



BBM371- Data Management

Lecture 1: Course policies,
Introduction to DBMS

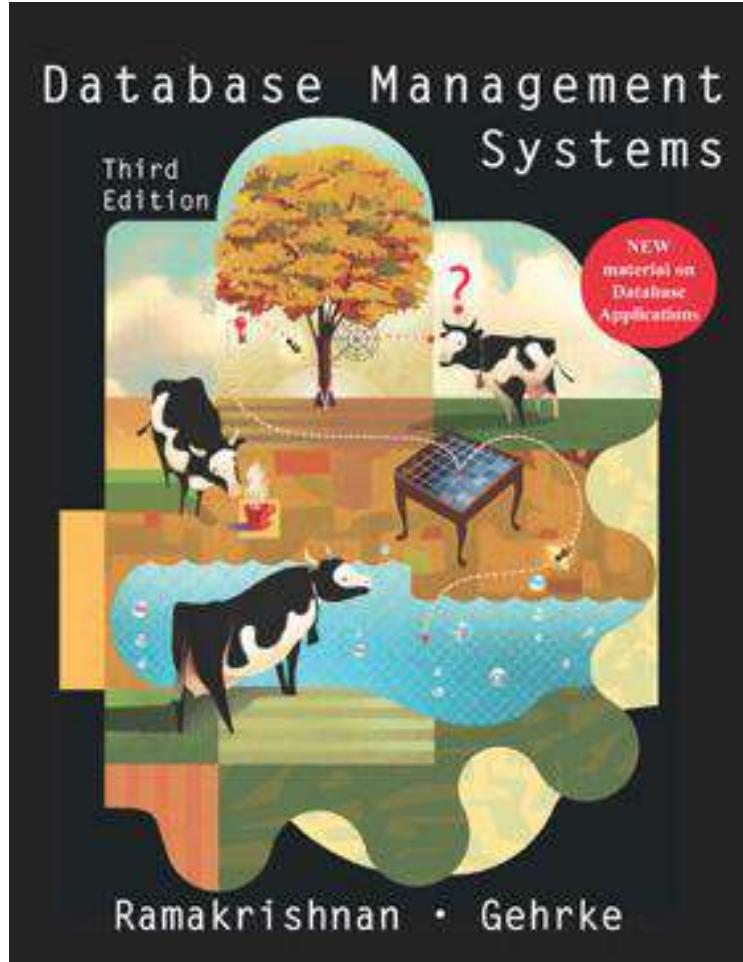
10.10.2019

Today

- ▶ **Introduction**
 - ⌚ About the class
 - ⌚ Organization of this course
- ▶ **Introduction to Database Management Systems (DBMS)**

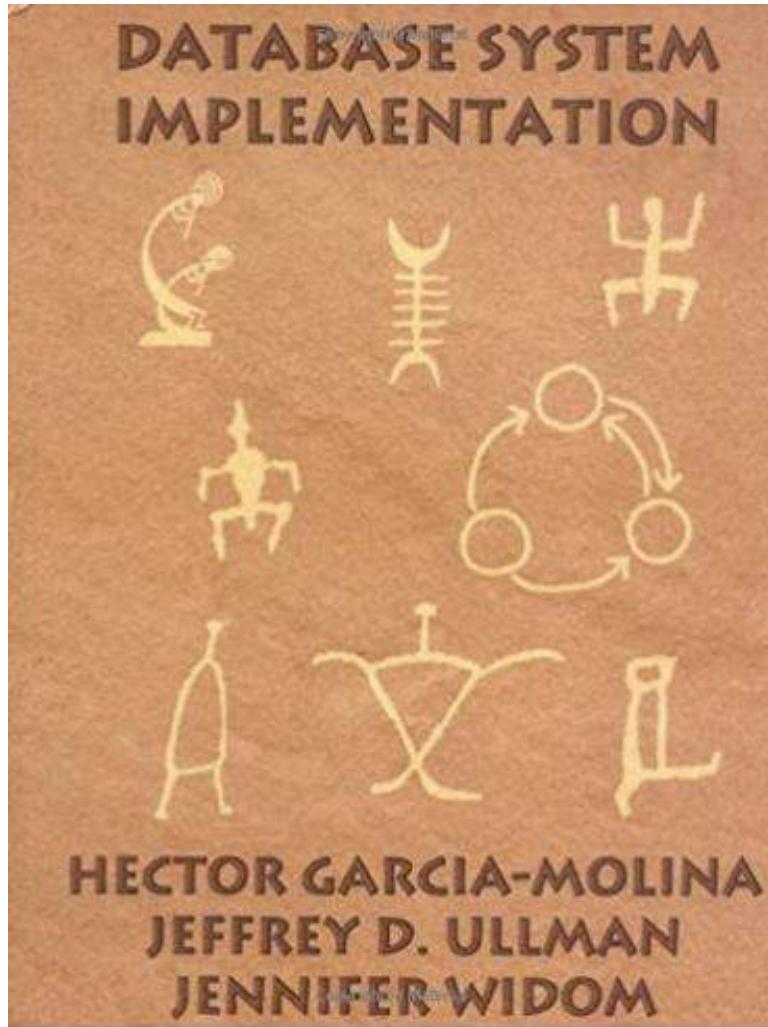
About the class

Reference Book - 1



Database Management Systems, Raghu Ramakrishnan, McGraw-Hill Education

Reference Book - 2

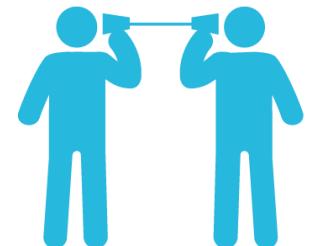


Database System Implementation, Hector
Garcia-Molina, Jeffrey D. Ullman, Jennifer
Widom

Communication

The course web page will be updated regularly throughout the semester with lecture notes, announcements and important dates.

<http://web.cs.hacettepe.edu.tr/~bbm371>



Course Work and Grading

- ▶ **2 midterm exams (50 points)**

- ⌚ Closed book and notes

- ▶ **Final exam (50 points)**

- ▶ Closed book and notes



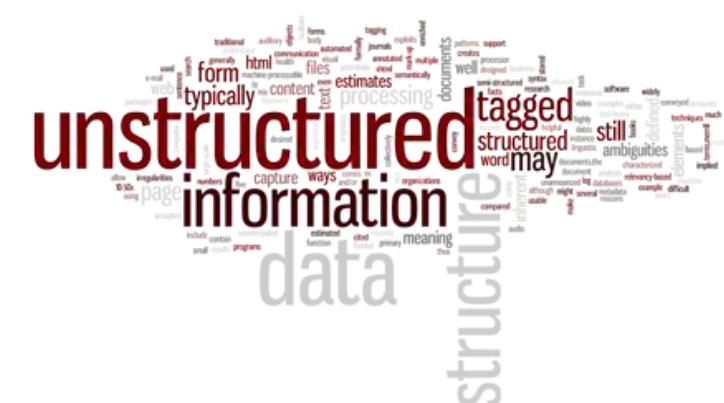
Course Overview (Tentative)

Date	Topic
10/10/2019	Course Policies, Introduction to Data Management
17/10/2019	Storage Devices
24/10/2019	Basic File Types
31/10/2019	Index Files
07/11/2019	Indexing & Cost
14/11/2019	Hash Tables
21/11/2019	Midterm I
28/11/2019	Hash-based Indexing
05/12/2019	Tree-based Indexing
12/12/2019	Tree-based Indexing (continued)
19/12/2019	External Sorting
26/12/2019	Midterm II
02/01/2020	External Sorting (Multi Disk - Multi Core)
09/01/2020	Spatial Data Management

Introduction to Database Management Systems

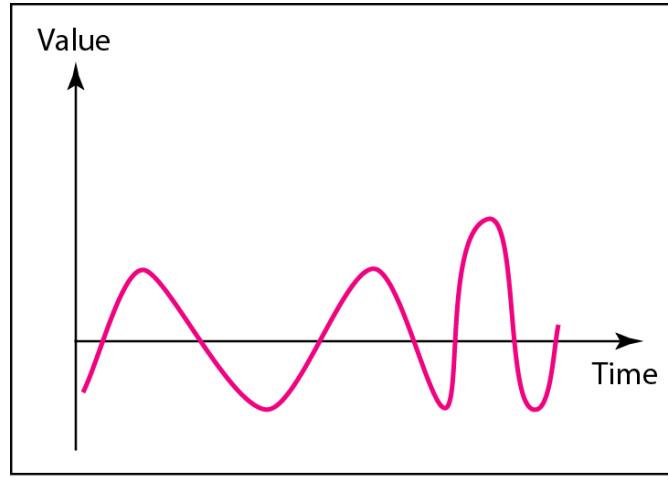
What is Data?

- **Data:** Almost any kind of unorganized fact(s).
 - **Examples:**
 - You throw a dice for a million times. Results are your data.
 - Anything you see in this classroom.
 - Music on a CD.
 - A computer file.

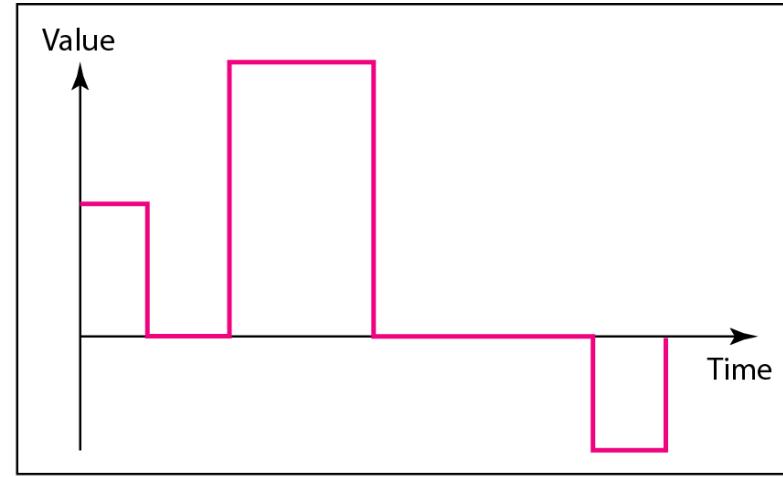


What is Signal?

- ▶ Signal is the encoding of the data that is needed for transmission.
- ▶ Analog
- ▶ Digital



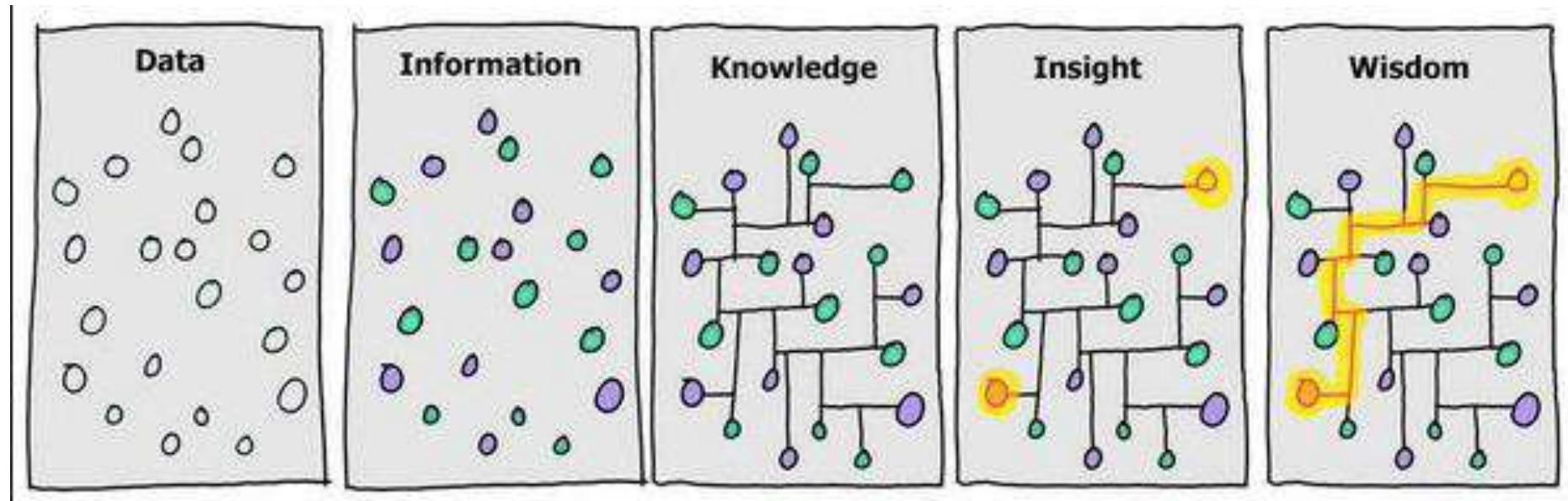
a. Analog signal



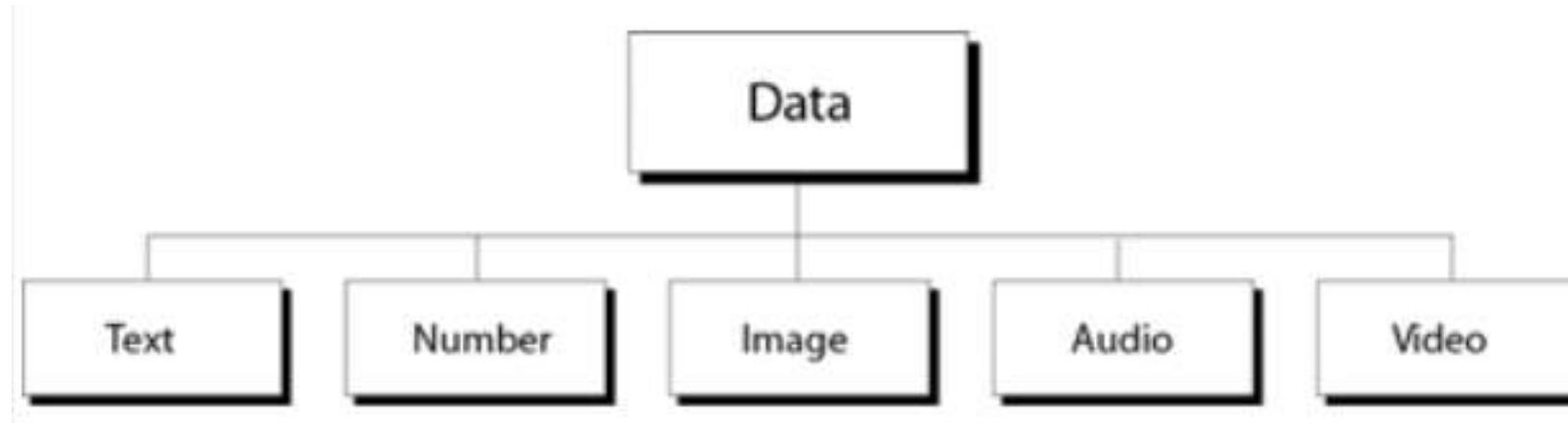
b. Digital signal

What is Information?

- Data becomes information when it is processed and organized and thereby it becomes useful.

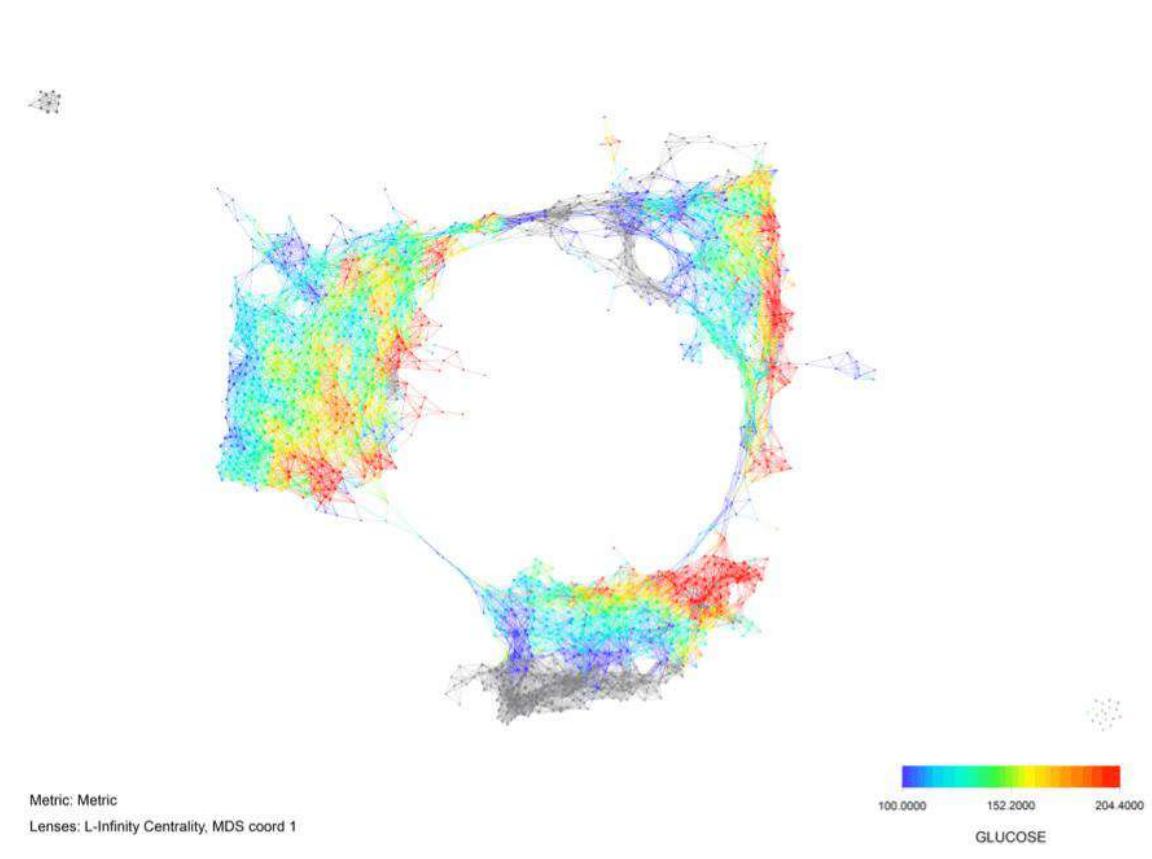


How to represent Data?



How to represent Complex Data?

- Relational
- Graph,
- Structured etc.



This data set, which delivered breakthrough insights into Type II diabetes is colored by glucose level. Those segments in red are distinct types of Type II diabetes.

What is Management?

Management: The process of dealing with things (or people)!

- ▶ Initiation/Setting Objectives
- ▶ Planning
- ▶ Design and Implementation
- ▶ Execution
- ▶ Monitoring and Control

Finally – What is Data Management? in this class...

- ▶ We will be interested in the following two concepts of data management:
 - ▶ Storage
 - ▶ Query Processing

What is a DBMS?

- ▶ A very large, integrated collection of data.
- ▶ Models real-world enterprise
- ▶ A Database Management System (DBMS) is a software package designed to store and manage databases
- ▶ Information about:
 - ▶ Entities: such as students, faculty, courses
 - ▶ Relationships: between entities for example a student is enrolled to a course



Data-Centric Applications

- ▶ Applications in which data plays an important role
 - ▶ Airline reservation systems
 - ▶ Data: aircrafts, flights, flight attendants, passengers, etc.
 - ▶ Banking applications
 - ▶ Data: clients, deposits, withdraws, etc.
 - ▶ Hospital systems
 - ▶ Data: patients, physicians, diagnosis, prescriptions, etc.
 - ▶ University systems
 - ▶ Data: students, teaching staff, courses, enrollments, etc.

History of DBMS

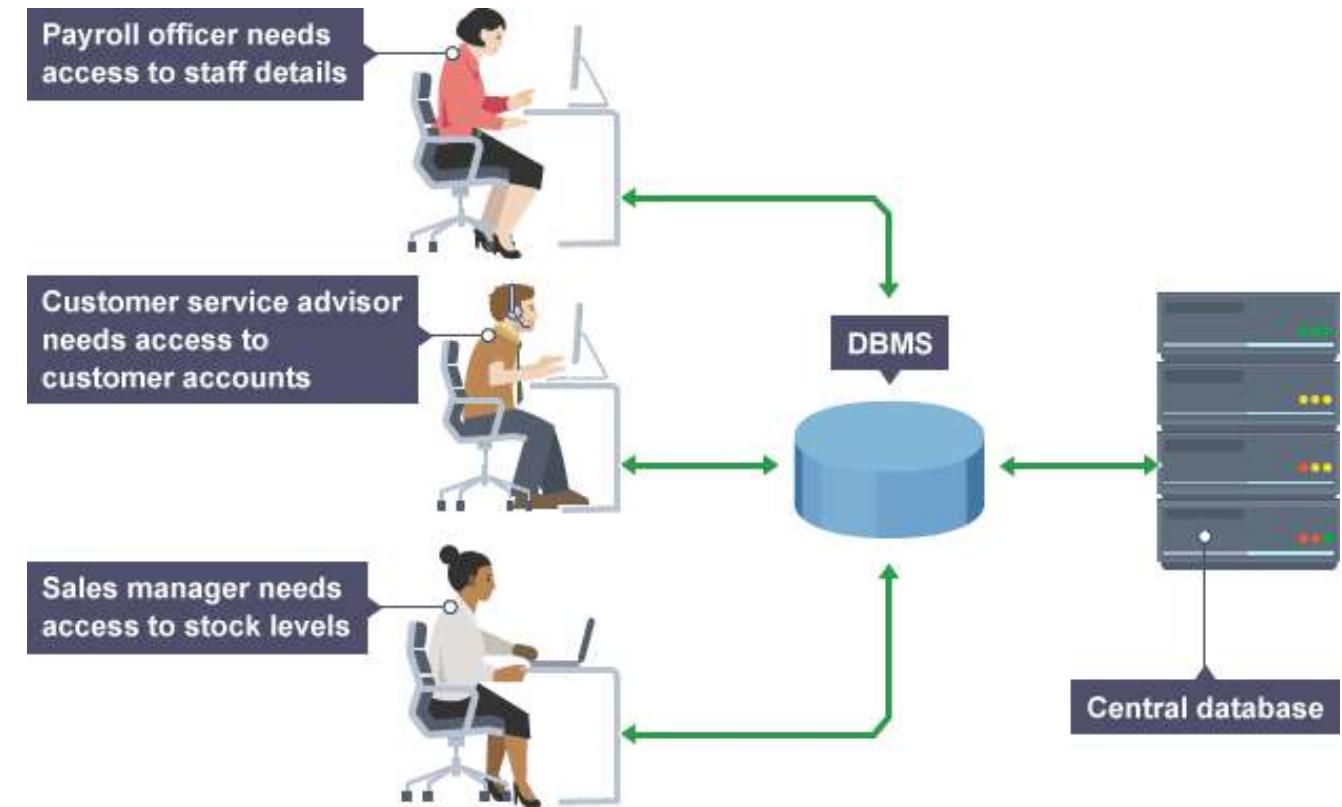
- ▶ Even from the early days of computers, data must be stored for applications
- ▶ Late 1960 IBM's Information Management System (IMS) for airline reservations.
- ▶ 1970s Edgar Codd proposed a relational data model
- ▶ 1980s database query language SQL was standardized
- ▶ 1990s Data warehouses, consolidating data from multiple data stores for analysis
- ▶ 2000s Web applications
- ▶ Now – Even larger volumes of data NoSQL databases

Files vs. DBMS

- ▶ Imagine writing a program for a bank
 - ▶ Customers, Accounts, Money Transfers
 - ▶ More than 500 GB (does not fit in memory)
- ▶ Application must stage large datasets between main memory and secondary storage (500GB RAM is not still cheap!)
- ▶ Must protect data from inconsistency (update in ATM should be consistent with bank branch)
- ▶ Crash recovery
- ▶ Security and access control
- ▶ Concurrency (Transaction management)

Why Use a DBMS?

- ▶ Data independence and efficient access
- ▶ Reduced application and development time
- ▶ Data integrity and security
- ▶ Uniform data administration
- ▶ Concurrent access
- ▶ Recovery from crashes



Example of a Traditional Database Application

Suppose we are building a system to store the information about:

- ▶ students
- ▶ courses
- ▶ professors
- ▶ who takes what, who teaches what

Can we do it without a DBMS ?

Sure we can! Start by storing the data in files:

students.txt

courses.txt

professors.txt



Now write C or Java programs to implement specific tasks

Doing it without a DBMS...

- ▶ Enroll “Mary Johnson” in “CSE444”:

Write a C/Java program to do the following:

Read ‘students.txt’

Read ‘courses.txt’

Find&update the record “Mary Johnson”

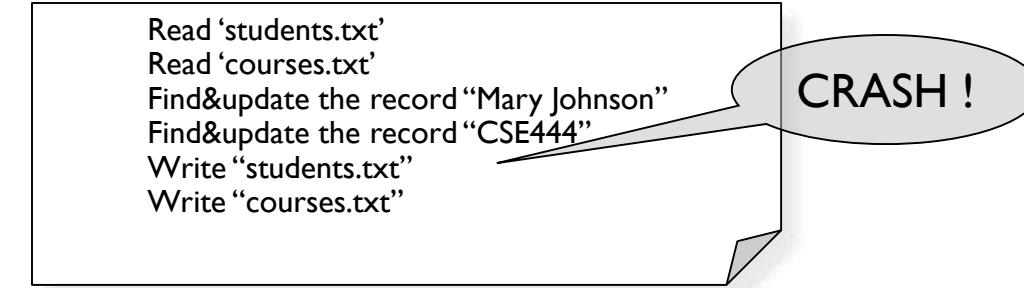
Find&update the record “CSE444”

Write “students.txt”

Write “courses.txt”

Problems without an DBMS...

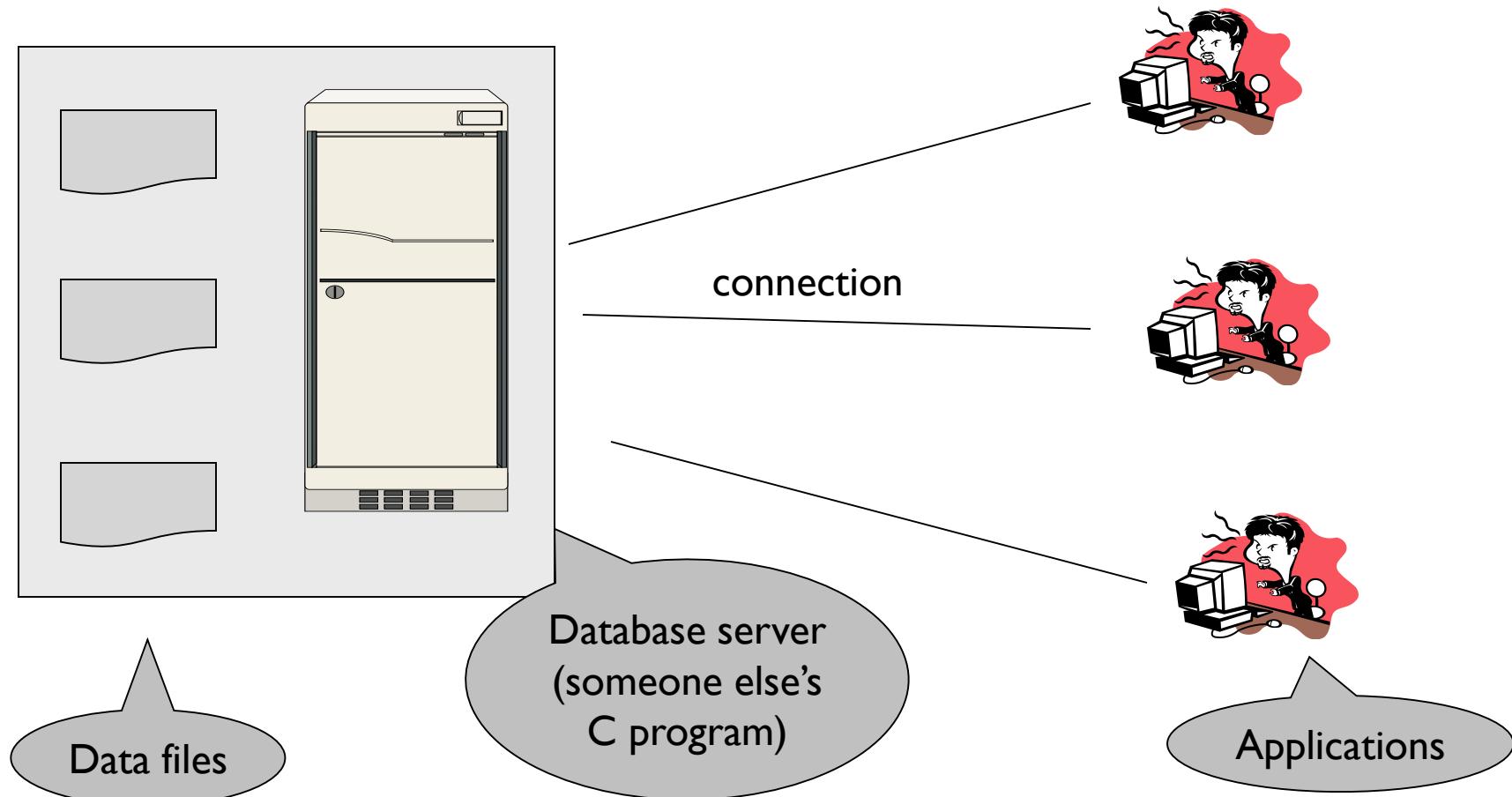
- ▶ System crashes:



- ▶ What is the problem ?
- ▶ Large data sets (say 50GB)
 - ▶ Why is this a problem ?
- ▶ Simultaneous access by many users
 - ▶ Lock students.txt – what is the problem ?

DBMS

“Client-server”



Why Study Databases?

- ▶ Shift from computation to information
 - ▶ Low-end users: Web Applications needs to organize information (a mess will not be effective)
 - ▶ High-end users: Scientific applications now have data management problems!
- ▶ Datasets increasing in diversity and volume
 - ▶ Digital libraries, interactive video, Human Genome project etc.
- ▶ DBMS encompasses most of CS
 - ▶ OS, languages, AI, multimedia etc.

Data Models

- ▶ A **data model** is a collection of concepts for describing data. (high-level)
- ▶ A **schema** is a description of a particular collection of data, using the given data model
- ▶ The **relational model of data** is the most widely used model today.
 - ▶ Main concept: **relation**, basically a table with rows and columns
 - ▶ Every relation has a **schema**, which describes the columns, or fields.
 - ▶ Schema is defined by: name of schema, the name of each **field** (or **attribute** or **column**) and type of each field
 - e.g.
 - Students(sid: string, name: string, login: string, age: integer, gpa:real)**

Entity: Student

- **Students(sid: string, name: string, login: string, age: integer, gpa: real)**

Sid	name	login	age	gpa
53666	Jones	<u>jones@cs</u>	18	3.4
53688	Smith	<u>smith@ee</u>	18	3.2
53650	Smith	<u>smith@math</u>	19	3.8

**Attribute
(field or column)**

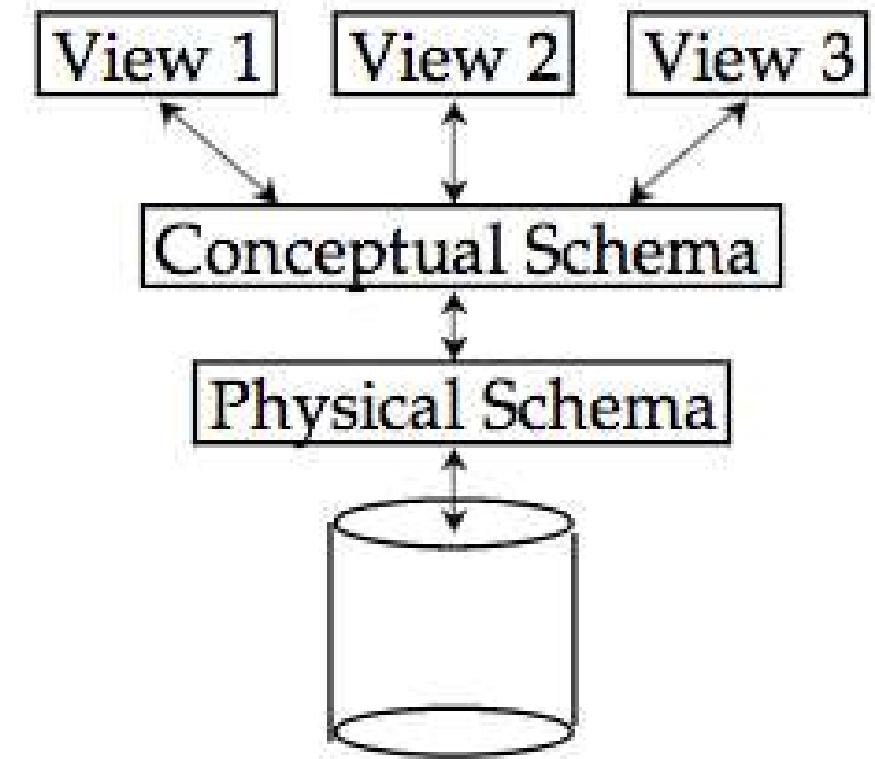
Record

Using age as a field is not a good idea, why?

Integrity Constraints: We can define the field sid to be unique or age to be larger than 0. Rules for records to satisfy

Levels of Abstraction

- ▶ Unlike programmers of early systems, programmer of relational system does not need to implement lower level details
- ▶ Many views, single conceptual (logical) schema and physical schema.
 - ▶ Views describe how users see the data.
 - ▶ Conceptual schema defines logical structure
 - ▶ Physical schema describes the files and indexes used



External Level

deptN	dChai	sNam	sPos
356	2	HS	Prof
....

Conceptual Layer

Dept-Table

deptno	dNam	dAdr	dChair
356	BilM	mmm	2
357	EleM	ggg	4

Staff-Table

sld	sNam	sBranc	sPos
1	Ebru	Inf.Ret	Assoc
2	HS	Inf.Ret	Prof

relation

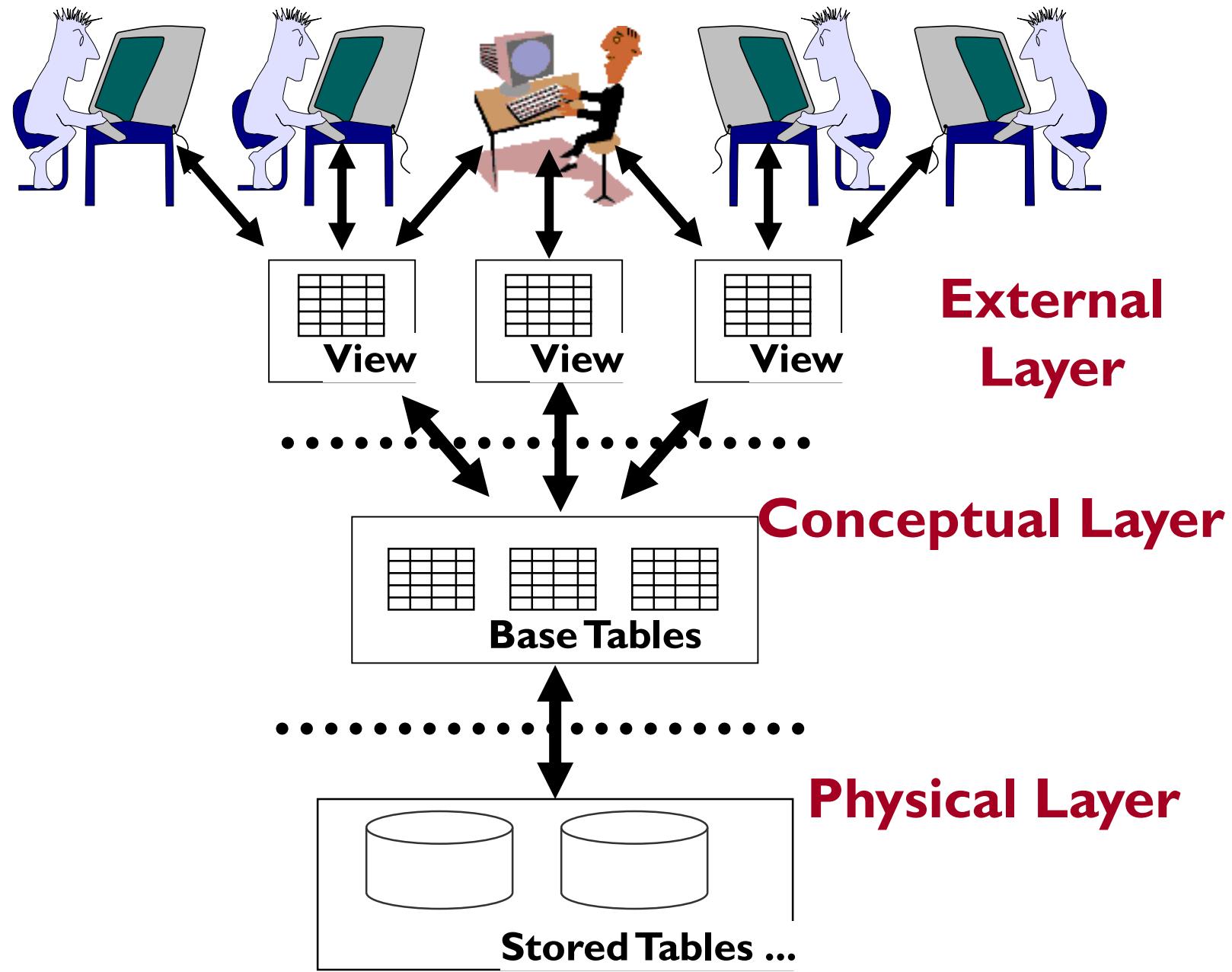
Physical Layer

Dept-File

356BilMmmm2357EleMggg4

Staff-File

1EbruInf.Ret.Assoc2HSInf.Ret.Prof

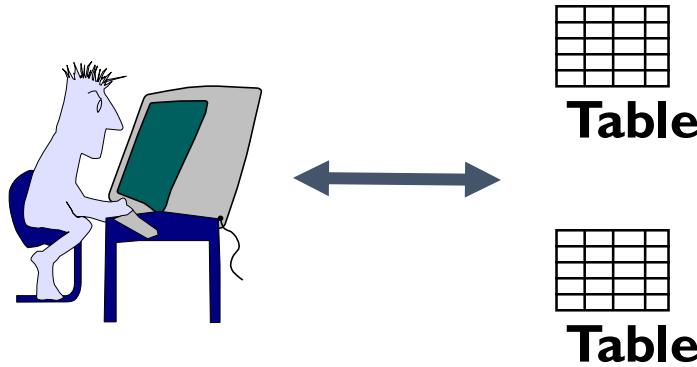


Physical View

- **The DBMS must know**
 - **exact physical location**
 - **precise physical structure**



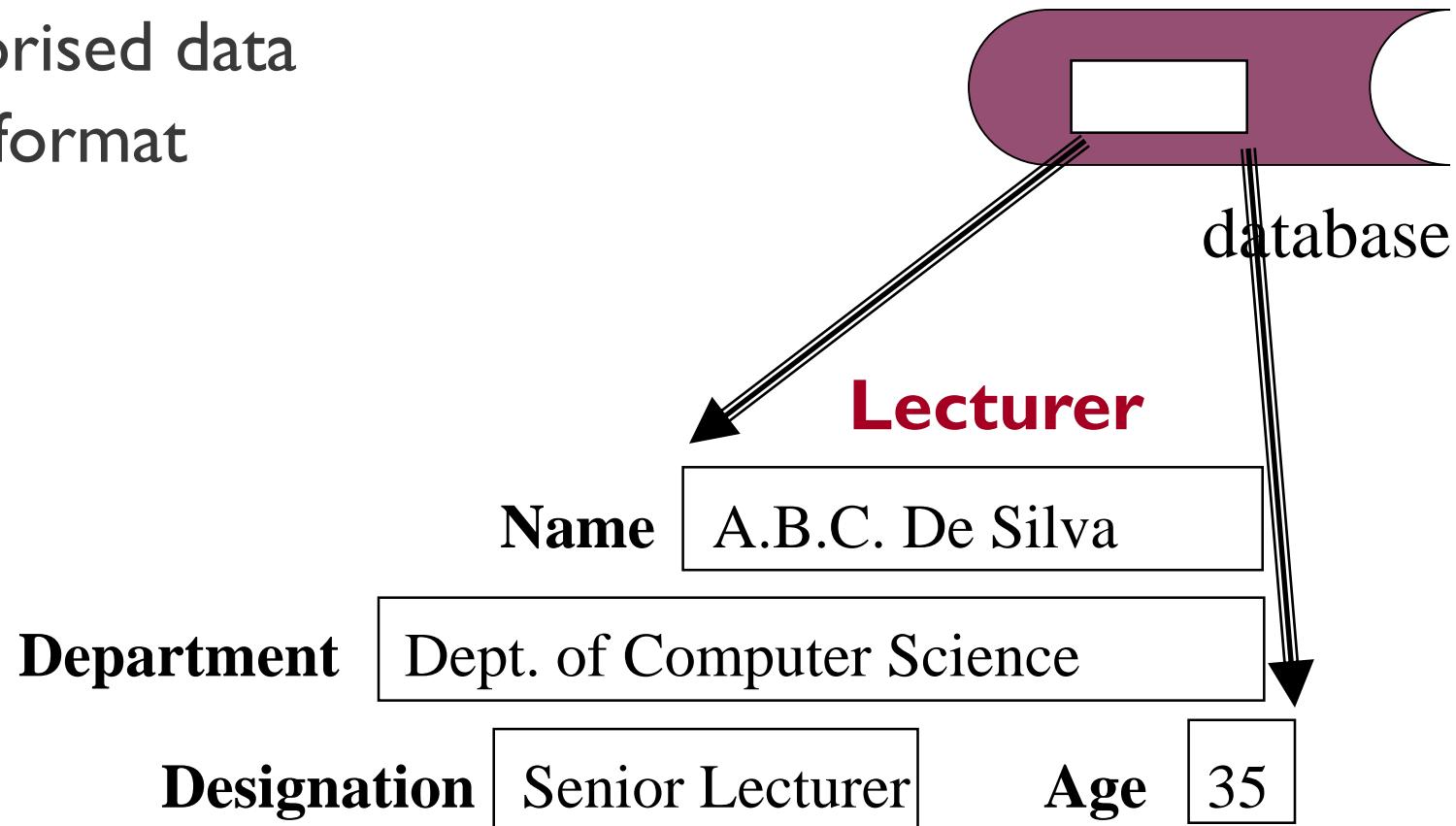
Conceptual Layer



- ▶ The conceptual model is a logical representation of the entire contents of the database.
- ▶ The conceptual model is made up of base tables.
- ▶ Base tables are “real” in that they contain physical records.

External View

- The user/application see
 - authorised data
 - own format



External View cont.

- ▶ External views allow to
 - ▶ hide unauthorised data
 - ▶ e.g. salary, dob
 - ▶ provide user view
 - ▶ e.g. view employee name, designation, department data taken from employee and department files
 - ▶ derive new attributes
 - ▶ e.g. age derived from dob

Example: University Database

- ▶ Conceptual schema:
 - ▶ Students(sid:string, name:string, login:string, age:integer, gpa:real)
 - ▶ Courses(cid:string, cname:string, credits:integer)
 - ▶ Enrolled(sid:string, cid:string, grade:string)
- ▶ Physical schema:
 - ▶ Relations stored as unordered files
 - ▶ Index on first column of Students
- ▶ External Schema (View):
 - ▶ Course_info(cid:string,enrollment:integer)

Data Independence

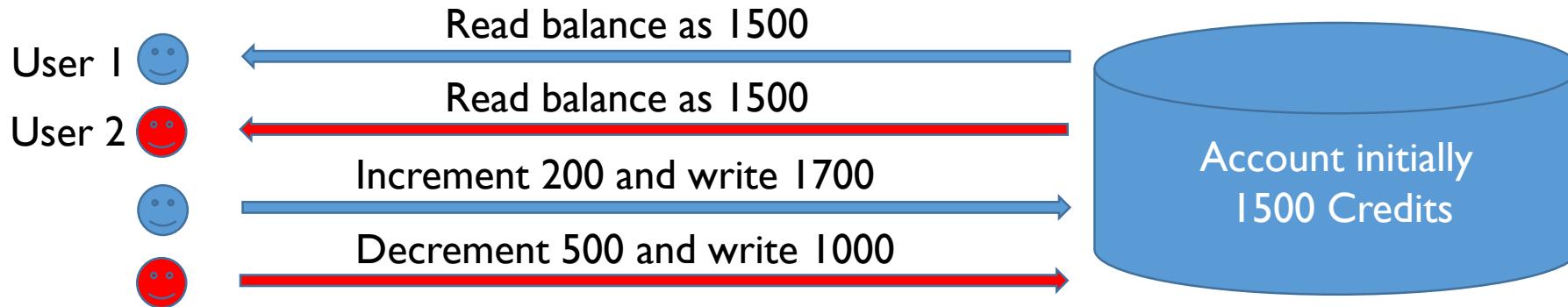
- ▶ Applications insulated from how data is structured and stored.
- ▶ **Logical data independence:** Protection from changes in logical structure of data.
- ▶ **Physical data independence:** Protection from changes in physical structure of data.

* *One of the most important benefits of using a DBMS!*

Concurrency Control

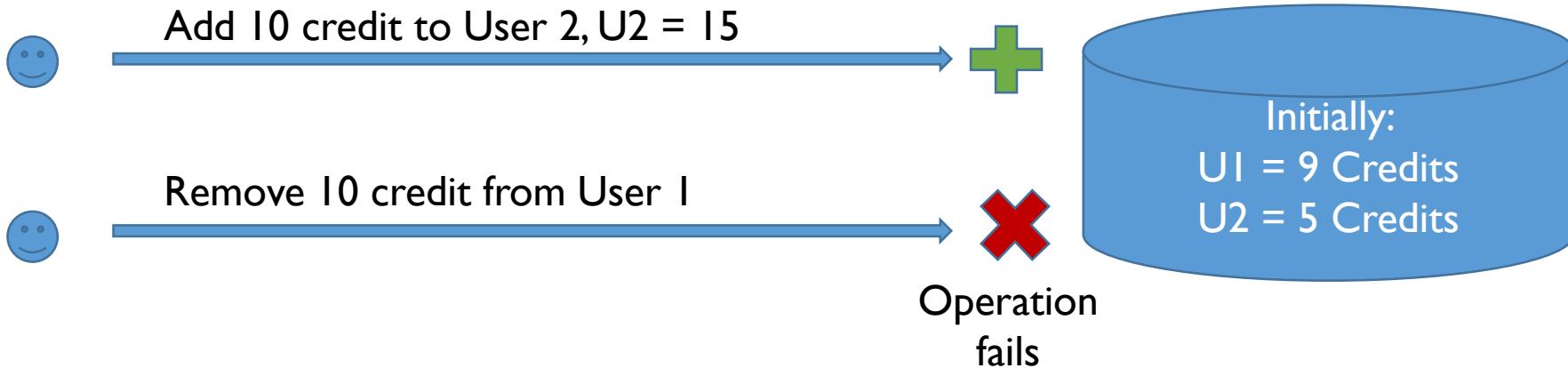
- ▶ Concurrent execution of user programs is essential for good DBMS performance
 - ▶ Because disk accesses are frequent and relatively slow, it is important to keep the CPU humming by working on several user programs concurrently.
- ▶ Interleaving actions of different user programs can lead to inconsistency
- ▶ DBMS ensures such problems don't arise.
- ▶ Users can pretend they are using a single-user system.

Transaction Example 1



- ▶ Two users performing operations on a joint account at the same time.
- ▶ If one reads before the other writes back, the first to write will be cancelled
- ▶ It will work ok if read and insert is atomic (not interrupted)
- ▶ To make sure, we can lock the account

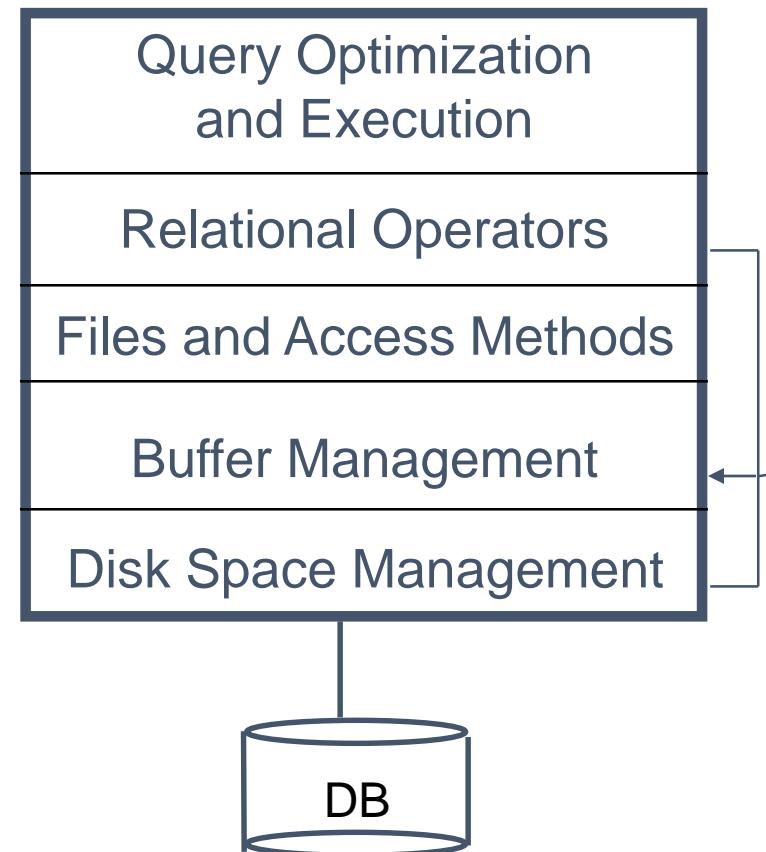
Transaction Example 2



- ▶ A prepaid mobile phone user will transfer 10 credits to User 2.
- ▶ This operation needs two steps
- ▶ If trying to remove 10 credits from User 1 fails for some reason, we have added 10 credits to U2 out of the blue
- ▶ If we perform the operation in a transaction, we can roll-back the changes.

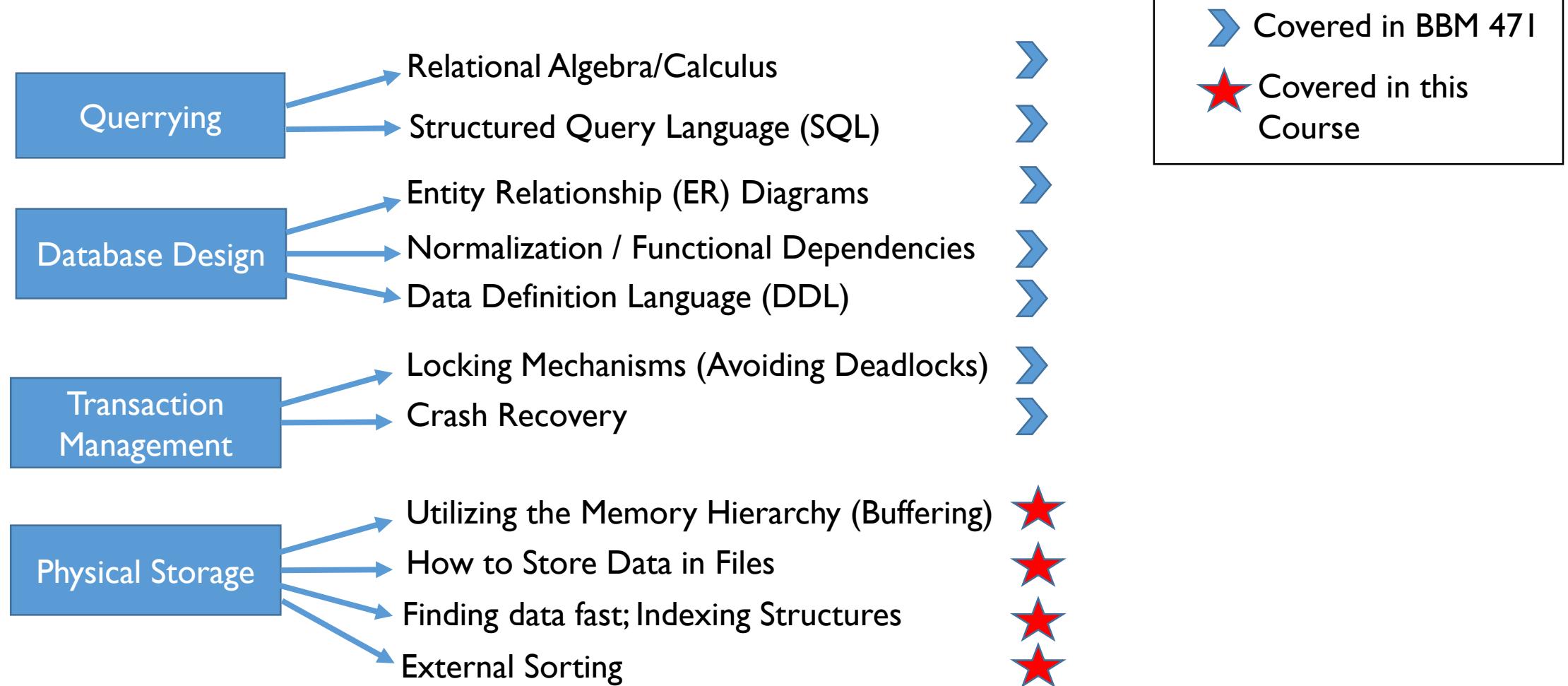
Structure of a DBMS

- ▶ A typical DBMS has a layered architecture.
- ▶ The figure does not show the concurrency control and recovery components.
- ▶ This is one of several possible architectures; each system has its own variations.



These layers must consider concurrency control and recovery

An overview of Database Concepts



Databases make these folks happy...

- ▶ End users and DBMS vendors
- ▶ DP application programmers
 - ▶ E.g. smart webmasters
- ▶ Database administrator
 - ▶ Design logical / physical schemas
 - ▶ Handles security and authorization
 - ▶ Data availability, crash recovery



End of the first lecture...

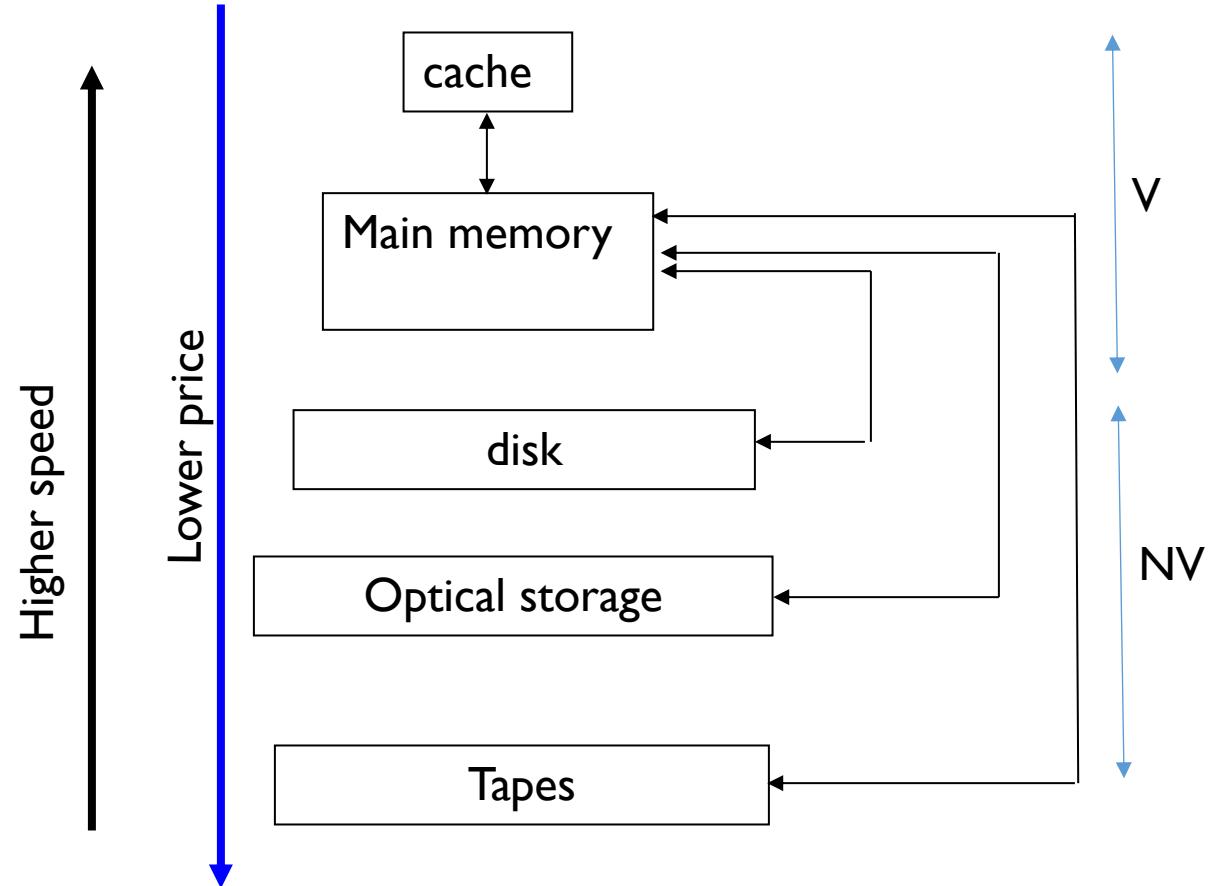


BBM371- Data Management

Lecture 2: Storage Devices

17.10.2019

Memory Hierarchy



Traveling the hierarchy:

1. speed (higher=faster)
2. cost (lower=cheaper)
3. volatility (between MM and Disk)
4. Data transfer (Main memory the “hub”)
5. Storage classes (P=primary, S=secondary, T=tertiary)

Storage Devices

► Direct Access Storage Devices

Hard disk: Huge vol., fast, expensive

USB, CD, DVD: Transportable, less vol., slow, cheaper



**Hard Disk -
random access storage**

► Serial Access Storage Devices

Tapes: Largest Size, Cheaper, Transportable, Slow



**Tape Drive –
sequential storage**



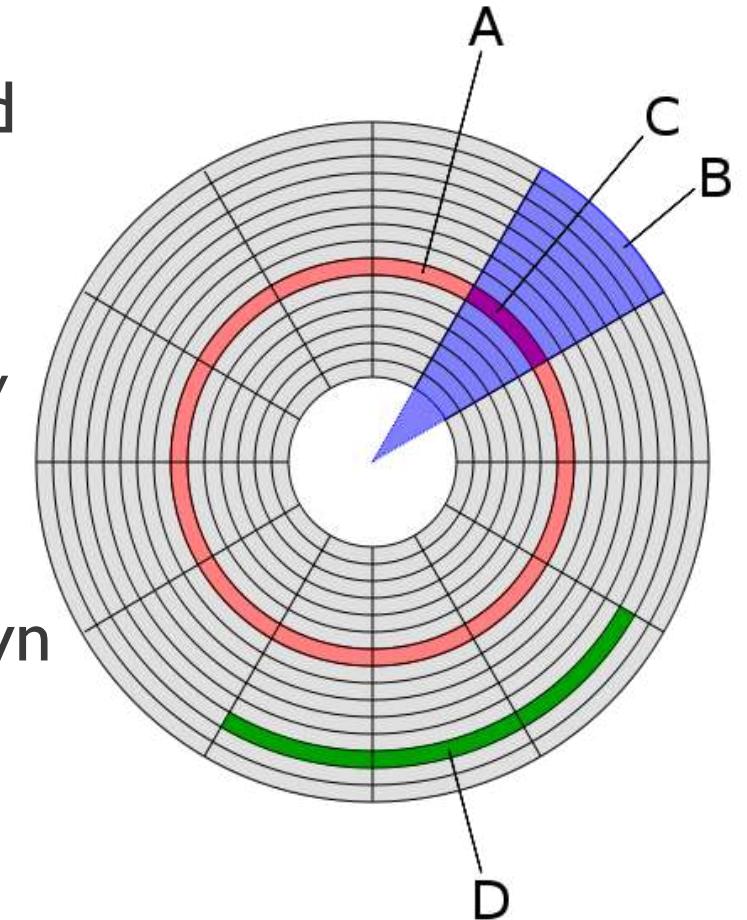
CD ROM / DVD



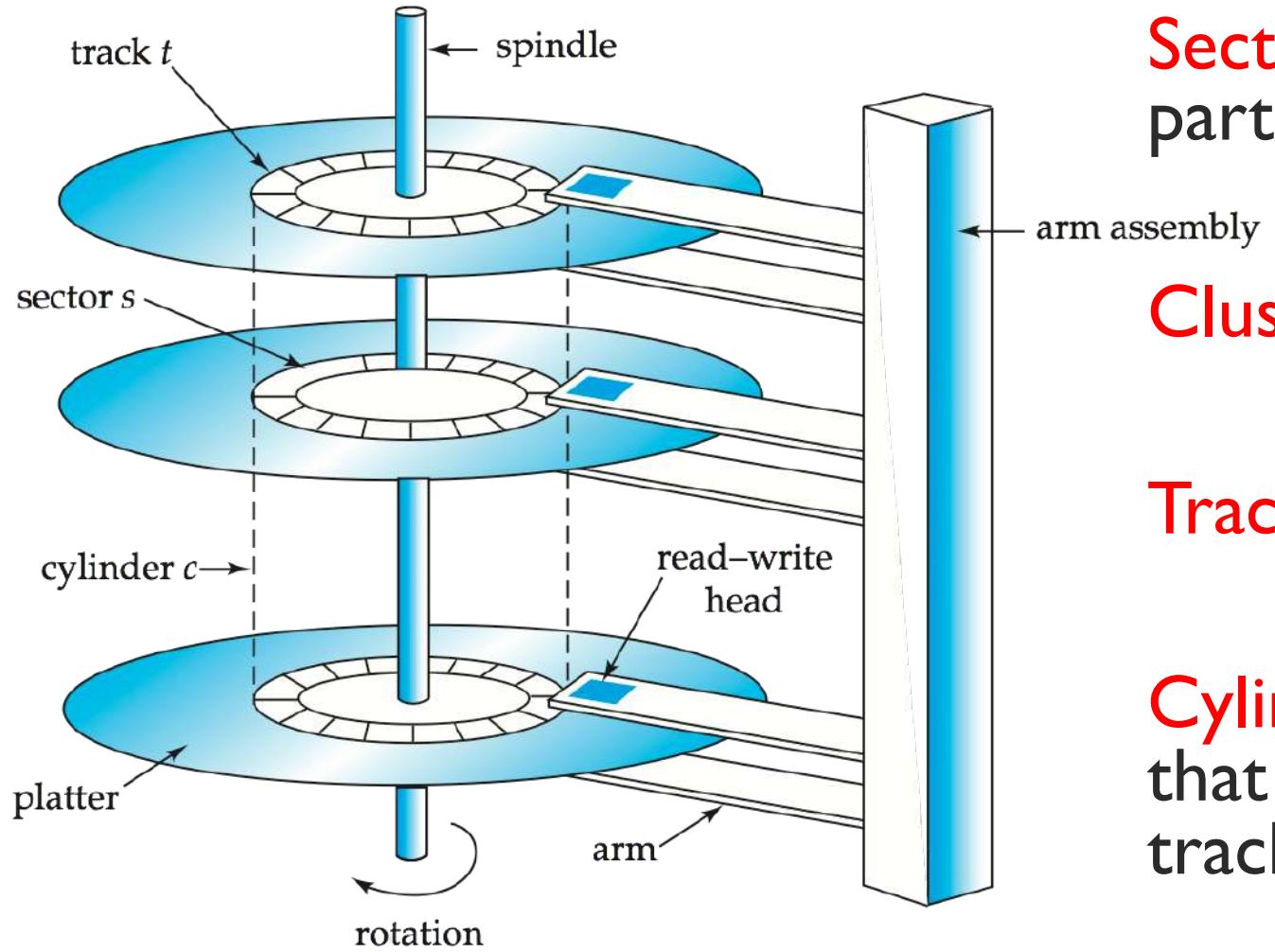
**Floppy Disk Drive –
random-access storage**

Organization of Disk

- ▶ Platters : May use both sides for storing data
- ▶ Track : A circular path where the data is stored magnetically (Shown as A)
- ▶ Sector : A subdivision of track. (C)
 - ▶ Normally sector means portion of disk enclosed by two radii and arc. (B)
- ▶ Cluster : Unit of data transferred, defined in terms of multiple sectors (e.g. 3 sectors) (Shown as D)
- ▶ This is usually a file system related setting



From Physical to Conceptual



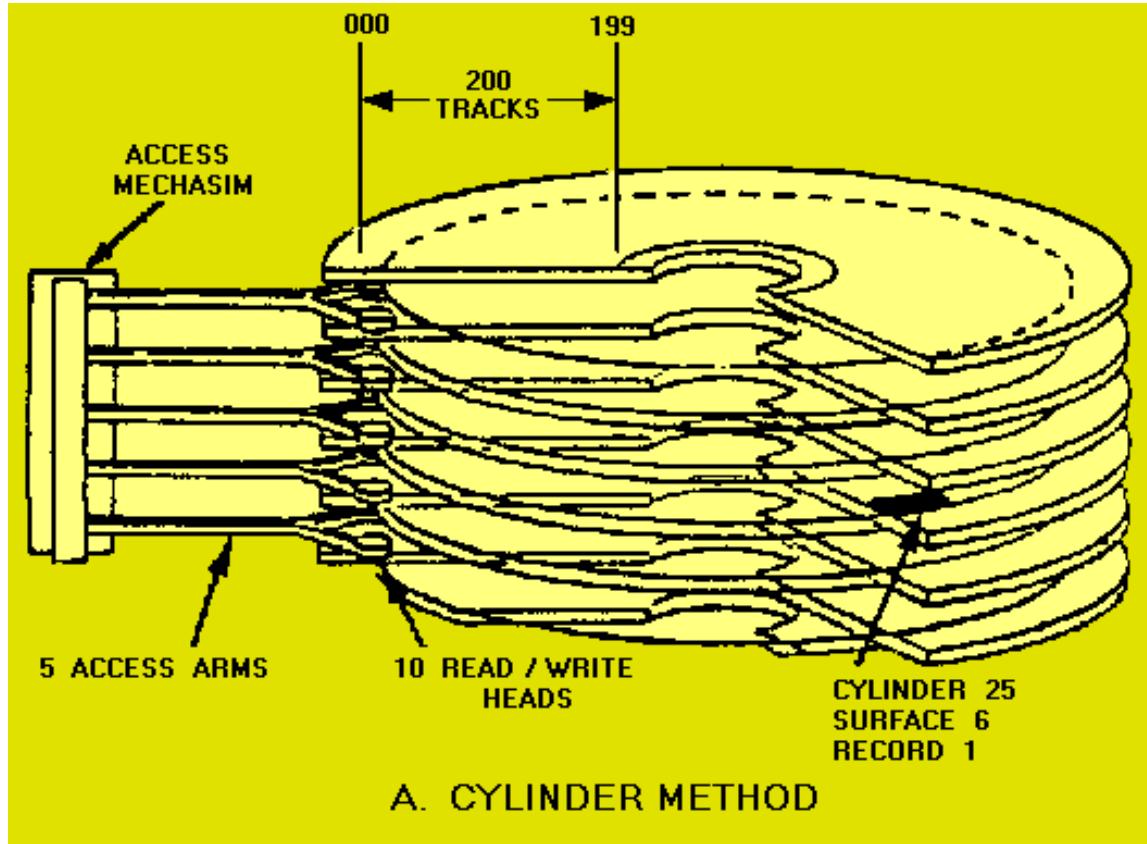
Sector: Minimum track partition for data storage

Cluster: Group of sectors

Track: One cycle on the platter

Cylinder: A semantic shape that consists of same level tracks on the platters

From Physical to Conceptual



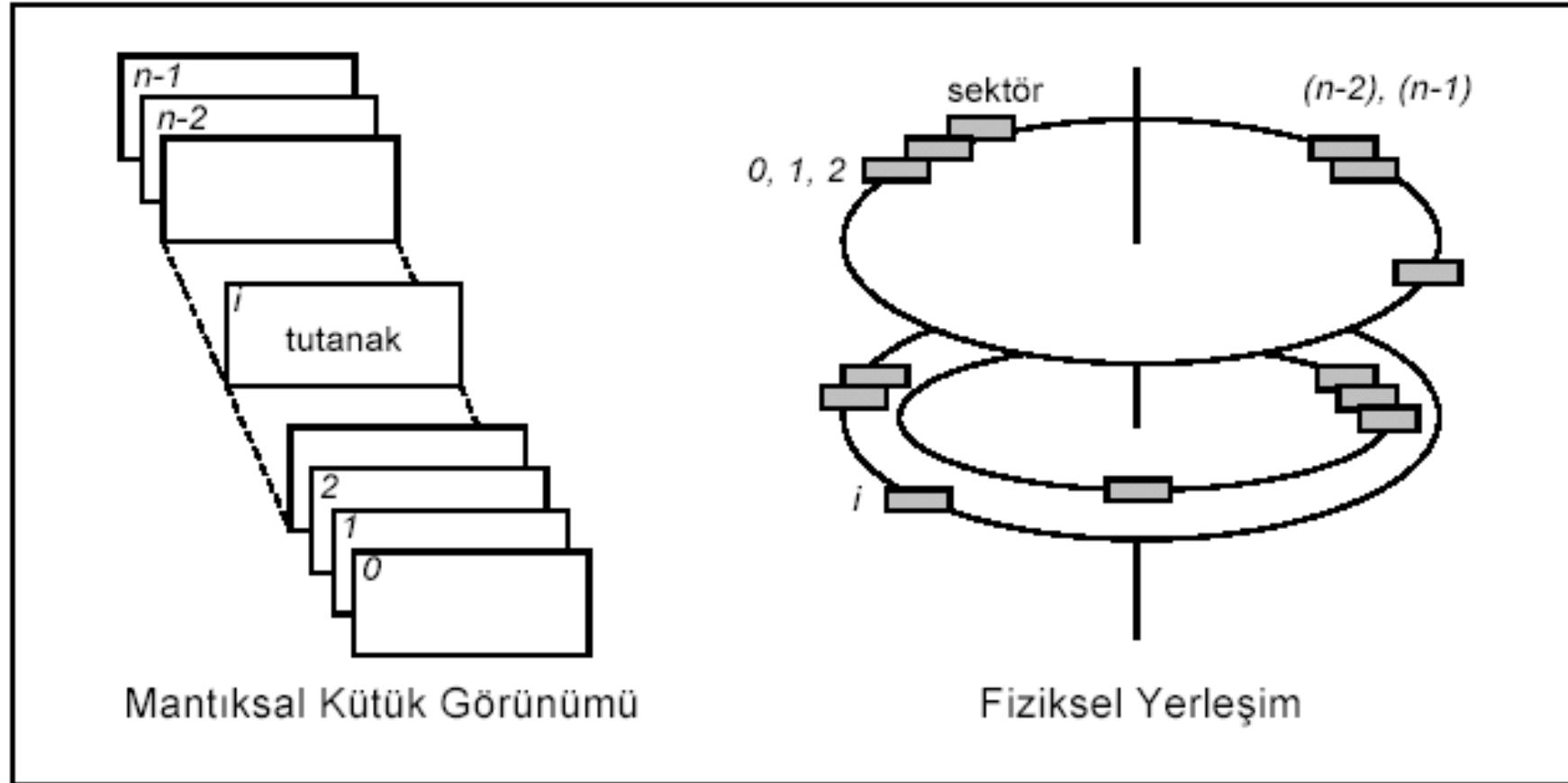
Sector: Minimum track partition for data storage

Cluster: Group of sectors

Track: One cycle on the platter

Cylinder: A semantic shape that consists of same level tracks on the platters

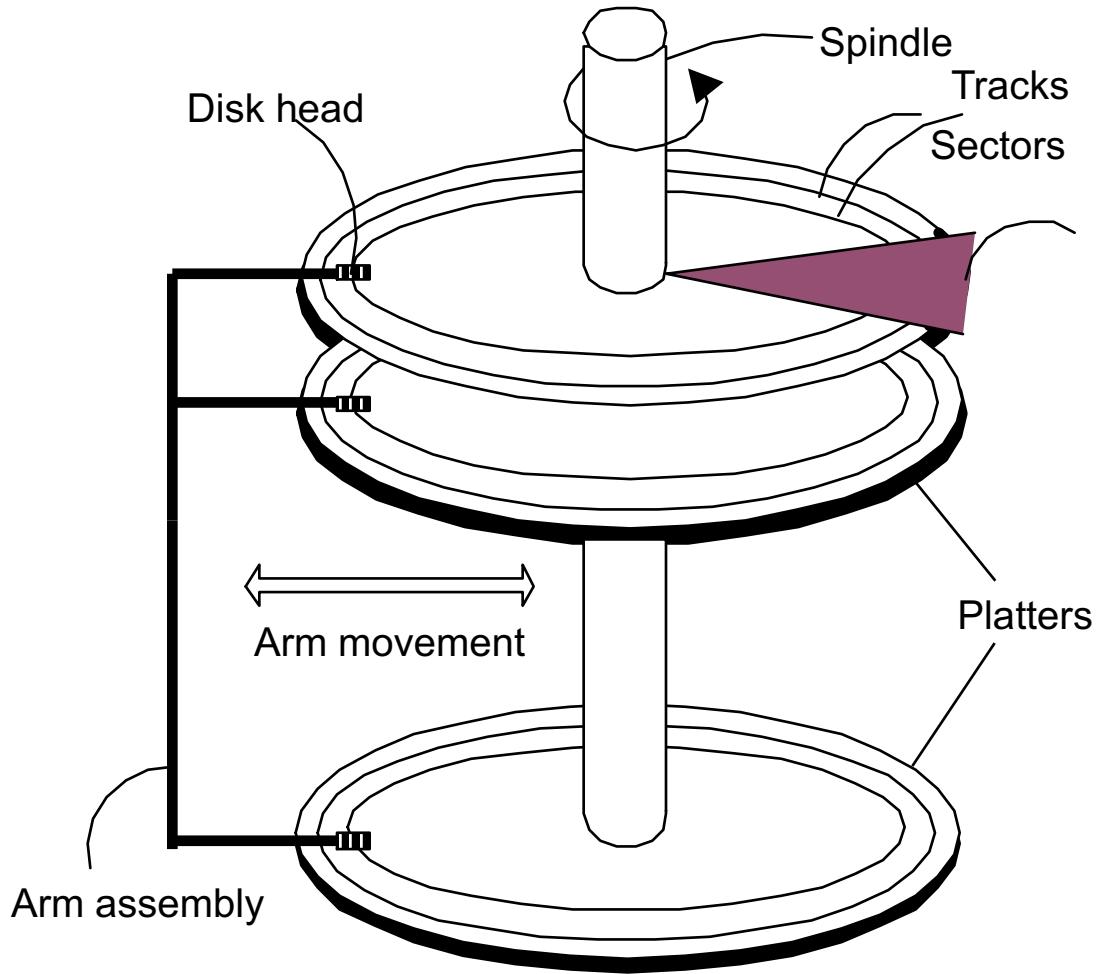
Relationship between Logical & Physical Models



Çizim 6.1. Kavramsal Kütük Modeli ve Fiziksel Yerleşimi

Components of a Disk

- ▶ The platters spin, to position for the desired **sector**
- ▶ The arm assembly is moved inwards or outwards to position the head on a desired **track**
- ▶ **Only one** head reads/writes at any one time
- ▶ **Block size** is a multiple of **sector size** (which is fixed)



Blocking

- ▶ Physical Block
 - ▶ Minimum data transferred between primary and secondary memory
- ▶ Logical Block
 - ▶ Minimum set of records transferred between primary and secondary memory
- ▶ Blocking Factor
 - ▶ The number of logical records per logical block
- ▶ Assume, one cluster=one block

Blocking (cont.)

- ▶ Entire physical blocks are moved from secondary to the continuous section of primary memory called as **buffer**
- ▶ A request for a disk block already in buffer does not require a disk transfer
- ▶ Speeding-up database operations depends on arranging data so that when a record in a block is needed, rest of the records on the same block will also be needed soon.
- ▶ Choosing too big block size will transfer too much unnecessary records, cluttering the buffer
- ▶ Choosing too small block size will guarantee a disk operation per each record request

Choosing Block Factor

- ▶ Not too large not too few
- ▶ Affected by
 - ▶ Storage media
 - ▶ Operating system
 - ▶ Record length
 - ▶ Fixed
 - ▶ Variable

Logical READ

Reading a record:

- ▶ The physical block(s) are transferred to the buffer (in primary memory)
 - ▶ Only perform physical disk transfer if not already in the buffer.
 - ▶ If it is in buffer this step is skipped (cache hit!)
- ▶ The record is extracted from the block
- ▶ Data in the record is transferred to variables of the program

Can you specify how a logical write is performed?

Access

- ▶ Access path : used to find a target record
 - ▶ Search mechanism
 - ▶ File organization
 - ▶ Secondary storage media
- ▶ Length of an access path:
 - ▶ The number of physical blocks accessed in the process of accessing a record

File design considerations

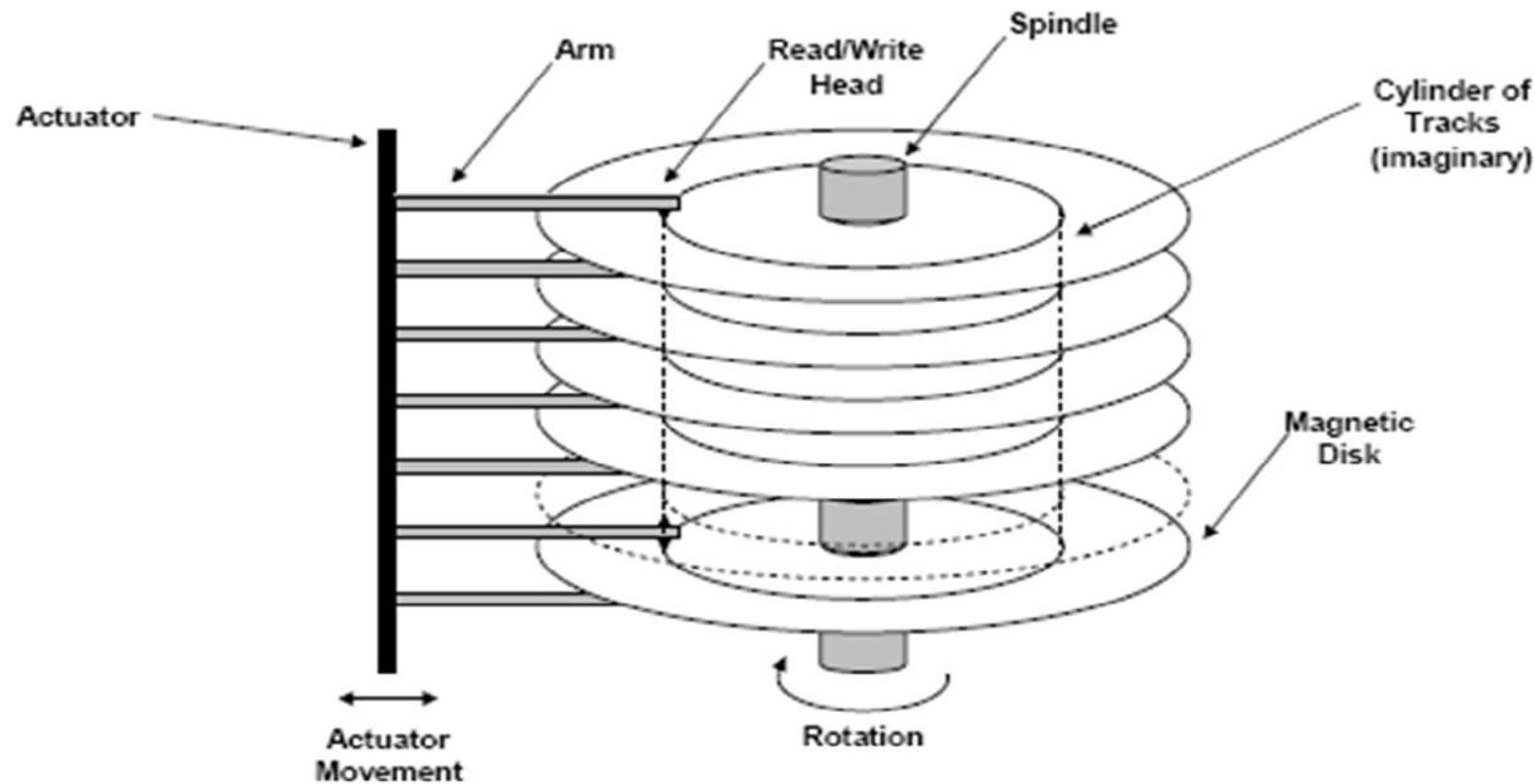
- ▶ Selection of blocking factor
- ▶ Organization of physical blocks in secondary storage
- ▶ Design of access method
- ▶ Select primary key
- ▶ File Growth

Types of Operations

- ▶ Retrieve_All
- ▶ Retrieve_One
- ▶ Retrieve_Next
- ▶ Retrieve_Previous
- ▶ Insert_One
- ▶ Update_One
- ▶ Retrieve_Few

What is the significance of cylinders?

- All information under a cylinder can be accessed without moving the arm that holds read-write heads.



Estimating Capacity of Disk

- The total capacity of disk is a function of cylinder, track & sector:

Track capacity = num of sectors per track * bytes per sector

Cylinder Capacity = num of tracks per cylinder * track capacity

Driver capacity = Number of Cylinder * Cylinder Capacity

Example

Question: Consider a disk with sector size=512 Bytes
2000 tracks per surface (of a platter), with 5 double sided platters
50 sectors per track.What is the capacity of disk?

Example

Question: Consider a disk with sector size=512 Bytes
2000 tracks per surface (of a platter), with 5 double sided platters
50 sectors per track. What is the capacity of disk?

Capacity of track: $50 \times 512 = 25600$

Capacity of a surface: $2000 \times 25600 = 51200000$

Capacity of disk: $51200000 \times 5 \times 2$

* How many cylinders do we have?

* Can we use 256 bytes, 2048, 51200 as block size?

- ▶ Too small, we can, Too big (for track)

Performance Measures of Disks

Measuring Disk Speed:

- ▶ Access time :
 - ▶ Seek time: time to move the arm over the correct track
 - ▶ ~5ms from track-to-track,
 - ▶ Rotational latency: time it takes to align the head with the sector to be accessed.
 - ▶ For average you can use half of the time it takes to complete a full-rotation (e.g. 8.3ms)
 - ▶ Data transfer rate: The time required to retrieve the data under the read/write head.

Analogy to taking a bus:

1. Seek time: time to get to bus stop
2. Latency time; time spent waiting at bus stop
3. Data transfer time: time spent riding the bus

Cost of Access

Access Time = Seek Time + Rotational Delay

Time to Read = Seek Time + Rotational Delay +
Transfer time

How can you calculate seek time, rotational delay and transfer time?

Answer

- ▶ Seek time: usually provided with the specs. of the disk
- ▶ Rotational delay: On average half of the time required for a complete revolution. Revolutions per minute (rpm) can be used to determine this
- ▶ Transfer time = (Num. of sectors transferred / Num. of sectors on a track) * rotation time

Example

ST3120022A : Barracuda 7200.7

Capacity : 120 GB

Interface : Ultra ATA/100

RPM : 7200 RPM

Seek time : 8.5 ms avg

Latency time? :

- ▶ $7200/60 = 120$ rotations/sec
- ▶ 1 rotation in 8.3 ms => So, Average Latency = 4.16 ms

Example 2

Disc specs:

- ▶ 20 plate, 800 track/plate, 25 sector/track, 512 byte/sector, 3600 rpm
- ▶ 7 ms from track to track, 28 ms avg. seek time, 50 msec max seek time

Avg. Rotational time =

Disk capacity =

Time to read track =

Time to read cylinder =

Time to read whole disc =

from one cylinder to another =

Example 2

Disc specs:

- ▶ 20 plate, 800 track/plate, 25 sector/track, 512 byte/sector, 3600 rpm
- ▶ 7 ms from track to track, 28 ms avg. seek time, 50 msec max seek time

Avg. Rotational time	$= 1/2 * (60000/3600) = 16.7/2 = 8.3\text{msec}$
Disk capacity	$= 25 * 512 * 800 * 20 = 204.8 \text{ MB}$
Time to read track	$= 1 \text{ rotation} = 16.7 \text{ msec}$
Time to read cylinder	$= 20 * 16.7 = 334\text{msec}$
Time to read whole disc from one cylinder to another	$= 800 * \text{time to read cylinder} + 799 * \text{time to pass from one cylinder to another}$ $= 800 * 334 + 799 * 7\text{msec} = 267 \text{ sec} + 5.59 \text{ sec} = 272.59 \text{ sec}$

Example 3

- ▶ Disc specs: avg. seek time 8 msec, 10.000 rpm, 170 sector/track, 512 byte/sector

Time to read one sector =

Time to read 10 sequential blocks =

Time to read random 10 block =

Example 3

- ▶ Disc specs: avg. seek time 8 msec, 10.000 rpm, 170 sector/track, 512 byte/sector

Time to read one sector = avg. seek time + rotational delay + time to transfer one sector

$$= 8 + (0.5 * 60.000 / 10.000) + (6 * 1/170) = 8 + 3 + 0.035 = 11.035 \text{ msec}$$

Time to read 10 sequential blocks = avg. seek time + rotational delay + 10 * time to transfer one sector = $8 + 3 + 10 * 0.035 = 11.35 \text{ msec}$

Time to read random 10 block = 10 * (avg. seek time + rotational delay + time to transfer one sector) = $10 * (8 + 3 + 0.035) = 113.5 \text{ msec}$

Your turn: Exercise

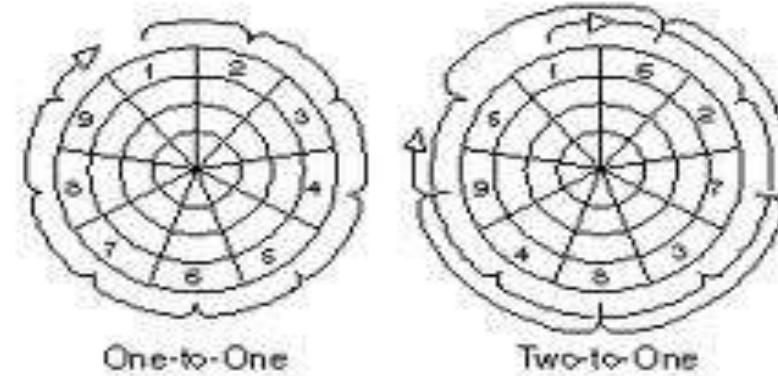
You have total 200.000 records stored in heap file and length of each record is 250 byte.

Here are the specifications of disk used for storage: 512 byte/sector, 20 sector/track, 5 sector/cluster, 3000 track/platter, one sided total 10 plate, rotational time 7200 rpm, seek time 8 msec. Records do not span between sectors

- ▶ Blocking factor of heap file: 10
- ▶ Number of blocks to store whole file: 20.000 blocks
- ▶ Number of blocks to estimate disk capacity: 120.000 blocks
- ▶ Cylinder number to store the data with cylinder based approach : 500
- ▶ Time to read one block : 14,225 ms
- ▶ Time to read file from beginning to end (in worst case): 284.500sec, 4741,67 min, 79,02 hour

Concepts Related to Sector

- ▶ Interleaving



- ▶ Extent

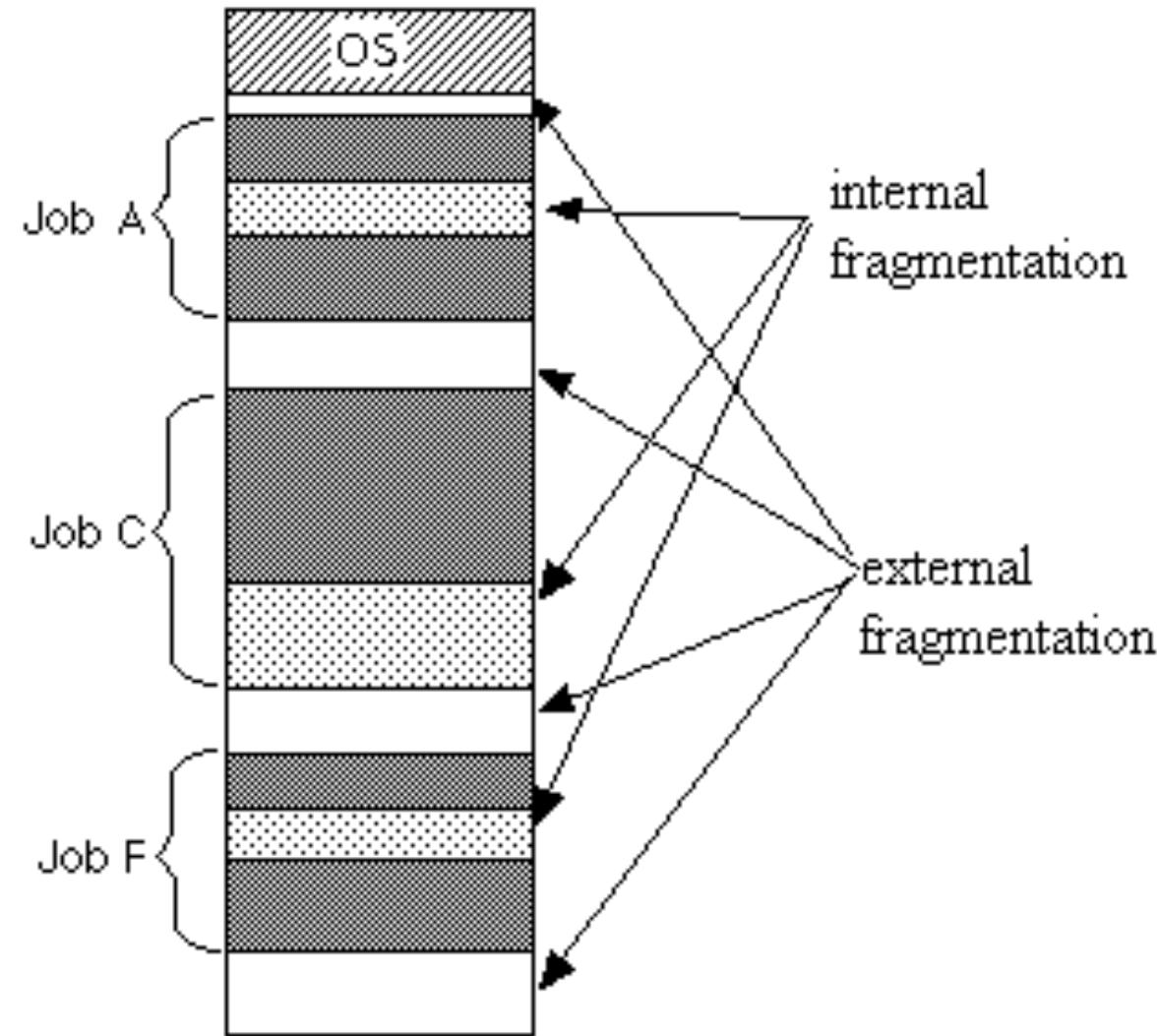
An **extent** is a contiguous area of storage in a computer [file system](#), reserved for a file. When a process creates a file, file-system management software [allocates](#) a whole extent. When writing to the file again, possibly after doing other write operations, the data continues where the previous write left off. This reduces or eliminates [file fragmentation](#) and possibly [file scattering](#) too.

External Fragmentation

- ▶ External fragmentation arises when free memory is separated into small blocks and is interspersed by allocated memory.
- ▶ The result is that, although free storage is available, it is effectively unusable because it is divided into pieces that are too small individually to satisfy the demands of the application. The term "external" refers to the fact that the unusable storage is outside the allocated regions.
- ▶ For example, consider a situation wherein a program allocates 3 continuous blocks of memory and then frees the middle block. The memory allocator can use this free block of memory for future allocations. However, it cannot use this block if the memory to be allocated is larger in size than this free block.
- ▶ External fragmentation also occurs in file systems as many files of different sizes are created, change size, and are deleted. The effect is even worse if a file which is divided into many small pieces is deleted, because this leaves similarly small regions of free spaces.

0x0000	0x1000	0x2000	0x3000	0x4000	0x5000	Comments
						Start with all memory available for allocation.
A	B	C				Allocated three blocks A, B, and C, of size 0x1000.
A		C				Freed block B. Notice that the memory that B used cannot be included for an allocation larger than B's size.

Internal vs External Fragmentation



Disk Failures

- ▶ **Intermittent Failures:** An attempt to read or write a sector is unsuccessful.
- ▶ Solution: **Checksums**
 - ▶ A simple form is based on the parity of the bits in the sector.
 - ▶ A sector computes an error checking code (called checksum) and stores next to sector
 - ▶ Example: If the sequence of bits in a sector were 01101000, then there is an odd number of 1's, so the parity bit is 1.
- ▶ For the parity information
 - ▶ Even number of 1s will have parity 0
 - ▶ Odd number of 1s will have parity 1
- ▶ Example: Data is 1010111
- ▶ If error changes data to 1110111 parity should be 0, but was 1. Detected the error!

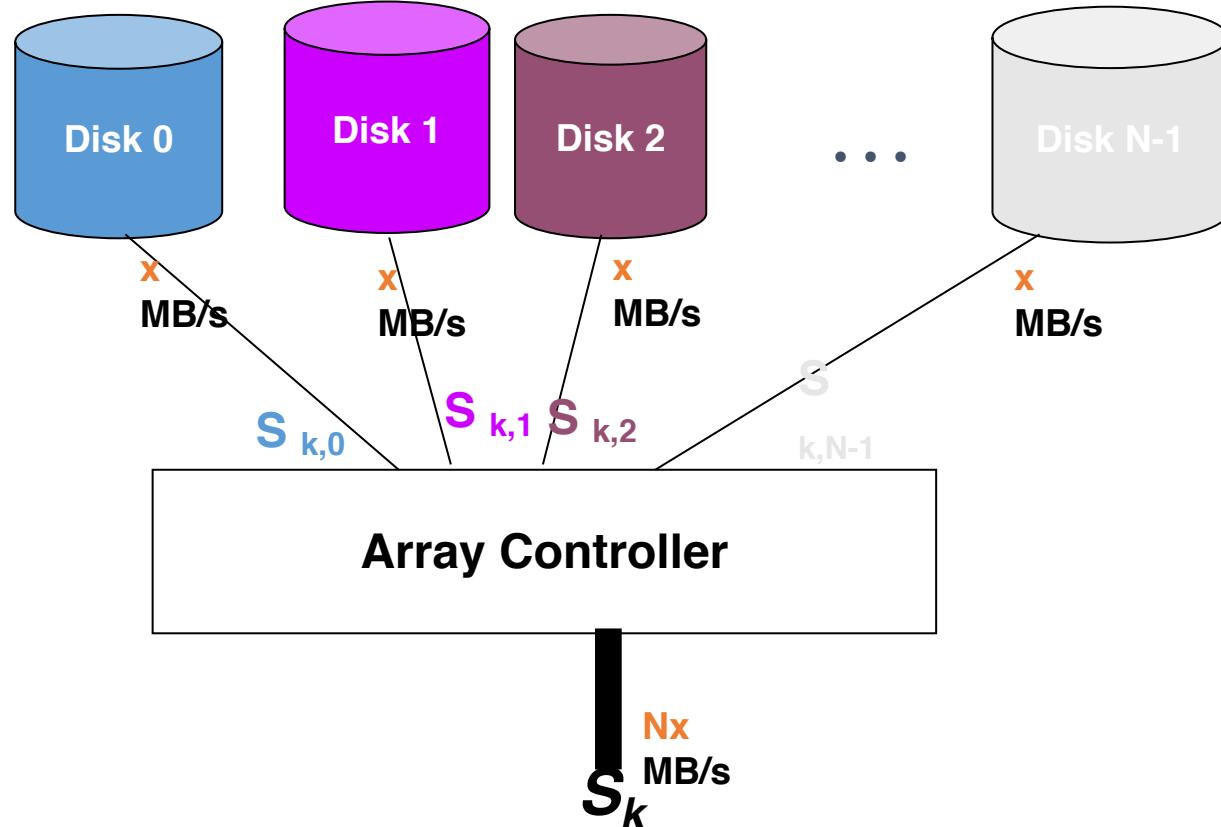
Mirroring as a Redundancy Technique

- ▶ **Disk crashes:** Data is permanently destroyed
- ▶ **Solution: Mirroring**
- ▶ Say a bank has k different hard drives
 - ▶ If each hard drive is error free 99% of the time
 - ▶ The probability that k hard drives will not have errors at the same time is:
$$\text{Availability} = (0.99)^k$$
, so for example if $k=20$ then it is 0.817
- ▶ If we keep multiple copies of the same data in r disks:
 - ▶ The probability of failure is 0.01
 - ▶ The probability of hard drives failing at the same time is $(0.01)^r$, if we assume that failures are independent.
 - ▶ Even if $r=2$, the probability of losing the data drops to 0.0001

Redundant Array of Inexpensive Disks (RAID)

- ▶ Disk Array: Arrangement of several disks that gives abstraction of a single, large disk.
- ▶ Goals: Increase performance and reliability.
 - ▶ Data striping: Data is partitioned
 - ▶ Partitions are distributed over several disks.
- ▶ Note that reliability decreases by reducing storage space utilization.
We use some hard drive space for redundant data.
- ▶ We extend the idea of parity checks through redundancy!
 - ▶ The common term for this class of strategies is RAID!

RAID



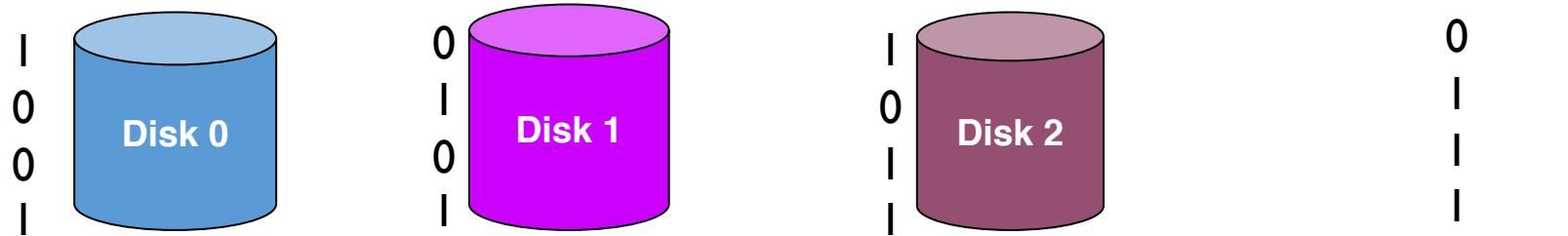
RAID

- ▶ **RAID Level 1:** We mirror each disk, one as a data disk and one as a redundant disk
- ▶ **RAID Level 2:** Use only one redundant disk that store parity blocks
- ▶ What happens if there are two or more disk crashes at the same time?
- ▶ **RAID Level 5:** Each disk is a redundant disk for some of the blocks
 - ▶ Example: If there are 4 disks, the disk 0 is redundant for blocks 4, 8, 12, and so on; disk 1 is redundant for blocks 1, 5, 9, and so on.
- ▶ **RAID Level 6:** It is designed for multiple disk crashes. It uses Hamming code.

RAID Error Correction

Data Striping by 4 bits, lets assume we have one parity for the 3 disks

100101011011



If Disk 0 Crashes, we can recover the bits by simply XOR all the others with parity bits.

Disk 0 = Disk 1 XOR Disk 2 XOR Parity

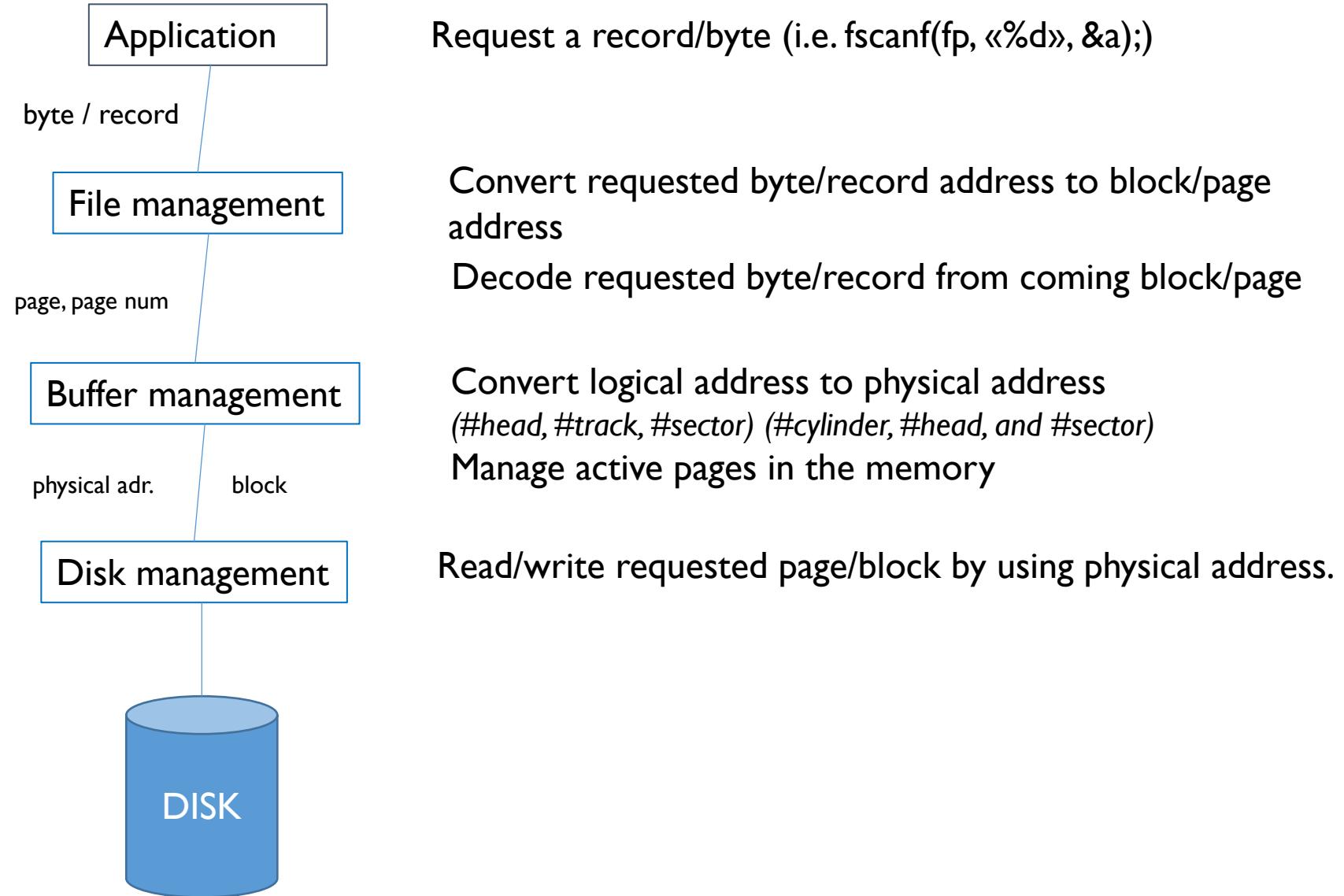


BBM 371 – Data Management

Lecture 3: File Concepts

24.10.2019

Journey of Byte



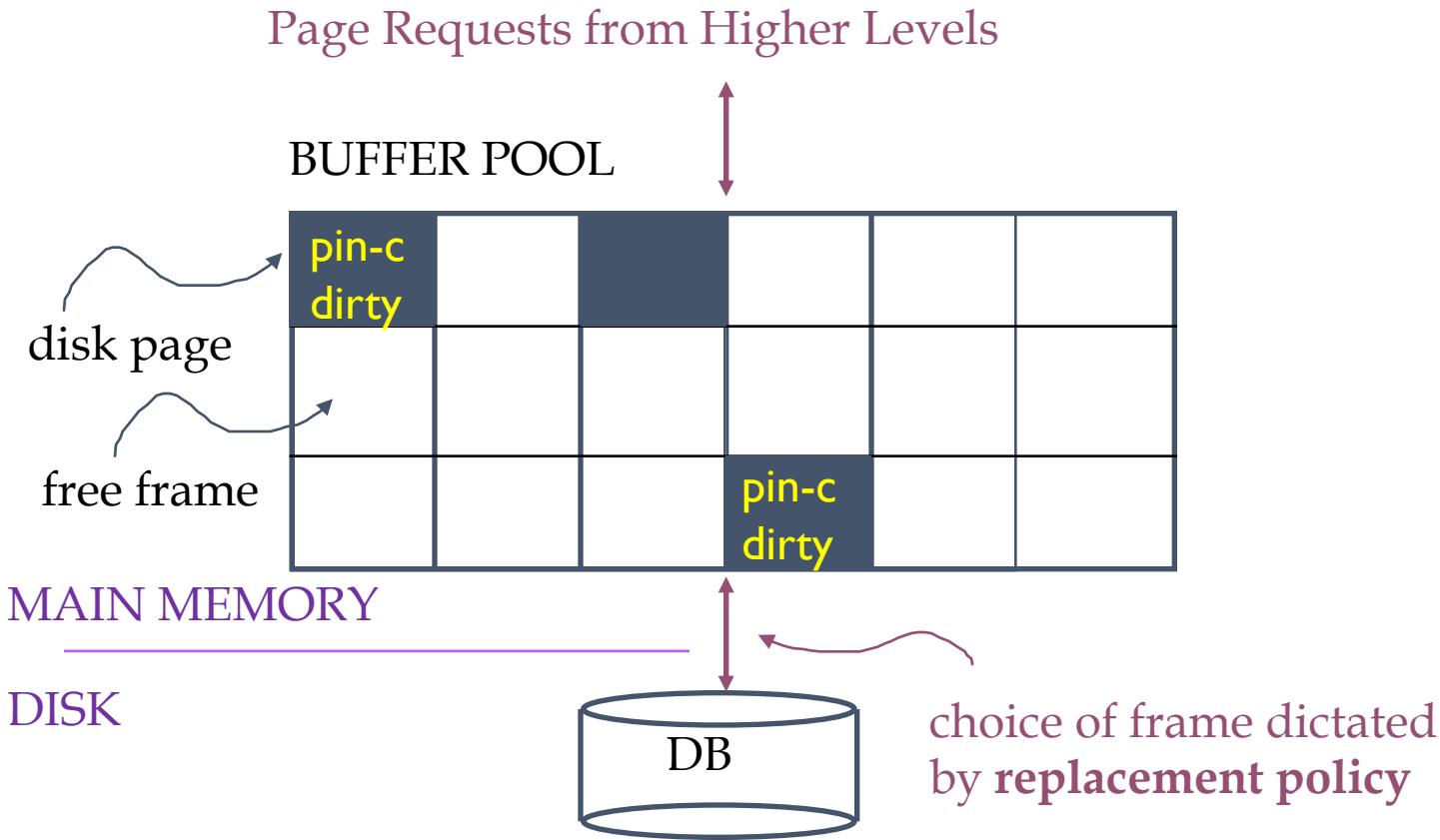
Disk Space Management

- ▶ Lowest layer of DBMS software manages space on disk.
- ▶ Higher levels call upon this layer to:
 - ▶ allocate/de-allocate a page
 - ▶ read/write a page
- ▶ Request for a sequence of pages must be satisfied by allocating the pages sequentially on disk! Higher levels don't need to know how this is done.

Buffer Management

- ▶ All Data Pages must be in memory in order to be accessed
- ▶ Buffer Manager
 - ▶ Deals with asking Disk Space Manager for pages from disk and store them into memory
 - ▶ Sends Disk Space Manager pages to be written to disk
- ▶ Memory is faster than Disk
 - ▶ Keep as much data as possible in memory
 - ▶ If enough space is not available, need a policy to decide what pages to remove from memory. **Replacement policy**

Buffer Management in a DBMS



- ▶ Data must be in RAM for DBMS to operate on it!
- ▶ Table of **<frame#, pageid>** pairs is maintained.

Buffer Pool

- ▶ Frame
 - ▶ Data structure that can hold a data page and control flags
- ▶ Buffer pool
 - ▶ Array of frames of size N
- ▶ In C

```
#define POOL_SIZE 100
#define PAGE_SIZE 4096
typedef struct frame {
    int pin_count;
    bool dirty;
    char page[PAGE_SIZE];
} frame;
frame buffer_pool[POOL_SIZE];
```

Operational mode

- ▶ All requested data pages must first be placed into the buffer pool.
- ▶ **pin_count** is used to keep track of number of transactions that are using the page
 - ▶ zero means nobody is using it
- ▶ **dirty** is used as a flag (dirty bit) to indicate that a page has been modified since read from disk
 - ▶ Need to flush it to disk if the page is to be evicted from pool
- ▶ Page is an array of bytes where the actual data is stored in
 - ▶ Need to interpret these bytes as int, char, Date data types supported by SQL
 - ▶ This is very complex and tricky!

Buffer replacement

- ▶ If we need to bring a page from disk, we need to find a frame in the buffer to hold it
- ▶ Buffer pool keeps track on the number of frames in use
 - ▶ List of frames that are free (Linked list of free frame nums)
- ▶ If there is a free frame, we use it
 - ▶ Remove from the list of free frames
 - ▶ Increment the pin_count
 - ▶ Store the data page into the byte array (page field)
- ▶ If the buffer is full, we need a policy to decide which page will be evicted

Buffer access & replacement algorithm

- ▶ Upon request of page X do
 - ▶ Look for page X in buffer pool
 - ▶ If found, ++pin_count , then return it
 - ▶ else, determine if there is a free frame Y in the pool
 - ▶ If frame Y is found
 - ▶ Increment its pin_count (++pin_count)
 - ▶ Read page from disk into the frame's byte array
 - ▶ Return it
 - ▶ else, use a replacement policy to find a frame Z to replace
 - ▶ Z must have $\text{pin_count} == 0$
 - ▶ If dirty bit is set, write data currently in Z to disk
 - ▶ Read the new page into the byte array in the frame Z
 - ▶ Increment the pin_count in Z (++pin_count)
 - ▶ Return it
 - ▶ else wait or abort transaction (insufficient resources)

Some remarks

- ▶ Need to make sure pin_count is 0
 - ▶ Nobody is using the frame
- ▶ Need to write the data to disk if dirty bit is true
- ▶ This latter approach is called Lazy update
 - ▶ Write to disk only when you have to!!!
 - ▶ Careful, if power fails, you are in trouble.
 - ▶ DBMS need to periodically flush pages to disk
 - ▶ Force write
- ▶ If no page is found with pin_count equal to 0, then either:
 - ▶ Wait until one is freed
 - ▶ Abort the transaction (insufficient resources)

Buffer Replacement policies

- ▶ LRU – Least Recently Used
 - ▶ Evicts the page that is the least recently used page in the pool.
 - ▶ Can be implemented by having a queue with the frame numbers.
 - ▶ Head of the queue is the LRU
 - ▶ Each time a page is used it must be removed from current queue position and put back at the end
 - ▶ This queue need a method `erase()` that can erase stuff from the middle of the queue
- ▶ LRU is the most widely used policy for buffer replacement
 - ▶ Most cache managers also use it

Other policies

- ▶ **Most Recently Used**
 - ▶ Evicts the page that was most recently accessed
 - ▶ Can be implemented with a priority queue
- ▶ **FIFO**
 - ▶ Pages are replaced in a strict First-In-First Out
 - ▶ Can be implemented with a FIFO List (queue in the strict sense)
- ▶ **Random**
 - ▶ Pick any page at random for replacement

Sample Buffer Pool

Page_no = 1 Pin_count = 3 Dirty = 1 Last Used: 12:34:05	Page_no = 2 Pin_count = 0 Dirty = 1 Last Used: 12:35:05	Page_no = 3 Pin_count = 1 Dirty = 0 Last Used: 12:36:05	Page_no = 4 Pin_count = 2 Dirty = 0 Last Used: 12:37:05	Page_no = 5 Pin_count = 0 Dirty = 0 Last Used: 12:38:05
Page_no = 6 Pin_count = 0 Dirty = 0 Last Used: 12:29:05	Page_no = 7 Pin_count = 1 Dirty = 1 Last Used: 12:20:05	Page_no = 8 Pin_count = 0 Dirty = 1 Last Used: 12:40:05	Page_no = 9 Pin_count = 2 Dirty = 0 Last Used: 12:27:05	Page_no = 10 Pin_count = 0 Dirty = 1 Last Used: 12:39:05

Which page should be removed if LRU is used as the policy:.....

Which page should be removed if MRU is used as the policy :.....

Which pages do not need to be written to disc, if it is removed:.....

Which pages could not be removed in this situation:.....

DBMS vs. OS File System

- ▶ OS does disk space & buffer management: why not let OS manage these tasks?
- ▶ Some limitations, e.g., files can't span disks.
- ▶ Buffer management in DBMS requires ability to:
 - ▶ pin a page in buffer pool, force a page to disk,
 - ▶ adjust replacement policy, and pre-fetch pages based on access patterns in typical DB operations.

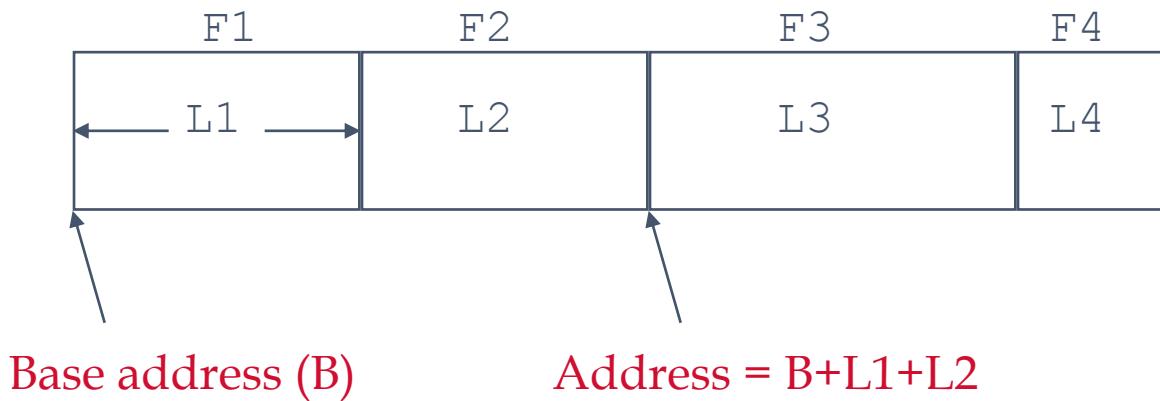
Record Formats

- ▶ Organization of records whether field length of record
 - ▶ Fixed
 - ▶ Variable

Note: Type and number of fields are identical for all tuples

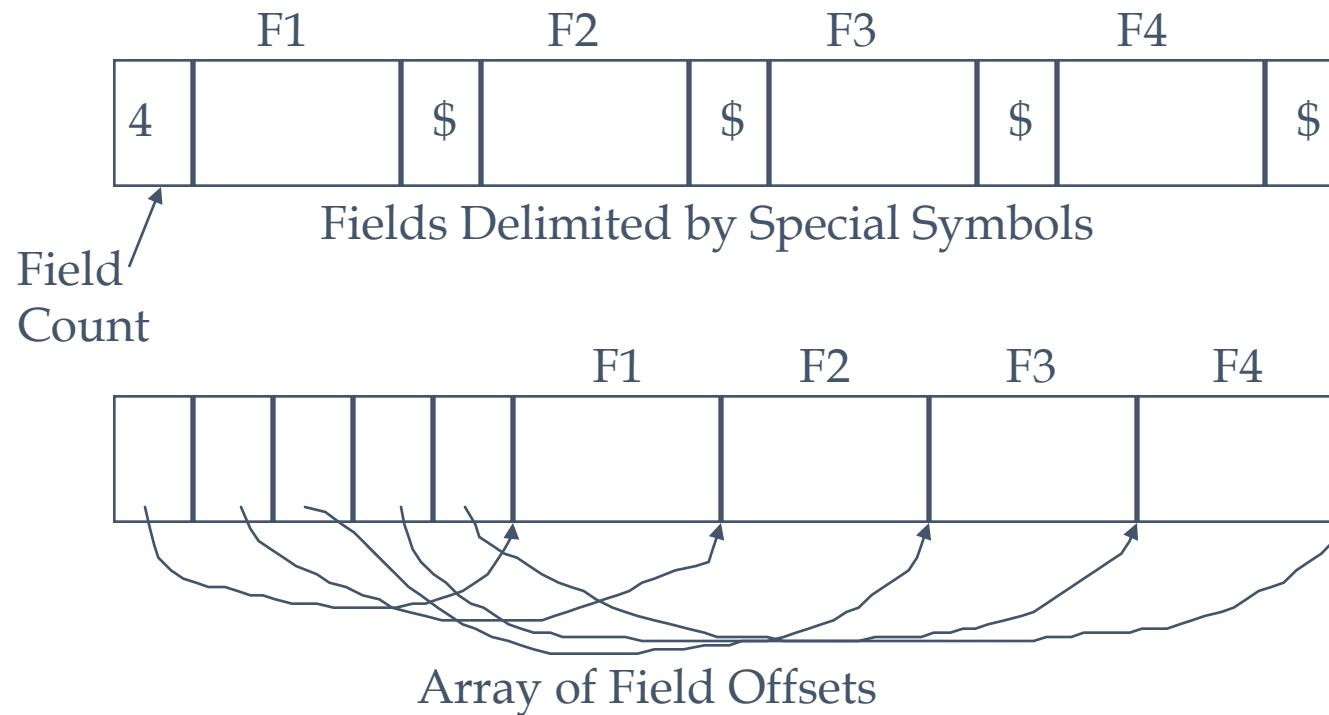
Fixed Length Records

- All fields can be placed continuous
- Finding i^{th} field address requires adding length of previous fields to base address.



Variable Length Records

- ▶ Two alternative formats (# fields is fixed):



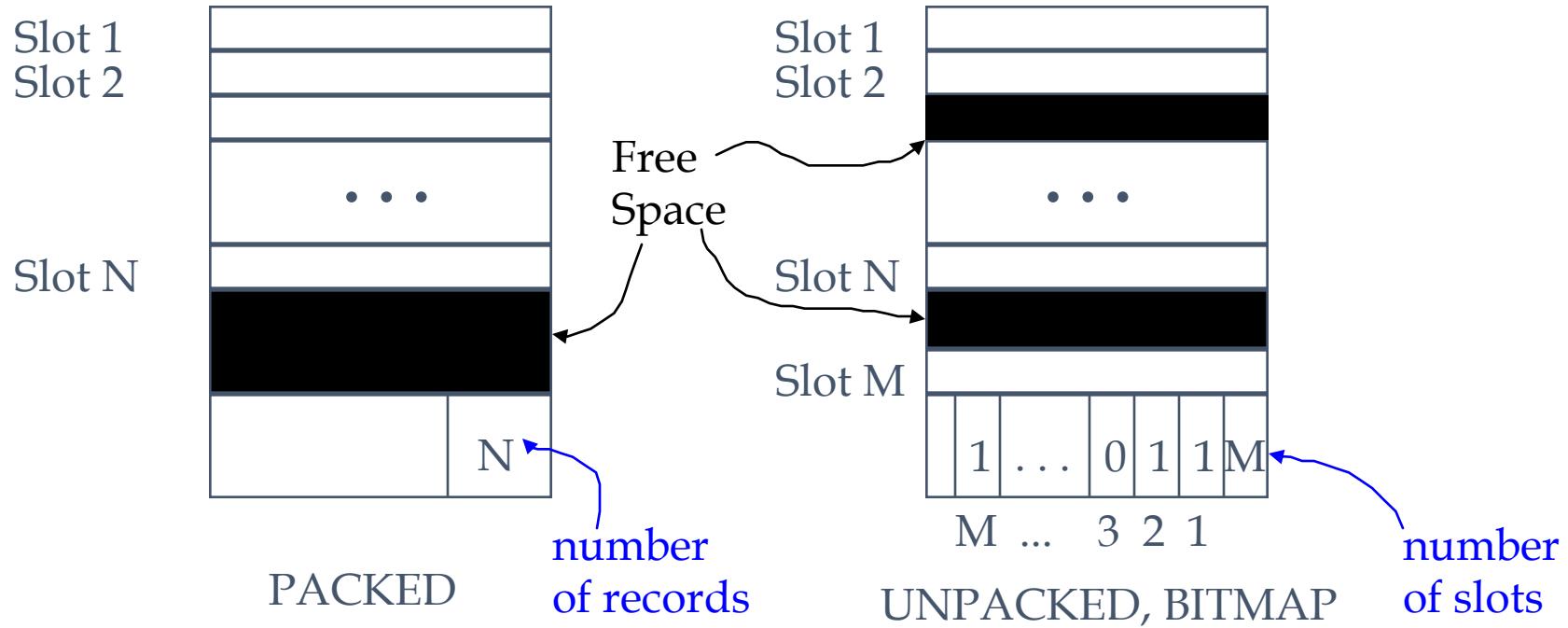
Variable Length Records(Cont.)

- ▶ In first
 - ▶ All previous fields must be scanned to access the desired records
- ▶ In Second
 - ▶ Second offers direct access to i^{th} field
 - ▶ Pointers to begin and end of the field
 - ▶ Efficient storage for nulls
 - ▶ Small directory overhead

Disadvantage of Variable Length

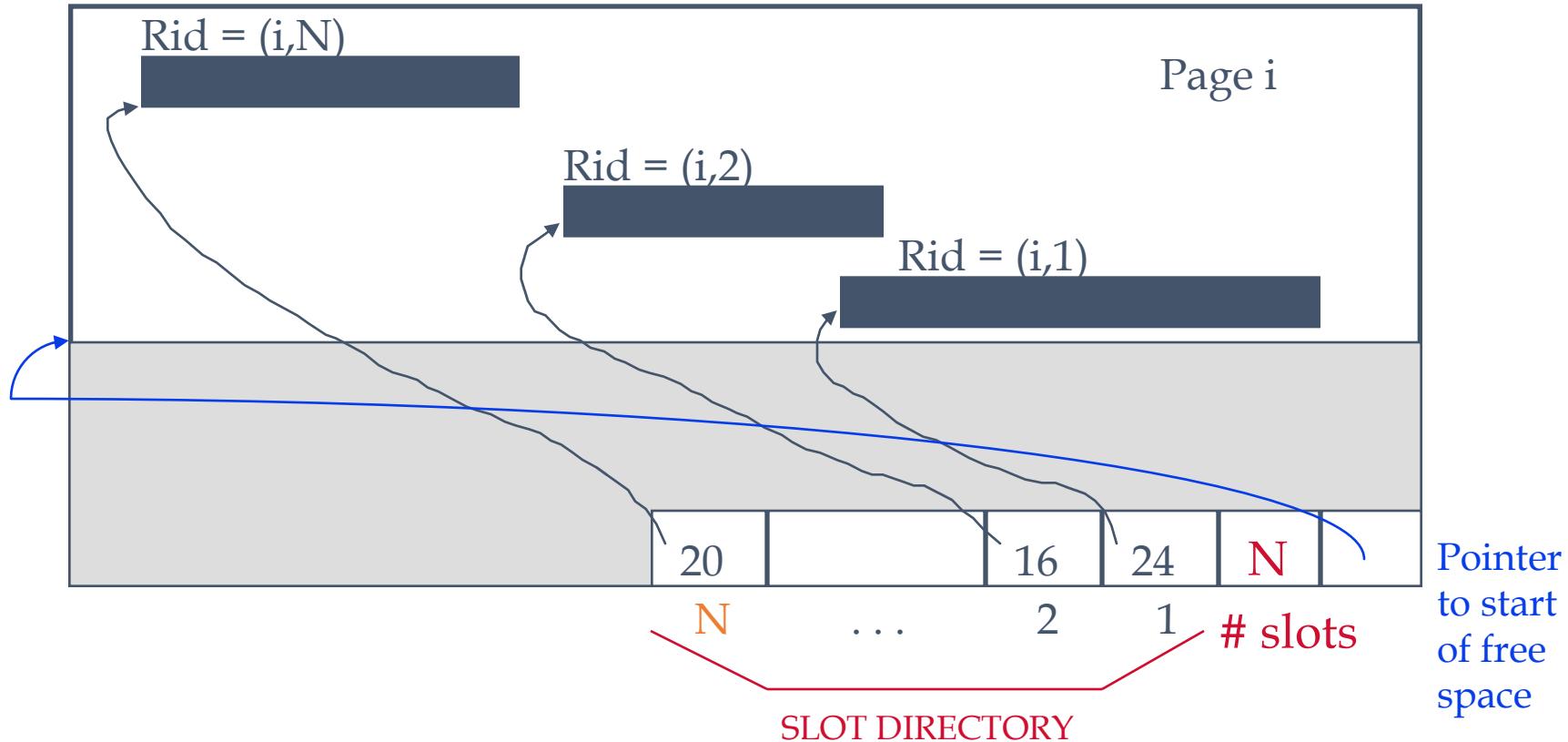
- ▶ If field grows to larger size:
 - ▶ Subsequent fields must be shifted
 - ▶ Offsets must be updated
- ▶ If after update, record does not fit in its current page:
 - ▶ memory address of the page is changed
 - ▶ references to old address must be updated
- ▶ If record does not fit in any page:
 - ▶ Record must be broken down to smaller records
 - ▶ Chaining must be set up for the smaller records

Page Formats: Fixed Length Records



- ▶ In first alternative, moving records for free space management changes memory address of record ; may not be acceptable.

Page Formats: Variable Length Records



- ▶ Can move records on page without changing memory address of records; so, attractive for fixed-length records too.

Page Formats: Variable Length Records

- ▶ Keep a directory for slots that show <record offset, record length>
- ▶ Keep a pointer to point free space
- ▶ For placement of a record
 - ▶ If it is possible, insert in free space
 - ▶ Reorganize page to combine wasted space then insert
 - ▶ Insert another page
- ▶ For deleting a record
 - ▶ Put –I to record offset information in directory

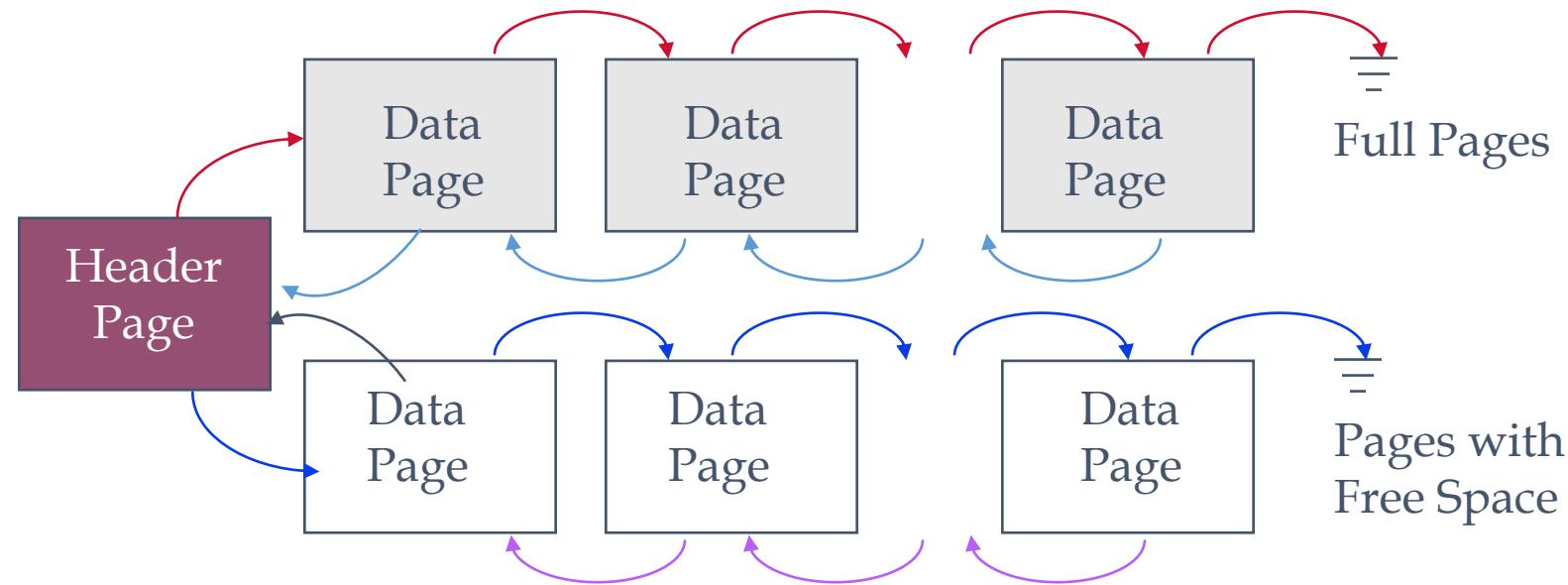
Files of Records

- ▶ Page or block is OK when doing I/O, but higher levels of DBMS operate on records, and files of records.
- ▶ **FILE:** A collection of pages, each containing a collection of records.
Must support:
 - ▶ insert/delete/modify record
 - ▶ read a particular record
 - ▶ scan all records (possibly with some conditions on the records to be retrieved)

Unordered (Heap) Files

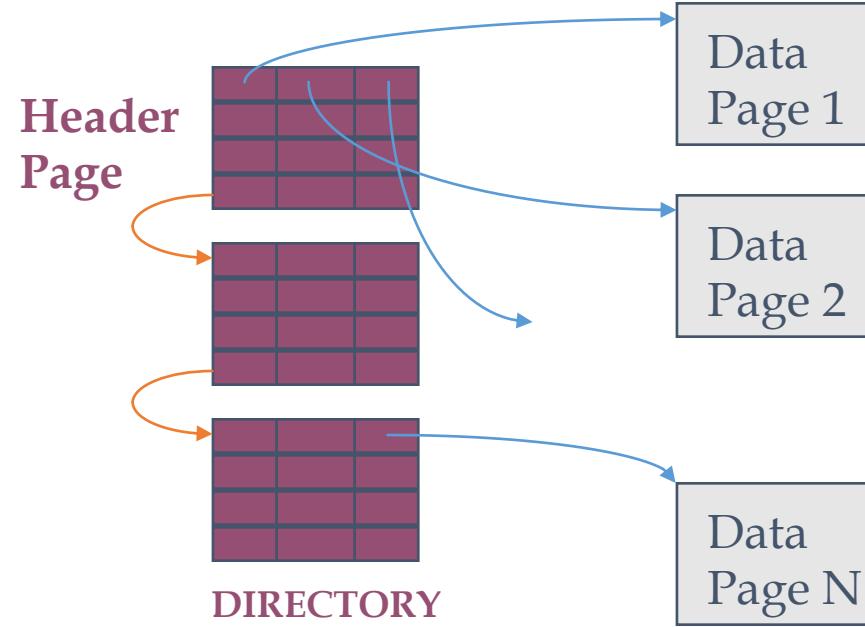
- ▶ Synonym of «Pile» and «Sequential»
- ▶ Simplest file structure as records are in no particular order.
- ▶ As file grows and shrinks, disk pages are allocated and de-allocated.
- ▶ To support record level operations, we must:
 - ▶ keep track of the pages in a file
 - ▶ keep track of free space on pages
 - ▶ keep track of the records on a page
- ▶ There are many alternatives for keeping track of this.

Heap File Implemented as a List



- ▶ The header page id and Heap file name must be stored someplace on disk.
- ▶ Each page contains two 'pointers' plus data.

Heap File Using a Page Directory



- ▶ The entry for a page can include the number of free bytes on the page.
- ▶ The directory is a collection of pages; linked list implementation is just one alternative

Searching on Heap Files

- ▶ **Equality search:** to search a record with given value of one or more of its fields
- ▶ **Range search:** to find all records which satisfy given min and max values for one of fields

- ▶ We must search the whole file.
- ▶ In general, (bf is blocking factor. N is the size of the file in terms of the number of records) :
 - ▶ At least 1 block is accessed (I/O cost : 1)
 - ▶ At most N/bf blocks are accessed.
 - ▶ On average $N/2bf$

- ▶ Thus, time to find and read a record in a file is approximately :

$$\text{Time to fetch one record} = (N/2bf) * \text{time to read one block}$$

Time to read one block = seek time + rotational delay + block transfer
time

More and more ...

- ▶ Time to read all records = N/bf * *time to read per block*
- ▶ Time to add new record
 - ▶ = time to read one block (for last block) + Time to write one block (for last block)
- ▶ if the last block is full
 - ▶ = time to read one block (for last block) + time to write new one block (for new last block)

More and more ...

- ▶ Time to update one fixed length record = Time to fetch one record + time to write one block
- ▶ Time to update one variable length record = Time to delete one record + time to add new record
- ▶ Time to delete one record = ??
You can mark the record (replace the first character with \$)

Exercise

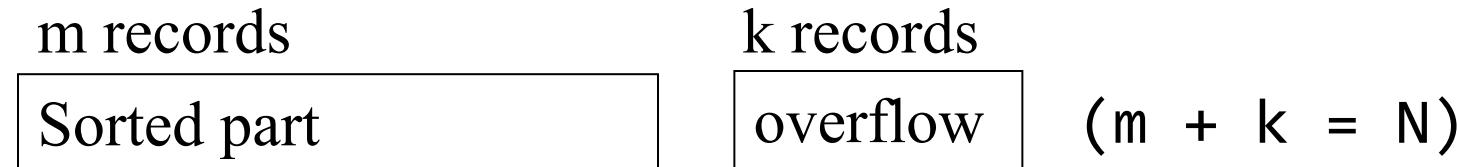
- ▶ FileA: 10000 records , BF = 100, 4 extents
- ▶ File B: 5000 records, BF = 150, 3 extents
- ▶ Time to find the number of common records of FileA and B
 - Time to read FileA= $4 * (\text{seek time} + \text{rotational delay}) + (10000/100) * \text{block transfer time}$
 - Time to read FileB = $3 * (\text{seek time} + \text{rotational delay}) + (5000/150) * \text{block transfer time}$
 - = Time to read FileA + 100 * Time to read FileB
(imagine you've got only two frames in the buffer pool.)
- ▶ Read FileA and compare each record of FileA with whole records in FileB

Sorted (Sequential) Files

- ▶ A sorted file should stay in order, but it is impossible.
 - ▶ Additions/deletions
 - ▶ A sorted file uses an overflow pages list for newly added records
 - ▶ Overflow pages list does not have an ordering
 - ▶ For equality search:
 - ▶ Search on sorted area
 - ▶ And then search on overflow area
- ***If there are too many overflow areas, the access time increase up to that of a sequential file.

Searching for a record

- We can do binary search (assuming fixed-length records) in the sorted part.



- Worst case to fetch a record :

$$T_F = \log_2 (m/bf) * \text{time to read per block.}$$

- If the record is not found, search the overflow area too. Thus total time is:

$$T_F = \log_2 (m/bf) * \text{time to read per block} + \\ k/bf * \text{time to read per block}$$



BBM371- Data Management

Lecture 4: Index Files

31.10.2019

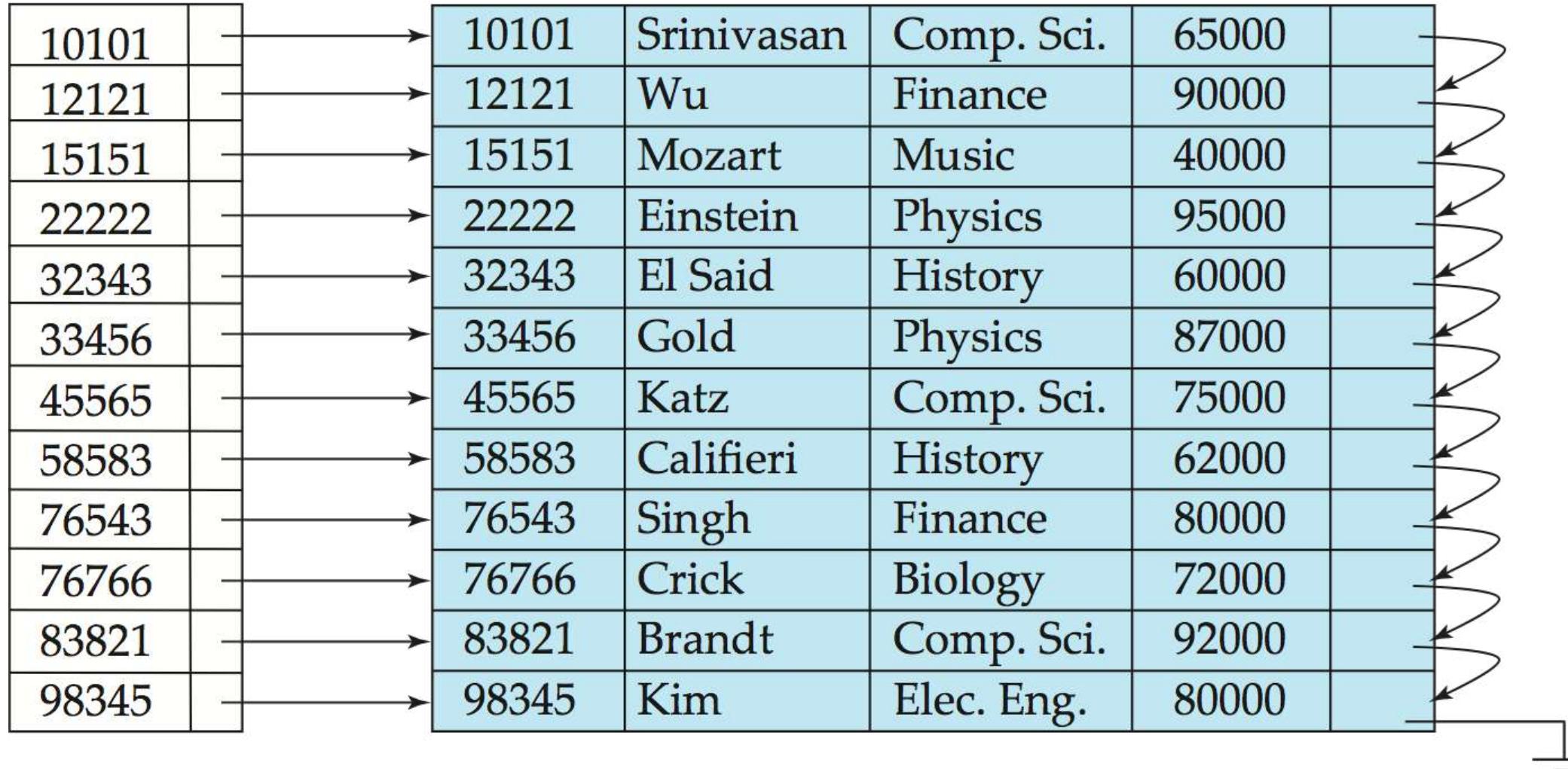
A Simple Index

- ▶ Consider the query: *SELECT * FROM R*
We have to scan all records in R
- ▶ For *SELECT * FROM R WHERE a=10*
It is possible to look at only a small fraction of records in R with a good indexing strategy.
- ▶ A few comments about our **Index Organization**:
 - ▶ The index is easier to use compared to the data file because
 - 1) it uses fixed-length records
 - 2) it is likely to be much smaller than the data file.
 - ▶ By requiring fixed-length records in the index file, we impose a limit on the size of the primary key field. This could cause problems.
 - ▶ The index could carry more information than the key and reference fields. (e.g., we could keep the length of each data file record in the index as well).

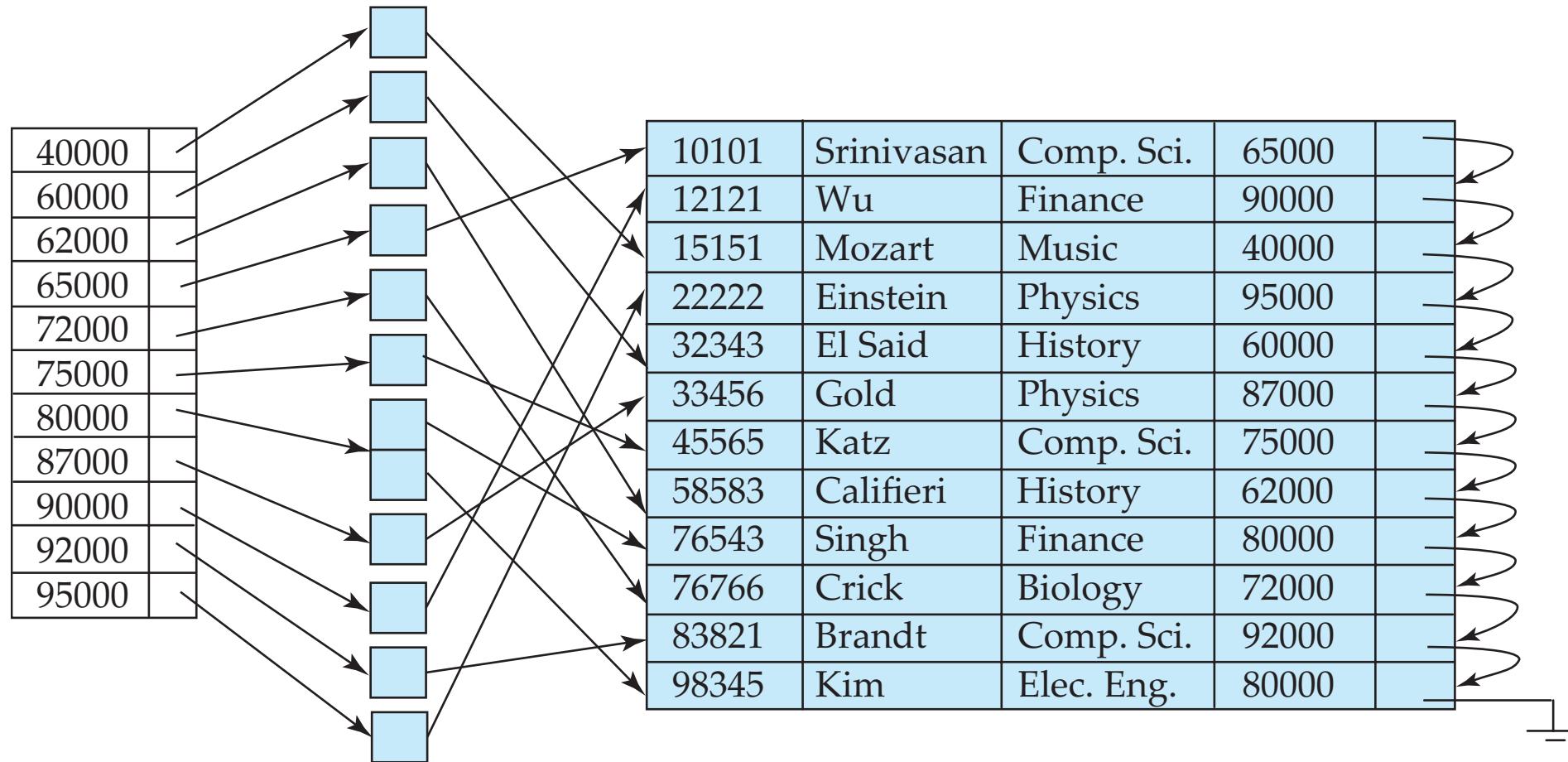
Some Terminology

- ▶ A data file stores the complete records.
- ▶ Data file can have one or more index files.
- ▶ Index files stores **search keys** and **pointers** to data-file records.
- ▶ Indexes can be **primary** or **secondary**
 - ▶ Primary Index: Data file is structured with respect to its search key. E.g. Primary key of table
 - ▶ Secondary Index: Built on some other attributes of the table, the order of records in the data-file are independent from the search key
- ▶ Sequential file: Records are sorted with respect to the primary key.

Primary Index Example

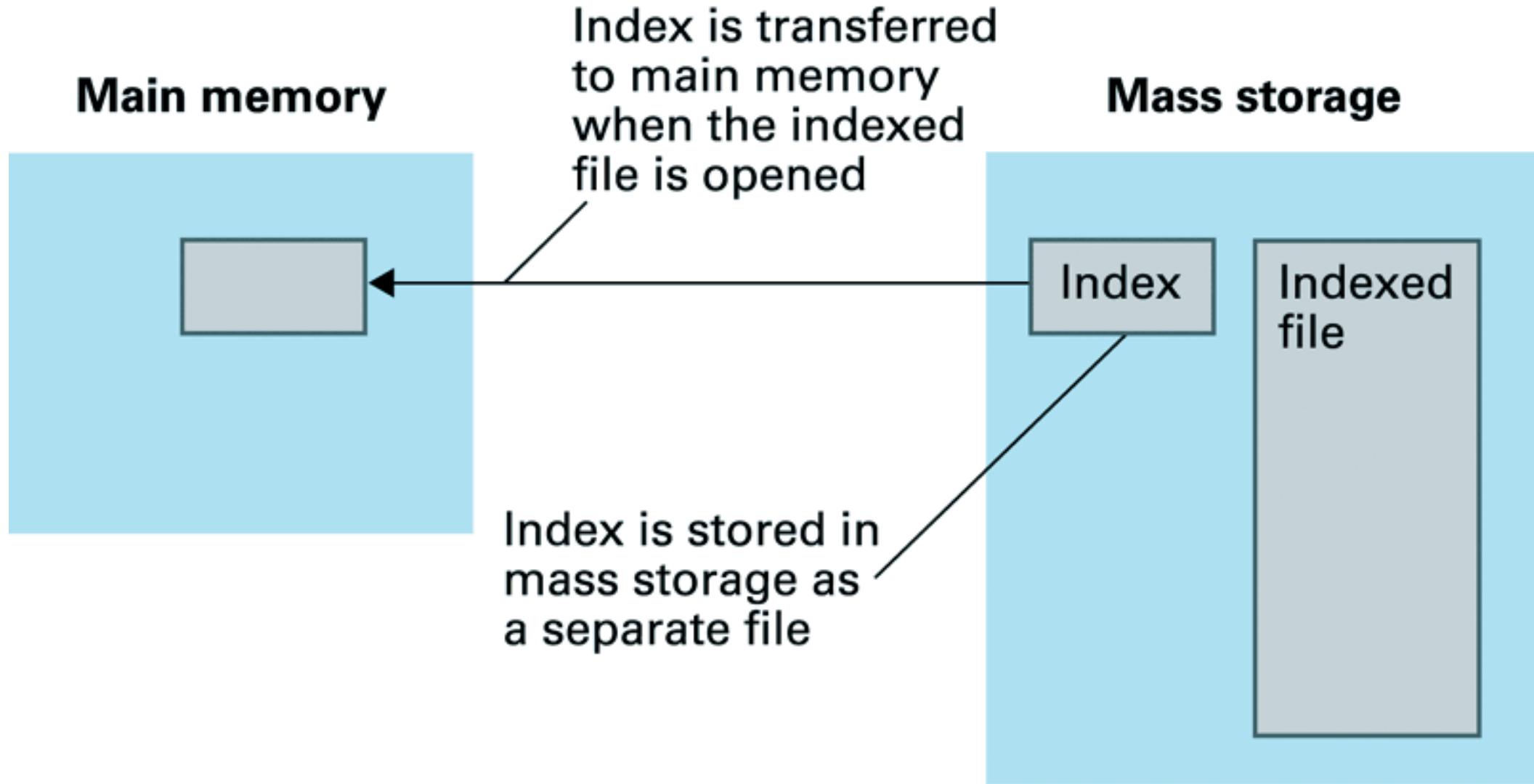


Secondary Index Example



Secondary index on *salary* field of *instructor*

Opening an indexed file



Basic Operations on an Indexed Entry-Sequenced File

- ▶ Assumption: the index is small enough to be held in memory. Later on, we will see what can be done when this is not the case.
 - ▶ Create the original empty index and data files
 - ▶ Load the index into memory before using it.
 - ▶ Rewrite the index file from memory after using it.
 - ▶ Add records to the data file and index.
 - ▶ Delete records from the data file (and index file).
 - ▶ Update records in the data file if update changes the key update the index file.

Creating, Loading and Re-writing

- ▶ The index is represented as an array of records. The loading into memory can be done sequentially, reading a large number of index records (which are short) at once.
- ▶ What happens if the index changed but its re-writing does not take place or takes place incompletely?
 - ▶ Use a mechanism for indicating whether or not the index is out of date.
 - ▶ Have a procedure that reconstructs the index from the data file in case it is out of date.

Record Addition

- ▶ When we add a record, both the data file and the index should be updated.
- ▶ In the data file the **byte-offset** of the new record should be saved.
 - ▶ If it is a simple **heap file**, the record can be added to an arbitrary location.
 - ▶ If it is a **sequential file**, the record should be added with respect to its primary key
- ▶ Since the index is sorted, the location of the new search key does matter: we have to shift all the keys that belong after the one we are inserting to open up space for the new record. However, this operation is not too costly as it is performed in memory.

Record Deletion

- ▶ Record deletion can be done using several methods
- ▶ In addition, however, the index record corresponding to the data record being deleted must also be deleted. Once again, since this deletion takes place in memory, the record shifting is not too costly.

Record Updating

- ▶ Record updating falls into two categories:
 - ▶ The update changes the value of the index key field.
 - ▶ The update does not affect the index key field.
- ▶ In the first case, both the index and data file may need to be reordered. The update is easiest to deal with if it is conceptualized as a delete followed by an insert (but the user needs not know about this).
- ▶ In the second case, the index does not need reordering, but the data file may. If the updated record is smaller than the original one, it can be re-written at the same location. If, however, it is larger, then a new spot has to be found for it. Again the delete/insert solution can be used.

Indexes that are too large to hold in memory

- ▶ Problems:

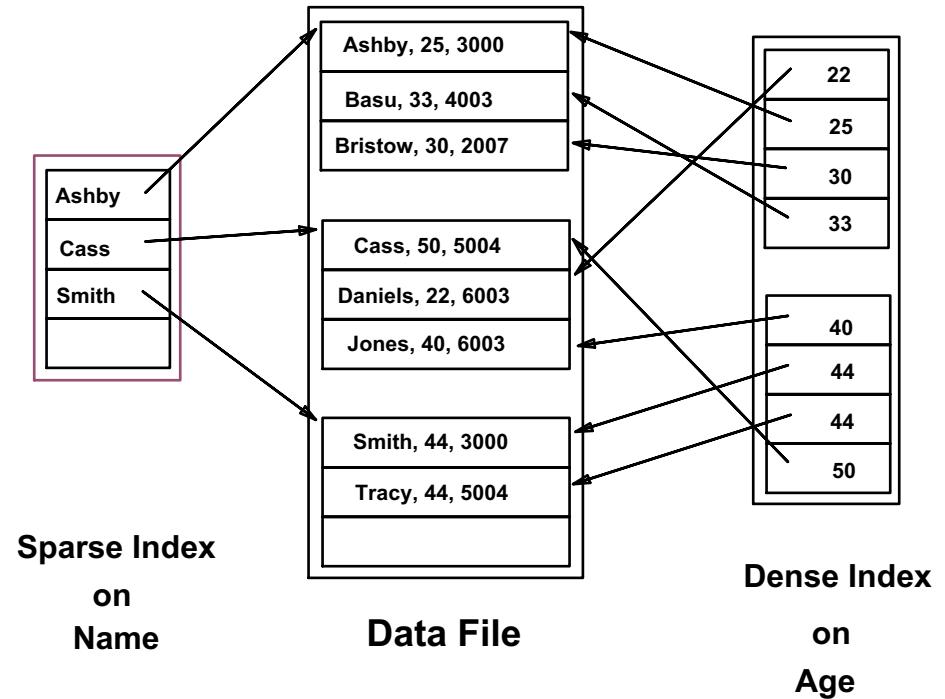
- ▶ Binary searching requires several seeks rather than being performed at memory speed.
- ▶ Index rearrangement requires shifting or sorting records on secondary storage ==> Extremely time consuming.

- ▶ Solutions:

- ▶ Use a hashed organization
- ▶ Use a tree-structured index (e.g., a B-Tree)

Index Classification

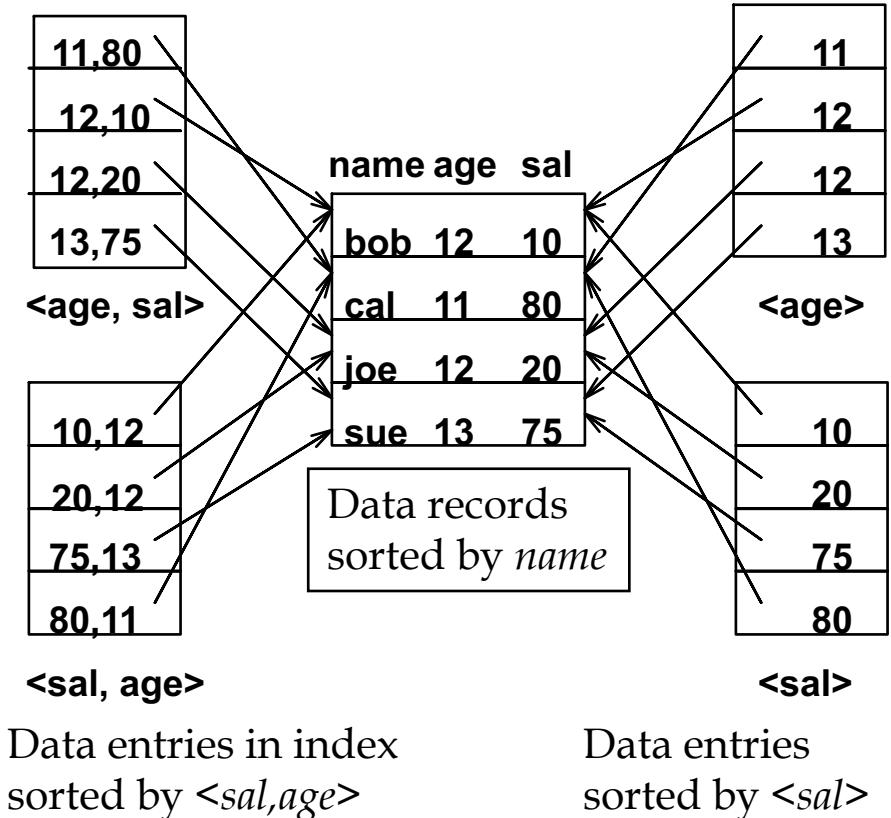
- ▶ Dense vs. Sparse: If there is at least one data entry per search key value (in some data record), then dense.
- ▶ Sparse indexes points only to a group of records, perhaps to first records in a block
- ▶ Sparse indexes are smaller; however, some useful optimizations are based on dense indexes.



Composite Search Keys

- Composite Search Keys: Search on a combination of fields.
 - Equality query: Every field value is equal to a constant value. E.g. $\langle \text{sal}, \text{age} \rangle$ index:
 - $\text{age}=20$ and $\text{sal}=75$
 - Range query: Some field value is not a constant. E.g.:
 - $\text{age}=20$ and $\text{sal} > 10$
- Using index $\langle \text{age}, \text{sal} \rangle$:
 - Can find $\text{age}=20$ and $\text{sal}>10$ efficiently
 - Not efficient for $\text{age}>20$ and $\text{sal}=10$

Examples of composite key indexes using lexicographic order.



Sequential File

10	
20	

Blocking Factor = 2

30	
40	

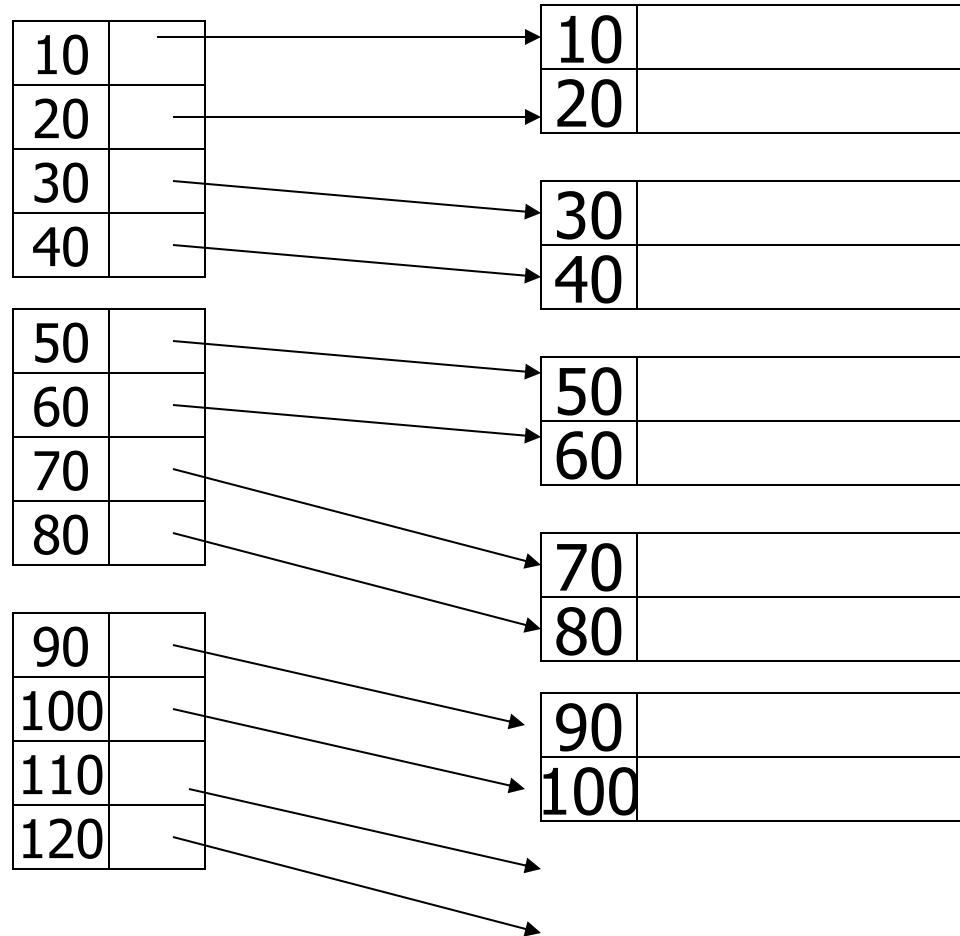
50	
60	

70	
80	

90	
100	

Dense Index

Index BF = 4



Sequential File

Data File
BF = 2

Sparse Index

10	
30	
50	
70	

Sequential File

10	
20	

30	
40	

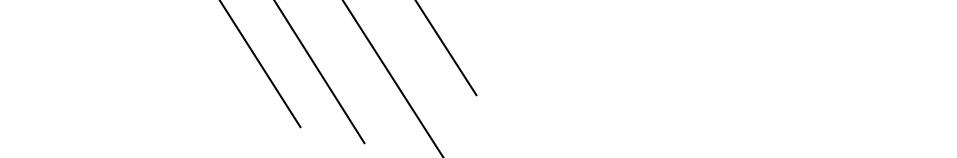
50	
60	

70	
80	

90	
100	

90	
110	
130	
150	

170	
190	
210	
230	



Sparse 2nd level

10	
90	
170	
250	

330	
410	
490	
570	

10	
30	
50	
70	

90	
110	
130	
150	

170	
190	
210	
230	

Sequential File

10	
20	

30	
40	

50	
60	

70	
80	

90	
100	

Sparse vs. Dense Tradeoff

- ▶ **Sparse:** Less index space per record can keep more of index in memory
 - ▶ The data-file must be sorted w.r.t. search key.
- ▶ **Dense:** Can tell if any record exists without accessing file
 - ▶ Can work even if the data file is not ordered!

Duplicate keys

10	
10	

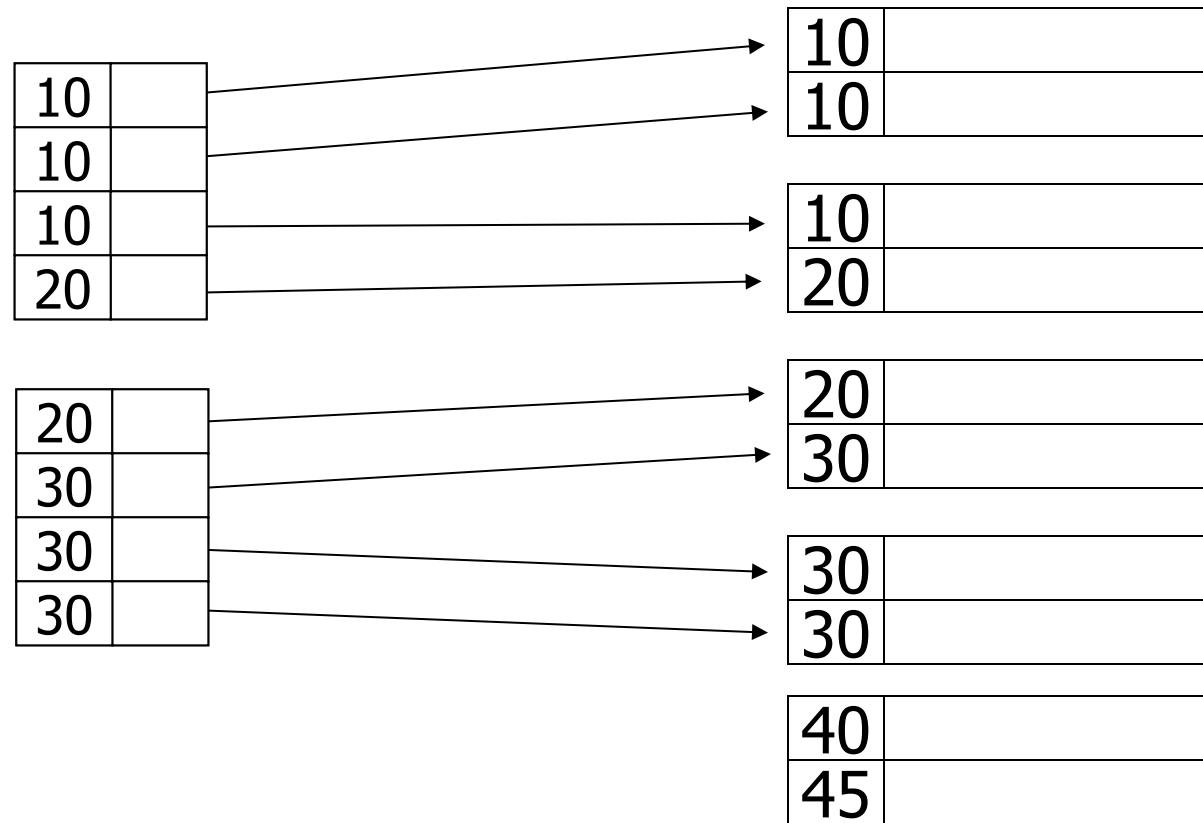
10	
20	

20	
30	

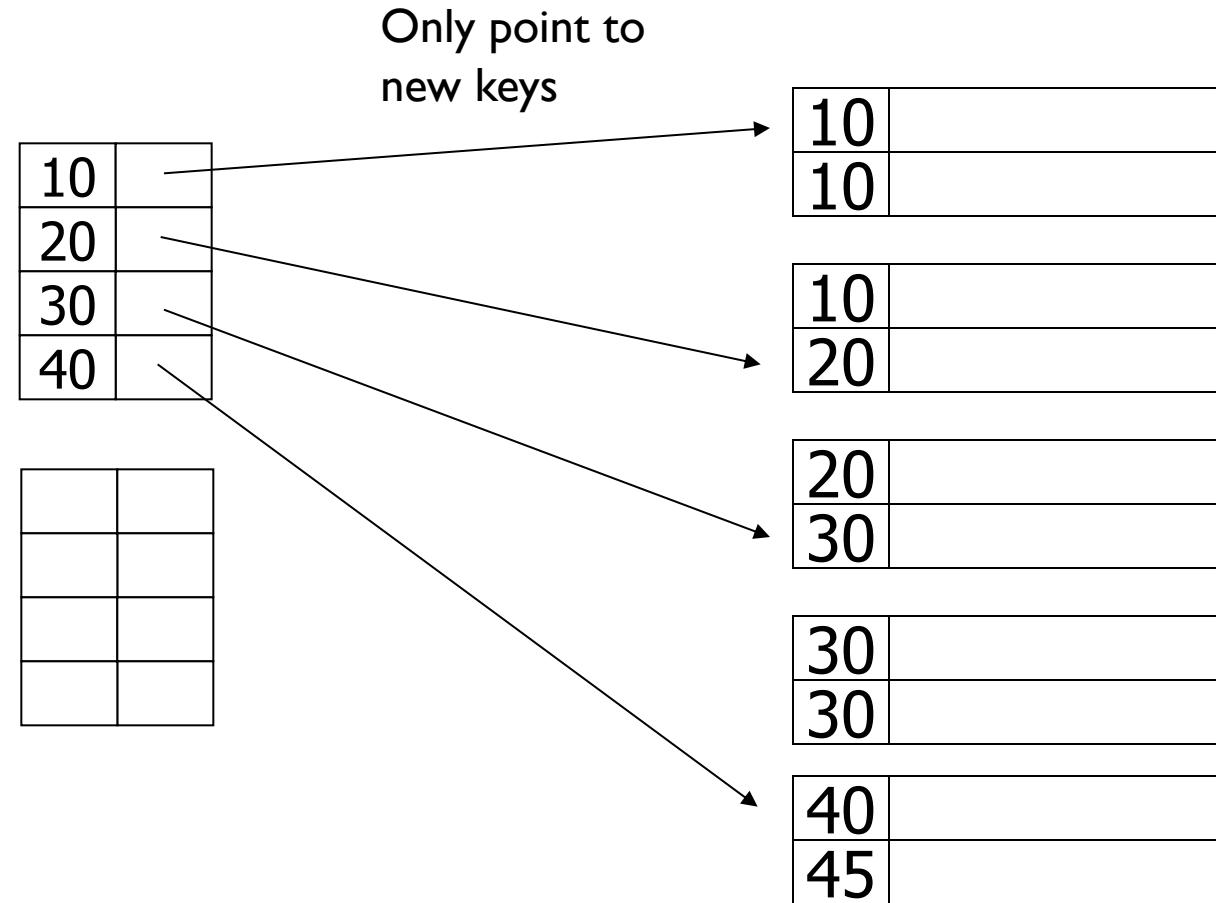
30	
30	

40	
45	

Dense index, one way to implement?

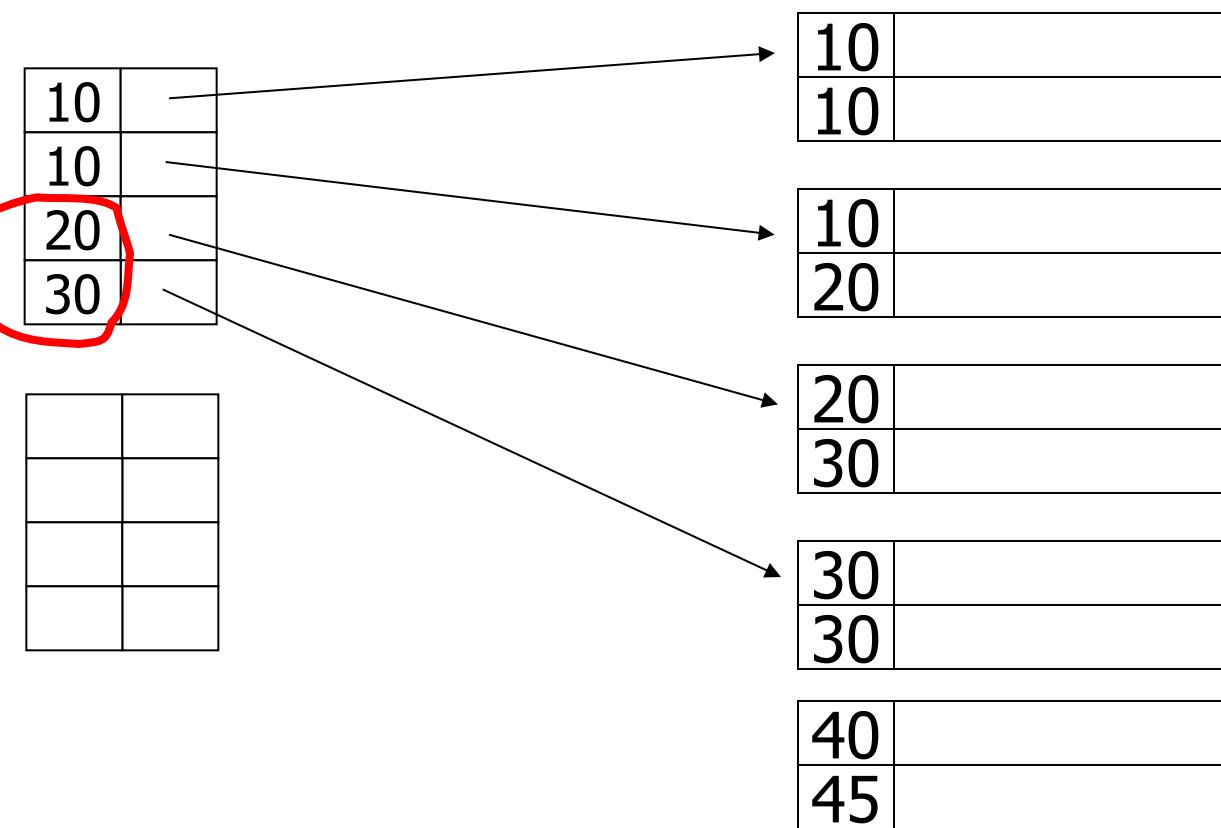


Duplicate Keys (Alternative)



Sparse index, one way?

careful if looking
for 20 or 30!

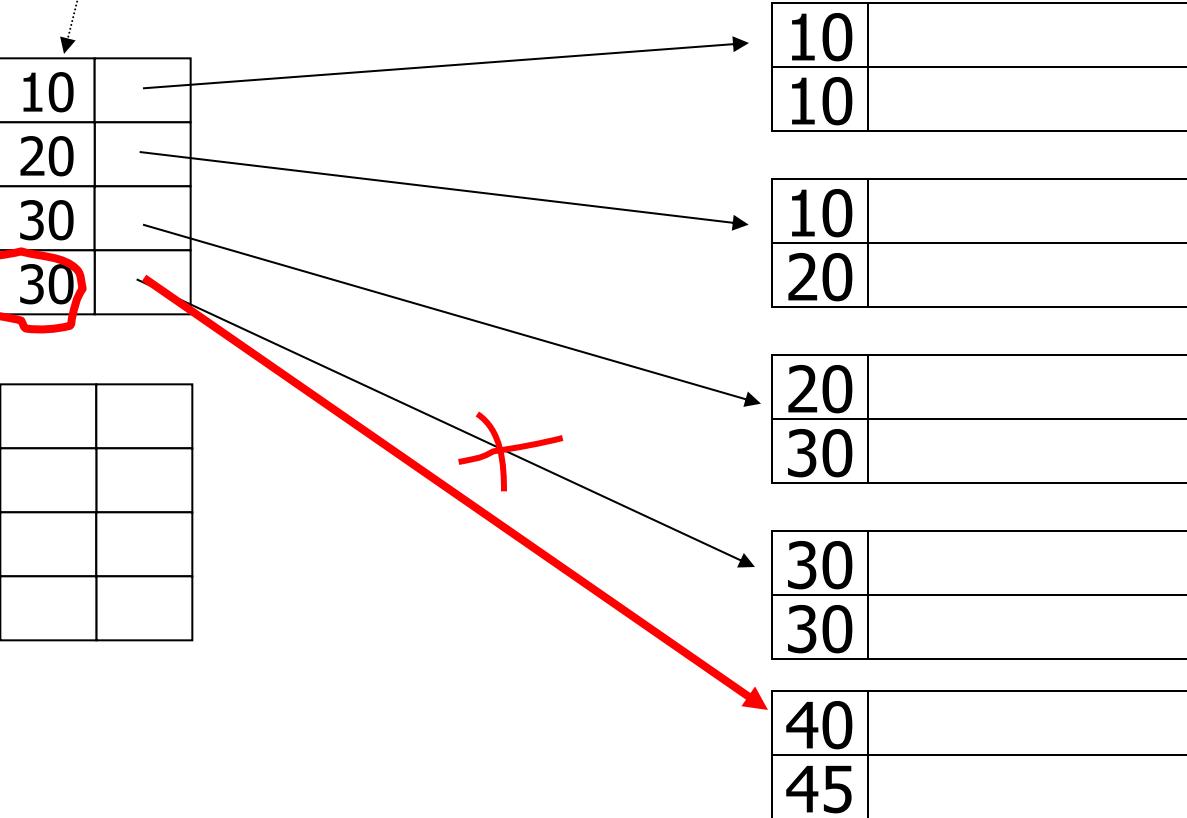


Sparse index, another way?

- place first new key from block

should
this be
40?

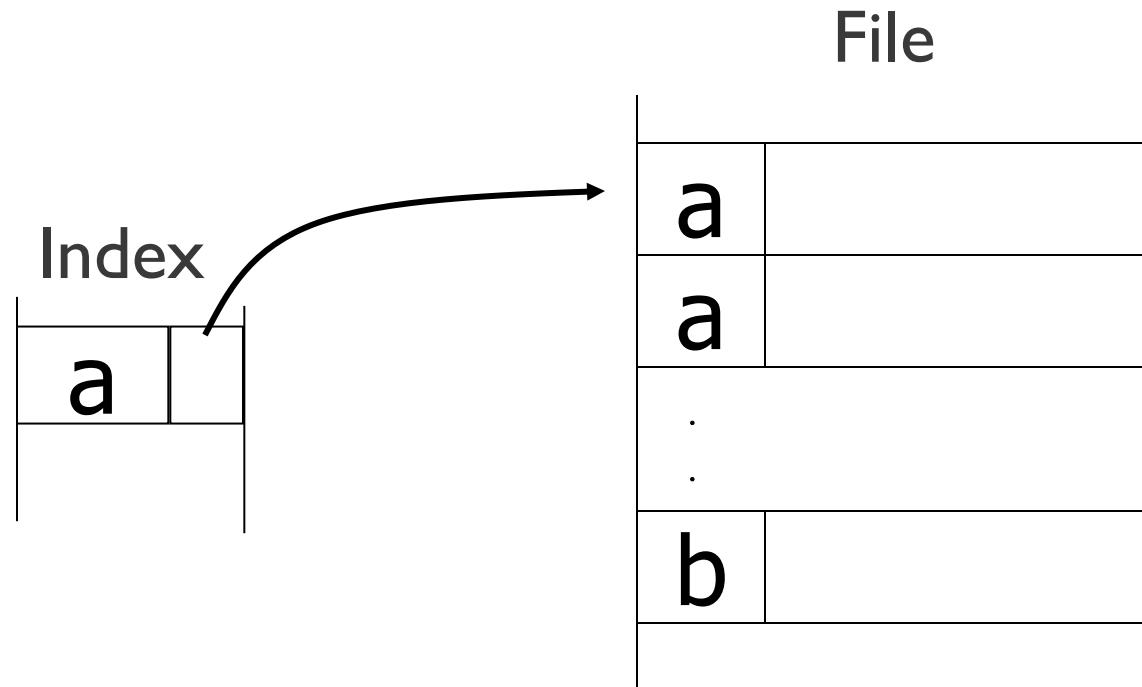
10	
20	
30	
30	



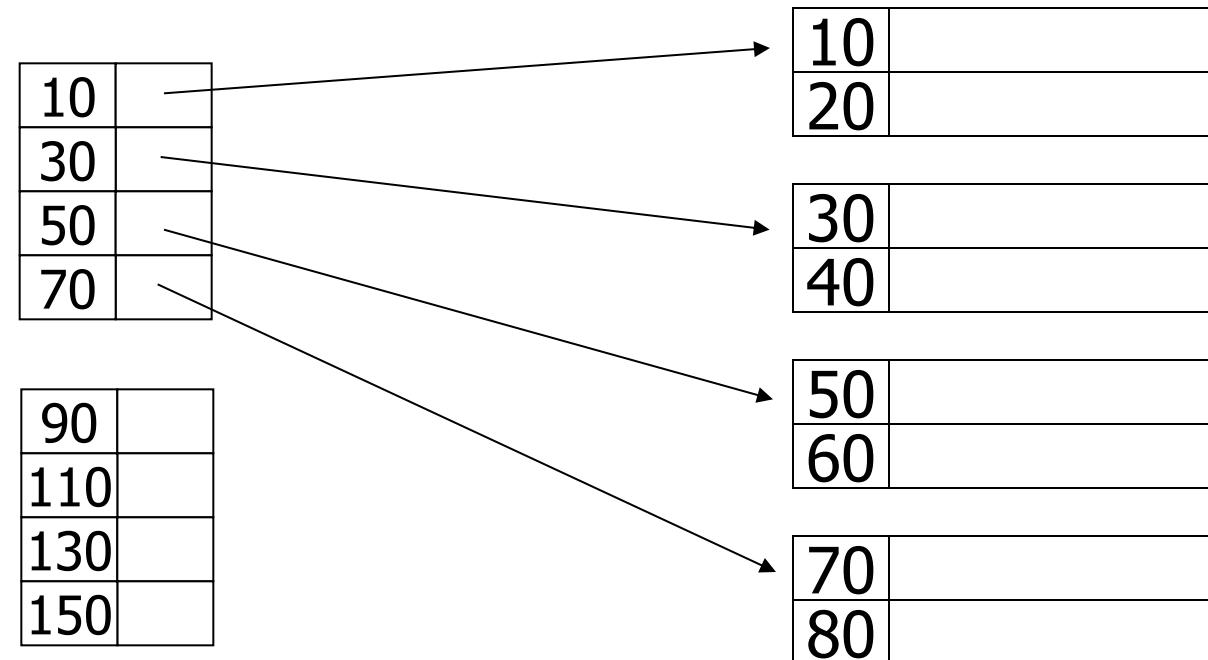
Duplicate values, primary index

Summary

- ▶ Index may point to first instance of each value only

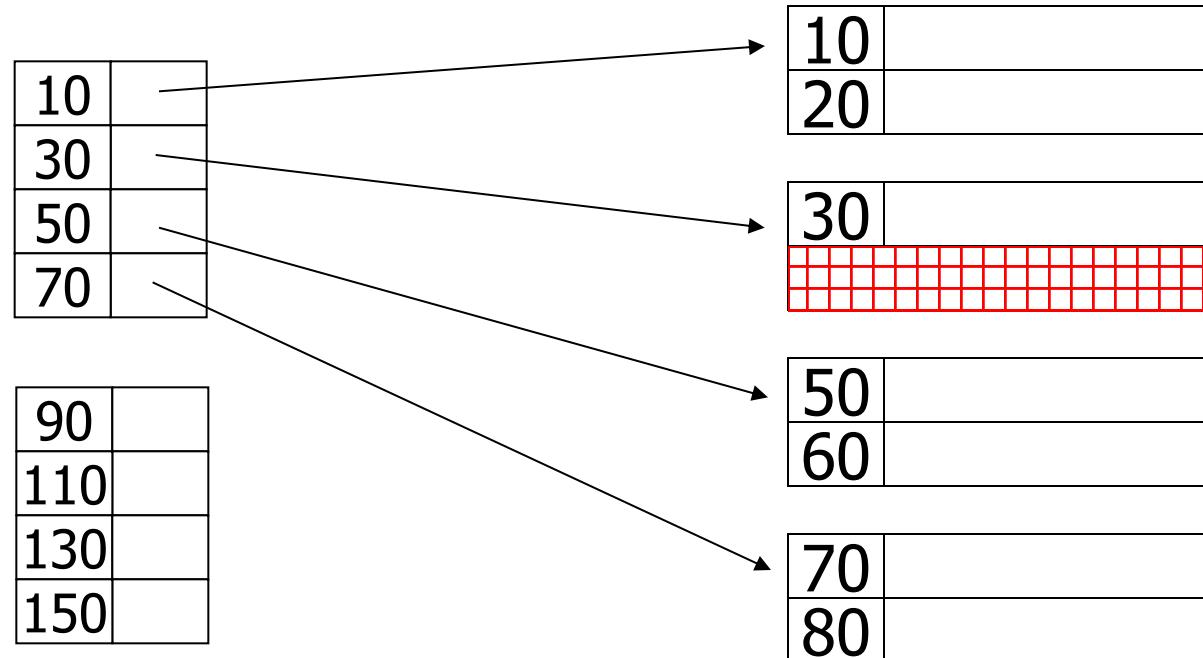


Deletion from sparse index



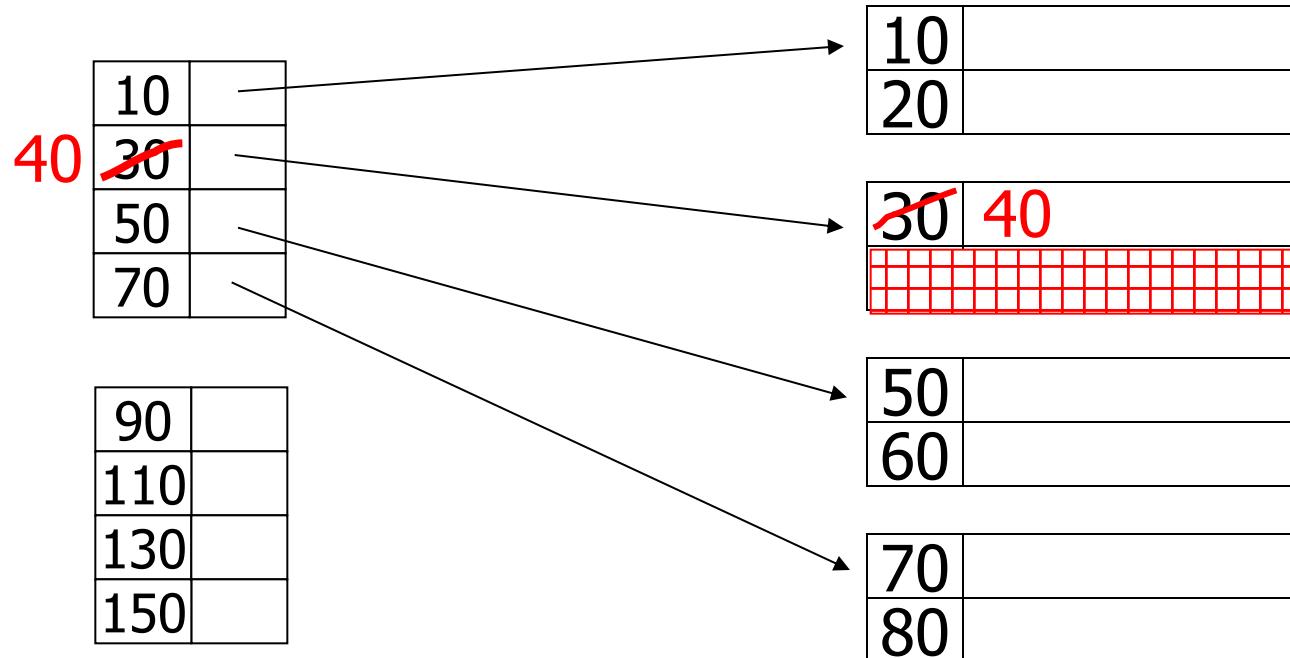
Deletion from sparse index

– delete record 40



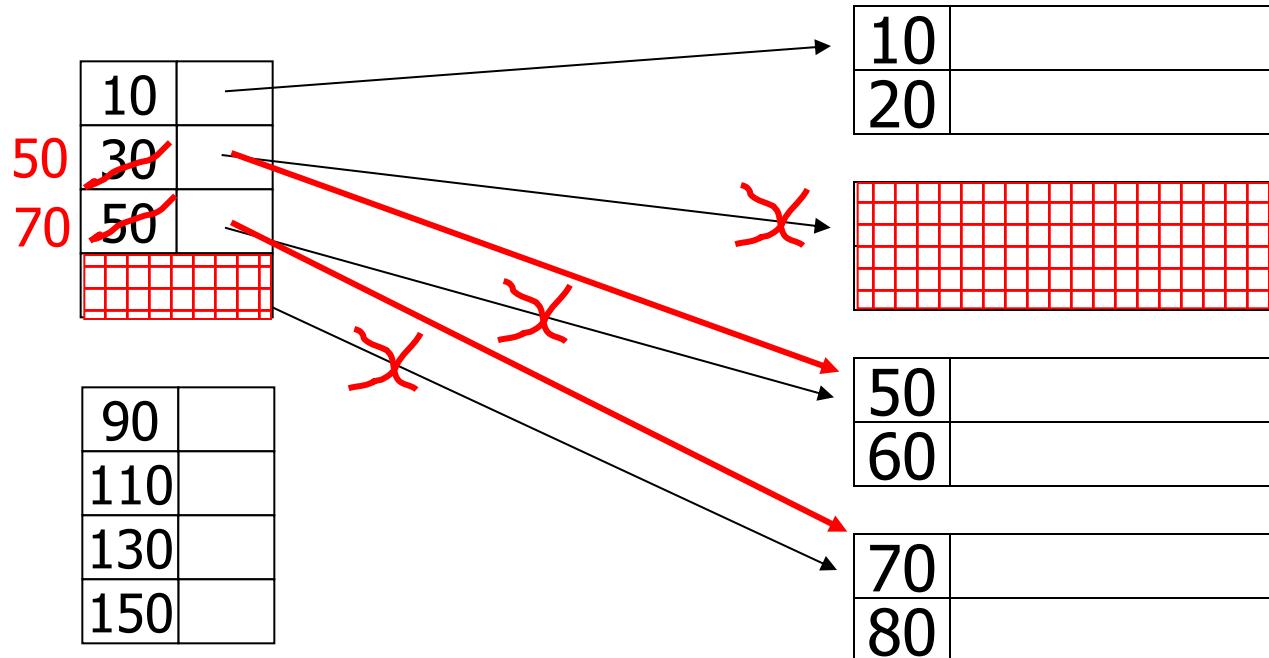
Deletion from sparse index

– delete record 30

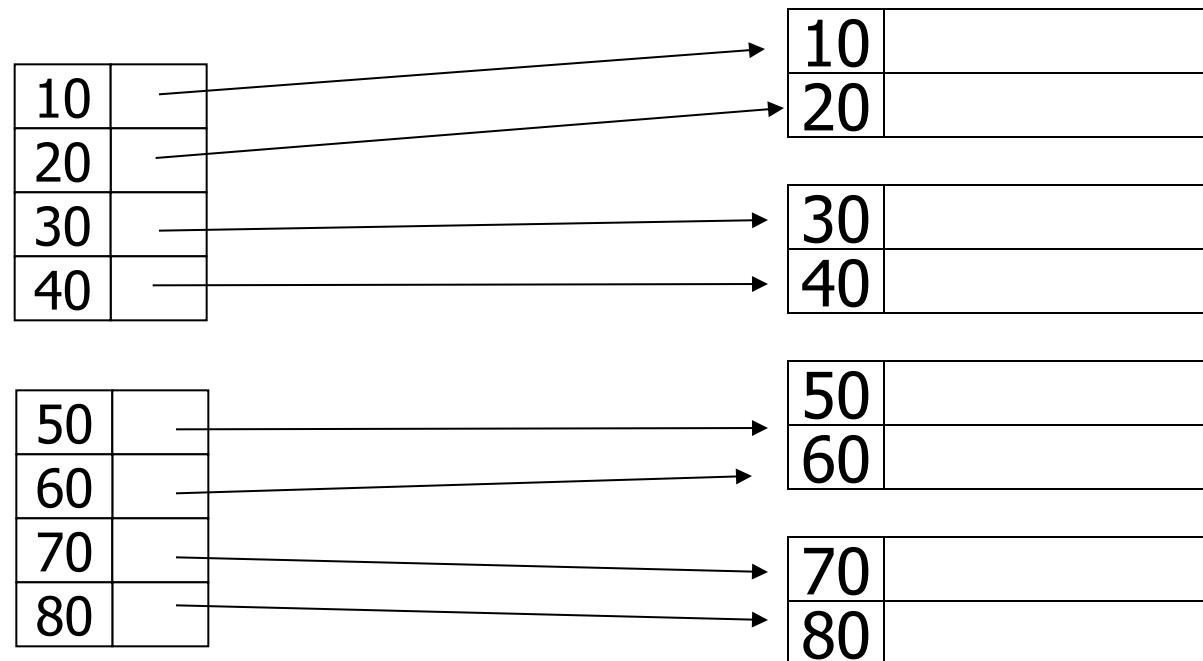


Deletion from sparse index

– delete records 30 & 40

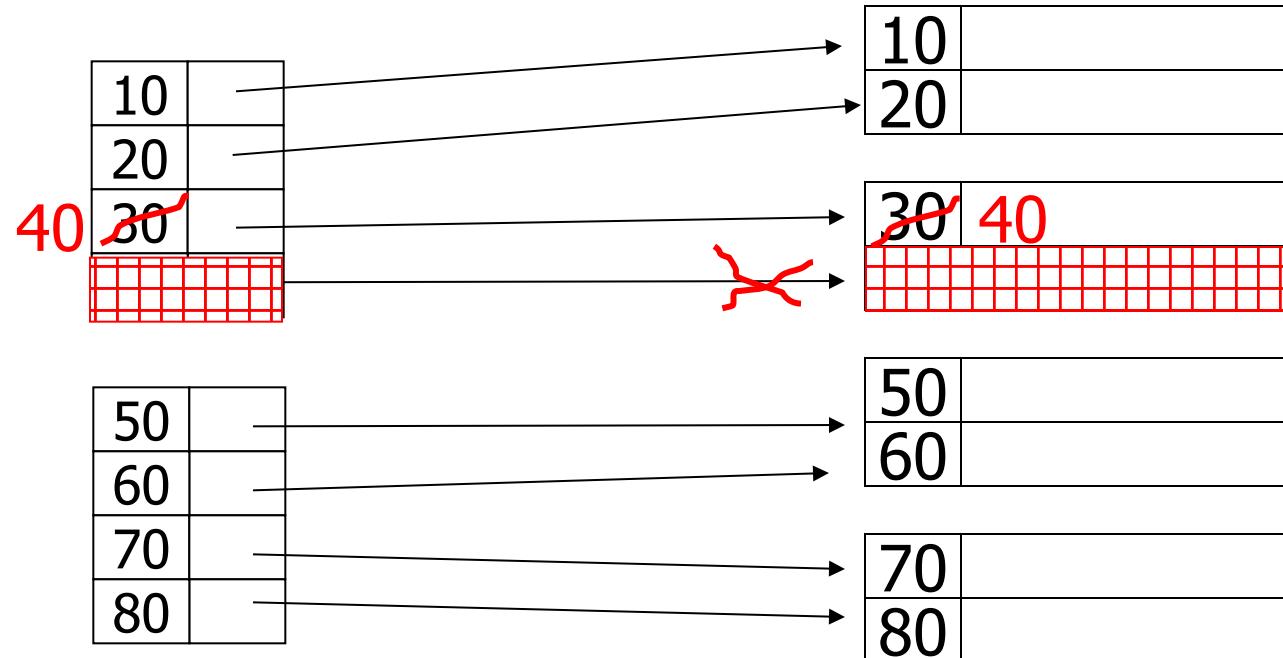


Deletion from dense index

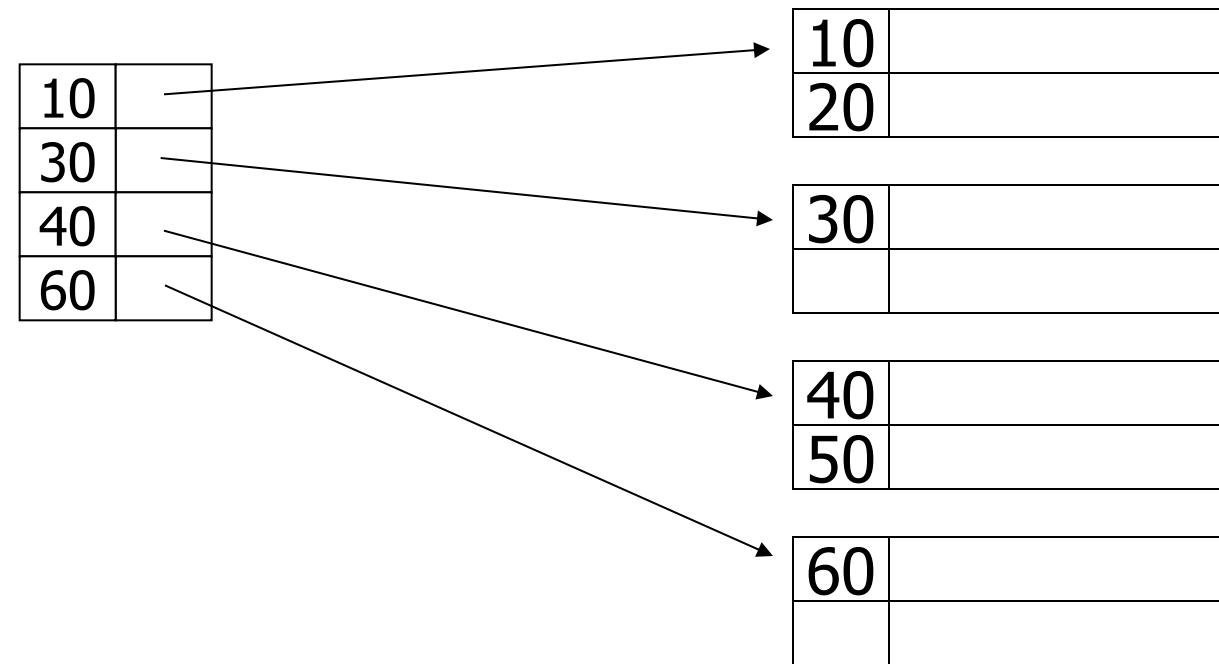


Deletion from dense index

– delete record 30

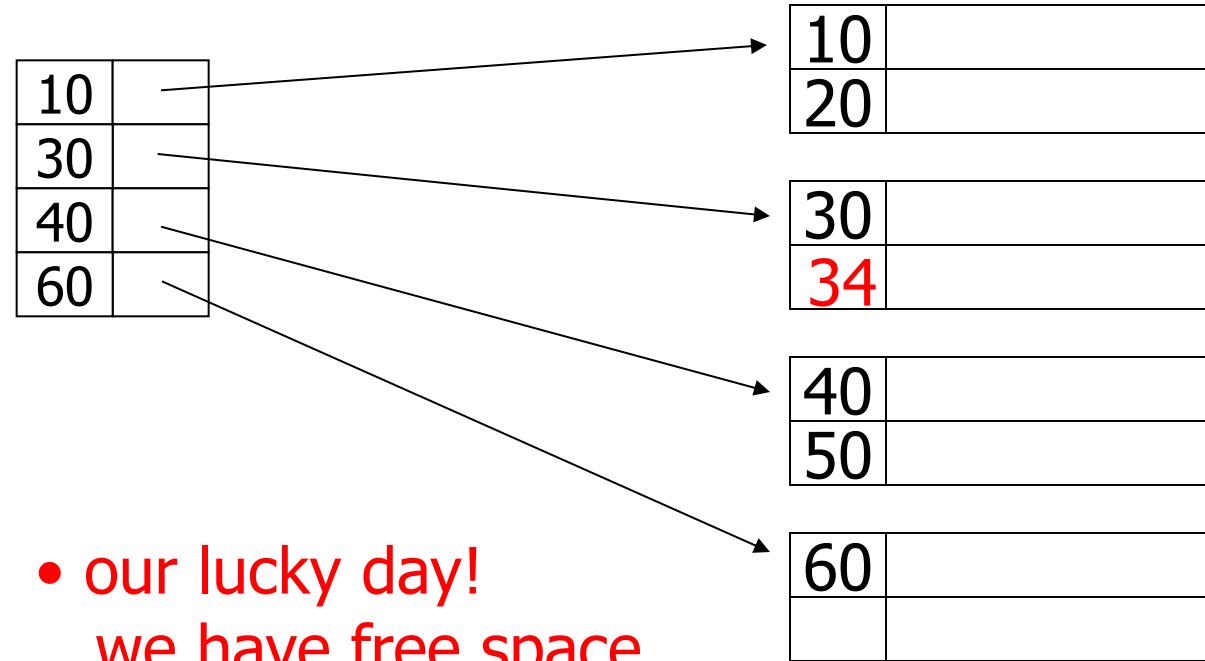


Insertion, sparse index case



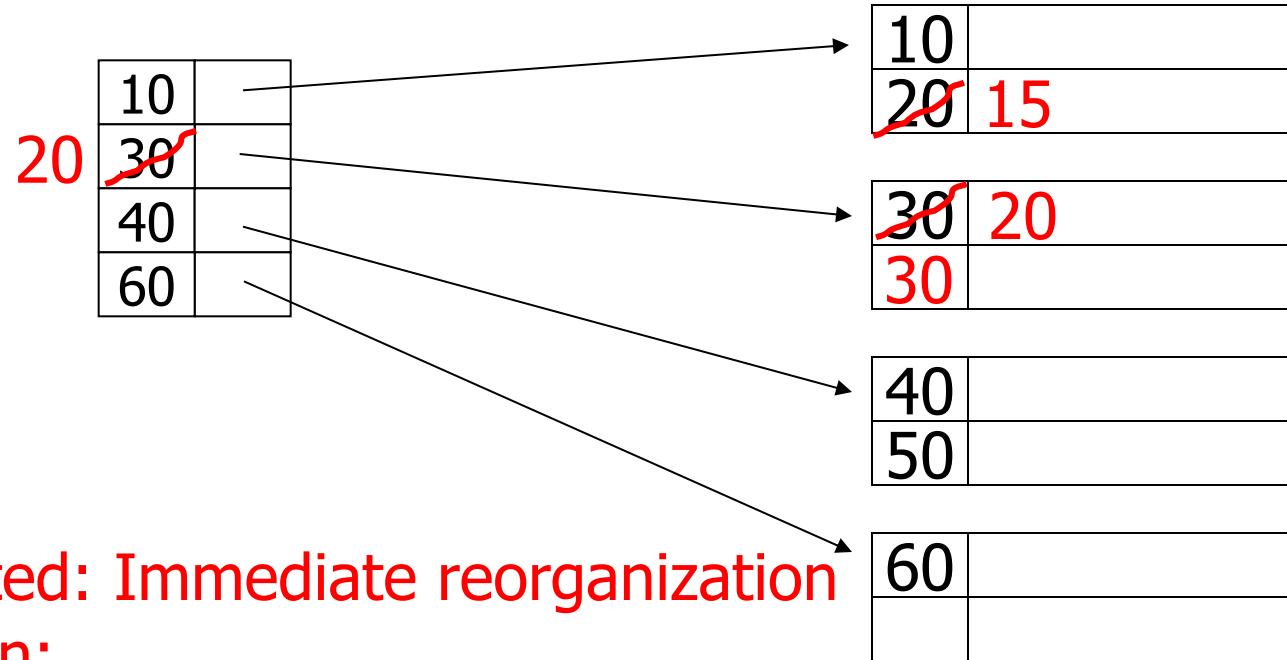
Insertion, sparse index case

– insert record 34



Insertion, sparse index case

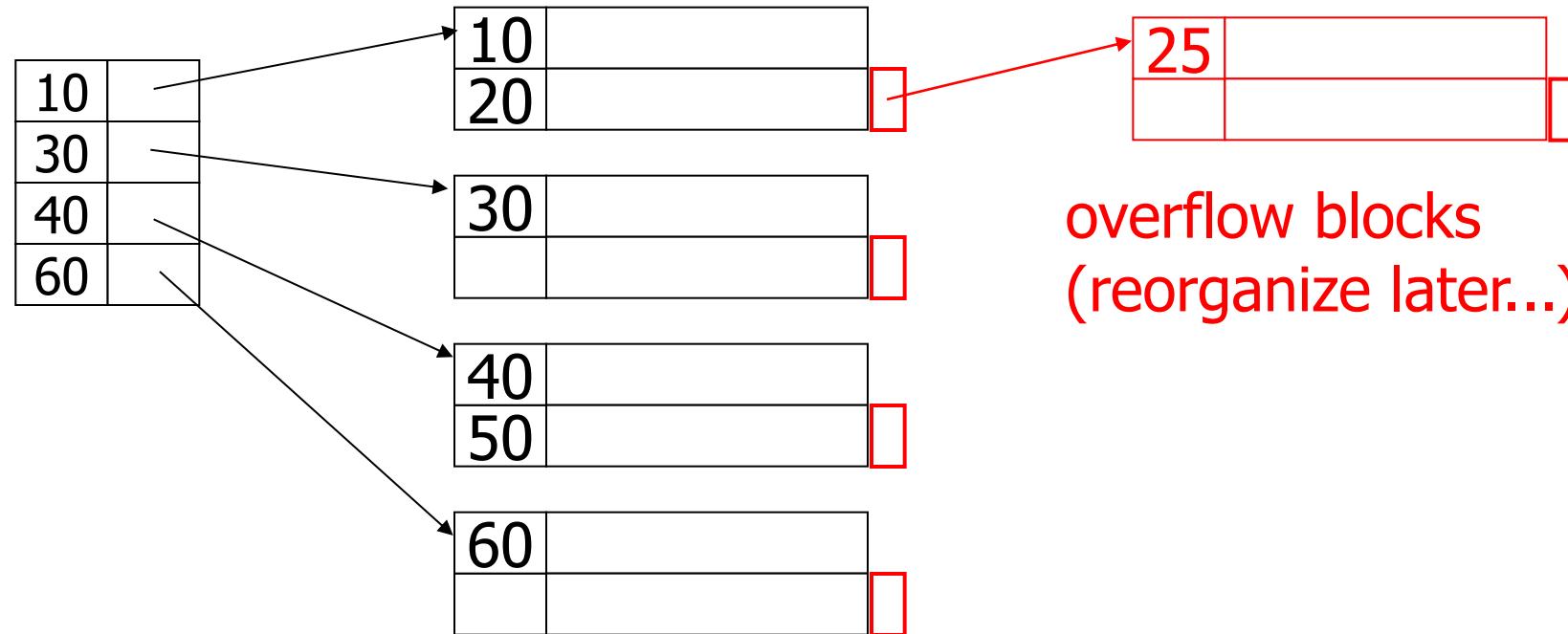
– insert record 15



- Illustrated: Immediate reorganization
- Variation:
 - insert new block (chained file)
 - update index

Insertion, sparse index case (alternate)

– insert record 25



Insertion, dense index case

- ▶ Similar
- ▶ Often more expensive ...

Continues

Sequence
field

30	
50	

20	
70	

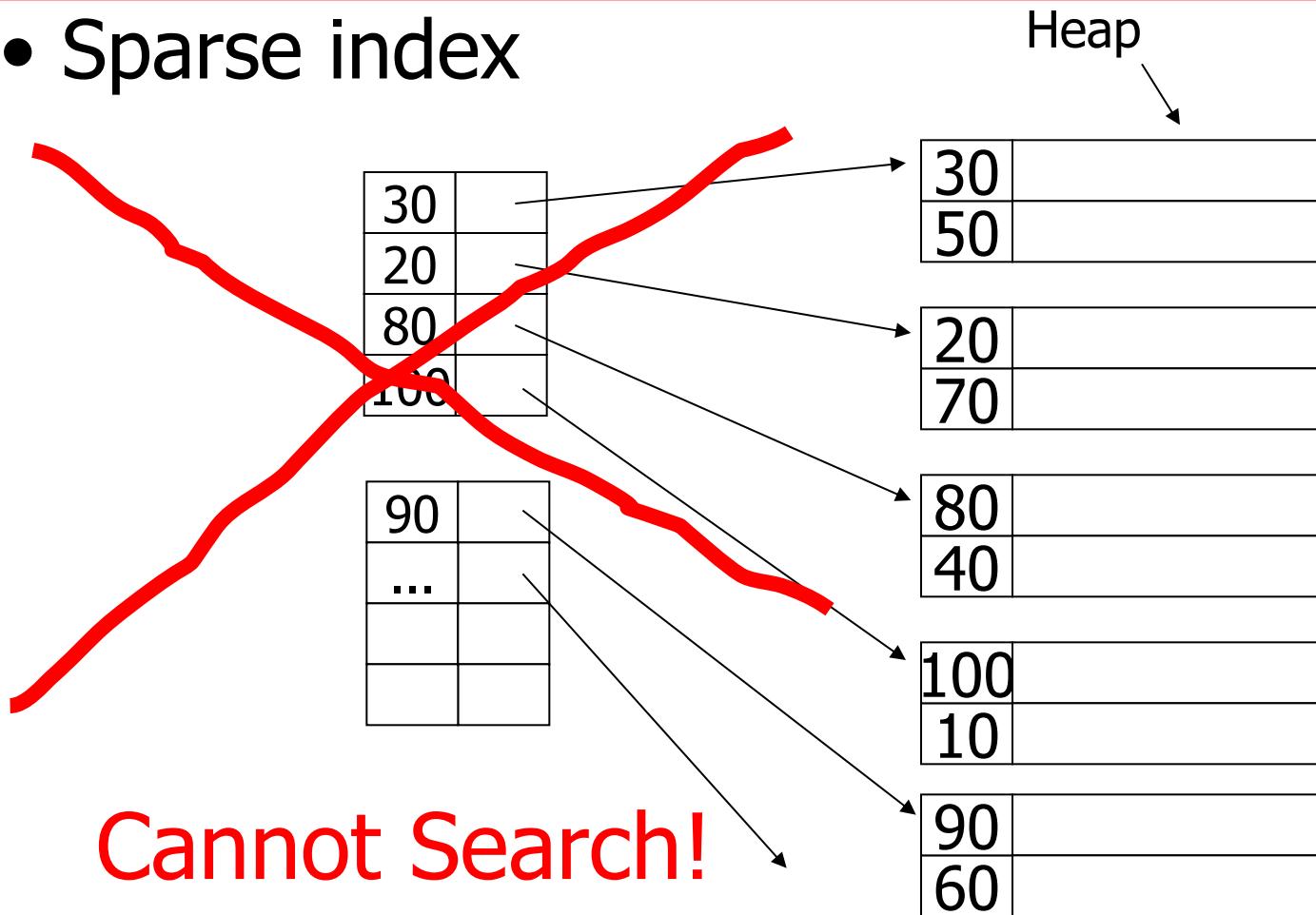
80	
40	

100	
10	

90	
60	

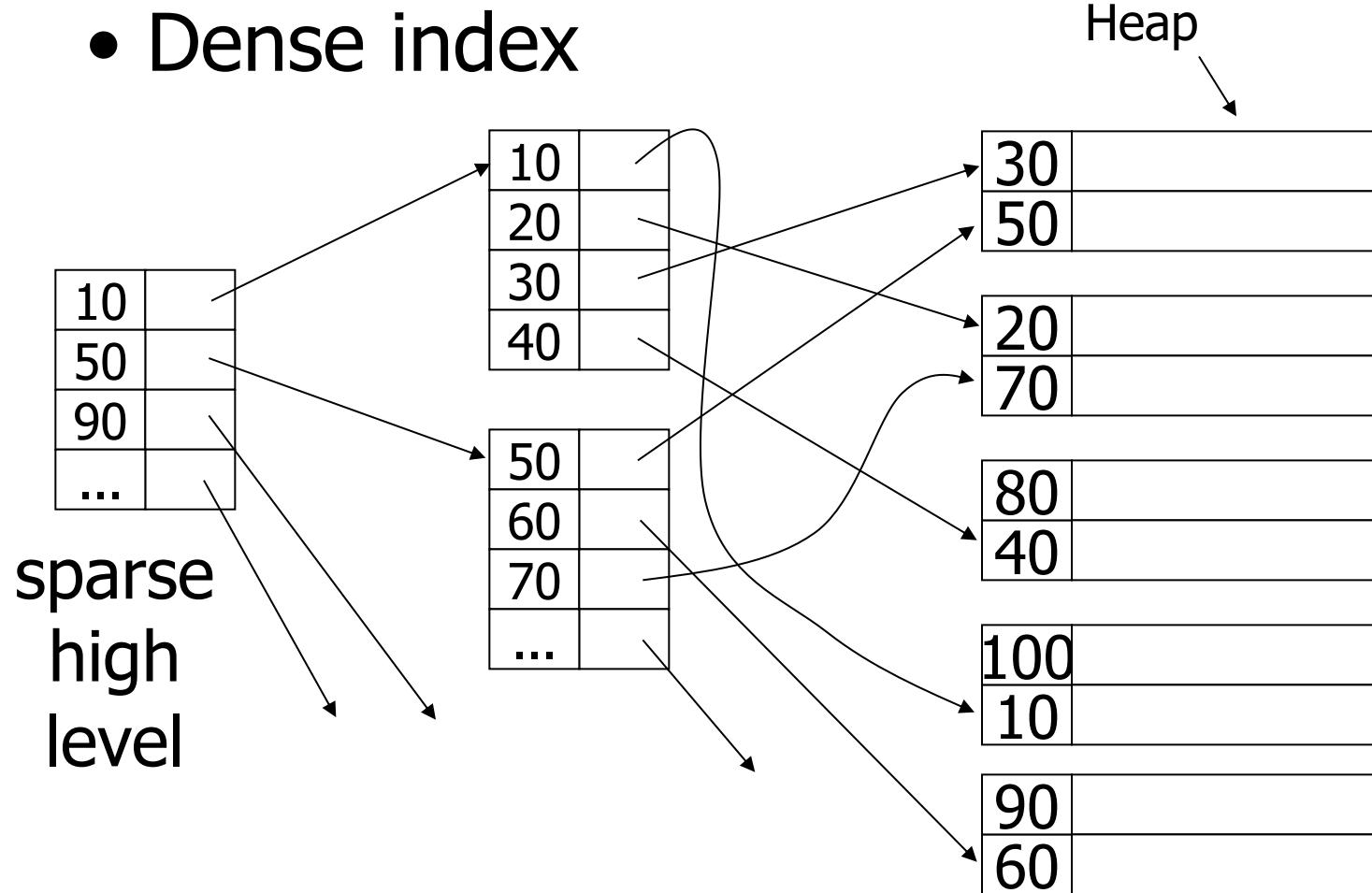
Sparse Index only for Sequential Data-file

- Sparse index



Multilevel indexes

- Dense index



With multilevel indexes:

- ▶ Sparse index; Lower-level should be sorted
 - ▶ Data-file is sorted, we can use sparse index
 - ▶ Sparse index on top of sparse even if data-file is Unordered!
- ▶ Other levels are sparse
 - ▶ Why not Dense??

Also: Pointers are record pointers

(not block pointers; not computed)

Duplicate values & secondary indexes

20	
10	

20	
40	

10	
40	

10	
40	

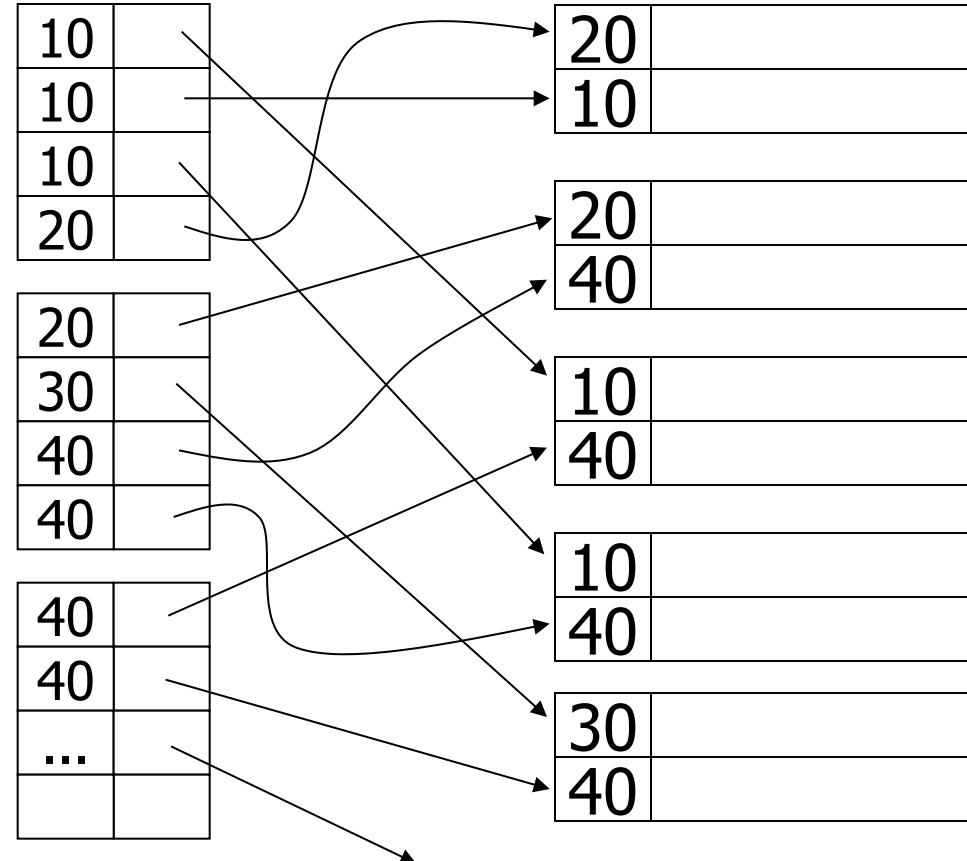
30	
40	

Duplicate values & secondary indexes

one option...

Problem:
excess overhead!

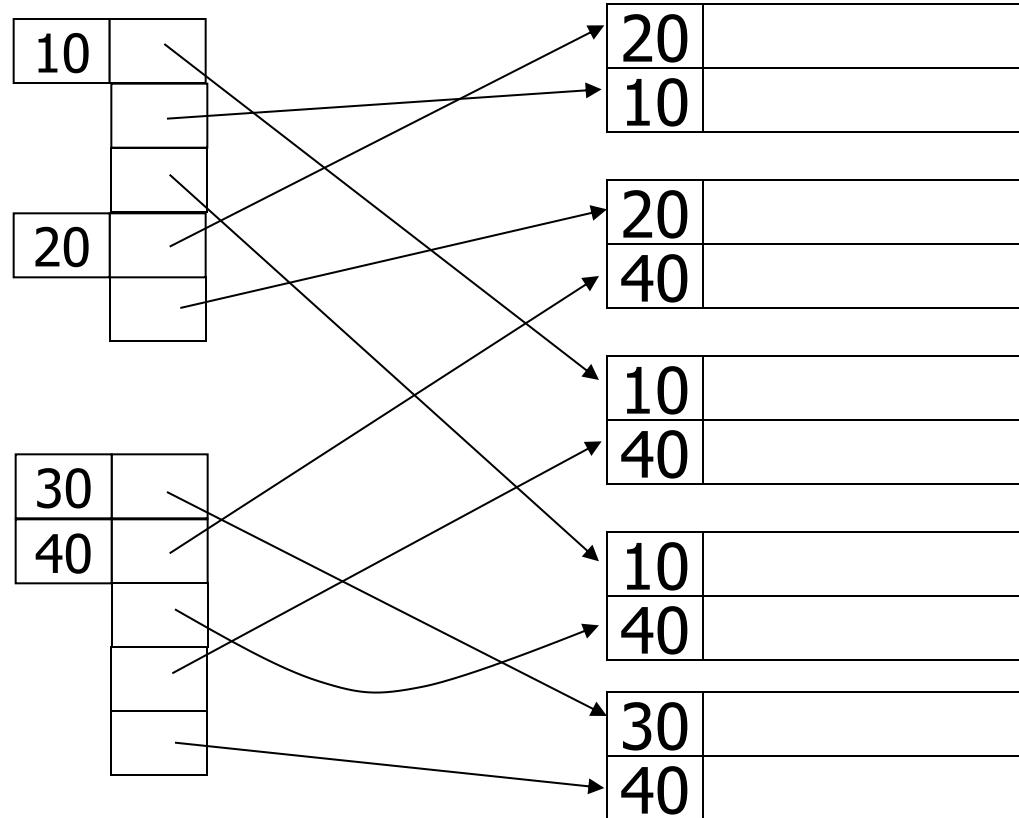
- disk space
- search time



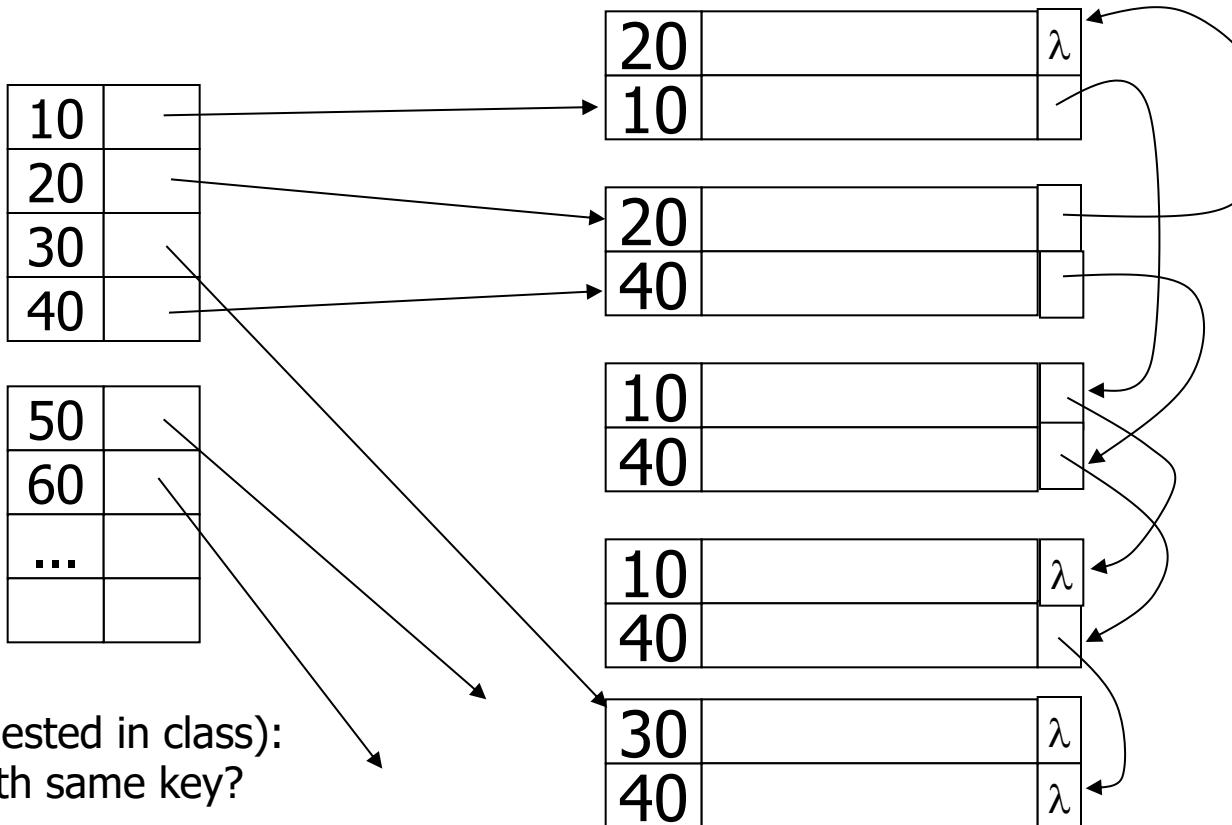
Duplicate values & secondary indexes

another option...

Problem:
variable size
records in
index!



Duplicate values & secondary indexes

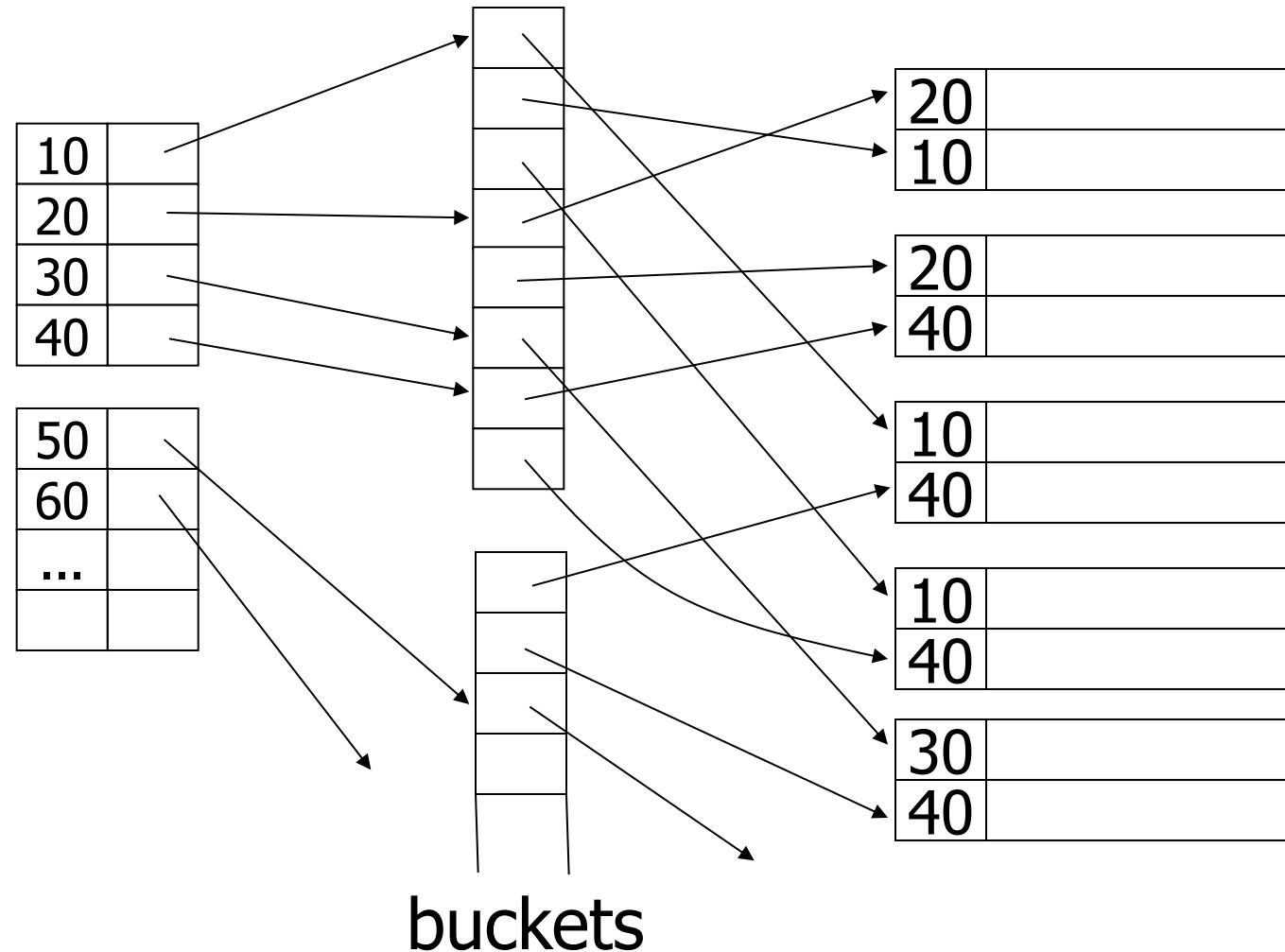


Another idea (suggested in class):
Chain records with same key?

Problems:

- Need to add fields to records
- Need to follow chain to know records

Duplicate values & secondary indexes



Example (Question)

- ▶ Suppose there is a data file of 4 GB (2^{32}) in a system with blocks of 1KB and fixed length records of 256Bytes.
- ▶ The records are stored in sorted order with respect to the key Student ID.
- ▶ Index stores a search key of 4 Bytes and a 4 Bytes of pointer. So, an index entry is 8 bytes.
- ▶ How many disk accesses do we need to find a record with a given Student ID:
 - ▶ Using sorted data file
 - ▶ Using dense index
 - ▶ Using sparse index

Example

- ▶ $2^{32} / 2^{10} = 2^{22}$ blocks are in the data file.
- ▶ Blocking Factor of data file = $2^{10}/2^8 = 2^2$
- ▶ $2^{22} \times \text{BF of data file} = 2^{24}$ records in the file
- ▶ With binary search we can have 22 disk accesses to find the record we are searching for in the worst case.
- ▶ If an index entry is 8 bytes we can fit into a block $2^{10}/2^3=2^7$ entries
 - ▶ Dense Index should be: $2^{24}/ 2^7=2^{17}$ blocks = $2^{17} \times 2^{10} = 2^{27} = 128\text{MB}$ index file. So, a binary search is 17 disk accesses.
 - ▶ Sparse Index should be: $2^{22}/2^7=2^{15}$ blocks = $2^{15} \times 2^{10} = 2^{25} = 32\text{MB}$ index file. So, a search is 15 disk accesses.
- ▶ What would happen if we had a two-level sparse index?



BBM371- Data Management

Lecture 5: Index Files and Comparing Cost Models

Indexes

- ▶ An index on a file speeds up selections on the **search key fields** for the index.
 - ▶ Any subset of the fields of a relation can be the search key for an index on the relation.
 - ▶ **Search key** is **not** the same as **key** (minimal set of fields that uniquely identify a record in a relation).
- ▶ An index contains a collection of **data entries**, and supports efficient retrieval of all data entries k^* with a given key value **k**.
 - ▶ Given data entry k^* , we can find record with key **k** in at most one disk I/O. (Details soon ...)

Alternatives for Data Entry k^* in Index

- ▶ In a data entry k^* we can store:
 - ▶ Data record with key value k , or
 - ▶ $\langle k, \text{rid of data record with search key value } k \rangle$, or
 - ▶ $\langle k, \text{list of rids of data records with search key } k \rangle$
- ▶ Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value k .
 - ▶ Examples of indexing techniques: B+ trees, hash-based structures
 - ▶ Typically, index contains auxiliary information that directs searches to the desired data entries

Alternatives for Data Entries (Contd.)

► Alternative I:

- If this is used, index structure is a file organization for data records (instead of a Heap file or sorted file).
- At most one index on a given collection of data records can use Alternative I. (Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency.)
- If data records are very large, # of pages containing data entries is high. Implies size of auxiliary information in the index is also large, typically.

Alternatives for Data Entries (Contd.)

► Alternatives 2 and 3:

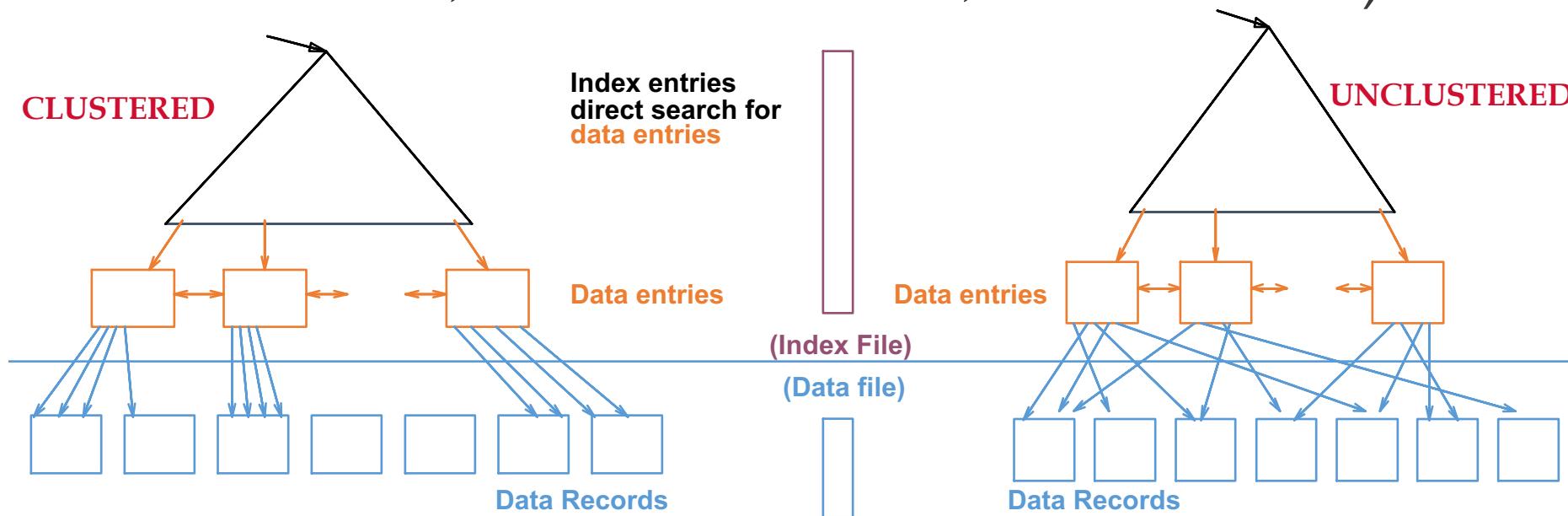
- Data entries typically much smaller than data records. So, better than Alternative 1 with large data records, especially if search keys are small. (Portion of index structure used to direct search, which depends on size of data entries, is much smaller than with Alternative 1.)
- Alternative 3 more compact than Alternative 2, but leads to variable sized data entries even if search keys are of fixed length.

Index Classification

- **Primary vs. secondary:** If search key contains primary key, then called primary index.
 - **Unique** index: Search key contains a candidate key.
- **Clustered vs. unclustered:** If order of data records is the same as, or 'close to', order of data entries, then called clustered index.
 - Alternative I implies clustered; in practice, clustered also implies Alternative I (since sorted files are rare).
 - A file can be clustered on at most one search key.
 - Cost of retrieving data records through index varies greatly based on whether index is clustered or not!

Clustered vs. Unclustered Index

- ▶ Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.
 - ▶ To build clustered index, first sort the Heap file (with some free space on each page for future inserts).
 - ▶ Overflow pages may be needed for inserts. (Thus, order of data recs is 'close to', but not identical to, the sort order.)



Comparing Storage Techniques

Cost Model for Our Analysis

We ignore CPU costs, for simplicity:

- ▶ **B**: The number of data pages
- ▶ **R**: Number of records per page
- ▶ **D**: (Average) time to read or write disk page
- ▶ Measuring number of page I/O's ignores gains of pre-fetching a sequence of pages; thus, even I/O cost is only approximated.
- ▶ Average-case analysis; based on several simplistic assumptions.

→ *Good enough to show the overall trends!*

Operations to Compare

- ▶ Scan: Fetch all records from disk
- ▶ Equality search
- ▶ Range selection
- ▶ Insert a record
- ▶ Delete a record

Assumptions in Our Analysis

- ▶ Single record insert and delete.
- ▶ Heap Files:
 - ▶ Equality selection on key; exactly one match.
 - ▶ Insert always at end of file.
- ▶ Sorted Files:
 - ▶ Files compacted after deletions.
 - ▶ Selections on sort field(s).
- ▶ Hashed Files:
 - ▶ No overflow buckets, 80% page occupancy.

Cost of Operations

	Heap File	Sorted File	Hashed File
Scan all recs	BD	BD	1.25 BD
Equality Search	0.5 BD	D $\log_2 B$	D
Range Search	BD	D ($\log_2 B + \# \text{ of pages with matches}$)	1.25 BD
Insert	2D	Search + BD	2D
Delete	Search + D	Search + BD	2D

Summary

- ▶ Many alternative file organizations exist, each appropriate in some situation.
- ▶ If selection queries are frequent, sorting the file or building an *index* is important.
- ▶ Index is a collection of data entries plus a way to quickly find entries with given key values.
- ▶ Data entries can be actual data records, $\langle \text{key}, \text{rid} \rangle$ pairs, or $\langle \text{key}, \text{rid-list} \rangle$ pairs.
 - ▶ Choice orthogonal to *indexing technique* used to locate data entries with a given key value.

Summary (Contd.)

- ▶ Can have several indexes on a given file of data records, each with a different search key.
- ▶ Indexes can be classified as clustered vs. unclustered, primary vs. secondary, and dense vs. sparse. Differences have important consequences for utility/performance.
- ▶ Understanding the nature of the *workload* for the application, and the performance goals, is essential to developing a good design.
 - ▶ What are the important queries and updates? What attributes/relations are involved?

Summary (Contd.)

- ▶ Indexes must be chosen to speed up important queries (and perhaps some updates!).
 - ▶ Index maintenance overhead on updates to key fields.
 - ▶ Choose indexes that can help many queries, if possible.
 - ▶ Build indexes to support index-only strategies.
 - ▶ Clustering is an important decision; only one index on a given relation can be clustered!
 - ▶ Order of fields in composite index key can be important.



BBM371- Data Management

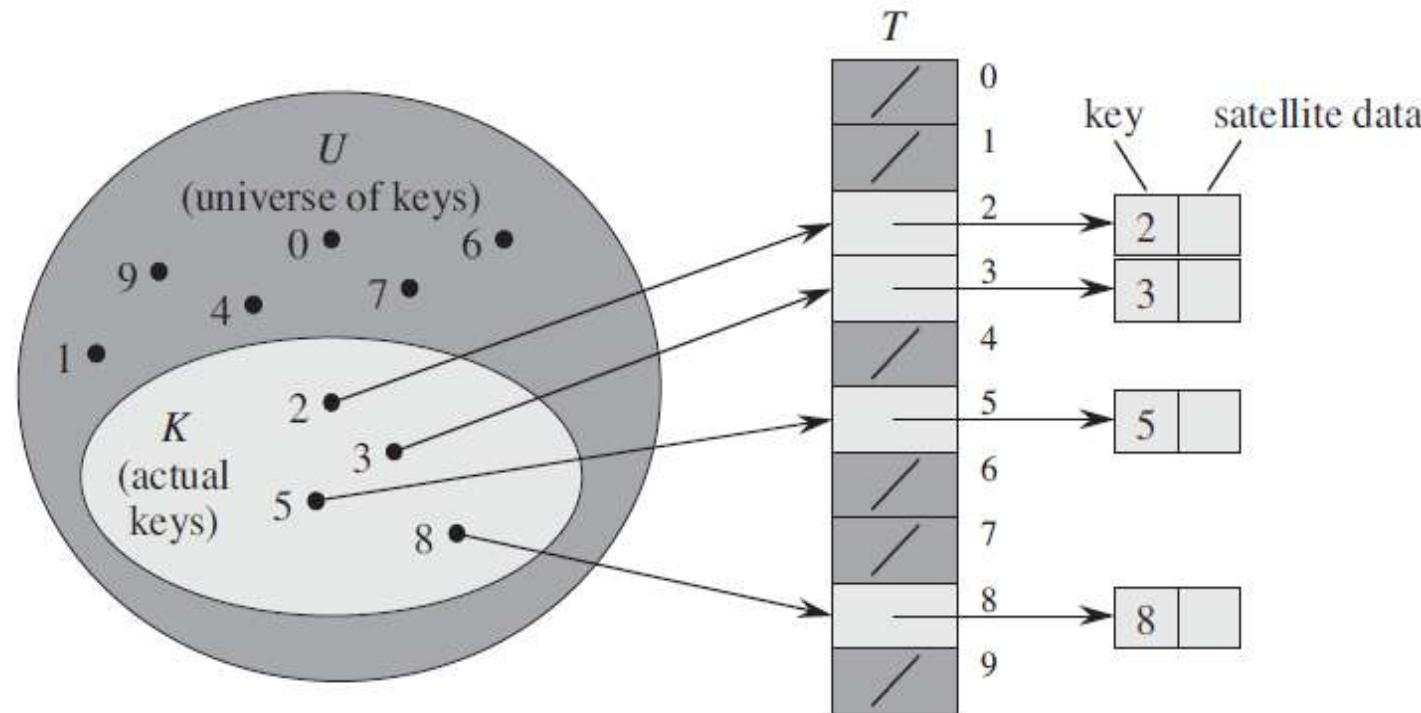
Lecture 6: Hash Tables

8.11.2018

Purpose of using hashes

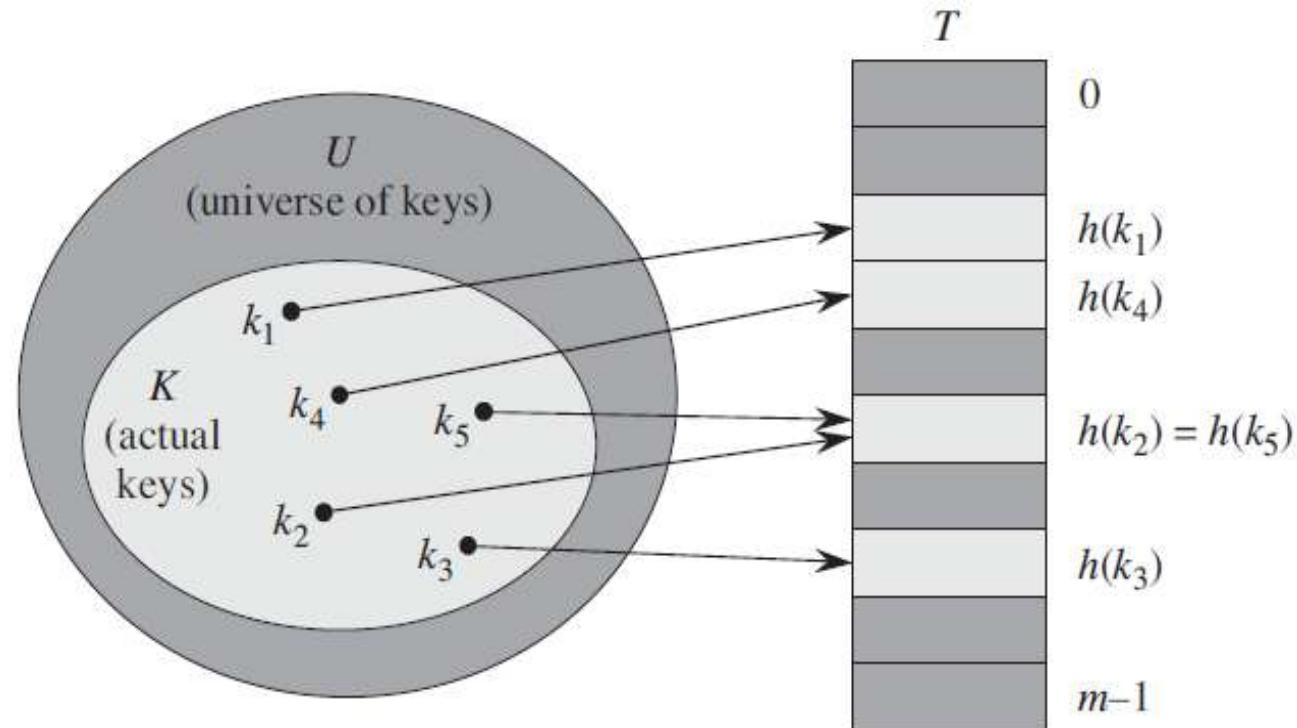
- A generalization of ordinary arrays:

- Direct access to an array index is $O(1)$, can we generalize direct access to any key
- If U is small (e.g. 9) we can do it with a table T .
 - Think of ASCII table to map characters to arrays.



Hash tables

- ▶ Used when the set of potential keys U is large and actual used keys K is small
 - ▶ Consider storing 5 keys from potential keys between numbers 0 and 1 billion



Hashing

- ▶ An element with key k is transformed with a hash function to map to slot $h(k)$ in hash table T .
- ▶ $h(k)$ is the hash value of key k
- ▶ Hash function reduces the range of key values from $|U|$ to $|T|$

Note: Now let's think address as record number in a file (or table)

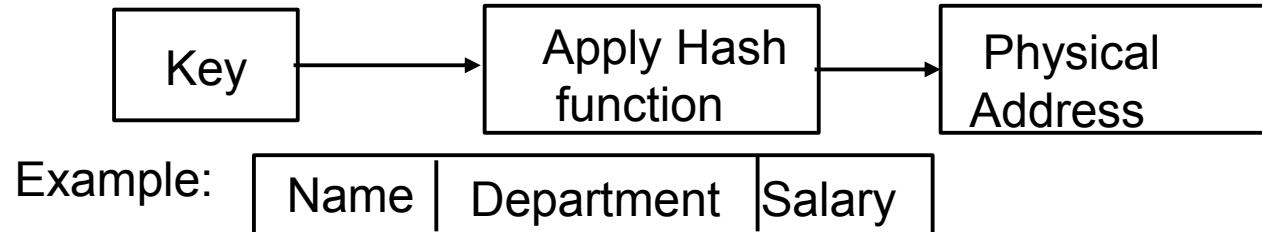
Example

- ▶ Hash function : Convert an arbitrary string key in ASCII format and multiply first two character and use rightmost three digit.

$$H(\text{Lowell}) = 4$$

- ▶ Can consider all possible strings as infinite number of keys, we map it to small range.

Hashing with Files



$h(K) = K \bmod m$
 $m = 70 - 90\%$ of
the expected number
of records

$$h(\text{James Adams}) = (74+65) \bmod 17 = 139 \bmod 17 = 3$$

	Name	Department	Salary	Overflow Pointer
0				
1				
2				
3	James Adams			~1
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15	Mary Jones			~1
16				
17	Henry Truman			~1

Records are mapped
to blocks in a file!

Collisions

- ▶ Larger set U is mapped to a small set T
- ▶ Having multiple-keys mapped to the same slot is called as collision.
- ▶ If we add n keys larger than the number of slots we have we must have at least one collision.
- ▶ Even if $n < |T|$ and mapping is random we can have collisions.
 - ▶ Think of birthday paradox; with more than 23 people in a room probability of having the same birthdate (day, month) is 50%

Good Hash Function

- ▶ Generated record numbers should be uniformly and randomly distributed over the file ($0 \leq h(key) < |T|$)
- ▶ The hashing function must minimize collisions (random distribution)
 - ▶ Worst case : Map all keys to slot 0. $O(|T|)$
- ▶ Should be easy to calculate

Load Factor

- ▶ Loading factor (LF), $\alpha = n / m$
n: number of keys
m: number of slots
- ▶ If uniform distribution ($1/m$) to get mapped to a slot, a slot will have an expectation of α elements.
- ▶ If m increases
 - ▶ Collision decreases
 - ▶ LF decreases
 - ▶ $0.5 > LF > 0.8$ is unacceptable
 - ▶ Storage requirements increases.
- ▶ Reduce collisions while keeping storage requirements low.

Hashing Transformations

- ▶ **Digit analysis**

Use specific digits from key; might not be random

- ▶ **Division method**

$$h_{ab}(k) = ((ak+b) \bmod p) \bmod m \quad p > m \text{ and } p \text{ is prime}$$

- ▶ **Radix transformation**

$$f(abc) = a * ||^2 + b * || + c$$

Overflow Management Techniques

- ▶ Direct organisation of file by using Hashing
 - ▶ Open Addressing
 - ▶ Linear search
 - ▶ Nonlinear search
 - ▶ Chaining
- ▶ Hash based indexing
 - ▶ Extendible Hashing
 - ▶ Linear Hashing

Open Addressing

- ▶ All elements occupy the hash table itself. (i.e. No pointers or overflow buckets)
- ▶ When collision occurs, the new record will be inserted in the first available slot after $h(k)$
- ▶ When searching for available slot, hash table is probed.
- ▶ Depending on $h(k)$ different probe sequences from all permutations of $0\dots m$ can be used for a slot.
- ▶ The probe sequence generation methods determine the performance of hash tables.

Insert Algorithm

HASH-INSERT(T, k)

```
1   $i = 0$ 
2  repeat
3       $j = h(k, i)$ 
4      if  $T[j] == \text{NIL}$ 
5           $T[j] = k$ 
6          return  $j$ 
7      else  $i = i + 1$ 
8  until  $i == m$ 
9  error “hash table overflow”
```

- ▶ $h(k, i)$: returns the i^{th} element of key k 's probe sequence.
- ▶ If all slots are probed ($i==m$), the hash table is full.

Search Algorithm

HASH-SEARCH(T, k)

```
1    $i = 0$ 
2   repeat
3        $j = h(k, i)$ 
4       if  $T[j] == k$ 
5           return  $j$ 
6        $i = i + 1$ 
7   until  $T[j] == \text{NIL}$  or  $i == m$ 
8   return NIL
```

- ▶ Similar code for search.
- ▶ If key is not encountered before a NIL value, key is not in T
 - ▶ In an edge case all nodes in probe sequence is filled, must check $i == m$

Linear Probing

$$h(k, i) = (h'(k) + i) \bmod m$$

- ▶ Always check the next index
- ▶ Increments index linearly with respect to i.
- ▶ Clustering problem

hash(10) = 2
hash(5) = 5
hash(15) = 7

0	72
1	
2	18
3	43
4	36
5	
6	6
7	

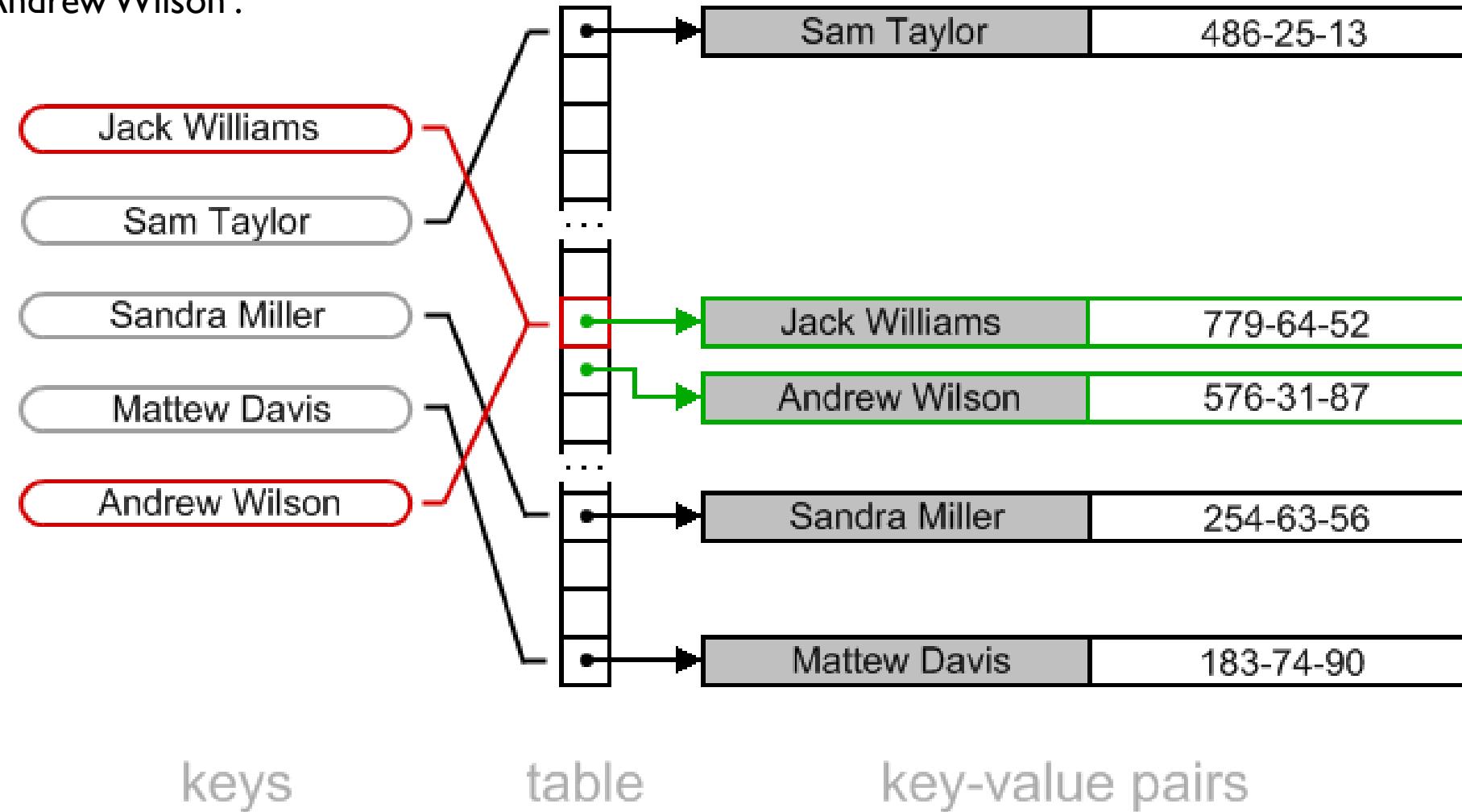
0	72
1	
2	18
3	43
4	36
5	10
6	6
7	

0	72
1	
2	18
3	43
4	36
5	10
6	6
7	5

0	72
1	15
2	18
3	43
4	36
5	10
6	6
7	5

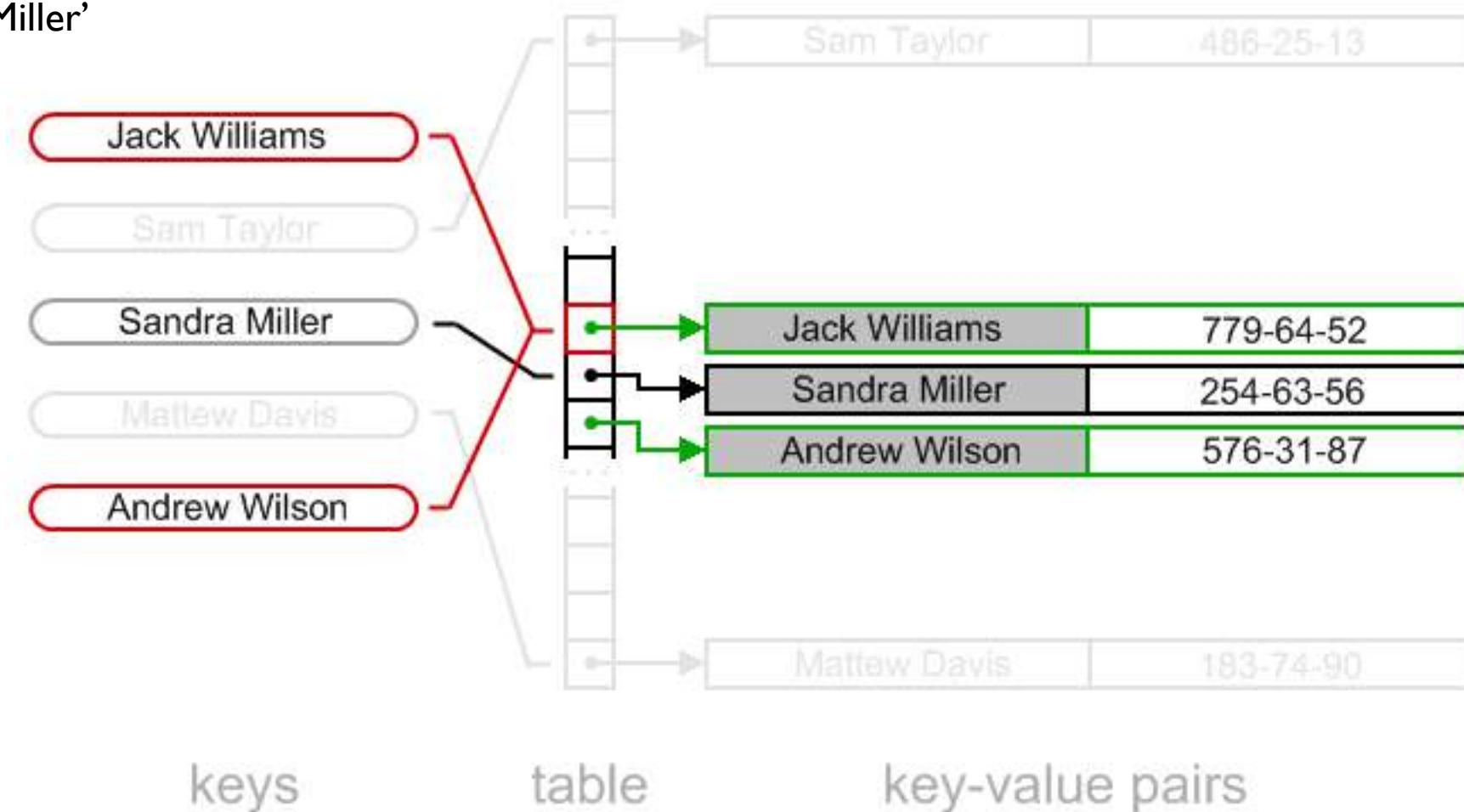
Linear Probing - insertion

Add 'Andrew Wilson':

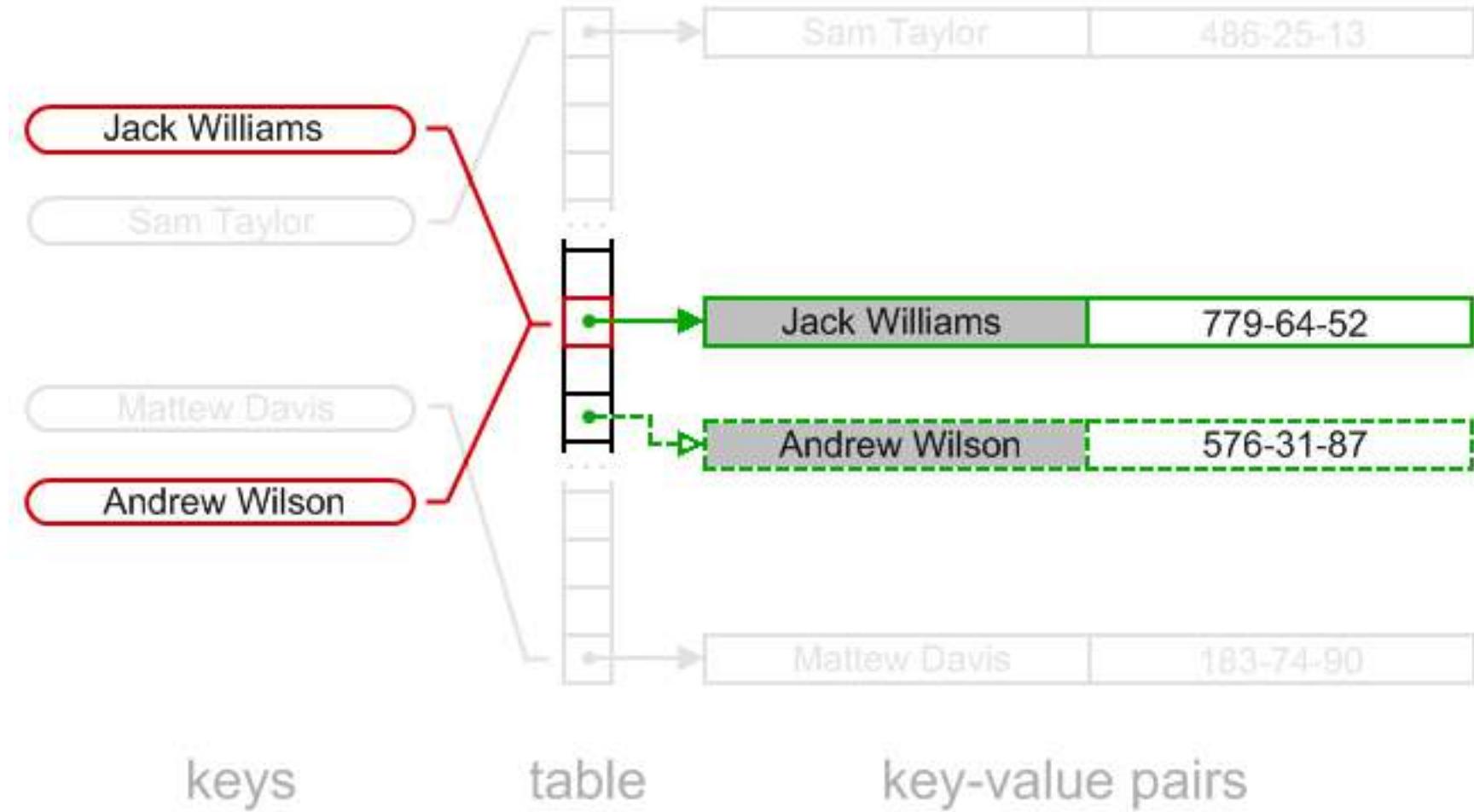


Linear search - deletion

Delete 'Sandra Miller'



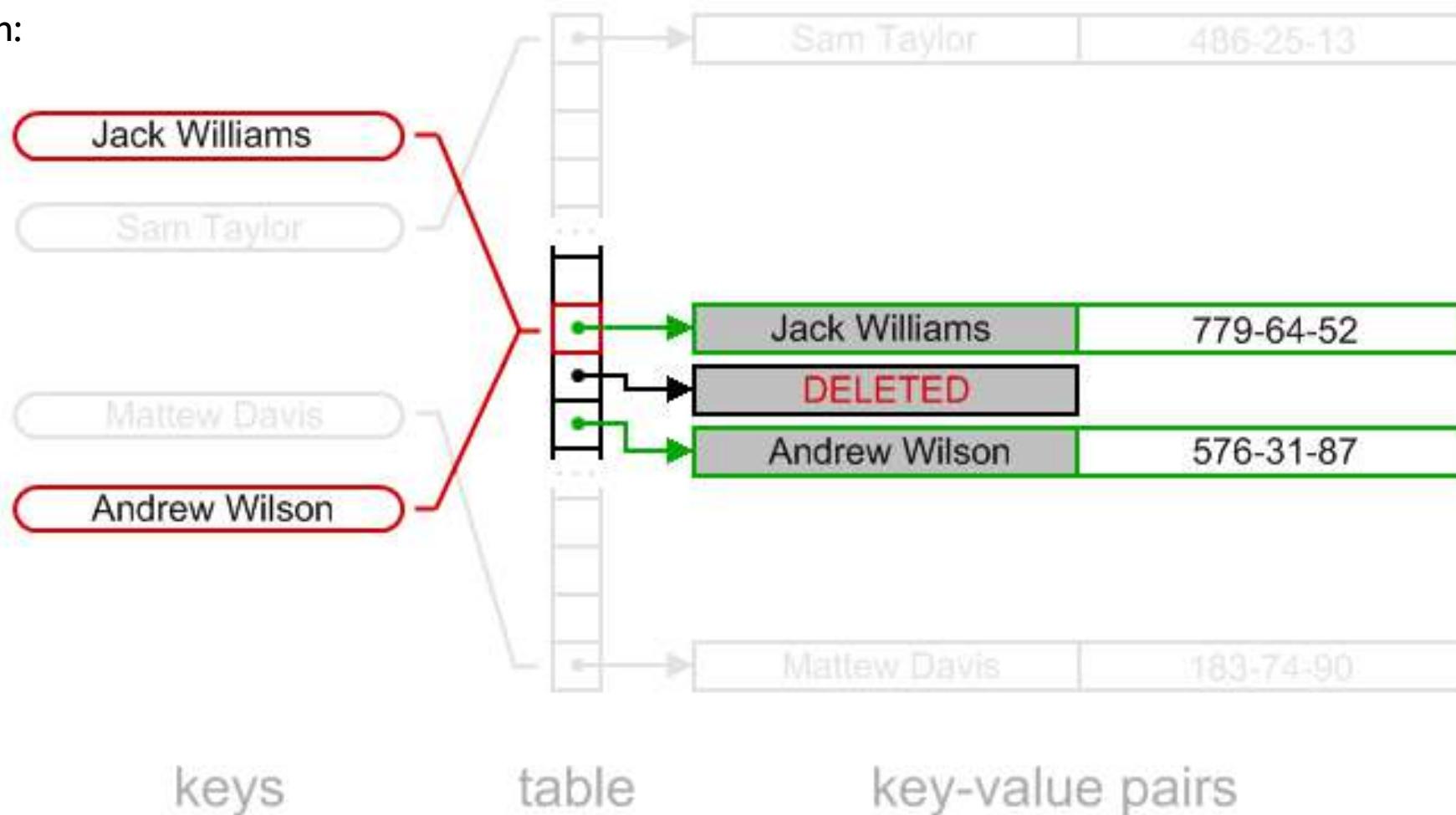
Linear search - deletion



How to find Andrew Wilson?

Linear search - deletion

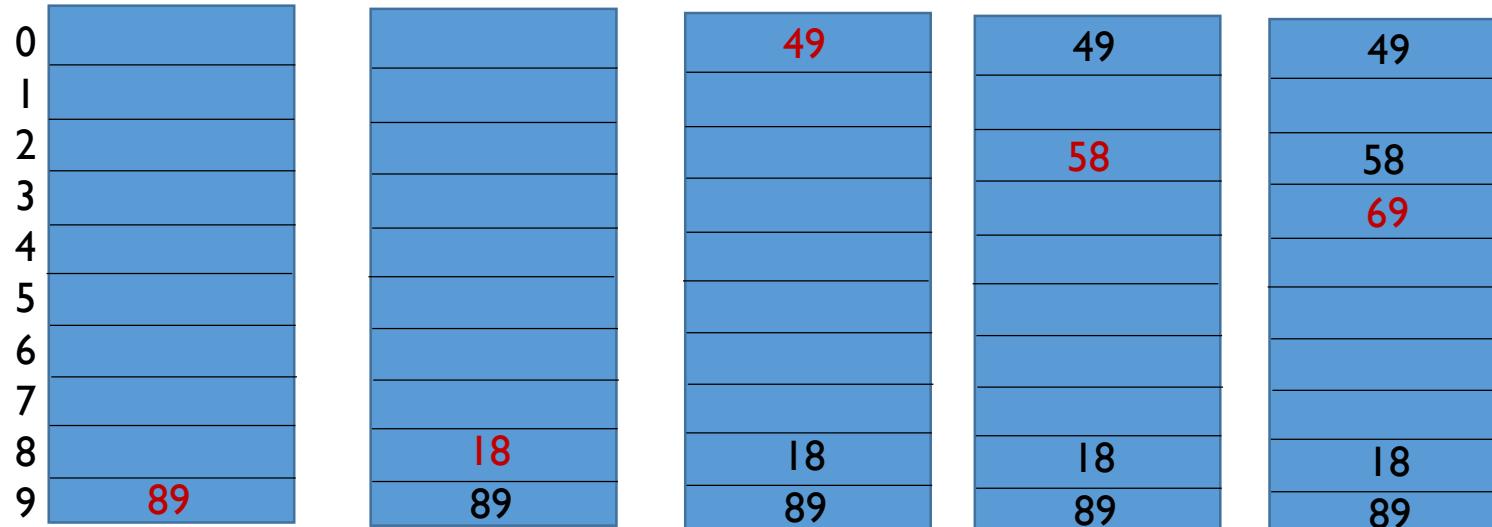
The solution:



Open Addressing – Quadratic Probing

$$h(k, i) = (h'(k) + c_1i + c_2i^2) \bmod m$$

- Instead of moving by one, move i^2



$c_1=0, c_2=1$
hash(89)=9
hash(18)=8
hash(49)=9
hash(49, 1) = 0
hash(58) = 8
hash(58, 1) = 9
hash(58,2) = 2
hash(69) = 9
hash(69,1) = 0
hash(69,2) = 3

Insert Algorithm – Double Hashing

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

- ▶ Two hash functions
 - ▶ h_1 to find the initial position
 - ▶ h_2 to find offset from initial position
- ▶ Different from linear and quadratic, two keys mapping to same slot can now use different offsets $h_2(k)$

Example 1 - Double Hashing

- Example:

- Table Size is 11 (0..10)

- Hash Function:

$$h_1(x) = x \bmod 11$$

$$h_2(x) = 7 - (x \bmod 7)$$

- Insert keys: 58, 14, 91

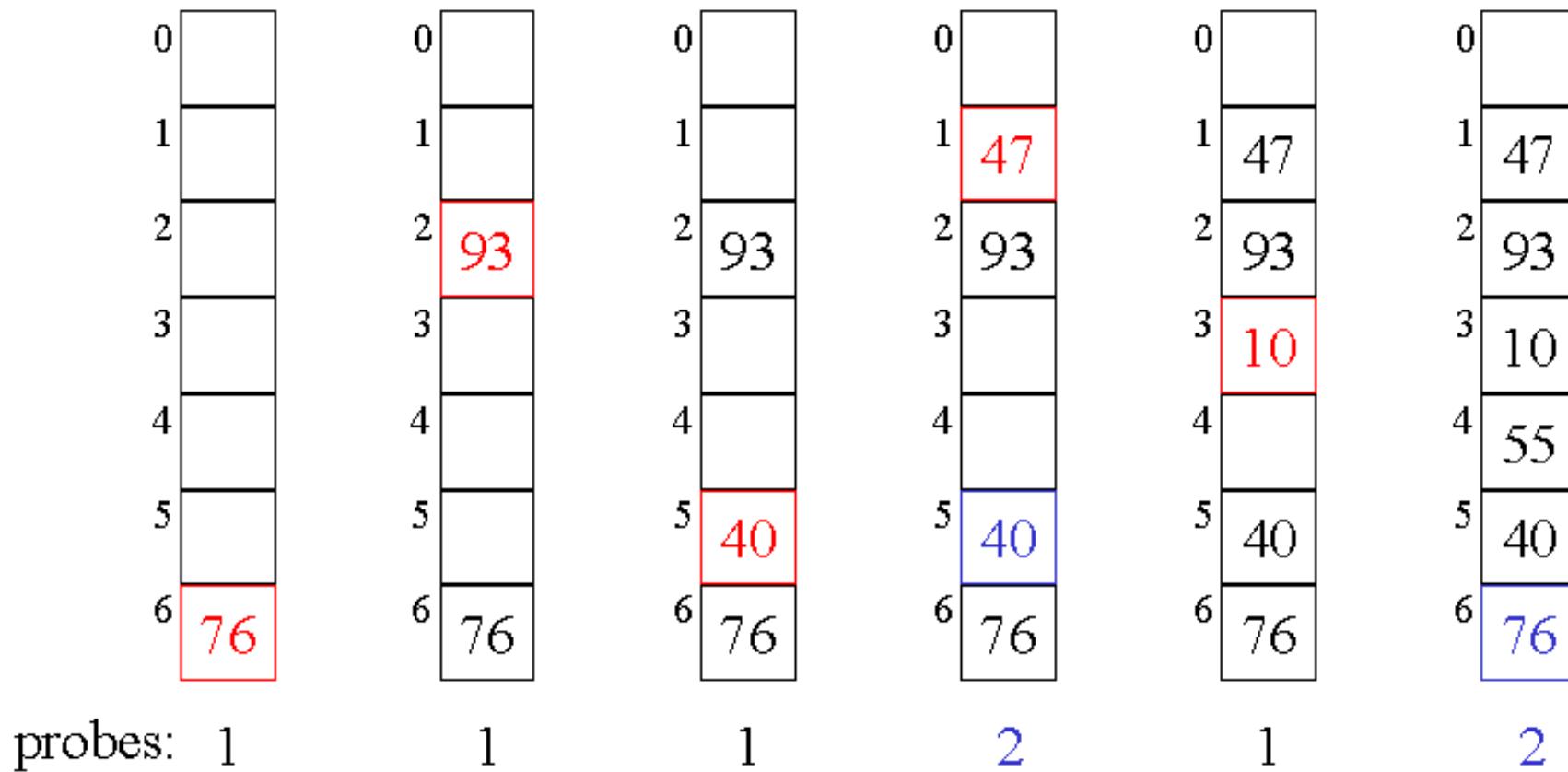
- $58 \bmod 11 = 3$

- $14 \bmod 11 = 3 \rightarrow 3+7=10$

- $91 \bmod 11 = 3 \rightarrow 3+7, 3+2*7 \bmod 11=6$

0	
1	
2	
3	58
4	
5	
6	91
7	
8	
9	
10	14

Example 2 – Double Hashing

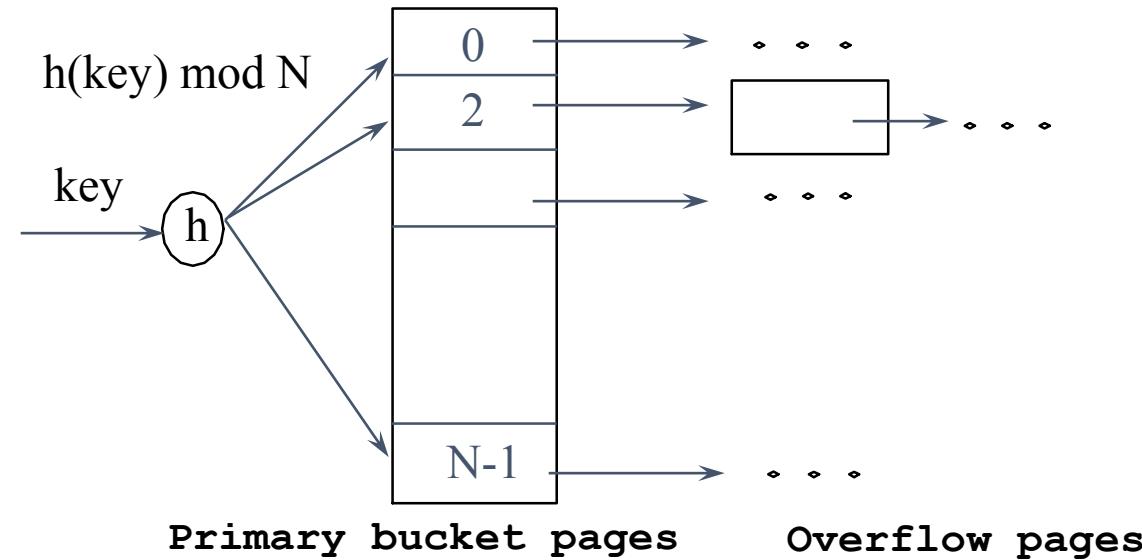


Dynamic Hashing Methods

- ▶ As for any index, 2 alternatives for data entries k^* :
 - $\langle k, \text{rid of data record with search key value } k \rangle$
 - $\langle k, \text{list of rids of data records with search key } k \rangle$
- ▶ Choice orthogonal to the *indexing technique*
- ▶ Hash-based indexes are best for *equality selections*. **Cannot** support range searches.

Static Hashing

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- $h(k) \bmod M$ = bucket to which data entry with key k belongs. ($M = \#$ of buckets)





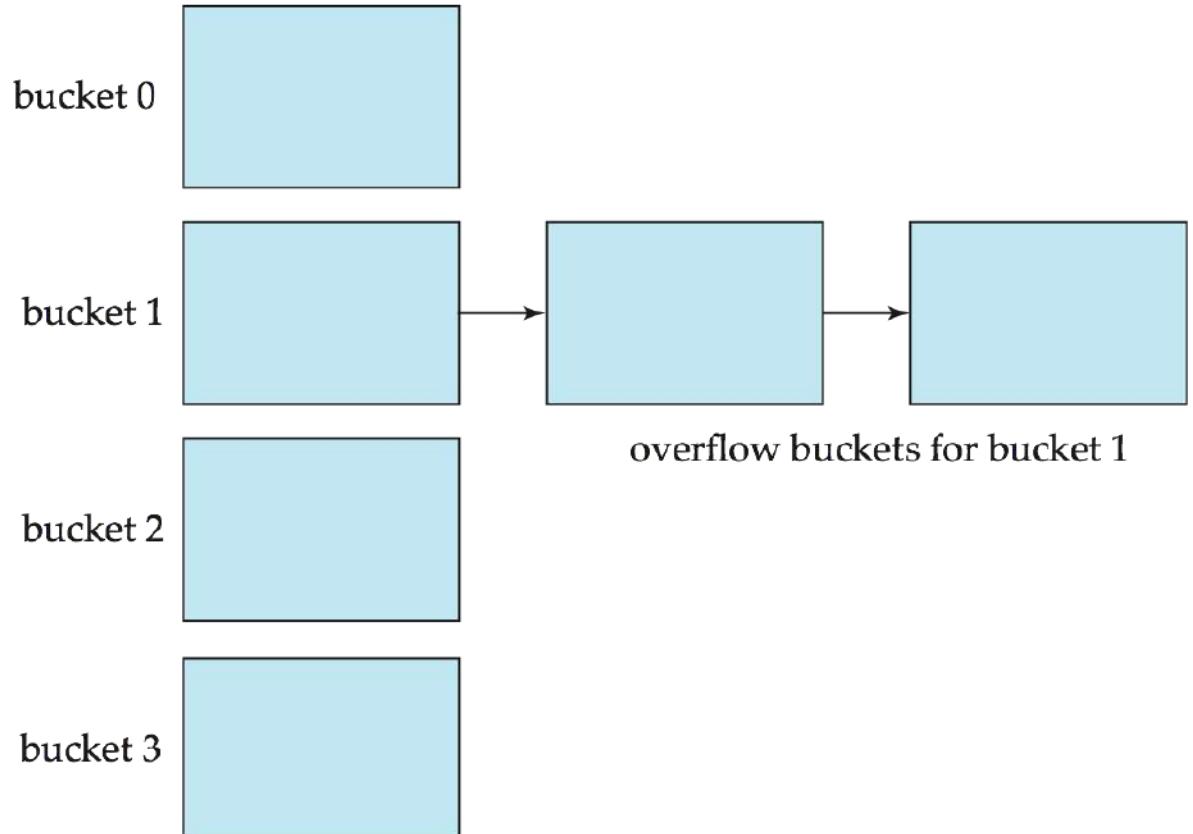
BBM371- Data Management

Lecture 7: Hash-based Indexing

15.11.2018

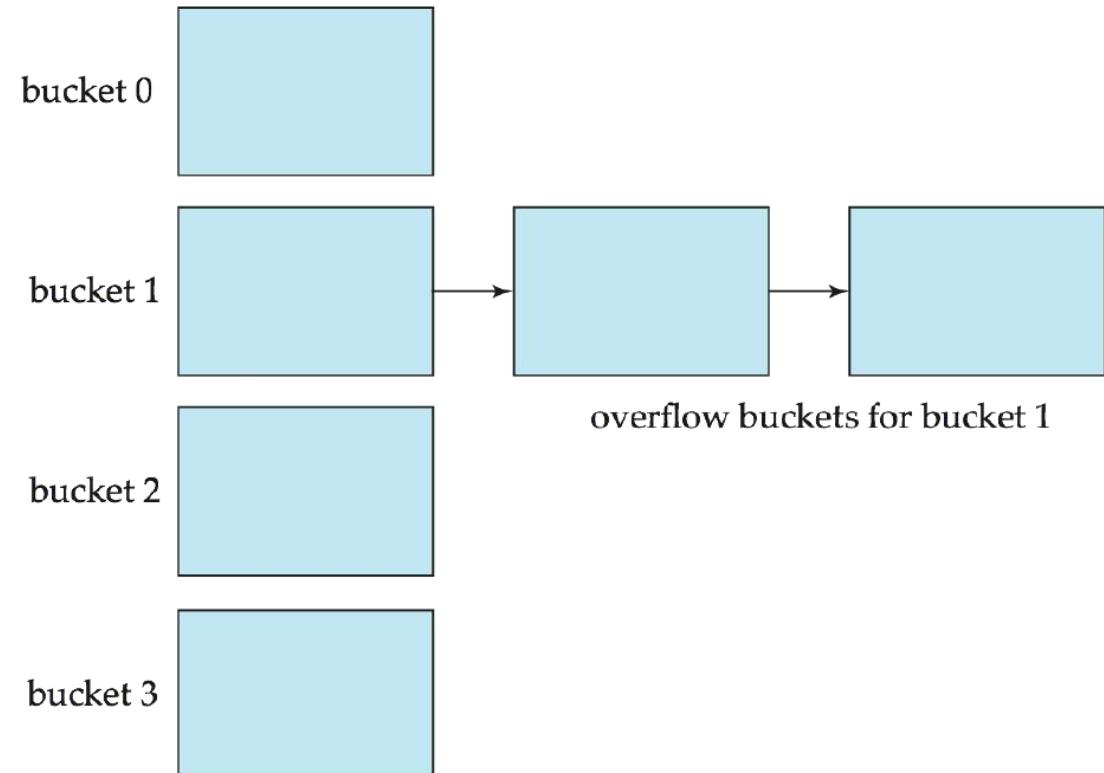
Static Hashing

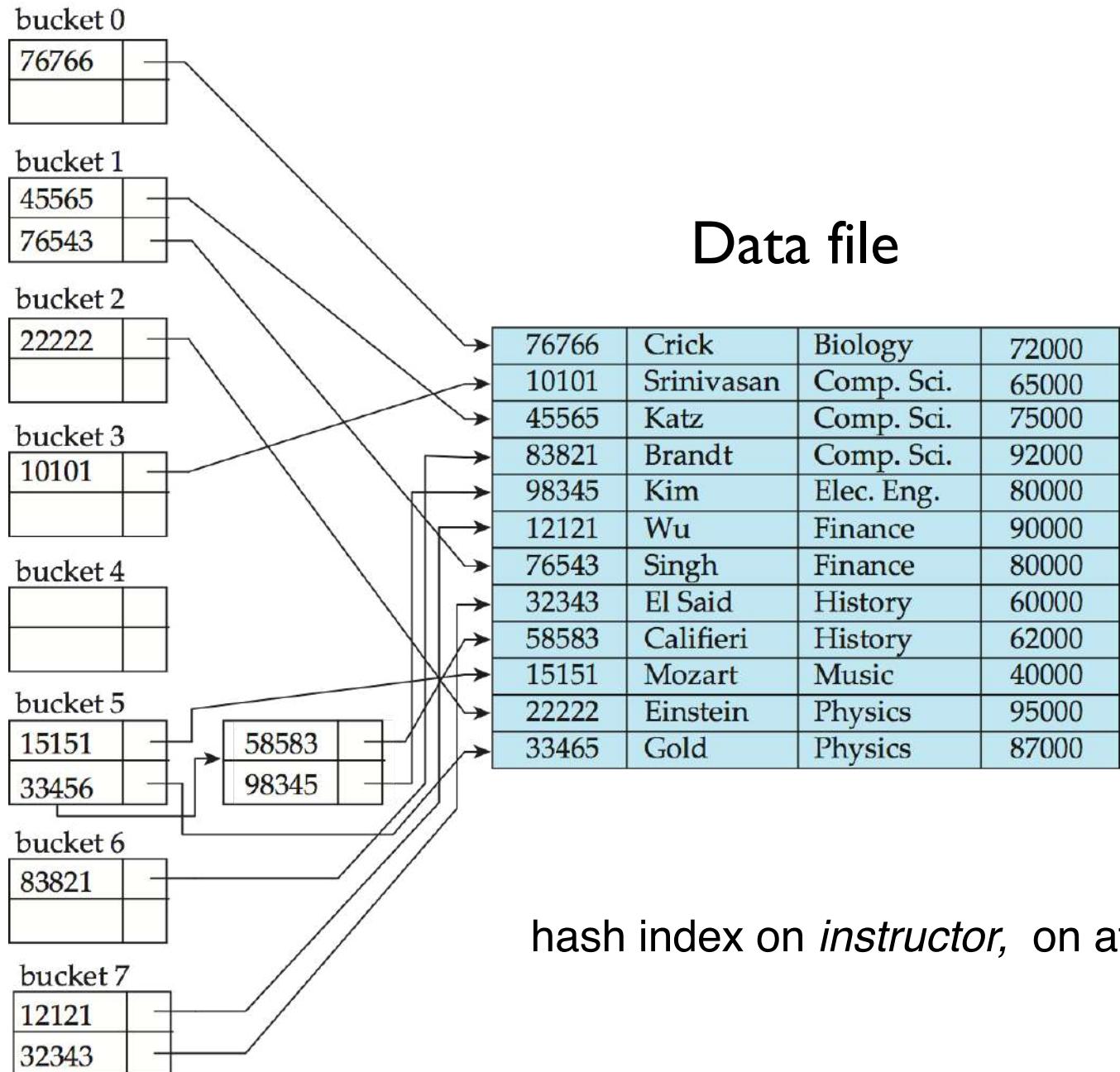
- ▶ Hash tables in the memory can be used in secondary storage as well
- ▶ 0 to $B-1$ indexes where B is the number of Buckets
 - ▶ Buckets is the block or blocks associated with an index
- ▶ Records are stored in buckets indexed with $h(k)$
 - ▶ k : search key of record, $h(k)$ is the hash of search key



Static Hashing

- ▶ If a bucket has too many records, a chain of overflow blocks can be added
- ▶ We can find the first block of each bucket with index i (without a disk access):
 - ▶ We can have an array in main memory with pointers to blocks
 - ▶ We have continuous allocation of buckets and buckets are fixed length





Static Hashing Insert

- ▶ **Insert(record r, search key k)**
- ▶ If Bucket $h(k)$ has space;
 - ▶ Insert r to the first available block (can be either overflow or first block)
- ▶ All blocks for bucket $h(k)$ are filled
 - ▶ Add a new overflow block to the chain
 - ▶ Add record to new block

Static Hashing Delete

- ▶ Delete(all records with search key k)
- ▶ Read first block of $h(k)$ bucket
- ▶ Search for the records with key k, any found is deleted
 - ▶ So we must search overflow blocks as well
- ▶ Can move records to empty spaces in earlier blocks.
- ▶ Optionally, when an overflow block is not needed anymore remove the block
 - ▶ Must avoid oscillations; frequent additions and removal to the same bucket can cause frequent block allocation and release.

Efficiency of Static Hashing

- ▶ Ideally if there are enough buckets;
 - ▶ Most records fit on one block of a bucket
 - ▶ Insertion or deletion takes only two disk I/O
- ▶ If file grows then long chains of blocks needed for buckets
- ▶ Solution: increase the number of buckets B dynamically
- ▶ Increasing B requires re-indexing from scratch
 - ▶ Cannot allow long and expensive operations for normal use of databases.

Dynamic Hashing

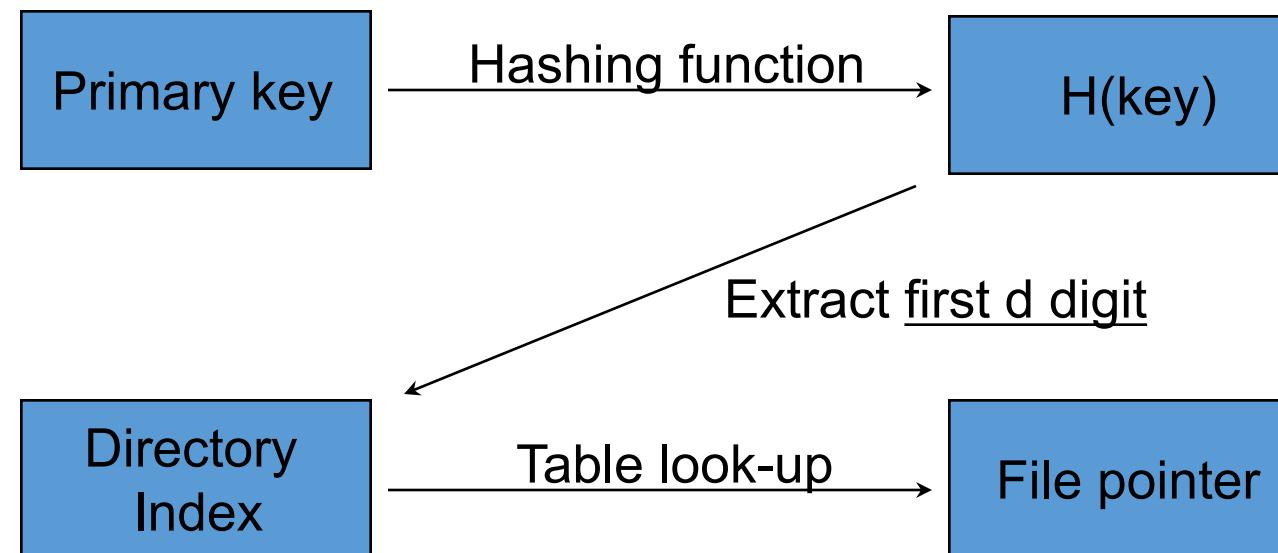
- ▶ Dynamic = Changing number of Buckets B dynamically
- ▶ Two methods
 - ▶ Extendible (or Extensible) Hashing: Grow B by doubling it
 - ▶ Linear Hashing: Grow B by incrementing it by 1
- ▶ To save storage space both methods can choose to shrink B dynamically
 - ▶ Must avoid oscillations when removes and additions are both common.

Extendible Hashing

- Idea: Use *directory of pointers to buckets*,
- double # of buckets B by *doubling the directory*, splitting just the bucket that overflowed!
- Directory much smaller than file, so doubling it is much cheaper.
- Only one page of data entries is split. *No overflow blocks*
- Trick lies in how hash function is adjusted!

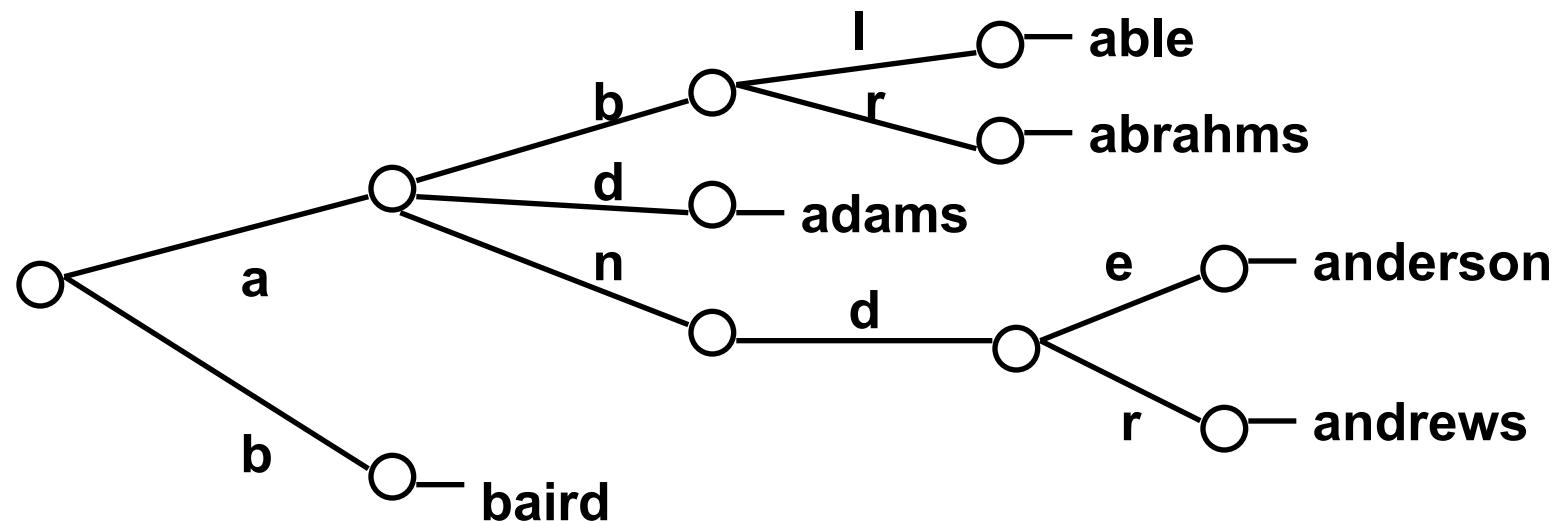
Extendible Hashing Overview

► Extendible Hashing



Hash Function Similar to Tries

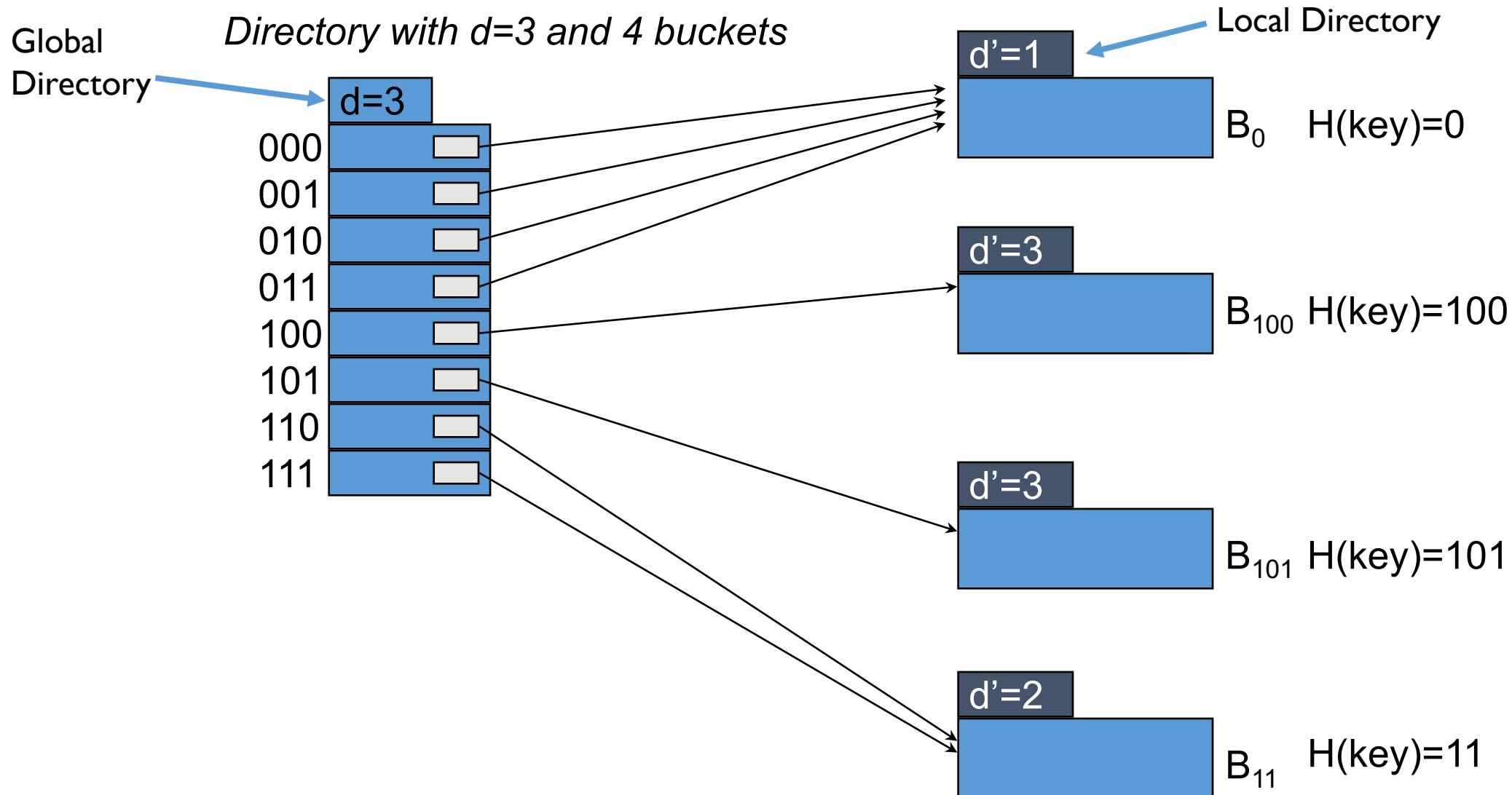
- ▶ Idea from Tries file (radix searching)
 - ▶ The branching factor of the tree is equal to the # of alternative symbols in each position of the key
 - e.g.) Radix 26 trie - *able, abrahms, adams, anderson, adnews, baird*
 - ▶ *Use the first d characters for branching*



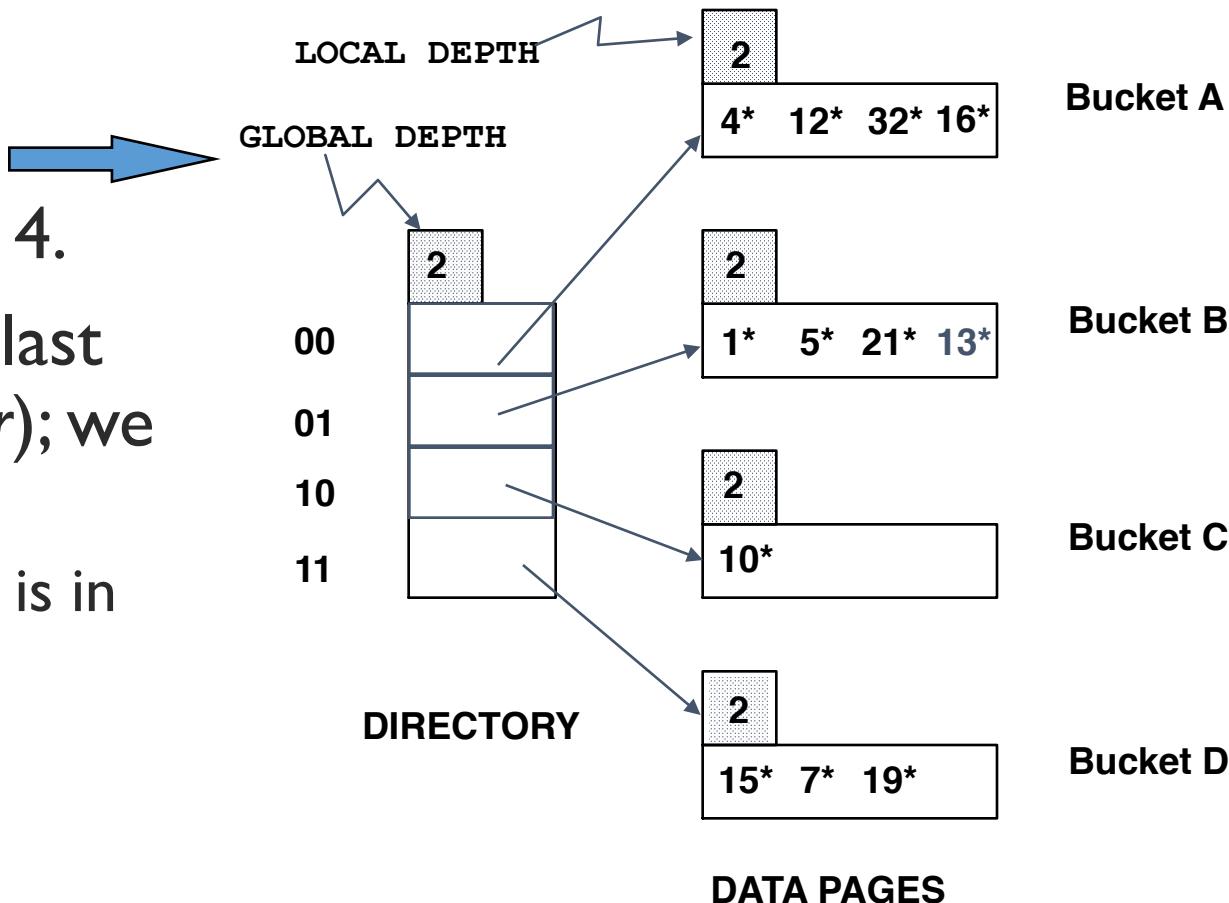
Extendible Hashing

- ▶ $h(k)$ maps keys to a fixed address space, with size the largest prime less than a power of 2 ($65531 < 2^{16}$)
- ▶ File pointers point to blocks of records known as buckets, where an entire bucket is read by one physical data transfer, buckets may be added to or removed from the file dynamically
- ▶ The d bits are used as an index in a directory array containing 2^d entries, which usually resides in primary memory
- ▶ The value d , the directory size(2^d), and the number of buckets change automatically as the file expands and contracts

Extendible Hashing Example



Example

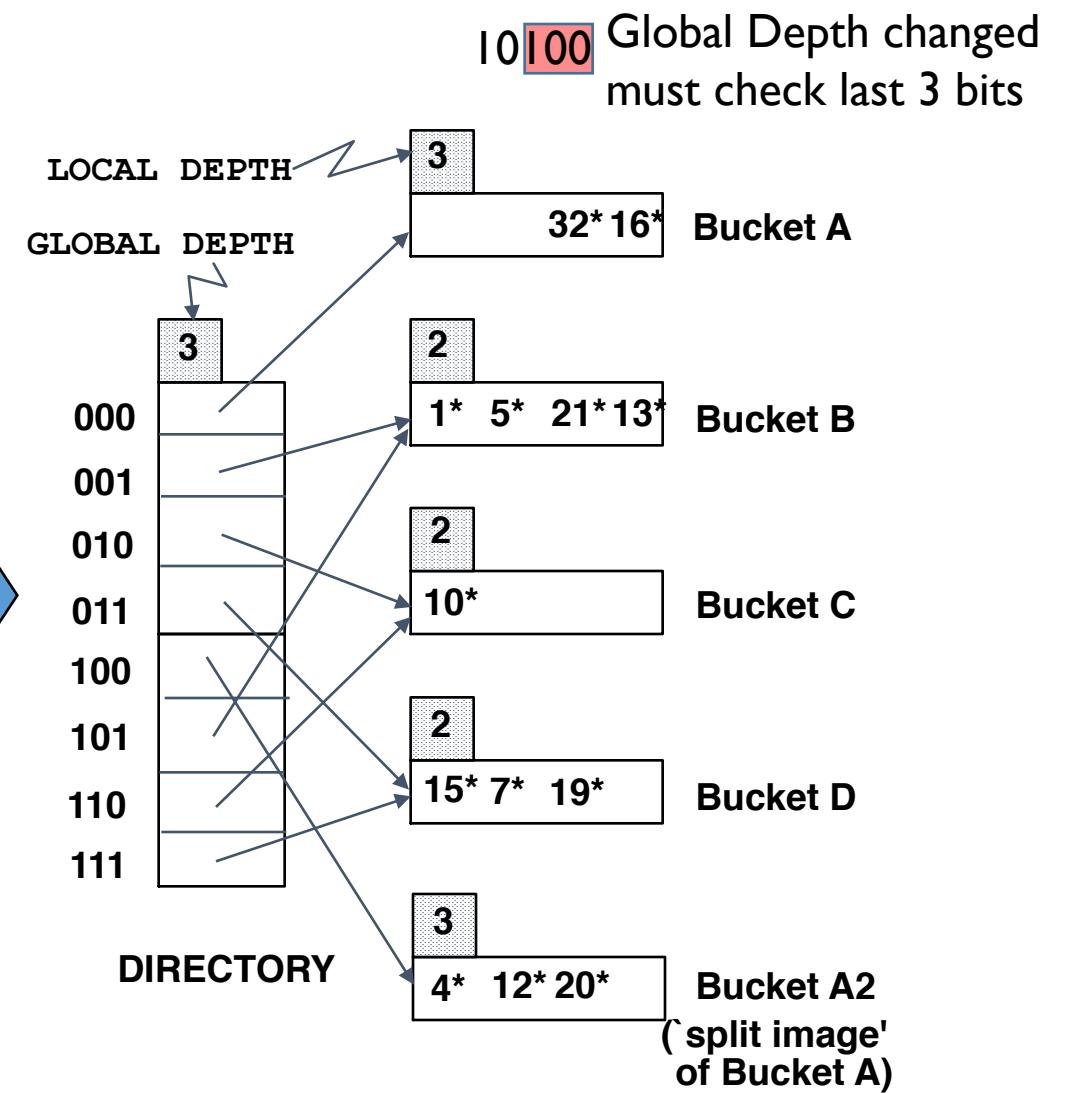
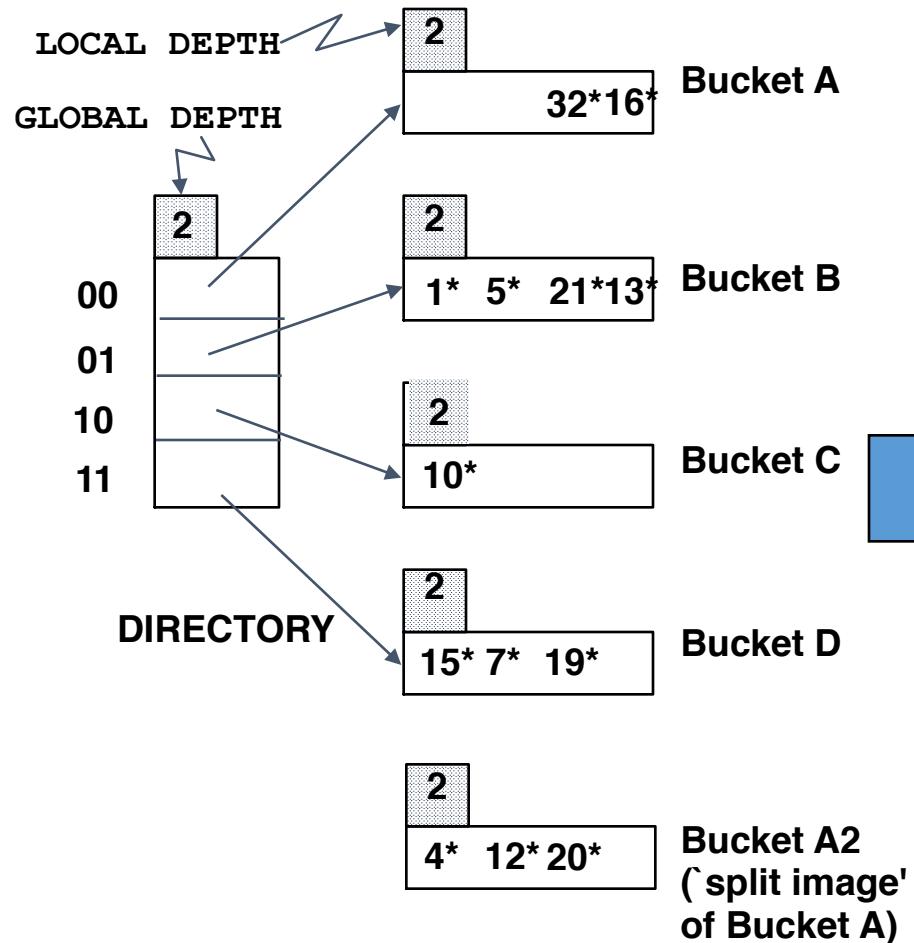


- Directory is array of size 4.
- To find bucket for r , take last '*global depth*' # bits of $\mathbf{h}(r)$; we denote r by $\mathbf{h}(r)$.
 - If $\mathbf{h}(r) = 5 = \text{binary } 101$, it is in bucket pointed to by 01.

- ❖ Insert: If bucket is full, *split* it (*allocate new page, re-distribute*).
- ❖ If necessary, double the directory. (As we will see, splitting a bucket does not always require doubling; we can tell by comparing *global depth* with *local depth* for the split bucket.)

Insert $h(r)=20$ (Causes Doubling)

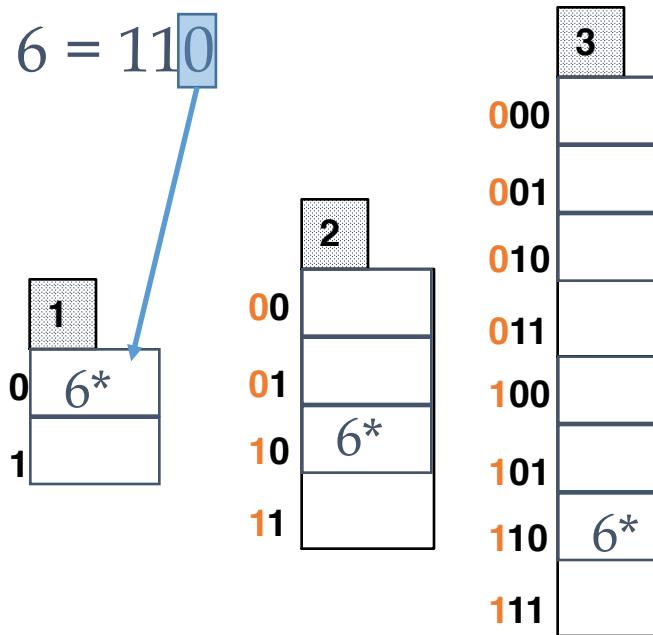
20 in binary is
10100
Check last two
Bits!



Points to Note

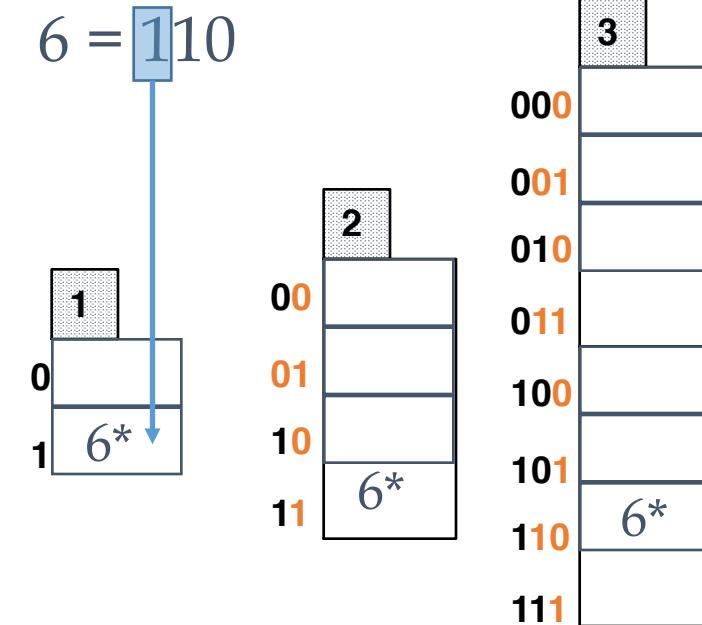
- ▶ $20 = \text{binary } 10100$. Last **2** bits (00) tell us r belongs in A or A2. Last **3** bits needed to tell which.
 - ▶ ***Global depth of directory:*** Max # of bits needed to tell which bucket an entry belongs to.
 - ▶ ***Local depth of a bucket:*** # of bits used to determine if an entry belongs to this bucket.
- ▶ When does bucket split cause directory doubling?
 - ▶ Before insert, *local depth of bucket* = *global depth*. Insert causes *local depth* to become > *global depth*; directory is doubled by ***copying it over*** and ‘fixing’ pointer to split image page. (Use of least significant bits enables efficient doubling via copying of directory!)

Directory Doubling



Least Significant

vs.

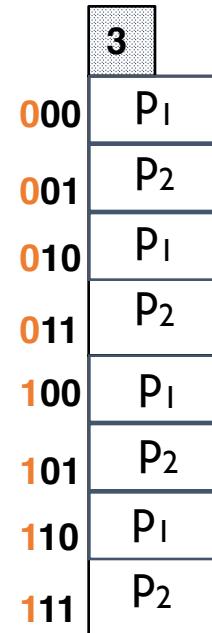
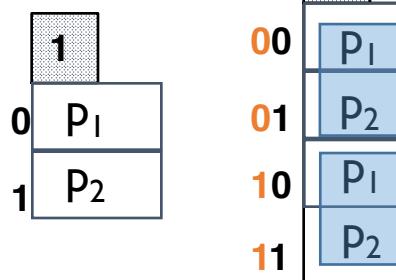


Most Significant

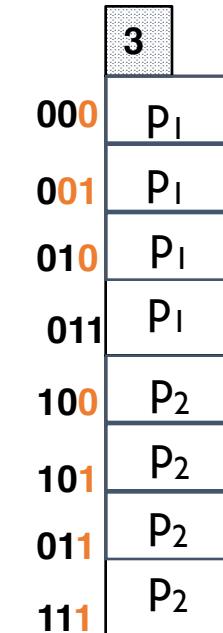
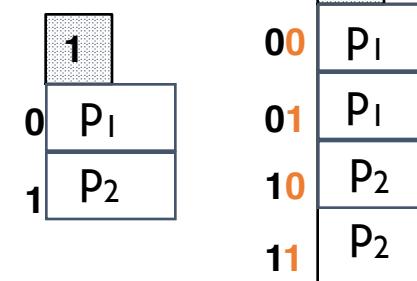
Directory Doubling

Why use least significant bits in directory?
→ Allows for doubling via copying!

Pointers are
in correct order!



Must reorder
pointers



Least Significant

vs.

Most Significant

Extendible Hashing Adv.

- ▶ Can dynamically adjust B as needed
- ▶ Directory only stores pointers to buckets, can be small
 - ▶ Do we need to store bucket index i?
- ▶ Doubling directory can be done easily by copying blocks twice

Extendible Hashing Disadv.

- ▶ Need additional Access to directory
 - ▶ But if it is small we can keep it in memory
- ▶ Must store local directory
- ▶ Can double directory unnecessarily if all keys are mapped to same bucket
 - ▶ Edge case? How many buckets will be formed if all records map to 0000000
- ▶ If directory is larger than memory will require multiple disk access
 - ▶ Doubling will be more expensive
- ▶ Buckets are allocated in different times, hard to get continuous areas.

Linear Hashing

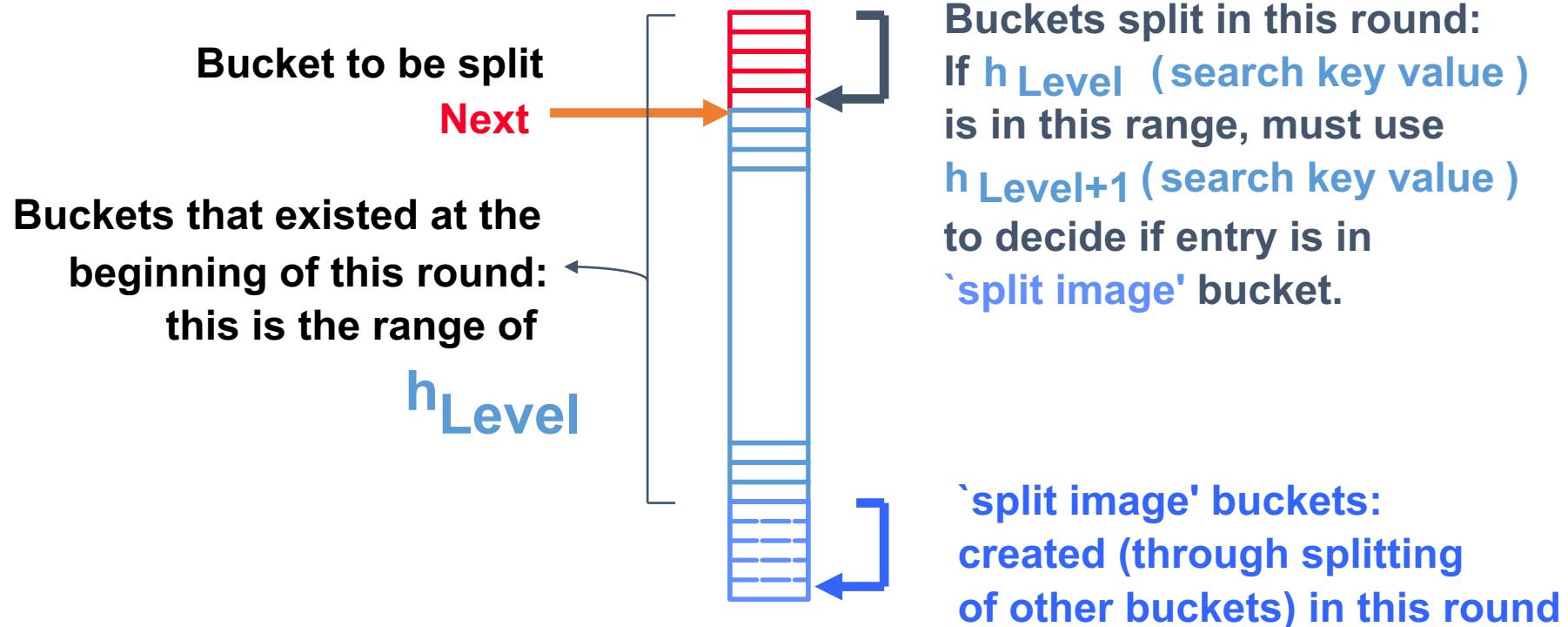
- ▶ This is another dynamic hashing scheme, an alternative to Extendible Hashing.
- ▶ LH handles the problem of long overflow chains without using a directory,
- ▶ Idea: Use a family of hash functions h_0, h_1, h_2, \dots
 - ▶ $h_i(key) = h(key) \text{ mod}(2^i N)$; $N = \text{initial } \# \text{ buckets}$
 - ▶ h is some hash function
 - ▶ If $N = 2^{d_0}$, for some d_0 , h_i consists of applying h and looking at the last d_i bits, where $d_i = d_0 + i$.
 - ▶ h_{i+1} doubles the range of h_i (similar to directory doubling)

Linear Hashing (Contd.)

- ▶ Directory avoided in LH by using overflow pages, and choosing bucket to split round-robin.
 - ▶ **Splitting proceeds in 'rounds'.** Round ends when all N_R initial (for round R) buckets are split. Buckets 0 to Next-1 have been split; Next to N_R yet to be split.
 - ▶ **Current round number is Level.**
 - ▶ **Search:** To find bucket for data entry r , find $\mathbf{h}_{\text{Level}}(r)$:
 - ▶ If $\mathbf{h}_{\text{Level}}(r)$ in range 'Next to N_R ', r belongs here.
 - ▶ Else, r could belong to bucket $\mathbf{h}_{\text{Level}}(r)$ or bucket $\mathbf{h}_{\text{Level}}(r) + N_R$; must apply $\mathbf{h}_{\text{Level+1}}(r)$ to find out.

Overview of LH File

- In the middle of a round.



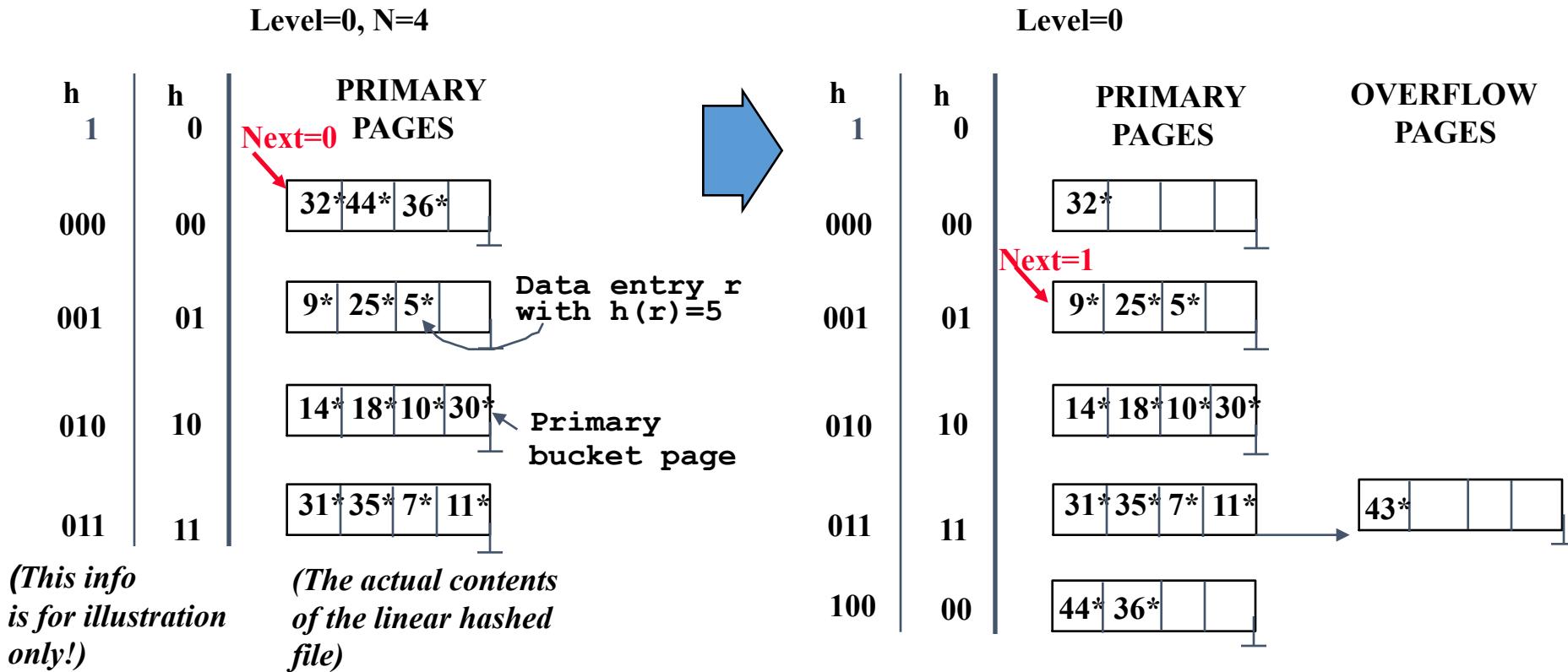
Linear Hashing (Contd.)

- ▶ **Insert:** Find bucket by applying $h_{Level}(r)$ or $h_{Level+1}(r)$:
 - ▶ If bucket to insert into is full:
 - ▶ Add overflow page and insert data entry.
 - ▶ (Maybe) Split Next bucket and increment Next.
 - ▶ Can choose any criterion to 'trigger' split.
 - ▶ Since buckets are split round-robin, long overflow chains are avoided
 - ▶ Doubling of directory in Extendible Hashing is similar; switching of hash functions is *implicit* in how the # of bits examined is increased.

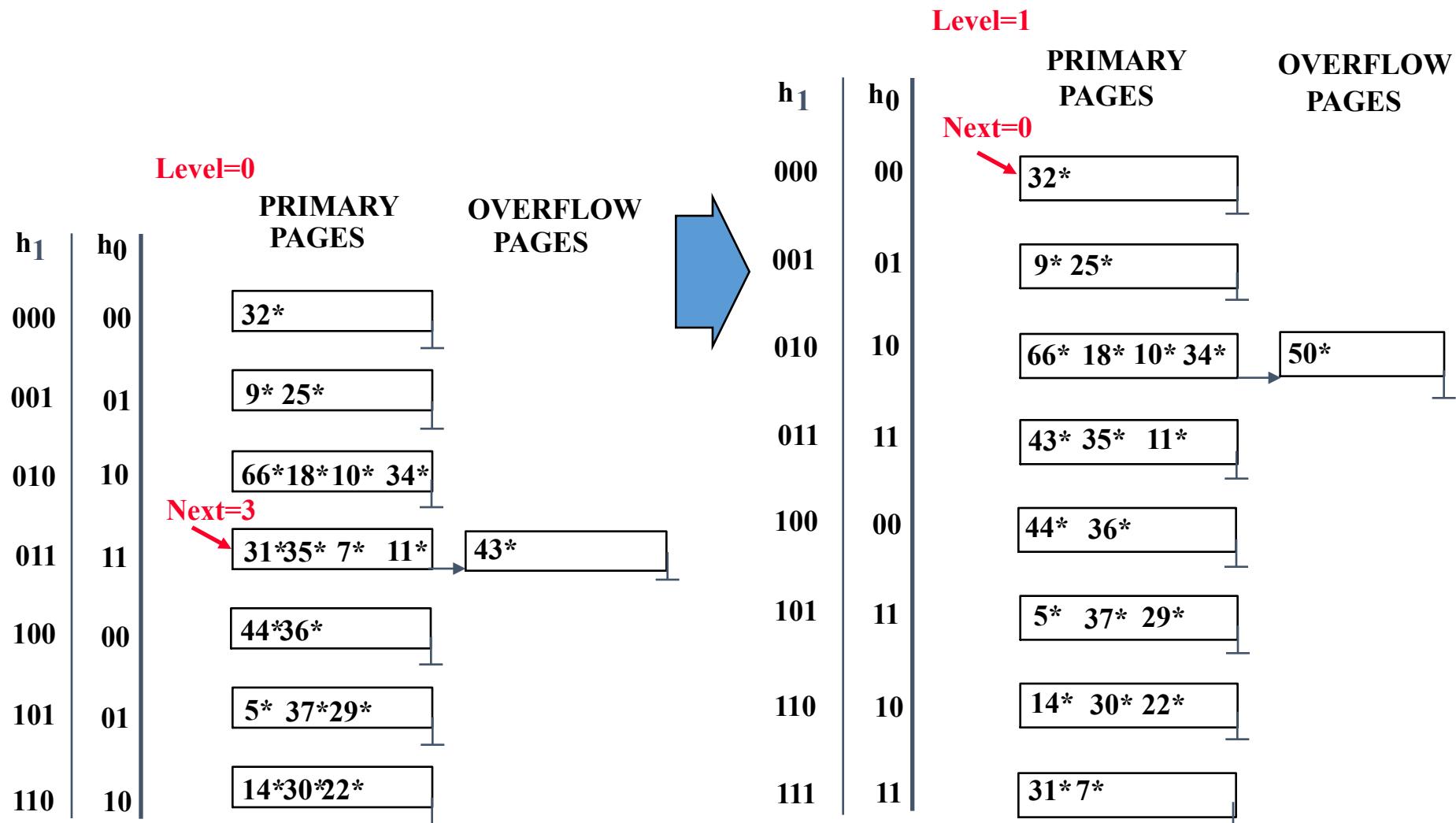
Example of Linear Hashing

- On split, $h_{Level+1}$ is used to re-distribute entries.

insert 43 → adds overflow → triggers split → increments Next



Example: End of a Round



Summary

- ▶ Hash-based indexes: best for equality searches, cannot support range searches.
- ▶ Static Hashing can lead to long overflow chains.
- ▶ Extendible Hashing avoids overflow pages by splitting a full bucket when a new data entry is to be added to it. (*Duplicates may require overflow pages.*)
 - ▶ Directory to keep track of buckets, doubles periodically.
 - ▶ Can get large with skewed data; additional I/O if this does not fit in main memory.

Summary (Contd.)

- ▶ Linear Hashing avoids directory by splitting buckets round-robin, and using overflow pages.
 - ▶ Overflow pages not likely to be long.
 - ▶ Duplicates handled easily.
 - ▶ Space utilization could be lower than Extendible Hashing, since splits not concentrated on 'dense' data areas.
 - ▶ Can tune criterion for triggering splits to trade-off slightly longer chains for better space utilization.
- ▶ For hash-based indexes, a *skewed data distribution* is one in which the *hash values* of data entries are not uniformly distributed!

Indexing vs Hashing

- ▶ Hashing good for probes given key

e.g., SELECT ...

FROM R

WHERE R.A = 5

Indexing vs Hashing

- ▶ INDEXING (Including B Trees) good for Range Searches:

e.g., SELECT

FROM R

WHERE R.A > 5



BBM371- Data Management

Lecture 8 - Tree Based Indexing

(B-Trees)

29.11.2018

Hash based indexing vs Tree based indexing

- ▶ Remember that hash based indexing is used for equality search and does not work for range search.
- ▶ Tree based indexing is used for range search (and also for equality search).

Select * from Ogrenci where Ogrenci.Age>40 and Ogrenci.Age<60

B-Tree

- ▶ **Rudolf Bayer** and **Ed McCreight** invented the B-tree while working at Boeing Research Labs in 1971 (Bayer & McCreight 1972), but they did not explain what, if anything, the B stands for. Douglas Comer explains:
 - ▶ The origin of "B-tree" has never been explained by the authors. As we shall see, "**balanced**," "**broad**," or "**bushy**" might apply. However, it seems appropriate to think of B-trees as "Bayer"-trees. (Comer 1979, p. 123)
 - ▶ Comer, Douglas (June 1979), "The Ubiquitous B-Tree", Computing Surveys 11 (2): 123–137, doi:10.1145/356770.356776, ISSN 0360-0300.

B-Tree

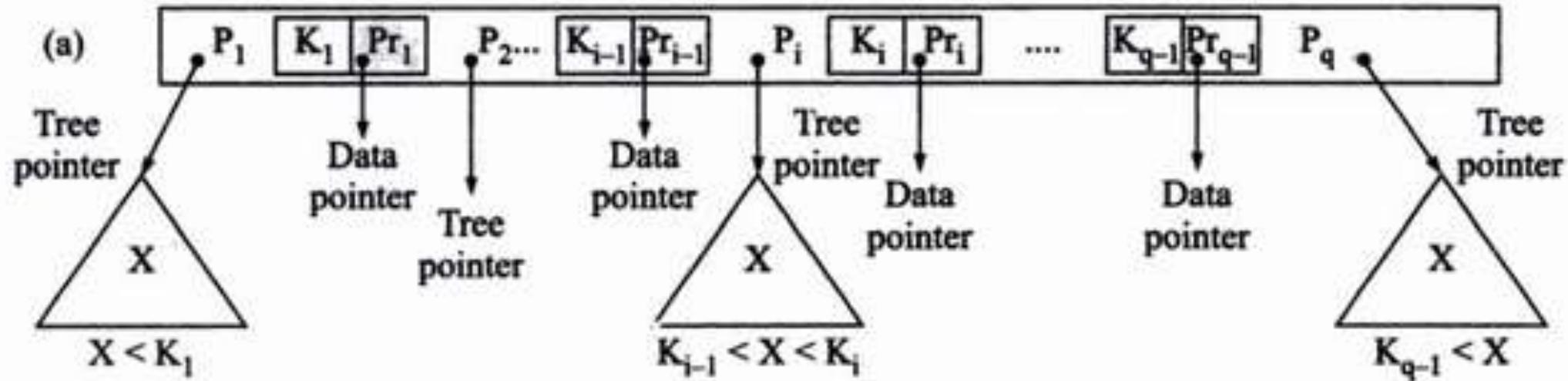
- ▶ B-tree is a specialized multiway tree designed especially for use on the disk
- ▶ B-Tree consists of a root node, intermediate nodes and leaf nodes containing the indexed field values in the leaf nodes of the tree.

B-Tree Characteristics

- ▶ In a B-tree each node may contain a large number of keys
- ▶ B-tree is designed to branch out in a large number of directions and to contain a lot of keys in each node so that the height of the tree is relatively small
- ▶ Tree is always **balanced**.
- ▶ Space wasted by deletion, if any, never becomes excessive

Node Structure in a B-Tree

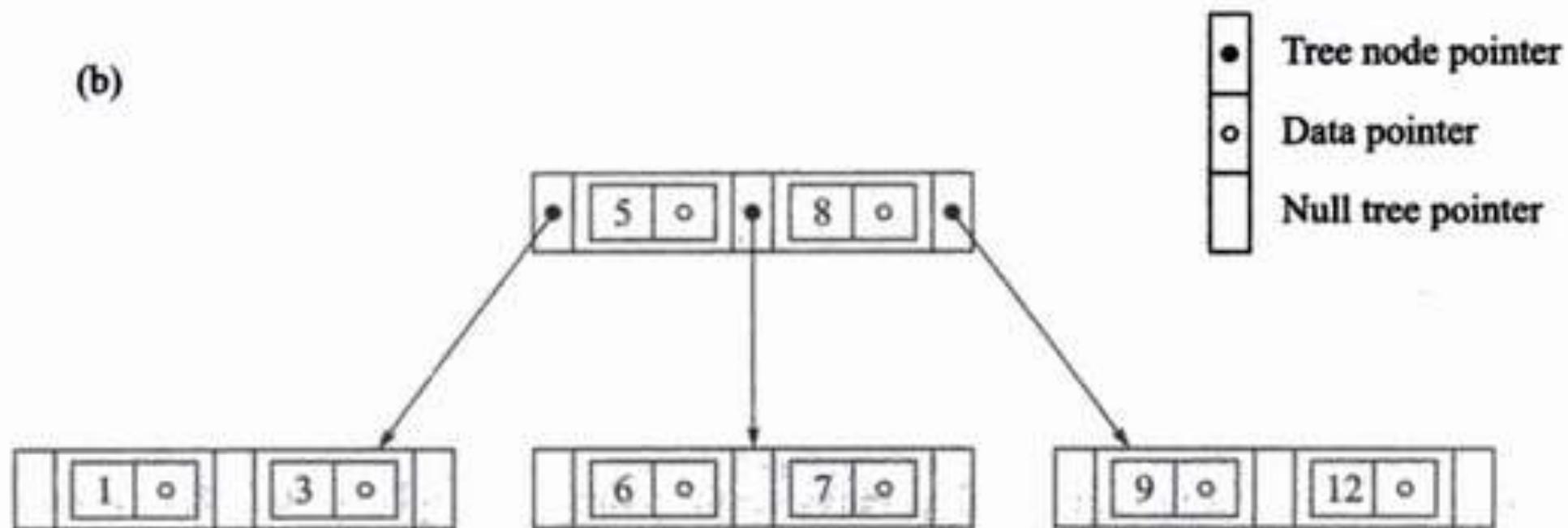
A node in a B-tree:



Each P is a tree pointer and each Pr is a data pointer.

Node Structure in a B-Tree

A B-Tree of min. order d=3:



Aspects of B Tree

- ▶ Specs of min ' $d+1$ ' order B Tree:
 1. Internal nodes (except for root) has at least d , at most $2d$ keys
 2. Root (if it is not a leaf) has at least 2 children
 3. All leaf nodes are in the same level (balanced tree)
 4. Level increase and decreases are handled towards up (not down)
 5. An internal node has at least d keys and $d+1$ children

Constructing a B-tree

- ▶ Suppose we start with an empty B-tree and keys arrive in the following order:

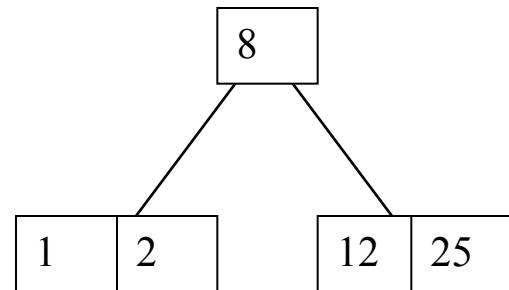
1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45

- ▶ We want to construct a B-tree of max order 5 ($d=2$).
- ▶ The first four items go into the root node:

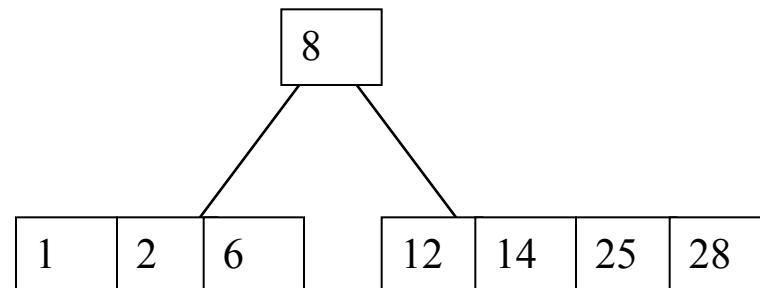
1	2	8	12
---	---	---	----

- ▶ Insertion of the fifth item in the root node violates condition 5
- ▶ Therefore, when 25 arrives, pick the middle key to make a new root

Constructing a B-tree (contd.)

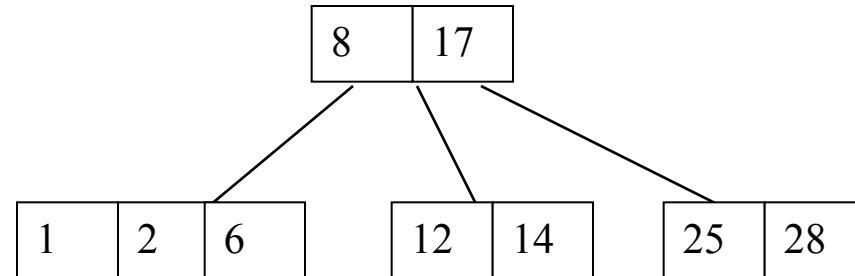


6, 14, 28 get added to the leaf nodes:

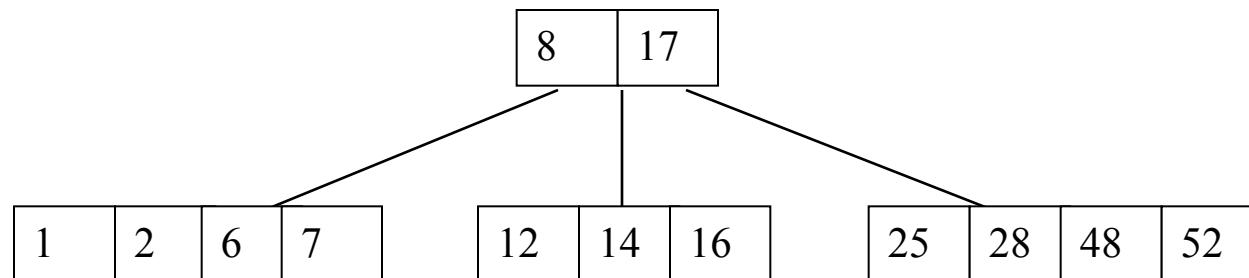


Constructing a B-tree (contd.)

Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf

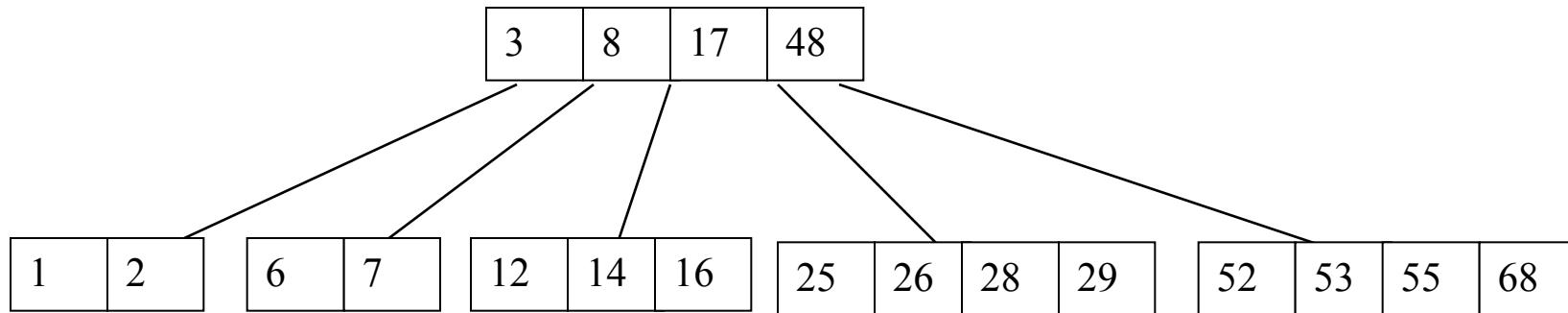


7, 52, 16, 48 get added to the leaf nodes



Constructing a B-tree (contd.)

Adding 68 causes us to split the rightmost leaf, promoting 48 to the root, and adding 3 causes us to split the left most leaf, promoting 3 to the root; 26, 29, 53, 55 then go into the leaves

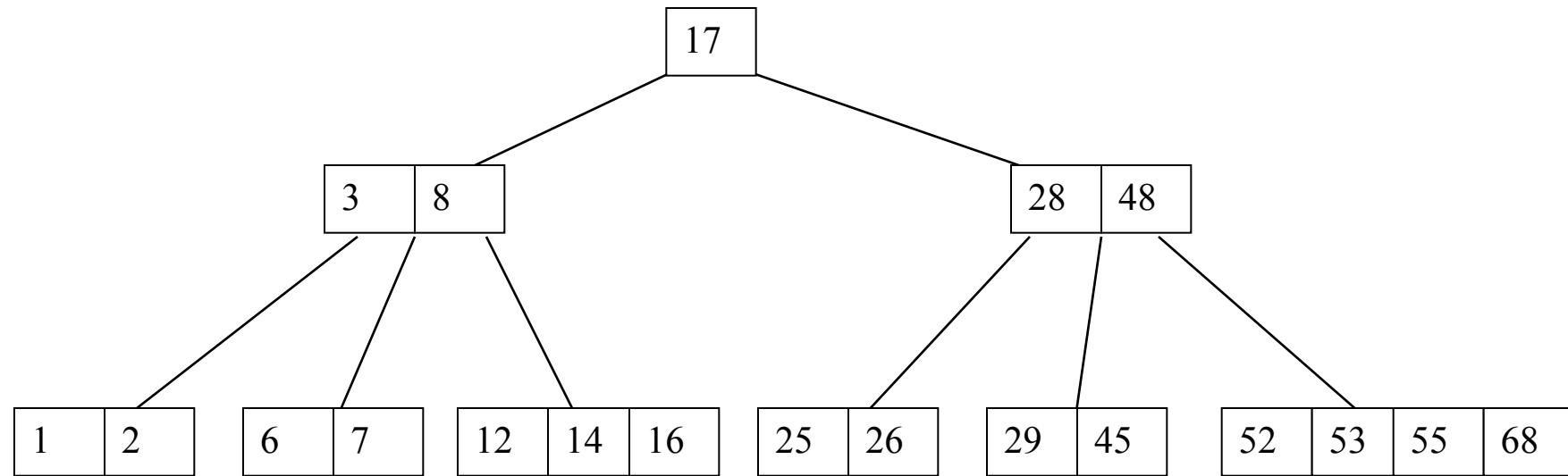


Adding 45 causes a split of

25	26	28	29
----	----	----	----

and promoting 28 to the root then causes the root to split

Constructing a B-tree (contd.)



Inserting into a B-Tree

- ▶ Attempt to insert the new key into a leaf
- ▶ If this would result in that leaf becoming too big, split the leaf into two, promoting the middle key to the leaf's parent
- ▶ If this would result in the parent becoming too big, split the parent into two, promoting the middle key
- ▶ This strategy might have to be repeated all the way to the top
- ▶ If necessary, the root is split in two and the middle key is promoted to a new root, making the tree one level higher

Structure of a B-Tree Node

- ▶ **Bucket Factor:** number of records inserted into one node
- ▶ **Fan-out :** number of children sourced from one node
 - ▶ High fan-out makes the tree bushy and therefore leads to low height.
 - ▶ A high fan-out makes the tree more efficient.
- ▶ The size of each node is equal to the size of the page/block

Exercise in Inserting a B-Tree

- ▶ Insert the following keys to a 5-way B-tree:
- ▶ 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56

Removal from a B-tree

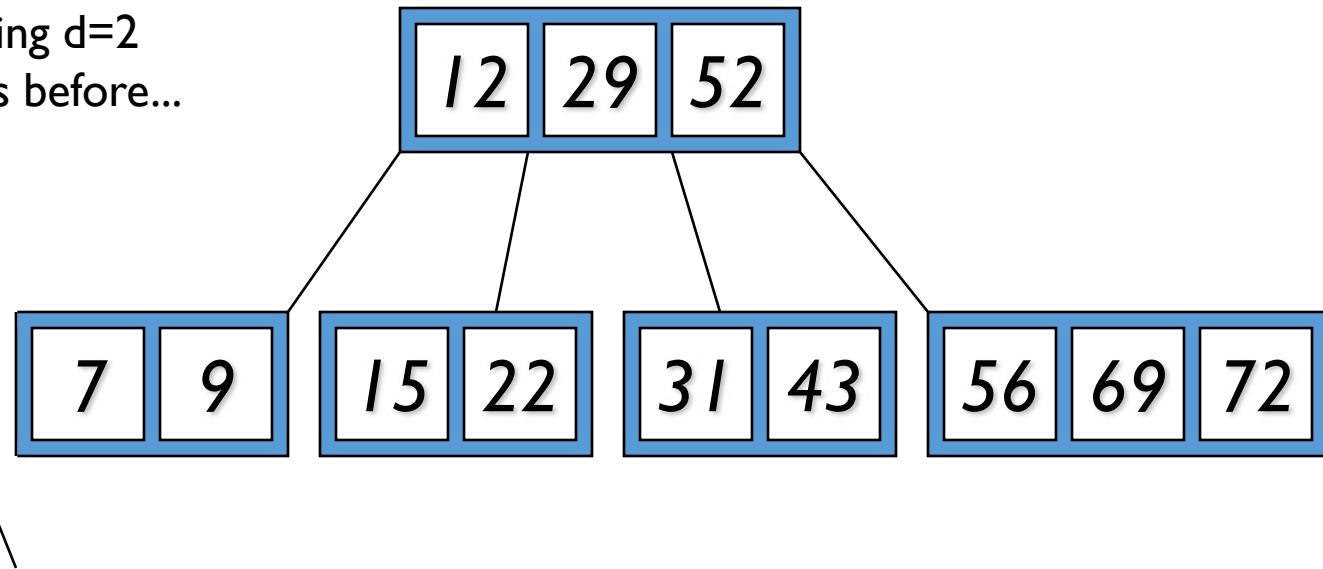
- ▶ During insertion, the key always goes *into* a leaf. For deletion we wish to remove *from* a leaf. There are three possible ways we can do this:
 - I. If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted.
 2. If the key is *not* in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf -- in this case we can delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.

Removal from a B-tree (2)

- ▶ If (1) or (2) lead to a child node containing less than the minimum number of keys then we have to look at the siblings immediately adjacent to the leaf in question:
 - ▶ 3: if one of them has more than the min. number of keys then we can promote one of its keys to the parent and take the parent key into our lacking leaf
 - ▶ 4: if neither of them has more than the min. number of keys then the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key) and the new leaf will have the correct number of keys; if this step leave the parent with too few keys then we repeat the process up to the root itself, if required

Type #1: Simple leaf deletion

Assuming d=2
B-Tree, as before...

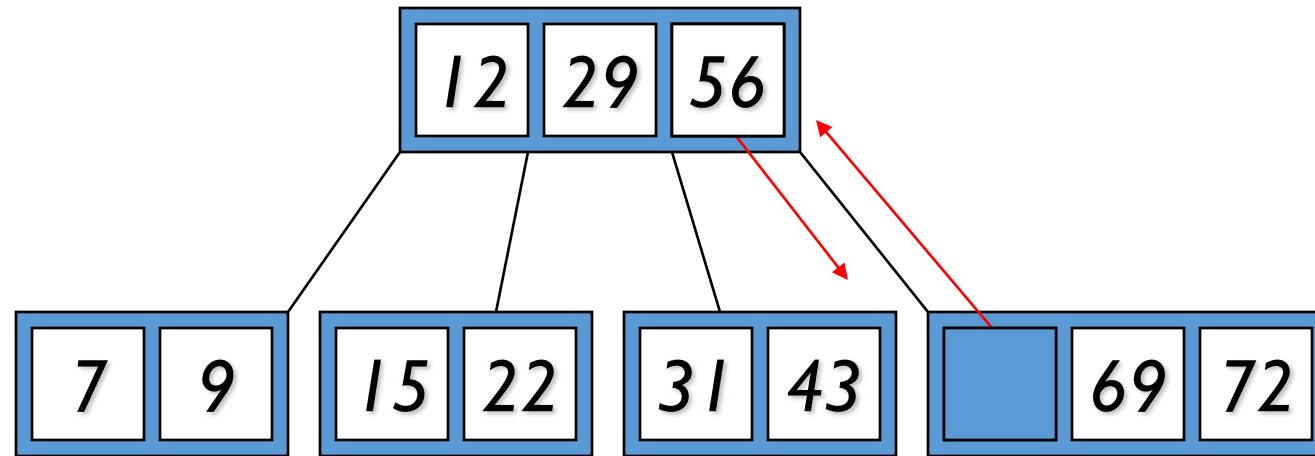


Delete 2: Since there are enough
keys in the node, just delete it

Note when printed: this slide is animated

Type #2: Simple non-leaf deletion

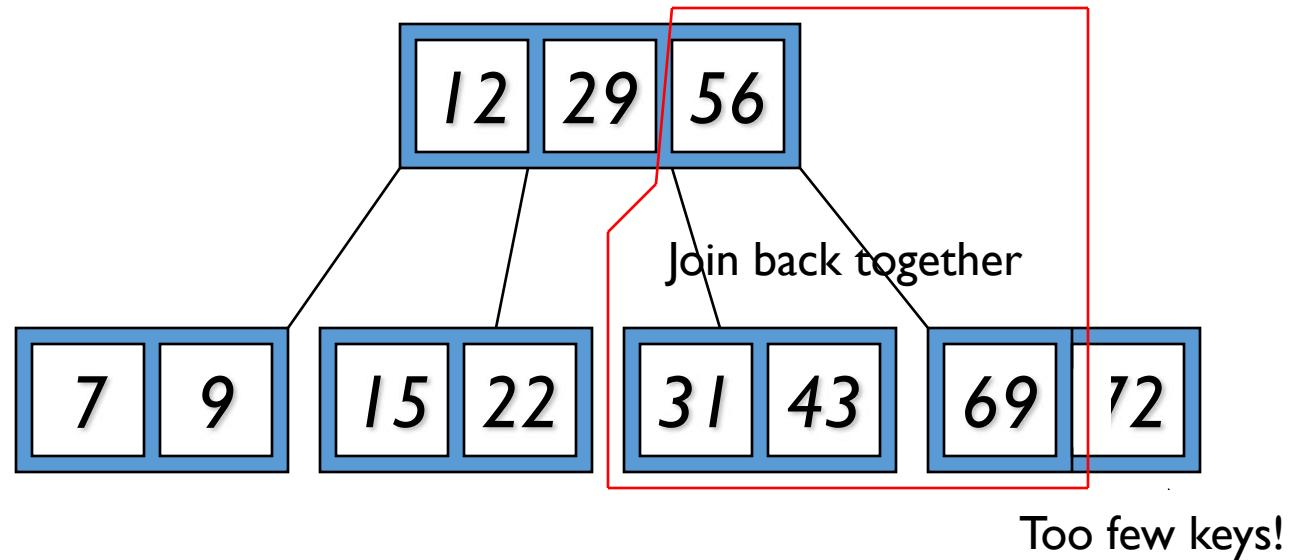
Delete 52:



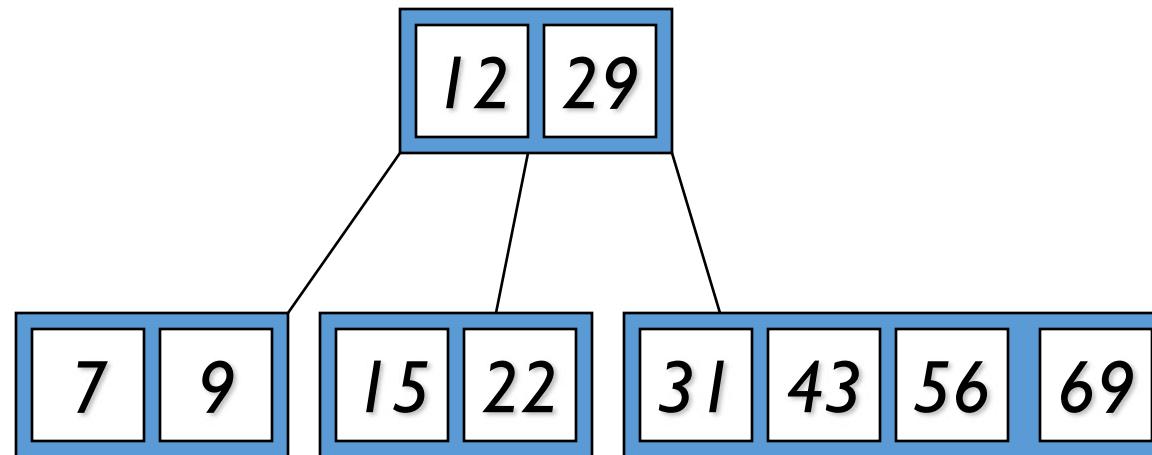
Note when printed: this slide is animated

Type #4: Too few keys in node and its siblings

Delete 72:

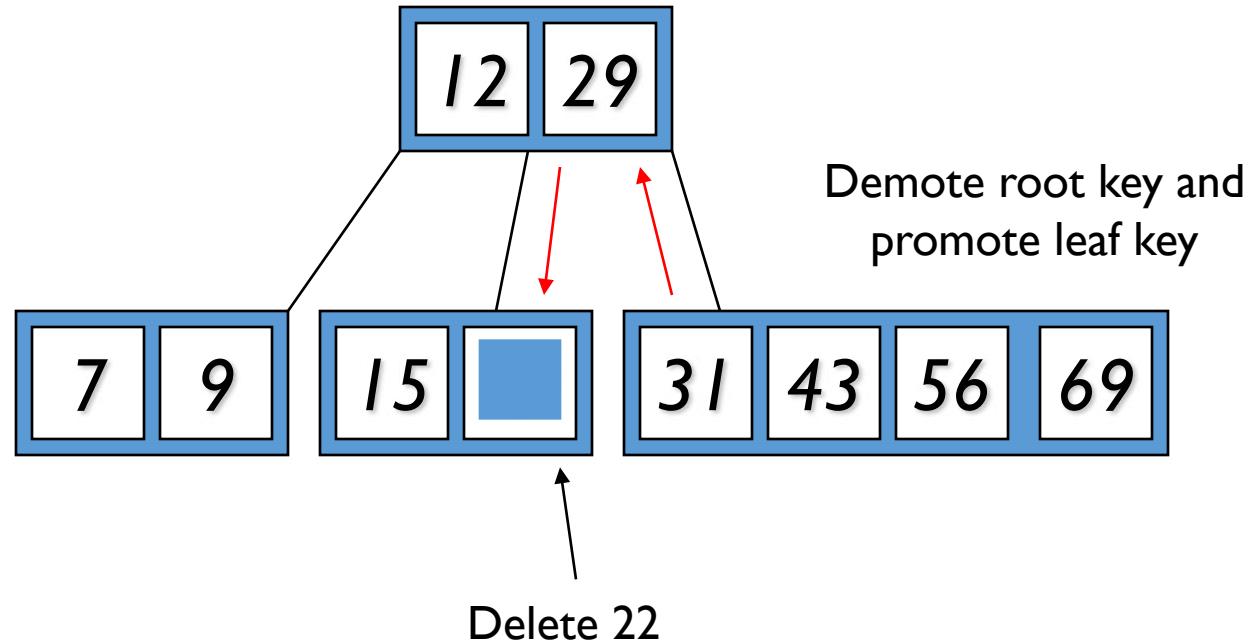


Type #4: Too few keys in node and its siblings



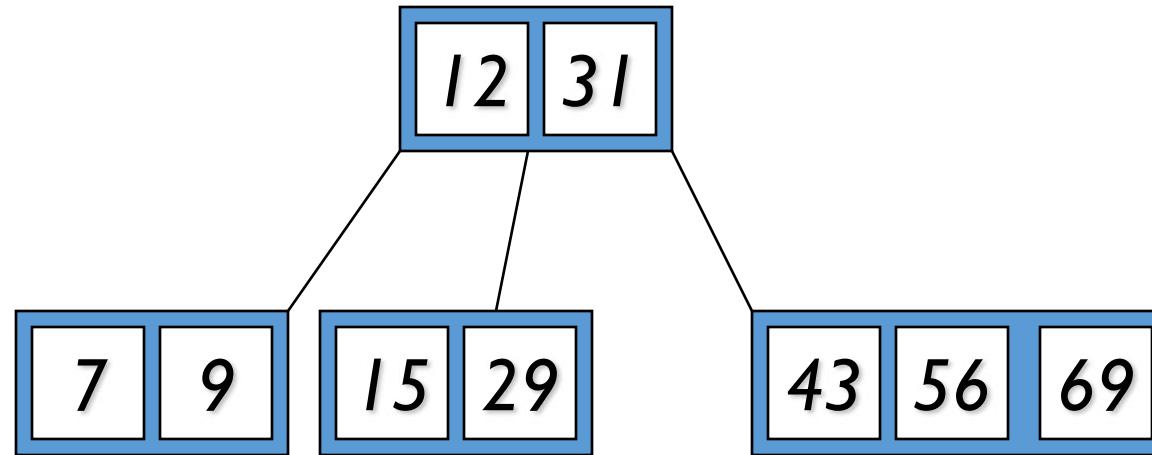
Note when printed: this slide is animated

Type #3: Enough siblings



Note when printed: this slide is animated

Type #3: Enough siblings



Note when printed: this slide is animated

Exercise for Removing from a B-Tree

- ▶ Given a max 5 order B-tree created by these data (last exercise):
3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56
- ▶ Add these further keys: 2, 6, 12
- ▶ Delete these keys: 4, 5, 7, 3, 14

Analysis of B-Trees

- ▶ The maximum number of items in a B-tree of max order m ($2d+1$) and height h :

$$\text{root} \quad m - 1$$

$$\text{level 1} \quad m(m - 1)$$

$$\text{level 2} \quad m^2(m - 1)$$

...

$$\text{level } h \quad m^h(m - 1)$$

- ▶ So, the total number of items is

$$(1 + m + m^2 + m^3 + \dots + m^h)(m - 1) =$$

$$[(m^{h+1} - 1)/ (m - 1)] (m - 1) = \mathbf{m^{h+1} - 1}$$

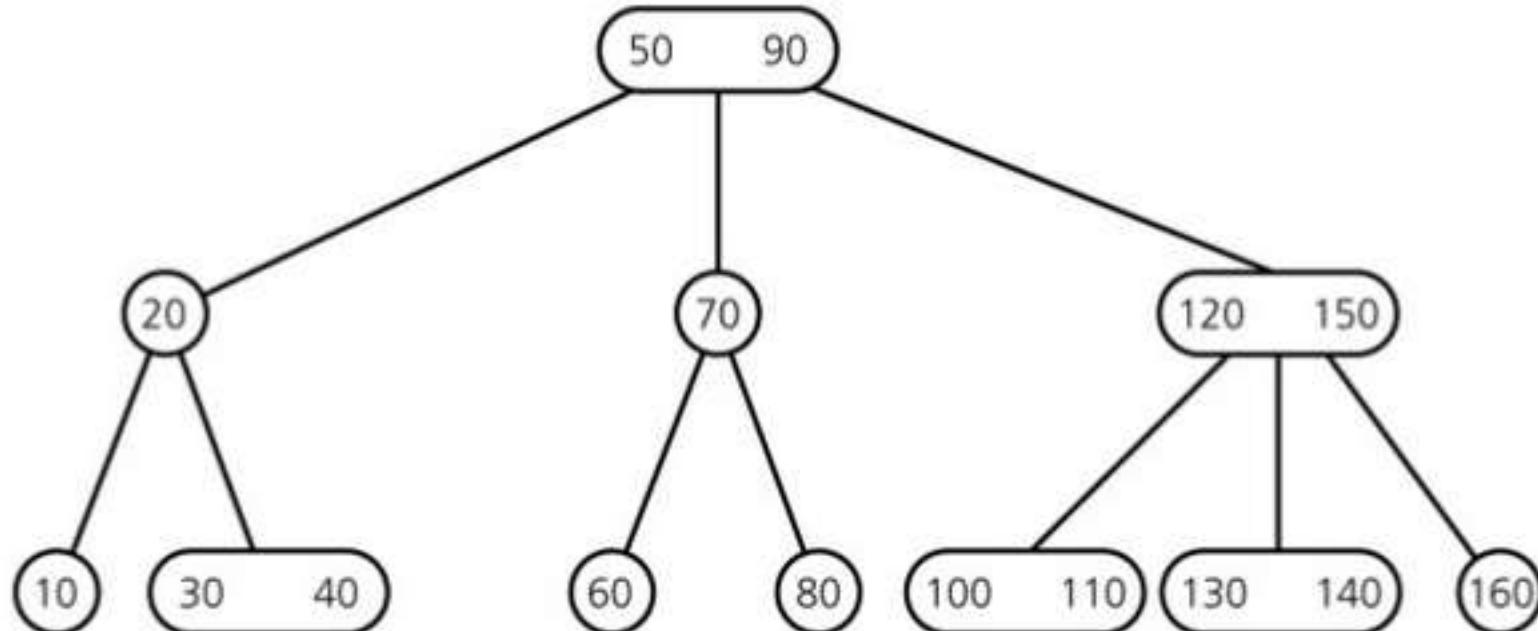
- ▶ When $m = 5$ and $h = 2$ this gives $5^3 - 1 = 124$

Reasons for using B-Trees

- ▶ When searching tables held on disc, the cost of each disc transfer is high but doesn't depend much on the amount of data transferred, especially if consecutive items are transferred
 - ▶ If we use a B-tree of max order 101, say, we can transfer each node in one disc read operation
 - ▶ A B-tree of order 101 and height 3 can hold $101^4 - 1$ items (approximately 100 million) and any item can be accessed with 3 disc reads (assuming we hold the root in memory)

2-3 Tree

- If we take $d=1$, we get a **2-3 tree**, in which non-leaf nodes have two or three children (i.e., one or two keys)
 - B-Trees are always balanced (since the leaves are all at the same level), so 2-3 trees make a good type of balanced tree





BBM371- Data Management

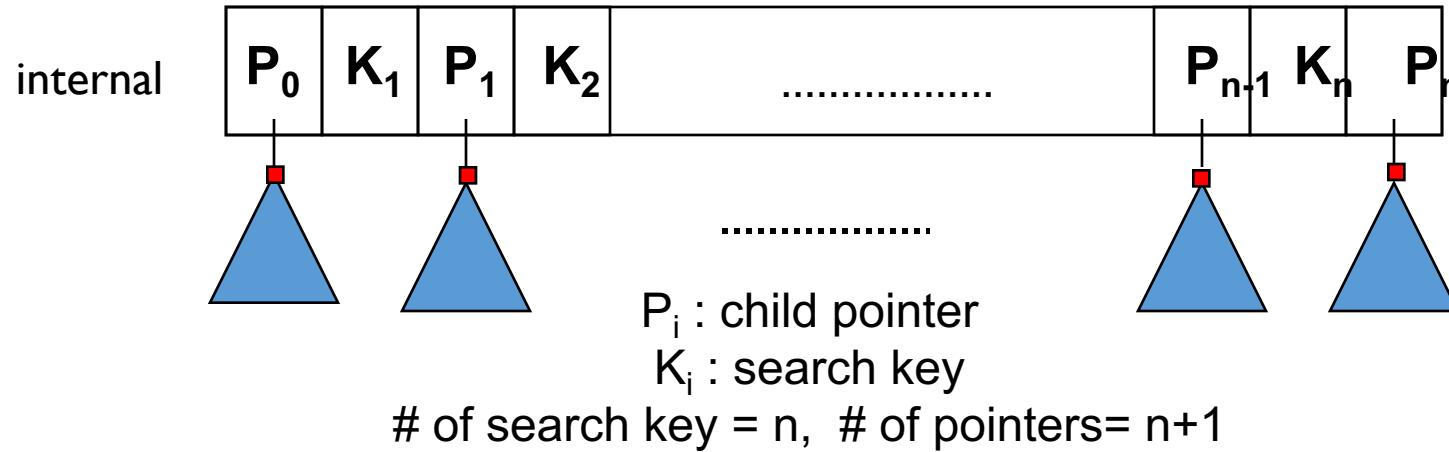
Lecture 9 – B+Trees

6.12.2018

Main Characteristics of a B+ tree

- ▶ B+ tree is always balanced.
- ▶ A minimum occupancy of 50 percent is guaranteed for each node except the root node.
- ▶ Searching for a record requires just a traversal from the root to the appropriate leaf.

B+ Tree: node structure



leaf

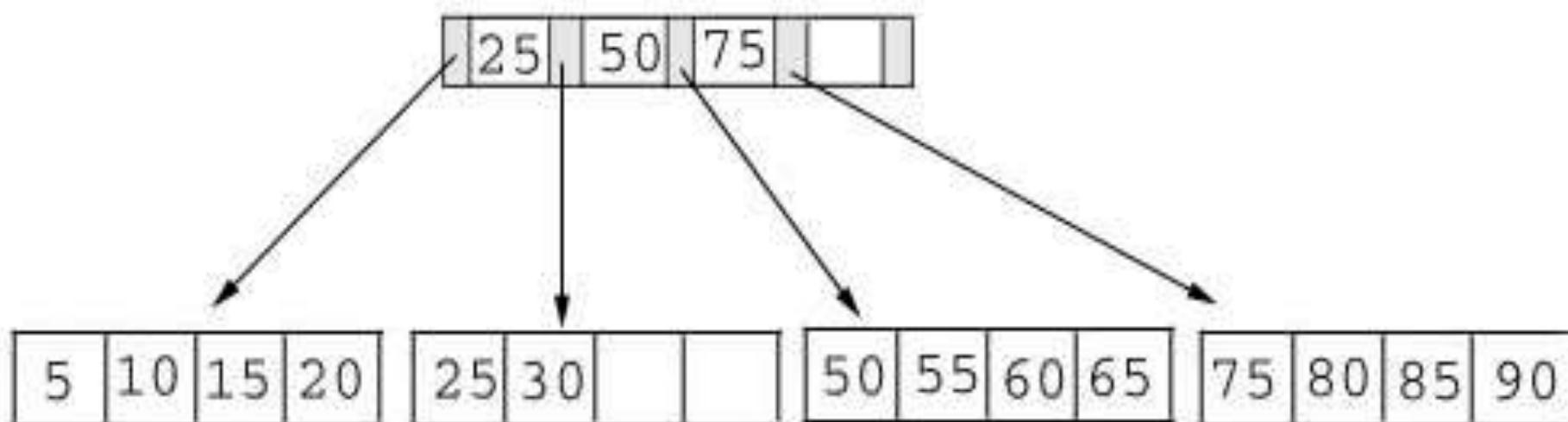
L	K_1	r_1	K_2	K_n	r_n	R
-----	-------	-------	-------	-------	-------	-------	-----

L, R : pointers to left and rights neighbours

More about B+ Tree

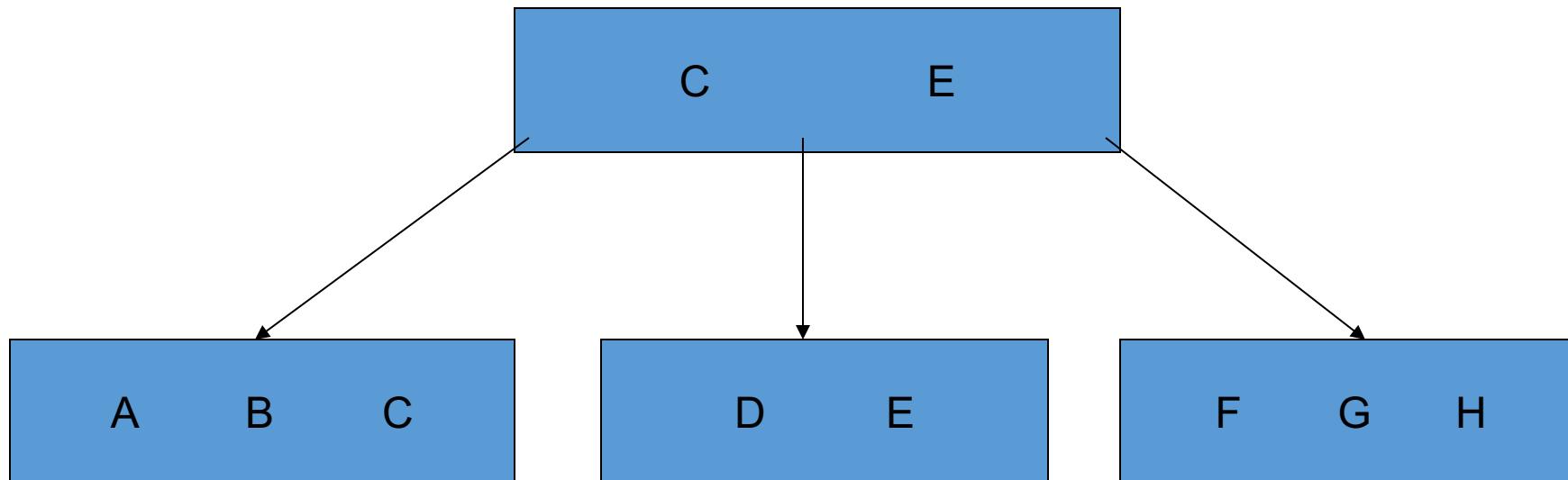
- ▶ Only leaf nodes contain data entries.
- ▶ Internal nodes contain indexes.
- ▶ Every data entry must appear in a leaf page.
- ▶ Leaf nodes are doubly linked to each other.

What is a B+ Tree



What is a B+ Tree

- ▶ Question: Is this a valid B+ Tree?



What is a B+ Tree

Answer:

1. Both trees in slide 3 and slide 4 are valid; how you store data in B+ Tree depends on your algorithm when it is implemented
2. As long as the number of data in each leaf are balanced, it doesn't matter how much data you stored in the leaves.

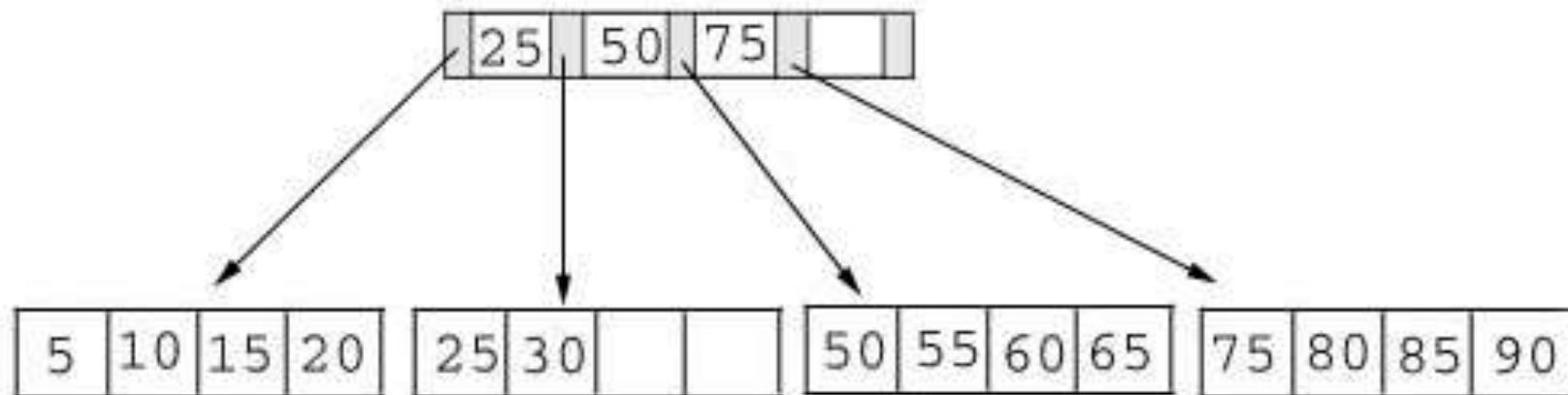
Searching

```
func tree_search (nodepointer, search key value K) returns nodepointer
    // Searches tree for entry
    if *nodepointer is a leaf, return nodepointer;
    else,
        if  $K < K_1$  then return tree_search( $P_0$ ,  $K$ );
        else,
            if  $K \geq K_m$  then return tree_search( $P_m$ ,  $K$ ); //  $m = \#$  entries
            else,
                find  $i$  such that  $K_i \leq K < K_{i+1}$ ;
                return tree_search( $P_i$ ,  $K$ )
    endfunc
```

Searching

- ▶ Since no structure change in a B+ tree during a searching process, so just compare the key value with the data in the tree, then give the result back.

For example: find the value **45**, and **15** in below tree.



Searching

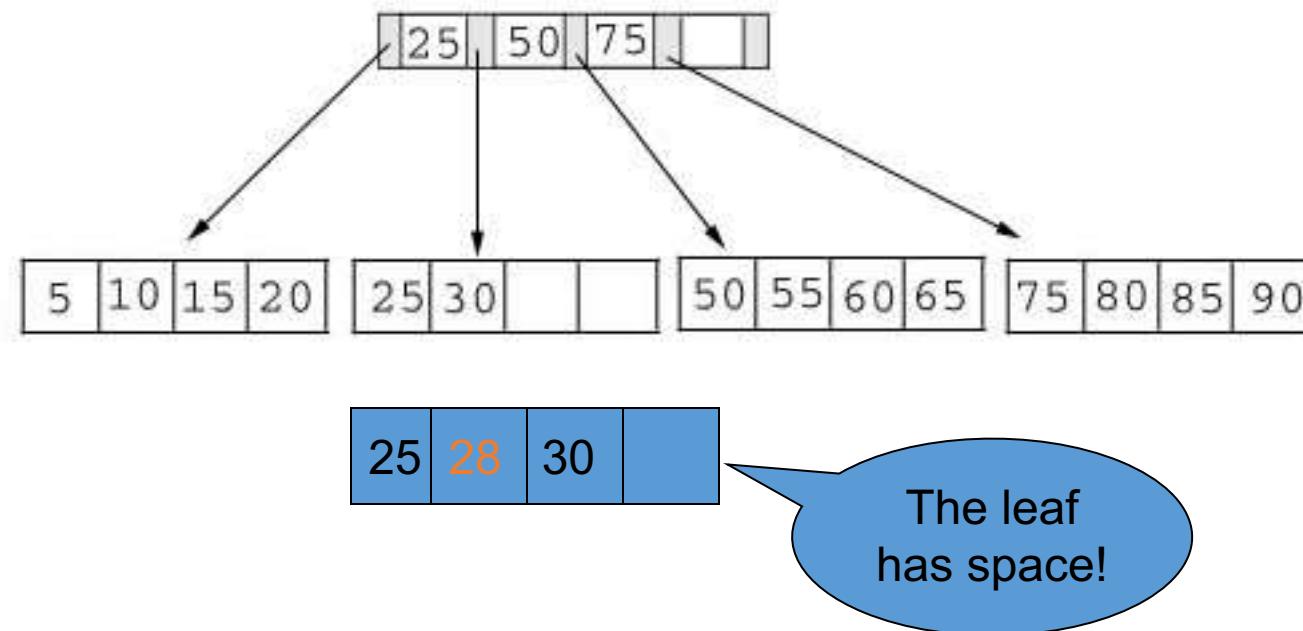
- Result:
 1. For the value of 45, not found.
 2. For the value of 15, return the position where the pointer located.

Insertion

- ▶ An entry is inserted at a leaf node that is searched from the root to the leaf node for an appropriate position.
- ▶ If there is not enough space in the leaf node, the node is split and the middle entry is copied into its parent.
- ▶ When an internal node is split, the middle entry is moved to the parent without copying it to the current node.

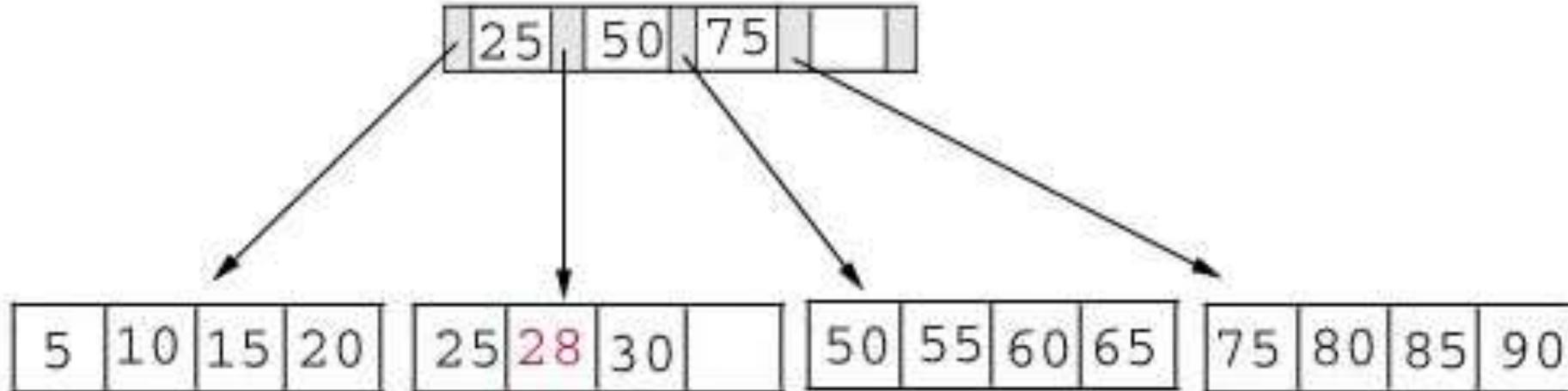
Insertion

- ▶ Since inserting a value into a B+ tree may cause the tree to be unbalanced, so rearrange the tree if needed.
- ▶ Example #1: insert **28** into the below tree.



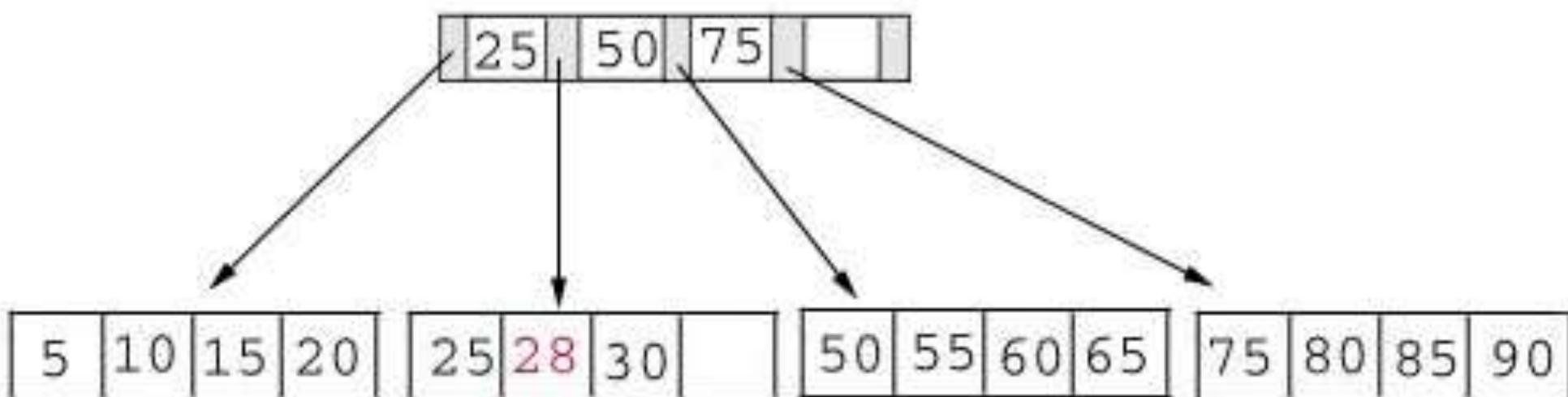
Insertion

- ▶ Result:



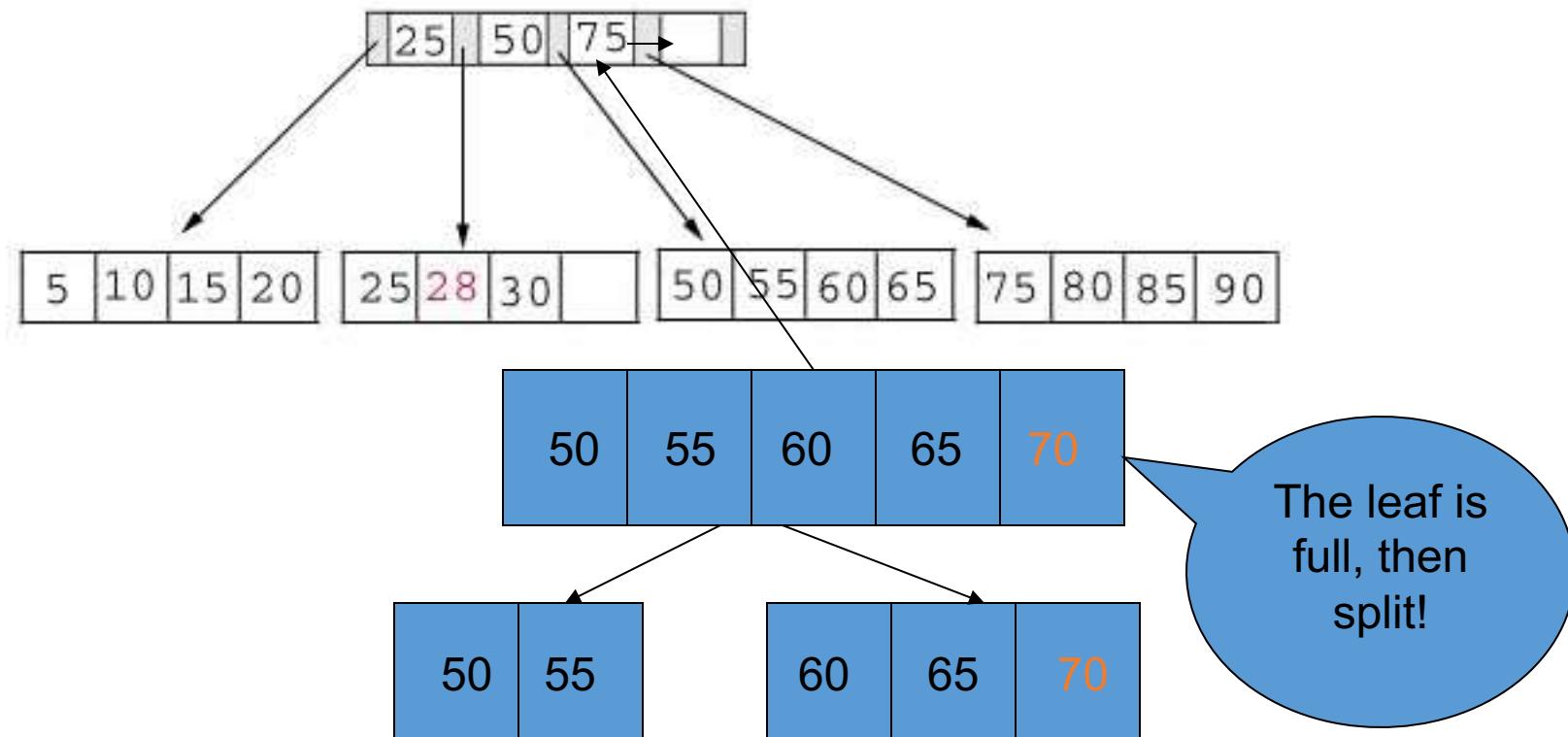
Insertion

- ▶ Example #2: insert **70** into below tree



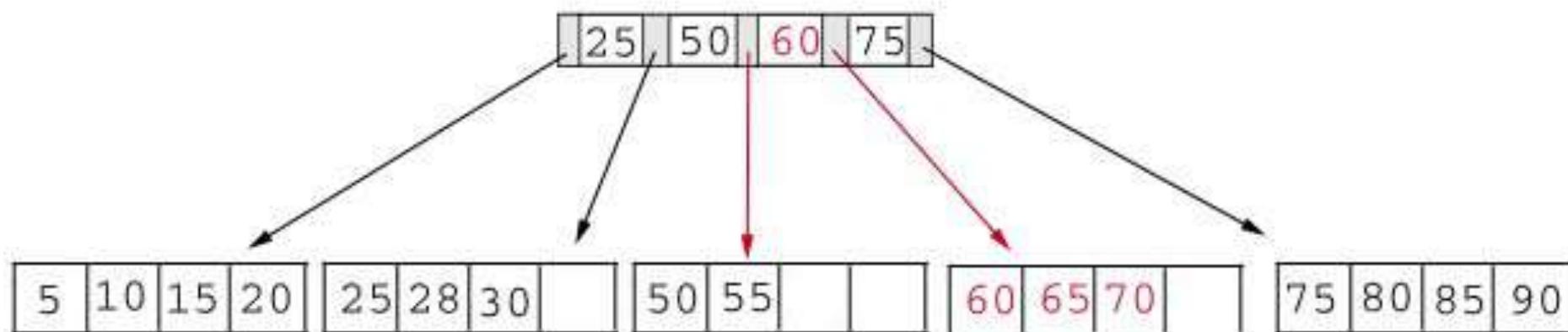
Insertion

- ▶ Process: split the tree



Insertion

- ▶ Result: chose the middle key 60, and place it in the index page between 50 and 75.



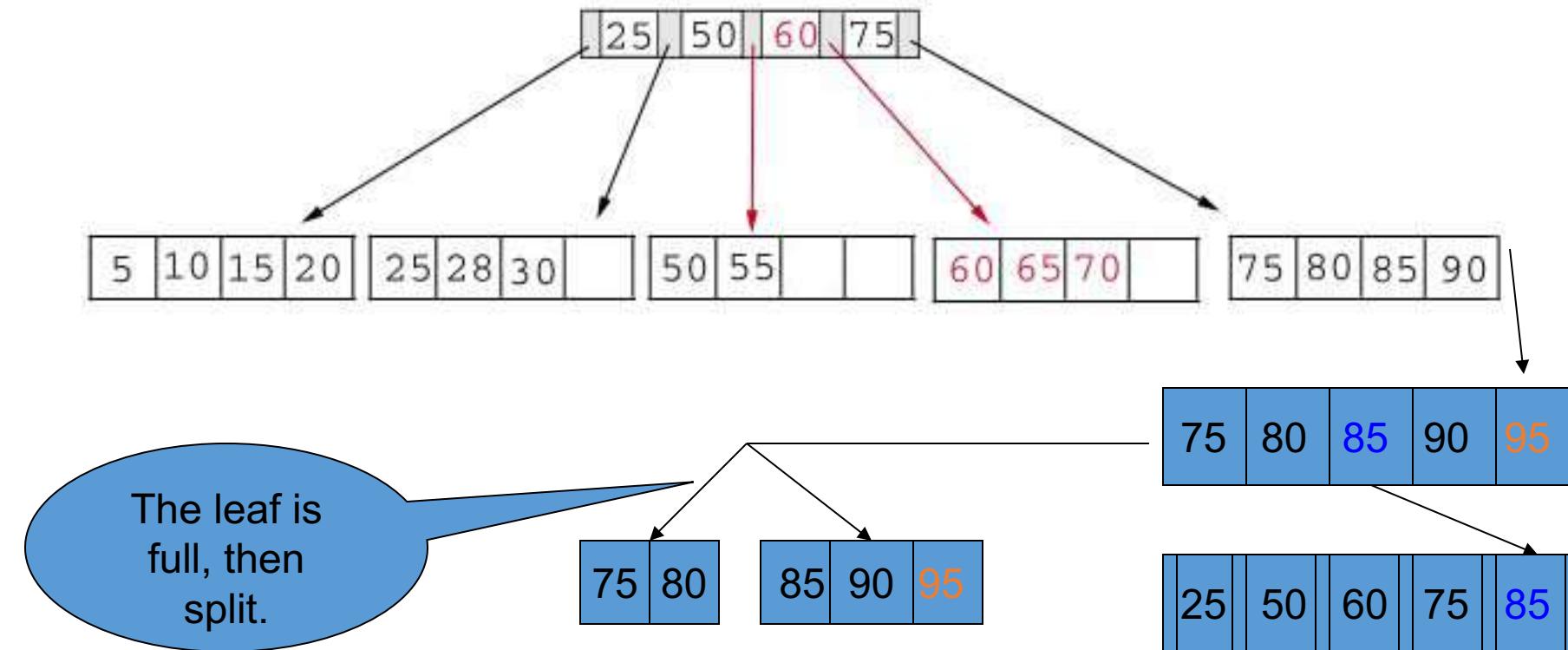
Insertion

The insert algorithm for B+ Tree

Data Page Full	Index Page Full	Action
NO	NO	Place the record in sorted position in the appropriate leaf page
YES	NO	<ol style="list-style-type: none">1. Split the leaf page2. Place Middle Key in the index page in sorted order.3. Left leaf page contains records with keys below the middle key.4. Right leaf page contains records with keys equal to or greater than the middle key.
YES	YES	<ol style="list-style-type: none">1. Split the leaf page.2. Records with keys < middle key go to the left leaf page.3. Records with keys \geq middle key go to the right leaf page.4. Split the index page.5. Keys < middle key go to the left index page.6. Keys $>$ middle key go to the right index page.7. The middle key goes to the next (higher level) index. <p>IF the next level index page is full, continue splitting the index pages.</p>

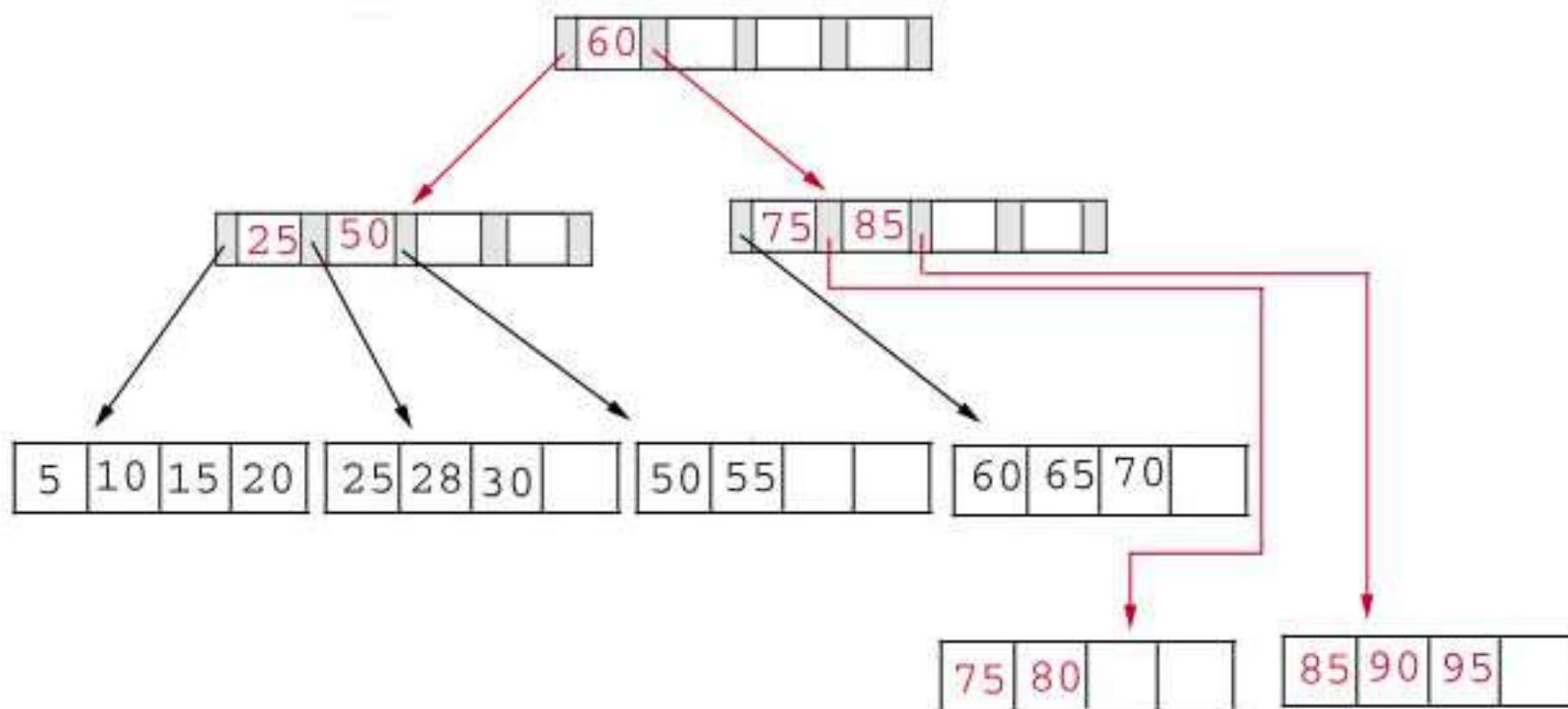
Insertion

- Exercise: add a key value **95** to the below tree.



Insertion

- Result: again put the middle key 60 to the index page and rearrange the tree.

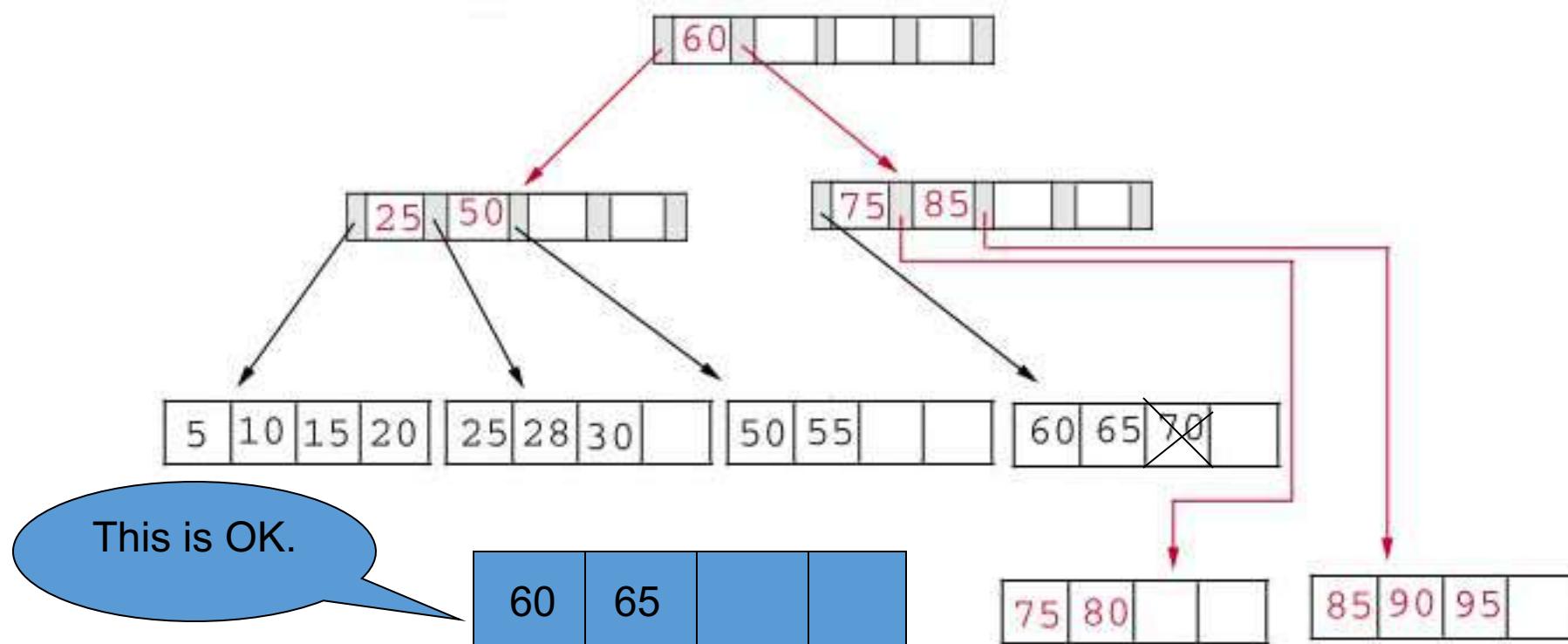


Deletion

- ▶ We go down to the leaf node where the entry belongs.
- ▶ Delete the entry. If it has too few entries in the leaf node, either redistribute the entries to the sibling nodes or merge the node with a sibling node.
- ▶ If entries are redistributed, their parent node must be updated to reflect this; the key value in the index entry pointing to the second node must be changed to the lowest search key in the second node.
- ▶ If two nodes are merged, their parent must be updated to reflect this by deleting the index entry for the second node.

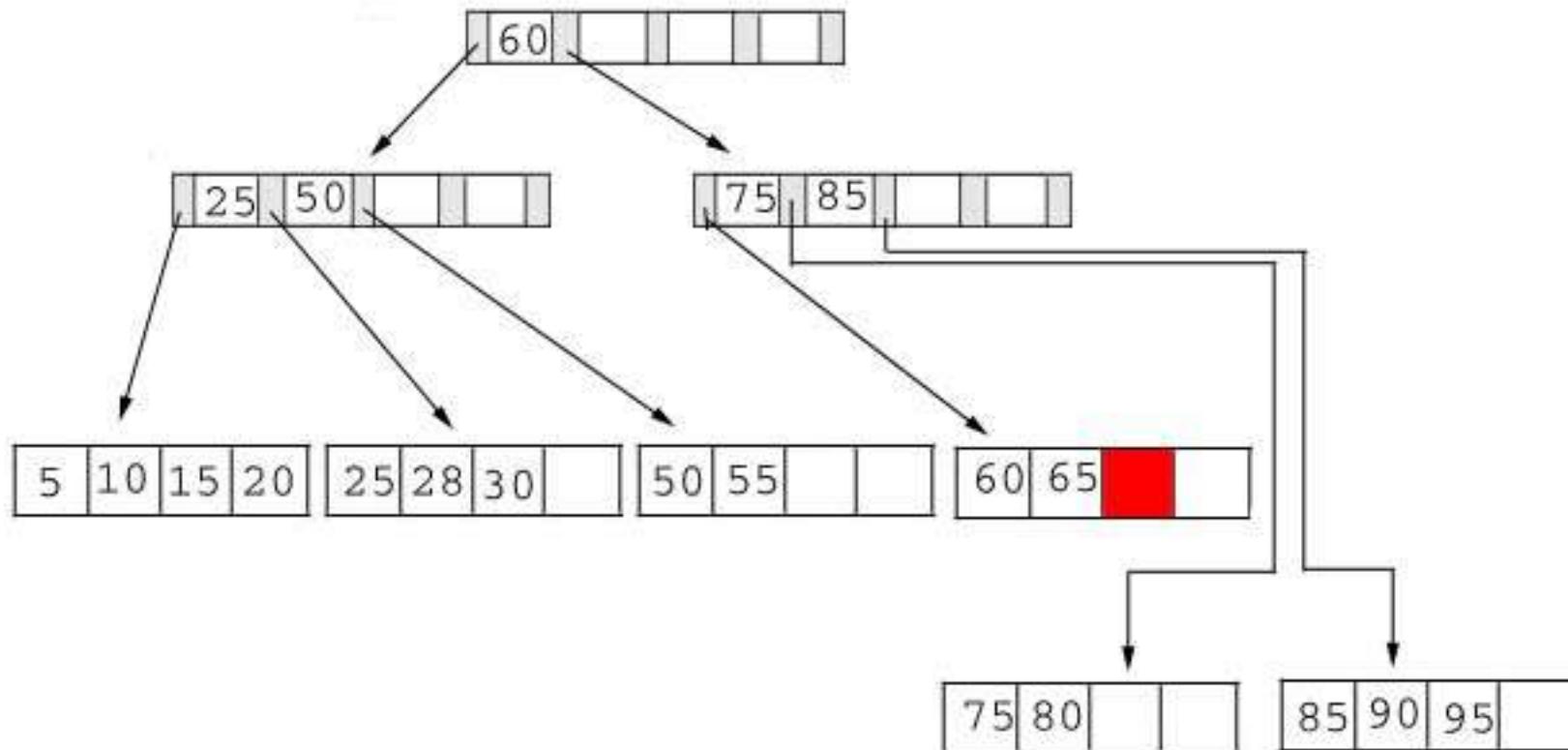
Deletion

- ▶ Same as insertion, the tree has to be rebuilt if the deletion result violates the rule of B+ tree.
- ▶ Example #1: delete **70** from the tree



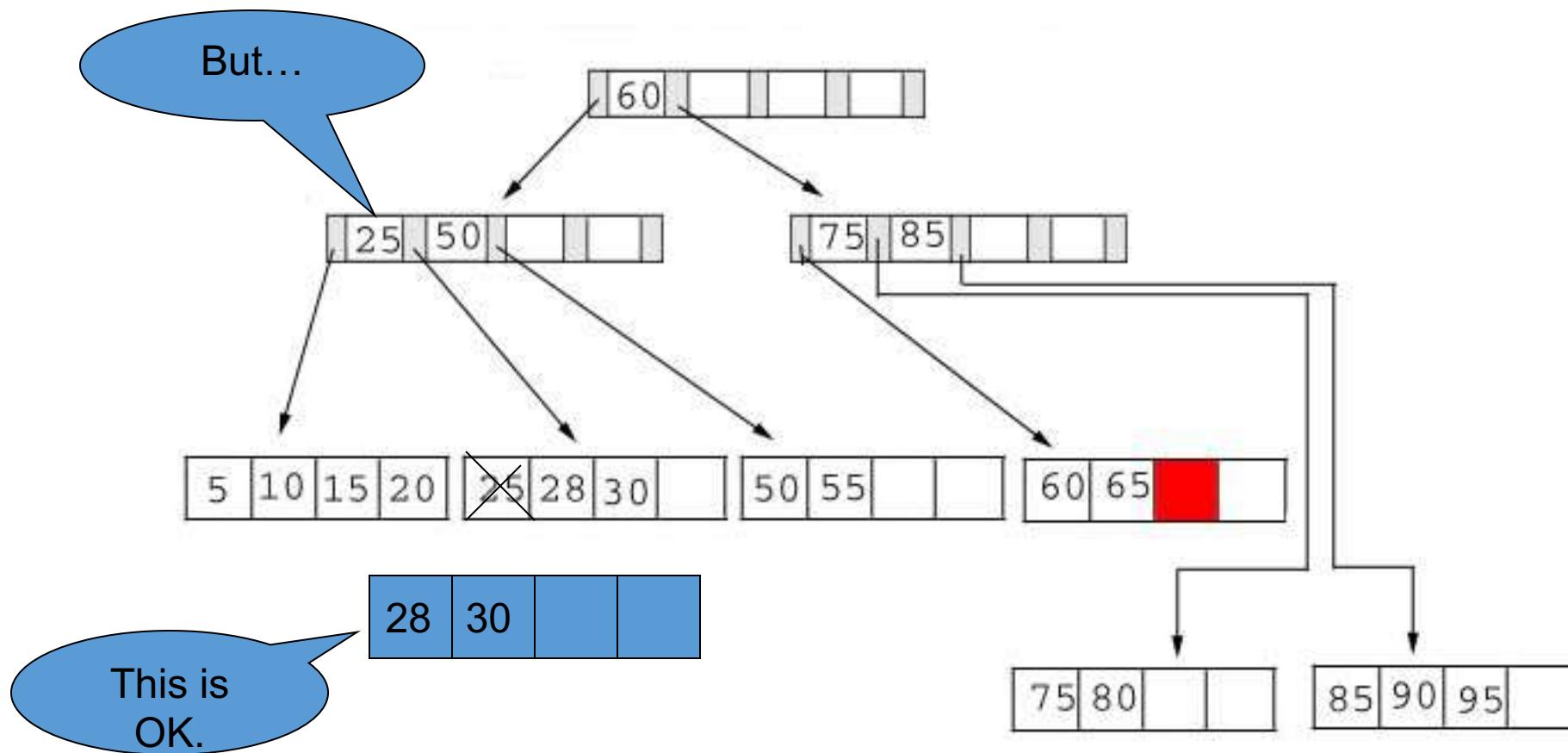
Deletion

► Result:



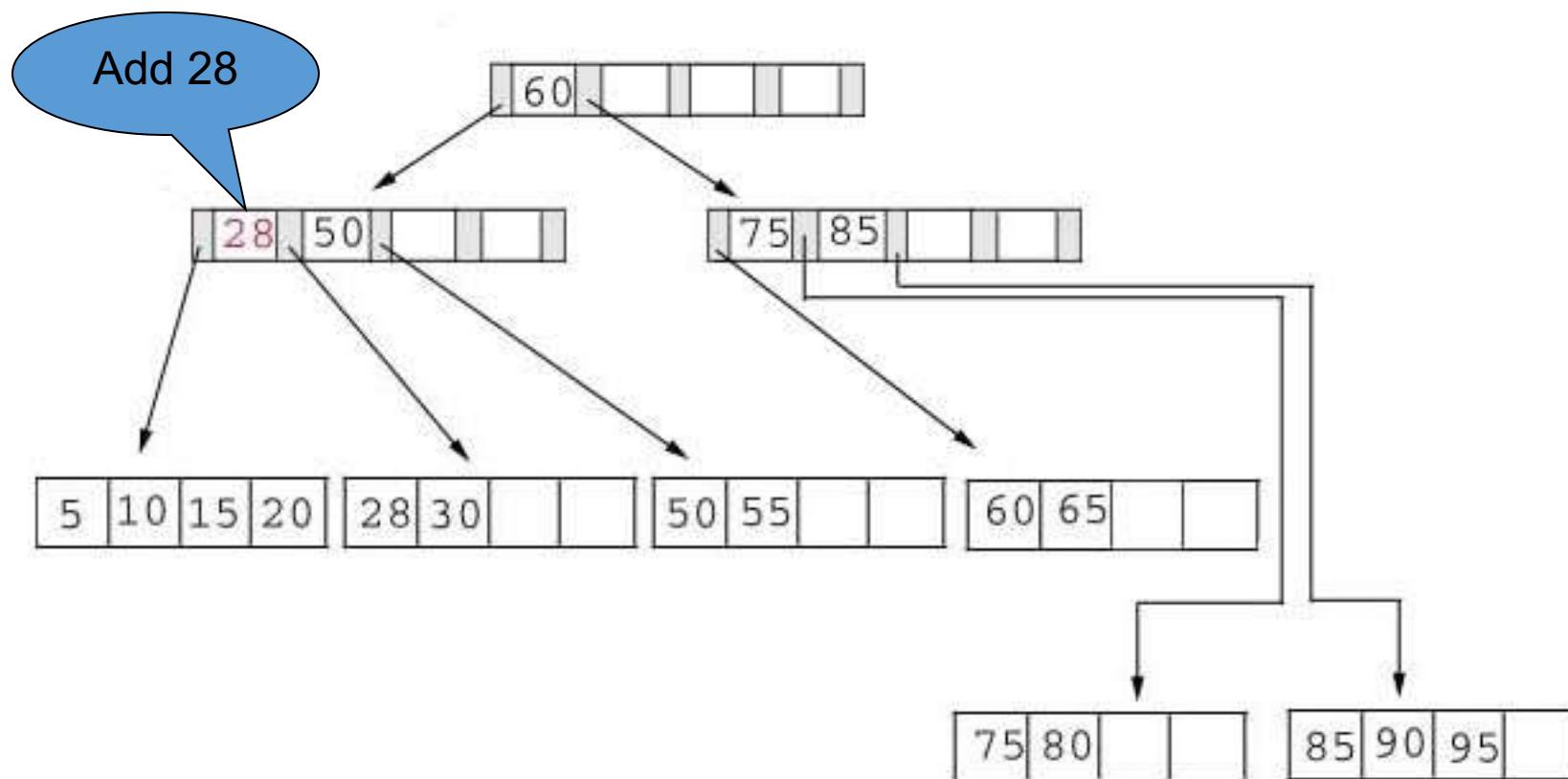
Deletion

Example #2: delete **25** from below tree, but **25** appears in the index page.



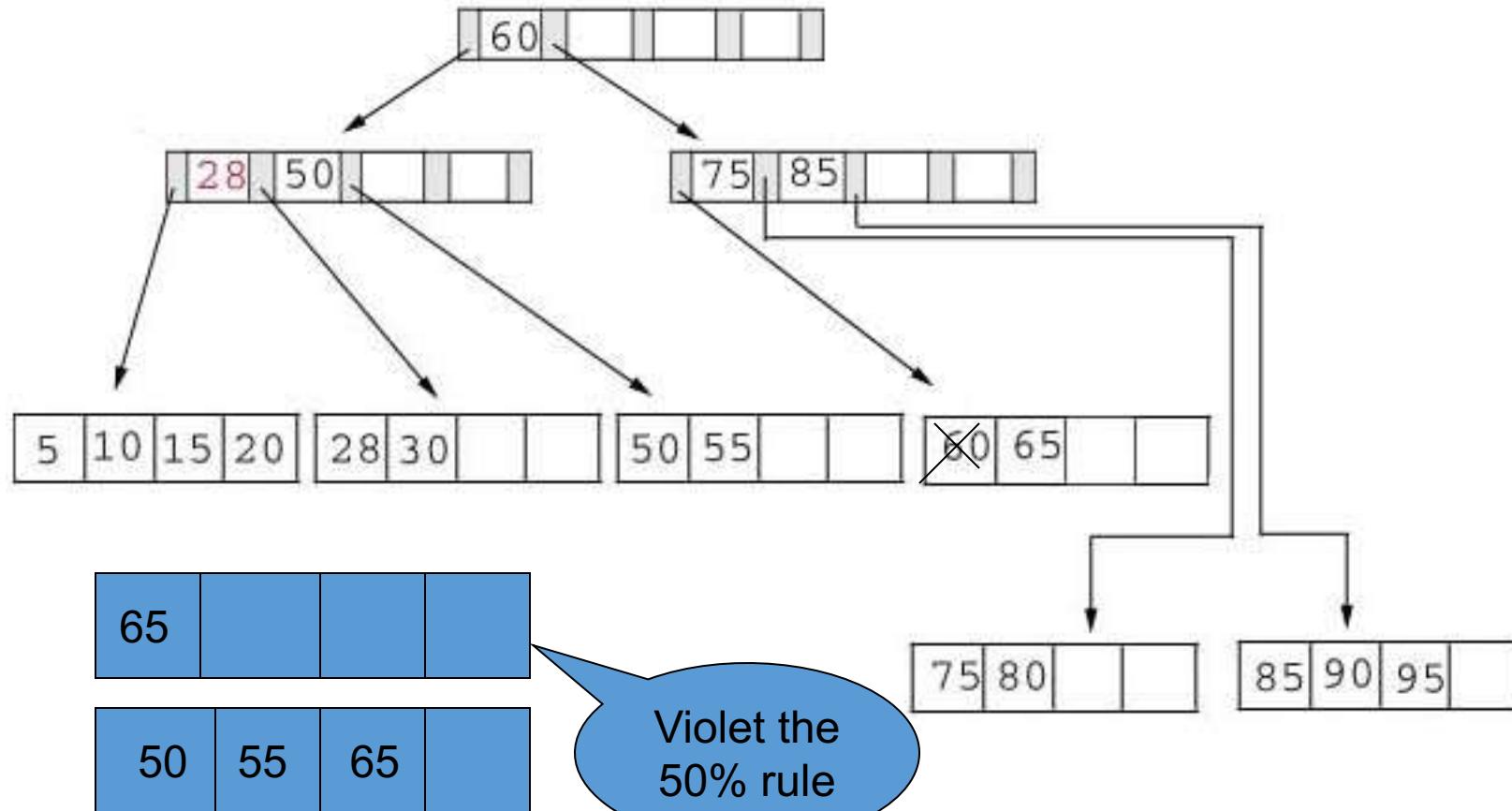
Deletion

- ▶ Result: replace **28** in the index page.



Deletion

- ▶ Example #3: delete **60** from the below tree



Deletion

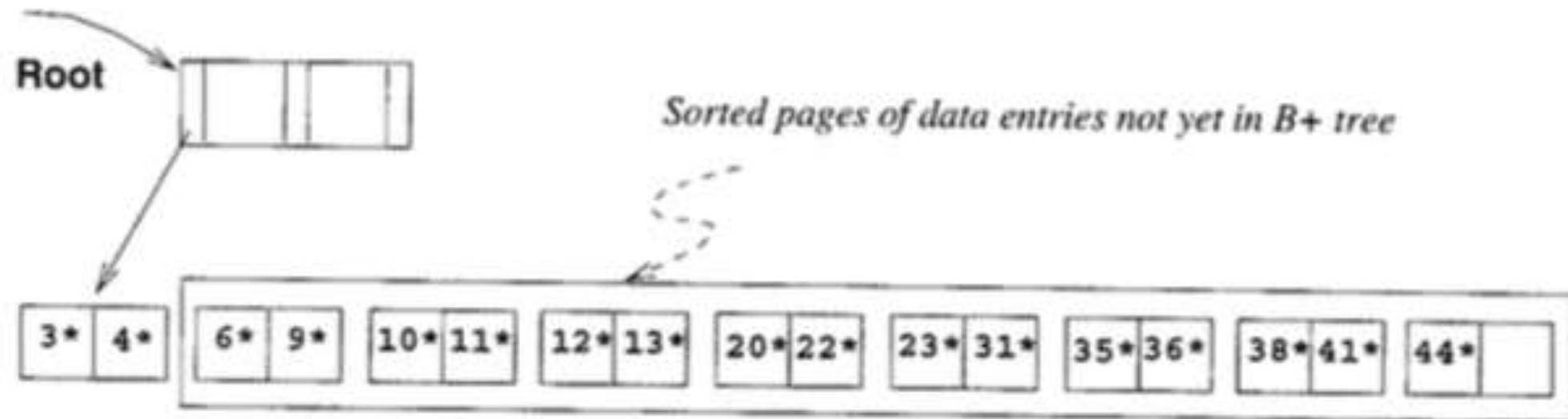
► Delete algorithm for B+ trees

Data Page Below Fill Factor	Index Page Below Fill Factor	Action
NO	NO	Delete the record from the leaf page. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index page, use the next key to replace it.
YES	NO	Combine the leaf page and its sibling. Change the index page to reflect the change.
YES	YES	<ol style="list-style-type: none">1. Combine the leaf page and its sibling.2. Adjust the index page to reflect the change.3. Combine the index page with its sibling. <p>Continue combining index pages until you reach a page with the correct fill factor or you reach the root page.</p>

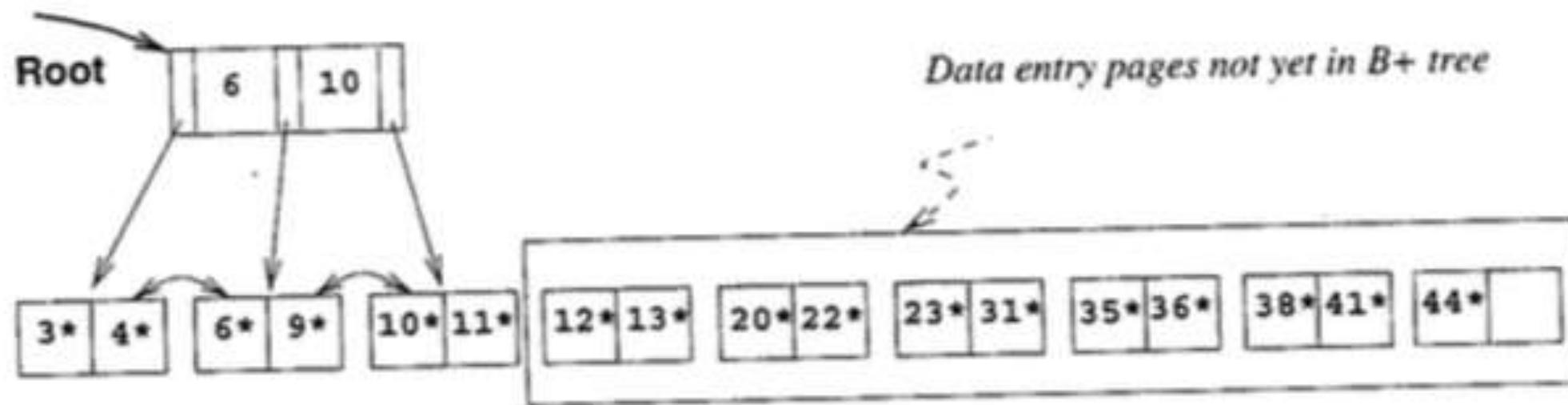
Bulk Loading in B+ Tree

- ▶ Inserting entries one by one is expensive because it requires searching from the root till the leaf node at each insertion.
- ▶ Instead, all the data to be inserted is sorted and inserted as a collection of data entries.

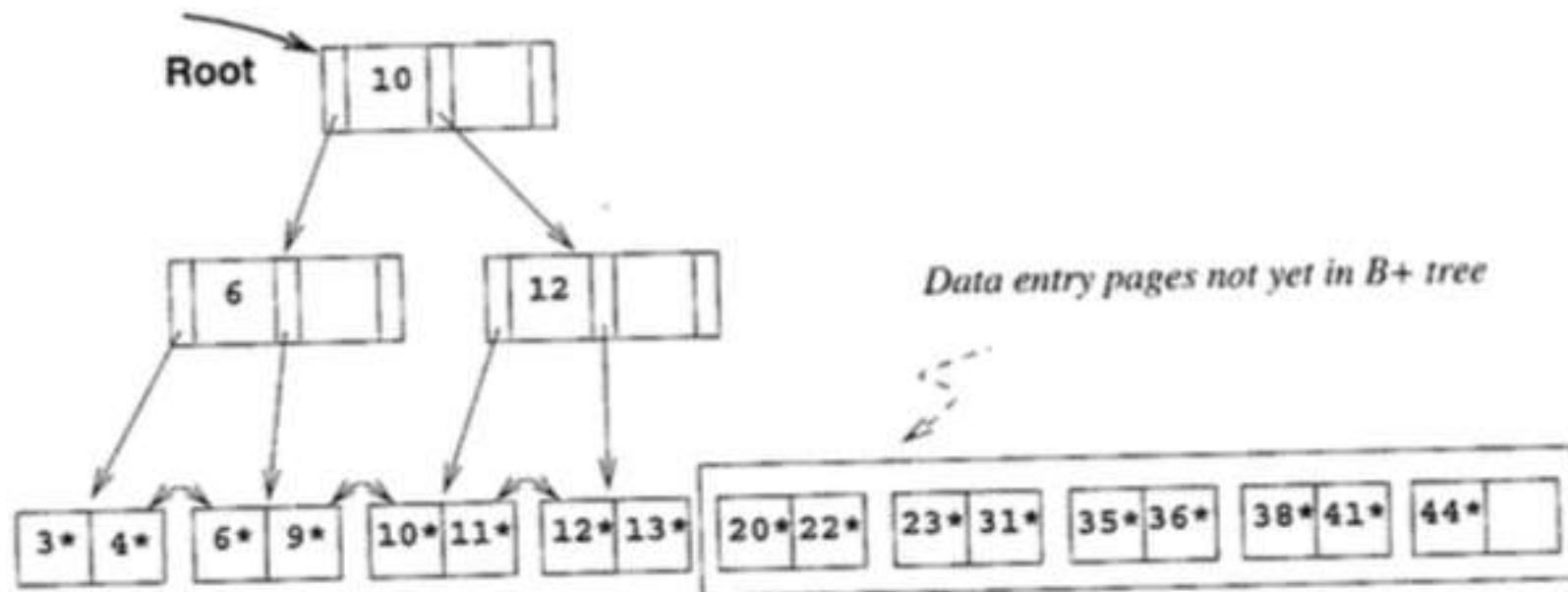
Bulk Loading in B+ Tree



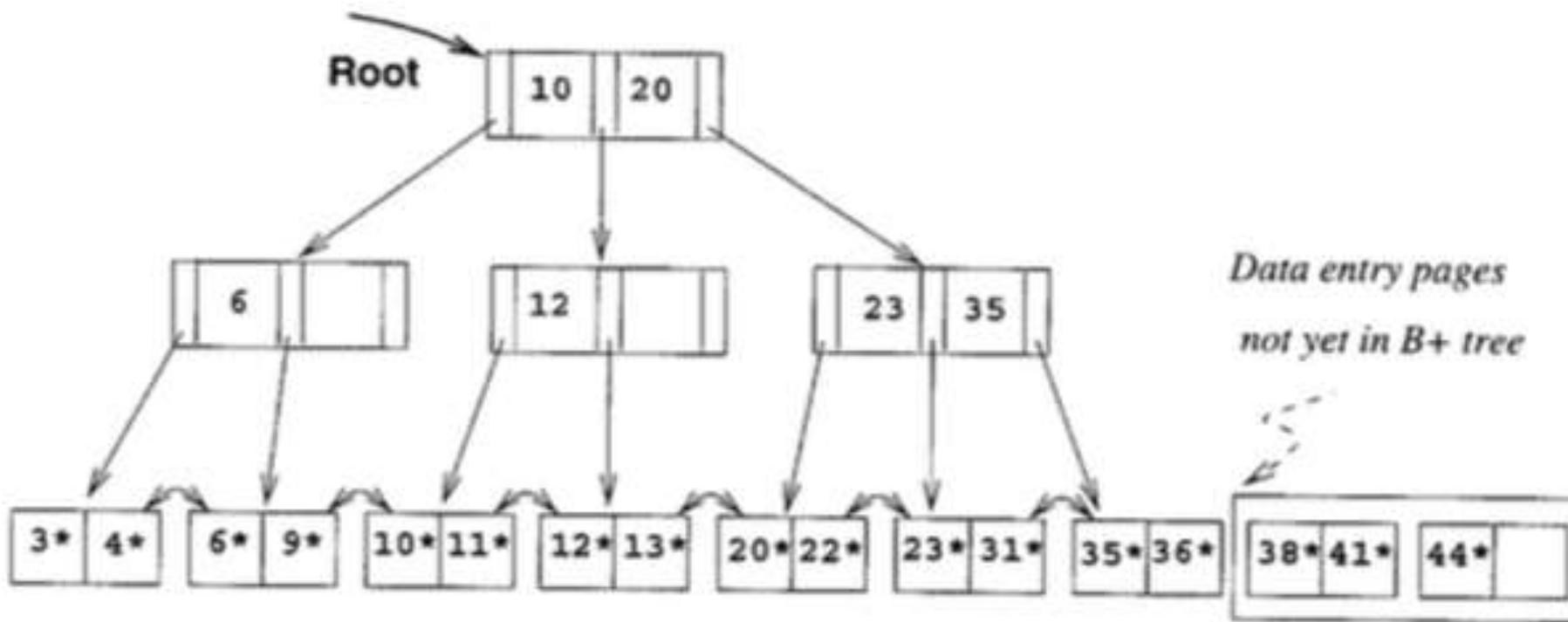
Bulk Loading in B+ Tree



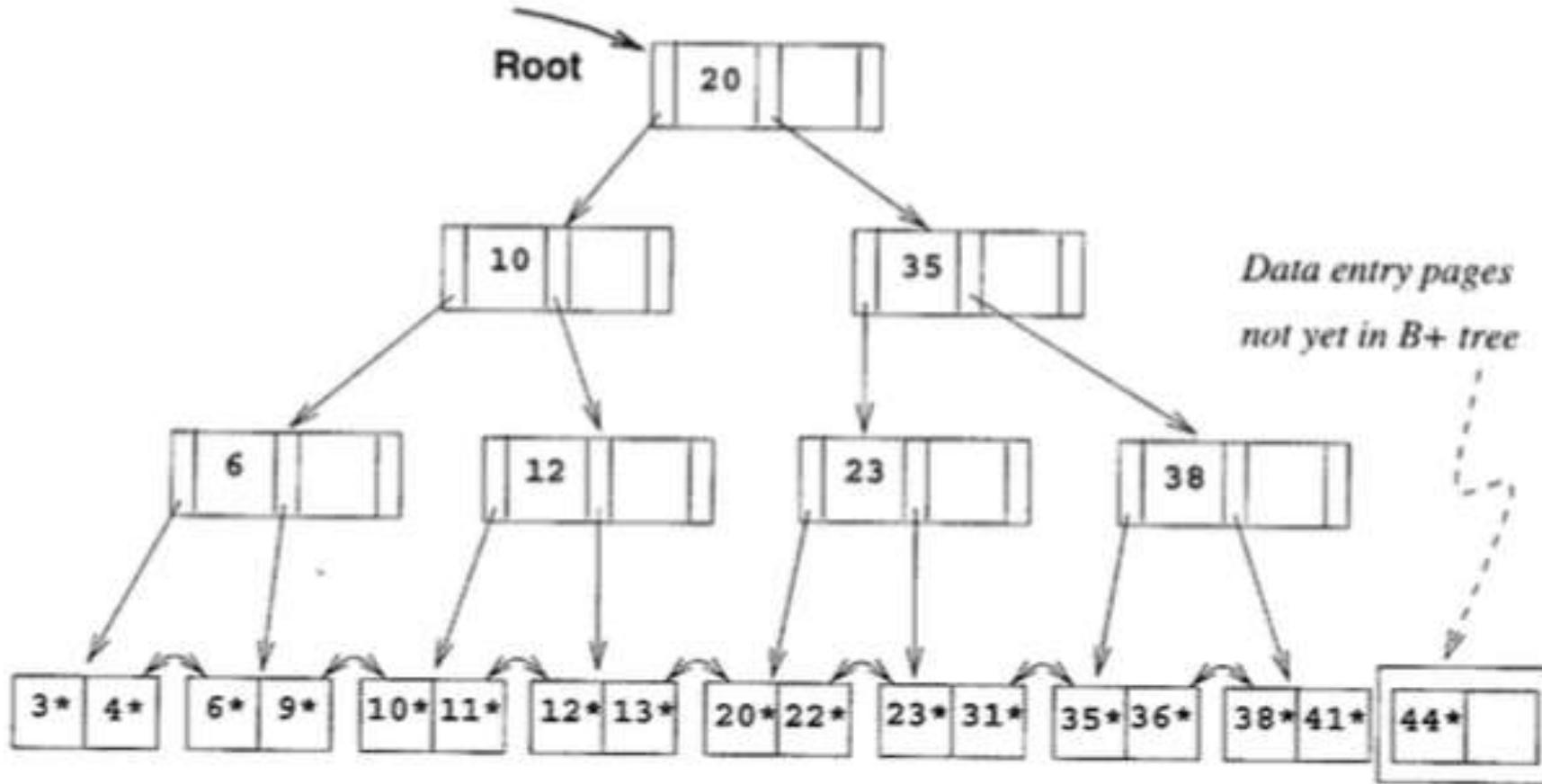
Bulk Loading in B+ Tree



Bulk Loading in B+ Tree



Bulk Loading in B+ Tree





BBM371- Data Management

Lecture 10 External Sorting

18.12.2019

Why Sort in Databases

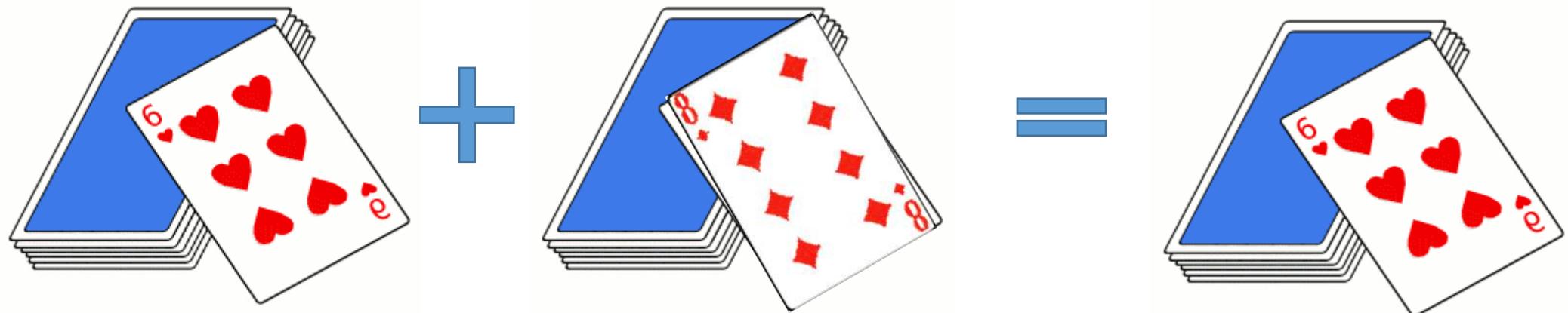
- ▶ Data can be requested in sorted order
 - ▶ `SELECT * FROM Foo ORDER BY bar`
- ▶ Loading data to an index. Bulk Loading in B+ Tree
- ▶ De-duplication (for example `DISTINCT` keyword in SQL). Remove duplicates by first ordering the records.
- ▶ Other query related operations such as joining multiple tables
- ▶ So, we will need to sort the keys, records for different needs

In-memory sorting vs. External Sorting

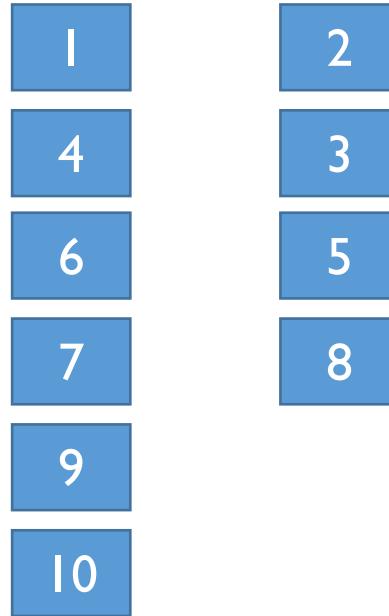
- ▶ You have seen sorting algorithms in your previous courses
- ▶ Why is it different for databases?
- ▶ The data can be much larger than the memory. In this case we have to think about the number of disk accesses.
- ▶ Running a sorting algorithm on disk will result in many record swaps that must be performed as disk operations. Will not work!
- ▶ We must design an algorithm which uses the limited primary memory wisely, and minimizes the disk accesses.
- ▶ Problem: Sort 1GB data with 1MB memory

Merge of Merge Sort

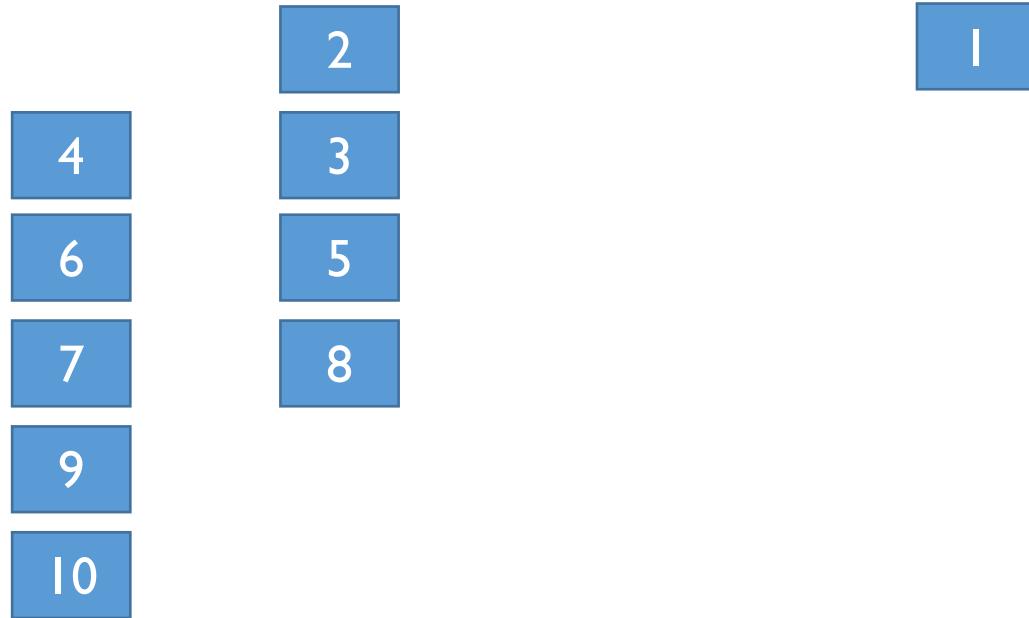
- ▶ Merge Sort algorithm is characterized by the merge operation
- ▶ Given two sorted lists, merge them to produce a single sorted list
- ▶ You can visualize the algorithm as merging two sorted deck of cards.
- ▶ Pick the smallest card (on top) from both decks.
- ▶ Since we know that the smaller card is not in the other deck, we can add it to the output deck.



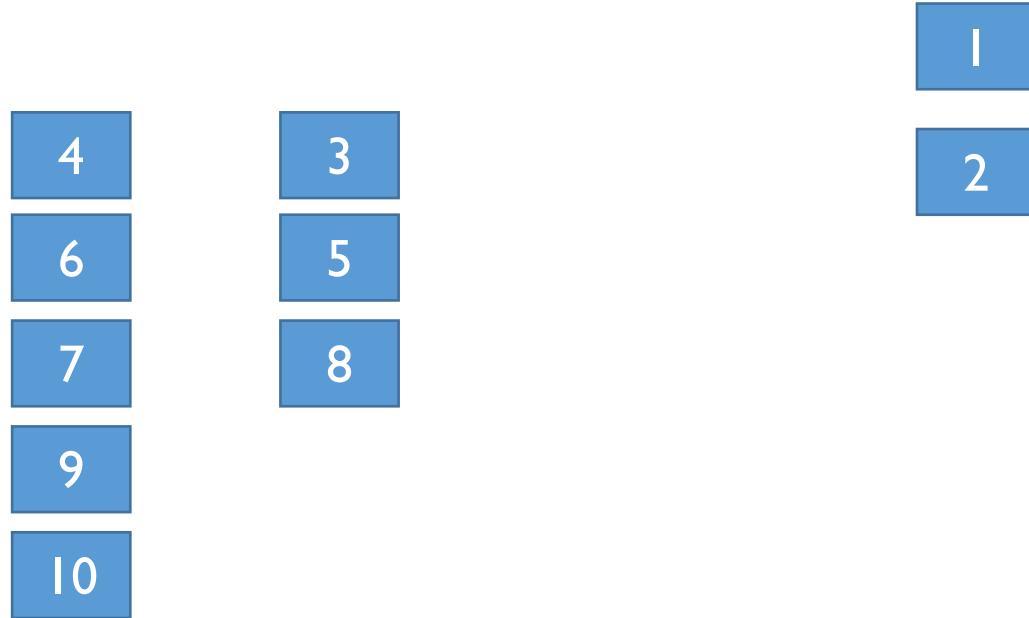
Merge Sort Illustration



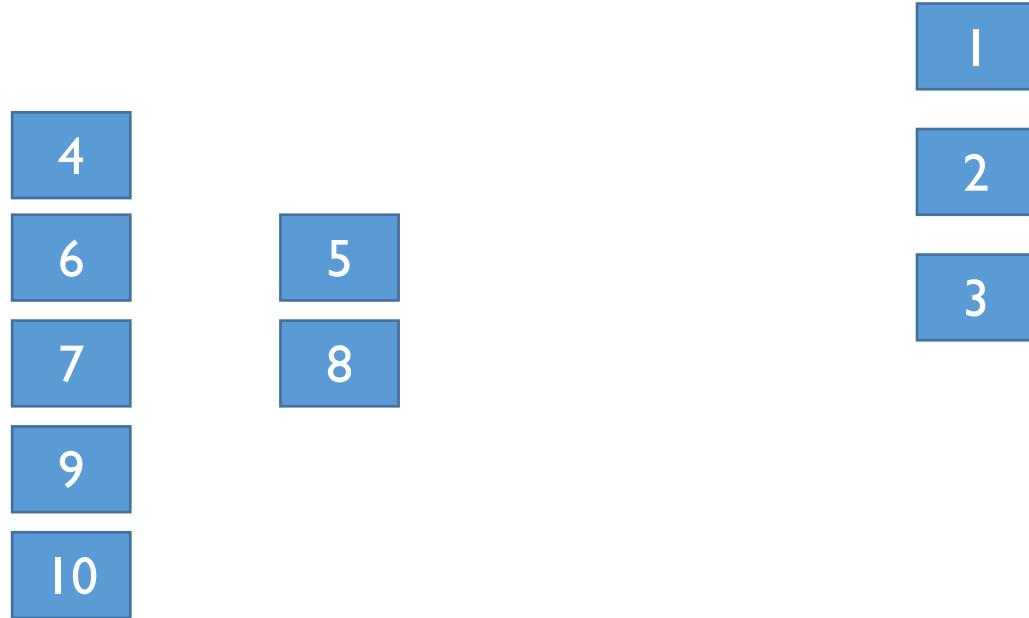
Merge Sort Illustration



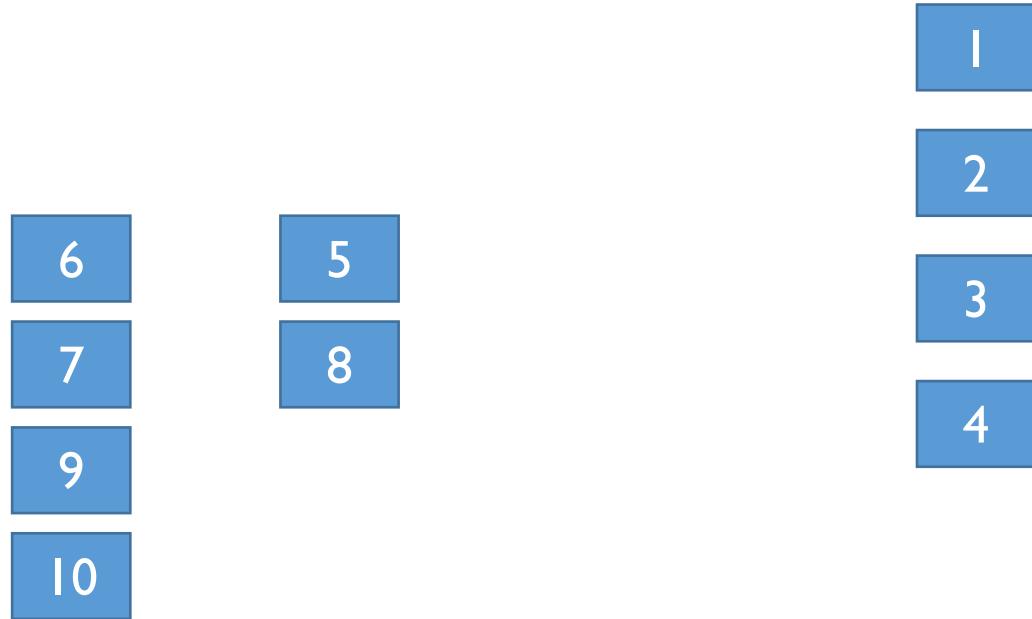
Merge Sort Illustration



Merge Sort Illustration



Merge Sort Illustration



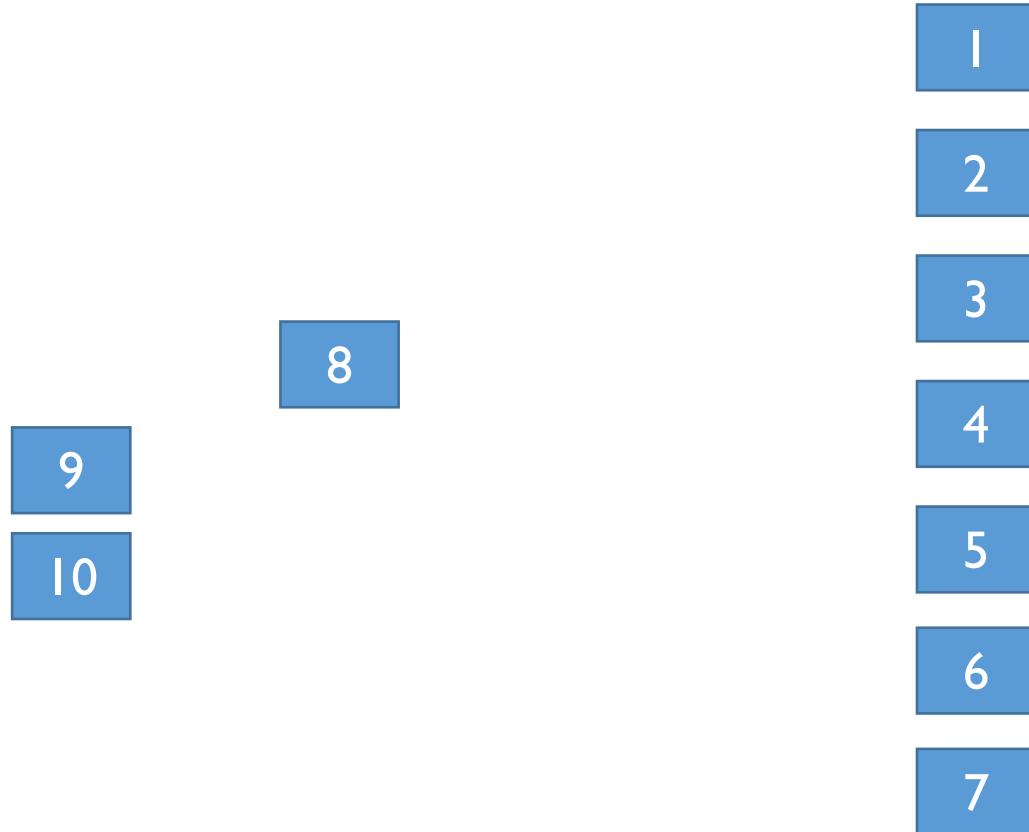
Merge Sort Illustration



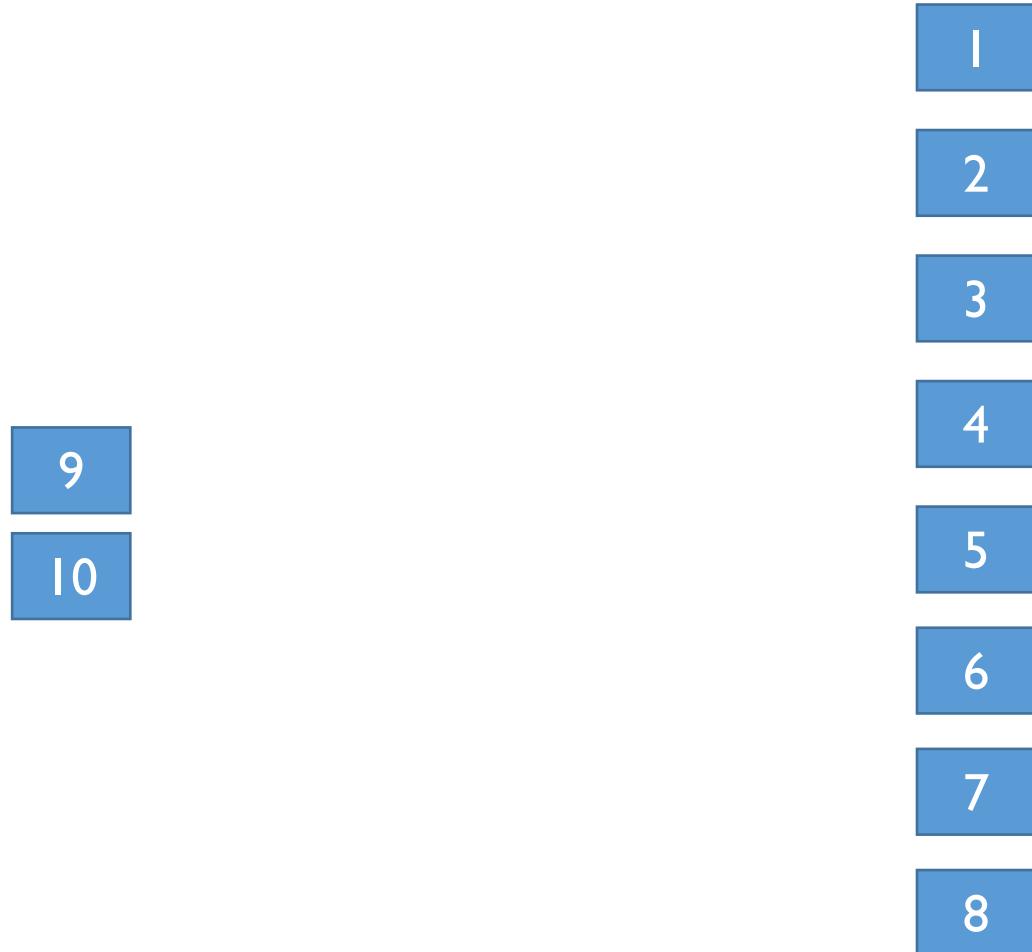
Merge Sort Illustration



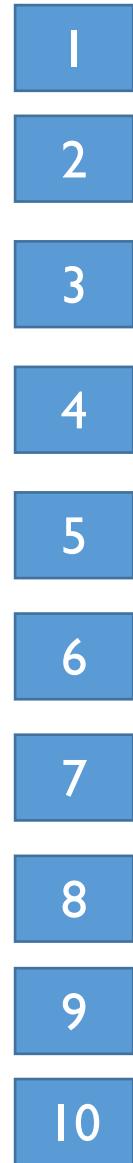
Merge Sort Illustration



Merge Sort Illustration



Merge Sort Illustration

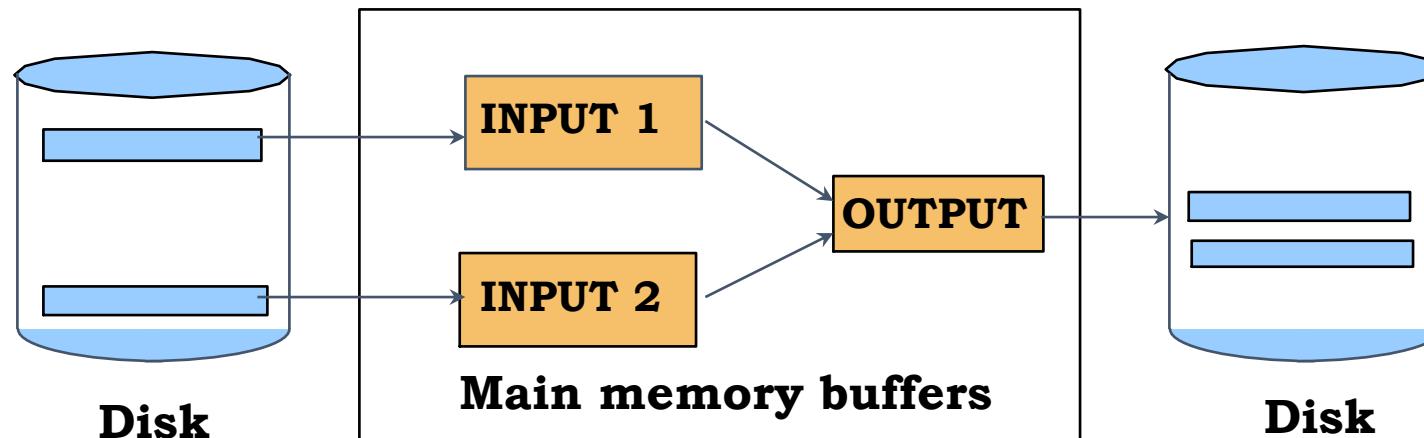


General Idea

- ▶ Use the memory for sorting a part of the file
 - ▶ We will first consider sorting a page in memory
- ▶ Use the merge operation to merge runs until the whole file is sorted!
- ▶ The merge can be implemented for any two sublists of arbitrarily large sizes, as we only need the top of the decks

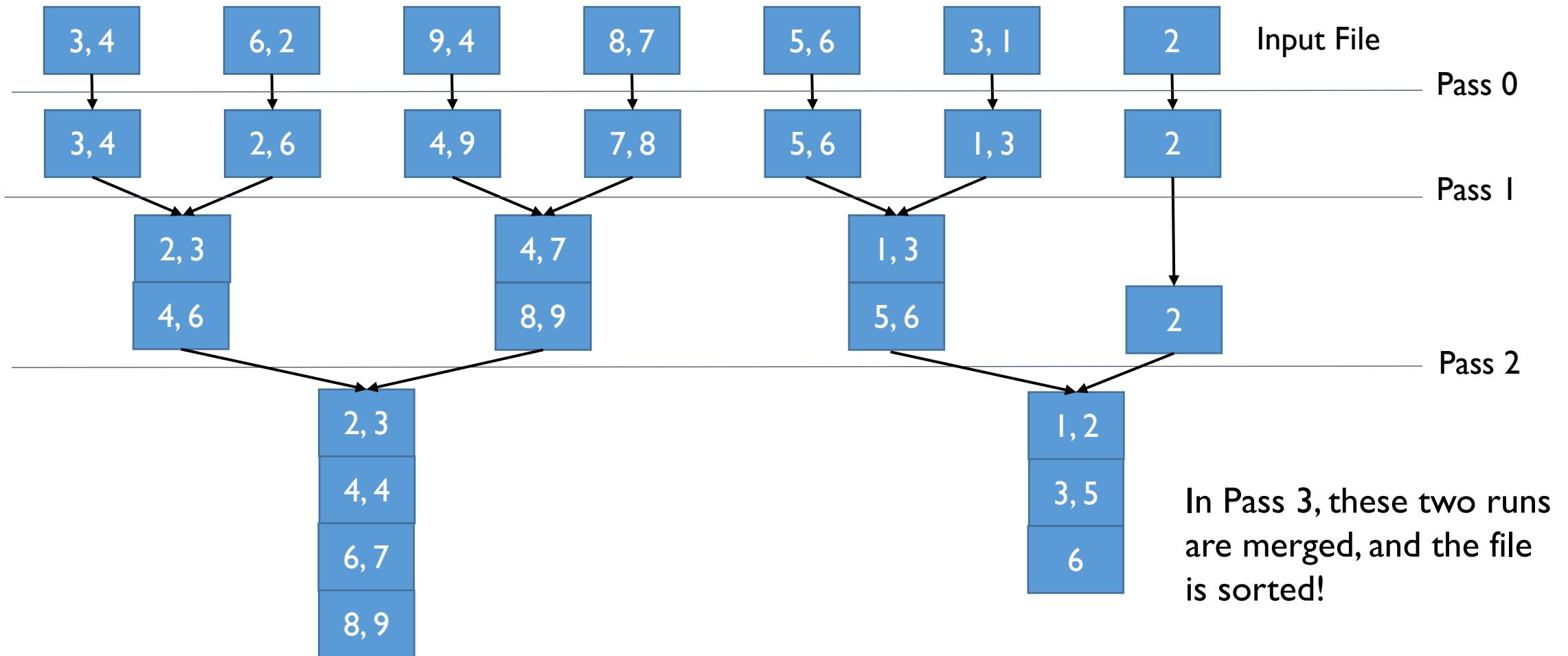
2-Way Sort

- ▶ Read a page, sort it, write it
 - ▶ Only one buffer page is used
 - ▶ Written sorted page is called as a run
- ▶ Pass 2, 3 ... etc:
 - ▶ 3 page memory buffer: 2 for reading runs, Merge & Write in one page



2-Way Merge Example

- ▶ Assume that we have 7 pages in disk.
- ▶ Each page can store 2 keys



2-Way Merge Sort Algorithm

- ▶ proc 2-way-extsort (file)
 - ▶ Read each page into memory, sort it, write it out //Produce runs that are one page long – Pass 0
 - //Pass i=1,2, ...
 - ▶ While the number of runs at end of previous pass is > 1
 - ▶ While there are runs to be merged from previous pass:
 - ▶ Choose next two runs (from previous runs)
 - ▶ Read each run into an input buffer; page at a time
 - ▶ Merge the runs and write to the output buffer
 - ▶ Force output buffer to disk one page at a time
- endproc

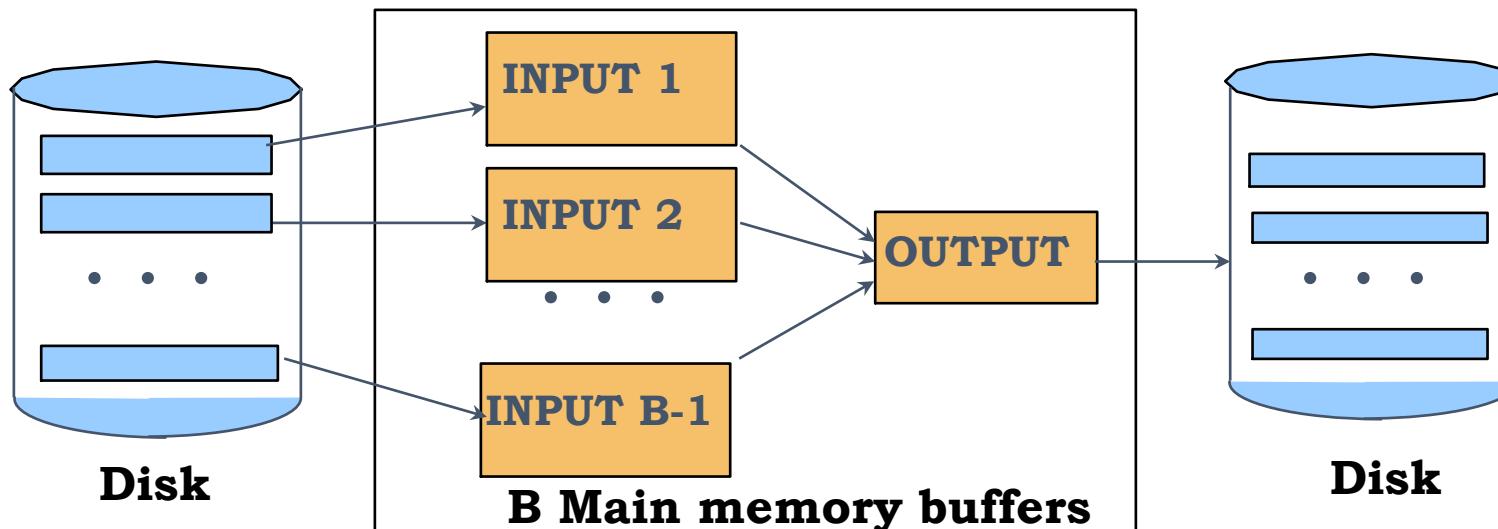
Requires just three buffer pages!

2-Way Merge Analysis

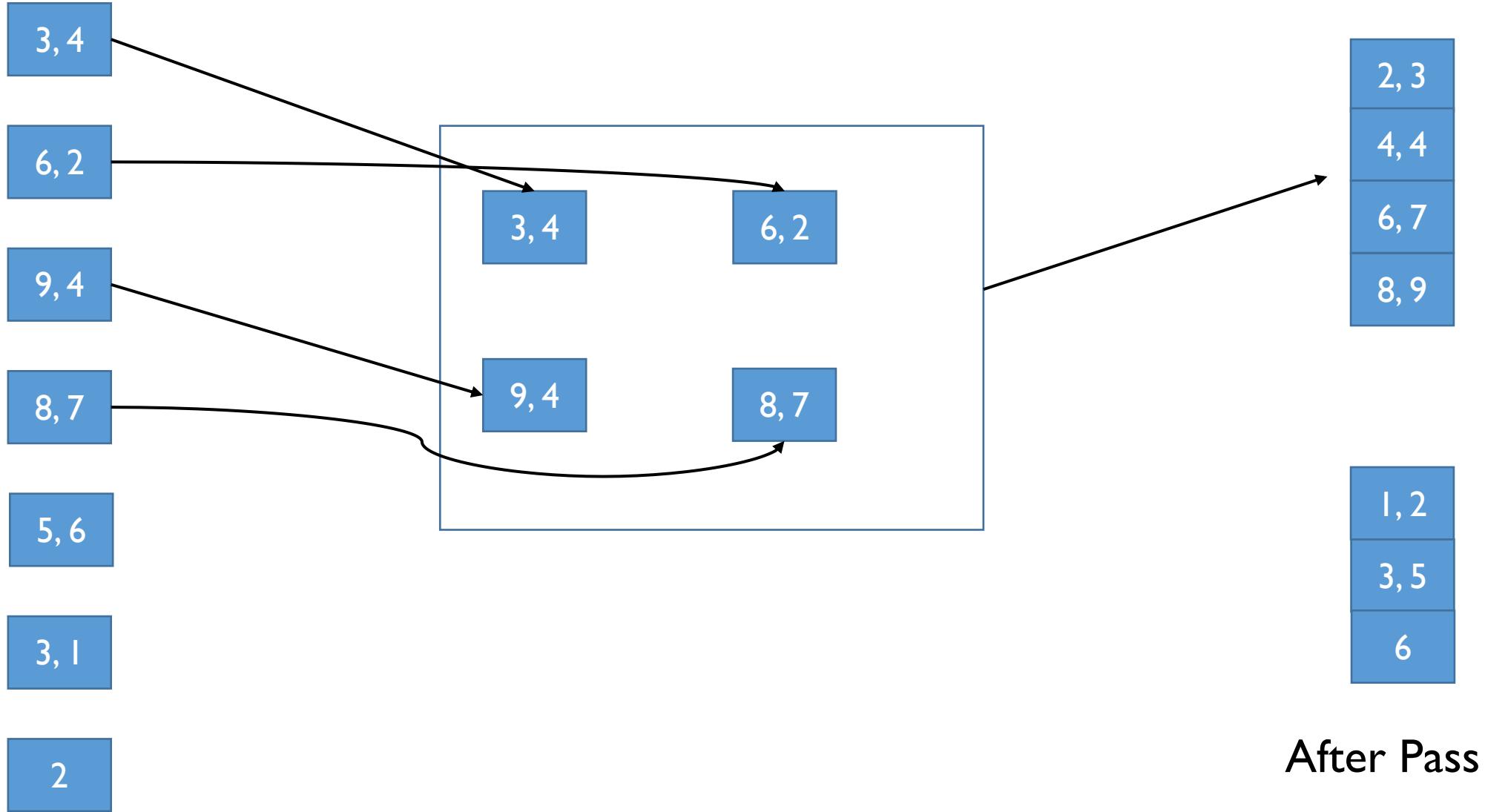
- ▶ The number of passes needed for sorting a file of 2^k is k
- ▶ At each step we are merging two runs. So $\text{ceil}(\log_2 N)$, where N is the number of pages in the file. If we add the initial Pass, $\text{ceil}(\log_2 N) + 1$
- ▶ In each pass we have to read and write each page. So, the cost for a pass is $2N$
- ▶ The total cost of this procedure is $2N * \text{ceil}(\log_2 N + 1)$
- ▶ So for our example, with 7 pages. We have 4 passes. At each pass we have $2 * 7$ disk access. The total cost is 56 disk accesses.

How to do better?

- ▶ The complexity increases as the number of passes increases.
- ▶ Instead of merging two runs at a time, we will merge as much runs as it is possible
 - ▶ Number of runs that can fit in the memory
- ▶ Instead of 2 we use B pages to read to memory (e.g. $B=4$ pages)
 - ▶ Pass 0 : Create $\text{ceil}(N/B)$ runs (N : number of pages in the file)
 - ▶ Pass $1 \dots k$: Merge $B-1$ runs (1 page is used for the output)



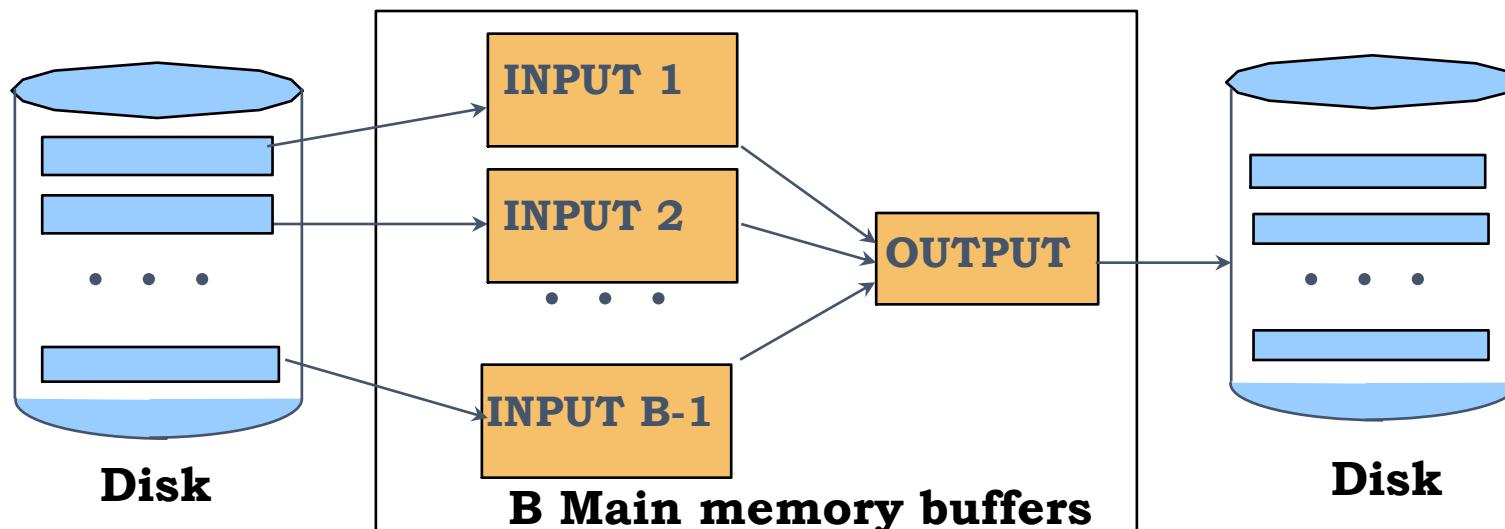
First Sort (Pass 0) B=4



After Pass 0

B-1 Way Merge

- ▶ After this step, we can merge $B-1=3$ different runs at the same time
- ▶ Algorithm for B-1 Way Merge is similar to two-way case:
 - ▶ At each step choose the smallest key
 - ▶ Write to output



Cost of External Merge Sort

- ▶ Number of passes:
- ▶ Cost = $2N * (\# \text{ of passes}) = 1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- ▶ E.g., with 5 buffer pages, to sort 108 page file:
 - ▶ Pass 0: $\lceil 108 / 5 \rceil = 22$ sorted runs of 5 pages each (last run is only 3 pages)
 - ▶ Pass 1: $\lceil 22 / 4 \rceil = 6$ sorted runs of 20 pages each (last run is only 8 pages)
 - ▶ Pass 2: 2 sorted runs, 80 pages and 28 pages
 - ▶ Pass 3: Sorted file of 108 pages

A note on complexity

- ▶ The asymptotic complexity of both algorithms is $O(N \log N)$
- ▶ The base of logarithm can be changed by the rule

$$\log_a b = \log_c b / \log_c a$$

So a different base changes only the constant of the cost!

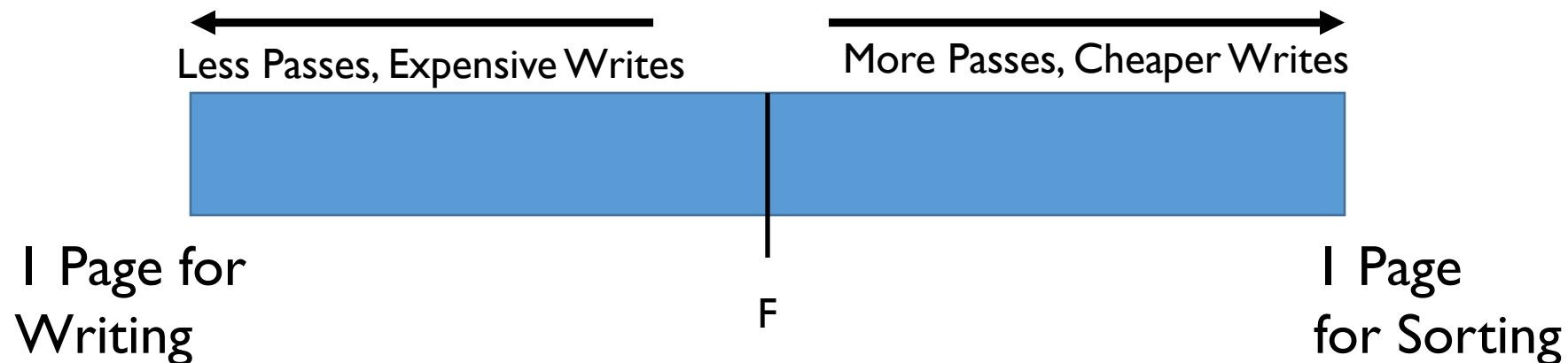
- ▶ However, when the base is (B-1) the number of passes will drastically decrease

Number of Passes of External Sort

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

Cost of Disk Operations

- ▶ We have seen that performing a disk write or read for a continuous sequence of files is more efficient.
- ▶ So, for example writing the whole file a page at a time will take more time than writing F pages at a time.
- ▶ Use $B-F$ pages for sorting, reserve F blocks for writing.



Sorting Records!

- ▶ Sorting has become a blood sport!
 - ▶ Parallel sorting is the name of the game ...
- ▶ Datamation: Sort 1M records of size 100 bytes
 - ▶ Typical DBMS: 5 minutes
 - ▶ 2001: .48sec at [UW](#)
- ▶ New benchmarks proposed:
 - ▶ Minute Sort: How many TB can your sort in 1 minute?
 - ▶ 2016: 37TB/55TB Tencent Sort at Tencent Corp., China
 - ▶ Cloud Sort: How much in \$ to sort 100 TB using a public cloud?
 - ▶ 2015: \$451 on 330 Amazon EC2 r3.4xlarge nodes, by profs at UCSD.
 - ▶ 2016: \$144 using Alibaba Cloud, by profs at Nanjing U, others

Example

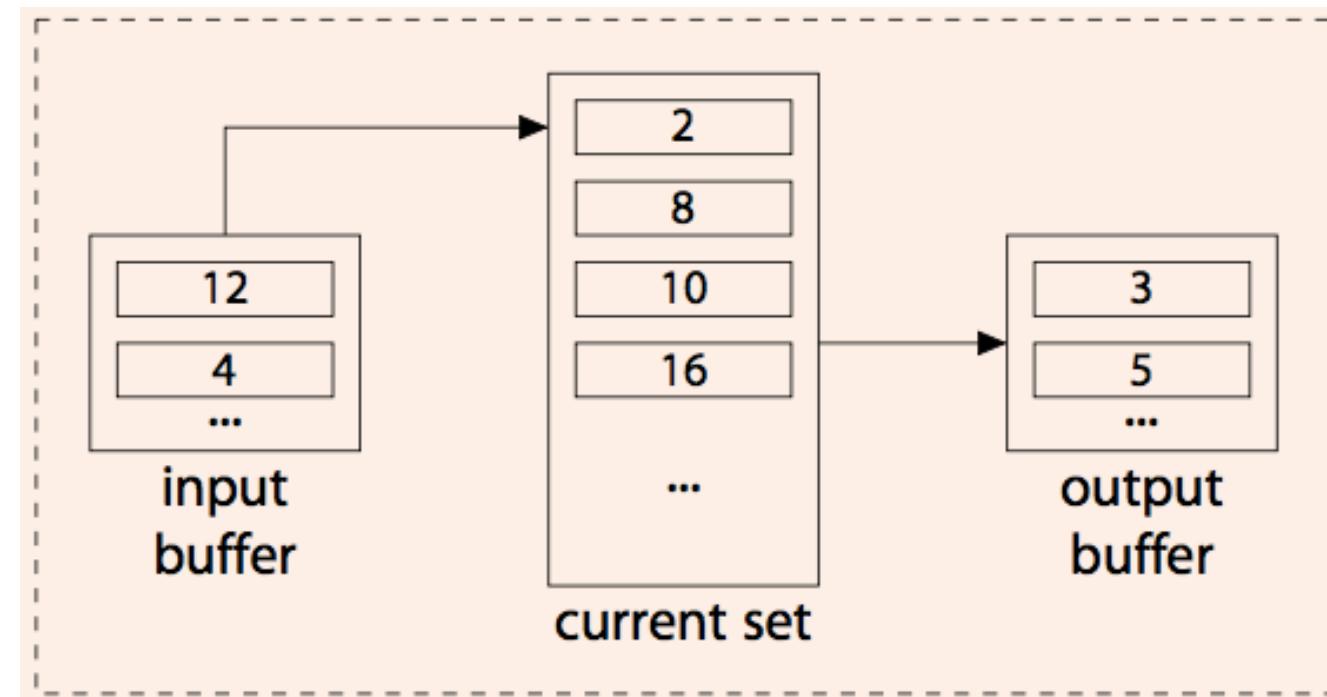
- ▶ Assuming that our most general external sorting algorithm is used. For a file with 2,000,000 pages and 17 available buffer pages, answer the following
 - 1. How many runs will you produce in the first pass?
 - 2. How many passes will it take to sort the file completely?
 - 3. What is the total I/O cost of sorting the file?
 - 4. How many buffer pages do you need to sort the file completely in just two passes?

Answer

- ▶ How many runs are produced in Pass 0
 - ▶ $\text{Ceil}(2000000/17) = 117648$ sorted runs.
- ▶ Number of passes required
 - ▶ $\text{Ceil}(\log_{16} 117648) + 1 = 6$ passes.
- ▶ Total number of disk accesses
 - ▶ $2*2000000*6 = 24000000$.
- ▶ How many Buffer pages do we need, to complete sort in two passes
 - ▶ We have to produce less than equal to $B-1$ runs after first pass. So $\text{ceil}(N/B)$ must be less than equal to $B-1$. For $2*10^6$ pages, if $B=10^3$ we have 2000 runs, $B=1415$ produces 1414 runs which can be merged in single pass.

Replacement Sort

- ▶ **Replacement sort** can help to further cut down the number of initial runs [N/B]: try to **produce initial runs with more than B pages**.
- ▶ Assume a buffer of B pages. Two pages are dedicated **input** and **output buffers**. The remaining $B - 2$ pages are called the **current set**:



Replacement Sort Algorithm

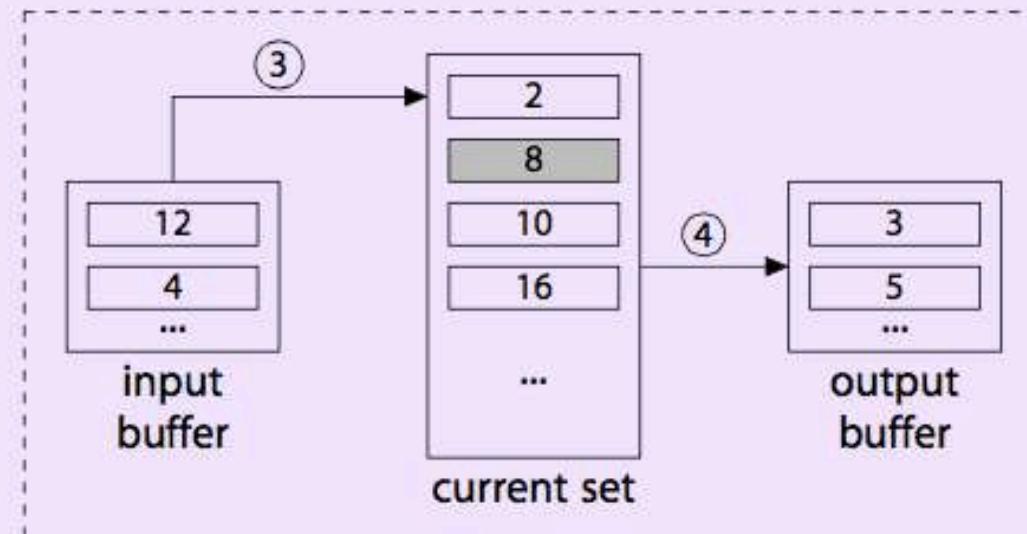
Replacement sort

- ① Open an empty run file for writing.
- ② Load next page of file to be sorted into input buffer.
If input file is exhausted, go to ④.
- ③ While there is space in the current set, move a record from input buffer to current set (if the input buffer is empty, reload it at ②).
- ④ In current set, pick record r with smallest key value k such that $k \geq k_{out}$ where k_{out} is the maximum key value in output buffer.¹ Move r to output buffer. If output buffer is full, append it to current run.
- ⑤ If all k in current set are $< k_{out}$, append output buffer to current run, close current run. Open new empty run file for writing.
- ⑥ If input file is exhausted, stop. Otherwise go to ③.

Replacement Sort

Example

Example (Record with key $k = 8$ will be the next to be moved into the output buffer; current $k_{out} = 5$)



- The record with key $k = 2$ remains in the current set and will be written to the subsequent run.

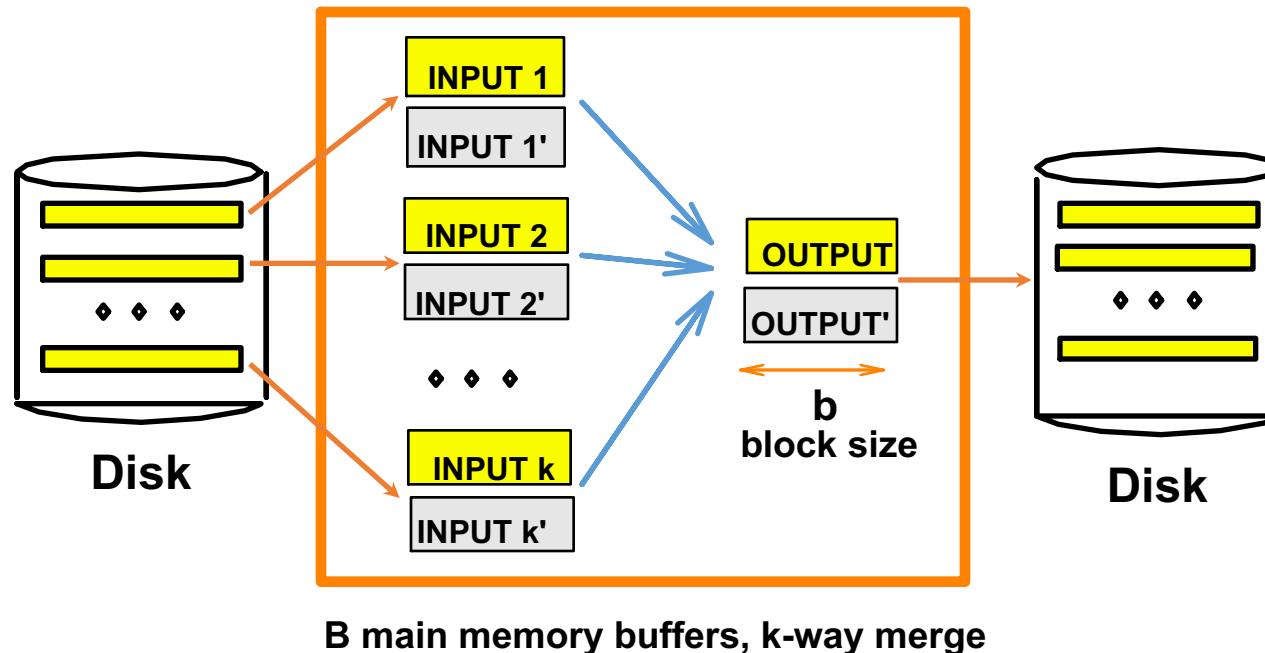
External Sort

Remarks

- ▶ External sorting follows a **divide and conquer** principle.
 - This results in a number of **independent (sub-)tasks**.
 - **Execute tasks in parallel** in a distributed DBMS or exploit multi-core parallelism on modern CPUs.
- ▶ To keep the CPU busy while the input buffer is reloaded (or the output buffer appended to the current run), use **double buffering**.

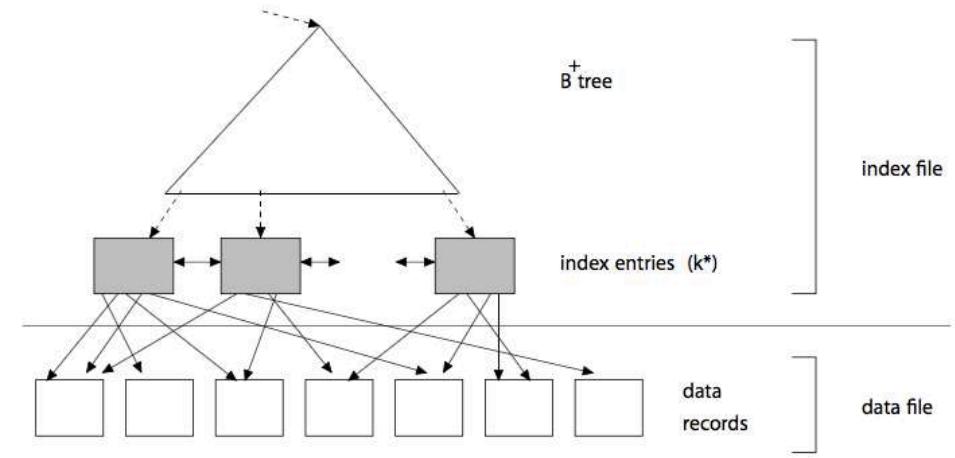
Double Buffering

- ▶ While waiting to read or write a block, the CPU will be idle as we have no data to process.
- ▶ To reduce wait time for I/O request to complete, can *prefetch* into 'shadow block'.
 - ▶ Potentially, more passes; in practice, most files still sorted in 2-3 passes.



B+ Trees for Sorting

- If a B+-tree matches a sorting task (i.e., B+-tree organized over key k with ordering θ), we *may* be better off to **access the index and abandon external sorting**.
- 1) If the B+-tree is **clustered**, then
 - the data file itself is already θ -sorted,
 - simply sequentially read the sequence set (or the pages of the data file).
- 2) If the B+-tree is **unclustered**, then
 - in the worst case, we have to initiate one I/O operation per record (not per page)! \Rightarrow do not consider the index.



Summary

- ▶ External sorting is important; DBMS may dedicate part of buffer pool just for sorting!
- ▶ External merge sort minimizes disk I/O cost.
- ▶ Using clustered B+ tree for sorting is always better than external sorting. Root to the leftmost leaf, then retrieve all leaf pages.
- ▶ Using an unclustered B+ tree for sorting could be a very bad idea.
- ▶ The best sorts are already fast but still we're still improving!



BBM371- Data Management

Lecture 12 Spatial Data Management

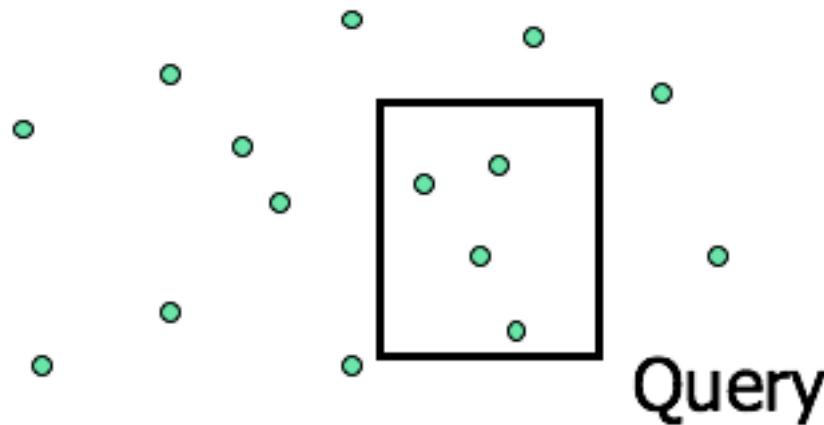
02.01.2020

Spatial and Geographic Databases

- Spatial databases store information related to spatial locations, and support efficient storage, indexing and querying of spatial data.
- Special purpose index structures are important for accessing spatial data, and for processing spatial queries.
- Computer Aided Design (CAD) databases store design information about how objects are constructed E.g.: designs of buildings, aircraft, layouts of integrated-circuits
- Geographic databases store geographic information (e.g., maps): often called **geographic information systems or GIS**.

The Problem

- ▶ Given a point set and a rectangular query, find the points enclosed in the query
- ▶ We allow insertions/deletions on line



Types of Spatial Data

- ▶ **Point Data**

- ▶ Points in a multidimensional space

- ▶ **Region Data**

- ▶ Objects have spatial extent with location and boundary
 - ▶ DB typically uses geometric approximations constructed using line segments, polygons

Types of Spatial Data

■ **Raster data** consist of bit maps or pixel maps, in two or more dimensions.

- Example 2-D raster image: satellite image of cloud cover, where each pixel stores the cloud visibility in a particular area.
- Additional dimensions might include the temperature at different altitudes at different regions, or measurements taken at different points in time.

■ Design databases generally do not store raster data.

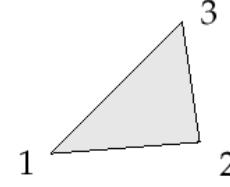
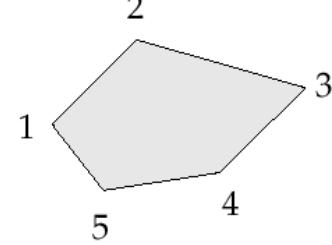
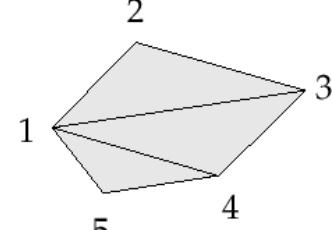
Types of Spatial Data

- **Vector data** are constructed from basic geometric objects: points, line segments, triangles, and other polygons in two dimensions, and cylinders, spheres, cuboids, and other polyhedrons in three dimensions.
- Vector format often used to represent map data.
 - Roads can be considered as two-dimensional and represented by lines and curves.
 - Some features, such as rivers, may be represented either as complex curves or as complex polygons, depending on whether their width is relevant.
 - Features such as regions and lakes can be depicted as polygons.

Representation of Vector Information

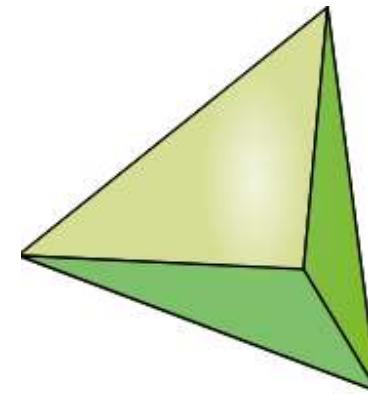
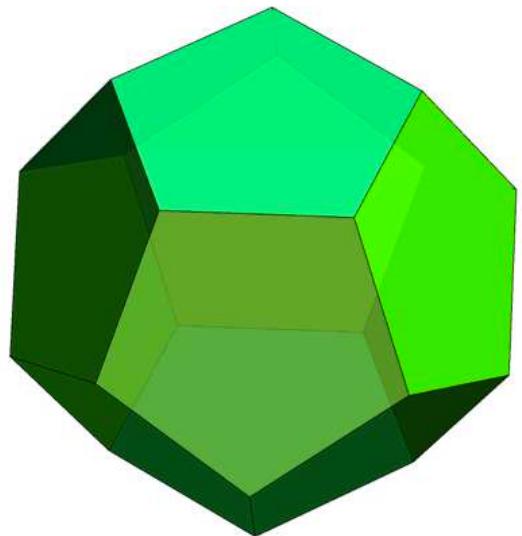
- Various geometric constructs can be represented in a database in a normalized fashion.
- Represent a line segment by the coordinates of its endpoints.
- Approximate a curve by partitioning it into a sequence of segments
 - Create a list of vertices in order, or
 - Represent each segment as a separate tuple that also carries with it the identifier of the curve (2D features such as roads).
- Closed polygons
 - List of vertices in order, starting vertex is the same as the ending vertex, or
 - Represent boundary edges as separate tuples, with each containing identifier of the polygon, or
 - Use **triangulation** – divide polygon into triangles
 - ▶ Note the polygon identifier with each of its triangles.

Representation of Vector Data

object	representation
line segment	 $\{(x_1, y_1), (x_2, y_2)\}$
triangle	 $\{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$
polygon	 $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4), (x_5, y_5)\}$
polygon	 $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), \text{ID1}\}$ $\{(x_1, y_1), (x_3, y_3), (x_4, y_4), \text{ID1}\}$ $\{(x_1, y_1), (x_4, y_4), (x_5, y_5), \text{ID1}\}$

Representation of Vector Data (Cont.)

- Representation of points and line segments in 3-D similar to 2-D, except that points have an extra z component
- Represent arbitrary polyhedra by dividing them into tetrahedrons, like triangulating polygons.

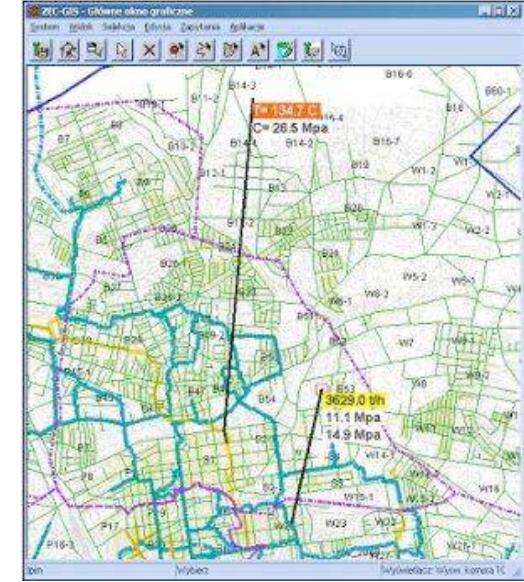


Types of Spatial Queries

- ▶ **Spatial Range Queries**
 - ▶ *“Find all cities within 50 miles of Madison”*
 - ▶ Query has associated region (location, boundary)
 - ▶ Answer includes overlapping or contained data regions
- ▶ **Nearest-Neighbour Queries**
 - ▶ *“Find the 10 cities nearest to Madison”*
 - ▶ Results must be ordered by proximity
- ▶ **Spatial Join Queries**
 - ▶ *“Find all cities near a lake”*
 - ▶ Expensive, join condition involves regions and proximity
- ▶ **Region queries** deal with spatial regions. e.g., ask for objects that lie partially or fully inside a specified region.

Applications of Spatial Data

- ▶ **Geographic Information Systems (GIS)**
 - ▶ E.g., ESRI's ArcInfo; OpenGIS Consortium
 - ▶ Geospatial information
 - ▶ All classes of spatial queries and data are common
 - ▶ **Computer-Aided Design/Manufacturing**
 - ▶ Store spatial objects such as surface of airplane fuselage
 - ▶ Range queries and spatial join queries are common
 - ▶ **Multimedia Databases**
 - ▶ Images, video, text, etc. stored and retrieved by content
 - ▶ First converted to *feature vector* form; high dimensionality
 - ▶ Nearest-neighbor queries are the most common



Applications of Spatial Data

Pizza loc: 555 E.El. Camino Real, Sunnyvale, CA Google Maps Microsoft Internet Explorer
File Edit View Favorites Tools Help

Saved Locations | Help

Google Maps Web Images Video News! News Maps more »
Pizza 555 E El. Camino Real, Sunnyvale, CA Search Businesses
Search the map Find businesses Get directions

Print Email Link to this page

Maps

Results 1-10 of about 20,205 for Pizza near 555 E El Camino Real, Sunnyvale, CA 94087 - [Modify search](#)

Categories: Restaurant Pizza, Restaurants

A [Sierra Sam's Pizza](#) - [more info »](#)
568B E El Camino Real, Sunnyvale, CA (408) 738-4996 - [4 reviews](#) - 0.1 mi S

B [Hbutlers BBQ](#) - [more info »](#)
568 E. El Camino Real, Sunnyvale, CA (408) 738-4996 - 0.1 mi S

C [Little Caesars Pizza](#) - [more info »](#)
1039 Sunnyvale Saratoga Rd, Sunnyvale, CA (408) 245-0607 - [2 reviews](#) - 0.5 mi SW

D [Round Table Pizza: Sunnyvale](#) - [more info »](#)
860 Old San Francisco Rd, Sunnyvale, CA (408) 245-9000 - [1 review](#) - 0.6 mi NE

E [Jakes of Sunnyvale](#) - [more info »](#)
174 E Fremont Ave, Sunnyvale, CA (408) 720-8884 - [4 reviews](#) - 0.8 mi S

F [Big Bite Pizza](#) - [more info »](#)
109 E Fremont Ave, Sunnyvale, CA (408) 481-0666 - 0.8 mi SW

G [Stuff Pizza](#) - [more info »](#)
826 E Fremont Ave # A, Sunnyvale, CA (408) 720-0700 - 0.9 mi SE

Map Satellite Hybrid

(15 items remaining) Downloading picture http://www.google.com/intl/en_ALL/mapfiles/dithshadow.gif...
© 2006 Google Mapdata ©2006 NAVTEQ Inc. Terms of Use
Internet

Applications of Spatial Data

S http://maps.google.com - from: 555 E El Camino Real, Sunnyvale, CA to: 355 N Wolfe Rd, Sunnyvale - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Saved Locations | Help

Web Images Video New! News Maps more »

555 E El Camino Real, Sunnyvale, CA to: 355 N Wolfe Rd, Sunnyvale, CA Get Directions

Search the map Find businesses Get directions

Print Email Link to this page

Maps

Start address: 555 E El Camino Real
Sunnyvale, CA 94087

End address: 355 N Wolfe Rd
Sunnyvale, CA 94085

Distance: 2.7 mi (about 6 mins)

Get reverse directions

1. Head northwest from E El Camino Real - go 0.5 mi
2. Turn right at S Sunnyvale Ave - go 0.7 mi
3. Continue on N Sunnyvale Ave - go 0.7 mi
4. Bear right and head toward E Maude Ave - go 322 ft
5. Bear right at E Maude Ave - go 0.5 mi
6. Turn right at N Wolfe Rd - go 0.3 mi
7. Arrive at 355 N Wolfe Rd
Sunnyvale, CA 94085

These directions are for planning purposes only. You may find that construction projects, traffic, or other events may cause road conditions to differ from the map results.

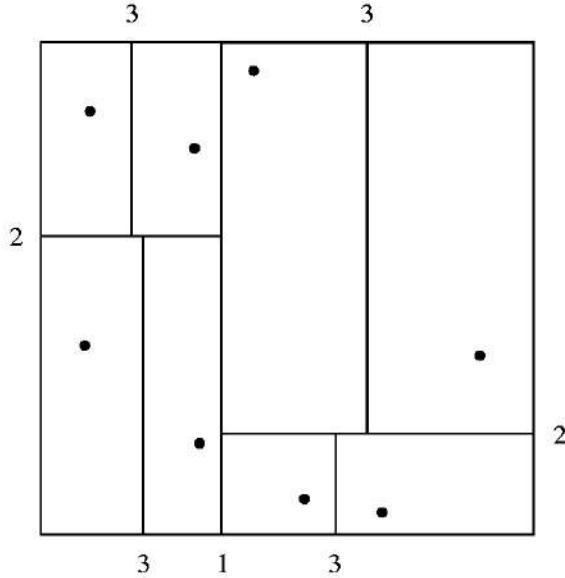
Map data ©2006 NAVTEQ™

Done Internet

Indexing of Spatial Data

- **k-d tree** - early structure used for indexing in multiple dimensions.
- Each level of a *k-d* tree partitions the space into two.
 - choose one dimension for partitioning at the root level of the tree.
 - choose another dimension for partitioning in nodes at the next level and so on, cycling through the dimensions.
- In each node, approximately half of the points stored in the sub-tree fall on one side and half on the other.
- Partitioning stops when a node has less than a given maximum number of points.
- The **k-d-B tree** extends the *k-d* tree to allow multiple child nodes for each internal node; well-suited for secondary storage.

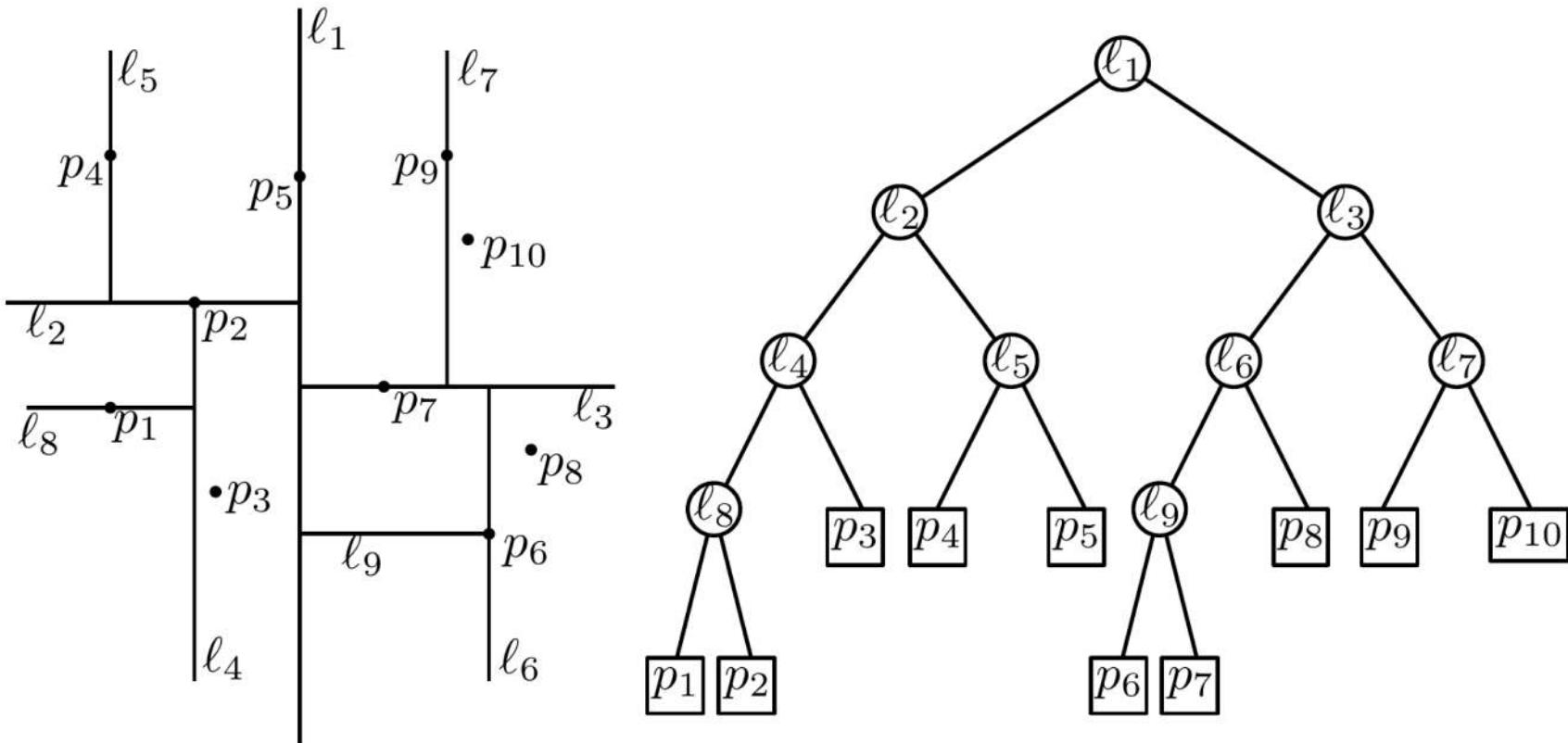
Division of Space by a k-d Tree



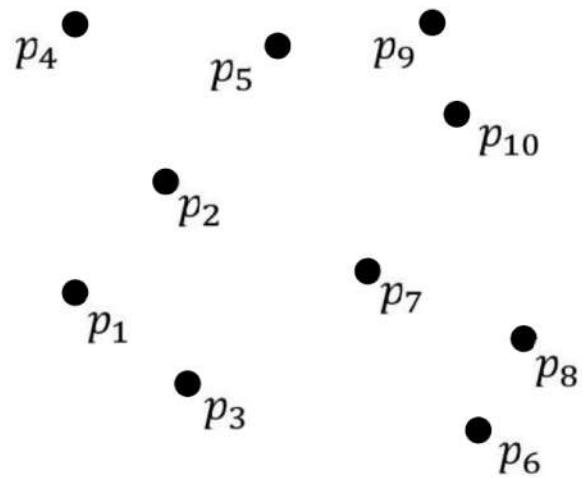
- Each line in the figure (other than the outside box) corresponds to a node in the *k-d* tree
 - the maximum number of points in a leaf node has been set to 1.
- The numbering of the lines in the figure indicates the level of the tree at which the corresponding node appears.

Kd-tree

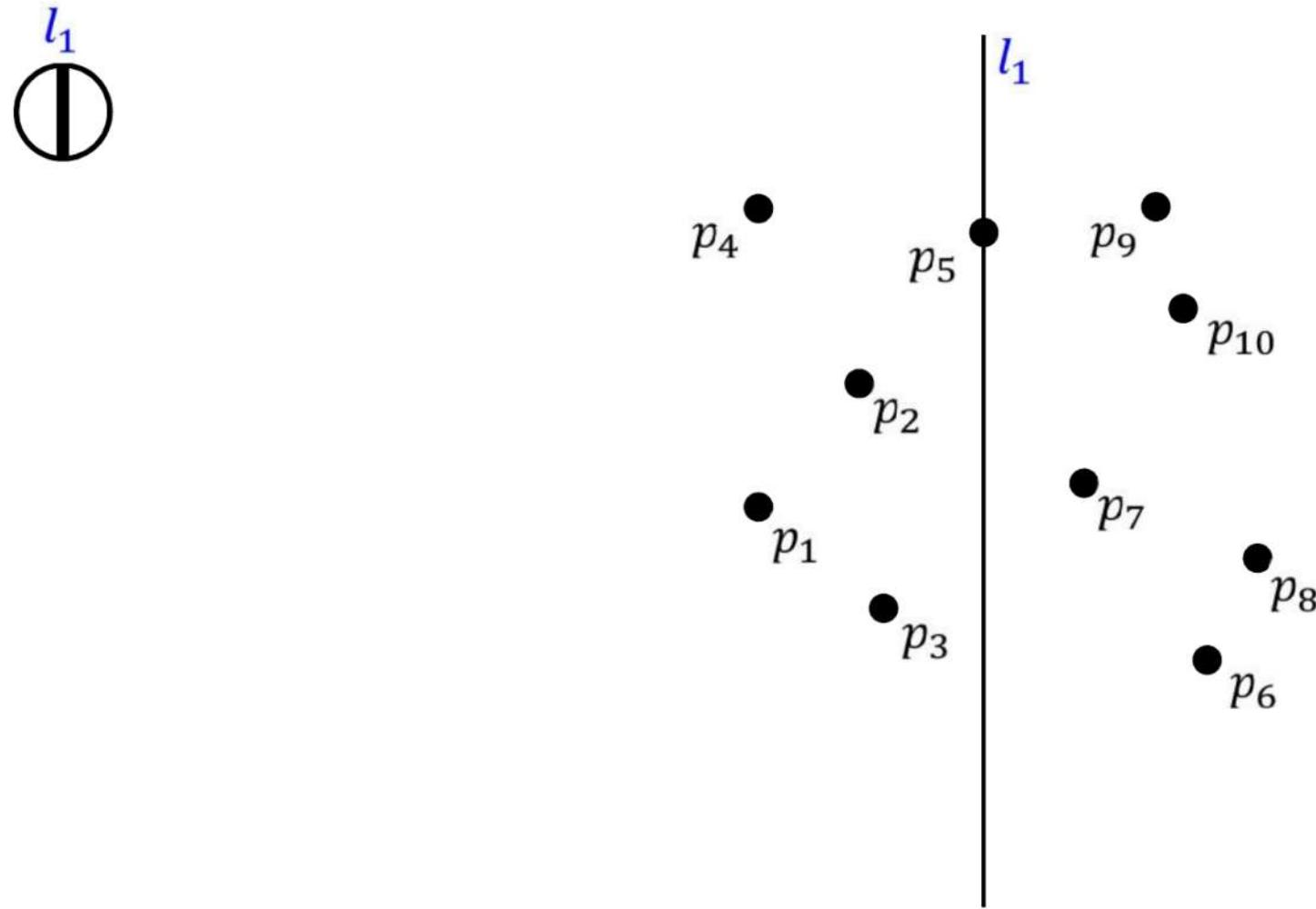
» Alternate between coordinates and split the point set

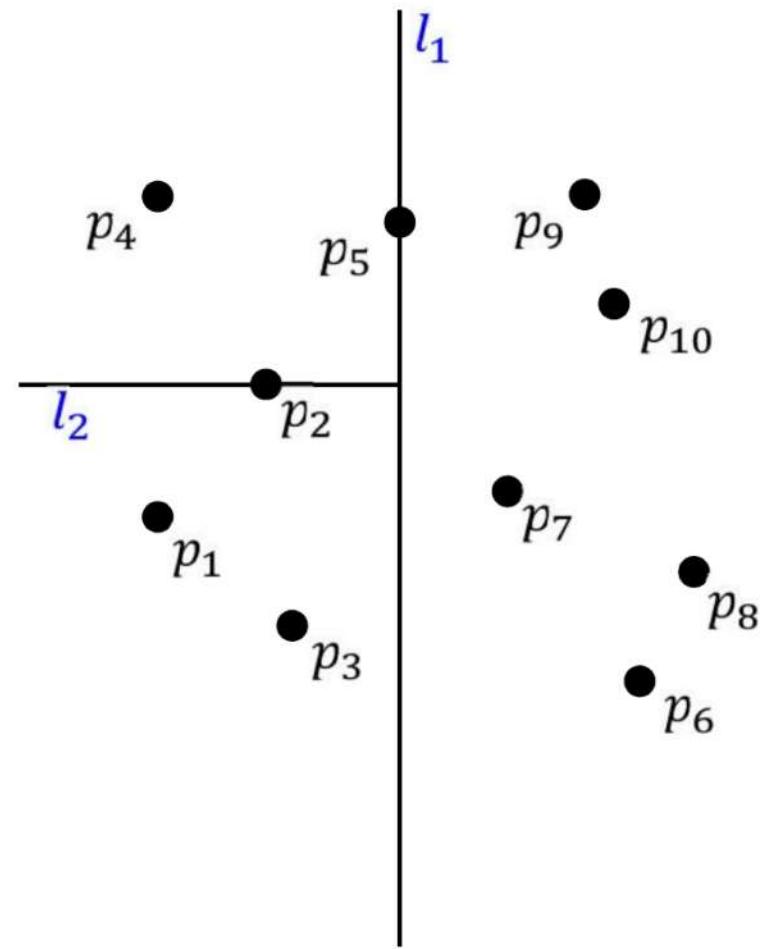
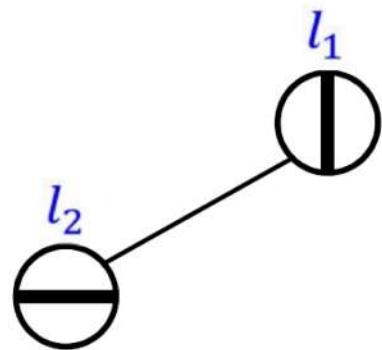


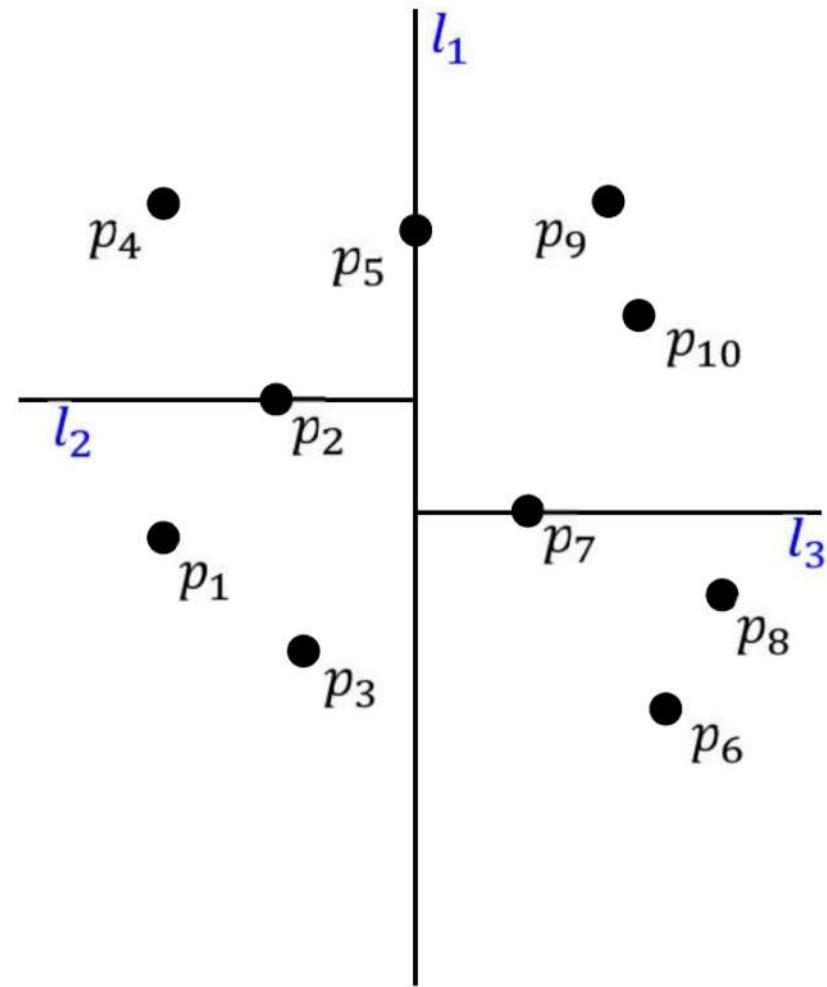
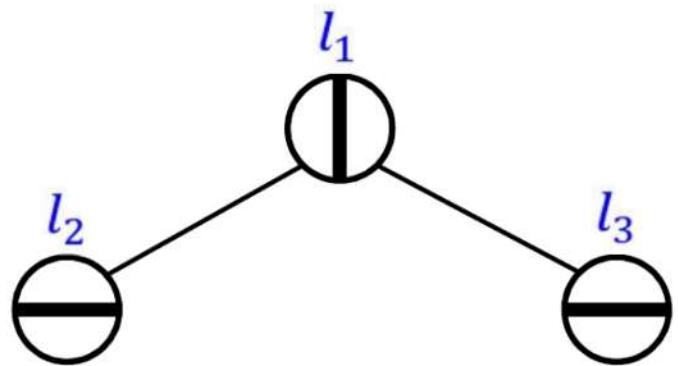
K-d Tree

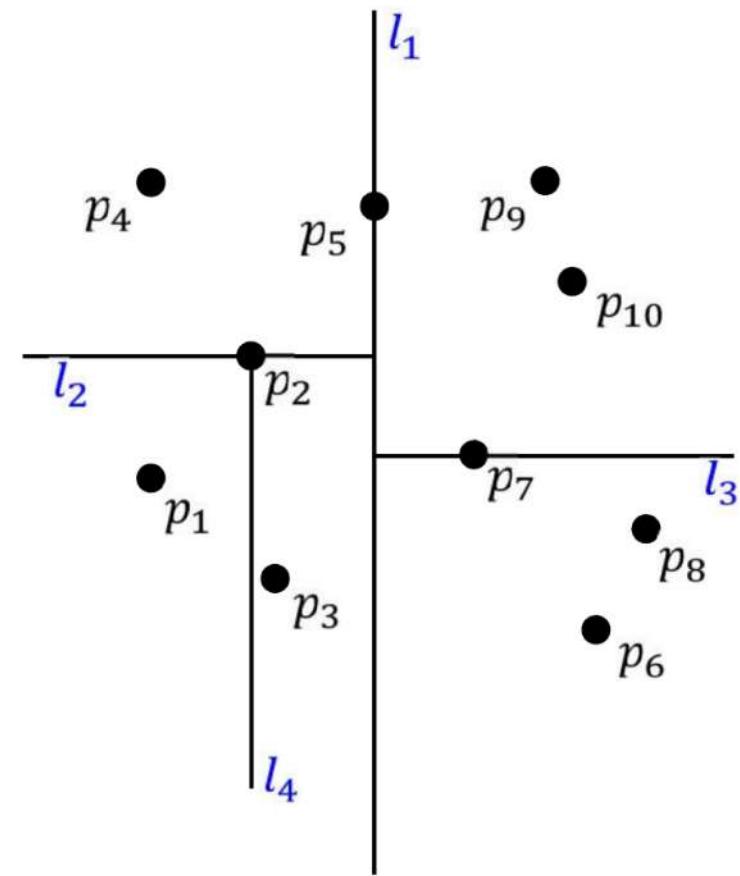
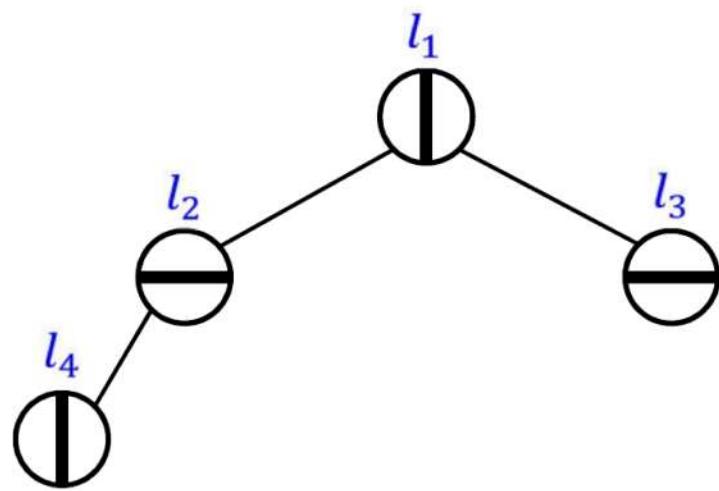


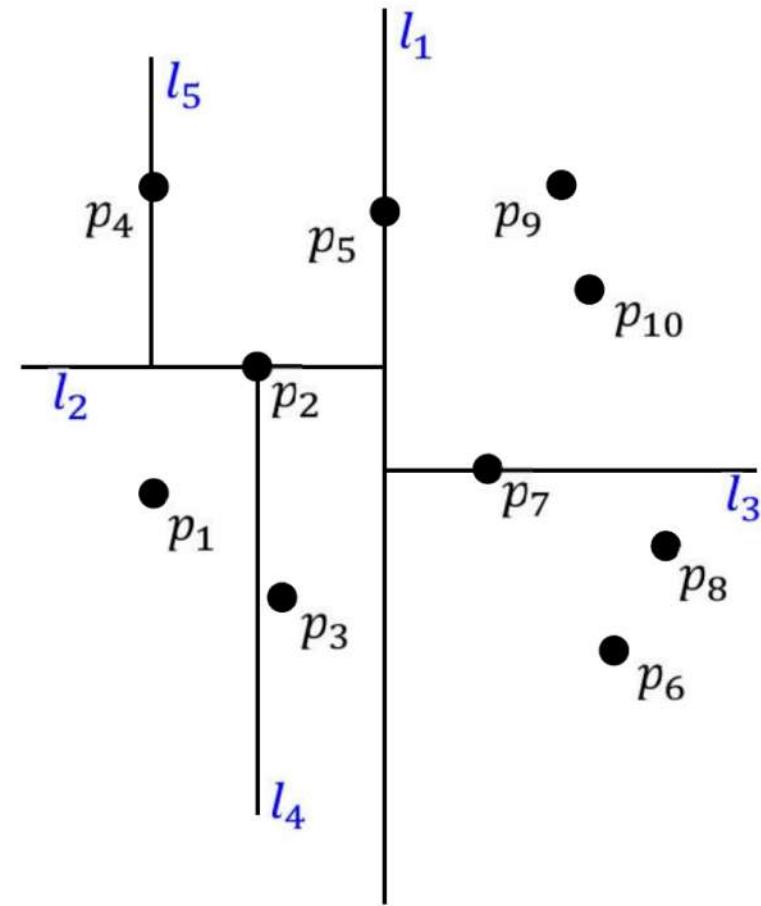
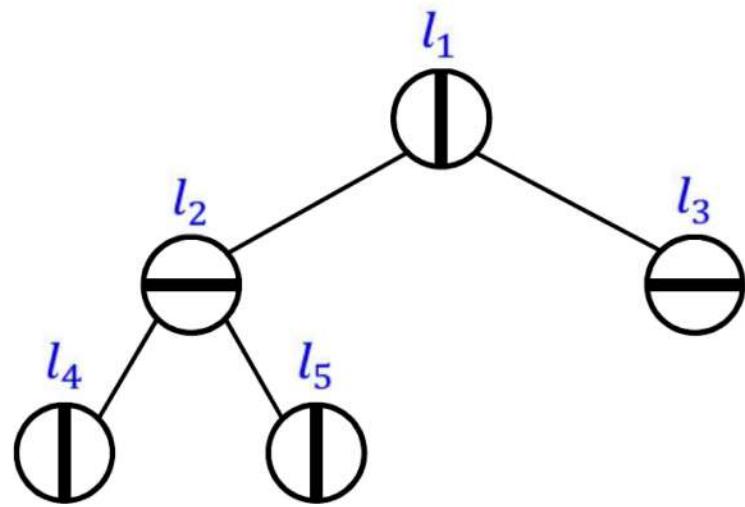
K-d tree

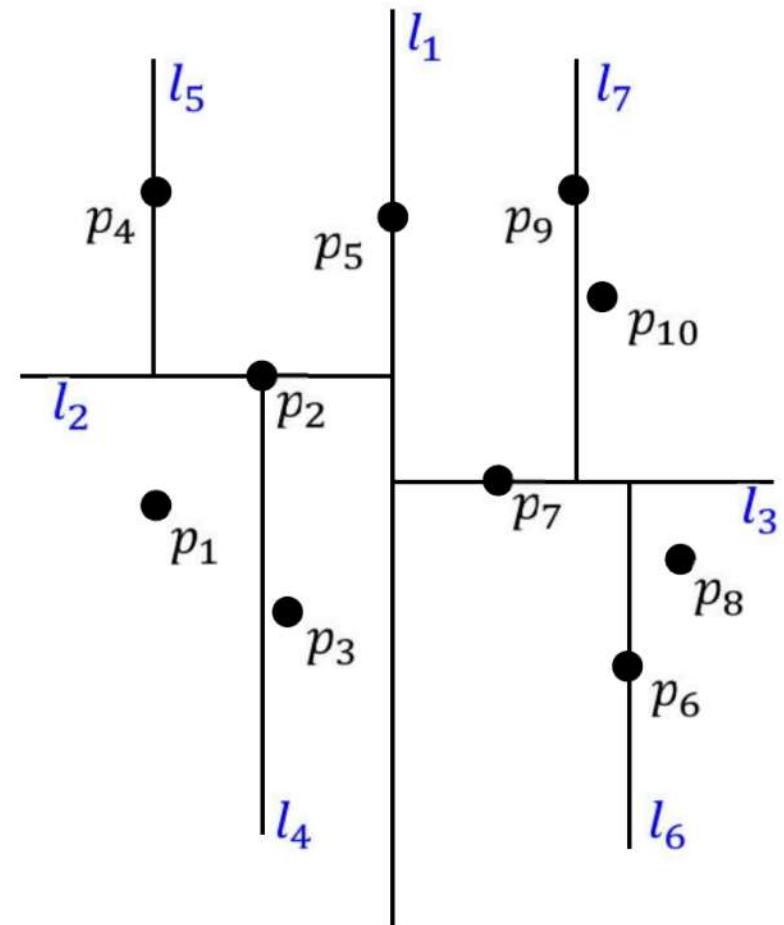
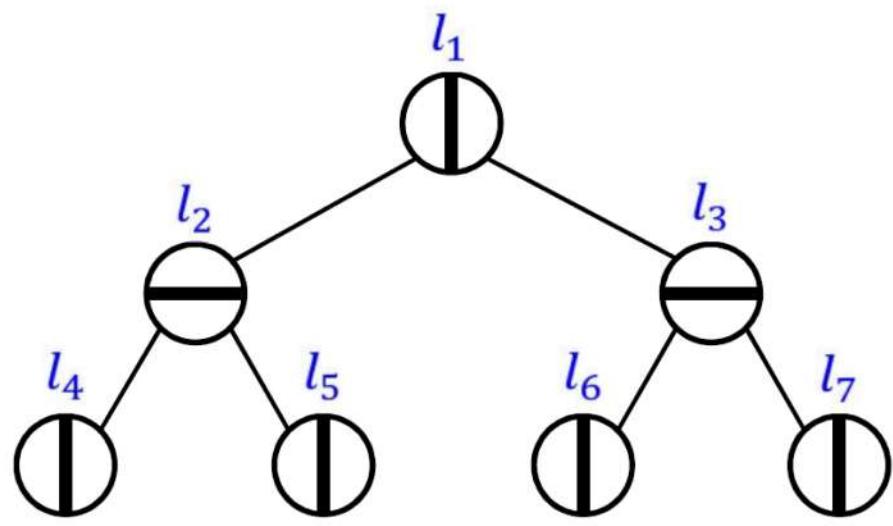


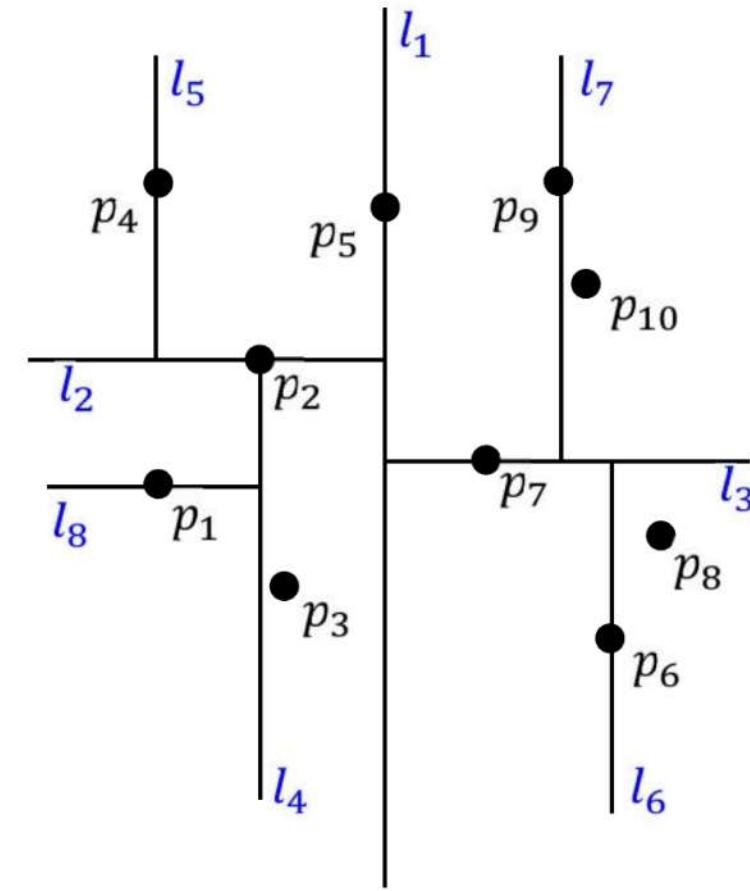
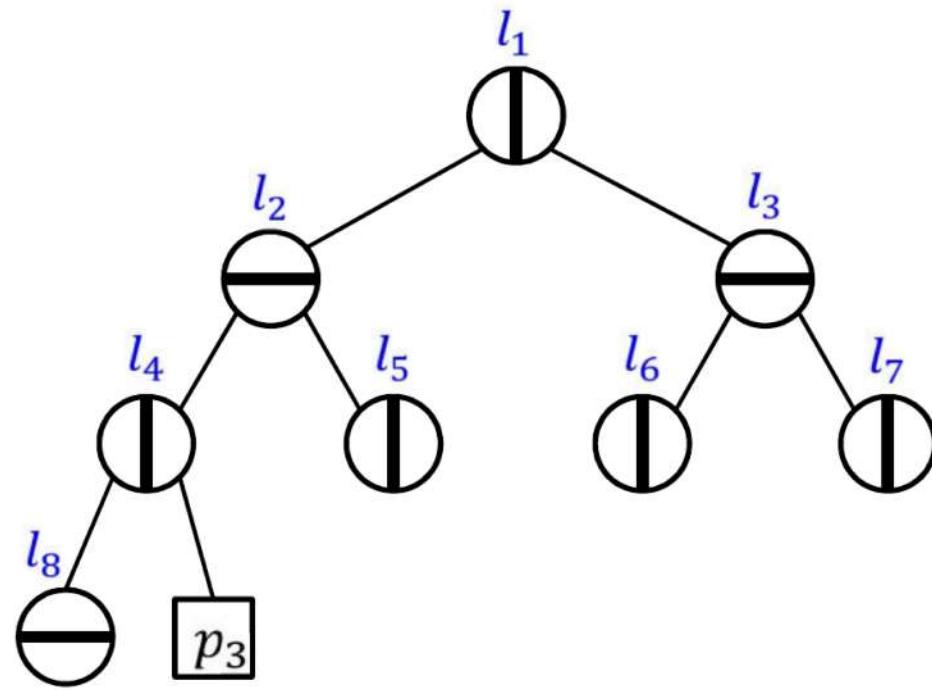


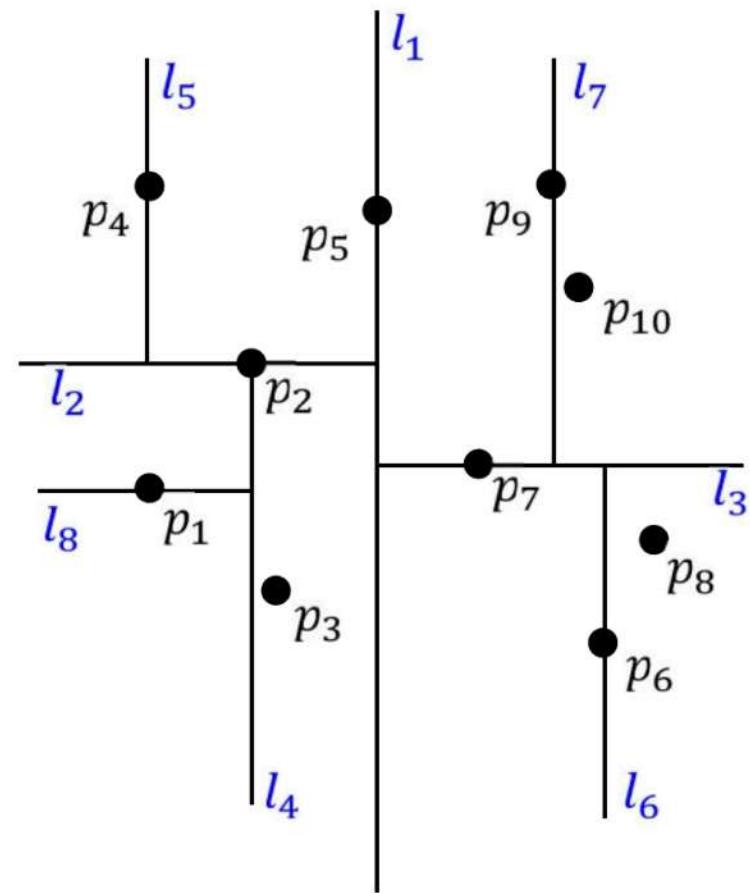
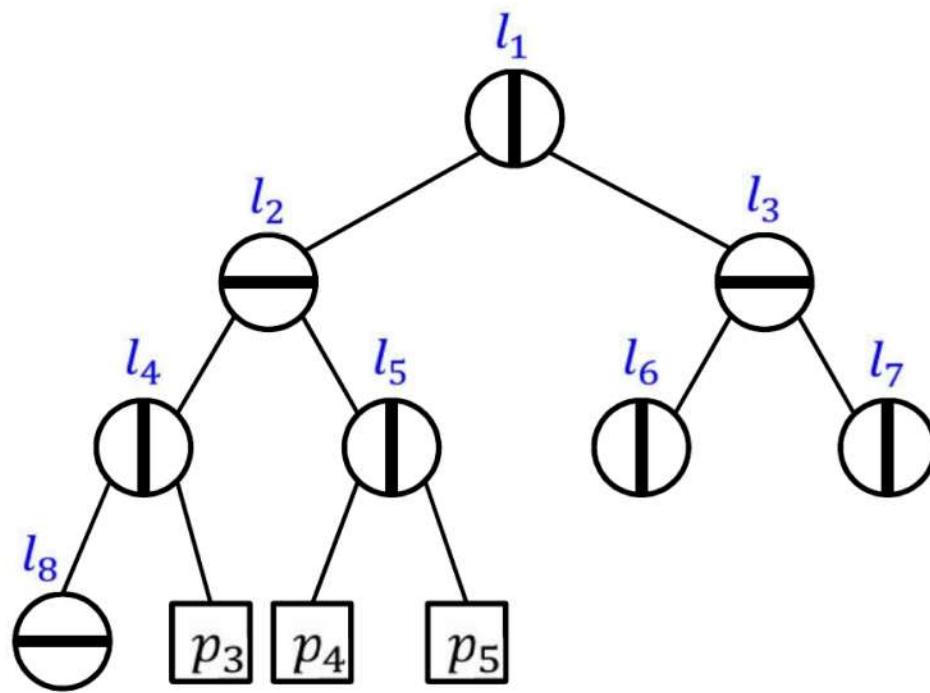


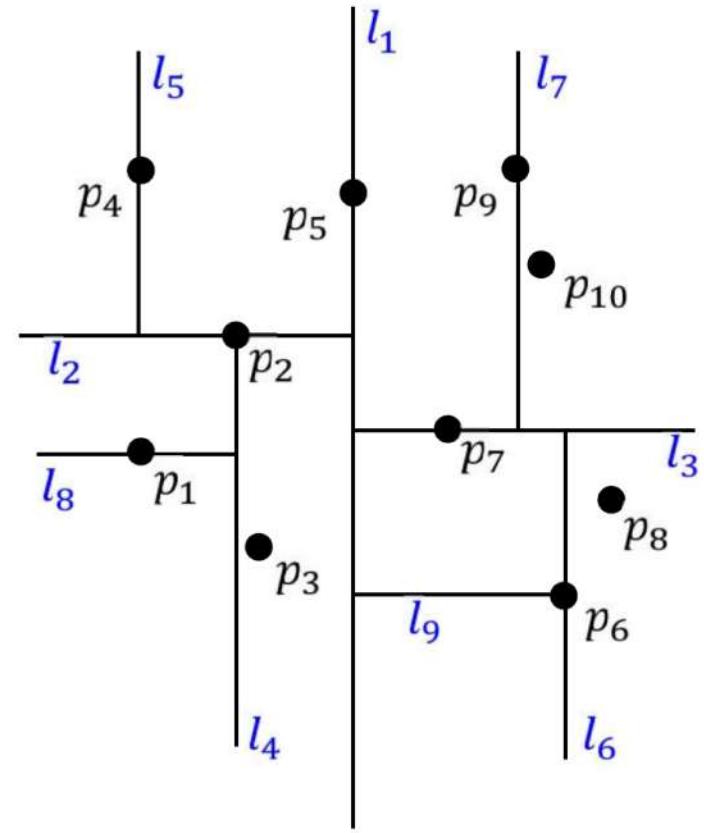
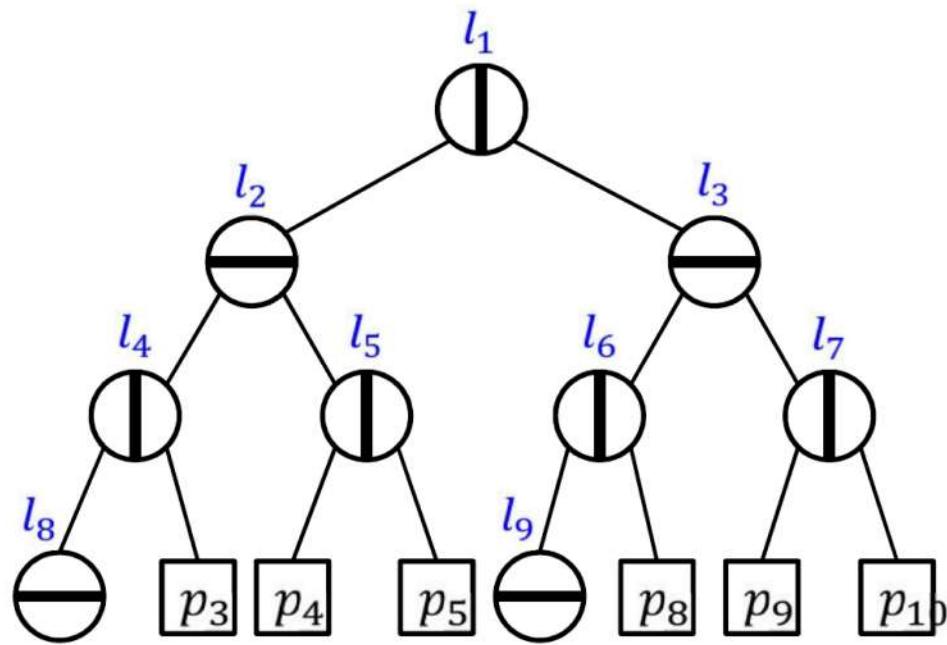


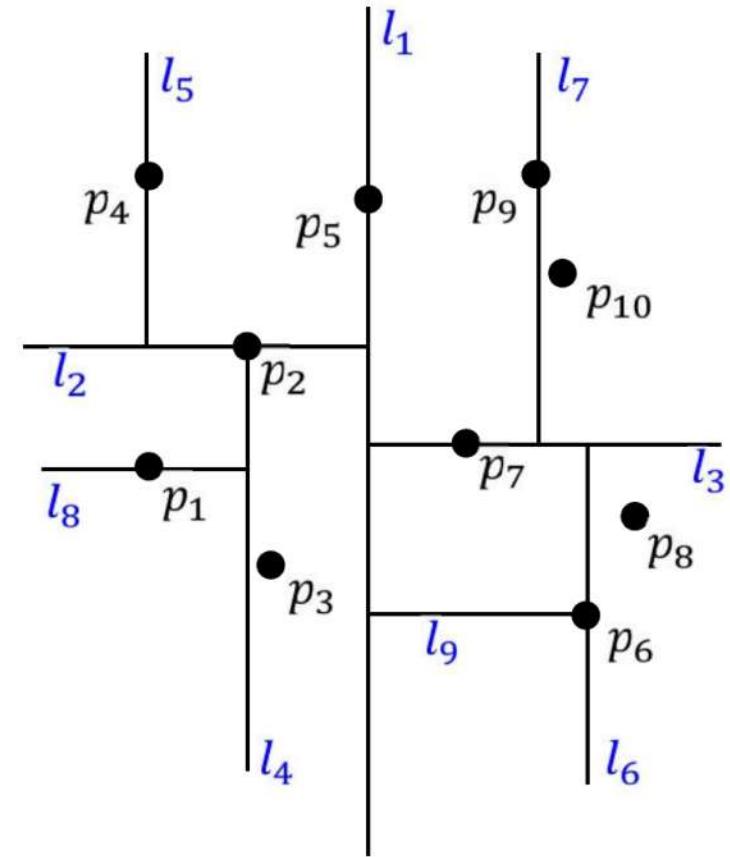
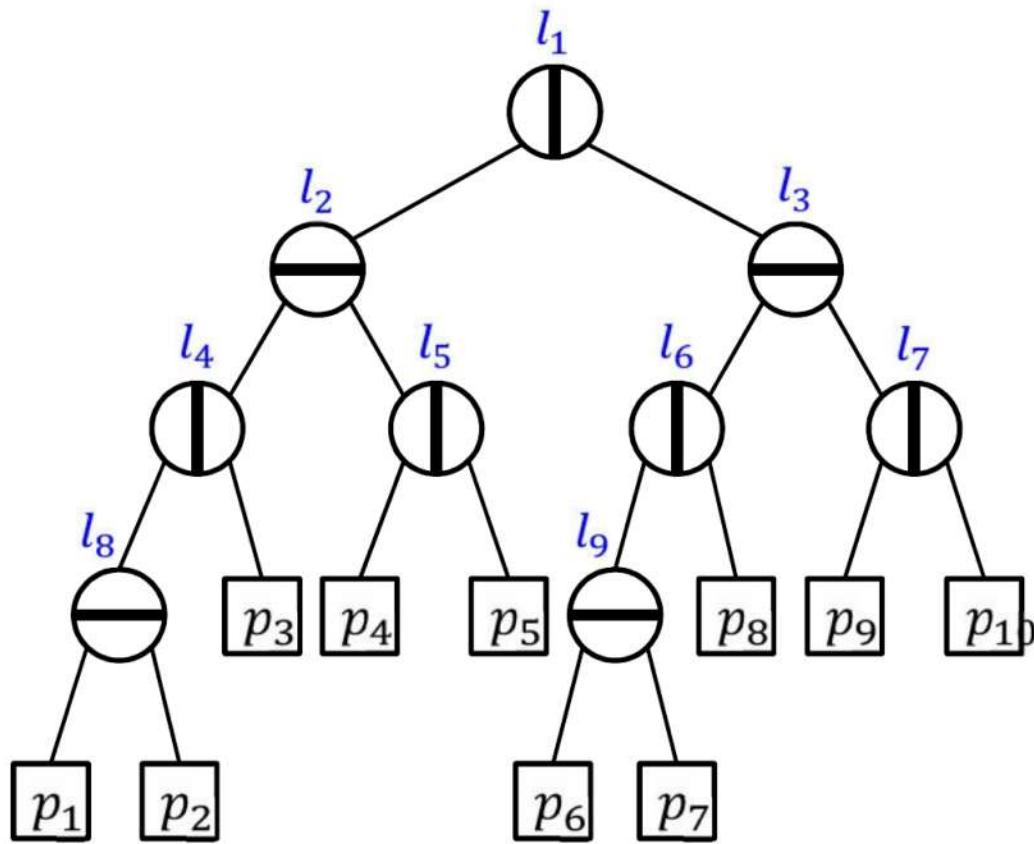








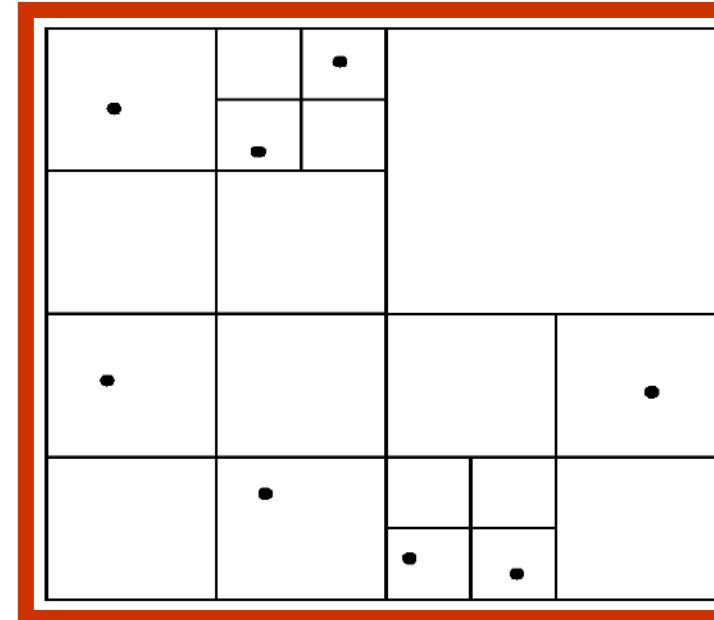




Division of Space by Quadtrees

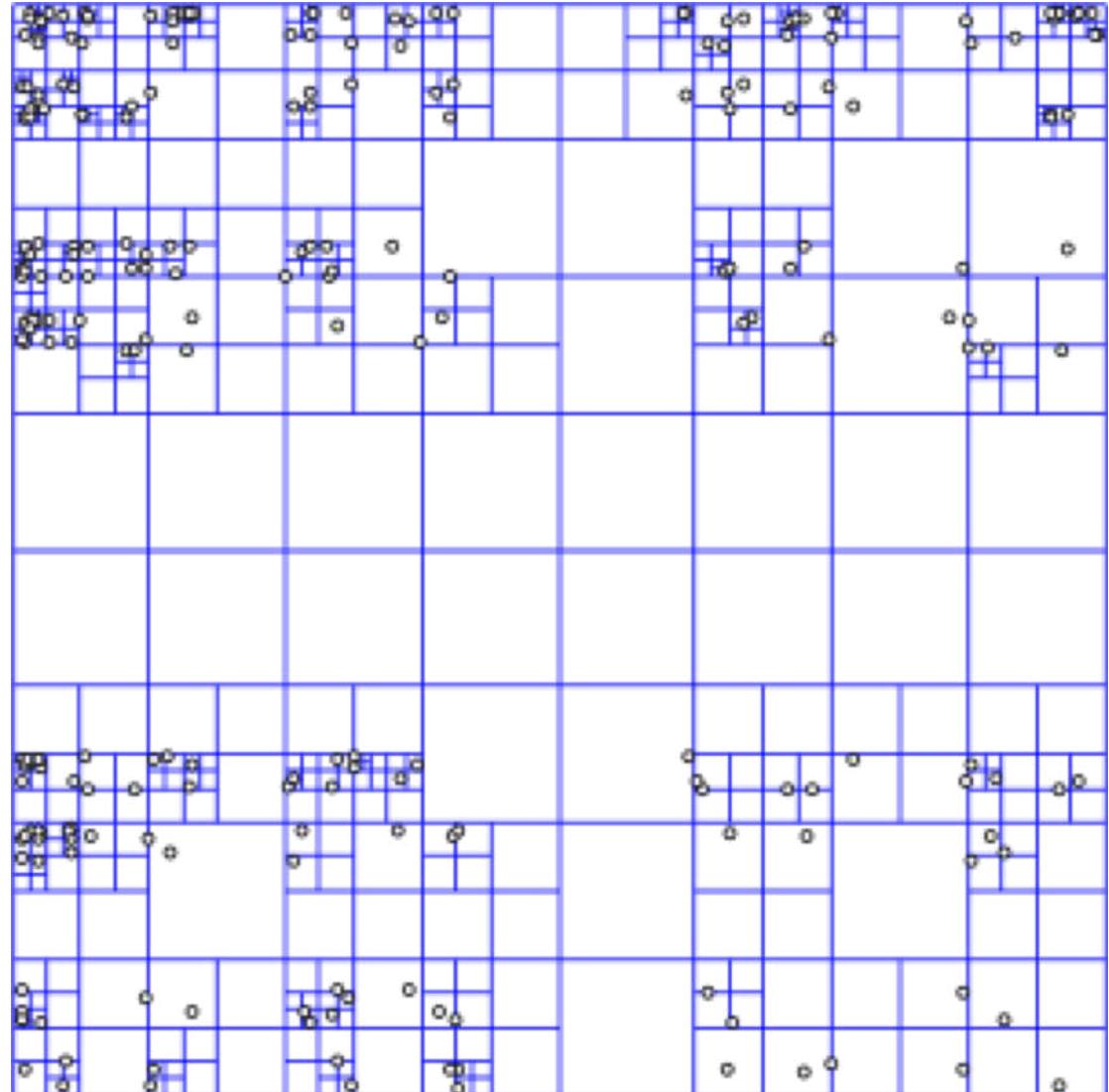
Quadtrees

- Each node of a quadtree is associated with a rectangular region of space; the top node is associated with the entire target space.
- Each non-leaf node divides its region into four equal sized quadrants
 - correspondingly each such node has four child nodes corresponding to the four quadrants and so on
- Leaf nodes have between zero and some fixed maximum number of points (set to 1 in example).



Quadtree

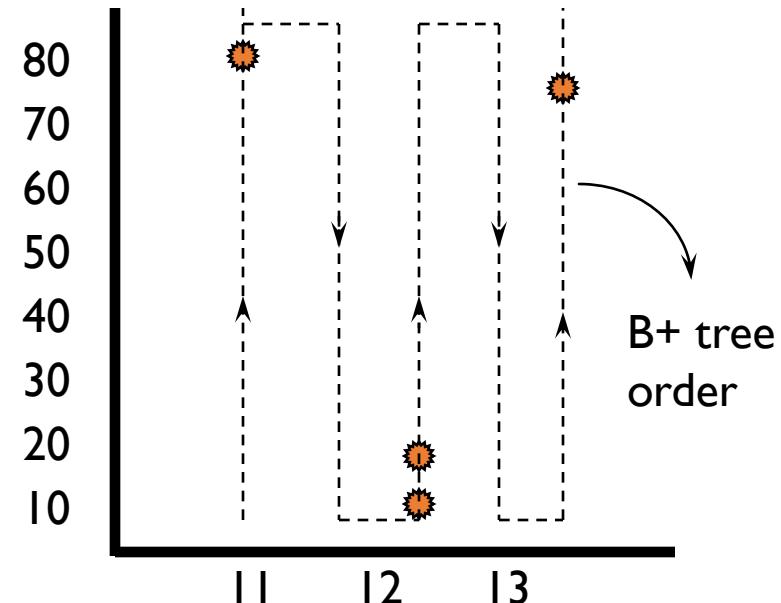
- ▶ Quadtree has more nodes for regions with more points
- ▶ The tree is not balanced
 - ▶ Depth of leaves is not the same
 - ▶ A region with more points will be in a larger depth



Single Dimensional Indexes

- ▶ B+ trees are fundamentally **single-dimensional** indexes.
- ▶ When we create a composite search key B+ tree, e.g., an index on **<age, sal>**, we effectively linearize the 2-dimensional space since we sort entries first by **age** and then by **sal**.

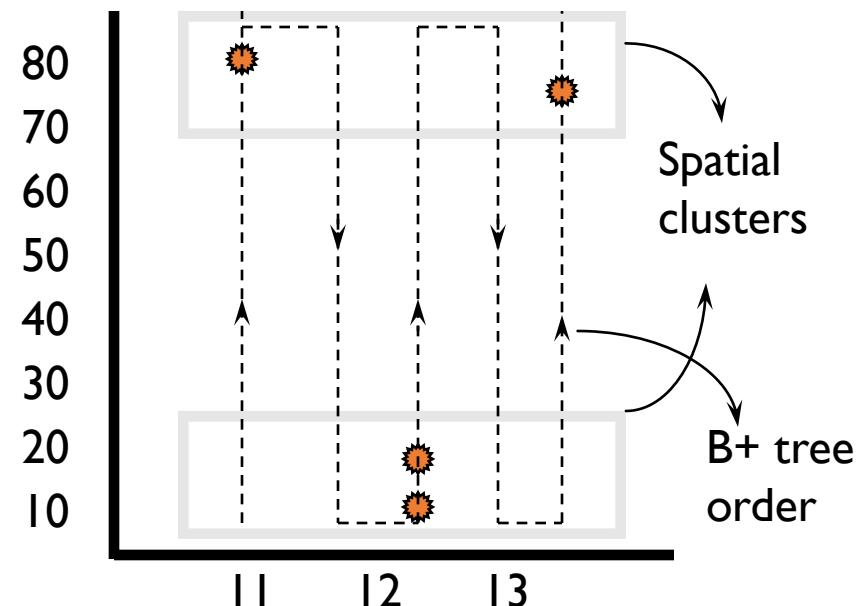
Consider entries:
 $\langle 11, 80 \rangle, \langle 12, 10 \rangle$
 $\langle 12, 20 \rangle, \langle 13, 75 \rangle$



Multidimensional Indexes

- ▶ A multidimensional index **clusters** entries so as to exploit “nearness” in multidimensional space.
- ▶ Keeping track of entries and maintaining a balanced index structure presents a challenge!

Consider entries:
 $\langle 11, 80 \rangle, \langle 12, 10 \rangle$
 $\langle 12, 20 \rangle, \langle 13, 75 \rangle$



Motivation for Multidimensional Indexes

- ▶ **Spatial queries (GIS, CAD).**
 - ▶ Find all hotels within a radius of 5 miles from the conference venue.
 - ▶ Find the city with population 500,000 or more that is nearest to Kalamazoo, MI.
 - ▶ Find all cities that lie on the Nile in Egypt.
 - ▶ Find all parts that touch the fuselage (in a plane design).
- ▶ **Similarity queries (content-based retrieval).**
 - ▶ Given a face, find the five most similar faces.
- ▶ **Multidimensional range queries.**
 - ▶ $50 < \text{age} < 55 \text{ AND } 80K < \text{sal} < 90K$

What is the difficulty?

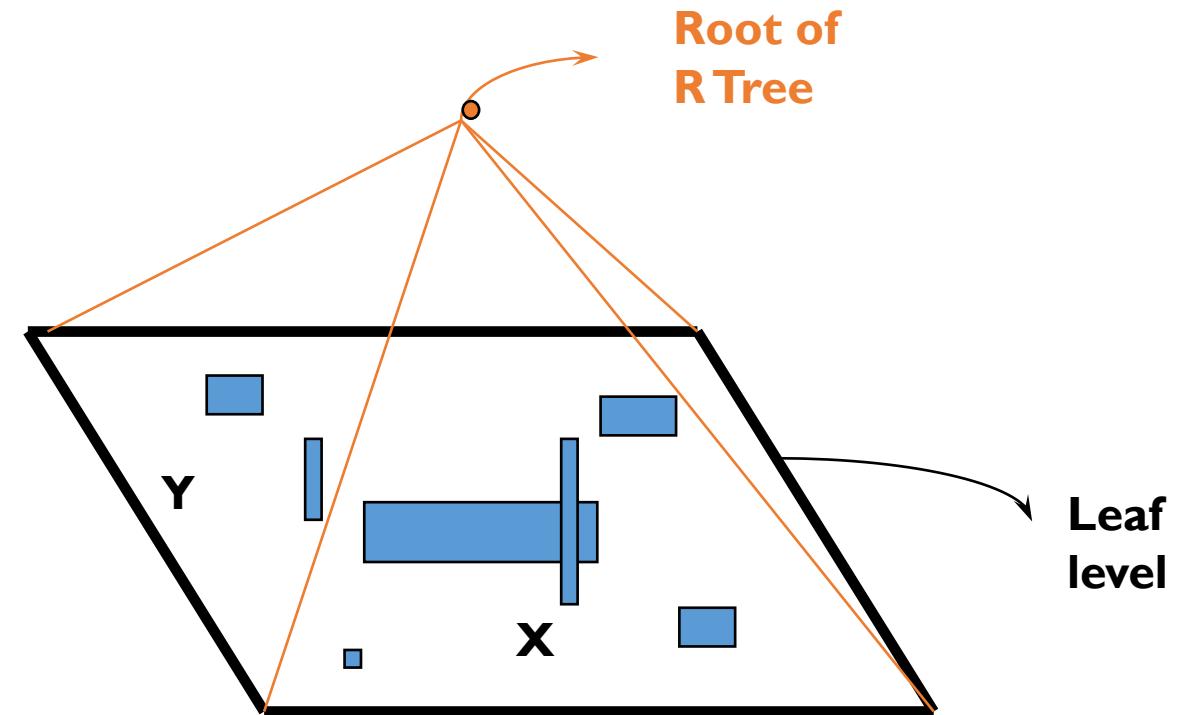
- ▶ An index based on spatial location needed.
 - ▶ One-dimensional indexes don't support multidimensional searching efficiently. (Why?)
 - ▶ Hash indexes only support point queries; want to support range queries as well.
 - ▶ Must support inserts and deletes gracefully.
- ▶ Ideally, want to support **non-point data** as well (e.g., lines, shapes).
- ▶ The R-tree meets these requirements, and variants are widely used today.

What's wrong with B-Trees?

- ▶ B-Trees cannot store new types of data
- ▶ Specifically people wanted to store geometrical data and multi-dimensional data
- ▶ The R-Tree provided a way to do that (thanx to Guttman '84)

R-Trees

- ▶ The R-tree is a tree-structured index that remains balanced on inserts and deletes.
- ▶ Each key stored in a leaf entry is intuitively a **box**, or collection of **intervals**, with one interval per dimension.
- ▶ Example in 2-D:



R-Trees

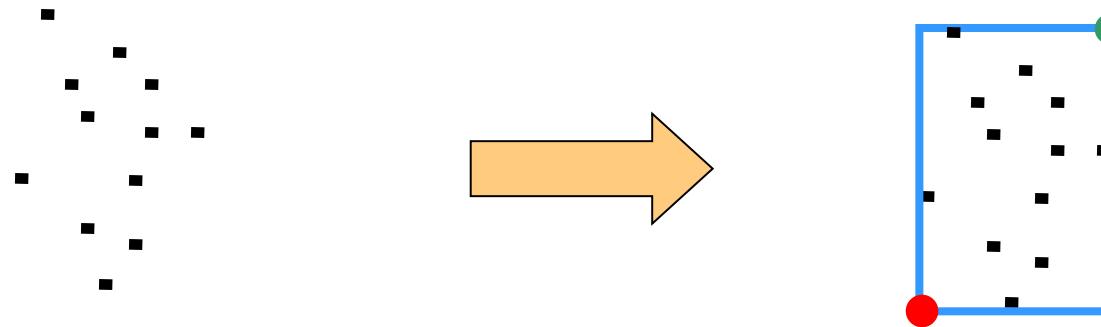
- ▶ R-Trees can organize any-dimensional data by representing the data by a minimum bounding box.
- ▶ Each node bounds it's children. A node can have many objects in it
- ▶ The leaves point to the actual objects (stored on disk probably)
- ▶ The height is always $\log n$ (it is height balanced)

R Tree Properties

- ▶ Leaf entry = < n-dimensional box, rid >
 - ▶ This is Alternative (2), with *key value* being a box.
 - ▶ Box is the tightest bounding box for a data object.
- ▶ Non-leaf entry = < n-dimensional box, ptr to child node >
 - ▶ Box covers all boxes in child node (in fact, subtree).
- ▶ All leaves at same distance from root.
- ▶ Nodes can be kept 50% full (except root).
 - ▶ Can choose a parameter m that is $\leq 50\%$, and ensure that every node is at least $m\%$ full.

Bounding Rectangle

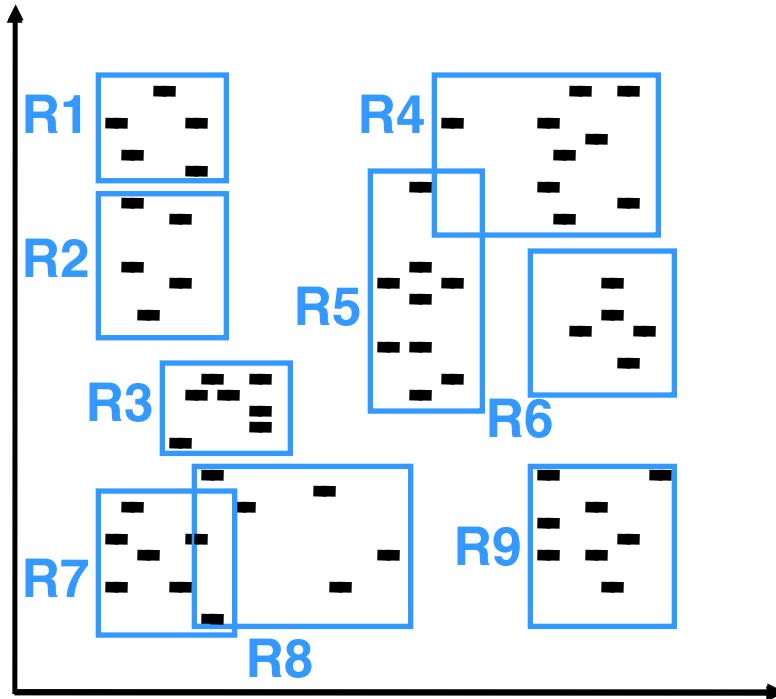
- ▶ Suppose we have a cluster of points in 2-D space...
 - ▶ We can build a “box” around points. The smallest box (which is axis parallel) that contains all the points is called a Minimum Bounding Rectangle (MBR)
 - ▶ also known as minimum bounding box



$$\text{MBR} = \{(L.x, L.y), (U.x, U.y)\}$$

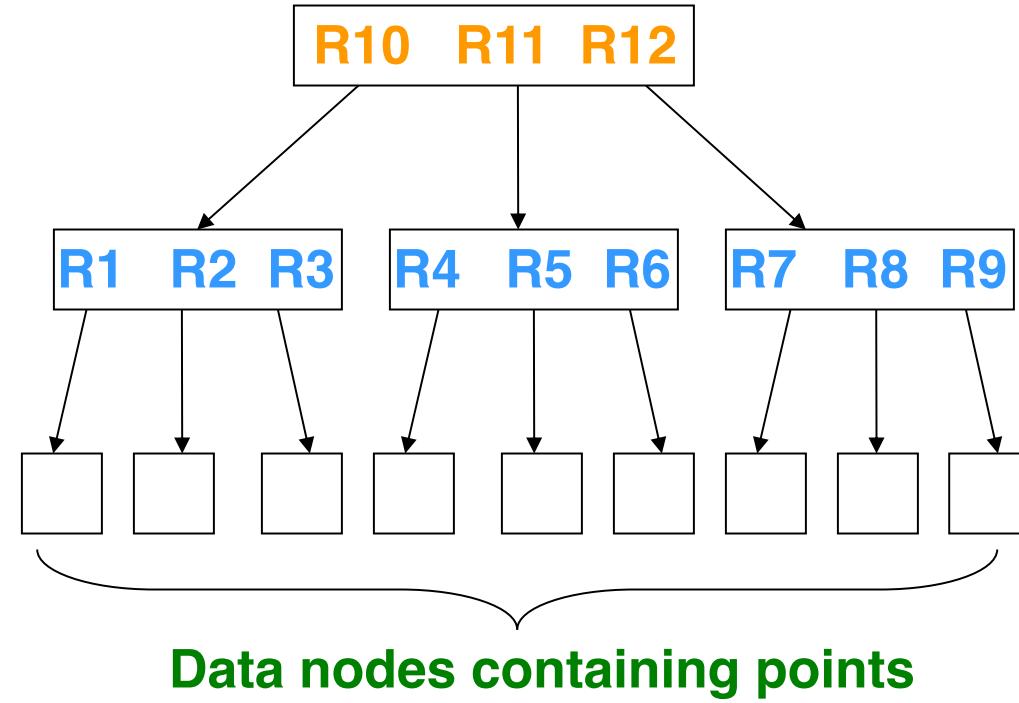
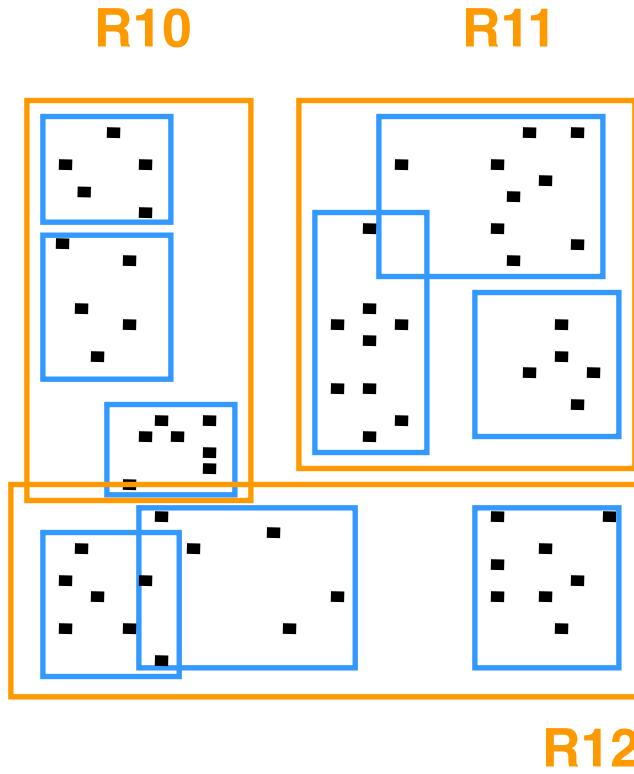
Clustering Points

- We can group clusters of datapoints into MBRs
 - Can also handle line-segments, rectangles, polygons, in addition to points



We can further recursively group MBRs into larger MBRs....

R-Tree Structure

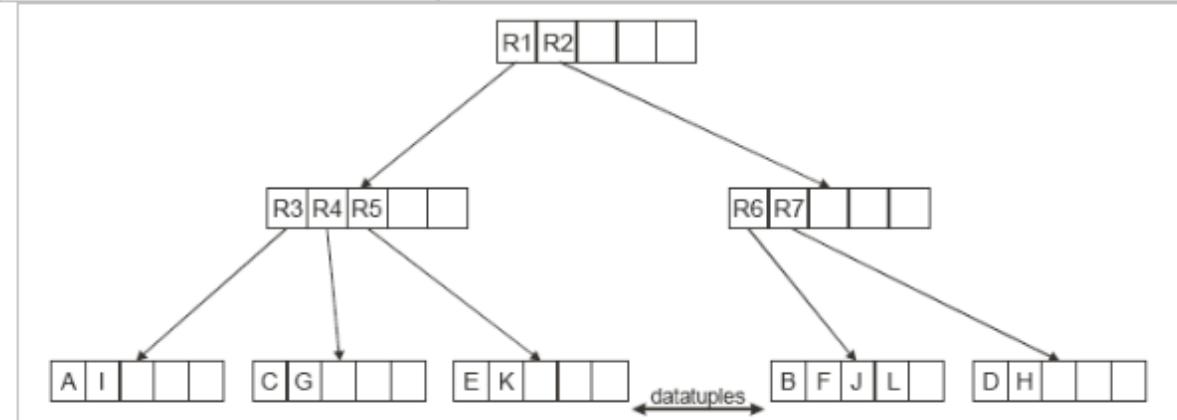


R Trees

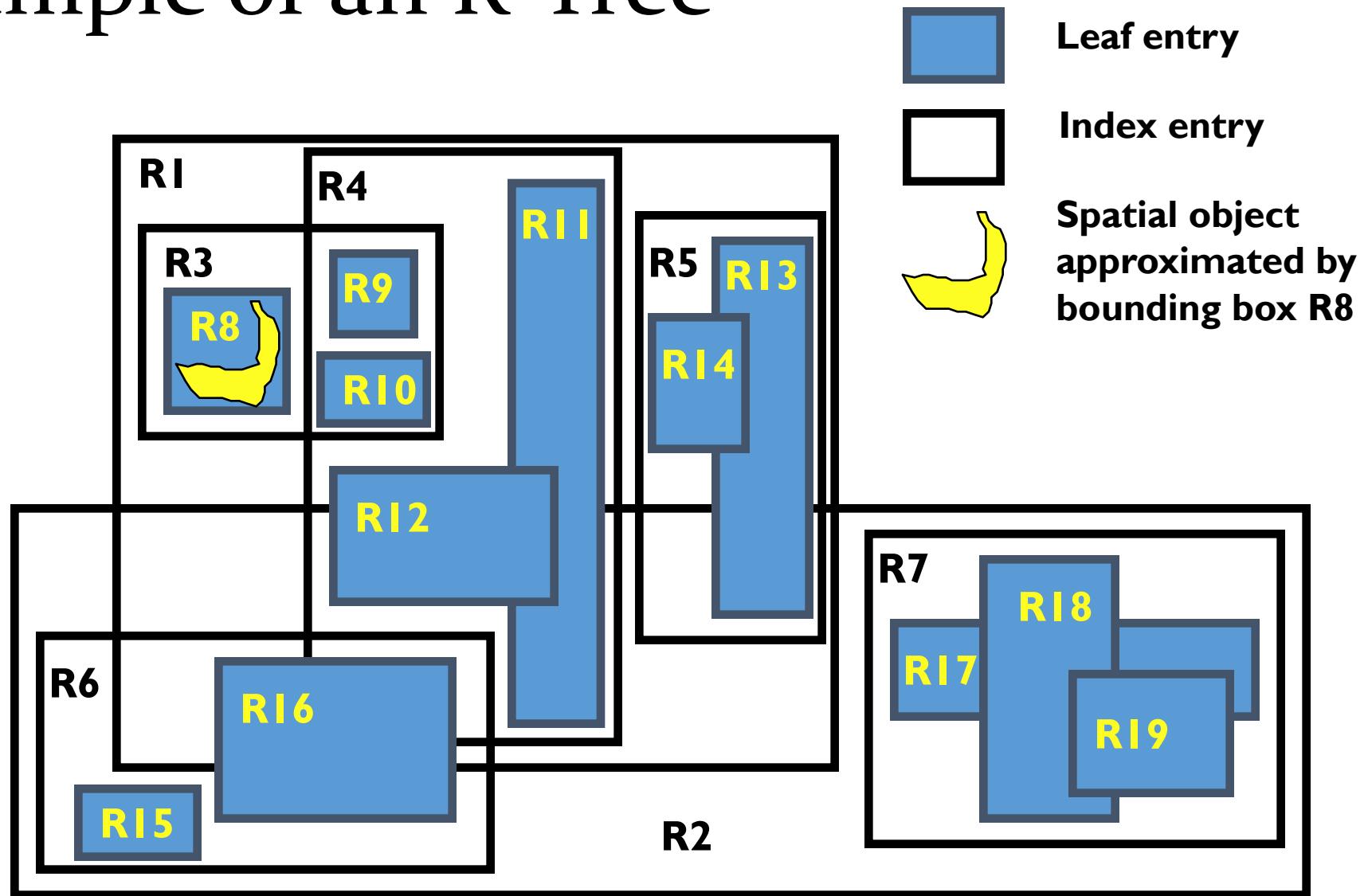
- A rectangular **bounding box** (a.k.a. MBR) is associated with each tree node.
 - Bounding box of a leaf node is a minimum sized rectangle that contains all the rectangles/polygons associated with the leaf node.
 - The bounding box associated with a non-leaf node contains the bounding box associated with all its children.
 - Bounding box of a node serves as its key in its parent node (if any)
 - *Bounding boxes of children of a node are allowed to overlap*
- A polygon is stored only in one node, and the bounding box of the node must contain the polygon
 - The storage efficiency of R-trees is better than that of k-d trees or quadtrees since a polygon is stored only once

R-Tree Example

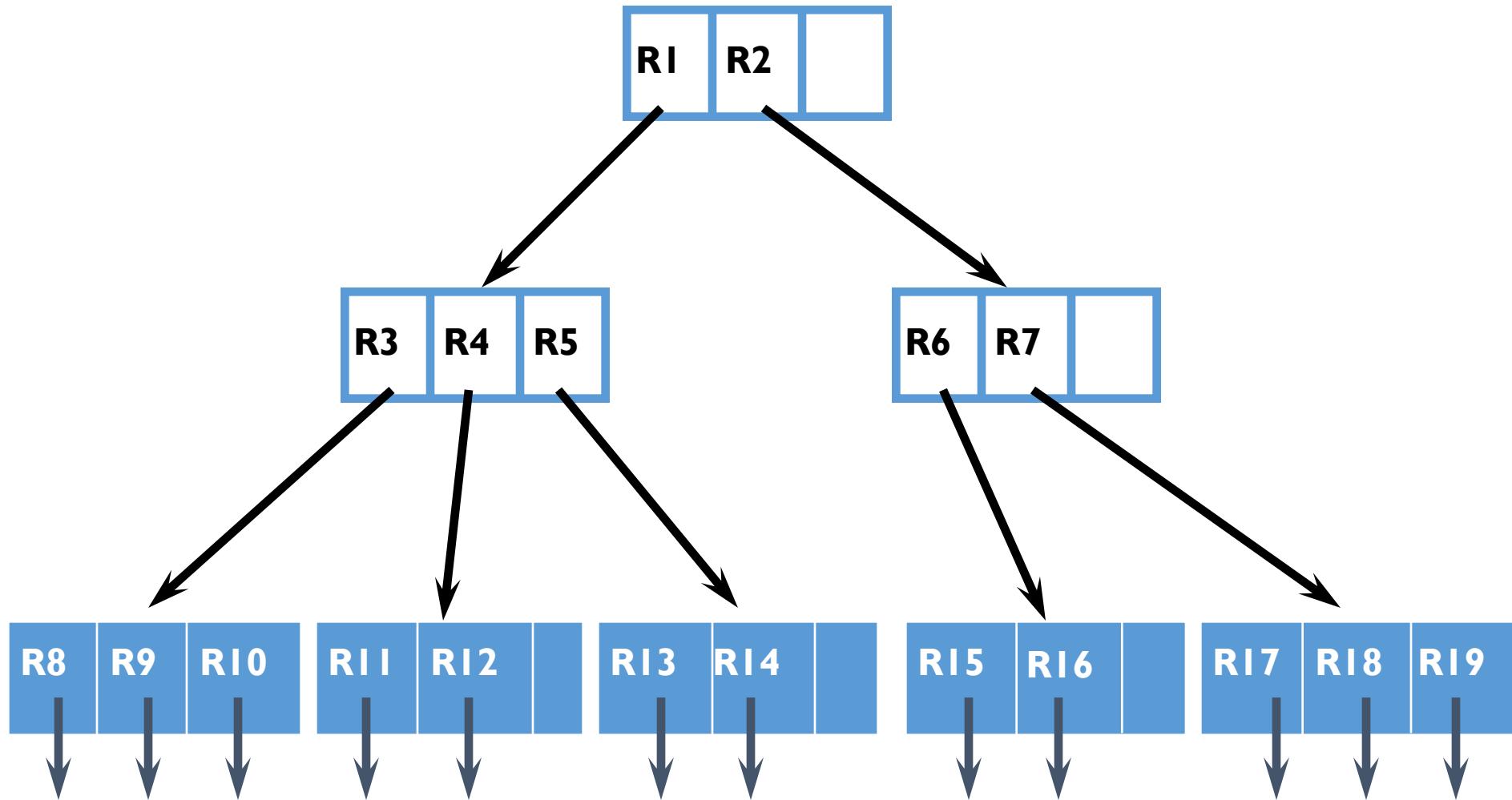
name	semester	credits
A	8	100
B	4	10
C	6	35
D	1	10
E	6	40
F	5	45
G	7	85
H	3	20
I	10	70
J	2	30
K	8	50
L	4	50



Example of an R-Tree

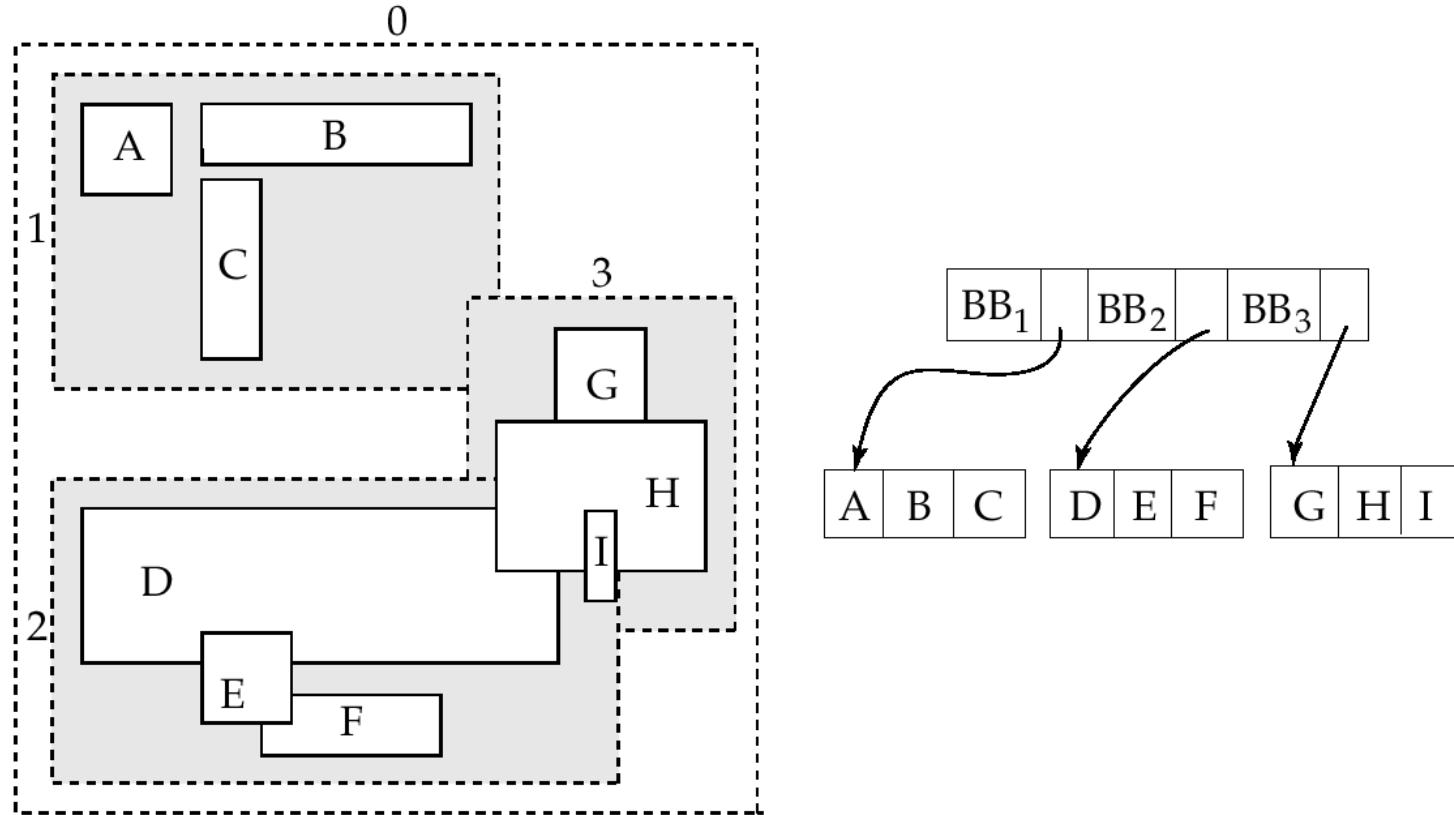


Example R-Tree (Contd.)



Example R-Tree

A set of rectangles (solid line) and the bounding boxes (dashed line) of the nodes of an R-tree for the rectangles. The R-tree is shown on the right.



Search for Objects Overlapping

Start at **root**.

1. If current node is non-leaf, for each entry $\langle E, \text{ptr} \rangle$, if **box** E overlaps Q , search subtree identified by **ptr**.
2. If current node is leaf, for each entry $\langle E, \text{rid} \rangle$, if E overlaps Q , rid identifies an object that might overlap Q .

*Note: May have to search **several** subtrees at each node!
(In contrast, a B-tree equality search goes to just one leaf.)*

Improving Search Using Constraints

- ▶ It is convenient to **store boxes** in the R-tree as approximations of arbitrary regions, because boxes can be represented compactly.
- ▶ But why not use **convex polygons** to approximate query regions more accurately?
 - ▶ Will reduce overlap with nodes in tree, and reduce the number of nodes fetched by avoiding some branches altogether.
 - ▶ Cost of overlap test is higher than bounding box intersection, but it is a main-memory cost, and can actually be done quite efficiently. Generally a win.

Insert Entry <B, ptr>

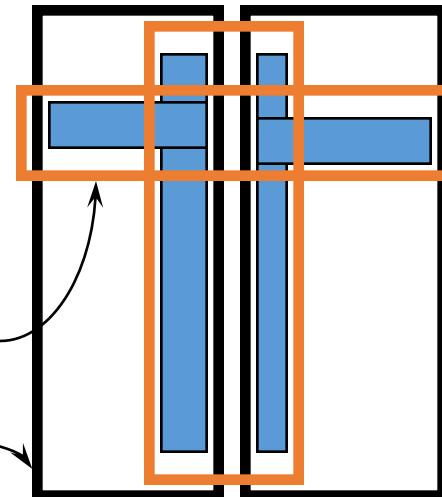
- ▶ Start at root and go down to “best-fit” leaf L.
 - ▶ Go to child whose box needs least enlargement to cover B; resolve ties by going to smallest area child.
- ▶ If best-fit leaf L has space, insert entry and stop. Otherwise, split L into L1 and L2.
 - ▶ Adjust entry for L in its parent so that the box now covers (only) L1.
 - ▶ Add an entry (in the parent node of L) for L2. (This could cause the parent node to recursively split.)

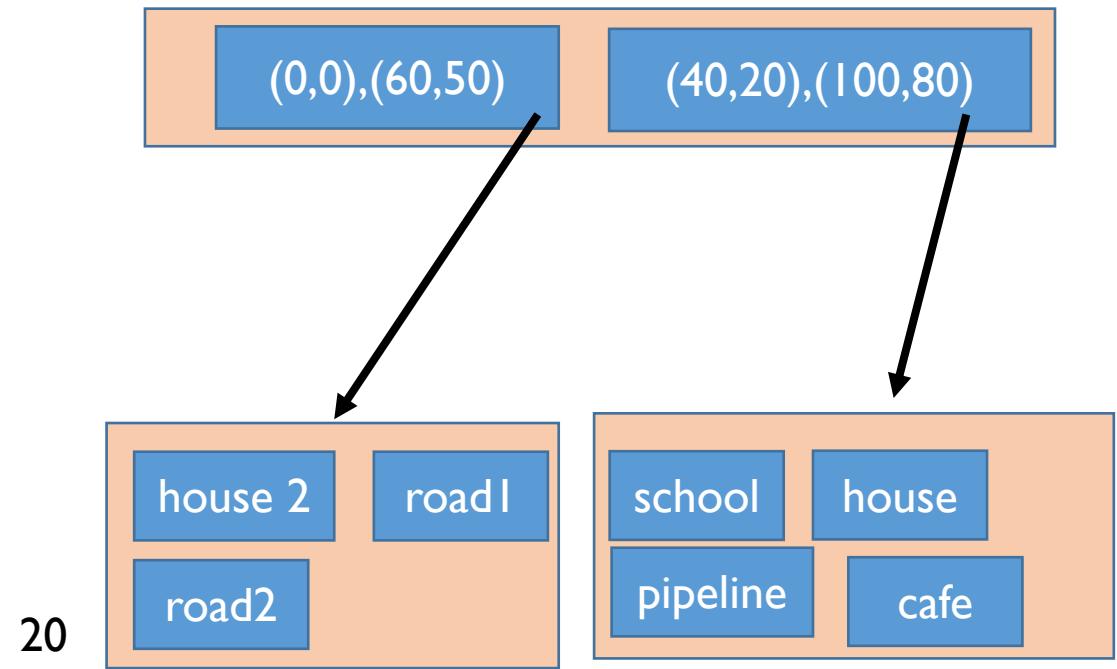
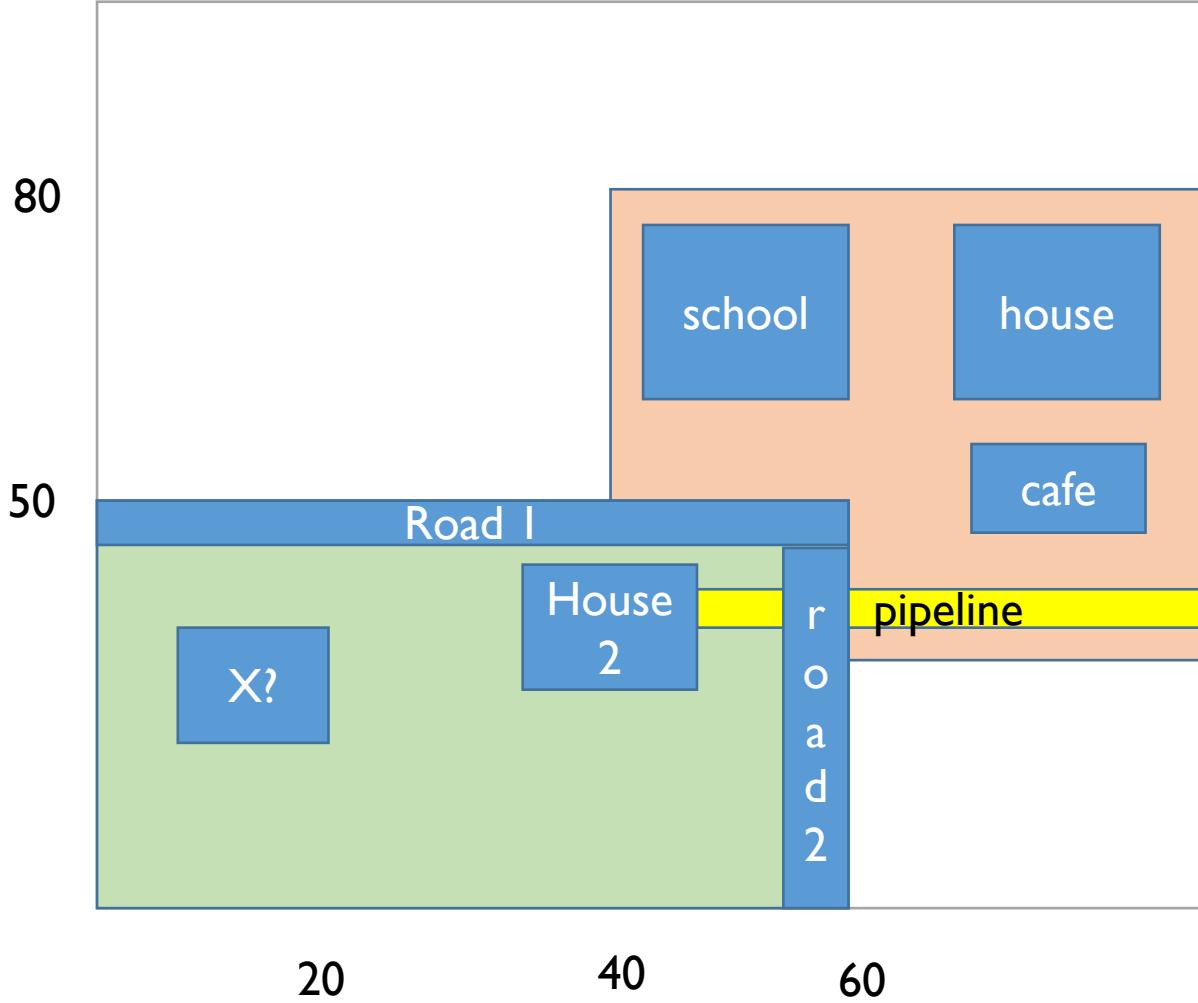
Splitting a Node During Insertion

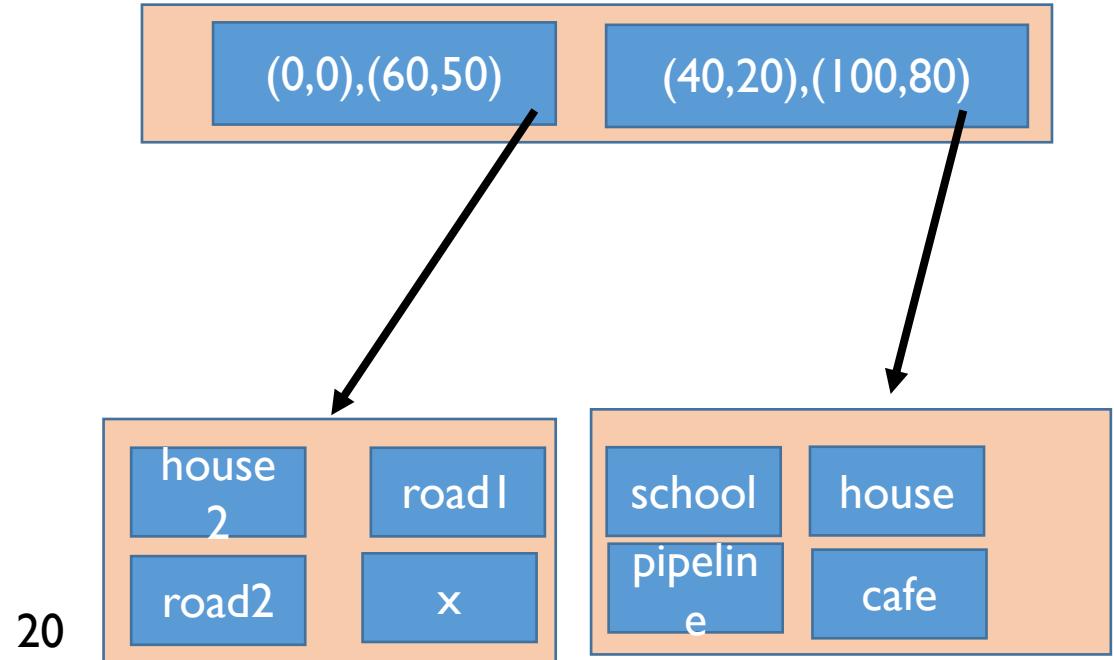
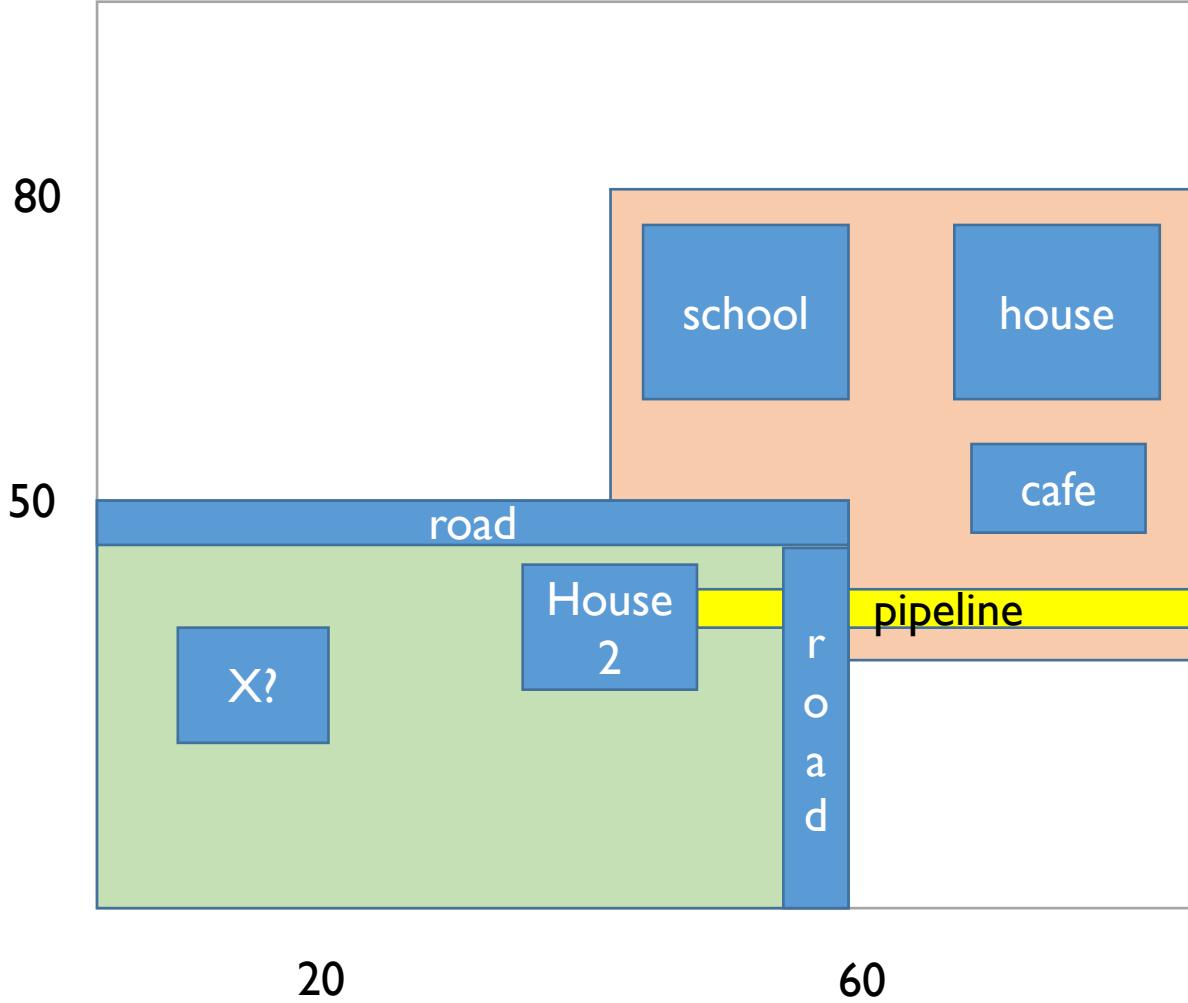
- ▶ The entries in node L plus the newly inserted entry must be distributed between L1 and L2.
- ▶ Goal is to reduce likelihood of both L1 and L2 being searched on subsequent queries.
- ▶ Idea: Redistribute so as to **minimize area** of L1 plus area of L2.

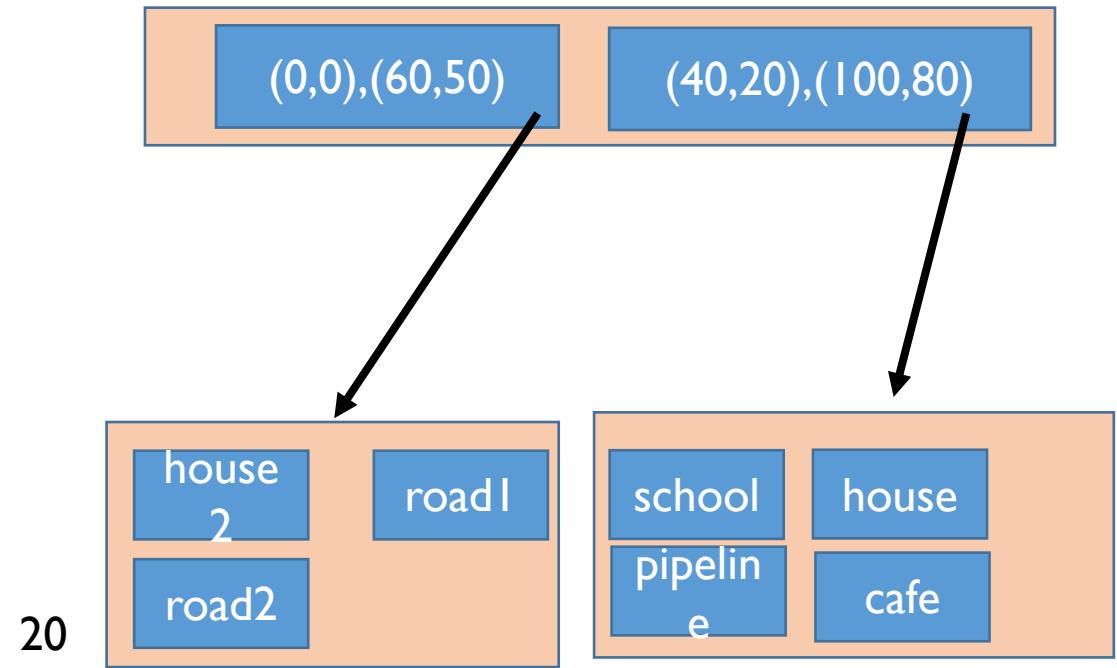
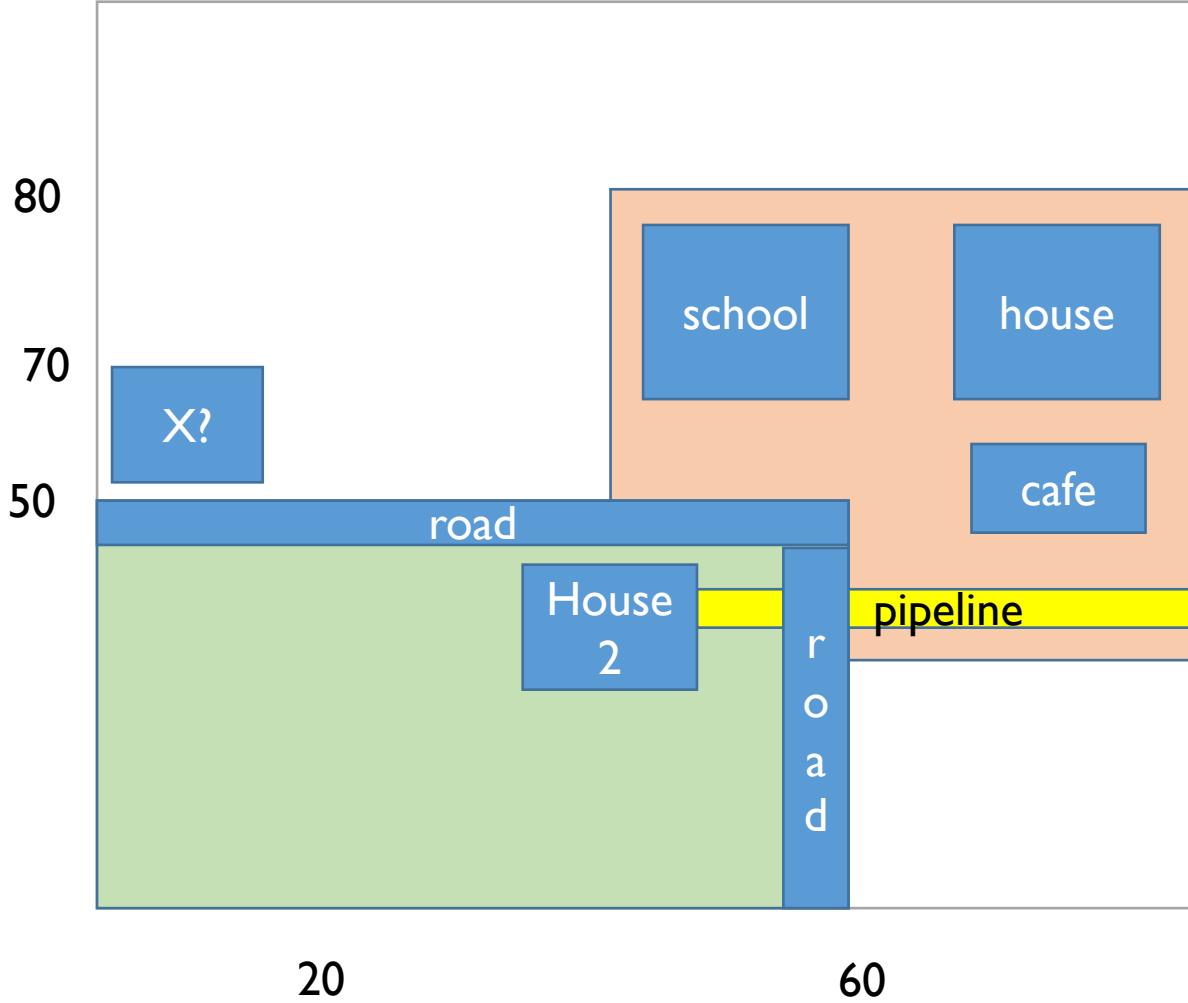
Exhaustive algorithm is too slow;
quadratic and linear heuristics are
described in the paper.

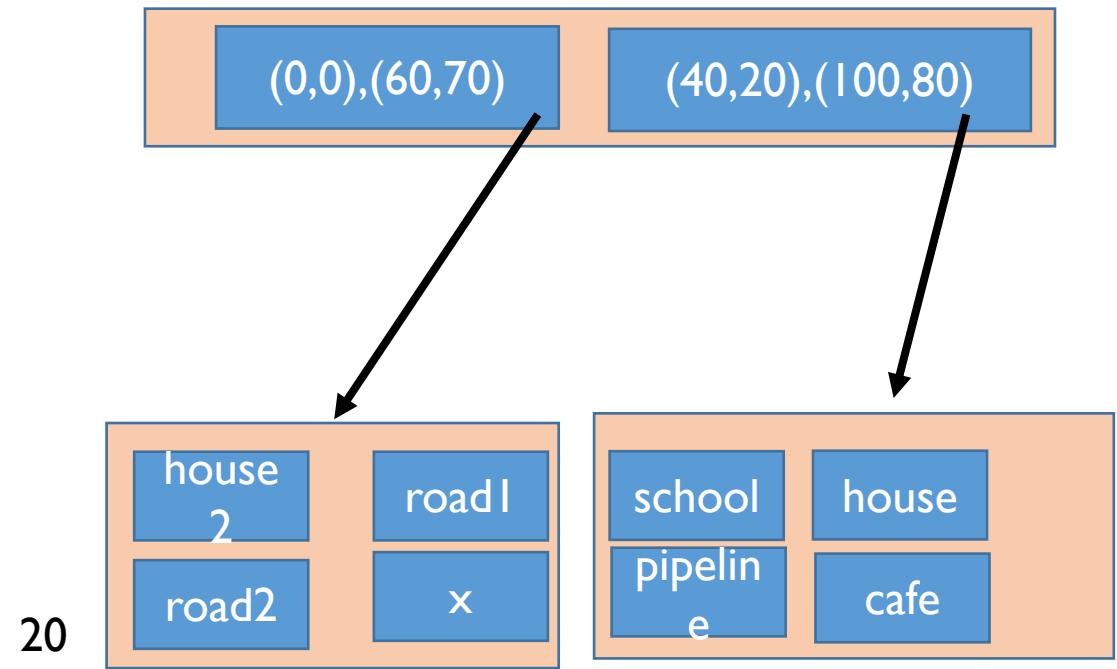
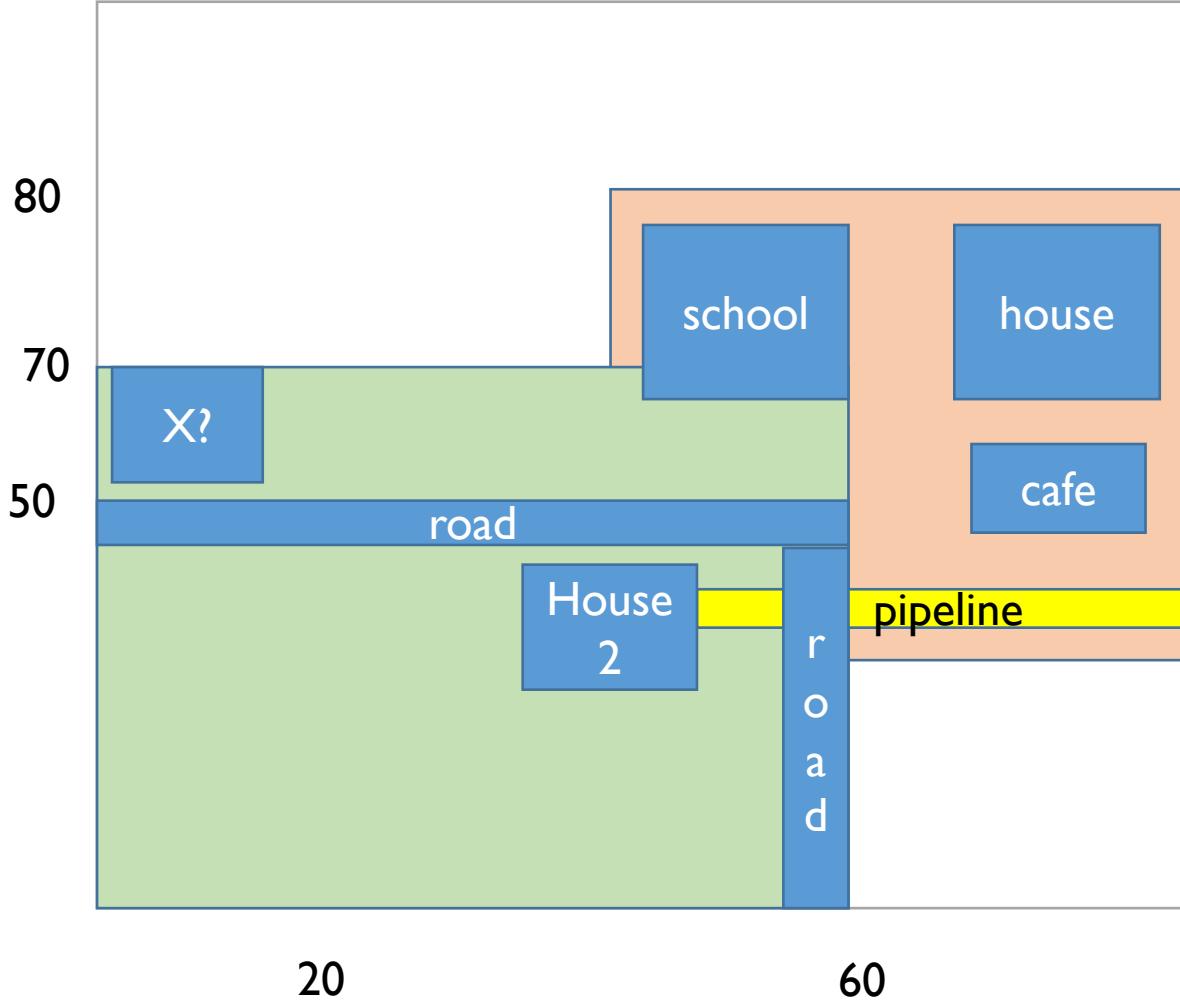
GOOD SPLIT!
BAD!

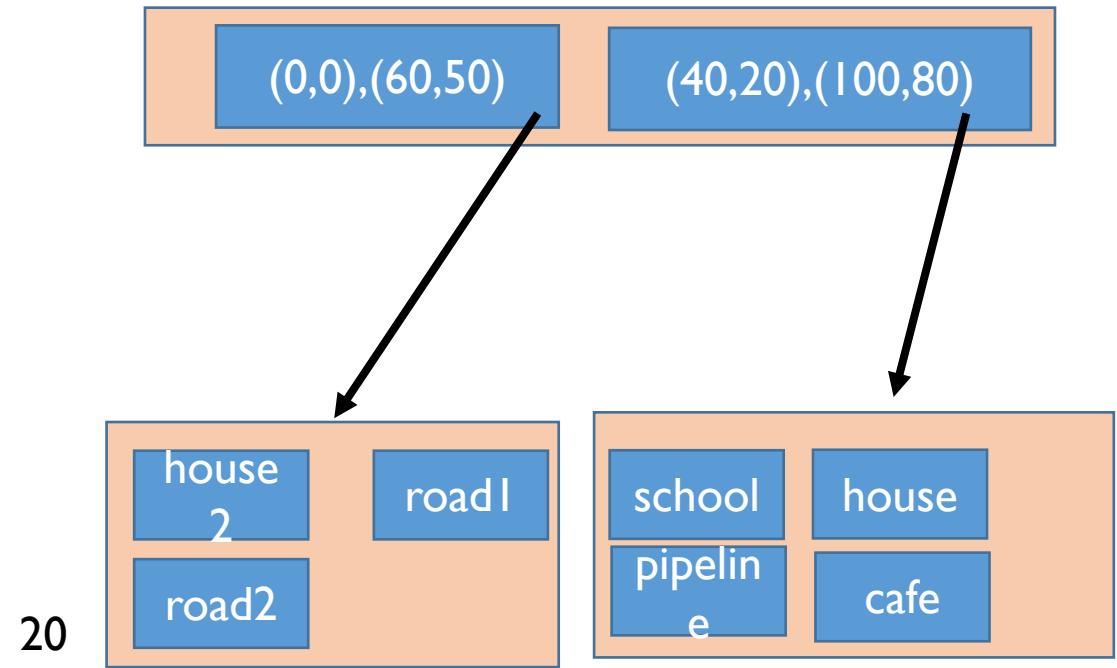
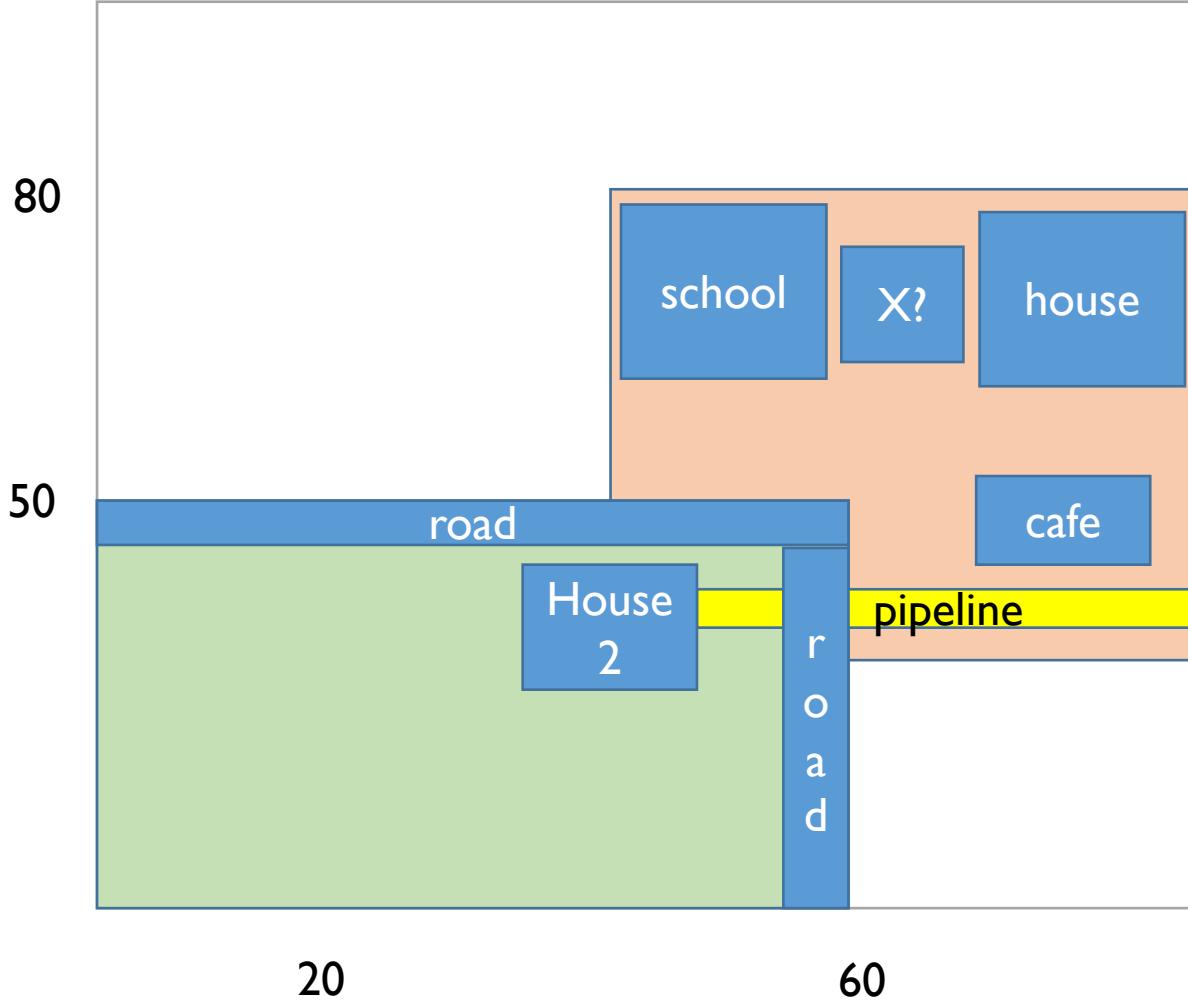


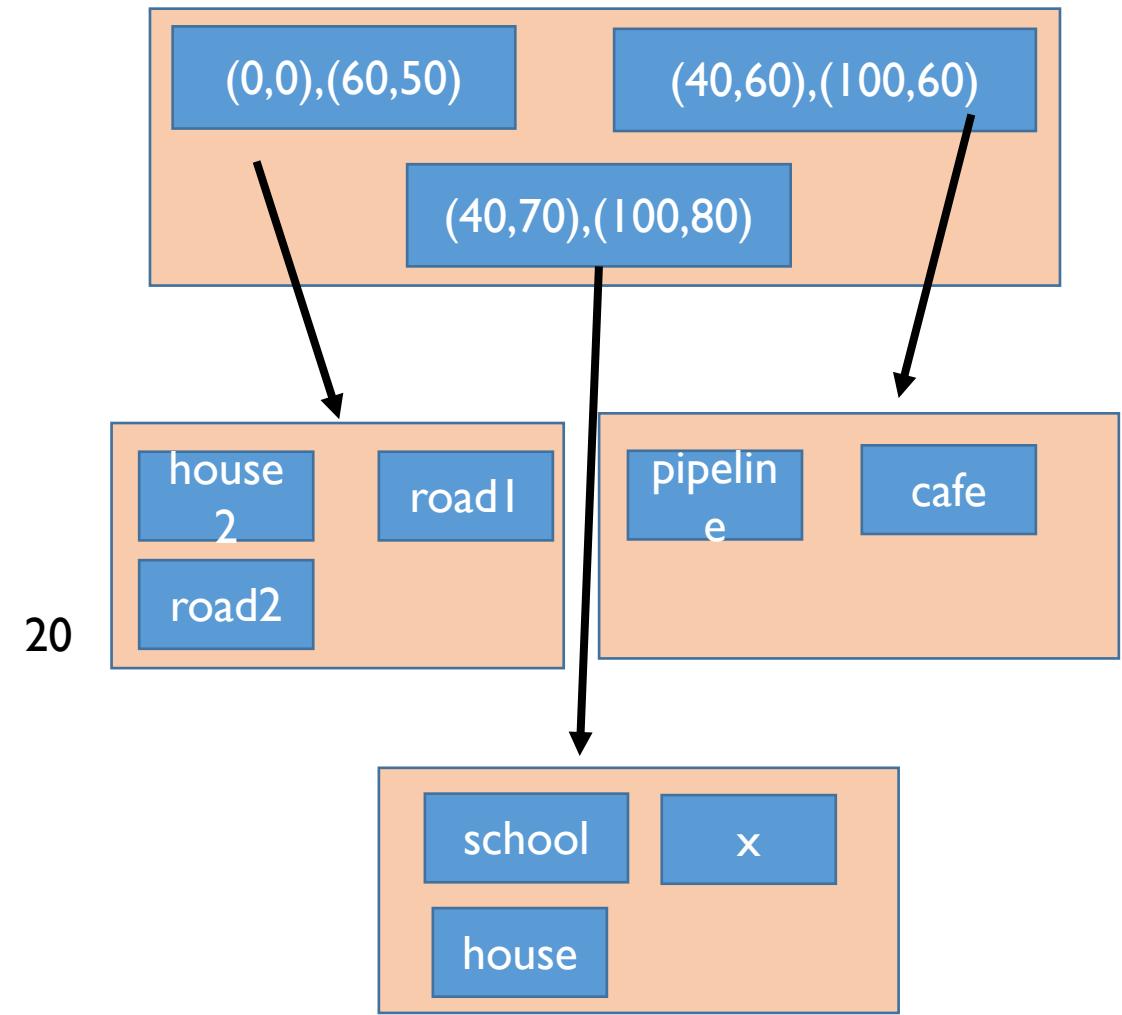
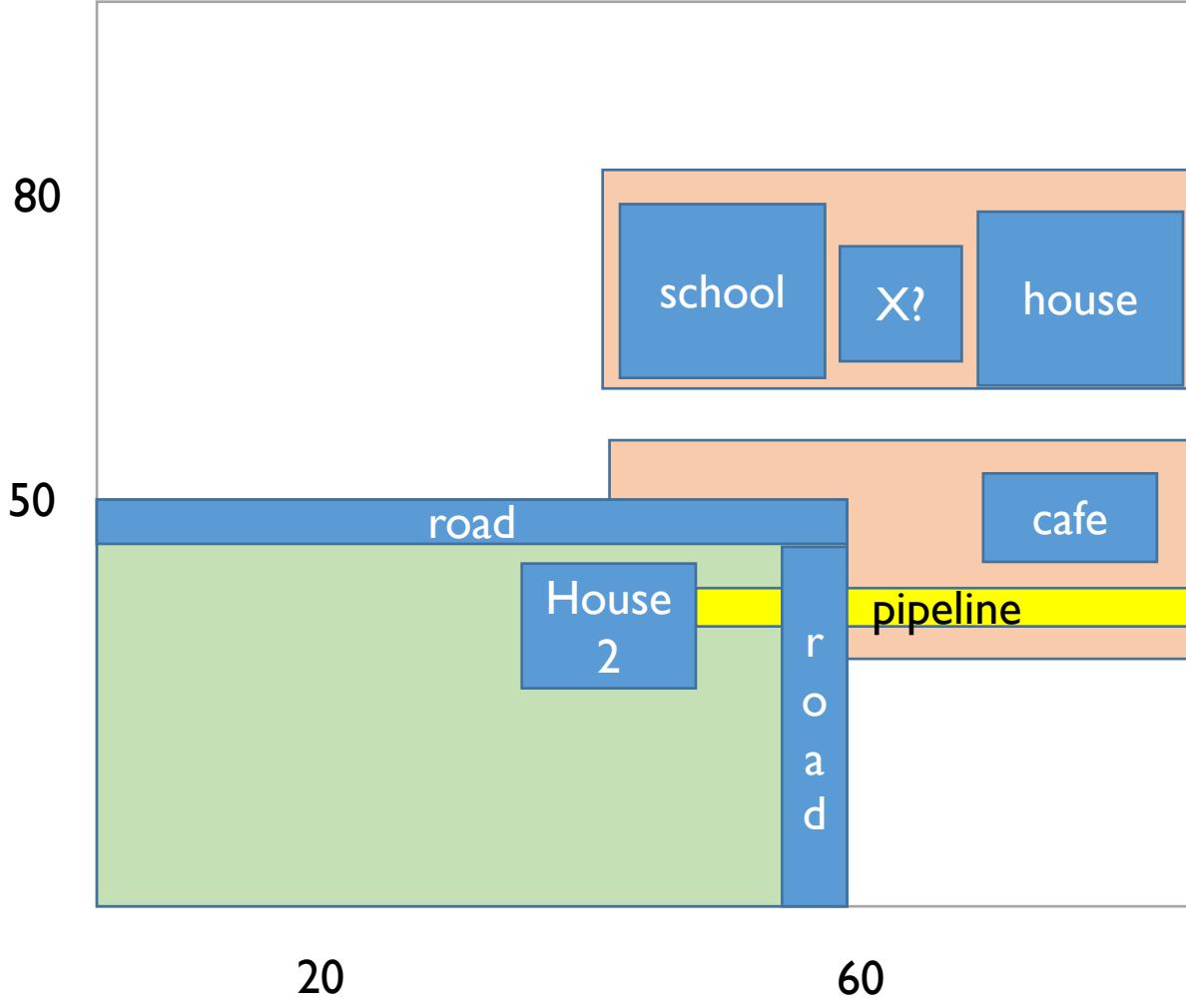












R-Tree Variants

- ▶ The R* tree uses the concept of forced reinserts to reduce overlap in tree nodes. When a node overflows, instead of splitting:
 - ▶ Remove some (say, 30% of the) entries and reinsert them into the tree.
 - ▶ Could result in all reinserted entries fitting on some existing pages, avoiding a split.
- ▶ R* trees also use a different heuristic, minimizing box perimeters rather than box areas during insertion.

Indexing High-Dimensional Data

- ▶ Typically, high-dimensional datasets are collections of points, not regions.
 - ▶ E.g., Feature vectors in multimedia applications.
 - ▶ Very sparse
- ▶ Nearest neighbor queries are common.
 - ▶ R-tree becomes worse than sequential scan for most datasets with more than a dozen dimensions

Comments on R-Trees

- ▶ Deletion consists of searching for the entry to be deleted, removing it, and if the node becomes under-full, deleting the node and then re-inserting the remaining entries.
- ▶ Overall, works quite well for 2 and 3 D datasets. Several variants (notably, R+ and R* trees) have been proposed; widely used.
- ▶ Can improve search performance by using a convex polygon to approximate query shape (instead of a bounding box) and testing for polygon-box intersection.

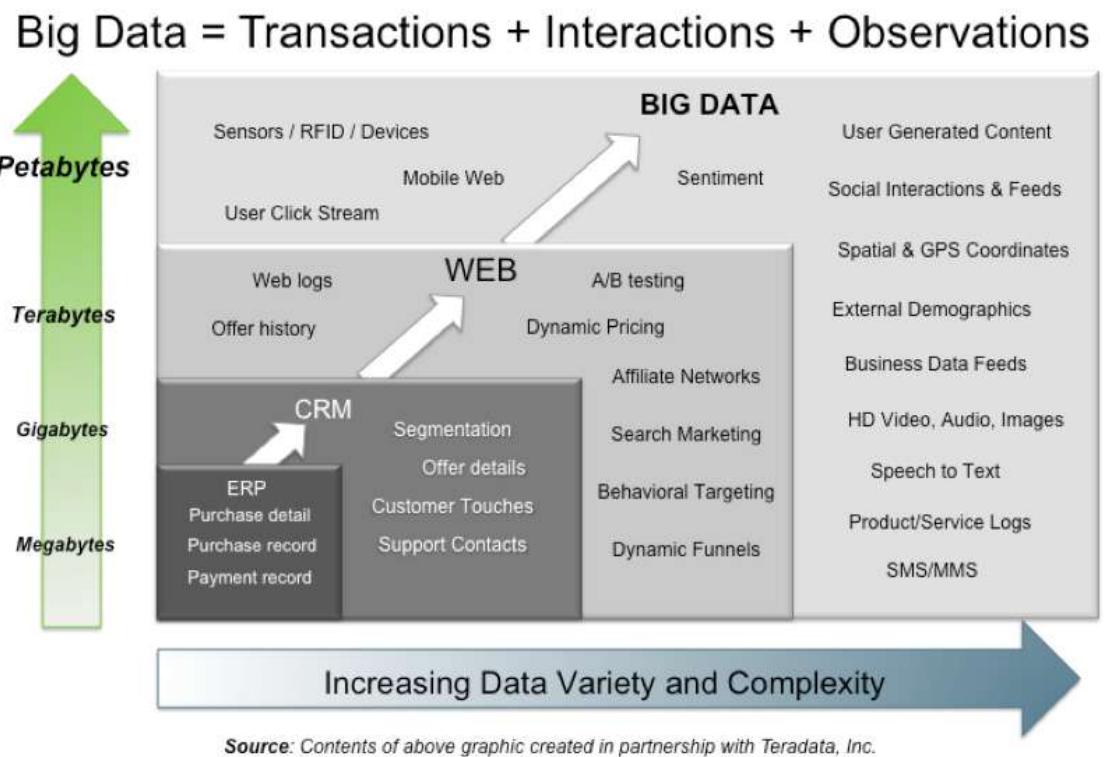
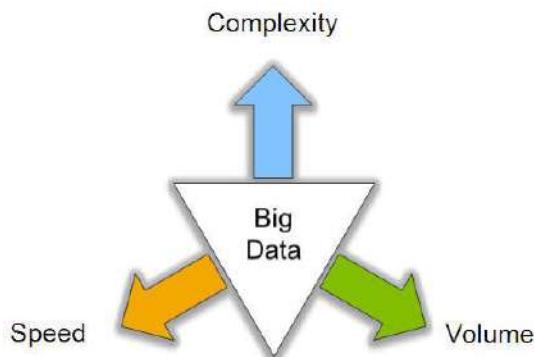
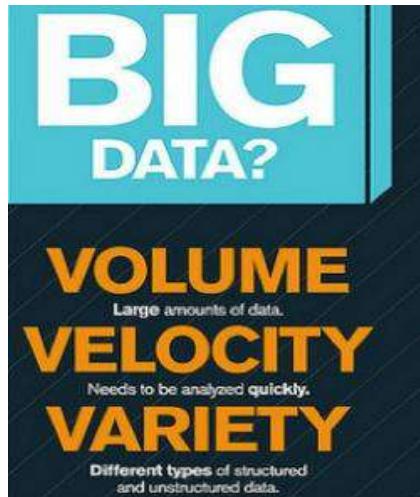
Introduction to Big Data

What's Big Data?

No single definition; here is from Wikipedia:

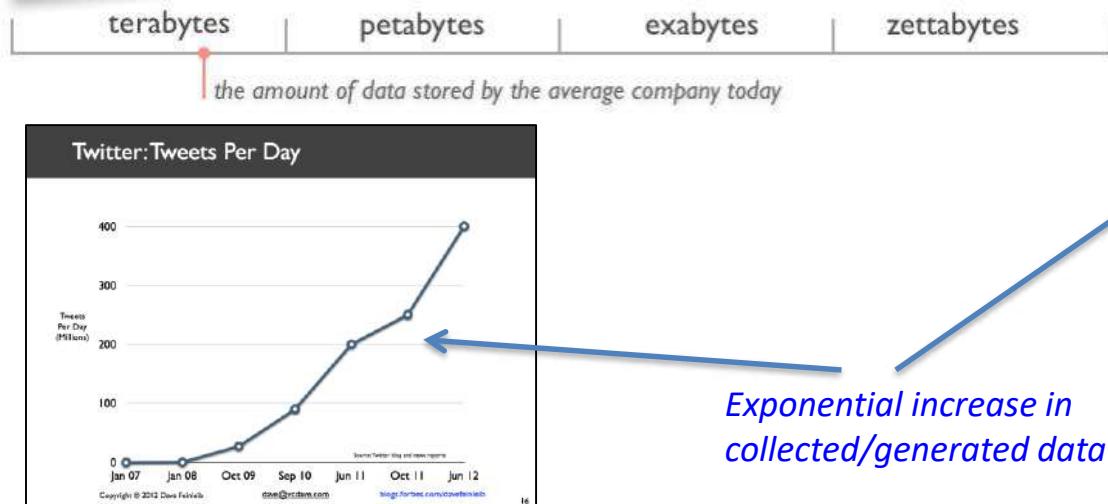
- **Big data** is the term for a collection of data sets so large and complex that it becomes difficult to process using on-hand database management tools or traditional data processing applications.
- The challenges include **capture, curation, storage, search, sharing, transfer, analysis, and visualization**.
- The trend to larger data sets is due to the additional information derivable from analysis of a single large set of related data, as compared to separate smaller sets with the same total amount of data, allowing correlations to be found to **"spot business trends, determine quality of research, prevent diseases, link legal citations, combat crime, and determine real-time roadway traffic conditions."**

Big Data: 3V's

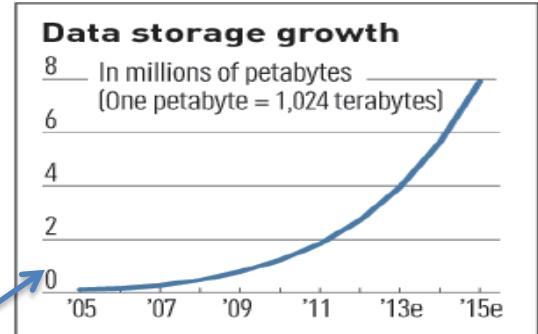
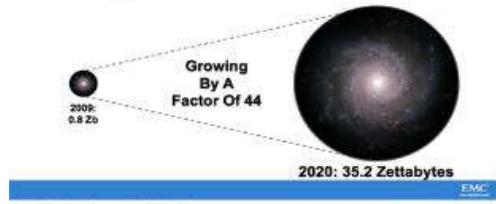


Volume (Scale)

- **Data Volume**
 - 44x increase from 2009 2020
 - From 0.8 zettabytes to 35zb
- Data volume is increasing exponentially



The Digital Universe 2009-2020





? TBs of
data every day

12+ TBs
of tweet data
every day

25+ TBs of
log data
every day

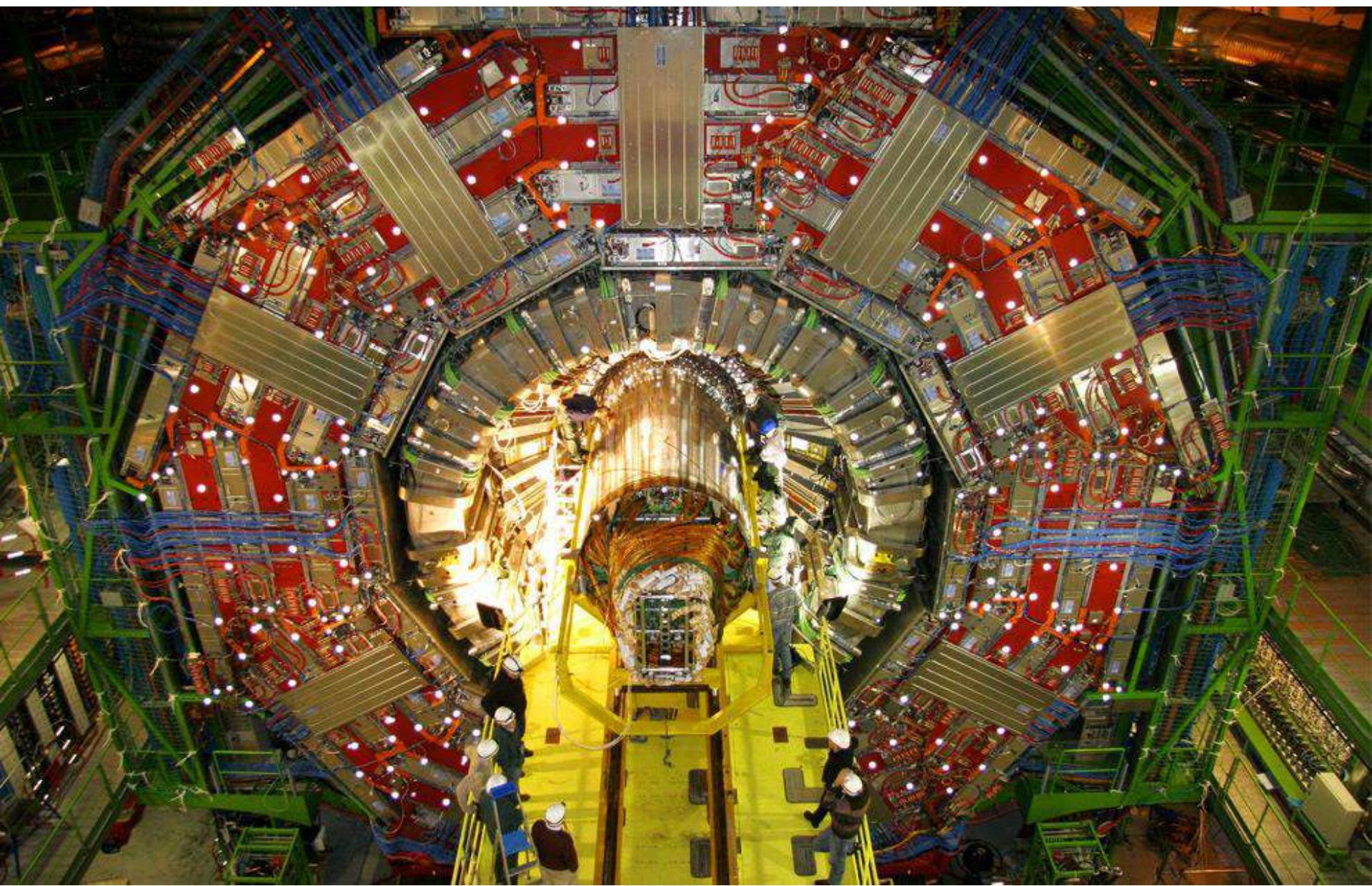
30 billion RFID
tags today
(1.3B in 2005)



4.6 billion
camera
phones
world wide

100s of
millions
of GPS
enabled
devices sold
annually

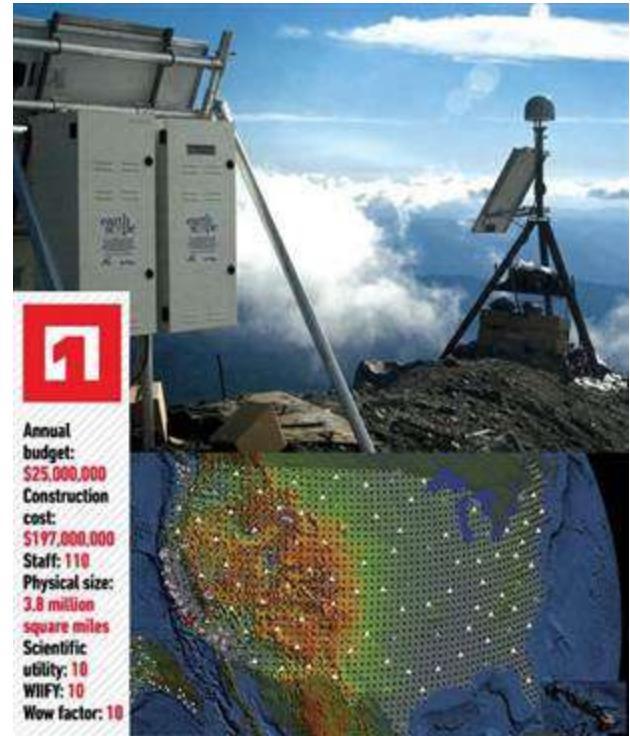
2+
billion
people on
the Web
by end
2011



CERN's Large Hydron Collider (LHC) generates 15 PB a year

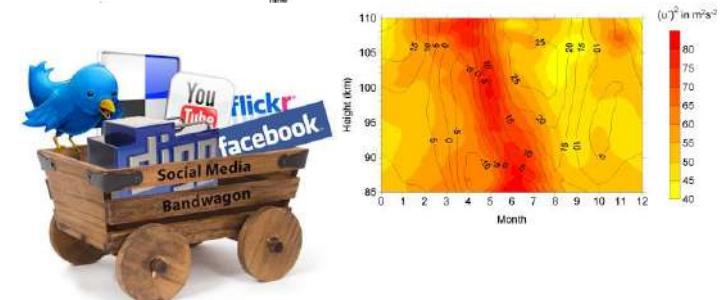
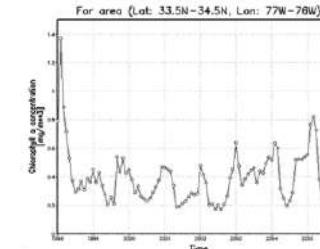
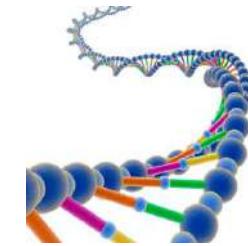
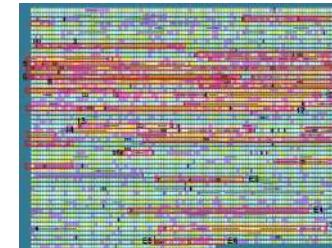
The Earthscope

- The Earthscope is the world's largest science project. Designed to track North America's geological evolution, this observatory records data over 3.8 million square miles, amassing 67 terabytes of data. It analyzes seismic slips in the San Andreas fault, sure, but also the plume of magma underneath Yellowstone and much, much more.
http://www.msnbc.msn.com/id/44363598/ns/technology_and_science-future_of_technology/#.TmetOdQ-ul)



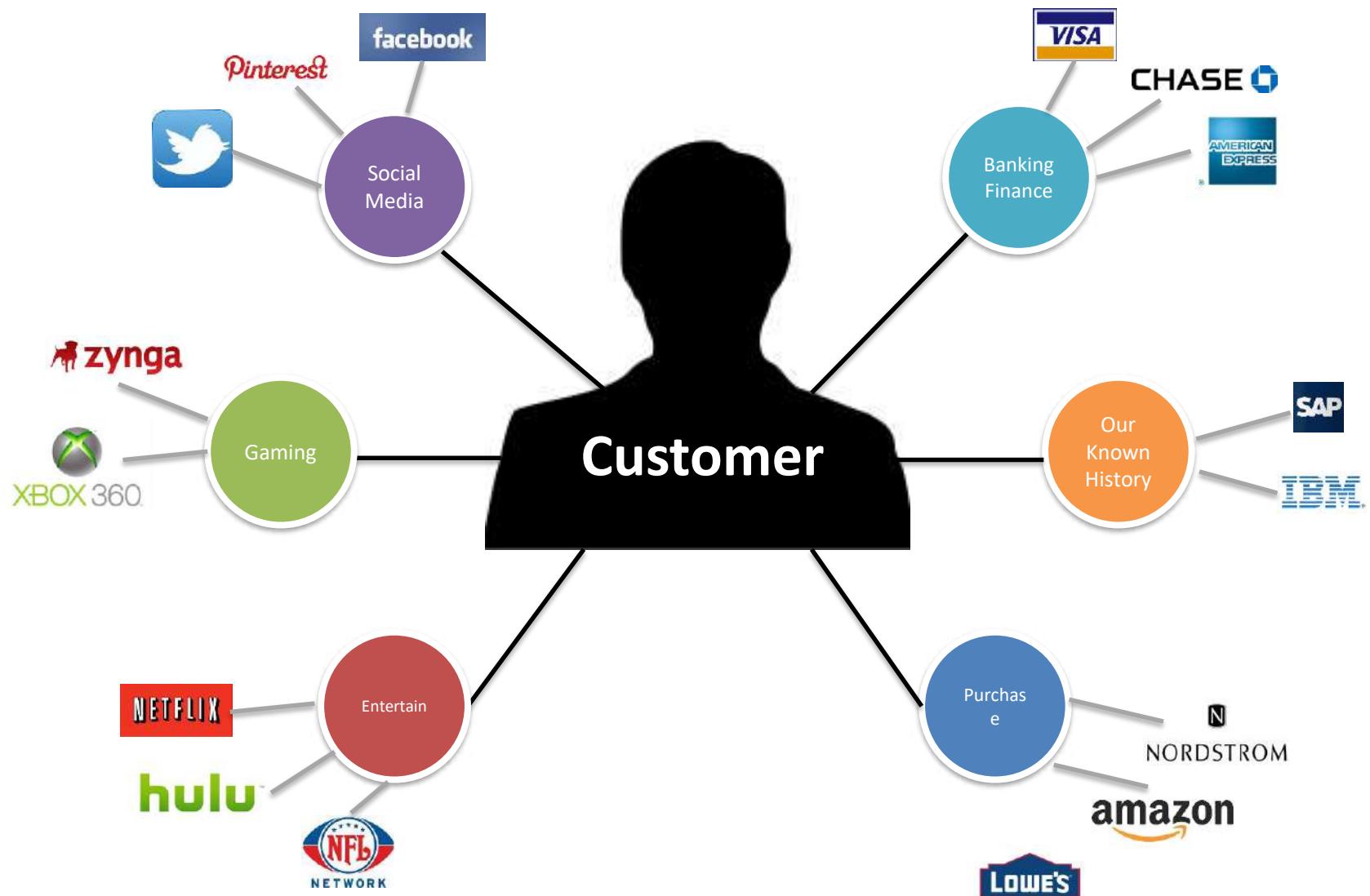
Variety (Complexity)

- Relational Data (Tables/Transaction/Legacy Data)
- Text Data (Web)
- Semi-structured Data (XML)
- Graph Data
 - Social Network, Semantic Web (RDF), ...
- Streaming Data
 - You can only scan the data once
- A single application can be generating/collecting many types of data
- Big Public Data (online, weather, finance, etc)



To extract knowledge → all these types of data need to linked together

A Single View to the Customer



Velocity (Speed)

- Data is generated fast and need to be processed fast
- Online Data Analytics
- Late decisions → missing opportunities
- **Examples**
 - **E-Promotions:** Based on your current location, your purchase history, what you like → send promotions right now for store next to you
 - **Healthcare monitoring:** sensors monitoring your activities and body → any abnormal measurements require immediate reaction



Real-time/Fast Data



Social media and networks
(all of us are generating data)



Scientific instruments
(collecting all sorts of data)



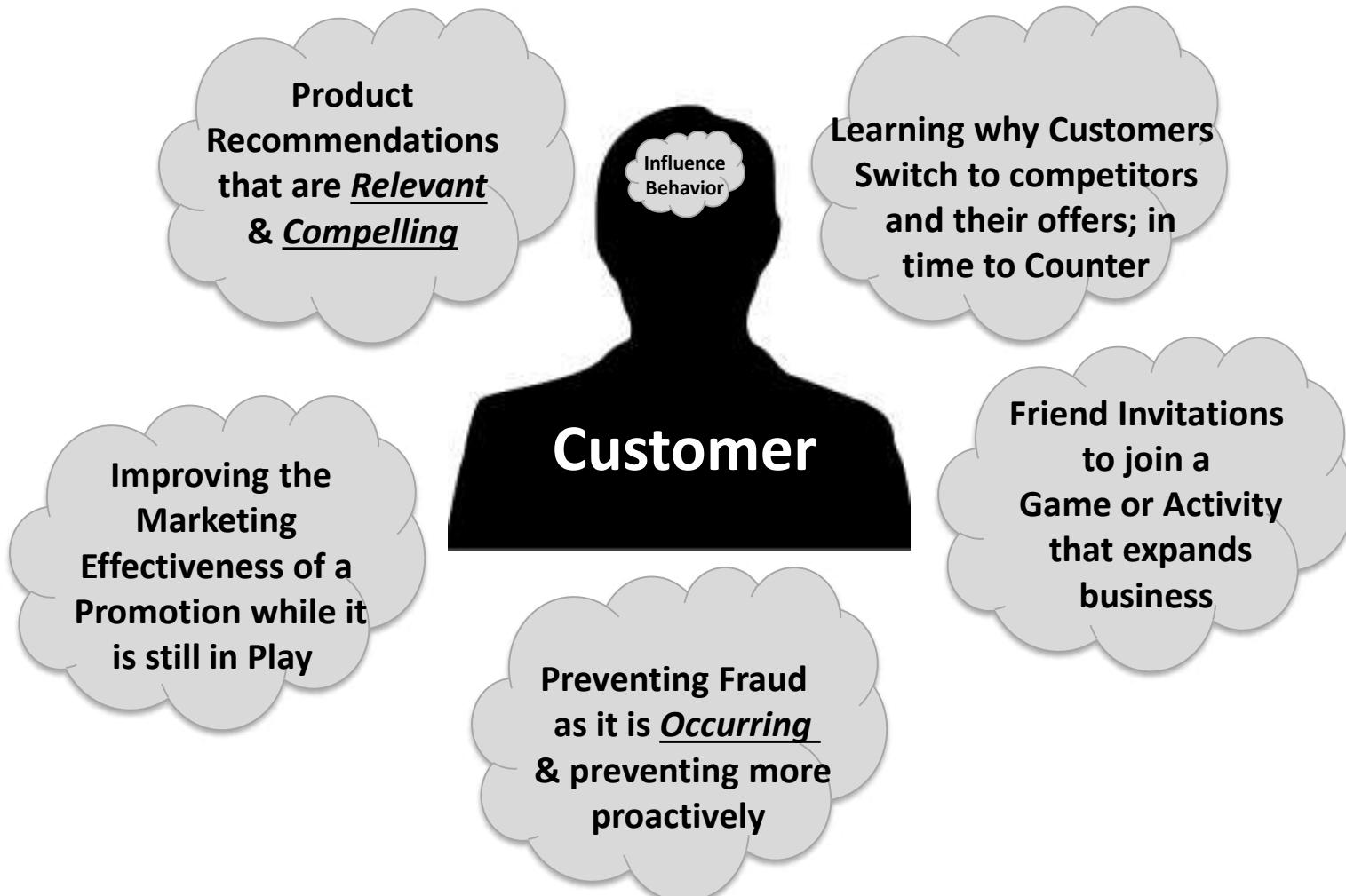
Mobile devices
(tracking all objects all the time)



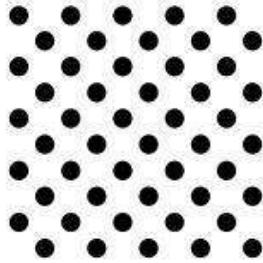
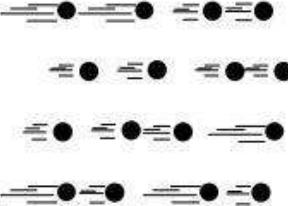
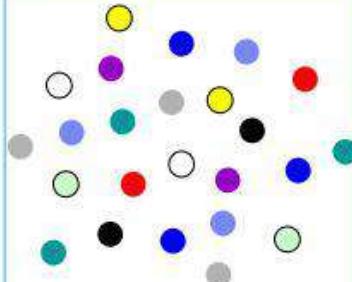
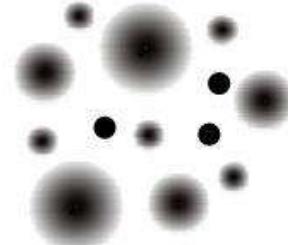
Sensor technology and networks
(measuring all kinds of data)

- The progress and innovation is no longer hindered by the ability to collect data
- But, by the ability to manage, analyze, summarize, visualize, and discover knowledge from the collected data in a timely manner and in a scalable fashion

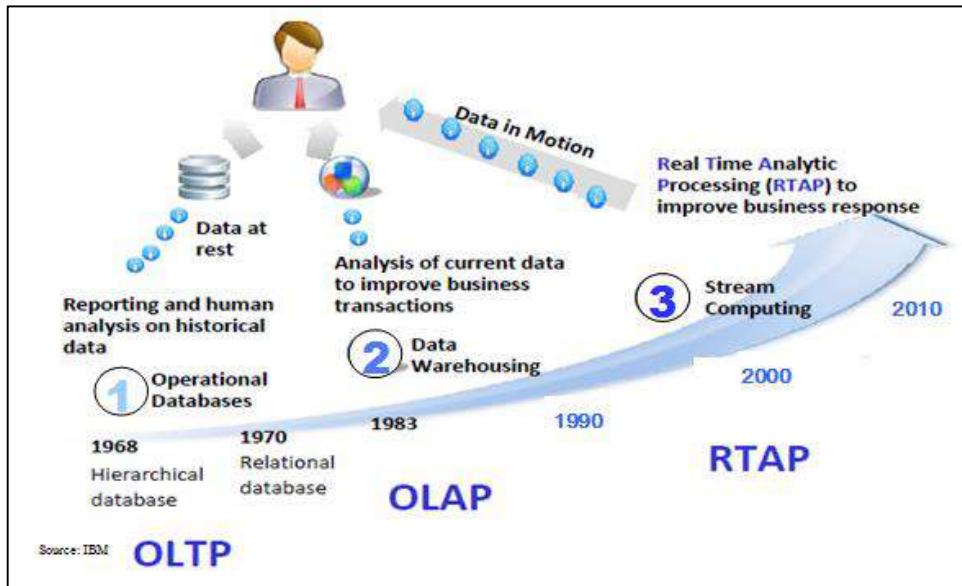
Real-Time Analytics/Decision Requirement



Some Make it 4V's

Volume	Velocity	Variety	Veracity*
			
Data at Rest Terabytes to exabytes of existing data to process	Data in Motion Streaming data, milliseconds to seconds to respond	Data in Many Forms Structured, unstructured, text, multimedia	Data in Doubt Uncertainty due to data inconsistency & incompleteness, ambiguities, latency, deception, model approximations

Harnessing Big Data



- **OLTP:** Online Transaction Processing (DBMSs)
- **OLAP:** Online Analytical Processing (Data Warehousing)
- **RTAP:** Real-Time Analytics Processing (Big Data Architecture & technology)

The Model Has Changed...

- **The Model of Generating/Consuming Data has Changed**

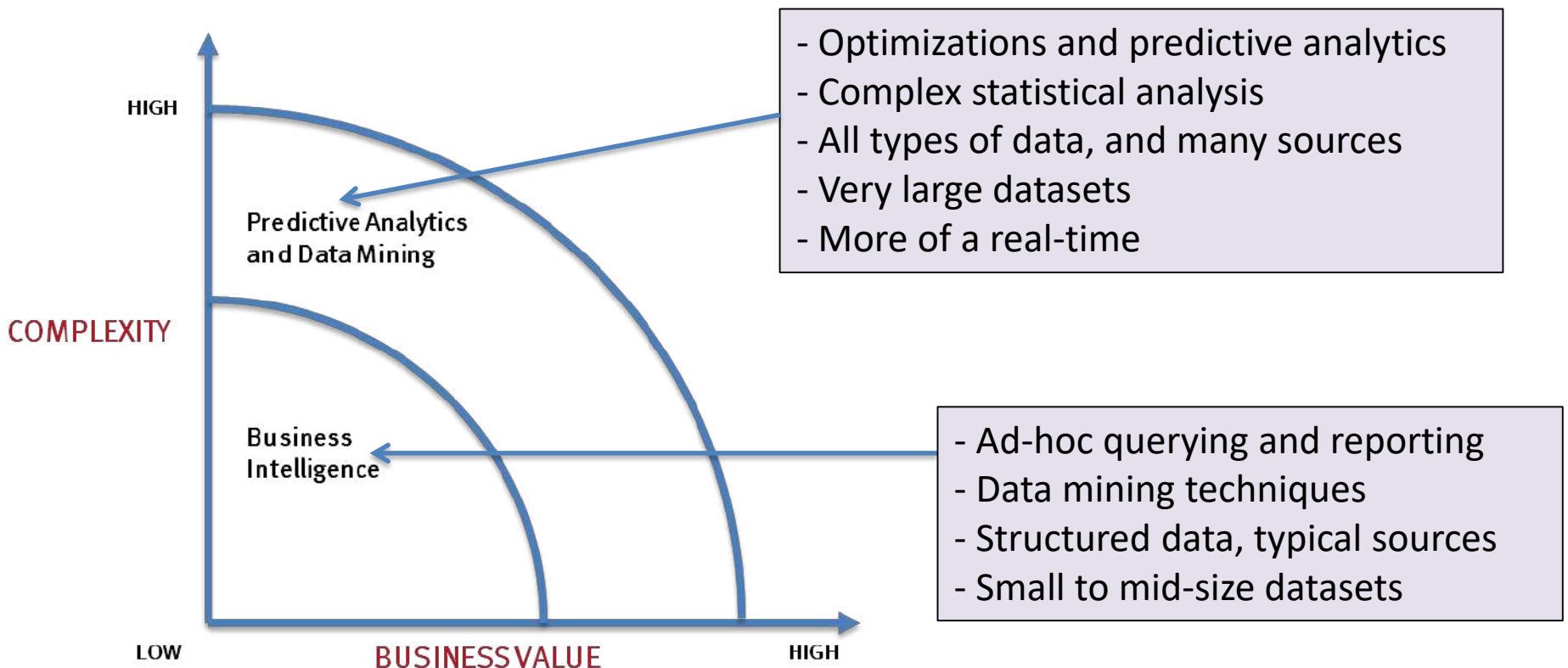
Old Model: Few companies are generating data, all others are consuming data



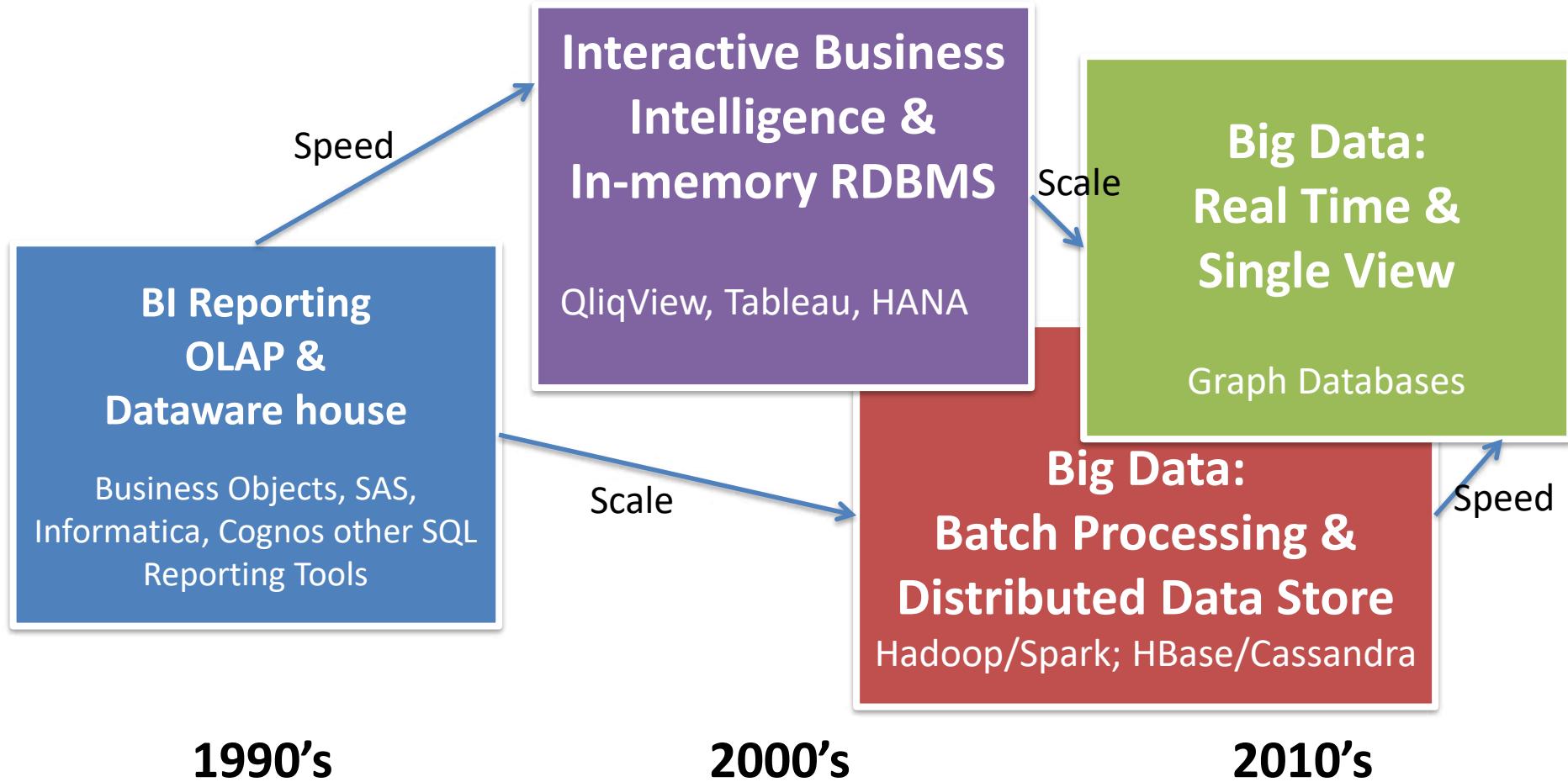
New Model: all of us are generating data, and all of us are consuming data



What's driving Big Data

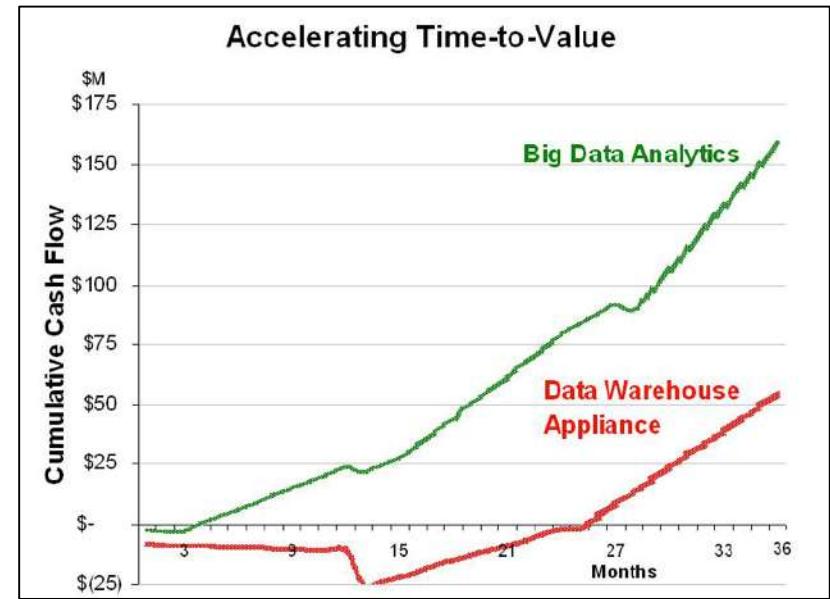


THE EVOLUTION OF BUSINESS INTELLIGENCE



Big Data Analytics

- Big data is more real-time in nature than traditional DW applications
- Traditional DW architectures (e.g. Exadata, Teradata) are not well-suited for big data apps
- Shared nothing, massively parallel processing, scale out architectures are well-suited for big data apps



The Big Data Landscape

Apps

Vertical Apps



Operational Intelligence



Data As A Service



Ad / Media Apps



TURNO

Business Intelligence



Analytics And Visualization



Infrastructure

Analytics Infrastructure



Operational Infrastructure



Infrastructure As A Service



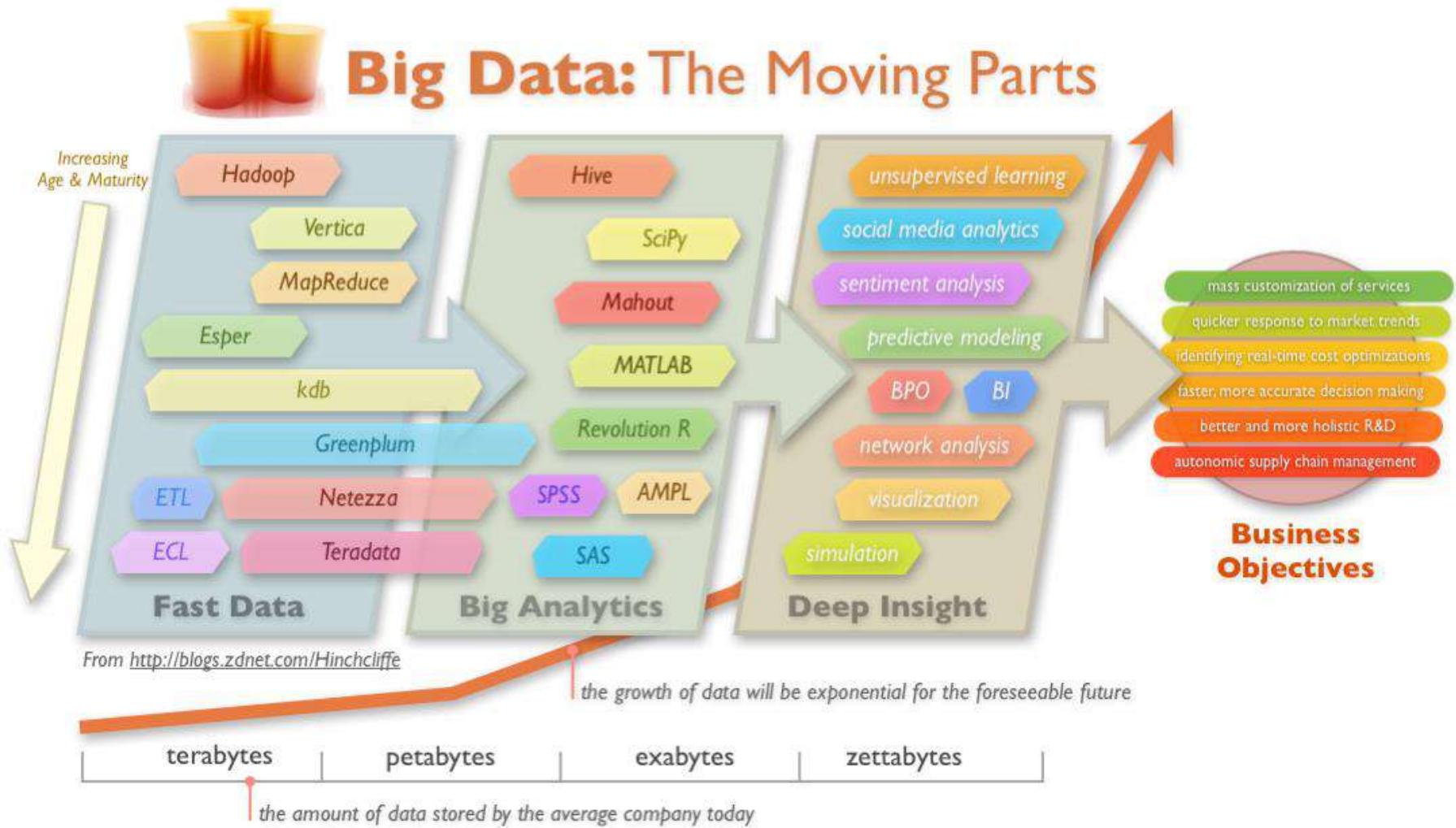
Structured Databases



Technologies

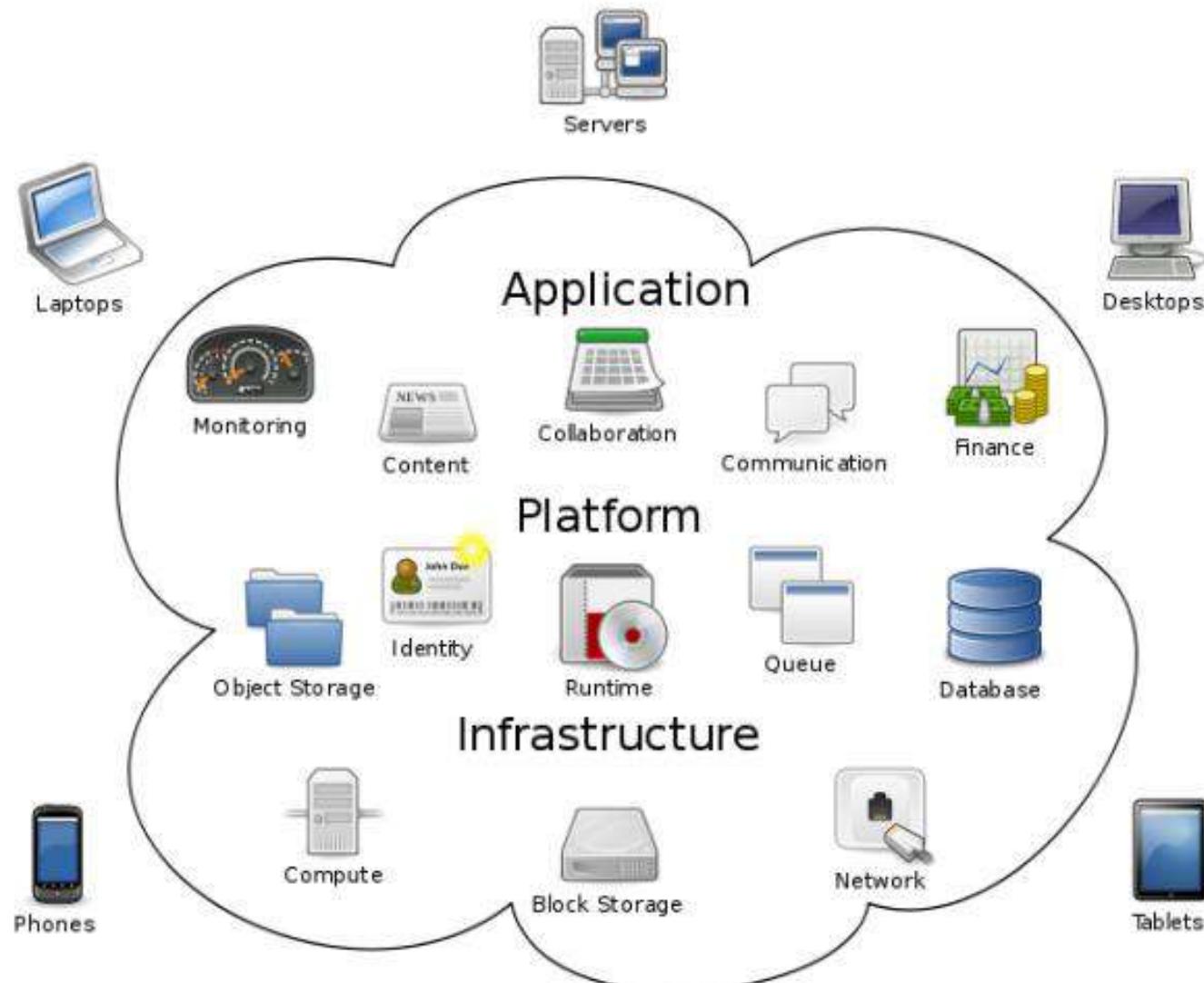


Big Data Technology



Cloud Computing

- IT resources provided as a service
 - Compute, storage, databases, queues
- Clouds leverage economies of scale of commodity hardware
 - Cheap storage, high bandwidth networks & multicore processors
 - Geographically distributed data centers
- Offerings from Microsoft, Amazon, Google, ...



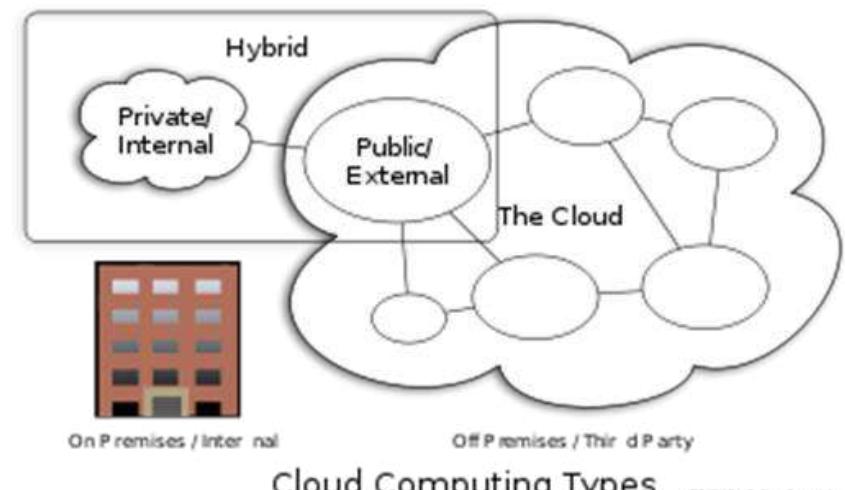
Cloud Computing

Benefits

- Cost & management
 - Economies of scale, “out-sourced” resource management
- Reduced Time to deployment
 - Ease of assembly, works “out of the box”
- Scaling
 - On demand provisioning, co-locate data and compute
- Reliability
 - Massive, redundant, shared resources
- Sustainability
 - Hardware not owned

Types of Cloud Computing

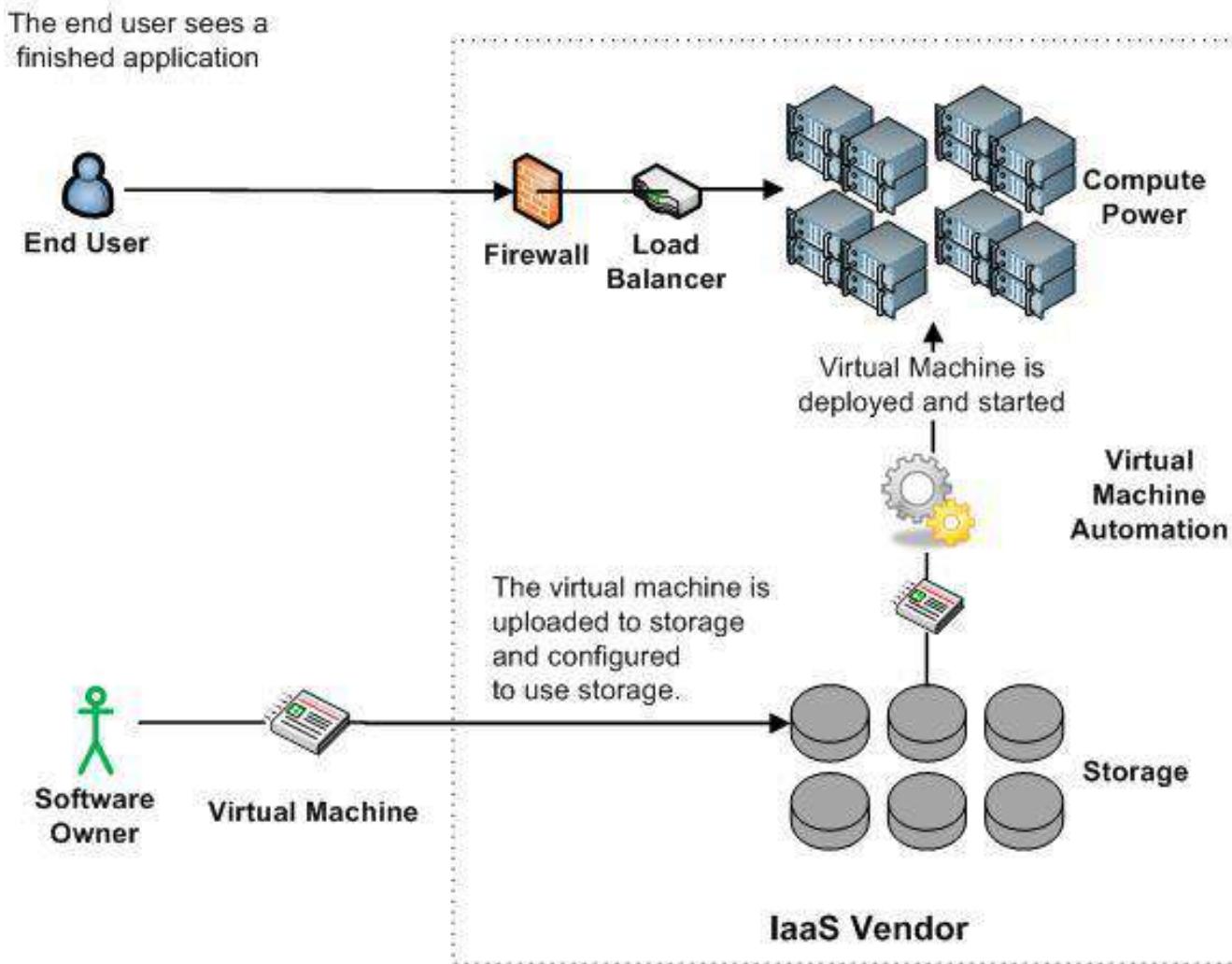
- **Public Cloud:** Computing infrastructure is hosted at the vendor's premises.
- **Private Cloud:** Computing architecture is dedicated to the customer and is not shared with other organisations.
- **Hybrid Cloud:** Organisations host some critical, secure applications in private clouds. The not so critical applications are hosted in the public cloud
 - **Cloud bursting:** the organisation uses its own infrastructure for normal usage, but cloud is used for peak loads.
- **Community Cloud**



Classification of Cloud Computing based on Service Provided

- Infrastructure as a service (IaaS)
 - Offering hardware related services using the principles of cloud computing. These could include storage services (database or disk storage) or virtual servers.
 - [Amazon EC2](#), [Amazon S3](#), [Rackspace Cloud Servers](#) and [Flexiscale](#).
- Platform as a Service (PaaS)
 - Offering a development platform on the cloud.
 - [Google's Application Engine](#), [Microsofts Azure](#), [Salesforce.com's force.com](#) .
- Software as a service (SaaS)
 - Including a complete software offering on the cloud. Users can access a software application hosted by the cloud vendor on pay-per-use basis. This is a well-established sector.
 - Salesforce.coms' offering in the online Customer Relationship Management (CRM) space, Googles [gmail](#) and Microsofts [hotmail](#), [Google docs](#).

Infrastructure as a Service (IaaS)



More Refined Categorization

- Storage-as-a-service
- Database-as-a-service
- Information-as-a-service
- Process-as-a-service
- Application-as-a-service
- Platform-as-a-service
- Integration-as-a-service
- Security-as-a-service
- Management/
Governance-as-a-service
- Testing-as-a-service
- Infrastructure-as-a-service

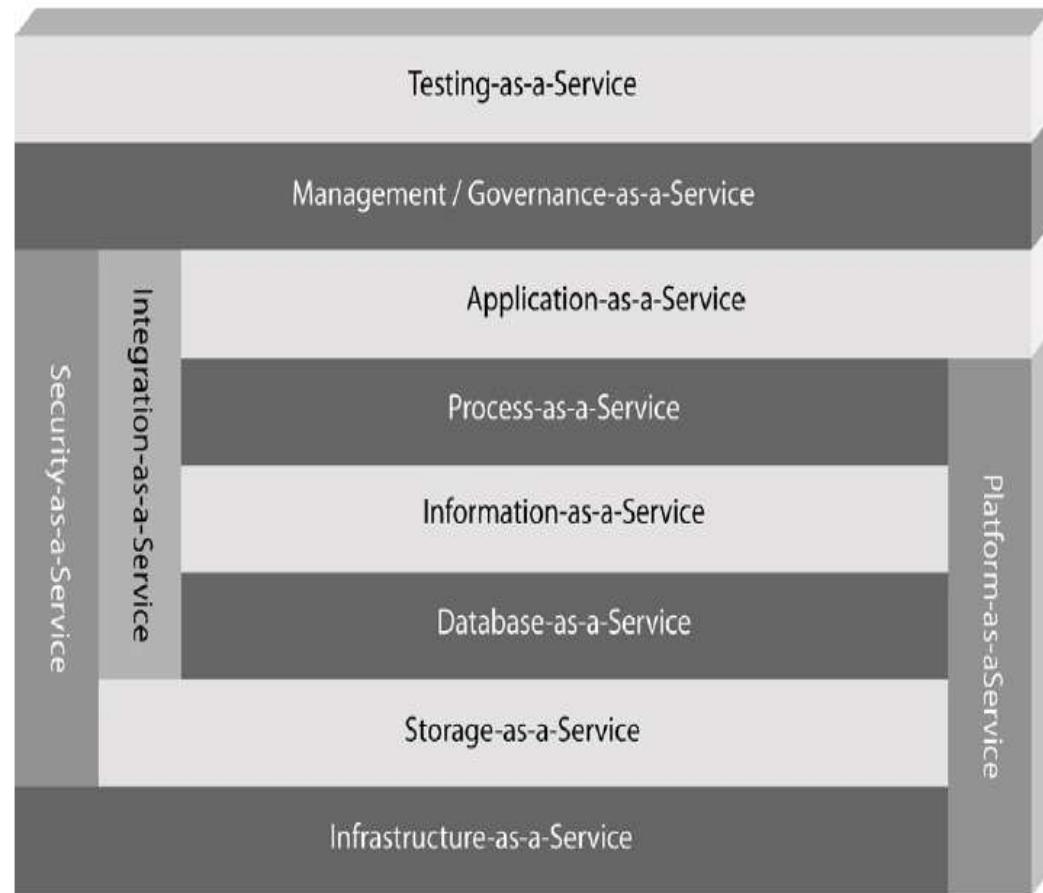


Figure 1: The patterns or categories of cloud computing providers allow you to use a discrete set of services within your architecture.

Key Ingredients in Cloud Computing

- Service-Oriented Architecture (SOA)
- Utility Computing (on demand)
- Virtualization (P2P Network)
- SAAS (Software As A Service)
- PAAS (Platform AS A Service)
- IAAS (Infrastructure AS A Servie)
- Web Services in Cloud

Everything as a Service

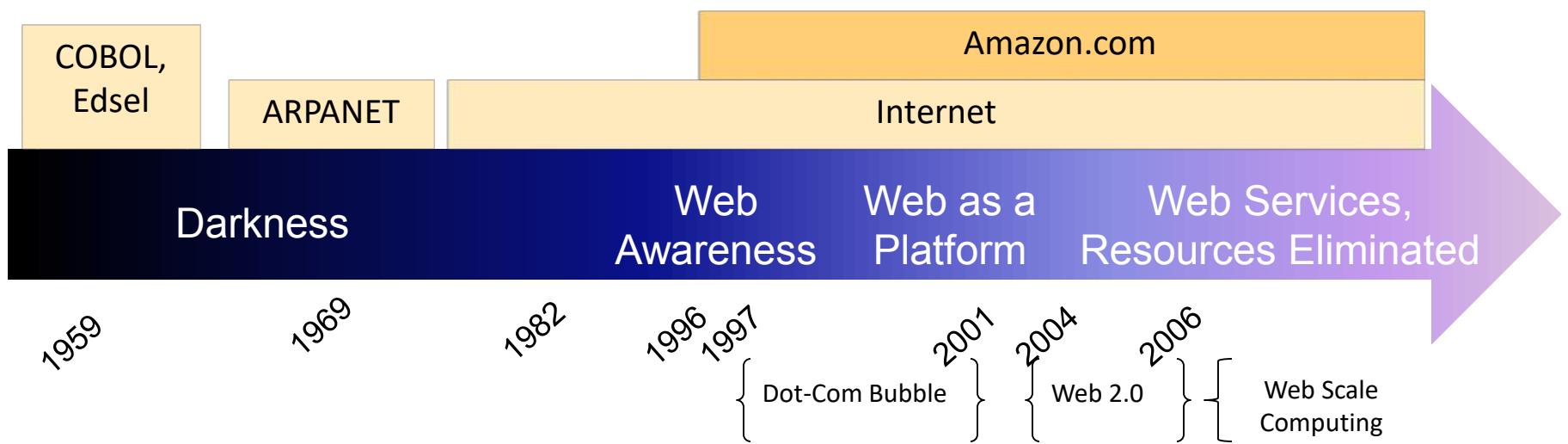
- Utility computing = Infrastructure as a Service (IaaS)
 - Why buy machines when you can rent cycles?
 - Examples: Amazon's EC2, Rackspace
- Platform as a Service (PaaS)
 - Give me nice API and take care of the maintenance, upgrades, ...
 - Example: Google App Engine
- Software as a Service (SaaS)
 - Just run it for me!
 - Example: Gmail, Salesforce

Cloud versus cloud

- Amazon Elastic Compute Cloud
- Google App Engine
- Microsoft Azure
- GoGrid
- AppNexus

The Obligatory Timeline Slide

(Mike Culver @ AWS)



AWS

- Elastic Compute Cloud – EC2 (IaaS)
- Simple Storage Service – S3 (IaaS)
- Elastic Block Storage – EBS (IaaS)
- SimpleDB (SDB) (PaaS)
- Simple Queue Service – SQS (PaaS)
- CloudFront (S3 based Content Delivery Network – PaaS)
- Consistent AWS Web Services API

What does Azure platform offer to developers?

Your Applications



Service Bus

Workflow

Access Control

...



Database

Analytics

Reporting

...



Live Services

Identity

Contacts

Devices

...

...

Compute

Storage

Manage

...



Windows® Azure™