VIII. Database Files and Indexing

Lecture Topics

- Files and Disks
- Indexing
- Managing Indices
- B-trees
- Clustering
- Multi-attribute Indices

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**.

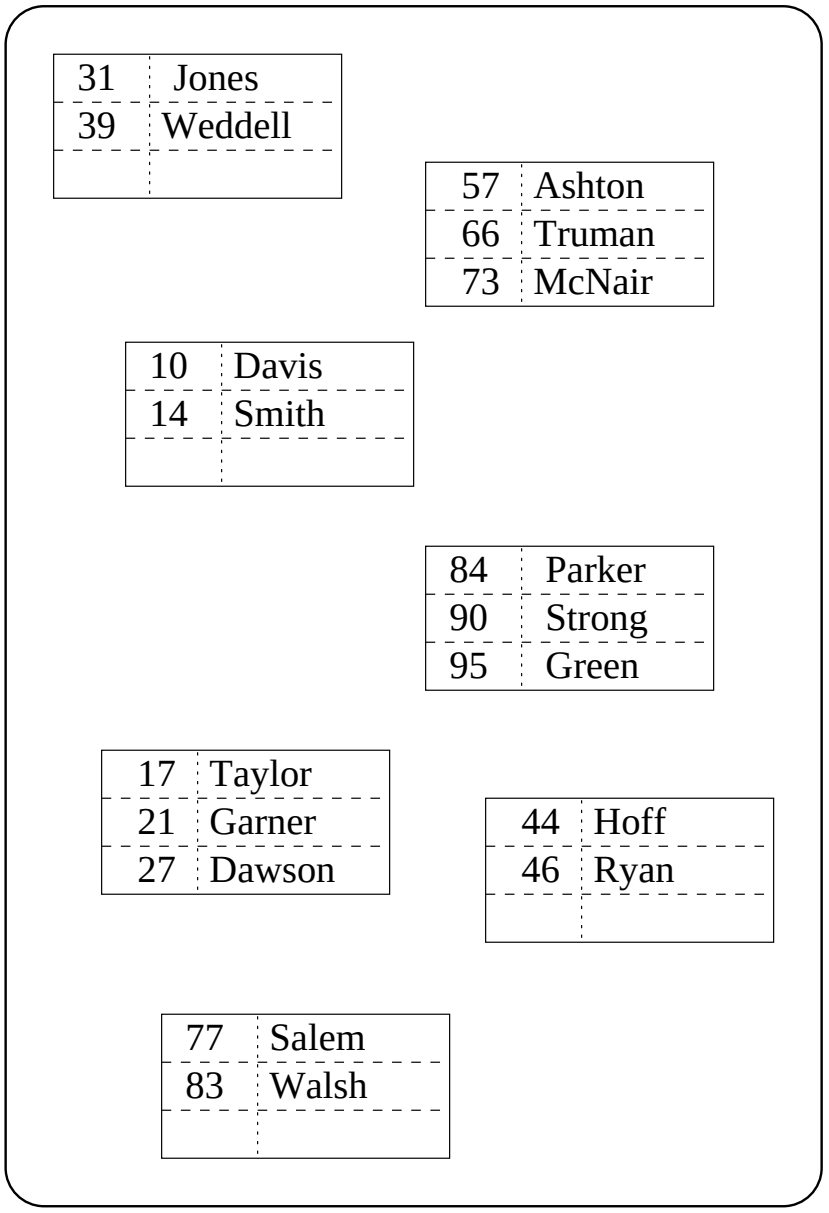
Transferring a block is a slow operation.

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	Weddell
44	Hoff
46	Ryan
57	Ashton
66	Truman
73	McNair
77	Salem
83	Walsh
84	Parker
90	Strong
95	Green

"People" RELATION



FILE

A Table Scan

select $*$ **from** People
where Surame = 'Smith'

select $*$ **from** People
where ID = 14

To answer these queries, the DBMS must search the blocks of the database file to check 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

Managing Indices

create index SurnameIndex
on People(Surname);

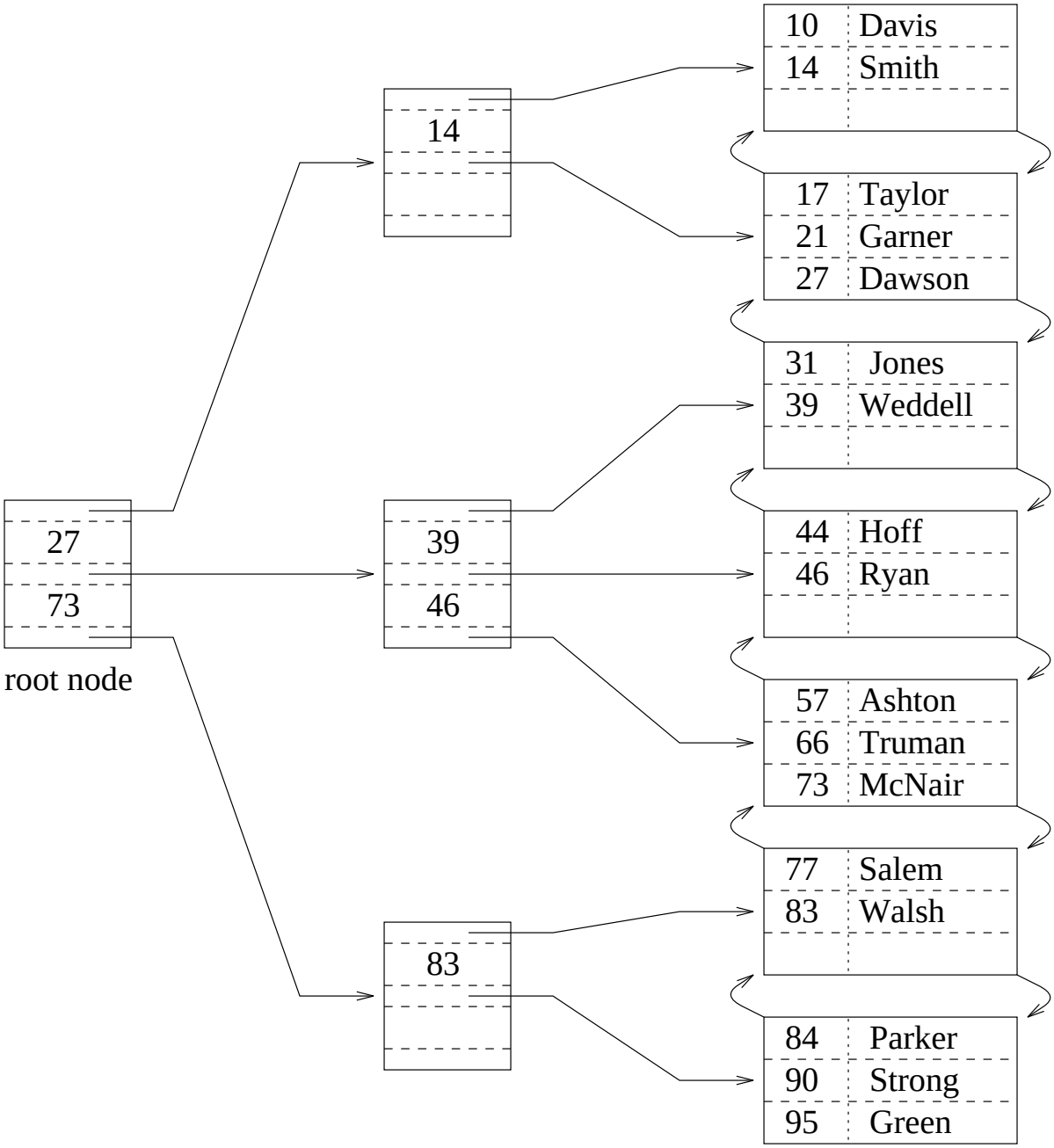
drop index SurnameIndex

There is no standard method for creating and dropping indices in TBITS SQL (SQL 92).

B-trees

- B-trees are widely-used index structures.
- B-trees are fully dynamic: they easily grow and shrink easily.
- 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 blocks are kept at least half full.

B-tree example



Index Blocks Data Blocks

B-tree blocks

- Index blocks
  - each block stores at most m values and $m + 1$ pointers
  - each block stores at least $\lfloor m/2 \rfloor$ values and $\lfloor m/2 \rfloor + 1$ pointers

$$\boxed{P_0 \mid V_1 \mid P_1 \mid V_2 \mid \cdots \mid V_m \mid P_m}$$

- Data blocks
  - each blocks stores at most n records
  - each blocks stores at least $\lfloor (n + 1)/2 \rfloor$ records
  - each block also contains two pointers

$$\boxed{P_b \mid R_1 \mid R_2 \mid R_3 \mid \cdots \mid R_n \mid P_f}$$

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 the following query?

select *
from R
where $A = c$

Here, c is a constant in the domain of A, and we are assuming that the result of this query will contain only one tuple.

B-Tree Cost Example (cont.)

Each data block holds at most $4096/256 = 16$ tuples.

Each data block holds $0.65 * 16 \approx 10$ tuples.

A total of $1000000/10 = 100000$ data blocks are needed to store the tuples of R.

Since each index block holds 100 pointers, there will be $100000/100 = 1000$ index blocks in the lowest level of the b-tree.

There will be $1000/100 = 10$ index blocks in the next level of the b-tree.

Counting the root, there are 3 levels of index nodes in the b-tree.

Retrieving the matching tuple will require only 4 block retrievals, one for each level of the index, plus one for the data block containing the tuple.

Without an index, 100,000 blocks would have to be retrieved. This could take 20 or 30 minutes.

Range Queries

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

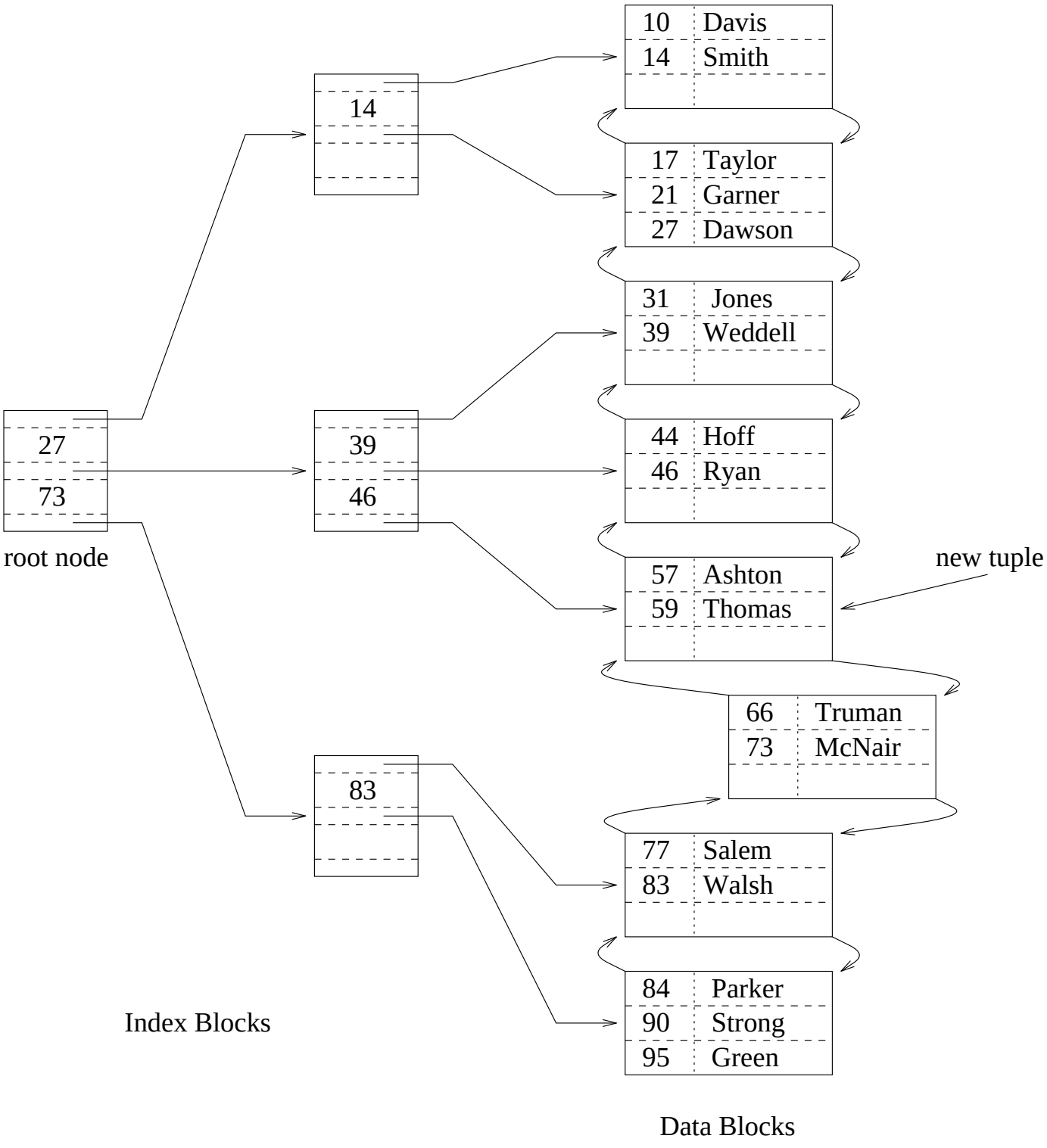
select *
from R
where $A \geq c$

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 $A > c$.

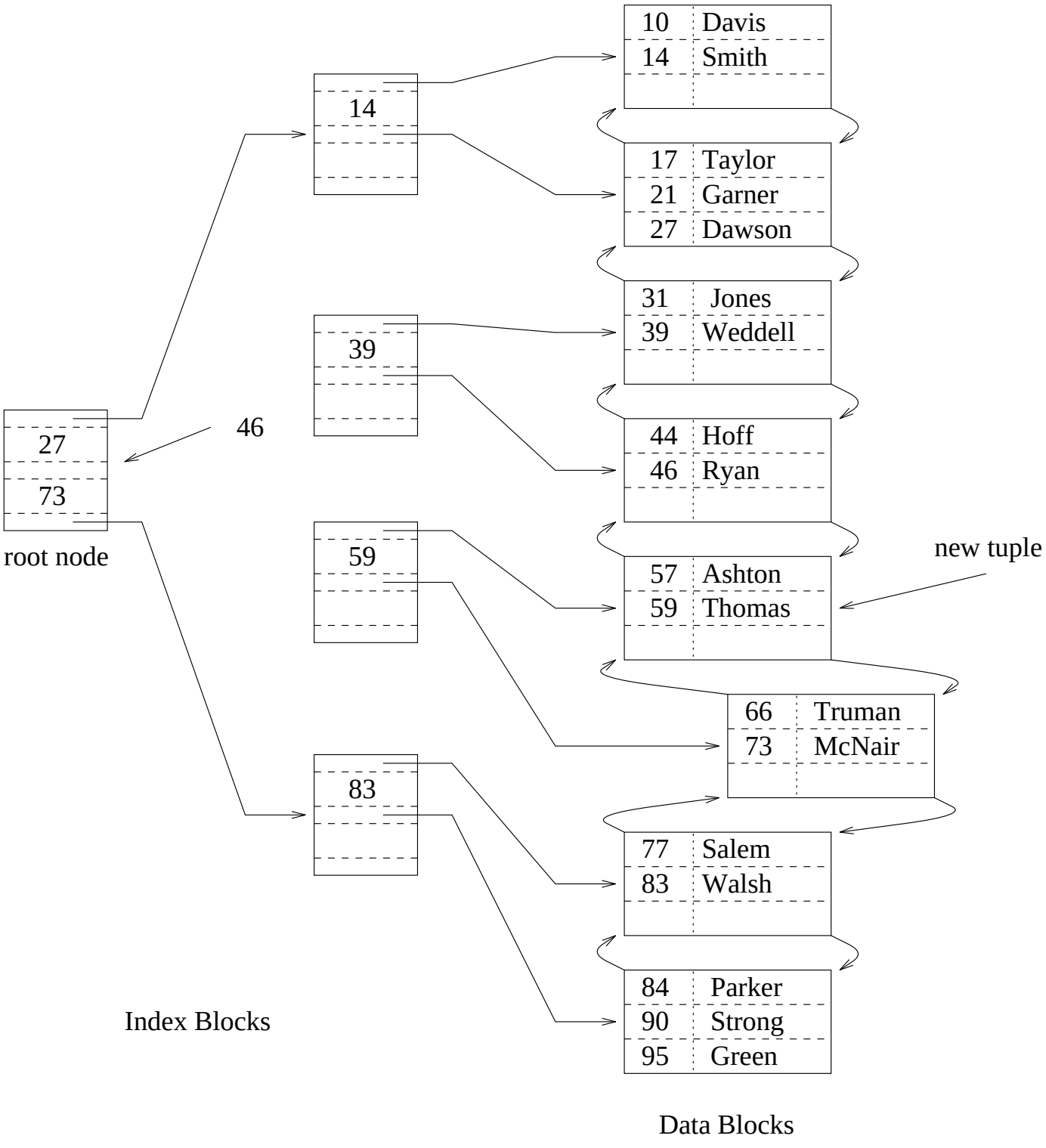
B-tree Insertions

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

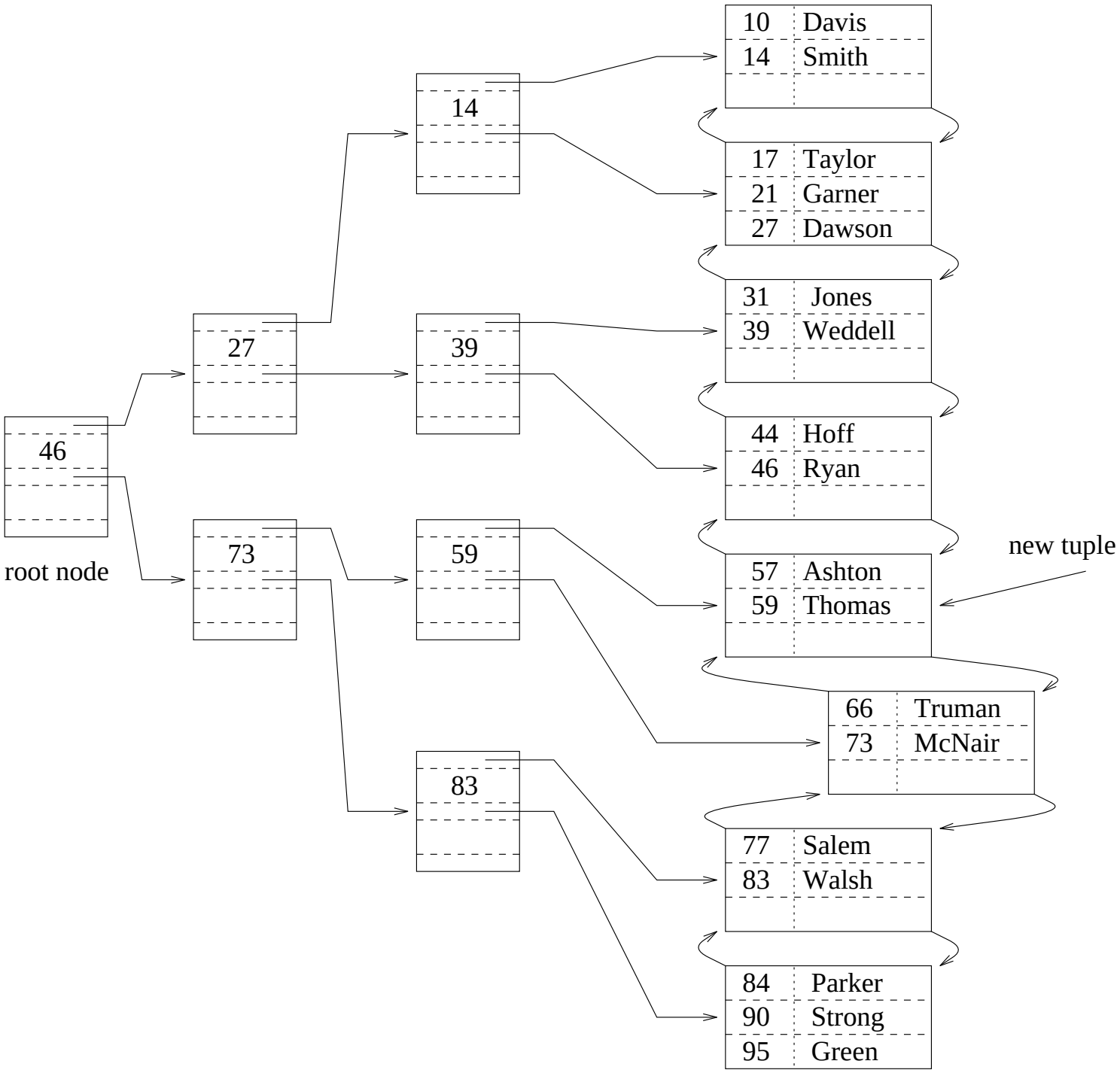
Insertion Example



Insertion Example (cont.)



Insertion Example (cont.)

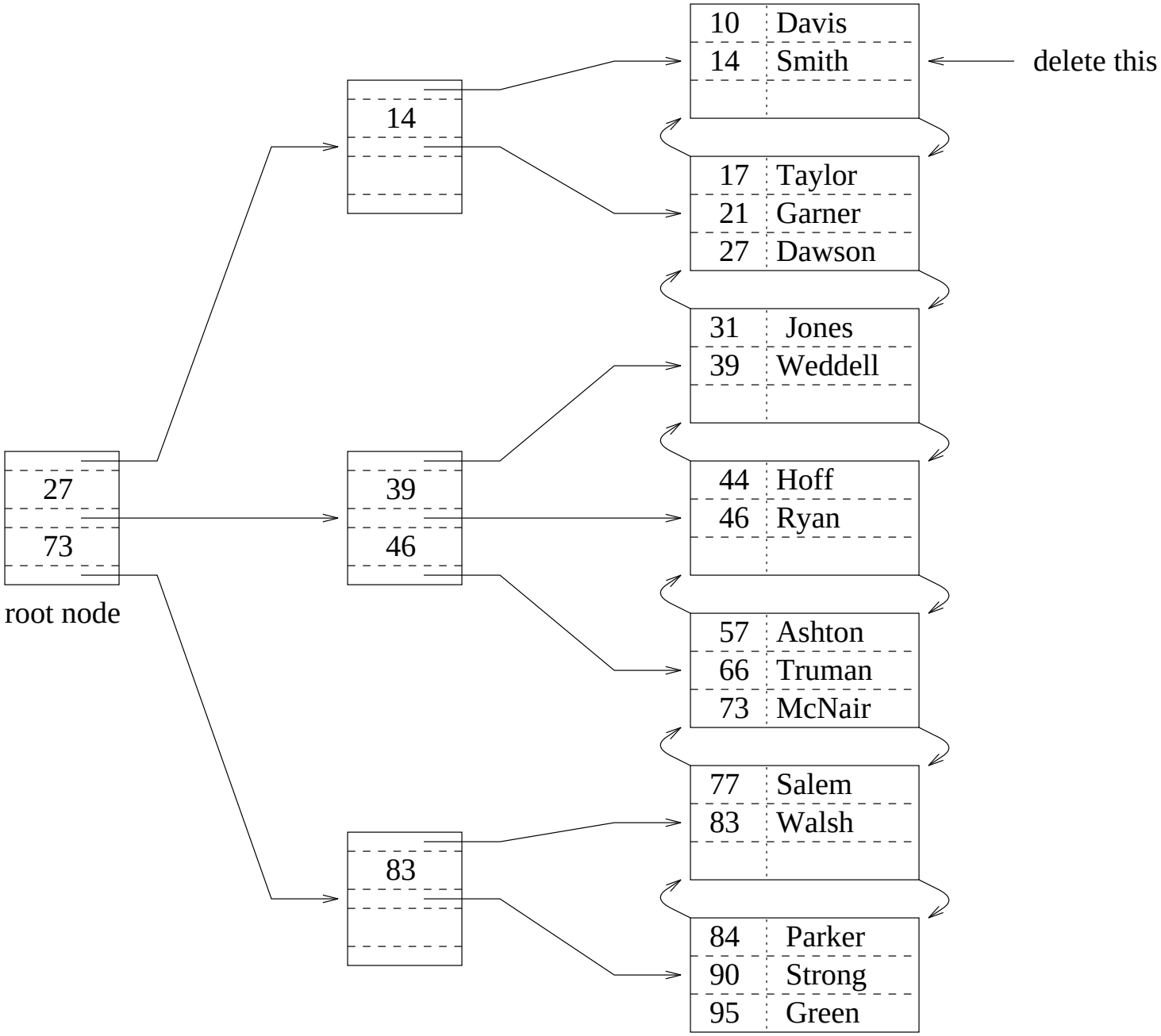


B-tree Deletions

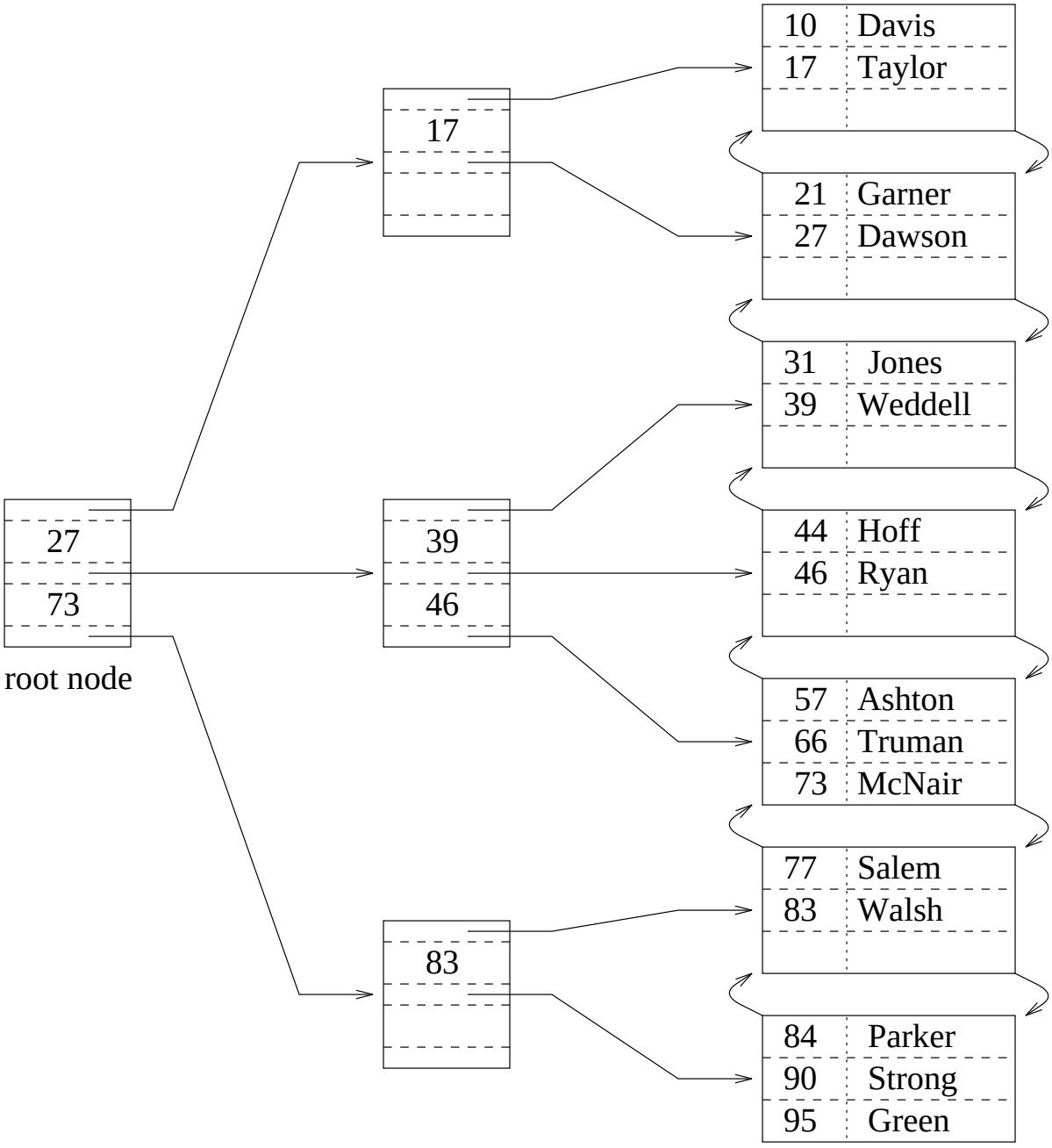
1. Determine data block where tuple is located.
2. Remove the tuple from the data block.
3. If the block is less than half full, either:
   - distribute remaining tuples to the block's sibling, remove the block from the b-tree, and delete the block's pointer from the parent index node, or
   - steal some tuples from the block's siblings, and place them in the block
4. 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.

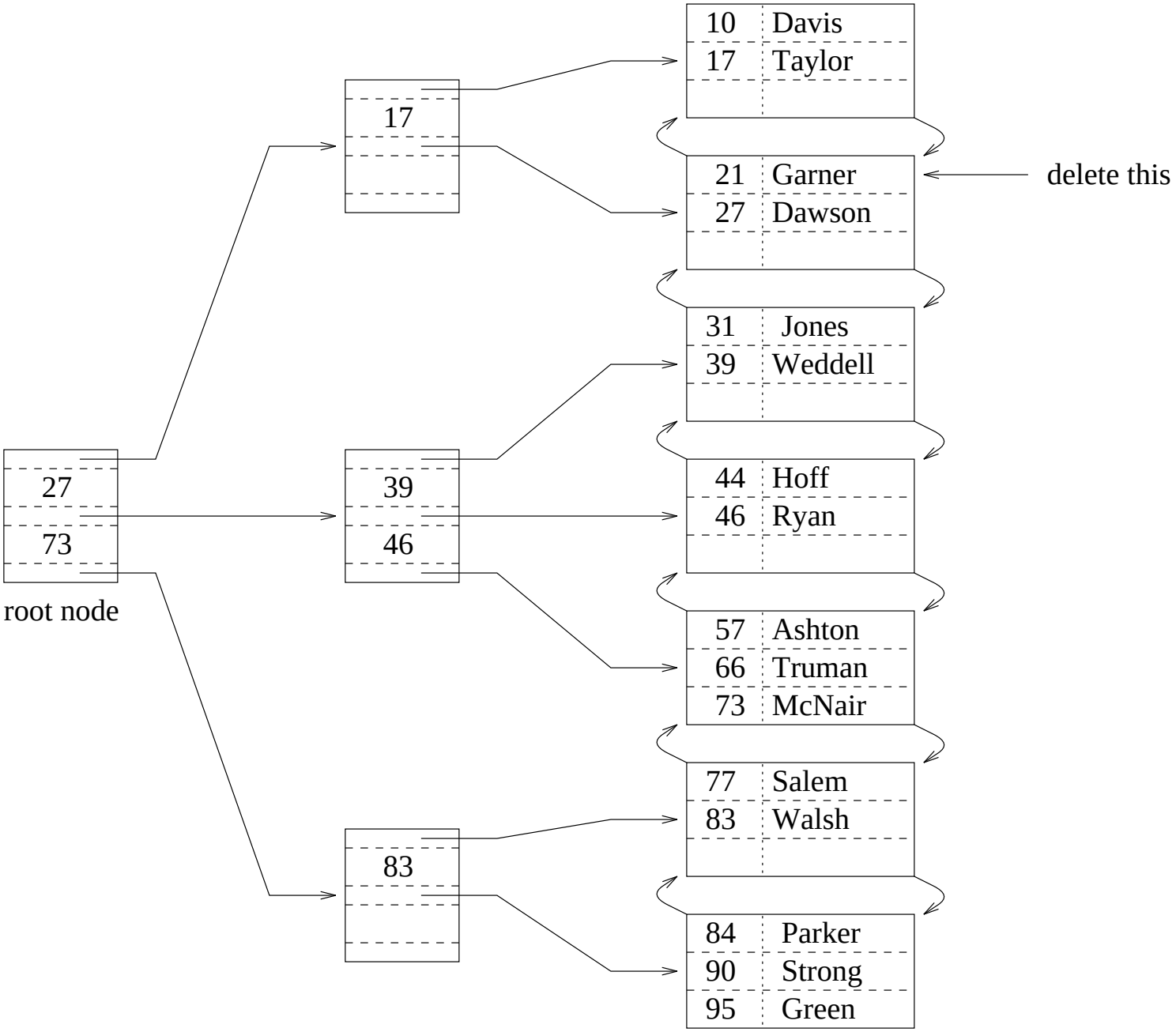
Deletion Example



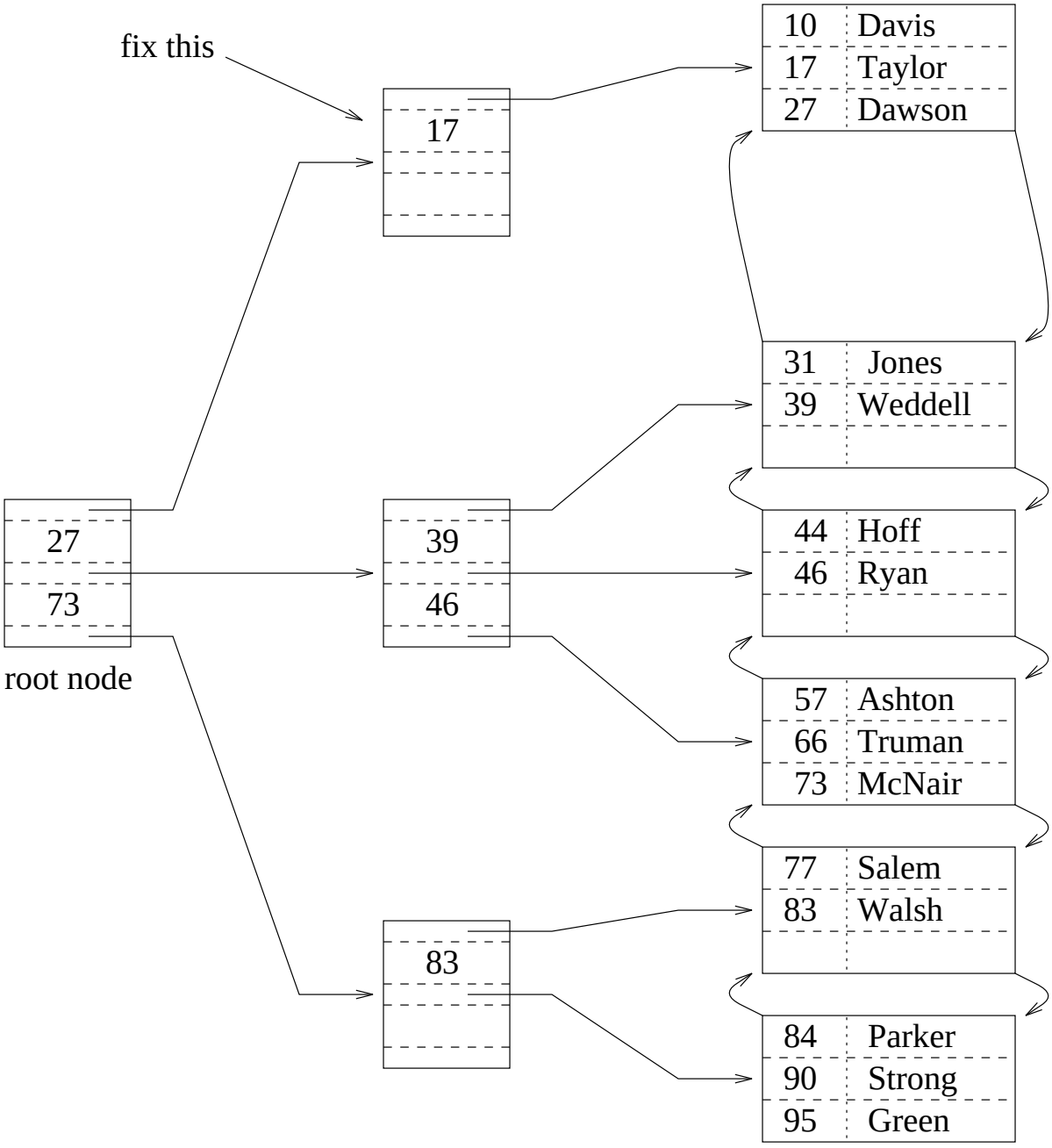
Deletion Example



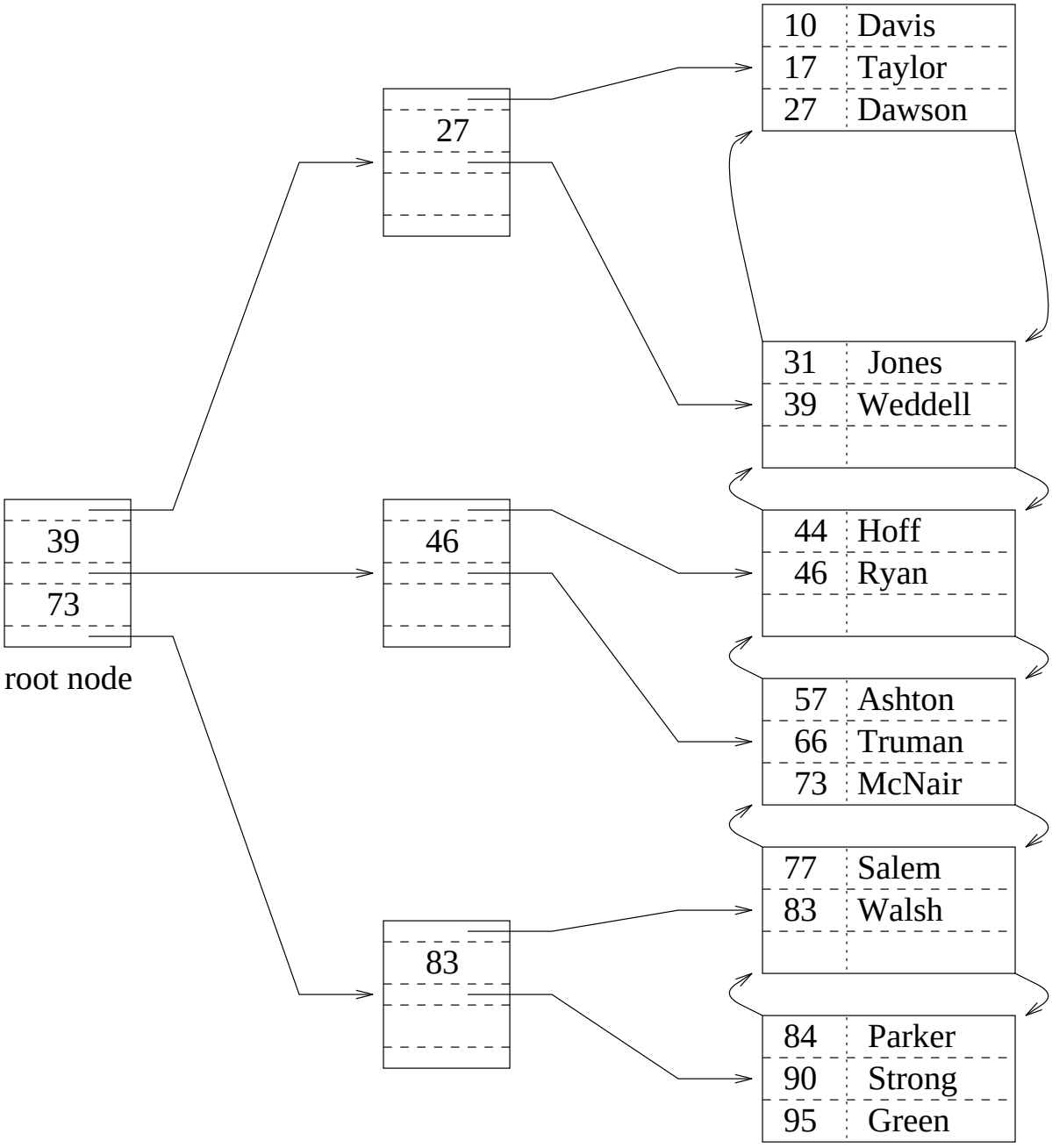
Deletion Example



Deletion Example



Deletion Example



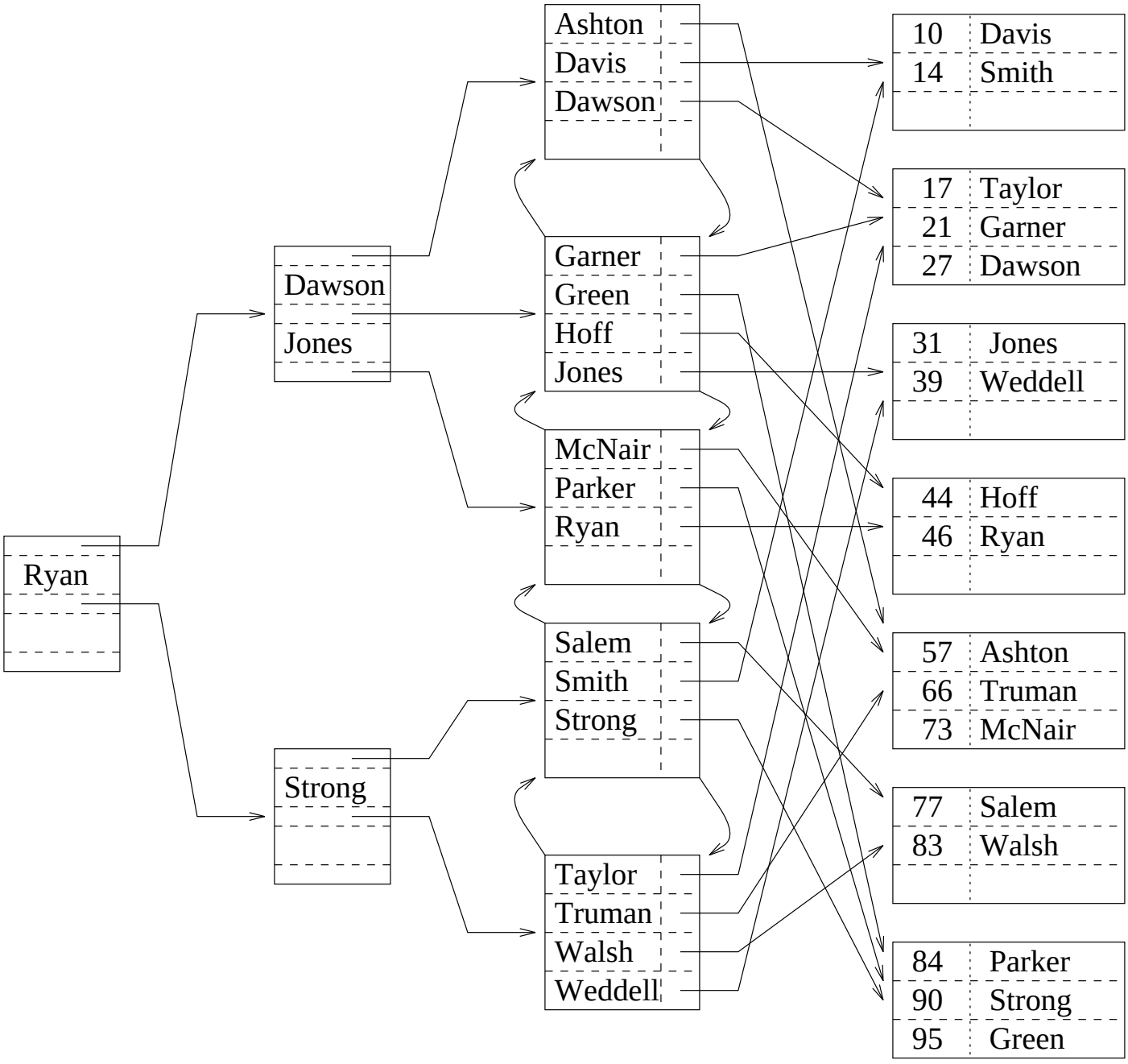
Clustering Index vs. Non-Clustering Index

An index on attribute A of a relation 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



Multi-Attribute Indices

It is possible to create an index on several attributes of the same relation. For example:

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

The order in which the attributes appear is important. In such an index, tuples (or tuple pointers) are organized first by Surname. Tuples with a common surname are then organized by Initials.

This index would be useful for a query like this:

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

or like this:

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

It would not be useful for a query like this:

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