Accessing and Indexing Data

Accessing and Indexing Data

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Kevin C.C. Chang, Professor

Computer Science @ Illinois

Learning Objectives

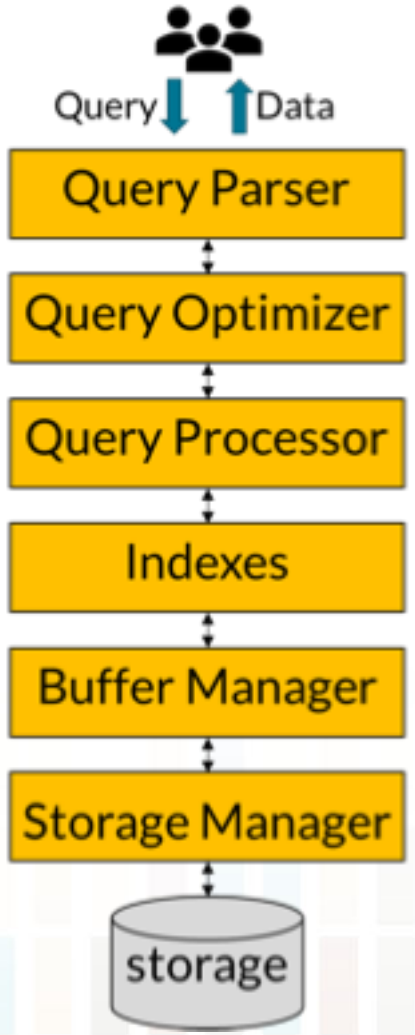
By the end of this video, you will be able to:

- Describe the general “stack” of components for answering queries in DBMS.
- Explain what accessing data means and why.
- Explain what indexing data means and why.

Querying a Database: The "Query-To-Data" Stack

Query: SELECT name, grade FROM Enrolls WHERE grade = 80 AND course = "CS411"

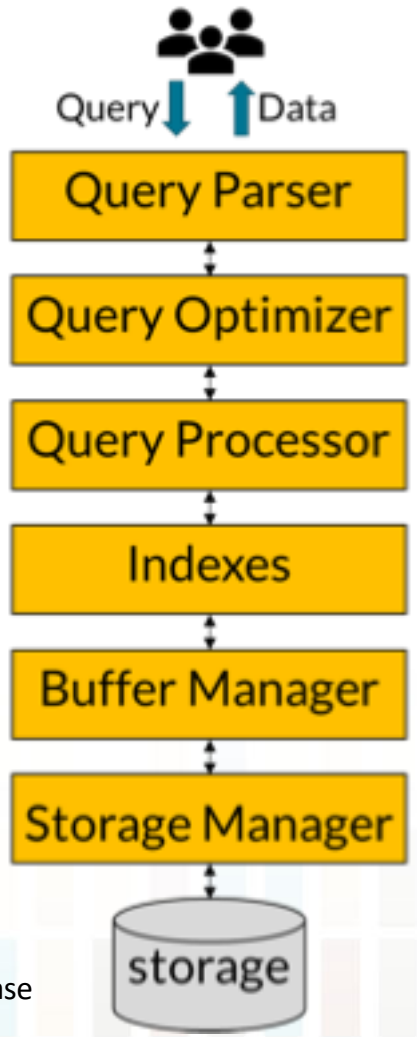
- **Query Parsing**:
  - Query parser parses the query.
- **Query Optimization**:
  - Query optimizer generates a "query plan".
- **Query processing**:
  - Query processor executes the plan.
- **Data Indexing**:
  - Indexes provides "maps" to where data is in storage.
- **Data Accessing**:
  - Buffer and storage managers manage I/O between storage and memory.



The big picture of querying a database

Accessing Data: Getting to Our Data!

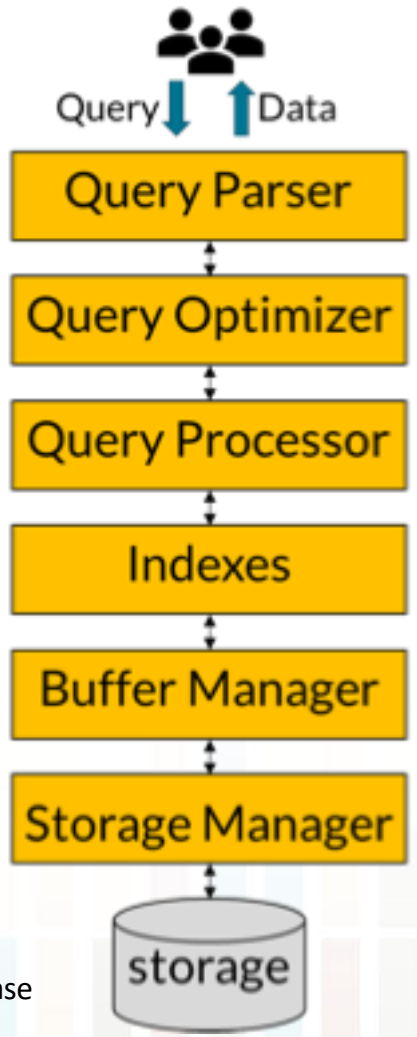
- Data is large.
  - Main memory is limited
- Data must be persistent.
  - Main memory is volatile.
- So, “big data” is stored in external storage.
- To answer a query, we must bring data to memory.
- Buffer manager:
  - Deals with “buffering” data in memory for processing.
- Storage manager:
  - Deals with reading and writing data to storage.



The big picture of querying a database

Indexing Data: Mapping Our Data!

- Data stored in storage is large.
- Data we need for querying is often a small fraction
  - E.g., those Enrolls tuples with grade = 80 and course = “CS411”
- How can we find necessary data quickly?
- Query processor needs “maps” to locate data
  - … without having to scan the entire database.
- An index provides such a “map” for locating data
  - By values, such as grade 80 or course “CS411”.



The big picture of querying a database

Accessing Data

Accessing and Indexing Data

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Kevin C.C. Chang, Professor
Computer Science @ Illinois

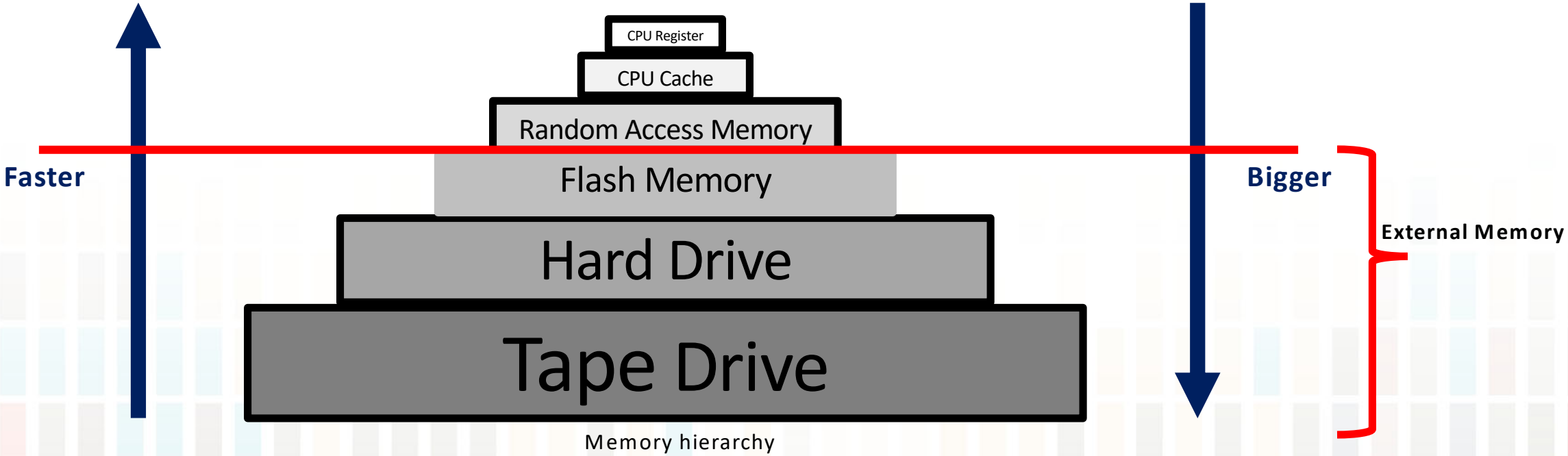
Learning Objectives

By the end of this video, you will be able to:

- Explain why a database system must use external memory.
- Describe the memory hierarchy and how database process queries with respect to the hierarchy.
- Explain how data is stored on and access from disks.
- Describe the modes of accessing data from disks.

Where Is Data? No Longer All in Memory

- Data cannot fit in main memory. It is big!
  - No matter how main memory becomes larger. Data grows too.
- Must utilize memory hierarchy– with **external memories**.

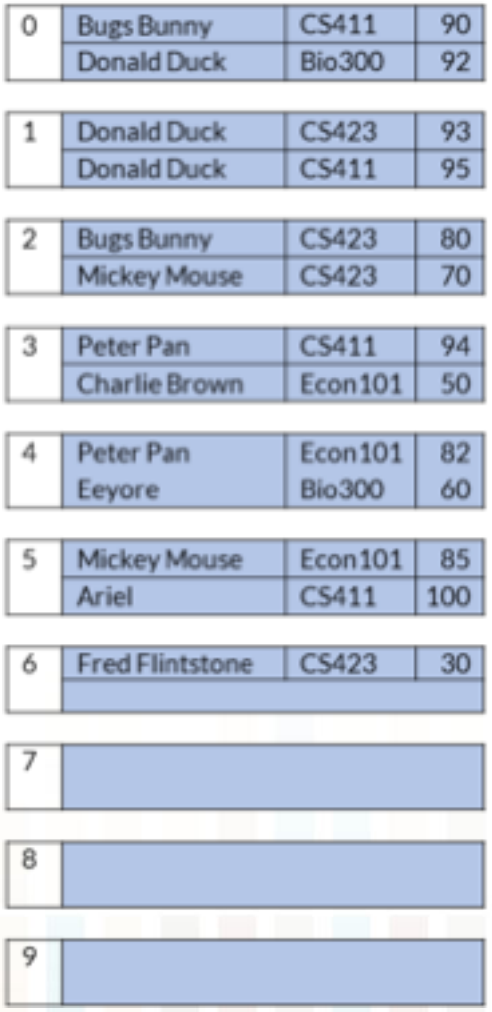


Storing Data in External Storage

- Storage = Array of blocks
- Unit of storage allocation/access: Block
  - Name not quite standardized– “page”, “cluster” too.
  - Each block has a fix size depending on the file system/OS.
  - Usually 1K, 4K, 8K bytes.
- Table is stored on a set of blocks
  - Contiguous or not.

name	course	grade
Bugs Bunny	CS411	90
Donald Duck	Bio300	92
Donald Duck	CS423	93
Donald Duck	CS411	95
Bugs Bunny	CS423	80
Mickey Mouse	CS423	70
Peter Pan	CS411	94
Charlie Brown	Econ101	50
Peter Pan	Econ101	82
Eeyore	Bio300	60
Mickey Mouse	Econ101	85
Ariel	CS411	100
Fred Flintstone	CS423	30

Enrolls table



Storing the Enrolls table on disk, in blocks.

Storage: The Evolution of Hard Disk Drive

1956

Parameter	Started with (1956)
Capacity (formatted)	3.75 megabytes[12]
Physical volume	68 cubic feet (1.9 m^3)[c][6]
Weight	?
Average access time	approx. 600 milliseconds[6]
Price	US$9,200 per megabyte (1961)[19]
Data density	2,000 bits per square inch[22]
Average lifespan	~2000 hrs MTBF[*citation needed*]

Improvement of HDD characteristics over time. Hard disk drive. Retrieved from https://en.wikipedia.org/wiki/Hard_disk_drive

IBM 350 Disk Storage Unit (1956).

How heavy was it?

A. 1 KG (2.2 LB)

B. 10 KG

C. 100 KG

D. 1 Ton (=1000 KG, 2200 LBs)

2017

Seagate Barracuda ST1000DM010
Part Number: ST1000DM010

GENERAL /

Capacity	1 TB
Bytes per Sector	4096
Interface	SATA 6Gb/s
Buffer Size	64 MB
Weight	14.11 oz
Manufacturer	Seagate Technology

PERFORMANCE /

Internal Data Rate	210 MBps
Seek Time	8.5 ms (average)
Drive Transfer Rate	600 MBps (external)
Average Latency	4.16 ms
Spindle Speed	7200 rpm
Track-to-Track Seek Time	1 ms

Example specification of a current hard drive. Seagate Barracuda ST1000DM010 - hard drive - 1 TB - SATA 6Gb/s Specifications. Retrieved from https://www.cnet.com/products/seagate-barracuda-st1000dm010-hard-drive-1-tb-sata-6gb-s/specs/

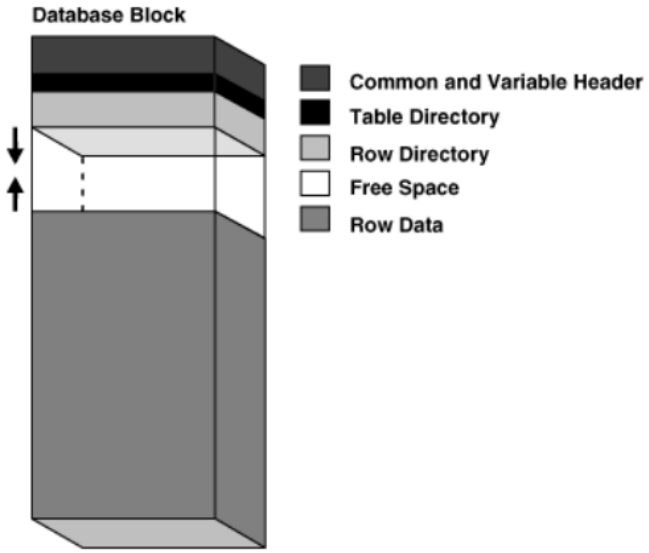
How Is Data Stored in Blocks?

ORACLE DATABASE

Data Block Format

The Oracle data block format is similar regardless of whether the data block contains table, index, or clustered data. Figure 2-2illustrates the format of a data block.

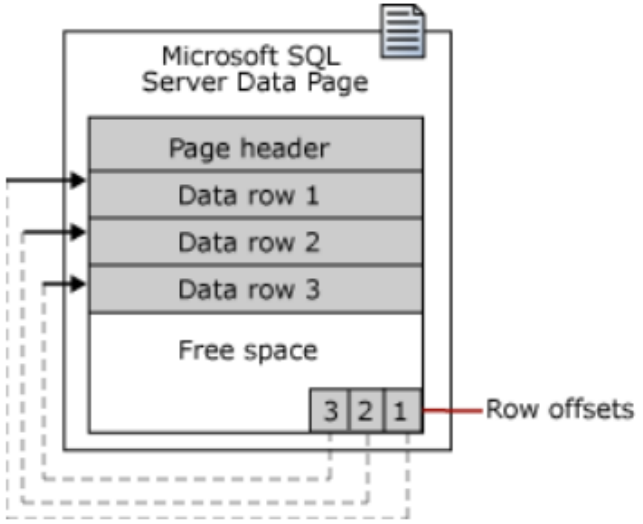
Figure 2-2 Data Block Format



Data block format, Oracle. Data blocks, extents, and segments, Oracle. Retrieved from https://docs.oracle.com/cd/B19306_01/server.102/b14220/logical.htm

Microsoft SQL Server

Data rows are put on the page serially, starting immediately after the header. A row offset table starts at the end of the page, and each row offset table contains one entry for each row on the page. Each entry records how far the first byte of the row is from the start of the page. The entries in the row offset table are in reverse sequence from the sequence of the rows on the page.

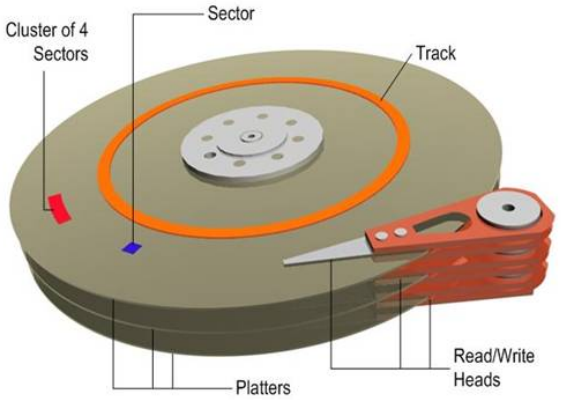


Microsoft SQL server data page. Understanding pages and extents, Microsoft. Retrieved from https://technet.microsoft.com/en-us/library/ms190969

Accessing Data on Disks: *Where Does the Time Go?*

- **Seek Time:** Time for disk head to move to the track of the block. Current: ~10 ms (1956: 600 ms).
- **Rotational Latency:** Time for first sector of the block to rotate under the head. Current: ~ 5 ms (1956: 25 ms).
- **Transfer Time:** Time to read or write data to the sectors. Current: ~200 MB/s (1956: 8800 char/s), or 0.02 ms for 4KB.

Internals of a 2.5-inch SATA hard disk drive. Hard disk drive. Retrieved from https://en.wikipedia.org/wiki/Hard_disk_drive



Simplified schematic diagram of hard drive internals. Disk partition alignment best practices for SQL Server. Retrieved from https://technet.microsoft.com/en-us/library/dd758814

Seagate Barracuda ST1000DM010
Part Number: ST1000DM010

GENERAL /

Capacity	1 TB
Bytes per Sector	4096
Interface	SATA 6Gb/s
Buffer Size	64 MB
Weight	14.11 oz

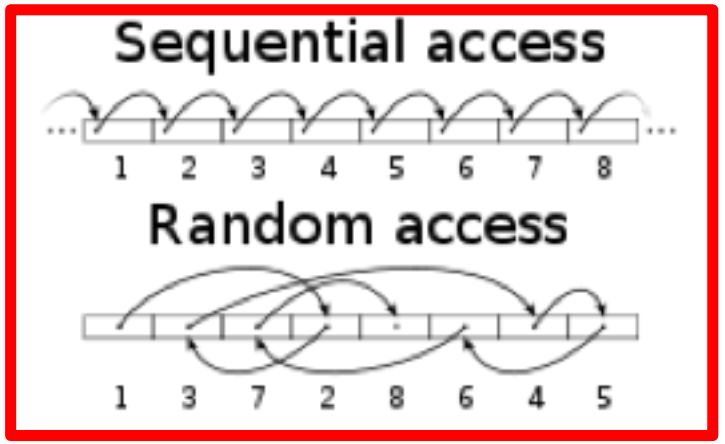
PERFORMANCE /

Internal Data Rate	210 MBps
Seek Time	8.5 ms (average)
Drive Transfer Rate	600 MBps (external)
Average Latency	4.16 ms
Spindle Speed	7200 rpm
Track-to-Track Seek Time	1 ms

Example specification of a current hard drive. Seagate Barracuda ST1000DM010 - hard drive - 1 TB - SATA 6Gb/s Specifications. Retrieved from https://www.cnet.com/products/seagate-barracuda-st1000dm010-hard-drive-1-tb-sata-6gb-s/specs/

Accessing Data on Disk: *There Are Two Modes*

- **Random Access**
  - Access data **randomly**- at arbitrary at positions.
  - Get(address) ⟶ Block
  - Cost = Seek Time + Rotational Latency + Transfer Time
  - For 4KB block: 10+5+0.02 ~ 15 ms

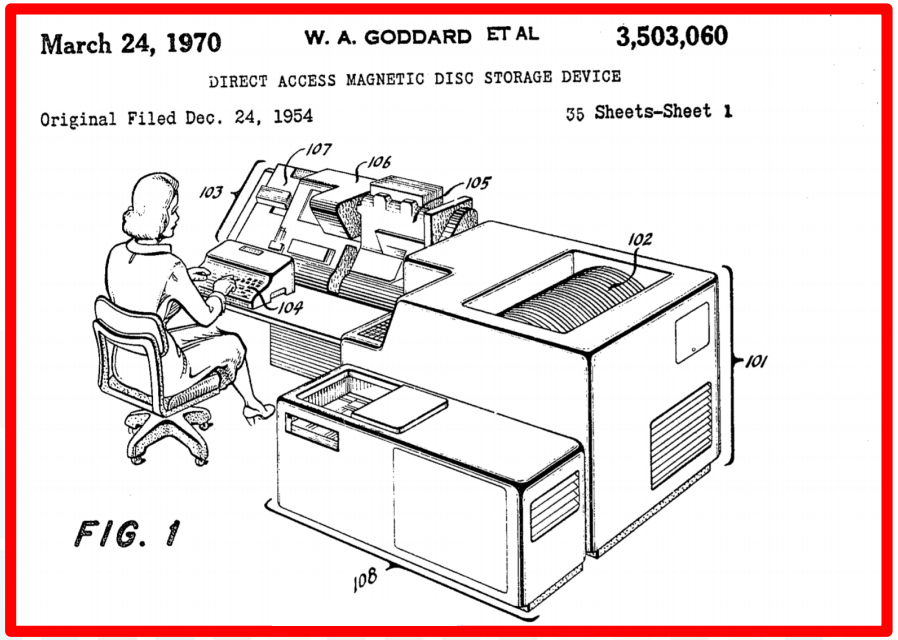


Random access compared to sequential access. Random access [Online image]. Retrieved from https://en.wikipedia.org/wiki/Random_access

- **Sequential Access**
  - Access data **sequentially**– in a predetermined, ordered sequence.
  - GetNext() ⟶ Block
  - Cost = Transfer Time
  - For 4KB block: 0.02 ms

Food for Thought

Where is the hard drive? Can you find it in the following picture, Fig.1, of the US Patent that invented the "direct access magnetic disc storage device"?


March 24, 1970 W. A. GODDARD ET AL 3,503,060

DIRECT ACCESS MAGNETIC DISC STORAGE DEVICE

Original Filed Dec. 24, 1954 35 Sheets-Sheet 1

FIG. 1

FIG. 1 is a pictorial view of the principal components of a machine incorporating the invention.

Fig. 1. William A Goddard and John J Lynott , "Direct access magnetic disc storage device." U.S. Patent 3,503,060, issued March 24, 1970

Indexing Data

Accessing and Indexing Data

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Kevin C.C. Chang, Professor
Computer Science @ Illinois

Learning Objectives

By the end of this video, you will be able to:

- Explain why indexing is important for a database system.
- Describe the main function of an index.
- Explain how the concept of indexing is used in real life.

Why Indexing: Where Is Data?

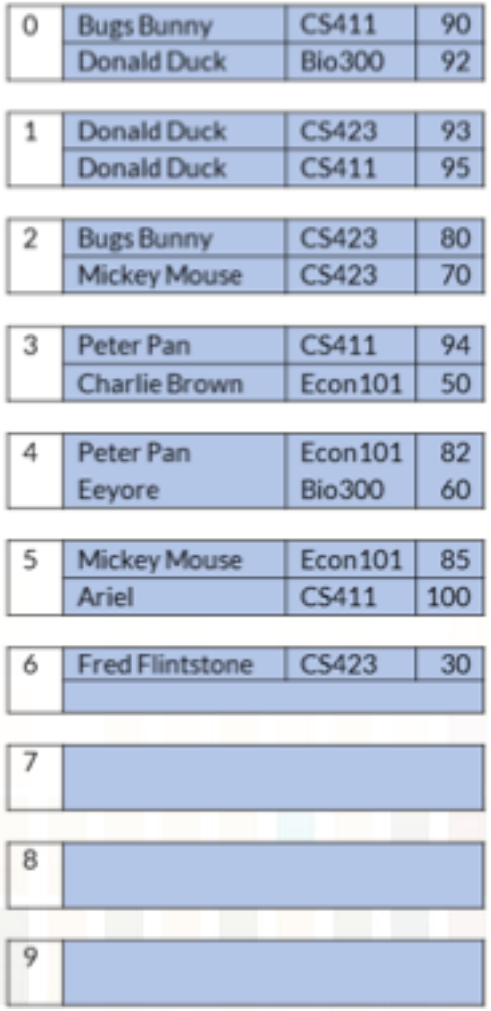
- DBMS should answer queries quickly-- many challenges.
- First of all, where is data? Each query may focus on some part of the database, and we need to locate it.
- What data we need is specified by values.
  - *Q1: Find students who scored **80** in **CS411**.*
  - *Q2: Find beers whose alcohol is **above 5%**.*

Regardless of Data Models or Query Languages: *Need to find Data to Answer Queries-- by Values*

- A query specifies data of interest by values in the "WHERE" conditions (or the like).
  - *Q1: Find students who scored 80 in CS411.*
    - Relational DB: SELECT name, grade FROM Enrolls WHERE grade = 80 AND course = "CS411"
    - Document DB: db.Enrolls.find({grade:80, course:"CS411"}, {name:1, grade:1}}
    - Graph DB: MATCH (s:Student)-[e:Enrolls {grade:80}]->(c:Course {title:"CS411"}) RETURN s.name, e.grade
  - *Q2: Find beers whose alcohol is above 5%.*
    - Relational DB: SELECT name, alcohol FROM Beers WHERE alcohol > 0.05
    - Document DB: db.Beers.find({alcohol:{$gt:0.05}}, {name:1, alcohol:1}}
    - Graph DB: MATCH (b:Beer) WHERE b.alcohol > 0.05 RETURN b.name, b.alcohol
- Find records (tuples, documents, nodes) that **match** certain **values**, or **search keys**.
  - For Q1: match = "equal" for search key 80 on grade attribute; "equal" for "CS411" on course.
  - For Q2: match = "greater than" for search key 0.05 on alcohol.

How De We Find Target Data?

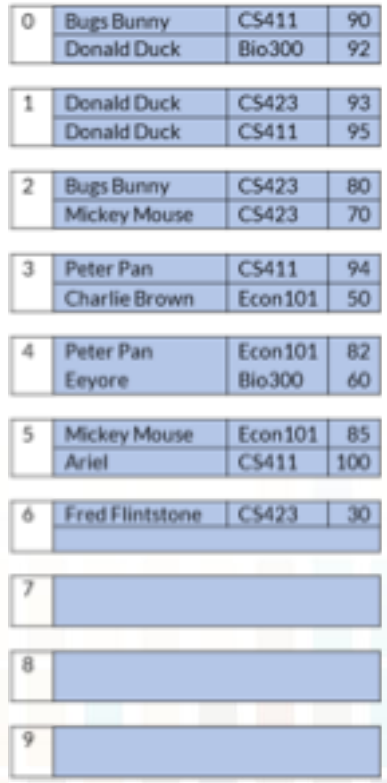
- Q1: Find students **who scored 80** in CS411.
- Go to disk. Then?



0	Bugs Bunny	CS411	90
	Donald Duck	Bio300	92
1	Donald Duck	CS423	93
	Donald Duck	CS411	95
2	Bugs Bunny	CS423	80
	Mickey Mouse	CS423	70
3	Peter Pan	CS411	94
	Charlie Brown	Econ101	50
4	Peter Pan	Econ101	82
	Eeyore	Bio300	60
5	Mickey Mouse	Econ101	85
	Ariel	CS411	100
6	Fred Flintstone	CS423	30
7			
8			
9			

Method 1: Table Scan

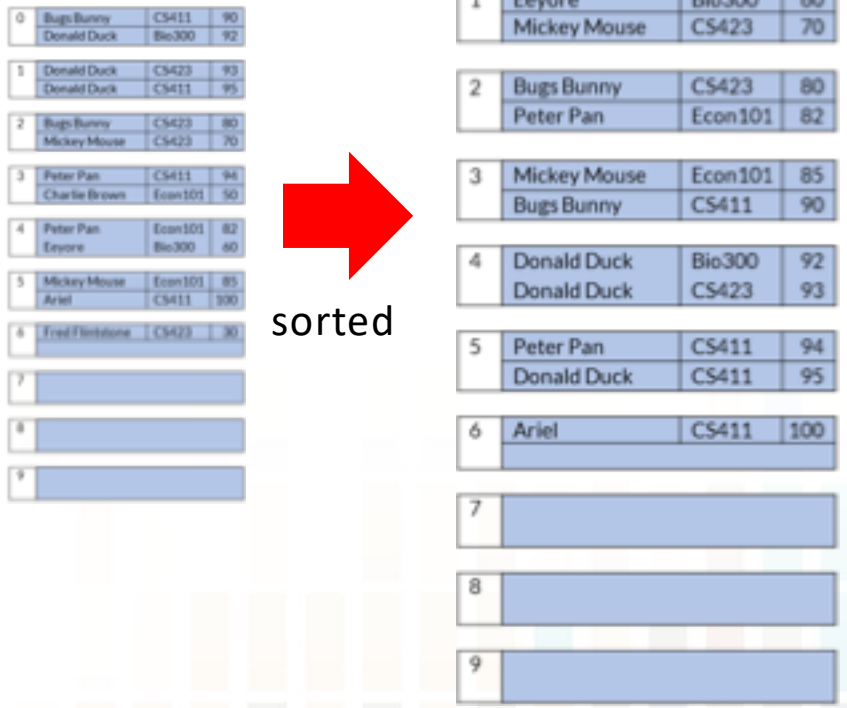
- Suppose table is stored in an unordered set of blocks.
  - Called a **heap file**.
- *Q1: Find students **who scored 80** in **CS411**.*
- Go to where file starts.
- Read next block. Process it.
  - Keep the tuples whose grade = 80 and course = “CS411”.
- Until the end of file.
- Cost = O(B), for a file of B blocks.



Heap file with blocks storing the Enrolls tuples.

Method 2: Binary Search on Sorted Files

- Suppose table is sorted on disk by the target attribute.
  - Called a **sorted file**.
- *Q1: Find students **who scored 80** in **CS411**.*
- Perform binary search to find target values.
  - Locate grade = 80.
- Check the remaining conditions.
  - Check course = “CS411”
- Cost = O($Log_2(B)$).



Sorted file (by grade) with blocks storing the Enrolls tuples.

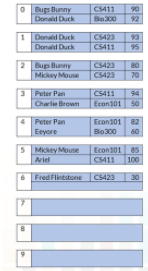
Can We Do Better?

If the table occupies B blocks:

- Table Scan:
  - Access B blocks.
- Binary Search on Sorted Files:
  - We can only sort the table in one order.
    - If by grade then not by course.
  - Even when sorted, binary search is still expensive.
    - $Log_2(B)$ blocks.

Method 1: Table Scan

- Suppose table is stored in an unordered set of blocks.
  - Called a **heap file**.
- *Q1: Find students **who scored 80** in **CS411**.*
- Go to where file starts.
- Read next block. Process it.
  - Keep the tuples whose grade = 80 and course = "CS411".
- Until the end of file.
- Cost = O(B), for a file of B blocks.



Heap file with blocks storing the Enrolls tuples.

Table scan method for accessing data

Method 2: Binary Search on Sorted Files

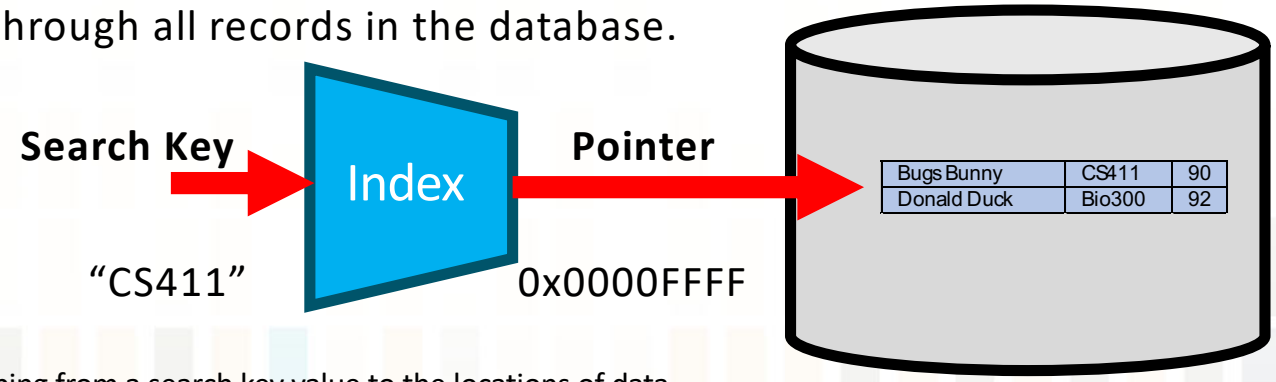
- Suppose table is sorted on disk by the target attribute.
  - Called a **sorted file**.
- *Q1: Find students **who scored 80** in **CS411**.*
- Perform binary search to find target values.
  - Locate grade = 80.
- Check the remaining conditions.
  - Check course = "CS411"
- Cost = O($Log_2(B)$).

Sorted file (by grade) with blocks storing the Enrolls tuples.

Binary search method for accessing data

Indexing: What Is an Index?

- Indexing is the action of building an index.
- Index: A data structure that provides a mapping from a **search key** value to the locations, or **pointers**, of data that matches the key.
- Search key = any subset of the fields of a relation
  - Search key is not the same as key (minimal set of fields uniquely identifying a record).
- It speeds up query processing
  - So we can quickly locate where data is.
  - So we do not need to scan through all records in the database.



Index: A mapping from a search key value to the locations of data

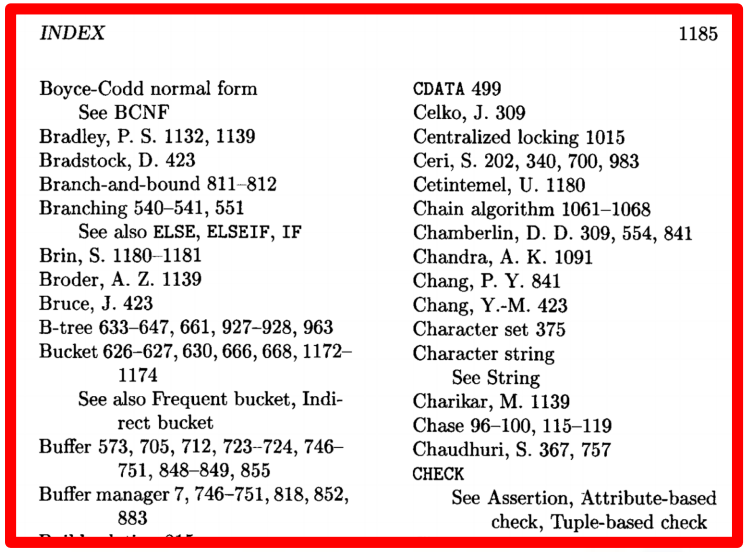
You Have Been Using Indexes!

Look for Them in the Real World

Highway food signs. Ask George: Why aren't there more independent restaurants listed on those interstate highway "Food" signs? [Online image]. Retrieved from https://www.stlmag.com/dining/Ask-George-Why-arent-there-more-independent-restaurants-listed-on-those-interstate-highway-Food-signs/

A road sign. The way home [Online image]. Retrieved from https://pixabay.com/en/at-the-court-of-sky-blue-sign-3218573/

A room directional sign. Directional sign [Online image]. Retrieved from https://www.mydoorsign.com/Directory/Florence-Directional-Sign/SKU-SE-3528.aspx



INDEX 1185

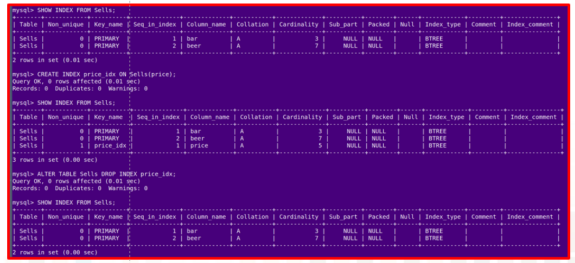
Concept index, in Database Systems: The Complete Book (2nd Ed.) by Hector Garcia-Molina, Jeffrey D. Ullman, , Jennifer Widom

Indexes in a Database

- DBMS auto manages or lets users manage indexes.
- DBMS uses indexes to speed up query processing.

Manipulating Indexes

- Most DBMS provides ways for users to create indexes.
  - But syntax may be slightly different. We use MySQL here.
- Create an index: **CREATE INDEX** <name> on <table>(<attributes>);
- Show indexes:
  SHOW INDEX FROM <table>;
- Delete an index
  ALTER TABLE <table>
  DROP INDEX < name>;



Indexing examples in MySQL

Manipulating indexes

```
mysql> SHOW INDEX FROM Sells;
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
| Sells |          0 | PRIMARY  |            1 | bar         | A         |     1097401 |     NULL | NULL   |      | BTREE      |         |               |
| Sells |          0 | PRIMARY  |            2 | beer        | A         |     1023765 |     NULL | NULL   |      | BTREE      |         |               |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
2 rows in set (0.00 sec)

mysql> EXPLAIN SELECT beer, bar FROM Sells WHERE price < 3;
+----+-------------+-------+------------+------+---------------+------+---------+------+---------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows    | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+---------+----------+-------------+
|  1 | SIMPLE      | Sells | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1097401 |    33.33 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+---------+----------+-------------+
1 row in set, 1 warning (0.00 sec)

mysql> SELECT beer, bar FROM Sells WHERE price < 3;
+------+-----------+
| beer | bar       |
+------+-----------+
| Bud  | Green Bar |
+------+-----------+
1 row in set (0.22 sec)
```

```
mysql> SHOW INDEX FROM Sells;
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
| Sells |          0 | PRIMARY  |            1 | bar         | A         |     1097401 |     NULL | NULL   |      | BTREE      |         |               |
| Sells |          0 | PRIMARY  |            2 | beer        | A         |     1023765 |     NULL | NULL   |      | BTREE      |         |               |
| Sells |          1 | price_idx|            1 | price       | A         |          18 |     NULL | NULL   |      | BTREE      |         |               |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
3 rows in set (0.00 sec)

mysql> EXPLAIN SELECT beer, bar FROM Sells WHERE price < 3;
+----+-------------+-------+------------+-------+---------------+-----------+---------+------+------+----------+-----------------------+
| id | select_type | table | partitions | type  | possible_keys | key       | key_len | ref  | rows | filtered | Extra                 |
+----+-------------+-------+------------+-------+---------------+-----------+---------+------+------+----------+-----------------------+
|  1 | SIMPLE      | Sells | NULL       | range | price_idx     | price_idx | 4       | NULL |    1 |   100.00 | Using where; Using index |
+----+-------------+-------+------------+-------+---------------+-----------+---------+------+------+----------+-----------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SELECT beer, bar FROM Sells WHERE price < 3;
+------+-----------+
| beer | bar       |
+------+-----------+
| Bud  | Green Bar |
+------+-----------+
1 row in set (0.00 sec)
```

Running a query on “price” without and with an index in MySQL

What Are the Desired Properties?

- **Fast**, of course!
- **Scalable** to larger data or more users.

- What else? (This is our Food for Thought.)

There's a great deal of research on this. Here's a quick summary.

89

Response Times: The 3 Important Limits

by Jakob Nielsen on January 1, 1993

Summary: There are 3 main time limits (which are determined by human perceptual abilities) to keep in mind when optimizing web and application performance.

Excerpt from Chapter 5 in my book Usability Engineering, from 1993:

The basic advice regarding response times has been about the same for thirty years [Miller 1968; Card et al. 1991]:

- **0.1 second** is about the limit for having the user feel that the system is **reacting instantaneously**, meaning that no special feedback is necessary except to display the result.
- **1.0 second** is about the limit for the **user's flow of thought** to stay uninterrupted, even though the user will notice the delay. Normally, no special feedback is necessary during delays of more than 0.1 but less than 1.0 second, but the user does lose the feeling of operating directly on the data.
- **10 seconds** is about the limit for **keeping the user's attention** focused on the dialogue. For longer delays, users will want to perform other tasks while waiting for the computer to finish, so they should be given feedback indicating when the computer expects to be done. Feedback during the delay is especially important if the response time is likely to be highly variable, since users will then not know what to expect.

Response time: The 2 important limits. “What is considered a good response time for a dynamic, personalized web application?”. Retrieved from https://stackoverflow.com/questions/164175/what-is-considered-a-good-response-time-for-a-dynamic-personalized-web-applicat

Food for Thought

The desired properties of a good index are missing a critical one (at least). What is that?

Hint: Order the following by how you prefer a DBMS to respond to queries:

a) *Equally fast for all queries.*
b) *Equally slow for all queries.*
c) *Very fast for some random queries; very slow for others.*

ISAM Index for Static Data

Accessing and Indexing Data

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Kevin C.C. Chang, Professor

Computer Science @ Illinois

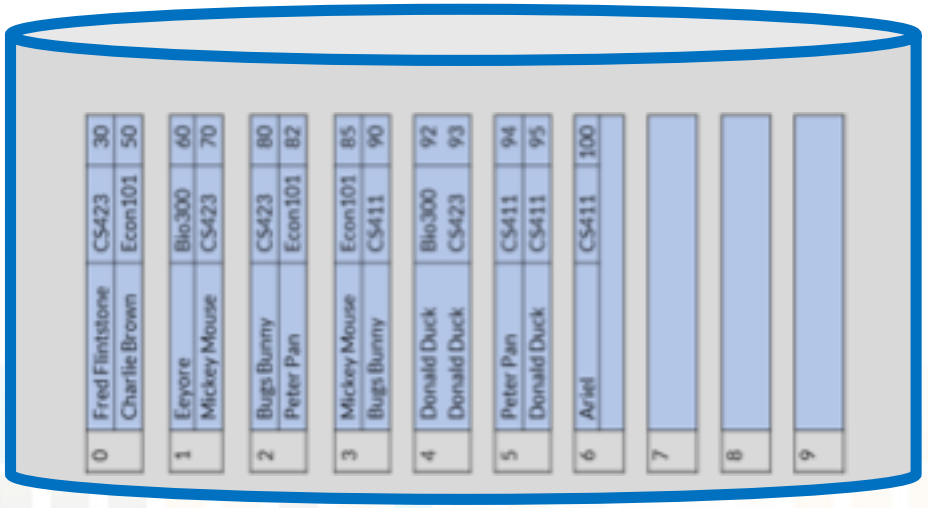
Learning Objectives

By the end of this video, you will be able to:

- Describe the key idea of ISAM.
- Identify the structure and basic operations of ISAM.
- Explain why ISAM is insufficient for dynamic data.

Let’s Index This Sorted File of Blocks

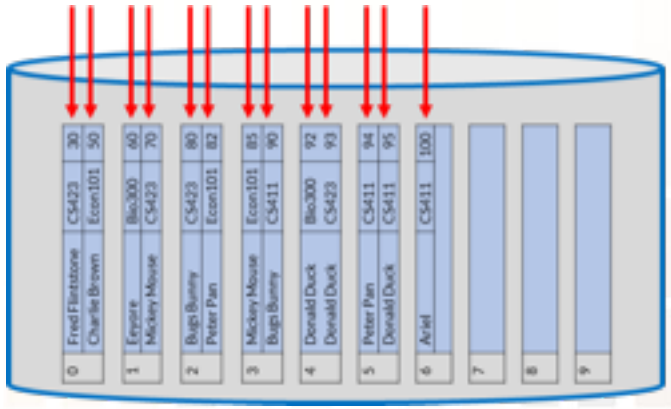
- We want to create ISAM index for the Enrolls table on the grade attribute.
- Assume blocks are already sorted by the attribute.
- Can also handle heap file (unsorted)– assume sorted for ease of discussion.
- How do we create “pointers” to search key values (grades) in the sorted file?



Enrolls table on disk, sorted by grades.

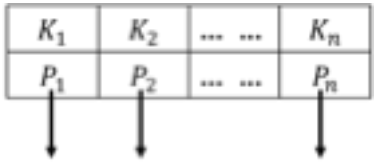
To index: We Ask –
What Pointers Shall We Store?

- An index is to provide "directions" – i.e., pointers to data.
  - An index entry is much like a street sign.
- Thus, our objective is to store a collection of pointers.
  - Then, we use keys to label these pointers to follow.
- Let's start with– having a pointer to **every key value**.
  - This is called a **dense index** – discuss later.
  - Other choices: to **every block**, or …



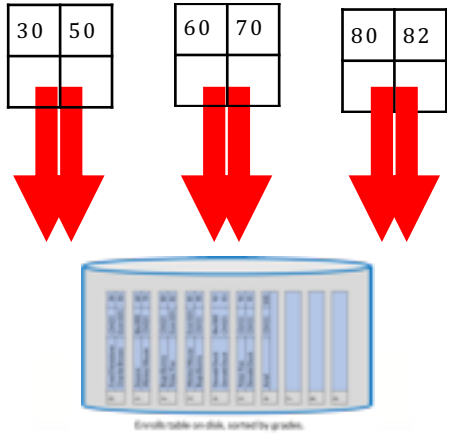
How Do We Store These Pointers? How to Label Them?
Our "Signs": Index Blocks
Case 1 – for Pointer to a Specific Key

- Just label the pointer with the key value: i.e., (K, P).
- Store n pointers and their n keys in a disk block.



Structure of a leaf index block

For $n = 2$:



- The "sign" says: To find tuple of key K_i, follow pointer P_i.
- Typical in a **tree index** (e.g., ISAM, B+ tree) as **leaf node** structure.
- How large is n? As many as a block can fit.

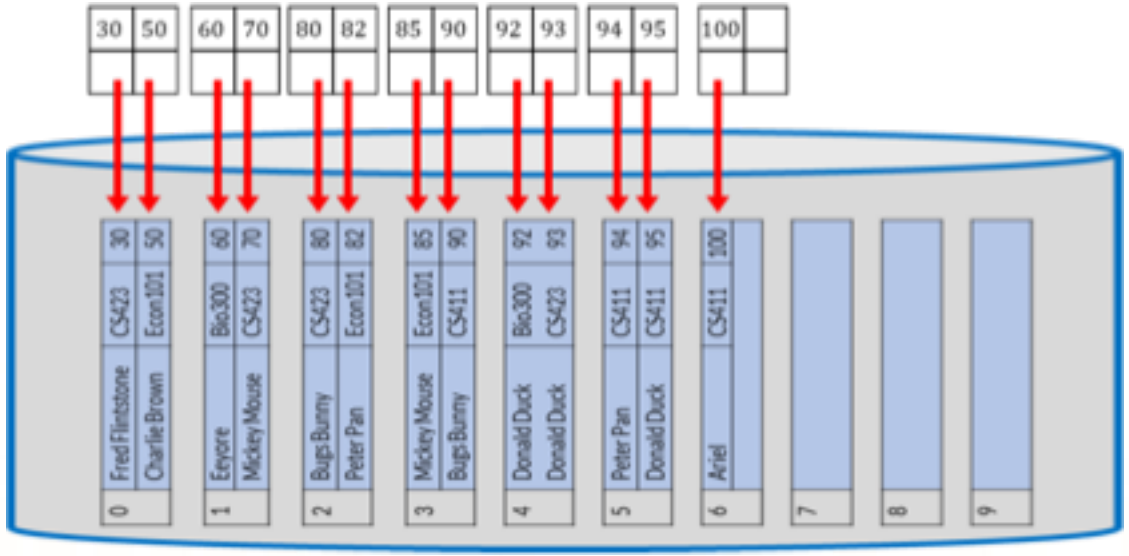
Stlmag 2014. Highway food signs [Online image]. Retrieved from https://www.stlmag.com/dining/Ask-George-Why-arent-there-more-independent-restaurants-listed-on-those-interstate-highway-Food-signs/

Pixabay 2018. The way home [Online image]. Retrieved from https://pixabay.com/en/at-the-court-of-sky-blue-sign-3218573/

Mydoordesign 2018. Directional sign [Online image]. Retrieved from https://www.mydoorsign.com/Directory/Florence-Directional-Sign/SKU-SE-3528.aspx

Create One Level of Index Blocks

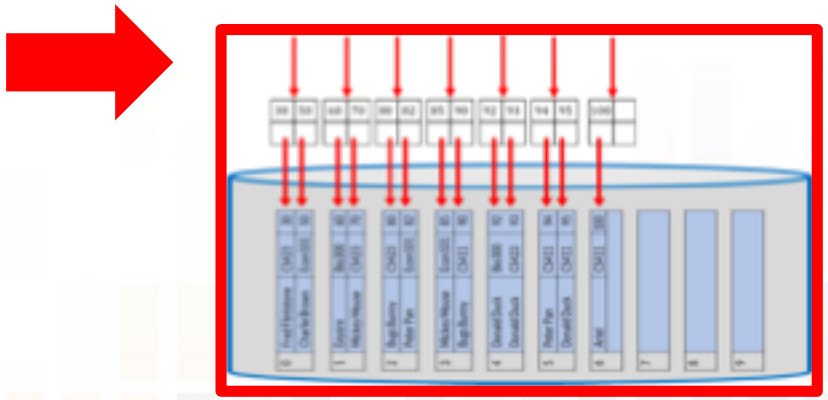
- This is useful, but not very useful (yet).



One level of an ISAM index

- Can we create one more level, to index these index nodes?

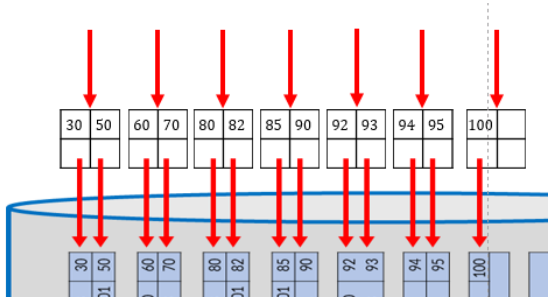
Again, What Pointers Do We Need?



How Do We Store These Pointers? How to Label Them?

Our "Signs": Index Blocks

Case 2 – for Pointer to a Set of Keys

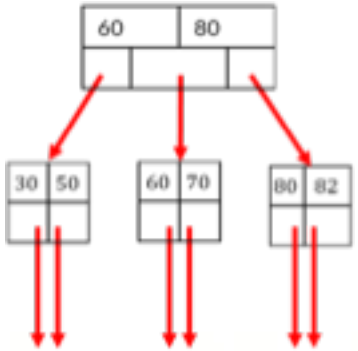


Example ISAM index

- Each pointer leads to a set of keys.
- We will label each pointer by a range $[K_i, K_{i+1})$.
- An index block contains n keys and $n + 1$ pointers.

Structure of a non-leaf index block

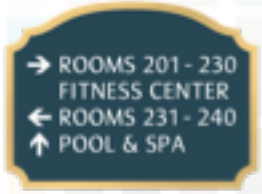
For $n = 2$:



Stlmag 2014. Highway food signs [Online image]. Retrieved from https://www.stlmag.com/dining/Ask-George-Why-arent-there-more-independent-restaurants-listed-on-those-interstate-highway-food-signs/

- The "sign" says: To find tuples of key in $[K_i, K_{i+1})$, follow pointer P_i.
- Typical in a **tree index** (e.g., ISAM, B+ tree) as **non-leaf node** structure.
- The number of pointers is the **fanout** $F = n + 1$.

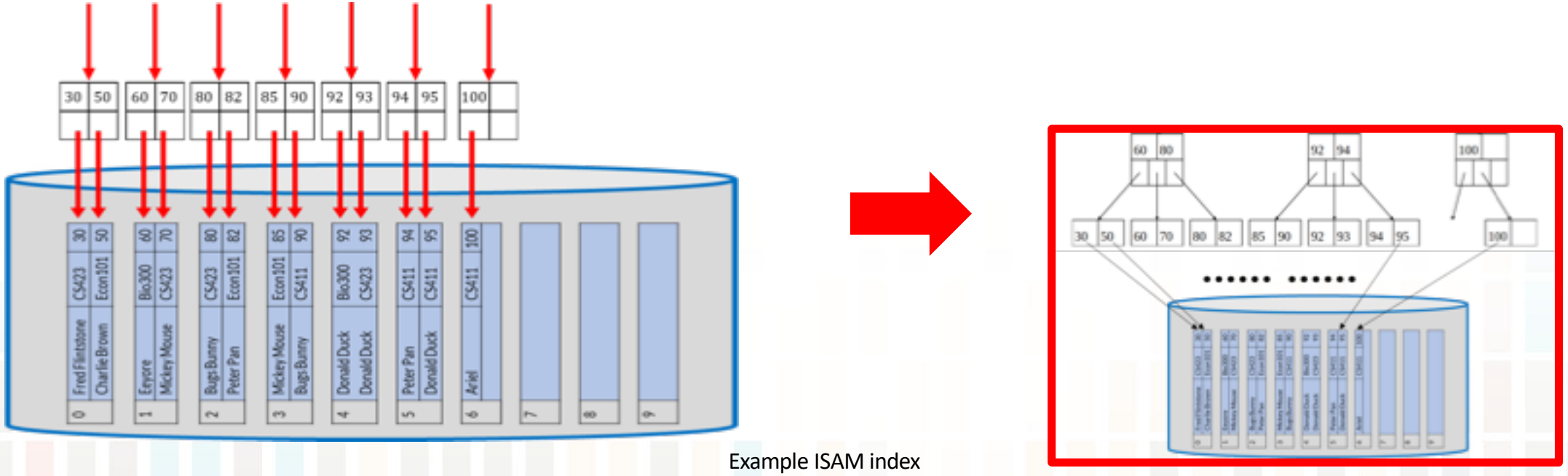
Pixabay 2018. The way home [Online image]. Retrieved from https://pixabay.com/en/at-the-court-of-sky-blue-sign-3238573/



Mydoordesign 2018. Directional sign [Online image]. Retrieved from https://www.mydoorsign.com/Directory/Florence-Directional-Sign/SKU-SE-3528.aspx

Add Next Level of Index Blocks

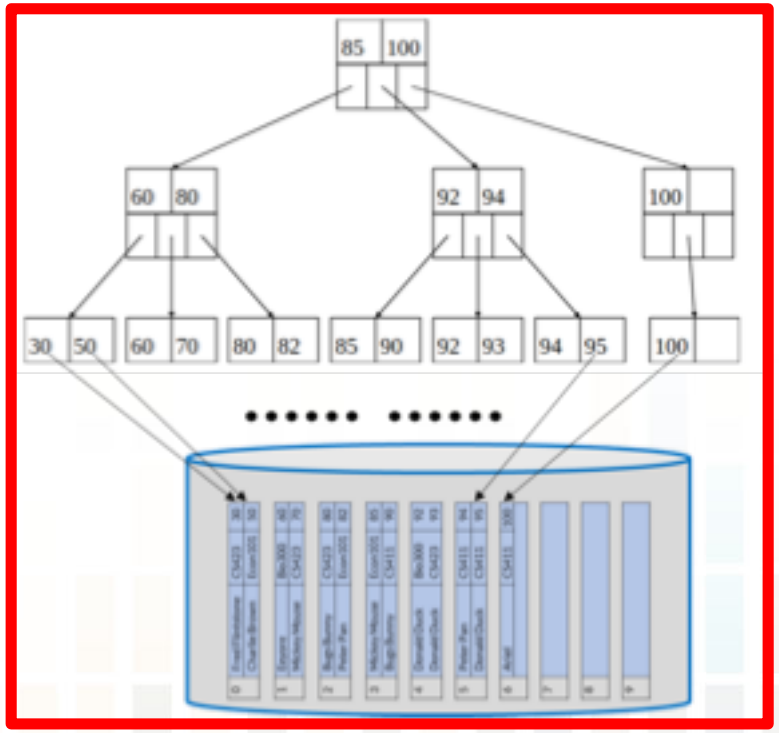
- Add index blocks to point to the last level.



Example ISAM index

ISAM – Indexed Sequential Access Method

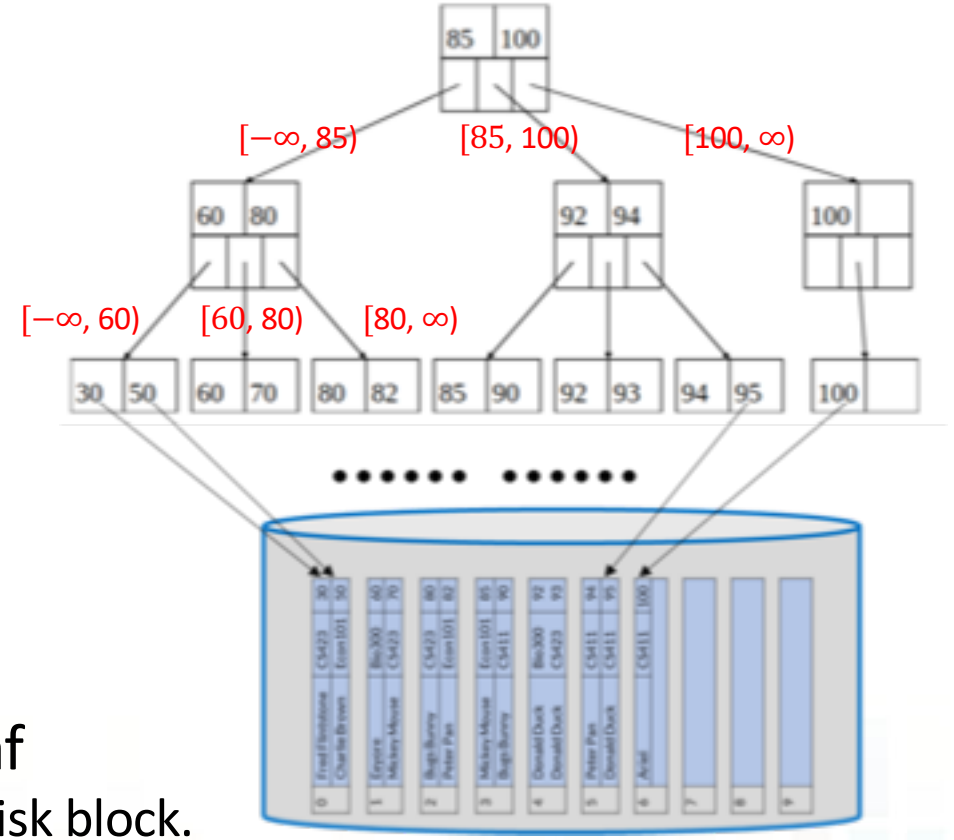
- Add one level of index blocks to point to data blocks (**leaf nodes**).
- Add one level of index blocks to point to the previous level (**non-leaf nodes**).
- …. ….
- Until there is only one index block (**root**).



Example ISAM index

ISAM: Lookup

- **Lookup(key) → pointer to tuple**
  - Start at the root.
  - **Follow pointers by the ranges they represent.**
  - Until reach a leaf node.
  - Check if key is there.
- *Q1: Find students who scored 80.*

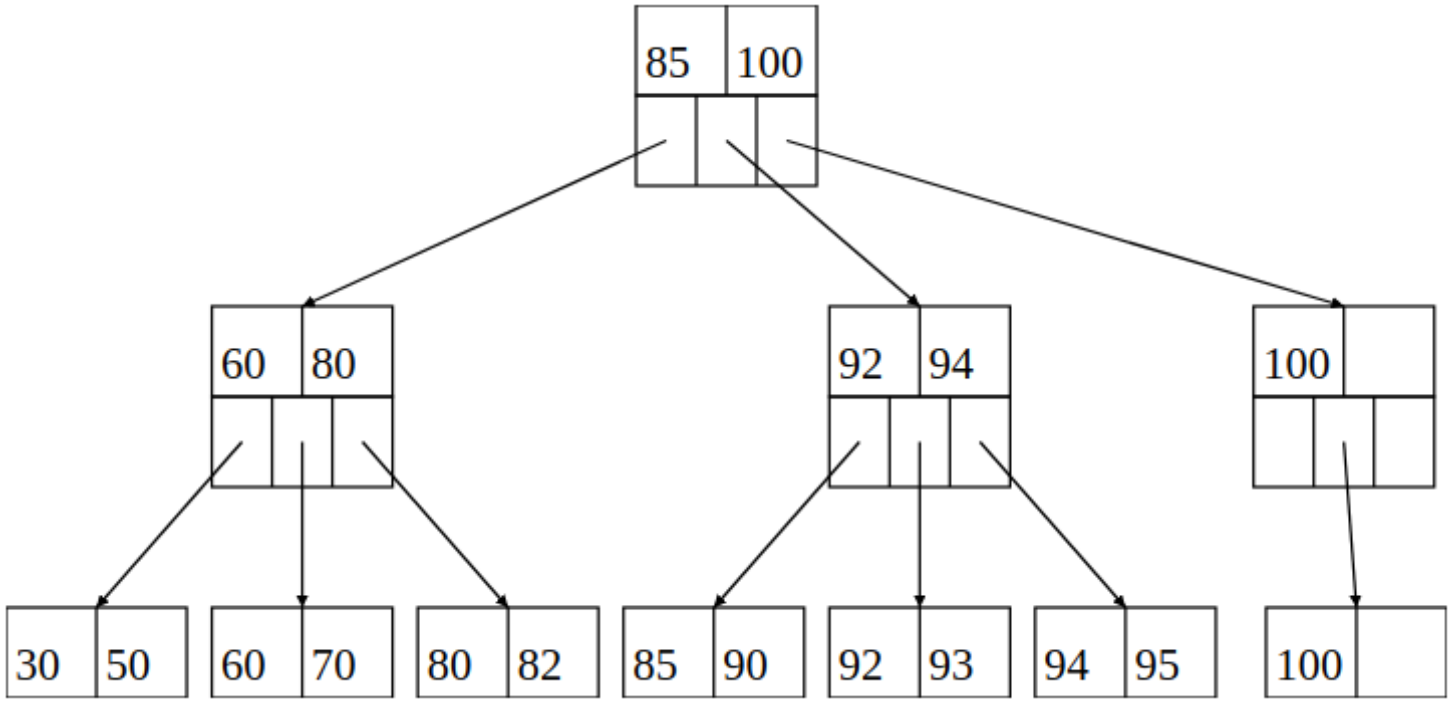


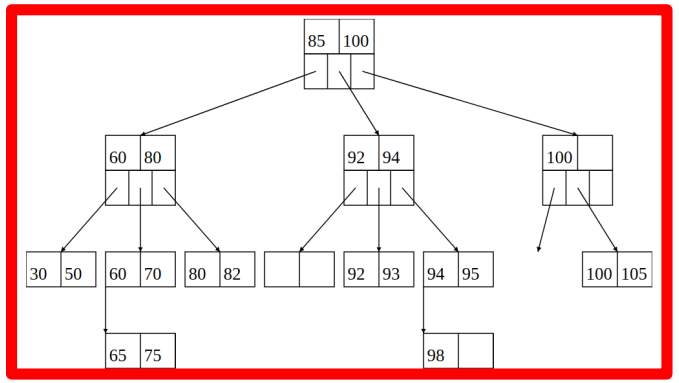
Search 80 in an example ISAM index

- Cost: **Number of pointer hops** to reach a leaf
  - Each pointer hop is a random access to fetch a disk block.
  - Number of pointer hops = height of tree h.
  - For indexing N tuples, with fanout $N \leq F^h$, or $h = \lceil Log_F(N) \rceil$.
  - E.g., $N = 1000000$, $F = 341$ (Why? Later!): cost = $h = \lceil Log_{341}(1000000) \rceil = 3$.

ISAM Insertion/Deletion Examples

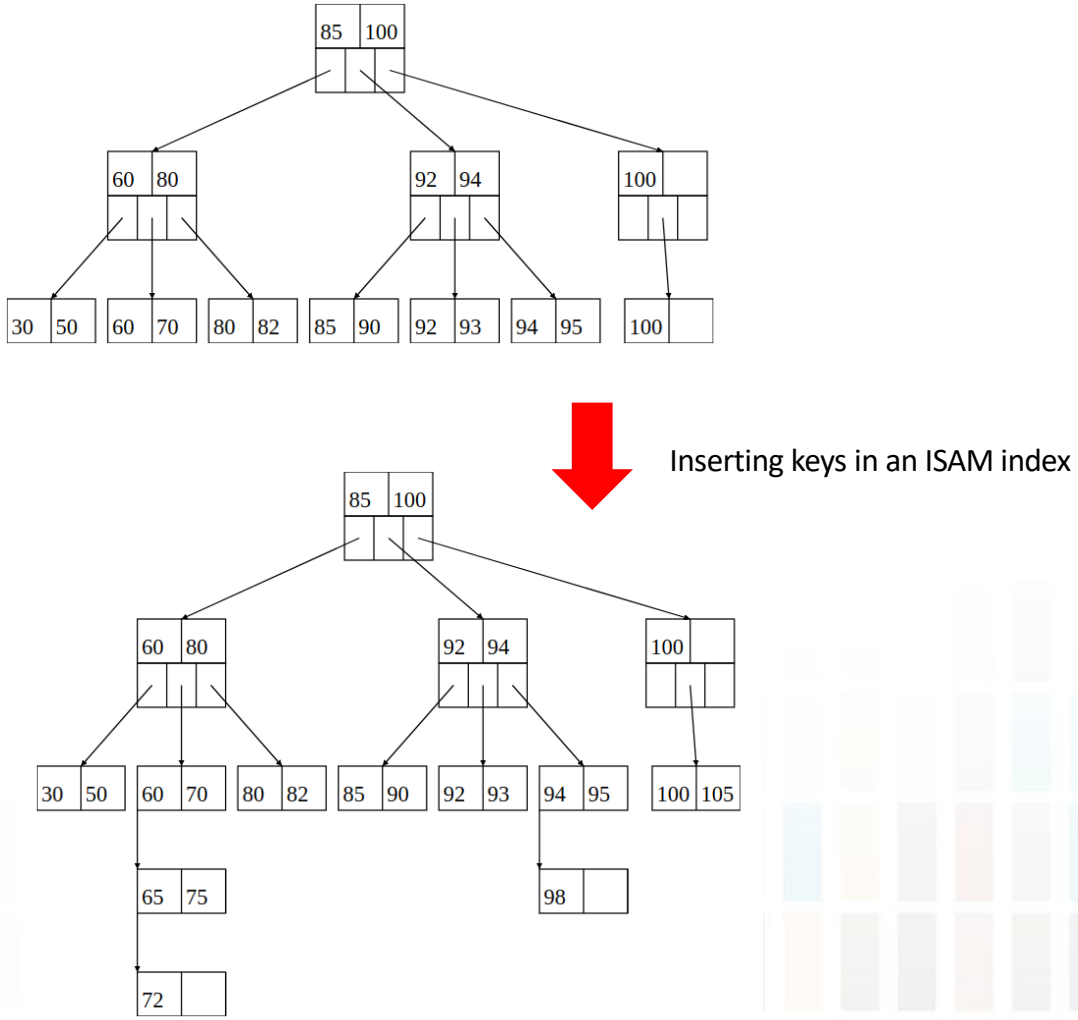
- Insert 105, 65, 98, 75, 72
- Delete 72, 85, 90





ISAM: How to Insert New Data?

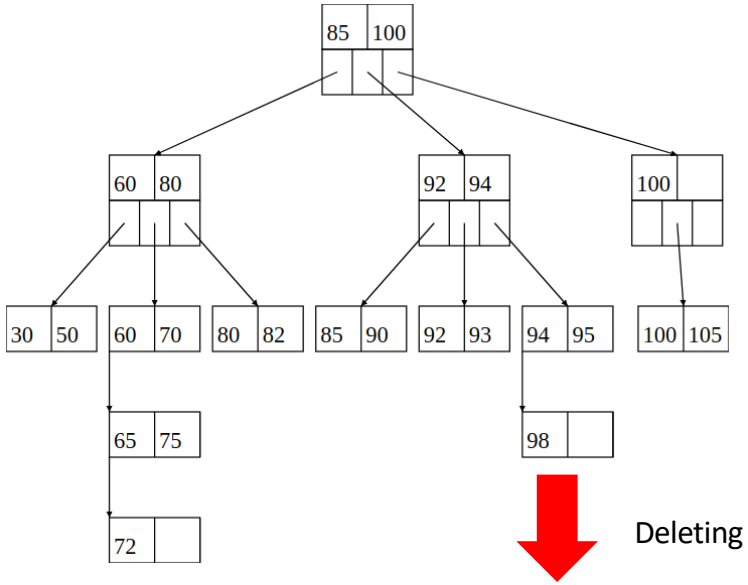
- Insert (105)
  - We are lucky to have a vacancy.
- Insert (65)
  - Use overflow blocks.
- Insert (98)
- Insert (75)
- Insert (72)



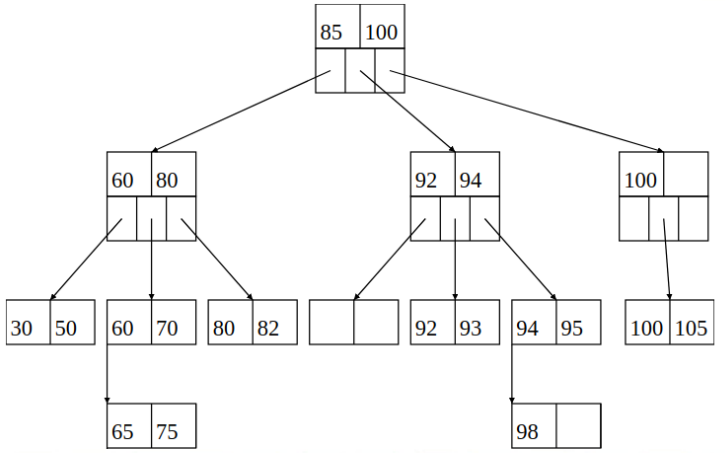
Inserting keys in an ISAM index

ISAM: How to Delete Data?

- Delete (72)
- Delete (85)
- Delete (90)

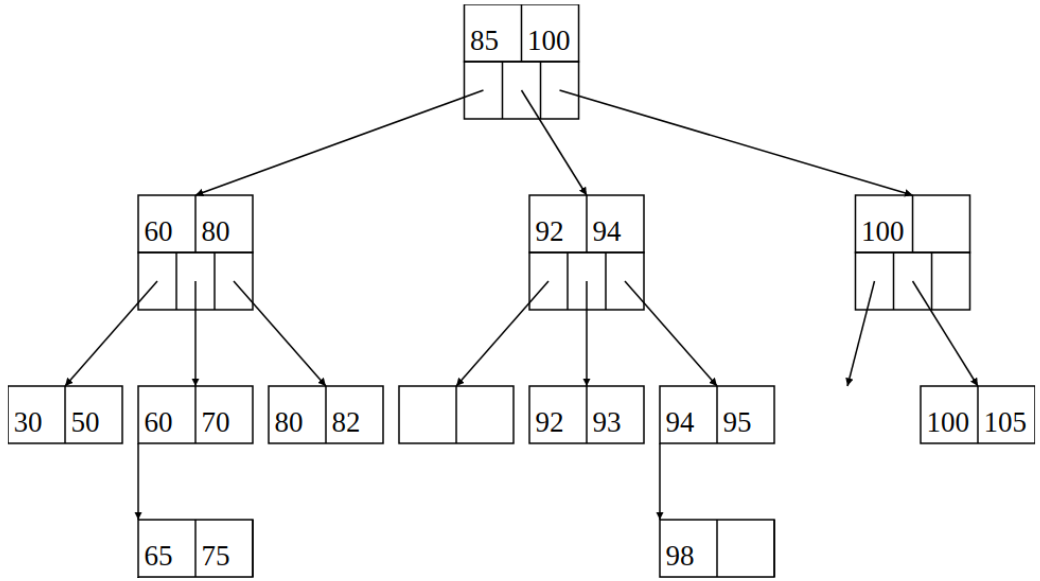


Deleting keys in an ISAM index



How Do You Like This Index?

- How does it satisfy our desired properties for response times?
- How about disk space utilization?



Food for Thought

We discussed ISAM for sorted files.
Can it index unsorted blocks?
How?

Sorted

0	Fred Flintstone	CS423	30
	Charlie Brown	Econ101	50
1	Eeyore	Bio300	60
	Mickey Mouse	CS423	70
2	Bugs Bunny	CS423	80
	Peter Pan	Econ101	82
3	Mickey Mouse	Econ101	85
	Bugs Bunny	CS411	90
4	Donald Duck	Bio300	92
	Donald Duck	CS423	93
5	Peter Pan	CS411	94
	Donald Duck	CS411	95
6	Ariel	CS411	100
7			
8			
9			

Unsorted

0	Bugs Bunny	CS411	90
	Donald Duck	Bio300	92
1	Donald Duck	CS423	93
	Donald Duck	CS411	95
2	Bugs Bunny	CS423	80
	Mickey Mouse	CS423	70
3	Peter Pan	CS411	94
	Charlie Brown	Econ101	50
4	Peter Pan	Econ101	82
	Eeyore	Bio300	60
5	Mickey Mouse	Econ101	85
	Ariel	CS411	100
6	Fred Flintstone	CS423	30
7			
8			
9			

Our example Enrolls table– sorted vs. unsorted.

B+ Tree: Structure and Capacity

Accessing and Indexing Data

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Kevin C.C. Chang, Professor

Computer Science @ Illinois

Learning Objectives

By the end of this video, you will be able to:

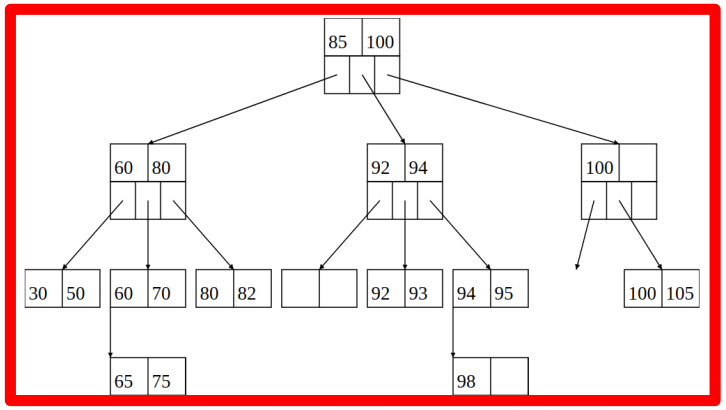
- Explain why ISAM falls short for indexing dynamic data.
- Describe the node structure and capacity of a B+ tree.
- Estimate how many tuples a B+ tree can index.
- Identify how B+ tree labels each pointer for what it leads to.
- **Salute an important data/CS contribution from UIUC alum.**

Watch Out:
What Does "**B**" Stand for in **B**-Tree?

- B+ tree is based on B-tree.
- For our discussion now, consider B-tree and B+ tree as the same.

ISAM Falls Short for (Dynamic) Databases

- Limitations: When database changes, tree can become quite “uneven”
  - With long and short paths at different parts of the tree.
    - Search time is fast or slow– unpredictable.
  - Nodes can become full or empty.
    - Full: Insertion requires overflow.
    - Empty: Waste disk storage space.



- What’s missing?

  Ability to **maintain** effectiveness under **dynamic changes:**
  - **Balance** of tree height.
  - **Utilization** of nodes (disk blocks).

B-Tree: Indexing Dynamic Data by Self Balancing

- A self-balancing tree data structure for indexing dynamic data.
- One of the most popular data structure in computer science.

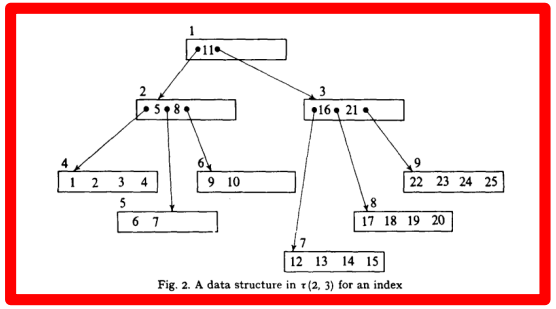
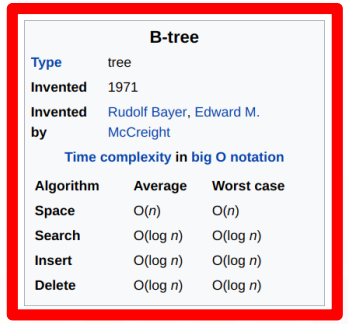


Fig. 2. A data structure in τ(2, 3) for an index

B-tree summary. B-tree [Online image]. Retrieved from https://en.wikipedia.org/wiki/B-tree

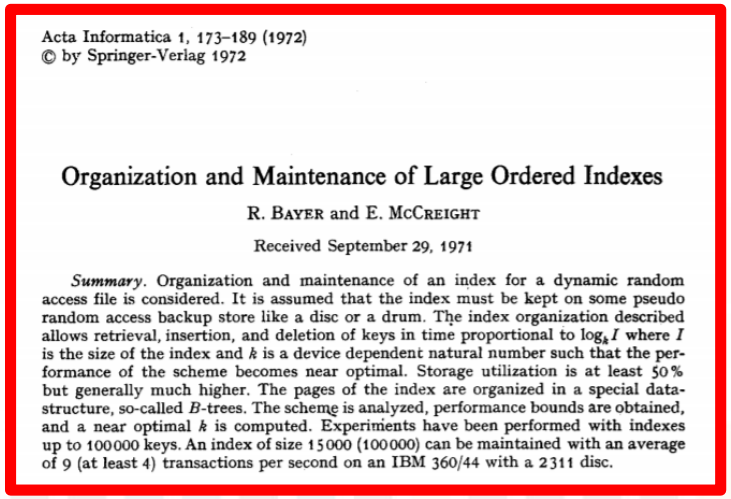


Example B-tree. Fig. 2 in Rudolf Bayer, Edward M. McCreight: Organization and Maintenance of Large Ordered Indices. Acta Informatica 1: 173-189(1972)

- Support update effectively: Maintain **balanced** structure
  - All root-to-leaf paths are of the same length.
- Allow uniform accesses in logarithmic time.
  - $O(Log\ (n))$ for accessing a dataset of n items.
- Optimized for large block-oriented access of data.
  - Good for accessing data in external memory.
- For our purpose, we discuss B+ tree.
  - B+ tree is a small variation/enhancement of the original B-tree.

Behind the Scene:
A UIUC (Alumnus) Contribution!

- B-tree was invented by Rudolf Bayer and Edward M. McCreight.
- UIUC alumnus! Rudolf Bayer studied Mathematics in Munich and at the University of Illinois, where he received his Ph.D. in 1966. After working at Boeing Research Labs he became an Associate Professor at Purdue University. He is a Professor of Informatics at the Technische Universität München since 1972 and … …



Acta Informatica 1, 173–189 (1972)
© by Springer-Verlag 1972

Organization and Maintenance of Large Ordered Indexes

R. BAYER and E. MCCREIGHT

Received September 29, 1971

Summary. Organization and maintenance of an index for a dynamic random access file is considered. It is assumed that the index must be kept on some pseudo random access backup store like a disc or a drum. The index organization described allows retrieval, insertion, and deletion of keys in time proportional to $\log_k I$ where I is the size of the index and k is a device dependent natural number such that the performance of the scheme becomes near optimal. Storage utilization is at least 50% but generally much higher. The pages of the index are organized in a special data-structure, so-called B-trees. The scheme is analyzed, performance bounds are obtained, and a near optimal k is computed. Experiments have been performed with indexes up to 100000 keys. An index of size 15000 (100000) can be maintained with an average of 9 (at least 4) transactions per second on an IBM 360/44 with a 2311 disc.

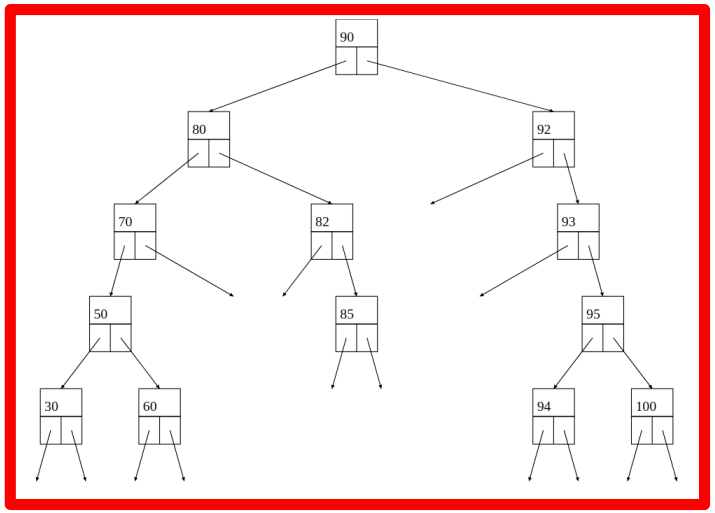
Rudolf Bayer, Edward M. McCreight: Organization and Maintenance of Large Ordered Indices. Acta Informatica 1: 173-189(1972)

Title page of the B-tree paper.

Whatever “**B**” Is–
It’s probably not “Binary”!

- Let’s try build a binary search tree for our example table.

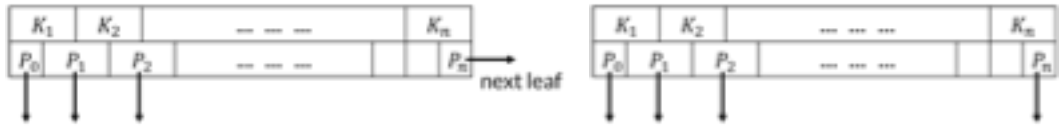
```
bst = BST()

bst.insert(90, ['Bugs Bunny', 'CS411', 90])
bst.insert(92, ['Donald Duck', 'Bio300', 92])
bst.insert(93, ['Donald Duck', 'CS423', 93])
bst.insert(95, ['Donald Duck', 'CS411', 95])
bst.insert(80, ['Bugs Bunny', 'CS423', 80])
bst.insert(70, ['Mickey Mouse', 'CS423', 70])
bst.insert(94, ['Peter Pan', 'CS411', 94])
bst.insert(50, ['Charlie Brown', 'Econ101', 50])
bst.insert(82, ['Peter Pan', 'Econ101', 82])
bst.insert(60, ['Eeyore', 'Bio300', 60])
bst.insert(85, ['Mickey Mouse', 'Econ101', 85])
bst.insert(100, ['Ariel', 'CS411', 100])
bst.insert(30, ['Fred Flintstone', 'CS423', 30])
```



- What do you think of this binary search tree?
- Does it address the issues of ISAM?

B+ Tree Node Structure: Like ISAM, But …
Not too Empty (and Not Too Full)

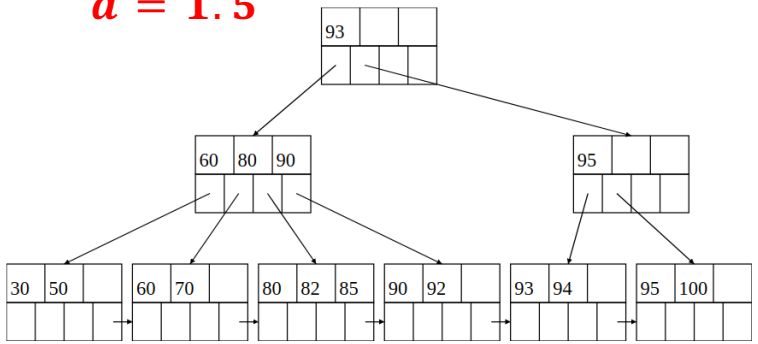
- Parameter d, the **degree** (or **order**).
  - $d = 1, 1.5, 2, 2.5, \ldots$
- Each node has n keys and $n + 1$ pointers.
- Leaf node: Internal node:



Degree (d)	(Internal) Min Keys	(Leaf) Min Keys	Max Keys (n)
1	1	1	2
1.5	1	2	3
2	2	2	4
2.5	2	3	5
3	3	3	6
3.5	3	4	7
4	4	4	8
4.5	4	5	9
5	5	5	10

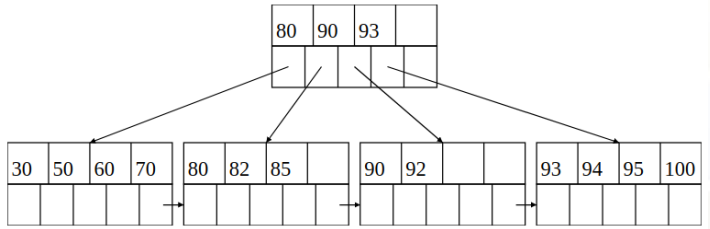
- Max. number of keys: $n = 2d$
- Min. number of keys: $n \approx d$, i.e., 50% of the max.
  - (Case 1) Leaf node: $n = \lceil d \rceil$ (thus, can be slightly larger than 50%).
  - (Case 2) Internal node: $n = \lfloor d \rfloor$ (thus, can be slightly smaller than 50%).
  - (Except) Root: 1

$\boldsymbol{d = 1.5}$



Exampel B+ Tree

$\boldsymbol{d = 2}$



Exampel B+ Tree

B+ Tree Node Capacity and Fanout

- How large is d ?
- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 byes
- $2d \times 4 \; + \; (2d + 1) \times 8 \; <= \; 4096$
- d = 170
- $n \; = \; 2d$ = 340
- Fanout (max) $F = n + 1$ = 341

How Large Data Can B+ Tree Handle?

- Typical degree: 100. Typical fill-factor: 67%.
  - Average fanout = 133
- Can index large data with a few levels:

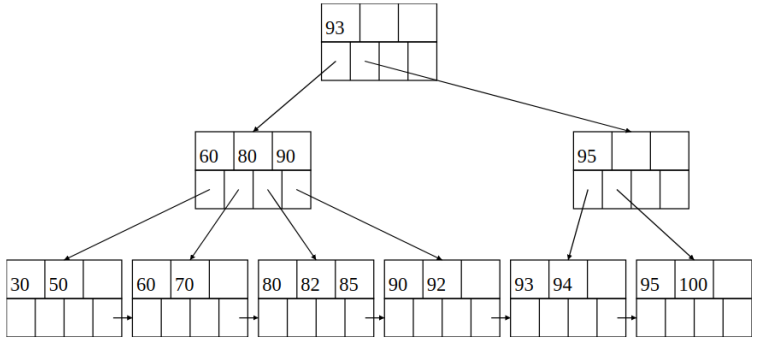
Height h	Number of Tuples N
1	133
2	17,689
3	2,352,637
4	312,900,721
5	41,615,795,893
6	5,534,900,853,769

- Can hold top levels in memory:

Hight	Total Blocks	Size (KB) for 4KB Blocks
1	1	4
2	134	536
3	17,823	71,292
4	2,370,460	9,481,840

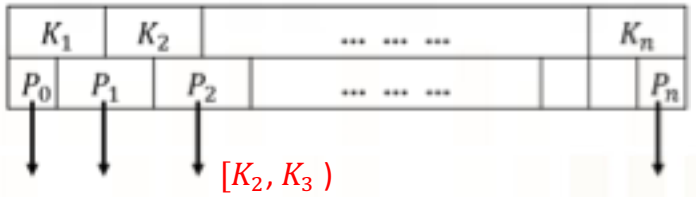
B+ Tree Pointers:
How to Label Pointers (for What They Lead to)?

- Pointer labeling: Same as in ISAM
- (Case 1) Leaf node: P_i points to tuples with key K_{i+1}
  - With last pointer K_n pointing to next leaf block.
- (Case 2) Internal node: P_i points to index blocks of keys $[K_i, K_{i+1})$.



Exampel B+ Tree with $d = 1.5$

B+ tree leaf node structure



B+ tree internal node structure

Food for Thought

What do you think the "B" stand for in B-tree? It was never explained in the paper.

A. Binary
B. Balanced
C. Bayer
D. Boeing
E. All of the above
F. None of the above

B+ Tree: Operations

Accessing and Indexing Data

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Kevin C.C. Chang, Professor

Computer Science @ Illinois

Learning Objectives

By the end of this video, you will be able to:

- Perform lookup of data in a B+ tree.
- Describe the principle of local changes for maintaining a B+ tree.
- Perform insertion and deletion operations.
- Explain how B+ tree can handle dynamic data.

B+ Tree: Operations

- Lookup tuples by a search key.
  - **Point** lookup: grade = 80.
  - **Range** lookup: grade > 80.

- Making changes:
  - Insertion of a tuple.
  - Deletion of a tuple.
  - Update of a tuple = deletion then insertion.

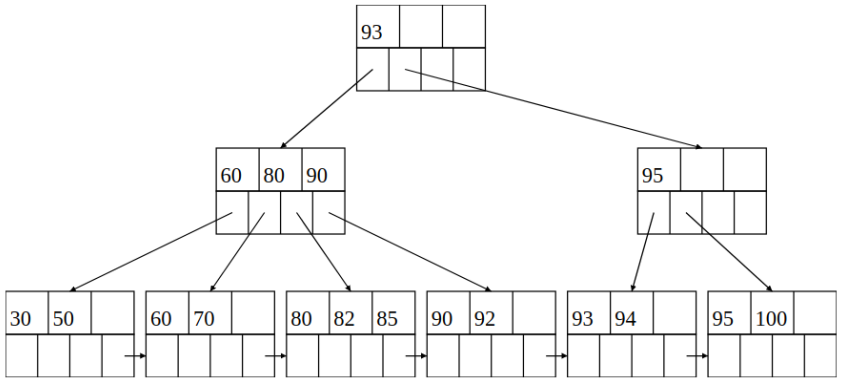
B+ Tree: Lookup

- Point queries: … WHERE grade = 80

  Lookup(key) → pointer to tuple

  Same as in ISAM:

  - Start at the root.
  - **Follow pointers by the ranges they represent.**
  - Until reach a leaf node.
  - Check if key is there.



Example B+ tree

- Range queries: … WHERE grade > 80

  Lookup($> key$) → pointers to tuples

  - Locate key or the closest value greater than key as in **Lookup(key)**
  - Then sequentially traverse following "next-leaf" pointers.

Maintaining a B+ Tree under Updates: The *Principle of Local Changes*

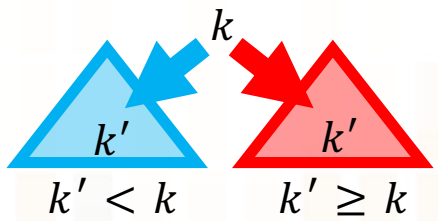
Principle: When a node becomes too full or too empty,
perform "local" changes to maintain the tree in a good shape.

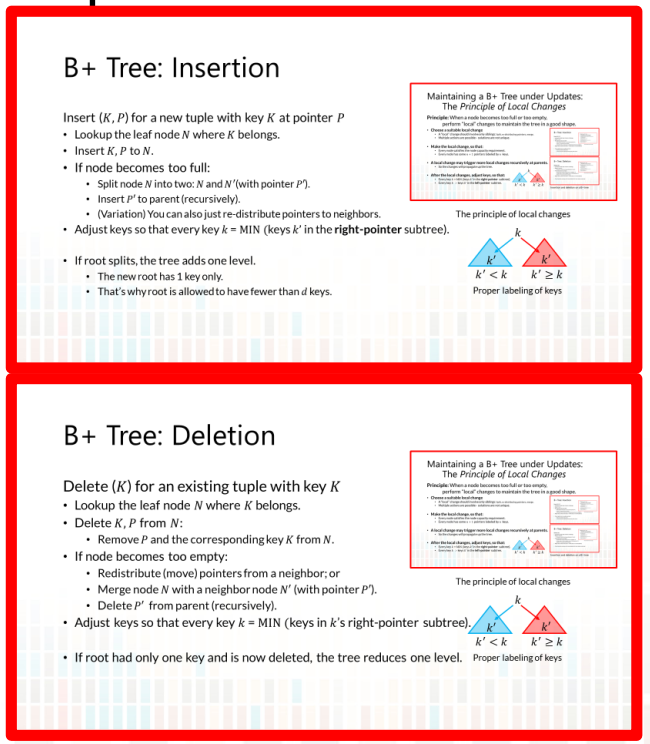
- **Choose a suitable local change**
  - A "local" change should involve only siblings: Split, re-distributing pointers, merge.
  - Multiple actions are possible– solutions are not unique.

- **Make the local change, so that:**
  - Every node satisfies the node capacity requirement.
  - Every node has some $n + 1$ pointers labeled by n keys.

- **A local change may trigger more local changes recursively at parents.**
  - So the changes will propagate up the tree.

- **After the local changes, adjust keys, so that:**
  - Every key k = MIN (keys k' in the **right-pointer** subtree).
  - Every key k > keys k' in the **left-pointer** subtree.

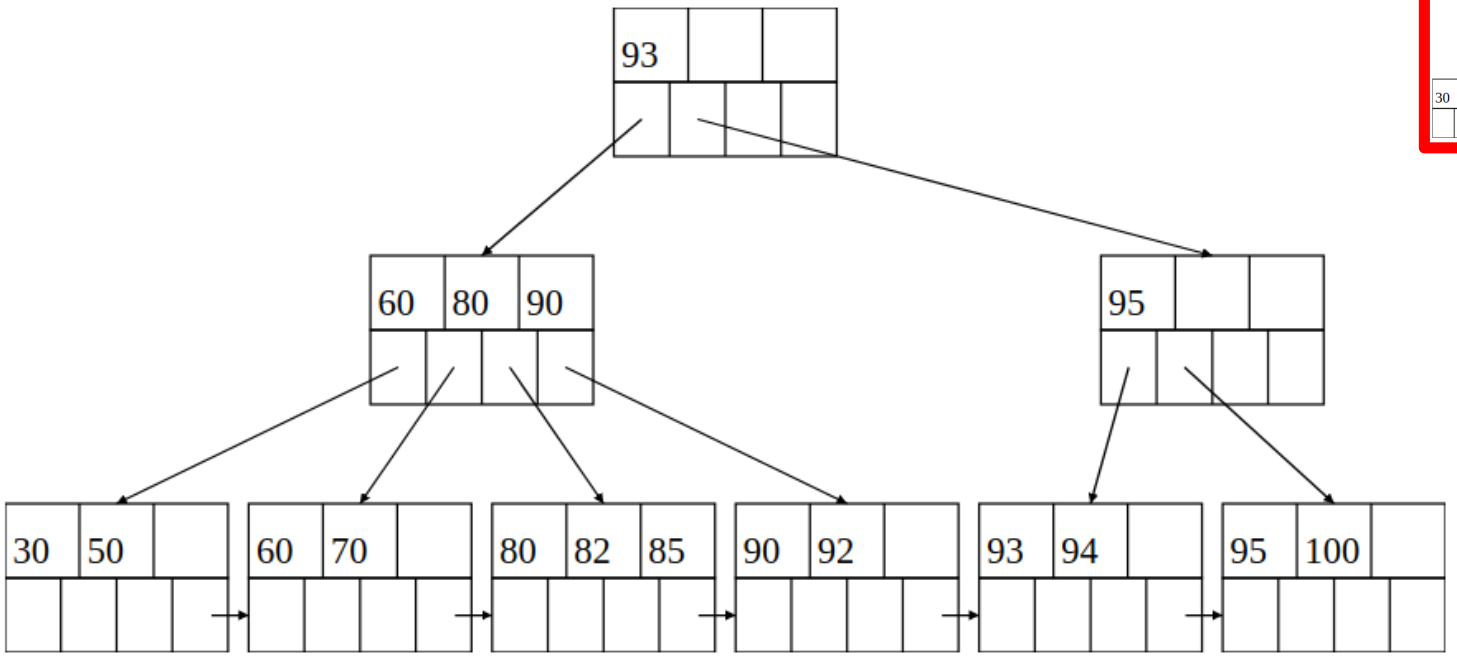


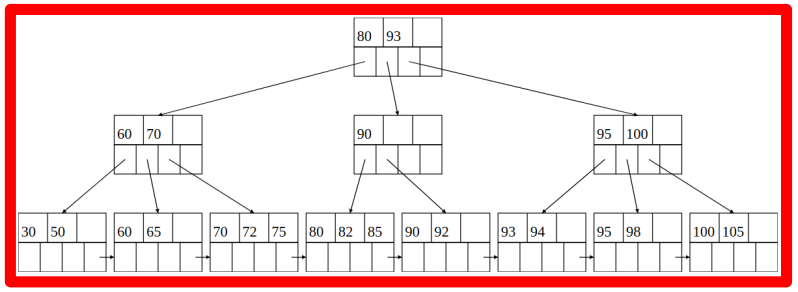


Insertion and deletion on a B+ tree

Insertion Examples

- Insert 105, 65, 98, 75, 72



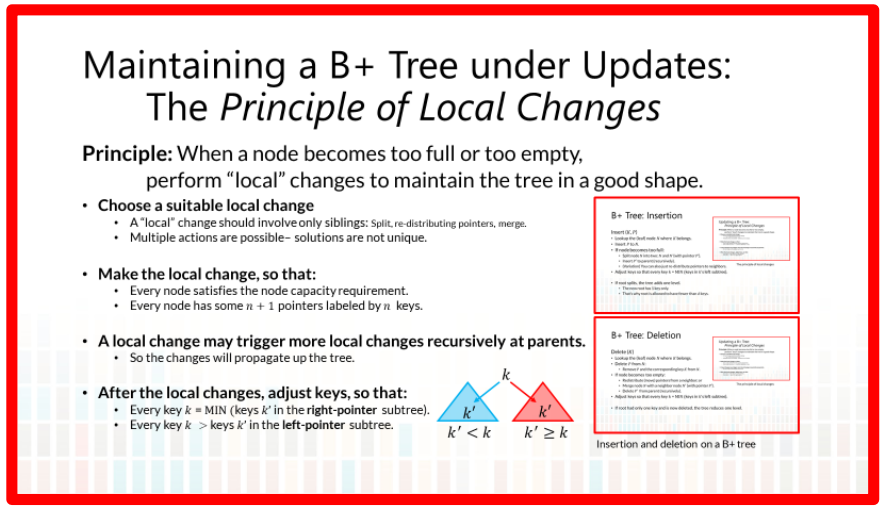


B+ Tree: Insertion

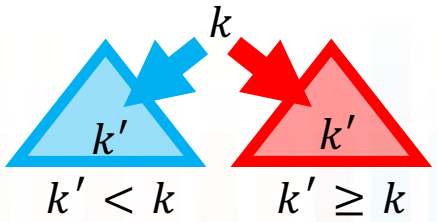
Insert (K, P) for a new tuple with key K at pointer P

- Lookup the leaf node N where K belongs.
- Insert K, P to N.
- If node becomes too full:
  - Split node N into two: N and N'(with pointer P').
  - Insert P' to parent (recursively).
  - (Variation) You can also just re-distribute pointers to siblings.
- Adjust keys so that every key k = MIN (keys k' in the **right-pointer** subtree).

- If root splits, the tree adds one level.
  - The new root has 1 key only.
  - That's why root is allowed to have fewer than d keys.



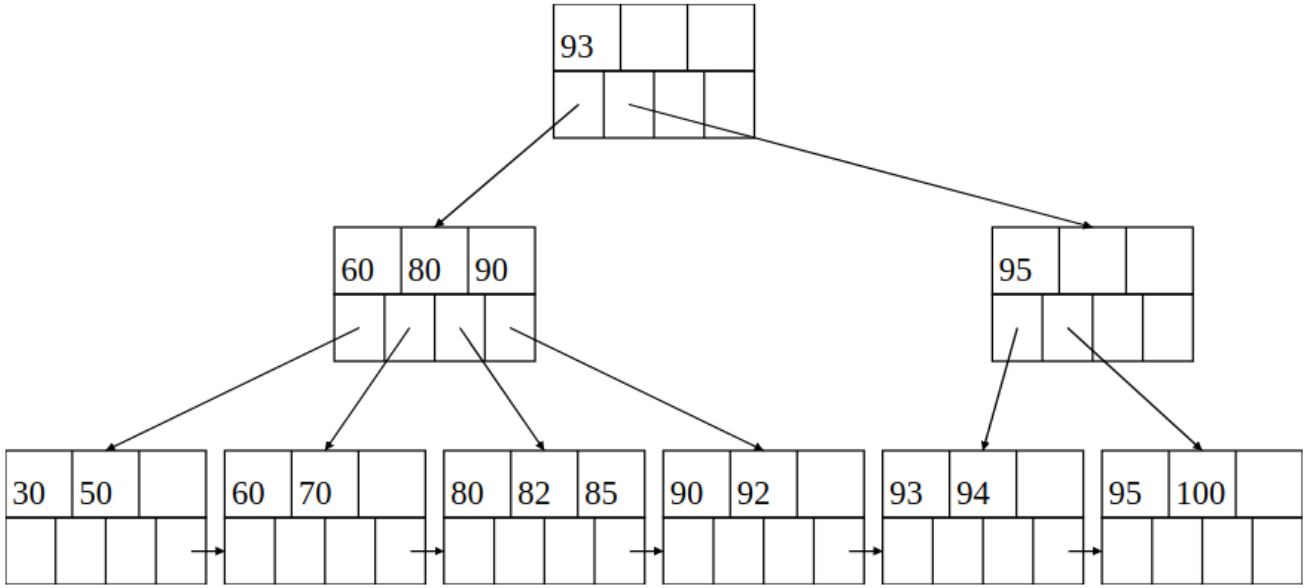
The principle of local changes

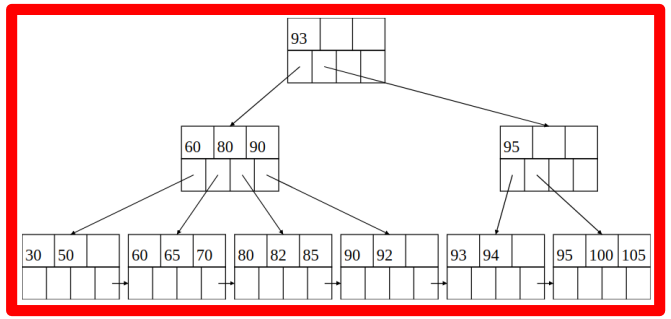


Proper labeling of keys

Insertion: Case 1 – Insert and Done!

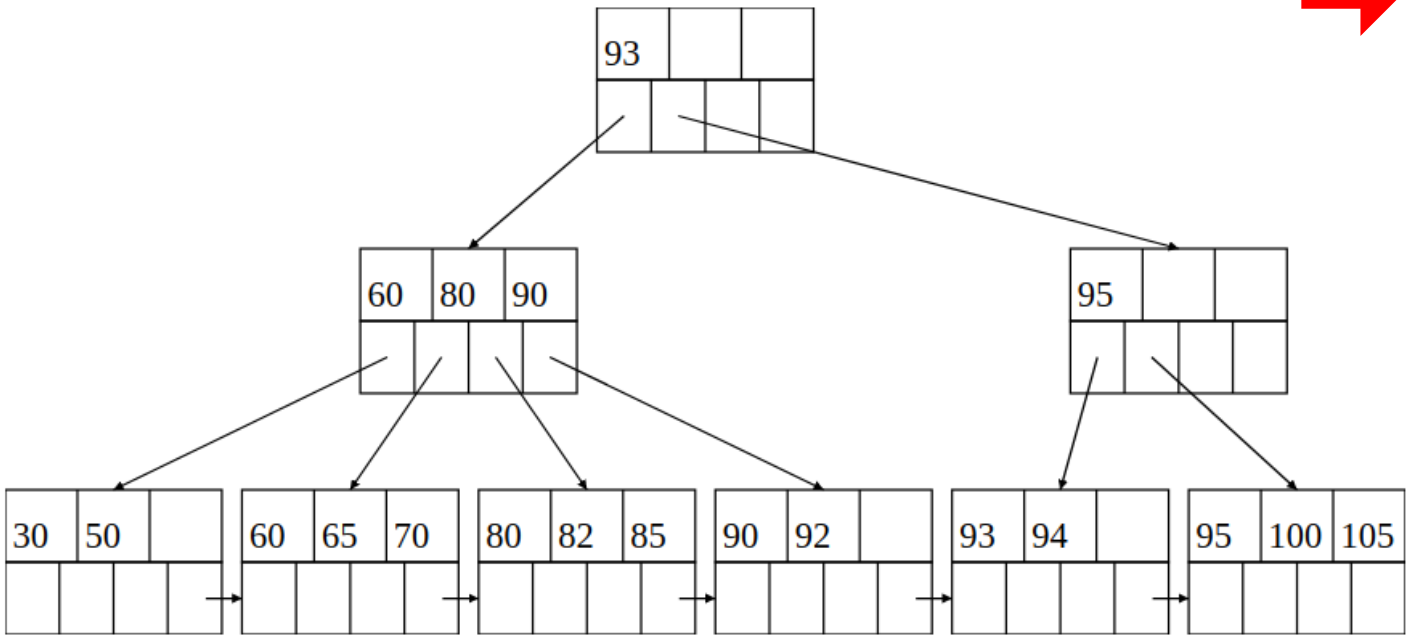
- Insert 105, 65

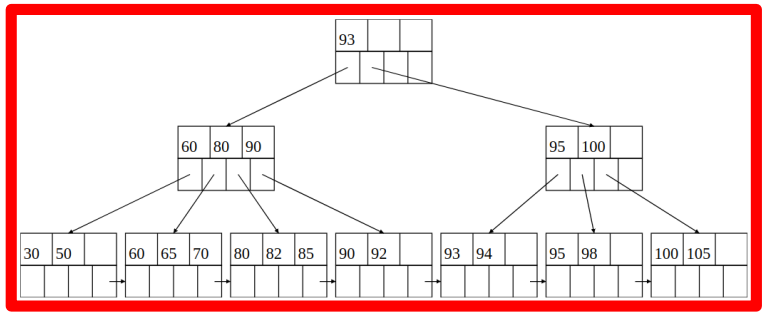




Insertion: Case 2a – Split (and Insert to Parent)

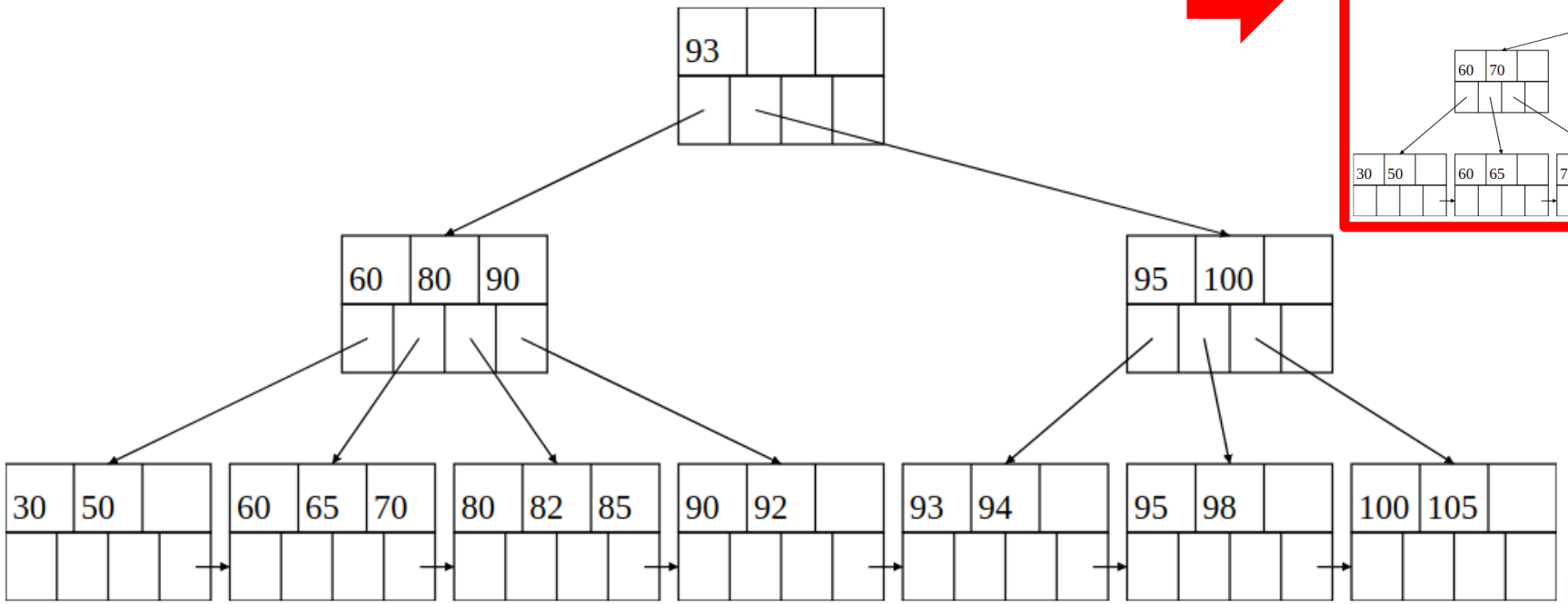
- Insert 98

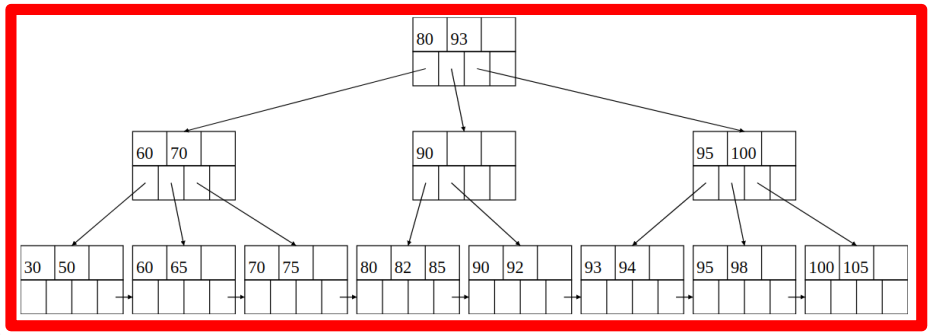




Insertion: Case 2b – More Splits

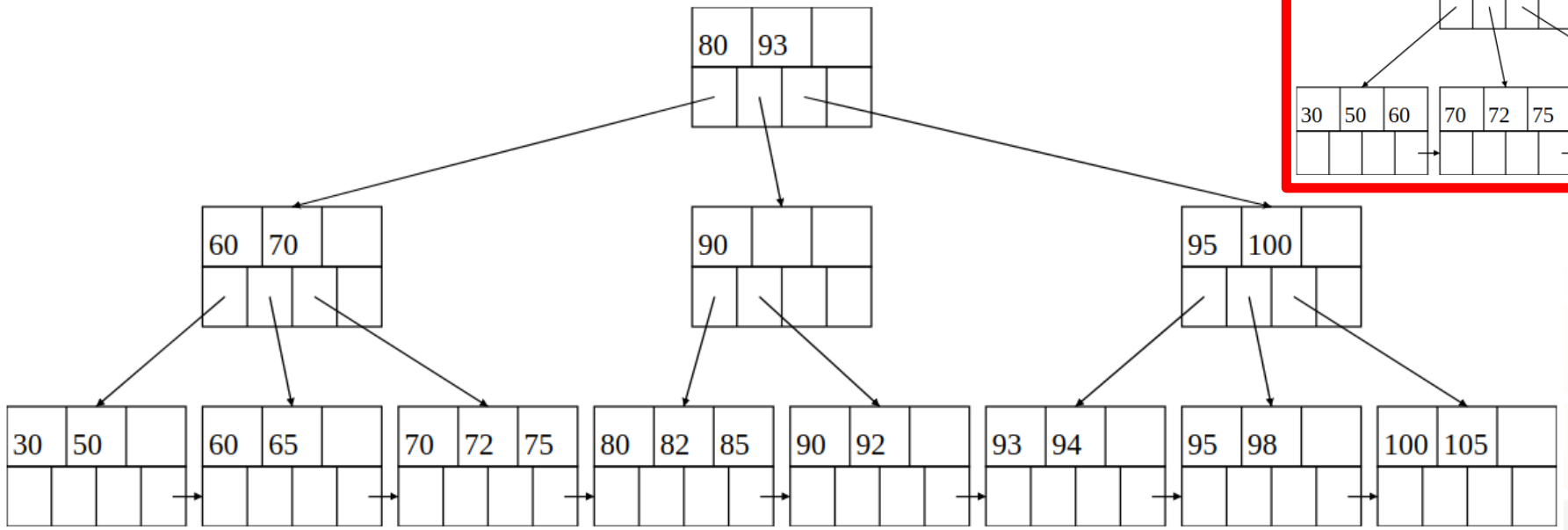
- Insert 75

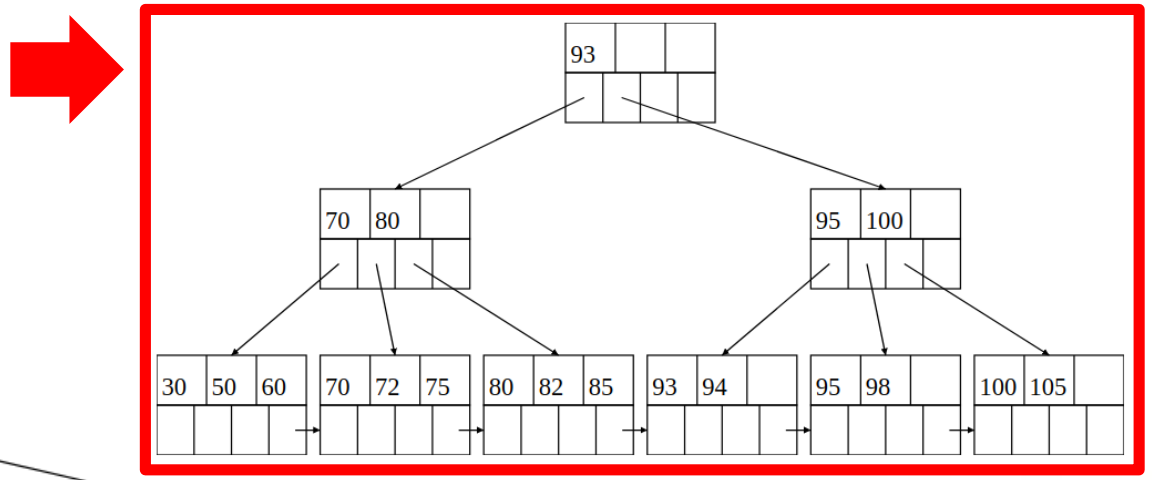




Deletion Examples

- Delete 90, 92, 65

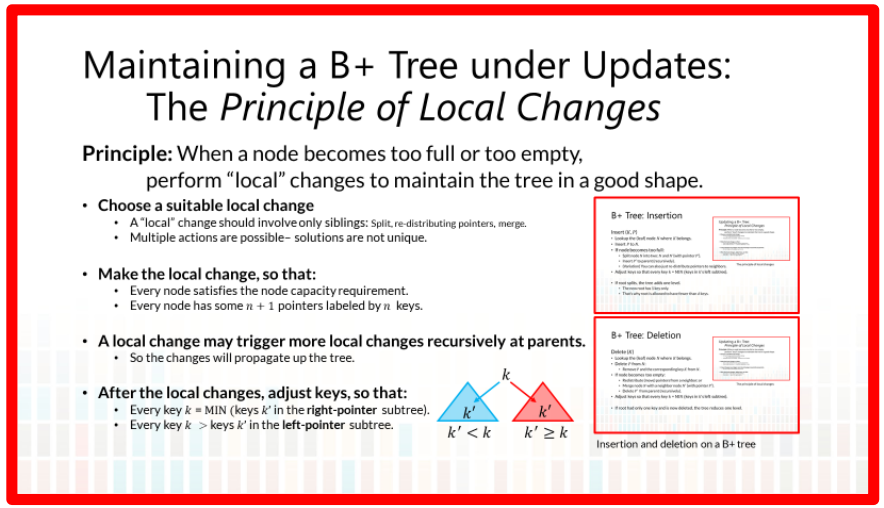




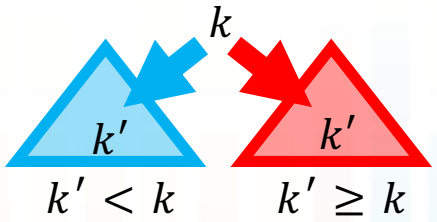
B+ Tree: Deletion

Delete (K) for an existing tuple with key K

- Lookup the leaf node N where K belongs.
- Delete K, P from N:
  - Remove P and the corresponding key K from N.
- If node becomes too empty:
  - Redistribute (move) pointers from a sibling; or
  - Merge node N with a sibling node N' (with pointer P').
  - Delete P' from parent (recursively).
- Adjust keys so that every key k = MIN (keys in k's right-pointer subtree).
- If root had only one key and is now deleted, the tree reduces one level.



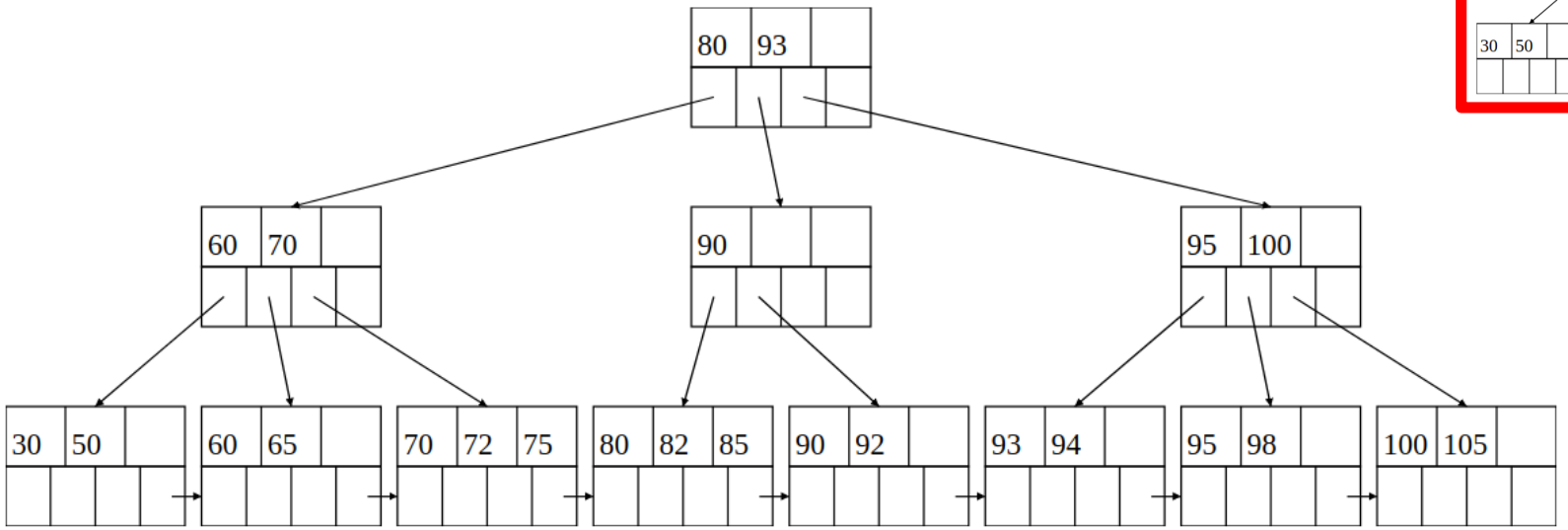
The principle of local changes

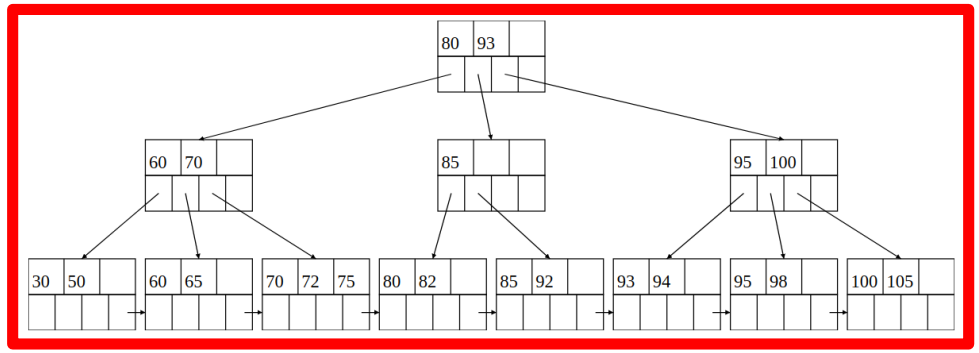


Proper labeling of keys

Deletion: Case 1 – Redistribute

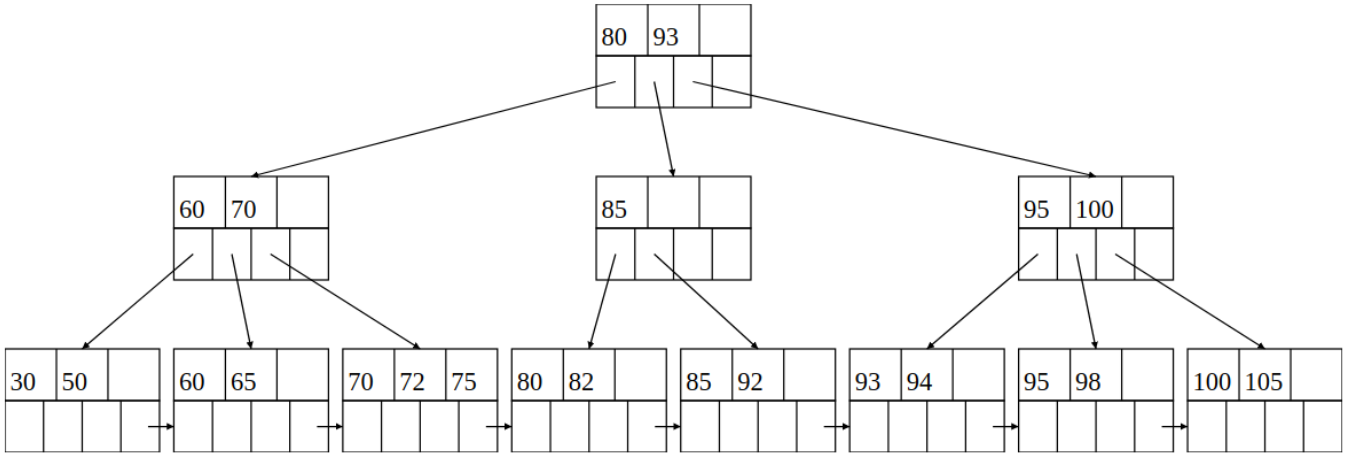
- Delete 90

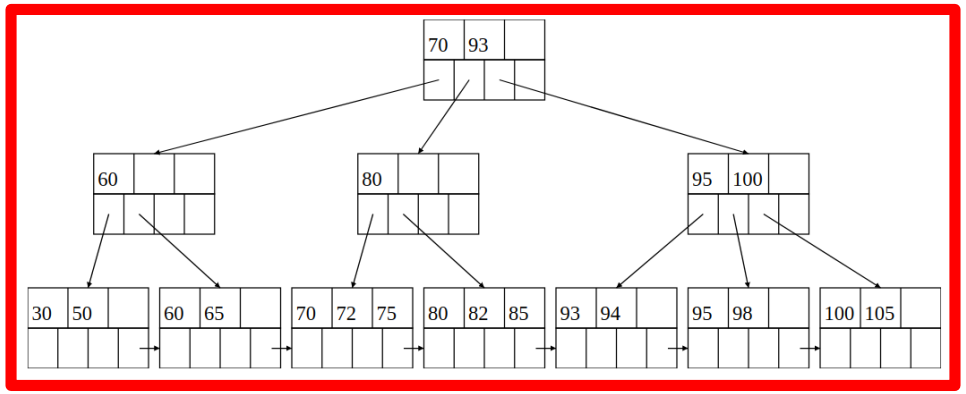




Deletion: Case 2a – Merge (and Delete from Parent)

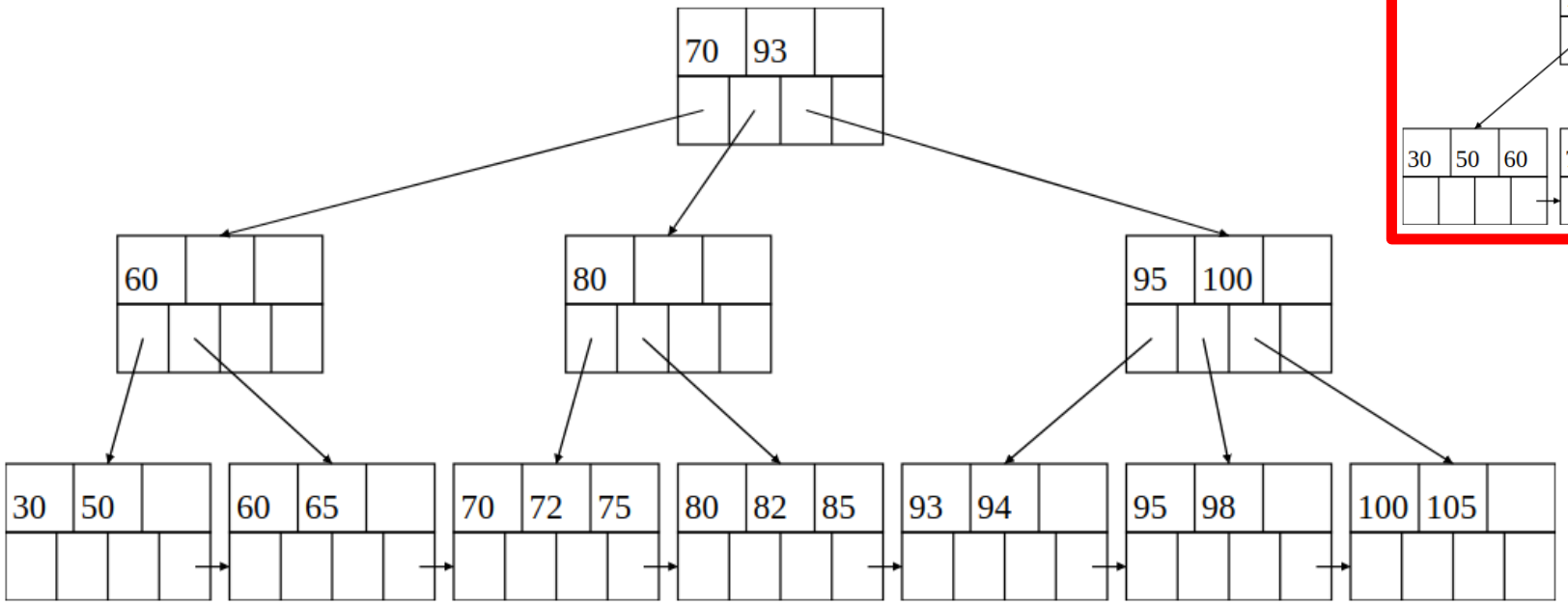
- Delete 92

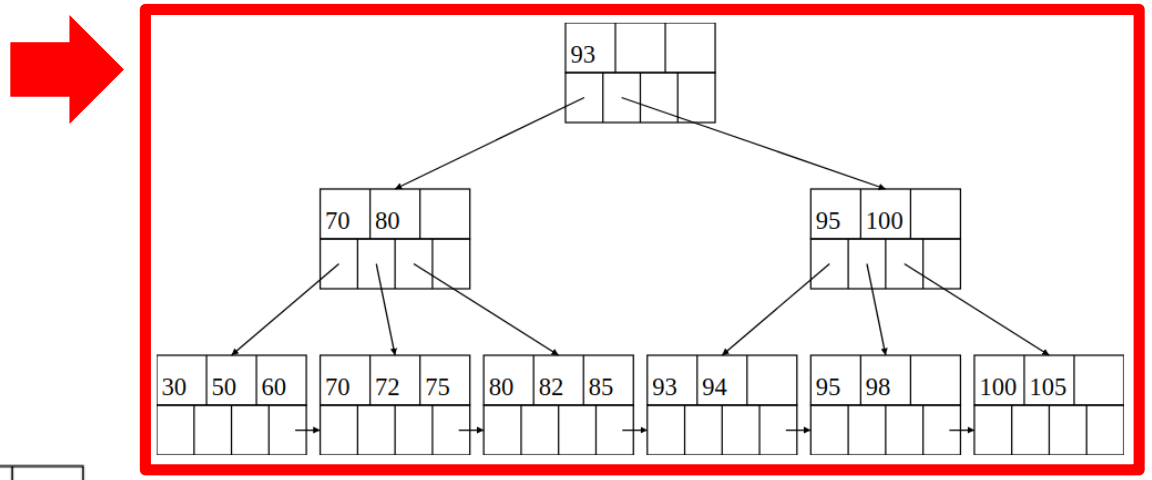




Deletion: Case 2b – More Merges

- Delete 65





Food for Thought

This is the original "B-tree" from Bayer and McCreight's 1972 paper. Does it look different from a B+ tree?

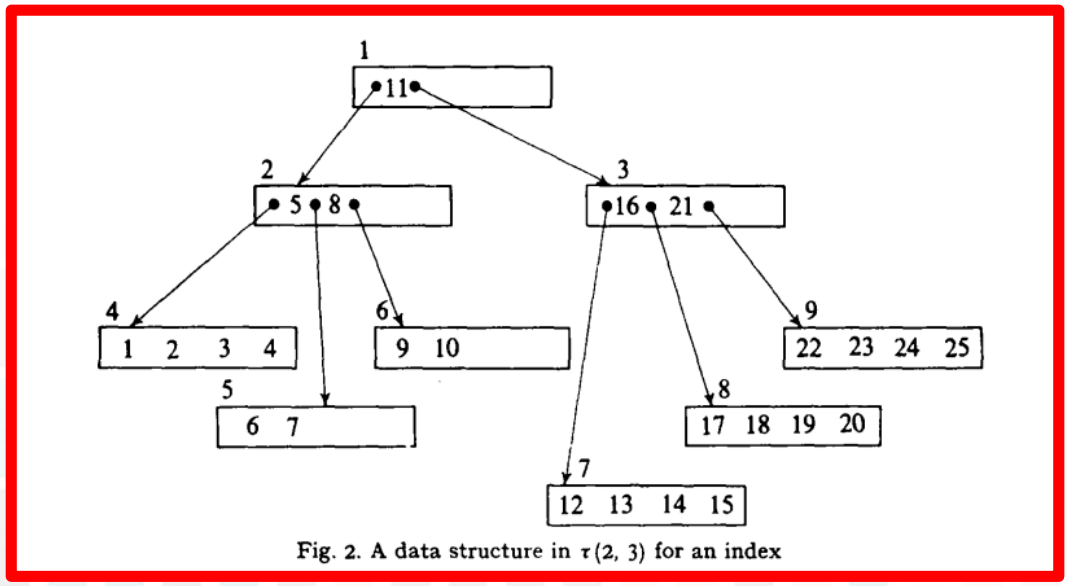


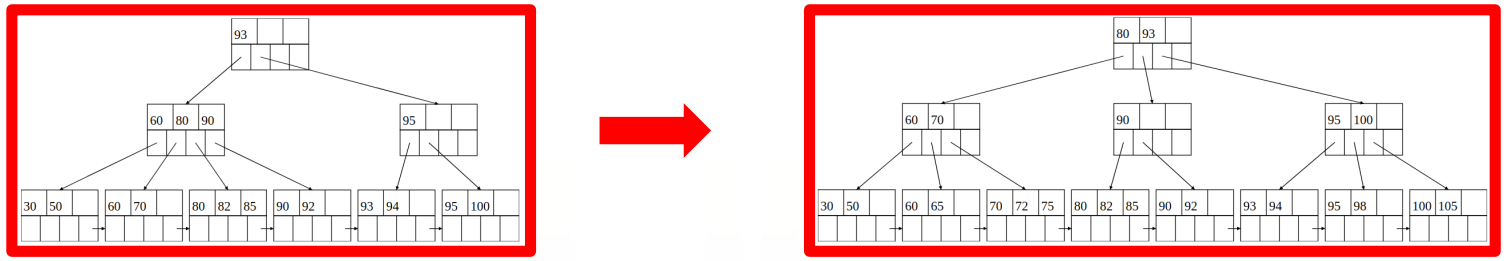
Fig. 2. A data structure in $\tau(2, 3)$ for an index

Example B-tree. Fig. 2 in Rudolf Bayer, Edward M. McCreight: Organization and Maintenance of Large Ordered Indices. Acta Informatica 1: 173-189(1972)

Food for Thought

Compare B+ tree and ISAM:
For the same data, after and after the same updates.
*How are they different—**before** and **after**?*
What would you choose:
For dynamic data? For static data?

B+ Tree



ISAM

Hash Indexing: Static Hash Tables

Accessing and Indexing Data

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Kevin C.C. Chang, Professor

Computer Science @ Illinois

Learning Objectives

By the end of this video, you will be able to:

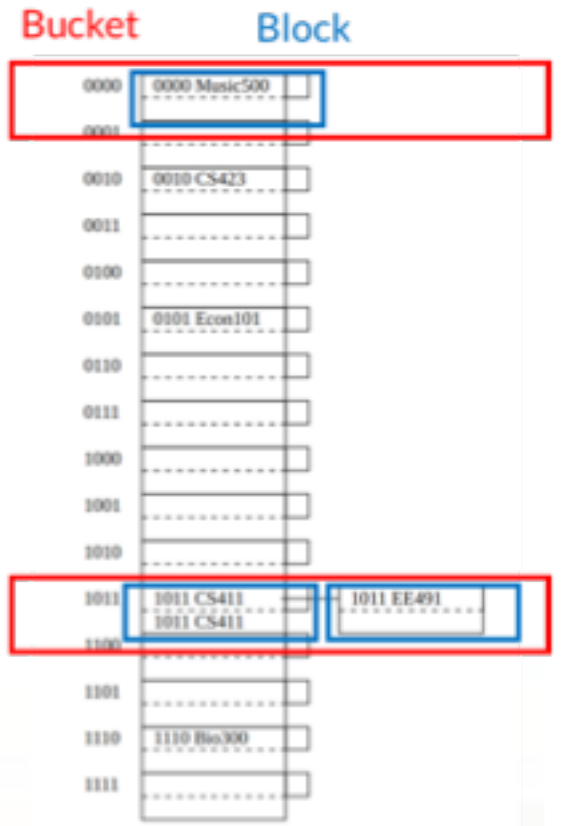
- Describe the structure of a static hash table.
- Perform lookup of data in a static hash table.
- Perform insertion and deletion operations.
- Explain why static hash tables are suitable for static data but not good for dynamic data.

Static Hash Table

- **Hash table**: Mapping **key** $K \longrightarrow$ **bucket** b by hashing
- **Hash function**: Compute key to hash value mapping
  - $h(K)$ maps key K to k-bit code in $[0, 2^k - 1]$ for addressing 2^k buckets.
  - Basic requirement: Uniform distribution of keys K to buckets $h(K)$.
- **Buckets**:
  - $n = 2^k$ buckets, numbered $0, 1, \ldots, 2^k - 1$, as addressed by the hash function.
  - Bucket $h(K)$ stores pointers for records of key K (or, store records directly).
- Stored in external storage
  - A bucket = one block.
  - Use overflow blocks when needed.

Key	Hash Code
CS423	0010
CS411	1011
Bio300	1110
Music500	0000
Econ101	0101
EE491	1011

Example hash function



Static hash table

Static Hash Table: Lookup

Lookup(K) → pointer to tuple

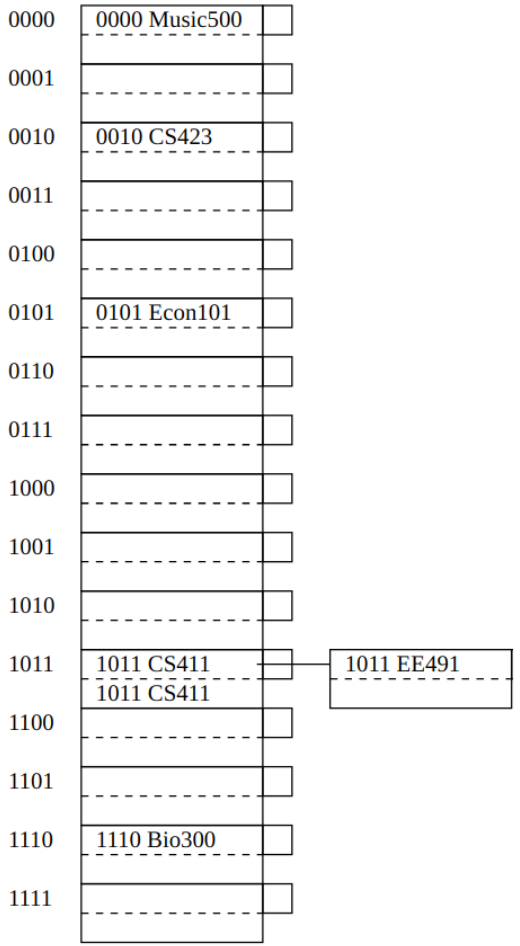
- Compute $h(K)$.
- Locate bucket B numbered by $h(K)$.
- Check if bucket contains K. Return pointer P if so.

Key	Hash Code
CS423	0010
CS411	1011
Bio300	1110
Music500	0000
Econ101	0101
EE491	1011

Example hash function

- Lookup K = "CS423"
  - Compute $h(K) = 0010$.
  - Read bucket 0010.
  - 1 disk access.

- Lookup K = "EE491"
  - Compute $h(K) = 1011$.
  - Read bucket 1011 with overflow blocks.
  - 2 disk accesses.



Static hash table

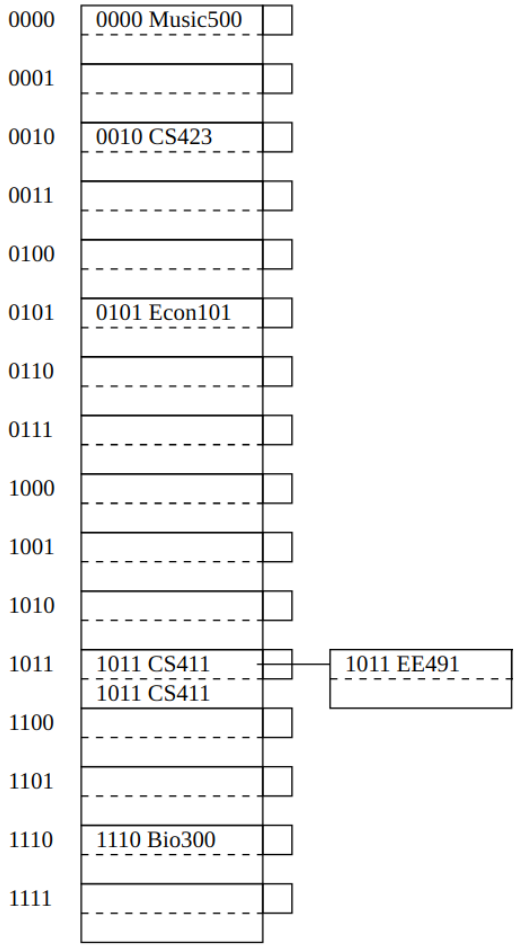
Static Hash Table: Insertions

Insert (K, P) for a new tuple with key K at pointer P

- Lookup the bucket B where K belongs.
- Place in B, if space.
- Create overflow block for B, if no space. Insert there.

Key	Hash Code
CS423	0010
CS411	1011
Bio300	1110
Music500	0000
Econ101	0101
EE491	1011

Example hash function

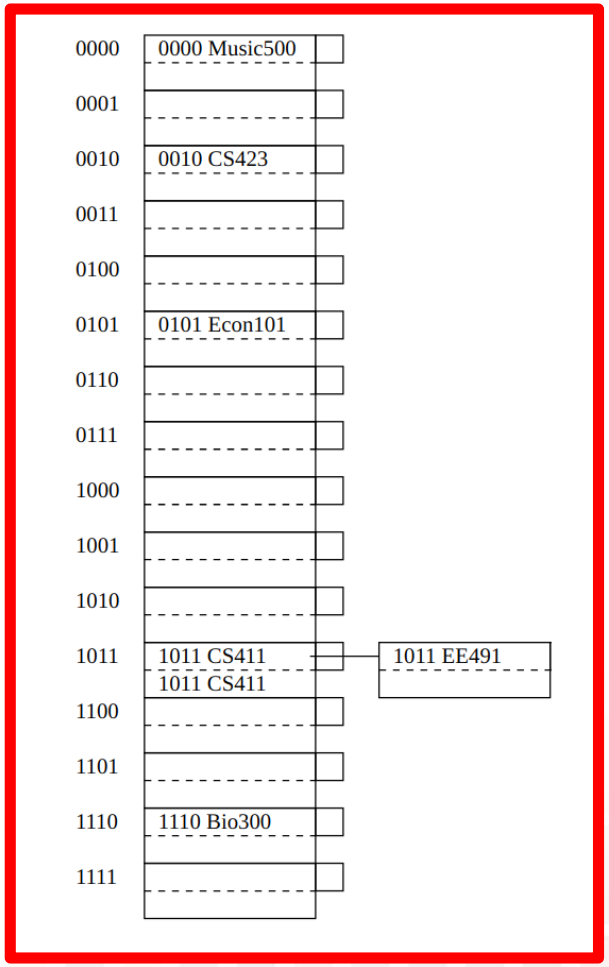


Static hash table

Static Hash Table: Good for Static Data

- For static data
  - Arrange hash functions and bucket sizes so there is no overflow.
  - Lookup performance is very good.

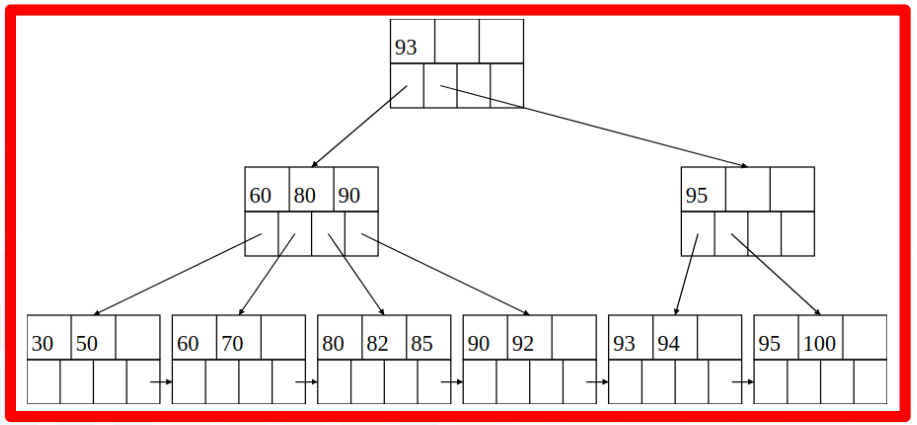
- For dynamic data
  - Block utilization may be low, if over allocate space for growth.
  - Lookup degrades considerably when many overflow blocks are formed.



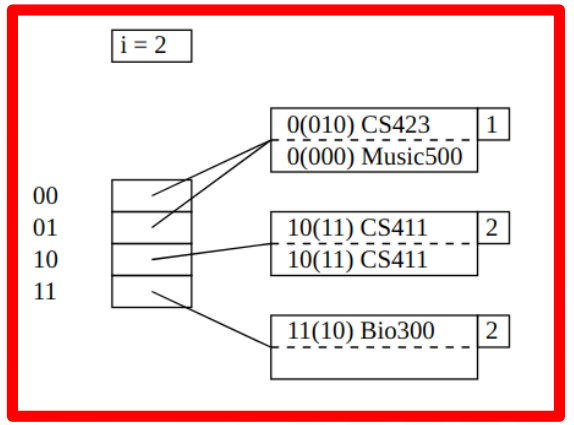
Static hash table

Food for Thought

Hashing supports very efficient lookup– but is limited in the kinds of queries. We have been using "grade" as example attribute for ISAM/B+ tree indexing.
Why do we change to "course title" for hash indexing?



Example B+ tree



Example extensible hash table

Hash Indexing: Extensible Hash Tables

Accessing and Indexing Data

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Kevin C.C. Chang, Professor

Computer Science @ Illinois

Learning Objectives

By the end of this video, you will be able to:

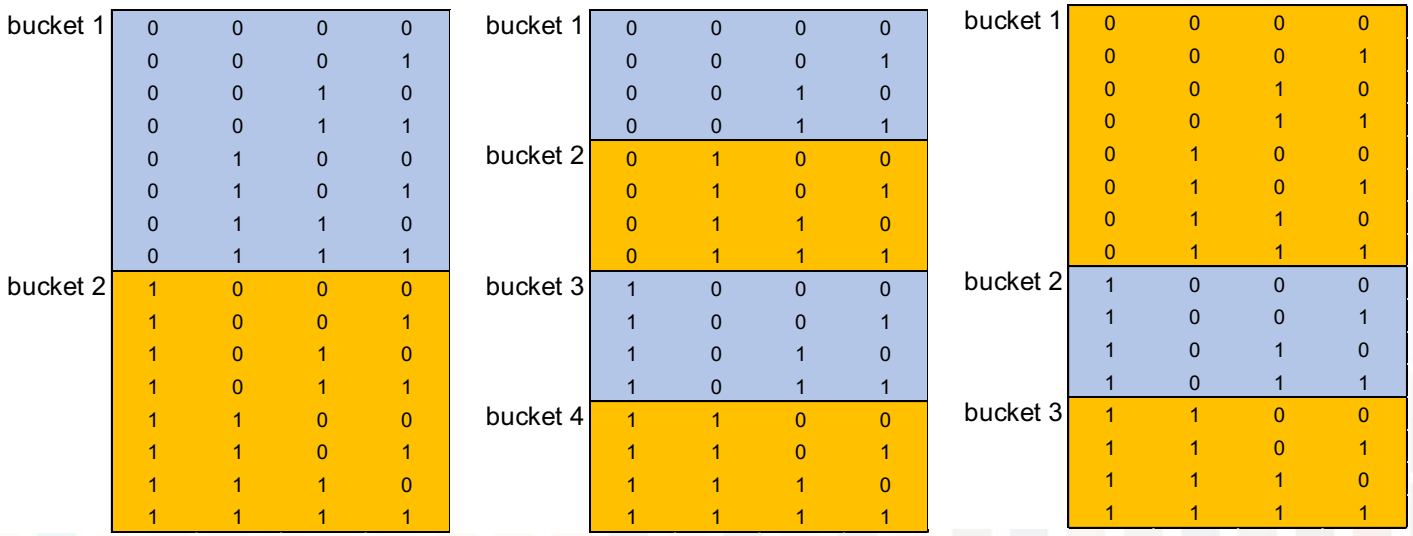
- State the principle underlying dynamic hash indexes.
- Describe the structure of an extensible hash table.
- Perform lookup of data in an extensible hash table.
- Perform insertion and deletion operations.

Can We Make Hash Tables Dynamic?
The *Principle of Hash-to-Bucket Organization*

Principle: Organize the hash code space, to allocate codes to buckets dynamically, depending on how many buckets are needed.

- Reserve larger-than-currently-needed k bits for future growth, with code space $\{0, 1, \ldots, 2^k - 1\}$.
- Use only i **most/least significant bits** (MSB/LSB), $i \leq k$, before reaching full capacity.
  - Partition code space using at most i bits into n buckets, depending on current number of records to index.
  - If **all buckets** use i bits, then $n = 2^i$.
  - If **some buckets** use $j < i$ bits, then $n < 2^i$.
- When database grows or shrinks,

  increase or decrease i and n.
  - So we can adjust the number of buckets.



$i = 1, n = 2$

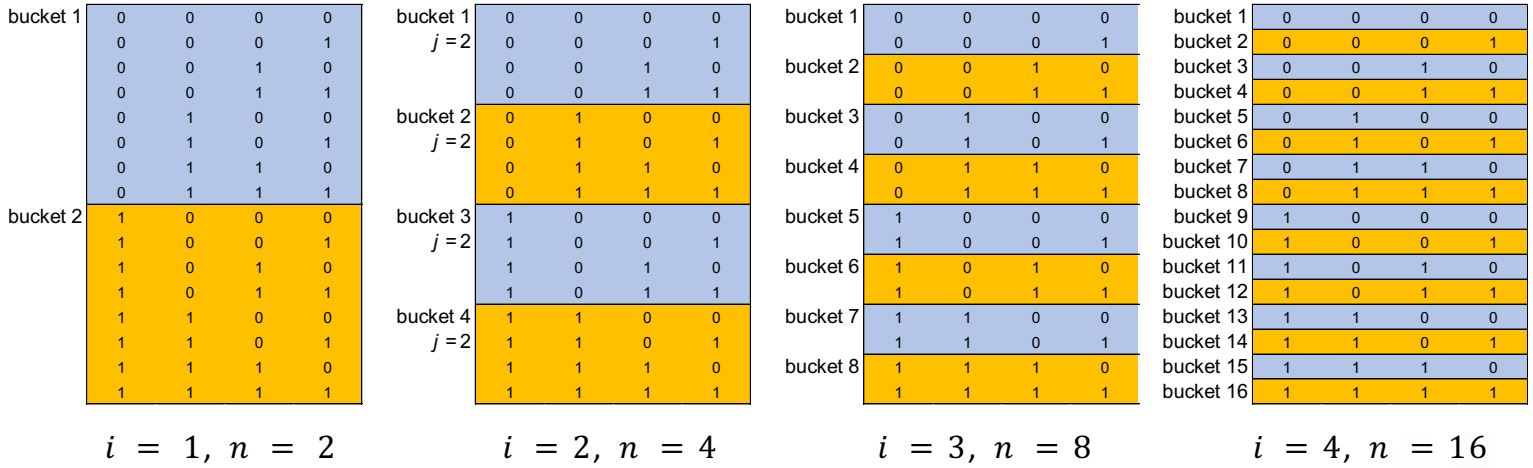
$i = 2$, all buckets use $j = 2$ bits, and thus $n = 4$

$i = 2$ but bucket 1 uses $j = 1$ bit, and thus $n = 3 < 4$

Organizing Hash Code Spaces to Buckets

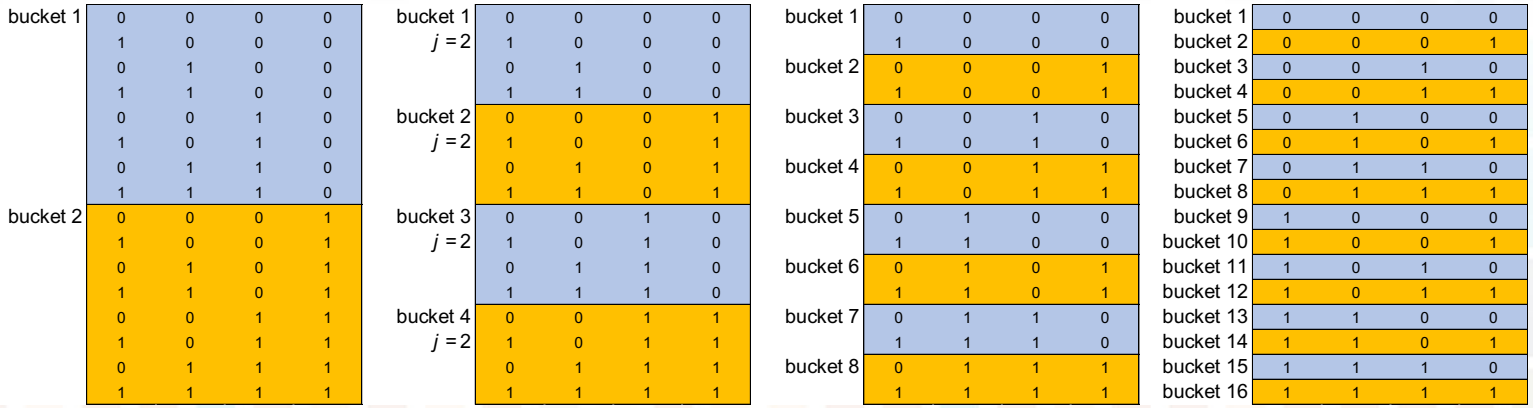
$k = 4$ bits, hash code space: 0 ~ 15, or 0000 ~ 1111

- Case 1: **Organized by MSB**
  Extensible Hash Table



$i = 1,\ n = 2$ $i = 2,\ n = 4$ $i = 3,\ n = 8$ $i = 4,\ n = 16$

- Case 2: **Organized by LSB**
  Linear Hash Table



Example hash-to-bucket organization

Organizing Hash Code Spaces to Buckets: Some Buckets May Be Merged

When using i bits, we can merge some of the 2^i buckets into one that uses $j \leq i$ bits with total $n \leq 2^i$ buckets.

- Case 1: **Organized by MSB**

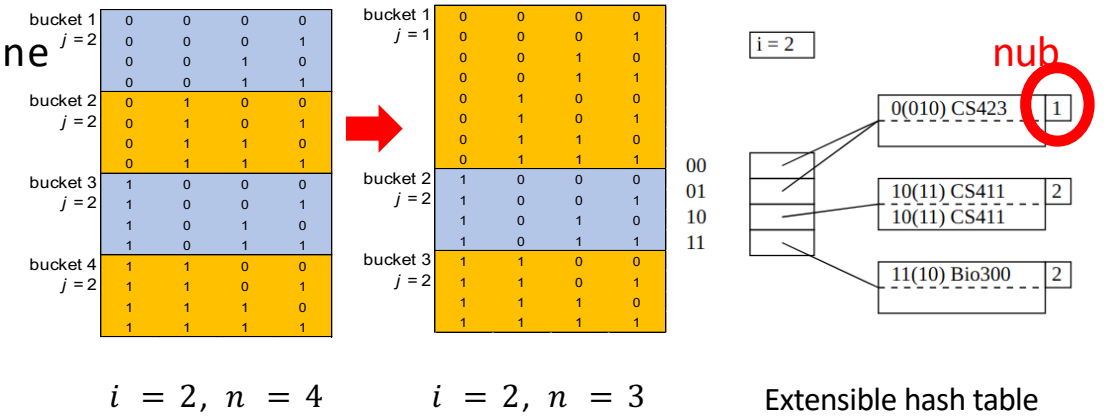
  Extensible Hash Table

  - Merge buckets sharing MSB– thus, they are neighboring buckets.
    - E.g., 00 + 01 → 0 (MSB)
  - Can merge more than two neighbors, thus $j \in \{1, 2, \dots, i\}$
  - **(nub number)** Record j bits used in the "nub" of the bucket.

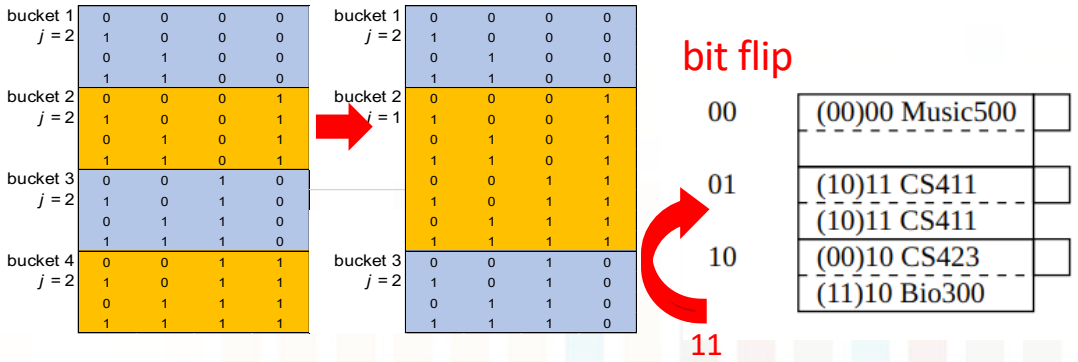
- Case 2: **Organized by LSB**

  Linear Hash Table

  - Merge buckets sharing LSB– thus, they are 2^{i-1} buckets apart.
    - E.g., 00 + 10 → 0 (LSB)
  - Allow merge at most two buckets, i.e., $j \in \{i-1, i\}$
  - **(bit flipping)** Called "bit flipping" 10 to 00.



$i = 2,\ n = 4$ $i = 2,\ n = 3$ Extensible hash table



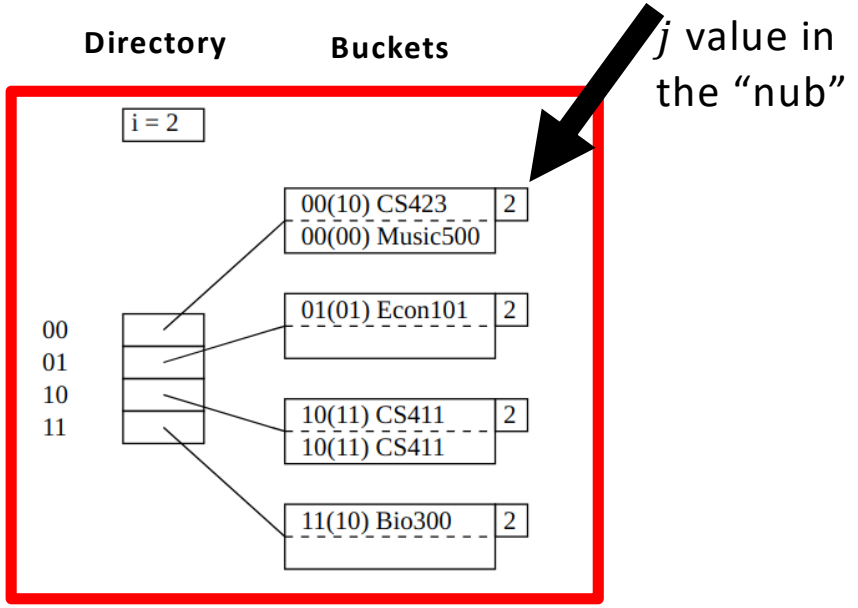
Example hash-to-bucket organization Linear hash table

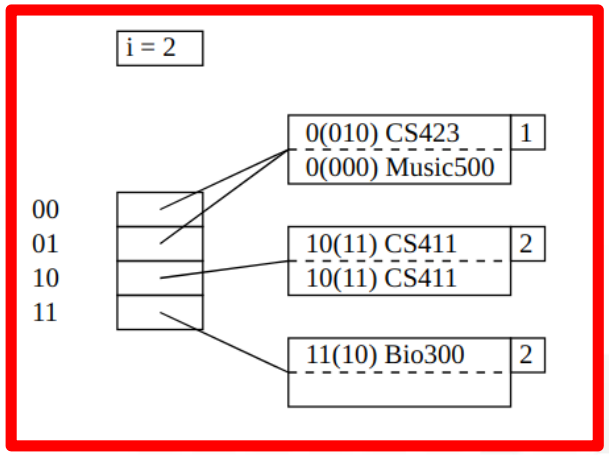
Extensible Hash Table

Key	Hash Code
CS423	0010
CS411	1011
Bio300	1110
Music500	0000
Econ101	0101
EE491	1011

Example hash function

- Hash values have k bits.
- Currently use $i \leq k$ MSB.
- Each bucket may use $j \leq i$ bits
  - As indicated in the "nub" of the bucket.
- Directory
  - Lists current i-bit codes and pointers to corresponding buckets.
- Buckets
  - Stores keys and pointers to tuples with those keys.
  - Or stores tuples directly.





Example extensible hash tables for $k = 4$ and $i = 2$

Extensible Hash Table: Lookup

Lookup(K) → pointer to tuple

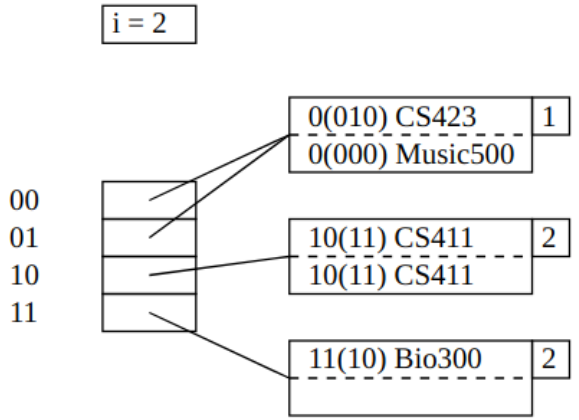
Suppose hash table currently uses i bits.

- Compute $h(K)$.
- Use the i MSB of the hash code.
- Check the directory and follow pointer to locate the bucket B.
- Check if key K is contained in B, and return its pointer P.

Key	Hash Code
CS423	0010
CS411	1011
Bio300	1110
Music500	0000
Econ101	0101
EE491	1011

Example hash function

- E.g.: Lookup("CS423")



Example extensible hash table

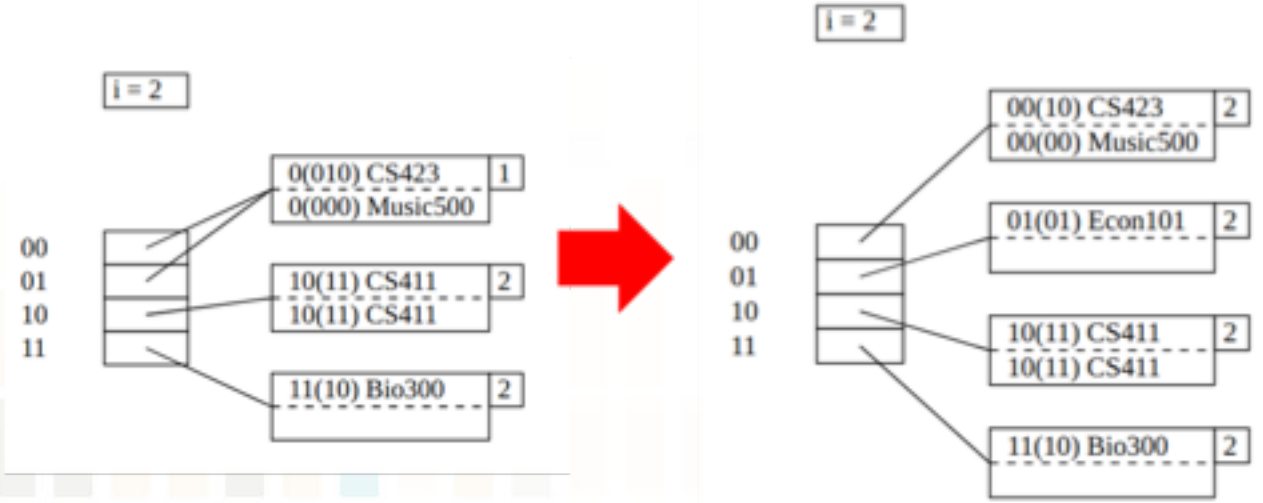
Insertions: Case 1 – Split Bucket

Insert (K, P) for a new tuple with key K at pointer P

- Lookup the bucket B where K belongs.
- Insert K, P to B.
- If bucket B becomes too full: Suppose B currently uses j bits.
  - (Case 1) If $j < i$: Split B into buckets B_1 and B_2 each using $j + 1$ bits.
    - That is, split B with $b_1 b_2 \ldots b_j$ into B_1 and B_2 with codes $b_1 b_2 \ldots b_j \mathbf{0}$ and $b_1 b_2 \ldots b_j \mathbf{1}$.
    - Distribute keys accordingly.

Key	Hash Code
CS423	0010
CS411	1011
Bio300	1110
Music500	0000
Econ101	0101
EE491	1011

Example hash function



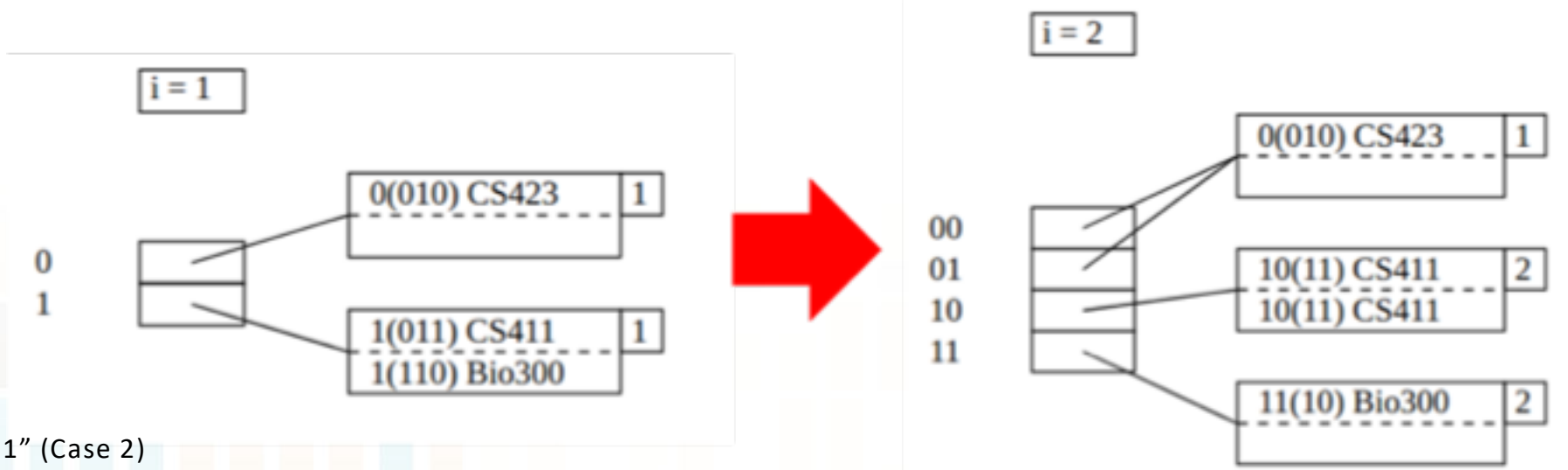
Insert “Econ101” (Case 1)

Insertions: Case 2 – $i = i + 1$ (Double Directory)

- (Case 2) If $j = i$:
  - Increase used bits i to $i + 1$.
  - Double the directory.
  - Split bucket B as in Case 1.

Key	Hash Code
CS423	0010
CS411	1011
Bio300	1110
Music500	0000
Econ101	0101
EE491	1011

Example hash function



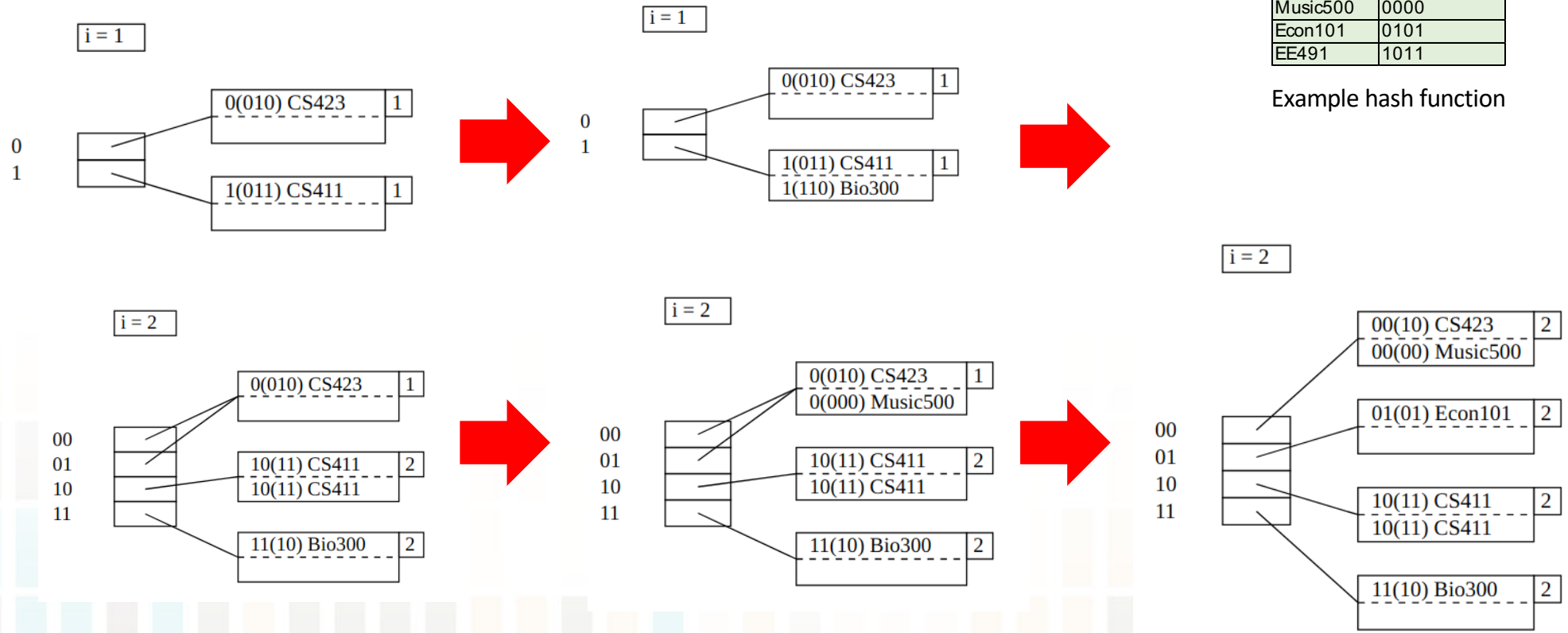
Insert “CS411” (Case 2)

Extensible Hash Table Insertion Examples

- Insert 'Bio300' (1110), 'CS411' (1011), 'Music500' (0000), 'Econ101' (0101)

Key	Hash Code
CS423	0010
CS411	1011
Bio300	1110
Music500	0000
Econ101	0101
EE491	1011

Example hash function



Extensible-Hash Table insertion examples

Hash Indexing: Linear Hash Tables

Accessing and Indexing Data

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Kevin C.C. Chang, Professor

Computer Science @ Illinois

Learning Objectives

By the end of this video, you will be able to:

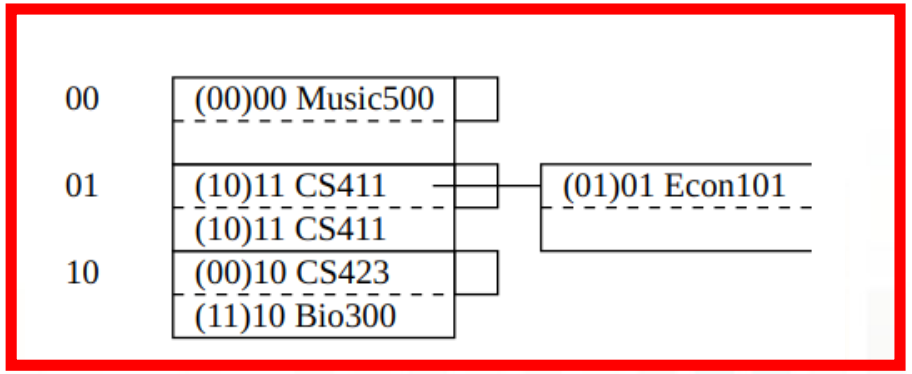
- Describe the structure of a linear hash table.
- Perform lookup of data in a linear hash table.
- Perform insertion and deletion operations.
- Compare and contrast extensible and linear hash tables.

Linear Hash Table

- Hash values have k bits.
- Currently use $i \leq k$ LSB.
- Buckets
  - Stores keys and pointers to tuples with those keys.
  - Or stores tuples directly.
- Overflow blocks
  - Some buckets may have overflow blocks.
- No directory!
- Average capacity (number of keys per bucket) u
  - Require that $\frac{r}{n} < u$, for r keys in n buckets.

Key	Hash Code
CS423	0010
CS411	1011
Bio300	1110
Music500	0000
Econ101	0101
EE491	1011

Example hash function



Example linear hash tables for $k = 4$ and $i = 2$

Linear Hash Table: Lookup

Lookup(K) $\rightarrow$ pointer to tuple

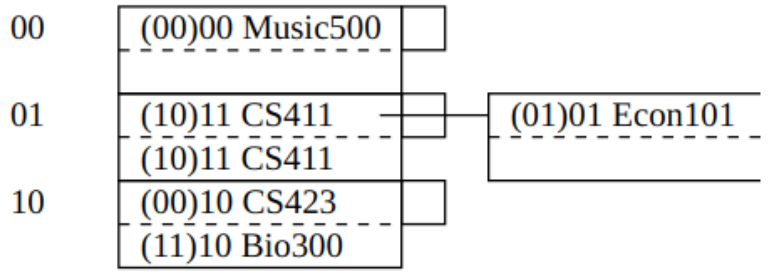
Suppose hash table currently uses i bits and there are n buckets.

- Compute $h(K)$.
- Use the i LSB of the hash code. Suppose the value is m.
  - (Case 1) If $m \leq n$, find bucket numbered m.
  - (Case 2) If $m > n$, "flip" the highest bit of m and go to Case 1.

- E.g.: Lookup("CS423"), Lookup("CS411")

Key	Hash Code
CS423	0010
CS411	1011
Bio300	1110
Music500	0000
Econ101	0101
EE491	1011

Example hash function



Example linear hash table

Linear Hash Table: Insertions

Key	Hash Code
CS423	0010
CS411	1011
Bio300	1110
Music500	0000
Econ101	0101
EE491	1011

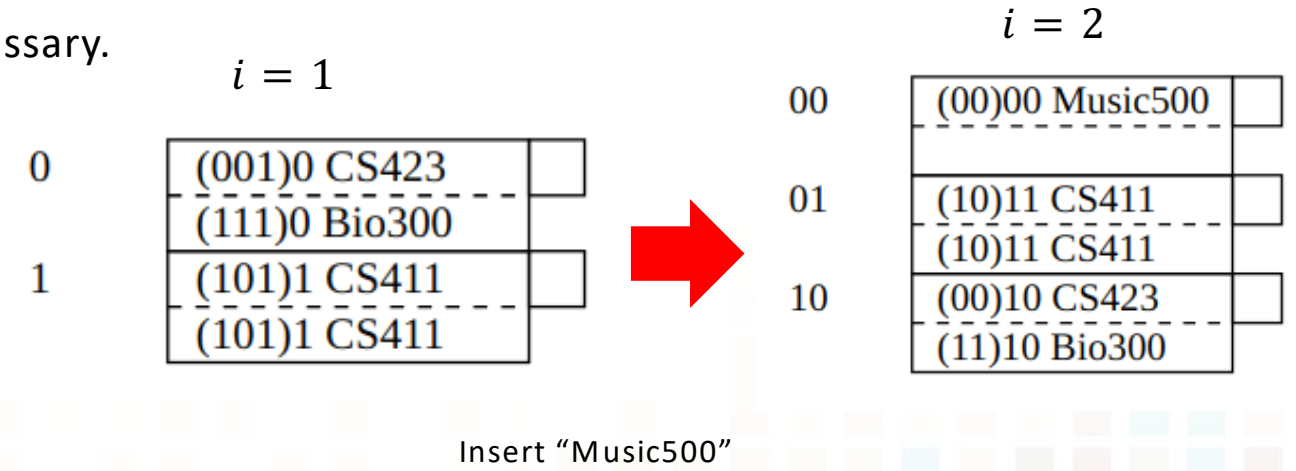
Example hash function

Insert (K, P) for a new tuple with key K at pointer P

Suppose hash table uses i bits, has r keys, n buckets,

and requires average capacity r/n no more than u.

- Compute $h(K)$. Locate bucket B with i LSB of the hash code.
- Insert K, P to B, using overflow block if necessary.
- If hash table becomes too full, i.e., $\frac{r}{n} > u$.
  - Add one more bucket numbered $n + 1$.
  - Get "flipped" entries to the new bucket.



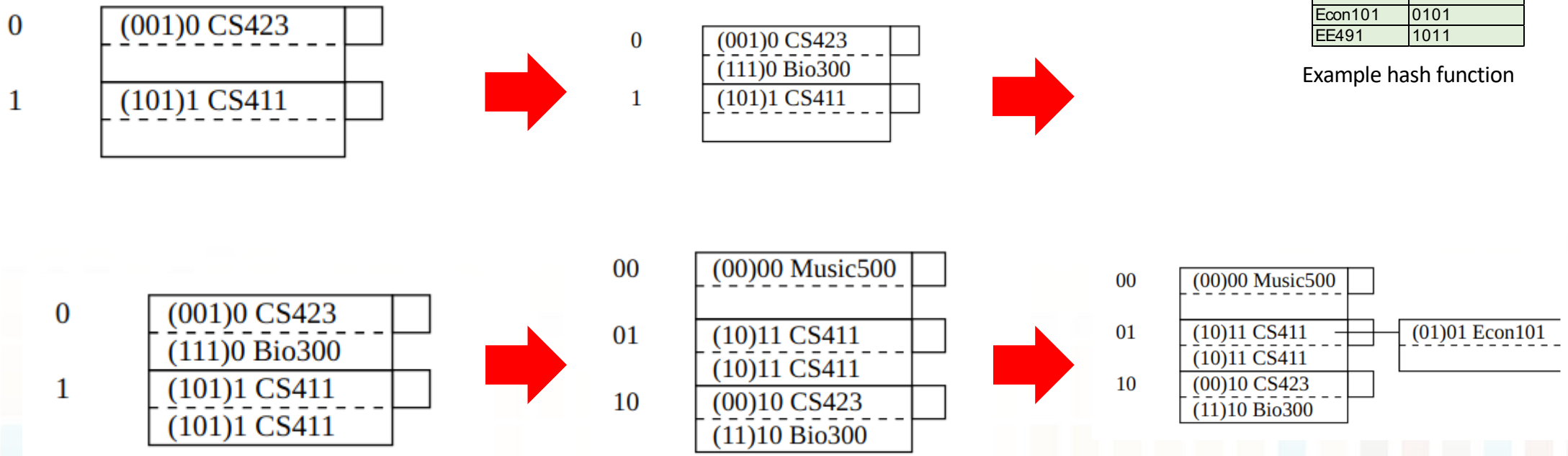
Insert "Music500"

Linear Hash Table Insertion Examples

- Insert 'Bio300' (1110), 'CS411' (1011), 'Music500' (0000), 'Econ101' (0101). Assume $u = 2$.

Key	Hash Code
CS423	0010
CS411	1011
Bio300	1110
Music500	0000
Econ101	0101
EE491	1011

Example hash function



Extensible-Hash Table insertion examples

Food for Thought

Why do we need a "directory" for an extensible hash table? Why not for a linear hash table?

Hint: If there are currently 3 buckets in an extensible hash table (say, for $k = 4$, $i = 2$), do we know what code each bucket has? How about linear hash?

Food for Thought

Compare extensible vs. linear hash tables.

Feature	Extensible Hash Table	Linear Hash Table
Used Bits	MSB	LSB
Directory	Yes	No
Flip Bits in Lookup	???	???
Lookup in Buckets	Check ??? Bucket(s)	Check ??? Bucket(s)
Increment of Growth	2X directory. +1 ~ 2^i buckets	+1 bucket
Overflow Blocks	???	Yes
Growth Criteria	???	???
???		

Comparison: extensible vs. linear hash tables

Index Classification

Accessing and Indexing Data

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Kevin C.C. Chang, Professor

Computer Science @ Illinois

Learning Objectives

By the end of this video, you will be able to:

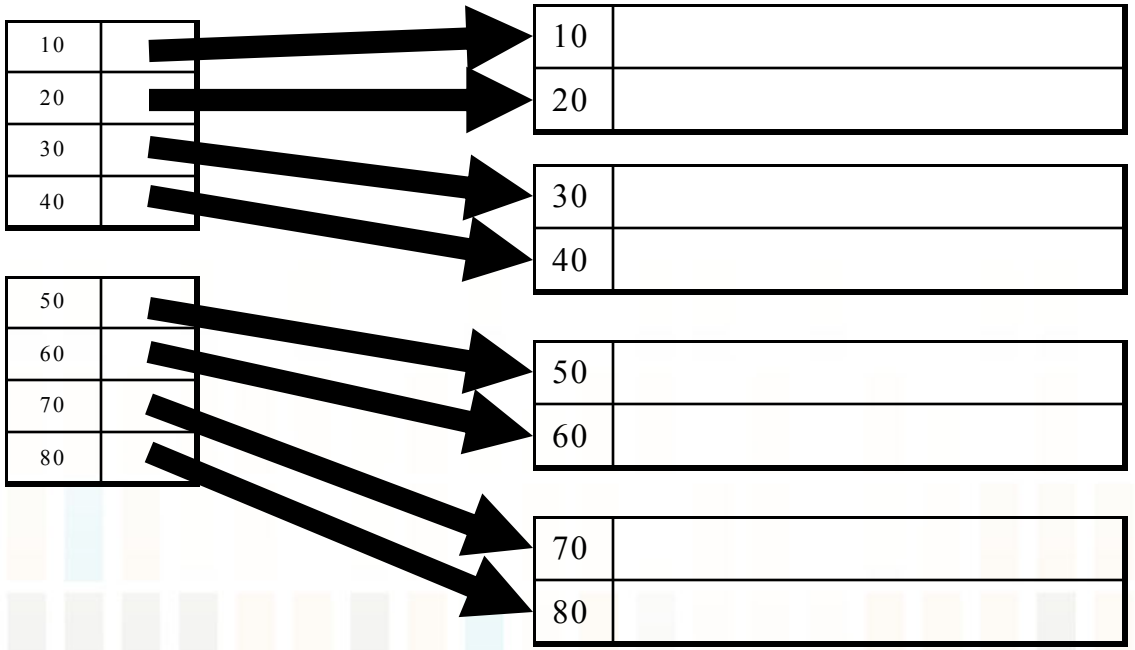
- Define dense vs. sparse indexes.
- Define clustered vs. unclustered indexes.
- Define primary vs. secondary indexes.
- Define tree vs. hash-based indexes.

Types of Indexes

- Clustered/unclustered
  - Clustered = records in storage are sorted in the order of index attribute.
  - Unclustered = otherwise.
- Dense/sparse
  - Dense = each record has an entry in the index.
  - Sparse = only some records have.
- Primary/secondary
  - Primary = on the primary key.
  - Secondary = on any key.

Clustered, Dense Index

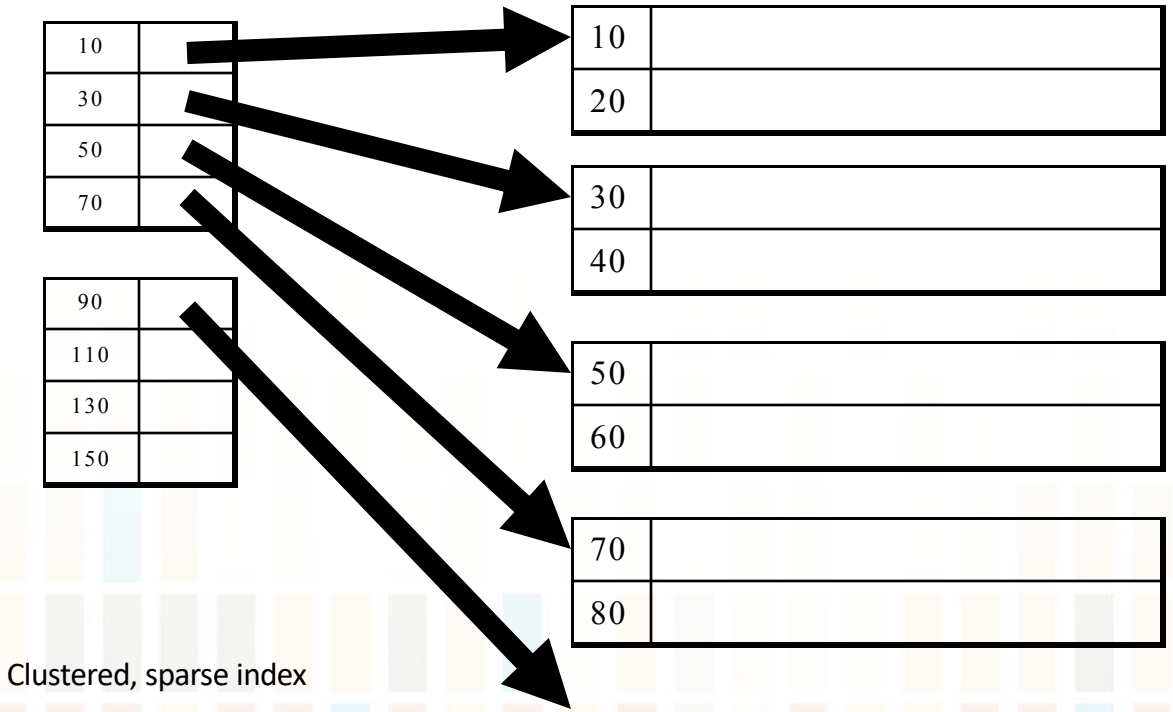
- Clustered: File is sorted on the index attribute.
- Dense: Keys cover all values.



Clustered, dense index

Clustered, Sparse Index

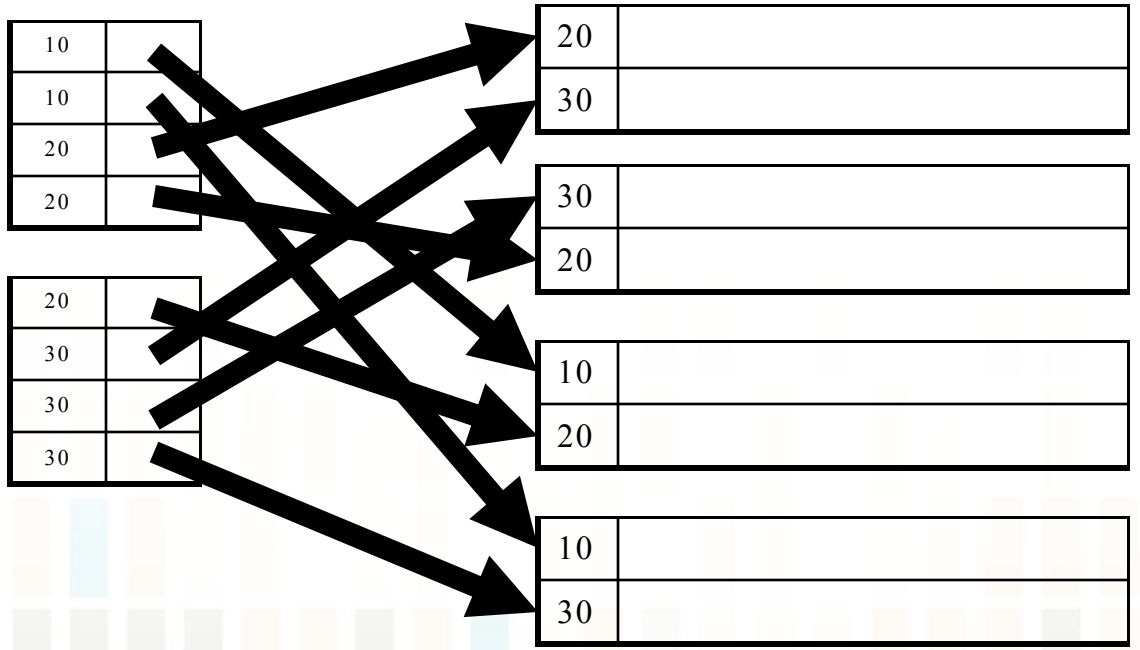
- Sparse index: one key per data block.



Clustered, sparse index

Unclustered Indexes: Always Dense

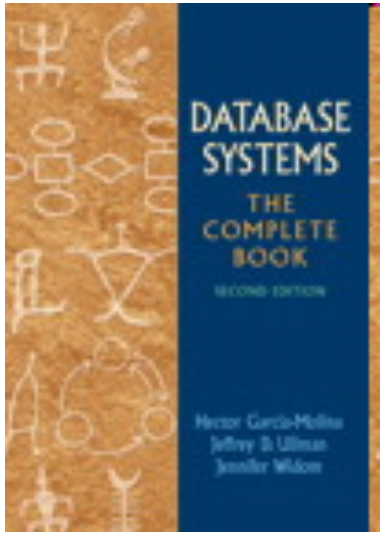
- Often for indexing other attributes than primary key
- Always dense (why ?)



Unclustered index

Quiz: Indexing in the Real Life -- *Consider a Textbook as Example?*

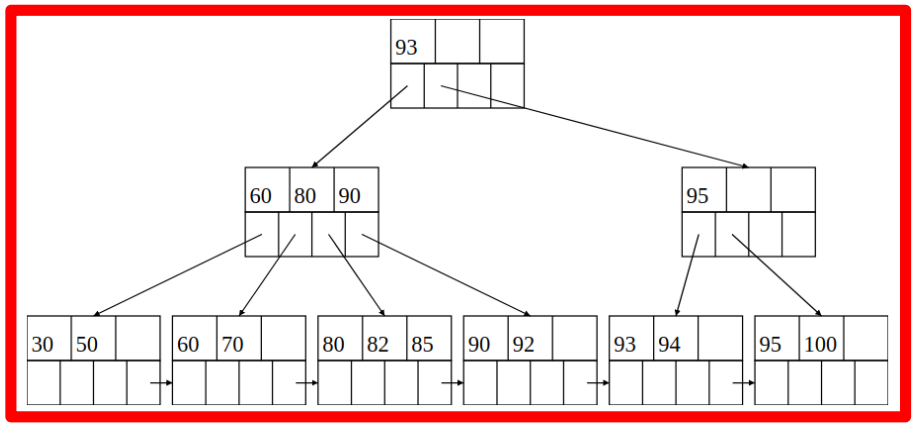
- How many indexes? Where?
- What are keys → Pointers → Records?
  - Table of contents:
    - Chapter name → Page number → Chapter
  - Concept index:
    - Concept name → Page number → Concept
- Clustered?
- Dense?
- Primary?



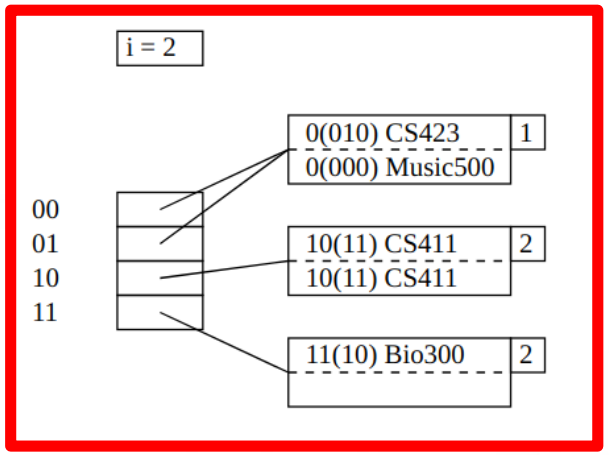
Concept index, in Database Systems: The Complete Book (2nd Ed.) by Hector Garcia-Molina, Jeffrey D. Ullman, , Jennifer Widom

Structures of Indexes

- Tree-based
  - ISAM
  - B-tree, B+ tree
  - R-tree
  - Quad tree, k-d tree, …
- Hash-based
  - Static hash table
  - Extensible/linear hash table
  - Grid file



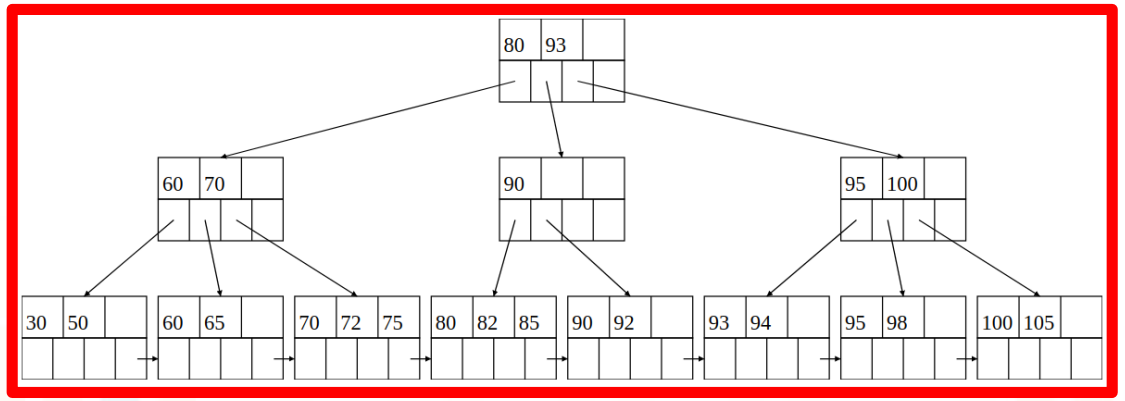
Example B+ tree



Example extensible hash table

Food for Thought

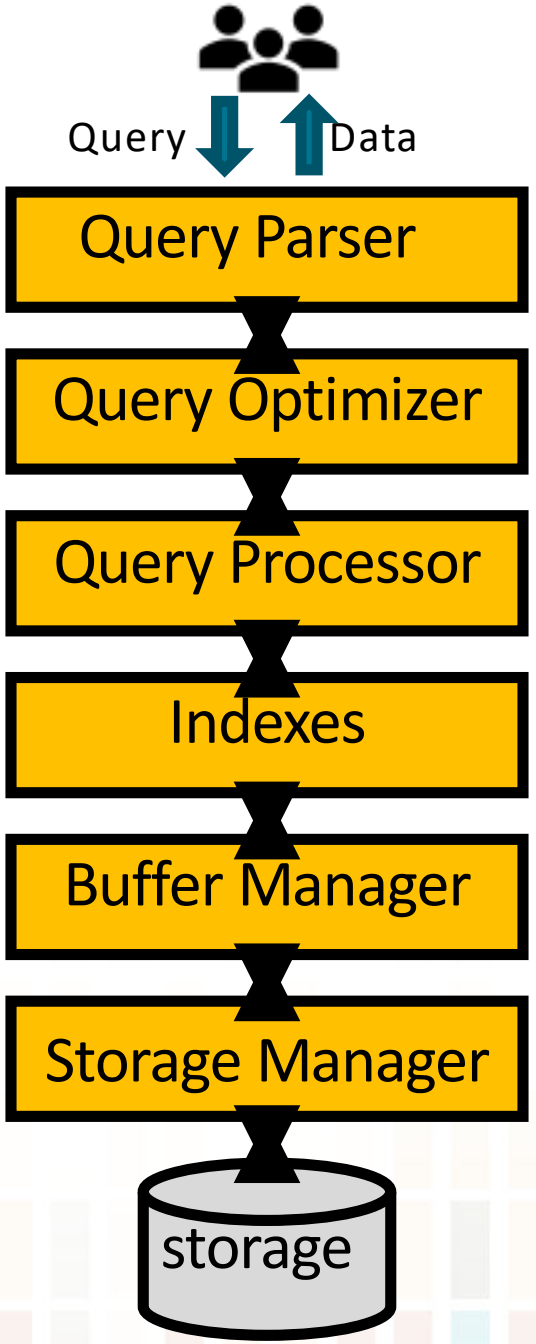
Can we make a B+ tree index clustered or unclustered? Dense or sparse? How about ISAM? How about extensible hash table?



Example B+ tree

The End

Figure


Query
Data
Query Parser
Query Optimizer
Query Processor
Indexes
Buffer Manager
Storage Manager
storage

Figure

name	course	grade
Bugs Bunny	CS411	90
Donald Duck	Bio300	92
Donald Duck	CS423	93
Donald Duck	CS411	95
Bugs Bunny	CS423	80
Mickey Mouse	CS423	70
Peter Pan	CS411	94
Charlie Brown	Econ101	50
Peter Pan	Econ101	82
Eeyore	Bio300	60
Mickey Mouse	Econ101	85
Ariel	CS411	100
Fred Flintstone	CS423	30

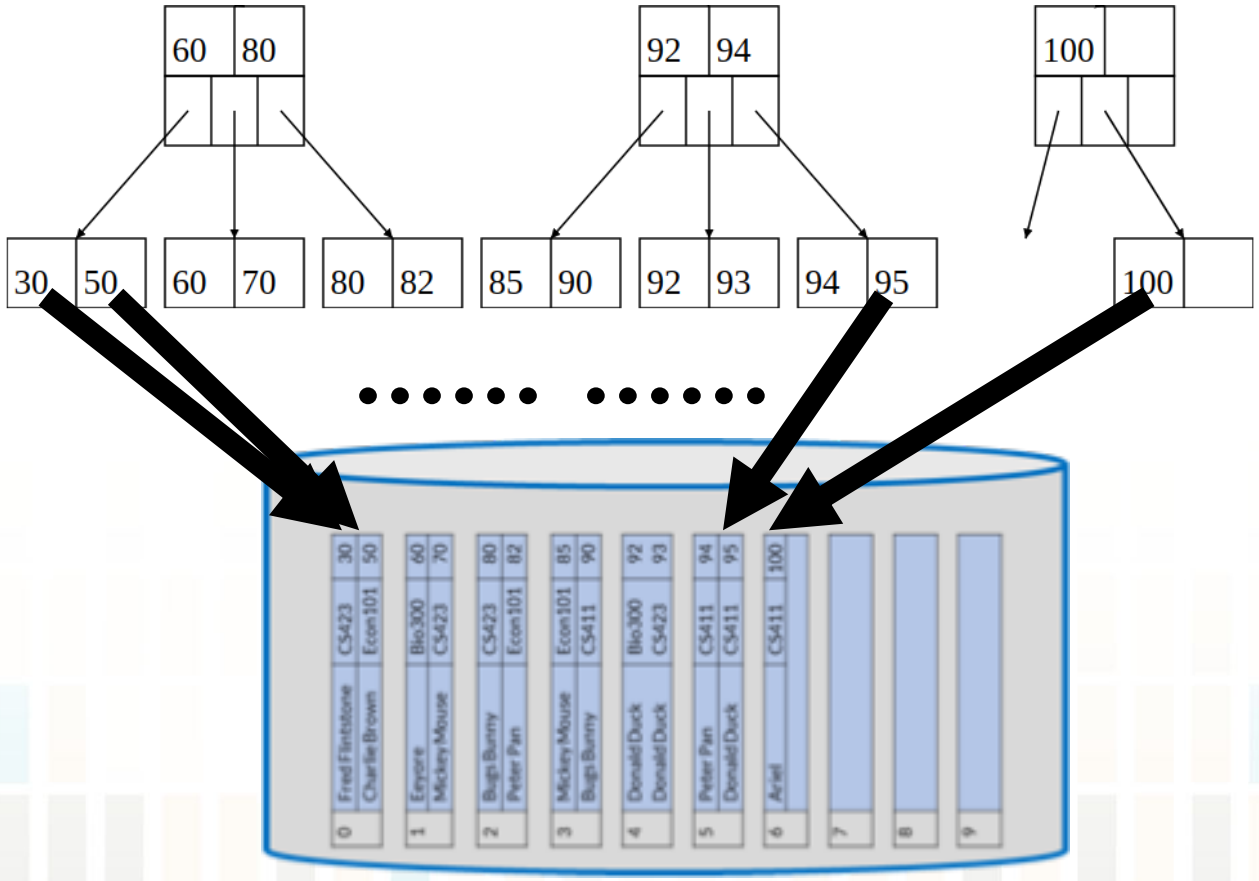
0	Fred Flintstone	CS423	30
	Charlie Brown	Econ101	50
1	Eeyore	Bio300	60
	Mickey Mouse	CS423	70
2	Bugs Bunny	CS423	80
	Peter Pan	Econ101	82
3	Mickey Mouse	Econ101	85
	Bugs Bunny	CS411	90
4	Donald Duck	Bio300	92
	Donald Duck	CS423	93
5	Peter Pan	CS411	94
	Donald Duck	CS411	95
6	Ariel	CS411	100
7			
8			
9			

0	Fred Flintstone	CS423	30
	Charlie Brown	Econ101	50
1	Eeyore	Bio300	60
	Mickey Mouse	CS423	70
2	Bugs Bunny	CS423	80
	Peter Pan	Econ101	80
3	Mickey Mouse	Econ101	80
	Bugs Bunny	CS411	90
4	Donald Duck	Bio300	90
	Donald Duck	CS423	90
5	Peter Pan	CS411	95
	Donald Duck	CS411	95
6	Ariel	CS411	100
7			
8			
9			

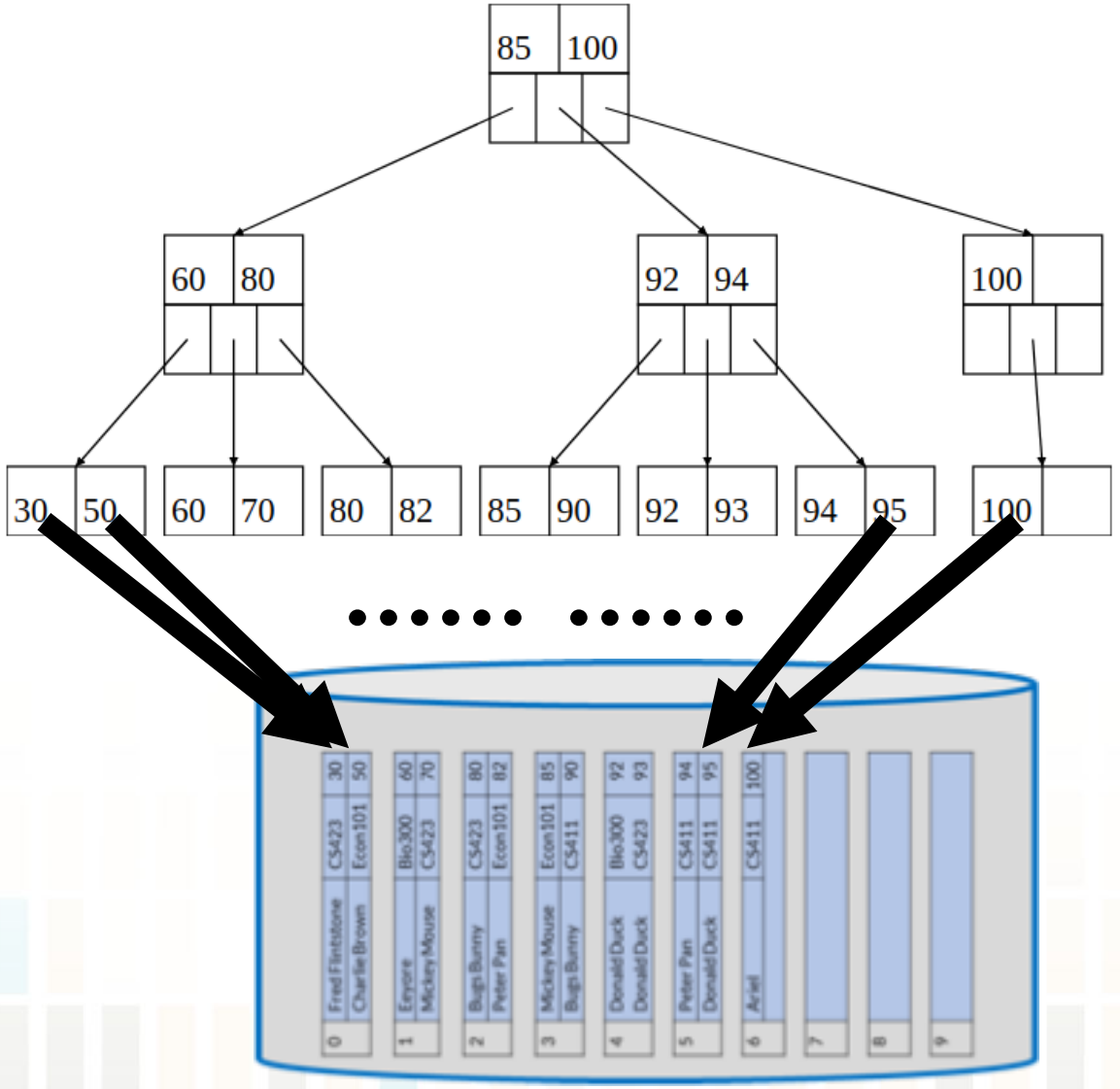
Figure

0	Bugs Bunny	CS411	90
	Donald Duck	Bio300	92
1	Donald Duck	CS423	93
	Donald Duck	CS411	95
2	Bugs Bunny	CS423	80
	Mickey Mouse	CS423	70
3	Peter Pan	CS411	94
	Charlie Brown	Econ101	50
4	Peter Pan	Econ101	82
	Eeyore	Bio300	60
5	Mickey Mouse	Econ101	85
	Ariel	CS411	100
6	Fred Flintstone	CS423	30
7			
8			
9			

Figure

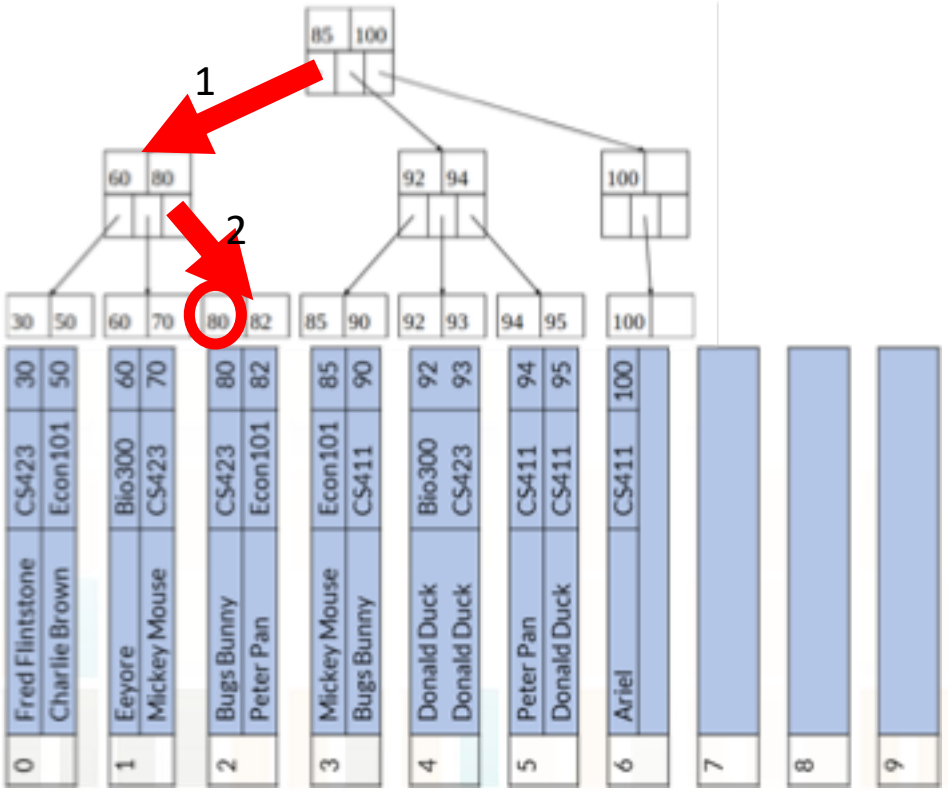

60 80
92 94
100
30 50
60 70
80 82
85 90
92 93
94 95
100
0 Fred Flintstone Charlie Brown CS423 Econ101 30 50
1 Eeyore Mickey Mouse Bio300 CS423 60 70
2 Bugs Bunny Peter Pan CS423 Econ101 80 82
3 Mickey Mouse Bugs Bunny Econ101 CS411 85 90
4 Donald Duck Donald Duck Bio300 CS423 92 93
5 Peter Pan Donald Duck CS411 CS411 94 95
6 Ariel CS411 100
7
8
9

Figure


85 100
60 80
92 94
100
30 50
60 70
80 82
85 90
92 93
94 95
100
0 Fred Flintstone Charlie Brown 30 50 CS423 Econ101
1 Eeyore Mickey Mouse 60 70 Bio300 CS423
2 Bugs Bunny Peter Pan 80 82 CS423 Econ101
3 Mickey Mouse Bugs Bunny 85 90 Econ101 CS411
4 Donald Duck Donald Duck 92 93 Bio300 CS423
5 Peter Pan Donald Duck 94 95 CS411 CS411
6 Ariel 100 CS411
7
8
9

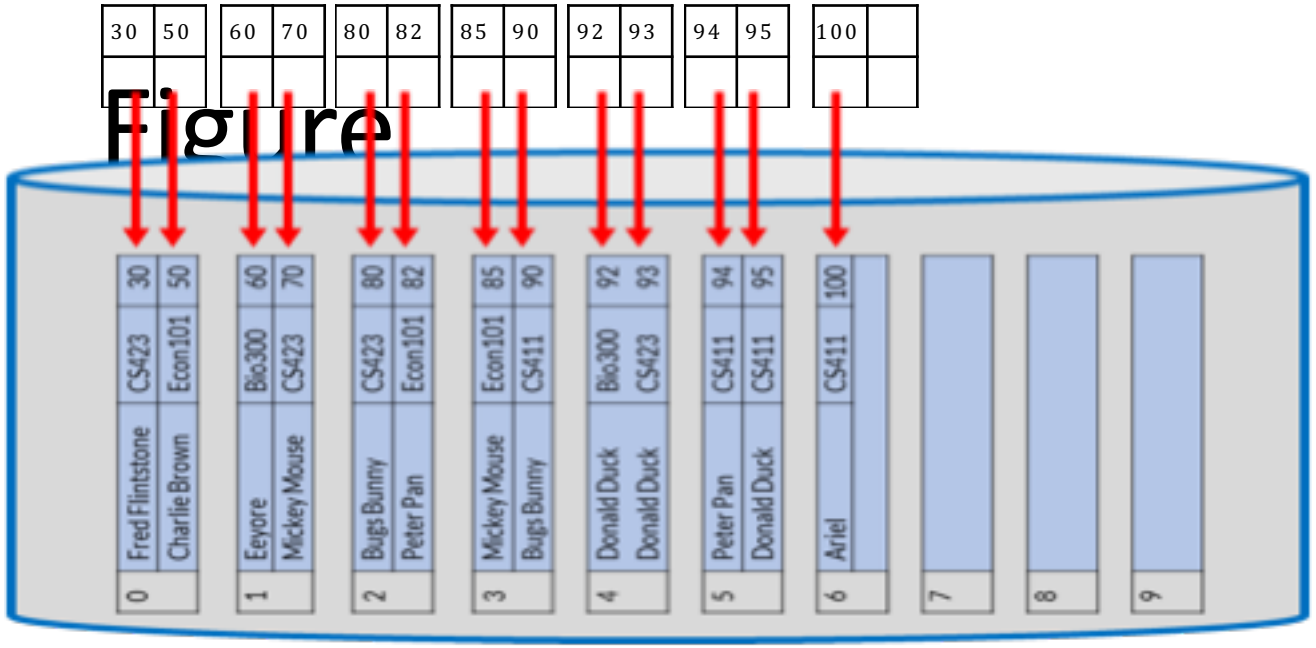
Figure

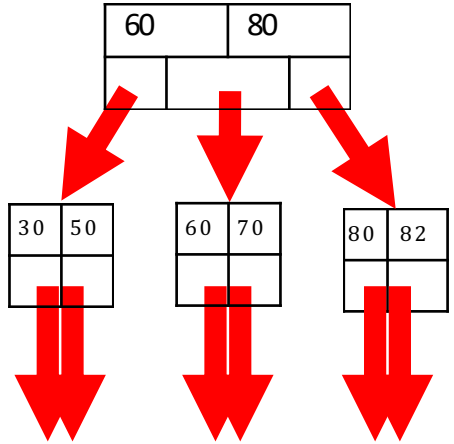
- *Q1: Find students **who scored 80** in CS411.*

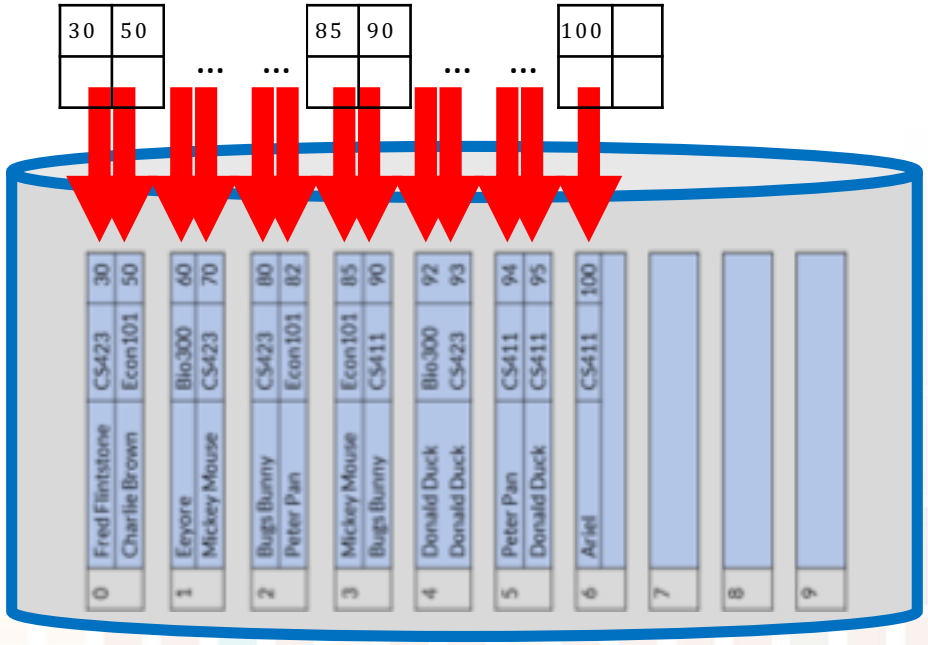


Search 80 in an example ISAM index

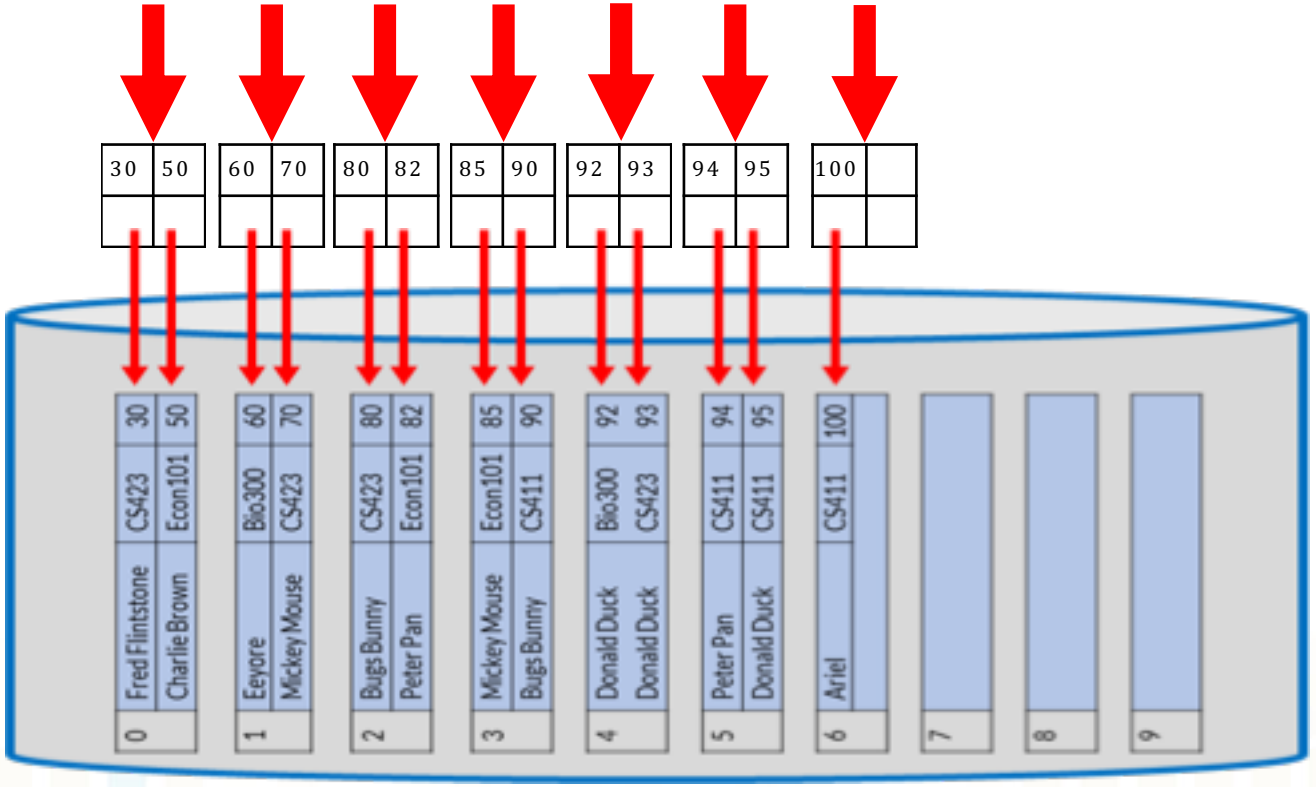
Figure


30 50
60 70
80 82
85 90
92 93
94 95
100
0 Fred Flintstone Charlie Brown CS423 Econ101 30 50
1 Eeyore Mickey Mouse Bio300 CS423 60 70
2 Bugs Bunny Peter Pan CS423 Econ101 80 82
3 Mickey Mouse Bugs Bunny Econ101 CS411 85 90
4 Donald Duck Donald Duck Bio300 CS423 92 93
5 Peter Pan Donald Duck CS411 CS411 94 95
6 Ariel CS411 100
7
8
9


60 80
30 50
60 70
80 82

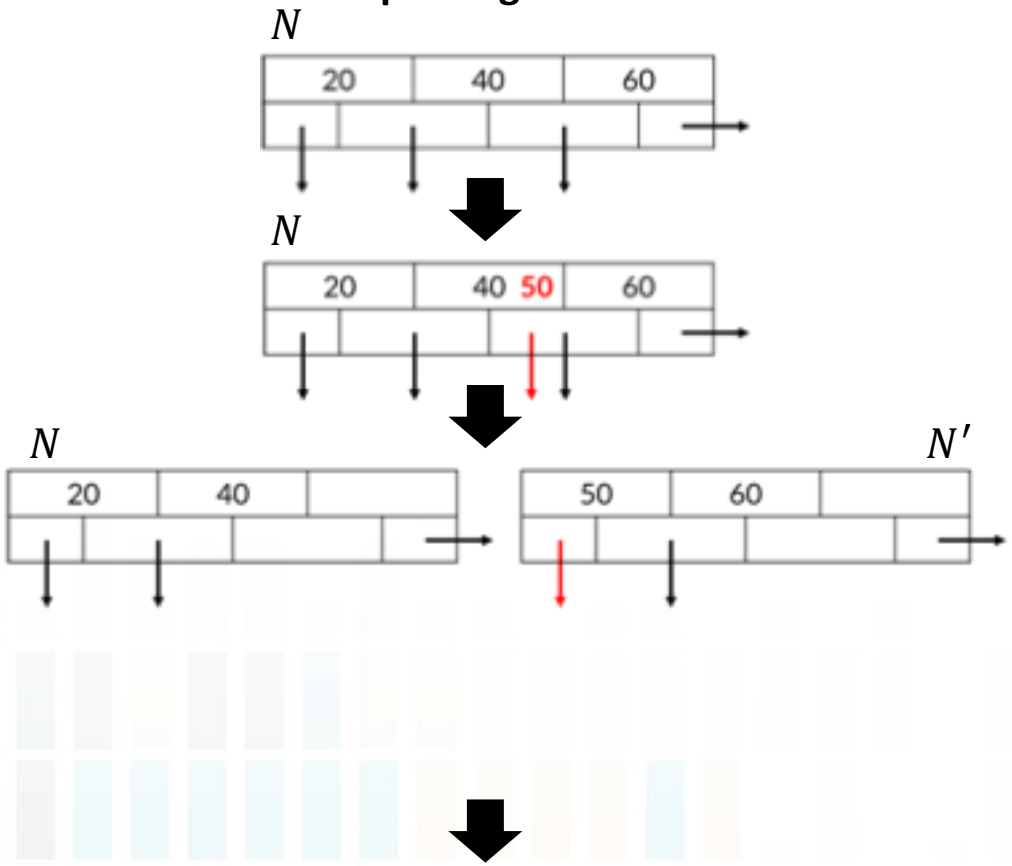

30 50
85 90
100
0 Fred Flintstone Charlie Brown CS423 Econ101 30 50
1 Eeyore Mickey Mouse Bio300 CS423 60 70
2 Bugs Bunny Peter Pan CS423 Econ101 80 82
3 Mickey Mouse Bugs Bunny Econ101 CS411 85 90
4 Donald Duck Donald Duck Bio300 CS423 92 93
5 Peter Pan Donald Duck CS411 CS411 94 95
6 Ariel CS411 100
7
8
9

Figure


30 50
60 70
80 82
85 90
92 93
94 95
100
0 Fred Flintstone CS423 30
Charlie Brown Econ101 50
1 Eeyore Bio300 60
Mickey Mouse CS423 70
2 Bugs Bunny CS423 80
Peter Pan Econ101 82
3 Mickey Mouse Econ101 85
Bugs Bunny CS411 90
4 Donald Duck Bio300 92
Donald Duck CS423 93
5 Peter Pan CS411 94
Donald Duck CS411 95
6 Ariel CS411 100
7
8
9

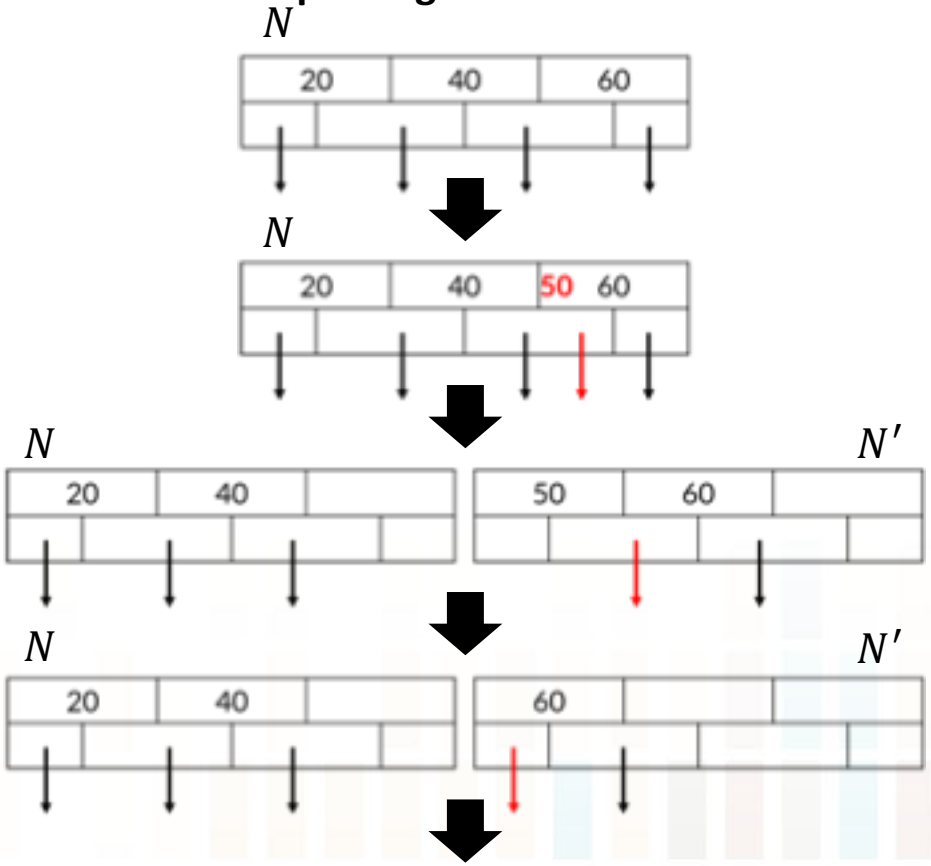
B-Tree Insertion/Splitting: Cases by Examples

Case 1: Splitting-- Leaf Node



Next: Insert P' (pointer of N') to parent.

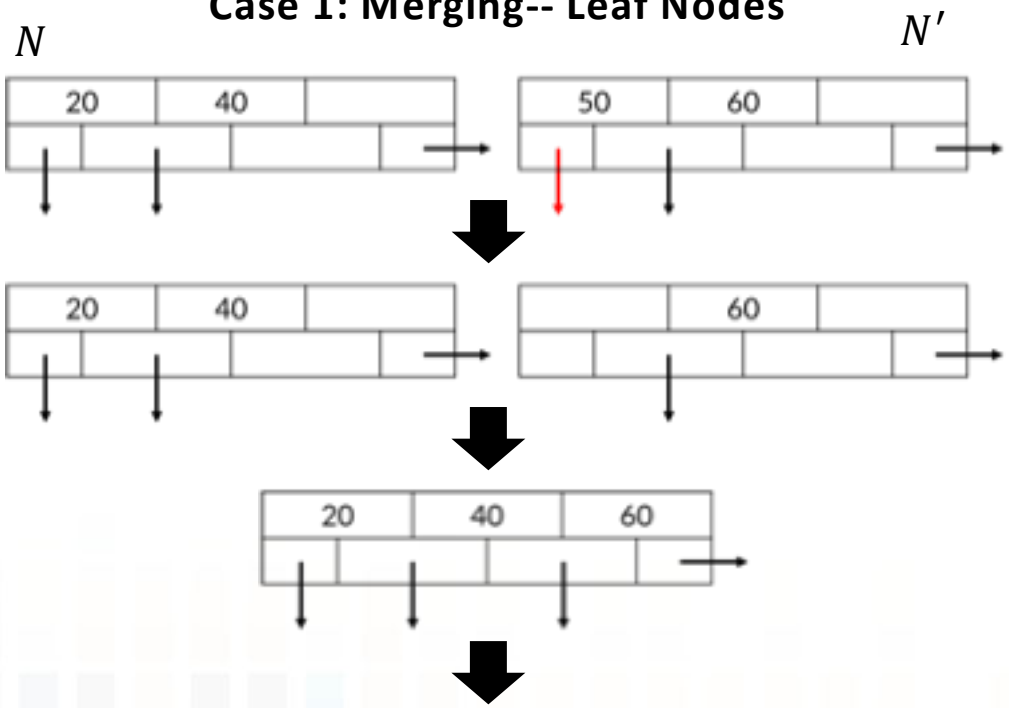
Case 2: Splitting– Internal Node



Next: Insert P' (pointer of N') to parent.

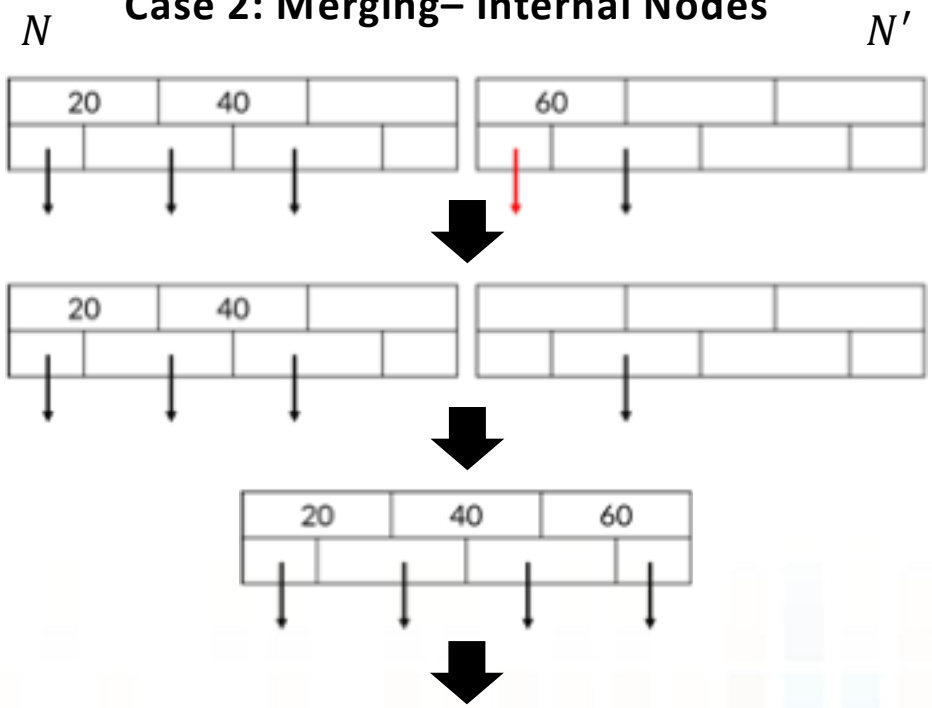
B-Tree Deletion/Merging: Cases by Examples

Case 1: Merging-- Leaf Nodes

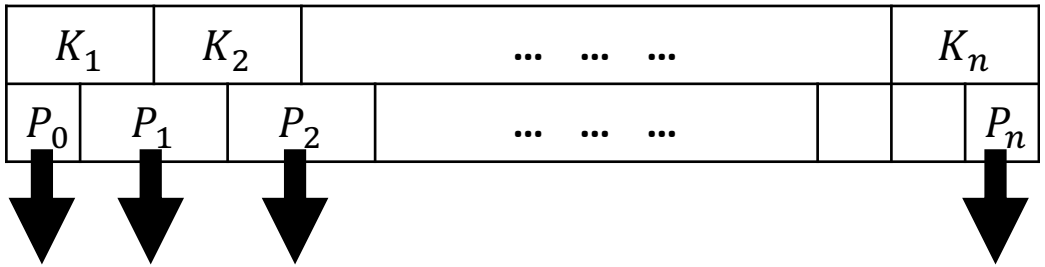


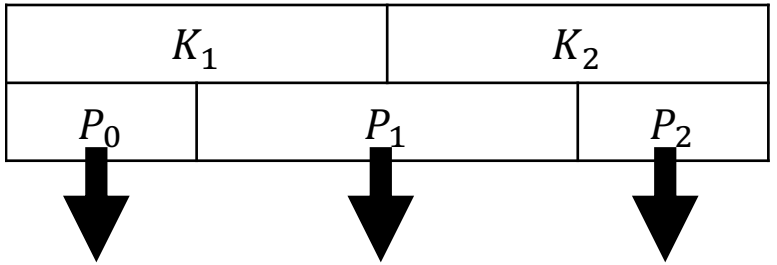
Next: Remove P' (pointer of N') from parent.

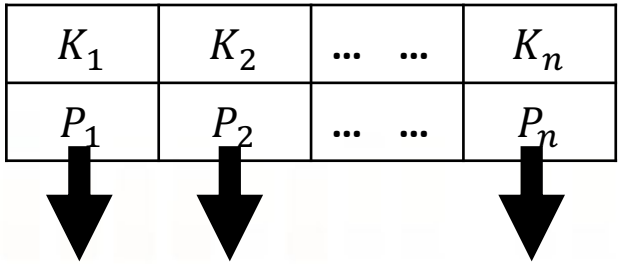
Case 2: Merging– Internal Nodes

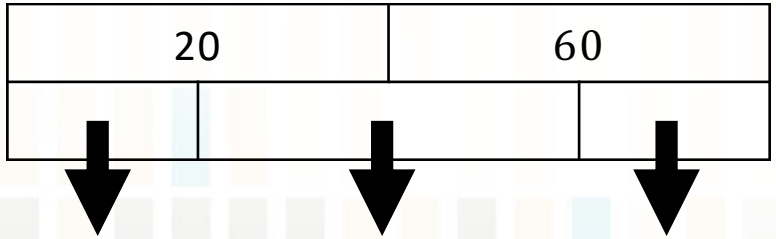


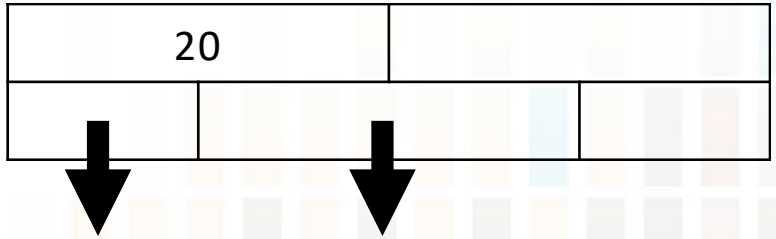
Next: Remove P' (pointer of N') from parent.

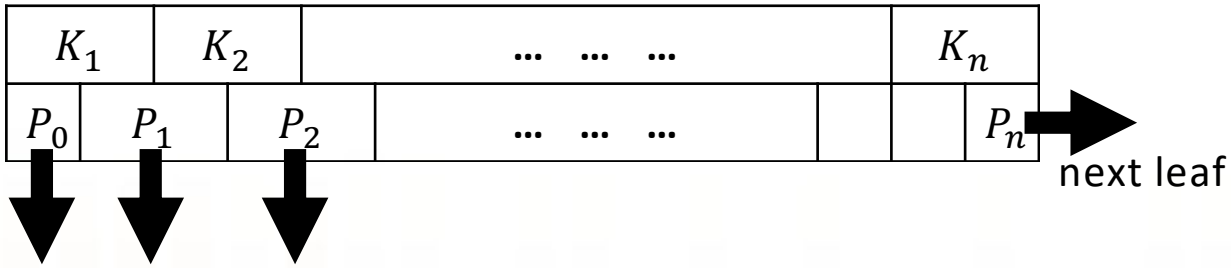

K_1 K_2 ... ... K_n
P_0 P_1 P_2 ... ... P_n

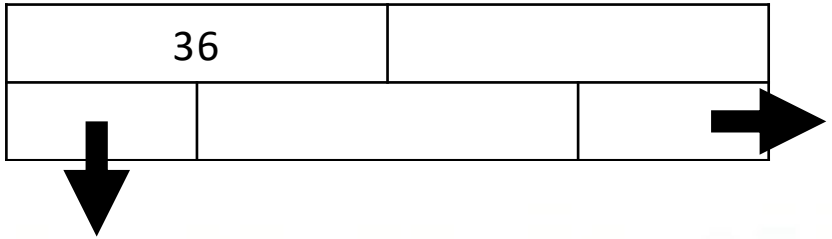

K_1 K_2
P_0 P_1 P_2

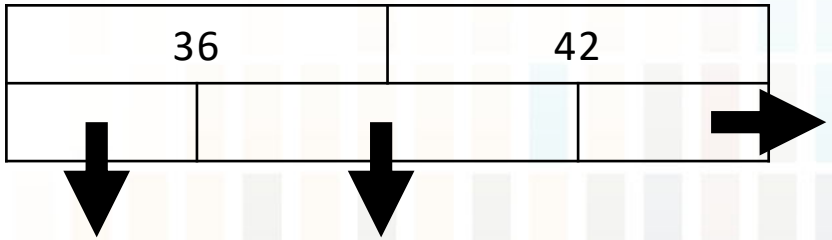

K_1 K_2 ... ... K_n
P_1 P_2 ... ... P_n

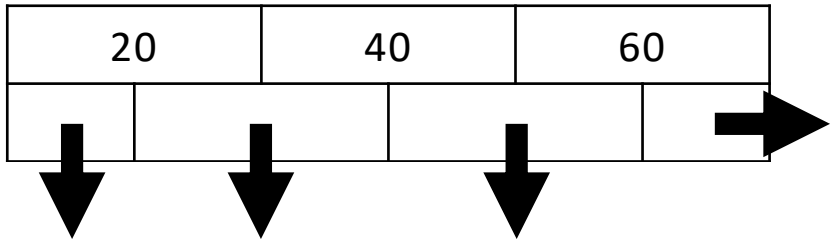

20 60

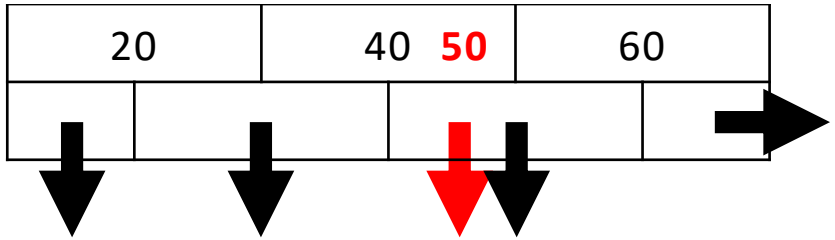

20

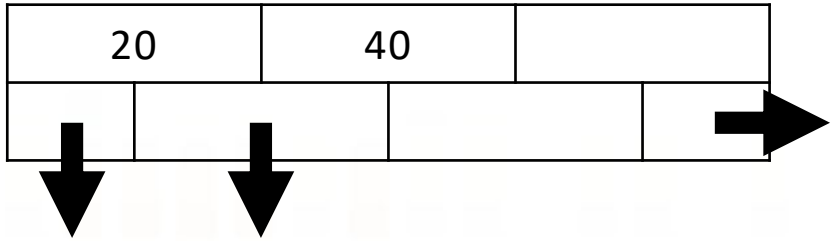

K_1
K_2
... ...
K_n
P_0
P_1
P_2
... ...
P_n
next leaf

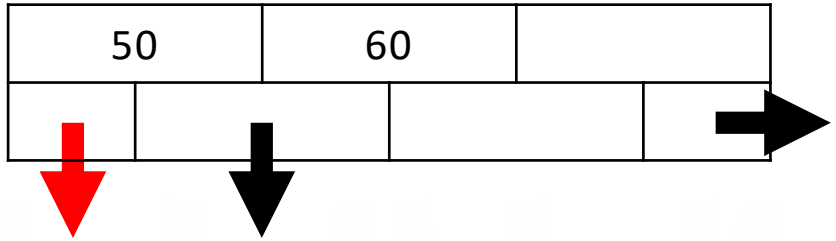

36

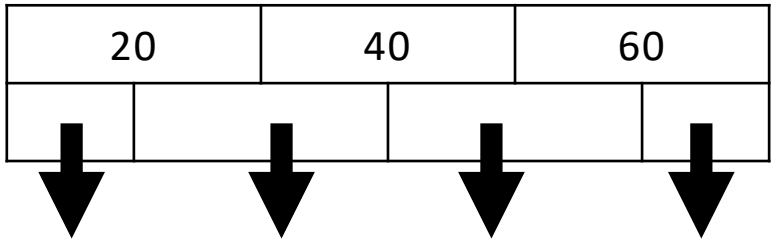

36
42

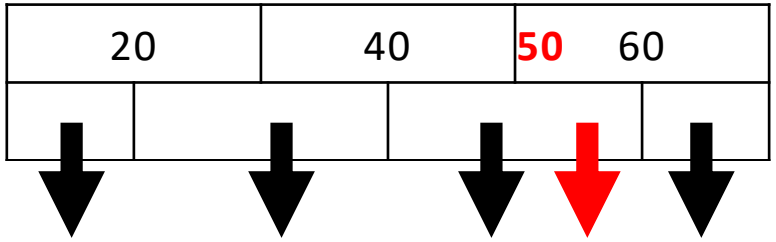

20
40
60

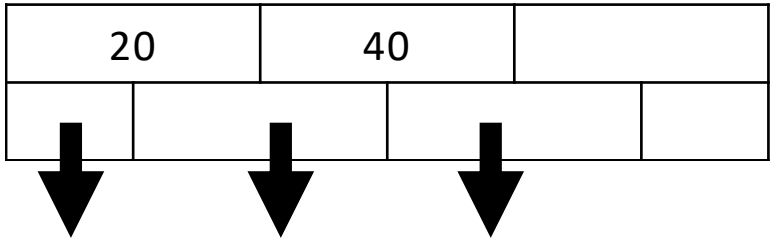

20
40
50
60

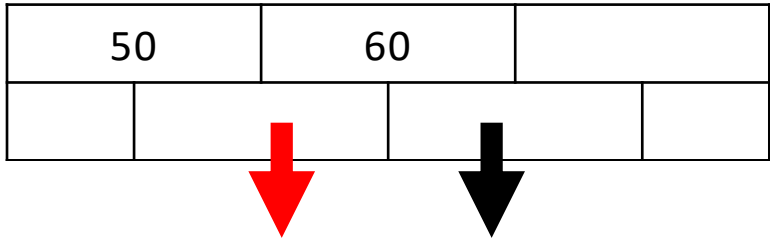

20
40

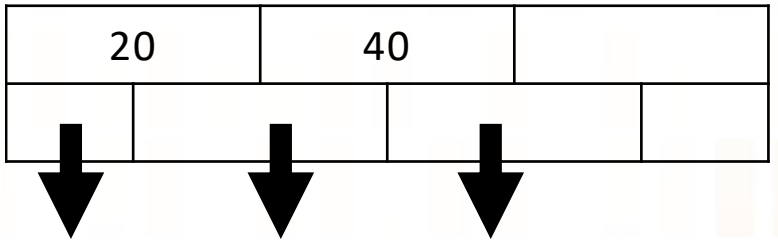

50
60


20
40
60

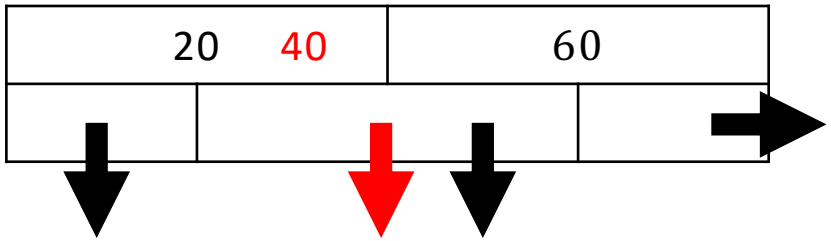

20
40
50
60

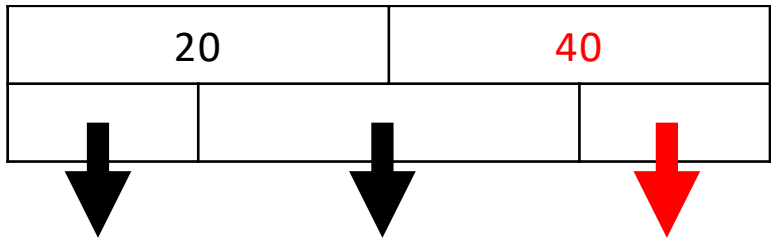

20
40

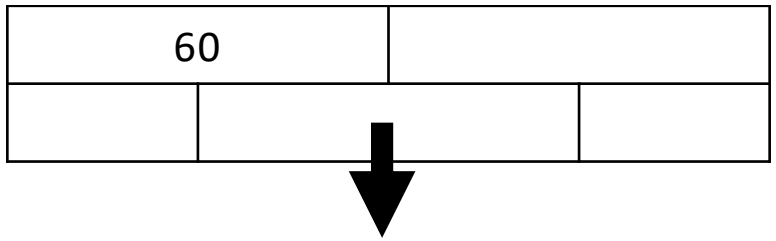

50
60

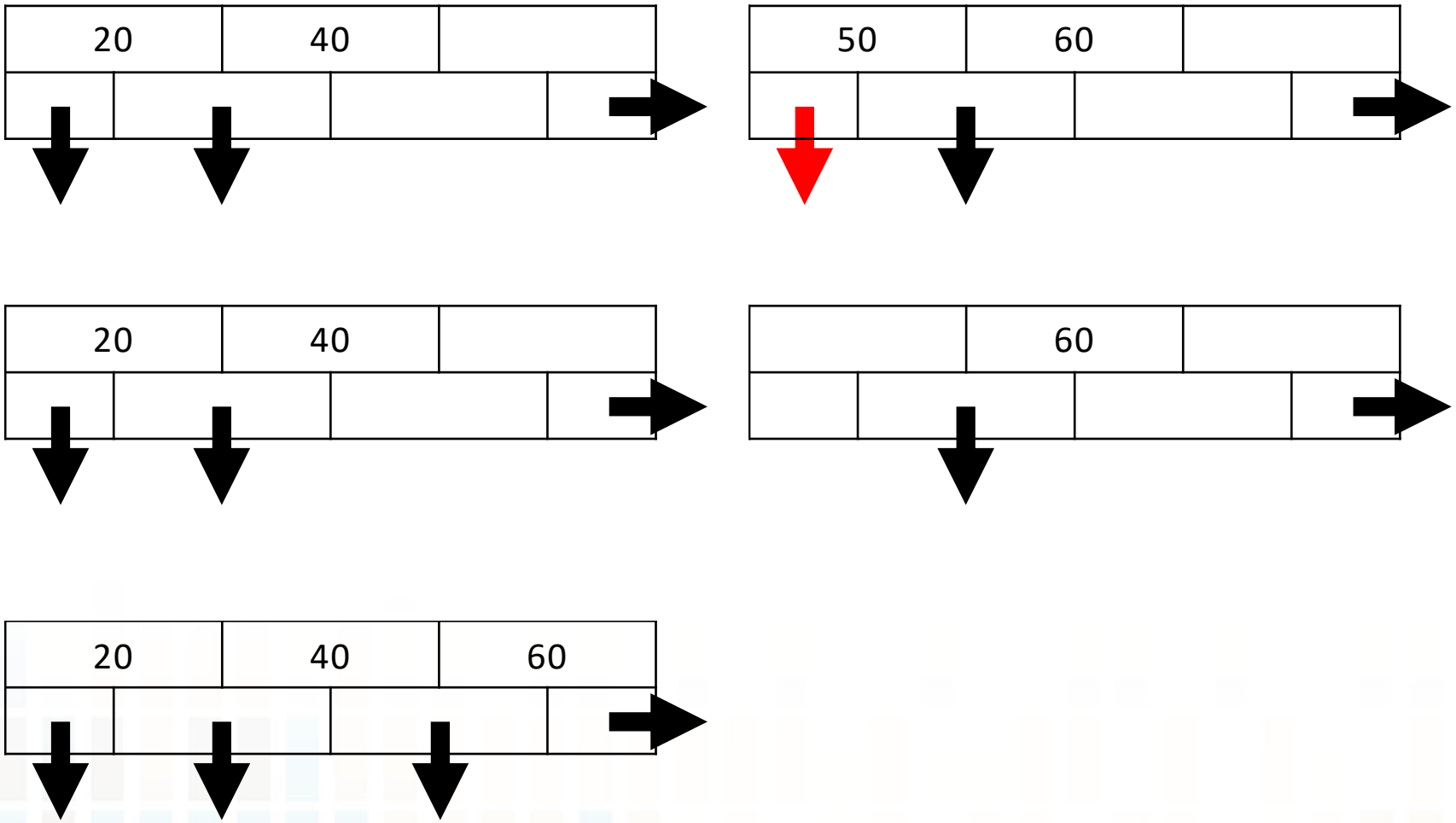

20
40

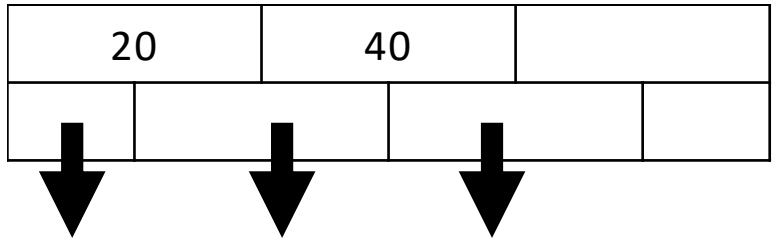

60

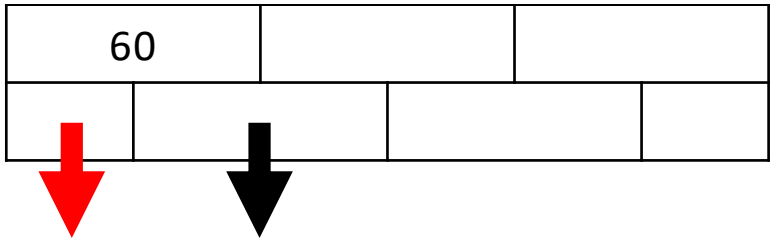

20
40
60

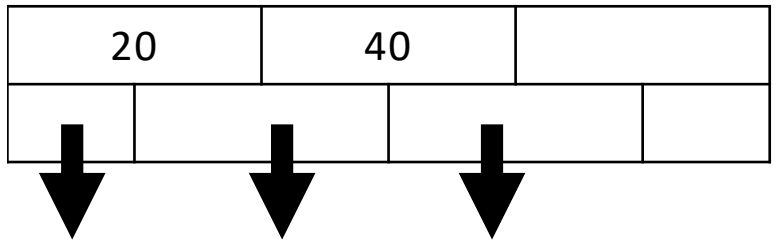

20
40

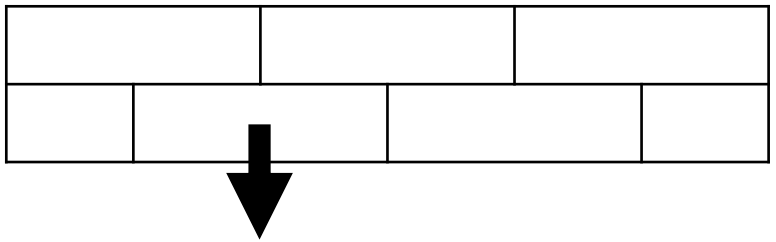

60

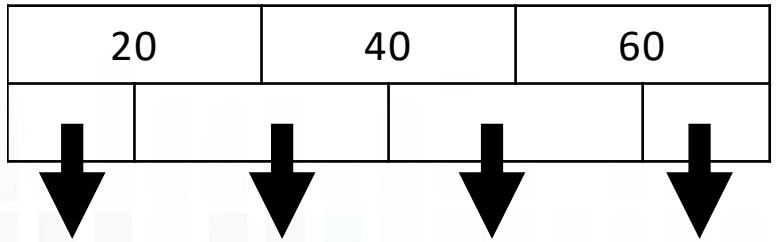

20
40
50
60
20
40
60
20
40
60

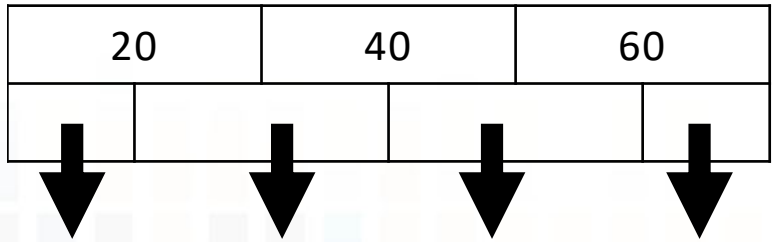

20
40


60


20
40




20
40
60


20
40
60

Figure

Degree (d)	(Internal) Min Keys	(Leaf) Min Keys	Max Keys (n)
1	1	1	2
1.5	1	2	3
2	2	2	4
2.5	2	3	5
3	3	3	6
3.5	3	4	7
4	4	4	8
4.5	4	5	9
5	5	5	10

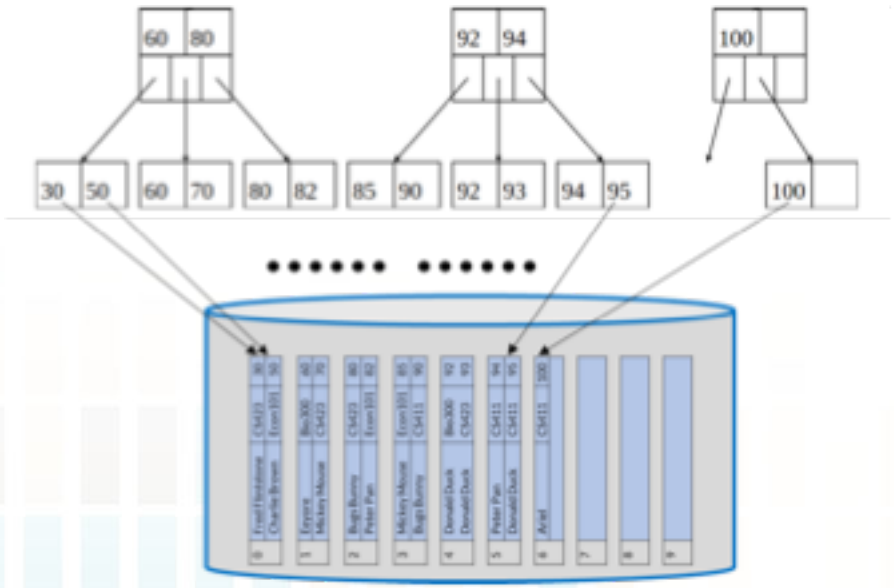
Figure

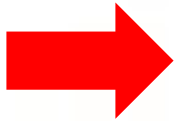
Ed McCreight answered a question on B-tree's name in 2013:

> Bayer and I were in a lunchtime where we get to think [of] a name. And ... B is, you know ... We were working for Boeing at the time, we couldn't use the name without talking to lawyers. So, there is a B. [The B-tree] has to do with balance, another B. Bayer was the senior author, who [was] several years older than I am and had many more publications than I did. So there is another B. And so, at the lunch table we never did resolve whether there was one of those that made more sense than the rest. What really lives to say is: the more you think about what the B in B-trees means, the better you understand B-trees."[5]

B-tree etymology. B-tree. Retrieved from https://en.wikipedia.org/wiki/B-tree.

Add One More Level – And Done!



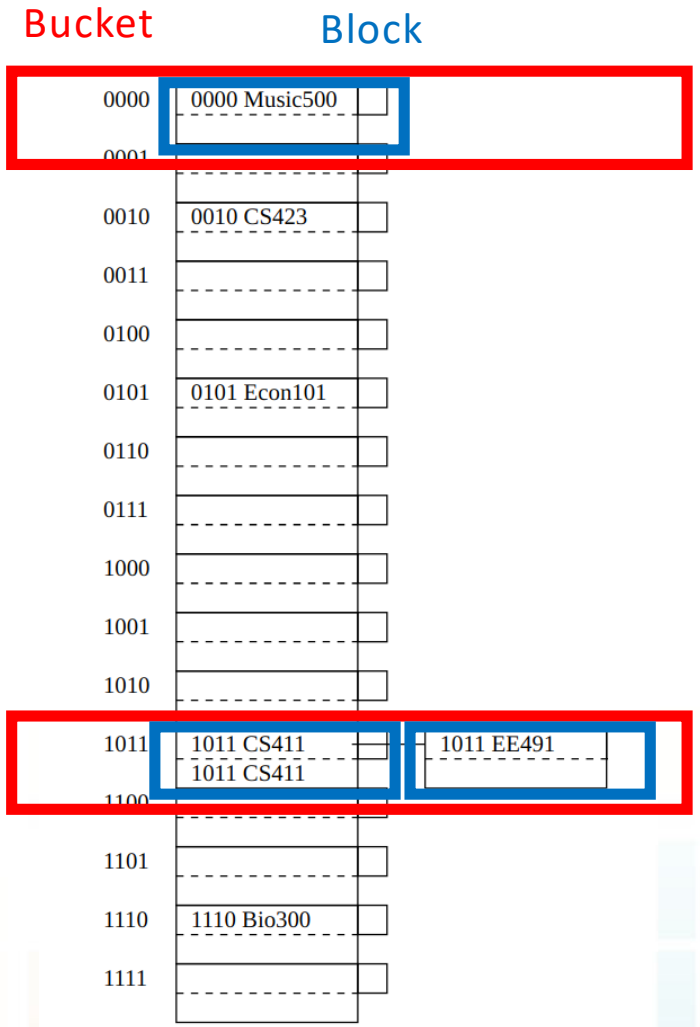


Example ISAM index

Figure

Key	Hash Code
CS423	0010
CS411	1011
Bio300	1110
Music500	0000
Econ101	0101
EE491	1011

Example hash function



Figure

Feature	Extensible Hash Table	Linear Hash Table
Used Bits	MSB	LSB
Directory	Yes	No
Flip Bits in Lookup	**Yes, flip lowest bits**	**Yes, flip highest bits**
Lookup in Buckets	Check **1** Bucket(s)	Check 1 or 2 Bucket(s)
Increment of Growth	2X directory. +1 ~ 2^i buckets	+1 bucket
Overflow Blocks	**No**	Yes
Growth Criteria	**When a bucket is too full**	**When average utilization too large**
How New Bucket Added	**Split from the to-be-inserted bucket**	**Add next-in-sequence bucket**

Extra

Food for Thought

Did you notice that the students just (strangely) have no "duplicate" grades. Can ISAM handle if there are duplicates?

0	Bugs Bunny	CS411	90
	Donald Duck	Bio300	92
1	Donald Duck	CS423	93
	Donald Duck	CS411	95
2	Bugs Bunny	CS423	80
	Mickey Mouse	CS423	70
3	Peter Pan	CS411	94
	Charlie Brown	Econ101	50
4	Peter Pan	Econ101	82
	Eeyore	Bio300	60
5	Mickey Mouse	Econ101	85
	Ariel	CS411	100
6	Fred Flintstone	CS423	30
7			
8			
9			

Our example Enrolls table

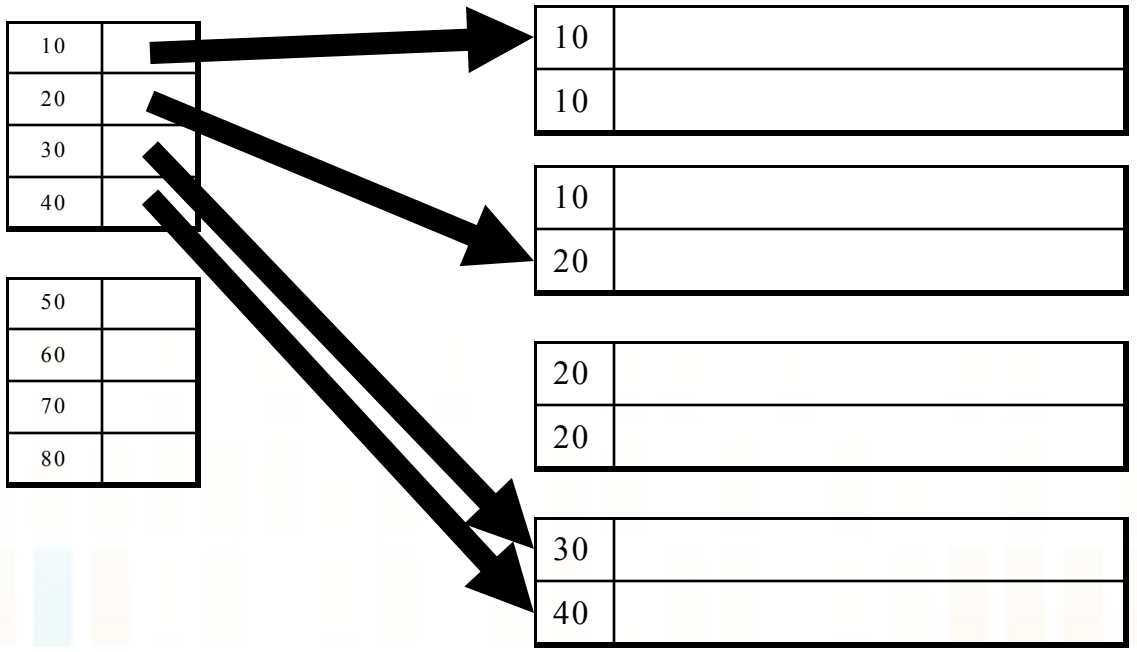
0	Fred Flintstone	CS423	30
	Charlie Brown	Econ101	50
1	Eeyore	Bio300	60
	Mickey Mouse	CS423	70
2	Bugs Bunny	CS423	80
	Peter Pan	Econ101	80
3	Mickey Mouse	Econ101	80
	Bugs Bunny	CS411	90
4	Donald Duck	Bio300	90
	Donald Duck	CS423	90
5	Peter Pan	CS411	95
	Donald Duck	CS411	95
6	Ariel	CS411	100
7			
8			
9			

Another example Enrolls table, with duplicate grades

How if duplicate keys?

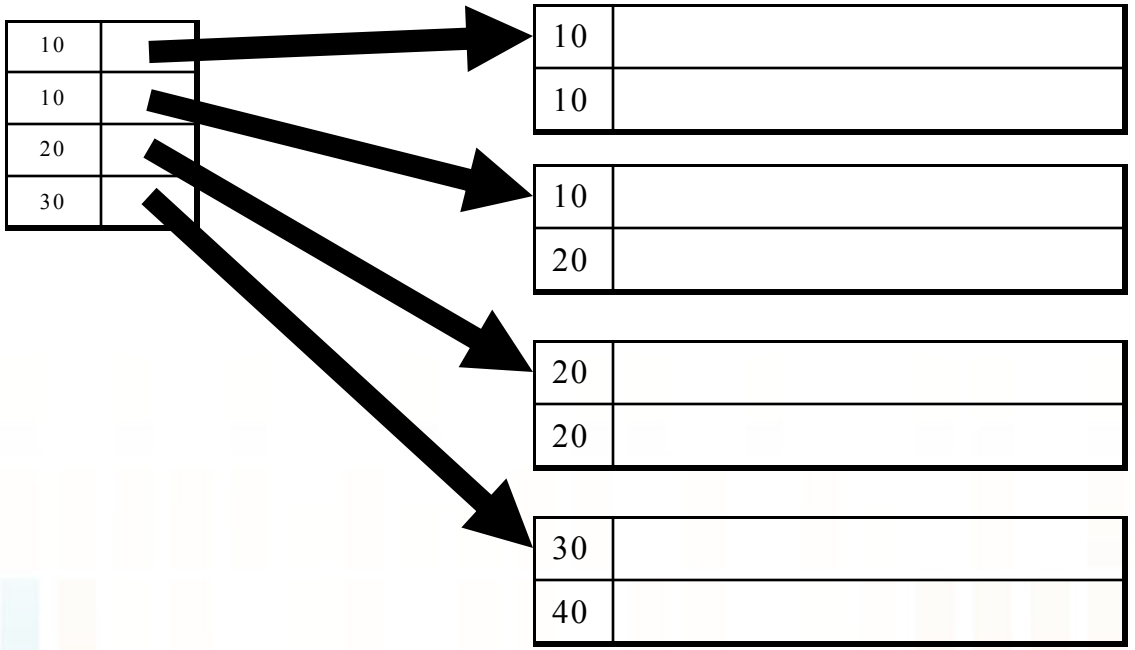
Clustered Index with Duplicate Keys

- Dense index: point to the first record with that key



Clustered Index with Duplicate Keys

- Sparse index: pointer to lowest search key in each block:



- OK?

  Try search for 20

Clustered Index with Duplicate Keys

- Better: pointer to lowest new search key in each block:
- Search for 20