EXTERNAL STRUCTURES AND B-TREES

by

David Toman

University of Waterloo

List of Slides

Introduction Persistent Data

- how do we store data *persistently*?

  ⇒ e.g., on disks (tapes, CDs, . . .)

- questions:
  1. why do we store data on external storage (disks)?
  2. what are the (physical) differences between accessing data in main memory vs. data in external storage?
  3. how do these impact design of data structures?

Disks

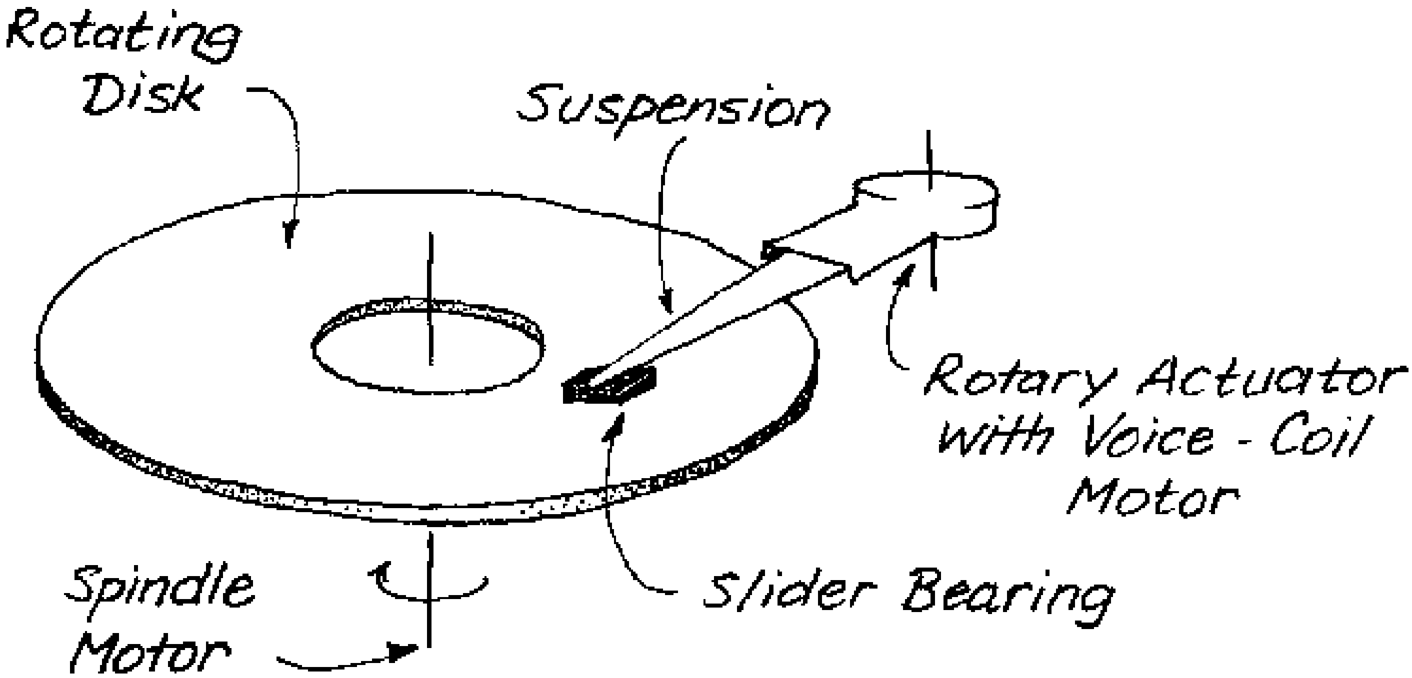


Figure 1: Major electromechanical components of a conventional high-performance rigid disk drive.

- seek time (to find the right track: 2-20ms)
- rotational delay (to get the right sector: 8-17ms)
- data transfer (1ms, 1-5MB/s)

but brings in data in **pages** (e.g., 512 bytes)

A Database Approach

- conceptually organize information in *relations*

  $\Rightarrow$ essentially an ADT

- use *disk-based* data structures to look for things

  $\Rightarrow$ looking for a record among 100k records: 120s
  $\Rightarrow$ with a (tree-like) index: less than 1s

- looking up a few records is very common

  $\Rightarrow$ support simple search conditions (point, range)
  $\Rightarrow$ additional "search" structures on top of files
  $\Rightarrow$ we trade space for execution time

- index structures

Interfaces of Access Methods

Queries: converts a sequence of requests for "next qualifying tuple" to a **sequence of database reads**.

iterator approach:

`open(I,C)`, **`get-next(I)`**, **`close(I)`**

$\Rightarrow$ returns a pointer (if found) or NULL (if not)
$\Rightarrow$ uses **`DB-read(rid)`** to get records into buffers
$\Rightarrow$ can be optimized if only one tuple always qualifies

Updates:

insertion and **deletion** based on record id

`insert(I,Rid)`, **`delete(I,Rid)`**

$\Rightarrow$ we need to *create/find* the record first (usually)

Primary vs. Secondary Indices

- in general, the **data entries** are stored
  1. directly in the index:

     $\Rightarrow$ a data record with key value k

     $\Rightarrow$ a **primary index** (often sparse)
  2. indirectly, using a foreign rid:

     $\Rightarrow$ a key value k and the matching rid pair

     $\Rightarrow$ a **secondary index** (usually dense)
- in case of duplicates:

  $\Rightarrow$ overflow chains
- this choices are (almost) orthogonal to the choice of the indexing method.

Tree-based Indices

- **Idea:** tree based data structure allow search in $\log n$

  $\Rightarrow$ how to make them work on disk?

  $\Rightarrow$ how to make them dynamic?

- B-trees:

  $\Rightarrow$ are widely-used index structures.

  $\Rightarrow$ are dynamic: they grow and shrink easily.

  $\Rightarrow$ have two parts: index blocks and data blocks.

  $\Rightarrow$ index and data blocks are kept at least half full.

- many other variants: R-tree, R*-tree, HB-tree, . . .

- even B-trees come in several flavours;

  $\Rightarrow$ we will discuss B+ trees.

B-tree blocks

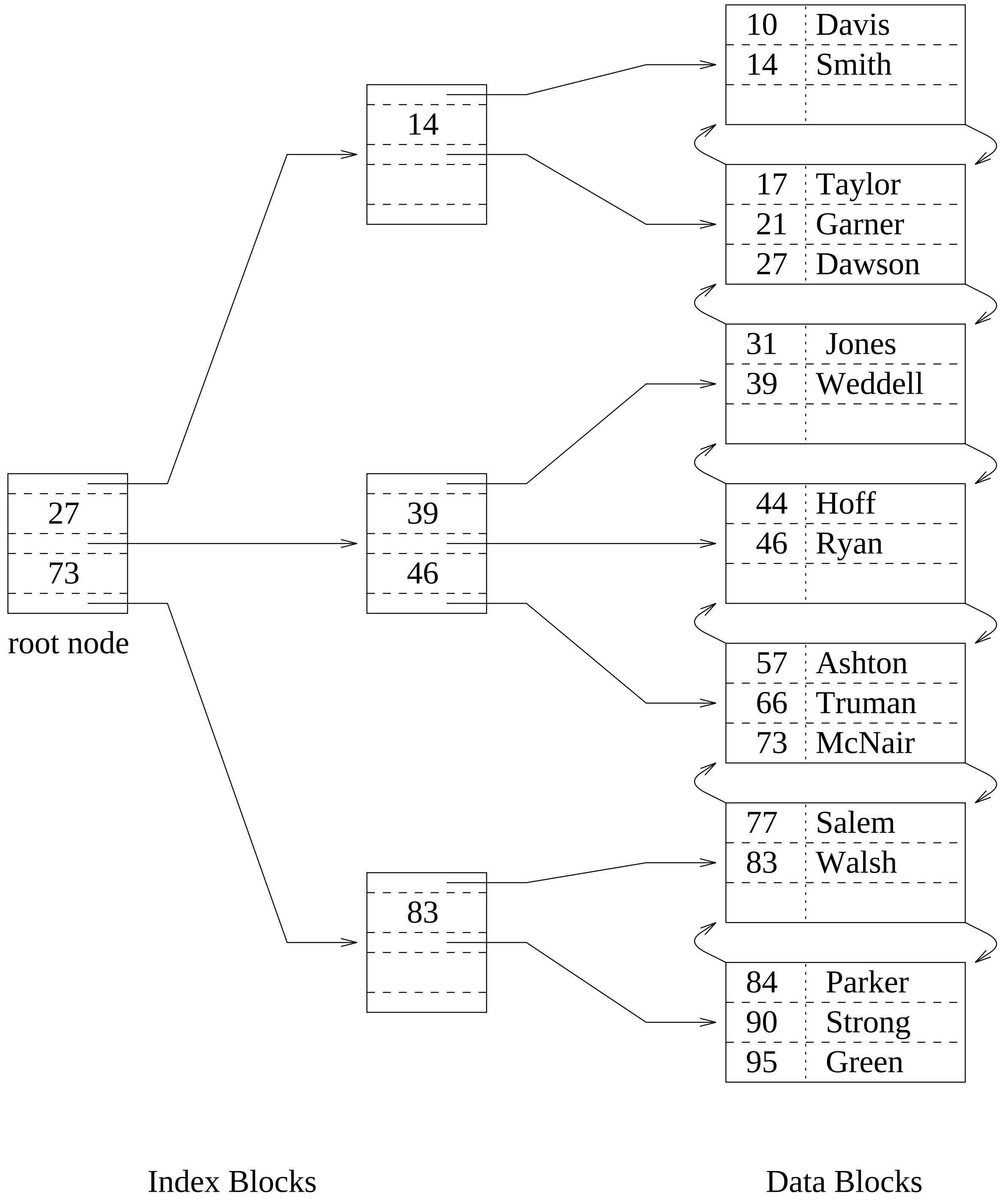
- Index blocks
  - $\Rightarrow$ each block stores at most m values and $m+1$ pointers
  - $\Rightarrow$ each block stores at least $\lfloor m/2 \rfloor$ values and $\lfloor m/2 \rfloor + 1$ pointers

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

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

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

Example: Sparse B+ Tree



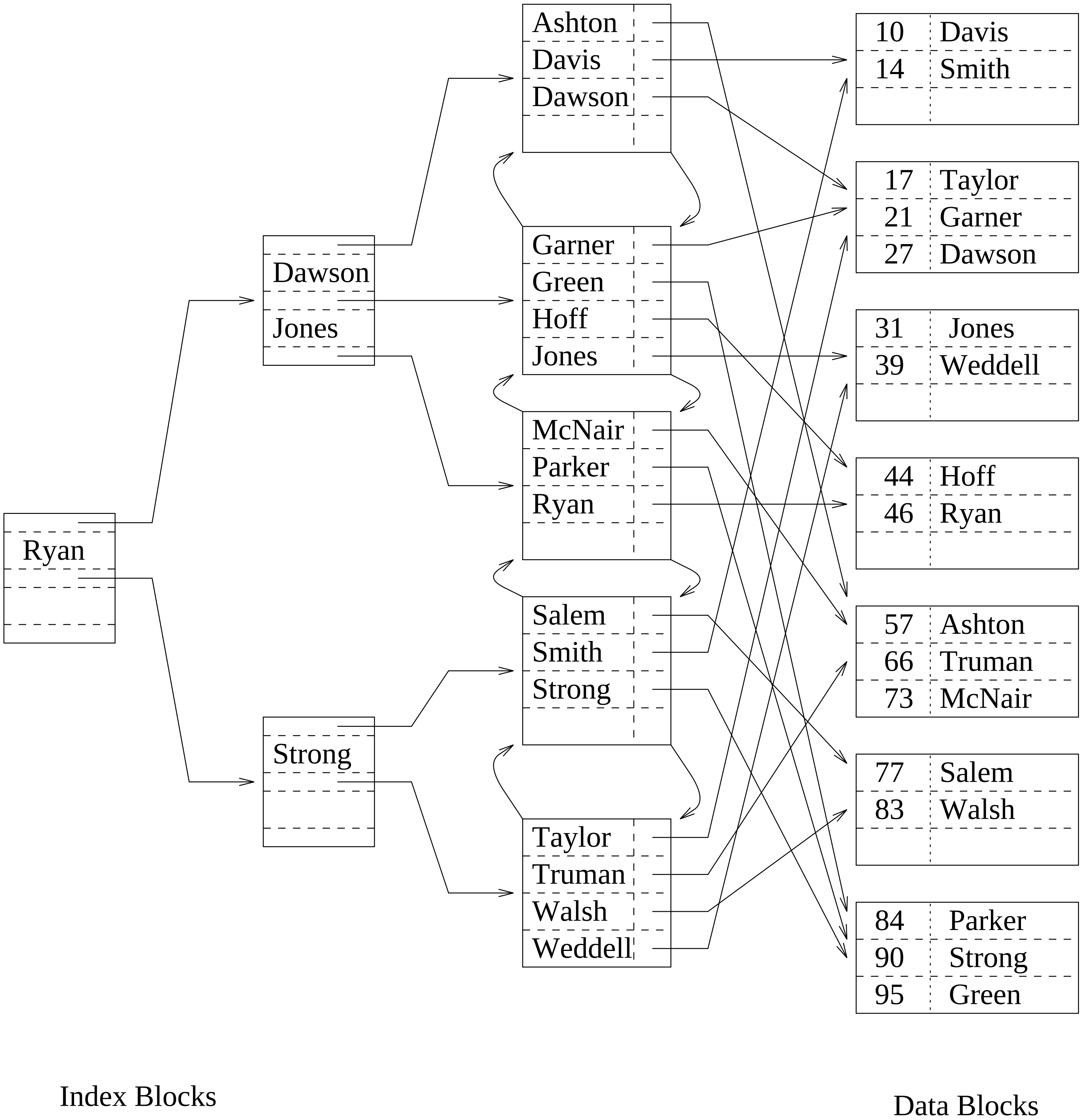
Searching a B+ Tree

- searching for all records that match a value of the search attribute is a simple recursive procedure:

```
iterator search(value key, rid root) {
  blk = DB-read(root);
  for(i=0; i<blk.N && blk.key[i]<key; i++);
  if (blk.is_leaf)
    return make-iterator(key,blk.rid[i]);
  else
    search(key,blk.rid[i]);
}
```

- the **`make-iterator`** depends on how the *data leaves* are organized (primary, secondary, with duplicates, etc.).
- *range queries* use the forward/backward pointers in the data blocks to get all the tuples in the requested range.

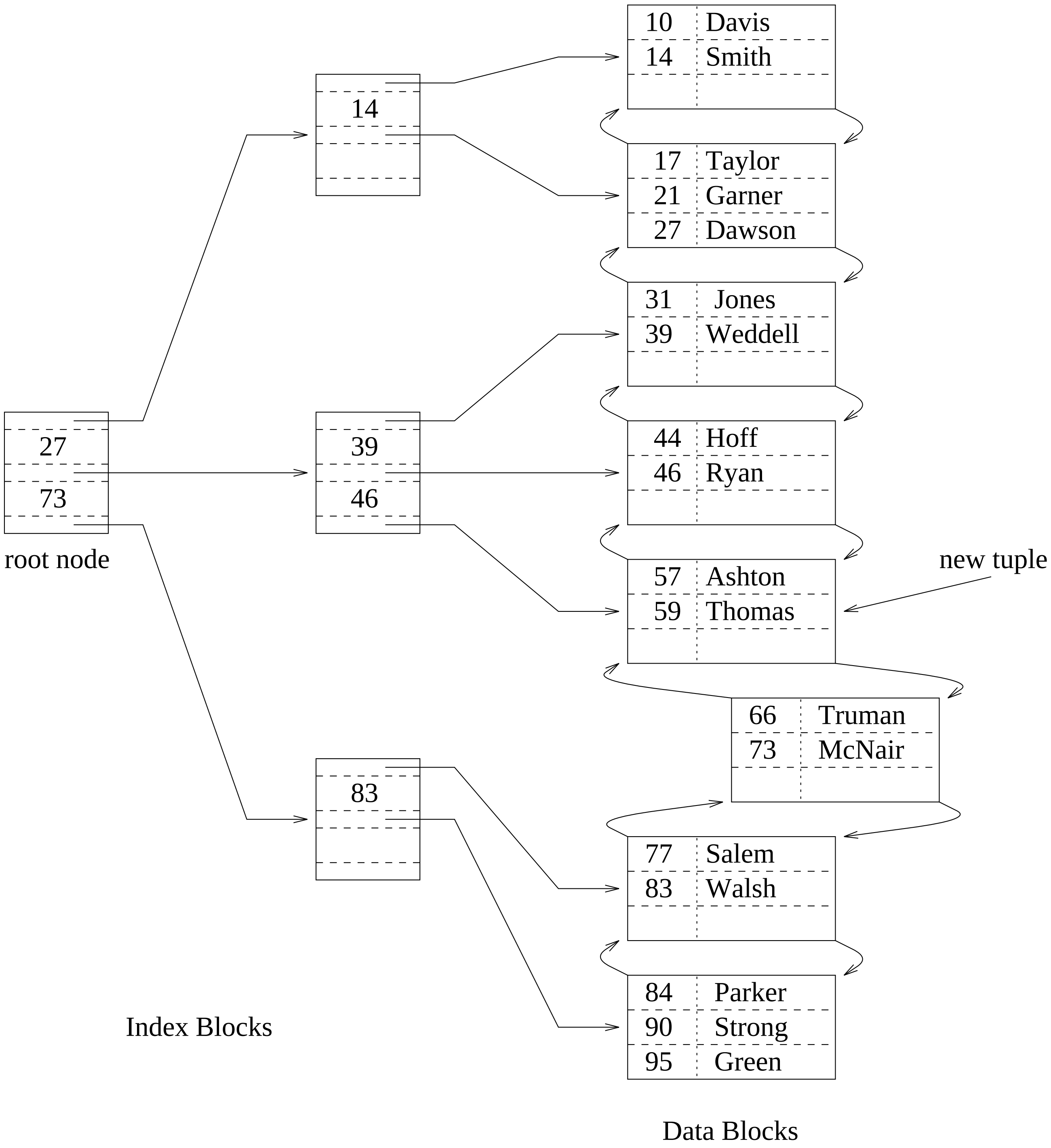
Example: Dense B+ Tree



Insertion into a B+ Tree

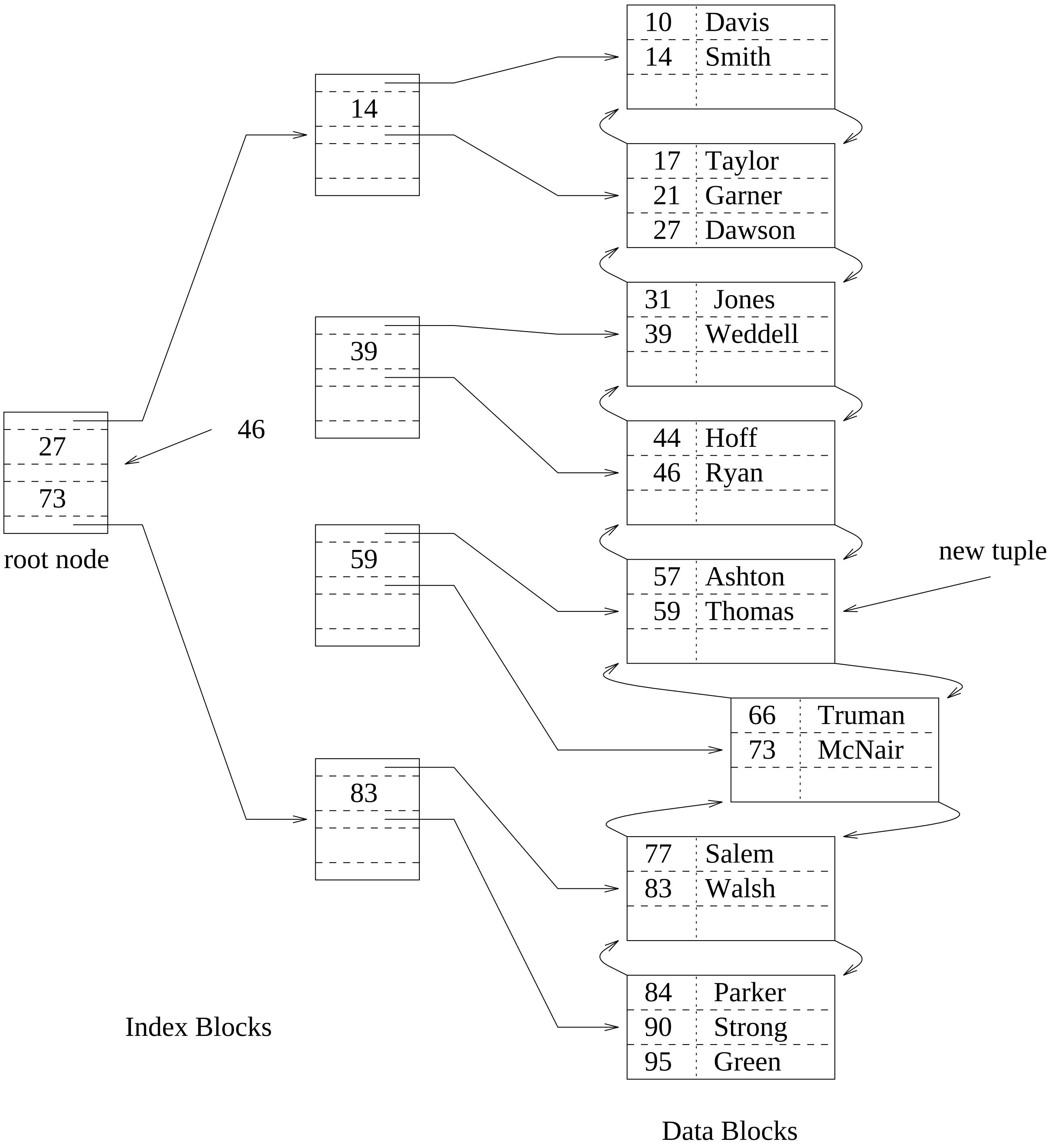
1. Determine data block where new tuple belongs.
2. If there is room in the block, place the tuple in it.
3. If there is no room, find an empty block, and move half of the records into the new block. This is called **splitting**.
4. Add an entry for the new block in the parent index block.
5. If the 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.

Example

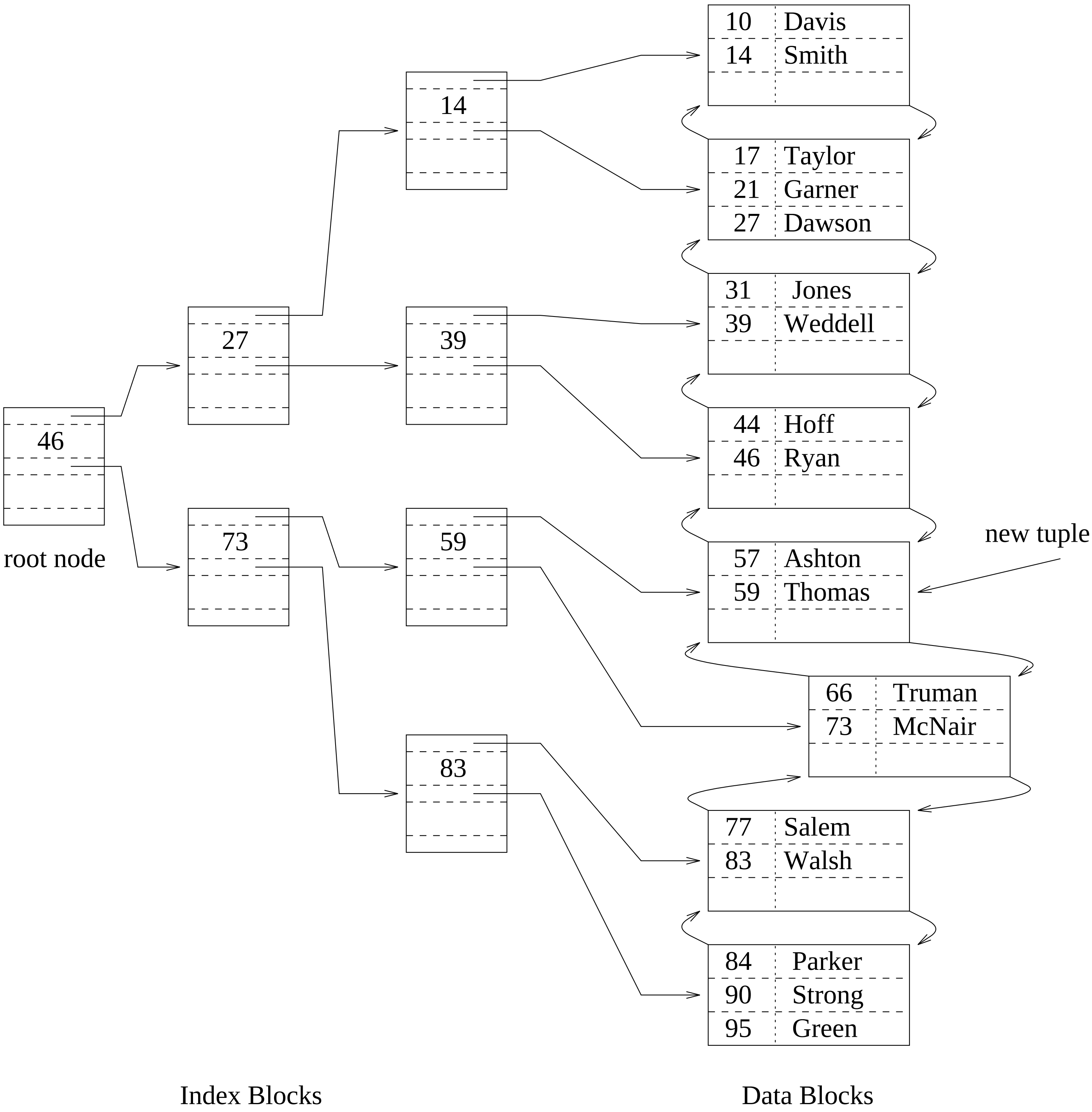


10 Davis
14 Smith
14
17 Taylor
21 Garner
27 Dawson
31 Jones
39 Weddell
27
73
39
46
44 Hoff
46 Ryan
root node
57 Ashton
59 Thomas
new tuple
66 Truman
73 McNair
83
77 Salem
83 Walsh
84 Parker
90 Strong
95 Green
Index Blocks
Data Blocks

Example (cont.)



Example (cont.)

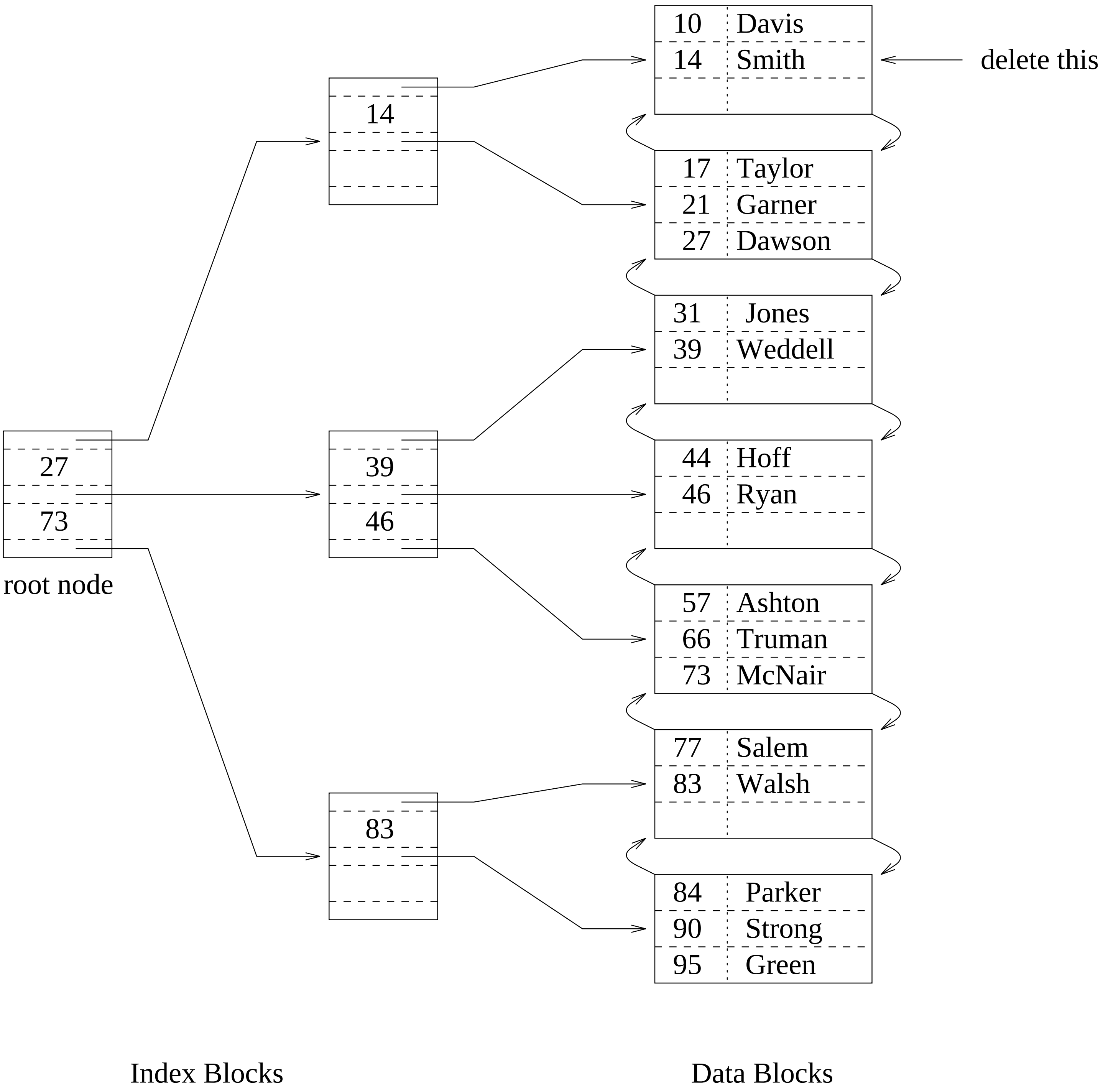


Deletion from a B+ Tree

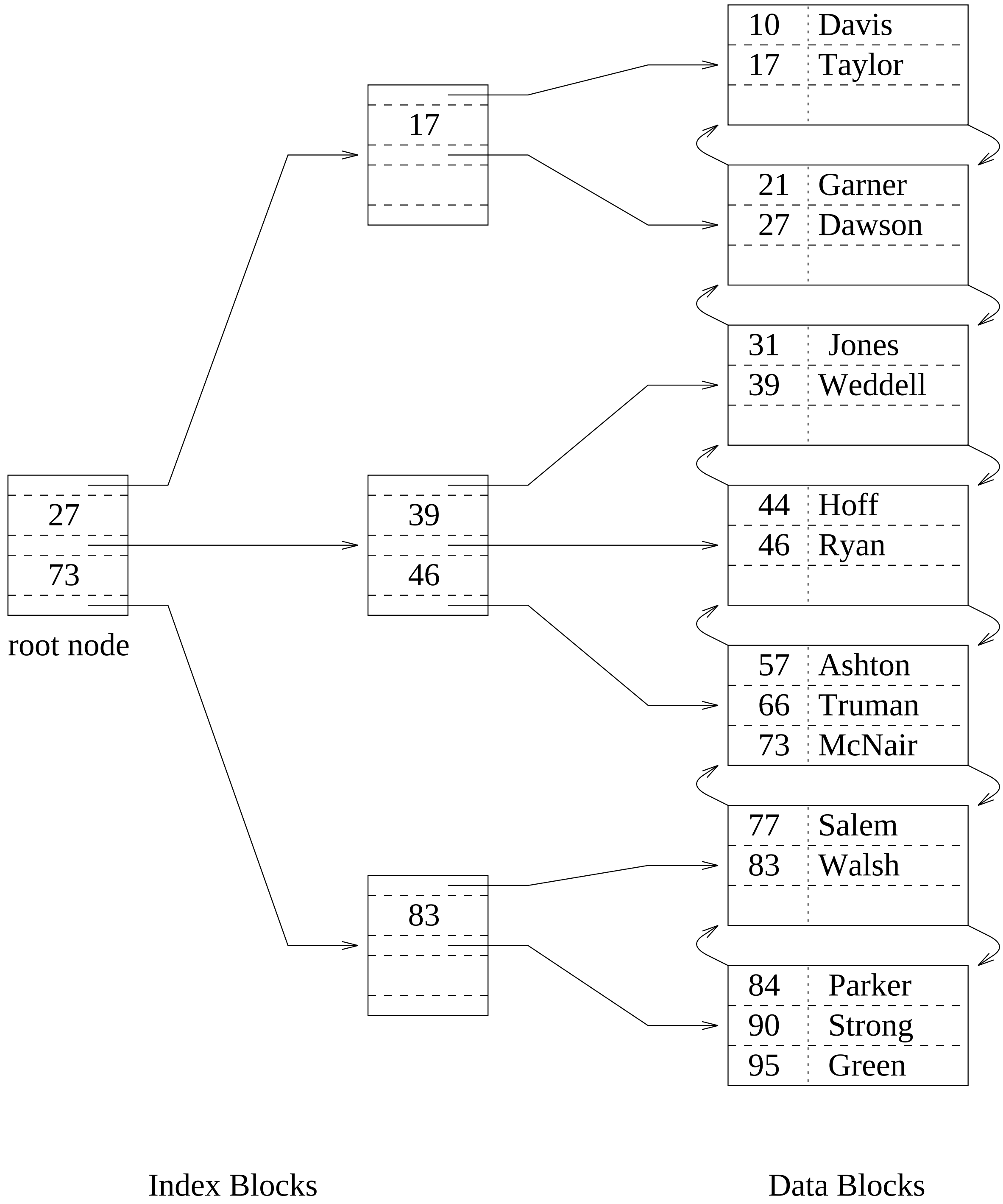
1. Determine data block where tuple is located.
2. Remove the tuple from the data block.
3. If the block is less than half full, either:
   - distribute remaining tuples to the block's sibling, remove the block from the b-tree, and delete the block's pointer from the parent index node, or
   - steal 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 all cases, blocks that remain in the b-tree must be at least half full.

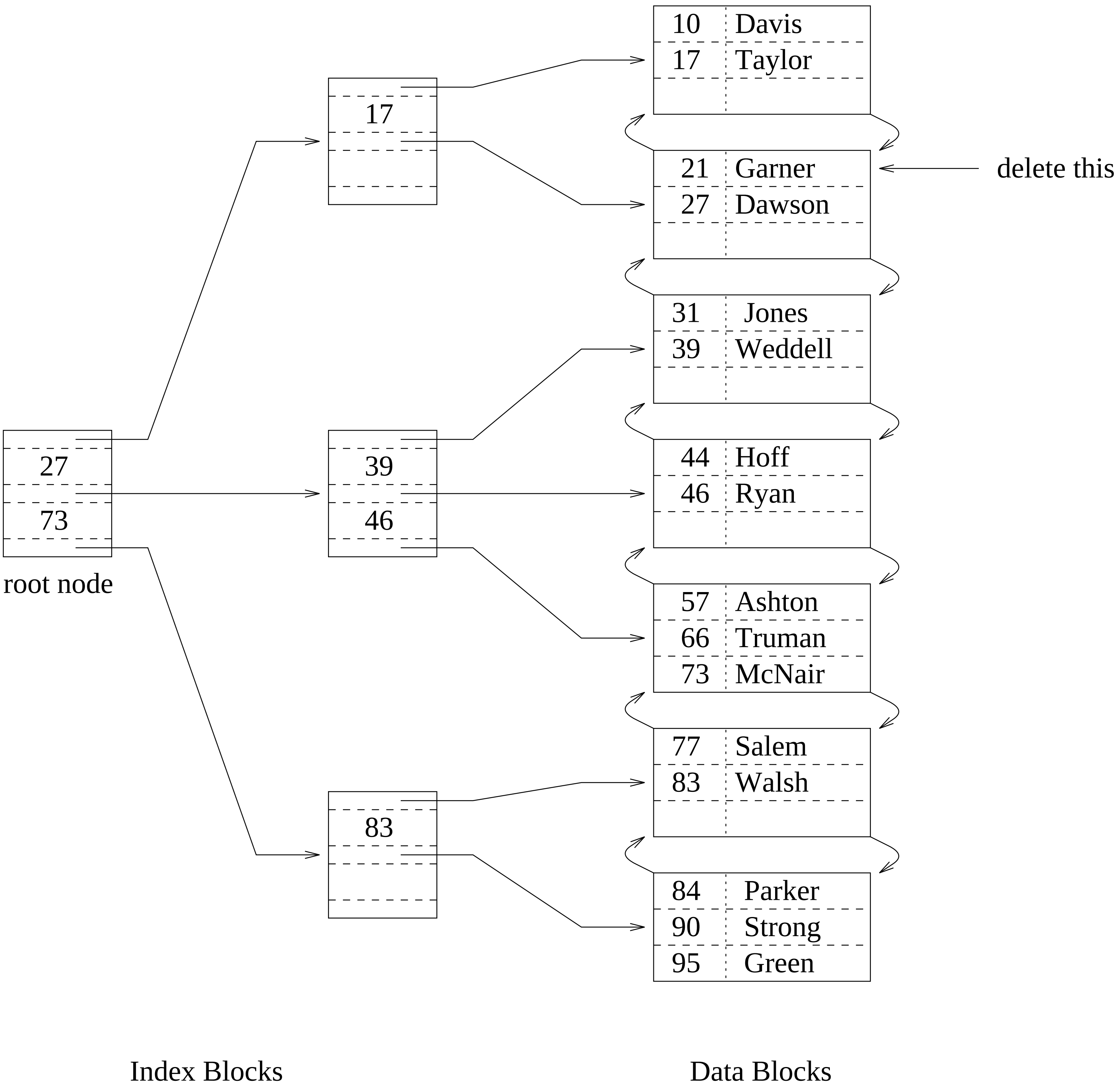
Example


10 Davis
14 Smith
delete this
14
17 Taylor
21 Garner
27 Dawson
31 Jones
39 Weddell
27
73
root node
39
46
44 Hoff
46 Ryan
57 Ashton
66 Truman
73 McNair
77 Salem
83 Walsh
83
84 Parker
90 Strong
95 Green
Index Blocks
Data Blocks

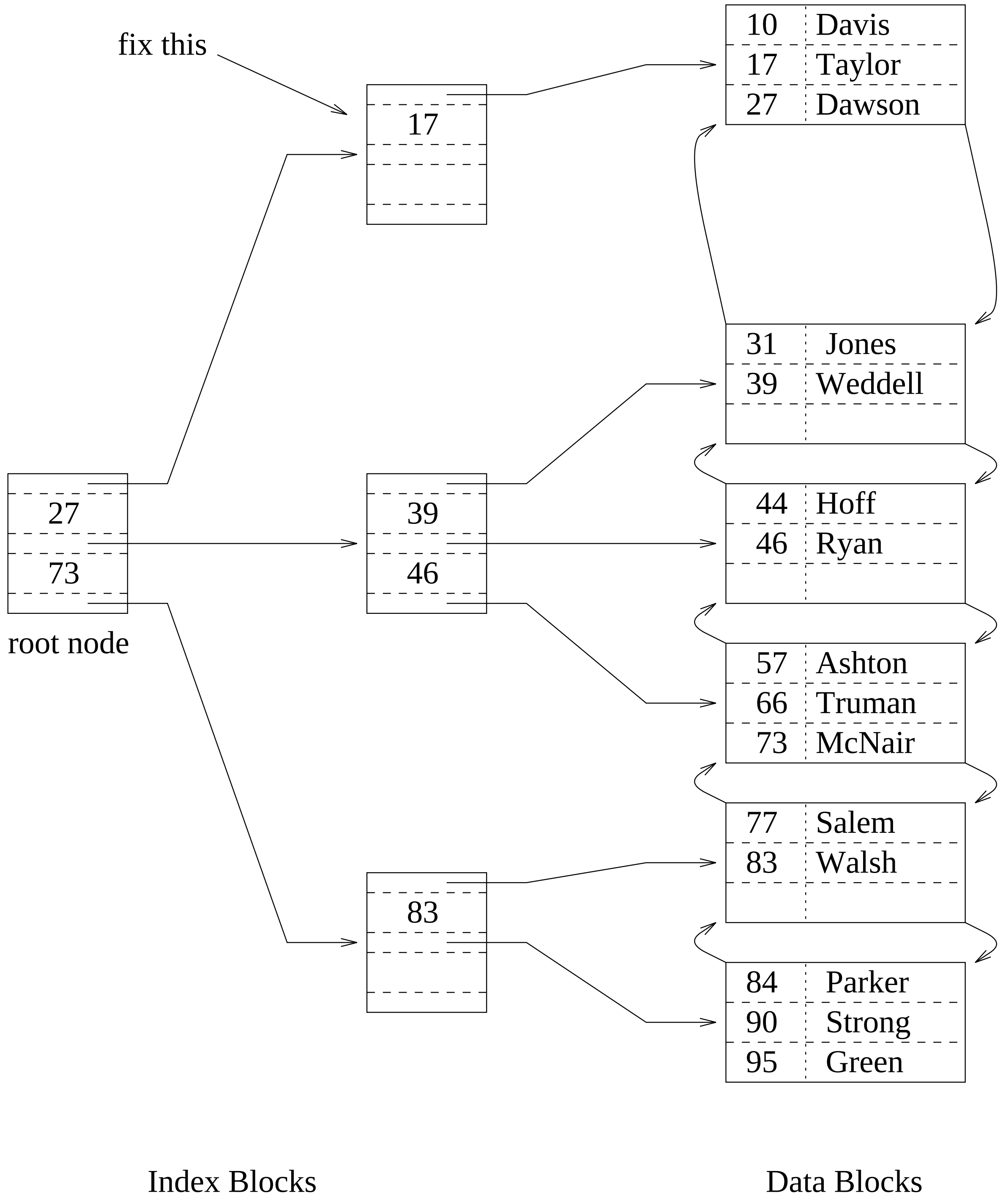
Example (cont.)



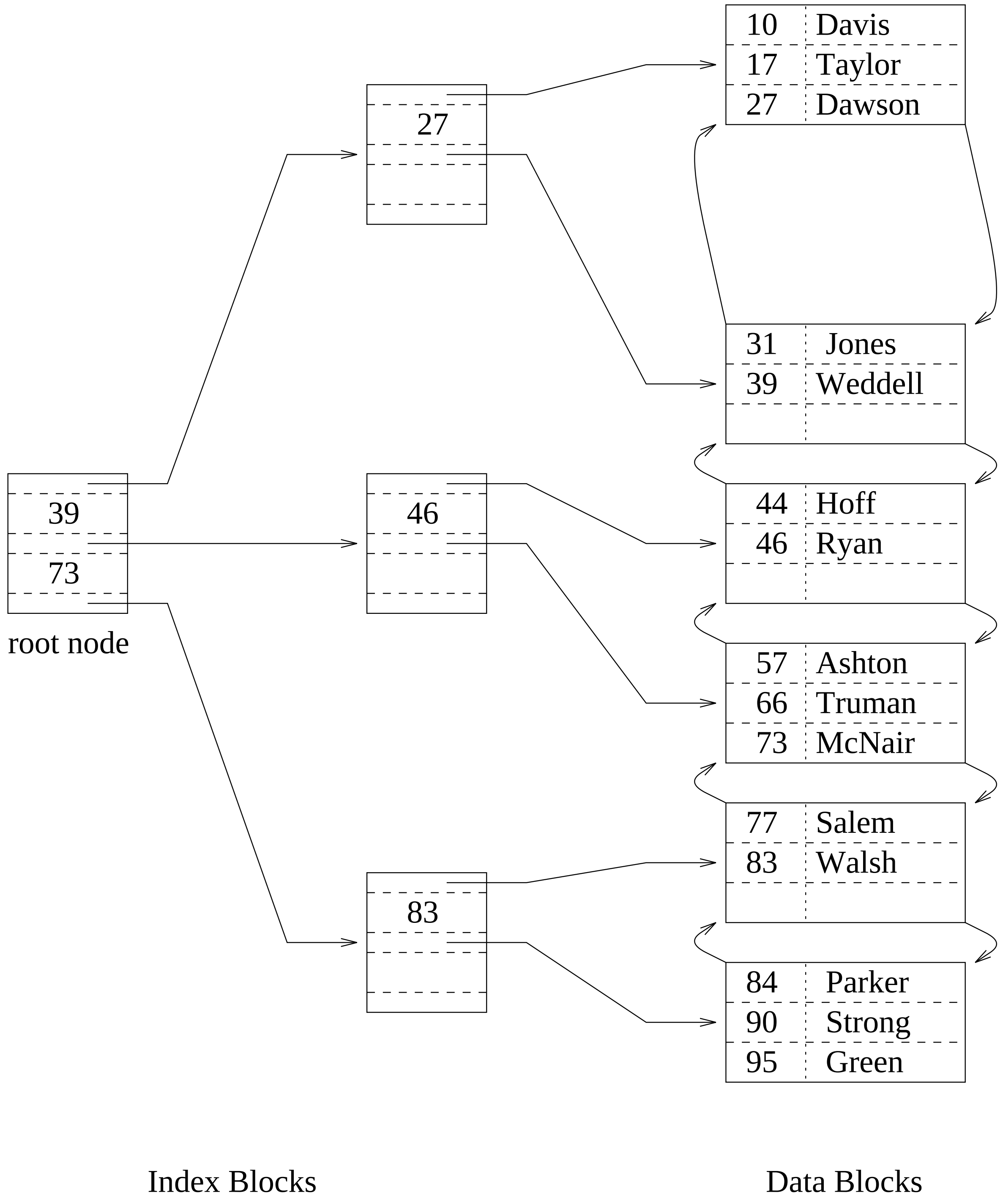
Example (cont.)



Example (cont.)



Example (cont.)



Costs in a B+ Tree

Worst case cost (in PAGE READS):

1. Search: $O(log_{l/2}n)$

   where l is the fanout and

   n the number of *distinct* values

   $\Rightarrow$ typically < 5 page accesses (as $l > 100$).

2. Insert/Delete: $O(log_{l/2}n)$

$\Rightarrow$ in all cases: the splits/recombinations
are **amortized** over $l/2$ operations.

Summary

- access methods translate requests for all records (satisfying a particular *search criterion* to a sequence of physical record reads.
- considerations: cost of search vs. cost of updates
- cost measured in number of page I/O
- extensions:

  $\Rightarrow$ multi-attribute indices

  $\Rightarrow$ multi-dimensional indices