CS-300: Data-Intensive Systems

Tree-Structured Indexing

(Chapter 14.1-14.4)

Prof. Anastasia Ailamaki, Prof. Sanidhya Kashyap

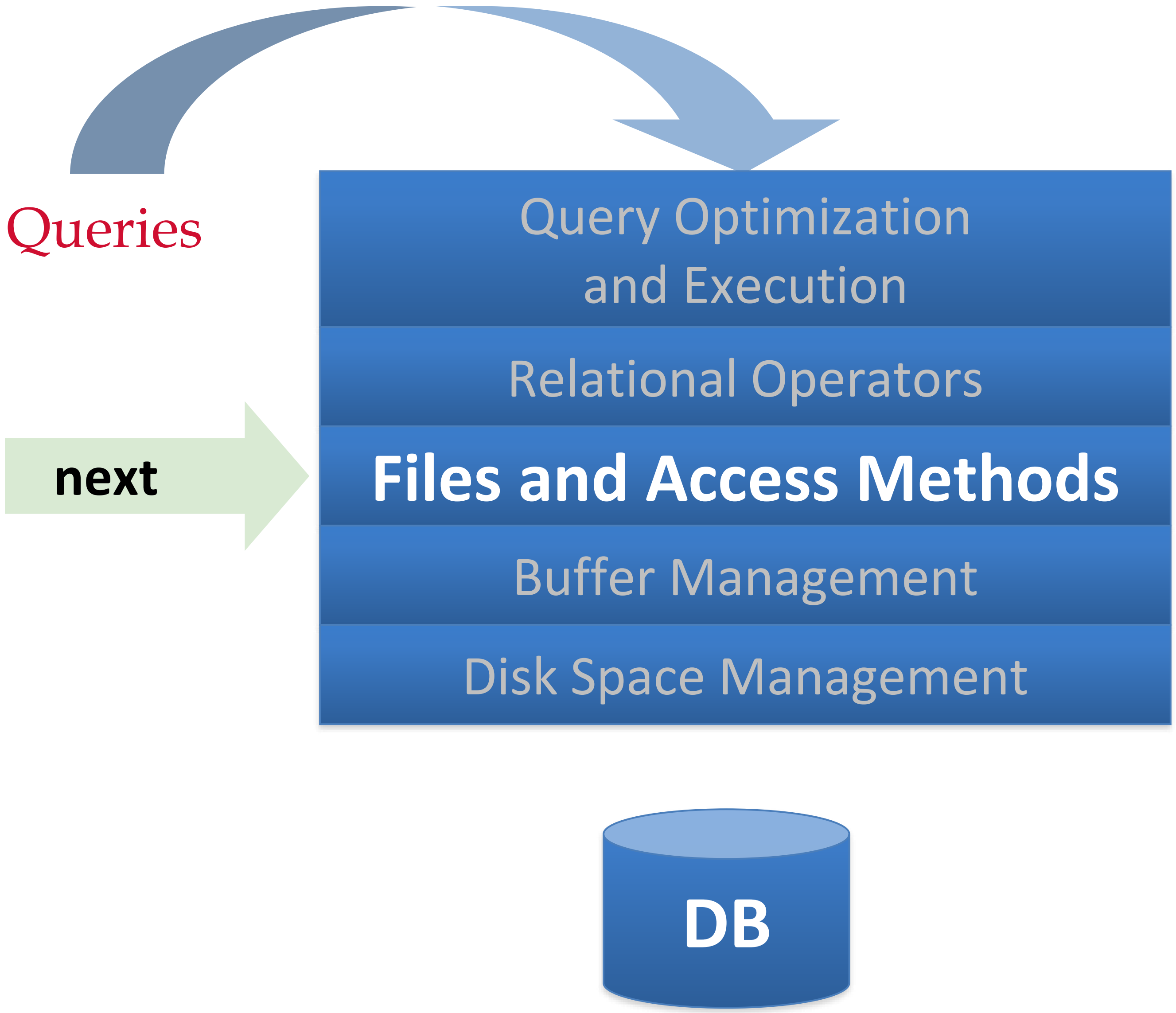
EPFL

DBMS bigger picture

Support DBMS execution engine to read/write data from pages!

Two types of data structures:

1. **Trees (ordered)**
2. Hash tables (unordered)



Today's focus

- B^+ Tree overview
- Operations on B^+ Tree

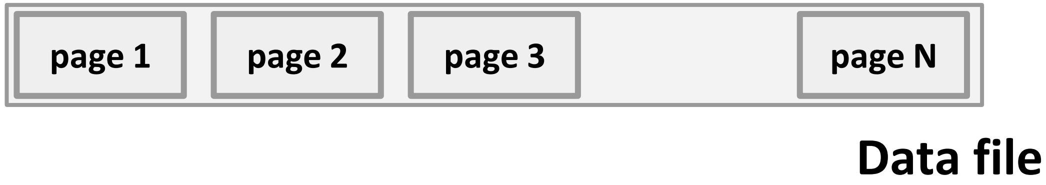
Index structures

- Recall: 3 alternatives for data entries k*:
  - Data record with key value **k**
  - <**k**, rid of data record with search key value **k**>
  - <**k**, list of rids of data records with search key **k**>
- Data is often indexed:
  - Speeds up lookup
  - Mandatory for primary keys
  - Useful for selective queries
- Choice is orthogonal to the *indexing technique* used to locate data entries k*
- Tree-structured indexing techniques support both range searches and equality searches

Example: range search

Let’s run a query: “Find all students with gpa > 3.0”

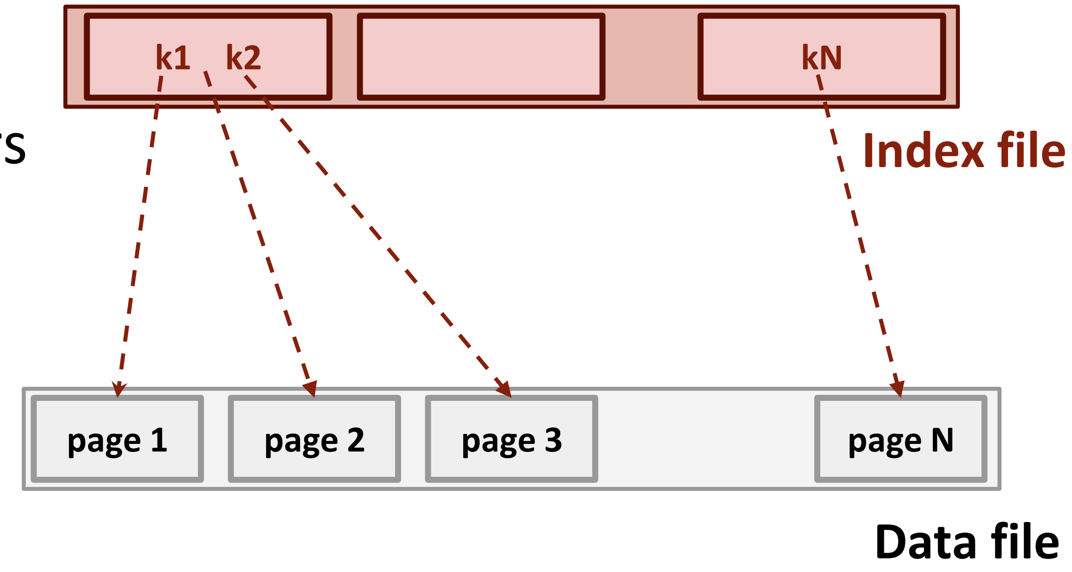
- If data is in a sorted file, do binary search to find first such student, then scan to find others
- Cost of maintaining sorted file + performing binary search in a database can be quite high!



Example: range search

Let's run a query: "Find all students with gpa > 3.0"

- If data is in a sorted file, do binary search to find first such student, then scan to find others
- Cost of maintaining sorted file + performing binary search in a database can be quite high!



Simple idea: Create an **'index'** file

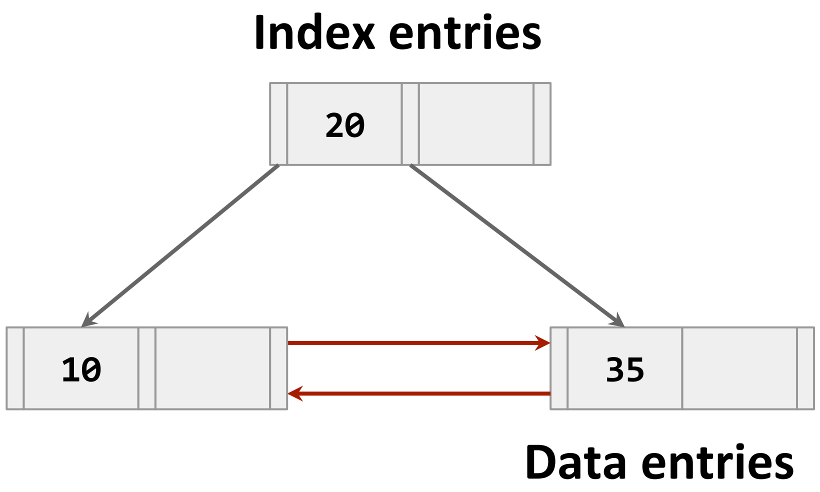
- Can do binary search on (smaller) index file

Basic idea of **B$^+$ Tree!**

B^+ Trees: The most widely-used index structure

- B-trees (including variants) are the preferred data structure for external storage

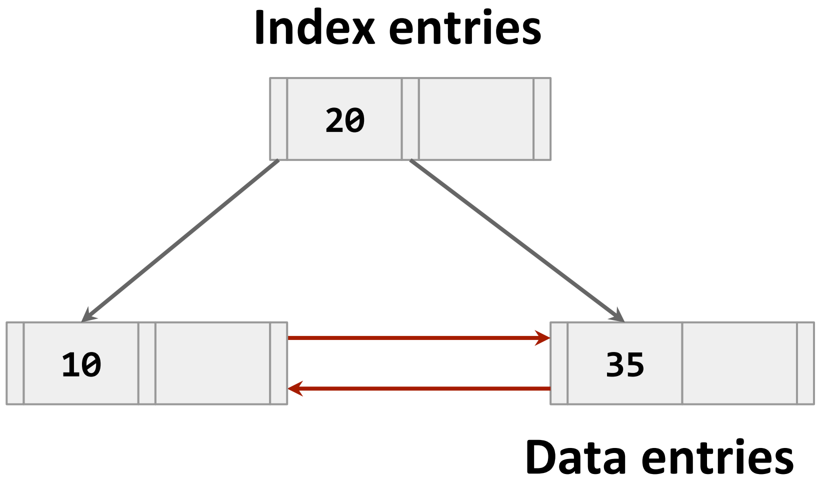
- Class of balanced tree data structures:
  - **B-Tree**
  - **B^+ Tree**
  - **B^* Tree**
  - **B^{link} Tree**
  - **B^{ε} Tree**



What is a B^+ Tree?

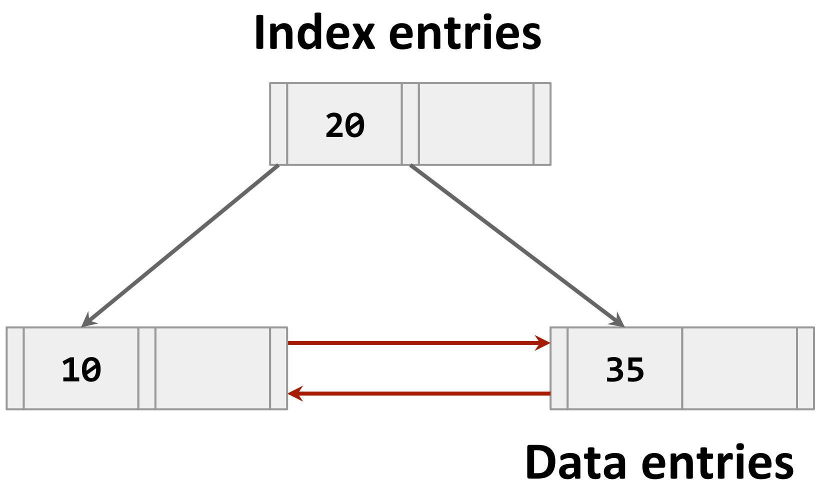
- A self-balancing (*height balanced*), ordered tree data structure that allows searches, sequential access, insertions, and deletions in $\mathbf{O(log_F N)}$
    - N: Number of leaf nodes
    - F: Fanout

    - Generalization of a binary search tree, since a node can have more than one children
    - Optimized for systems that read and write large blocks of data

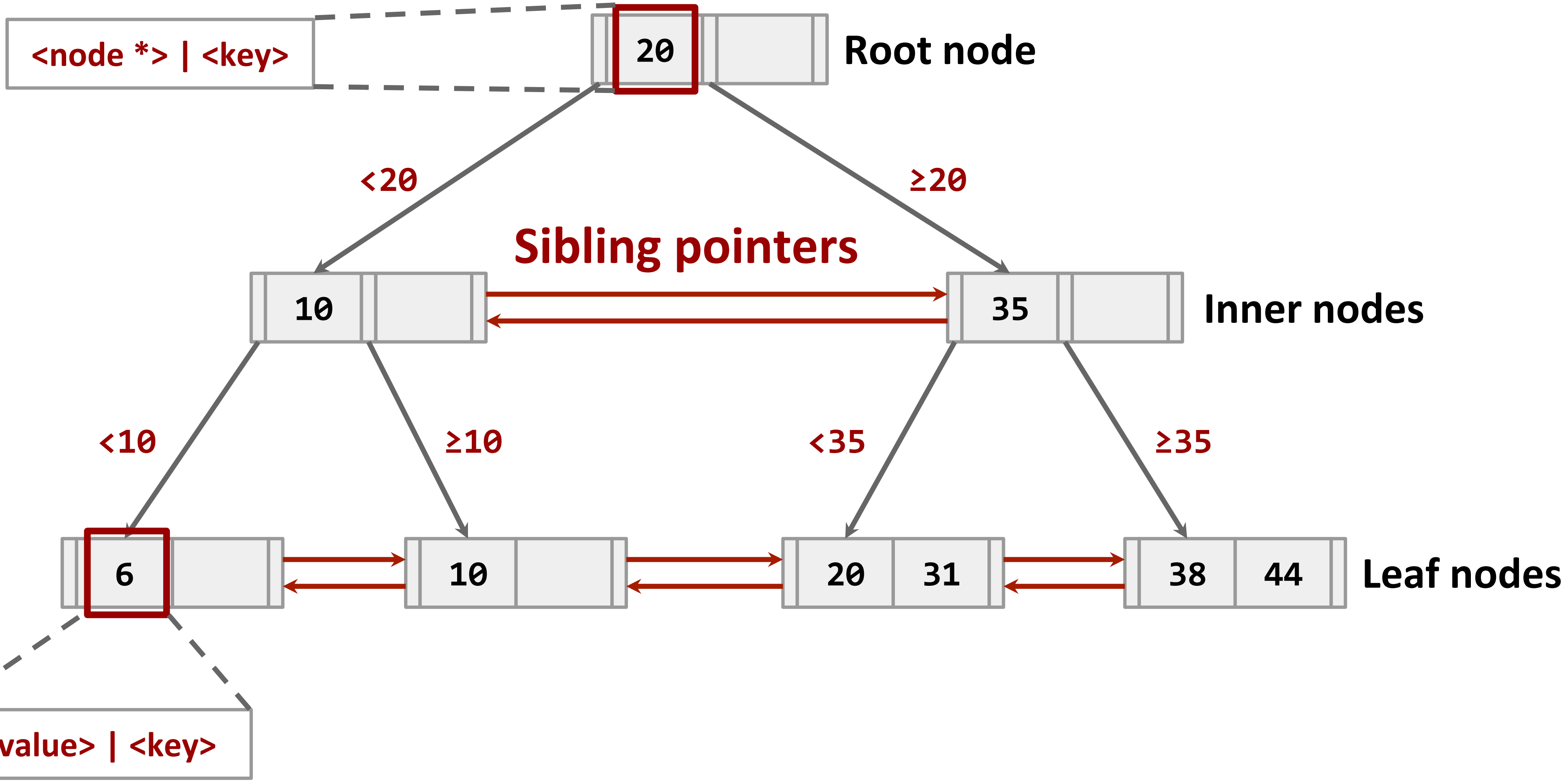


B^+ Tree properties

- A B^+ Tree is an ***d***-way search tree with the following properties:
  - Perfectly balanced
    - Every leaf node is at the same depth in the tree
  - Every node other than root is at least half-full
    - **d ≤ #keys ≤ 2d**
      - *d is also called order of the tree*
  - Nodes are of three types: *root*, *inner,* and *leaf*
  - Every inner node with **k** keys has **k+1** non-null children

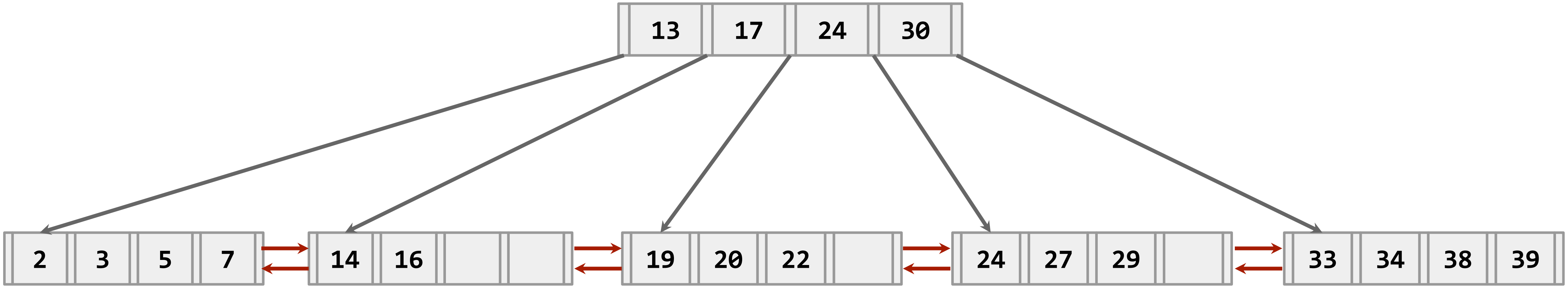


B^+ Tree example



B⁺ Tree another example

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



Based on the search for 15, we <u>know</u> it is not in the tree!

B^+ Tree: Lookup/search operation

Looks for a search key v within the tree:

- Start by setting ***C*** to the root node
- While the current node (***C***) is not a leaf node:
  a. Identify the smallest index i where v is less than or equal to the key of i.
  b. If no such index exists, set ***C*** to the last non-null pointer in ***C***.
  c. If v equal to the key of i, move to the right child pointer
  d. Otherwise, move to the left child pointer
- If the leaf node contains an entry with key equal to v return it
- Otherwise, return null → no record with key v exists

Lookup can return the concrete entry or just the position of the appropriate leaf page

https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

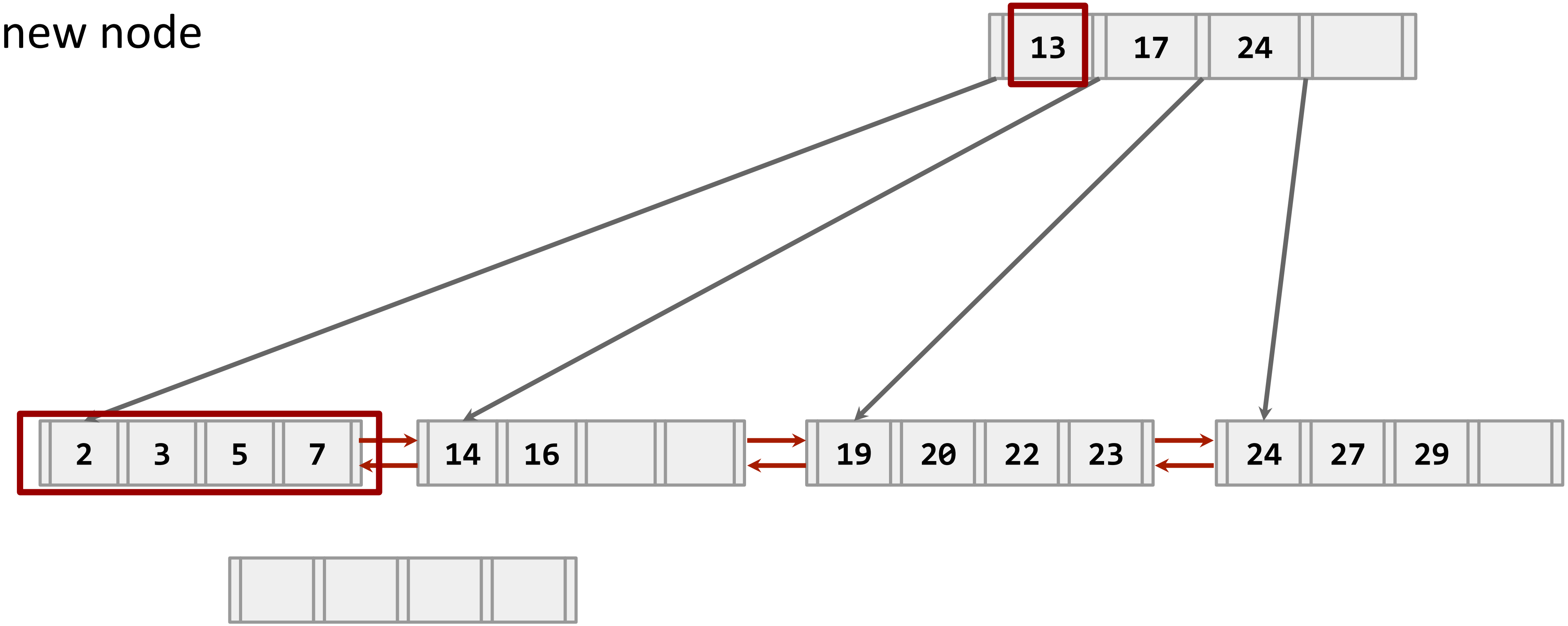
B^+ Tree: Insert operation

- Find the correct leaf node ***L***
- Insert data entry into ***L*** in sorted order
  - If *L* has enough space, done!
  - Else, split ***L*** *into* ***L*** *and a new node* ***L2***
    - Redistribute entries evenly, copy up middle key
    - Insert index entry pointing to ***L2*** into parent of ***L***
- This can happen recursively
  - To split index node, redistribute entries evenly, but push up middle key
- Splits "grow" tree; root split increases height
  - Tree growth: gets wider or one level taller at top

https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

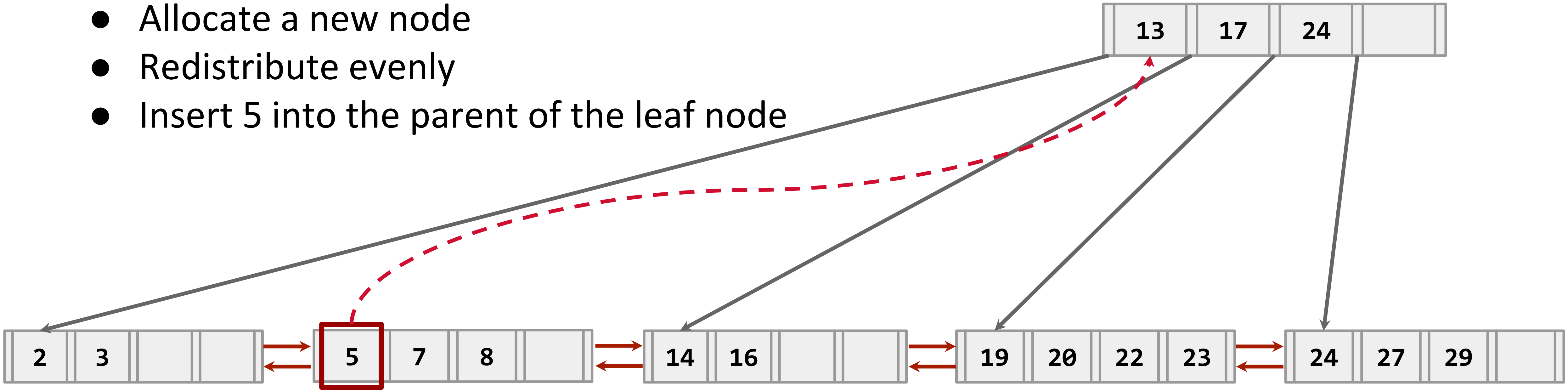
B⁺ Tree example: Insert 8

- Node for 8 will be present in the first leaf node
- Node is already full
- Allocate a new node



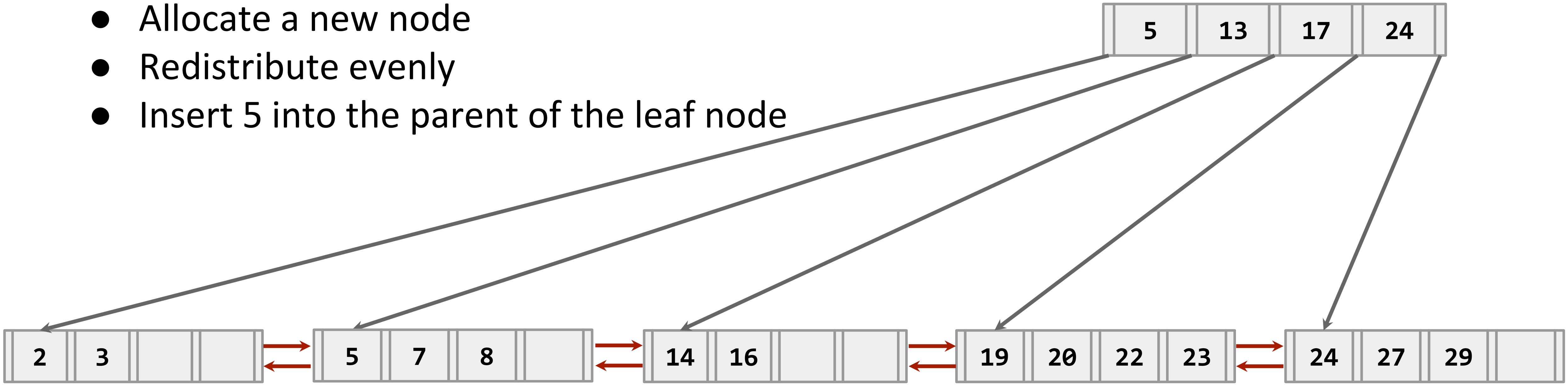
B+ Tree example: Insert 8

- Node for 8 will be present in the first leaf node
- Node is already full
- Allocate a new node
- Redistribute evenly
- Insert 5 into the parent of the leaf node



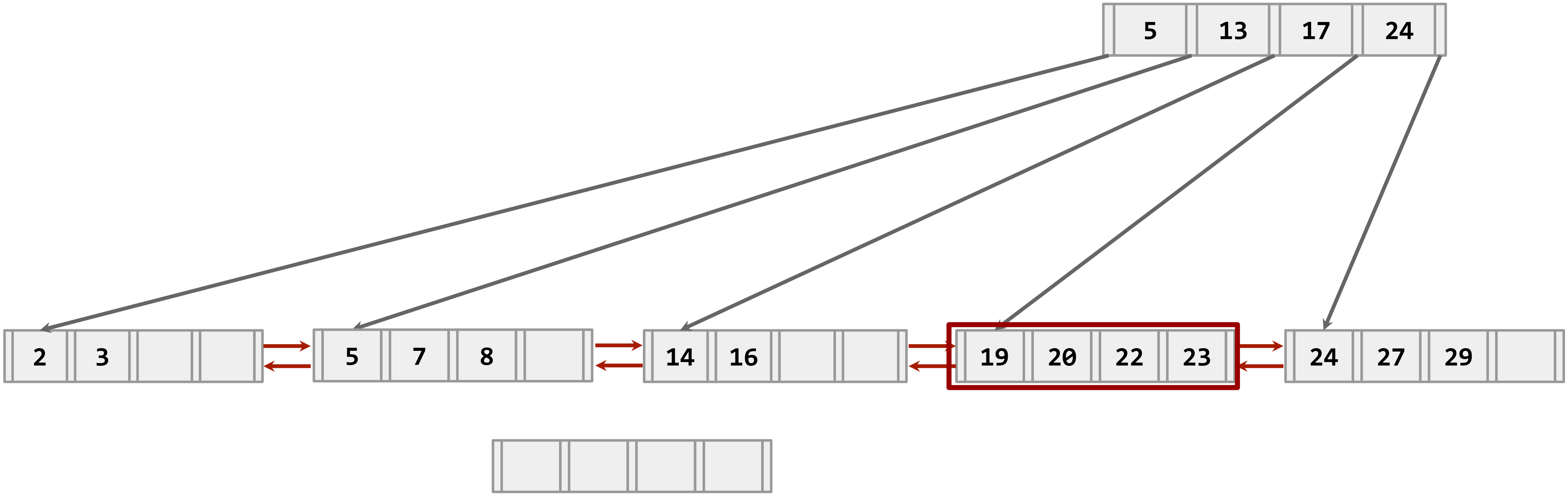
B^{+} Tree example: Insert 8

- Node for 8 will be present in the first leaf node
- Node is already full
- Allocate a new node
- Redistribute evenly
- Insert 5 into the parent of the leaf node



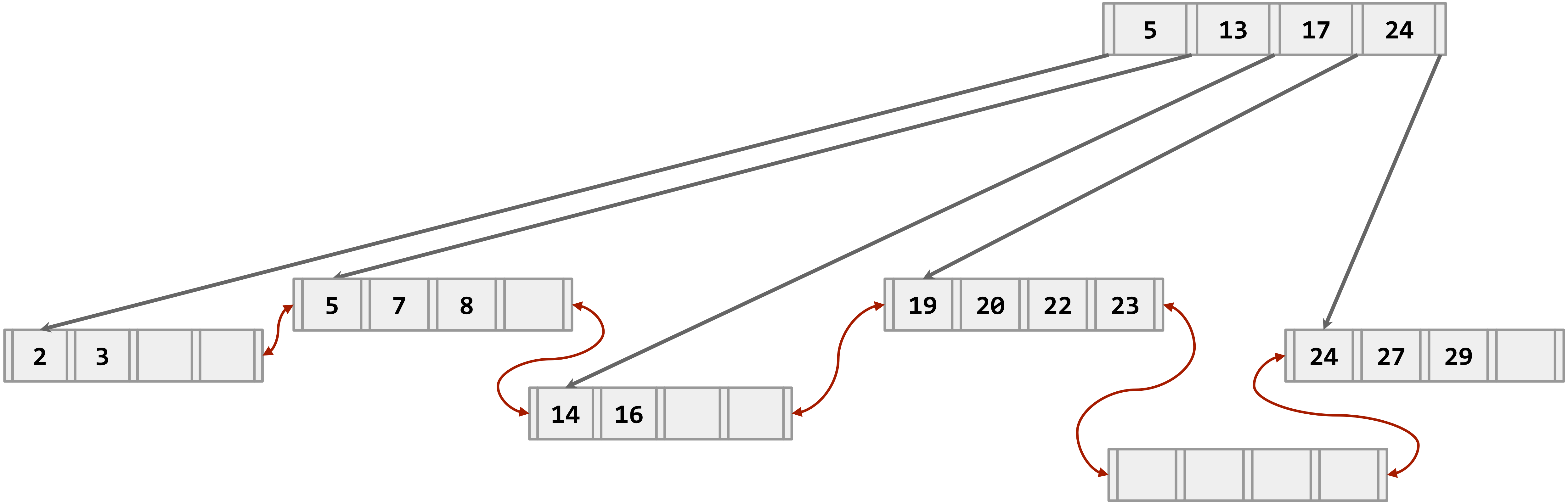
B^+ Tree example: Insert 21 now

- Allocate a new node as the leaf node is full
- Split the node with values 19–23 evenly



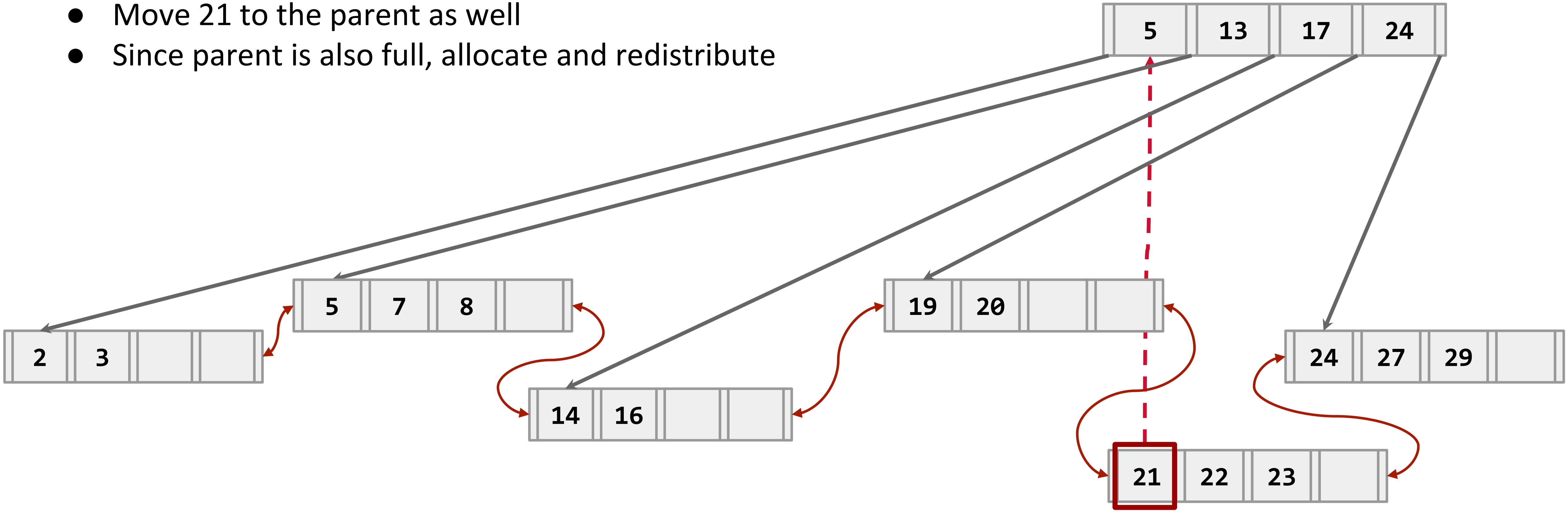
B+ Tree example: Insert 21 now

- Allocate a new node as the leaf node is full
- Split the node with values 19–23 evenly



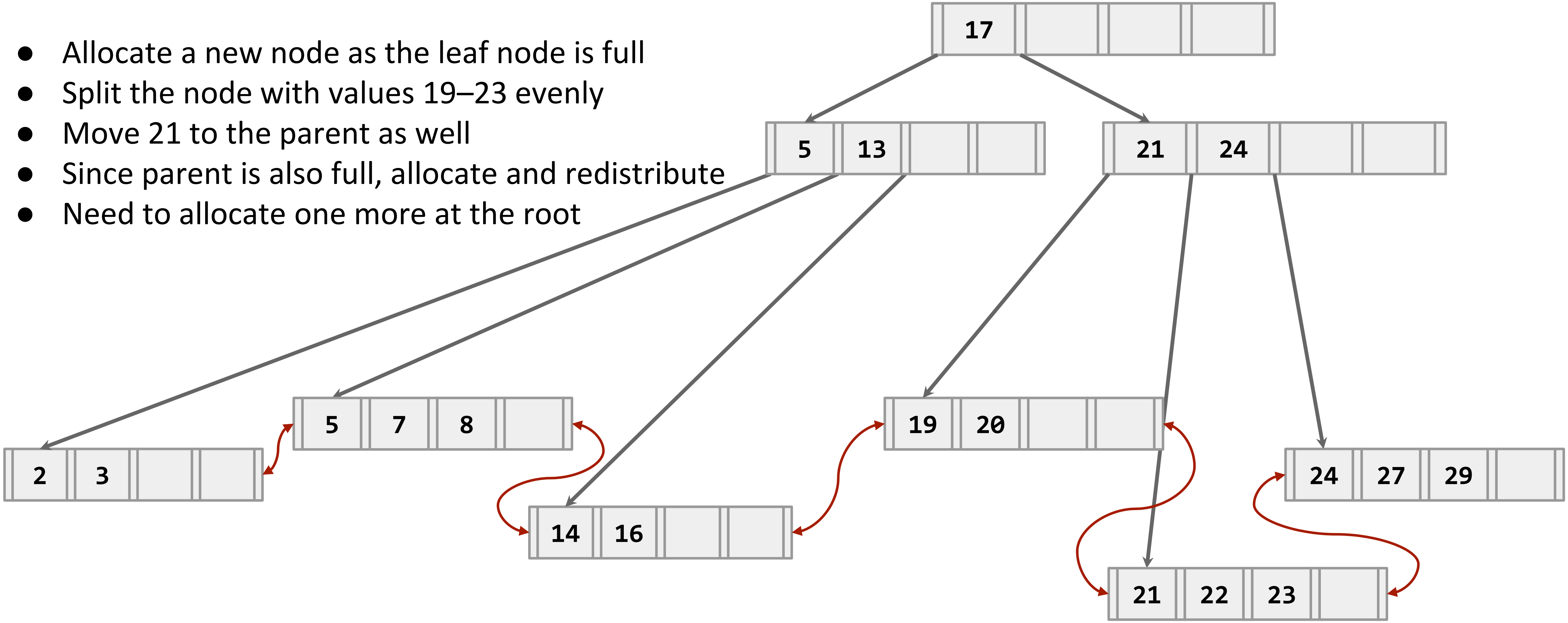
B$^+$ Tree example: Insert 21 now

- Allocate a new node as the leaf node is full
- Split the node with values 19–23 evenly
- Move 21 to the parent as well
- Since parent is also full, allocate and redistribute



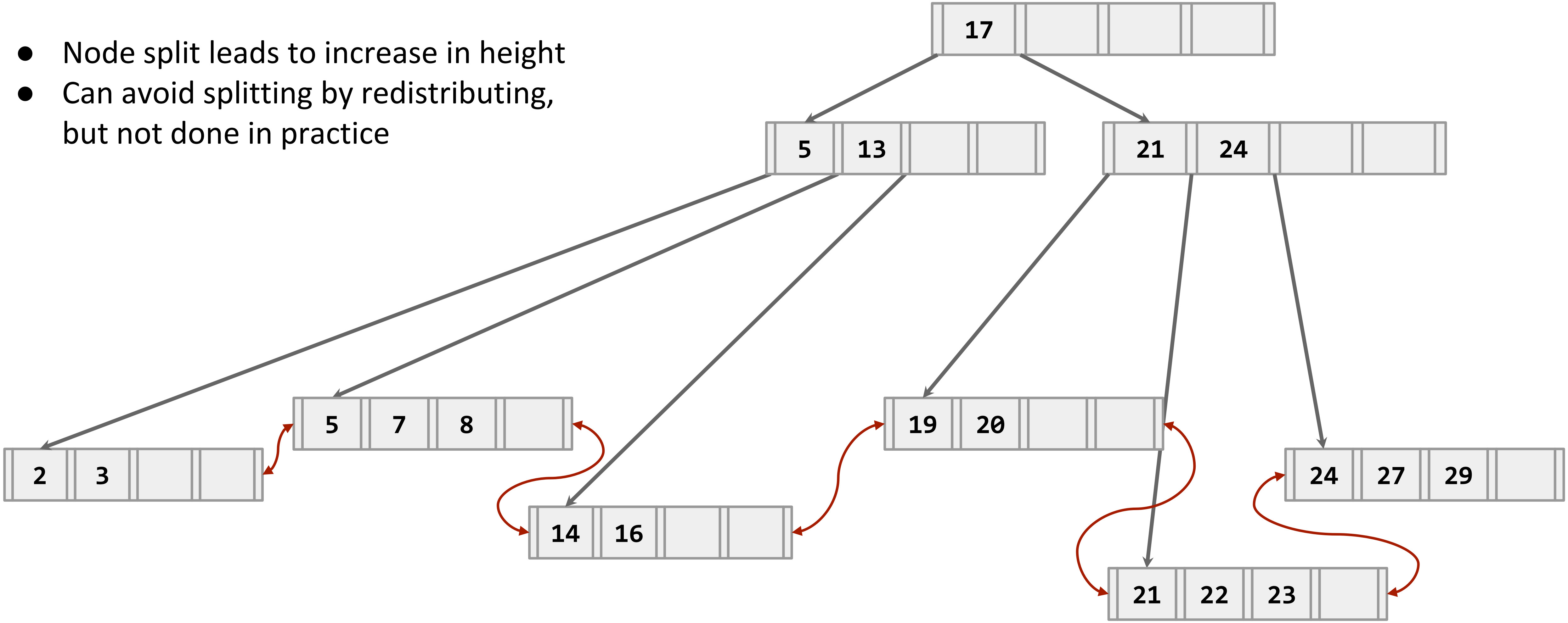
B$^+$ Tree example: Insert 21 now

- Allocate a new node as the leaf node is full
- Split the node with values 19–23 evenly
- Move 21 to the parent as well
- Since parent is also full, allocate and redistribute
- Need to allocate one more at the root



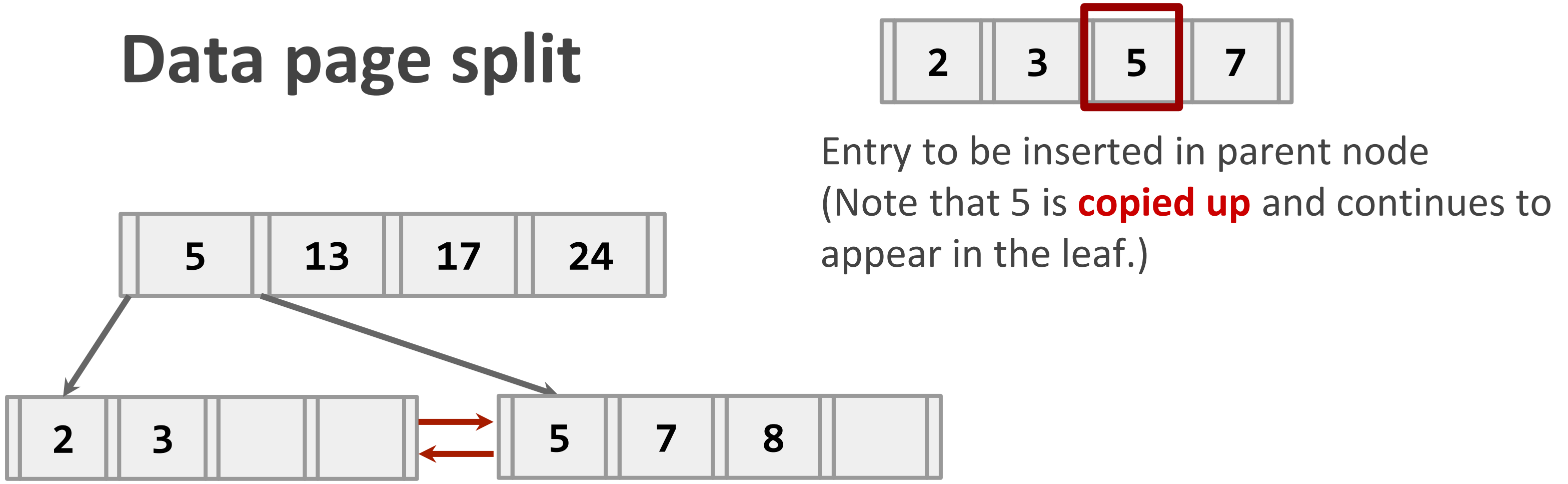
B+ Tree root splitting

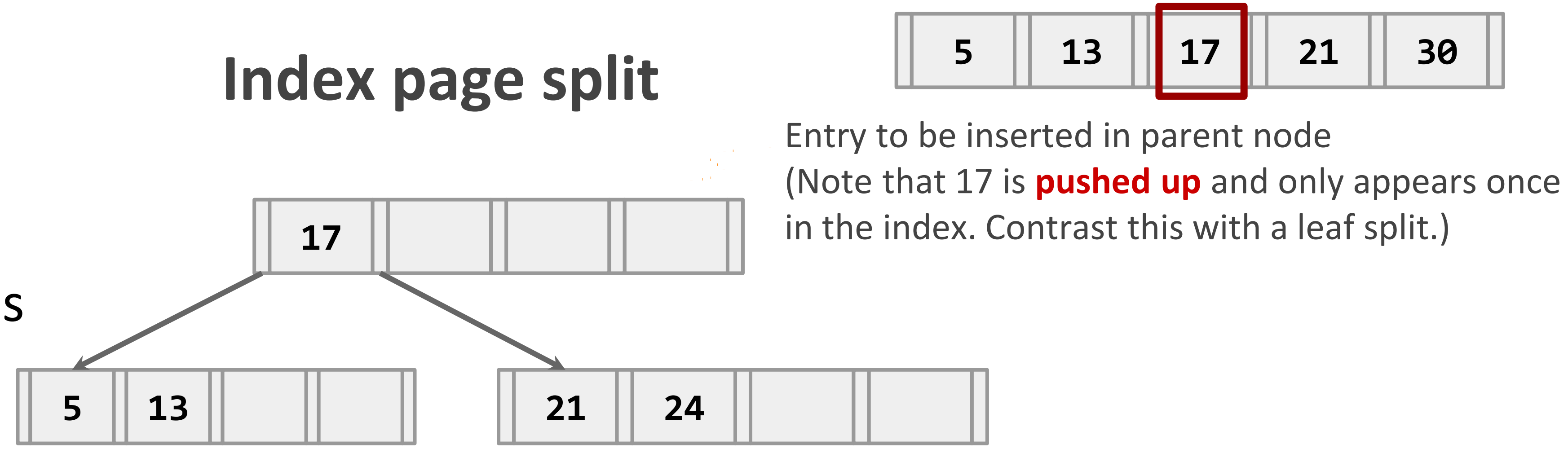
- Node split leads to increase in height
- Can avoid splitting by redistributing, but not done in practice



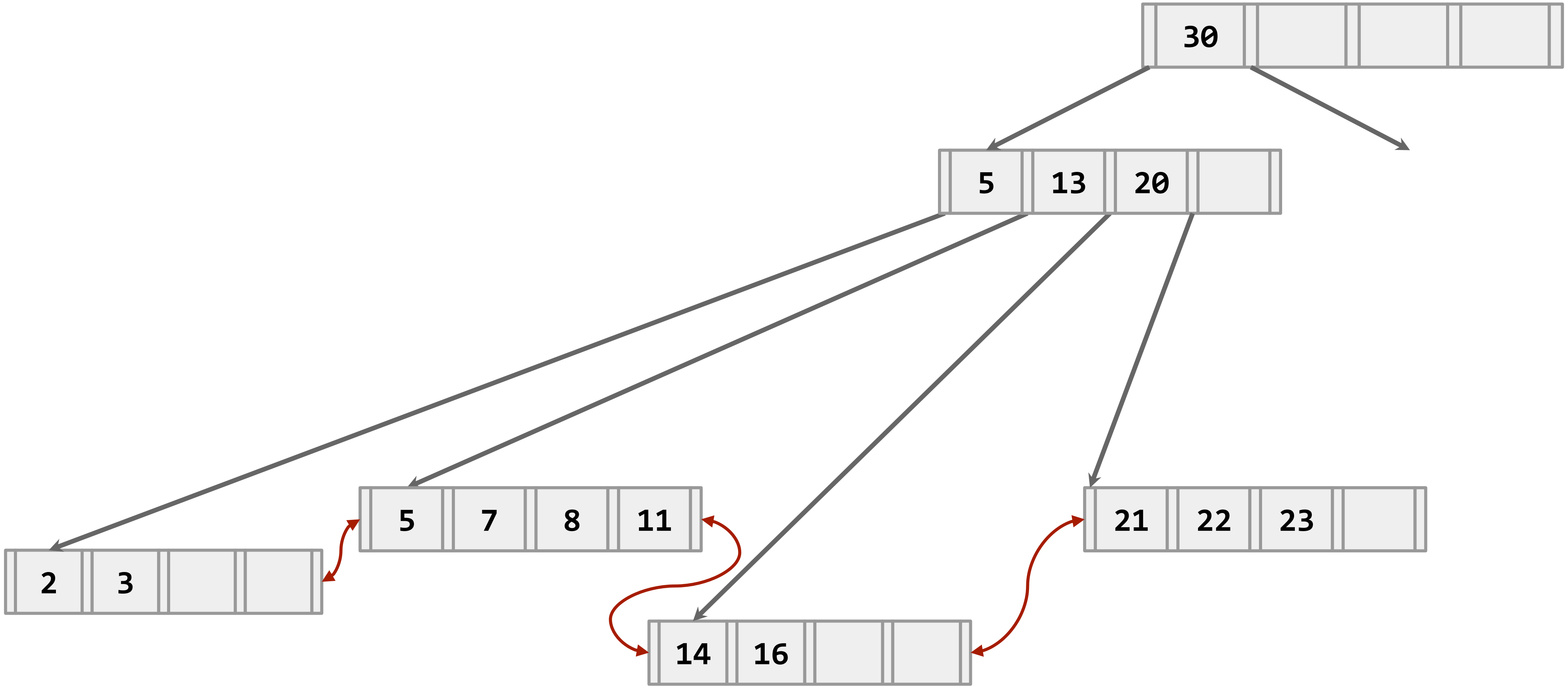
Data vs index page split

- Observe how minimum occupancy is guaranteed in both leaf and index page splits
- Note difference between copy-up and push-up; be sure you understand the reasons for this.

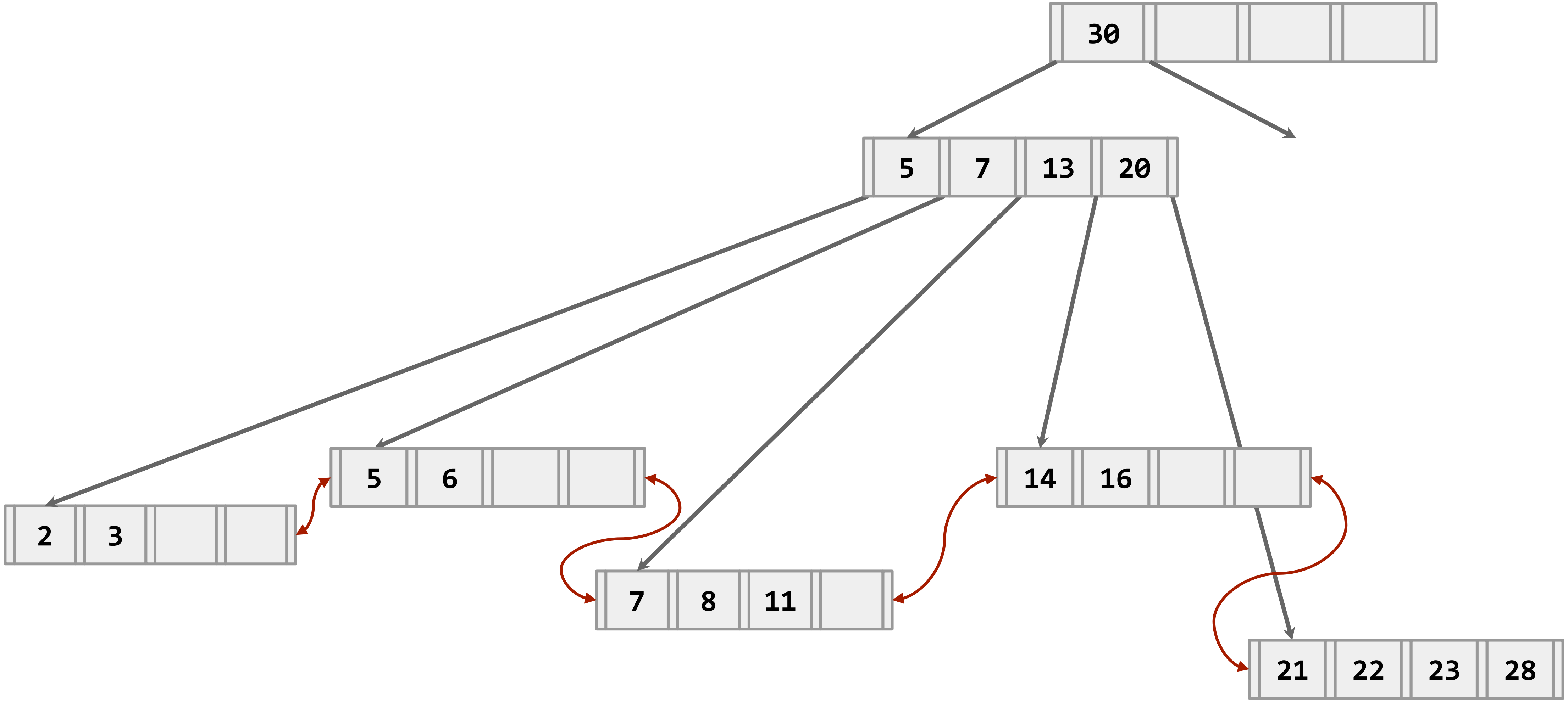




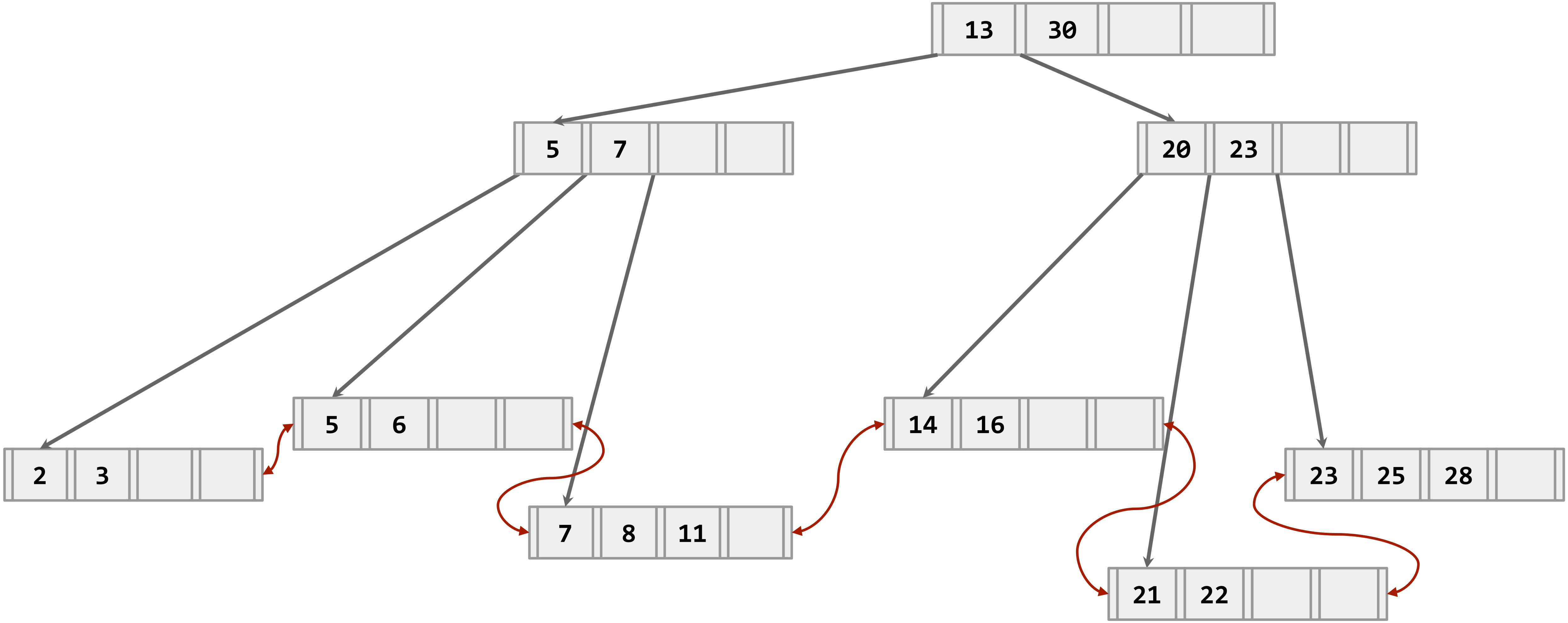
B+ Tree: Before inserting 28, 6, and 25



B$^+$ Tree: After inserting 28, 6



B$^+$ Tree: After inserting 28, 6, 25



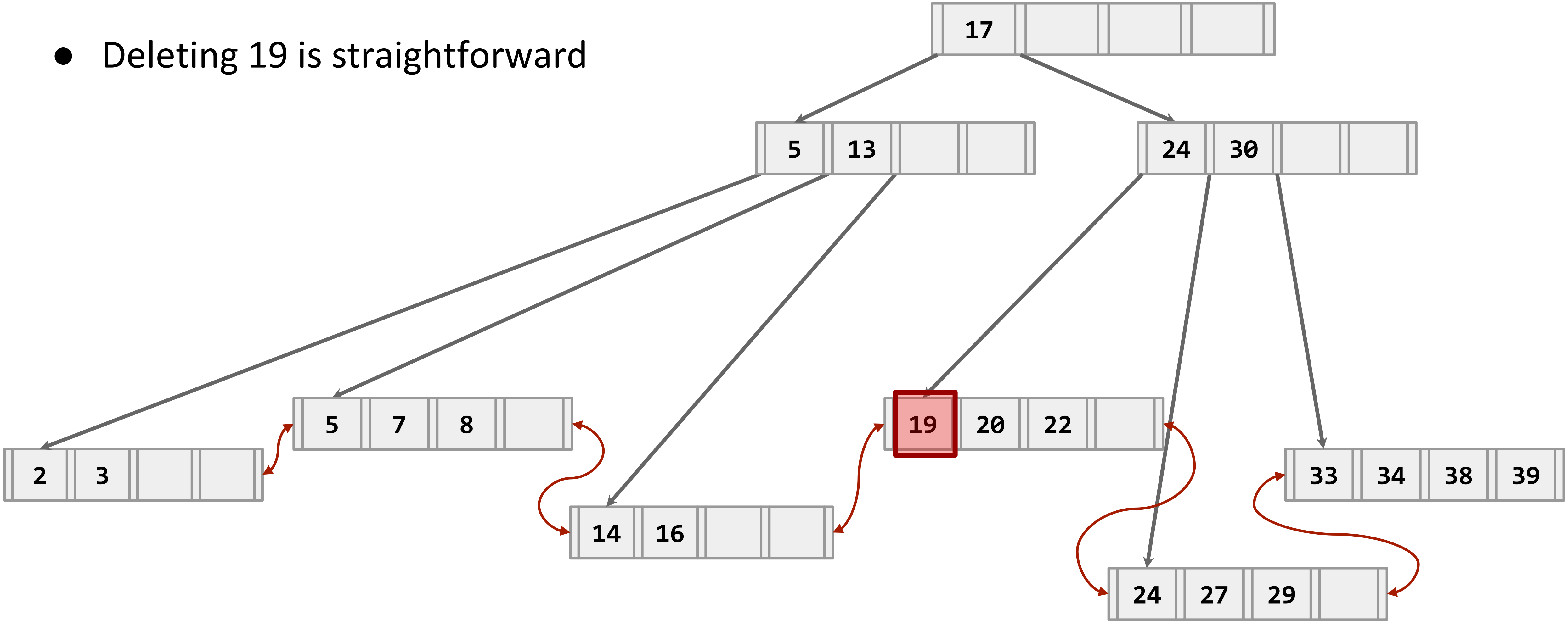
B$^+$ Tree: Delete operation

- Start at root, find leaf ***L*** where entry belongs
- Remove the entry
  - If L is at least half-full, done!
  - If L has only **d-1** entries,
    - Try to redistribute, borrowing from ***sibling*** (adjacent node with same parent as *L*)
    - If redistribution fails, merge *L* and sibling
- On merge, delete entry from parent of *L*
  - Either the entry pointing to *L,* or the one pointing to sibling
- Propagate merge to root, as needed
  - Height decreases

https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

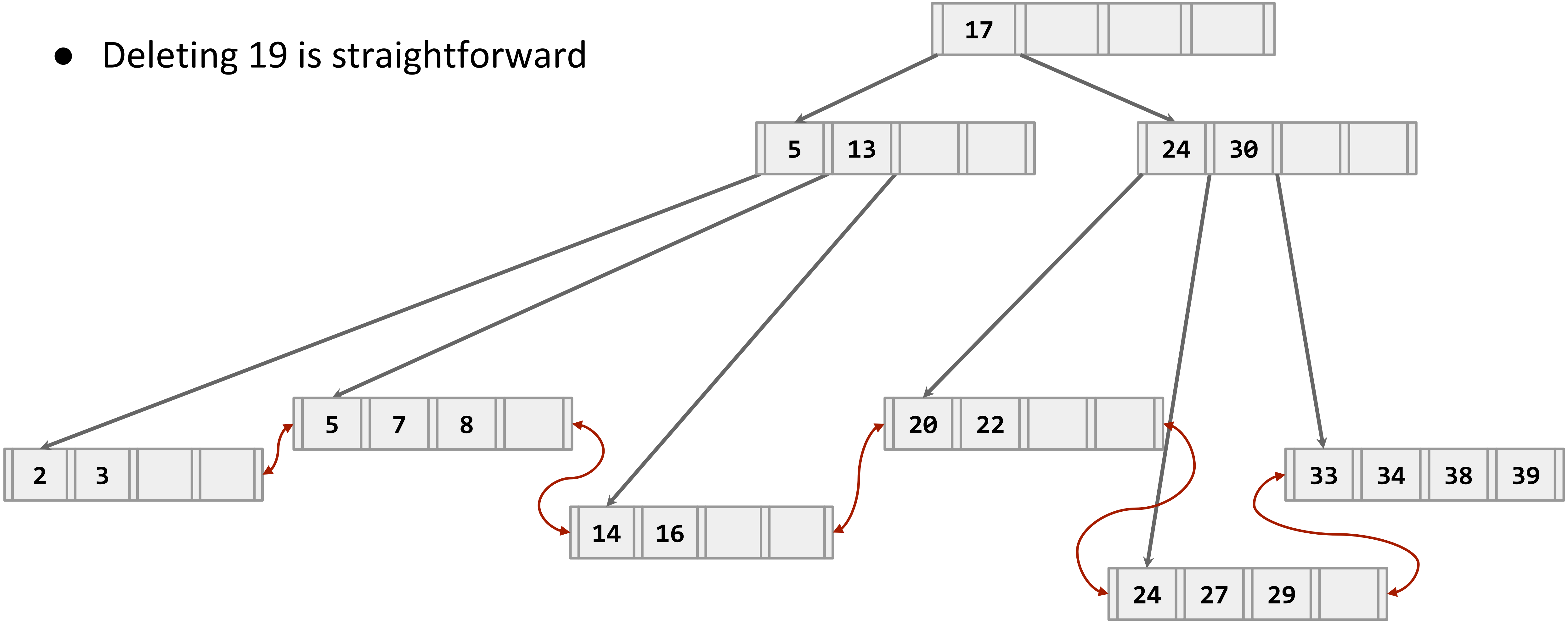
B+ Tree example: Delete 19 and 20

- Deleting 19 is straightforward



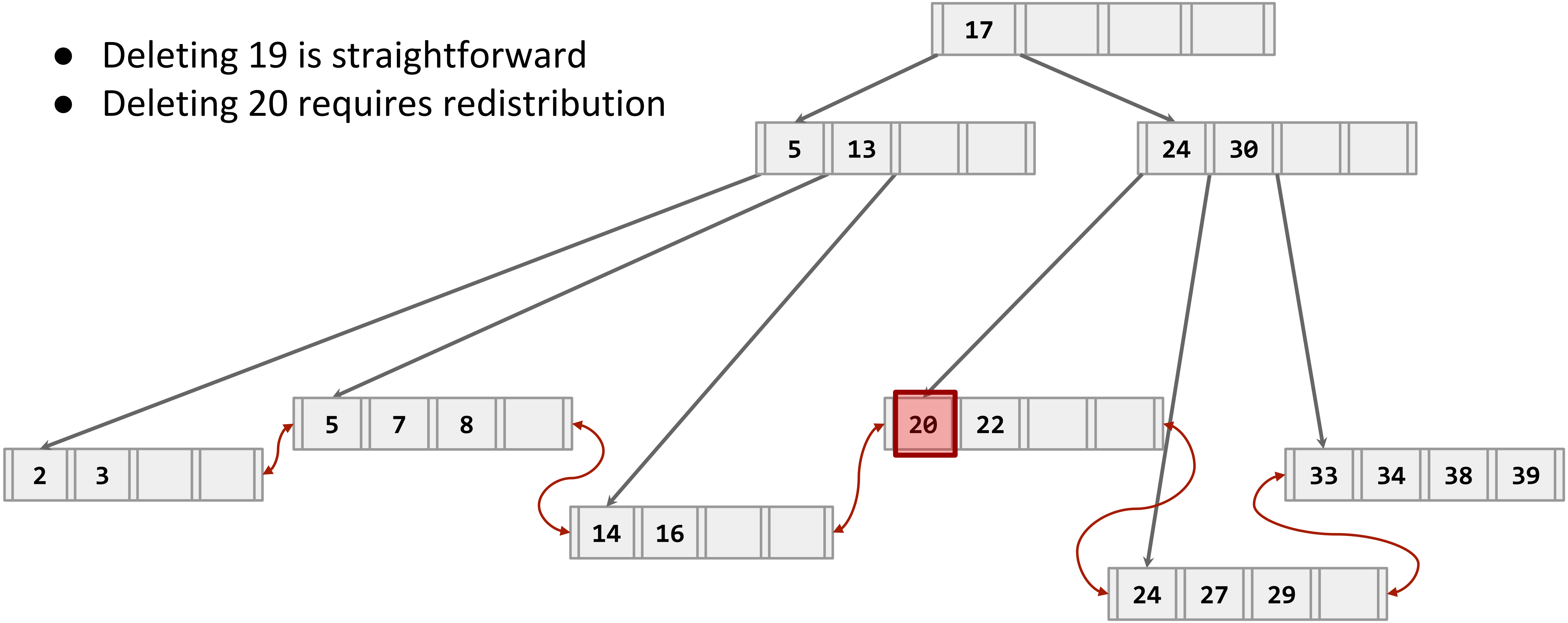
B$^+$ Tree example: Delete 19 and 20

- Deleting 19 is straightforward



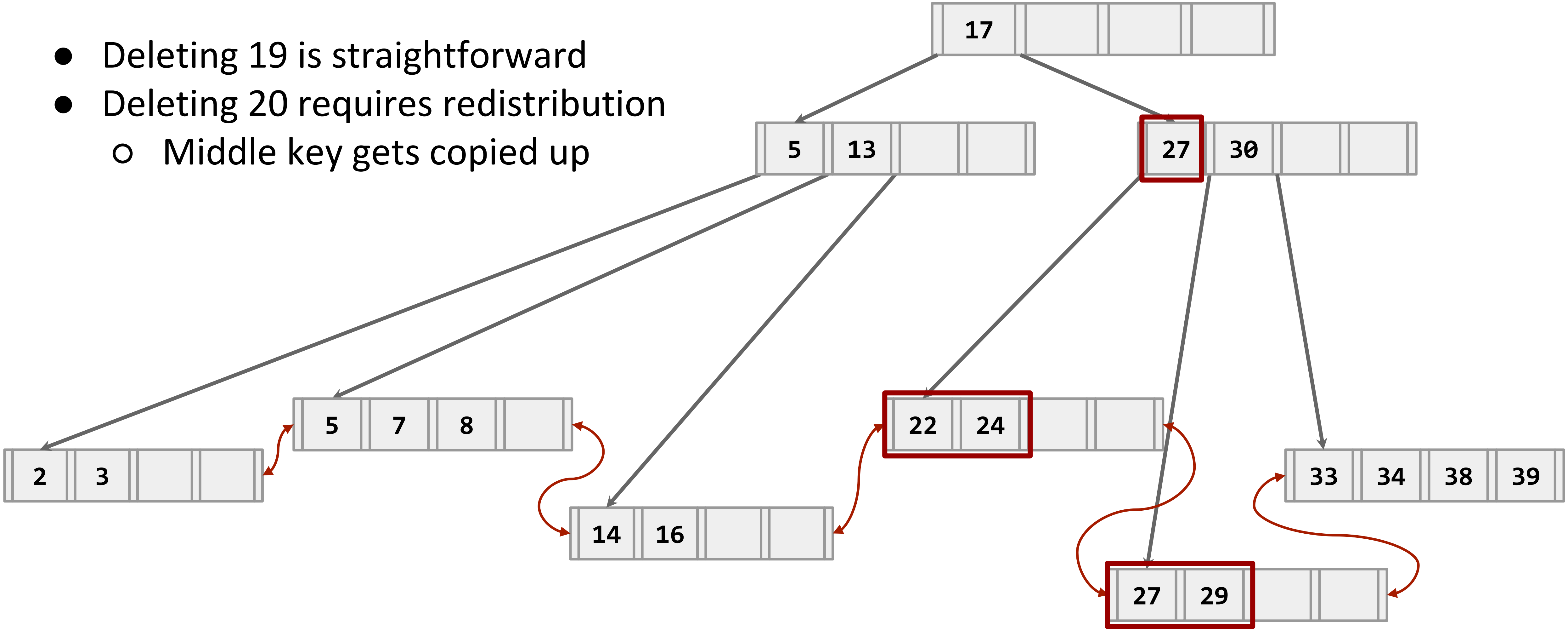
B^+ Tree example: Delete 19 and 20

- Deleting 19 is straightforward
- Deleting 20 requires redistribution



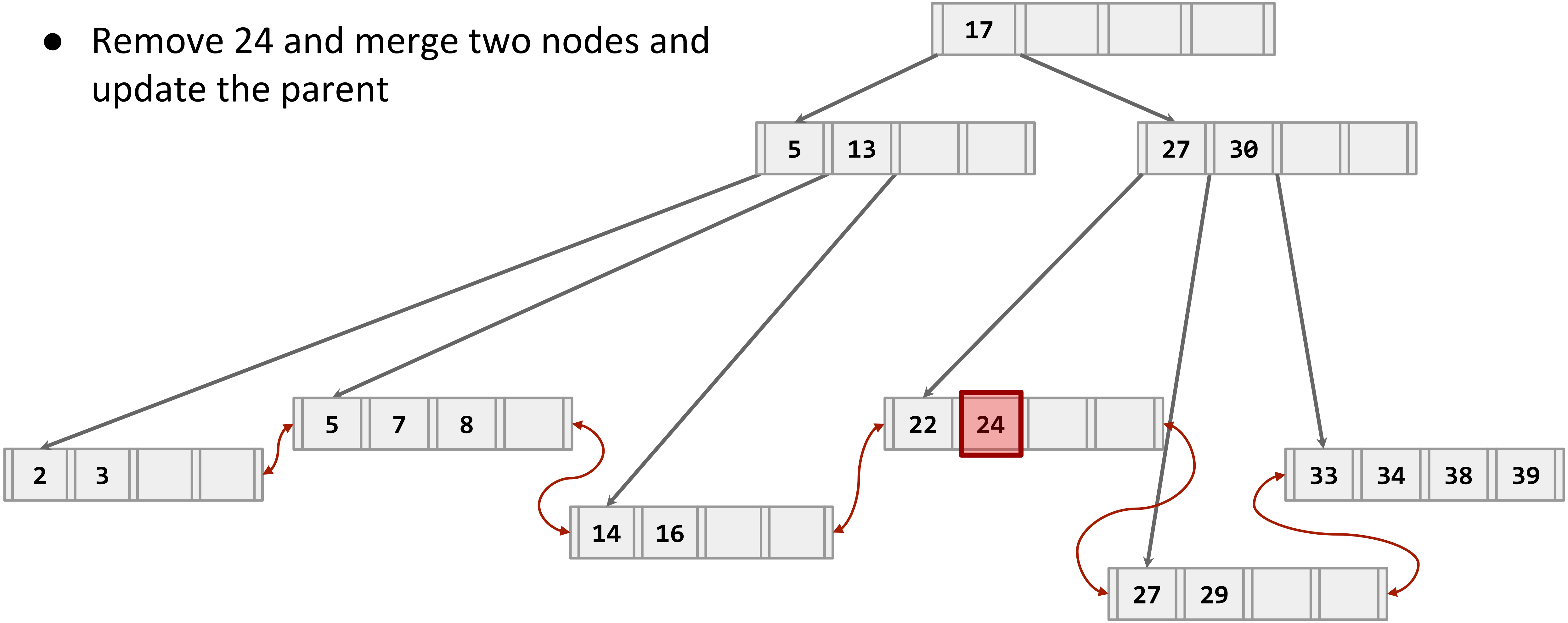
B+ Tree example: Delete 19 and 20

- Deleting 19 is straightforward
- Deleting 20 requires redistribution
  - Middle key gets copied up



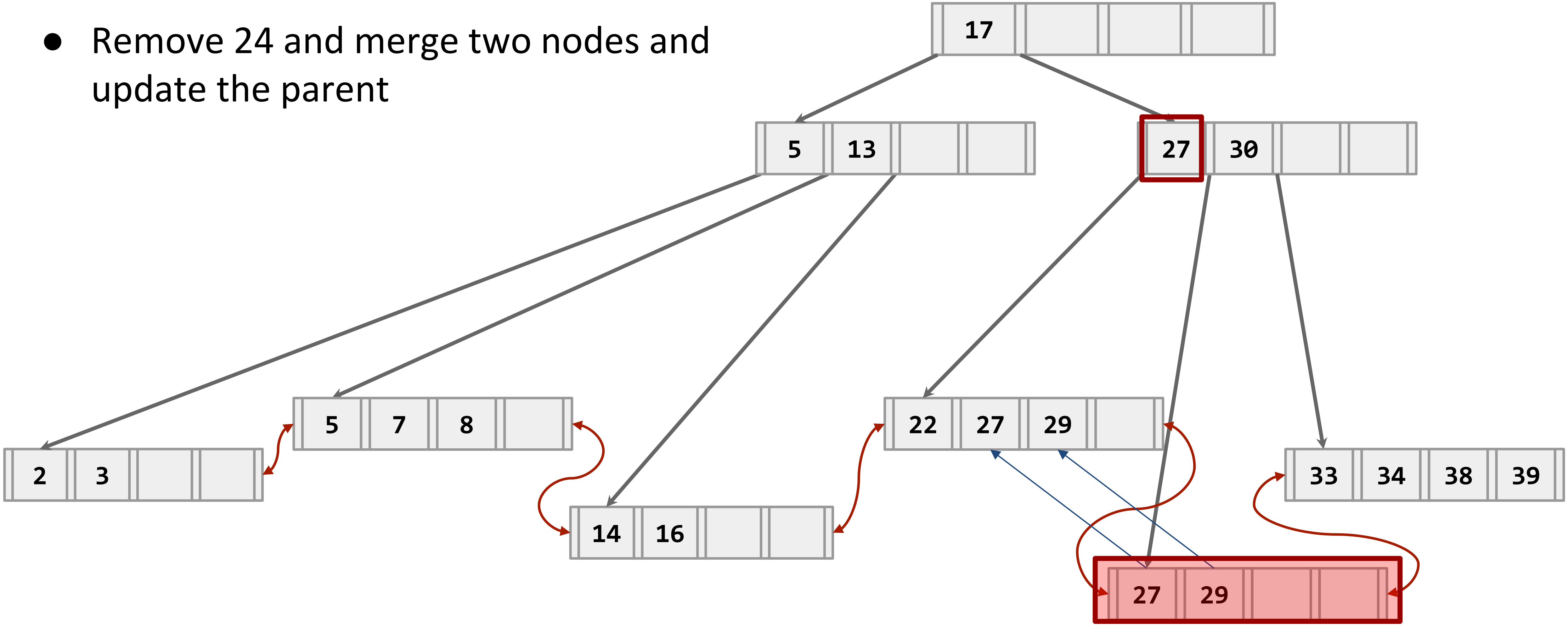
B^+ Tree example: Delete 24 now

- Remove 24 and merge two nodes and update the parent



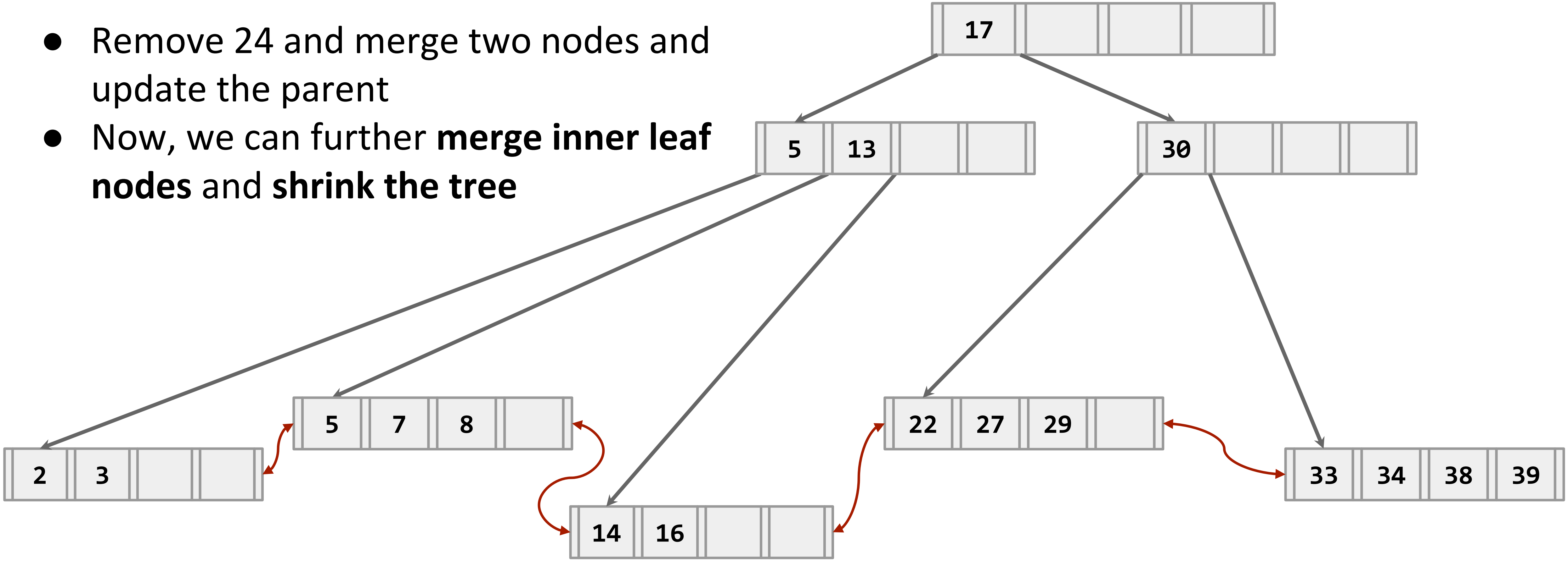
B^+ Tree example: Delete 24 now

- Remove 24 and merge two nodes and update the parent



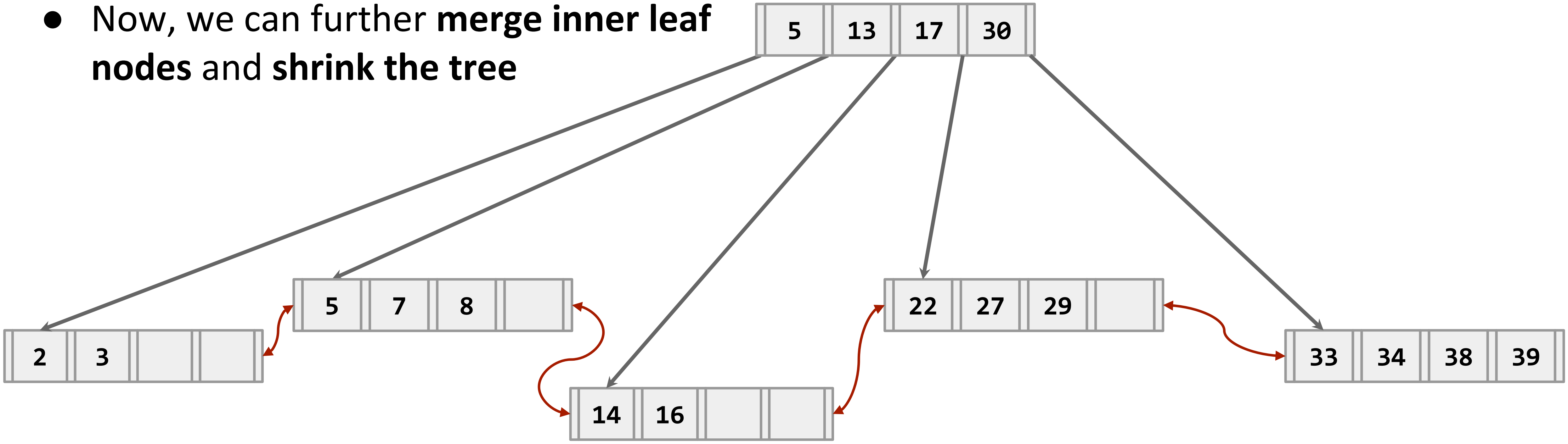
B⁺ Tree example: Delete 24 now

- Remove 24 and merge two nodes and update the parent
- Now, we can further **merge inner leaf nodes** and **shrink the tree**



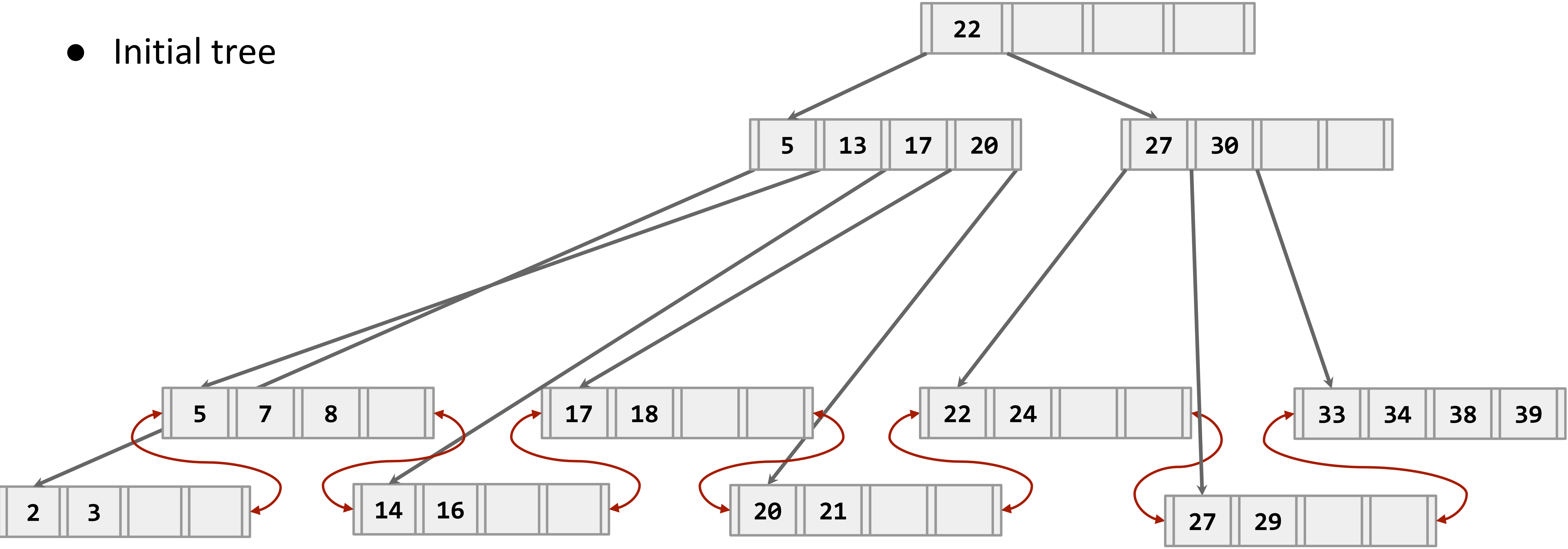
B+ Tree example: Delete 24 now

- Remove 24 and merge two nodes and update the parent
- Now, we can further **merge inner leaf nodes** and **shrink the tree**



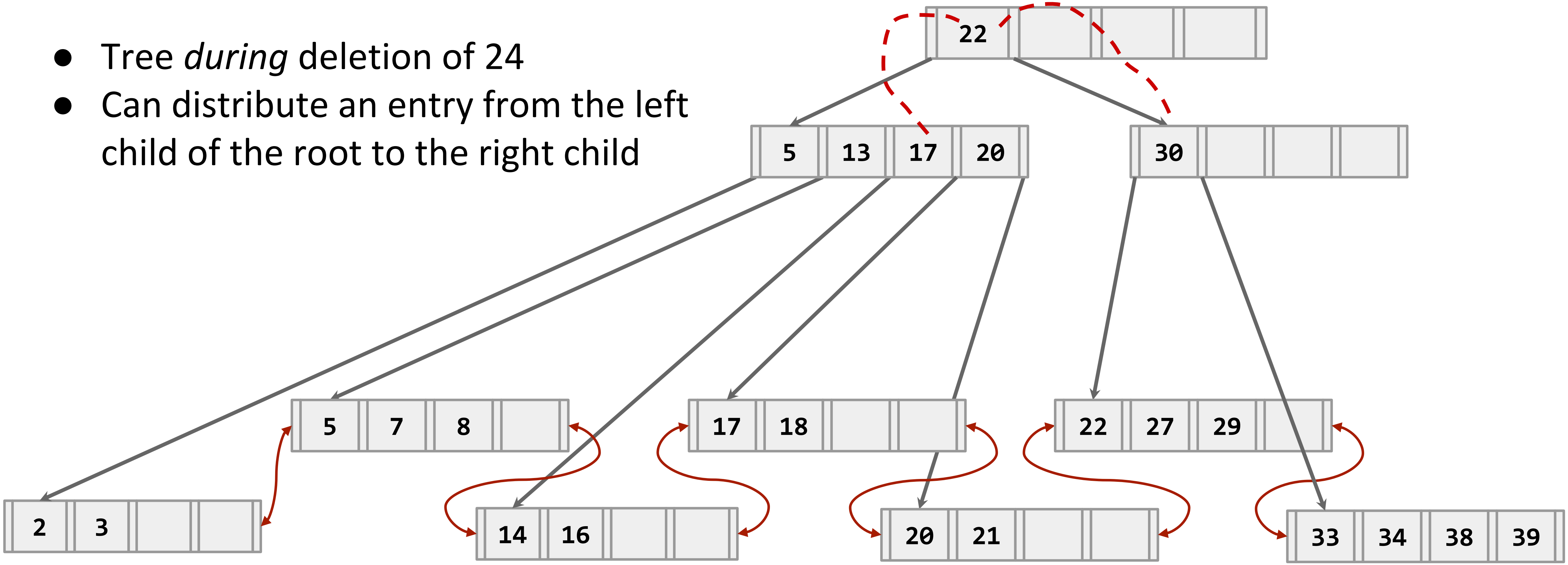
Another take at non-leaf redistribution

- Initial tree



Another take at non-leaf redistribution

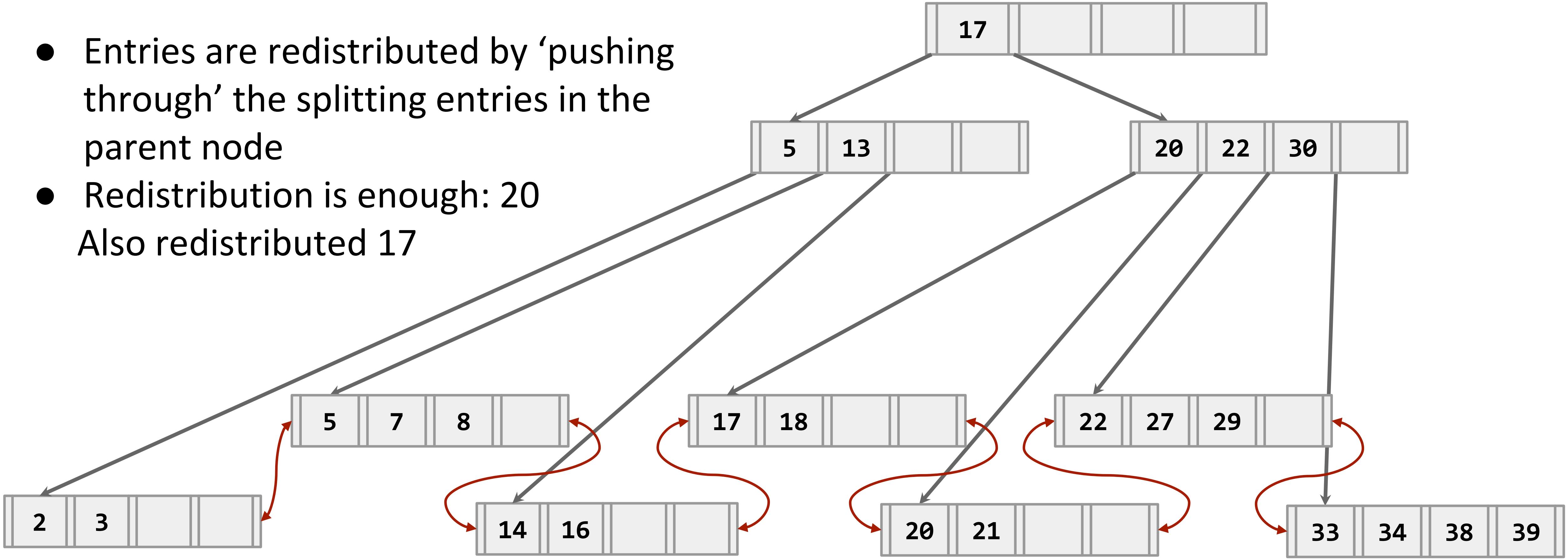
- Tree *during* deletion of 24
- Can distribute an entry from the left child of the root to the right child



Another take at non-leaf redistribution

- Entries are redistributed by ‘pushing through’ the splitting entries in the parent node
- Redistribution is enough: 20

Also redistributed 17



Clustered indexes

- The table is **physically** stored in the sort order specified by the primary key
    - Can be either heap- or index-organized storage

- Some DBMSs always use a clustered index
    - If a table does not contain the primary key, the DBMS will automatically make a hidden primary key

- Meanwhile, other DBMSs do not support them!

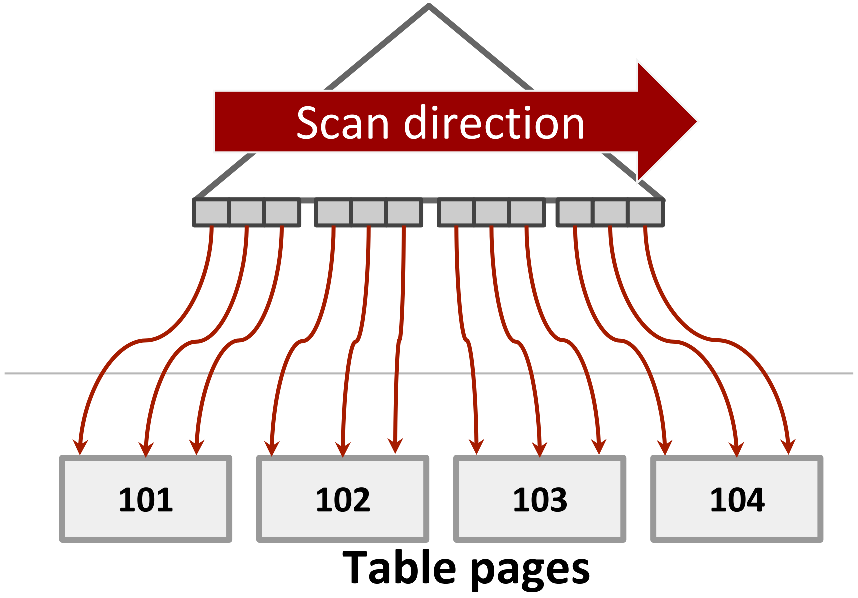
B$^+$ Tree traversal

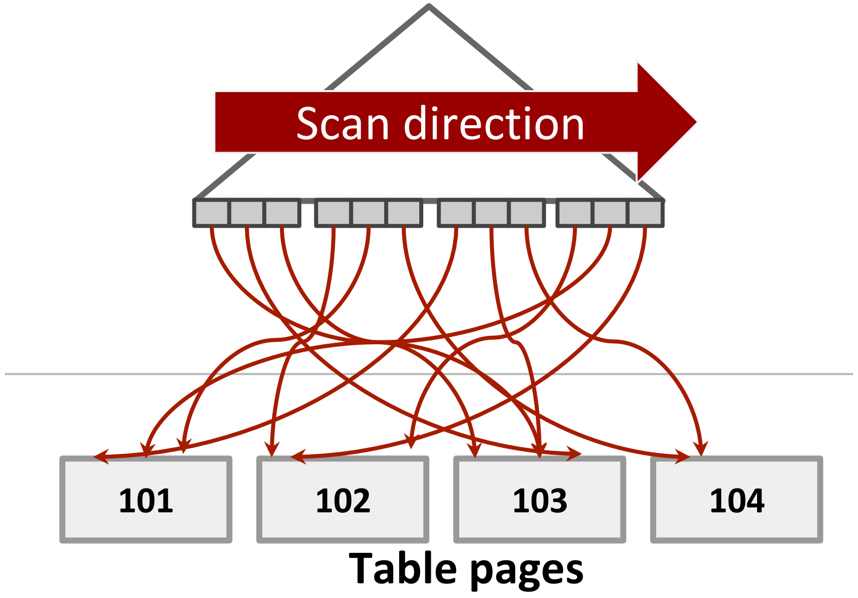
Clustered:

- Traverse to the leftmost leaf page and then retrieve tuples from all leaf pages
- This will always be better than sorting data for each query

Non-clustered:

- For non-clustered index, retrieving records in the order they appear in the leaves causes redundant page reads
- Better approach: Find all pages the query needs and then sort them based on their page ID





B^+ Tree design choices

- Node size
- Merge threshold
- Variable-length keys
- Intro-node search

Node size

- The slower the storage device, the larger the optimal node size for the tree:
  - HDD: ~1MB
  - SSD: ~10KB
  - In-memory: ~512B

- Optimal sizes can vary depending on the workload
  - Leaf node scans vs root-to-leaf traversals

Merge threshold

- Some DBMS do not always merge nodes when they are half full
  - (data sizes are growing, we expect more insertions than deletions)
  - Average occupancy rate for nodes is around 67%

- Delaying a merge operation may reduce the amount of reorganization
- Sometimes, it is better to just let smaller nodes exist and then periodically rebuild entire tree
  - Example: PostgreSQL calls their implementation as a “non-balanced” B^+ Tree

Variable-length keys

- **Pointers**
  - Store the keys as pointers to the tuple's attribute
- **Variable-length nodes**
  - The size of each node in the index may vary
  - Requires careful memory management
- **Padding**
  - Always pad the key to be max length of the key type
- **Key Map / Indirection**
  - Embed an array of pointers that map to the key + value list within the node

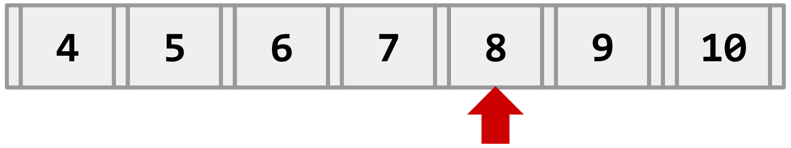
Intra-node search

- **Linear search**
  - Scan node keys from beginning to end
  - High performance using SIMD instructions

Intra-node search

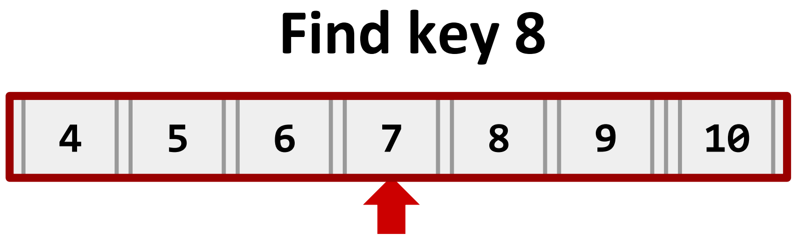
- **Linear search**
  - Scan node keys from beginning to end
  - High performance using SIMD instructions

Find key 8



Intra-node search

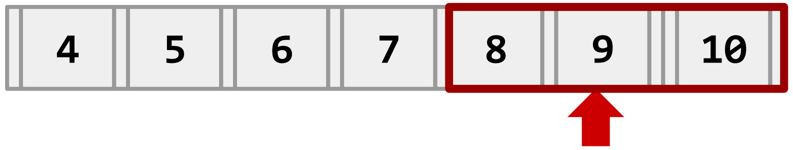
- **Linear search**
  - Scan node keys from beginning to end
  - High performance using SIMD instructions
- **Binary search**
  - Jump to middle key, pivot left/right depending on comparison



Intra-node search

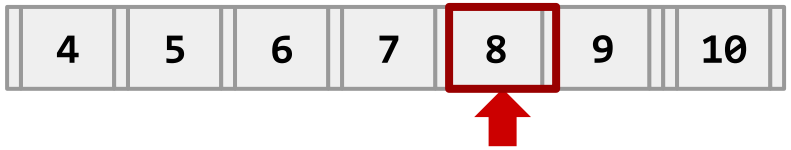
- **Linear search**
  - Scan node keys from beginning to end
  - High performance using SIMD instructions
- **Binary search**
  - Jump to middle key, pivot left/right depending on comparison

Find key 8



Intra-node search

- **Linear search**
  - Scan node keys from beginning to end
  - High performance using SIMD instructions
- **Binary search**
  - Jump to middle key, pivot left/right depending on comparison



Intra-node search

- **Linear search**
  - Scan node keys from beginning to end
  - High performance using SIMD instructions
- **Binary search**
  - Jump to middle key, pivot left/right depending on comparison
- **Interpolation**
  - Approximate the location of desired key based on known distribution of keys

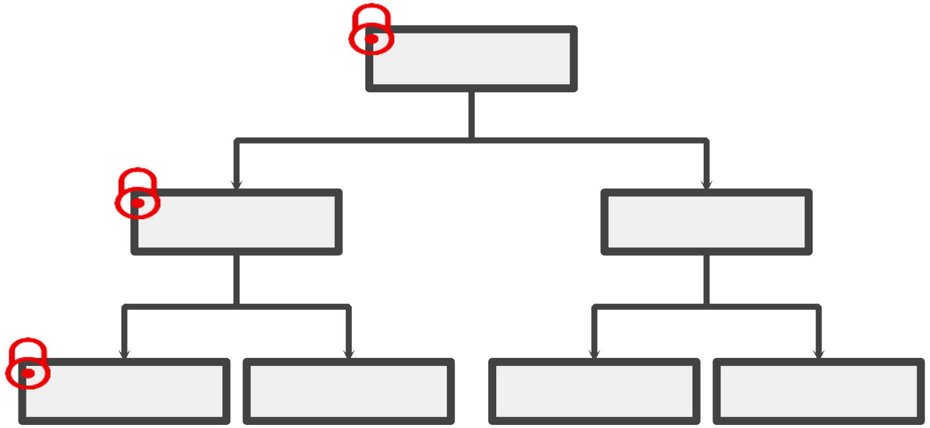
4	5	6	7	8	9	10

x: search key
arr[]: array where elements need to be searched
low: starting index in arr[]
high: ending index in arr[]

Index of x: $\mathrm{low} + \frac{(\mathrm{x}-\mathrm{arr[low]})*(\mathrm{high}-\mathrm{low})}{\mathrm{arr[high]}-\mathrm{arr[low]}} = 0 + \frac{(8-4)*(6-0)}{10-4} = 4$

Concurrently accessing B$^+$ Tree

- Handling concurrent access for the tree is not straightforward:
  - Simple page locking/latching is not enough
  - Will protect against “simple” (single page) changes
  - However, pages depend on each other

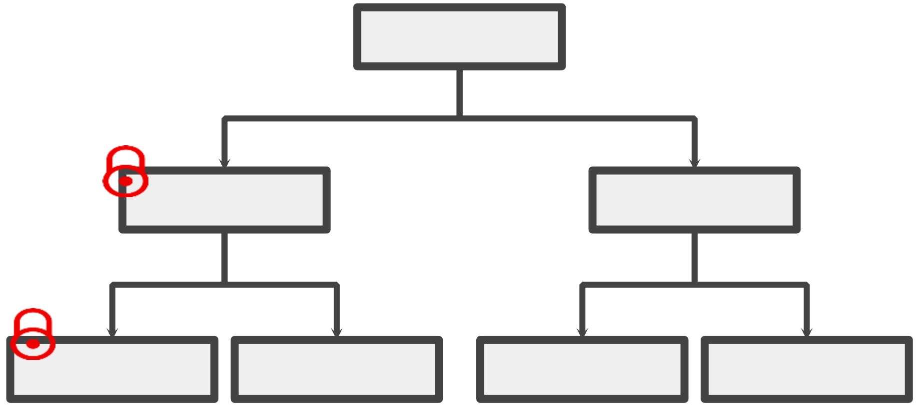


- Classical technique is **lock coupling**
  - A thread latches both the page and its parent page
  - i.e., latch the root first, latch the first level, release the root, latch the second level etc.
  - Prevents conflicts, as pages can only be split when the parent is latched
  - No deadlocks, as the latches are ordered (canonical)

Concurrently accessing B^+ Tree

- Handling inserts:
  - When a leaf is split, the entry is propagated up
  - Might go up all the way to the root
  - But we only have locked one parent

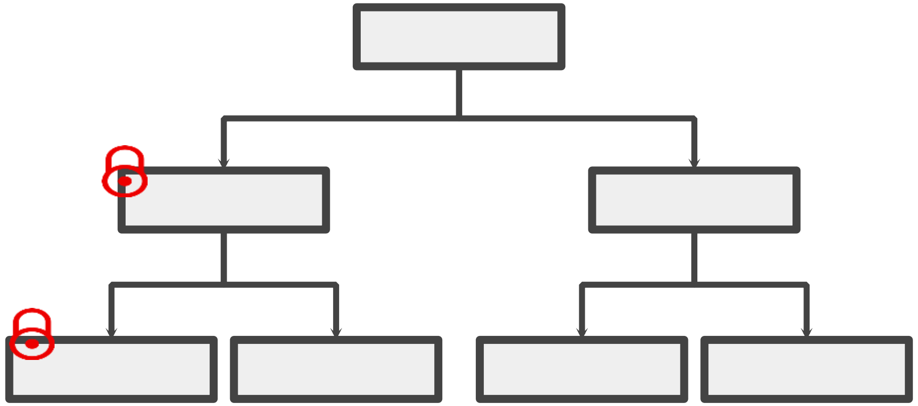
- Naive lock coupling can result in deadlocks



Concurrently accessing B^+ Tree

Alternative approach: Use restart or optimistic coupling

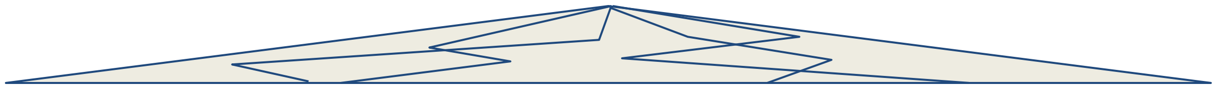
1. First try to insert using simple lock coupling
2. If we do not split the inner node, everything is fine
3. Otherwise, release all latches
4. Restart the operation, but now hold all the latches all the way to the root
5. All operations can now be executed safely



- Greatly reduces concurrency
- A rare scenario
- Simple to implement

B⁺ Tree in practice (cool facts!)

- Typical order: 100. Typical fill-factor: 67%.
  - Average fanout = 2*100*0.67 = 134
- Typical capacities:
  - Height 4: 134^4 = 322,417,936 entries
  - Height 3: 134^3 = 2,406,104 entries
- Top levels can always be in memory:
  - Level 1 = 1 page = 8 KB
  - Level 2 = 134 pages = 1 MB
  - Level 3 = 17,956 pages = 140 MB



Summary

- Tree indexes are ideal for range-searches
  - Also good for equality search
- B^+ Tree is a versatile, dynamic data structure
  - Inserts/deletes leave tree height-balanced
  - High fanout means depth rarely more than 3 or 4
  - Almost always better than maintaining a sorted file
  - 67% occupancy on an average
- Most widely-used index in database systems
  - One of the most optimized component of a DBMS