Database files and indexing

Lecture topics

- files and disks
- indexing
- SQL and indices

References:

- text 3rd edition: Chapter 6, sections 1-3, 6; Chapter 16, section 4.1; Chapter 20, section 6; background material in Chapter 5
- text 4th edition: Chapter 14, sections 1-3, 6; Chapter 16, sections 1, 2.1; Chapter 18, section 6; background material in Chapter 13
- B-tree supplemental material from any decent data-structures text

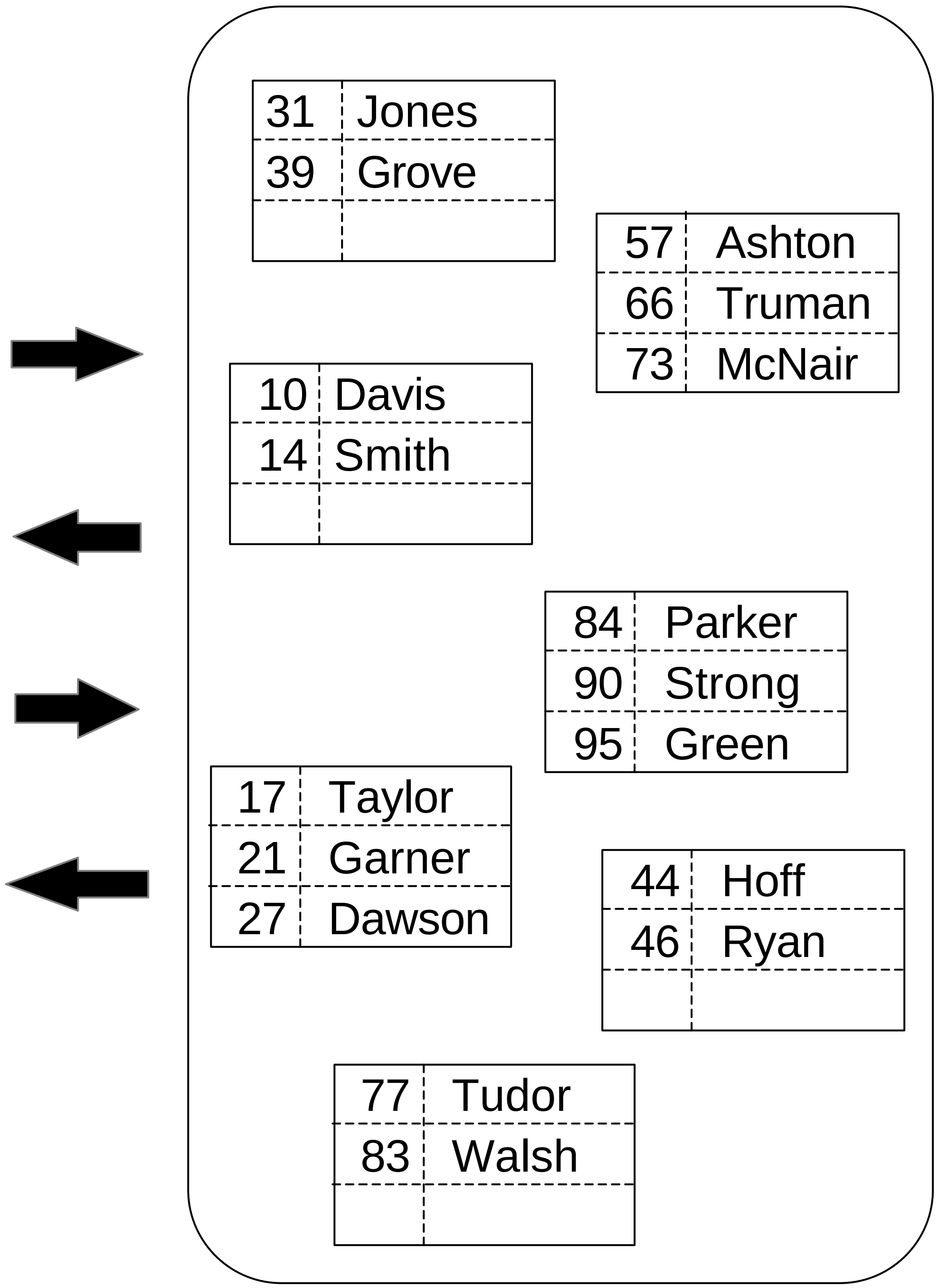
File systems and disks

- databases are stored in files
  - one file per relation, or
  - one file for entire database
- files reside on disks
- to access (read, modify, update, delete) data, the DBMS must transfer it temporarily to a **buffer** in main memory
- data is transferred between disk and main memory in units called **blocks**
- transferring a block is a slow operation
- disk access times dominate query execution times

Storing tuples in file blocks

ID	Surname
10	Davis
14	Smith
17	Taylor
21	Garner
27	Dawson
31	Jones
39	Grove
44	Hoff
46	Ryan
57	Ashton
66	Truman
73	McNair
77	Tudor
83	Walsh
84	Parker
90	Strong
95	Green

“People”
Relation



Disk storage

A table scan

```
select * from People
where Surname = 'Smith'

select * from People
where ID = 14
```

- to answer these queries, the DBMS must search blocks of the database file to look for matching tuples

- the purpose of an **index** is to reduce the number of blocks that must be checked by the DBMS

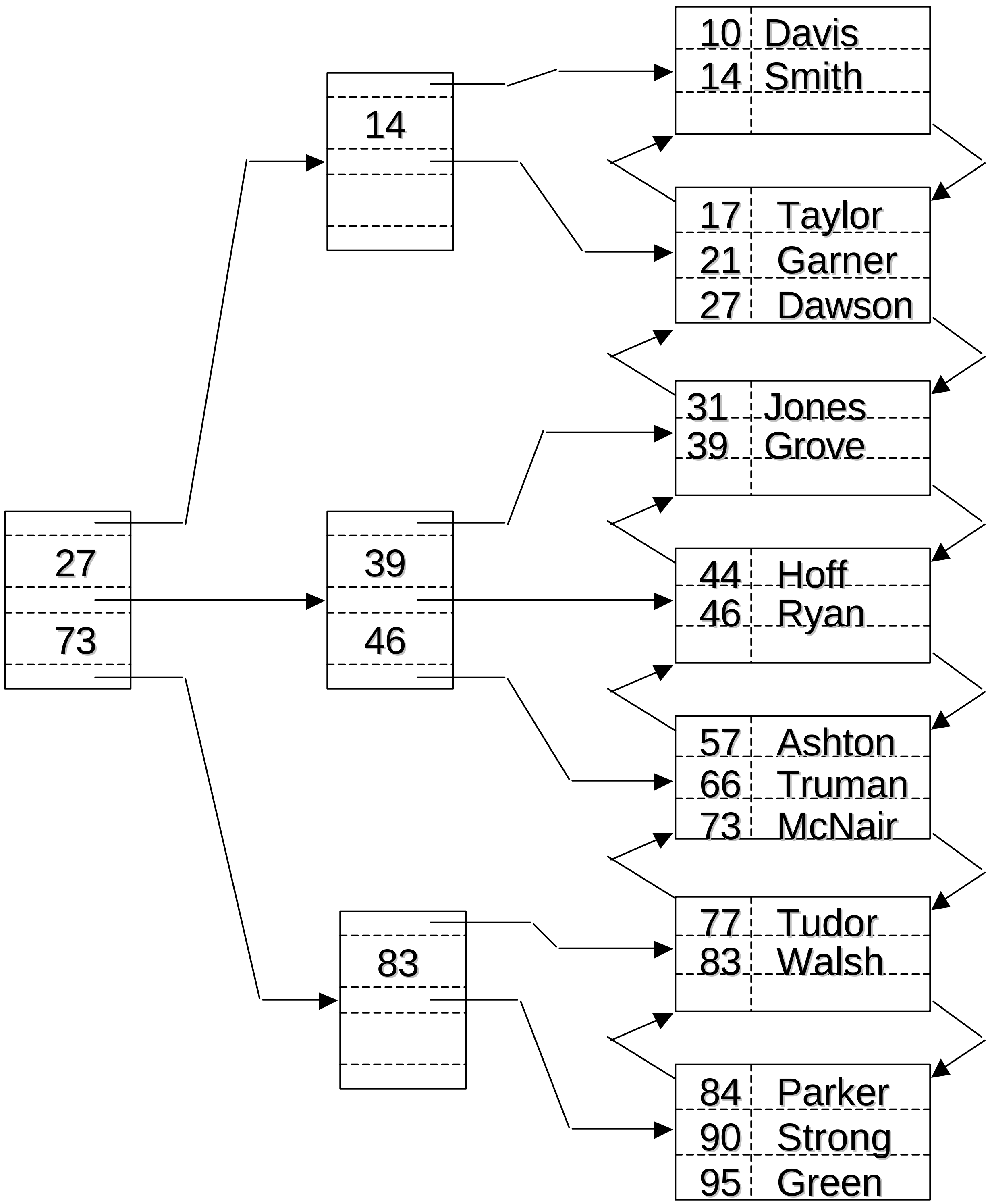
Indexing

- an index consists of extra information (a data structure) added to a file to provide faster access to data.
- an index is defined on one or more attributes of a relation.
- generally, an index defined on attribute *A* of relation *R* will:
  - substantially reduce execution time for selections that specify conditions involving *A*
  - increase execution time for insertions or deletions of tuples from *R*
  - increase the size of the file required to store *R*

B-trees

- B-trees are widely-used index structures
- B-trees are fully dynamic: they easily grow and shrink
- B-trees come in several flavours: we will discuss B^+-trees
- B-trees have two parts: index blocks and data blocks
- B-tree index and data pages are kept at least half full

B-tree example



B-tree blocks

- index blocks:
  - each block stores a maximum of *m* keys and *m* + 1 pointers
  - each block stores at least $\lfloor m/2 \rfloor$ keys and $\lfloor m/2 \rfloor + 1$ pointers

P_0	K_1	P_1	K_2	...	K_m	P_m

- data blocks:
  - each block stores a maximum of *n* rows
  - each block stores at least $\lfloor (n + 1)/2 \rfloor$ records
  - each block also contains two pointers

P_b	R_1	R_2	R_3	...	R_n	P_f

B-tree cost example

- suppose that a b-tree index is defined on attribute *A* of relation *R*, with the following properties:
  - there are 4K (4096) bytes per block
  - each tuple of R occupies 256 bytes
  - there are 1,000,000 tuples in *R*
  - data blocks are 65% full, on average
  - each index block holds up to 100 pointers
  - index blocks are 100% full
- how many blocks will the DBMS have to retrieve from the disk to answer:

```
select *
from R
where A = c
```

(where *c* is a constant in domain of *A*, and assume result contains only one tuple.

...continued

- each data block holds at most 4096/256 = 16 tuples
- each data block holds 0.65 * 16 ≈ 10 tuples
- 1,000,000÷10 = 100,000 data blocks to store *R*
- each index block holds 100 pointers, so 100,000÷100 = 1,000 index blocks in the lowest level of the b-tree
- 1,000÷100 = 10 index blocks in the next level
- counting the root, there are 3 levels of index blocks
- retrieving the matching tuple requires only 4 block retrievals (3 index plus 1 data)

- without an index, 100,000 blocks maximum, 50,000 average would have to be retrieved

Range queries

- b-trees can also help for **range queries**:

```
select *
from R
where A ≥ c
```

- If a b-tree is defined on *A*, we can use it to find the tuples for which $A = c$. Using the forward pointers in the data blocks, we can then find tuples for which $A > c$.

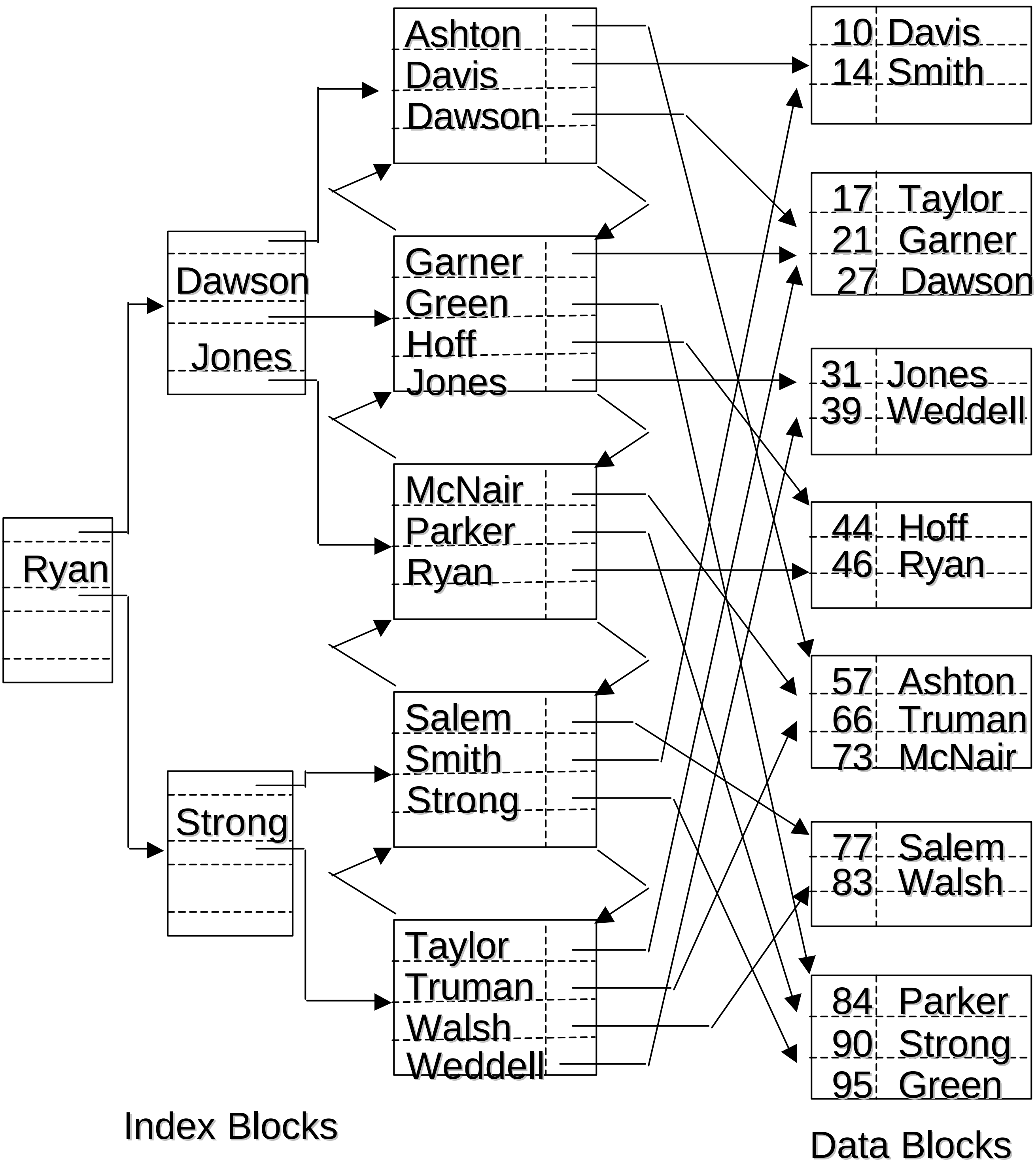
Clustering index versus non-clustering index

- an index on an attribute *A* is a **clustering** index if tuples with similar values for *A* are stored together in the same block

- other indices are **non-clustering** (or secondary) indices

- a relation may have at most one clustering index, and any number of non-clustering indices

Non-clustering index example



Managing indices

- Current SQL standards do not specify how to manage indices
- Many commercial implementations have something like:

```
create index SurnameIndex
on People(Surname);

drop index SurnameIndex
```

Multi-attribute indices

- possible to create an index on several attributes of the same relation. E.g.:

```
create index NameIndex
on People(Surname, Initials)
```

- attribute order is important: in this example:
  - tuples are organized first by Surname
  - tuples with a common surname are then organized by Initials

...continued

- `NameIndex` would be useful for:

```
select *
from Student
where Surname = 'Smith'
```

or:

```
select *
from Student
where Surname = 'Smith'
and Initials = 'A. B.'
```

- `NameIndex` would not be useful for:

```
select *
from Student
where Initials = ‘A.B.’
```

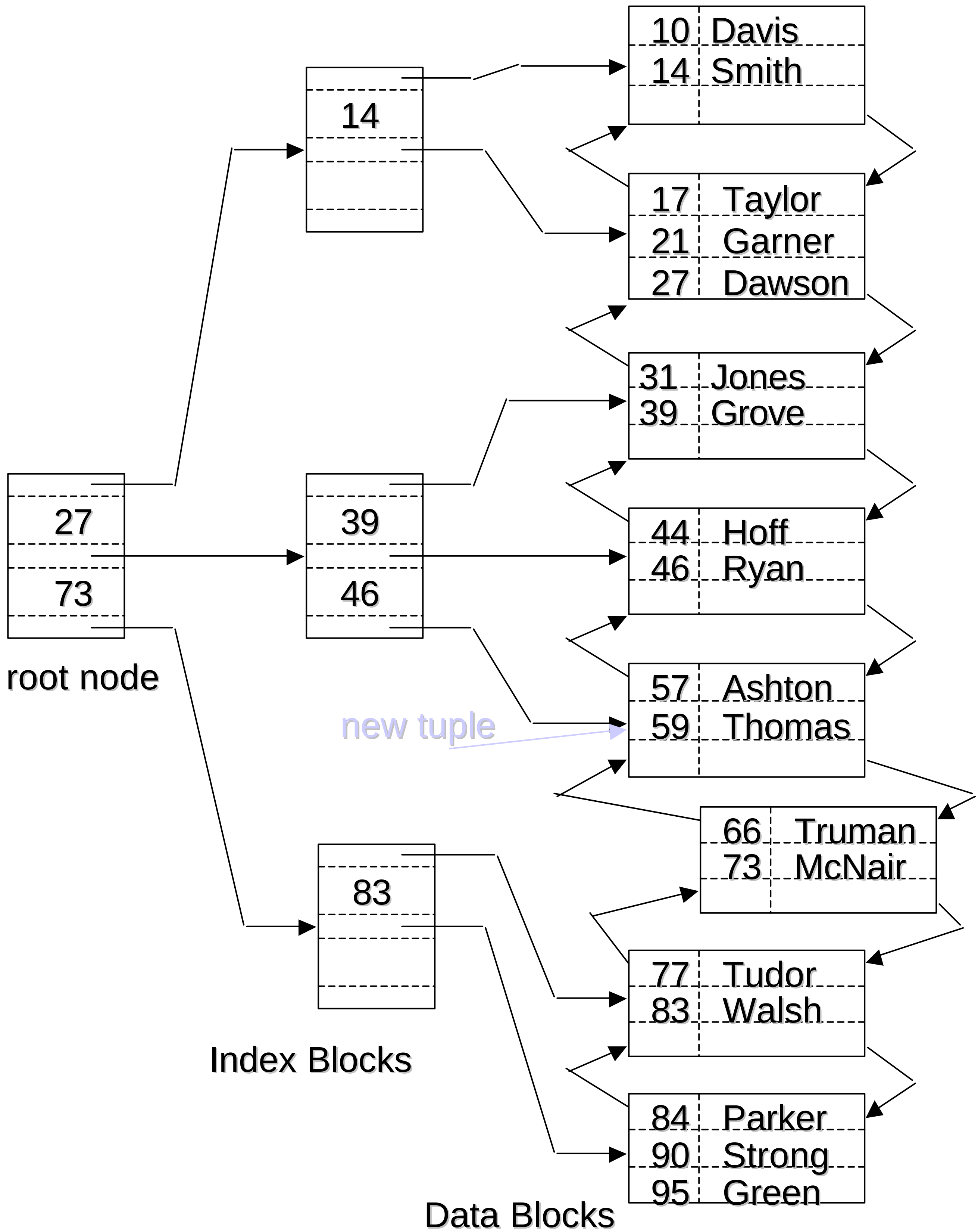
Supplementary material

- B-tree insertion and deletion procedures

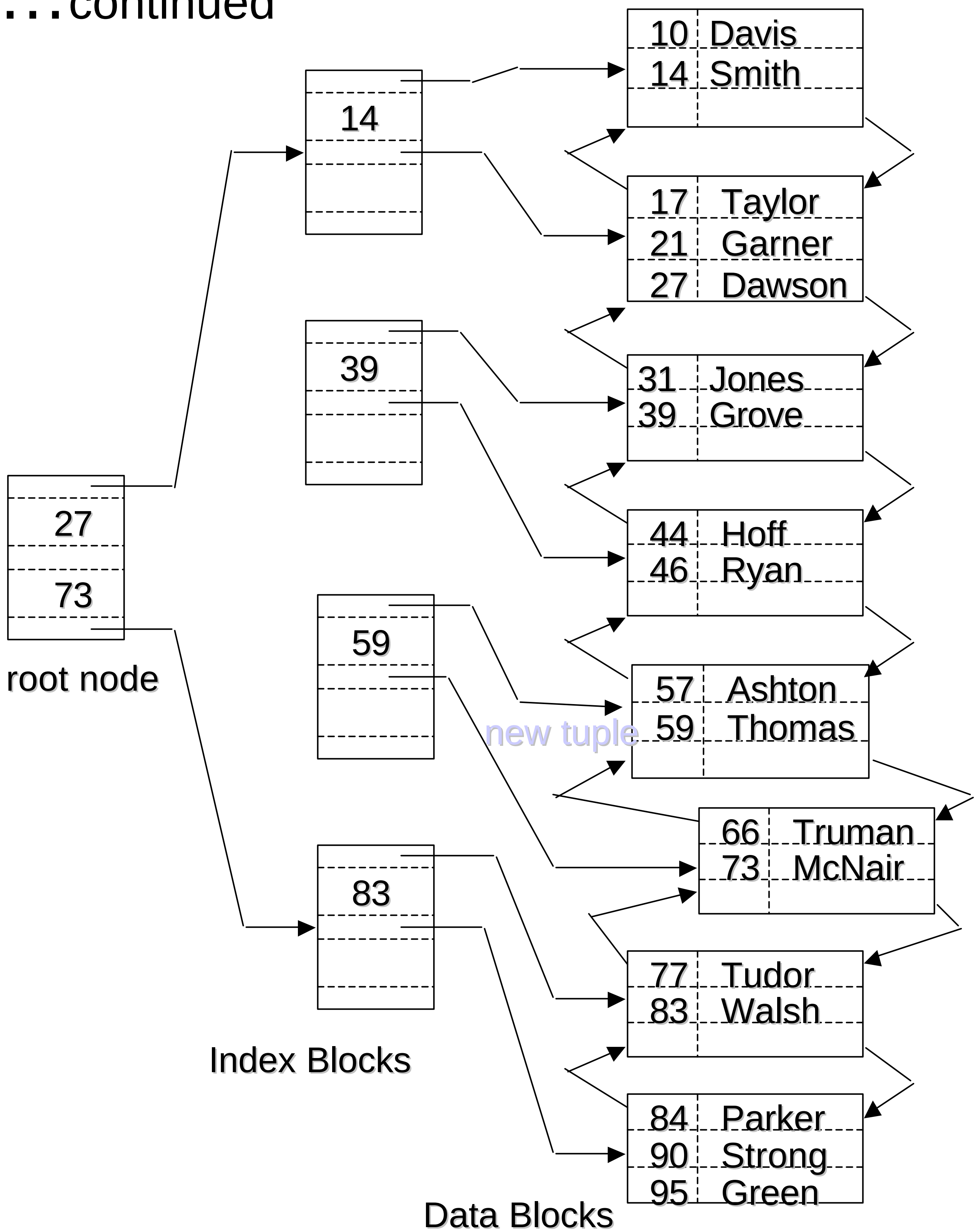
B-tree Insertions

- Determine data block where new tuple belongs.
- If there is room in the block, place the tuple in it.
- If there is no room, find an empty block, and move half of the records into the new block. This is called **splitting**.
- Add an entry for the new block in the parent index block.
- If the index block is full, it may split. In this case, the middle pointer is promoted to the next higher index level.
- Splitting may continue all the way to the root of the b-tree.

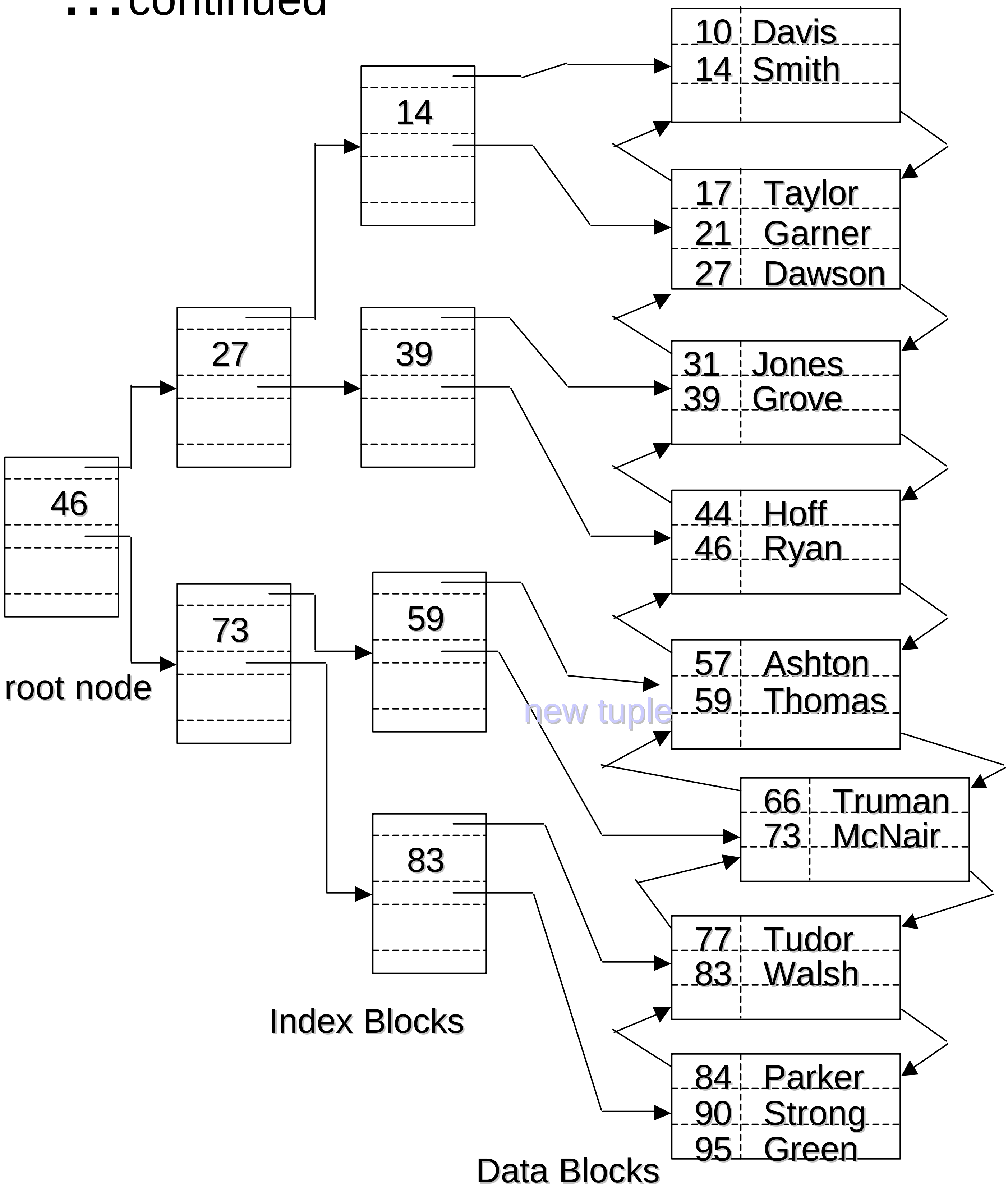
Insertion example



...continued


10 Davis
14 Smith
17 Taylor
21 Garner
27 Dawson
31 Jones
39 Grove
44 Hoff
46 Ryan
57 Ashton
59 Thomas
new tuple
66 Truman
73 McNair
77 Tudor
83 Walsh
84 Parker
90 Strong
95 Green
14
39
59
83
27
73
root node
Index Blocks
Data Blocks

...continued

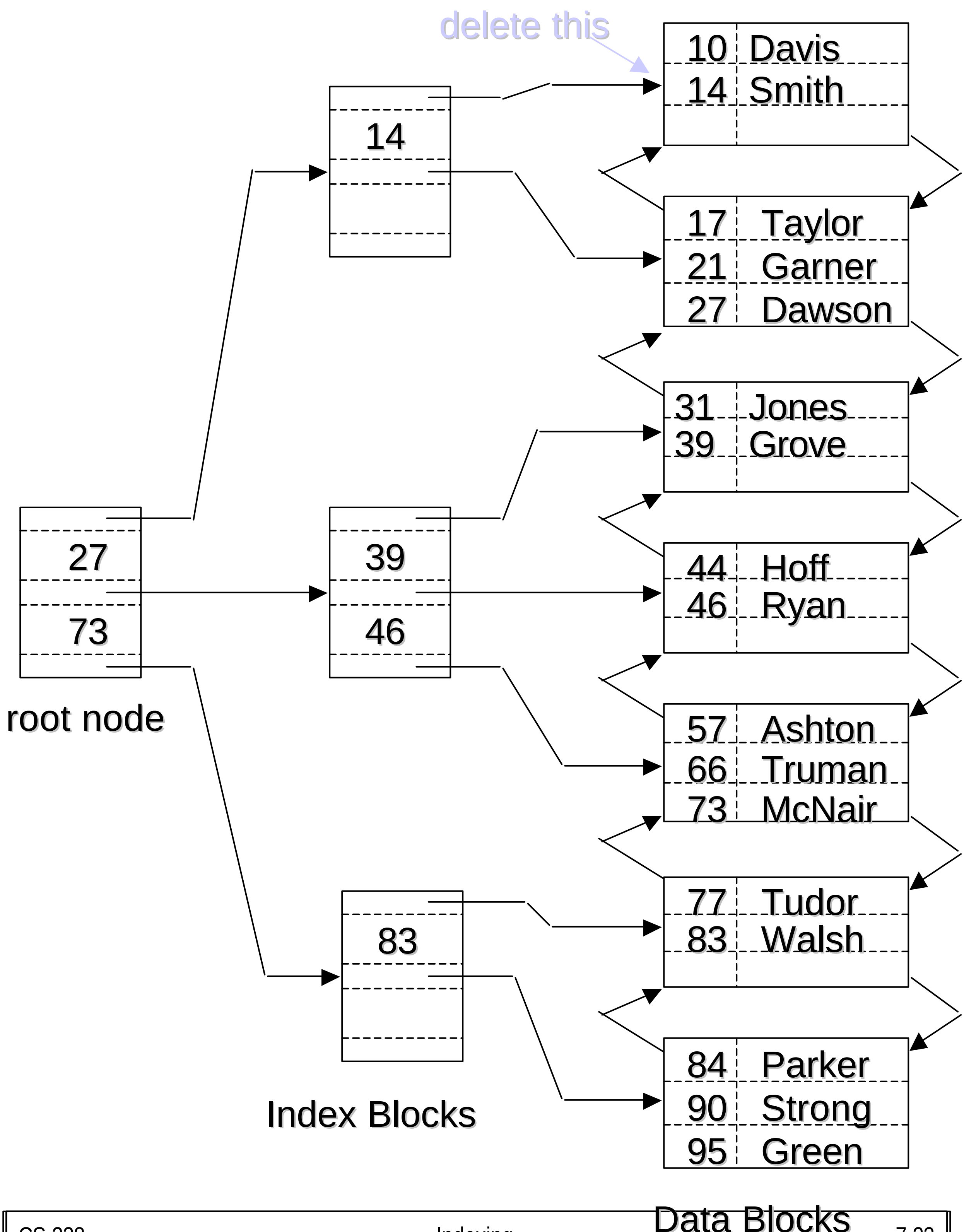


14
27
39
46
root node
73
59
83
new tuple
Index Blocks
10 Davis
14 Smith
17 Taylor
21 Garner
27 Dawson
31 Jones
39 Grove
44 Hoff
46 Ryan
57 Ashton
59 Thomas
66 Truman
73 McNair
77 Tudor
83 Walsh
84 Parker
90 Strong
95 Green
Data Blocks

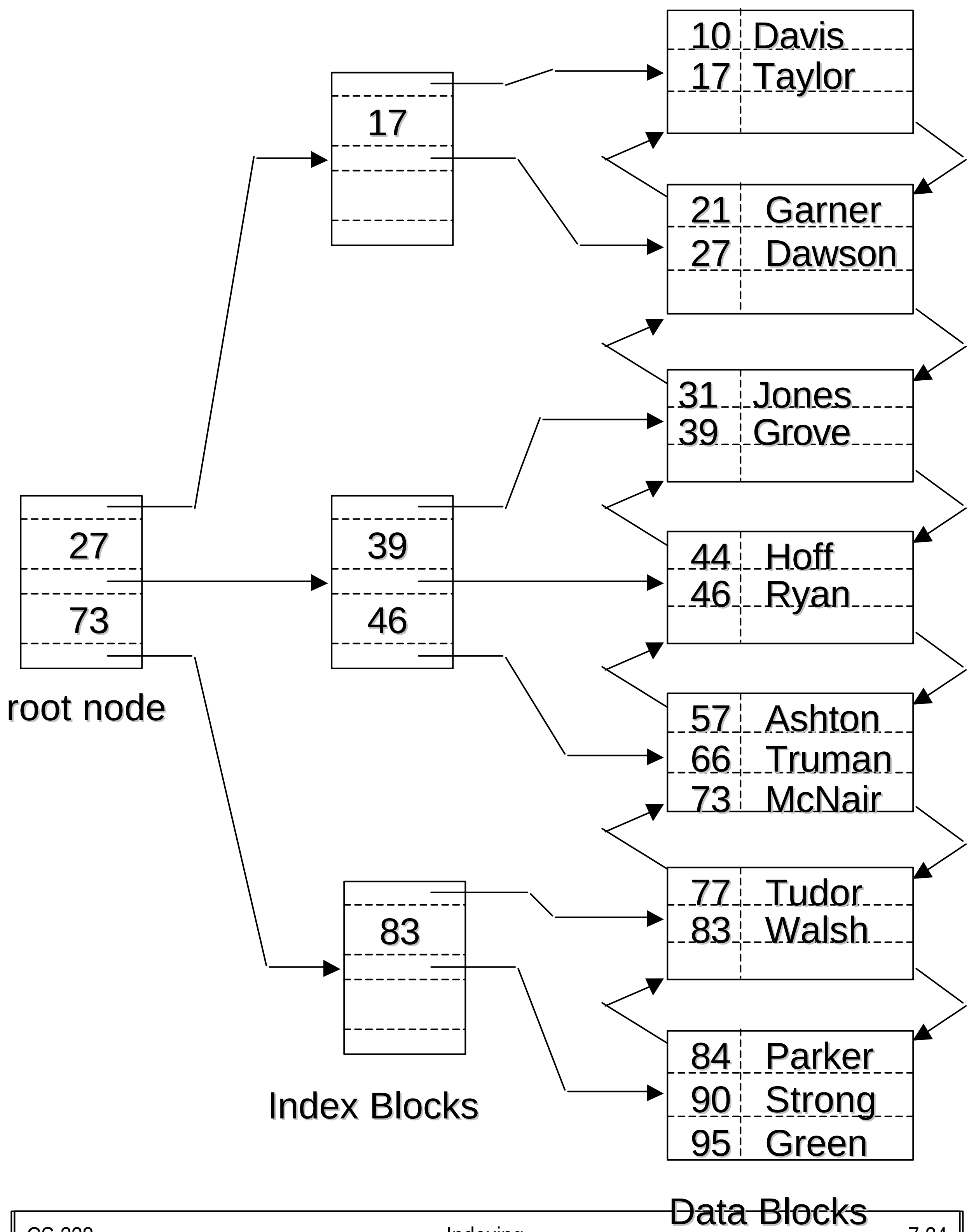
B-tree Deletions

- Determine data block where tuple is located.
- Remove the tuple from the data block.
- If the block is less than half full, either:
  – distribute remaining tuples to the block’s sibling, remove the block from the b-tree, and delete the block’s pointer from the parent index node, or
  – steal some tuples from the block’s siblings, and place them in the block
- If a data block is removed, its pointer must be deleted from its parent’s index node. Deletion of pointers may cascade all the way to the root.
- In either case, all blocks that remain in the b-tree must be at least half full.

Deletion example

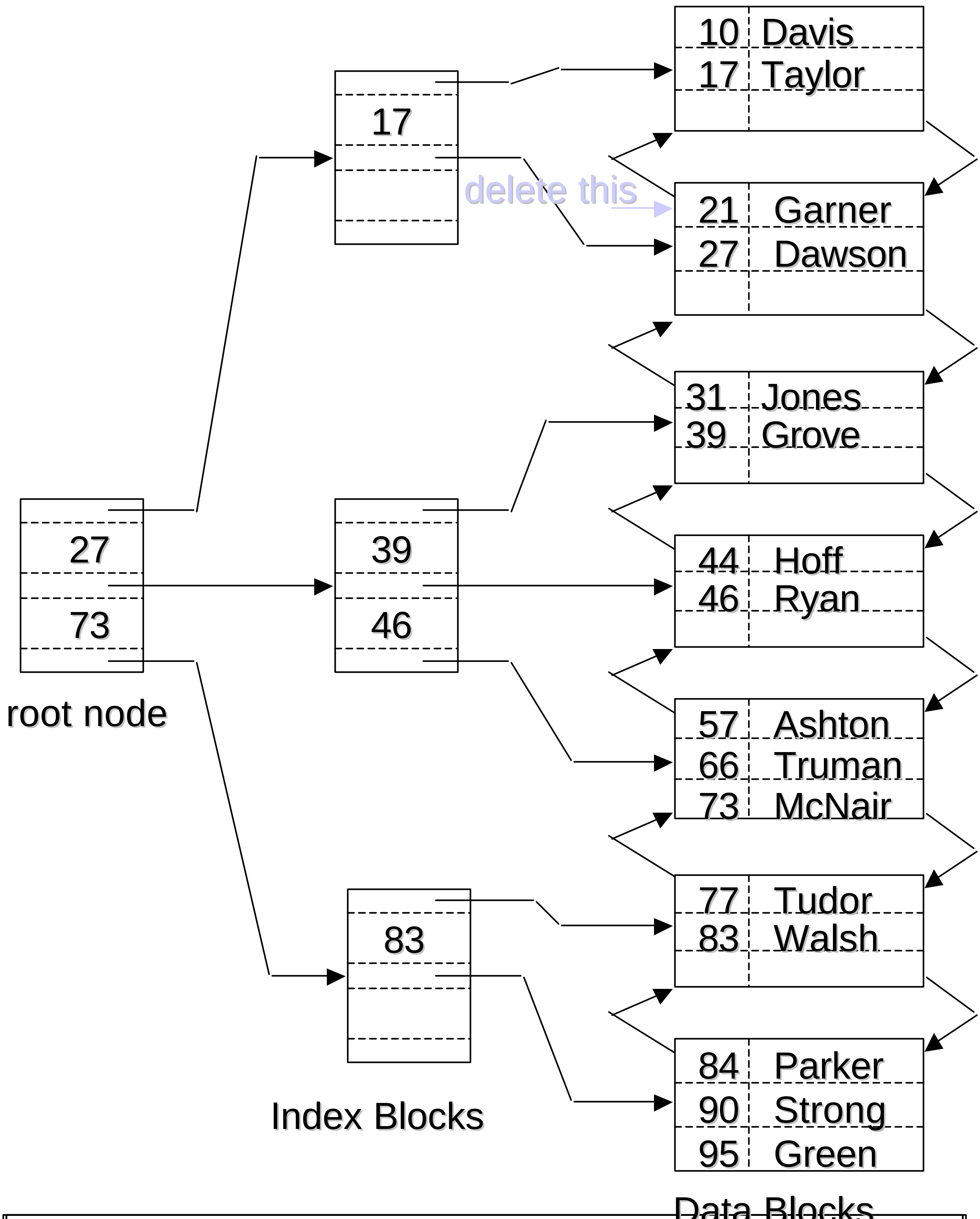


...continued

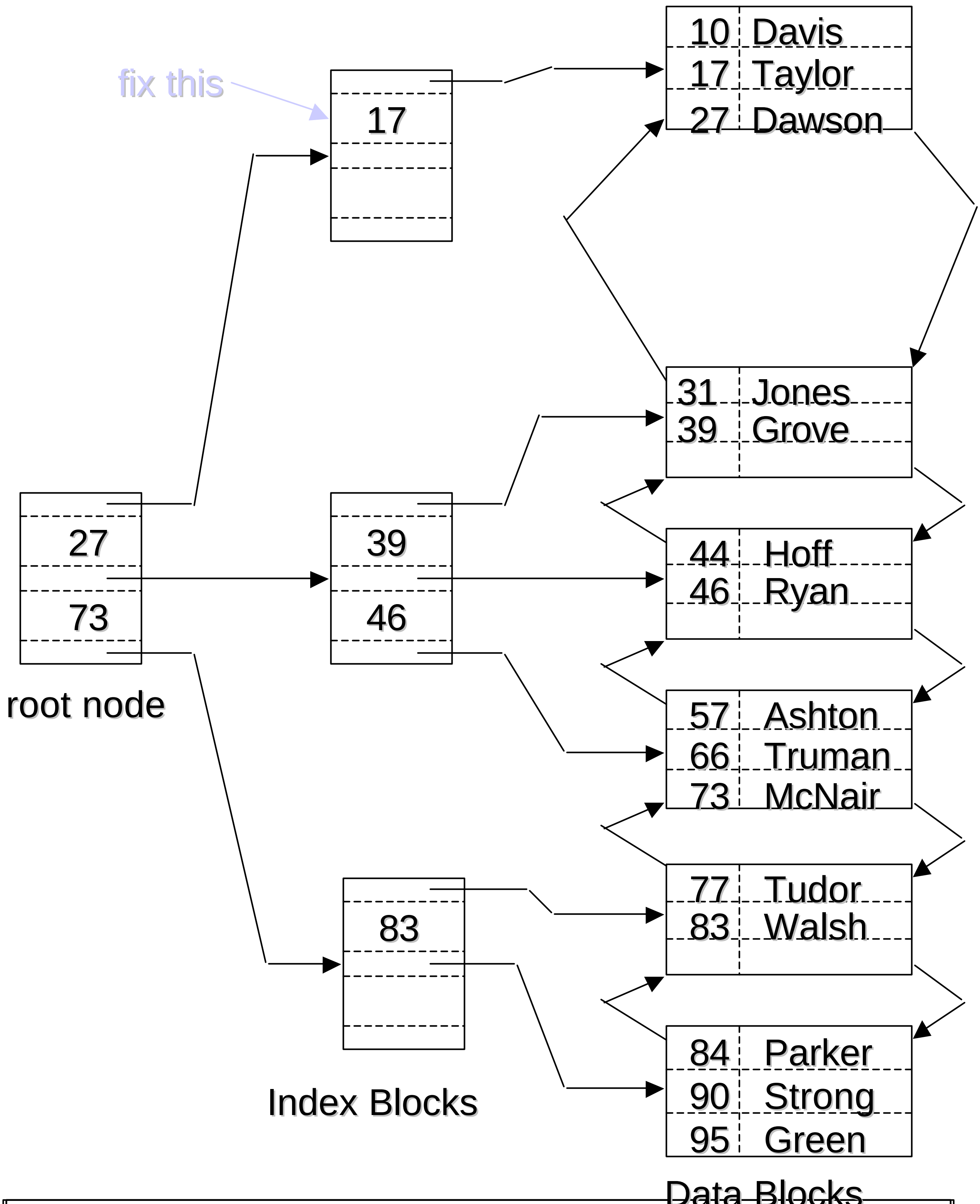


10 Davis
17 Taylor
17
21 Garner
27 Dawson
31 Jones
39 Grove
27
73
39
46
44 Hoff
46 Ryan
root node
57 Ashton
66 Truman
73 McNair
83
77 Tudor
83 Walsh
84 Parker
90 Strong
95 Green
Index Blocks
Data Blocks

Another deletion example

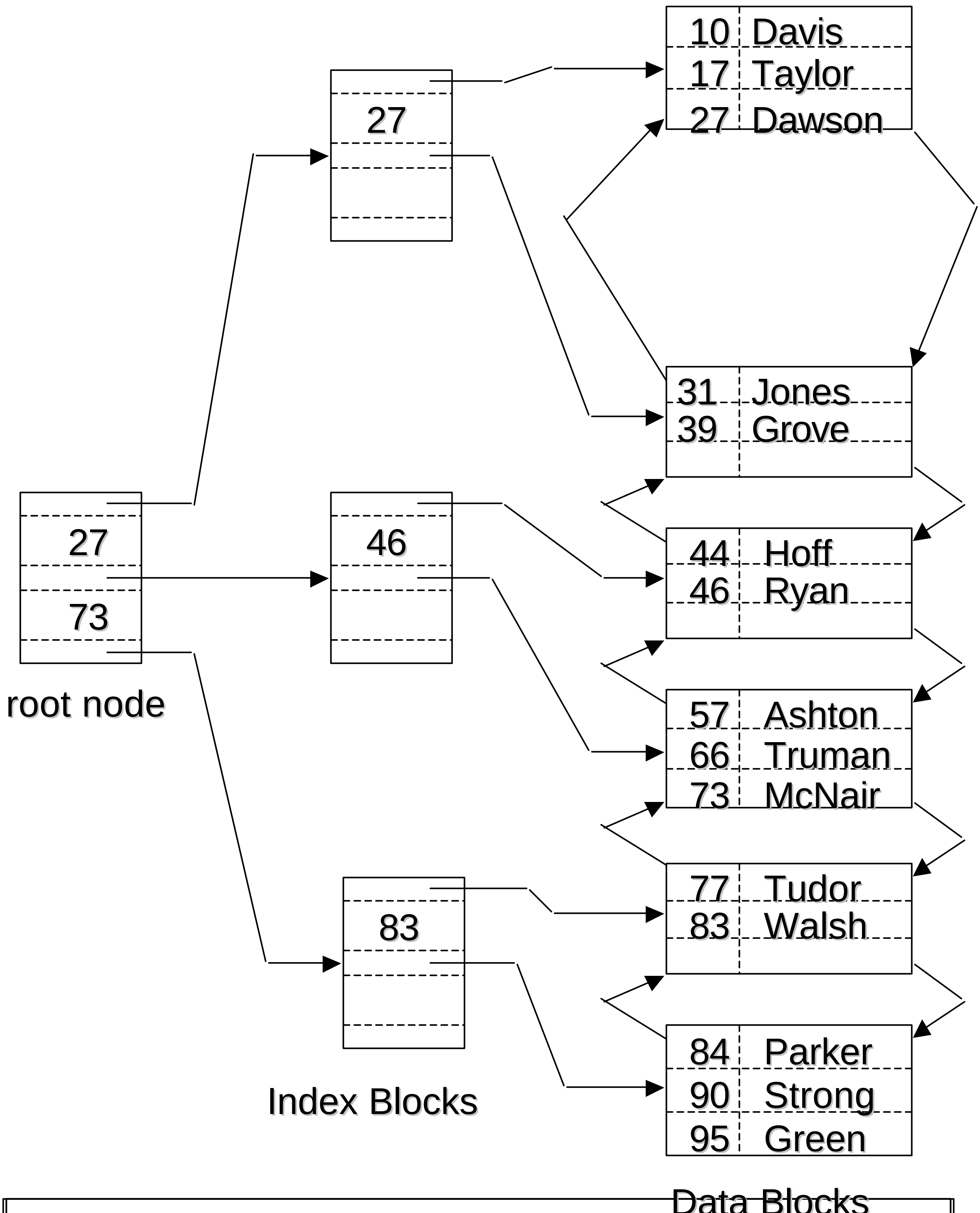


...continued



fix this
17
27
73
root node
39
46
83
Index Blocks
10 Davis
17 Taylor
27 Dawson
31 Jones
39 Grove
44 Hoff
46 Ryan
57 Ashton
66 Truman
73 McNair
77 Tudor
83 Walsh
84 Parker
90 Strong
95 Green
Data Blocks

...continued



27
73
root node
27
46
83
Index Blocks
10 Davis
17 Taylor
27 Dawson
31 Jones
39 Grove
44 Hoff
46 Ryan
57 Ashton
66 Truman
73 McNair
77 Tudor
83 Walsh
84 Parker
90 Strong
95 Green
Data Blocks