Indexing 3

Instructor: Beom Heyn Kim

beomheynkim@hanyang.ac.kr

Department of Computer Science

Overview

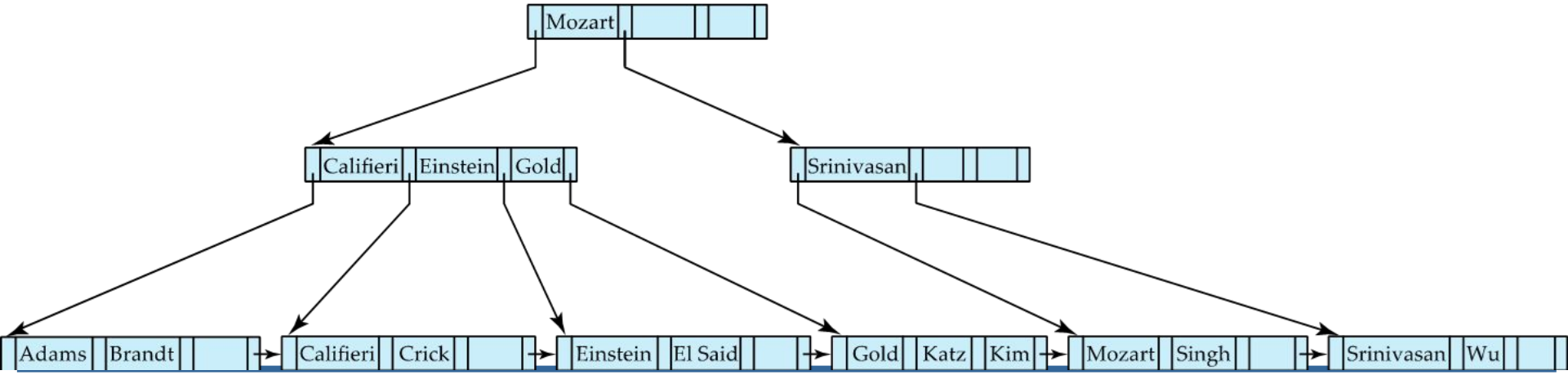
- B+-Tree Index Files II
- Assignments

Queries on B+-Trees

function *find*(*v*)

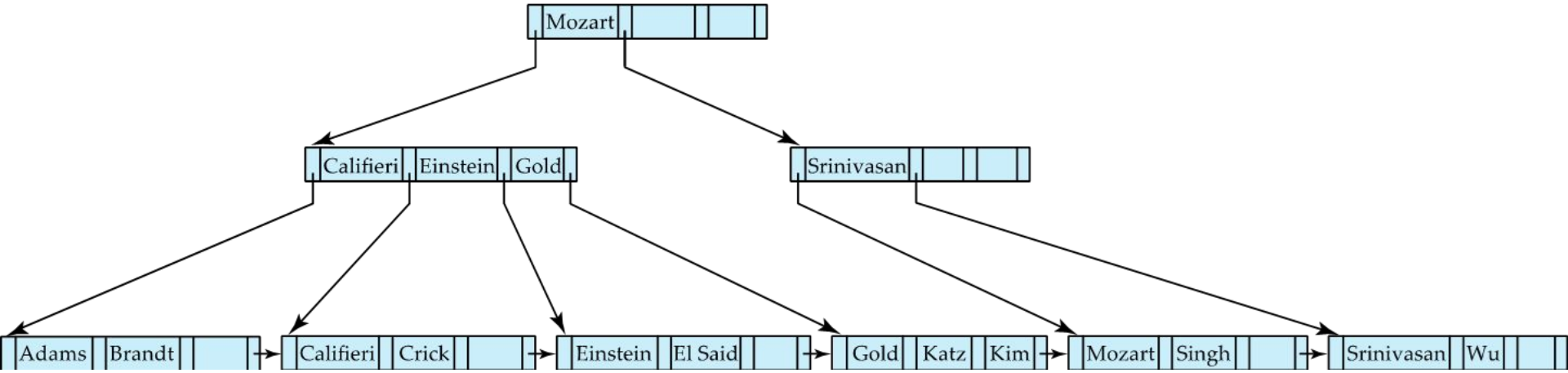
1. *C=root*
2. **while** (C is not a leaf node)
   1. Let *i* be least number s.t. $V \leq K_i$.
   2. **if** there is no such number *i then*
   3. Set *C* = *last non-null pointer in C*
   4. **else if** (v = C.K_i) Set C = P_{i+1}
   5. **else set** *C* = C.P_i
3. **if** for some *i*, $K_i = V$ **then** return C.P_i
4. **else** return null /* no record with search-key value *v* exists. */

Try:
1. find(Crick)
2. find(Wu)
3. find(Lee)



Queries on B+-Trees (Cont.)

- **Range queries** find all records with search key values in a given range
  - See book for details of **function** *findRange*(*lb, ub*) which returns set of all such records
  - Real implementations usually provide an iterator interface to fetch matching records one at a time, using a *next*() function



Queries on B+-Trees (Cont.)

- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and n is typically around 100 (40 bytes per index entry).
- With 1 million search key values and n = 100
  - at most $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup traversal from root to leaf.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

Non-Unique Keys

- If a search key a_i is not unique, create instead an index on a composite key (a_i, A_p), which is unique
  - A_p could be a primary key, record ID, or any other attribute that guarantees uniqueness
- Search for $a_i = v$ can be implemented by a range search on composite key, with range (v, -∞) to (v, +∞)
  - -∞ denotes the smallest and +∞ denotes the largest possible values of A_p
- But more I/O operations are needed to fetch the actual records
  - If the index is clustering, all accesses are sequential
  - If the index is non-clustering, each record access may need an I/O operation

Updates on B+-Trees

- Insertion and deletion are more complicated than lookup
- Nodes never become too large or too small.
  - Insertion may **split** a node
  - Deletion may **coalesce** nodes (if a node becomes too small (fewer than $\lceil n/2 \rceil$ pointers)
- Overview of Insertion
  - Using the same technique as for lookup from find, find the leaf node where the inserted value would appear
  - Insert an entry in the leaf node, maintaining search keys in order
- Overview of Deletion
  - Using the same technique as for lookup from find, find the entry/entries to be deleted
  - Remove the entry in the leaf node
  - Entries to the right of the deleted entry are shifted left by one position (no gap allowed between entries)

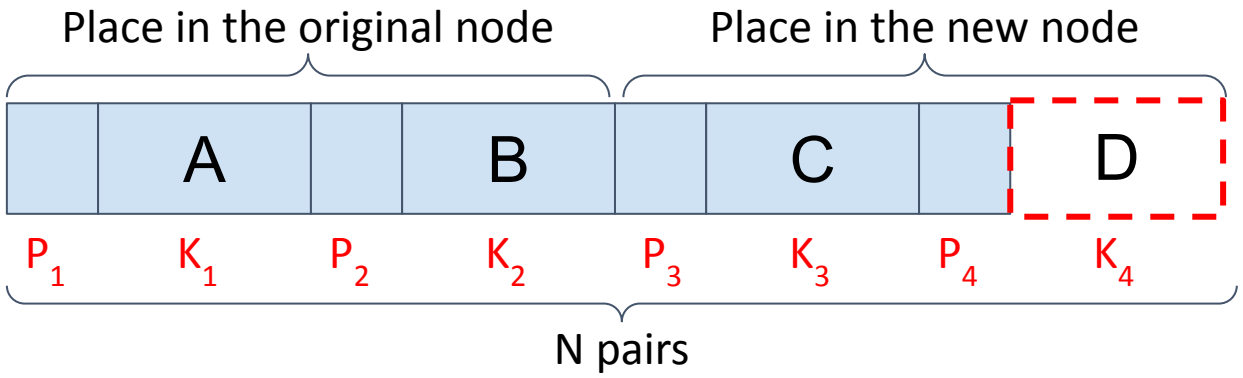
Updates on B+-Trees: Insertion

- Assume record already added to the file. Let
  - *pr* be pointer to the record, and let
  - v be the search key value of the record
- Find the leaf node in which the search-key value would appear
  - If there is room in the leaf node, insert (v, *pr*) pair in the leaf node
  - Otherwise, split the node (along with the new (*v, pr*) entry) as discussed in the next slide, and propagate updates to parent nodes.

Updates on B+-Trees: Insertion (Cont.)

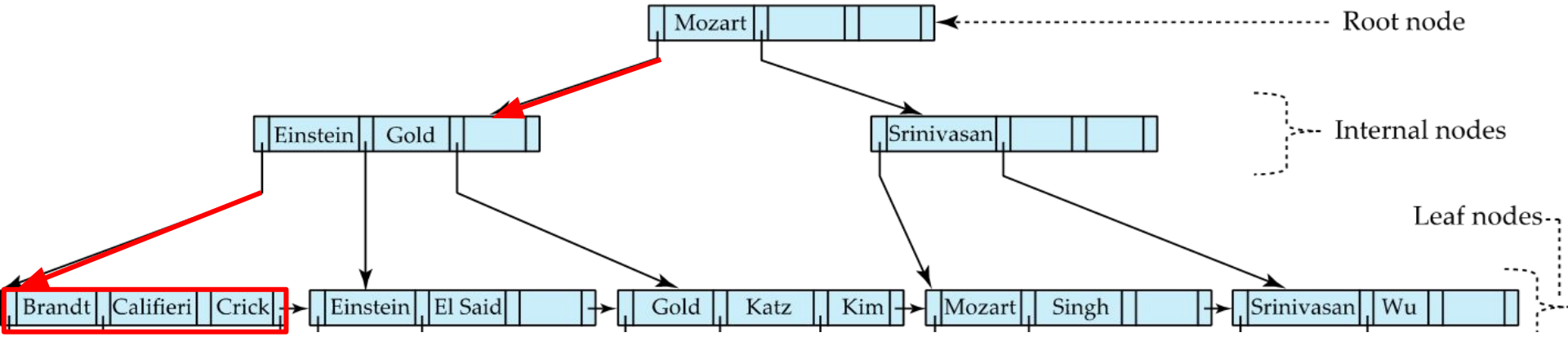
- Splitting a leaf node:
  - take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order.
  - Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be *p,* and let *k* be the least key value in *p.* Insert (*k,p*) in the parent of the node being split
  - If the parent is full, split it and **propagate** the split further up.
- Splitting of nodes proceeds upwards till a node that is not full is found.
  - In the worst case the root node may be split increasing the height of the tree by 1.

Splitting for inserting D:



B+-Tree Insertion

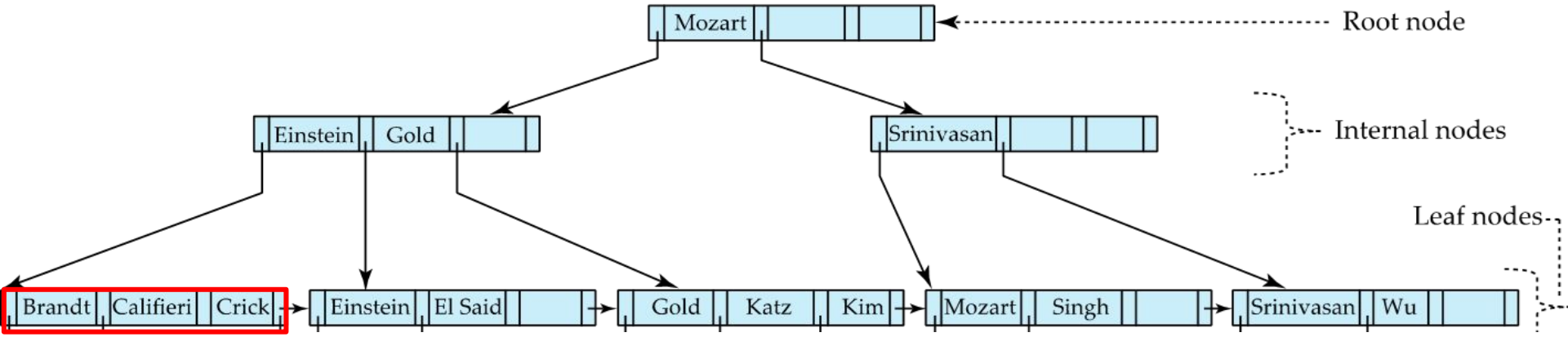
B$^+$-Tree before insertion of "Adams"



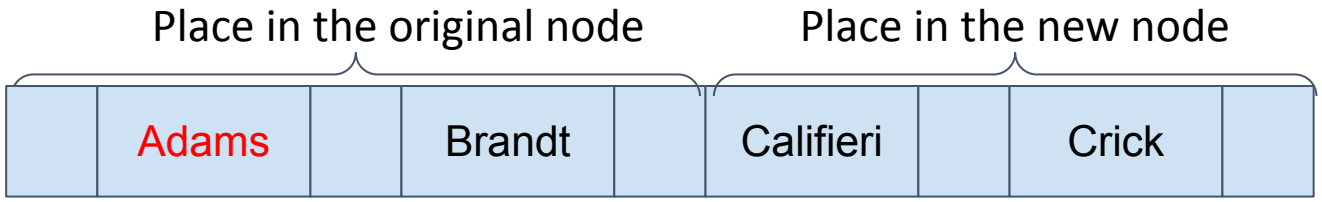
Node to store Adams

B+-Tree Insertion

B+-Tree before insertion of “Adams”

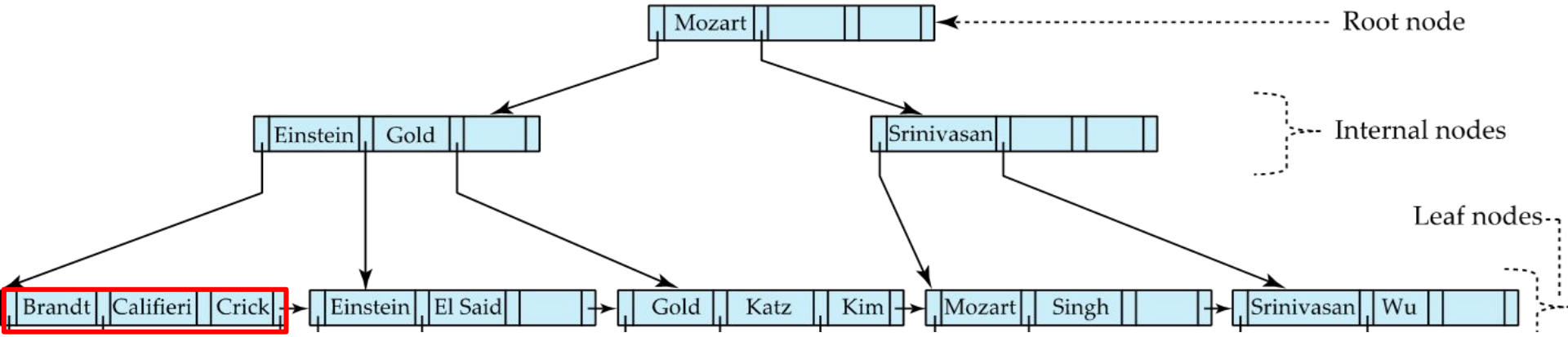


Need to split the node



B+-Tree Insertion

B+-Tree before insertion of “Adams”



Need to split the leaf node

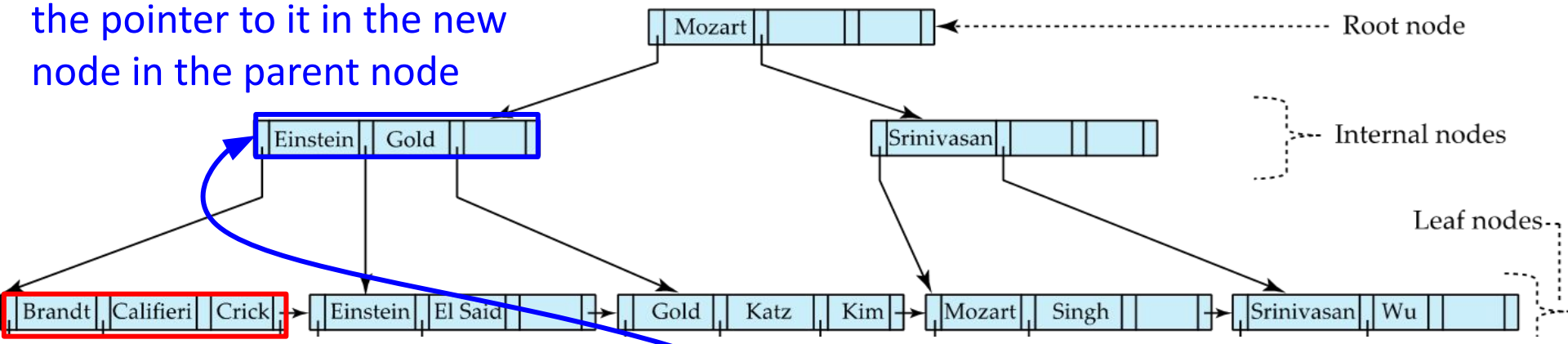
new node

original node

B+-Tree Insertion

B+-Tree before insertion of “Adams”

Insert the least search-key and the pointer to it in the new node in the parent node

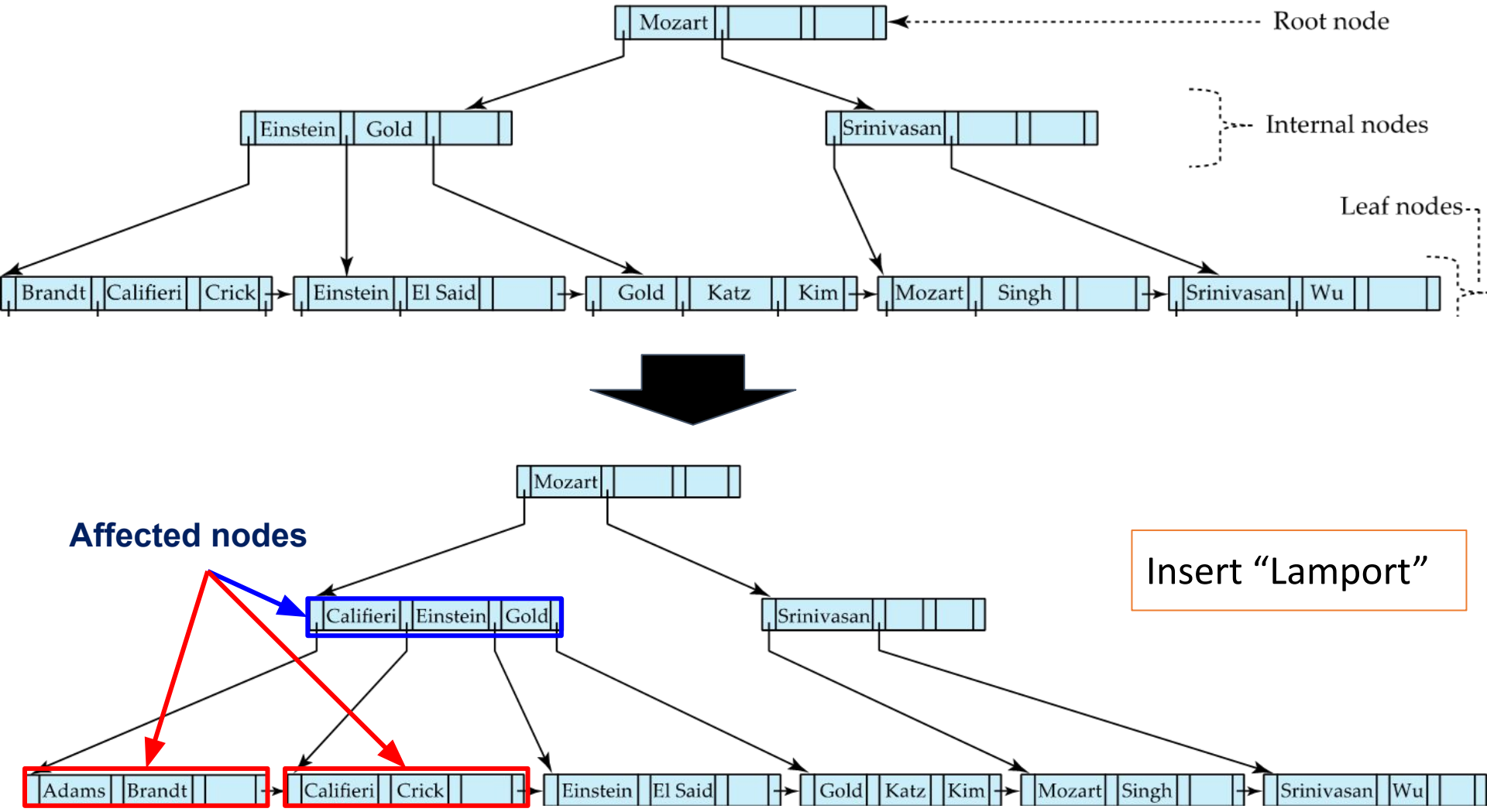


Need to split the leaf node

new node

original node

B+-Tree Insertion

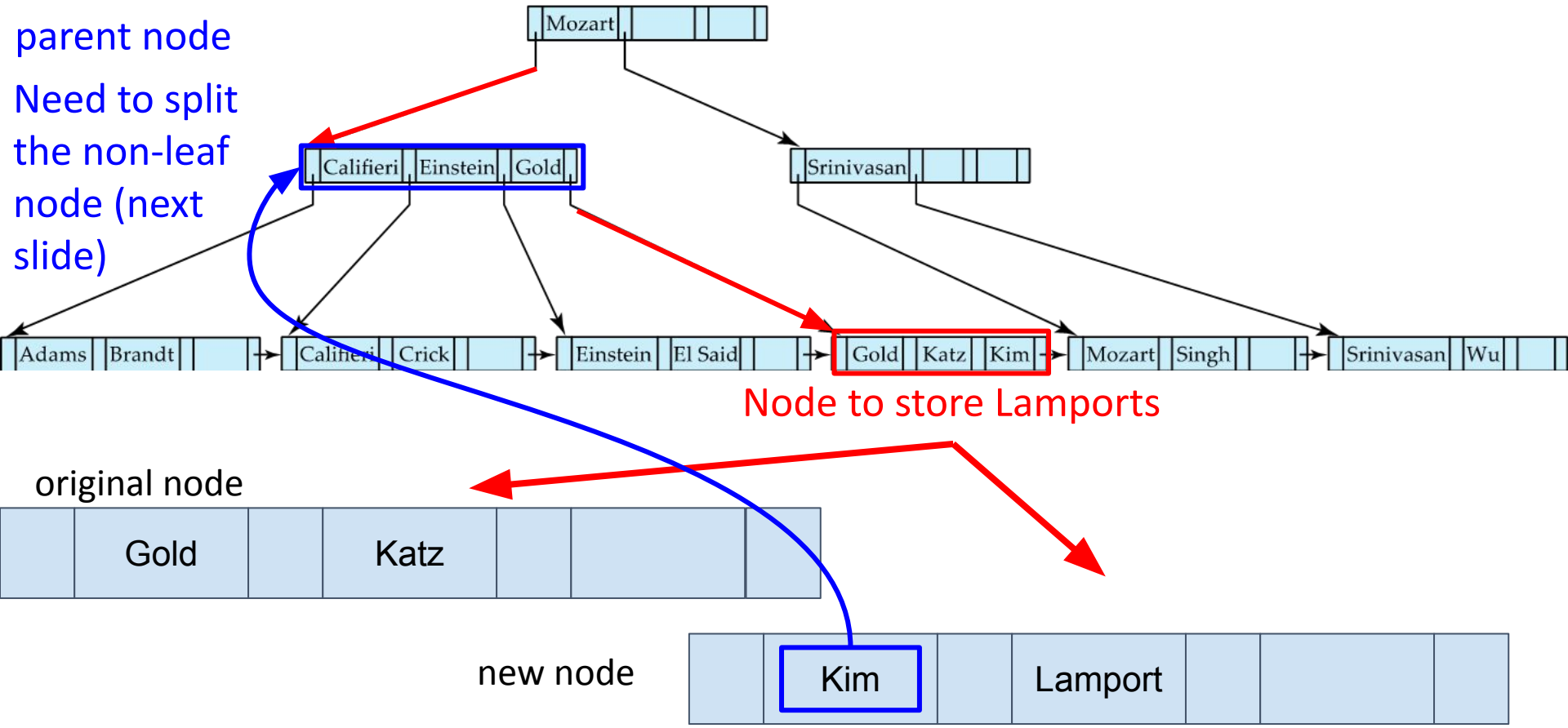


B⁺-Tree before and after insertion of “Adams”

B+-Tree Insertion

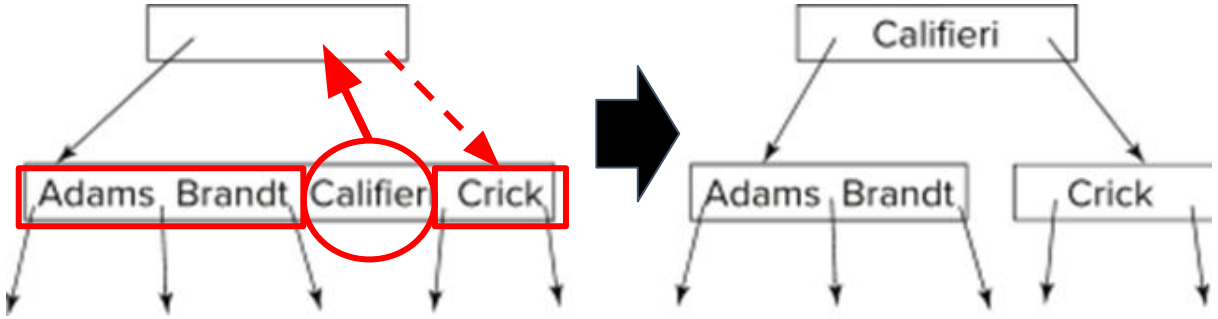
Insert the least search-key and the pointer to it in the new node in the parent node

Need to split the non-leaf node (next slide)



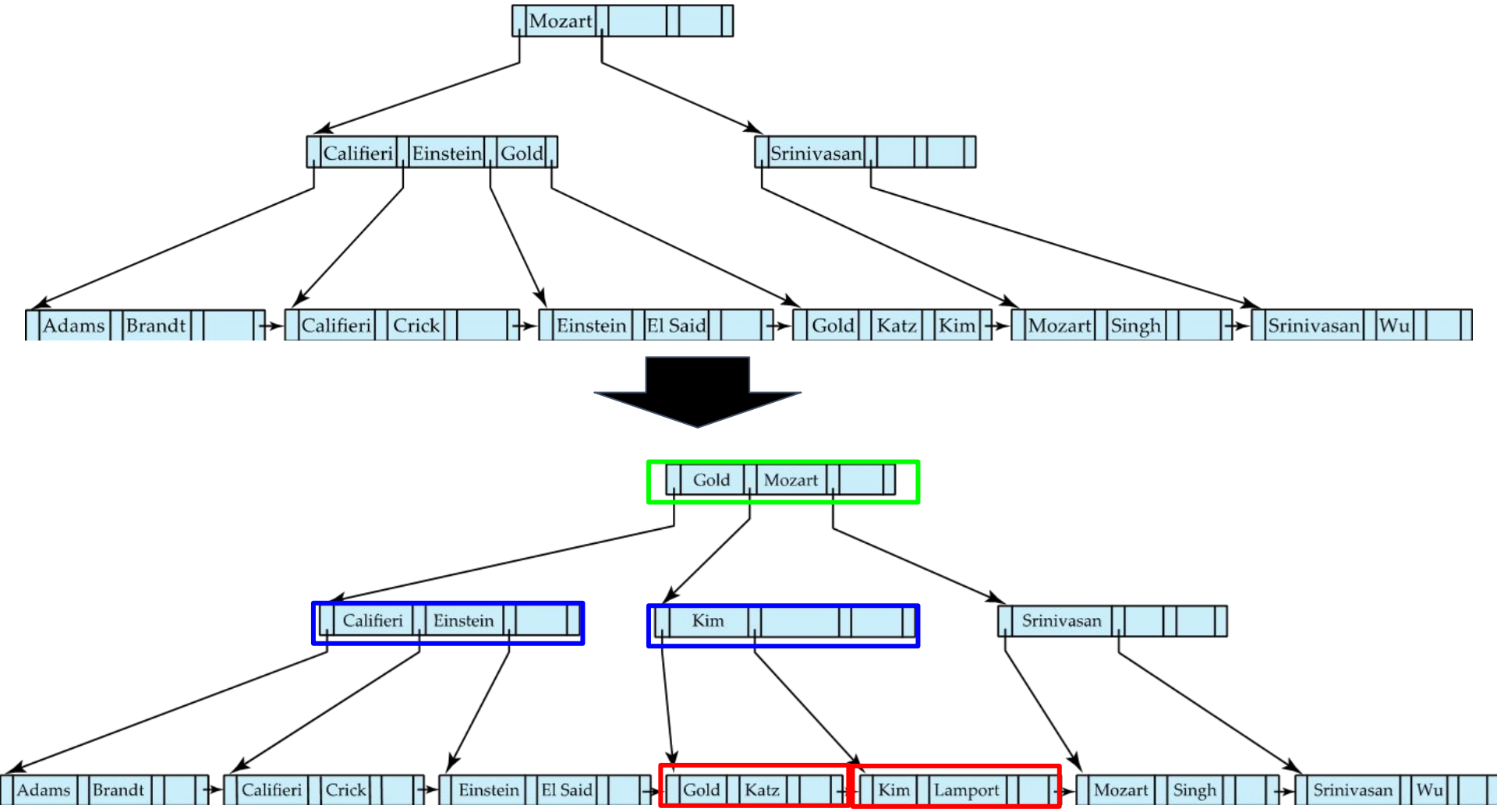
Insertion in B+-Trees (Cont.)

- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
  - Copy N to an in-memory area M with space for n+1 pointers and n keys
  - Insert (k,p) into M
  - Copy $P_1, K_1, \ldots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node N
  - Copy $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \ldots, K_n, P_{n+1}$ from M into newly allocated node N'
  - Insert $(K_{\lceil n/2 \rceil}, N')$ into parent N
- Example



- **Read pseudocode in book!**

B+-Tree Insertion



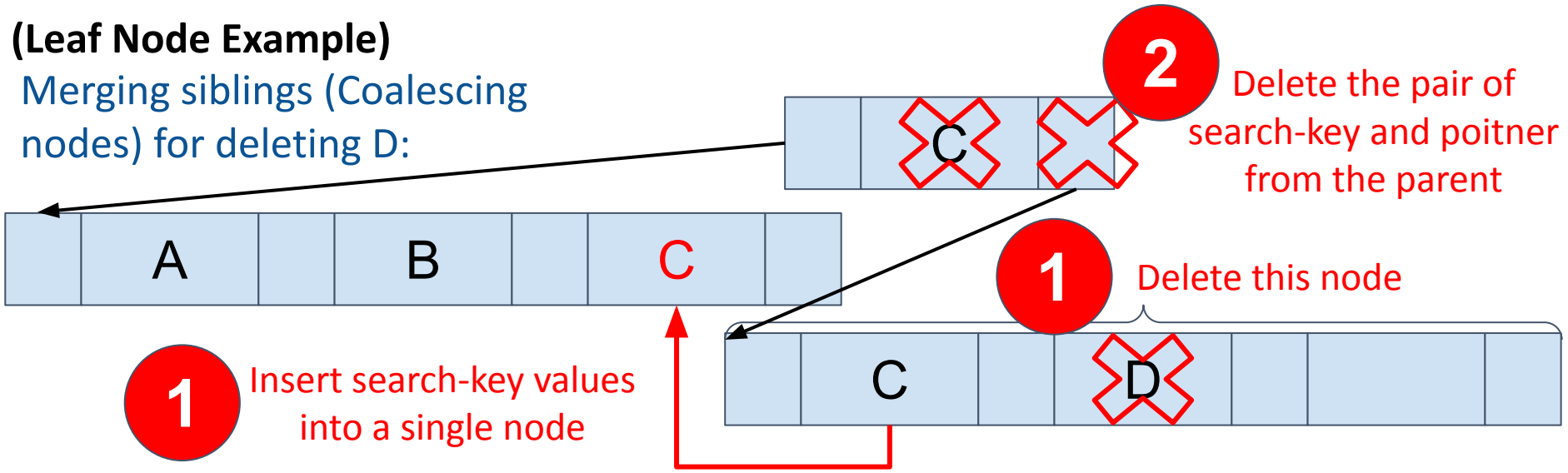
B$^+$-Tree before and after insertion of "Lamport"

Updates on B+-Trees: Deletion

- Assume record already deleted from file. Let *V* be the search key value of the record, and *Pr* be the pointer to the record.
  - Remove (*Pr, V*) from the leaf node
  - If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings***:
    1. Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
    2. Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.

(Leaf Node Example)

Merging siblings (Coalescing nodes) for deleting D:

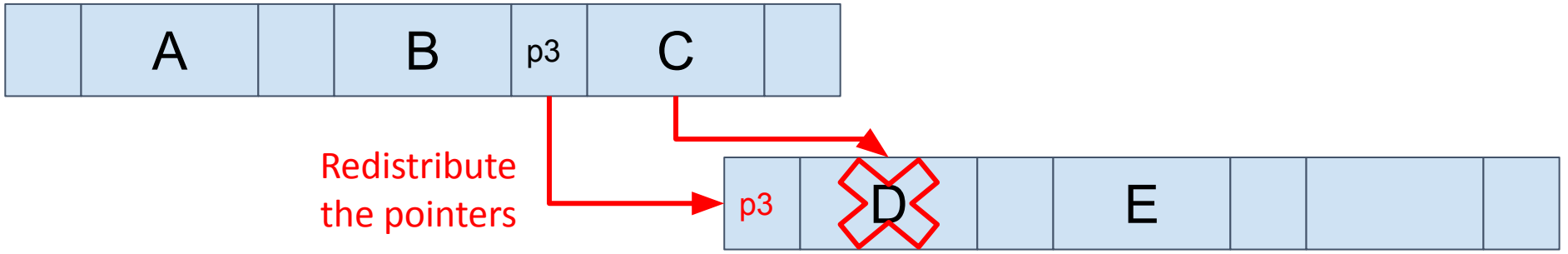


Updates on B+-Trees: Deletion

- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
  - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards till a node which has ⌈*n*/2⌉ or more pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.
- **Read pseudocode in book!**

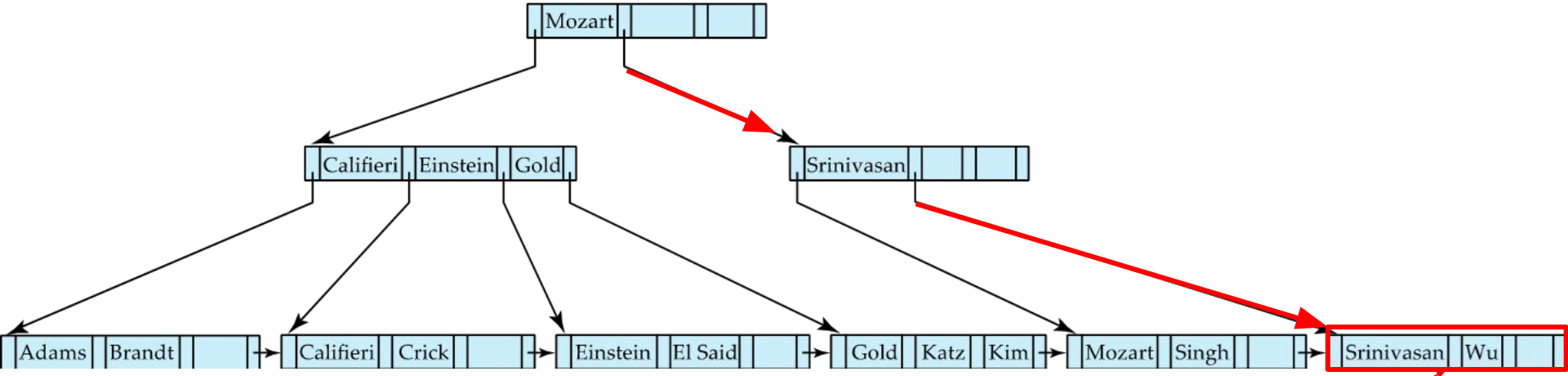
(Leaf Node Example)

Redistributing entries for deleting D:



Examples of B+-Tree Deletion

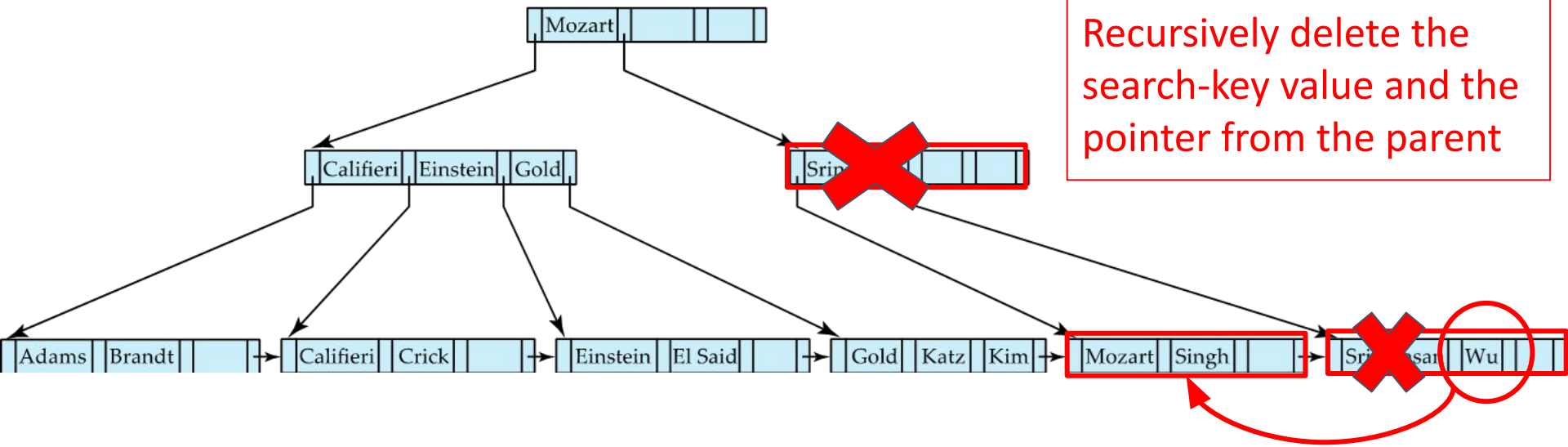
Before deleting “Srinivasan”



After deleting Srinivasan, the node becomes underfull

Examples of B+-Tree Deletion (Cont.)

Before deleting "Srinivasan"

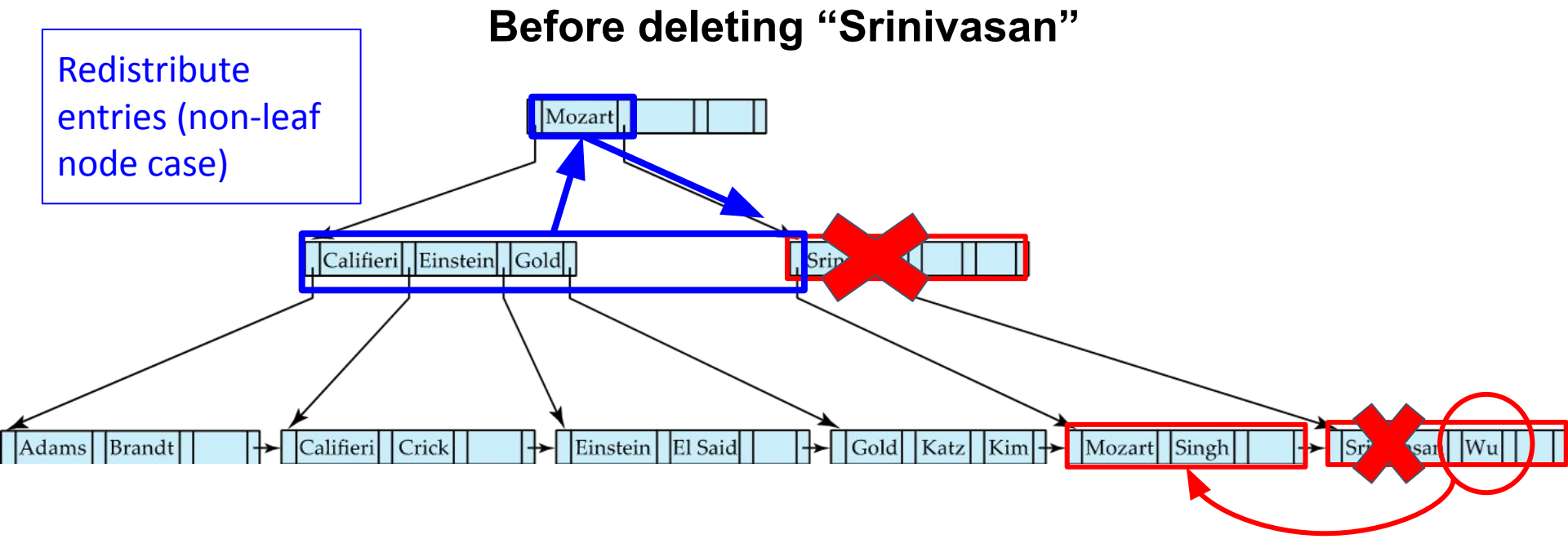


Recursively delete the search-key value and the pointer from the parent

Merge: Inserting Wu into the sibling node and delete the node used to store Srinivasan

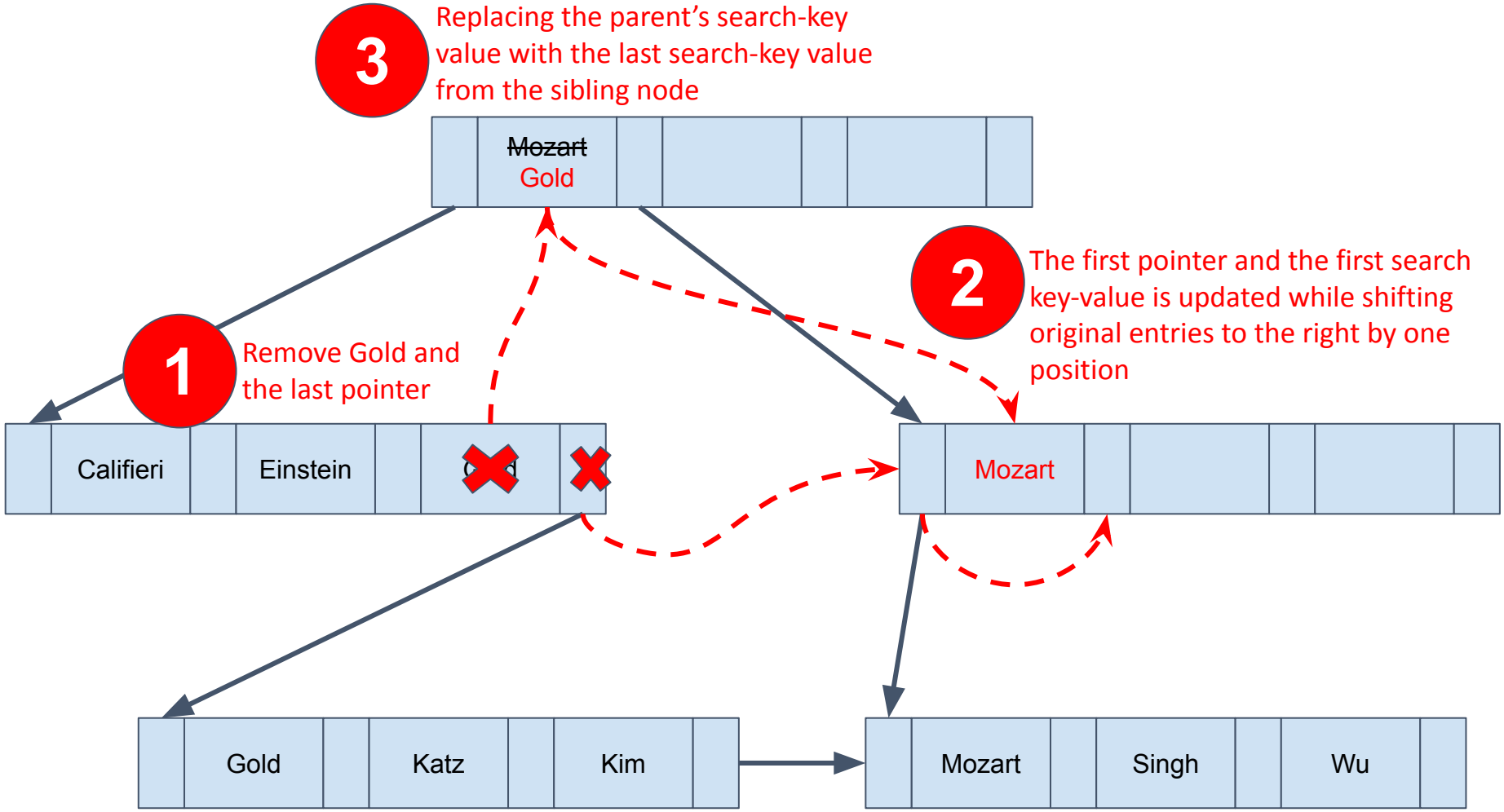
Examples of B+-Tree Deletion (Cont.)

Redistribute entries (non-leaf node case)



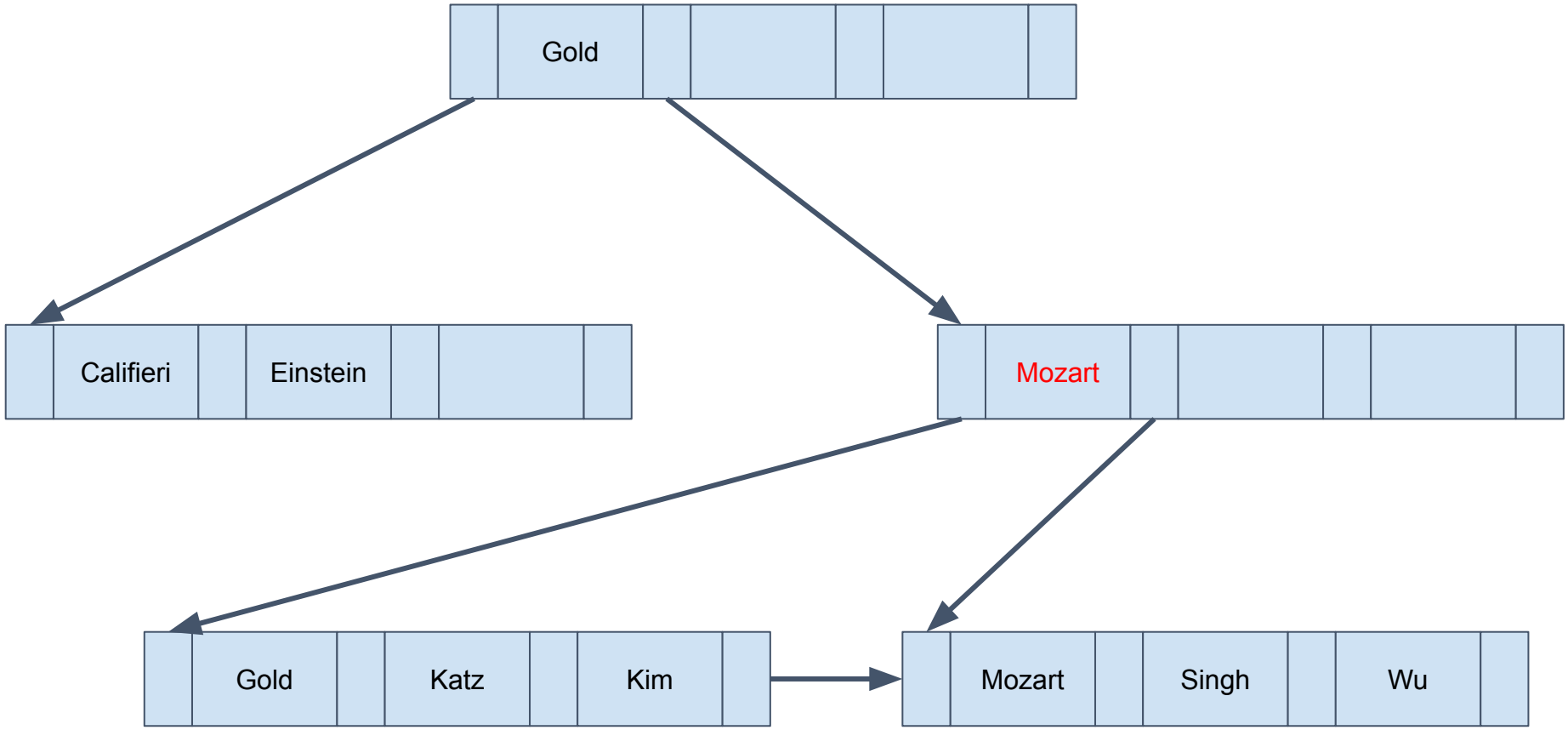
- **Deleting “Srinivasan” causes merging of underfull leaf nodes and redistribution of non-leaf nodes**

Examples of B+-Tree Deletion (Cont.)

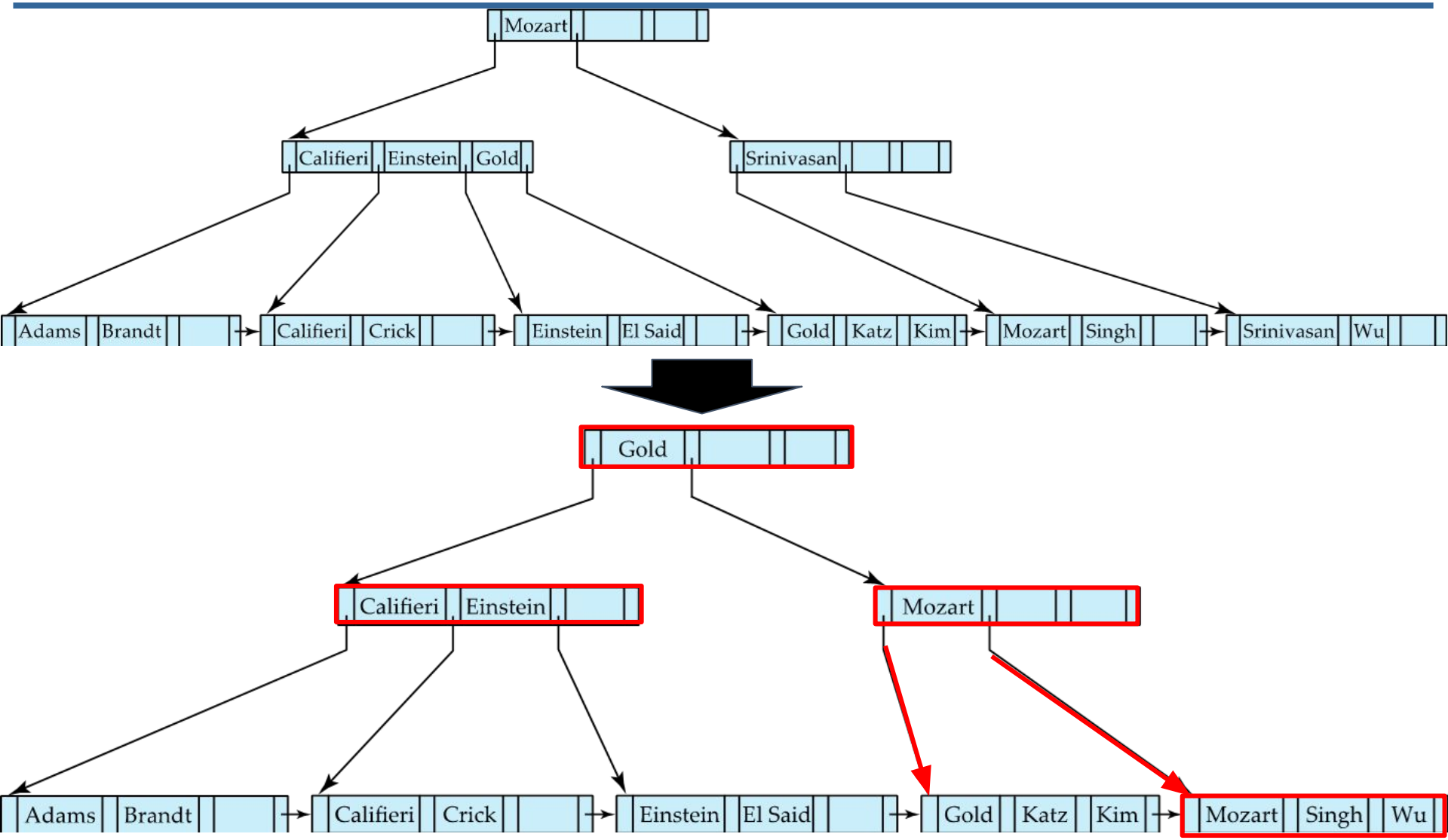


Examples of B+-Tree Deletion (Cont.)

Redistribution result after deleting “Srinivasa”

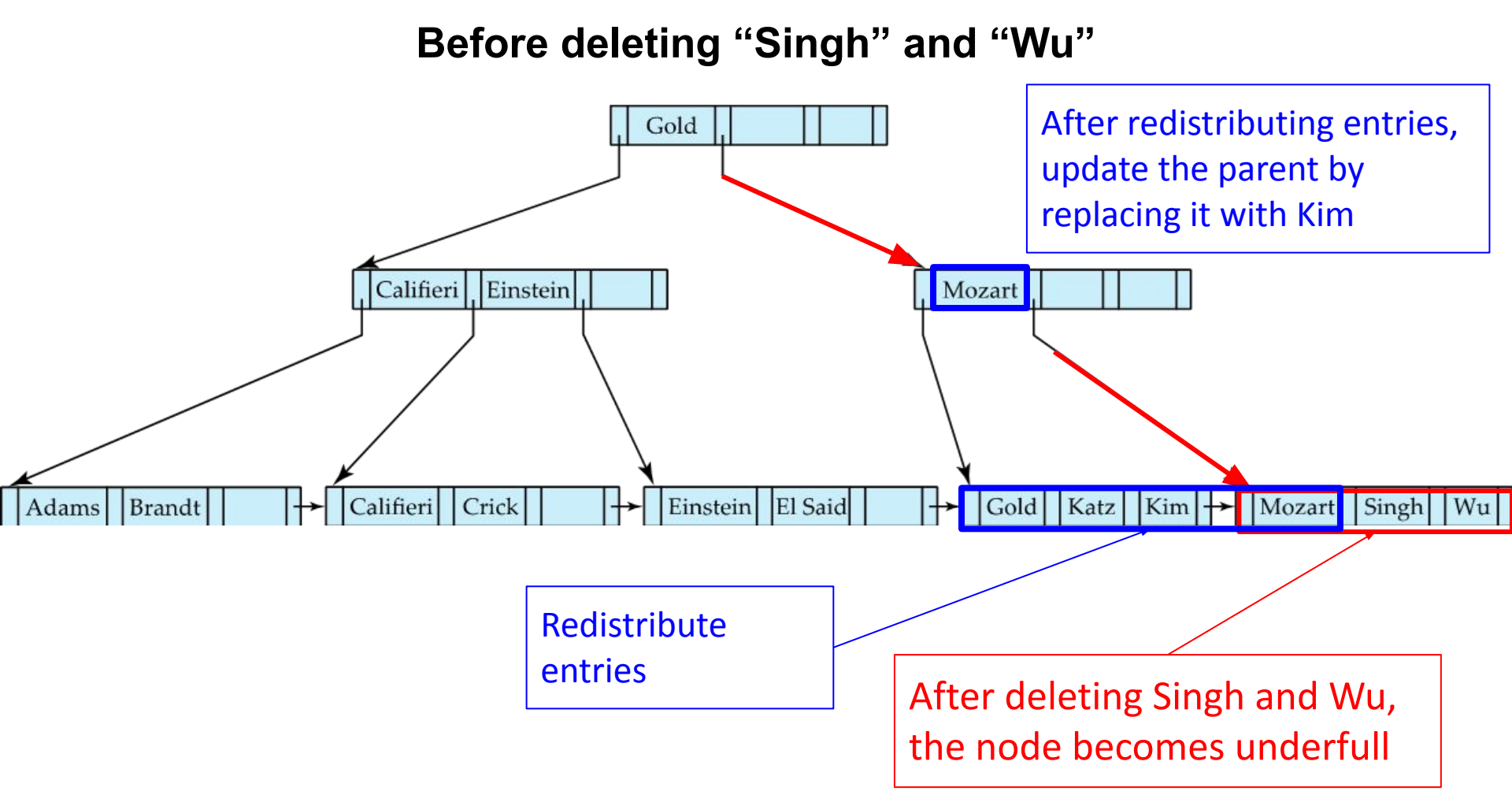


Examples of B+-Tree Deletion (Cont.)

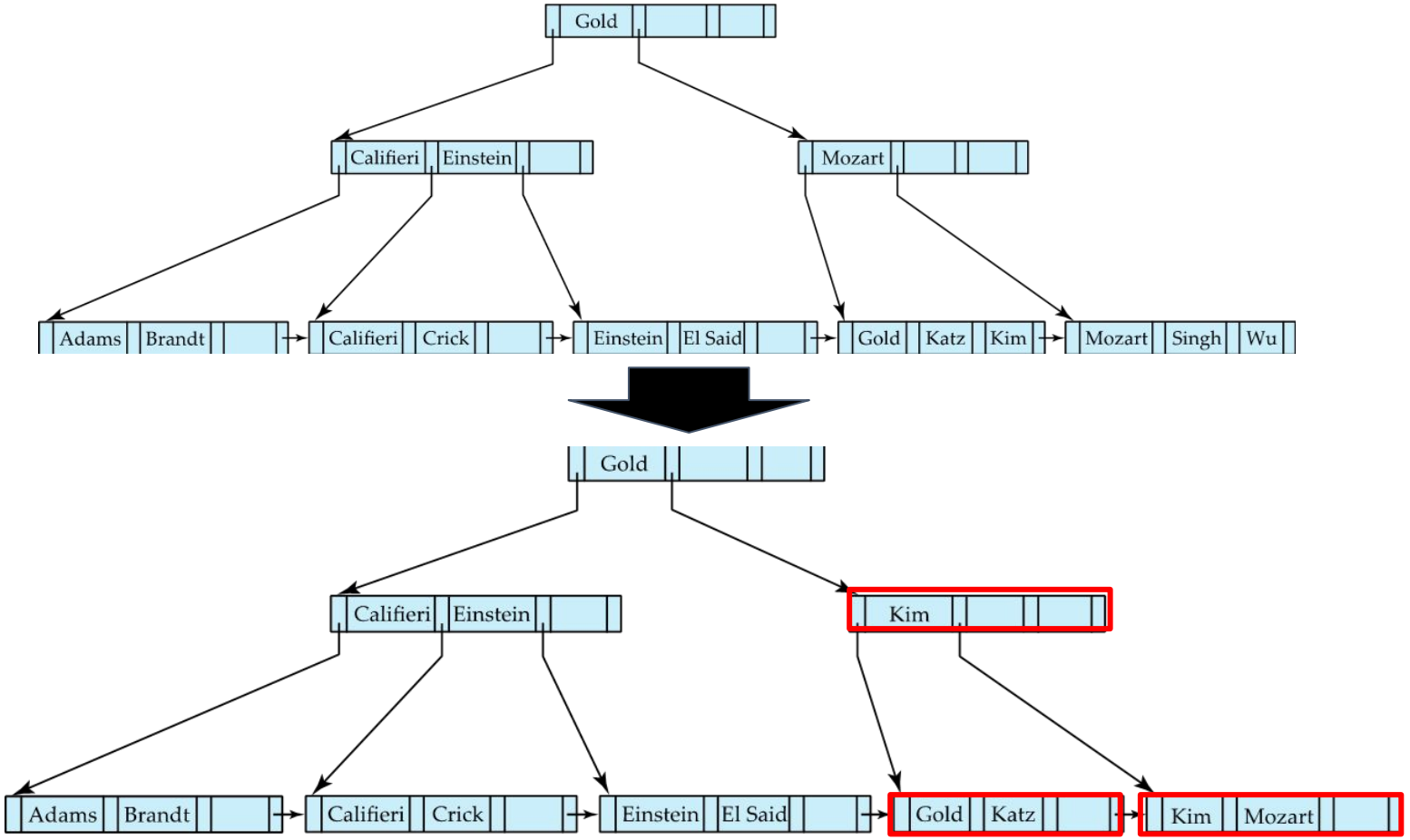


Before and after deleting "Srinivasan"

Examples of B+-Tree Deletion (Cont.)



Examples of B+-Tree Deletion (Cont.)

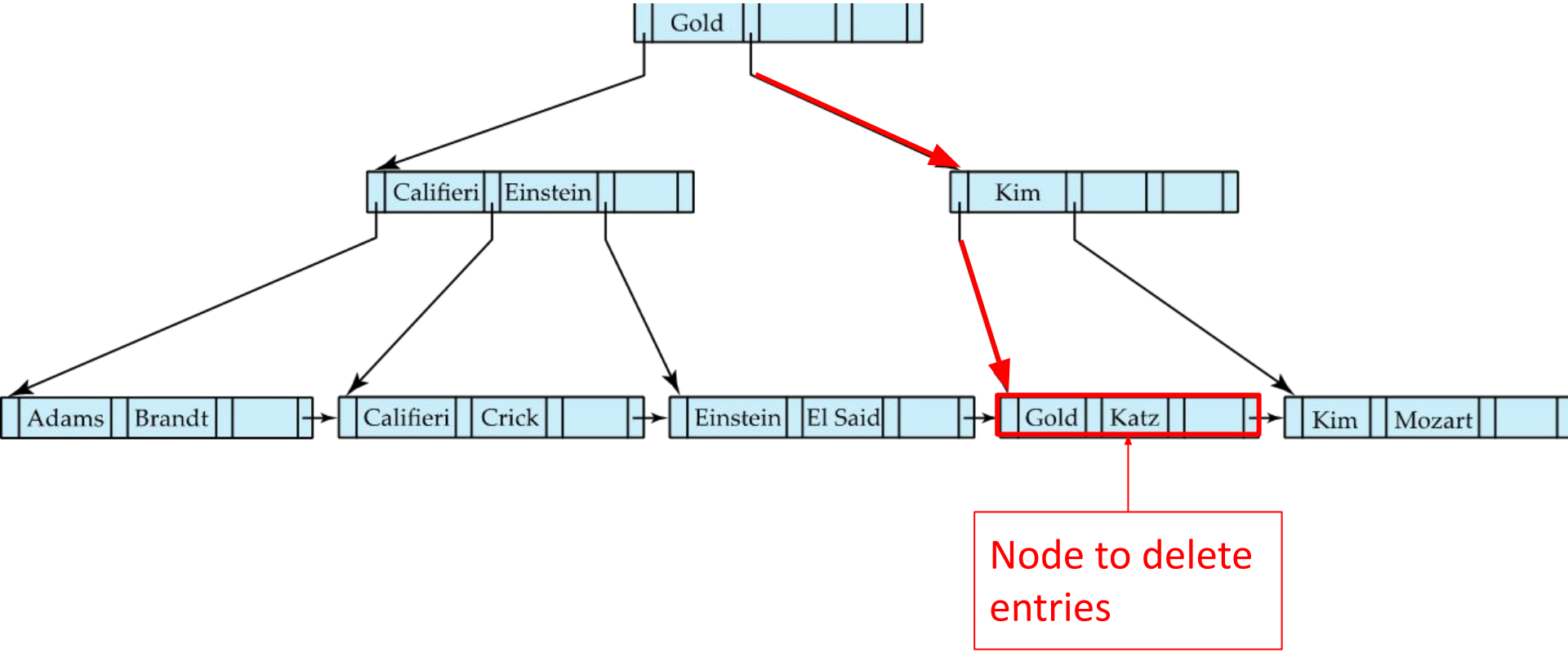


Before and after deleting "Singh" and "Wu"

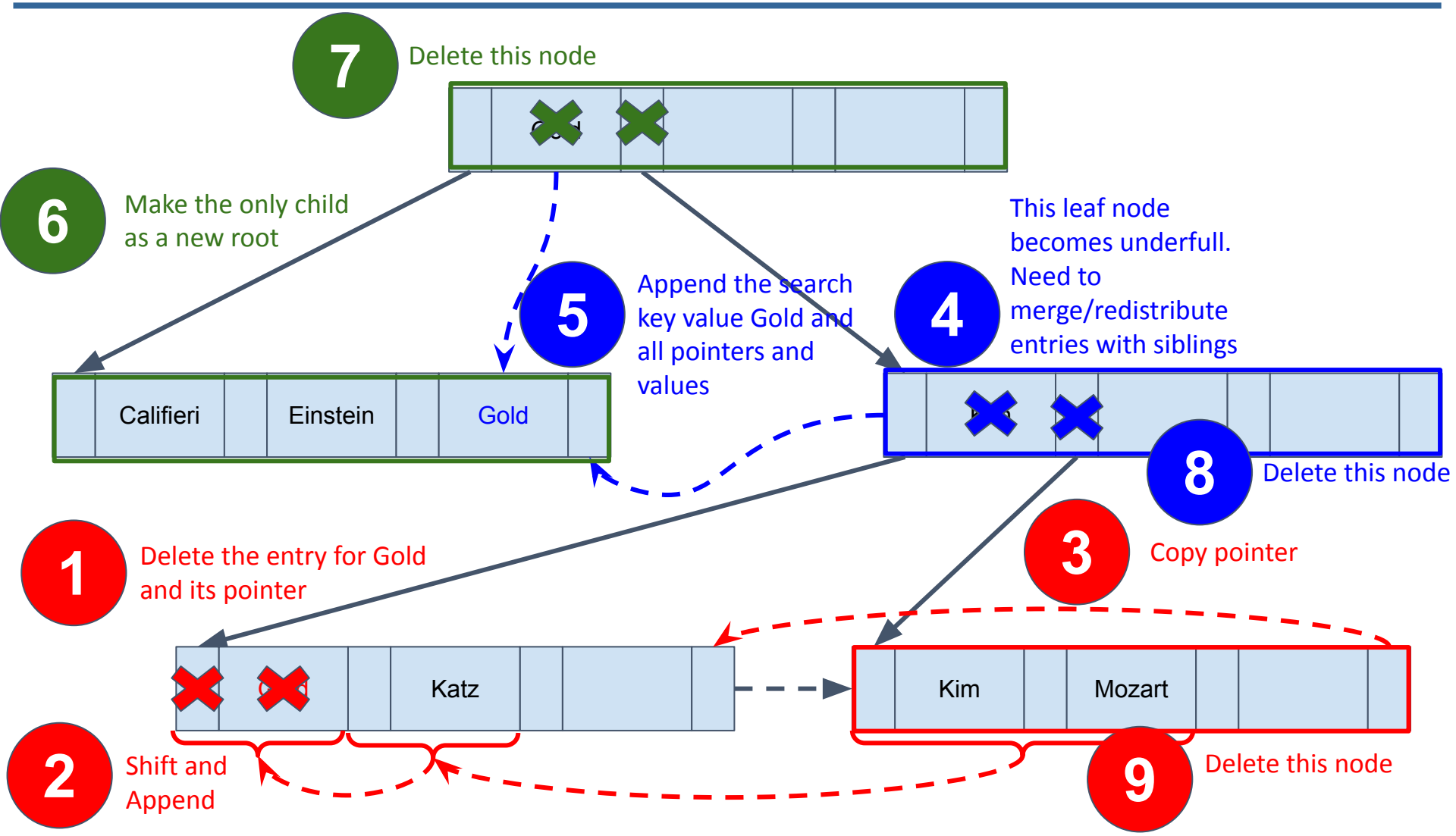
- **Leaf containing Singh and Wu became underfull, and borrowed a value Kim from its left sibling**
- **Search-key value in the parent changes as a result**

Example of B+-tree Deletion (Cont.)

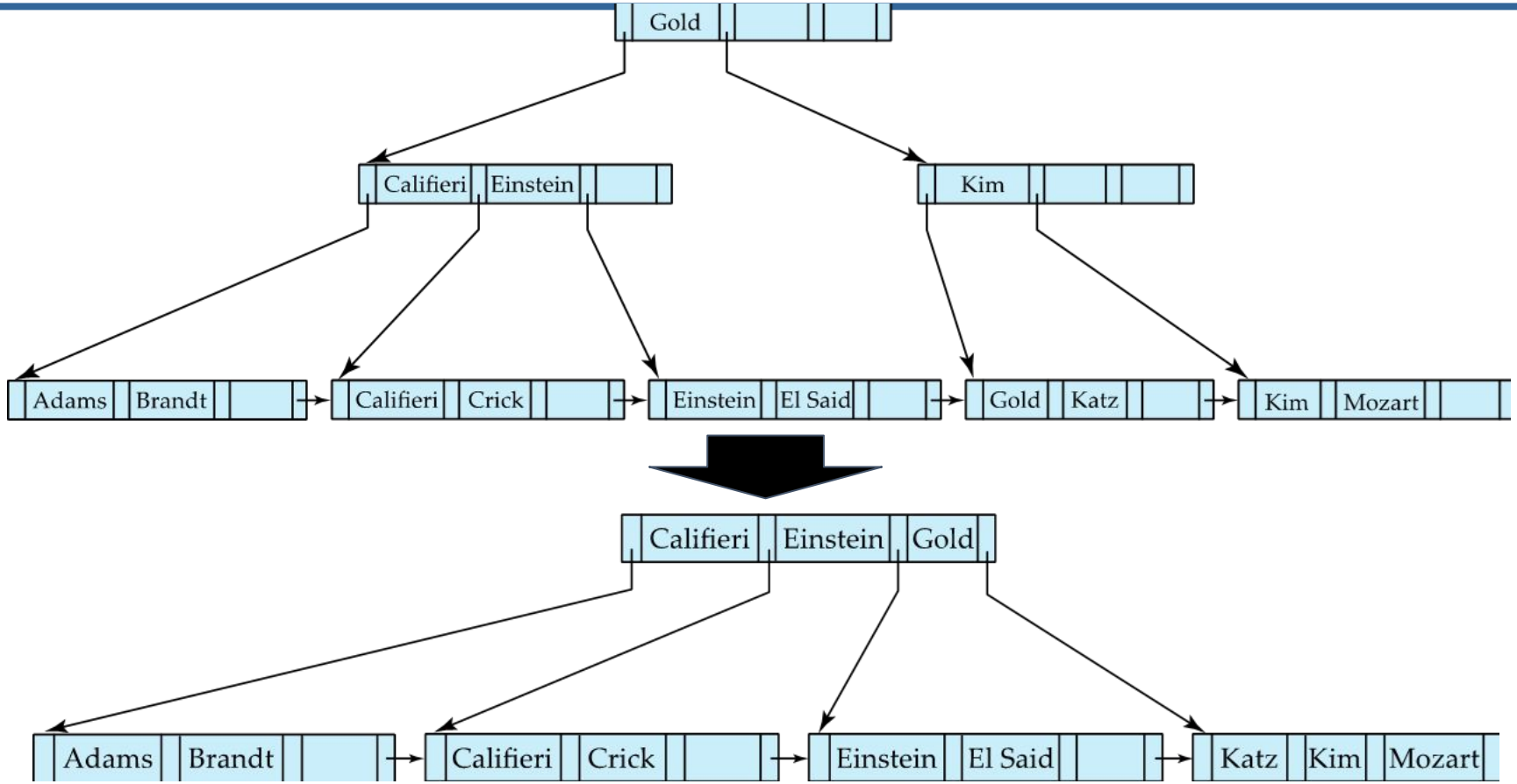
Before deletion of "Gold"



Example of B+-tree Deletion (Cont.)



Example of B+-tree Deletion (Cont.)



Before and after deletion of "Gold"

- **Node with Gold and Katz became underfull, and was merged with its sibling**
- **Parent node becomes underfull, and is merged with its sibling**
  - **Value separating two nodes (at the parent) is pulled down when merging**
- **Root node then has only one child, and is deleted**

Complexity of Updates

- Cost (in terms of number of I/O operations) of insertion and deletion of a single entry proportional to height of the tree
  - With K entries and maximum fanout of n, worst case complexity of insert/delete of an entry is $O(\log_{\lceil n/2 \rceil}(K))$
- In practice, number of I/O operations is less:
  - Internal nodes tend to be in buffer
  - Splits/merges are rare, most insert/delete operations only affect a leaf node
- Average node occupancy depends on insertion order
  - 2/3rds with random, ½ with insertion in sorted order

Non-Unique Search Keys

- Alternatives to scheme described earlier
  - Store a bucket (or a list) of record pointers with a search-key value in the leaf node of the B+-Tree
    - Low space overhead – no need to store search key values multiple times
    - Variable sized bucket can cause difficulties and issues (e.g. may get larger than the node itself)
  - Store search key values once per record
    - Significantly complex to handle split and search on internal nodes – may be more than one leaf node containing the same search key value
    - High space overhead due to the repeated store of the same search-key value
- Duplicate search key introduces significant complication, so most database systems only handle unique search keys
  - Automatically add record-ides or other attributes to make nonunique search keys unique

Overview

- B+-Tree Index Files II
- Assignments

Assignments

- Reading: Ch14.3.2-14.3.5
- Practice Excercises: 14.1, 14.2, 14.3a, 14.4 (for 14.3a), 14.7

Solutions to the Practice Excercises:
https://www.db-book.com/Practice-Exercises/index-solu.html

The End