CMSC 461, Database Management Systems
Spring 2018

Lecture 11 – Indexing and Hashing Part 2

These slides are based on "Database System Concepts" 6th edition book (whereas some quotes and figures are used from the book) and are a modified version of the slides which accompany the book (http://codex.cs.yale.edu/avi/db-book/db6/slide-dir/index.html), in addition to the 2009/2012 CMSC 461 slides by Dr. Kalpakis

Dr. Jennifer Sleeman

https://www.csee.umbc.edu/~jsleem1/courses/461/spr18

Logistics

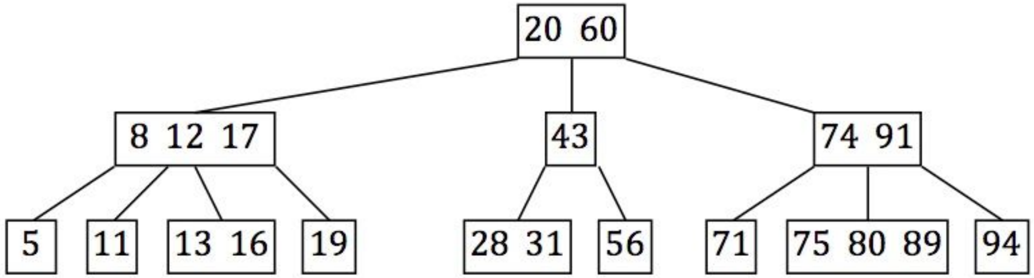
- Project Phase 2 due

B-Trees and B^+-Trees

First:
B-Tree and B^+-Tree Background

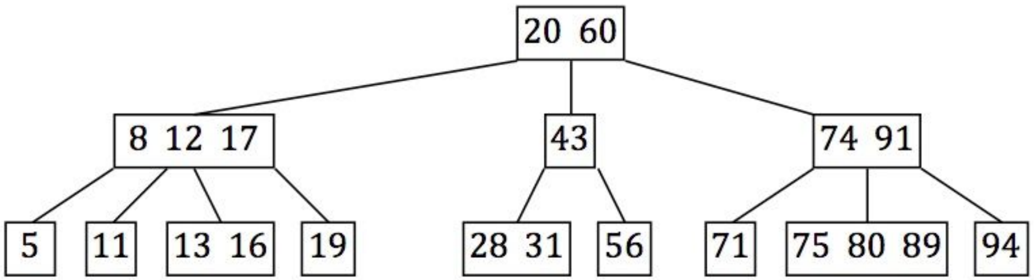
(A,B) Trees

- Each node has between *a* and *b* children
- Each node stores between *a-1* and *b-1* entries
- What is *a*?
- What is *b*?



B-Trees

- Generalization of a binary search tree
- Self-balancing
- Search, insert and delete O(log n)
- Optimized for reading/writing large blocks of data
- Type of (a,b) tree



B-Trees - Properties

A B-tree of order *n* has the following properties:

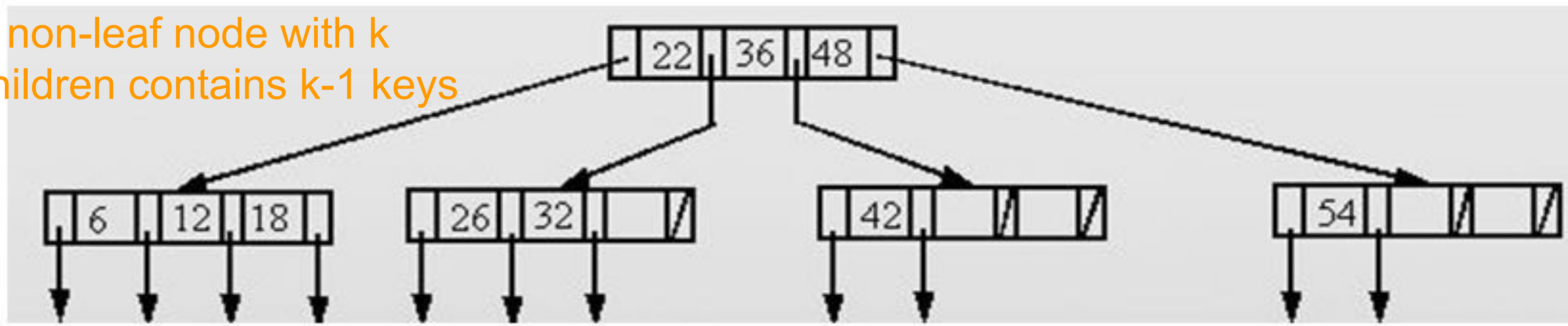
1. Every node has at most *n* children
2. A non-leaf node with *k* children contains *k-1* keys
3. Root is going to have at least two children if it is not a leaf node
4. Every non-leaf node except root has at least $\lceil n/2 \rceil$ children
5. All leaves on the same level

B-Trees - Properties

Root is going to have at least two children if it is not a leaf node

Example: A B-tree of order 4

A non-leaf node with k children contains k-1 keys



Every node has at most n children

All leaves on the same level

Every non-leaf node except root has at least $\lceil n/2 \rceil$ children

Image from
http://slideplayer.com/slide/5107822/

Extension: B+Trees

1. With a B+ tree:
   a. Internal nodes have no data
   b. Only the leaves have data
   c. Each internal node still has (up to) N-1 keys

Extension: B+Trees

- Order property:
  – subtree between two keys *x* and *y* contain leaves with values v such that $x \leq v < y$
- Leaf nodes have up to *L* sorted keys

B^+-Tree Index Files

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

B^+-Tree Index Files

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 $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n–1)/2 \rceil$ 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.

B+-Tree Node Structure

- Typical node

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	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

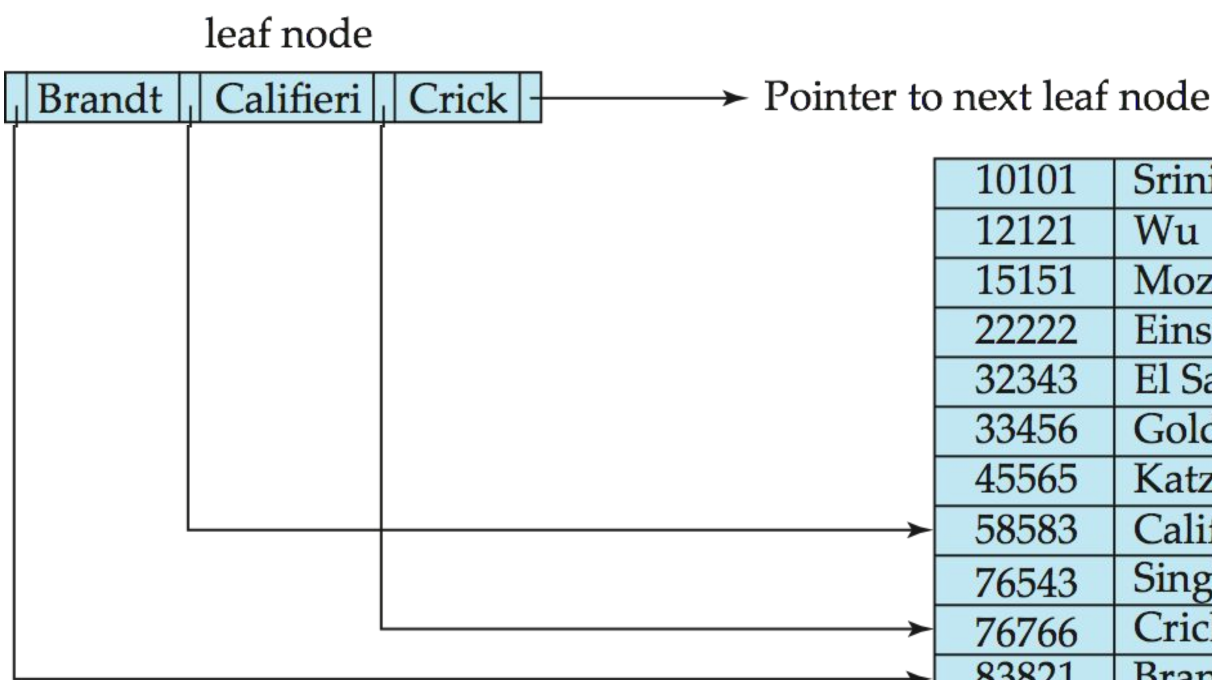
  $$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

  (Initially assume no duplicate keys, address duplicates later)

Based on and image from "Database System Concepts" book and slides, 6th edition

Leaf Nodes in B+-Trees

- Properties of a leaf node:
  - For i = 1, 2, . . ., 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

Based on and image from "Database System Concepts" book and slides, 6th edition

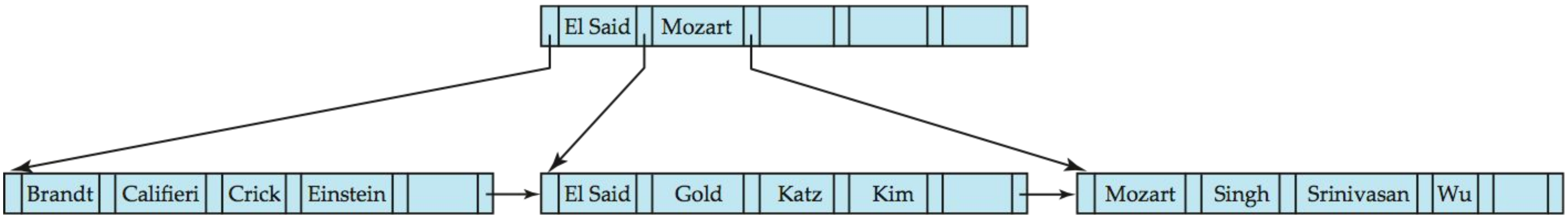
Non-Leaf Nodes in B$^+$-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with *m* pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
  - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}

P_1	K_1	P_2	$\cdots$	P_{n-1}	K_{n-1}	P_n

Based on and image from "Database System Concepts" book and slides, 6th edition

Example of B$^+$-tree

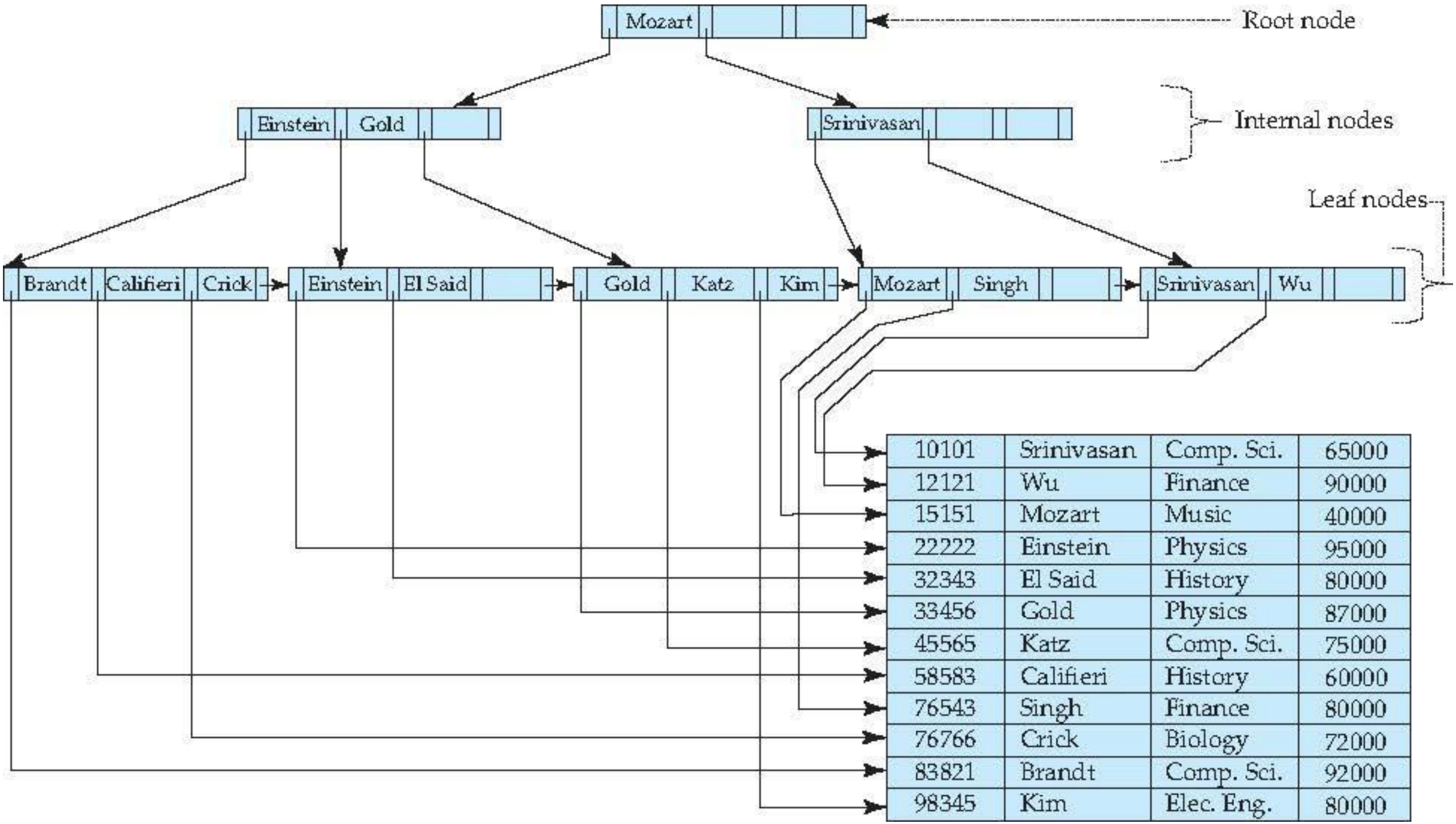


- B$^+$-tree for *instructor* file (n = 6)

- Leaf nodes must have between 3 and 5 values ($\lceil(n-1)/2\rceil$ and $n-1$, with $n = 6$).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil(n/2)\rceil$ and n with n =6).
- Root must have at least 2 children.

Based on and image from "Database System Concepts" book and slides, 6th edition

Example of B^+-Tree



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

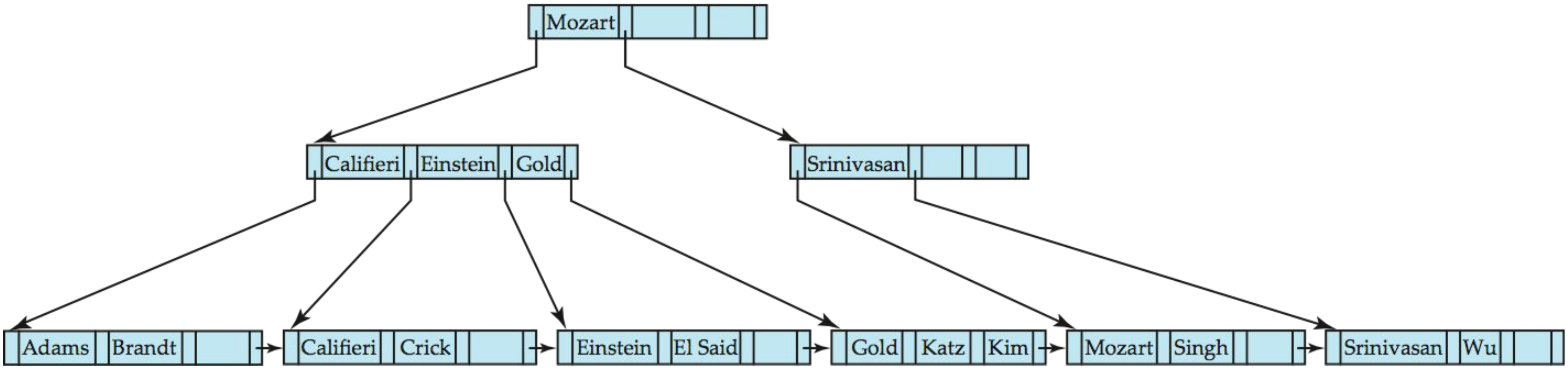
Observations about B^+-trees

- 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* $\lceil n/2 \rceil$ values
    - Next level has at least 2* $\lceil n/2 \rceil$ * $\lceil n/2 \rceil$ 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 (as we shall see).

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.



Based on and image from "Database System Concepts" book and slides, 6th edition

Queries on B^+-Trees

- 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

Queries on B^+-Trees

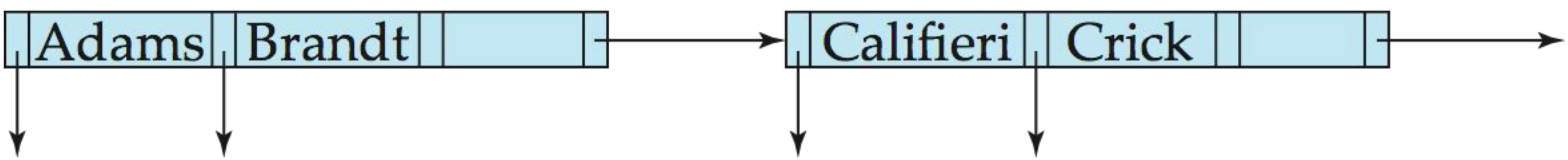
- With 1 million search key values and $n = 100$
  - at most $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

Updates on B^+-Trees: Insertion

1. Find the leaf node in which the search-key value would appear
2. If the search-key value is already present in the leaf node
   1. Add record to the file
   2. If necessary add a pointer
3. If the search-key value is not present, then
   1. Add the record to the main file
   2. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
   3. Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the next slide.

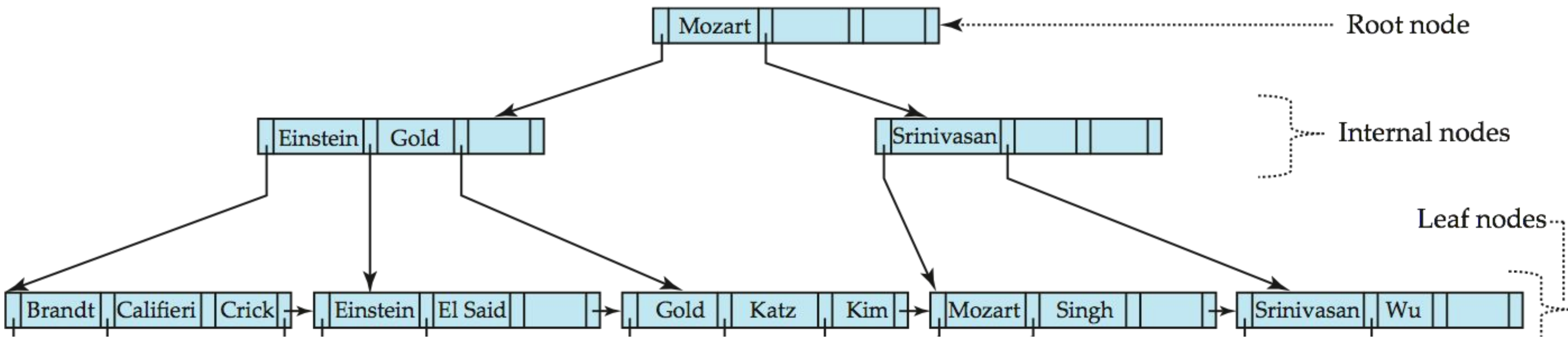
Updates on B^+-Trees: Insertion

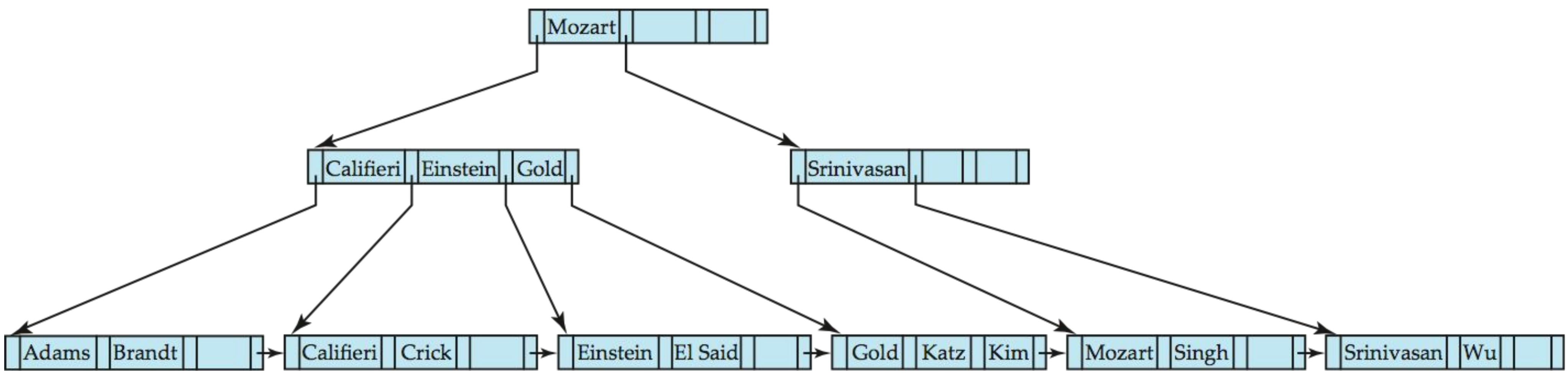
- Splitting a leaf node:
  - take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be *p,* and let *k* be the least key value in *p.* Insert (*k,p*) 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^+-Trees Insertion

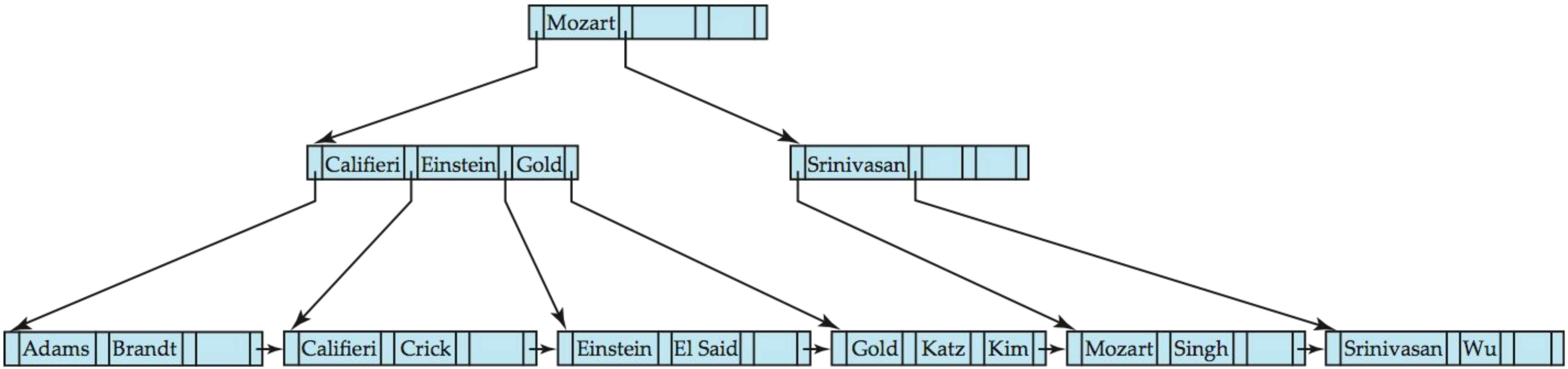


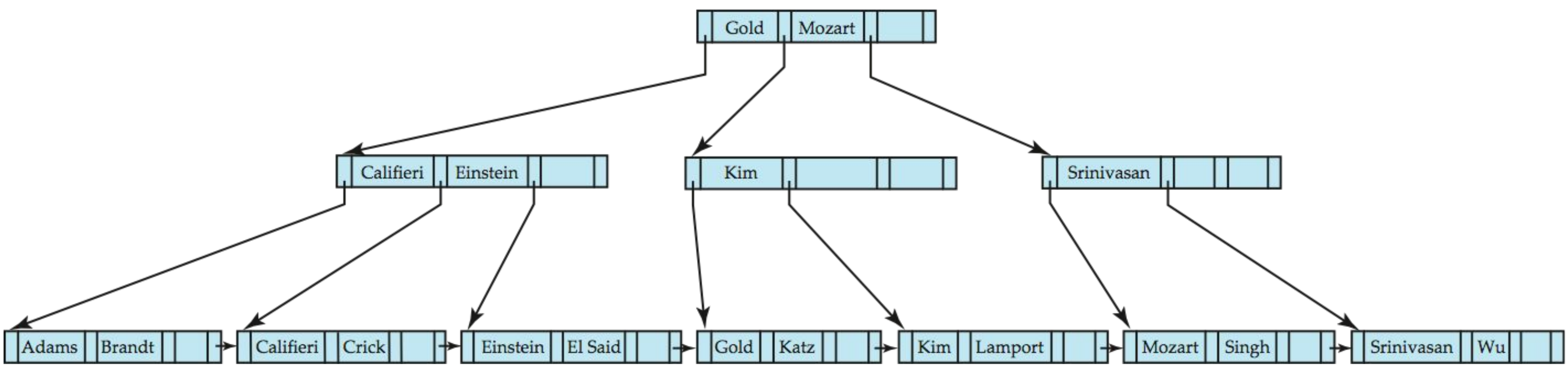


- B^+-Tree before and after insertion of “Adams”

Based on and image from “Database System Concepts” book and slides, 6th edition

B^+-Trees Insertion



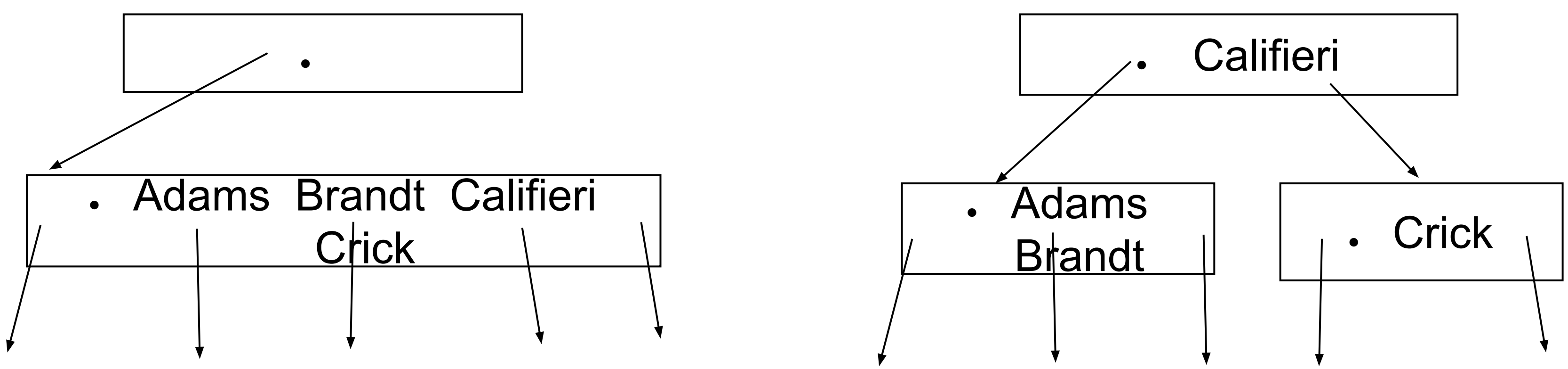


- B^+-Tree before and after insertion of “Lamport”

Based on and image from “Database System Concepts” book and slides, 6th edition

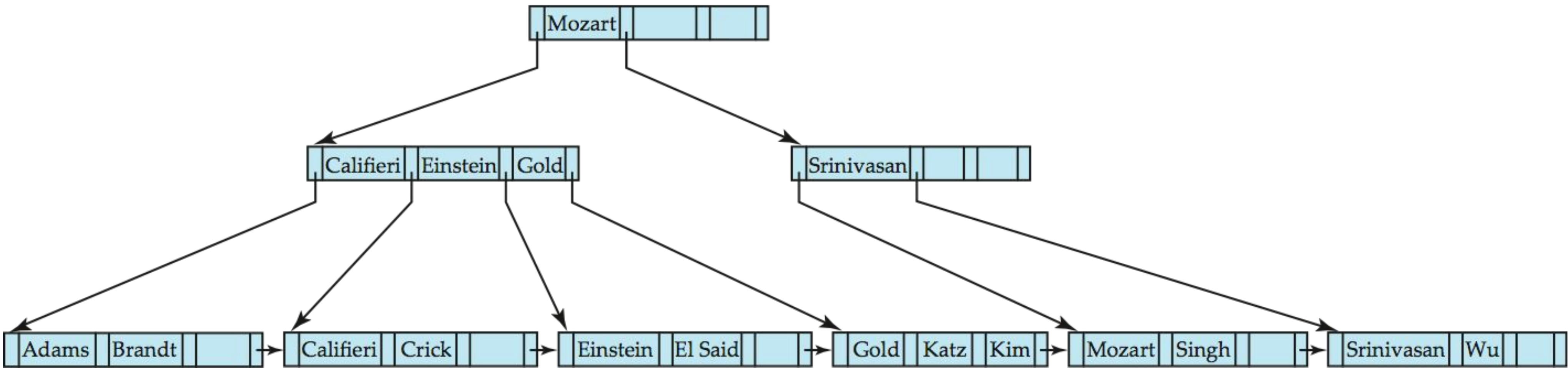
Insertion in B^+-Trees

- 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
- **Read pseudocode in book!**

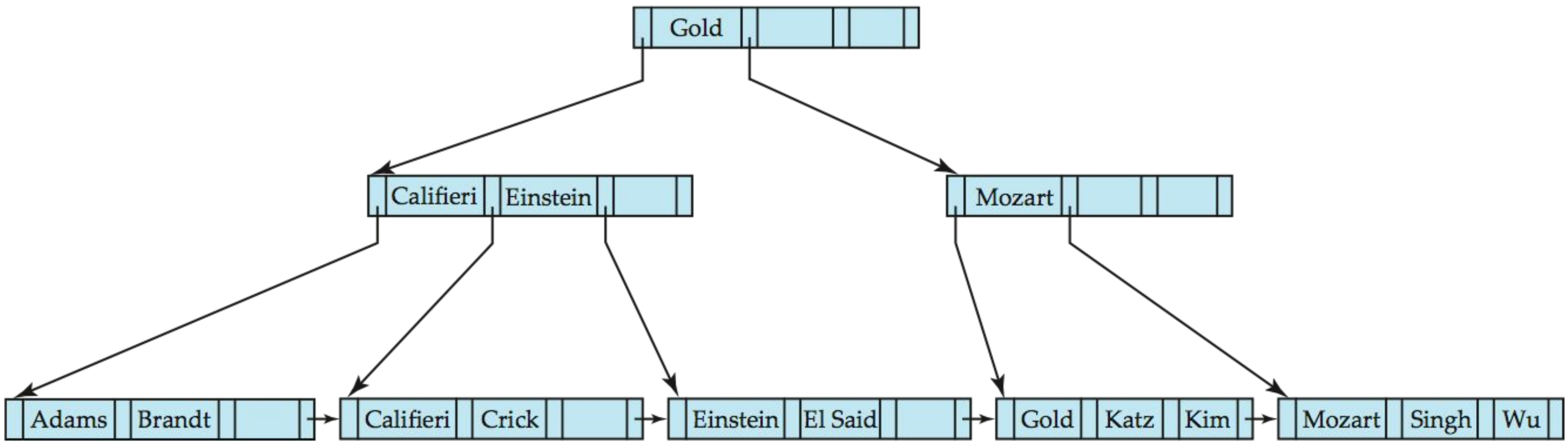


Based on and image from "Database System Concepts" book and slides, 6th edition

Example of B⁺-tree Deletion



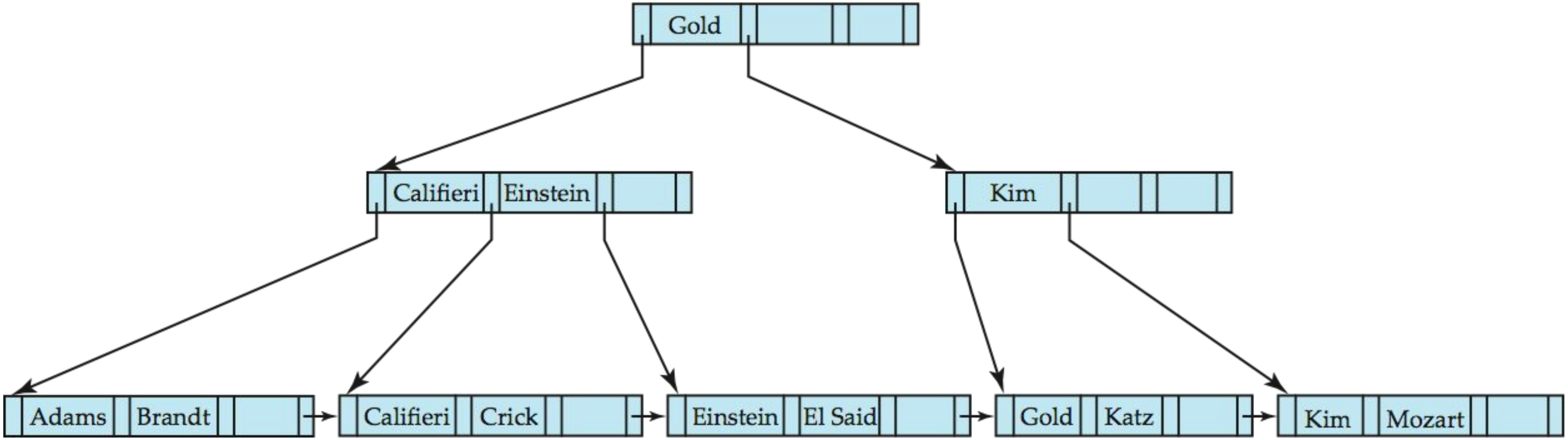
- Before and after deleting "Srinivasan"



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

Based on and image from "Database System Concepts" book and slides, 6th edition

Example of B$^+$-tree Deletion



- 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

Based on and image from "Database System Concepts" book and slides, 6th edition

Updates on B+-Trees: Deletion

- Find the record to be deleted, and remove it from the main file
- Remove (search-key value, pointer) from the leaf node

Updates on B+-Trees: Deletion

- 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:***
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - 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.

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.

Updates on B^+-Trees: Deletion

- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ 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.

B-Tree Index Files

- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Generalized B-tree leaf node

P_1	K_1	P_2	$\cdots$	P_{n-1}	K_{n-1}	P_n

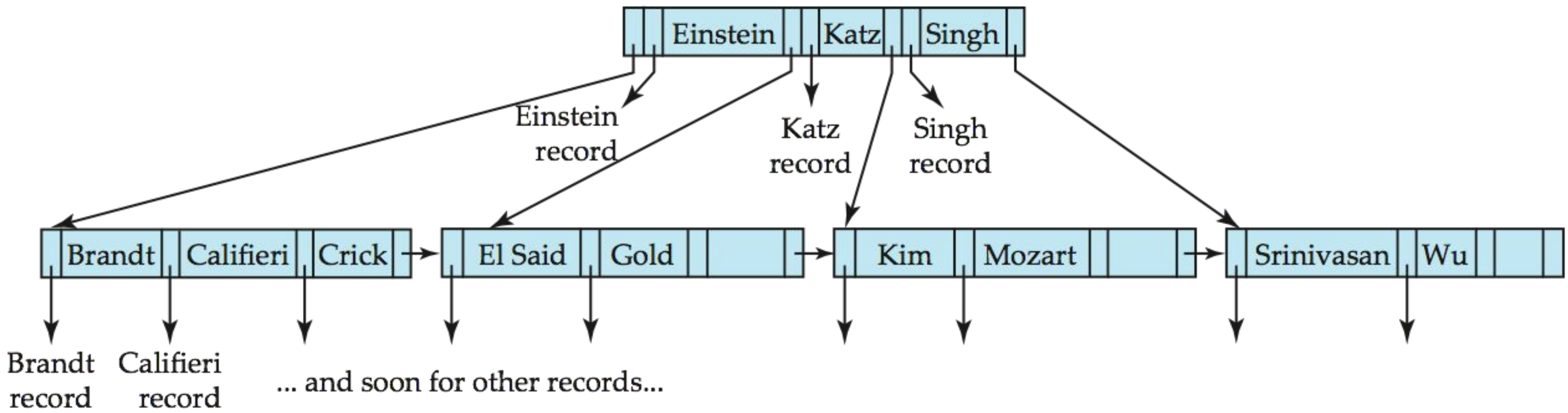
(a)

P_1	B_1	K_1	P_2	B_2	K_2	$\cdots$	P_{m-1}	B_{m-1}	K_{m-1}	P_m

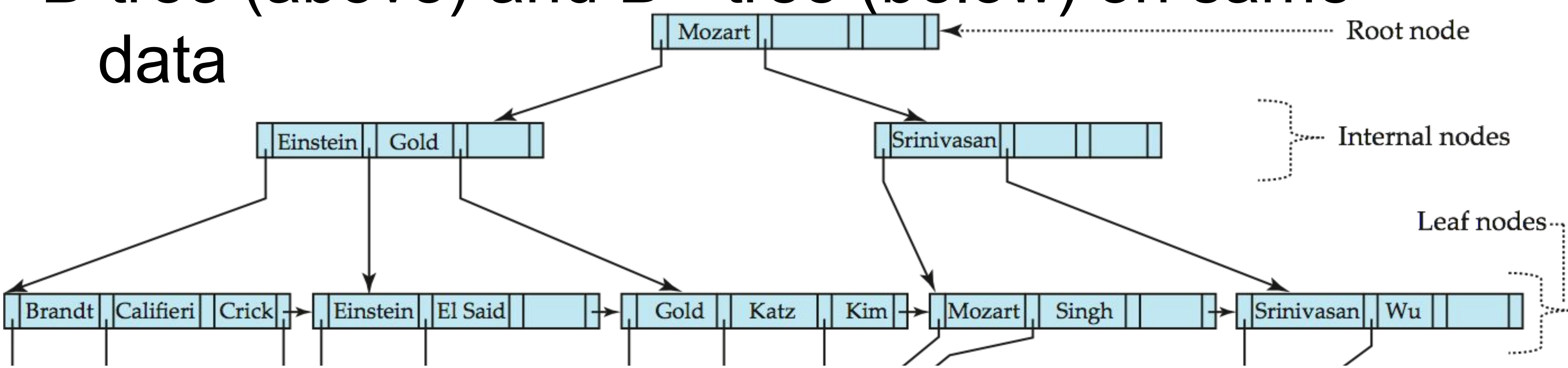
(b)

Non-leaf node – pointers Bi are the bucket or file record pointers.

B-Tree Index File Example



B-tree (above) and B+-tree (below) on same data



B-Tree Index Files

- Advantages of B-Tree indices:
  - May use less tree nodes than a corresponding B^+-Tree.
  - Sometimes possible to find search-key value before reaching leaf node.

B-Tree Index Files

- Disadvantages of B-Tree indices:
  - Only small fraction of all search-key values are found early
  - Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B^+-Tree
  - Insertion and deletion more complicated than in B^+-Trees
  - Implementation is harder than B^+-Trees.
- Typically, advantages of B-Trees do not out weigh disadvantages.