SNS COLLEGE OF TECHNOLOGY

COIMBATORE

AN AUTONOMOUS INSTITUTION

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A' Grade

Approved by AICTE New Delhi & affiliated to the Anna University, Chennai

DEPARTMENT OF MCA

Course Name : 19CAT609 - DATA BASE MANAGEMENT SYSTEM

Class : I Year / II Semester

Unit III - DATA STORAGE

Topic IV – Indexing - Tree structured indexing

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute or set of attributes used to look up records in a
  •file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer • →

- Index files are typically much smaller than the original file
- Two basic kinds of indices:

  - **Ordered indices:** search keys are stored in some sorted order
  - **Hash indices:** search keys are distributed uniformly across
    •“buckets” using a “hash function”.

Basic Concepts

- Indices are typically much smaller than the original, e.g.:
  - Table of contents in a book
  - Index in a book
  - Catalog in a library
  - Inventory in a warehouse

3

Ordered Indices

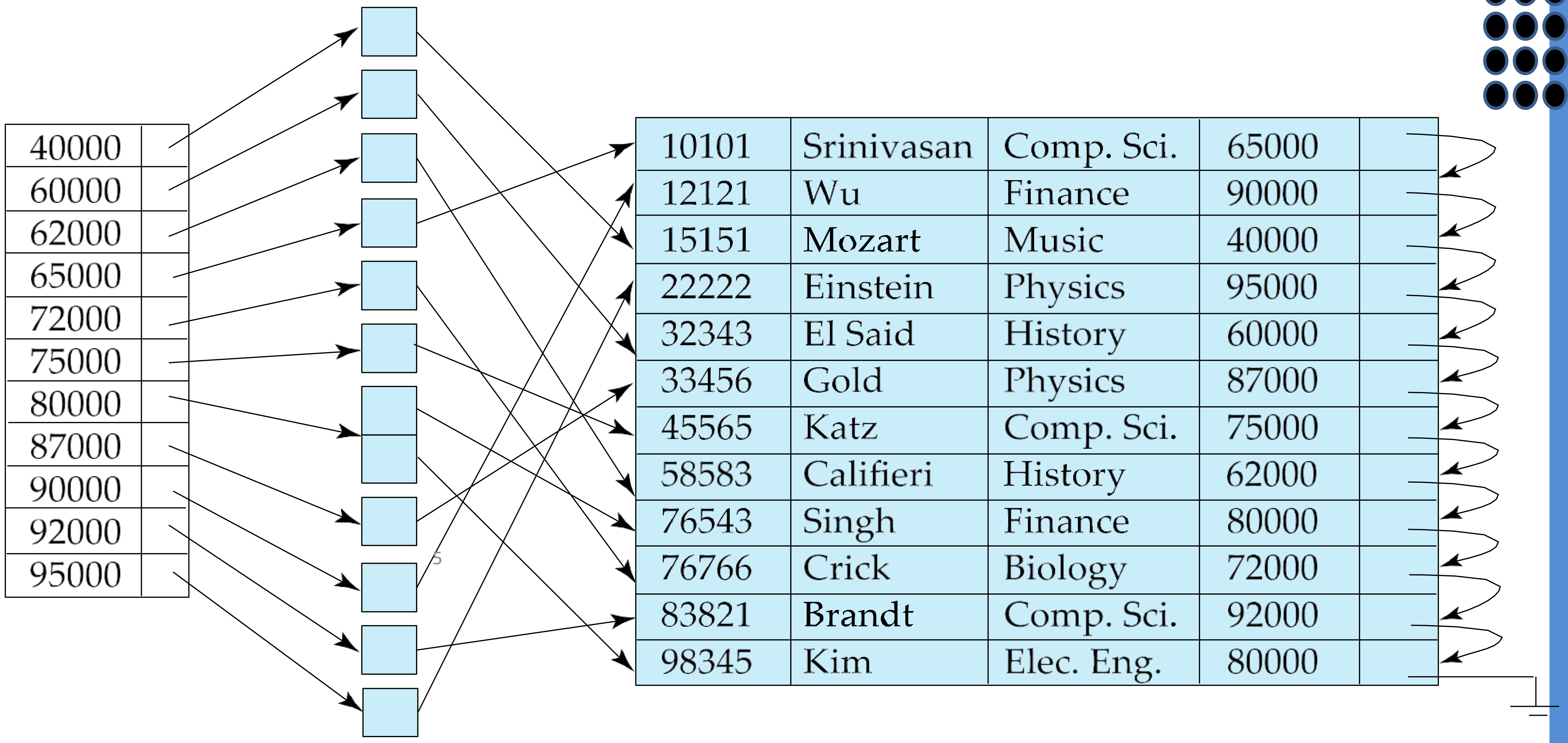
- In an **ordered index,** index entries are stored sorted on the search key value. E.g., author catalog in library.
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index**.**
- Index-sequential file**:** ordered sequential file with a primary index.

4

Secondary Indices Example

Secondary index on
salary* field of *instructor

- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense



Primary and Secondary Indices

- Indices offer substantial benefits when searching for records.
- BUT: Updating indices imposes overhead on database modification --when a file is modified, every index on the file must be updated,
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - Each record access may fetch a new block from disk
  - Block fetch requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access

Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
  - Example 1: In the *instructor* relation stored sequentially by ID, we may want to find all instructors in a particular department
  - Example 2: as above, but where we want to find all instructors with a specified salary or with salary in a specified range of values

7

- We can have a secondary index with an index record for each search-key value

B^+-Tree Index

B^+-tree indices are an alternative to indexed-sequential files.

- Disadvantage of indexed-sequential files
  - performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required.
- Advantage of B^+-tree index files:
  - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.

8

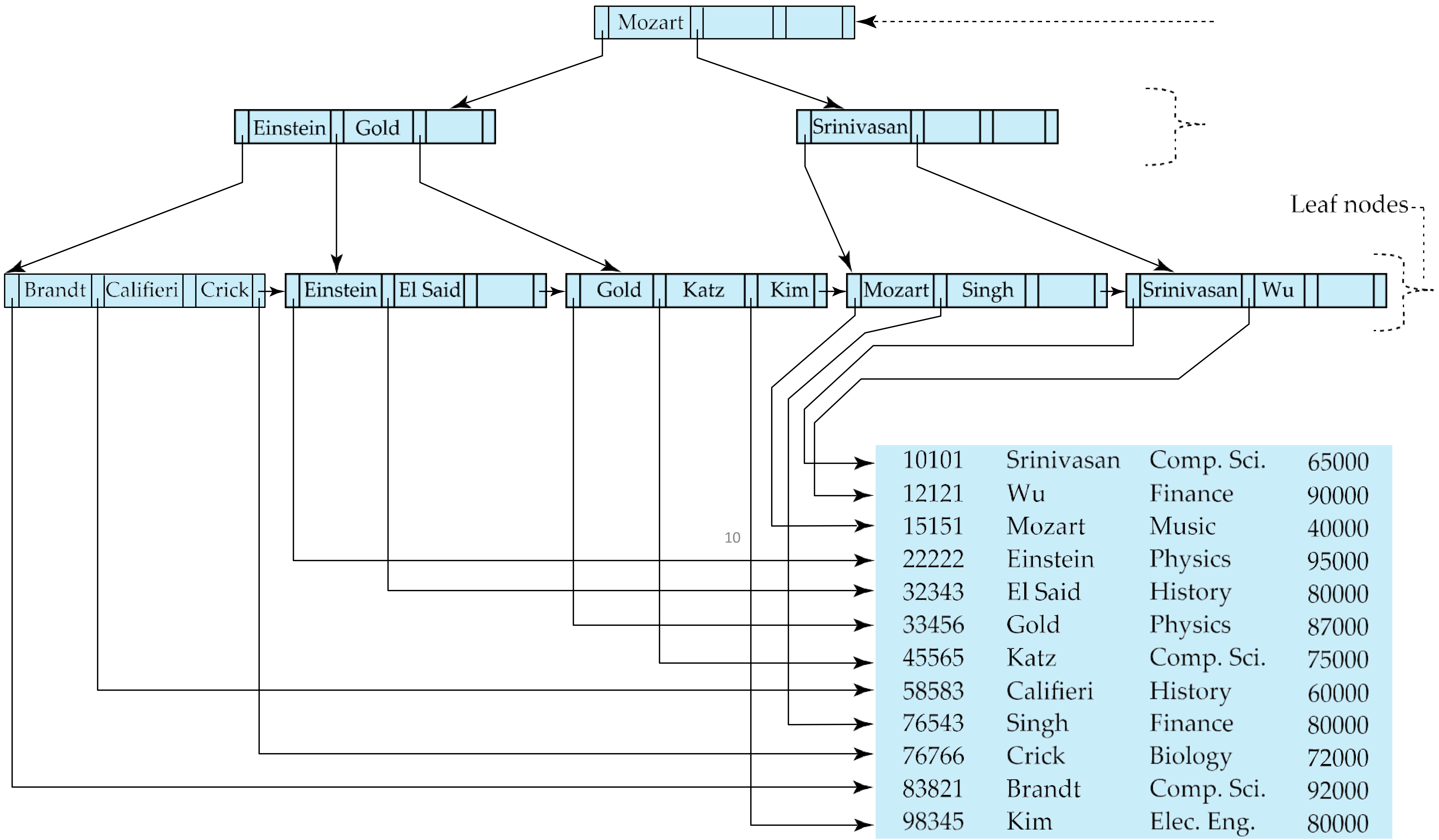
- (Minor) disadvantage of B^+-trees:
  - extra insertion and deletion overhead, space overhead.
- Advantages of B^+-trees outweigh disadvantages
  - B^+-trees are used extensively

B^+-Tree Index

B^+-tree indices are an alternative to indexed-sequential files.

- Disadvantage of indexed-sequential files
  - performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required.
- Advantage of B^+-tree index files:
  - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B^+-trees:
  - extra insertion and deletion overhead, space overhead.
- Advantages of B^+-trees outweigh disadvantages
  - B^+-trees are used extensively

Example of B+-Tree



B^+-Tree Index Files (Cont.)

A B^+-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between ⌈n/2⌉ and n children.
- A leaf node has between ⌈(n–1)/2⌉ and n–1 values
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and (n–1) values.

11

Typical node

P	K_1	P_2	…	P_{n-}	K_{n-}	P_n

K_i are the search-key values

- P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).

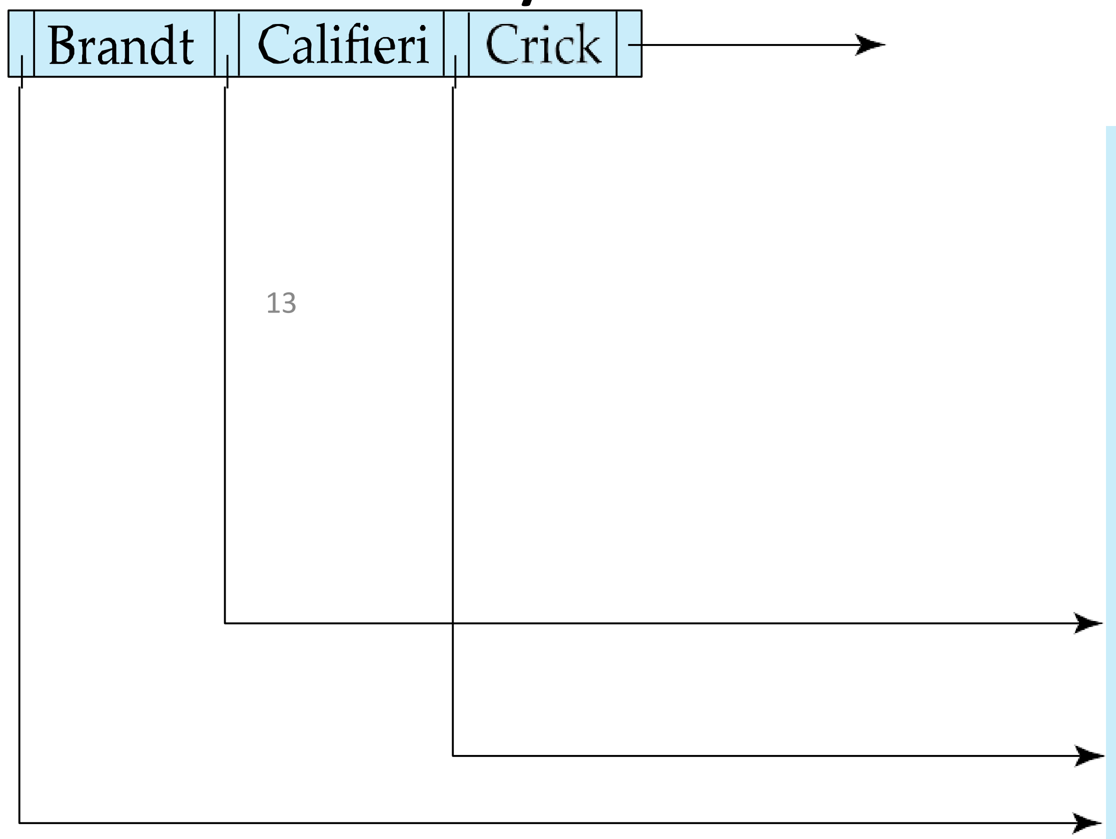
- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < . . . < K_{n–1}$$

(Initially assume no duplicate keys, address duplicates later)

Properties of a leaf node:

- For $i = 1, 2, \ldots, n-1$, pointer P_i points to a file record with search-key value K_i,
- If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than or equal to L_j's search-key values
- P_n points to next leaf node in search-key order



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

- Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close.
- The non-leaf levels of the B+-tree form a hierarchy of sparse indices.
- The B+-tree contains a relatively small number of levels
  - Level below root has at least 2* ⌈n/2⌉ values
  - Next level has at least 2* ⌈n/2⌉ * ⌈n/2⌉ values
  - .. etc.
  - If there are K search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
  - thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time.

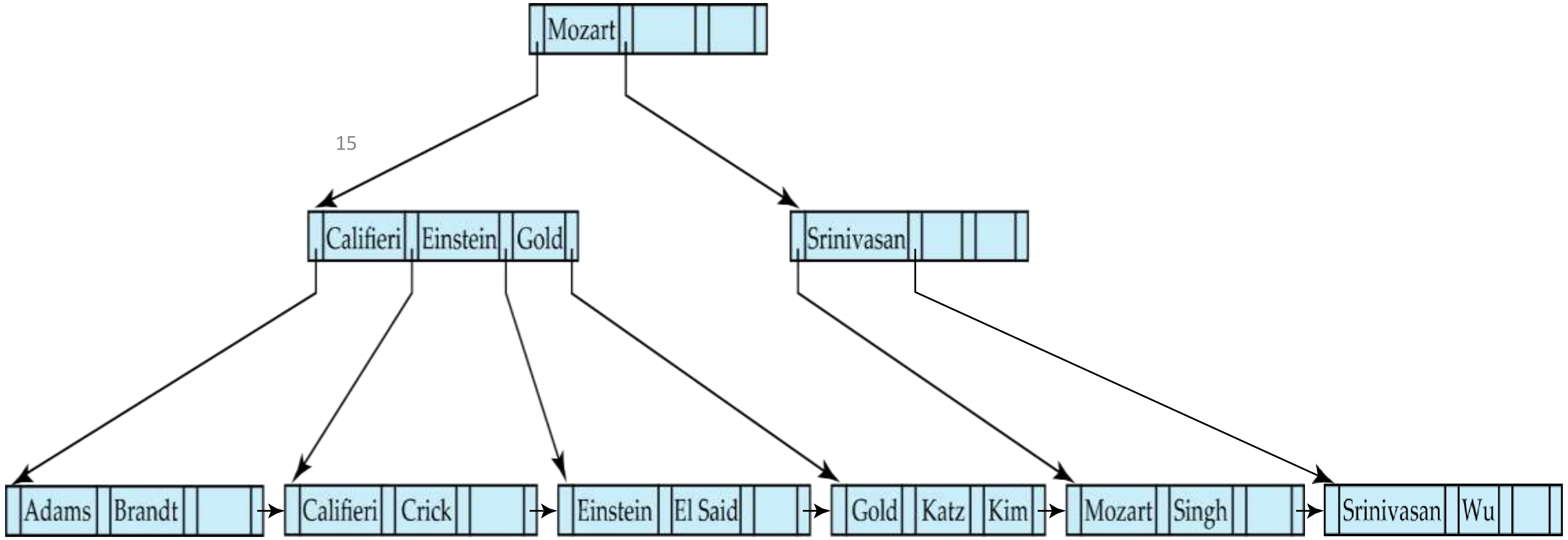
14

Queries on B⁺-Trees

- Find record with search-key value *V.*
  1. *C=root*
  2. While C is not a leaf node {
     1. Let *i* be least value s.t. $V \leq K_i$.
     2. If no such exists, set *C = last non-null pointer in C*
     3. Else { if ($V = K_i$) Set C = P_{i+1} else set $C = P_i$}

     }
  3. Let *i* be least value s.t. $K_i = V$
  4. If there is such a value *i,* follow pointer P_i to the desired record.
  5. Else no record with search-key value *k* exists.

15



Handling Duplicates

- With duplicate search keys
  - In both leaf and internal nodes,
    - we cannot guarantee that $K_1 < K_2 < K_3 < . . . < K_{n–1}$
    - but can guarantee $K_1 \leq K_2 \leq K_3 \leq . . . \leq K_{n–1}$
  - Search-keys in the subtree to which Pi points
    - are $\leq K_{i,}$, but not necessarily < Ki,
    - To see why, suppose same search key value V is present in two leaf node Li and Li+1. Then in parent node Ki must be equal to V

16

Queries on B^{+}-Trees (Cont.)

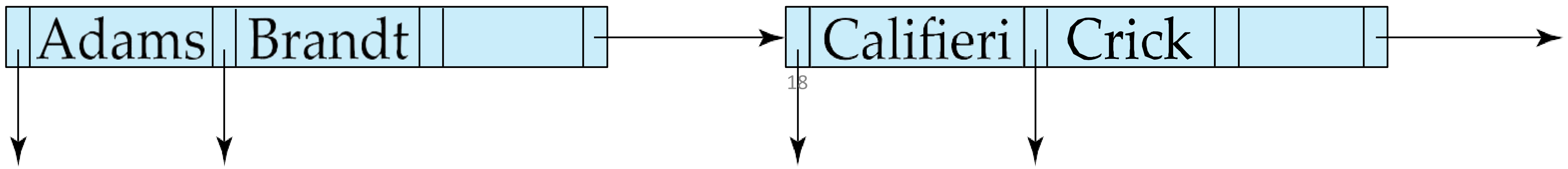
- If there are *K* search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and *n* is typically around 100 (40 bytes per index entry).
- With 1 million search key values and *n* = 100
  - at most $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup.
- 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

17

Queries on B$^+$-Trees (Cont.)

- Splitting a leaf node:
  - take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order.

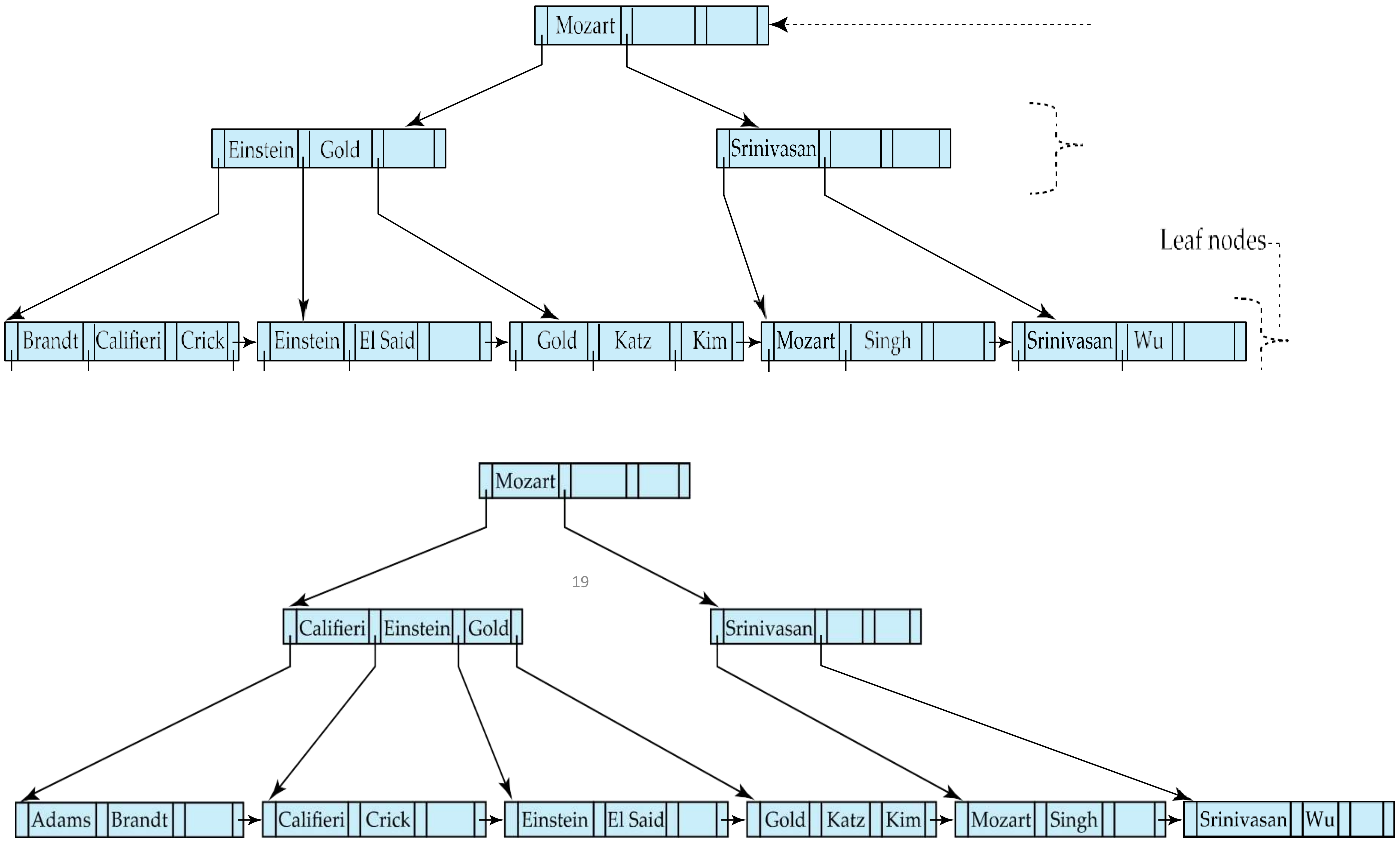
  Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be *p,* and let *k* be the least key value in *p.* Insert (*k*,*p*) in the parent of the node being split.
  - If the parent is full, split it and **propagate** the split further up.
- Splitting 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.



Result of splitting node containing Brandt, Califieri and Crick on inserting Adams

Next step: insert entry with (Califieri,pointer-to-new-node) into parent

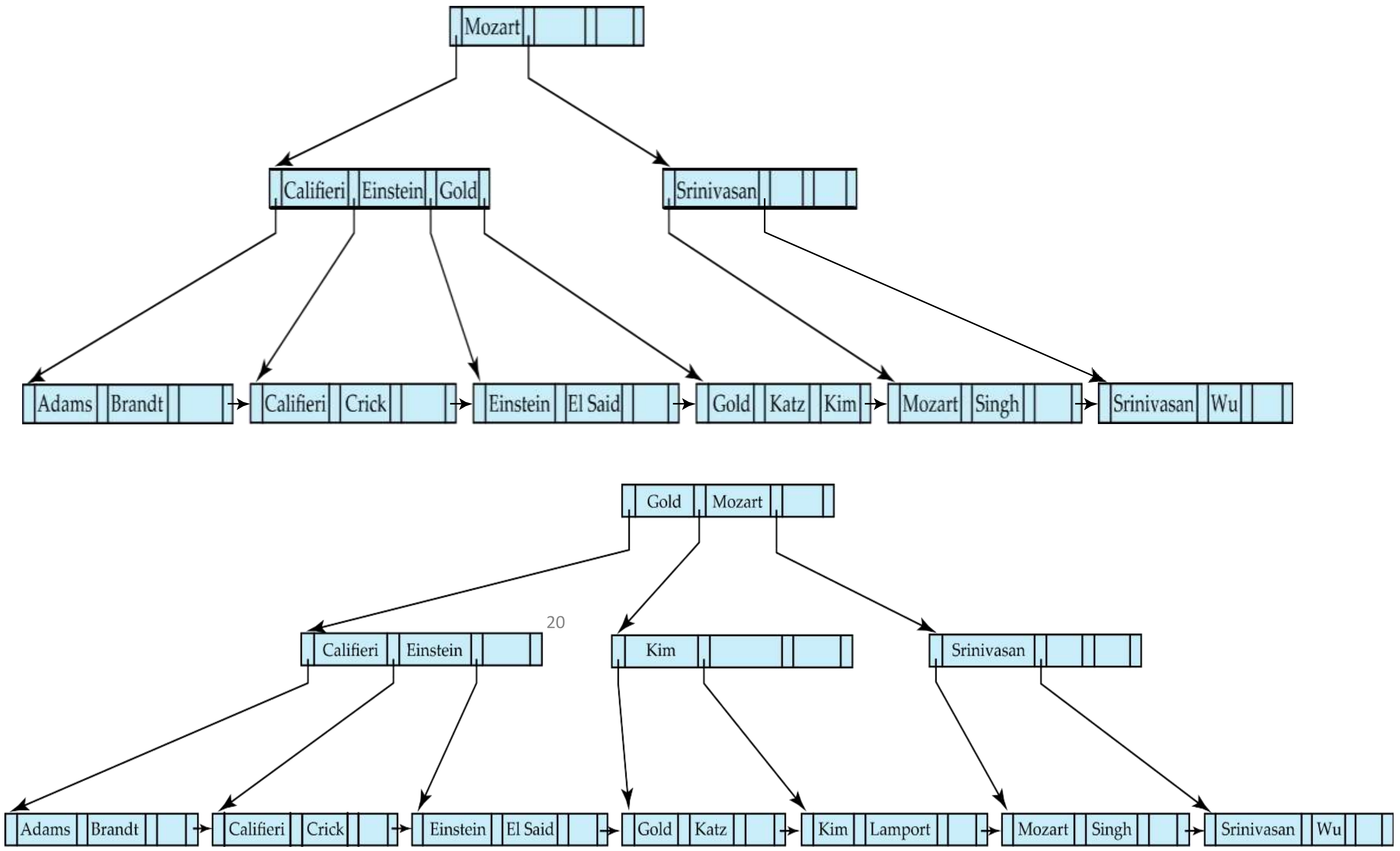
B+-Tree Insertion



19

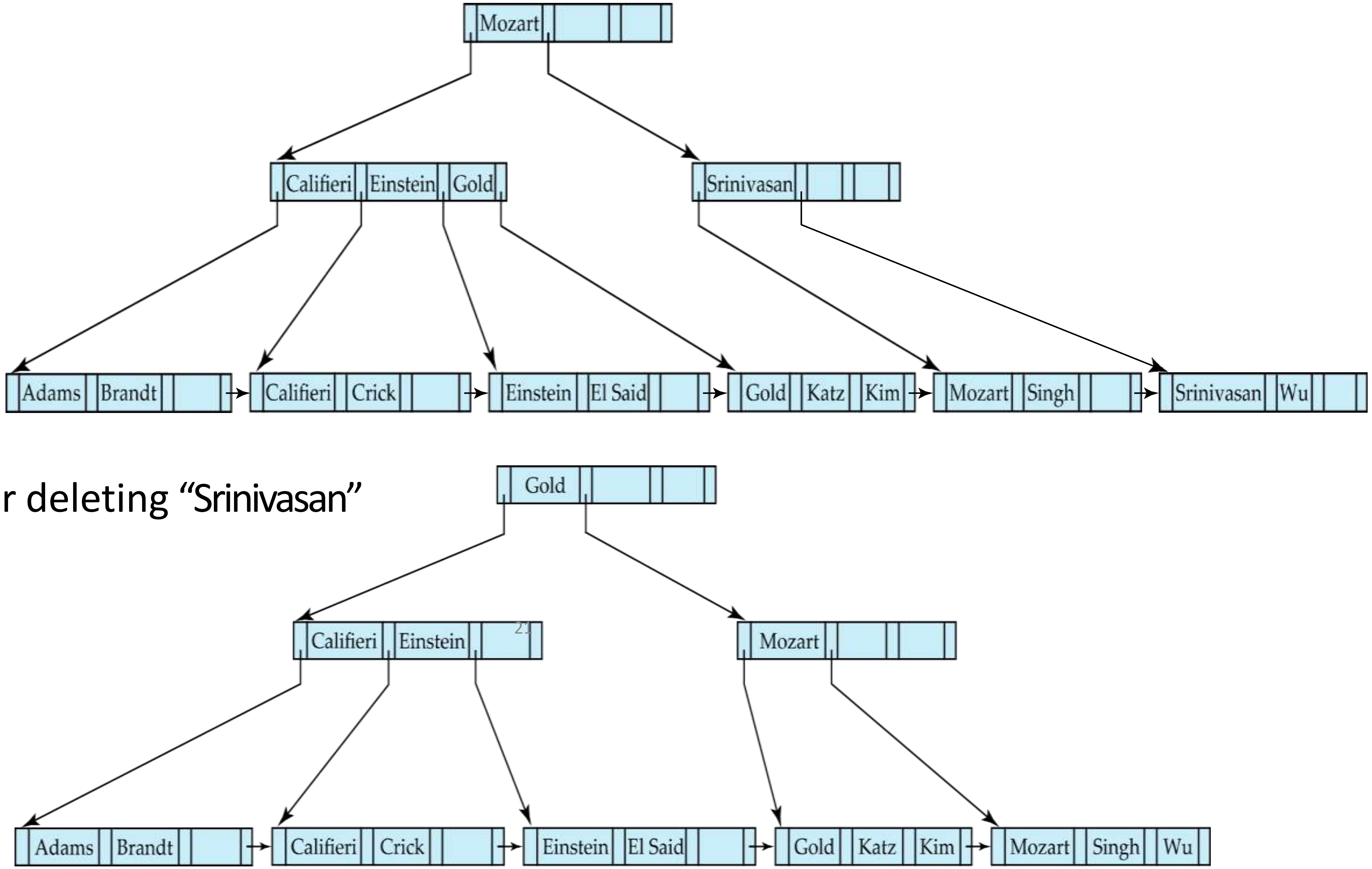
B+-Tree before and after insertion of "Adams"

B+-Tree Insertion



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

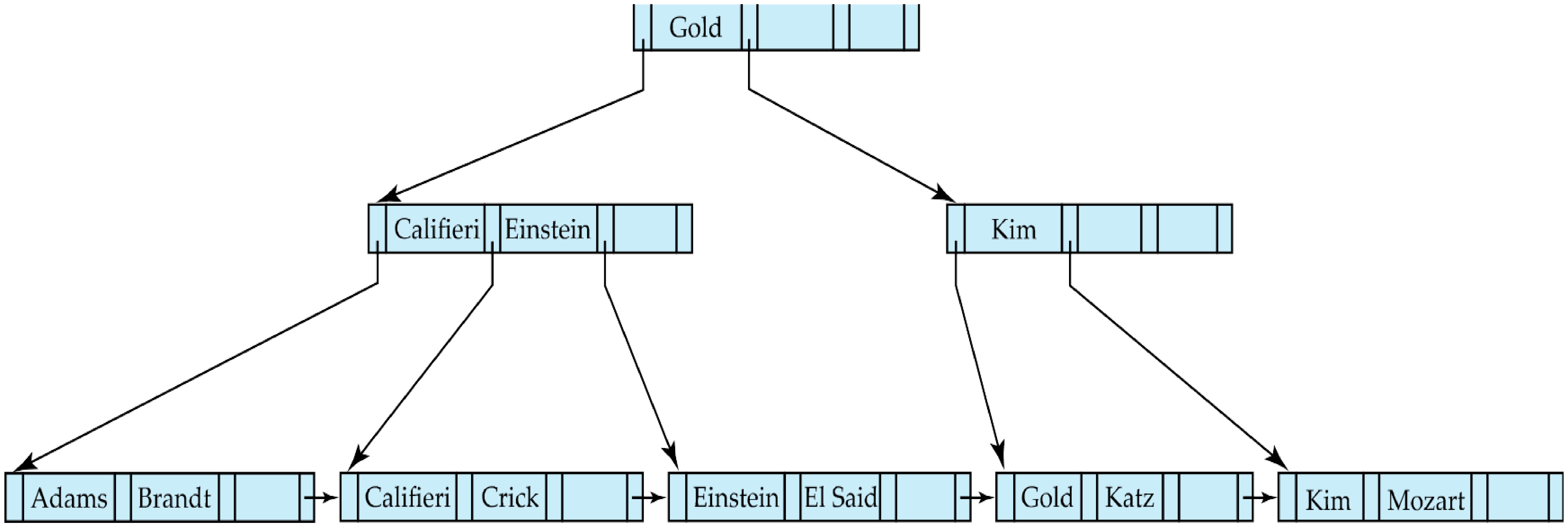
Examples of B+-Tree Deletion



Before and after deleting "Srinivasan"

- Deleting "Srinivasan" causes merging of under-full leaves

Examples of B+-Tree Deletion (Cont.)

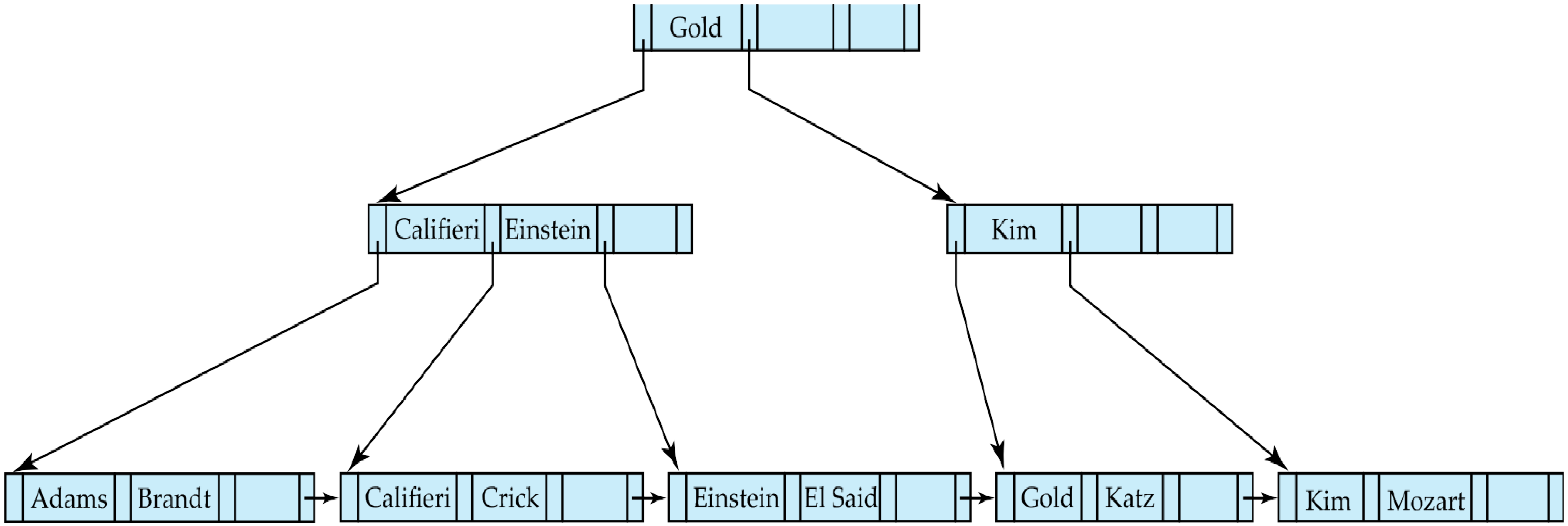


22

Deletion of "Singh" and "Wu" from result of previous example

- 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

Examples of B+-Tree Deletion (Cont.)



23

Deletion of “Singh” and “Wu” from result of previous example

- 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

Non-Unique Search Keys

- Alternatives to scheme described earlier
  - Buckets on separate block (bad idea)
  - List of tuple pointers with each key
    - Extra code to handle long lists
    - Deletion of a tuple can be expensive if there are many duplicates on search key (why?)
    - Low space overhead, no extra cost for queries
  - Make search key unique by adding a record-identifier
    - Extra storage overhead for keys
    - Simpler code for insertion/deletion
    - Widely used

Reference

1. https://www.tutorialspoint.com/dbms/dbms_file_structure.htm#:~:text=Relative%20data%20and%20information%20is,blocks%20that%20can%20store%20records.
2. https://www.javatpoint.com/dbms-file-organization
3. https://www.tutorialspoint.com/dbms/dbms_storage_system.htm

THANK YOU

26