B^+-Tree

Needed Capabilities

- quickly scan all data
- find exact (equality) matched data
- do efficient range searches
- do efficient joins
- do efficient insert/delete/updates
- For SSD/HDD resident data $\Rightarrow$ fewest I/O's

Fast Lookup

- Hash Tables
- Balanced Binary Trees
- B+ Tree is balanced binary tree for disks

What are we storing?

- In leaf nodes we can store
  - rows (record) for the primary index
    - need not be same as primary key
  - index_value, *rid* for a unique secondary index
    - e.g. for candidate keys, or even primary key
  - index_value, *rid*-list for secondary index

Primary Index

- Clustered, Sorted
- Choice of Primary Index should be done carefully
  - study the workload
  - joins and range searches are specially valuable
  - primary key can make a good primary index and should be first (default) choice

B+ tree nodes

- Optimized for disk-resident data
- Keep pages reasonably full
- B+ tree nodes are disk pages
- Two kinds
  - Internal
  - Leaf

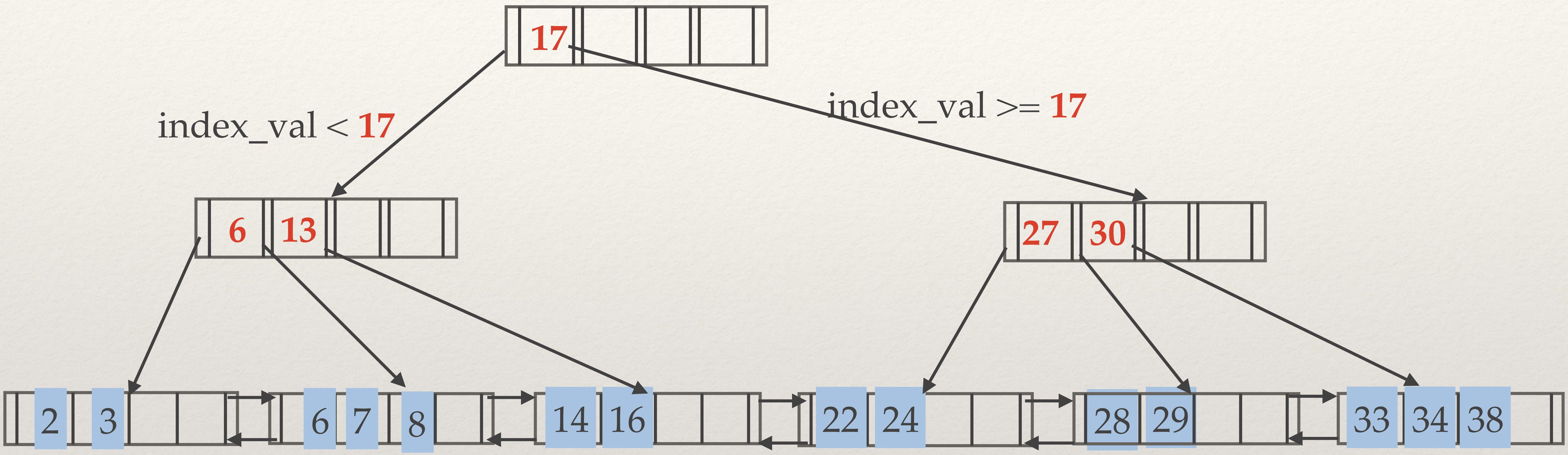
Internal Nodes

- Keep the index_values (keys) and "pointers" to pages
- (p_0, key_1, p_1, …, key_n, p_n)
- p_i is the pointer to subtree containing keys between key_i and key_i+1; usually > key_i and ≤ key_i+1
- p_0 points to the subtree all keys ≤ key_1
- p_n points to the subtree all keys > key_n

Leaf Node

- contains keys with data, such as
  - integrated record / rows i.e. keys are part of the record
  - set of (key, record) pairs
  - set of (key: list of *rids*)
  - set of (key, *rid*) pairs

B+ tree



- index_val = key
- 2 indicates record with key value of 2.

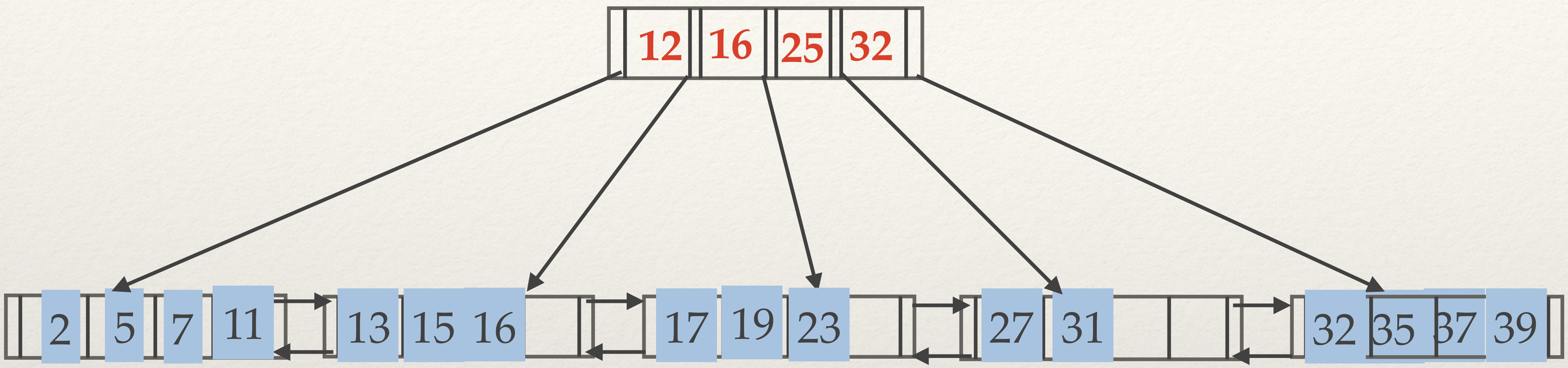
Properties

- Every node (except possibly root) is at least 1/2 full
  - if max entries is $2*d$, then must have at least d entries
  - d is called order of the tree
- Every leaf is at the same distance from root
  - tree is balanced to avoid bad cases

Properties (contd.)

- Search at $\log_F(N)$ cost
  - F is the fanout (at least *d*)
  - N is the number of leaf pages
- Insert / delete atleast $\log_F(N) + 1$, possibly a little more
  - 1 more I/O to write the updated leaf page
  - may need additional I/O to update interior nodes

Simple Example



- search for PI value = 7
- search for PI value = 18
- search for PI value > 31

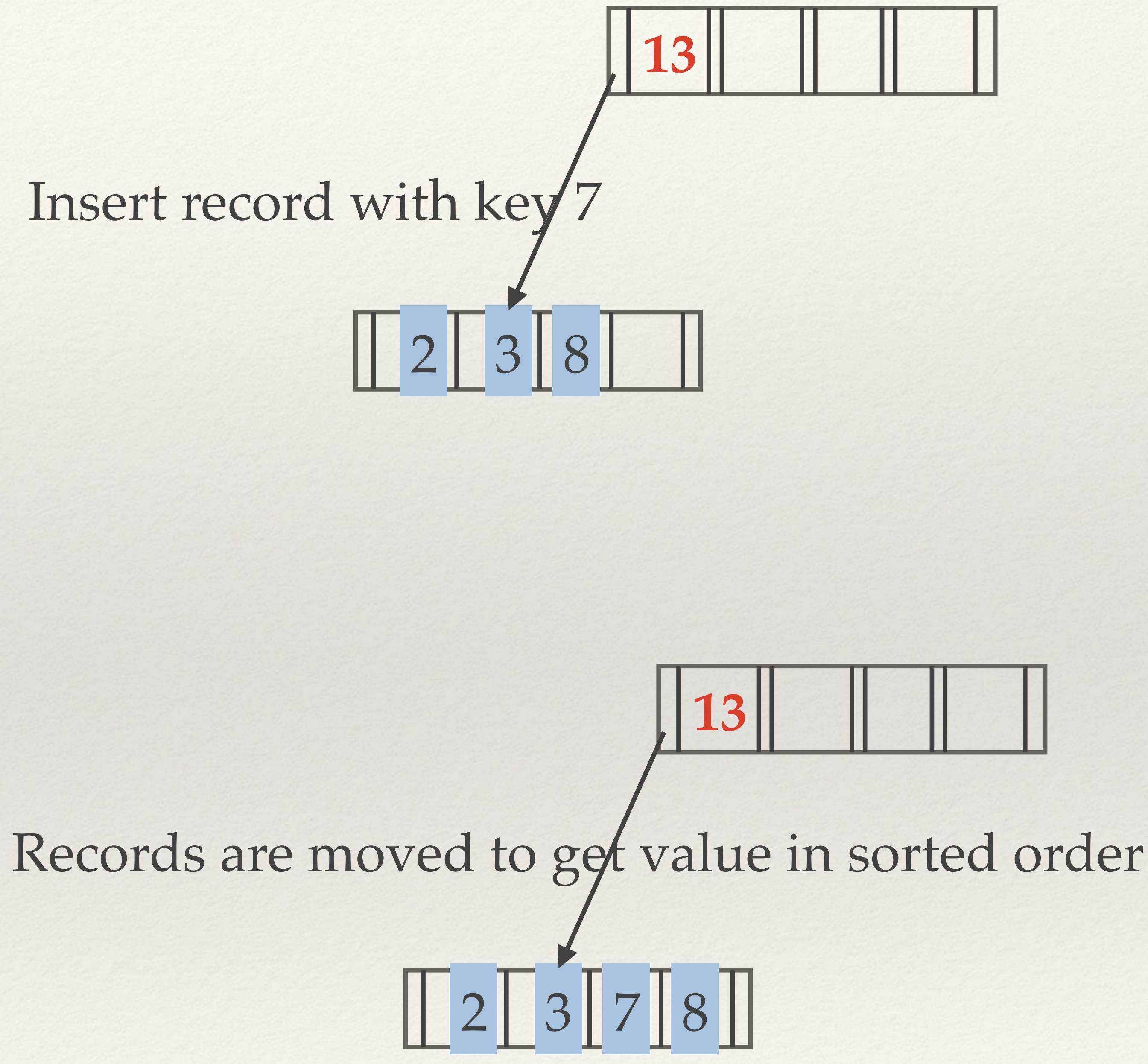
B+ trees

- For a 4K page, 8 byte keys, 8 byte “pointer”
- Typical Order(d) = 100, i.e. max($2d$) = 200
- average fanout = 133
- So at height of 4 we have 133^4 = 313M records
- height of 3: 133^3 = 2.3M records
- Often hold top levels in buffer pool
- Larger page sizes => more fanout

Inserting Data

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

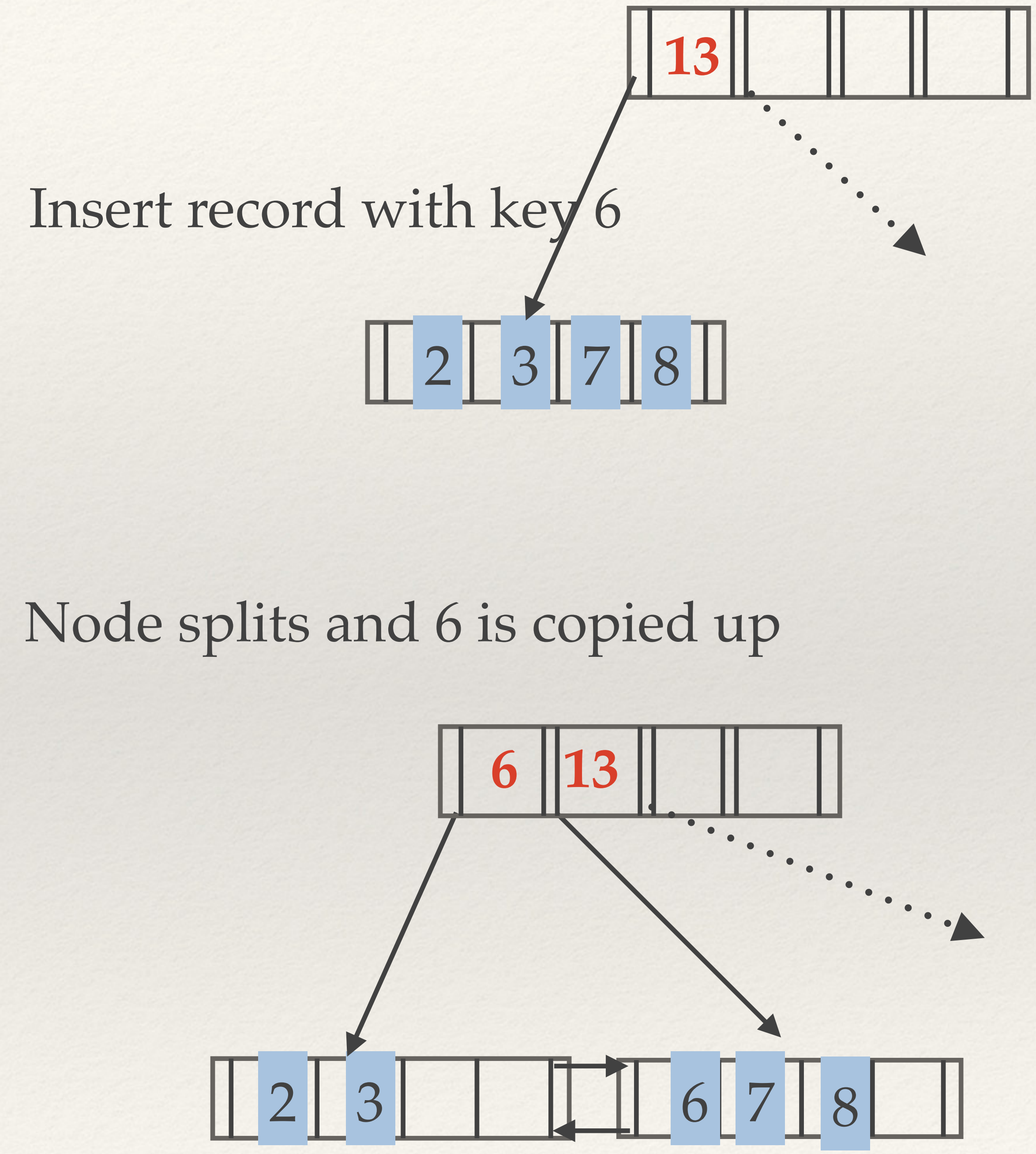
Insert into Leaf



- If there is space in page, records are moved to create a space in the proper order
- The record is then inserted in page

Insert into Leaf

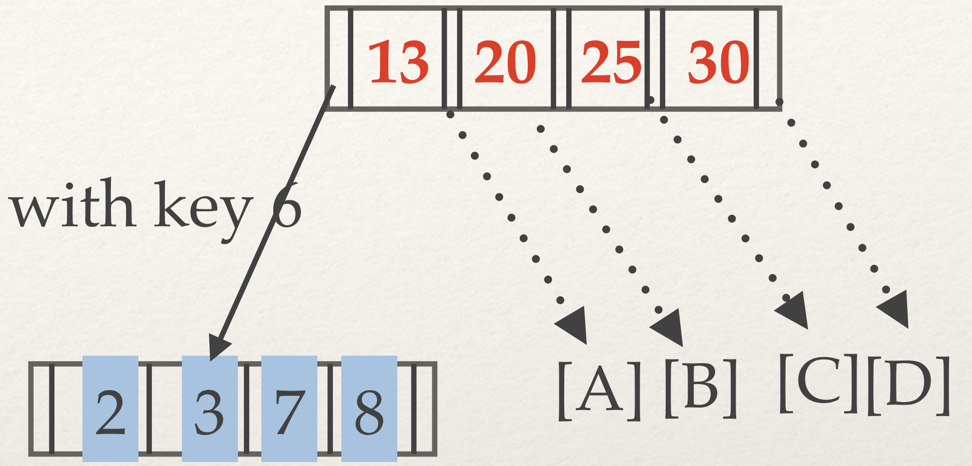
- Observe how minimum occupancy is guaranteed in both leaf and index page splits.
- The “middle valued” key is **copied-up**.
- It happens to be the value of inserted key (with record) in this case



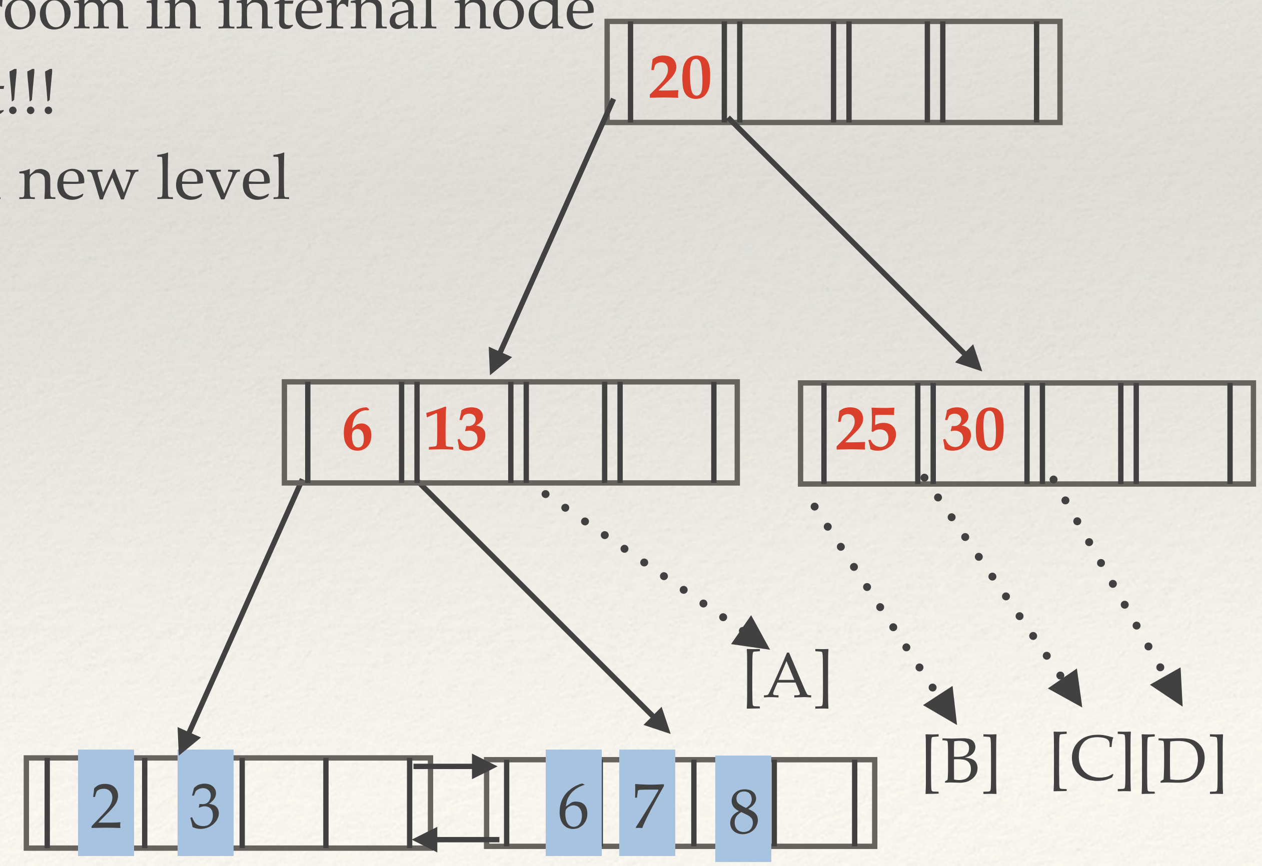
Insert into Internal Node

- Observe how minimum occupancy is guaranteed in both leaf and index page splits.
- If there is a split, the "middle valued" key is **pushed-up**
- New level may be needed

- Insert record with key 6



- Node splits and 6 is copied up
- No room in internal node
- Split!!!
- Add new level

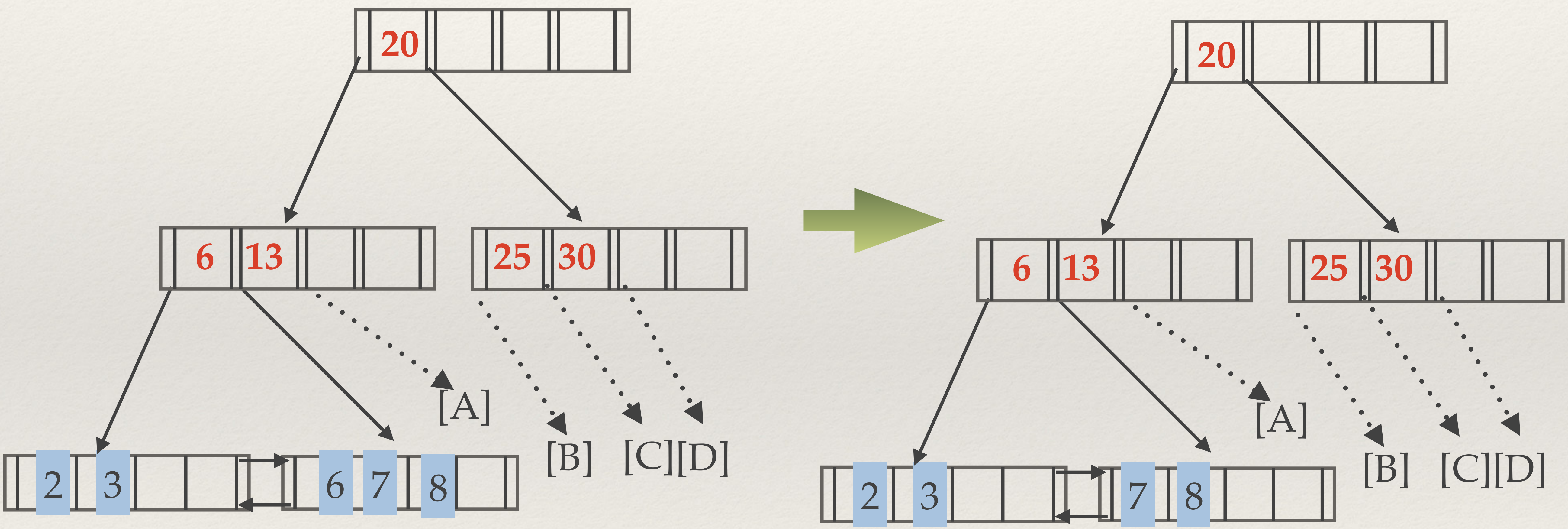


Deleting an entry from B+ Tree

- 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 re-distribute, borrowing from sibling (adjacent node with same parent as L).
    - If re-distribution fails, merge L and sibling.
- If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
- Merge could propagate to root, decreasing height.

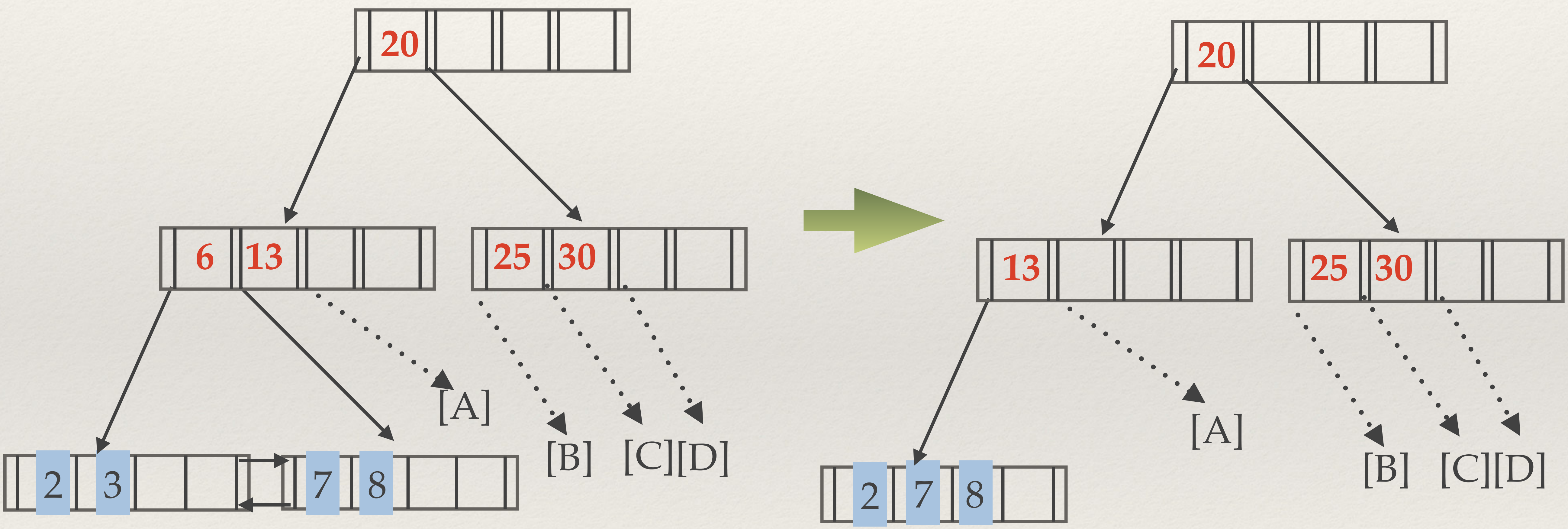
Delete record with Key 6

- Note that delete is not symmetric with insert



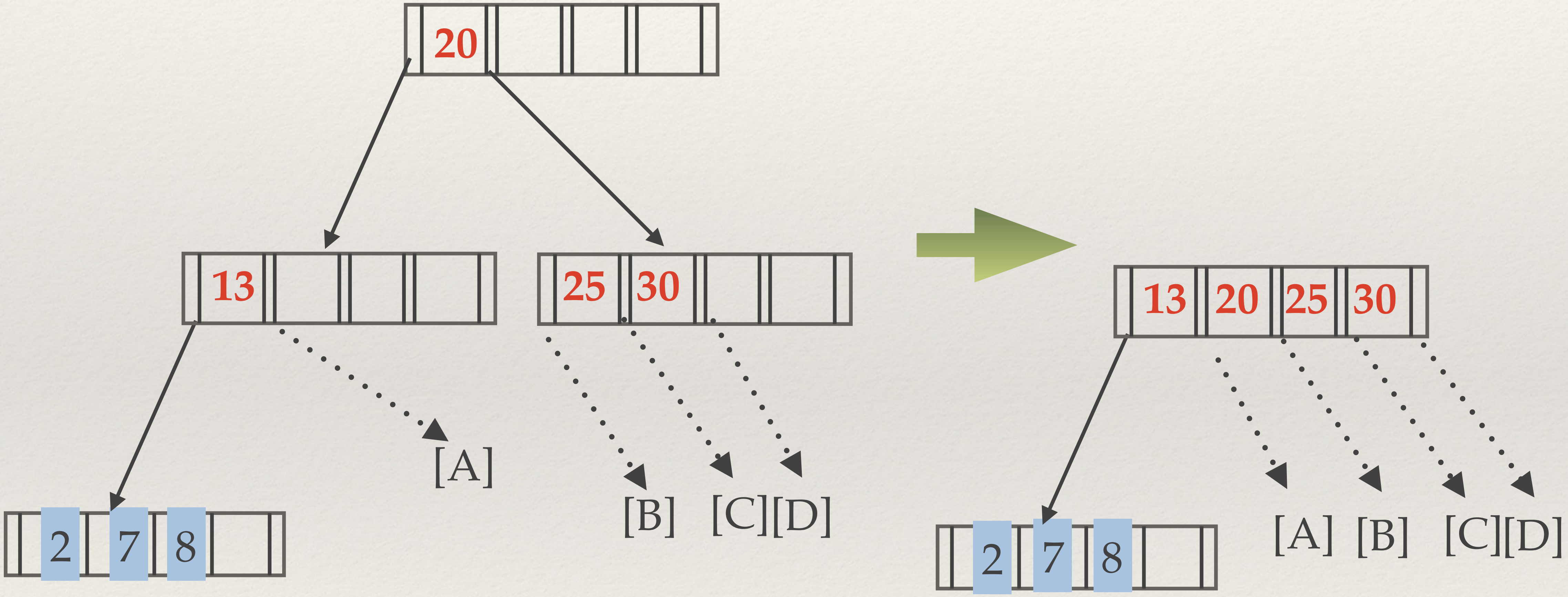
Delete record with Key 3

- No redistribution possible (both only 1/2 full)
- merge with sibling
- But what about the internal node with 13?



Delete record with Key 3

- No redistribution possible (both only 1/2 full)
- merge with sibling
- If propagated to root, remove a level

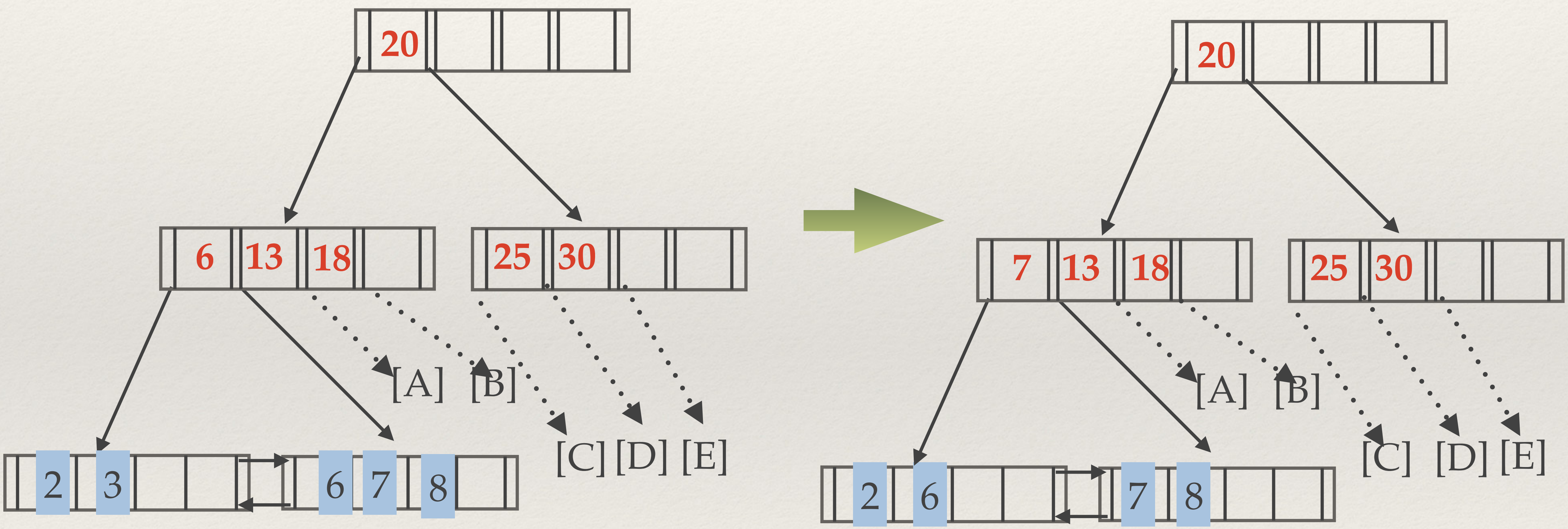


Deletion contd.

- What about a different order?

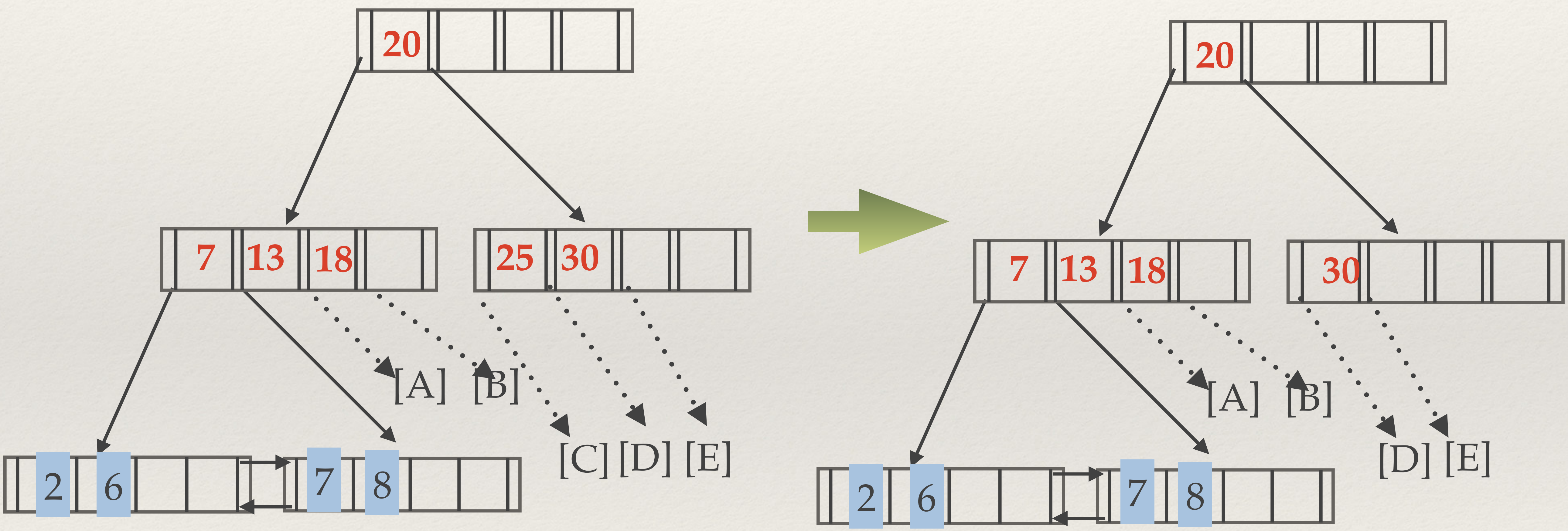
Delete record with Key 3

- Redistribute from sibling
- Adjust the key (copy-up from sibling here)



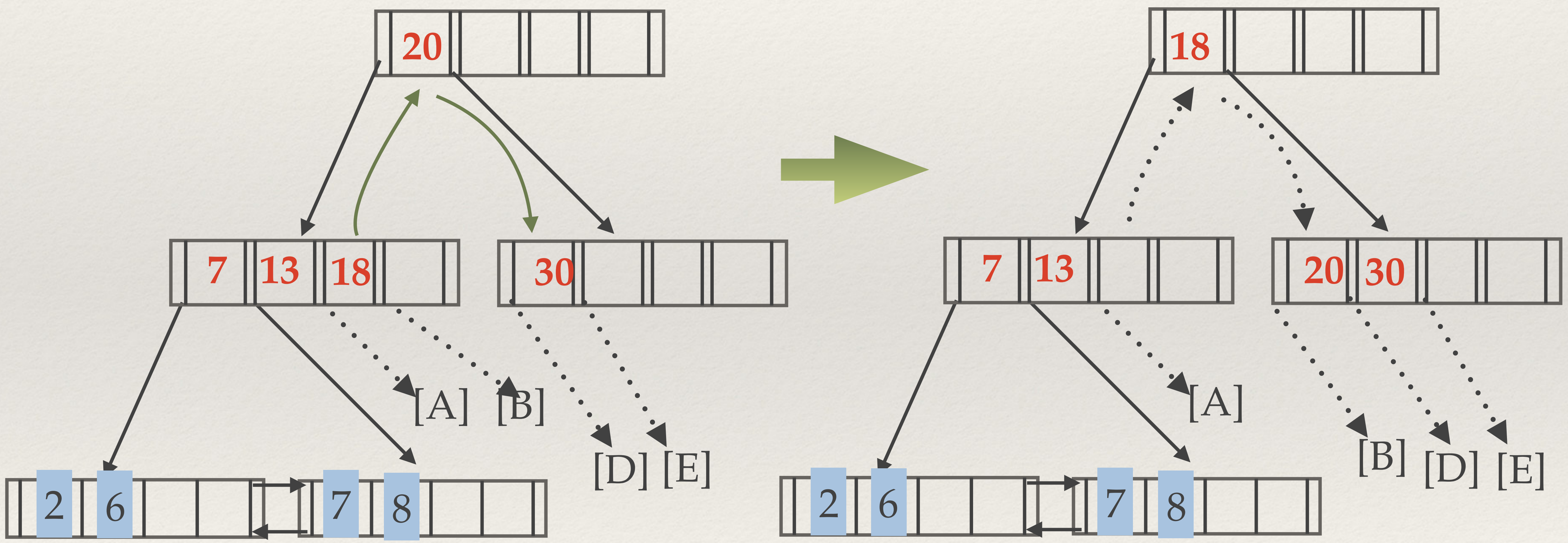
Delete all records from page [C]

- the right internal node is empty
- delete the key from internal
- borrow from sibling



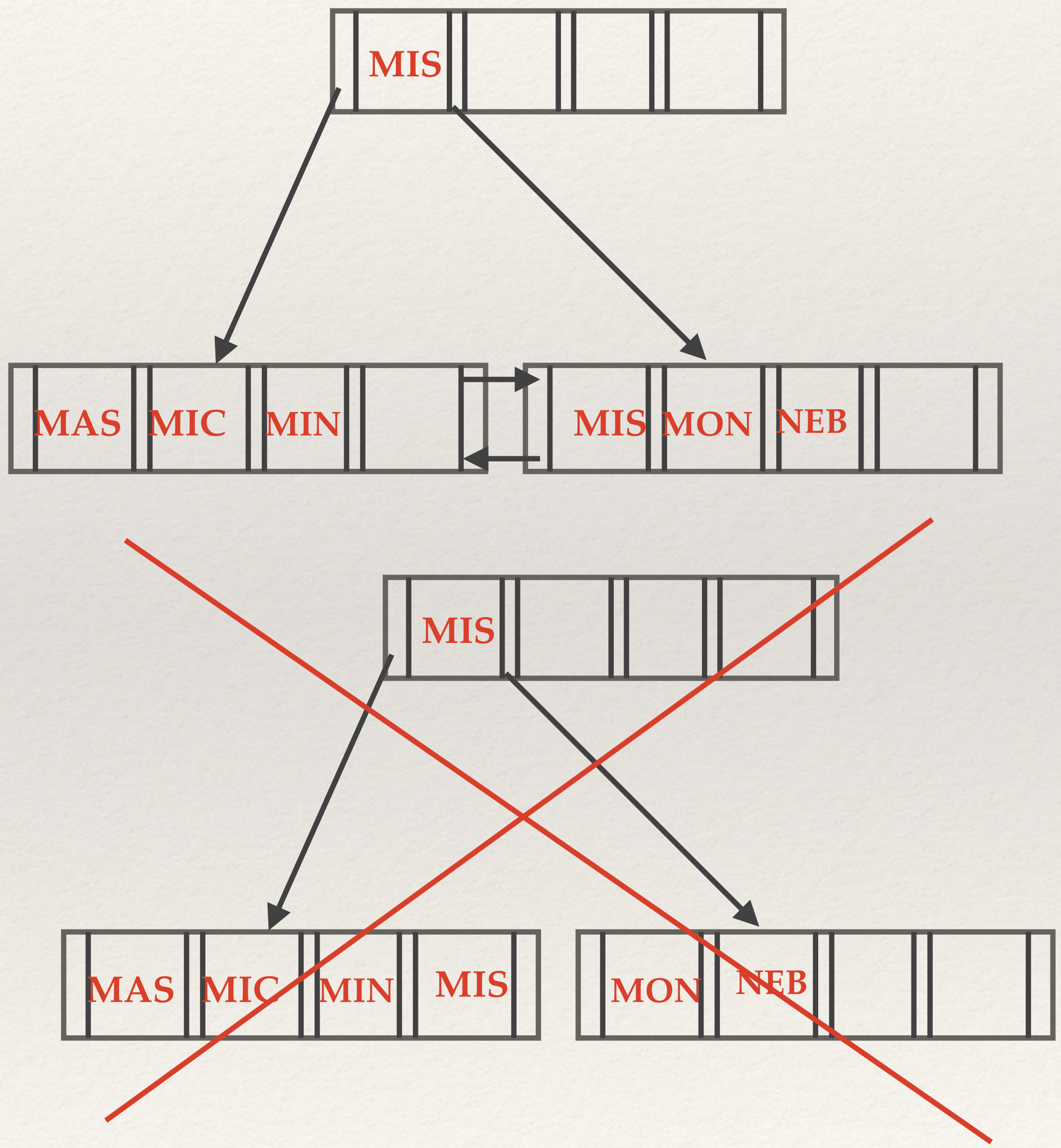
Borrowing from Sibling - Internal Node

- Entries are redistributed by **pushing through** the splitting entry in the parent node



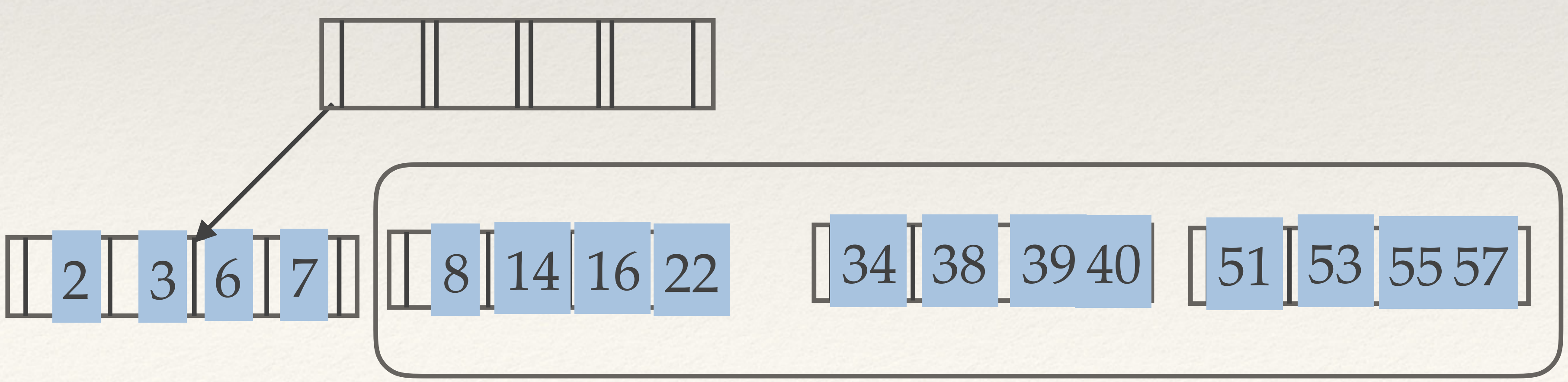
Prefix Key Compression

- Since the internal (index) nodes only "direct" search can just use prefix for longer strings, say first three letters of state names: MAS, MIC, MIS, MIS,MON, etc.
- Need to ensure that index entry is > all (uncompressed) key values to left
- Insert/Delete need additional care



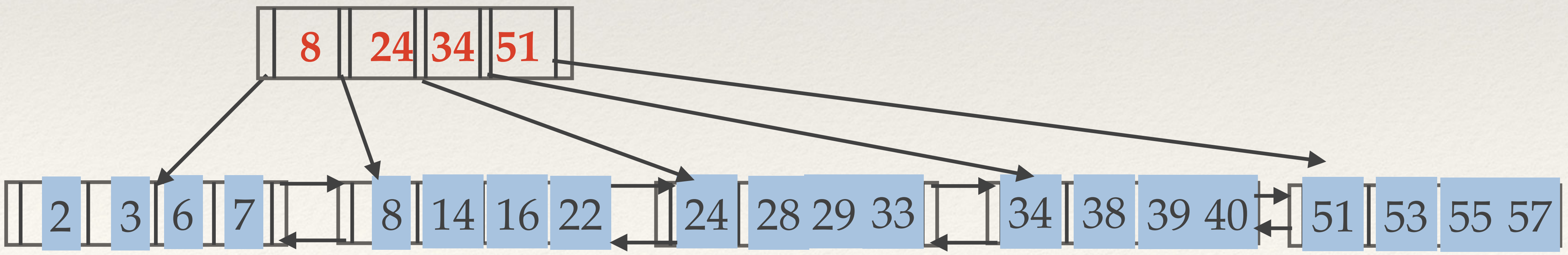
Bulk Loading into a B+ Tree

- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
- Bulk Loading can be done much more efficiently.
- Initialization: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.



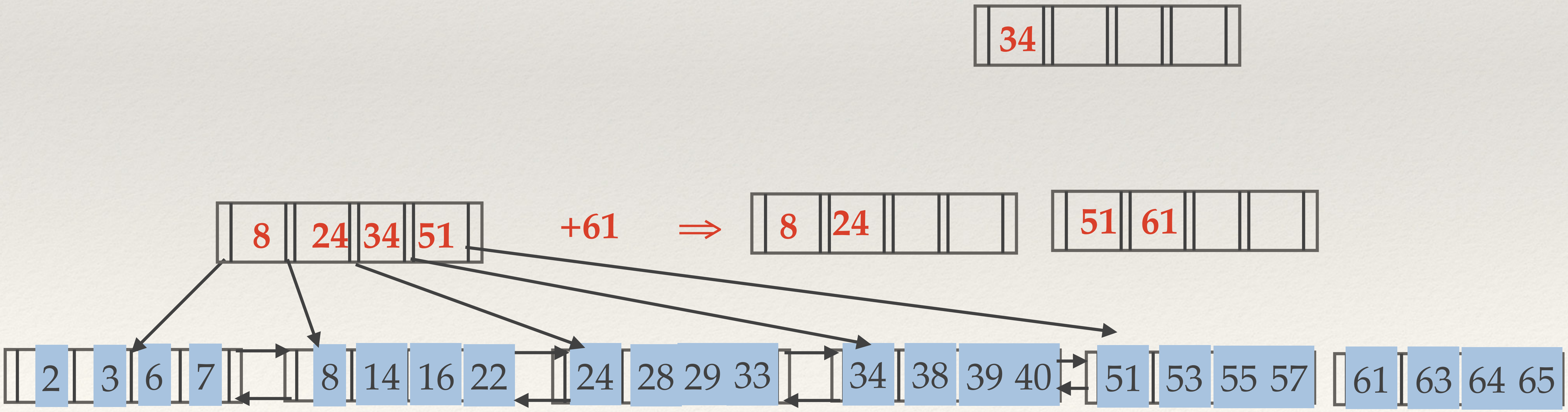
Bulk Loading into a B+ Tree

- Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)
- Much faster than repeated inserts, especially when one considers locking!



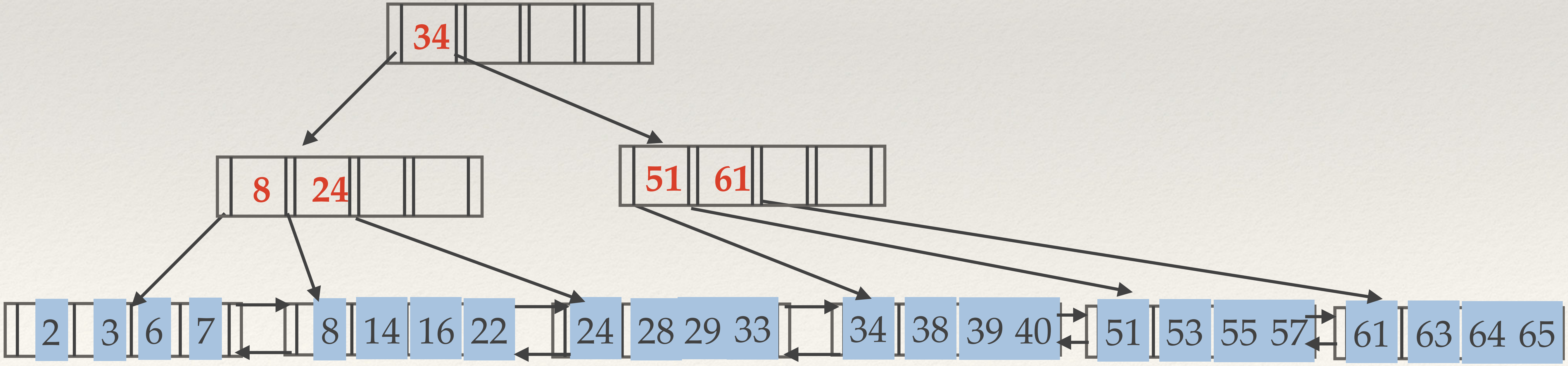
Bulk Loading into a B+ Tree

- Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)
- Makes for a compact (fuller) tree



Bulk Loading into a B+ Tree

- Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)
- Makes for a compact (fuller) tree



Bulk Loading

- Option 1: multiple inserts.
  - Slow
  - Does not guarantee sequential storage of leaves
- Option 2: Bulk Loading
  - Has advantages for concurrency control.
  - Fewer I/Os during build.
  - Leaves likely to be stored sequentially
  - Can control “fill factor” on pages

How many pointers?

- “order” or minimum number of pointers on node is usually replaced with “page half full” criterion
- leaves may contain variable number of entries due to variable size records

Review

- B+ Trees are ideal data structure for most work loads
- Bulk loading makes it faster and that's what is done as much as possible
- Widely used and optimized, including Teradata®
- Heap Files may be good if no equality or range search