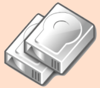
Chapter 4

Tree-Structured Indexing

ISAM and B$^+$-trees

Architecture and Implementation of Database Systems
Summer 2013

Torsten Grust
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen



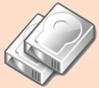
Ordered Files and Binary Search

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

```
SELECT *
FROM   CUSTOMERS
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`.



Ordered Files and Binary Search

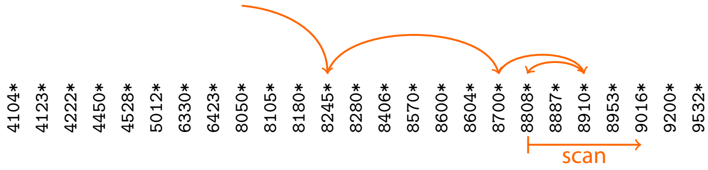
How could we prepare for such queries and evaluate them efficiently?

```sql
SELECT *
FROM   CUSTOMERS
WHERE  ZIPCODE BETWEEN 8880 AND 8999
```

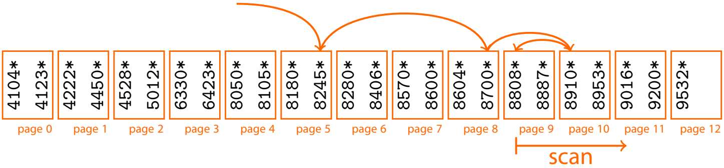
We could

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:

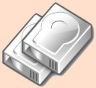


Ordered Files and Binary Search

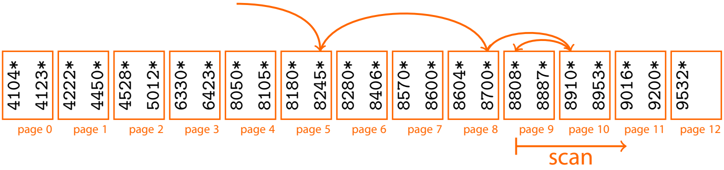


✔ We get **sequential access** during the **scan phase**.

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



Ordered Files and Binary Search

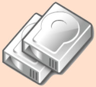


✔ We get **sequential access** during the **scan phase**.

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

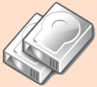
✗ We need to read about **as many pages** for this.

The whole point of binary search is that we make **far, unpredictable jumps**. This largely defeats page prefetching.



Tree-Structured Indexing

- 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^+-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 a subsequent chapter.)
- B^+-trees refine the idea underlying the rather static ISAM scheme and add efficient support for **insertions** and **deletions**.



Indexed Sequential Access Method (ISAM)

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).

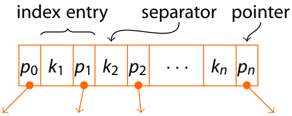
ISAM ...

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

Indexed Sequential Access Method (ISAM)

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:

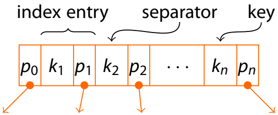


2. ISAM leads to **sparse** index structures. In an index entry

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

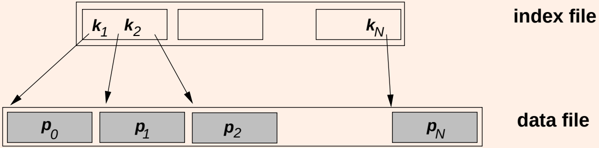
key k_i is the first (*i.e.*, the minimal) `A` value on the data file page pointed to by p_i (p_i: page no).

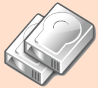
Indexed Sequential Access Method (ISAM)



- 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 $i = 2, \ldots, n$.
- We obtain a **one-level ISAM structure**.

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





Searching ISAM

SQL range selection on field A

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

To support **range selections**:

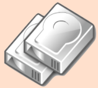
1. Conduct a **binary search on the index file** for a key of value *lower*.
2. Start a **sequential scan of the data file** from the page pointed to by the index entry (scan until field A exceeds *upper*).

Indexed Sequential Access Method ISAM

- 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

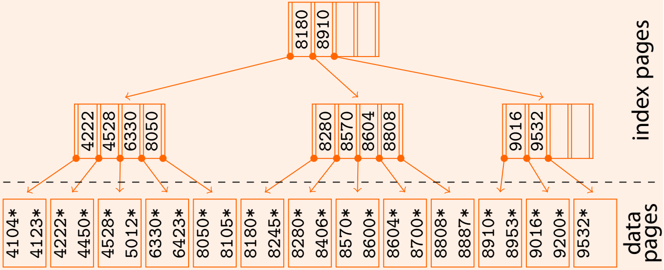
Recursively apply the index creation step: treat the topmost index level like the data file and add an additional index layer on top.
Repeat, until the the top-most index layer fits on a single page (the **root page**).



Multi-Level ISAM Structure

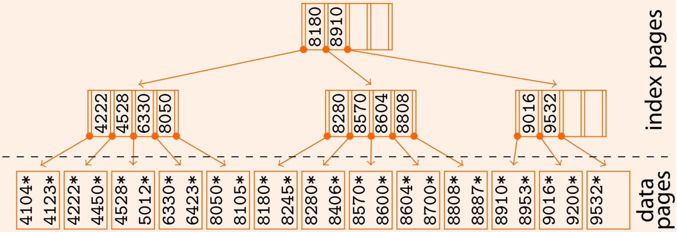
This recursive index creation scheme leads to a **tree-structured** hierarchy of index levels:

Multi-level ISAM structure



Multi-Level ISAM Structure

Multi-level ISAM structure



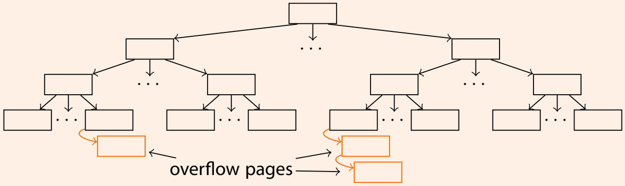
- Each ISAM tree node corresponds to one page (disk block).
- To create the ISAM structure for a given data file, proceed **bottom-up**:
  1. Sort the data file on the search key field.
  2. Create the index leaf level.
  3. If the top-most index level contains more than one page, repeat.

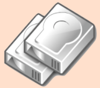
Multi-Level ISAM Structure: Overflow Pages

- 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

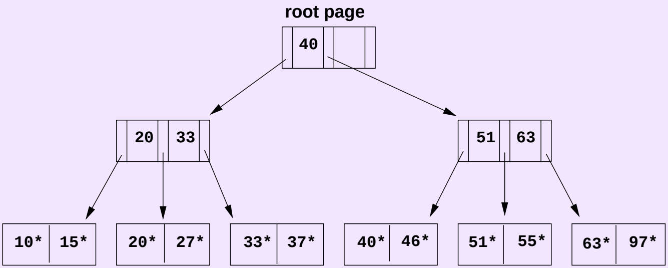




Multi-Level ISAM Structure: Example

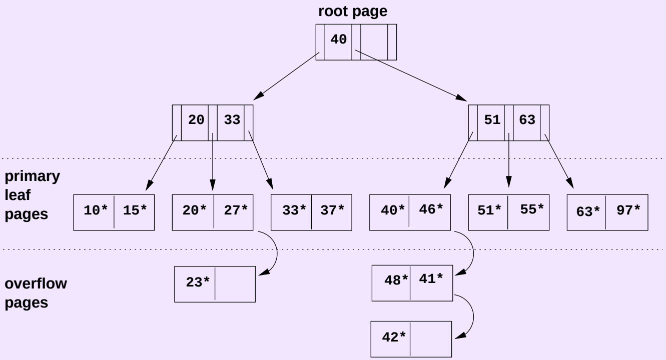
Eeach page can hold two index entries plus one (the left-most) page pointer:

Example (Initial state of ISAM structure)



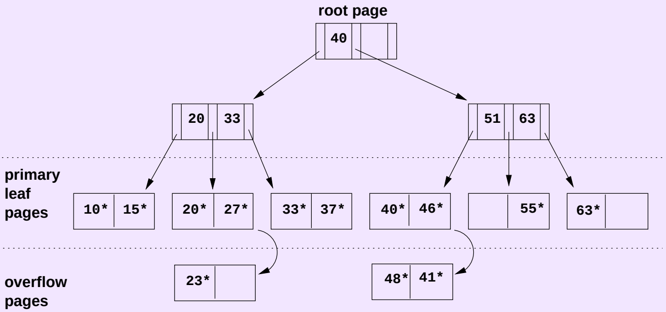
Multi-Level ISAM Structure: Insertions

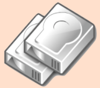
Example (After insertion of data records with keys 23, 48, 41, 42)



Multi-Level ISAM Structure: Deletions

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





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?

ISAM: Too Static?

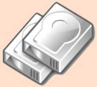
- The non-leaf levels of the ISAM structure have not been touched at all by the data file updates.
- This may lead to index key entries which do not appear in the index leaf level (*e.g.*, key value `51` on the previous slide).

✎ 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 preseve the **separator propery** of the index key entries, maintenance of overflow chains is required.

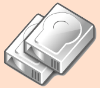
⇒ ISAM may **lose balance** after heavy updating. This complicates life for the query optimizer.



ISAM: Being Static is Not All Bad

- 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).

$\Rightarrow$ ISAM may be the index of choice for relatively static data.



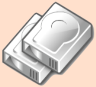
ISAM: Efficiency of Searches

- 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

$$N \cdot {}^{1}\!/\!{}_{F} \quad .$$



ISAM: Efficiency of Searches

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

$$N \cdot {}^{1}\!/_{F} \cdot {}^{1}\!/_{F} \cdots \quad .$$

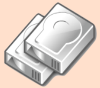
- Index searching ends after s steps when the search space has been reduced to size 1 (*i.e.*, we have reached the index leaf level and found the data page that contains the wanted record):

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

- 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.

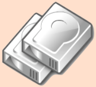


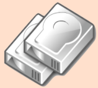
B^+-trees: A Dynamic Index Structure

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

- Search performance is only dependent on the **height** of the B^+-tree (because of high fan-out *F*, the height rarely exceeds 3).
- **No overflow chains** develop, a B^+-tree remains **balanced**.
- B^+-trees offer efficient **insert/delete procedures**, the underlying data file can grow/shrink **dynamically**.
- B^+-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$^+$-trees: Basics

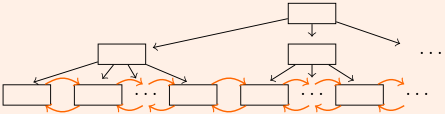
B$^+$-trees resemble ISAM indexes, where

- leaf nodes are connected to form a **doubly-linked list**, the so-called **sequence set**,[2]
- 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.

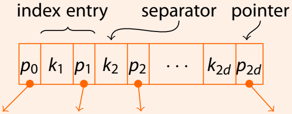
Sketch of B$^+$-tree structure (data pages not shown)



[2] This is not a strict B$^+$-tree requirement, although most systems implement it.

B$^+$-trees: Non-Leaf Nodes

B$^+$-tree inner (non-leaf) node



B$^+$-tree **non-leaf nodes** use the same internal layout as inner ISAM nodes:

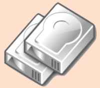
- The **minimum** and **maximum number of entries** n is bounded by the **order** d of the B$^+$-tree:

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

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

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

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



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.
- Note that we can use all index entry variants Ⓐ, Ⓑ, Ⓒ to implement the leaf entries:
  - For variant Ⓐ, the B^+-tree represents the index as well as the data file itself. Leaf node entries thus look like

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

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

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

B^+-tree: Search

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

B^+-tree search

Function: `search` (k)

return `tree_search` (k, root);

Function: `tree_search` (k, node)

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

  return *node*;

switch k **do**

  case $k < k_1$

    return `tree_search` (k, p_0);

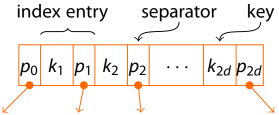
  case $k_i \leq k < k_{i+1}$

    return `tree_search` (k, p_i);

  case $k_{2d} \leq k$

    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.

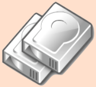


node page layout

B^+-tree: Insert

- Remember that B^+-trees remain **balanced**[3] no matter which updates we perform. Insertions and deletions have to preserve this invariant.

[3] All paths from the B^+-tree root to any leaf are of equal length.



B^+-tree: Insert

- Remember that B^+-trees remain **balanced**[3] no matter which updates we perform. Insertions and deletions have to preserve this invariant.
- The basic principle of B^+-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** . . . ?

[3] All paths from the B^+-tree root to any leaf are of equal length.

B^+-tree: Insert

- Remember that B^+-trees remain **balanced**[3] no matter which updates we perform. Insertions and deletions have to preserve this invariant.
- The basic principle of B^+-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 dependent on tree height only, so placing $k*$ somewhere else (even near p) is *no* option either.

[3] All paths from the B^+-tree root to any leaf are of equal length.

B$^+$-tree: Insert

- 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.

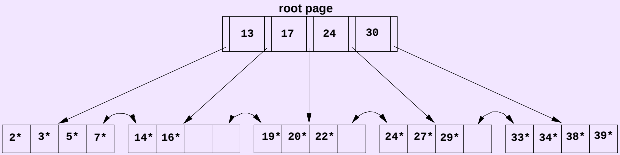
     Splitting happens recursively and may eventually lead to a split of the root node (increasing the tree height).

  4. Distribute the entries of p and the new entry $\langle k, q \rangle$ onto pages p and p'.

B^+-tree: Insert

Example (B^+-tree insertion procedure)

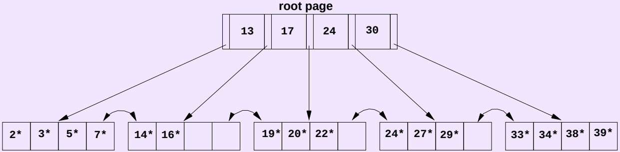
1. Insert record with key $k = 8$ into the following B^+-tree:



B^+-tree: Insert

Example (B^+-tree insertion procedure)

1. Insert record with key $k = 8$ into the following B^+-tree:

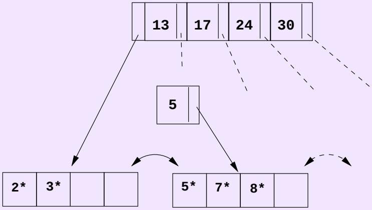


2. The left-most leaf page p has to be split. Entries $2*$, $3*$ remain on p, entries $5*$, $7*$, and $8*$ (new) go on new page p'.

B^+-tree: Insert and Leaf Node Split

Example (B^+-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**:

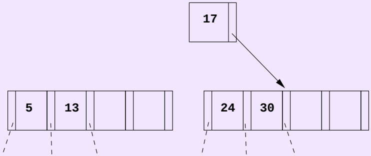


- Note that, after such a **leaf node split**, the new separator key $k' = 5$ is **copied up** the tree: the entry $5*$ itself has to remain in its leaf page.

B+-tree: Insert and Non-Leaf Node Split

Example (B+-tree insertion procedure)

4. 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):

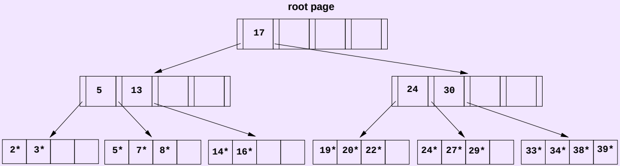


- Note that, for a **non-leaf node** split, we can simply **push up** the middle key (17). Contrast this with a leaf node split.

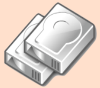
B^+-tree: Insert and Root Node Split

Example

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



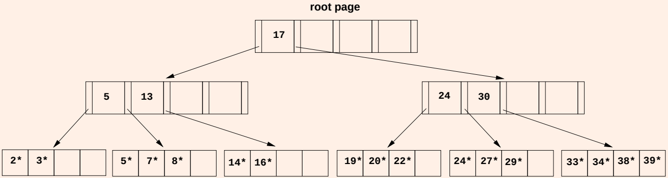
- Splitting the old root and creating a new root node is the *only* situation in which the **B^+-tree height increases**. The B^+-tree thus remains balanced.



B+-tree: Insert

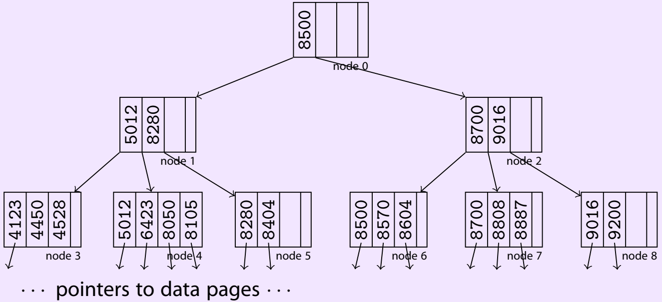
✎ Further key insertions

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

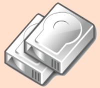


B^+-tree Insert: Further Examples

Example (B^+-tree insertion procedure)

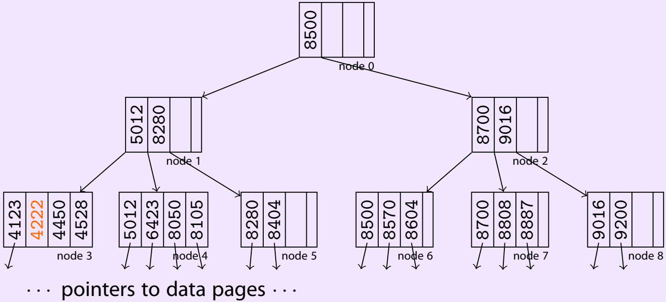


Insert new entry with key 4222.



B+-tree Insert: Further Examples

Example (B+-tree insertion procedure)



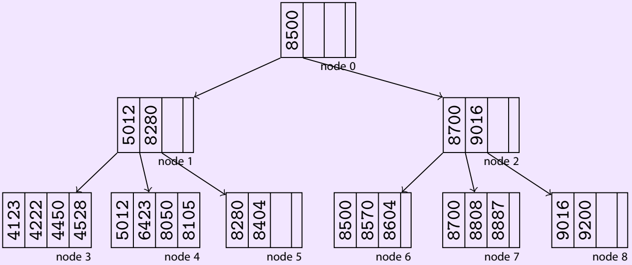
Insert new entry with key 4222.

⇒ Enough space in node 3, simply insert.

⇒ Keep entries **sorted within nodes**.

B^+-tree Insert: Further Examples

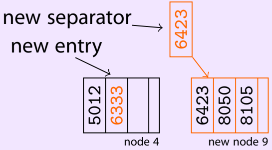
Example (B^+-tree insertion procedure)



Insert key 6333.

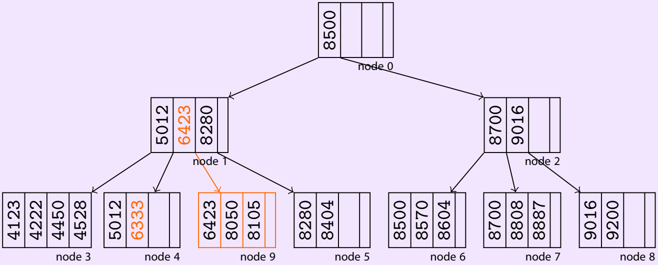
⇒ Must **split** node 4.

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



B^+-tree Insert: Further Examples

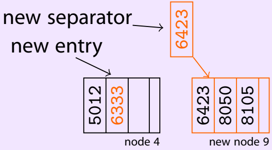
Example (B^+-tree insertion procedure)



Insert key 6333.

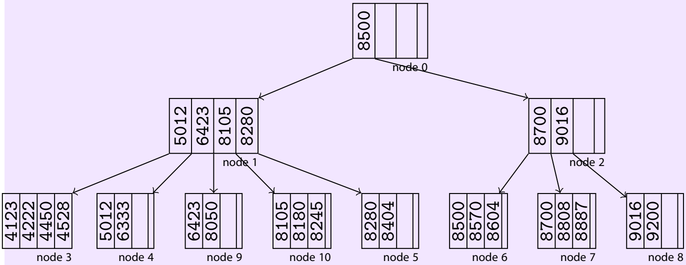
⇒ Must **split** node 4.

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



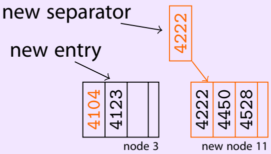
B$^+$-tree Insert: Further Examples

Example (B$^+$-tree insertion procedure)



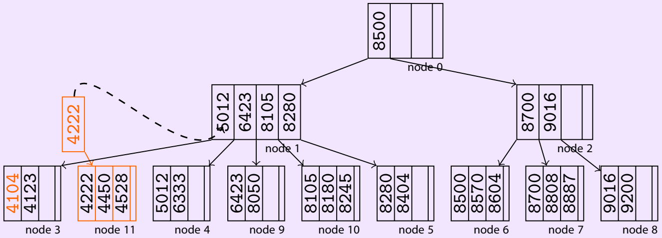
After 8180, 8245, insert key 4104.

⇒ Must **split** node 3.



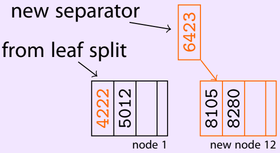
B^+-tree Insert: Further Examples

Example (B^+-tree insertion procedure)



After 8180, 8245, insert key 4104.

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

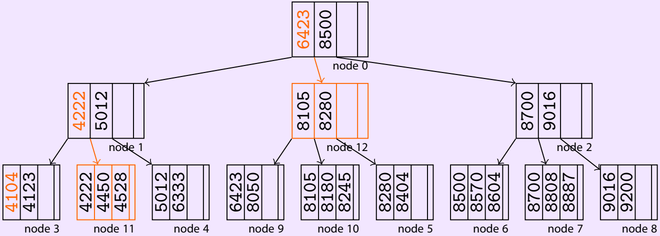


Unlike during leaf split, separator key does **not** remain in inner node.

✎ **Why?**

B^+-tree Insert: Further Examples

Example (B^+-tree insertion procedure)

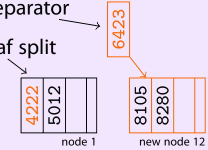


After 8180, 8245, insert key 4104.

- ⇒ 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.

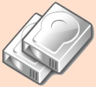
✎ **Why?**



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.

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, 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^+-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^+-tree: Insertion Algorithm

B^+-tree insertion algorithm

```
Function: tree_insert (k, rid, node)
if node is a leaf then
    return leaf_insert (k, rid, node);
else
    switch k do
        case k < k_1
            ⟨sep, ptr⟩ ← tree_insert (k, rid, p_0);
        case k_i ≤ k < k_{i+1}
            ⟨sep, ptr⟩ ← tree_insert (k, rid, p_i);
        case k_{2d} ≤ k
            ⟨sep, ptr⟩ ← tree_insert (k, rid, p_{2d});
    if sep is null then
        return ⟨null, null⟩;
    else
        return non_leaf_insert (sep, ptr, node);
```

(lines for `switch k do` … `case` $k_{2d} \le k$): see `tree_search ()`

Function: `leaf_insert` (k, rid, $node$)

if another entry fits into *node* **then**

  insert $\langle k, rid \rangle$ into *node* ;

  return $\langle \mathbf{null}, \mathbf{null} \rangle$;

else

  allocate new leaf page p ;

  let $\left\{ \langle k_1^+, rid_1^+ \rangle, \ldots, \langle k_{2d+1}^+, rid_{2d+1}^+ \rangle \right\}$:= entries from *node* $\cup \{\langle k, rid \rangle\}$

    leave entries $\langle k_1^+, rid_1^+ \rangle, \ldots, \langle k_d^+, rid_d^+ \rangle$ in *node* ;

    move entries $\langle k_{d+1}^+, rid_{d+1}^+ \rangle, \ldots, \langle k_{2d+1}^+, rid_{2d+1}^+ \rangle$ to p ;

  return $\langle k_{d+1}^+, p \rangle$;

Function: `non_leaf_insert` (k, ptr, $node$)

if another entry fits into *node* **then**

  insert $\langle k, ptr \rangle$ into *node* ;

  return $\langle \mathbf{null}, \mathbf{null} \rangle$;

else

  allocate new leaf page p ;

  let $\left\{ \langle k_1^+, p_1^+ \rangle, \ldots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle \right\}$:= entries from *node* $\cup \{\langle k, ptr \rangle\}$

    leave entries $\langle k_1^+, p_1^+ \rangle, \ldots, \langle k_d^+, p_d^+ \rangle$ in *node* ;

    move entries $\langle k_{d+2}^+, p_{d+2}^+ \rangle, \ldots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle$ to p ;

    set $p_0 \leftarrow p_{d+1}^+$ in p;

  return $\langle k_{d+1}^+, p \rangle$;

B^+-tree: Insertion Algorithm

B^+-tree insertion algorithm

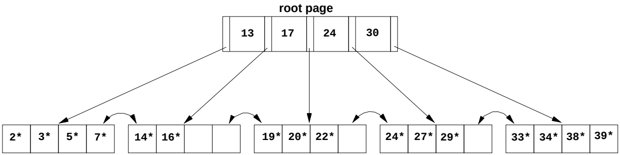
Function: `insert` (k, rid)

$\langle key, ptr \rangle \leftarrow$ `tree_insert` (k, rid, $root$);
if key is not **null then**
  allocate new root page r;
  populate r **with**
    $p_0 \leftarrow root$;
    $k_1 \leftarrow key$;
    $p_1 \leftarrow ptr$;
  $root \leftarrow r$;

- `insert` (k, rid) is called from outside.
- Variable *root* contains a pointer to the B^+-tree root page.
- Note how leaf node entries point to *rids*, while inner nodes contain pointers to other B^+-tree nodes.

B^+-tree Insert: Redistribution

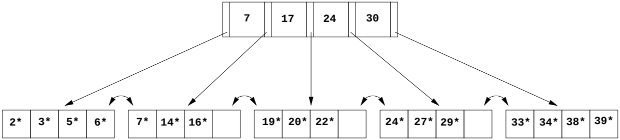
- We can further **improve the average occupancy** of B^+-tree using a technique called **redistribution**.
- Suppose we are trying to insert a record with key $k = 6$ into the B^+-tree below:



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

B^+-tree Insert: Redistribution

- 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^+-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).

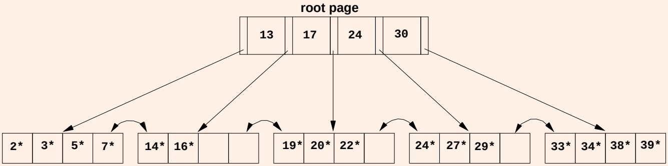
B$^+$-tree Insert: Redistribution

✎ Redistribution makes a difference

Insert a record with key $k = 30$

1. without redistribution,
2. using leaf level redistribution

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

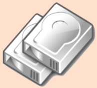


B^+-tree: Delete

- The principal idea to implement B^+-tree deletion comes as no surprise:

  1. To delete a record with key k, use `search`(k) to locate the leaf page p containing the record.
     Let m denote the number of entries on p.

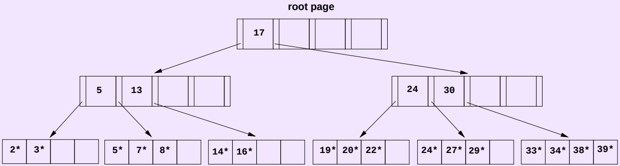
  2. If $m > d$ then p has sufficient occupancy: simply delete $k*$ from p (if $k*$ is present on p at all).
     Otherwise . . . ?



B^+-tree: Delete

Example (B^+-tree deletion procedure)

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

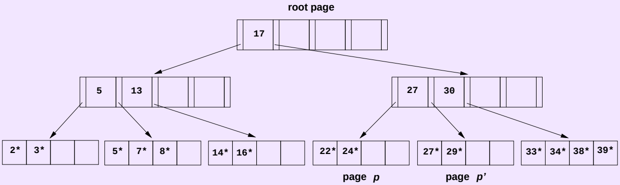


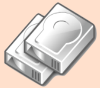
2. A call to `search(19)` leads us to leaf page p containing entries $19*$, $20*$, and $22*$. We can safely remove $19*$ since $m = 3 > 2$ (**no page underflow** in p after removal).

B^+-tree: Delete and Leaf Redistribution

Example (B^+-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:



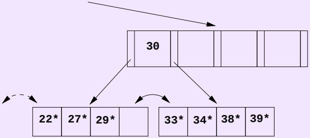


B^+-tree: Delete and Leaf Merging

Example (B^+-tree deletion procedure)

4. We continue and delete entry 24∗ from p. Redistribution is no option now (sibling p' has minimial occupancy of $d = 2$). We now have $m_p + m_{p'} = 1 + 2 < 2 \cdot d$ however: B^+-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.

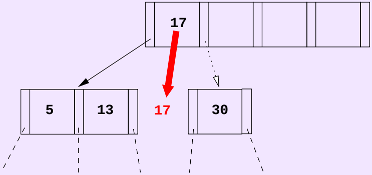
B^+-tree: Delete and Non-Leaf Node Merging

Example (B^+-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:

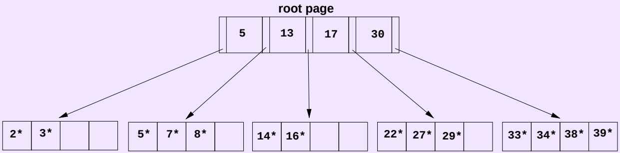


The missing key value, namely the separator of the two nodes (17), **is pulled down** (and thus deleted) from the parent to form the complete merged node.

B$^+$-tree: Root Deletion

Example (B$^+$-tree deletion procedure)

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



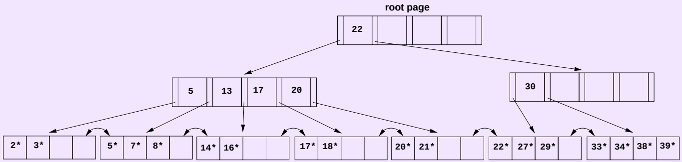
- This is the *only* situation in which the **B$^+$-tree height decreases**. The B$^+$-tree thus remains balanced.

B^+-tree: Delete and Non-Leaf Node Redistribution

Example (B^+-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^+-tree:

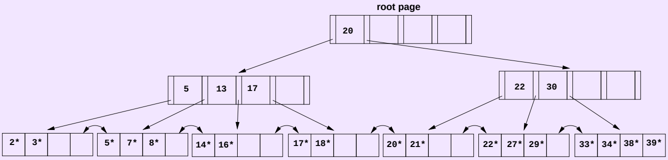


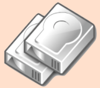
   - The non-leaf node with entry 30 underflowed. Its left sibling has two entries (17 and 20) to spare.

B^+-tree: Delete and Non-Leaf Node Redistribution

Example (B^+-tree deletion procedure)

❽ We redistribute entry 20 by "**rotating it through**" the parent. The former parent entry 22 is pushed down:





Merge and Redistribution Effort

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

DB2 B^+-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^+-tree: Duplicates

- As discussed here, the B^+-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^+-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 identitifers for each tuple in a table, and
2. are *not* dependent on tuple order and *never* rise again.

B$^+$-tree: Duplicates (IBM DB2 Tuple Identities)

DB2 Expose IBM DB2 tuple identity

```
$ db2
(c) Copyright IBM Corporation 1993,2007
Command Line Processor for DB2 Client 9.5.0
[...]
db2 => CREATE TABLE FOO(ROWID INT GENERATED ALWAYS AS IDENTITY,
                        text varchar(10))
db2 => INSERT INTO FOO VALUES (DEFAULT, 'This␣is'), ...
db2 => SELECT * FROM FOO

ROWID       TEXT
----------- ----------
          1 This is
          2 nothing
          3 but a
          4 silly
          5 example!
```

B$^+$-tree: Duplicates (IBM DB2 Tuple Identities)

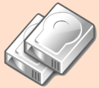
Expose IBM DB2 tuple identity (continued)

```
db2 => DELETE FROM FOO WHERE TEXT = 'silly'
db2 => SELECT * FROM FOO

ROWID        TEXT
----------- ----------
          1 This is
          2 nothing
          3 but a
          5 example!

db2 => INSERT INTO FOO VALUES (DEFAULT, 'I␣am␣new.')
db2 => SELECT * FROM FOO

ROWID        TEXT
----------- ----------
          1 This is
          2 nothing
          3 but a
          6 I am new.
          5 example!
```

B+-tree: Duplicates

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

❶ Use variant Ⓒ (see slide 3.22) to represent the index data entries $k*$:

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

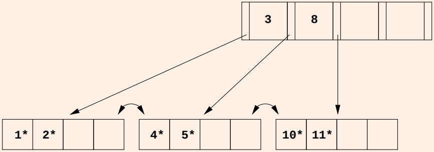
- Each duplicate record with key field k makes the list of *rids* grow. Key k is not repeatedly stored (space savings).
- B+-tree search and maintenance routines largely unaffected. Index data entry size varies, however (this affects the B+-tree **order** concept).
- Implemented in IBM Informix Dynamic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

B^+-tree: Duplicates

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

Impact of duplicate insertion on `search`

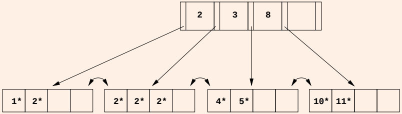
Given the following B^+-tree of order $d = 2$, perform insertions (do not use redistribution): `insert(2,·)`, `insert(2,·)`, `insert(2,·)`:



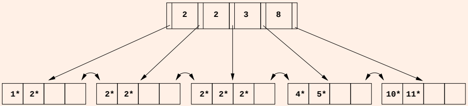
B^+-tree: Duplicates

✎ Impact of duplicate insertion on `search`

The resulting B^+-tree is shown here. Now apply `insert(2,·)`, `insert(2,·)` to this B^+-tree:



We get the tree depicted below:



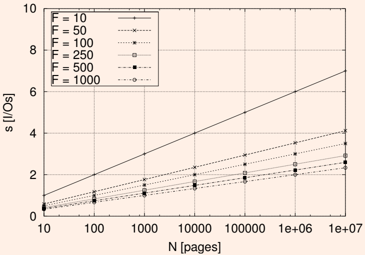
⇒ In `search`: in a non-leaf node, follow the **left-most** page pointer p_i such that $k_i \leqslant k \leqslant k_{i+1}$. In a leaf, also check the left sibling.

B^+-tree: Key Compression

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

$$s = \log_F N \quad .$$

Tree index search effort dependent on fan-out F



$\Rightarrow$ It clearly pays off to invest effort and **try to maximize the fan-out** F of a given B^+-tree implementation.

B^+-tree: Key Compression

- Index entries in inner (*i.e.*, non-leaf) B^+-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)

switch k **do**
- **case** $k < k_1$
  - …
- **case** $k_i \leqslant k < k_{i+1}$
  - …
- **case** $k_{2d} \leqslant k$
  - …

B^+-tree: Key Compression

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

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

- 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

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

- 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)`

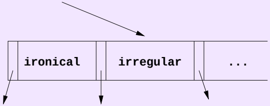
switch k **do**
  case $k < k_1$
    $\ldots$
  case $k_i \leqslant k < k_{i+1}$
    $\ldots$
  case $k_{2d} \leqslant k$
    $\ldots$

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

B^+-tree: Key Compression

Example (Searching a B^+-tree node with `VARCHAR(·) keys`)

To guide the search across this B^+-tree node



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

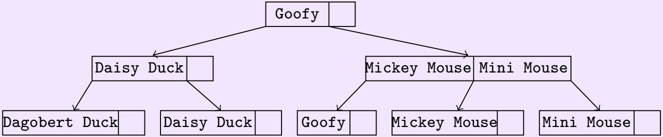
We must preserve the B^+-tree semantics, though:

> *All index entries stored in the subtree left of* `iro` *have keys* $k <$ `iro` *and index entries stored in the subtree right of* `iro` *have keys* $k \geqslant$ `iro` *(and* $k <$ `irr`*).*

B^+-tree: Key Suffix Truncation

Example (Key suffix truncation, B^+-tree with order $d = 1$)

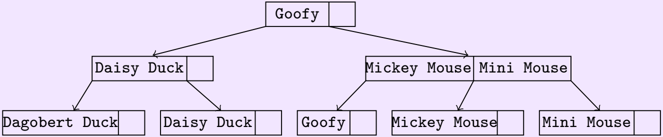
Before key suffix truncation:



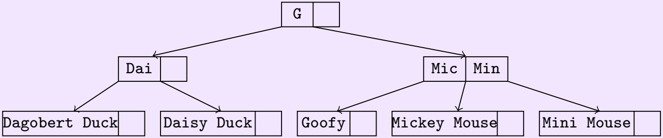
B$^+$-tree: Key Suffix Truncation

Example (Key suffix truncation, B$^+$-tree with order $d = 1$)

Before key suffix truncation:



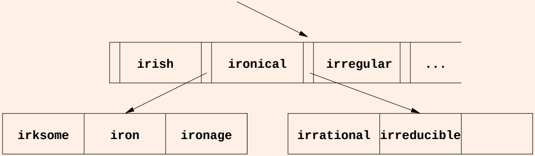
After key suffix truncation:



B^+-tree: Key Suffix Truncation

✎ Key suffix truncation

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



B^+-tree: Key Prefix Compression

- Observation: Keys within a B^+-tree node often share a **common prefix**.

Example (Shared key prefixes in inner B^+-tree nodes)

Key prefix compression:

- Store common prefix only once (*e.g.*, as "k_0")
- 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^+-tree: Bulk Loading

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

Table and index creation

```
db2 => CREATE TABLE FOO (ID INT, TEXT VARCHAR(10));

[... insert 1,000,000 rows into table foo ...]

db2 => CREATE INDEX FOO_IDX ON FOO (A ASC)
```

- The last SQL command initiates 1,000,000 calls to the B^+-tree `insert(·)` procedure—a so-called index **bulk load.**

⇒ The DBMS will traverse the growing B^+-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$^+$-tree: Bulk Loading

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

B$^+$-tree bulk loading algorithm

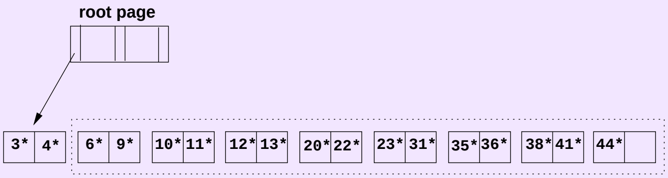
1. For each record with key k in the data file, create a **sorted list** of pages of index leaf entries $k*$.

   Note: For index variants Ⓑ or Ⓒ, this does *not* imply to sort the data file itself. (For variant Ⓐ, 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^+-tree: Bulk Loading

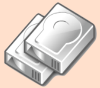
Example (State of bulk load after step ②, order of B^+-tree $d = 1$)



(Index leaf pages not yet in B^+-tree are framed .)

✎ Bulk loading continued

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



B^+-tree: Bulk Loading

- 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^+-tree root-to-leaf traversals altogether:

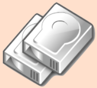
B^+-tree bulk loading algorithm (continued)

3. For each leaf level page *p*, insert the index entry

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

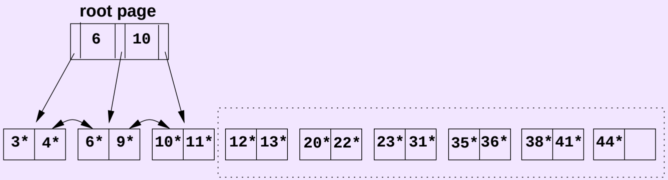
into the right-most index node just above the leaf level.

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.



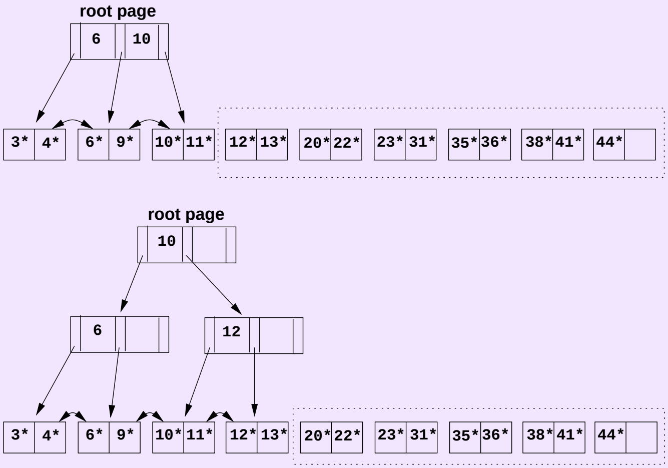
B$^+$-tree: Bulk Loading

Example (Bulk load continued)



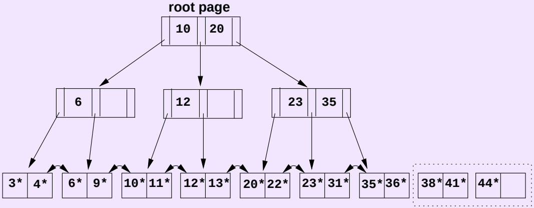
B^+-tree: Bulk Loading

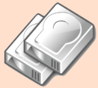
Example (Bulk load continued)



B+-tree: Bulk Loading

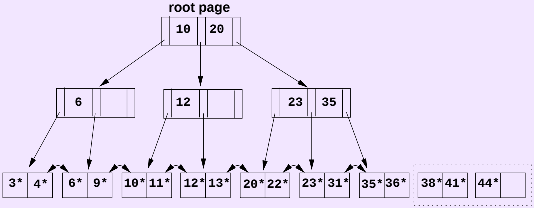
Example (Bulk load continued)

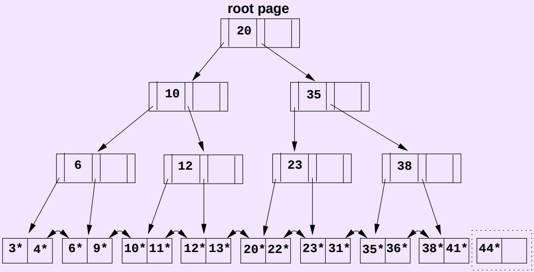


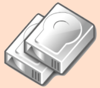


B+-tree: Bulk Loading

Example (Bulk load continued)







Composite Keys

B^+-trees can (in theory[4]) 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^+-tree.

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

[4] 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?**

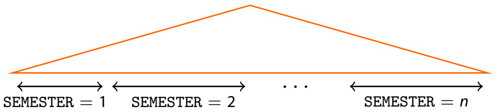
Partitioned B-trees

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^+-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**.