

Databases

Indexes:

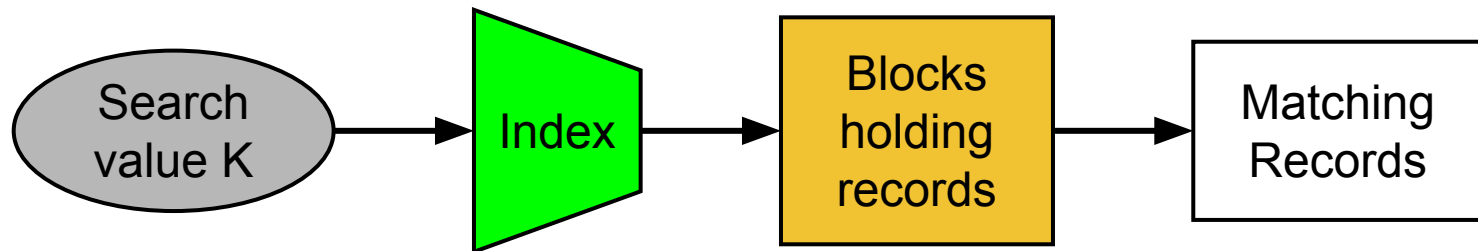
Introduction & B+ trees

Flavio Bertini

flavio.bertini@unipr.it

Indexes

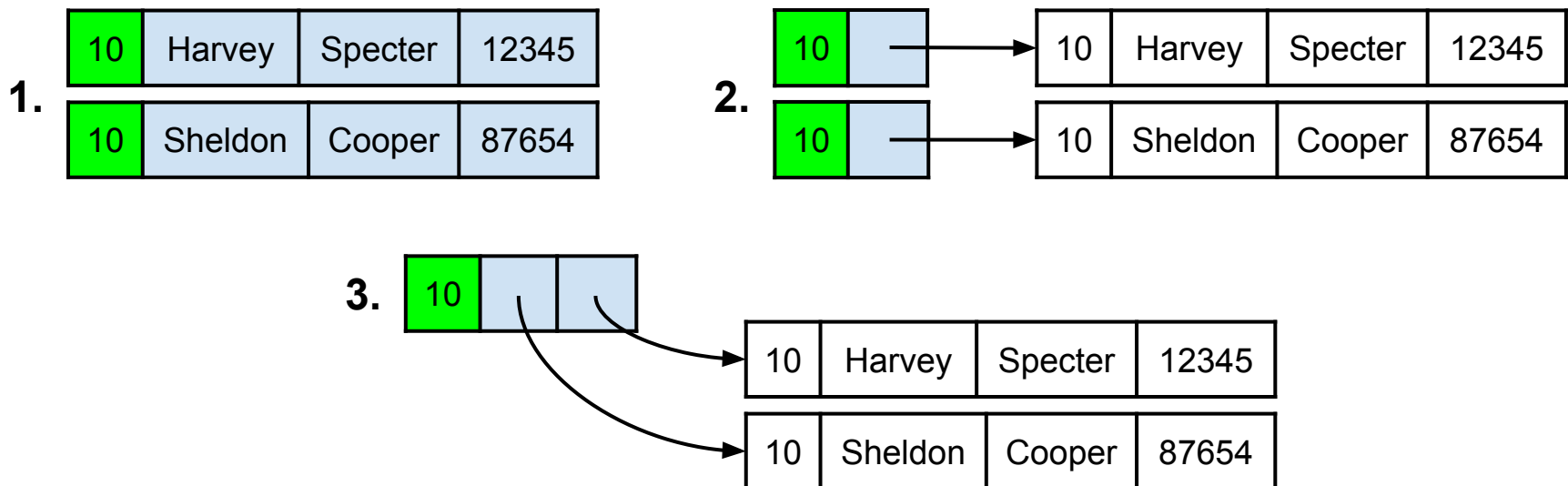
- An index is a data structure with **ancillary information** that supports **efficient access** to the data.
- The **search key** is defined using some attributes.
 - **Search keys are not primary keys**, that means that the same search key can store duplicate values.
 - In the absence of ambiguity, we will only use the term *key*.
- An index is a pair **<key, label>** and supports the efficient retrieve of all labels with a given key value K.



Labels

- Labels could be:

1. The **data records** themselves.
2. The **record identifier** (RID) of the record with key value K.
3. A **list of RID** of records with key value K.




- Labels representation is independent of the search method.

Labels Representation: Observations

- In a DB, you can have **at most one index** over the data **using the first representation**.
- Using the first representation, the **size of the index** is **the same** as the **size of the data** records.
- As you can see, the same search key (10) can store duplicate values.
- The **third representation** is the more **compact** solution, but the labels have **variable sizes**.

Guess the true one ...

Given a single file:

-  **A.** At most one “data record” index could be used at a time.
- B.** Only one “record identifier” index could be used at a time.
- C.** “Record identifier” index is more compact than “record identifier list” index.
- D.** The “data record” index is more useful when the data size is bigger.



Create an Index in SQL

```
CREATE [UNIQUE] INDEX IndexName ON Table(AttributesList)
```

```
CREATE INDEX EmpIdx ON Employee(Surname, Name)
```

```
DROP INDEX IndexName
```

```
DROP INDEX EmpIdx
```

Please note that CREATE INDEX is not SQL standard, but it is the most common syntax.

Sequential File Index - an example

Index File

10	—
20	—
30	—
40	—

50	—
60	—
70	—
80	—

90	—
100	—
	—
	—

Sequential File

10	
20	

30	
40	

50	
60	

70	
80	

90	
100	

Each search key of the Index File has a pointer to the record of the Sequential File, and the search key is the primary key of the relation.

Indexes Classification

- A **primary** index is an index on a set of attributes that includes the primary key, and the data is sorted on the same attributes. Otherwise, the index is **secondary**.
- A **dense** index has at least one search key value for each key value from the data file. Otherwise, the index is **sparse**.
- An index is **clustered** if the record order reflects (is the same, or similar) to the label order. Otherwise, the index is **unclustered**.

A Dense and Clustered Index

Index File

10	
20	
30	
40	

50	
60	
70	
80	

90	
100	

Data File

10	
20	

30	
40	

50	
60	

70	
80	

90	
100	

A Sparse and Clustered Index

Index File

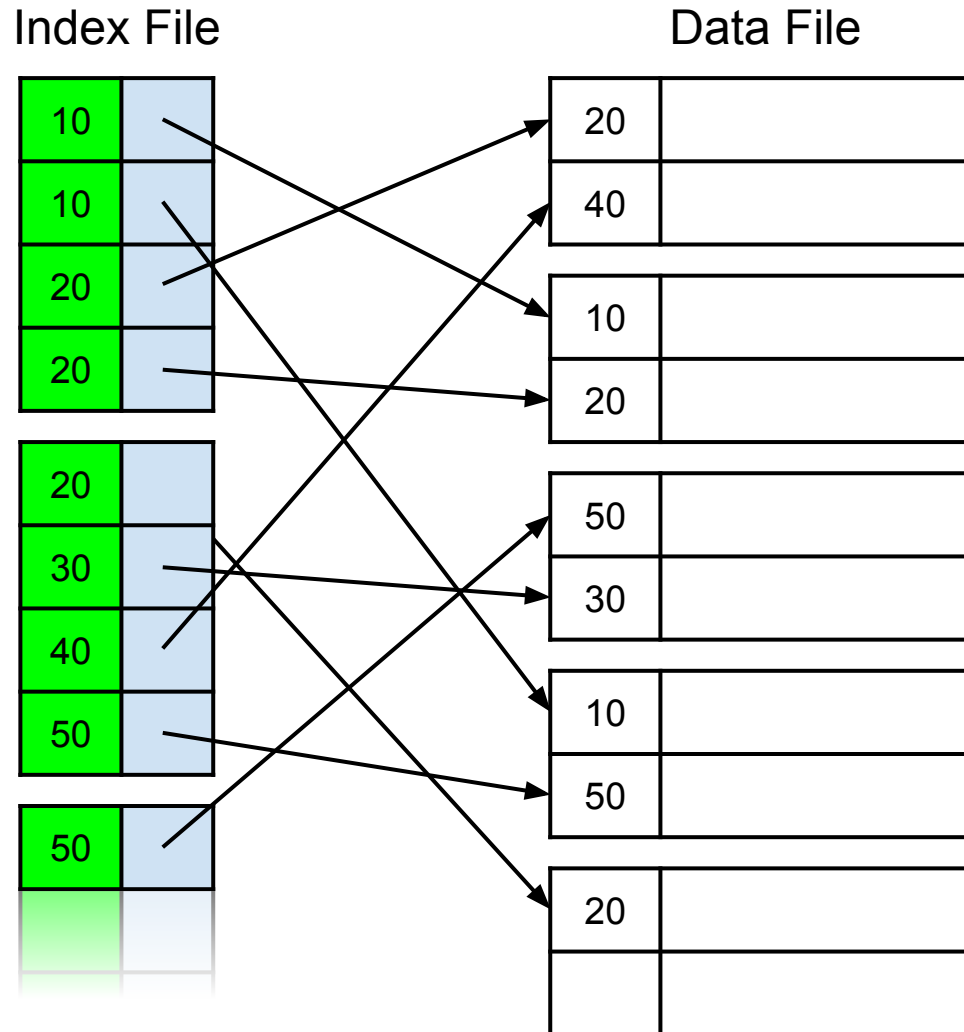
10	
30	
50	
70	
90	

Data File

10	
20	
30	
40	
50	
60	
70	
80	
90	
100	

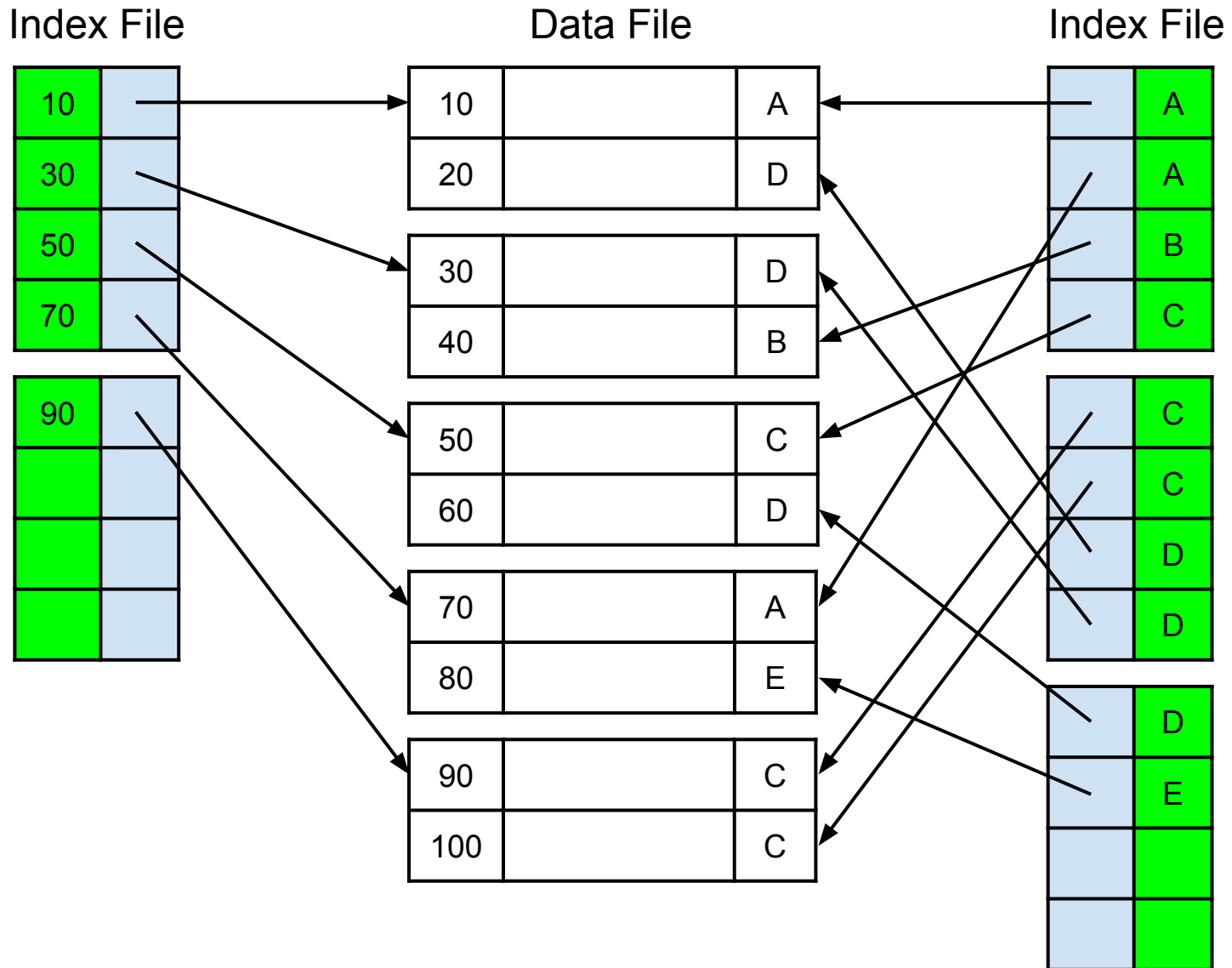
The space required to store a sparse index is less than for dense index.

A Secondary, Dense and Unclustered Index




Unclustered indexes provide less efficient data accesses: three records with the same value (i.e., 20) are stored in three different data blocks.

Clustered vs Unclustered Indexes



Guess the false one ...

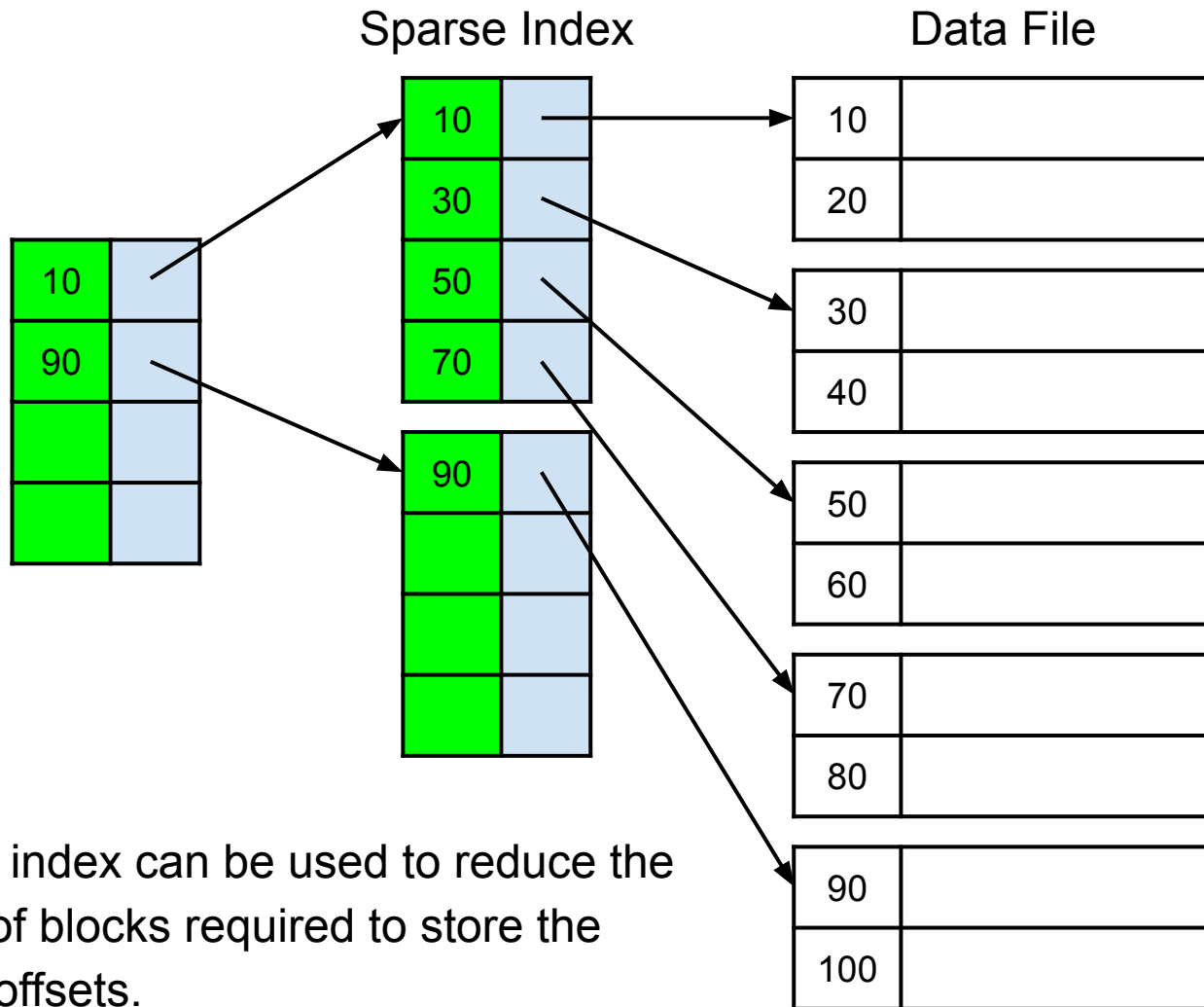
Given a single file:

- A.** A dense index could be secondary.
- B.** A sparse index could be clustered.
-  **C.** A sparse index could be unclustered.
- D.** A secondary index could be clustered.

Search with Dense Index - an example

- Imagine a relation of **1,000,000 tuples** stored in **100,000 blocks** of **4096 bytes**. The total space required by the data is over **400 megabytes**, probably too much to keep and navigate efficiently in main memory.
- However, suppose that the **key field is 30 bytes**, and **pointers are 8 bytes**. Then with a reasonable amount of block-header space we can keep **100 key-pointer pairs** in a **single 4096 bytes block**.
- A **dense index** therefore requires **10,000 blocks**, or **40 megabytes**. We might be able to allocate main memory buffers for these blocks.
- Further, we can use **dense index and binary search**. Thus, we only **need to access 13 or 14 blocks** ($\approx \log_2 10,000$) in a binary search for a key.
- Moreover, we might be able to keep the most important blocks in main memory, thus retrieving the record for any key with significantly **fewer than 14 disk I/O operations**.

Multiple Levels of Indexing



- A sparse index can be used to reduce the number of blocks required to store the records' offsets.
- We can use **index of indexes**.

B+ trees


- Multiple levels of indexing are very helpful in speeding up queries, allowing to perform searches involving constant access for each level plus the time required to scan the RID list.
- There is a **dynamic structure** that is commonly used in commercial systems, namely **B-tree**. It automatically maintains as many **levels of index** as is appropriate for the **size of the file** being indexed.
- B-trees manage the space on the blocks they use so that every block is **between half used and completely full** (except for the root).
- There is a **parameter n** associated with each B-tree index. Each block will have space for **n search key** values and **n + 1 pointers**.
- B+ trees are a specific type of B-trees.

Find the Value of n ...

Let's assume that:

- Block size: 4096 B.
- Key size: 4 B.
- Pointer size: 8 B.
- There is no header information kept on the blocks.

The value of n is:

-  A. 340
- B. 4096
- C. 48
- D. 8

Hint: we want to find the largest integer value of n such that

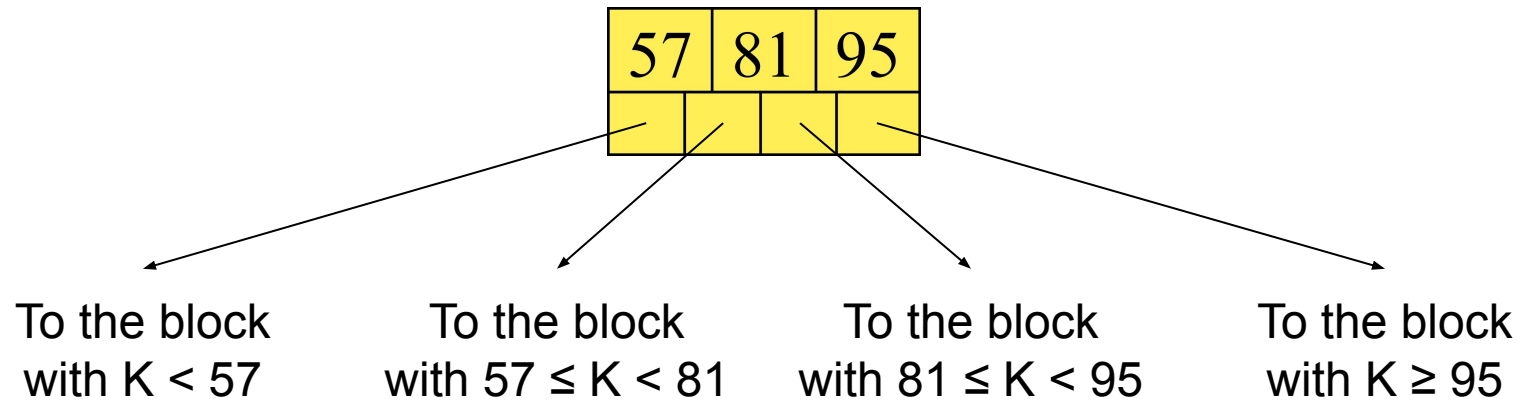
$$4n + 8(n + 1) < 4096$$

B+ trees Rules

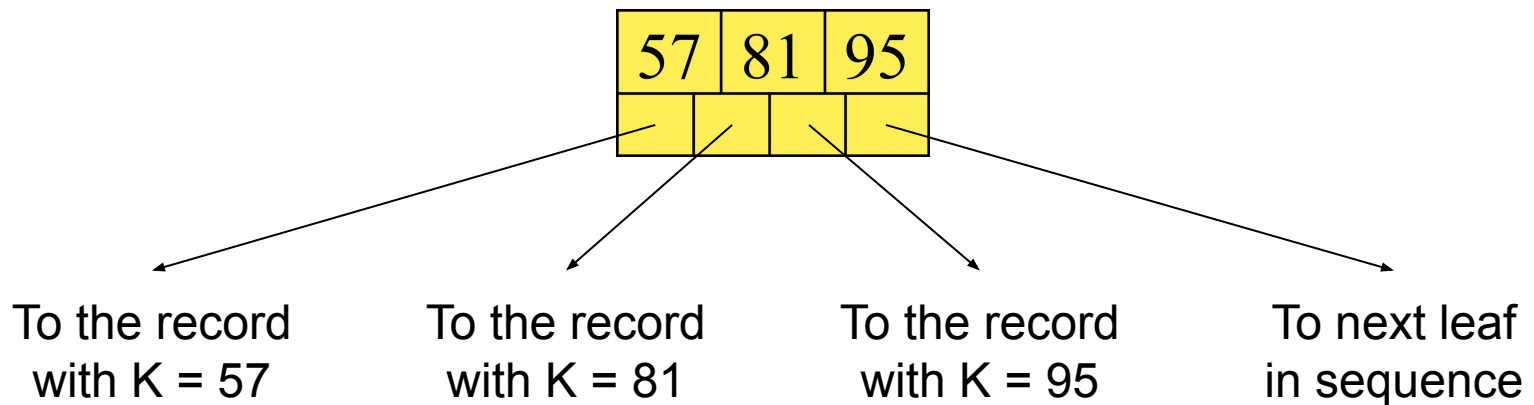
- The **keys in leaf** nodes are **copies of keys from the data file**. These keys are distributed among the leaves in sorted order, from left to right.
- At **the root**, there are **at least two used pointers** (with at least two records data in the file). All pointers point to blocks at the level below.
- At an **interior node**, all used pointers point to blocks at the next lower level, and at least them $\lceil (n+1)/2 \rceil$ must to be used.
- At a **leaf**, the last pointer points to the next leaf block to the right. Among the other pointers in a leaf block, at least $\lfloor (n+1)/2 \rfloor$ of them are used and point to data records.

B+ trees: Interior vs Leaf Nodes

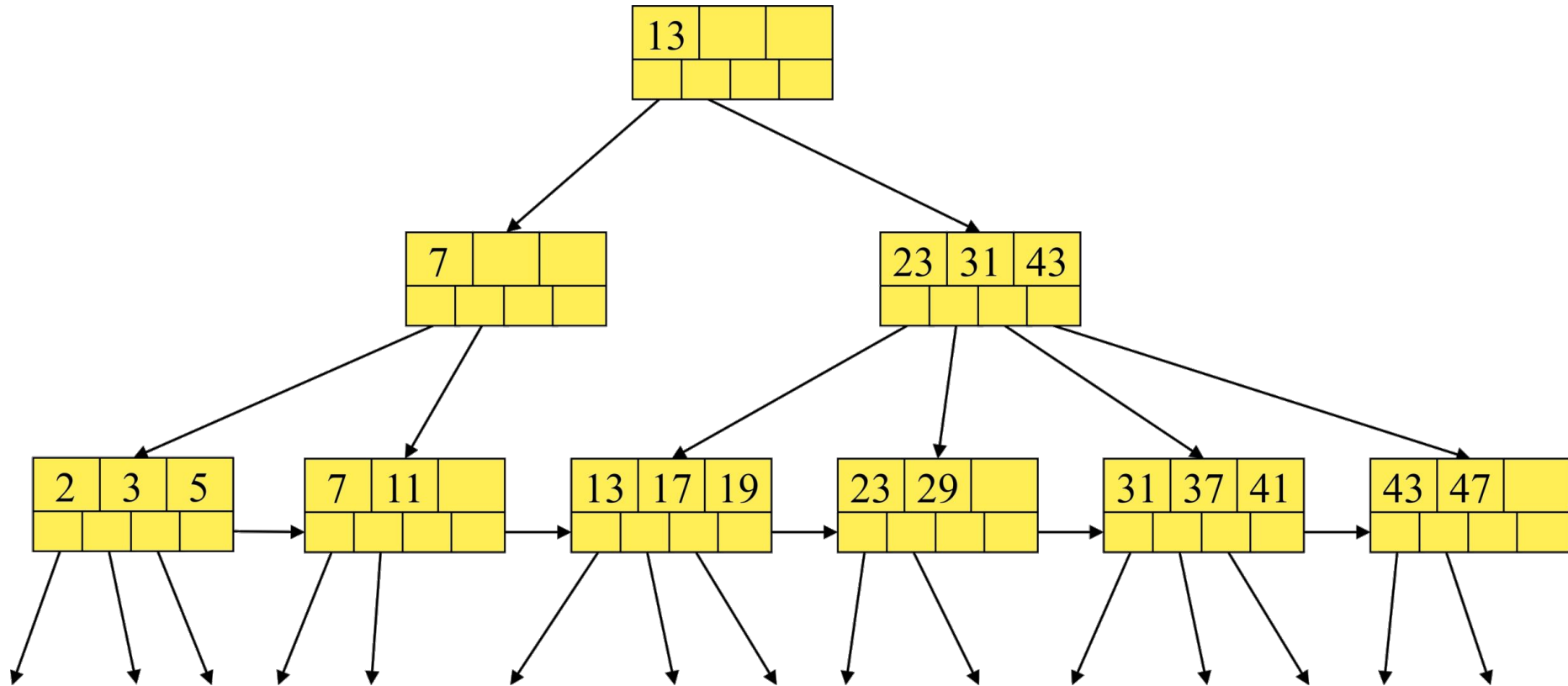
Interior node



Leaf node



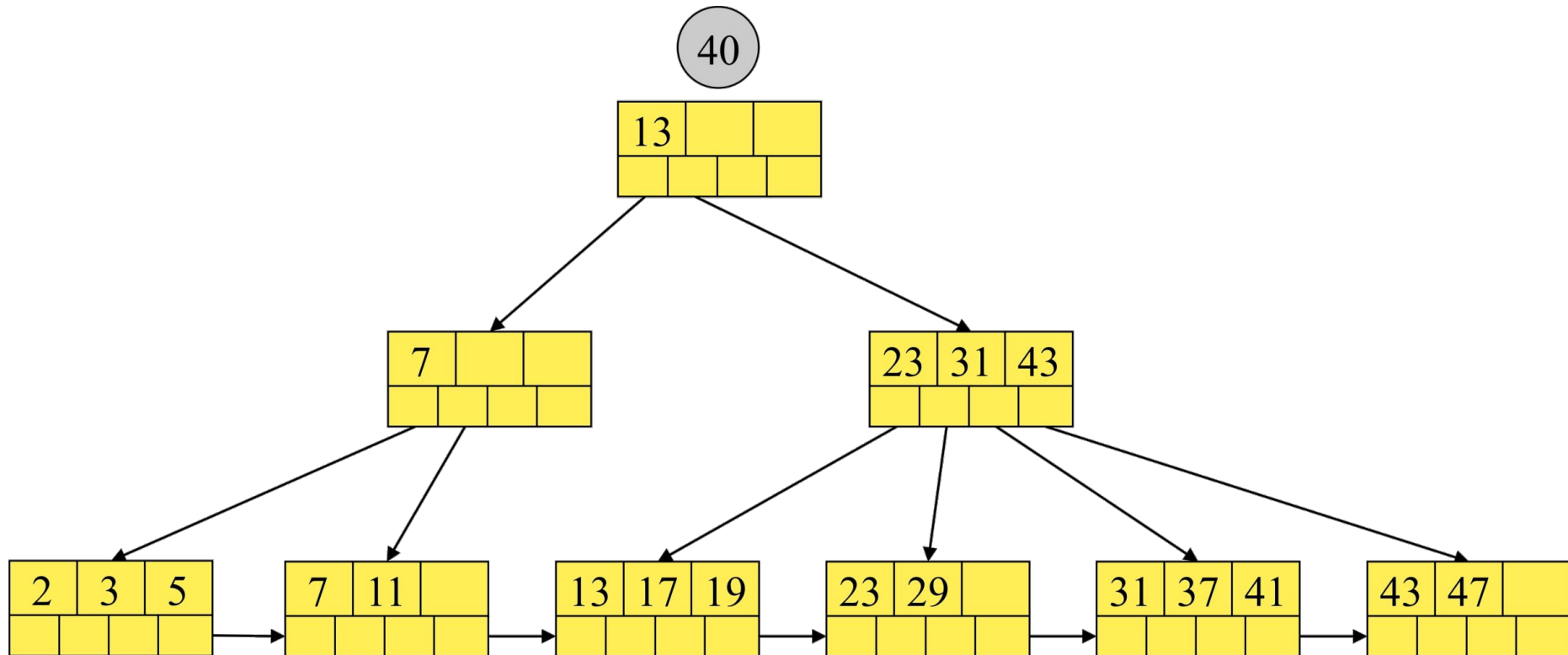
B+ trees - an example



Data Records

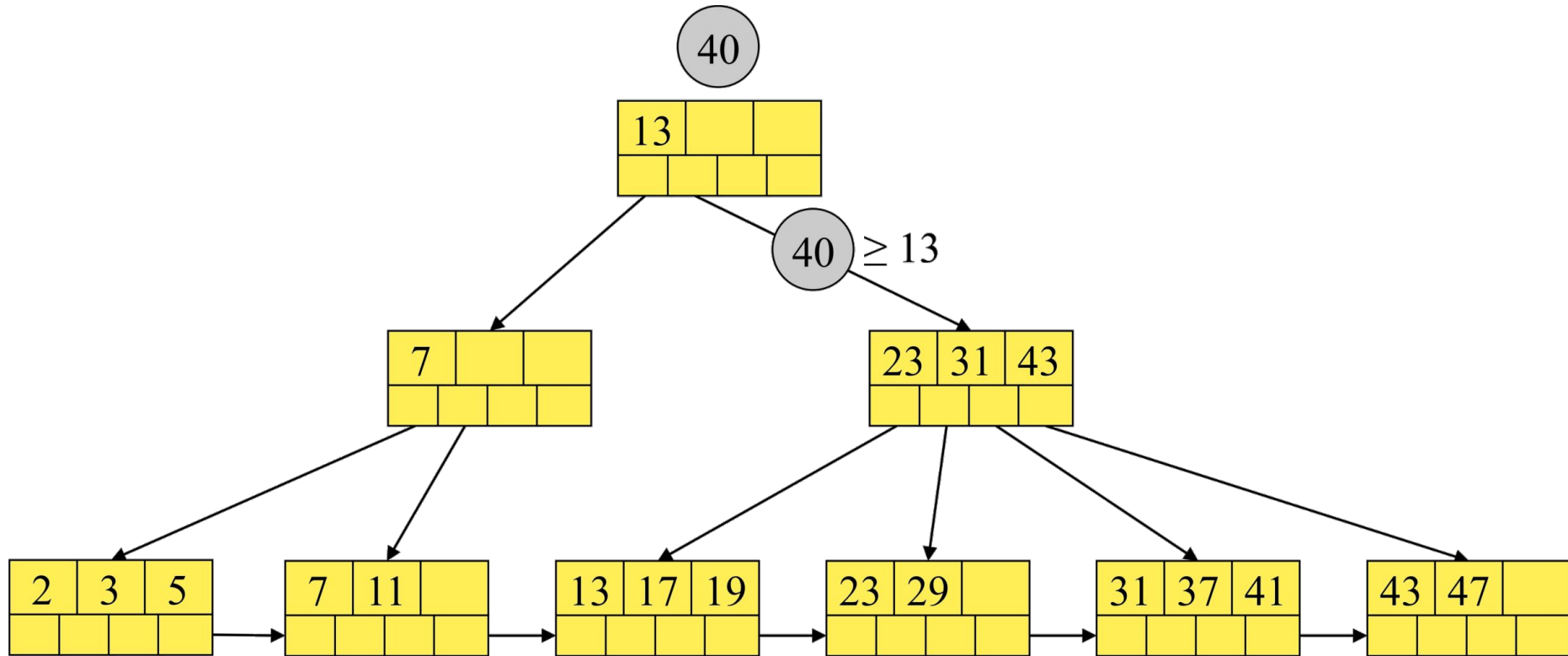
B+ trees and Equality Search (1)

We are looking for the data record with search key $K=40$.



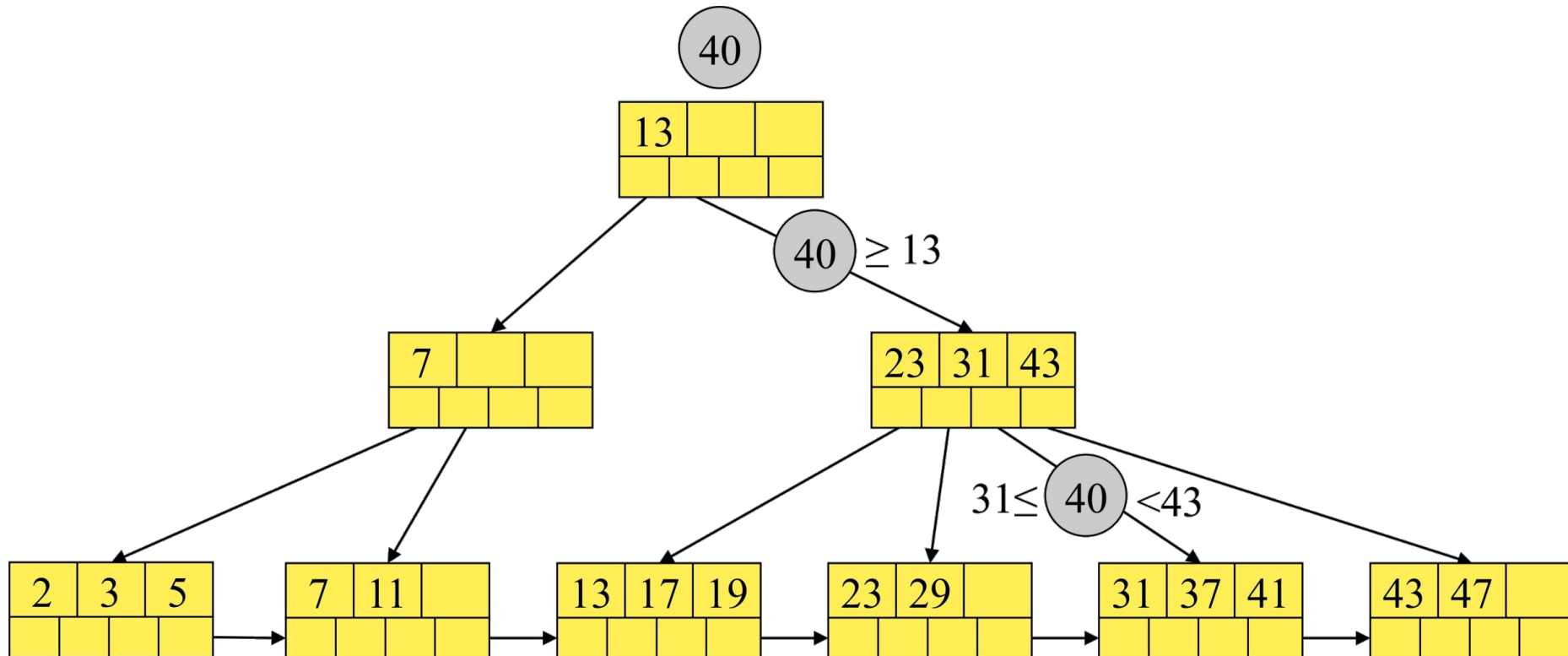
B+ trees and Equality Search (2)

We are looking for the data record with search key $K=40$.



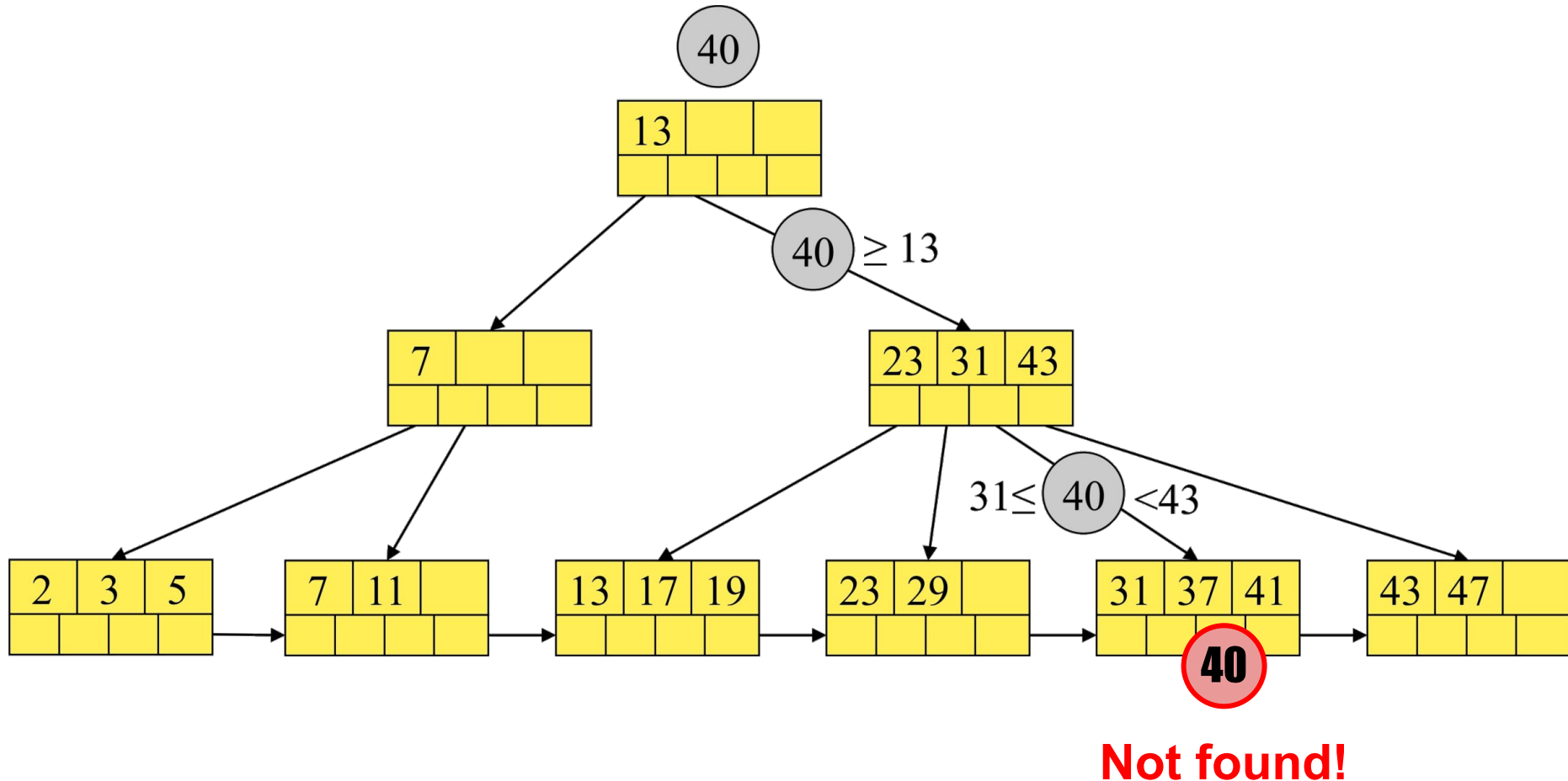
B+ trees and Equality Search (3)

We are looking for the data record with search key $K=40$.



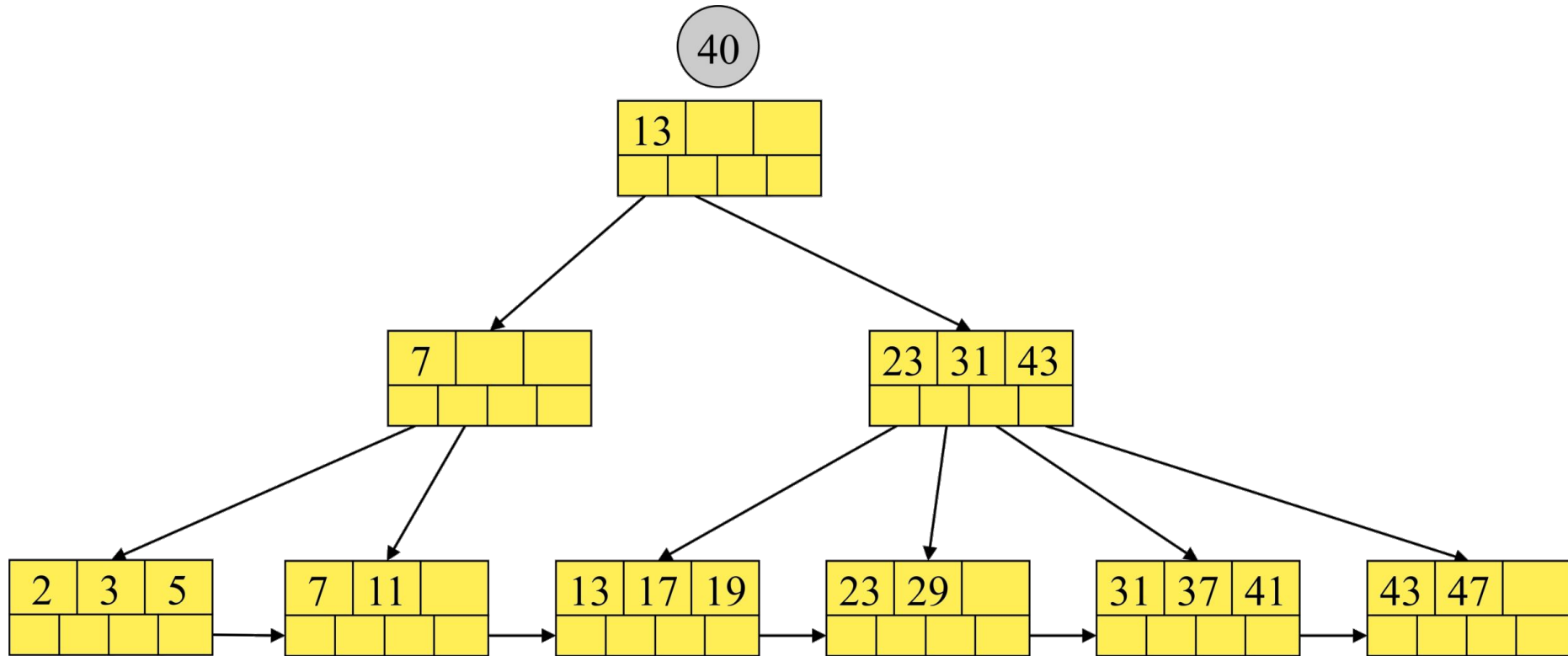
B+ trees and Equality Search (4)

We are looking for the data record with search key $K=40$.



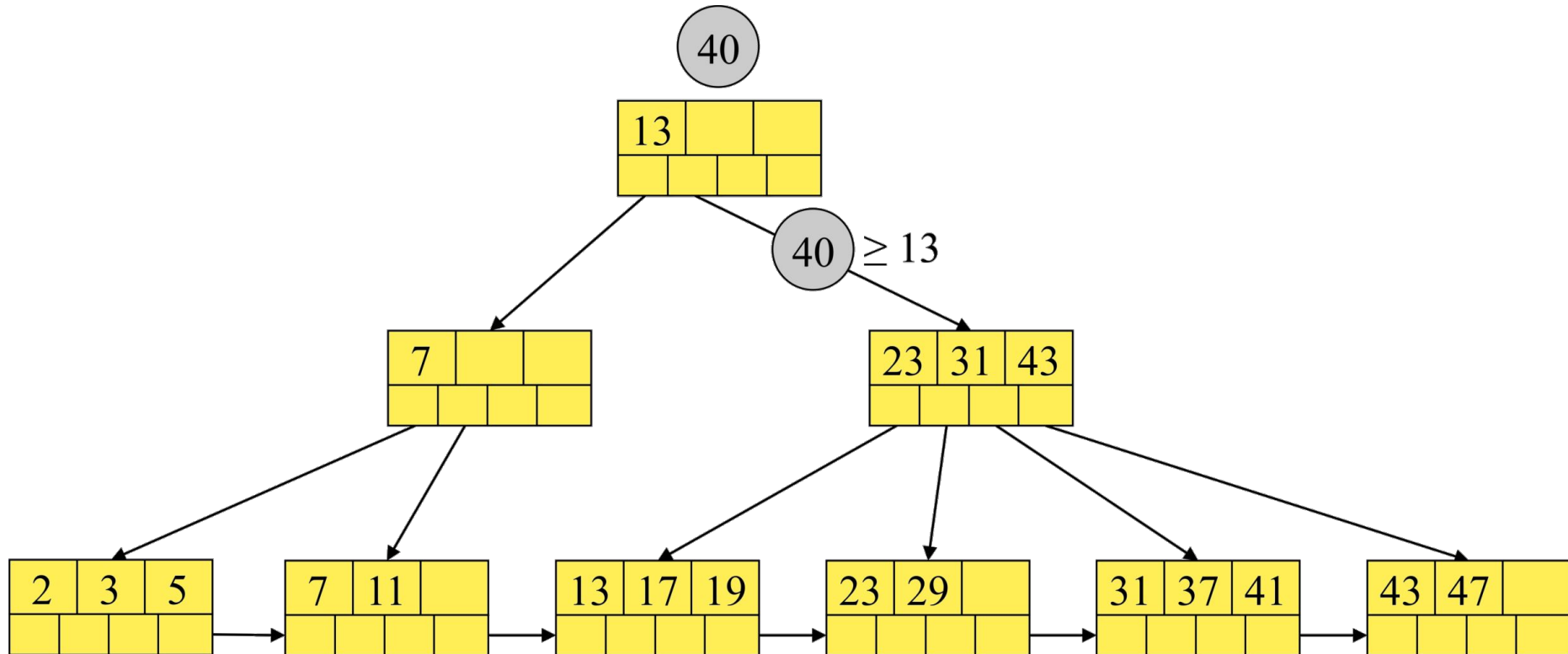
B+ trees and Range Search (1)

We are looking for the data records with search key $K > 40$.



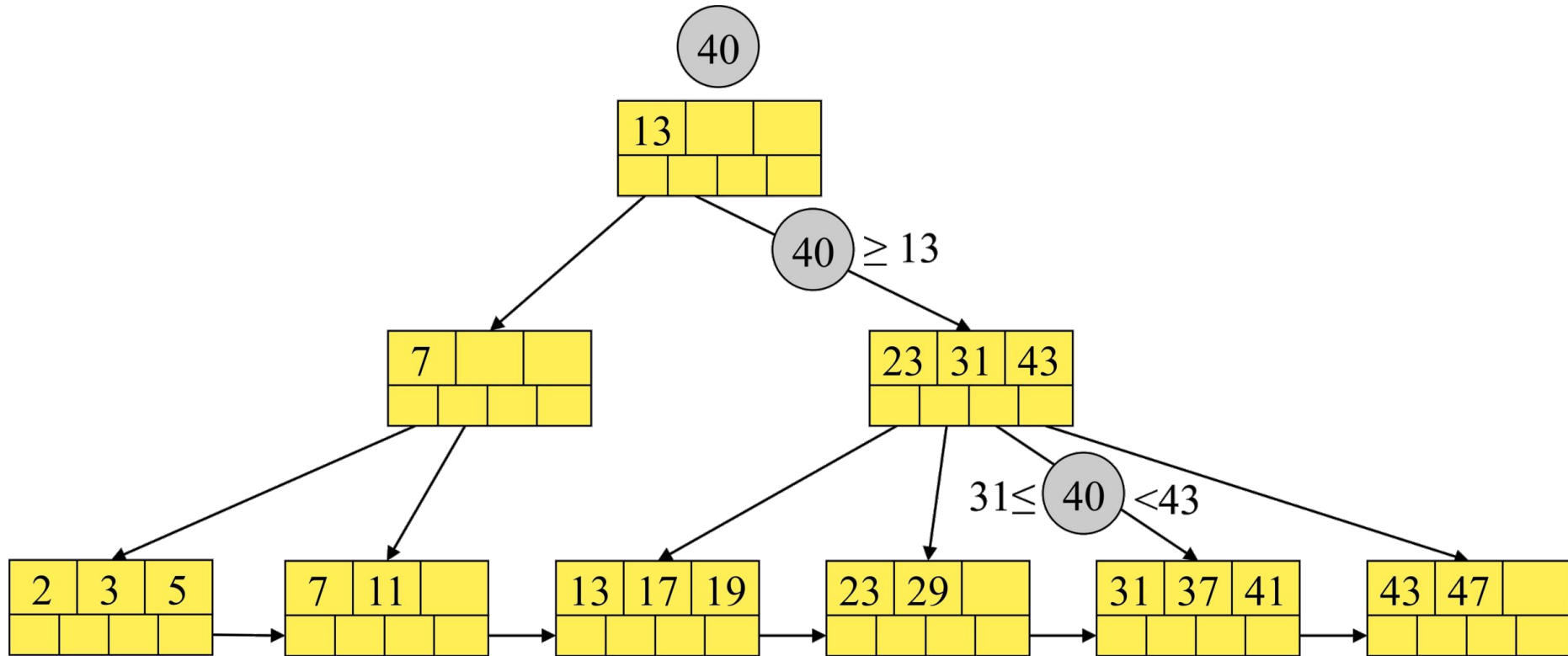
B+ trees and Range Search (2)

We are looking for the data records with search key $K > 40$.



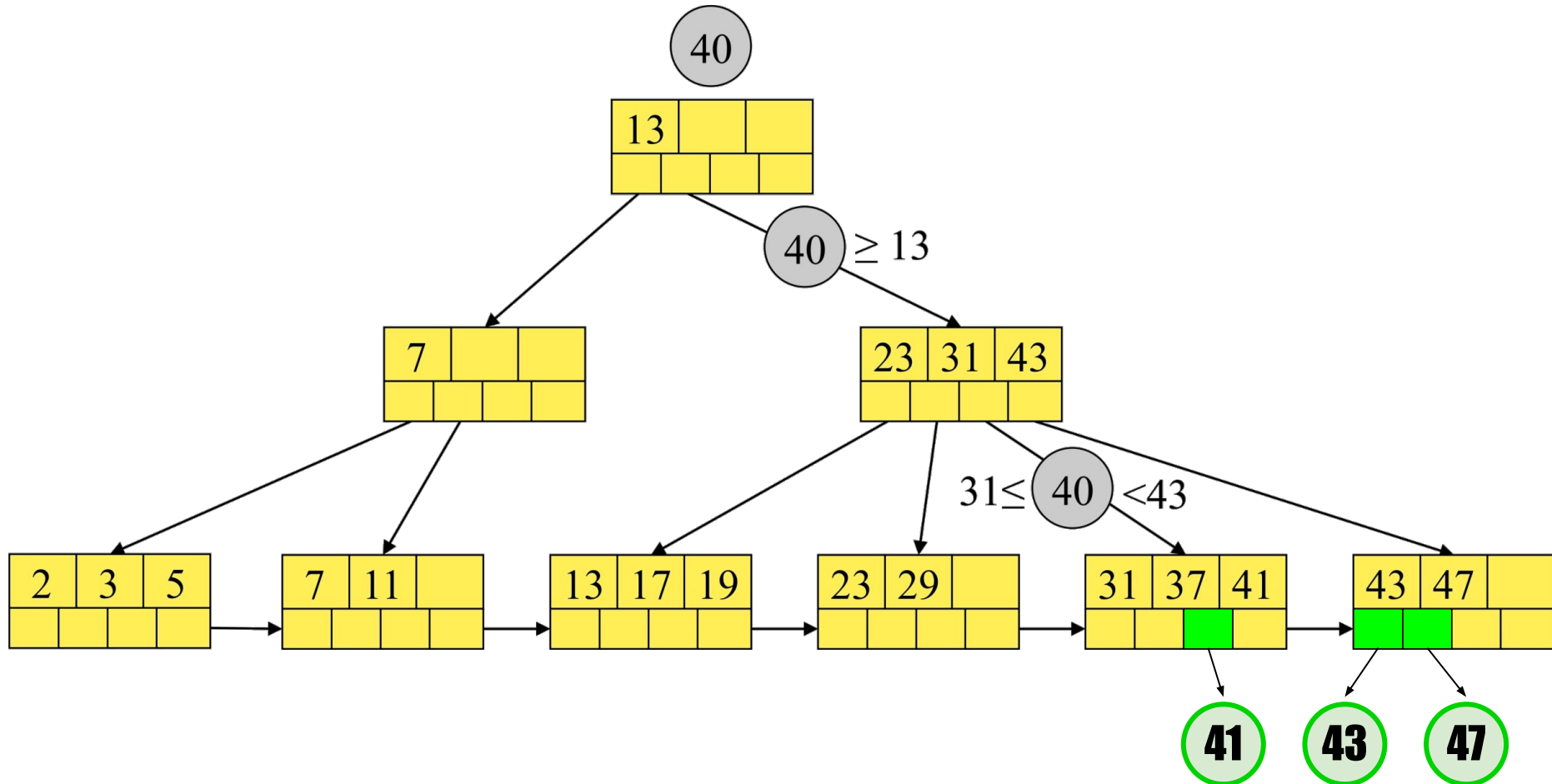
B+ trees and Range Search (3)

We are looking for the data records with search key $K > 40$.

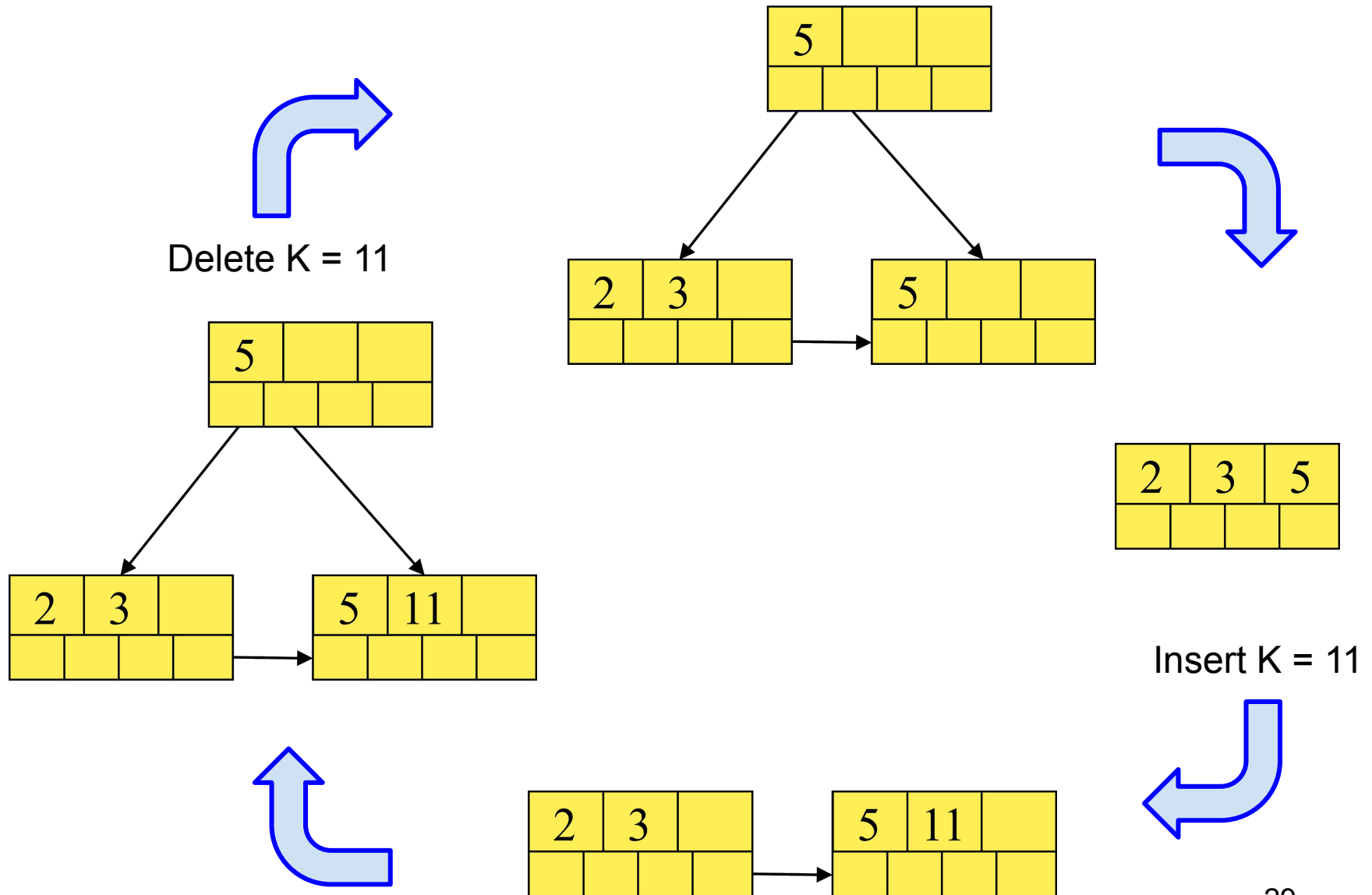


B+ trees and Range Search (4)

We are looking for the data records with search key $K > 40$.



Insertion and Deletion: Height Variation



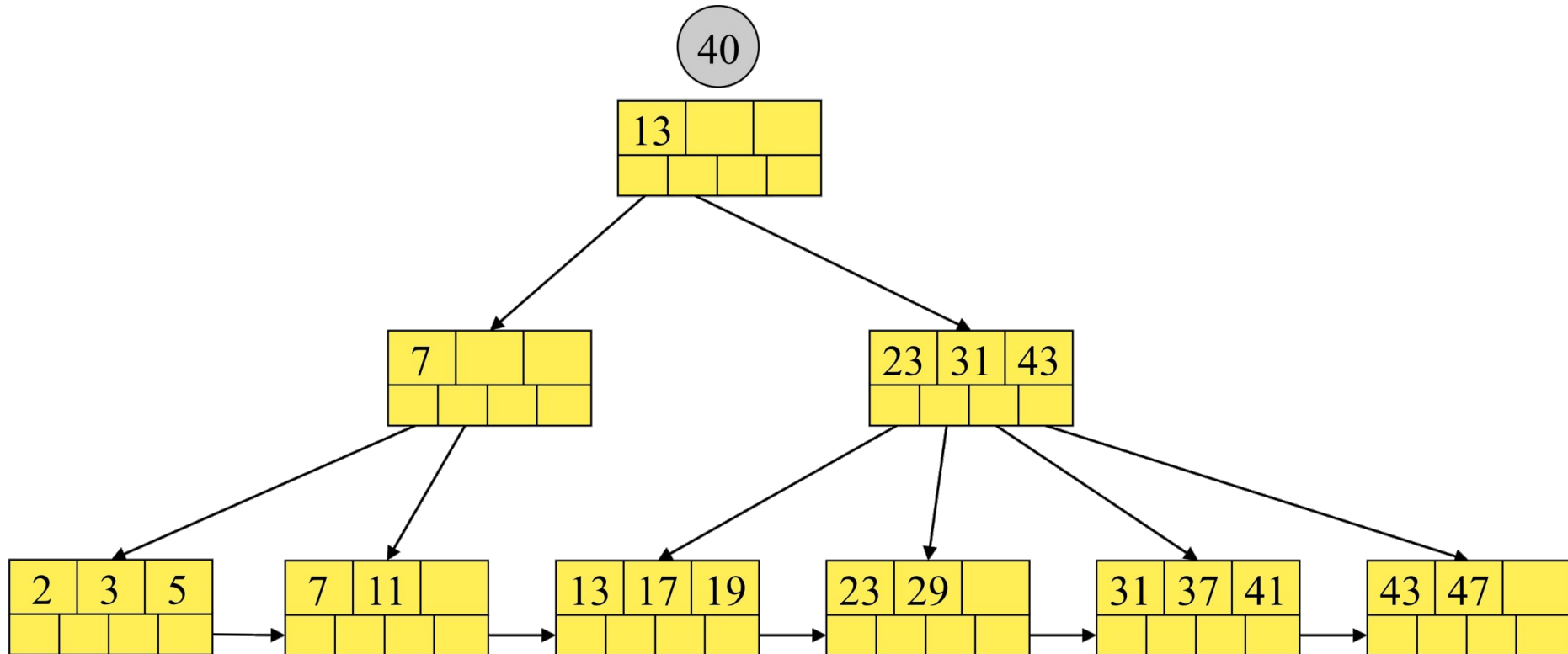
Insertion Into B+ trees

The insertion is, in principle, recursive:

- We try to find a place for the new key in the appropriate leaf, and we put it there if there is room.
- If there is no room in the proper leaf, we split the leaf into two and divide the keys between the two new nodes, so each is half full or just over half full.
- The splitting of nodes at one level appears to the level above as if a new key-pointer pair needs to be inserted at that higher level. We may thus recursively apply this strategy to insert at the next level: if there is room, insert it; if not, split the parent node and continue up the tree.
- As an exception, if we try to insert into the root, and there is no room, then we split the root into two nodes and create a new root at the next higher level.
- **PLEASE NOTE** - When you split an inner node, the central key is not inserted into one of the new nodes, but only propagated upwards to the root.

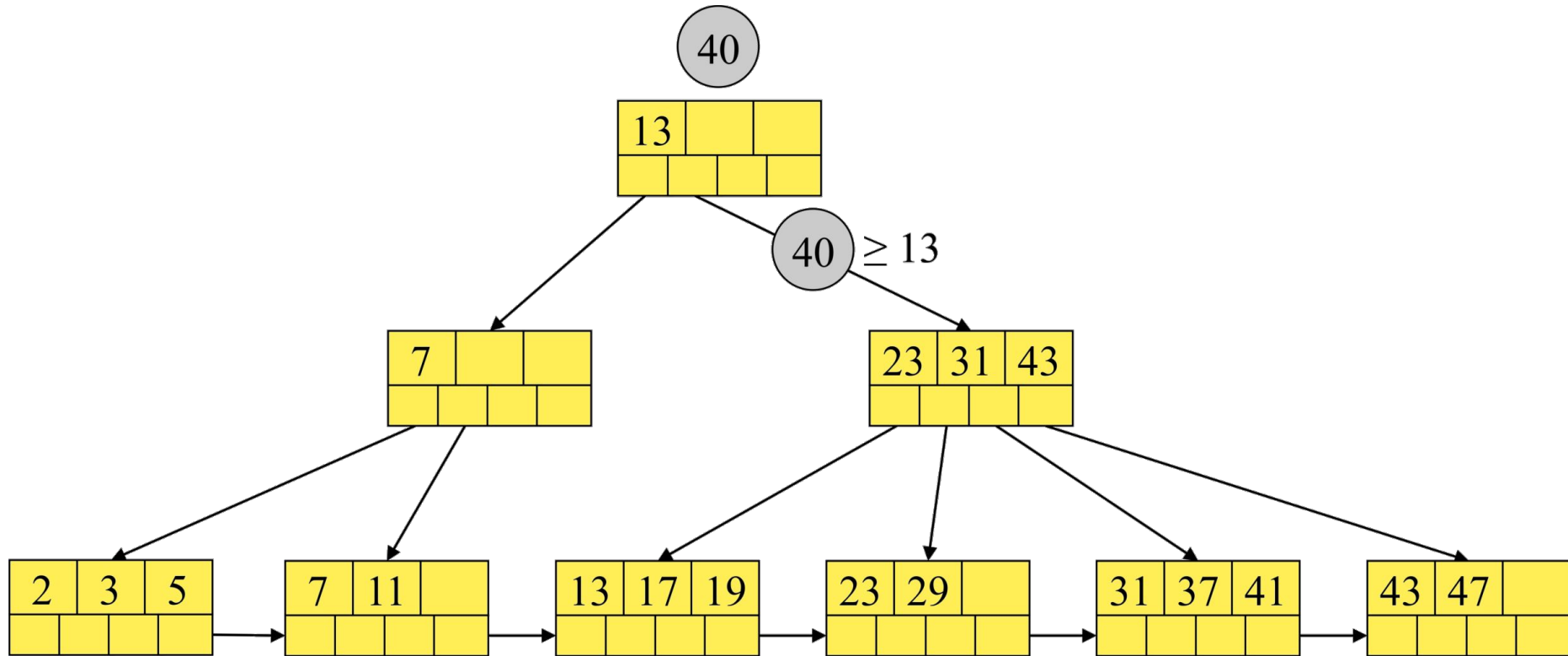
B+ trees and Insertion: Lookup (1)

We are looking for a place for the new key (K=40)
in the appropriate leaf.



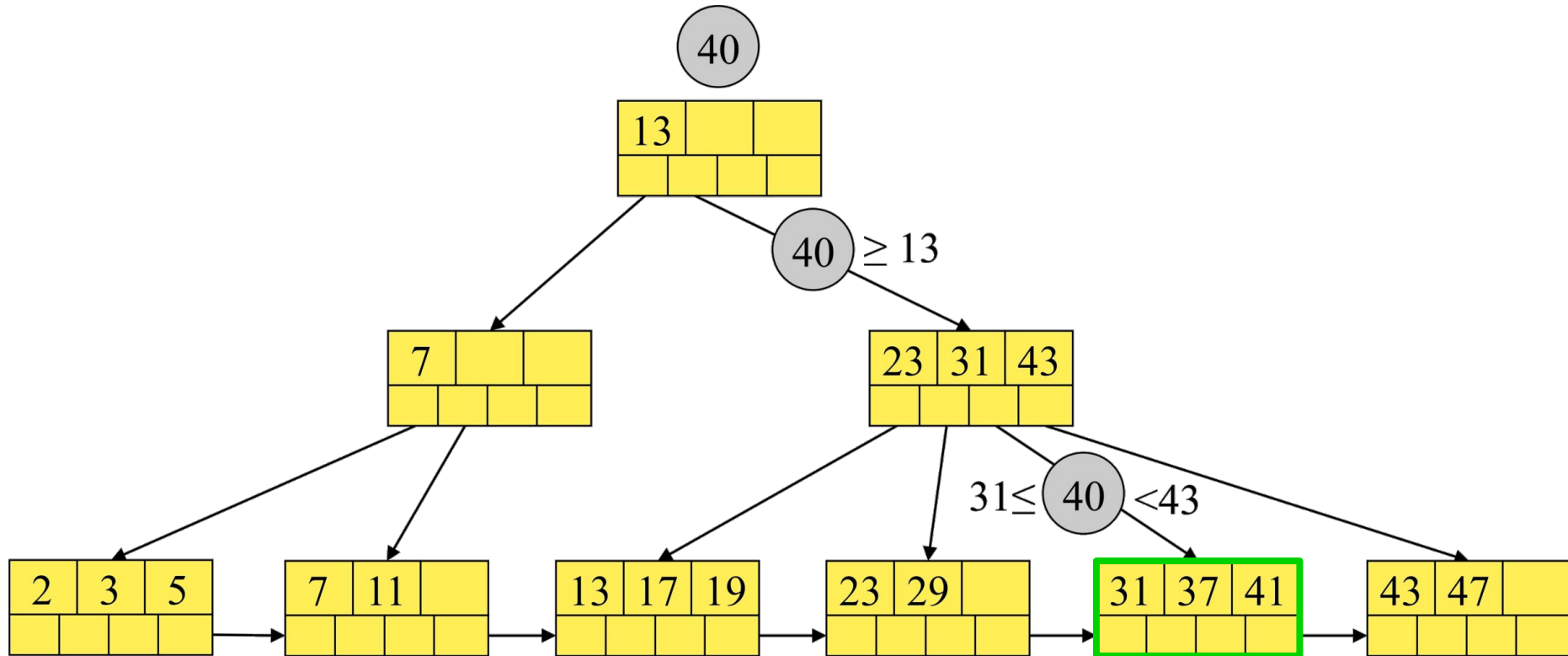
B+ trees and Insertion: Lookup (2)

We are looking for a place for the new key (K=40)
in the appropriate leaf.



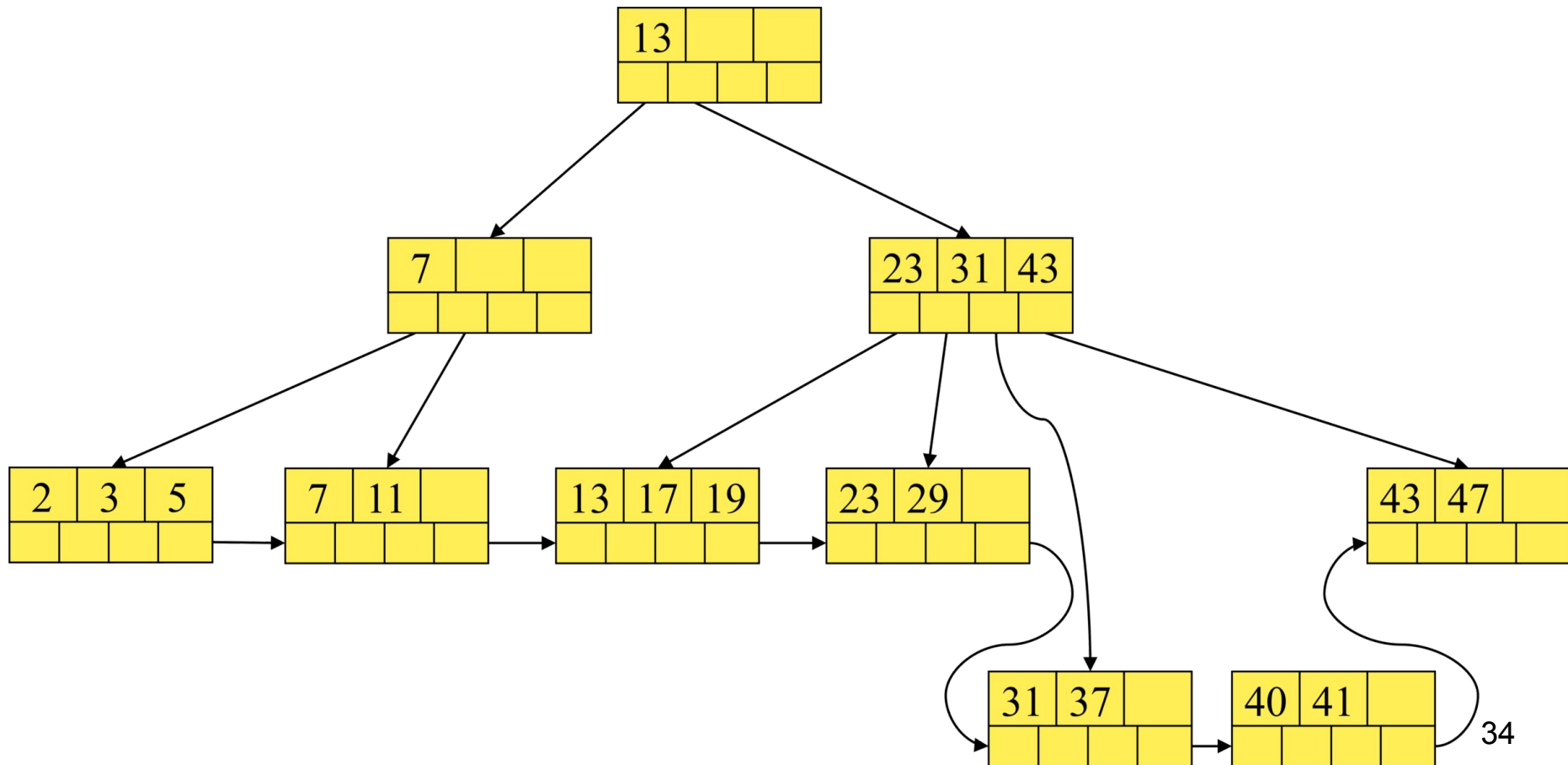
B+ trees and Insertion: Lookup (3)

We are looking for a place for the new key ($K=40$) in the appropriate leaf.



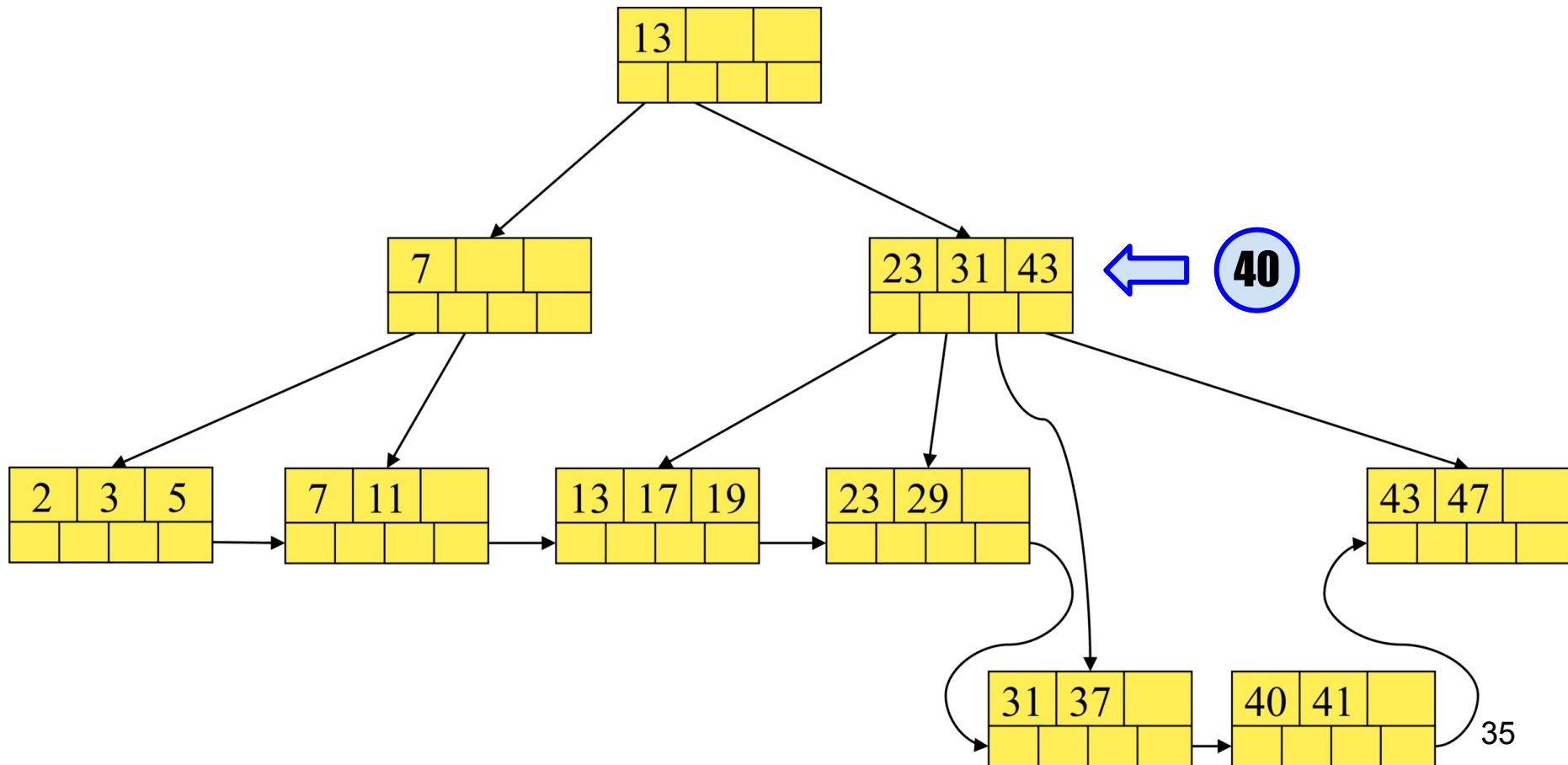
B+ trees and Insertion: Split (1)

If there is no room in the proper leaf, we split the leaf into two and divide the keys between the two new nodes.



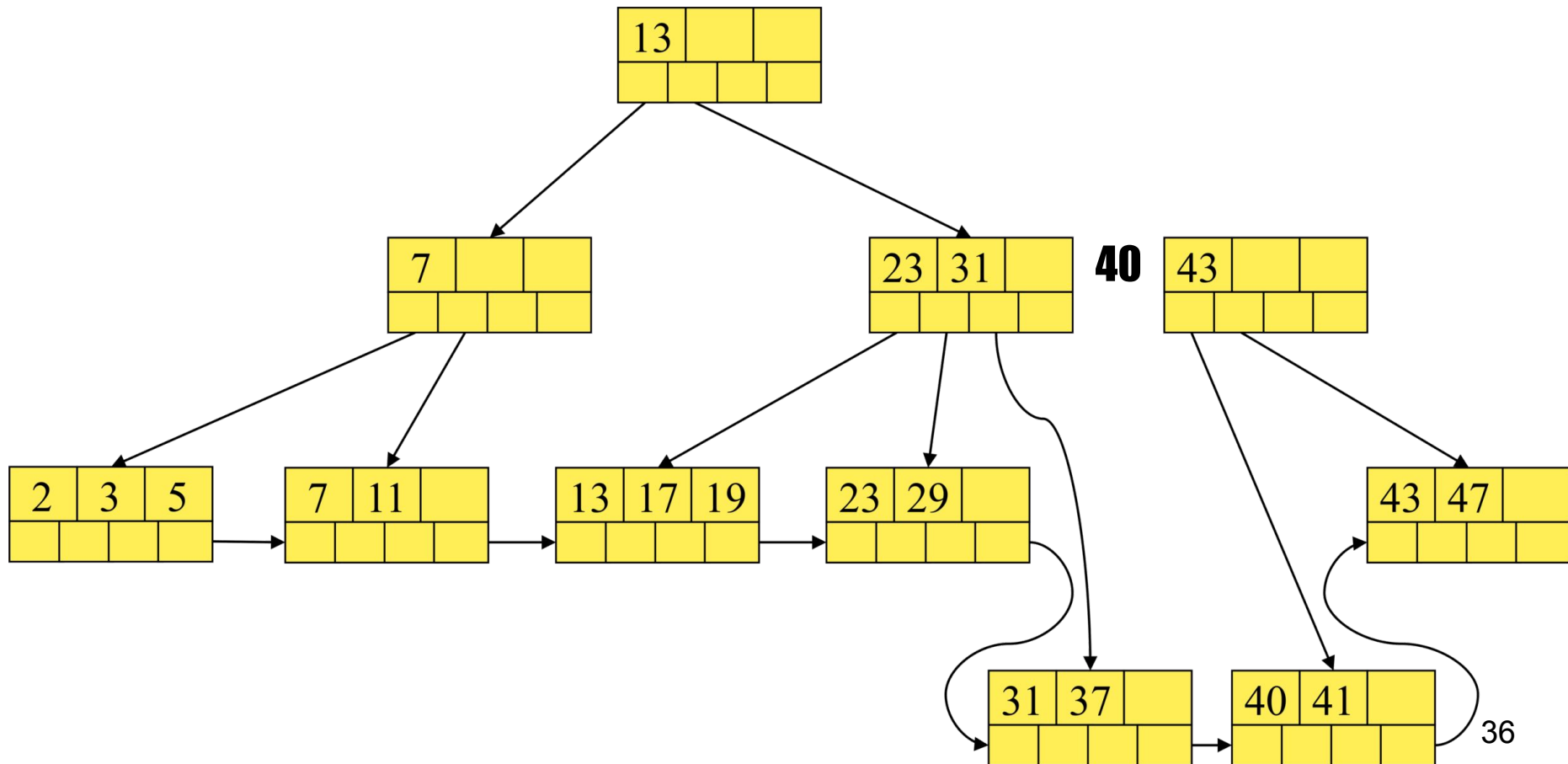
B+ trees and Insertion: Split (2)

If there is no room in the proper leaf, we split the leaf into two and divide the keys between the two new nodes.



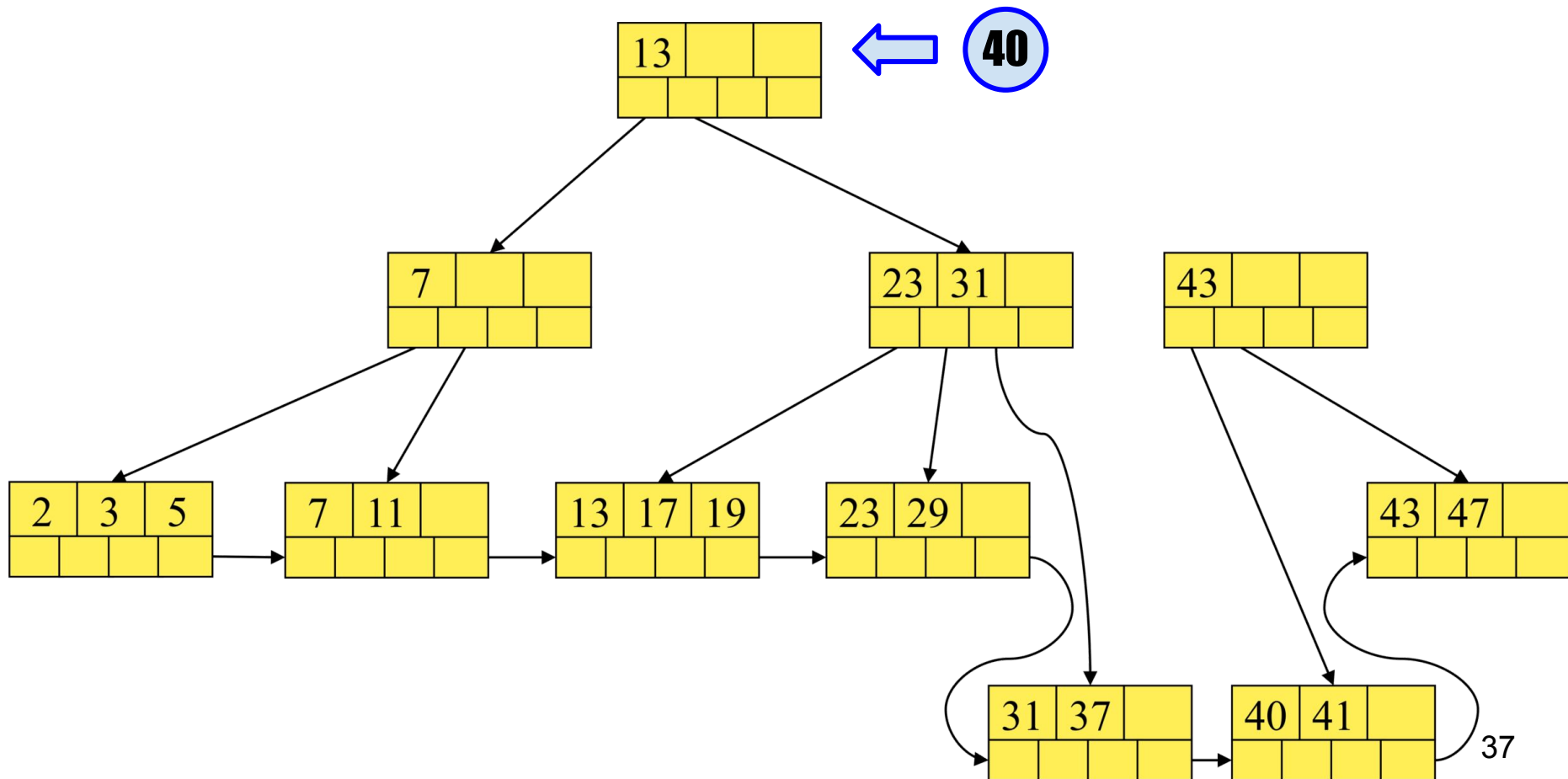
B+ trees and Insertion: Propagation (1)

The splitting of nodes at one level appears to the level above as if a new key-pointer pair needs to be inserted at that higher level.



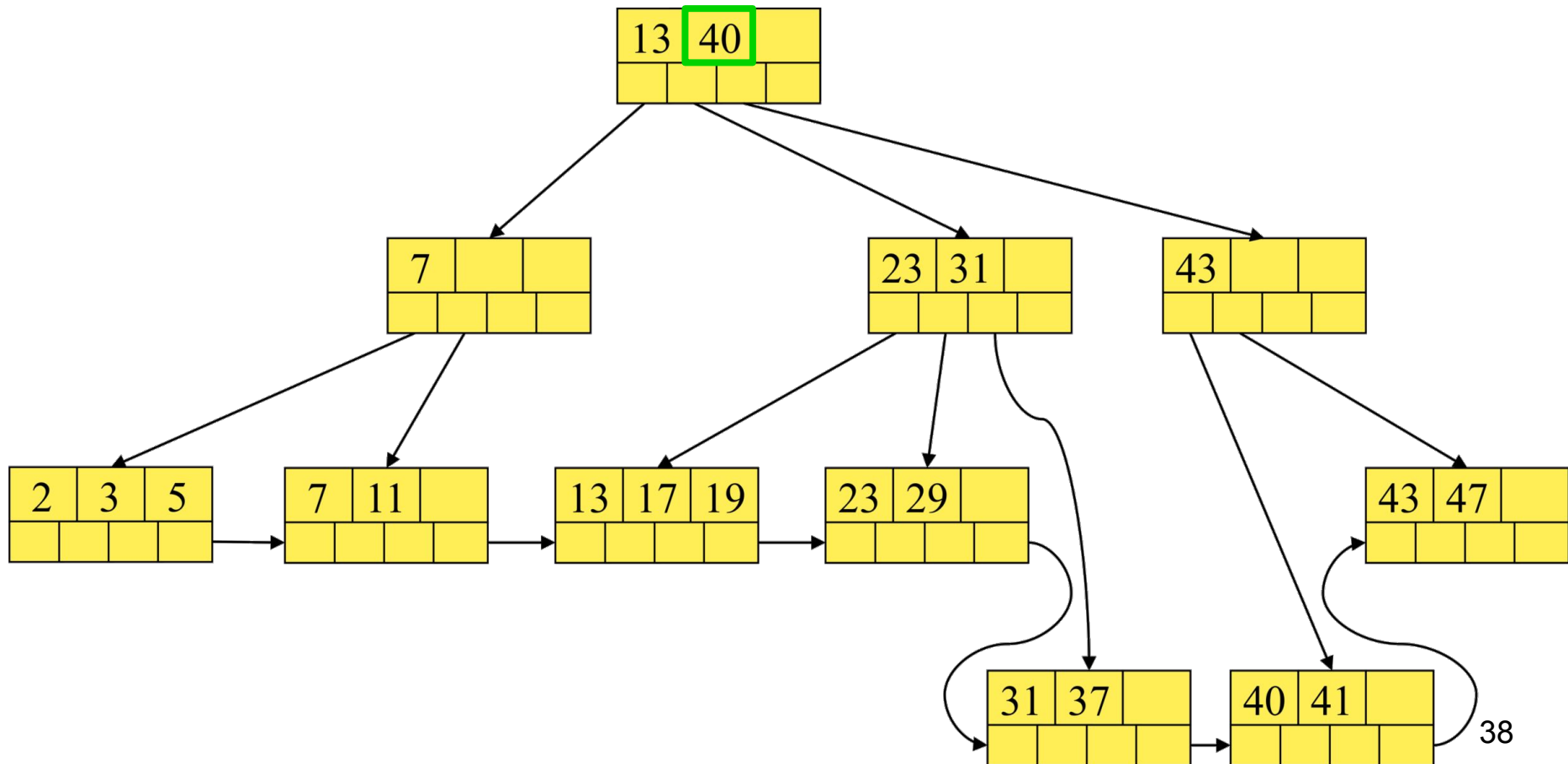
B+ trees and Insertion: Propagation (2)

The splitting of nodes at one level appears to the level above as if a new key-pointer pair needs to be inserted at that higher level.



B+ trees and Insertion: Propagation (3)

The splitting of nodes at one level appears to the level above as if a new key-pointer pair needs to be inserted at that higher level.



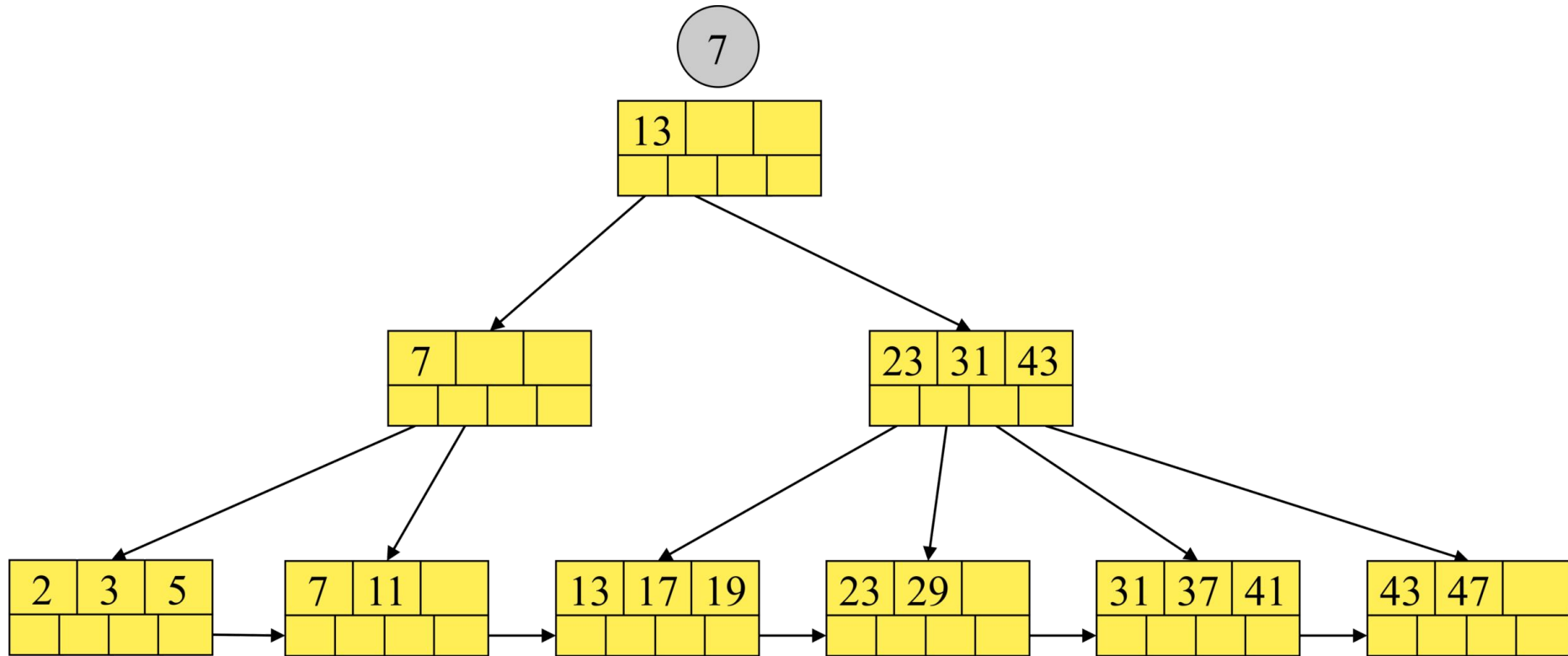
Deletion From B-Trees

The deletion is, in principle, recursive:

- We look for the key and we delete it from the leaf. We do not need to delete the key from the intermediate nodes.
- If the leaf still has at least the minimum number of keys, then there is nothing more to be done.
- Otherwise:
 - If one of the adjacent siblings of node has more than the minimum number of keys, then one key-pointer pair can be moved to the node. The keys at the parent must be adjusted.
 - Otherwise, it means that two adjacent node have no more keys than are allowed in a single node and they can be merged. We need to adjust the keys at the parent. If the parent is still full enough, then we are done. If not, then we recursively apply the deletion algorithm at the parent.

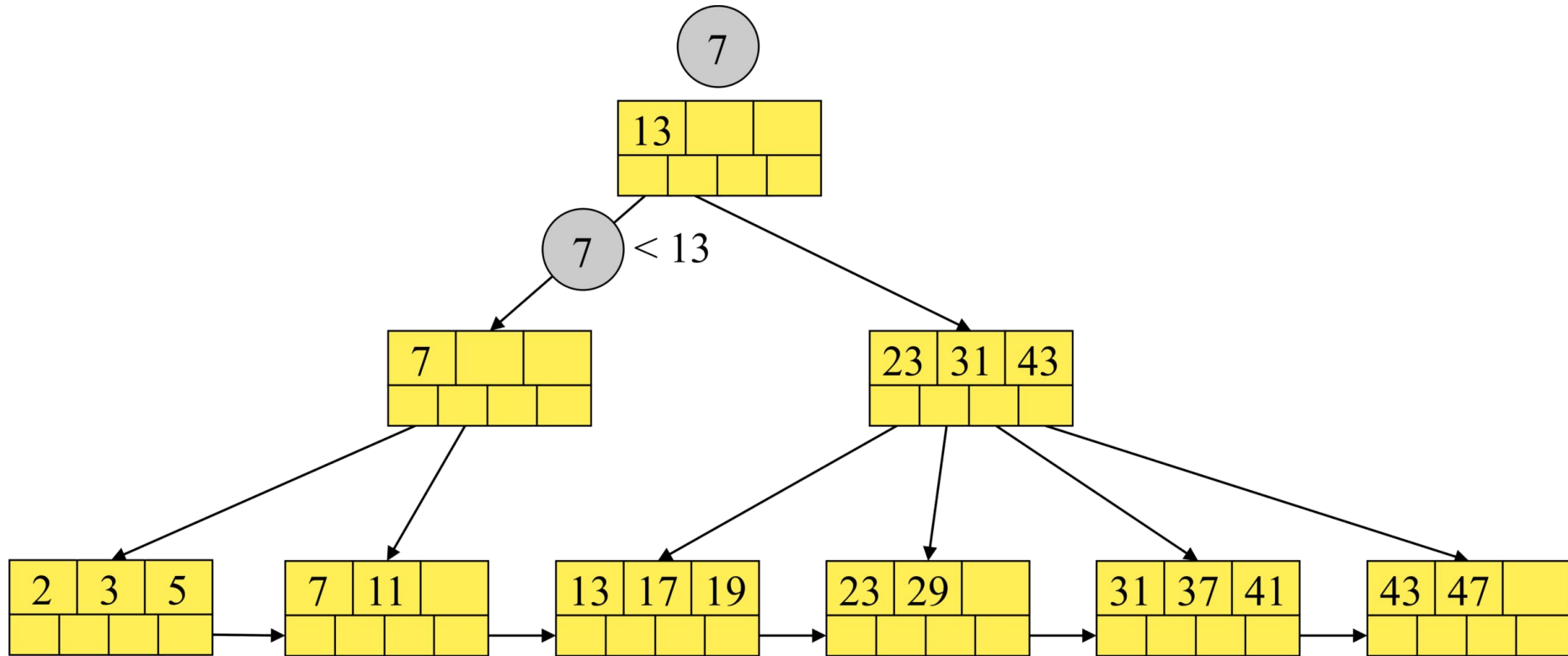
B+ trees and Deletion: Lookup (1)

We are looking for the leaf with the key (K=7) to be deleted.



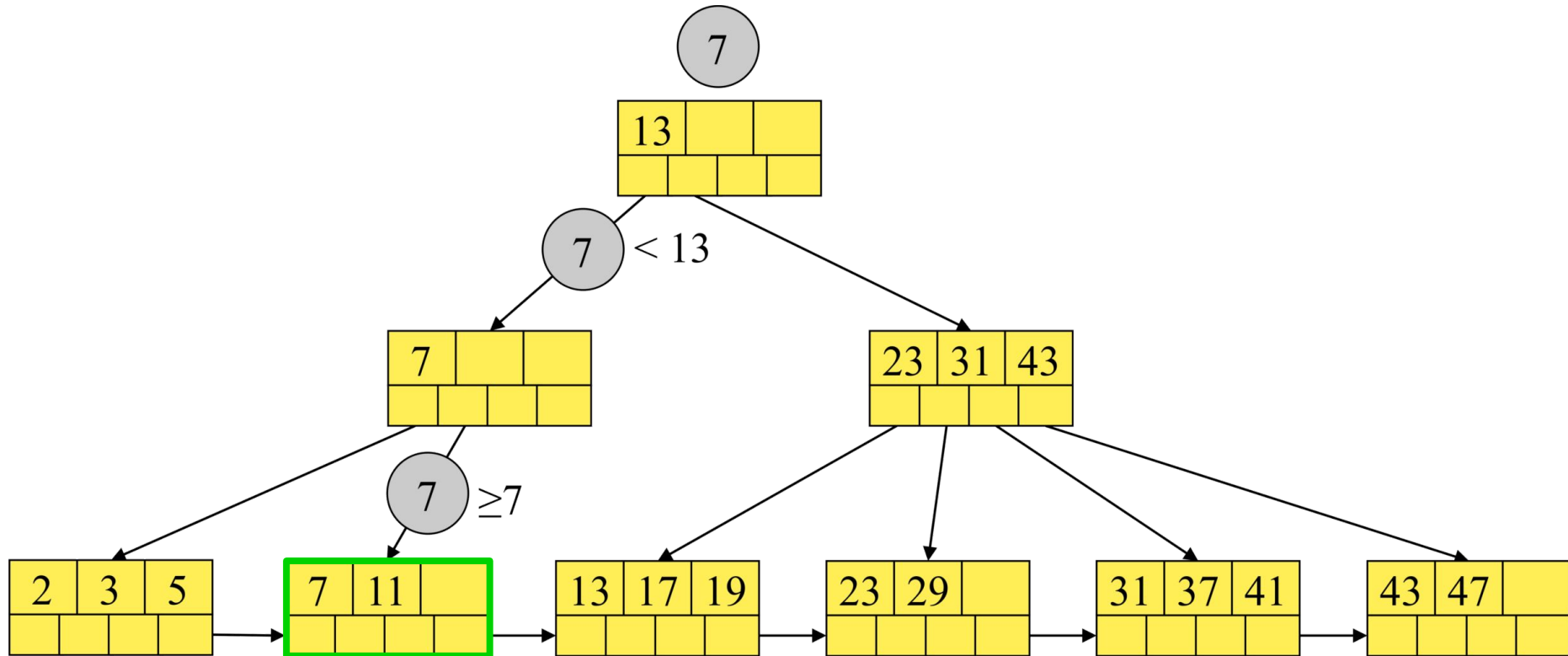
B+ trees and Deletion: Lookup (2)

We are looking for the leaf with the key (K=7) to be deleted.



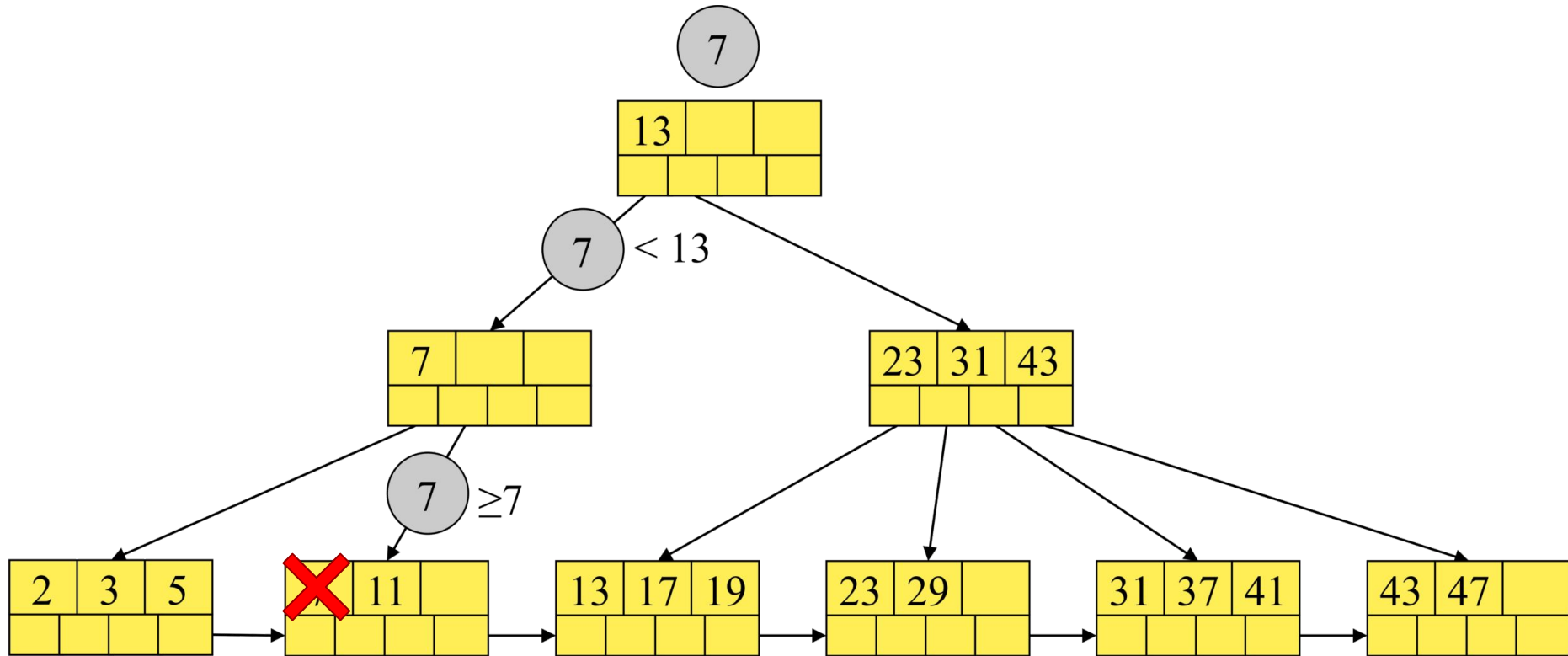
B+ trees and Deletion: Lookup (3)

We are looking for the leaf with the key (K=7) to be deleted.



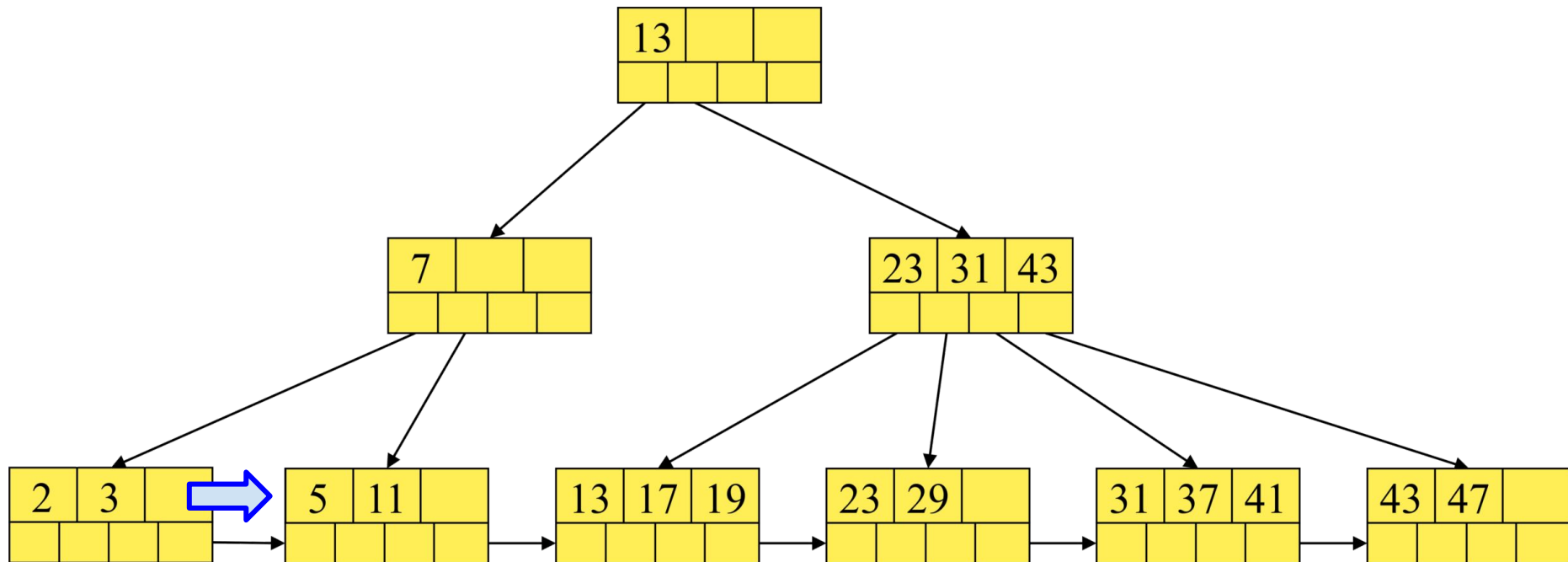
B+ trees and Deletion: Deletions

This key is found in the second leaf. We delete it, its associated pointer, and the record that pointer points to.



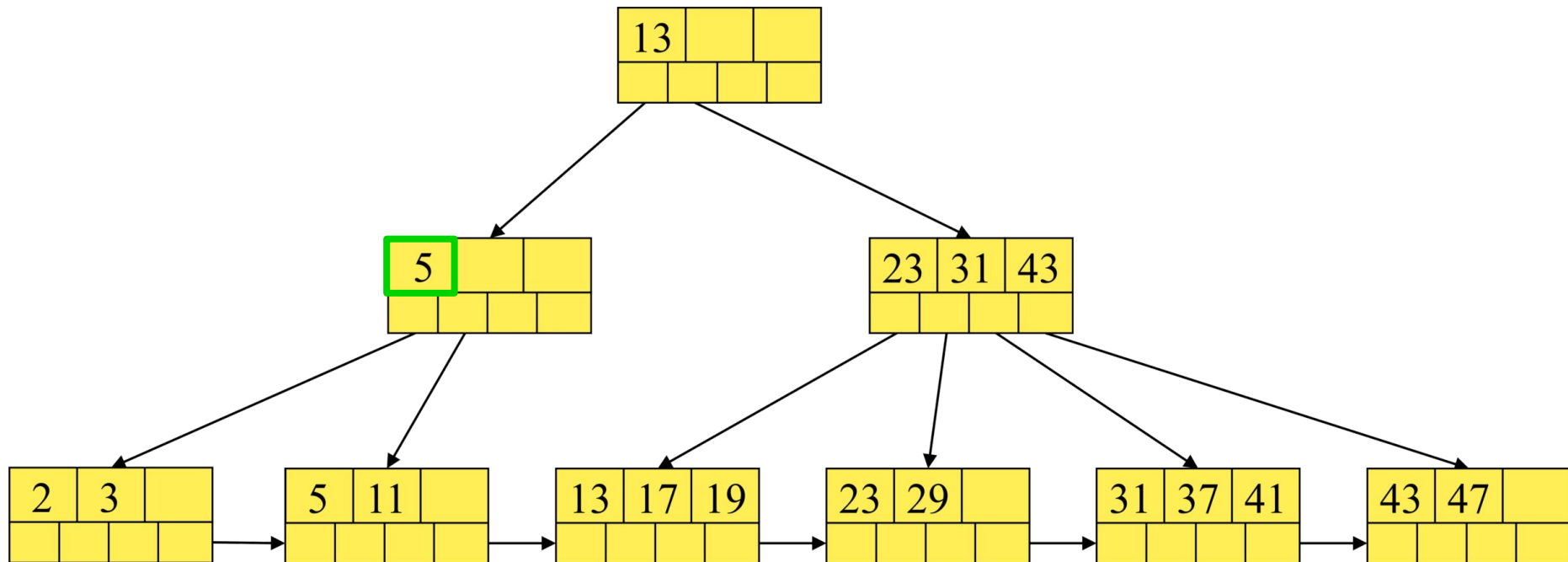
B+ trees and Deletion: Rebalance (1)

The second leaf now has only one key, and we need at least two in every leaf.



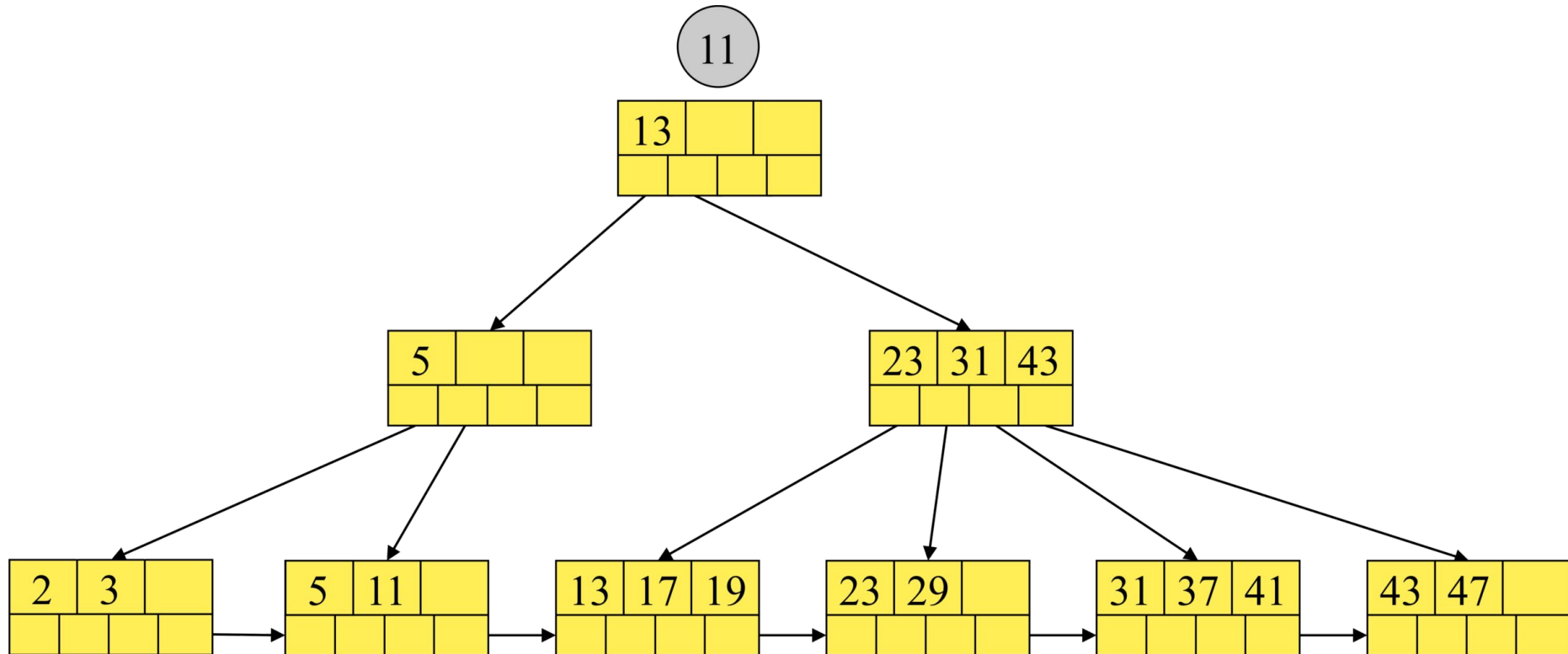
B+ trees and Deletion: Rebalance (2)

Because the lowest key in the second leaf is now 5, the key in the parent of the first two leaves has been changed from 7 to 5.



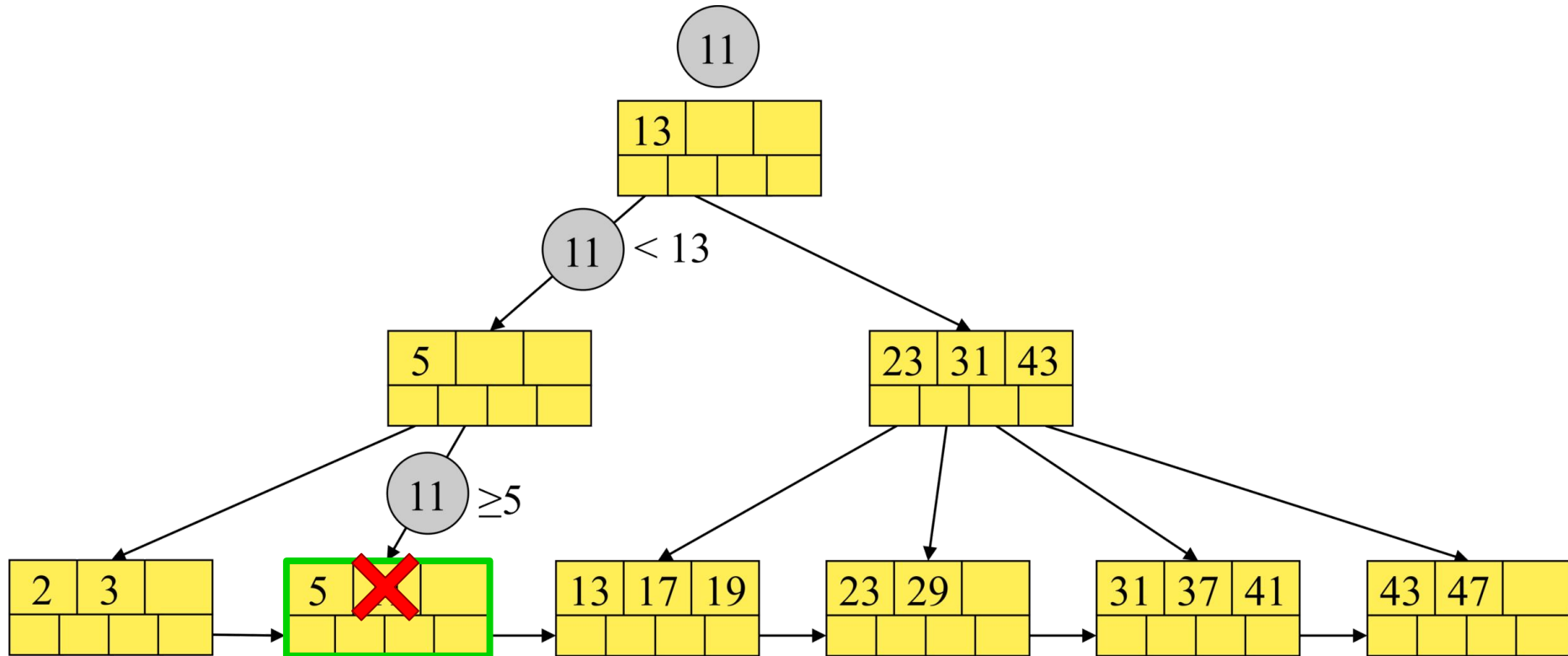
B+ trees and Deletion: Lookup (1)

We are looking for the leaf with the key (K=11) to be deleted.



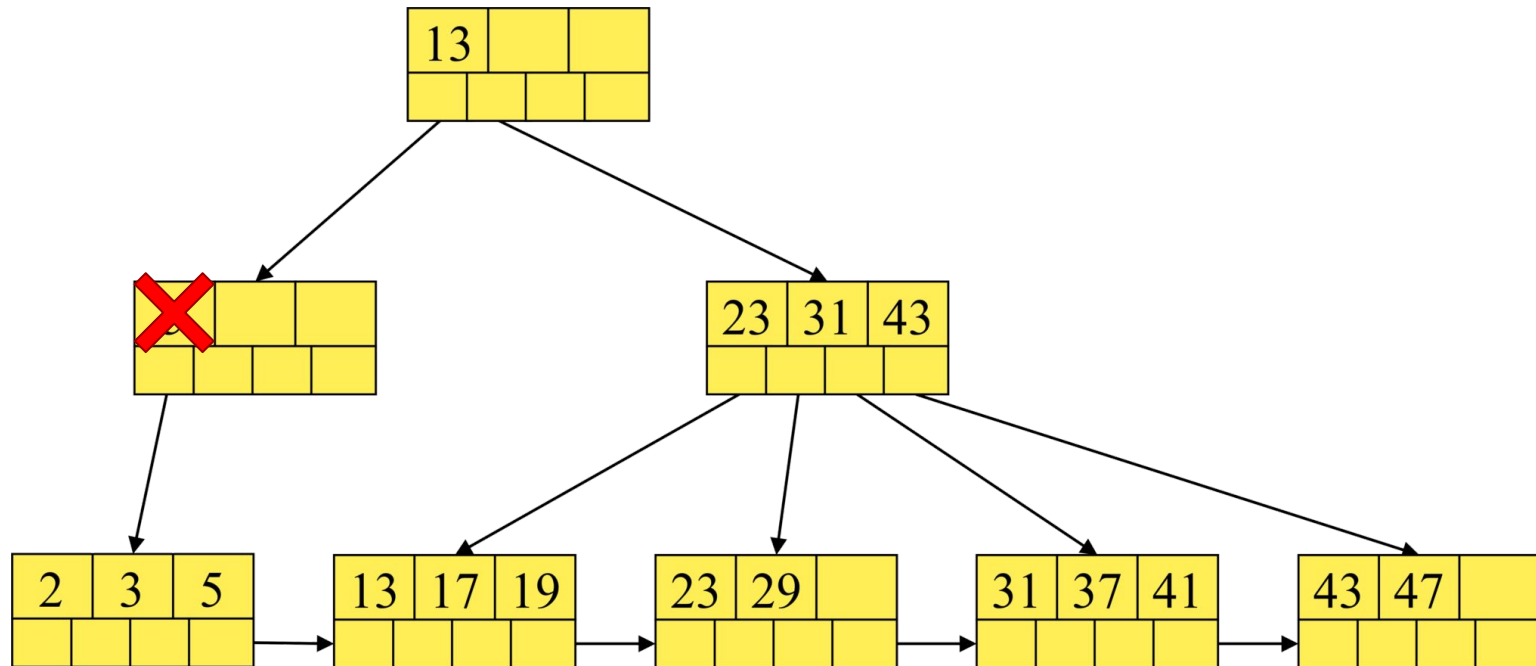
B+ trees and Deletion: Lookup (2)

We are looking for the leaf with the key (K=11) to be deleted.



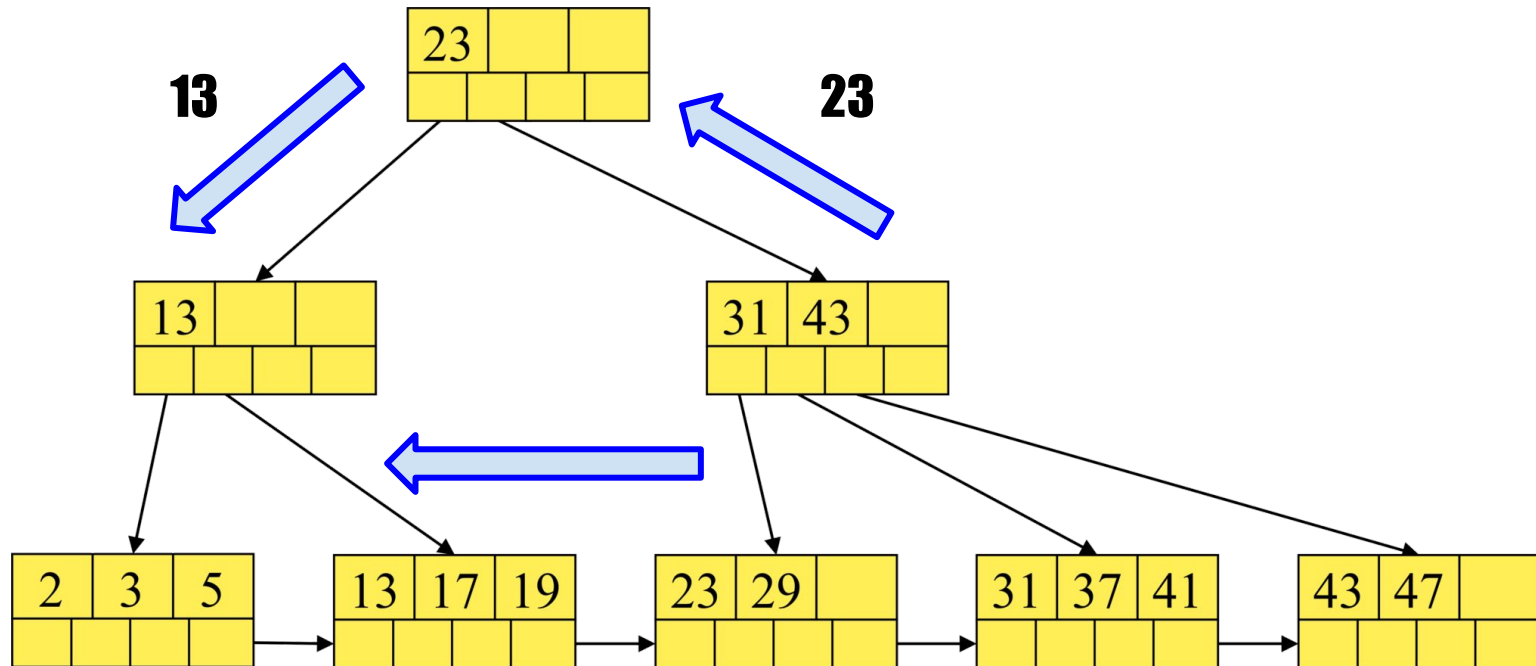
B+ trees and Deletion: Merge

We cannot borrow from the first leaf. Additionally, there is no sibling to the right from which to borrow. Thus, we need to merge the second leaf with a sibling, namely the first leaf.



B+ trees and Deletion: Rebalance

The pointers and keys in the parent are adjusted to reflect the new situation at its children. The key 5 is no longer relevant and is deleted.



Search Complexity with B+ trees

- It depends on B+ tree height L (i.e., the number of levels).
- Let's assume we have **N leaves** and on average $n/2$ pointers per leaf. What is the resulting height of the B+ tree?
 - Level 1: 1 node (the root).
 - Level 2: $n/2$ nodes (by assumption).
 - Level 3: $(n/2)^2$ nodes.
 - ...
 - **Level L : $(n/2)^{L-1}$ nodes.**
- If $(n/2)^{L-1} \leq N \leq (n/2)^L$ then $L-1 \leq \log_{(n/2)} N$ and $L \geq \log_{(n/2)} N$.

$$\log_{(n/2)} N \leq L \leq \log_{(n/2)} N + 1$$

Search Complexity - an example (1)

Let's assume that:

- Block size: 4096 B.
- Key size: 4 B.
- Pointer size: 8 B.
- There is no header information kept on the blocks.

The value of n is:

- A. 340
- B. 4096
- C. 48
- D. 8

Hint: we want to find the largest integer value of n such that

$$4n + 8(n + 1) < 4096$$

Search Complexity - an example (2)

- In the previous example, we determined that 340 key-pointer pairs could fit in one block.
- Suppose that the average block (except the root) has an occupancy midway between the minimum ($n/2$, i.e., 170) and the maximum (n , i.e., 340). A typical block has 255 pointers.
- Let's consider a B+tree of 3 levels. With a root ($L=1$), 255 children ($L=2$), and $255^2 = 65,025$ leaves ($L=3$), we shall have about 16.6 million pointers to data records.
- The first two levels occupy $(4096 + 4096 \cdot 255) = 1\text{MB}$ and can be kept in main memory.
- If so, then every search through a 3-level B+ tree requires just two disk read to retrieve the RID.