CS354: DATABASE

Storage, File Organization, Indexing & Hashing

Classification of Physical Storage Media

- Speed with which data can be accessed
- Cost per unit of data
- Reliability
  - data loss on power failure or system crash
  - physical failure of the storage device
- Can differentiate storage into:
  - **volatile storage:**
    - loses contents when power is switched off
  - **non-volatile storage**:
    - Contents persist even when power is switched off.
    - Includes secondary and tertiary storage, as well as battery backed up main-memory.

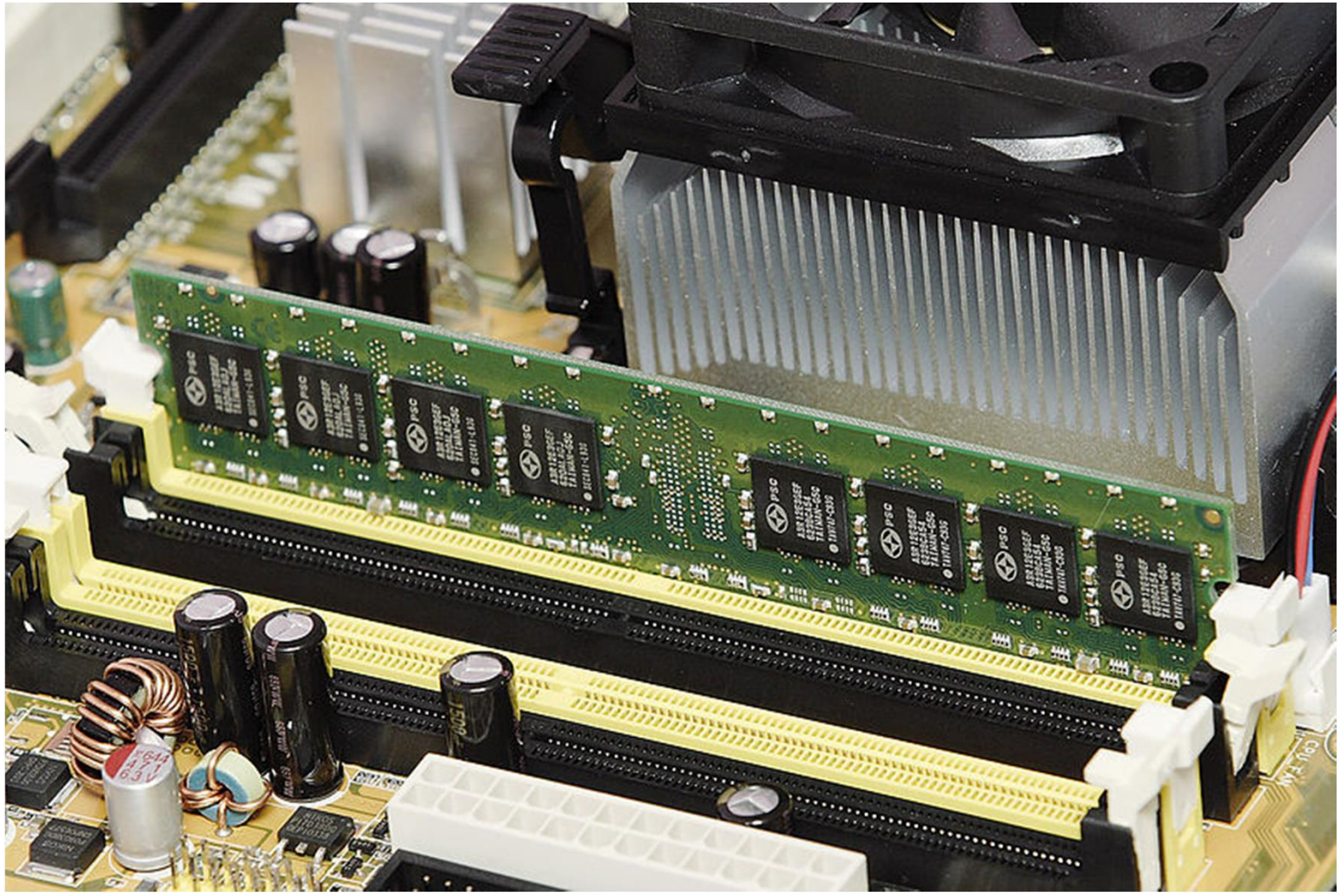
CACHE

- Fastest and most costly form of storage
- Volatile in nature
- Managed by the computer system hardware
- This is to compensate the speed difference between the main memory access time and processor logic.

Main Memory

- Fast access (10s to 100s of nanoseconds; 1 nanosecond = 10^{-9} seconds)
- Generally too small (or too expensive) to store the entire database
- Capacities of up to a few Gigabytes widely used currently
- Capacities have gone up and per-byte costs have decreased steadily and rapidly (roughly factor of 2 every 2 to 3 years)
  - **Volatile** — contents of main memory are usually lost if a power failure or system crash occurs.

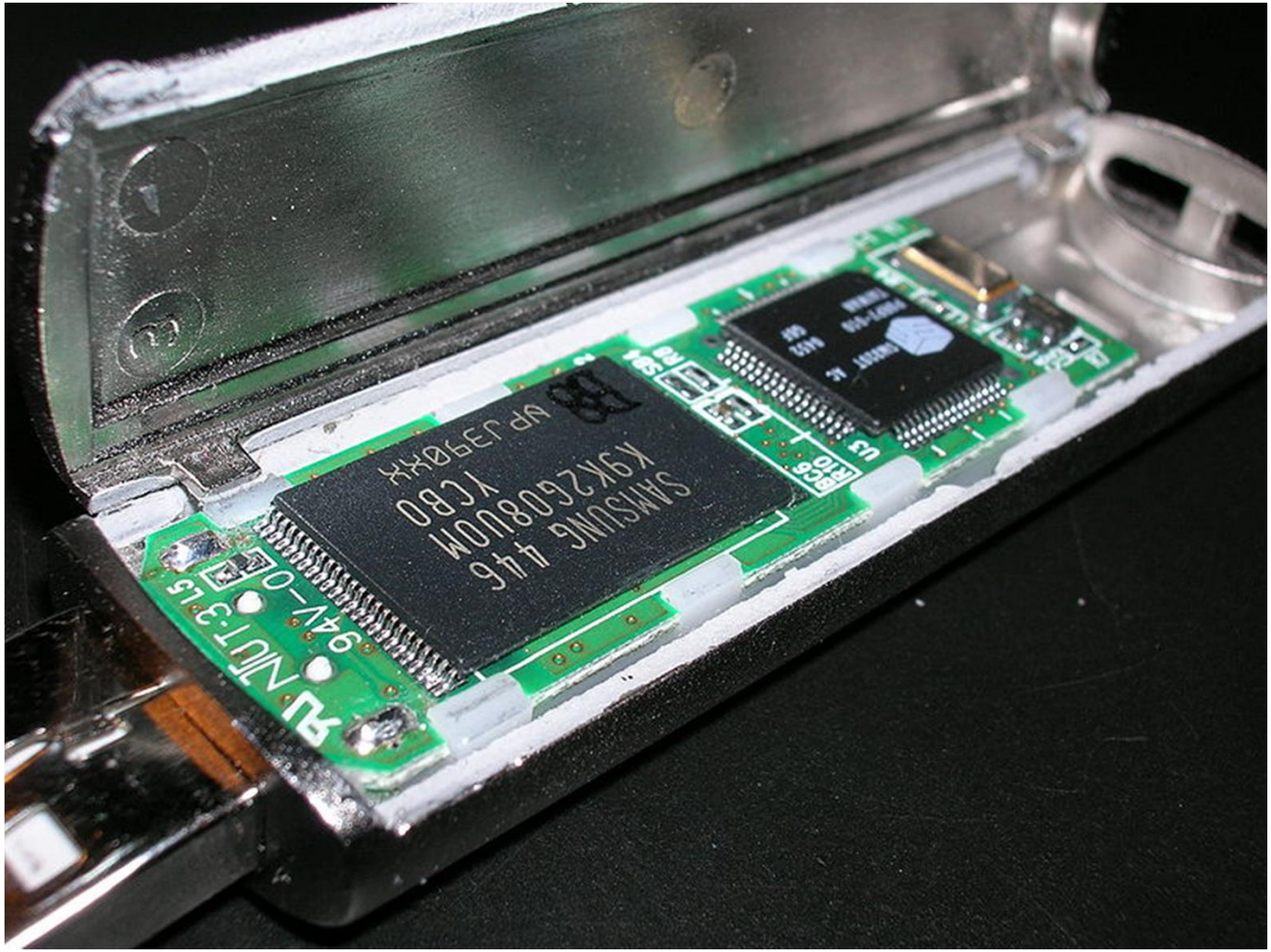
Main Memory



Flash Memory

- Data survives power failure
- Data can be electrically erased and reprogrammed
- Can support only a limited number (around1M) of write/erase cycles.
- Reads are roughly as fast as main memory
- But writes are slow (few microseconds), erase is slower
- Widely used in embedded devices such as digital cameras, phones, and USB keys

Flash Memory



Magnetic Disk

- Much slower access than main memory
- Data is stored on spinning disk, and read/written magnetically
- Primary medium for the long-term storage of data; typically stores entire database.
- Data must be moved from disk to main memory for access, and written back for storage
  - **direct-access** – possible to read data on disk in any order, unlike magnetic tape
  - Capacities range up to roughly few TBs
    - Much larger capacity and cost/byte is less than main memory/flash memory
    - Growing constantly and rapidly with technology improvements (factor of 2 to 3 every 2 years)
  - Survives power failures and system crashes
    - disk failure can destroy data, but is rare

Magnetic Disk

Optical Storage

- non-volatile, data is read optically from a spinning disk using a laser
- CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
- Blu-ray disks: 27 GB to 54 GB
- Write-one, read-many (WORM) optical disks used for archival storage (CD-R, DVD-R, DVD+R)
- Multiple write versions also available (CD-RW, DVD-RW, DVD+RW, and DVD-RAM)
- Reads and writes are slower than with magnetic disk
- **Juke-box** systems, with large numbers of removable disks, a few drives, and a mechanism for automatic loading/unloading of disks available for storing large volumes of data

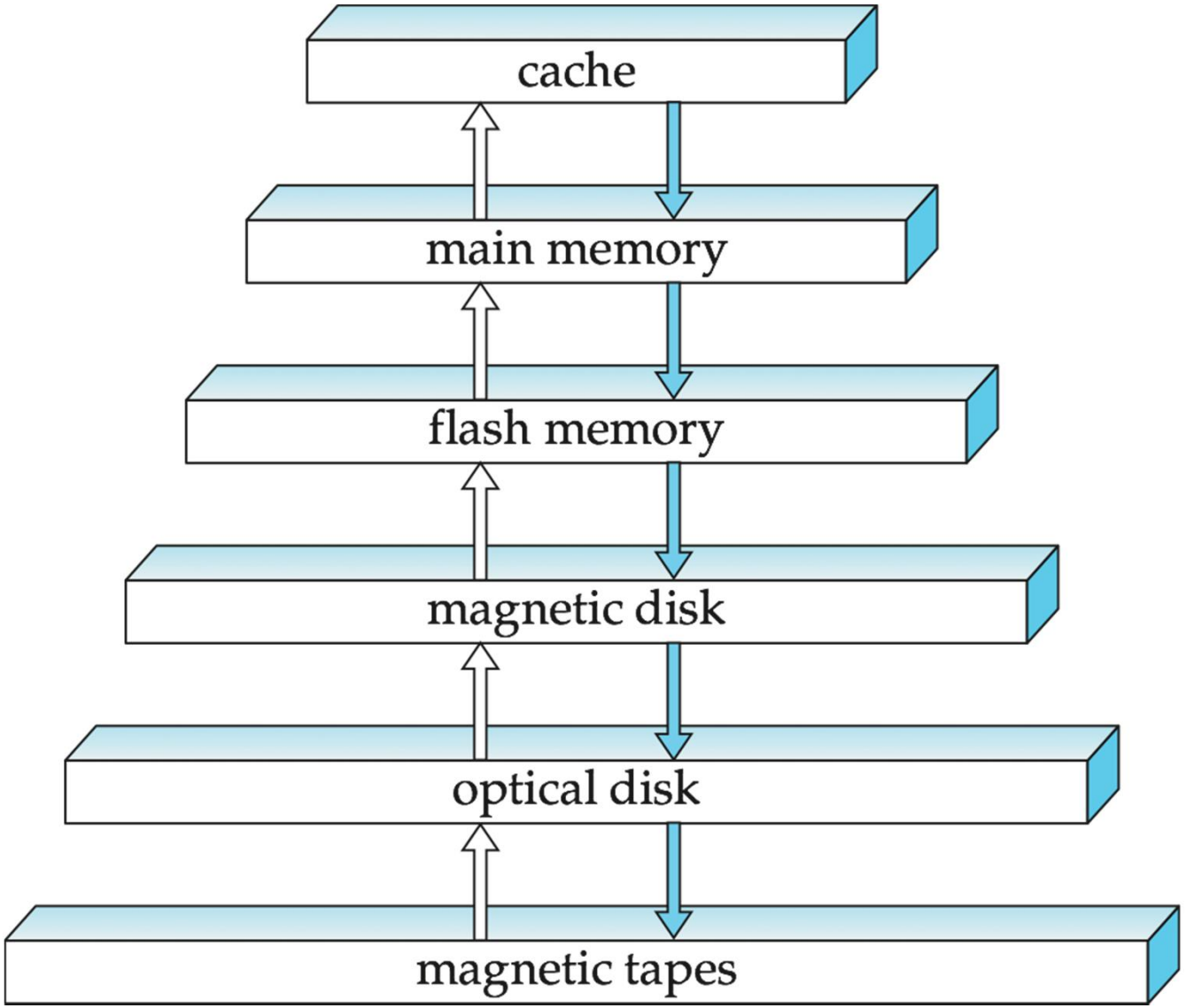
Optical Storage

TAPE STORAGE

- non-volatile, used primarily for backup (to recover from disk failure), and for archival data
- **sequential-access** – much slower than disk
- very high capacity (40 to 300 GB tapes available)
- tape can be removed from drive ⇒ storage costs much cheaper than disk, but drives are expensive
- Tape jukeboxes available for storing massive amounts of data
- hundreds of terabytes (1 terabyte = 1000 gigabytes) to even multiple **petabytes** (1 petabyte = 1000 terabytes)

Tape Storage

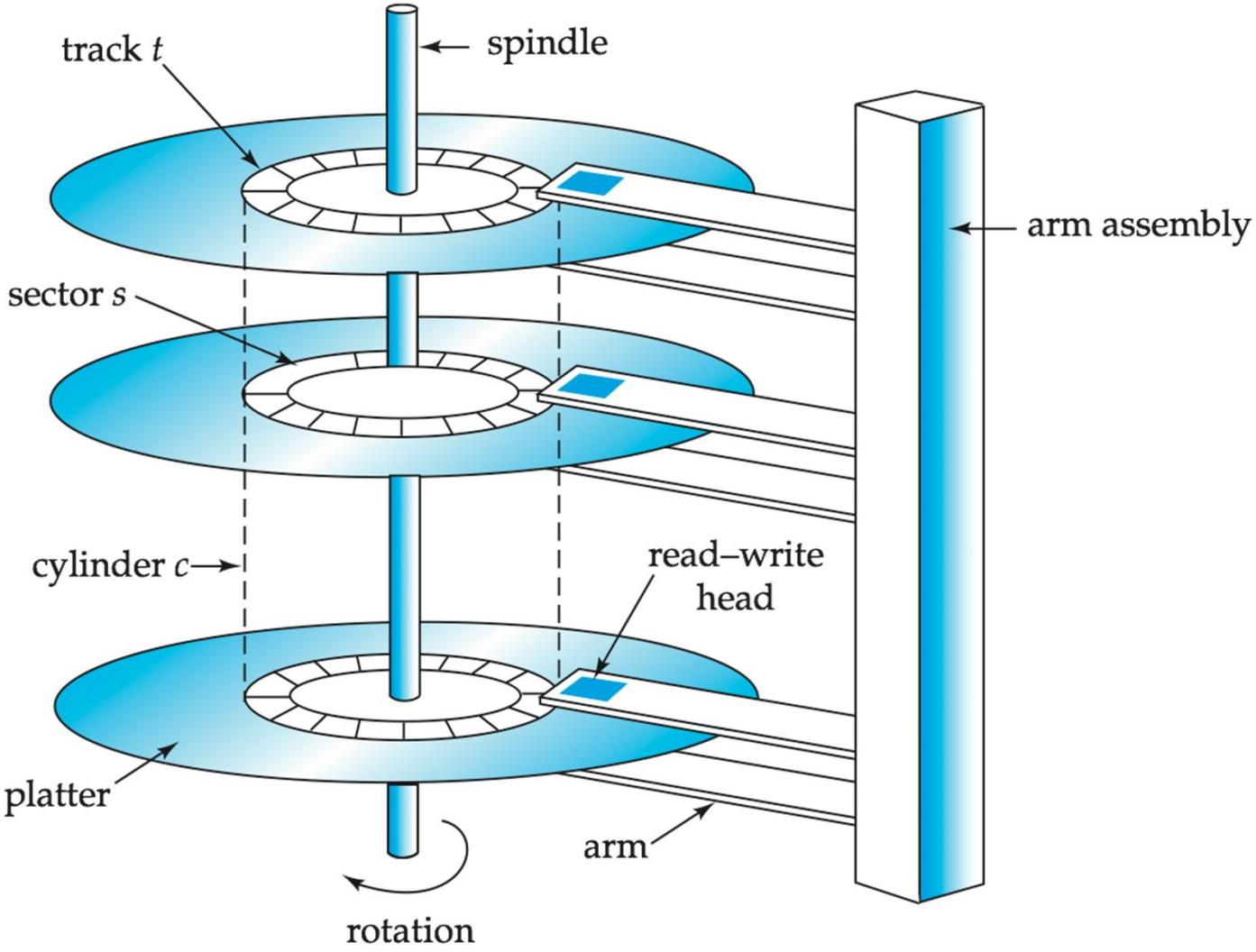
Storage Hierarchy



Storage Hierarchy (Cont.)

- **primary storage:** Fastest media but volatile (cache, main memory).
- **secondary storage:** next level in hierarchy, non-volatile, moderately fast access time
  - also called **on-line storage**
  - E.g. flash memory, magnetic disks
- **tertiary storage:** lowest level in hierarchy, non-volatile, slow access time
  - also called **off-line storage**
  - E.g. magnetic tape, optical storage

Magnetic Hard Disk Mechanism



NOTE: Diagram is schematic, and simplifies the structure of actual disk drives

Magnetic Disks

- **Read-write head**
  - Positioned very close to the platter surface (almost touching it)
  - Reads or writes magnetically encoded information.
- Surface of platter divided into circular **tracks**
  - Over 50K-100K tracks per platter on typical hard disks
- Each track is divided into **sectors.**
  - A sector is the smallest unit of data that can be read or written.
  - Sector size typically 512 bytes
  - Typical sectors per track: 500 to 1000 (on inner tracks) to 1000 to 2000 (on outer tracks)
- To read/write a sector
  - disk arm swings to position head on right track
  - platter spins continually; data is read/written as sector passes under head
- Head-disk assemblies
  - multiple disk platters on a single spindle (1 to 5 usually)
  - one head per platter, mounted on a common arm.
- **Cylinder** *i* consists of i^{th} track of all the platters

Magnetic Disks (Cont.)

- Earlier generation disks were susceptible to head-crashes
  - Surface of earlier generation disks had metal-oxide coatings which would disintegrate on head crash and damage all data on disk
  - Current generation disks are less susceptible to such disastrous failures, although individual sectors may get corrupted

DISK CONTROLLER

- interfaces between the computer system and the disk drive hardware.
- accepts high-level commands to read or write a sector
- initiates actions such as moving the disk arm to the right track and actually reading or writing the data
- Computes and attaches **checksums** to each sector to verify that data is read back correctly
- Multiple disks connected to a computer system through a controller

DISK BLOCK ACCESS

- Requests for disk I/O are generated both by the file system and by the virtual memory manager
- Each request specifies the address on the disk to be referenced
  - This address is in the form of *block number*
- A **block** is a logical unit consisting of a fixed number of contiguous sectors
  - It may range from 512 bytes to several kilobytes
- Data are transferred between disk and main memory in units of blocks

Performance Measures of Disks

- The main measures of the qualities of a disk are
  - Capacity
  - Access time
  - Data transfer rate
  - Reliability

Access Time

- It is the time when a read/write request is issued to when data transfer begins
- It is also sum of *seek time* and *rotational latency time*
- **Seek time:** the time for repositioning the arm under correct track
  - Typically ranges from 2 to 30 miliseconds
- **Rotational latency time:** the time spent waiting for the sector to be appeared under read/write head
  - Typically ranges from 4 to 11.1 miliseconds per rotation
  - On an average half rotation is required for the beginning of the desired sector to appear under the head

Data Transfer Rate

- The rate at which the data can be retrieved from or stored to the disk
- Data transfer begins when the first sector of the data to be accessed has come under head
- Current disk systems claim to support maximum transfer rate of 25 to 40 megabytes per second

MTTF: Mean Time To Failure

- A measure of the reliability of the disk
- It is the amount of time that, on average, the system runs continuously without any failure
- According to vendors, the MTTF of disks ranges from 30,000 hrs to 1,200,000 hrs i.e., about 3.4 to 136 years

Redundant Array of Independent Disks (RAID)

- Disk organization techniques that manage a large numbers of disks, providing a view of a single disk of
  - high capacity and high speed by using multiple disks in parallel,
  - high reliability by storing data redundantly, so that data can be recovered even if a disk fails

- Originally a cost-effective alternative to large, expensive disks
  - 'I' in RAID originally stood for "inexpensive"
  - Today RAIDs are used for their higher reliability and performance, rather than for economic reasons.
    - The "I" is interpreted as independent

Improvement of Reliability via Redundancy

- **Redundancy** – store extra information that can be used to rebuild information lost in a disk failure
- E.g., **Mirroring** (or **shadowing**)
  - Duplicate every disk. Logical disk consists of two physical disks.
  - Every write is carried out on both disks
  - Reads can take place from either disk
  - If one disk in a pair fails, data still available in the other
  - Data loss would occur only if a disk fails, and its mirror disk also fails before the system is repaired
  - Probability of combined event is very small
  - Except for dependent failure modes such as fire or building collapse or electrical power surges

- **Mean time to data loss** depends on **mean time to failure**, and **mean time to repair**
- Power failure, natural disasters such as earthquake, fire, floods may result in the damage of both disks at the same time

Improvement in Performance via Parallelism

- Two main goals of parallelism in a disk system:
  1. Load balance multiple small accesses to increase throughput
  2. Parallelize large accesses to reduce response time.
- Improve transfer rate by striping data across multiple disks.
  - Bit level striping
  - Block level striping

BIT-LEVEL STRIPING

- **Bit-level striping** – split the bits of each byte across multiple disks
  - In an array of eight disks, write bit *i* of each byte to disk *i*.
  - Each access can read data at eight times the rate of a single disk.
  - But seek/access time worse than for a single disk
  - Bit level striping is not used much any more

BLOCK-LEVEL STRIPING

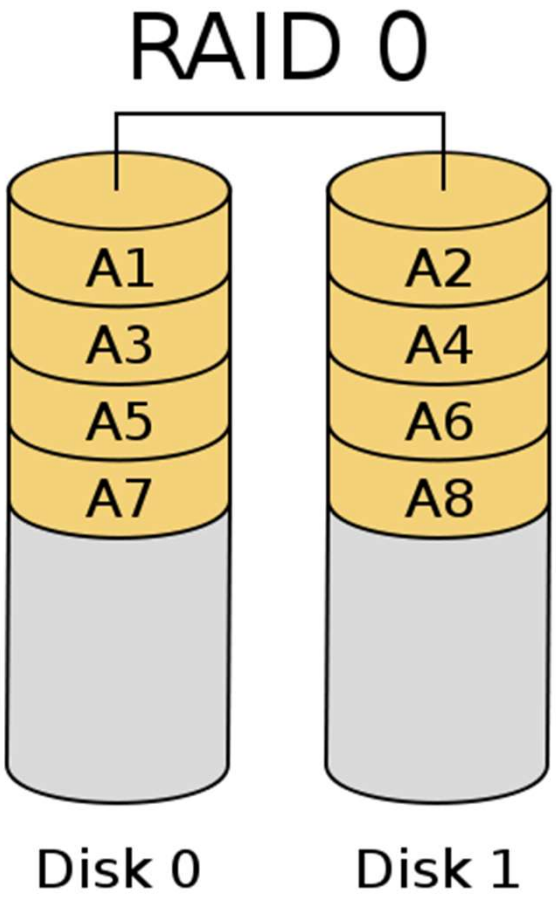
- **Block-level striping** – with n disks, block i of a file goes to disk (i mod n) + 1
  - Requests for different blocks can run in parallel if the blocks reside on different disks
  - A request for a long sequence of blocks can utilize all disks in parallel

RAID Levels

- Mirroring provides high reliability but it is expensive
- Striping provides high data transfer rates, but does not improve reliability
- Schemes to provide redundancy at lower cost by using disk striping combined with parity bits
- Different RAID organizations, or RAID levels, have differing cost, performance and reliability characteristics

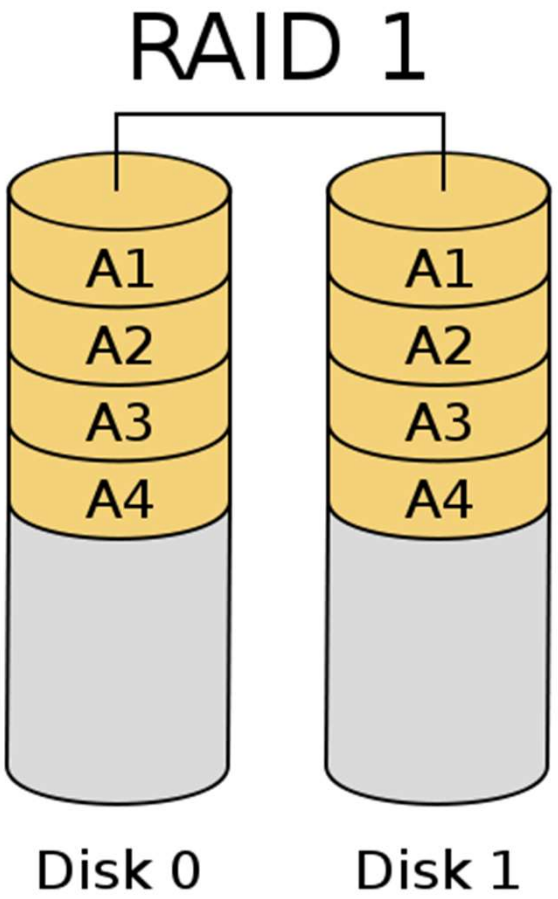
RAID 0

- **RAID Level 0**: Block striping; non-redundant.
  - Data are split up in blocks that get written across all the drives in the array
  - It is ideal for non-critical storage of data that have to be read/written at a high speed



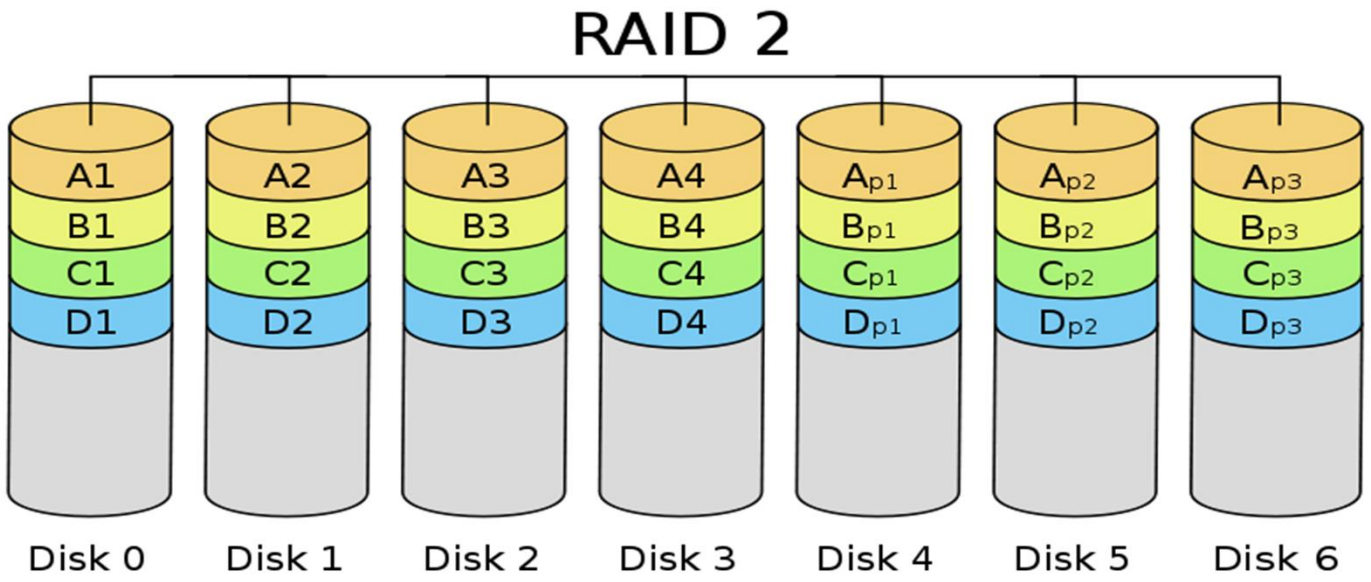
RAID 1

- **RAID Level 1**: Mirrored disks
- Data are stored twice by writing them to the both the data disk(s) and a mirror disk(s)
- **RAID Level 1+0 or 10:** if RAID 1 is combined with RAID 0 to improve performance
- RAID-1 is ideal for mission critical storage, for instance for accounting systems. It is also suitable for small servers in which only two disks will be used.
- Popular for applications such as storing log files in a database system.



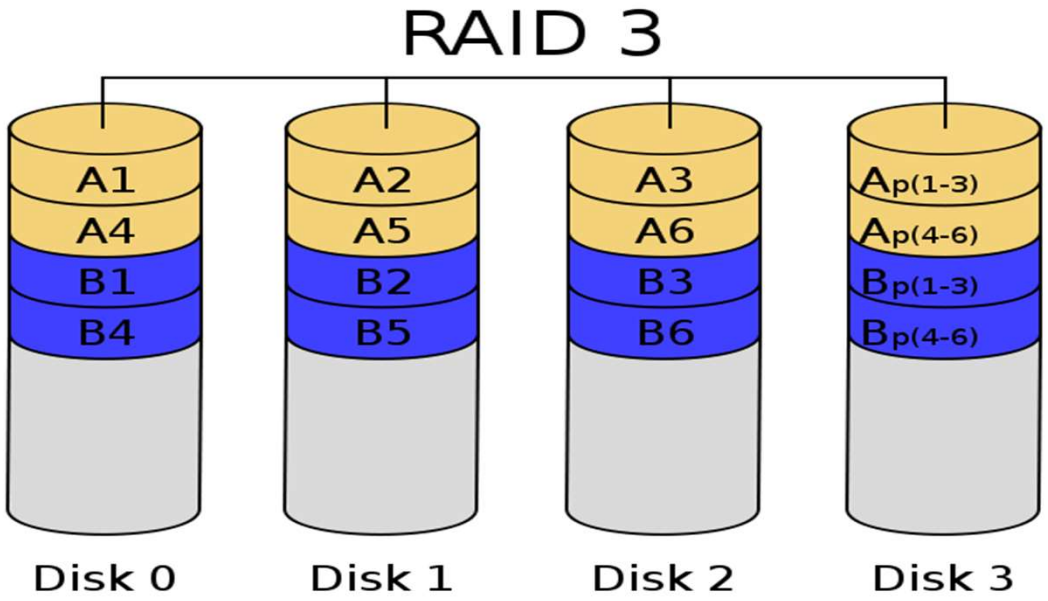
RAID 2

- **RAID Level 2**: Memory-Style Error-Correcting-Codes (ECC) with bit striping.



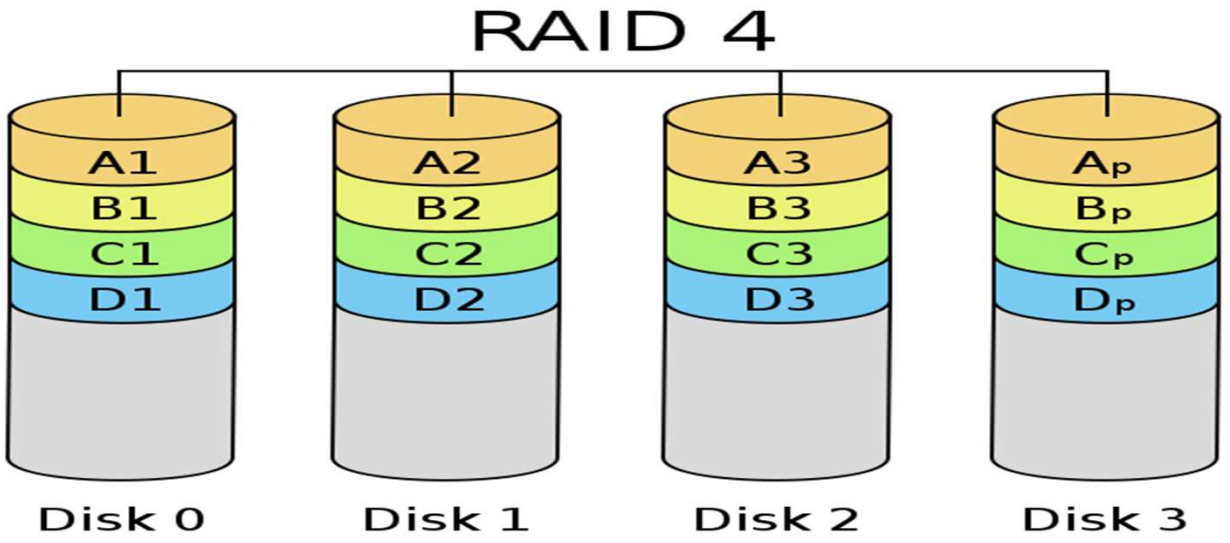
RAID 3

- **RAID Level 3**: Bit-Interleaved Parity
  - a single parity bit is enough for error correction, not just detection, since we know which disk has failed
    - When writing data, corresponding parity bits must also be computed and written to a parity bit disk
    - To recover data in a damaged disk, compute XOR of bits from other disks (including parity bit disk)



RAID 4

- **RAID Level 4:** Block-Interleaved Parity; uses block-level striping, and keeps a parity block on a separate disk for corresponding blocks from *N* other disks.
  - When writing data block, corresponding block of parity bits must also be computed and written to parity disk
  - To find value of a damaged block, compute XOR of bits from corresponding blocks (including parity block) from other disks.

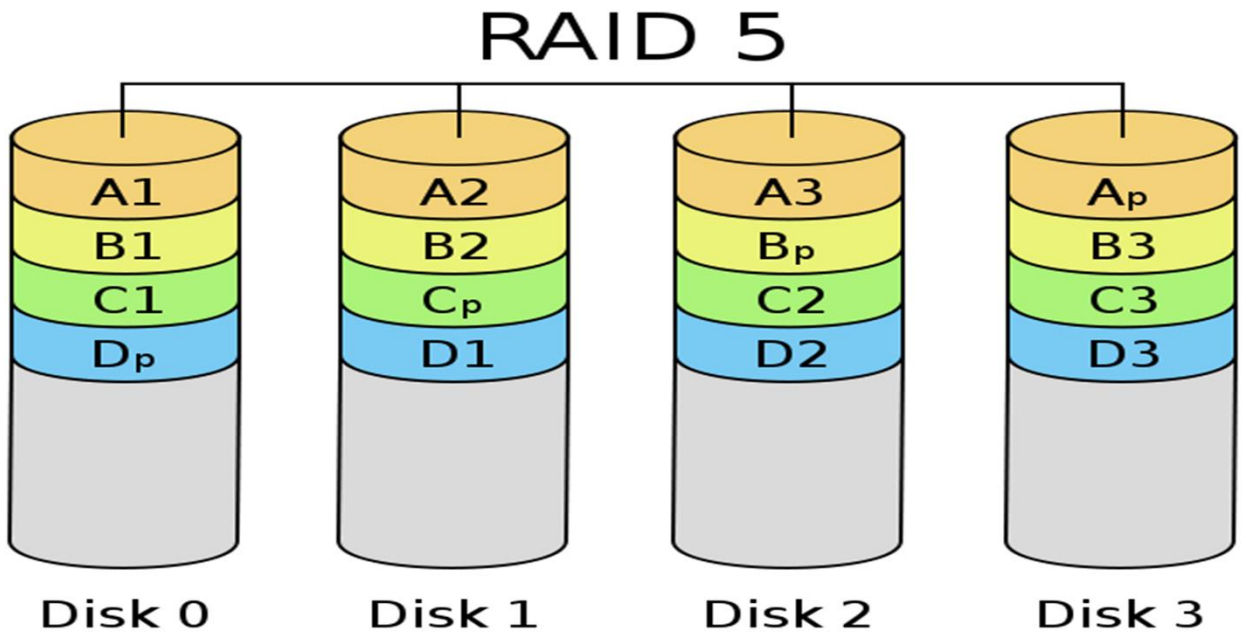


RAID Levels (Cont.)

- **RAID Level 4** (Cont.)
  - Provides higher I/O rates for independent block reads than Level 3
    - block read goes to a single disk, so blocks stored on different disks can be read in parallel
  - Provides high transfer rates for reads of multiple blocks than no-striping
  - Before writing a block, parity data must be computed
    - Can be done by using old parity block, old value of current block and new value of current block (2 block reads + 2 block writes)
    - Or by recomputing the parity value using the new values of blocks corresponding to the parity block
      - More efficient for writing large amounts of data sequentially
  - Parity block becomes a bottleneck for independent block writes since every block write also writes to parity disk

RAID 5

- **RAID Level 5:** Block-Interleaved Distributed Parity; partitions data and parity among all $N + 1$ disks, rather than storing data in N disks and parity in 1 disk.

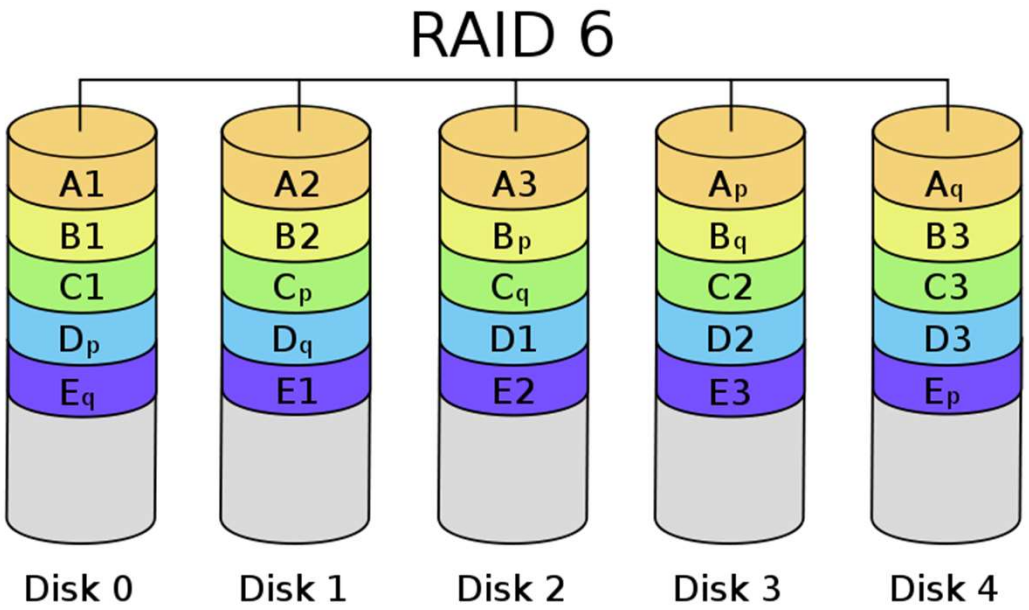


RAID 5 (CONTD.)

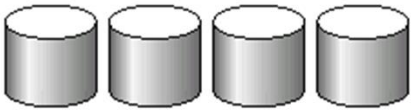
- Higher I/O rates than Level 4.
  - Block writes occur in parallel if the blocks and their parity blocks are on different disks.
- Subsumes Level 4: provides same benefits, but avoids bottleneck of parity disk.

RAID 6

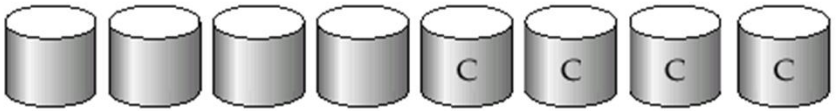
- **RAID Level 6**: P+Q Redundancy scheme; similar to Level 5, but stores extra redundant information to guard against multiple disk failures.
  - Better reliability than Level 5 at a higher cost; not used as widely.



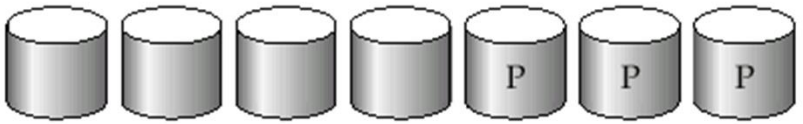
RAID Levels at a glance



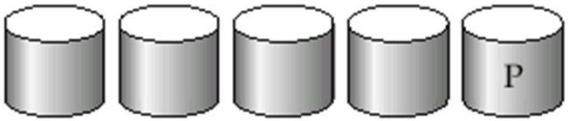
(a) RAID 0: nonredundant striping



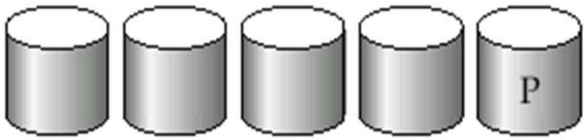
(b) RAID 1: mirrored disks



(c) RAID 2: memory-style error-correcting codes

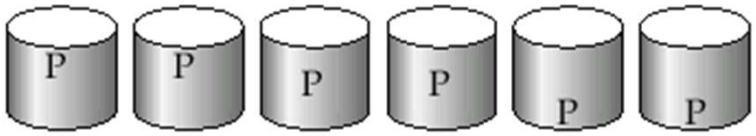


(d) RAID 3: bit-interleaved parity



(e) RAID 4: block-interleaved parity

(f) RAID 5: block-interleaved distributed parity



(g) RAID 6: P + Q redundancy

Choice of RAID Level

- Factors in choosing RAID level
  - Monetary cost
  - Performance: Number of I/O operations per second
  - Performance during failure
  - Performance during rebuild of failed disk
    - Including time taken to rebuild failed disk

Choice of RAID Level (contd.)

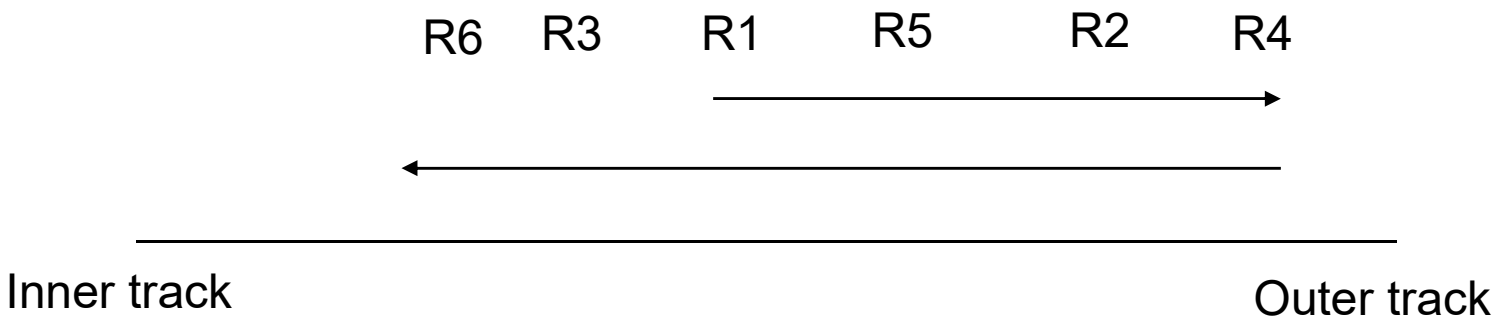
- RAID 0 is used only when data safety is not important
  - E.g. data can be recovered quickly from other sources
- Level 2 and 4 never used since they are subsumed by 3 and 5
- Level 3 is not used anymore since bit-striping forces single block reads to access all disks, wasting disk arm movement, which block striping (level 5) avoids
- Level 6 is rarely used since levels 1 and 5 offer adequate safety for most applications

Choice of RAID Level (Cont.)

- Level 1 provides much better write performance than level 5
  - Level 5 requires at least 2 block reads and 2 block writes to write a single block, whereas Level 1 only requires 2 block writes
  - Level 1 preferred for high update environments such as log disks
- Level 1 had higher storage cost than level 5
  - disk drive capacities increasing rapidly (50%/year) whereas disk access times have decreased much less (x 3 in 10 years)
  - I/O requirements have increased greatly, e.g. for Web servers
  - When enough disks have been bought to satisfy required rate of I/O, they often have spare storage capacity
    - so there is often no extra monetary cost for Level 1!
- Level 5 is preferred for applications with low update rate, and large amounts of data
- Level 1 is preferred for all other applications

Optimization of Disk-Block Access

- **Buffering:** in-memory buffer to cache disk blocks
- **Read-ahead:** Read extra blocks from a track in anticipation that they will be requested soon
- **Disk-arm-scheduling** algorithms re-order block requests so that disk arm movement is minimized
  - **elevator algorithm**



File Organization

- The database is stored as a collection of *files*. Each file is a sequence of *records*. A record is a sequence of *fields*.
- The records are mapped to disk blocks
- A block may contain several records
- Blocks are unit of storage allocation and data transfer
- Generally, most of the records are smaller than a block
  - Exceptions are image, video, etc.
- Each record is entirely contained in a single block
  - No record is partly in one block and partly in another block

- In relational database, tuples of distinct relations are generally of different sizes
- One approach:
  - assume record size is fixed
  - each file has records of one particular type only
  - different files are used for different relations

  This case is easiest to implement; will consider variable length records later.

FIXED LENGTH RECORDS

- Let's consider a file of instructor records for University database system
  - ***type*** *instructor* = ***record***

    ID varchar(5);

    name varchar(20);

    dept_name varchar(20);

    salary numeric(8,2);

    end*;*
- Each character occupies 1 byte and numeric occupies 8 byte
- Thus each instructor record is 53 bytes long

Fixed-Length Records

- Simple approach:
  - Use first n bytes for first record and next n bytes for next record where n is the size of each record.
  - Record access is simple but records may cross blocks
    - Modification: do not allow records to cross block boundaries

- Deletion of record i: alternatives-
  - move records $i + 1, \ldots, n$ to $i, \ldots, n - 1$
  - move record n to i
  - do not move records, but link all free records on a *free list*

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Deleting record 3 and compacting

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

All these records shifted one place up

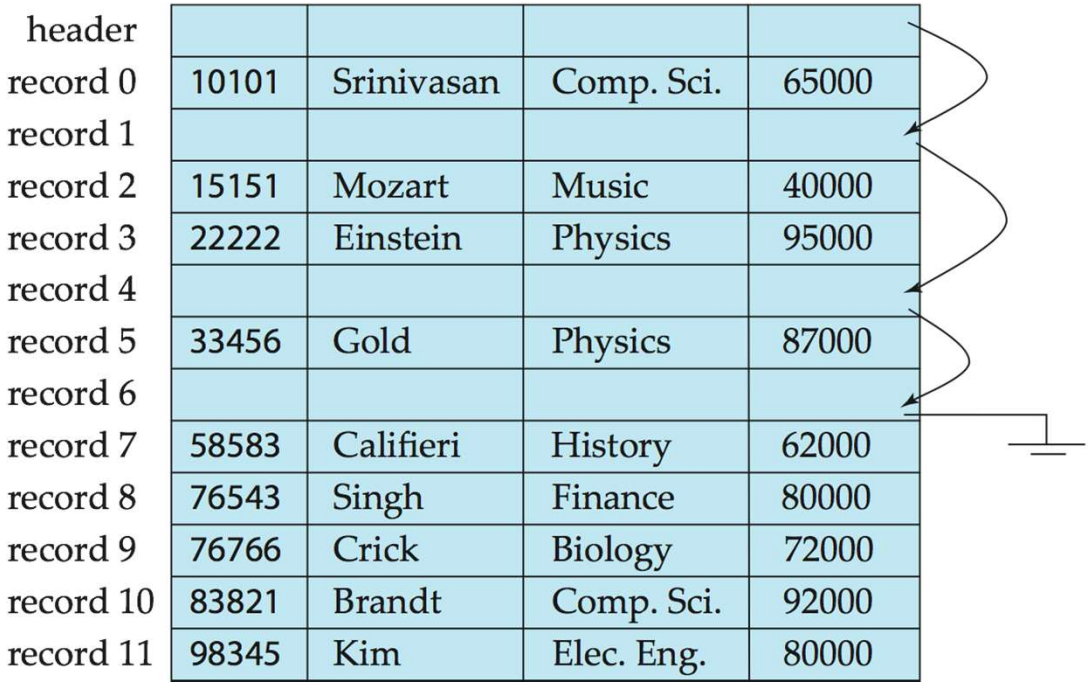
Deleting record 3 and moving last record

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000

Record 11 is only moved up to fill the empty slot

FREE LISTS

- Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on
- Can think of these stored addresses as pointers since they "point" to the location of a record.
- More space efficient representation: reuse space for normal attributes of free records to store pointers. (No pointers stored in in-use records.)



header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

VARIABLE-LENGTH RECORDS

- Variable-length records arise in database systems in several ways:
  - Storage of multiple record types in a file.
  - Record types that allow variable lengths for one or more fields such as strings (**varchar**)
  - Record types that allow repeating fields (used in some older data models).
- Variable length representation
- Fixed length representation

BYTE STRING REPRESENTATION

- A simple variable length representation method
- Attaches a special *end of record* (⊥) symbol to the end of each record
- Each record is stored as a string of consecutive bytes
- Let's consider the following account information
  - **type** account = **record**

```
Branch_name:char(22)
Account_info: array[1..infinity] of
                record;
                        account_no: char(10);
                        balance: real;
                end
end
```

Byte String Representation

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

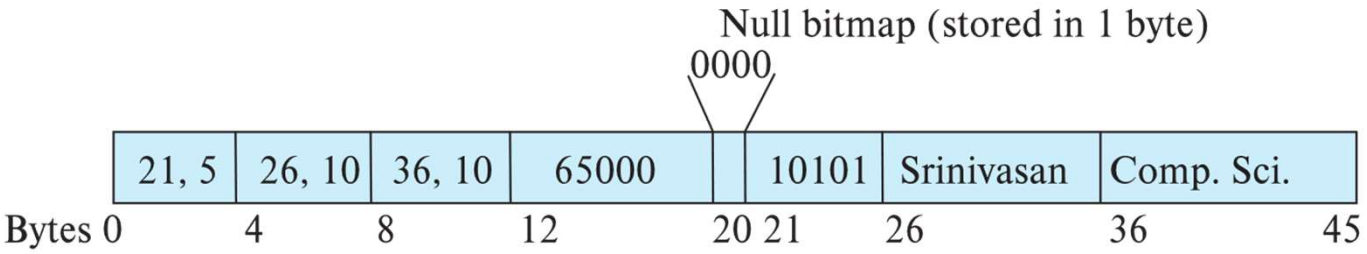
Problems

Not easy to reuse space occupied by the deleted record

No space for records to grow longer

VARIABLE-LENGTH RECORDS (OFFSET AND LENGTH)

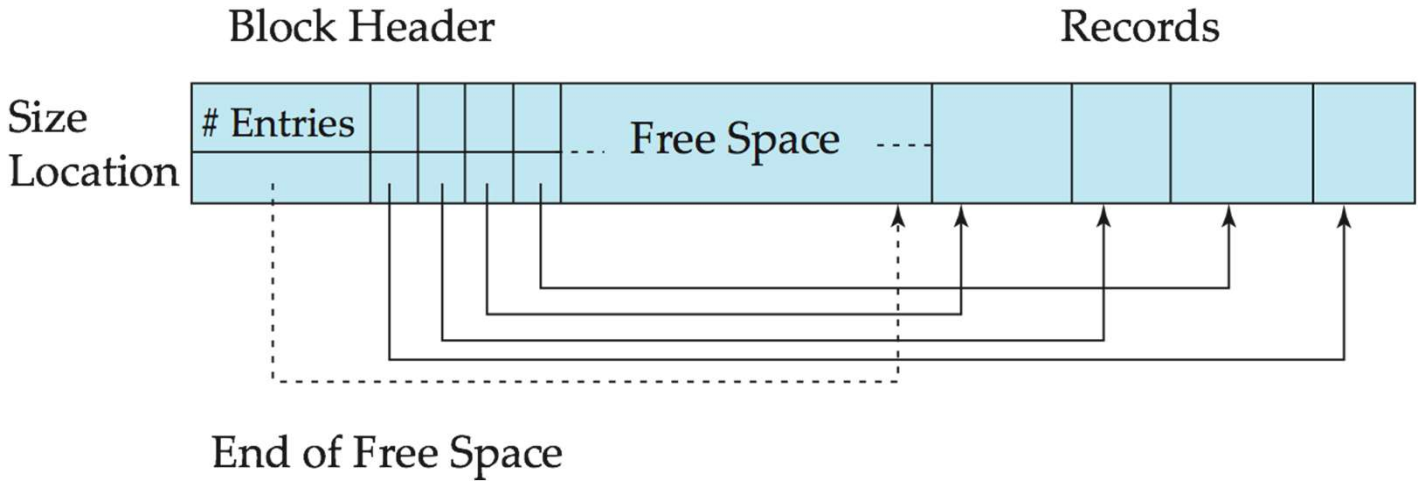
- Attributes are stored in order
- Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes
- Null values represented by null-value bitmap



Slotted Page Structure

- *Slotted page structure* is Commonly used for organizing variable length records within a block

VARIABLE-LENGTH RECORDS: SLOTTED PAGE STRUCTURE



- **Slotted page** header contains:
  - number of record entries
  - end of free space in the block
  - location and size of each record
- Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.
- Pointers should not point directly to record — instead they should point to the entry for the record in header.

Fixed Length Representation

Using reserved space method

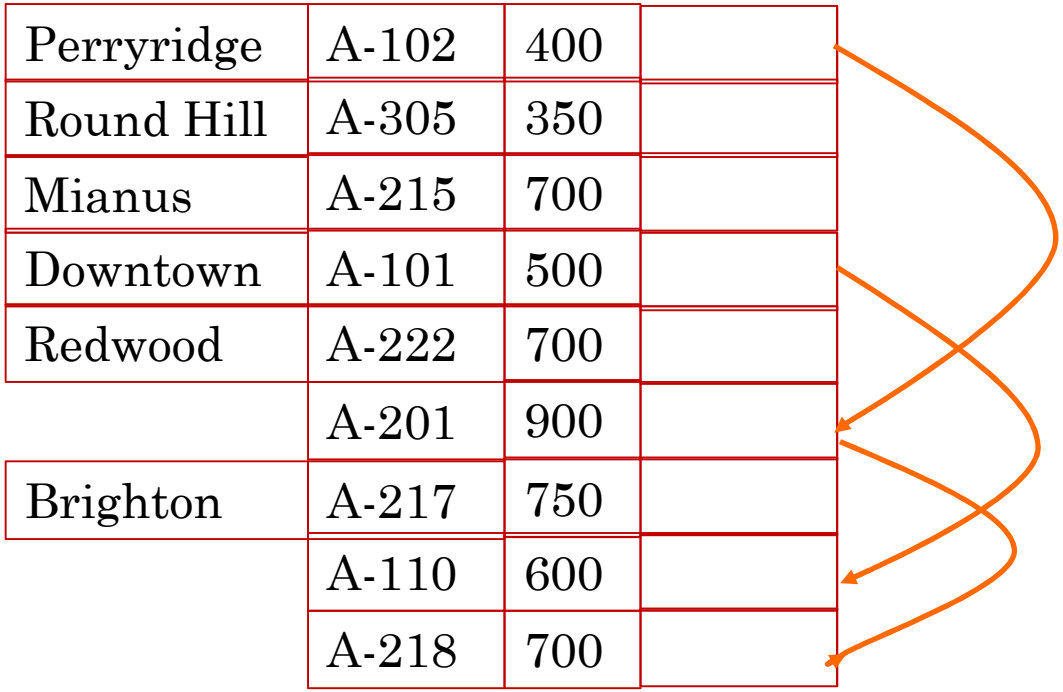
Perryridge	A-102	400	A-201	900	A-218	700
Round Hill	A-305	350	⊥	⊥	⊥	⊥
Mianus	A-215	700	⊥	⊥	⊥	⊥
Downtown	A-101	500	A-110	600	⊥	⊥
Redwood	A-222	700	⊥	⊥	⊥	⊥
Brighton	A-217	750	⊥	⊥	⊥	⊥

If there is a maximum record length that is never exceeded, then this scheme can be used

Unused space shorter than the maximum length is filled with a special null or end of character

Another Fixed Length Representation

Using list representation



Perryridge	A-102	400	
Round Hill	A-305	350	
Mianus	A-215	700	
Downtown	A-101	500	
Redwood	A-222	700	
	A-201	900	
Brighton	A-217	750	
	A-110	600	
	A-218	700	

The records are chained together by pointers

Wastage of space except the first record in the chain

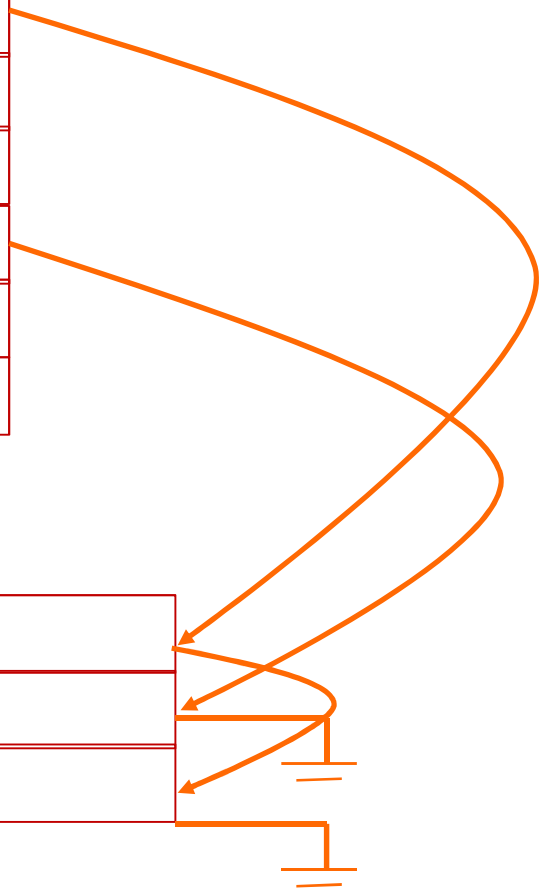
Alternate list Representation

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-110	600	
A-218	700	



- We have seen how records are represented in a file structure
- A relation is a set of records
- How to organize them in a file?

Organization of Records in Files

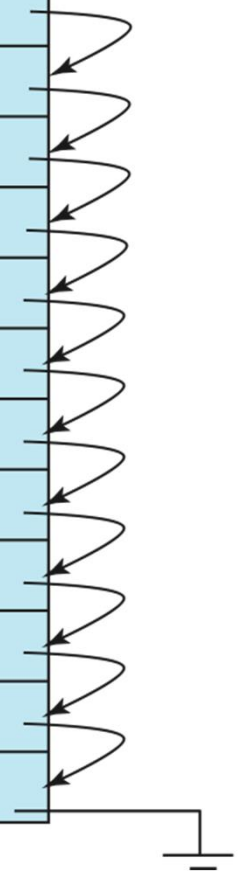
- **Heap** – a record can be placed anywhere in the file where there is space
- **Sequential** – store records in sequential order, based on the value of the search key of each record
- **Hashing** – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed
- Generally, records of each relation are stored in a separate file. However, in a **multitable clustering file organization** records of several different relations can be stored in the same file
  - Motivation: store related records on the same block to minimize I/O

Sequential File Organization

- Designed for efficient processing of records in sorted order based on some search key
- **Search key** is
  - any attribute or set of attributes
  - It need not be the primary key or superkey
  - Used for fast retrieval of records in search key order
- The records are linked together by pointers
- The pointer in each record points to the next record in search key order
- To minimize the no. of block accesses the records are stored physically in search key order or as close to search key order as possible

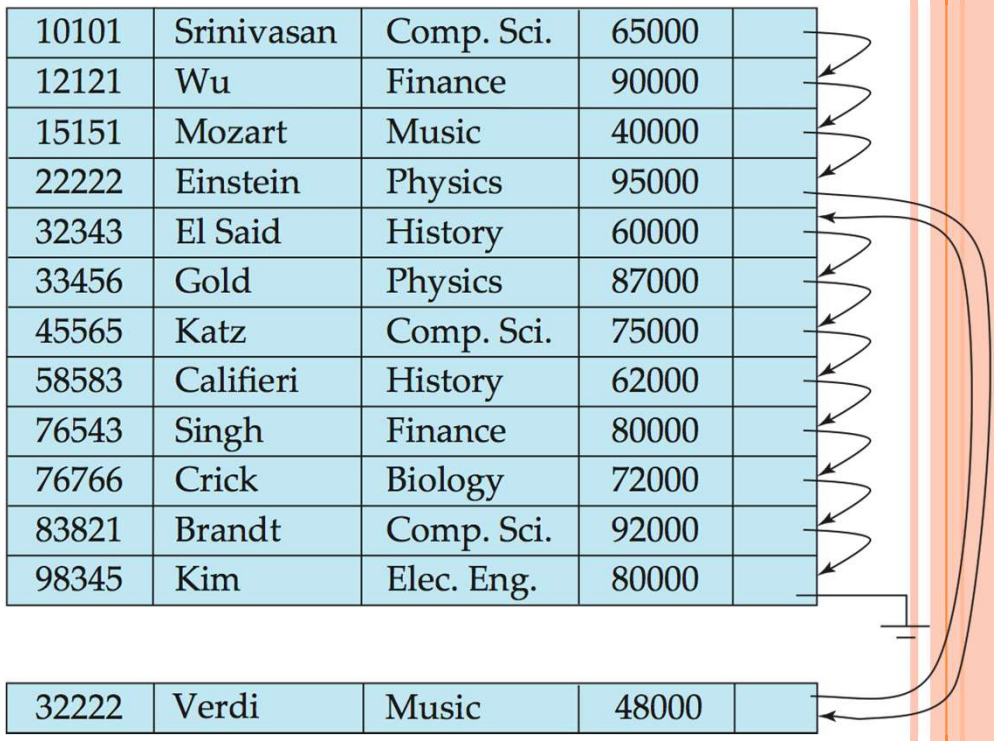
EXAMPLE

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	



Sequential File Organization (Cont.)

- **Deletion** – use pointer chains
- **Insertion** –locate the position where the record is to be inserted
  - if there is free space insert there
  - if no free space, insert the record in an overflow block
  - In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order



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	

32222	Verdi	Music	48000	

MULTITABLE CLUSTERING FILE ORGANIZATION

Store several relations in one file using a **multitable clustering** file organization

department

dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Physics	Watson	70000

instructor

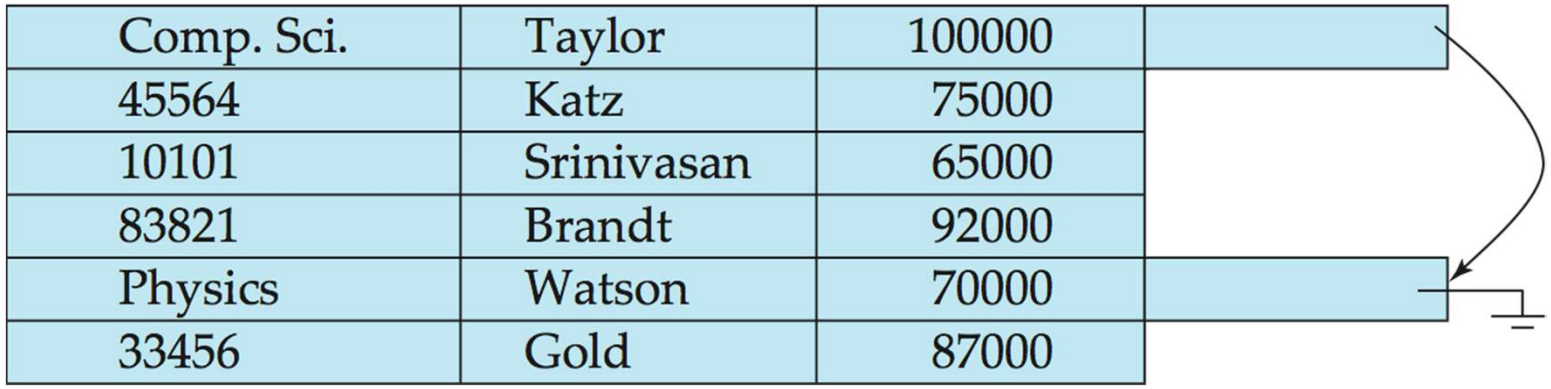
ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

multitable clustering of *department* and *instructor*

Comp. Sci.	Taylor	100000
45564	Katz	75000
10101	Srinivasan	65000
83821	Brandt	92000
Physics	Watson	70000
33456	Gold	87000

Multitable Clustering File Organization (cont.)

- good for queries involving *department* $\bowtie$ *instructor*, and for queries involving one single department and its instructors
- bad for queries involving only *department*
- results in variable size records
- Can add pointer chains to link records of a particular relation



Comp. Sci.	Taylor	100000	
45564	Katz	75000	
10101	Srinivasan	65000	
83821	Brandt	92000	
Physics	Watson	70000	
33456	Gold	87000	

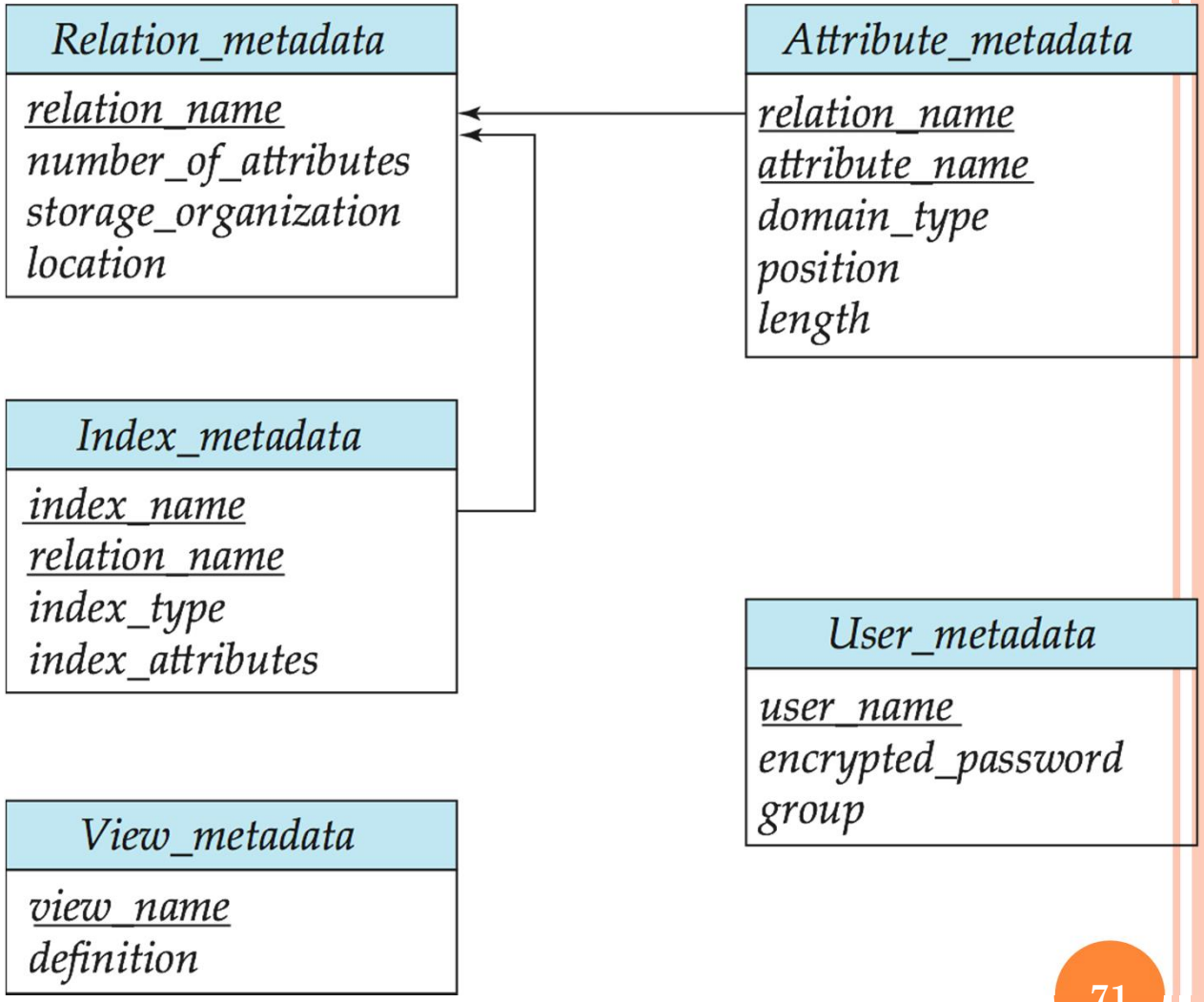
Data Dictionary Storage

The **Data dictionary** (also called **system catalog**) stores **metadata**; that is, data about data, such as

- Information about relations
  - names of relations
  - names, types and lengths of attributes of each relation
  - names and definitions of views
  - integrity constraints
- User and accounting information, including passwords
- Statistical and descriptive data
  - number of tuples in each relation
- Physical file organization information
  - How relation is stored (sequential/hash/…)
  - Physical location of relation

Relational Representation of System Metadata

- Relational representation on disk
- Specialized data structures designed for efficient access, in memory



Indexing and Hashing: Basic Concepts

- 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.
- An **index file** consists of records (called **index entries**) of the form

- 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

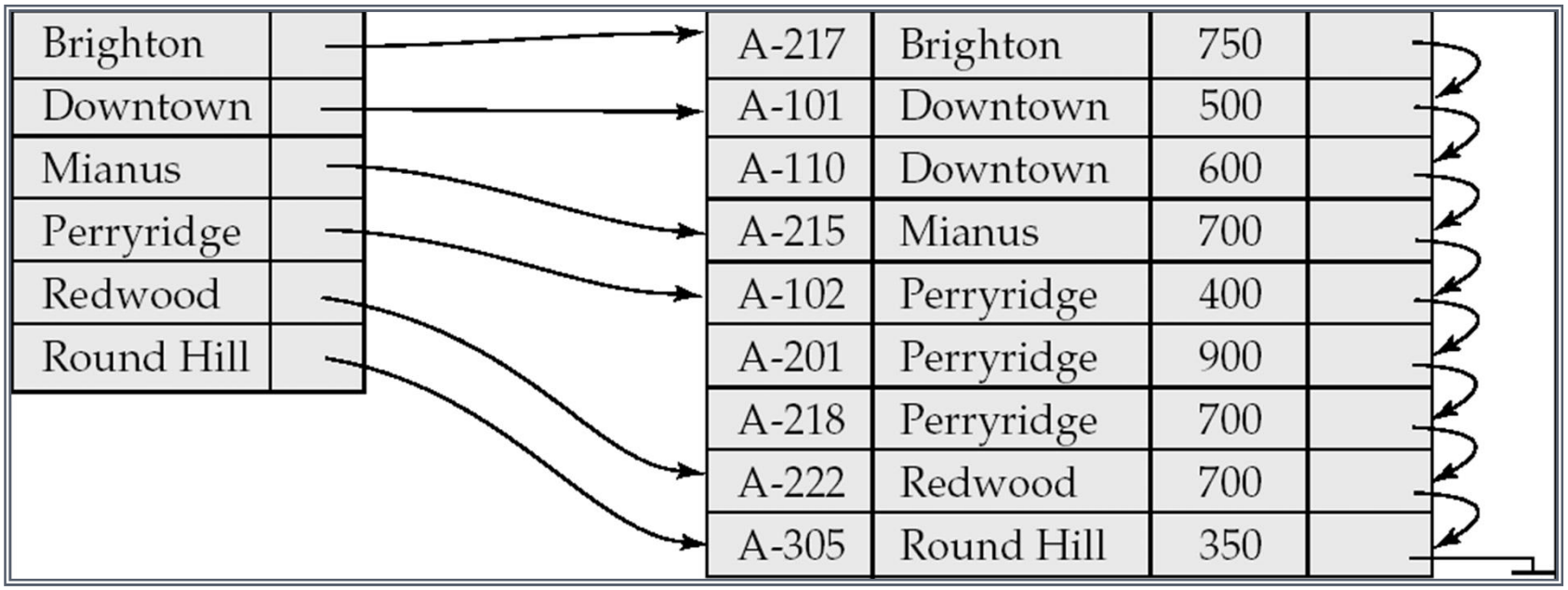
- Access types supported efficiently
  - records with a specified value in the attribute
  - or records with an attribute value falling in a specified range of values (e.g. 10000 < *salary* < 40000)
- Access time
  - The time it takes to find a particular data item, or set of items
- Insertion time includes
  - The time it takes to find the correct place to insert new data item
  - The time it takes to update the index structure
- Deletion time includes
  - The time it takes to find the item to be deleted
  - The time to update the index structure
- Space overhead
  - The additional space occupied by an index structure
  - Usually worthwhile to sacrifice some space to achieve improved performance

Ordered Indices

- In an **ordered index**, index entries are stored sorted on the search key value. E.g., author catalog in library.
- **Primary index**: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called **non-clustering index**
- **Index-sequential file**: ordered sequential file with a primary index.
- Types-
  - Dense
  - Sparse

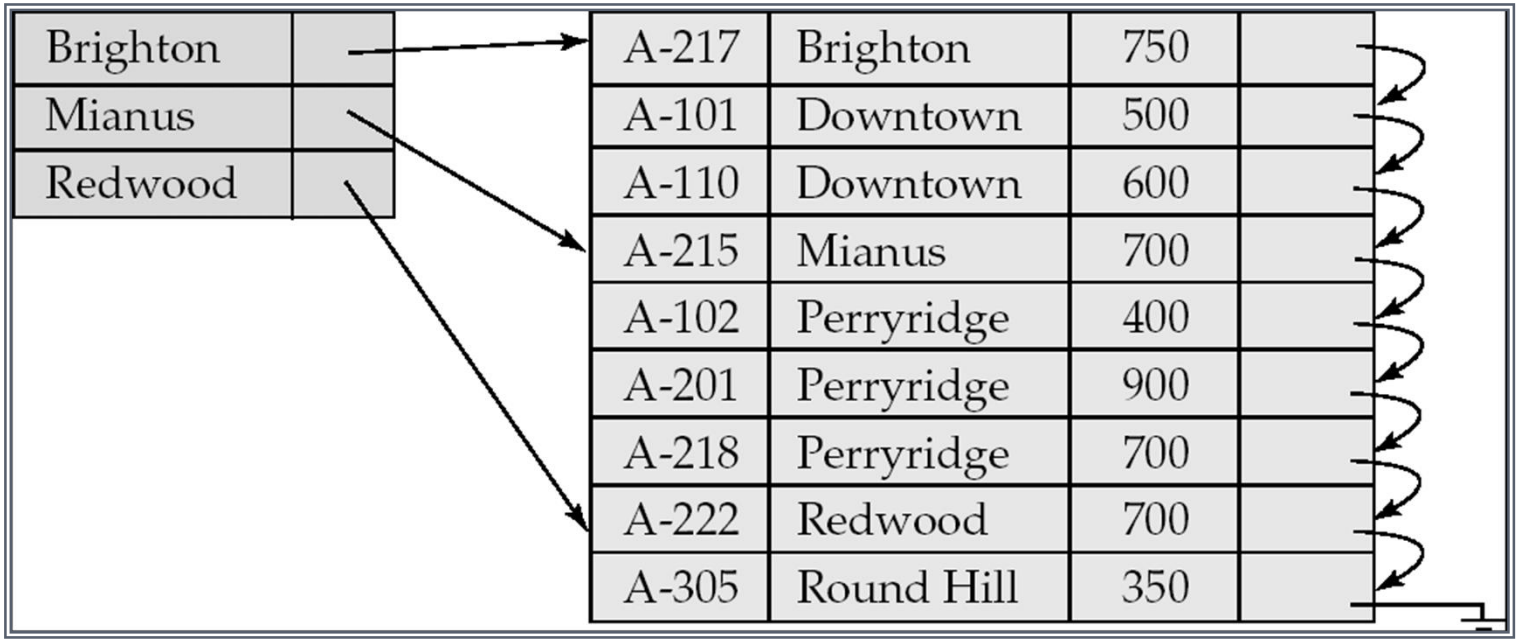
Dense Index Files

- Dense index — Index record appears for every search-key value in the file.



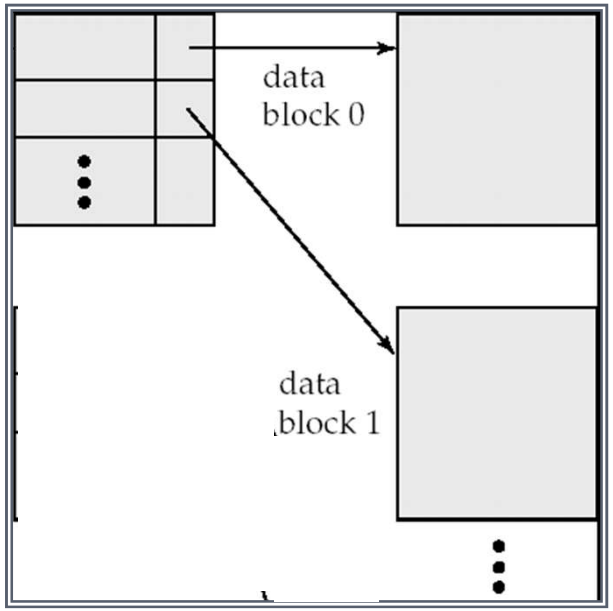
Sparse Index Files

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



Sparse Index Files (Cont.)

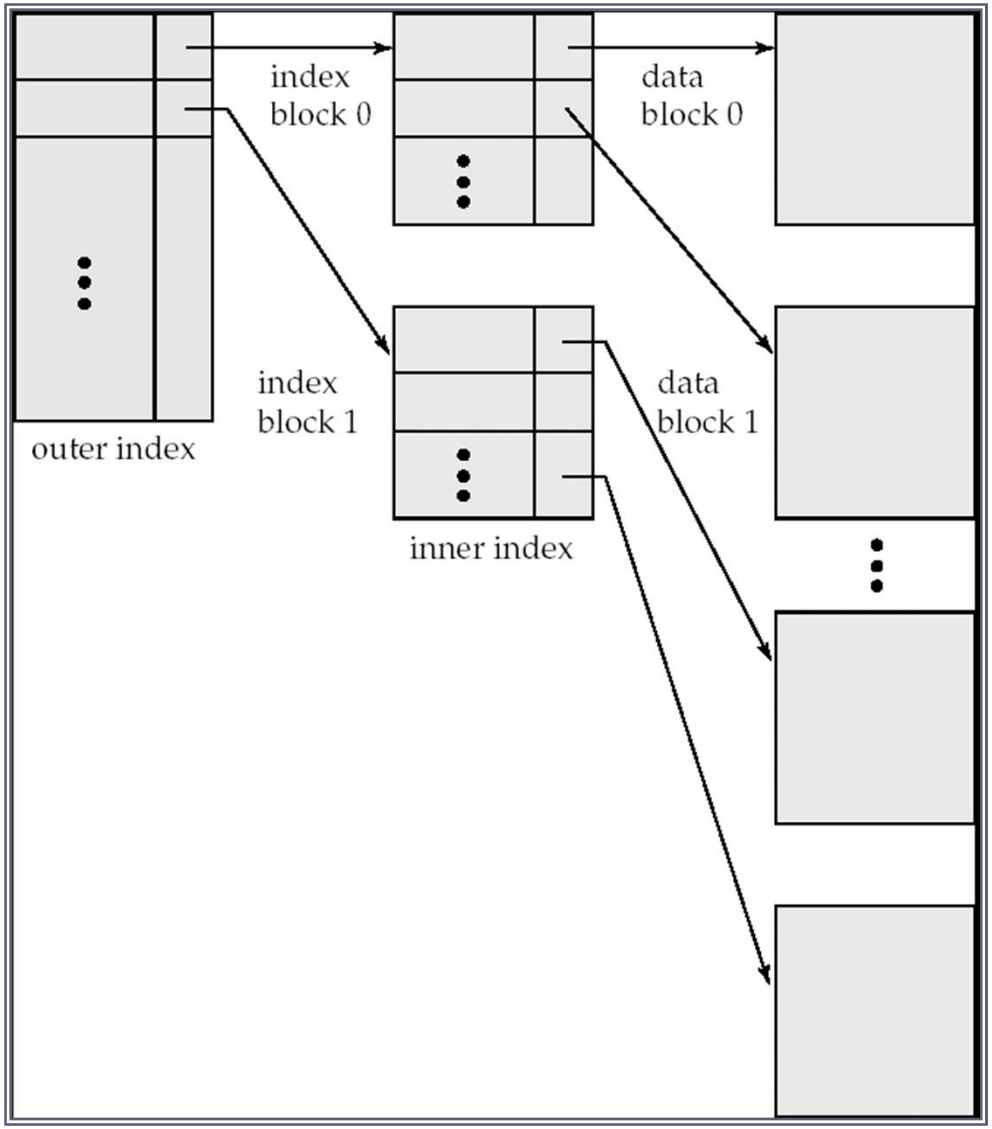
- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than dense index for locating records.
- **Good tradeoff**: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



MULTILEVEL INDEX

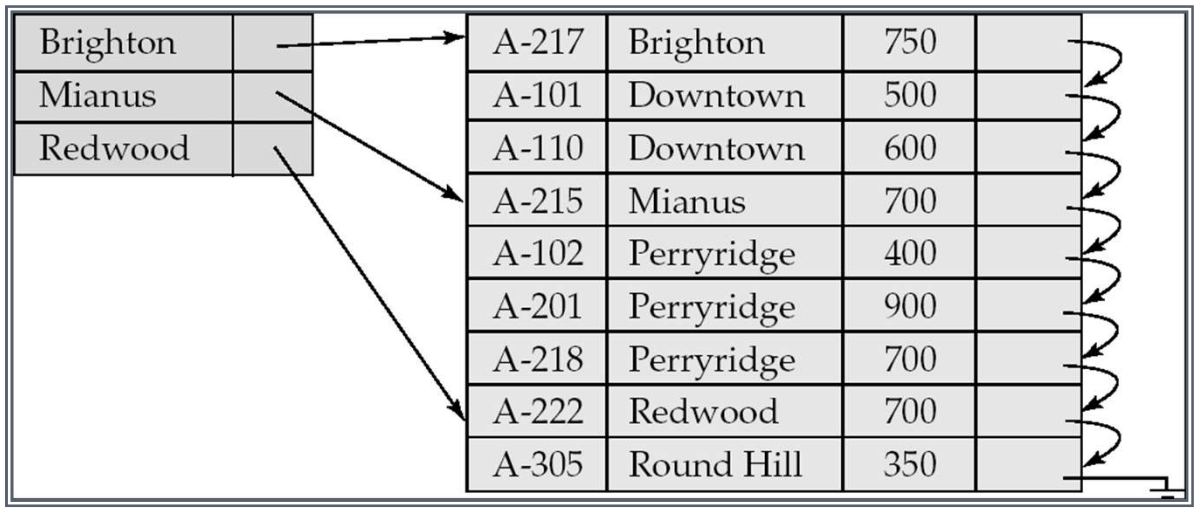
- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit 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.)



Index Update: Record Deletion

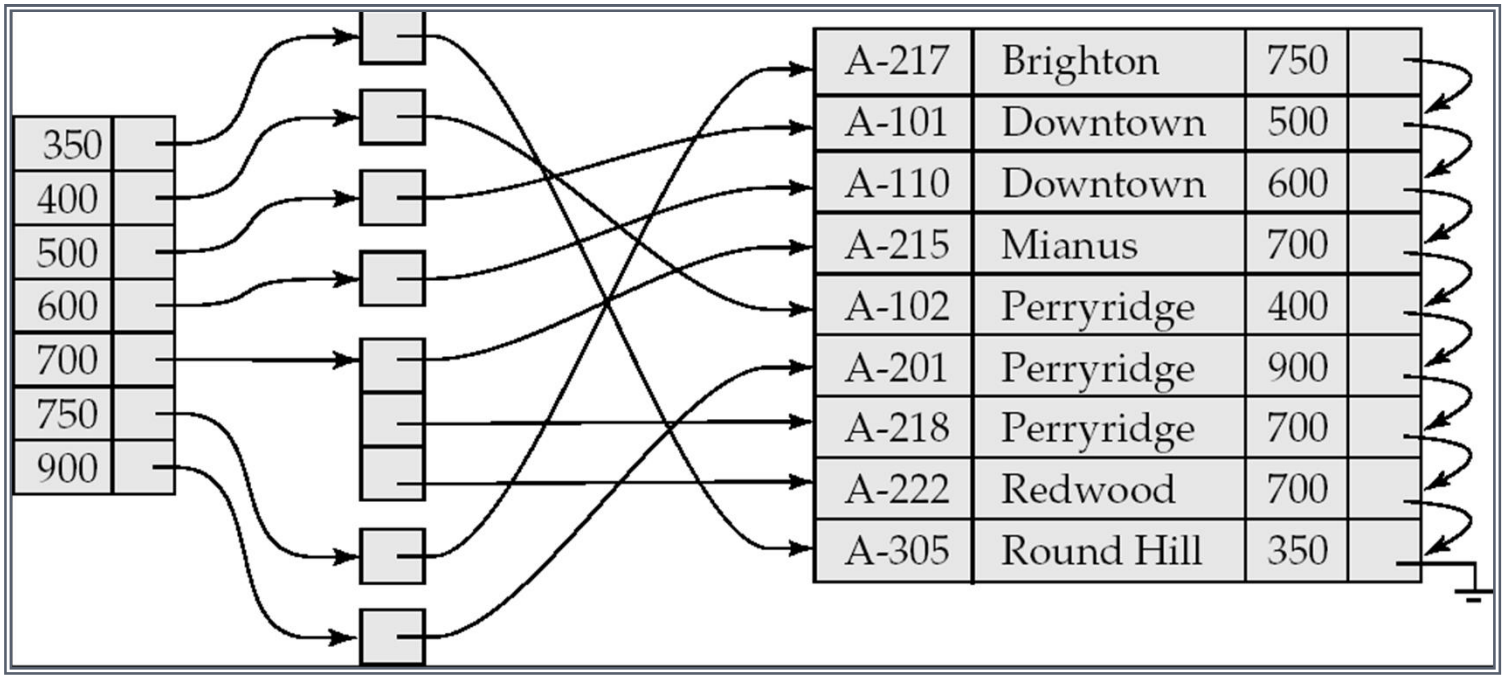
- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.
- Single-level index deletion:
  - **Dense indices** – deletion of search-key: similar to file record deletion.
  - **Sparse indices** –
    - if deleted key value exists in the index, the value is replaced by the next search-key value in the file (in search-key order).
    - If the next search-key value already has an index entry, the entry is deleted instead of being replaced.



Index Update: Record Insertion

- Single-level index insertion:
  - Perform a lookup using the key value from inserted record
  - **Dense indices** – if the search-key value does not appear in the index, insert it.
  - **Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.
    - If a new block is created, the first search-key value appearing in the new block is inserted into the index.
- Multilevel insertion (as well as deletion) algorithms are simple extensions of the single-level algorithms

Secondary Indices Example



Secondary index on *balance* field of *account*

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

Indices on Multiple Keys

- A search key containing more than one attribute is referred as a composite search key
- If the index attributes are A1, …, An then the tuple of values can be represented of the form (a1,…,an)
- The ordering of search key is lexicographic ordering
- For example: for the case of two attributes search keys, (a1,a2)<(b1,b2) if either a1<b1 or a1=b1 and a2<b2

Primary and Secondary Indices

- Indices offer substantial benefits when searching for records.
- BUT: Updating indices imposes overhead on database modification --when a file is modified, every index on the file must be updated,
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - Each record access may fetch a new block from disk
  - Block fetch requires about 5 to 10 micro seconds, versus about 100 nanoseconds for memory access

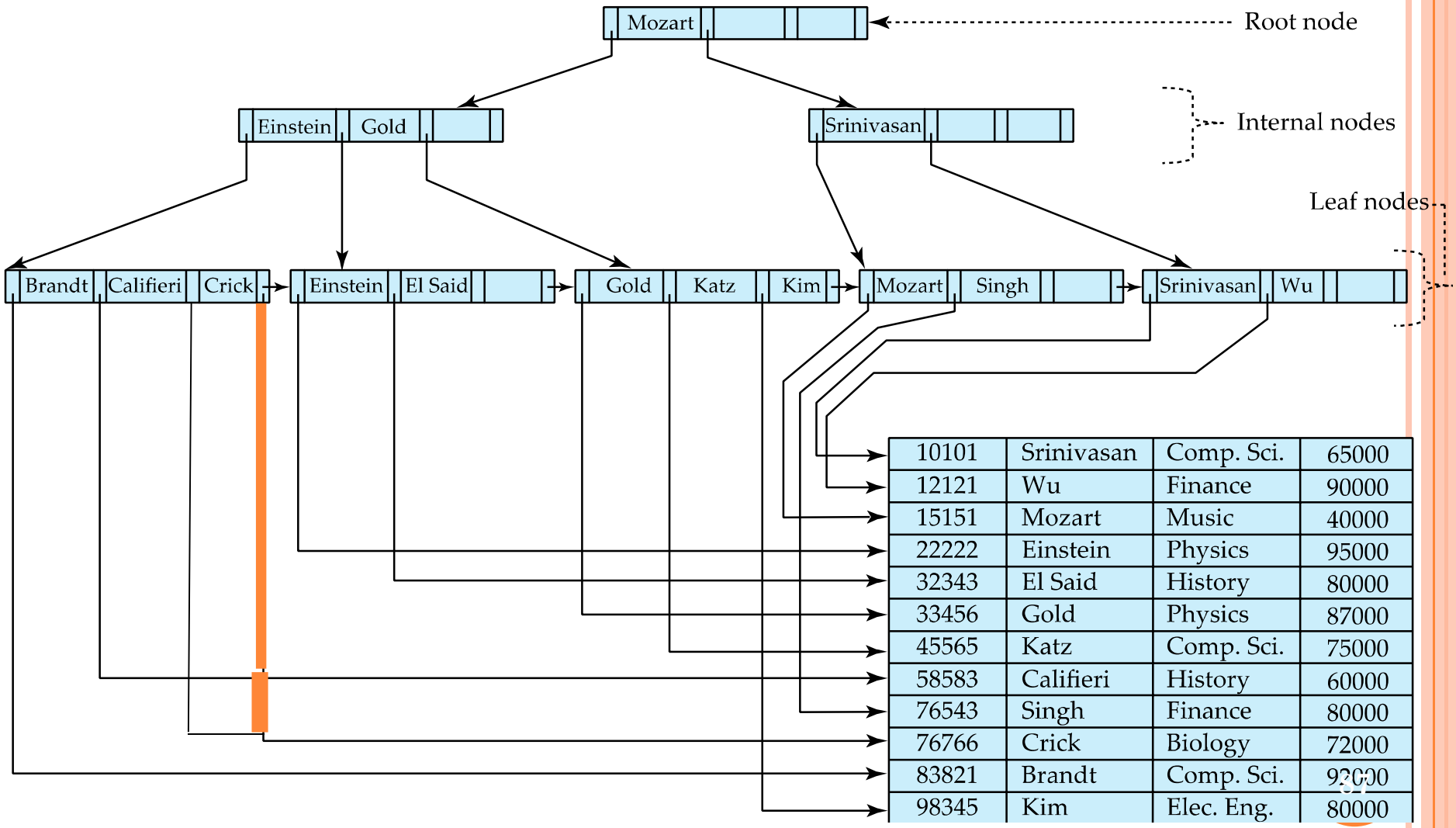
Indexed Sequential File

- Disadvantage of indexed-sequential files
  - Performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required.

B^+-Tree Index 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

Example of B+-Tree



B+-Tree Index Files (Cont.)

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

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n–1)/2 \rceil$ and $n–1$ values
- 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	...	P_{n-1}	K_{n-1}	P_n

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

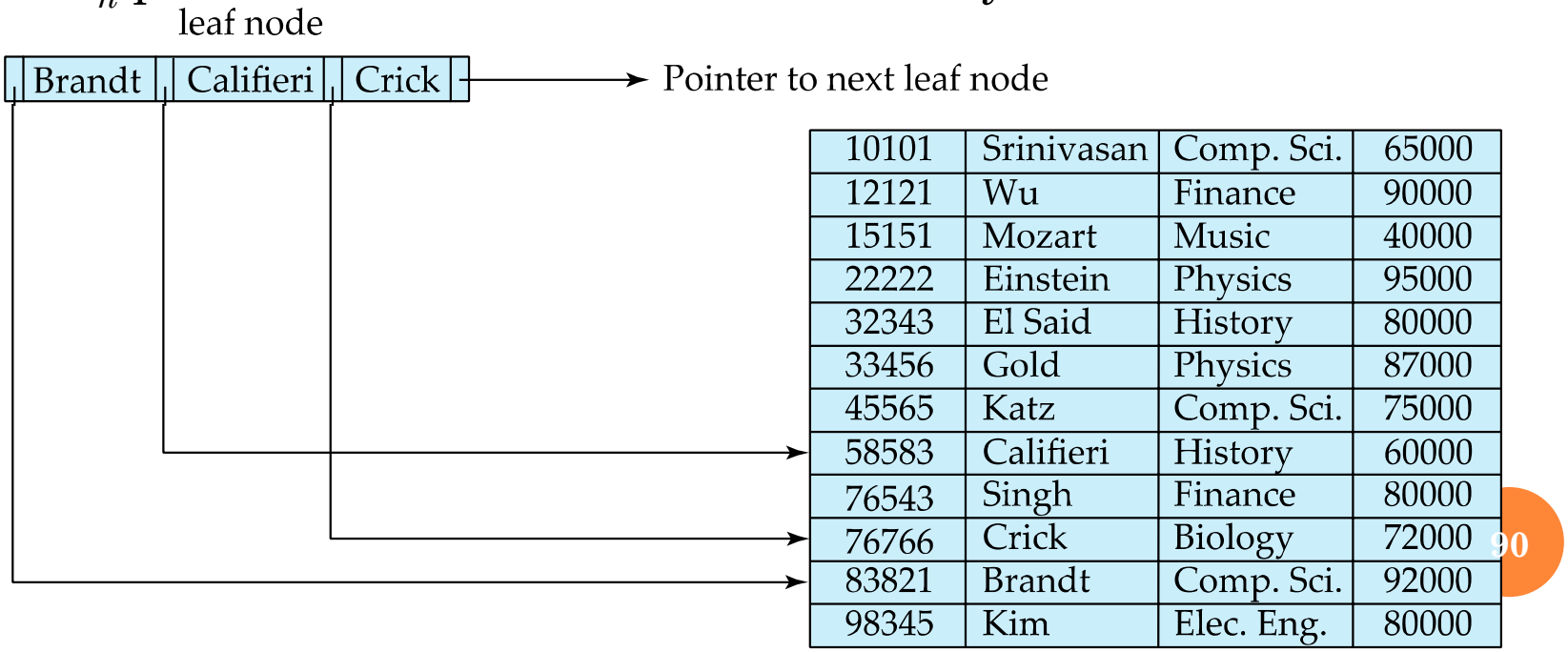
$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

(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



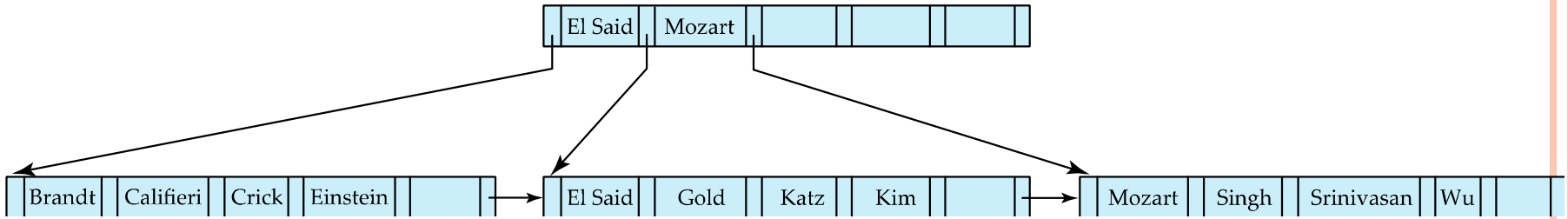
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

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 $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
  - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}
  - General structure

EXAMPLE OF B+-TREE

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



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

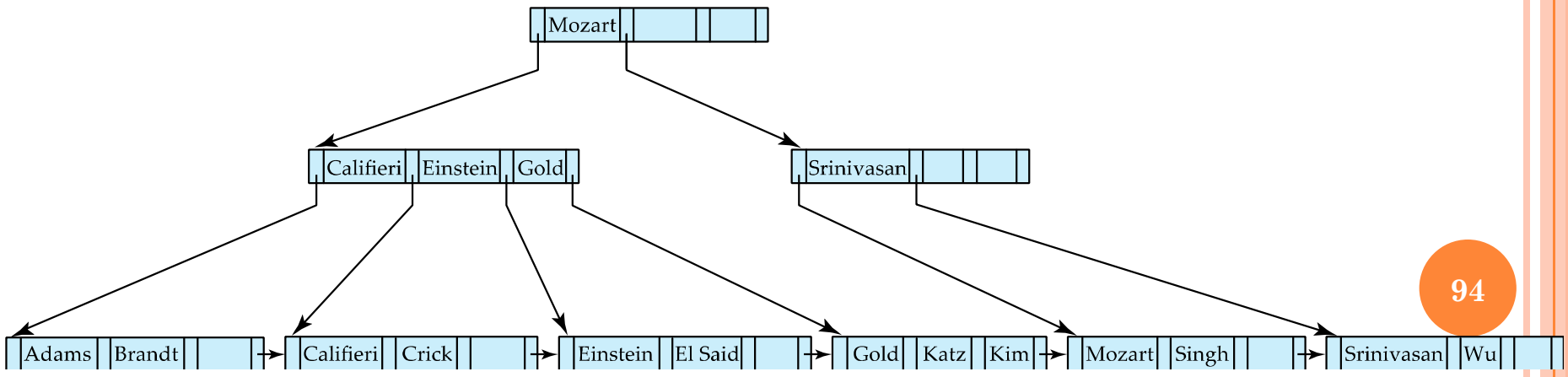
Observations about B^+-trees

- Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close.
- The non-leaf levels of the B^+-tree form a hierarchy of sparse indices.
- The B^+-tree contains a relatively small number of levels
  - 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 (as we shall see).

Queries on B+-Trees

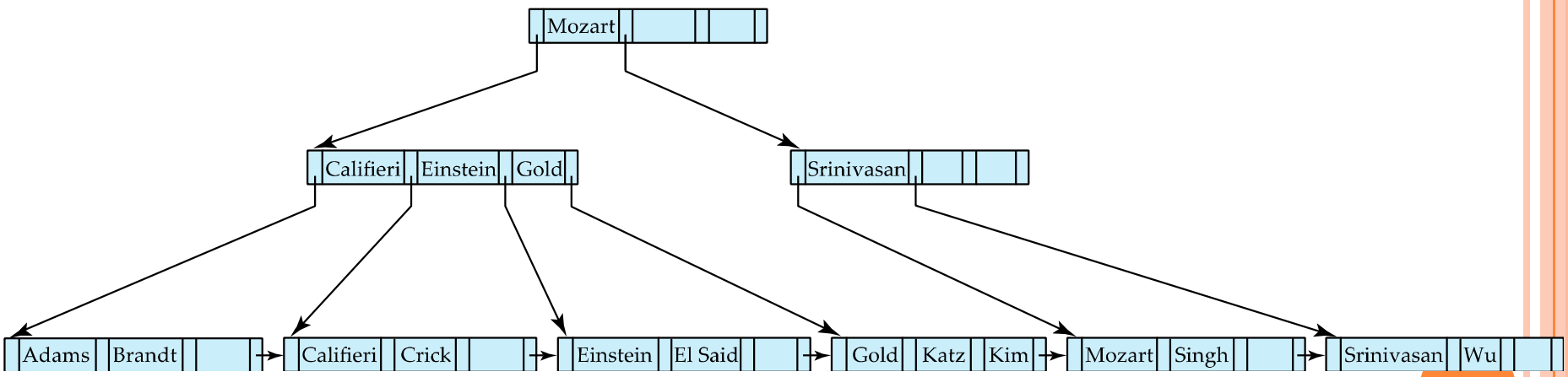
function *find*(*v*)

1. *C=root*
2. **while** (C is not a leaf node)
    1. Let *i* be least number s.t. $V \leq K_i$.
    2. **if** there is no such number *i then*
    3. Set *C* = *last non-null pointer in C*
    4. **else if** (v = C.K_i) Set C = P_{i+1}
    5. **else set** *C* = C.P_i
3. **if** for some *i*, $K_i = V$ **then** return C.P_i
4. **else** return null /* no record with search-key value *v* exists. */



QUERIES ON B+-TREES (CONT.)

- **Range queries** find all records with search key values in a given range
  - See book for details of **function** *findRange*(*lb*, *ub*) which returns set of all such records
  - Real implementations usually provide an iterator interface to fetch matching records one at a time, using a *next*() function



Queries on B+-Trees (Cont.)

- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and n is typically around 200 (with search key as 12 bytes and pointer size as 8 bytes).
- With 1 million search key values and $n = 100$
  - at most $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup traversal from root to leaf.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

NON-UNIQUE KEYS

- If a search key a_i is not unique, create instead an index on a composite key (a_i , A_p), which is unique
  - A_p could be a primary key, record ID, or any other attribute that guarantees uniqueness
- Search for $a_i = v$ can be implemented by a range search on composite key, with range $(v, -\infty)$ to $(v, +\infty)$
- But more I/O operations are needed to fetch the actual records
  - If the index is clustering, all accesses are sequential
  - If the index is non-clustering, each record access may need an I/O operation

UPDATES ON B^+-TREES: INSERTION

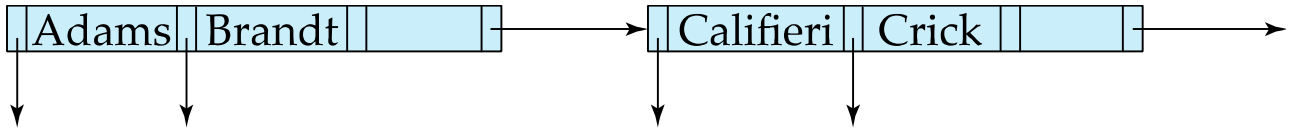
Assume record already added to the file. Let

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

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

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

- Splitting a leaf node:
  - take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be p, and let k be the least key value in p. Insert (k,p) in the parent of the node being split.
  - If the parent is full, split it and **propagate** the split further up.
- Splitting of nodes proceeds upwards till a node that is not full is found.
  - In the worst case the root node may be split increasing the height of the tree by 1.

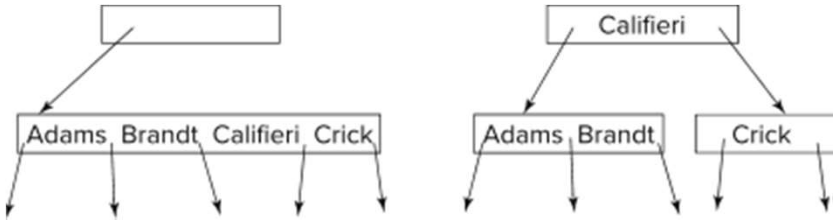


Result of splitting node containing Brandt, Califieri and Crick on inserting Adams

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

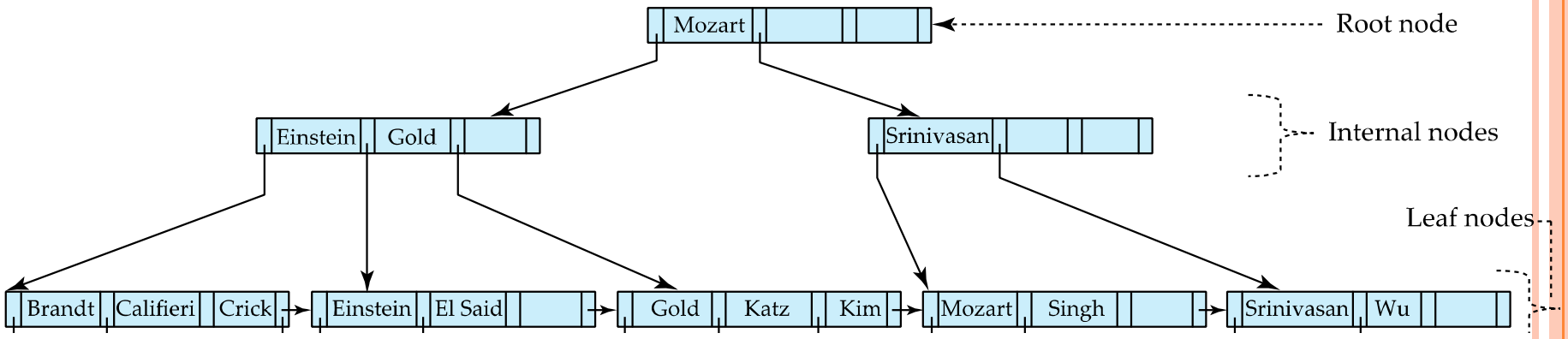
INSERTION IN B+-TREES (CONT.)

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

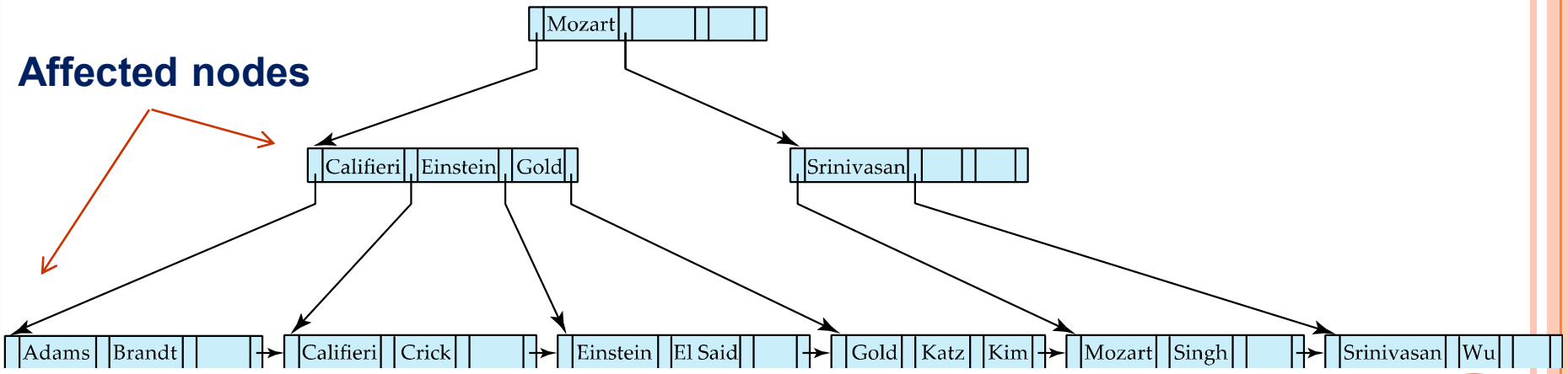


- **Read pseudocode in book!**

B+-Tree Insertion

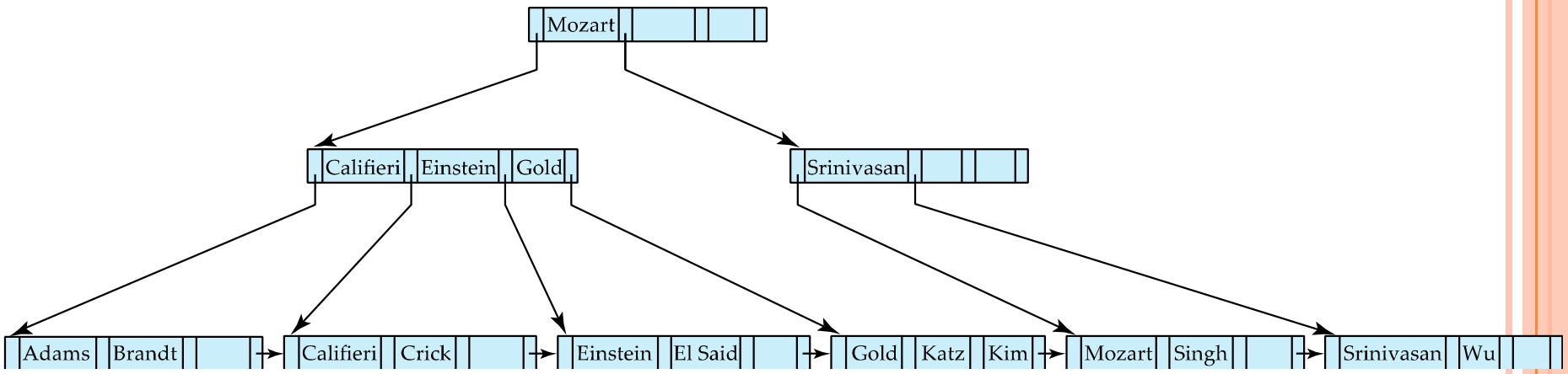


B+-Tree before insertion of "Adams"



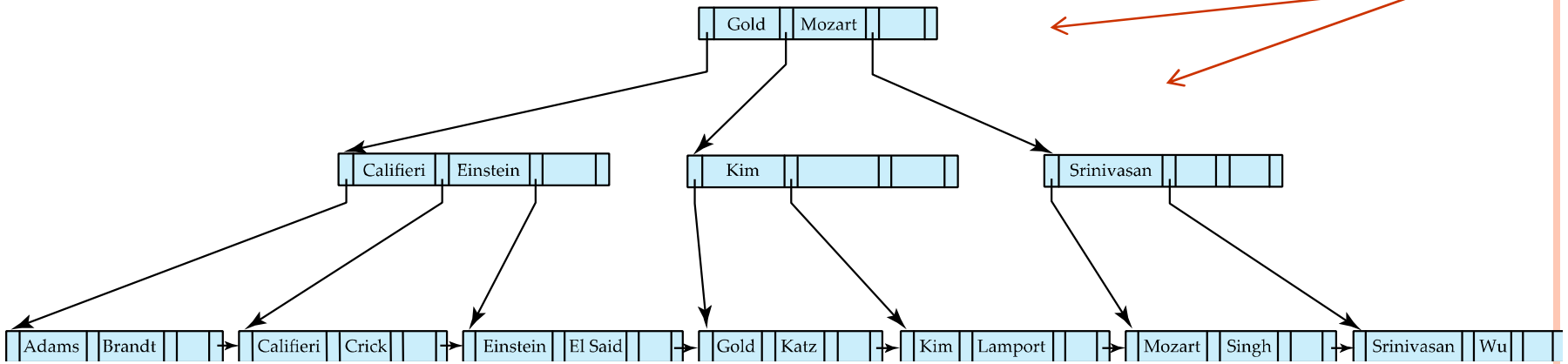
B+-Tree after insertion of "Adams"

B+-Tree Insertion



B+-Tree before insertion of "Lamport"

Affected nodes



Affected nodes

B+-Tree after insertion of "Lamport"

UPDATES ON B+-TREES: DELETION

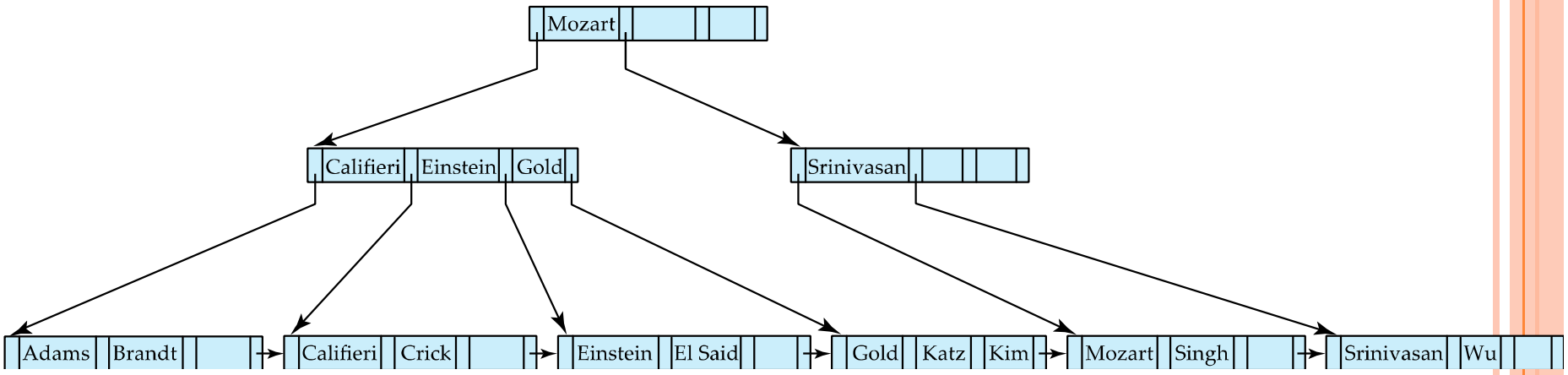
Assume record already deleted from file. Let *V* be the search key value of the record, and *Pr* be the pointer to the record.

- Remove (*Pr, V*) from the leaf node
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings***:
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.

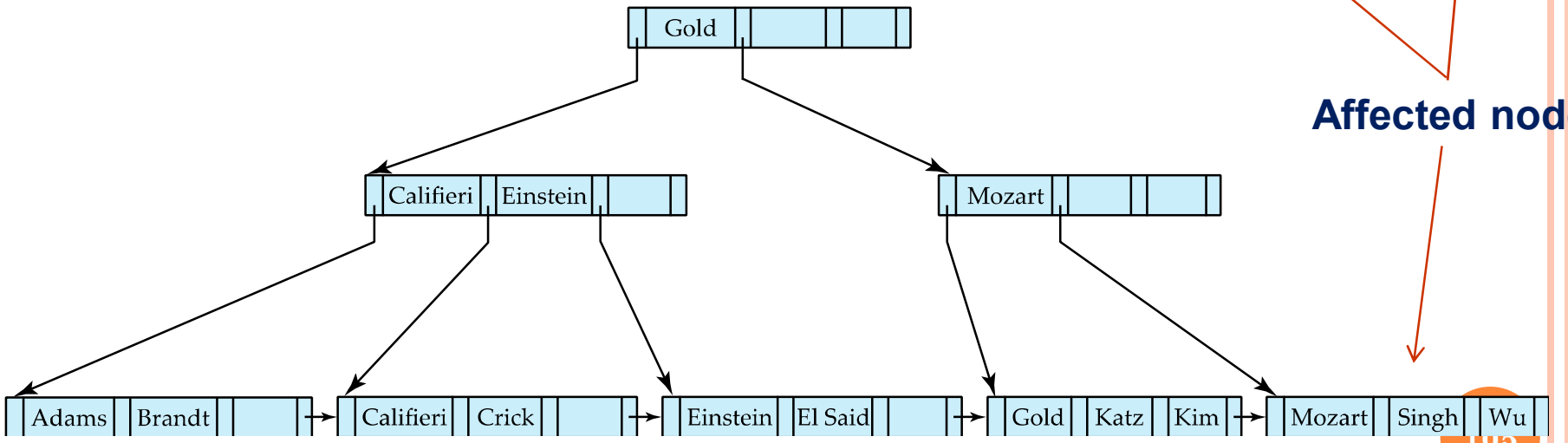
Updates on B+-Trees: Deletion

- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
  - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

Examples of B+-Tree Deletion



Before deleting "Srinivasan"

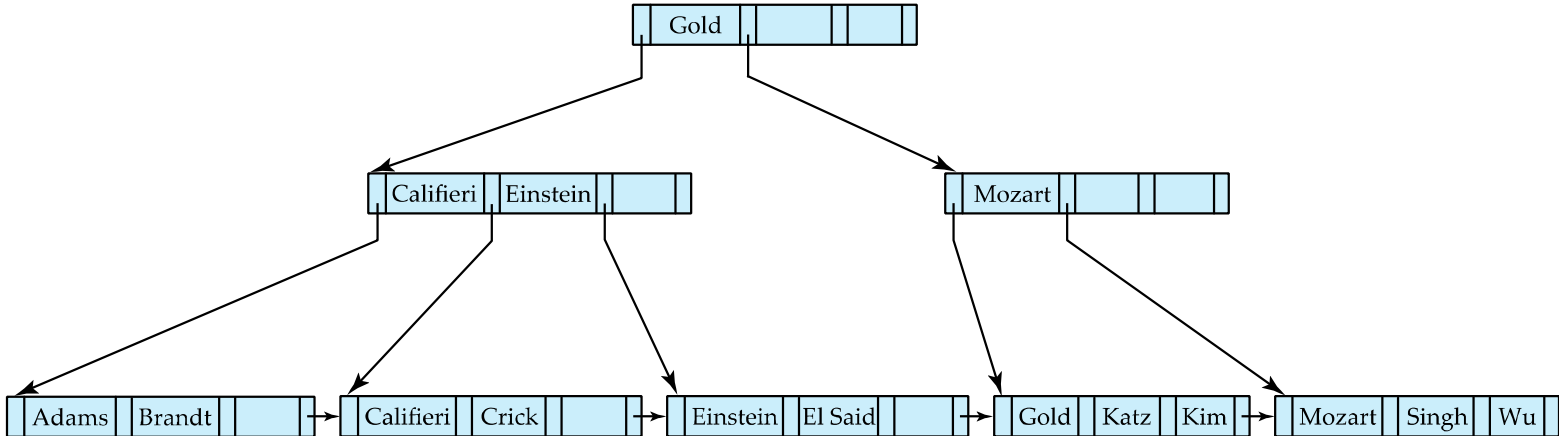


Affected nodes

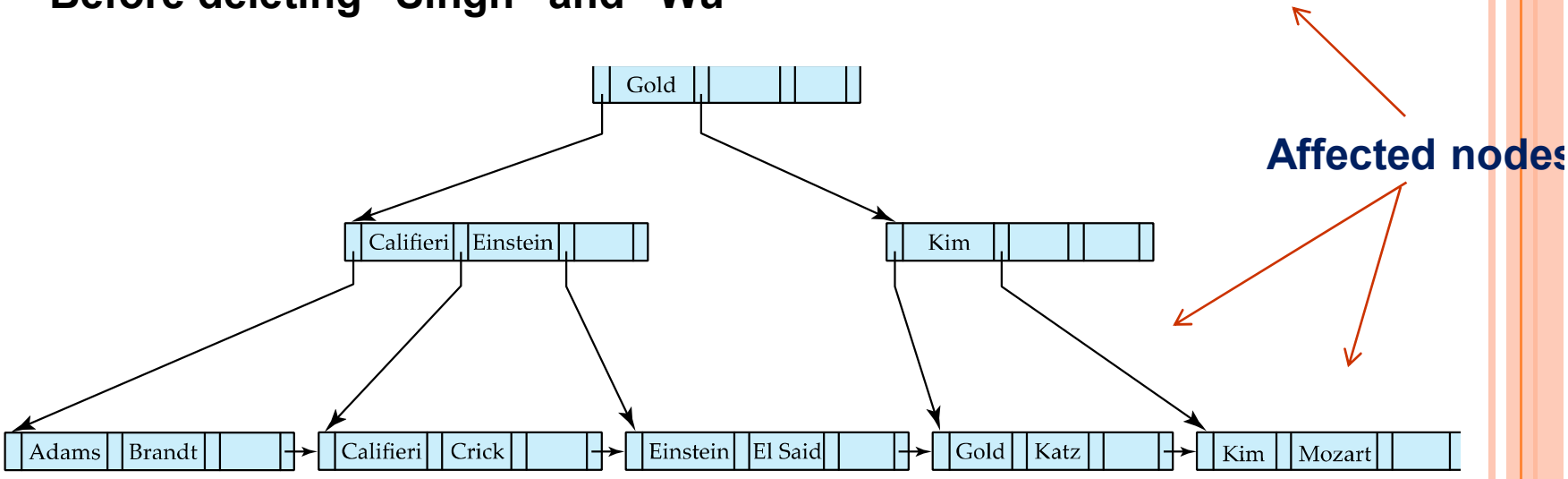
- Deleting "Srinivasan" causes **merging** of under-full leaves

After deleting "Srinivasan"

Examples of B+-Tree Deletion (Cont.)



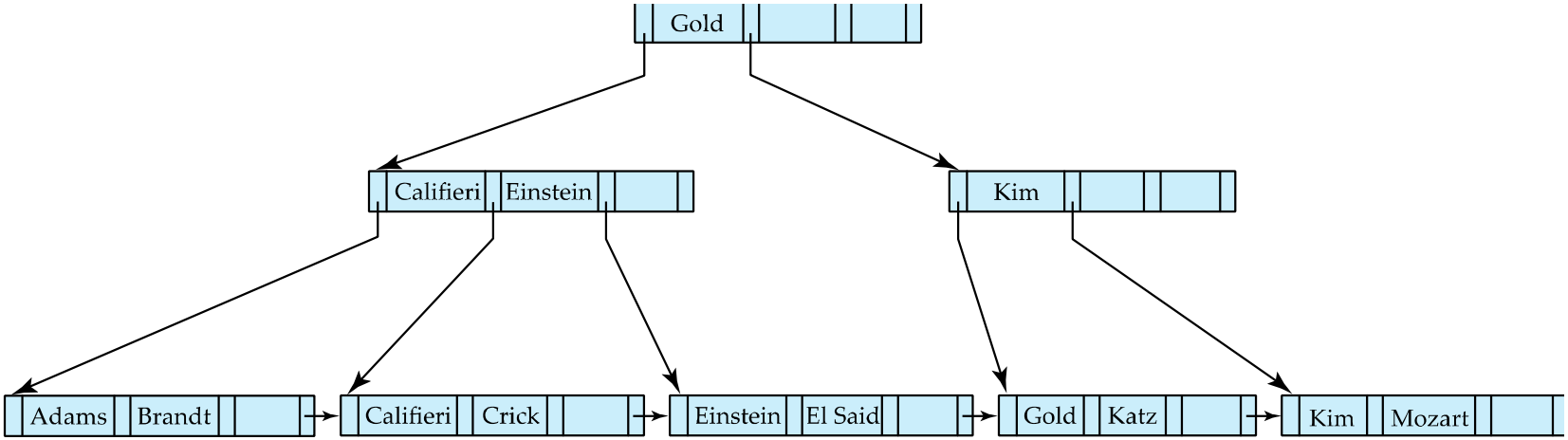
Before deleting "Singh" and "Wu"



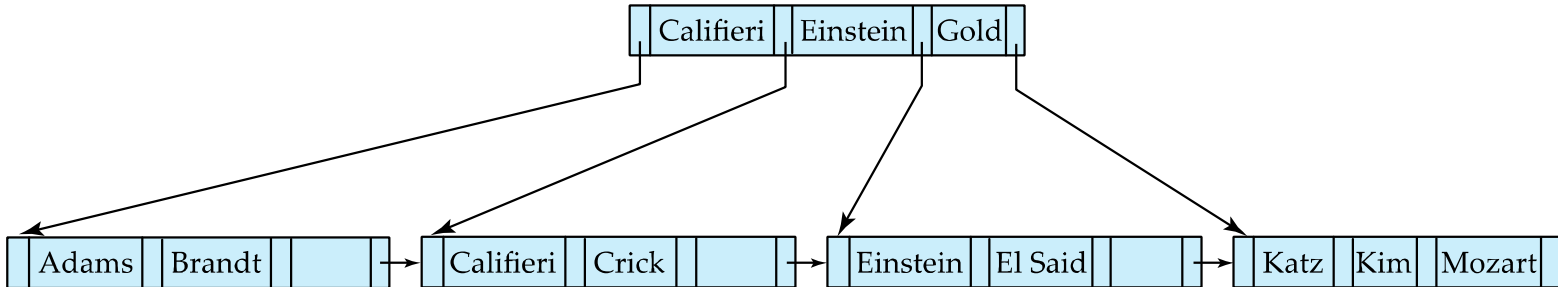
- Leaf containing Singh and Wu became underfull, and **borrowed a value** Kim from its left sibling
- Search-key value in the parent changes as a result

After deleting "Singh" and "Wu"

Example of B+-tree Deletion (Cont.)



Before deletion of "Gold"



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

After deletion of "Gold"

Complexity of Updates

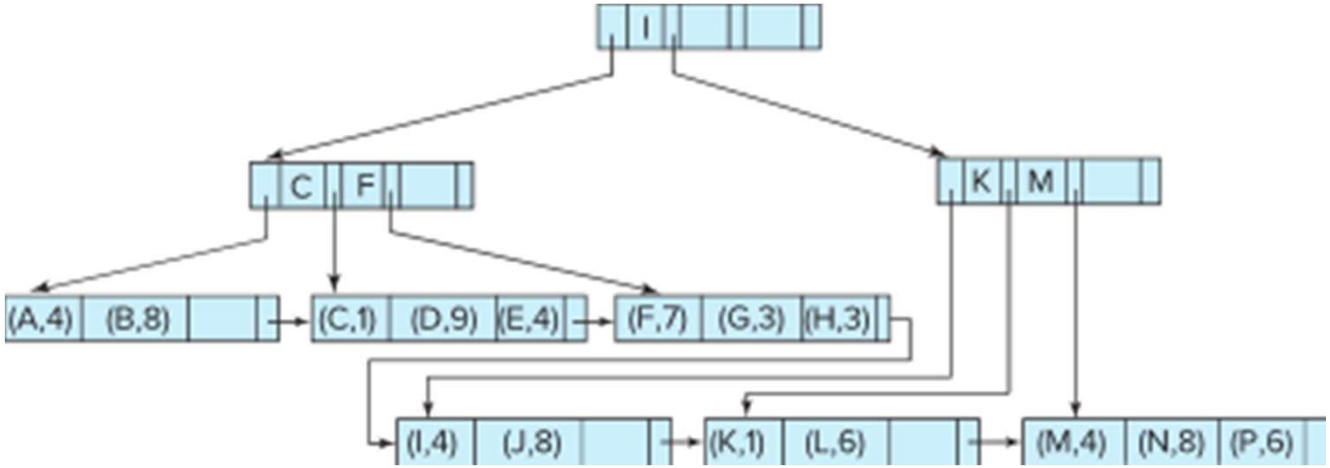
- Cost (in terms of number of I/O operations) of insertion and deletion of a single entry proportional to height of the tree
  - With K entries and maximum fanout of n, worst case complexity of insert/delete of an entry is $O(\log_{\lceil n/2 \rceil}(K))$
- In practice, number of I/O operations is less:
  - Internal nodes tend to be in buffer
  - Splits/merges are rare, most insert/delete operations only affect a leaf node
- Average node occupancy depends on insertion order
  - 2/3rds with random, ½ with insertion in sorted order

B^+-Tree File Organization

- B^+-Tree File Organization:
  - Leaf nodes in a B^+-tree file organization store records, instead of pointers
  - Helps keep data records clustered even when there are insertions/deletions/updates
- Leaf nodes are still required to be half full
  - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B^+-tree index.

B+-Tree File Organization (Cont.)

- Example of B+-tree File Organization



- Good space utilization important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries

B-Tree Index Files

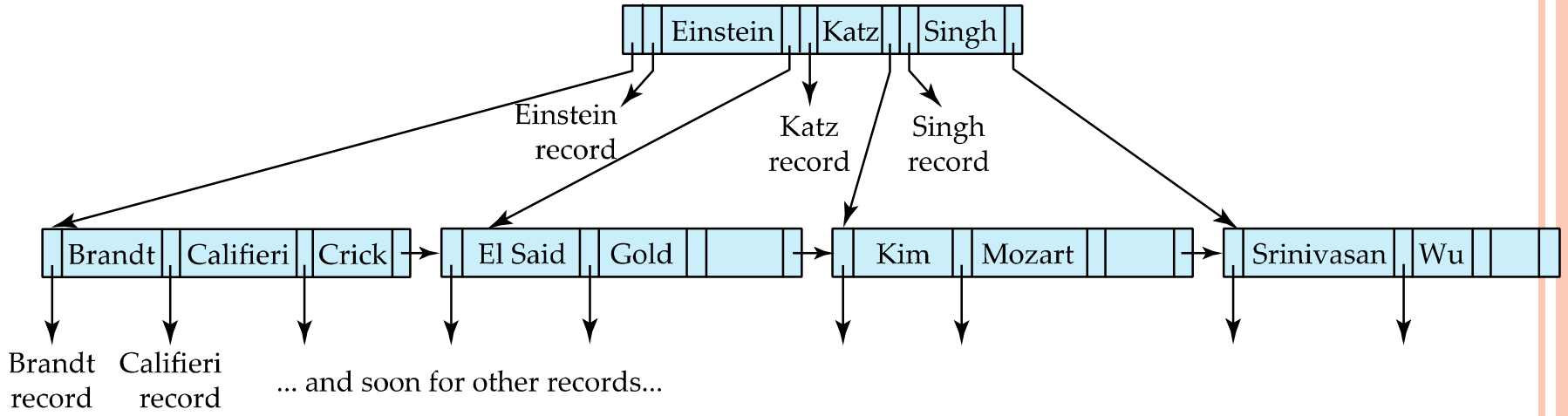
- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Generalized B-tree leaf node

(a)

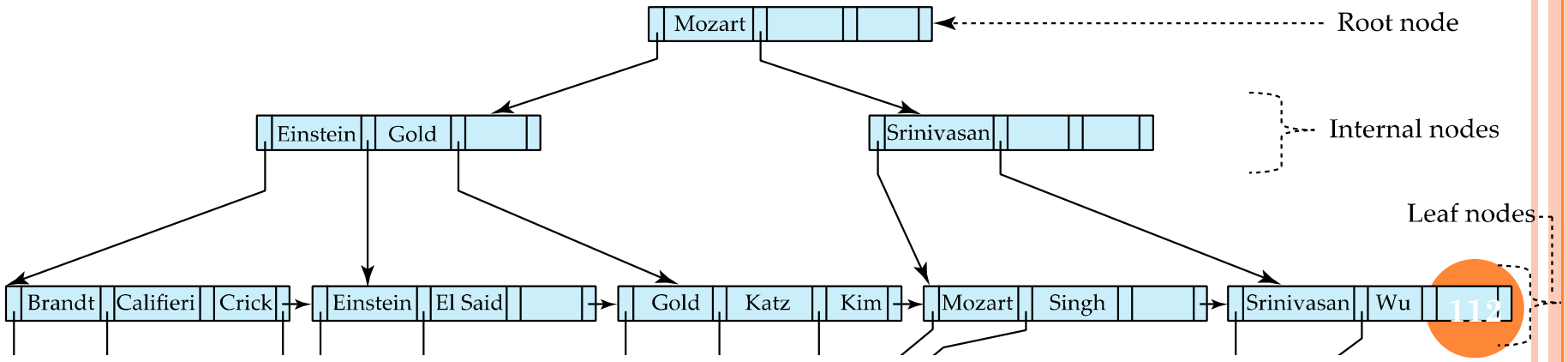
(b)

- Nonleaf node – pointers Bi are the bucket or file record pointers.

B-Tree Index File Example



B-tree (above) and B+-tree (below) on same data



B-Tree Index Files (Cont.)

- Advantages of B-Tree indices:
  - May use less tree nodes than a corresponding B^+-Tree.
  - Sometimes possible to find search-key value before reaching leaf node.
- Disadvantages of B-Tree indices:
  - Only small fraction of all search-key values are found early
  - Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B^+-Tree
  - Insertion and deletion more complicated than in B^+-Trees
  - Implementation is harder than B^+-Trees.
- Typically, advantages of B-Trees do not out weigh disadvantages.

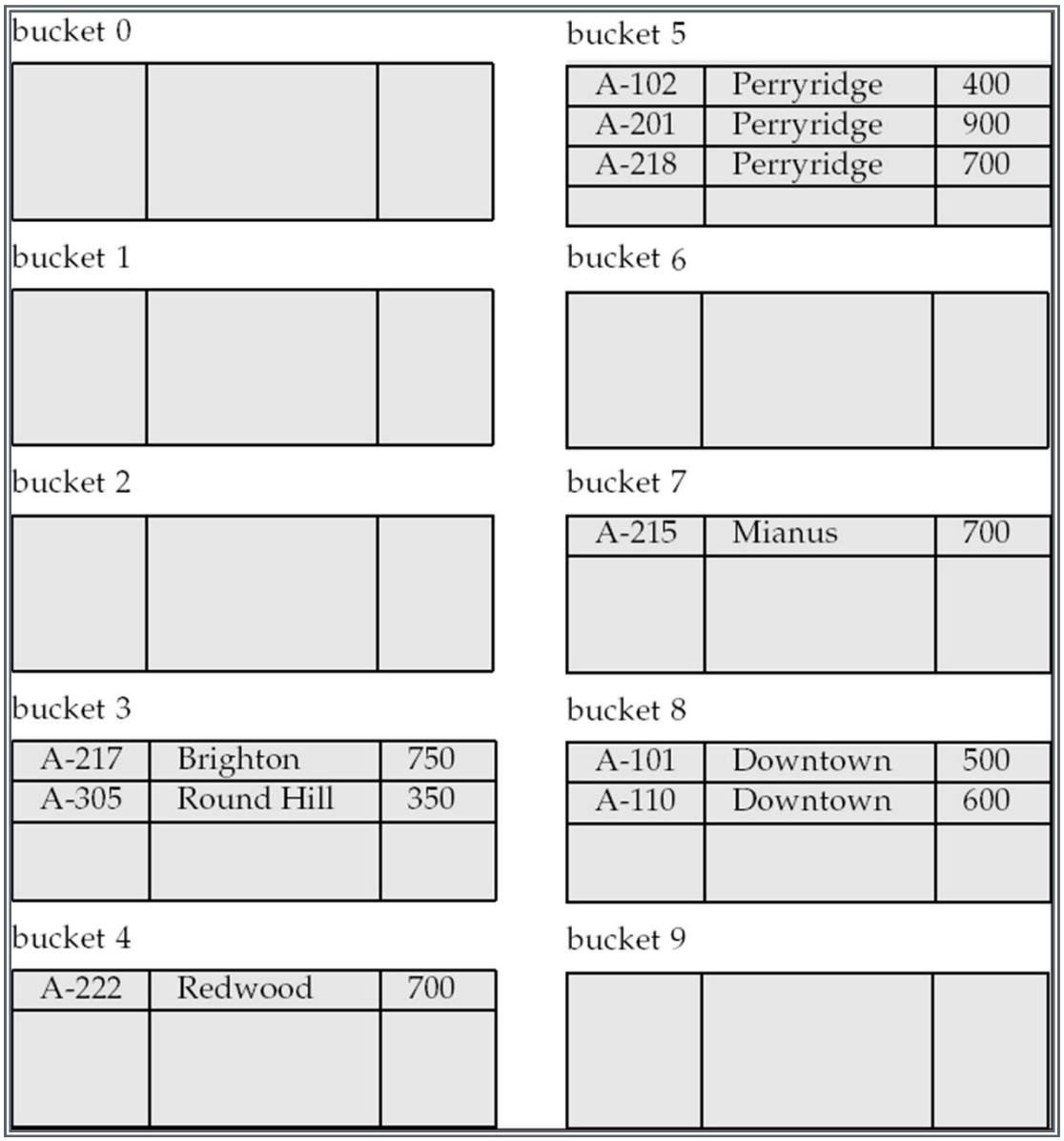
STATIC HASHING

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

Example of Hash File Organization

- Let's consider the hash file organization of *account* file, using *branch_name* as search key
- There are 10 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 10*
  - E.g. h(Perryridge) = 5 h(Round Hill) = 3 h(Brighton) = 3

Example of Hash File Organization



Hash file organization of *account* file, using *branch_name* as key

HASH FUNCTIONS

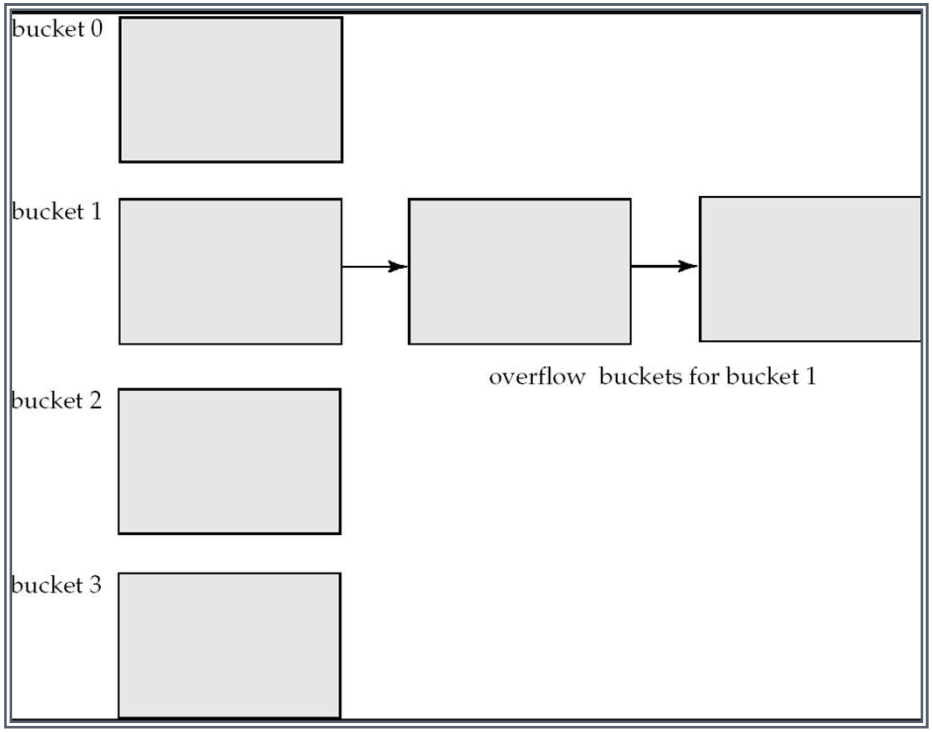
- Worst hash function maps all search-key values to the same bucket;
  - this makes access time proportional to the number of search-key values in the file.
- Hash function should be-
  - uniform
  - random
- Typical hash value should not be correlated to any externally visible ordering on the search key value
  - Like alphabet ordering, length of the search keys, etc.

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 (Cont.)

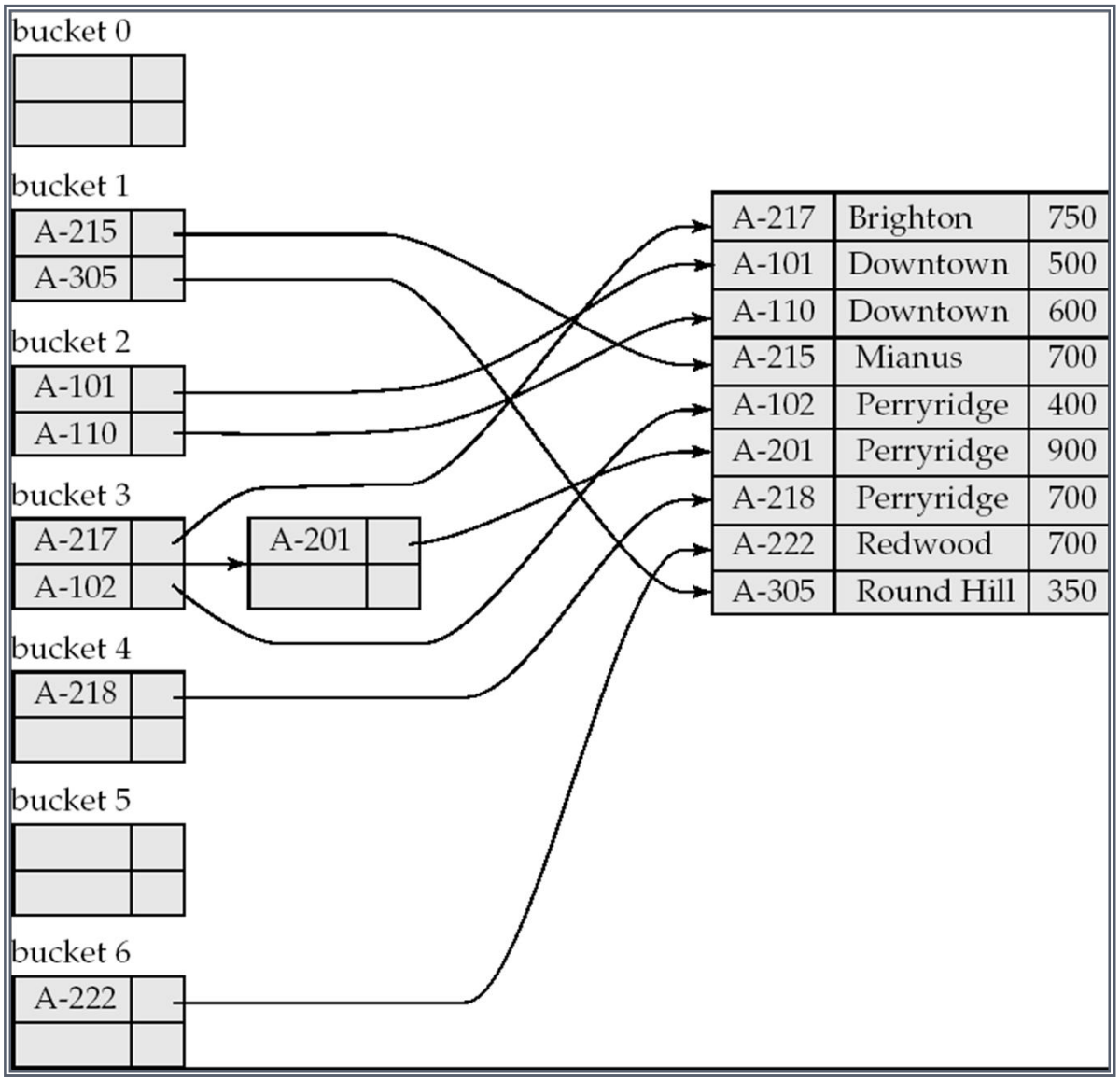
- Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.



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.
- 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



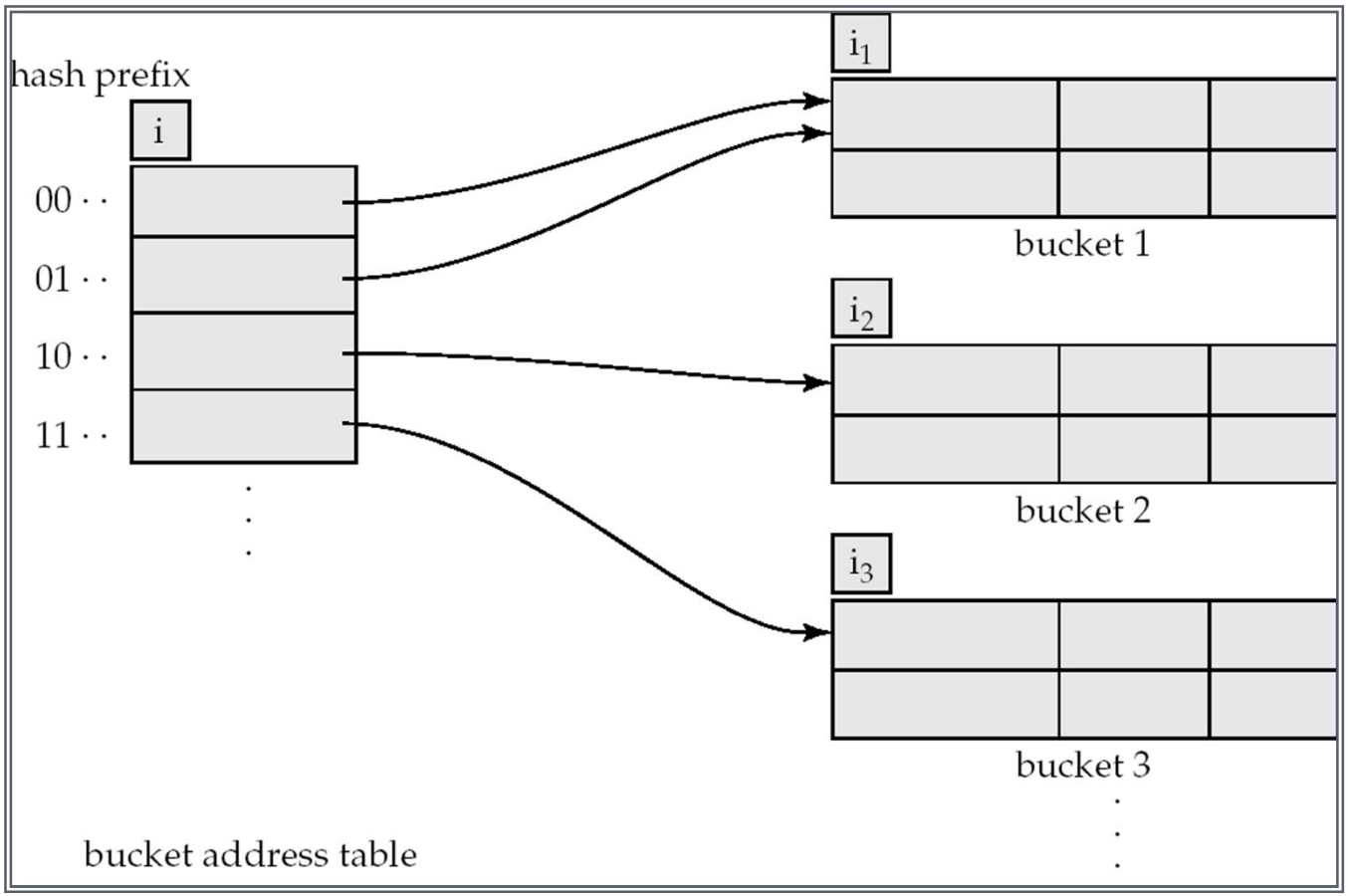
Deficiencies of Static Hashing

- 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 underfull).
  - 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, $0 \le i \le 32$.
    - 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 typically $< 2^i$
    - 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 $i_1 = i - 1$

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
    - Overflow buckets used instead in some cases

Insertion in Extendable Hash Structure (Cont)

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

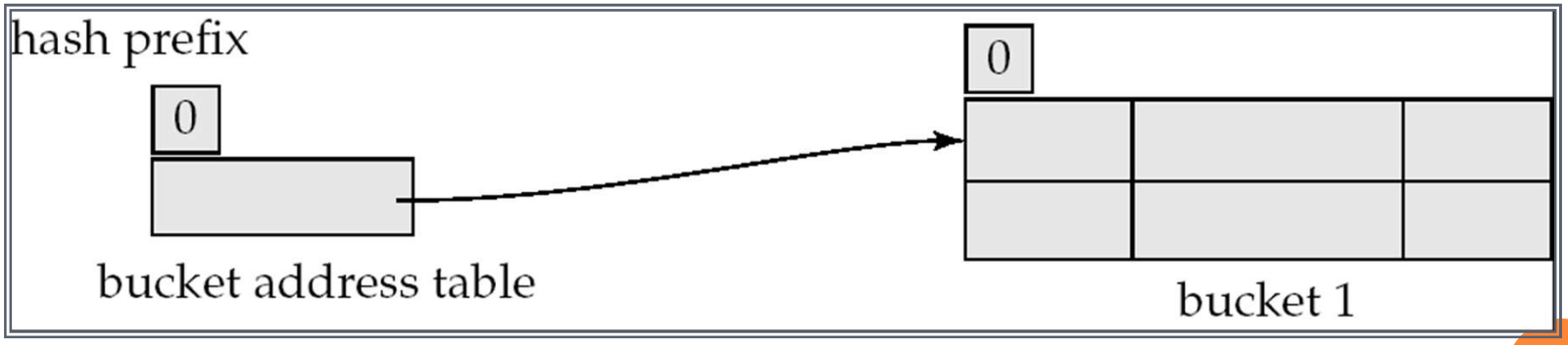
- If $i > i_j$ (more than one pointer to bucket j)
  - allocate a new bucket z, and set $i_j = i_z = (i_j + 1)$
  - Update the second half of the bucket address table entries originally pointing to j, to point to z
  - remove each record in bucket j and reinsert (in j or z)
  - recompute new bucket for 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

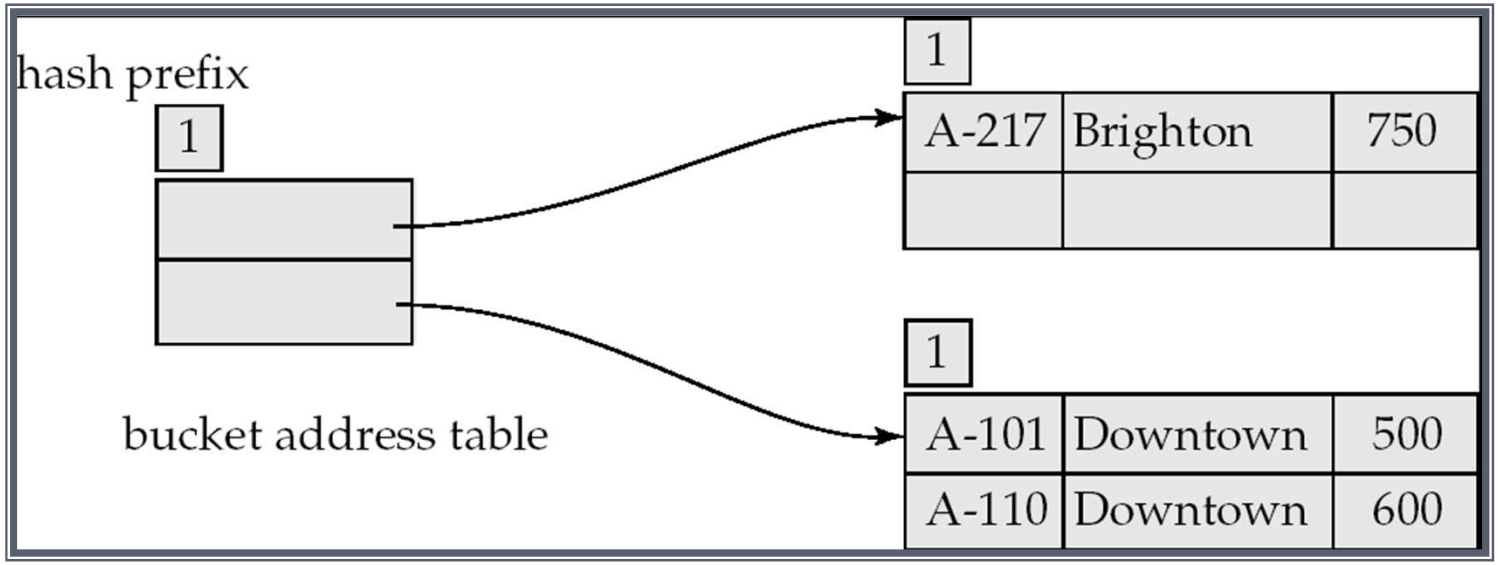
branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



Initial Hash structure, bucket size = 2

EXAMPLE (CONT.)

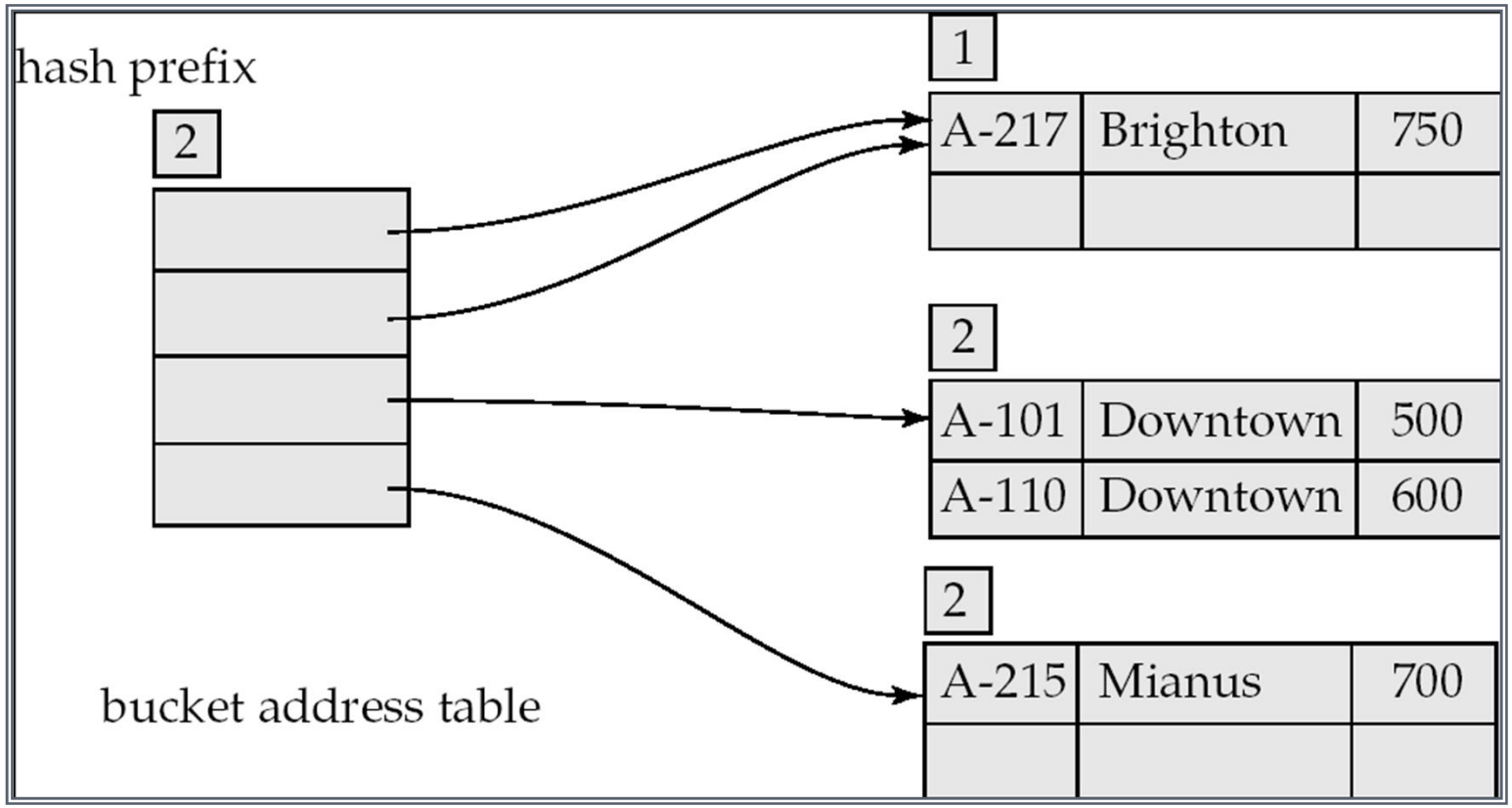
- Hash structure after insertion of one Brighton and two Downtown records



Now Insert Mianus record

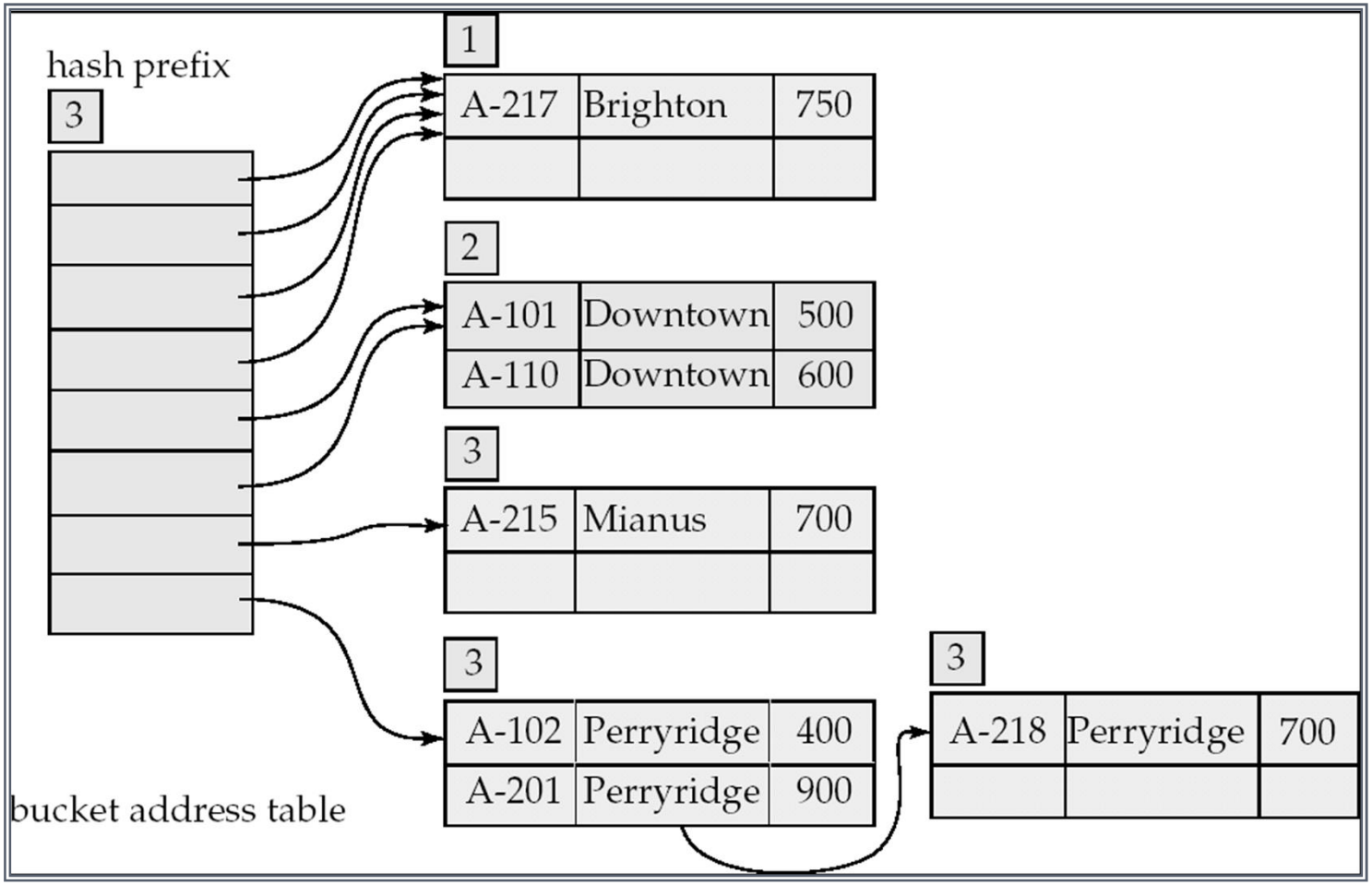
EXAMPLE (CONT.)

Hash structure after insertion of Mianus record



Now insert Perryridge record three times

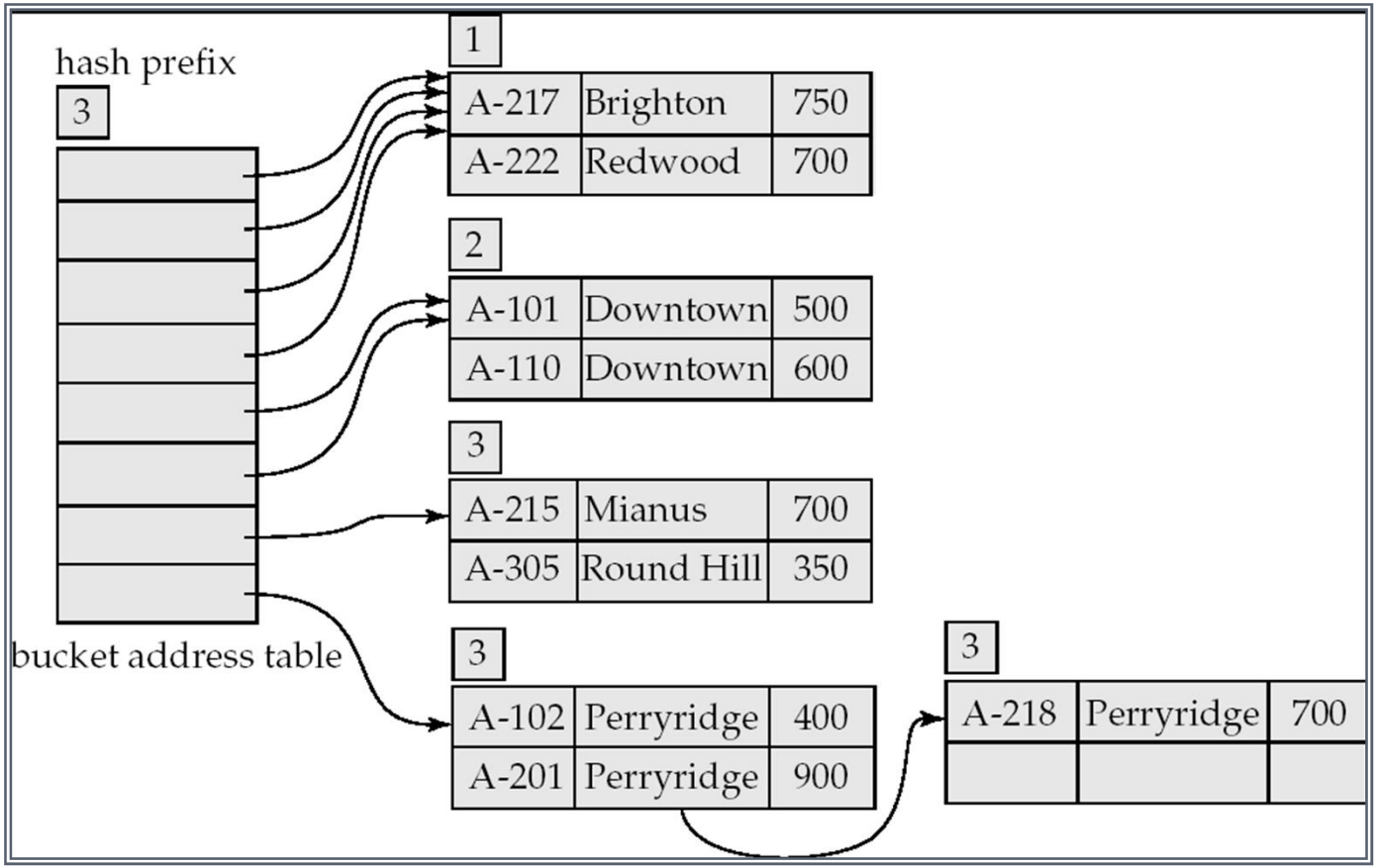
EXAMPLE (CONT.)



Hash structure after insertion of three Perryridge records

EXAMPLE (CONT.)

- Hash structure after insertion of Redwood and Round Hill records



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 file organization to store 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

LINEAR HASHING

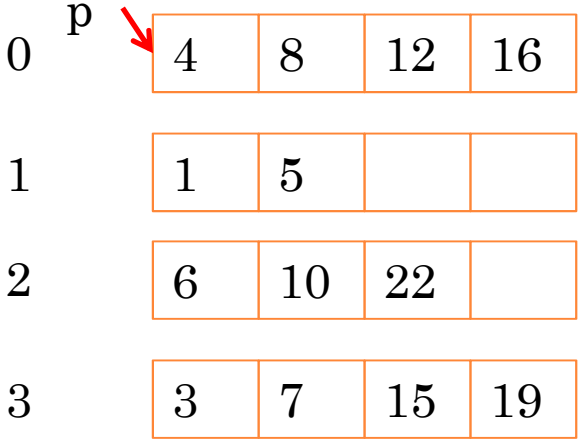
- Another dynamic hashing technique
- It grows and shrinks one bucket at a time
- Unlike extendable hashing, it does not use a bucket address table
- When an overflow occurs it does not always split the overflow bucket
- The no. of buckets grows and shrinks in a linear fashion
- Overflow are handled by creating a chain of buckets
- Hashing function changes dynamically
- Atmost two hashing functions can be used at any given instant

Linear Hashing: Initial Layout

- The linear hashing has m initial buckets labeled 0 through m-1
- An initial hashing function $h_0(k) = f(k) \% m$
- For simplicity, we assume that $h_0(k) = k \% m$
- A pointer p which points to the bucket to be split next
- The bucket split occurs whenever an overflow bucket is generated

Bucket#

Overflow buckets

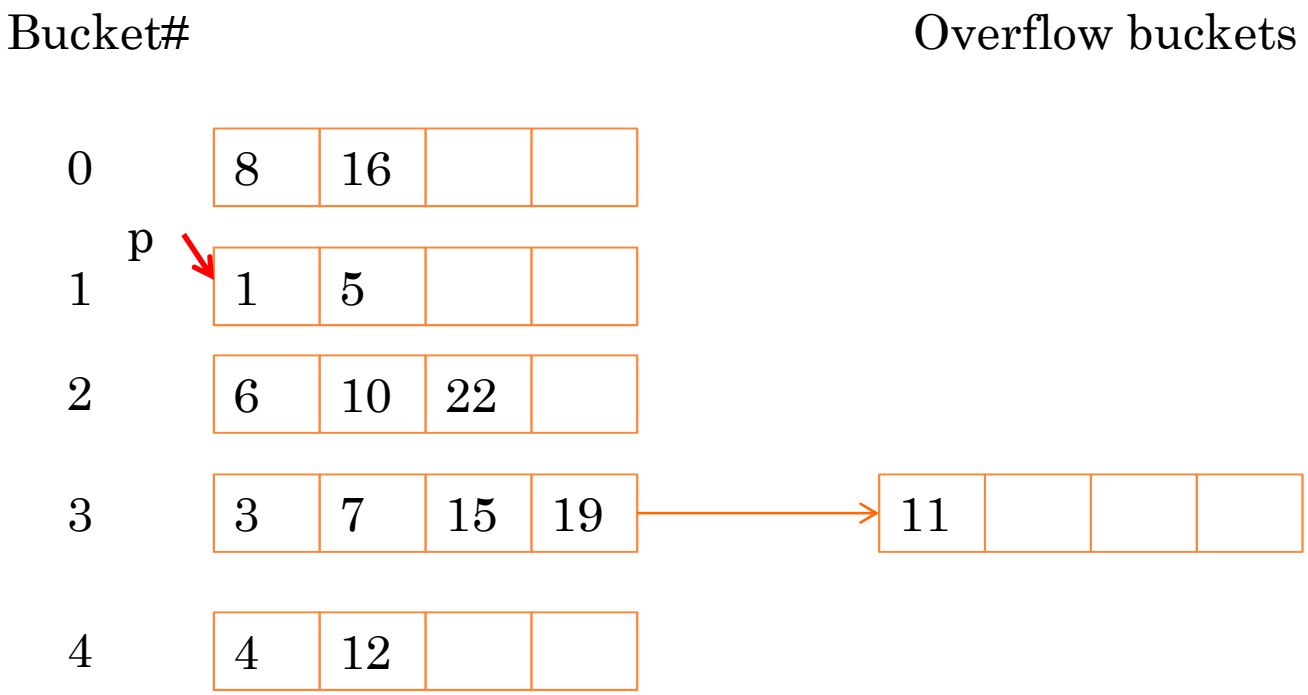


Here m=4, p=0, $h_0(k)=k \%4$

Linear Hashing: Bucket Split

- When the first overflow occurs (it may occur in any bucket)
  - Bucket 0 which is pointed by p is split into two buckets (original bucket# 0 and a new bucket bucket# m)
  - A new empty bucket is also added in the overflown bucket to accommodate the overflow
  - The search values originally mapped into bucket 0 (using function h_0) are now distributed between buckets 0 and m using a new hashing function h_1

Now let's try to insert a new record with key 11



Here p=1, $h_0(k)=k \%4$, $h_1(k)=k \%8$

- In case of insertion and overflow condition,
  - If ($h_0(k)<p$) then use $h_1(k)$
  - a new split that will attach a new bucket *m+1* and the contents of bucket *1* will be distributed using h_1 between buckets *1* and *m+1*
- For every split, *p* is increased by *1*
- The necessary property for linear hashing to work-
  - The search values that were originally mapped by h_0 to some bucket *j* must be remapped using h_1 to bucket *j* or *j+m*
  - An example of such hashing function would be $h_1(k) = k \% 2m$

Linear Hashing: Round and Hash Function advancement

- After enough overflows, all original *m* buckets will be split
- This marks the end of splitting round 0
- During round 0, p went subsequently from 0 to m-1
- At the end of round 0, the linear hashing scheme has a total of 2m buckets
- Hashing function h_0 is no longer needed as all 2m buckets can be addressed by hashing function h_1
- Variable p is reset to 0 and a new round (namely splitting round 1) starts
- A new hash function h_2 will start to be used

- In general, linear hashing involves a family of hash functions h_0, h_1, h_2, and so on
- Let the initial hash function is $h_0(k) = f(k) \% m$
- Then any later hash function can be defined as $h_i(k) = f(k) \% 2^i m$
- This will guarantee that if h_i hashes a key to bucket j
  - then h_{i+1} will hash the same key to either j or bucket $j+2^i m$

LINEAR HASHING: SEARCHING

- A search scheme is needed to map a key k to a bucket
- It works as follows-
  - If $h_i(k) >= p$ choose bucket $h_i(k)$ since the bucket has not been split yet
  - If $h_i(k) < p$ choose bucket $h_{i+1}(k)$ which can either be $h_i(k)$ or its split image $h_i(k) + 2^i m$

Comparison of Ordered Indexing and Hashing

- The following issues must be considered
  - 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

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, 20000-49999, 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	*name*	*gender*	*address*	*income_level*
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	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)
- 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

BITMAP INDICES (CONT.)

- Bitmap indices generally very small compared with relation size
  - E.g. if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation.
    - If number of distinct attribute values is 8, bitmap is only 1% of relation size
- Deletion needs to be handled properly
  - Existence bitmap to note if there is a valid record at a record location
  - Needed for complementation
    - NOT(A=v): *(NOT bitmap-A-v) AND ExistenceBitmap*
- Should keep bitmaps for all values, even null value
  - To correctly handle SQL null semantics for NOT(A=v):
    - intersect above result with (NOT *bitmap-A-Null*)