

CSE 444: Database Internals

Lectures 5-6 Indexing

CSE 444 - Spring 2019

1

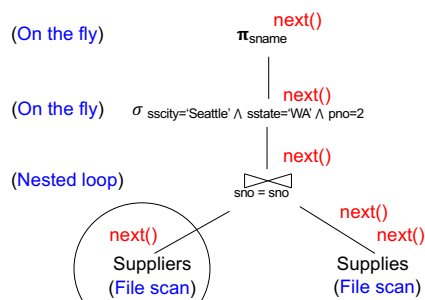
Announcements

- HW1 due Friday
 - Turn in a hard copy before/after class or during office hour.
- Lab1 is due on Wednesday, 11pm
- 544M first reading due Friday... but flexible

CSE 444 - Spring 2019

2

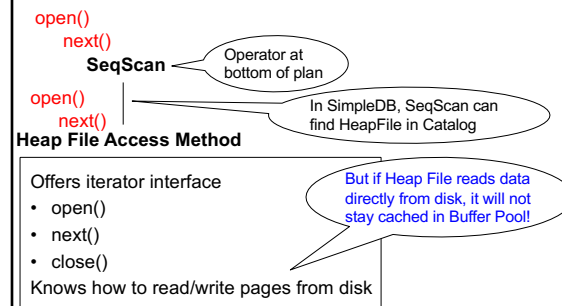
Query Execution How it all Fits Together



CSE 444 - Spring 2019

3

Query Execution In SimpleDB



CSE 444 - Spring 2019

4

Basic Access Method: Heap File

API

- **Create** or **destroy** a file
- **Insert** a record
- **Delete** a record with a given rid (rid)
 - rid: unique tuple identifier (more later)
- **Get** a record with a given rid
 - Not necessary for sequential scan operator
 - But used with indexes
- **Scan** all records in the file

CSE 444 - Spring 2019

5

But Often Also Want....

- **Scan** all records in the file that match a **predicate** of the form **attribute op value**
 - Example: Find all students with GPA > 3.5
- Critical to support such requests efficiently
 - Why read all data from disk when we only need a small fraction of that data?
- This lecture and next, we will learn how

CSE 444 - Spring 2019

6

Searching in a Heap File

File is **not sorted** on any attribute

Student(sid: int, age: int, ...)

30	18	...
70	21	
20	20	
40	19	
80	19	
60	18	
10	21	
50	22	

— 1 record

} 1 page

CSE 444 - Spring 2019

7

Heap File Search Example

- 10,000 students
- 10 student records per page
- **Total number of pages: 1,000 pages**
- Find student whose sid is 80
 - **Must read on average 500 pages**
- Find all students older than 20
 - **Must read all 1,000 pages**
- **Can we do better?**

CSE 444 - Spring 2019

8

Sequential File

File **sorted on an attribute**, usually on primary key

Student(sid: int, age: int, ...)

10	21	...
20	20	
30	18	
40	19	
50	22	
60	18	
70	21	
80	19	

CSE 444 - Spring 2019

9

Sequential File Example

- Total number of pages: 1,000 pages
- Find student whose sid is 80
 - **Could do binary search, read $\log_2(1,000) \approx 10$ pages**
- Find all students older than 20
 - **Must still read all 1,000 pages**
- **Can we do even better?**
- **Note: Sorted files are inefficient for inserts/deletes**

CSE 444 - Spring 2019

10

Creating Indexes in SQL

```
CREATE TABLE V(M int, N varchar(20), P int);
```

```
CREATE INDEX V1 ON V(N)
```

```
CREATE INDEX V2 ON V(P, M)
```

```
select *
from V
where P=55 and M=77
```

```
select *
from V
where P=55
```

CSE 444 - Spring 2019

11

Outline

- **Index structures**
- **Hash-based indexes**
- **B+ trees**

} Today

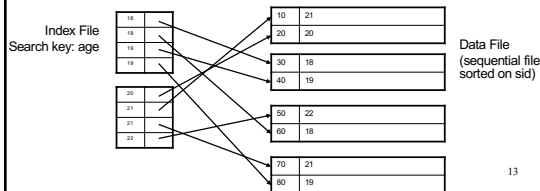
} Next time

CSE 444 - Spring 2019

12

Indexes

- **Index**: data structure that organizes data records on disk to optimize selections on the **search key fields** for the index
- An index contains a collection of **data entries**, and supports efficient retrieval of all data entries with a given search key value **k**
- **Indexes are also access methods!**
 - So they provide the same API as we have seen for Heap Files
 - And efficiently support scans over tuples matching predicate on search key



13

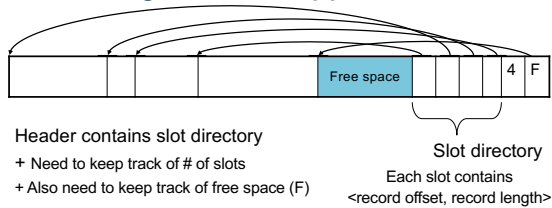
Indexes

- **Search key** = can be any set of fields
 - not the same as the primary key, nor a key
- **Index** = collection of data entries
- **Data entry** for key **k** can be:
 - **(k, RID)**
 - **(k, list-of-RIDs)**
 - **The actual record with key k**
 - In this case, **the index is also a special file organization**
 - Called: "indexed file organization"

CSE 444 - Spring 2019

14

Page Format Approach 2



Can handle variable-length records
Can move tuples inside a page without changing RIDs
RID is (PageID, SlotID) combination

CSE 444 - Spring 2019

15

Different Types of Files

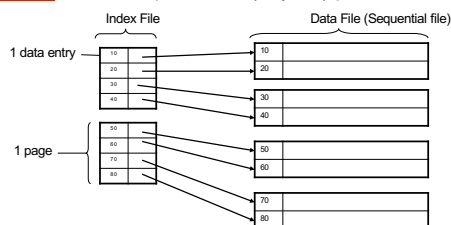
- For the data inside base relations:
 - **Heap file** (tuples stored without any order)
 - **Sequential file** (tuples sorted on some attribute(s))
 - **Indexed file** (tuples organized following an index)
- Then we can have additional **index files** that store (key,rid) pairs
- Index can also be a "**covering index**"
 - Index contains (search key + other attributes, rid)
 - Index suffices to answer some queries

CSE 444 - Spring 2019

16

Primary Index

- **Primary index** determines location of indexed records
- **Dense index**: sequence of (key,rid) pairs

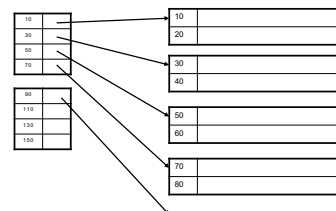


CSE 444 - Spring 2019

17

Primary Index

- **Sparse index**

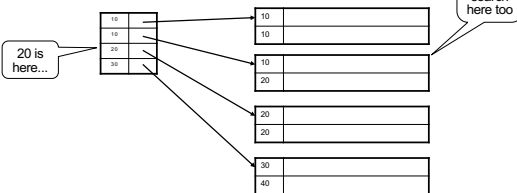


CSE 444 - Spring 2019

18

Primary Index with Duplicate Keys

- Sparse index: pointer to lowest search key on each page: Example search for 20

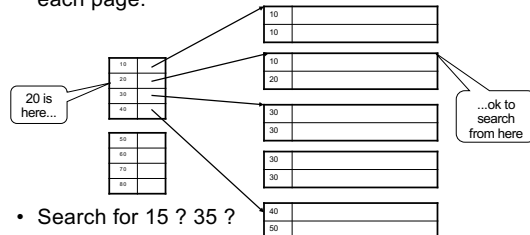


CSE 444 - Spring 2019

19

Primary Index with Duplicate Keys

- Better: pointer to **lowest new search key** on each page:



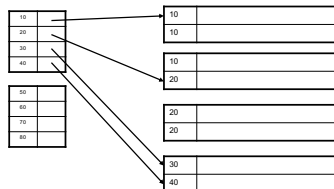
- Search for 15 ? 35 ?

CSE 444 - Spring 2019

20

Primary Index with Duplicate Keys

- Dense index:



CSE 444 - Spring 2019

21

Primary Index: Back to Example

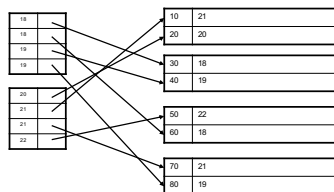
- Let's assume all pages of index fit in memory
- Find student whose sid is 80
 - Index (dense or sparse) points directly to the page
 - Only need to read 1 page from disk.
- Find all students older than 20
- How can we make *both* queries fast?

CSE 444 - Spring 2019

22

Secondary Indexes

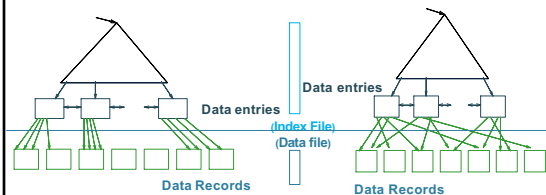
- Do not determine placement of records in data files
- Always dense (why ?)



CSE 444 - Spring 2019

23

Clustered vs. Unclustered Index



CLUSTERED

UNCLUSTERED

Clustered = records close in index are close in data

CSE 444 - Spring 2019

24

Clustered/Unclustered

- Primary index = clustered by definition
- Secondary indexes = usually unclustered

Secondary Indexes

- Applications
 - Index unsorted files (heap files)
 - When necessary to have multiple indexes
 - Index files that hold data from two relations
 - Called “clustered file”
 - Notice the different use of the term “clustered”!

Index Classification Summary

- Primary/secondary
 - Primary = determines the location of indexed records
 - Secondary = cannot reorder data, does not determine data location
- Dense/sparse
 - Dense = every key in the data appears in the index
 - Sparse = the index contains only some keys
- Clustered/unclustered
 - Clustered = records close in index are close in data
 - Unclustered = records close in index may be far in data
- B+ tree / Hash table / ...

Ex1. Primary Dense Index (tid)

tid	user	time	content
10	2	05:03:00	"..."
20	1	12:05:07	"..."
30	2	18:12:00	"..."
40	3	00:16:13	"..."
50	4	10:10:13	"..."
60	1	04:09:07	"..."
70	2	12:08:34	"..."
80	4	11:08:09	"..."

- **Dense:** an “index key” for every database record
 - (In this case) every “database key” appears as an “index key”
- **Primary:** determines the location of indexed records
- Also, **Clustered:** records close in index are close in data

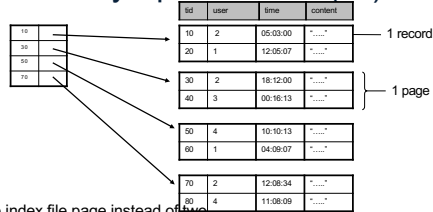
Improve from Primary Clustered Index?

Clustered Index can be made Sparse
(normally one key per page)

Ex2. Draw a primary sparse index on “tid”

tid	user	time	content
10	2	05:03:00	"..."
20	1	12:05:07	"..."
30	2	18:12:00	"..."
40	3	00:16:13	"..."
50	4	10:10:13	"..."
60	1	04:09:07	"..."
70	2	12:08:34	"..."
80	4	11:08:09	"..."

Ex2. Primary Sparse Index (tid)



- Only one index file page instead of two

CSE 444 - Spring 2019

31

Large Indexes

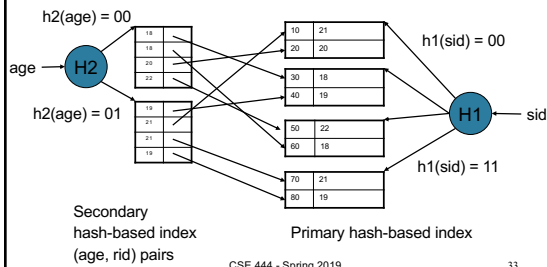
- What if index does not fit in memory?
- Would like to index the index itself
 - Hash-based index
 - Tree-based index

CSE 444 - Spring 2019

32

Hash-Based Index

Good for point queries but not range queries



CSE 444 - Spring 2019

33

Tree-Based Index

- How many index levels do we need?
- Can we create them automatically? **Yes!**
- Can do something even more powerful!

CSE 444 - Spring 2019

34

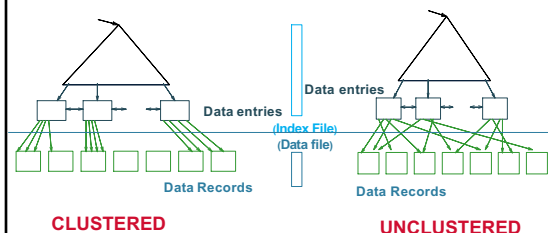
B+ Trees

- Search trees
- Idea in B Trees
 - Make 1 node = 1 page (= 1 block)
- Idea in B+ Trees
 - Keep tree balanced in height – dynamic rather than static
 - Make leaves into a linked list : facilitates range queries

CSE 444 - Spring 2019

35

B+ Trees

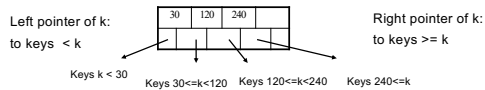


CSE 444 - Spring 2019

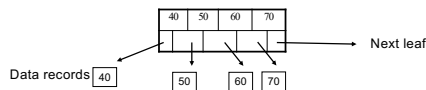
36

B+ Trees Basics

- Parameter d = the degree
- Each node has $d \leq m \leq 2d$ keys (except root)
- Each node also has $m+1$ pointers



- Each leaf has $d \leq m \leq 2d$ keys:



CSE 444 - Spring 2019

37

B+ Trees Properties

- For each node except the root, maintain 50% occupancy of keys
- Insert and delete must rebalance to maintain constraints

CSE 444 - Spring 2019

38

Searching a B+ Tree

- Exact key values:
 - Start at the root
 - Proceed down, to the leaf
- Range queries:
 - Find lowest bound as above
 - Then sequential traversal

Select name
From Student
Where age = 25

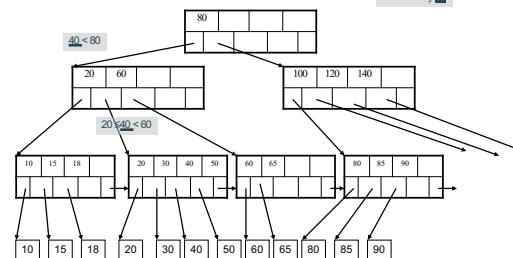
Select name
From Student
Where $20 \leq \text{age}$
and $\text{age} \leq 30$

CSE 444 - Spring 2019

39

B+ Tree Example

$d = 2$



CSE 444 - Spring 2019

40

B+ Tree Design

- How large d ?
- Example:
 - Key size = 4 bytes
 - Pointer size = 8 bytes
 - Block size = 4096 bytes
- $2d \times 4 + (2d+1) \times 8 \leq 4096$
- $d = 170$

CSE 444 - Spring 2019

41

B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
 - average fanout = 133
- Typical capacities
 - Height 4: $133^4 = 312,900,700$ records
 - Height 3: $133^3 = 2,352,637$ records
- Can often hold top levels in buffer pool
 - Level 1 = 1 page = 8 Kbytes
 - Level 2 = 133 pages = 1 Mbyte
 - Level 3 = 17,689 pages = 133 Mbytes

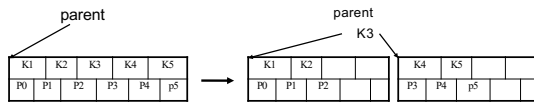
CSE 444 - Spring 2019

42

Insertion in a B+ Tree

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:



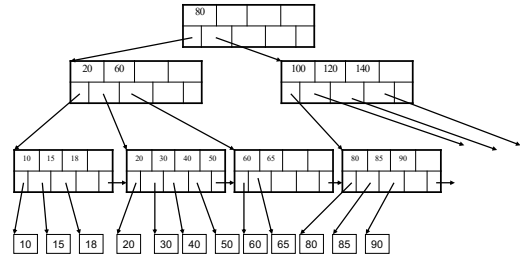
- If leaf, also keep K3 in right node
- When root splits, new root has 1 key only

CSE 444 - Spring 2019

43

Insertion in a B+ Tree

Insert K=19

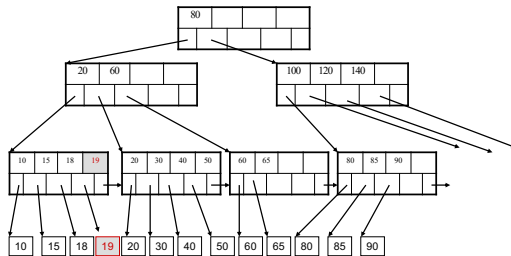


CSE 444 - Spring 2019

44

Insertion in a B+ Tree

After insertion

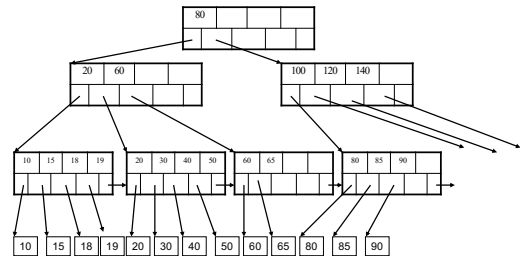


CSE 444 - Spring 2019

45

Insertion in a B+ Tree

Now insert 25

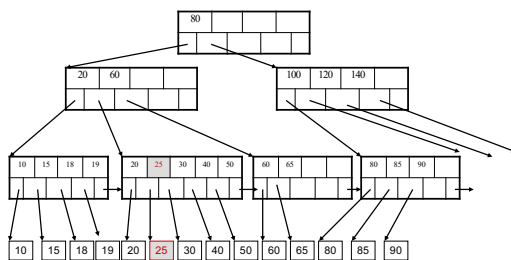


CSE 444 - Spring 2019

46

Insertion in a B+ Tree

After insertion

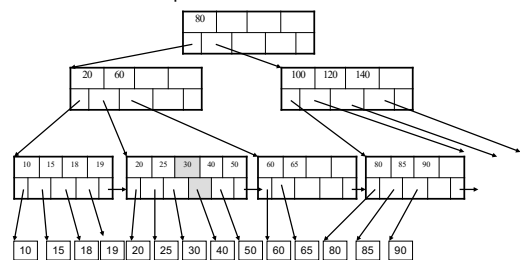


CSE 444 - Spring 2019

47

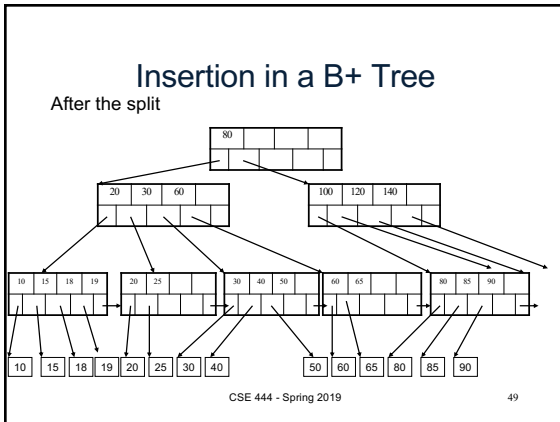
Insertion in a B+ Tree

But now have to split !



CSE 444 - Spring 2019

48

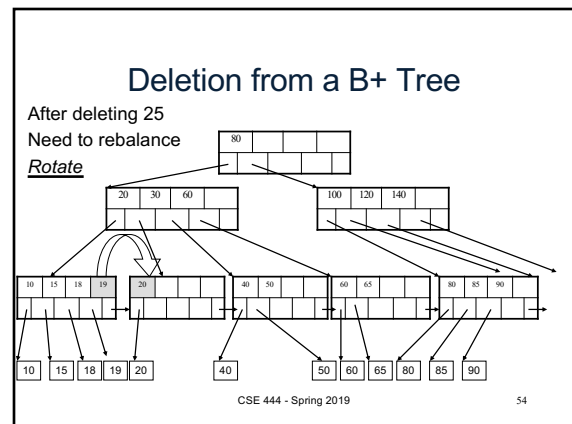
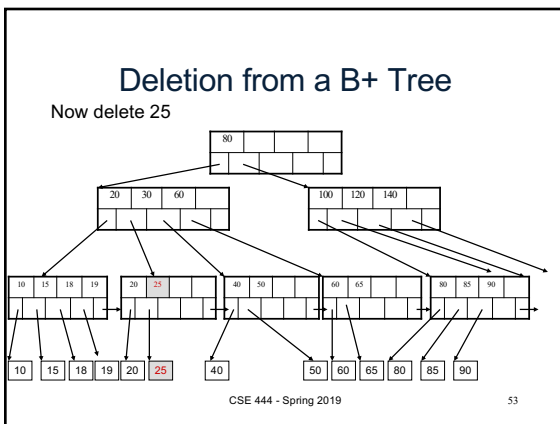
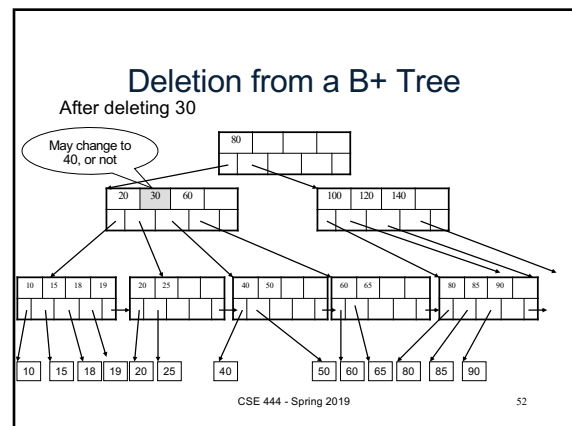
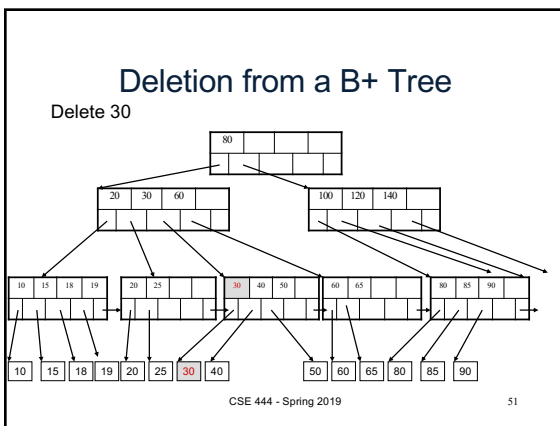


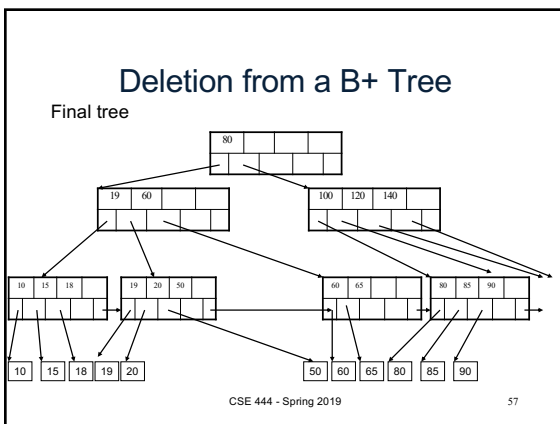
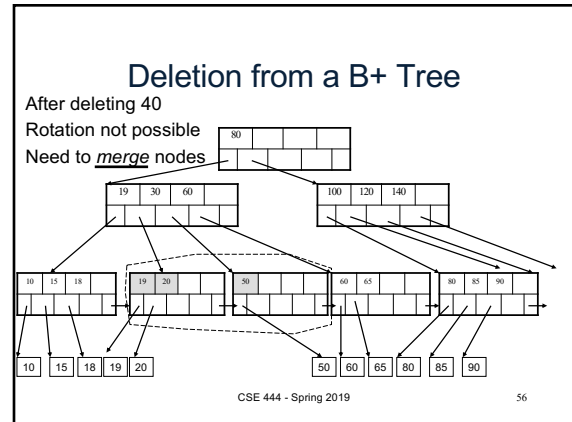
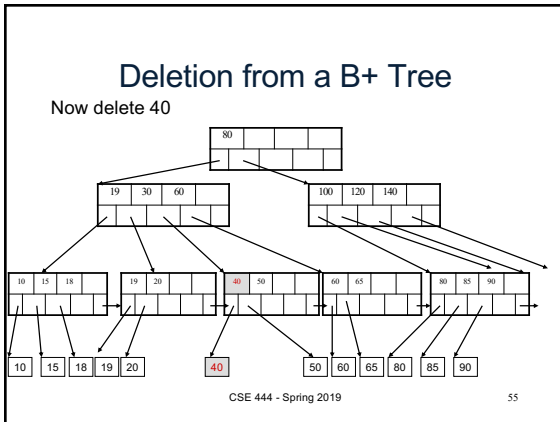
Deletion in a B+ Tree

Delete (K, P)

- Find leaf where K belongs, delete
- Check for capacity
- If leaf below capacity, search adjacent nodes (left first, then right) for extra tuples and rotate them to new leaf
- If adjacent nodes at 50% full, merge
- Update and repeat algorithm on parent nodes if necessary

CSE 444 - Spring 2019 50





Summary on B+ Trees

- Default index structure on most DBMSs
- Very effective at answering 'point' queries:
productName = 'gizmo'
- Effective for range queries:
50 < price AND price < 100
- Less effective for multirange:
50 < price < 100 AND 2 < quant < 20

CSE 444 - Spring 2019 58