

Systems Infrastructure for Data Science

Web Science Group

Uni Freiburg

WS 2013/14

Lecture II: Indexing

Indexing

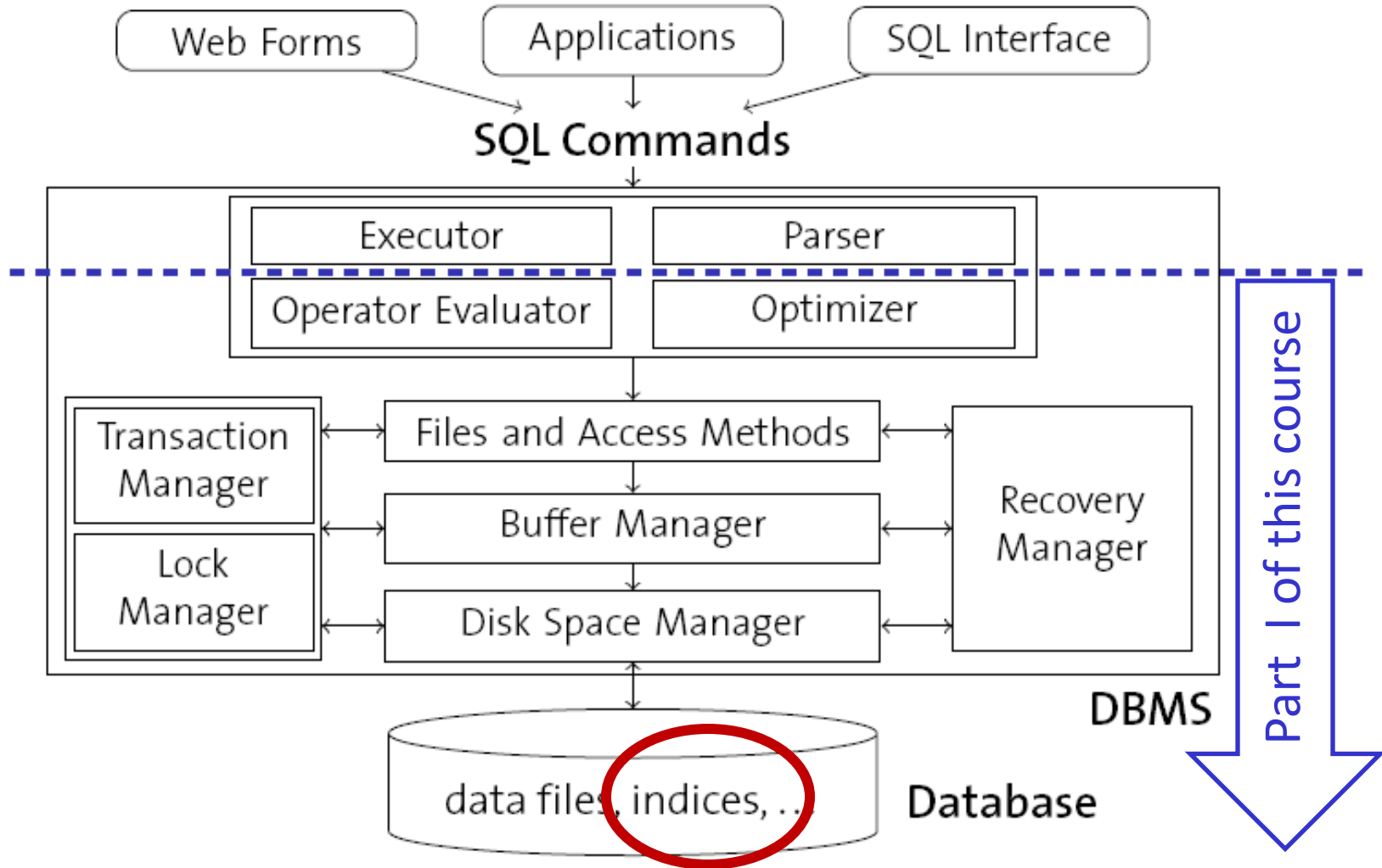


Figure inspired by Ramakrishnan/Gehrke: "Database Management Systems", McGraw-Hill 2003.

Database File Organization and Indexing

- Remember: Database tables are implemented as files of records:
 - A **file** consists of one or more **pages**.
 - Each page contains one or more **records**.
 - Each record corresponds to one **tuple** in a table.
- **File organization:** Method of arranging the records in a file when the file is stored on disk.
- **Indexing:** Building data structures that organize data records on disk in (multiple) ways to optimize search and retrieval operations on them.

File Organization

- Given a query such as the following:

```
SELECT *  
  FROM CUSTOMERS  
 WHERE ZIPCODE BETWEEN 8800 AND 8999
```

- How should we organize the storage of our data files on disk such that we can evaluate this query efficiently?

Heap Files?

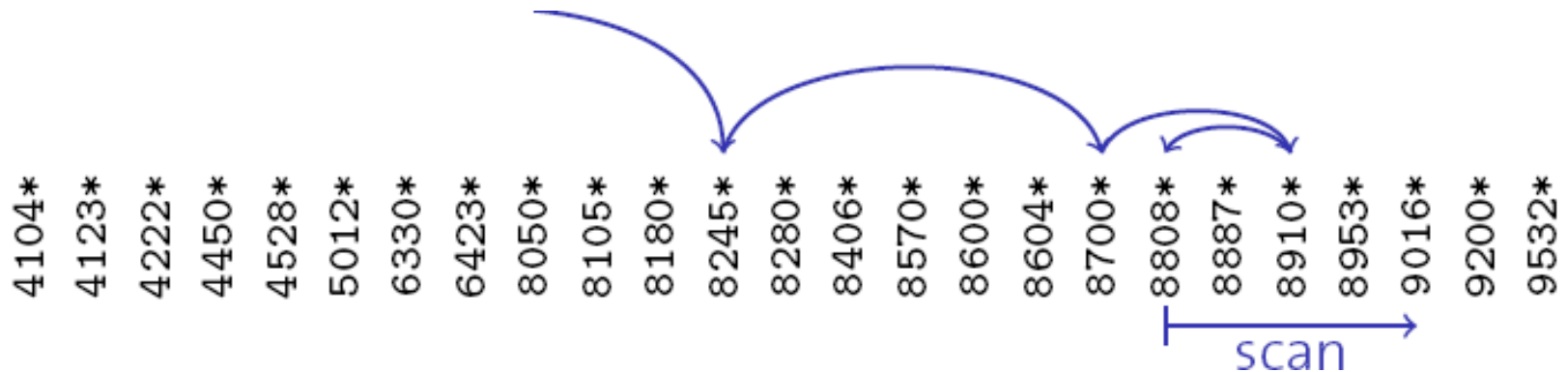
```
SELECT *  
  FROM CUSTOMERS  
 WHERE ZIPCODE BETWEEN 8800 AND 8999
```

- A heap file stores records in **no particular order**.
- Therefore, CUSTOMER table consists of records that are randomly ordered in terms of their ZIPCODE.
- **The entire file must be scanned**, because the qualifying records could appear anywhere in the file and we don't know in advance how many such records exist.

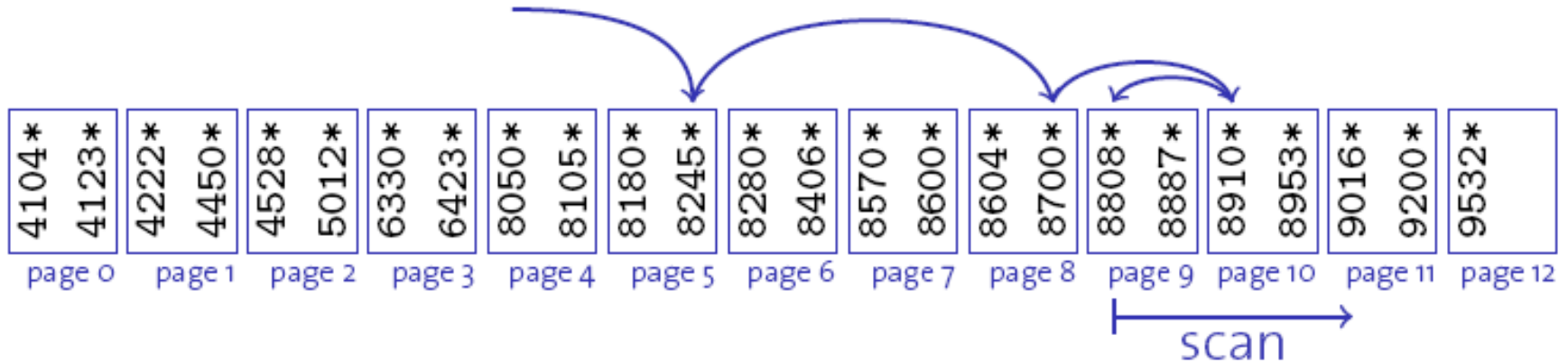
Sorted Files?

```
SELECT *  
FROM CUSTOMERS  
WHERE ZIPCODE BETWEEN 8800 AND 8999
```

- **Sort** the CUSTOMERS table in ZIPCODE order.
- Then use **binary search** to find the first qualifying record, and **scan** further as long as ZIPCODE < 8999.



Are Sorted Files good enough?

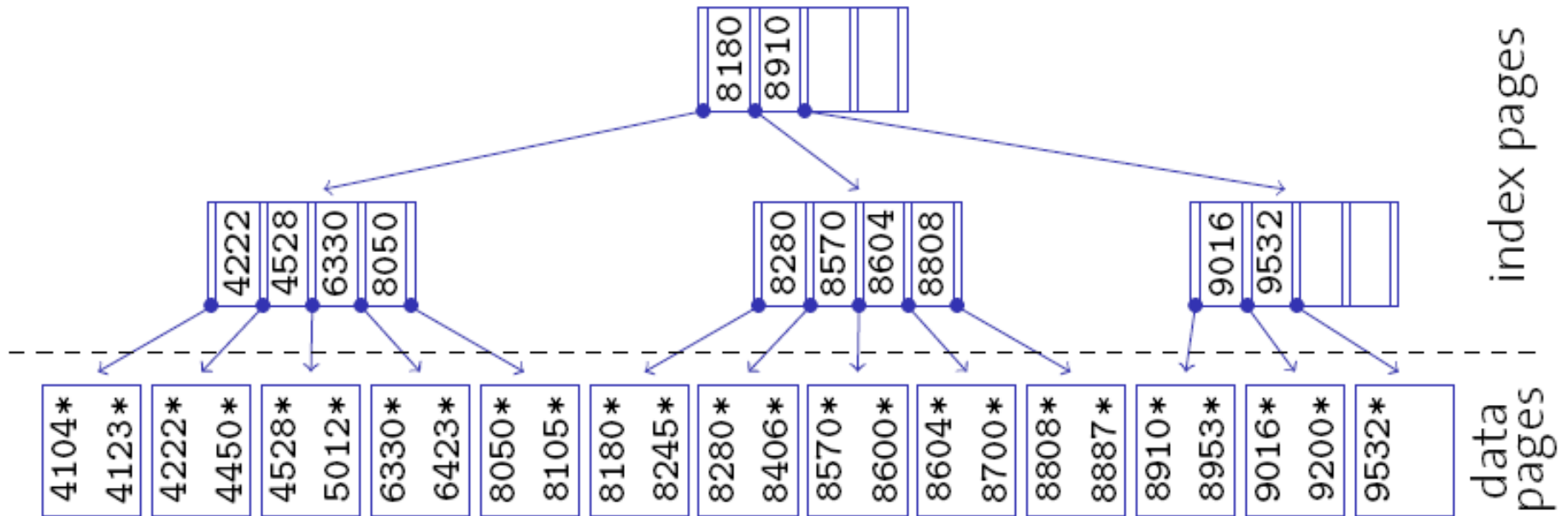


- ✓ Scan phase: We get **sequential access** during this phase.
- ✗ Search phase: We need to read **$\log_2 N$** records during this phase (N: total number of records in the CUSTOMER table).
 - We need to fetch as many pages as are required to access these records.
 - Binary search involves unpredictable jumps that makes prefetching difficult.
- ✗ What about insertions and deletions?

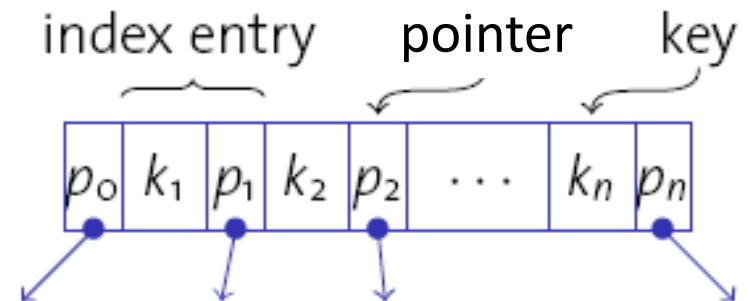
Tree-based Indexing

- Can we reduce the number of pages fetched during the **search phase**?
- Tree-based indexing:
 - Arrange the data entries in sorted order by search key value (e.g., ZIPCODE).
 - Add a hierarchical search data structure on top that directs searches for given key values to the correct page of data entries.
 - Since the index data structure is much smaller than the data file itself, the binary search is expected to fetch a smaller number of pages.
 - Two alternative approaches: **ISAM** and **B⁺-tree**.

ISAM: Indexed Sequential Access Method

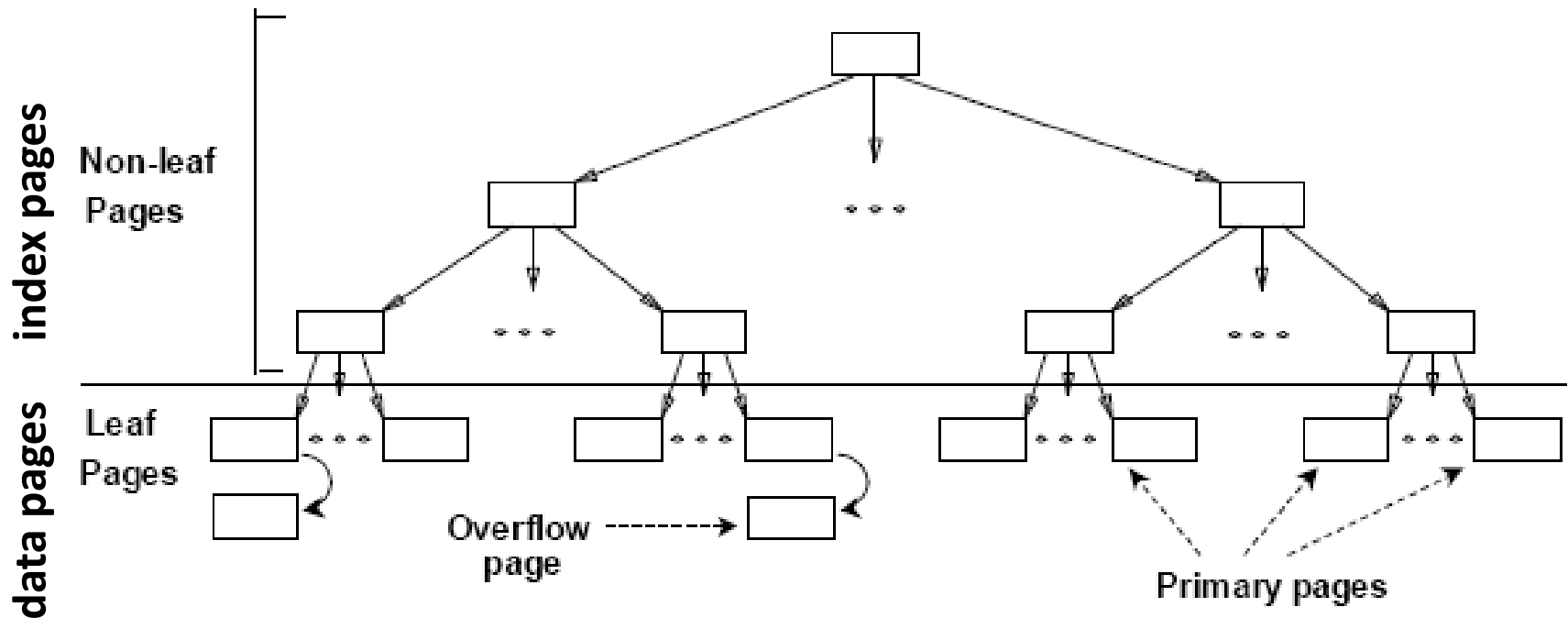


- All nodes are of the size of a page.
 - hundreds of entries per page
 - large fan-out, low depth
- Search cost $\sim \log_{\text{fan-out}} N$
- Key k_i serves as a “separator” for the pages pointed to by p_{i-1} and p_i .



ISAM Index Structure

- **Index pages** stored at non-leaf nodes
- **Data pages** stored at leaf nodes
 - **Primary** data pages & **Overflow** data pages

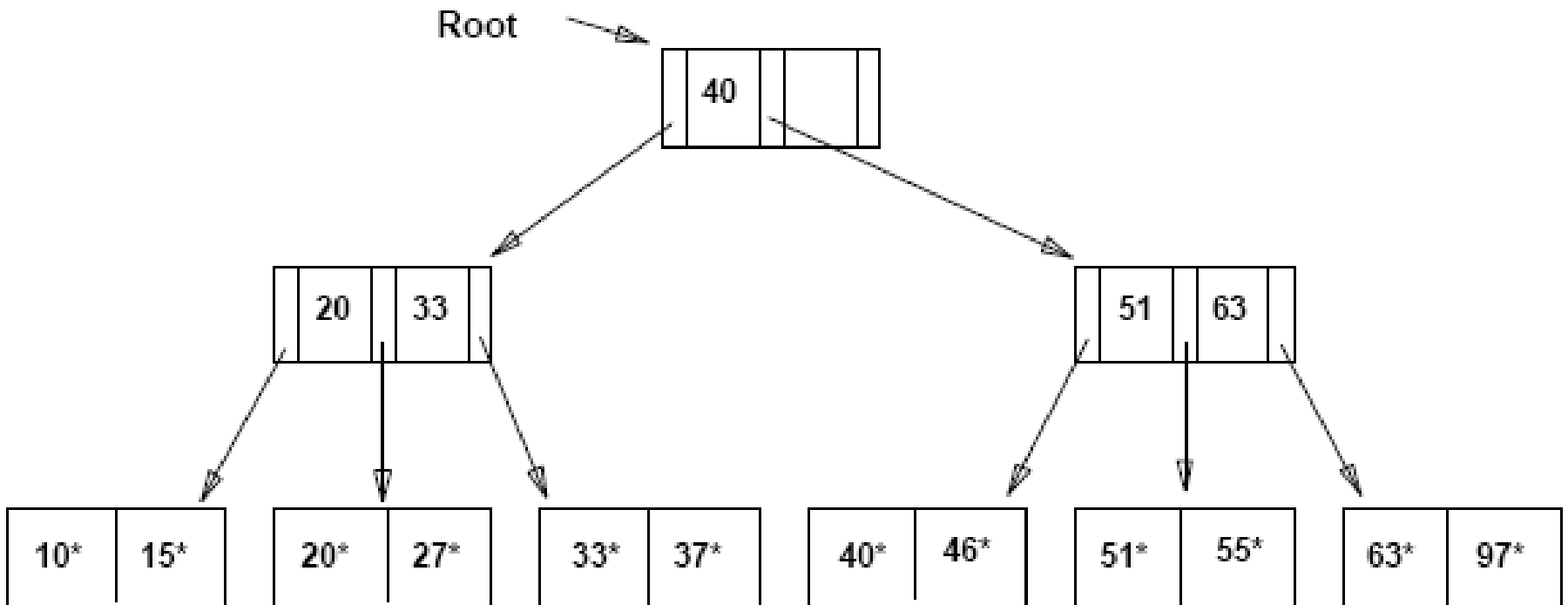


Updates on ISAM Index Structure

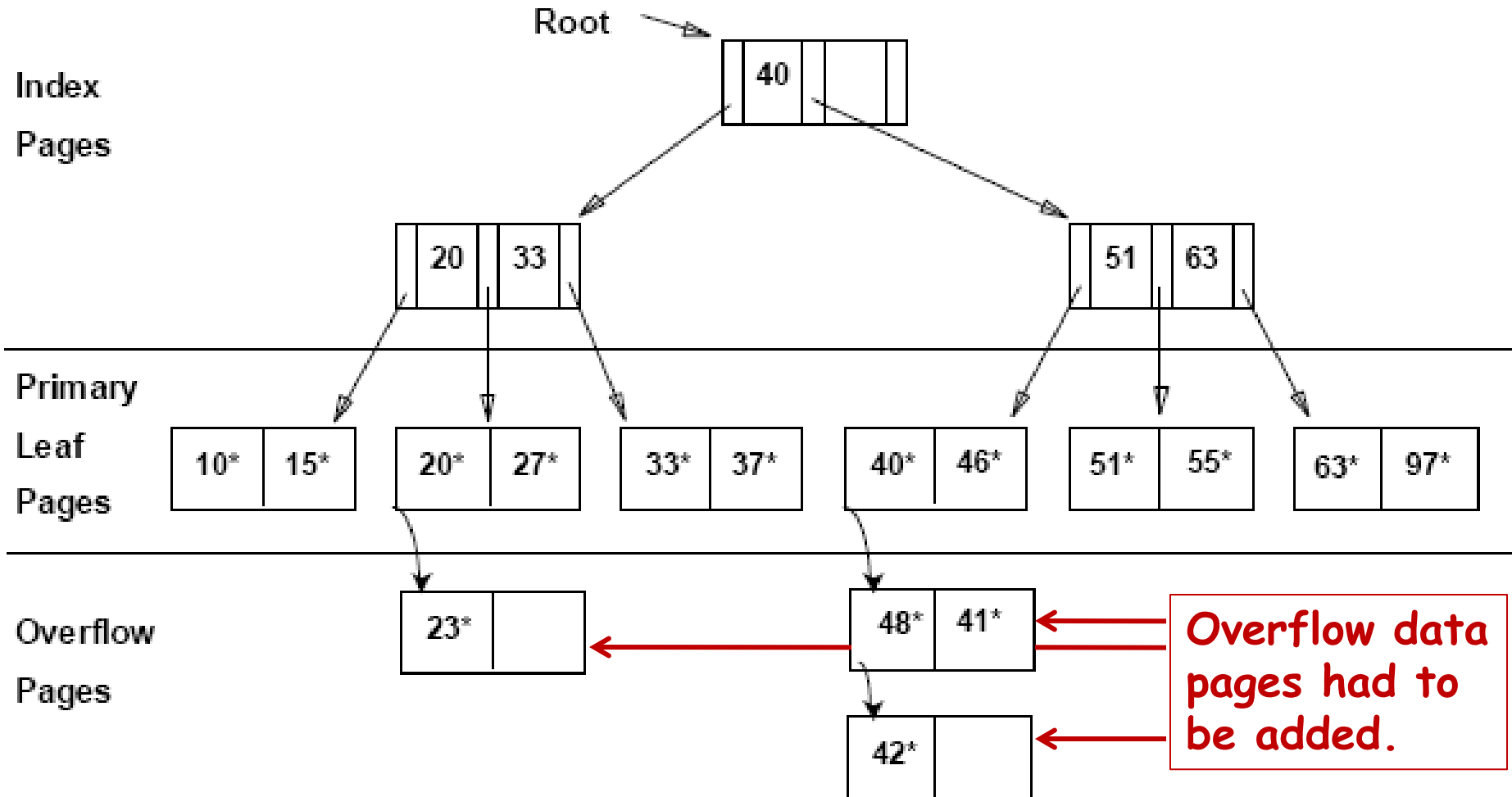
- ISAM index structure is inherently **static**.
 - **Deletion** is not a big problem:
 - Simply remove the record from the corresponding data page.
 - If the removal makes an overflow data page empty, remove that overflow data page.
 - If the removal makes a primary data page empty, keep it as a placeholder for future insertions.
 - Don't move records from overflow data pages to primary data pages even if the removal creates space for doing so.
 - **Insertion** requires more effort:
 - If there is space in the corresponding primary data page, insert the record there.
 - Otherwise, an **overflow data page** needs to be added.
 - **Note that the overflow pages will violate the sequential order.**
 - ISAM indexes degrade after some time.

ISAM Example

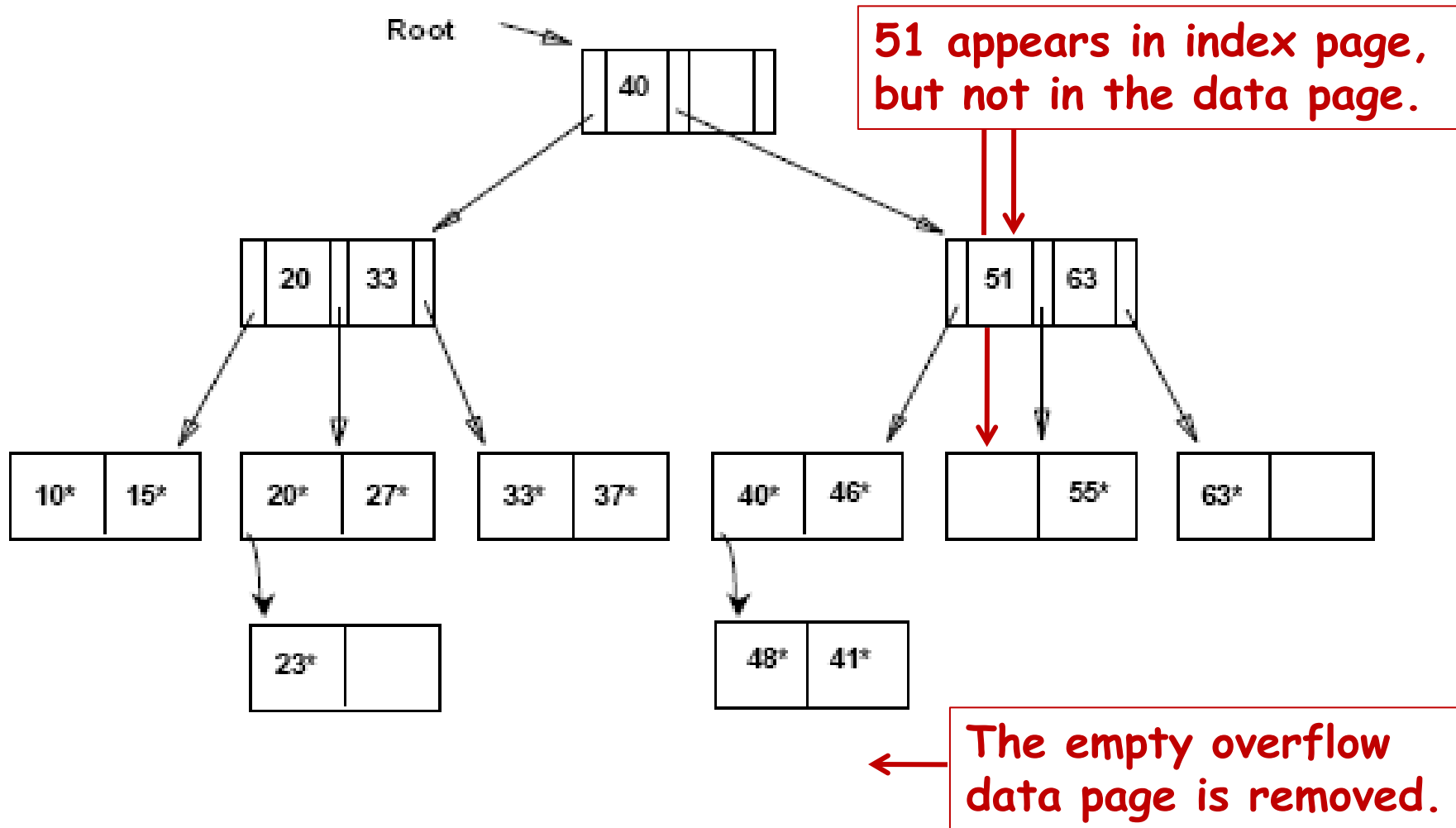
- Assume: Each node can hold two entries.



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



... Then Deleting 42*, 51*, 97*



ISAM: Overflow Pages & Locking

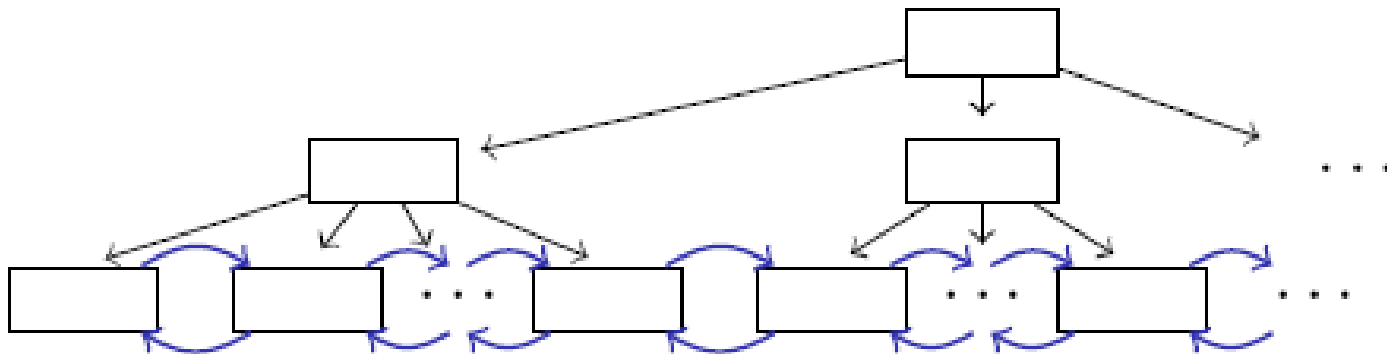
- The non-leaf pages that hold the index data are static; updates affect only the leaf pages.
 - May lead to **long overflow chains**.
- Leave some **free space** during index creation.
 - Typically ~ 20% of each page is left free.
- Since ISAM indexes are static, pages **need not be locked** during index access.
 - Locking can be a serious bottleneck in dynamic tree indexes (particularly near the root node).
- ISAM may be the index of choice for relatively static data.

B⁺-trees: A Dynamic Index Structure

- The **B⁺-tree** is derived from the ISAM index, but is **fully dynamic** with respect to updates.
 - **No overflow chains**; B⁺-trees remain **balanced** at all times.
 - Gracefully adjusts to insertions and deletions.
 - **Minimum occupancy** for all B⁺-tree nodes (except the root): 50% (typically: 67 %).
 - Original version:
 - **B-tree**: R. Bayer and E. M. McCreight, “Organization and Maintenance of Large Ordered Indexes”, Acta Informatica, vol. 1, no. 3, September 1972.

B⁺-trees: Basics

- B⁺-trees look like ISAM indexes, where
 - leaf nodes are, generally, **not in sequential order** on disk
 - leaves are typically connected to form a **doubly-linked list**
 - leaves may contain **actual data** (like the ISAM index) or just **references** to data pages (e.g., record ids (rids))
 - We will assume the latter case, since it is the more common one.
 - each B⁺-tree node contains between d and $2d$ entries (d is the **order** of the B⁺-tree; the root is the only exception).



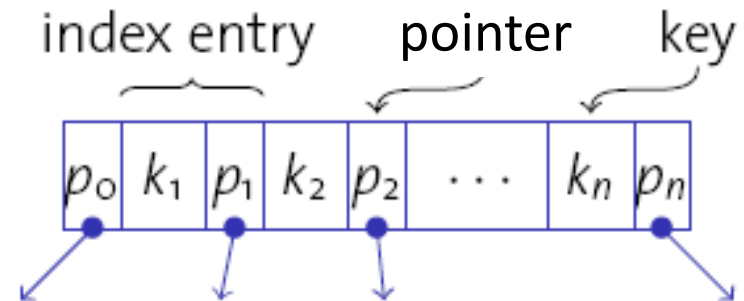
Searching a B⁺-tree

```
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 < k0
6     | | return tree_search (k, p0);
7   | case ki ≤ k < ki+1
8     | | return tree_search (k, pi);
9   | case k2d ≤ k
10  | | return tree_search (k, p2d);
```

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

- Node page layout:

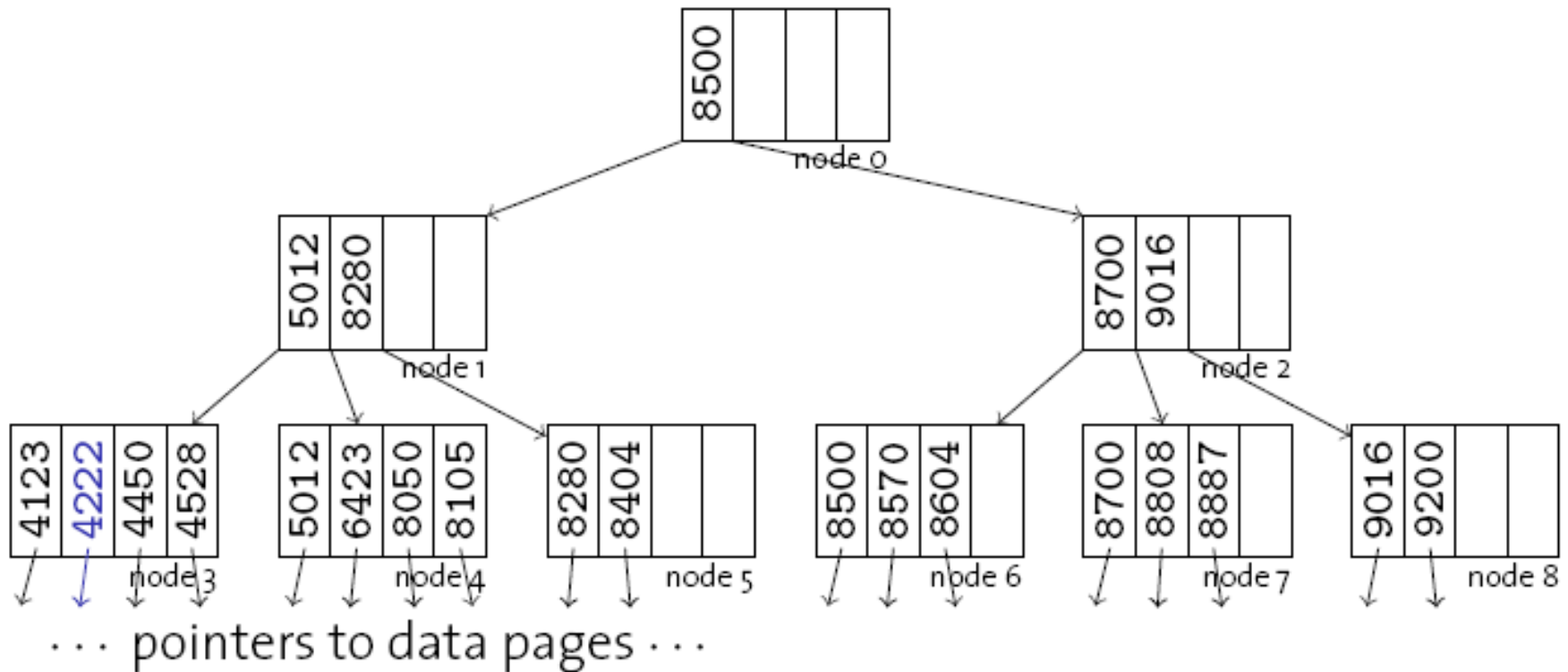


Insertion to a B⁺-tree: Overview

- The B⁺-tree needs to remain **balanced** after every update (i.e., every root-to-leaf path must be of the same length).
 - We cannot create overflow pages.
- Sketch of the insertion procedure for entry $\langle k, p \rangle$ (key value k pointing to data page p):
 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** $\langle k, p \rangle$ 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 .

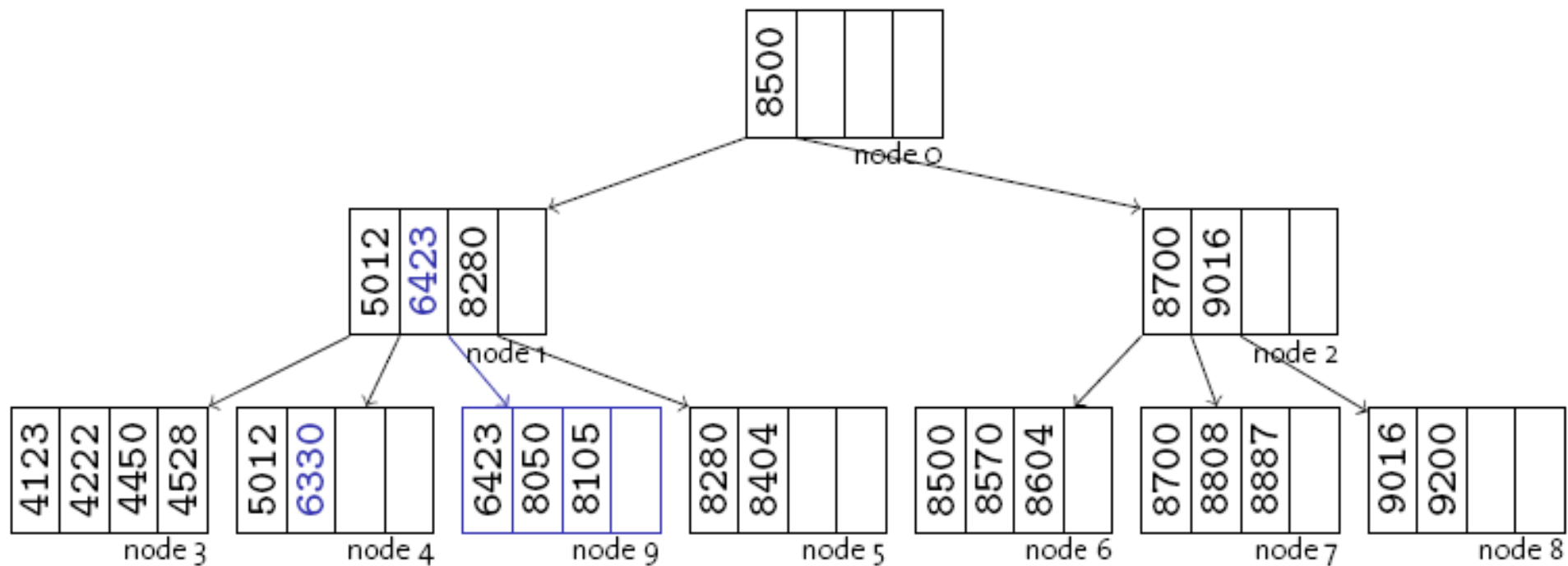
Splitting happens recursively and may eventually lead to a split of the root node (increasing the height of the tree).

Insertion to a B⁺-tree: Example

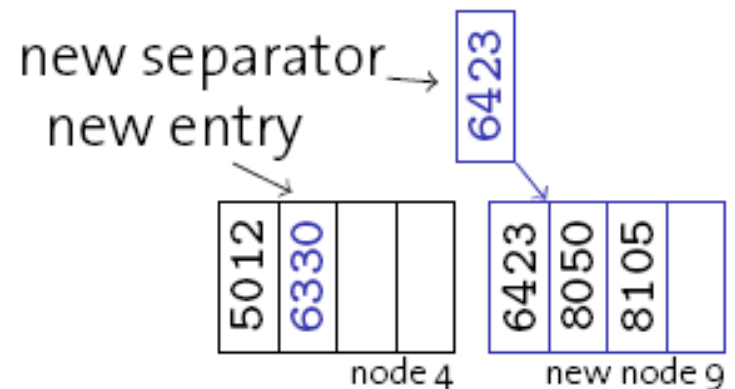


- Insert new entry with key **4222**.
 - Enough space in node 3, simply insert without split.
 - Keep entries **sorted within nodes**.

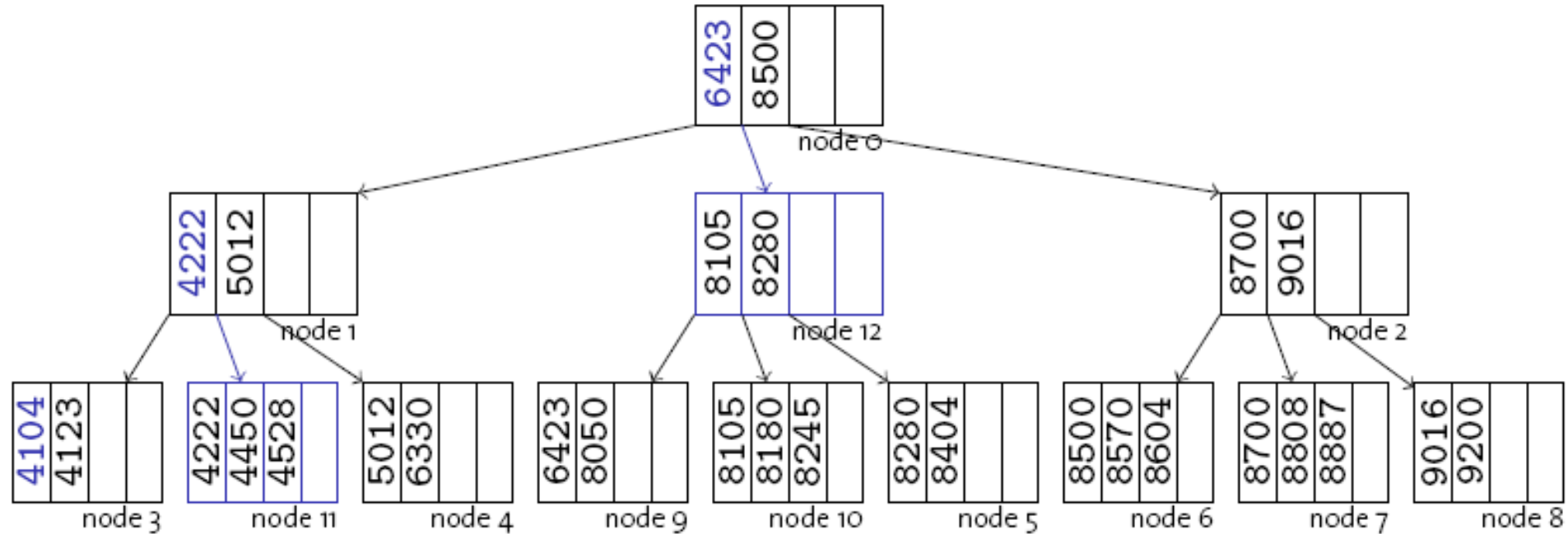
Insertion to a B⁺-tree: Example



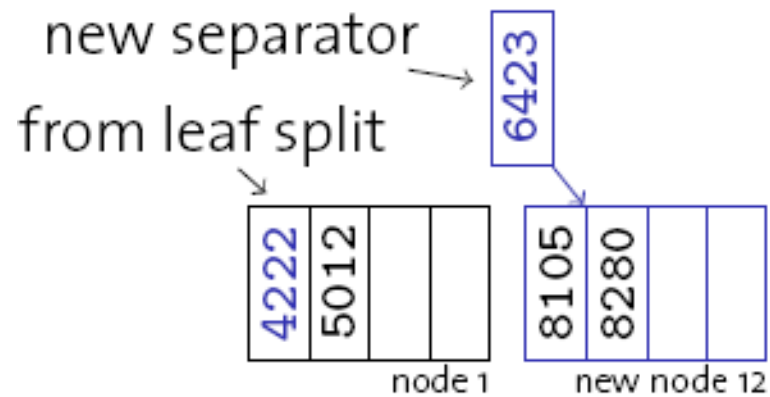
- Insert key **6330**.
 - Must **split** node 4.
 - **New separator** goes into node 1 (including pointer to new page).



Insertion to a B⁺-tree: Example



- After **8180, 8245**, insert key **4104**.
 - Must **split** node 3.
 - Node 1 **overflows** => split it!
 - **New separator** goes into root.
- Note: Unlike during leaf split, separator key does not remain in inner node.



Insertion to a B⁺-tree: 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.**

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  $k < k_0$ 
7        $\langle sep, ptr \rangle \leftarrow$  tree_insert (k, rid, p0);
8     case  $k_i \leq k < k_{i+1}$ 
9        $\langle sep, ptr \rangle \leftarrow$  tree_insert (k, rid, pi);
10    case  $k_{2d} \leq k$ 
11       $\langle sep, ptr \rangle \leftarrow$  tree_insert (k, rid, p2d);
12    if sep is null then
13      return  $\langle$  null, null  $\rangle$ ;
14    else
15      return split (sep, ptr, node);
```

} see tree_search ()

```

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   take  $\{\langle k_1^+, p_1^+ \rangle, \dots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle\} :=$  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  $\langle k_{d+1}^+, p_{d+1}^+ \rangle, \dots, \langle k_{2d}^+, p_{2d}^+ \rangle$  to  $p$ ;
10  return  $\langle k_{d+1}^+, p \rangle$ ;           2d+1  2d+1

```

```

1 Function: split ( $k, ptr, node$ )
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   take  $\{\langle k_1^+, p_1^+ \rangle, \dots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle\} :=$  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  $\langle k_{d+1}^+, p_{d+1}^+ \rangle, \dots, \langle k_{2d}^+, p_{2d}^+ \rangle$  to  $p$ ;
10  set  $p_0^+ \leftarrow p_{d+1}^+$  in  $node$ ;           2d+1  2d+1
11  return  $\langle k_{d+1}^+, p \rangle$ ;

```

```

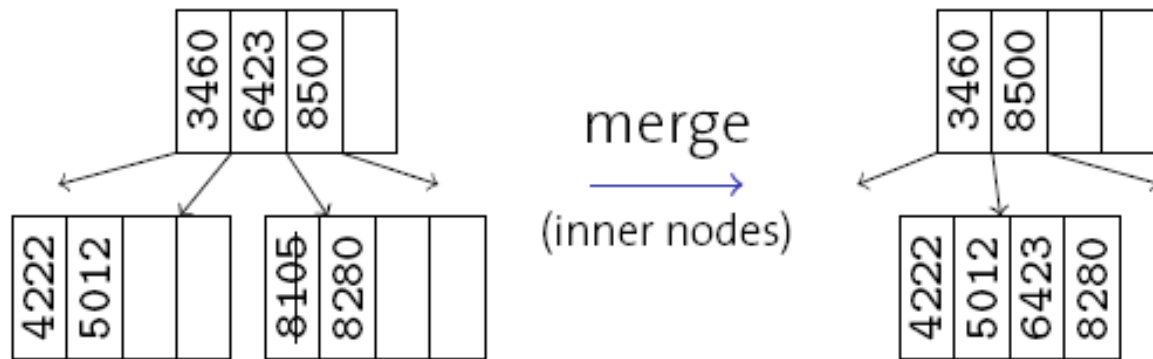
1 Function: insert (k, rid)
2  $\langle key, ptr \rangle \leftarrow$  tree_insert (k, rid, root);
3 if key is not null then
4     allocate new root page  $r$ ;
5     populate  $n$  with
6          $p_0 \leftarrow$  root;
7          $k_1 \leftarrow$  key;
8          $p_1 \leftarrow$  ptr;
9      $root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

```

- *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.

Deletion from a B⁺-tree

- If a node is sufficiently full (i.e., contains at least $d+1$ entries), we may simply remove the entry from the node.
 - Note: Afterwards, **inner nodes** may contain keys that no longer exist in the database. This is perfectly legal.
- Merge nodes in case of an underflow (i.e., “undo” a split):



- “Pull” separator (i.e., key 6423) into merged node.

Deletion from a B⁺-tree

- It is not that easy:



- Merging only works if two neighboring nodes were 50% full.
- Otherwise, we have to **re-distribute**:
 - “rotate” entry through parent

B⁺-trees in Real Systems

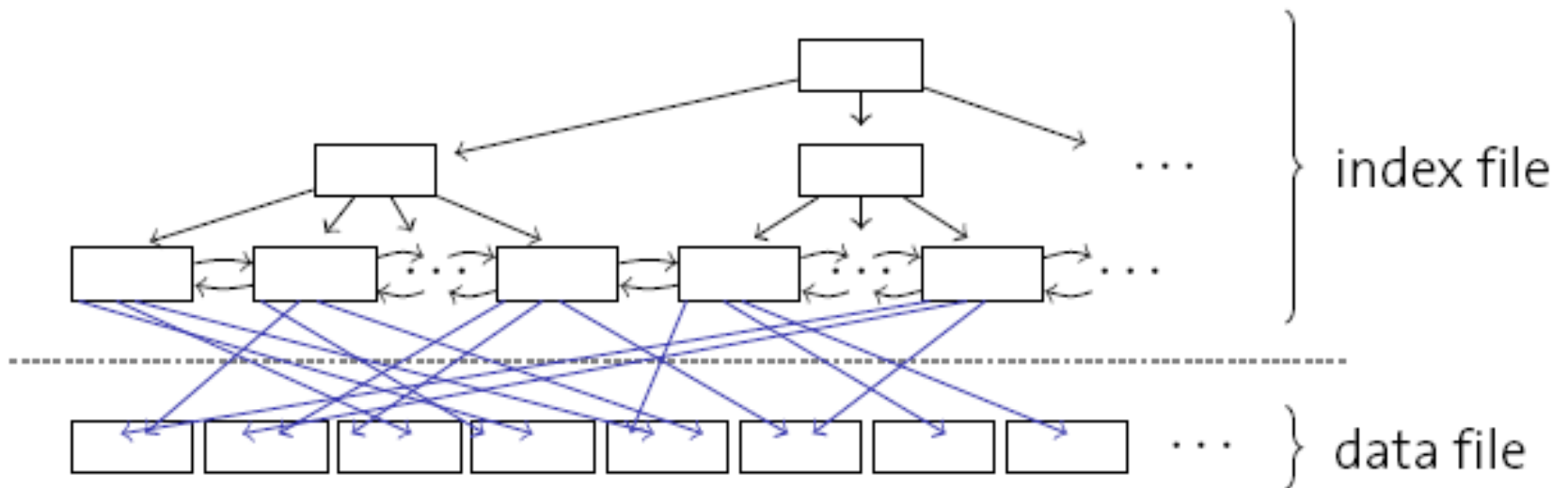
- Actual systems often avoid the cost of merging and/or redistribution, but relax the minimum occupancy rule.
- **Example: IBM DB2 UDB**
 - The “**MINPCTUSED**” parameter controls when the system should try a leaf node merge (“on-line index reorganization”).
 - This is particularly easy because of the pointers between adjacent leaf nodes.
 - Inner nodes are never merged (need to do a full table reorganization 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”).

What is stored inside the leaves?

- Basically there are three alternatives:
 1. The full data entry k^* . Such an index is inherently **clustered** (e.g., ISAM).
 2. A $\langle k, rid \rangle$ pair, where rid is the record id of the data entry.
 3. A $\langle k, \{rid_1, rid_2, \dots\} \rangle$ pair, where the items in the rid list rid_i are record ids of data entries with search key value k .
- 2 and 3 are reasons why we want record ids to be stable.
- 2 seems to be the most common one.

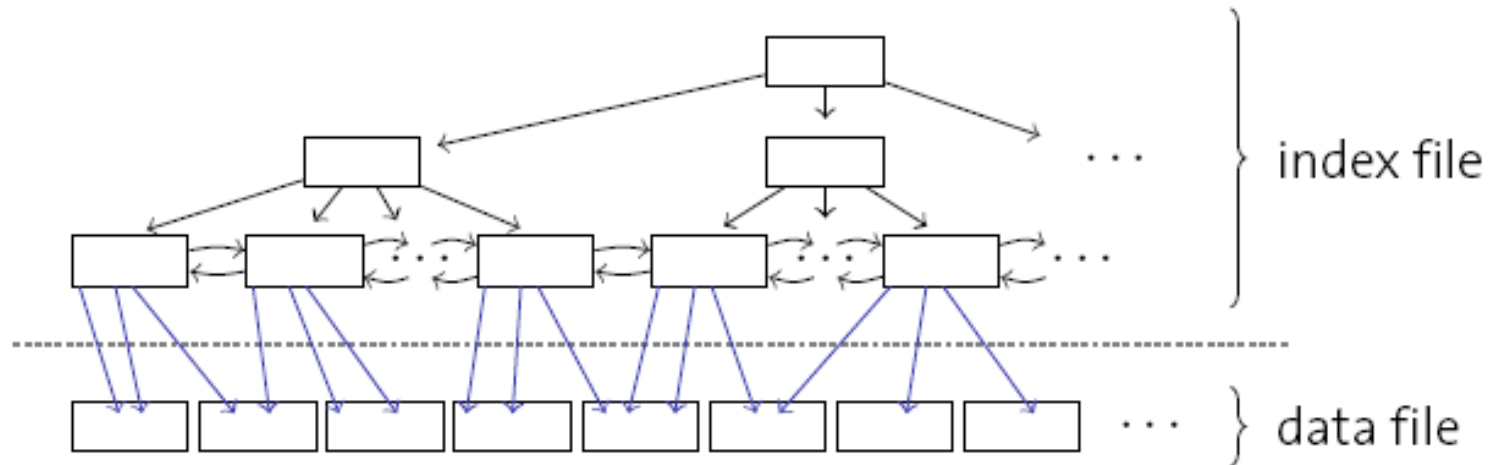
B⁺-trees and Sorting

- A typical situation according to alternative 2 looks as follows:



Clustered B⁺-trees

- If the data file was sorted, the scenario would look different:



- We call such an index a **clustered index**.
 - Scanning the index now leads to **sequential access**.
 - This is particularly good for **range queries**.

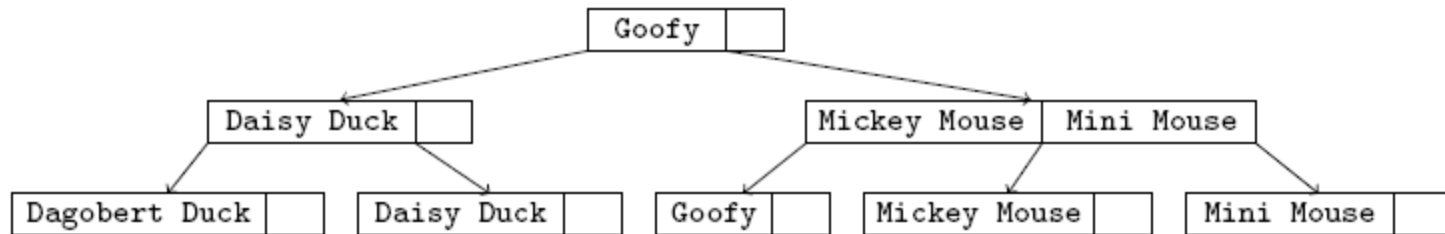
Index-organized Tables

- Alternative 1 is a special case of a clustered index.
 - index file = data file
 - Such a file is often called an **index-organized table**.
- Example: Oracle 8i

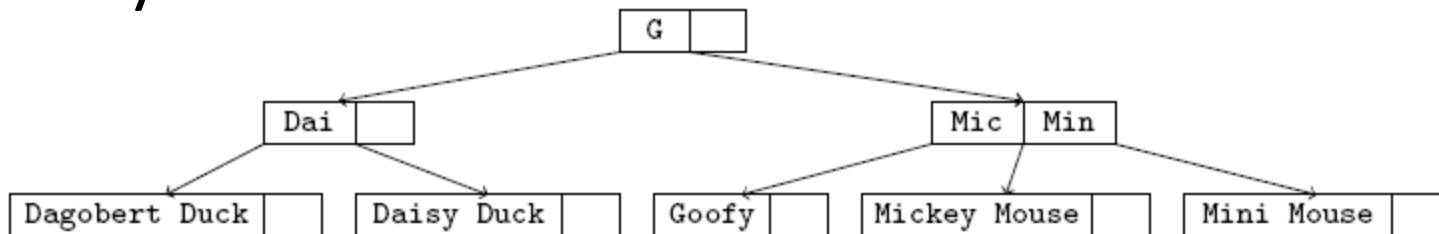
```
CREATE TABLE(...  
            ...,  
            PRIMARY KEY(...))  
ORGANIZATION INDEX;
```

Key Compression: Suffix Truncation

- B⁺-tree fan-out is proportional to the number of index entries per page, i.e., inversely proportional to the key size.
 - Reduce key size, particularly for **variable-length strings**.



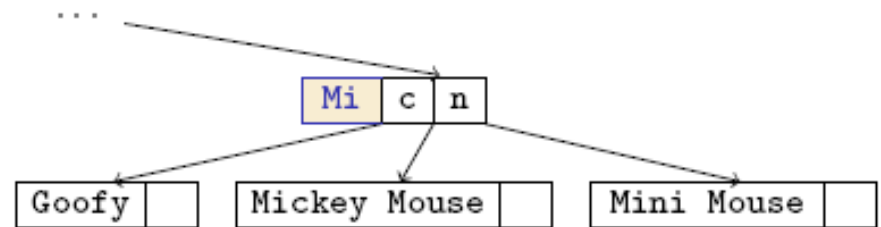
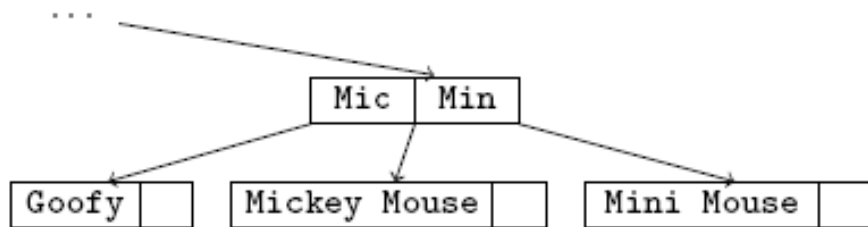
- **Suffix truncation:** Make separator keys only as long as necessary:



- Note that separators need not be actual data values.

Key Compression: Prefix Truncation

- Keys within a node often share a **common prefix**.



- **Prefix truncation:**

- Store common prefix **only once** (e.g., as “ k_0 ”).
- Keys have become highly discriminative now.

R. Bayer, K. Unterauer, “Prefix B-Trees”, ACM TODS 2(1), March 1977.

B. Bhattacharjee et al., “Efficient Index Compression in DB2 LUW”, VLDB’09.

Composite Keys

- B⁺-trees can in theory be used to index everything with a defined **total order** such as:
 - integers, strings, dates, etc., and
 - concatenations thereof (based on lexicographical order)
- Example: In most SQL dialects:

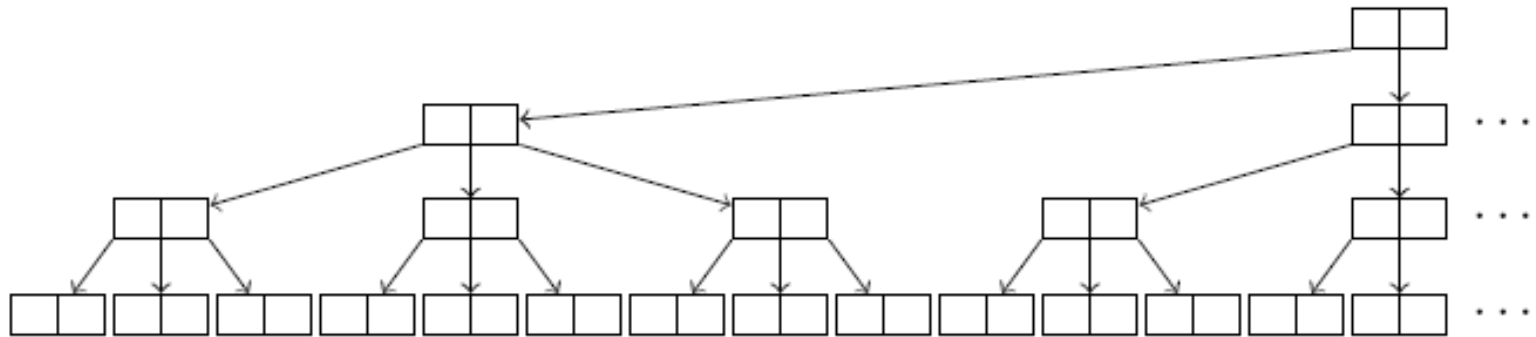
```
CREATE INDEX ON TABLE CUSTOMERS (LASTNAME, FIRSTNAME);
```

- 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’03.

Bulk-Loading B⁺-trees

- Building a B⁺-tree is particularly easy when the input is sorted.



- Build B⁺-tree **bottom-up** and **left-to-right**.
- Create a parent for every $2d+1$ un-parented nodes.
 - Actual implementations typically leave some space for future updates (e.g., DB2's “**PCTFREE**” parameter).

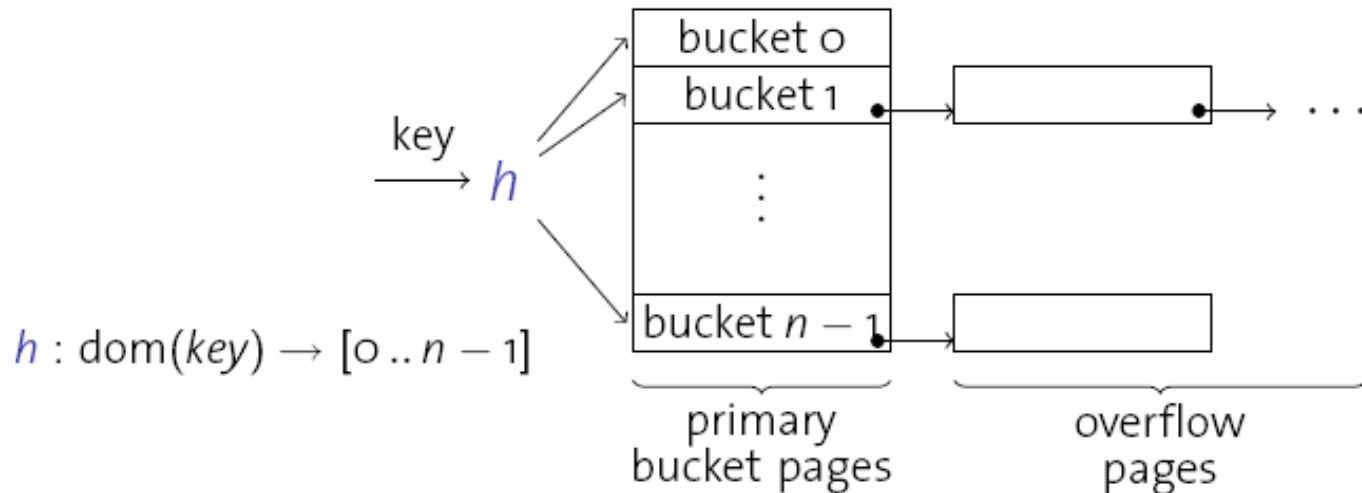
Stars, Pluses, ...

- In the foregoing we described the **B⁺-tree**.
- Bayer and McCreight originally proposed the **B-tree**:
 - Inner nodes contain data entries, too.
- There is also a **B*-tree**:
 - Keep non-root nodes at least 2/3 full (instead of 1/2).
 - Need to **redistribute on inserts** to achieve this
 - => Whenever two nodes are full, split them into three.
- Most people say “B-tree” and mean any of these variations. Real systems typically implement B⁺-trees.
- “B-trees” are also used outside the database domain, e.g., in modern **file systems** (ReiserFS, HFS, NTFS, ...).

Hash-based Indexing

- B⁺-trees are by far the predominant type of indices in databases. An alternative is hash-based indexing.
- Hash indexes can only be used to answer **equality selection queries** (not range selection queries).
- Like in tree-based indexing, static and dynamic hashing techniques exist; their trade-offs are similar to ISAM vs. B⁺-trees.

Hash-based Indexing



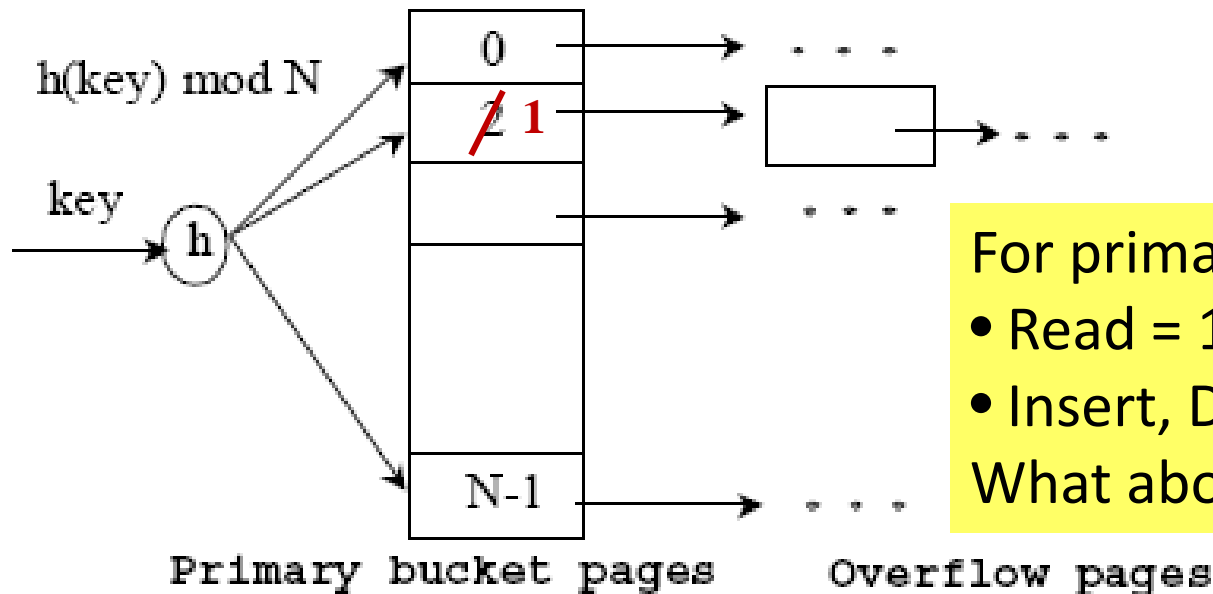
- Records in a file are grouped into **buckets**.
- A bucket consists of a **primary page** and possibly **overflow pages** linked in a chain.
- **Hash function:**
 - Given a the search key of a record, returns the corresponding bucket number that contains that record.
 - Then we search the record within that bucket.

Hash Function

- A good hash function distributes values in the domain of the search key uniformly over the collection of buckets.
- Given N buckets $0 .. N-1$, $h(value) = (a*value + b)$ works well.
 - $h(value) \bmod N$ gives the bucket number.
 - a and b are constants to be tuned.

Static Hashing

- Number of primary pages is fixed.
- Primary pages are allocated sequentially and are never de-allocated. Use overflow pages if need more pages.
- $h(k) \bmod N$ gives the bucket to which the data entry with search key k belongs. (N : number of buckets)



For primary pages:

- Read = 1 disk I/O
- Insert, Delete = 2 disk I/Os

What about the overflow pages?

Problems with Static Hashing

- Number of buckets n is fixed.
 - How to choose n ?
 - Many deletions => space is wasted
 - Many insertions => long overflow chains that degrade search performance
- Static hashing has similar problems and advantages as in ISAM.
- Rehashing solution:
 - Periodically rehash the whole file to restore the ideal (i.e., no overflow chains and 80% occupancy)
 - Takes long and makes the index unusable during rehashing.

Dynamic Hashing

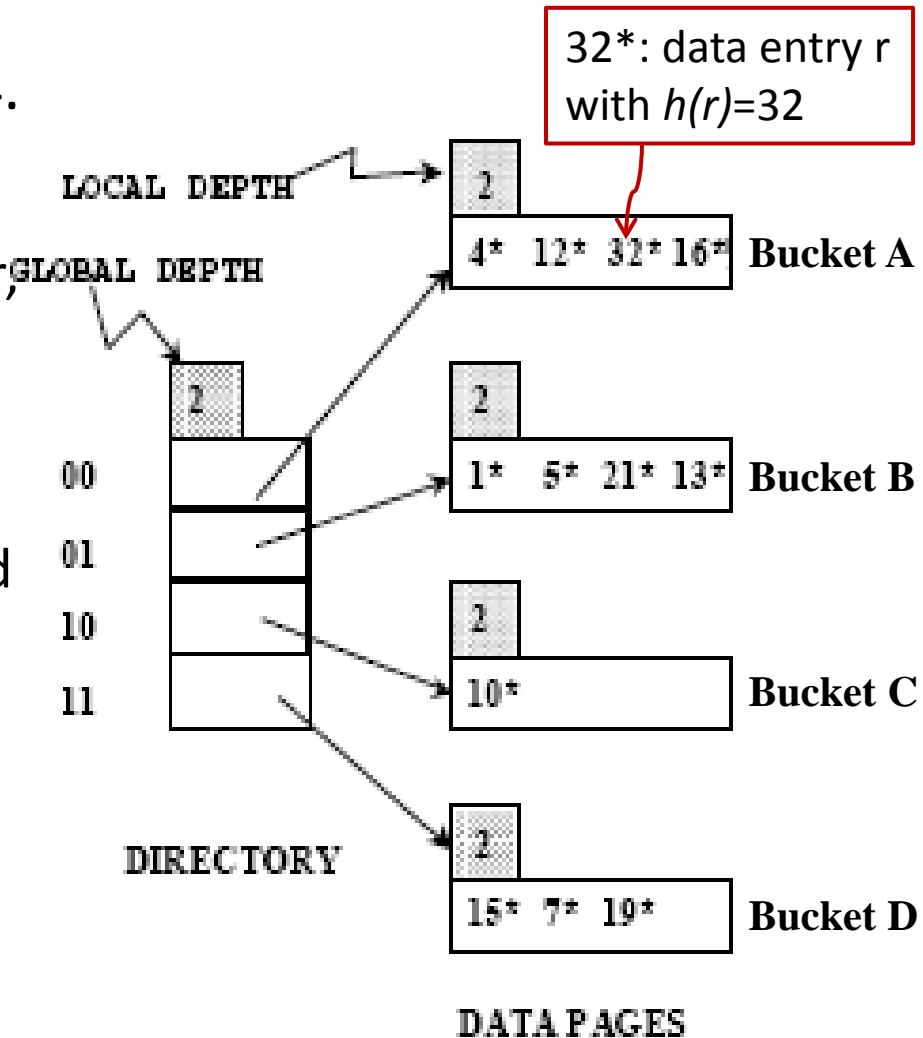
- To deal with the problems of static hashing, database systems use **dynamic hashing** techniques:
 - **Extendible hashing**
 - **Linear hashing**
- Note that: Few real systems support true hash indexes (such as PostgreSQL).
- More popular uses of hashing are:
 - support for B⁺-trees over hash values (e.g., SQL Server)
 - the use of hashing during query processing => **hash join**

Extendible Hashing: The Idea

- Overflows occur when bucket (primary page) becomes full. Why not re-organize the file by doubling the number of buckets?
 - Reading and writing all pages is expensive!
- **Idea:** Use a directory of pointers to buckets; double the number of buckets by doubling the directory and splitting just the bucket that overflowed.
 - Directory is much smaller than file, so doubling it is much cheaper. Only one page of data entries is split.
 - **No overflow pages!**
 - Trick lies in how the hash function is adjusted.

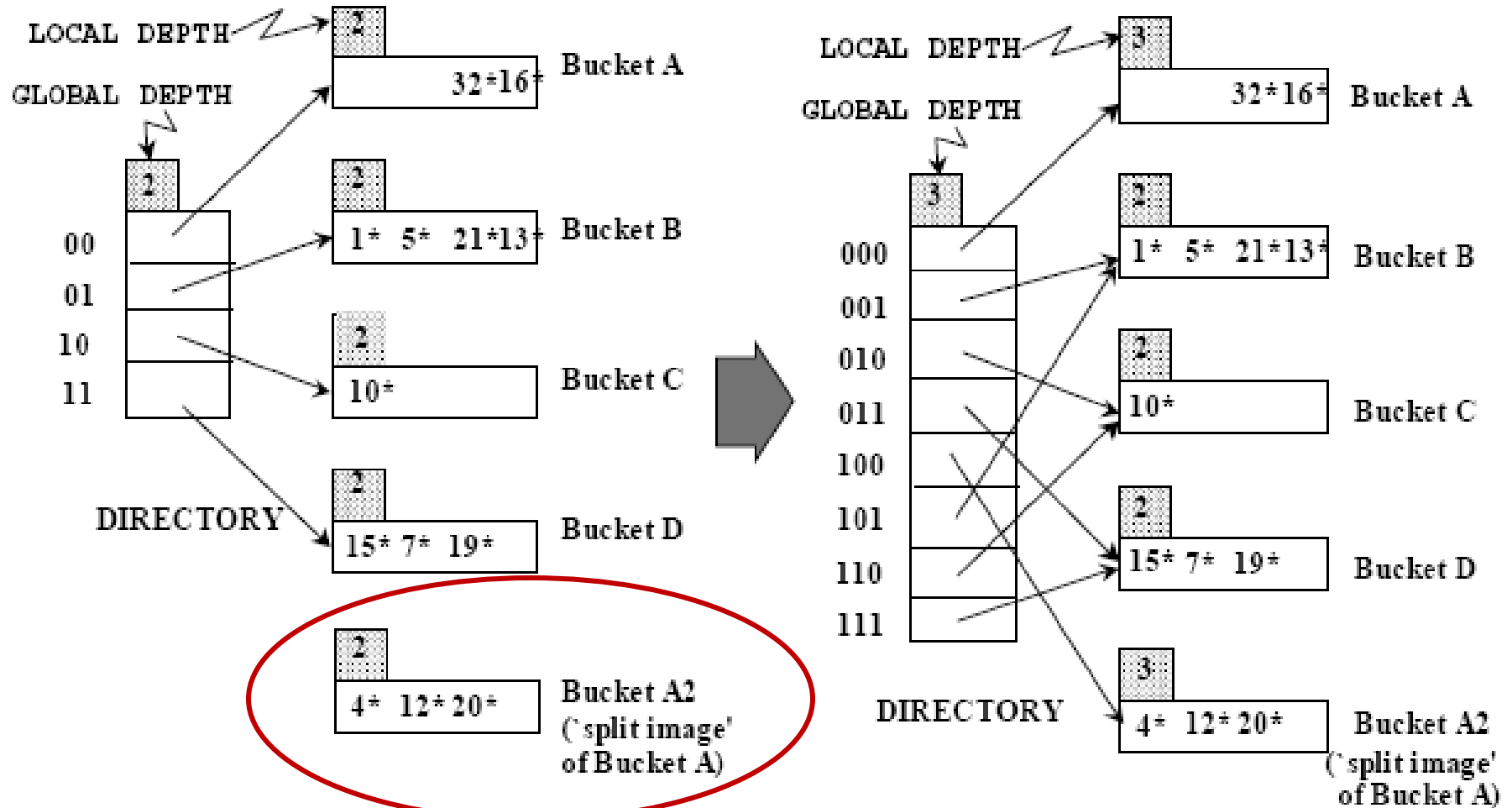
Extendible Hashing: An Example

- The directory is an array of size 4.
- Search:
 - To find the bucket for search key r , take the last “global depth” number of bits of $h(r)$:
 - $h(r) = 5 = \text{binary } 101 \Rightarrow$ The data entry for r is in the bucket pointed to by **01**.
- Insertion:
 - If the bucket is full, **split** it.
 - If “necessary”, **double** the directory.



Extendible Hashing: Directory Doubling

Insert 20^* : $h(r) = 20 = \text{binary } 10100$

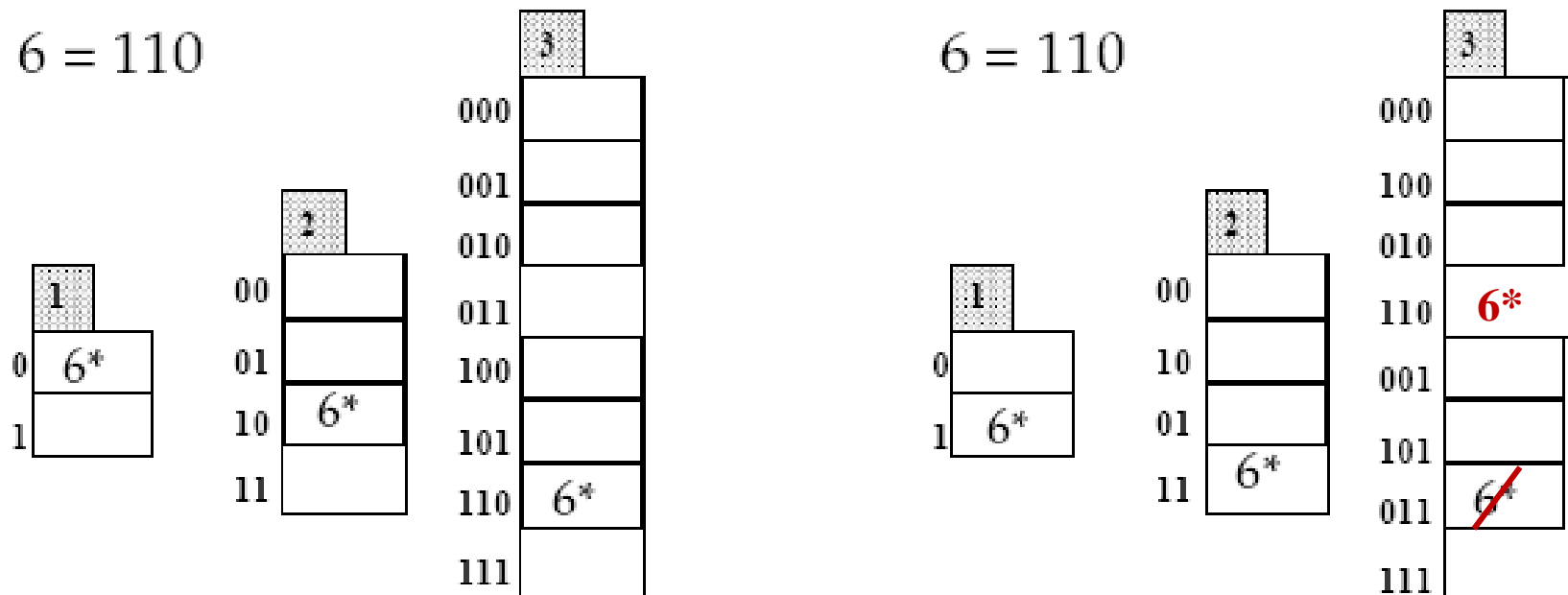


Extendible Hashing: Directory Doubling

- 20 = binary 10100. The last 2 bits (00) tell us that r belongs in bucket **A or A2**. The last 3 bits are needed to tell which.
 - **Global depth** of directory = maximum number of bits needed to tell which bucket an entry belongs to.
 - **Local depth** of a bucket = number of bits used to determine if an entry belongs to a given bucket.
- When does a bucket split cause directory doubling?
 - Before the insertion and split, **local depth = global depth**.
 - After the insertion and split, **local depth > global depth**.
 - Directory is doubled by copying it over and fixing the pointer to the split image page.
 - After the doubling, **global depth = local depth**.

Extendible Hashing: Directory Doubling

- Using the **least significant bits** enables efficient doubling via copying of directory.



Least Significant

vs.

Most Significant

Extendible Hashing: Other Issues

- Efficiency:
 - If the directory fits in memory, an equality selection query can be answered with 1 disk I/O. Otherwise, 2 disk I/Os are needed.
- Deletions:
 - If removal of a data entry makes a bucket empty, then that bucket can be merged with its “split image”.
 - Merging buckets decreases the local depth.
 - If each directory element points to the same bucket as its split image, then we can halve the directory.

Linear Hashing: The Idea

- Linear Hashing handles the problem of long overflow chains without using a directory.
- **Idea:** Use a family of hash functions h_0, h_1, h_2, \dots , such that
 - h_{i+1} 's range is twice that of h_i .
 - First, choose an initial hash function h and number of buckets N .
 - Then, $h_i(\text{key}) = h(\text{key}) \bmod (2^i N)$.
 - If $N = 2^{d_0}$, for some d_0 , h_i consists of applying h and looking at the last d_i bits, where $d_i = d_0 + i$.
 - Example: Assume $N = 32 = 2^5$. Then:
 - $d_0 = 5$ (i.e., look at the last 5 bits)
 - $h_0 = h \bmod (1 \cdot 32)$ (i.e., buckets in range 0 to 31)
 - $d_1 = d_0 + 1 = 5 + 1 = 6$ (i.e., look at the last 6 bits)
 - $h_1 = h \bmod (2 \cdot 32)$ (i.e., buckets in range 0 to 63)
 - ... and so on.

Linear Hashing: Rounds of Splitting

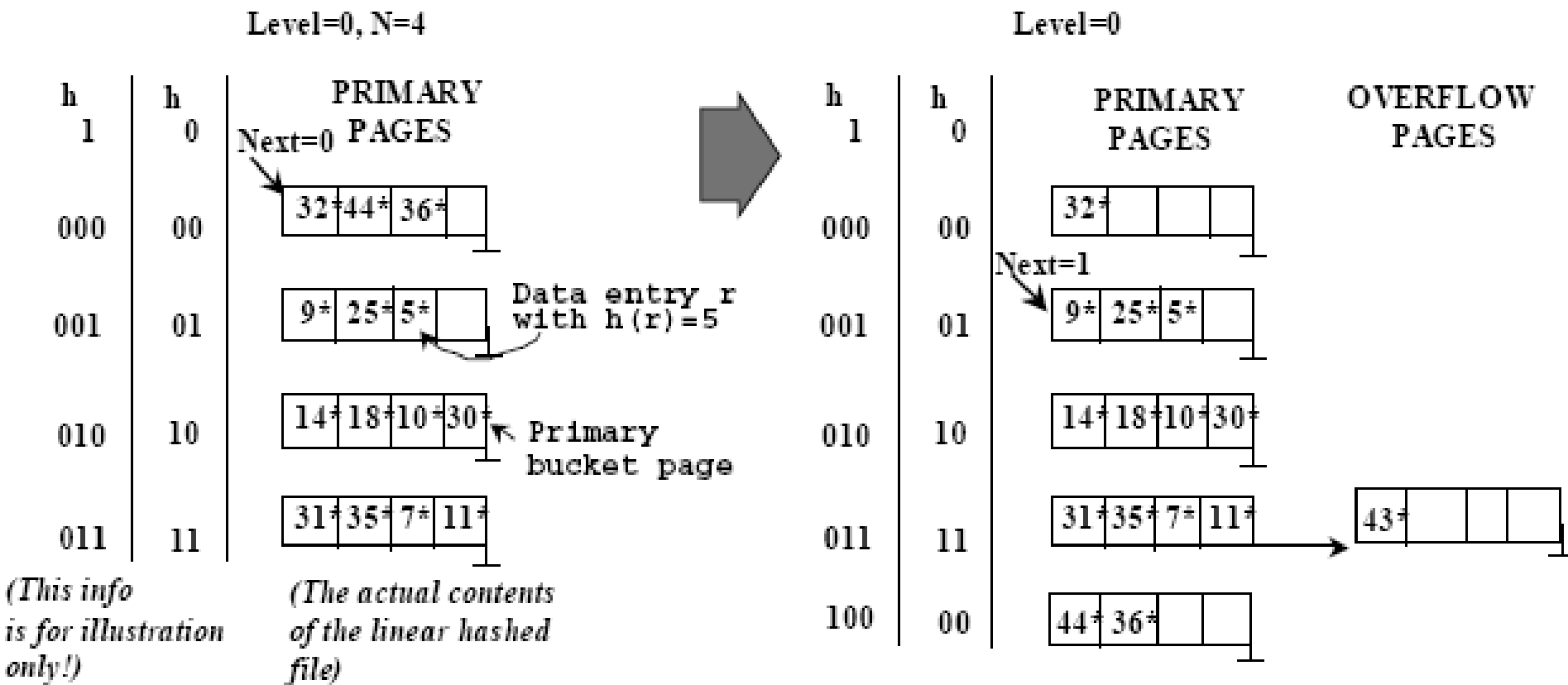
- Directory is avoided in Linear Hashing by using overflow pages, and choosing bucket to split in a round-robin fashion.
 - Splitting proceeds in “rounds”. A round ends when all N_R initial (for round R) buckets are split.
 - Current round number is “Level”. During the current round, only h_{Level} and $h_{Level+1}$ are in use.
 - Search: To find bucket for a data entry r , find $h_{Level}(r)$:
 - Assume: Buckets 0 to $Next-1$ have been split; $Next$ to N_R yet to be split.
 - If $h_{Level}(r)$ in range “ $Next$ to N_R ”, r belongs here.
 - Else, r could belong to bucket $h_{Level}(r)$ or bucket $h_{Level}(r) + N_R$; must apply $h_{Level+1}(r)$ to find out.

Linear Hashing: Insertion

- Insertion: Find bucket by applying h_{Level} and $h_{Level+1}$:
 - If bucket to insert into is full:
 - Add overflow page and insert data entry.
 - Split *Next* bucket and increment *Next*.
- Since buckets are split round-robin, long overflow chains don't develop!
- Similar to directory doubling in Extendible Hashing.

Linear Hashing: An Example

- On split, $h_{Level+1}$ is used to re-distribute entries.



Summary of Hash-based Indexing

- Hash-based indexes are best for **equality selection queries**; they cannot support range selection queries.
- **Static Hashing** can lead to **long overflow chains**.
- **Dynamic Hashing**: Extendible or Linear.
 - **Extendible Hashing** avoids overflow pages by splitting a full bucket when a new data entry is to be added to it.
 - **Directory** to keep track of buckets, doubles periodically.
 - **Linear Hashing** avoids directory by splitting buckets **round-robin** and using **overflow pages**.
 - Overflow pages are not likely to be long (usually at most 2).

Indexing Recap

- Indexed Sequential Access Method (ISAM)
 - A **static**, tree-based index structure.
- B⁺-trees
 - The database index structure; indexing based on any kind of (linear) **order**; adapts **dynamically** to inserts and deletes; low tree heights (~3-4) guarantee fast lookups.
- Clustered vs. Unclustered Indexes
 - An index is clustered if its underlying data pages are ordered according to the index; fast **sequential access** for clustered B⁺-trees.
- Hash-Based Indexes
 - **Extendible hashing** and **linear hashing** adapt dynamically to the number of data entries.