Database System
Architecture and Implementation

Tree-Structured Indexes

Orientation

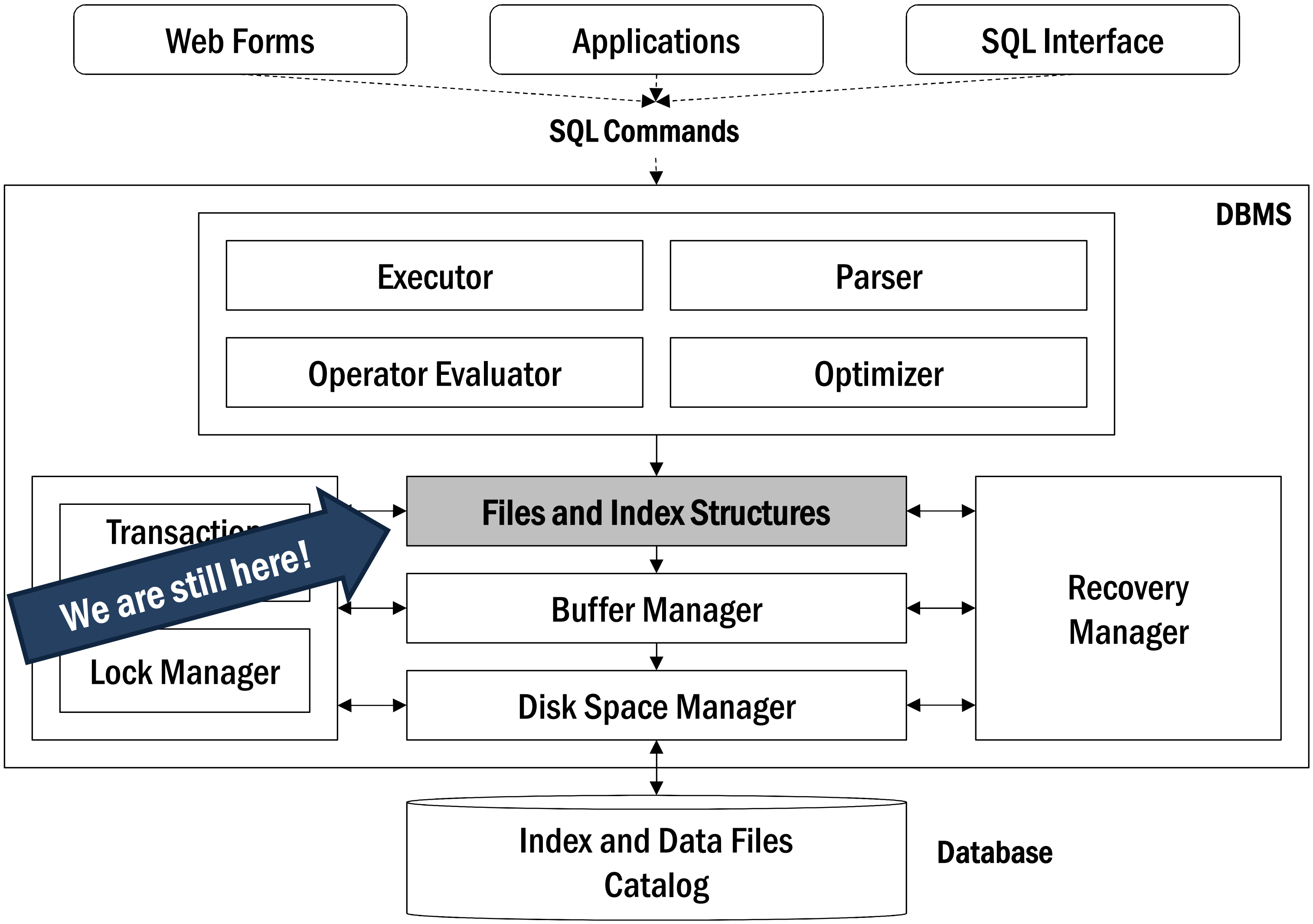
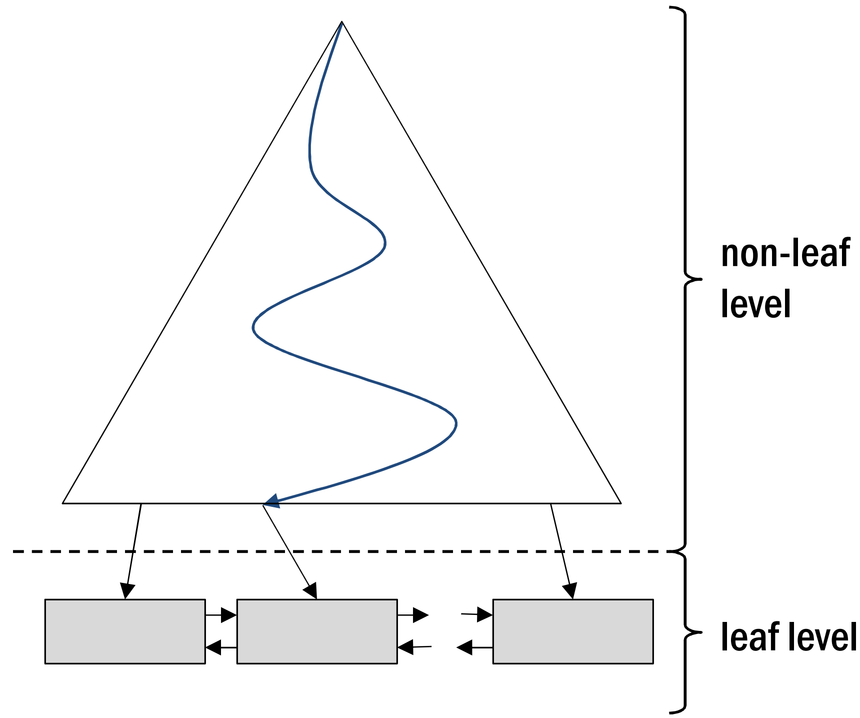


Figure Credit: Raghu Ramakrishnan and Johannes Gehrke: "Database Management Systems", McGraw-Hill, 2003.

Module Overview

- Binary search
- ISAM
- B+ trees
  - search, insert, and delete
  - duplicates
  - key compression
  - bulk loading



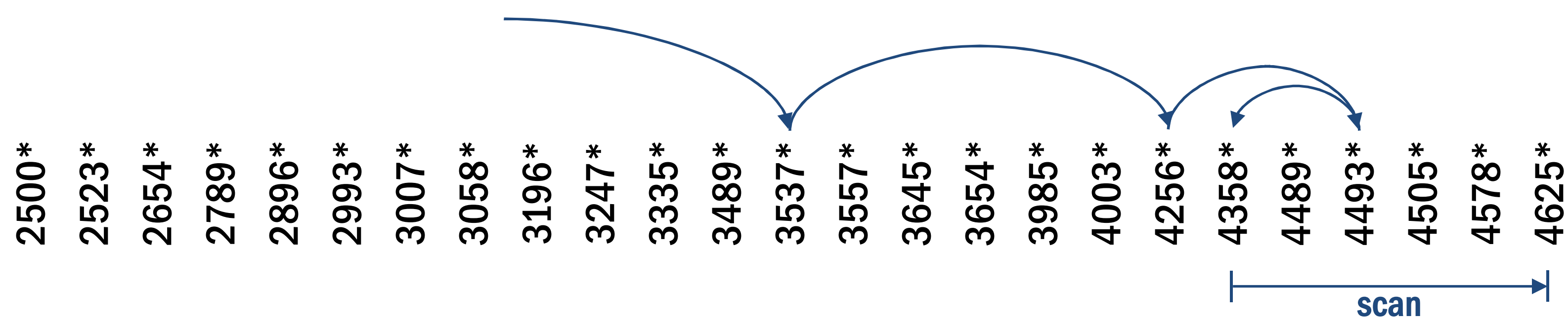
Binary Search

How could we prepare for such queries and evaluate them efficiently

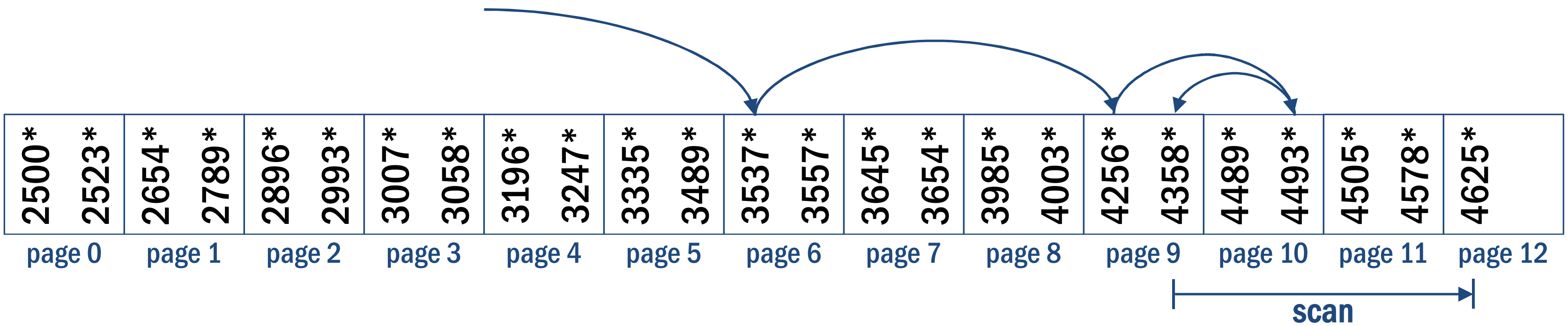
```
SELECT *
FROM   Employees
WHERE  Sal BETWEEN 4300 AND 4600
```

- We could
  1. **sort** the table on disk (in `Sal`-order)
  2. **use binary search** to find the first qualifying tuple, then scan as long as `Sal < 4600`

Again, let k^* denote the full record with key k



Binary Search



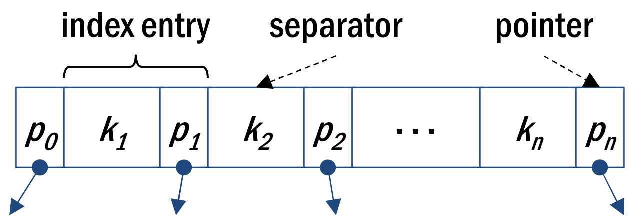
- Page I/O operations
  - during the **scan phase**, pages are **accessed sequentially**
  - during the **search phase**, $\log_2$(*#tuples*) need to be read
  - about **the same number of pages** as tuples need to be read!
- Binary search is that it makes **far, unpredictable jumps**, which largely defeat page prefetching

Tree-Structured Indexing

- Intuition
  - improve binary search by introducing an **auxiliary structure** that only contains **one record per page** of the original (data) file
  - use this idea recursively until all records fit into **one single page**
- This simple idea naturally leads to a **tree-structured** organization of the indexes
  - ISAM
  - B+ trees
- Tree-structures indexes are particularly useful if **range selections** (and thus sorted file scans) need to be supported

Indexed Sequential Access Method

- ISAM
  - acts as **static replacement** for the binary search phase
  - reads **considerable fewer pages** than binary search
- 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

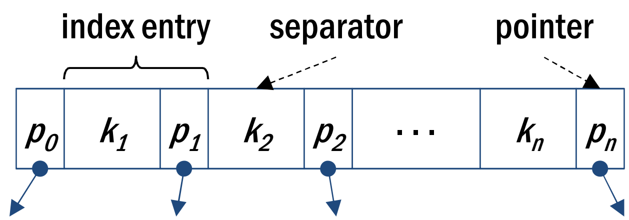


  2. ISAM leads to **sparse** index structures, since in an index entry

$$\langle k_i, \uparrow p_i \rangle$$

key k_i is the first (i.e., minimal) **A**-value on the data file page pointed to by p_i, where p_i is the page number

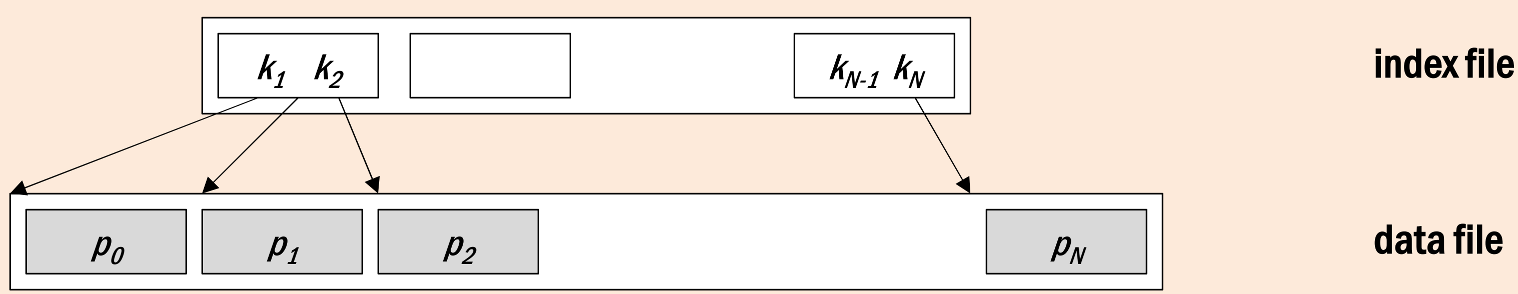
Indexed Sequential Access Method



3. in the index file, the k_i serve as separators between the contents of pages p_{i-1} and p_i
4. it is guaranteed that $k_{i-1} < k_i$ for $i = 2, \ldots, n$

- We obtain a **one-level ISAM structure**

One-level ISAM structure for $N + 1$ pages



Searching in ISAM

SQL query with range selection on field A

```
SELECT *
FROM   R
WHERE  A BETWEEN lower AND upper
```

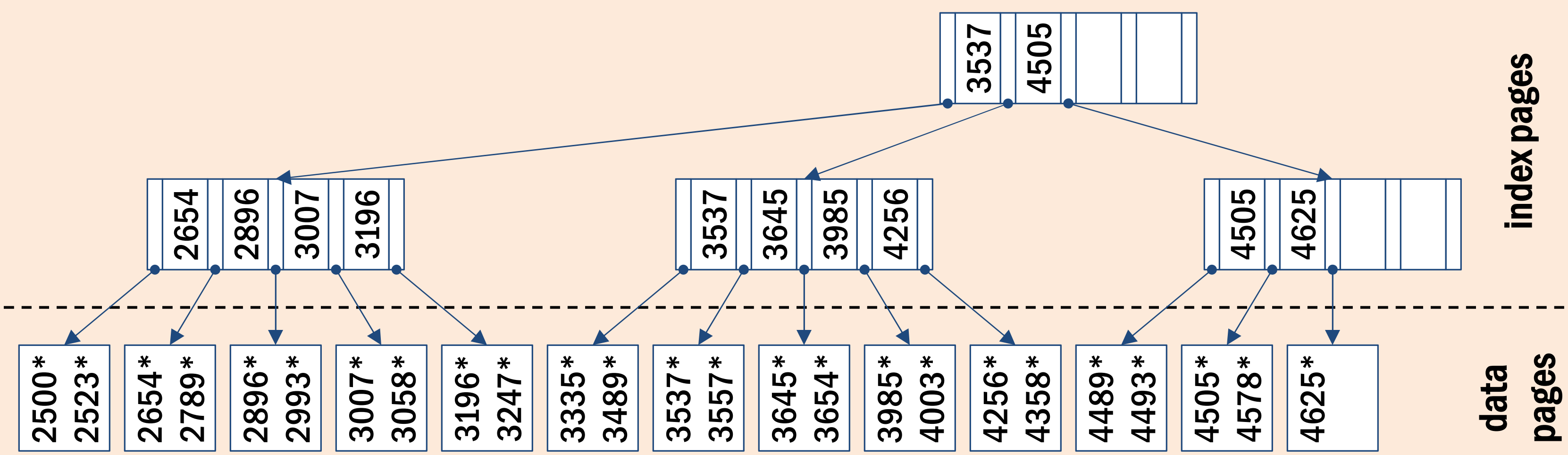
- To support range selection
  1. conduct a **binary search on the index file** for a key of value *lower*
  2. start a **sequential scan of the data file** from the page pointed to by the index entry and scan until field **A** exceeds *upper*
- Index file size is likely to be **much smaller** than data file size
  - searching the index is far more efficient than searching the data file
  - however, for large data files, even the index file might be too large to support fast searches

Multi-Level ISAM Structure

- **Recursively** apply the index creation step
  - treat the top-most index level like the data file and add an additional index layer on top
  - repeat until the top-most index layer fits into a single page (root page)
- This recursive index creation scheme leads to a **tree-structured** hierarchy of index levels

Multi-Level ISAM Structure

Example

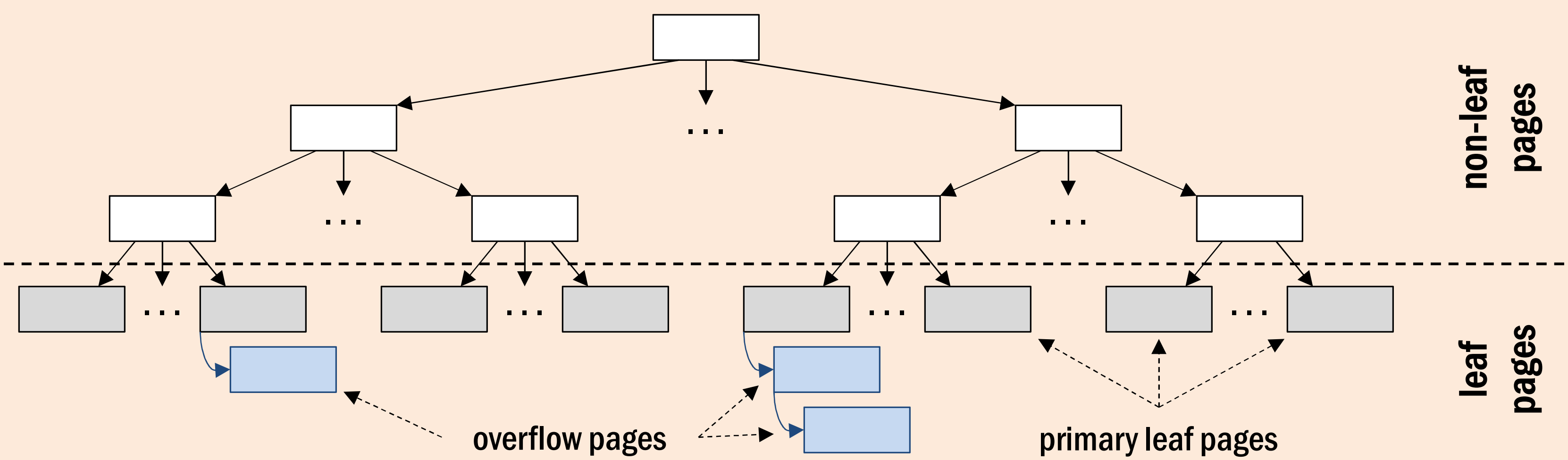


- Each ISAM tree node corresponds to **one page** (disk block)
- ISAM structure for a given data file is created **bottom up**
  1. sort the data file on the search key field
  2. create the index leaf level
  3. if top-most index level contains more than one page, repeat

ISAM Overflow Pages

- The upper levels of the ISAM tree always remain **static**: updates in the data file **do not affect** the upper tree levels
  - if **space is available** on the corresponding leaf page, insert record there
  - otherwise, create and maintain a chain of **overflow pages** hanging off the full primary leaf page (overflow pages are **not ordered** in general)
- Over time, **search performance in ISAM can degrade**

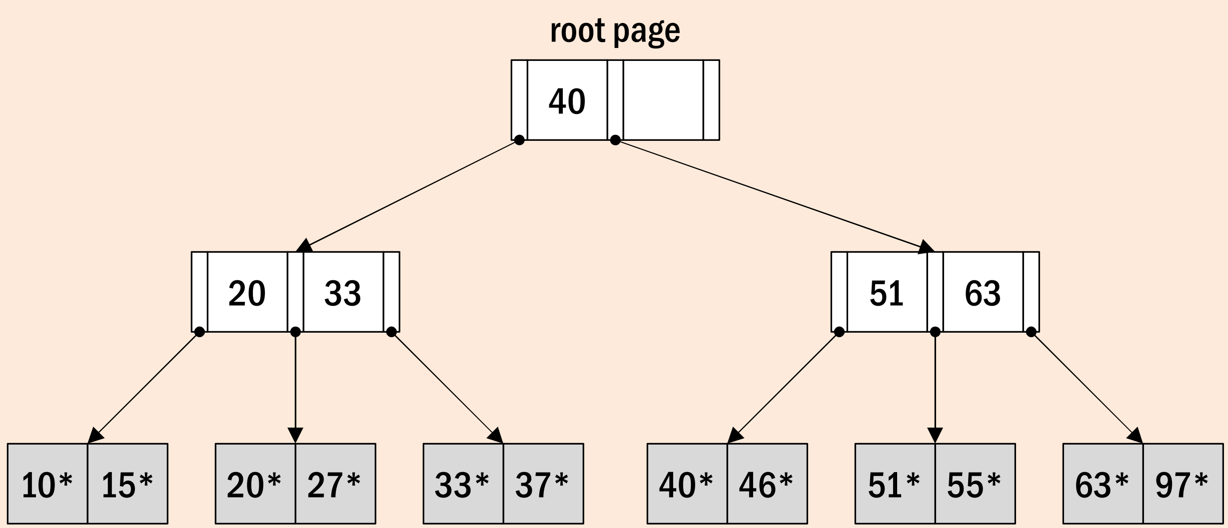
Multi-level ISAM structure with overflow pages



ISAM Example: Initial State

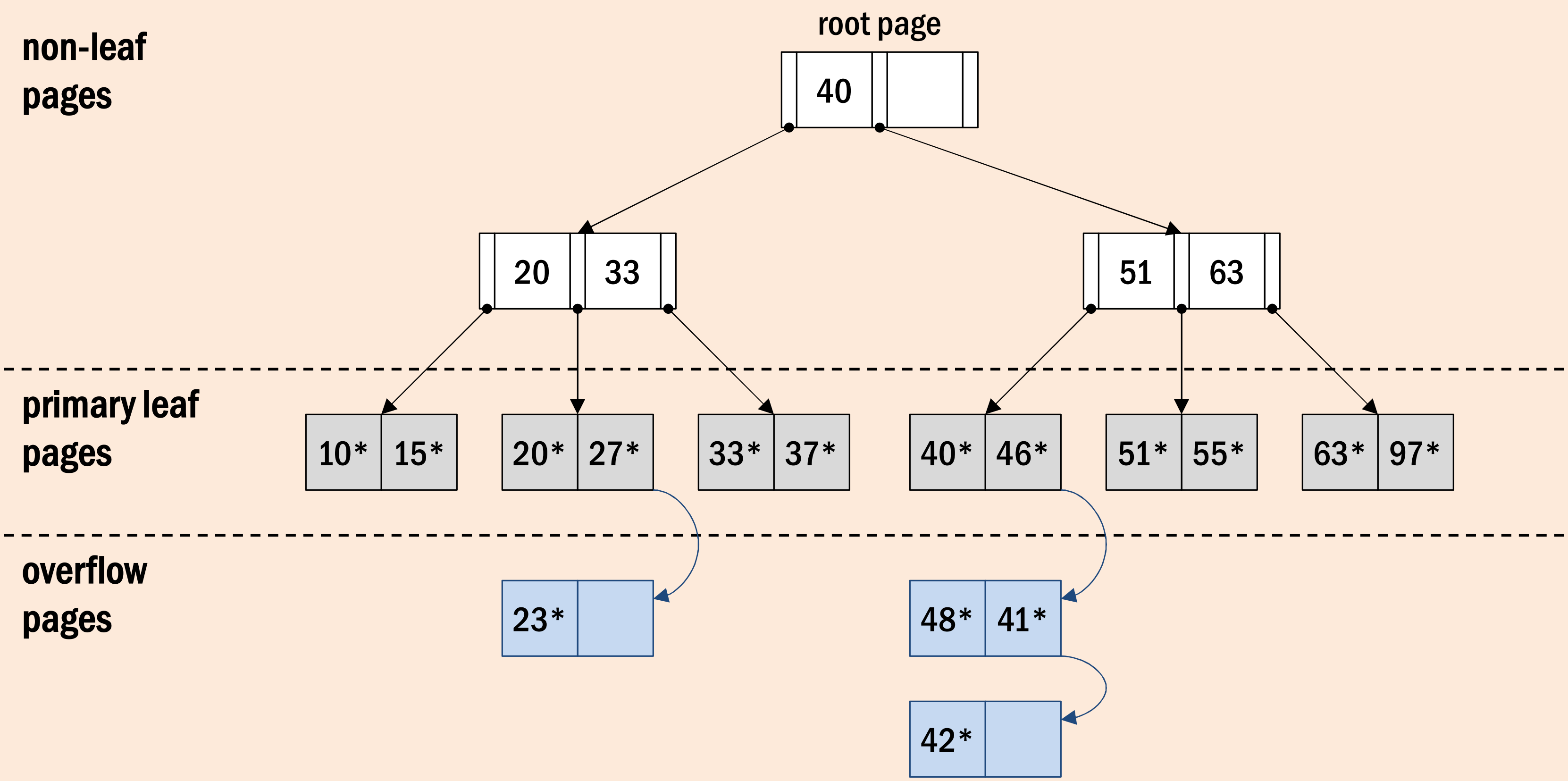
- Each page can hold **two** index entries **plus one** (the left-most) page pointer

Initial State of ISAM structure



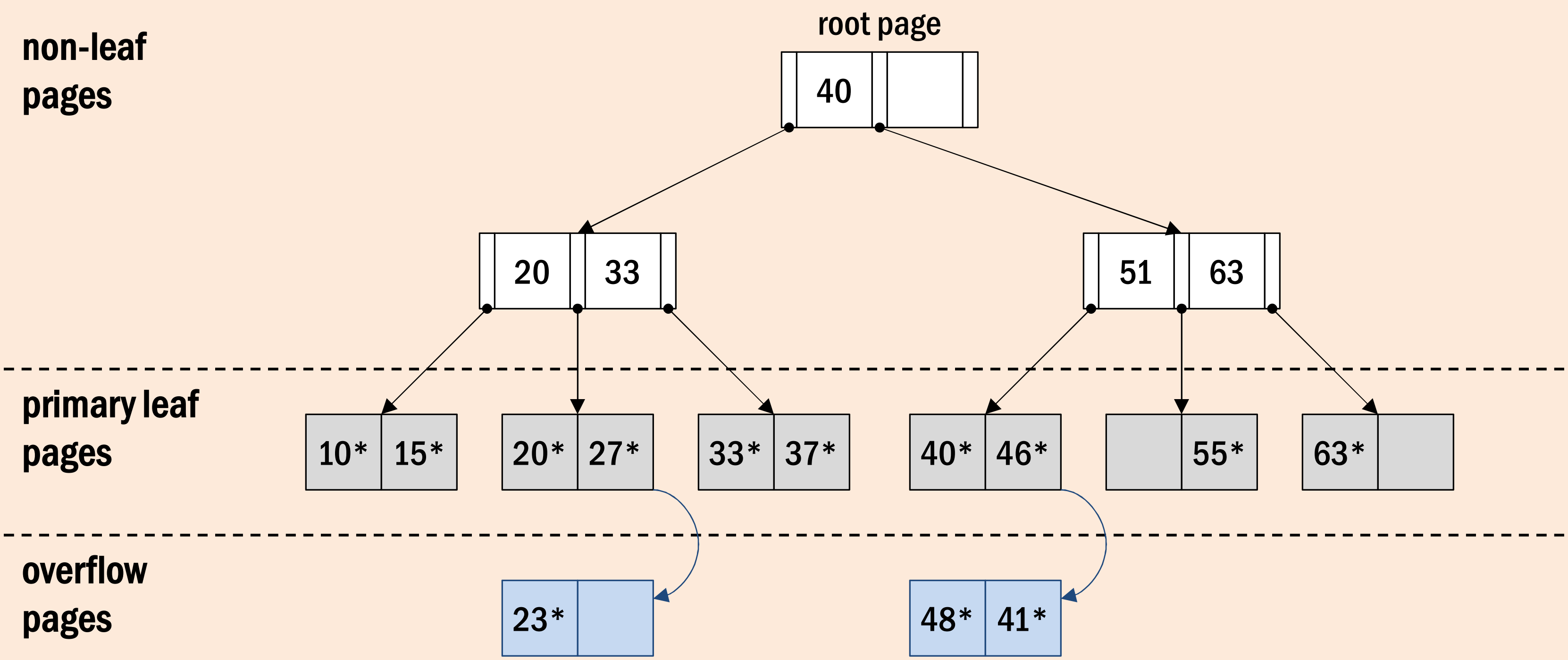
ISAM Example: Insertions

ISAM structure after insertion of data records with keys 23, 48, 41, and 42



ISAM Example: Deletions

ISAM structure after deletion of data records with keys 42, 51, and 97



Is ISAM Too Static?

- Recall that ISAM structure is **static**
  - non-leaf levels are **not touched at all** by updates to the data file
  - may lead to **orphaned index key entries**, which do not appear in the index leaf level (e.g., key value **51** on the previous slide)

Orphaned index key entries

Does an index key entry like **51** (on the previous slide) cause problems during index key searches?

- To preserve the **separator property** of index key entries, it is necessary to maintain overflow chains
- ISAM may **lose balance** after heavy updating, which complicates the life for the query optimizer

Is ISAM Too Static?

- Recall that ISAM structure is **static**
  - non-leaf levels are **not touched at all** by updates to the data file
  - may lead to **orphaned index key entries**, which do not appear in the index leaf level (e.g., key value **51** on the previous slide)

Orphaned index key entries

Does an index key entry like **51** (on the previous slide) cause problems during index key searches?

↳ No, since the index keys maintain their separator property.

- To preserve the **separator property** of index key entries, it is necessary to maintain overflow chains
- ISAM may **lose balance** after heavy updating, which complicates the life for the query optimizer

Static Is Not All Bad

- Leaving **free space** during index creation reduces the insertion/overflow problem (typically ≈ 20% free space)
- Since ISAM indexes are static, **pages do not need to be locked** during concurrent 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

ISAM-style implementations

- **MySQL**
  - implements and extends ISAM as MyISAM, which is the default storage engine
- **Berkeley DB**
- **Microsoft Access**

Fan-Out

Definition

The average number of children for a non-leaf node is called the **fan-out** of the tree. If every non-leaf node has n children, a tree of height h has n^h leaf pages.

↳ In practice, nodes do not have the same number of children, but using the **average value** F for n is a good **approximation** to the number of leaf pages F^h.

Exercise: Number of children

Why can non-leaf nodes have **different** numbers of children?

Fan-Out

Definition

The average number of children for a non-leaf node is called the **fan-out** of the tree. If every non-leaf node has *n* children, a tree of height *h* has n^h leaf pages.

↳ In practice, nodes do not have the same number of children, but using the **average value** *F* for *n* is a good **approximation** to the number of leaf pages F^h.

Exercise: Number of children

Why can non-leaf nodes have **different** numbers of children?

↳ Index entries $k*$ can be of variable length if the index is built on a variable-length key. Additionally, index entries $k*$ of variant □ and □ can be of variable length because variable-length records or lists of *rids* are stored in the index entries.

Cost of Searches in ISAM

- Let N be the number of pages in the data file and let F denote the fan-out of the ISAM tree
  - when the index search begins, the search space is of size N
  - with the help of the root page, the index search is guided to a sub-tree of size
$$N \cdot {}^{1}\!/_{F}$$
  - as the index search continues down the tree, the search space is repeatedly reduced by a factor of F
$$N \cdot {}^{1}\!/_{F} \cdot {}^{1}\!/_{F} \cdots$$
  - the index search ends after s steps, when the search space has been reduced to size 1 (i.e., when it reaches the index leaf level and hits the data page that contains the desired record)
$$N \cdot \left({}^{1}\!/_{F}\right)^{s} \stackrel{\text{def}}{=} 1 \quad \Leftrightarrow \quad s = \log_F N$$

Cost of Searches in ISAM

- **Exercise: Binary search vs. tree**

Assume a data file consists of 100 million leaf pages. How many page I/O operations will it take to find a value using **binary search** and an **ISAM tree with fan-out 100**?

Cost of Searches in ISAM

Exercise: Binary search vs. tree

Assume a data file consists of 100 million leaf pages. How many page I/O operations will it take to find a value using **binary search** and an **ISAM tree with fan-out 100**?

- **binary search**

$$\log_2(100{,}000{,}000) \approx 25 \text{ page I/O operations}$$

- **ISAM tree with fan-out 100**

$$\log_{100}(100{,}000{,}000) = 4 \text{ page I/O operations}$$

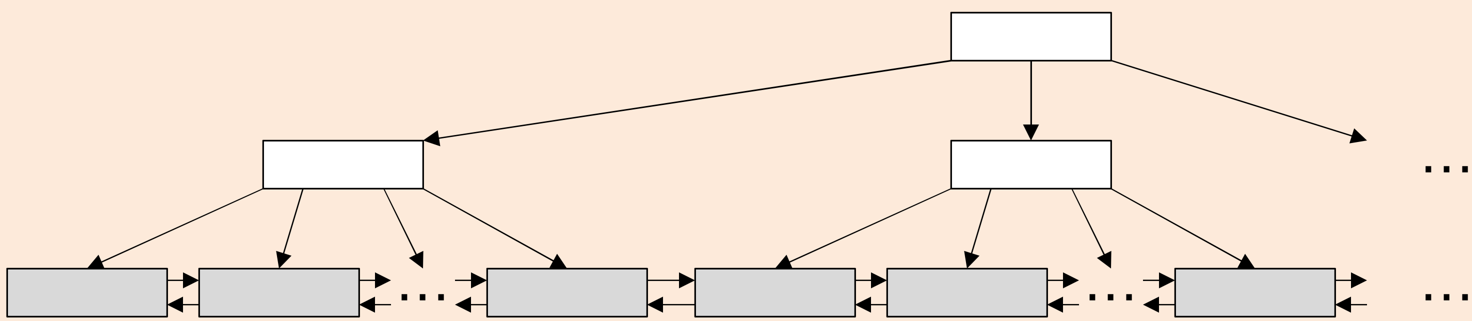
B+ Tree Properties

- The **B+ tree index structure** is derived from the ISAM index structure, but is fully dynamic w.r.t. updates
  - search performance is only dependent on the **height** of the B+ tree (because of a high fan-out, the height rarely exceeds 3)
  - B+ trees **remains balanced**, **no overflow chains** develop
  - B+ trees support efficient **insert/delete operations**, where the underlying data file can grow/shrink dynamically
  - B+ tree nodes (with the exception of the root node) are **guaranteed to have a minimum occupancy of 50%** (typically 66%)

B+ Trees Structure

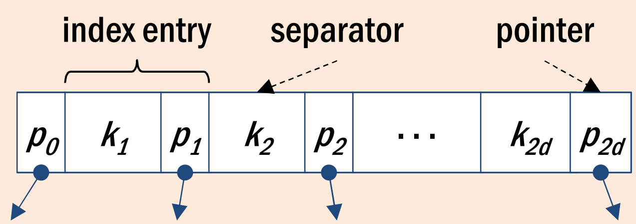
- Differences between B+ tree structure and ISAM structure
  - leaf nodes are connected to form a **doubly-linked list**, the so-called **sequence set**
    - ↳ *not a strict requirement, but implemented in most systems*
  - leaves may contain **actual data records** (variant □) or just **references to records** on data pages (variants □ and □)
    - ↳ *instead, ISAM leaves are the data pages themselves*

Sketch of B+ tree structure (data pages not shown)



B+ Tree Non-Leaf Nodes

B+ inner (non-leaf) node



- B+ 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+ tree

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

  - a node contains $n + 1$ pointers, where pointer p_i $(1 \leq i \leq n - 1)$ points to a sub-tree in which all key values k are such that

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

    (p_0 points to a sub-tree with key values $< k_1$, p_{2d} points to a sub-tree with key values $\geq k_{2d}$)

B+ Tree Leaf Nodes

- B+ tree leaf nodes contain pointers to data **records** (not **pages**)
- A **leaf node entry** with key value *k* is denoted as *k** as before
- All index entry variants □, □, and □ can be used to implement the leaf entries
  - for variant □, the B+ tree represents the index as well as the data file itself and leaf node entries therefore look like

$$k_i* = \langle k_i, \langle . . . \rangle\rangle$$

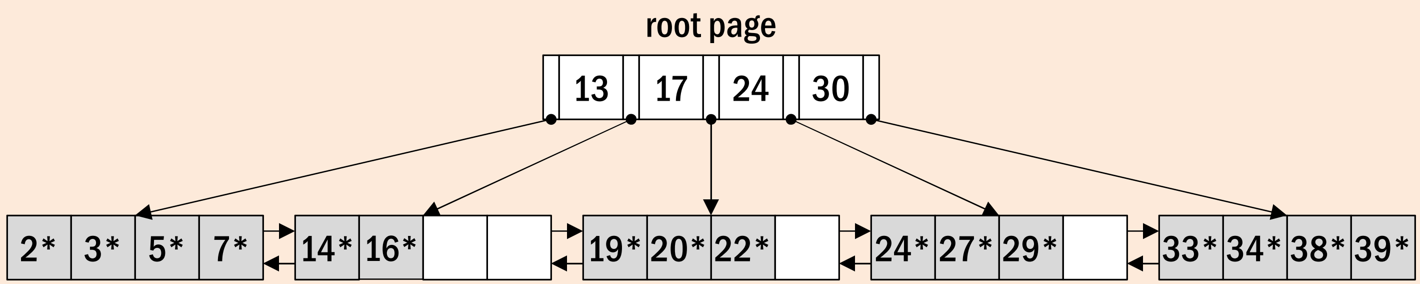
  - for variants □ and □, the B+ tree is managed in a file separate from the actual data file and leaf node entries look like

$$k_i* = \langle k_i, rid\rangle$$

$$k_i* = \langle k_i, [rid_1, rid_2, ...]\rangle$$

B+ Tree Search

Example of a B+ tree with order *d* = 2



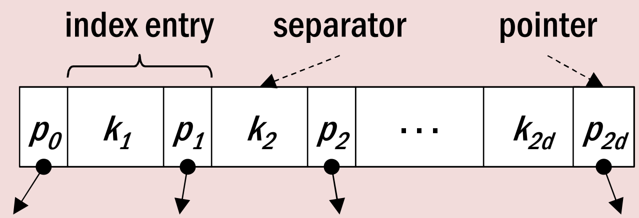
- Each node contains between 2 and 4 entries (order *d* = 2)
- Example of B+ tree searches
  - for entry 5*, follow the left-most child pointer, since 5 < 13
  - for entries 14* or 15*, follow the second pointer, since 13 ≤ 14 < 17 and 13 ≤ 15 < 17 (because 15* cannot be found on the appropriate leaf, it can be concluded that it is not present in the tree)
  - for entry 24*, follow the fourth child pointer, since 24 ≤ 24 < 30

B+ Tree Search

Searching in a B+ tree

```
function search(k): ↑node
  return treeSearch(root, k)
end

function treeSearch(↑node, k): ↑node
  if node is a leaf node then return ↑node
  else
    if k < k_1 then return treeSearch(p_0, k);
    else
      if k ≥ k_2d then return treeSearch(p_2d, k);
      else
        find i such that k_i ≤ k < k_i+1;
        return treeSearch(p_i, k)
end
```



B+ Tree Insert

- B+ trees remain **balanced** regardless of the updates performed
  - **invariant:** all paths from the root to any leaf must be of **equal length**
  - insertions and deletions have to **preserve** this invariant

Basic principle of insertion into a B+ tree with order *d*

To insert a record with key *k*

1. start with root node and **recursively** insert entry into appropriate child node
2. descend down tree until **leaf node is found,** where entry belongs
   (let *n* denote the leaf node to hold the record and *m* the number of entries in *n*)
3. if $m < 2 \cdot d$, there is capacity left in *n* and k^* **can be stored in leaf node *n***
   - **Otherwise…?**

- We *cannot* start an overflow chain hanging off p as this solution would violate the balancing invariant
- We *cannot* place k^* elsewhere (even close to *n*) as the cost of `search(k)` should only be dependent on the tree's height

B+ Tree Insert

Splitting nodes

If a node *n* is full, it must be **split**

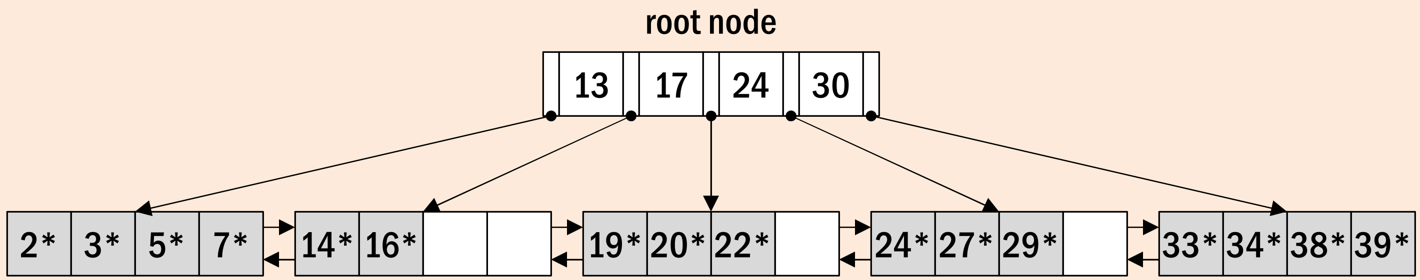
1. create a **new node** *n'*
2. distribute the entries of *n* and the new entry *k* **over** *n* **and** *n'*
3. insert an entry ↑ *n'* pointing to the new node *n'* **into its parent**

Splitting can therefore **propagate up the tree**. If the root has to be split, a new root is created and the **height of the tree increases** by 1.

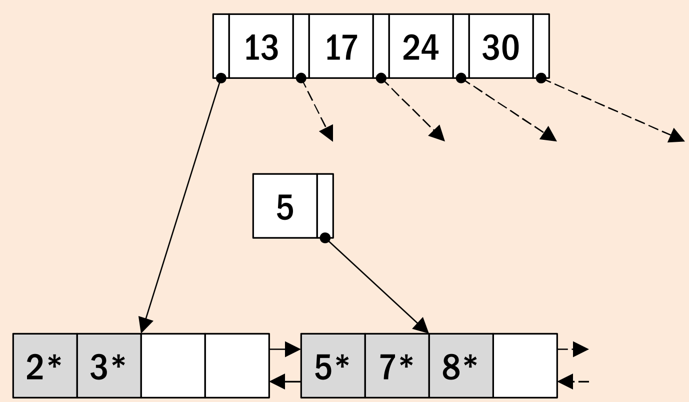
B+ Tree Insert

Example: Insertion into a B+ tree with order *d* = 2

1. insert record with key *k* = 8 into the following B+ tree



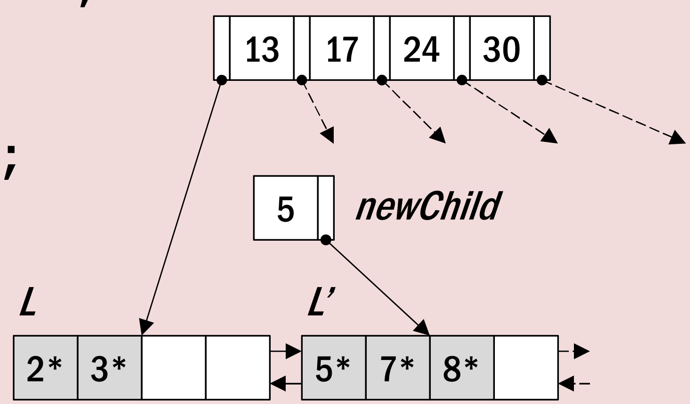
2. the new record has to be inserted into the left-most leaf node *n*
3. since *n* is **already full**, it has to be split
4. create a **new leaf node** *n'*
5. entries 2* and 3* remain on *n*, whereas entries 5*, 7* and 8* (new) go into *n'*
6. key *k'* = 5 is the **new separator** between nodes n and n' and has to be **inserted into their parent** (copy up)



B+ Tree Insert

Insert into B+ tree of degree *d* (leaf nodes)

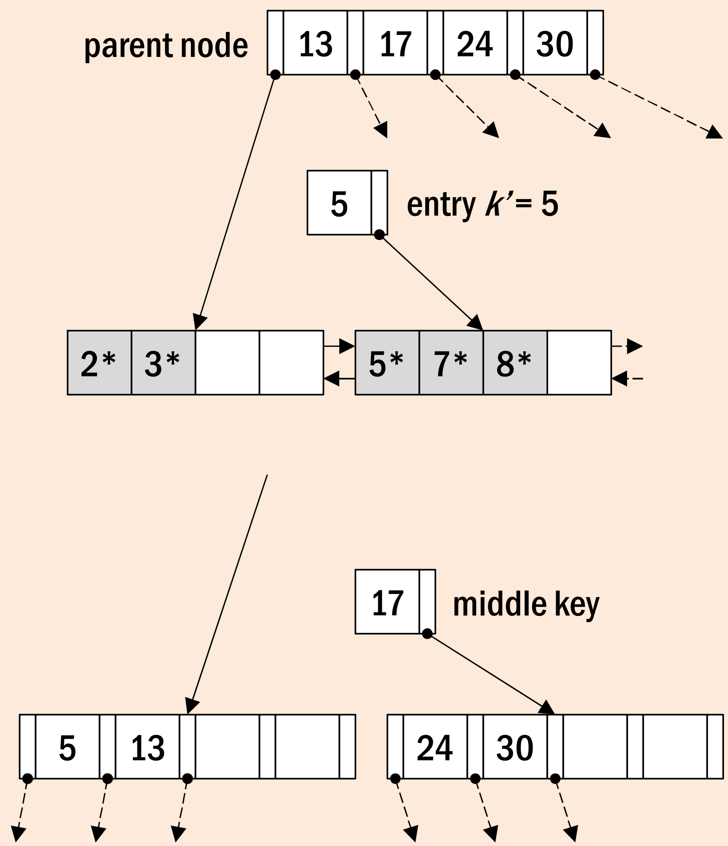
function insert(↑ *node*, *k** **) :** ↑ *newChild*
  if *node* is a non-leaf node, say *N* **then . . .**
  if *node* is a leaf node, say *L* **then**
    if *L* has space **then**
      put *k** on *L*; ↑ *newChild* ← null; **return;**
    else
      split *L*: first *d* entries stay, rest move to new node *L'*;
      put *k** on *L* or *L'*;
      set sibling pointers in *L* and *L'*;
      ↑ *newChild* ← @(⟨smallest key value on *L'*, ↑ *L'*⟩);
      return;
endproc;



B+ Tree Insert

Example: Insertion into a B+ tree with order $d = 2$ (cont'd)

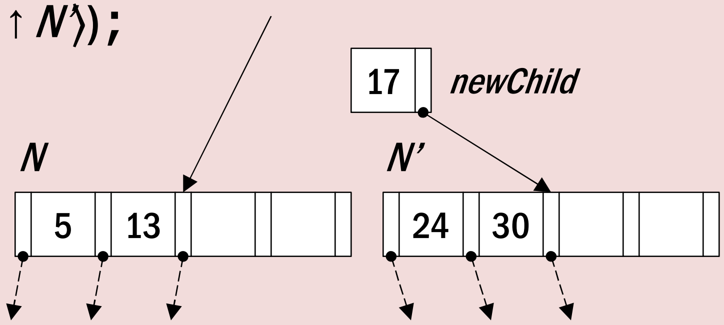
7. to insert entry $k' = 5$ into parent node, **another split** has to occur
8. parent node is **also full** since it already has $2d$ keys and $2d + 1$ pointers
9. with the new entry, there is a **total** of $2d + 1$ keys and $2d + 2$ pointers
10. form **two minimally full non-leaf nodes,** each containing d keys and $d + 1$ pointers, **plus an extra key**, the **middle** key
11. middle key plus pointer to second non-leaf node constitute a **new index entry**
12. new index entry has to be **inserted into parent** of split non-leaf node (push up)



B+ Tree Insert

Insert into B+ tree of degree *d* (non-leaf nodes)

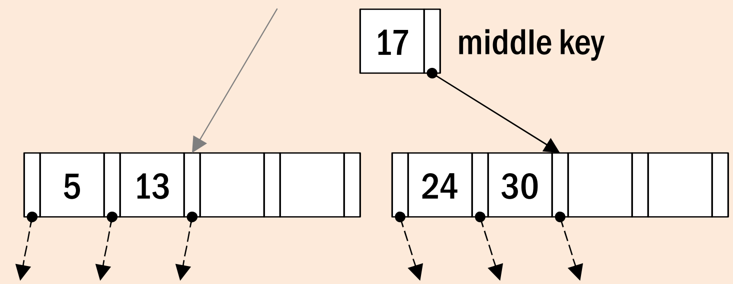
```
function insert(↑node, k*): ↑newChild
  if node is a non-leaf node, say N then
    find i such that k_i ≤ k < k_{i+1};
    ↑newChild = insert(p_i, k);
    if ↑newChild is null then return;
    else
      if N has space then put newChild on it; ↑newChild ← null; return;
      else
        split N:  first d key values and d + 1 pointers stay,
                  last d key values and d + 1 pointers move to new node N';
        ↑newChild ← @(⟨smallest key value on N', ↑N'⟩);
        if N is root then ...
        return;
  if node is a leaf node, say L then ...
endproc;
```

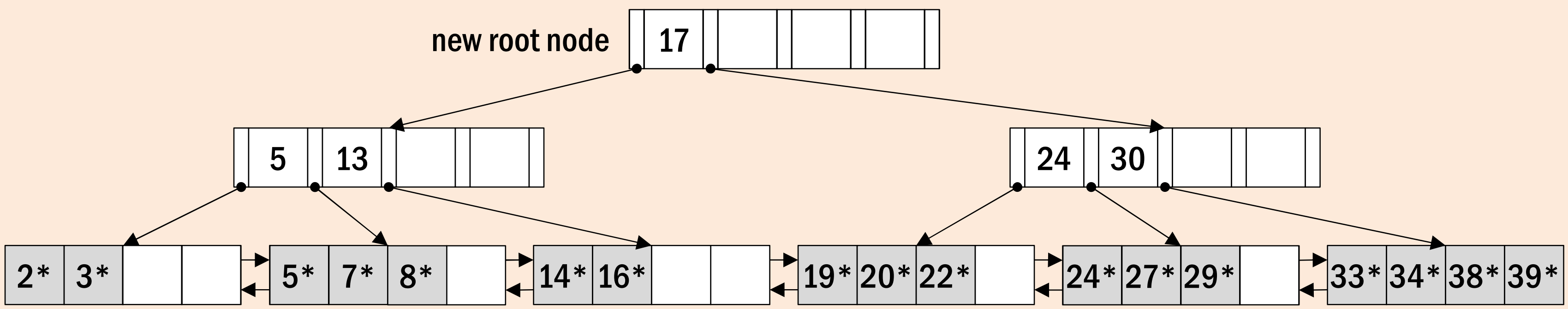


B+ Tree Insert

Example: Insertion into a B+ tree with order *d* = 2 (cont'd)

13. parent node that was split, was the (old) **root node** of the B+ tree
14. create a **new root node** to hold the entry that distinguishes the two split index pages





B+ Tree Insert

Insert into B+ tree of degree *d* (root node)

```
function insert(↑ node, k*): ↑ newChild
  if node is a non-leaf node, say N then
    ...
        split N:   first d key values and d + 1 pointers stay,
                   last d key values and d + 1 pointers move to new node N';
        ↑ newChild ← @(⟨smallest key value on N', ↑ N'⟩);
        if N is root then
          create new node with ⟨ ↑ N, newChild⟩;
          make the tree's root node pointer point to the new node
        return;
  if node is a leaf node, say L then ...
endproc;
```

B+ Tree Root Node Split

- Splitting starts at the leaf level and continues upward as long as index nodes are fully occupied
- Eventually, the root node might be split
  - root node is the only node that may have an occupancy of < 50%
  - tree height only increases if the root is split

How often do you expect a root split to happen?

B+ Tree Root Node Split

- Splitting starts at the leaf level and continues upward as long as index nodes are fully occupied
- Eventually, the root node might be split
  - root node is the only node that may have an occupancy of < 50%
  - tree height only increases if the root is split

How often do you expect a root split to happen?

Assume a B+ tree over 8 byte integers with 4 kB pages and with pointers encoded as 8 byte integers.

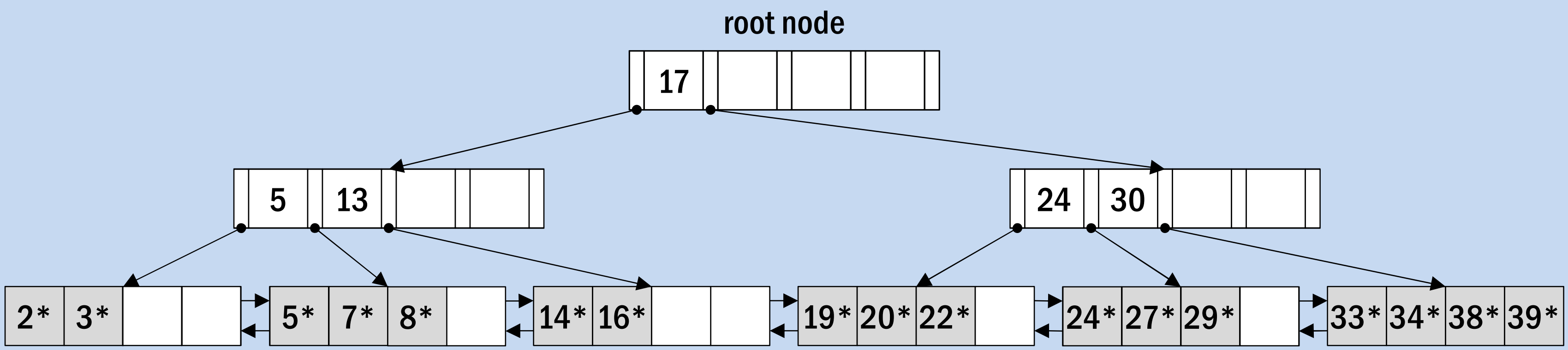
- 128-256 index entries/page (fan-out *F*)
- an index of height h indexes at least 128^h records, typically more

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

B+ Tree Insert

Further key insertions

How does the insertion of records with keys *k* = **23** and *k* = **40** alter the B+ tree?

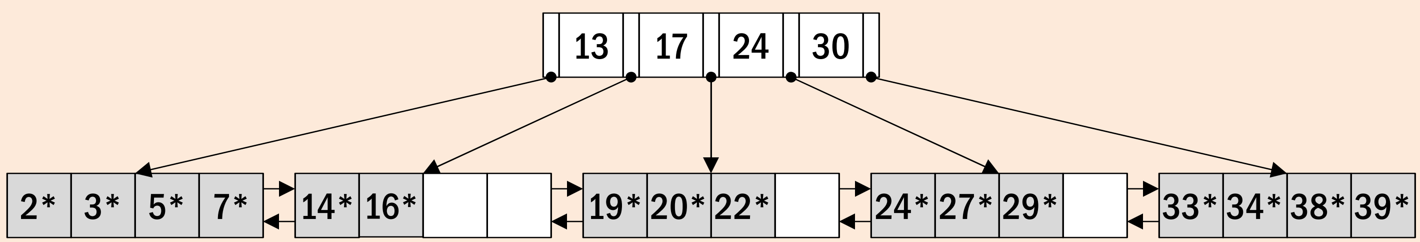


B+ Tree Insert with Redistribution

- Redistribution further improves average occupancy in a B+ tree
  - before a node *n* is split, its entries are **redistributed** with a sibling
  - a **sibling** of a node *n* is a node that is immediately to the left or right of *N* and has the same parent as *n*

Example: Insertion with redistribution into a B+ tree with order $d = 2$

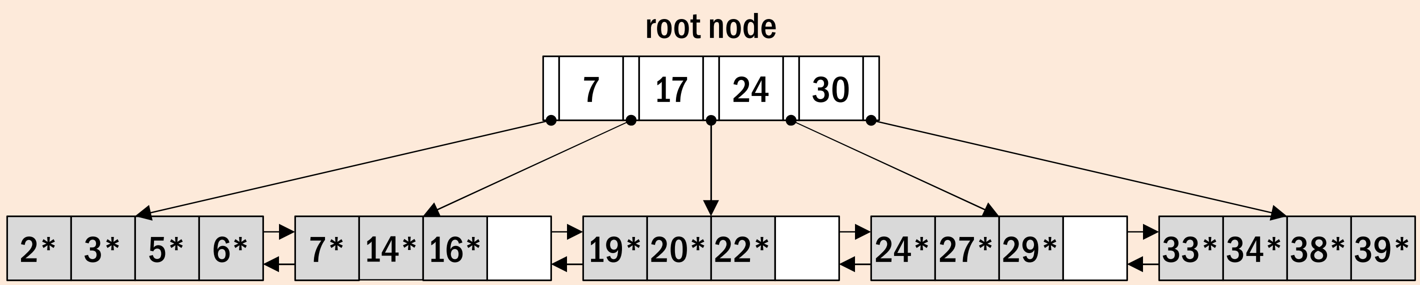
1. insert record with key $k = 6$ into the following B+ tree



2. the new record has to be inserted into the left-most leaf node, say *n*, which is **full**
3. however, the (only) sibling of *n* **only has two entries and can accommodate more**
4. therefore, insert of $k = 6$ can be handled with a **redistribution**

B+ Tree Insert with Redistribution

Example: Insertion with redistribution into a B+ tree with order *d* = 2



5. redistribution “**rotates**” values through the parent node from node *n* to its sibling

Slides Credit: Michael Grossniklaus – Uni-Konstanz

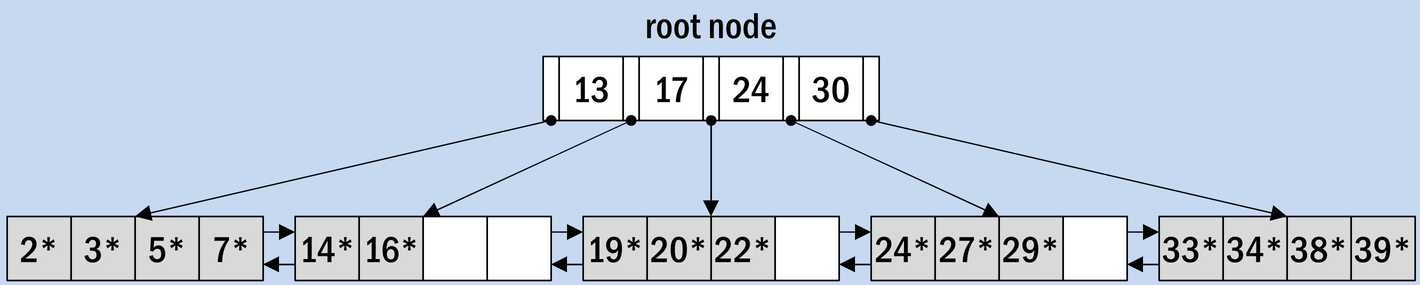
B+ Tree Insert with Redistribution

Redistribution makes a difference

Insert a record with key *k* = **`30`**

- without redistribution
- using leaf-level redistribution

into the B+ tree shown below. How does the tree change?



B+ Tree Delete

- B+ tree deletion algorithm follows the same basic principle as the insertion algorithm

Basic principle of deletion from a B+ tree with order *d*

To delete a record with key *k*

1. start with root node and **recursively** delete entry from appropriate child node
2. descend down tree until **leaf node is found,** where entry is stored (let *n* denote the leaf node that holds the record and *m* the number of entries in *n*)
3. if $m > d$, *n* does not have minimum occupancy and k^* **can simply be deleted from leaf node** *n*
   - **Otherwise...?**

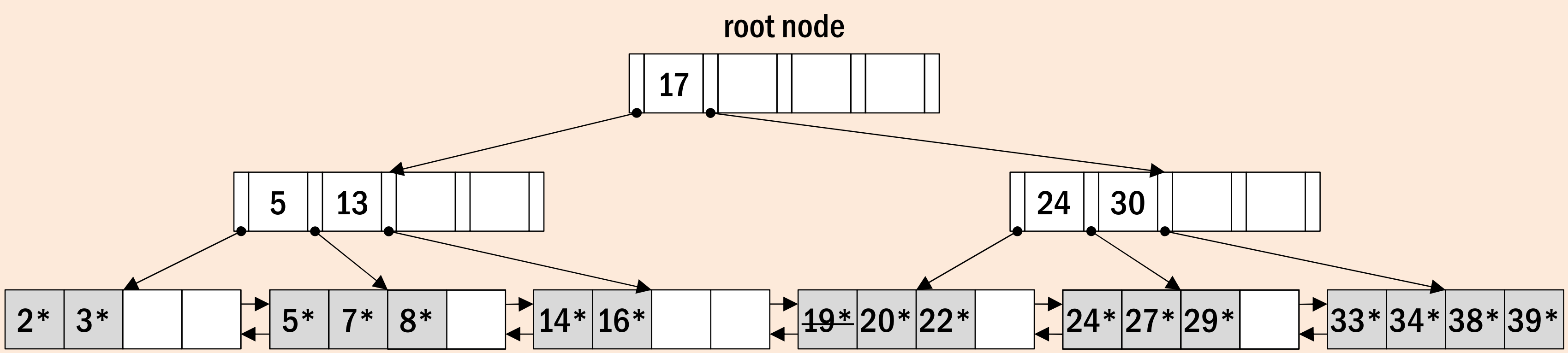
B+ Tree Delete

- Two techniques to handle the case that number of entries *m* of a node *n* falls under the **minimum occupancy threshold** *d*
- Redistribution
  - redistribute entries between *n* and an adjacent siblings
  - update parent to reflect redistribution: **change entry** pointing to second node to lowest search key in second node
- Merge
  - merge node *n* with an adjacent sibling
  - update parent to reflect merge: **delete entry** pointing to second node
  - if last entry in root is deleted, the height of the tree decreases by 1

B+ Tree Delete

Example: Deletion from a B+ tree with order *d* = 2

1. delete record with key *k* = **`19`** (i.e., entry **`19*`**) from the following B+ tree

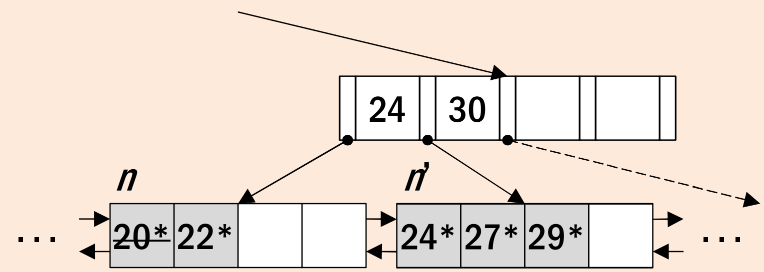


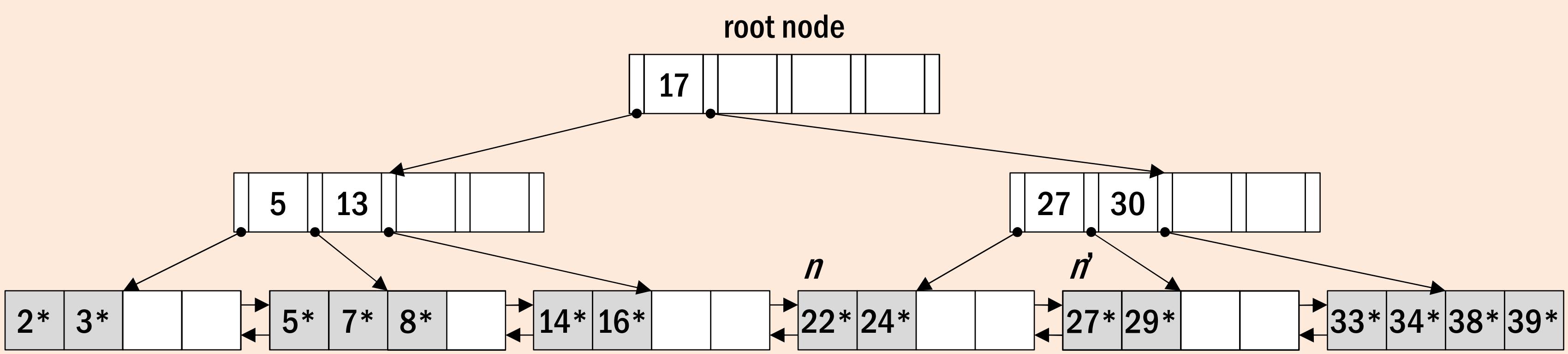
2. recursive tree traversal ends at leaf node *n* containing entries **`19*`**, **`20*`**, and **`22*`**
3. since *m* = 3 > 2, there is **no node underflow** in *n* after removal and entry **`19*`** can safely be deleted

B+ Tree Delete

Example: Deletion from a B+ tree with order *d* = 2 (cont'd)

4. subsequent deletion of record with key *k* = **`20`** (i.e., entry **`20*`**) results in **underflow** of node *n* as it already has minimal occupancy *d* = 2
5. since the (only) sibling *n'* of *n* has 3 > 2 entries (**`24*`**, **`27*`**, and **`29*`**), **redistribution** can be used to deal with the underflow of *n*
6. move entry **`24*`** to *n* and copy up the **new splitting key `27`**, which is the new smallest key value on *n'*

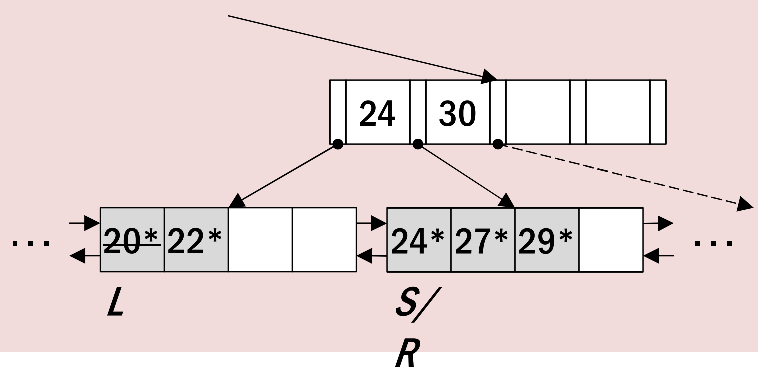




B+ Tree Delete

Delete from B+ tree of degree *d* (leaf nodes: redistribution)

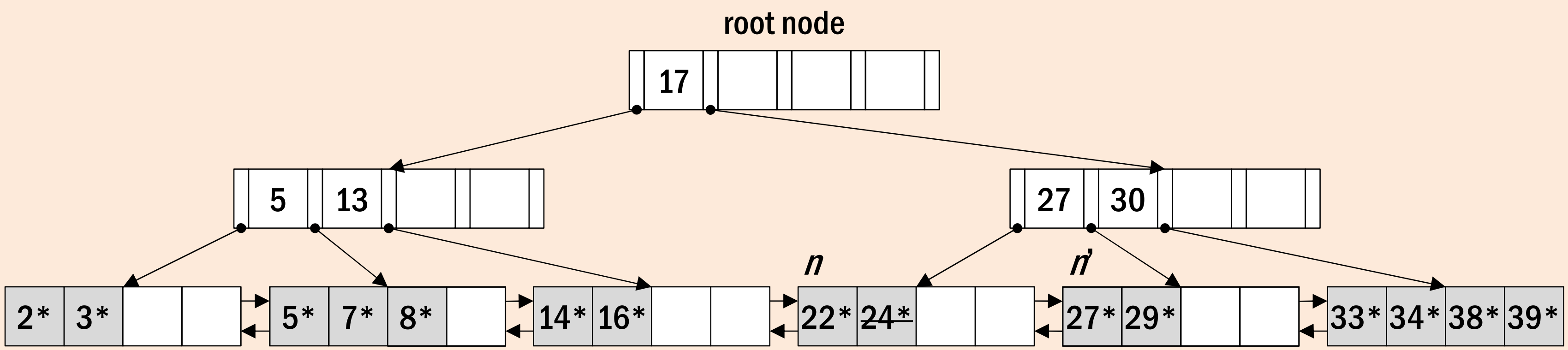
```
function delete( ↑parent, ↑node, k* ) : ↑oldChild
   if node is a non-leaf node, say N then . . .
   if node is a leaf node, say L then
      if L has entries to spare then remove k*; ↑oldChild ← null; return;
      else get a sibling S of L;
         if S has extra entries then
            redistribute entries evenly between L and S;
            find entry for right node, say R, in parent;
            replace key value in parent by new low-key value in R;
            ↑oldChild ← null; return;
         else . . .
endproc;
```



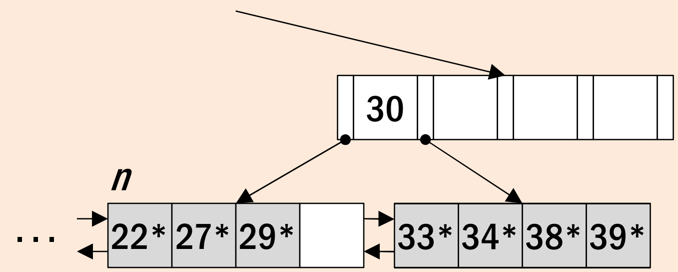
B+ Tree Delete

Example: Deletion from a B+ tree with order *d* = 2 (cont'd)

7. suppose record with key *k* = **24** (i.e., entry **24***) is deleted next



8. again, leaf-node *n* **underflows** as it only contains 1 < 2 entries after deletion
9. redistribution is **not an option** as (only) sibling *n'* of *n* just contains two entries (**27*** and **29***)
10. together *n* and *n'* contain 3 > 2 entries and can therefore be **merged**: move entries **27*** and **29*** from *n'* to *n*, then delete node *n'*



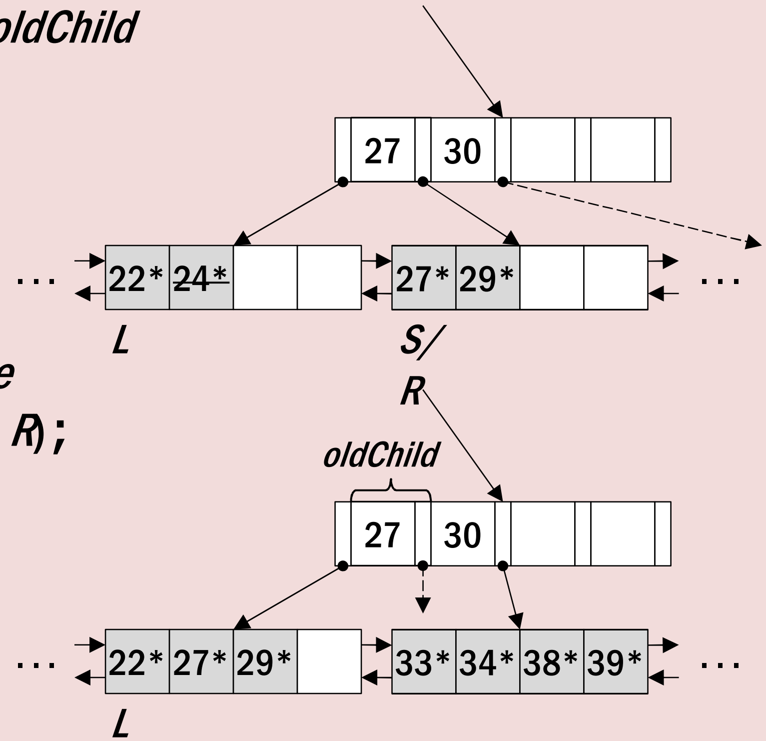
11. note that separator **27** between *n* and *n'* is no longer needed and therefore **discarded** (**recursively deleted**) from parent

Slides Credit: Michael Grossniklaus – Uni-Konstanz

B+ Tree Delete

Delete from B+ tree of degree *d* (leaf nodes: merge)

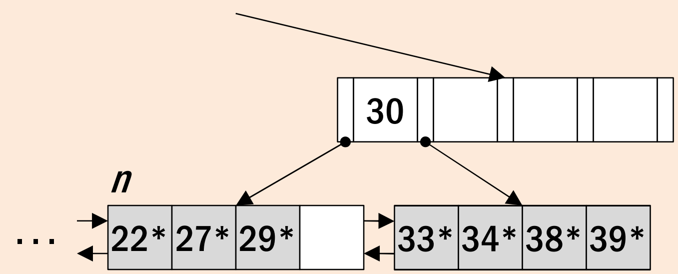
```
function delete( ↑parent, ↑node, k*) : ↑oldChild
  if node is a non-leaf node, say N then . . .
  if node is a leaf node, say L then
    if L has entries to spare then . . .
    else . . .
      if S has extra entries then . . .
      else merge L and S, let R be the right node
        ↑oldChild ← @(current entry in parent for R);
        move all entries from R to node on left;
        discard empty node R;
        adjust sibling pointers;
        return;
endproc;
```



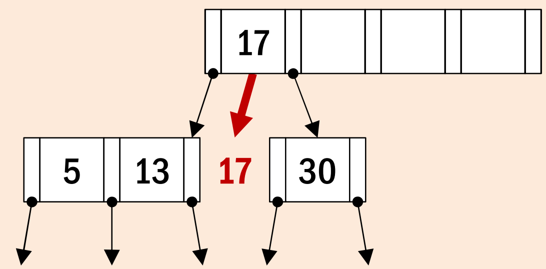
B+ Tree Delete

Example: Deletion from a B+ tree with order $d = 2$ (cont'd)

12. now, parent of n **underflows** as it only contains $1 < 2$ entries after deletion of entry $\langle$**27**, $\uparrow n'\rangle$
13. redistribution is **not an option** as its sibling just contains two entries (**5** and **13**)
14. therefore, **merge the nodes** into a new node with $\underbrace{d}_{left} + \underbrace{(d - 1)}_{right}$ keys and $\underbrace{d + 1}_{left} + \underbrace{d}_{right}$ pointers

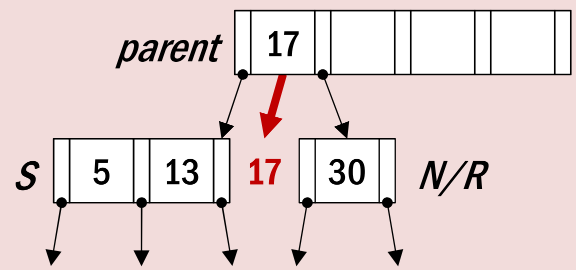


15. since a complete node needs to contain $2d$ keys and $2d + 1$ pointers, a **key value is missing**
16. missing key value is **pulled down** (i.e., deleted) from the parent to complete the merged node



B+ Tree Delete

Delete from B+ tree of degree *d* (non-leaf nodes: merge)

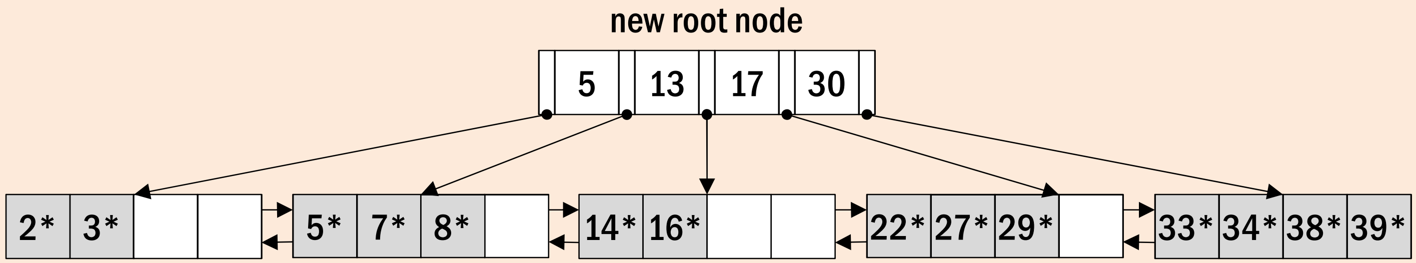



```
function delete(↑parent, ↑node, k*) : ↑oldChild
  if node is a non-leaf node, say N then
    find i such that k_i ≤ k < k_{i+1};
    ↑oldChild = delete(p_i, k);
    if ↑oldChild is null then return;
    else
      remove oldChild from N;
      if N has entries to spare then ↑oldChild ← null; return;
      else get a sibling S of N, using ↑parent
        if S has extra entries then . . .
        else merge L and S, let R be the right node
          ↑oldChild ← @(current entry in parent for R);
          pull splitting key from parent down into left node;
          move all entries from R to left node; discard empty node R; return;
  if node is a leaf node, say L then . . .
endproc;
```

B+ Tree Delete

Example: Deletion from a B+ tree with order *d* = 2 (cont'd)

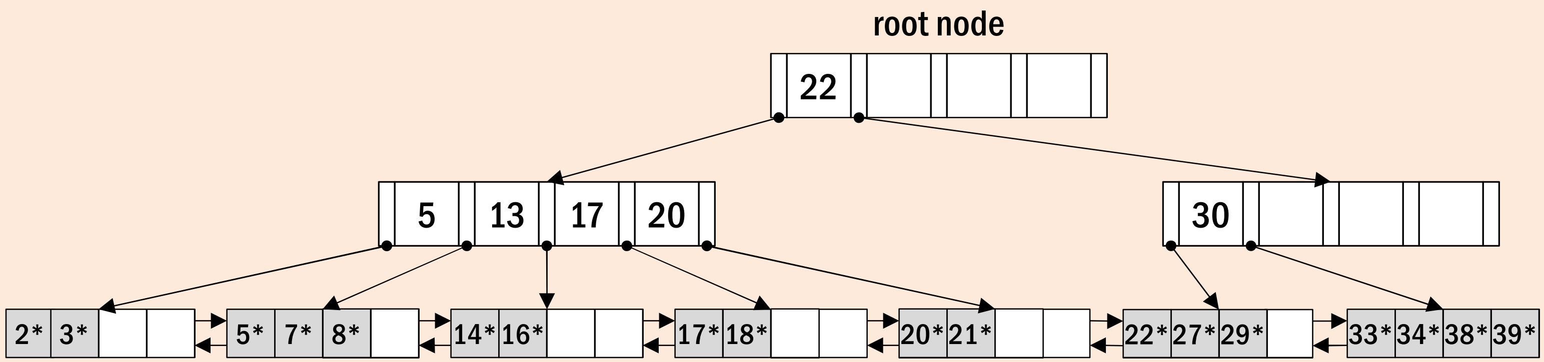
17. since the last remaining entry in the root was discarded, the merged node becomes the **new root**



B+ Tree Delete

Example: Deletion from a B+ tree with order *d* = 2 (cont'd)

18. suppose the following B+ tree is encountered **during** deletion

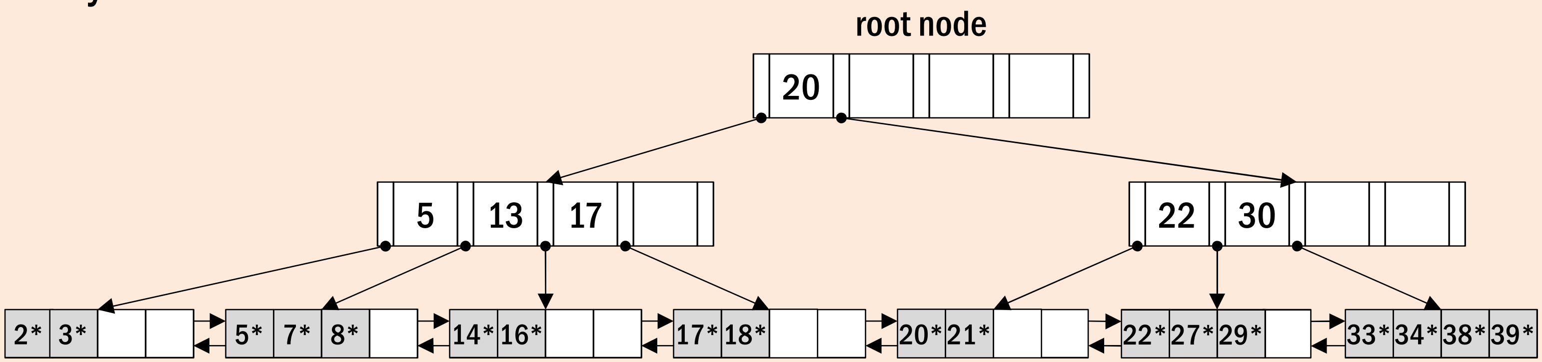


19. notice that the non-leaf node with entry `30` underflows
20. but its (only) sibling has two entries (`17` and `20`) to spare

B+ Tree Delete

Example: Deletion from a B+ tree with order *d* = 2 (cont'd)

21. redistribute entries by "**rotating**" entry `20` through the parent and pushing former parent entry `22` down



B+ Tree Delete

Delete from B+ tree of degree *d* (non-leaf nodes: redistribution)

```
function delete(↑parent, ↑node, k*) : ↑oldChild
  if node is a non-leaf node, say N then . . .
    if ↑oldChild is null then . . .
    else . . .
      if N has entries to spare then . . .
      else get a sibling S of N, using ↑parent
        if S has extra entries then
          redistribute entries evenly between N and S through parent;
          ↑oldChild ← null; return;
        else . . .
  if node is a leaf node, say L then . . .
endproc;
```

Merge and Redistribution Effort

- Actual DBMS implementations often avoid the cost of merging and/or redistribution by relaxing the minimum occupancy rule

B+ tree deletion in DB2

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

Slide Credit: Torsten Grust, University of Tübingen, Germany

B+ Trees and Duplicates

- As discussed here, B+ tree `search`, `insert`, (and `delete`) procedures ignore the presence of **duplicate** key values
- This assumption is often reasonable
  - 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

Treatment of duplicate keys in DB2

Since duplicate keys add to the B+ tree complexity, IBM DB2 **forces uniqueness** by forming a composite key of the form ⟨*k*, *id*⟩, where *id* is the unique tuple identity of the data record with key *k*

Tuple identities are

1. **system-maintained** unique identifiers for each tuple in a table
2. **not** dependent on tuple order
3. **immutable**

Slide Credit: Torsten Grust, University of Tübingen, Germany

B+ Trees and Duplicates

Other approaches alter the B+ tree implementation to add real support for duplicates

1. Use variant □ to represent the index data entries $k*$

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

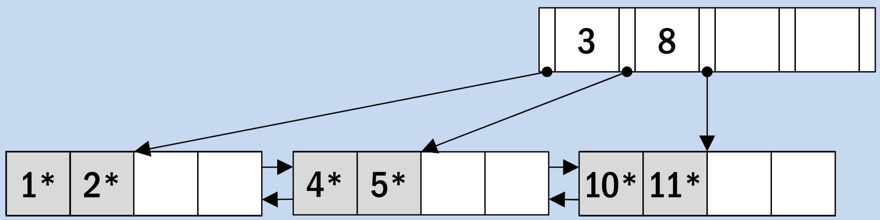
   – each duplicate record with key field *k* makes the list of *rids* grow
   – key *k* is not repeated stored, which saves space
   – B+ tree search and maintenance routines largely unaffected
   – index data entry size varies, which affect the B+ tree **order** concept
   – implemented, for example, in IBM Informix Dynamic Server

2. Treat duplicate value like any other value in the `insert` and `delete` procedures
   – doing so affects the `search` procedure
   – see example on following slides

B+ Trees and Duplicates

- **Example: Impact on duplicate insertion on `search(`*k*`)`**

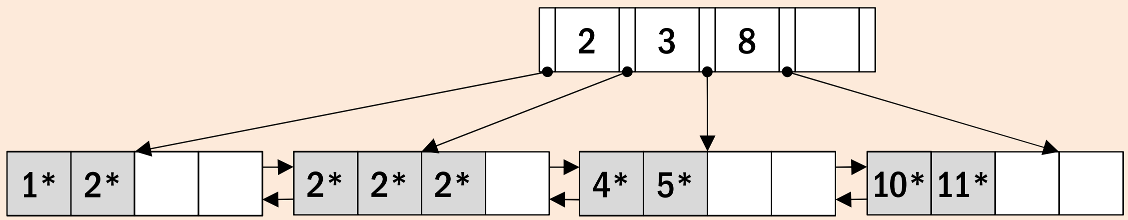
Insert **three records** with key $k = 2$ into the following B+ tree of order $d = 2$ (without using redistribution)



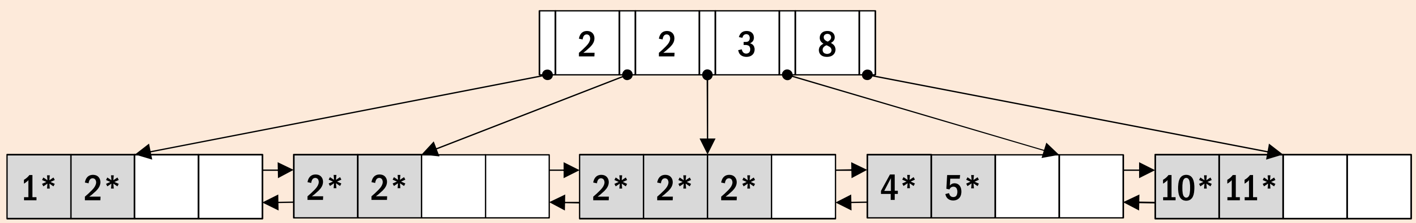
B+ Trees and Duplicates

Example: Impact on duplicate insertion on `search(`*k*`)`

Below the B+ tree that results from the exercise on the previous slide is shown



The same B+ tree after inserting **another two records** with key $k = 2$, is shown below



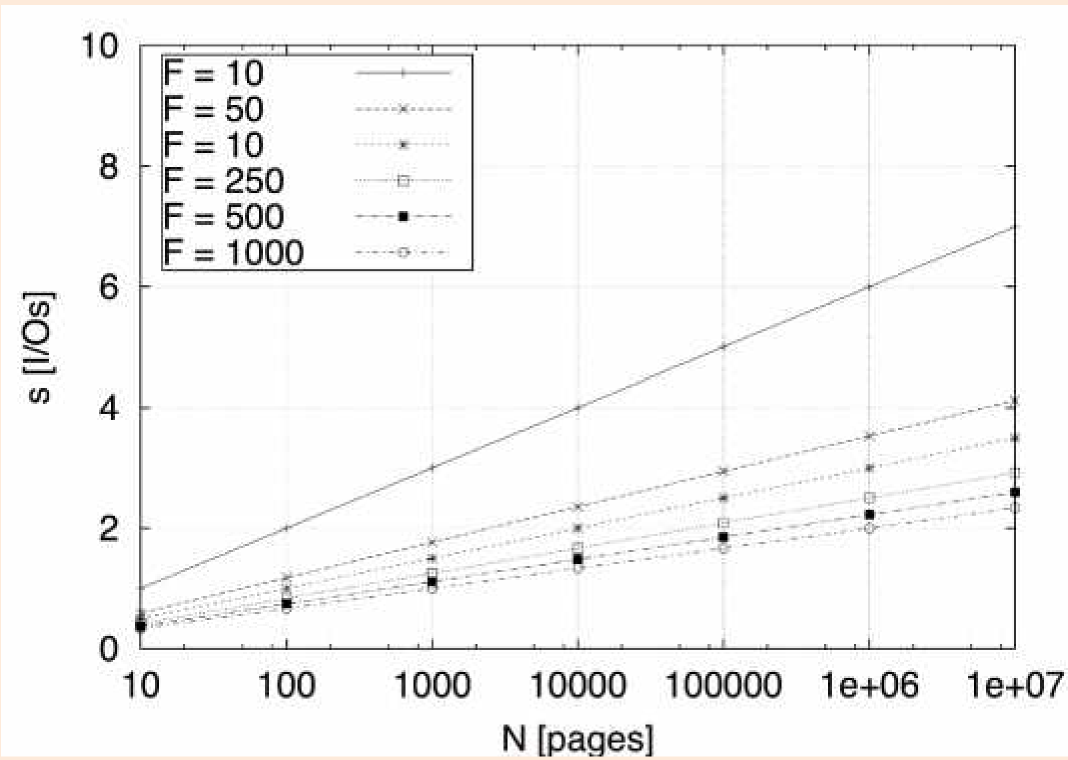
`search(`*k*`)`

- **non-leaf nodes**: follow the left-most node pointer p_i, such that $k_i \leq k \leq k_{i+1}$
- **leaf nodes**: also check right sibling (and its right sibling and its right sibling and...)

Key Compression in B+ Trees

- Recall that the **fan-out** *F* is a deciding factor in the search I/O effort *s* in an ISAM or B+ tree for a file of *N* pages: $s = \log_F N$

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+ tree implementation

Key Compression in B+ Trees

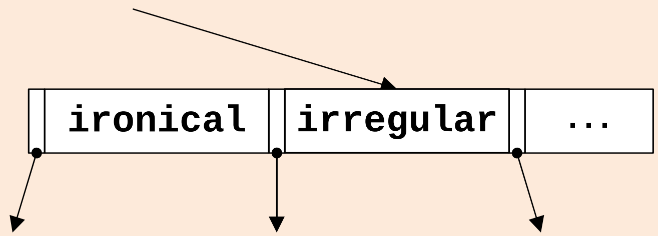
- Index entries in non-leaf B+ tree nodes are pairs $\langle k_i, \uparrow p_i \rangle$
  - **size of page pointers** depends on pointer representation of DBMS or hardware specifics
  - $|\uparrow p_i| \ll |k_i|$, especially for key field types like **`CHAR(·)`** or **`VARCHAR(·)`**
- To **minimize key size**, recall that key values in non-leaf nodes only direct calls to the appropriate leaf pages
  - actual key values are **not** needed
  - **arbitrary values** could be chosen as long as the separator property is maintained
  - for text attributes, a good choice can be **prefixes** of key values

Excerpt of `search(`k`)`

```
if k < k_1 then ...
else
  if k ≥ k_2d then ...
  else
    find i such that k_i ≤ k < k_i+1;
    ...
```

Key Compression in B+ Trees

Example: Searching a B+ tree node with VARCHAR(·) keys

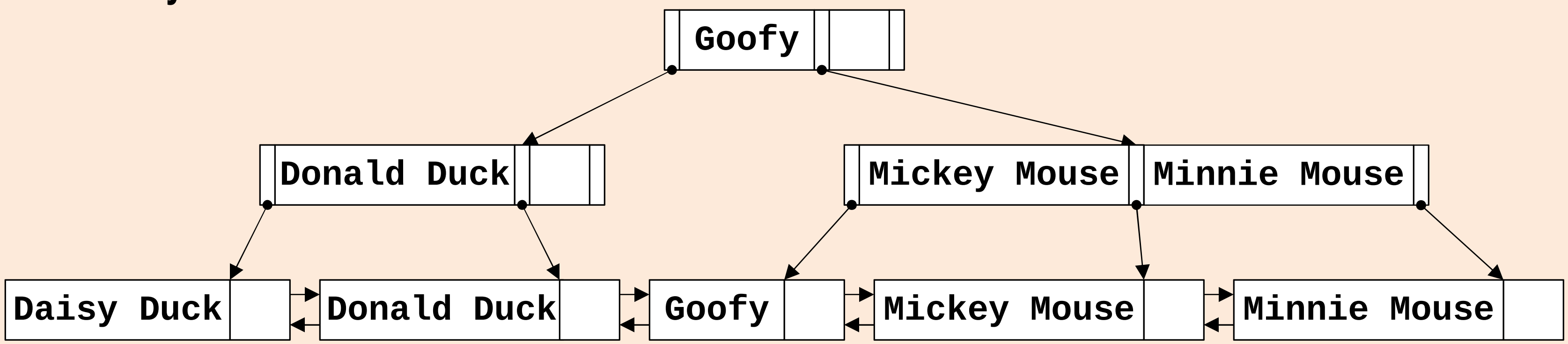


- To guide searches across this B+ tree node, it is sufficient to store the **prefixes** `iro` and `irr`
- B+ tree semantics must be preserved
  - all index entries stored in the sub-tree left of `iro` have keys $k <$ `iro`
  - all index entries stored in the sub-tree right of `iro` have keys $k \geq$ `iro` (and $k <$ `irr`)

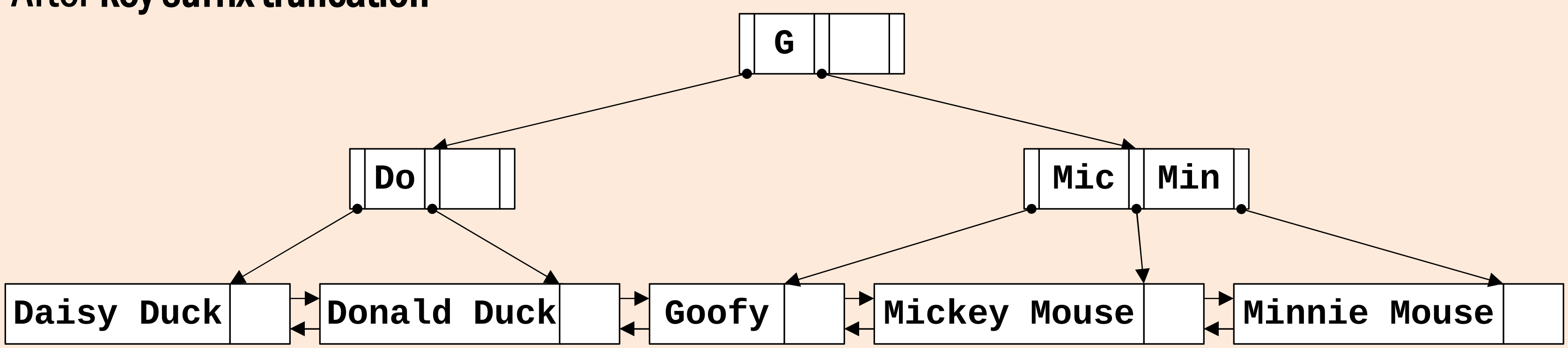
B+ Tree Key Suffix Truncation

Example: Key suffix truncation in a B+ tree with order *d* = 1

Before **key suffix truncation**



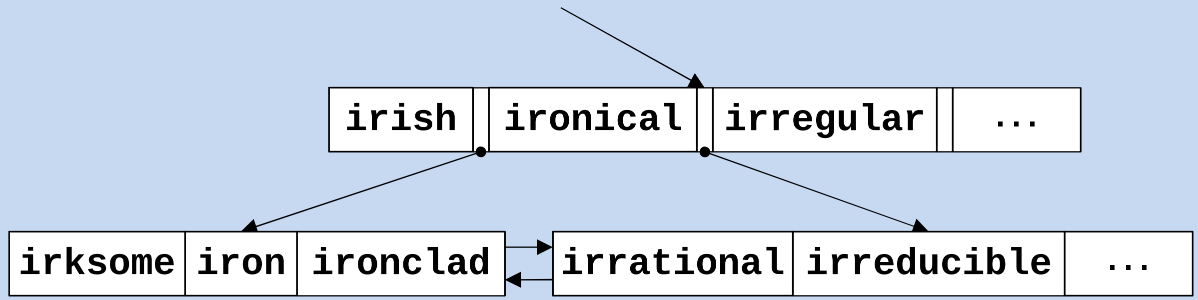
After **key suffix truncation**



B+ Tree Key Suffix Truncation

Key suffix truncation

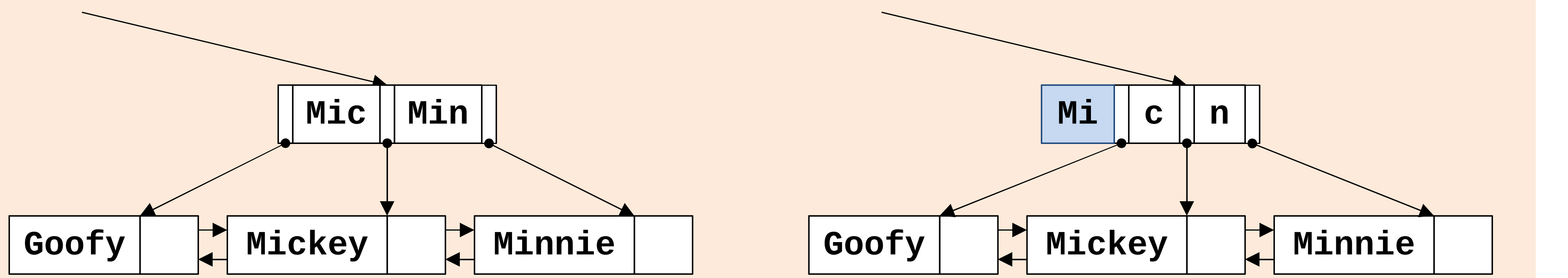
How would a B+ tree key compressor alter the key entries in the non-leaf node of this B+ tree excerpt?



Key Prefix Compression in B+ Trees

- Keys within a B+ tree node often share a **common prefix**

Example: Shared key prefixes in non-leaf B+ tree nodes



- **Key prefix compression**
  - store common prefix only once (e.g., as "key" k_0)
  - keys have become highly discriminative now
- Violating the 50% occupancy rule can help to improve the effectiveness of prefix compression

B+ Tree Bulk Loading

Database log: table and index creation

```
CREATE TABLE t1 (id INT, text VARCHAR(10));
```

... insert 1,000,000 rows into table `t1` ...

```
CREATE INDEX t1_idx ON t1 (id ASC);
```

- Last SQL command initiates one million B+ tree `insert(·)` calls, a so-called index **bulk load**
- DBMS will traverse the growing B+ tree index from its root down to the leaf pages one million times

This is bad...

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

B+ Tree Bulk Loading

- Most DBMS provide a B+ tree **bulk loading utility** to reduce the cost of operations like the one on the previous slide

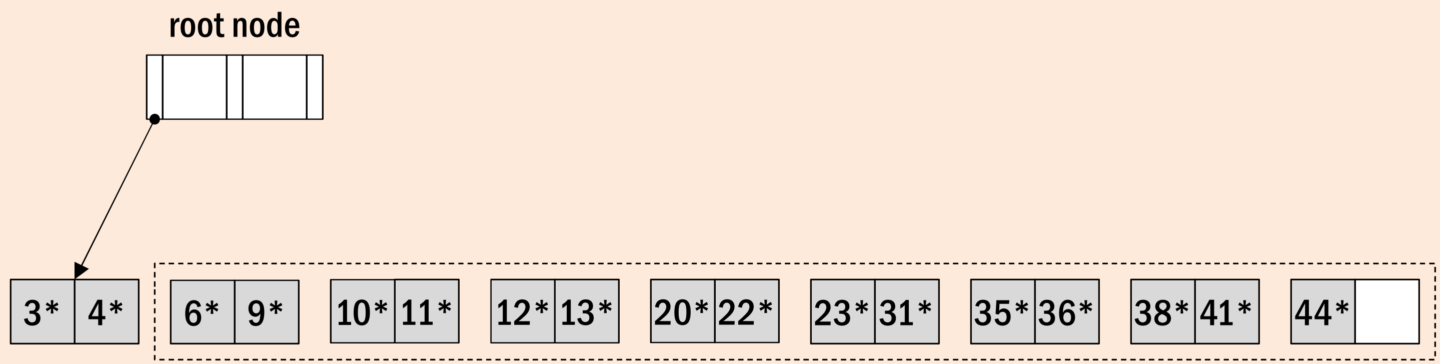
B+ tree bulk loading algorithm

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 □ or □ this does **not** imply to sort the data file itself (variant □ effectively creates a clustered index)

2. allocate an **empty index root node** and let its p_0 node pointer point to the first page of the sorted *k** entries

B+ Tree Bulk Loading

Example: State of bulk load of a B+ tree with order *d* = 1 after Step 2



Index leaf pages that are not yet in the B+ tree are **framed**

B+ tree bulk loading continued

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

B+ Tree Bulk Loading

- As the *k** are sorted, any insertion will hit the **right-most index node** (just above the leaf level)
- A specialized `bulk_insert(·)` procedure avoids B+ tree root-to-leaf traversals altogether

B+ tree bulk loading algorithm (cont'd)

3. for each leaf level node *n*, insert the index entry

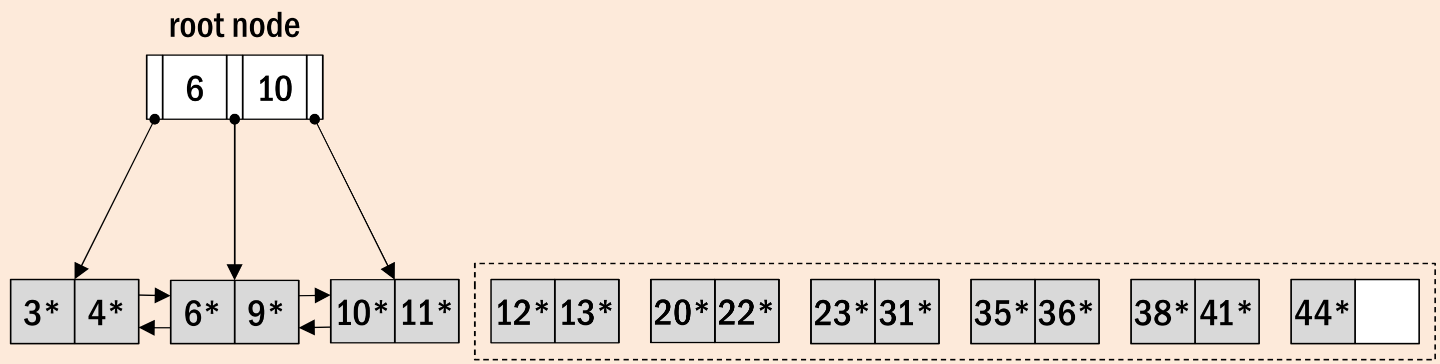
   ⟨minimum key on *n*, ↑ *n*⟩

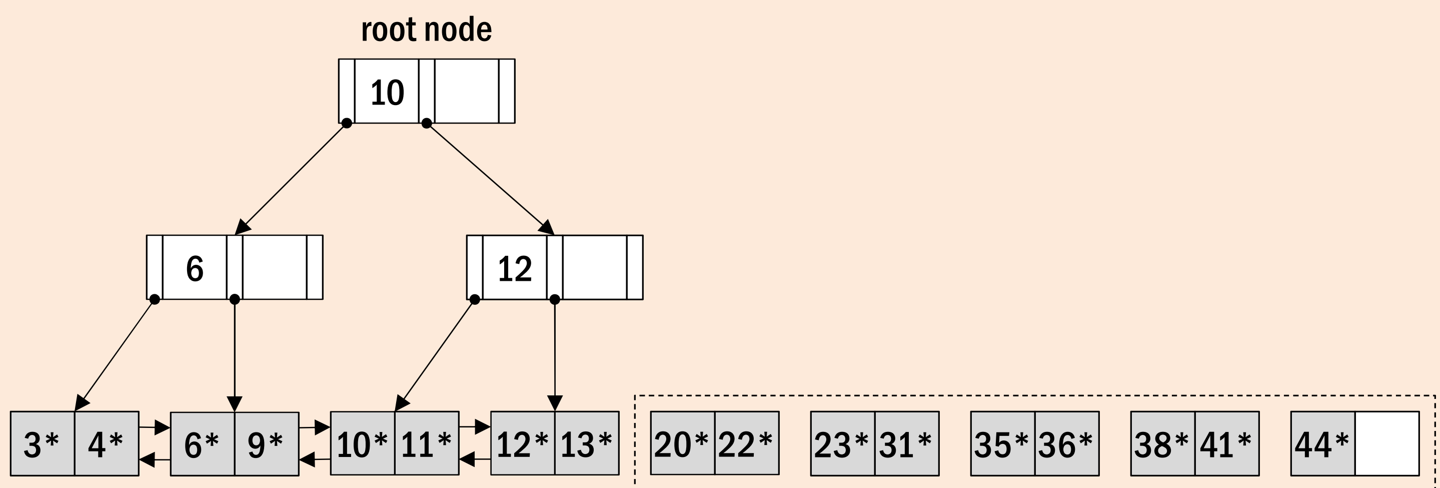
   into the right-most index node just above the leaf level

↳ the right-most node is filled **left-to-right**, splits only occur on the **right-most paths** from the leaf level up to the root

B+ Tree Bulk Loading

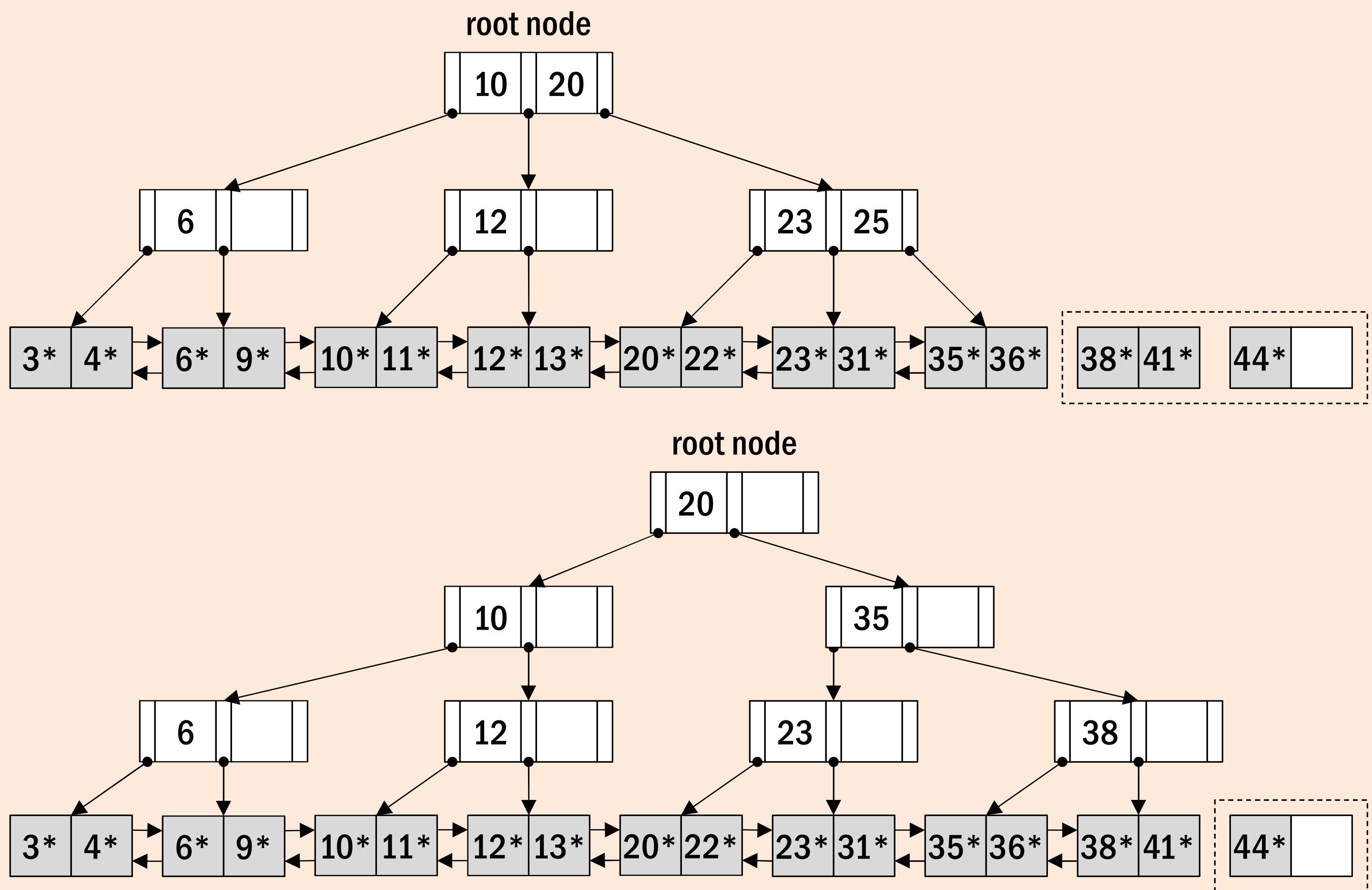
Example: State of bulk load of a B+ tree with order $d = 1$ (cont'd)





B+ Tree Bulk Loading

Example: State of bulk load of a B+ tree with order *d* = 1 (cont'd)



B+ Tree Bulk Loading

- Bulk-loading is more (time) efficient
  - tree traversals are saved
  - less page I/O operations are necessary, i.e., buffer pool is used more effectively
- As seen in the example, bulk-loading is also more space-efficient as all leaf nodes are filled up completely

Space efficiency of bulk-loading

How would the resulting tree in the previous example look like, if you used the standard `insert(·)` routine on the sorted list of index entries *k**?

B+ Tree Bulk Loading

- Bulk-loading is more (time) efficient
  - tree traversals are saved
  - less page I/O operations are necessary, i.e., buffer pool is used more effectively
- As seen in the example, bulk-loading is also more space-efficient as all leaf nodes are filled up completely

Space efficiency of bulk-loading

How would the resulting tree in the previous example look like, if you used the standard `insert(·)` routine on the sorted list of index entries $k*$?

↳ inserting sorted data into a B+ tree yields minimum occupancy of (only) d entries in all nodes

A Note on B+ Tree Order

- Recall that B+ tree definition uses the concept of **order** *d*
- Order concept is useful for presenting B+ tree algorithms, but is hardly every used in practical implementations
  - key values may often be of variable length
  - duplicates may lead to variable number of rids in an index entry k* according to variant □
  - leaf and non-leaf nodes may have different capacities due to index entries of variant □
  - key compression may introduce variable-length separator values
- Therefore, the order concept is relaxed in practice and replaced with a physical space criterion, e.g., every node needs to be at least half-full

A Note on Clustered Indexes

- Recall that a clustered index stores actual data records inside the index structure (variant entries)
- In case of a B+ tree index, splitting and merging leaf nodes **moves data records** from one page to another
  - depending on the addressing scheme used, *rid* of a record may change if it is moved to another page
  - even with the TID addressing scheme (records can be moved within a pages, uses forwarding address to deal with moves across pages), the performance overhead may be intolerable
  - some systems use the search key of the clustered index as a (location independent) record address for other, non-clustered indexes in order to avoid having to update other indexes or to avoid many forwards

B+ Tree Invariants

- **Order**: d
- **Occupancy**
  - each non-leaf node holds at least d and at most $2d$ keys (exception: root node can hold at least 1 key)
  - each leaf node holds between d and $2d$ index entries
- **Fan-out**: each non-leaf node holding m keys has $m + 1$ children
- **Sorted order**
  - all nodes contain entries in ascending key-order
  - child pointer p_i $(1 \leq i < m)$ if an internal node with m keys $k_1, \ldots, k_m$ leads to a sub-tree where all keys k are $k_i \leq k < k_{i+1}$
  - p_0 points to a sub-tree with keys $k < k_1$ and p_m to a sub-tree with keys $k \geq k_m$
- **Balance**: all leaf nodes are on the same level
- **Height**: $\log_F N$
  - N is the total number of index entries/record and F is the average fan-out
  - because of high fan-out, B+ trees generally have a low height

Database System Architecture and Implementation

TO BE CONTINUED...

Indexes

- If the basic organization of a file does not support a specific operation, we can **additionally** maintain an auxiliary structure, an **index**, which adds the needed support

Example

```
SELECT A, B, C
FROM   R
WHERE  A > 0 AND A < 100
```

If the file for table **R** is sorted on **C**, it **cannot** be used to evaluate *Q* more efficiently. A solution is to add an index *that supports range queries* on **A.**

Indexes

- A DBMS uses indexes like **guides**, where each guide is specialized to accelerate searches on a specific attribute (or a combination of attributes) of the records in its associated file

Usage of index on attribute A

$k \longrightarrow \boxed{\text{index}} \longrightarrow k^* \longrightarrow \langle \ldots, \mathbf{A} = k, \ldots \rangle$

1. Query index for the location of a record with **A** = *k* (*k* is the **search key**)
2. The index responds with an associated **index entry** *k**
   (*k** contains enough information to access the actual record in the file)
3. Read the actual record by using the guiding information in *k**: the record will have an **A**-field with value *k*.

The Small Print: Only "exact match" indexes will return records that contain the value *k* for field **A**. In the more general case of "similarity" indexes, the records are not guaranteed to contain the value *k*, they are only candidates for having this value.

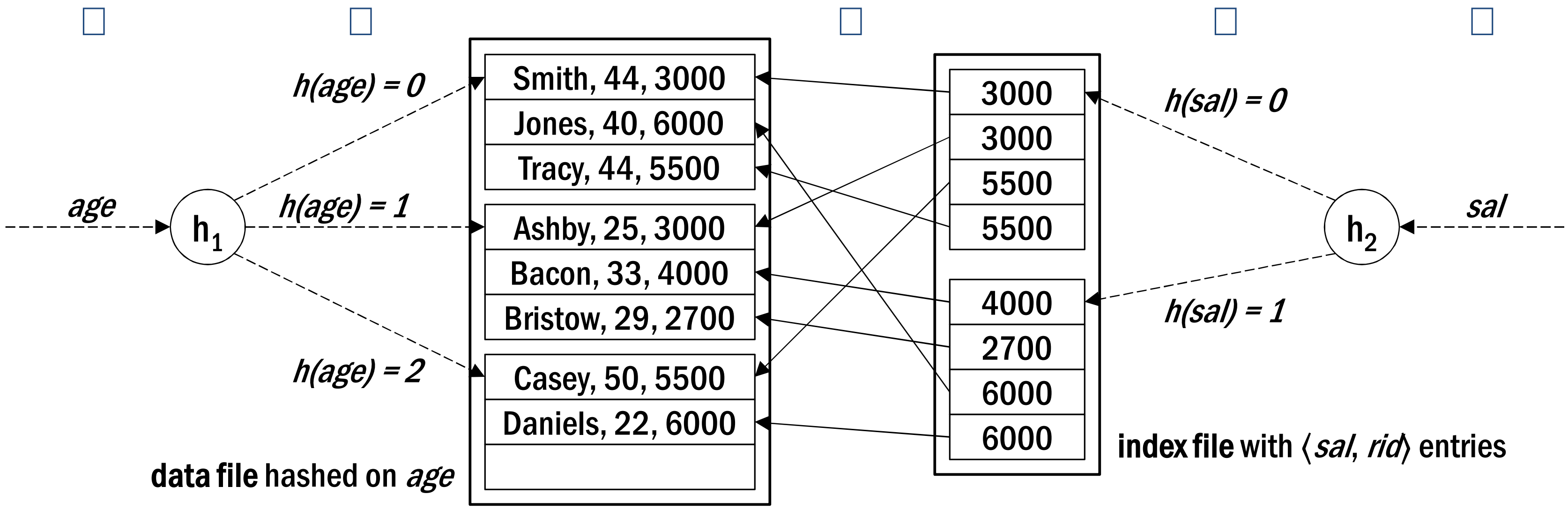
Index Entries

Index Entry Design

Variant	Index entry k^*
□	$\langle k, \langle \ldots, \mathrm{A} = k, \ldots \rangle\rangle$
□	$\langle k, rid \rangle$
□	$\langle \mathrm{k}, [rid_1, rid_2, \ldots] \rangle$

- Remarks
  - With variant □, there is no need to store the data records in addition to the index—the index itself is a special file organization
  - If we build multiple indexes for a file, at most one of these should use variant □ to avoid redundant storage of records
  - Variants □ and □ use *rid*(s) to point into the actual data file
  - Variant □ leads to fewer index entries if multiple records match a search key *k*, but index entries are of variable length

Index Example



- **Data file** contains ⟨*name*, *age*, *sal*⟩ records and is hashed on age, using hash function h_1 (index entry variant □)
- **Index file** contains ⟨*sal*, *rid*⟩ index entries (variant □), pointing to data file (hash function h_2)
- This file organization plus index **efficiently** supports equality searches on both key *age* and key *sal*

Clustered vs. Unclustered Indexes

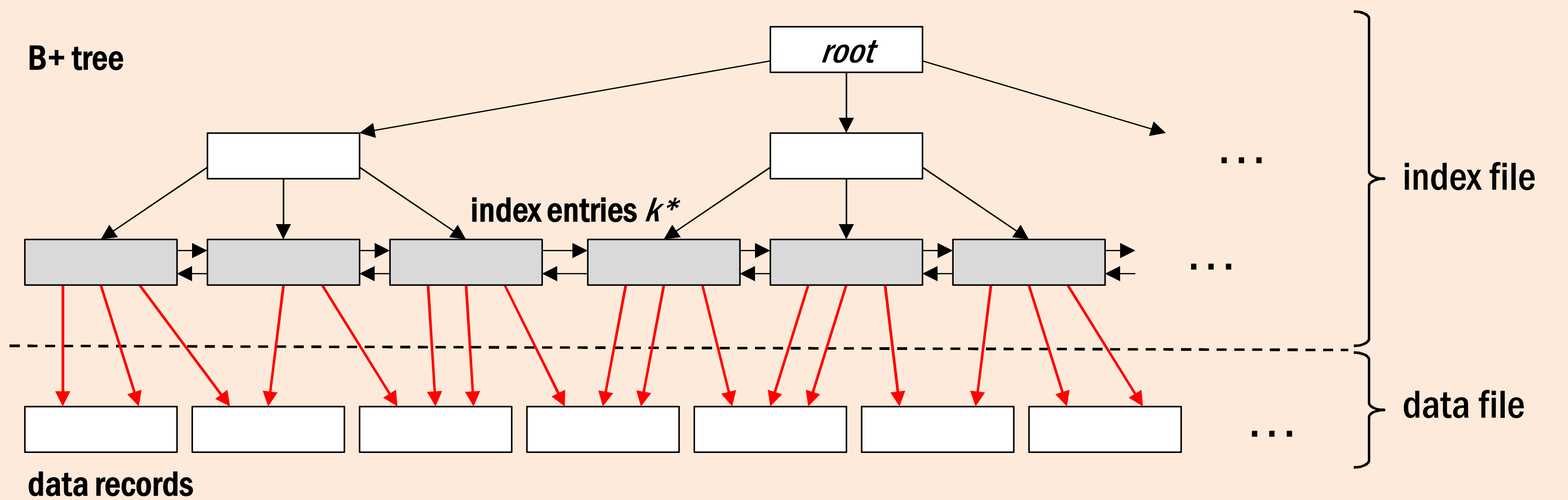
- Suppose, **range selections** such as *lower* ≤ **A** ≤ *upper* need to be supported on records of a data file
- If an index on field **A** is maintained, range selection queries could be evaluated using the following algorithm
  1. **query the index** *once* for a record with field **A** = *lower*
  2. **sequentially scan the data file** from there until we encounter a record with field **A** > *upper*

Name the assumption!

Which important assumption does the above algorithm make in order for this **switch from index to data file** to work efficiently?

Clustered vs. Unclustered Indexes

Index over data file with matching sort order



- Remark
  - in a B+ tree, for example, the index entries *k** stored in the leaves are sorted on the key *k*

Clustered vs. Unclustered Indexes

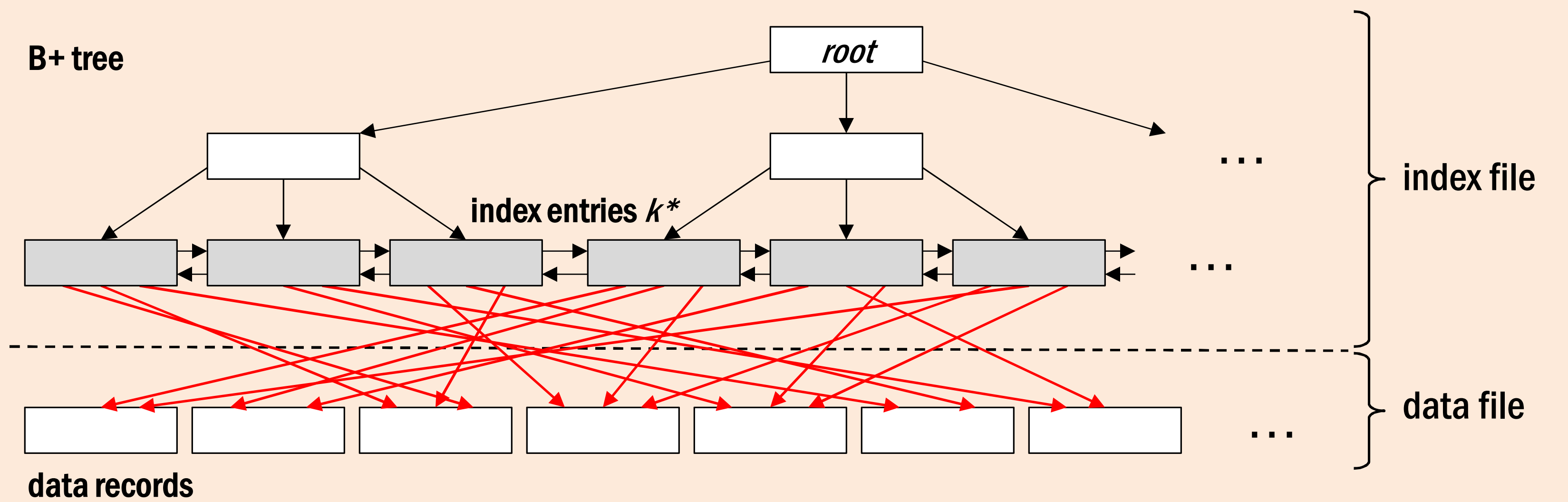
Definition: Clustered Index

If the **data file** associated with an index **is sorted on the index search key**, the index is said to be **clustered**

- In general, the cost for a range selection grows tremendously if the index on **A** is **unclustered**
  - proximity of index entries **does not** imply proximity of data records
  - as before, the index can be queried for a record with **A** = *lower*
  - however, to continue the scan it is necessary to **revisit the index entries**, which point to **data pages scattered** all over the data file
- Remarks
  - an index that uses entries $k*$ of variant □, is clustered by definition
  - a data file can have at most one clustered index (but any number of unclustered indexes)

Clustered vs. Unclustered Indexes

Unclustered index



Clustered vs. Unclustered Indexes

Variant in Oracle 8i

```
CREATE TABLE … ( … PRIMARY KEY ( … )) ORGANIZATION INDEX;
```

Clustered indexes in DB2

Create a clustered index **`IXR`** on table **R**, index key is attribute **A**

```
CREATE INDEX IXR ON R(A ASC) CLUSTER;
```

From the DB2 V9.5 manual

> *"[***`CLUSTER`***] specifies that the index is* ***the*** *clustering index of the table. The* ***cluster factor*** *of a clustering index is maintained or improved dynamically as data is inserted into the associated table, by attempting to* ***insert new rows physically close to the rows for which the key values of this index are in the same range****. Only one clustering index may exist for a table so* **`CLUSTER`** *may not be specified if it was used in the definition of any existing index on the table (SQLSTATE 55012). A clustering index may not be created on a table that is defined to use append mode (SQLSTATE 428D8)."*

Clustered vs. Unclustered Indexes

Cluster a table based on an existing index in PostgreSQL

Reorganize the rows of table **R** so that their physical order matches the *existing* index **`IXR`**

```
CLUSTER R USING IXR;
```

- If **`IXR`** indexes attribute **A** of **R**, rows will be sorted in ascending **A** order
- Range queries will touch less pages, which additionally, will be physically adjacent
- **Note**: Generally, future insertions will compromise the perfect **A** order
  - may issue **`CLUSTER R`** again to re-cluster
  - in **`CREATE TABLE`**, use **`WITH(fillfactor =`** f **`)`**, $f \in$ **`10`**…**`100`**, to reserve space for subsequent insertions

- The SQL-92 and SQL-99 standard do not include any statement for the specification (creation, dropping) of index structures
  - SQL **does not even require** SQL systems to provide indexes at all!
  - almost all SQL implementations support one or more kinds of indexes

Slide Credit: Torsten Grust, University of Tübingen, Germany

Dense vs. Sparse Indexes

- Another advantage of a **clustered index** is the fact that it can be designed to be **space efficient**

Definition: Sparse Index

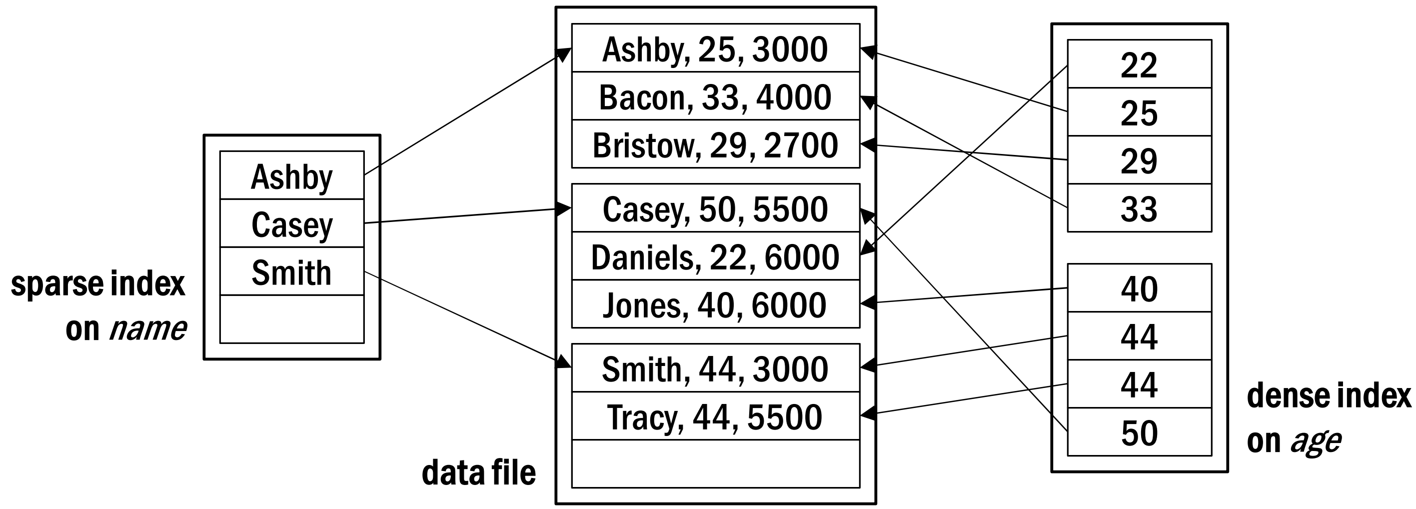
To keep the size of the index small, maintain **one index entry** $k*$ **per data file page** (not one index entry per data record). The key k is the **smallest key** on that page.

Indexes of this kind are called **sparse.** Otherwise indexes are referred to as **dense.**

Search a record with field A = k in a sparse A-index

1. Locate the largest index entry $k'*$ such that $k'* \leq k'$
2. Then access the page pointed to by $k'*$
3. Scan this page (and the following pages, if needed) to find records with $\langle \ldots, \mathbf{A} = k, \ldots \rangle$.
   Since the data file is clustered (i.e., sorted) on field **A**, matching records are guaranteed to be found in proximity

Dense vs. Sparse Index Example



- Again, the data file contains ⟨*name*, *age*, *sal*⟩ records
- Two indexes are maintained for the data file
  - **clustered sparse index** on field *name*
  - **unclustered dense index** on field *age*
- Both indexes use entry variant □ to point into the data file

Dense vs. Sparse Indexes

- Final remarks
  - sparse indexes need 2-3 orders of magnitude **less space** that dense indexes
  - it is **not possible** to build a sparse index that is unclustered (i.e., there is at most one sparse index per file)

SQL queries and index exploitation

How do you propose to evaluate the following SQL queries?

- ```
  SELECT MAX(age)
  FROM   employees
  ```
- ```
  SELECT MAX(name)
  FROM   employees
  ```

Primary vs. Secondary Indexes

Terminology

In the literature, there is often a distinction between **primary** (mostly used for indexes on the primary key) and **secondary** (mostly used for indexes on other attributes) indexes.

This terminology, however, is not very uniform and some text books may use those terms for different properties.

For example, some text books use **primary** to denote variant □ of indexes, whereas **secondary** is used to characterize the other two variants □ and □.

Multi-Attribute Indexes

- Each of the indexing techniques sketched so far can be applied to a **combination of attribute values** in a straightforward way
  - concatenate indexed attributes to form an index key,
    e.g., ⟨`lastname`, `firstname`⟩ → `searchkey`
  - define index on `searchkey`
  - index will support lookup based on both attribute values,
    e.g., … `WHERE lastname='Doe' AND firstname='John'` …
  - possibly, it will also support lookup based on a "prefix" of values,
    e.g., … `WHERE lastname='Doe'` …
- So-called **multi-dimensional indexes** provide support for **symmetric** lookups for all subsets of the indexed attributes
- Numerous such indexes have been proposed, in particular for geographical and geometric applications