BTree Indexes

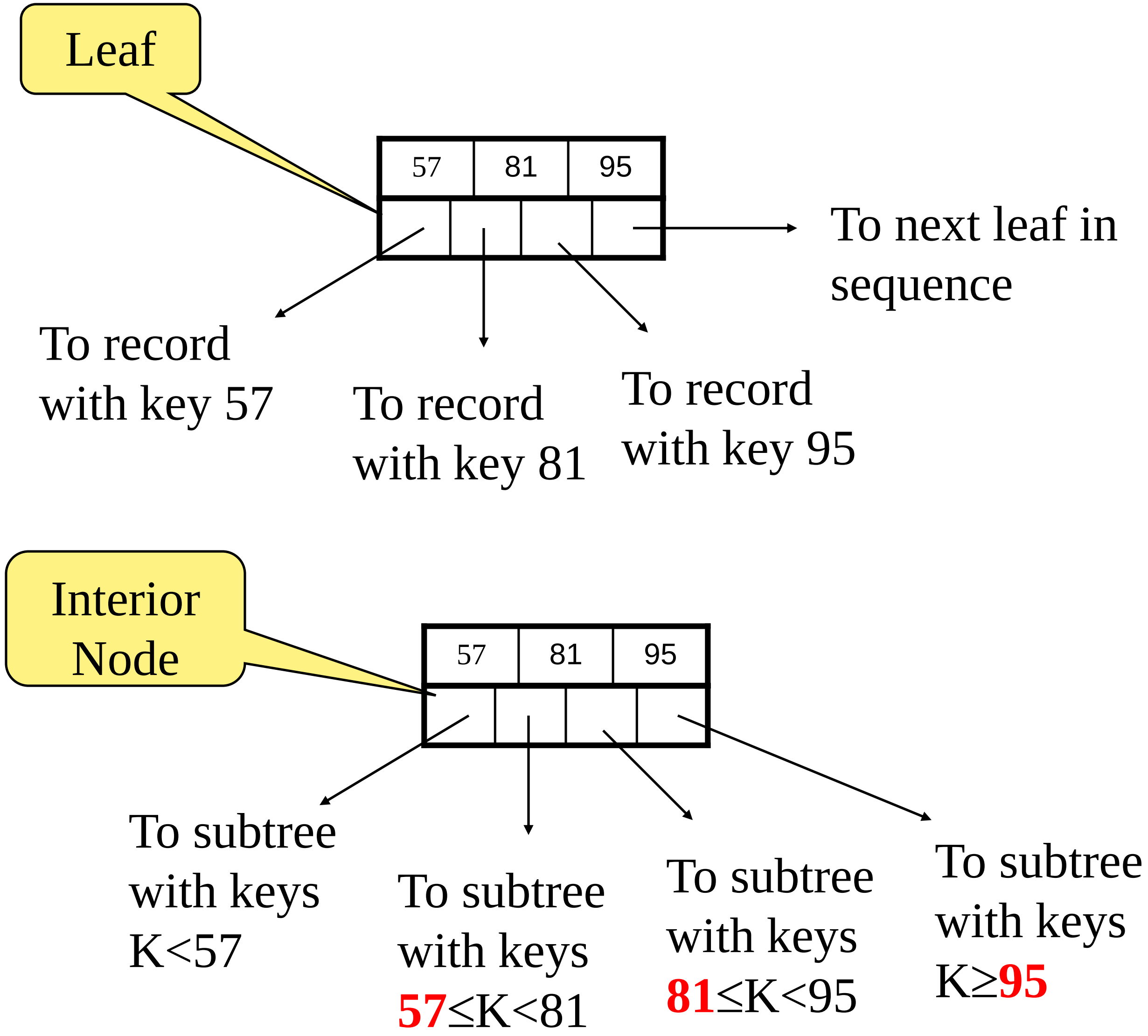
Why Indexes?

- Help answering a lookup query like

```
SELECT *
FROM R
WHERE a=10;
```

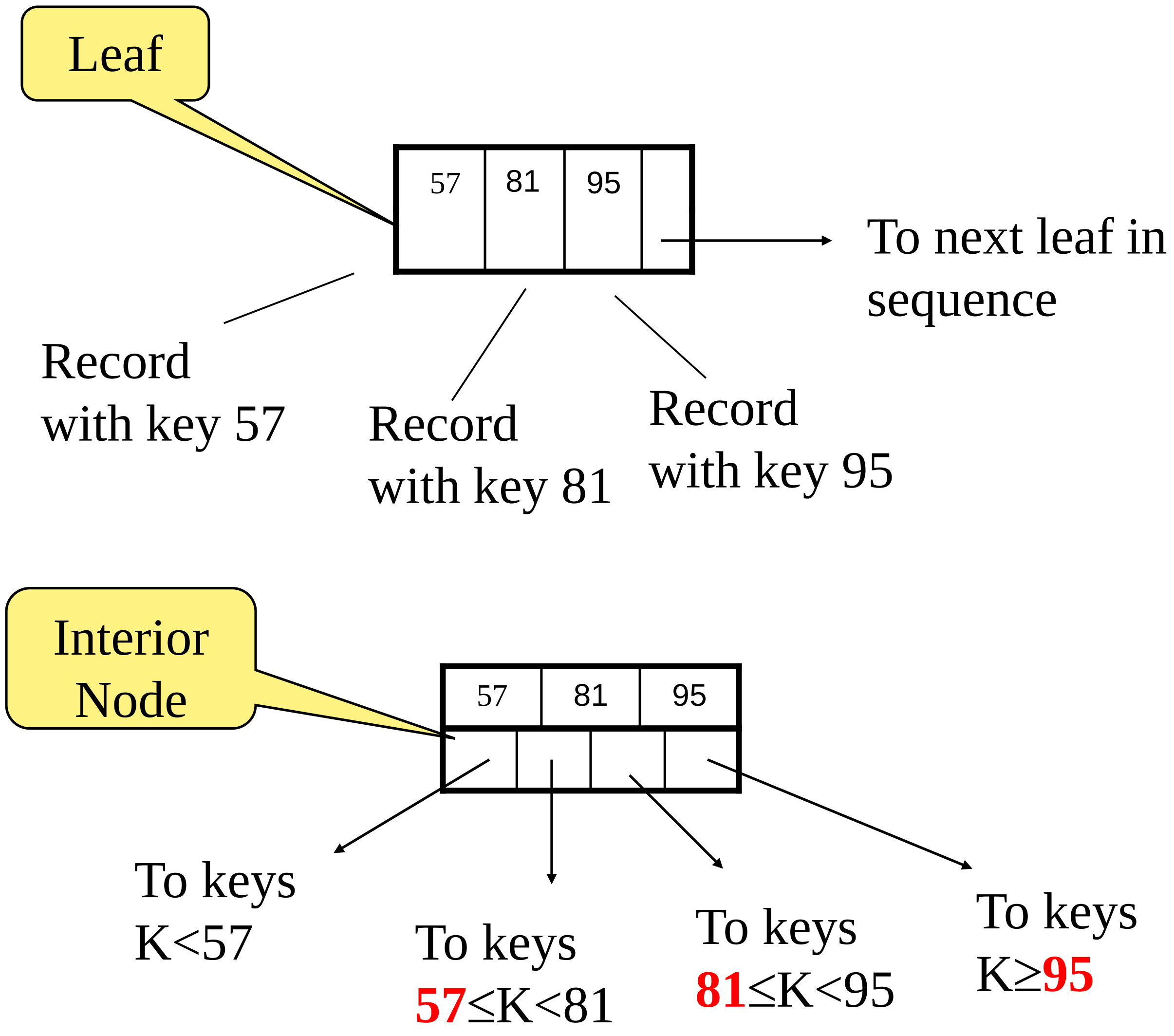
- An **index** is a **data-structure** that
  - takes the value of one or more attributes and
  - finds the records with that value “quickly”
  - without having to look at more than a small fraction of all possible records

BTrees, Un-clustered index: A leaf and interior node



57, **81**, and **95** are the least keys we can reach by via the corresponding pointers.

Clustered index: Leaf with records and interior node



57, **81**, and **95** are the least keys we can reach by via the corresponding pointers.

Types of B-Trees

- Un-clustered index
  - Key-pointers in the leaves

- clustered index
  - Records in the leaves

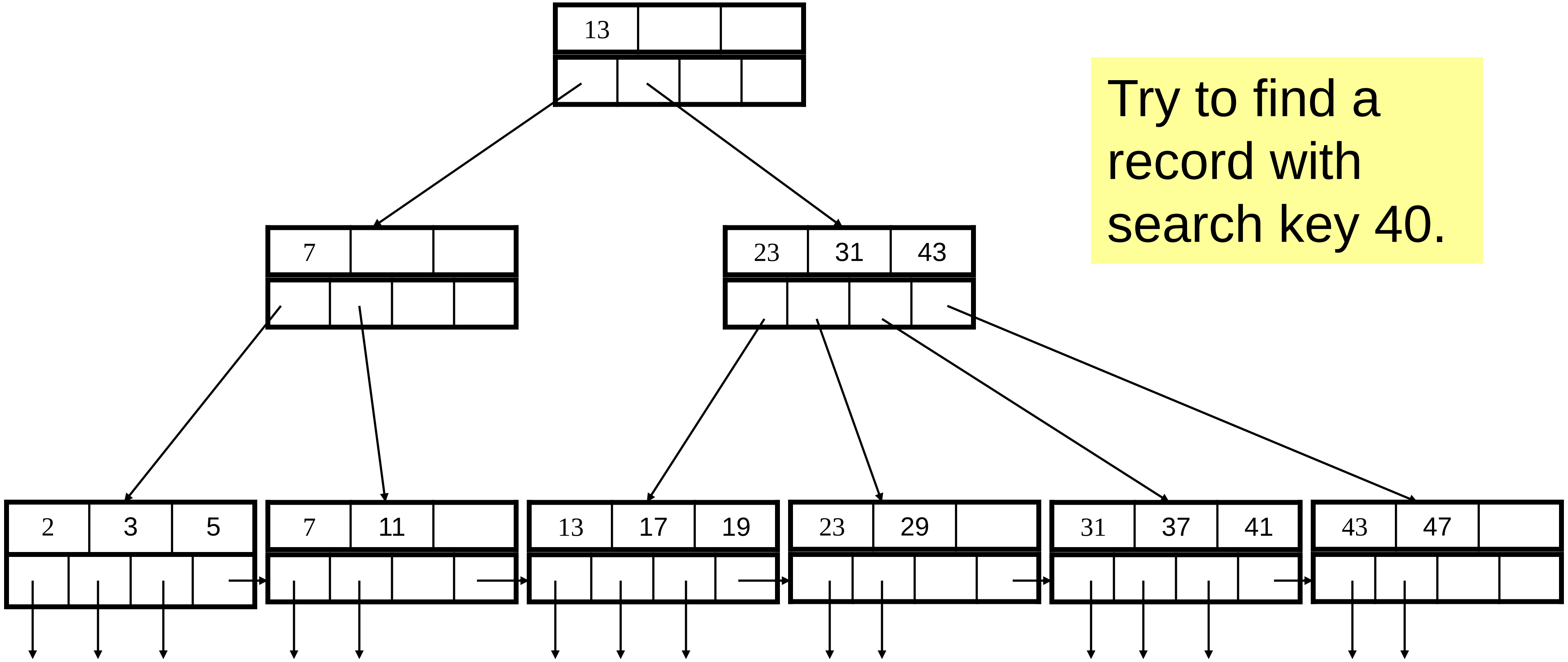
Operations in B-Tree

Will illustrate with unclustered case, but straightforward to generalize for the clustered case.

Operations

1. Lookup
2. Insertion
3. Deletion

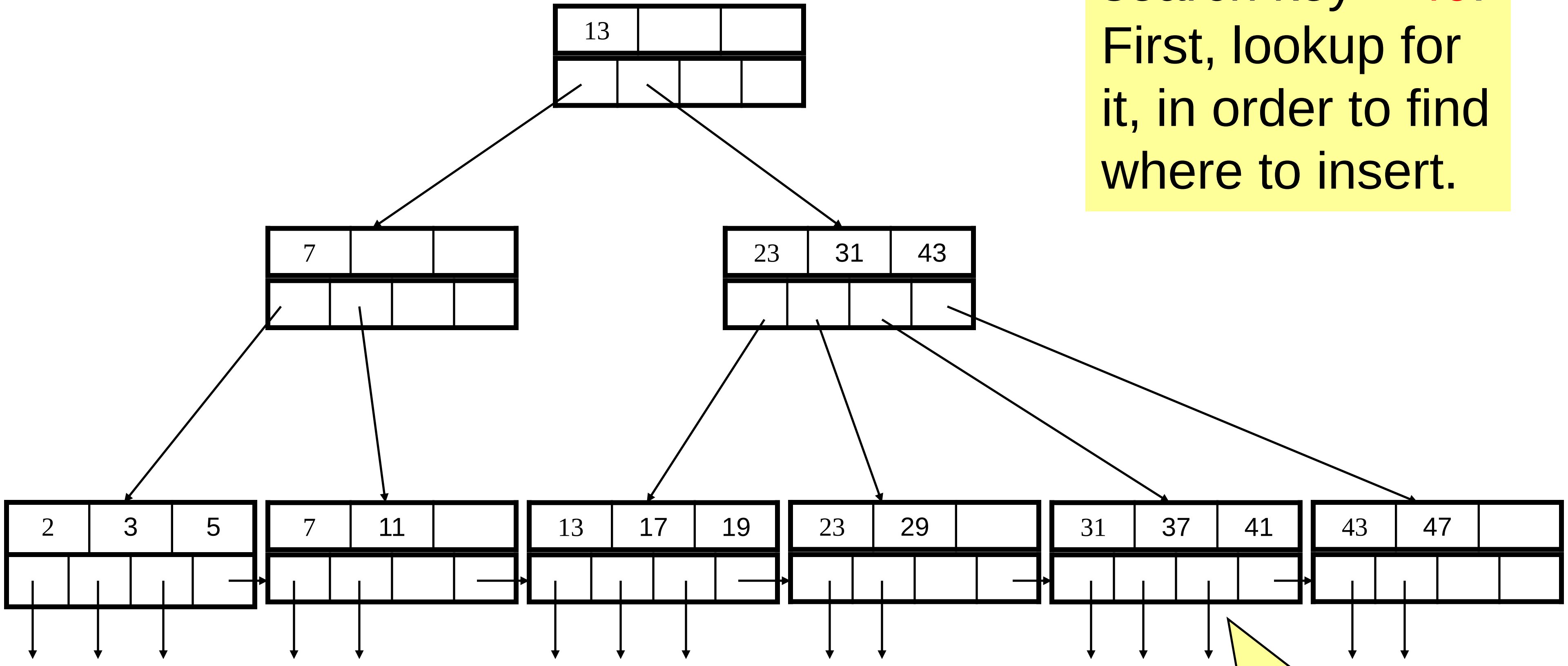
Lookup



Try to find a record with search key 40.

Recursive procedure:
If we are at a leaf, look among the keys there. If the *i*-th key is K, the the *i*-th pointer will take us to the desired record.
If we are at an internal node with keys $K_1, K_2, \ldots, K_n$, then if $K<K_1$ we follow the first pointer, if $K_1 \leq K<K_2$ we follow the second pointer, and so on.

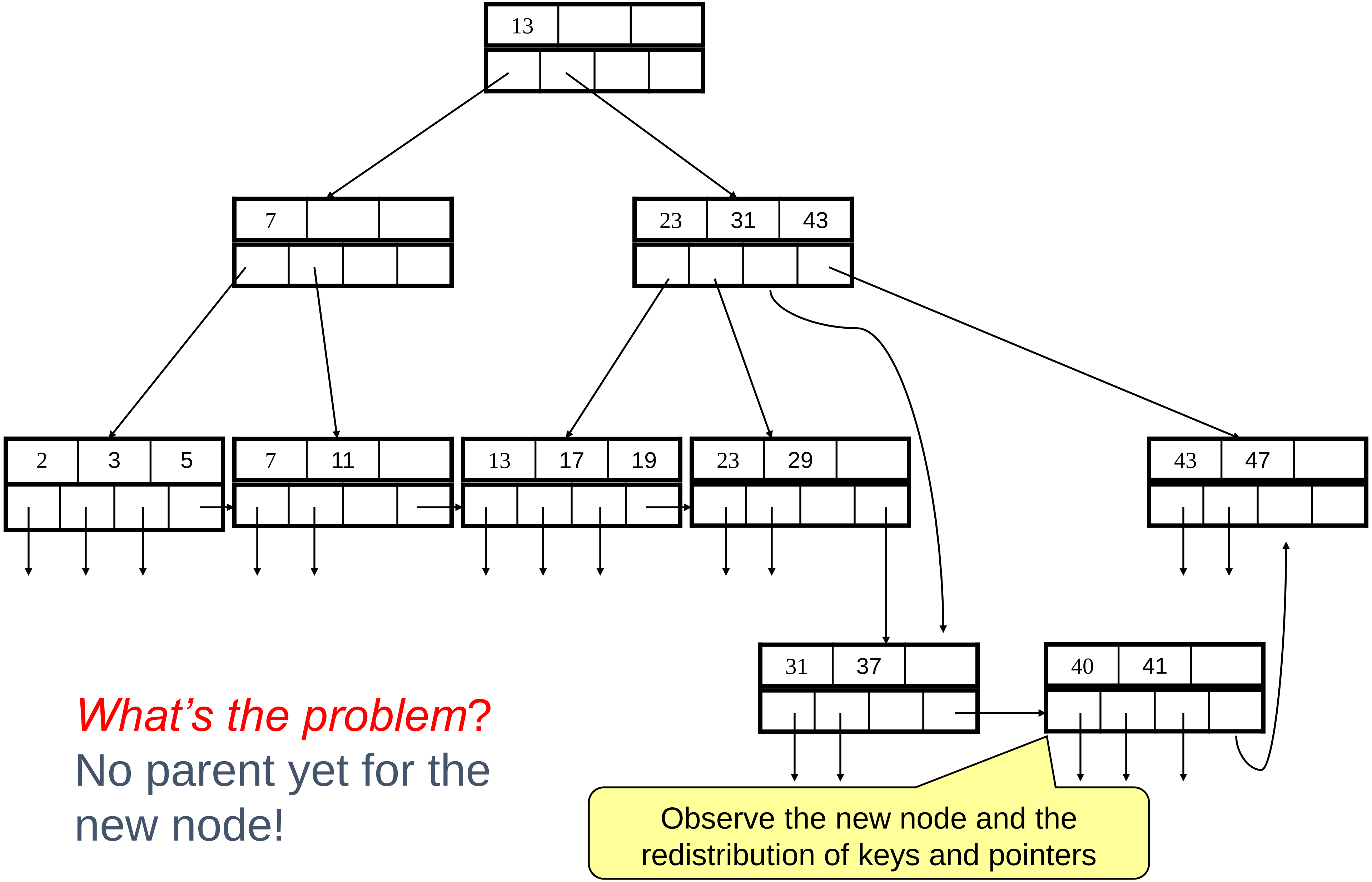
Insertion



Try to insert a search key = 40. First, lookup for it, in order to find where to insert.

It has to go here, but the node is full!

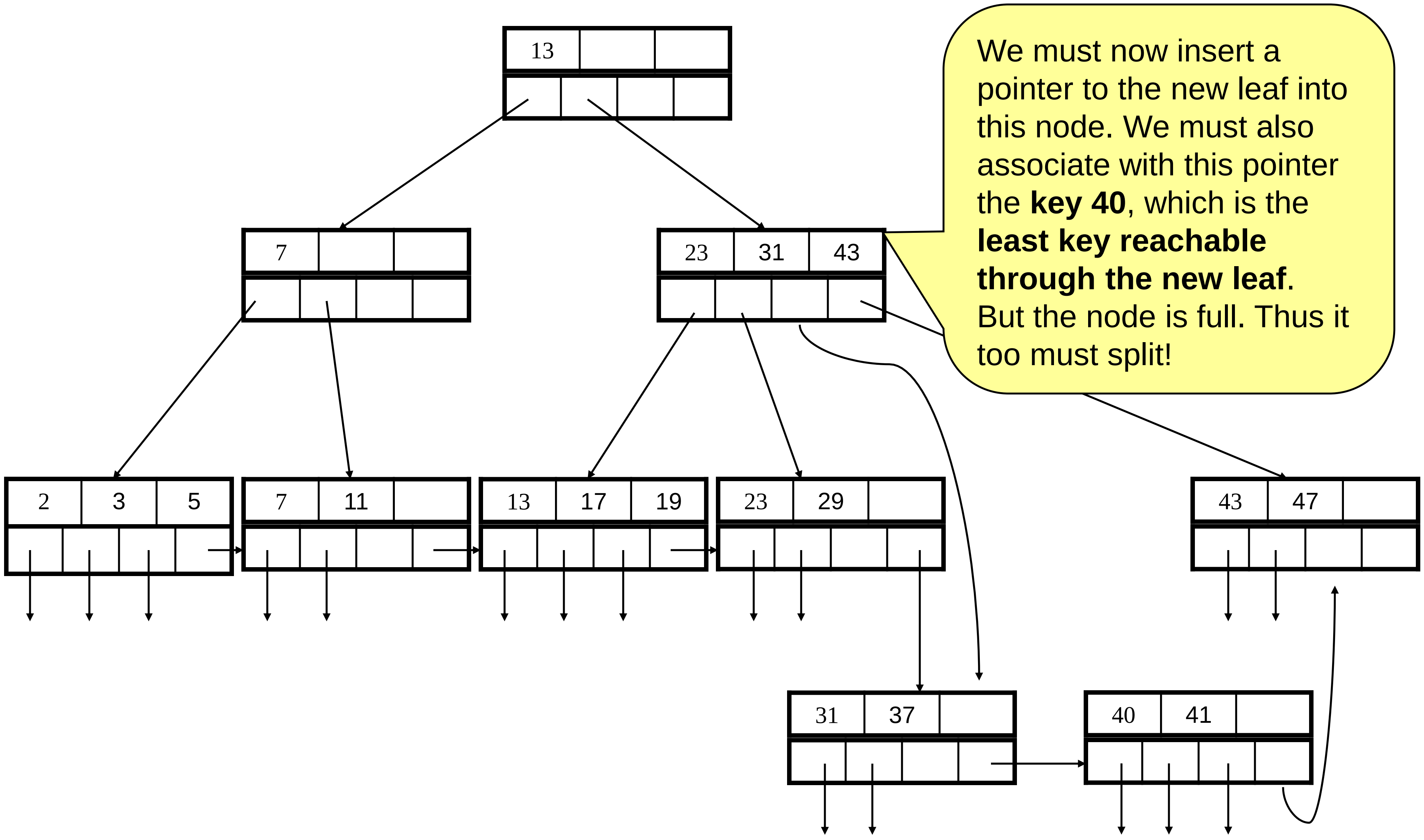
Beginning of the insertion of key 40



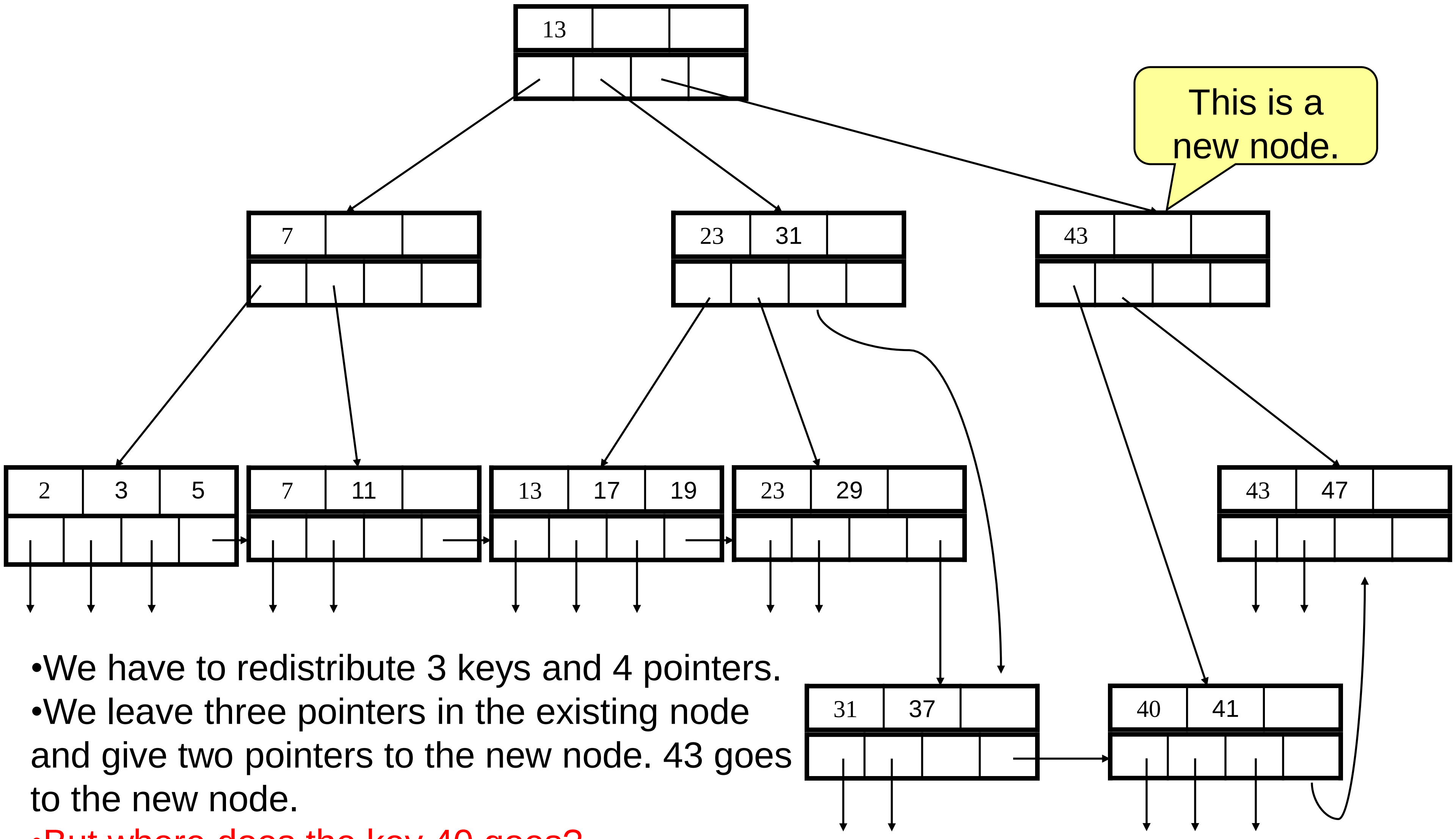
What's the problem?

No parent yet for the new node!

Continuing of the Insertion of key 40

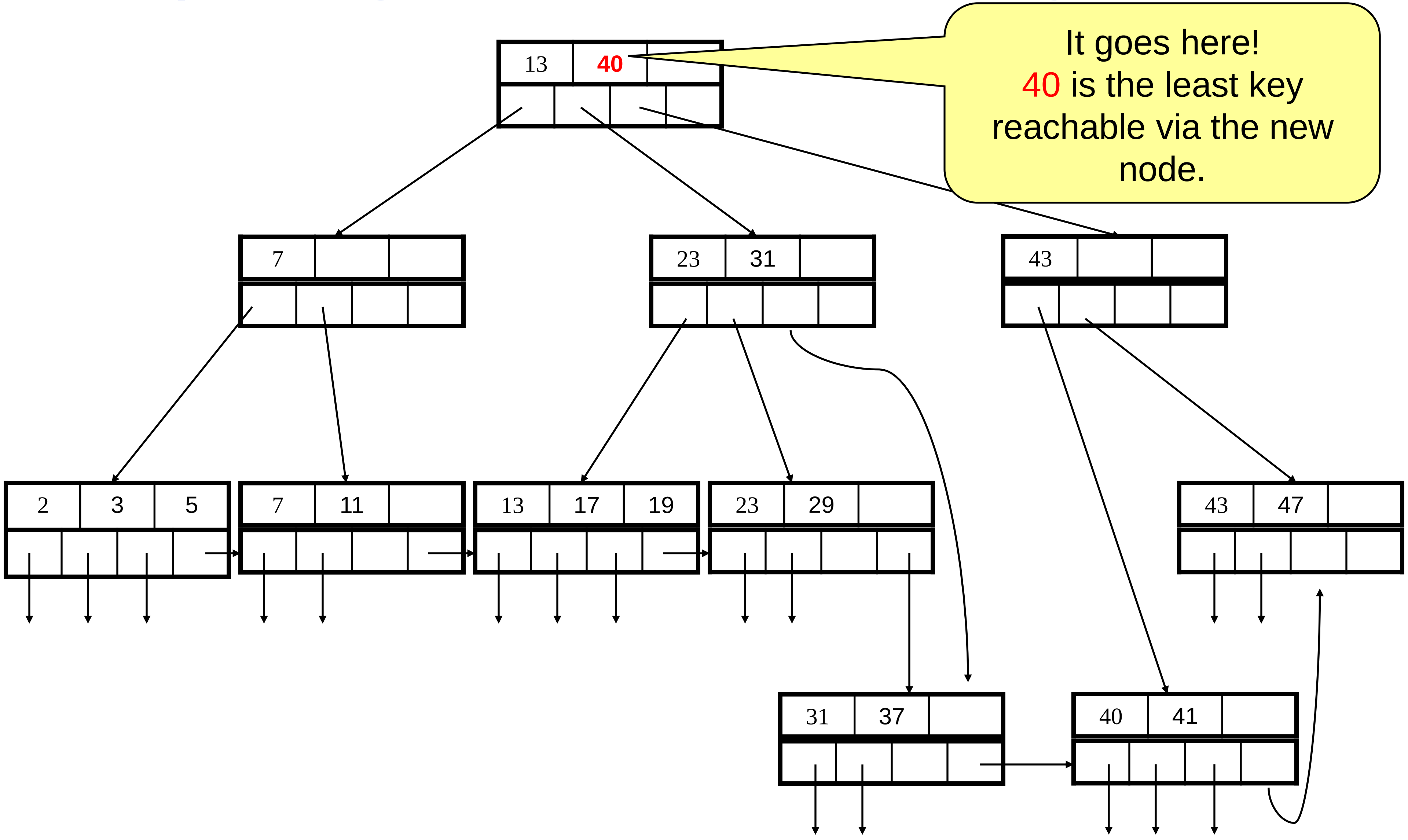


Completing of the Insertion of key 40



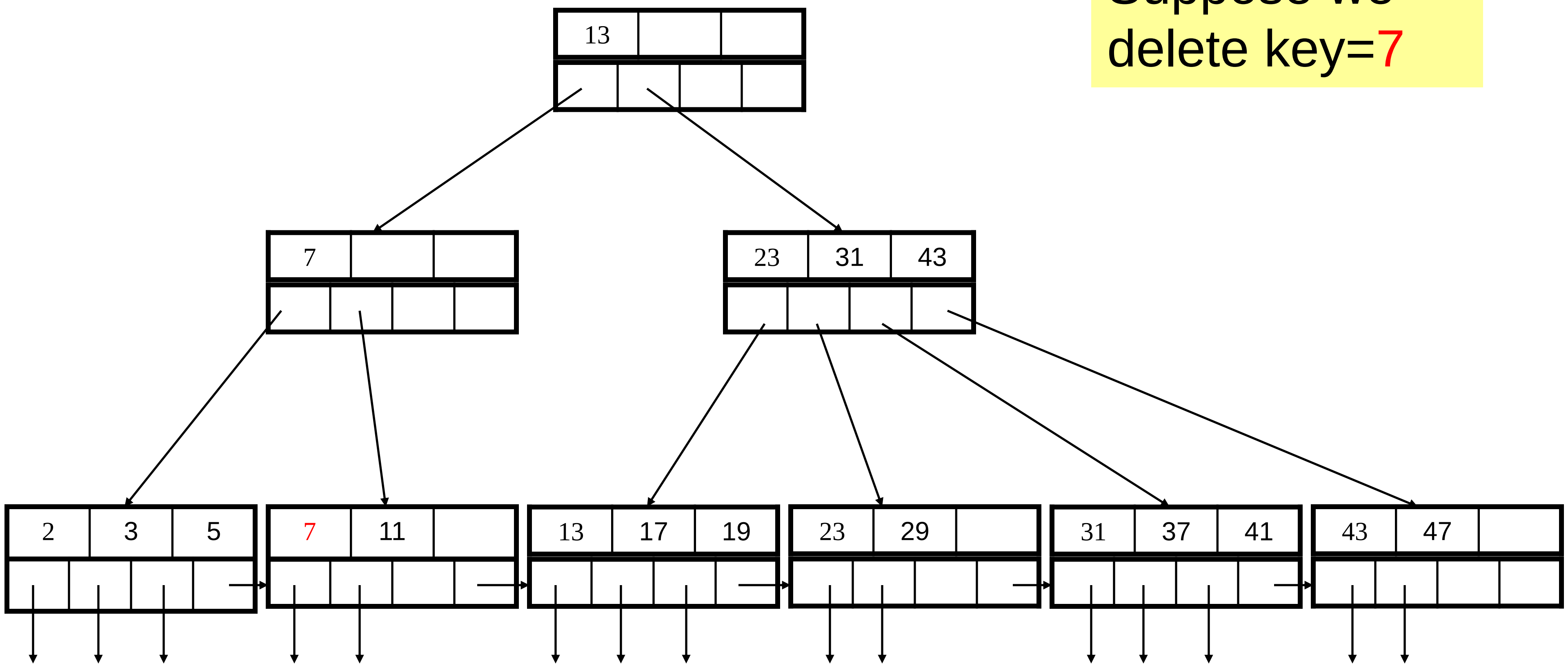
•We have to redistribute 3 keys and 4 pointers.
•We leave three pointers in the existing node and give two pointers to the new node. 43 goes to the new node.
•But where does the key 40 goes?
•40 is the least key reachable via the new node.

Completing the Insertion of key 40

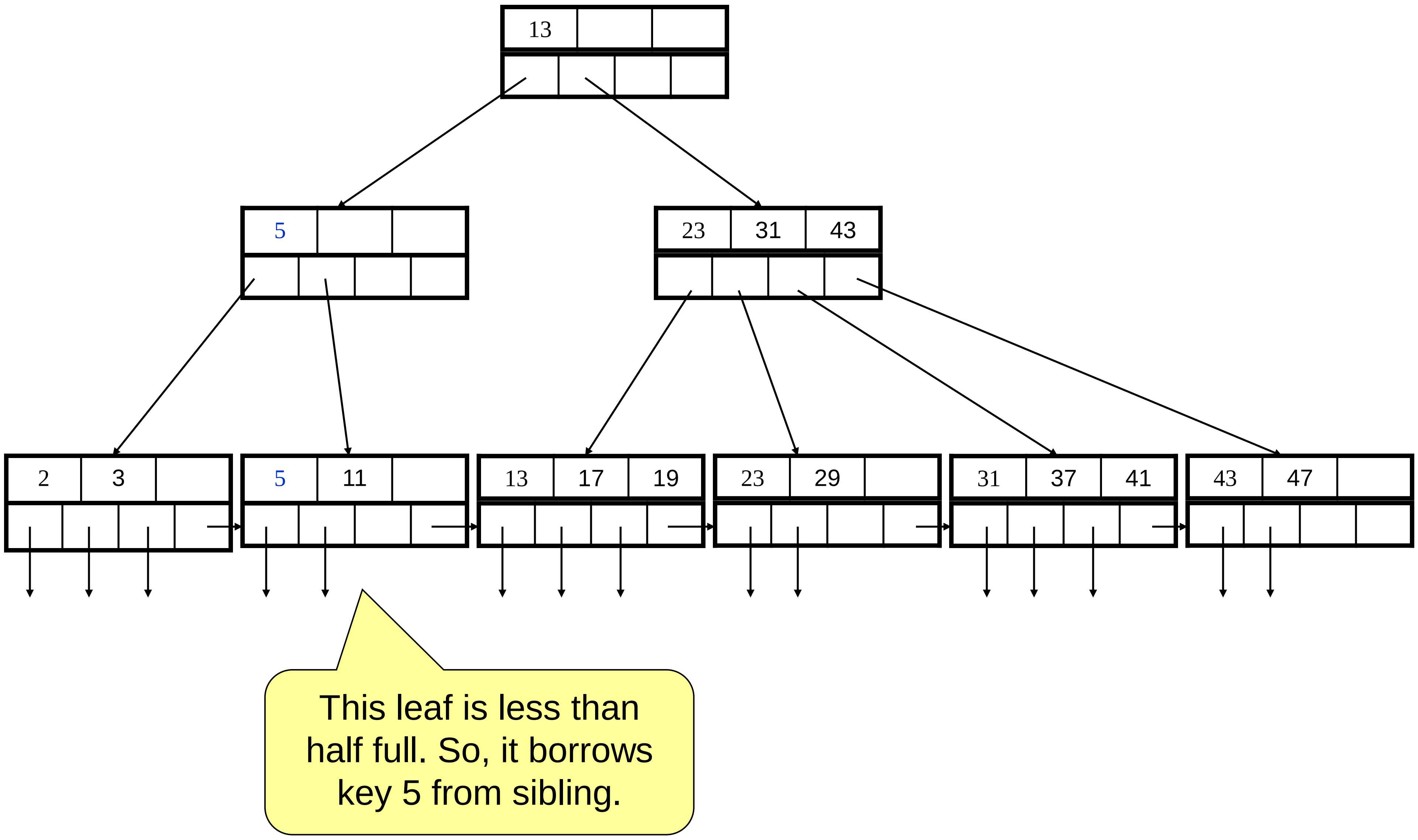


Deletion

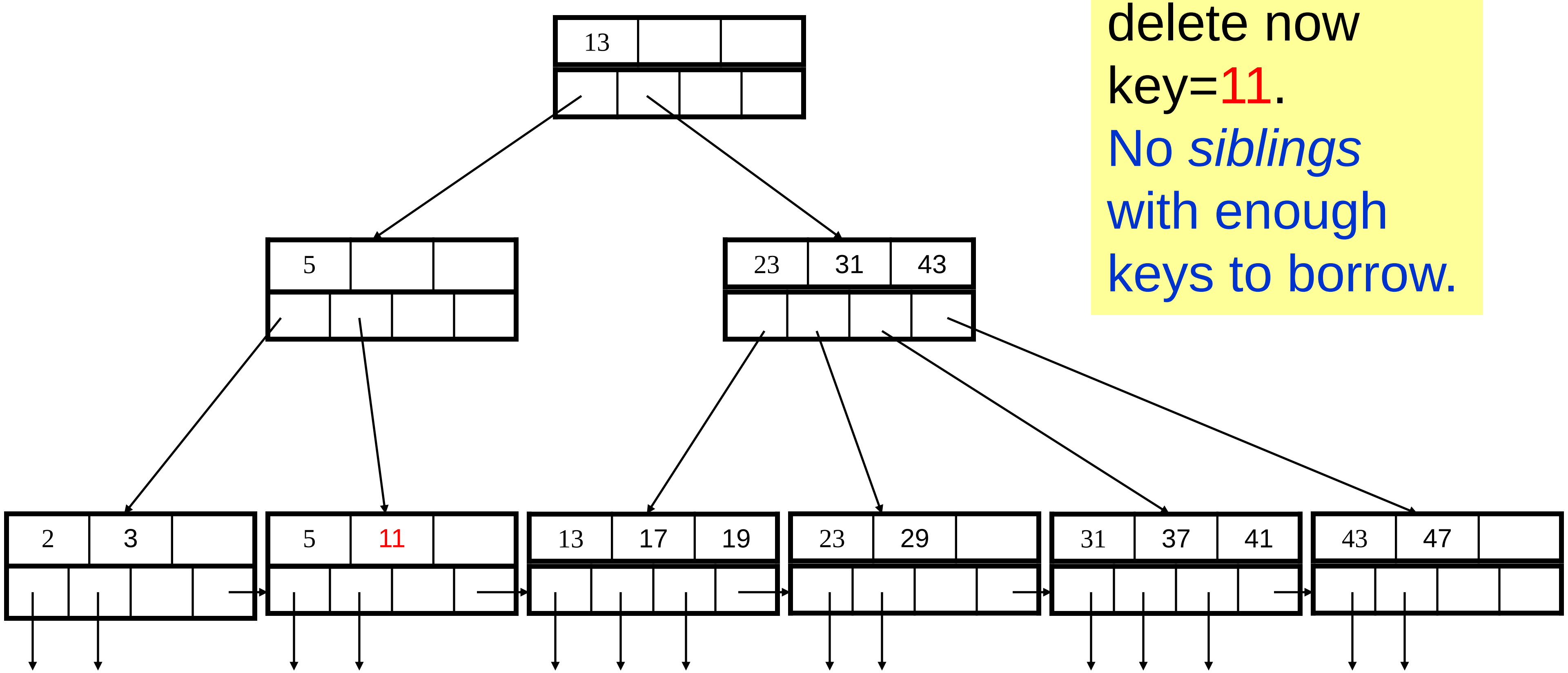
Suppose we delete key=7



Deletion (Raising a key to parent)

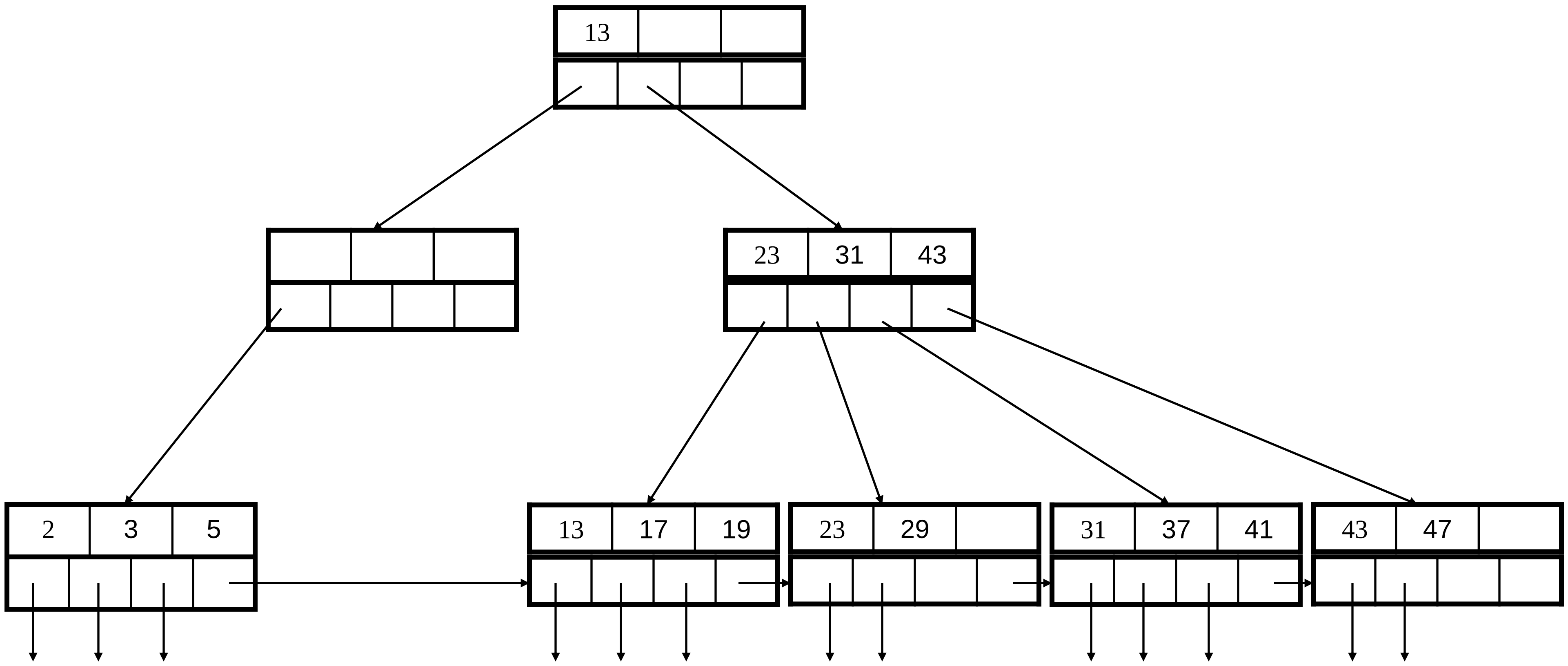


Deletion



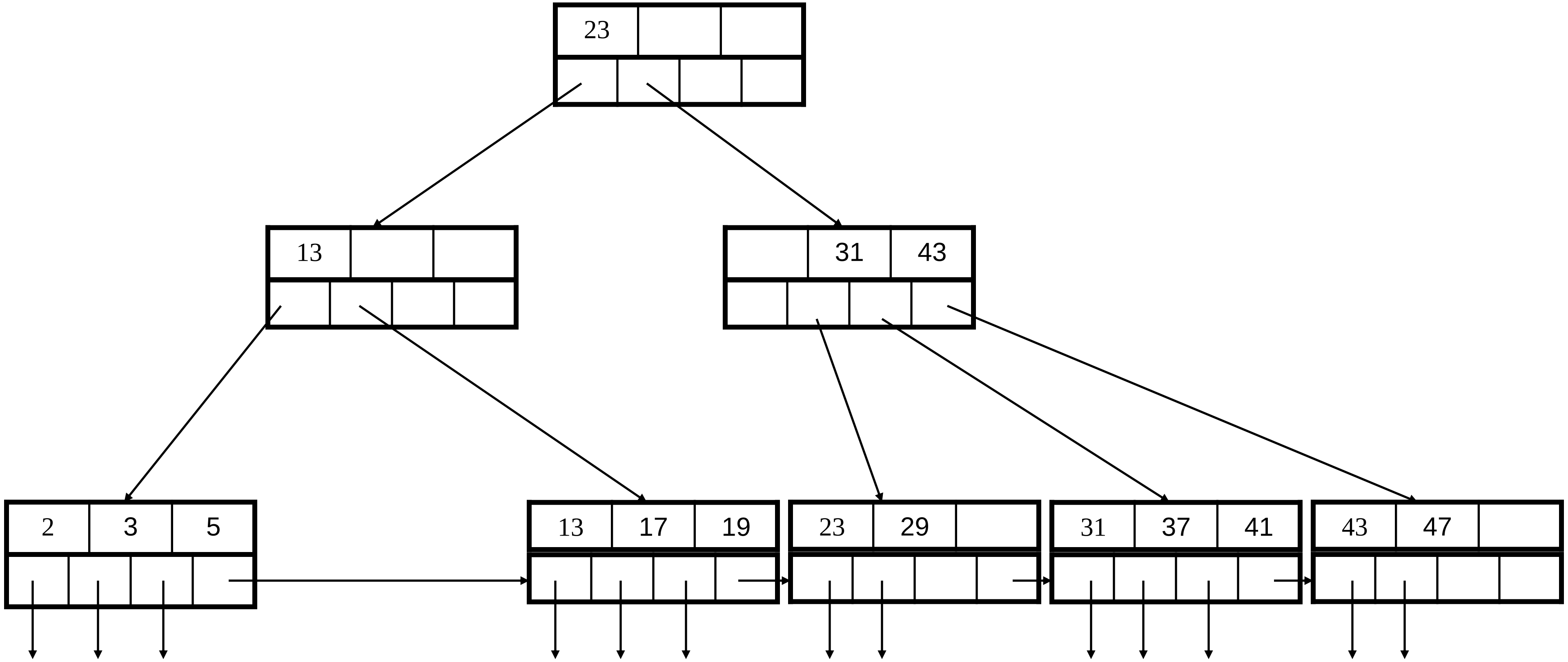
Suppose we delete now key=11.
No siblings with enough keys to borrow.

Deletion



We merge, i.e. delete a block from the index.
However, the parent ends up not having any key.

Deletion



Parent: Borrow pointer from sibling!

Structure of B-trees with real blocks

- Degree *n* means that all nodes have space for
  - *n* search keys and
  - *n*+1 pointers

- **Node = block**
- Let
  - block size be 16,384 Bytes,
  - key 20 Bytes,
  - pointer 20 Bytes.
- Let's solve for *n*:

$$20n + 20(n+1) \leq 16{,}384$$

$$\Rightarrow \; n \leq 409$$

Example

- n = 409, however a typical node has 300 keys
- At level 3 we have:

  300^2 nodes, which means

  $300^3 \approx 27{,}000{,}000$ records can be indexed.

- Suppose record = 1024 Bytes ➔ we can index a file of size

  $\approx$ 27 GB

- If the root is kept in main memory accessing a record requires only 3 disk I/O

Building a B-Tree from Existing Data

- Sort first, then build B-Tree.

Pointer Intersection

Example:

`Movies(title,year,length,studio);`

Assume indexes on `studio` and `year`.

```
SELECT title
FROM Movies
WHERE studio='Disney' AND year = 2021;
```

Use index on **studio** to obtain pointers to records with **Disney** as studio, but don't follow them.

Suppose we got 1000 such pointers.

Use index on **year** to obtain pointers to records with **2021** as year, but don't follow them.

Suppose we got 1000 such pointers.

Intersect the first set of pointers with the second.

Suppose we got 20 such pointers (typical number of movies Disney makes a year).

Follow these 20 pointers and retrieve the records.