Indexing: B$^+$-Tree

CS 377: Database Systems

Recap: Indexes

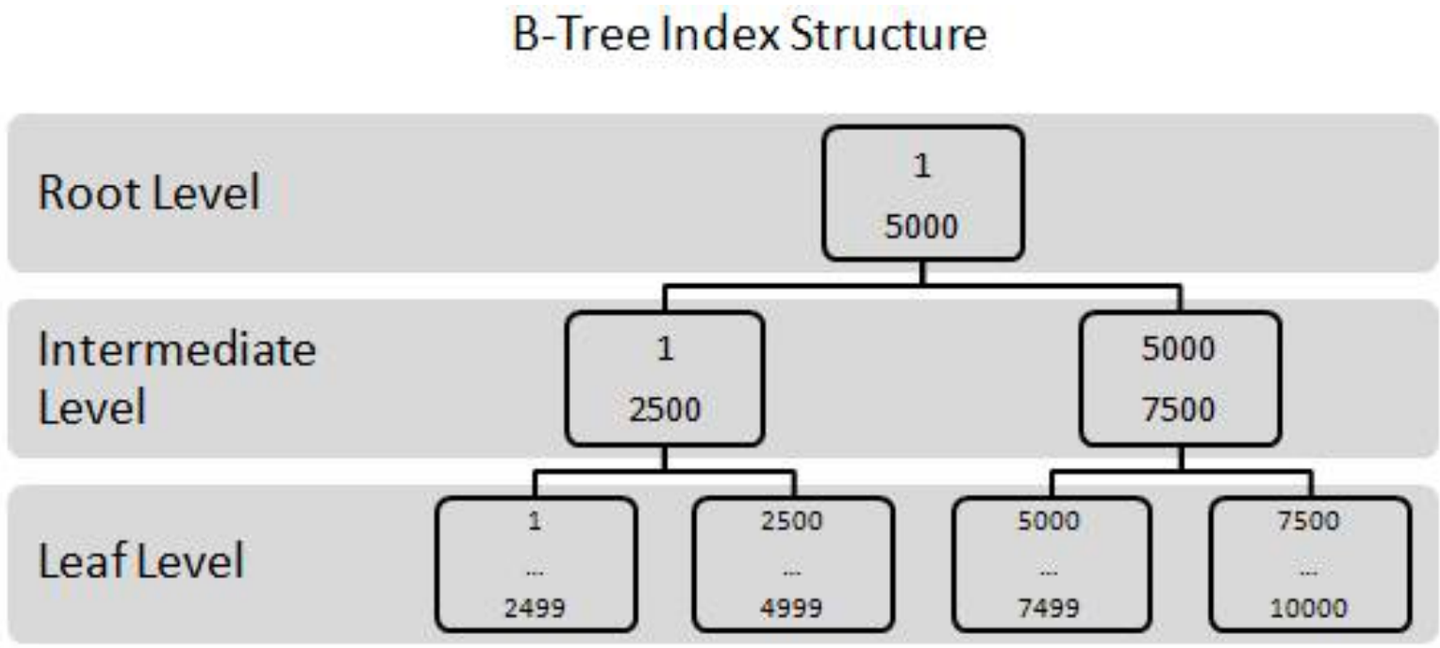
- Data structures that organize records via trees or hashing
  - Speed up search for a subset of records based on values in a certain field (search key)
    - Search key need not be the same as the key!
- Hash index
  - Good for equality search
  - In expectation: O(1) I/Os and CPU performance for search and insert

B^+-Tree

- Dynamic, multi-level tree data structure
  - Adjusted to be height-balanced (all leaf nodes are at same depth)
  - Good performance guarantee — supports efficient equality and range search
- Widely used in DBMS

B$^+$-Tree Basics

- Order p: maximum number of children at each node
- Every node contains m entries, with
- Minimum 50% occupancy
- Only exception is root node



https://www.simple-talk.com/iwritefor/articlefiles/735-image012.jpg

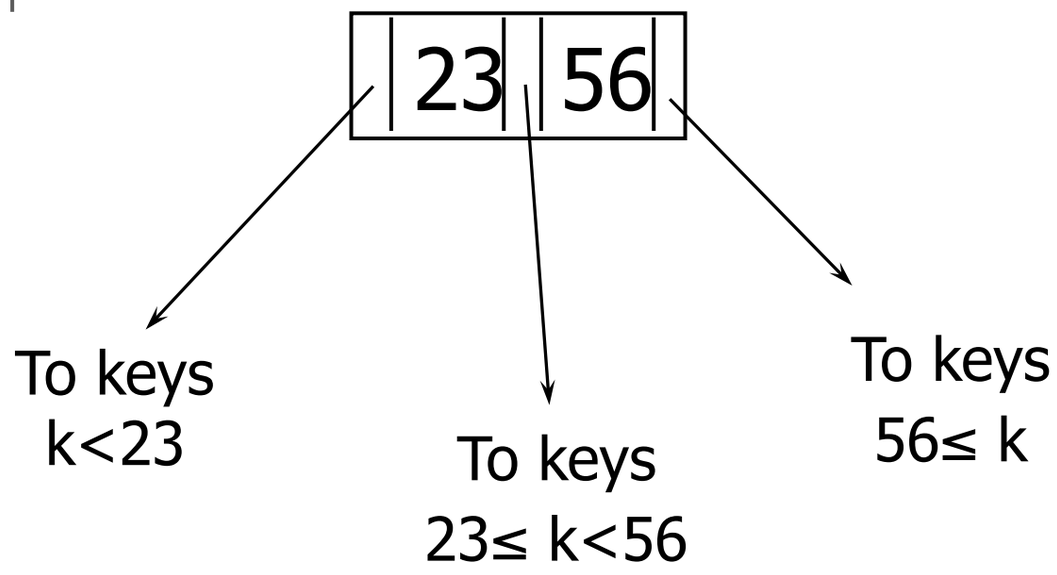
B^+-Tree Node Structure

- Typical node

  - K_i are the search-key values
    - P_i are the points to the children (for non-leaf nodes) or pointers to records (for leaf nodes)
- Search keys in nodes are ordered
  - $K_1 < K_2 < \ldots < K_{n-1}$

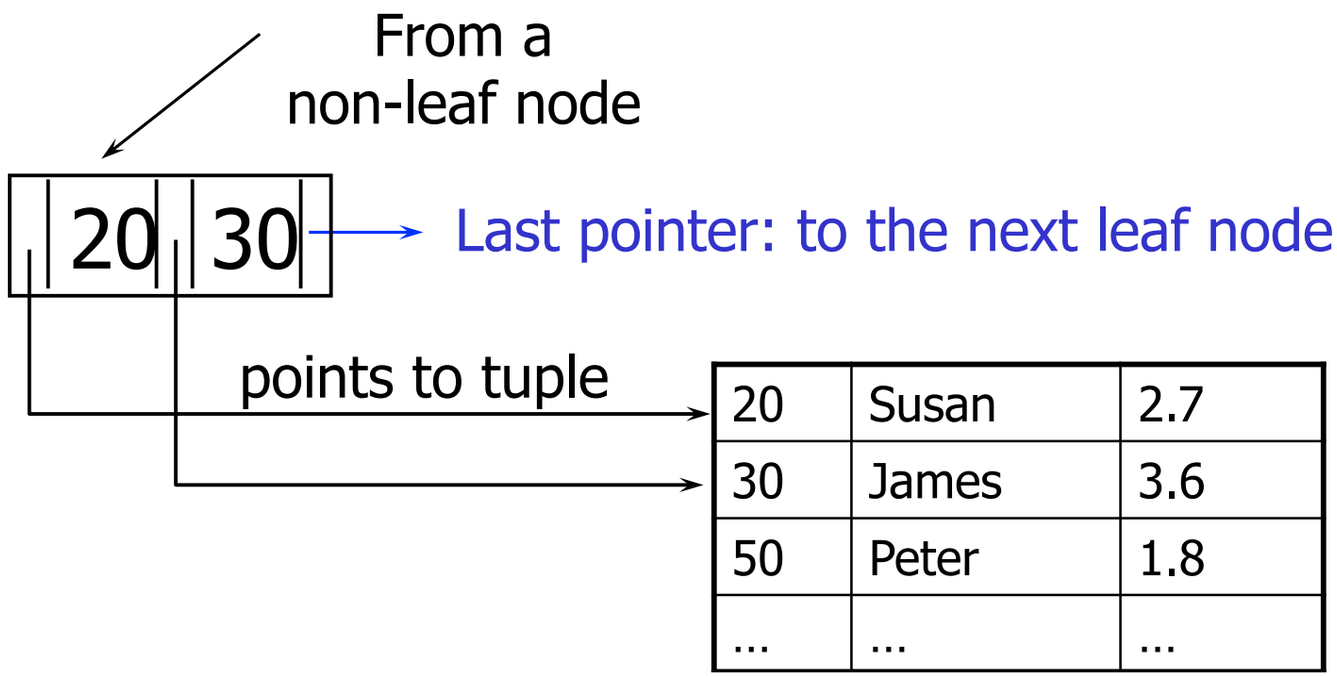
B^+-Tree: Non-Leaf Node

- Multi-level sparse index on the leaf nodes
- All search-keys in the subtree to which P_1 points to are less than K_1
- All search-keys in the subtree to which P_n points to have values greater than K_{n-1}



B^+-Tree: Leaf Node

- All pointers (except the last one) point to tuples or records
  - Search key values are sorted in order
- Last pointer (P_n) points to next leaf node in search-key order



Example: B+-Tree

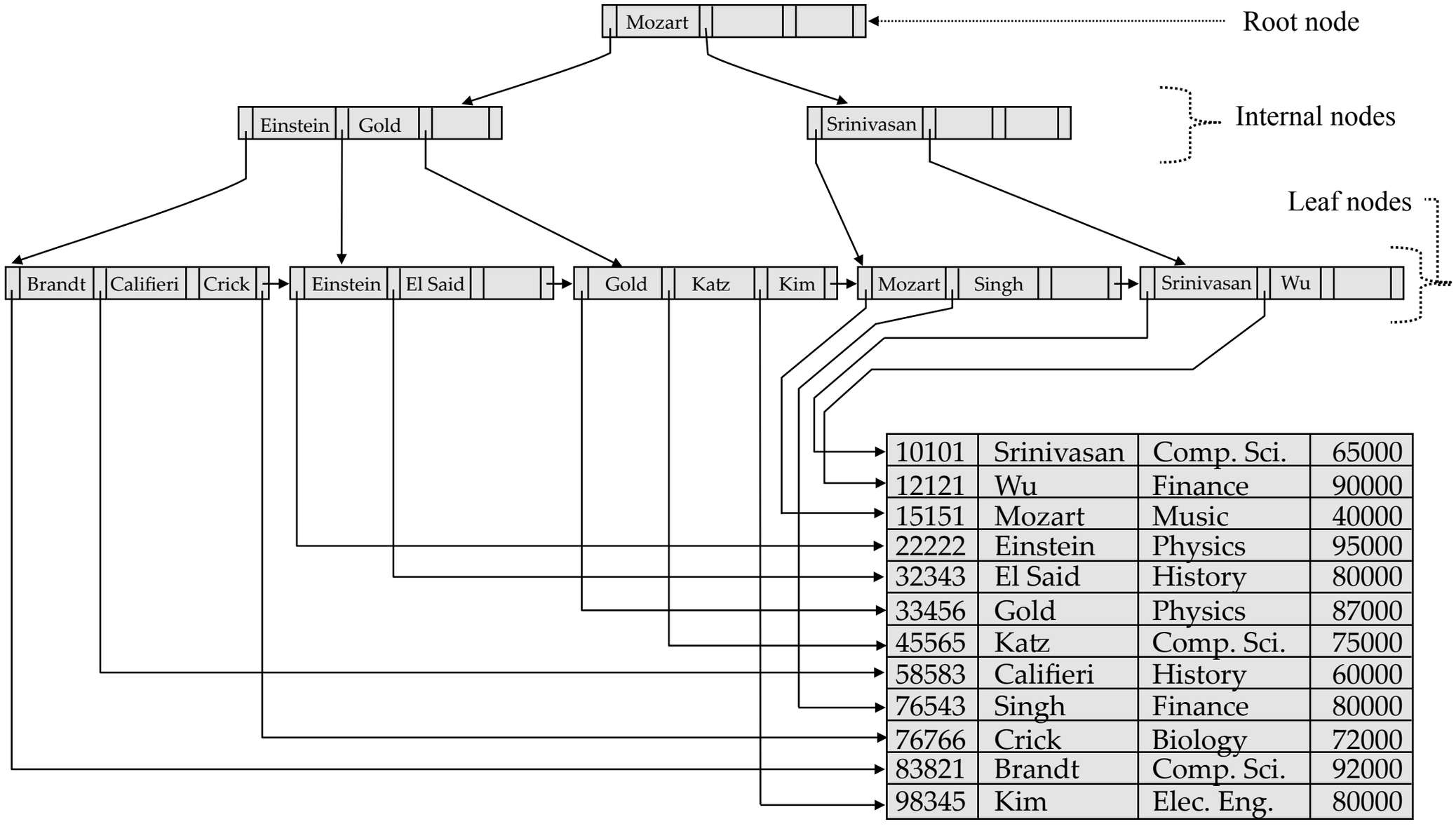


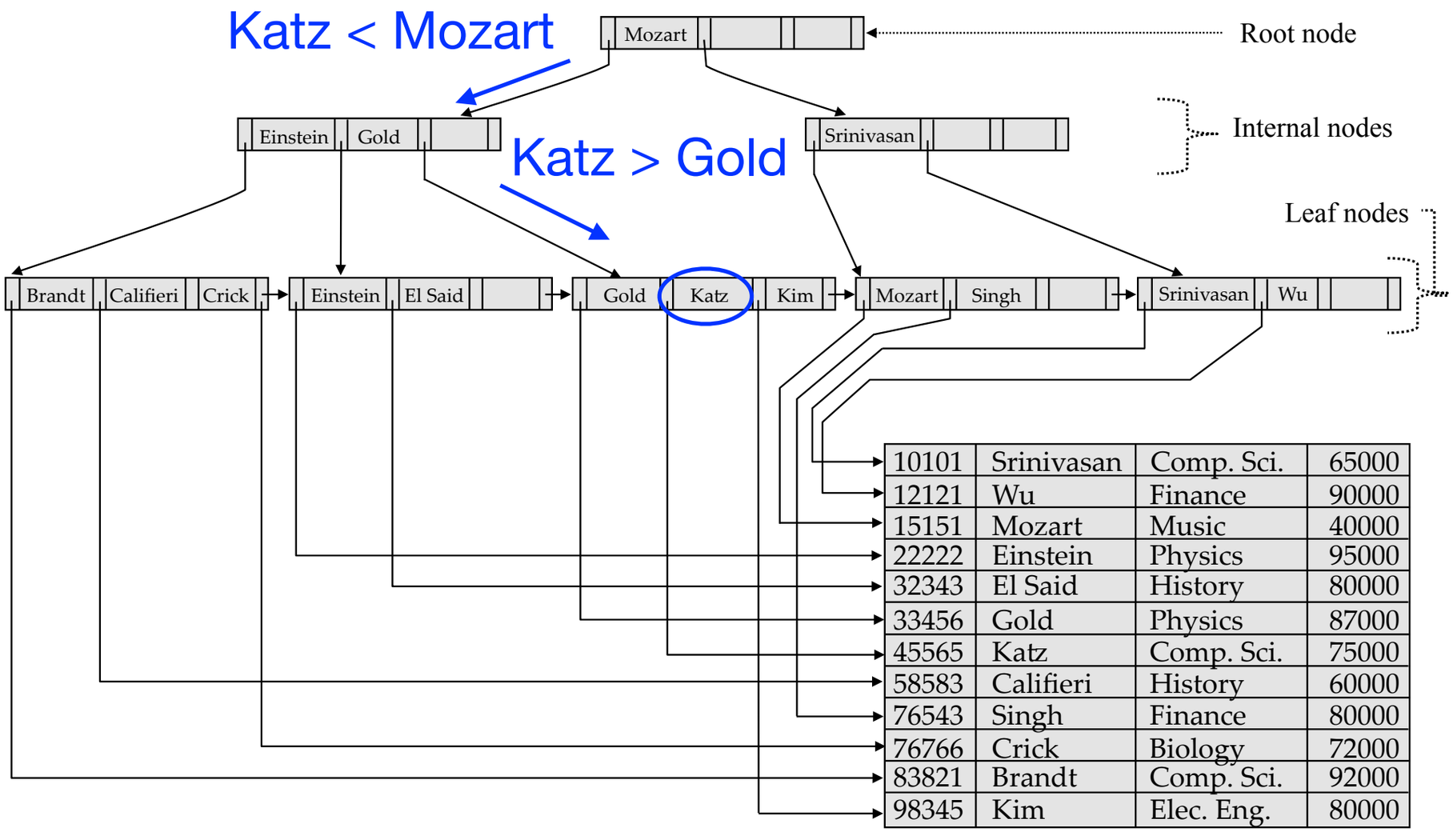
Figure from Database System Concepts book

B$^+$-Tree Search

- Start from root
- Examine index entries in non-leaf nodes to find the correct children
    - Searched using a binary or linear search
- Traverse down the tree until a leaf node is reached

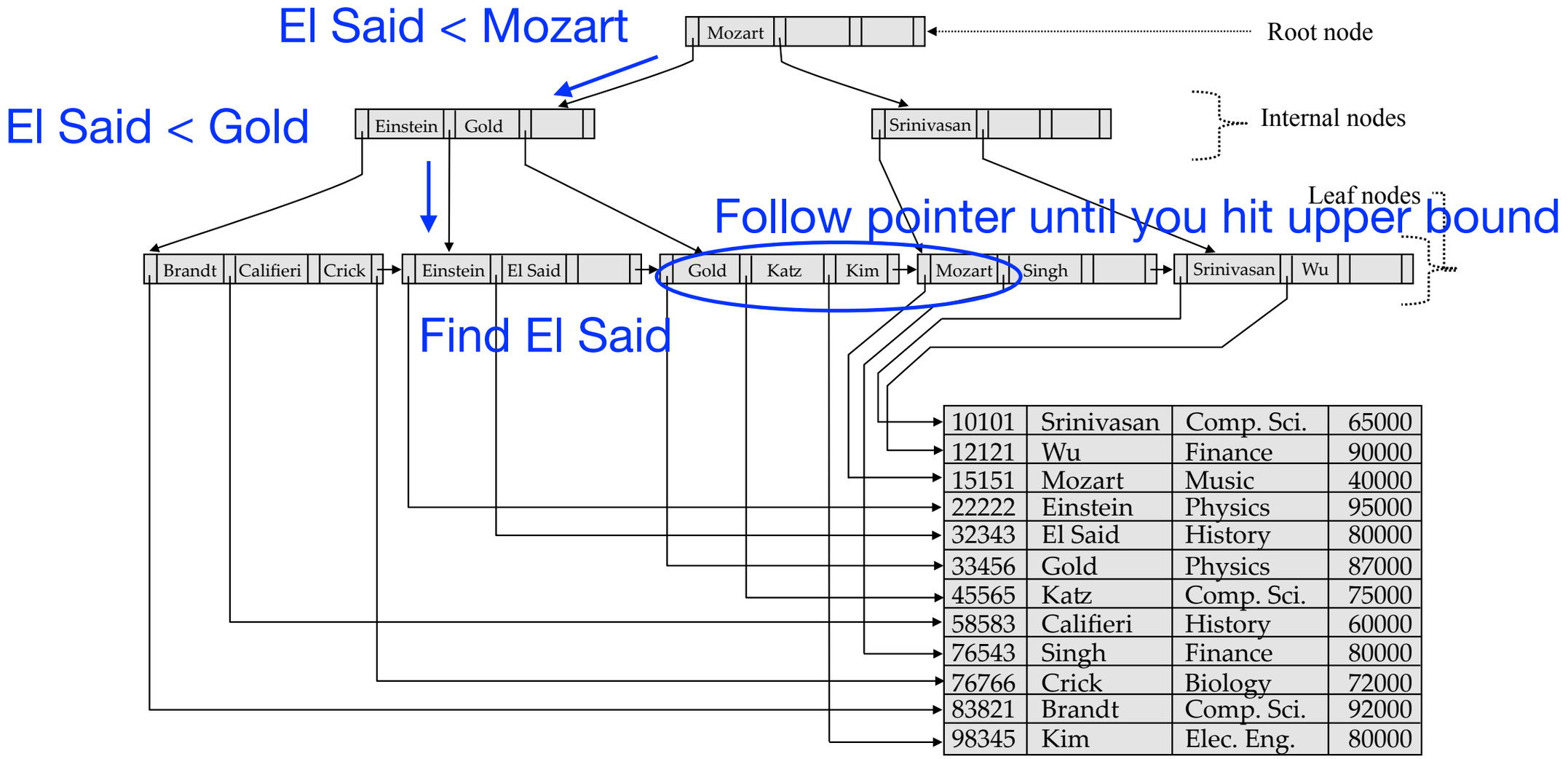
Example: B$^+$-Tree Exact Query

SELECT * FROM instructor WHERE name = ‘Katz’;



Example: B$^+$-Tree Range Query

SELECT * FROM instructor WHERE name > “El Said”
AND name < “Singh”;

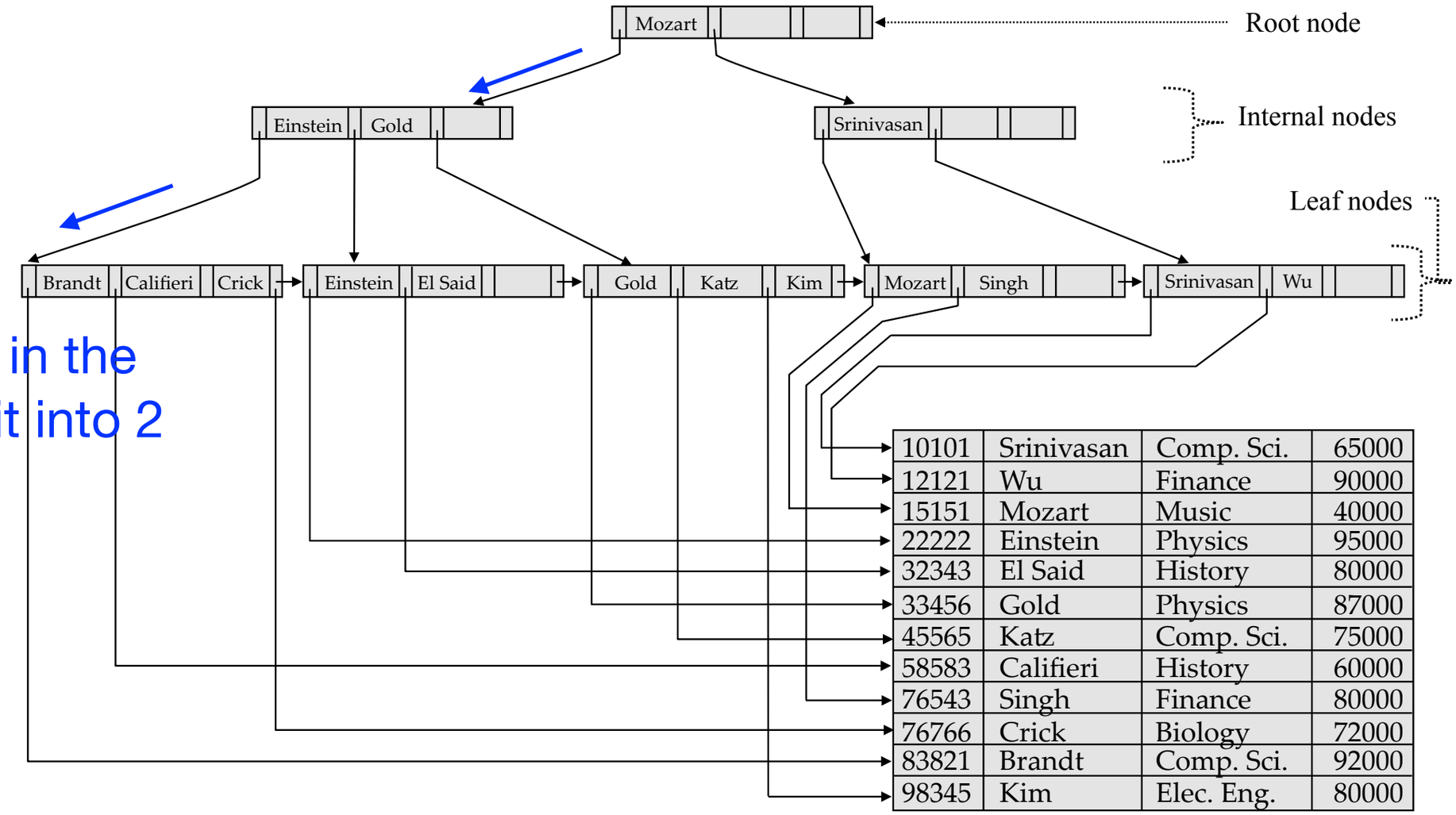


B^+-Tree Insert

- Find the leaf node in which the search-key value would appear
- If the search-key value is already present in the leaf node
  - Add the record to the file
  - Add pointer to the bucket (if necessary)
- If search-key value is not present
  - Add record to main file
  - If room in the leaf node, insert (key, pointer) pair in leaf node
  - Otherwise split the node along with the new (key, pointer) pair

Example: B$^+$-Tree Insert

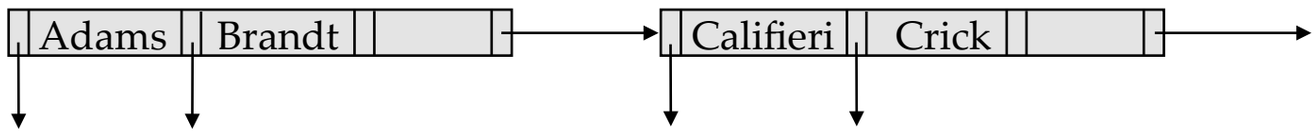
INSERT INTO instructor(name) VALUES('Adams');



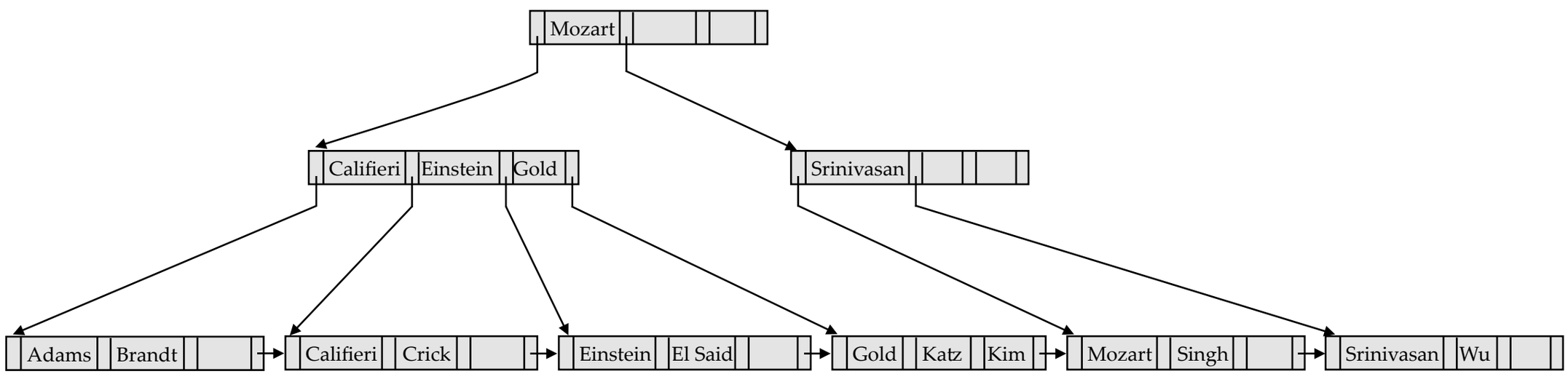
No room in the leaf - split into 2

Example: B$^+$-Tree Insert (2)

Split so that Adams and Brandt on one side, Califieri and Crick on the other

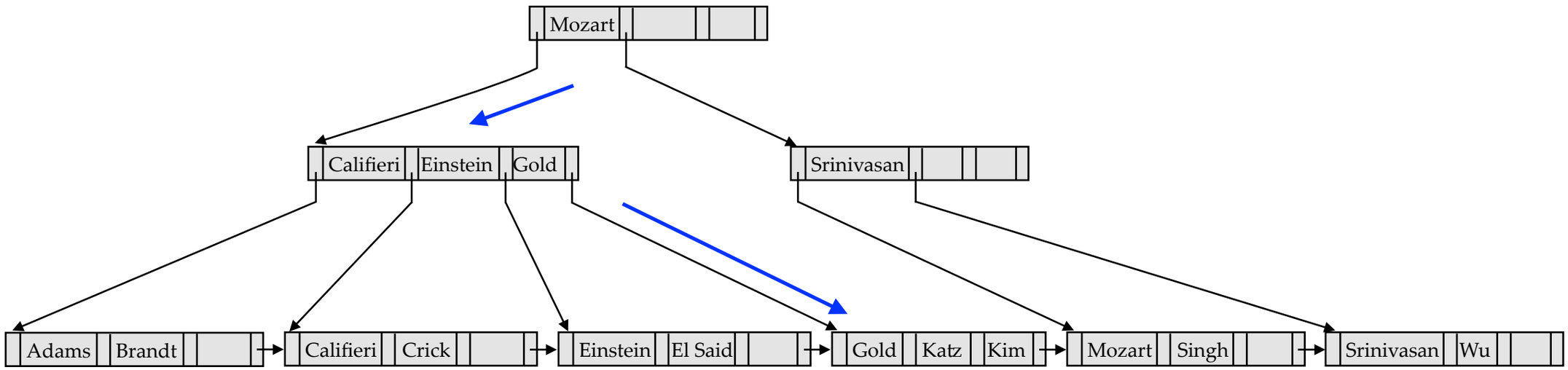


Since we are introducing new leaf node, we need to update the parent leaf…



Example: B$^+$-Tree Insert (3)

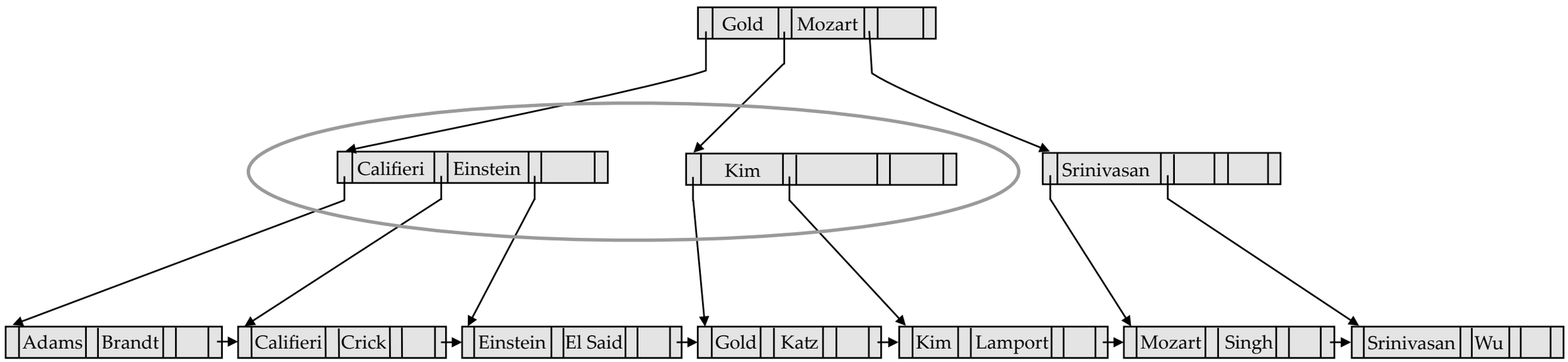
INSERT INTO instructor(name) VALUES('Lamport');



No room in the leaf - split into 2

Example: B$^+$-Tree Insert (4)

But after split, there is no room in the parent node either, so parent needs to be split which affects the root

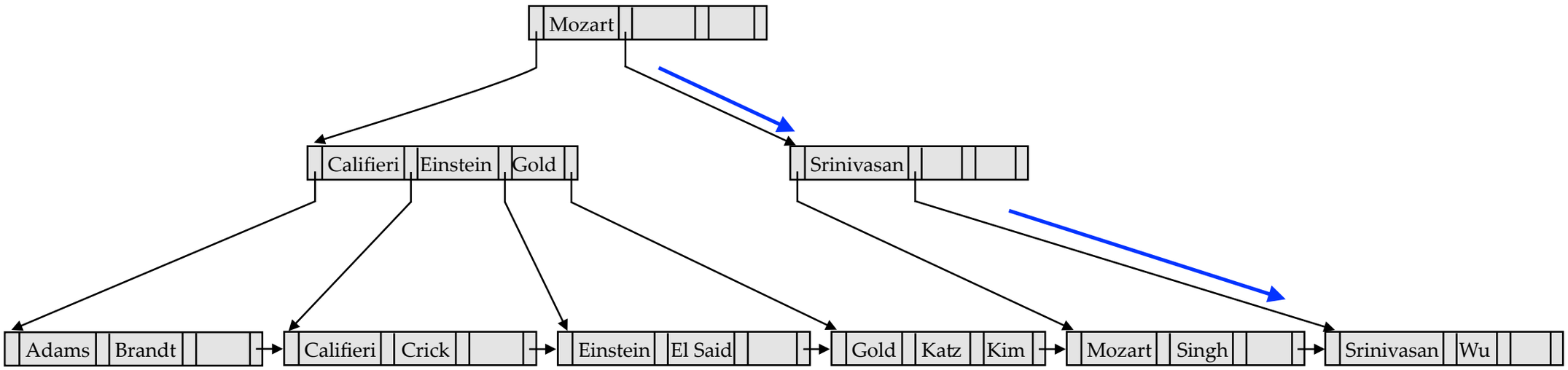


B^+-Tree Deletion

- Find the leaf node in which the search-key value appears and delete it from the main file and bucket
- Delete the (key, pointer) pair from the leaf node
- If underflow occurs (leaf node is under minimum size)
  - Merge with sibling (reduce tree pointers from parent nodes)
  - Redistribute entries from left or right sibling if merge not possible

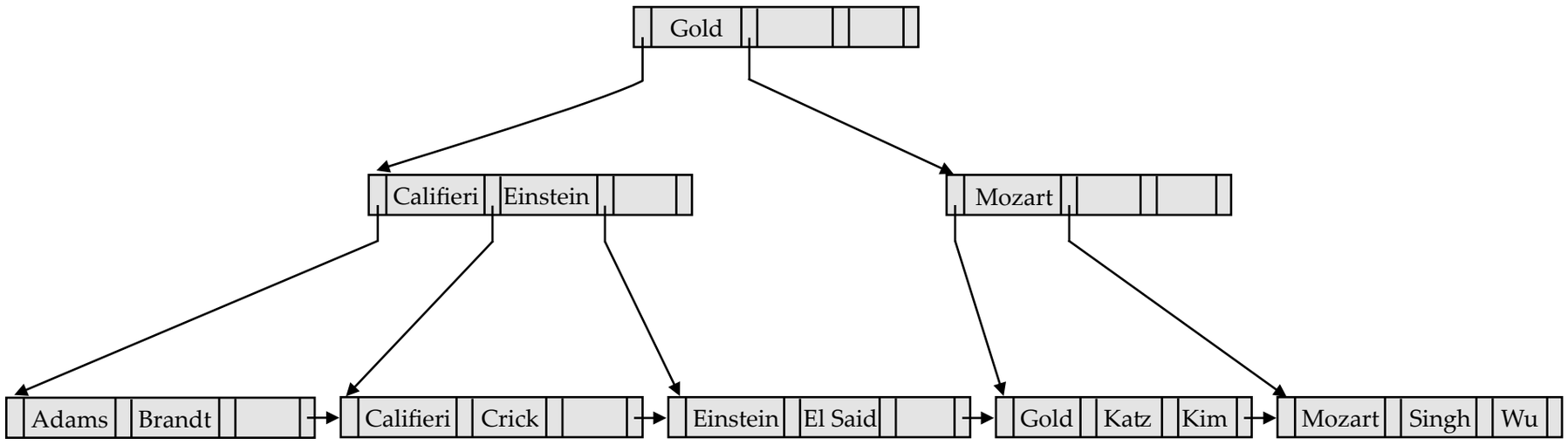
Example: B$^+$-Tree Delete

DELETE FROM instructor where name = ‘Srinivasan’;



After the deletion, only Wu is in the leaf node, and that is too empty since it needs to be at least 50% occupied => must merge with a sibling node or redistribute entries between the nodes

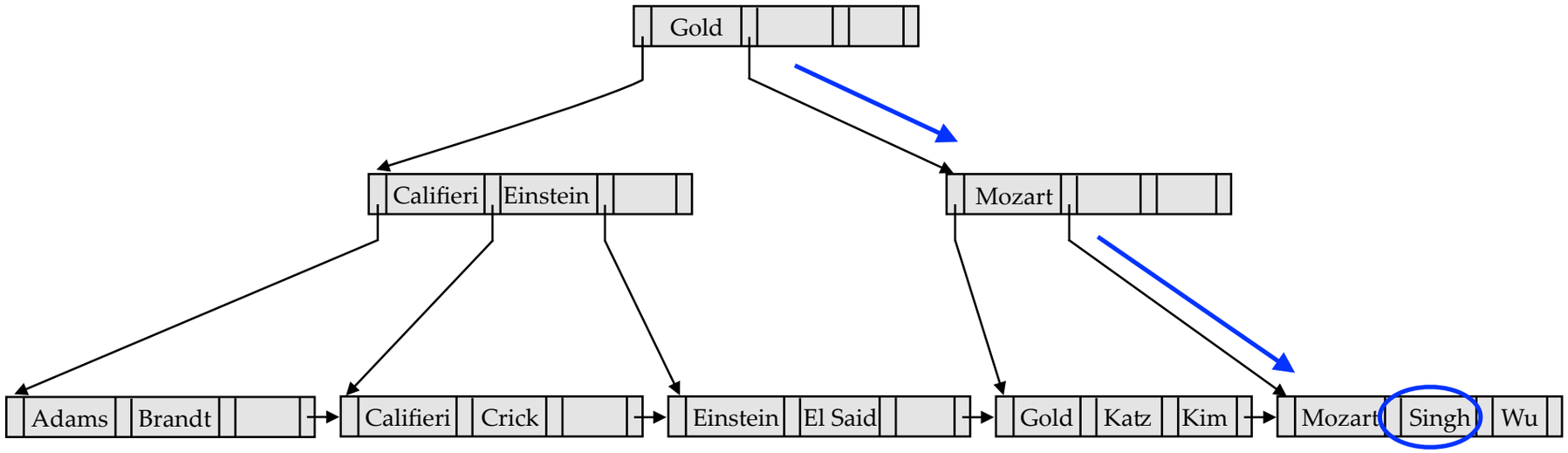
Example: B$^+$-Tree Delete (2)



Merged with the previous sibling and delete from parent, but parent also only has one pointer so we must either merge or redistribute… since merging is not possible, must redistribute

Example: B$^+$-Tree Delete (3)

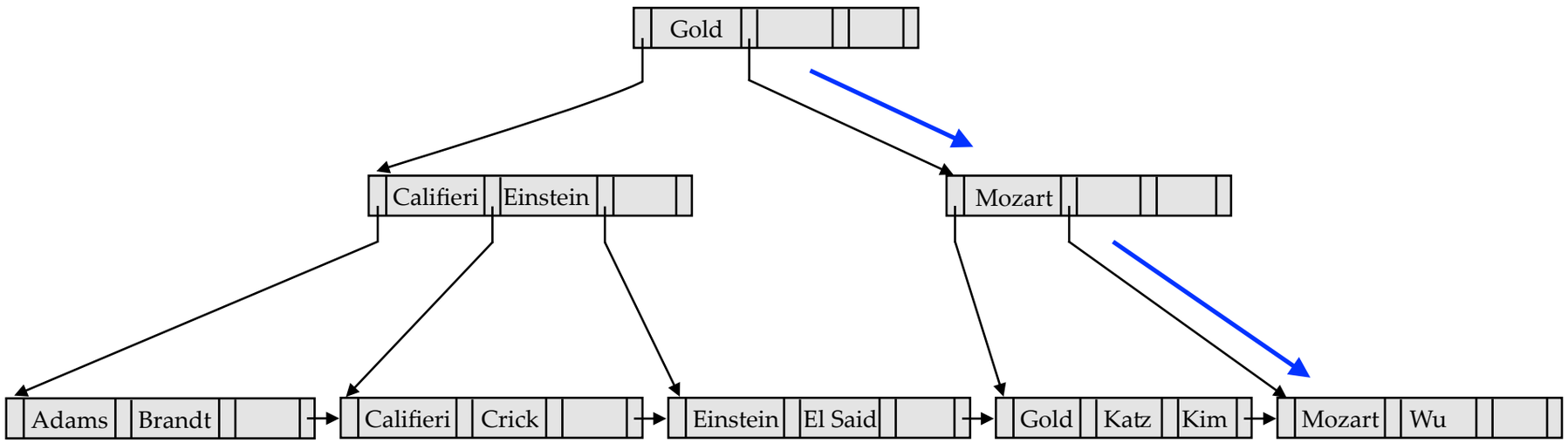
DELETE FROM instructor where name = ‘Singh’;



Easy case - just delete!

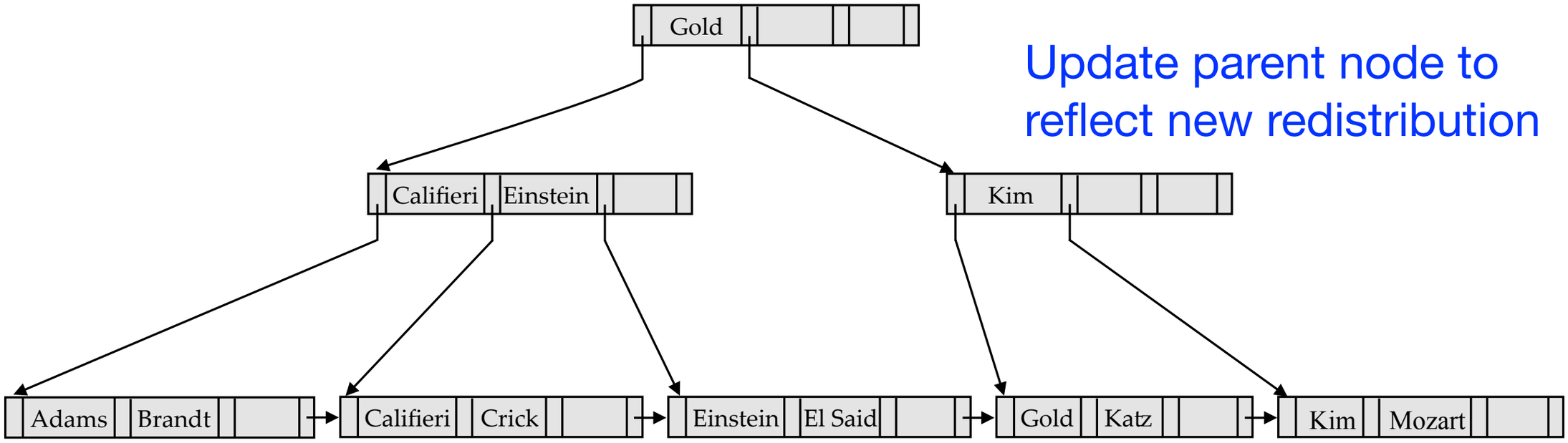
Example: B$^+$-Tree Delete (4)

DELETE FROM instructor where name = ‘Wu’;



Deleting Wu makes the leaf undefiled and not possible to merge with sibling - so redistribute

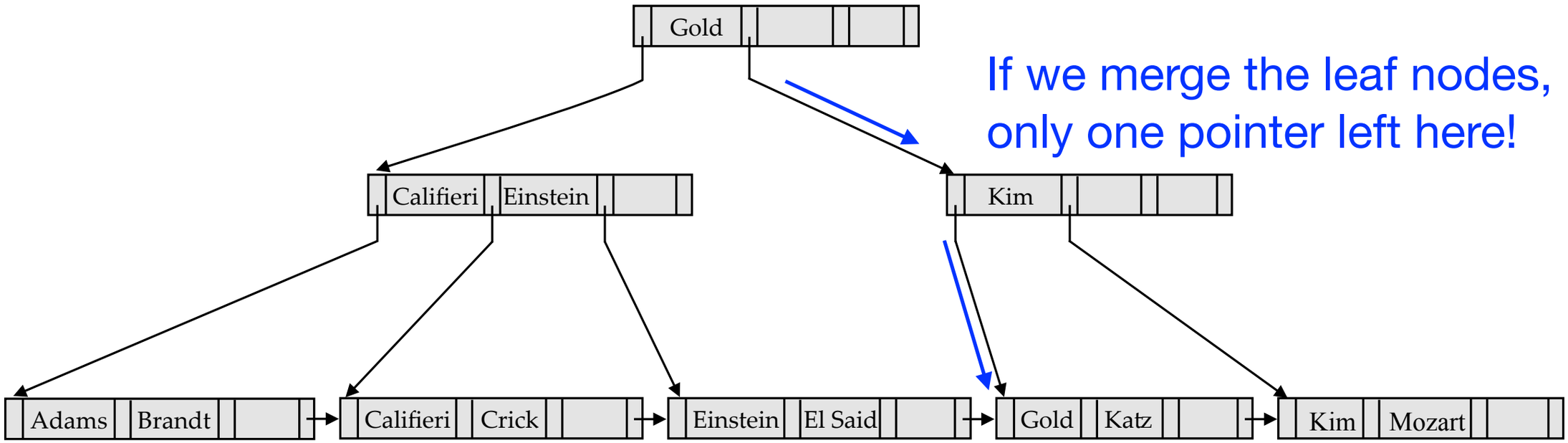
Example: B$^+$-Tree Delete (5)



Update parent node to reflect new redistribution

Example: B$^+$-Tree Delete (6)

DELETE FROM instructor where name = ‘Gold’;

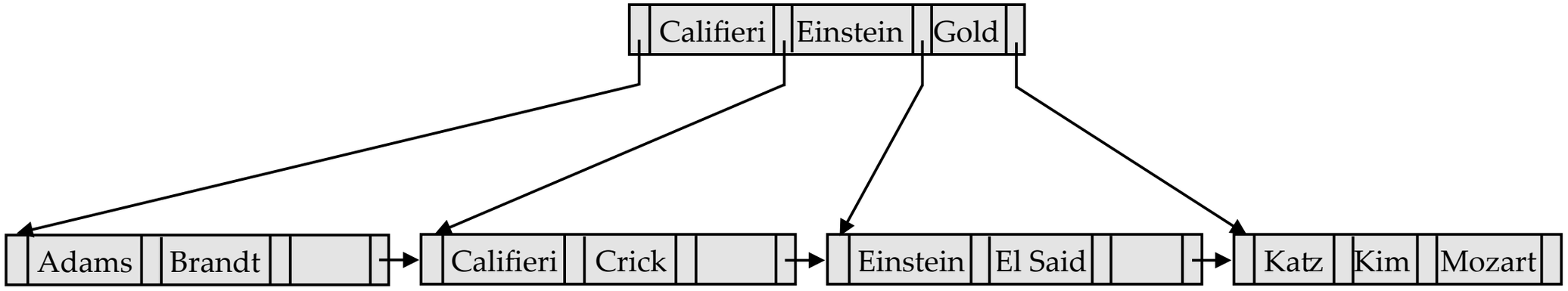


If we merge the leaf nodes, only one pointer left here!

Merge Katz with the sibling on the right

Example: B$^+$-Tree Delete (7)

Delete original root node to avoid condition where root has only one child



Depth of tree has now decreased by 1!

B+-Tree: Dealing with Duplicates

- Can have many data entries with the same key value (e.g., Year of a movie)
- Solution 1:
  - All entries with a given key value reside on a single page
  - Use overflow pages
- Solution 2:
  - Allow duplicate key values in data entries
  - Modify search to deal with duplicates

B^+-Tree Performance

- How many I/O's are required for each operation?
  - Worst case cost of insertion / deletion are proportional to the height of the tree (more or less)
    - Height is roughly the $\log_{n/2}$ (number of records)
    - Fanout can be typically large (in the hundreds) - many keys and pointers can fit into one block
- A 4-level tree is enough for "typical" tables

B^+-Tree Index Files

- Alternative to indexed-sequential files
- (Pro) Automatically reorganizes itself with small, local changes when dealing with insertions/deletions
- (Pro) Reorganization of entire file is not required to maintain performance
- (Con) Extra insertion and deletion overhead
- (Con) Additional space required
- Cool visualization - https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

Multiple-Key Access

- What if we want to access more than one attribute such as a combination of attributes?

- Example:
  SELECT * from EMPLOYEE WHERE dno = 4 AND age = 59;

- Assume that we may have created index on dno and/or age

Multiple-Key Access Strategies

- Dno has index, age does not: access tuples with dno=4 using index then do linear search to satisfy age = 59

- Age has index, dno does not: access tuples with age = 59 using the index and then do linear search to satisfy dno = 4

- Dno and age have indices: get the set of pointers and find the intersection of these records or pointers

But what if the number of records that meet each condition is individually large, but the intersection is small? Are the methods efficient?

Composite Search Keys

- Search on a combination of attributes (e.g., age and dno)
- Lexicographic ordering $(a_1, a_2) < (b_1, b_2)$ if either
  - $a_1 < b_1$ or
  - $a_1 = b_1$ and $a_2 < b_2$
- Suppose we have an index on (dno, age)
  - Can efficiently handle queries with dno = 4 and age = 59
  - Can also efficiently handle cases with dno = 4 and age < 59

Beyond B$^+$-Tree and Hashing

- Tree-based indexes: R-trees and variants, GIST, etc.
- Text indexes: inverted-list index, suffice arrays, etc.
- Other tricks: bitmap index, bit-sliced index, etc.

Choosing Indexes

- What indexes should we create?
  - Which relations should have indexes?
  - What field(s) should be in the search key?
  - Should we build several indexes?
- For each index, what kind should it be?
  - Clustered?
  - Hash/tree?

Choosing Indexes (2)

- Consider best plan using current index and see if a better plan is possible with an additional index — if so, create
    - Must understand how DBMS evaluates queries and creates query evaluation plans
- Consider tradeoffs: faster queries but slower updates and more storage

Choosing Indexes (3)

- Attributes in WHERE clause are candidates for index keys
  - Exact match condition suggests hash index
  - Indexes also speed up joins
  - Range query suggests tree index
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions
  - Order of attributes is important for range queries

Index Demo on IMDB (imdb-index-example.sql)

Tuning Indexes

- Initial choice of indexes may have to be revised
    - Certain queries may take too long to run without an index
    - Certain indexes may not get used at all
    - Certain indexes undergo too much updating because the attribute undergoes frequent changes

Indexing: Recap

- B+-Tree
- Composite search keys
- Choosing indexes