UC San Diego

CSE 132C
Database System Implementation

Arun Kumar

Topic 2: Indexing

Chapters 10 and 11 of Cow Book

Slide ACKs: Jignesh Patel, Paris Koutris

Motivation for Indexing

- Consider the following SQL query:

Movies (M)

<u>MovieID</u>	Name	Year	Director

```
SELECT * FROM Movies WHERE Year=2017
```

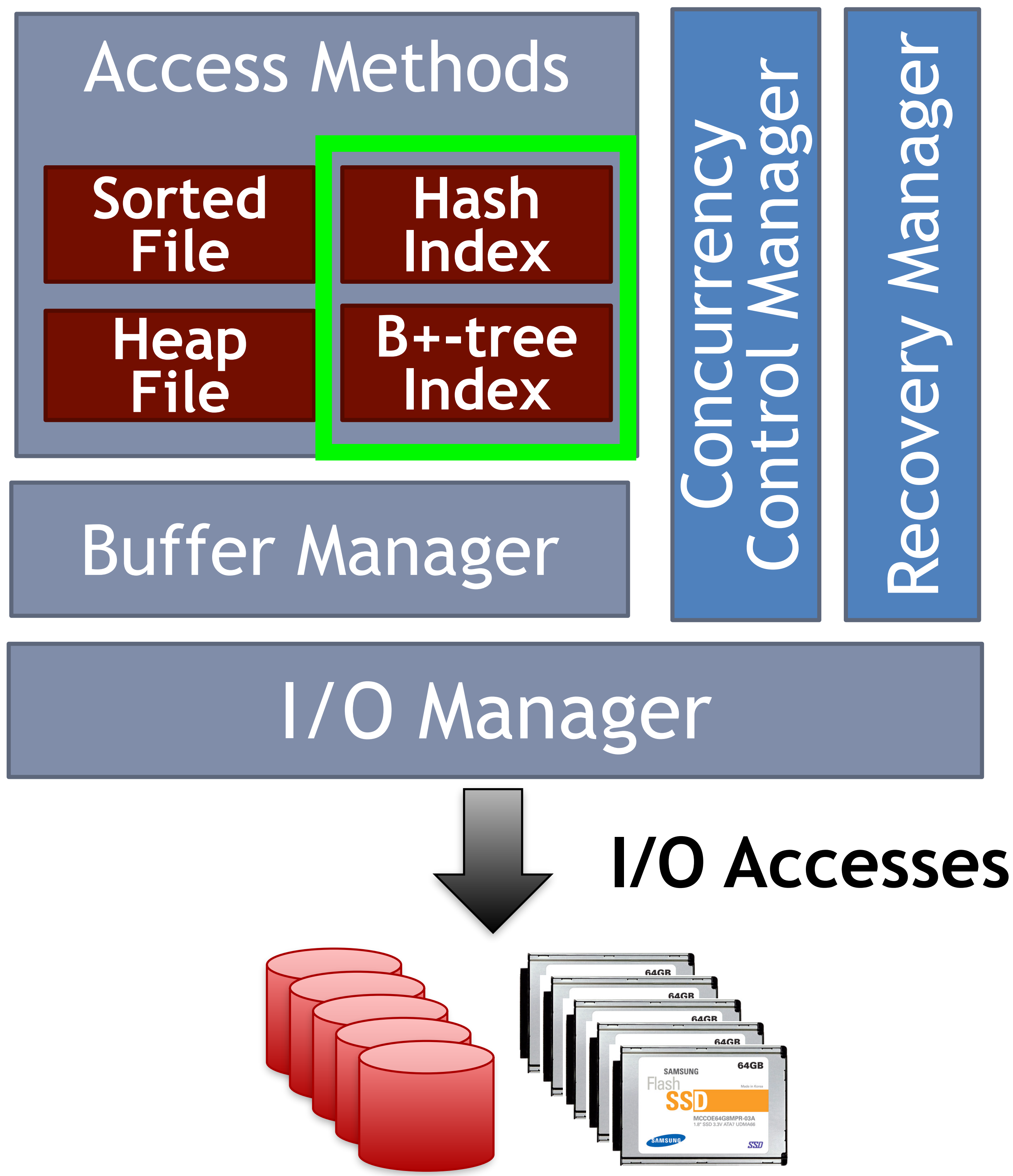
Q: *How to obtain the matching records from the file?*

- Heap file? Need to do a linear scan! O(N) I/O and CPU
- "Sorted" file? Binary search! $O(\log_2(N))$ I/O and CPU

Indexing helps retrieve records faster for selective predicates!

```
SELECT * FROM Movies WHERE Year>=2000 AND Year<2010
```

Another View of Storage Manager



Indexing: Outline

- **Overview and Terminology**
- **B+ Tree Index**
- **Hash Index**

Indexing

- **Index**: A data structure to speed up record retrieval
  - **Search Key**: Attribute(s) on which file is indexed; also called **Index Key** (used interchangeably)
  - Any *permutation* of any *subset* of a relation's attributes can be index key for an index
  - Index key need not be a primary/candidate key
- Two main types of indexes:
  - **B+ Tree** index: good for both range and *equality* search
  - **Hash** index: good for *equality* search

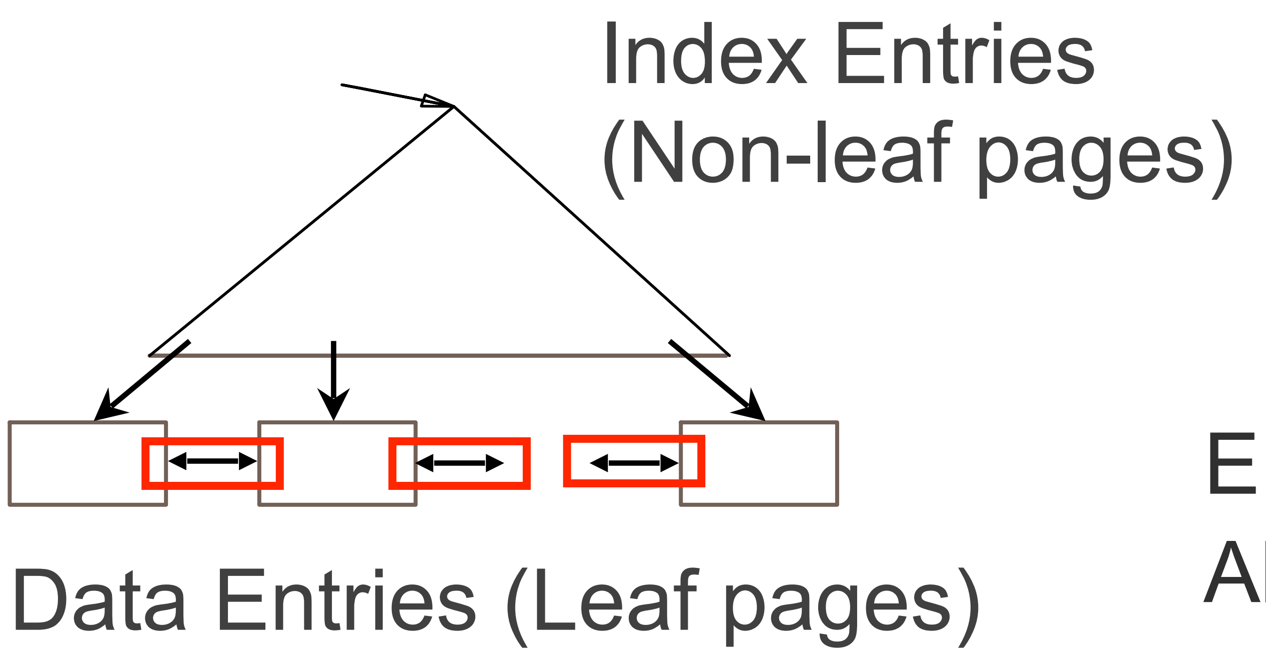
Overview of Indexes

- Need to consider efficiency of search, insert, and delete
  - Primarily optimized to *reduce (disk) I/O cost*
- **B+ Tree** index:
  - $O(\log_F(N))$ I/O and CPU cost for equality search (N: number of "data entries"; F: "fanout" of non-leaf node)
  - Range search, Insert, and Delete all start with an equality search
- **Hash** index:
  - O(1) I/O and CPU cost for equality search
  - Insert and delete start with equality search
  - Not "good" for range search!

What is stored in the Index?

- 2 things: Search/index **key values** and **data entries**
- Alternatives for data entries for a given key value *k*:
  - **AltRecord**: Actual data records of file that match *k*
  - **AltRID**: <*k*, RID of a record that matches *k*>
  - **AltRIDlist**: <*k*, list of RIDs of records that match *k*>
- API for operations on records:
  - **Search** (IndexKey); could be a predicate for B+Tree
  - **Insert** (IndexKey, data entry)
  - **Delete** (IndexKey); could be a predicate for B+Tree

Overview of B+ Tree Index



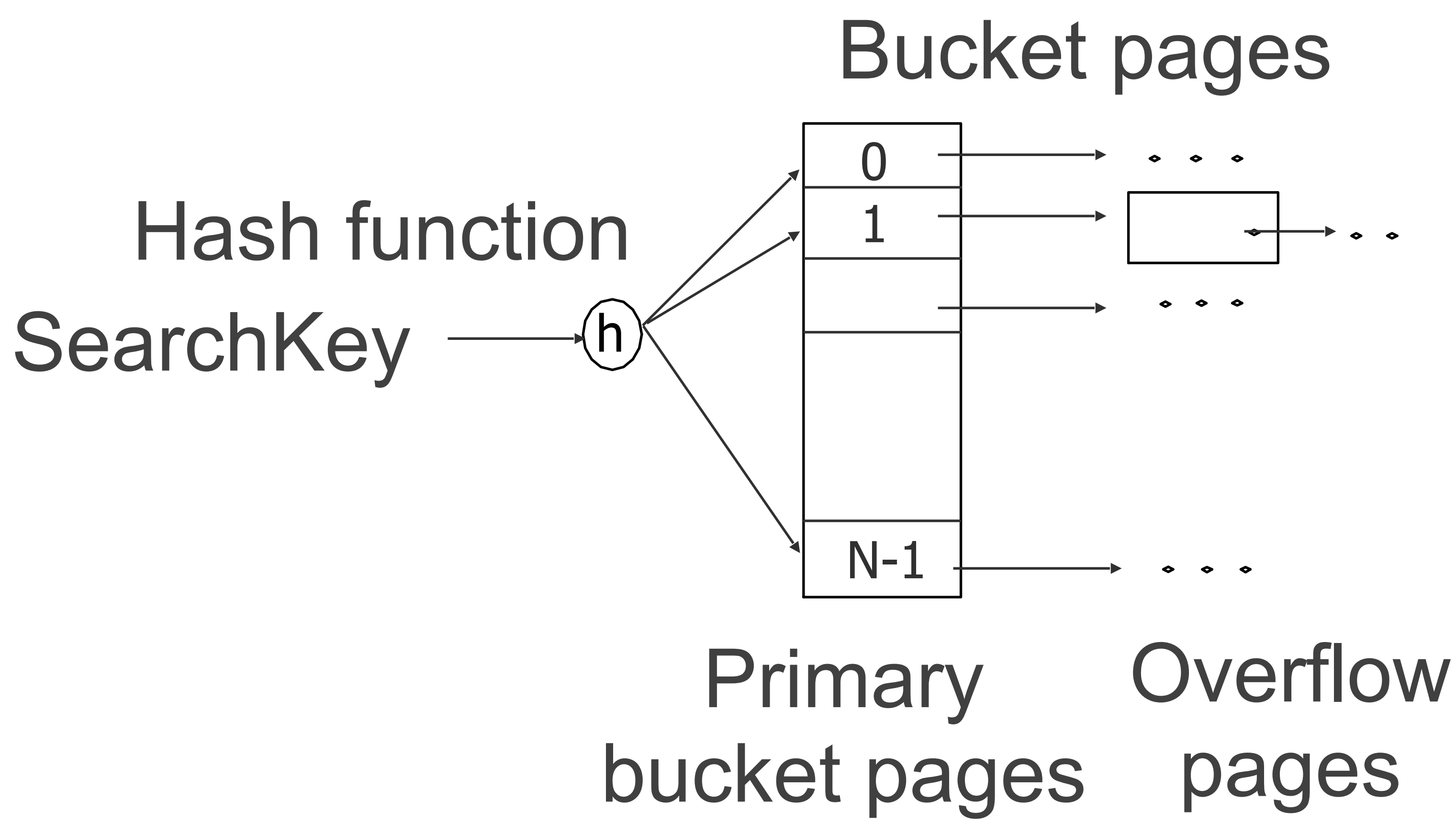
Entries of the form:
(IndexKey value, PageID)

Entries of the form:
AltRID: (IndexKey value, RID)

- Non-leaf pages do not contain data values; they contain [d, 2d] index keys; d is **order** parameter
- Height-balanced tree; only root can have [1,d) keys
- Leaf pages in *sorted order* of IndexKey; connected as a doubly linked list

Q: *What is the difference between "B+ Tree" and "B Tree"?*

Overview of Hash Index



- Bucket pages have data entries (same 3 Alternatives)
- Hash function helps obtain O(1) search time

Trade-offs of Data Entry Alternatives

- Pros and cons of alternatives for data entries:
  - **AltRecord**: Entire file is stored as an index! If records are long, data entries of index are large and search time could be high
  - **AltRID** and **AltRIDlist**: Data entries typically smaller than records; often faster for equality search
  - **AltRIDlist** has more compact data entries than AltRID but entries are variable-length

Q: *A file can have at most one AltRecord index. Why?*

More Indexing-related Terminology

- **Composite** Index: IndexKey has > 1 attributes
- **Primary** Index: IndexKey contains the primary key
- **Secondary** Index: Any index that not a primary index
- **Unique** Index: IndexKey contains a candidate key
  - All primary indexes are unique indexes!

MovieID	**Name**	**Year**	**Director**	**IMDB_URL**

Index on MovieID?
Index on Year?
Index on Director?
Index on IMDB_URL?
Index on (Year,Name)?

IMDB_URL is a candidate key

More Indexing-related Terminology

- **Clustered** index: order in which records are laid out is same as (or "very close to") order of IndexKey domain
  - Matters for (range) search performance!
  - AltRecord implies index is clustered. Why?
  - In practice, clustered almost always implies AltRecord
  - In practice, a file is clustered on at most 1 IndexKey
- **Unclustered** index: an index that is not clustered

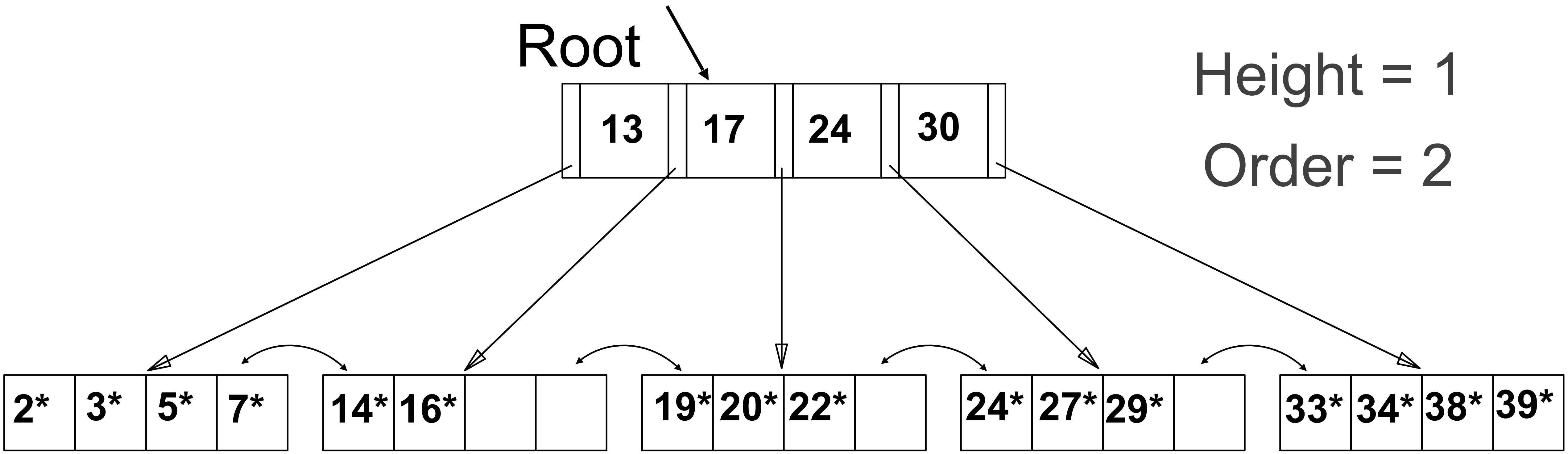
<u>MovieID</u>	Name	Year	Director	IMDB_URL

Index on Year? Index on (Year, Name)?

Indexing: Outline

- **Overview and Terminology**
- **B+ Tree Index**
- **Hash Index**

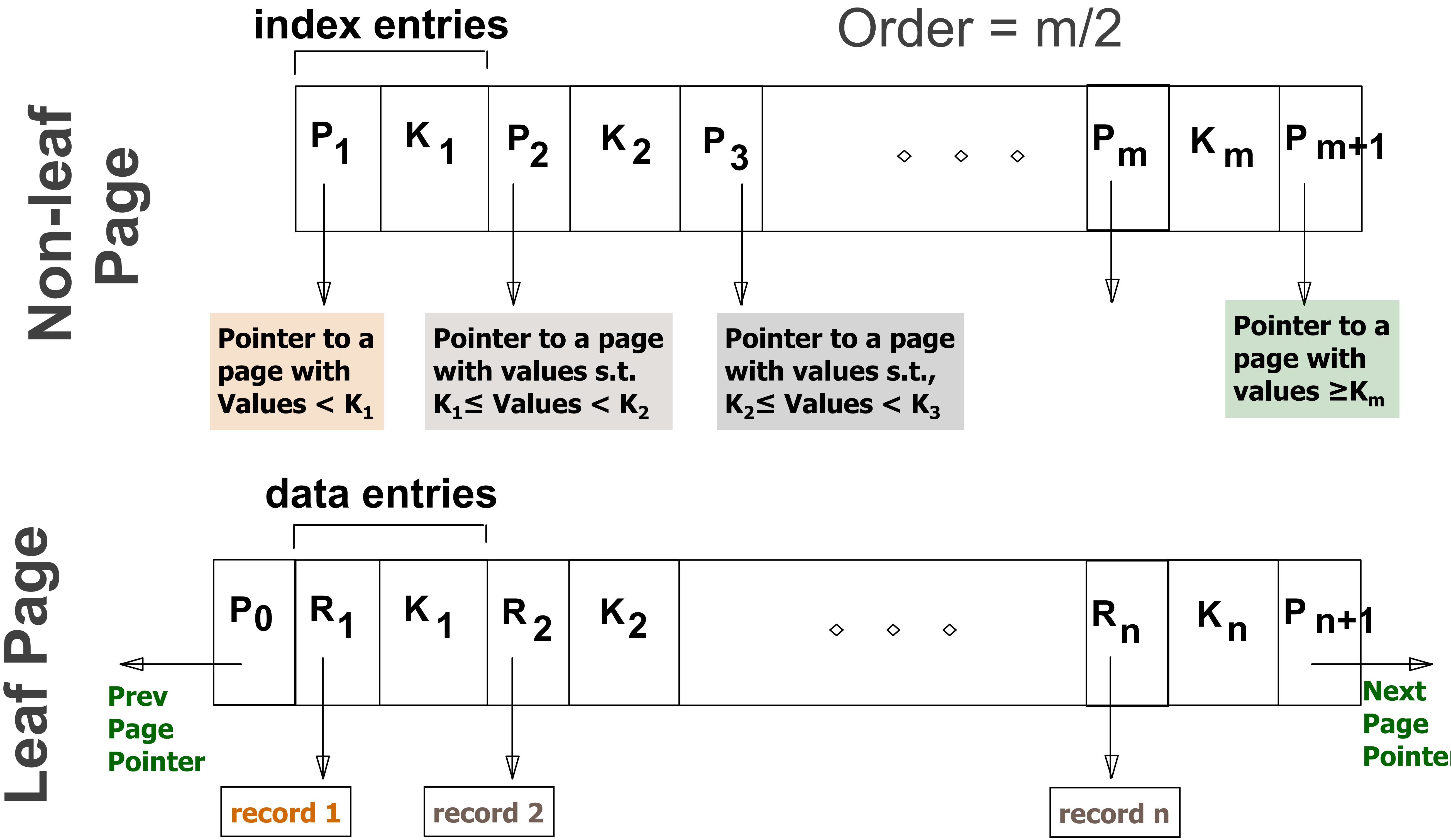
B+ Tree Index: Search



- Given SearchKey *k*, start from root; compare *k* with IndexKeys in non-leaf/index entries; descend to correct child; keep descending like so till a leaf node is reached
- Comparison within non-leaf nodes: binary/linear search

Examples: search 7*; 8*; 24*; range [19*,33*]

B+ Tree Index: Page Format



B+ Trees in Practice

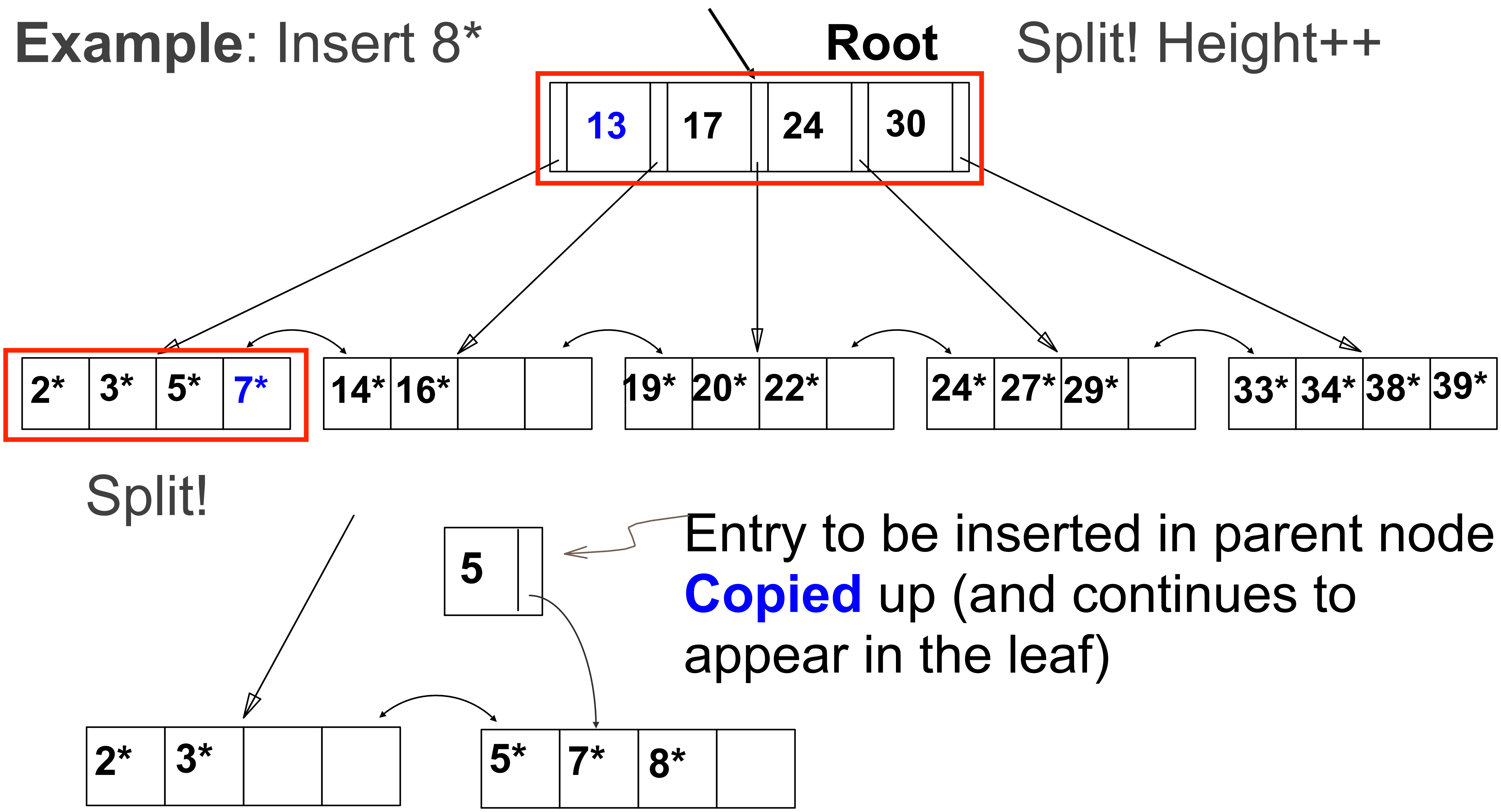
- Typical order value: 100 (so, non-leaf node can have up to 200 index keys)
  - Typical occupancy: 67%; so, typical "**fanout**" = 133
- Computing the tree's capacity using fanout:
  - Height 1 stores 133 leaf pages
  - Height 4 store 133^4 = 312,900,700 leaf pages
- Typically, higher levels of B+Tree cached in buffer pool
  - Level 0 (root) = 1 page = 8 KB
  - Level 1 = 133 pages ~ 1 MB
  - Level 2 = 17,689 pages ~ 138 MB and so on

B+ Tree Index: Insert

- Search for correct leaf *L*
- Insert data entry into *L*; if *L* has enough space, *done!*
  Otherwise, must **split** *L* (into new L and a new leaf L')
  - Redistribute entries evenly, **copy up** middle key
  - Insert index entry pointing to *L'* into parent of *L*
- A split might have to *propagate upwards recursively*:
  - To split non-leaf node, redistribute entries evenly, but **push up** the middle key (not copy up, as in leaf splits!)
- Splits "grow" the tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top*.

B+ Tree Index: Insert

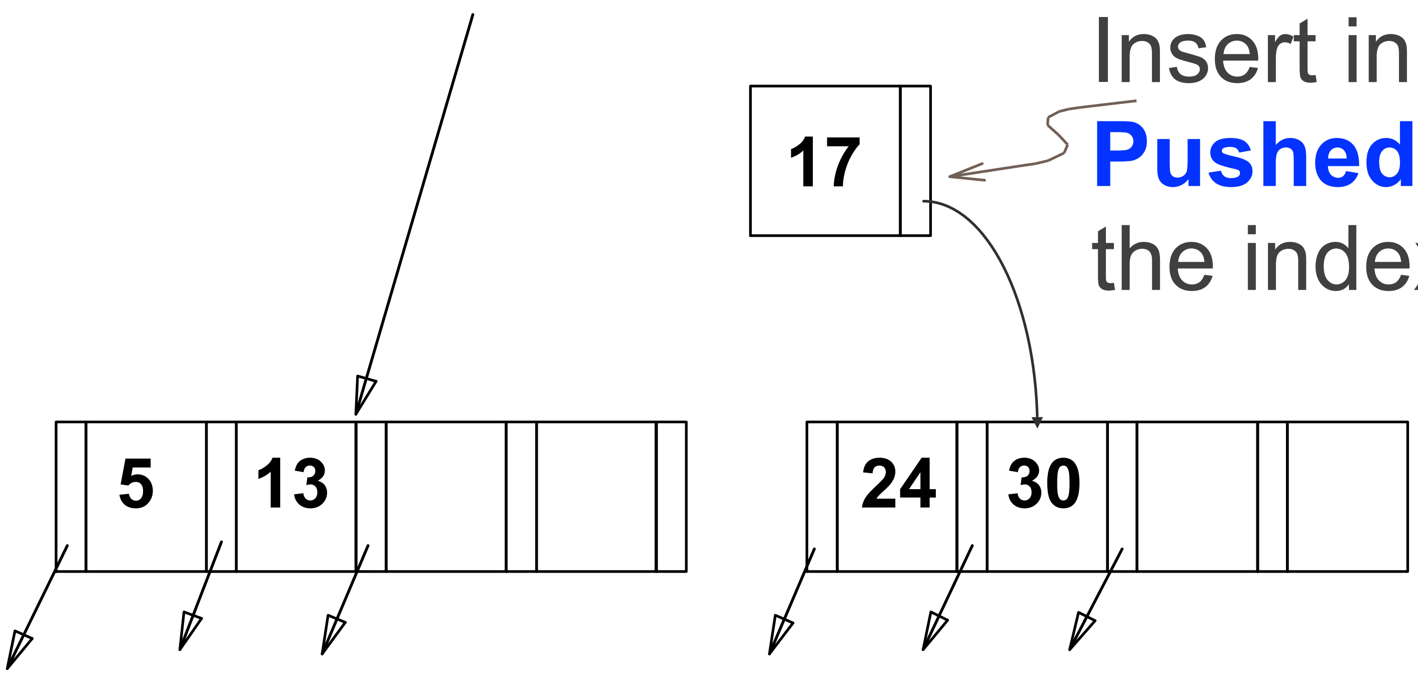
Example: Insert 8*



Entry to be inserted in parent node
Copied up (and continues to appear in the leaf)

B+ Tree Index: Insert

Example: Insert 8*

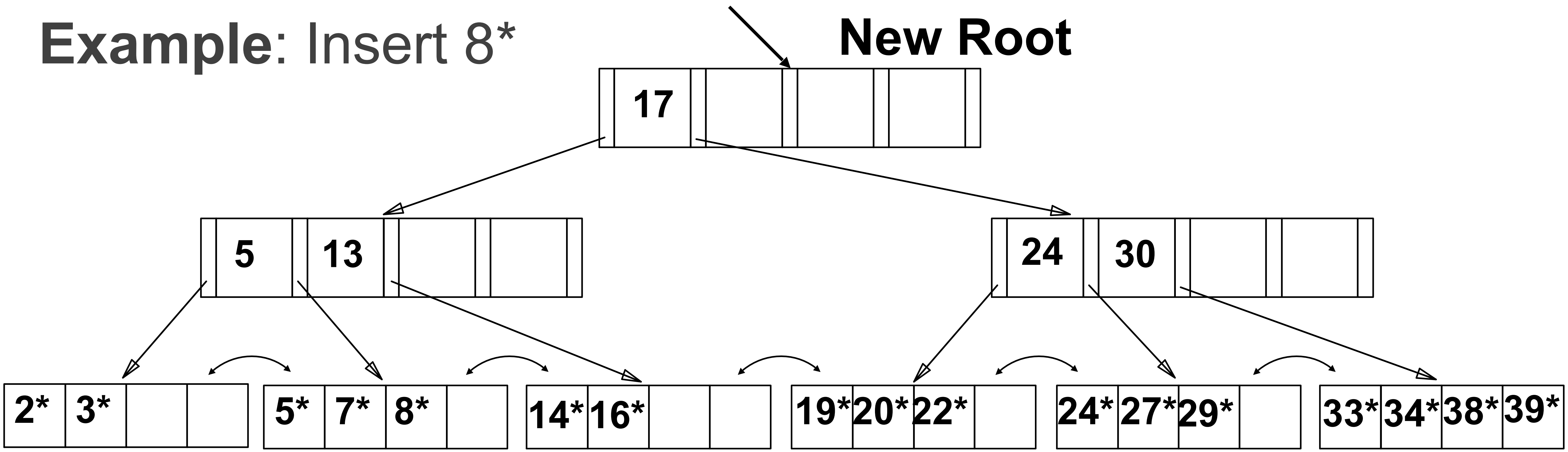


Insert in parent node.
Pushed up (and only appears once in the index)

Minimum occupancy is guaranteed in both leaf and non-leaf page splits

B+ Tree Index: Insert

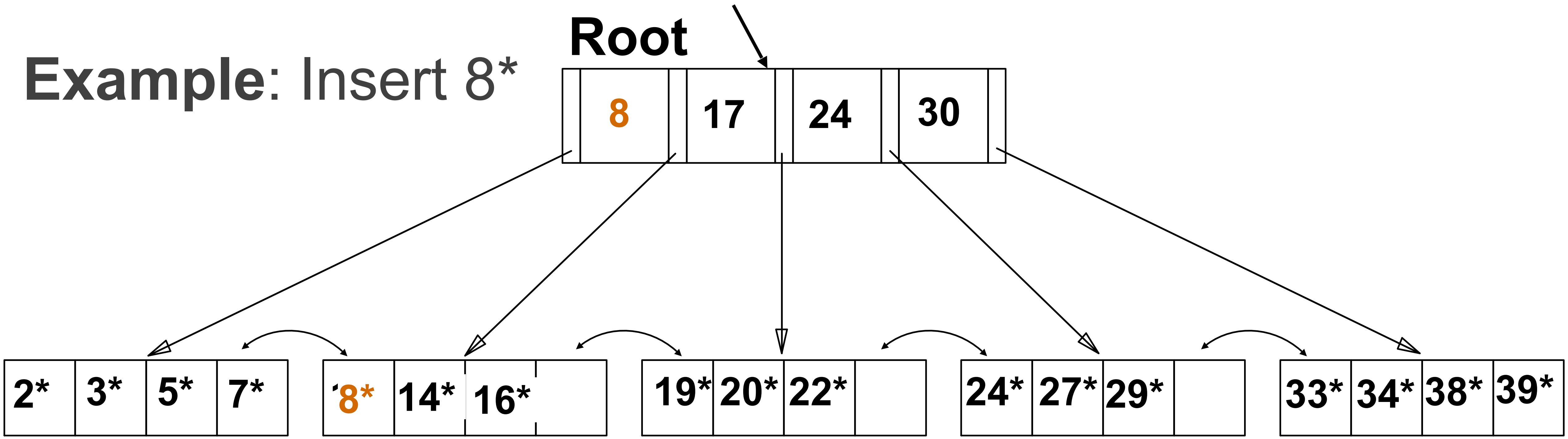
Example: Insert 8*



- Recursive splitting went up to root; height went up by 1
- Splitting is somewhat expensive; is it avoidable?
  - Can *redistribute* data entries with left or right sibling, if there is space!

Insert: Leaf Node Redistribution

Example: Insert 8*

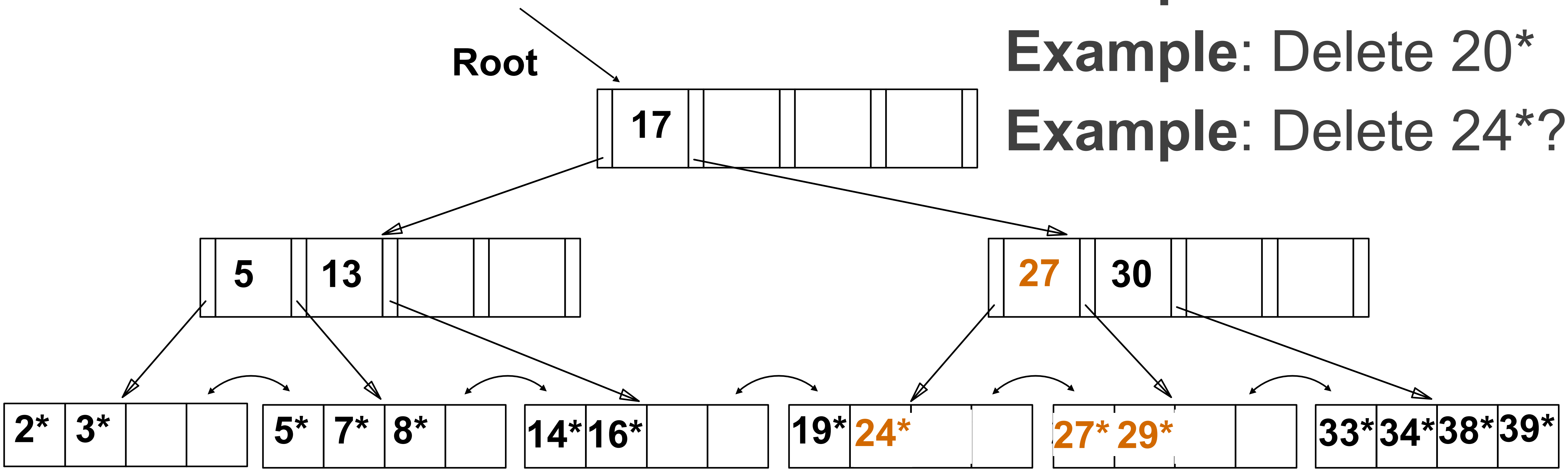


- Redistributing data entries with a sibling improves page occupancy at leaf level and avoids too many splits; but usually *not* used for non-leaf node splits
  - Could increase I/O cost (checking siblings)
  - Propagating internal splits is better amortization
  - Pointer management headaches

B+ Tree Index: Delete

- Start at root, find leaf *L* where entry belongs
- Remove the entry; if *L* is at least half-full, *done!* Else, if *L* has only d-1 entries:
  - Try to **re-distribute**, borrowing from sibling *L'*
  - If re-distribution fails, **merge** *L* and *L'* into single leaf
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- A merge might have to propagate upwards recursively to root, which decreases height by 1

B+ Tree Index: Delete



Example: Delete 22*

Example: Delete 20*

Example: Delete 24*?

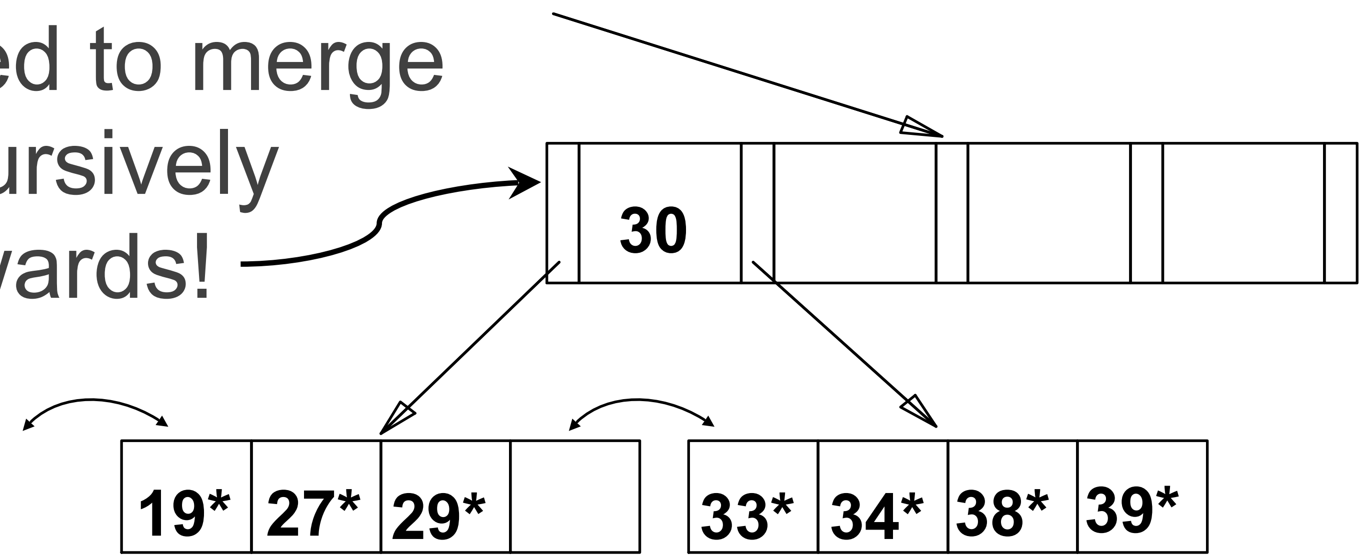
- Deleting 22* is easy
- Deleting 20* is followed by *redistribution* at leaf level. Note how middle key is **copied up**.

B+ Tree Index: Delete

Example: Delete 24*

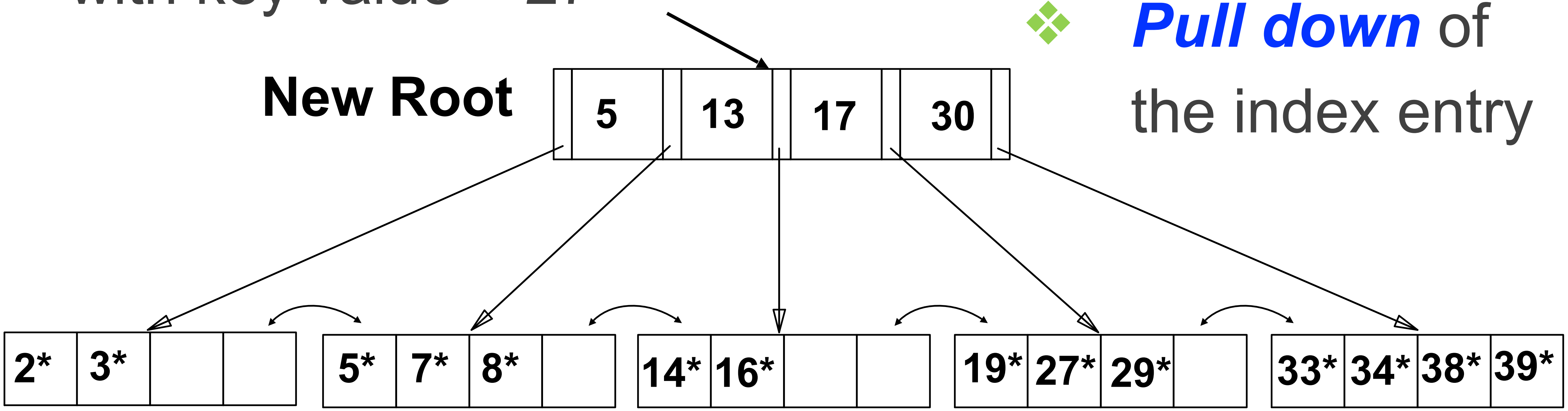
- Must merge leaf nodes!
- In non-leaf node, ***remove*** index entry with key value = 27

Need to merge recursively upwards!



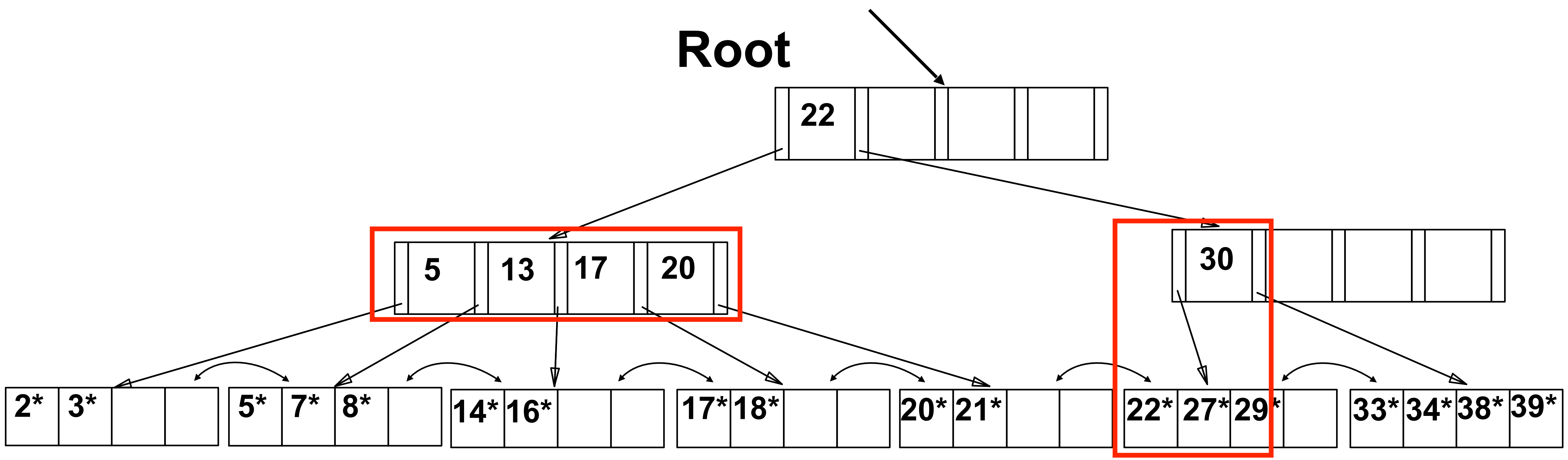
- ***Pull down*** of the index entry

New Root



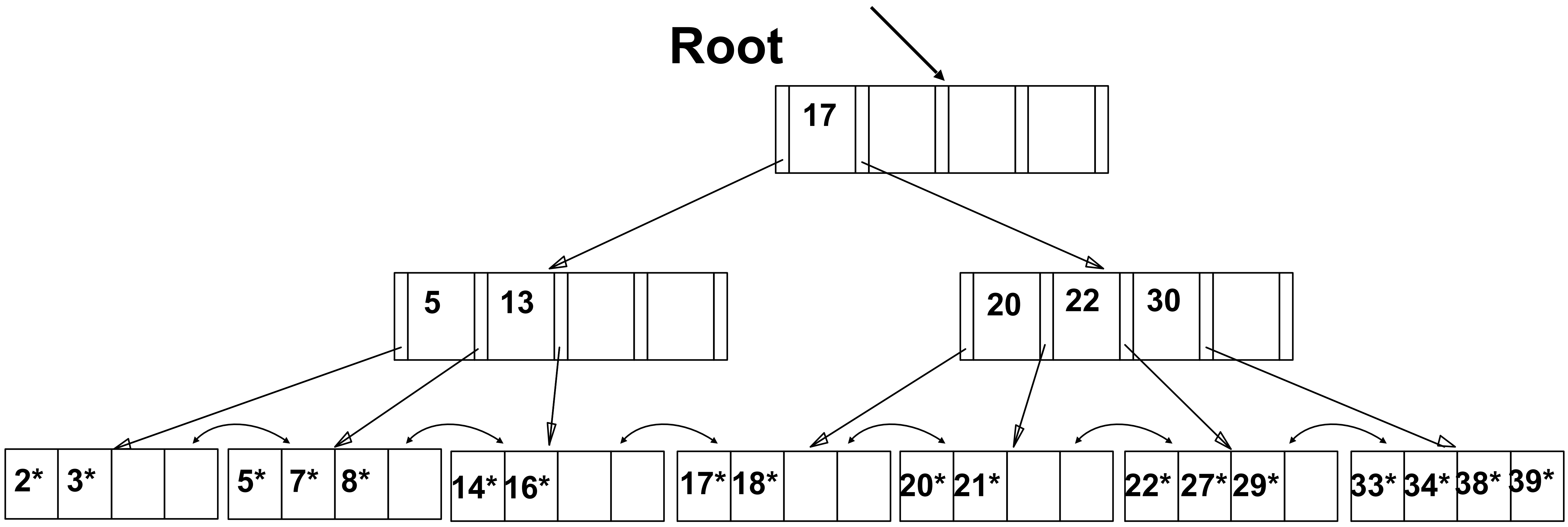
Delete: Non-leaf Node Redistribution

- Suppose this is the state of the tree when deleting 24*
- Instead of merge of root's children, we can also redistribute entry from left child of root to right child



Delete: After Redistribution

- **Rotate** IndexKeys through the parent node
- It suffices to re-distribute index entry with key 20; for illustration, 17 also re-distributed



Delete: Redistribution Preferred

- Unlike Insert, where redistribution is discouraged for non-leaf nodes, Delete prefers redistribution over merge decisions at both leaf or non-leaf levels. Why?
  - Files usually grow, not shrink; deletions are rare!
  - High chance of redistribution success (high fanouts)
  - Only need to propagate changes to parent node

Handling Duplicates/Repetitions

- Many data entries could have same IndexKey value
  - Related to AltRIDlist vs AltRID for data entries
  - Also, single data entry could still span multiple pages
- Solution 1:
  - All data entries with a given IndexKey value reside on a single page
  - Use “overflow” pages, if needed (not inside leaf list)
- Solution 2:
  - Allow repeated IndexKey values among data entries
  - Modify Search appropriately
    - Use RID to get a *unique* composite key!

Order Concept in Practice

- In practice, *order* (d) concept replaced by physical space criterion: *at least half-full*
  - Non-leaf pages can typically hold many more entries than leaf pages, since leaf pages could have long data records (AltRecord) or RID lists (AltRIDlist)
  - Often, different nodes could have different # entries:
    - Variable sized IndexKey
    - AltRecord and variable-sized data records
    - AltRIDlist could leads to different numbers of data entries sharing an IndexKey value

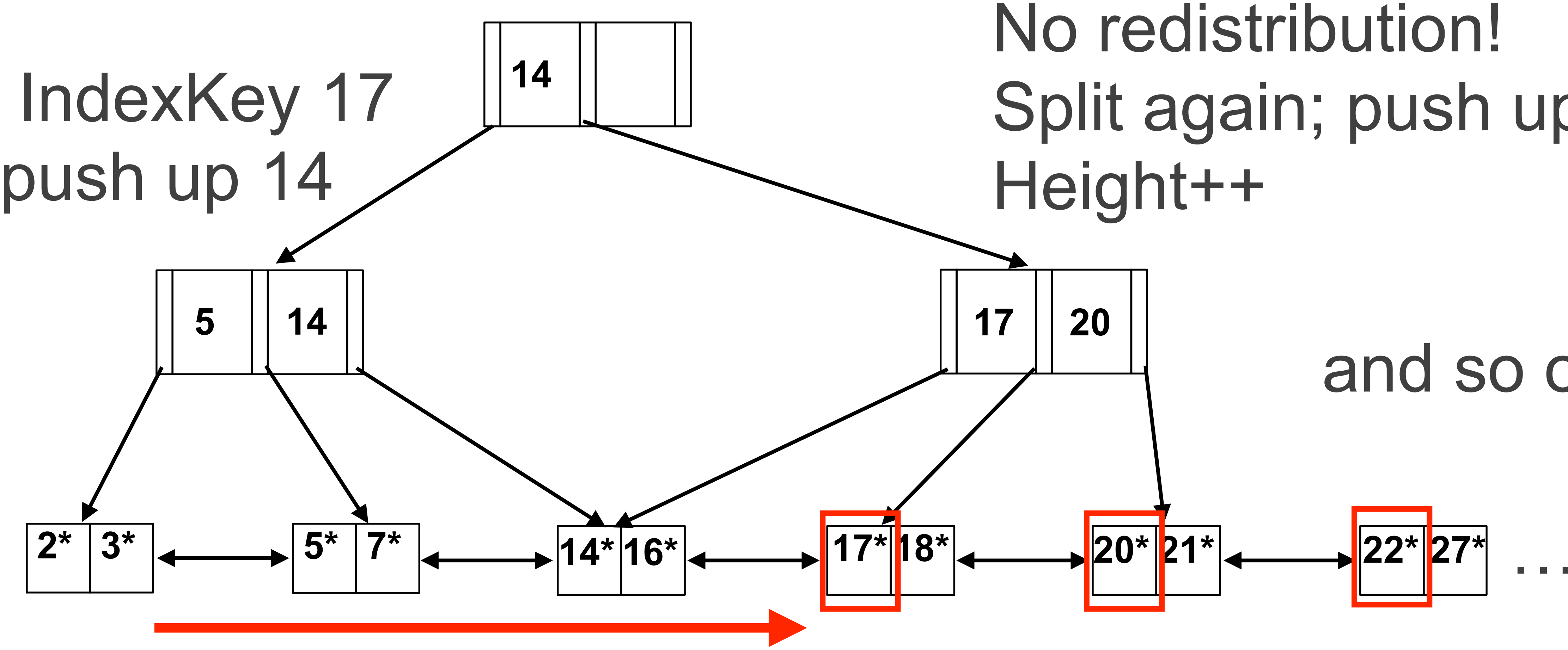
B+ Tree Index: Bulk Loading

- Given an existing file we want to index, multiple record-at-a-time Inserts are wasteful (too many IndexKeys!)
  - Bulk loading avoids this overhead; reduces I/O cost
- 1) Sort data entries by IndexKey (AltRecord sorts file!)
- 2) Create empty root page; *copy* leftmost IndexKey of leftmost leaf page to root (non-leaf) and assign child
- 3) Go left to right; Insert only leftmost IndexKey from each leaf page into index as usual (NB: fewer keys!)
- 4) When non-leaf fills up, follow usual Split procedure and recurse upwards, if needed

B+ Tree Index: Bulk Loading

Insert IndexKey 17
Split; push up 14

Insert IndexKey 22
No redistribution!
Split again; push up 20
Height++



and so on

Sorted pages of data entries

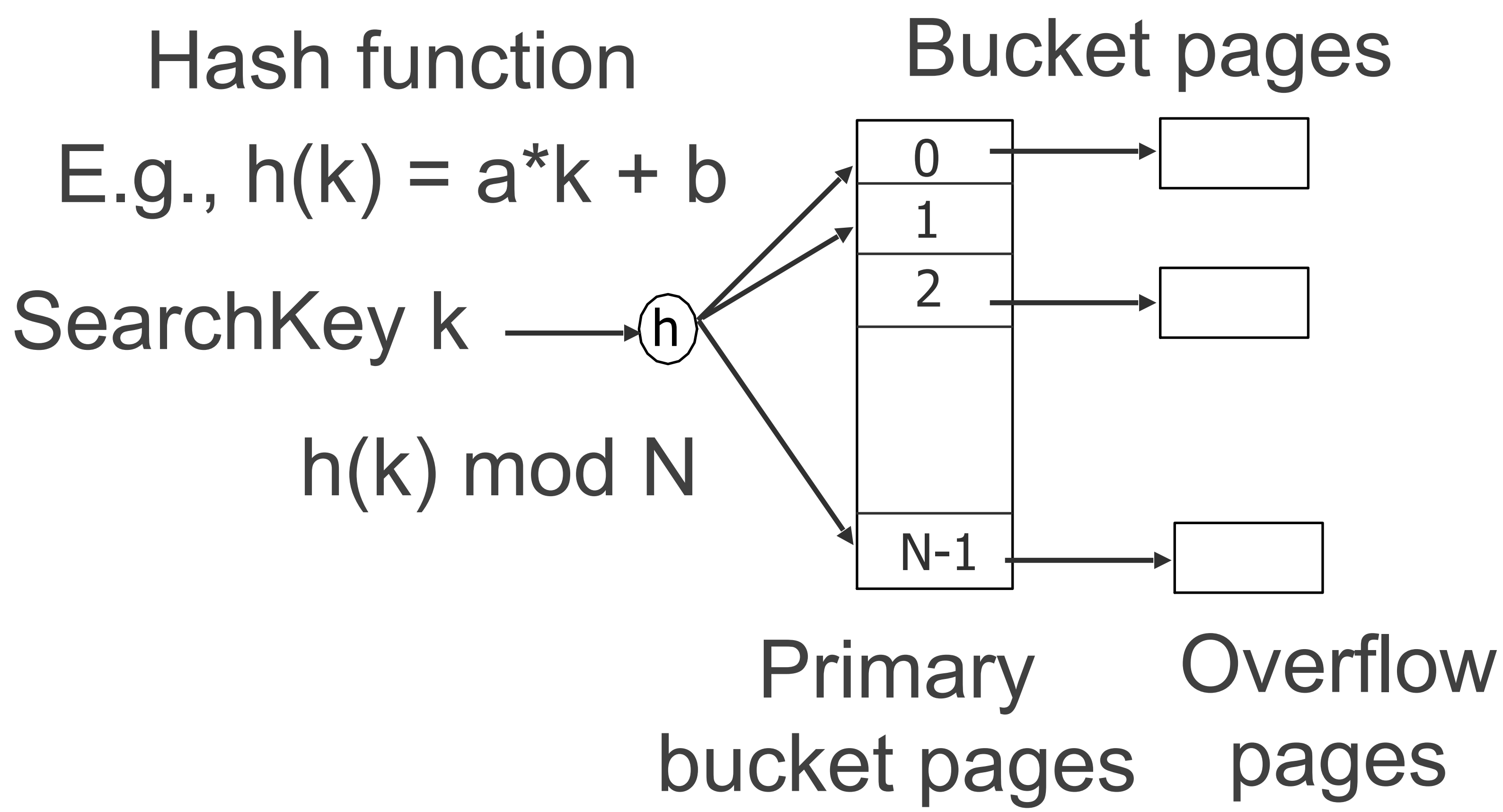
Indexing: Outline

- **Overview and Terminology**
- **B+ Tree Index**
- **Hash Index**

Overview of Hash Indexing

- Reduces search cost to nearly O(1)
- Good for *equality* search (but not for range search)
- Many variants:
  - Static hashing
  - Extendible hashing
  - Linear hashing, etc. (we will not discuss these)

Static Hashing



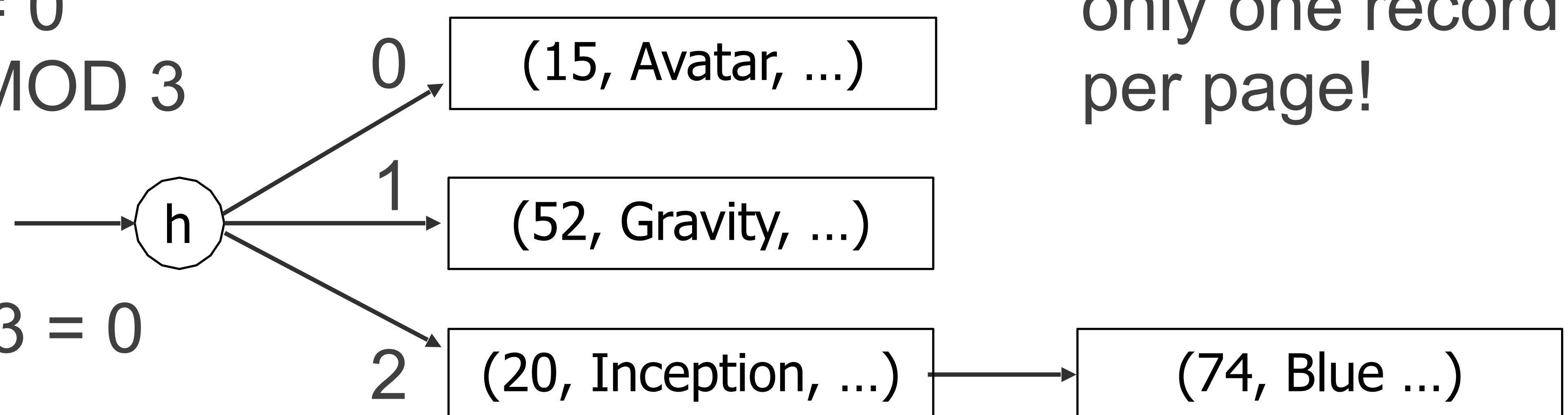
- N is fixed; primary bucket pages never deallocated
- Bucket pages contain data entries (same 3 Alts)
- Search:
  - Overflow pages help handle hash collisions
  - Average search cost is O(1) + #overflow pages

Static Hashing: Example

MovieID	Name	Year	Director
20	Inception	2010	Christopher Nolan
15	Avatar	2009	Jim Cameron
52	Gravity	2013	Alfonso Cuaron
74	Blue Jasmine	2013	Woody Allen

Hash function:
a = 1, b = 0
h(k) = k MOD 3

N = 3



Say, AltRecord and only one record fits per page!

15 MOD 3 = 0

74 MOD 3 = 2

Static Hashing: Insert and Delete

- Insert:
  - Equality search; find space on primary bucket page
  - If not enough space, add overflow page
- Delete:
  - Equality search; delete record
  - If overflow page becomes empty, remove it
  - Primary bucket pages are never removed!

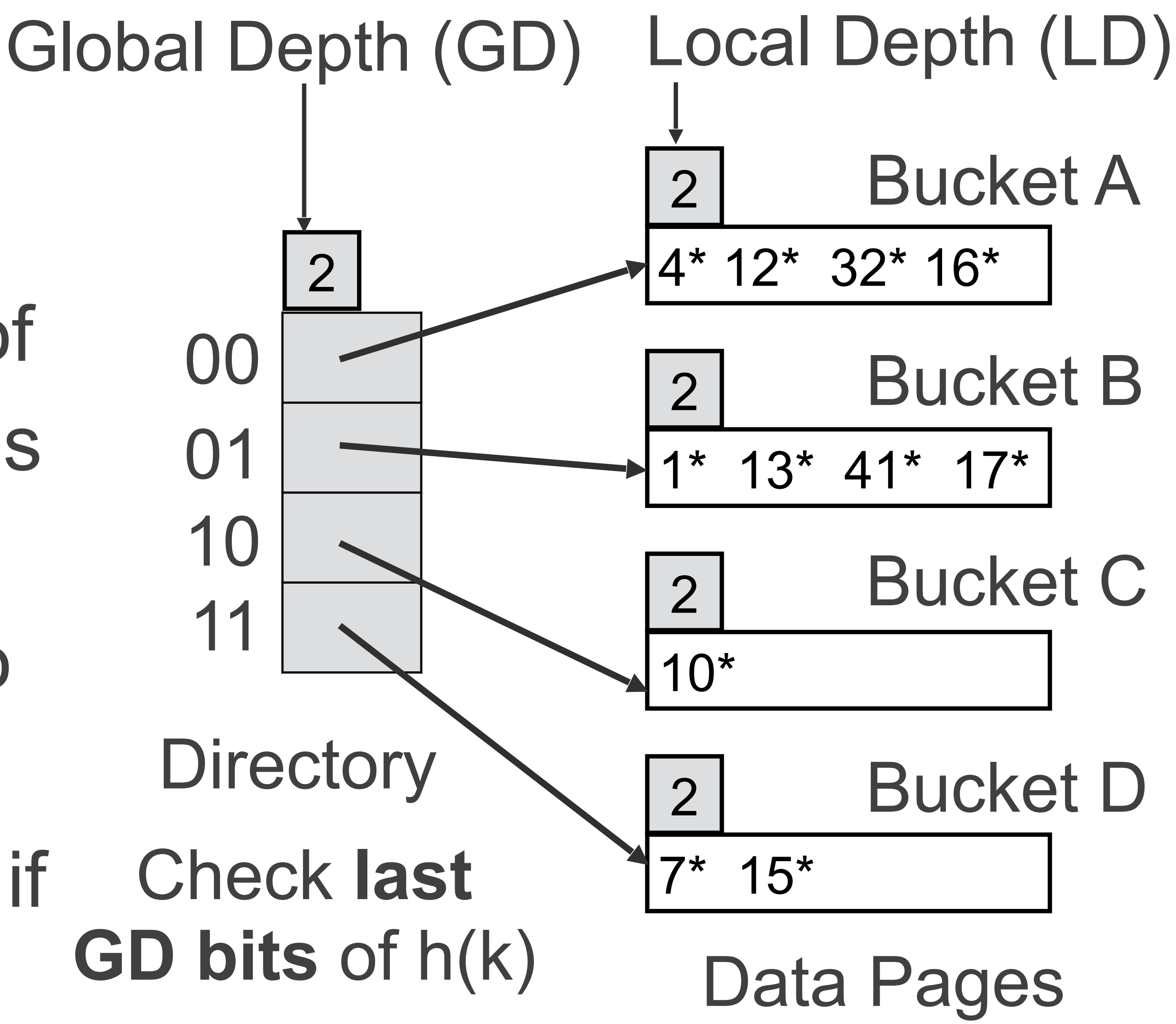
Static Hashing: Issues

- Since N is fixed, #overflow pages might grow and degrade search cost; deletes waste a lot of space
- Full reorg. is expensive and could block query proc.
- Skew in hashing:
  - Could be due to “bad” hash function that does not “spread” values—but this issue is well-studied/solved
  - Could be due to skew in the data (duplicates); this could cause more overflow pages—difficult to resolve

Extendible (dynamic) hashing helps resolve first two issues

Extendible Hashing

- **Idea:** Instead of hashing directly to data pages, maintain a dynamic directory of pointers to data pages
- Directory can grow and shrink; chosen to double/halve
- Search I/O cost: 1 (2 if direc. not cached)



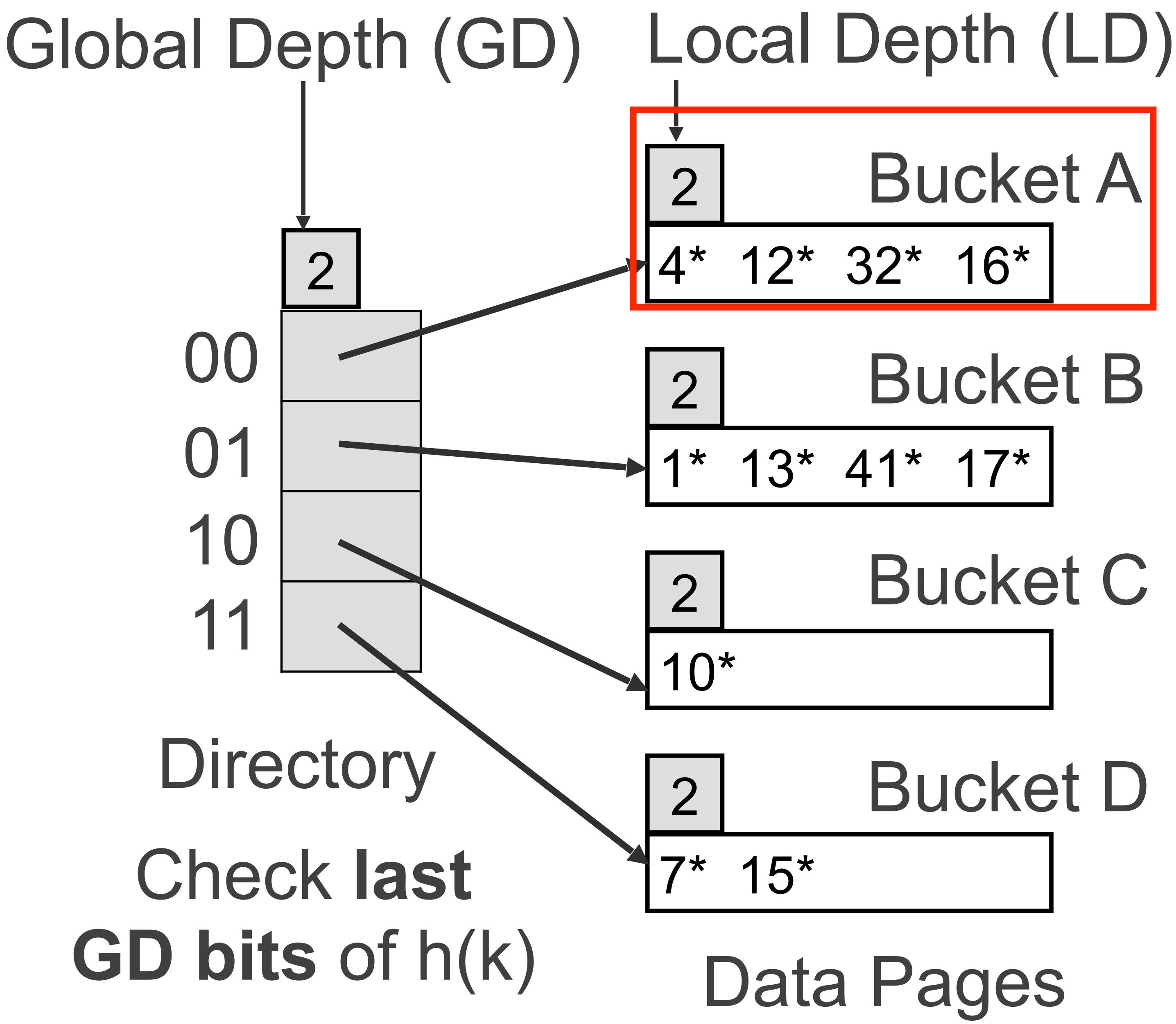
Check **last GD bits** of h(k)

Example: Search 17* 10001 Search 42* …10

Extendible Hashing: Insert

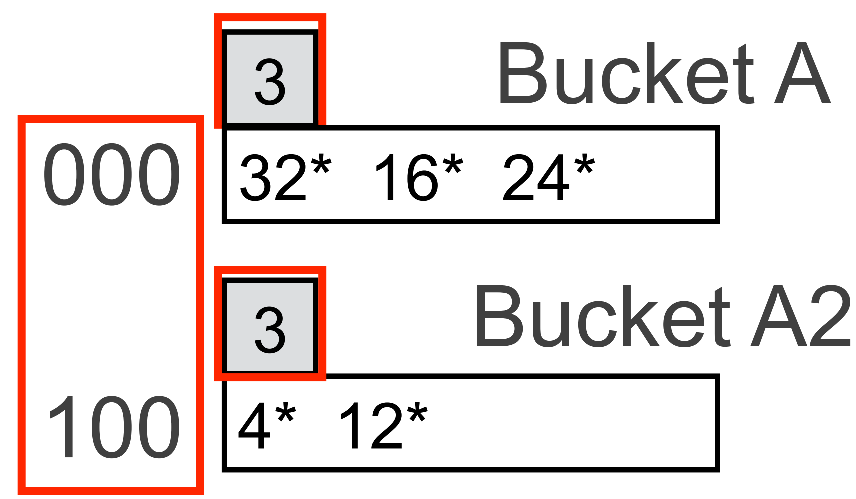
- Search for k; if data page has space, add record; done
- If data page has no more space:
  - If LD < GD, create **Split Image** of bucket; LD++ for both bucket and split image; insert record properly
  - If LD == GD, create Split Image; LD++ for both buckets; insert record; but also, **double** the directory and GD++! Duplicate other direc. pointers properly
- Direc. typically grows in spurts

Extendible Hashing: Insert

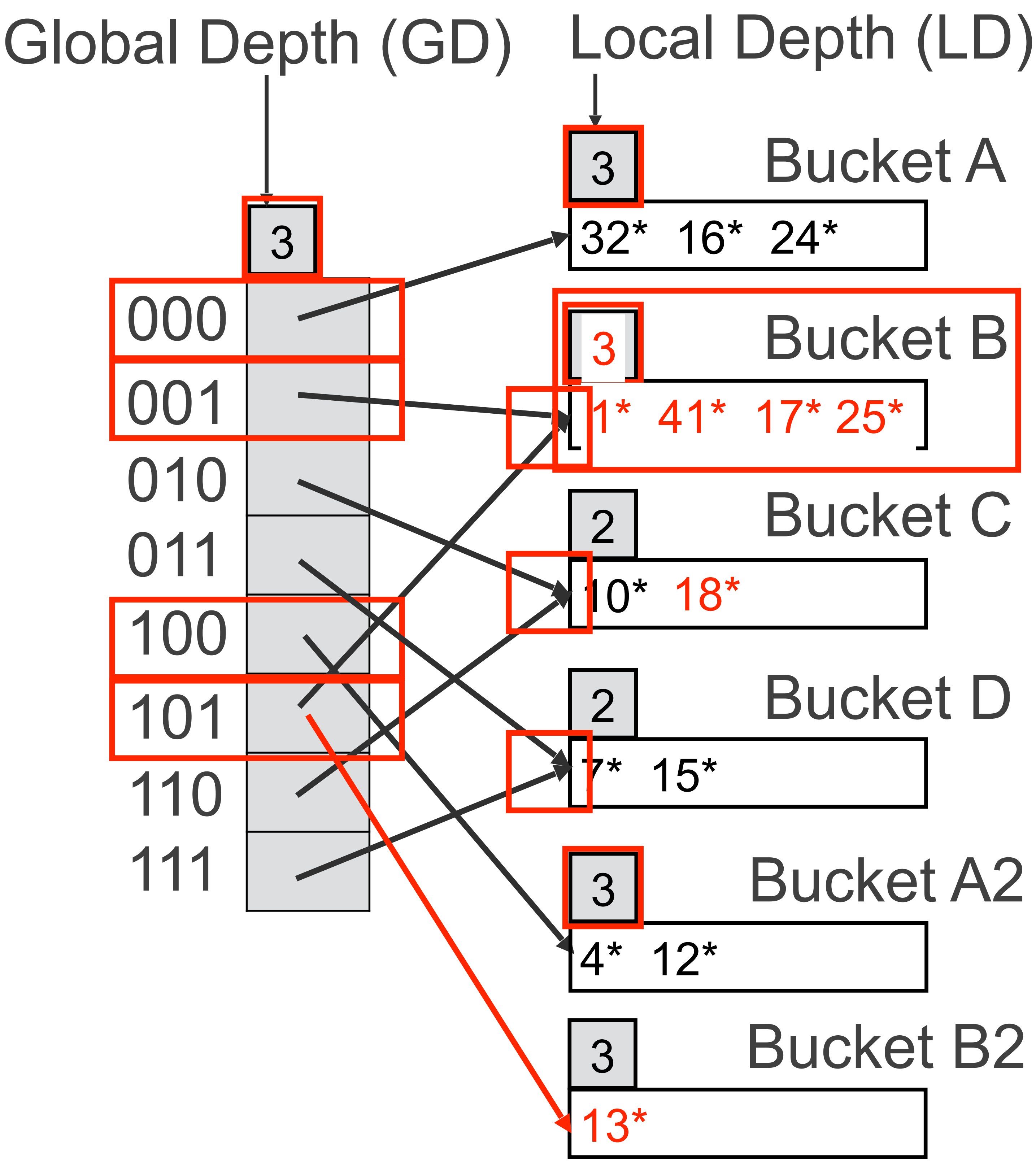


Example:
Insert 24* …00

No space in bucket A
Need to split and LD++
Since LD was = GD,
GD++ and direc. doubles



Extendible Hashing: Insert



Example: Insert 24*

Example: Insert 18*

…010

Example: Insert 25*

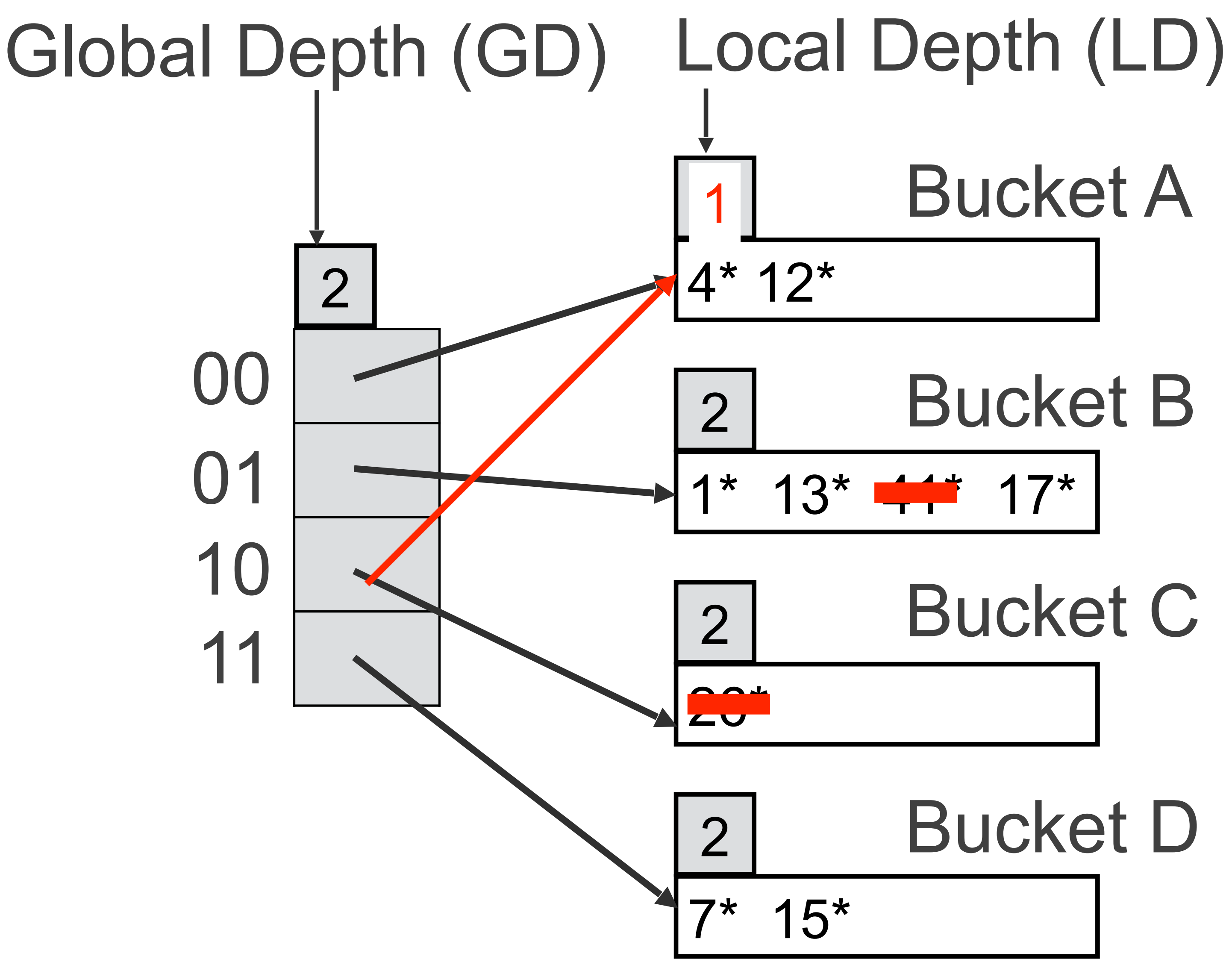
…001

Need to split bucket B!
Since LD < GD, direc. <u>does not double</u> this time
Only LD++ on old bucket and modify direc. ptrs

Extendible Hashing: Delete

- Search for k; delete record on data page
- If data page becomes empty, we can **merge** with another data page with same last LD-1 bits (its "historical split image") and do LD--; update direc. ptrs
  - Advanced (optional): In rare cases, hist. split image may have split further; then just let this page sit empty for now
- If all split images get merged back and if all buckets end up with LD < GD, **shrink** direc. by half; GD--
- Never does a bucket's LD become > GD!

Extendible Hashing: Delete



Example: Delete 41*

…01

Example: Delete 26*

…10

Bucket C is now empty
Can merge with A; LD--

Q: *Why did we pick A to merge with?*

In practice, deletes and thus, bucket merges are rare

So, directory shrinking is even more uncommon

Static Hashing vs Extendible Hashing

***Q:** Why not let N in static hashing grow and shrink too?*

- Extendible hash direc. size is typically much smaller than data pages; with static hash, reorg. of all data pages is far more expensive
- Hashing skew is a common issue for both; in static hash, this could lead to large # overflow pages; in extendible hash, this could blow up direc. size (this is OK)
- If too many data entries share search key (duplicates), overflow pages needed for extendible hash too!

Indexing: Outline

- **Overview and Terminology**
- **B+ Tree Index**
- **Hash Index**