DB 2

09 – Ordered Indexes (B$^+$Trees)

Summer 2020

Torsten Grust
Universität Tübingen, Germany

1 ¦ Q_8 — Filtering an Indexed Table

Sequential scan (**Seq Scan**) and interpreted predicate evaluation go a long way. Large input tables call for significantly more **efficient support for value-based row access:**

```sql
SELECT i.b, i.c
FROM   indexed AS i
WHERE  i.a = 42 [i.c = 0.42] -- either filter on i.a or i.c
```

Assume column a is **primary key** in table indexed: expect query workload that frequently identifies rows via predicates a = *k*. **Indexes** can support such queries.

Primary Key and Indexes

```
CREATE TABLE indexed (a int PRIMARY KEY, -- ⇒ NOT NULL
                      b text,
                      c numeric(3,2));    -- ± d.dd
```

DBMS expects predicates a = *k* and creates an **index on column a**—a data structure associated with and maintained in addition to table indexed—to speed up evaluation:

```
CREATE INDEX indexed_a ON indexed USING btree (a);
```

1. Whenever possible/promising, index indexed_a[1] is (also) consulted when table indexed is queried. 🚀
2. When indexed is updated, indexed_a is maintained. ⚙

[1] PostgreSQL chooses index name indexed_pkey but let's follow a *‹table›_‹column›* naming scheme here.

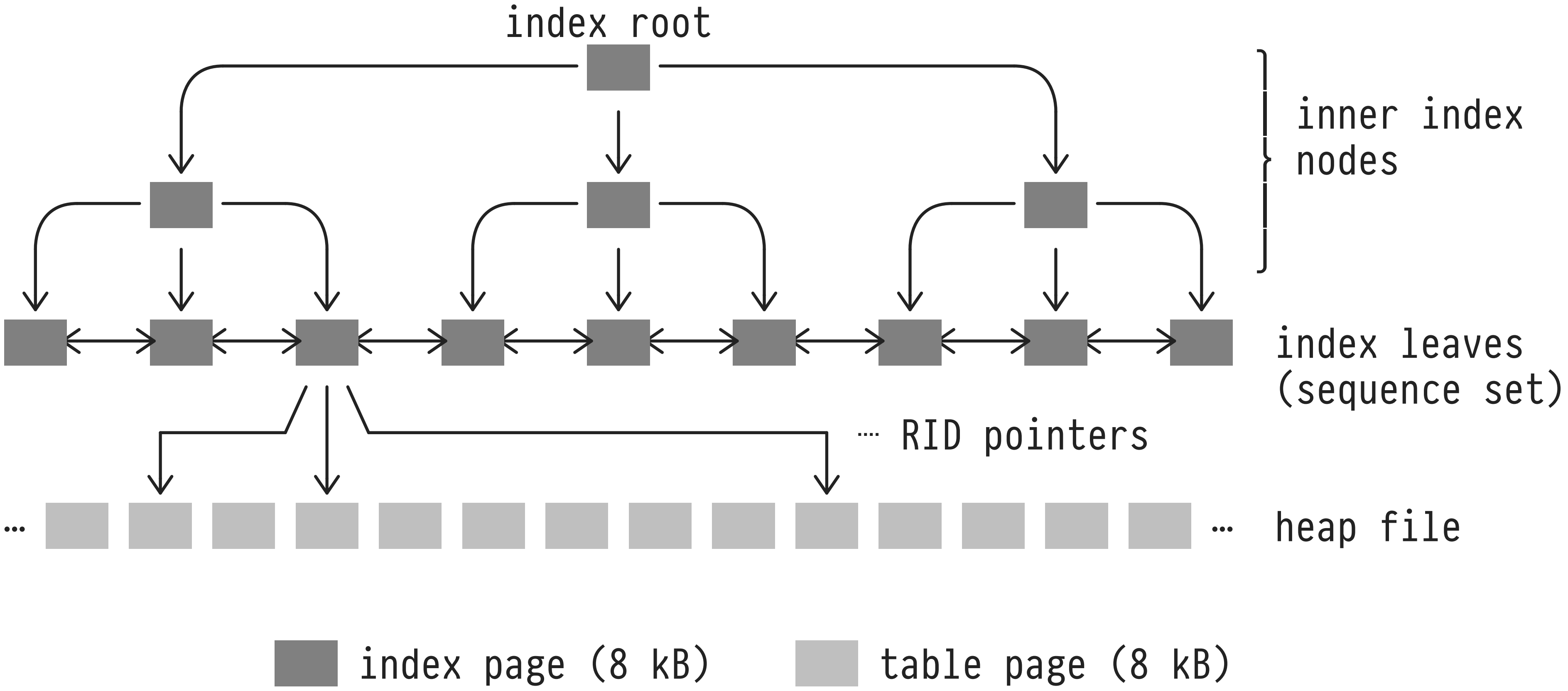
Using EXPLAIN on Q_8

```
EXPLAIN VERBOSE
  SELECT i.b, i.c
  FROM   indexed AS i    -- 10^6 rows
  WHERE  i.a = 42;       -- selection on key column a ⇒ ⩽ 1 row will qualify
```

```
                         QUERY PLAN
--------------------------------------------------------------------------
 Index Scan using indexed_a on indexed i  (cost=0.42..8.44 rows=1 …)
   Output: b, c
   Index Cond: (i.a = 42)
```

- DBMS uses **Index Scan** (instead of **Seq Scan**), index scan will evaluate predicate i.a = *k*.
- System expects small result of a single row (rows=1), i.e., the predicate is assumed to be *very selective*.

2 ¦ B$^+$Trees: Ordered Indexes



Anatomy of a B$^+$Tree

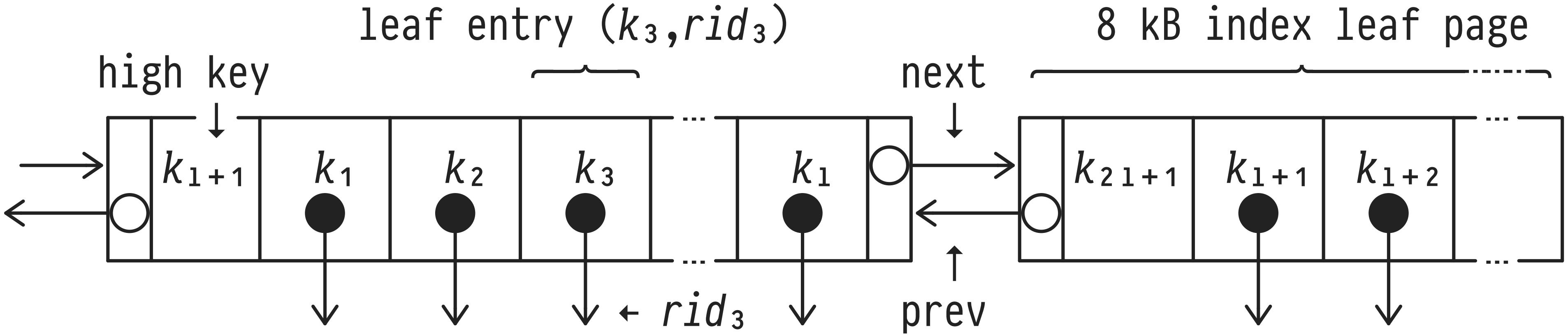
B$^+$Trees: Ordered Indexes

Notes on B$^+$Tree anatomy:

- A **B$^+$Tree[2] index I on column $T(a)$** is an *ordered*, *n-ary* ($n \gg 2$), *balanced*, *block-oriented*, *dynamic* search tree.
- Inner nodes and leaves are formed by 8 kB **index pages.**
- Each inner node holds n−1 values of column a (**separators**) that allow to navigate the search tree structure.
- Leaves form a bidirectional chain, the **sequence set.**
- **Leaves use RIDs to point to rows** in the heap file of table T: besides a column values, I holds no data of T.

[2] Invented by Bayer and McCreight (1969) at Boeing Labs. The “B” in “B$^+$Tree” does *not* stand for Bayer, binary, balanced, block, or Boeing. (We tried to find out, but Rudolf Bayer wouldn't say.)

B⁺Trees: Inside a Leaf Node



- Uses pointers prev/next to form the chained **sequence set.**
- **Leaf entries are ordered** by index keys k_i: $k_i \leqslant k_{i+1}$.
- RID rid_i points to a row t of T with $t.a = k_i$.
- The **high key** holds smallest key of *next* leaf (if any).

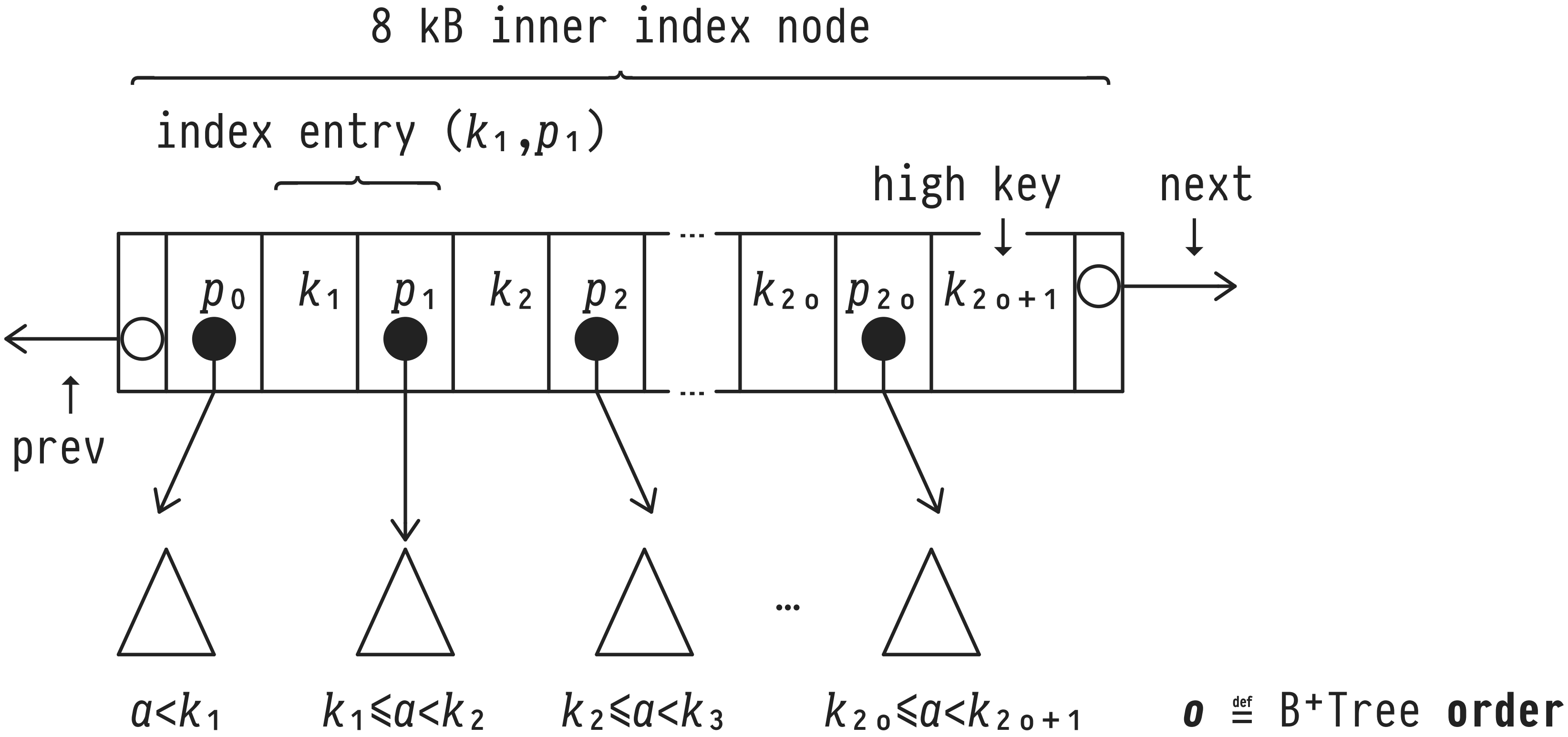
B$^+$Trees: How to Find Rows t With $t.a = k$?

As described, a B$^+$Tree is a **dense** index structure: every row t of T is represented by one leaf entry.

- The sequence set is ordered by keys k_i ⇒ a *binary search* for a key $k = k_i$ may sound viable, **BUT** the search would
  1. need to inspect $\log_2(|T|)$ keys in the sequence set and access just as many pages 👎, and
  2. “jump around” the sequence set in an unpredictable fashion, thus leading to random I/O. 👎

B$^+$Trees exploit the sequence set ordering and erect an ***n*-ary search tree structure** (n large!) atop the leaf entries.

B⁺Trees: Inside an Inner Node



- The **separator** keys k_i are ordered: $k_i \leqslant k_{i+1}$.
- Page pointers p_j point to index (leaf or inner) nodes.

B⁺Trees: Notes on Inner Nodes

- Space in inner nodes is used economically: in a B⁺Tree of **order** o, any inner node—but the root node—is guaranteed to hold between o and $2 \times o$ ($\overset{\text{def}}{\equiv}$ **fan-out** F) index entries.
- Given predicate $t.a = k$, perform **binary search inside node** to find B⁺Tree subtree with $k_i \leqslant k < k_{i+1}$.
- B⁺Tree is **balanced:** subtrees △ are of identical height.
- **Path length** s from B⁺Tree root to leaf node **predictable:**

$$|T| \times \underbrace{1/F \times \cdots \times 1/F}_{s \text{ times}} = 1 \Leftrightarrow s = \log_F(|T|)$$

3 ¦ Index Scan

A B^+Tree is *the* index structure to support the evaluation of these kinds of conditions:[3]

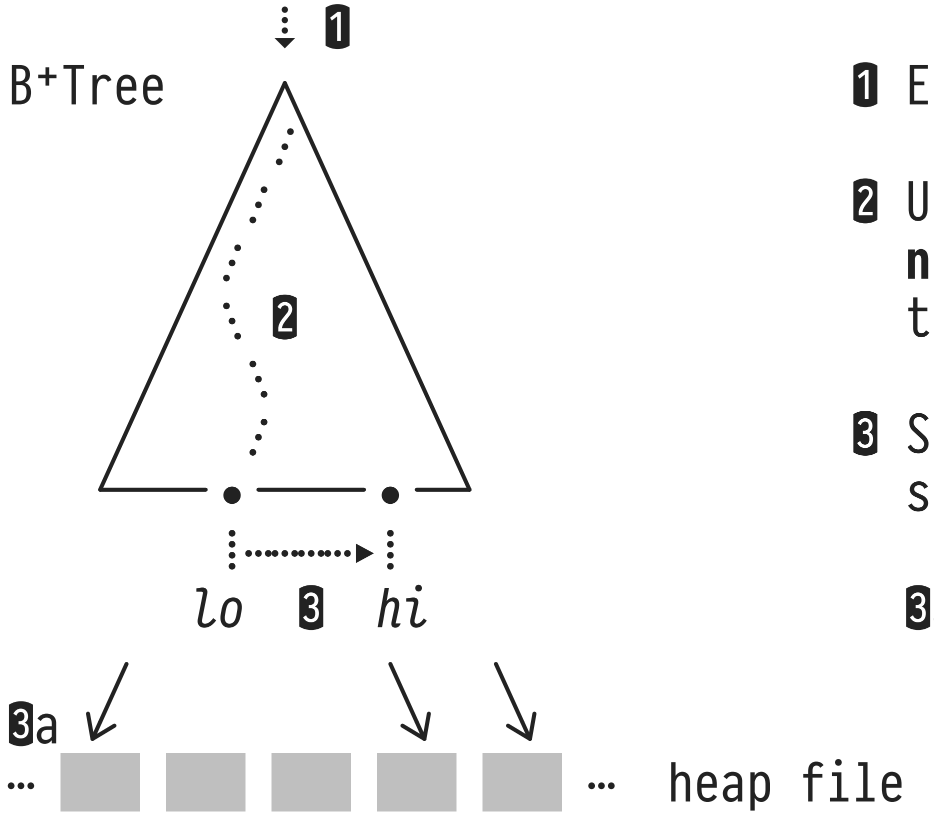
1. **Range predicates:** $lo \leqslant a \leqslant hi$
2. **Half-open ranges:** $lo \leqslant a$ or $a \leqslant hi$
3. **Equality predicates:** $a = lo$

- An **Index Scan** on index I for column $T(a)$ is parameterized by such a condition (PostgreSQL EXPLAIN: Index Cond).
- **Index Scan** uses lo to navigate the search tree structure and locate the start of relevant sequence set section.

[3] Half-open ranges are special range predicates where $hi = \infty$ ($lo = \infty$). Equality predicates are special range predicates where $lo = hi$.

Index Scan for Condition $lo \leqslant a \leqslant hi$

An **index scan** accesses the B+Tree index *and* the heap file:



❶ Enter at B+Tree root page

❷ Use key *lo* to **navigate the inner nodes** (search tree) until we reach the leaf level

❸ Scan leaf entries in the sequence set section $lo \leqslant a \leqslant hi$, **extract RIDs**

❸a For each RID, **access heap file for table** *T* and return matching row

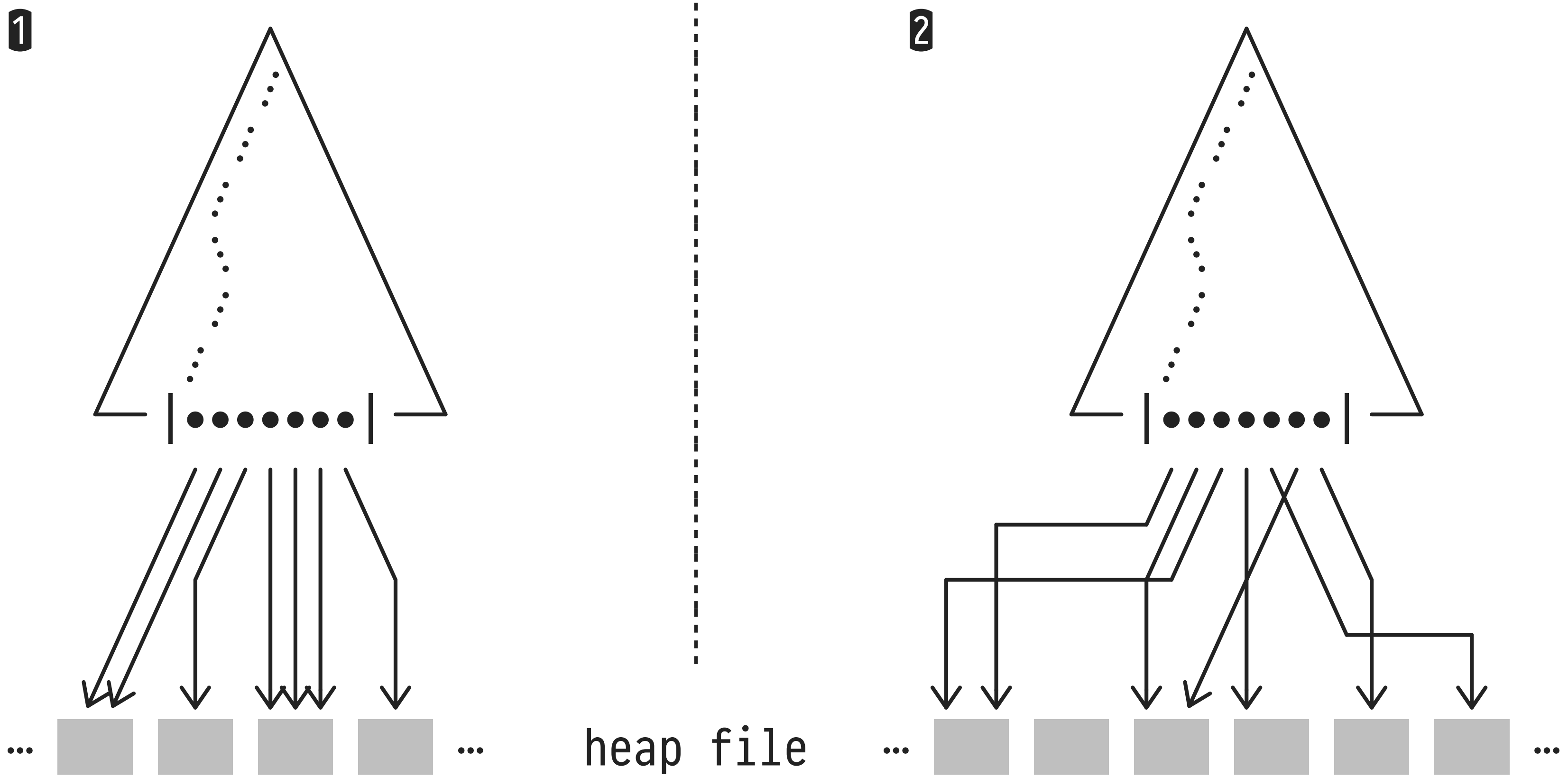
Navigating the Inner Nodes

Phase ❷ runs a vanilla traversal of a 2×o-way search tree:

```
Search(lo):                                  } returns entry point
  return TreeSearch(lo, root);               } for scan of
                                             } sequence set

TreeSearch(lo, node):
  if (node is a leaf)
    return node;
  switch lo                                  }
    case $lo < k_1$                          }
      return TreeSearch(lo, $p_0$);          } use binary search
    case $k_i \leqslant lo < k_{i+1}$        } to implement
      return TreeSearch(lo, $p_i$);          } subtree choice
    case $k_{2o} \leqslant lo$               }
      return TreeSearch(lo, $p_{2o}$);       }
```

4 ¦ Order of Leaf Entries *vs* Order of Table Rows



❶ Order of leaf entry keys k_i ≡ row order in heap file. 👍

❷ Order of k_i in sequence set and row order do *not* match.

Clustered Indexes

Index I for column $T(a)$ is **clustered** if the order of leaf entries coincides with T's row order (i.e., both I's sequence set and T's heap file are ordered by a):

Given entries $\langle k_i, p_i \rangle$ and $\langle k_j, p_j \rangle$, $k_i \leqslant k_j \Rightarrow p_i \leqslant p_j$.

- An **Index Scan** over a *clustered* index
  1. collects matching rows from adjacent heap file pages (⇒ sequential I/O 👍),
  2. will find many matching rows on each accessed heap file page (⇒ less page I/O 👍).

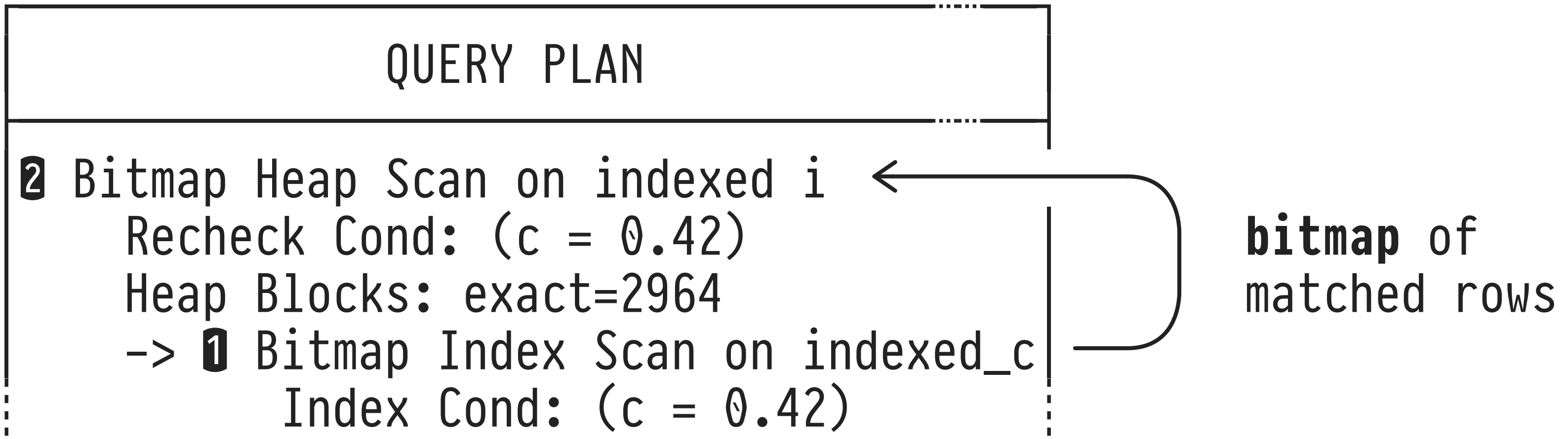
Non-Clustered Indexes

Sad fact: only *one*—among the many possible—indexes for a table may be clustered. Most indexes are non-clustered.

- An **Index Scan** over a *non-clustered* index
  1. will find matching rows potentially scattered across all heap file pages (⇒ random I/O 👎),
  2. will find few matching rows on each accessed heap file page and may access the same page more than once (⇒ as many page I/Os as matching rows 👎).

PostgreSQL addresses this challenge through **RID sorting**, implemented via **Bitmap Index Scan** & **Bitmap Heap Scan**.

Bitmap Index Scan & Bitmap Heap Scan



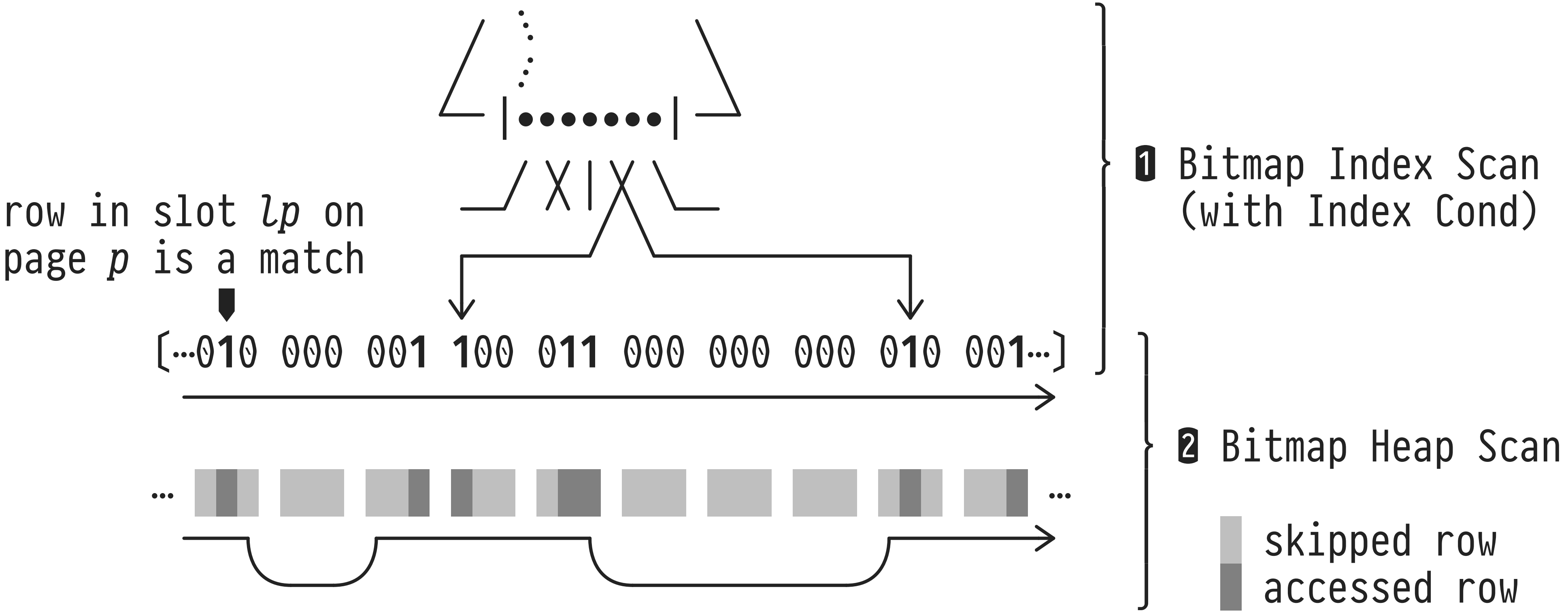
```
QUERY PLAN
❷ Bitmap Heap Scan on indexed i
    Recheck Cond: (c = 0.42)
    Heap Blocks: exact=2964
    -> ❶ Bitmap Index Scan on indexed_c
          Index Cond: (c = 0.42)
```

bitmap of
matched rows

❶ **Bitmap Index Scan:** perform Index Scan and create **bitmap** that encodes *heap file locations* of rows matching the Index Cond. Do *not* access rows in heap file yet.

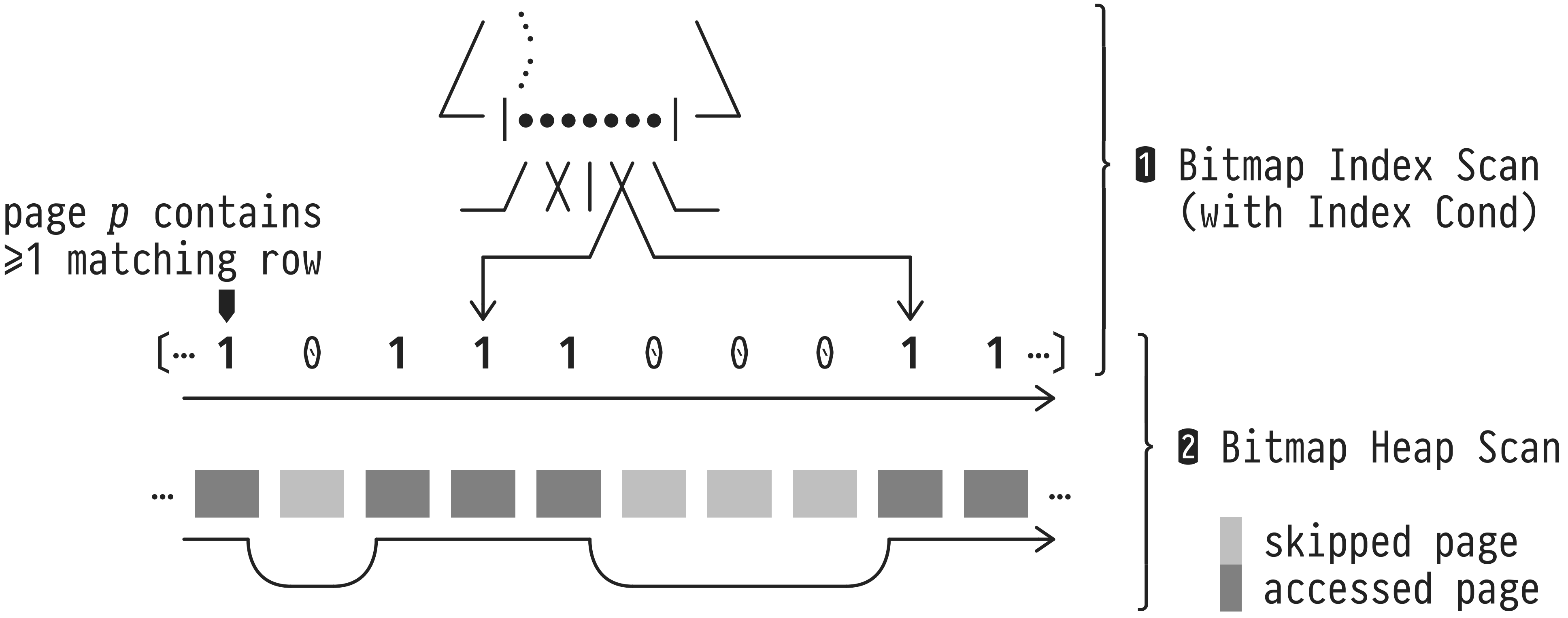
❷ **Bitmap Heap Scan:** scan heap file once, only access those rows (pages) that have been marked **1** in the bitmap.

Bitmap Index Scan & Bitmap Heap Scan: Row-Level Bitmap



Bitmap Heap Scan performs one sequential scan (with skips) of the heap file, regardless of RID order in sequence set.

Bitmap Index Scan & Bitmap Heap Scan: Page-Level Bitmap



Working memory tight ⇒ build **page-level** bitmap. ⚠️ In ❷, need to **recheck condition** for all rows on accessed pages.

5 ¦ CLUSTERing Based on an Index

If the workload depends on top performance of particular predicates supported by *non-clustered* index *I*, we may

> **physically reorder the rows of underlying table's** *T* **heap file** to coincide with the key order in *I*'s sequence set (i.e., *I* will become a *clustered* index[4]):

```
CLUSTER [VERBOSE] ‹T› USING ‹I›;
CLUSTER ‹T›;    -- re-cluster once T's rows get out of order
```

- ⚠ Subsequent updates on *T* can destroy the perfect clustering. (May need to re-cluster *T* in intervals.)

[4] At a price, of course: formerly *clustered* indexes on *T* will turn into *non-clustered* indexes.

B+Trees…

1. **economically utilize space** in inner/leaf nodes (minimum node occupancy 50%, typical fill factor 67%),
2. are **balanced** trees and thus require a **predictable number of page I/Os** to traverse from root to sequence set—enables query optimizer to forecast B+Tree access cost.

DBMSs maintain properties 1. and 2. when rows are **inserted** into/**deleted** from an B+Tree-indexed table.[5]

[5] Some real B+Tree implementations of row deletion deviate from the textbook to keep things simpler.

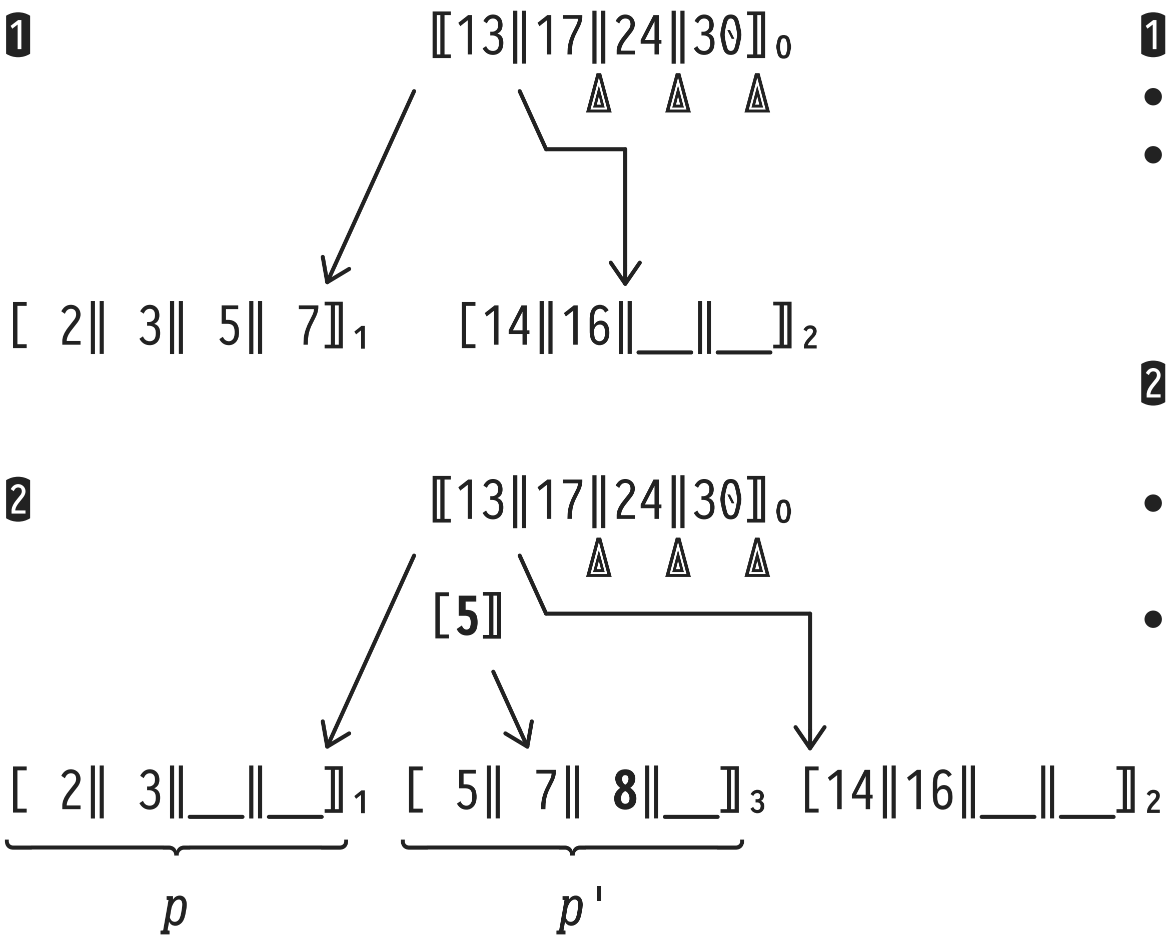
B$^+$Tree Insertion for New Entry $\langle k,rid \rangle$ (Sketch)

1. Use Search(k) to **find leaf page** p which should hold the entry for k.
2. If p has **enough space** to hold new entry (i.e., at most $2 \times o - 1$ entries in p), **simply insert** $\langle k,rid \rangle$ 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 B$^+$Tree height).

   - **Distribute** the entries of p and new entry $\langle k,rid \rangle$ onto pages p and p'.

B⁺Tree Insertion and Leaf Node Split



❶ Insert new entry ‹**8**,*rid*›

- Search(**8**) returns leaf *p* = 1
- Leaf 1 is full ⇛ split

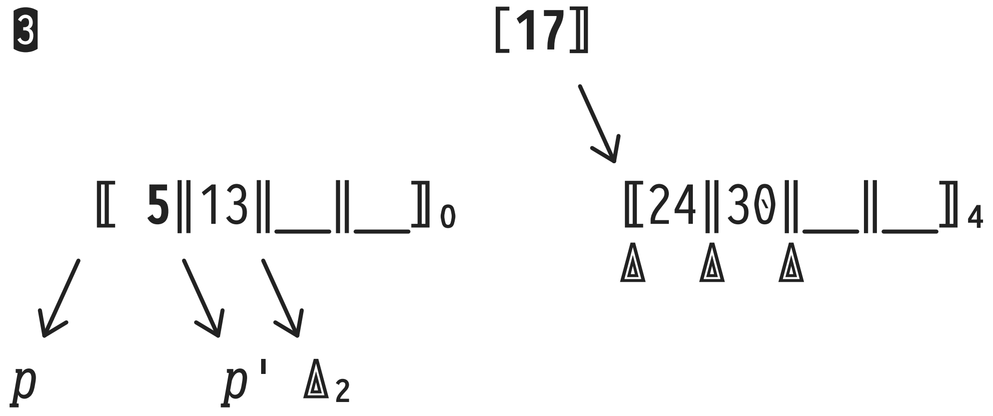
❷ Leaf 1 split into leaves
p = 1 and *p*' = 3

- Distribute {2,3,5,7,**8**} between leaves 1 and 3
- **Copy**↻ new separator [**5**⟧ into parent node 0

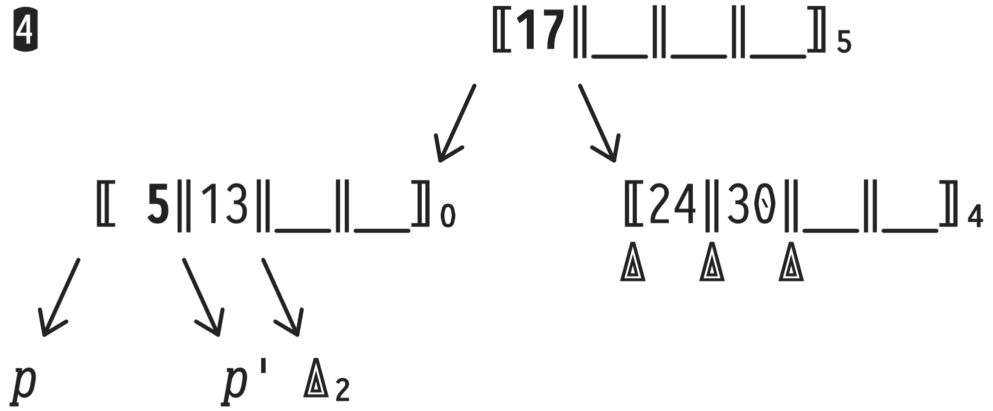
p'

B+Tree Insertion and Inner Node Split

❸



❹



❸ Inner node 0 (here: root) is full ⇒ split

- Inner node 0 splits into old node 0 and new *p*'' = 4
- Distribute {**5**,13,24,30} ▲ between nodes 0 and 4
- **Move** new separator [**17**⟧ into parent of node 0 ↘ *p*''

❹ Split node 0 has been the old root

- Create new root node 5, has ⟦**17**‖ as only entry
- B+Tree height has increased

B+Tree Insertion Notes

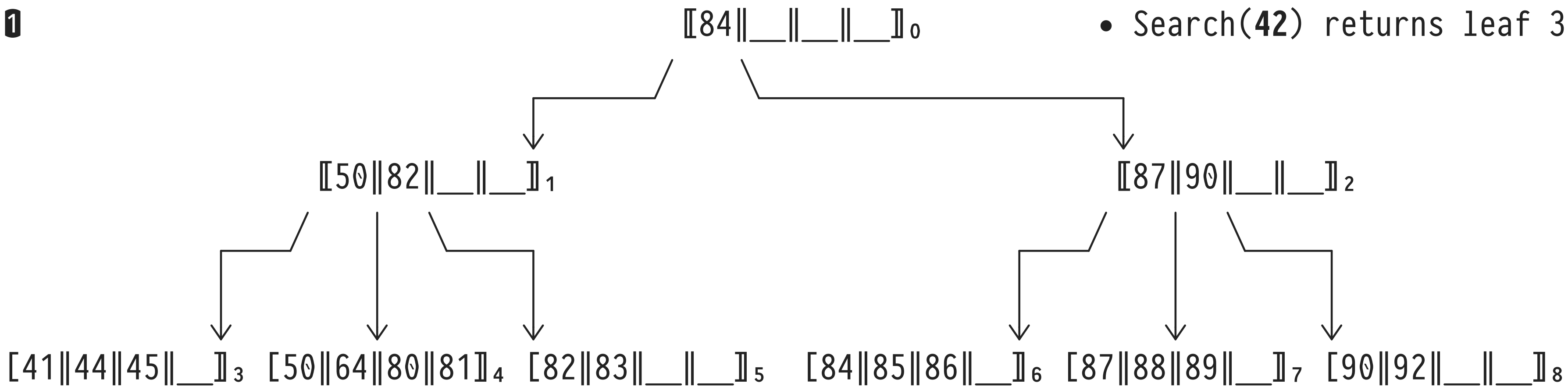
- Splitting starts at the leaf level and continues upward as long as inner index nodes are fully occupied (holding $2 \times o$ entries).

- ⚠️ Unlike during a *leaf* split, an *inner* node split **moves**[6] the new separator [***sep***⟧ discriminating between p and p' upwards and recursively inserts it into the parent. **Q:** Why?

- **Q:** How often do you expect a root node split to happen?

[6] A leaf node split **copies** the new separator upwards, i.e., the entry [***sep***⟧ also remains at the leaf level.

B⁺Tree Insertion Example: Insert ‹42,*rid*›



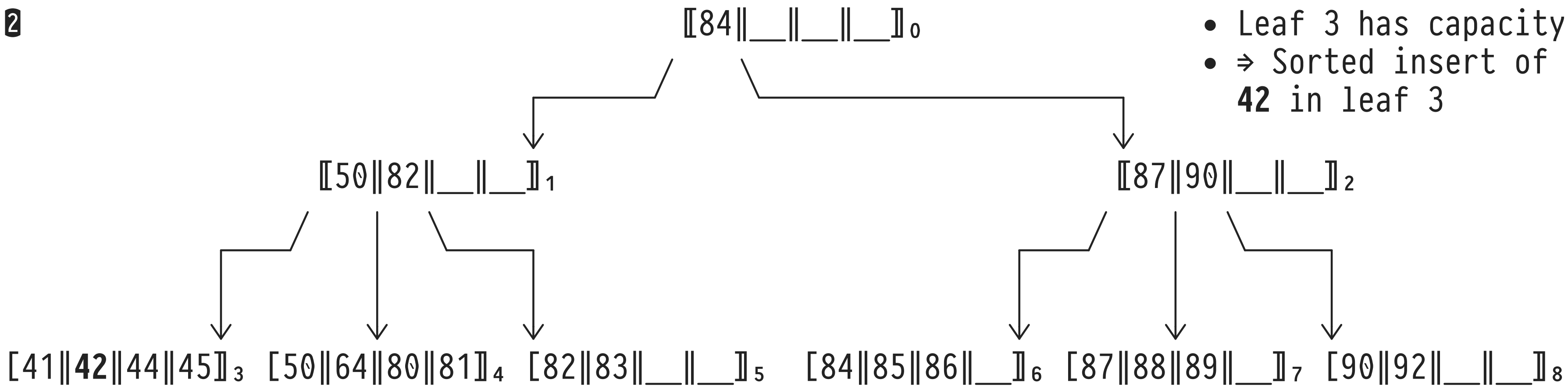
1

⟦84‖__‖__‖__⟧$_0$

⟦50‖82‖__‖__⟧$_1$ ⟦87‖90‖__‖__⟧$_2$

[41‖44‖45‖__⟧$_3$ [50‖64‖80‖81⟧$_4$ [82‖83‖__‖__⟧$_5$ [84‖85‖86‖__⟧$_6$ [87‖88‖89‖__⟧$_7$ [90‖92‖__‖__⟧$_8$

- Search(**42**) returns leaf 3



2

⟦84‖__‖__‖__⟧$_0$

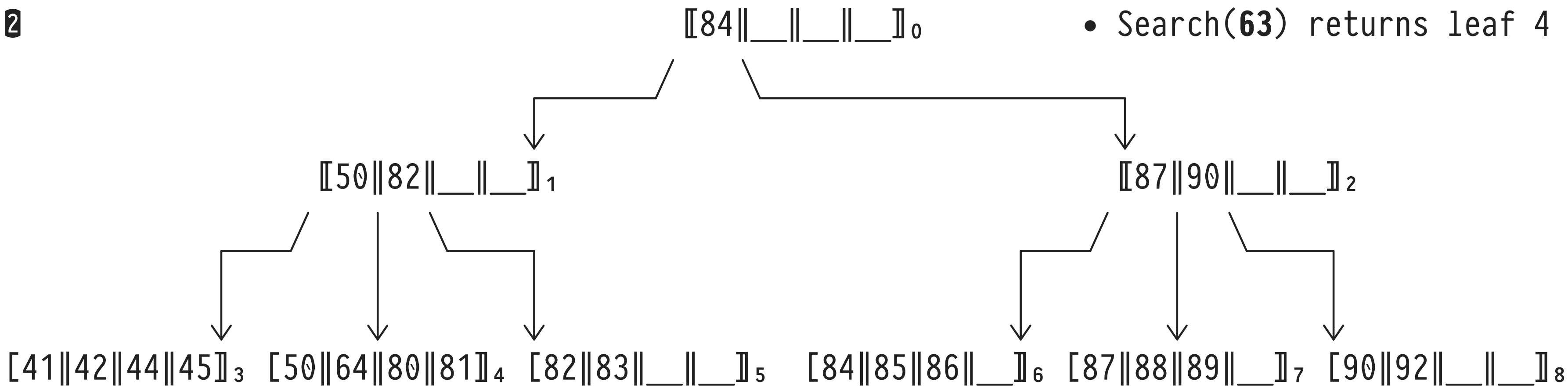
⟦50‖82‖__‖__⟧$_1$ ⟦87‖90‖__‖__⟧$_2$

[41‖**42**‖44‖45⟧$_3$ [50‖64‖80‖81⟧$_4$ [82‖83‖__‖__⟧$_5$ [84‖85‖86‖__⟧$_6$ [87‖88‖89‖__⟧$_7$ [90‖92‖__‖__⟧$_8$

- Leaf 3 has capacity
- ⇒ Sorted insert of **42** in leaf 3

B$^+$Tree Insertion Example: Insert ‹63,*rid*›

2



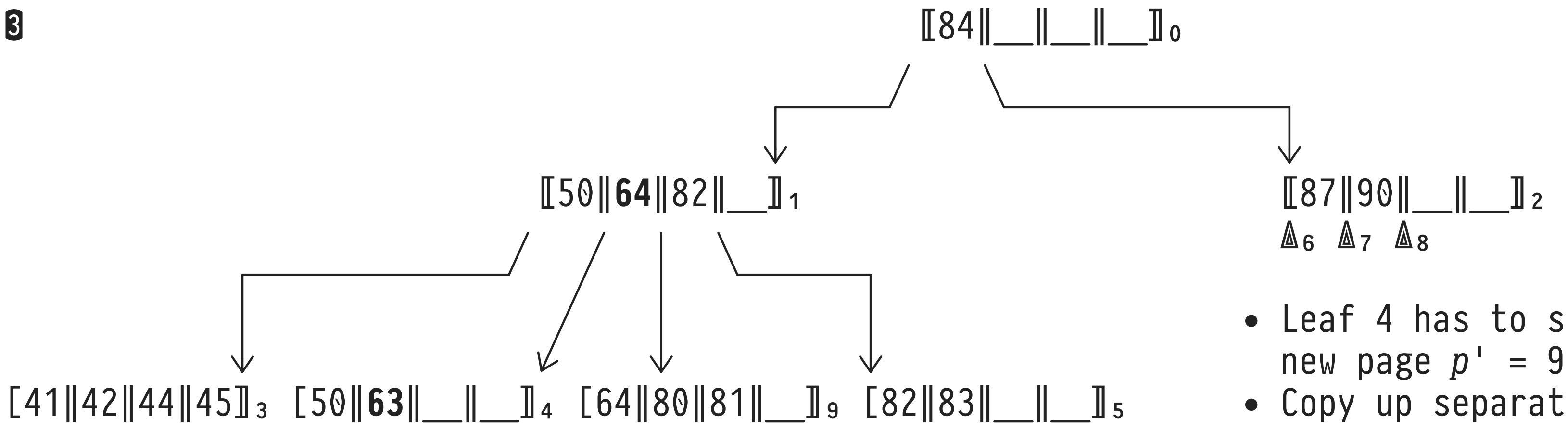
⟦84‖__‖__‖__⟧$_0$

⟦50‖82‖__‖__⟧$_1$ ⟦87‖90‖__‖__⟧$_2$

[41‖42‖44‖45⟧$_3$ [50‖64‖80‖81⟧$_4$ [82‖83‖__‖__⟧$_5$ [84‖85‖86‖__⟧$_6$ [87‖88‖89‖__⟧$_7$ [90‖92‖__‖__⟧$_8$

- Search(**63**) returns leaf 4

3



⟦84‖__‖__‖__⟧$_0$

⟦50‖**64**‖82‖__⟧$_1$ ⟦87‖90‖__‖__⟧$_2$

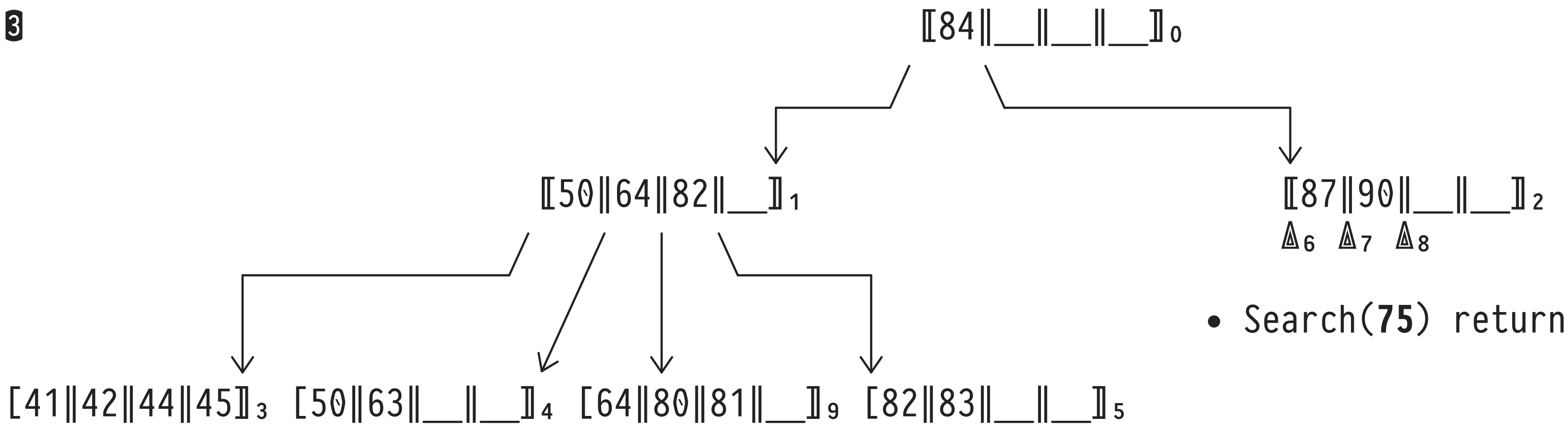
△$_6$ △$_7$ △$_8$

[41‖42‖44‖45⟧$_3$ [50‖**63**‖__‖__⟧$_4$ [64‖80‖81‖__⟧$_9$ [82‖83‖__‖__⟧$_5$

- Leaf 4 has to split, new page *p*' = 9
- Copy up separator [**64**⟧

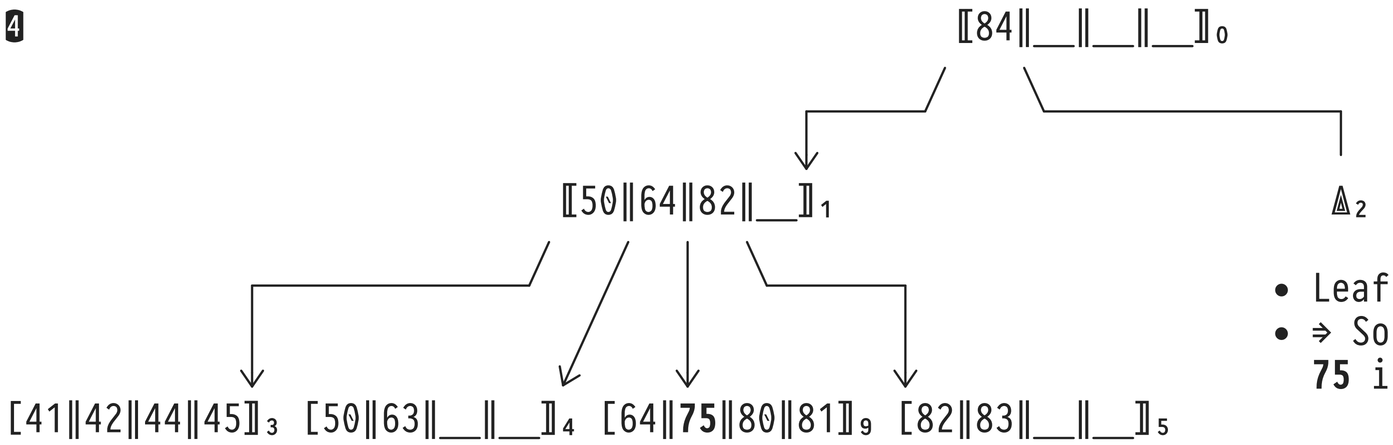
B+Tree Insertion Example: Insert ‹75,*rid*›

3



- Search(**75**) returns leaf 9

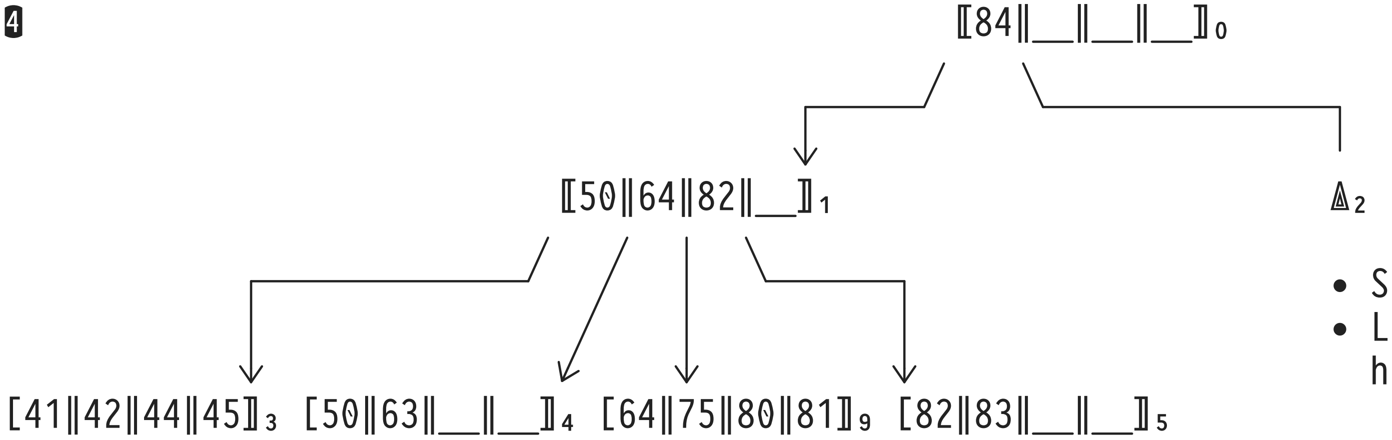
4



- Leaf 9 has capacity
- ⇛ Sorted insert of **75** in leaf 9

B$^+$Tree Insertion Example: Insert ‹77,*rid*›

4



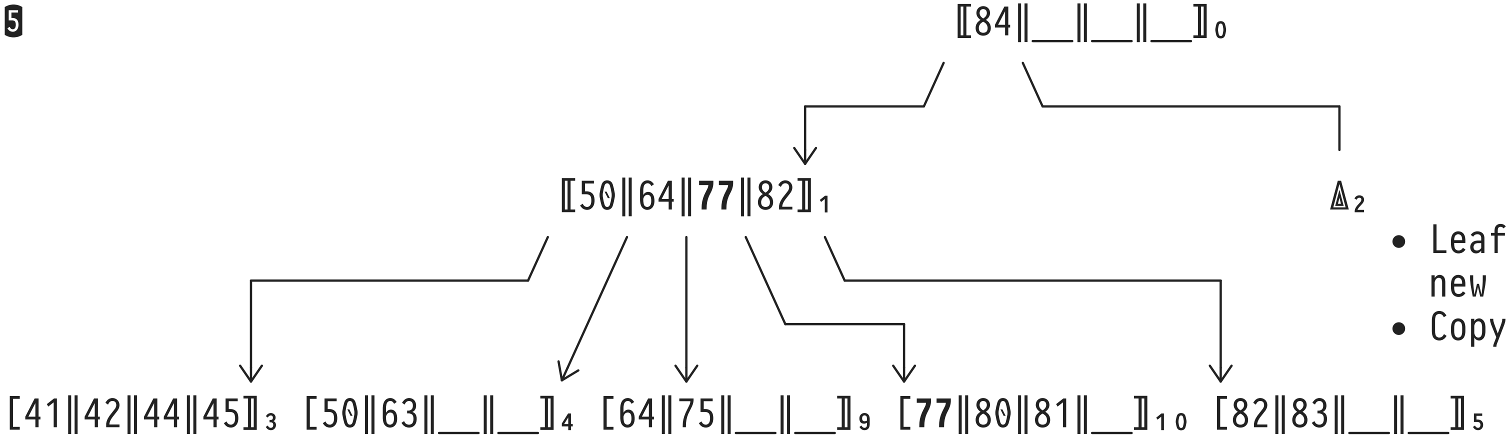
⟦84‖__‖__‖__⟧$_0$

⟦50‖64‖82‖__⟧$_1$ △$_2$

[41‖42‖44‖45⟧$_3$ [50‖63‖__‖__⟧$_4$ [64‖75‖80‖81⟧$_9$ [82‖83‖__‖__⟧$_5$

- Search(**77**) returns leaf 9
- Leaf 9 is full (already holds 2×*o* = 4 entries)

5



⟦84‖__‖__‖__⟧$_0$

⟦50‖64‖**77**‖82⟧$_1$ △$_2$

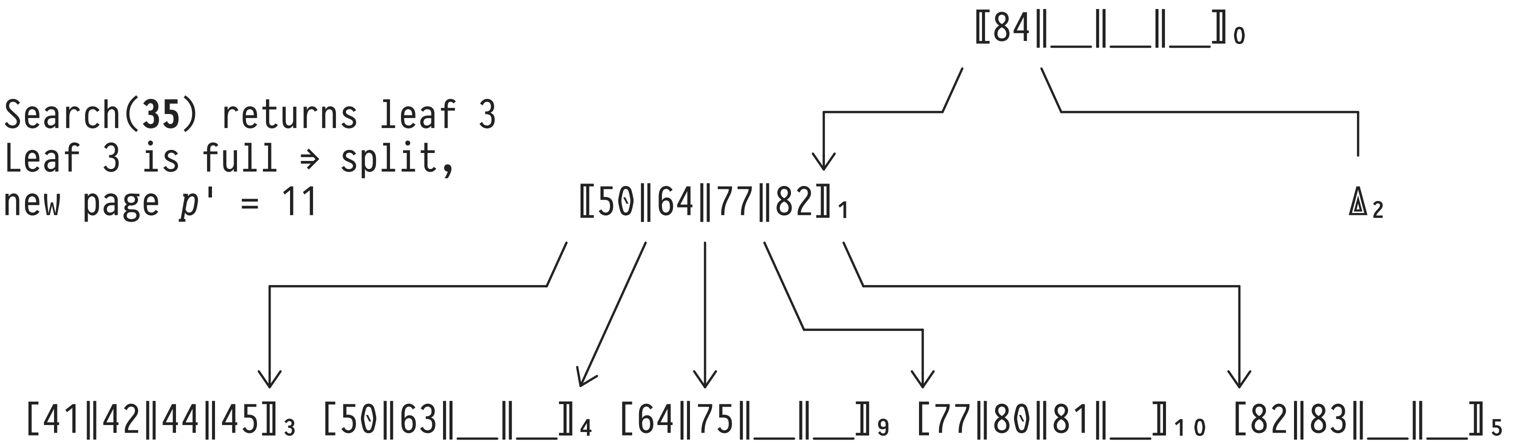
[41‖42‖44‖45⟧$_3$ [50‖63‖__‖__⟧$_4$ [64‖75‖__‖__⟧$_9$ [**77**‖80‖81‖__⟧$_{10}$ [82‖83‖__‖__⟧$_5$

- Leaf 9 has to split, new page *p*' = 10
- Copy up separator ⟦**77**⟧

B$^+$Tree Insertion Example: Insert ‹35,*rid*›

5

- Search(**35**) returns leaf 3
- Leaf 3 is full ⇒ split,
  new page *p*' = 11



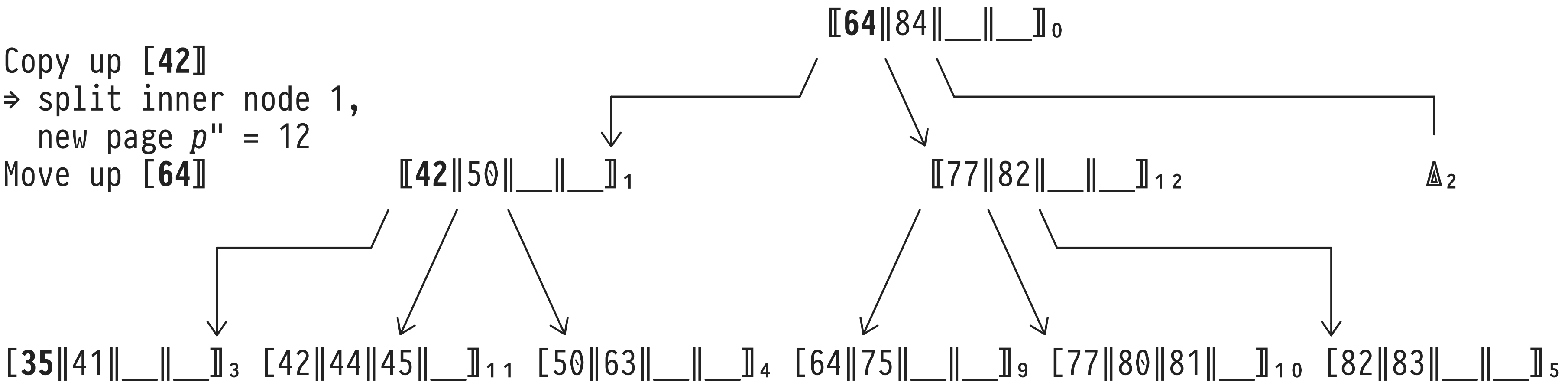
⟦84‖__‖__‖__⟧$_0$

⟦50‖64‖77‖82⟧$_1$ △$_2$

[41‖42‖44‖45⟧$_3$ [50‖63‖__‖__⟧$_4$ [64‖75‖__‖__⟧$_9$ [77‖80‖81‖__⟧$_{10}$ [82‖83‖__‖__⟧$_5$

6

- Copy up [**42**⟧
  ⇒ split inner node 1,
  new page *p*" = 12
- Move up [**64**⟧



⟦**64**‖84‖__‖__⟧$_0$

⟦**42**‖50‖__‖__⟧$_1$ ⟦77‖82‖__‖__⟧$_{12}$ △$_2$

[**35**‖41‖__‖__⟧$_3$ [42‖44‖45‖__⟧$_{11}$ [50‖63‖__‖__⟧$_4$ [64‖75‖__‖__⟧$_9$ [77‖80‖81‖__⟧$_{10}$ [82‖83‖__‖__⟧$_5$

B^+Tree Insertion Algorithm[7] (1)

```
TreeInsert(‹k,rid›,node):
  if (node is a leaf)
    return LeafInsert(‹k,rid›,node);
  else
    switch k                                                 ⎫
      case $k < k_1$                                         ⎪
        ‹sep,ptr› ← TreeInsert(‹k,rid›,$p_0$);               ⎪
      case $k_i \leqslant k < k_{i+1}$                       ⎬ see Search()
        ‹sep,ptr› ← TreeInsert(‹k,rid›,$p_i$);               ⎪
      case $k_{2o} \leqslant k$                              ⎪
        ‹sep,ptr› ← TreeInsert(‹k,rid›,$p_{2o}$);            ⎭
    if (sep = ⊥)                                             ⎫ upwards split?
      return ‹⊥,⊥›;                                          ⎬ ⇒ no
    else                                                     ⎪
      return InnerInsert(‹sep,ptr›,node);                    ⎭ ⇒ yes
```

[7] Note: ‹*sep*,*ptr*› ≡ ⟦*sep*⟧ in our discussion above.

B^+Tree Insertion Algorithm (2)

```
LeafInsert(⟨k,rid⟩,node):
  if (node has < 2×o entries)
    insert ⟨k,rid⟩ into node;
    return ⟨⊥,⊥⟩;                } ⟨⊥,_⟩ ≡ no upwards split required
  else
    p' ← allocate leaf page;
    $[\langle k_1,rid_1\rangle,\ldots,\langle k_{2o+1},rid_{2o+1}\rangle]$ ← entries of node ∪ ⟨k,rid⟩;
    node ← $[k_1|rid_1|\ldots|k_o|rid_o|\_\_\|\_\_⟧$;
    p'   ← $[k_{o+1}|rid_{o+1}|\ldots|k_{2o+1}|rid_{2o+1}|\_\_\|\_\_⟧$;
    return $\langle k_{o+1},p'\rangle$;            } new separator to be copied upwards
```

- **Copy** upwards: entry $\langle k_{o+1},rid_{o+1}\rangle$ remains in leaf p'.

B^+Tree Insertion Algorithm (3)

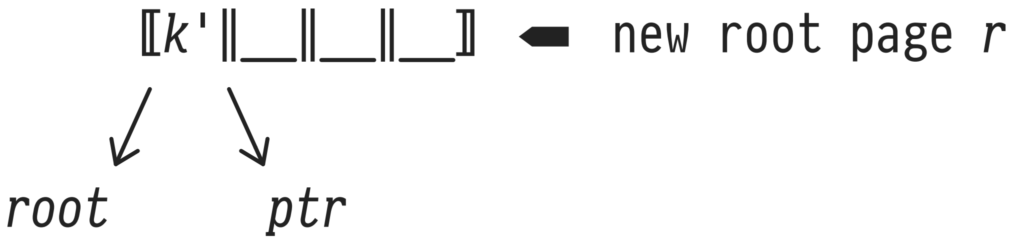
```
InnerInsert(⟨sep,ptr⟩,node):
  if (node has < 2×o entries)
  | insert ⟨sep,ptr⟩ into node;
  | return ⟨⊥,⊥⟩;                } ⟨⊥,_⟩ ≡ no upwards split required
  else
  | p' ← allocate inner node page;
  | $[p_0,\langle k_1,p_1\rangle,\dots,\langle k_{2o+1},p_{2o+1}\rangle]$ ← entries of node ∪ ⟨sep,ptr⟩;
  | node ← $[p_0|k_1|p_1|\dots|k_o|p_o|\_\_\|\_\_⟧$;
  | p'   ← $[p_{o+1}|k_{o+2}|p_{o+2}|\dots|k_{2o+1}|p_{2o+1}|\_\_\|\_\_⟧$;
  ⌊ return $\langle k_{o+1},p'\rangle$;       } new separator to be moved upwards
```

- **Move** upwards: new entry $\langle k_{o+1},p'\rangle$ returned for insertion at parent. No entry $\langle k_{o+1},\_\rangle$ remains at level of *node*/*p'*.

B⁺Tree Insertion Algorithm (Top Level)

Insert(‹*k*,*rid*›) is the top-level B⁺Tree insertion routine:

```
Insert(‹k,rid›):
  ‹k',ptr› ← TreeInsert(‹k,rid›,root);  } root ≡ old root page
  if (k' ≠ ⊥)
   │ r ← [root|k'|ptr|__‖__‖__];        } r ≡ new root page
   └ root ← r
```



⟦*k*'‖__‖__‖__⟧ ← new root page *r*

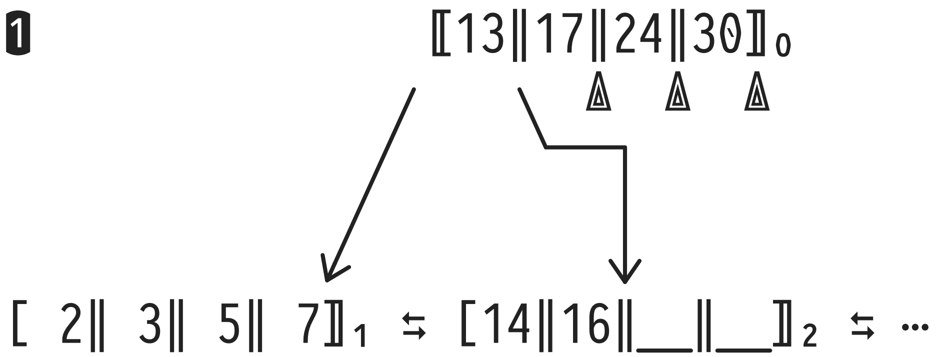
root

ptr

- Note: Insert() may leave us with a new root node that violates the minimum occupancy rule. ¯\(ツ)/¯

B⁺Tree Insertion: Redistribution (1)

Can improve average occupancy and delay height increase on B⁺Tree insertion through **redistribution:**



❶

⟦13‖17‖24‖30⟧$_0$

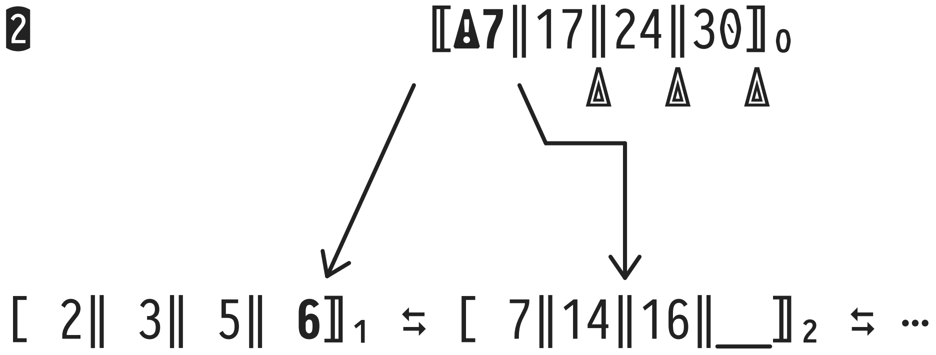
⍋ ⍋ ⍋

[2‖ 3‖ 5‖ 7⟧$_1$ ⇆ [14‖16‖__‖__⟧$_2$ ⇆ …

❶ Insert new entry ‹**6**,*rid*›
- Search(**6**) returns leaf 1
- Leaf 1 is full, but its right **sibling** 2 has capacity

- Use sequence set chain pointers (⇆) to inspect **sibling** nodes for spare capacity.
- **Push** entry from overflowing node to sibling and ⚠️ **update separator in parent node** to reflect this redistribution.

B$^+$Tree Insertion: Redistribution (2)



❷

⟦▲**7**‖17‖24‖30⟧$_0$

△ △ △

[2‖ 3‖ 5‖ **6**⟧$_1$ ⇆ [7‖14‖16‖__⟧$_2$ ⇆ ⋯

❷ Push entry ‹7,*rid*'› to leaf 2

- Place ‹**6**,*rid*› in leaf 1
- Update separator (13 → **7**) in parent node 0
- B$^+$Tree remains at height 2

- Inspecting node sibling involves additional page I/O. 👎
- Actual implementations use redistribution on the index leaf level only (if at all).

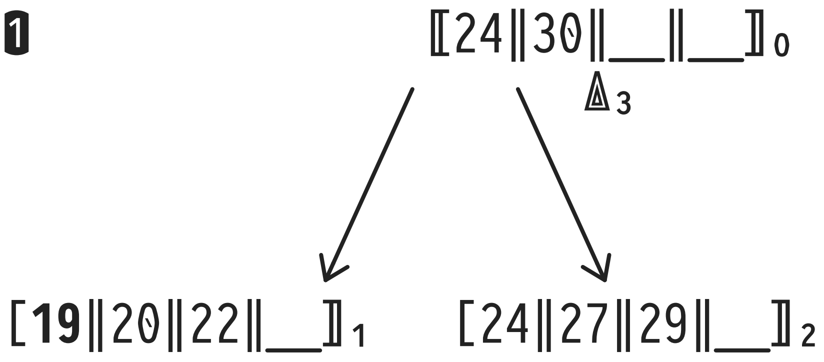
7 ┊ B$^+$Tree Deletion of Entry With Key k (Sketch)

1. Use Search(k) to **find the leaf** p holding entry ‹k,rid›.
2. **Simply delete** ‹k,rid› from p.[8]
3. If p now holds < o entries, leaf p **underflows.** Any sibling of p with spare entries?
   - ○ Yes, use **redistribution** to move an entry into p.
   - ○ No, **merge** p and a sibling leaf p' of o entries. Delete⟳ the now obsolete separator of p and p' in their parent node.

   Deletion propagates upwards and may eventually leave the root node empty (decreases B$^+$Tree height).

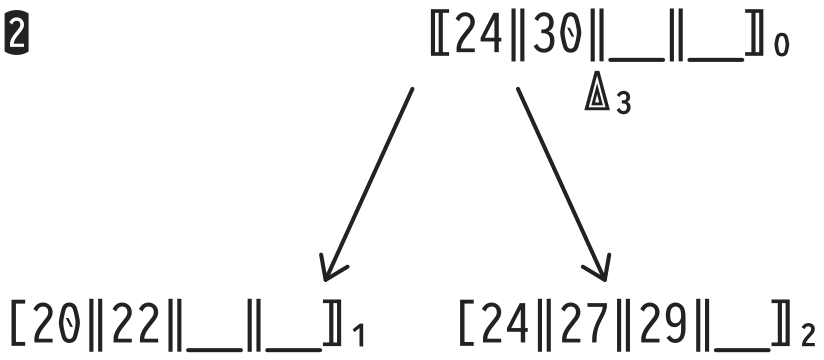
[8] **Q:** If ‹k,rid› is the leftmost entry in p, do we need to update the associated separator entry in p's parent node? Why not?

B+Tree Deletion (No Underflow)



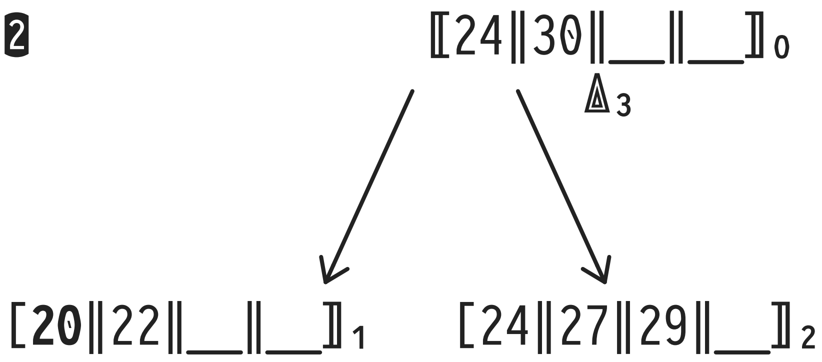
1 Delete entry with key *k* = **19**

- Search(**19**) returns leaf 1
- Leaf 1 has > *o* entries, node will not underflow



2 Simply delete entry ‹**19**,*rid*› from leaf 1

B$^+$Tree Deletion and Redistribution



❷

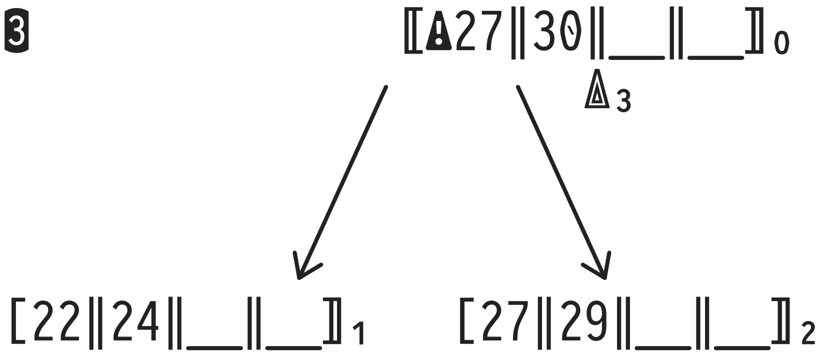
⟦24‖30‖__‖__⟧$_0$

△$_3$

[**20**‖22‖__‖__⟧$_1$ [24‖27‖29‖__⟧$_2$

❷ Delete entry with key *k* = **20**

- Search(**20**) returns leaf 1
- Leaf 1 has minimum occupancy of *o* entries ⇛ will underflow



❸

⟦⚠27‖30‖__‖__⟧$_0$

△$_3$

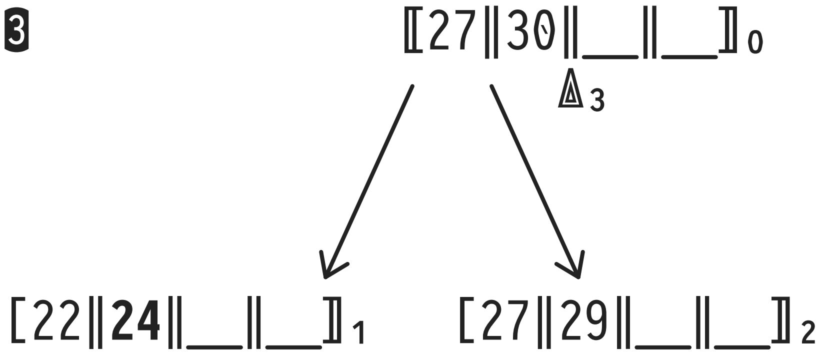
[22‖24‖__‖__⟧$_1$ [27‖29‖__‖__⟧$_2$

❸ Sibling *p*' = 2 has one entry to spare ⇛ redistribution

- Move entry ‹24,*rid*'› from leaf 2 to leaf 1
- Update separator (24 → 27) in parent node 0

B+Tree Deletion and Leaf Node Merging

❸



⟦27‖30‖__‖__⟧$_0$

⟁$_3$

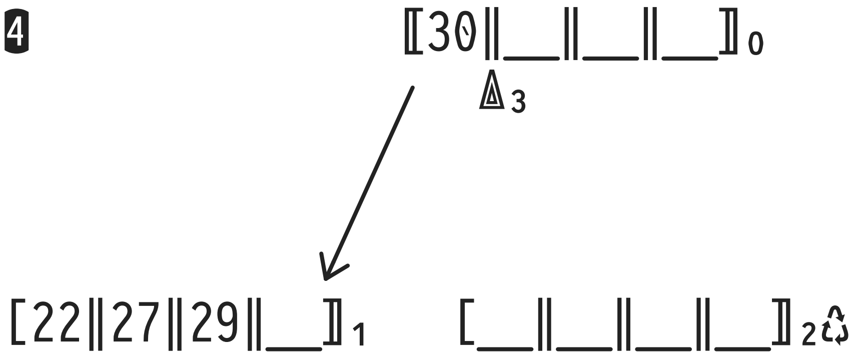
[22‖**24**‖__‖__⟧$_1$

[27‖29‖__‖__⟧$_2$

❸ Delete entry with key k = **24**

- Search(**24**) returns leaf 1
- Leaf 1 has minimum occupancy, no sibling with spare entries

❹



⟦30‖__‖__‖__⟧$_0$

⟁$_3$

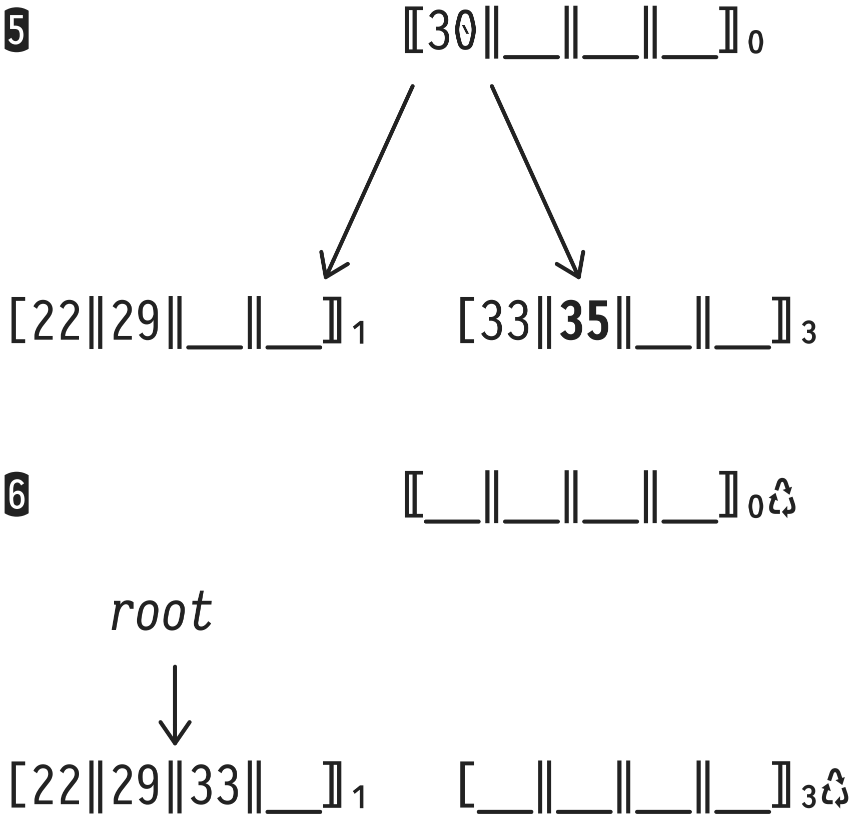
[22‖27‖29‖__⟧$_1$

[__‖__‖__‖__⟧$_2$♻

❹ Merge leaf nodes 1 and 2, mark empty page 2 as garbage

- In parent 0, delete⟲ obsolete separator [27⟧

B⁺Tree Deletion and Leaf Node Merging (Empty Root)



❺ ⟦30‖__‖__‖__⟧$_0$

[22‖29‖__‖__⟧$_1$ [33‖**35**‖__‖__⟧$_3$

❻ ⟦__‖__‖__‖__⟧$_0$♺

root

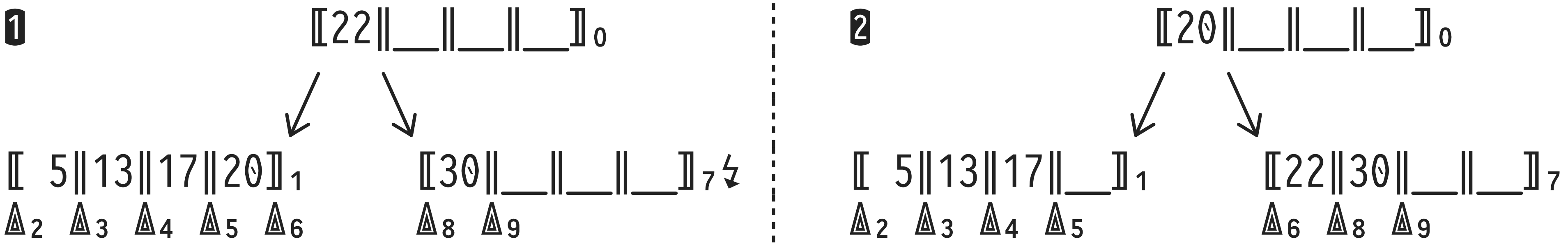
[22‖29‖33‖__⟧$_1$ [__‖__‖__‖__⟧$_3$♺

❺ Delete entry with key *k* = **35**
- Search(**35**) returns leaf 3
- Leaf 3 has minimum occupancy, no sibling with spare entries

❻ Merge leaf nodes 1 and 3, mark empty page 3 as garbage
- In parent 0, delete⟲ obsolete separator [30⟧
- Old root empty (⇒ garbage), mark page 1 as the new root
- B⁺Tree height decreases

B⁺Tree Deletion and Inner Node Redistribution

- **Redistribution** is also defined for **inner nodes.** Suppose we encounter underflow ❶ during ↻ deletion propagation:

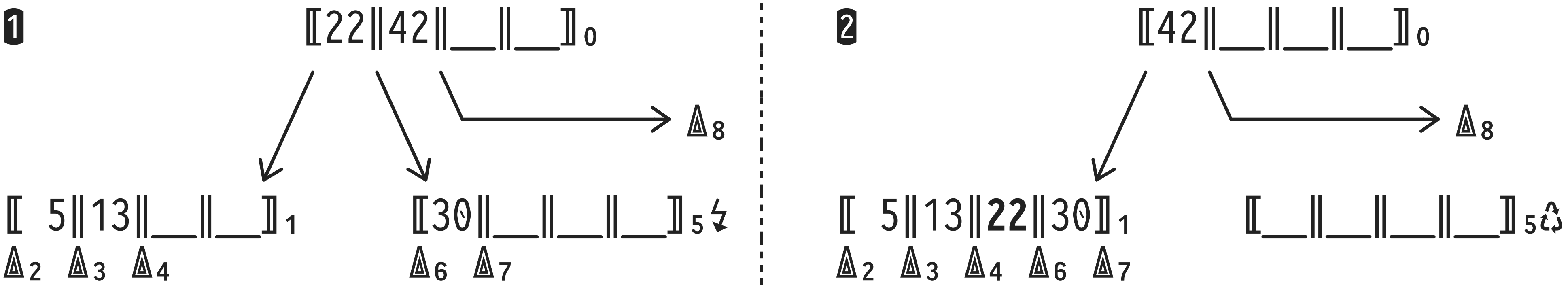


- Inner node 1 has two spare entries. “Rotate entry [20⟧ through parent” to underflowed inner node 7.

  NB: Semantics of subtree $\triangle_6$ (holds index entries with $k \geqslant 20 \wedge k < 22$) are preserved.

B⁺Tree Deletion and Inner Node Merging

- Likewise, **inner nodes** may also be **merged.** The underflow in ❶ cannot be handled by redistribution:



❶

⟦22‖42‖__‖__⟧$_0$

$\triangle_8$

⟦ 5‖13‖__‖__⟧$_1$ ⟦30‖__‖__‖__⟧$_5$ϟ

$\triangle_2$ $\triangle_3$ $\triangle_4$ $\triangle_6$ $\triangle_7$

❷

⟦42‖__‖__‖__⟧$_0$

$\triangle_8$

⟦ 5‖13‖**22**‖30⟧$_1$ ⟦__‖__‖__‖__⟧$_5$♻

$\triangle_2$ $\triangle_3$ $\triangle_4$ $\triangle_6$ $\triangle_7$

- Note how the separator **22** has been **pulled down**↻ from the parent to discriminate between subtrees $\triangle_4$ and $\triangle_6$:
  - $\triangle_4$: $k \geqslant 13 \land k < 22$
  - $\triangle_6$: $k \geqslant 22 \land k < 30$

8 ¦ B+Trees: Key Compression

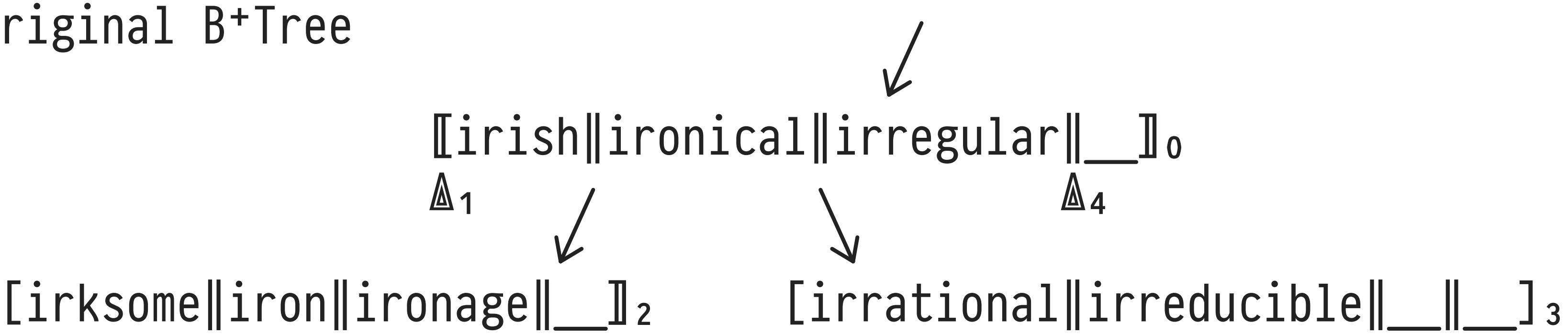
The higher the **fan-out** F, the more index entries fit in a B+Tree of fixed height. How to maximize F?

- For entries $\langle k,p \rangle$ in indexes over text/char columns, we may have $|k| \gg |p|$.[9] Can we reduce the size of k?
- 💡 Search() and TreeInsert() do *not* inspect the actual key values but only use $<$/$\leqslant$ to direct tree traversals.
    - ⇒ May **shorten (truncate) string keys** as long as the ordering relation is preserved.
    - This applies to index entries in inner nodes only. Leaf level keys remain as is.

[9] The implementation (thus size) of page pointers p is prescribed by the DBMS. Nothing to win here.

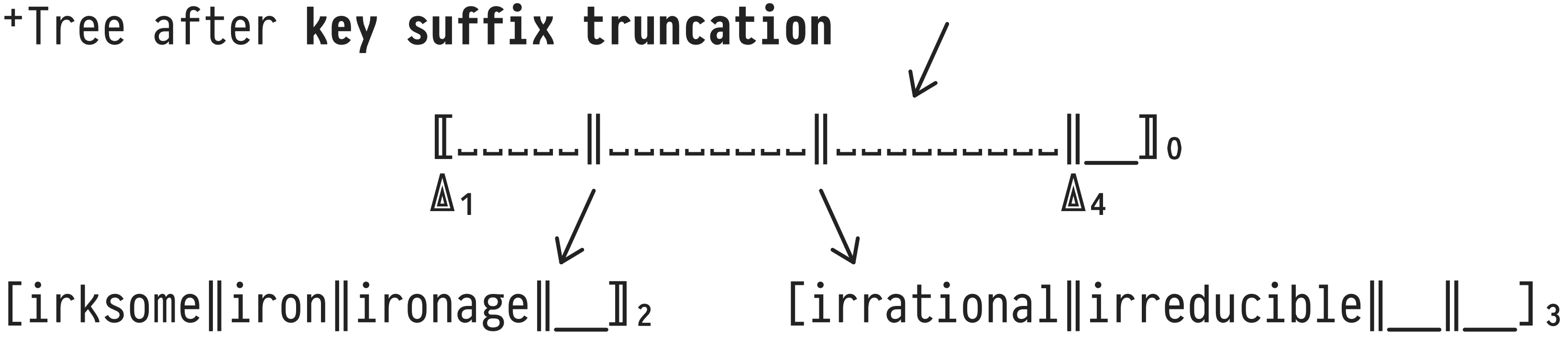
B+Trees: Key Suffix Truncation

❶ Original B+Tree



```
                                              ↙
              ⟦irish‖ironical‖irregular‖__⟧0
              ⍋1     ↙          ↘          ⍋4
[irksome‖iron‖ironage‖__⟧2     [irrational‖irreducible‖__‖__]3
```

❷ B+Tree after **key suffix truncation**



```
                                              ↙
              ⟦␣␣␣␣␣‖␣␣␣␣␣␣␣␣‖␣␣␣␣␣␣␣␣␣‖__⟧0
              ⍋1     ↙          ↘          ⍋4
[irksome‖iron‖ironage‖__⟧2     [irrational‖irreducible‖__‖__]3
```

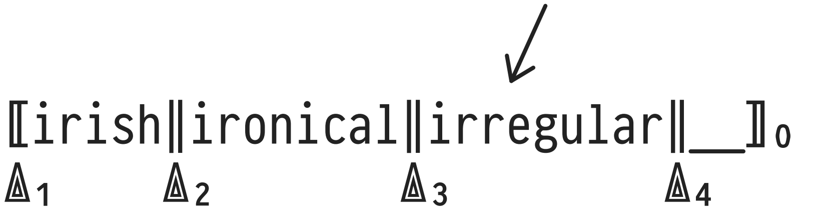
⚠️ While truncating, preserve the **separator** semantics.

B+Trees: Key Prefix Compression

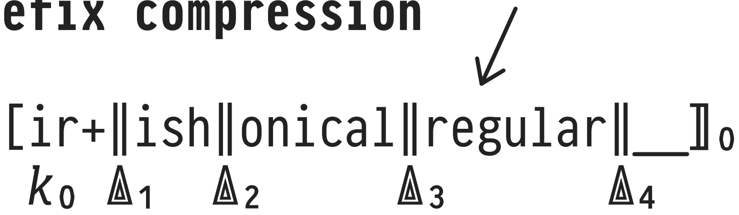
Observation: string keys within a B+Tree inner node often **share a common prefix.**

- 💡 Store common prefix only once (e.g., as "k_0").
- Violating the 50% occupancy rule can help compression.

❶ Original B+Tree


⟦irish‖ironical‖irregular‖__⟧$_0$
$\mathbb{A}_1$ $\mathbb{A}_2$ $\mathbb{A}_3$ $\mathbb{A}_4$

❷ B+Tree after **key prefix compression**


[ir+‖ish‖onical‖regular‖__⟧$_0$
k_0 $\mathbb{A}_1$ $\mathbb{A}_2$ $\mathbb{A}_3$ $\mathbb{A}_4$

9 ¦ B⁺Tree Bulk Loading

Grab a hot cup of ☕ and start a war on Stack Overflow:[10]

Q: *Which order of operations is better?*

❶ **CREATE TABLE** *T* (…);
❷ **INSERT INTO** *T* **VALUES** (‹5×10^6 *rows*›);
❸ **CREATE INDEX** *I* **ON** *T* **USING** btree (…);

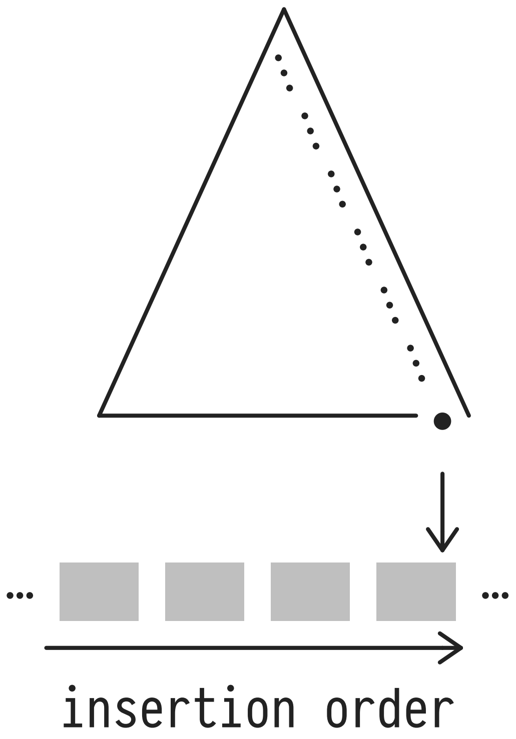
or

❶ **CREATE TABLE** *T* (…);
❸ **CREATE INDEX** *I* **ON** *T* **USING** btree (…); ◀┐
❷ **INSERT INTO** *T* **VALUES** (‹5×10^6 *rows*›); ◀┘

[10] See, for example, https://stackoverflow.com/questions/5910486/indexes-on-a-table-database

B$^+$Tree Bulk Loading

If insertions happen in index key order (i.e., ascending values of k), we observe a particular B$^+$Tree access pattern:



- TreeInsert() will always traverse path ⋱, will always hit the righmost leaf.

⇛ Fix rightmost leaf in buffer, insert next entry right there (*no* traversal from root). Node splits only occur along path ⋱.

- We effectively create a clustered index.

heap file (sorted on keys k)

10 ┆ Q_8 — Filtering a Table

```
SELECT i.b, i.c
FROM   indexed AS i
WHERE  i.a = 42 [i.c = 0.42] -- either filter on i.a or i.c
```

Indexes in MonetDB play a secondary role and are *not* organized in tree shapes.

MMDBMSs try to exploit that data resides in directly-adressable memory and primarily aim to avoid access to separate index data structures (to avoid pointer chasing and potential cache misses).

Using EXPLAIN on Q_8: Filter on Column a

```
sql> EXPLAIN SELECT i.b, i.c
             FROM   indexed AS i
             WHERE  i.a = 42;
   ⋮
  indexed :bat[:oid] := sql.tid(sql, "sys", "indexed");
  a0      :bat[:int] := sql.bind(sql, "sys", "indexed", "a", 0:int);
  p1      :bat[:oid] := algebra.thetaselect(a0, indexed, 42:int, "=="); ◀ ≡ a = 42
  c0      :bat[:sht] := sql.bind(sql, "sys", "indexed", "c", 0:int);
  c       :bat[:sht] := algebra.projection(p1, c0);
  b0      :bat[:str] := sql.bind(sql, "sys", "indexed", "b", 0:int);
  b       :bat[:str] := algebra.projection(p1, b0);
   ⋮
```

- MonetDB uses `algebra.thetaselect(…, 42:int, "==")` to implement the predicate filter.

Using EXPLAIN on Q_8: Filter on Column c[11]

```
sql> EXPLAIN SELECT i.b, i.c
             FROM   indexed AS i
             WHERE  i.c = 0.42;
   ⋮
  indexed :bat[:oid] := sql.tid(sql, "sys", "indexed");
  c0      :bat[:sht] := sql.bind(sql, "sys", "indexed", "c", 0:int);
  p1      :bat[:oid] := algebra.thetaselect(c0, indexed, 42:sht, "=="); ⬅ ≡ c = 0.42
  c       :bat[:sht] := algebra.projection(p1, c0);
  b0      :bat[:str] := sql.bind(sql, "sys", "indexed", "b", 0:int);
  b       :bat[:str] := algebra.projection(p1, b0);
   ⋮
```

- Plan is nearly identical (modulo access to the a BAT).
- MonetDB *appears* to use the same `algebra.thetaselect(…, 42:sht, "==")` MAL operation.

[11] Note how MonetDB maps the domain of type numeric(3,2) of column c, i.e., the set $N_{3,2} \equiv \{-9.99,…,9.99\}$ with $|N_{3,2}| = 1999$, to a 16-bit value of type :sht. Nifty.

BAT Tail Properties

When MonetDB constructs a BAT t, a family of tail column **properties** $prop(t)$ is derived/maintained:[12]

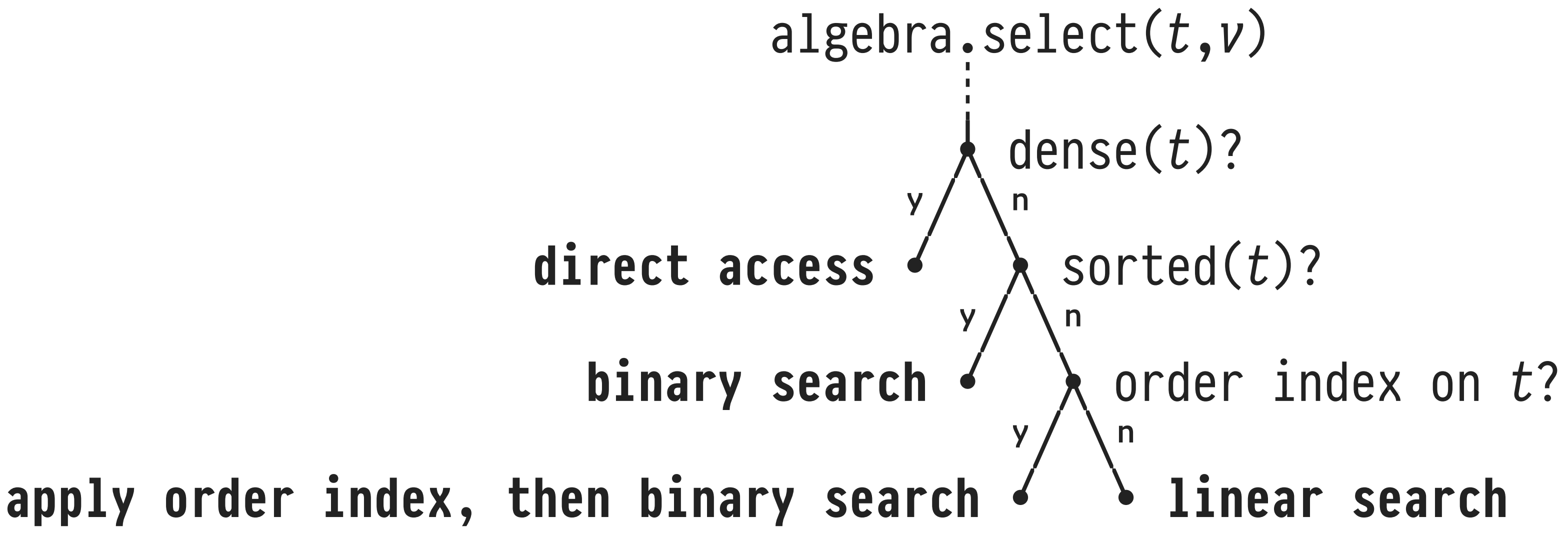
BAT Property $prop(t)$	Description
dense (tails of type :oid only)	ascending values, no gaps
key	unique values
sorted	ascending values
revsorted	descending values
nil/nonil	at least one/no nil value

- Use bat.info(t) to inspect current properties of t.
- ⚠ Incomplete: t's tail may be sorted although sorted(t) = false (⇒ but not ⇔).

[12] Additional properties *nokey*, *nosorted*, *norevsorted* give “proofs” (tail positions) why property does not hold. Example: nosorted = 3 ≡ tail value for row 3@0 < tail value for row 2@0.

MonetDB: Tactical Optimization

MAL operations inspect BAT properties at *query runtime*, select one of several efficient implementations:



- This is coined **tactical optimization** (as opposed to strategical query optimization at *query compile time*).

The Tactics of algebra.select: dense(*t*)

If input BAT *t* is **dense**, use **positional access** and **slicing** to evaluate equality and range selections:

algebra.select(*t*,42@0)

head	tail
0@0	39@0
1@0	40@0
2@0	41@0
3@0	42@0
4@0	43@0
5@0	44@0

offset 3 = 42@0-39@0

hseqbase(*t*)

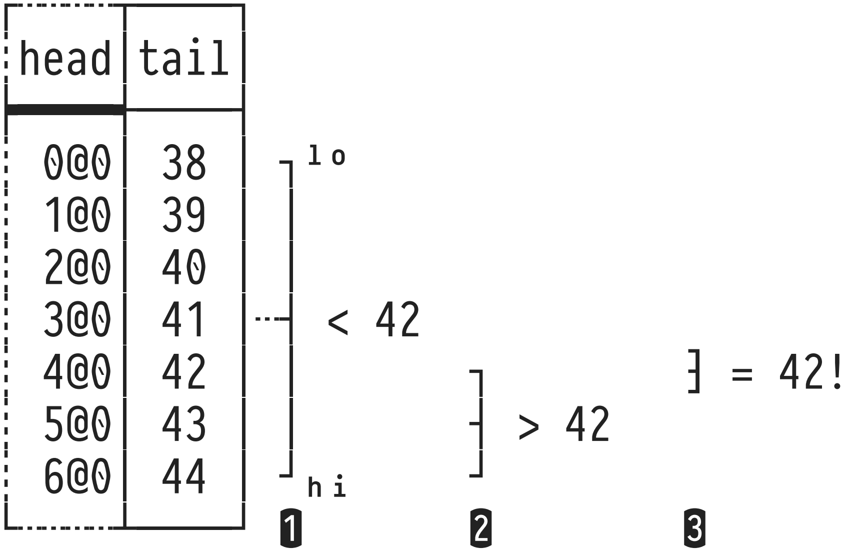
algebra.select(*t*,40@0,42@0,*t*,*t*,*f*)

head	tail
0@0	39@0
1@0	40@0
2@0	41@0
3@0	42@0
4@0	43@0
5@0	44@0

↕ ≡ algebra.slice(*t*,1,3)

The Tactics of algebra.select: sorted(t)

algebra.select(t,42)



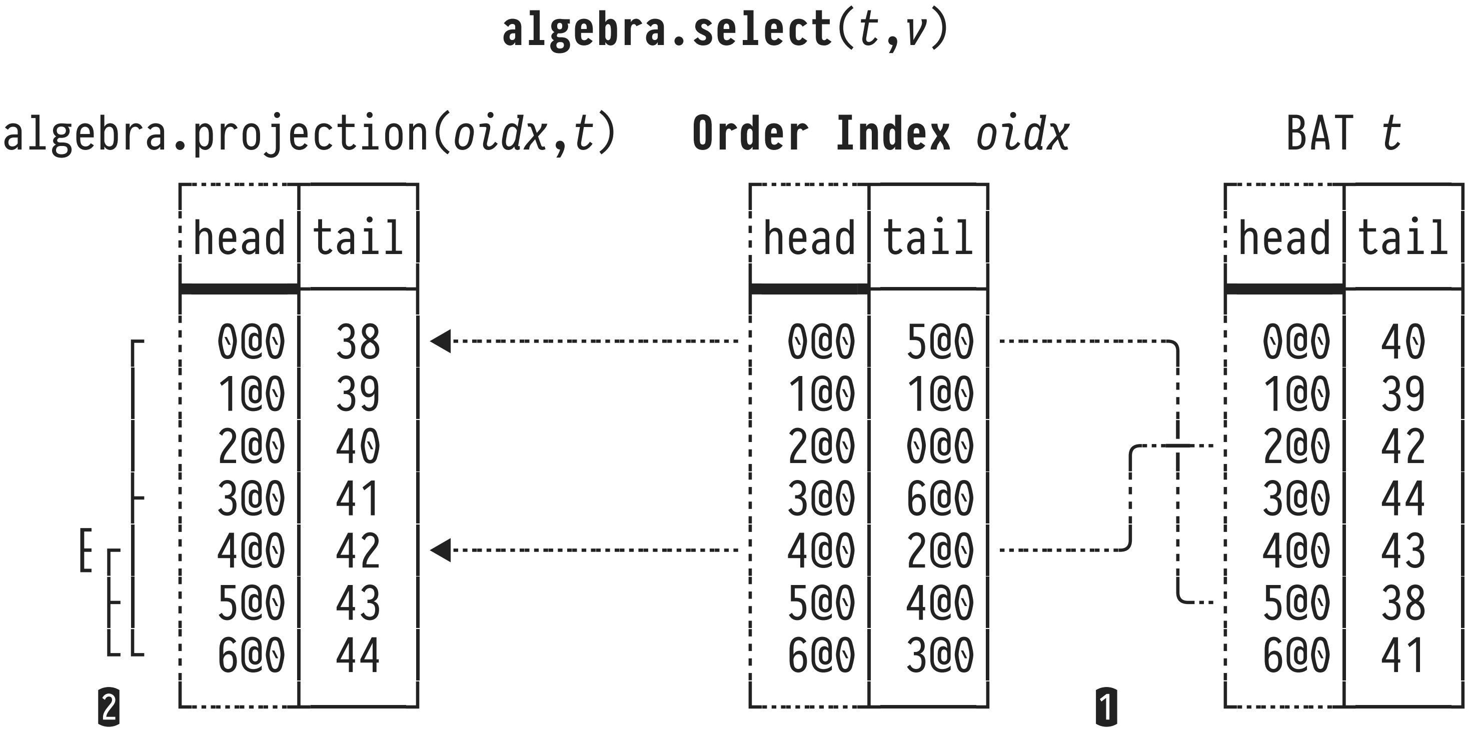
head	tail
0@0	38
1@0	39
2@0	40
3@0	41
4@0	42
5@0	43
6@0	44

Binary Search:

- Test middle value (pivot) between limits *lo* and *hi*
- Recurse into upper or lower partition based on test
- Finishes in $\log_2(|t|)$ steps

- **NB:** Unpredictable branches (≶ 42?) and jumps of pivot position less than ideal for CPU.

The Tactics of algebra.select: Order Indexes



algebra.select(*t*,*v*)

algebra.projection(*oidx*,*t*)

head	tail
0@0	38
1@0	39
2@0	40
3@0	41
4@0	42
5@0	43
6@0	44

❷

Order Index *oidx*

head	tail
0@0	5@0
1@0	1@0
2@0	0@0
3@0	6@0
4@0	2@0
5@0	4@0
6@0	3@0

❶

BAT *t*

head	tail
0@0	40
1@0	39
2@0	42
3@0	44
4@0	43
5@0	38
6@0	41

- Row [*i*@0,*j*@0] ∈ *oidx*: value at offset *j* is *i*th largest in tail. Tactic: ❶ Apply *oidx*, ❷ then use binary search.

Creating Order Indexes (On the Fly)

MonetDB may *automatically* create a temporary order index to support predicates $lo \leqslant a \leqslant hi$ or other order-sensitive queries (e.g., ORDER BY, GROUP BY).

- Check current properties of column BATs and presence of indexes in MonetDB system table sys.storage:

```
sql> SELECT column, sorted, revsorted, "unique", orderidx
     FROM   sys.storage('sys', 'indexed');
+----------------+--------+-----------+--------+----------+
| column         | sorted | revsorted | unique | orderidx |
+================+========+===========+========+==========+
| a              | true   | null      | true   |        0 |
| b              | null   | null      | null   |        0 |
| c              | false  | false     | null   |        0 |
+----------------+--------+-----------+--------+----------+
```

Creating Order Indexes (Manually)

If this seems beneficial for the **query workload**, clients may *manually* create an order index.

- ⚠ Order indexes are **static** (i.e., not maintained under updates—costly) ⇒ underlying table must be *read-only*:

  ```
  ‹create and populate table T›
  sql> ALTER TABLE T SET READ ONLY;
  sql> CREATE ORDERED INDEX I ON T(a);
  ```

  ○ Order index *I* is made persistent (in a `*.torderidx` disk file) and will be used by future `algebra.select()`s on column *a*.

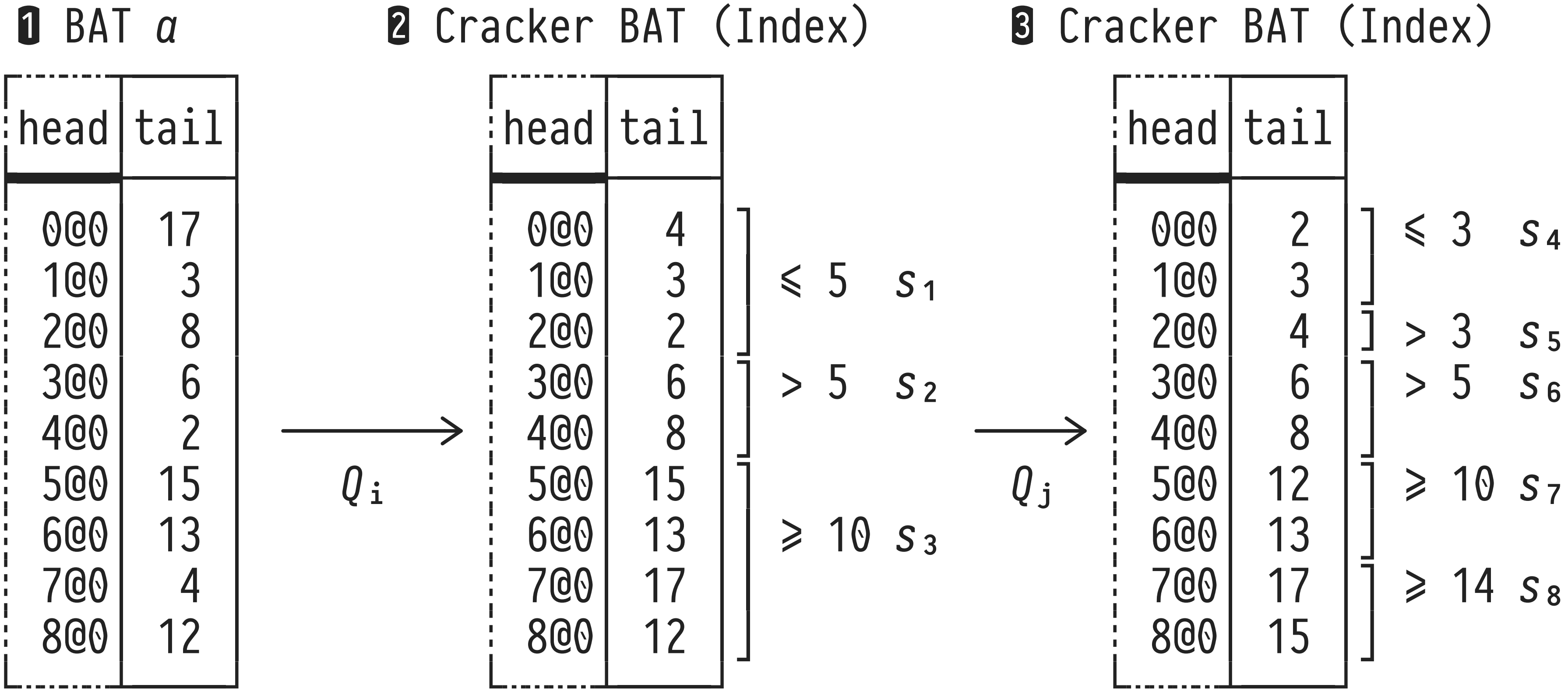
11 ┊ Column Cracking

With **column cracking,**[13] MonetDB introduced a **self-organizing** (partially) ordered index structure.

- A **cracker index** for column a is created/updated as a **by-product of processing range predicates** $lo \leqslant a \leqslant hi$.
    - In the cracker index, the a values $\in [lo,hi]$ are stored physically contiguous.
- If the **query workload** focuses only on a subset of column a, that part is indexed with fine granularity (while the other parts remain largely non-indexed).

[13] *“Database Cracking”*, S. Idreos, M. Kersten, S. Manegold. Proc. CIDR, Asilomar (CA, USA), 2007.

Column Cracking As a By-Product of Query Processing



❶ BAT *a*

head	tail
0@0	17
1@0	3
2@0	8
3@0	6
4@0	2
5@0	15
6@0	13
7@0	4
8@0	12

$\xrightarrow{\quad}$ Q_i

❷ Cracker BAT (Index)

head	tail	
0@0	4	⩽ 5 s_1
1@0	3	
2@0	2	
3@0	6	> 5 s_2
4@0	8	
5@0	15	⩾ 10 s_3
6@0	13	
7@0	17	
8@0	12	

$\xrightarrow{\quad}$ Q_j

❸ Cracker BAT (Index)

head	tail	
0@0	2	⩽ 3 s_4
1@0	3	
2@0	4	> 3 s_5
3@0	6	> 5 s_6
4@0	8	
5@0	12	⩾ 10 s_7
6@0	13	
7@0	17	⩾ 14 s_8
8@0	15	

- Q_i: … **WHERE** *a* > 5 **AND** *a* < 10 ┊ Result: slice s_2
- Q_j: … **WHERE** *a* > 3 **AND** *a* < 14 ┊ Result: slices $s_5 + s_6 + s_7$

Column Cracking Notes

- $\forall x \in s_i,\ y \in s_{i+1}: x < y$: a fully cracked column ($\forall_i |s_i| = 1$) is completely ordered. This is uncommon (workload skew).

- First cracking step (❶→❷) copies source BAT. All further steps physically reorganize the cracker BAT.

- MonetDB implements slicing in terms of *views*[14] of the cracker BAT, no data copying involved. Cost free.

- Physical cracker index reorganization (“tail shuffling”) can be efficiently performed *in-situ*.

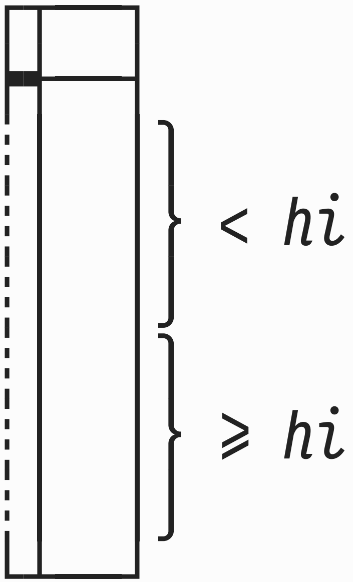
[14] A possible BAT view: (*source BAT*, *first row*, *last row*).

Cracker Index Reorganization For Predicate $a < hi$

Reorganize column vector $a[]$ between row offsets $start$ and end, relocate its elements *in-situ*:

```
CrackInTwo($a$,$start$,$end$,$hi$):
 while ($start < end$)
    if ($a[start] < hi$)
       $start \leftarrow start + 1$;
    else
       while ($a[end] \geq hi \land end > start$)
          $end \leftarrow end - 1$;
       swap($a[start]$, $a[end]$);  ⁂
       $start \leftarrow start + 1$;
       $end \leftarrow end - 1$;
```

Result



- ⁂ Either $a[start] \geq hi \land a[end] < hi$ or $start = end$.