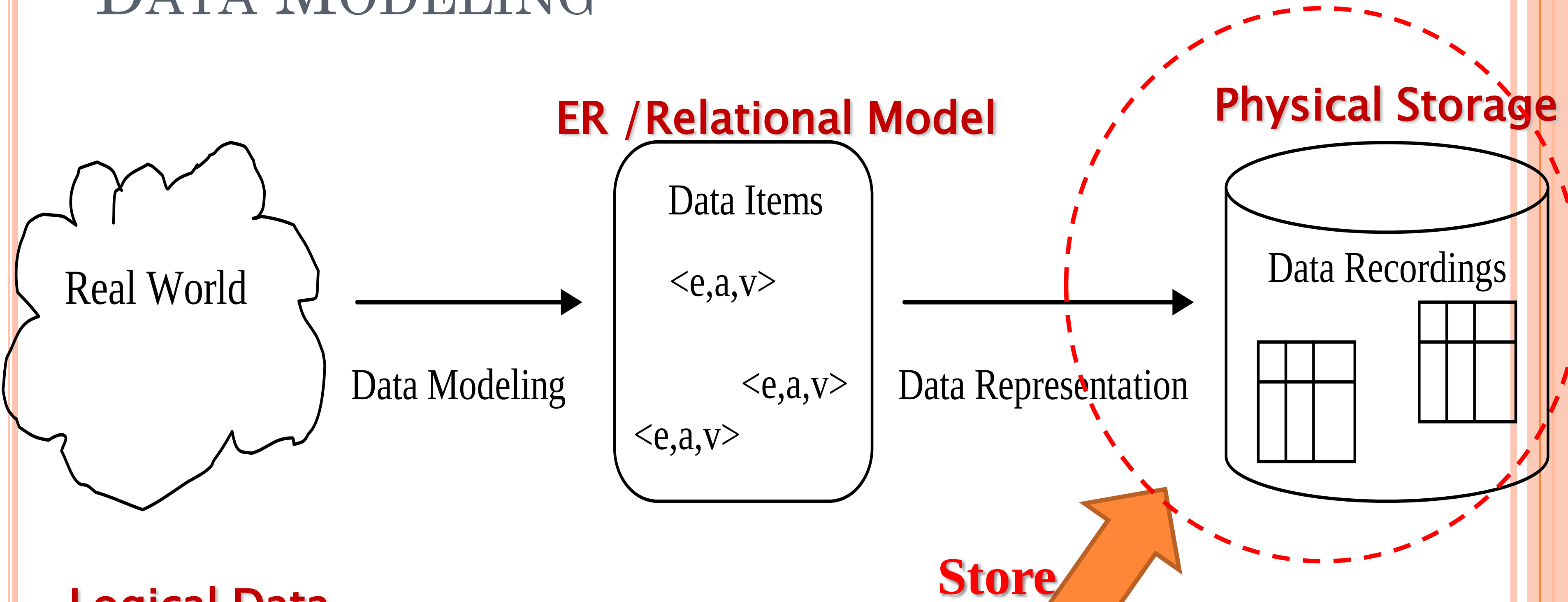
Physical Database Design

Outline

- Overview of Physical Database Design
- Inputs of Physical Database Design
- File Structures
- Query Optimization
- Index Selection
- Additional Choices in Physical Database Design

DATA MODELING



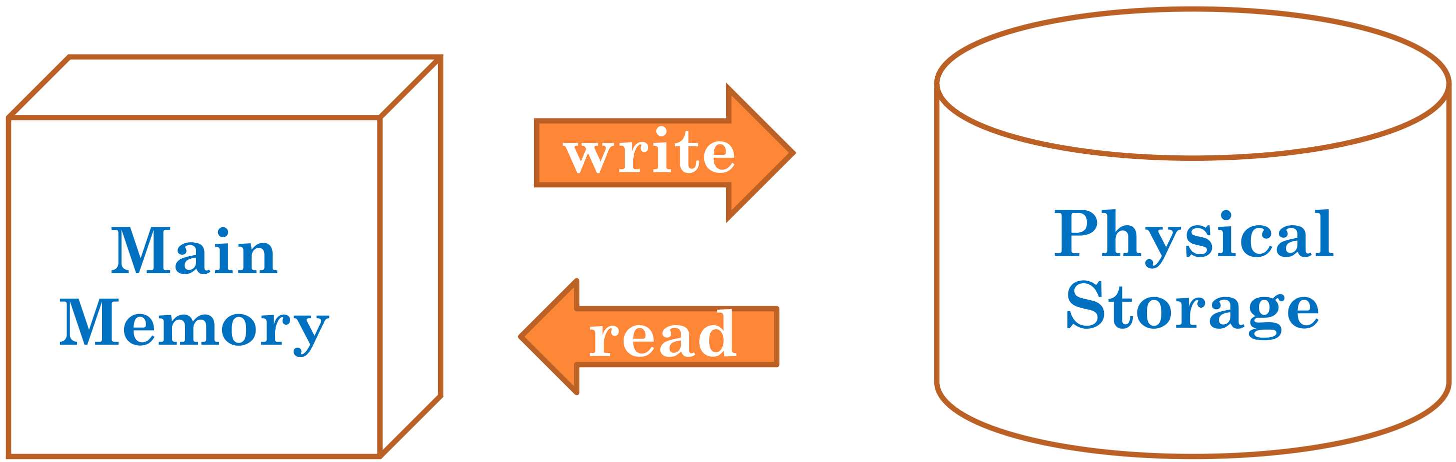
Logical Data

StdSSN	StdLastName	StdMajor	StdClass	StdGPA
123-45-6789	WELLS	IS	FR	3.00
124-56-7890	NORBERT	FIN	JR	2.70
234-56-7890	KENDALL	ACCT	JR	3.50

Overview of Physical Database Design

- Importance of the *process and environment* of physical database design
  - Process: inputs, outputs, objectives
  - Environment: file structures and query optimization
- Physical Database Design is characterized as a series of *decision-making processes.*
- *Decisions* involve the storage level of a database: file structure and optimization choices.

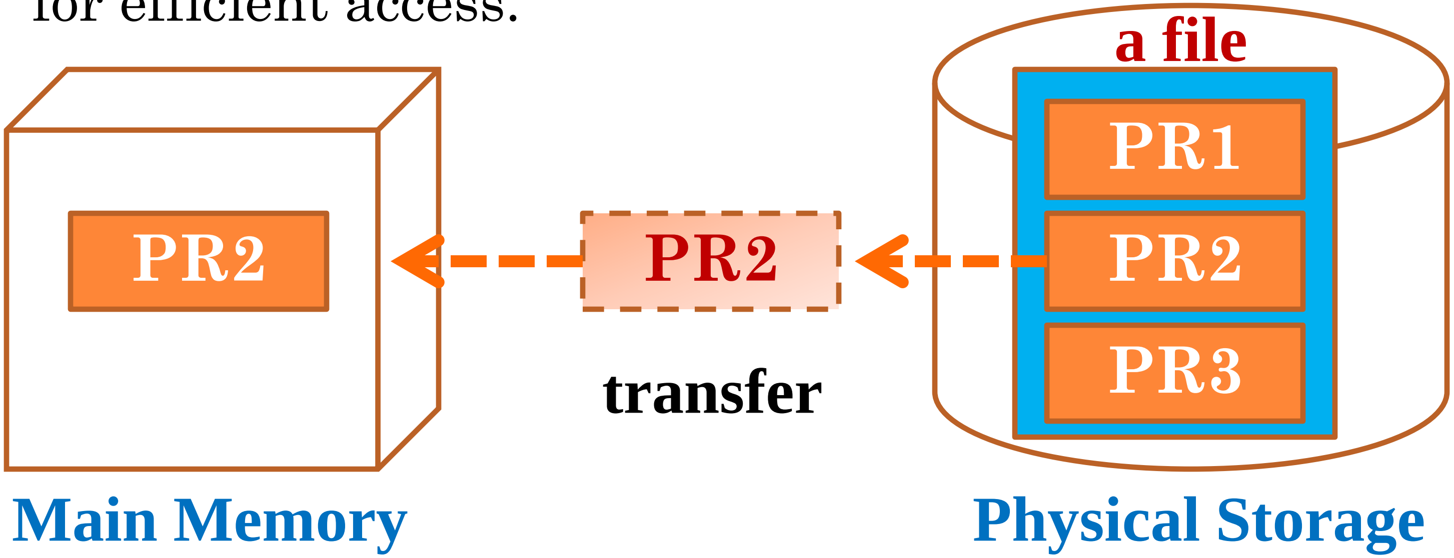
Storage Level of Databases



- The storage level is closest to the hardware and operating system.
- CPU can process data that are stored in main memory.

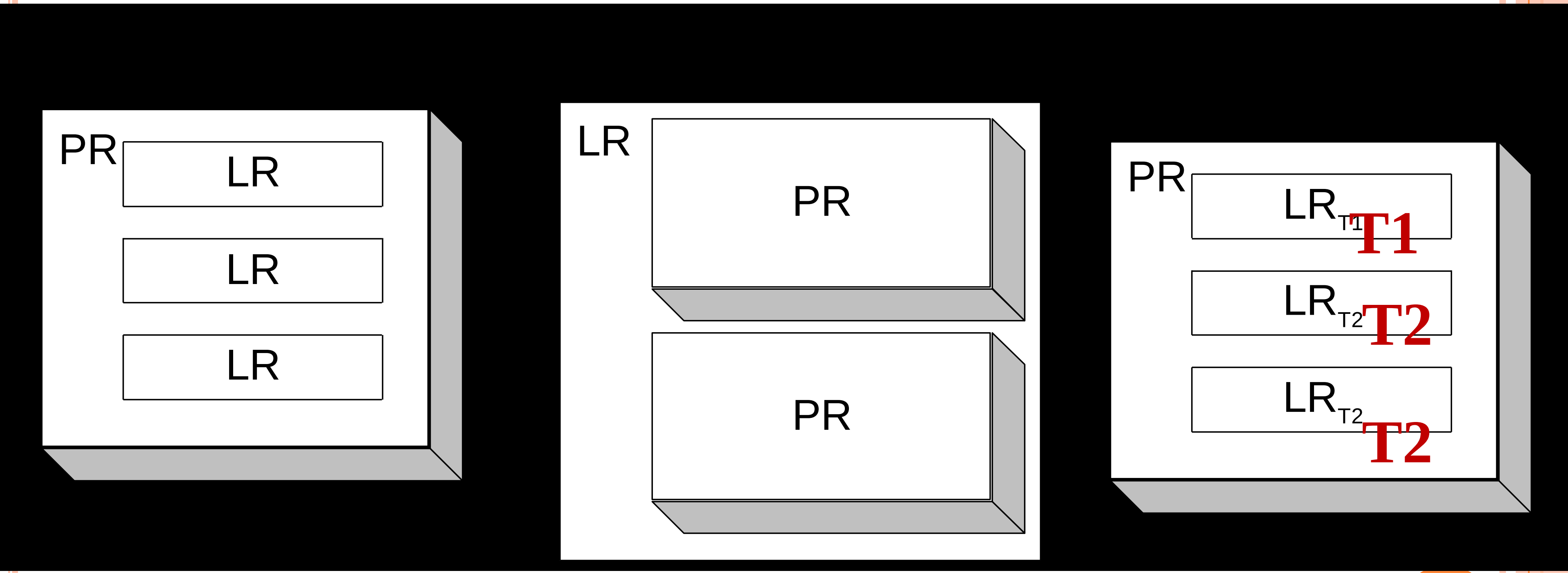
Storage Level of Databases

- At the logical level,
  - A database consists of many tables
  - A table consists of many logical records
- At the storage level,
  - A table is a file
  - A file is a collection of **physical records** organized for efficient access.



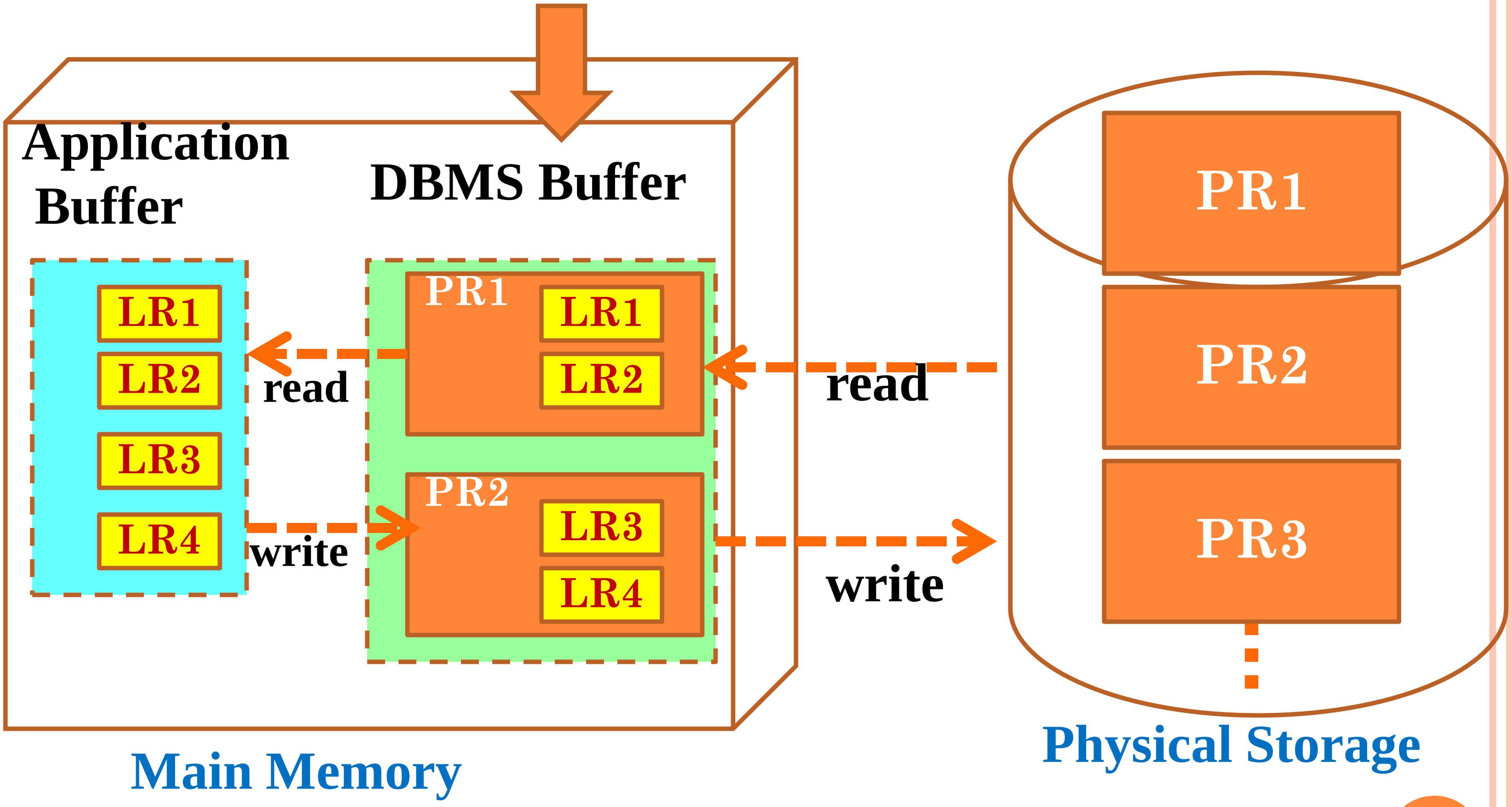
Relationships between Logical Records (LR) and Physical Records (PR)

(a) **(b)** **(c)**



A logical record (LR) is a row in a table

Storage Level of Databases



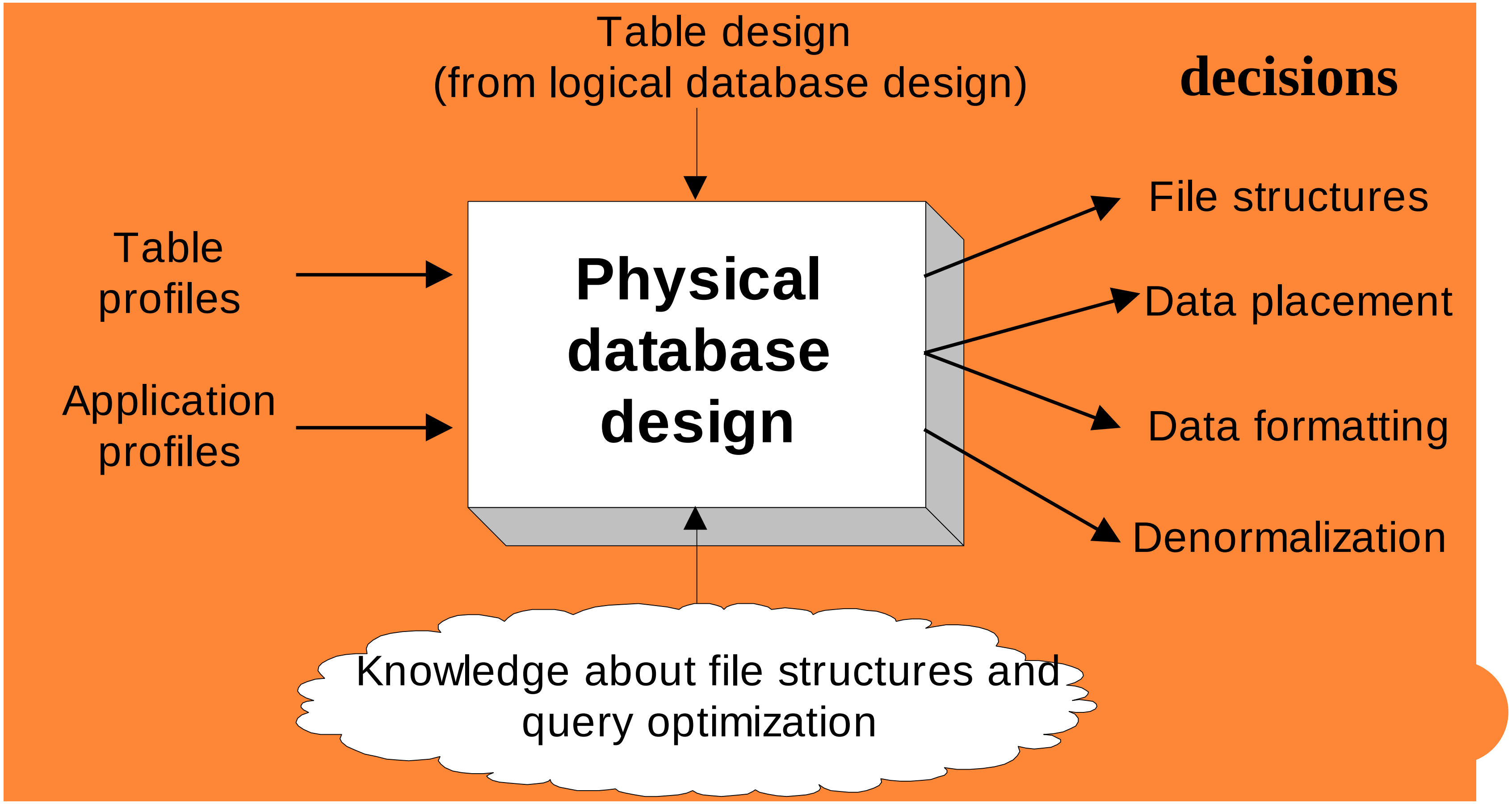
Objectives of of physical database design

- Minimize response time to access and change a database.
- Minimizing computing resources is a substitute measure for response time.
- Database resources
  - Physical record transfers
  - CPU operations
  - Communication network usage (distributed processing)

Constraints

- Main memory and disk space are considered as constraints rather than resources to minimize.
- Minimizing main memory and disk space can lead to high response times (เร็วขึ้น).
- Thus, reducing the number of physical record accesses can improve response time.
- CPU usage also can be a factor in some database applications.

Inputs, Outputs, and Environment



Inputs of Physical Database Design

- Physical database design requires inputs specified in sufficient detail.
- Table profiles and application profiles are important and sometimes difficult-to-define inputs.

Table Profile

- A table profile summarizes a table as a whole, the columns within a table, and the relationships between tables.

Typical Components of a Table Profile

Component	**Statistics**
Table	Number of rows and physical records
Column	Number of unique values, distribution of values
Relationship	Distribution of the number of related rows

Application profiles

- Application profiles summarize the queries, forms, and reports that access a database.

Typical Components of an Application Profile

Application Type	**Statistics**
Query	Frequency; distribution of parameter values
Form	Frequency of insert, update, delete, and retrieval operations to the main form and the subform
Report	Frequency; distribution of parameter values

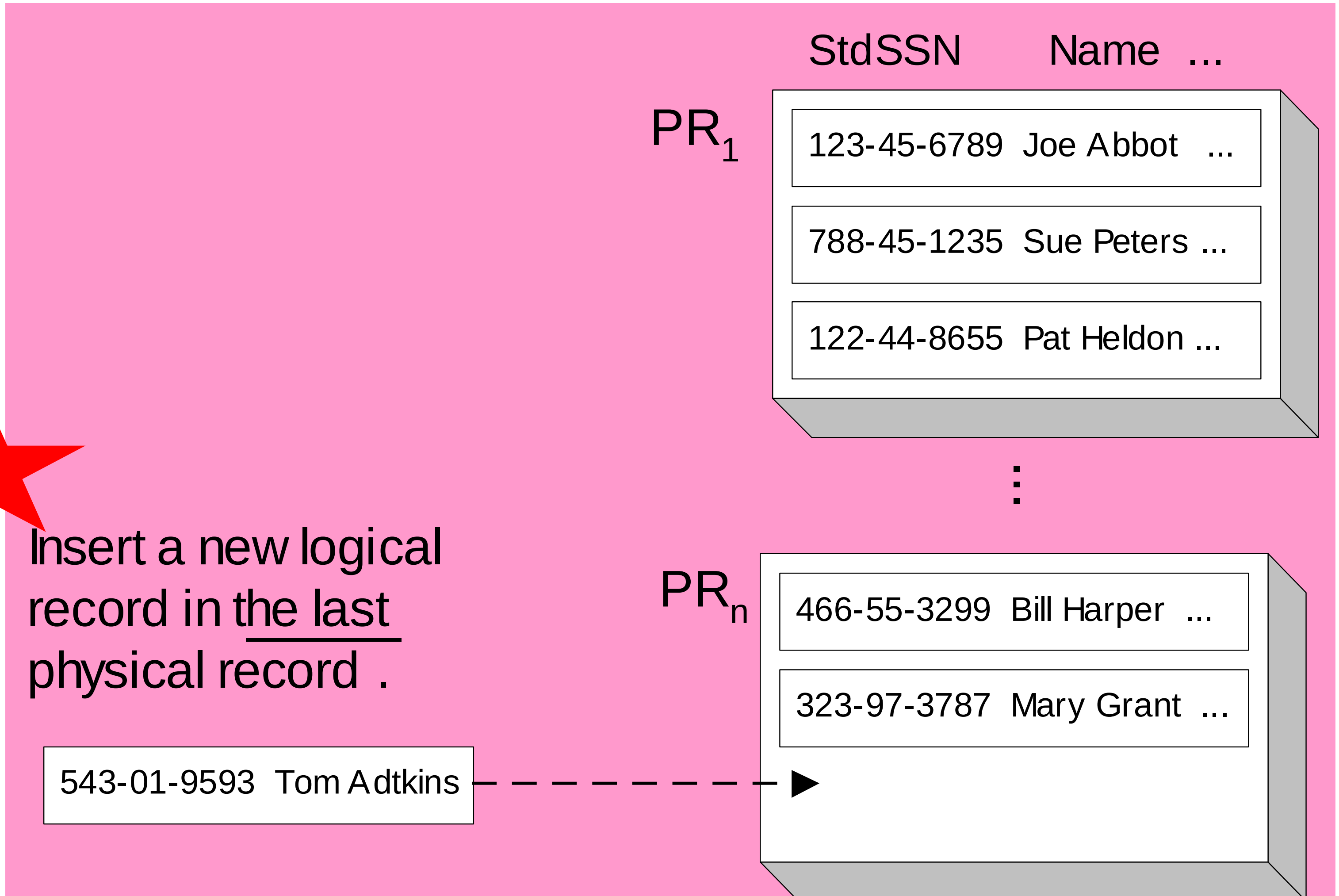
File structures

- Selecting among alternative file structures is one of the most important choices in physical database design.
- In order to choose intelligently, you must understand characteristics of available file structures.

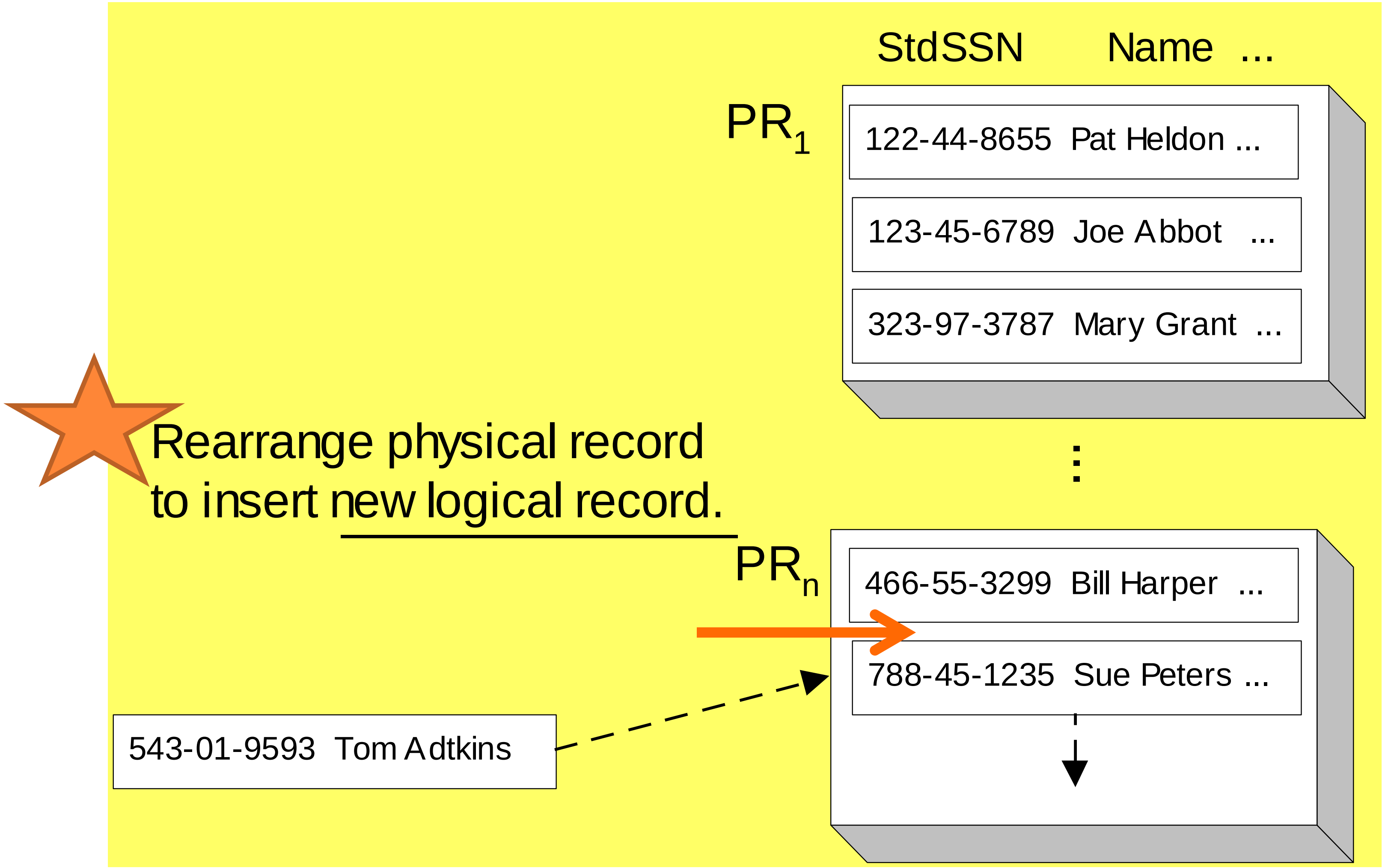
Sequential Files

- Simplest kind of file structure
- Unordered: insertion order
- Ordered: key order
- Simple to maintain
- Provide good performance for processing large numbers of records

Unordered Sequential File

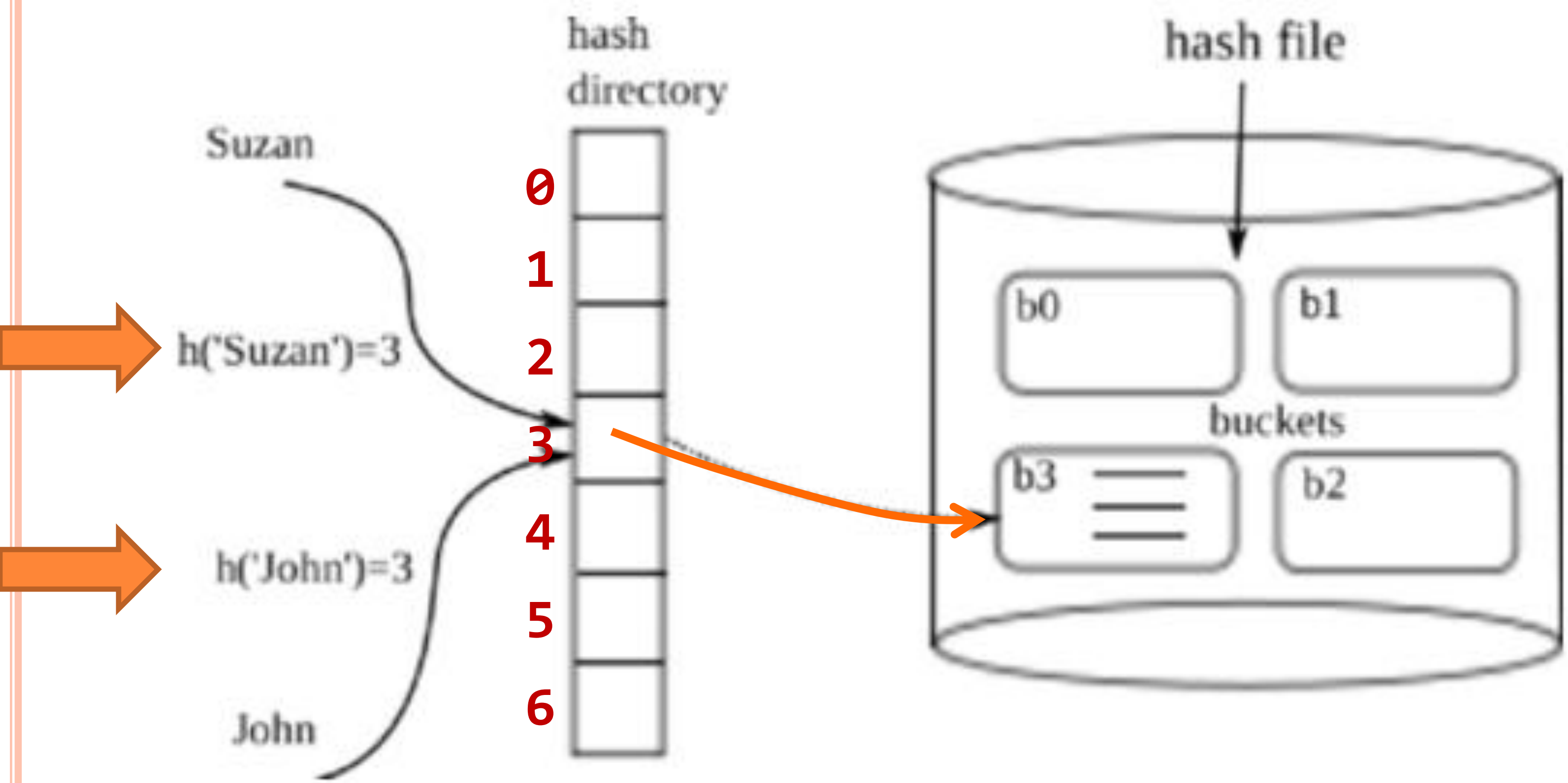


ORDERED SEQUENTIAL FILE



Hash Files

HashFn(key) = Address



Hash Files

- Support fast access unique key value
- Converts a key value into a physical record address
- Mod function: typical hash function
  - Divisor: large prime number close to the file capacity
  - Physical record number: hash function plus the starting physical record number

122448655 mod 97 = 26,

26+starting PR# = 176 = location of data
(150)

(Physical record number)

Hash File after Insertions

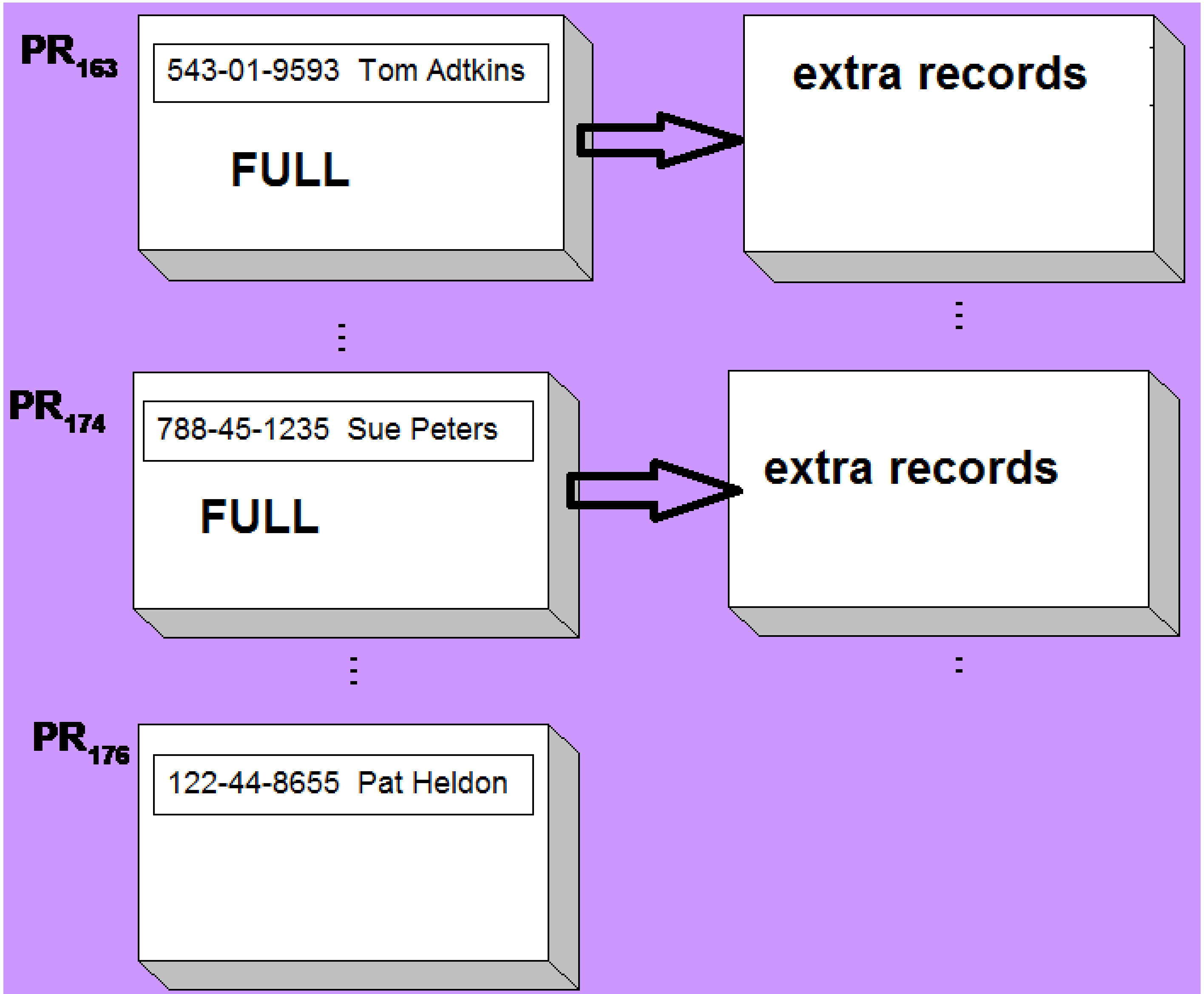
Example: Hash Function Calculations for StdSSN Key

122448655 mod 97 = 26,

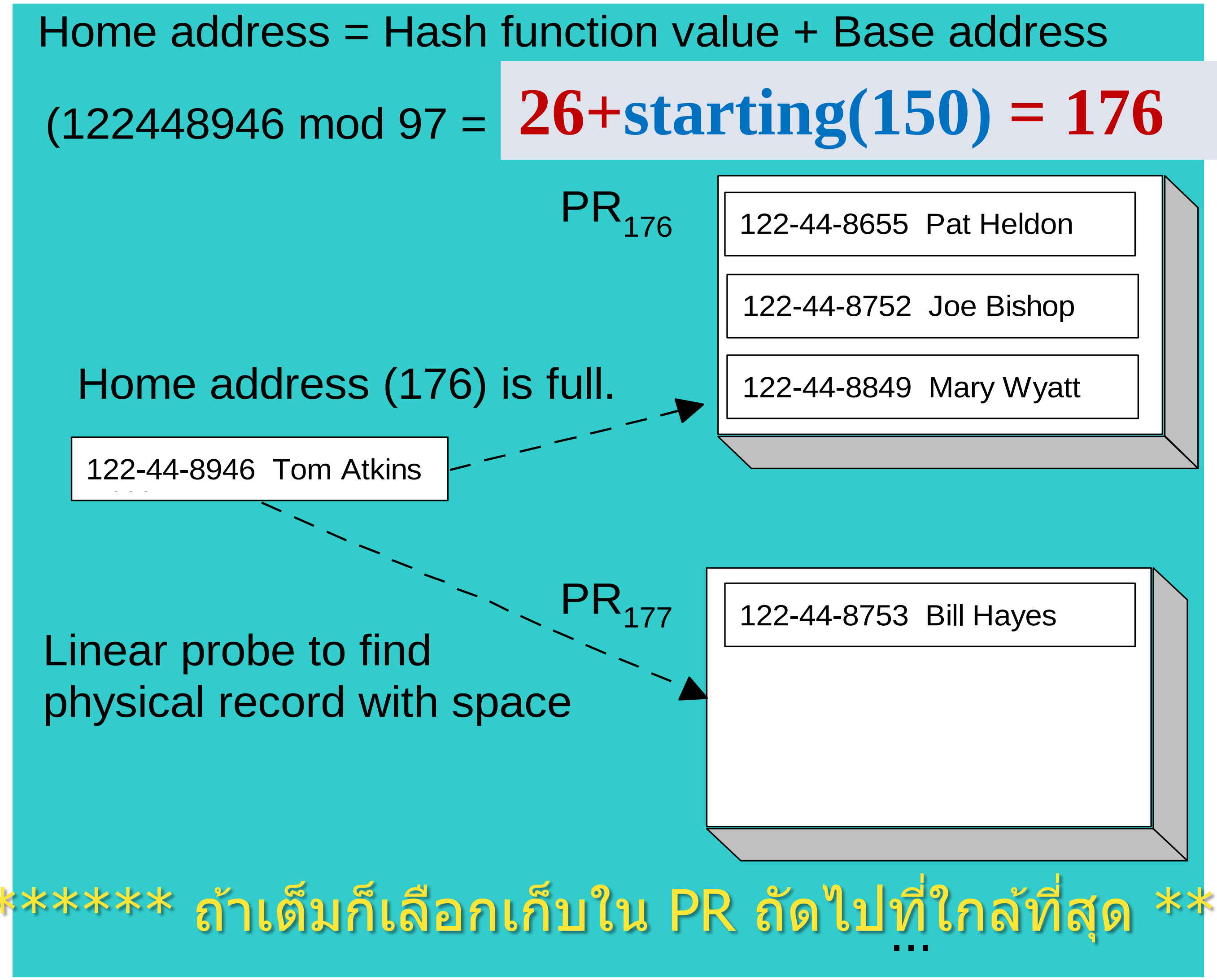
26+starting PR#(150) = 176

StdSSN	StdSSN Mod 97	PR Number
122448655	**26**	**176**
123456789	39	189
323973787	92	242
466553299	80	230
788451235	24	174
543019593	13	163

Handling Collisions

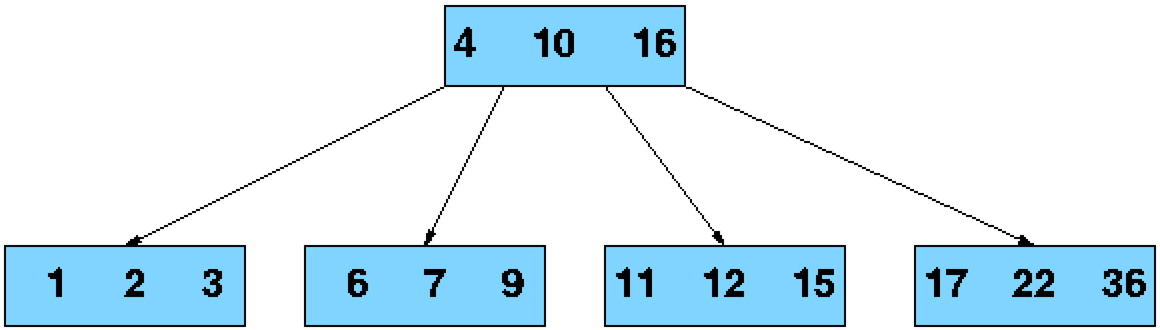


Linear Probe Collision Handling During an Insert Operation



Btrees

root



Leaf nodes

Leaves

root

Height = 1
Order = 4

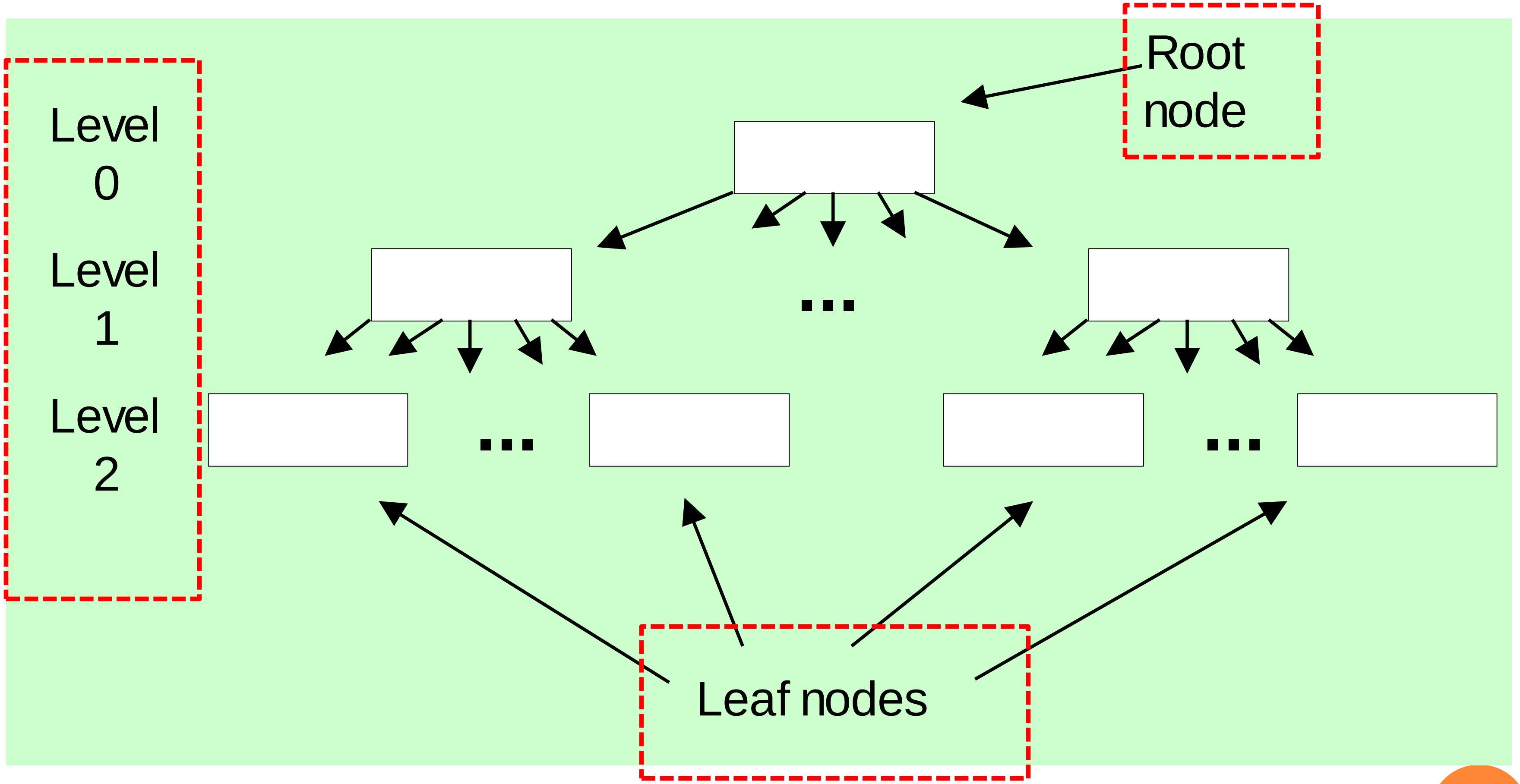
Btree characteristics:

- Balanced
  - Max access= height of Btree
- Bushy: multi-way tree
  - Ideal Btree : **wide** (bushy) but **short** (few levels).
- Block-oriented
  - each node is a physical record
- Dynamic
  - Btree changes as logical records are inserted and deleted

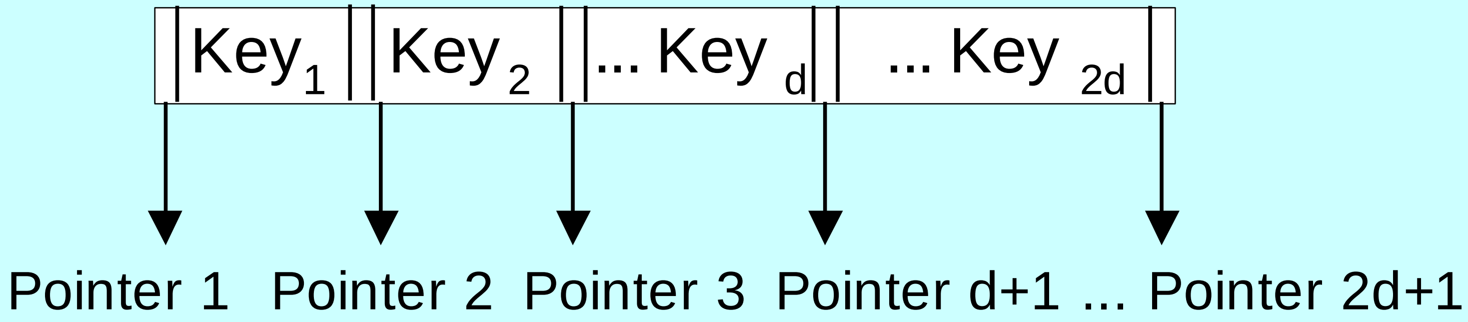
WHY BTREES ?

- Sequential files →
  - well on sequential search
  - poorly on key search
- Hash files →
  - well on key search
  - poorly on sequential search,
- Btree is a compromise and widely used file structure.
  - good performance on both sequential search and key search.

Structure of a Btree of Height 3



BTREE NODE CONTAINING KEYS AND POINTERS

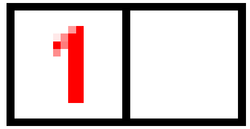


Each non root node contains at least half capacity
(*d* keys and *d*+1 pointers).

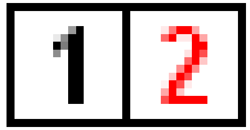
Each non root node contains at most full capacity
(2*d* keys and 2 *d*+1 pointers).

maximum of two keys

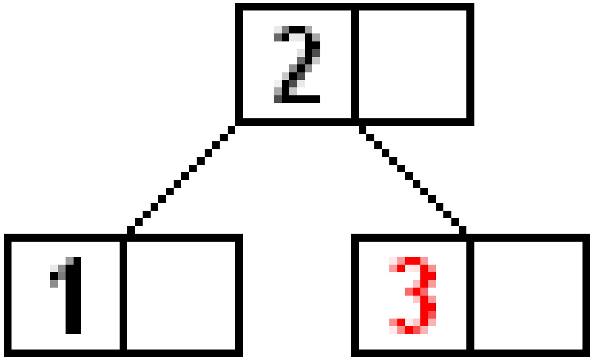
insert 1.



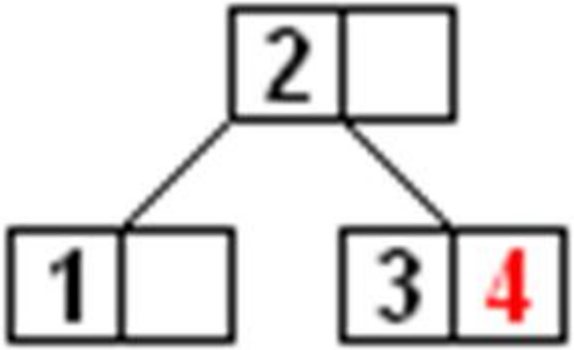
insert 2.



insert 3.

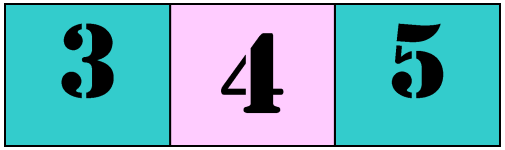


insert 4.

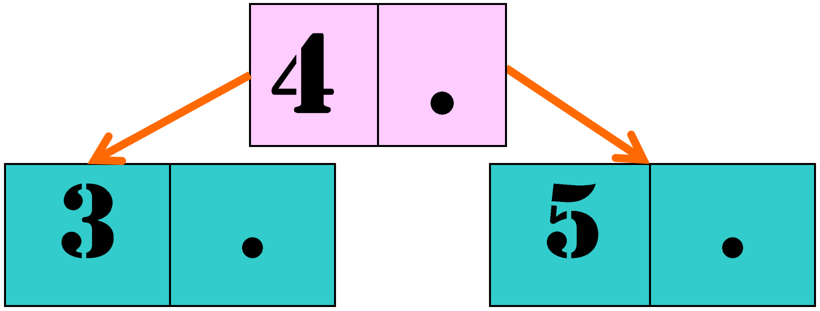


5 ?full

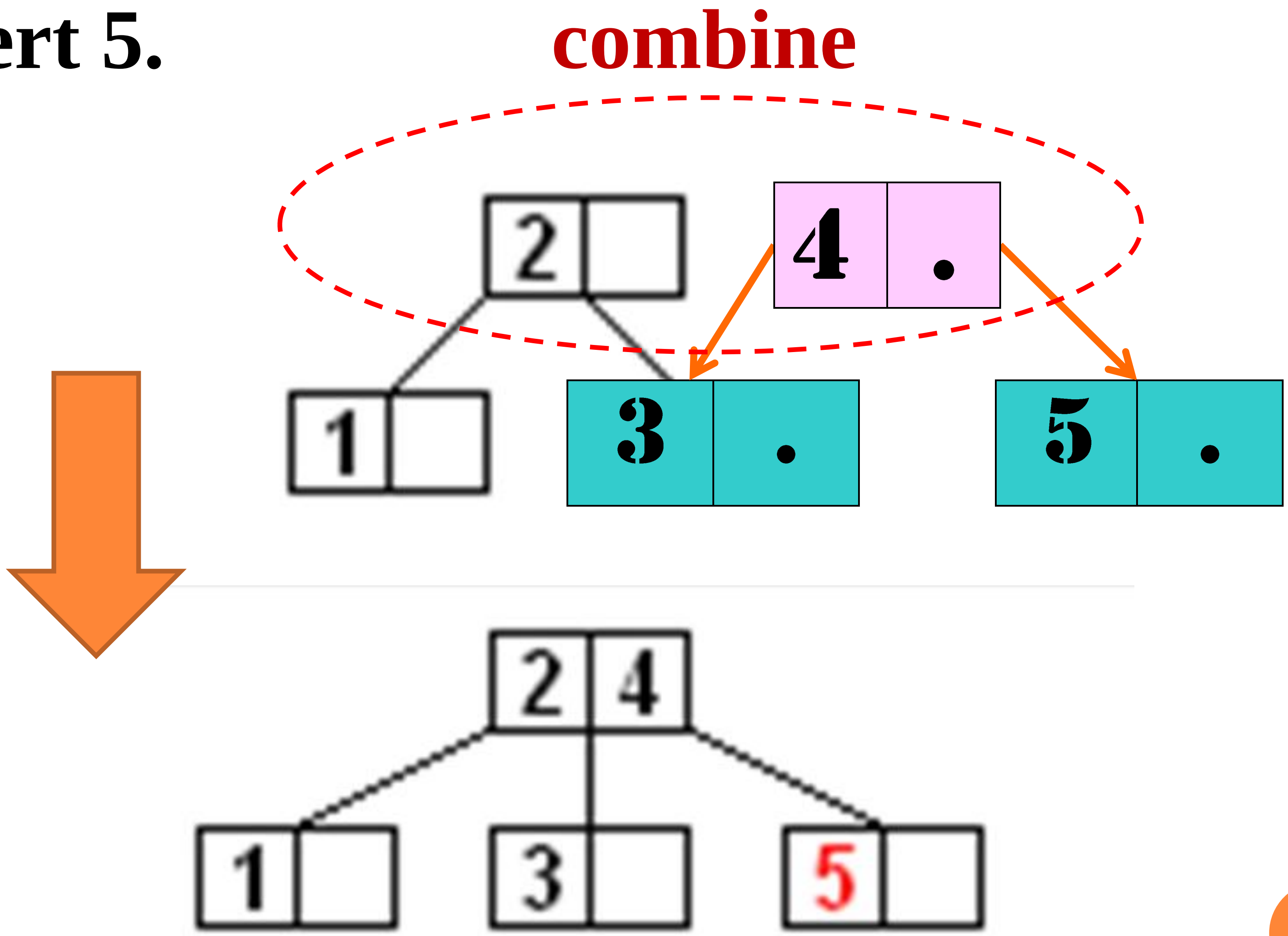
insert 5 ?



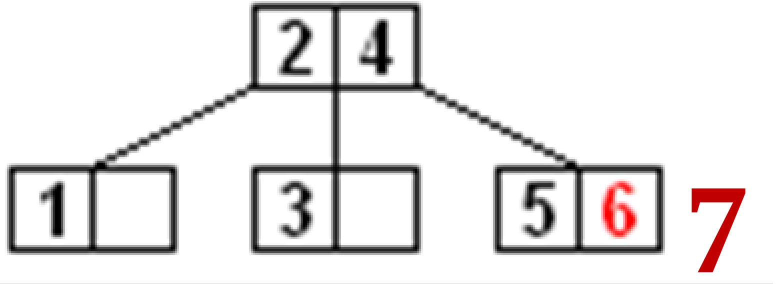
Split



insert 5.



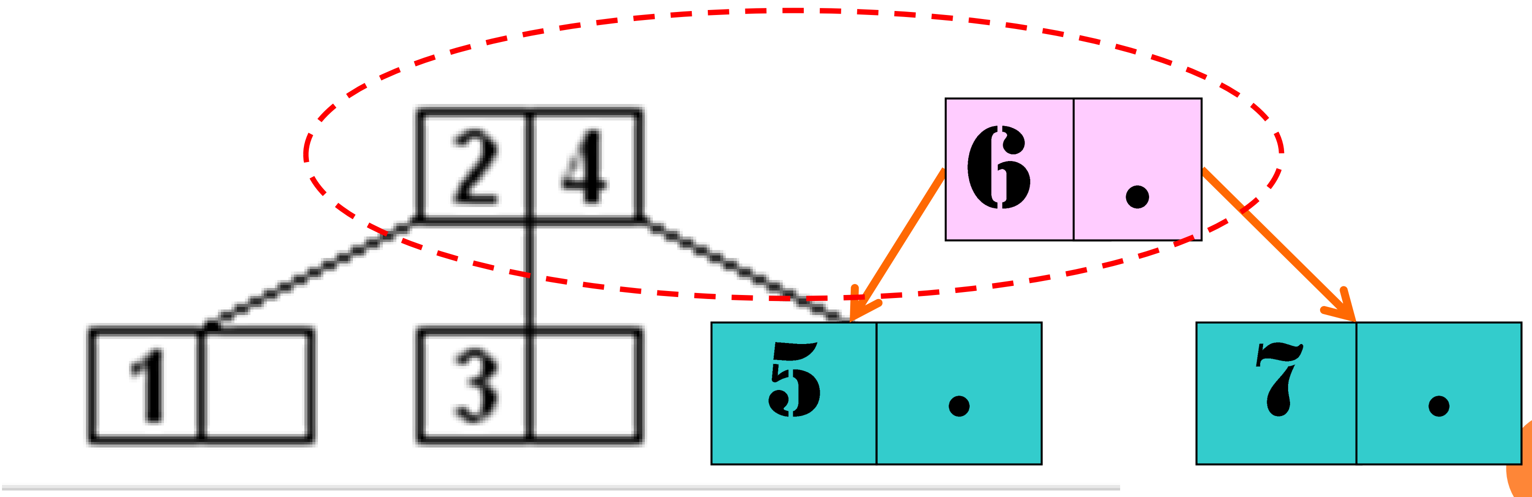
insert 6.



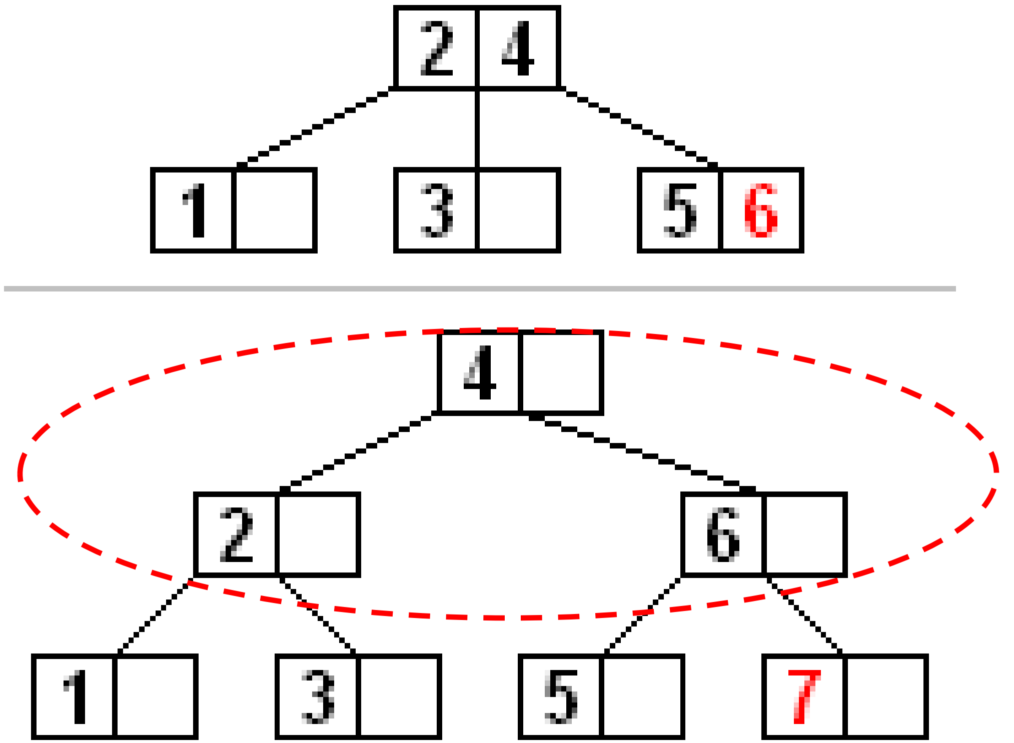
7 ?full

insert 7.

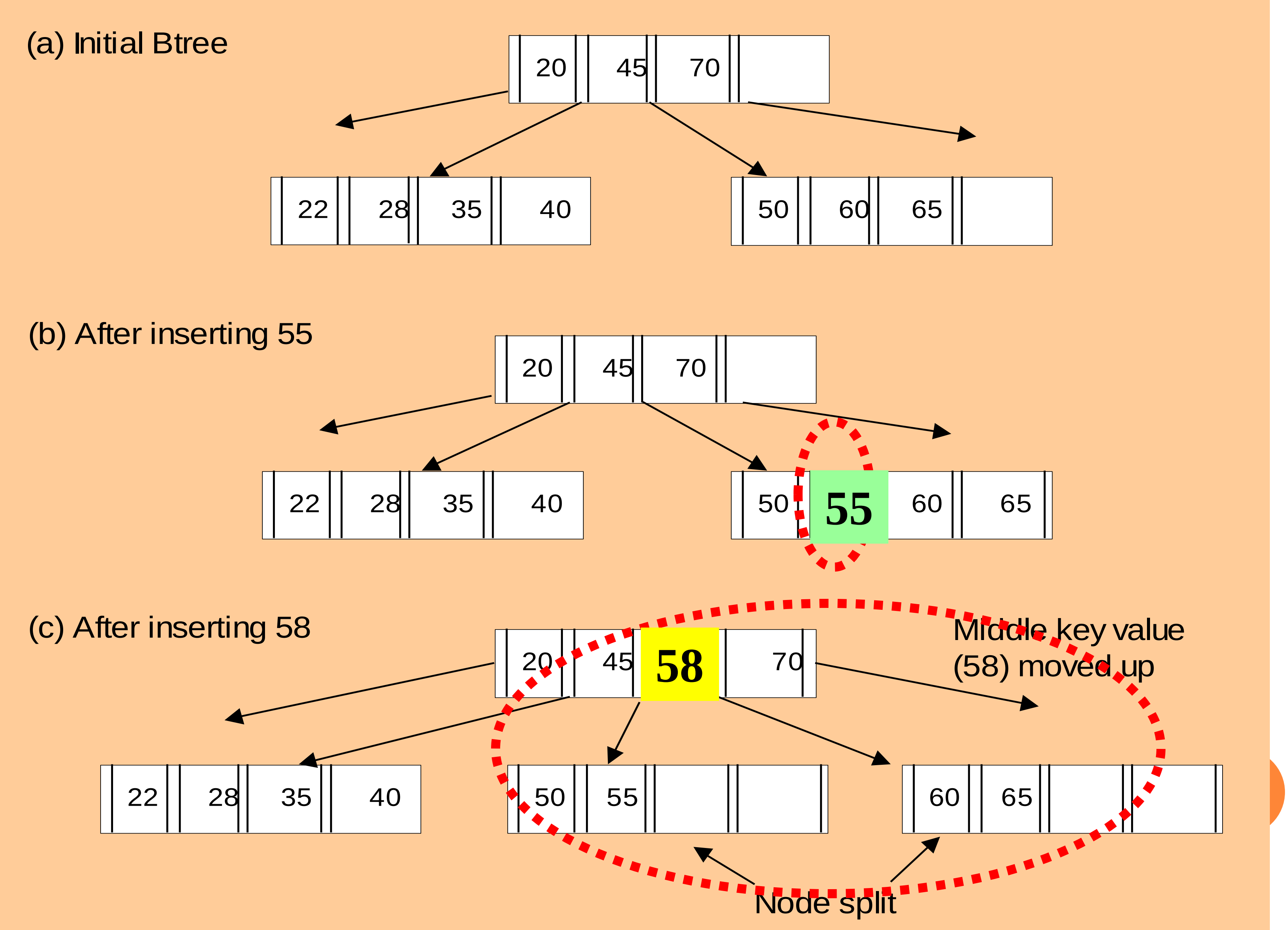
combine



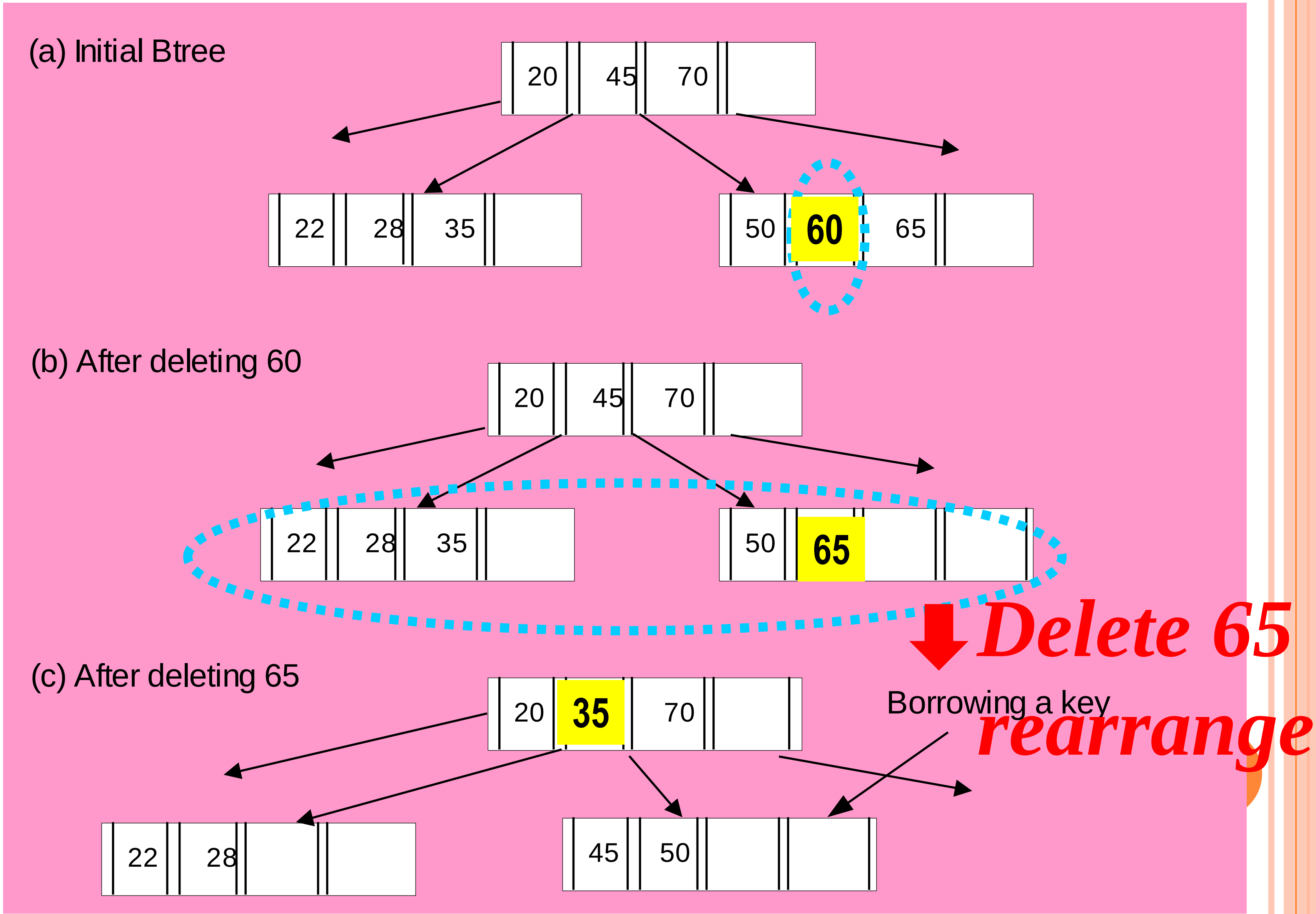
After insert 7.

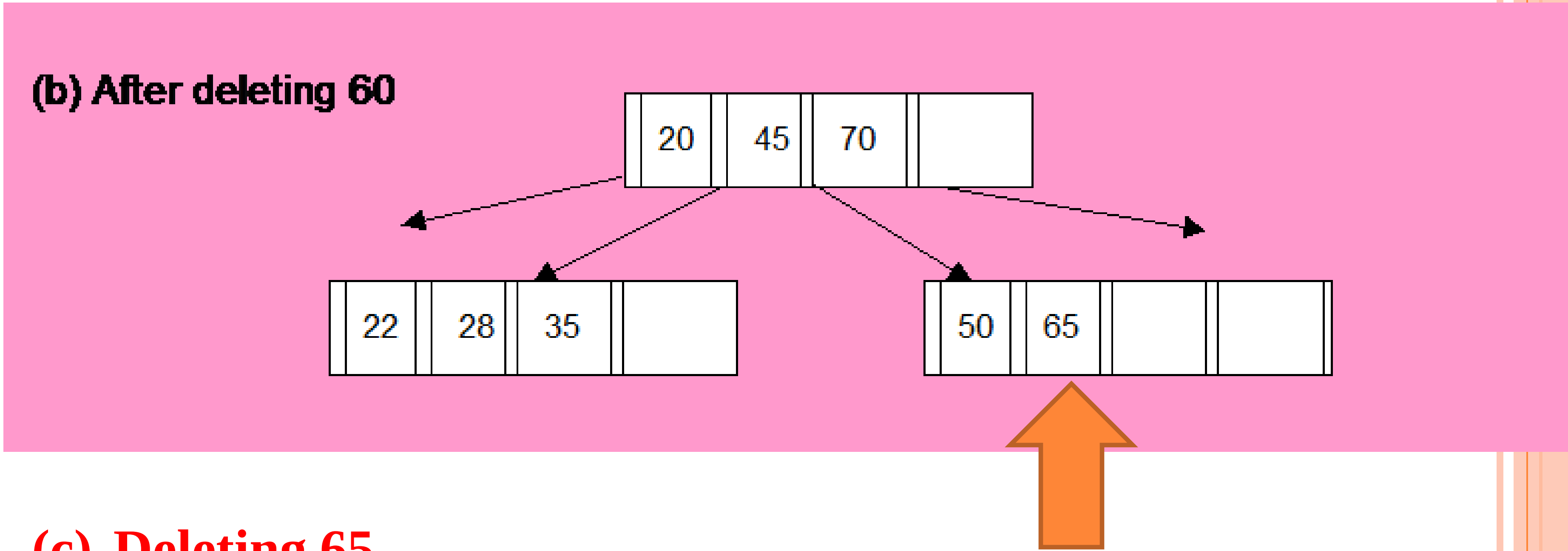


Btree Insertion Examples



BTREE DELETION EXAMPLES

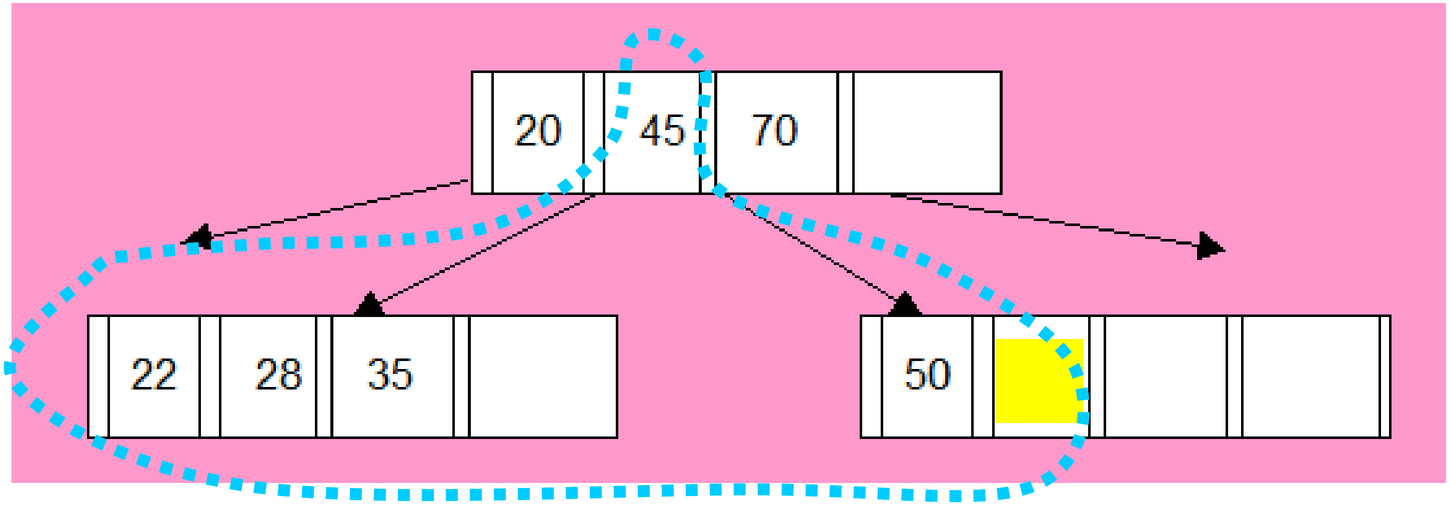




(c) Deleting 65

needs to restructure the tree because it will <u>not be half full</u>

Deleting 65



1. Combine nodes

22, 28, 35, 45, 50

2. Split nodes (แบ่งครึ่ง)

22, 28

35, 45, 50

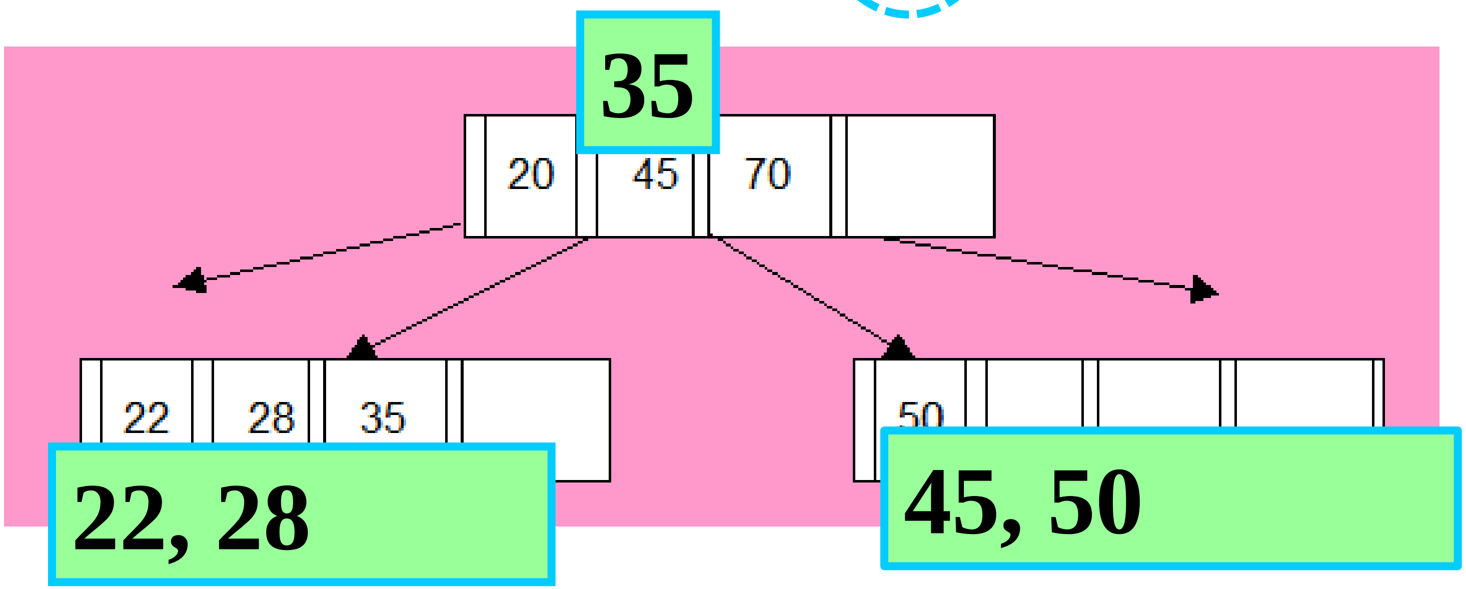
Deleting 65

3. Insert 35 in upper node

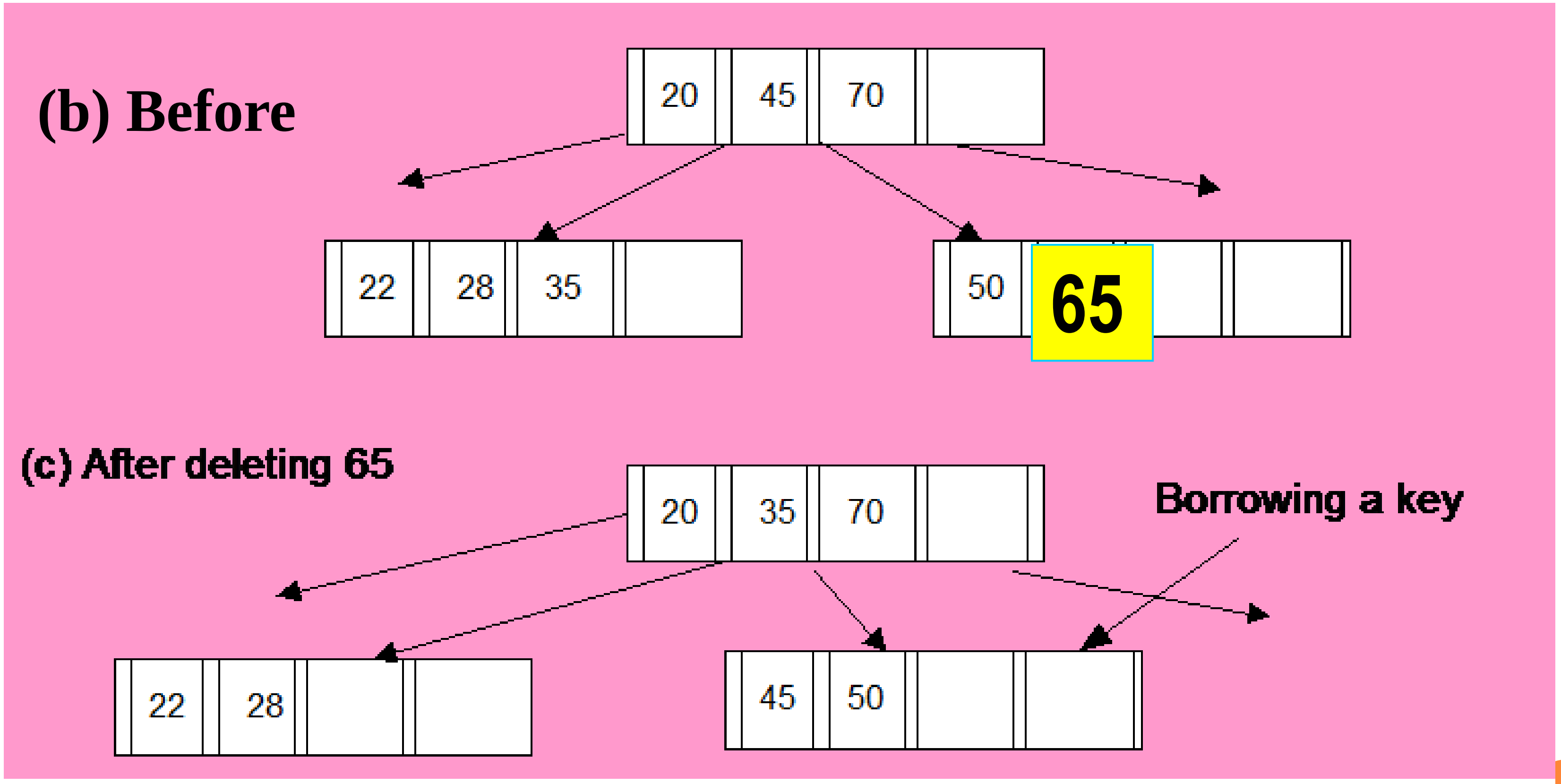
35

22, 28

~~35~~, 45, 50



After delete 65



Cost of Operations

- The **height of Btree** dominates the number of physical record accesses operation.
- **Logarithmic search cost**
  - Upper bound of height: log function'
  - Log base (t): minimum number of keys in a node
- The cost to insert a key =

  [the cost to locate the nearest key] +

  [the cost to change nodes].***or add a new level***

If $n \geq 1$, then for any n-key B-tree of height h and mimimum degree $t \geq 2$,

$$height = h \leq \log_t[(n+1)/2]$$

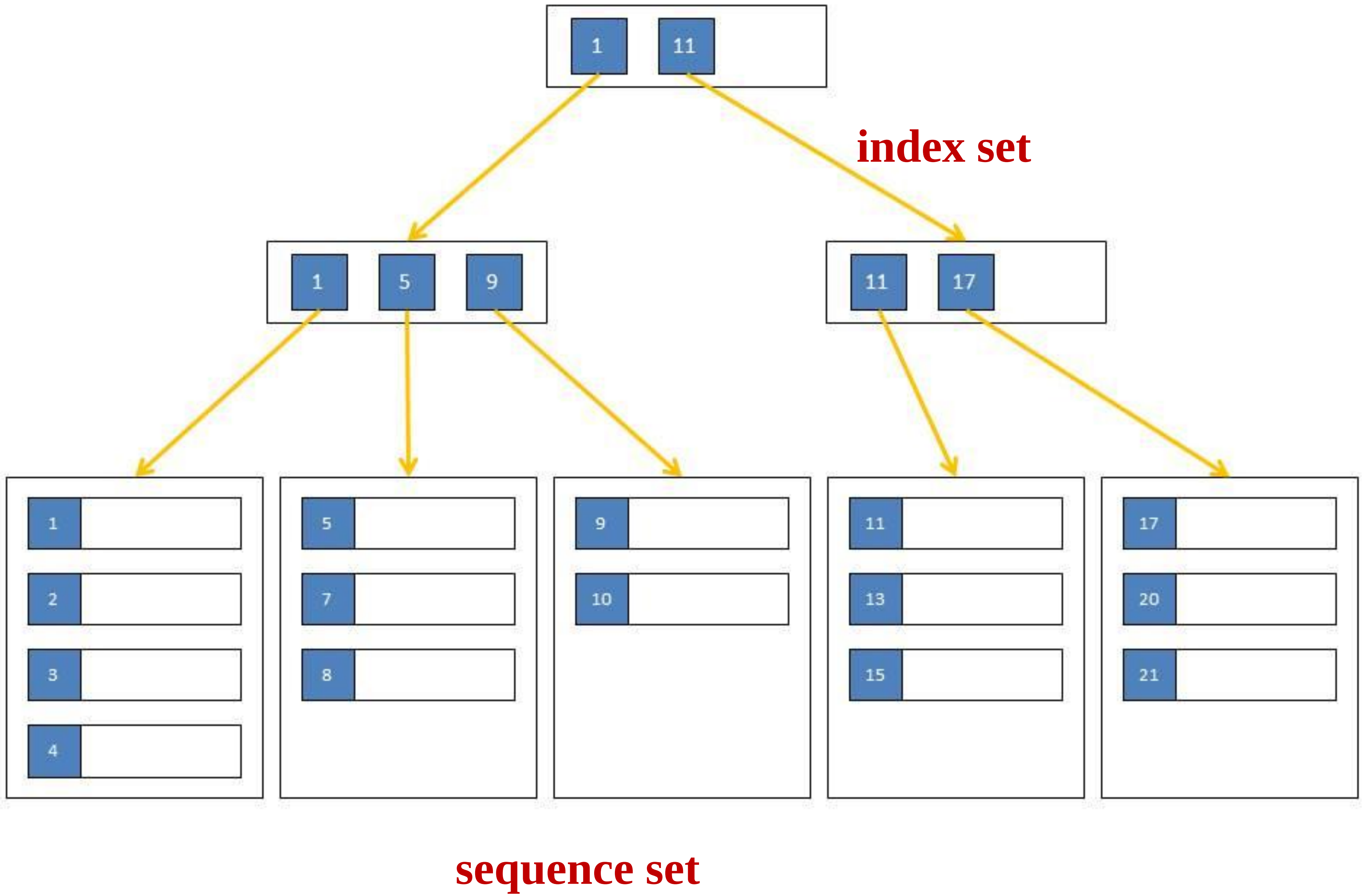
QUIZ

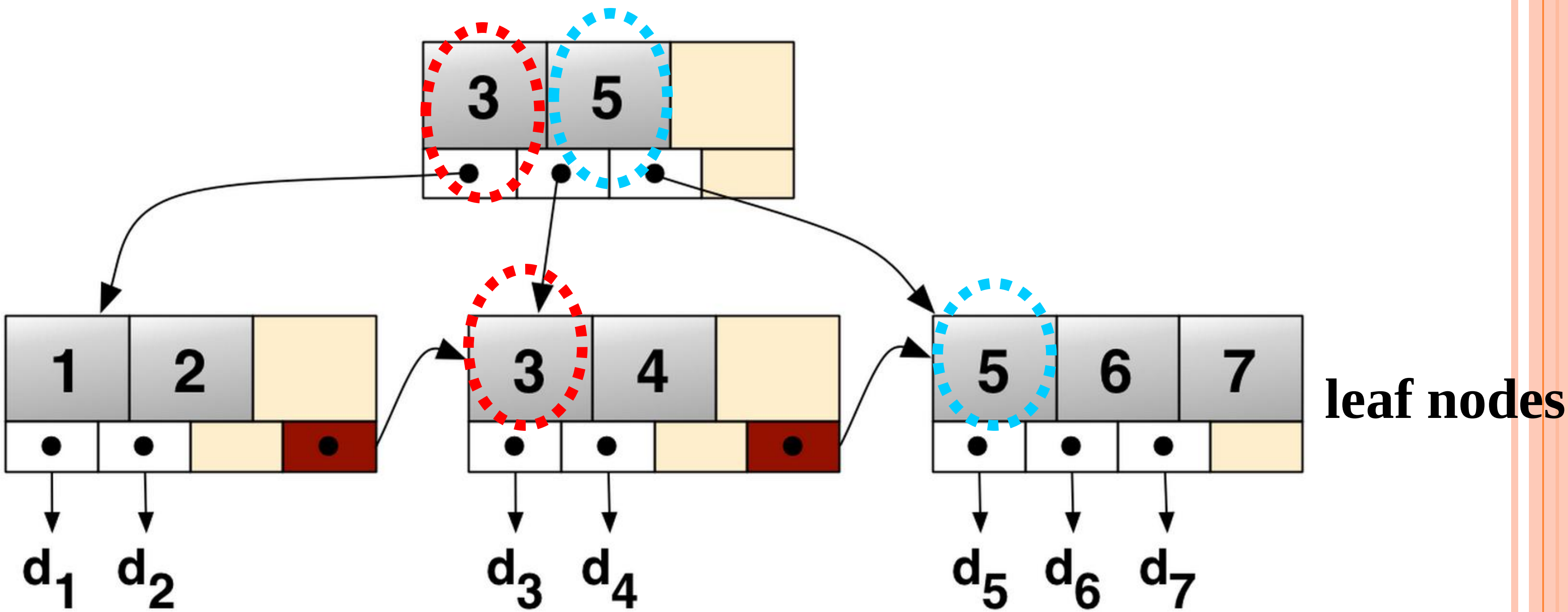
- Show the B-tree that results when inserting the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in an initially empty B-tree of order 3 (max pointers = 3).

B+Tree

- Provides improved performance on *sequential and range searches.*
- In a B+tree, all keys are *redundantly* stored in the leaf nodes.
- To ensure that physical records are not replaced, the B+tree variation is usually implemented.

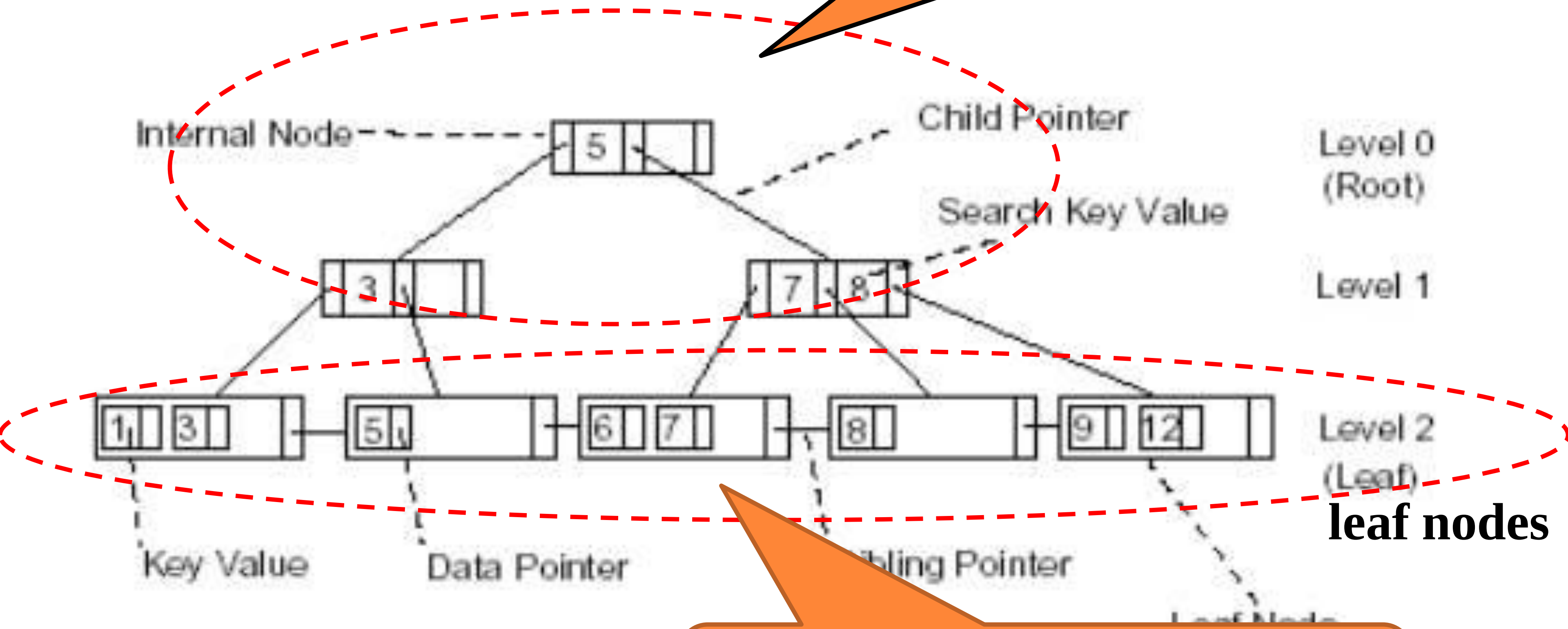
B+Tree


1
11
index set
1
5
9
11
17
1
2
3
4
5
7
8
9
10
11
13
15
17
20
21
sequence set



d1 ,d2, ..: Data Pointers point to physical records

Index keys:
support **key** search



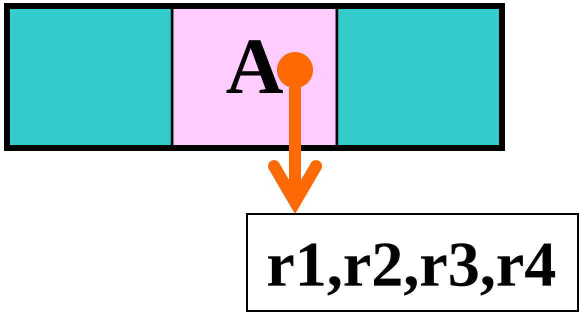
Sequence list of all index keys

Index Matching

- Determining usage of an index for a query
- Complexity of condition determines match.
- Single column indexes: =, <, >, <=, >=, IN <list of values>, BETWEEN, IS NULL, LIKE ‘Pattern’ (meta character not the first symbol)
- Composite indexes: more complex and restrictive rules

Btree and hash files

- Work best for columns with unique values
  - E.g., ID
- Btrees index nodes can store a list of row identifiers for non unique columns
  - the list of row identifiers can be very long



Bitmap Index

- Can be useful for stable columns with few values
- Bitmap:
  - String of bits: **0 (no match) or 1 (match)**
  - One bit for each row
- Bitmap index record
  - Column value (non unique columns)
  - Bitmap (e.g., 0011100)
  - DBMS converts bit position into row identifier.

Bitmap Index on Student Grade

Student Id	Student Grade				
	A	B	C	D	F
101	0	1	0	0	0
102	0	1	0	0	0
103	1	0	0	0	0
104	0	0	1	0	0
105	0	1	0	0	0
106	0	0	0	1	0
107	0	1	0	0	0
108	0	1	0	0	0
109	1	0	0	0	0
110	0	1	0	0	0
111	0	0	1	0	0
112	0	1	0	0	0
113	1	0	0	0	0
114	0	1	0	0	0
115	0	0	0	1	0
116	1	0	0	0	0
117	0	0	0	0	1
118	0	1	0	0	0
119	0	1	0	0	0
120	0	0	1	0	0

0 (no match)
1 (match)

ตาราง bitmap

Grade	**Bitmap**
A	**001000001…**
B	**110010110…**
C	**00010000001…**

Bitmap Index Example

Faculty Table

RowId	FacSSN	…	FacRank
1	098-55-1234		Asst
2	123-45-6789		Asst
3	456-89-1243		Assc
4	111-09-0245		Prof
5	931-99-2034		Asst
6	998-00-1245		Prof
7	287-44-3341		Assc
8	230-21-9432		Asst
9	321-44-5588		Prof
10	443-22-3356		Assc
11	559-87-3211		Prof
12			

Bitmap Index on FacRank

FacRank	Bitmap
Asst	110010010001
Assoc	001000100100
Prof	000101001010

Faculty Table

RowId	FacSSN	…	FacRank		
1	098-55-1234		Asst	1	Asst
2	123-45-6789		Asst	1	Asst
3	456-89-1243		Assc	0	
4	111-09-0245		Prof	0	
5	931-99-2034		Asst	1	Asst
6	998-00-1245		Prof	0	
7	287-44-3341		Assc	0	
8	230-21-9432		Asst	1	Asst
9	321-44-5588		Prof	0	
10	443-22-3356		Assc	0	
11	559-87-3211		Prof	0	
12	220-44-5688		Asst	1	Asst

FacRank	Bitmap
Asst	110010010001
Assoc	001000100100
Prof	000101001010

Bitmap Index on FacRank

Column value	Bitmap
FacRank	**Bitmap**
Asst	110010010001
Assoc	001000100100
Prof	000101001010

- Asst , 110010010001
- Row# 1,2,5,8,12 are Asst. Prof.

Bitmap Join Index

- Bitmap identifies rows of a related table.
- Represents a pre-computed join
- Typically used in query dominated environments such as data warehouses (Chapter 16)

CourseNo	BitmapJoin
204351	110
204111	001

CourseNo	OfferingBitmap
204351	**110**
204111	?

Course

CourseNo	CrsDesc	CrsUnits
204351	*Database*	*3*
204111	*C#*	*3*

Primary key

Offering

OfferNo	OffLocation	OffTime	CourseNo
111	*CPE 203*	*1/2014*	*204351*
222	*CPE 204*	*2/2014*	*204351*
333	*CPE 204*	*2/2014*	*204111*

Foriegn key

Summary of File Structures

	sequential files				
	Unordered	**Ordered**	**Hash**	**B+tree**	**Bitmap**
Sequential search	Y	Y	Extra PRs	Y	N
Key search	Linear	Linear	Constant time	Logarithmic	Y
Range search	N	Y	N	Y	Y
Usage	Primary only	Primary only	Primary or secondary	Primary or secondary	Secondary only

3.A bitmap index: range searches

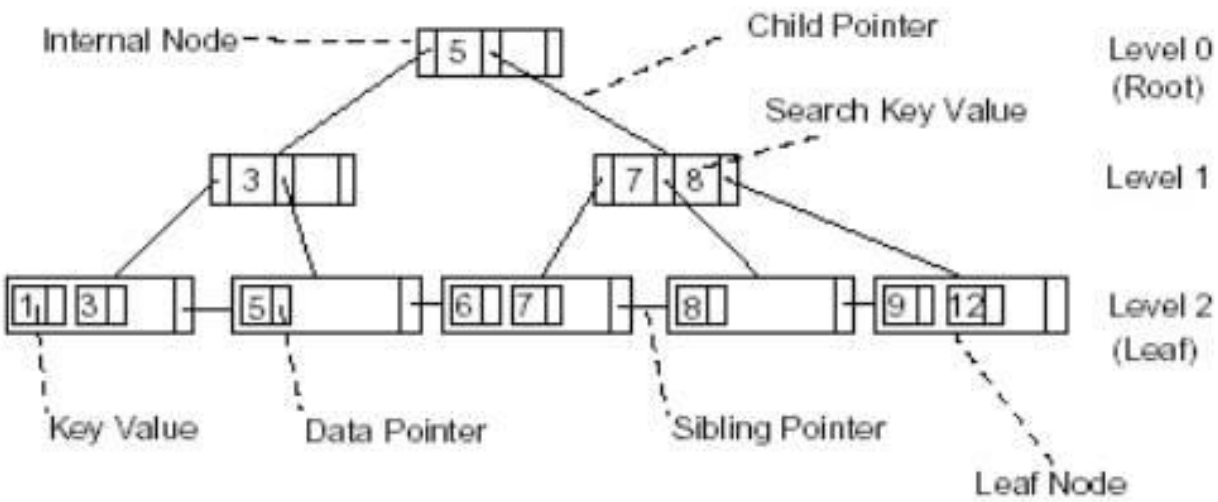
CourseNo	Bitmap
204101	1100001
204102	0010010
204103	0001000

101-102

Union = 1110011

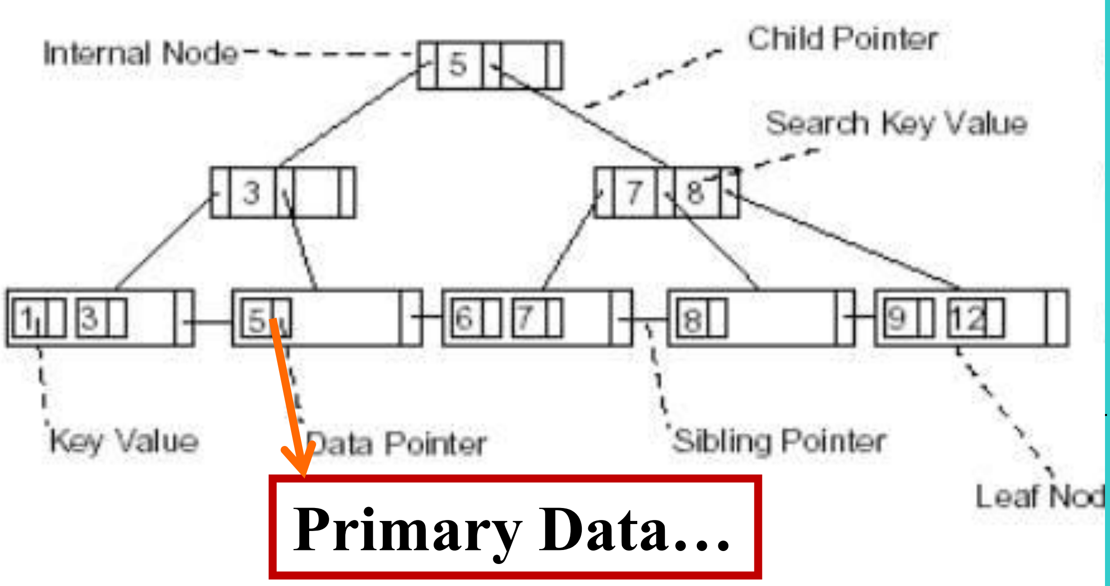
4.PRIMARY FILE STRUCTURE VS SECONDARY FILE STRUCTURE

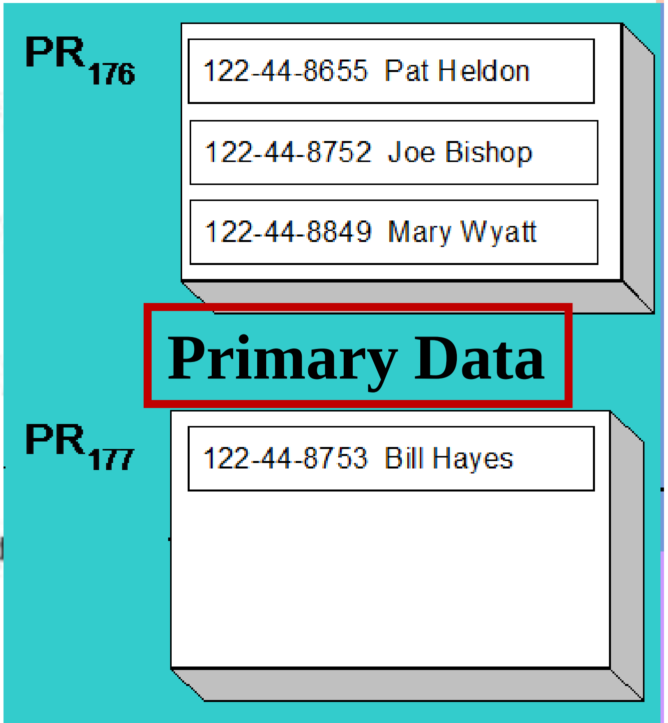
- **Primary file structure**
  - store all the data of a table
- **Secondary file structure**
  - store only key data along with pointers to the data records



sequential files

	Unordered	Ordered	Hash	B+tree	Bitmap
Usage	Primary only	Primary only	Primary or secondary	Primary or secondary	Secondary only

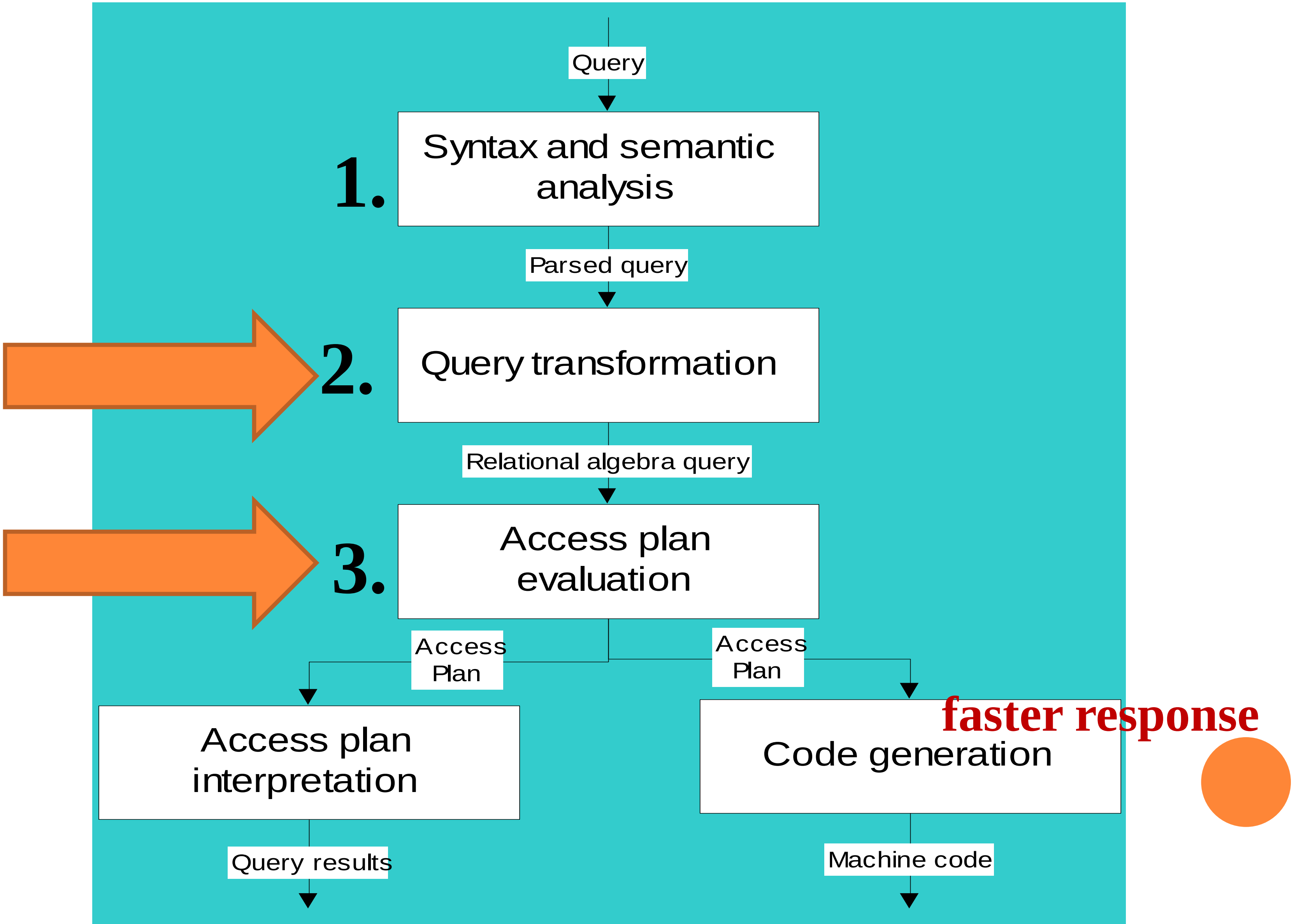




Query Optimization

- Query optimizer determines implementation of queries.
- Major improvement in software productivity
- You can sometimes improve the optimization result through knowledge of the optimization process.

Translation Tasks



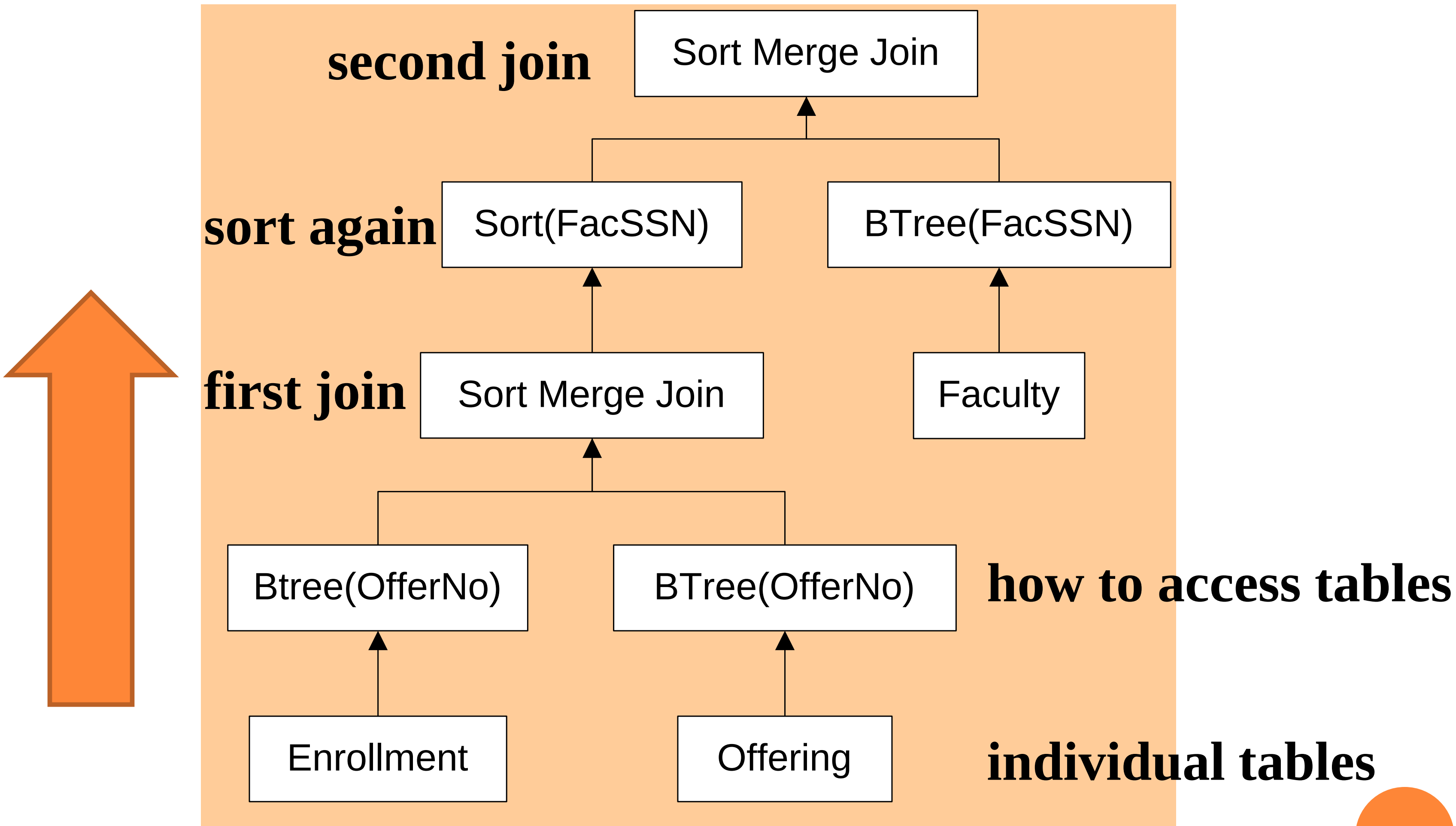
Translation Tasks

1. Analyzes a query for syntax and simple semantic errors
2. Transforms a query into a simplified and standardized format so that the query can be executed faster.
3. Determines how to implement an access plan
   - Access plan: **how to implement a query as operations on files**
     - file structures to access tables,
     - the order of joining tables,
     - the algorithm to join tables

Access plan

- Each operation in an access plan has a corresponding cost formula that estimates the physical record accesses and CPU operations.
- The cost formulas use table profiles to estimate the number of rows in a result.
- The query optimization component chooses the access plan with the lowest cost.

Access Plans



see Oracle Plan in http://docs.oracle.com/cd/B19306_01/server.102/b14211/ex_plan.htm

Access Plan Evaluation

- Optimizer evaluates thousands of access plans
- Access plans vary by join order, file structures, and join algorithm.
- Some optimizers can use multiple indexes on the same table.
- Access plan evaluation can consume significant resources
  - ex., when the query contains more than four tables

Optimization Tips I

- Detailed and current statistics needed
- Save access plans for repetitive queries
- Review access plans to determine problems
- Use hints carefully to improve results

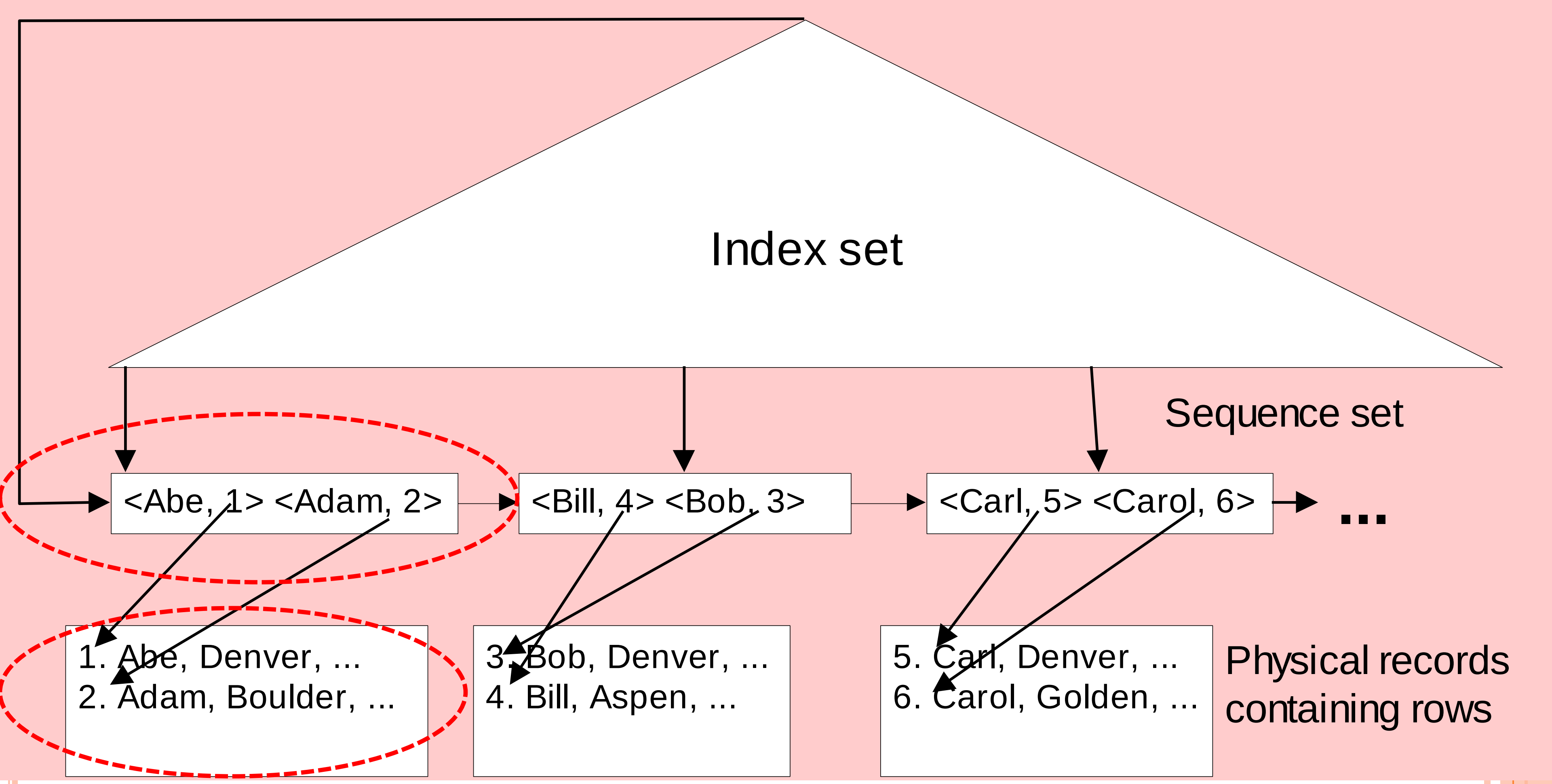
Optimization Tips II

- Replace Type II nested queries with separate queries.
- For conditions on join columns, test the condition on the parent table.
- Do not use the HAVING clause for row conditions.

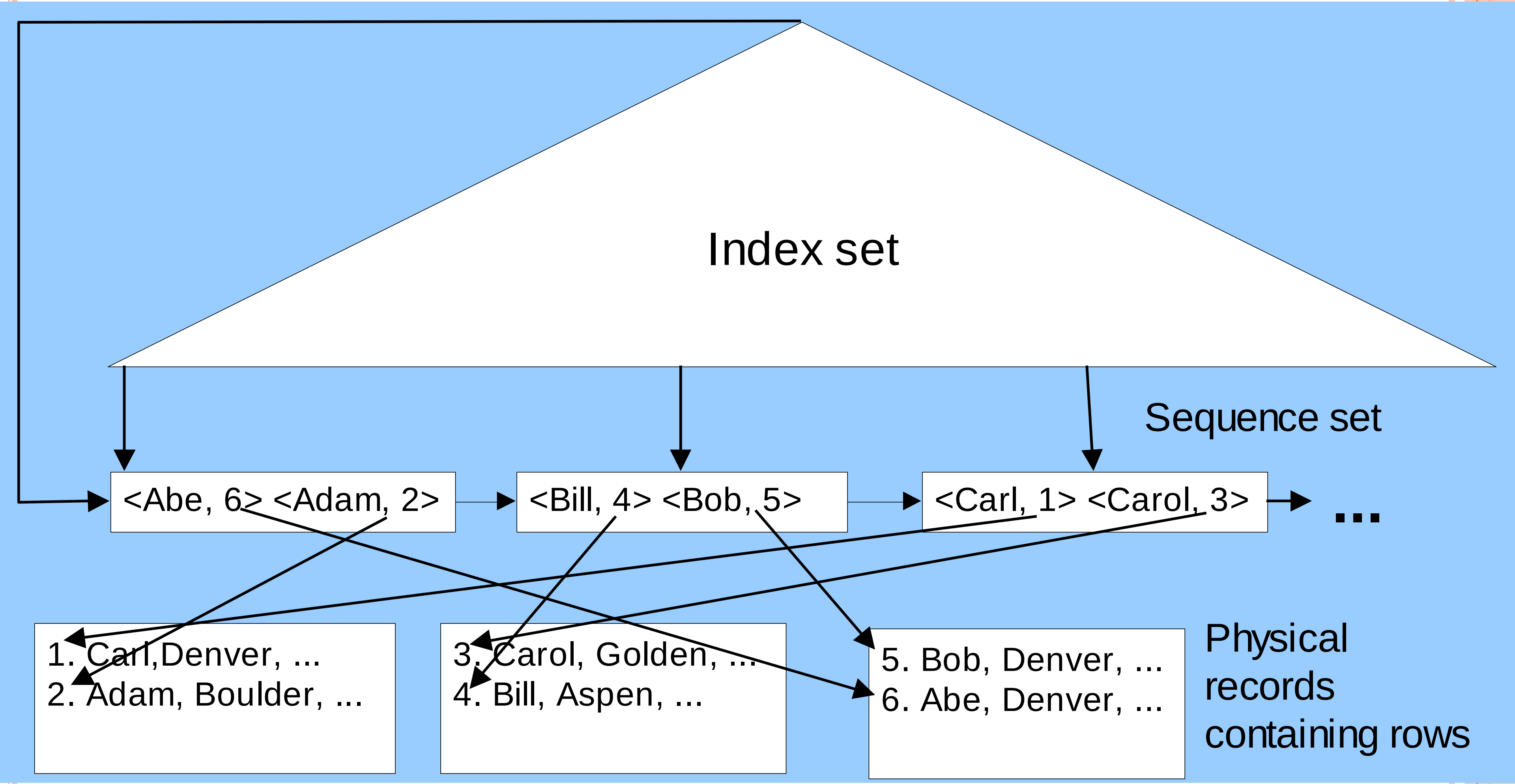
Index Selection

- Most important decision
- Difficult decision
- Choice of clustered and nonclustered indexes

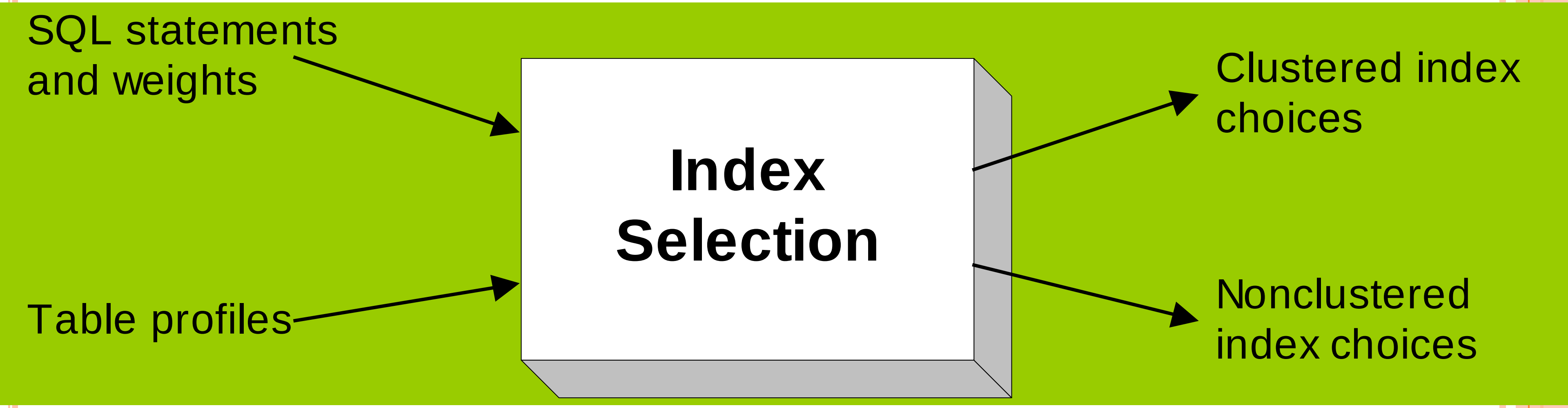
Clustering Index Example



Nonclustering Index Example



Inputs and Outputs of Index Selection



W: frequency of a statement + its importance

Trade-offs in Index Selection

- Balance retrieval against update performance
- Nonclustering index usage:
  - Few rows satisfy the condition in the query
  - Join column usage if a small number of rows result in child table
- Clustering index usage:
  - Larger number of rows satisfy a condition than for nonclustering index
  - Use in sort merge join algorithm to avoid sorting
  - More expensive to maintain

INDEX SELECTION RULES

- A primary key is a good candidate for a clustering index.
- To support joins, consider indexes on foreign keys.
- A frequently updated column is not a good index candidate.
- Volatile tables (lots of insertions and deletions) should not have many indexes.
- Stable columns with few values are good candidates for bitmap indexes if the columns appear in WHERE conditions.

Index Creation

- To create the indexes, the CREATE INDEX statement can be used.
- The word following the INDEX keyword is the name of the index.
- CREATE INDEX is not part of SQL:1999.

Example:

```
CREATE INDEX StdGPAIndex ON Student (StdGPA)
CREATE UNIQUE INDEX OfferNoIndex ON Offering
(OfferNo)
CREATE BITMAP INDEX OffYearIndex ON Offering
(OffYear)
```

Denormalization

- Additional choice in physical database design
- Denormalization combines tables so that they are easier to query.
- Use carefully because normalized designs have important advantages.

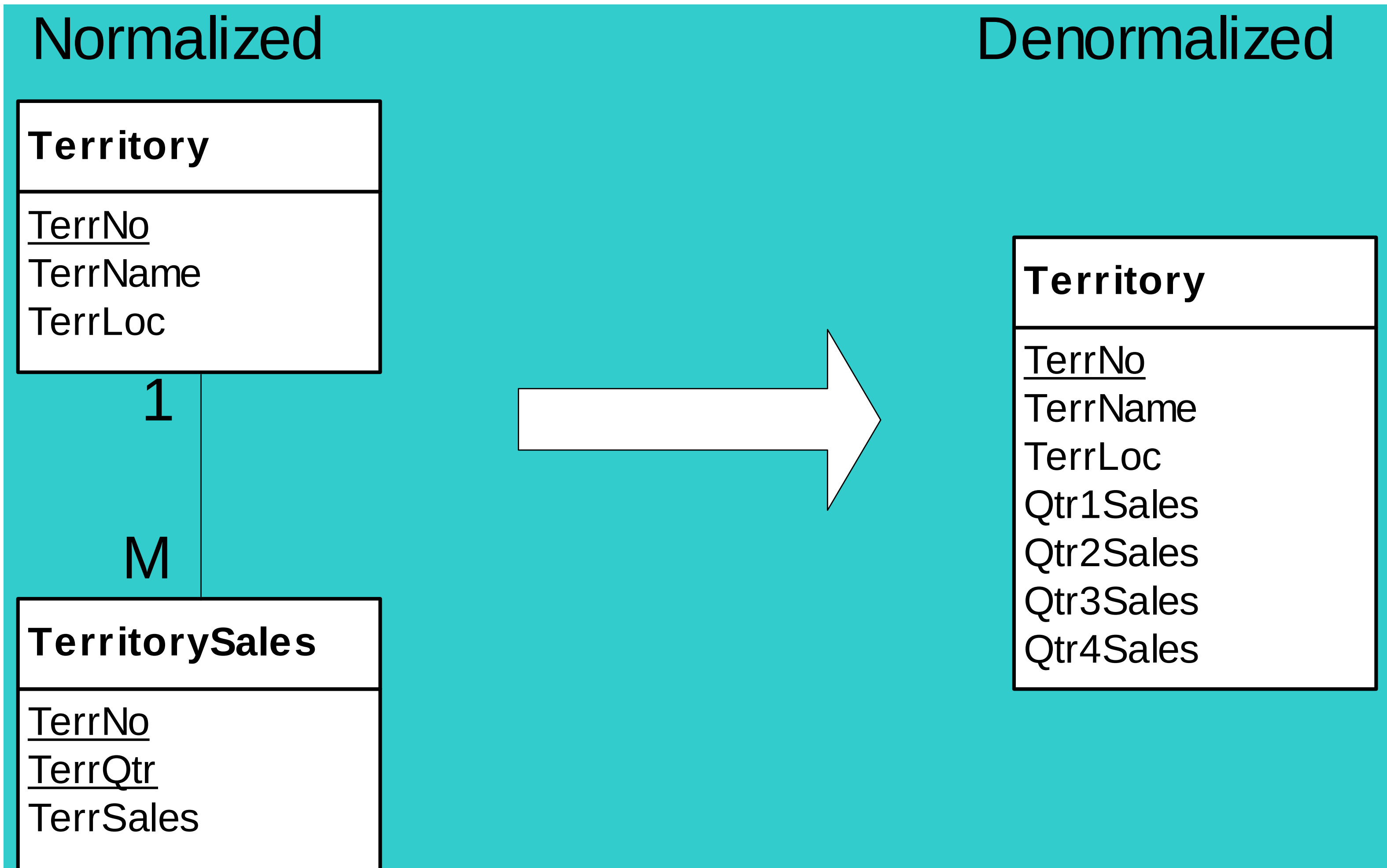
Normalized designs

- Better update performance
- Require less coding to enforce integrity constraints
- Support more indexes to improve query performance

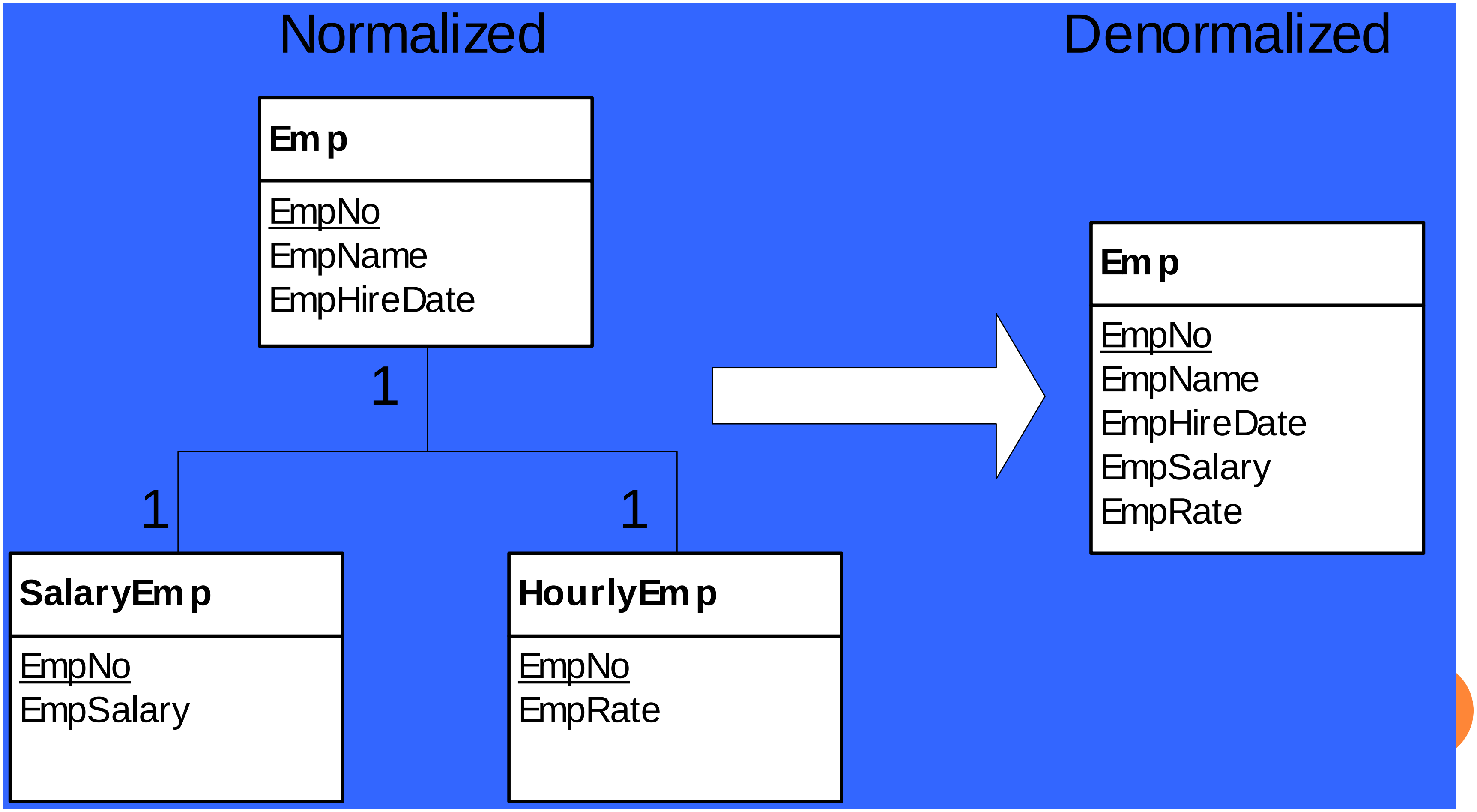
Repeating Groups

- A repeating group is a collection of associated values.
- The rules of normalization force repeating groups to be stored in an M table separate from an associated one table.
- If a repeating group is always accessed with its associated one table, denormalization may be a reasonable alternative.

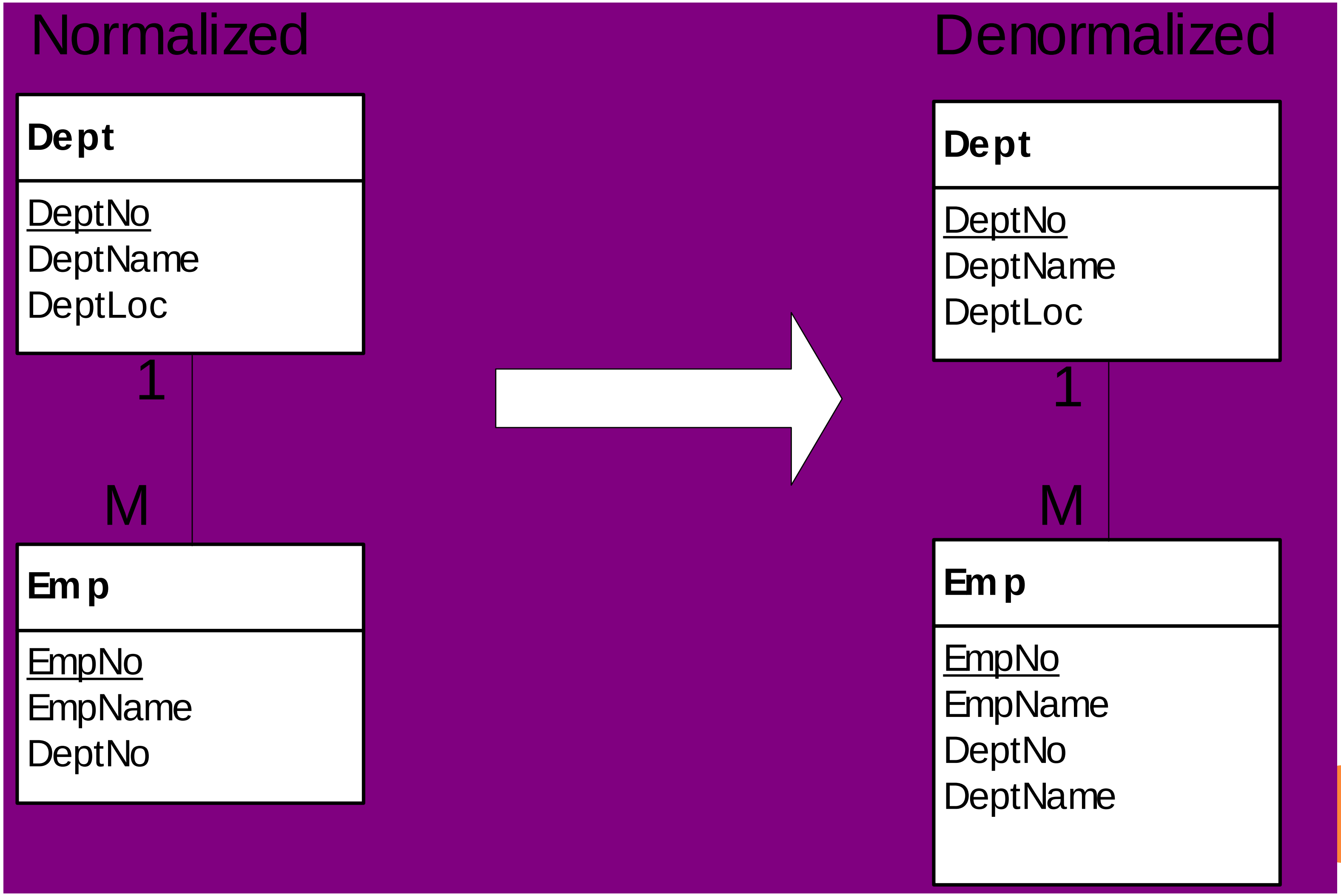
Denormalizing a Repeating Group



Denormalizing a Generalization Hierarchy



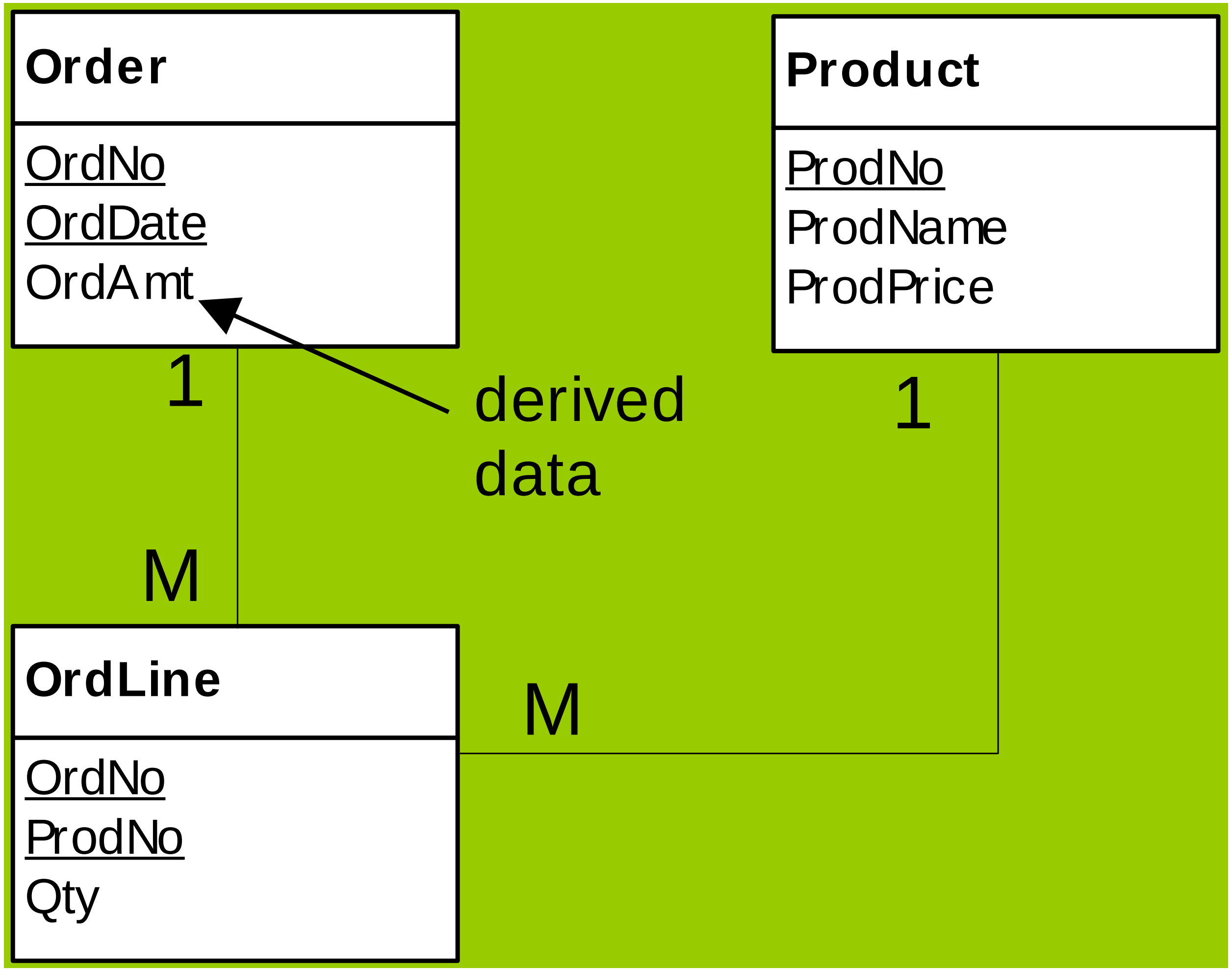
Codes and Meanings



Record Formatting

- Compression is a trade-off between input-output and processing effort.
- Derived data is a trade-offs between query and update operations.

Storing Derived Data to Improve Query Performance

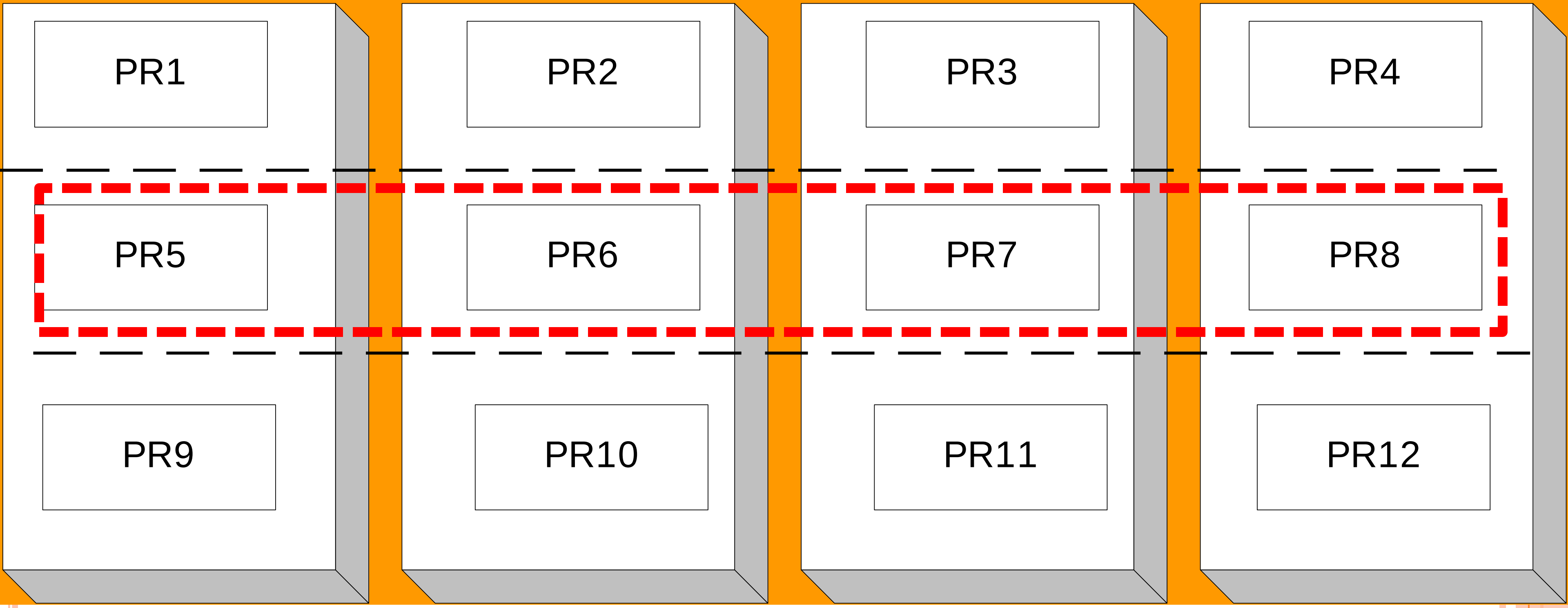


Parallel Processing

- Retrieving many records can be improved by reading physical records in parallel.
- Many DBMSs provide parallel processing capabilities with RAID systems.
- RAID is a collection of disks (a disk array) that operates as a single disk.

Striping in RAID Storage Systems

Each stripe consists of four adjacent physical records. Three stripes are shown separated by dotted lines.



Other Ways to Improve Performance

- Transaction processing: add computing capacity and improve transaction design.
- Data warehouses: add computing capacity and store derived data.
- Distributed databases: allocate processing and data to various computing locations.

Summary

- Goal: minimize computing resources
- Table profiles and application profiles must be specified in sufficient detail.
- Environment: file structures and query optimization
- Monitor and possibly improve query optimization results
- Index selection: most important decision
- Other techniques: denormalization, record formatting, and parallel processing