Physical database design

CMT220
Databases & Modelling

Cardiff School of **Computer Science & Informatics**

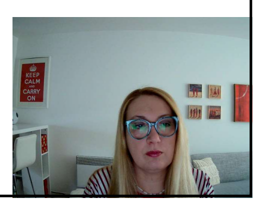
http://www.cs.cf.ac.uk

1

1

Lecture

- in the last few lectures, we looked into database design issues
- in particular, we studied functional dependencies and normalisation
- in this lecture we consider a range of issues concerning physical database design, i.e. issues concerning database performance



2

Physical database design

- physical database design involves:
  - translating a logical design (tables) into a technical specification for storing and retrieving data
  - creating a design that will provide adequate performance and ensure database integrity and security
  - optimising both processing and space, but processing efficiency is usually more important
- it is about making decisions, not just implementation
  - the decisions made at this stage will have a major impact on data accessibility, response time, etc.

CARDIFF UNIVERSITY PRIFYSGOL CAERDYDD

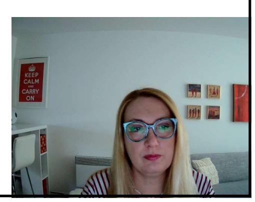
3

3

Data type

- selecting a data type for an attribute so that it
  - minimises storage space
  - represents all possible values
  - improves data integrity
  - supports all data manipulations

CARDIFF UNIVERSITY PRIFYSGOL CAERDYDD



4

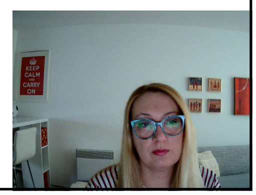
Data coding

- an attribute with a limited number of possible values can be translated into a code

Product#	Description	Finish
P1	chair	C
P2	desk	A
P3	table	C
…	…	…

Code	Value
A	birch
B	maple
C	oak

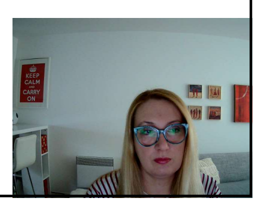
- data coding is good for query performance and storage



5

Indexing

- the relational model says that order does not matter
- from a practical point of view (e.g. finding information fast), order is very important
- indexes are to do with "ordering" data



6

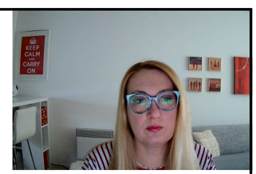
Analogy

Index

- book index is an ordered list of headings and associated pointers to pages where useful material relating to that heading is mentioned
- we do not have to read the whole book page by page in order to find specific information
- instead, using an index we can jump straight to the relevant page

CARDIFF UNIVERSITY PRIFYSGOL CAERDYDD

Index



- an index on an attribute (or a set of attributes) of a table is a data structure that makes it efficient to access values of the attribute

Index

Dept #	Pointer
10002	Record 101
10003	Record 273
10004	Record 33
10005	Record 5
10006	Record 576
10007	Record 11
10008	Record 460
10009	Record 33
1010	Record 411

Table

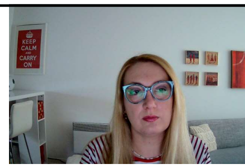
Employee Table

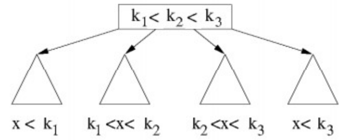
Record #	Social Security	Employee Name	Phone	Dept #
Record 1	708-88-9639	Bailey Workman	555-555-9878	10002
Record 2	030-74-8520	Patricia Spencer	555-555-6321	10002
Record 3	020-87-8852	Jeanette Williams	555-555-7785	10003
Record 4	000-56-9636	Timothy James	555-555-1479	10004
Record 5	000-56-9636	Nicole Kaupp	555-555-0036	10005

The Index is Accessed and the Pointer is Used To Locate The Actual Record in the Table

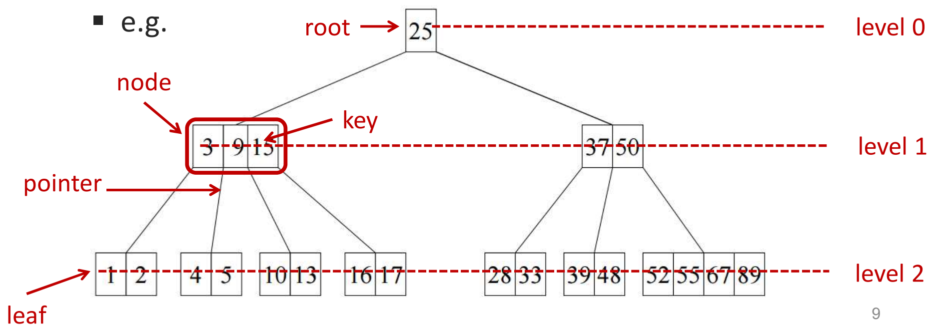
CARDIFF UNIVERSITY PRIFYSGOL CAERDYDD

B–tree





- a widely used indexing structure
- B-tree is a self-balancing tree that keeps data sorted and allows searches, sequential access, insertions and deletions in logarithmic time
- e.g.



9

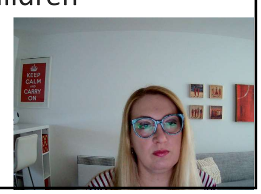
9

B–tree of order *m*

1. all leaf nodes are at the same level
2. each <u>internal</u> node has at least $\lceil m/2 \rceil$ and at most *m* <u>children</u>
3. each <u>leaf</u> node has at least [*m*/2] and at most (*m* – 1) <u>keys</u>
4. each node with *j* children has got (*j* – 1) keys
5. the root is either a leaf or has at least 2 children

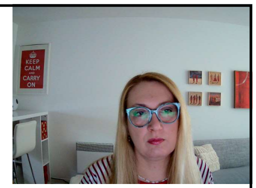
The **ceiling function** [x] of a real number x is the smallest integer that is not smaller than x, e.g. [3.14] = 4.

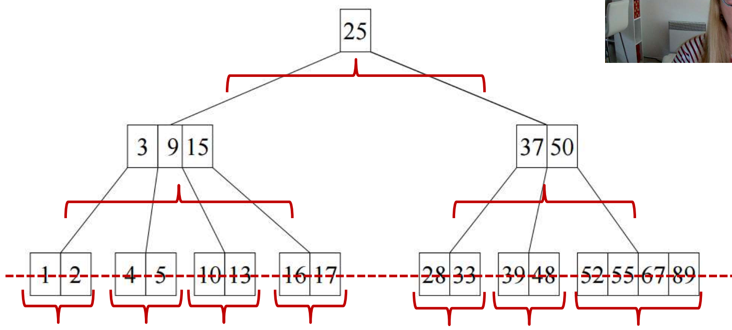
The **integer part** [x] of a real number x is the largest integer y such that |y| ≤ |x|, e.g. [3.14] = 3.



10

Example: B–tree of order 5



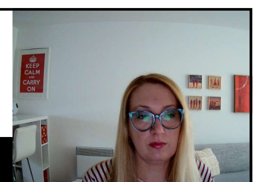


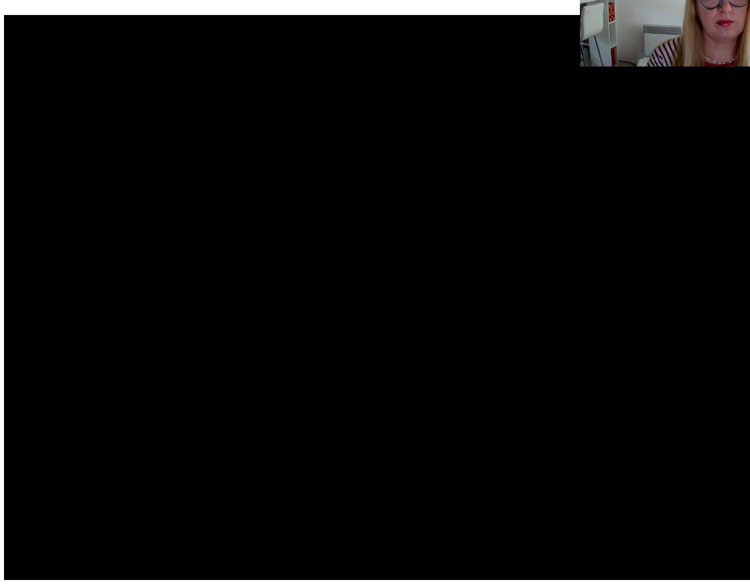
1. all leaf nodes are at the same level
2. each internal node has at least 3 and at most 5 children
3. each leaf node has at least 2 and at most 4 keys
4. each node with j children has got $(j-1)$ keys
5. the root is either a leaf or has at least 2 children

11

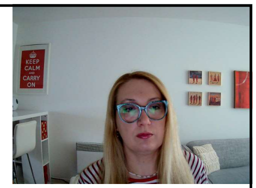
11

Example: B–tree of order 3





12

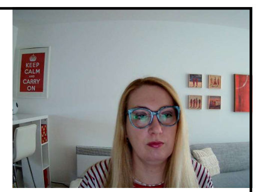


Search in a B–tree

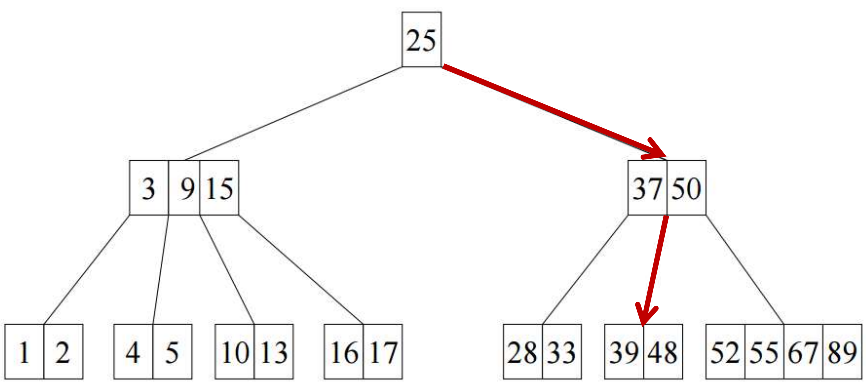
- the search for a record with key x as a recursive function search(node, x)
- if node is null, return null // not found!
- otherwise, iterate through records $(k_1, r_1) \ldots, (k_m, r_m)$ of the node
  - if $x = k_i$, then return r_i // found it!
  - if $x < k_i$, then search($child_i$, x) // search left sub-tree
  - if $k_i < x$, then search($child_{i+1}$, x) // search right sub-tree

13

13



Example: find 40

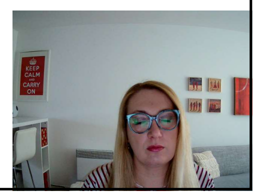


14

14

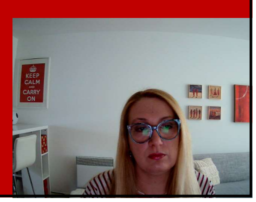
Inserting into a B–tree

- search the B–tree to find the node where the item is to be inserted
- if the node is not full, then insert the item into the node and maintain order
- if the node is full, then it has to be split



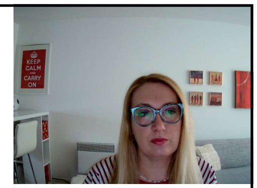
15

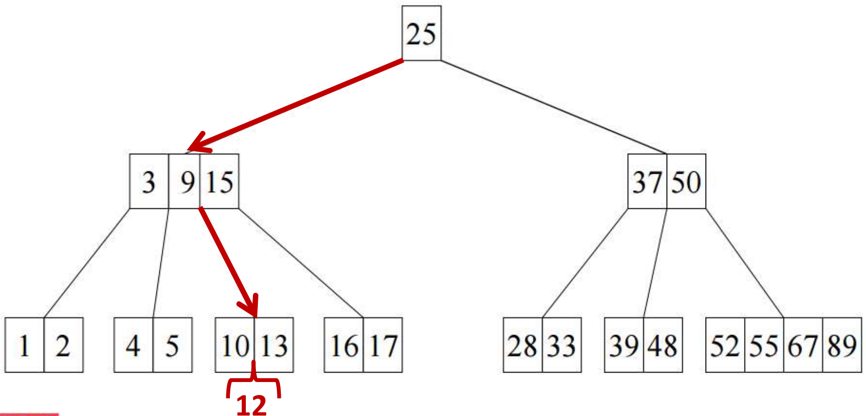
Inserting into a B–tree



16

Example: insert 12

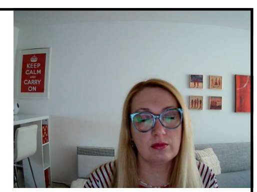


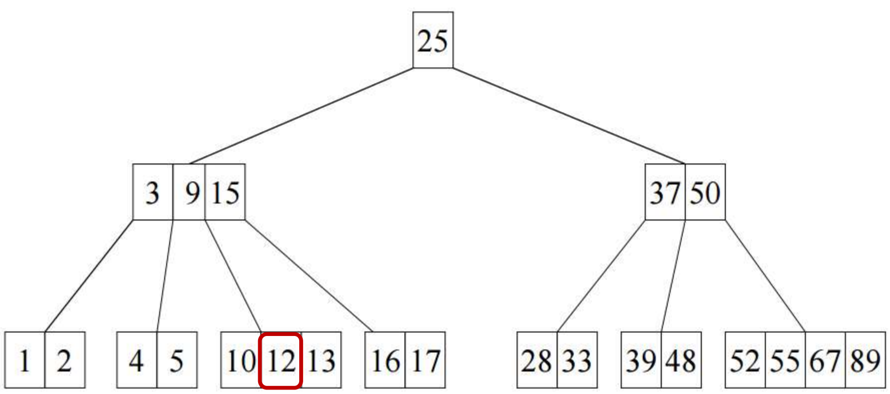


17

17

Example: insert 12

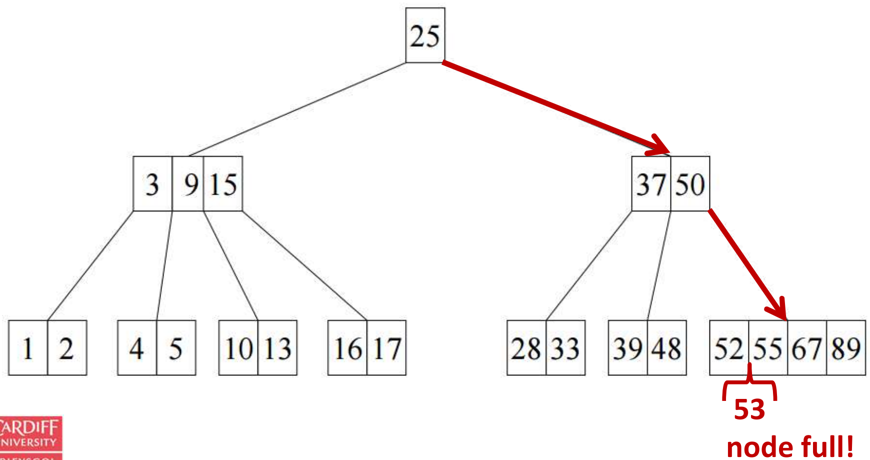




18

18

Example: insert 53

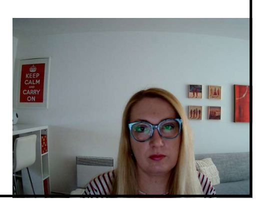


19

19

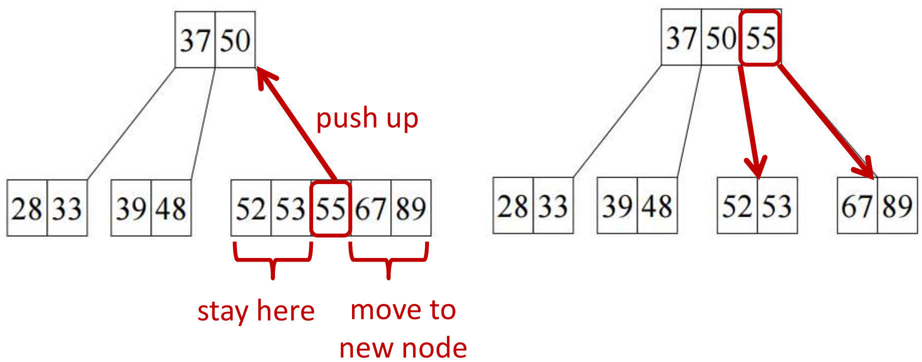
Splitting a node

1. find middle value of old keys in the node and the new key, e.g. 52, 53, 55, 67, 89
2. keep records with keys smaller than middle (e.g. 52, 53) in the old node
3. put records with keys greater than middle (e.g. 67, 89) in the new node
4. push middle record (e.g. 55) up into parent node



20

Splitting a node

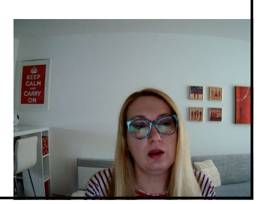


21

21

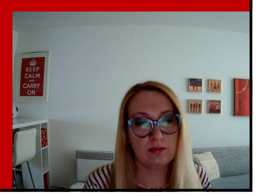
Inserting into a B–tree

- if pushing the middle record up into parent node makes it full (overflow), then split it the parent node and push its middle record upwards
- continue doing this until either:
  - some space is found in an ancestor node, or
  - a new root node is created



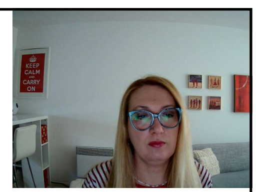
22

Deleting from a B–tree



23

Deleting from a B–tree



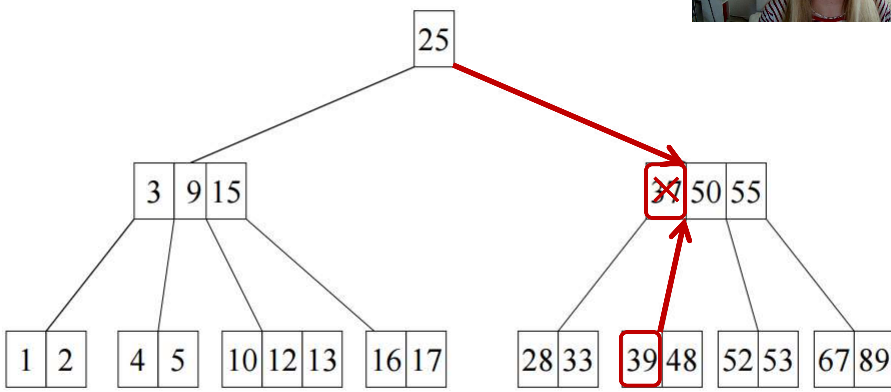
1. search the B–tree to find the node where the item is to be deleted
2. delete the key and replace it by its in–order successor, which will be found in a leaf node
3. remove successor from its leaf node

- this may cause underflow (fewer than $\lceil m/2 \rceil$ records in the node)
- depending on how many records the sibling has, this is fixed either by fusion or by transfer

24

24

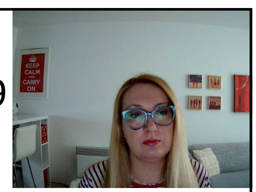
Example: delete 37

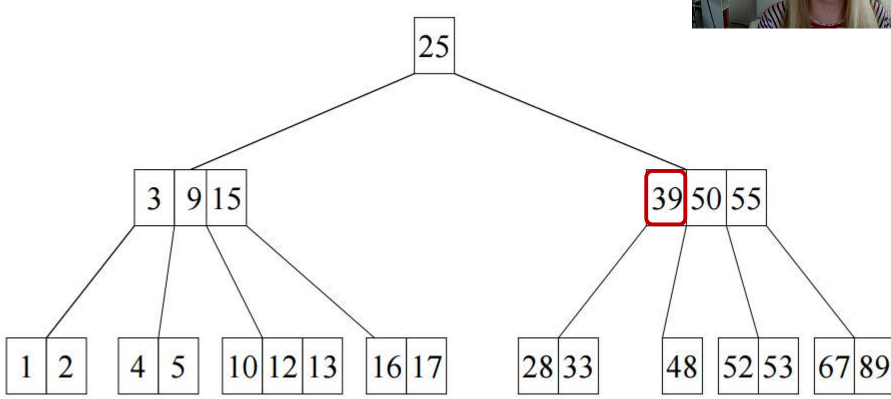


25

25

Example: delete 37 – replace by 39

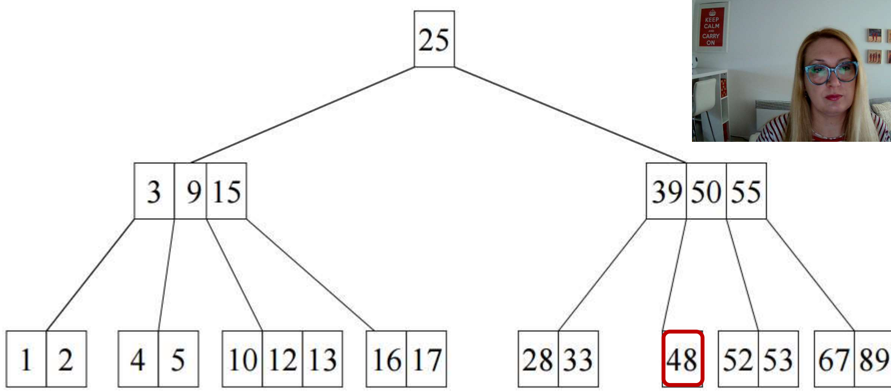




26

26

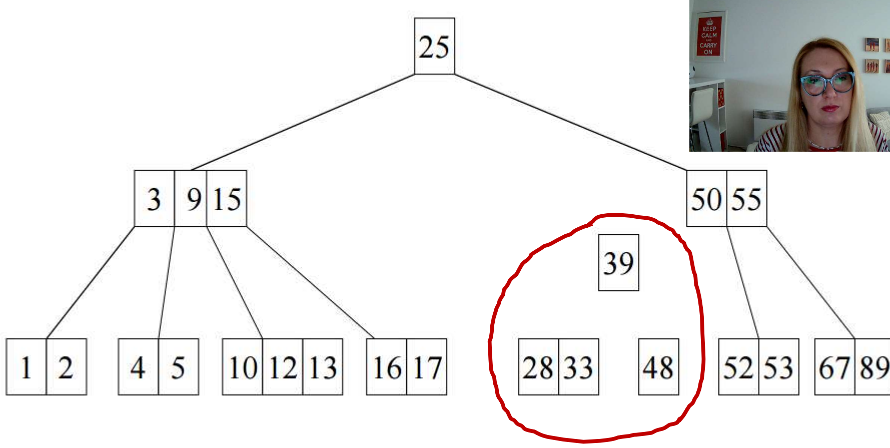
Example: delete 37 – fix underflow (fusion)



27

27

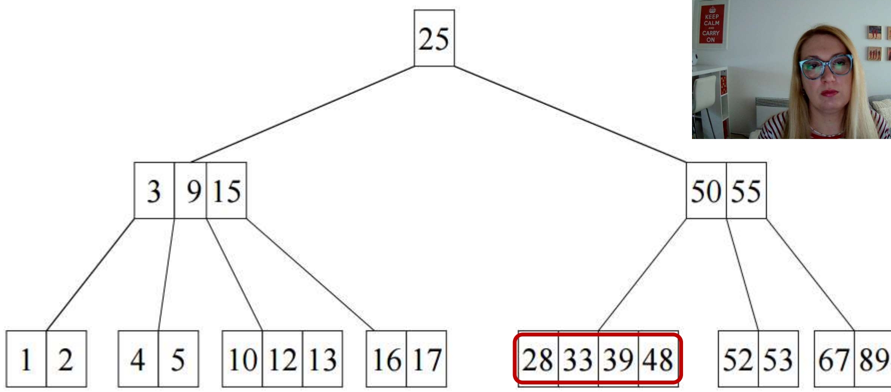
Example: delete 37 – fix underflow (fusion)



28

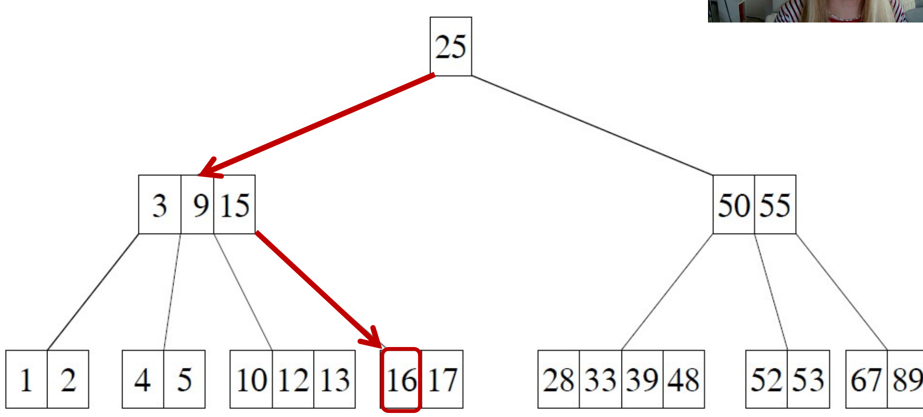
28

Example: delete 37 – fix underflow (fusion)



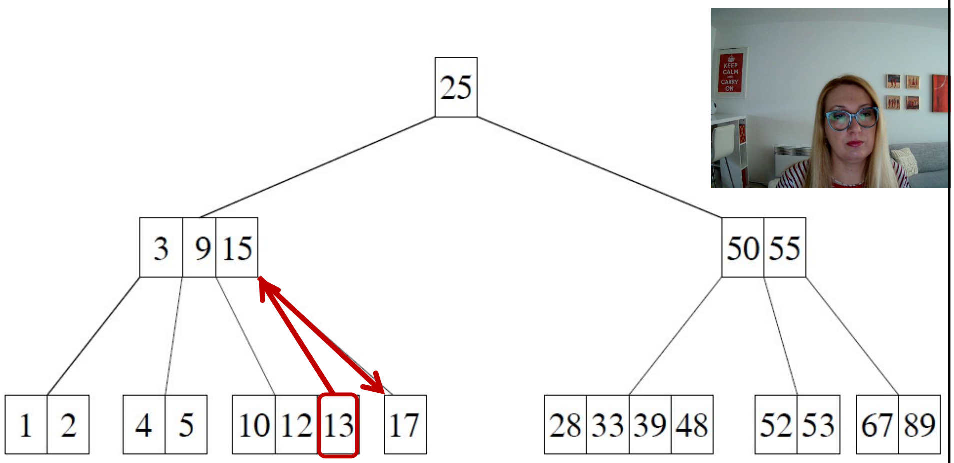
29

Example: delete 16



30

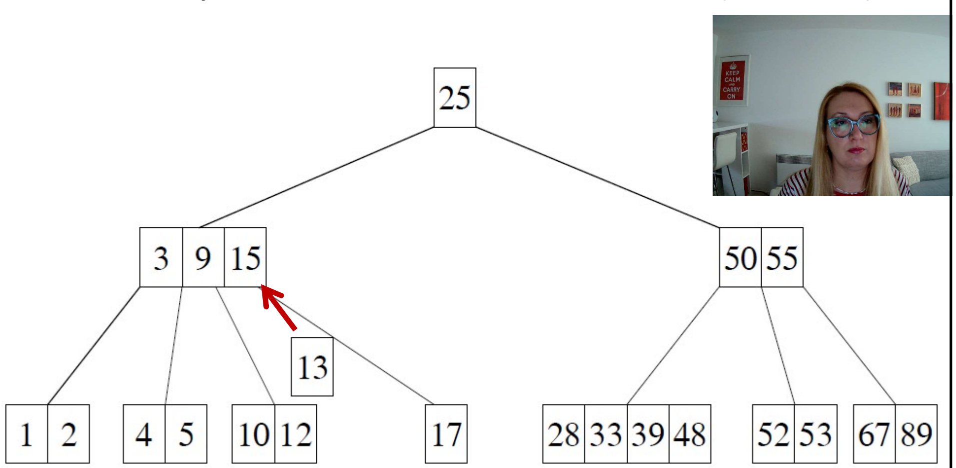
Example: delete 16 – fix underflow (transfer)



31

31

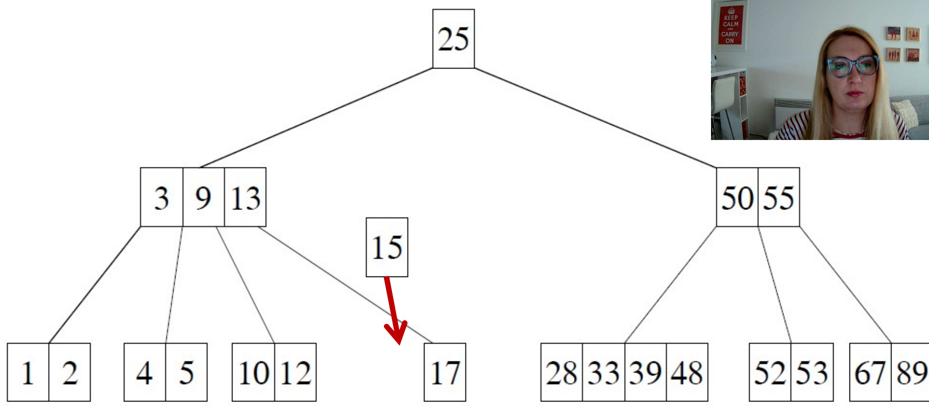
Example: delete 16 – fix underflow (transfer)



32

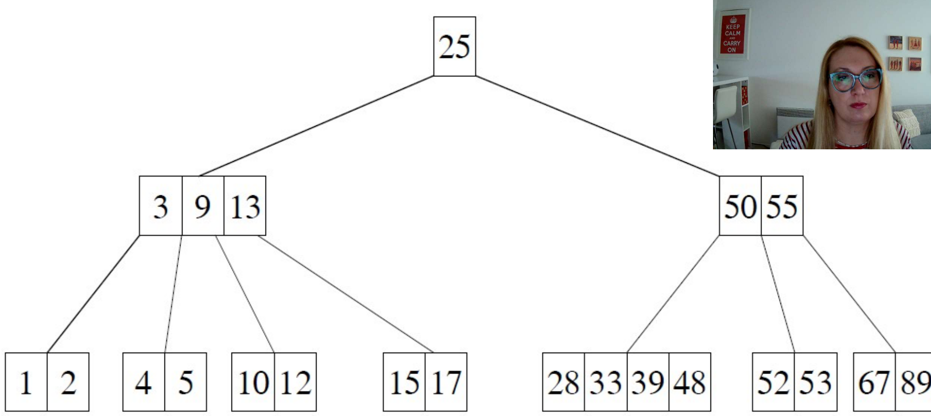
32

Example: delete 16 – fix underflow (transfer)



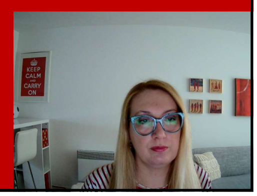
33

Example: delete 16 – fix underflow (transfer)



34

Efficiency

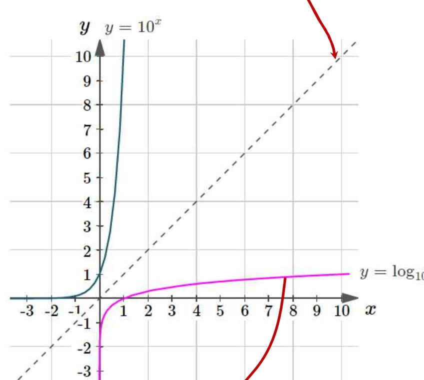


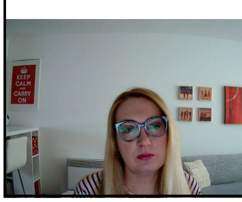
35

Efficiency of B–trees

- space complexity: O(*n*)
- time complexity

Operation	Average	Worst
search	O(log(*n*))	O(log(*n*))
insert	O(log(*n*))	O(log(*n*))
delete	O(log(*n*))	O(log(*n*))





36

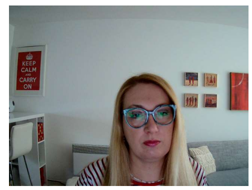
36

Is a B–tree index useful?

- appropriate use of index can help data retrieval:

```
SELECT  Student.S#, Module.title
FROM    Student, Module
WHERE   Student.S# = Module.S#
AND     Student.name = 'John';
```

- assumptions:
  - table Student has n tuples
  - table module has m tuples
  - tuples are unordered

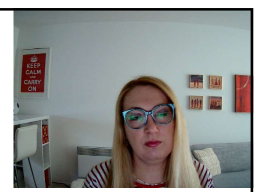


- Q: How many comparisons do we have to perform in order to answer the query?
  (a) with or (b) without a B–tree index on S#

37

37

Conclusion



- an index file is much smaller than a table, thus more efficient to handle
- if small enough, could be kept in the main memory
- multiple indexes may be created for a single table
- an index file may be ordered, but there is no need for a table to be ordered
- if a table is updated frequently, index may not be good
- if a column contains few distinct values, then index like B–trees may not be useful

38

38