Chapter 11: Indexing and Hashing

Database System Concepts, 6th Ed.

Chapter 11: Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B^+-Tree Index Files
- Multiple-Key Access
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Bitmap Indices
- Index Definition in SQL

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
- **Search Key** - 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

- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

Index Evaluation Metrics

- Access types supported efficiently. E.g.,
  - records with a specified value in the attribute (point query)
  - or records with an attribute value falling in a specified range of values (range query)
- Access time
- Insertion time
- Deletion time
- Space overhead

Chapter 11: Indexing and Hashing

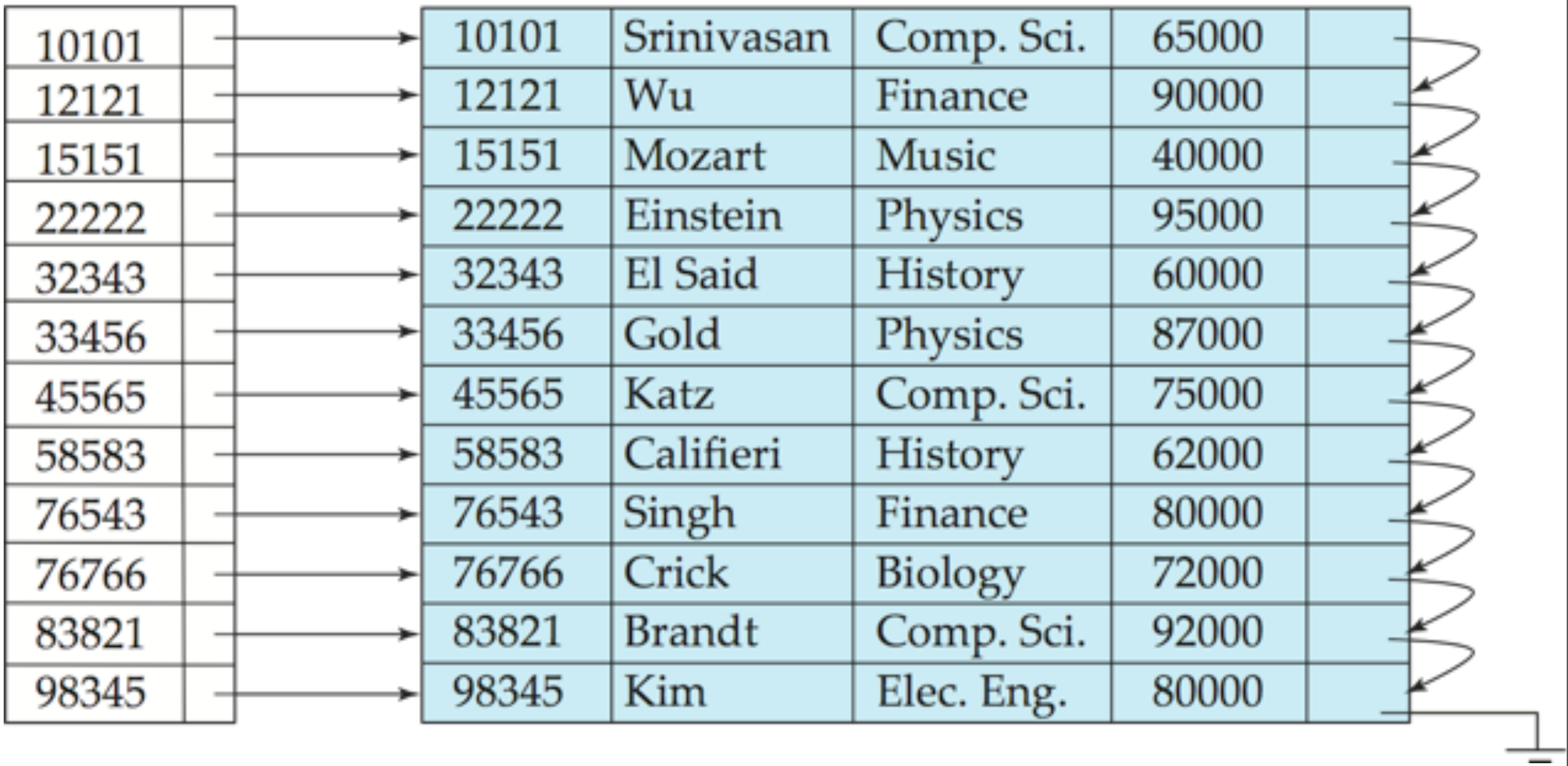
- Basic Concepts
- **Ordered Indices**
- B^+-Tree Index Files
- Multiple-Key Access
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Bitmap Indices
- Index Definition in SQL

Ordered Indices

- **ordered index:** index entries are stored sorted on the search key value.
- **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**.**
- **indexed-sequential file:** ordered sequential file with a primary index.

Dense Index Files

- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation



Sparse Index Files

- **Sparse Index**: contains index records for **only some search-key values**.
  - *Applicable when records are sequentially ordered on search-key*
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - *Search file sequentially* starting at the record to which the index record points

10101	
32343	
76766	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Sparse Index Files (Cont.)

- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than dense index for locating records.

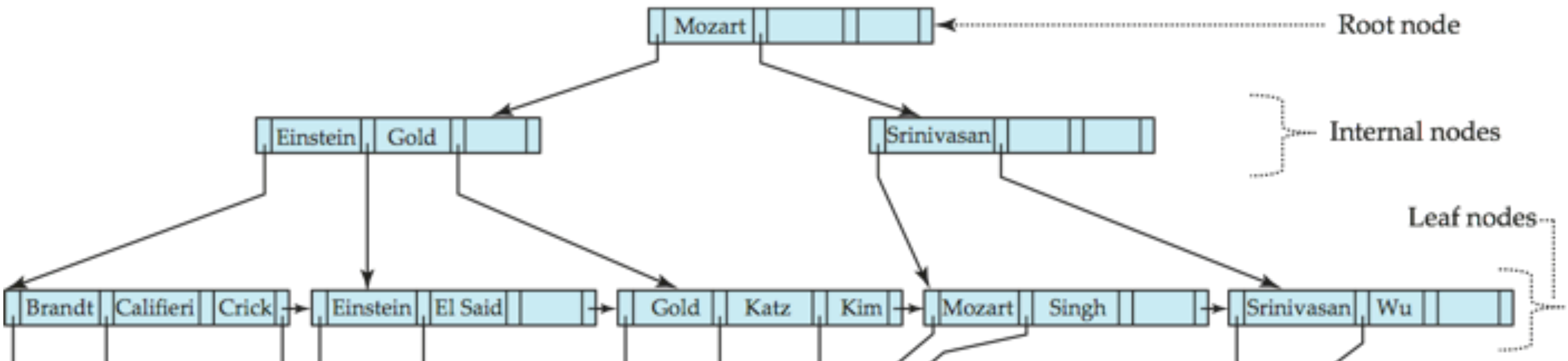
Chapter 11: Indexing and Hashing

- Basic Concepts
- Ordered Indices
- **B+-Tree Index Files**
- Multiple-Key Access
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Bitmap Indices
- Index Definition in SQL

B+-Tree Index Files

- 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



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
- 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**

B^+-Tree Node Structure

- Typical node

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

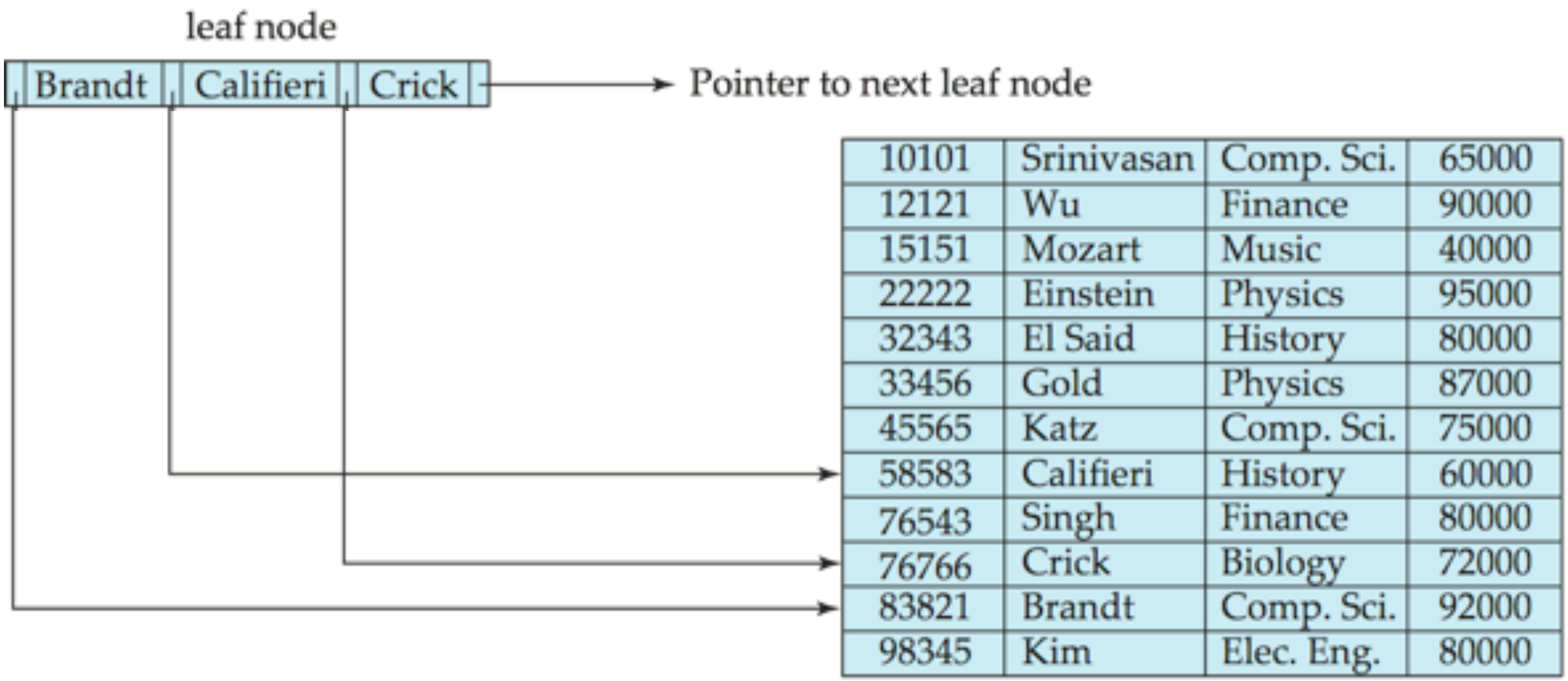
  - K_i are the search-key values
  - P_i are **pointers to children** (for **non-leaf nodes**) or pointers to **records** or buckets of records (for **leaf nodes**).
- The search-keys in a node are ordered

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

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,
- P_n points to next leaf node in search-key order

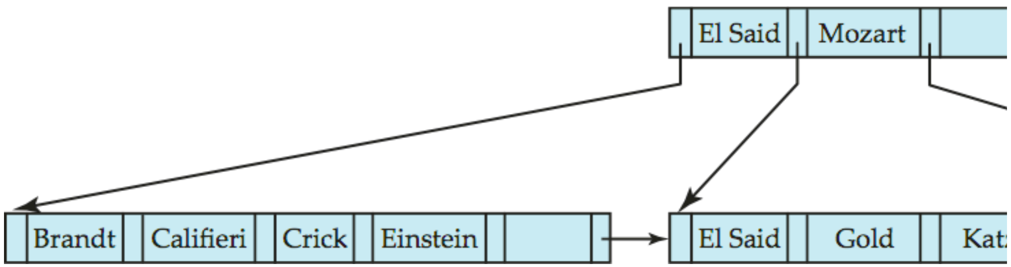


10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

Non-Leaf Nodes in B+-Trees

- For a non-leaf node with *n* 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	…	P_{n-1}	K_{n-1}	P_n



Queries on B+-Trees

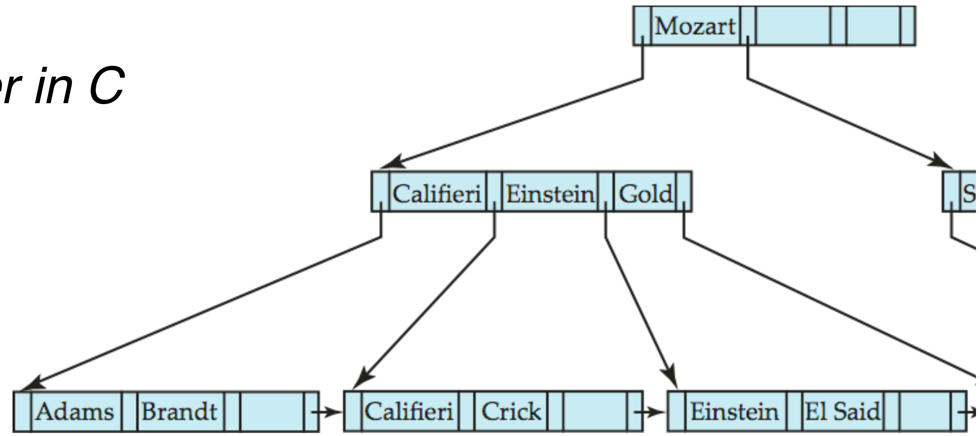
- Find record with search-key value *V.*
  1. *C=root*
  2. While C is not a leaf node {
     1. Let *i* be least value s.t. $K_i \geq V$.
     2. If no such exists,
        - *C = last non-null pointer in C*
     3. Else if ($V = K_i$)
        - $C = P_{i+1}$
     - else
        - $C = P_i$

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

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. $K_i \geq V$.
     2. If no such exists,
        – *C = last non-null pointer in C*
     3. Else if ($V = K_i$)
        – $C = P_{i+1}$
     - else
        – $C = P_i$

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



Queries on B+-Trees (Cont.)

- If there are K search-key values, 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

Updates on B^+-Trees: Insertion

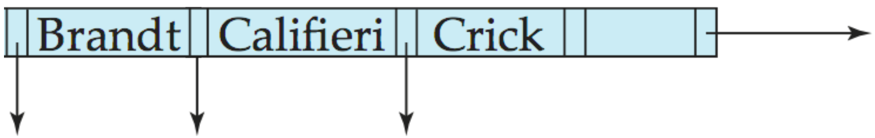
1. Find the **leaf node** in which the search-key value would appear
2. If the search-key value is already present in the leaf node
    1. Add record to the file
    2. If necessary add a pointer to the bucket.
3. If the search-key value is not present, then
    1. add the record to the main file (and create a bucket if necessary)
    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 (Cont.)

- Splitting a leaf node:
  - Keep the first $\lceil n/2 \rceil$ (search-key value, pointer) pairs in the original node, and place 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 (**overfull**), split it and **propagate** the split further up (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.

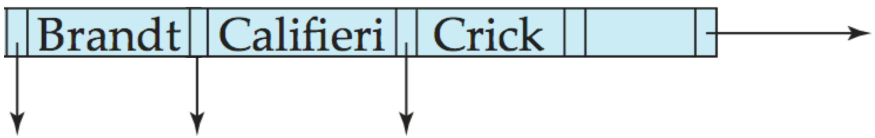
Updates on B+-Trees: Insertion (Cont.)

- Splitting a leaf node:
  - Keep the first $\lceil n/2 \rceil$ (search-key value, pointer) pairs in the original node, and place 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 (**overfull**), split it and **propagate** the split further up (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.



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

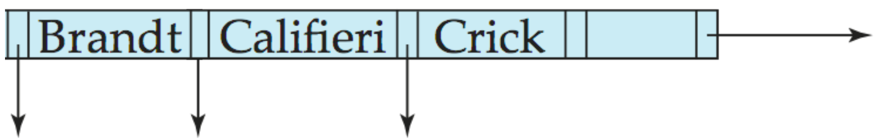
- Splitting a leaf node:
  - Keep the first $\lceil n/2 \rceil$ (search-key value, pointer) pairs in the original node, and place the rest in a new node.
  - Let the new node be *p,* and let *k* be the least key value in *p.* Insert (*k,p*) in the parent of the node being split.
  - If the parent is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



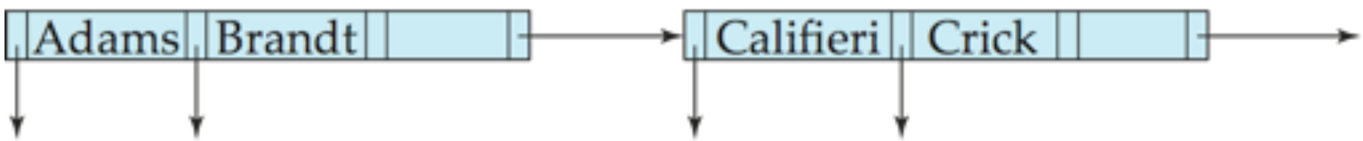
After inserting Adams

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

- Splitting a leaf node:
  - Keep the first $\lceil n/2 \rceil$ (search-key value, pointer) pairs in the original node, and place 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 (**overfull**), split it and **propagate** the split further up (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.

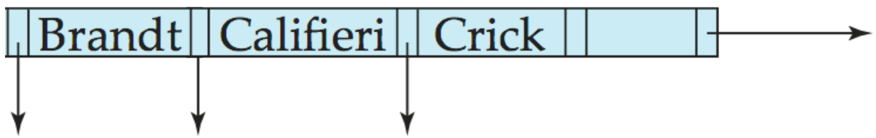


After inserting Adams

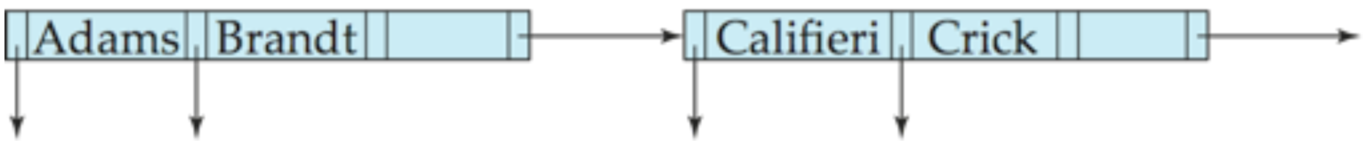


Updates on B+-Trees: Insertion (Cont.)

- Splitting a leaf node:
  - Keep the first $\lceil n/2 \rceil$ (search-key value, pointer) pairs in the original node, and place 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 (**overfull**), split it and **propagate** the split further up (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.

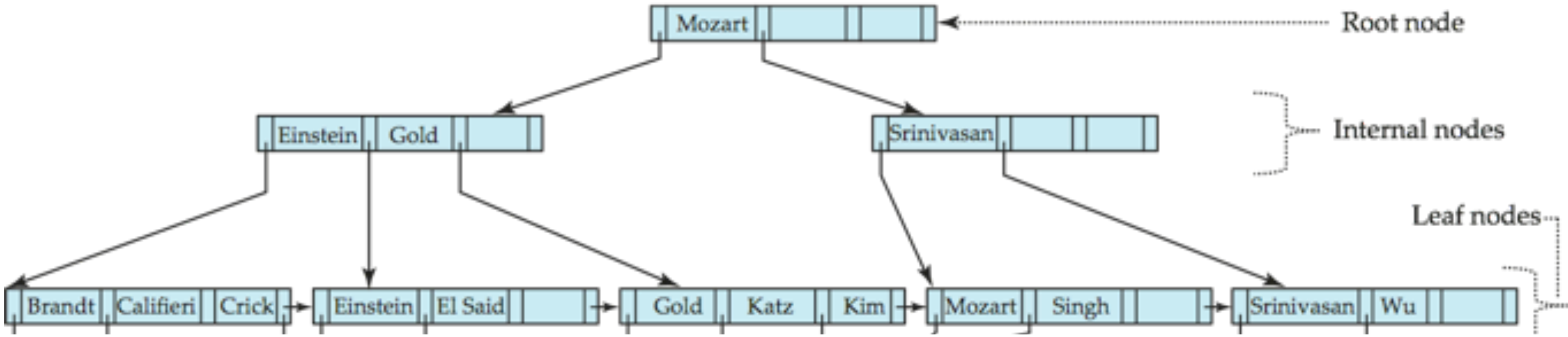


After inserting Adams

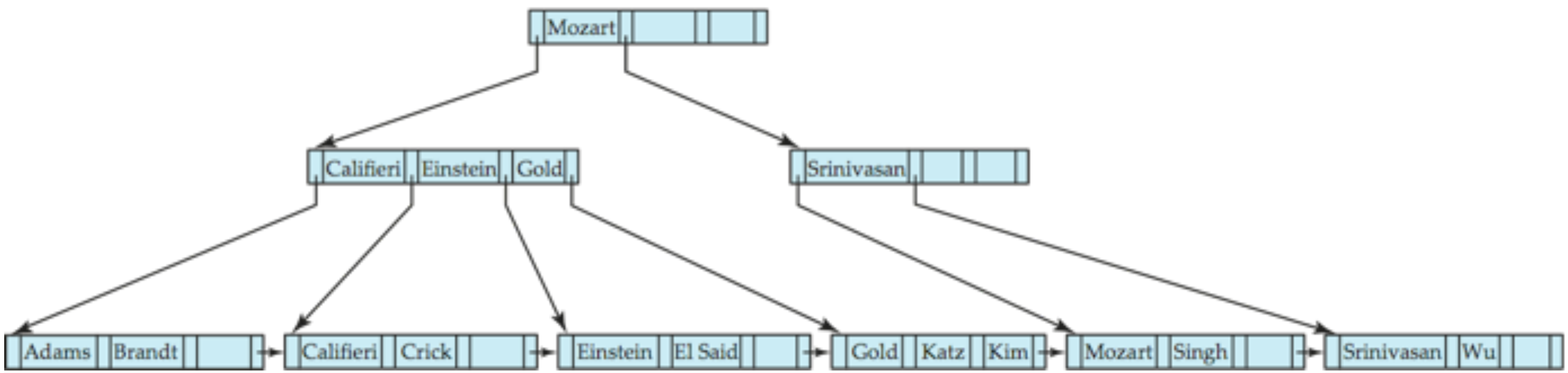


Next step: insert entry with (Califieri,pointer-to-new-node) into parent

B+-Tree Insertion

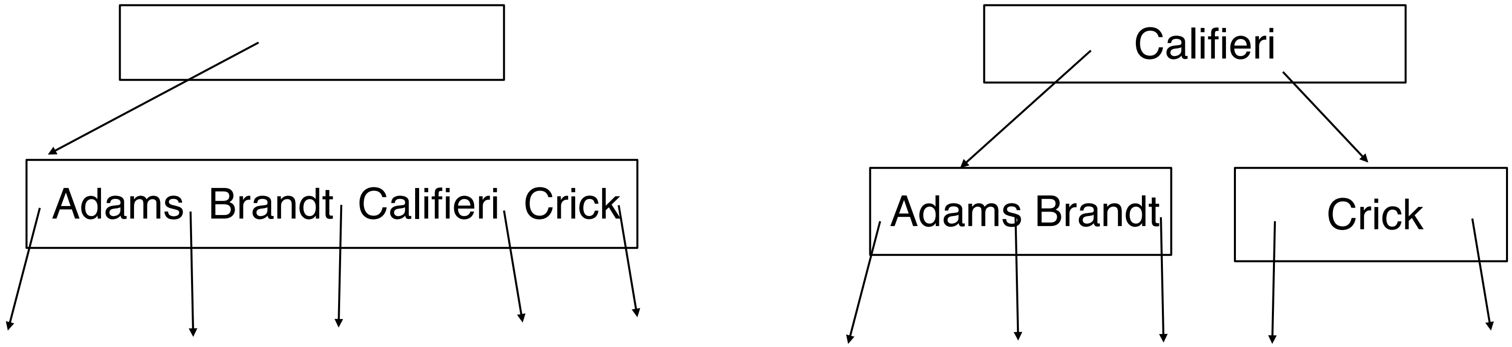


B+-Tree before and after insertion of "Adams"



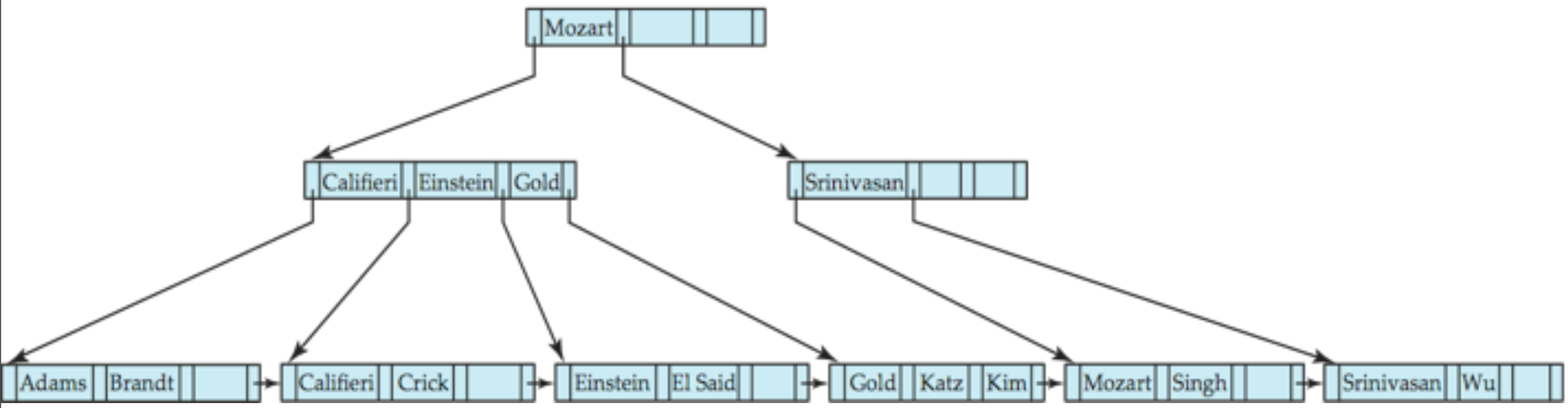
Insertion in B+-Trees (Cont.)

- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N

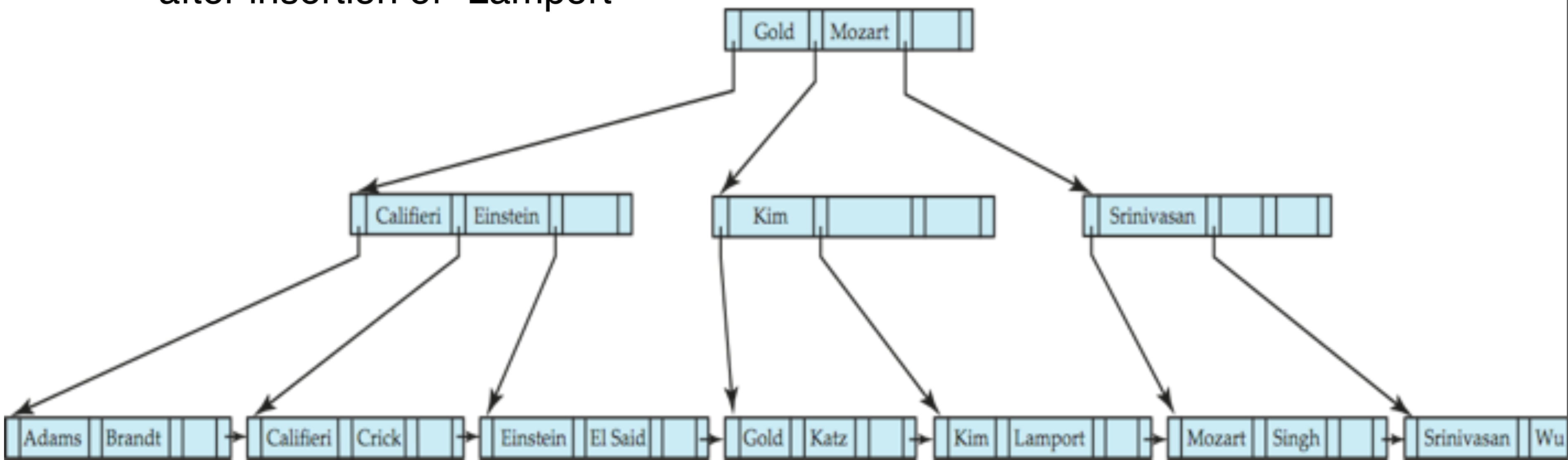


  - Copy N to an in-memory area M with space for n+1 pointers and n keys
  - Insert (k,p) into M
  - Copy $P_1, K_1, \ldots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node N
  - Copy $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \ldots, K_n, P_{n+1}$ from M into newly allocated node N'
  - Insert $(K_{\lceil n/2 \rceil}$,N') into parent N
- **Read pseudocode in book!**

B+-Tree Insertion



after insertion of "Lamport"



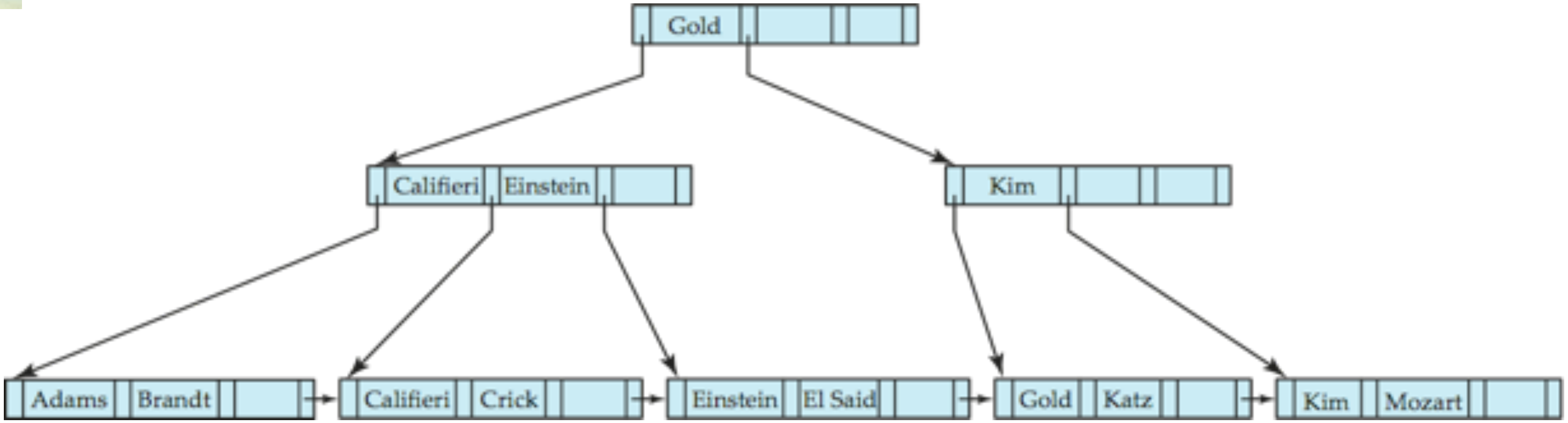
Updates on B+-Trees: Deletion

- Find the record to be deleted, and remove it from the main file
- Remove (search-key value, pointer) from the leaf node if the bucket has become empty
- If the node has too few entries due to the removal (**underfull**), 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.

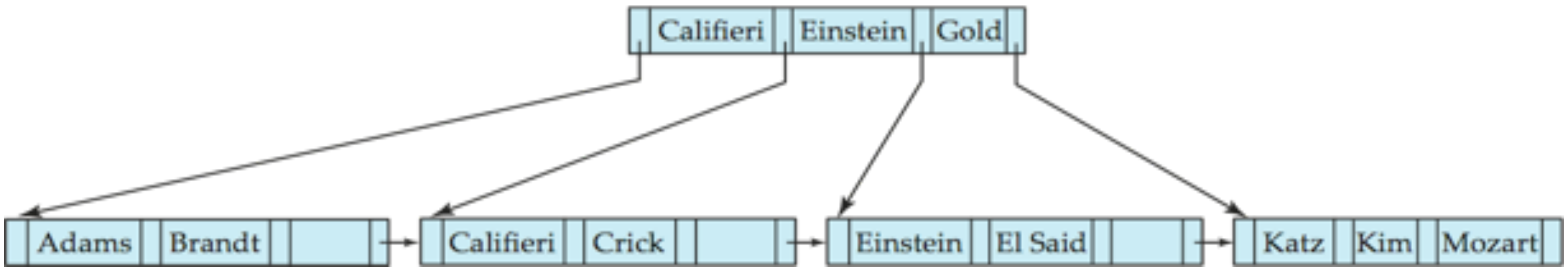
Updates on B+-Trees: Deletion

- 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**:
  - 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 (the tree height decreases).

Example of B+-tree Deletion

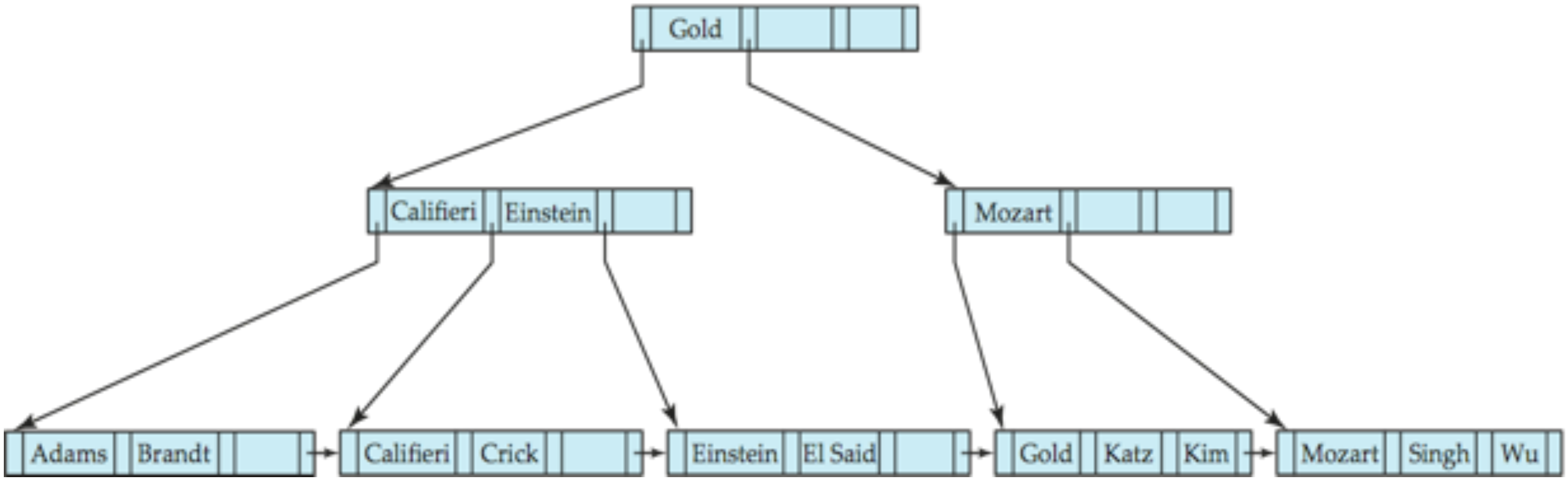


after deletion of "Gold"

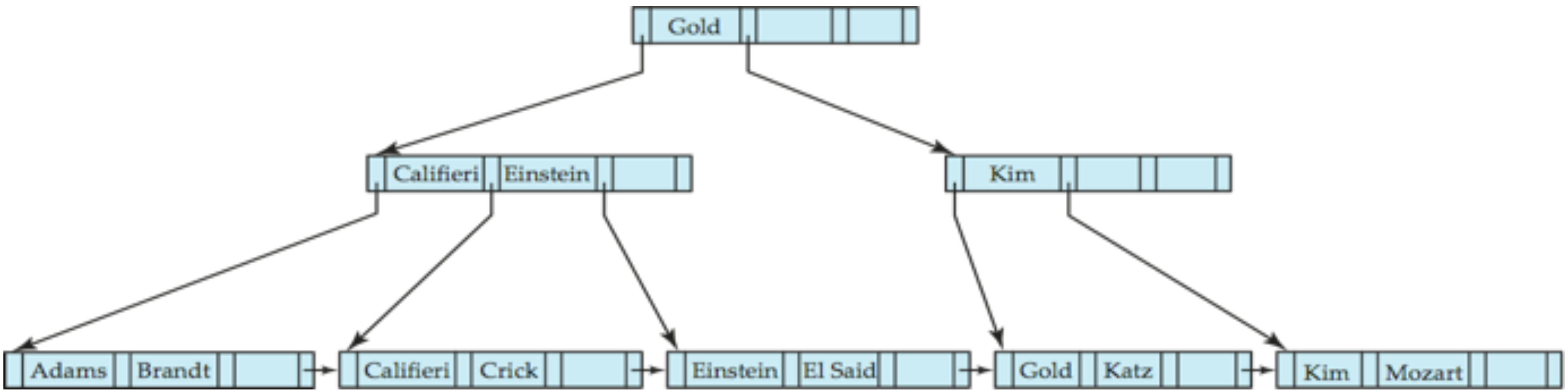


- Node with Gold and Katz became underfull, and was **merged** with its sibling
- Parent node becomes underfull, and is merged with its sibling
  - **Value separating two nodes** (at the parent) is pulled down when **merging**
- Root node then has only one child, and is deleted

Examples of B+-Tree Deletion (Cont.)

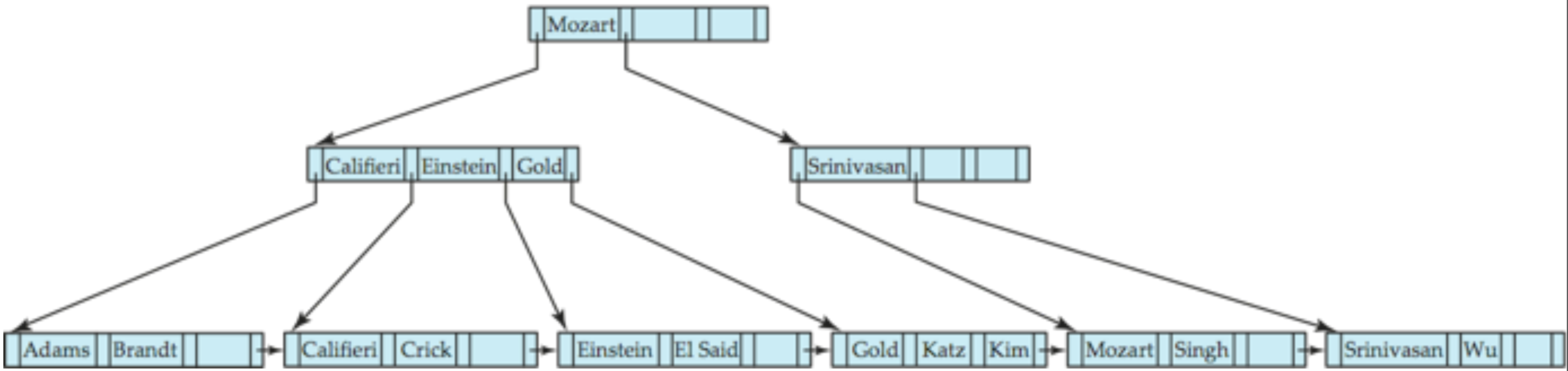


Deletion of "Singh" and "Wu"

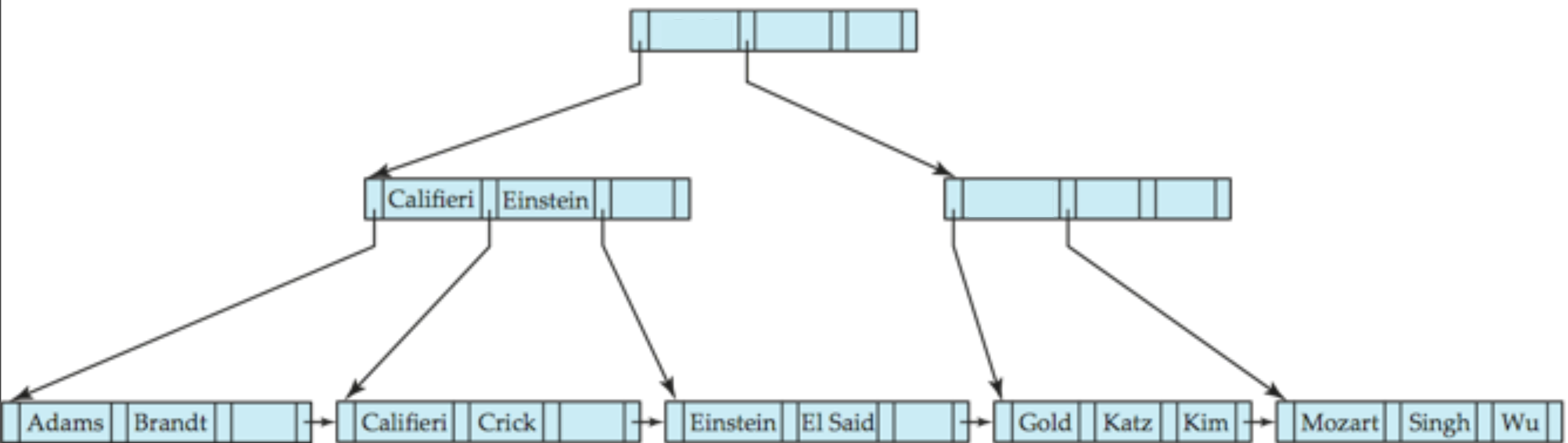


- Leaf containing Singh and Wu became underfull, and borrowed a value Kim from its left sibling (**rebalancing**)

Examples of B+-Tree Deletion

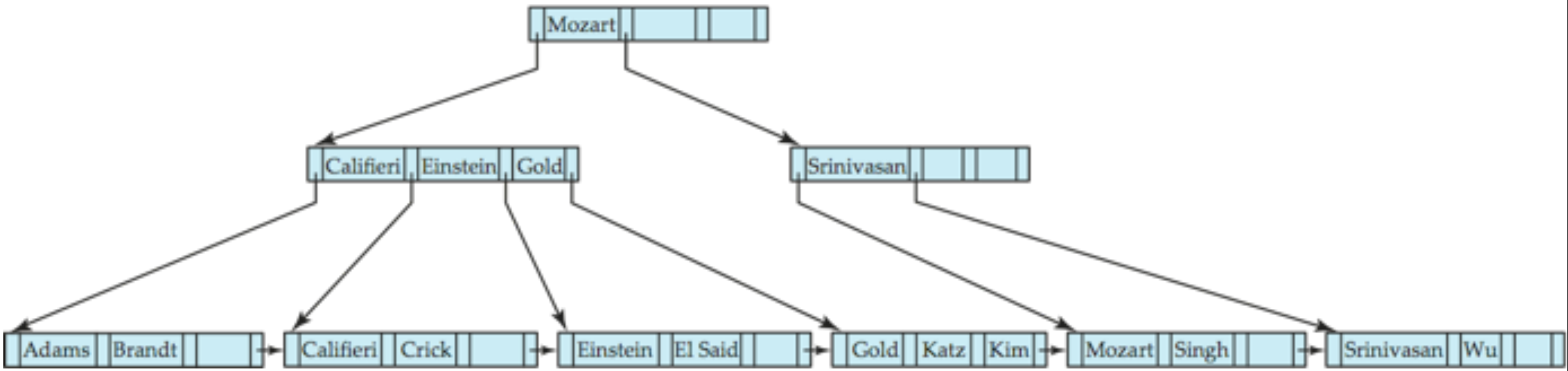


Before and after deleting "Srinivasan"

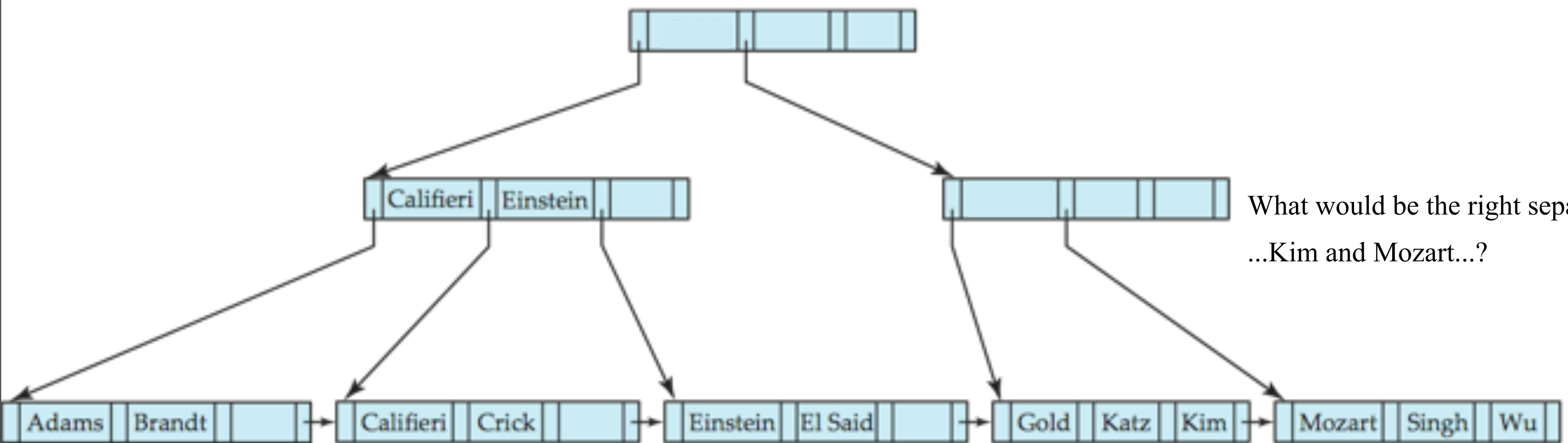


- Deleting "Srinivasan" causes merging of under-full leaves

Examples of B+-Tree Deletion



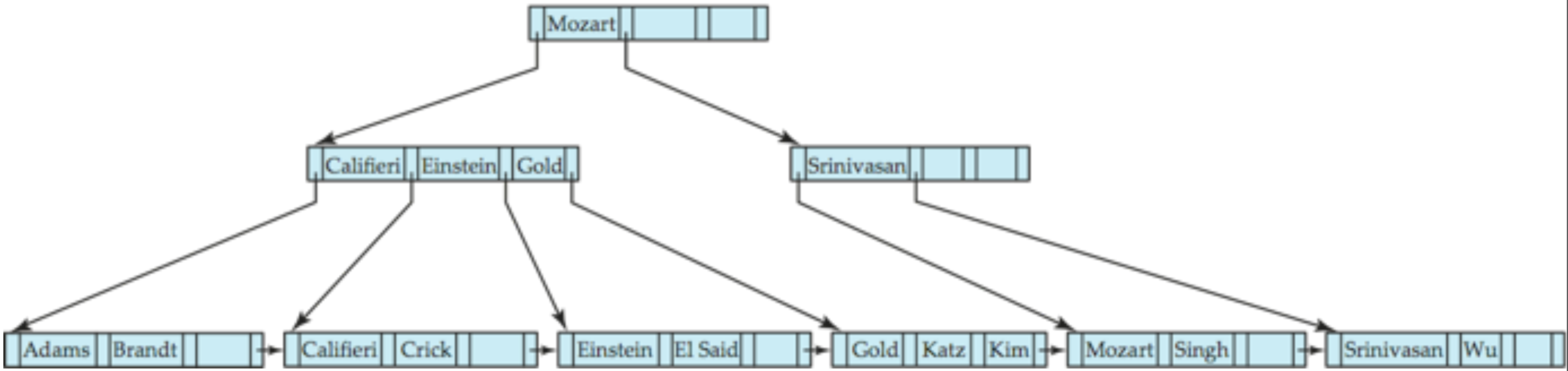
Before and after deleting "Srinivasan"



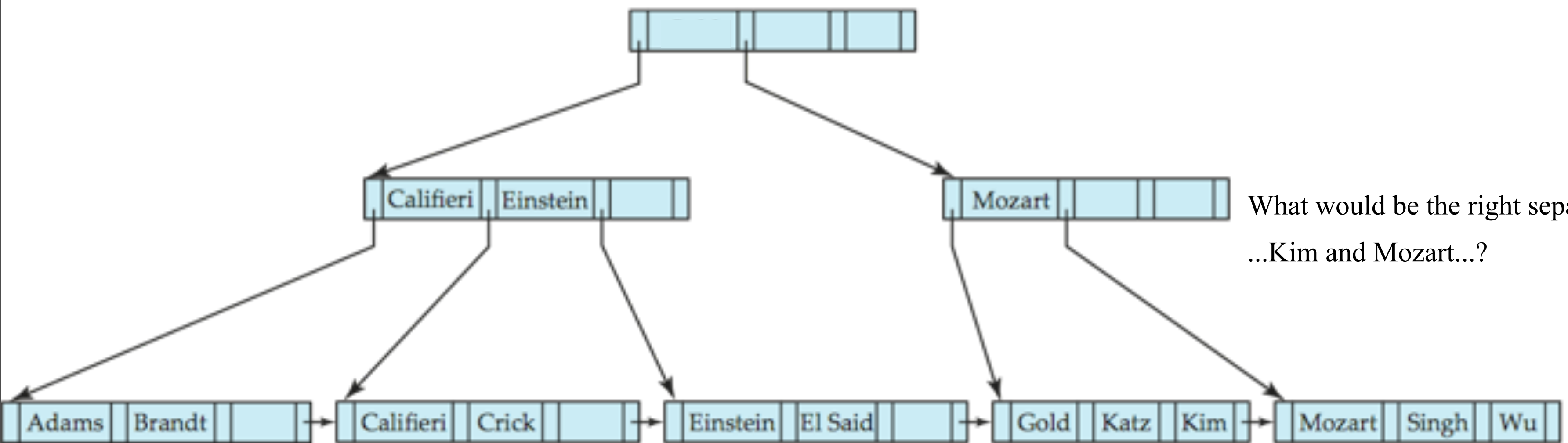
What would be the right separator for ...Kim and Mozart...?

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

Examples of B+-Tree Deletion



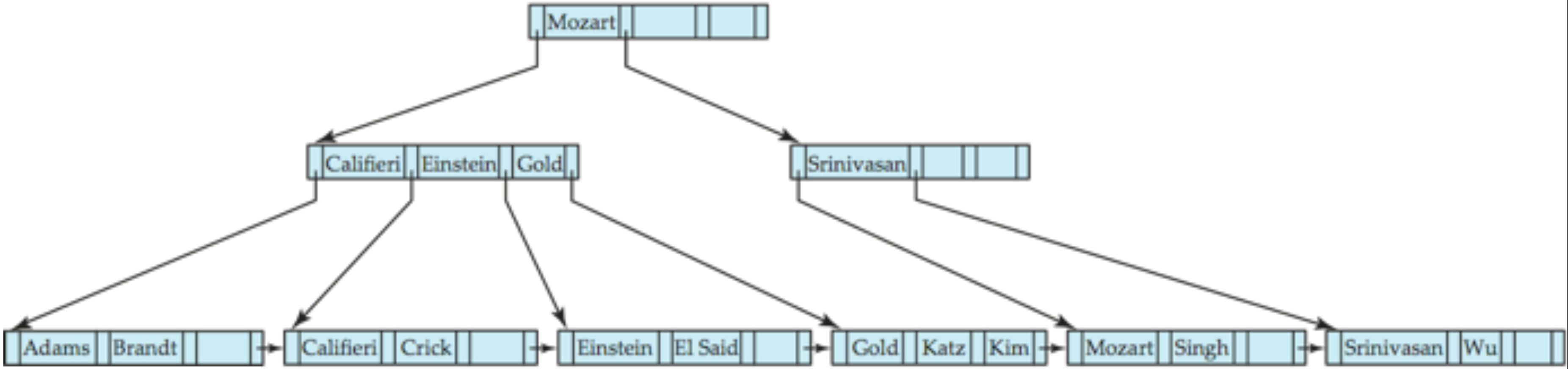
Before and after deleting "Srinivasan"



What would be the right separator for ...Kim and Mozart...?

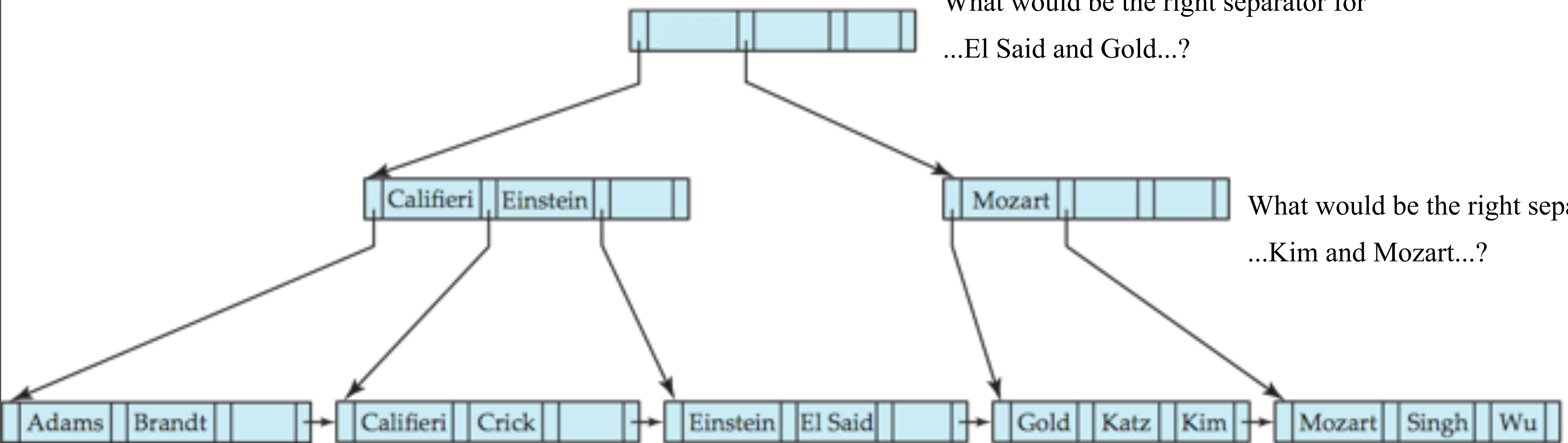
- Deleting "Srinivasan" causes merging of under-full leaves

Examples of B+-Tree Deletion



Before and after deleting “Srinivasan”

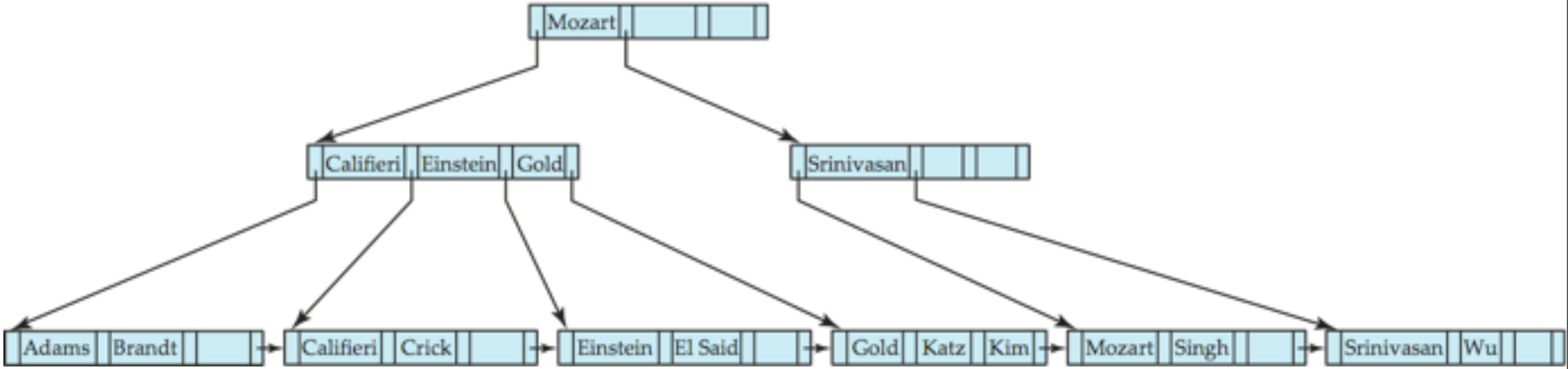
What would be the right separator for ...El Said and Gold...?



What would be the right separator for ...Kim and Mozart...?

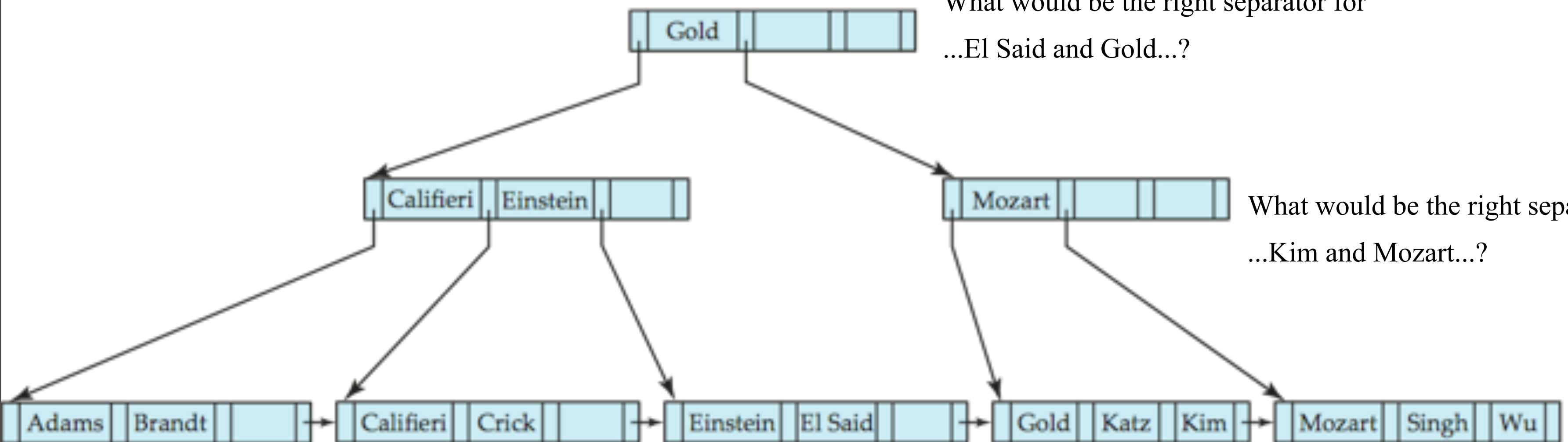
- Deleting “Srinivasan” causes merging of under-full leaves

Examples of B+-Tree Deletion



Before and after deleting "Srinivasan"

What would be the right separator for ...El Said and Gold...?



What would be the right separator for ...Kim and Mozart...?

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

Chapter 11: Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B^+-Tree Index Files
- **Multiple-Key Access**
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Bitmap Indices
- Index Definition in SQL

Multiple-Key Access

- Use multiple indices for certain types of queries.
- Example:

  select *ID*

  from *instructor*

  where *dept_name* = "Finance" **and** *salary* = 80000

- Possible strategies for processing query using indices on single attributes:
  1. Use **index on *dept_name*** to find instructors with department name Finance; test *salary = 80000*
  2. Use **index on *salary*** to find instructors with a salary of $80000; test *dept_name =* "Finance".
  3. Use ***dept_name* index** to find pointers to all records pertaining to the "Finance" department. Similarly **use index on *salary***. Take intersection of both sets of pointers obtained.

Indices on Multiple Keys

- **Composite search keys** are search keys containing more than one attribute
  - E.g. (*dept_name, salary*)
- Lexicographic ordering: $(a_1, a_2) < (b_1, b_2)$ if either
  - $a_1 < b_1$, or
  - $a_1=b_1$ and $a_2 < b_2$

Indices on Multiple Keys

- **Composite search keys** are search keys containing more than one attribute
  - E.g. (*dept_name, salary*)
- Lexicographic ordering: $(a_1, a_2) < (b_1, b_2)$ if either
  - $a_1 < b_1$, or
  - $a_1 = b_1$ and $a_2 < b_2$

(ID, dept_name, salary)

1, Art, 20000

2, Art, 40000

3, Art, 60000

4, Art, 80000

5, Business, 20000

6, Business, 40000

7, Business, 60000

8, Business, 80000

9, Finance, 20000

10, Finance, 40000

11, Finance, 60000

12, Finance, 80000

Indices on Multiple Attributes

Suppose we have an index on combined search-key (*dept_name, salary*).

- Can efficiently handle

  where *dept_name* = "Finance" **and** *salary* = 80000

- Can also efficiently handle

  where *dept_name* = "Finance" **and** *salary* < 80000

- But cannot efficiently handle

  where *dept_name* < "Finance" **and** *balance* = 80000

Indices on Multiple Attributes

Suppose we have an index on combined search-key (*dept_name, salary*).

- Can efficiently handle

  where *dept_name* = "Finance" **and** *salary* = 80000
- Can also efficiently handle

  where *dept_name* = "Finance" **and** *salary* < 80000
- But cannot efficiently handle

  where *dept_name* < "Finance" **and** *balance* = 80000

(ID, dept_name, salary)
1, Art, 20000
2, Art, 40000
3, Art, 60000
4, Art, 80000
5, Business, 20000
6, Business, 40000
7, Business, 60000
8, Business, 80000
9, Finance, 20000
10, Finance, 40000
11, Finance, 60000
12, Finance, 80000

Chapter 11: Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B^+-Tree Index Files
- Multiple-Key Access
- **Static Hashing**
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Bitmap Indices
- Index Definition in SQL

Static Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function**.
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B.*
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Example of Hash File Organization

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

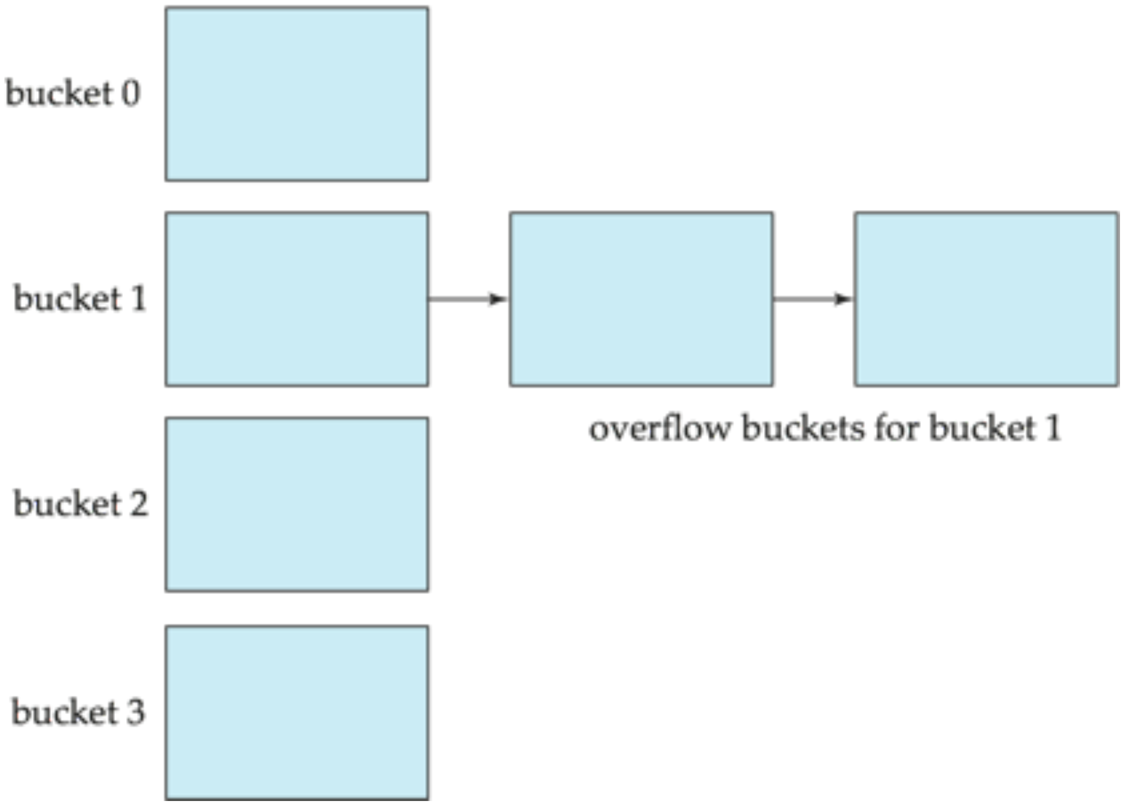
Hash file organization of *instructor* file, using ***dept_name*** as key

Handling of Bucket Overflows

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records.
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using ***overflow chaining (or bucket chaining)***.

Handling of Bucket Overflows (Cont.)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.



Deficiencies of Static Hashing

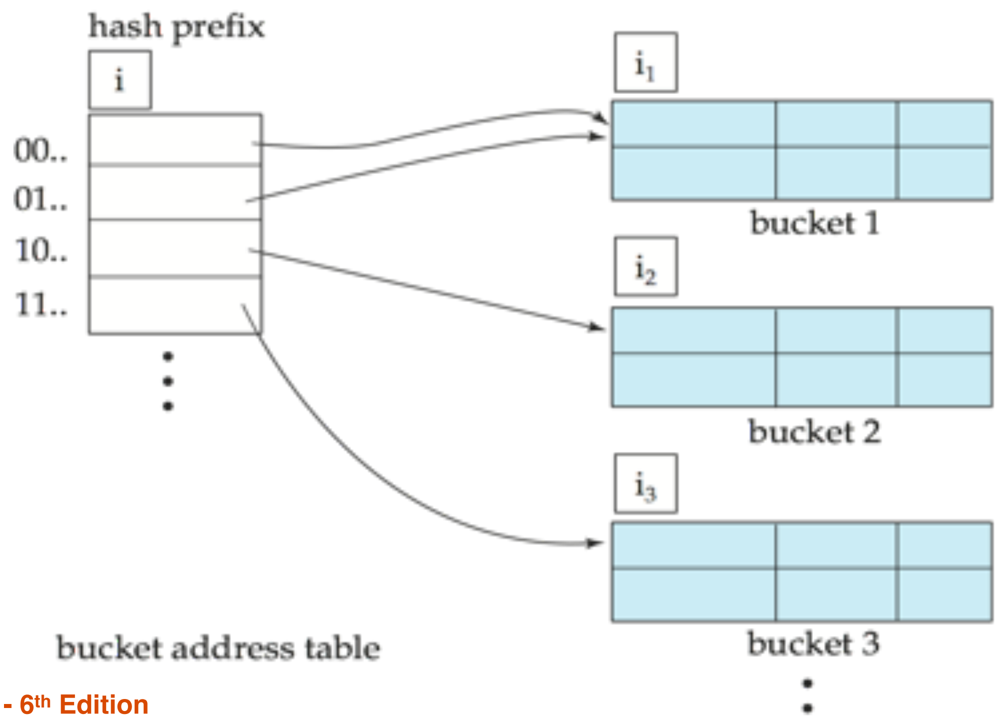
- In static hashing, function *h* maps search-key values to **a fixed set of bucket addresses**. Databases grow or shrink with time.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
  - If database shrinks, again space will be wasted.

Chapter 11: Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B^+-Tree Index Files
- Multiple-Key Access
- Static Hashing
- **Dynamic Hashing**
- Comparison of Ordered Indexing and Hashing
- Bitmap Indices
- Index Definition in SQL

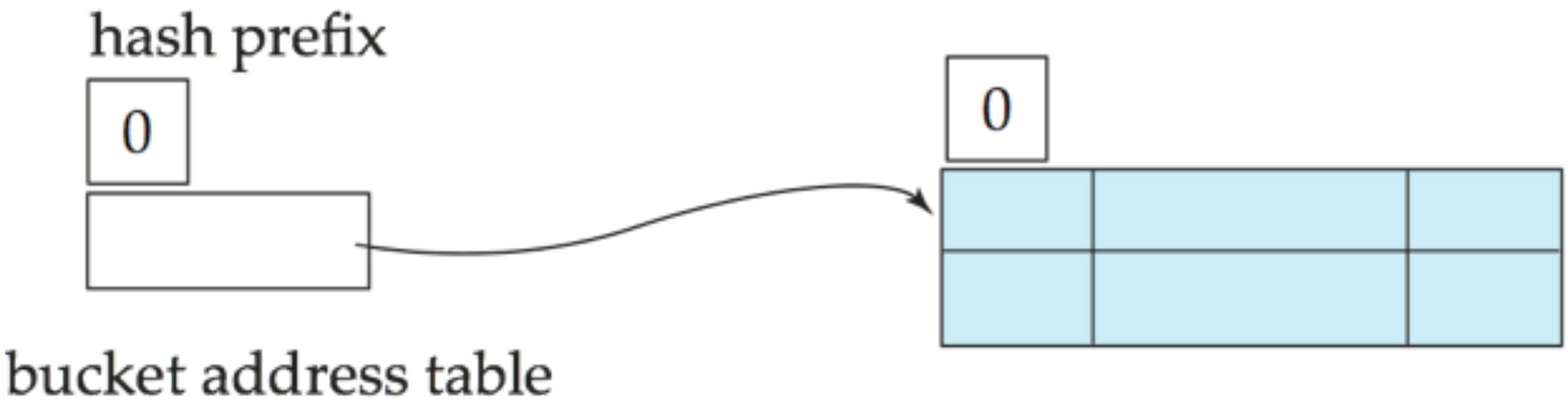
Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- **Extendable hashing** – one form of dynamic hashing
  - At any time use only a prefix of the hash function to index into a table of bucket addresses.
  - The length of the prefix grows and shrinks as the size of the database grows and shrinks..



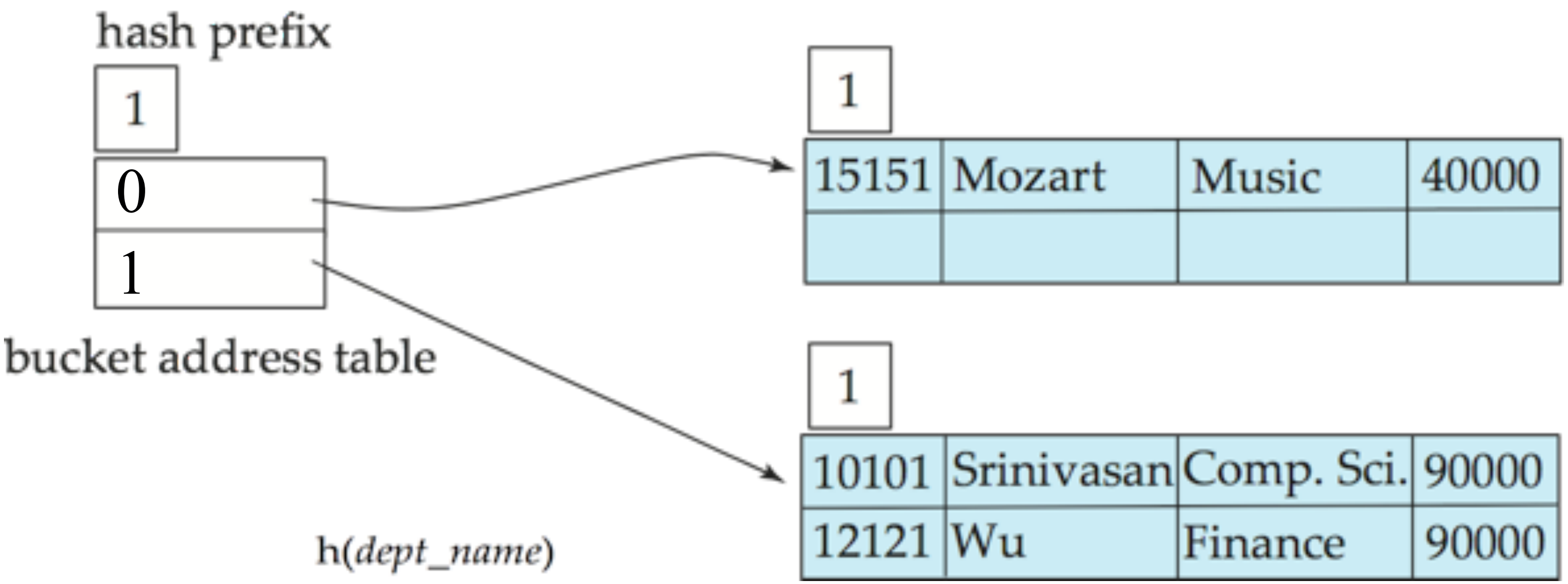
Example (Cont.)

- Initial Hash structure; 1 bucket; bucket size = 2



Example (Cont.)

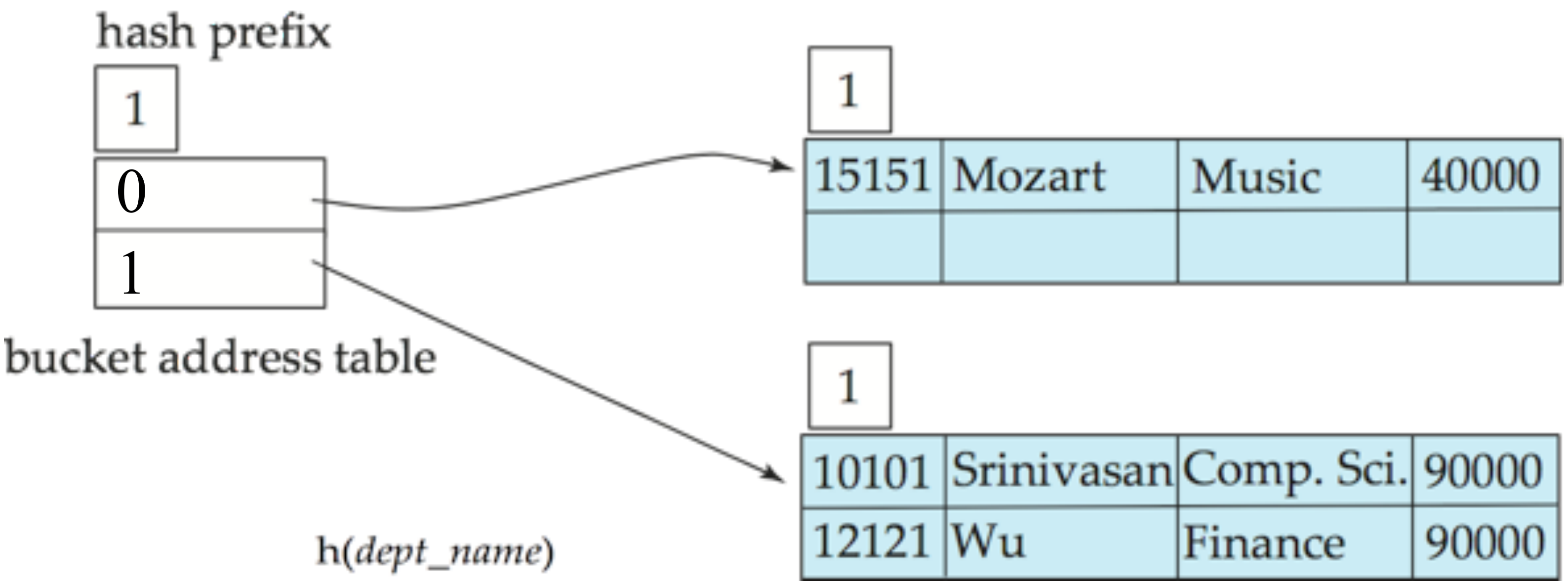
- Hash structure after insertion of “Mozart”, “Srinivasan”, and “Wu” records



dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

Example (Cont.)

- Hash structure after insertion of "Mozart", "Srinivasan", and "Wu" records

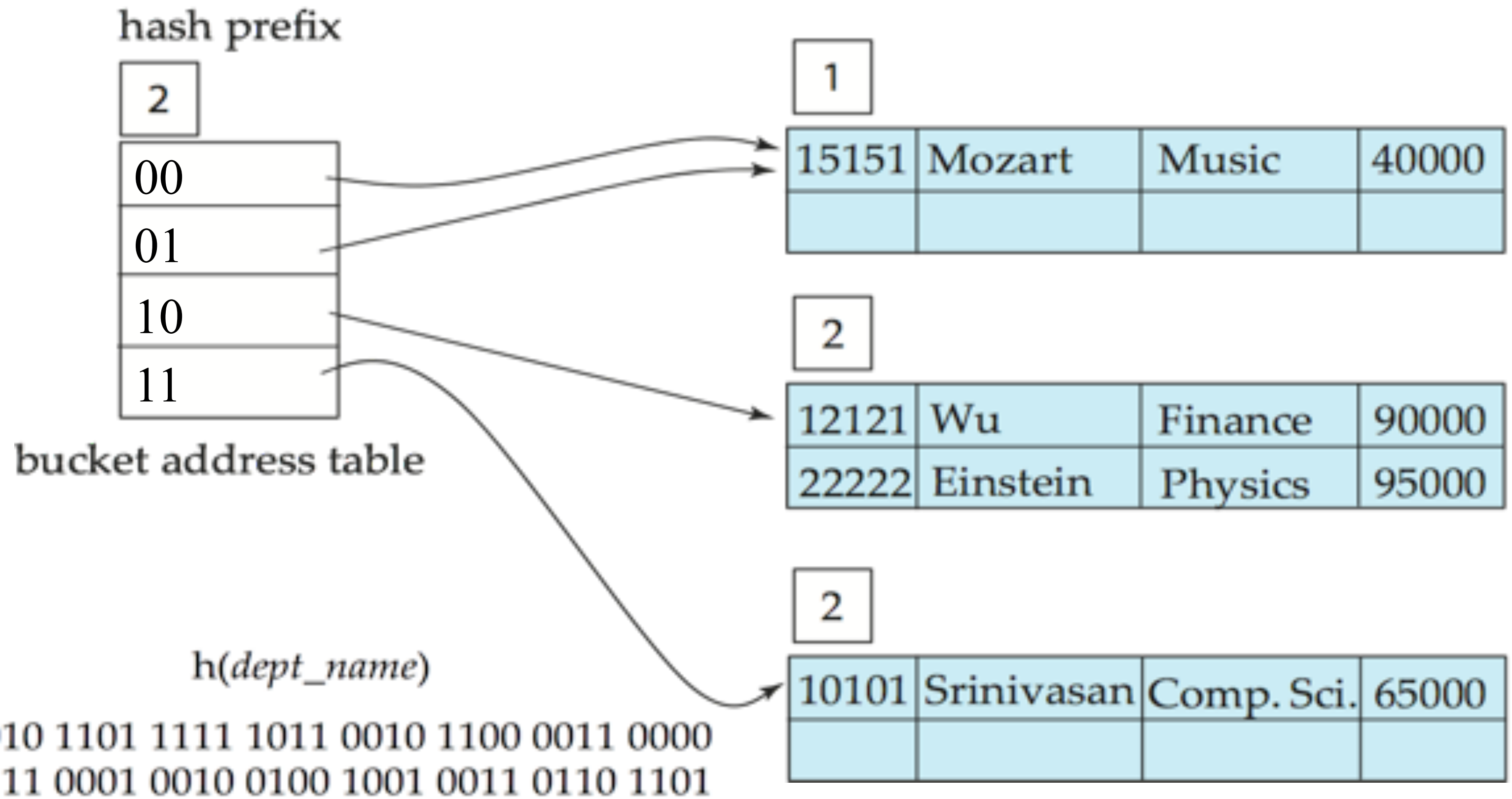


dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

- What if a record about Einstein in **Physics** is entered?

Example (Cont.)

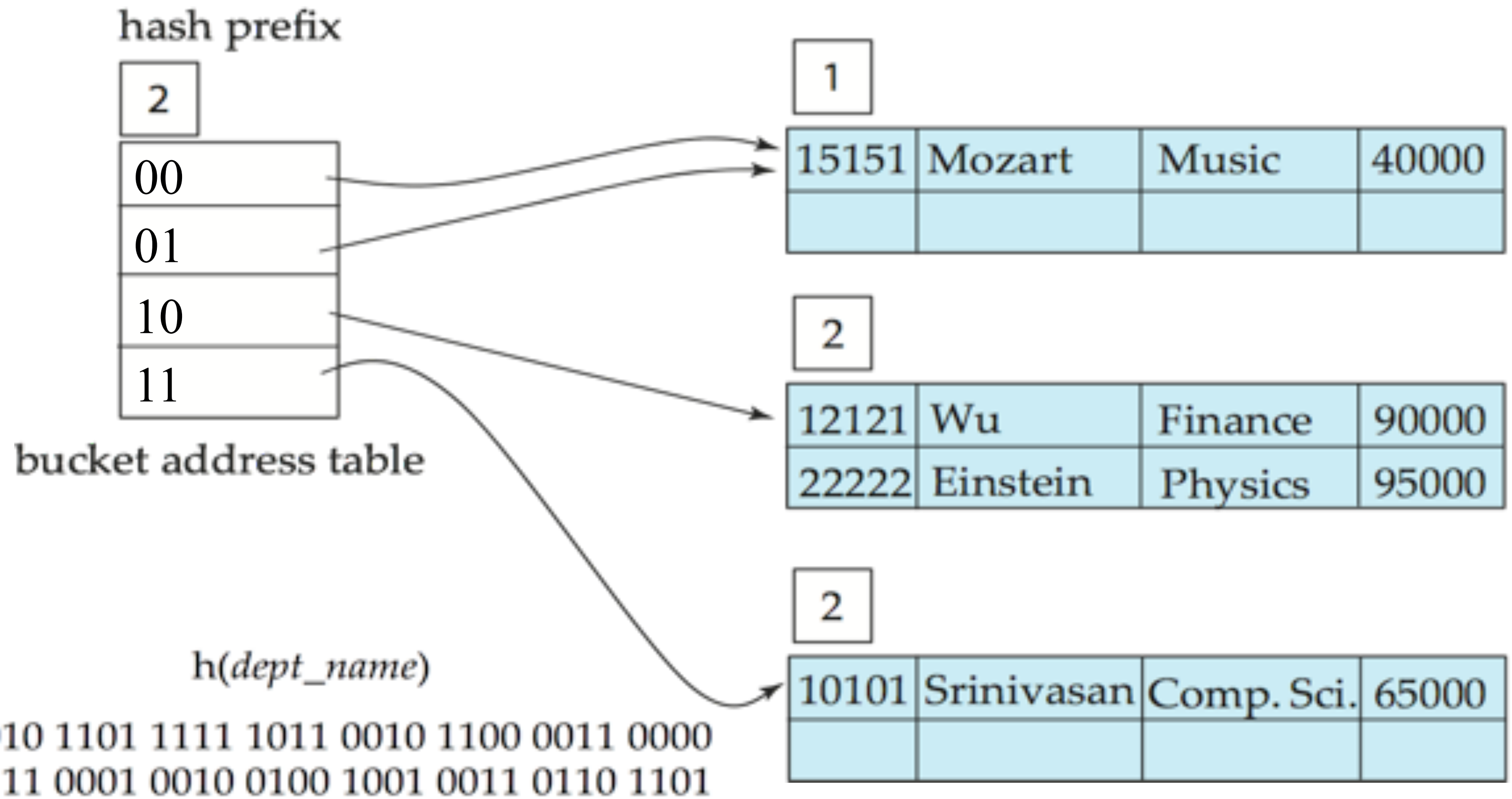
- Hash structure after insertion of Einstein record



dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

Example (Cont.)

- Hash structure after insertion of Einstein record

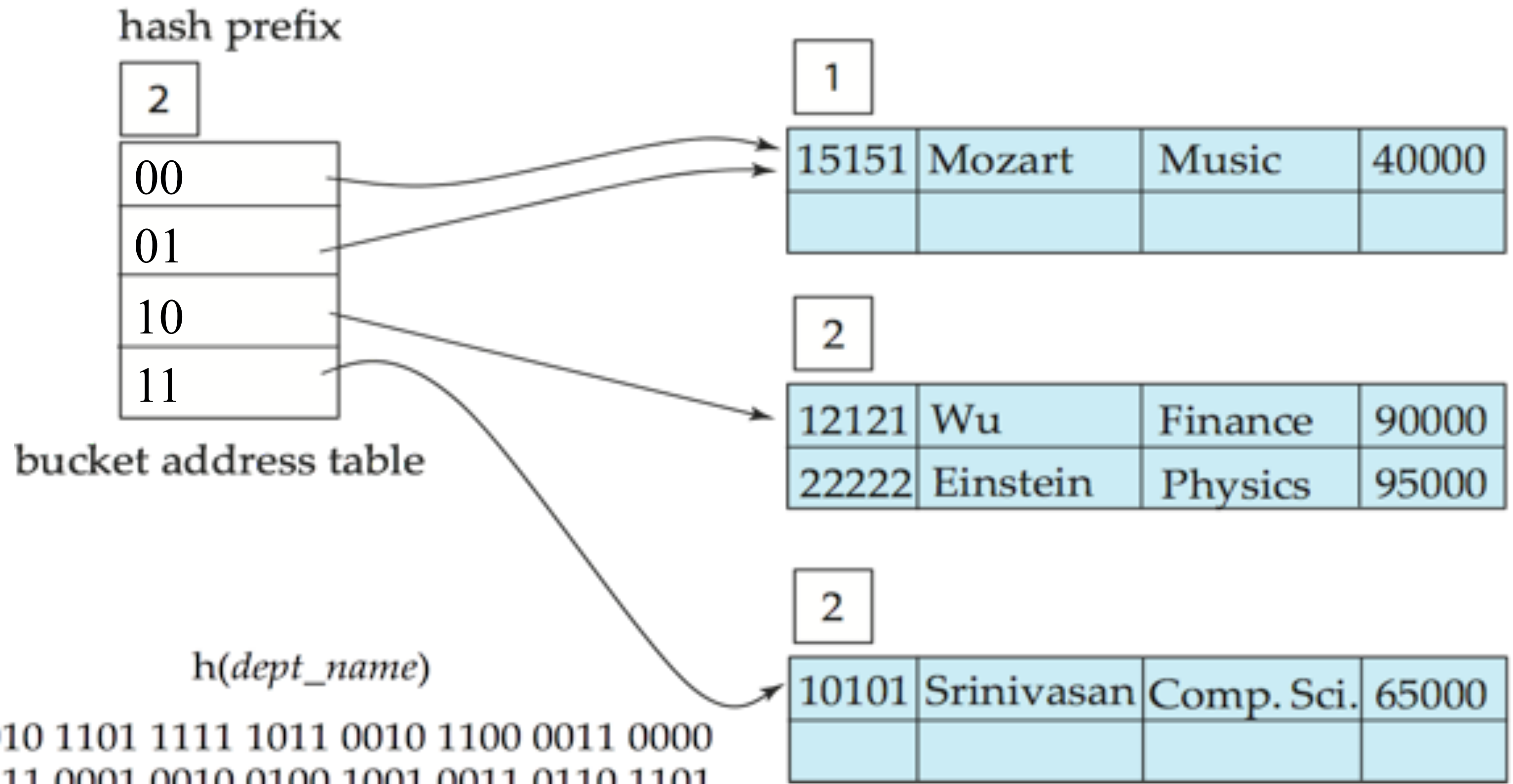


dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

What if (Katz, Comp. Sci.) and (Brandt, Comp. Sci.) are inserted?

Example (Cont.)

- Hash structure after insertion of Einstein record



dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

What if (Katz, Comp. Sci.) and (Brandt, Comp. Sci.) are inserted?

Chaining!!

Chapter 11: Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B^+-Tree Index Files
- Multiple-Key Access
- Static Hashing
- Dynamic Hashing
- **Comparison of Ordered Indexing and Hashing**
- Bitmap Indices
- Index Definition in SQL

Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization (hashing)
- Is it desirable to optimize average access time at the expense of worst-case access time? (hashing)
- Expected type of queries:
  - Hashing is generally better at point queries.
  - If range queries are common, ordered indices are to be preferred
- In practice:
  - PostgreSQL supports hash indices, but discourages use due to poor performance
  - Oracle supports static hash organization, but not hash indices
  - SQLServer supports only B^+-trees

Chapter 11: Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B+-Tree Index Files
- Multiple-Key Access
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- **Bitmap Indices**
- Index Definition in SQL

Bitmap Indices

- designed for **efficient querying on multiple keys**
  - Bitmap for each attribute has as many bits as records
  - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute (0 otherwise)

record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Chapter 11: Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B+-Tree Index Files
- Multiple-Key Access
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Bitmap Indices
- **Index Definition in SQL**

Index Definition in SQL

- Create an index

  create index <index-name> **on** <relation-name>
  (<attribute-list>)

  E.g.: **create index** *b-index* **on** *branch(branch_name)*

- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported
- To drop an index

  drop index <index-name>

- Most database systems allow specification of type of index, and clustering.

Chapter 11: Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B^+-Tree Index Files
- Multiple-Key Access
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Bitmap Indices
- Index Definition in SQL