CSE 541: Database Systems I

Tree Structures

Slides Credit: Tianzheng Wang

Range Search

StudentID	Name	GPA
100	John	3.0
101	Jack	3.0
102	May	3.5

Query:
Find all students with GPA > 3

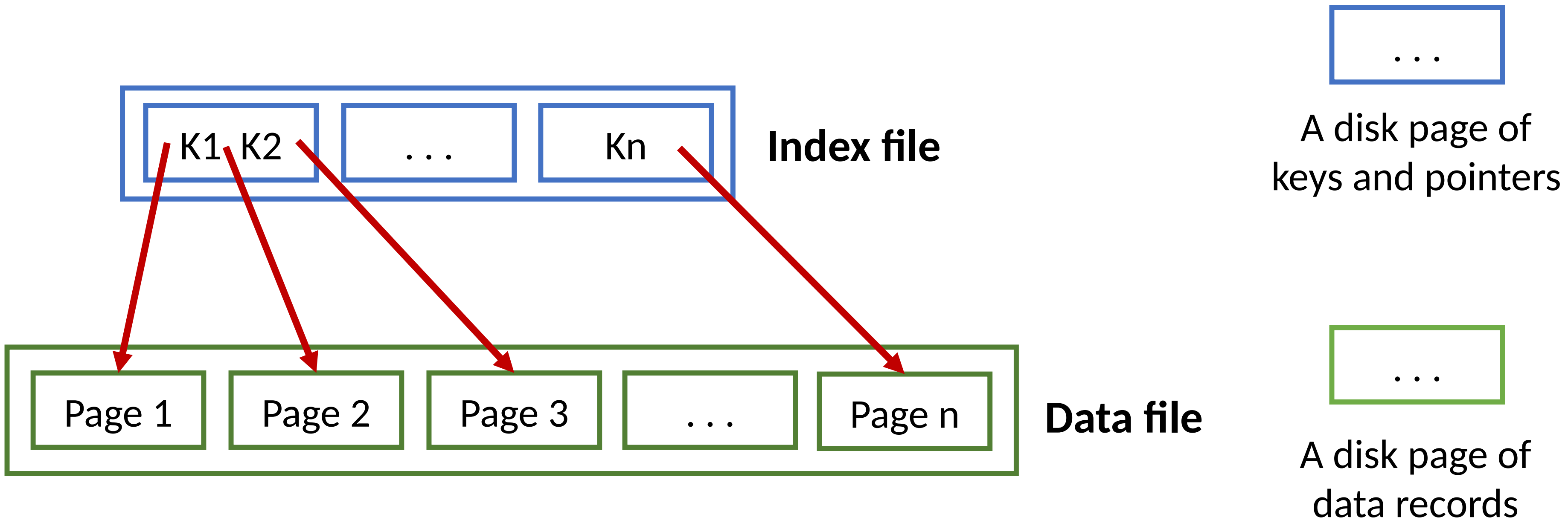
Assume data is in sorted file (stored disk/SSDs)

- Do binary search to find the first student
- Then scan to find the rest

➔Potentially high cost of binary search (many random I/Os)

Index Sequential Access Method (ISAM)

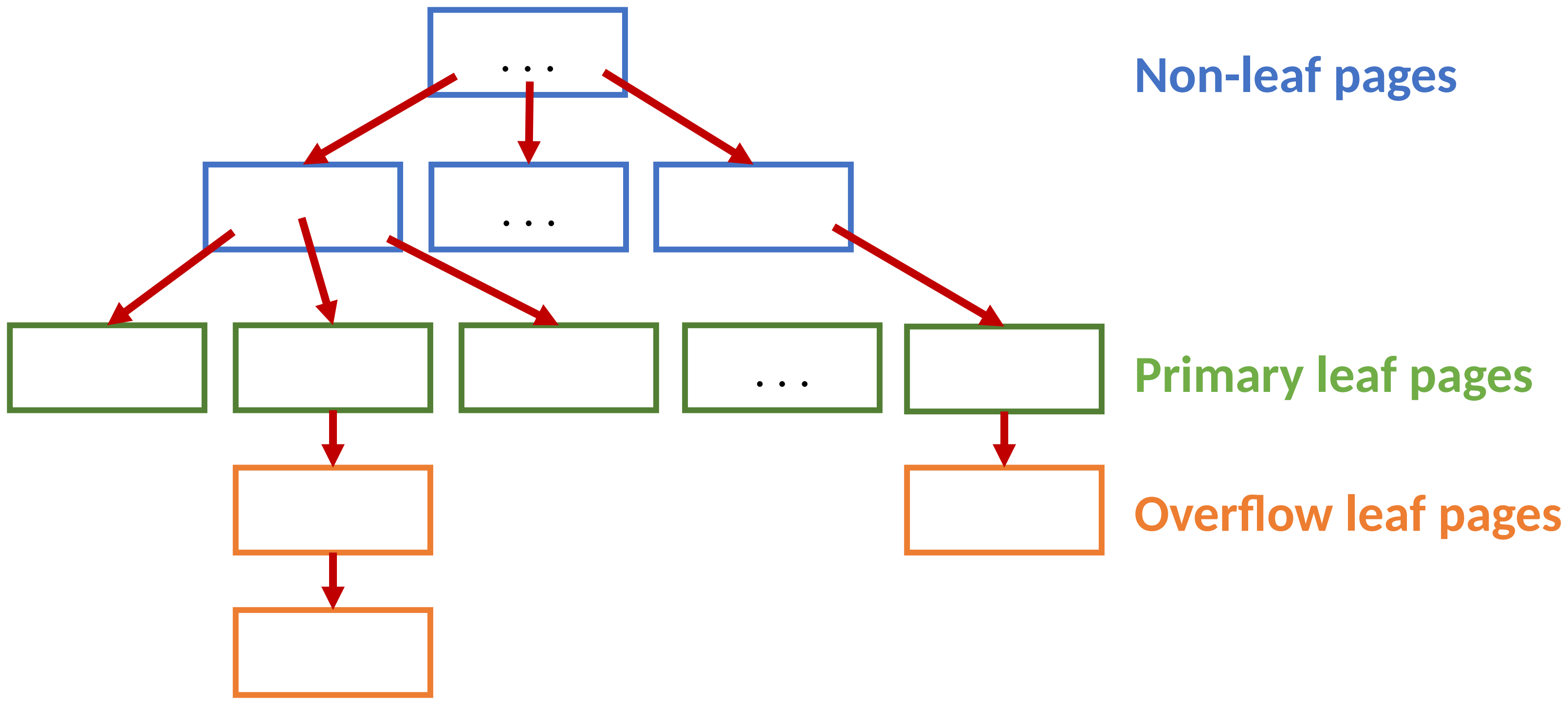
An “index file” that points to actual records



- Now we can do binary search on the (smaller) index file
- But the index file may still be quite large
  - Solution: apply the same idea repeatedly

ISAM Tree

- File creation
  - Allocate leaf pages sequentially, sorted by key
  - Then allocate index (non-leaf) pages
    - Index entries: <key, page ID>: “direct” search for data entries (in leaf)
  - Then allocate space for overflow leaf pages

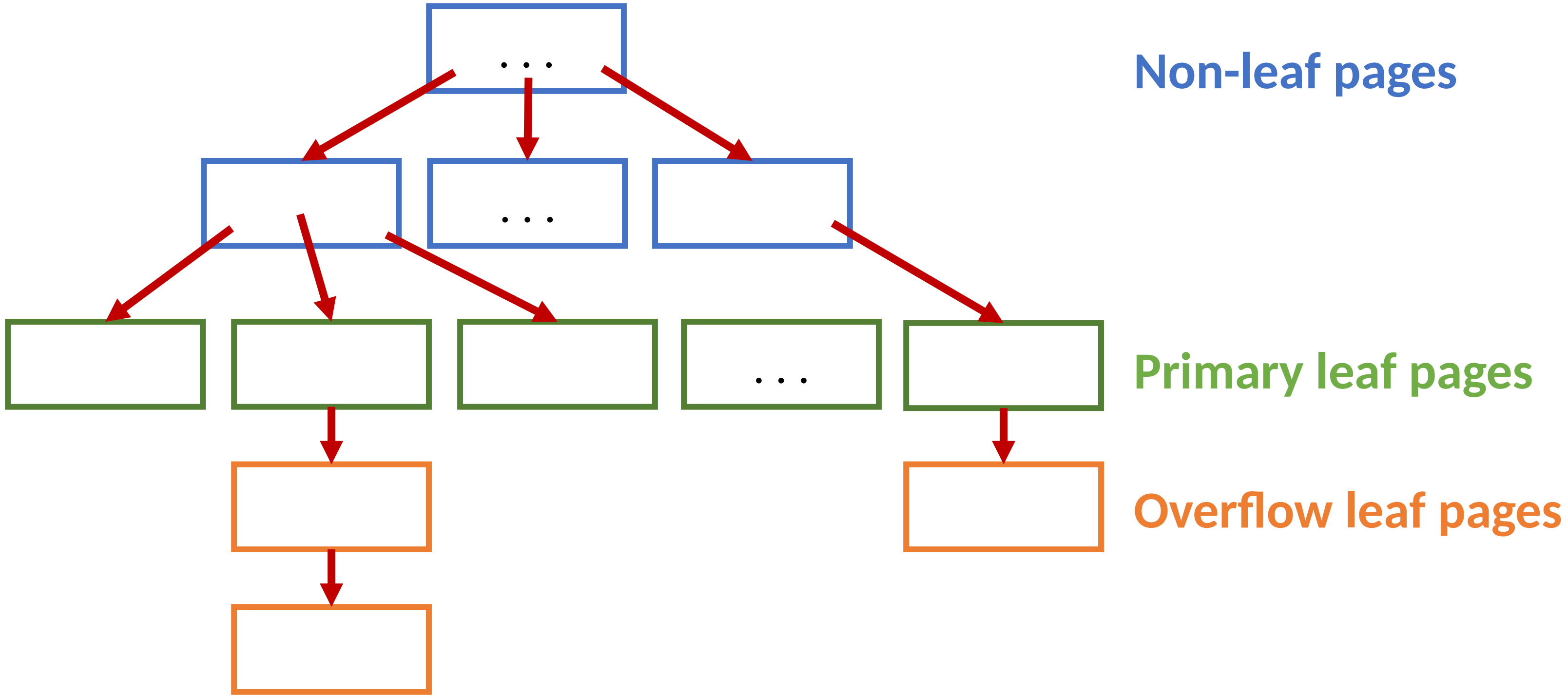


ISAM Tree

- Search: start from root, use key comparisons to go to leaf
- Insert: find leaf node, put it there
- Delete: find and delete in leaf node; deallocate node if empty

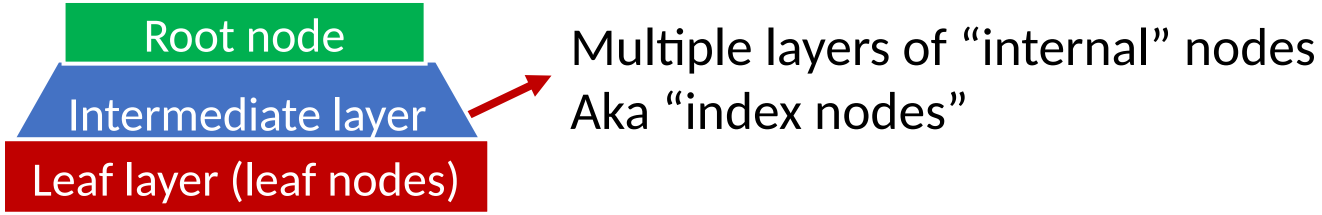
Static structure: insert/delete affect only leaf nodes, no need to lock non-leaf pages

Main problem: long overflow chains – high overhead



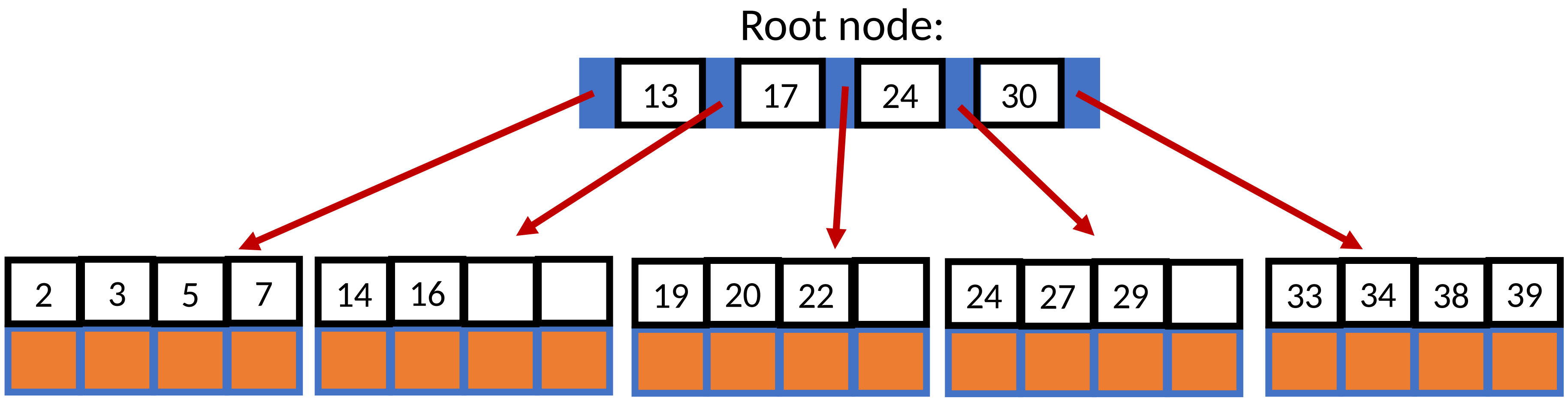
B+-Tree ("B-Tree")

- Dynamic structure, probably the most widely used DBMS index



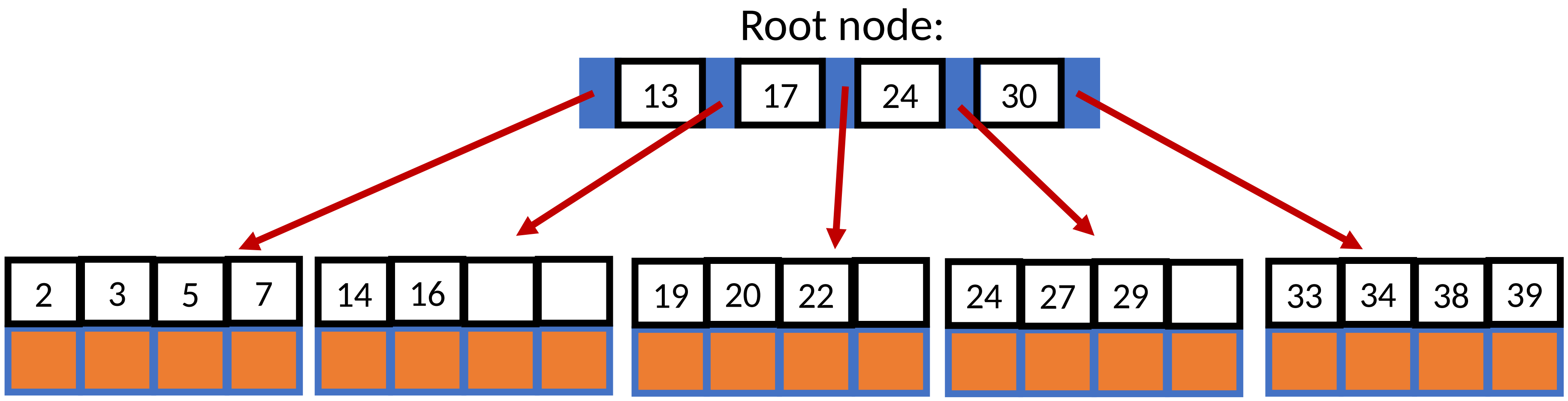
- Each node occupies a storage page
- Balanced: all paths from root to the lowest level have the same length
  - Insert/delete complexity $Log_F N$ (F – fanout, N – # leaf nodes)
    - Fanout – number of child pointers in internal node
  - Minimum 50% occupancy (except for root node)
    - D <= M <= 2D entries per node
    - 2D: order of the tree (i.e., max capacity)
  - Data entries are only stored in leaf nodes
    - Internal nodes only for "guiding the traffic" to leaf nodes
    - Other variants of B-Trees may store data entries with internal nodes

B-Tree: Example



- Root node
  - Initially a leaf node; becomes internal node as tree grows
- Internal node
  - Store pointers (addresses) and separator keys
  - N + 1 pointers, N keys, keys can be variable-length
- During traversal, compare target key and separator key
  - E.g., go left if <, go right if >=

B-Tree: Example



- Leaf node: store M keys and M payloads
  - Payload can be real record data or addresses
  - Both keys and payloads can be variable-length
  - May be chained together to accelerate range scans
- Internal vs. leaf nodes
  - Internal nodes usually hold more entries than leaf nodes do
  - Internal nodes change more slowly than leaf nodes

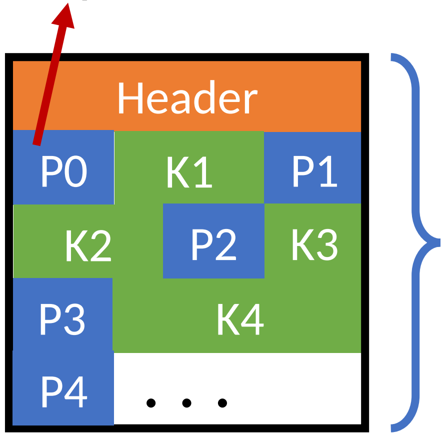
Node Format

Pn: pointer to child node
Kn: separator key

Alternative 1:

- May need to rearrange keys upon insert
- Not efficient to search – linear scan

P0: pointer to the left-most child



Fixed size:
E.g., 4KB, 8KB

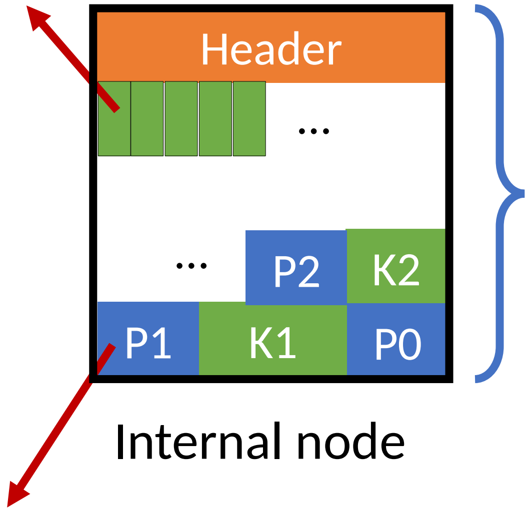
Internal node

Leaf node: similar, but with equal number of keys and payloads

Alternative 2:

- Upon insert, only need to rearrange slots which are fixed size
- Faster search: binary search on slots
    - Debatable in practice: depends on node size

Key-pointer slots: in-node offset of the i-th key.
Grows "downward"



Fixed size:
E.g., 4KB, 8KB

Internal node

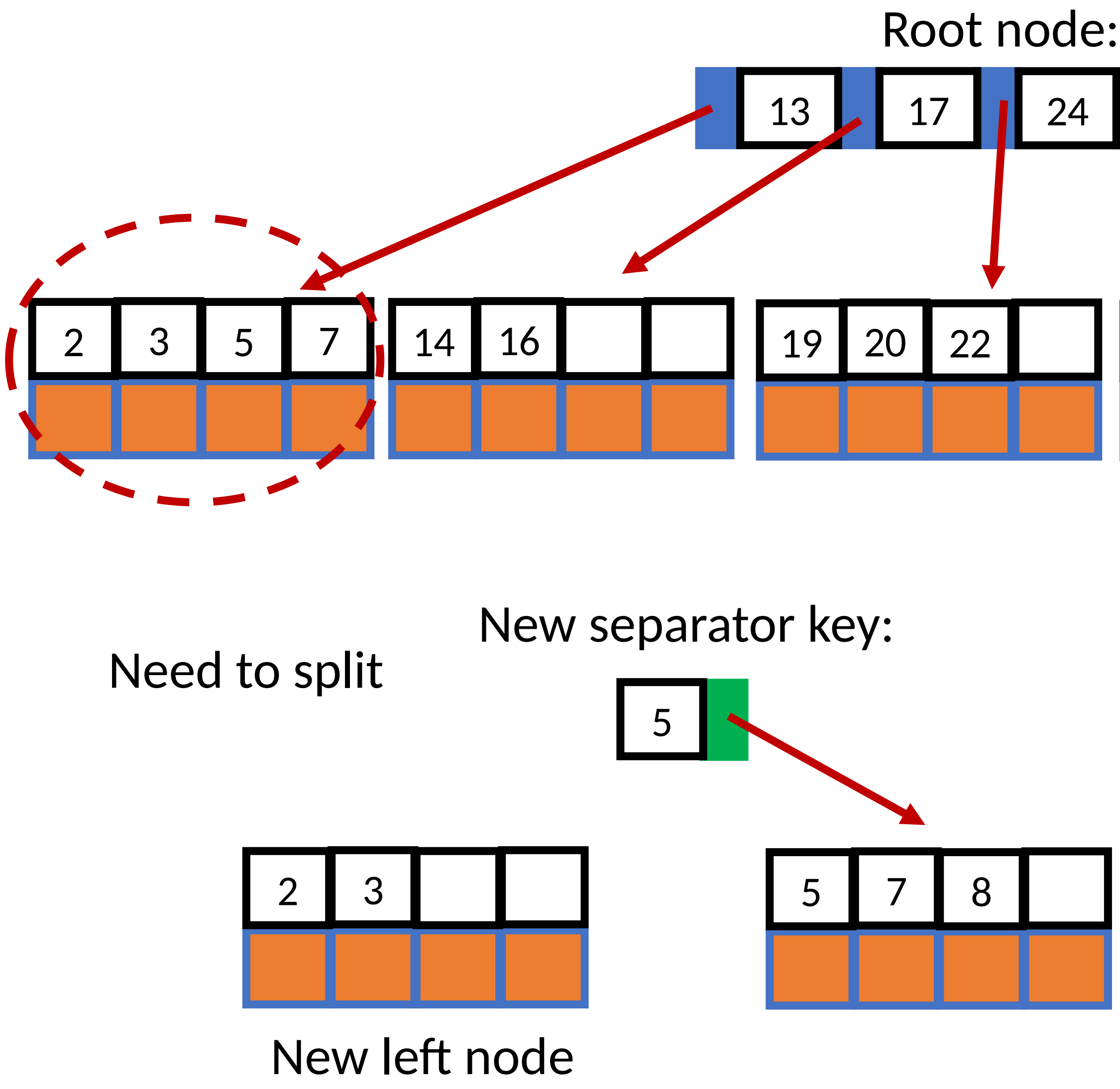
Actual key/pointer storage: grows "upward"

B-Tree Insert

- Traverse the tree to find the correct leaf L
- Put data entry into L
  - Done if L has enough space
    - Recall each node has fixed size/maximum size limit – typically a page
  - Otherwise split L into new node L1 and L2
    - Redistribute entries evenly
    - Insert the middle key (new separator key) to parent node

    ➔ May cause further splits, which will get propagated to the root level and grow the tree

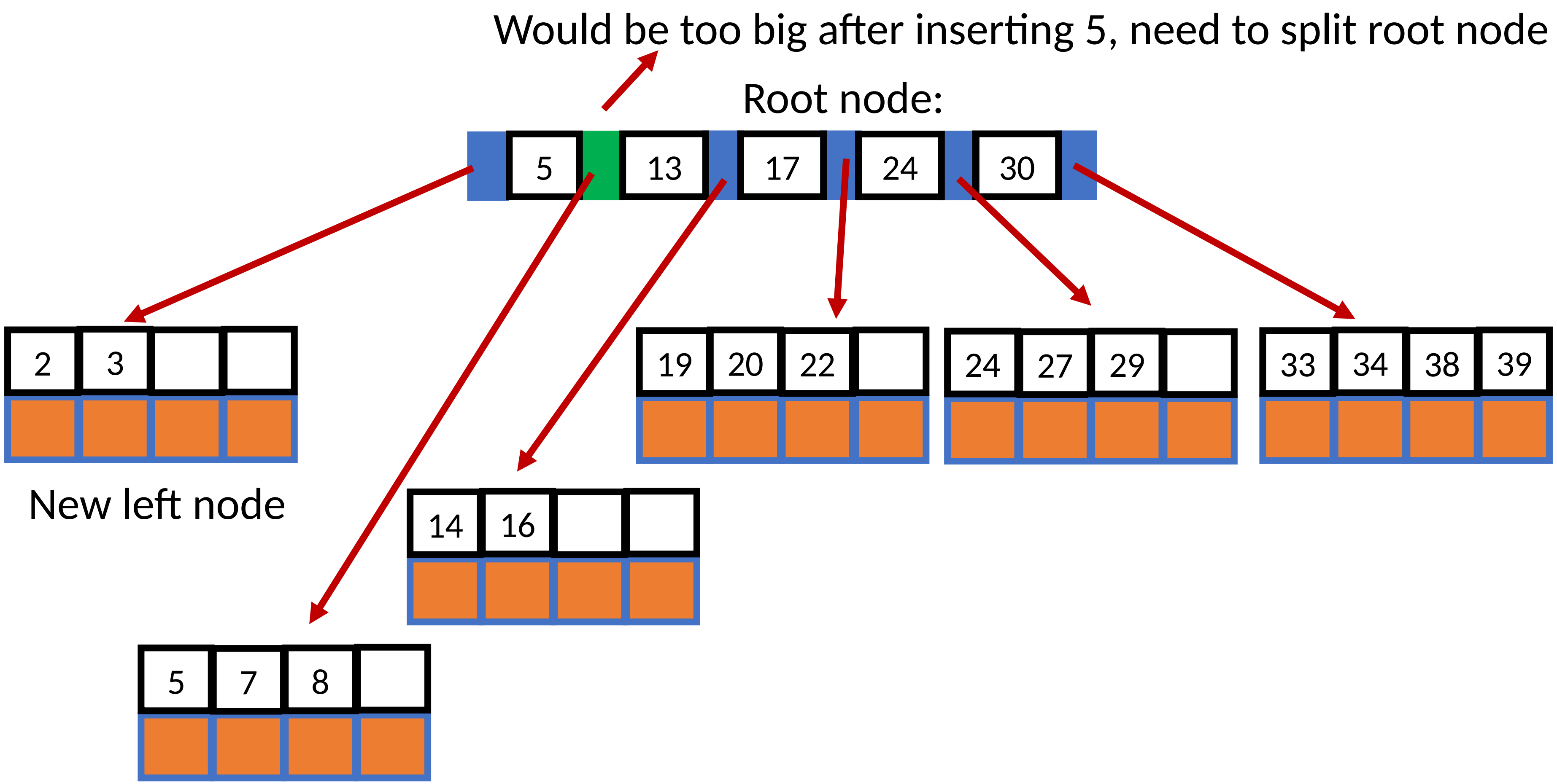
B-Tree Insert



Note: 5 is the new separator key, and is copied up. It still exists in the leaf node.

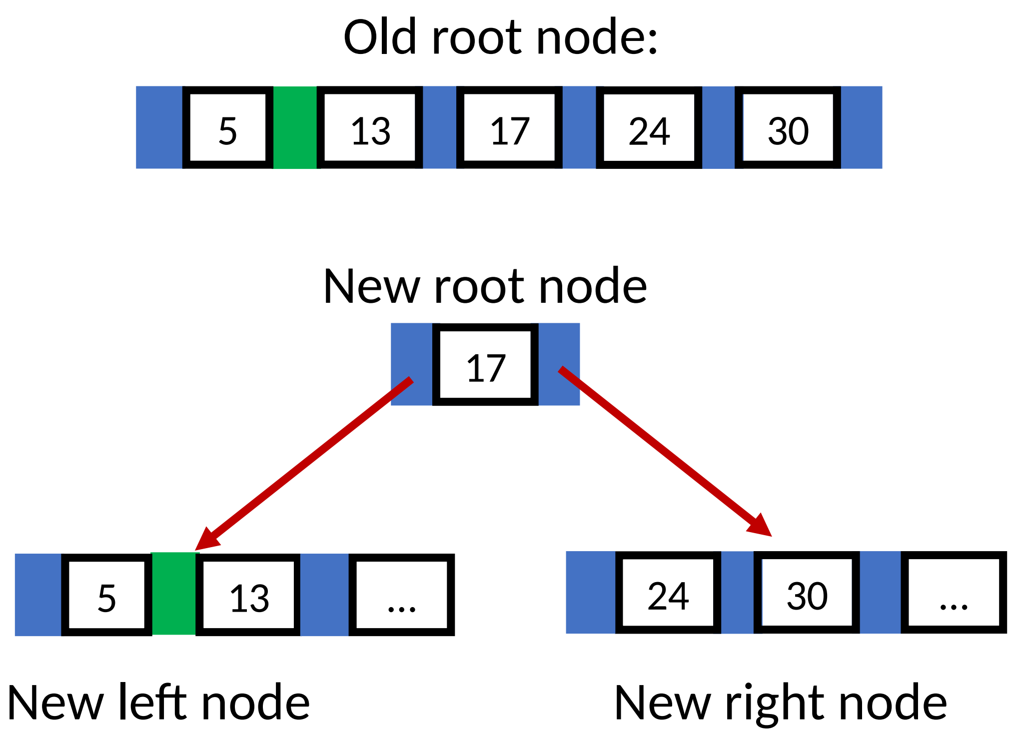
B-Tree Insert

Insert 5 to parent node (root)



B-Tree Insert

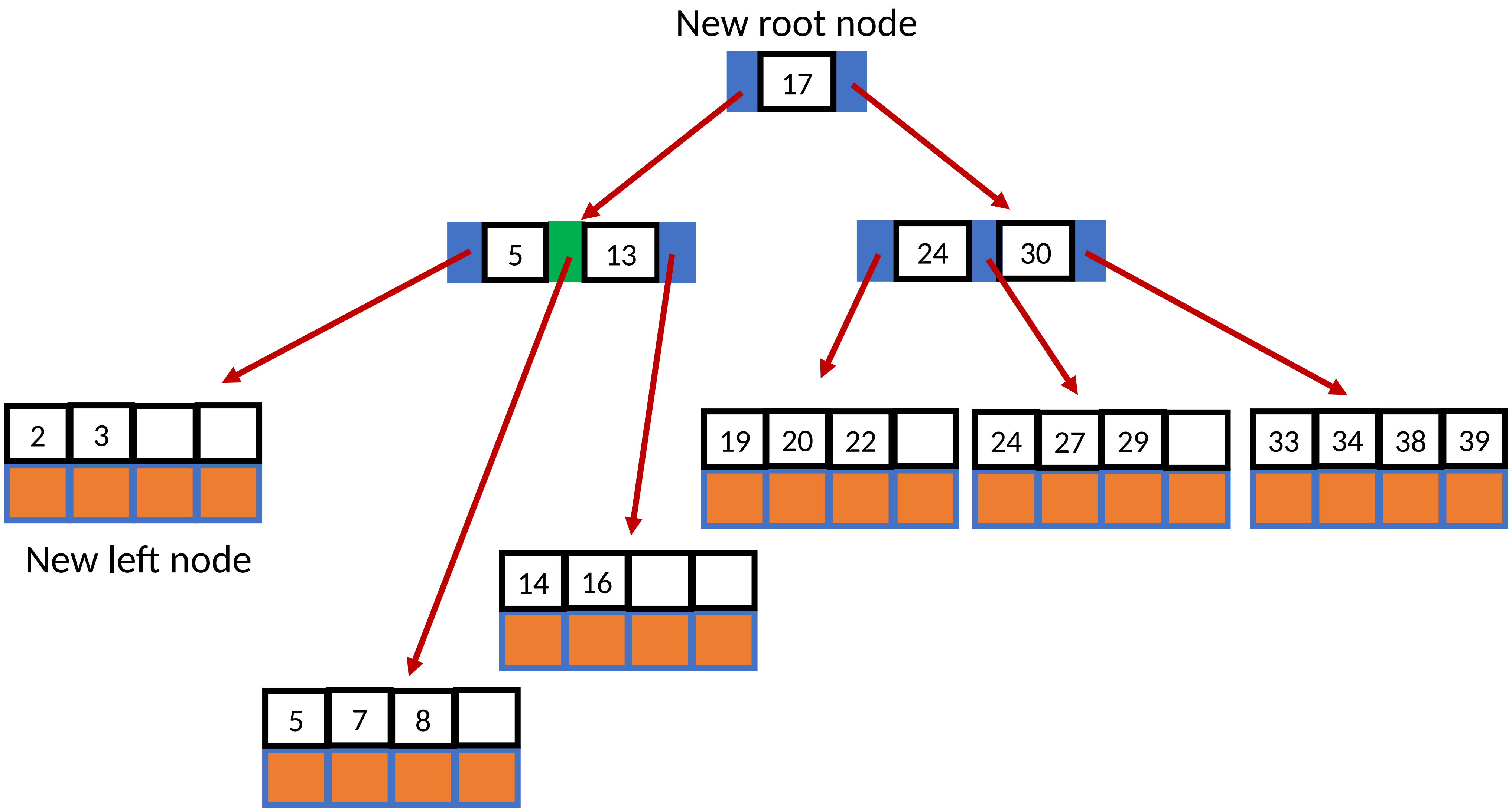
Insert 5 to parent node (root)



Note:

- 17 is the new separator key, and is pushed up. It does not exist in the lower level child node
- New root generated, height increased

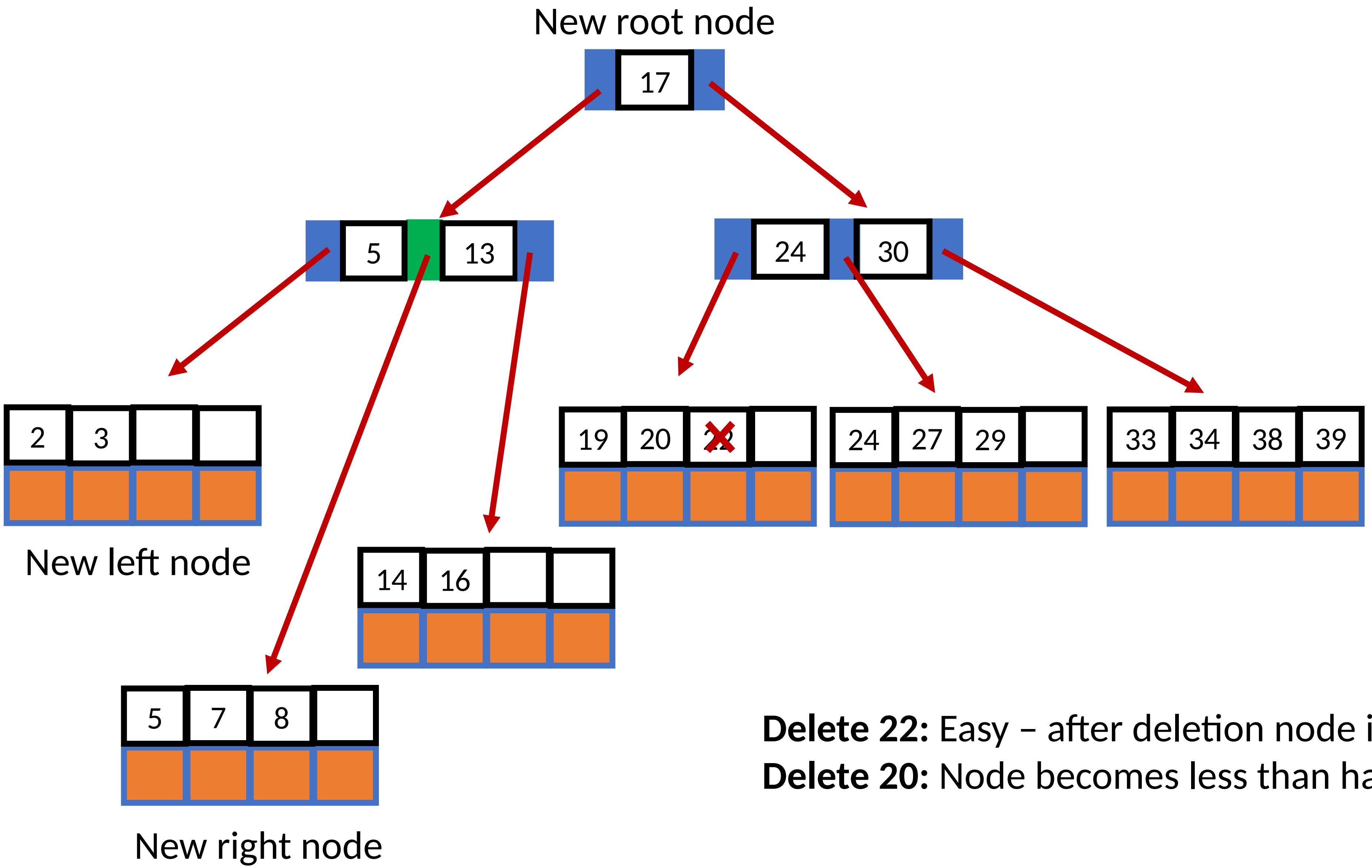
B-Tree Insert



B-Tree Delete

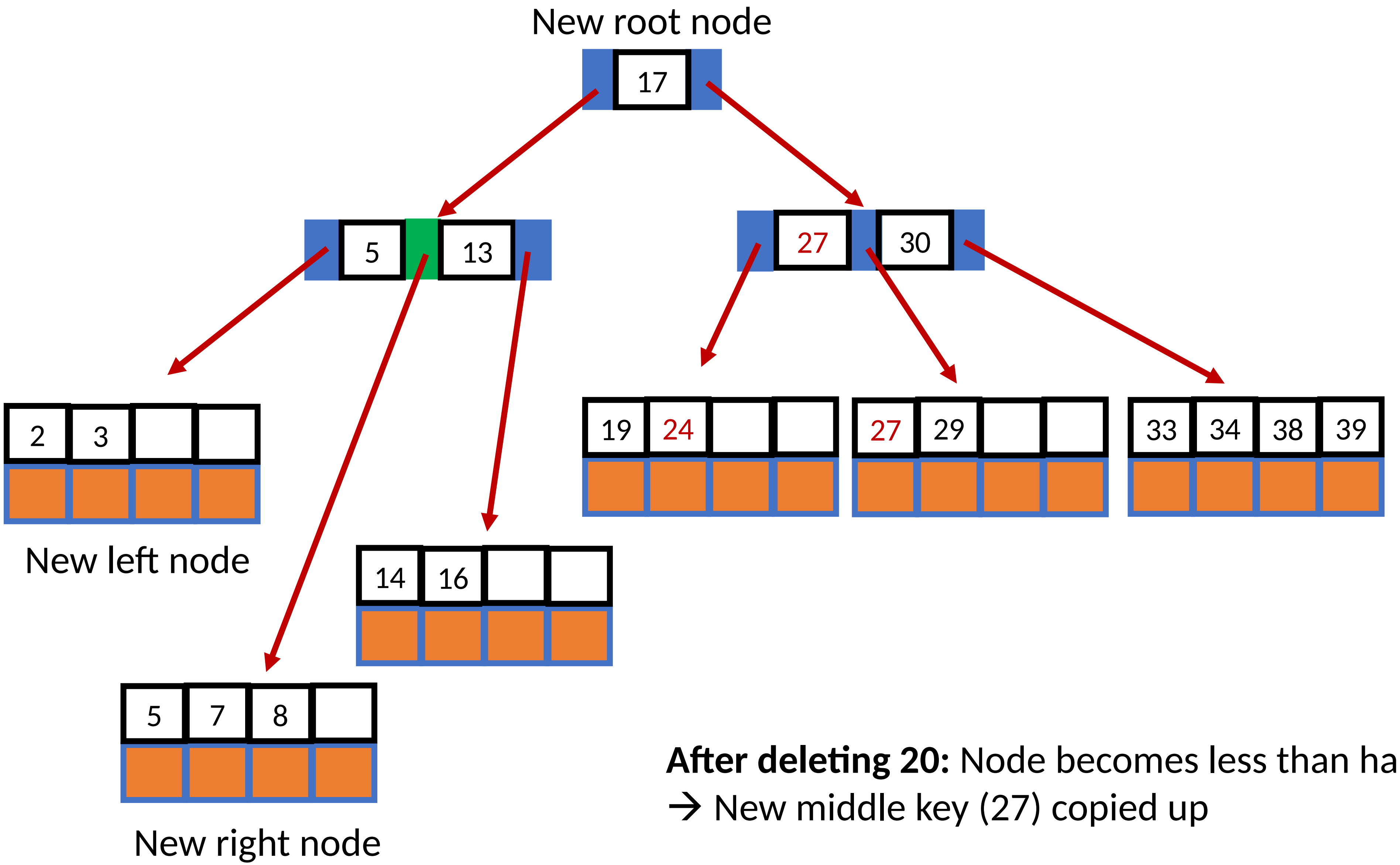
- Traverse the tree to find the correct leaf L
- Remove entry
  - Done if L is at least half full
  - Otherwise redistribute or merge
    - Redistribute: borrow from sibling
    - Merge: combine with sibling to get a single node, delete key pointer at the parent node
      - If the parent node then becomes less than half full, continue to merge
      - Might propagate to the upper levels and shrink the tree

B-Tree Delete



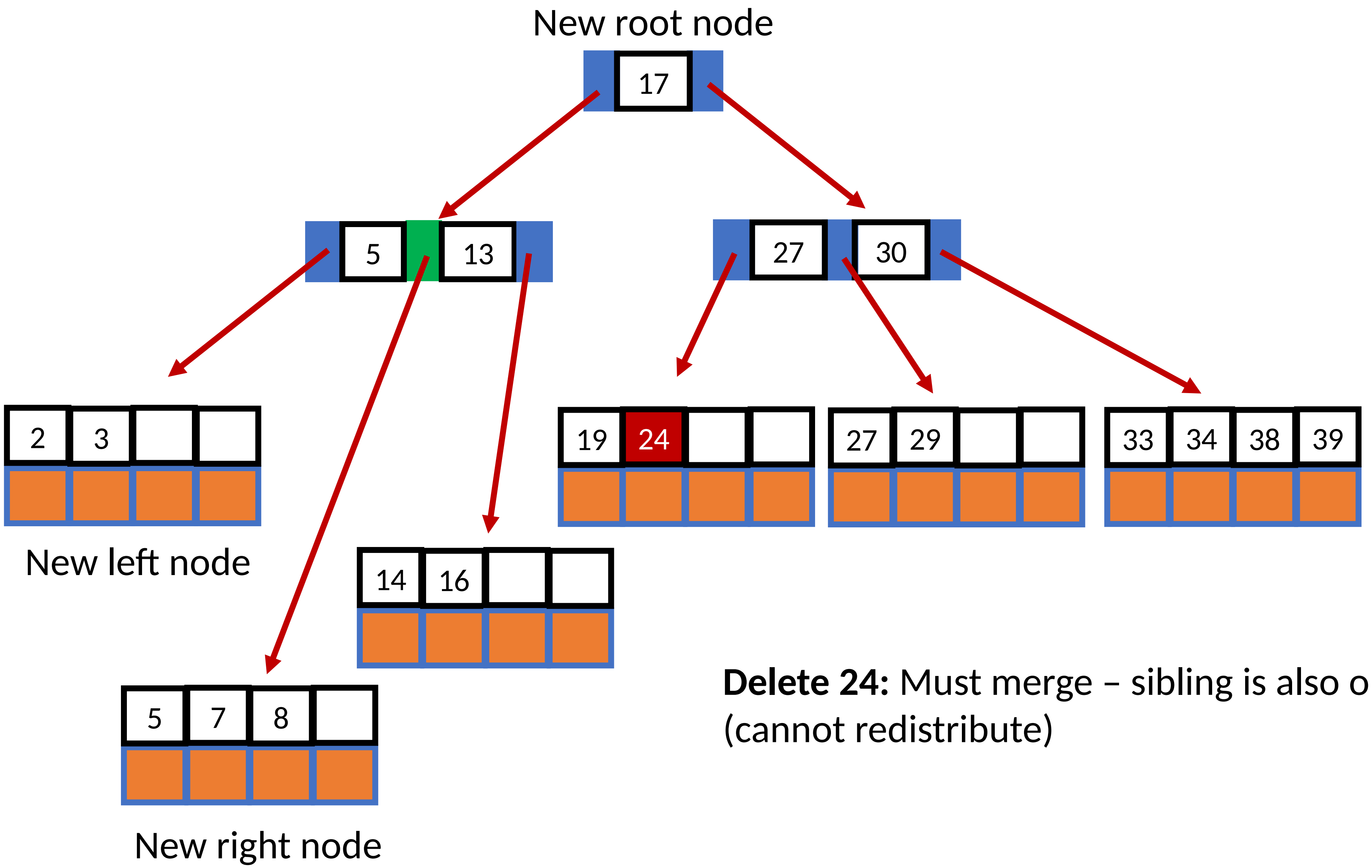
Delete 22: Easy – after deletion node is still half-full
Delete 20: Node becomes less than half-full, redistribute

B-Tree Delete



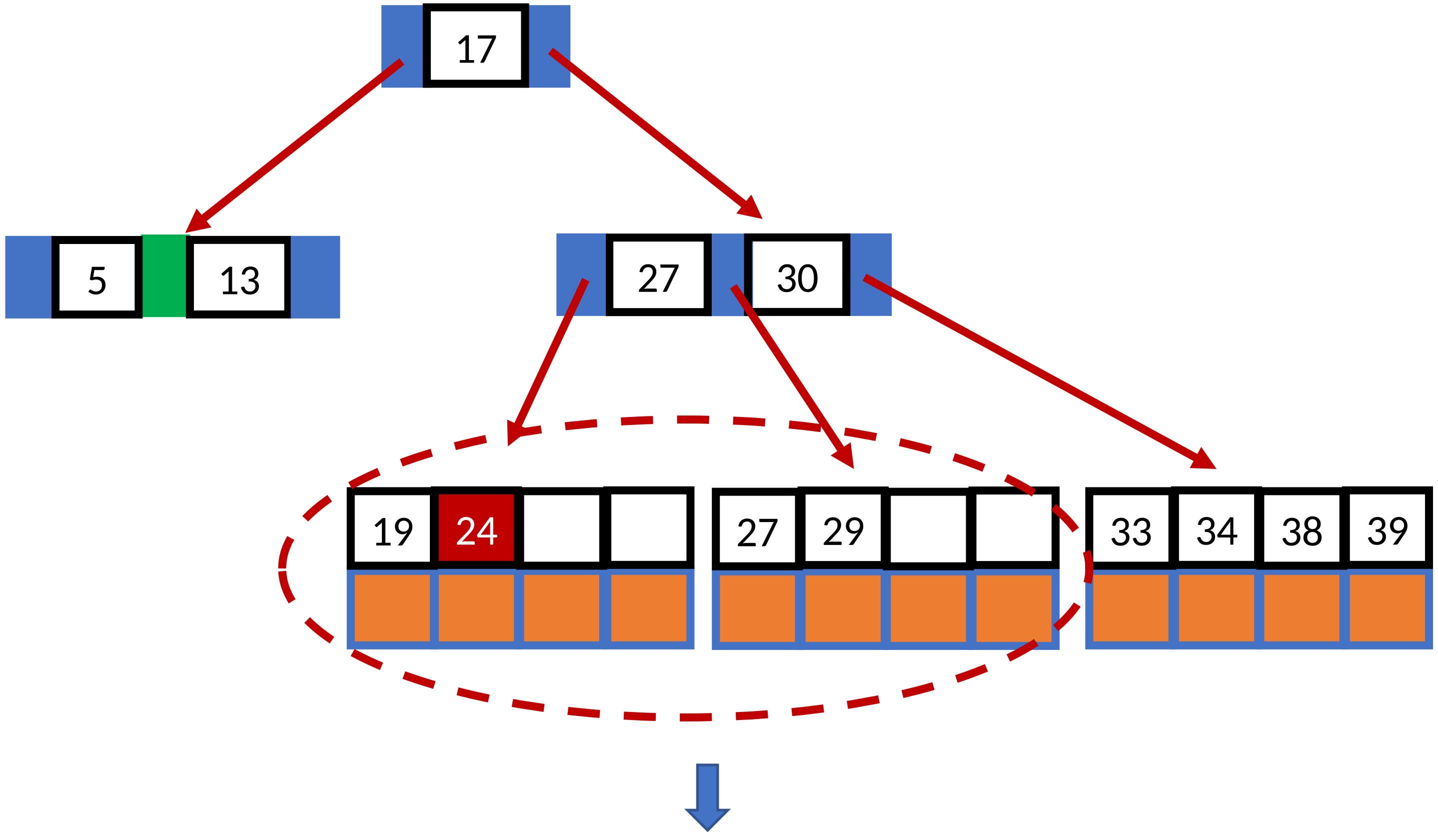
After deleting 20: Node becomes less than half-full, redistribute
→ New middle key (27) copied up

B-Tree Delete

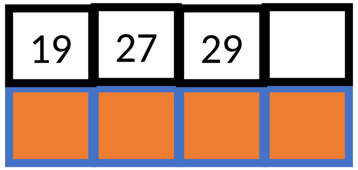


Delete 24: Must merge – sibling is also only half-full (cannot redistribute)

B-Tree Delete

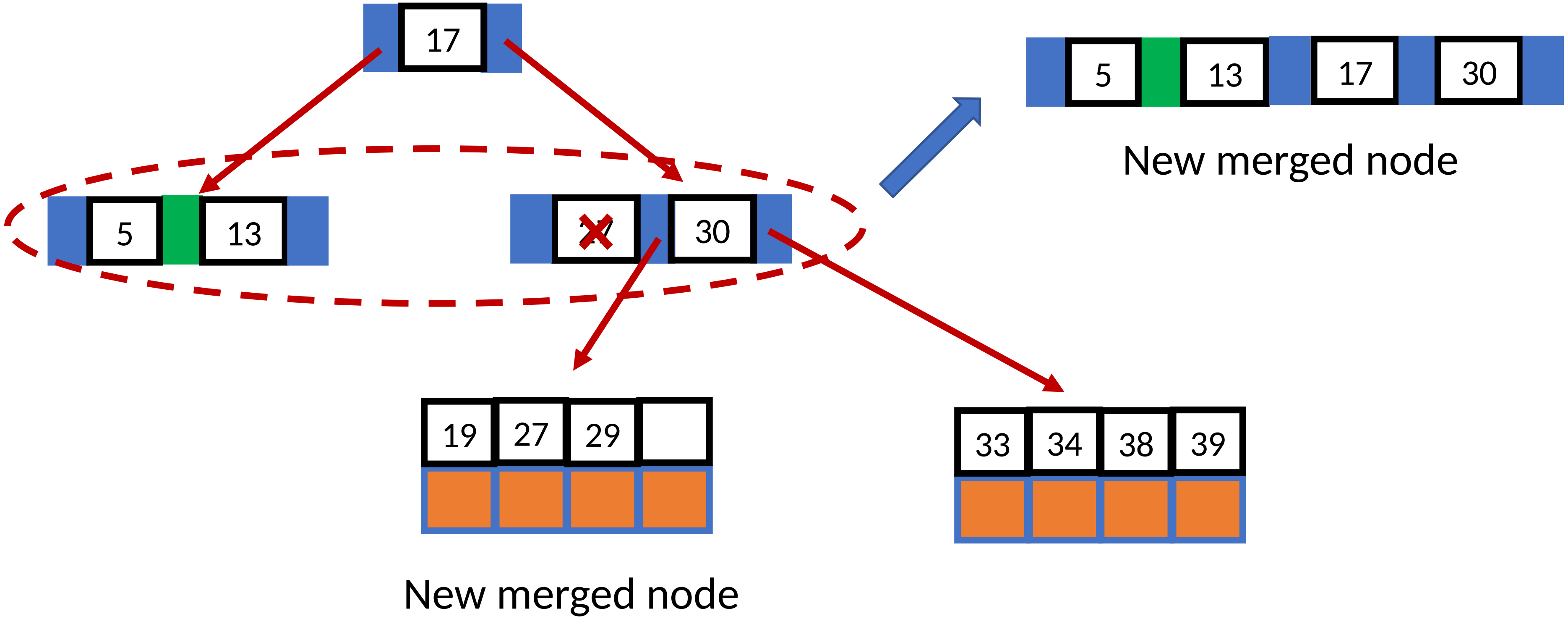


New merged node:



Then remove index entry (27) in parent node

B-Tree Delete



After Removing index entry (27) in parent node → parent node becomes less than half full, must merge again

- Pull down the index entry in parent node (17)

B-Trees in Practice

- Redistribution rarely implemented
- Deletions/merge are often not done right away
  - Marked as ‘deleted’ (tombstone) first, then periodically collected
    - E.g., at night when server load is low
- A stack can be used during traversal to avoid re-traversal
  - Remember the parent node for structural modification operations (SMOs): split/merge
- Inheritance used for implementing node types in memory
  - Root can be internal or leaf node
  - Base “node” class, inherited leaf and internal node types
- Need concurrency control itself
  - Lock based (relatively easy)
  - Lock-free (hard)

Mutexs, spin locks, etc. Called “latches” in database world

B-Tree Bulk Loading

Say we have a table and would like to create a new index

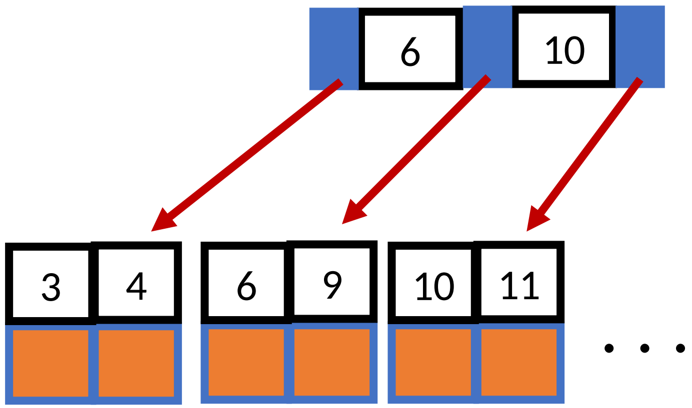
Alternative 1: create an empty tree and do inserts to it

- Easy to implement
- One record at a time
- Need to traverse the tree many times from root to the leaf level → high overhead

B-Tree Bulk Loading

Alternative 2: build the tree bottom-up

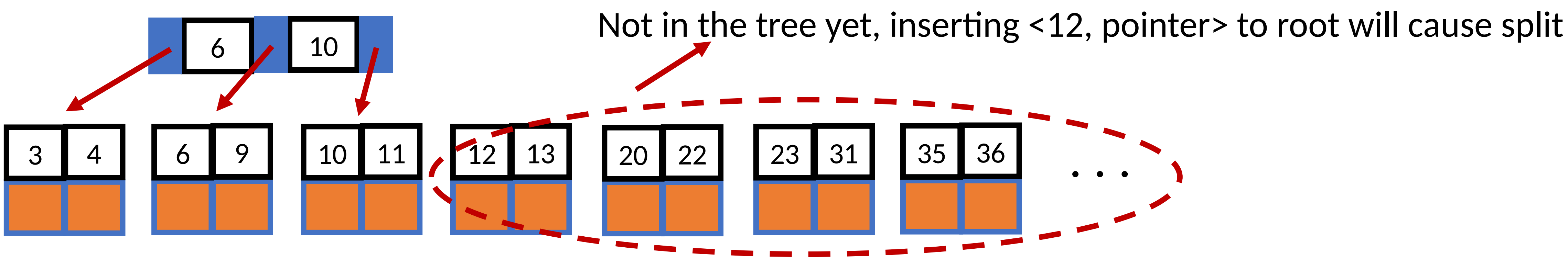
- Step 1: sort the data entries to be inserted to the tree
  - If data entries are RID pointers, no need to sort real data
  - Store these data entries in pages → these are the leaf pages
- Step 2: create an empty root node, point its left-most pointer to the first leaf node
- Step 3: add one entry in the root node for each leaf node, until the root node is full
  - Add <low key value on node, pointer to page>
  - Start from the first leaf node not in the tree

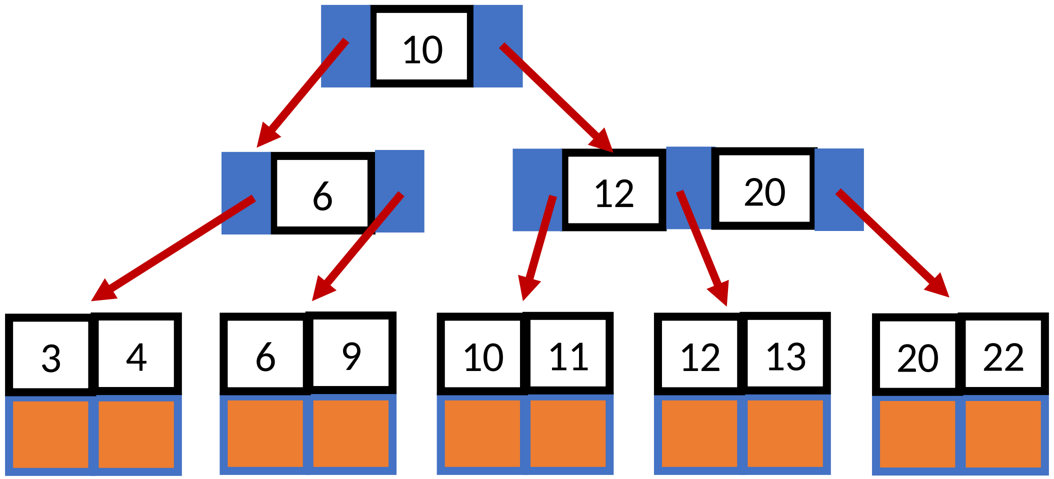


(e.g., assuming at most two keys per node)

B-Tree Bulk Loading

Alternative 2: build the tree bottom-up





- Step 4: split the root node if needed
- Step 5: keep adding keys to the right-most internal node just above the leaf level, split and grow tree as needed

. . .

B-Tree Key Compression

Desired: high fanout (child pointers per node to lower level)

- Fewer nodes (lower height) ➔ reduced I/O

Prefix key compression: Extract the common prefix and store only the unique suffix for each key

- Sorted keys in nodes tend to have the same prefix

Compute	Compression	Compile

➔

Comp	ute	ression	ile

- Suffix truncation: Store only the prefix of each key that is “enough” to route traffic in internal nodes

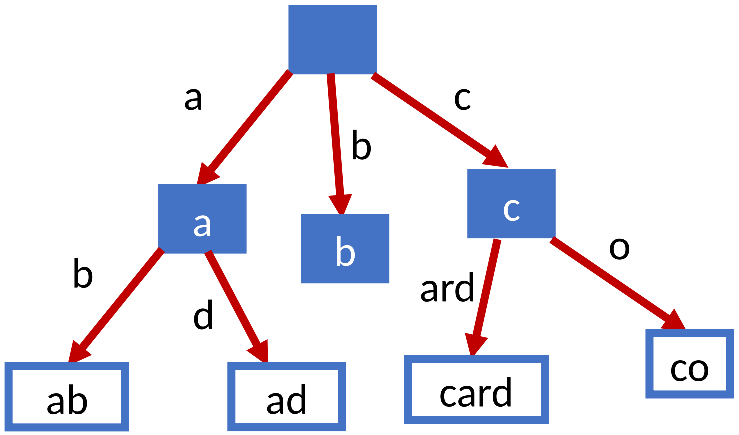
Dannon Yogurt	David Smith	Devarakonda Murthy

➔

Dan	Dav	Dev

Other Trees

- Radix tree (trie) and its variants
    - Keys represented implicitly by path



Keys:
- ab
- ad
- card
- co