Indexing

Database System Concepts

Indexing

- **Review of Basic Concepts and Ordered Indices**
- B^+-Tree Index Files
- Static Hashing
- Dynamic Hashing
- Multiple-Key Access
- Creation of Indices

A Few Questions

- Do you know that time taken to access some data in disk >> time taken to access some data in main memory?
- Do you know that in accessing some data in disk, the whole disk block containing the required data has to be brought from disk into main memory?
  - Blocks are units of both storage allocation and data transfer.

  - Disk block read requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access

- If you want to access some data in a database, is it a good idea to read ***every*** record in the database to search for the desired data?

  - What if the database is so small that it can be stored in main memory?
  - What if the database is so large that it must be stored in disk?
- If the records are sorted in the database, do you know any good searching algorithms to reduce the search time? (binary search)
  - If the database occupies 1,000,000 blocks, how many blocks have to be read by using binary search for the desired data? ($\lceil \log_2(1{,}000{,}000) \rceil = 20$)
  - How much time it takes, if a block read takes 10 ms? (0.2 sec, is it a long time to you?)

Review of Basic Concepts

- It is ***inefficient*** to read ***every*** record in a (large) database to search for desired data.
- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library

- **Search Key** - attribute or set of attributes used to look up records in a file.

  - A **sequential file** stores records in sequential order, based on the value of the search key of each record.

- An **in**rds (called **index entries**) of the form

search-key	pointer

  Index files are typically much ***smaller*** than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

Index Evaluation Metrics

NO one technique is the best.

- Access types
  - records with a specified value in the attribute
  - records with an attribute value falling in a specified range of values.
- Access time
  - time to find a data item
- Insertion time
  - time to find the correct insertion point
  - time to update the index structure
- Deletion time
  - time to find the item to be deleted
  - time to update the index structure
- Space overhead occupied by the index structure

Ordered Indices

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

Dense Index Files

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

10101	
12121	
15151	
22222	
32343	
33456	
45565	
58583	
76543	
76766	
83821	
98345	

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	

Dense Index Files (Cont.)

- An index on *dept_name*, with *instructor* file sorted on *dept_name*

Biology	
Comp. Sci.	
Elec. Eng.	
Finance	
History	
Music	
Physics	

76766	Crick	Biology	72000	
10101	Srinivasan	Comp. Sci.	65000	
45565	Katz	Comp. Sci.	75000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	
12121	Wu	Finance	90000	
76543	Singh	Finance	80000	
32343	El Said	History	60000	
58583	Califieri	History	62000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
33465	Gold	Physics	87000	

Is it a **primary index**?

Is it a **dense primary index**?

Sparse Index Files

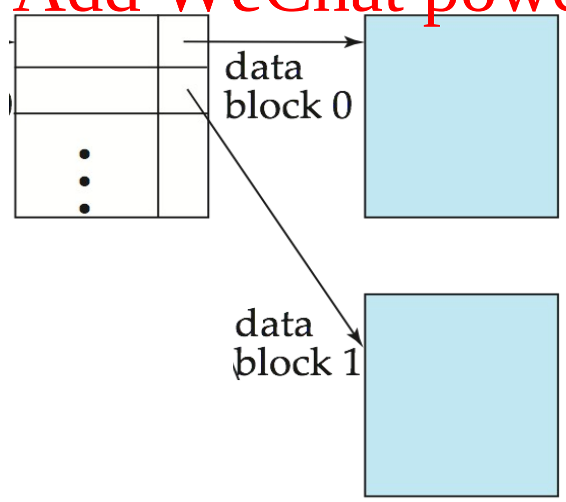
- **Sparse Index**: contains index records for only ***some*** search-key values.
  - Applicable ***only if*** the index is a primary index (why?)
- To locate a record with search-key value *K*,
  - Find index record with largest search-key value ≤ *K*
  - Search file sequentially starting at the record 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.)

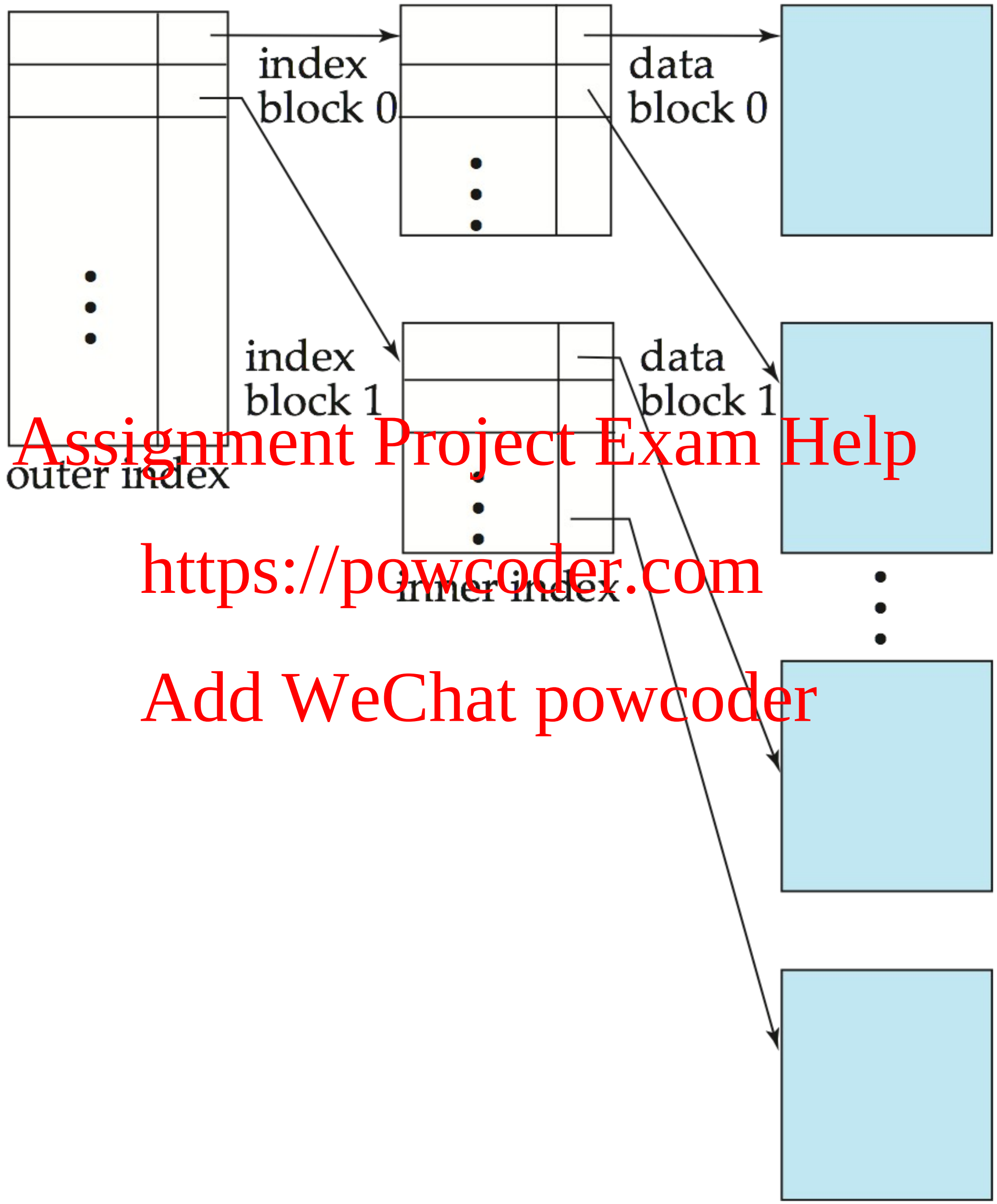
- Compared to dense indices:
  - 👍 Less space and less maintenance overhead for insertions and deletions.
  - 👎 Generally slower than dense index for locating records.
- Good tradeoff: ***one index entry per block*** (sparse index with an index entry for every block in file, corresponding to least search-key value in the block)
  - dominant cost: time to bring a block from disk into main memory
  - minimize block accesses while keeping index size small



Multilevel Index

- If primary index does not fit in memory, access becomes expensive, due to ***disk block reads***.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

Multilevel Index (Cont.)



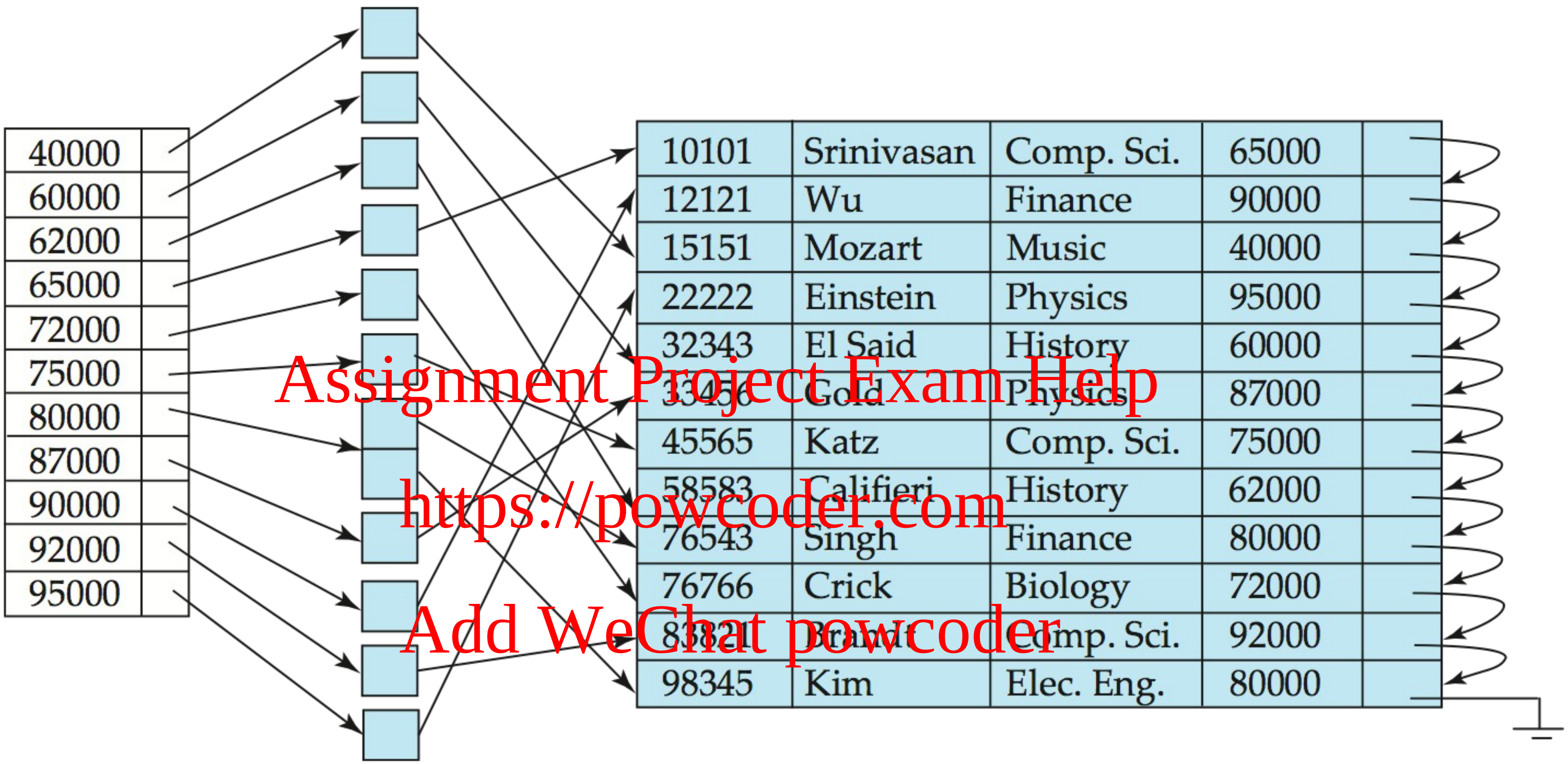
Secondary Indices

- *Reminder*: **Secondary index** is an index whose search key specifies an order ***different*** from the sequential order of the file.

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

- We can have a secondary index with an index record for ***every*** search-key value, and a pointer to ***every*** record in the file, i.e., must be ***dense***.

Secondary Indices Example



Secondary index on *salary* field of *instructor*

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

Primary and Secondary Indices

- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive on magnetic disk
  - Each record access may fetch a new block from disk

- Indices offer substantial benefits when searching for records.

- BUT: Updating indices imposes overhead on database modification

  - when a record is inserted or deleted, ***every*** index on the relation must be updated
  - When a record is updated, any index on an updated attribute must be updated

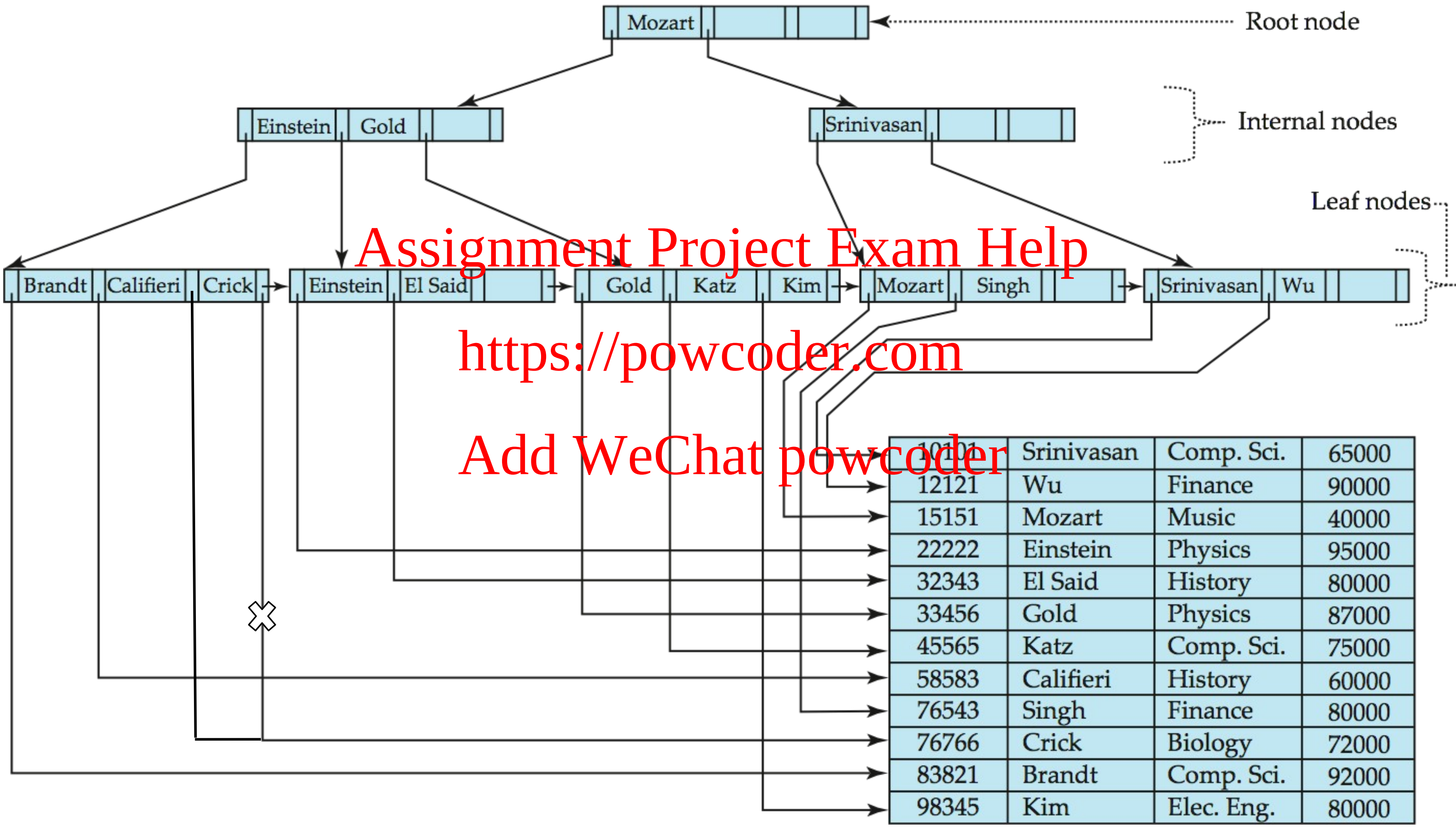
Indexing

- Review of Basic Concepts and Ordered Indices
- **B+-Tree Index Files**
- Static Hashing
- Dynamic Hashing
- Multiple-Key Access
- Creation of Indices

B^+-Tree Index Files

- Disadvantage of index-sequential files
  - Performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required to restore sequential order.
- 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 ***balanced*** tree satisfying the following properties:

- All paths from **root** to **leaf** are of the same length
- Each **internal node** (not a root or a leaf) has between $\lceil n/2 \rceil$ and n children, where n is fixed for a particular tree
- 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	$\dots$	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 (for leaf nodes).

- The search-keys in a node are ordered

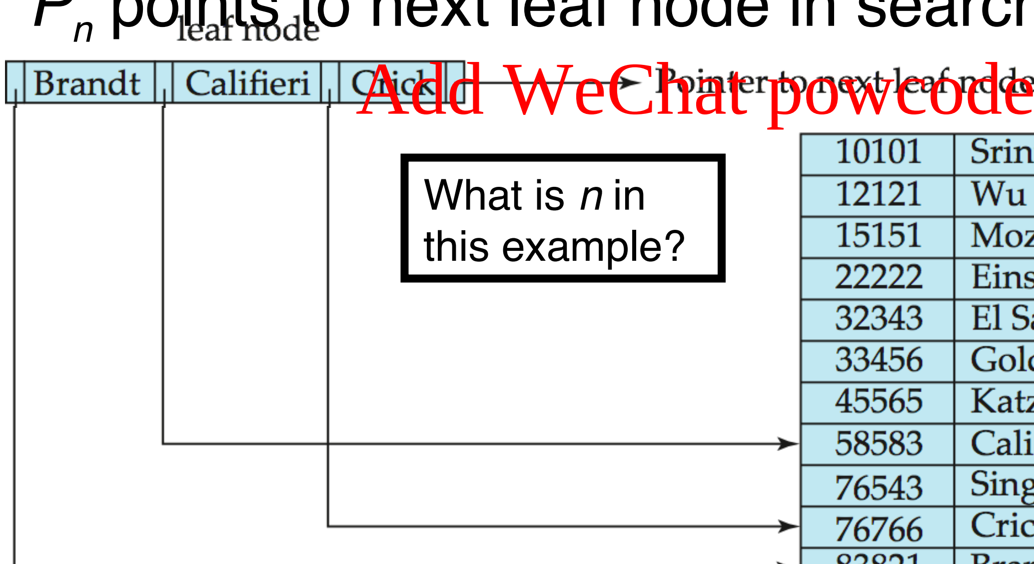
$$K_1 < K_2 < K_3 < \dots < K_{n-1}$$

(assume no duplicate keys)

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 L_j's search-key values
- P_n points to next leaf node in search-key order



What is n in this example?

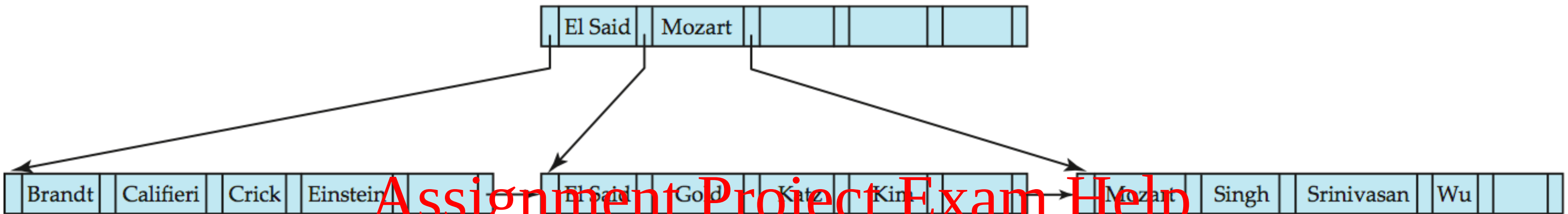
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
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 ($m \leq n$):
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq m - 1$, 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_m points have values greater than or equal to K_{m-1}

P_1	K_1	P_2	$\ldots$	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 $n = 6$).
- Internal nodes 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

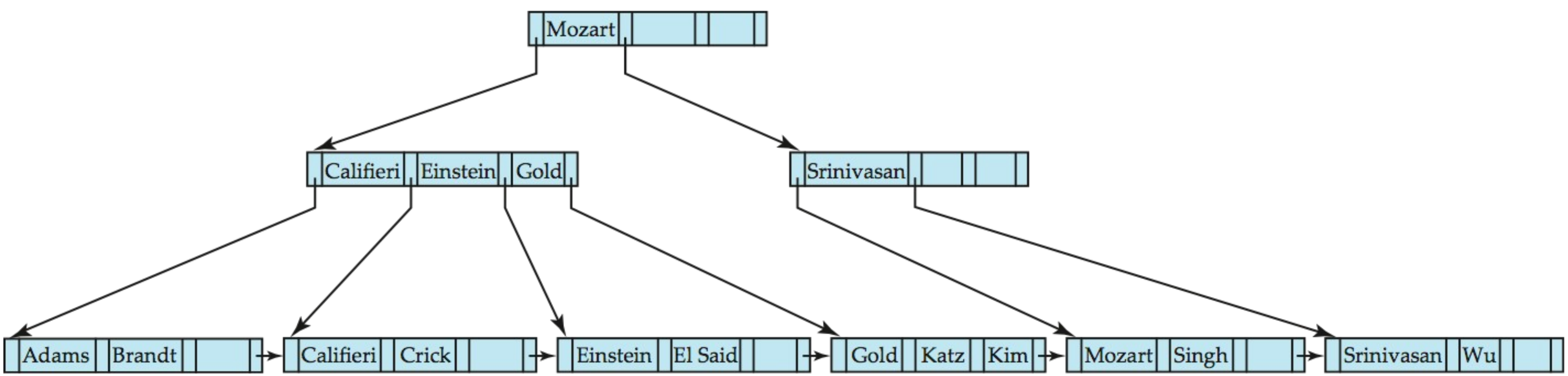
- Typically, a node is made to be the same size as a disk block.
- Since the inter-node connections are done by pointers, "logically" close blocks need **not** be "physically" close.
- The non-leaf levels of the B$^+$-tree form a hierarchy of sparse indices.

- The B$^+$-tree contains a relatively small number of levels. 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.

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 the value s.t. $K_i = V$ /* C is now a leaf node */
  4. If there is such a value *i,* follow pointer P_i to the desired record.
  5. Else no record with search-key value *V* exists.



Queries on B+-Trees (Cont.)

- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- Example: 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 and every node access may need a disk I/O

Updates on B⁺-Trees: Insertion

1. Find the leaf node in which the search-key value would appear

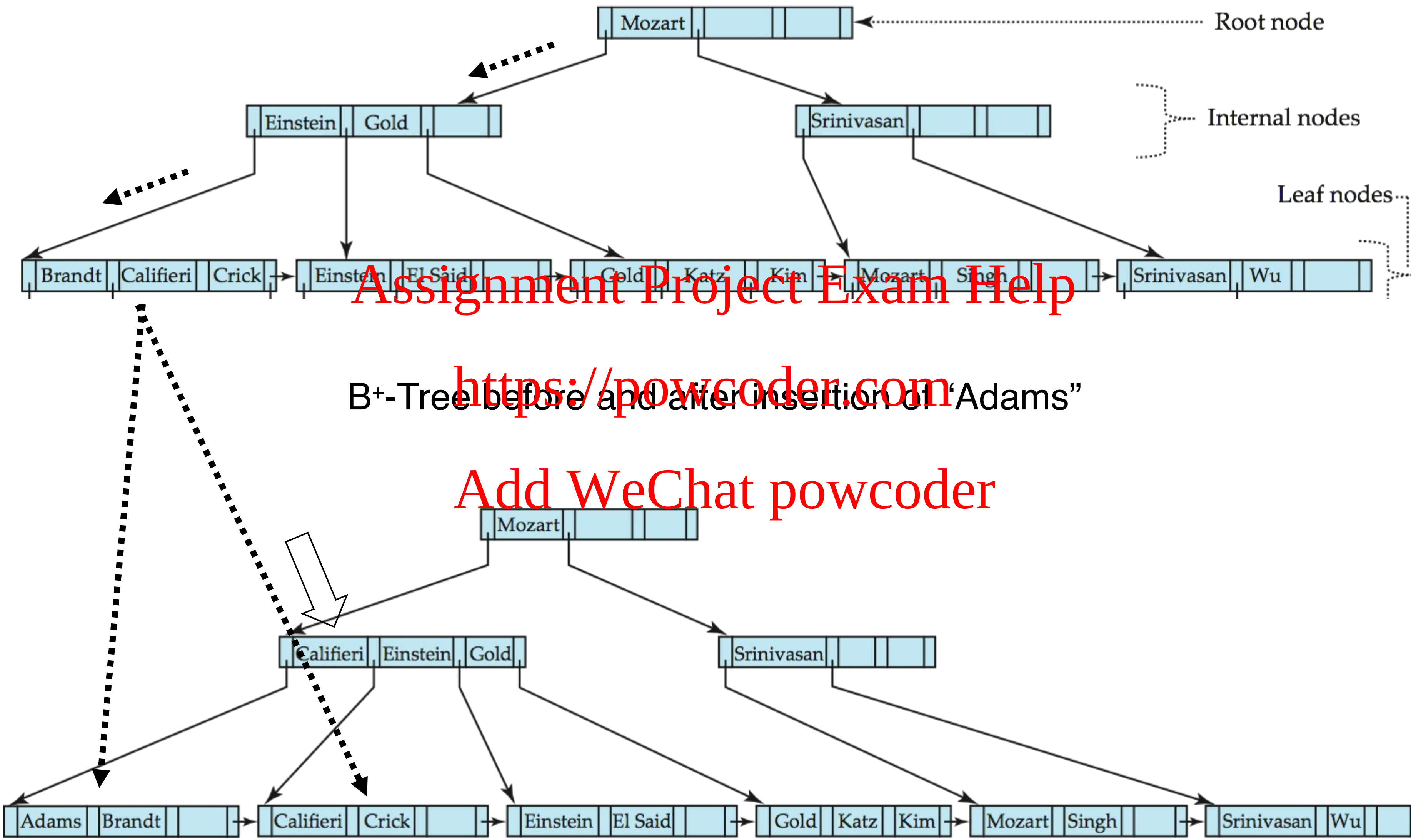
2. If there is room in the leaf node, insert (search-key value, record pointer) pair in the leaf node in sorted order

3. Otherwise, split the node (along with the new entry) as discussed in the next slide.

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.

B+-Tree Insertion



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

Insertion in B⁺-Trees (Cont.)

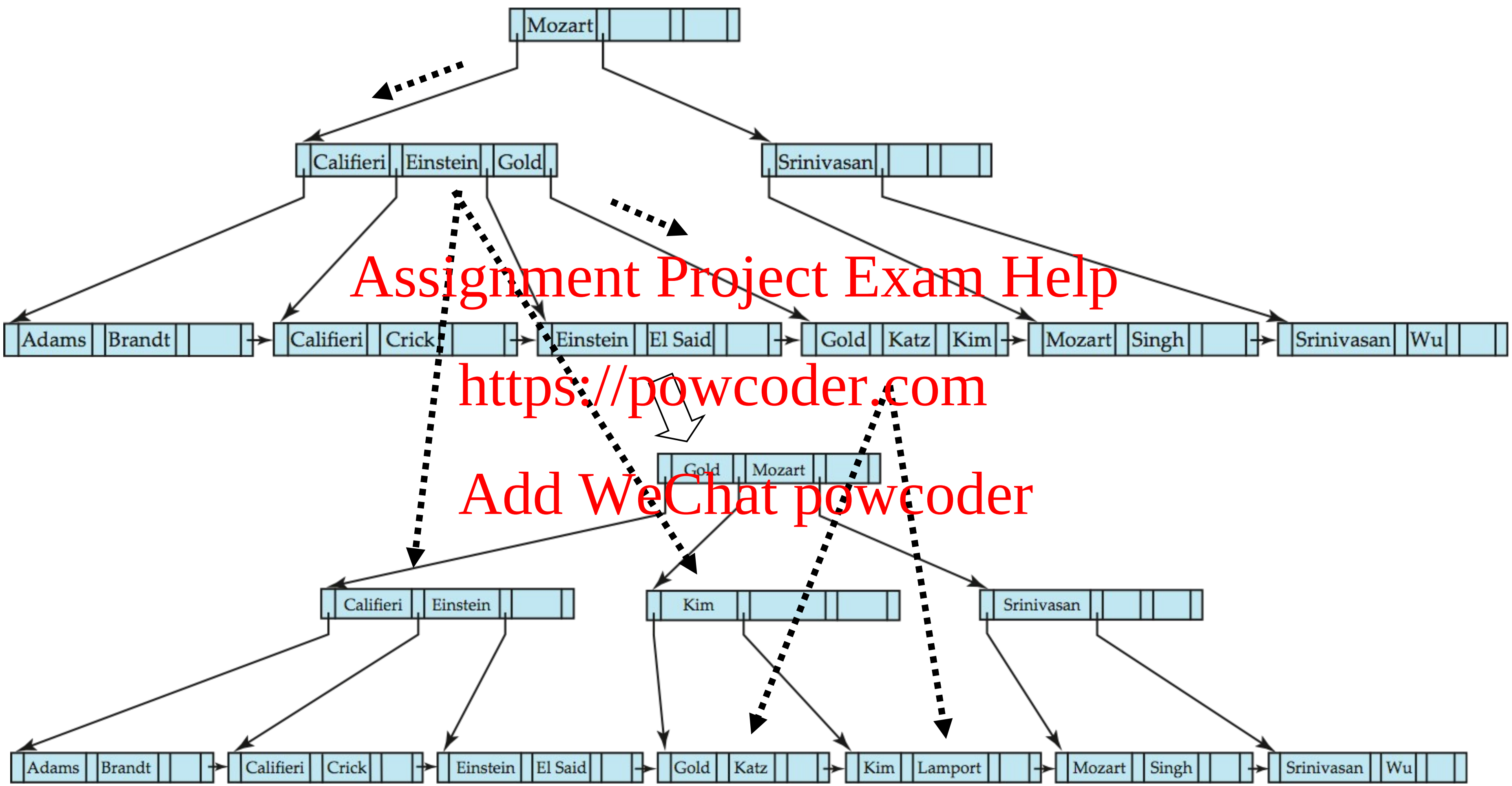
- Splitting a non-leaf node: when inserting (*k*,*p*) into an already full internal node *N*
  - Copy *N* to an in-memory area *M* with space for *n*+1 pointers and *n* keys

  - Insert (*k*,*p*) into *M*

  - Copy $P_1, K_1, \ldots, K_{\lceil (n+1)/2 \rceil - 1}, P_{\lceil (n+1)/2 \rceil}$ 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 of *N*
- **Read pseudocode in book!**

B+-Tree Insertion



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

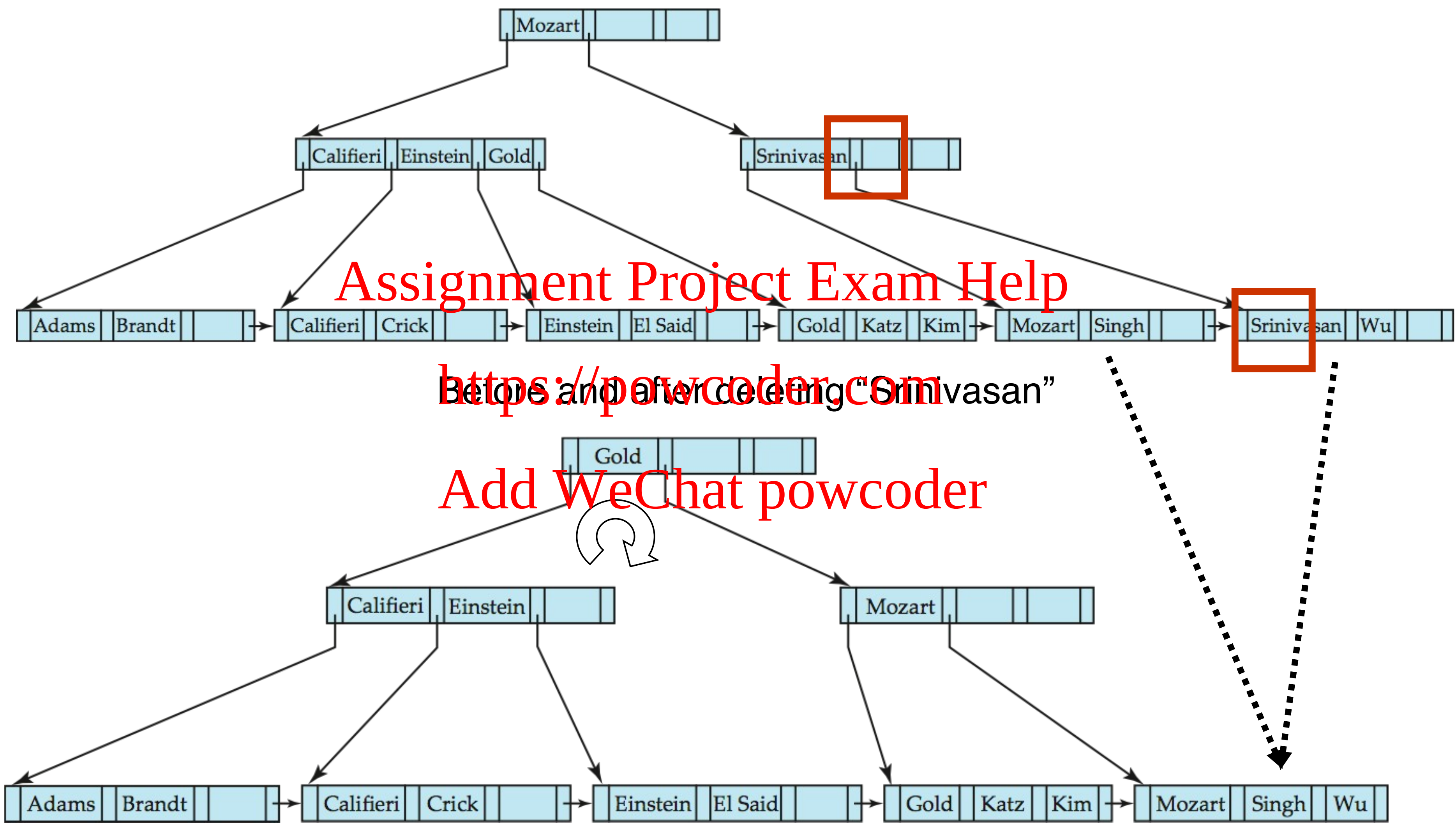
Updates on B$^+$-Trees: Deletion

- Perform a lookup on the search-key value of the deleted record to find the leaf node containing the entry to be deleted.
- Remove (search-key value, record pointer) 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.

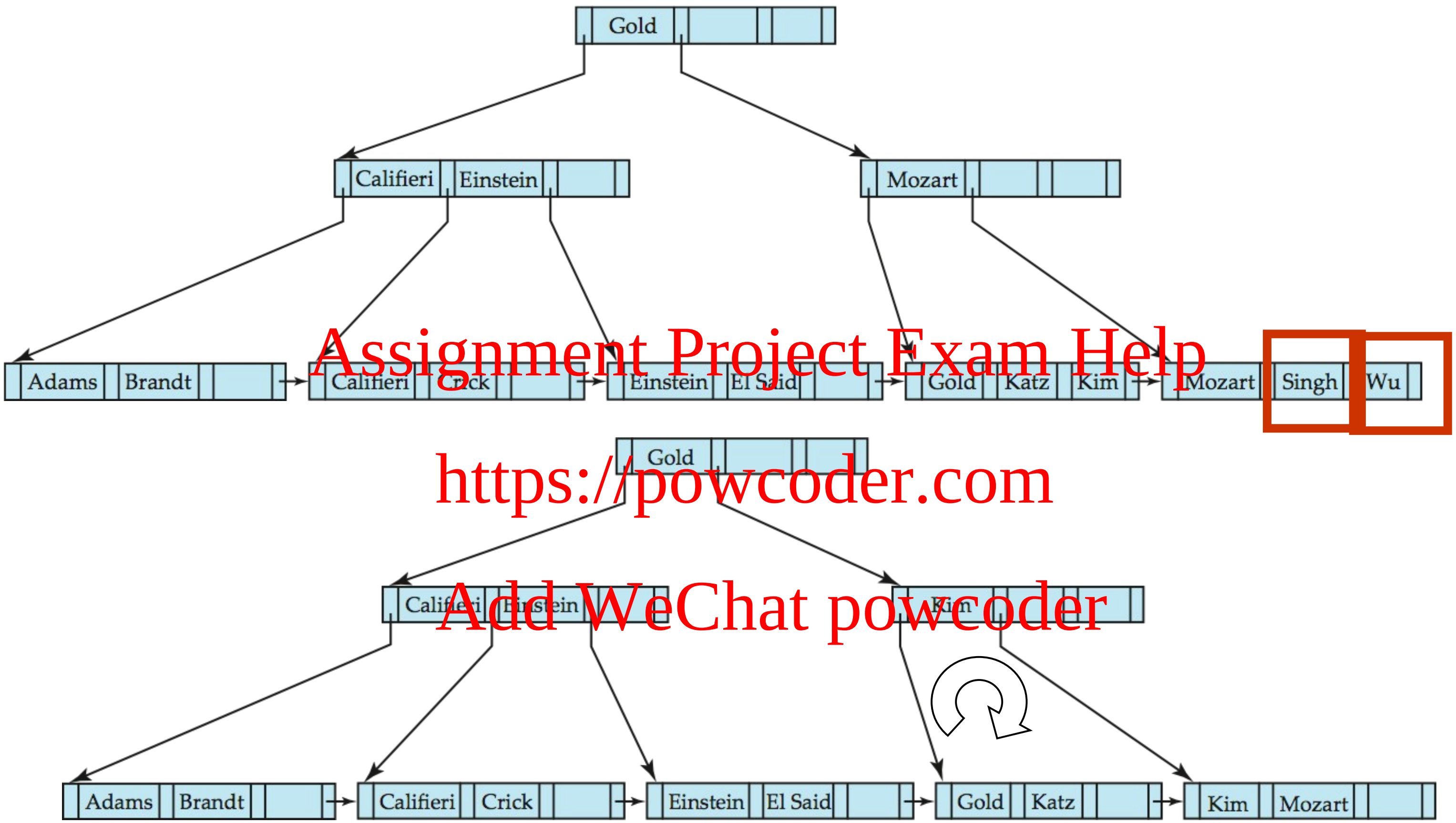
Examples of B+-Tree Deletion



Before and after deleting "Srinivasan"

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

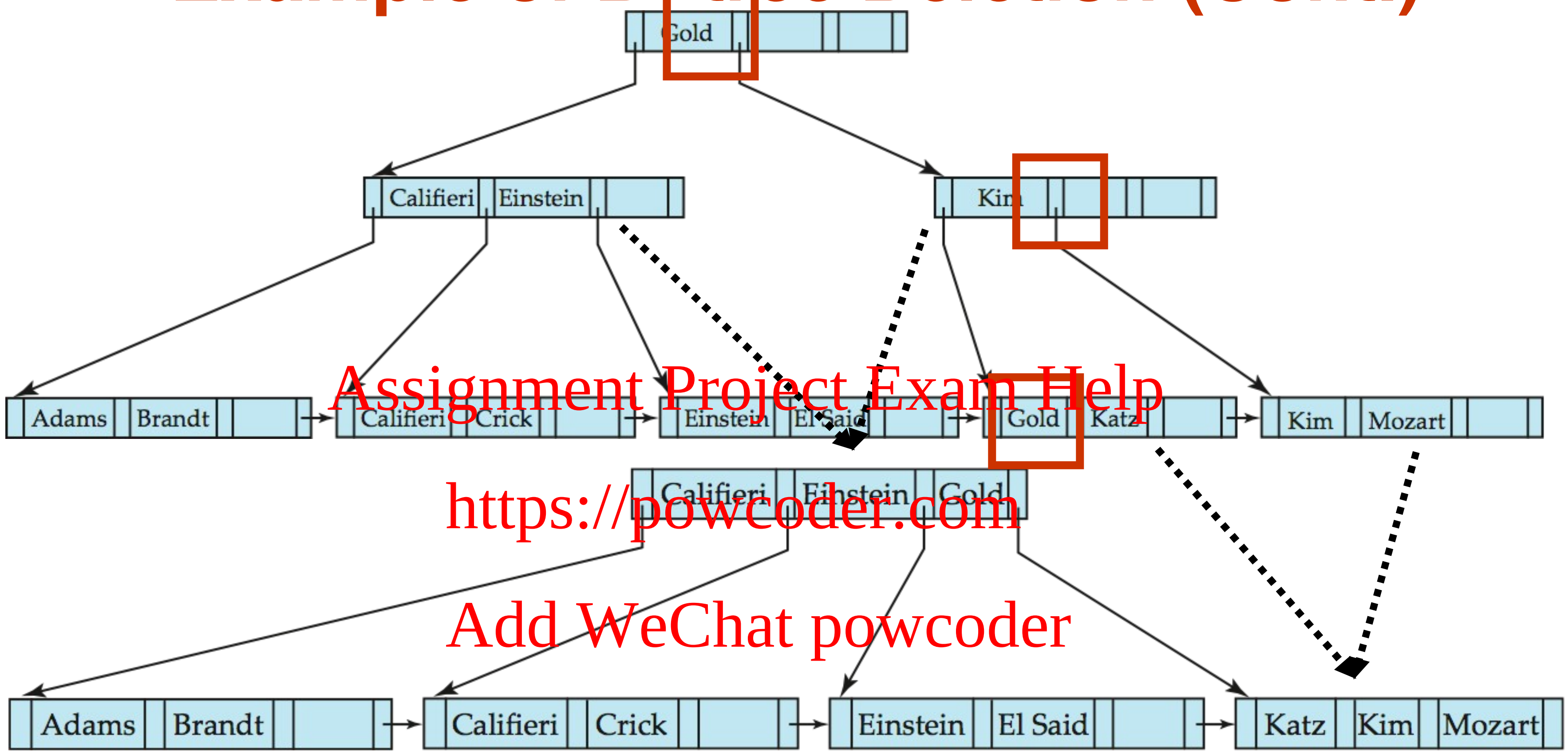
Examples of B+-Tree Deletion (Cont.)



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

Example of B⁺-tree Deletion (Cont.)



Before and after deletion of “Gold” from earlier example

- 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

More Observations about B+-trees

- Although insertion and deletion operations on B+-trees are complicated, they require relatively few expensive I/O operations.
  - 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 number of pointers in 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 than the worst-case.
  - Even if the relation is very large, it is quite likely that most of the nonleaf nodes are already in the memory
  - Splits/merges are rare, most insert/delete operations only affect a leaf node.

Bulk Loading and Bottom-Up Build

- Inserting entries one-at-a-time into a B^+-tree requires 1 I/O per entry
  - Assuming leaf level does not fit in memory
  - Can be very inefficient to insert a large number of entries at a time into a non-clustering index (**bulk loading**)
- Example: consider to build a non-clustering B^+-tree index on a relation with 100 million records
  - If each ***random*** I/O operation takes 10 msec on a disk, it would take at least 1 million seconds to build the index.

vs

  - If the size of each record is 100 bytes and the disk can transfer data at 50 Mbytes/sec, it would take 200 sec to read the entire relation.

Bulk Loading and Bottom-Up Build (cont.)

- Efficient alternative 1:
  - Create and sort entries first (using efficient external-memory sort algorithms to be discussed later)
  - Insert the entries into the B+-tree in sorted order
    - All entries that go to a particular leaf node will appear consecutively.
    - Much improved I/O performance.
  - Example:
    - If each leaf contains 100 entries, the leaf level will contain 1 million nodes.
    - If each ***sequential*** I/O operation takes 1 msec on a disk, the same index can be built in 1000 seconds.
- Efficient alternative 2: **Bottom-up B+-tree construction**
  - As before, create and sort entries
  - And then create tree layer-by-layer, starting with leaf level
  - Implemented as part of bulk-load utility by most database systems

Indexing on Flash

- Flash storage is structured as pages and the B^+-tree index structure can be used with flash-based SSDs (solid state disks).
- Since flash pages are smaller than disk blocks, B^+-tree node size is also smaller $\rightarrow$ taller trees and more I/O operations to access data.

  - The impact on read performance is small because random page reads are so much faster with flash storage (20 to 100 microseconds)

- Writes are not in-place (i.e., every update turns into a copy + write of an entire page), and (eventually) require a more expensive erase (2 to 4 msec per block erase).
- Bulk-loading and bottom-up construction still useful since they minimize page erases.

Indexing in Main Memory

- Main memory is large and cheap enough to keep operational data in-memory and B+-tree can be used to index in-memory data.
- When reading a memory location, if it is present in cache, the CPU can complete the read in 1 or 2 nanoseconds, whereas a cache miss results in about 50-100 nanoseconds of delay.
- B+- trees with small nodes that fit in cache line (typically about 64 bytes) are preferable to reduce cache misses
  - Otherwise, search for a key value within a large B+-tree node spanning multiple cache lines results in many cache misses.
- Key idea: use large node size to optimize disk access, but structure data within a node using a tree with small node size, instead of using an array.

Indexing

- Review of Basic Concepts and Ordered Indices
- B^+-Tree Index Files
- **Static Hashing**
- Dynamic Hashing
- Multiple-Key Access
- Creation of Indices

Static Hashing

- A **bucket** is a unit of storage containing one or more entries (a bucket is typically a disk block).
- We obtain the bucket of an entry 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.

- Hash function is used to locate entries for access, insertion as well as deletion.

- Entries with ***different*** search-key values may be mapped to the ***same*** bucket; thus entire bucket has to be searched sequentially to locate an entry.
- In a **hash index**, buckets store entries with pointers to records.
- In a **hash file-organization**, buckets store records.

Hash Functions

- The 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.

- A good hash function gives an average-case lookup time that is a (small) constant, independent of the number of search keys 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.
  - Is a hash function, which maps *name* beginning with the *i*th letter of the alphabet to the *i*th bucket, uniform?

- An 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.
  - Is a hash function, which divides *salary* into ranges, 1-10000, 10001-20000, etc, random?

Example of Hash File Organization

- Typical hash functions perform computation on the internal binary representation of the search-key.
  - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned.
  - Hash file organization of *instructor* file, using *dept_name* as key (See figure in next slide.)
    - 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.
    - E.g. h(Music) = 1 h(History) = 2
      h(Physics) = 3 h(Elec. Eng.) = 3

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

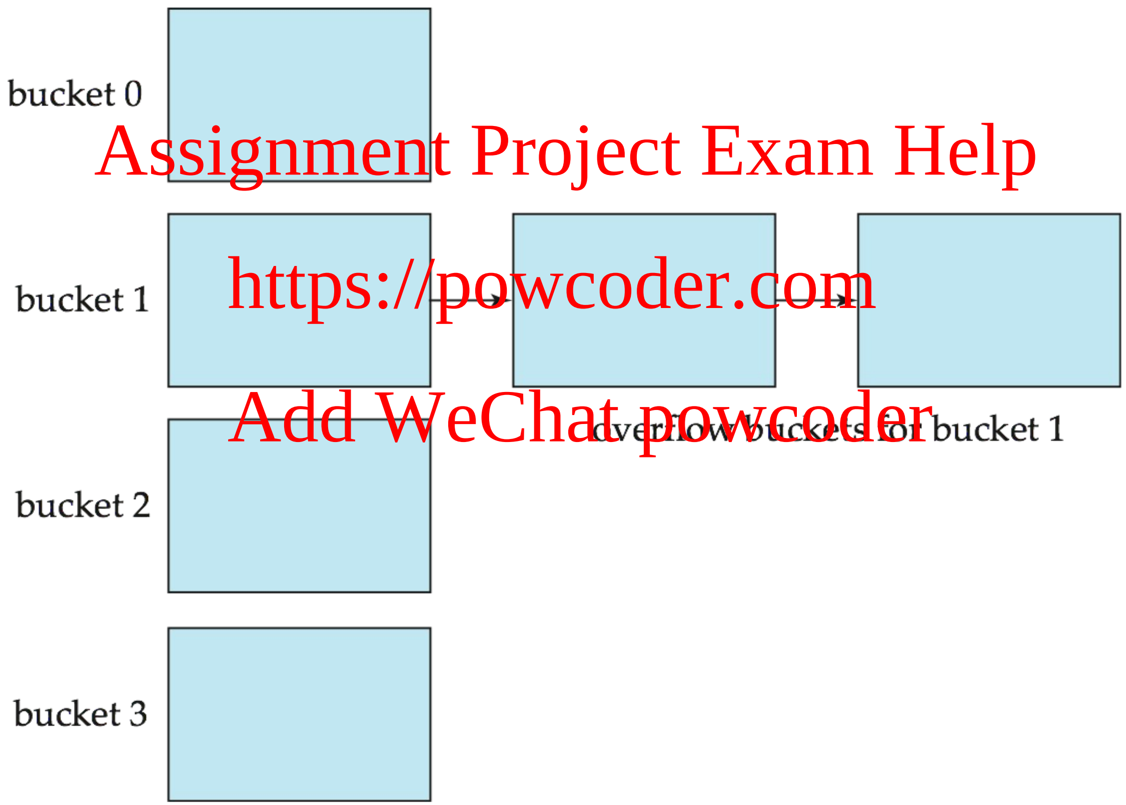
Hash file organization of *instructor* file, using *dept_name* as key (see previous slide for details).

Handling of Bucket Overflows

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records.
    - poor hash function
    - multiple records have same search-key value
- Although the probability of bucket overflow can be reduced (e.g., by increasing the number of buckets), it cannot be eliminated; it is handled by using **overflow buckets**.

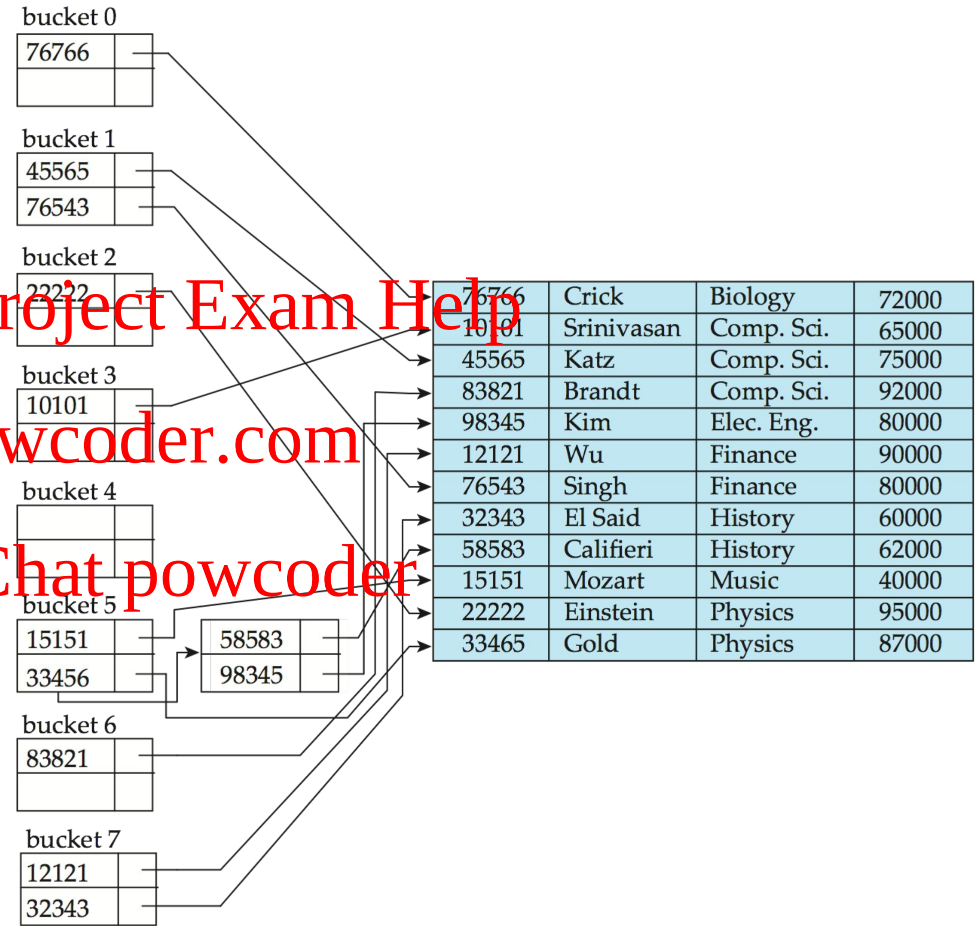
Handling of Bucket Overflows (Cont.)

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



Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.



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

Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a ***fixed*** set of bucket addresses *B*. However, 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.
  - If database shrinks again, space will be wasted.

- 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.

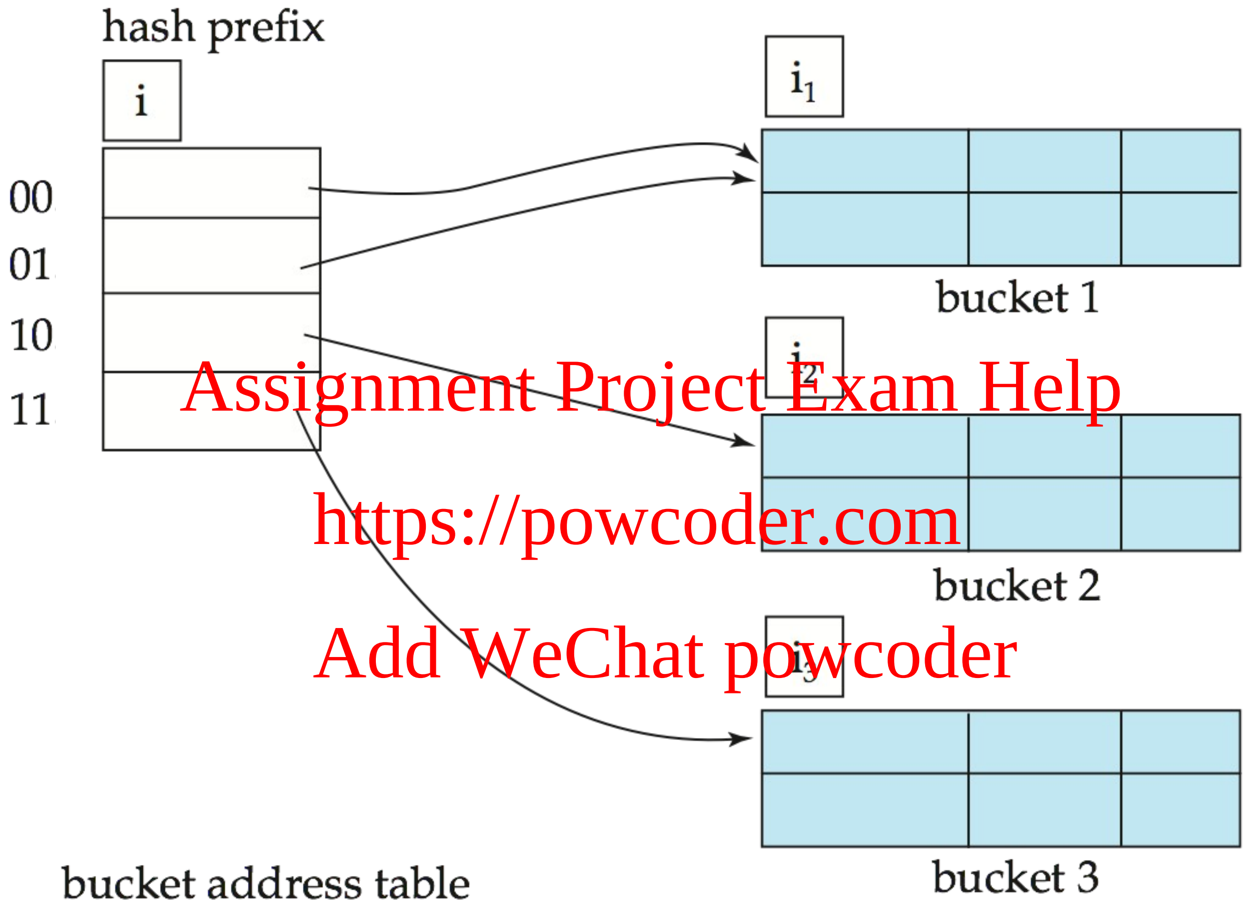
Indexing

- Review of Basic Concepts and Ordered Indices
- B+-Tree Index Files
- Static Hashing
- **Dynamic Hashing**
- Multiple-Key Access
- Creation of Indices

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
  - Hash function generates values over a large range — typically *b*-bit integers, with $b = 32$ (> 4 billion).
  - At any time, use only a prefix of the hash function to index into a **bucket address table**.
  - 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.
  - Multiple entries in the bucket address table may point to a bucket. Thus, actual number of buckets is < 2^i.
  - The number of buckets also changes dynamically due to coalescing and splitting of buckets.
  - Each bucket j stores the length of common hash prefix, i_j
    - All the entries that point to the same bucket have the same values on the first i_j bits.

General Extendable Hash Structure



Suppose $i = 2$, $i_1 = 1$ and $i_2 = i_3 = 2$

Use of Extendable Hash Structure

- To locate the bucket containing search-key value 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

- 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 (next slide.)
    - Overflow buckets used instead in some cases (will see shortly)

Insertion in Extendable Hash Structure (Cont)

To split a bucket *j* when inserting record with search-key value K_j:

- 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)
- 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.

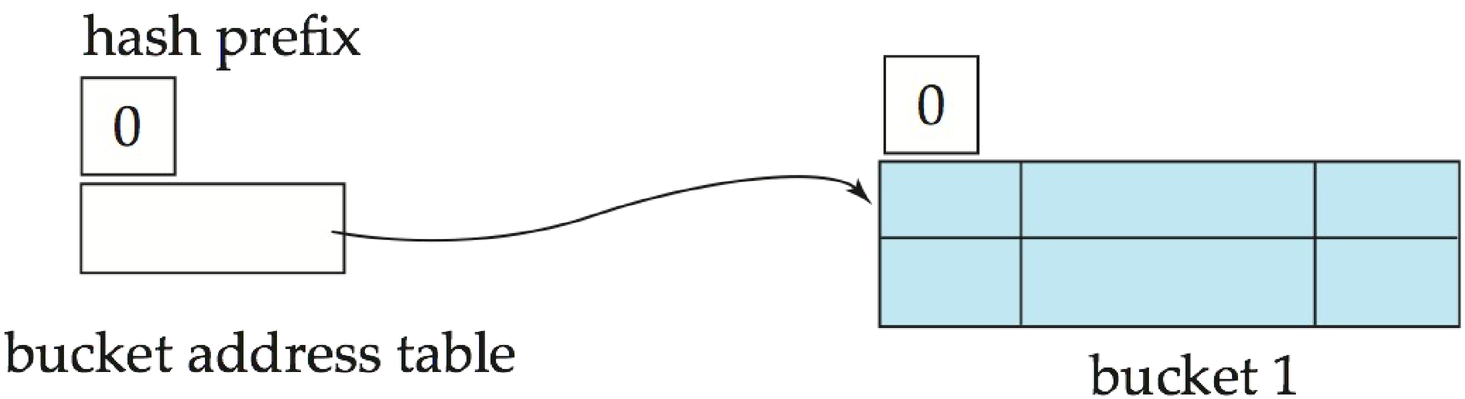
Deletion in Extendable Hash Structure

- 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

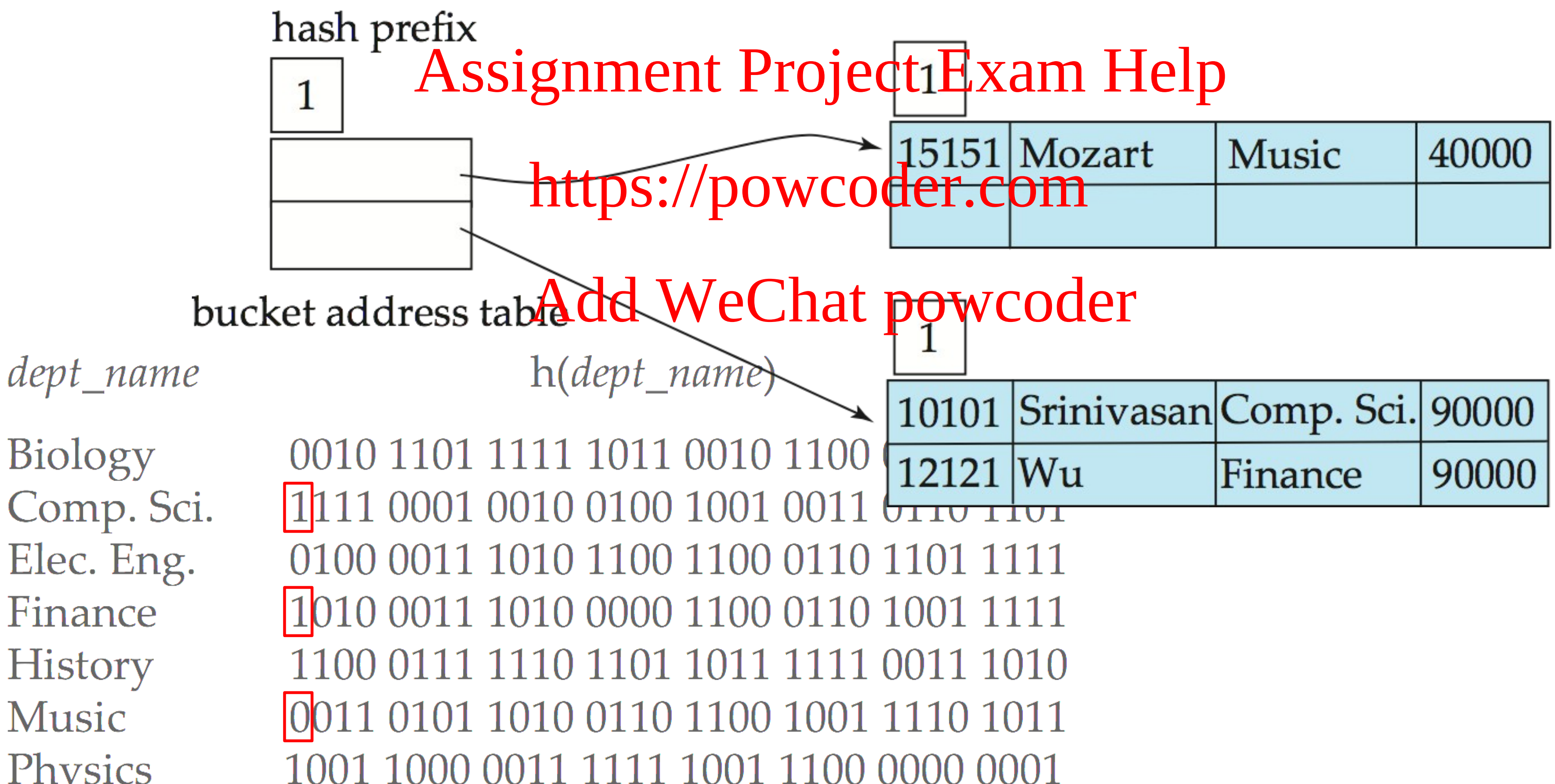
Hash function



Initial Hash structure; bucket size = 2

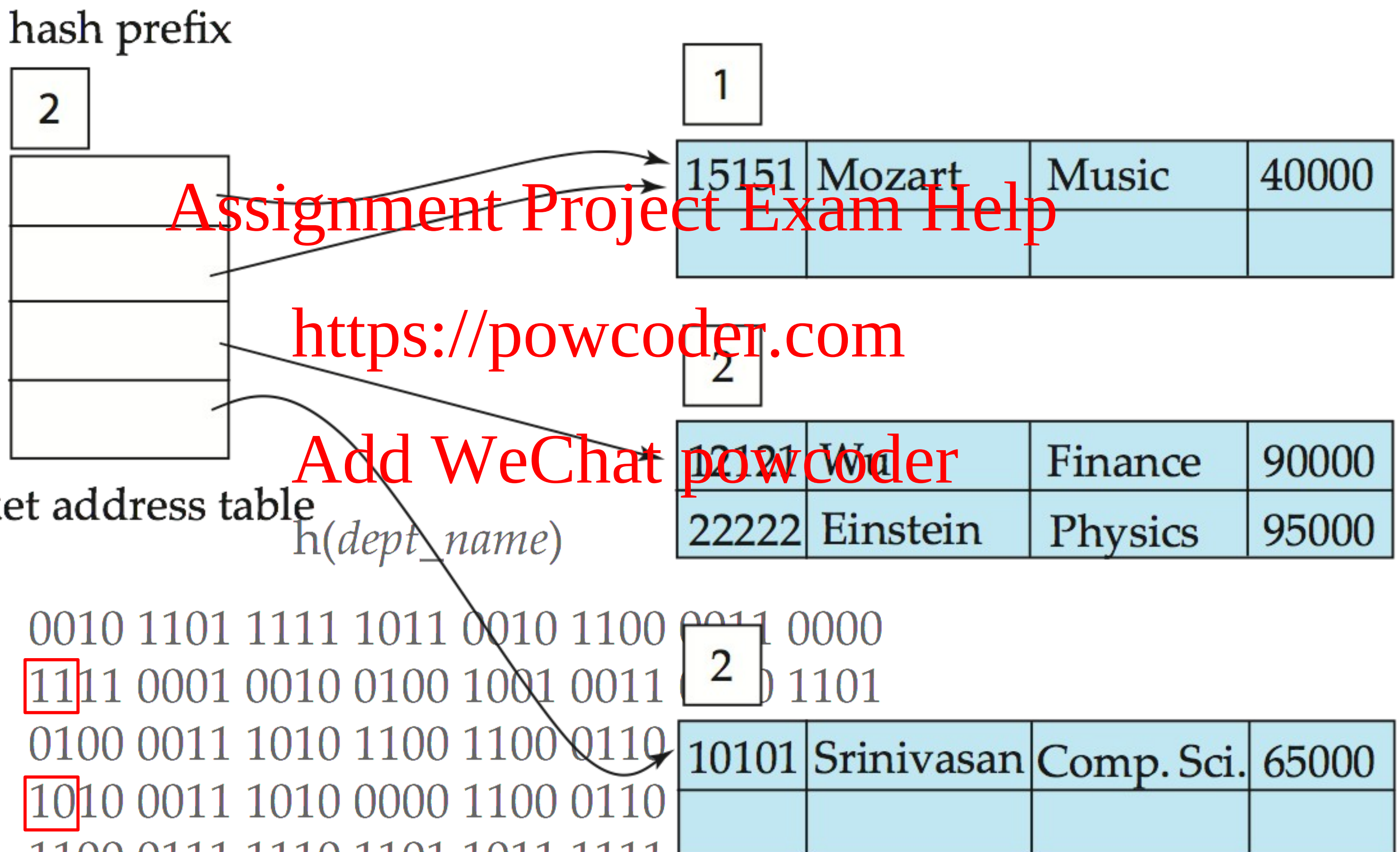
Example (Cont.)

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



Example (Cont.)

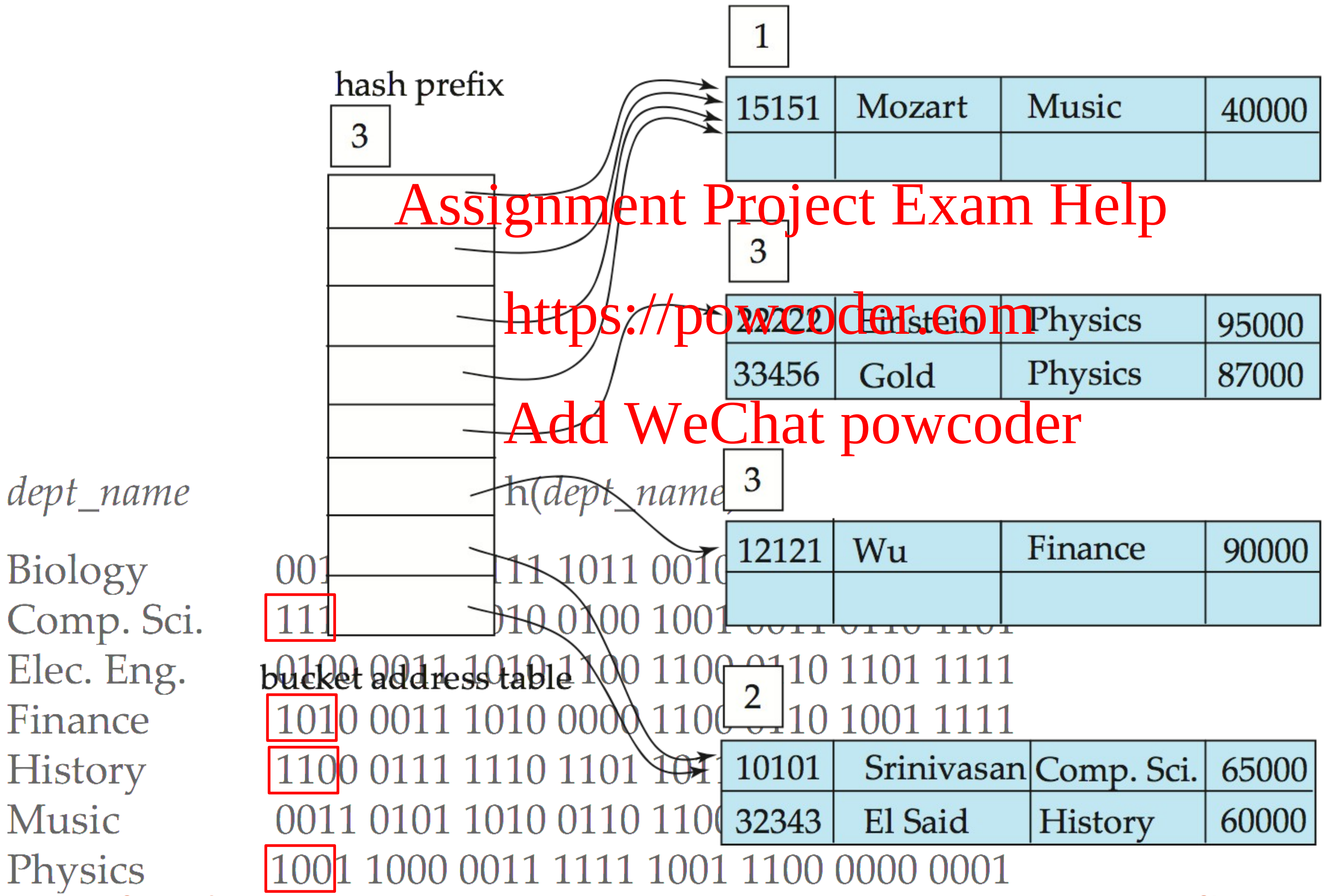
- Hash structure after insertion of Einstein record



dept_name	*h(dept_name)*
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 [illegible] 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 [illegible]
Finance	1010 0011 1010 0000 1100 0110 [illegible]
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

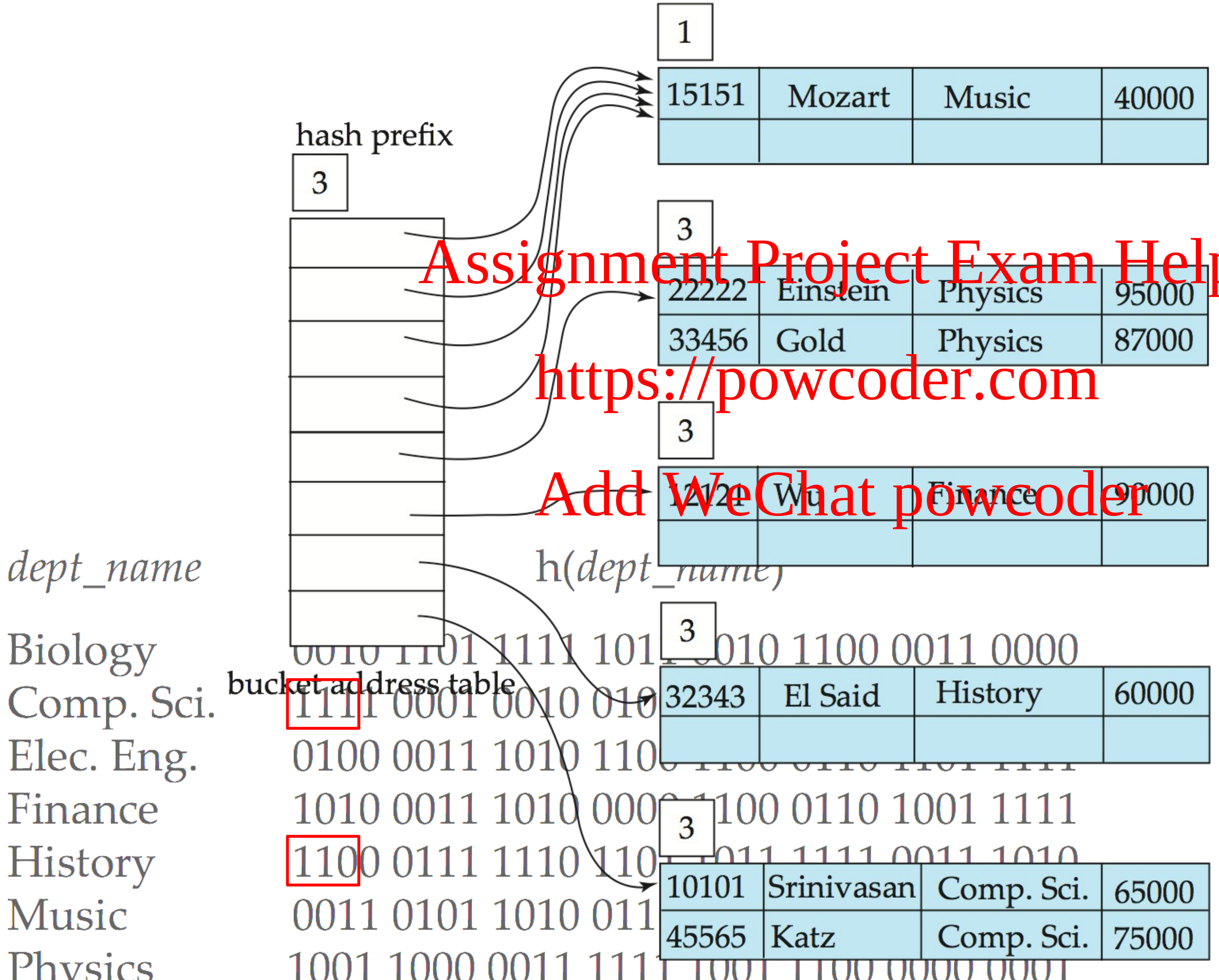
Example (Cont.)

- Hash structure after insertion of El Said and Gold records



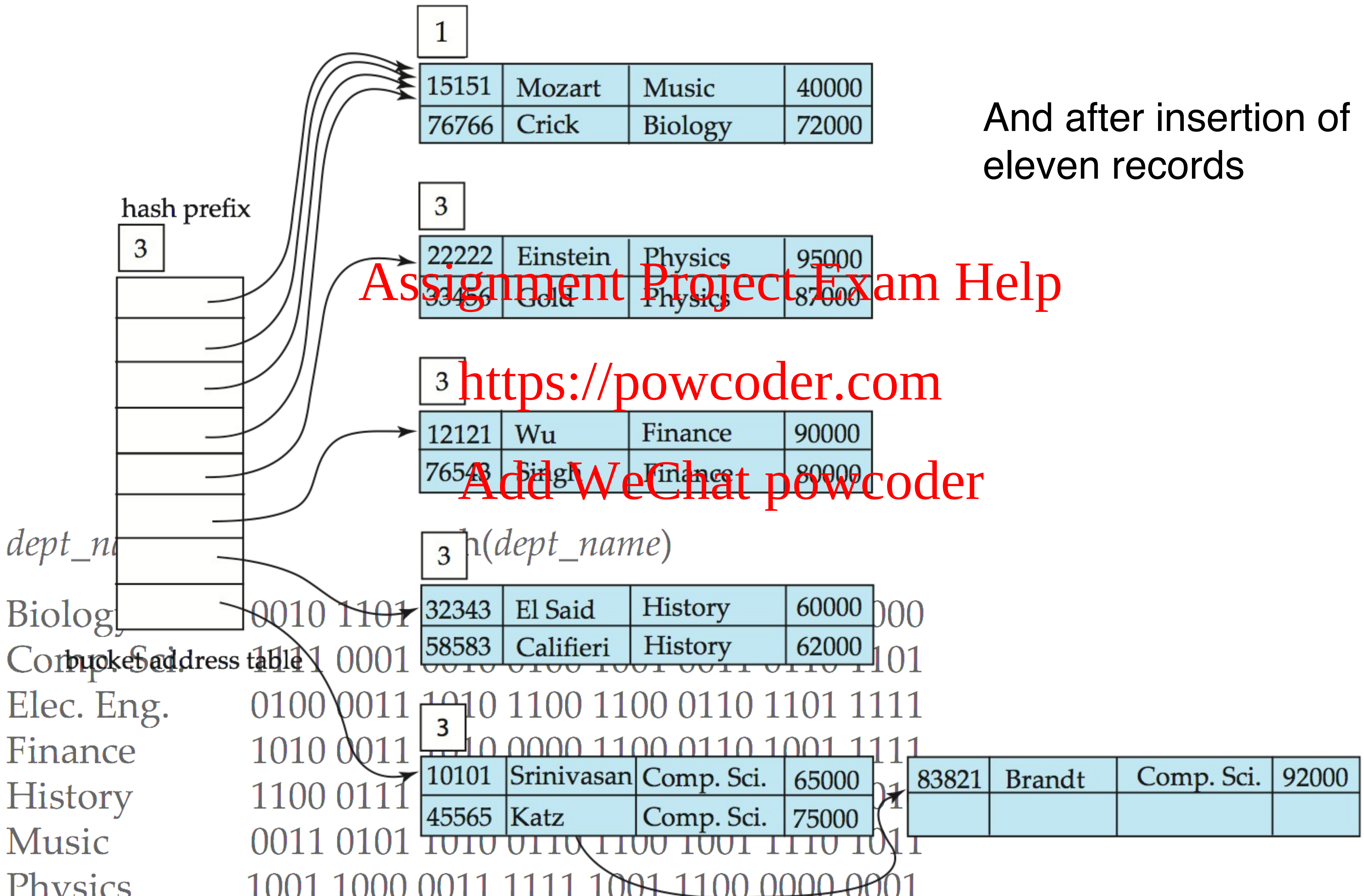
Example (Cont.)

- Hash structure after insertion of Katz record

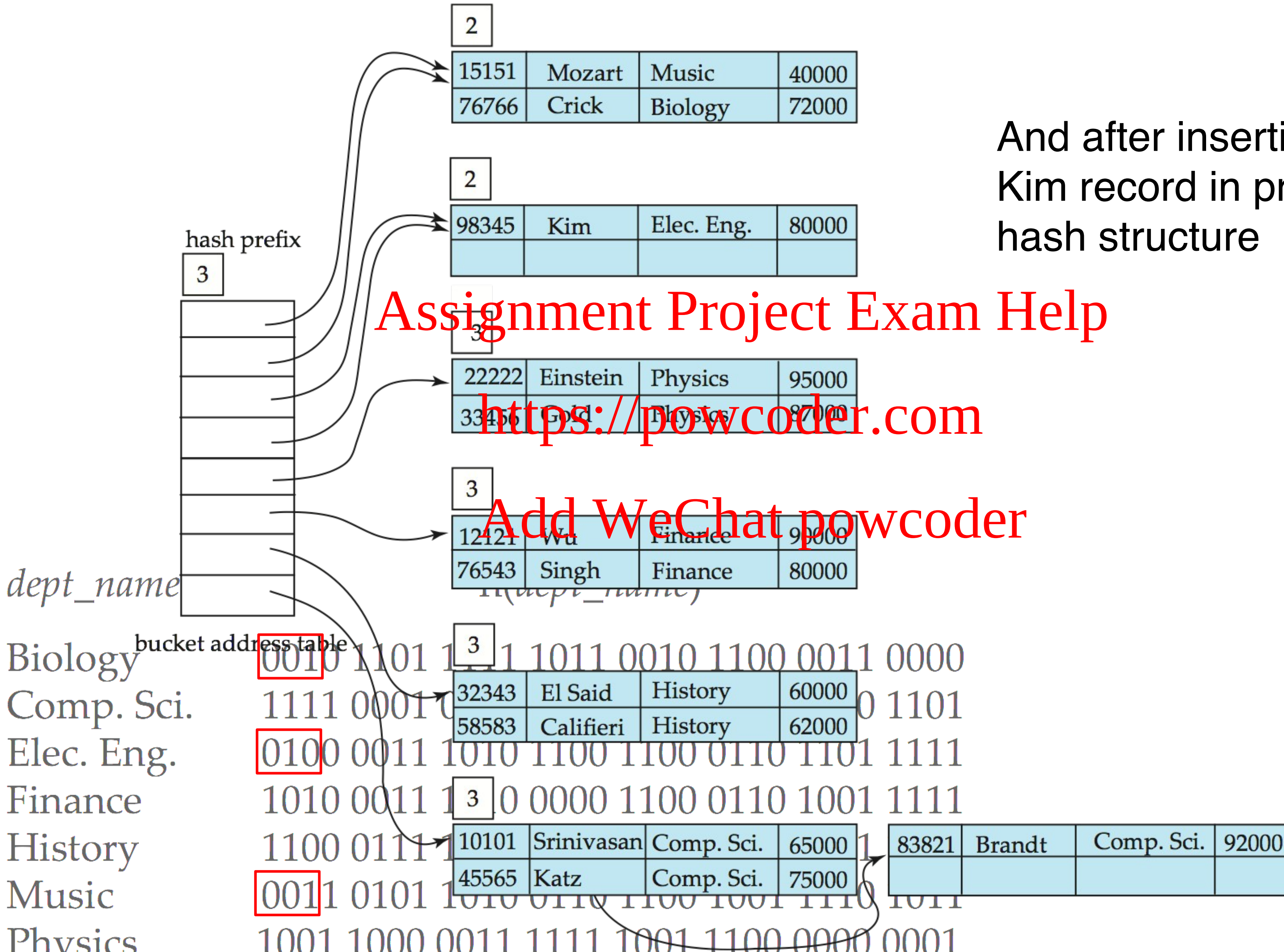


Example (Cont.)

And after insertion of eleven records



Example (Cont.)



And after insertion of Kim record in previous hash structure

Extendable Hashing

- Benefits of extendable hashing:
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
    - no buckets need to be reserved for future growth

- Disadvantages of extendable hashing
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big (larger than memory)
    - Solution: B^+-tree structure to locate desired record in bucket address table
  - Changing size of bucket address table is an expensive operation

Ordered Indexing versus Hashing

- Design considerations:
  - 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:
    - 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:
  - Hash-indices are extensively used in-memory but not used much on disk.
  - Oracle supports static hash organization, but not hash indices.
  - SQL Server and PostgreSQL do not support hashing on disk.

Indexing

- Review of Basic Concepts and Ordered Indices
- B^+-Tree Index Files
- Static Hashing
- Dynamic Hashing
- **Multiple-Key Access**
- Creation of Indices

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

- Using separate indices could be less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.
- **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 Attributes

- Example: suppose we have an index on combined search-key (*dept_name, salary*).
- With the **where** clause
  where *dept_name* = "Finance" **and** *salary* = 80000
  the index on (*dept_name, salary*) can be used to fetch only records that satisfy both conditions.
- Can also efficiently handle
  where *dept_name* = "Finance" **and** *salary* < 80000
- But cannot efficiently handle
  where *dept_name* < "Finance" **and** *salary* = 80000
  - May fetch many records that satisfy the first but not the second condition due to the ordering of records in the file.

Other Features

- **Covering indices**
  - Add extra attributes to index so (some) queries can avoid fetching the actual records.
  - Example: consider an index on *ID* attribute of *instructor* relation.
    - If we store values of *salary* attribute in the index, we can answer queries that require salary without accessing the instructor record.
  - Store extra attributes only at leaf.
  - Particularly useful for secondary indices.

Indexing

- Review of Basic Concepts and Ordered Indices
- B^+-Tree Index Files
- Static Hashing
- Dynamic Hashing
- Multiple-Key Access
- **Creation of Indices**

Creation of Indices

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

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

  - E.g.: **create index** *dept_index* **on** *instructor*(*dept_name*)

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

- To drop an index

  drop index <index-name>

- Indices on primary key created automatically by all databases.
- Some database also create indices on foreign key attributes.
  - For example, such an index might be useful for queries in which the **join attribute** ID is a **foreign-key attribute** ID of *takes* references the **primary-key attribute** ID of *student*.
    - $takes \bowtie \sigma_{name='Shankar'}(student)$