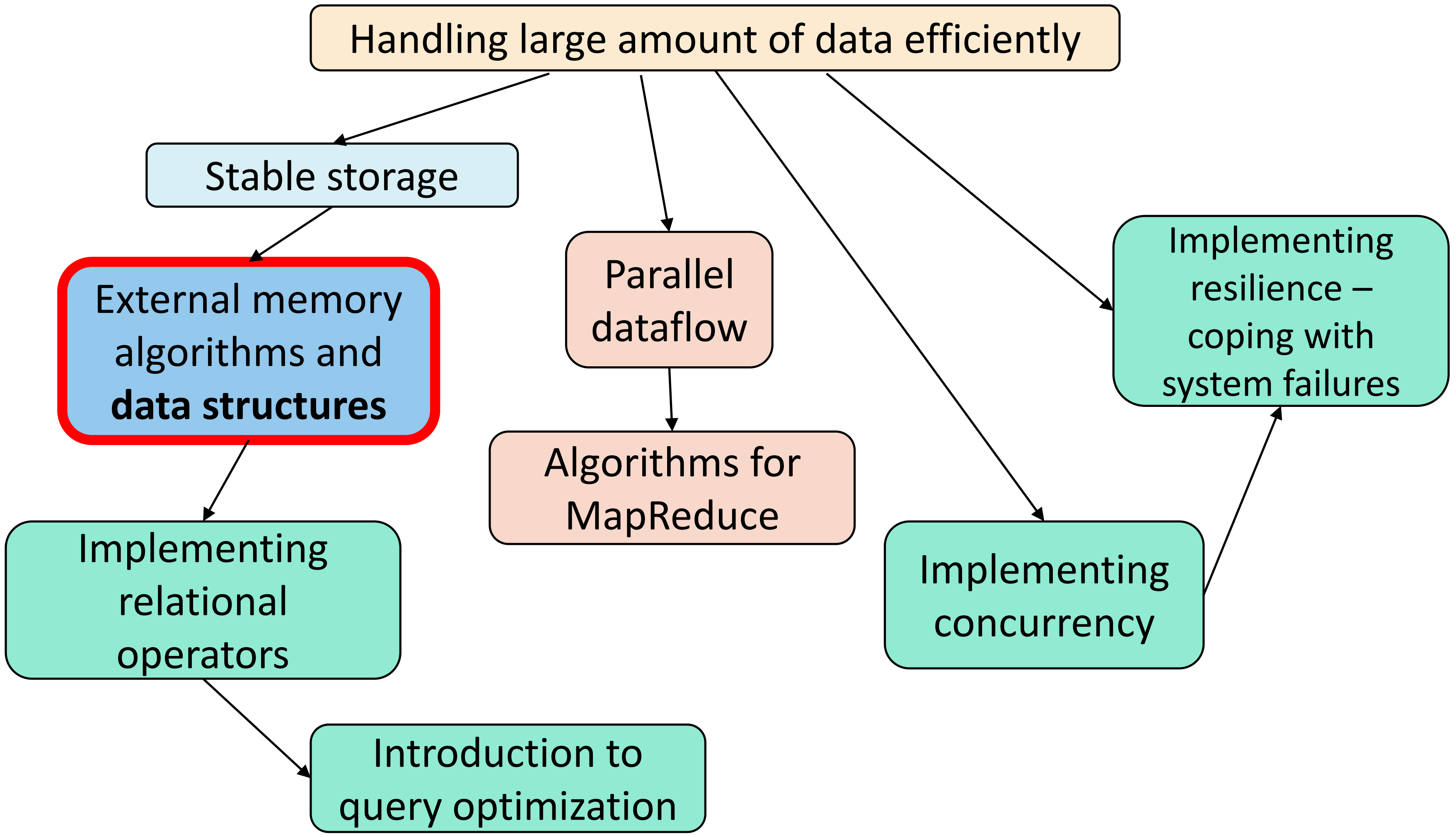
Roadmap



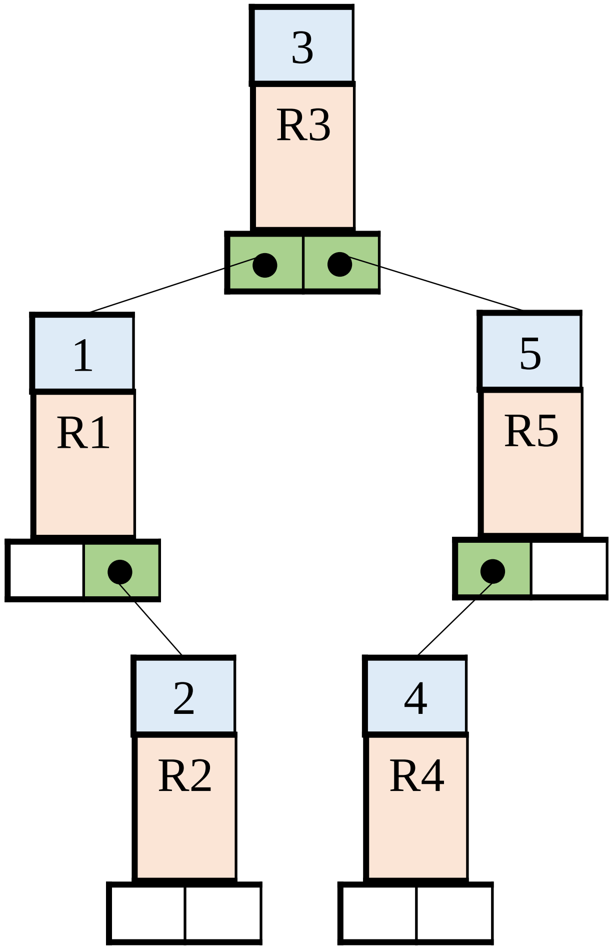
Handling large amount of data efficiently
Stable storage
External memory algorithms and **data structures**
Parallel dataflow
Algorithms for MapReduce
Implementing resilience – coping with system failures
Implementing concurrency
Implementing relational operators
Introduction to query optimization

Lecture 02.05

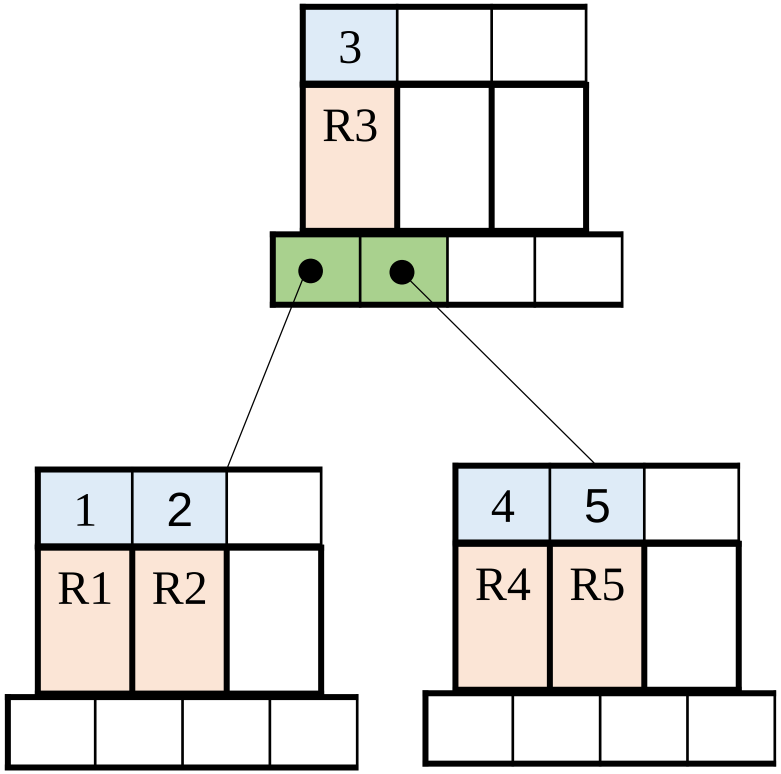
B-trees

By Marina Barsky
Winter 2017, University of Toronto

From binary search trees to k-2k B-trees

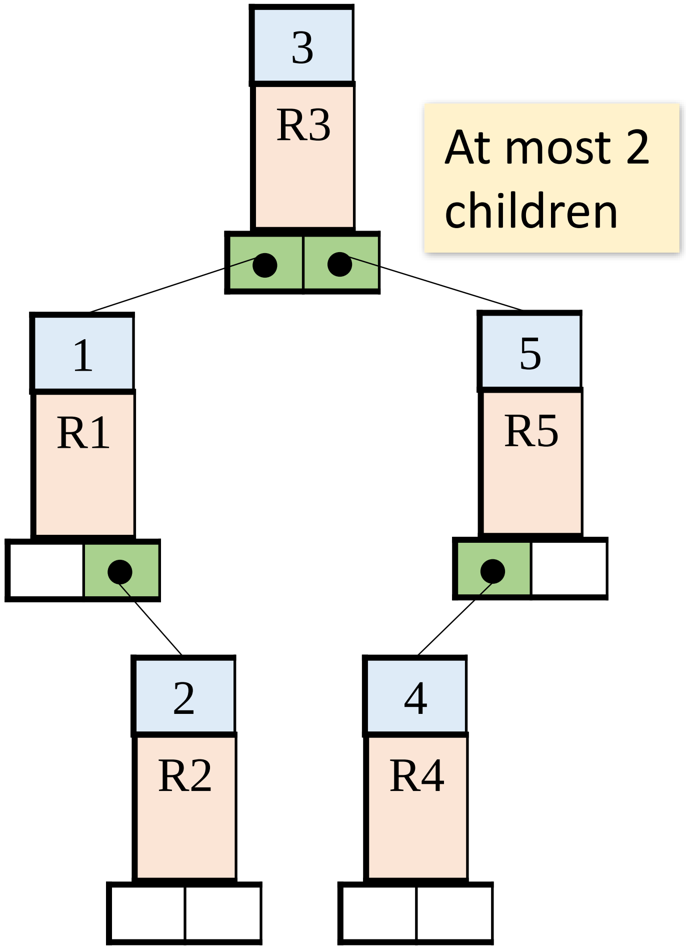


Binary search tree

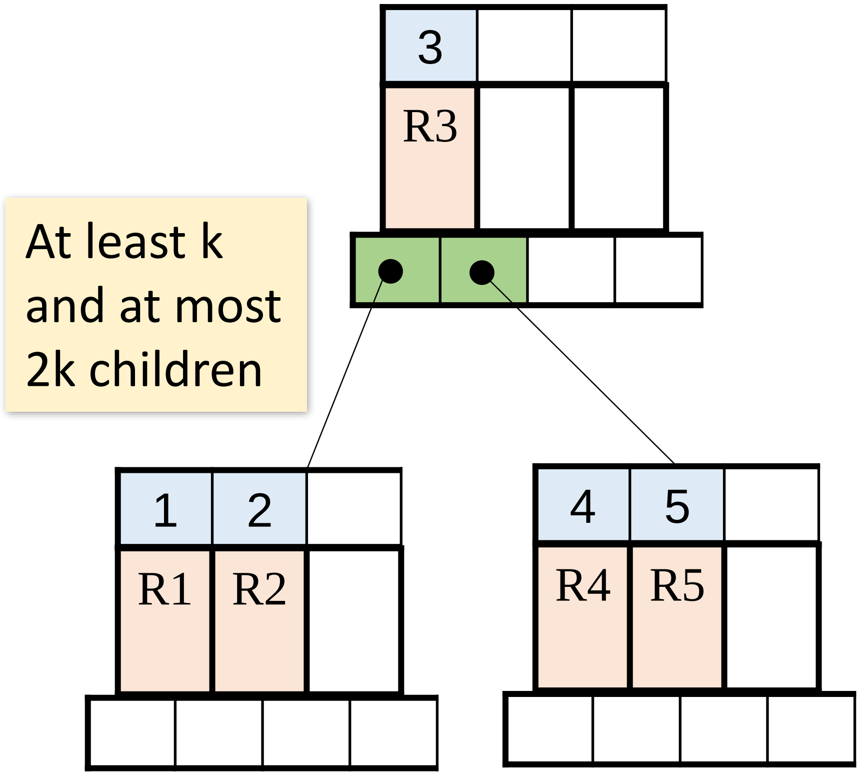


2-4 B-tree

From binary search trees to k-2k B-trees



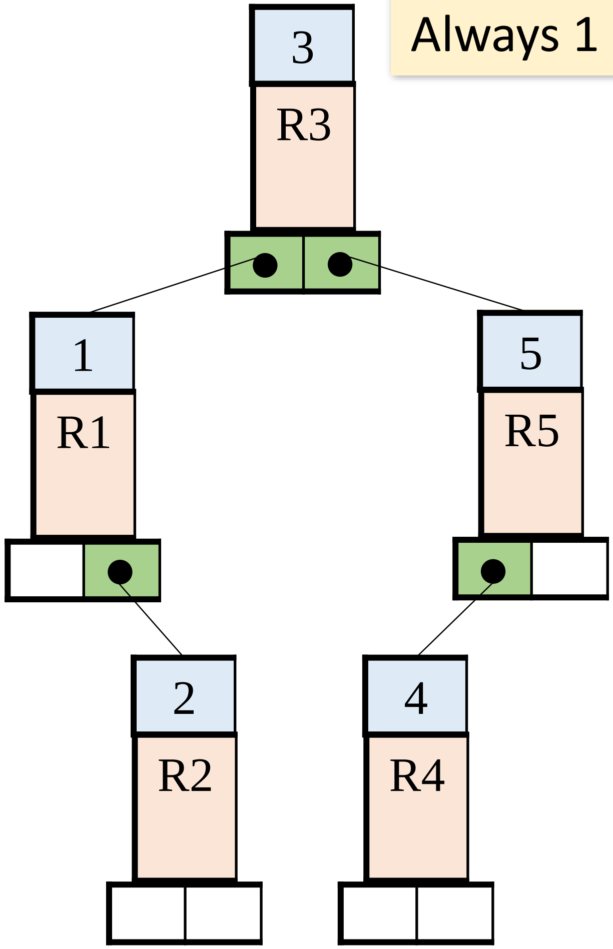
Binary search tree



2-4 B-tree

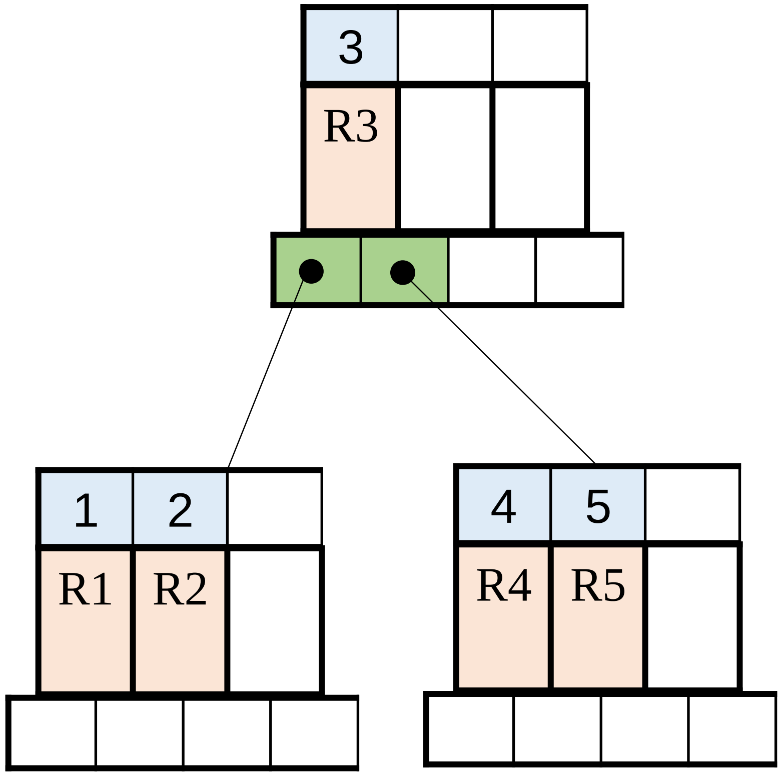
From binary search trees to k-2k B-trees

Always 1 key



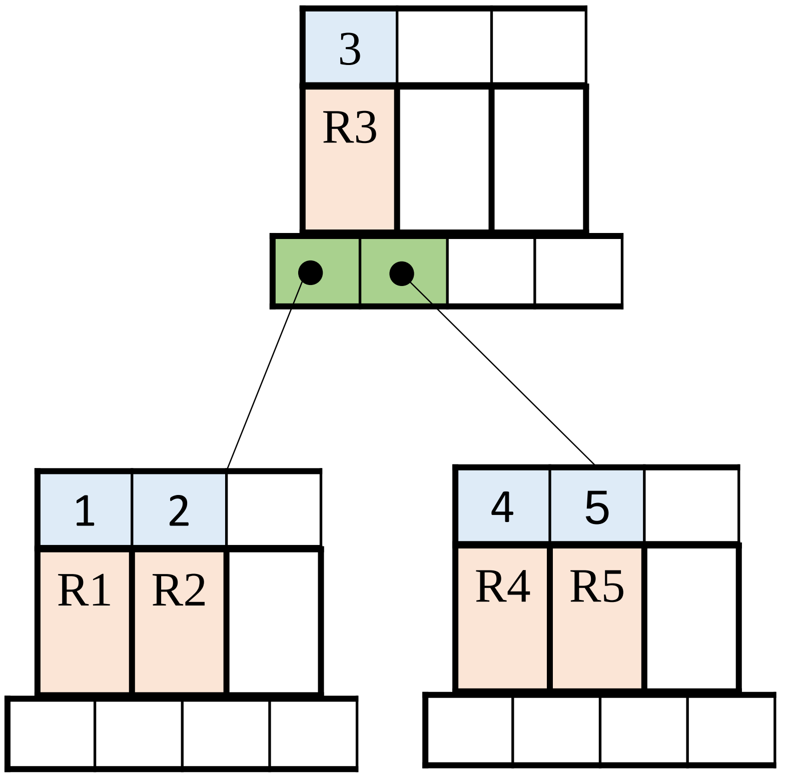
Binary search tree

At least k-1 and at most 2k-1 keys



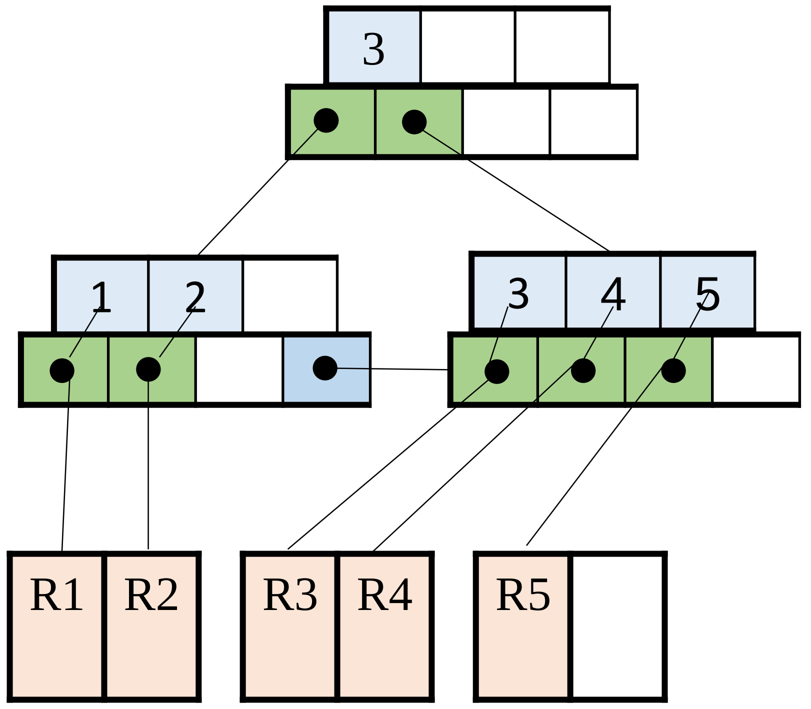
2-4 B-tree

From B-trees to B+ trees



Data stored together with the key

B-tree



Data stored separately

B+ tree

B+ tree vs. B-tree

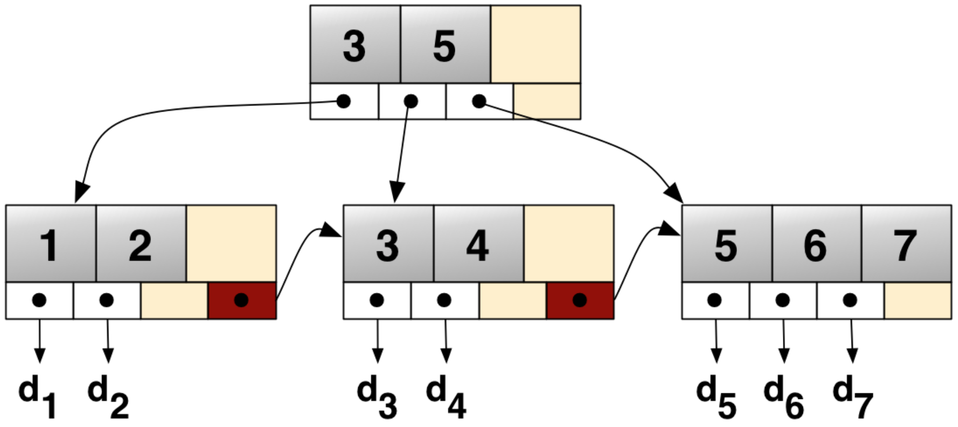
1. B+ -tree is a B-tree where internal nodes contain only keys and navigation pointers (not records, not pointers to records), and all the records (or pointers to records) are stored in leaves.
2. In B+ tree each internal node is stored in a page, and more keys fit in a single page. The navigational part of the index is overall smaller, and partly manageable in RAM.
3. The leaf nodes of B+ trees are linked, so doing a full scan of all objects in a tree requires just one linear pass through all the leaf nodes. A B tree, on the other hand, would require a traversal of every level in the tree (random access).

B+ - tree (or simply B-tree)

- B+ tree is the only variant of B-trees used in DBMS
- In all research papers and implementations: we say B-tree - imply B+ -tree

k-2*k* B trees: properties I

- Each node contains p pointers: ***k <= p <= 2k***



k=2.

Each node has 2,3,or 4 child pointers and between 1 and 3 keys

k-2*k* B trees: properties II

- There are 2 types of nodes:
  - **Internal** (non-leaf) node:
    - all p pointers point to the child nodes
    - p-1 keys contain navigational info
  - **Leaf** node: 1 pointer points to the next leaf
    - p-1 key-value pairs (child pointers), where value can be RID or the entire record
    - the number of child pointers is at least k*

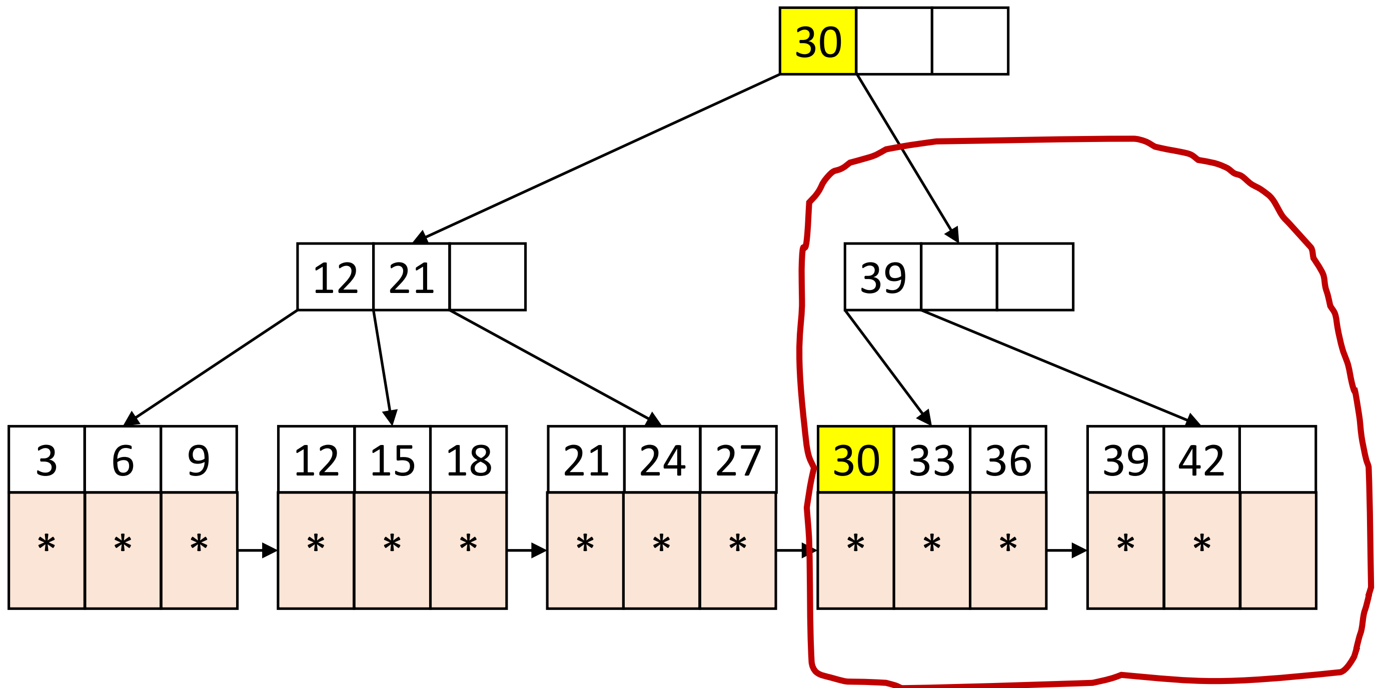
---

* In practice, the leaf node may have its own parameters on min and max key-value pairs, because the size of a value in key-value pair can be larger than the pointer used in the internal node. However, for the purposes of this lecture, we assume that min number of key-value pairs is k, and max is 2k-1 (1 pointer points to the next leaf).

k-2*k* B trees: properties III

- Each key in an internal node guides the search:

  All keys in the left sub-tree of a given key *X* have key value < *X*, all keys in the right sub-tree have key value >= *X*



Data entry

- In a key-value pair of B-tree leaves, the value can be:
  - RID – which gives an exact location of a data record in a data file: RID=<block ID, slot #>
  - List of RIDs – in case of multiple duplicate keys
  - Data record itself

- Let’s call any of this value types a ***data entry***

Degree=order=fanout =branching factor

- Degree *d (=2*k)* means that all internal nodes have space for at most *d* child pointers

Example

- Each node is stored in 1 block of size 4096 bytes
- Let
  - key 4 Bytes,
  - pointer 8 Bytes.
- Let's solve for *d*:

$4(d - 1) + 8(d) \leq 4096$

$\Rightarrow \; d \leq 341$

B-tree capacity: example

$d \approx 300$

a typical node is 67% full (*fill factor*) ≈ 200 keys in each node

We have:

- 200 keys at the root
- At level 2 – for each key – another 200 keys – total 200^2 nodes
- At level 3: 200^3
- At level 4: $200^4 \approx 16 \times 10^8$ records can be indexed.

- Suppose each record = 1 KB - we can index a relation of size

  $16 \times 10^8 \times 10^3 \approx 1.6$ TB

- If the root and levels 2 and 3 are kept in main memory, then finding RID requires 1 disk I/O!

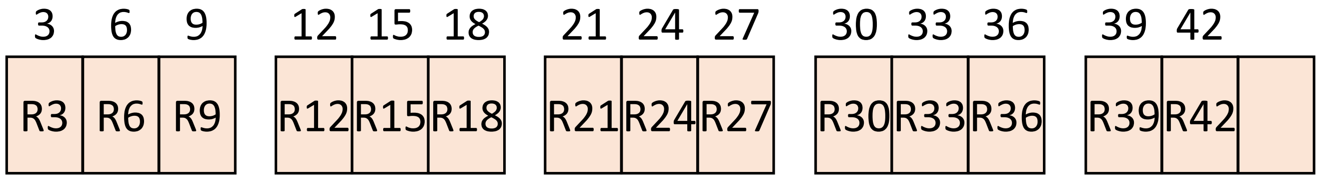
Buffering top-level nodes

- Often top levels are held in buffer pool:
    - Level 1 = 1 page = 4 KB
    - Level 2 = 200 pages = 800 KB
    - Level 3 = 40,000 pages = 160 MB

- In this case, in practice, lookup requires 1 disk I/O

B-tree construction from sorted records (bulk load)

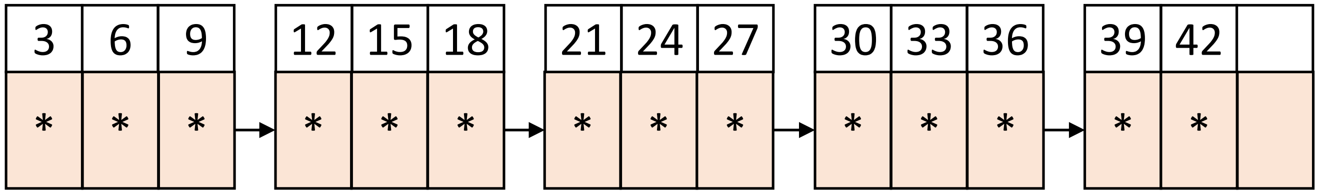
Input: list of records sorted by key:



- Output: 2-4 B-tree index

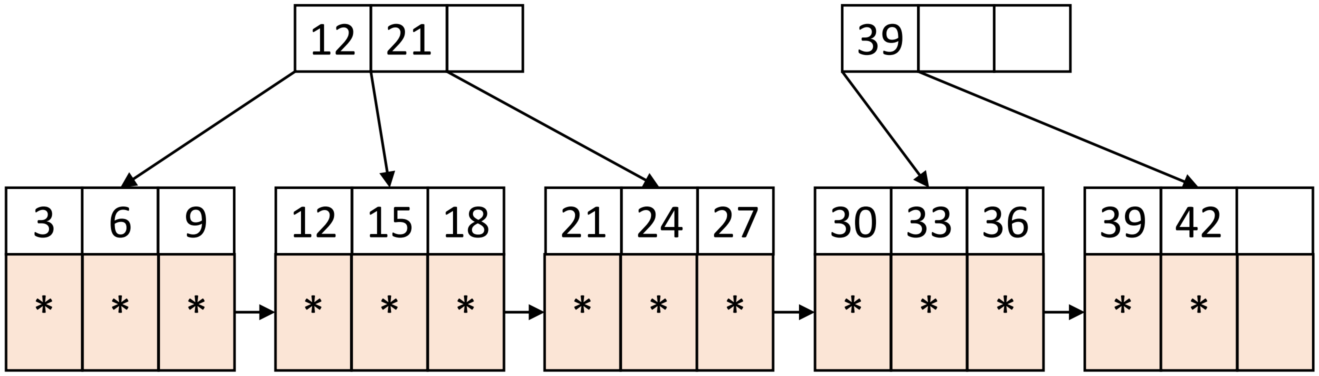
B-tree construction
1. Leafs

- Distribute records among blocks – 3 records per block
- Link leafs sequentially



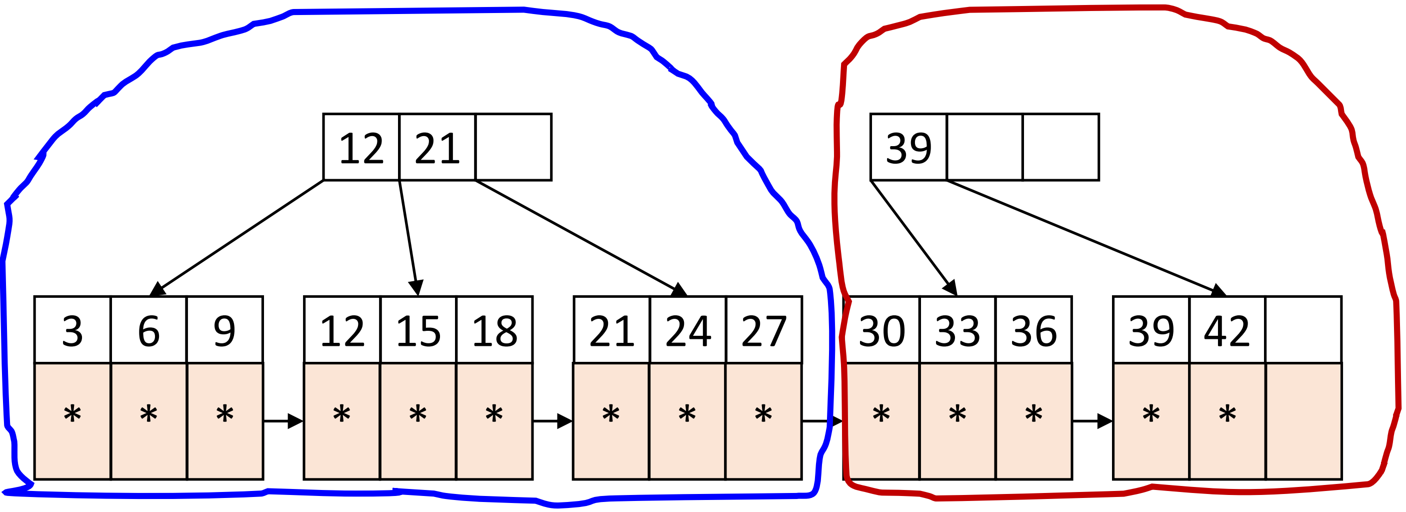
B-tree construction
2. Parent level

- Each parent node can hold at most 4 pointers – but we have 5 leafs
- We cannot put 4 children and 1 child – because each parent node has to hold at least 2 pointers
- We build one parent for 3 nodes, and another parent for 2 nodes



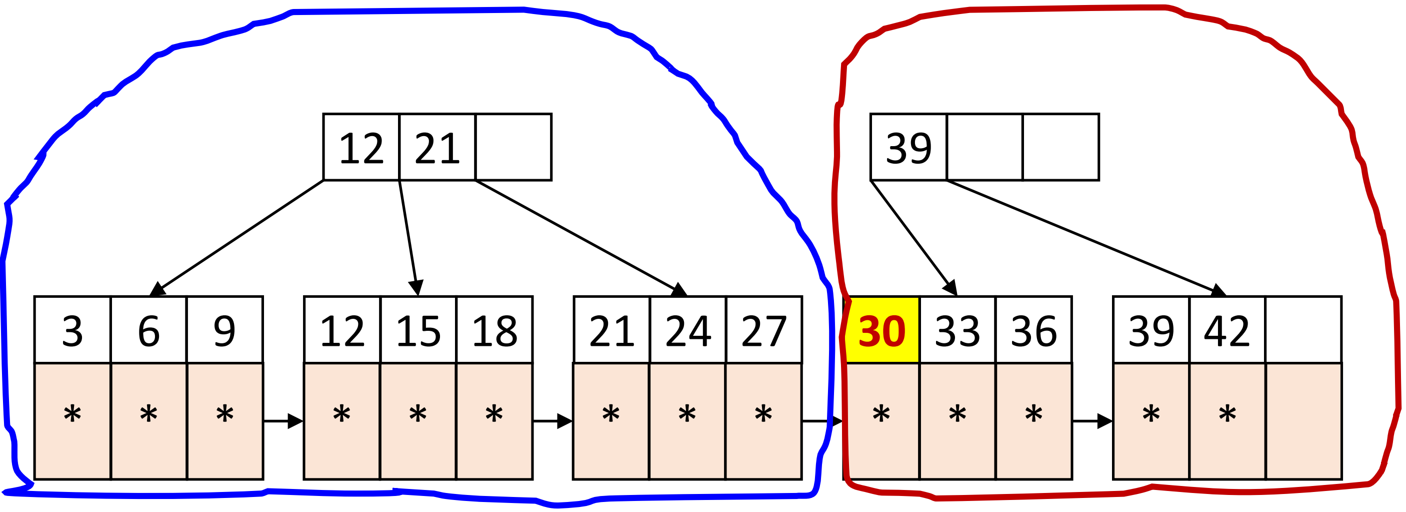
B-tree construction
3. Parent for level 2

- We need to divide keys in the left subtree from the keys in the right sub-tree



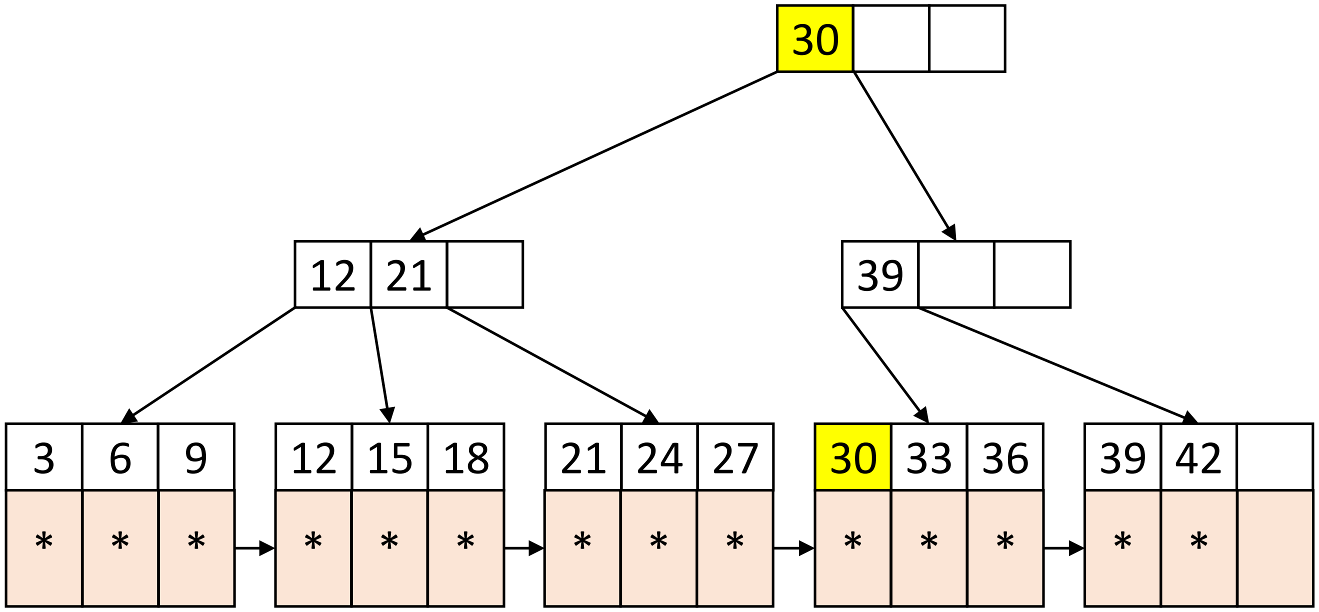
B-tree construction
3. Parent for level 2

- We need to divide keys in the left subtree from the keys in the right sub-tree
- All keys in the right subtree are >=30



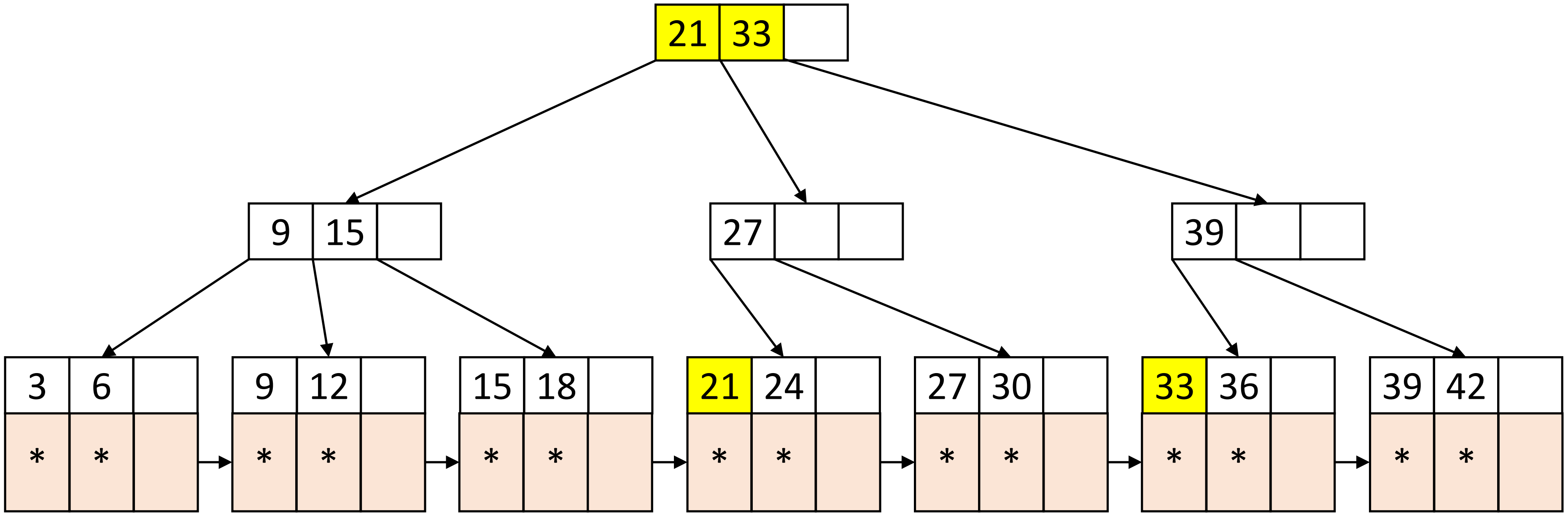
B-tree construction
3. Parent for level 2

- We need to divide keys in the left subtree from the keys in the right sub-tree
- All keys in the right subtree are >=30



B-tree construction: leave space

- In practice, the nodes will be half-filled to leave space for future insertions



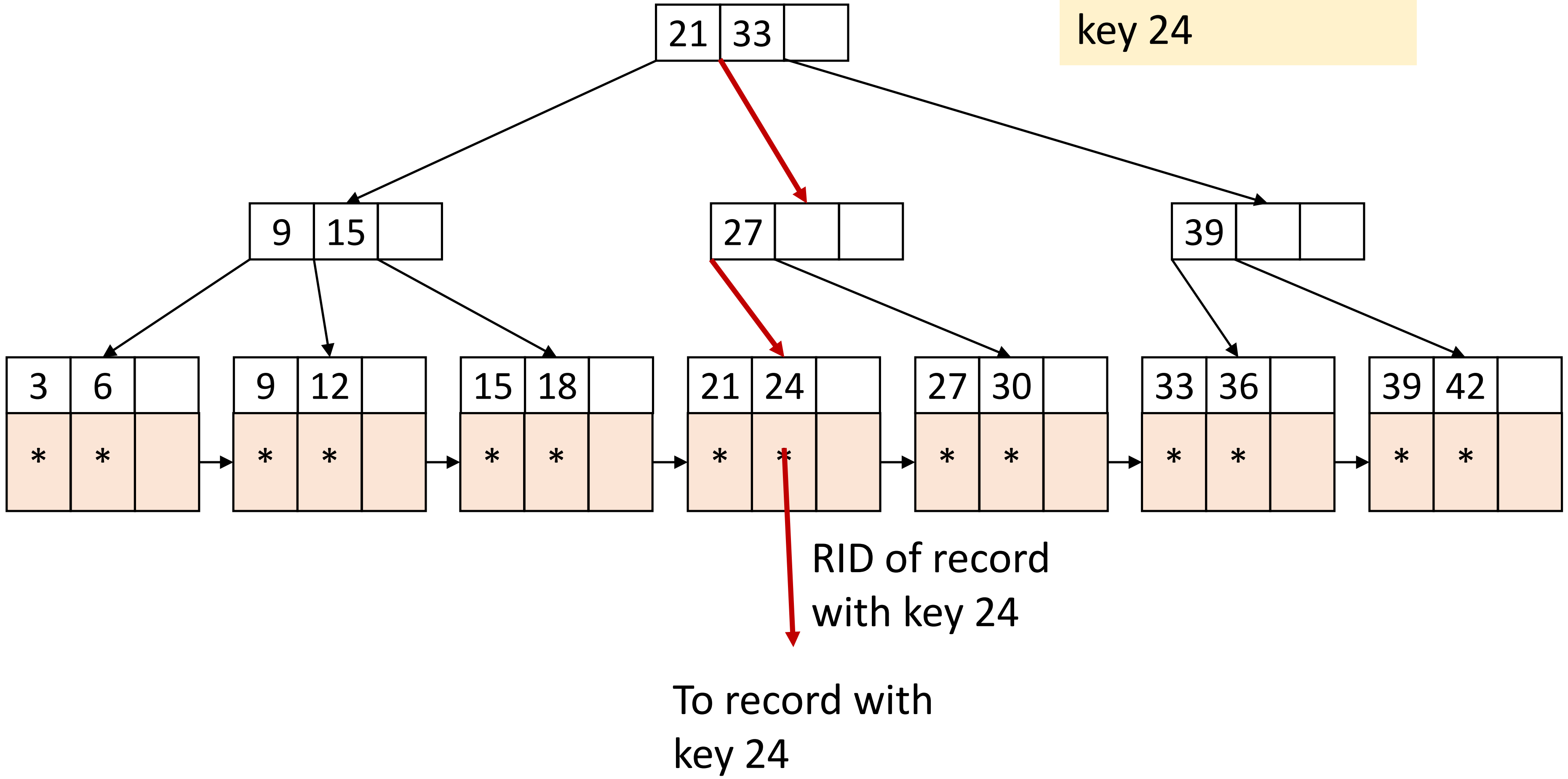
B-tree lookup

Recursive procedure:

- Ends when we are at a leaf. In this case, look among the keys there. If the *i*-th key is K, then the *i*-th pointer will contain RID of the desired record.
- If we are at an internal node with keys $K_1,K_2,\ldots,K_d$, then if $K<K_1$ we call *lookup* with the first child node, if $K_1 \leq K < K_2$ we use the second child, and so on.

B-tree lookup example

Find record with key 24



B-tree in action

- When data are inserted or removed from a node, its number of child nodes changes.
- In order to maintain the pre-defined capacity range, internal nodes must be joined or split.

- B-tree is a dynamic data structure with a guaranteed upper bound for lookup, insertion and deletion: **$O(\log_d N)$** disk I/Os

where

N – total number of leaf nodes (leaf blocks)

d – branching factor

B-tree: range search

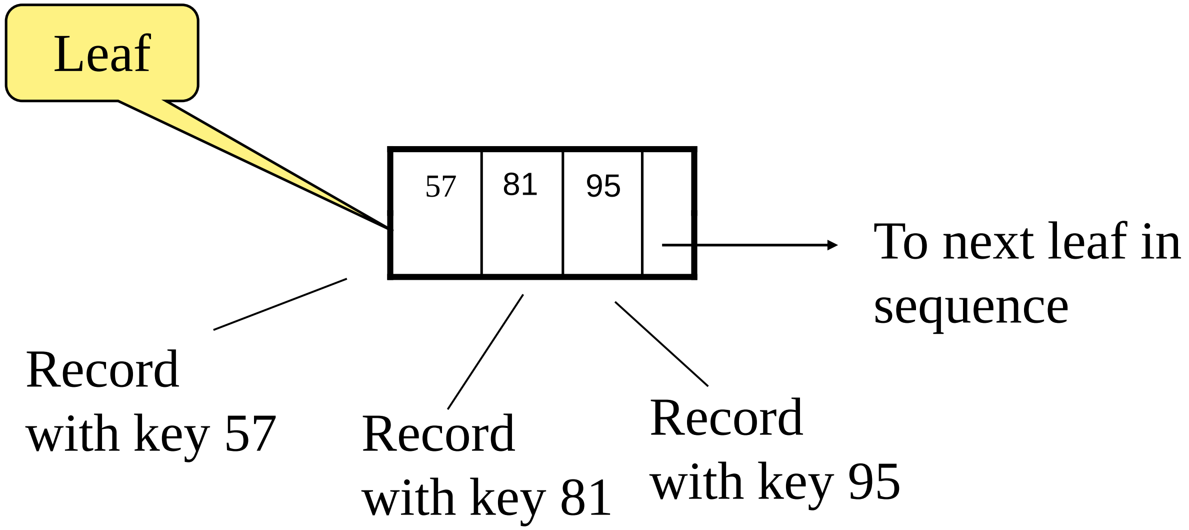
- Query: select all records where key is in range [*x*,*y*]
  - Use *x* as a search key
  - Once at the leaf: scan the data entries to find *x* or the first key that is > *x* (if *x* is not there)
  - After that, data entries are retrieved sequentially until the first record with key > *y*

B Trees: Summary

- Searching:
    - *$log_d(N)$* – Where *d* is the order, and *N* is the maximum total number of entries in all the leafs
- Insertion:
    - Find the leaf to insert into
    - If full, split the node, and adjust index accordingly
    - Similar cost as searching
- Deletion
    - Find the leaf node
    - Delete
    - May not remain half-full; must adjust the index accordingly
        - Either borrow 1 key from the sibling
        - Or merge with the sibling if there are not enough keys to borrow

B-Tree File Organization

- Store the records at the leaves
- This is called a ***clustered index*** or ***clustered file organization***
- Sorted order is maintained dynamically without overflow pages



File organizations that we know

- Heap
- Sequential
- Hash
- Clustered (B-tree with data records at the leaves)

Cost of operations for different file organizations (in disk I/Os)

N - number of data blocks
R – number of records

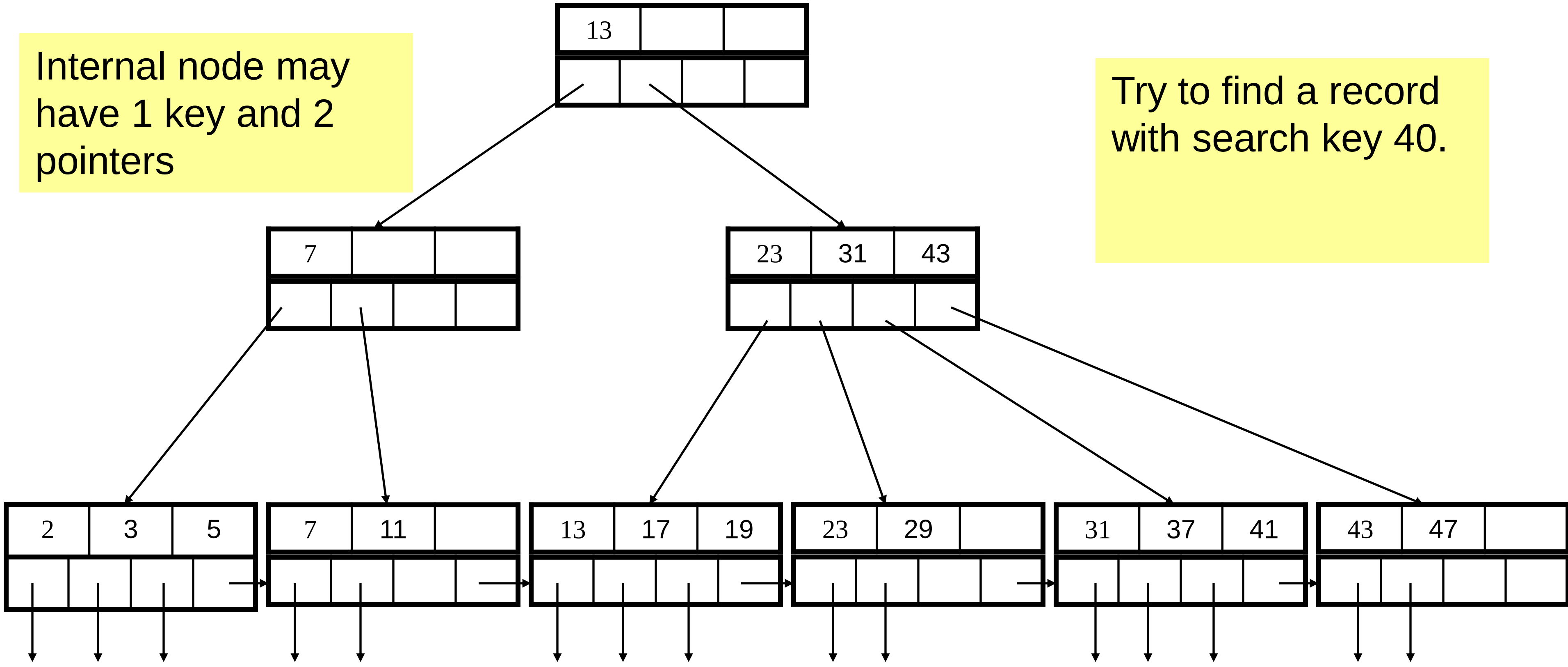
	Scan	Equality	Range	Insert	Delete
(1) Heap	N	0.5N	N	2	0.5N +1
(2) Sorted	N	$\log_2 N$	$\log_2 N$ + output	$\log_2 N + N$	$\log_2 N + N$
(3) Hashed	N	1	N	2	2
(4) Clustered	N	$\log_d R$	$\log_d R$ + num. of result pages	$\log_d R + 2$	$\log_d R + 2$

More examples of B-tree in action

Lookup

Internal node may have 1 key and 2 pointers

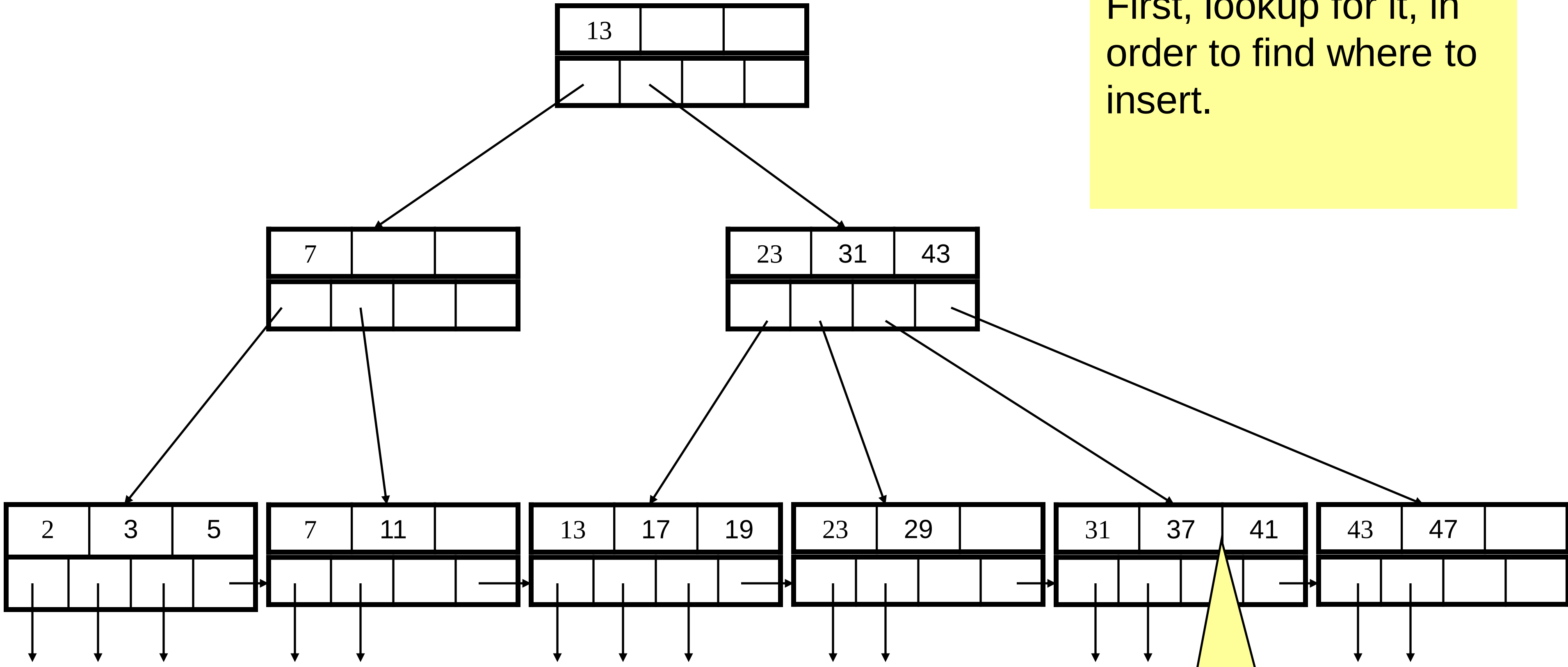
Try to find a record with search key 40.



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 contain RID of the desired record.
If we are at an internal node with keys $K_1, K_2, \ldots, K_d$, 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.

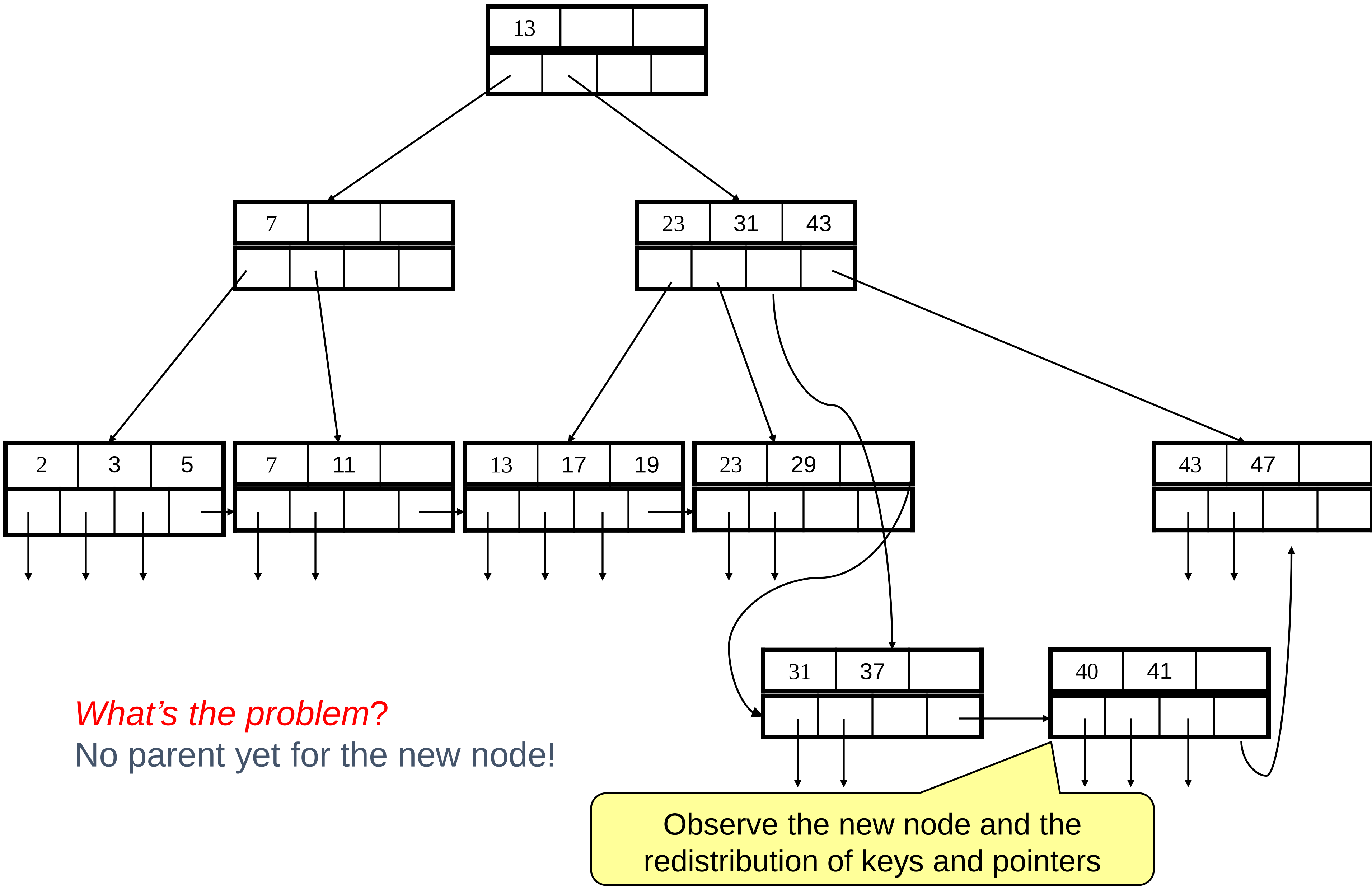
Insertion

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



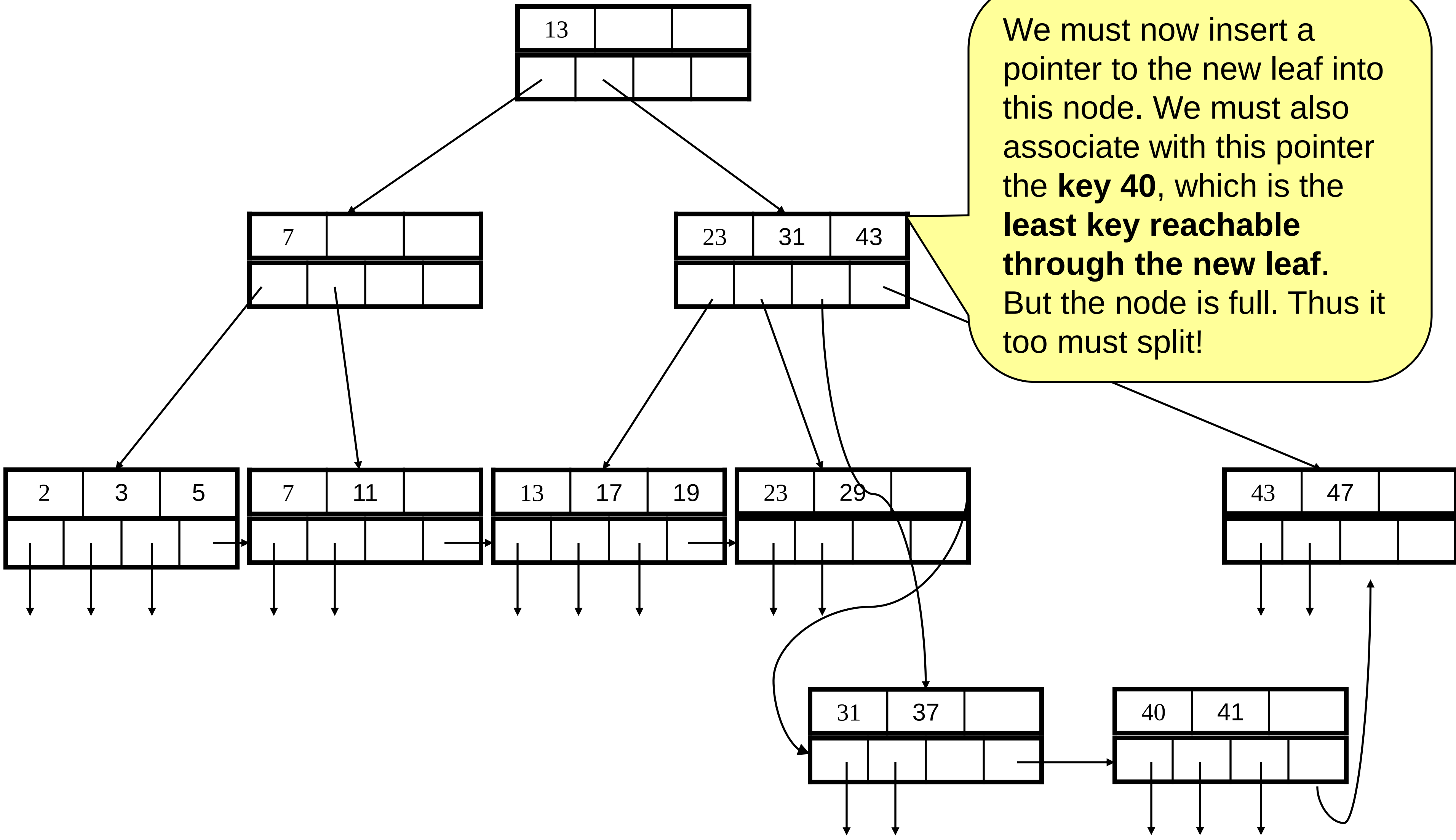
It has to go here, but the node is full!

Beginning of the insertion of key 40


13
7
23
31
43
2
3
5
7
11
13
17
19
23
29
43
47
31
37
40
41
Observe the new node and the redistribution of keys and pointers

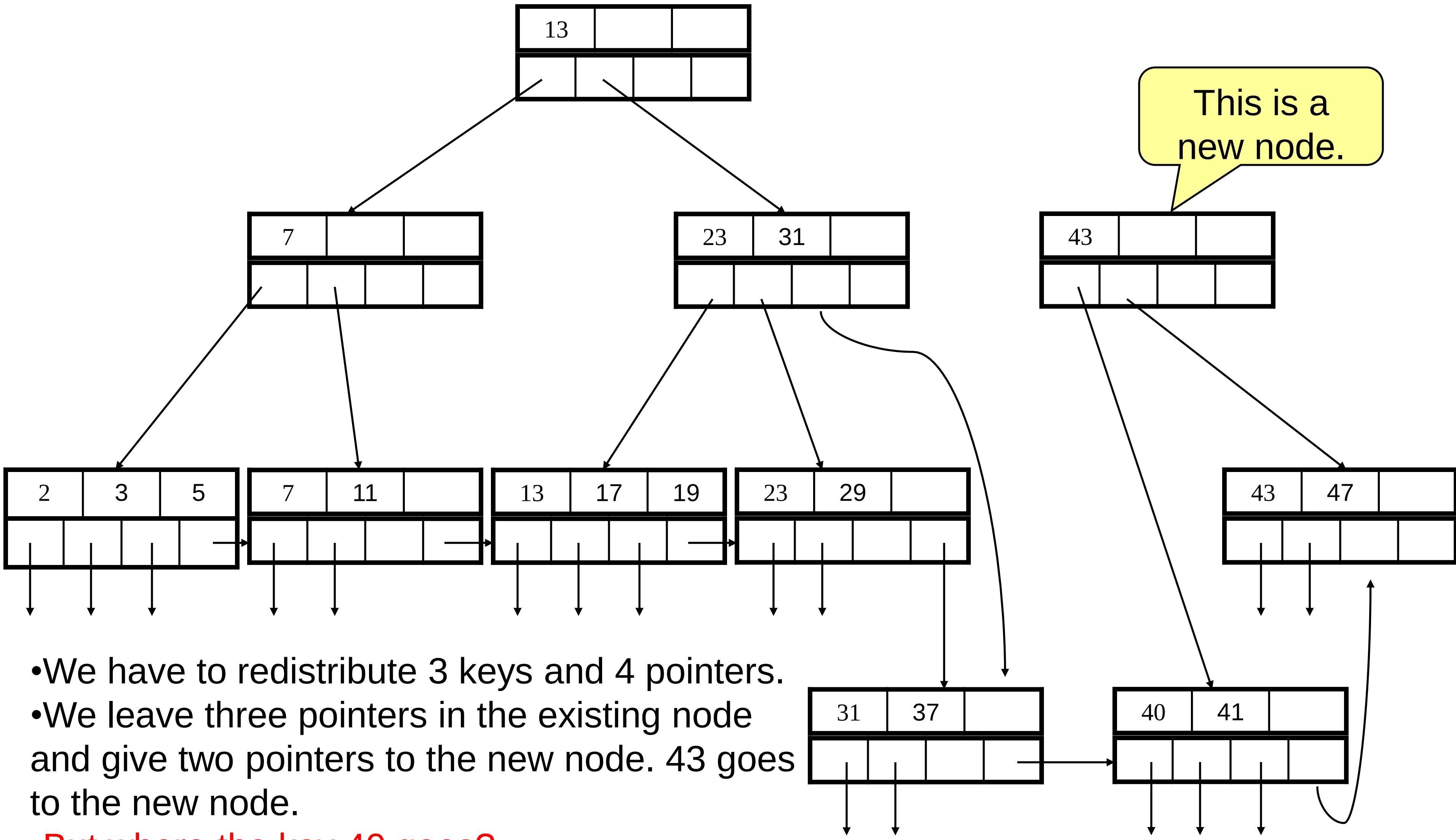
What's the problem?
No parent yet for the new node!

Continuing the Insertion of key 40



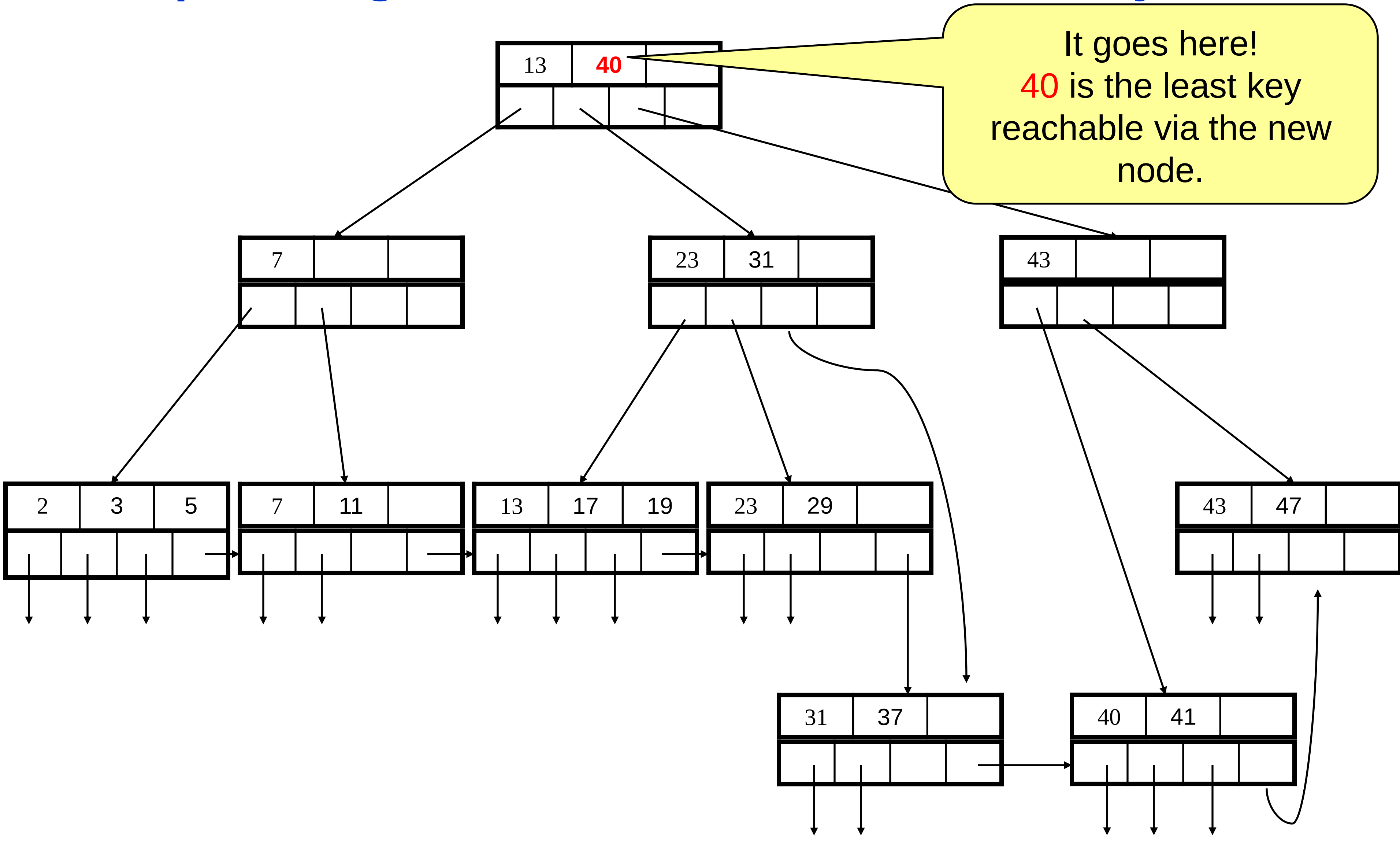
We must now insert a pointer to the new leaf into this node. We must also associate with this pointer the **key 40**, which is the **least key reachable through the new leaf**. But the node is full. Thus it too must split!

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 the key 40 goes?
•40 is the least key reachable via the new node.

Completing of the Insertion of key 40

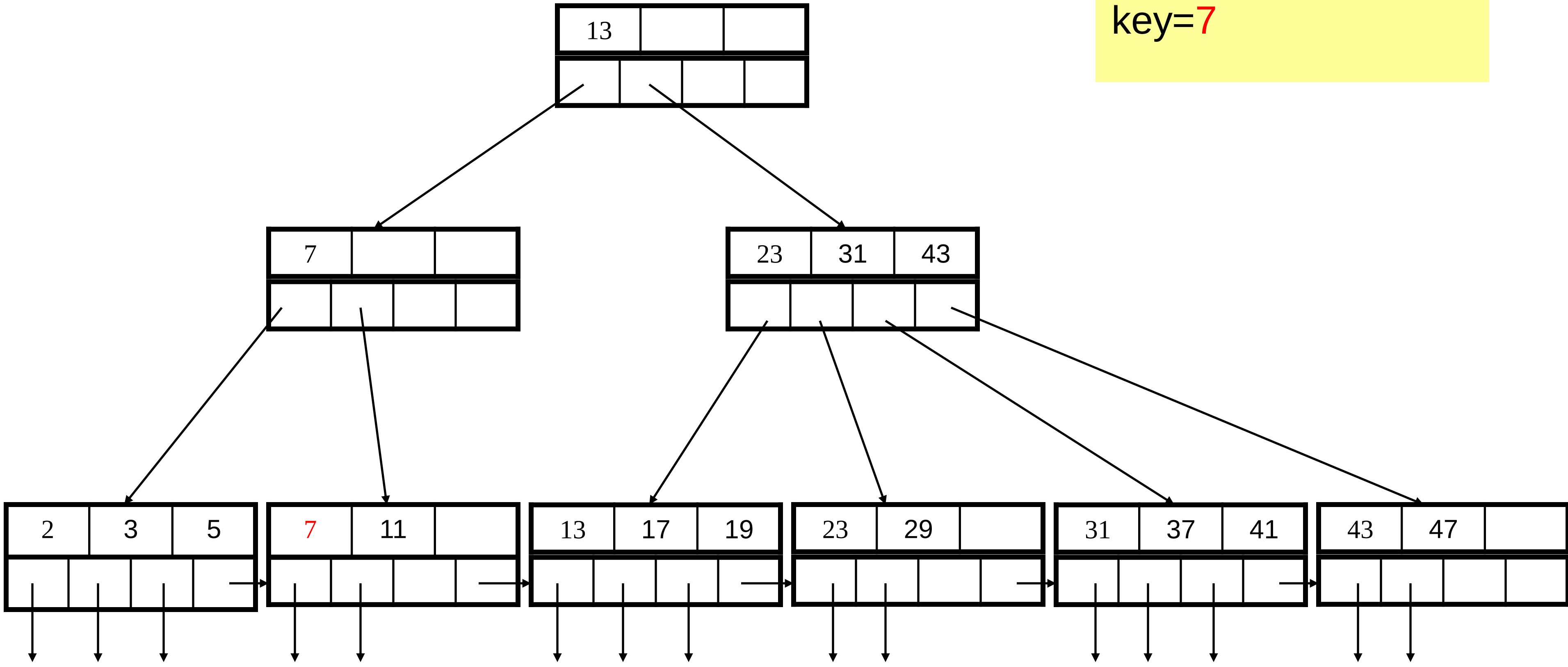


Insertion in words

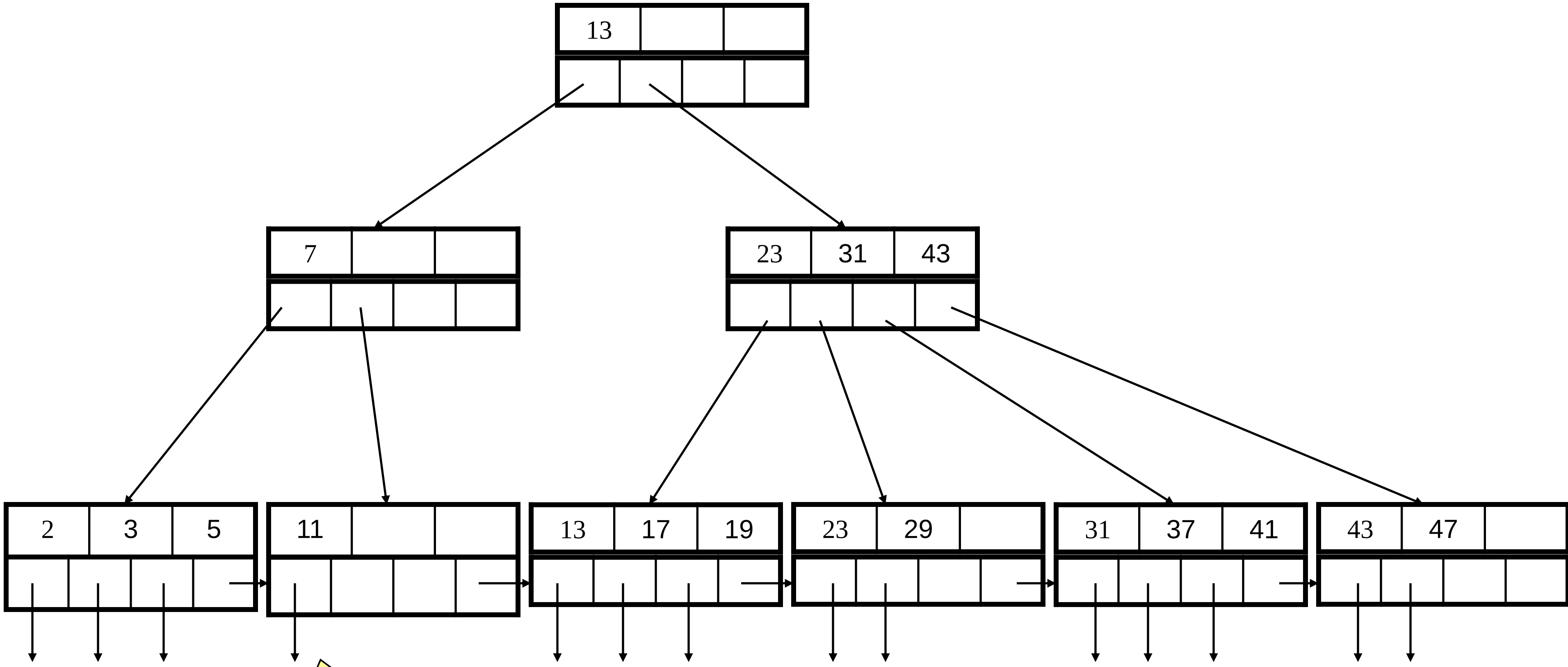
- We try to find a place for the new key in the appropriate leaf, and we put it there if there is room.
- If there is no room in the proper leaf, we "split" the leaf into two and divide the keys between the two new nodes, so each is half full or just over half full.
    - Split means "add a new block"
- 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 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;
    - The new root has the two nodes resulting from the split as its children.

Deletion

Suppose we delete key=7

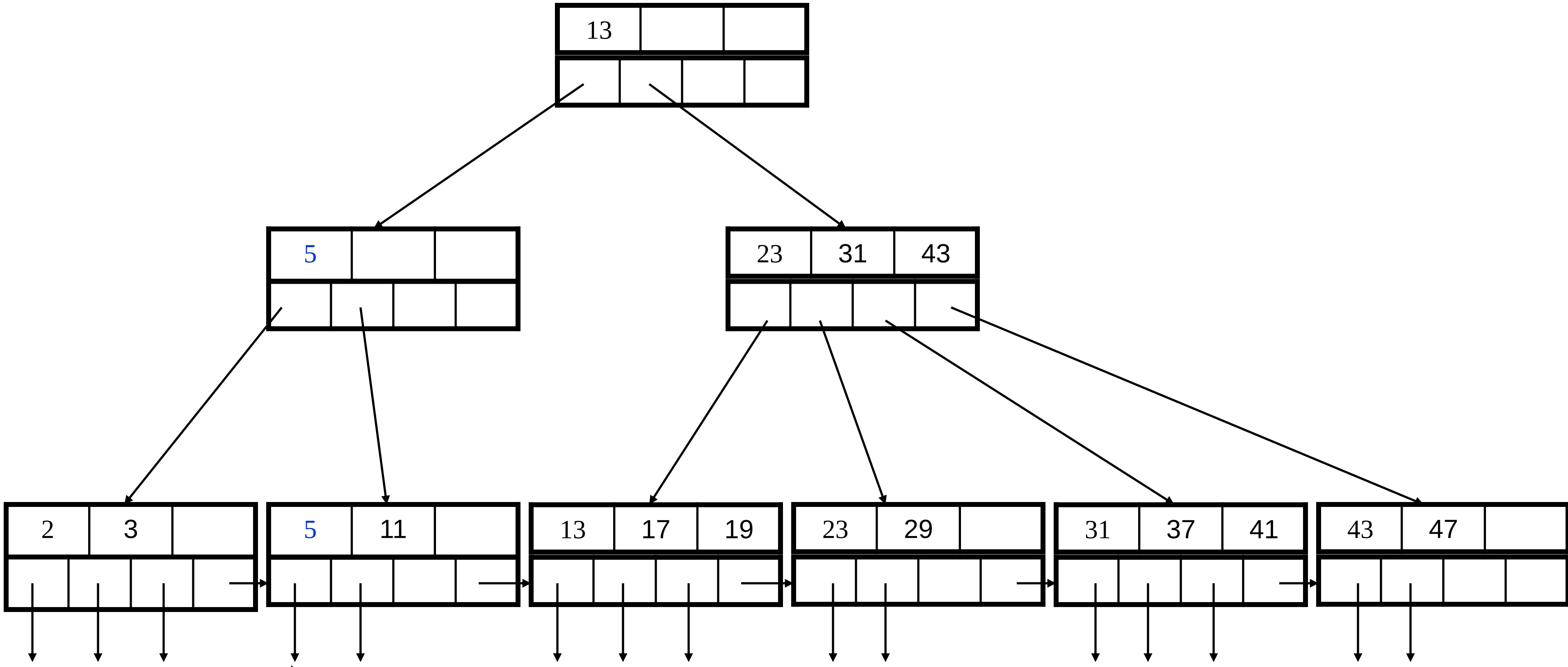


Deletion



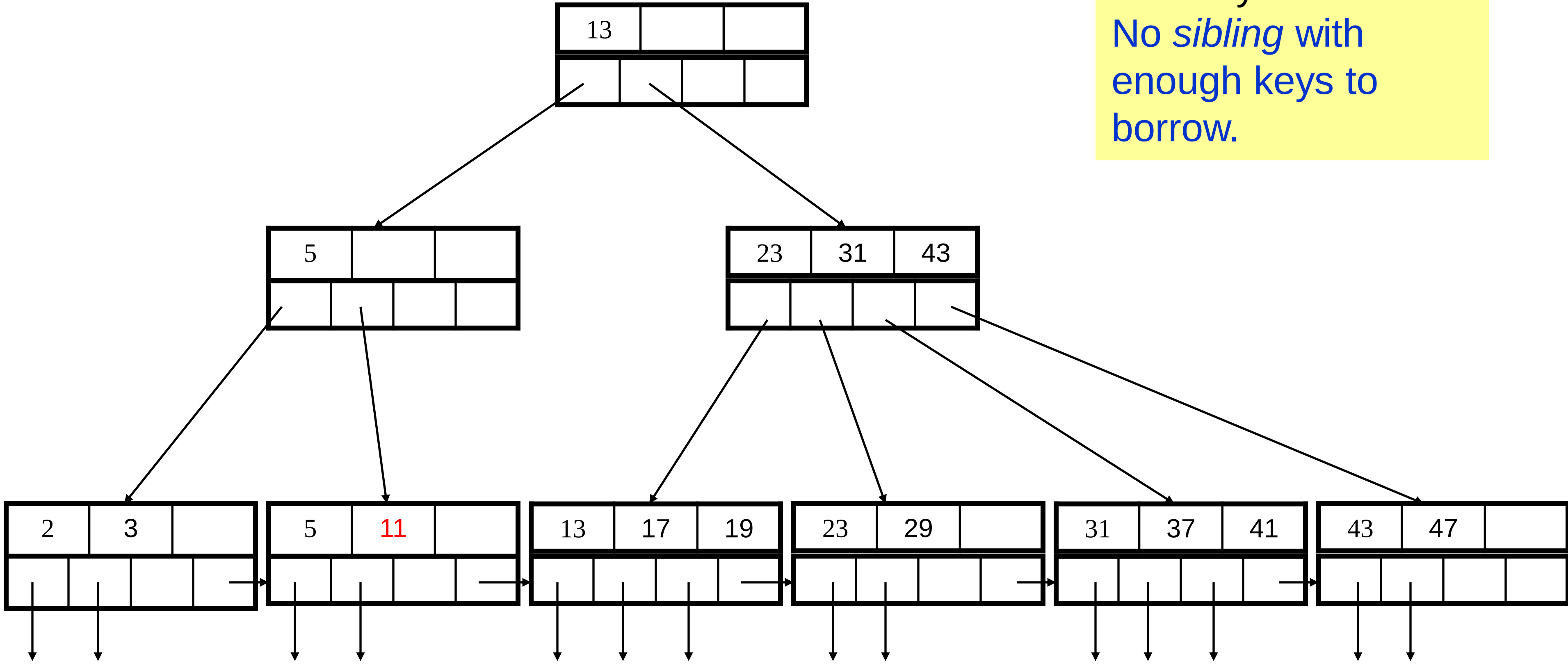
This leaf node is less than half full.

Deletion (Raising a key to parent)



This node is less than half full. So it borrows key 5 from the sibling, and updates parent node

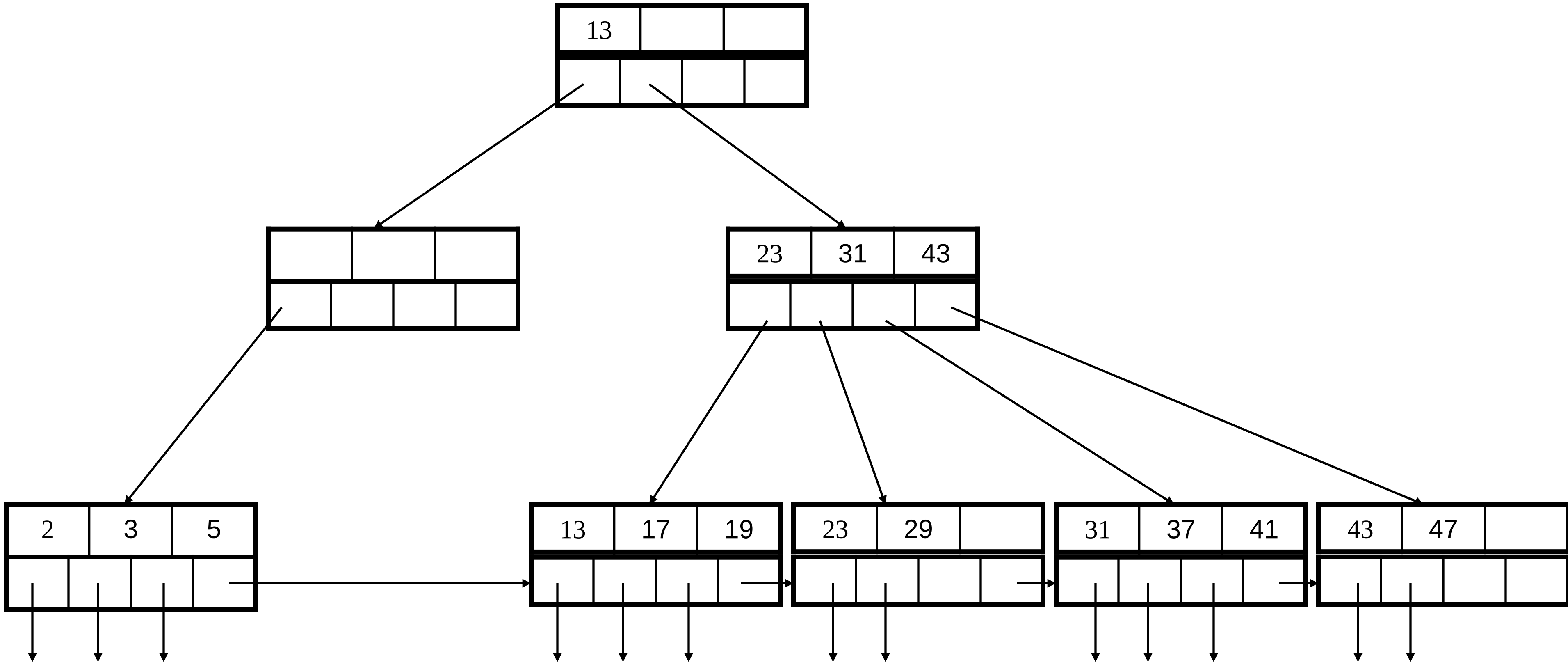
Deletion



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

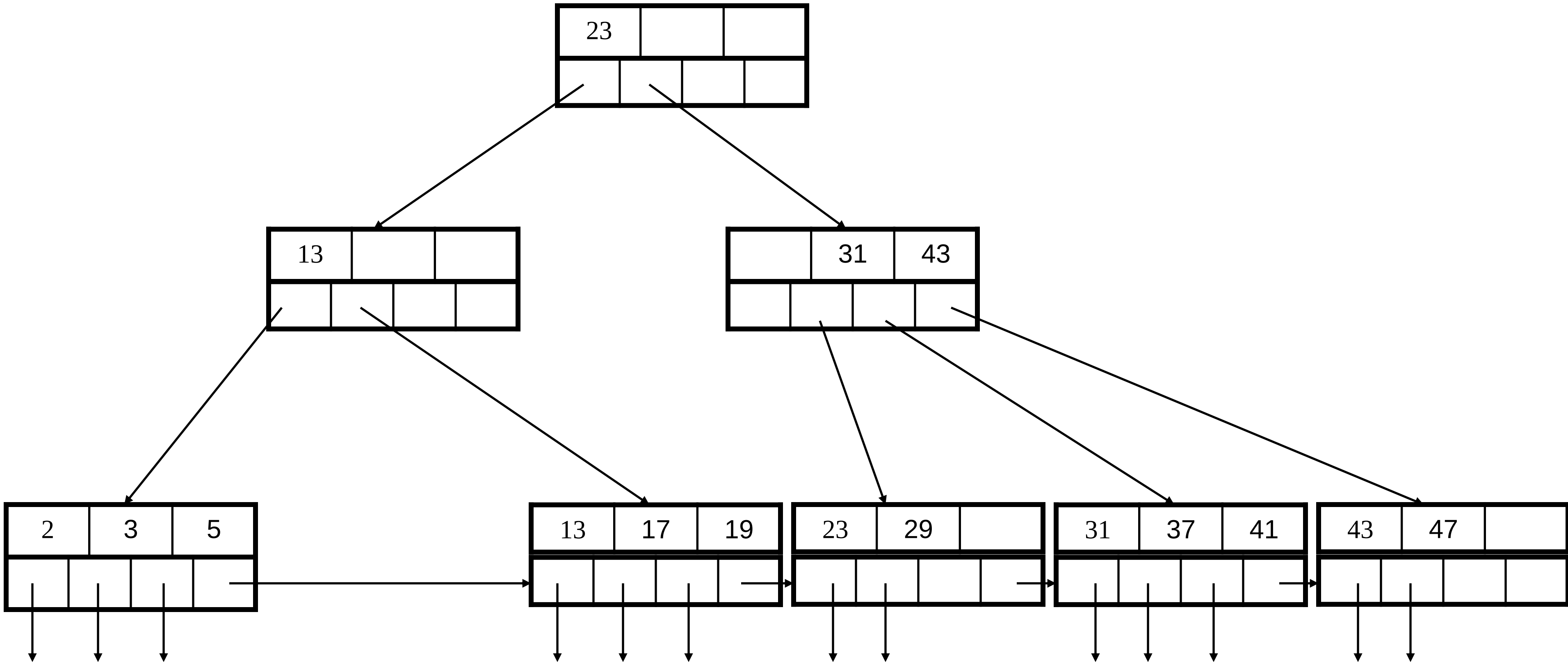
Note that node (13,17,29) is not a sibling – because it has a different parent

Deletion



We merge, i.e. delete a block from the index. However, the parent ends up having 1 pointer and zero keys

Deletion



Parent: Borrow pointer from sibling!

Deletion in words

- We find a place of the deleted key in the appropriate leaf, and remove the corresponding entry
- If the leaf node was at a minimum capacity before the deletion, it is now below minimum
  - If its most populous sibling contains more than d/2 children – borrow one and update parent pointer
  - Else if there are no nodes to borrow – merge current node with its sibling
- Update parent pointer. If there are less than d/2 children – borrow key from right sibling

Exercise (at home)

- First, build B-tree from records with sorted keys:

2,3,5,7,11,13,17,19,23,29,31,37,41,43,47

- Insert key 1
- Insert keys 14,15,16
- Delete key 23
- Delete all keys >23 (in turn)