In these notes, we study a family of data structures called B-trees, in particular, the variant called B^+-trees.

Structure of B-trees:

- A B-tree organizes its blocks into a tree that is balanced, meaning all paths from the root to any leaf have the same length.
- There are 3 layers in a B-tree
  a) root, b) intermediate layer, and c) leaf nodes.
- A parameter n is associated with each B-tree.
  1. Each block has space for n key-values and $n+1$ pointers (disk pointers/addrs, or db addresses).
  2. n is typically large, also called a "fanout" index. E.g. say a disk block is 4KB and

a key is 8 bytes and pointer is also 8 bytes. Assume that the block header is 128 bytes. The largest value of n would be:

$$8n + 8(n+1) + 128 \leq 4096$$

$$\text{or} \quad n \leq \left\lfloor \frac{3960}{16} \right\rfloor = 246.$$

B-tree structure and properties:

1. The keys in leaf node are copies of the keys from the data file. The keys in each leaf node are in sorted order, ASC (left → right)

2. The leaf nodes are organized in sorted order by the keys.

3. At the root, there are at least 2 pointers (except when the file has only one record).

4. The leaf node has exactly n keys, and with each key, an associated pointer (total n ptrs)

that points to a designated record in the data file with this key value.

A B^+-tree leaf node: $n = 3$

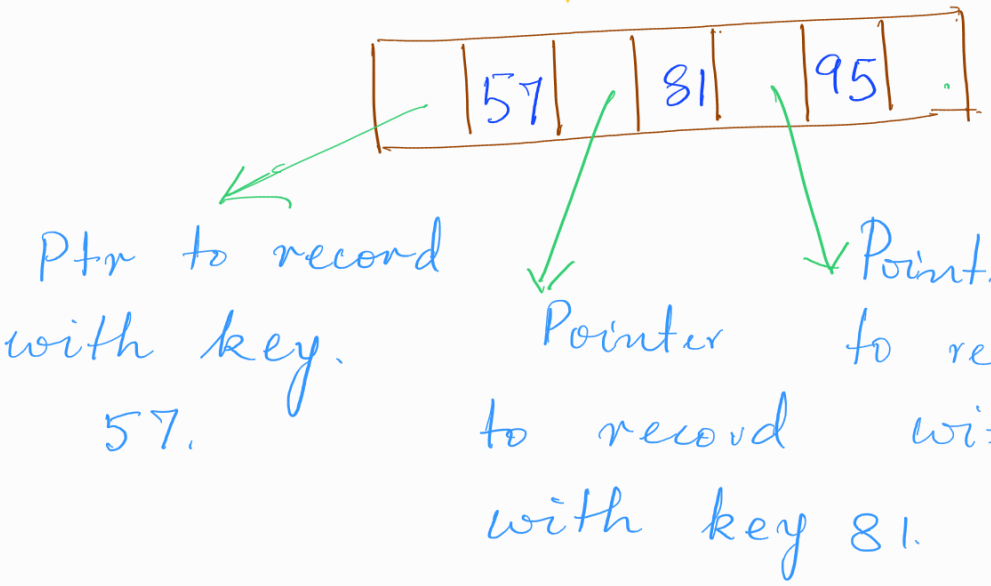


Fig: A B^+-tree leaf node. $n = 3$.

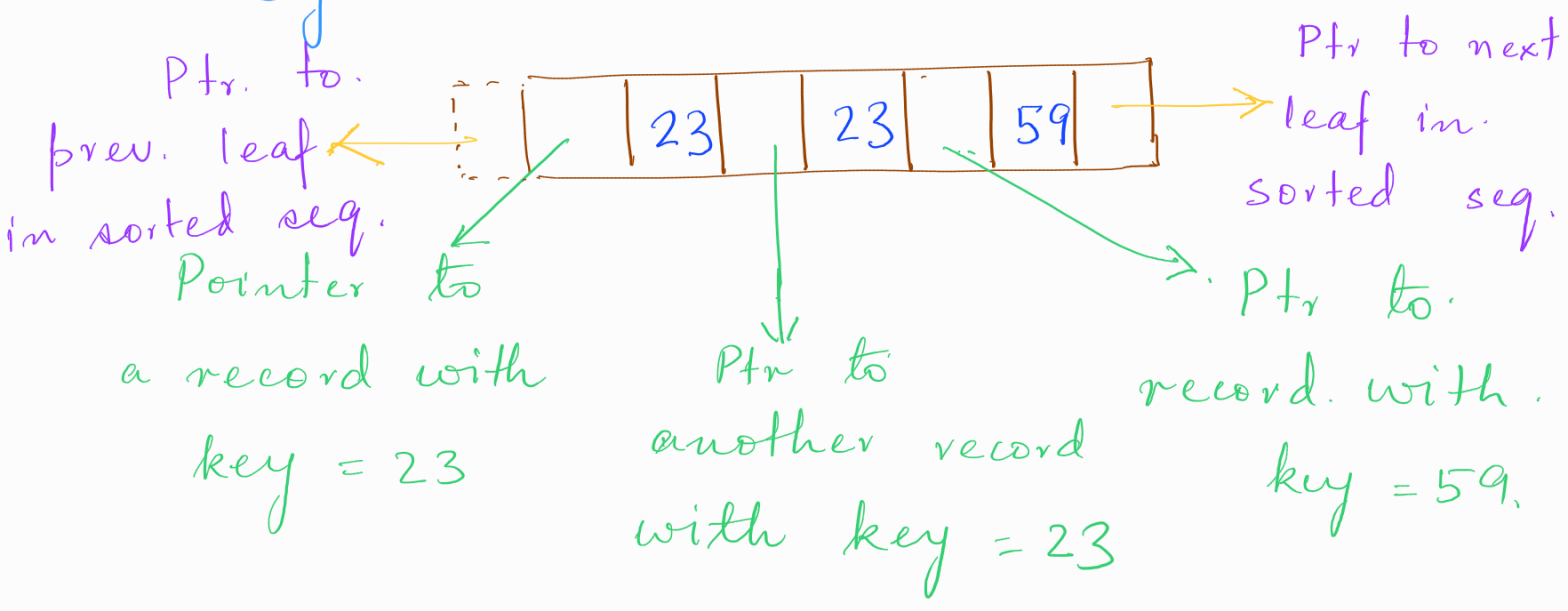
There are 3 (key, pointer) pairs. Each pointer points to a record with that key.

Note: The search key attribute used by the B-tree need not be a primary (or candidate) key of the data file (table). There can be multiple records with the same key value in the data file.

The B-tree leaf nodes have exactly the same number of (key,

pointer) pairs, for every distinct key value, the pointers point to each of the distinct records in the data file with the same key value.

E.g.



The example also shows some "structural" pointers: a pointer to the leaf's immediate right leaf neighbor (in sorted order) and pointer to its immediate left leaf neighbor

Note: Important: B^+-trees manage occupancy of blocks with keys and pointers so that each block is at least half-full. A leaf node in a B^+-tree with para-

meter n has upto n (key, ptr) pairs corresponding to the data file, and, has at least $\lfloor (n+1)/2 \rfloor$ key, pointer pairs corresp. to the data file.

[This does not include structural pointers, left leaf ptr, right leaf ptr, also parent node ptr, etc.].

To repeat, a leaf node has between $\lfloor (n+1)/2 \rfloor$ and n (key, ptr) pairs.

3. An interior node can have upto $n+1$ pointers to its children nodes.

An internal node; B^+-tree $n = 3$.

To all keys K in this subtree $K < 14$.

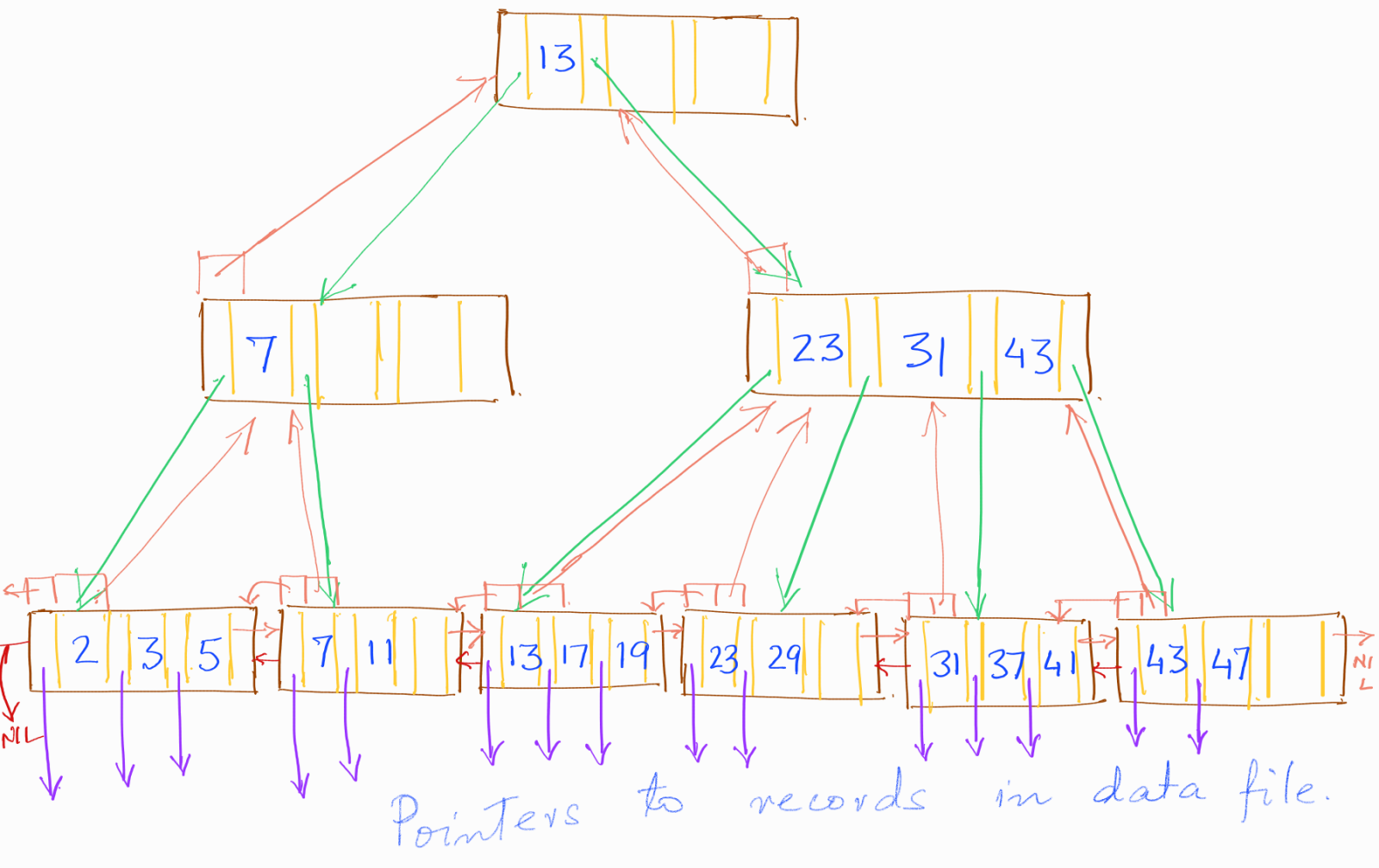
To all keys in this subtree $14 \le K < 52$.

To all keys. $52 \le K < 78$

To all keys. ≥ 78 in this subtree

1. Internal nodes (normally) do not store pointer to its right/left neighbor at the same level. Tree structure is maintained, so "structural pointer" to its parent is maintained to allow efficient navigation from a node to its parent or to any child.

Important: Internal nodes have between $\left\lceil \frac{n+1}{2} \right\rceil$ pointers and $(n+1)$ pointers. (With some exceptions), an internal node has 1 less key than the number of pointers stored for its children. (See ex. above).


13
7
23 31 43
2 3 5
7 11
13 17 19
23 29
31 37 41
43 47
NIL
NIL
Pointers to records in data file.

This shows an example of a B-tree, with $n = 3$, and has 3 levels.

At the leaves, keys appear in order.

1. All leaf blocks have either 2 or 3 key, ptr pairs.

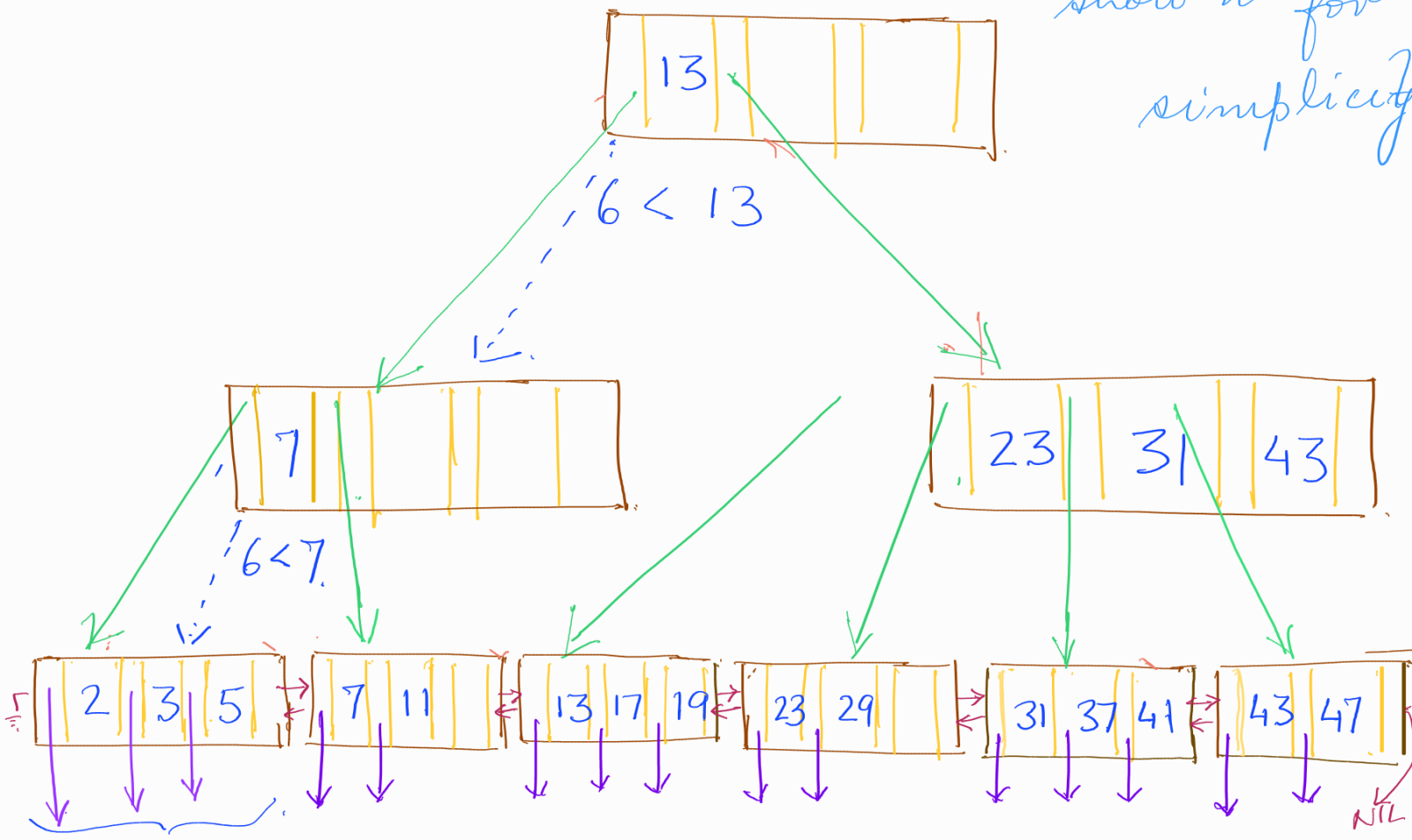
2. Root node has only 2 ptrs, its minimum. The key value 13 is the smallest key in its right child.

3. At root node: to search for keys, if key < 13, we take the left child path and if key ≥ 13, we go down the right child path.

For simplicity, assume that for our examples, the keys are integer values (primes in this example).

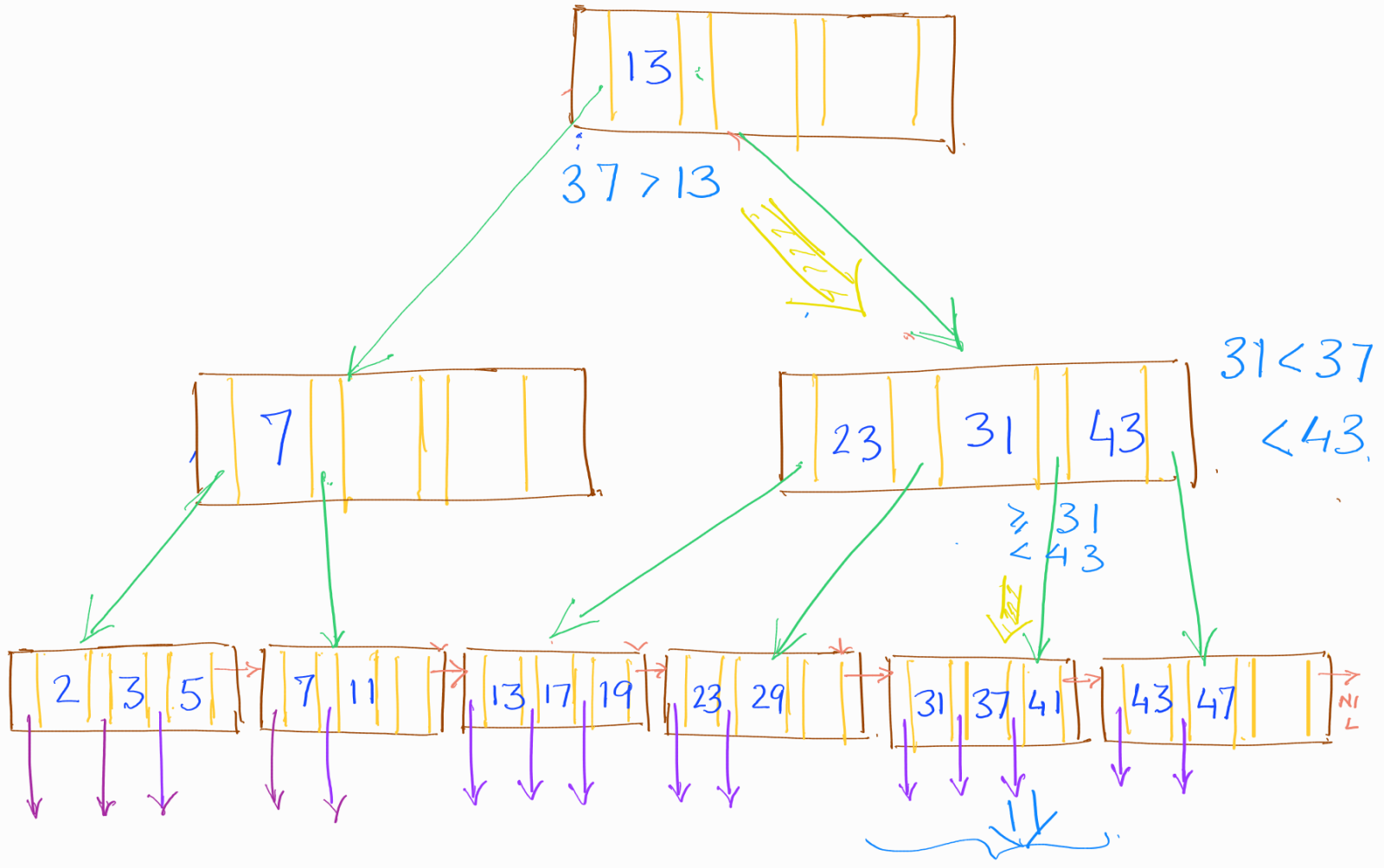
So for keys ≤ 12, at root, we follow the left child path, and for keys ≥ 13, we follow right child path.

E.g. Search for 6 [B^+-tree structural pointers are not shown for simplicity]



search for 6: not found.

E.g. Search for 37.



Search for 37. in leaf. Follow pointer.

Insertion into B-trees.

The main steps are as follows. Suppose we wish to insert a key value K along with a pointer P to a record in the data file.

1. First search for key value

K in the B-tree. [We may find the value; this might be another record being inserted with the same key value.]. The search may end negatively (i.e., key not found) at a leaf node, or may end positively, again at some leaf node. We will now try to insert the (K, P) pair in this leaf node.

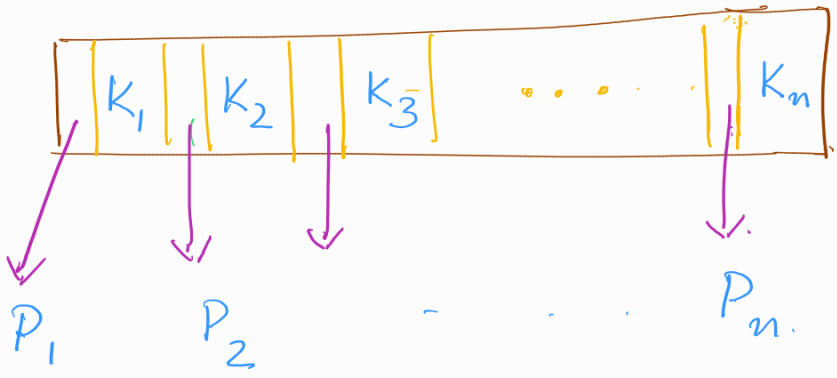
2. There are two possibilities.

1) The leaf node is not at full occupancy, that is, the number of keys in this leaf node is $< n$.

2) The leaf node is full.

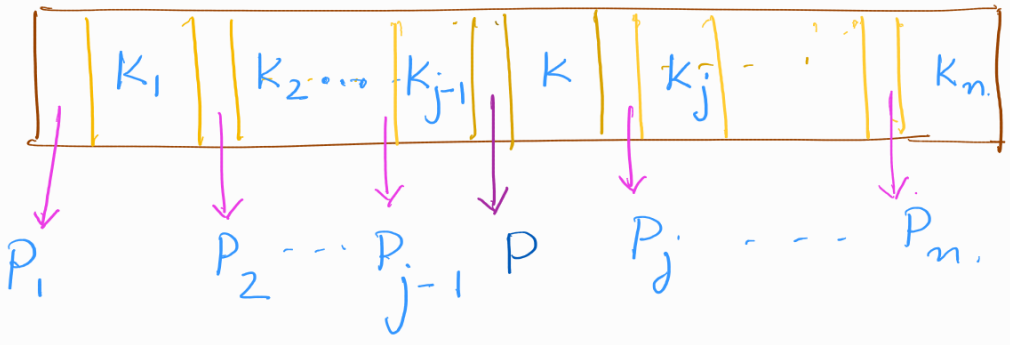
3. The first case is simple. The (K, P) pair is inserted into this leaf node in its proper sorted position.

4. The second case is when the leaf node has n key, ptr pairs and is full.



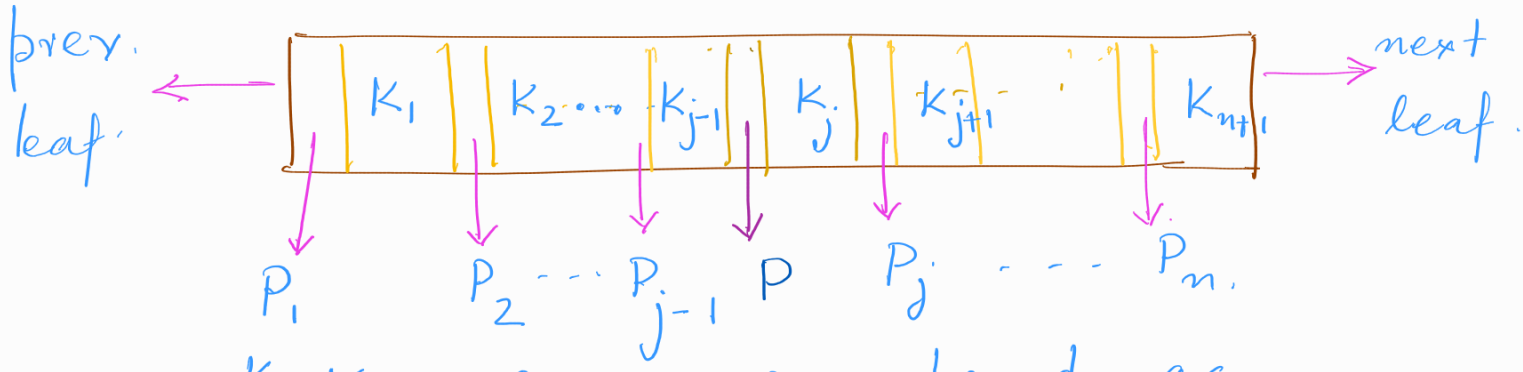
Current leaf node

Temporarily, we expand its room (imaginary) and insert (K, P) into its correct position among the leaf keys $K_1, K_2, \cdots, K_n$

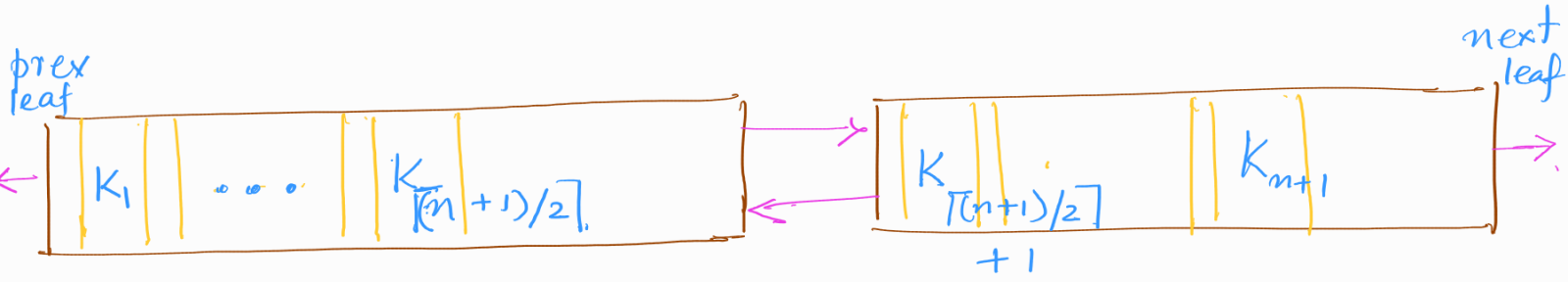


Temporary node: n+1 keys in sorted order with (K, P) inserted in the correct position.

SPLIT this node: Now split this node into 2 parts: as equal as possible.



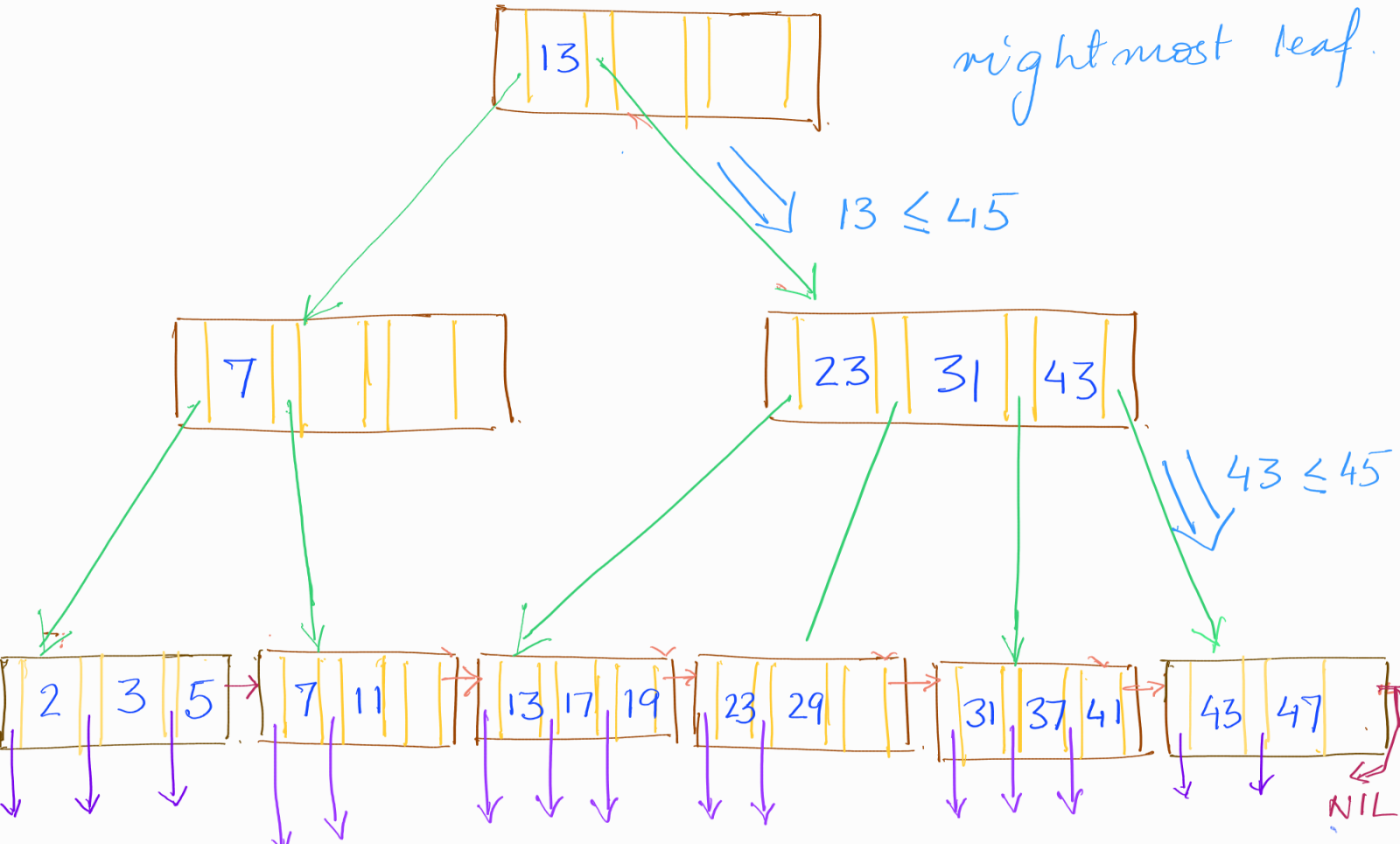
Keys are renumbered as $K_1, K_2, \ldots, K_{n+1}$. Identity of K is not crucial. Partition this leaf evenly.



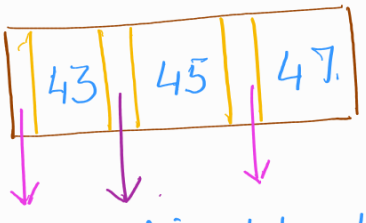
The left leaf partition has $\lceil \frac{n+1}{2} \rceil$ keys arranged in order $K_1, \ldots, K_{\lceil (n+1)/2 \rceil}$. The right leaf partition has $\lfloor \frac{n+1}{2} \rfloor$ keys.

We now insert the pair, $\left(K_{\lceil \frac{n+1}{2} \rceil}, \text{addr of right partition.} \right)$ into the parent of the original leaf. Assume for simplicity, that all keys are distinct.

Suppose we wish to insert 45. Search path leads us to rightmost leaf.



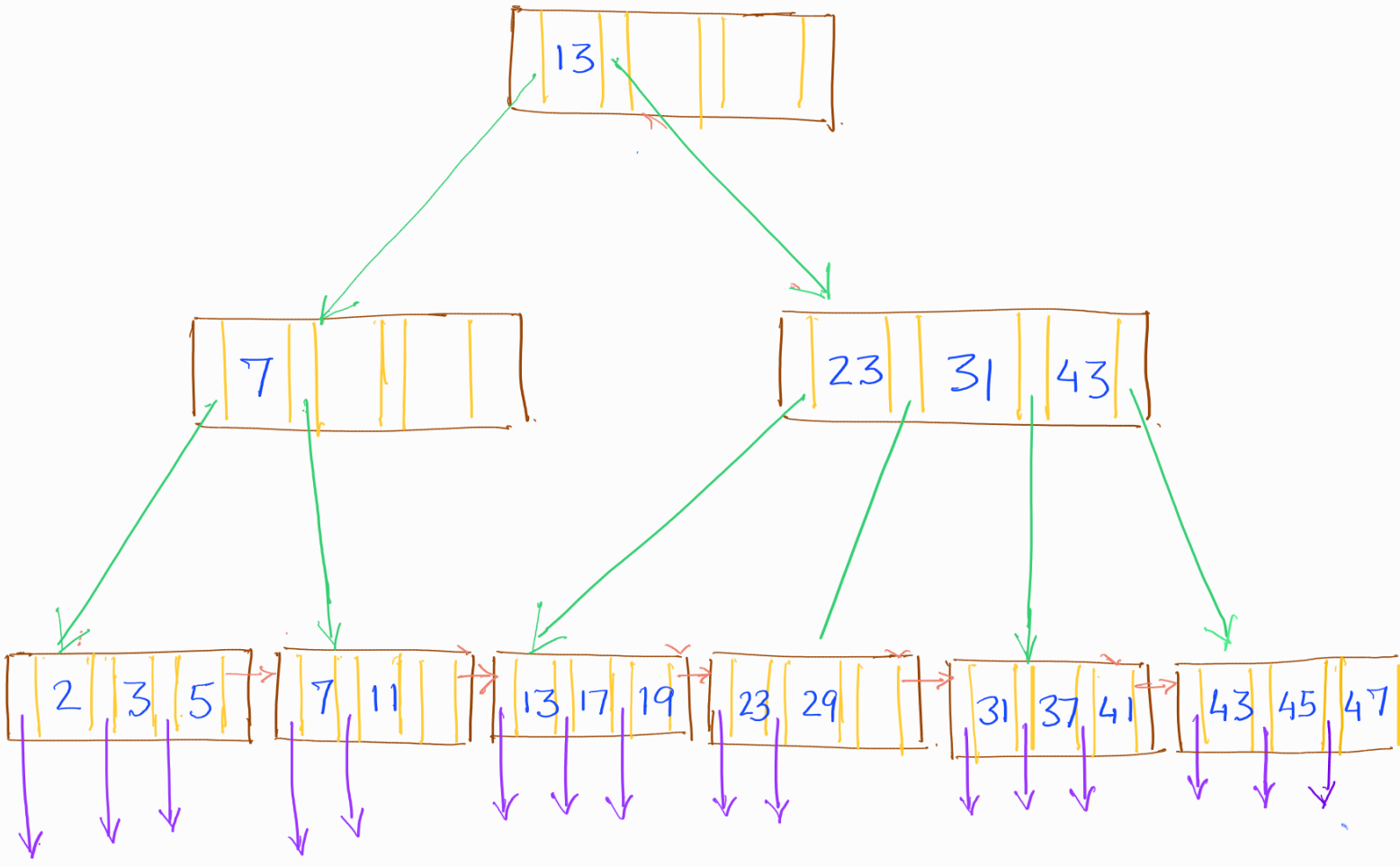
There is sufficient space in the leaf block.



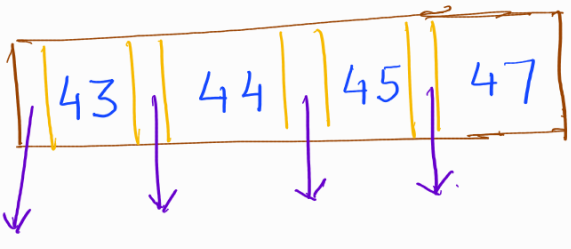
New leaf block.

(45, ↓) pair is inserted.

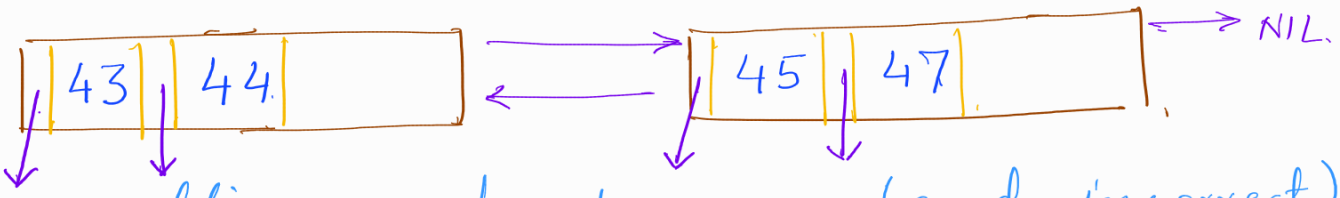
B-tree after inserting 45.



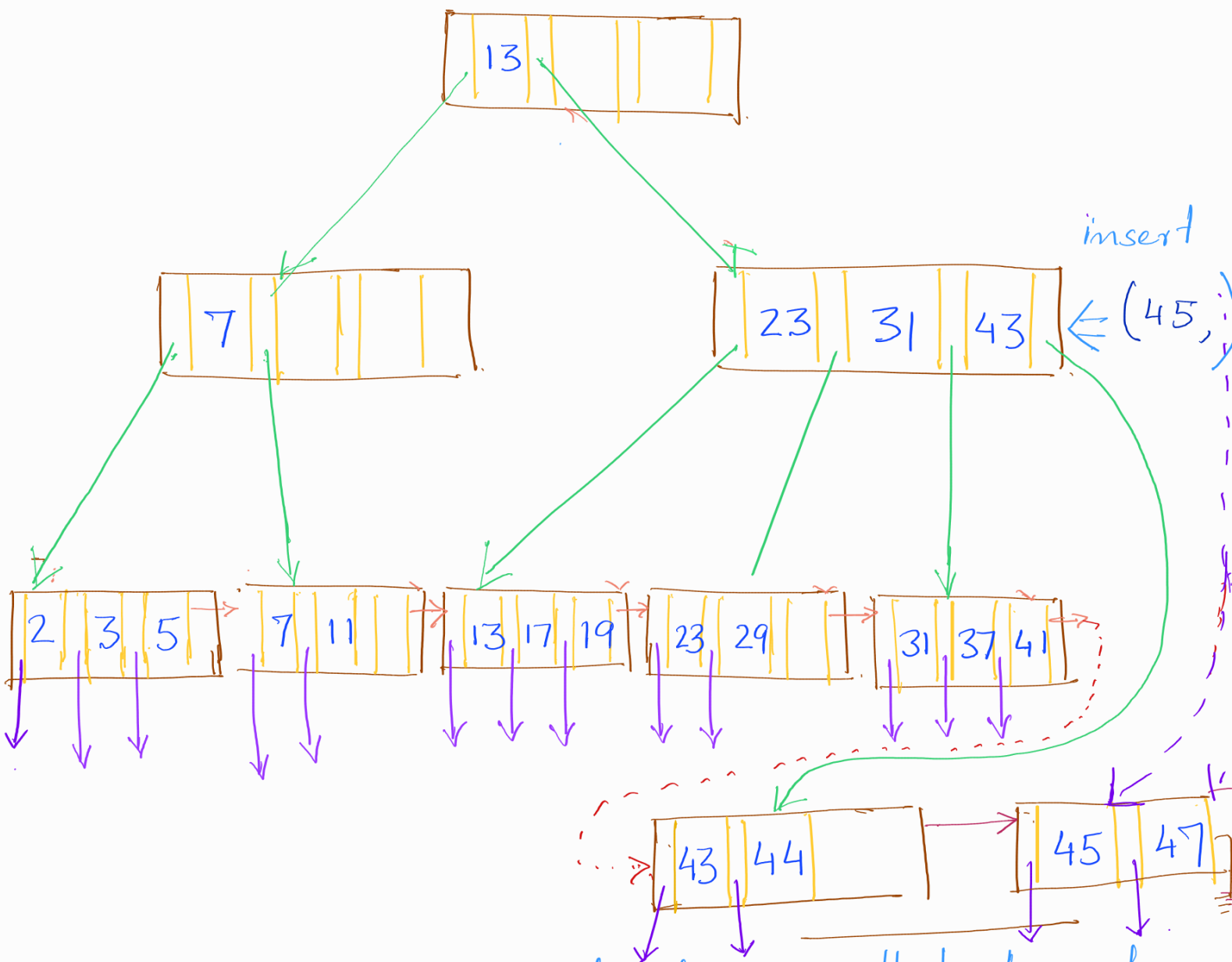
Now suppose we insert 44. Following search procedure, we follow right child ptrs. $13 \leq 44$, $43 \leq 44$, to again arrive at the right most leaf. However, the leaf node is full. Temporarily, we insert the key 44 along with its pointer into the leaf:



This is split into 2 leaf blocks.

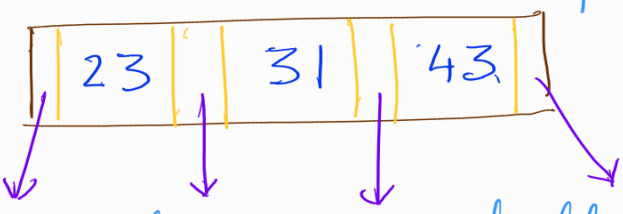


The resulting temporary (and incorrect) structure is shown here.



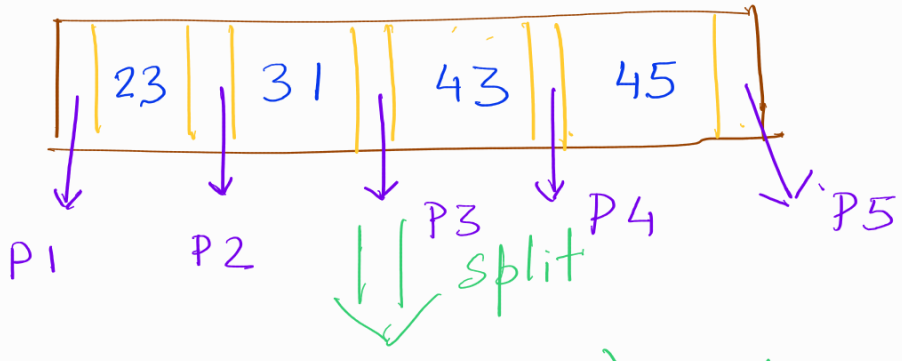
We have to insert the smallest key of the right partition. 45 together with pointer to this right partition leaf block into its parent

The parent has 4 pointers $(4 = 3+1 = n+1)$.



This is already full. We recursively attempt to insert the pair (45,).

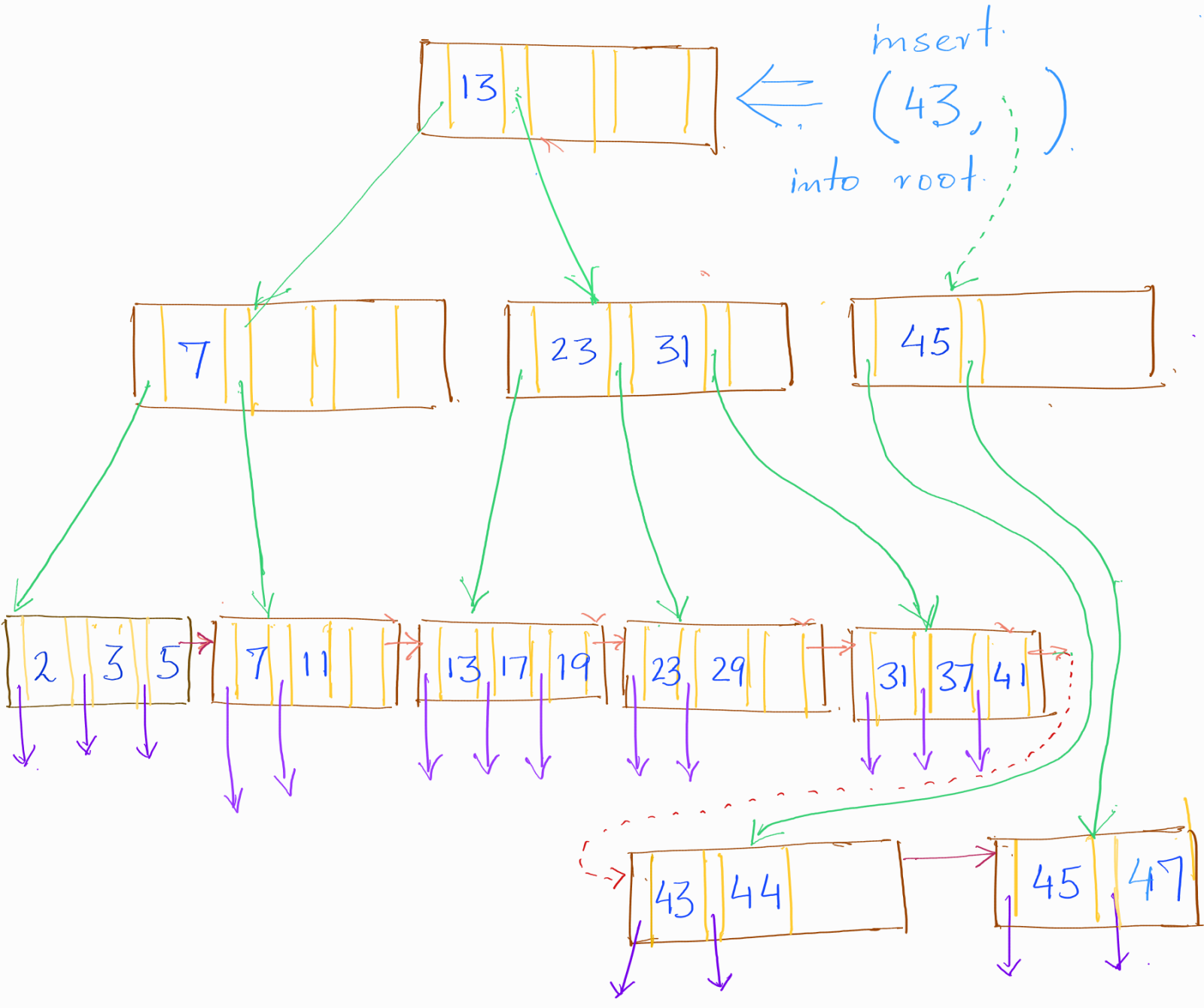
Temporarily, we create an artificial internal node structure:



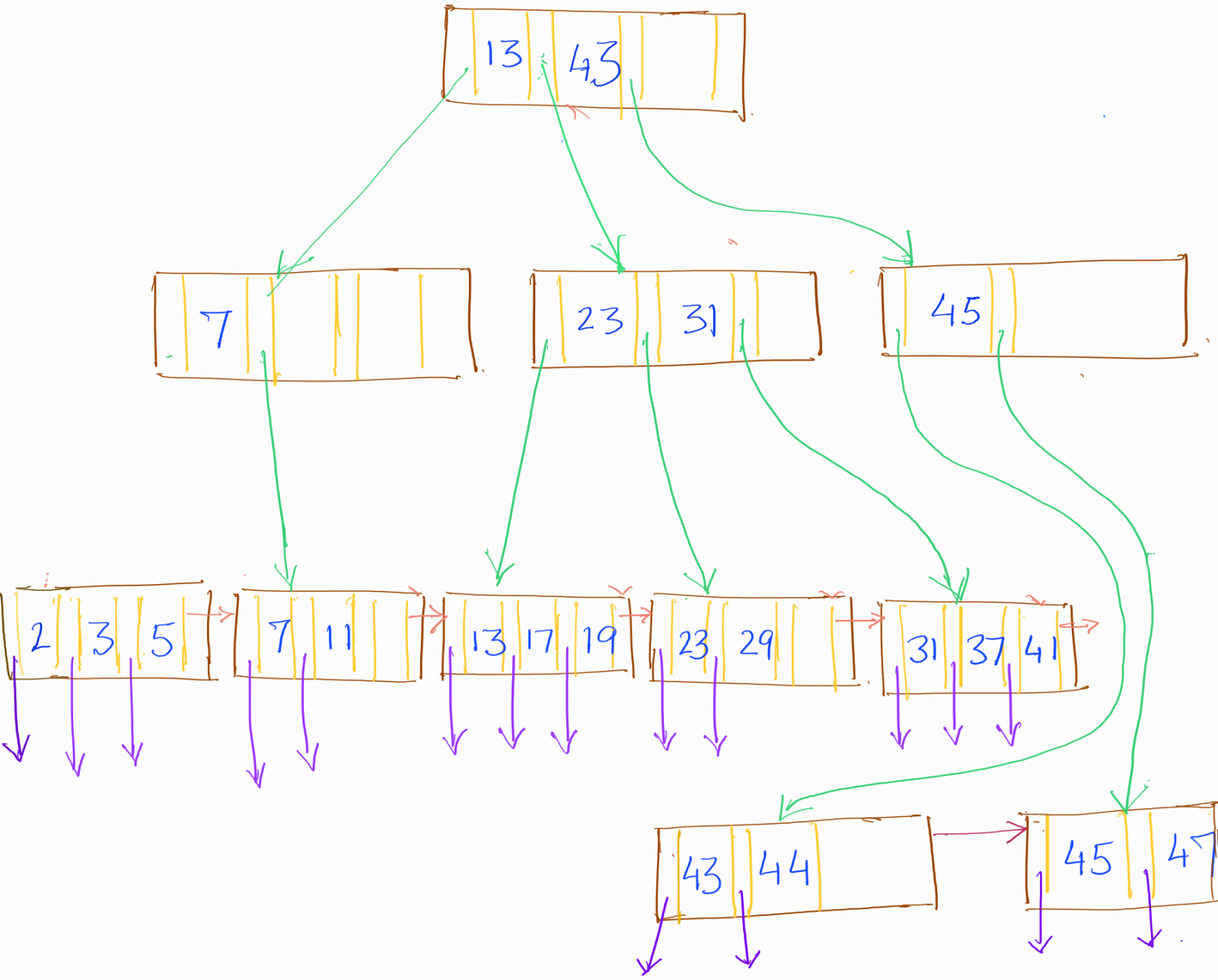
insert (43, P3) into parent.

Note that an internal node with $n+1$ pointers and 1 additional key, ptr pair gets $n+2$ pointers. The first $\lceil \frac{n+2}{2} \rceil$ pointers and its corresp. $\lceil \frac{n+2}{2} \rceil - 1$ keys are placed in the left partition. The right partition keeps the rightmost

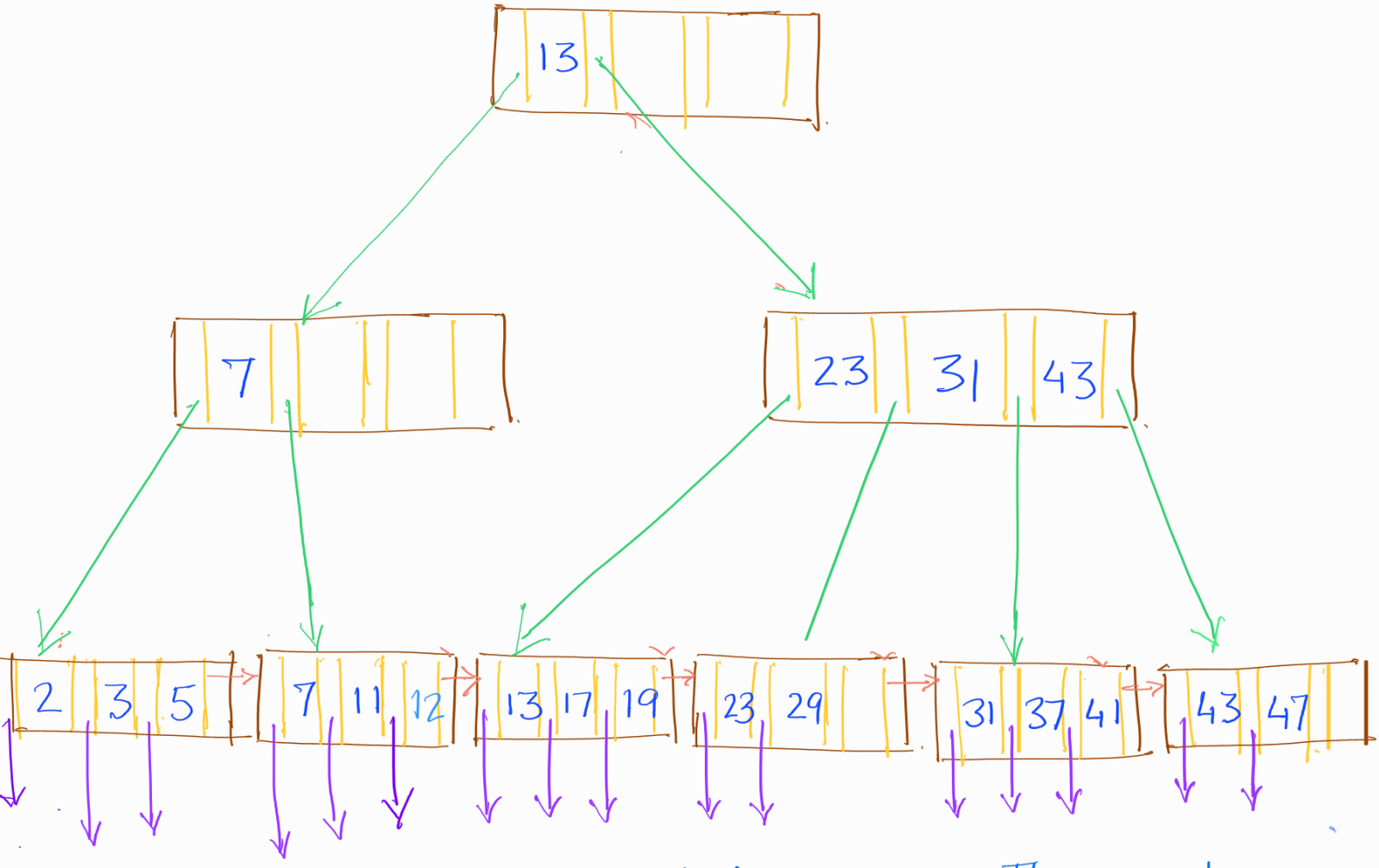
$n+2 - \left\lceil \frac{n+2}{2} \right\rceil - 1$ pointers and the corresponding right most. corresp. keys. The median (middle) key together with its ptr, here it is $(43, \downarrow)$ is propagated up to its parent for insertion.



Insertion into root is direct, as it is not full. So (43,) is inserted into the root. The final B-tree is:



Deletion of a key, pointer pair from a B-tree. Suppose we start from a B-tree similar to the one prior to our insertions.



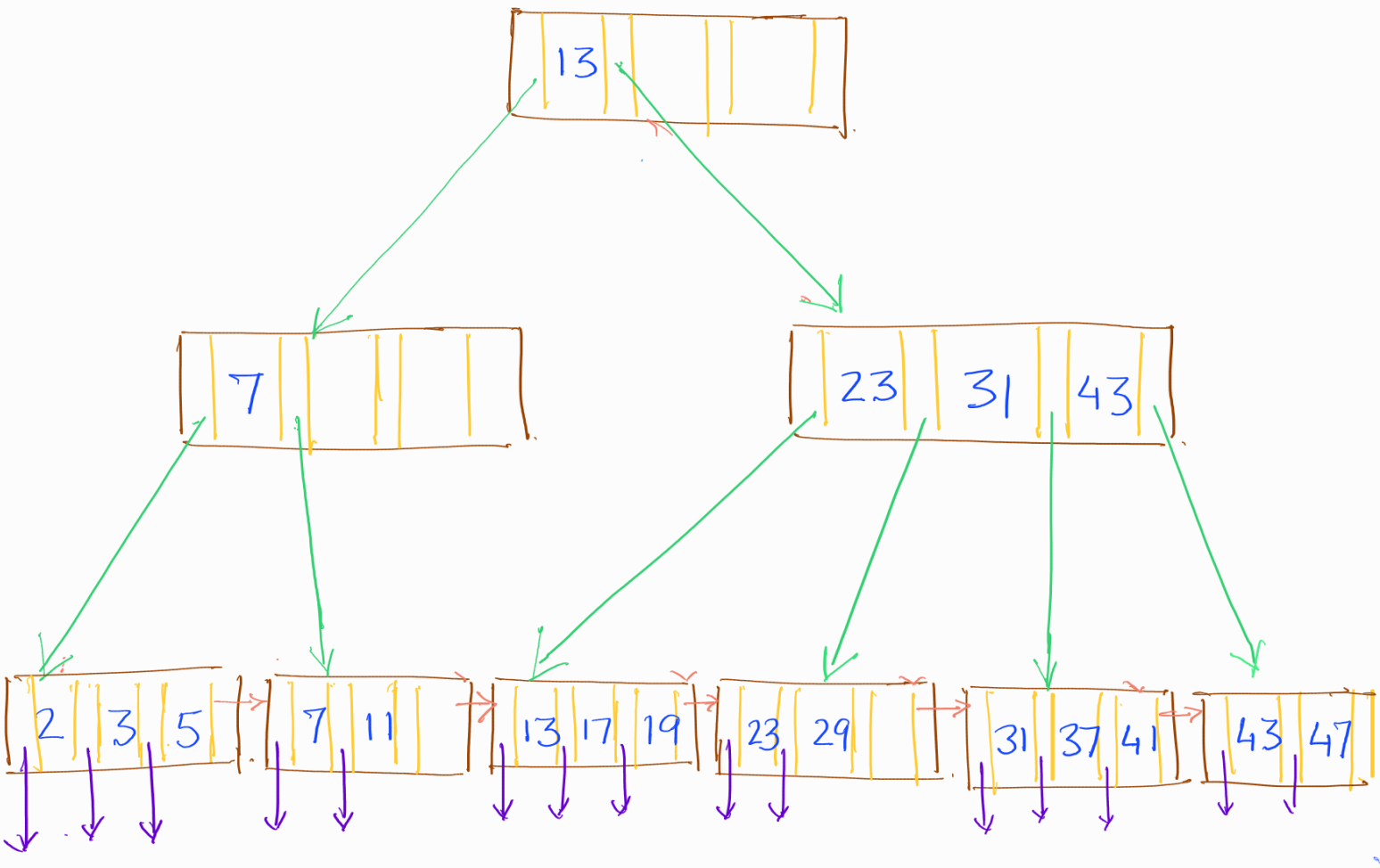
Suppose we delete 12. Then by searching for 12, we are led to the second leaf from the left. Deleting 12 and pointer to the record is possible, since after deletion, there

remain 2 (key, ptr.) pairs in that leaf, which is within limits.

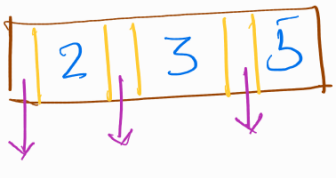
$n = 3$, leaf nodes have.

$\lfloor \frac{n+1}{2} \rfloor$ ... n key, ptr. pairs.

The new B-tree is:



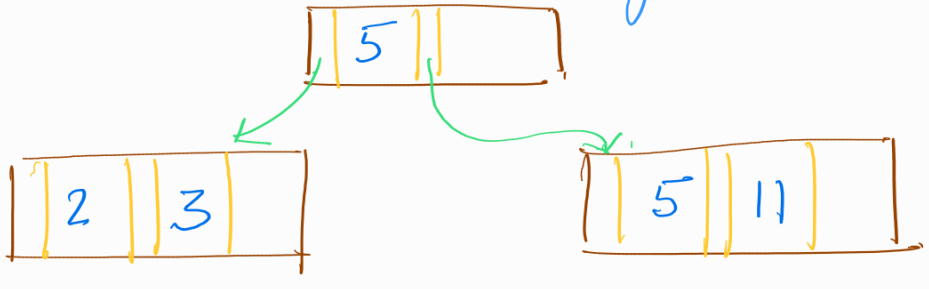
Now suppose we delete 7. The leaf node containing 7 is at minimum occupancy. Upon deletion, we are left with 1 (key, ptr) pair. Its sibling (left) is

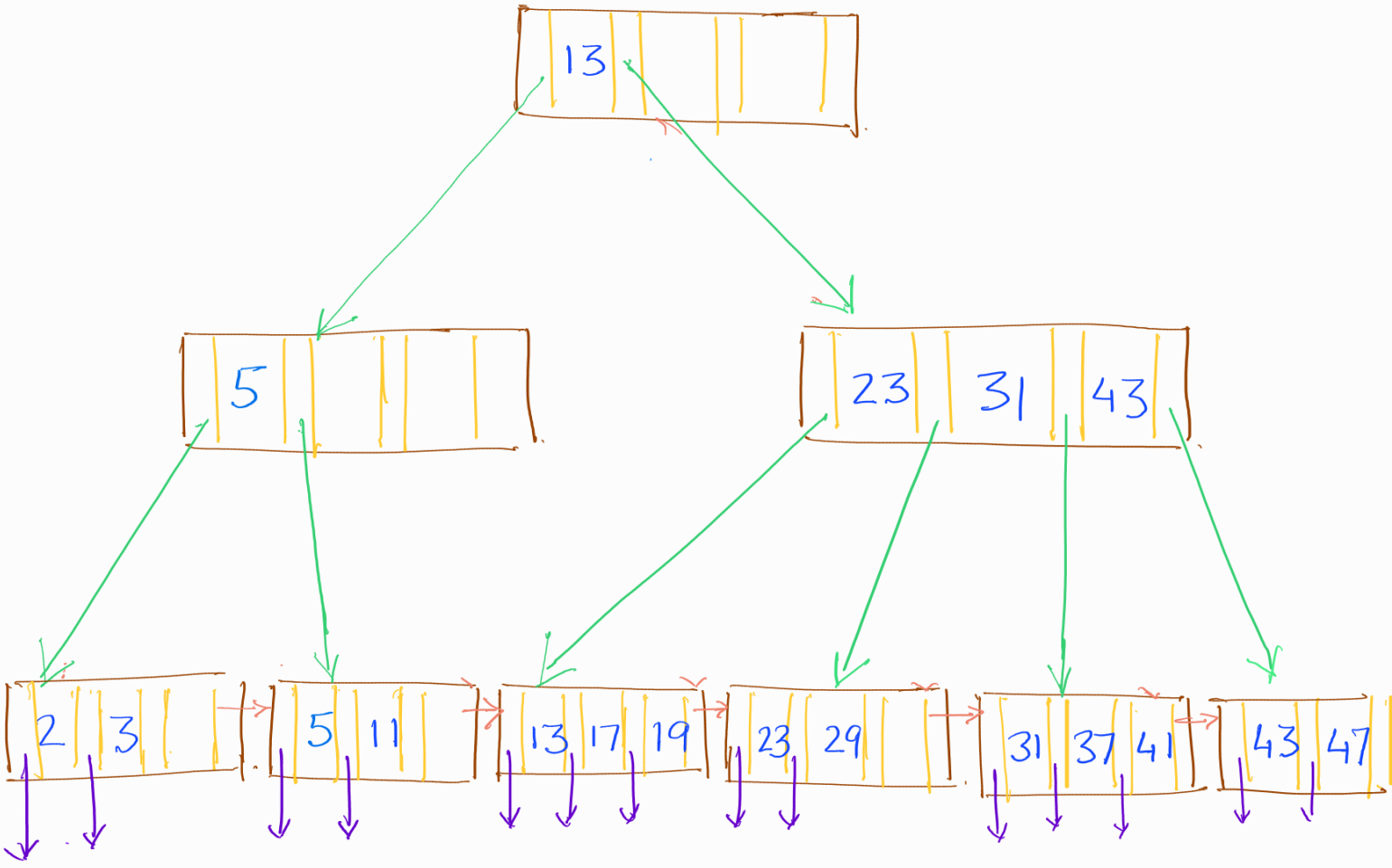


which is full.

We now borrow a key-ptr pair from the sibling since even after transferring the largest key 5 of the first leaf to the sibling, it is left with 2 (key, ptr) pairs (2,3), which is within the permissible limits. Likewise, by adding 5 into the original leaf that contained 7, which was deleted, would now bring it to 2 keys (5, 11).

One final editing is needed. The pointer to the second leaf in the parent is the smallest key value in that child node, which was 7. After the rearrangement, the key 7 in the parent is now replaced by the smallest key of the second child, which is 5. The new B-tree is as follows.



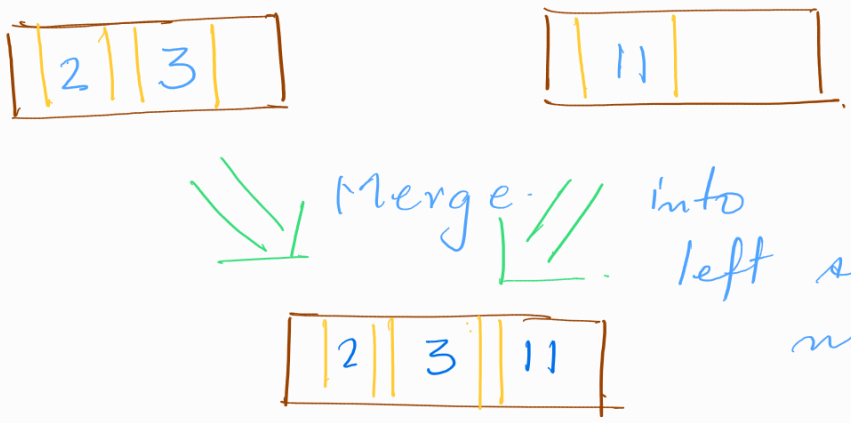


Now suppose we delete 5. Upon search we are led to the second leaf from the left. It is at its minimum occupancy. By deleting 5, this leaf would have only one key, which is below min. of

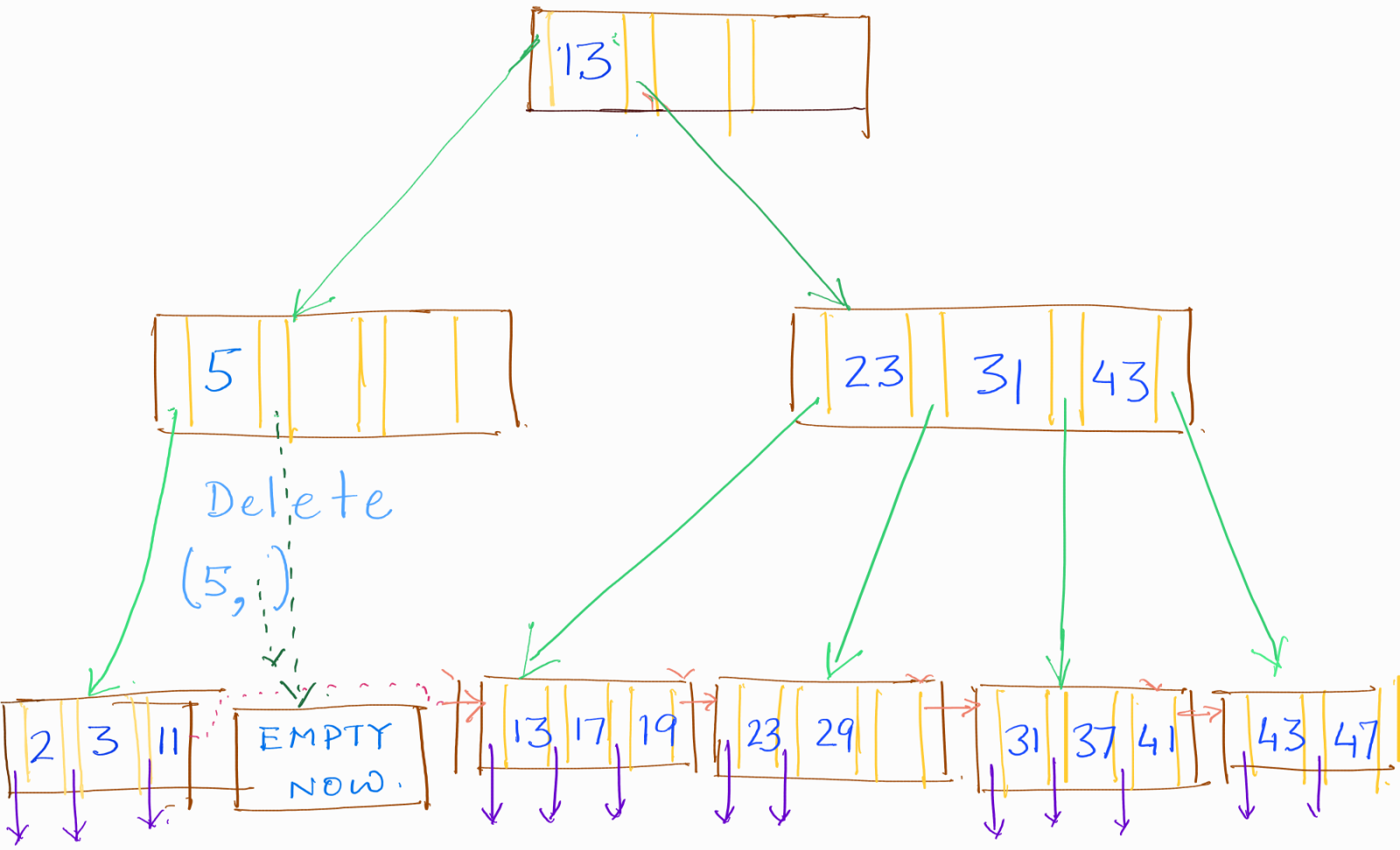
$$\left\lfloor \frac{n+1}{2} \right\rfloor = \left\lfloor \frac{3+1}{2} \right\rfloor = 2.$$

As before, we consider its (only) sibling, its left neighbor. However this too has the minimum number 2 of keys, and so

borrowing is not possible. The only option left is to merge the two sibling nodes.

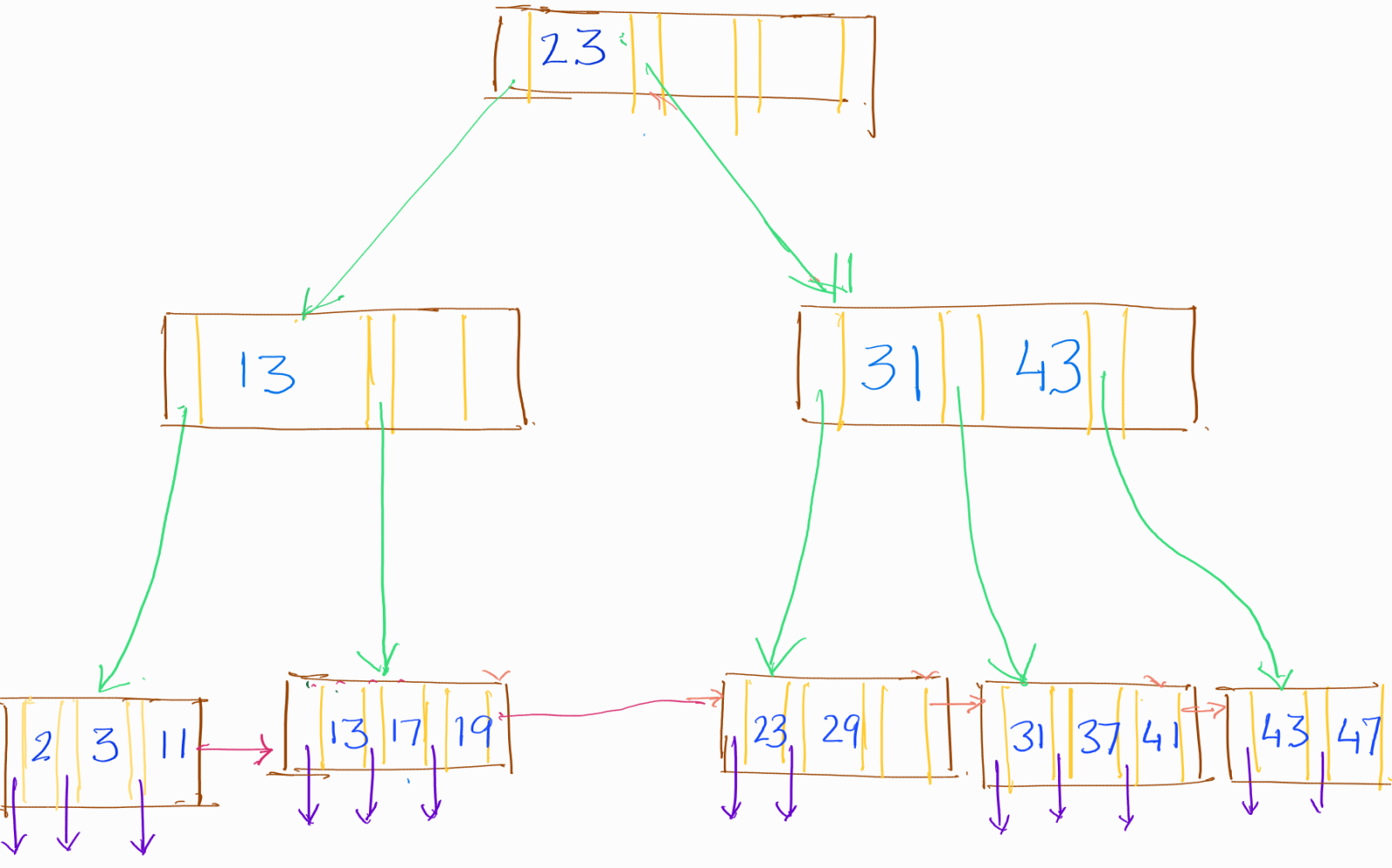


The parent node has to be corrected, earlier it contained the key 5 and pointers to the left and right sibling node. Now, we have to delete the key 5 and the pointer to right child. This situation is depicted below.



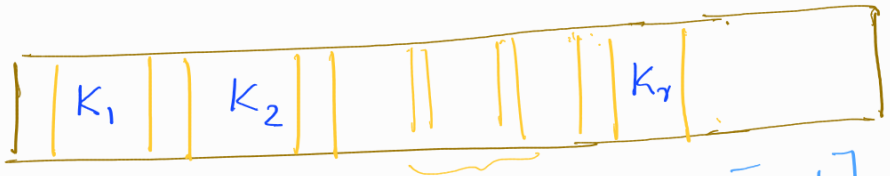
The deletion problem is propagated one level up the tree. The node from where key 5 has to be deleted has 2 pointers, one of which (the right ptr) has to be deleted. That would leave it with no key and 1 pointer. We consider its sibling, the right one. It has 4 pointers and 3 keys. We now decide to borrow an appropriate key and pointer pair. The pointer must be the ptr to the left child of the sibling node. Its

min value is 13. So this is the new key to replace 5. The left most key in the right sibling is 23. That cannot continue as its pointer has shifted left.



Since, no further restructuring is required, this 23 is now the left most key of the right sibling subtree. It therefore replaces 13 at the root (i.e., its parent).

B-Trees with duplicate occurrences of keys: There is a slight change in the definition of how keys are stored at internal nodes. Suppose the internal node is something like this:

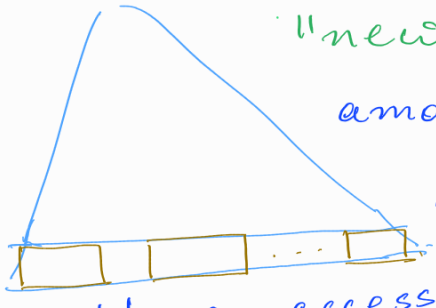


r keys.
$r+1$ pointers

$$\left\lceil \frac{n+1}{2} \right\rceil \leq r+1 \leq n+1$$

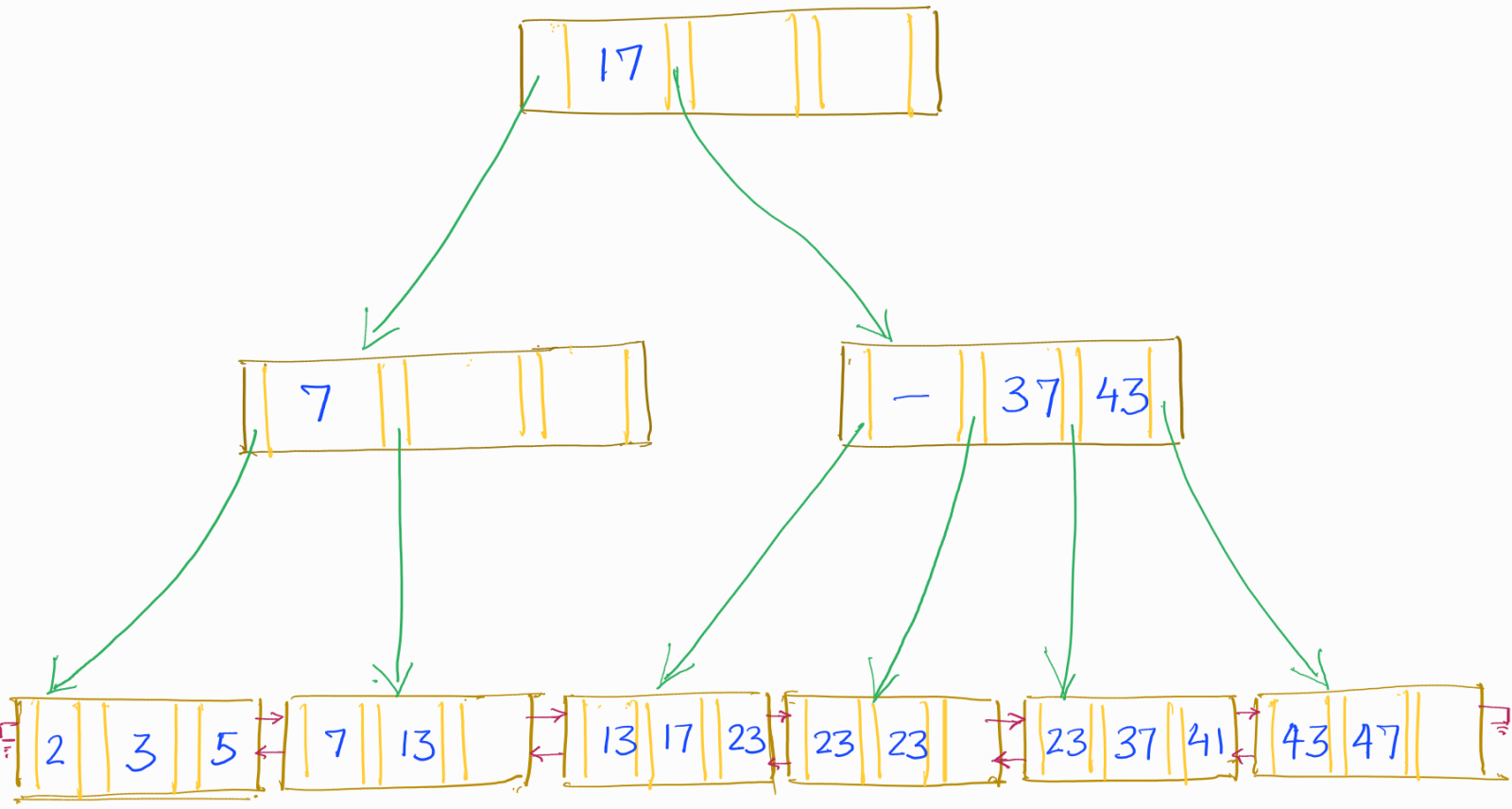
New key in a subtree: There are no occurrences of K_i in the portion of the full B-tree to the left of this subtree.

K_i is the smallest "new" key occurring among the leaves in this subtree.



Subtree accessible from $(i+1)^{st}$ pointer.

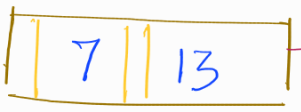
Consider an example.



The tree has duplicate key values.

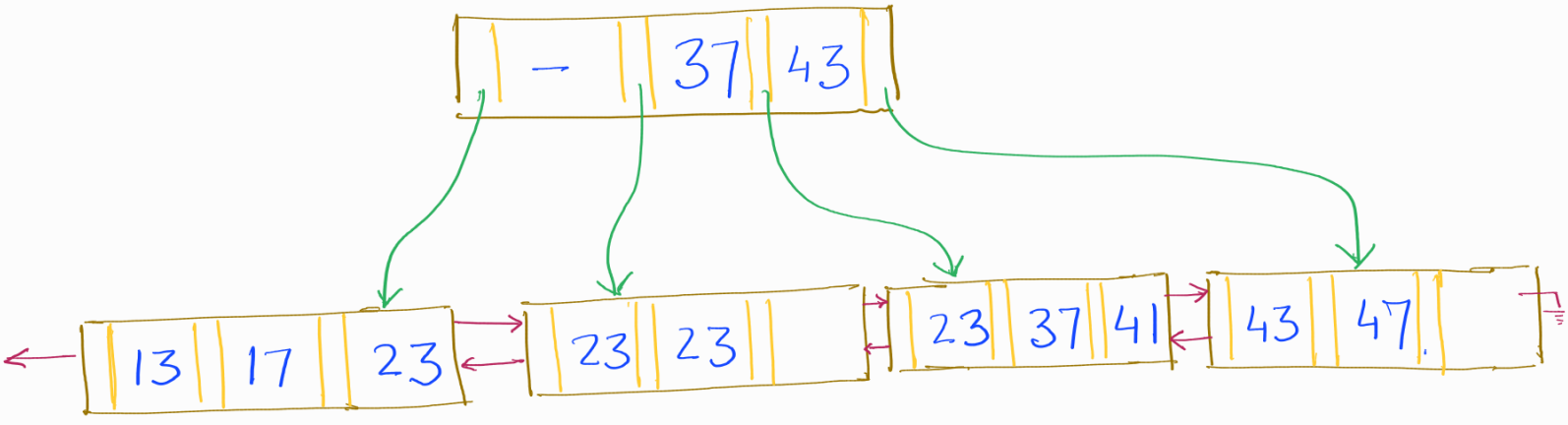
1. The root has key 17, not 13. The smallest key among the leaves of the subtree pointed by the second child of the root is 13. But 13 occurs to the left of this subtree and hence is "not new". The next key 17 is the smallest new key in this subtree. Hence it is the key at the root.

Note therefore that a search for 13 will proceed down the first child of root, then the 2nd child of.

the first node at level 2. We then obtain the leaf node 

. We obtain the key value 13 and follow the pointer to the next right neighboring leaf node until we find a key > 13.

2. The second child of the root has second key 37, which is the



smallest "new" key in the leaf node pointed by it. (and not 23, as it is not new).

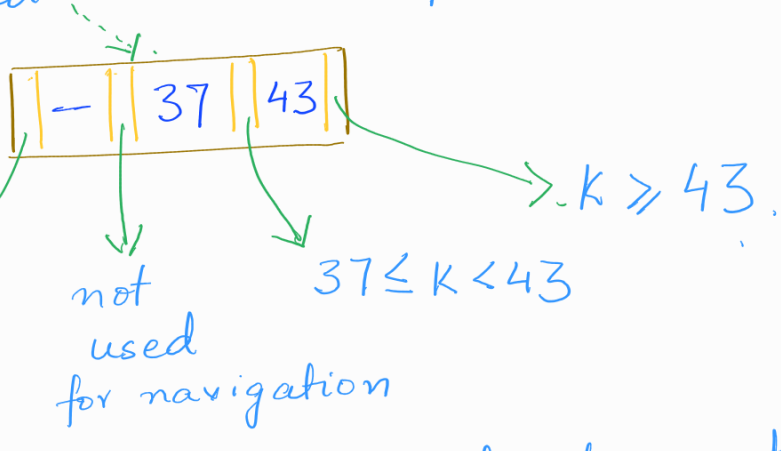
More interestingly, the pointer to the second child of the node has only 23, and this is not new. Hence the key is left as —.

2a. Put another way, if we were searching for any key from the root, and reached the second child

of the root, we would never want to start at its second child. If we were searching for 23 or anything lower, we want to start at its first child. If we were looking for 17, we would find it; if we were searching for 23, we will find the first of all the occurrences of 23, and navigate through right neighbor pointers until we do not find 23 any more.

3. Searching for 13, we will not reach the second child of the root; instead we are directed to the first child.

4. If we are searching for any key between 24 and 36, we are directed to the second child of the root.



Navigation via search key at the second child node of root.

Remarks :

1. Range queries can be solved using B trees : E.g.

a)
```
select *
from R.
where R.Key >= 40
```

b)
```
select *
from R
where R.Key >= 10
and R.Key <= 50
```

To search for keys in the range $[a, b]$ (or, (a, b) or $(a, b]$ etc.) look up to find key a. This leads us to the leaf that would contain the first occurrence of a if it exists, or not. We search this leaf node for successors of a and follow right neighbor pointers until the first occurrence of b is found or a key greater than b is found.

Like-wise, we can search for $[a, \infty)$ or $(-\infty, a]$, the latter by searching for key value a and following previous neighbor pointers.

2. lookups are O(height of B-tree). Insertion and deletion are 2 passes, one downward and one back upwards,

with at most one split (or merge) at each level.

No. of records = N.

B-tree parameter = n.

No of leaves $\leq \left\lceil \frac{N}{\lfloor n+1/2 \rfloor} \right\rceil = L$.

No. of internal nodes at height 1 $\leq \left\lfloor \frac{L}{\lceil \frac{n+1}{2} \rceil} \right\rfloor = L_1$

No. of internal nodes at height 2 $\leq \left\lfloor \frac{L_1}{\lceil \frac{n+1}{2} \rceil} \right\rfloor = L_2$.

...

This continues till the first index h s.t. $L_h \leq n+1$.

or $\left(\frac{L_1}{\lceil \frac{n+1}{2} \rceil}\right)^{h-1} \leq n+1$.

Suffices if $h \leq 1 + \log_{\lceil \frac{n+1}{2} \rceil} \frac{L_1}{n+1}$.

— x —