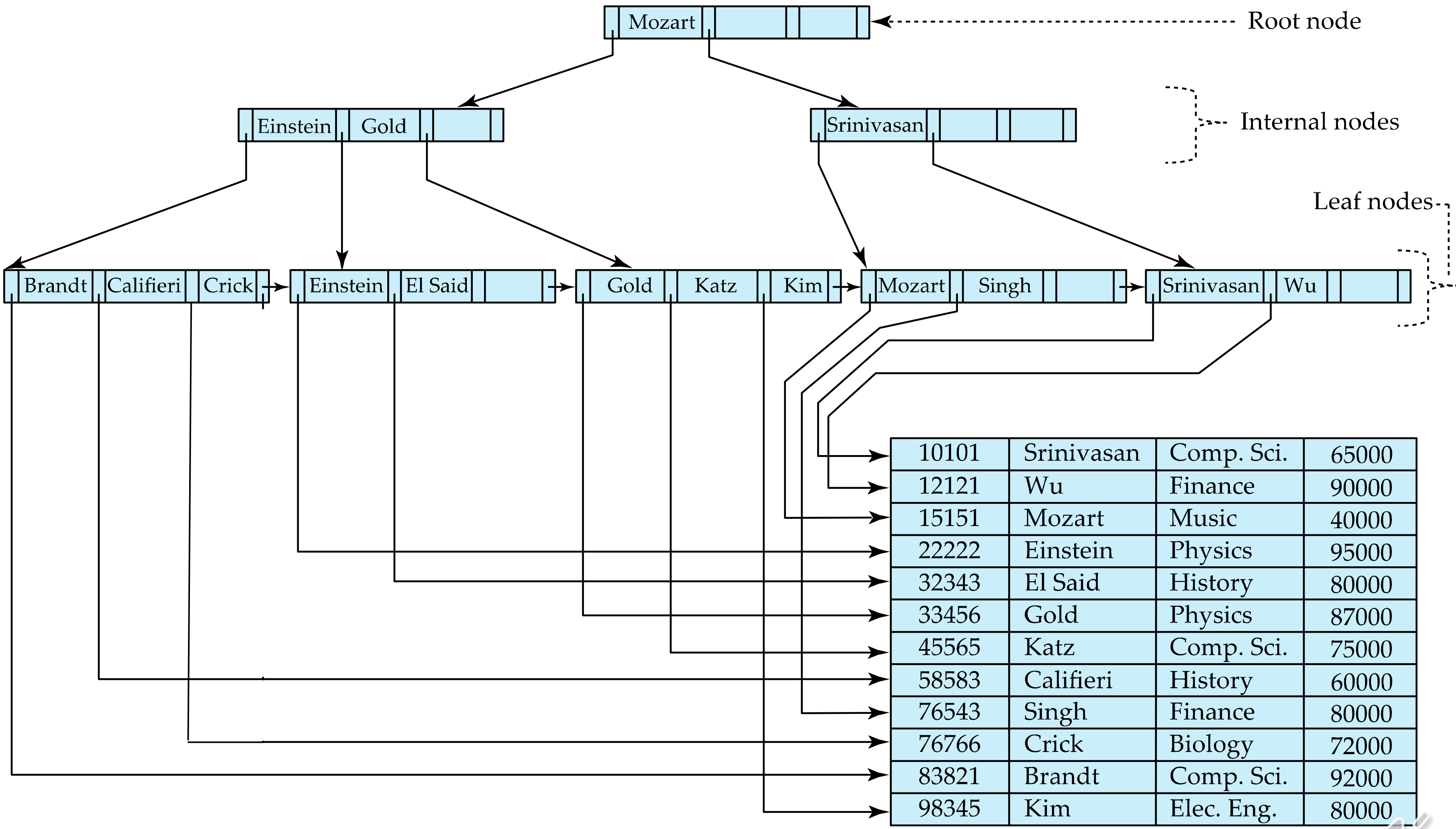
B^+-Tree Index Files

- Disadvantage of indexed-sequential files
  - Performance degrades as file grows for both
    - index lookups and
    - sequential scans of data

    since many overflow blocks get created for both.

    Assume a multi-level index for sequential files – more levels for large records, reading to many disk reads
  - Periodic reorganization of entire file is required.
- Advantage of B^+-tree files (more later):
  - 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 in commercial databases

Example of B+-Tree

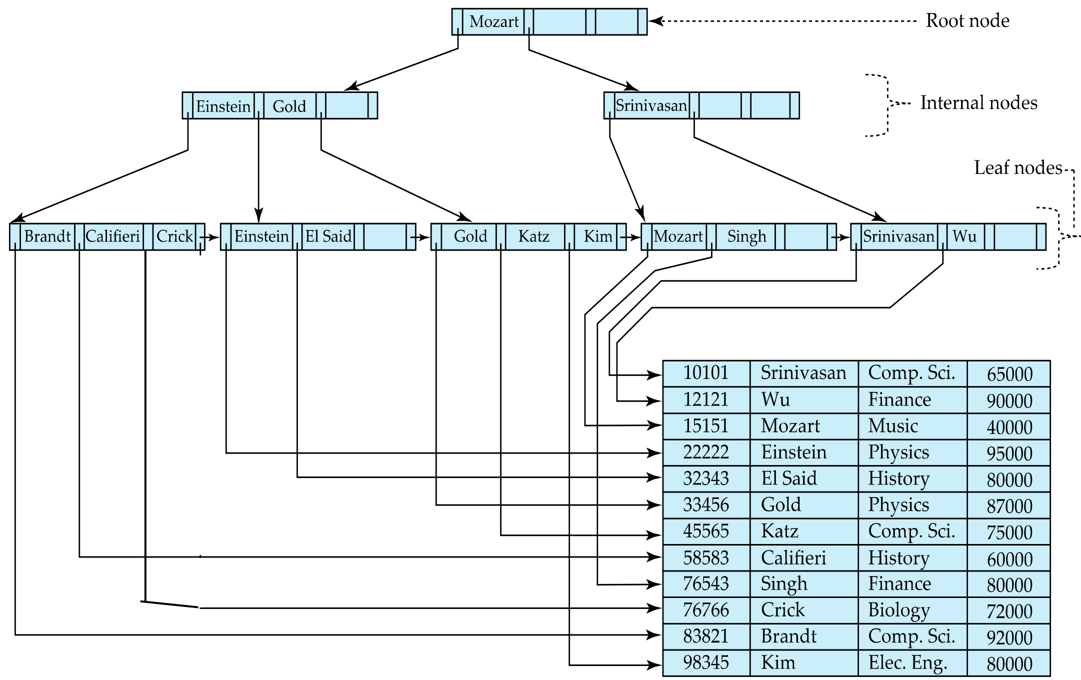


10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

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
- n is fixed for a particular tree (4 in the example, max values in leaf node + 1)
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n\text{–}1)/2 \rceil$ 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.



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

- Typical node

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

  - K_i are the search-key values
  - There are n-1 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

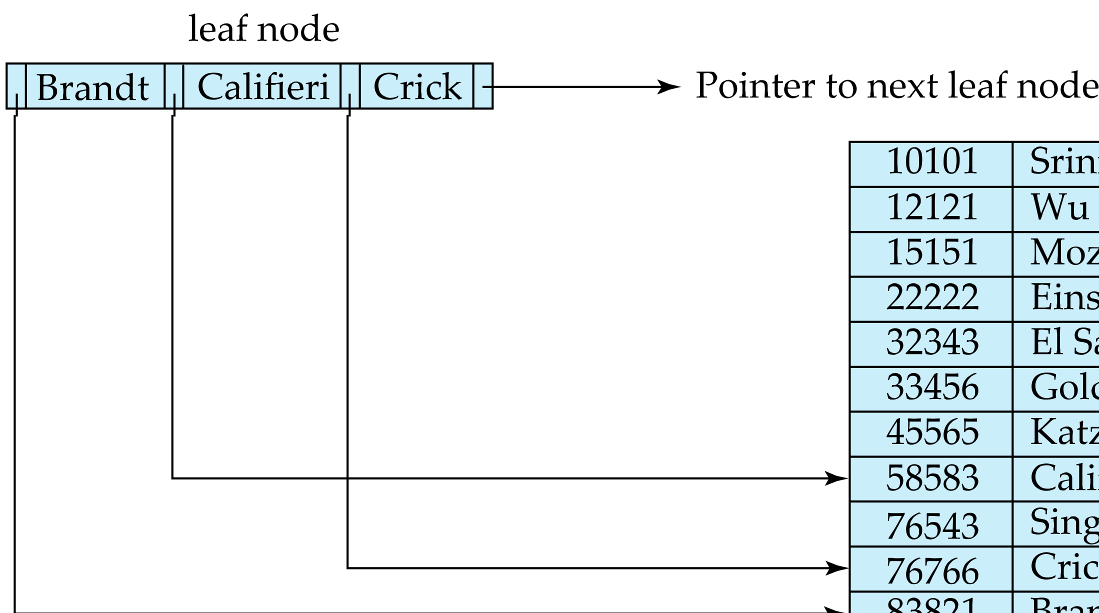
$$K_1 < K_2 < K_3 < . . . < K_{n-1}$$

(Initially assume no duplicate keys, address duplicates later)

Leaf Nodes in B⁺-Trees

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. Why? To help with range queries



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

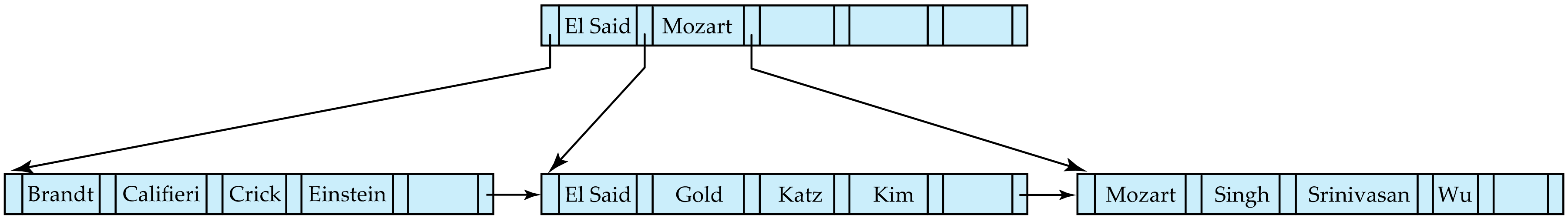
Non-Leaf Nodes in B+-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with *m* pointers (m <=n) :
  - 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}
  - General structure

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

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.

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 $\lceil n/2 \rceil$ nodes (approximation: not 2)
  - Next level has at least $\lceil n/2 \rceil * \lceil n/2 \rceil$ nodes
  - .. etc.
  - Level p has at least $\lceil n/2 \rceil^{p}$ nodes
  - Each leaf node has a minimum of $\lceil (n-1)/2 \rceil$ keys
  - If there are N records (N search key values) in a file, N = $\lceil n/2 \rceil^{p}$ X $\lceil (n-1)/2 \rceil$, assuming p is the path length of the tree
  - Assuming n is much larger than 1, $\lceil (n-1)/2 \rceil \sim \lceil n/2 \rceil$
  - Therefore N = $\lceil n/2 \rceil^{p+1}$

$$\log_{\lceil n/2 \rceil} N = p + 1$$

The number of nodes to be accessed (height) = p+1 as the root needs to be included

Observations about B$^+$-trees

- $$\log_{\lceil n/2 \rceil} N = p + 1$$

  The number of nodes to be accessed (height) = p+1 as the root needs to be included

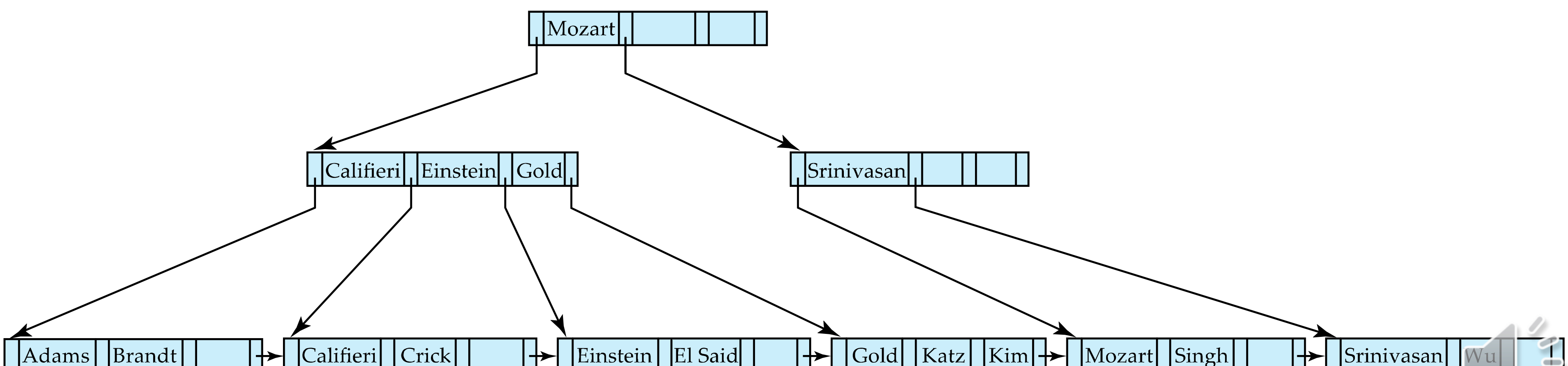
  h <= $\lceil \log_{\lceil n/2 \rceil}(N) \rceil$

- 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

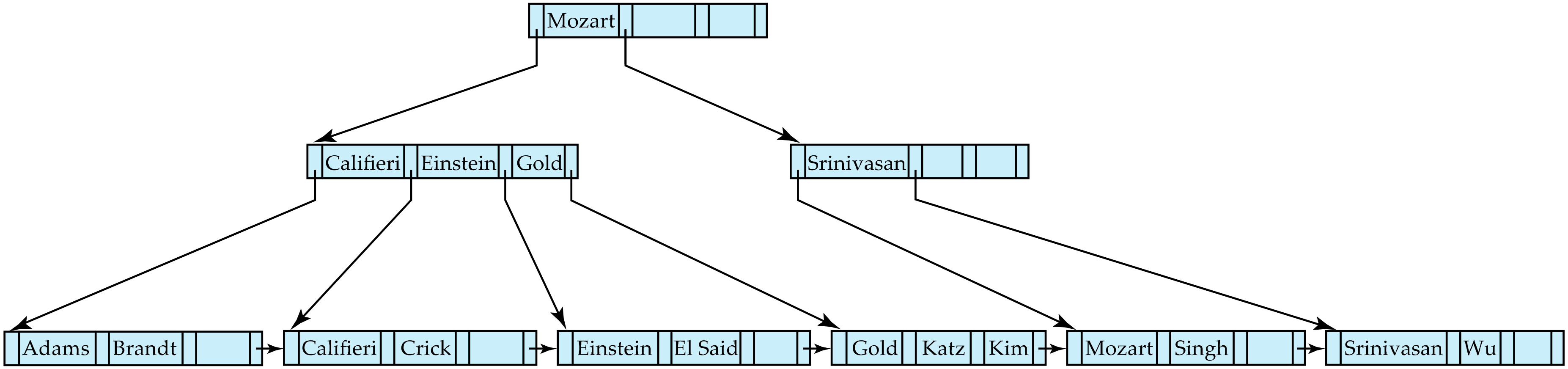
function *find*(*v*)

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. */



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



Observations about B+-trees

$$h \le \lceil \log_{\lceil n/2 \rceil}(N) \rceil$$

- Typically, the node size is chosen to be the same as the size of a disk block, which is typically 4 kilobytes
- Assume that
  - Search key size = 32 bytes
  - Disk pointer size = 8 bytes
  - $n * 8 + (n - 1)32 \le 4 * 2^{10}$
  - $40n - 32 \le 1024 * 4 = 4096$
  - $40n \le 4128$
  - $n \le 103$
  - Assume n = 100 and there are 1 million records to be stored. That is, .N=1000000
  - Therefore the number of B+ tree nodes to be accessed to retrieve a record <= $\lceil \log_{\lceil 100/2 \rceil}(1000000) \rceil = 4$

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

- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
- That is **<=** $\lceil \log_2(1000000) \rceil = 20$
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds
  - Using a B+ tree, it takes only 4 disk reads to access a record among 1 million records, in the worst case!

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, $-\infty$) to (v, $+\infty$)
- 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 – recap.

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

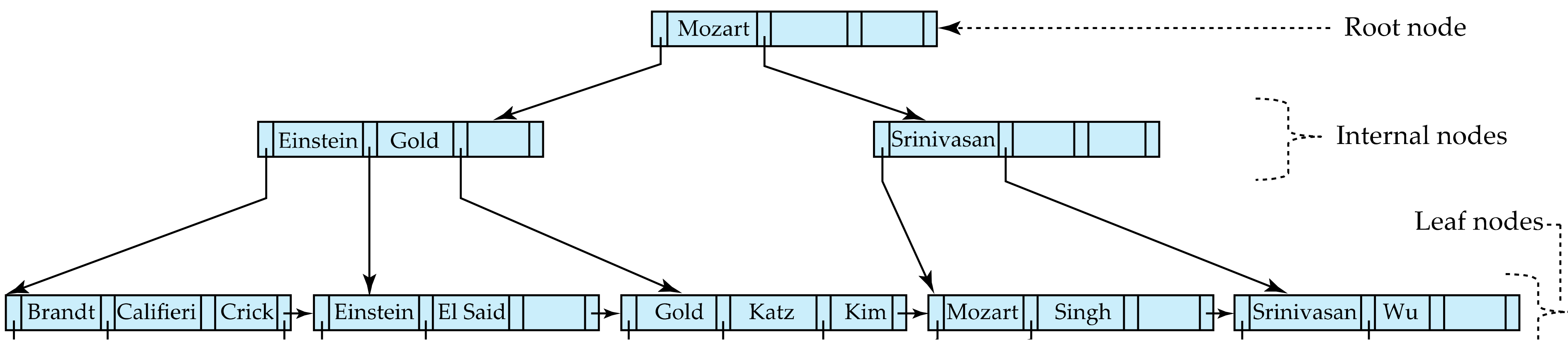
- All paths from root to leaf are of the same length
- *n* is fixed for a particular tree (4 in the example, max values in leaf node + 1)
- **Each node that is not a root or a leaf – an internal node - has between $\lceil n/2 \rceil$ and *n* children.**
- **A leaf node has between $\lceil (n–1)/2 \rceil$ 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.**

Updates on B$^+$-Trees: Insertion

Assume the record is already added to the file. Let

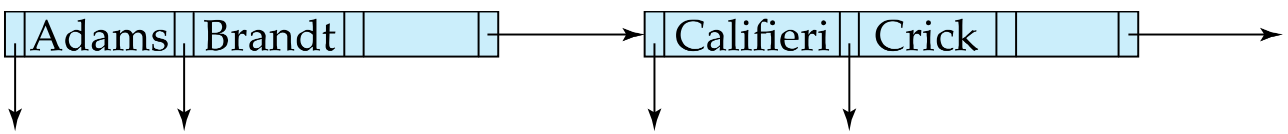
- *pr* be pointer to the record, and let
- v be the search key value of the record

1. Find the leaf node in which the search-key value would appear
   1. If there is room in the leaf node, insert (v, *pr*) pair in the leaf node
   2. Otherwise, split the node (along with the new (*v, pr*) entry) as discussed in the next slide, and propagate updates to parent nodes.



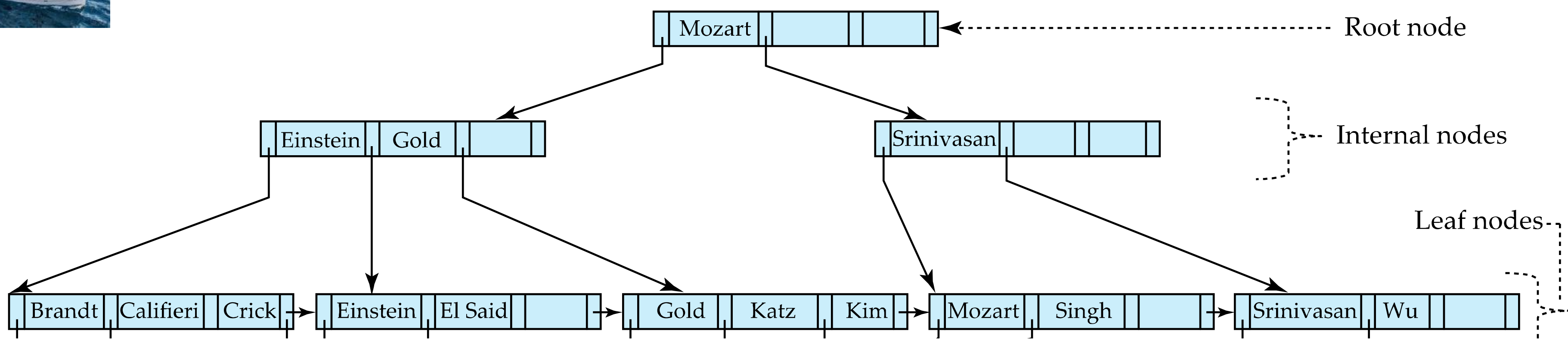
Updates on B+-Trees: Insertion (Cont.)

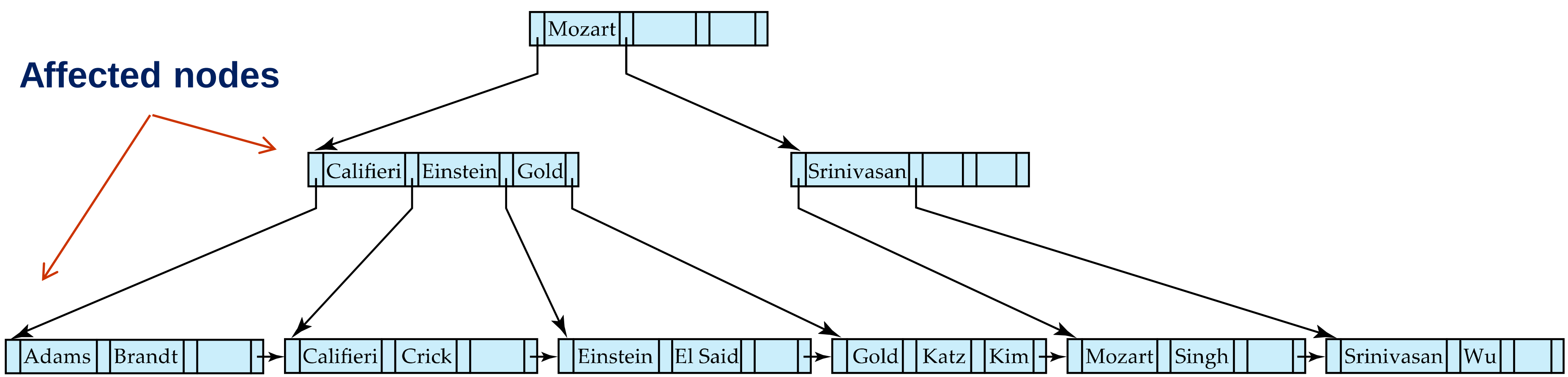
- Splitting a leaf node:
  - take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be *p,* and let *k* be the least key value in *p.* Insert (*k,p*)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



Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
Next step: insert entry with (Califieri, pointer-to-new-node) into parent

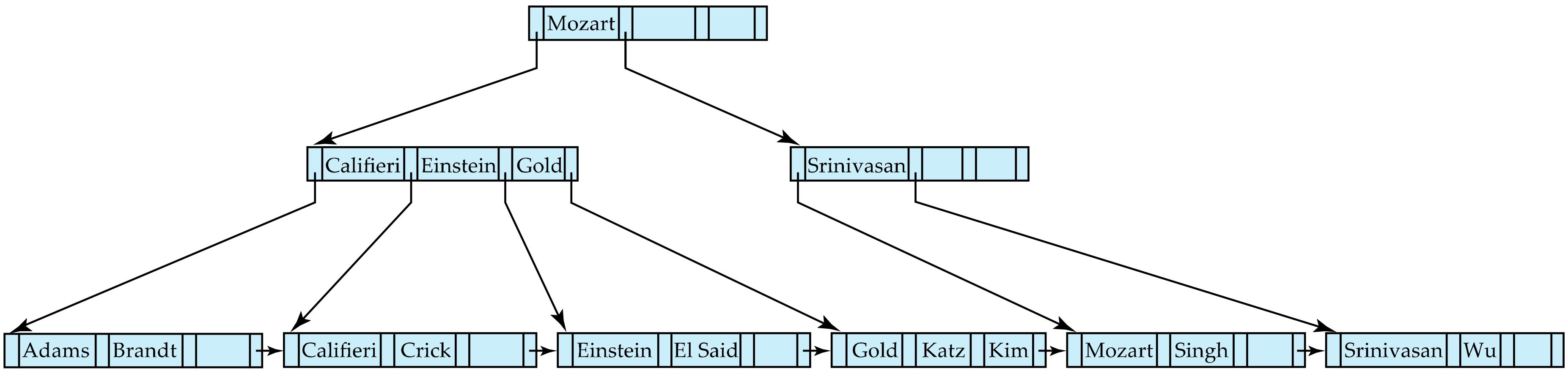
B+-Tree Insertion





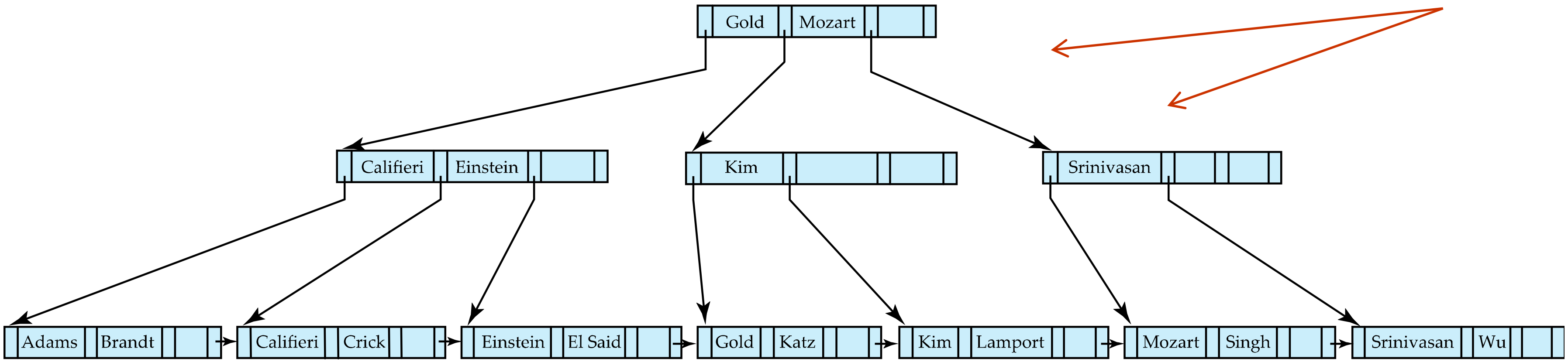
B+-Tree before and after insertion of "Adams"

B$^+$-Tree Insertion



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

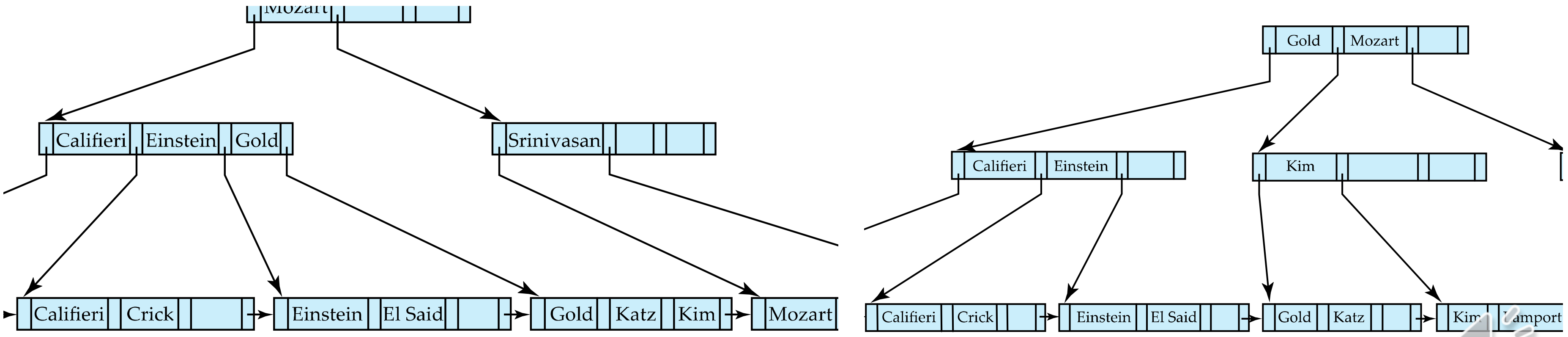
Affected nodes



Affected nodes

Insertion in B+-Trees (Cont.)

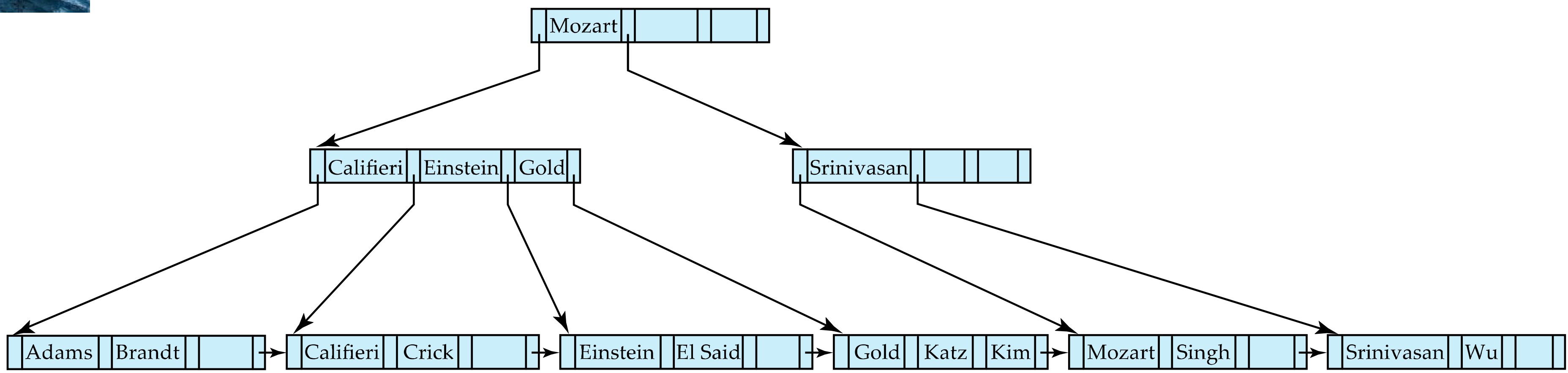
- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
  - Copy N to an in-memory area M with space for n+1 pointers and n keys
  - Insert (k,p) into M
  - Copy $P_1, K_1, \ldots, K_{\lceil (n+1)/2 \rceil - 1}, P_{\lceil (n+1)/2 \rceil}$ from M back into node N
  - Copy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



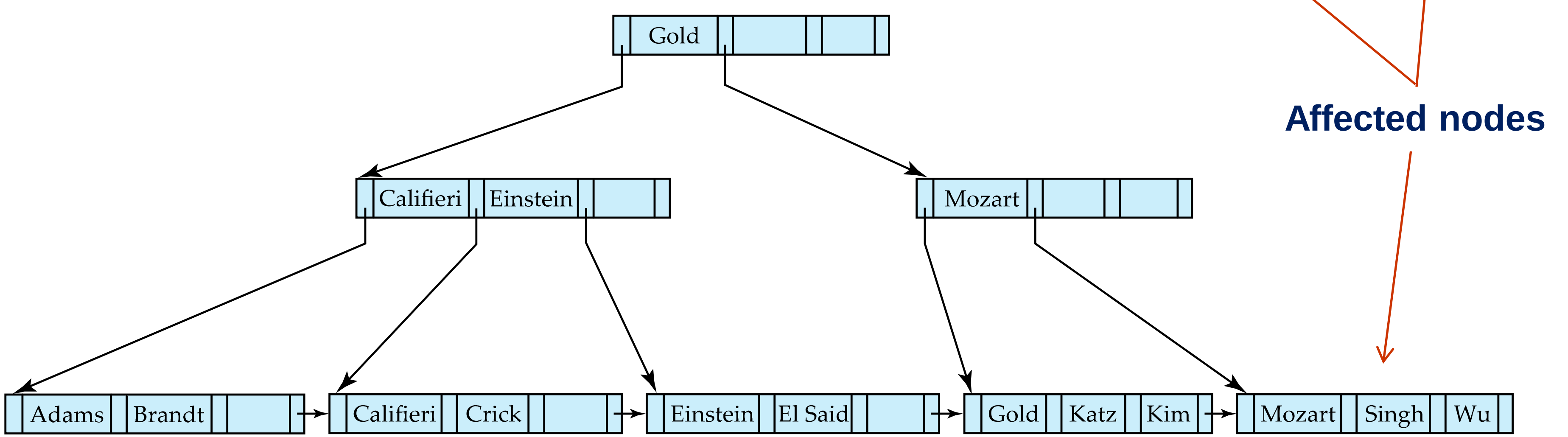
Insertion in B⁺-Trees (Cont.)

- **Read the pseudocode in the book to understand all the cases**
- Visualization:
  - https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

Examples of B⁺-Tree Deletion



Before and after deleting "Srinivasan"



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

Updates on B$^+$-Trees: Deletion

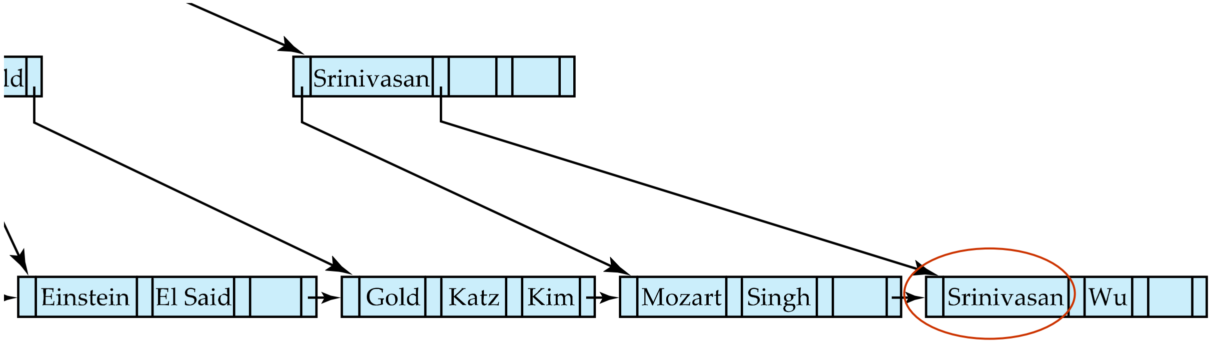
Assume that a record is already deleted from a file. Let *V* be the search key value of the record, and *Pr* be the pointer to the record.

- 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***:
  - 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.

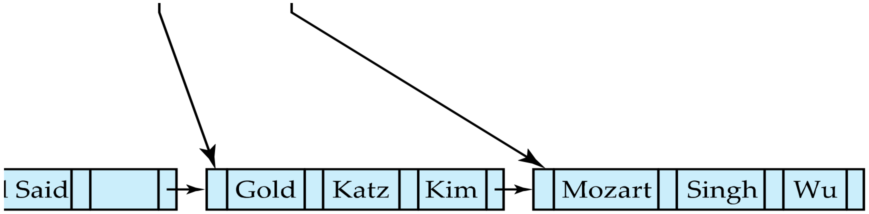
Updates on B+-Trees: Deletion

- 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.
- 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.

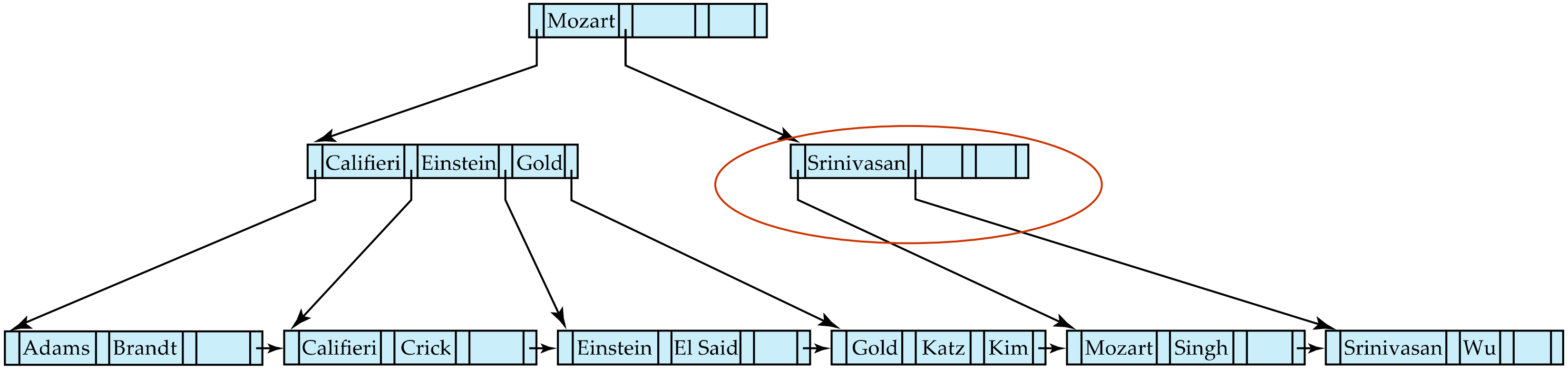
Deleting “Srinivasan”



1. Delete (Srinivasan, pointer to Srinivasan) from leaf node N
2. N has too few values/pointers
3. N’= sibling node of Mozart, K’=Srinivas of parent(N)
4. Entries in N and N’ can fit in a single node
5. N is NOT a predecessor of N’
6. N is a leaf node. Append all (key,pointer) pairs in N to N’. Set N’.P_n= N.P_n

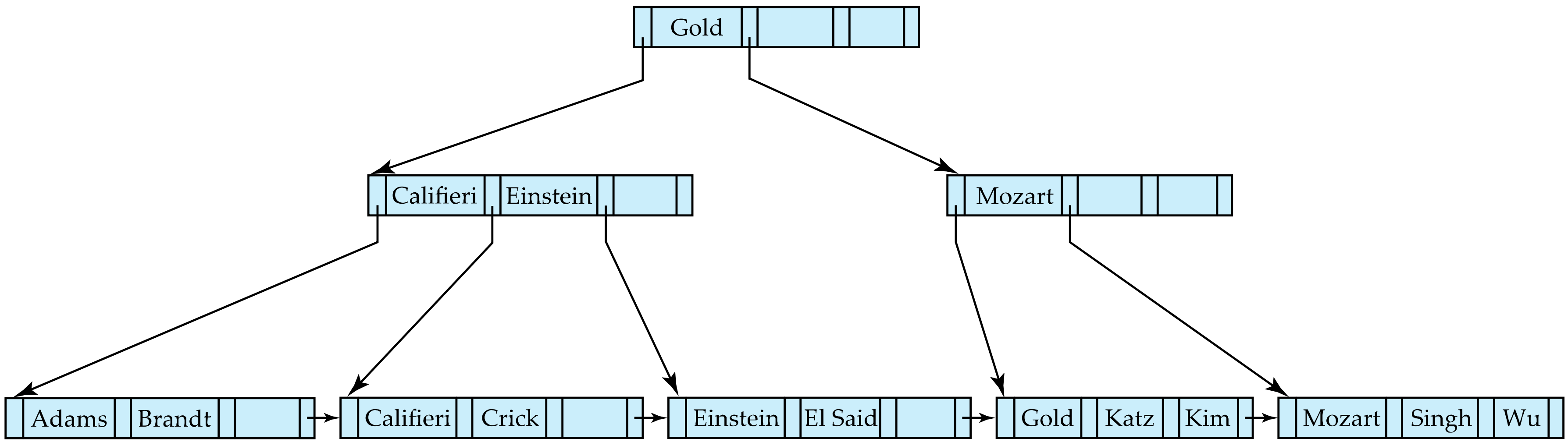


Deleting “Srinivasan” cont.

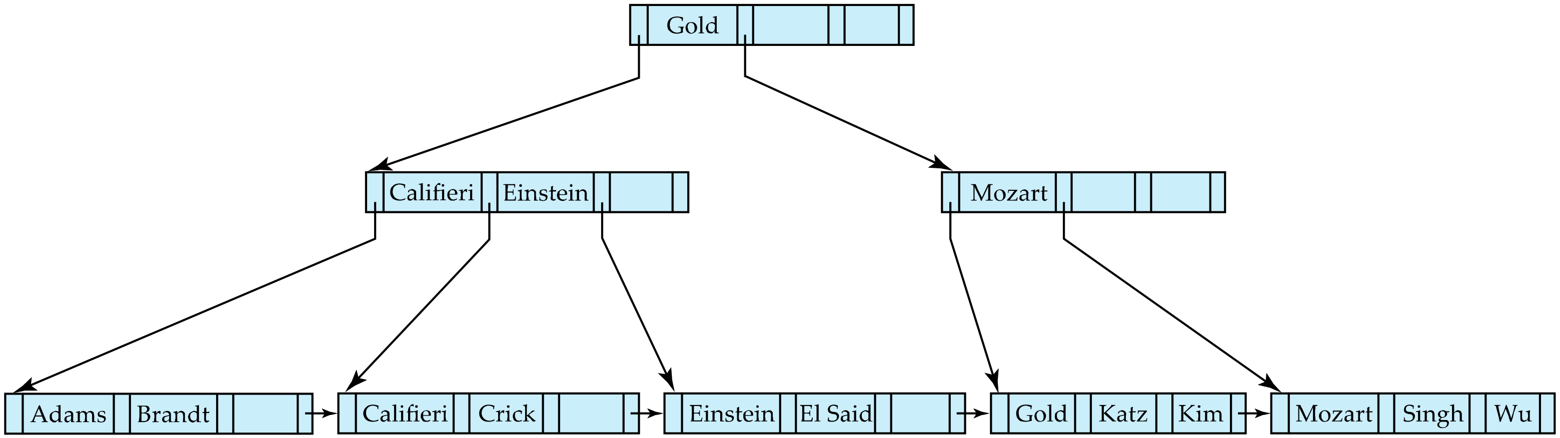


7. Delete entry (parent(N), K’,N) /* N is the leaf node with Srinivasan as one value */
   - Delete Srinivasan of parent. Now N is the parent node
   - N has only one pointer now – the one pointing to Mozart. N has too few children
   - Let N’= sibling node of Califeri and K’= Mozart. Since N and N’ cannot be merged, redistribute
   - N’ is a predecessor sibling. N is a non-leaf node
   - Let N’. P_m= that last pointer in N’, that is, pointer after Gold
   - Remove (Gold, pointer after Gold) from N’
   - *Insert pointer after Gold to N. But now N has only two pointers and no value between them.*
   - Therefore insert (N’. P_m, K’), that is, (pointer after Gold, Mozart) as the 1st pointer and value in N
   - Replace Mozart in parent(N) by N’.K_{m-1}=Gold
8. Delete leaf node of “Srinivasan”

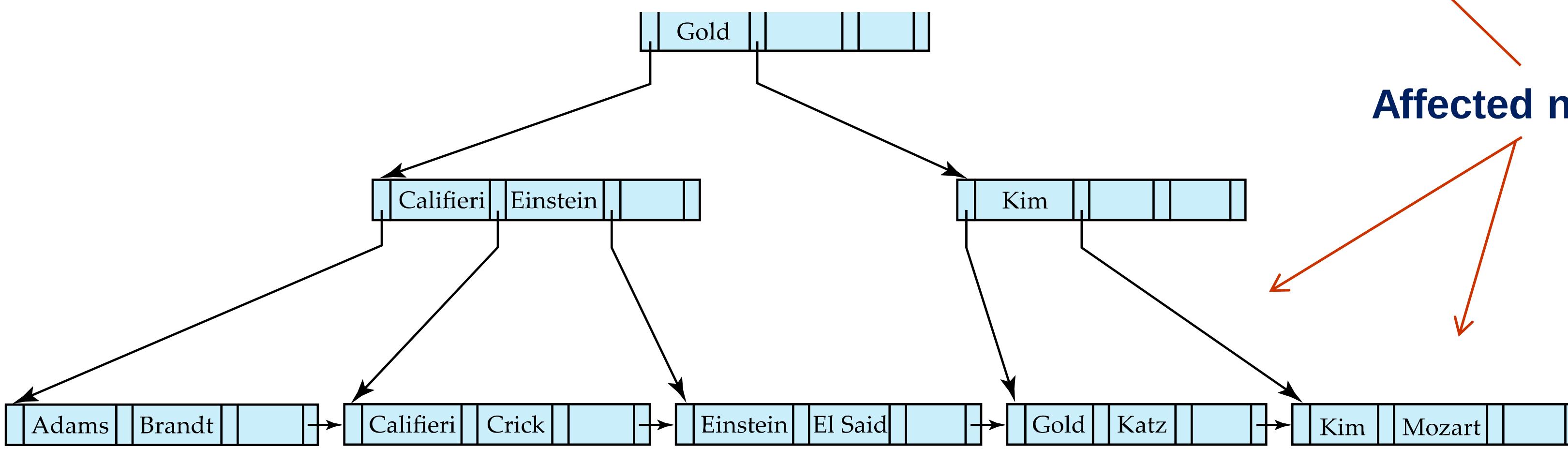
Deleting “Srinivasan” cont.



Examples of B^{+}-Tree Deletion (Cont.)



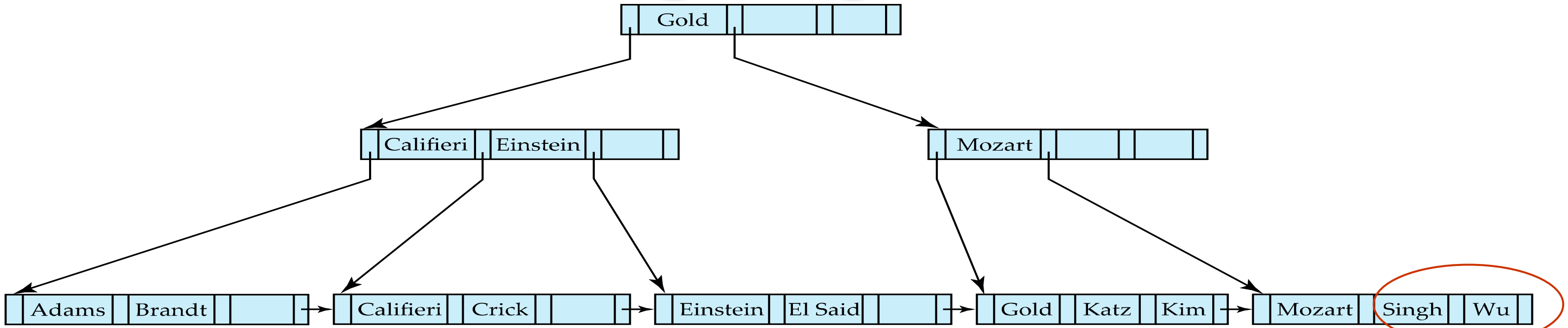
Before and after deleting "Singh" and "Wu"



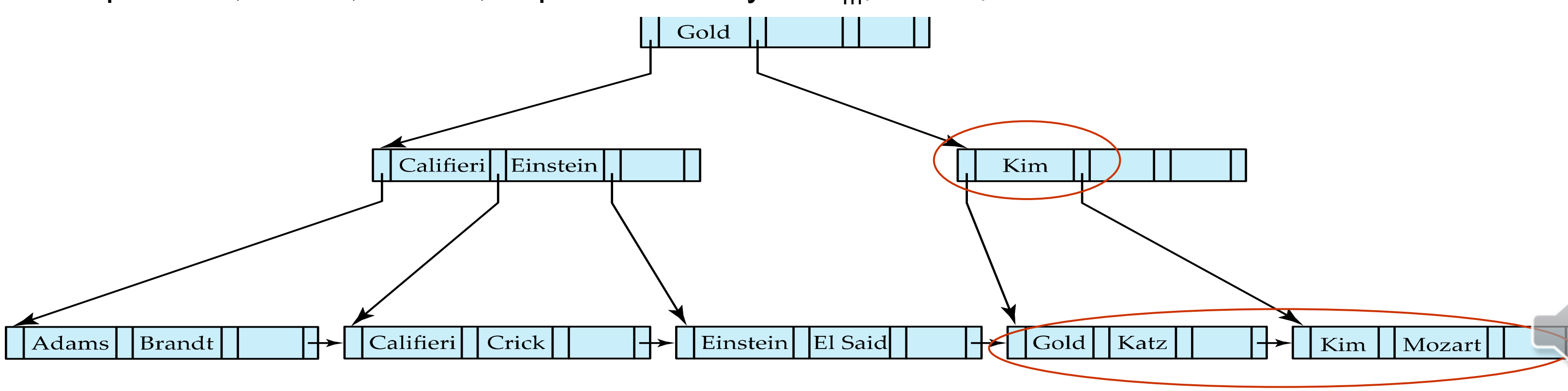
Affected nodes

- 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

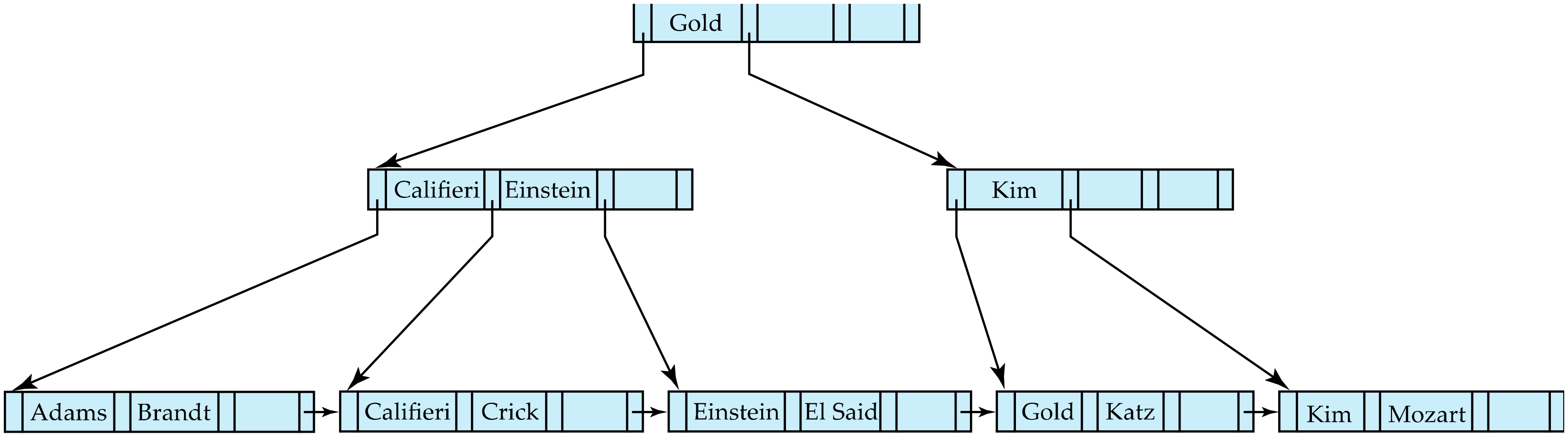
Deleting Singh and Wu



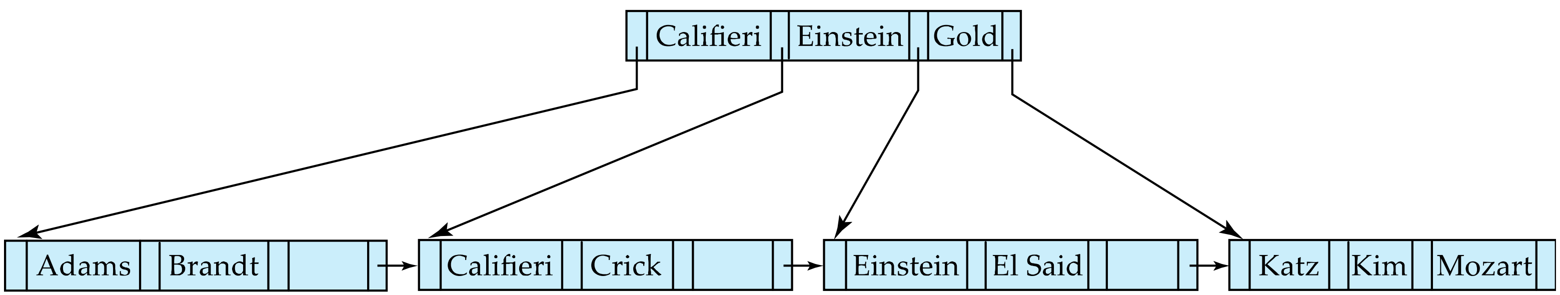
- Delete Singh and Wu and their pointers from leaf node
- N has too few nodes
- Let N’ be the node of Gold (sibling of Singh/Wu)
- Let K′ be the value between pointers N and N′ in parent(N). K’ is Mozart
- Entries of N and N’ cannot fit on a node, therefore redistribute
- N’ is a predecessor of N and N is a leaf node
- Let (N’.P_m, N’.K_m), ie, (pointer to Kim, Kim) be the last pairs in N’
- Remove (pointer to Kim, Kim**)** from N’ and insert into N
- Replace K’, that is, Mozart, in parent of N by N’.K_m, that is, Kim



Example of B⁺-tree Deletion (Cont.)

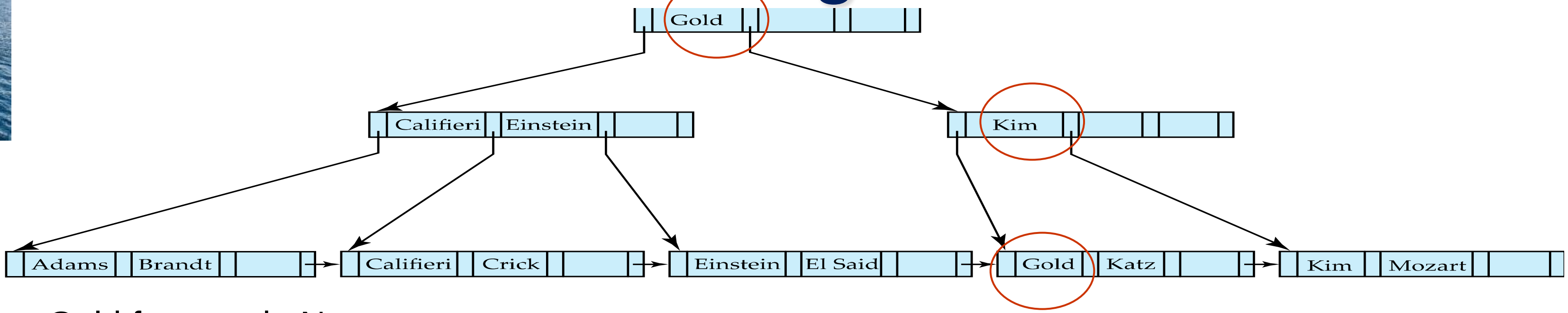


Before and after deletion of "Gold"



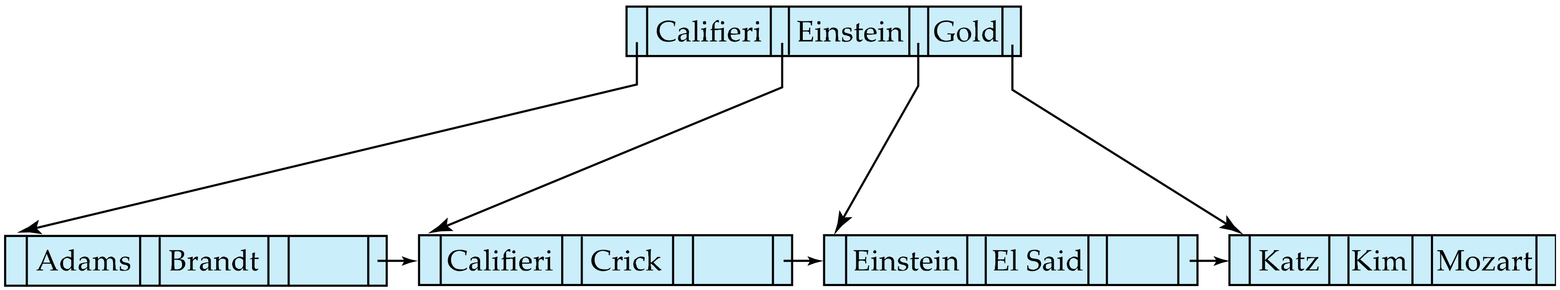
- Node with Gold and Katz became underfull, and was merged with its sibling
- Parent node becomes underfull, and is merged with its sibling
  - Value separating two nodes (at the parent) is pulled down when merging
- Root node then has only one child, and is deleted

Deleting Gold



- Delete Gold from node N
- N has now too few values. Let N' be the next child of the parent of N, that is, the node starting with Kim
- Let K' be the value in parent between N and N' (K' is Kim)
- Entries in N and N' can fit in a node. N is a predecessor of N'. Therefore swap the values in the variables N and N' in this algorithm – there is no change to the tree
- N (now node with Kim) is a leaf. Append all entries in N to N' (node with Gold)
- Delete (parent of N, K', N.) – recursive deletion
  - Assume N is now parent of leaf node of Kim. .Delete Kim and its pointer to the child leaf node. N has no values now
  - Let the sibling node of Califeri be N'
  - Let K' be the value in parent between N and N' (K' is Gold)
  - Coalesce N and N' – that is, node with Califeri and node with Kim
  - N is not a leaf. Append K' (that is Gold) and (pointer, value) pairs in N to node with Califeri
  - Delete parent of Kim
    - Delete Gold from root. Root has only one child now. Make the child the root
  - Delete node of Kim
- Delete node N

Deleting Gold cont.



Complexity of Updates

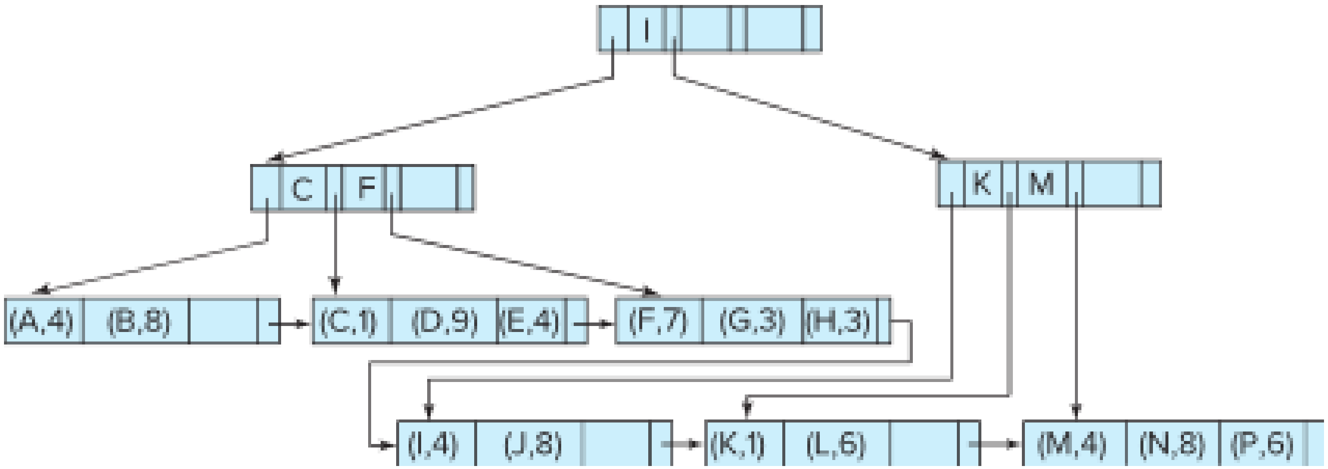
- Cost (in terms of number of I/O operations) of insertion and deletion of a single entry proportional to height of the tree
  - With K entries and maximum fanout (maximum number of pointers of a node) of n, worst case complexity of insert/delete of an entry is $O(\log_{\lceil n/2 \rceil}(K))$
- In practice, 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. Why is this so?

B^+-Tree File Organization

- B^+-Tree File Organization:
  - Leaf nodes in a B^+-tree file organization store records, instead of pointers
  - Helps keep data records clustered even when there are insertions/deletions/updates
- Leaf nodes are still required to be half full
  - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B^+-tree index.
  - If a block has enough free space, the record is stored in the block
  - Otherwise, as in B+-tree insertion
    - the system splits the block in two
    - redistributes the records in it (in the B+-tree–key order) to create space for the new record (this avoids file reorganization))
    - The split propagates up the B+-tree in the normal fashion

B+-Tree File Organization (Cont.)

- Example of B+-tree File Organization



- Good space utilization important since records use more space than pointers.

Creating indices in SQL

- create index *<index-name> on <relation-name> (<attribute-list>);*
  - Example: create index *dept_index on instructor (dept_name);*
- drop index *<index-name>;*
- If a relation is declared to have a primary key, most database systems automatically create an index on the primary key. Why?