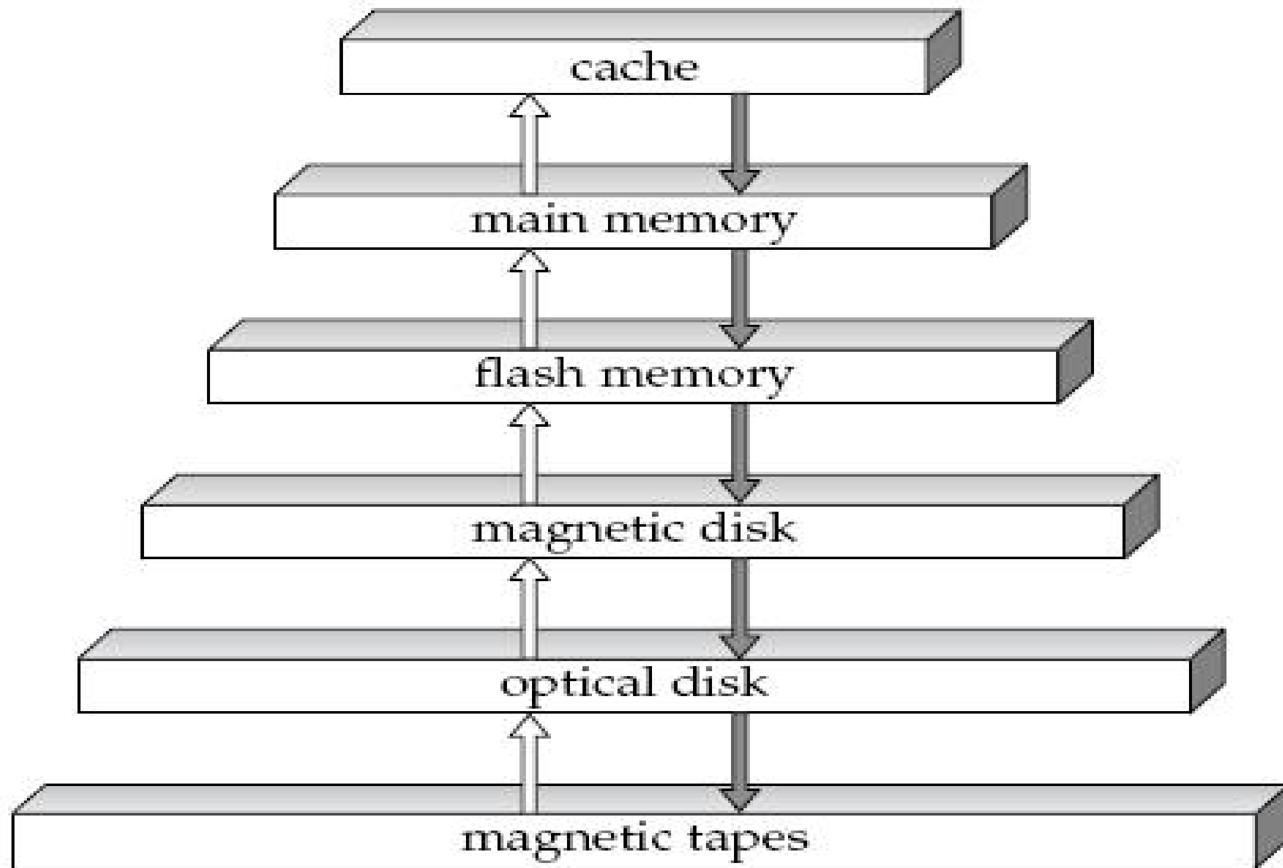
23CY502
Database Management Systems

P.Prasanna Kumari
Assistant Professor,
CSE(Cyber Security)

UNIT-V

Data on External Storage

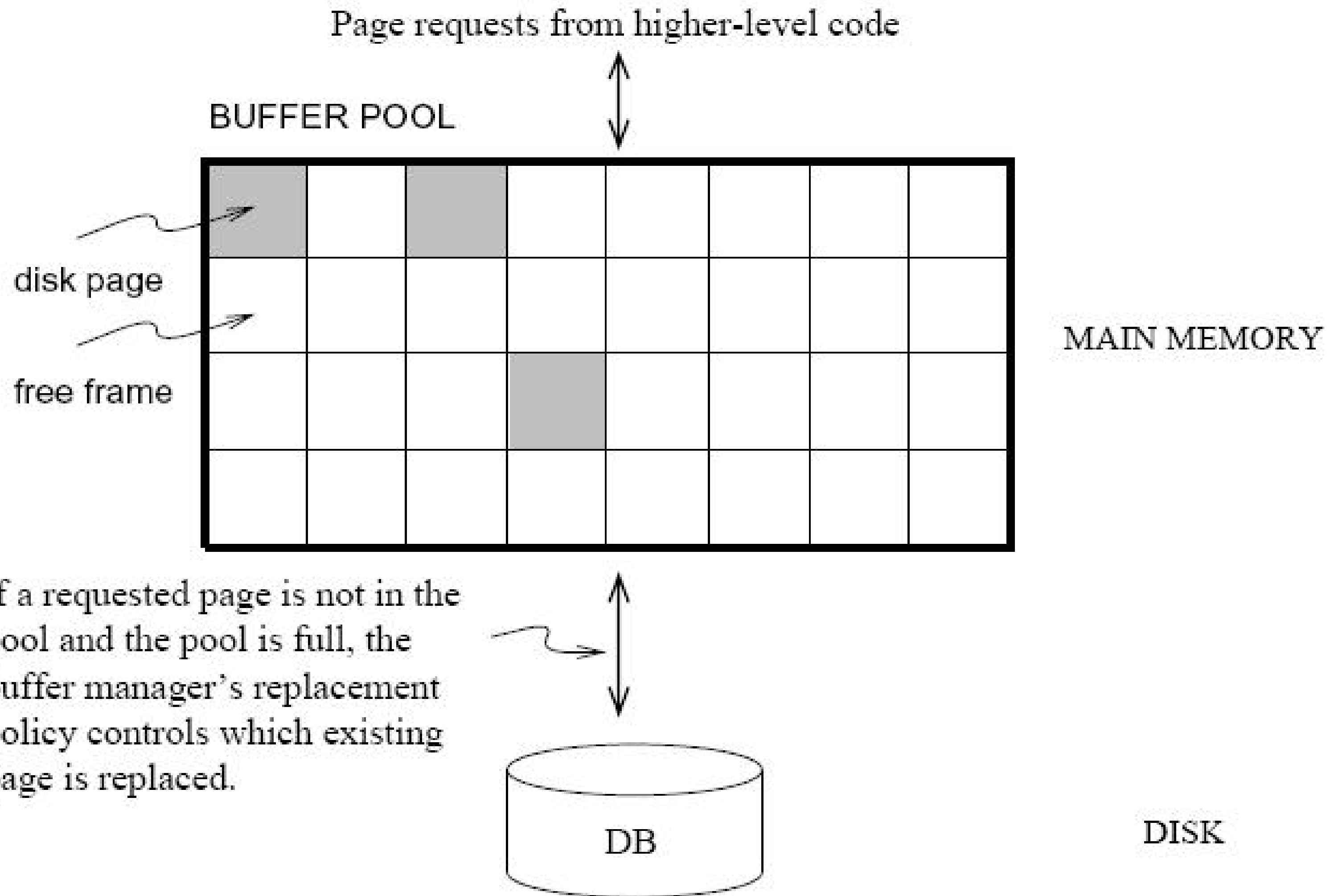
Storage-device hierarchy



Buffer Manager

- Files reside permanently on disks.
- Each file is partitioned into fixed-length storage units called **blocks.**
- The **buffer** is the part of main memory available for storage of copies of disk blocks.
- The subsystem responsible for the allocation of buffer space is called the **buffer manager.**
- The **buffer manager is** the software layer that is responsible for bringing pages from disk to main memory as needed.
- The buffer manager manages the available main memory by partitioning it into a collection of pages, which we collectively refer to as the **buffer pool.**
- **The main** memory pages in the buffer pool are called **frames; it is convenient to think of them** as slots that can hold a page.

The buffer pool



Buffer Manager techniques

- **Buffer replacement strategy:** When there is no room left in the buffer, a block must be removed from the buffer. Most operating systems use a least recently used (LRU) scheme.

- **Pinned blocks**: Most recovery systems require that a block should not be written to disk while an update on the block is in progress. A block that is not allowed to be written back to disk is said to be pinned.

- **Forced output of blocks**: There are situations in which it is necessary to write back the block to disk, even though the buffer space that it occupies is not needed. This write is called the forced output of a block.

Record Structure

- The database is stored as a collection of files.
- Each file is a sequence of records.
- A record is a sequence of fields.

Types of records

- **Fixed-Length Records**: every record in the file has exactly the same size (in bytes).
- **Variable-Length Records:** different records in the file have different sizes.

Fixed-Length Records

Let us consider a file of account records for bank database. Each record of this file is defined as:

Account-number: char (10);
Branch-name: char (22);
Balance: real; //Real size=8

Record size= 10+22+8= 40 bytes

A simple approach is to use the first 40 bytes for the first record, the next 40 bytes for the second record, and so on.

There are two problems with this simple approach:

1. It is difficult to delete a record from this structure. The space occupied by the record to be deleted must be filled with some other record of the file, or we must have a way of marking deleted records so that they can be ignored.

2. Unless the block size happens to be a multiple of 40 (which is unlikely), some records will cross block boundaries. That is, part of the record will be stored in one block and part in another. It would thus require two block accesses to read or write such a record.

Deletion of record 1st approach

- When a record is deleted, we could move the record that came after it into the *space* occupied by the deleted record, and so on, until every record following the deleted record has been moved ahead. Such an approach requires moving a large number of records.

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Deletion of record 2nd approach

- It might be easier simply to move the final record of the file into the space occupied by the deleted record. It is undesirable to move records to occupy the space freed by a deleted record, since doing so requires additional block accesses

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 8	A-218	Perryridge	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600

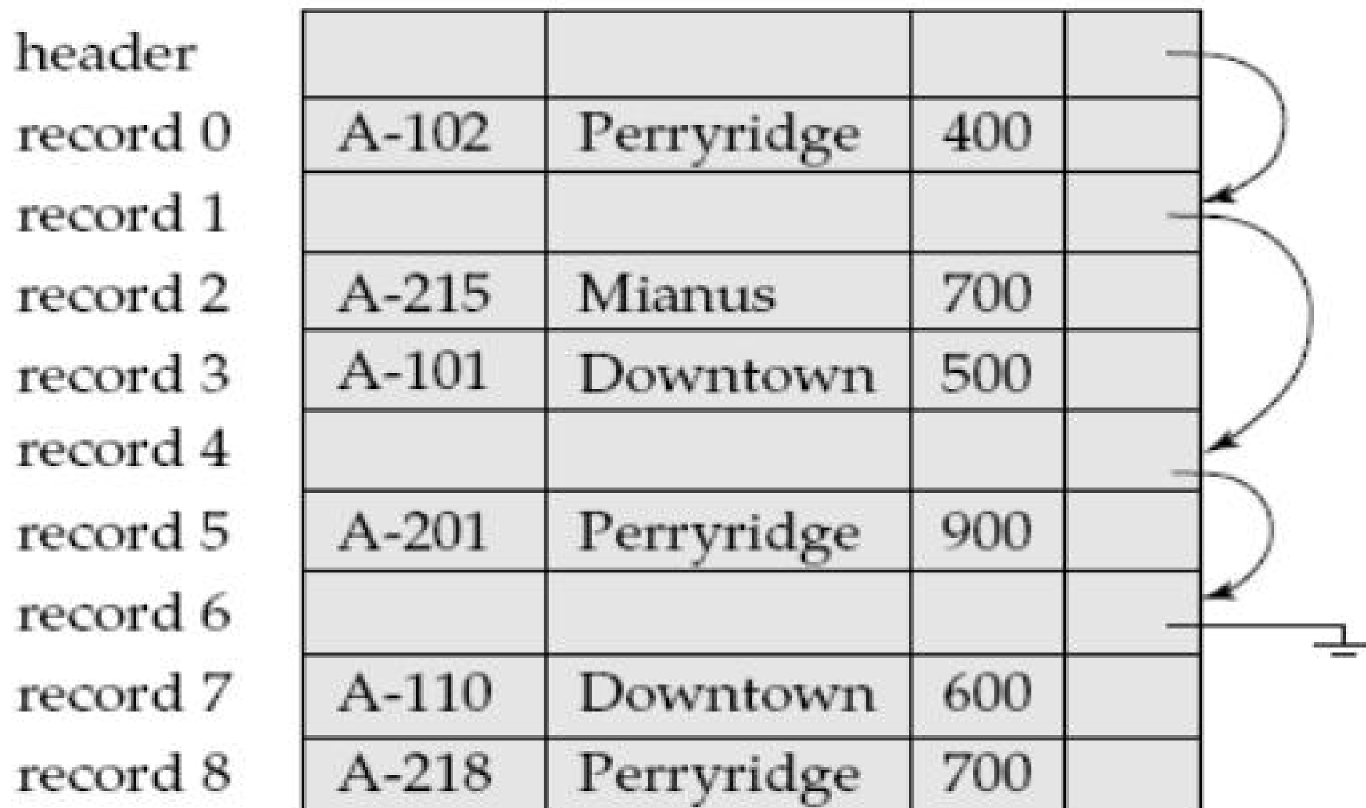
Deletion of record 3rd approach

- Since insertions tend to be more frequent than deletions, it is acceptable to leave open the space occupied by the deleted record, and to wait for a subsequent insertion before reusing the space.
- A simple marker on a deleted record is not sufficient, since it is hard to find this available space when an insertion is being done.
- Thus, we need to introduce an additional structure.

File header

- At the beginning of the file, we allocate a certain number of bytes as a **file header.**
- The header will contain a variety of information about the file.
- For now, all we need to store there is the address of the first record whose contents are deleted.
- We use this we can think of these stored addresses as pointers, since they point to the location of a record.
- The deleted records thus form a linked list, which is often referred to as a **free list.**
- On insertion of a new record, we use the record pointed to by the header.
- We change the header pointer to point to the next available record.
- If no space is available, we add the new record to the end of the file.

Continued…….

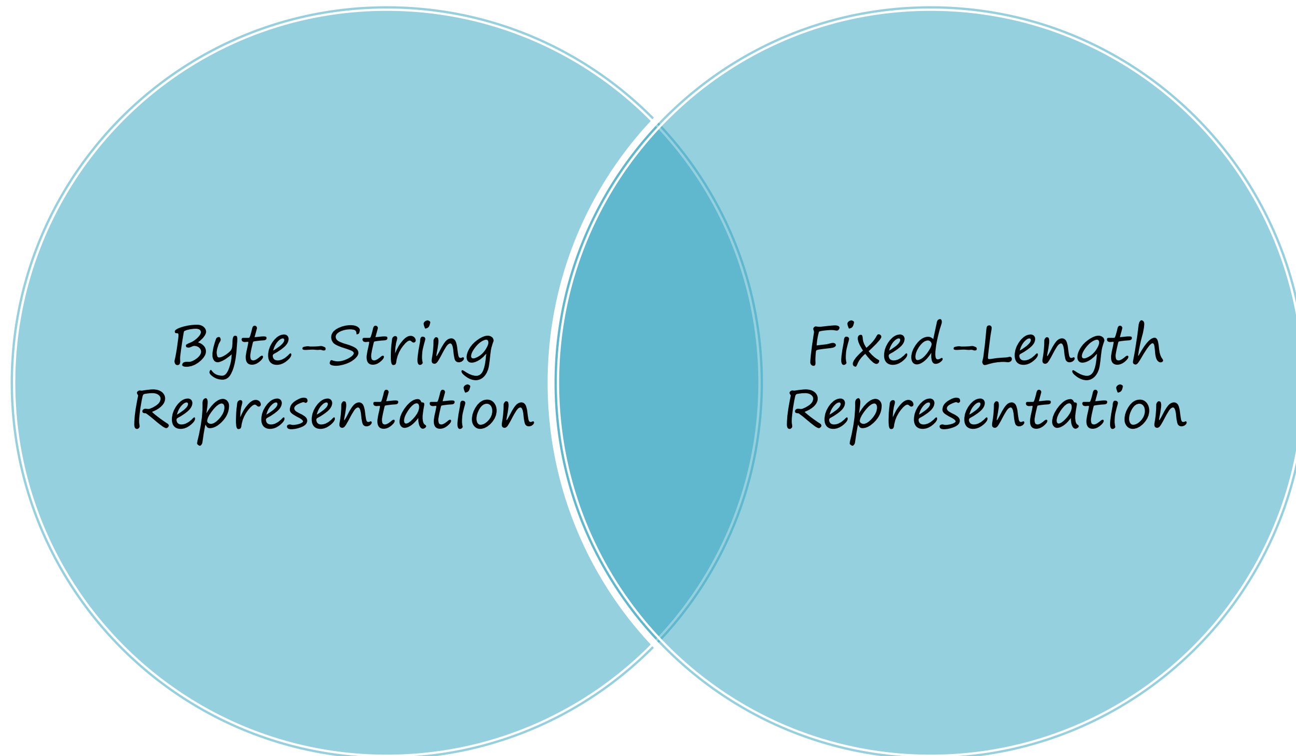


Variable-Length Records

Variable-length records arise in database systems in several ways:

- Record types that allow variable lengths for one or more fields.
- Record types that allow repeating fields (used in some older data models).

Techniques for implementing variable-length records



Byte-String Representation

- A simple method for implementing variable-length records is to attach a special end-of-record (⊥) symbol to the end of each record

0	Perryridge	A-102	400	A-201	900	A-218	700	⊥
1	Round Hill	A-305	350	⊥				
2	Mianus	A-215	700	⊥				
3	Downtown	A-101	500	A-110	600	⊥		
4	Redwood	A-222	700	⊥				
5	Brighton	A-217	750	⊥				

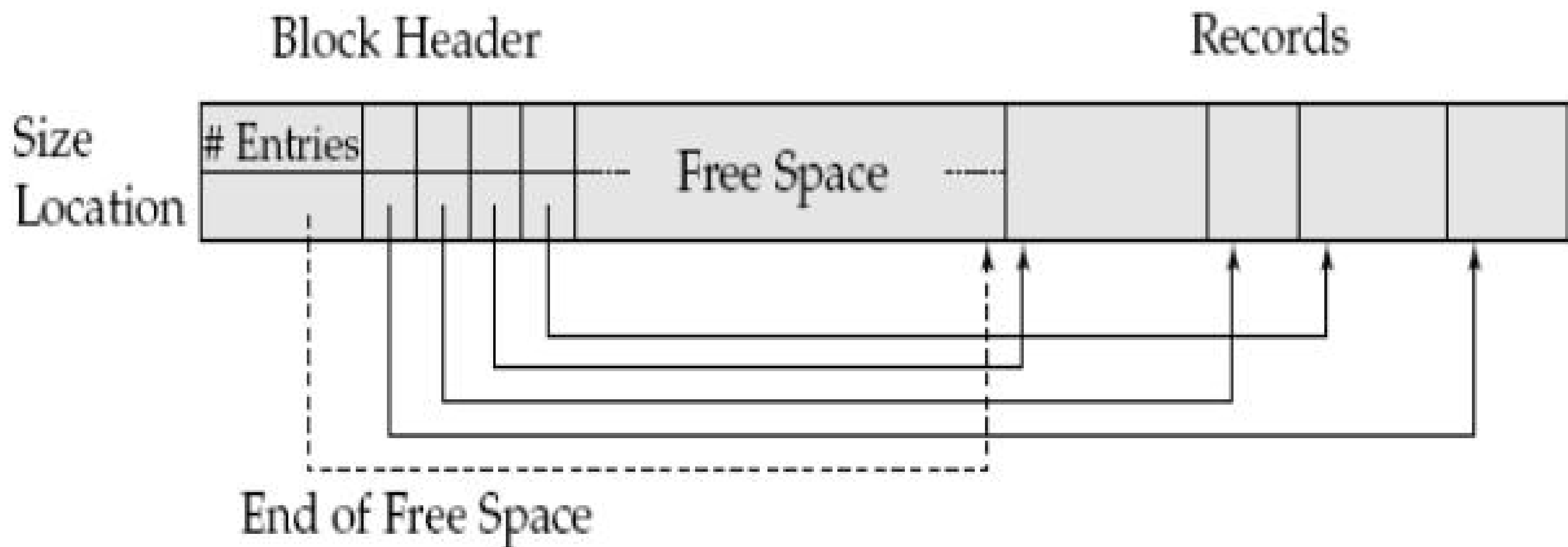
Byte-string representation disadvantages:

It is not easy to reuse space occupied formerly by a deleted record.

There is no space, in general, for records to grow longer.

Slotted-page structure

- A modified form of the byte-string representation, called the **slotted-page structure**, is commonly used for organizing records within a single block.



- There is a header at the beginning of each block, containing the following information:

1. The number of record entries in the header
2. The end of free space in the block
3. An array whose entries contain the location and size of each record

- The actual records are allocated contiguously in the block, starting from the end of the block.
- The free space in the block is contiguous, between the final entry in the header array, and the first record.
- If a record is inserted, space is allocated for it at the end of free space, and an entry containing its size and location is added to the header.
- If a record is deleted, the space that it occupies is freed, and its entry is set to deleted.

Fixed-Length Representation

NRCM
your roots to success...

- Another way to implement variable-length records efficiently in a file system is to use one or more fixed-length records to represent one variable-length record.

There are two ways of doing this:

1. **Reserved space**: If there is a maximum record length that is never exceeded, we can use fixed-length records of that length. Unused space is filled with a special null, or end-of-record, symbol.
2. **List representation**: We can represent variable-length records by lists of fixed length records, chained together by pointers.

Reserved space method

0	Perryridge	A-102	400	A-201	900	A-218	700
1	Round Hill	A-305	350	⊥	⊥	⊥	⊥
2	Mianus	A-215	700	⊥	⊥	⊥	⊥
3	Downtown	A-101	500	A-110	600	⊥	⊥
4	Redwood	A-222	700	⊥	⊥	⊥	⊥
5	Brighton	A-217	750	⊥	⊥	⊥	⊥

Reserved-space method

- The reserved-space method is useful when most records have a length close to the maximum.

- Otherwise, a significant amount of space may be wasted.

List representation method

NRCM
your roots to success...

0	Perryridge	A-102	400	
1	Round Hill	A-305	350	
2	Mianus	A-215	700	
3	Downtown	A-101	500	
4	Redwood	A-222	700	
5		A-201	900	
6	Brighton	A-217	750	
7		A-110	600	
8		A-218	700	

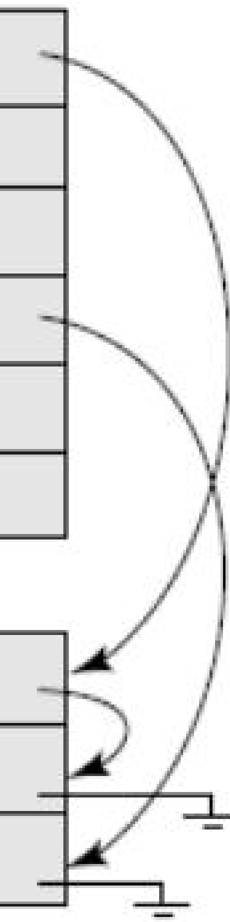
Anchor-block and overflow-block structures

anchor block

Perryridge	A-102	400	
Round Hill	A-305	350	
Mianus	A-215	700	
Downtown	A-101	500	
Redwood	A-222	700	
Brighton	A-217	750	

overflow block

A-201	900	
A-218	700	
A-110	600	



File Organization and Indexing - Clustered Indexes

File organization

- File organization includes the way records and blocks are placed on the storage medium.

- A file **organization is a way of arranging the records in a file when the file is stored** on disk.

- There are two types of file organization
  - Primary File Organizations
  - Secondary File Organizations

Primary File Organizations

- Unordered or Heap or Pile Files
- Ordered or Sorted or sequential Files
- Hash or Direct Files

Unordered or Heap or Pile Files

- Records are placed in the file in the order in which they are inserted.
- Inserting a new record is very efficient.
- Searching can be done by linear search (inefficient).
- Deletion is very inefficient.

Ordered or Sorted or sequential Files

- It store records in sequential order, based on the value of the **search key** of each record.
- An attribute or set of attribute used to look up records in a file is called a **search key**.

Advantages of Ordered Files

- Reading of the records in order of the **ordering field** is extremely efficient, because no sorting is required.

- Finding the next record is fast.

Disadvantages of Ordered Files

- Searches on non-ordering fields are inefficient.
- Insertion and deletion of records are very expensive.

Hash or Direct Files

- Hash function computed on some attribute of each record; the result specifies where record should be placed.
- The pages in a hashed file are grouped into buckets.
- Given a bucket number, the hashed file structure allows us to find the primary page for that bucket.
- The bucket to which a record belongs can be determined by applying a special function called a hash function, to the search field(s).
- Insertions and deletions are fast.

Secondary File Organizations

- Secondary file organization uses the index to access the records.
- An index is a data structure that speeds up certain operations on a file.
- An index for a file in a database system works in the same way as the index in any textbook.
- If we want to learn about a particular topic (specified by a word or a phrase) , we can search for the topic in the index at the back of the book.
- Indexes provide faster access to data.

Types of Indexes

- **Single-level ordered indexes**
  - Primary indexes
  - Secondary indexes
  - Clustering indexes
- **Multi-level Indexes**
- **Dynamic Multi-level indexes using B-trees and B+-trees**

Primary indexes

- An index is called a primary index if the search key includes the primary key.
- A **Primary Index** is constructed of two parts: The **first field** is the same data type of the **primary key** of a file block of the data file and the **second field** is file **block pointer**.

INDEX FILE
(<K(i), P(i)> entries)

BLOCK ANCHOR PRIMARY KEY VALUE	BLOCK POINTER
Aaron, Ed	•
Adams, John	•
Alexander, Ed	•
Allen, Troy	•
Anderson, Zach	•
Arnold, Mack	•
⋮	
⋮	
⋮	
Wong, James	•
Wright, Pam	•

DATA FILE

NAME (PRIMARY KEY FIELD)	SSN	BIRTHDATE	JOB	SALARY	SEX
Aaron, Ed					
Abbott, Diane					
⋮					
Acosta, Marc					
Adams, John					
Adams, Robin					
⋮					
Akers, Jan					
Alexander, Ed					
Alfred, Bob					
⋮					
Allen, Sam					
Allen, Troy					
Anders, Keith					
⋮					
Anderson, Rob					
Anderson, Zach					
Angeli, Joe					
⋮					
Archer, Sue					
Arnold, Mack					
Arnold, Steven					
⋮					
Atkins, Timothy					
⋮					
Wong, James					
Wood, Donald					
⋮					
Woods, Manny					
Wright, Pam					
Wyatt, Charles					
⋮					
Zimmer, Byron					

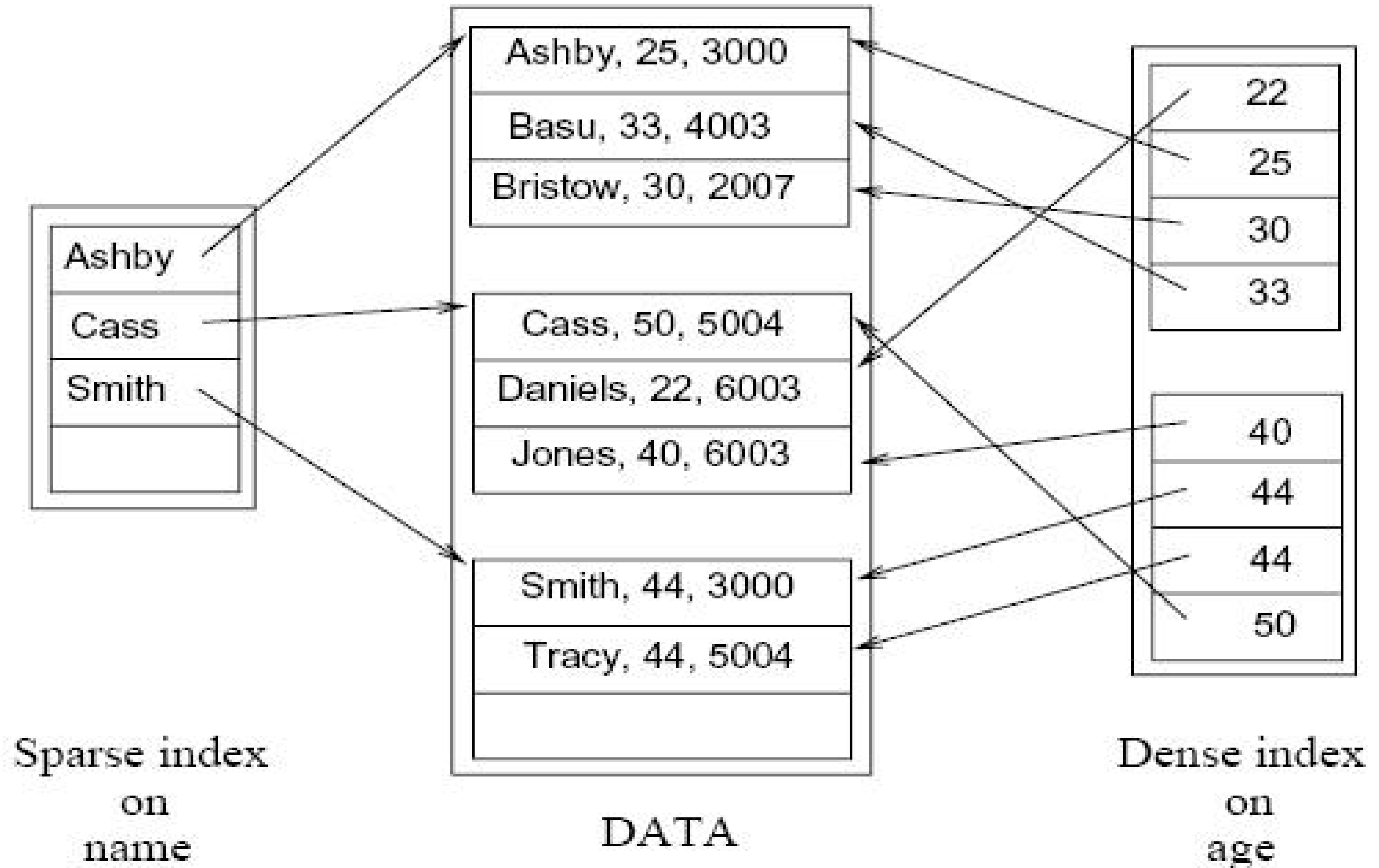
Problem with a primary index

A major problem with a primary index—as with any ordered file—is insertion and deletion of records.

Indexes can also be characterized as

- Dense: A **dense index has an index entry for every** search key value (and hence every record) in the data file.
- Sparse (nondense): A **sparse (or nondense) index, on the other** hand, has index entries for only some of the search values.
- A primary index is hence a nondense (sparse) index, since it includes an entry for each disk block of the data file rather than for every search value (or every record).

Sparse versus Dense Indexes



Clustering Indexes

- If records of a file are physically ordered on a non key field—which does not have a distinct value for each record—that field is called the **clustering field.**

- A clustering index is also an ordered file with two fields; the first field is of the same type as the clustering field of the data file, and the second field is a block pointer.

Primary Index
and
Secondary Index

Primary indexes

- An index is called a primary index if the search key includes the primary key.
- A **Primary Index** is constructed of two parts: The **first field** is the same data type of the **primary key** of a file block of the data file and the **second field** is file **block pointer**.

DATA FILE

(PRIMARY KEY FIELD)

NAME	SSN	BIRTHDATE	JOB	SALARY	SEX
Aaron, Ed					
Abbott, Diane					
⋮					
Acosta, Marc					
Adams, John					
Adams, Robin					
⋮					
Akers, Jan					
Alexander, Ed					
Alfred, Bob					
⋮					
Allen, Sam					
Allen, Troy					
Anders, Keith					
⋮					
Anderson, Rob					
Anderson, Zach					
Angeli, Joe					
⋮					
Archer, Sue					
Arnold, Mack					
Arnold, Steven					
⋮					
Atkins, Timothy					
⋮					
Wong, James					
Wood, Donald					
⋮					
Woods, Manny					
Wright, Pam					
Wyatt, Charles					
⋮					
Zimmer, Byron					

INDEX FILE
(<K(i), P(i)> entries)

BLOCK ANCHOR PRIMARY KEY VALUE	BLOCK POINTER
Aaron, Ed	•
Adams, John	•
Alexander, Ed	•
Allen, Troy	•
Anderson, Zach	•
Arnold, Mack	•
⋮	
⋮	
Wong, James	•
Wright, Pam	•

Problem with a primary index

A major problem with a primary index as with any ordered file is insertion and deletion of records.

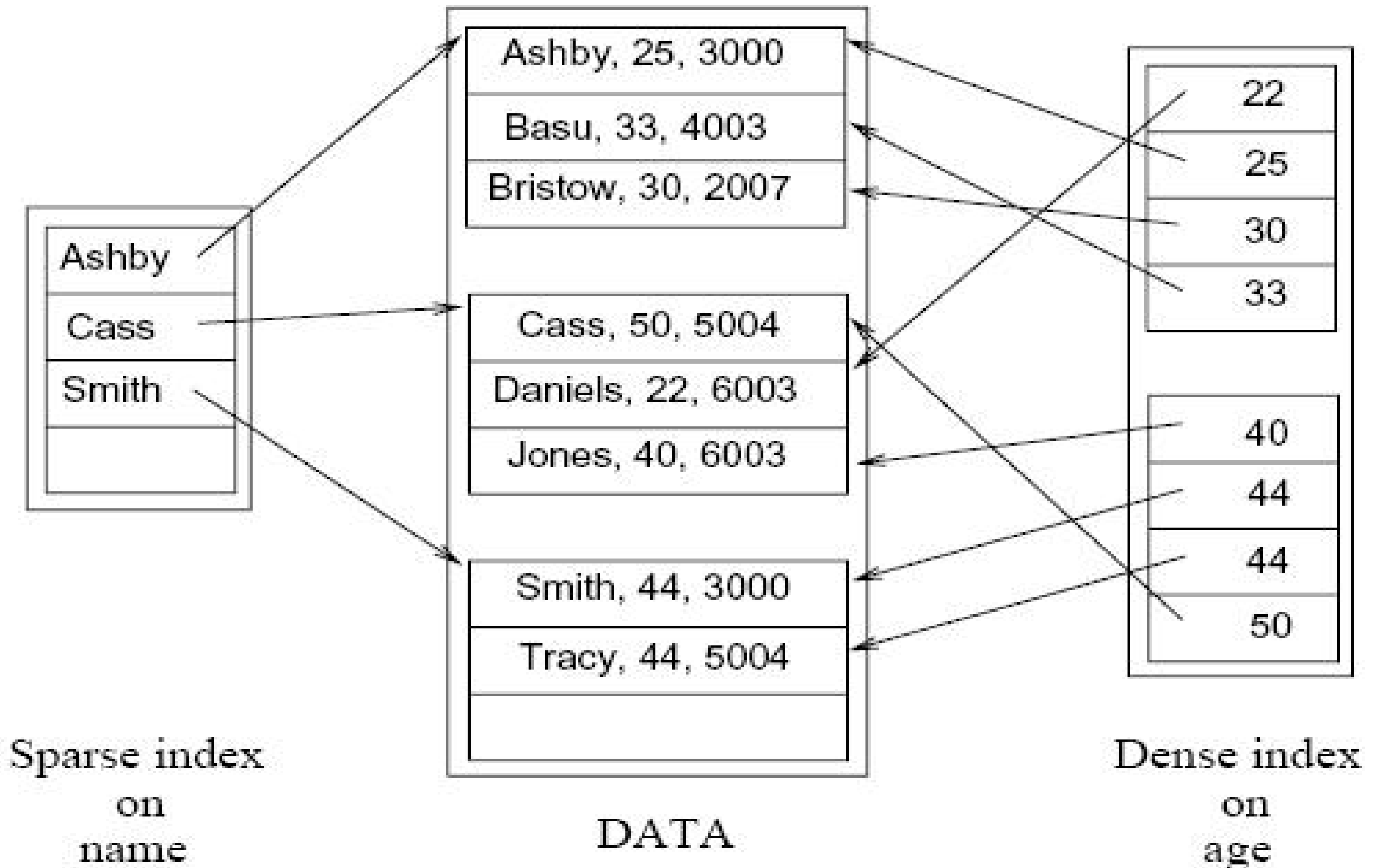
Indexes can also be characterized as

- Dense: A **dense index has an index entry for every** search key value (and hence every record) in the data file.

- Sparse (non-dense): A **sparse (or non-dense) index, on the other** hand, has index entries for only some of the search values.

- A primary index is hence a non-dense (sparse) index, since it includes an entry for each disk block of the data file rather than for every search value (or every record).

Sparse versus Dense Indexes



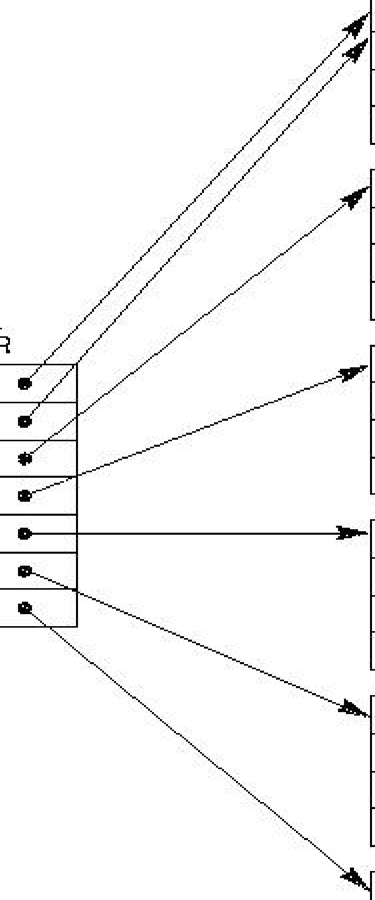
Clustering Indexes

- If records of a file are physically ordered on a non-key field—which does not have a distinct value for each record—that field is called the **clustering field.**

- A clustering index is also an ordered file with two fields; the first field is of the same type as the clustering field of the data file, and the second field is a block pointer.

DATA FILE

INDEX FILE
(<K(i), P(i)> entries)



CLUSTERING FIELD VALUE	BLOCK POINTER
1	•
2	•
3	•
4	•
5	•
6	•
8	•

DEPTNUMBER	NAME	SSN	JOB	BIRTHDATE	SALARY
1					
1					
1					
2					
2					
3					
3					
3					
3					
3					
4					
4					
5					
5					
5					
5					
6					
6					
6					
6					
6					
8					
8					
8					

Secondary Indexes

- A **Secondary Index** is an **ordered file** with two fields.
- The **first** is of the same data type as some **non-ordering field** and the second is either a block or a record pointer.
- If the **entries** in this non-ordering field **must** be **unique** this field is sometime referred to as a **Secondary Key**. This results in a dense index.

INDEX FILE
(<K(i), P(i)> entries)

INDEX FIELD VALUE	BLOCK POINTER
1	•
2	•
3	•
4	•
5	•
6	•
7	•
8	•
9	•
10	•
11	•
12	•
13	•
14	•
15	•
16	•
17	•
18	•
19	•
20	•
21	•
22	•
23	•
24	•

DATA FILE

INDEXING FIELD (SECONDARY KEY FIELD)

	INDEXING FIELD			
	9			
	5			
	13			
	8			
	6			
	15			
	3			
	17			
	21			
	11			
	16			
	2			
	24			
	10			
	20			
	1			
	4			
	23			
	18			
	14			
	12			
	7			
	19			
	22			

Comparison between indexes

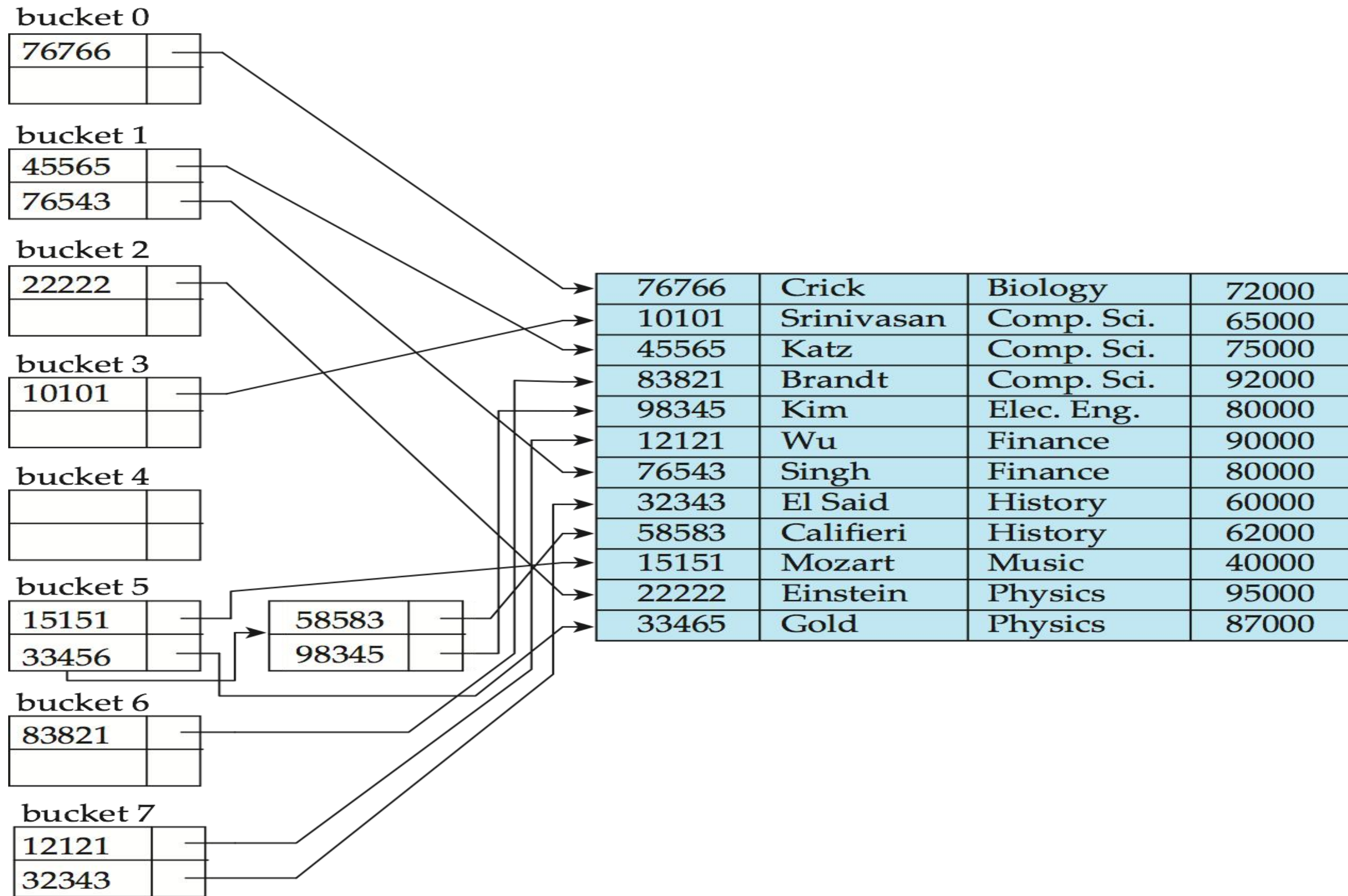
	Number of (First-level) Index Entries	Dense or Nondense	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no
Secondary (key)	Number of records in data file	Dense	No

Index data Structures – Hash Based Indexing

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.
- 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*

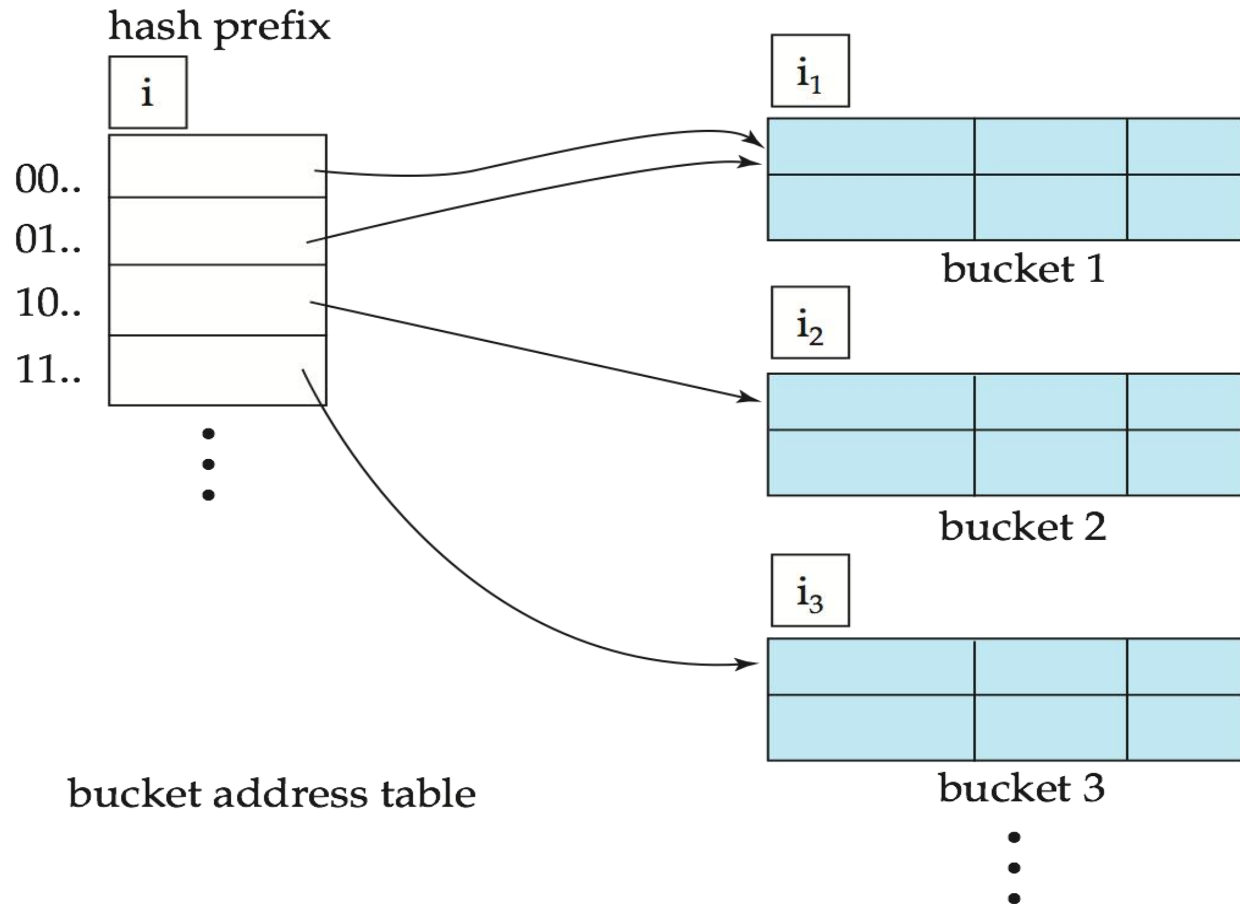
Deficiencies of Static Hashing

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

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 $b = 32$.
  - At any time use only a prefix of the hash function to index into a table of bucket addresses.
  - 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 (why?)
  - Thus, actual number of buckets is < 2^i
    - The number of buckets also changes dynamically due to coalescing and splitting of buckets.

General Extendable Hash Structure



In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$ (see next slide for details)

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
- 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

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*)
  - Re-compute 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

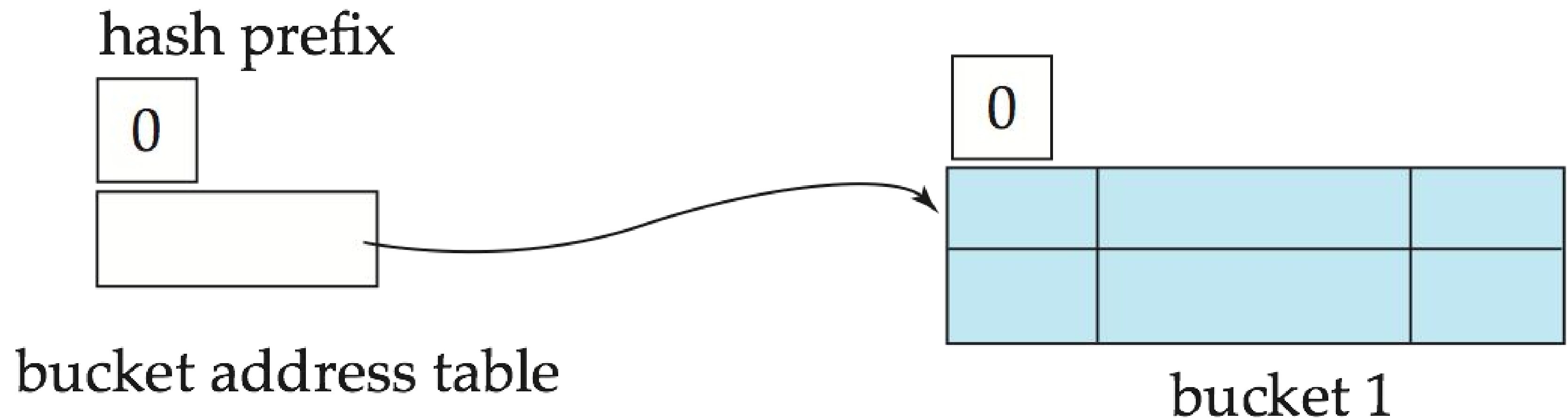
- To delete a key value,
  - locate it in its bucket and remove it.
  - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
  - Coalescing of buckets can be done (can coalesce only with a "*buddy*" bucket having same value of i_j and same $i_j - 1$ 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

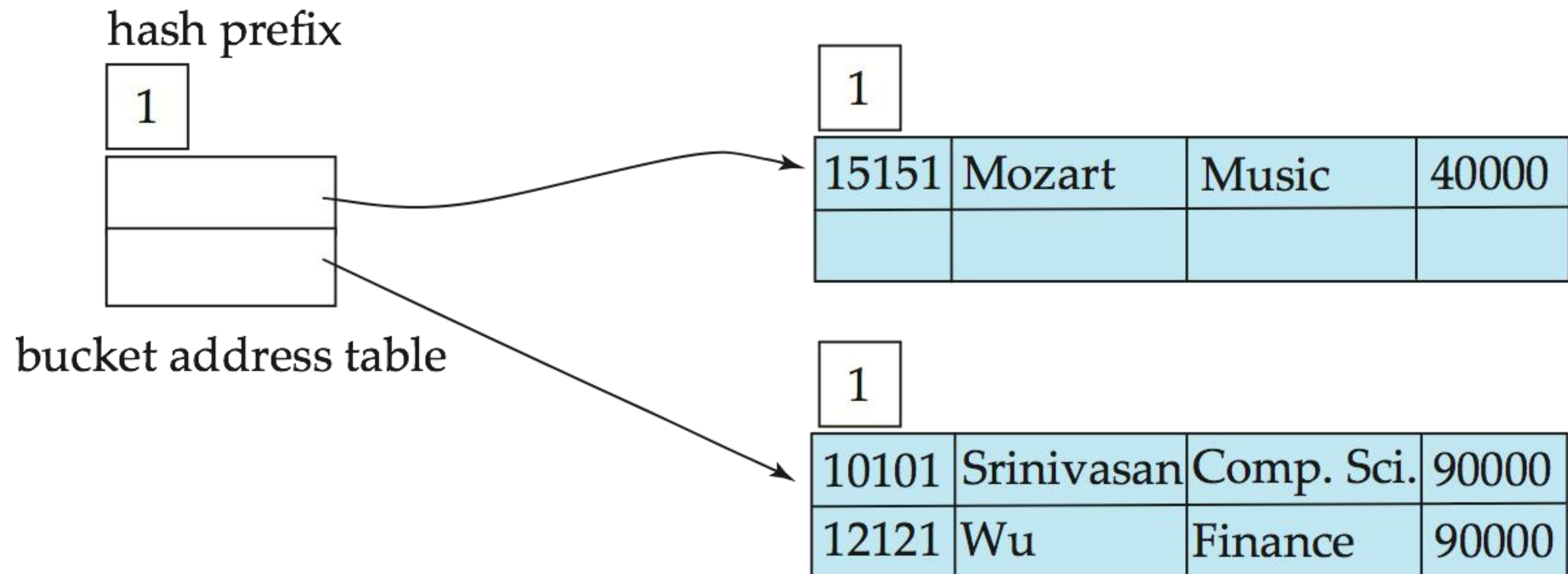
Example (Cont.)

- Initial Hash structure; bucket size = 2



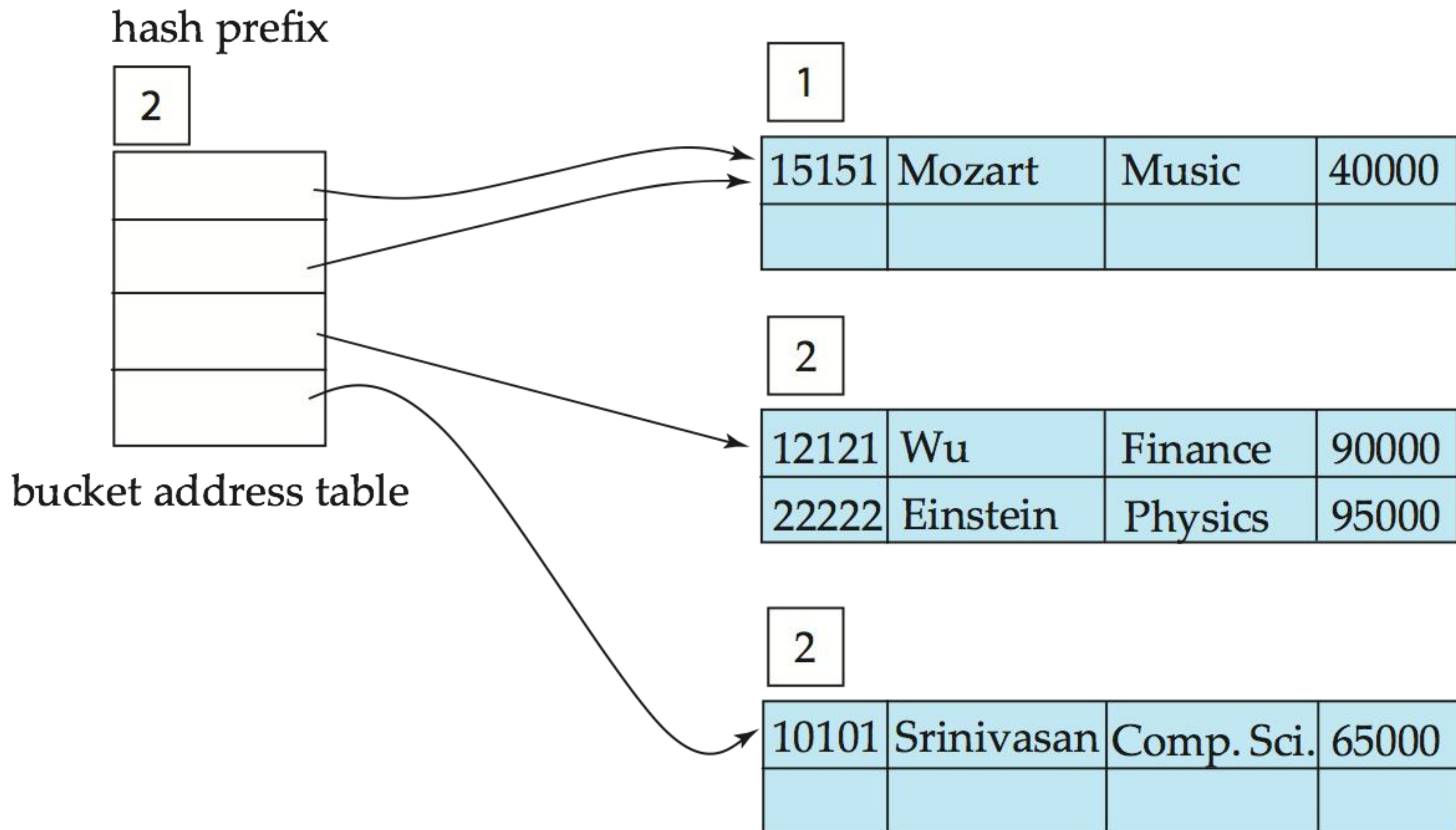
Example (Cont.)

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



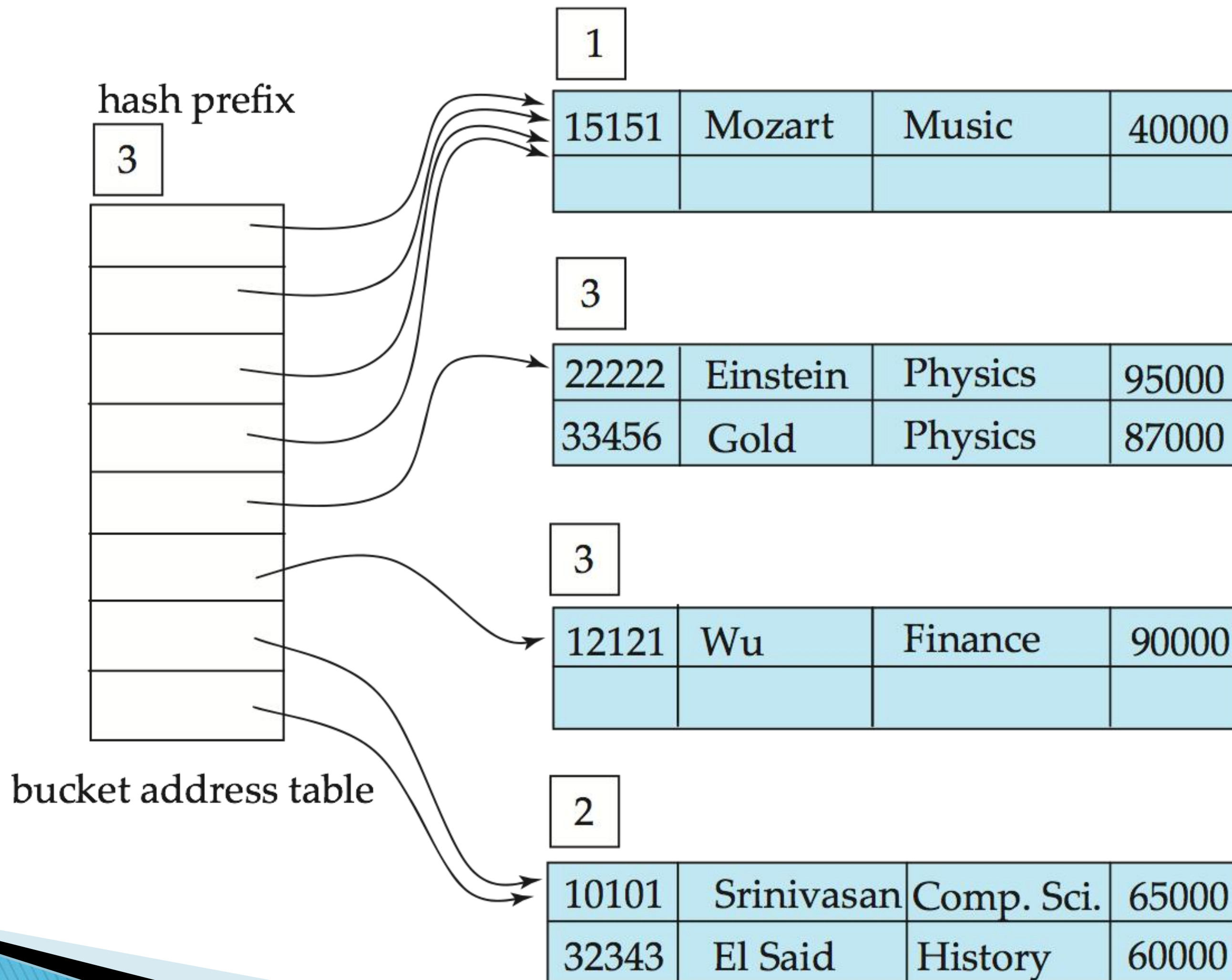
Example (Cont.)

- Hash structure after insertion of Einstein record



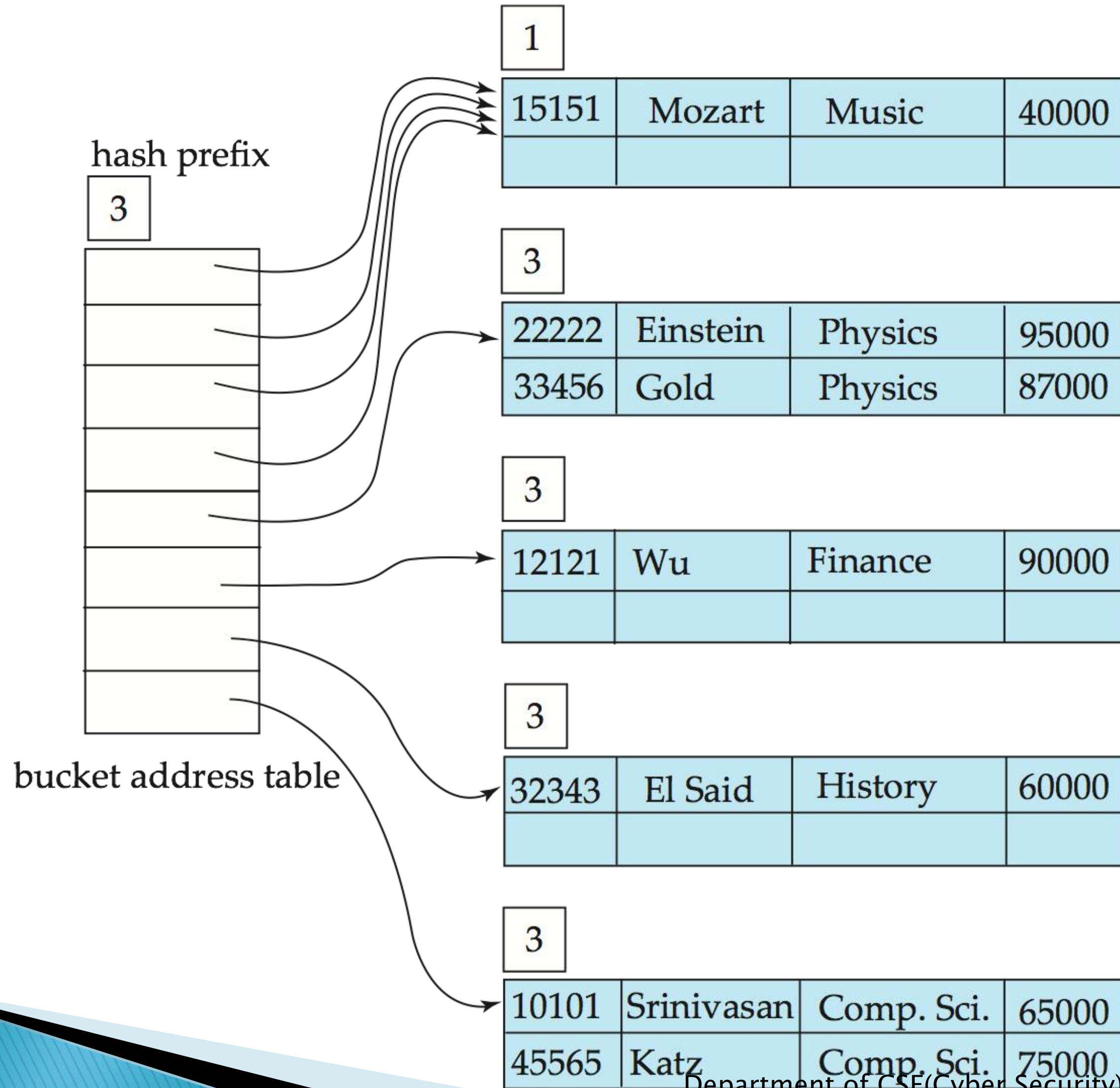
Example (Cont.)

- Hash structure after insertion of Gold and El Said records

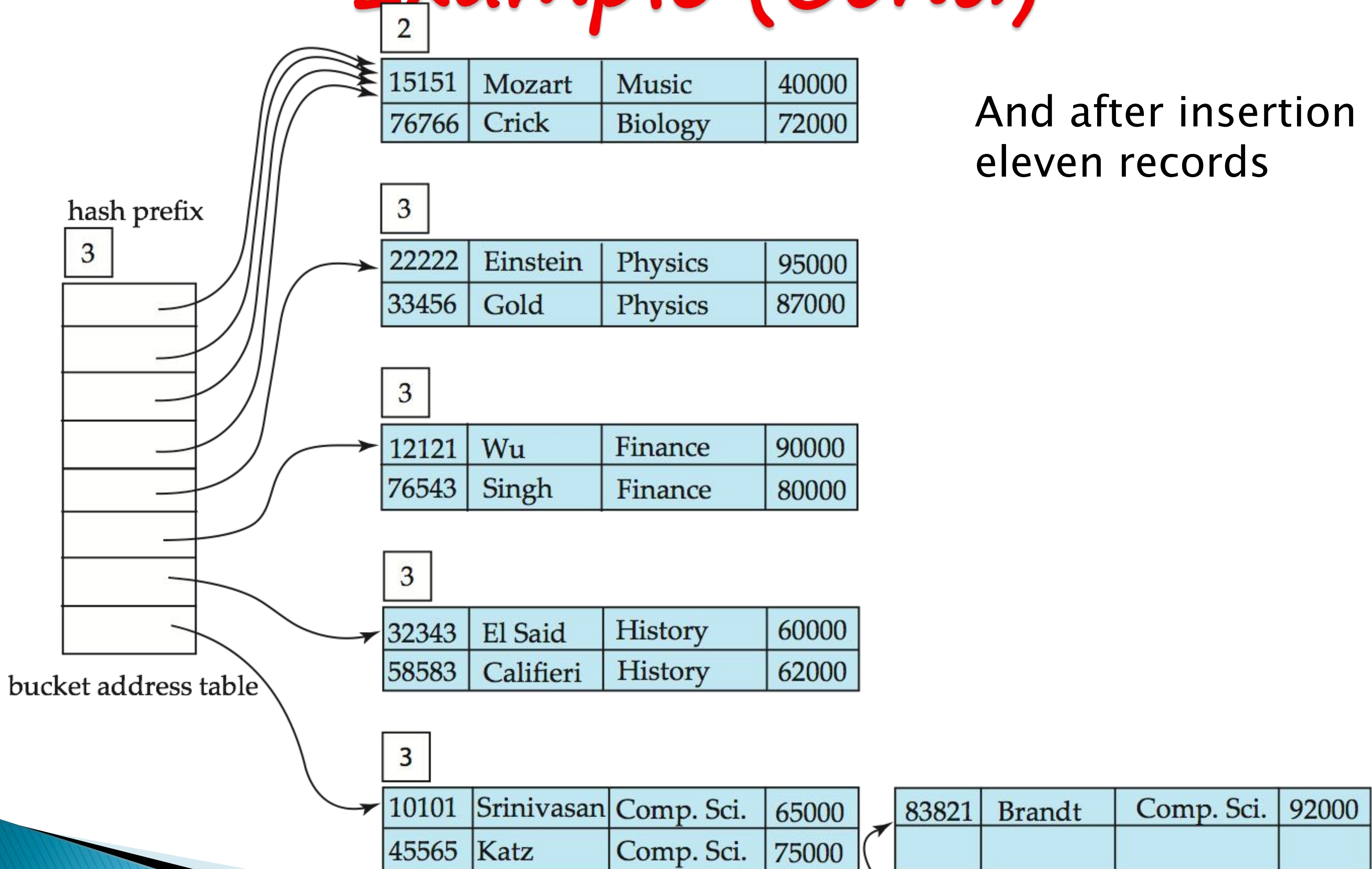


Example (Cont.)

- Hash structure after insertion of Katz record

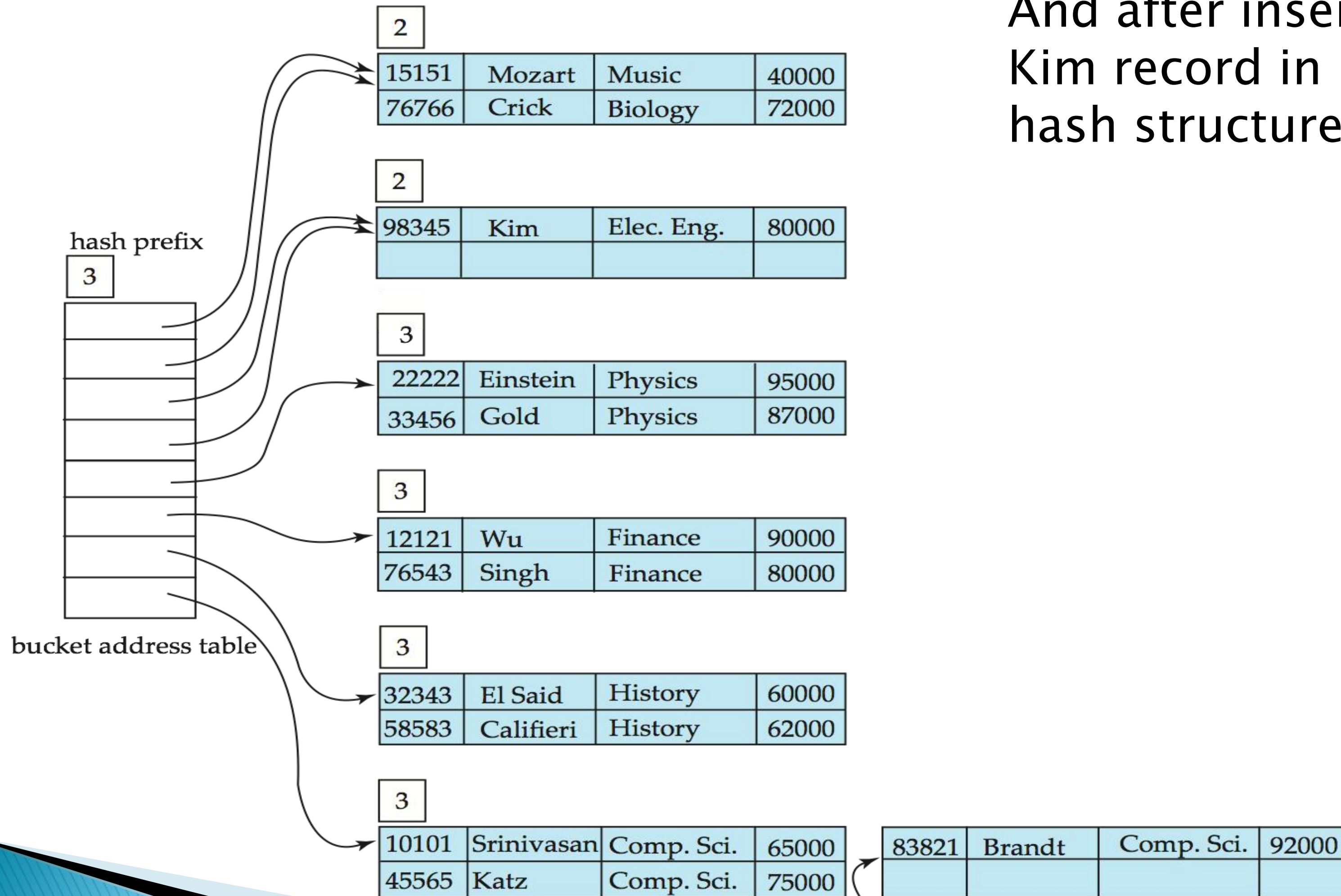


Example (Cont.)



And after insertion of eleven records

Example (Cont.)



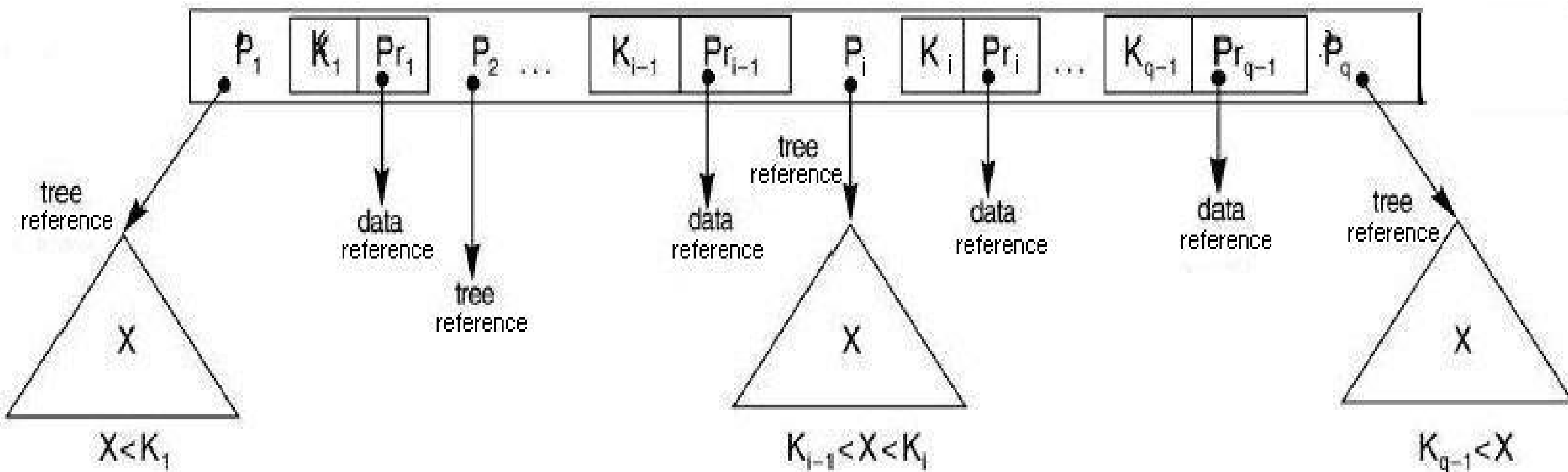
And after insertion of Kim record in previous hash structure

B tree

A B-tree of order m (the maximum number of children for each node) is a tree which satisfies the following properties:

- Every node has at most m children.
- Every node (except root and leaves) has at least $m/2$ children.
- The root has at least two children if it is not a leaf node.
- All leaves appear in the same level, and carry information.
- A non-leaf node with k children contains k–1 keys.
- B-trees are always height balanced, with all leaf nodes at the same level.
- Update and search operations affect only a few disk pages, so performance is good.

The node structure of B tree

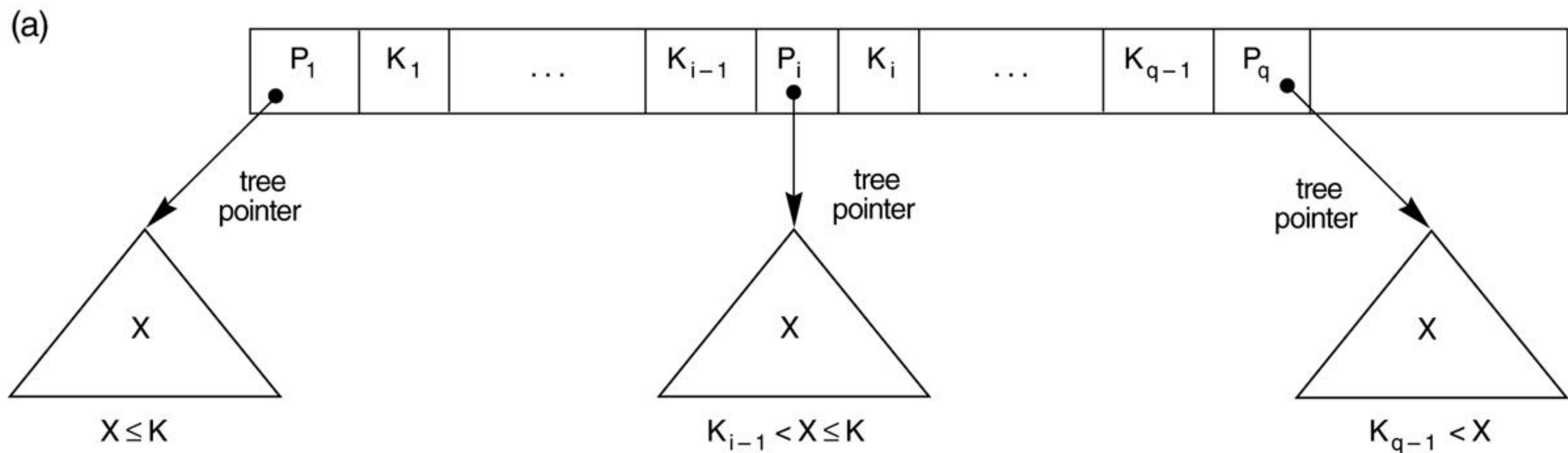


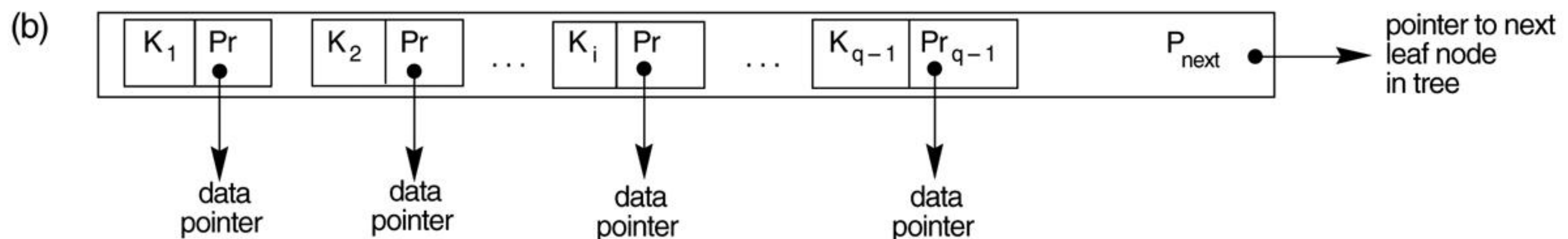
- Note:
  - Corresponding to each key there is a data reference that refers to the data record for that key in secondary memory.
  - In our representations we will omit the data references.

B^+ tree

- Similar to B trees, with a few slight differences.
- B^+ tree is a balanced tree in which the internal nodes direct the search and the leaf nodes contain the data entries.

The nodes of a B+-tree. (a) Internal node of a B+-tree with q -1 search values. (b) Leaf node of a B+-tree with q – 1 search values and q – 1 data pointers.





Difference between B-tree and B+-tree

- In a B-tree, pointers to data records exist at all levels of the tree. In a B+-tree, all pointers to data records exists at the leaf-level nodes
- A B+-tree can have less levels (or higher capacity of search values) than the corresponding B-tree.
- In a B-tree insertion and deletion are simple process. In B+ tree insertion and deletion are complex process.
- In a B-tree, internal nodes store both the keys and data value. In B+ tree Internal nodes store just keys.

Comparison of File Organizations

Comparing File Organizations

- Heap files (random order; insert at eof)
- Sorted files, sorted on *<age, sal>*
- Clustered B+ tree file, Alternative (1), search key *<age, sal>*
- Heap file with unclustered B + tree index on search key *<age, sal>*
- Heap file with unclustered hash index on search key *<age, sal>*

Operations to Compare

- Scan: Fetch all records from disk
- Equality search (e.g., “age = 30”)
- Range selection (e.g., “age > 30”)
- Insert a record
- Delete a record

Parameters of the Analysis

	B = # data pages	R = #records/page	D = disk page I/O time	C = process single record	H = apply Hash function	F = index tree fan-out
Typical value			15 mlsec	100 nanosec	100 nanosec	100

Assumptions in Our Analysis

- Heap Files:
  - Equality selection on key; exactly one match.
- Sorted Files:
  - Files compacted after deletions.
  - Clustered files: pages typically 67% full.
  - ⇒ Total number pages needed = 1.5 B.
- Indexes:
  - Alt (2), (3): data entry size = 10% size of record
  - Hash: No overflow buckets.
    - 80% page occupancy.
    - ⇒ Index size = 1.25 B data size.
    - ⇒ #data entries/page = 10 (0.8R) = 8R.
  - Tree: 67% page occupancy of index pages (this is typical).
    - ⇒ #leaf pages = (1.5 B) 0.1 = 0.15 B.
    - ⇒ #data entries/page = 10 (0.67R) = 6.7R.

Scanning Cost

- Heap file: B(D + RC).
  - for each page (B)
  - Read the page (D)
  - For each record (R), process the record (C).
- Sorted File: B(D + RC).
  - Have to go through all pages.
- Clustered File: 1.5B (D+RC).
  - Pages only 67% full.
- Unclustered Tree Index: >BR(D+C). Bad!
  - for each record (BR)
  - retrieve page and find record (D + C).

Exercise for Group Work

1. Estimate how long an equality search takes in
   (i) a heap file (ii) a sorted file (iii) a hash file, hashed on the search key, with at most one record matching the search key (i.e., the search is on a key field).

2. Estimate how long an insertion takes in
   (i) a heap file (ii) a sorted file (iii) a hash file.

	B = # data pages	R = #records/page	D = disk page I/O time	C = process single record	H = apply Hash function	F = index tree fan-out
Typical value			15 msec	100 nanosec	100 nanosec	100

Cost of Operations

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap					
(2) Sorted					
(3) Clustered					
(4) Unclustered Tree index					
(5) Unclustered Hash index					

Indexes and Performance Tuning

8.5 Indexes and Performance Tuning

- One of a DBA's most important duties is to deal with performance problems
- One of the most effective methods for improving performance is to choose the best indexes for the given workload
- Why not index everything?
  - What are the two costs of an index?
- Part of a DBA's job: choose indexes so the database will "run fast".
  - What information does the DBA need, to do this?

Choice of Indexes

- What indexes should we create?
  - Which relations should have indexes? What field(s) should be the search key? Should we build several indexes?
- What kinds of indexes should we create?
  - Clustered? Hash/tree?
- Before creating an index, must also consider the impact on updates in the workload!
  - Trade-off: Indexes can make queries go faster, updates slower. Require disk space, too.
- Also consider dropping indexes

8.5.1 Understanding the Workload

- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
- For each update in the workload:
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

Composite Search Keys

- To retrieve Emp records with *age*=30 AND *sal*=4000, an index on <*age,sal*> would be better than an index on *age* or an index on *sal*.
    - Choice of index key orthogonal to clustering etc.
- If condition is: $20 < age < 30$ AND $3000 < sal < 5000$:
    - Clustered tree index on <*age,sal*> or <*sal,age*> is best.
- If condition is: *age*=30 AND $3000 < sal < 5000$:
    - Clustered <*age,sal*> index much better than <*sal,age*> index!
- Composite indexes are larger, updated more often.

Indexes in the real world

- Types of indexes supported
  - Oracle, SQLServer and DB2 support only B+Tree indexes. Postgres supports hash indexes but does not recommend using them.
  - MySQL?
  - Everyone uses hash indexes for hash joins, but they are constructed on the fly.

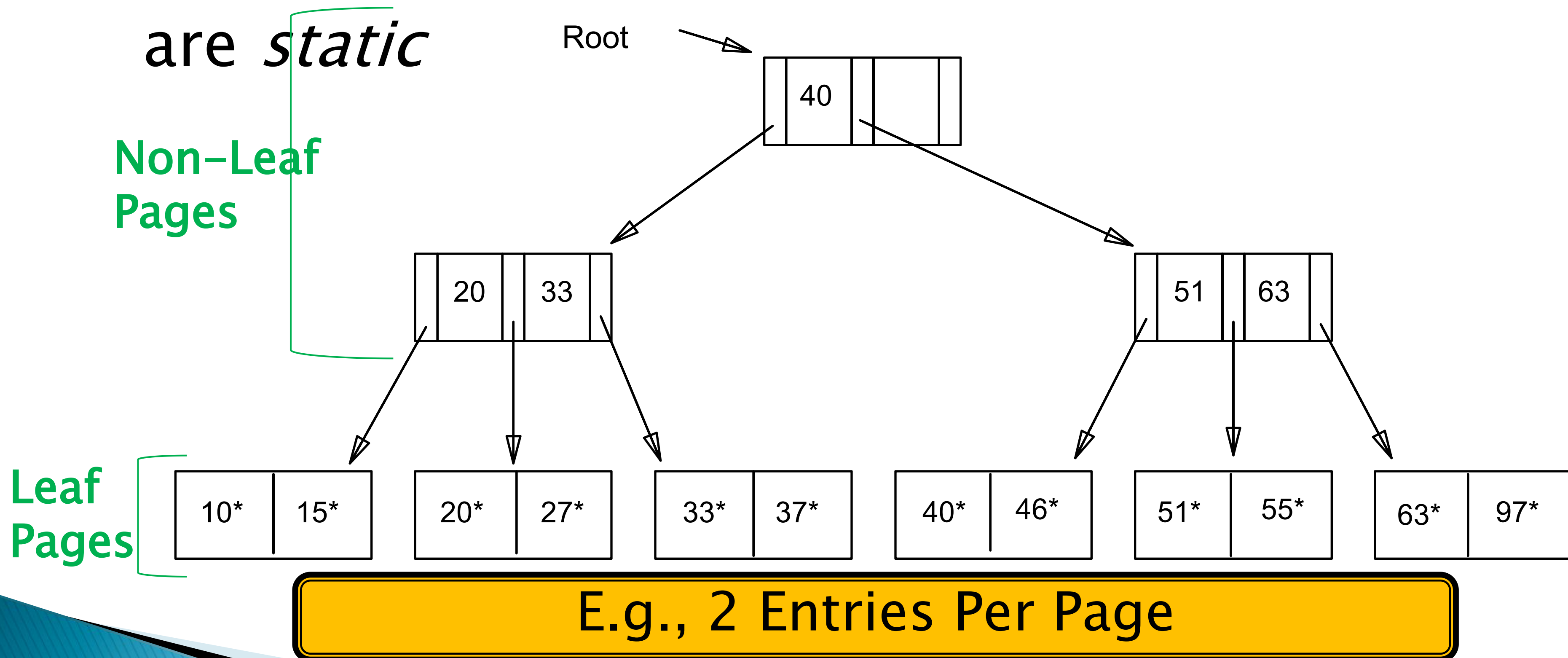
Clustering in the real world

- Oracle, SQL Server:
  - Declares a clustered index on any primary key.
  - It uses alternative 1.
- DB2, Postgres
  - The user can define an index to be clustered.
  - The DBMS uses alternative 2.
  - At first the index will be clustered.
  - If you also specify a percentage of free space, it will place subsequent inserts/updates (Postgres:updates only) in sorted order so the index should stay clustered for a while.
  - It's up to you to recluster the index if overflow chains get to be long.
- MySQL

Indexed Sequential Access Methods (ISAM)

ISAM Trees

- Indexed Sequential Access Method (ISAM) trees are *static*

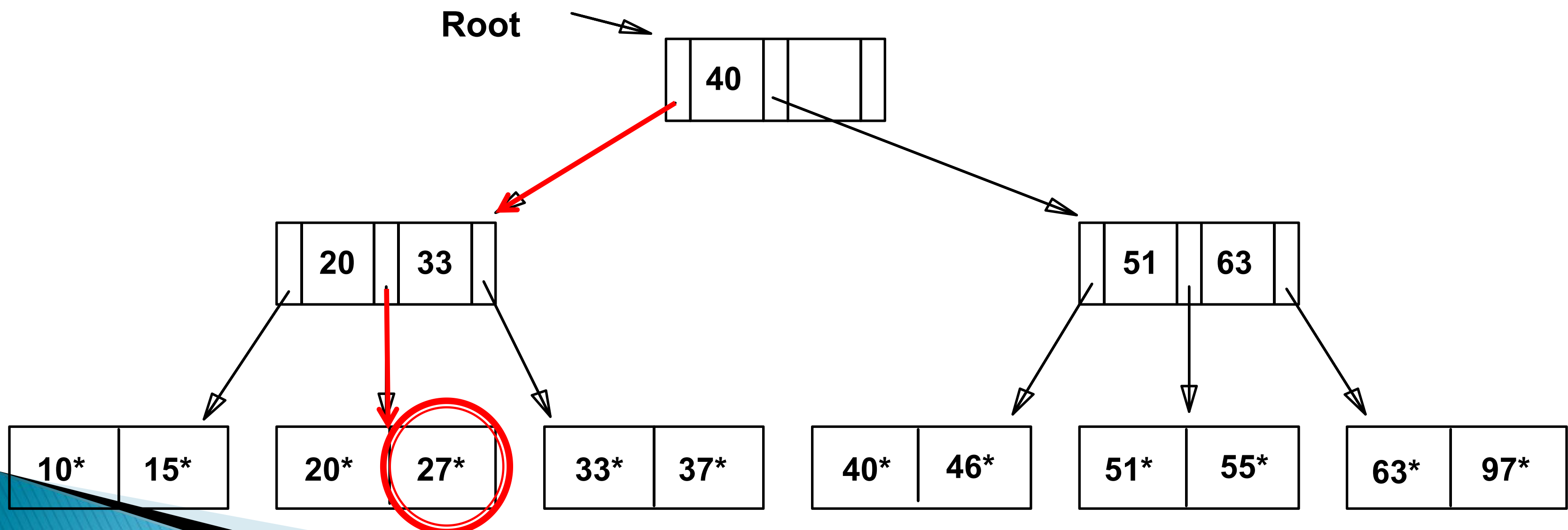


ISAM File Creation

- How to create an ISAM file?
  - All leaf pages are allocated *sequentially* and *sorted* on the search key value
  - If Alternative (2) or (3) is used, the data records are created and *sorted* before allocating leaf pages
  - The non-leaf pages are subsequently allocated

ISAM: Searching for Entries

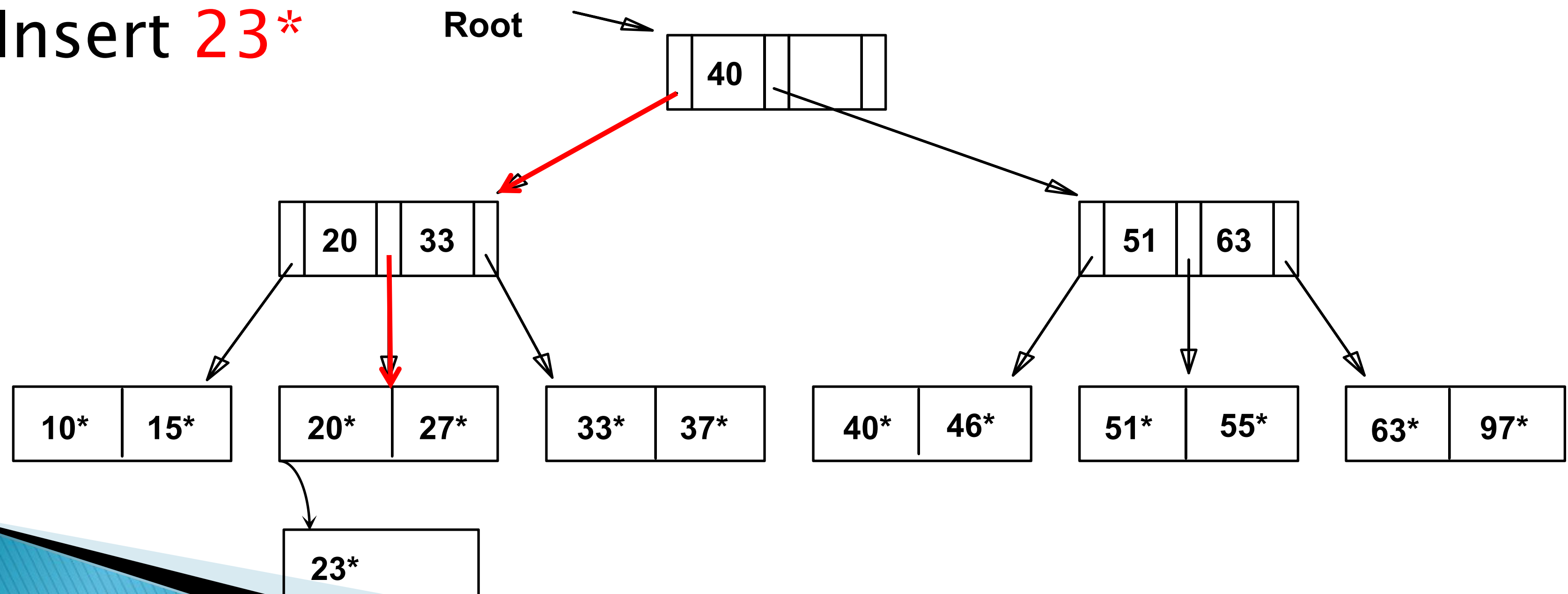
- Search begins at root, and key comparisons direct it to a leaf
- Search for 27*



ISAM: Inserting Entries

- The appropriate page is determined as for a search, and the entry is inserted (with overflow pages added if necessary)

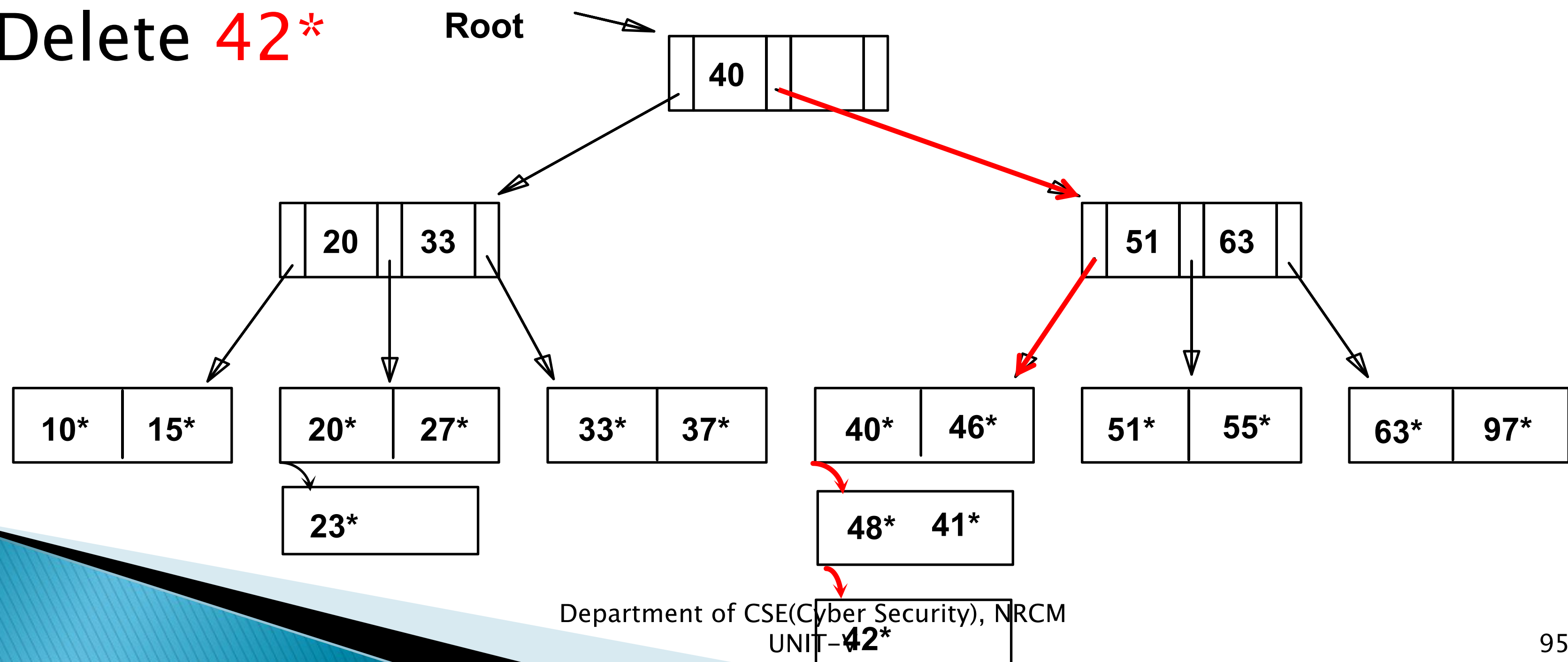
- Insert 23*



ISAM: Deleting Entries

- The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)

- Delete 42*



ISAM: Some Issues

- Once an ISAM file is created, insertions and deletions affect only the contents of leaf pages (i.e., *ISAM is a static structure*!)
- Since index-level pages are *never* modified, there is no need to *lock* them during insertions/deletions
  - Critical for concurrency!
- Long overflow chains can develop easily
  - The tree can be initially set so that ~20% of each page is free
- If the data distribution and size are relatively static, ISAM might be a good choice to pursue!

NRCM
your roots to success...

THANK YOU