
UMBC

AN HONORS UNIVERSITY IN MARYLAND


CMSC 461, Database Management Systems
Spring 2018

# Lecture 18 – Indexing and Hashing Part 3

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

- Homework #4 due 4/9/2018
- Homework #5 due 4/18/2018
- Phase 4 due 4/23/2018

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

# $B^+$-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

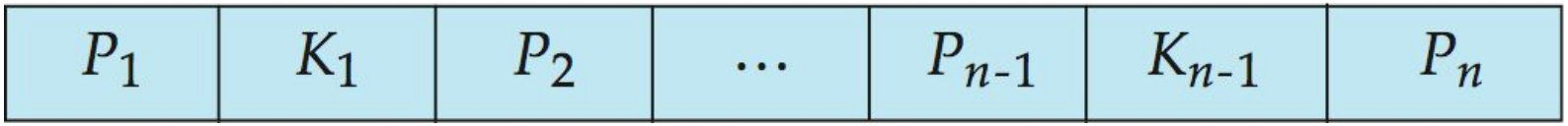
# $B^+$-Tree Index Files

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



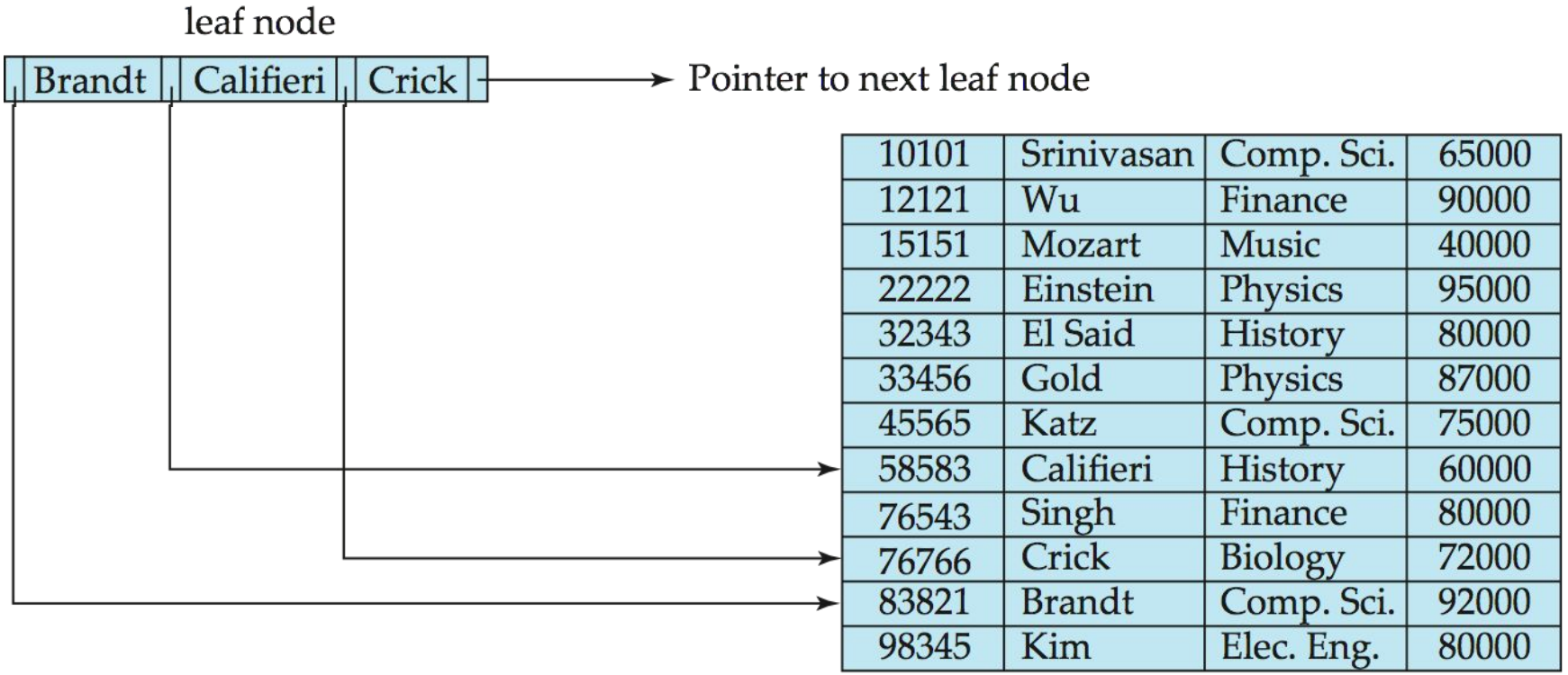

  - $K_i$ are the search-key values
  - $P_i$ are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

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

  (Initially assume no duplicate keys, address duplicates later)

# 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

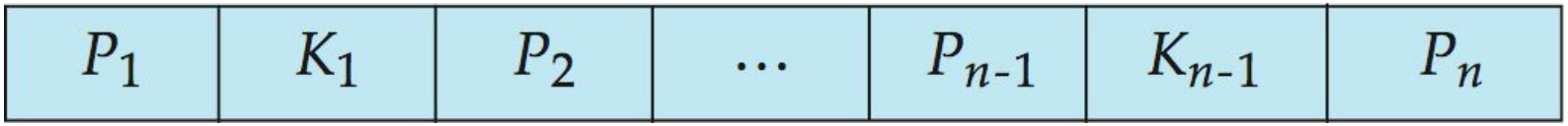



| | | | |
|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 12121 | Wu | Finance | 90000 |
| 15151 | Mozart | Music | 40000 |
| 22222 | Einstein | Physics | 95000 |
| 32343 | El Said | History | 80000 |
| 33456 | Gold | Physics | 87000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 58583 | Califieri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

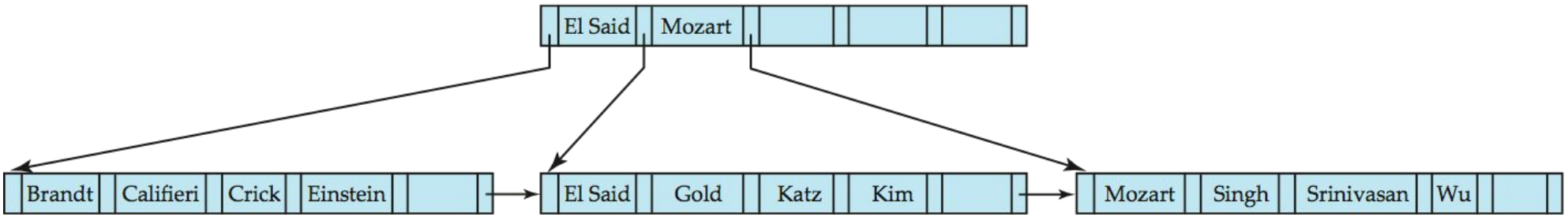
Based on and image from "Database System Concepts" book and slides, 6th edition

# 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 $2 \le i \le n - 1$, 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 $K_{n-1}$



# Example of B$^+$-tree




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

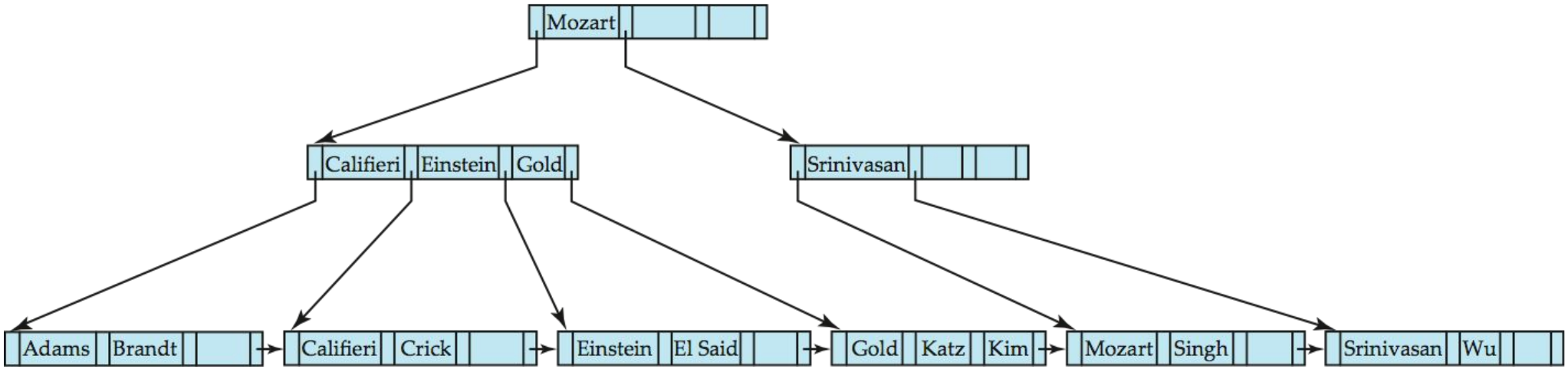
- Leaf nodes must have between 3 and 5 values ($\lceil(n-1)/2\rceil$ and $n-1$, with $n$ = 6).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil n/2\rceil$ and $n$ with $n$ =6).
- Root must have at least 2 children.

# Queries on B$^+$-Trees

Find record with search-key value *V.*
Start a C, compare V to $K_i$

While C is not a leaf node do the following:
if (V= $K_i$ ) Set C = $P_{i+1}$
else Set $C = P_i$
once $K_i = V$

1. If there is such a value *i,* follow pointer $P_i$ to the desired record.
2. Else no record with search-key value *k* exists.




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

# 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.

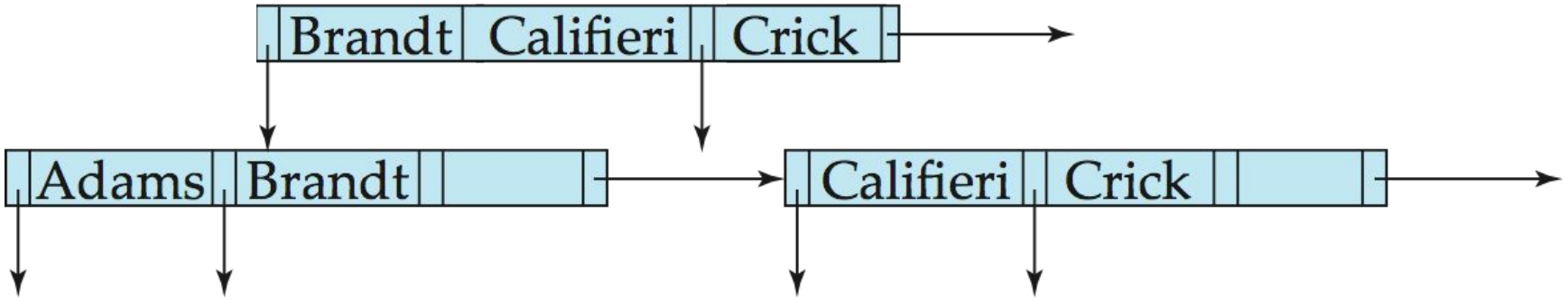
# Updates on B$^+$-Trees: Insertion

| *Leaf Full* | *Index Full* | *Action* |
|---|---|---|
| No | No | Place the record in sorted position in the appropriate leaf node |
| Yes | No | 1. Split the leaf<br>2. Place middle key in the index node in sorted order<br>3. Left leaf node contains records with keys below the middle key.<br>4. Right leaf node contains records with keys equal to or greater than the middle key. |
| Yes | Yes | 1. Split the leaf node.<br>2. Records with keys < middle key go to the left leaf node.<br>3. Records with keys >= middle key go to the right leaf node.<br>4. Split the index node.<br>5. Keys < middle key go to the left index node.<br>6. Keys > middle key go to the right index node.<br>7. The middle key goes to the next (higher level) index.<br>8. IF the next level index node is full, continue splitting the index node. |

Adaption: https://www.sci.unich.it/~acciaro/bpiutrees.pdf

# Updates on B$^+$-Trees: Insertion

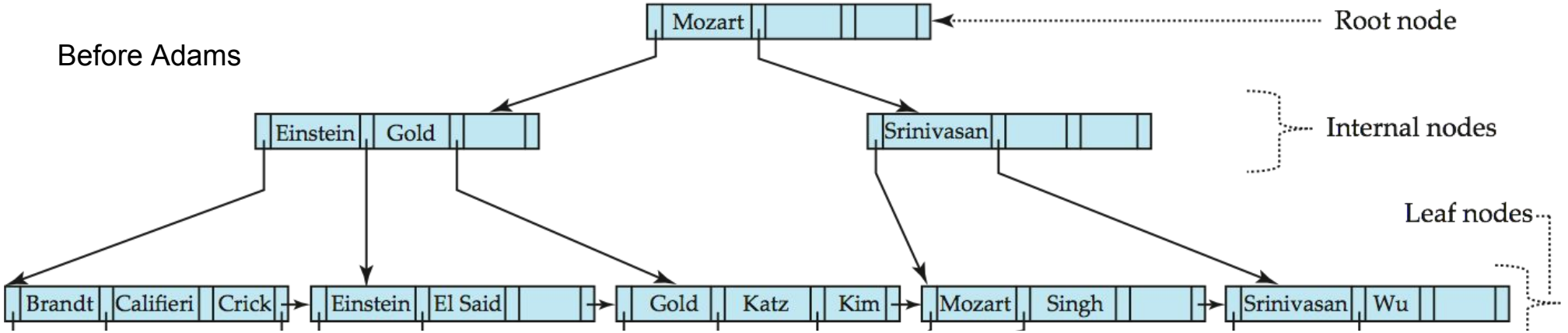
- Splitting a leaf node:
  - take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first [*n*/2] in the original node, and the rest in a new node.
  - let the new node be *p,* and let *k* be the least key value in *p.* Insert (*k,p*) in the parent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



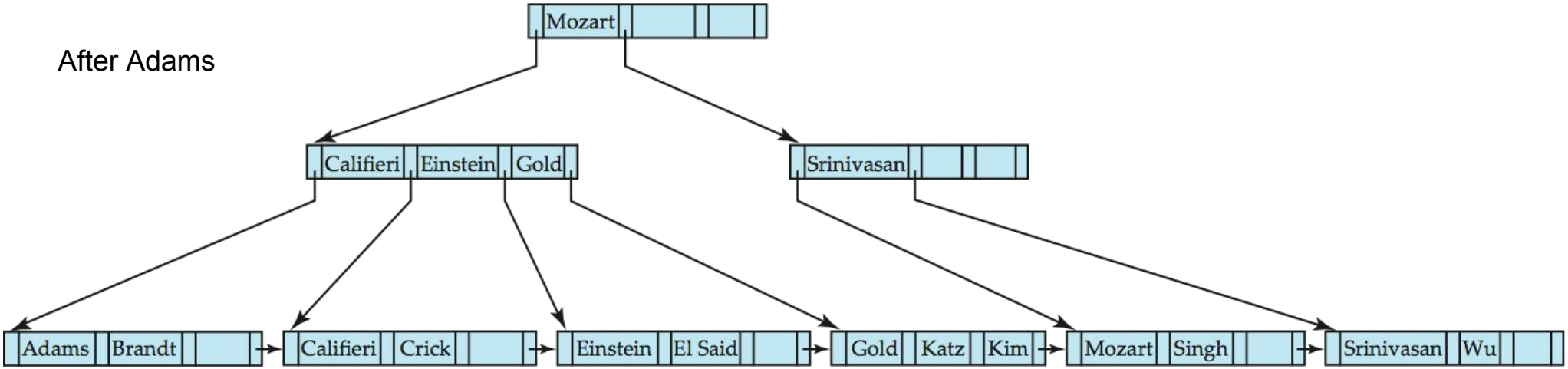

- Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
- Next step: insert entry with (Califieri,pointer-to-new-node) into parent

# B+-Trees Insertion

Before Adams




After Adams

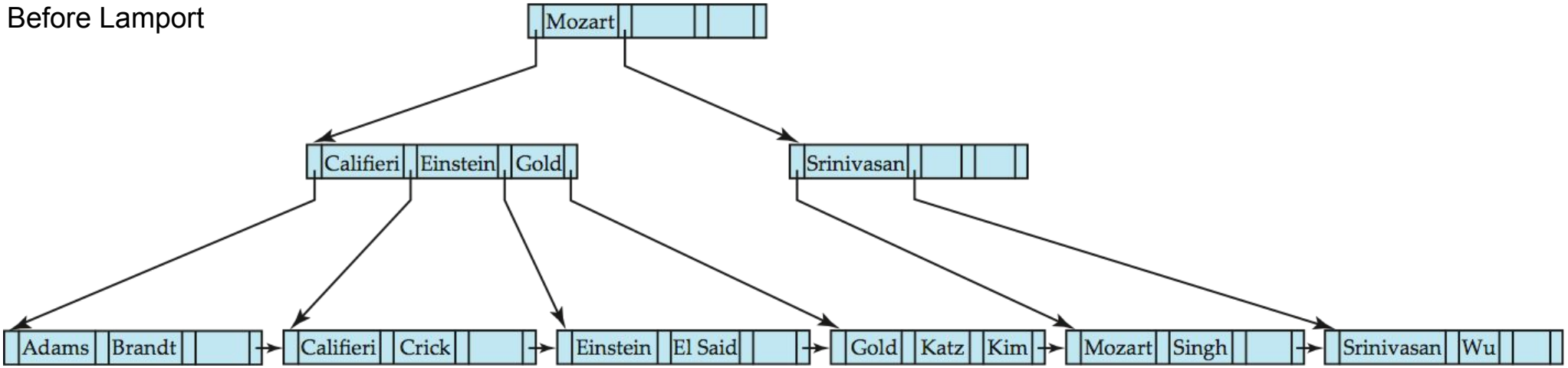



- B+-Tree before and after insertion of “Adams”

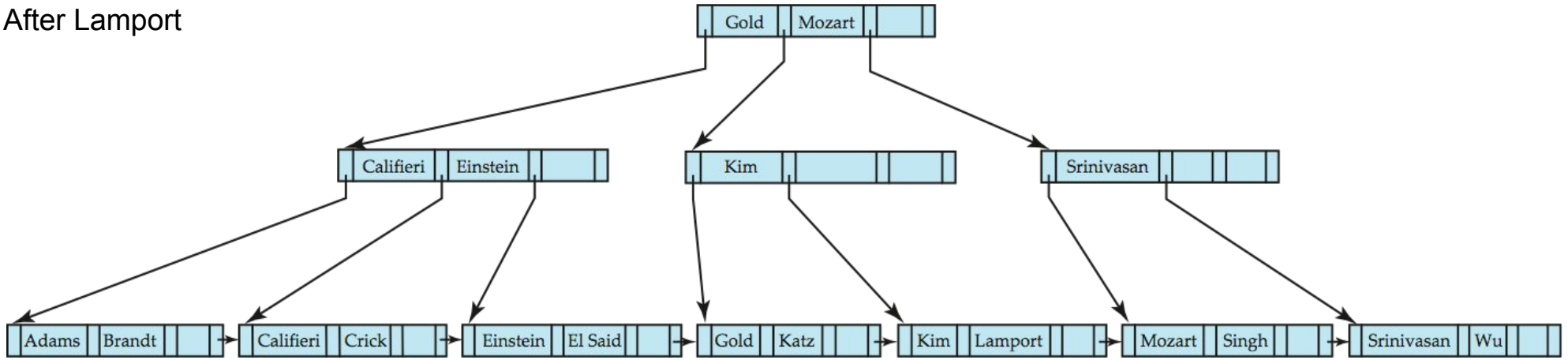
Based on and image from “Database System Concepts” book and slides, 6th edition

# B+-Trees Insertion

Before Lamport




After Lamport




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

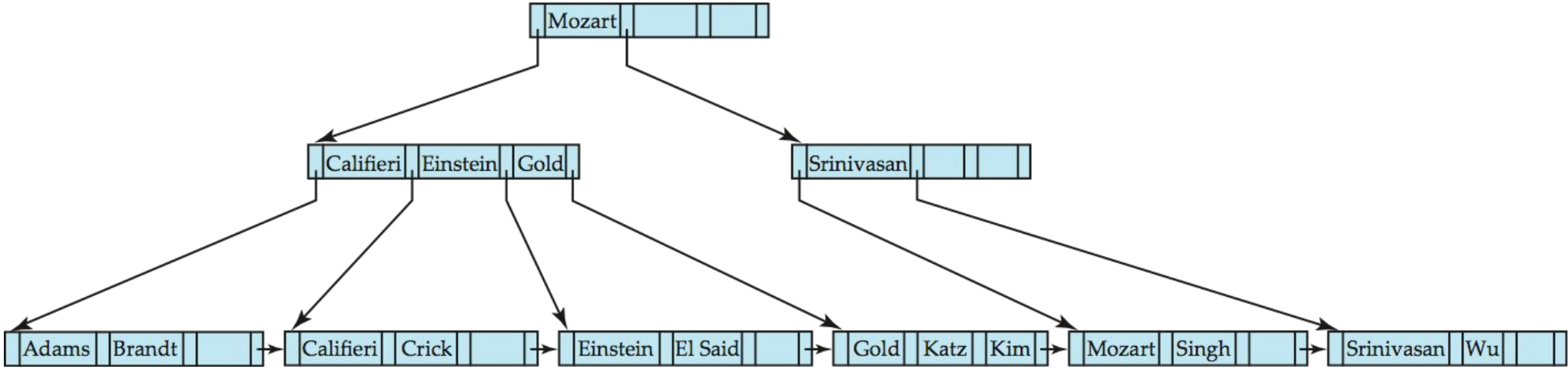
Based on and image from “Database System Concepts” book and slides, 6th edition

# Updates on B$^+$-Trees: Deletion

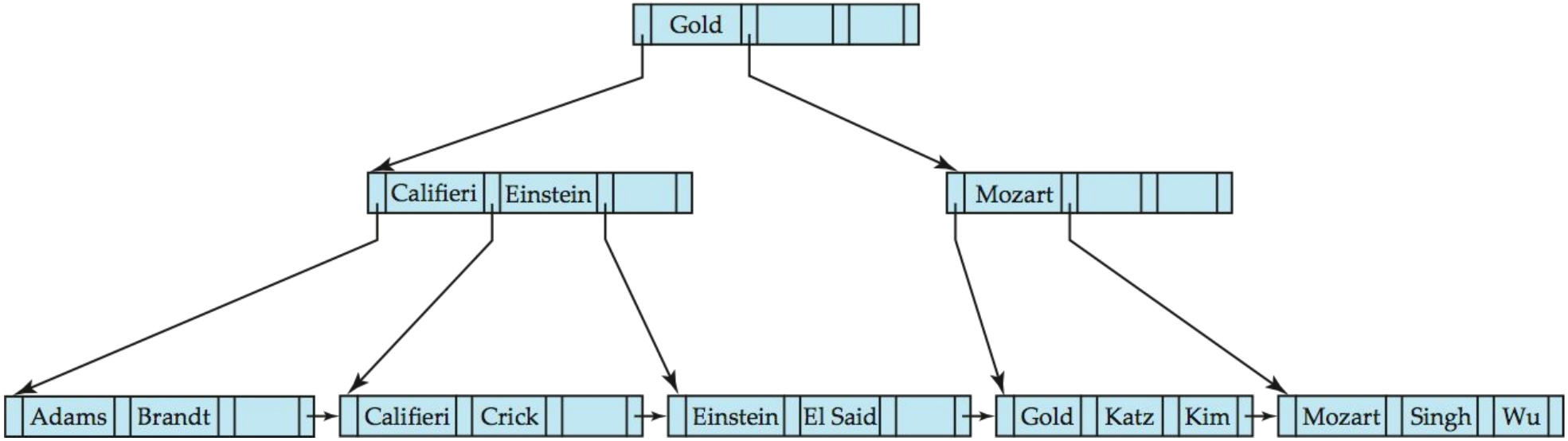
| *Leaf Node Below Fill Factor* | *Index Node Below Fill Factor* | *Action* |
|---|---|---|
| No | No | 1. Delete the record from the leaf node<br>2. Arrange keys in ascending order to fill void<br>3. If the key of the deleted record appears in the index node, use the next key to replace it. |
| Yes | No | 1. Combine the leaf node and its sibling<br>2. Change the index node to reflect the change |
| Yes | Yes | 1. Combine the leaf node and its sibling<br>2. Adjust the index node to reflect the change<br>3. Combine the index node with its sibling<br>4. Continue combining index node until you reach a node with the correct fill factor or you reach the root node |

Adaption: https://www.sci.unich.it/~acciaro/bpiutrees.pdf

# Example of B+-tree Deletion



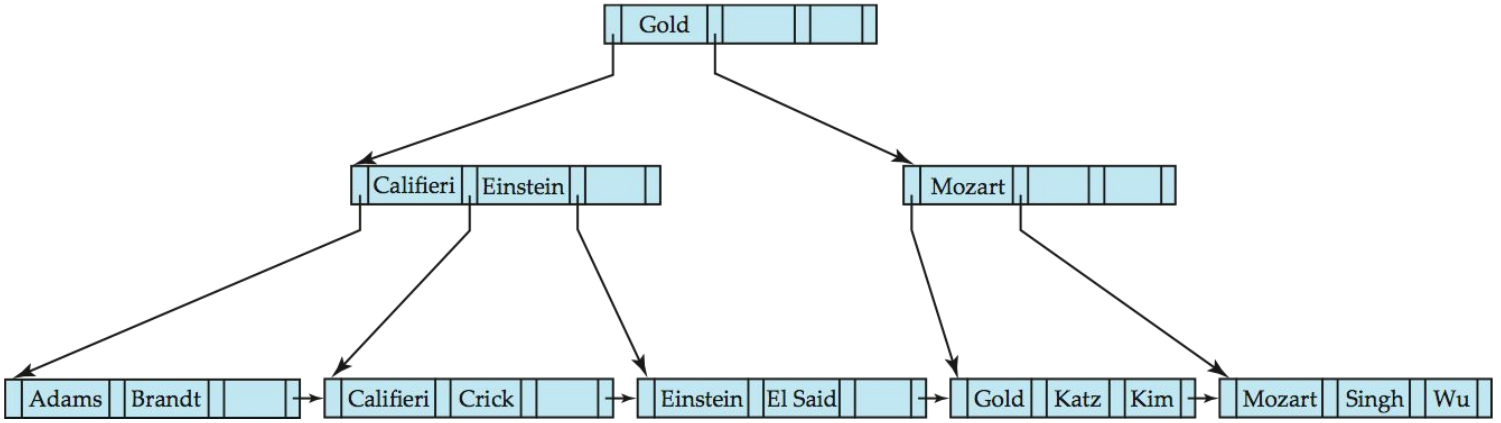

Before and after deleting “Srinivasan”

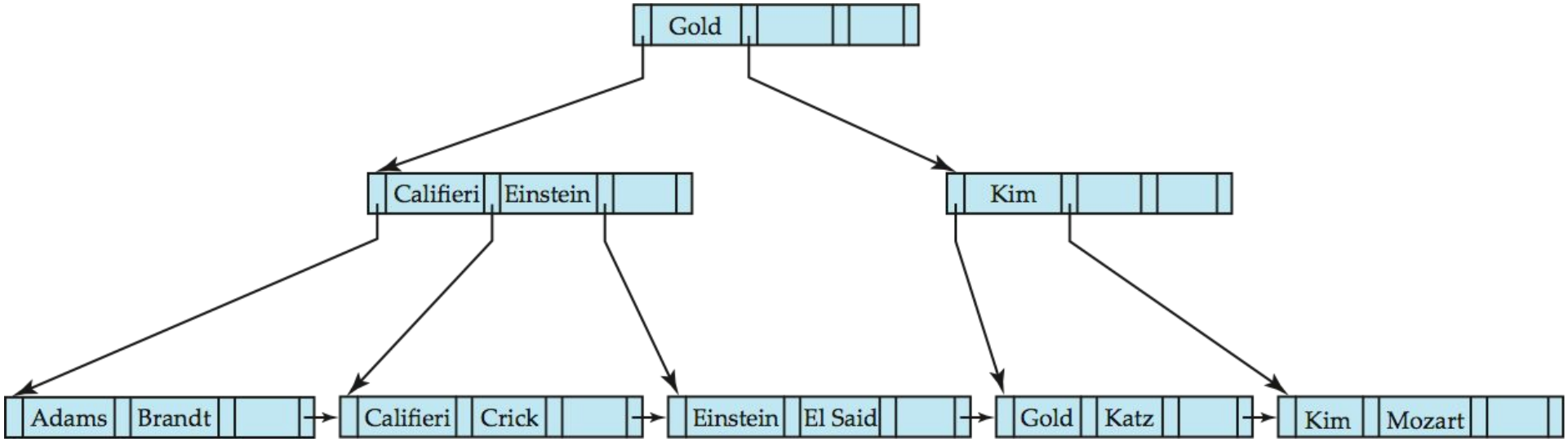



Deleting “Srinivasan” causes merging of under-full leaves

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

# Hashing

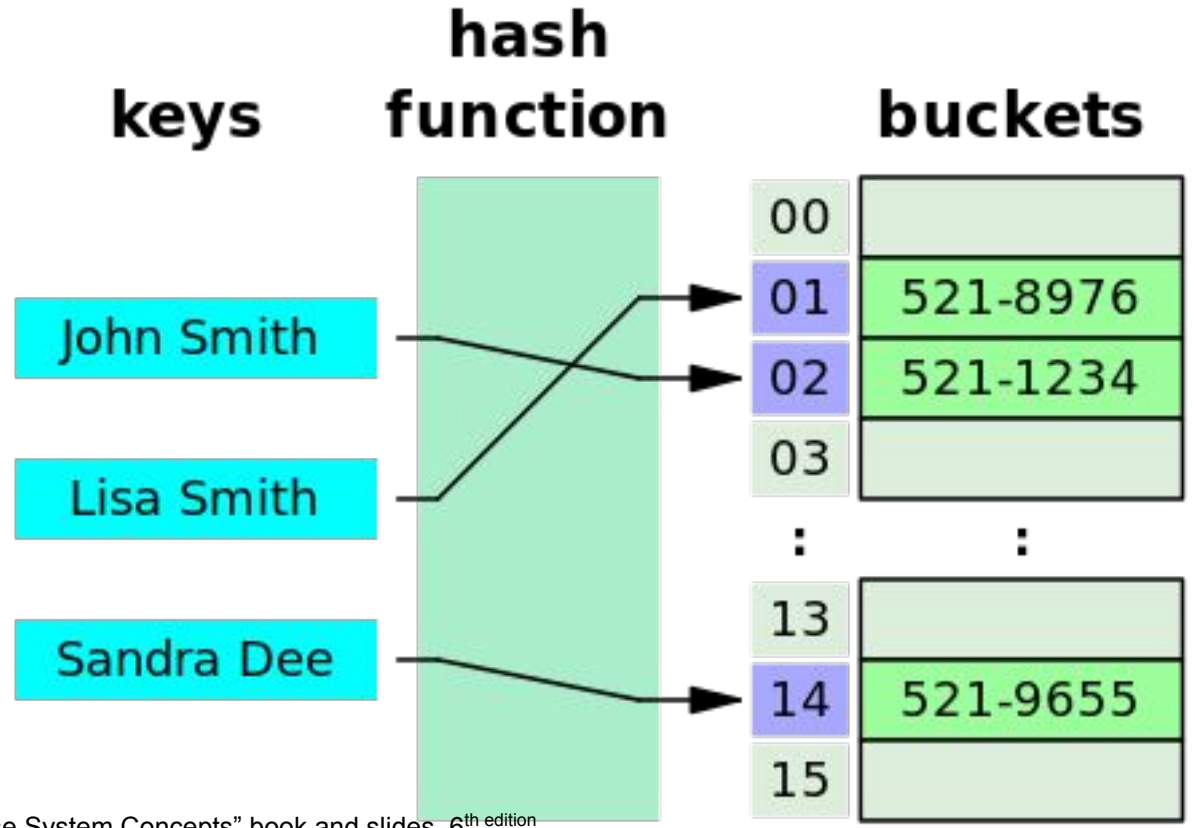
# Static Hashing

- 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**.

# Static Hashing

- Hash function ***h*** is a function from the set of all search-key values ***K*** to the set of all bucket addresses *B.*

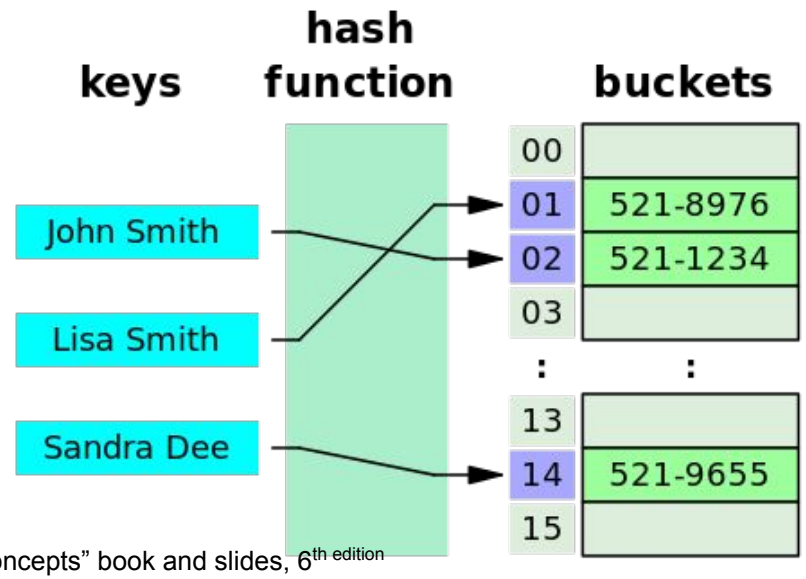
*h($K_i$)=address of the bucket where record is stored*




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

# Static Hashing

- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket
- Entire bucket has to be searched sequentially to locate a record




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

# Hash Functions

- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.

# Hash Functions

- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file

AND...should be easy/fast to compute

# Hash Functions

- Typical hash functions perform computation on the internal binary representation of the search-key.

# 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

# Example of Hash File Organization

- There are 8 buckets,
- The binary representation of the *i*th character is assumed to be the integer *i*.
- The hash function returns the sum of the binary representations of the characters modulo 8

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

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |

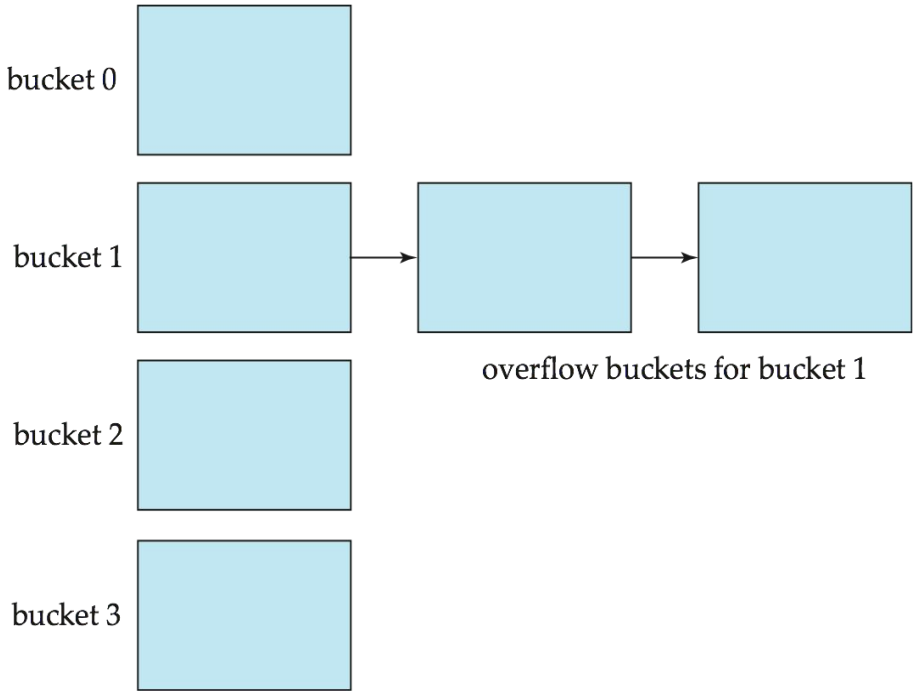
h(Music) = 1 h(History) = 2
h(Physics) = 3 h(Elec. Eng.) = 3

# Handling of Bucket Overflows

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using ***overflow buckets***.

# Handling of Bucket Overflows

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed hashing**.
  - An alternative, called **open hashing**, which does not use overflow buckets, is not suitable for database applications.

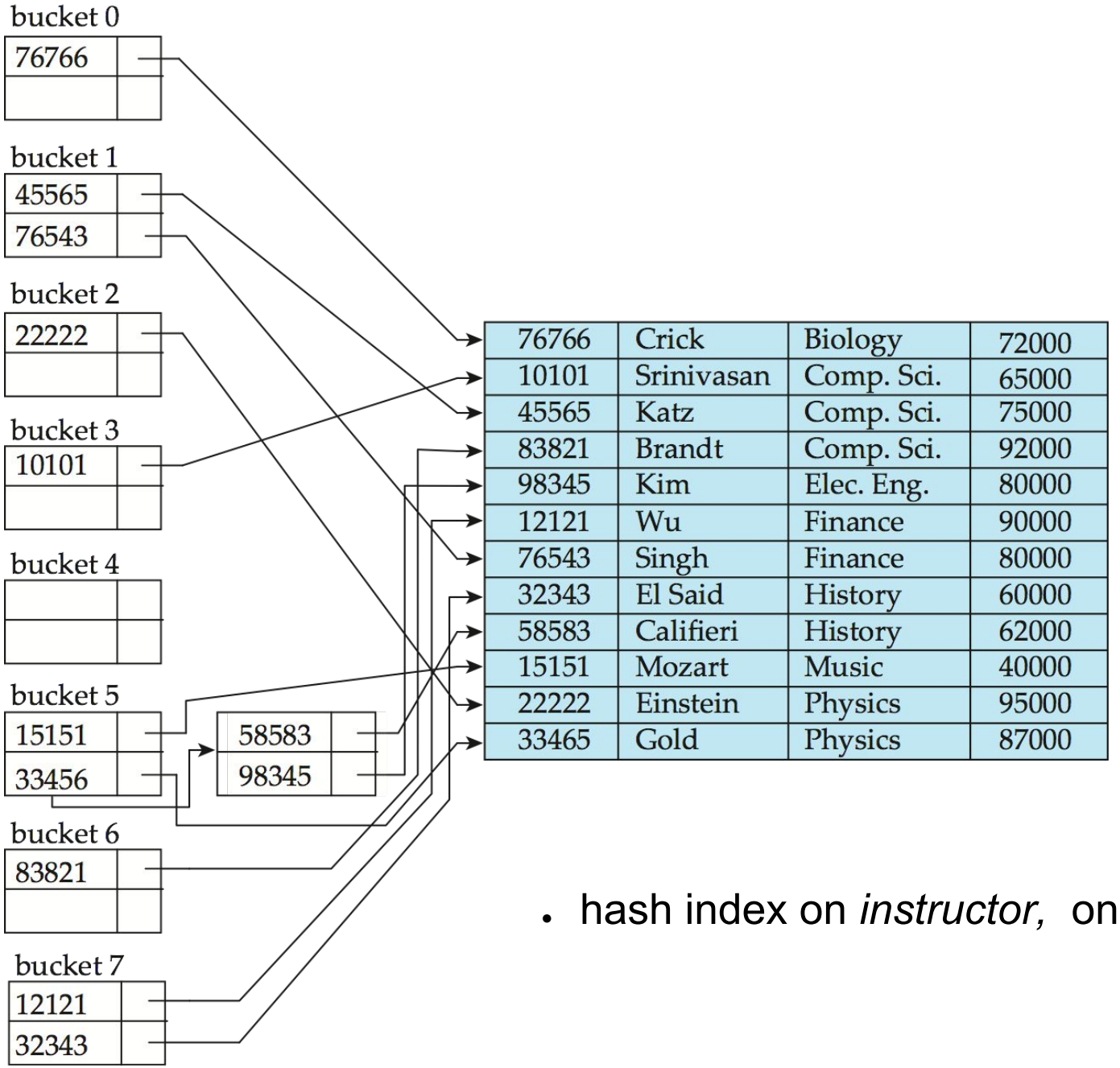



Based on and image from "Database System Concepts" book and slides, 6th edition

# Hash Indices

- Hashing can also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indices are always secondary indices
  - if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
  - However, we use the term hash index to refer to both secondary index structures and hash organized files.

# Example of Hash Index




- hash index on *instructor,* on attribute *ID*

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

# Static Hashing

What do you think?

# Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses. Databases grow or shrink with time.

# Deficiencies of Static Hashing

# Deficiencies of Static Hashing

# Deficiencies of Static Hashing

- One solution: periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically

# Dynamic Hashing

# 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

# Extendable Hashing

- Hash function generates values over a large range — typically *b*-bit integers, with *b* = 32.
- At any time use only a prefix of the hash function to index into a table of bucket addresses

# Extendable Hashing

Let the length of the prefix be *i* bits,
$0 \leq i \leq 32$

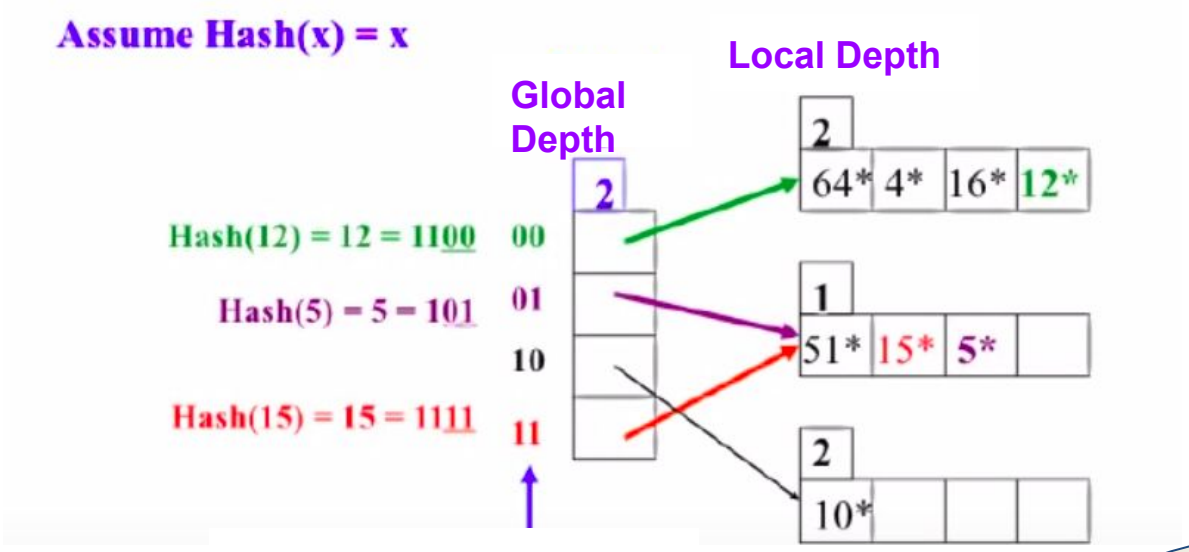
Bucket address table size = $2^i$.

Initially $i = 0$

Value of *i* grows and shrinks as the size of the database grows and shrinks.

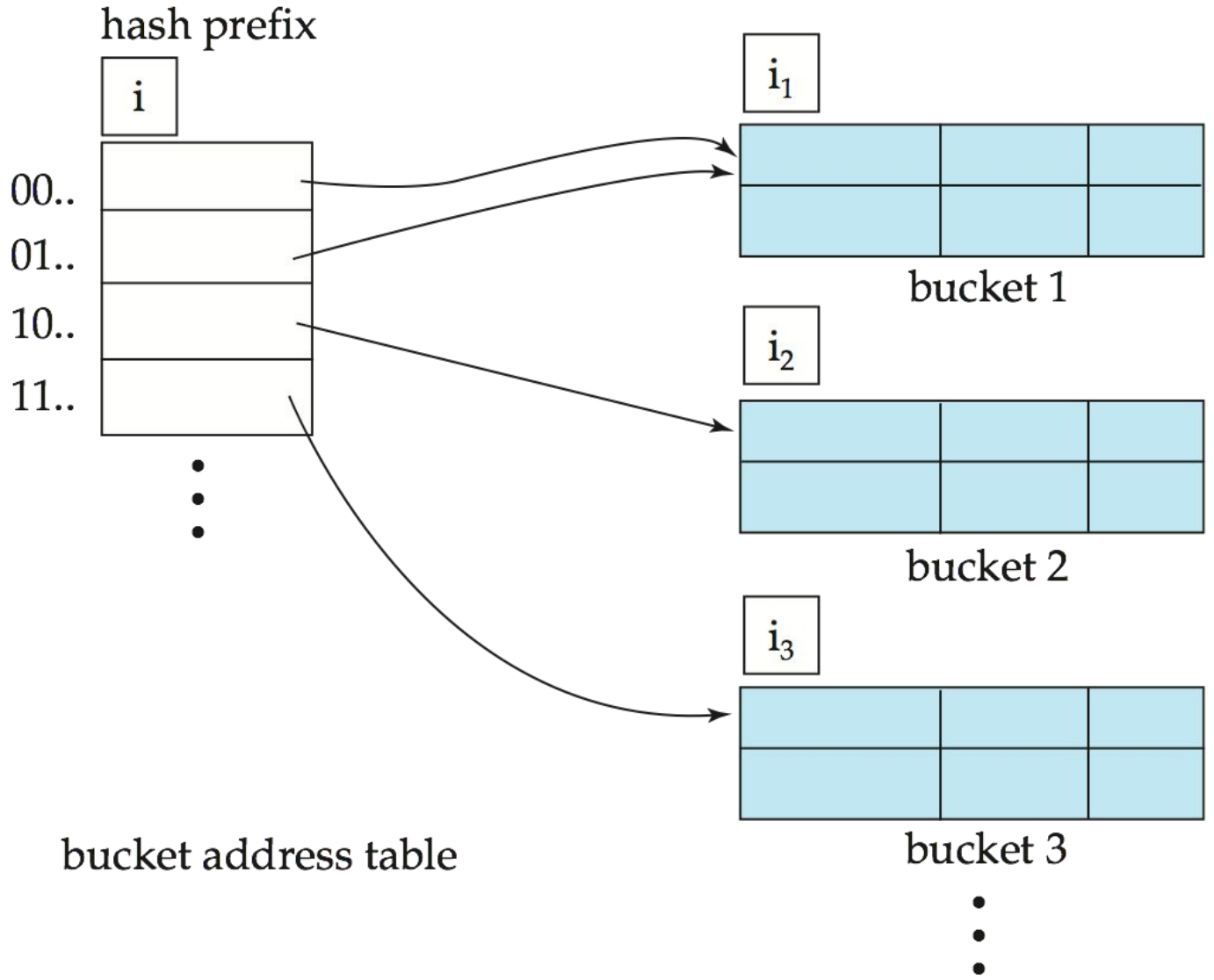
# Extendable Hashing

- Multiple entries in the bucket address table may point to a bucket
- Actual number of buckets is < $2^i$
- The number of buckets changes dynamically due to coalescing and splitting of buckets




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

# General Extendable Hash Structure




In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$

# Use of Extendable Hash Structure

- Each bucket *j* stores a value $i_j$
  - All the entries that point to the same bucket have the same values on the first $i_j$ bits.
- To locate the bucket containing search-key $K_j$:
  1. Compute $h(K_j) = X$
  2. Use the first *i* high order bits of *X* as a displacement into bucket address table, and follow the pointer to appropriate bucket

# Insertion in Extendable Hash Structure

To insert a record with search-key value $K_j$

- Follow same procedure as look-up and locate the bucket, say *j*.
- If there is room in the bucket *j* insert record in the bucket.
- Else the bucket must be split and insertion re-attempted

# Insertion in Extendable Hash Structure

If $i > i_j$ (more than one pointer to bucket $j$)

- Allocate a new bucket $z$, and set $i_j = i_z = (i_j + 1)$
- Update the second half of the bucket address table entries originally pointing to $j$, to point to $z$
- Remove each record in bucket $j$ and reinsert (in $j$ or $z$)
- Recompute new bucket for $K_j$ and insert record in the bucket (further splitting is required if the bucket is still full)

# Insertion in Extendable Hash Structure

If $i = i_j$ (only one pointer to bucket $j$)

If $i$ reaches some limit $b$, or too many splits have happened in this insertion, create an overflow bucket

Else

- Increment $i$ and double the size of the bucket address table.
- Replace each entry in the table by two entries that point to the same bucket.
- Recompute new bucket address table entry for $K_j$
  Now $i > i_j$ so use the first case above.

# Deletion in Extendable Hash Structure

To delete a key value,

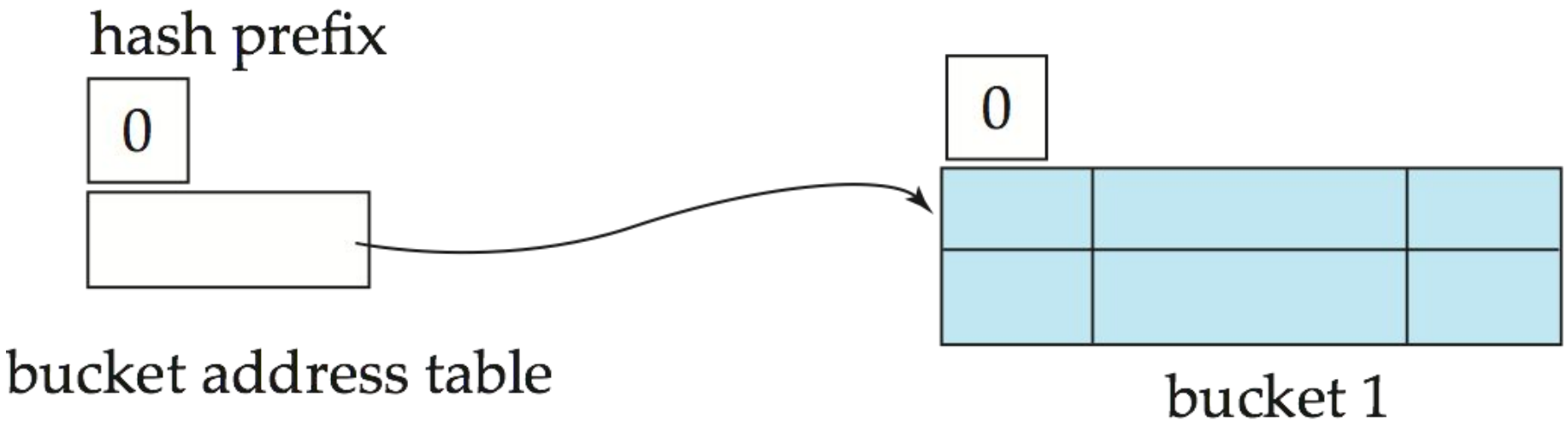
- Locate it in its bucket and remove it.
- The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
- Coalescing of buckets can be done (can coalesce only with a "*buddy*" bucket having same value of $i_j$ and same $i_j$ –1 prefix, if it is present)
- Decreasing bucket address table size is also possible
  - Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

# Use of Extendable Hash Structure: Example

| *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.)

Initial Hash structure; bucket size = 2

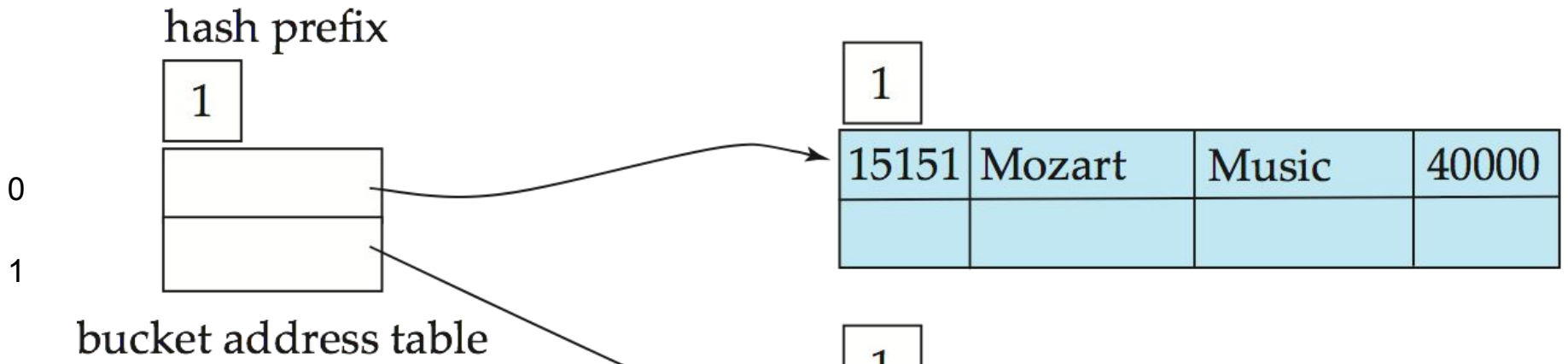



| *dept_name* | h(*dept_name*) |
|---|---|
| Biology | 0010 1101 1111 1011 0010 1100 0011 0000 |
| Comp. Sci. | 1111 0001 0010 0100 1001 0011 0110 1101 |
| Elec. Eng. | 0100 0011 1010 1100 1100 0110 1101 1111 |
| Finance | 1010 0011 1010 0000 1100 0110 1001 1111 |
| History | 1100 0111 1110 1101 1011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

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

# Example (Cont.)

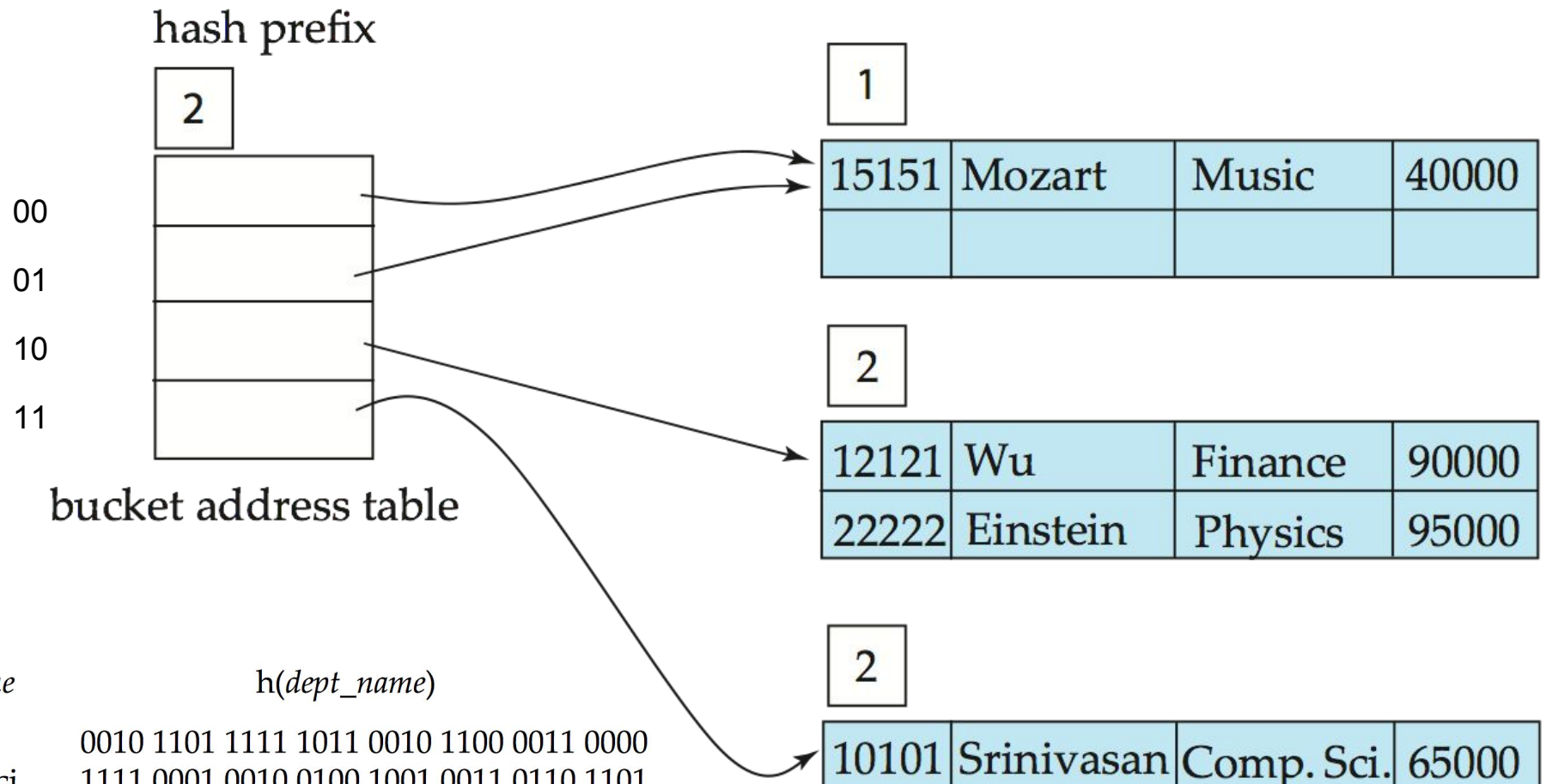
Hash structure after insertion of “Mozart”, “Srinivasan”, and “Wu” records




| *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.)

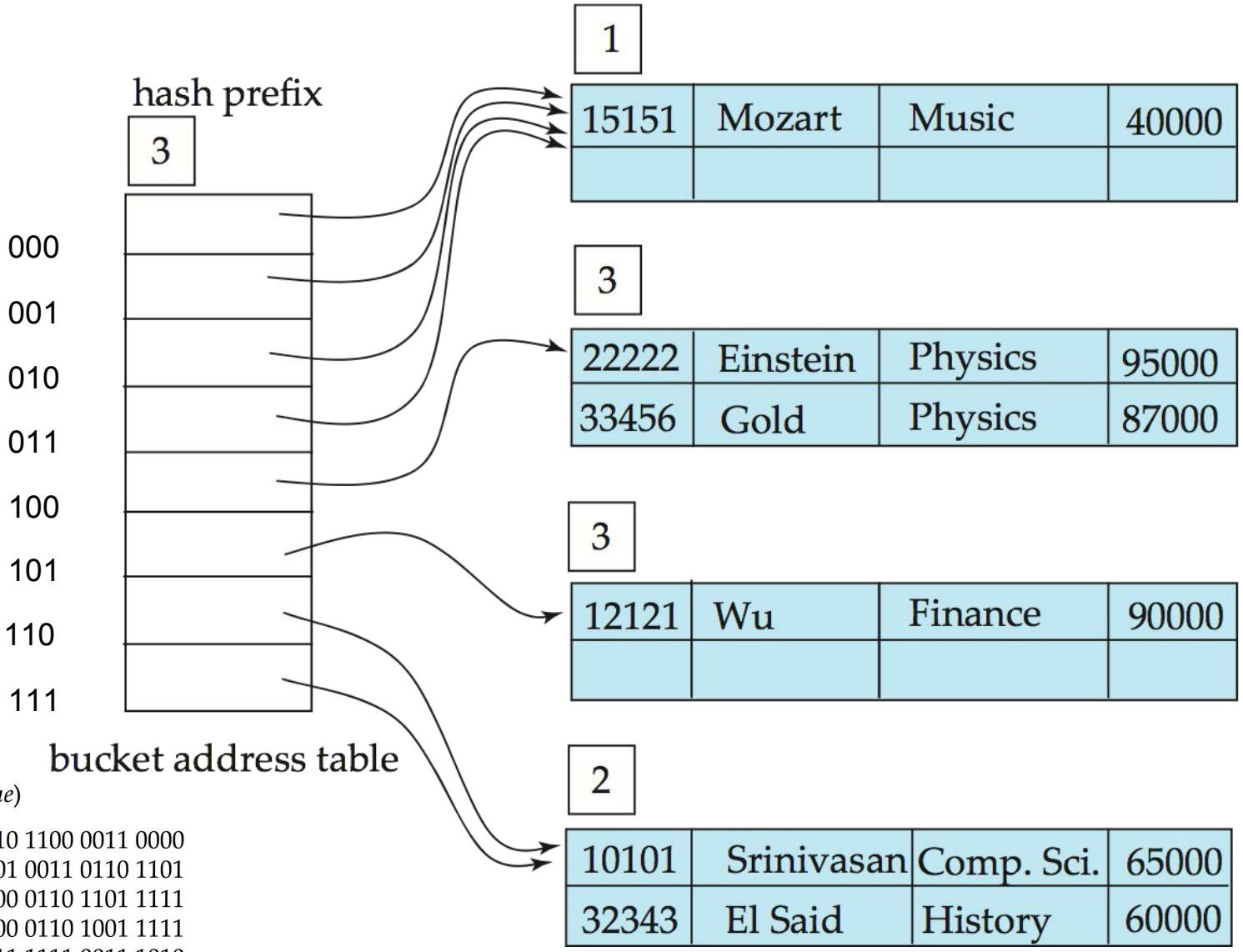
Hash structure after insertion of Einstein record




| *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 Gold and El Said records

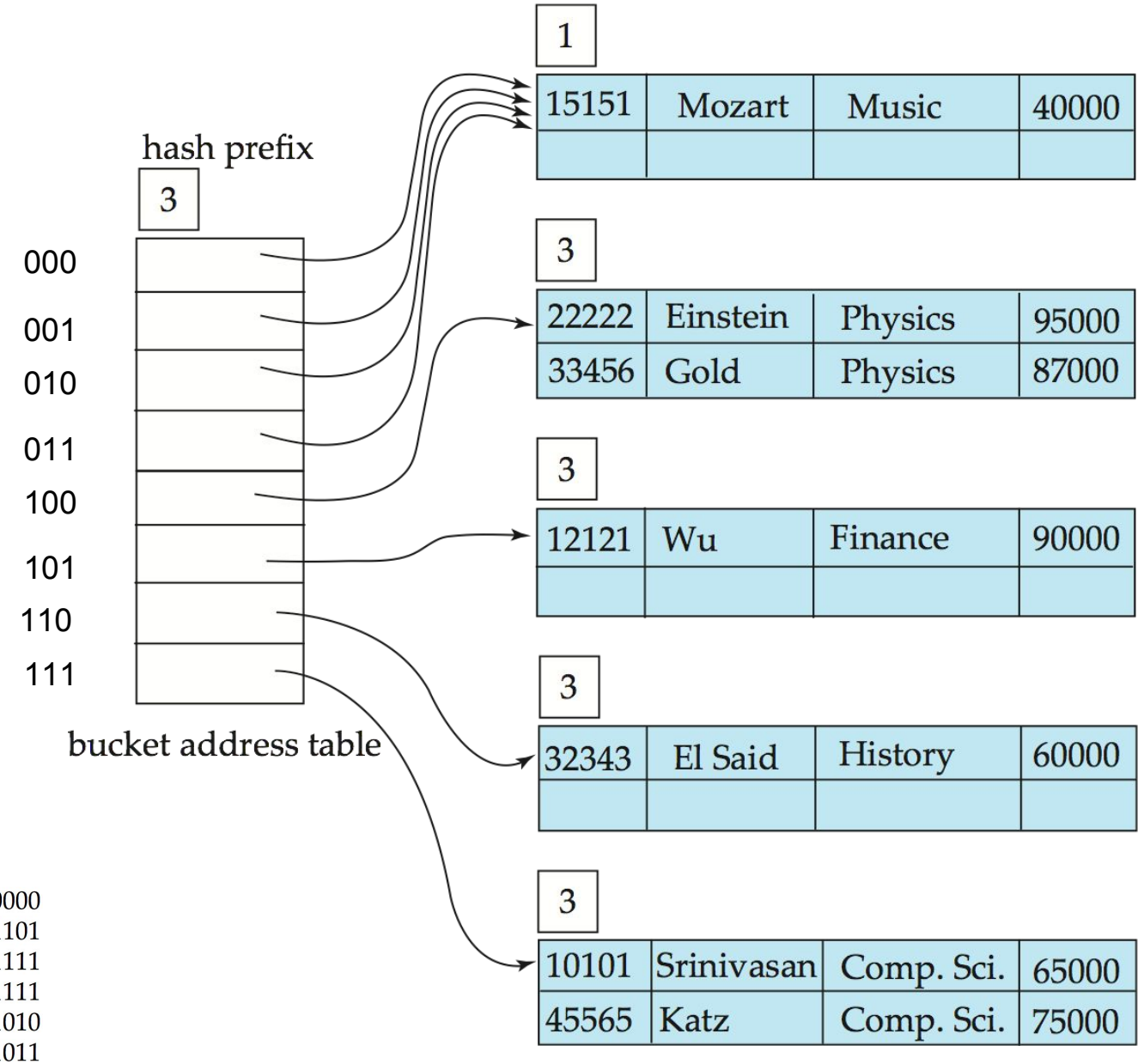



| *dept_name* | h(*dept_name*) |
|---|---|
| Biology | 0010 1101 1111 1011 0010 1100 0011 0000 |
| Comp. Sci. | 1111 0001 0010 0100 1001 0011 0110 1101 |
| Elec. Eng. | 0100 0011 1010 1100 1100 0110 1101 1111 |
| Finance | 1010 0011 1010 0000 1100 0110 1001 1111 |
| History | 1100 0111 1110 1101 1011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

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

# Example (Cont.)

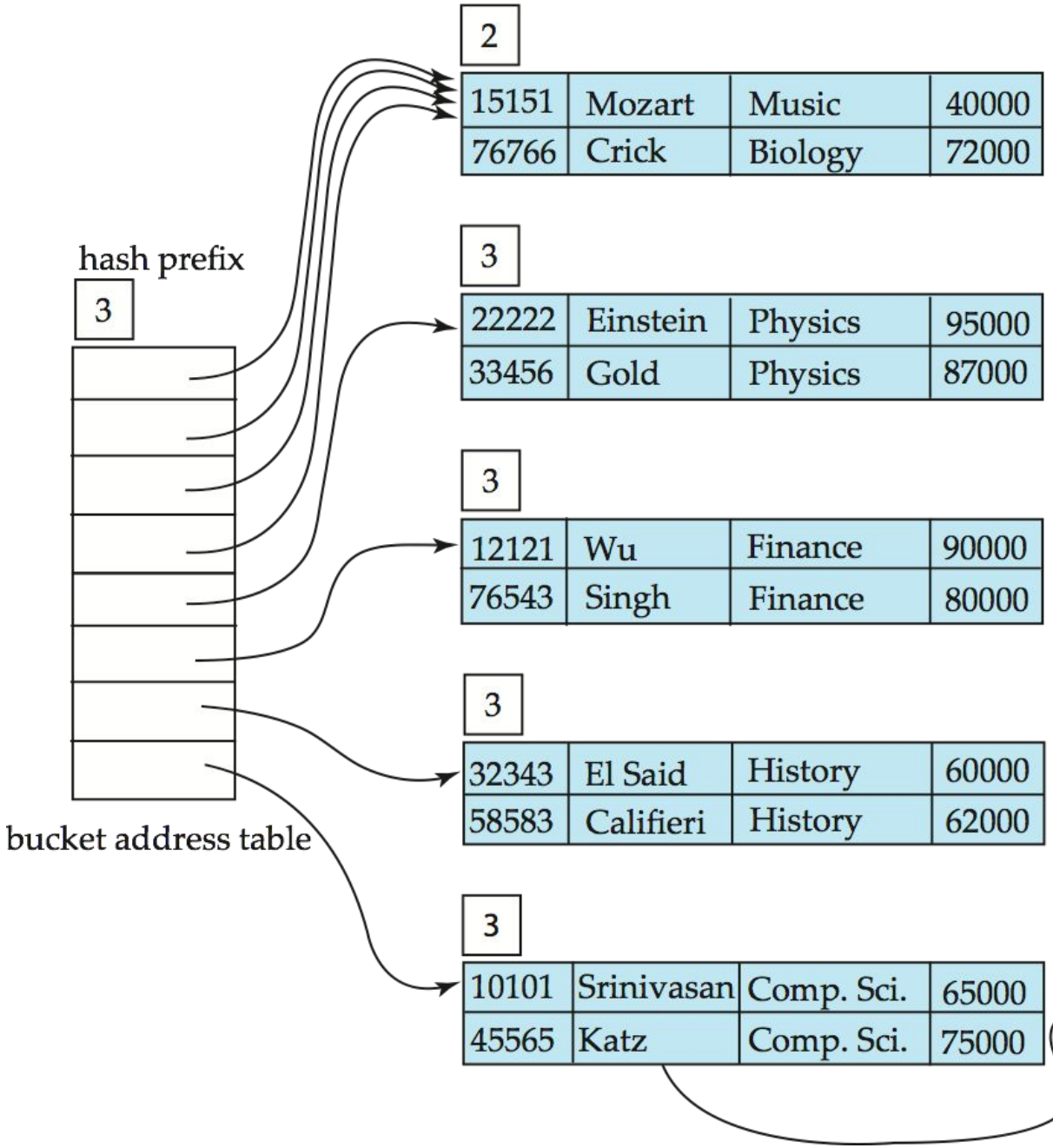
Hash structure after insertion of Katz record




| *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 |

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

# Example (Cont.)

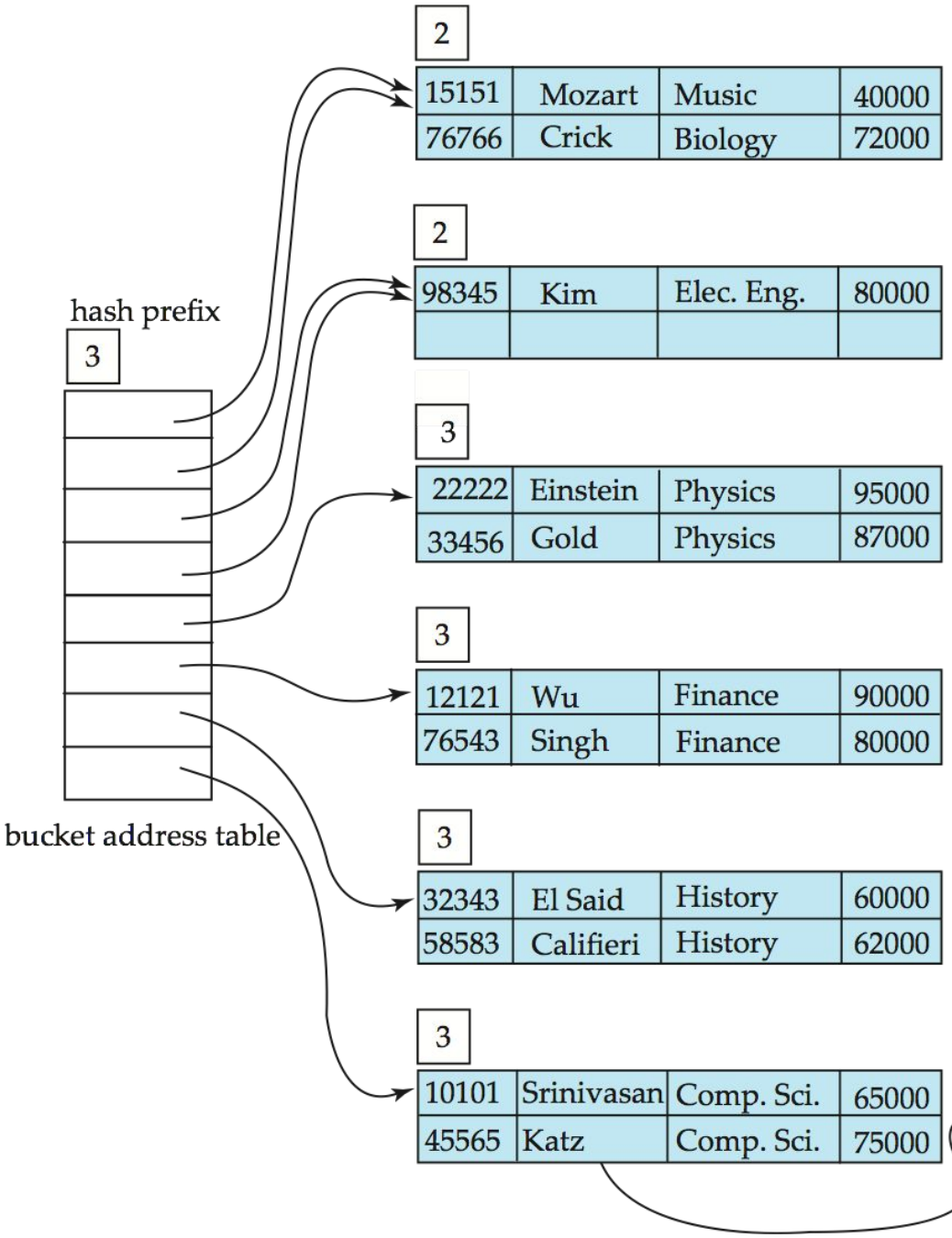



And after insertion of eleven records

| *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 |

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

# Example (Cont.)




And after insertion of Kim record in previous hash structure

| *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 |

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

# Extendable Hashing vs. Other Schemes

- Benefits of extendable hashing:
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- Disadvantages of extendable hashing
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big (larger than memory)
    - Cannot allocate very large contiguous areas on disk either
    - Solution: $B^+$-tree structure to locate desired record in bucket address table
  - Changing size of bucket address table is an expensive operation
- Linear hashing is an alternative mechanism
  - Allows incremental growth of its directory (equivalent to bucket address table)
  - At the cost of more bucket overflows

# Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries

# Comparison of Ordered Indexing and Hashing

Expected type of queries:

- Hashing is generally better at retrieving records having a specified value of the key.
- If range queries are common, ordered indices are to be preferred

In practice:

- PostgreSQL supports hash indices, but discourages use due to poor performance
- MySQL supports hash indices
- Oracle supports static hash organization, but not hash indices
- SQLServer supports only $B^+$-trees

# Bitmap Indices

# Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from, say, 0
  - Given a number *n* it must be easy to retrieve record *n*
    - Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
  - E.g. gender, country, state, …
  - E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- A bitmap is simply an array of bits

# Bitmap Indices (Cont.)

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute
  - Bitmap 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, and is 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 |

# Bitmap Indices (Cont.)

- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
  - Intersection (and)
  - Union (or)
  - Complementation (not)

# Bitmap Indices (Cont.)

- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
  - E.g. 100110 AND 110011 = 100010
    100110 OR 110011 = 110111
    NOT 100110 = 011001
  - Males with income level L1: 10010 AND 10100 = 10000
    - Can then retrieve required tuples.
    - Counting number of matching tuples is even faster

# 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)*

# Index Definition in SQL

- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate 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.