Indexing

Outline

- Basic Concepts
- Ordered Indices
- B^+-Tree
- Hashing

Why indexing?

Index

Why indexing?

- Many queries reference only a small no. records

  E.g., “Find the total number of credits earned by the student with ID 22201”

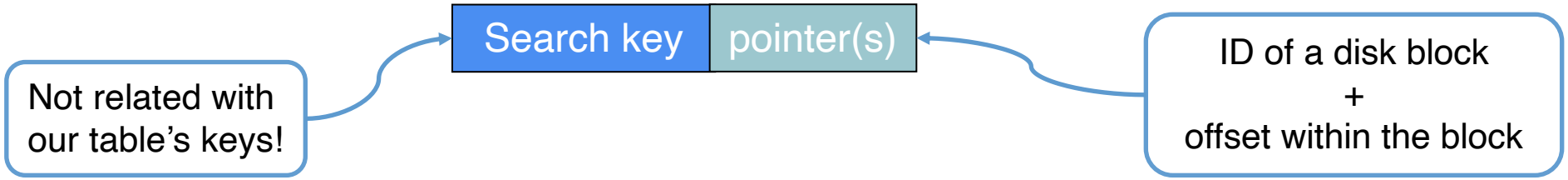
  - To read the whole *student* relation just to find the tuple with ID 22201 is **inefficient**

- **Indices** speed up access to desired data.
  - They are additional structures that we associate with record files.
  - We can have *more than one index per relation*.

- By default, DBMS create indices on:
  - primary keys to check uniqueness constraint
  - foreign keys to improve *joins* (not all of them)

Basic Concepts

- An **index file** consists of *index entries* of the form:

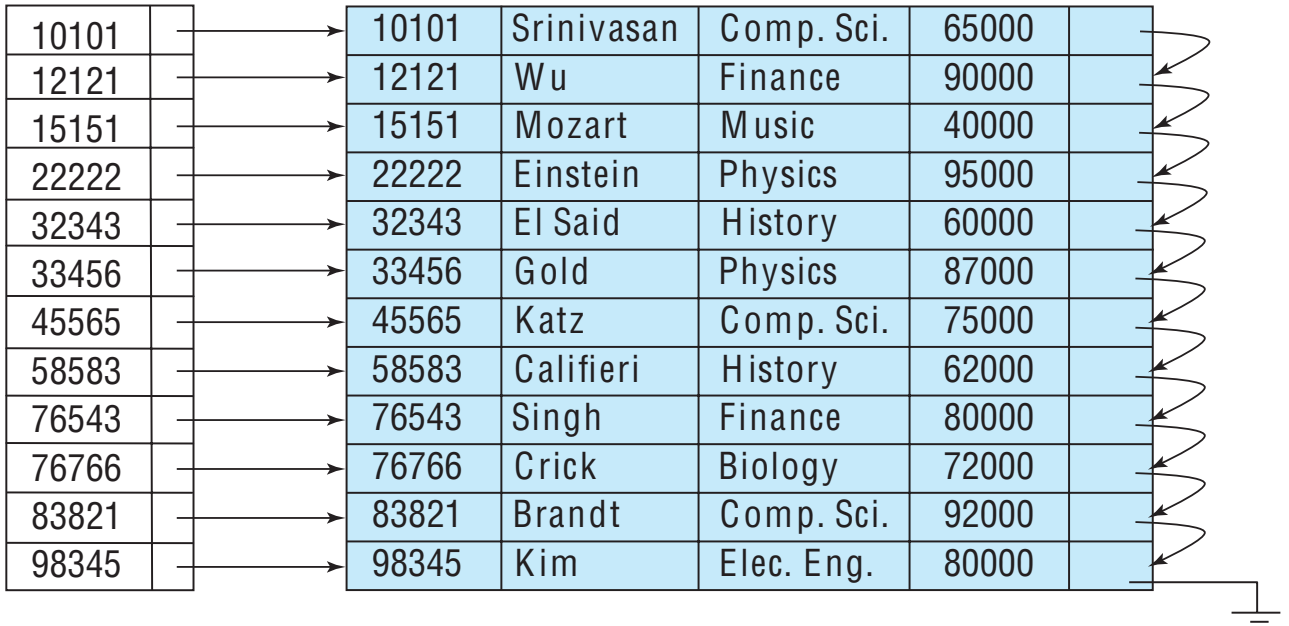


  - **Search key**: (set of) attributes used to search for records.

- Two types of indices:
  - **Ordered**: based on a sorted ordering of *search keys*
  - **Hash**: by uniformly distributing *search keys* across a group of "*buckets*".
    - The assigned bucket is determined by a "*hash function*".

Ordered Indices

- Index entries are sorted by search-key value.
- **Clustering index** (**primary index**): index whose search key also defines the order of the file.



- **Nonclustering index** (**secondary index**): index whose search key doesn't define the order of the file.

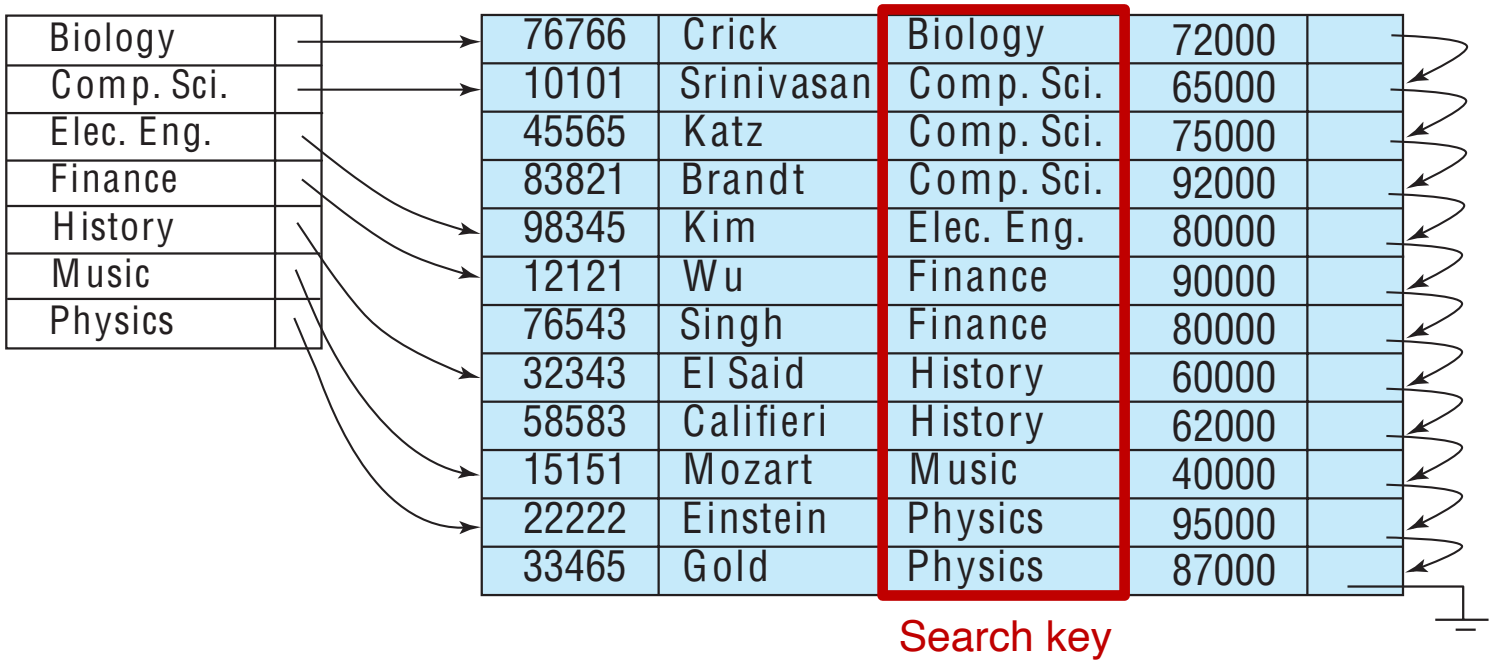
Exercise: How many indices?

- Is it convenient to have many indices on our relation to accelerate *all* the queries?

- How many primary indices can we have on our relation?
  - And secondary indices?

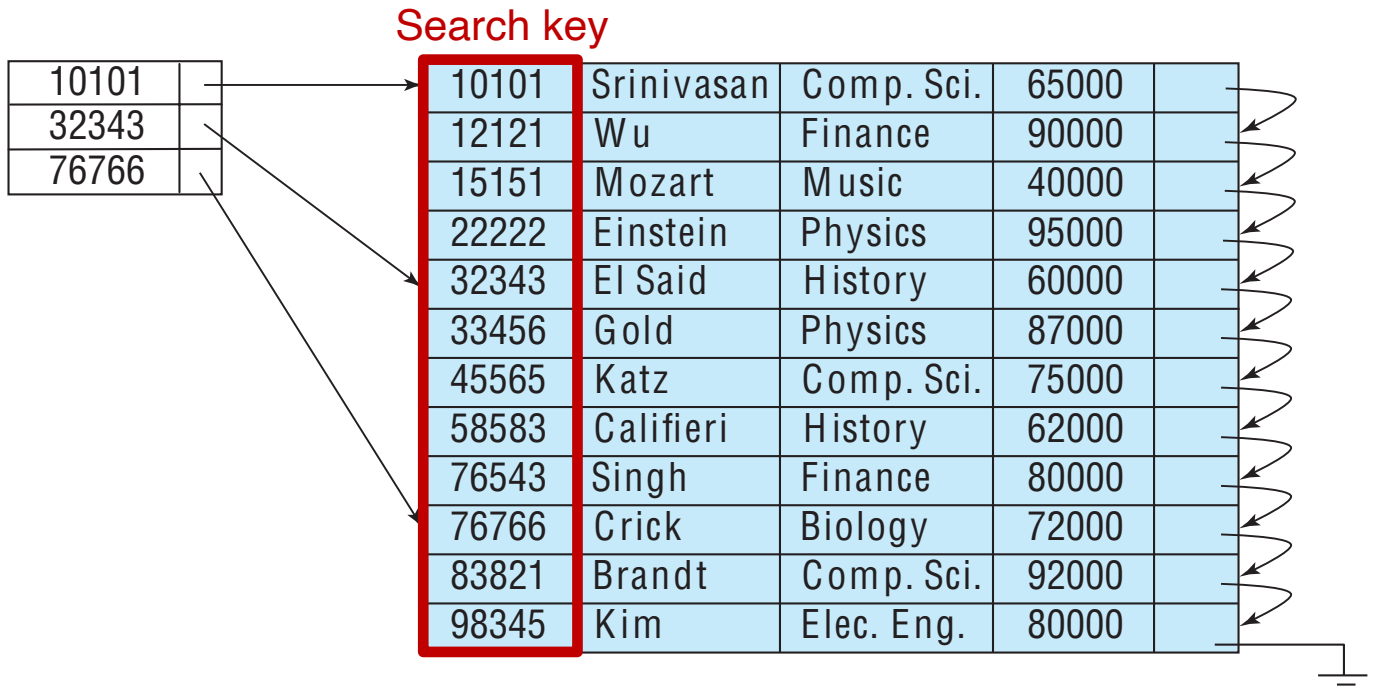
Ordered Indices

- A *primary index* can be **dense** or **sparse**
- **Dense index**: has an index entry for every search-key value in the file.



Ordered Indices

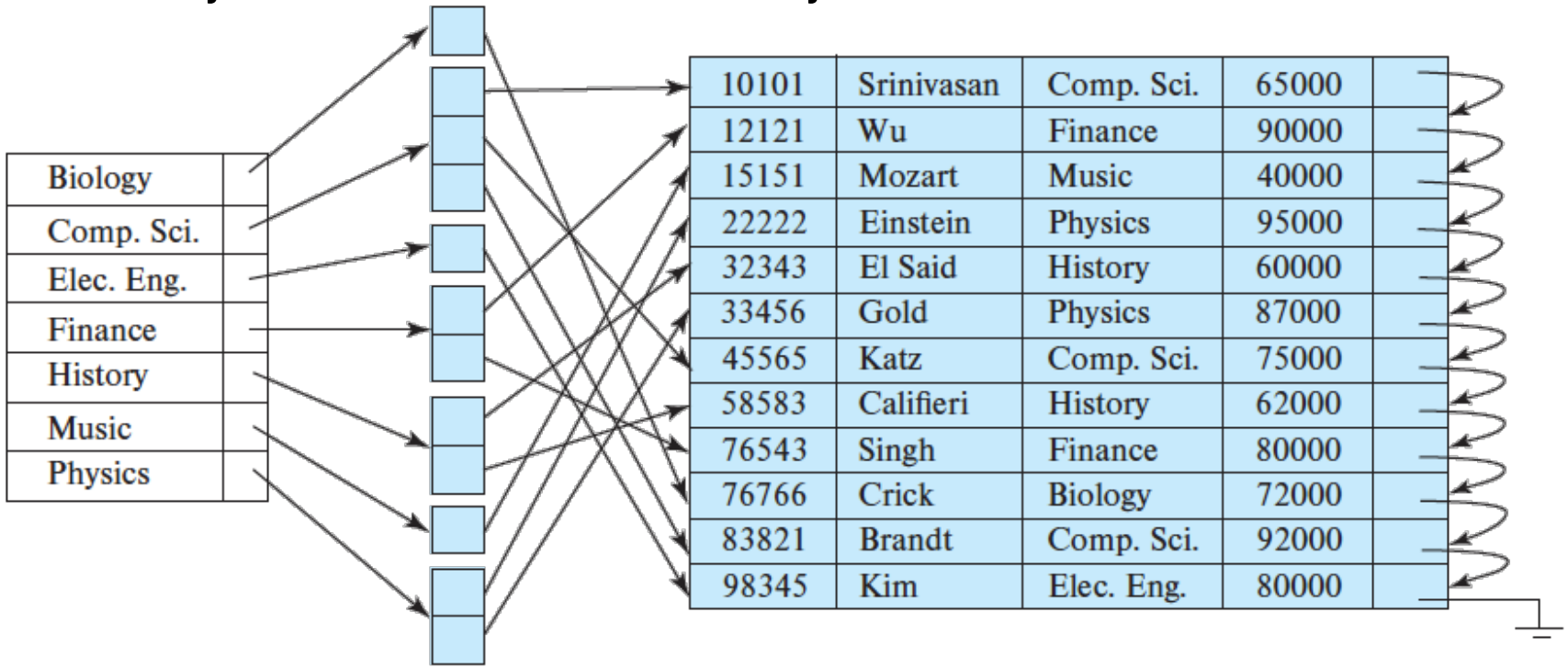
- A *primary index* can be **dense** or **sparse**
- **Sparse Index**: has a few index entries, only for some search-key values



- To locate a record with *search-key K*:
  1. Find index entry with largest *search-key S* where $S < K$
  2. Search for *K* sequentially from the record pointed by *S*

Secondary Indices

- A *secondary index* can only be **dense**
  - If the search-key is not unique, index entry points to a "*bucket*" with pointers to every record with that search-key value.

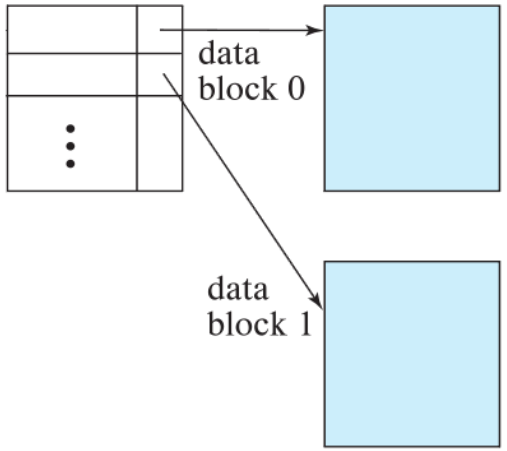


Dense vs Sparse Indices

- Dense indices are generally faster for record location.
- Sparse indices require less space and impose less maintenance overhead for insertions and deletions.

Trade-off between *access time* and *space overhead*

- If index is *primary*, use a sparse index with an *index entry per block*.



Why is this efficient?

- If not, use a sparse index on top of dense index (multilevel index)

Indices

- In general, indices offer **benefits** **for record search**
- But they also imposes **overhead** **on database modification.**

Index Update: Insertion

- We might need to update indices with record insert/delete
- **Insertion** of a record in the relation:
  1. Look for the search-key value of the record to be inserted.
  2. Which type of index is it?
     a) *Dense*:
        - If it is not in the index, add the search-key value
        - If it is the 1st record with that search-key, index entry is updated
        - Otherwise, no change needed
     b) *Sparse*:
        - If a new block is required, create a new index entry with the first search-key value of the new block
        - If the new record has the smallest search-key in its block, index entry is updated
        - Otherwise, no change needed

update = delete+insert

Assuming an index entry per block

Index Update: Deletion

- We might need to update indices with record insert/delete
- **Deletion** of a record from the relation:
  1. Look for the search-key value of the record to delete
  2. Which type of index is it?
     a) *Dense*:
        - If it is the only record with that search-key, delete the index entry
        - If it is the 1st record with that search-key, index entry is updated
        - Otherwise, no change needed
     b) *Sparse:*
        - If there is no index entry for the search-key, no change needed
        - If it exists and other records have the same search-key, the index entry is updated
        - Otherwise, find the *next search-key* in the file and try to create a new index entry for it

update = delete+insert

Assuming an index entry per block

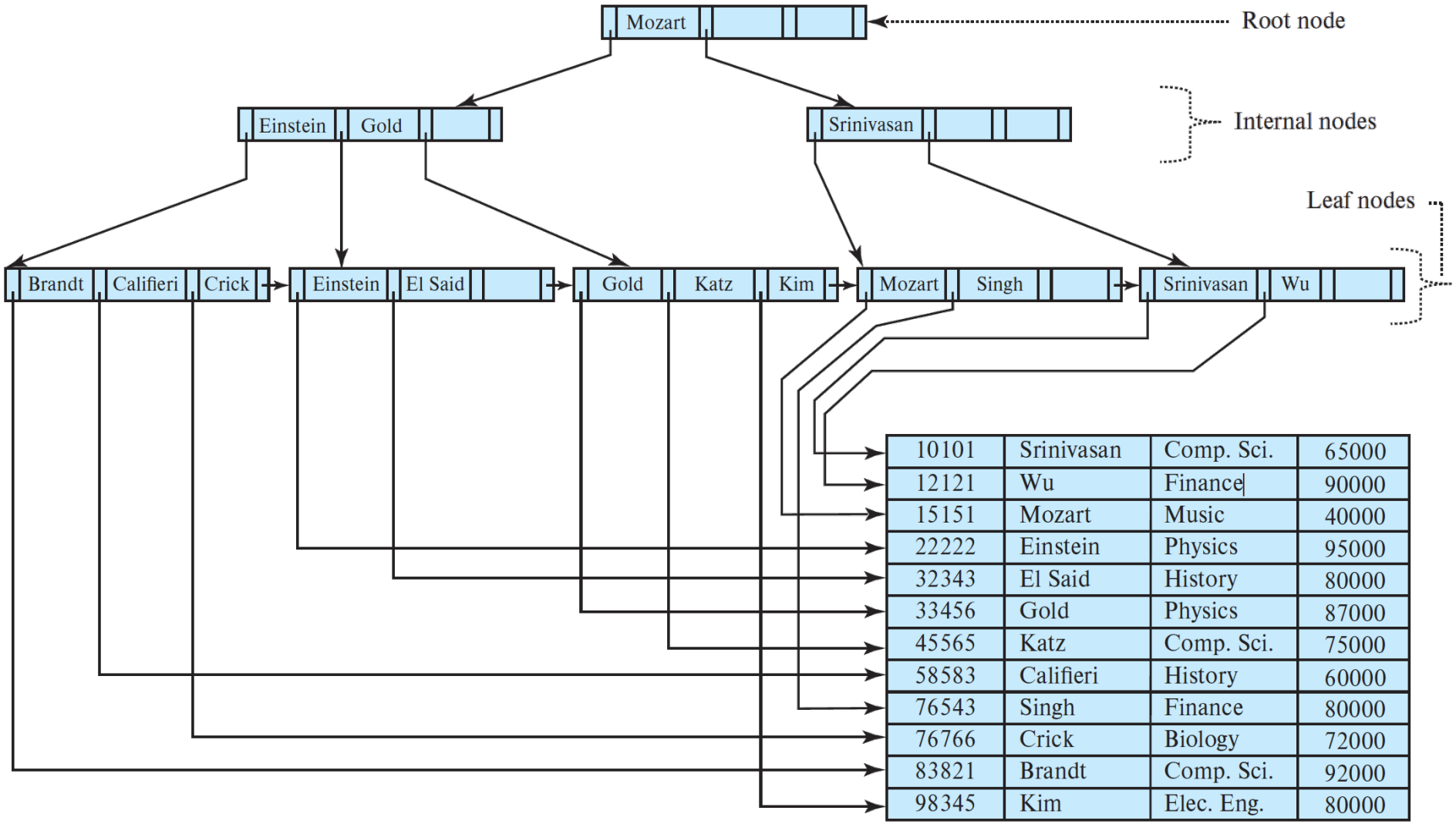
B+-Tree Indices

- Performance with previous indices degrades as the record file grows
- **B+-tree index** structures maintain efficiency even with insertion/deletion of data
  - Balanced trees (equal depth)
  - Reorganizes itself with small, local changes
    - Does not require reorganization of entire file
  - The **root node** has 2 to n children
  - Each **intermediate node**, $\lceil n/2 \rceil$ to n children
  - Each **leaf node** has $\lceil (n-1)/2 \rceil$ to $n-1$ values
  - Extra insertion and deletion (computation) **overhead**, space **overhead**.

The maximum length n of a node is fixed for a tree

Extensively used!

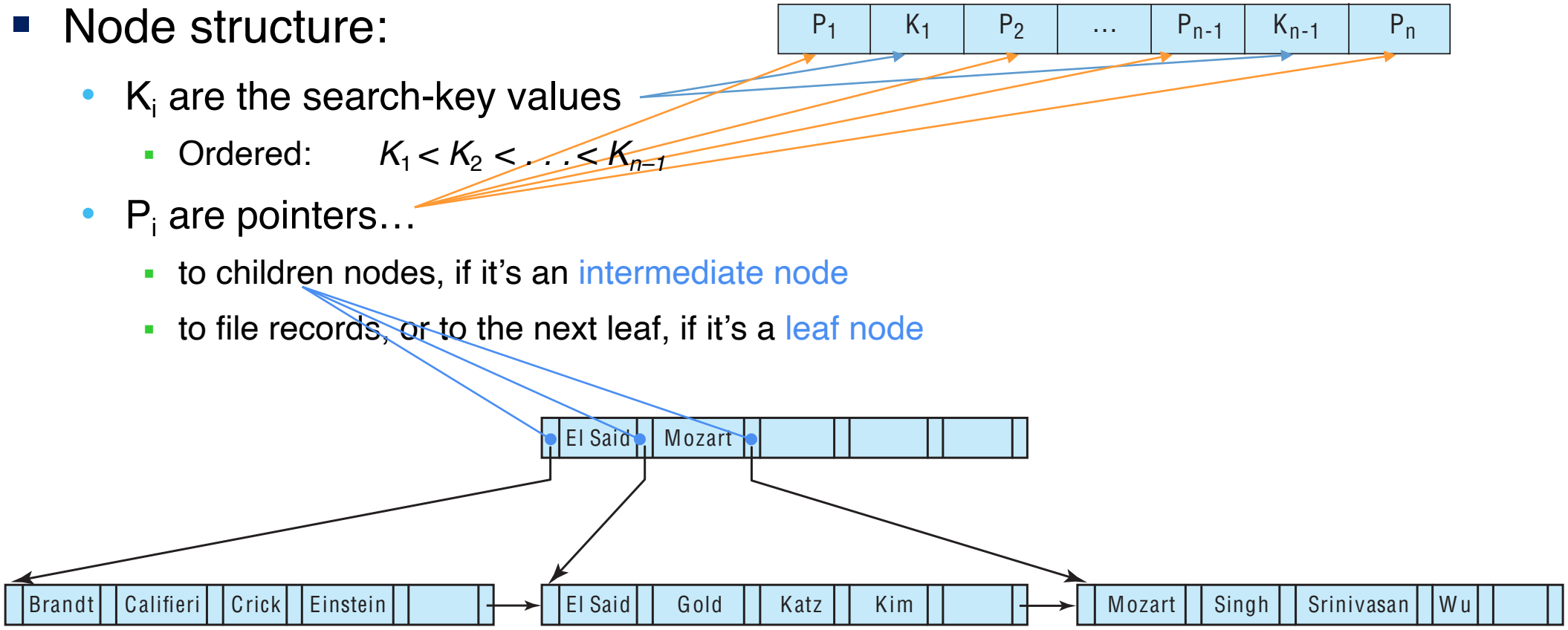
B+-Tree Indices



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

B+-Tree Node Structure

- Node structure:
  - K_i are the search-key values
    - Ordered: $K_1 < K_2 < \ldots < K_{n-1}$
  - P_i are pointers…
    - to children nodes, if it's an intermediate node
    - to file records, or to the next leaf, if it's a leaf node

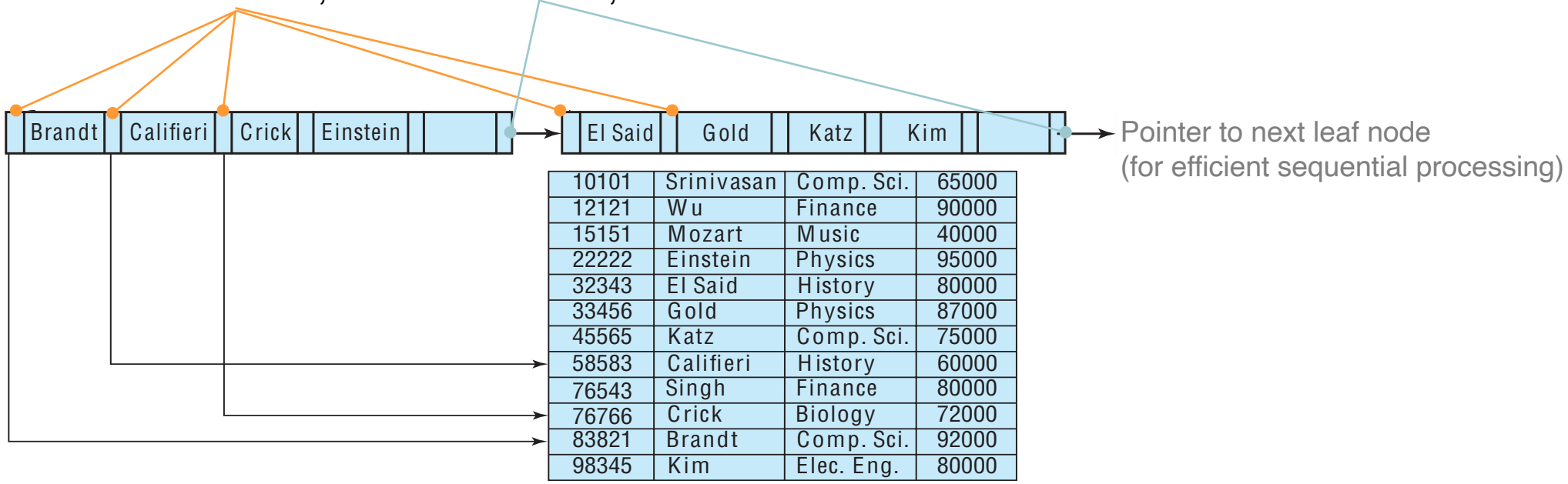


Any search-key k' in the subtree growing under P_i is: $k' < K_i$
Any search-key k' in the subtree growing under P_{i+1} is: $k' >= K_i$

B+-Tree Node Structure

- Node structure:

  - K_i are the search-key values
    - Ordered: $K_1 < K_2 < \ldots < K_{n-1}$
  - P_i are pointers…
    - to children nodes, if it's an intermediate node
    - to file records, or to the next leaf, if it's a leaf node

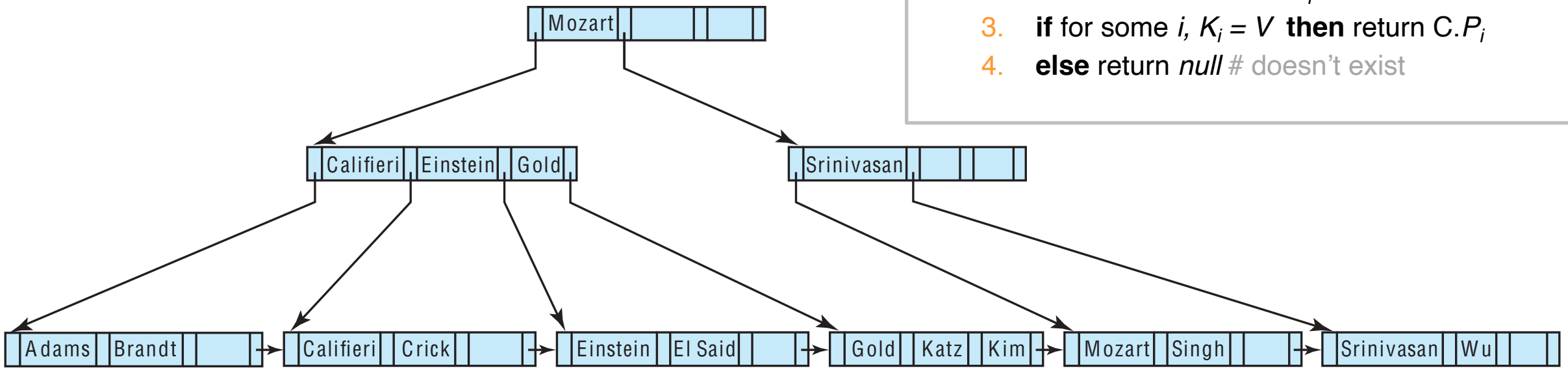


B+-Tree Structure

- The B+-tree contains a relatively small no. levels
  - **Searches** can be conducted **efficiently**
  - L^{th} level has at least $2*\lceil n/2 \rceil^{(L-1)}$ values
  - For K search-key values, depth is lower than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
  - E.g., with 10^6 search-keys and $n = 100$, tree depth is not longer than $log_{50}(10^6) = 4$
    - A node is generally the same size as a disk block (4 KB)
    - If we use 40 bytes per index entry, n =100

Queries on B+-Trees

Find "Katz"

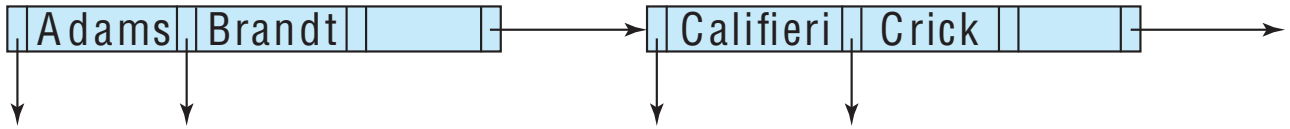


function *find(v)*

1. *C=root*
2. **while** (C is not a leaf node)
   1. Let *i* be smallest number s.t. $V \leq C.K_i$.
   2. **if** there is no such number *i then*
   3. Set C = *last non-null pointer in* $C.P_j$
   4. **else if** ($V = C.K_i$) Set C = $C.P_{i+1}$
   5. **else** Set $C = C.P_i$
3. **if** for some *i*, $K_i = V$ **then** return $C.P_i$
4. **else** return *null* # doesn't exist

Updates on B+-Trees: Insertion

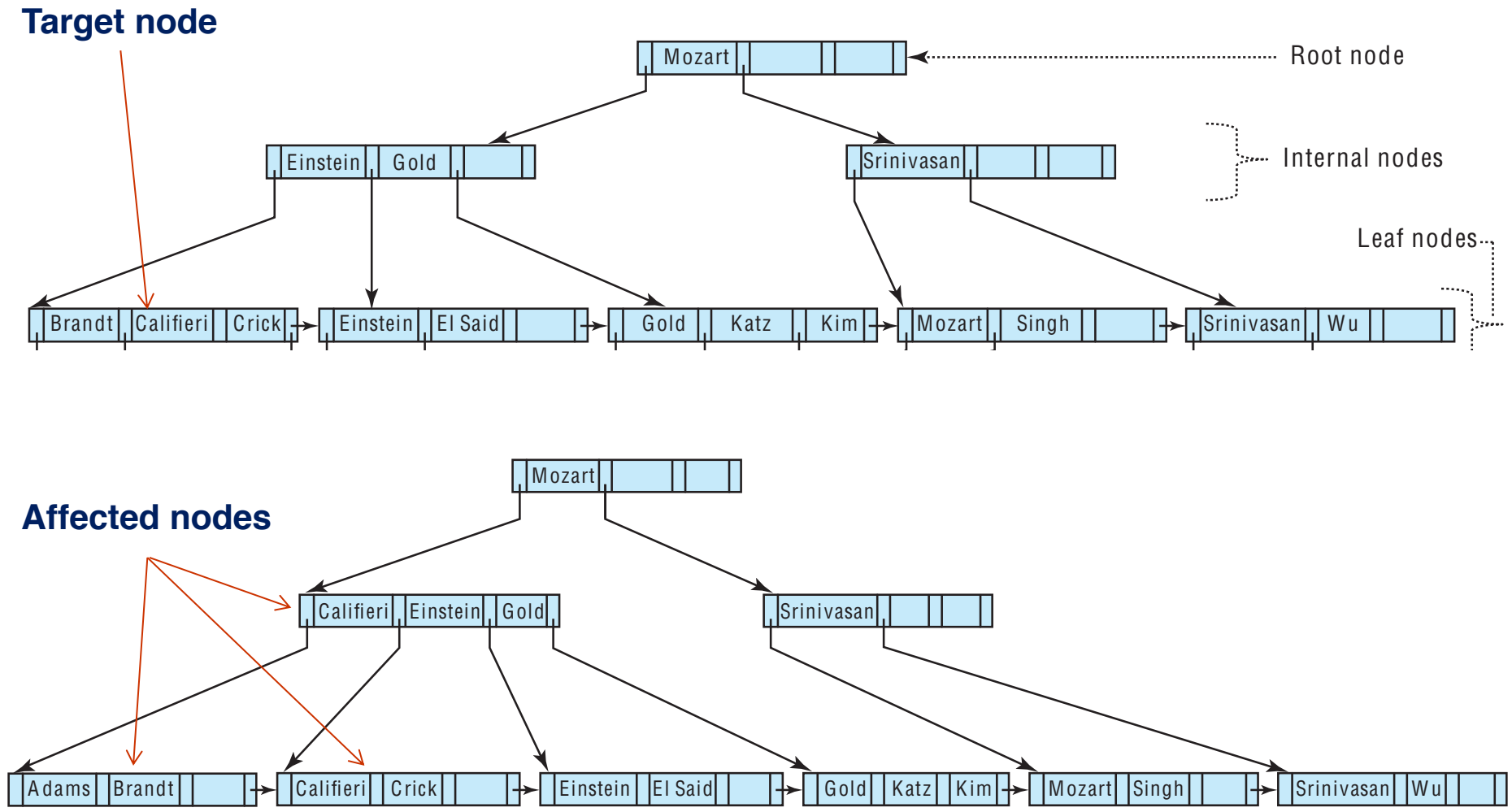
- New entry with pointer P and search-key value V
- Find leaf where the search-key V should be
  1. Is there room? Insert (V, P) there
  2. No? Split the leaf node:
     - Leave the first $\lceil n/2 \rceil$ pairs (V_i, P_i) in the original node N, and move the rest to a new node, M.
     - Add a pointer from N to M at last position of N
     - Let V_M be the smallest search-key value in M, and P_M a pointer to M. Add (V_M, P_M) to the parent of the node being split.



     - The split might propagate up if the parent is full too

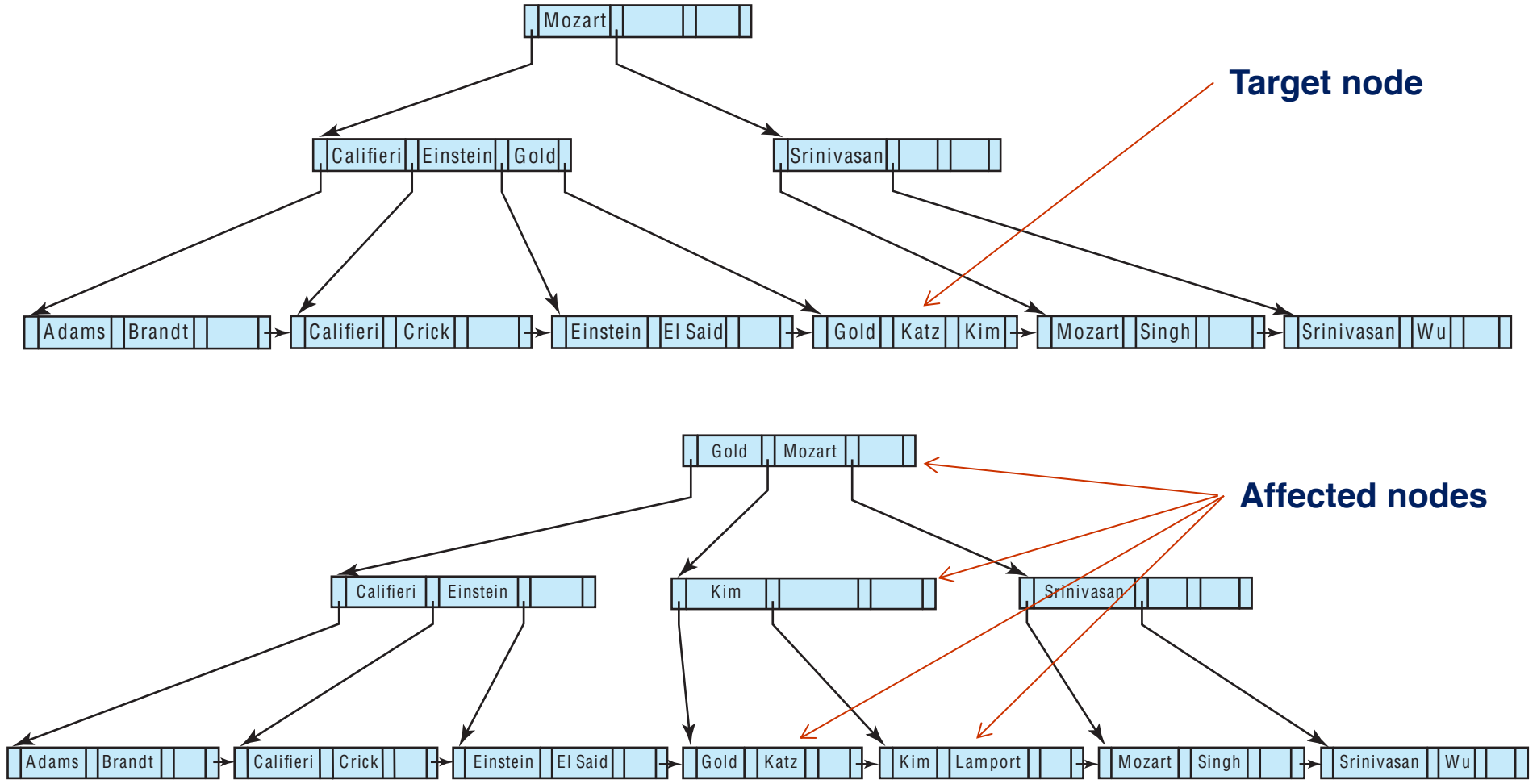
Worst case: it propagates up to the root, and this is full too. It needs to be split, increasing the depth of the tree by 1

Updates on B+-Trees: Insertion



B+-Tree *before* and *after* **insertion** of "Adams"

Updates on B+-Trees: Insertion



B+-Tree *before* and *after* **insertion** of "Lamport"

Bulk Loading on B$^+$-Trees

- **Bulk loading**: to insert a large no. entries at a time
- Inserting entries *one-at-a-time* into a B$^+$-tree requires $\geq$ 1 IO per entry
  Very inefficient for bulk loading
- Alternative #1: Sort entries and insert in sorted order
  - Insertion will go to existing block (or cause a split)
  - Much improved IO performance, but most leaves end half full
- Alternative #2: **Bottom-up construction**
  1. Sort entries and create firstly the leaf level separating entries in leaves
  2. Then, create the intermediate levels upwards
  - **Very** efficient if it is done from scratch (initial tree empty)

Exercise: ordered insertion

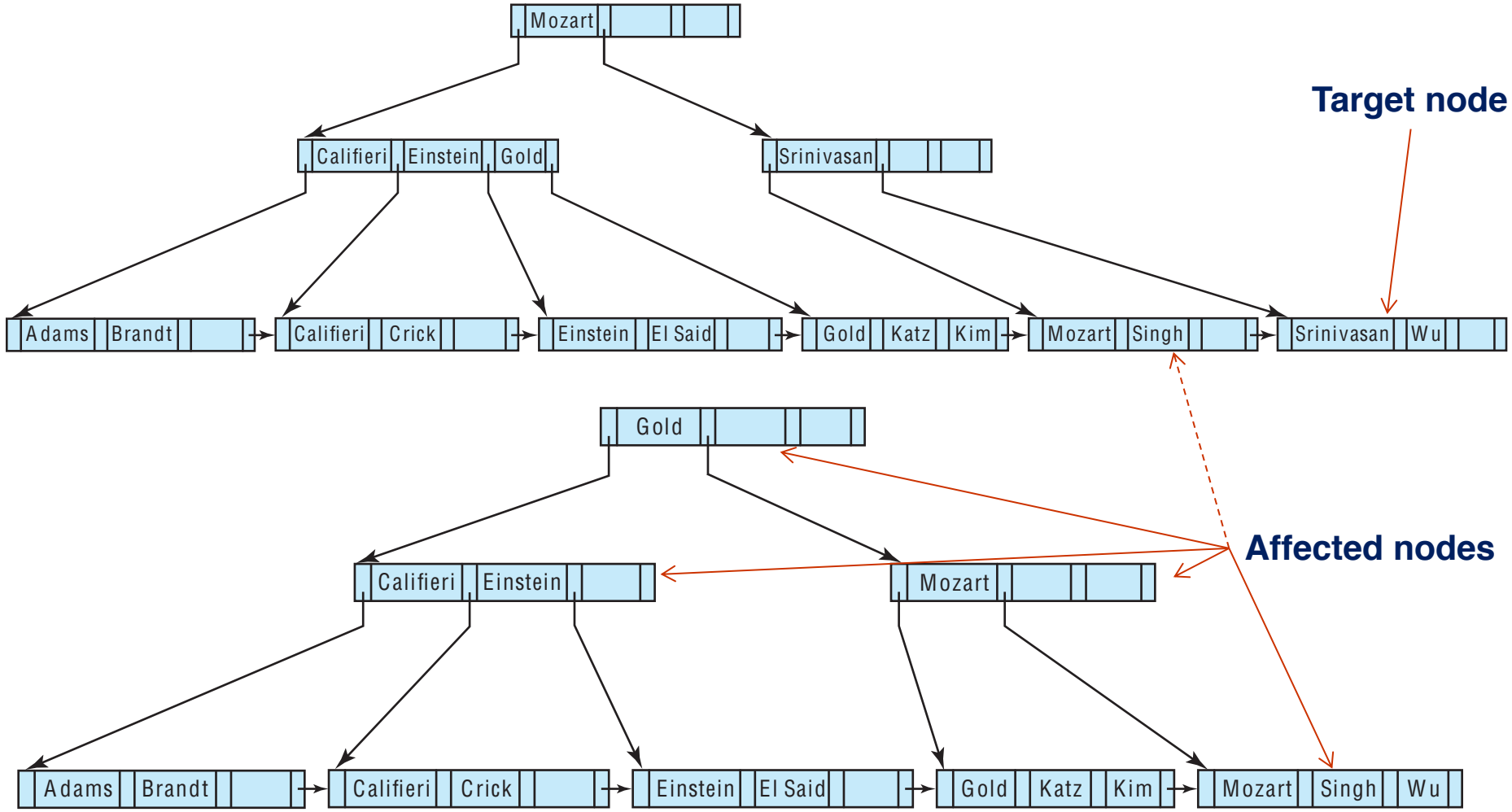
- In the previous *alternative #1*, where sorted entries are inserted sequentially, we said that most leaves end half full.
  - Could you explain why?

Updates on B+-Trees: Deletion

Delete entry with search-key *V* and pointer *P*:

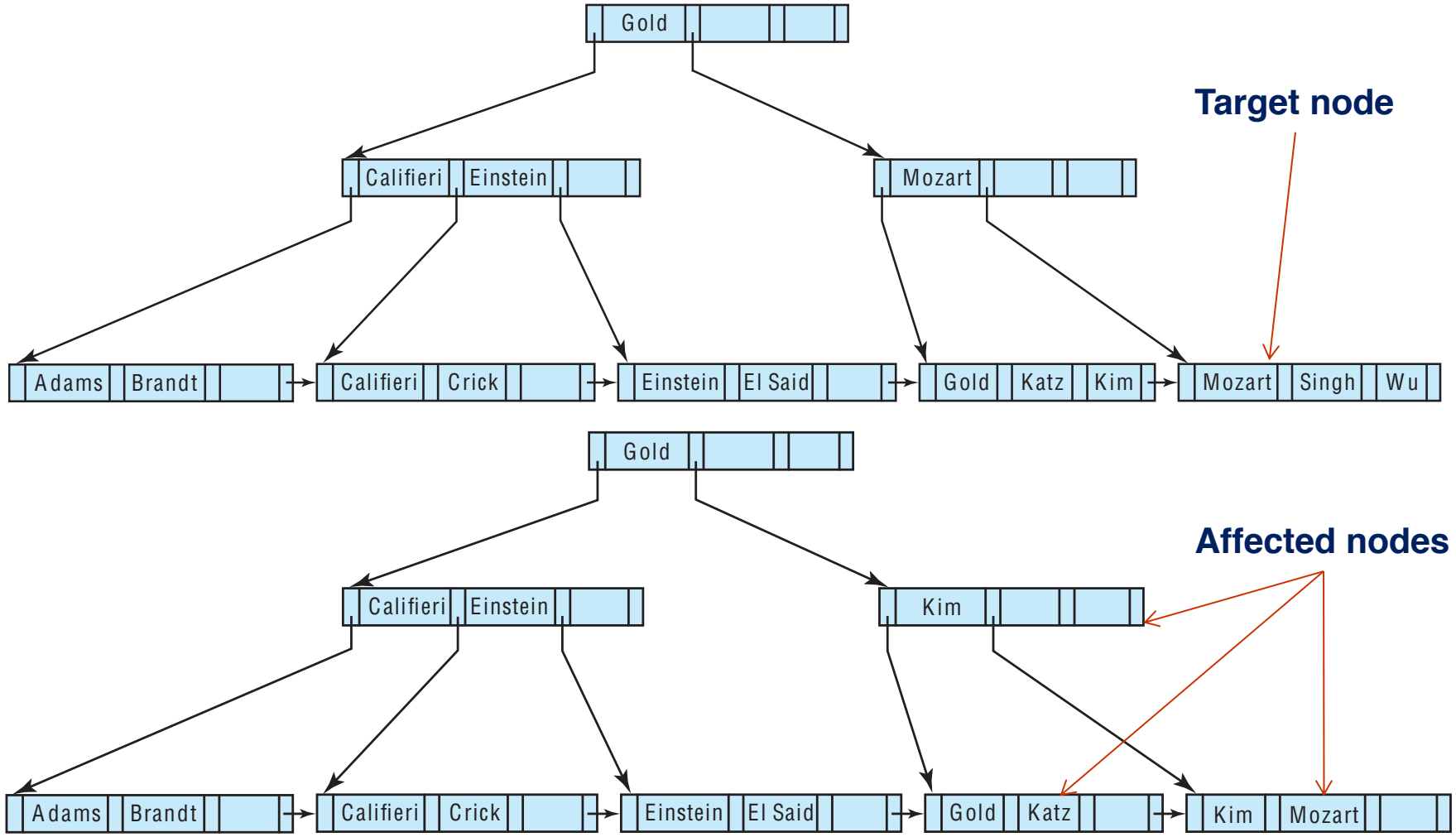
1. Locate *V* in the tree
2. Remove (*V, P*) from the leaf node
3. If no. entries in the leaf is now $< \lceil (n-1)/2 \rceil$:
    1. If there is enough free space, ***merge with a sibling node***
        - Combine all search-keys into a single node, separated by search-key K_{i-1} from parent node where P_i is the pointer to the empty node
        - Delete pair (K_{i-1}, P_i) *from* parent node
        - The deletion might propagate up if the parent is *empty* too
    2. Otherwise, **redistribute pointers**:
        - Move among siblings so that both have at least $\lceil n/2 \rceil$ child pointers
        - The search-key which was between pointers P_i and P_{i+1} now separated in two siblings moves to the parent. The search-key previously in the parent is moved to the corresponding sibling.

Updates on B+-Trees: Deletion



B+-Tree *before* and *after* **deletion** of "Srinivasan"

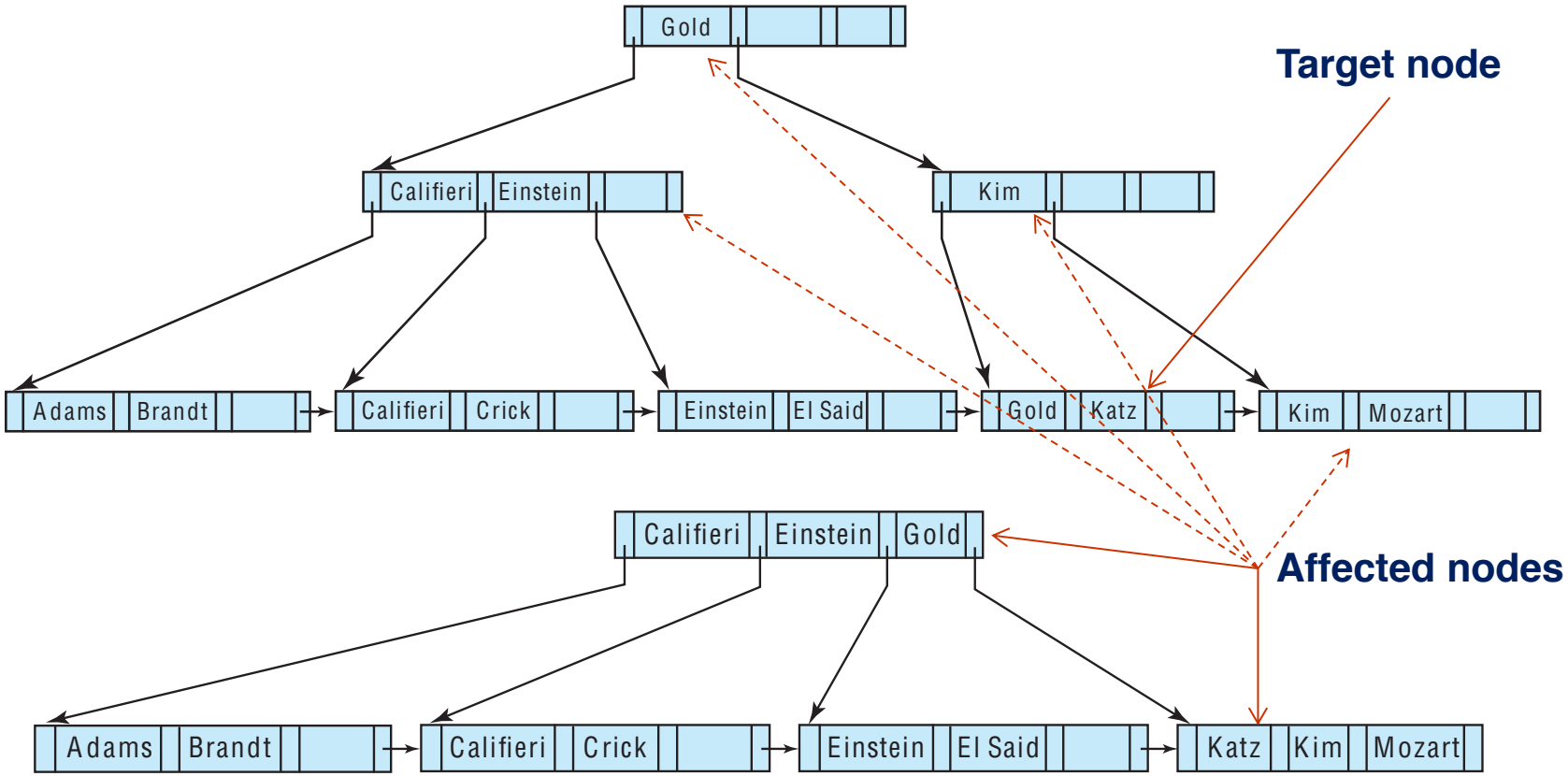
Updates on B+-Trees: Deletion



B+-Tree *before* and *after* **deletion** of "Singh" and "Wu"

Updates on B+-Trees: Deletion

If the *root* node has only one pointer after (cascade) deletion, it is deleted and the sole child becomes the root.



B+-Tree *before* and *after* **deletion** of "Gold"

Updates on B+-Trees: Complexity

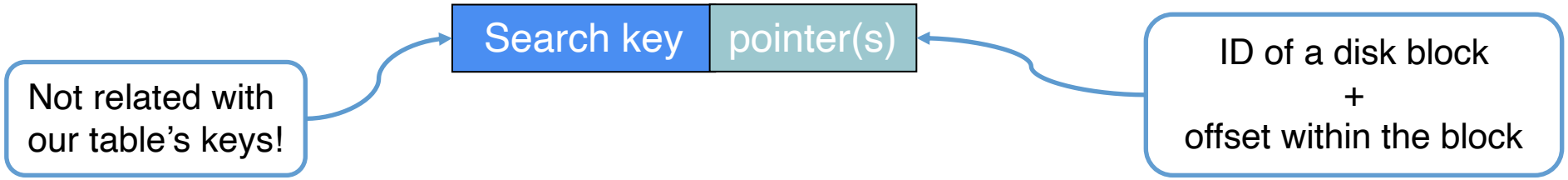
- **Cost** of insertion/deletion of an entry is proportional to tree-depth
  In terms of number of I/O operations
  - Good news: worst case complexity is $O(\log_{\lceil n/2 \rceil}(K))$
    K: number of records, n: max. number of pointers

- In practice, number of I/O operations is lower:
  - Internal nodes tend to be in buffer
  - Splits/merges are rare, most insert/delete operations only affect a leaf

- Av. **node occupancy** depends on insertion order
  - 2/3 with random order, ½ with insertion in sorted order

Basic Concepts

- An **index file** consists of *index entries* of the form:



  - **Search key**: (set of) attributes used to search for records.

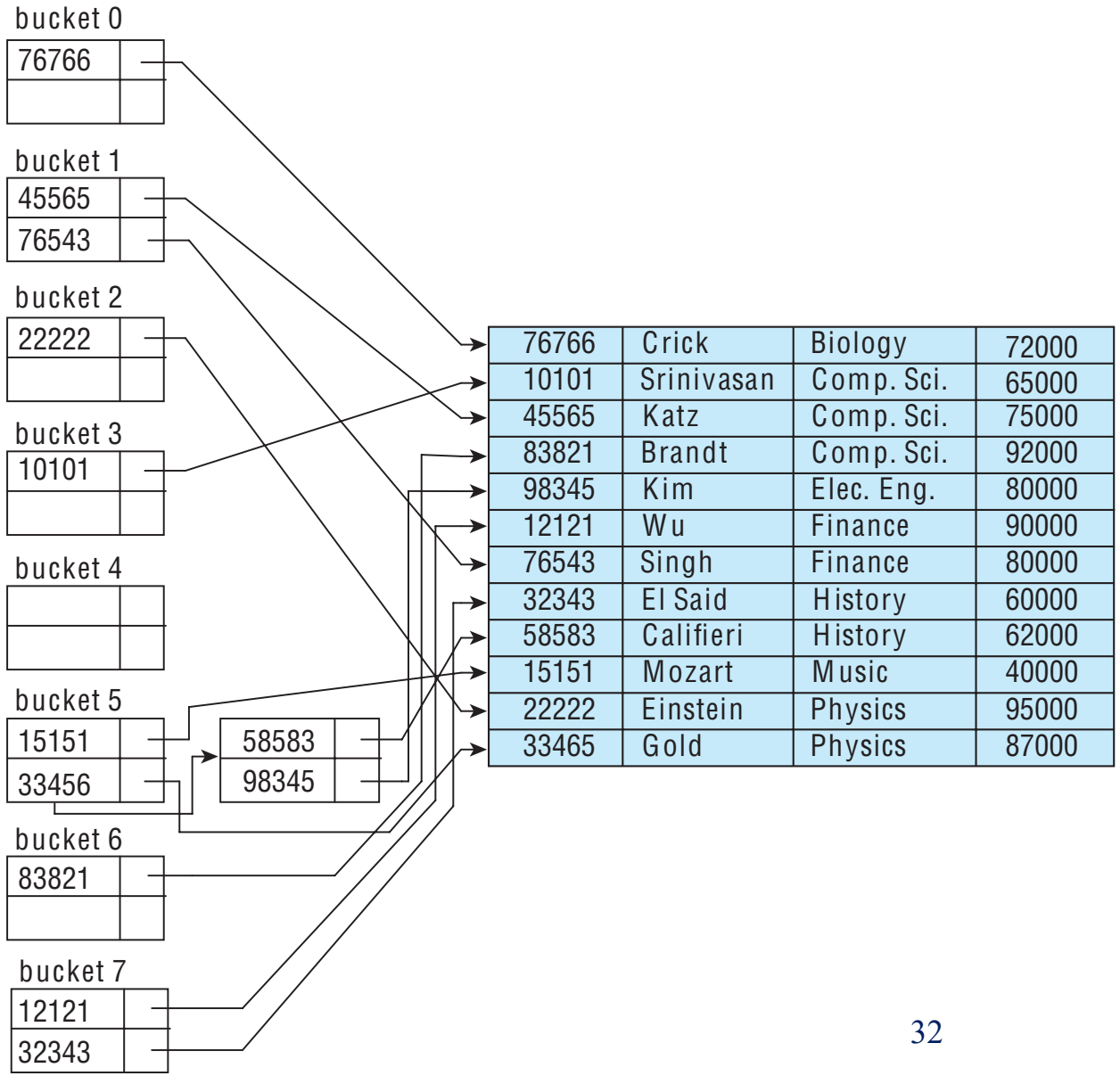
- Two types of indices:
  - **Ordered**: based on a sorted ordering of *search keys*
  - **Hash**: by uniformly distributing *search keys* across a group of "*buckets*".
    - The assigned bucket is determined by a "*hash function*".

Hashing

- **Bucket**: storage portion that contains several index entries
  - A linked-list of blocks
- Search-key values are assigned to buckets using a **hash function**

h : {K: set of all search-keys} → {B: set of all bucket addresses}

- Different search-keys may be mapped to the same bucket
  - Entire bucket is searched sequentially to locate an entry



Ordered vs Hash Indexing

- Appropriate type of queries:
  - Hashing better at retrieving records with a specific key-search value.
  - Ordered indices better for range queries

- In practice: Hashing is not that popular
  - PostgreSQL supports hash indices, but discourages its use due to poor performance
  - Oracle supports static hash organization, but not hash indices
  - SQLServer supports only B^+-trees

Exercise: range queries in hash indices

- Why is a hash index not the best choice for a search key on which range queries are likely?

Indexing