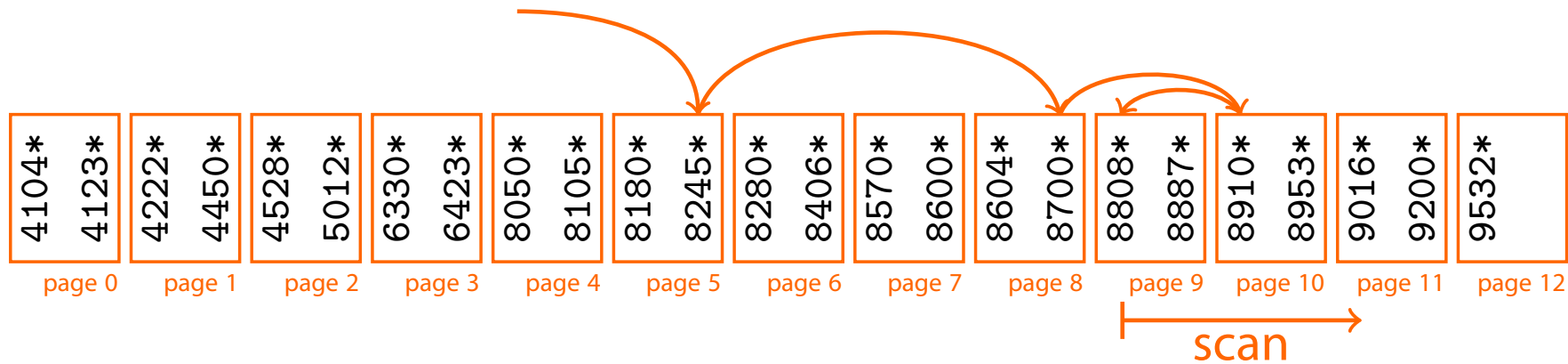
### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

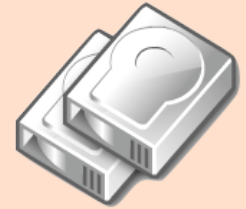
Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

# Ordered Files and Binary Search



✓ We get **sequential access** during the **scan phase**.

We need to read  $\log_2(\# \text{ tuples})$  tuples during the **search phase**.



## Binary Search

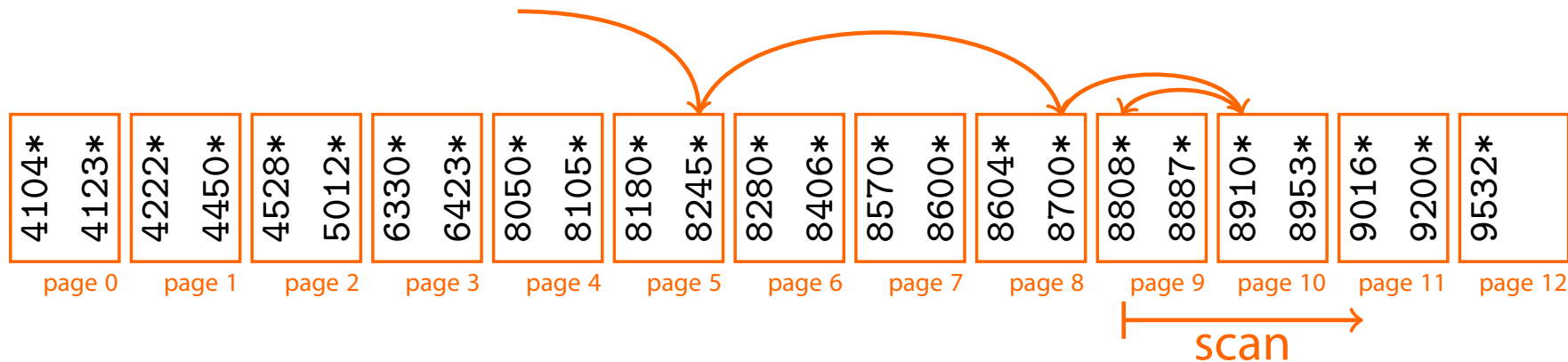
### ISAM

- Multi-Level ISAM
- Too Static?
- Search Efficiency

### B<sup>+</sup>-trees

- Search
- Insert
- Redistribution
- Delete
- Duplicates
- Key Compression
- Bulk Loading
- Partitioned B<sup>+</sup>-trees

# Ordered Files and Binary Search

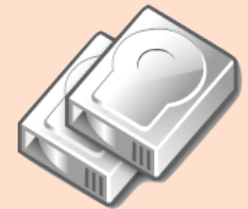


✓ We get **sequential access** during the **scan phase**.

We need to read  $\log_2(\# \text{ tuples})$  tuples during the **search phase**.

✗ We 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 page prefetching.



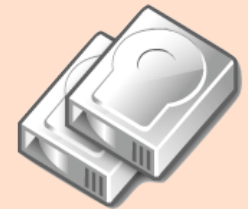
## Binary Search

### ISAM

- Multi-Level ISAM
- Too Static?
- Search Efficiency

### B<sup>+</sup>-trees

- Search
- Insert
- Redistribution
- Delete
- Duplicates
- Key Compression
- Bulk Loading
- Partitioned B<sup>+</sup>-trees



## Binary Search

### ISAM

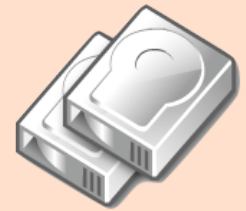
Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

- This chapter discusses two **index structures** which especially shine if we need to support **range selections** (and thus sorted file scans): **ISAM** files and **B<sup>+</sup>-trees**.
- Both indexes are based on the same simple idea which naturally leads to a **tree-structured** organization of the indexes. (Hash indexes are covered in the following chapter.)
- B<sup>+</sup>-trees refine the idea underlying the rather static ISAM scheme and add efficient support for **insertions** and **deletions**.

# Indexed Sequential Access Method (ISAM)



## Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

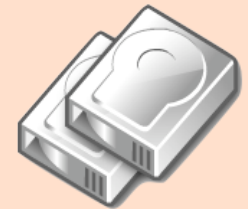
Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

**Remember:** range selections on ordered files may use **binary search** to locate the lower range limit as a starting point for a sequential scan of the file (until the upper limit is reached).

## ISAM ...

- ...acts as a replacement for the binary search phase, and
- touches considerably fewer pages than binary search.

# Indexed Sequential Access Method (ISAM)

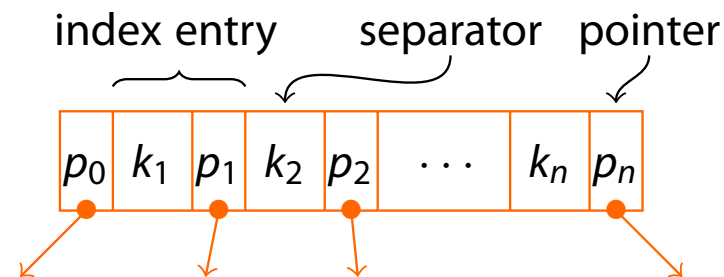


Multi-Level ISAM  
Too Static?  
Search Efficiency

Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

To support range selections on field A:

- 1 In addition to the A-sorted data file, maintain an **index file** with entries (records) of the following form:



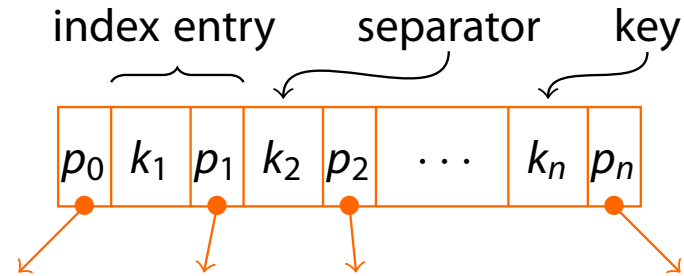
- 2 ISAM leads to **sparse** index structures. In an index entry

$$\langle k_i, \text{pointer to } p_i \rangle ,$$

key  $k_i$  is the first (i.e., the minimal) A value on the data file page pointed to by  $p_i$  ( $p_i$ : page no).

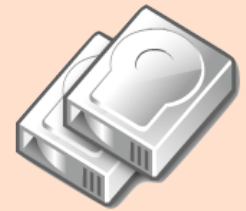
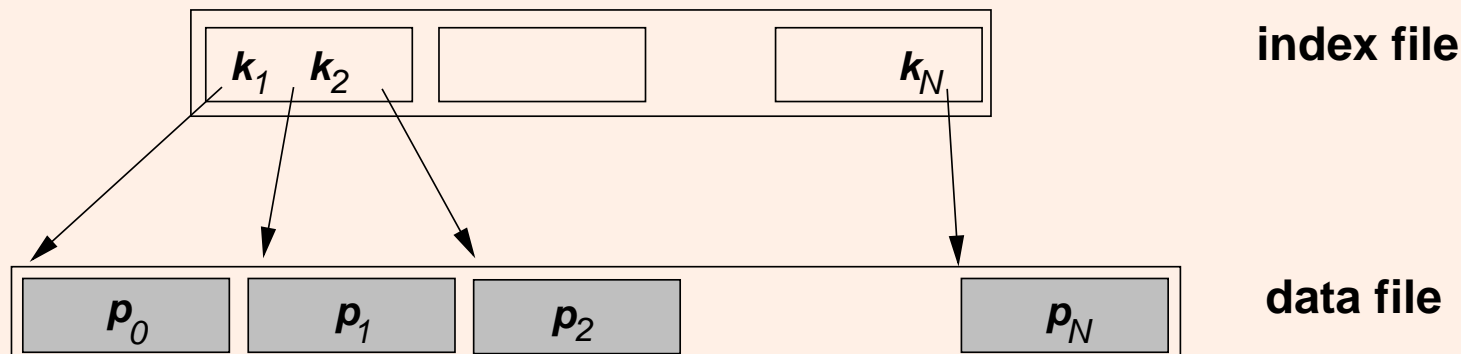


# Indexed Sequential Access Method (ISAM)



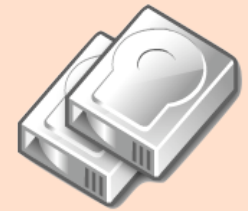
- In the index file, the  $k_i$  serve as **separators** between the contents of pages  $p_{i-1}$  and  $p_i$ .
- It is guaranteed that  $k_{i-1} < k_i$  for  $i = 2, \dots, n$ .
- We obtain a **one-level ISAM structure**.

## One-level ISAM structure of $N + 1$ pages



Multi-Level ISAM  
Too Static?  
Search Efficiency

Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees



## Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

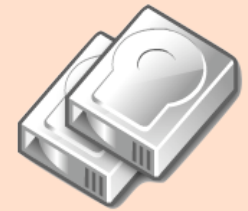
## SQL range selection on field A

```
1 SELECT *  
2 FROM R  
3 WHERE A BETWEEN lower AND upper
```

To support **range selections**:

- 1 Conduct a **binary search on the index file** for a key of value *lower*.
- 2 Start a **sequential scan of the data file** from the page pointed to by the index entry (scan until field A exceeds *upper*).

# Indexed Sequential Access Method ISAM



## Binary Search

### ISAM

#### Multi-Level ISAM

Too Static?

Search Efficiency

### B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

- The size of the index file is likely to be **much smaller** than the data file size. Searching the index is far more efficient than searching the data file.
- For large data files, however, even the index file might be too large to allow for fast searches.

## Main idea behind ISAM indexing

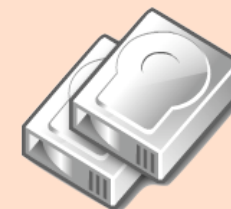
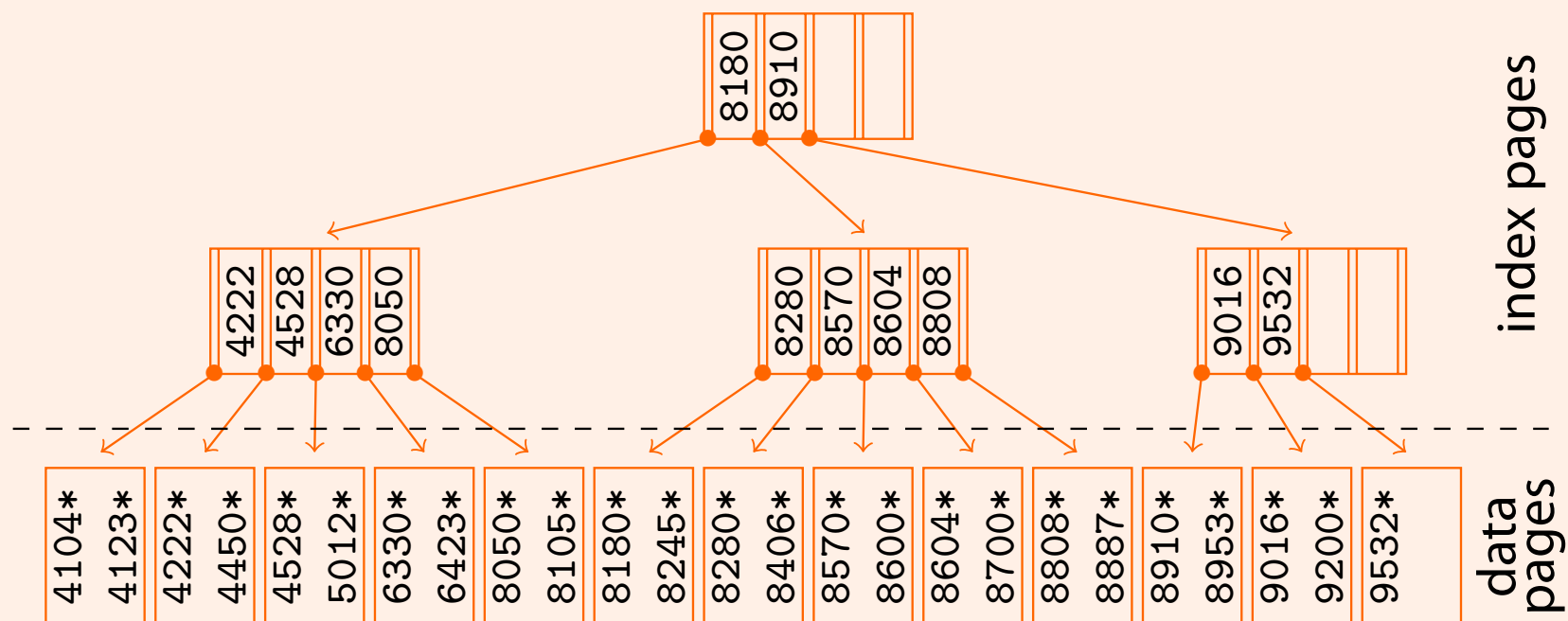
**Recursively** apply the index creation step: treat the topmost index level like the data file and add an additional index layer on top.

Repeat, until the the top-most index layer fits on a single page (the **root page**).

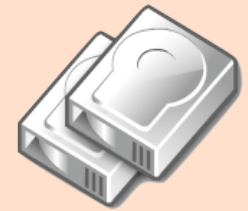
# Multi-Level ISAM Structure

This recursive index creation scheme leads to a **tree-structured** hierarchy of index levels:

## Multi-level ISAM structure



# Multi-Level ISAM Structure



## Binary Search

## ISAM

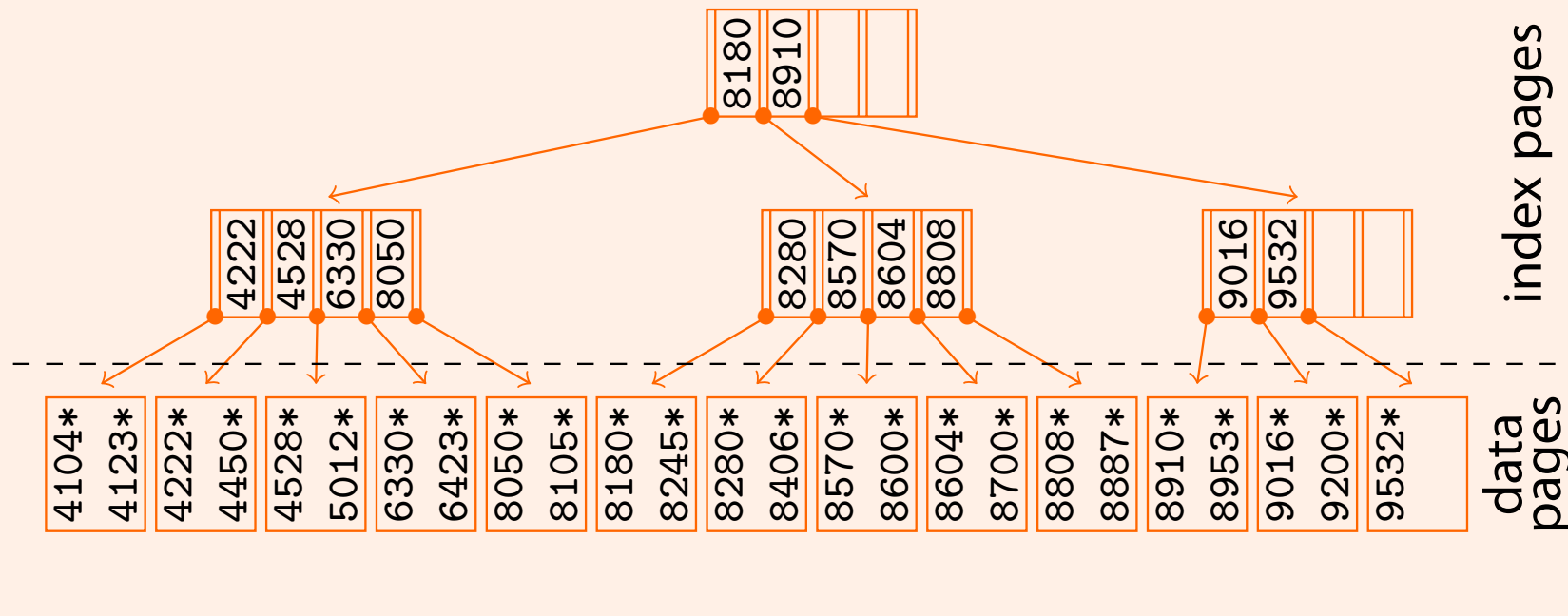
### Multi-Level ISAM

Too Static?  
Search Efficiency

## B<sup>+</sup>-trees

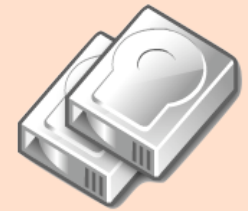
Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

## Multi-level ISAM structure



- Each ISAM tree node corresponds to one page (disk block).
- To create the ISAM structure for a given data file, proceed **bottom-up**:
  - 1 Sort the data file on the search key field.
  - 2 Create the index leaf level.
  - 3 If the top-most index level contains more than one page, repeat.

# Multi-Level ISAM Structure: Overflow Pages



## Binary Search

## ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

## B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

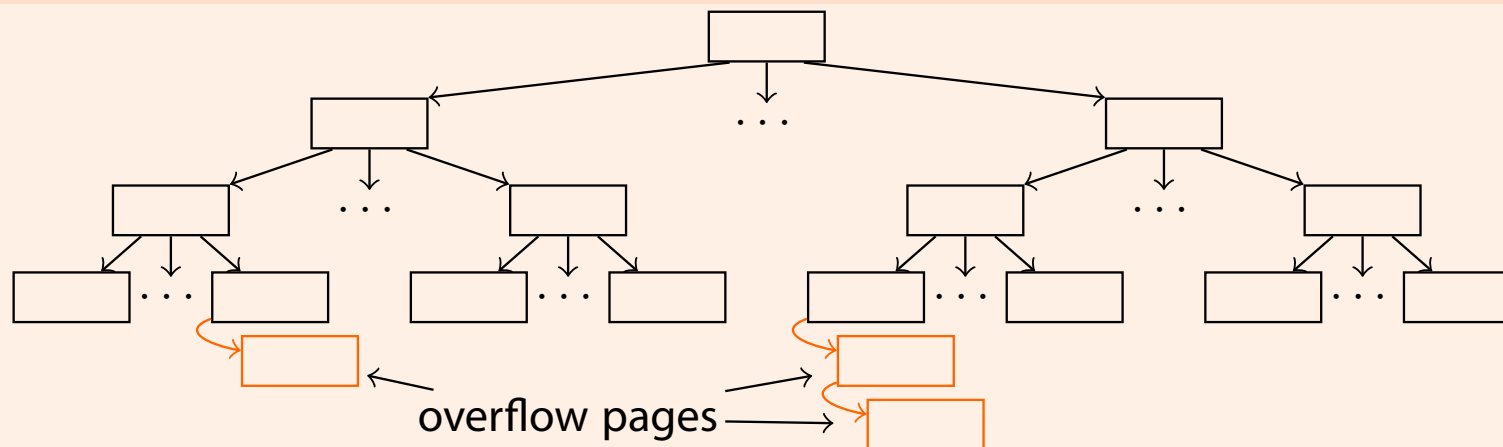
Key Compression

Bulk Loading

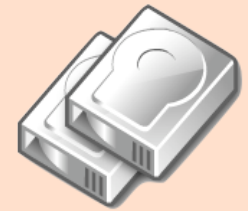
Partitioned B<sup>+</sup>-trees

- The upper index levels of the ISAM tree remain **static**: insertions and deletions in the data file do *not* affect the upper tree layers.
  - Insertion of record into data file: if space is left on the associated leaf page, insert record there.
  - Otherwise create and maintain a chain of **overflow pages** hanging off the full primary leaf page. **Note**: the records on the overflow pages are **not ordered** in general.
- ⇒ Over time, **search performance in ISAM can degrade**.

## Multi-level ISAM structure with overflow pages

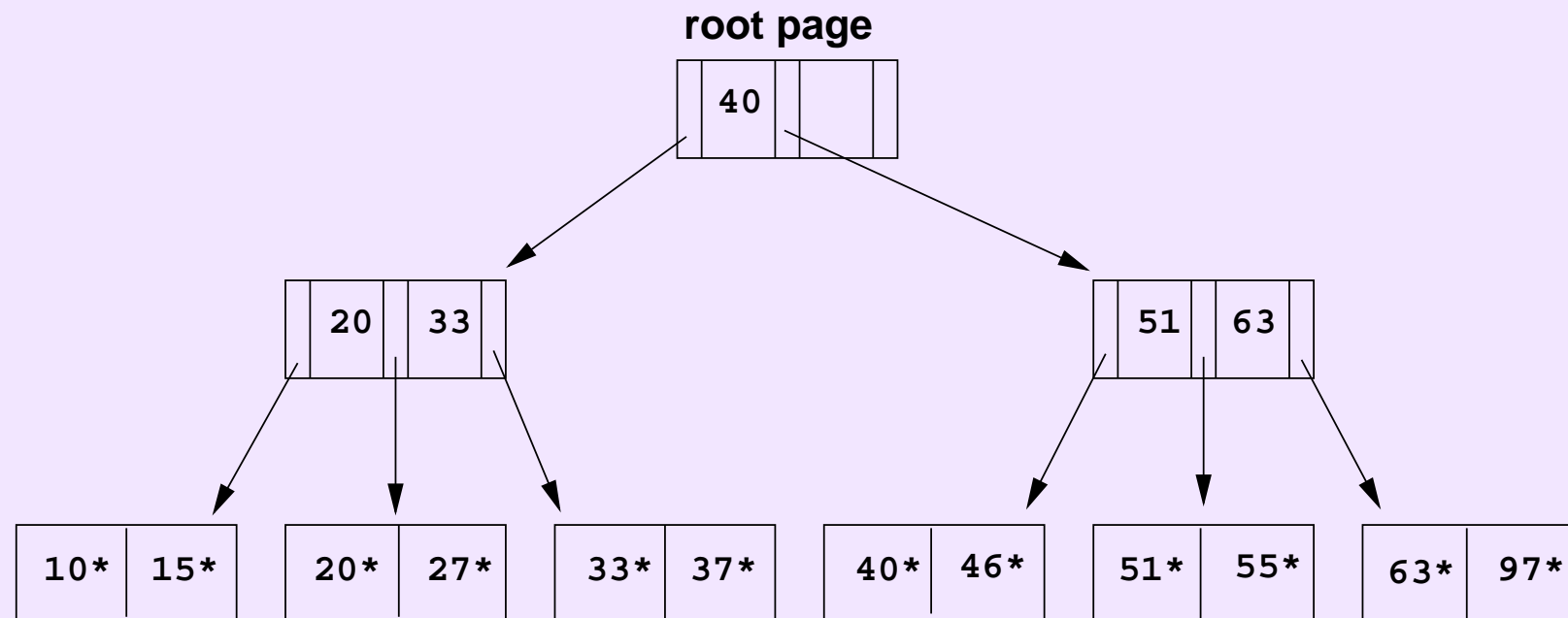


# Multi-Level ISAM Structure: Example

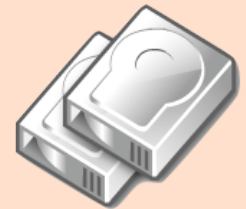


Each page can hold two index entries plus one (the left-most) page pointer:

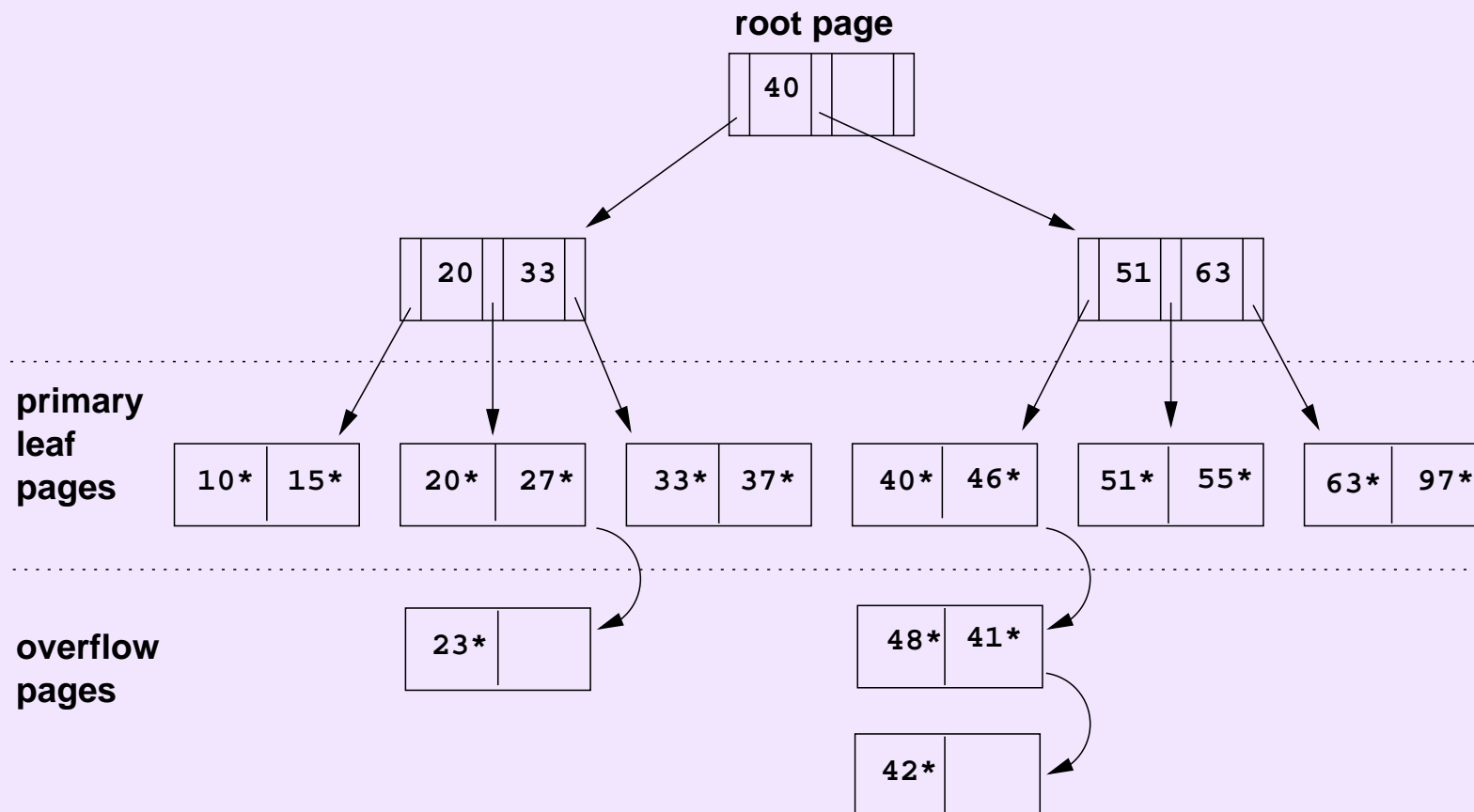
## Example (Initial state of ISAM structure)



# Multi-Level ISAM Structure: Insertions



Example (After insertion of data records with keys 23, 48, 41, 42)



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

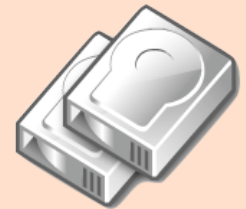
Key Compression

Bulk Loading

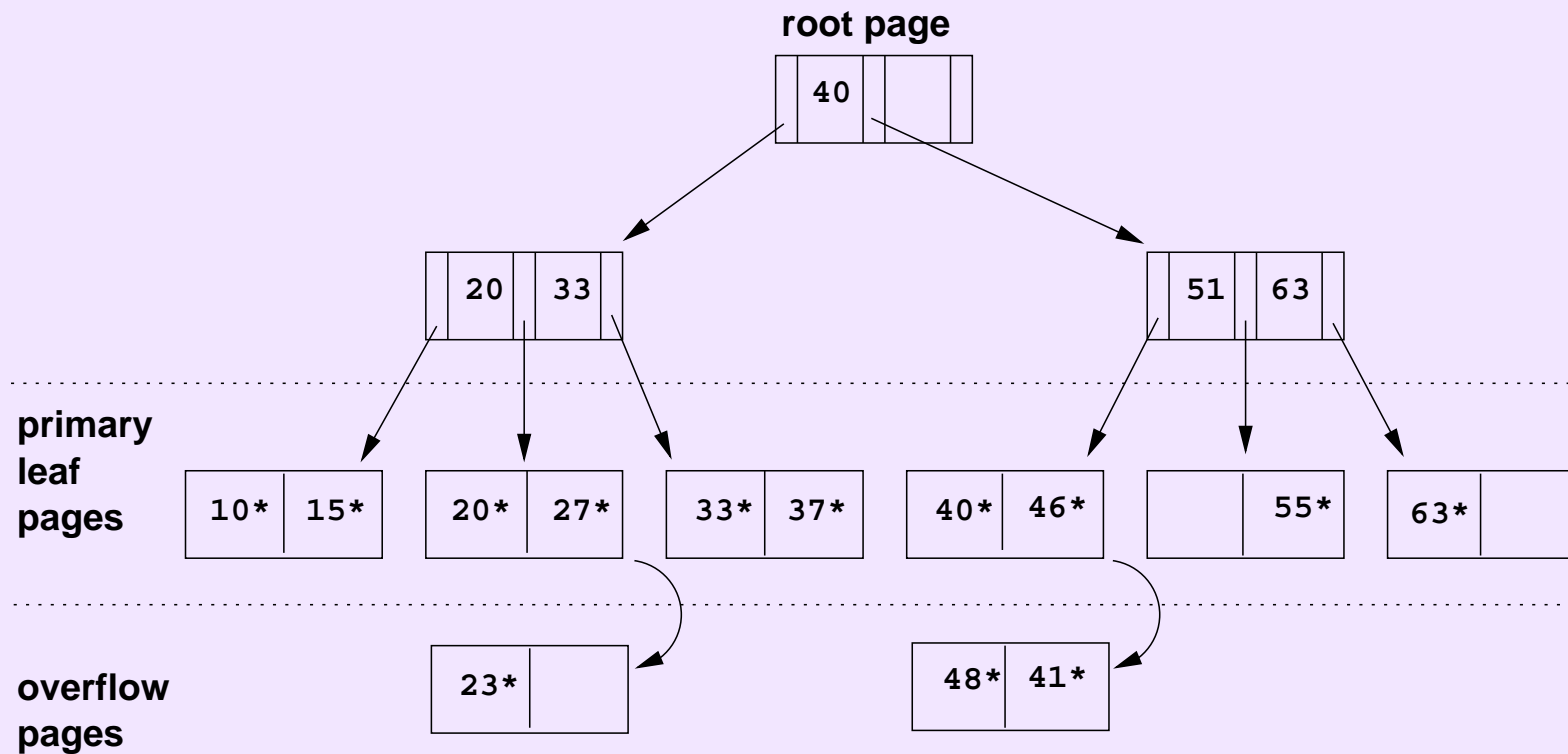
Partitioned B<sup>+</sup>-trees



# Multi-Level ISAM Structure: Deletions



## Example (After deletion of data records with keys 42, 51, 97)



### Binary Search

### ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

### B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

# ISAM: Too Static?

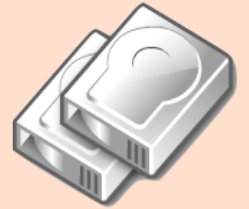
- The non-leaf levels of the ISAM structure have not been touched at all by the data file updates.
- This may lead to index key entries which do not appear in the index leaf level (e.g., key value 51 on the previous slide).

## Orphaned index entries

Does an index key entry like 51 above lead to problems during index key searches?

Tree-Structured  
Indexing

Torsten Grust



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

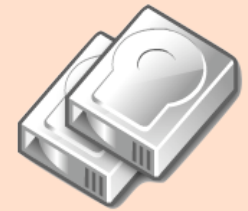
Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

# ISAM: Too Static?



- The non-leaf levels of the ISAM structure have not been touched at all by the data file updates.
- This may lead to index key entries which do not appear in the index leaf level (e.g., key value 51 on the previous slide).

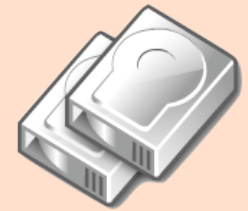
## Orphaned index entries

Does an index key entry like 51 above lead to problems during index key searches?

No, since the index keys maintain their separator property.

- To preserve the **separator property** of the index key entries, maintenance of overflow chains is required.
- ⇒ ISAM may **lose balance** after heavy updating. This complicates life for the query optimizer.

# ISAM: Being Static is Not All Bad



## Binary Search

### ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

### B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

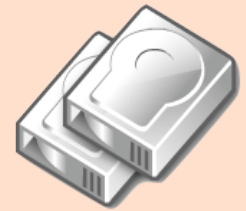
Partitioned B<sup>+</sup>-trees

- Leaving free space during index creation reduces the insertion/overflow problem (typically  $\approx 20\%$  free space).
- Since ISAM indexes are static, **pages need not be locked** during concurrent index access.
  - Locking can be a serious bottleneck in dynamic tree indexes (particularly near the index root node).



⇒ ISAM may be the index of choice for relatively static data.

# ISAM: Efficiency of Searches



## Binary Search

### ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

### B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

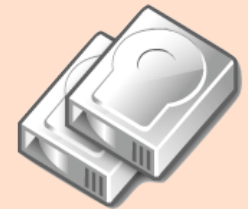
- Regardless of these deficiencies, ISAM-based searching is the most efficient **order-aware** index structure discussed so far:

## Definition (ISAM fanout)

- Let  $N$  be the number of pages in the data file, and let  $F$  denote the **fanout** of the ISAM tree, *i.e.*, the maximum number of children per index node
- The fanout in the previous example is  $F = 3$ , typical realistic fanouts are  $F \approx 1,000$ .
- When index searching starts, the search space is of size  $N$ . With the help of the root page we are guided into an index subtree of size

$$N \cdot 1/F .$$

# ISAM: Efficiency of Searches



## Binary Search

### ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

### B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

- As we search down the tree, the search space is repeatedly reduced by a factor of  $F$ :

$$N \cdot 1/F \cdot 1/F \dots$$

- Index searching ends after  $s$  steps when the search space has been reduced to size 1 (*i.e.*, we have reached the index leaf level and found the data page that contains the wanted record):

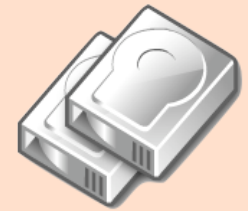
$$N \cdot (1/F)^s \stackrel{!}{=} 1 \quad \Leftrightarrow \quad s = \log_F N .$$

- Since  $F \gg 2$ , this is significantly more efficient than access via binary search ( $\log_2 N$ ).

## Example (Required I/O operations during ISAM search)

Assume  $F = 1,000$ . An ISAM tree of **height 3** can index a file of one billion ( $10^9$ ) pages, *i.e.*, 3 I/O operations are sufficient to locate the wanted data file page.

# B<sup>+</sup>-trees: A Dynamic Index Structure



## Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

## B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

The **B<sup>+</sup>-tree index** structure is derived from the ISAM idea, but is fully dynamic with respect to updates:

- Search performance is only dependant on the **height** of the B<sup>+</sup>-tree (because of high fan-out  $F$ , the height rarely exceeds 3).
- **No overflow chains** develop, a B<sup>+</sup>-tree remains **balanced**.
- B<sup>+</sup>-trees offer efficient **insert/delete procedures**, the underlying data file can grow/shrink **dynamically**.
- B<sup>+</sup>-tree nodes (despite the root page) are **guaranteed to have a minimum occupancy of 50 %** (typically  $2/3$ ).

↗ Original publication (B-tree): R. Bayer and E.M. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, vol. 1, no. 3, September 1972.

# B<sup>+</sup>-trees: Basics

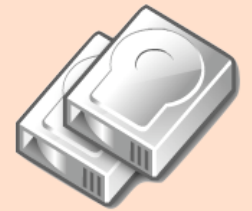
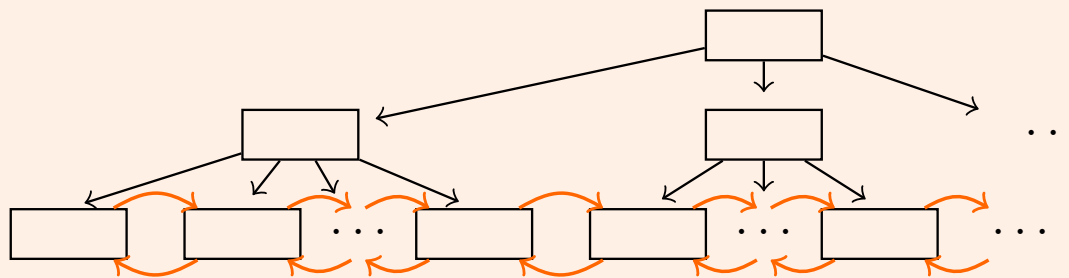
B<sup>+</sup>-trees resemble ISAM indexes, where

- leaf nodes are connected to form a **doubly-linked list**, the so-called **sequence set**,<sup>1</sup>
- leaves may contain **actual data records** or just **references to records** on data pages (i.e., index entry variants **B** or **C**).

*Here we assume the latter since this is the common case.*

Remember: ISAM leaves were the **data pages** themselves, instead.

## Sketch of B<sup>+</sup>-tree structure (data pages not shown)

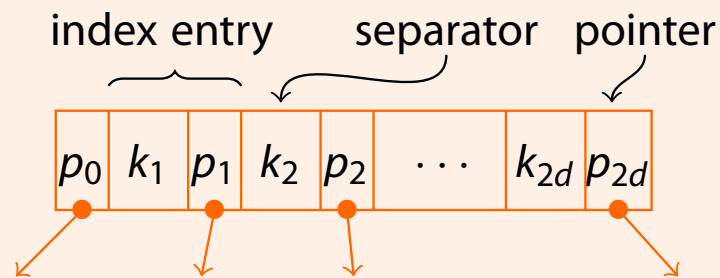


<sup>1</sup>This is not a strict B<sup>+</sup>-tree requirement, although most systems implement it.



# B<sup>+</sup>-trees: Non-Leaf Nodes

## B<sup>+</sup>-tree inner (non-leaf) node



B<sup>+</sup>-tree **non-leaf nodes** use the same internal layout as inner ISAM nodes:

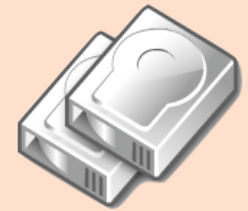
- The **minimum** and **maximum number of entries**  $n$  is bounded by the **order**  $d$  of the B<sup>+</sup>-tree:

$$d \leq n \leq 2 \cdot d \quad (\text{root node: } 1 \leq n \leq 2 \cdot d) .$$

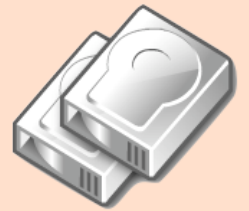
- A node contains  $n + 1$  pointers. Pointer  $p_i$  ( $1 \leq i \leq n - 1$ ) points to a subtree in which all key values  $k$  are such that

$$k_i \leq k < k_{i+1} .$$

( $p_0$  points to a subtree with key values  $< k_1$ ,  $p_n$  points to a subtree with key values  $\geq k_n$ ).



# B<sup>+</sup>-tree: Leaf Nodes



## Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

## B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

- B<sup>+</sup>-tree leaf nodes contain pointers to data **records** (not **pages**). A leaf node entry with key value  $k$  is denoted as  $k_*$  as before.
- Note that we can use all index entry variants **A**, **B**, **C** to implement the leaf entries:
  - For variant **A**, the B<sup>+</sup>-tree represents the index as well as the data file itself. Leaf node entries thus look like

$$k_j^* = \langle k_j, \langle \dots \rangle \rangle .$$

- For variants **B** and **C**, the B<sup>+</sup>-tree lives in a file distinct from the actual data file. Leaf node entries look like

$$k_j^* = \langle k_j, rid \rangle .$$

# B<sup>+</sup>-tree: Search

Below, we assume that key values are **unique** (we defer the treatment of **duplicate key values**).

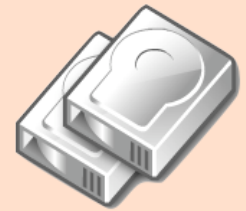
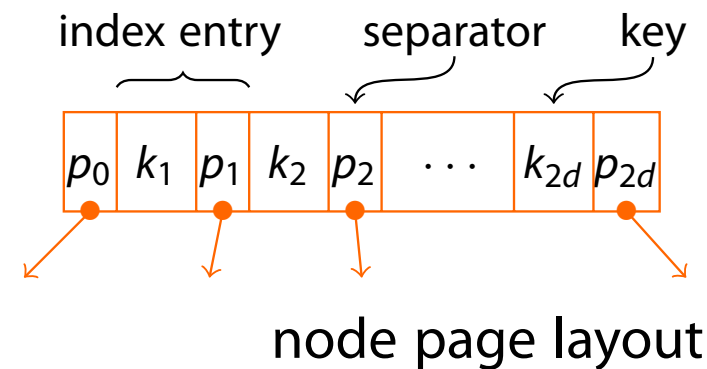
## B<sup>+</sup>-tree search

```
1 Function: search(k)
2 return tree_search(k, root);
```

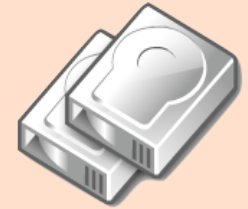
---

```
1 Function: tree_search(k, node)
2 if node is a leaf then
3   return node;
4 switch k do
5   case  $k < k_1$ 
6     return tree_search(k,  $p_0$ );
7   case  $k_i \leq k < k_{i+1}$ 
8     return tree_search(k,  $p_i$ );
9   case  $k_{2d} \leq k$ 
10    return tree_search(k,  $p_{2d}$ );
```

- Function search(*k*) returns a pointer to the leaf node page that contains potential hits for search key *k*.



# B<sup>+</sup>-tree: Insert



## Binary Search

### ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

### B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

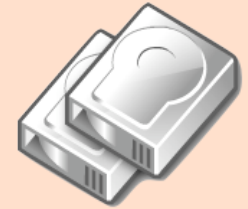
Partitioned B<sup>+</sup>-trees

- Remember that B<sup>+</sup>-trees remain **balanced**<sup>2</sup> no matter which updates we perform. Insertions and deletions have to preserve this invariant.

---

<sup>2</sup>All paths from the B<sup>+</sup>-tree root to any leaf are of equal length.

# B<sup>+</sup>-tree: Insert



## Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

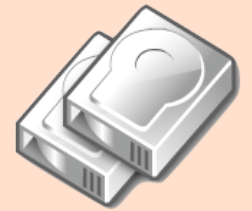
Partitioned B<sup>+</sup>-trees

- Remember that B<sup>+</sup>-trees remain **balanced**<sup>2</sup> no matter which updates we perform. Insertions and deletions have to preserve this invariant.
- The basic principle of B<sup>+</sup>-tree insertion is simple:
  - ① To insert a record with key  $k$ , call  $\text{search}(k)$  to find the page  $p$  to hold the new record.  
Let  $m$  denote the number of entries on  $p$ .
  - ② If  $m < 2 \cdot d$  (i.e., there is capacity left on  $p$ ), store  $k^*$  in page  $p$ . **Otherwise ...?**

---

<sup>2</sup>All paths from the B<sup>+</sup>-tree root to any leaf are of equal length.

# B<sup>+</sup>-tree: Insert



## Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

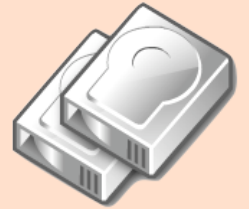
Partitioned B<sup>+</sup>-trees

- Remember that B<sup>+</sup>-trees remain **balanced**<sup>2</sup> no matter which updates we perform. Insertions and deletions have to preserve this invariant.
- The basic principle of B<sup>+</sup>-tree insertion is simple:
  - ① To insert a record with key  $k$ , call `search(k)` to find the page  $p$  to hold the new record.  
Let  $m$  denote the number of entries on  $p$ .
  - ② If  $m < 2 \cdot d$  (i.e., there is capacity left on  $p$ ), store  $k^*$  in page  $p$ . **Otherwise ...?**
- We must *not* start an overflow chain hanging off  $p$ : this would violate the balancing property.
- We want the cost for `search(k)` to be dependant on tree height only, so placing  $k^*$  somewhere else (even near  $p$ ) is *no* option either.

---

<sup>2</sup>All paths from the B<sup>+</sup>-tree root to any leaf are of equal length.

# B<sup>+</sup>-tree: Insert



## Binary Search

### ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

### B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

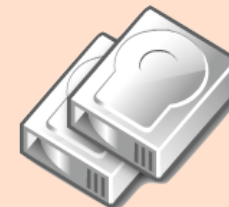
Bulk Loading

Partitioned B<sup>+</sup>-trees

- Sketch of the insertion procedure for entry  $\langle k, q \rangle$  (key value  $k$  pointing to *rid*  $q$ ):
  - 1 **Find leaf page**  $p$  where we would expect the entry for  $k$ .
  - 2 If  $p$  has **enough space** to hold the new entry (*i.e.*, at most  $2d - 1$  entries in  $p$ ), **simply insert**  $\langle k, q \rangle$  into  $p$ .
  - 3 Otherwise node  $p$  must be **split** into  $p$  and  $p'$  and a new **separator** has to be inserted into the parent of  $p$ .

Splitting happens recursively and may eventually lead to a split of the root node (increasing the tree height).
  - 4 Distribute the entries of  $p$  and the new entry  $\langle k, q \rangle$  onto pages  $p$  and  $p'$ .

# B<sup>+</sup>-tree: Insert



## Binary Search

## ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

## B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

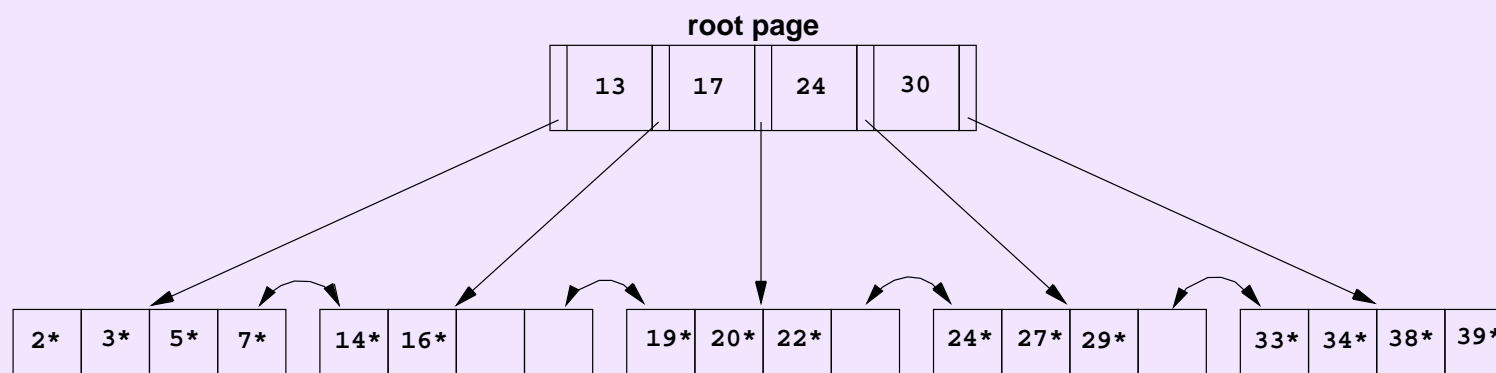
Key Compression

Bulk Loading

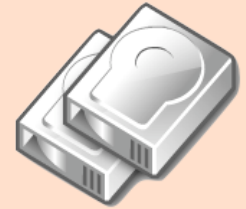
Partitioned B<sup>+</sup>-trees

## Example (B<sup>+</sup>-tree insertion procedure)

① Insert record with key  $k = 8$  into the following B<sup>+</sup>-tree:

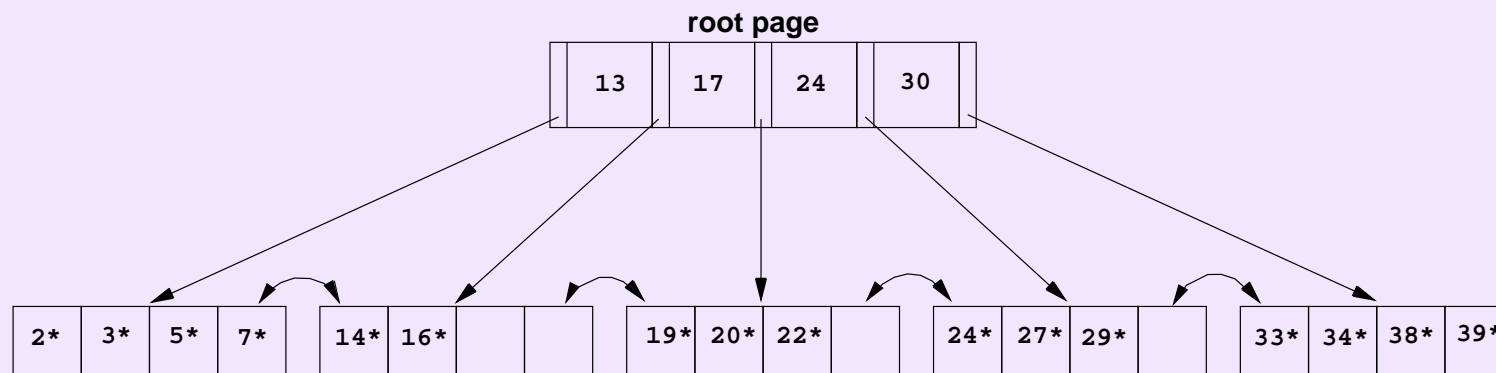






## Example (B<sup>+</sup>-tree insertion procedure)

- 1 Insert record with key  $k = 8$  into the following B<sup>+</sup>-tree:



- 2 The left-most leaf page  $p$  has to be split. Entries 2\*, 3\* remain on  $p$ , entries 5\*, 7\*, and 8\* (new) go on new page  $p'$ .

### Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

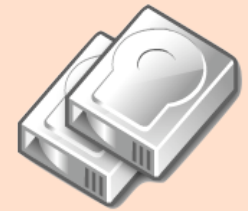
Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Insert and Leaf Node Split



## Binary Search

## ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

## B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

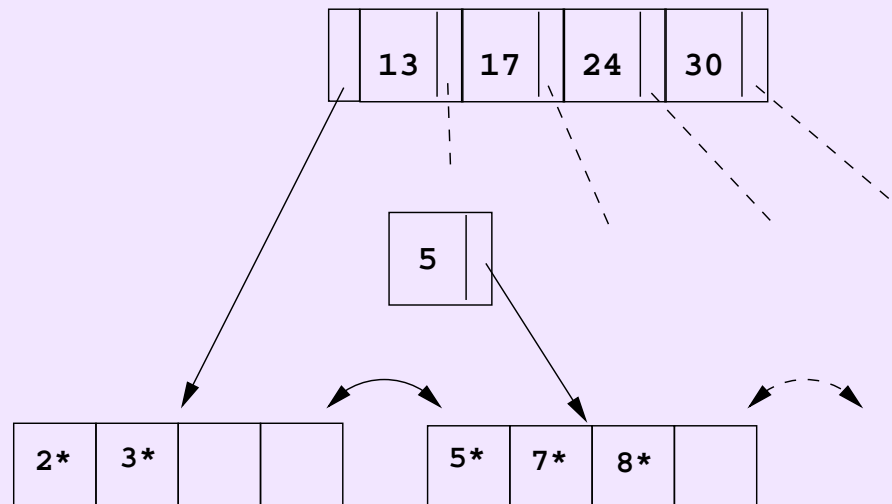
Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

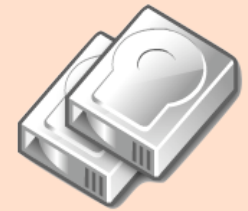
## Example (B<sup>+</sup>-tree insertion procedure)

- 3 Pages  $p$  and  $p'$  are shown below.  
Key  $k' = 5$ , the **new separator** between pages  $p$  and  $p'$ , has to be **inserted into the parent** of  $p$  and  $p'$  **recursively**:



- Note that, after such a **leaf node split**, the new separator key  $k' = 5$  is **copied up** the tree: the entry 5\* itself has to remain in its leaf page.

# B<sup>+</sup>-tree: Insert and Non-Leaf Node Split



## Binary Search

## ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

## B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

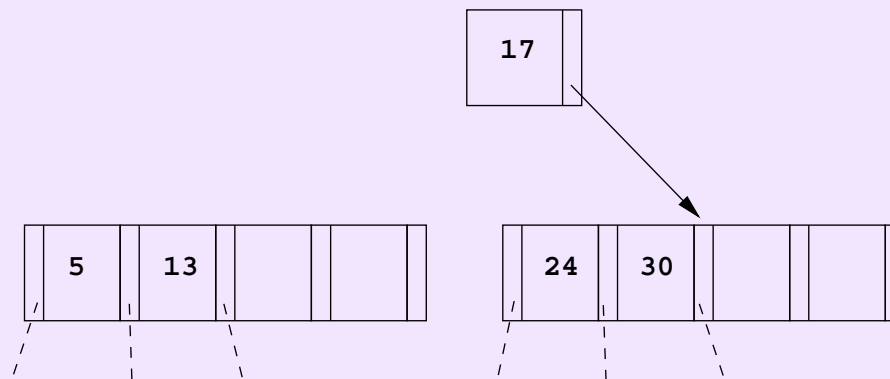
Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

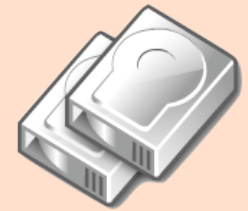
## Example (B<sup>+</sup>-tree insertion procedure)

- The insertion process is propagated upwards the tree: inserting key  $k' = 5$  into the parent leads to a **non-leaf node split** (the  $2 \cdot d + 1$  keys and  $2 \cdot d + 2$  pointers make for two new non-leaf nodes and a **middle key** which we propagate further up for insertion):



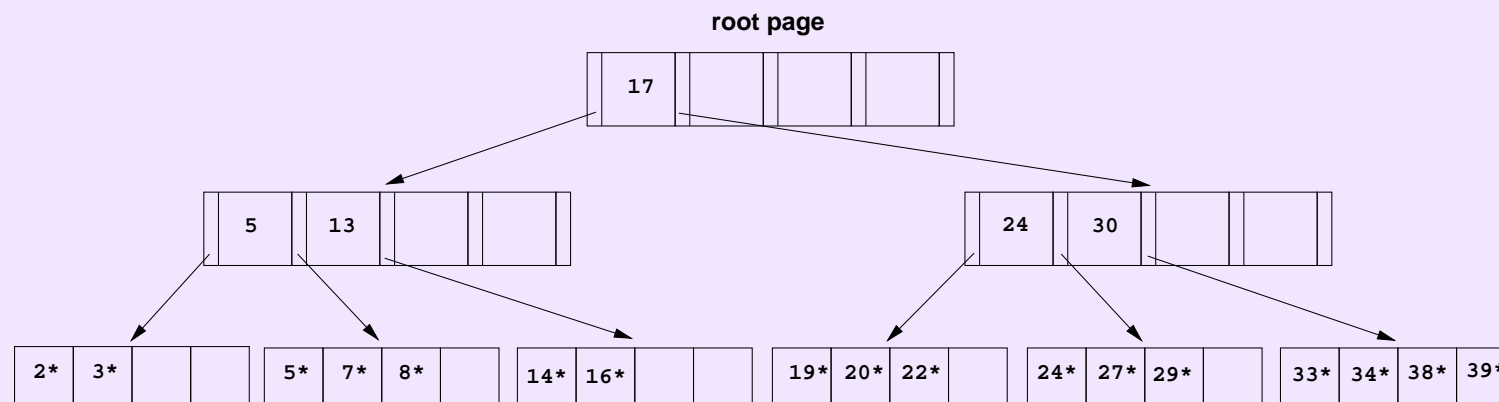
- Note that, for a **non-leaf node split**, we can simply **push up** the middle key (17). Contrast this with a leaf node split.

# B<sup>+</sup>-tree: Insert and Root Node Split



## Example

- Since the split node was the root node, we create a **new root node** which holds the pushed up middle key only:



- Splitting the old root and creating a new root node is the *only* situation in which the **B<sup>+</sup>-tree height increases**. The B<sup>+</sup>-tree thus remains balanced.

### Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

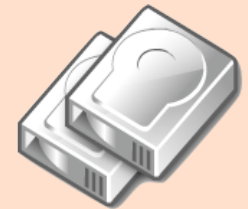
Duplicates

Key Compression

Bulk Loading

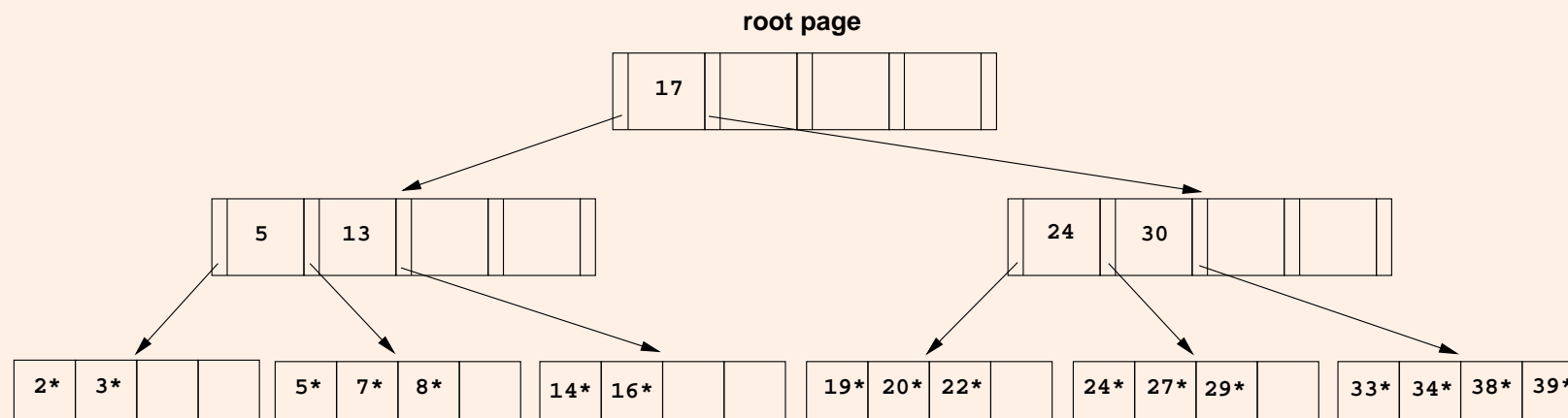
Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Insert



## Further key insertions

How does the insertion of records with keys  $k = 23$  and  $k = 40$  alter the B<sup>+</sup>-tree?



### Binary Search

### ISAM

- Multi-Level ISAM
- Too Static?
- Search Efficiency

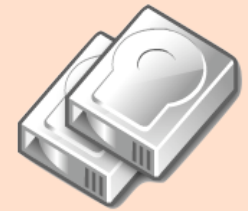
### B<sup>+</sup>-trees

Search

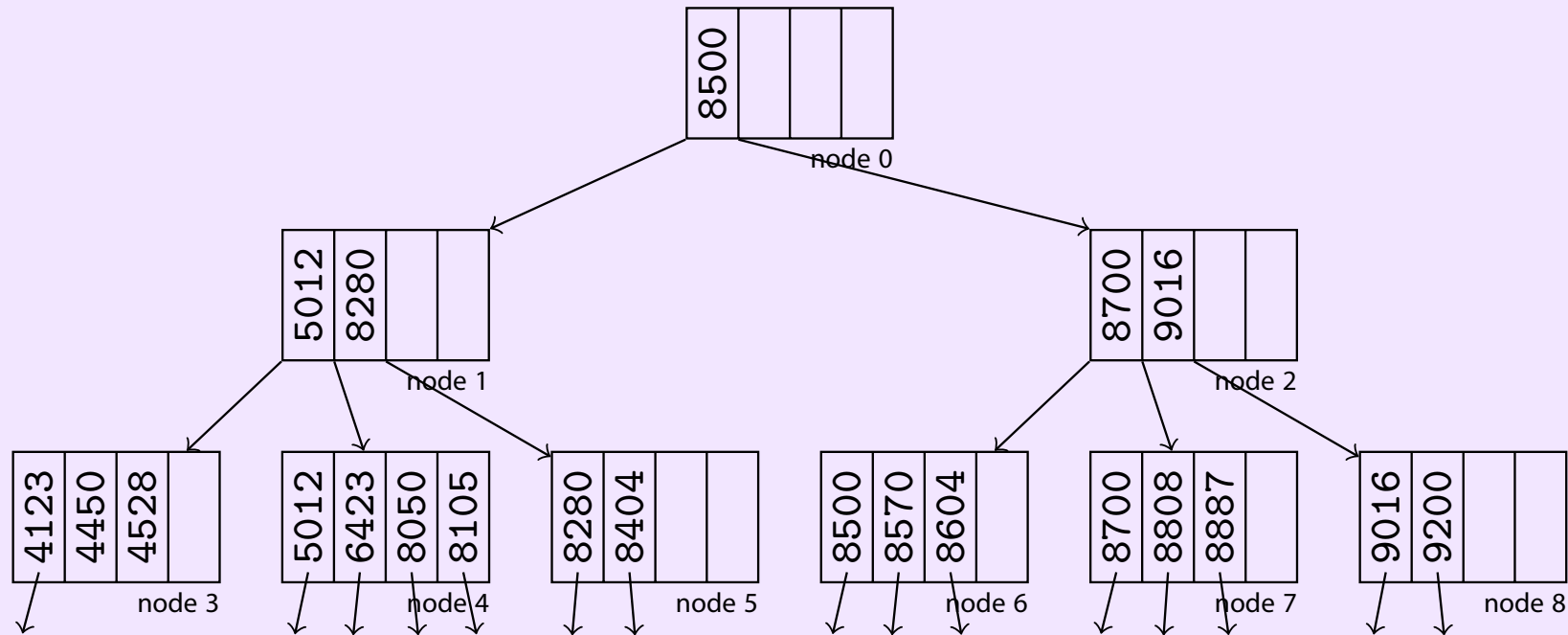
Insert

- Redistribution
- Delete
- Duplicates
- Key Compression
- Bulk Loading
- Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree Insert: Further Examples



## Example (B<sup>+</sup>-tree insertion procedure)



... pointers to data pages ...

Insert new entry with key 4222.

### Binary Search

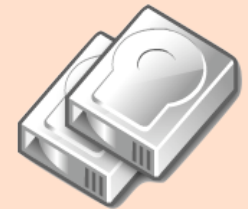
### ISAM

- Multi-Level ISAM
- Too Static?
- Search Efficiency

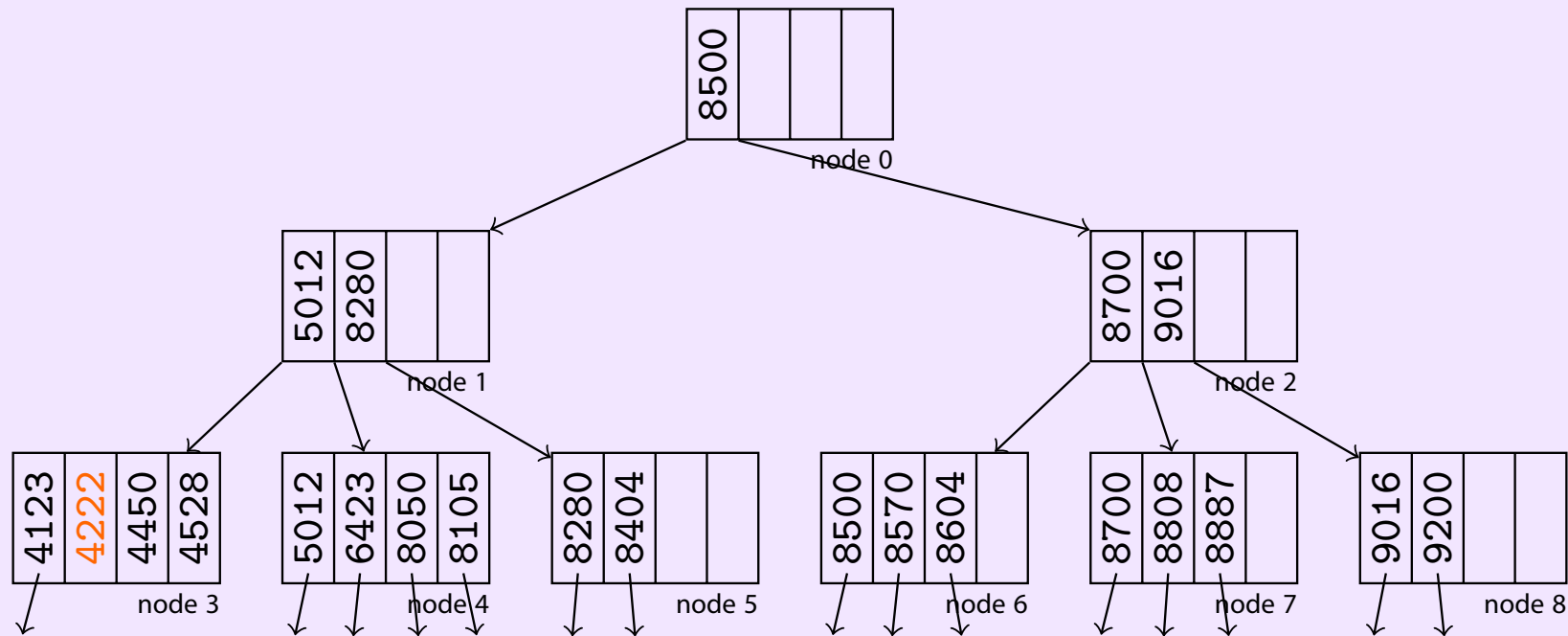
### B<sup>+</sup>-trees

- Search
- Insert
- Redistribution
- Delete
- Duplicates
- Key Compression
- Bulk Loading
- Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree Insert: Further Examples



## Example (B<sup>+</sup>-tree insertion procedure)



... pointers to data pages ...

Insert new entry with key 4222.

⇒ Enough space in node 3, simply insert.

⇒ Keep entries **sorted within nodes**.

### Binary Search

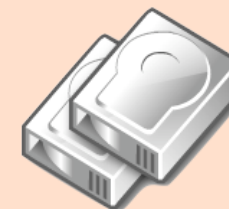
### ISAM

- Multi-Level ISAM
- Too Static?
- Search Efficiency

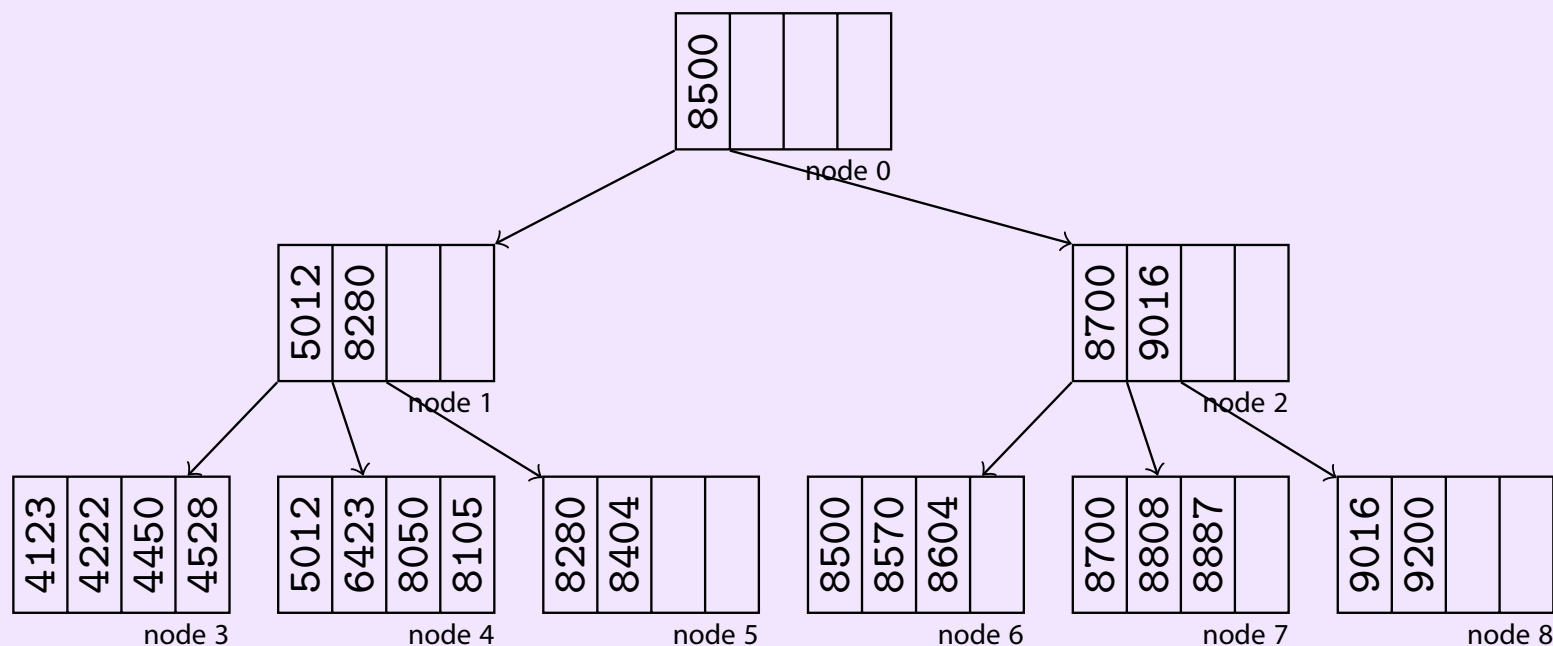
### B<sup>+</sup>-trees

- Search
- Insert
- Redistribution
- Delete
- Duplicates
- Key Compression
- Bulk Loading
- Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree Insert: Further Examples



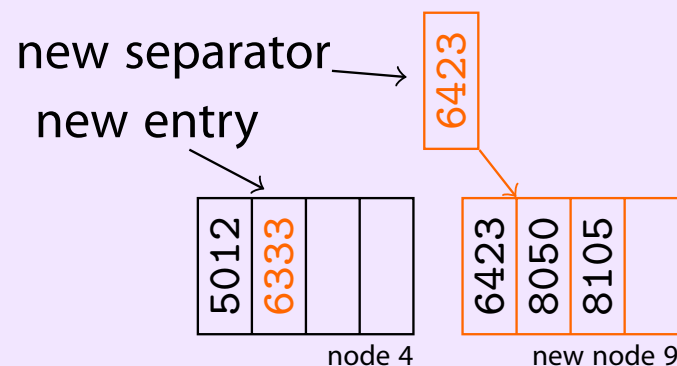
## Example (B<sup>+</sup>-tree insertion procedure)



Insert key **6333**.

⇒ Must **split** node 4.

⇒ **New separator** goes into node 1 (including pointer to new page).



### Binary Search

### ISAM

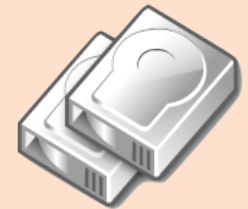
- Multi-Level ISAM
- Too Static?
- Search Efficiency

### B<sup>+</sup>-trees

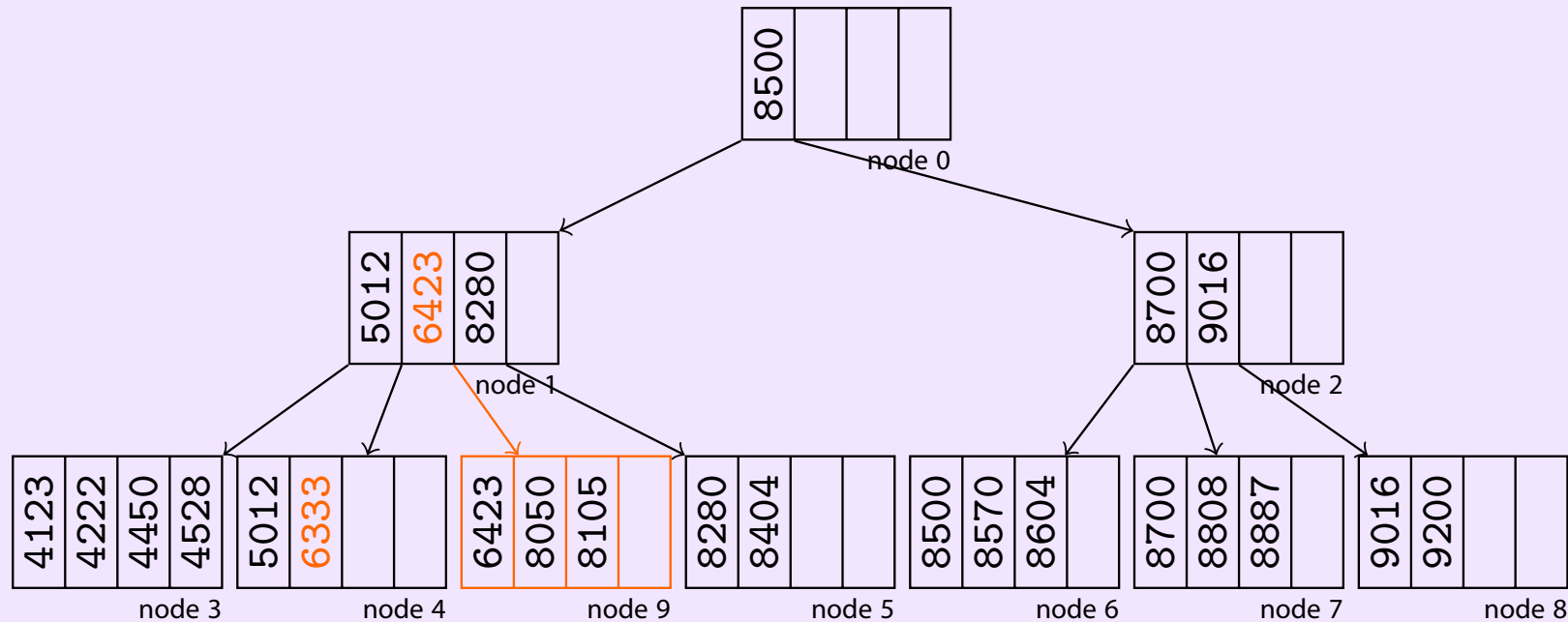
- Search
- Insert
- Redistribution
- Delete
- Duplicates
- Key Compression
- Bulk Loading
- Partitioned B<sup>+</sup>-trees



# B<sup>+</sup>-tree Insert: Further Examples

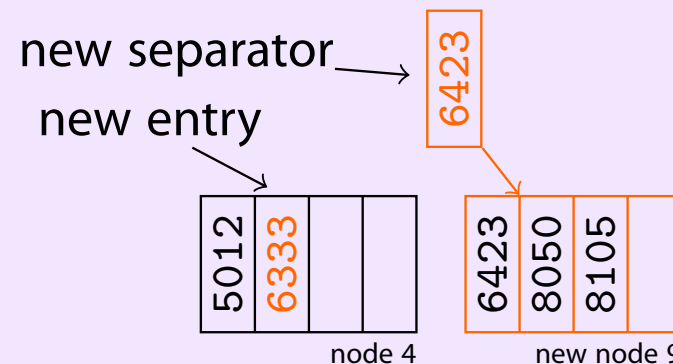


## Example (B<sup>+</sup>-tree insertion procedure)



Insert key 6333.

- ⇒ Must **split** node 4.
- ⇒ **New separator** goes into node 1 (including pointer to new page).



### Binary Search

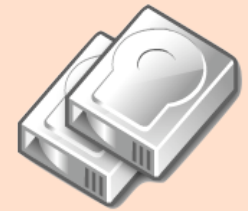
### ISAM

- Multi-Level ISAM
- Too Static?
- Search Efficiency

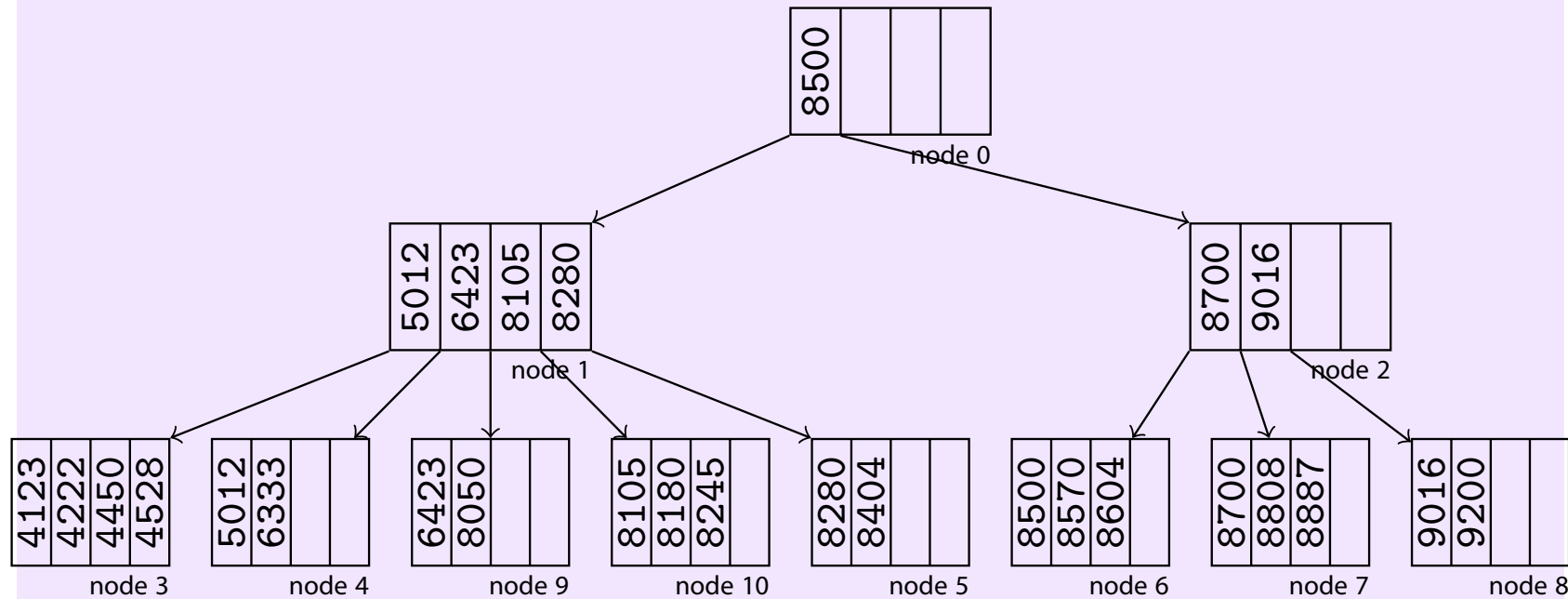
### B<sup>+</sup>-trees

- Search
- Insert
- Redistribution
- Delete
- Duplicates
- Key Compression
- Bulk Loading
- Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree Insert: Further Examples

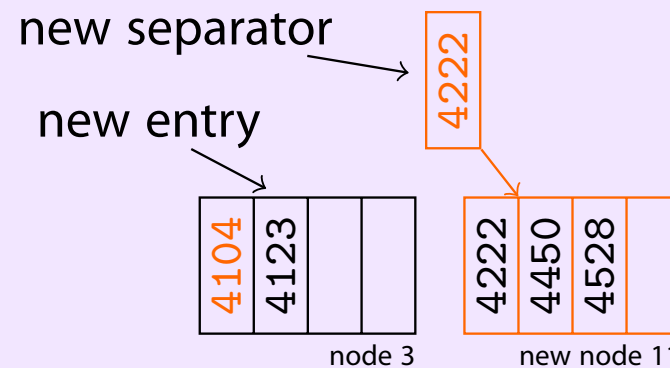


## Example (B<sup>+</sup>-tree insertion procedure)



After 8180, 8245, insert key 4104.

⇒ Must **split** node 3.



### Binary Search

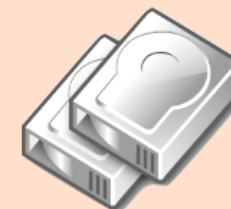
### ISAM

- Multi-Level ISAM
- Too Static?
- Search Efficiency

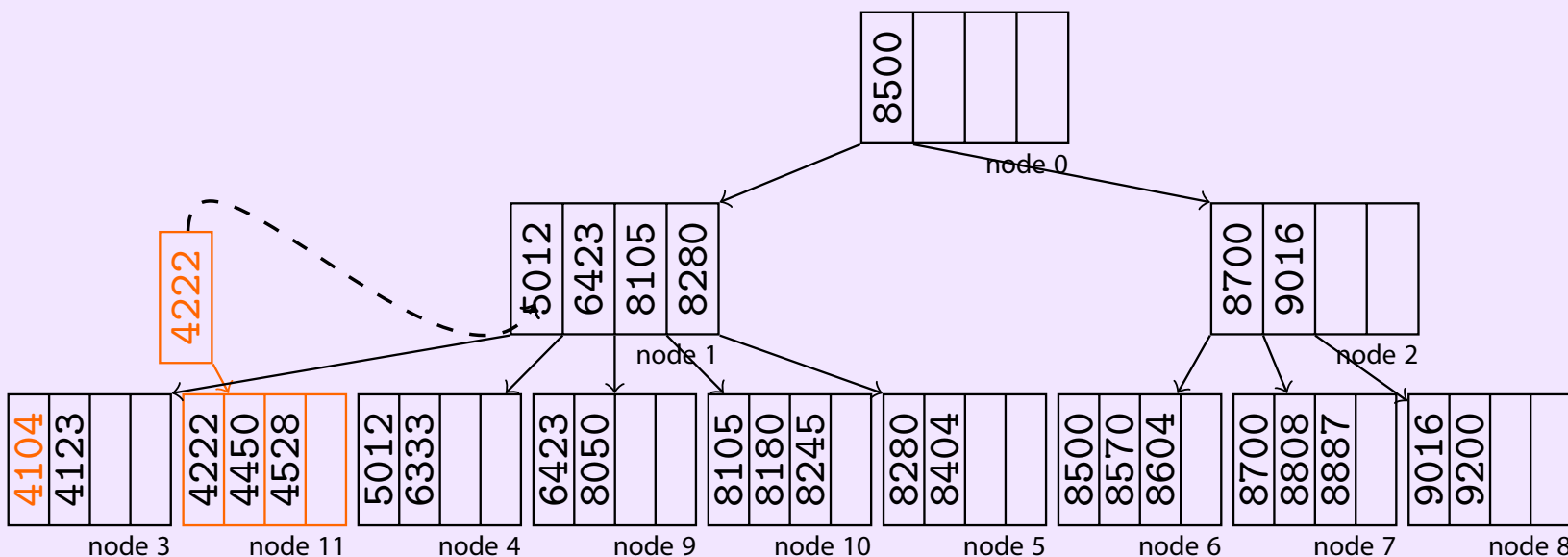
### B<sup>+</sup>-trees

- Search
- Insert
- Redistribution
- Delete
- Duplicates
- Key Compression
- Bulk Loading
- Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree Insert: Further Examples



## Example (B<sup>+</sup>-tree insertion procedure)

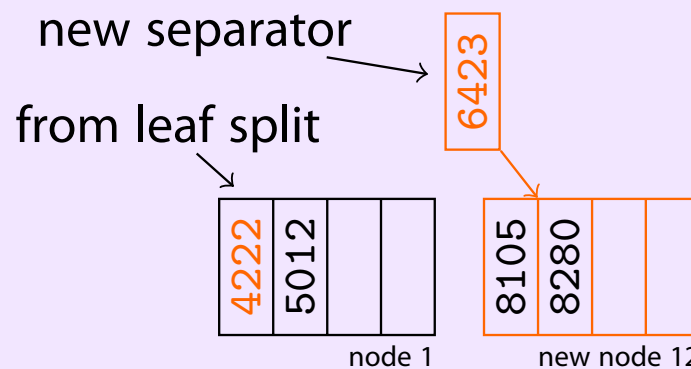


After 8180, 8245, insert key 4104.

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

Unlike during leaf split, separator key does **not** remain in inner node.

**Why?**



### Binary Search

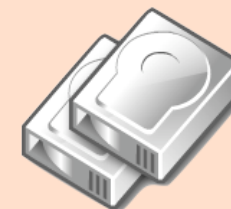
### ISAM

- Multi-Level ISAM
- Too Static?
- Search Efficiency

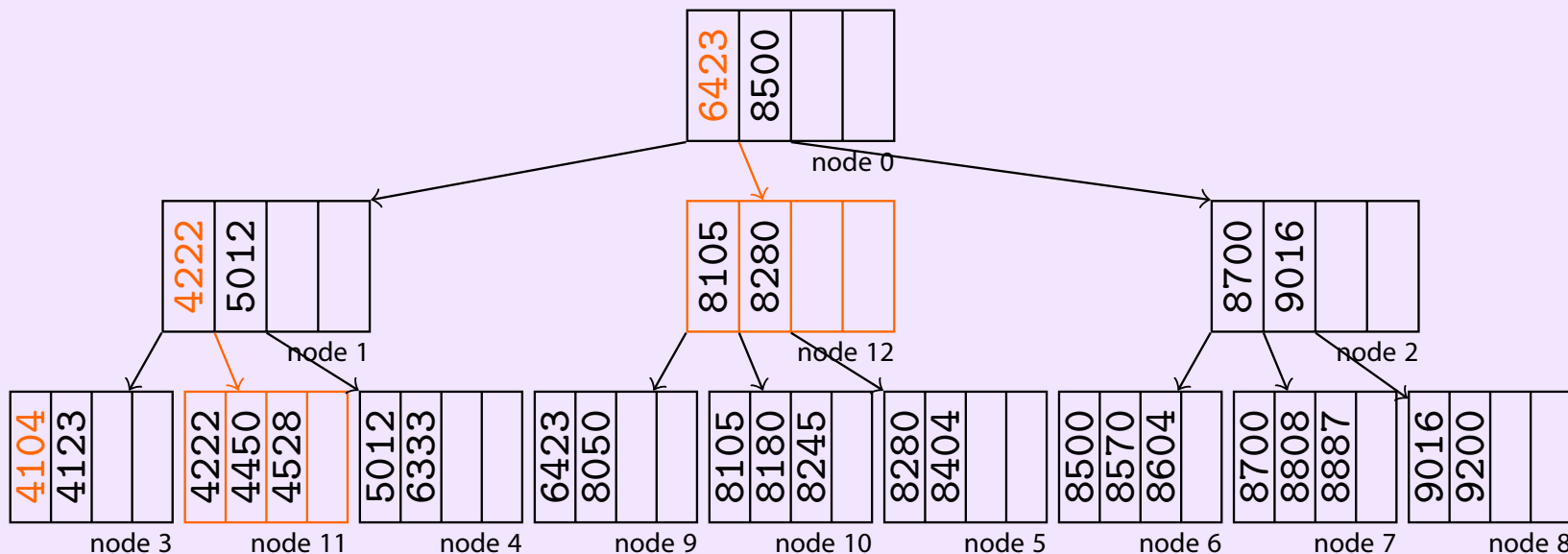
### B<sup>+</sup>-trees

- Search
- Insert**
- Redistribution
- Delete
- Duplicates
- Key Compression
- Bulk Loading
- Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree Insert: Further Examples



## Example (B<sup>+</sup>-tree insertion procedure)

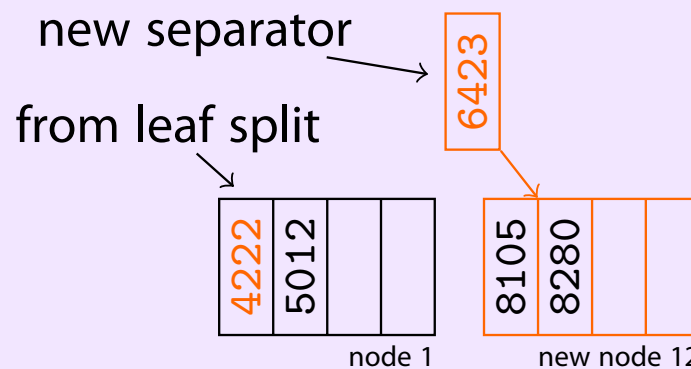


After 8180, 8245, insert key 4104.

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

Unlike during leaf split, separator key does **not** remain in inner node.

**Why?**



### Binary Search

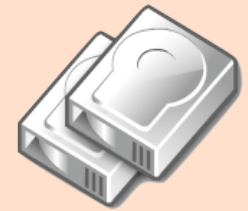
### ISAM

- Multi-Level ISAM
- Too Static?
- Search Efficiency

### B<sup>+</sup>-trees

- Search
- Insert**
- Redistribution
- Delete
- Duplicates
- Key Compression
- Bulk Loading
- Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Root Node Split



## Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

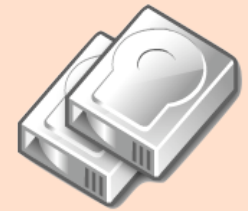
Partitioned B<sup>+</sup>-trees

- 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?**

# B<sup>+</sup>-tree: Root Node Split



## Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

- 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.

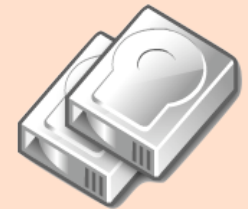
## How often do you expect a root split to happen?

*E.g.*, B<sup>+</sup>-tree over 8 byte integers, 4 KB pages; pointers encoded as 8 byte integers.

- 128–256 index entries/page (fan-out  $F$ ).
- An index of height  $h$  indexes **at least**  $128^h$  records, typically more.

$h$	# records
2	16,000
3	2,000,000
4	250,000,000

# B<sup>+</sup>-tree: Insertion Algorithm



## B<sup>+</sup>-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  $k < k_1$ 
7       | |  $\langle sep, ptr \rangle \leftarrow$  tree_insert (k, rid,  $p_0$ );
8     | case  $k_i \leq k < k_{i+1}$ 
9       | |  $\langle sep, ptr \rangle \leftarrow$  tree_insert (k, rid,  $p_i$ );
10    | case  $k_{2d} \leq k$ 
11      | |  $\langle sep, ptr \rangle \leftarrow$  tree_insert (k, rid,  $p_{2d}$ );
12    | if sep is null then
13      | | return  $\langle \text{null}, \text{null} \rangle$ ;
14    | else
15      | | return non_leaf_insert (sep, ptr, node);
```

} see tree\_search ()

### Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

Partitioned B<sup>+</sup>-trees

```

1 Function: leaf_insert ( $k, rid, node$ )
2 if another entry fits into  $node$  then
3   insert  $\langle k, rid \rangle$  into  $node$  ;
4   return  $\langle \text{null}, \text{null} \rangle$ ;
5 else
6   allocate new leaf page  $p$  ;
7   let  $\{ \langle k_1^+, p_1^+ \rangle, \dots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle \} :=$  entries from  $node \cup \{ \langle k, rid \rangle \}$ 
8     leave entries  $\langle k_1^+, p_1^+ \rangle, \dots, \langle k_d^+, p_d^+ \rangle$  in  $node$  ;
9     move entries  $\langle k_{d+1}^+, p_{d+1}^+ \rangle, \dots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle$  to  $p$  ;
10  return  $\langle k_{d+1}^+, p \rangle$ ;

```

---

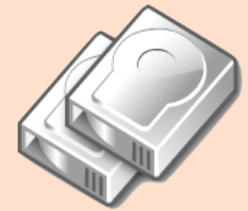
```

1 Function: non_leaf_insert ( $k, ptr, node$ )
2 if another entry fits into  $node$  then
3   insert  $\langle k, ptr \rangle$  into  $node$  ;
4   return  $\langle \text{null}, \text{null} \rangle$ ;
5 else
6   allocate new leaf page  $p$  ;
7   let  $\{ \langle k_1^+, p_1^+ \rangle, \dots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle \} :=$  entries from  $node \cup \{ \langle k, ptr \rangle \}$ 
8     leave entries  $\langle k_1^+, p_1^+ \rangle, \dots, \langle k_d^+, p_d^+ \rangle$  in  $node$  ;
9     move entries  $\langle k_{d+2}^+, p_{d+2}^+ \rangle, \dots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle$  to  $p$  ;
10    set  $p_0 \leftarrow p_{d+1}^+$  in  $p$ ;
11  return  $\langle k_{d+1}^+, p \rangle$ ;

```



# B<sup>+</sup>-tree: Insertion Algorithm



## Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

Duplicates

Key Compression

Bulk Loading

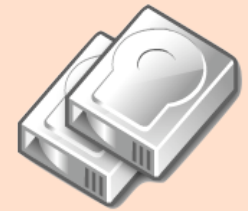
Partitioned B<sup>+</sup>-trees

## B<sup>+</sup>-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 r 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.
- Variable *root* contains a pointer to the B<sup>+</sup>-tree root page.
- Note how leaf node entries point to *rids*, while inner nodes contain pointers to other B<sup>+</sup>-tree nodes.

# B<sup>+</sup>-tree Insert: Redistribution



## Binary Search

## ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

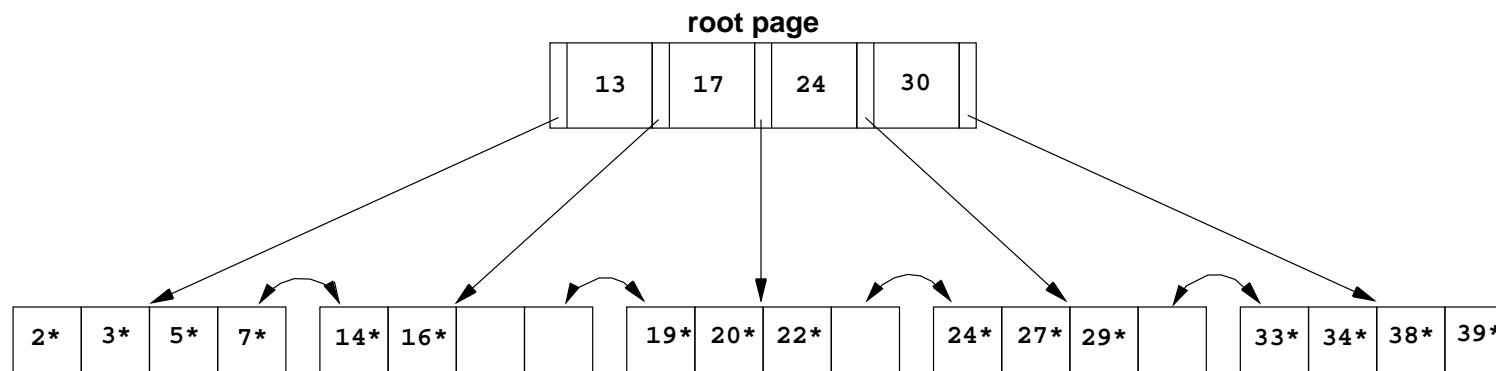
## B<sup>+</sup>-trees

Search  
Insert

## Redistribution

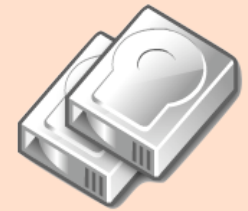
Delete  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

- We can further **improve the average occupancy** of B<sup>+</sup>-tree using a technique called **redistribution**.
- Suppose we are trying to insert a record with key  $k = 6$  into the B<sup>+</sup>-tree below:



- The left-most leaf is full already, its right **sibling** still has capacity, however.

# B<sup>+</sup>-tree Insert: Redistribution



## Binary Search

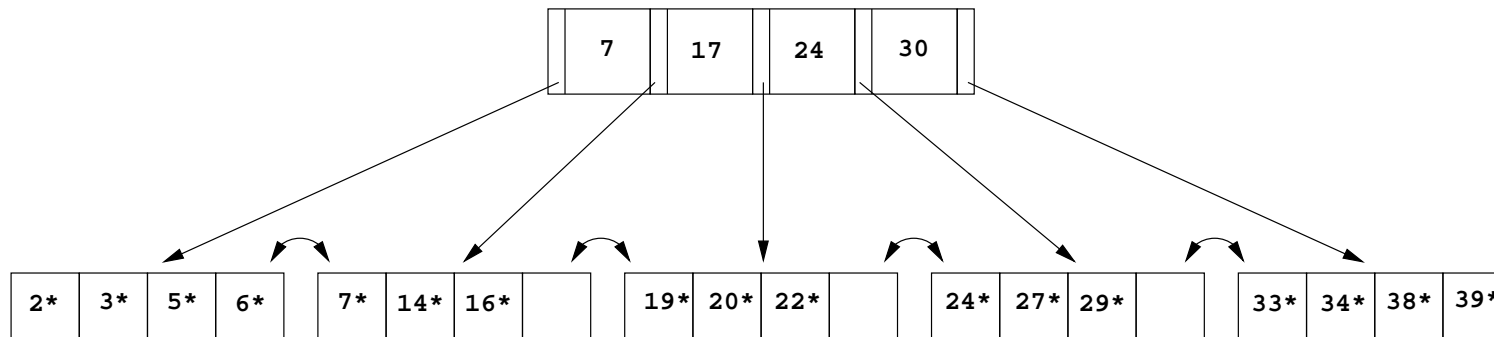
## ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

## B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

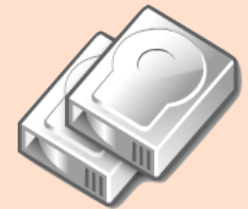
- In this situation, we can avoid growing the tree by **redistributing** entries between siblings (entry 7\* moved into right sibling):



- We have to **update the parent node** (new separator 7) to reflect the redistribution.
  - Inspecting one or both neighbor(s) of a B<sup>+</sup>-tree node involves additional I/O operations.
- ⇒ Actual implementations often use redistribution on the leaf level only (because the sequence set page chaining gives direct access to both sibling pages).



# B<sup>+</sup>-tree Insert: Redistribution

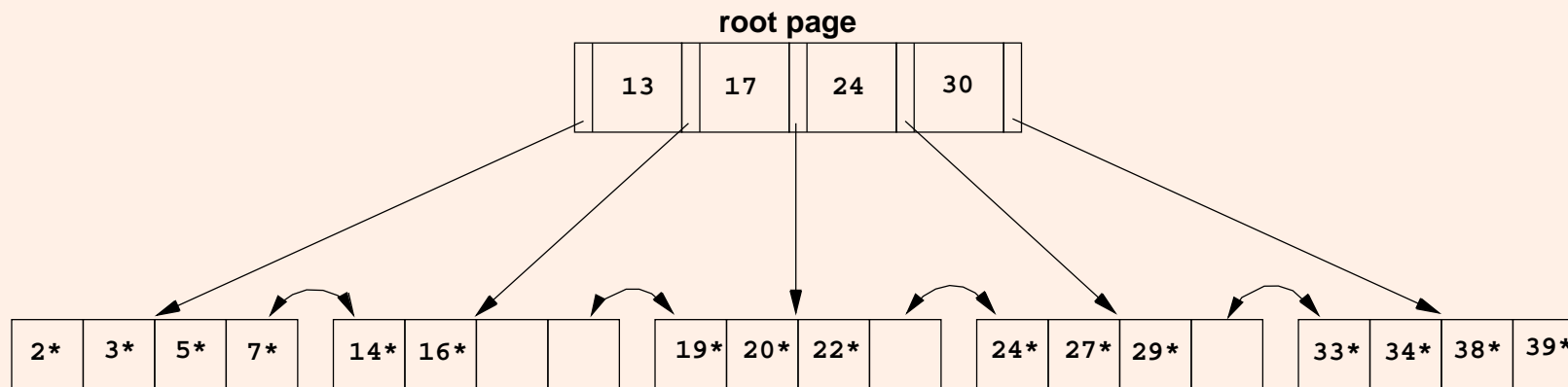


## Redistribution makes a difference

Insert a record with key  $k = 30$

- 1 without redistribution,
- 2 using leaf level redistribution

into the B<sup>+</sup>-tree shown below. How does the tree change?



### Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

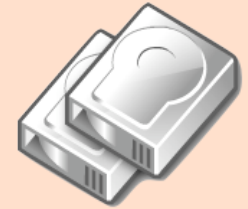
### B<sup>+</sup>-trees

Search  
Insert

### Redistribution

Delete  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Delete



## Binary Search

### ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

### B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

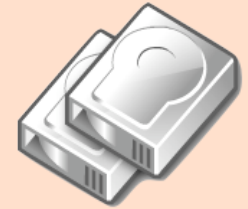
Duplicates

Key Compression

Bulk Loading

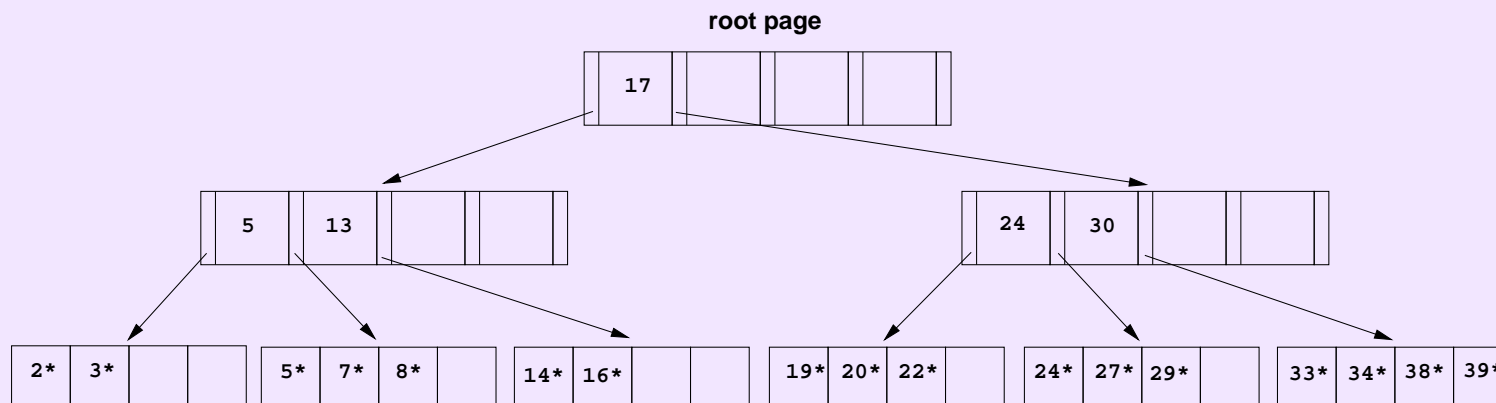
Partitioned B<sup>+</sup>-trees

- The principal idea to implement B<sup>+</sup>-tree deletion comes as no surprise:
  - 1 To delete a record with key  $k$ , use  $\text{search}(k)$  to locate the leaf page  $p$  containing the record.  
Let  $m$  denote the number of entries on  $p$ .
  - 2 If  $m > d$  then  $p$  has sufficient occupancy: simply delete  $k^*$  from  $p$  (if  $k^*$  is present on  $p$  at all).  
**Otherwise ...?**



## Example (B<sup>+</sup>-tree deletion procedure)

- 1 Delete record with key  $k = 19$  (i.e., entry 19\*) from the following B<sup>+</sup>-tree:



- 2 A call to search(19) leads us to leaf page  $p$  containing entries 19\*, 20\*, and 22\*. We can safely remove 19\* since  $m = 3 > 2$  (no page underflow in  $p$  after removal).

### Binary Search

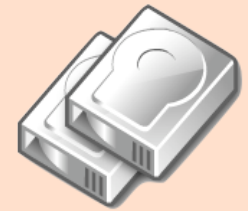
### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Delete and Leaf Redistribution

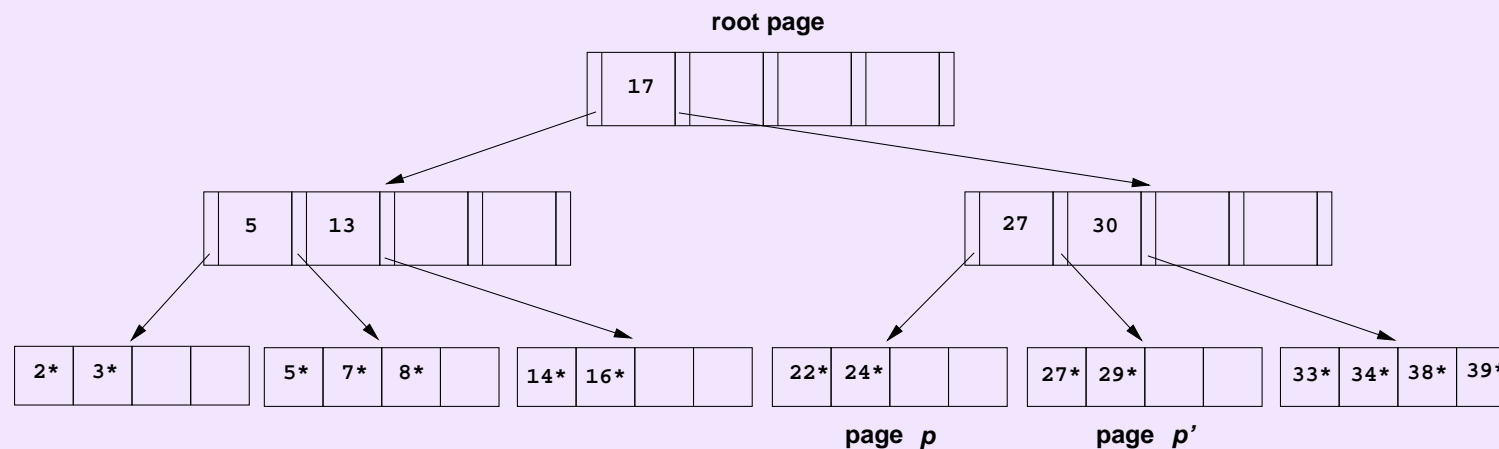


## Example (B<sup>+</sup>-tree deletion procedure)

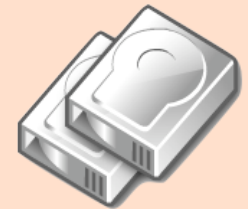
- ③ Subsequent deletion of 20\*, however, lets  $p$  **underflow** ( $p$  has minimal occupancy of  $d = 2$  already).

We now use **redistribution** and borrow entry 24\* from the right **sibling**  $p'$  of  $p$  (since  $p'$  hosts 3  $>$  2 entries, redistribution won't let  $p'$  underflow).

The smallest key value on  $p'$  (27) is the **new separator** of  $p$  and  $p'$  in their common parent:



# B<sup>+</sup>-tree: Delete and Leaf Merging



## Binary Search

## ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

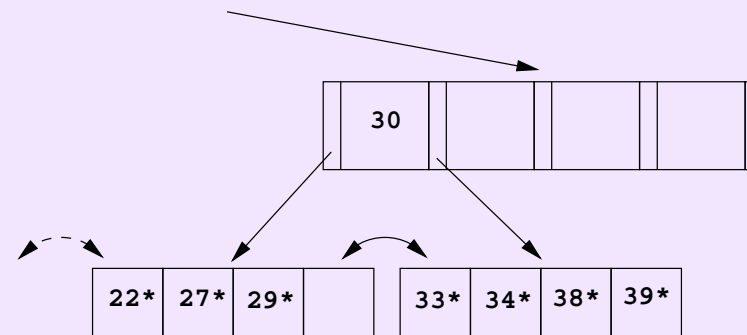
## B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
**Delete**  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

## Example (B<sup>+</sup>-tree deletion procedure)

- 4 We continue and delete entry 24\* from  $p$ . Redistribution is no option now (sibling  $p'$  has minimal occupancy of  $d = 2$ ). We now have  $m_p + m_{p'} = 1 + 2 < 2 \cdot d$  however: B<sup>+</sup>-tree deletion thus **merges leaf nodes  $p$  and  $p'$** .

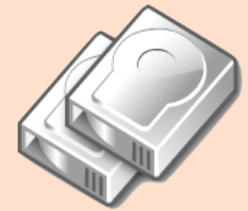
Move entries 27\*, 29\* from  $p'$  to  $p$ , then delete page  $p'$ :



- **NB:** the separator 27 between  $p$  and  $p'$  is no longer needed and thus **discarded (recursively deleted)** from the parent.



# B<sup>+</sup>-tree: Delete and Non-Leaf Node Merging



## Binary Search

## ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

## B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
**Delete**  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

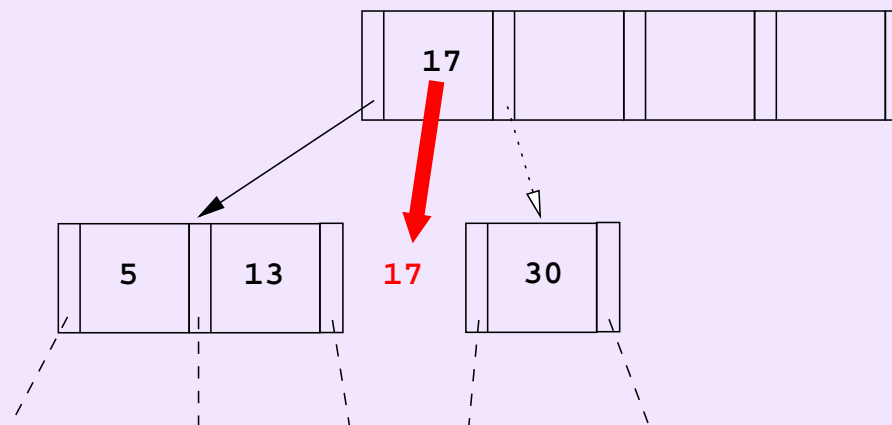
## Example (B<sup>+</sup>-tree deletion procedure)

- 5 The parent of  $p$  experiences underflow. Redistribution is no option, so we **merge with left non-leaf sibling**.

After merging we have

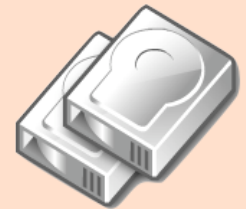
$$\underbrace{d}_{\text{left}} + \underbrace{(d-1)}_{\text{right}} \text{ keys and } \underbrace{d+1}_{\text{left}} + \underbrace{d}_{\text{right}} \text{ pointers}$$

on the merged page:



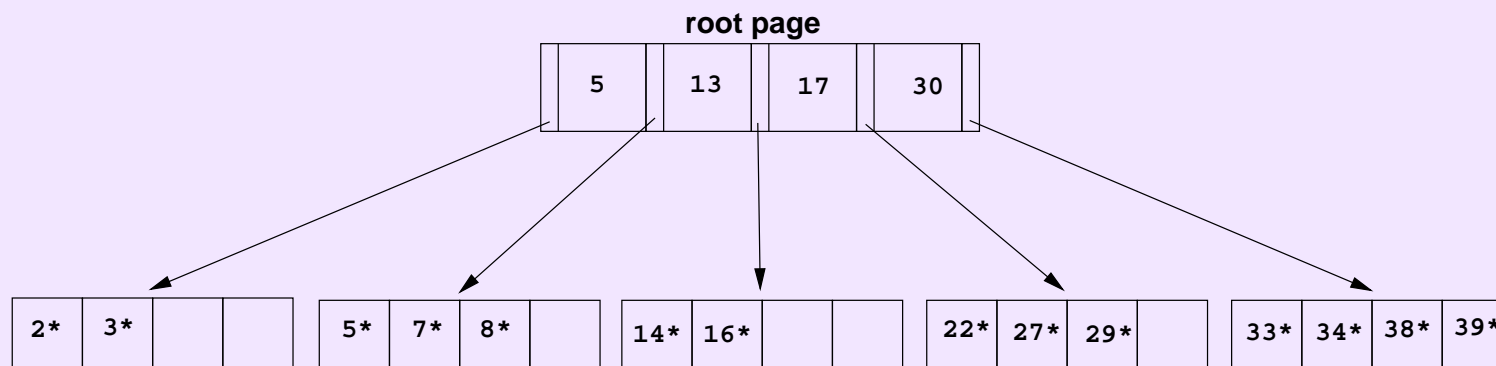
The missing key value, namely the separator of the two nodes (17), **is pulled down** (and thus deleted) from the parent to form the complete merged node.

# B<sup>+</sup>-tree: Root Deletion



## Example (B<sup>+</sup>-tree deletion procedure)

- Since we have now deleted the last remaining entry in the root, we discard the root (and make the merged node the new root):



- This is the *only* situation in which the **B<sup>+</sup>-tree height decreases**. The B<sup>+</sup>-tree thus remains balanced.

### Binary Search

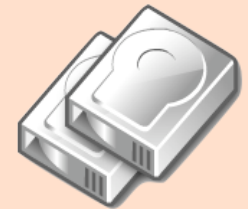
### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
**Delete**  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

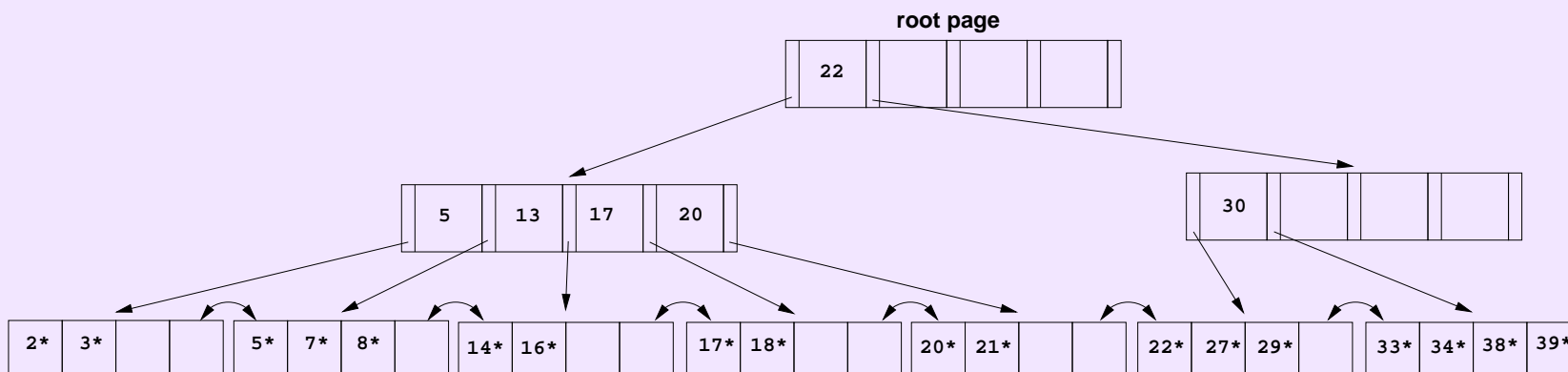
# B<sup>+</sup>-tree: Delete and Non-Leaf Node Redistribution



## Example (B<sup>+</sup>-tree deletion procedure)

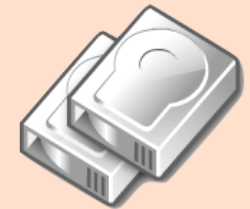
7 We have now seen **leaf node merging** and **redistribution** as well as **non-leaf node merging**. The remaining case of **non-leaf node redistribution** is straightforward:

- Suppose *during* deletion we encounter the following B<sup>+</sup>-tree:



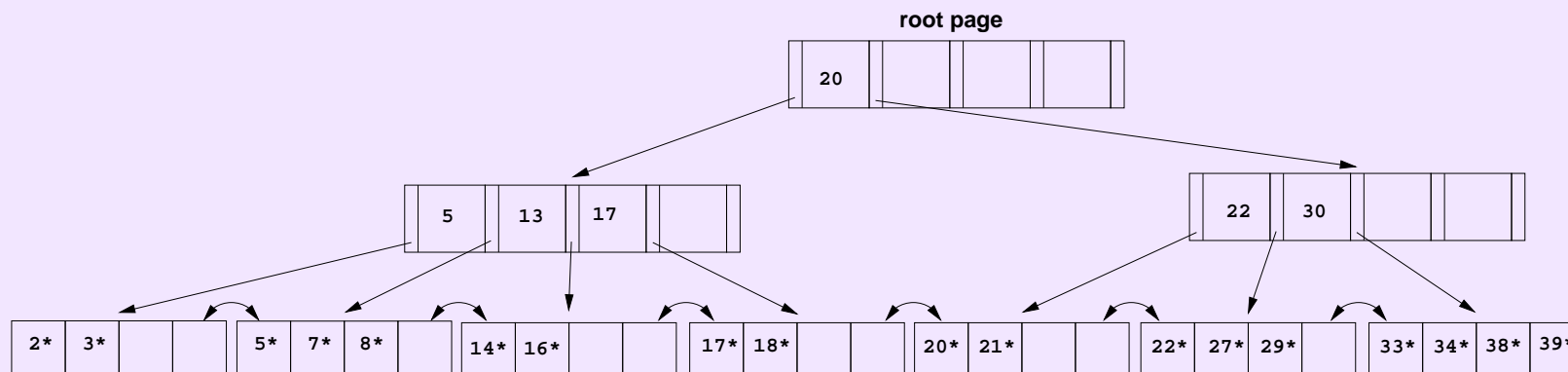
- The non-leaf node with entry 30 underflowed. Its left sibling has two entries (17 and 20) to spare.

# B<sup>+</sup>-tree: Delete and Non-Leaf Node Redistribution



## Example (B<sup>+</sup>-tree deletion procedure)

- ⑧ We redistribute entry 20 by “rotating it through” the parent. The former parent entry 22 is pushed down:



### Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

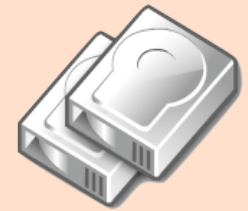
### B<sup>+</sup>-trees

Search  
Insert  
Redistribution

### Delete

Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

# Merge and Redistribution Effort



## Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

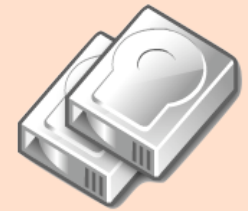
Search  
Insert  
Redistribution  
**Delete**  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

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

## DB2<sup>®</sup> B<sup>+</sup>-tree deletion

- System parameter MINPCTUSED (*minimum percent used*) controls when the kernel should try a **leaf node merge** (“online index reorg”).  
(This is particularly simple because of the sequence set pointers connecting adjacent leaves, see slide 40.)
- **Non-leaf nodes are never merged** (a “full index reorg” is required to achieve this).
- To improve concurrency, deleted index entries are merely **marked as deleted** and only removed later (IBM DB2 UDB type-2 indexes).

# B<sup>+</sup>-tree: Duplicates



## Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete

### Duplicates

Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

- As discussed here, the B<sup>+</sup>-tree search, insert (and delete) procedures ignore the presence of **duplicate** key values.
- Often this is a reasonable assumption:
  - If the key field is a **primary key** for the data file (*i.e.*, for the associated relation), the search keys  $k$  are unique by definition.

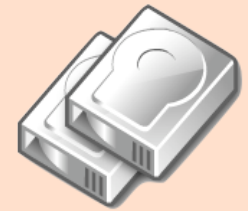
## DB2® Treatment of duplicate keys

Since duplicate keys add to the B<sup>+</sup>-tree complexity, IBM DB2 **forces uniqueness** by forming a composite key of the form  $\langle k, id \rangle$  where  $id$  is the unique **tuple identity** of the data record with key  $k$ .

### Tuple identities

- 1 are system-maintained unique identifiers for each tuple in a table, and
- 2 are *not* dependent on tuple order and *never* rise again.

# B<sup>+</sup>-tree: Duplicates (IBM DB2 Tuple Identities)



## Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete

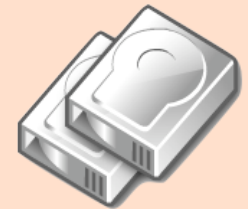
### Duplicates

Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

## DB2<sup>®</sup> Expose IBM DB2 tuple identity

```
1 $ db2
2 (c) Copyright IBM Corporation 1993,2007
3 Command Line Processor for DB2 Client 9.5.0
4 [...]
5 db2 => CREATE TABLE FOO(ROWID INT GENERATED ALWAYS AS IDENTITY,
6         text varchar(10))
7 db2 => INSERT INTO FOO VALUES (DEFAULT, 'This is'), ...
8 db2 => SELECT * FROM FOO
9
10 ROWID      TEXT
11 -----
12          1 This is
13          2 nothing
14          3 but a
15          4 silly
16          5 example!
```

# B<sup>+</sup>-tree: Duplicates (IBM DB2 Tuple Identities)



## DB2® Expose IBM DB2 tuple identity (continued)

```
1 db2 => DELETE FROM F00 WHERE TEXT = 'silly'
```

```
2 db2 => SELECT * FROM F00
```

```
3
4 ROWID          TEXT
5 -----
6              1 This is
7              2 nothing
8              3 but a
9              5 example!
```

```
11 db2 => INSERT INTO F00 VALUES (DEFAULT, 'I am new.')
```

```
12 db2 => SELECT * FROM F00
```

```
13
14 ROWID          TEXT
15 -----
16              1 This is
17              2 nothing
18              3 but a
19              6 I am new.
20              5 example!
```

### Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete

### Duplicates

Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees



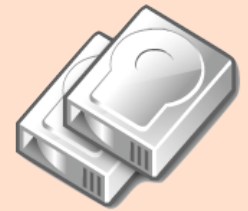
# B<sup>+</sup>-tree: Duplicates

Other approaches alter the B<sup>+</sup>-tree implementation to add real awareness for duplicates:

- 1 Use variant **C** (see slide 0.0) to represent the index data entries  $k^*$ :

$$k^* = \langle k, [rid_1, rid_2, \dots] \rangle$$

- Each duplicate record with key field  $k$  makes the list of *rids* grow. Key  $k$  is not repeatedly stored (space savings).
- B<sup>+</sup>-tree search and maintenance routines largely unaffected. Index data entry size varies, however (this affects the B<sup>+</sup>-tree **order** concept).
- Implemented in IBM Informix Dynamic Server, for example.



## Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

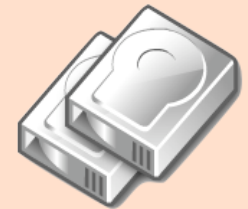
### B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete

### Duplicates

Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Duplicates



## Binary Search

## ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

## B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete

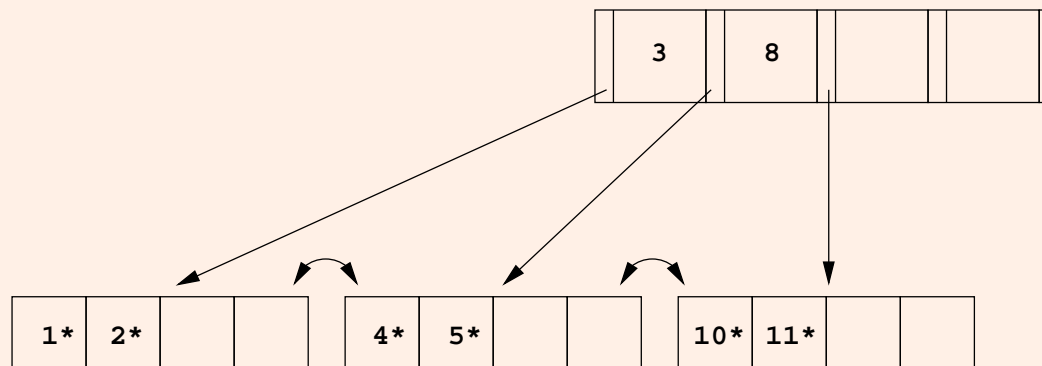
## Duplicates

Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

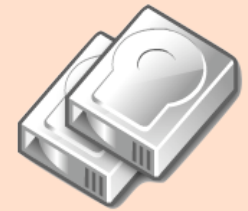
- 2 Treat duplicate key values like any other value in insert and delete. This affects the search procedure.

### Impact of duplicate insertion on search

Given the following B<sup>+</sup>-tree of order  $d = 2$ , perform insertions (do not use redistribution): insert(2, ·), insert(2, ·), insert(2, ·):

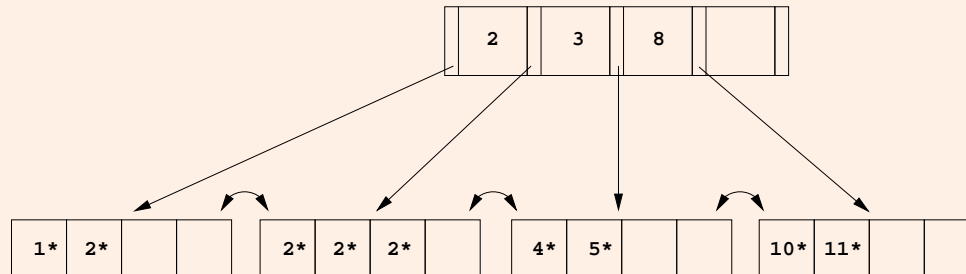


# B<sup>+</sup>-tree: Duplicates

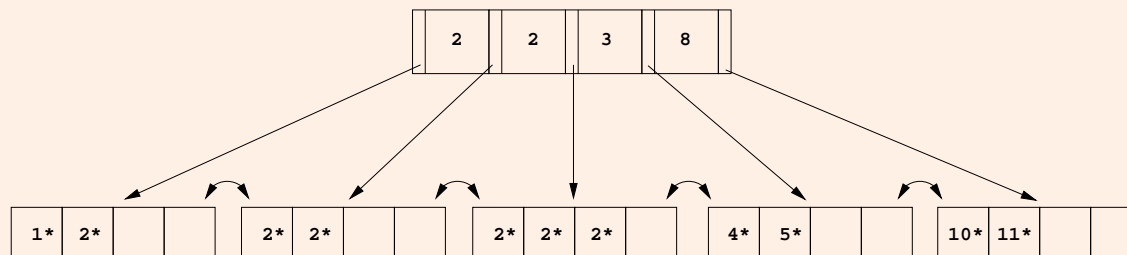


## Impact of duplicate insertion on search

The resulting B<sup>+</sup>-tree is shown here. Now apply  $\text{insert}(2, \cdot)$ ,  $\text{insert}(2, \cdot)$  to this B<sup>+</sup>-tree:



We get the tree depicted below:



⇒ In search: in a non-leaf node, follow the **left-most** page pointer  $p_i$  such that  $k_i \leq k \leq k_{i+1}$ . In a leaf, also check the left sibling.

### Binary Search

### ISAM

- Multi-Level ISAM
- Too Static?
- Search Efficiency

### B<sup>+</sup>-trees

- Search
- Insert
- Redistribution
- Delete

### Duplicates

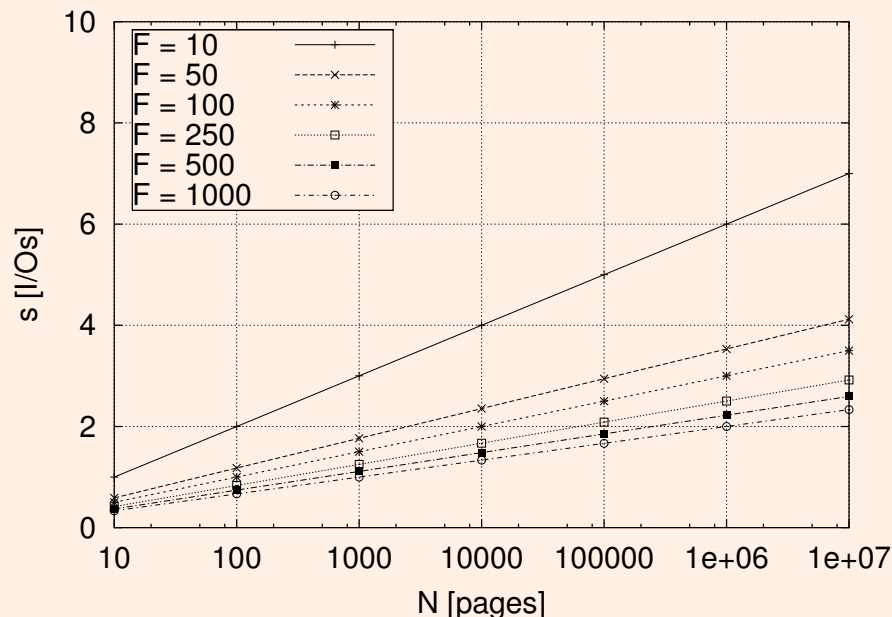
- Key Compression
- Bulk Loading
- Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Key Compression

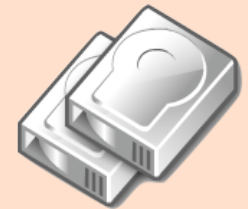
- Recall the search I/O effort  $s$  in an ISAM or B<sup>+</sup>-tree for a file of  $N$  pages. The **fan-out**  $F$  has been the deciding factor:

$$s = \log_F N .$$

## Tree index search effort dependent on fan-out $F$



⇒ It clearly pays off to invest effort and **try to maximize the fan-out**  $F$  of a given B<sup>+</sup>-tree implementation.



### Binary Search

#### ISAM

- Multi-Level ISAM
- Too Static?
- Search Efficiency

#### B<sup>+</sup>-trees

- Search
- Insert
- Redistribution
- Delete
- Duplicates

#### Key Compression

- Bulk Loading
- Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Key Compression

- Index entries in inner (*i.e.*, non-leaf) B<sup>+</sup>-tree nodes are pairs

$$\langle k_i, \text{pointer to } p_i \rangle .$$

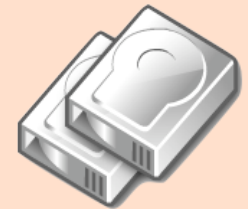
- The representation of page pointers is prescribed by the DBMS's pointer representation, and especially for key field types like CHAR(·) or VARCHAR(·), we will have

$$| \text{pointer} | \ll | k_i | .$$

- To **minimize the size of keys**, observe that key values in inner index nodes are used only to direct traffic to the appropriate leaf page:

## Excerpt of search(*k*)

```
1 switch k do
2   case k < k1
3     | ...
4   case ki ≤ k < ki+1
5     | ...
6   case k2d ≤ k
7     | ...
```



## Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

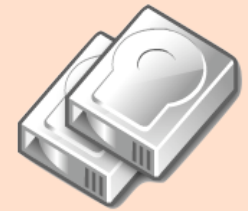
### B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete  
Duplicates

### Key Compression

Bulk Loading  
Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Key Compression



## Binary Search

### ISAM

- Multi-Level ISAM
- Too Static?
- Search Efficiency

### B<sup>+</sup>-trees

- Search
- Insert
- Redistribution
- Delete
- Duplicates

### Key Compression

- Bulk Loading
- Partitioned B<sup>+</sup>-trees

- Index entries in inner (*i.e.*, non-leaf) B<sup>+</sup>-tree nodes are pairs

$$\langle k_i, \text{pointer to } p_i \rangle .$$

- The representation of page pointers is prescribed by the DBMS's pointer representation, and especially for key field types like CHAR(·) or VARCHAR(·), we will have

$$| \text{pointer} | \ll | k_i | .$$

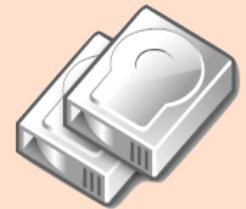
- To **minimize the size of keys**, observe that key values in inner index nodes are used only to direct traffic to the appropriate leaf page:

## Excerpt of search(*k*)

```
1 switch k do
2   case k < k1
3     | ...
4   case ki ≤ k < ki+1
5     | ...
6   case k2d ≤ k
7     | ...
```

⇒ The actual key values are *not* needed as long as we maintain their separator property.

# B<sup>+</sup>-tree: Key Compression



## Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

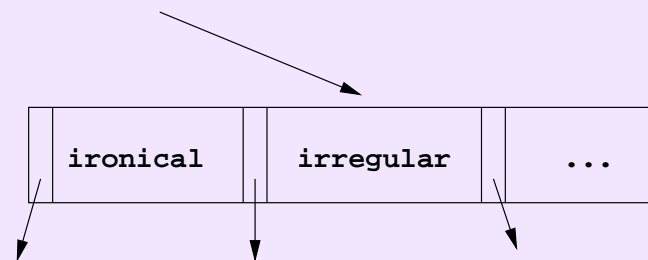
Search  
Insert  
Redistribution  
Delete  
Duplicates

### Key Compression

Bulk Loading  
Partitioned B<sup>+</sup>-trees

## Example (Searching a B<sup>+</sup>-tree node with VARCHAR(.) keys)

To guide the search across this B<sup>+</sup>-tree node



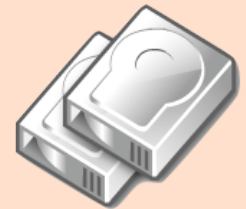
it is sufficient to store the **prefixes** iro and irr.

We must preserve the B<sup>+</sup>-tree semantics, though:

*All index entries stored in the subtree left of iro have keys  $k < \text{iro}$  and index entries stored in the subtree right of iro have keys  $k \geq \text{iro}$  (and  $k < \text{irr}$ ).*

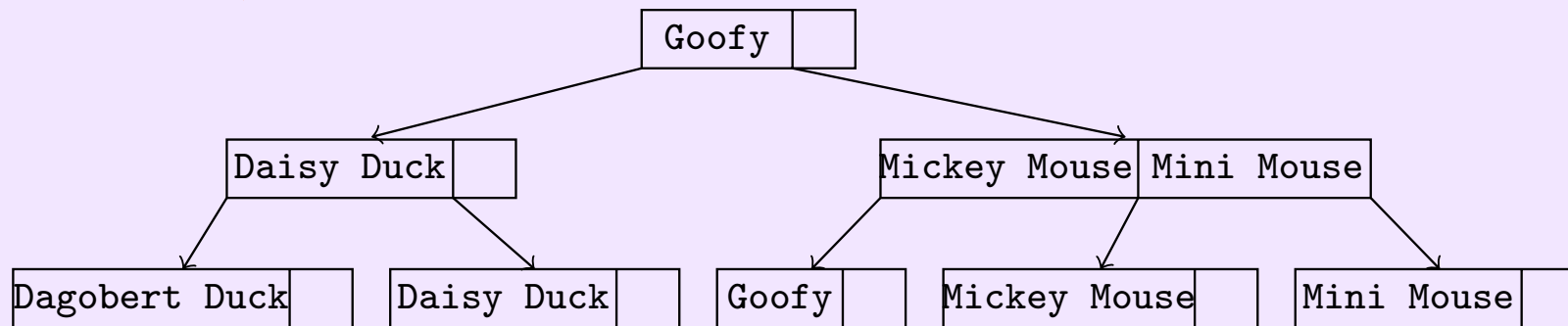


# B<sup>+</sup>-tree: Key Suffix Truncation



Example (Key suffix truncation, B<sup>+</sup>-tree with order  $d = 1$ )

Before **key suffix truncation**:



Binary Search

ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

B<sup>+</sup>-trees

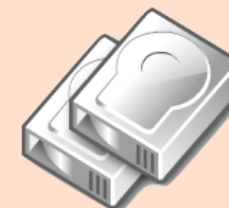
Search  
Insert  
Redistribution  
Delete  
Duplicates

Key Compression

Bulk Loading  
Partitioned B<sup>+</sup>-trees

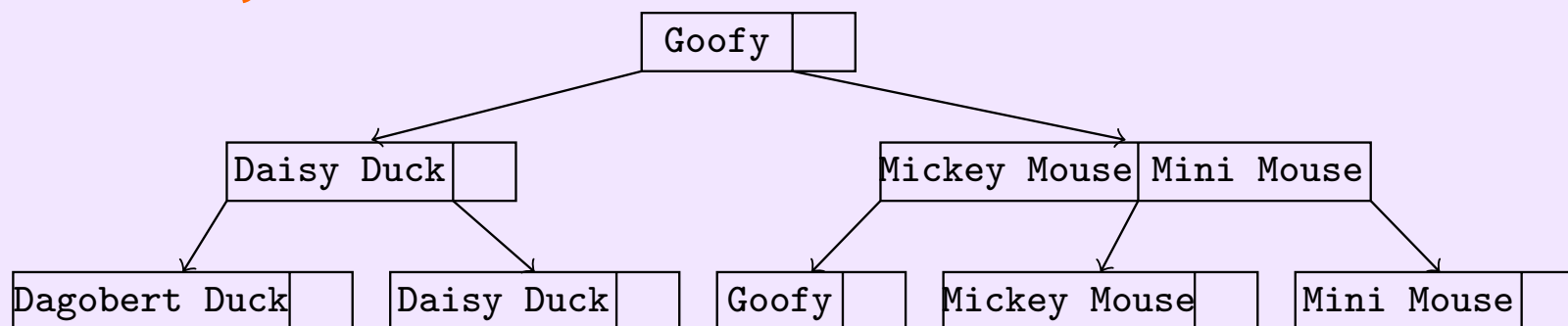


# B<sup>+</sup>-tree: Key Suffix Truncation

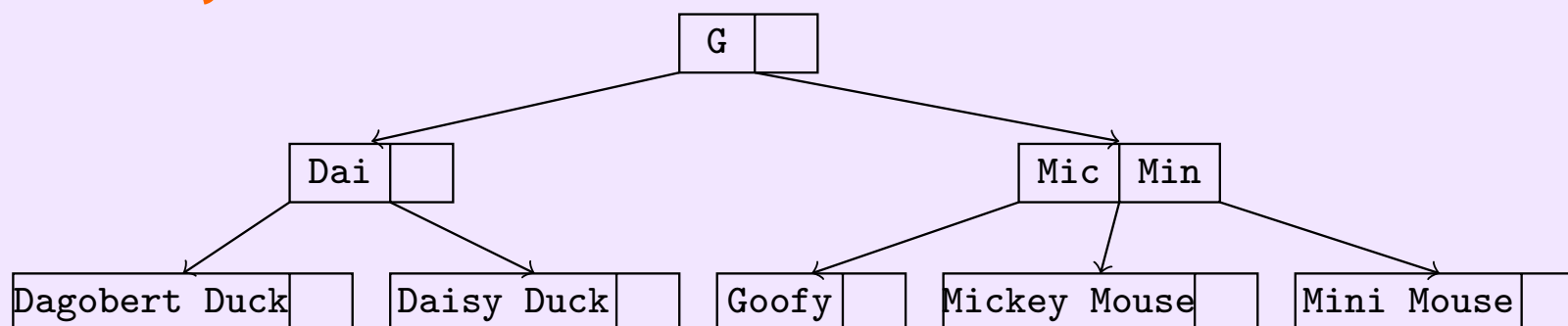


Example (Key suffix truncation, B<sup>+</sup>-tree with order  $d = 1$ )

Before key suffix truncation:



After key suffix truncation:



Binary Search

ISAM

Multi-Level ISAM

Too Static?

Search Efficiency

B<sup>+</sup>-trees

Search

Insert

Redistribution

Delete

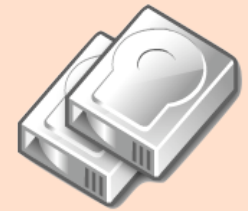
Duplicates

Key Compression

Bulk Loading

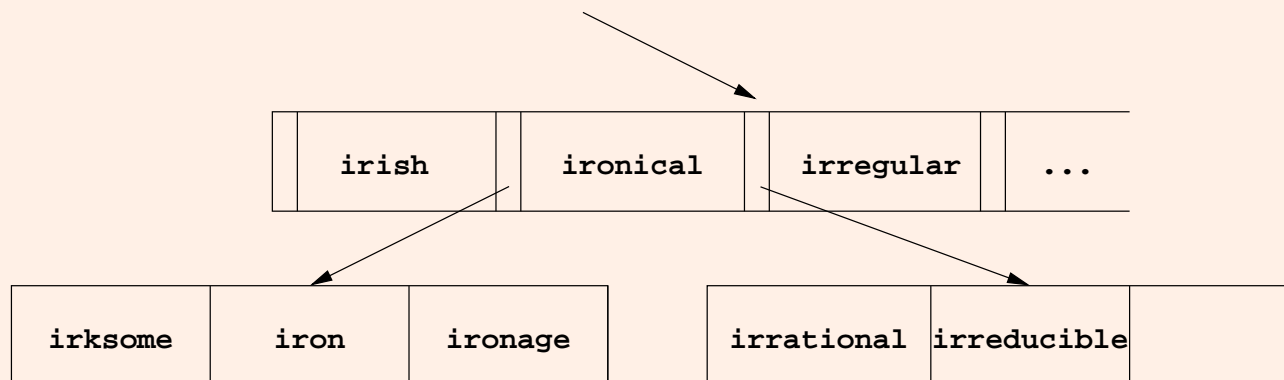
Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Key Suffix Truncation



## Key suffix truncation

How would a B<sup>+</sup>-tree **key compressor** alter the key entries in the inner node of this B<sup>+</sup>-tree snippet?



## Binary Search

## ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

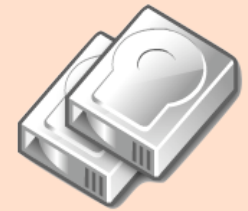
## B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete  
Duplicates

## Key Compression

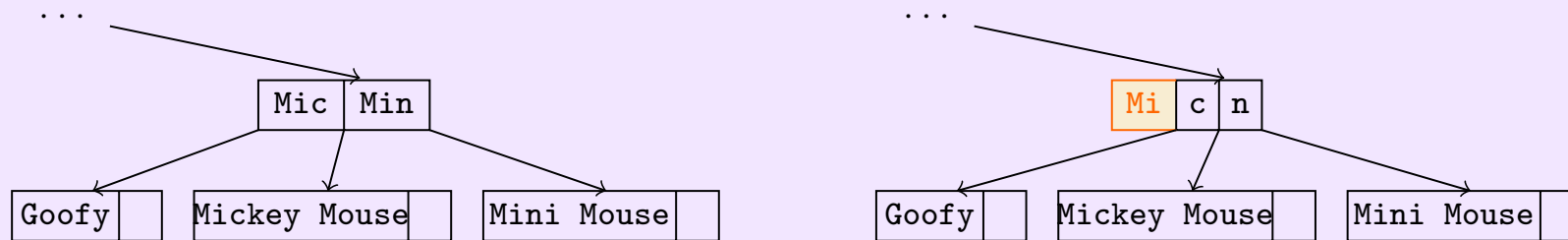
Bulk Loading  
Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Key Prefix Compression



- Observation: Keys within a B<sup>+</sup>-tree node often share a **common prefix**.

## Example (Shared key prefixes in inner B<sup>+</sup>-tree nodes)



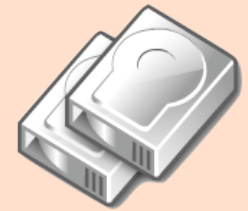
## Key prefix compression:

- Store common prefix only once (e.g., as “k<sub>0</sub>”)
- Keys have become highly discriminative now.

Violating the 50% occupancy rule can help to improve the effectiveness of prefix compression.

↗ Rudolf Bayer, Karl Unterauer: Prefix B-Trees. *ACM TODS* 2(1), March 1977.

# B<sup>+</sup>-tree: Bulk Loading



## Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression

### Bulk Loading

Partitioned B<sup>+</sup>-trees

- Consider the following database session (this might as well be commands executed on behalf of a database transaction):

## Table and index creation

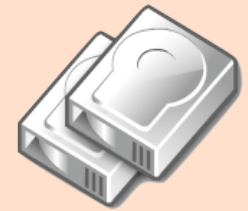
```
1 db2 => CREATE TABLE FOO (ID INT, TEXT VARCHAR(10));
2
3 [... insert 1,000,000 rows into table foo ...]
4
5 dn2 => CREATE INDEX FOO_IDX ON FOO (A ASC)
```

- The last SQL command initiates 1,000,000 calls to the B<sup>+</sup>-tree `insert(·)` procedure—a so-called index **bulk load**.
- ⇒ The DBMS will traverse the growing B<sup>+</sup>-tree index from its root down to the leaf pages 1,000,000 times.

## This is bad ...

...but at least it is not as bad as swapping the order of row insertion and index creation. Why?

# B<sup>+</sup>-tree: Bulk Loading



## Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression

### Bulk Loading

Partitioned B<sup>+</sup>-trees

- Most DBMS installations ship with a **bulk loading utility** to reduce the cost of operations like the above.

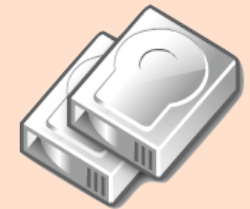
## B<sup>+</sup>-tree bulk loading algorithm

- 1 For each record with key  $k$  in the data file, create a **sorted list** of pages of index leaf entries  $k^*$ .

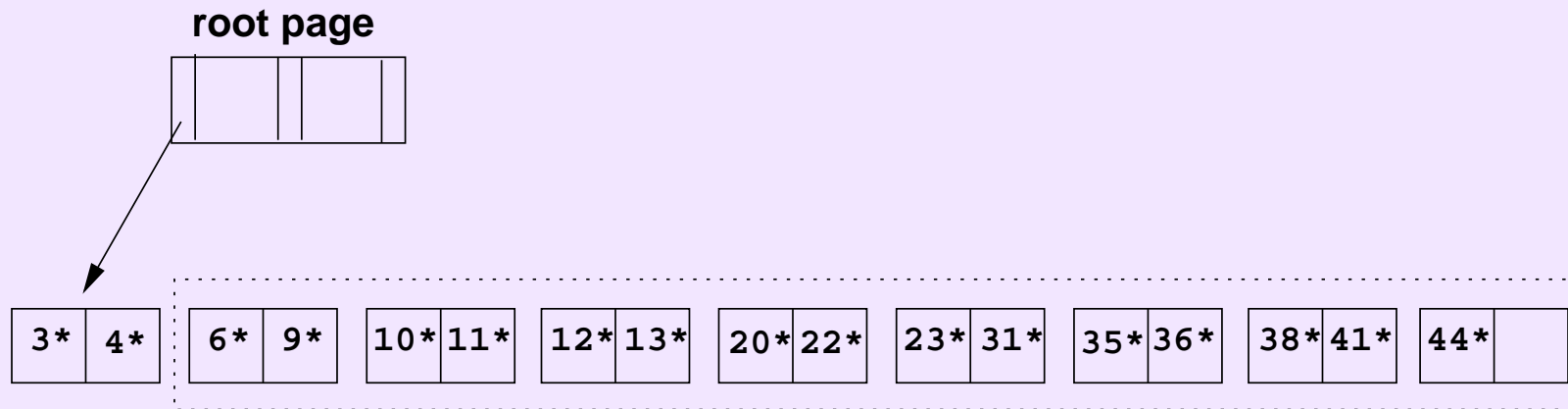
**Note:** For index variants **B** or **C**, this does *not* imply to sort the data file itself. (For variant **A**, we effectively create a clustered index.)

- 2 Allocate an empty index root page and let its  $p_0$  page pointer point to the first page of sorted  $k^*$  entries.

# B<sup>+</sup>-tree: Bulk Loading



Example (State of bulk load after step 2, order of B<sup>+</sup>-tree  $d = 1$ )



(Index leaf pages not yet in B<sup>+</sup>-tree are framed .)

## Bulk loading continued

Can you anticipate how the bulk loading process will proceed from this point?

### Binary Search

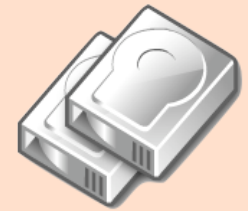
### ISAM

- Multi-Level ISAM
- Too Static?
- Search Efficiency

### B<sup>+</sup>-trees

- Search
- Insert
- Redistribution
- Delete
- Duplicates
- Key Compression
- Bulk Loading
- Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Bulk Loading



## Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression

### Bulk Loading

Partitioned B<sup>+</sup>-trees

- We now use the fact that the  $k_*$  are **sorted**. Any insertion will thus hit the **right-most index node** (just above the leaf level).
- Use a specialized `bulk_insert(·)` procedure that avoids B<sup>+</sup>-tree root-to-leaf traversals altogether:

## B<sup>+</sup>-tree bulk loading algorithm (continued)

- 3 For each leaf level page  $p$ , insert the index entry

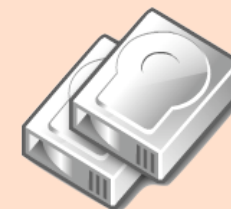
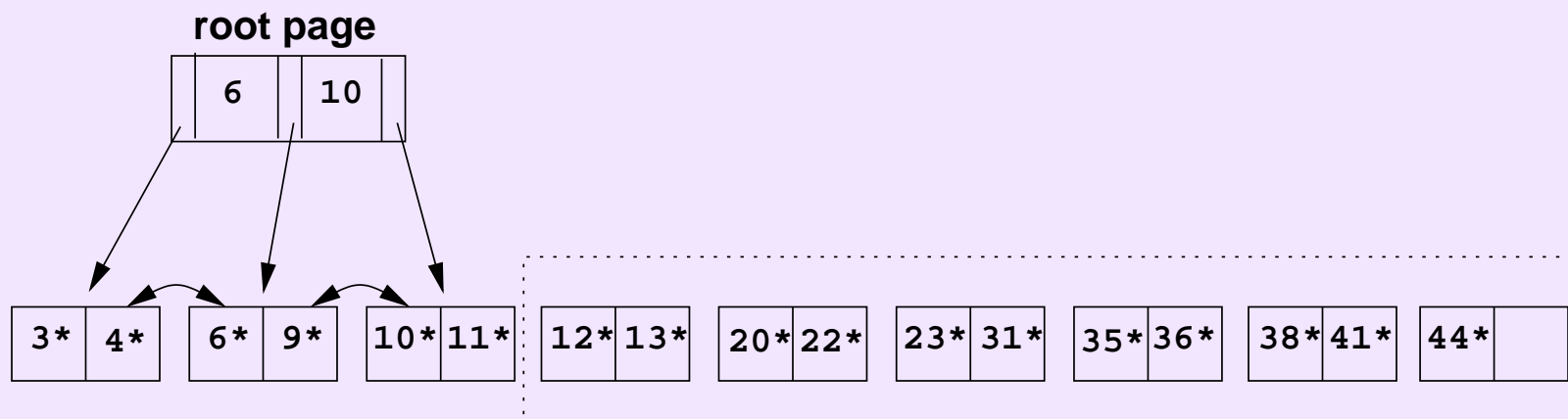
$\langle \text{minimum key on } p, \text{ pointer to } p \rangle$

into the right-most index node just above the leaf level.

The right-most node is filled **left-to-right**. Splits occur only on the **right-most path** from the leaf level up to the root.

# B<sup>+</sup>-tree: Bulk Loading

## Example (Bulk load continued)



### Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

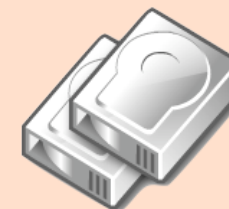
Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression

### Bulk Loading

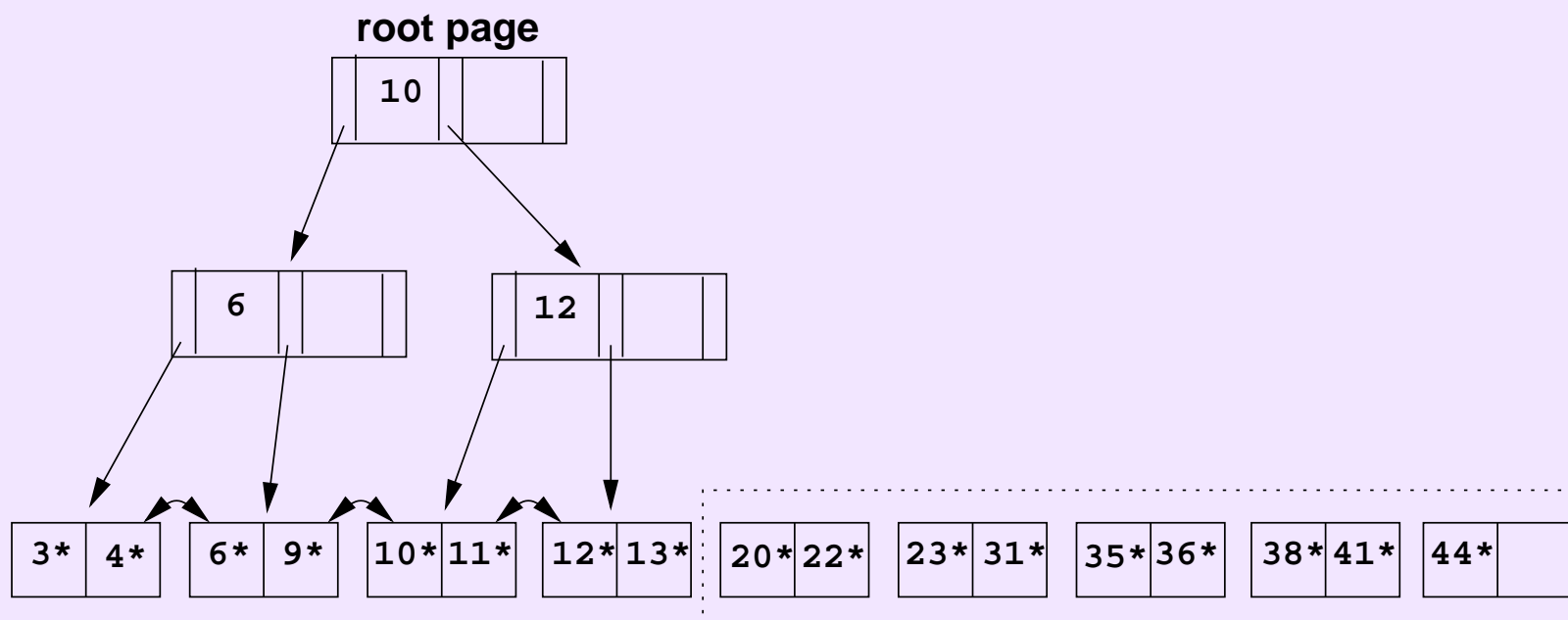
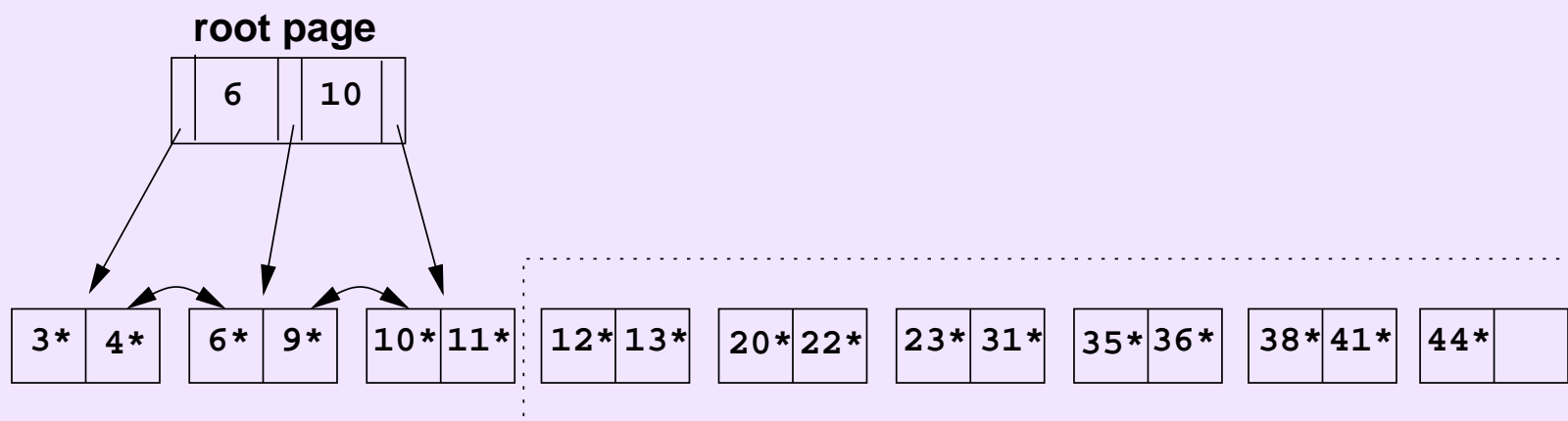
Partitioned B<sup>+</sup>-trees



# B<sup>+</sup>-tree: Bulk Loading



## Example (Bulk load continued)



### Binary Search

### ISAM

- Multi-Level ISAM
- Too Static?
- Search Efficiency

### B<sup>+</sup>-trees

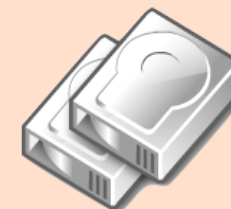
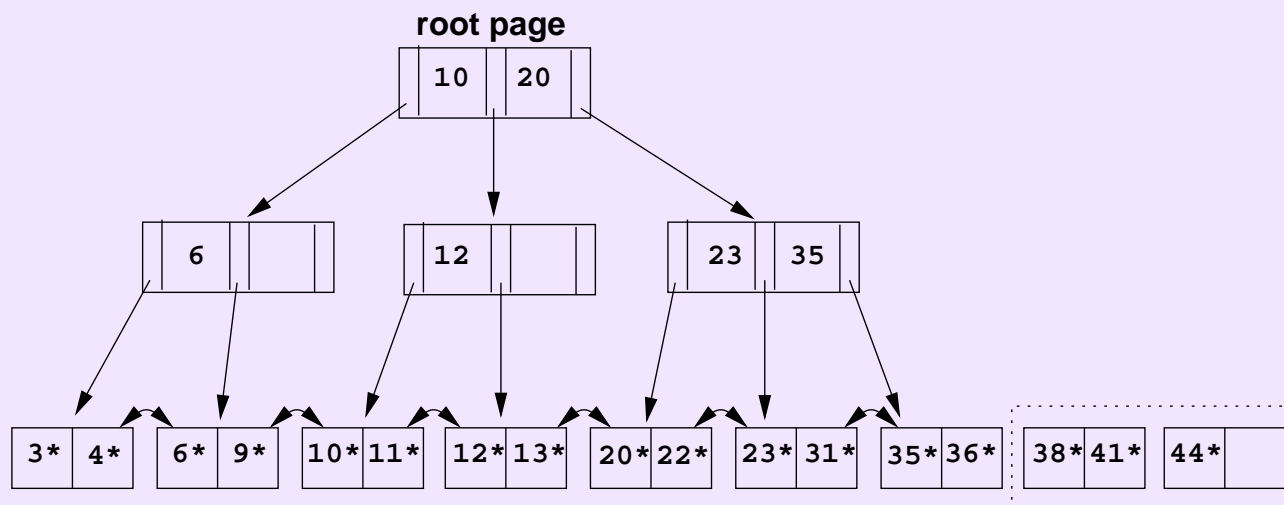
- Search
- Insert
- Redistribution
- Delete
- Duplicates
- Key Compression

### Bulk Loading

- Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Bulk Loading

## Example (Bulk load continued)



### Binary Search

### ISAM

- Multi-Level ISAM
- Too Static?
- Search Efficiency

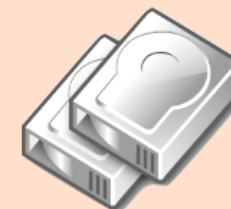
### B<sup>+</sup>-trees

- Search
- Insert
- Redistribution
- Delete
- Duplicates
- Key Compression

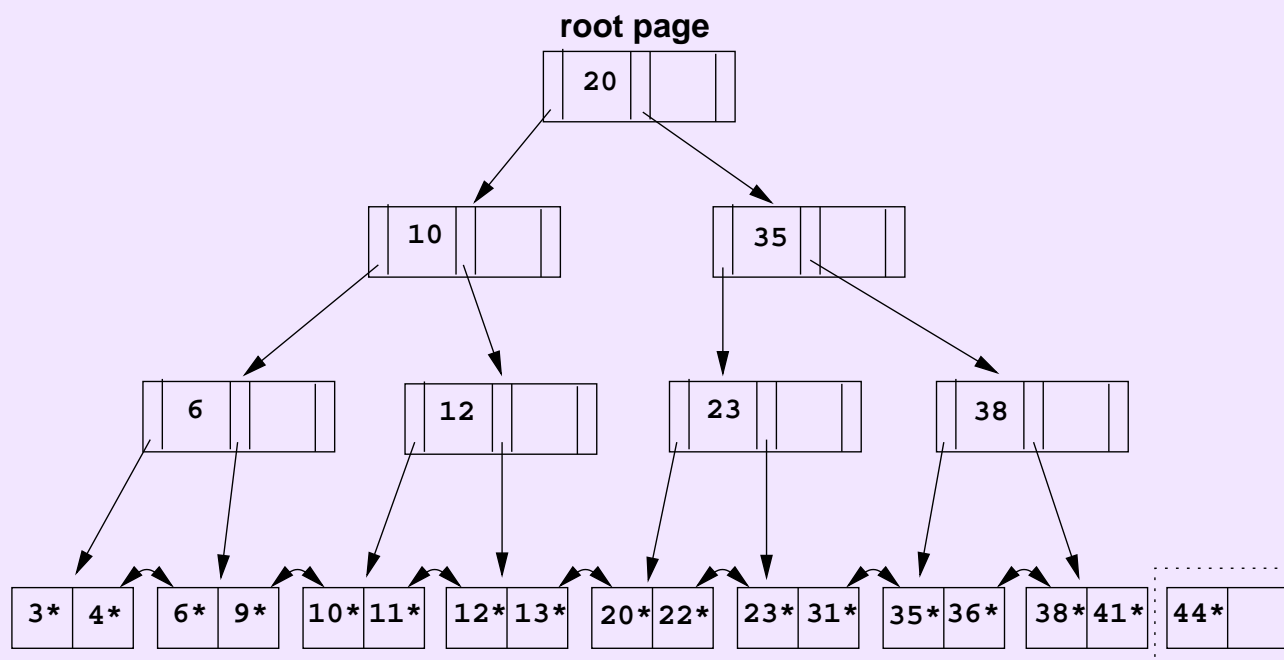
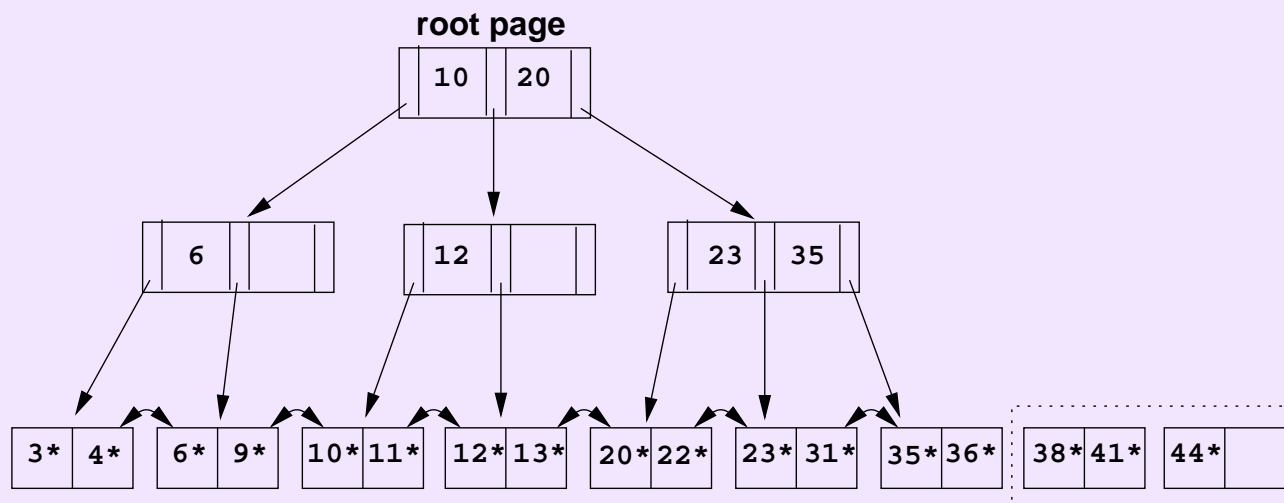
### Bulk Loading

- Partitioned B<sup>+</sup>-trees

# B<sup>+</sup>-tree: Bulk Loading



## Example (Bulk load continued)



### Binary Search

### ISAM

- Multi-Level ISAM
- Too Static?
- Search Efficiency

### B<sup>+</sup>-trees

- Search
- Insert
- Redistribution
- Delete
- Duplicates
- Key Compression

### Bulk Loading

- Partitioned B<sup>+</sup>-trees

# Composite Keys

B<sup>+</sup>-trees can (in theory<sup>3</sup>) be used to index everything with a defined **total order**, *e.g.*:

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

Possible in most SQL DDL dialects:

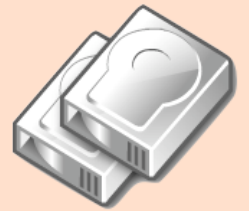
## Example (Create an index using a composite (concatenated) key)

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

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

- Leading index attributes effectively **partition** the resulting B<sup>+</sup>-tree.

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

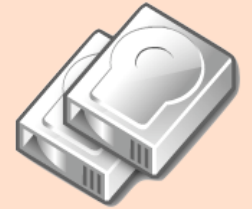


Multi-Level ISAM  
Too Static?  
Search Efficiency

Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees

<sup>3</sup>Some implementations won't allow you to index, *e.g.*, large character fields.

# Partitioned B-trees



**Example (Index with composite key, low-selectivity key prefix)**

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



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

## Binary Search

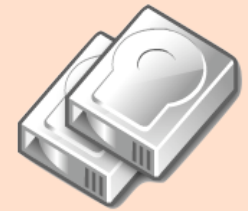
### ISAM

- Multi-Level ISAM
- Too Static?
- Search Efficiency

### B<sup>+</sup>-trees

- Search
- Insert
- Redistribution
- Delete
- Duplicates
- Key Compression
- Bulk Loading
- Partitioned B<sup>+</sup>-trees

# Partitioned B-trees



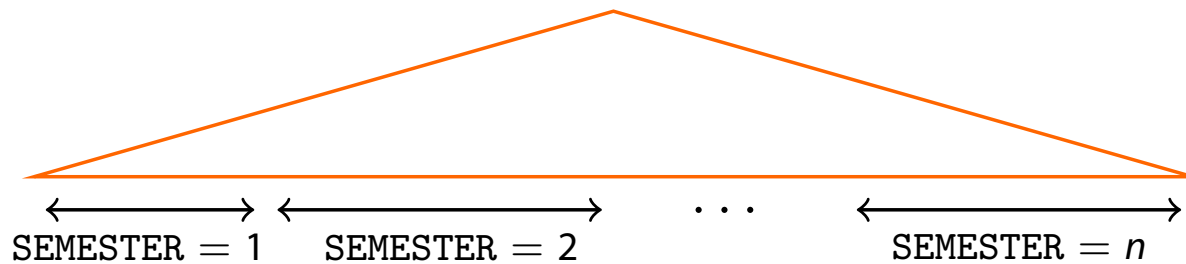
## Example (Index with composite key, low-selectivity key prefix)

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



### What types of queries could this index support?

The resulting B<sup>+</sup>-tree is going to look like this:



It can efficiently answer queries with, *e.g.*,

- **equality predicates** on SEMESTER **and** ZIPCODE,
- **equality** on SEMESTER and **range predicate** on ZIPCODE, or
- a **range predicate** on SEMESTER **only**.

### Binary Search

### ISAM

Multi-Level ISAM  
Too Static?  
Search Efficiency

### B<sup>+</sup>-trees

Search  
Insert  
Redistribution  
Delete  
Duplicates  
Key Compression  
Bulk Loading  
Partitioned B<sup>+</sup>-trees