Databases and Information Systems

Views, Triggers, Transactions, Indices

Dmitriy Traytel
slides partly by Marcos Vaz Salles

UNIVERSITY OF COPENHAGEN

Boyce-Codd Normal Form (BCNF)

- Relation R with FDs F is in **BCNF** if, for all $X \rightarrow Y$ in F^+
  - $Y \subseteq X$ (called a trivial FD), or
  - X contains a key for R.

- In other words, R is in BCNF if the only non-trivial FDs that hold over R are key constraints.
  - No dependency in R that can be predicted using FDs alone.
  - If we are shown two tuples that agree upon the A value, we cannot infer the C value in one tuple from the C value in the other.
  - If example relation was in BCNF and A → C, the 2 tuples would have to be identical (A is a key).

A	B	C
10	1	3
10	2	?

Source: Ramakrishnan & Gehrke

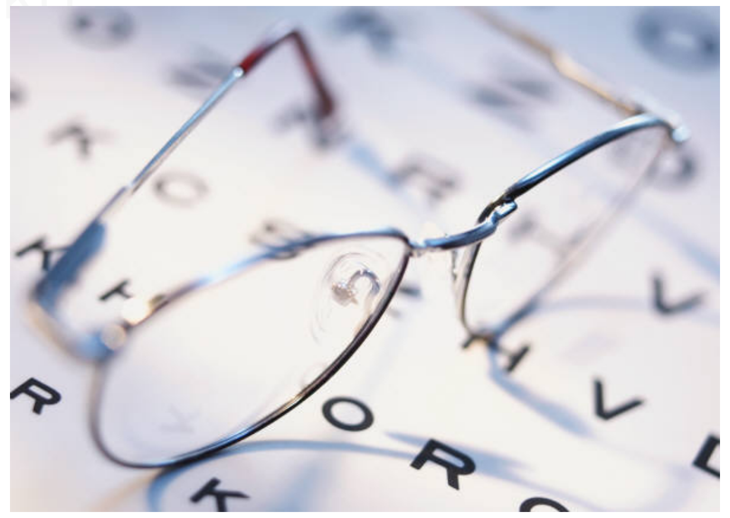
Decomposition into BCNF

- Consider relation R with FDs F. If X → Y violates BCNF, decompose R into R - Y and XY.
  - Repeated application of this idea will give us a collection of relations that are in BCNF; lossless-join decomposition, and guaranteed to terminate.
  - e.g., CSJDPQV, key C, JP → C, SD → P, J → S
  - To deal with SD → P, decompose into SDP, CSJDQV.
  - To deal with J → S, decompose CSJDQV into JS and CJDQV
- In general, several dependencies may cause violation of BCNF. The order in which we "deal with" them could lead to different sets of relations!

Exercise

- Courses(course,teacher,hour,room,student,grade)
- Abbreviated CTHRSG
- FDs: $C \rightarrow T$, $HR \rightarrow C$, $HT \rightarrow R$, $HS \rightarrow R$, $CS \rightarrow G$
- Determine all the keys
- Decompose into BCNF

What should we learn today?



- Explain the concept of (**materialized**) **views** and create them in SQL
- Explain the concept of **triggers** and create them in SQL
- Argue for when to use (**materialized**) **views** vs. **triggers**
- Explain the **ACID** properties of transactions
- Formulate **transaction programs** in SQL
- Identify the main types of representations and access methods for relations in DBMS, namely **heap files** and **indexes**
- Explain the core algorithms for insertion, deletion, and search in a **B$^+$-tree**

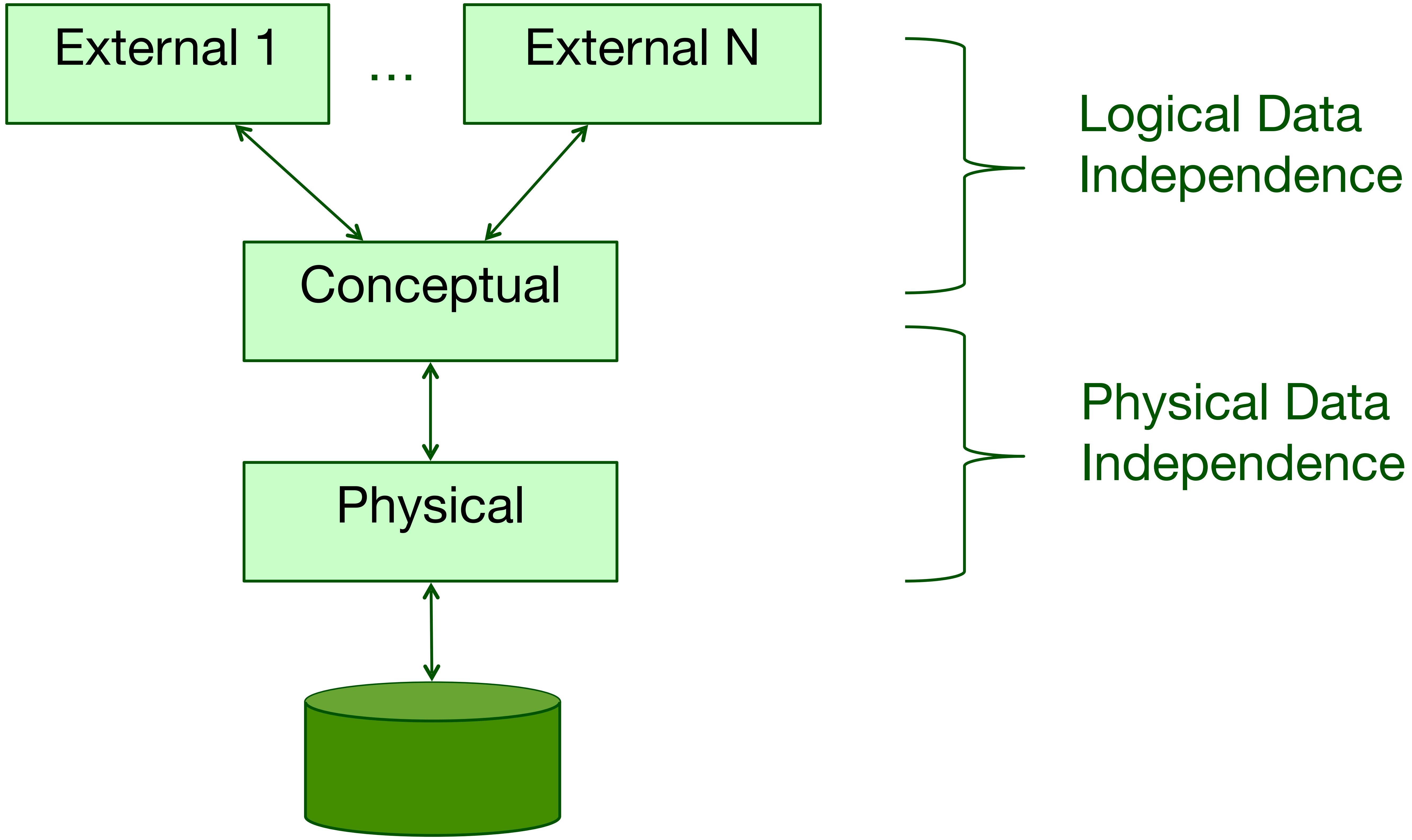
Views

- Special construct to add logical table to **schema** of the database

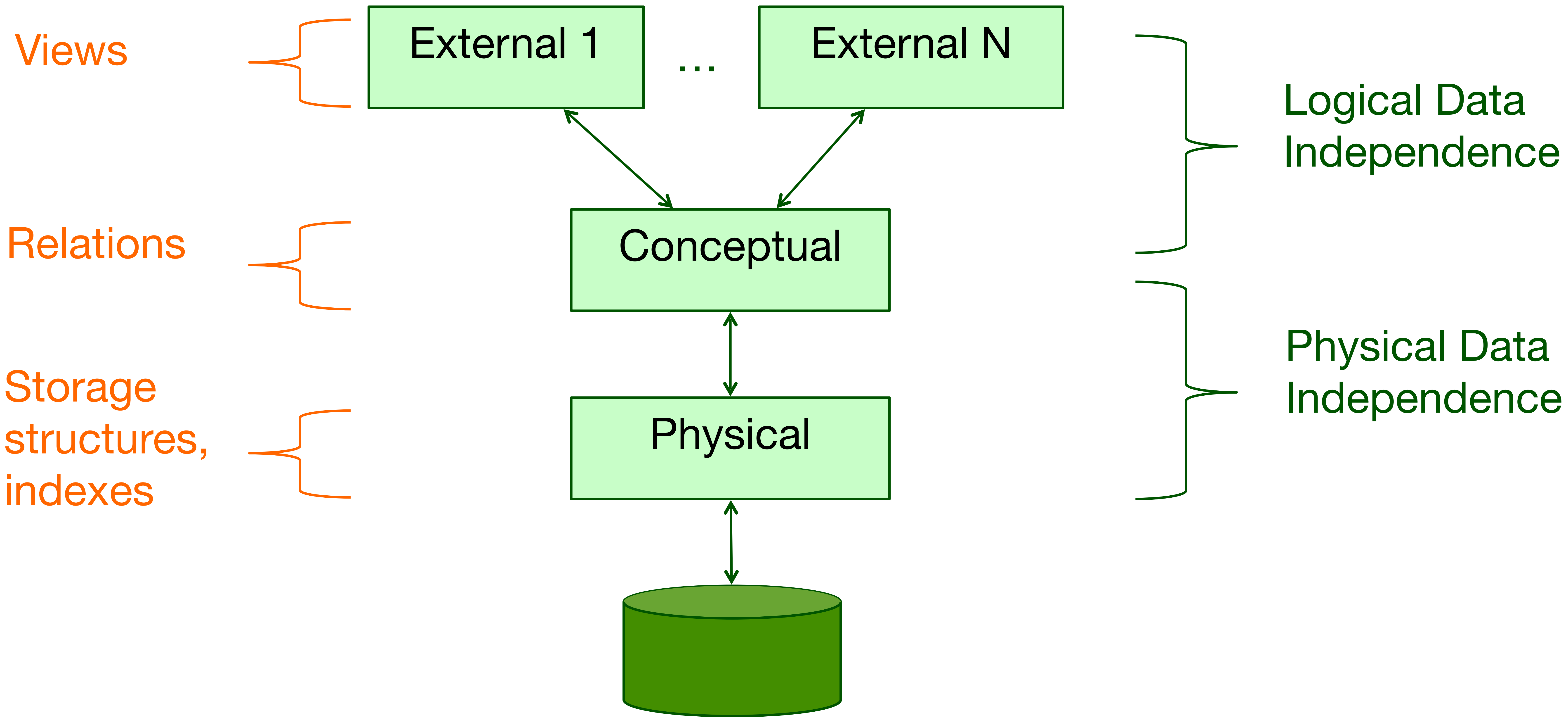
```
CREATE VIEW Red_Green_Sailors (sid, sname) AS
  SELECT DISTINCT S.sid, S.sname
  FROM   Sailors S, Boats B, Reserves R
  WHERE  S.sid=R.sid AND R.bid=B.bid
    AND  (B.color=‘red’ OR B.color=‘green’)
```

```
DROP VIEW Red_Green_Sailors
```

Remember this picture?



Remember this picture?



Why use views?

- Hide some data from some users
- Make some queries easier / more natural
- Modularity of database access

Real applications tend to use lots and lots (and lots and lots!) of views

Why use views?

- Hide some data from some users
- Make some queries easier / more natural
- Modularity of database access

Real applications tend to use lots and lots (and lots and lots!) of views

- Virtual vs. Materialized Views
  + Improve query performance
  + Transparency to applications
  – Materialized view could be LARGE!
  – View maintenance under updates

Source: Widom (partial)

Materialized View Example

```
CREATE MATERIALIZED VIEW CA-CS AS
SELECT C.cName, S.sName
FROM College C, Student S, Apply A
WHERE C.cName = A.cName AND S.sID = A.sID
AND C.state = ‘CA’ AND A.major = ‘CS’
```

- Can use CA-CS as if it’s a table (it is!)
  - DBMS will store a copy of result of view query

Source: Widom (partial)

Materialized View Example

```
CREATE MATERIALIZED VIEW CA-CS AS
SELECT C.cName, S.sName
FROM College C, Student S, Apply A
WHERE C.cName = A.cName AND S.sID = A.sID
AND C.state = ‘CA’ AND A.major = ‘CS’
```

- Can use CA-CS as if it’s a table (it is!)
  - DBMS will store a copy of result of view query
- But: Modifications to base data invalidate view
  - DBMS can be given a policy to refresh view, e.g., daily
  - Incremental maintenance possible in some cases

Source: Widom (partial)

Materialized View Example

```
CREATE MATERIALIZED VIEW CA-CS AS
SELECT C.cName, S.sName
FROM College C, Student S, Apply A
WHERE C.cName = A.cName AND S.sID = A.sID
AND C.state = ‘CA’ AND A.major = ‘CS’
```

- Can use CA-CS as if it’s a table (it is!)
  - DBMS will store a copy of result of view query
- But: Modifications to base data invalidate view
  - DBMS can be given a policy to refresh view, e.g., daily
  - Incremental maintenance possible in some cases
- Also: If you can update the view, base tables must stay in sync
  - DBMS must propagate updates through view (not always possible…)

Source: Widom (partial)

View Maintenance

- Two steps:
  - Propagate: Compute changes to view when data changes.
  - Refresh: Apply changes to the materialized view table.

- Maintenance policy: Controls when we do refresh.
  - Immediate: As part of the operation that modifies the underlying data tables.
    - + Materialized view is always consistent
    - – Updates are slowed down
  - Deferred: Some time later, as a separate operation.
    - – View becomes inconsistent
    - + Can scale to maintain many views without slowing down updates

View Maintenance

- Two steps:
  - Propagate: Compute changes to view when data changes.
  - Refresh: Apply changes to the materialized view table.

- Maintenance policy: Controls when we do refresh.
  - Immediate: As part of the operation that modifies the underlying data tables.
    - \+ Materialized view is always consistent
    - – Updates are slowed down
  - Deferred: Some time later, as a separate operation.
    - – View becomes inconsistent
    - \+ Can scale to maintain many views without slowing down updates
    - Lazy: Delay refresh until next query on view; then refresh before answering the query.
    - Periodic (Snapshot): Refresh periodically. Queries possibly answered using outdated version of view tuples. Widely used, especially for asynchronous replication in distributed databases, and for warehouse applications.
    - Event-based: E.g., refresh after a fixed number of updates to underlying data tables.

Source: Ramakrishnan & Gehrke

Triggers

- "Event-Condition-Action Rules"
  - When event occurs, check condition; if true, do action

- Why would we need triggers?

  1) Move monitoring logic from application into DBMS

  2) Enforce constraints
     - Beyond what constraint system supports
     - Automatic constraint "repair"

- Implementations vary significantly
  - Different DBMS may support different kinds of events and actions
  - Different DBMS may allow triggers to be written in different languages (e.g., through stored procedures)

Source: Widom (partial)

Triggers in SQL

```
CREATE TRIGGER name
BEFORE|AFTER|INSTEAD OF events

[ referencing-variables ]

[ FOR EACH ROW ]

WHEN ( condition )

action
```

A Simple Example of a Trigger

Trigger name in database schema

Event: After update of attribute

Access to tuple values

Row level trigger

Condition

Action
Prevent lower values
Reset to old value

```
CREATE TRIGGER AUR_NetWorthTrigger
AFTER UPDATE OF netWorth ON MovieExec
REFERENCING
    OLD ROW AS OldTuple
    NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
       UPDATE MovieExec
       SET netWorth = OldTuple.netWorth
       WHERE cert# = NewTuple.cert#;
```

NOTE: Not PostgreSQL syntax

- **Granularity**
  - Row-level: Event = Change of a single row (a single UPDATE statement might result in multiple events)
  - Statement-level: Event = Statement (a single UPDATE statement that changes multiple rows is a single event).

Source: Lassen (partial)

Aggregate Maintenance with Triggers

```
Orders(order_num, item_num, quantity, store_id, vendor_id)
Items(item_num, price)
VendorOutstanding(vendor_id, amount)
```

- Insertions

```
INSERT INTO orders VALUES (1000350,7825,100,'xxxxxx6944','vendor4');
```

- Queries (first without, then with redundant tables)

```
SELECT vendor_id, SUM(quantity * price)
FROM orders NATURAL JOIN items
GROUP BY vendor_id;
```

vs.

```
SELECT * FROM vendorOutstanding;
```

Source: Shasha and Bonnet (partial)

Aggregate Maintenance with Triggers (contd.)

```
CREATE TRIGGER
trUpsertVendorOutstanding
AFTER INSERT ON orders
FOR EACH ROW
EXECUTE PROCEDURE
pyUpsertVendorOutstanding();
```

```
CREATE OR REPLACE FUNCTION pyUpsertVendorOutstanding()
RETURNS trigger
AS $$
        # new values inserted
        new_quantity = TD["new"]["quantity"]
        new_item_num = TD["new"]["item_num"]
        new_vendor_id = TD["new"]["vendor_id"]

        # Python code that accesses DB follows
        …
$$ LANGUAGE plpython3u;
```

See script trigger-example.sql in Absalon

Aggregate Maintenance with Triggers (contd.)

```
CREATE OR REPLACE FUNCTION pyUpsertVendorOutstanding()
RETURNS trigger
AS $$
        # new values inserted
        new_quantity = TD["new"]["quantity"]
        new_item_num = TD["new"]["item_num"]
        new_vendor_id = TD["new"]["vendor_id"]

        # prepare upsert SQL statement
        upsert_stmt = plpy.prepare(
                ("INSERT INTO vendor_outstanding "
                 "as v(vendor_id, amount) VALUES"
                 "($1, $2 * (SELECT items.price "
                 "       FROM items WHERE items.item_num = $3)) "
                 "ON CONFLICT (vendor_id) DO "
                 "UPDATE SET amount = v.amount + EXCLUDED.amount"),
                 ["text", "int", "int"])

        # execute upsert on vendor_outstanding
        plpy.execute(upsert_stmt, [new_vendor_id, new_quantity, new_item_num])
$$ LANGUAGE plpython3u;
```

Other Uses of Triggers

- Maintaining an audit trail or history of modifications
  - In an auction, record audit trail of bids per user
- Automatically populating attributes
  - Whenever a bid is updated by a user, record the current time (ignoring any other time value given by application)
- Making updates conditional
  - Disallow users to revise bids down, only allow bids to be revised to be higher than current value

Source: Lassen (partial)

Tricky Issues in Triggers

- Row-level vs. Statement-level
  - **New/Old Row** and **New/Old Table**
  - **Before**, **Instead** Of, **After**
- Multiple triggers activated at same time
- Trigger actions activating other triggers (chaining)
  - Also self-triggering, cycles, nested invocations
- Conditions in **When** vs. as part of action

Source: Widom (partial)

DBMS vs. Multiple Users

- Support for concurrent access necessary
  - Lower response time, users do not have to queue behind large jobs
- Make maximal use of CPU / disk resources for performance
  - With single user at a time, access to disk wastes huge amount of CPU cycles
  - Processing user requests concurrently increases throughput

Source: Ramakrishnan & Gehrke

Transactions

- Reliable unit of work against database
- ACID Properties
  - Atomicity: transactions are all-or-nothing
  - Consistency: transaction takes database from one consistent state to another
  - Isolation: transaction executes as if it was the only one in the system
  - Durability: once transaction is “committed”, results are persistent in the database

Source: Ramakrishnan & Gehrke (partial)

Transactions in SQL

Transaction T1: TRANSFER

```
BEGIN;
  UPDATE accounts
  SET balance = balance - 10
  WHERE name = 'Dmitriy';
  --
  UPDATE accounts
  SET balance = balance + 10
  WHERE name = 'Magnus';
COMMIT;
```

Transaction T2: INTEREST

```
BEGIN;
  UPDATE accounts
  SET balance = balance * 1.01;
COMMIT;
```

- BEGIN and COMMIT keywords delimit transaction
- COMMIT confirms that work should be made durable in the database

Transactions in SQL

Transaction T1: TRANSFER

```
BEGIN;
  UPDATE accounts
  SET balance = balance - 10
  WHERE name = 'Dmitriy';
  --
  UPDATE accounts
  SET balance = balance + 10
  WHERE name = 'Magnus';
ROLLBACK;
```

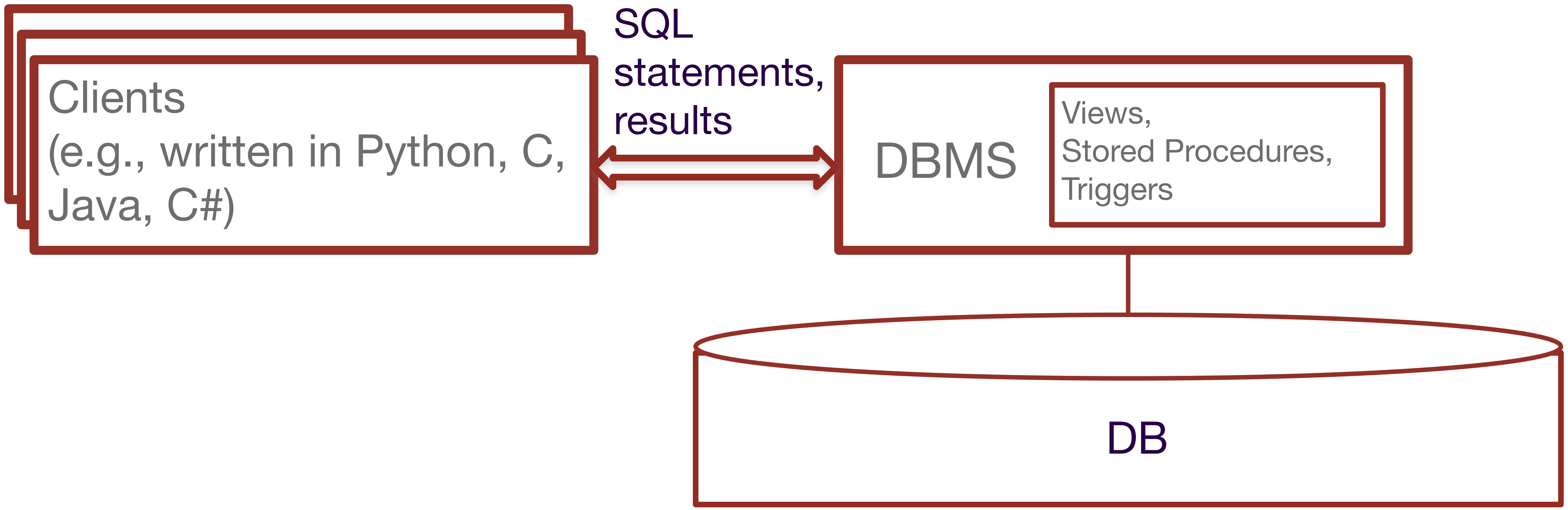
Transaction T2: INTEREST

```
BEGIN;
  UPDATE accounts
  SET balance = balance * 1.01;
ROLLBACK;
```

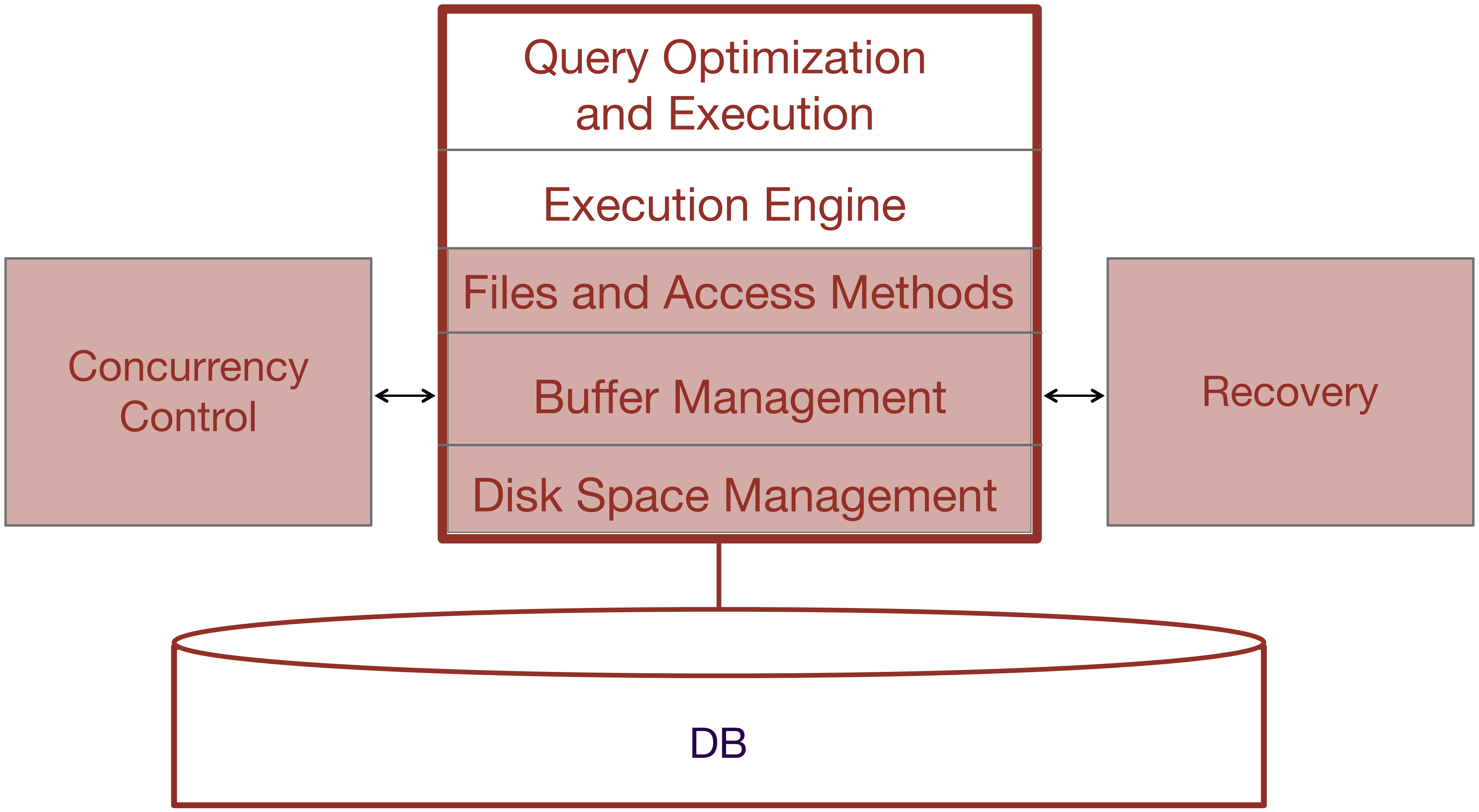
- ROLLBACK implies that effects of transactions should be undone
- In this example, database back to original state

Where does the code run?

- Previous example could be input directly in psql
- Transaction often run from client program through a **driver** or **database adapter**
  - e.g., Psycopg (http://initd.org/psycopg/)

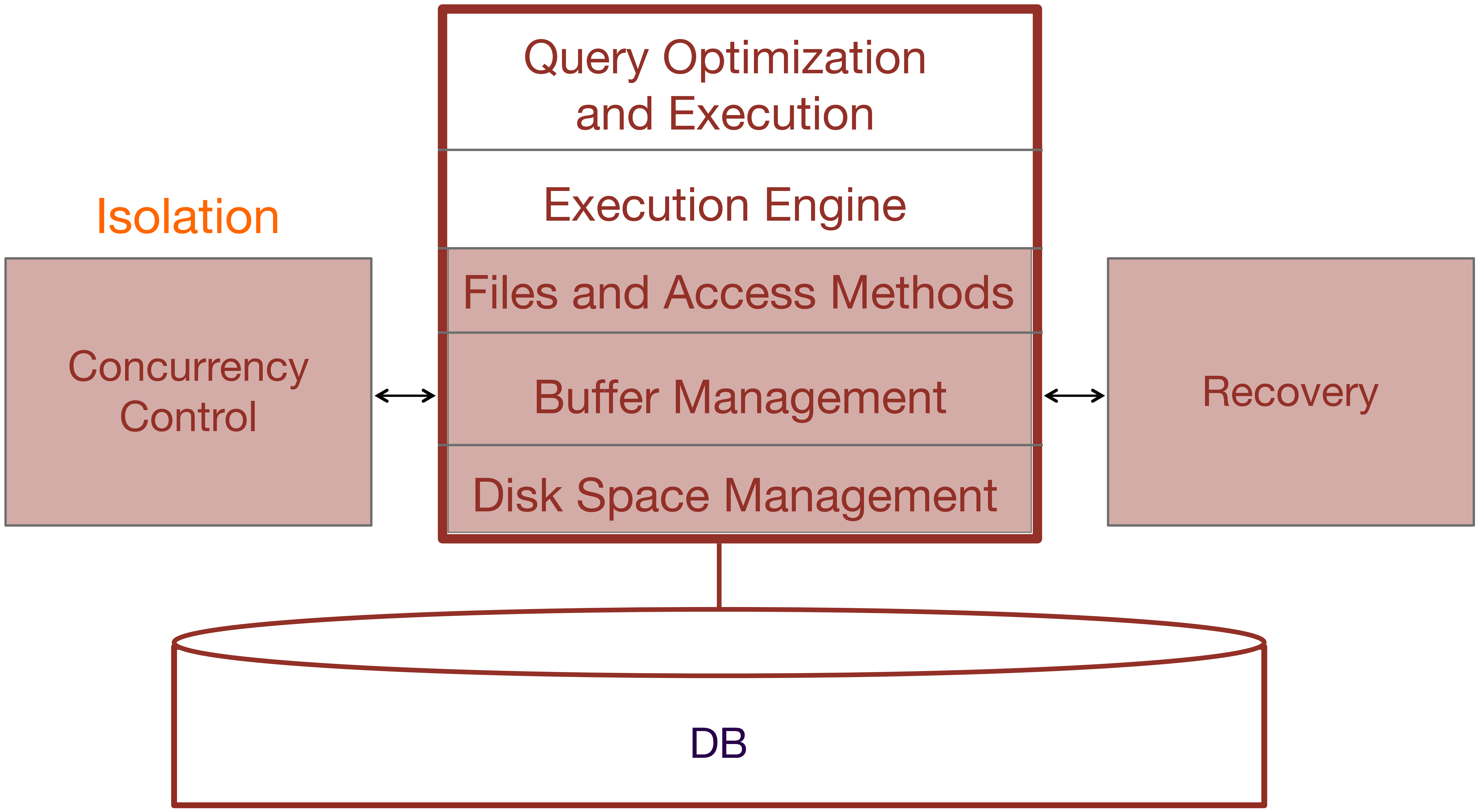


Classic Architecture of a DBMS



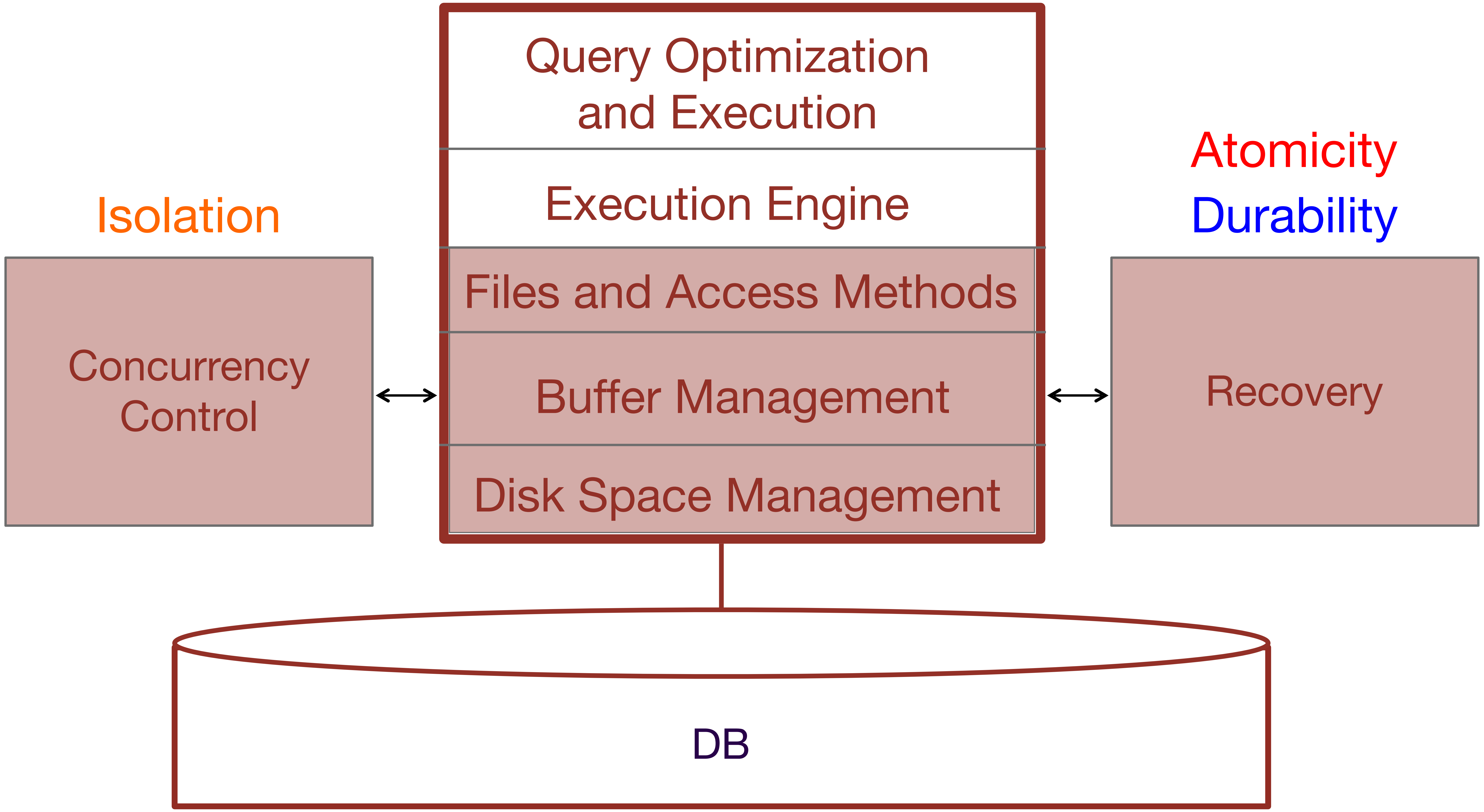
Source: Ramakrishnan & Gehrke (partial)

Classic Architecture of a DBMS



Source: Ramakrishnan & Gehrke (partial)

Classic Architecture of a DBMS



Source: Ramakrishnan & Gehrke (partial)

Discussion: Transactions in SQL

- Take script txn_example.sql from Absalon and open two psql terminals

- Run first this code fragment:

```
BEGIN;
  UPDATE accounts
  SET balance = balance * 1.01;
-- do not commit yet
```

- In a separate terminal, then run:

```
BEGIN;
  UPDATE accounts
  SET balance = balance + 10
  WHERE name = ‘Dmitriy';
…
```

- Discussion: If you run the scripts above in PostgreSQL via two psql terminals, what are the outcomes? Why?
- Can you now commit or rollback and see what happens to the balances?

Questions so far?

A Query

```
SELECT DISTINCT X.location, X.date, count(Y.date)
FROM covid X JOIN covid Y
ON X.location = Y.location AND X.new_cases > Y.new_cases
GROUP BY X.location, X.date
```

A Query

```
SELECT DISTINCT X.location, X.date, count(Y.date)
FROM covid X JOIN covid Y
ON X.location = Y.location AND X.new_cases > Y.new_cases
GROUP BY X.location, X.date
```

Wait for 25 seconds!

A Query

```
SELECT DISTINCT X.location, X.date, count(Y.date)
FROM covid X JOIN covid Y
ON X.location = Y.location AND X.new_cases > Y.new_cases
GROUP BY X.location, X.date
```

Wait for 25 seconds!

```
 …
 Zimbabwe                            | 2021-05-16 |    99
 Zimbabwe                            | 2021-05-17 |   125
 Zimbabwe                            | 2021-05-18 |   191
 Zimbabwe                            | 2021-05-19 |   160
 Zimbabwe                            | 2021-05-20 |   191
 Zimbabwe                            | 2021-05-21 |   222
 Zimbabwe                            | 2021-05-22 |   147
(72976 rows)
```

A(nother?) Query

```
CREATE INDEX covidi ON covid(location, new_cases, date);

SELECT DISTINCT X.location, X.date, count(Y.date)
FROM covid X JOIN covid Y
ON X.location = Y.location AND X.new_cases > Y.new_cases
GROUP BY X.location, X.date
```

A(nother?) Query

```
CREATE INDEX covidi ON covid(location, new_cases, date);

SELECT DISTINCT X.location, X.date, count(Y.date)
FROM covid X JOIN covid Y
ON X.location = Y.location AND X.new_cases > Y.new_cases
GROUP BY X.location, X.date
```

```
 …
 Zimbabwe                                | 2021-05-16 |    99
 Zimbabwe                                | 2021-05-17 |   125
 Zimbabwe                                | 2021-05-18 |   191
 Zimbabwe                                | 2021-05-19 |   160
 Zimbabwe                                | 2021-05-20 |   191
 Zimbabwe                                | 2021-05-21 |   222
 Zimbabwe                                | 2021-05-22 |   147
(72976 rows)
```

A(nother?) Query

```
CREATE INDEX covidi ON covid(location, new_cases, date);

SELECT DISTINCT X.location, X.date, count(Y.date)
FROM covid X JOIN covid Y
ON X.location = Y.location AND X.new_cases > Y.new_cases
GROUP BY X.location, X.date
```

Wait for 7.5 seconds!

```
 …
 Zimbabwe                              | 2021-05-16 |    99
 Zimbabwe                              | 2021-05-17 |   125
 Zimbabwe                              | 2021-05-18 |   191
 Zimbabwe                              | 2021-05-19 |   160
 Zimbabwe                              | 2021-05-20 |   191
 Zimbabwe                              | 2021-05-21 |   222
 Zimbabwe                              | 2021-05-22 |   147
(72976 rows)
```

A(nother?) Query

```
CREATE INDEX covidi ON covid(location, new_cases, date);

SELECT DISTINCT X.location, X.date, count(Y.date)
FROM covid X JOIN covid Y
ON X.location = Y.location AND X.new_cases > Y.new_cases
GROUP BY X.location, X.date
```

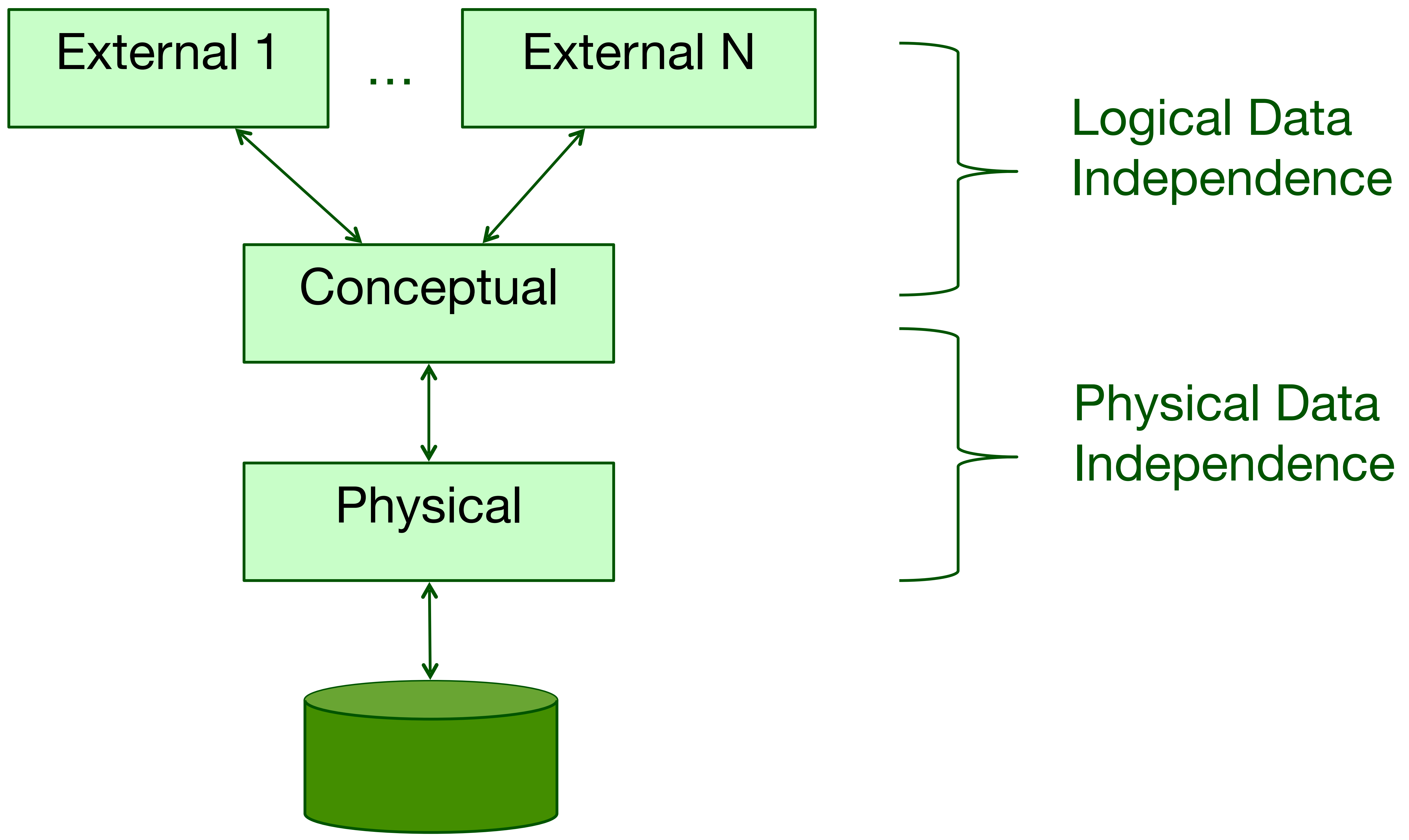
0.2 s

7.3 s

Wait for 7.5 seconds!

```
 …
 Zimbabwe                         | 2021-05-16 |    99
 Zimbabwe                         | 2021-05-17 |   125
 Zimbabwe                         | 2021-05-18 |   191
 Zimbabwe                         | 2021-05-19 |   160
 Zimbabwe                         | 2021-05-20 |   191
 Zimbabwe                         | 2021-05-21 |   222
 Zimbabwe                         | 2021-05-22 |   147
(72976 rows)
```

Physical Data Independence



How can we physically represent relations?

Employees

<u>**ssn**</u>	**name**	**lot**
0983763423	John	10
9384392483	Jane	10
3743923483	Jill	20

Departments

<u>**did**</u>	**dname**	**budget**
101	Sales	10K
105	Purchasing	20K
108	Databases	1000K

Works_In

<u>**ssn**</u>	<u>**did**</u>	**since**
0983763423	101	1 Jan 2003
0983763423	108	2 Jan 2003
9384392483	108	1 Jun 2002

Source: Ramakrishnan & Gehrke

Files of Records

- DBMS software operates on records, and files of records.
- File: A collection of pages (or blocks), each containing a collection of records. Must support:
  - insert/delete/modify record
  - read a particular record (specified using record id)
  - scan all records (possibly with some conditions on the records to be retrieved)

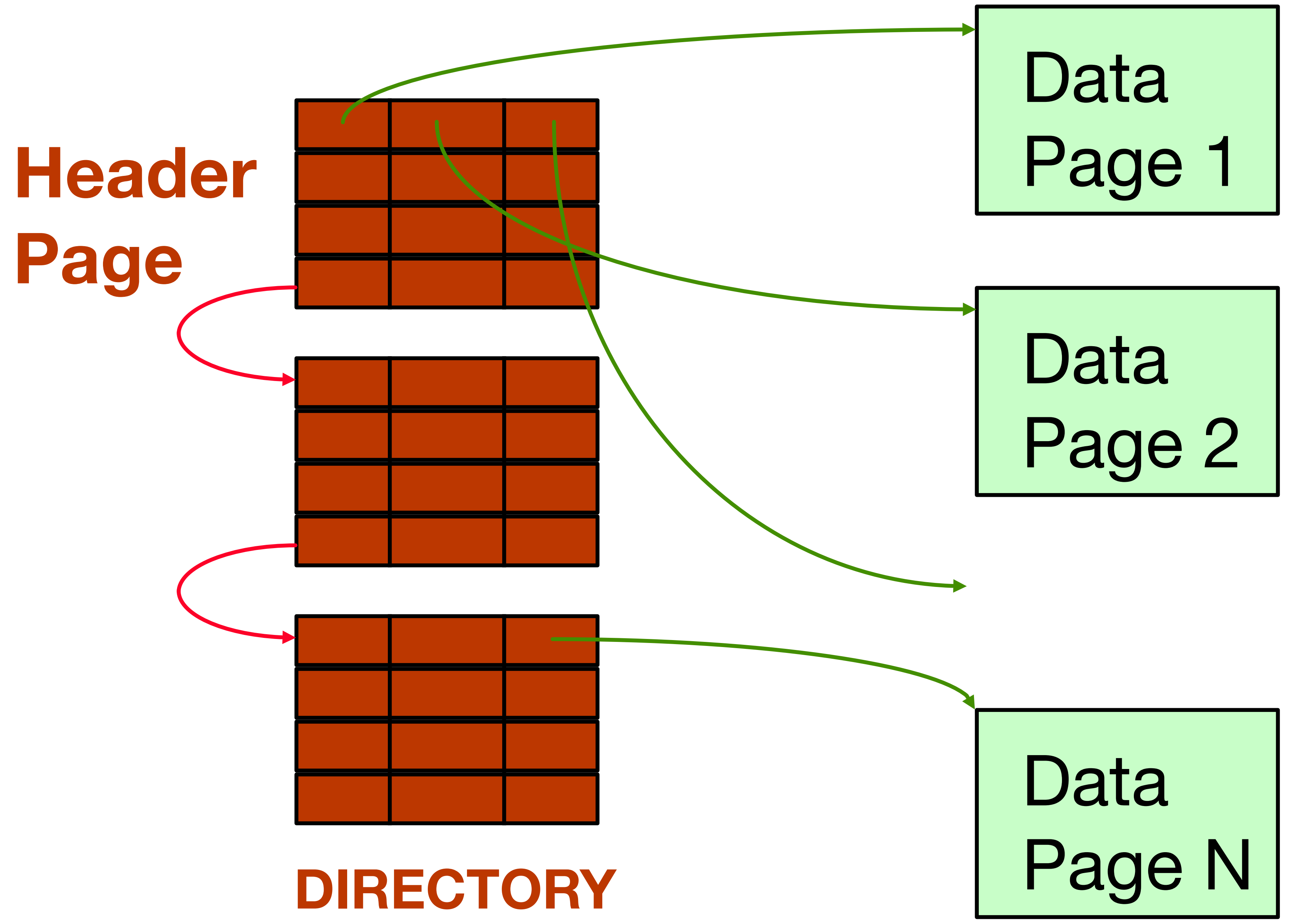
Source: Ramakrishnan & Gehrke (partial)

Unordered (Heap) Files

- Simplest file structure contains records in no particular order.
- As file grows and shrinks, disk pages are allocated and de-allocated.
- To support record level operations, we must:
  - keep track of the **pages** in a file
  - keep track of **free space** on pages
  - keep track of the **records** on a page
- There are many alternatives for keeping track of this.

Source: Ramakrishnan & Gehrke

Heap File Using a Page Directory



- The entry for a page can include the number of free bytes on the page.
- The directory is a collection of pages; e.g., implemented via linked list
  - Much smaller than linked list of all heap file pages!

Source: Ramakrishnan & Gehrke

Alternative Storage Organizations

- Classically, heap file follow a **row-store** organization
  - n-ary Storage Model (NSM)
  - “array of tuples”
- Alternatives:
  - **Column-store**: Decomposition Storage Model (DSM)
    - “tuple of arrays”
  - **Hybrid**: Partition Attributes Across (PAX)
    - “array of tuples of arrays”

RI	Key	fname	lname
1	77	Frank	Meier
2	12	Simon	Schmidt
3	42	Hugo	Müller
4	11	Hans	Meier
5	25	Jens	Dittrich
6	76	Hugo	Schmidt

RID	Key
1	77
2	12
3	42
4	11
5	25
6	76

RID	fname
1	Frank
2	Simon
3	Hugo
4	Hans
5	Jens
6	Hugo

RID	lname
1	Meier
2	Schmidt
3	Müller
4	Meier
5	Dittrich
6	Schmidt

Illustration by: Jens Dittrich, Saarland University

How to access individual records when you do not have the record ID?

```
SELECT *
FROM   departments
WHERE  did >= 100
  AND  did <= 200
```

```
SELECT *
FROM   employees
WHERE  name = ‘John’
```

- **Heap file: scan the whole file!**
- Is there a way to map attribute values directly to records in heap file?

How to access individual records when you do not have the record ID?

```
SELECT *
FROM   departments
WHERE  did >= 100
  AND  did <= 200
```

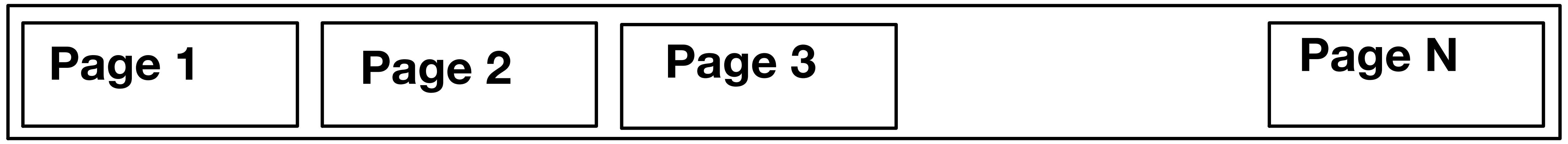
```
SELECT *
FROM   employees
WHERE  name = ‘John’
```

- **Heap file: scan the whole file!**
- Is there a way to map attribute values directly to records in heap file?

Indices are data structures to do so!

Range Searches

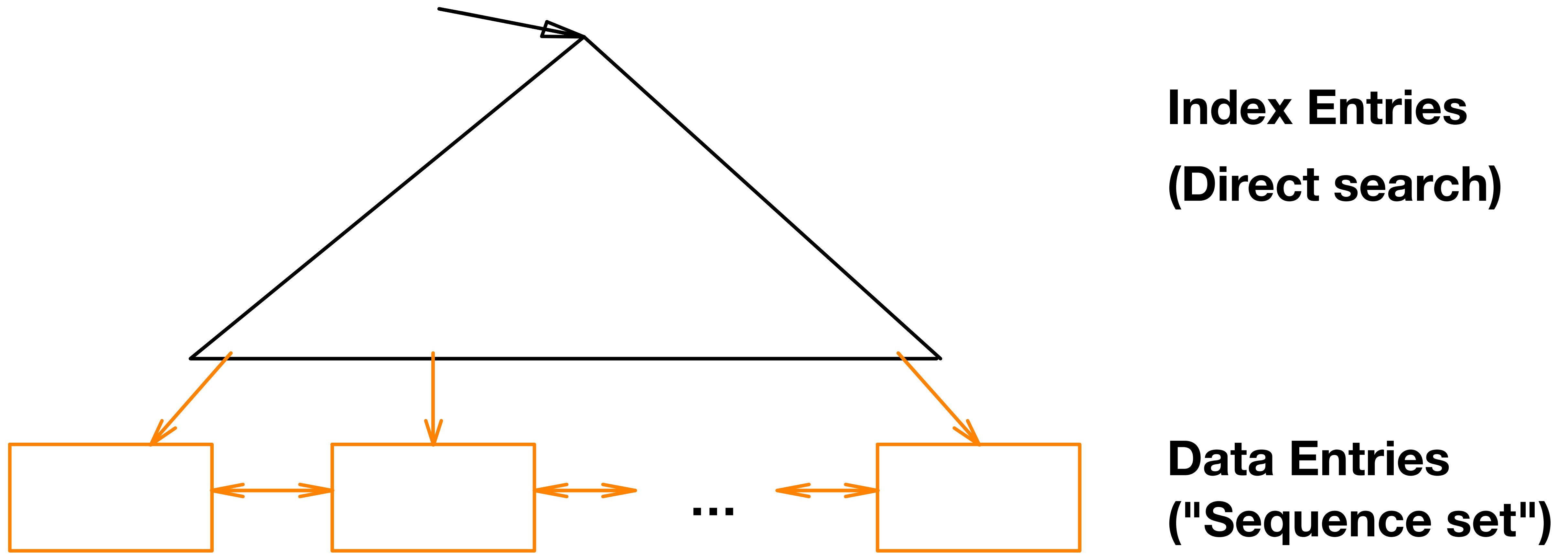
- "Find all students with gpa > 3.0"
  - If data entries are sorted, do binary search to find first such student, then scan to find others.
- Problem?



Data (Entries) File

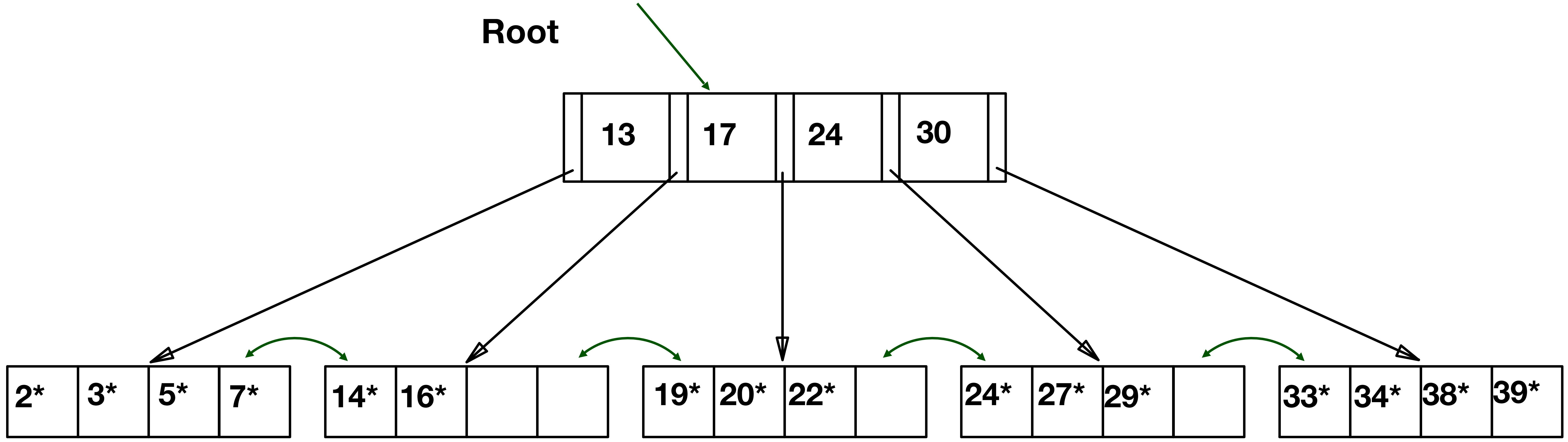
B+ Tree: The Most Widely Used Index

- Insert/delete at $log_{\text{fanout}}(\frac{\text{number of entries}}{\text{leaf capacity}})$ cost; keep tree *height-balanced*.
- Minimum 50% occupancy (except for root). Each node contains $d \leq m \leq 2d$ entries. The parameter d is called the *order* of the tree.
- Supports equality and range-searches efficiently.



Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...



Based on the search for 15, we know it is not in the tree!*

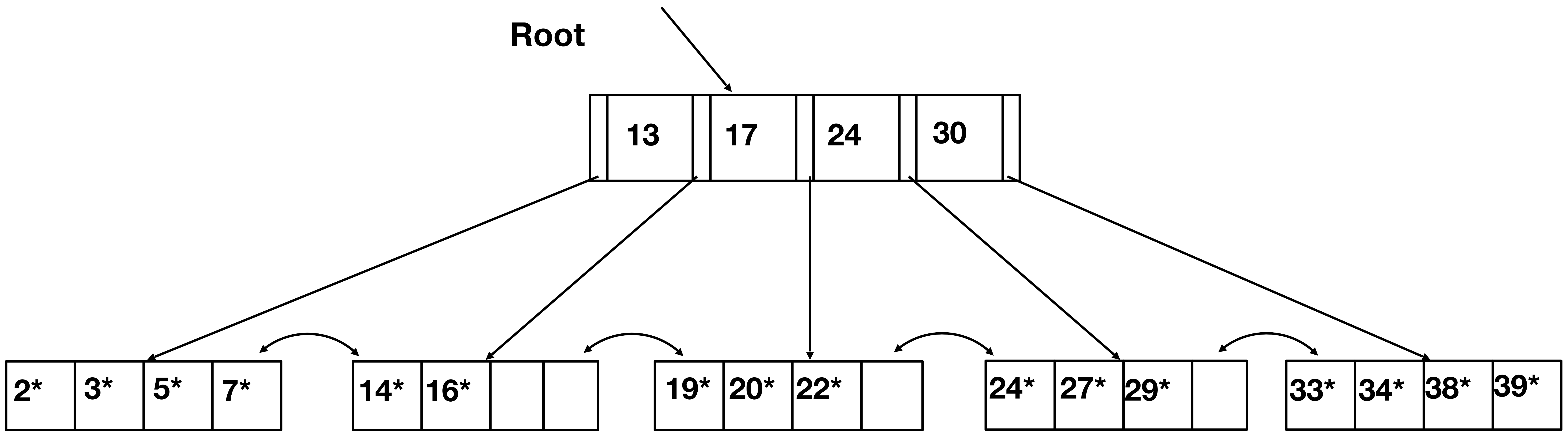
Source: Ramakrishnan & Gehrke

B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,721 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in main memory:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

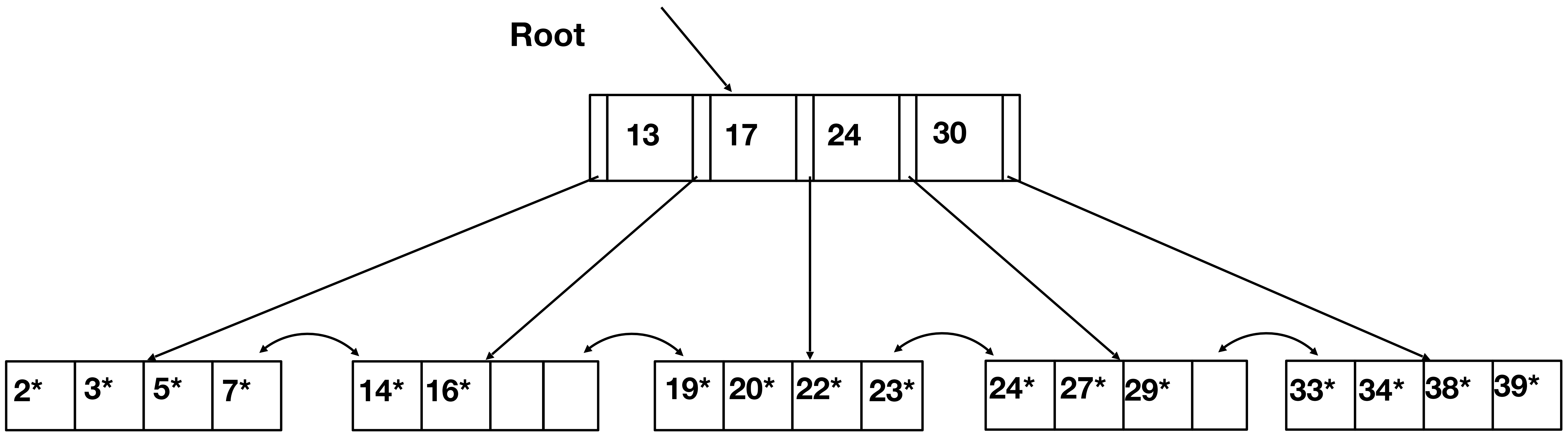
Source: Ramakrishnan & Gehrke

Inserting 23* ...



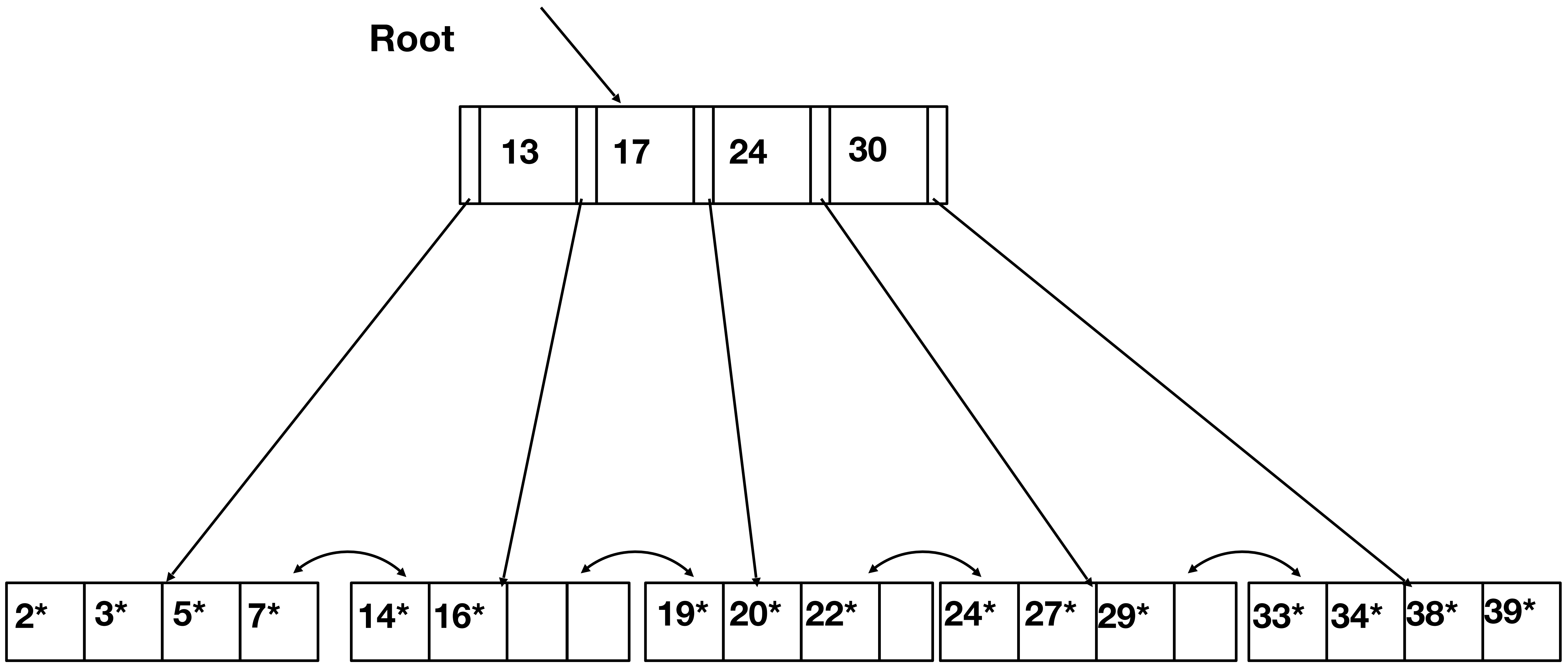
Source: Ramakrishnan & Gehrke

After Inserting 23*



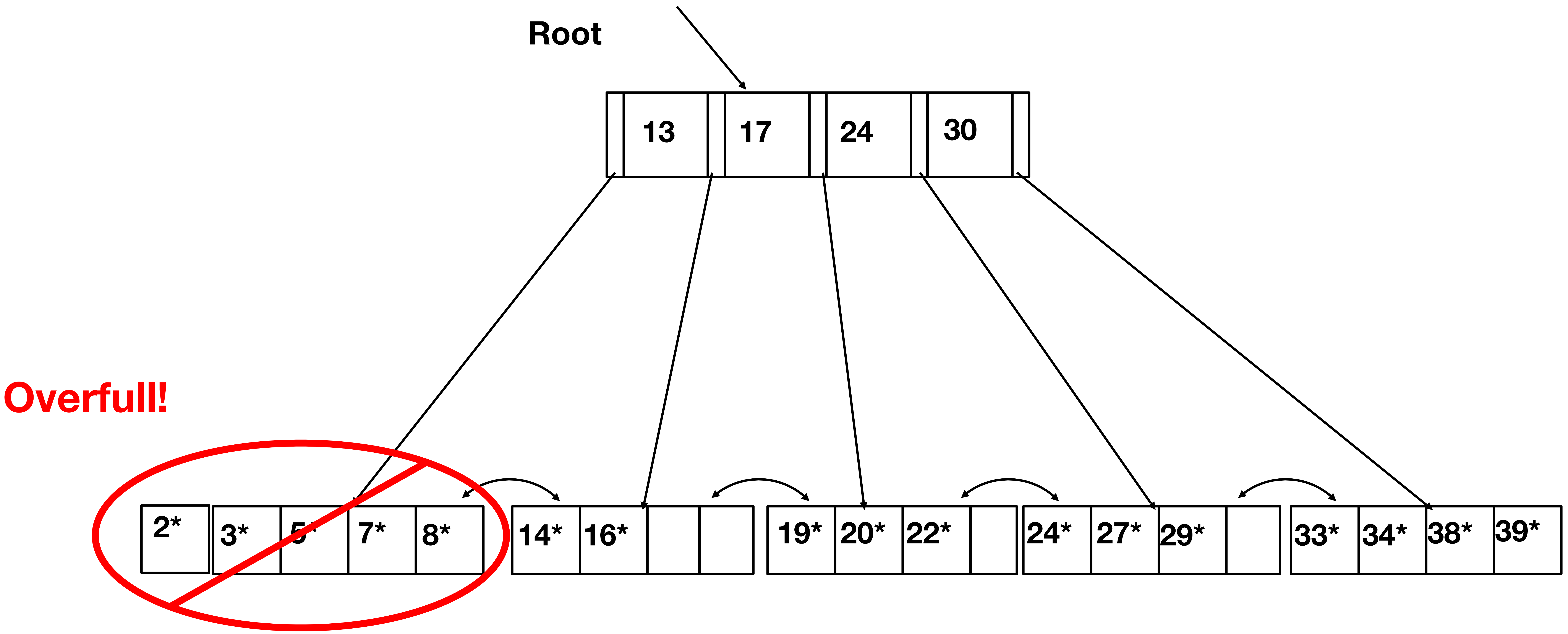
Source: Ramakrishnan & Gehrke

Inserting 8* …



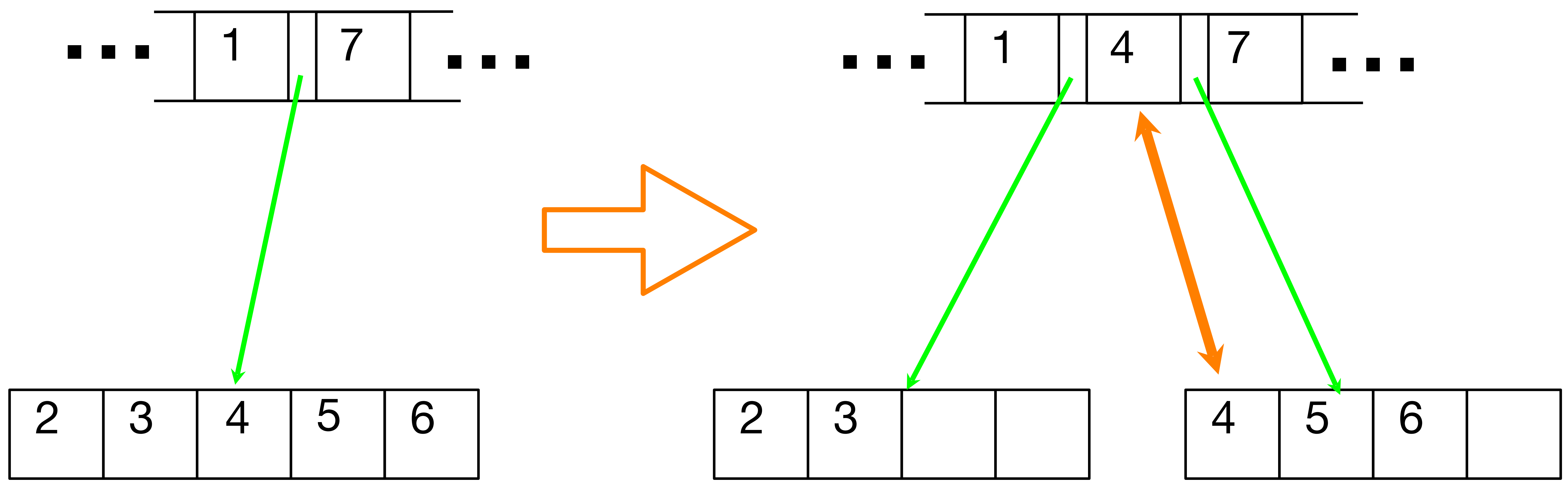
Source: Ramakrishnan & Gehrke

Inserting 8* ...



Source: Ramakrishnan & Gehrke

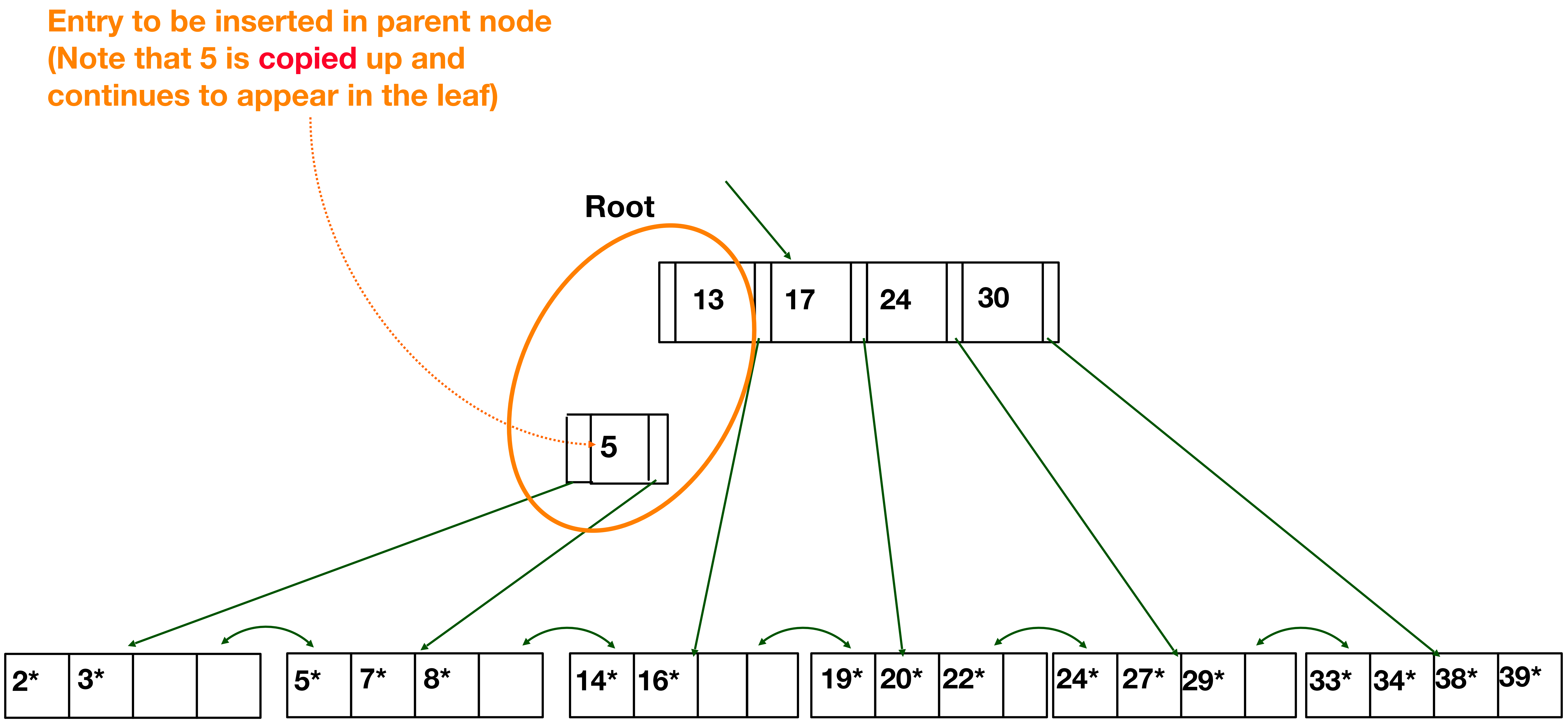
Splitting an Overfull Leaf



- New nodes not underfull: 2d+1 -> d and d + 1
- Single pointer in parent replaced by two pointers with key between
- Key **copied** from right node of new pair.

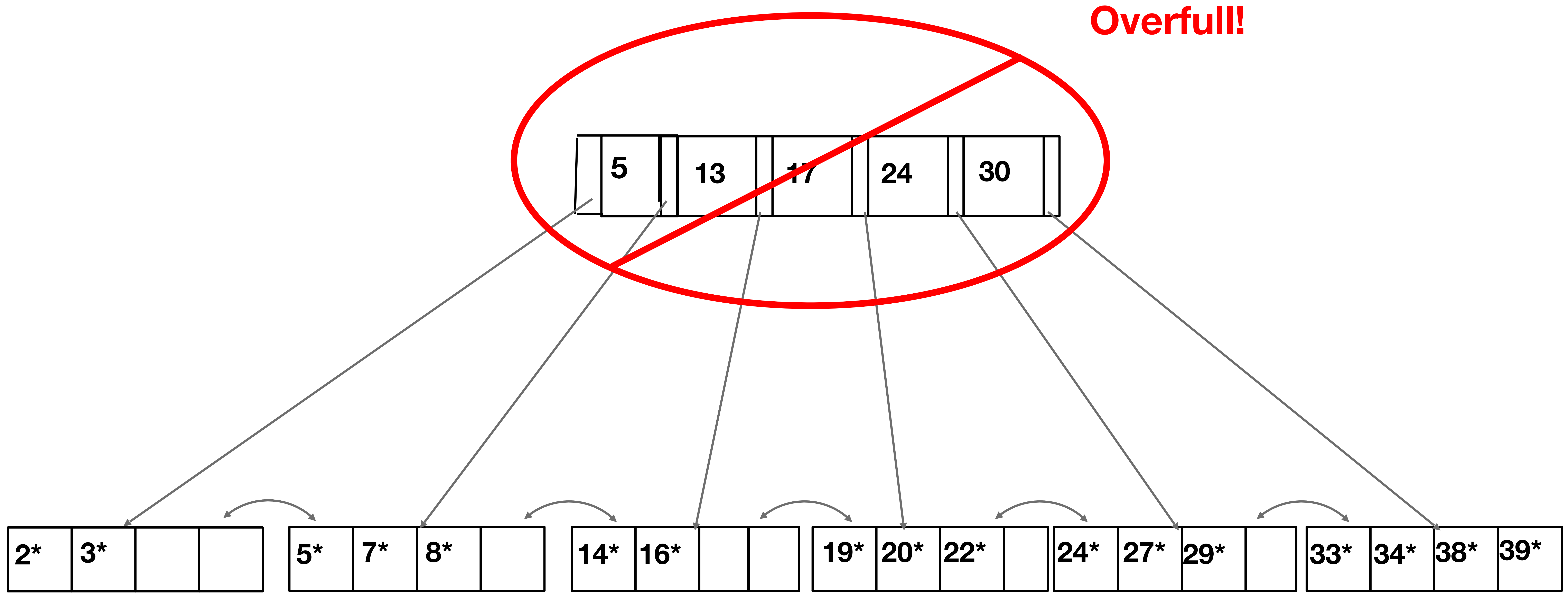
Source: Ramakrishnan & Gehrke

Inserting 8* ...


Entry to be inserted in parent node
(Note that 5 is copied up and
continues to appear in the leaf)
Root
13
17
24
30
5
2*
3*
5*
7*
8*
14*
16*
19*
20*
22*
24*
27*
29*
33*
34*
38*
39*

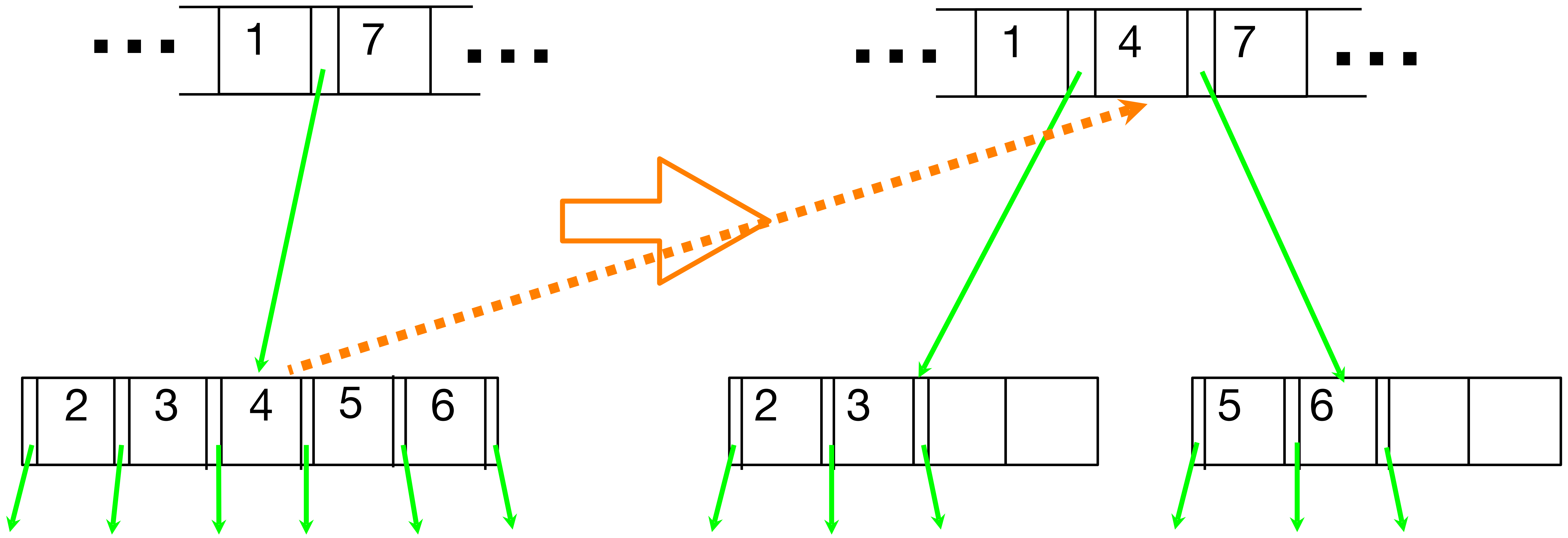
Inserting 8* …

(Note that 5 is **copied** up and continues to appear in the leaf)



Source: Ramakrishnan & Gehrke

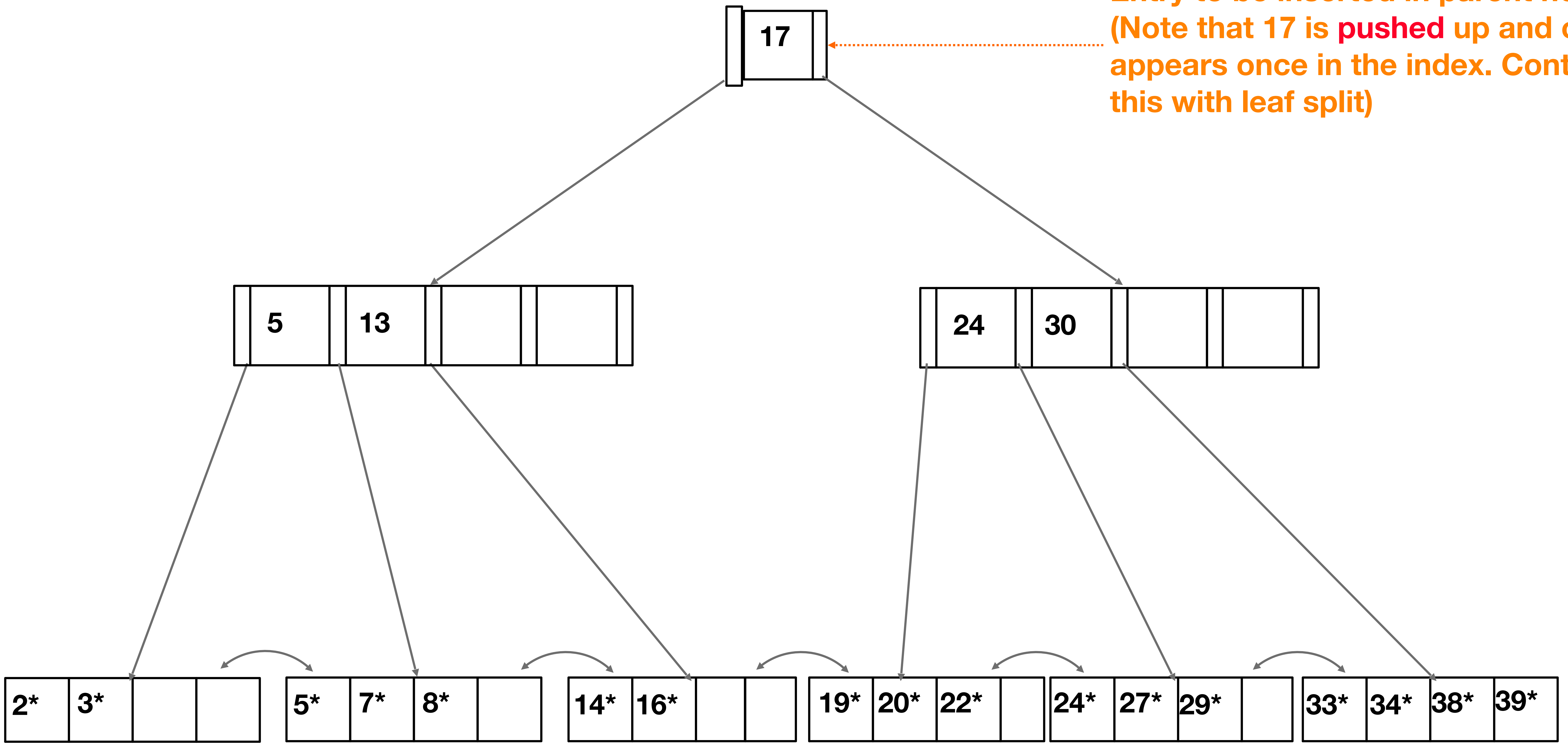
Splitting an Overfull Nonleaf ...



- Like leaf split except
- 2d+1 -> d and d (and 1)
- Key **moved** from right node of new pair, not copied.

Source: Ramakrishnan & Gehrke

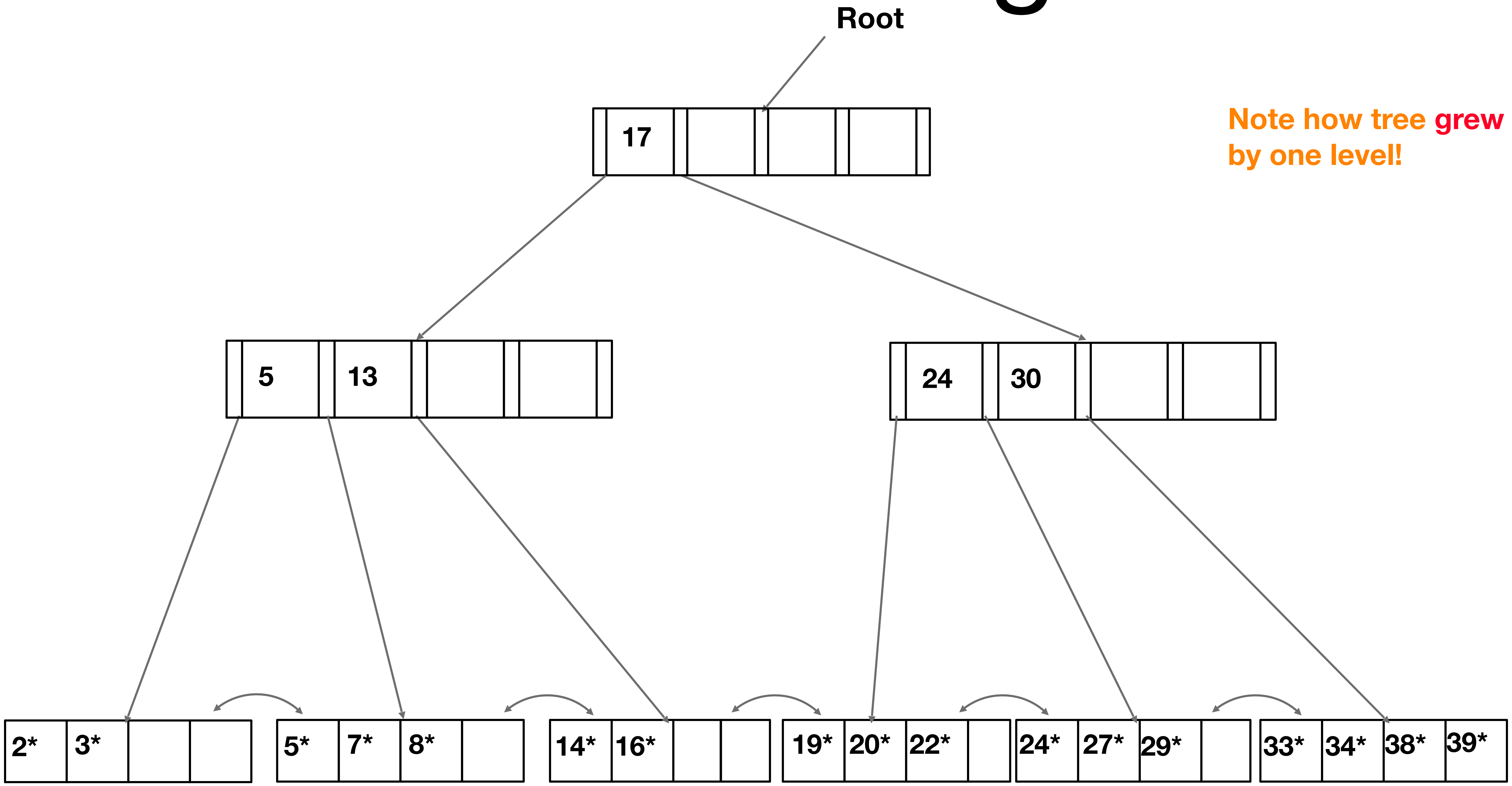
Inserting 8* …



Entry to be inserted in parent node (Note that 17 is **pushed** up and only appears once in the index. Contrast this with leaf split)

Source: Ramakrishnan & Gehrke

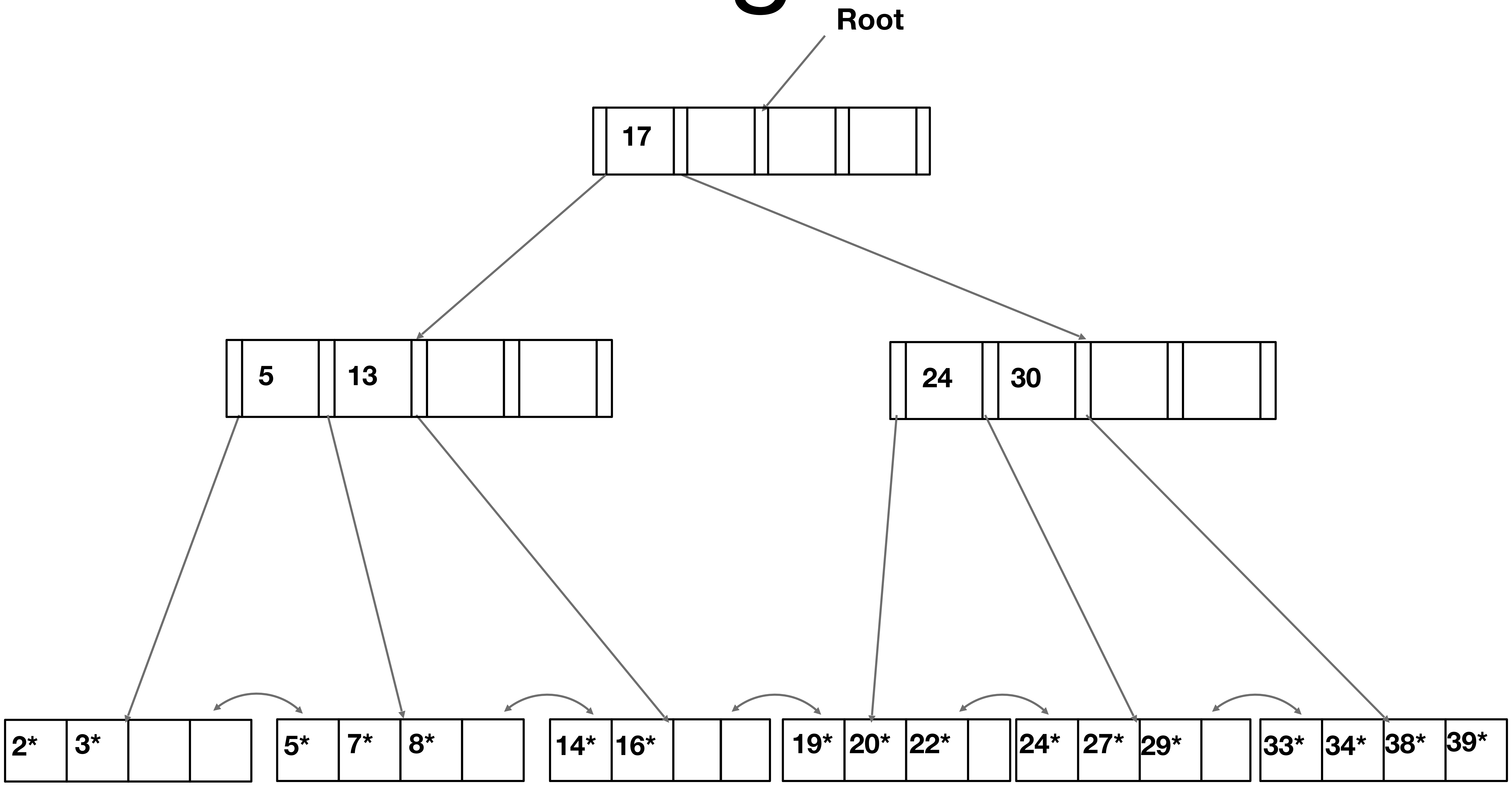
After Inserting 8*



Note how tree **grew** by one level!

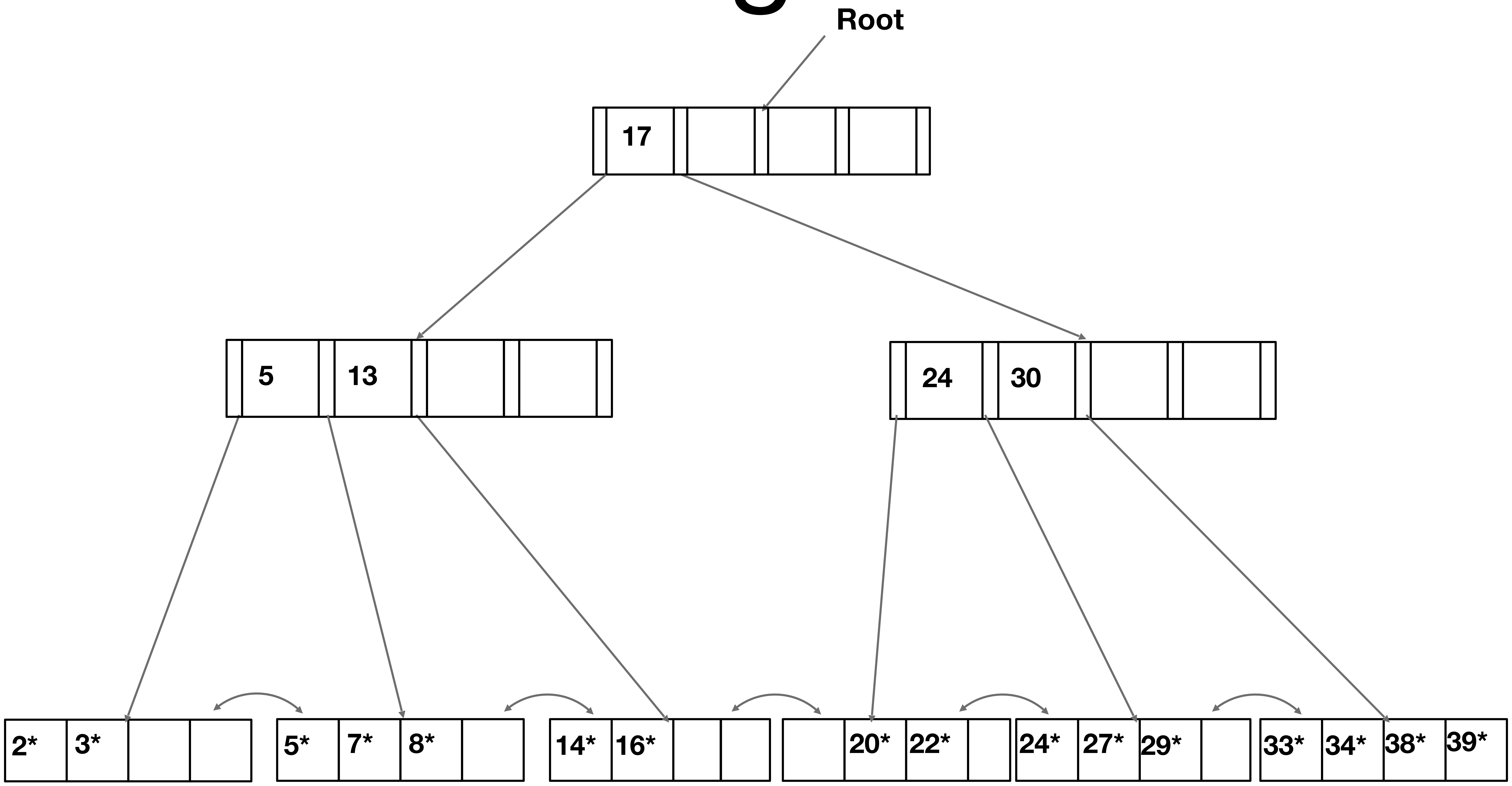
Source: Ramakrishnan & Gehrke

Deleting 19* …

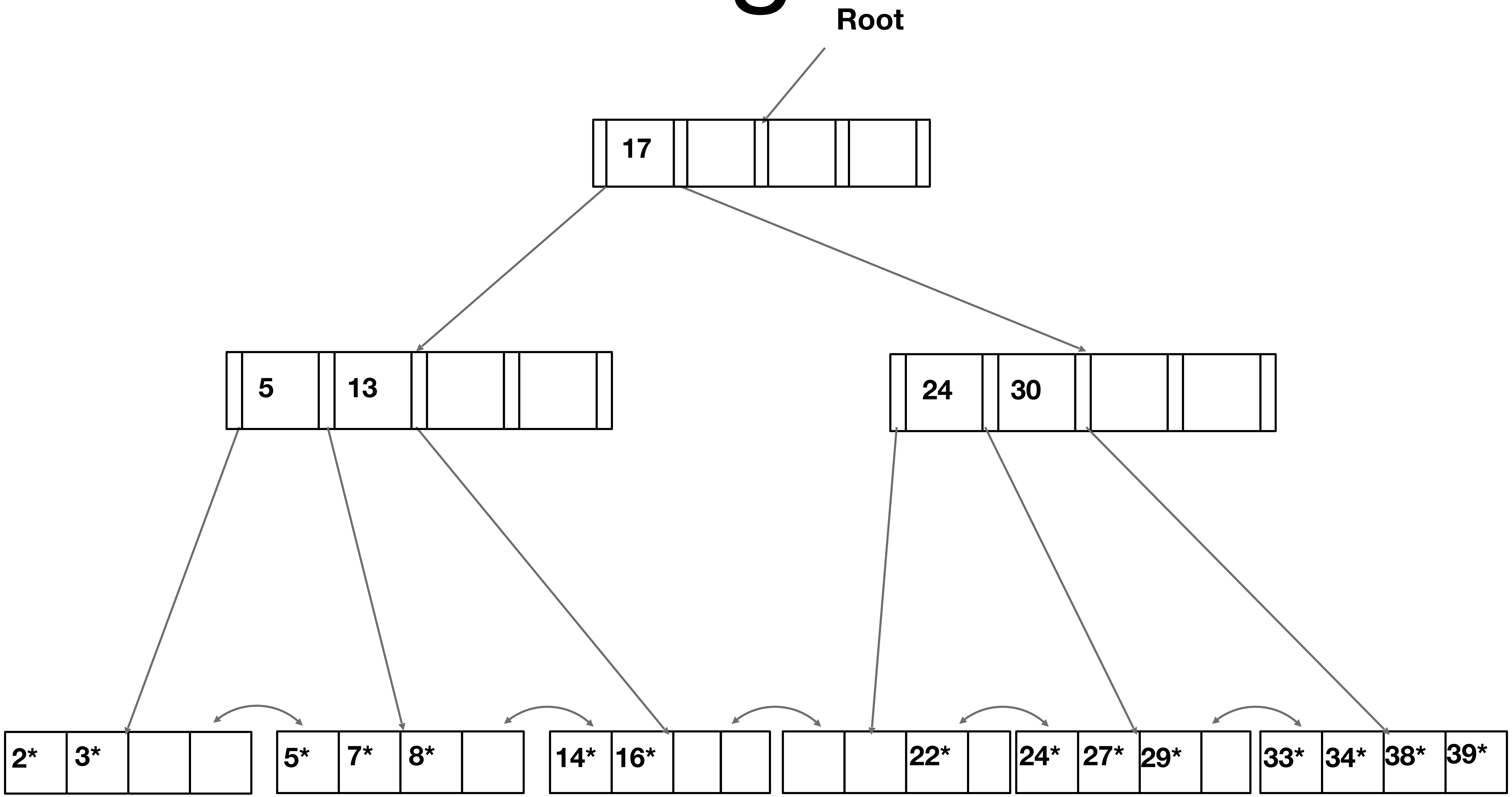


Source: Ramakrishnan & Gehrke

Deleting 20* …

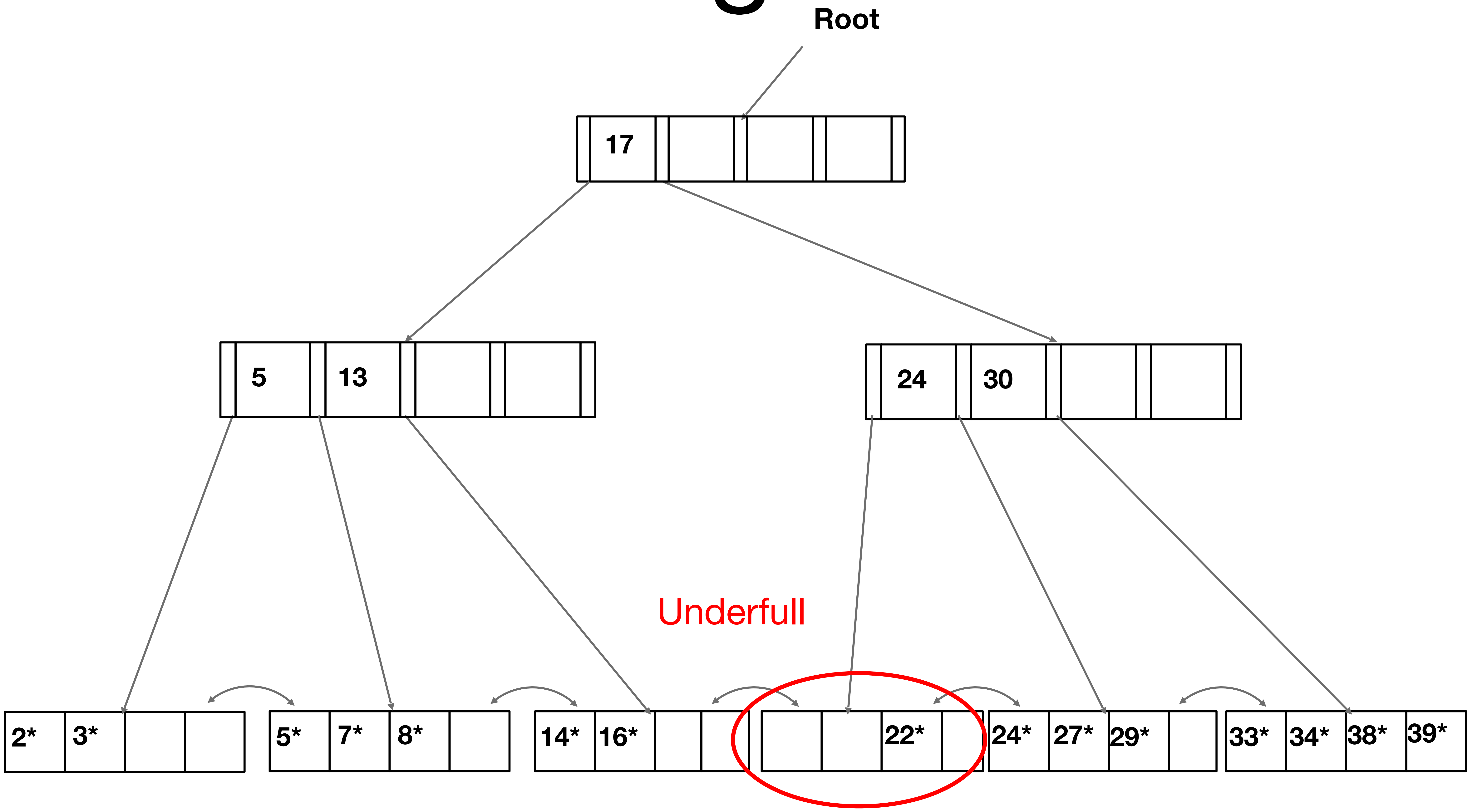


Deleting 20* …

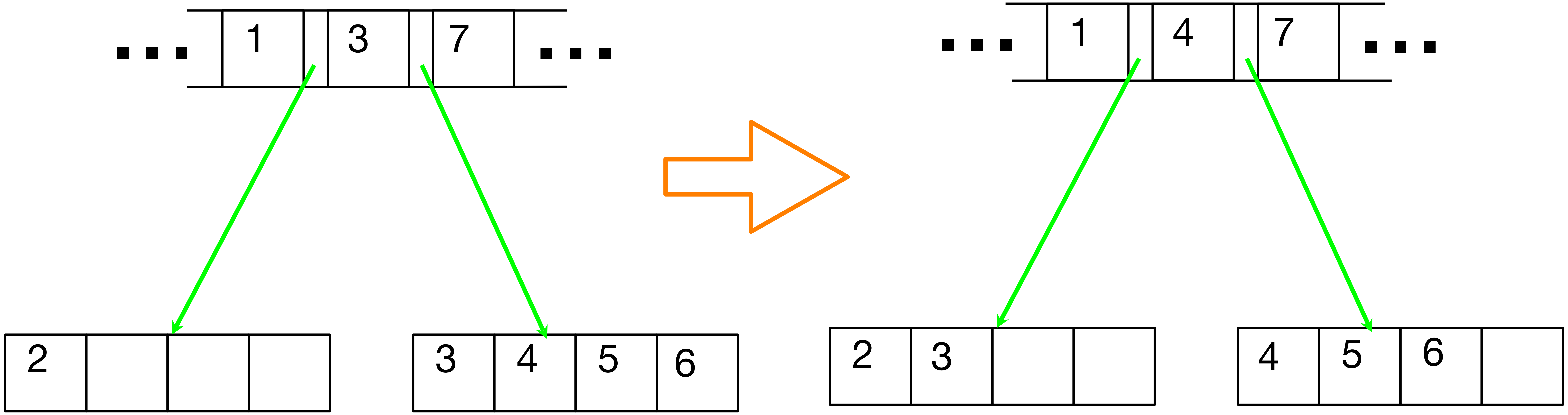


Source: Ramakrishnan & Gehrke

Deleting 20* …



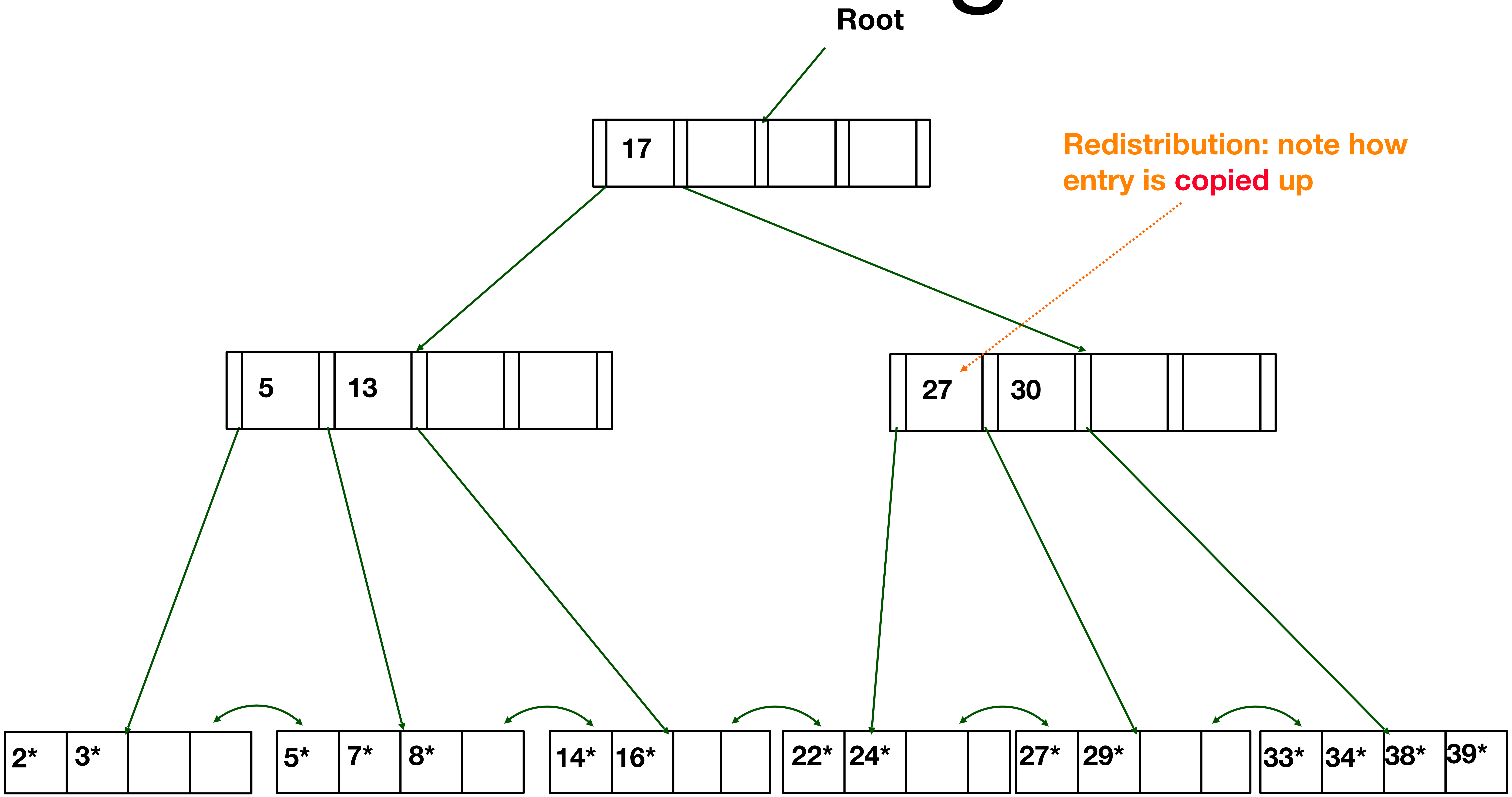
Redistributing Underfull Leaves ...



- Key in parent is modified
- Degree of parent unchanged
- Note there are 3 disk writes

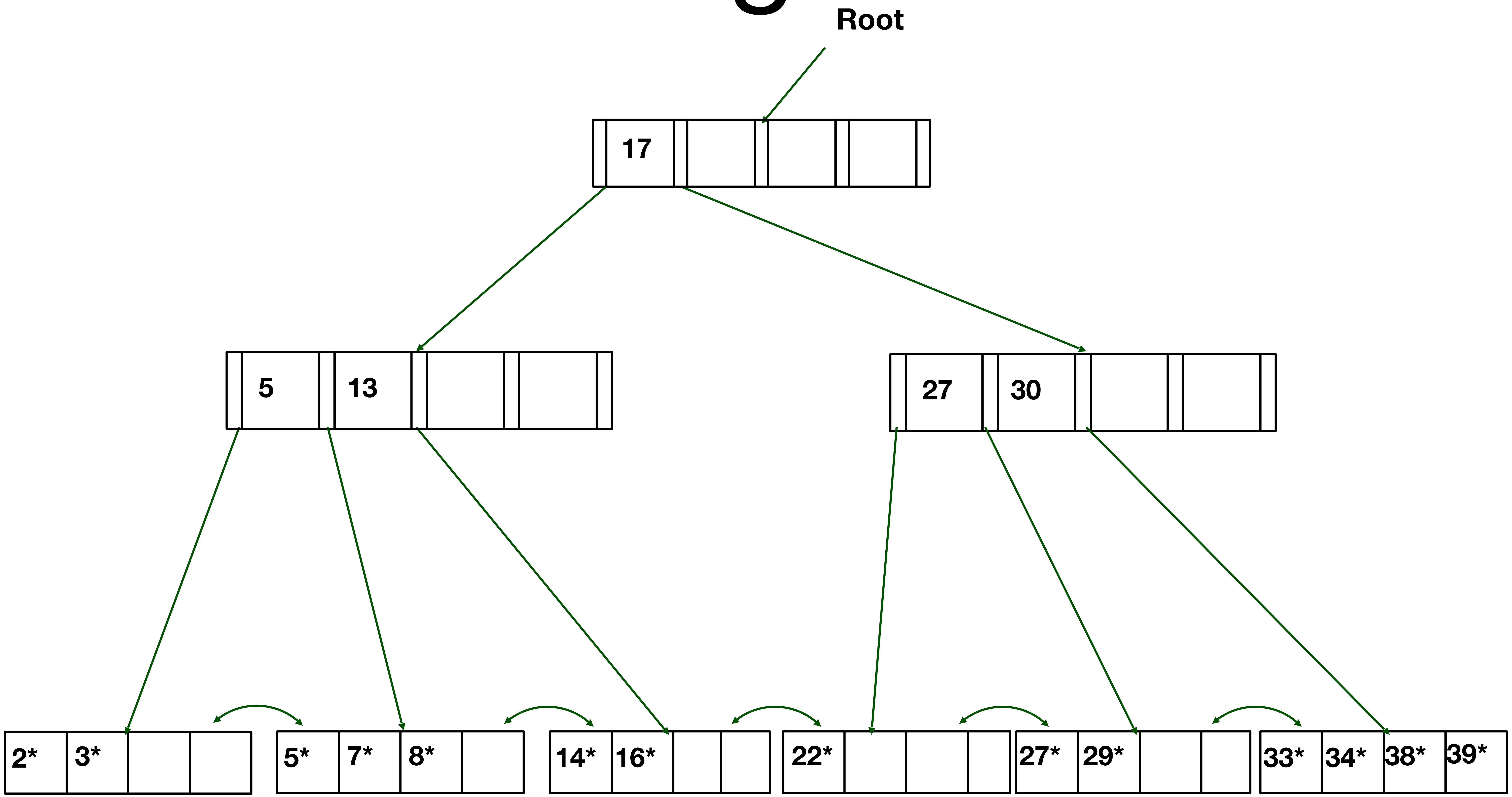
Source: Ramakrishnan & Gehrke

After Deleting 20*

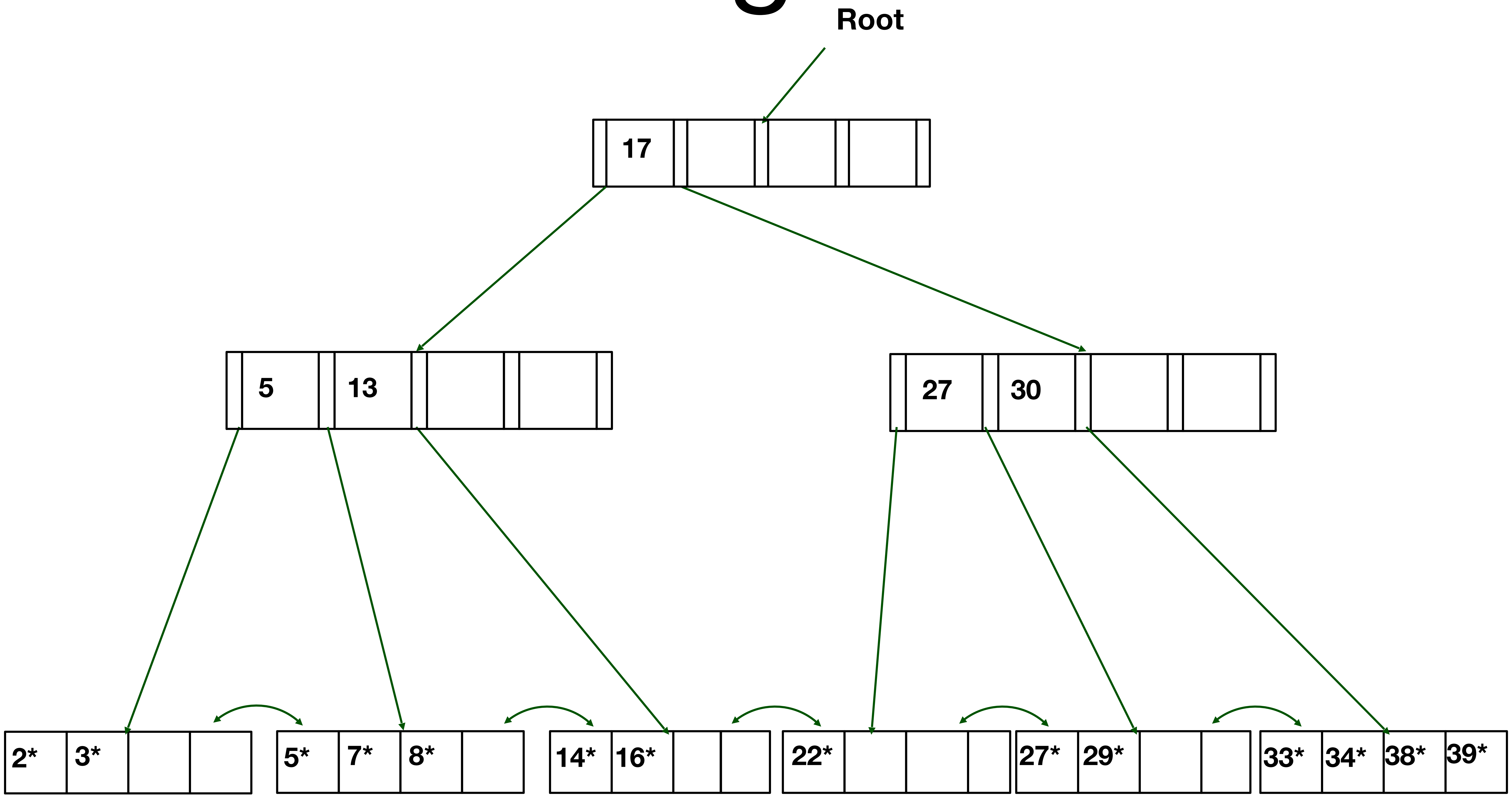


Source: Ramakrishnan & Gehrke

Deleting 24* …

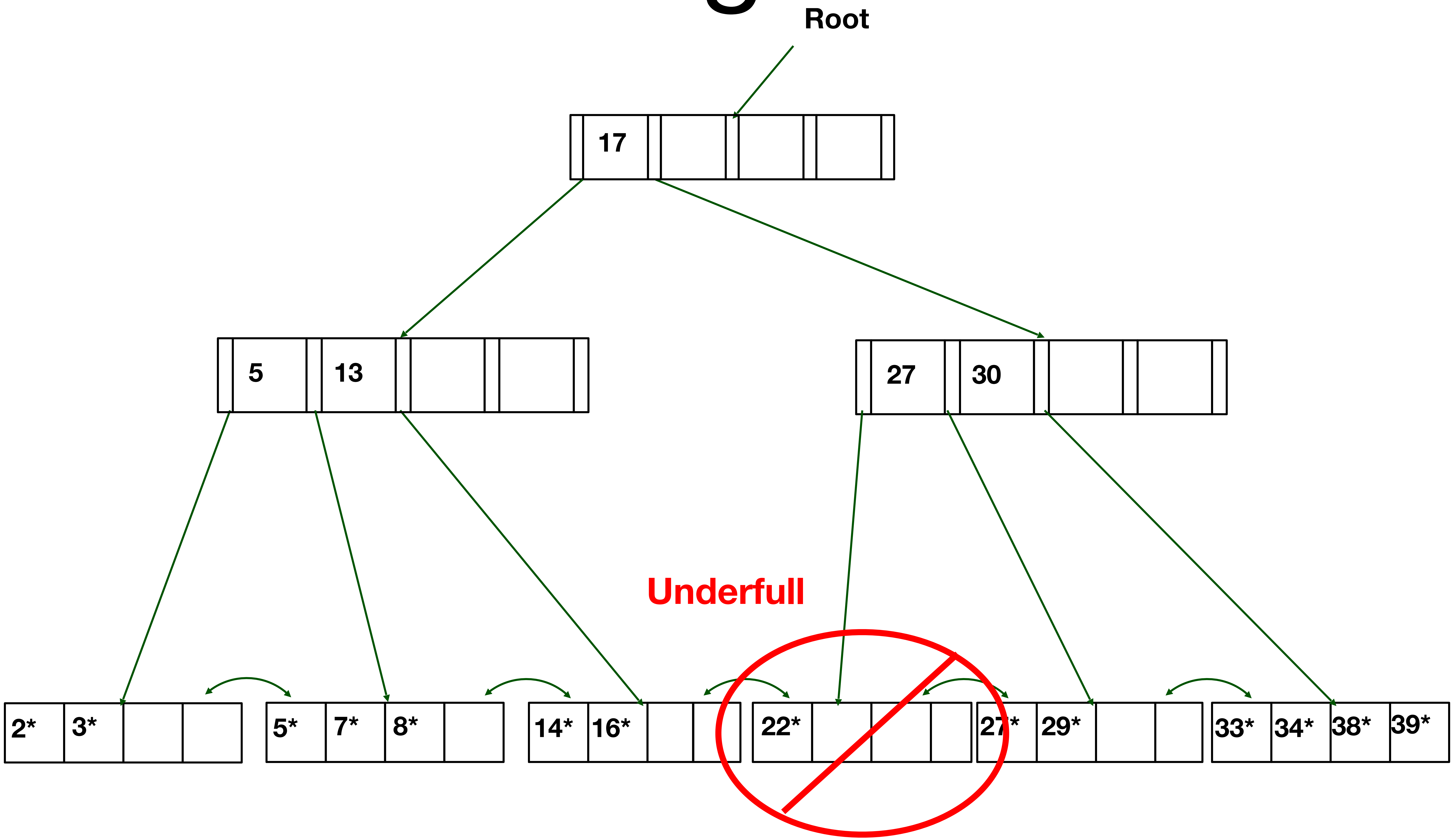


Deleting 24* ...



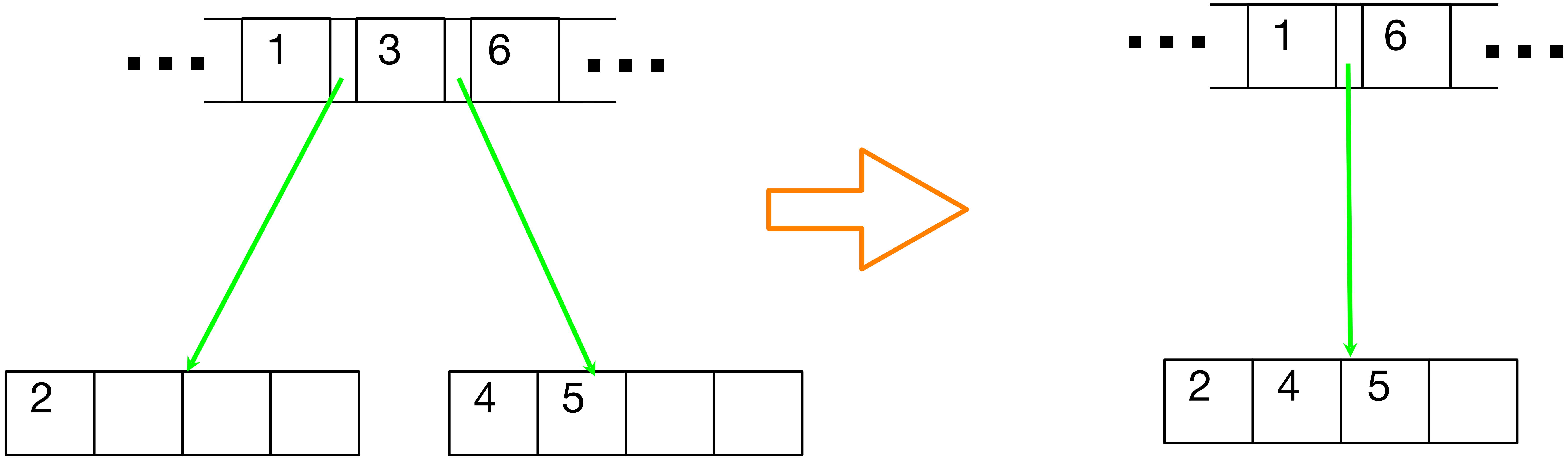
Source: Ramakrishnan & Gehrke

Deleting 24* …



Source: Ramakrishnan & Gehrke

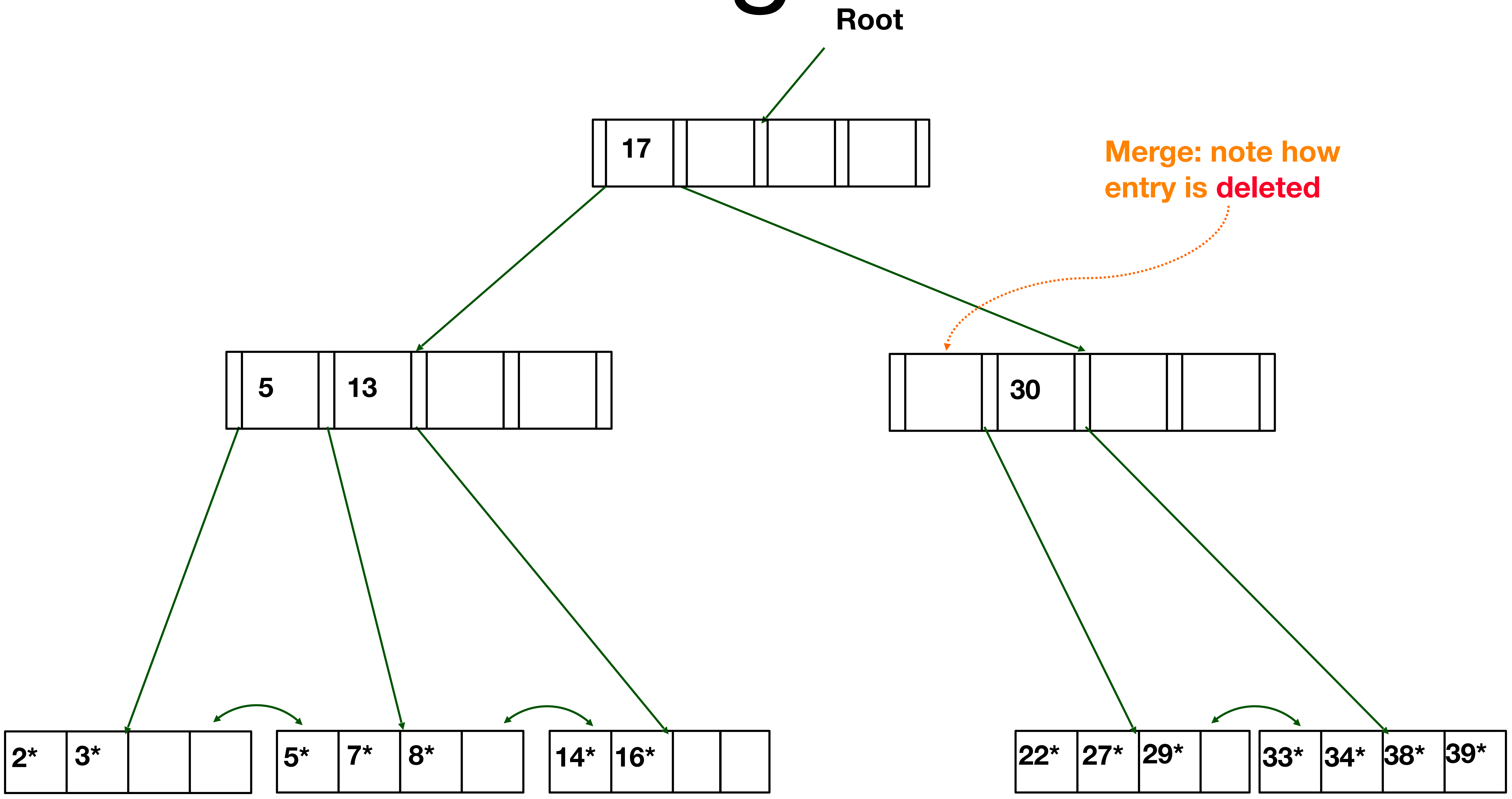
Merging/Deleting Underfull Leaves ...



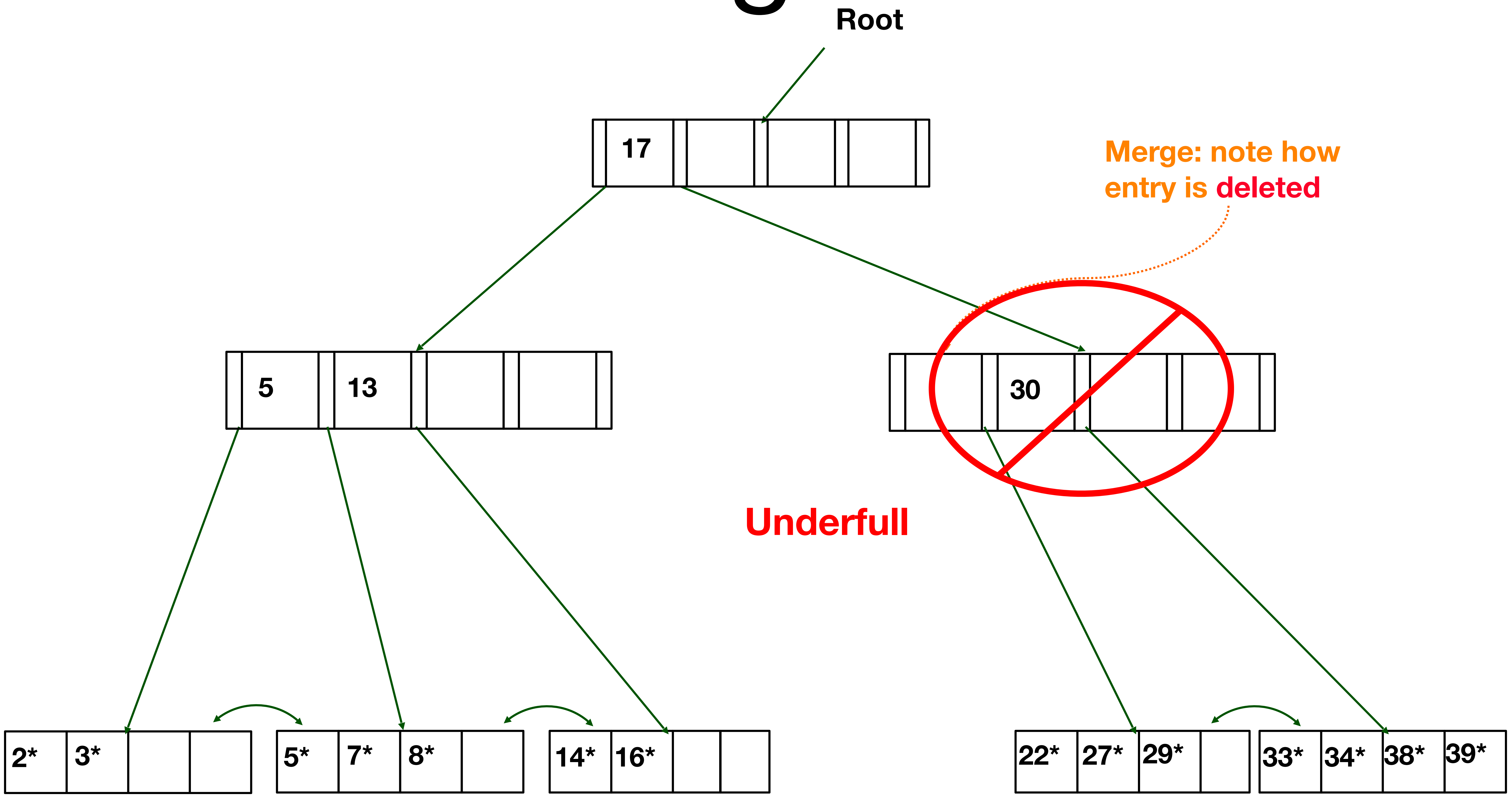
- If the node fits in its sibling, delete node ...
- This **deletes** a key from parent
- Parent may become underfull, so repeat

Source: Ramakrishnan & Gehrke

Deleting 24* ...

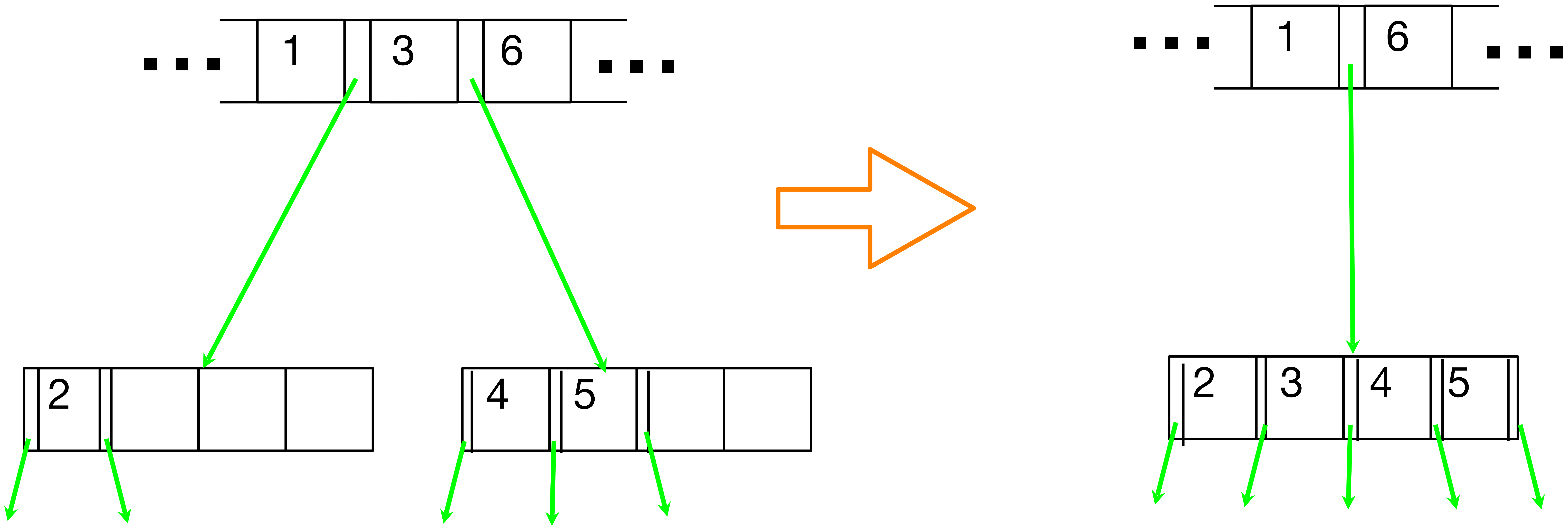

Root
17
Merge: note how entry is deleted
5
13
30
2*
3*
5*
7*
8*
14*
16*
22*
27*
29*
33*
34*
38*
39*

Deleting 24* ...



Root
17
5
13
30
Merge: note how entry is deleted
Underfull
2*
3*
5*
7*
8*
14*
16*
22*
27*
29*
33*
34*
38*
39*

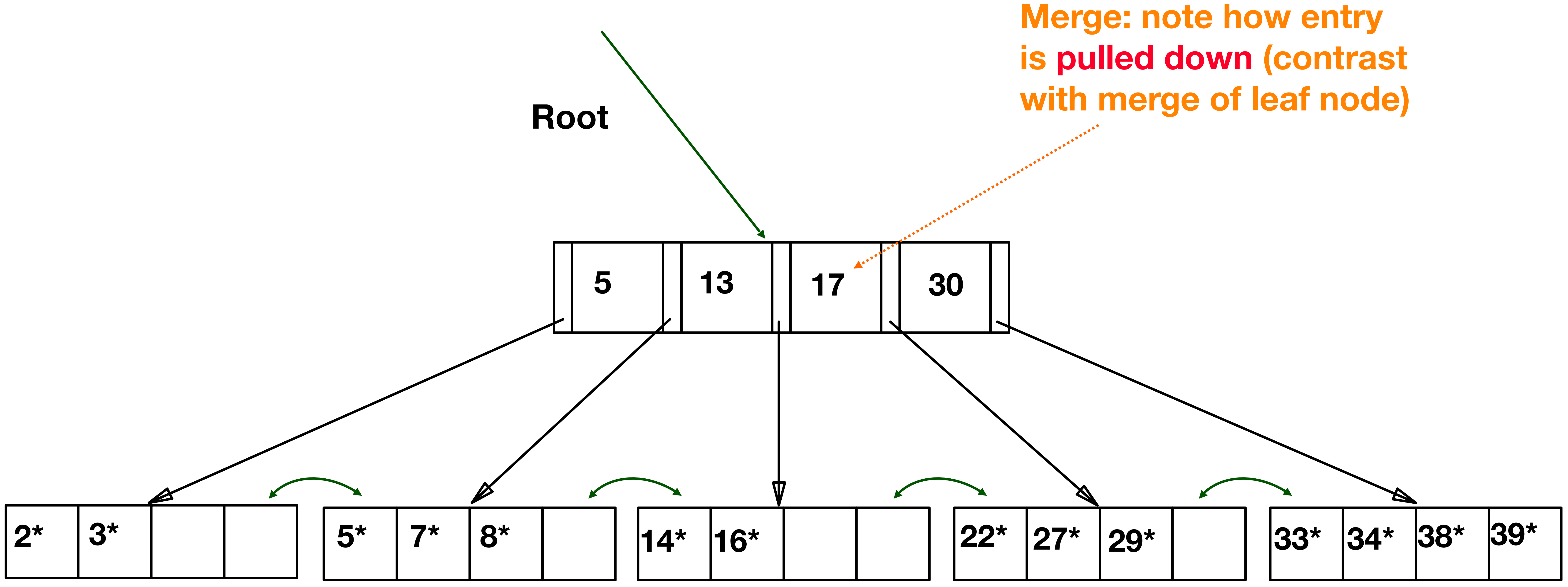
Merging Underfull Nonleaves ...



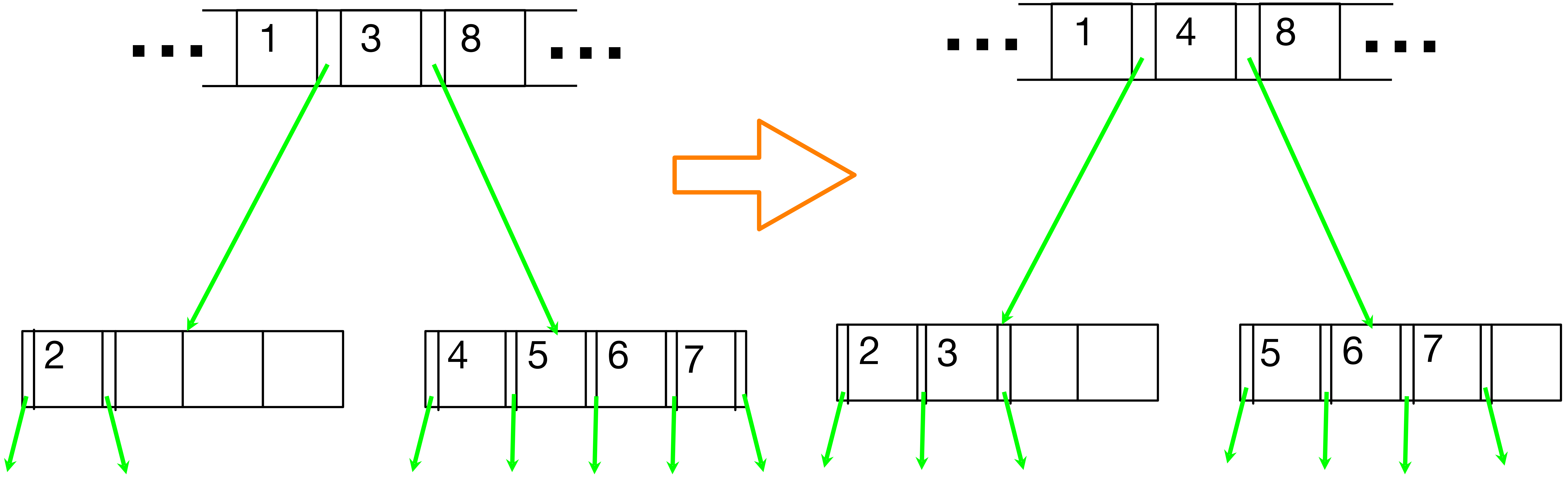
- If the node fits in its sibling, delete ...
- This **moves** a key from parent to merged child
- Parent may become underfull, so repeat

Source: Ramakrishnan & Gehrke

Deleting 24* …


Root
Merge: note how entry is pulled down (contrast with merge of leaf node)
5
13
17
30
2*
3*
5*
7*
8*
14*
16*
22*
27*
29*
33*
34*
38*
39*

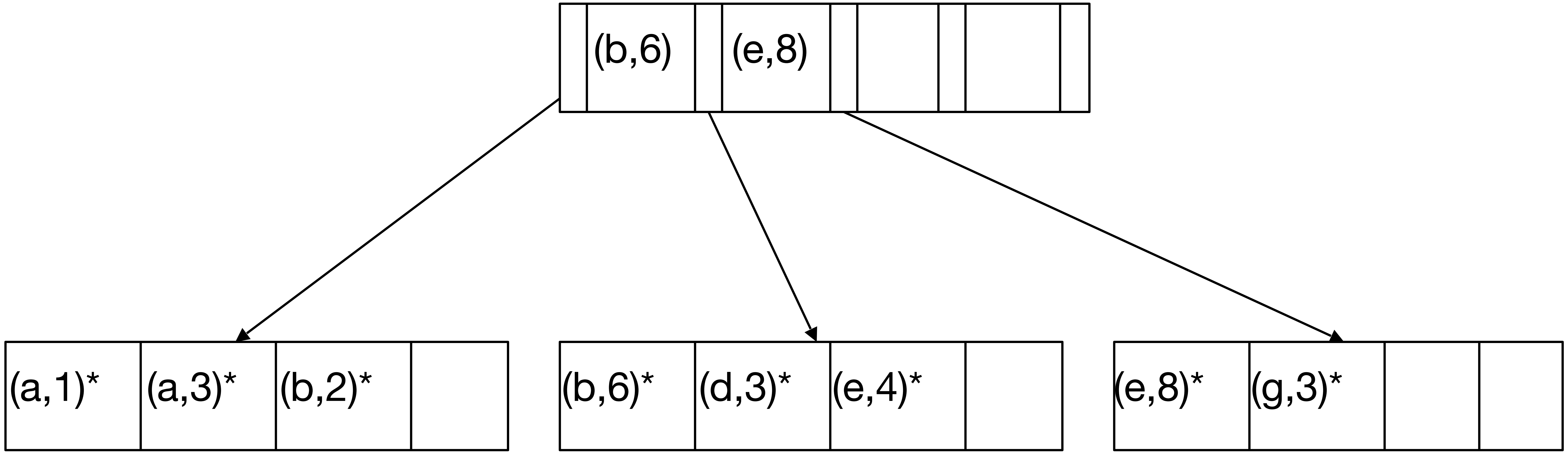
Redistributing Underfull Nonleaves ...



- Must redistribute if sibling node is full
- Keys are moved rather than copied
- Degree of parent unchanged

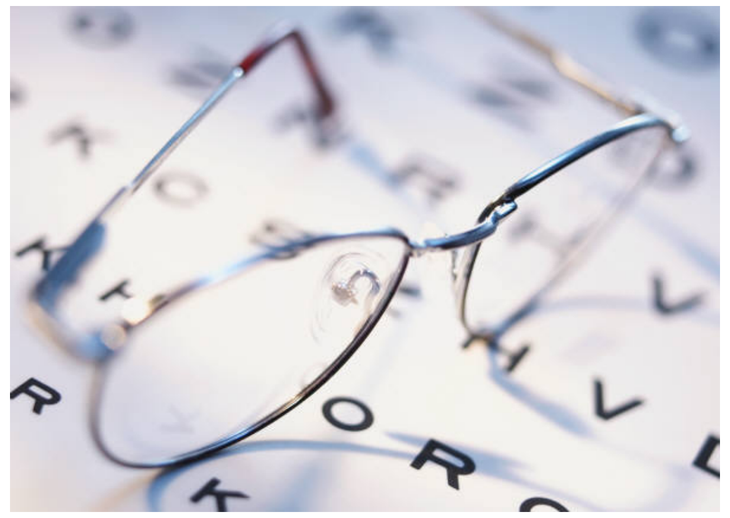
Source: Ramakrishnan & Gehrke

Composite Search Keys



What should we learn today?

- Explain the concept of (**materialized**) **views** and create them in SQL
- Explain the concept of **triggers** and create them in SQL
- Argue for when to use (**materialized**) **views** vs. **triggers**
- Explain the **ACID** properties of transactions
- Formulate **transaction programs** in SQL



- Identify the main types of representations and access methods for relations in DBMS, namely **heap files** and **indexes**
- Explain the core algorithms for insertion, deletion, and search in a **B+-tree**