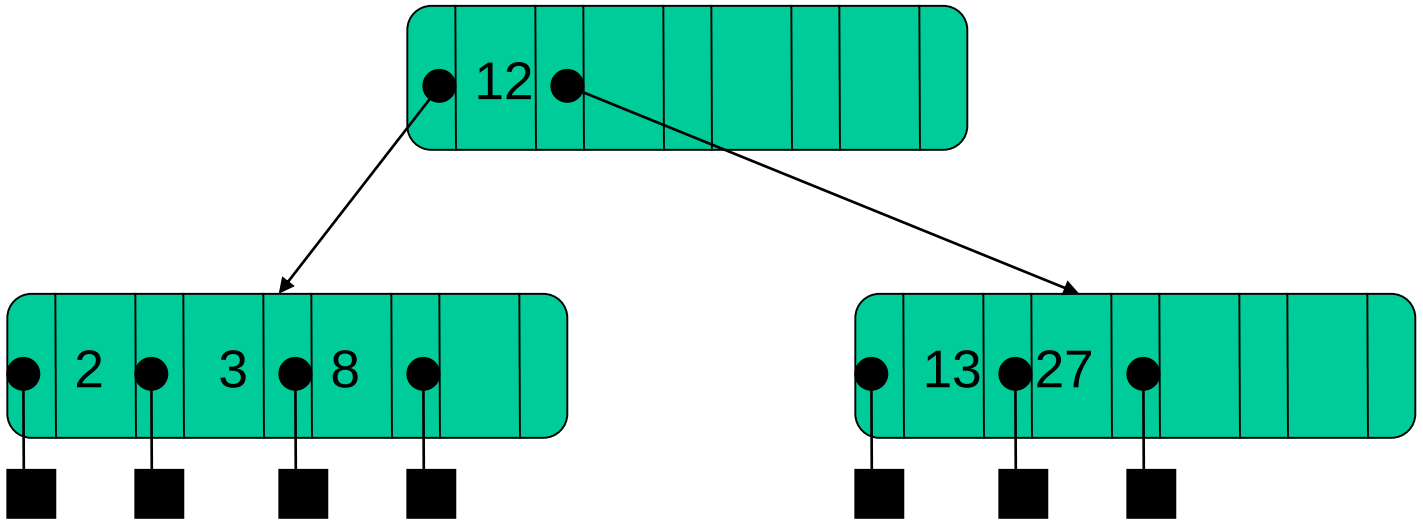
B-Trees

Idea 1: Use *m*-way search trees. (vs binary)

Idea 2: Don't require that each node always be full. Empty space will permit insertion without rebalancing. Allowing empty space after a deletion can also avoid rebalancing.

Idea 3: Rebalancing will sometimes be necessary: figure out how to do it in time proportional to the height of the tree.

B-Tree Example with m = 5



The root has between 2 and *m (order)* children unless the tree is the root alone.

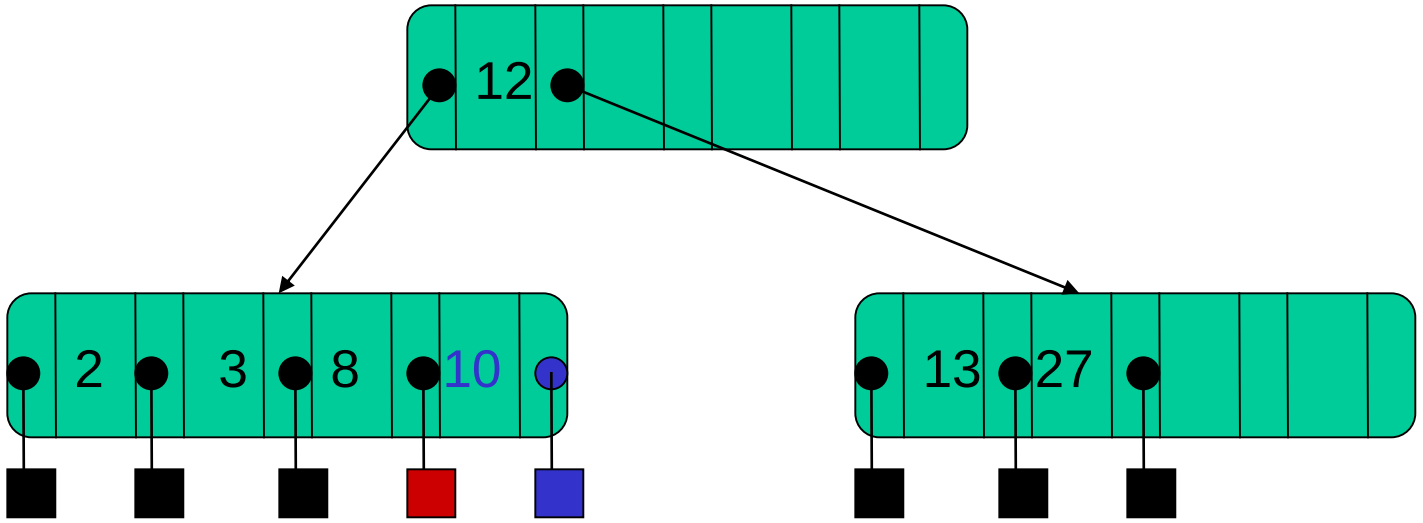
Each non-root internal node has between *m*/2 and *m* children. m=5 → 2.

Each non-root has at least m/2 and at most m-1 keys

Left/right children are small/greater than parent

All leaf nodes are at the same level.

Insert 10

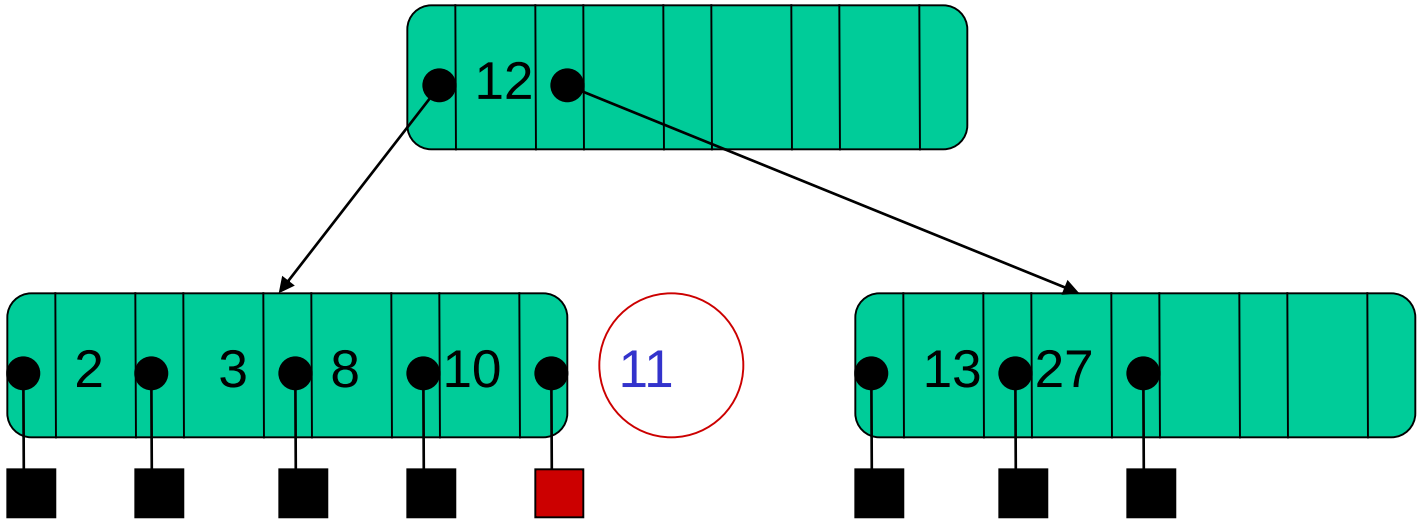


We find the location for 10 by following a path from the root using the stored key values to guide the search.

The search falls out the tree at the 4th child of the 1st child of the root.

The 1st child of the root has room for the new element, so we store it there.

Insert 11

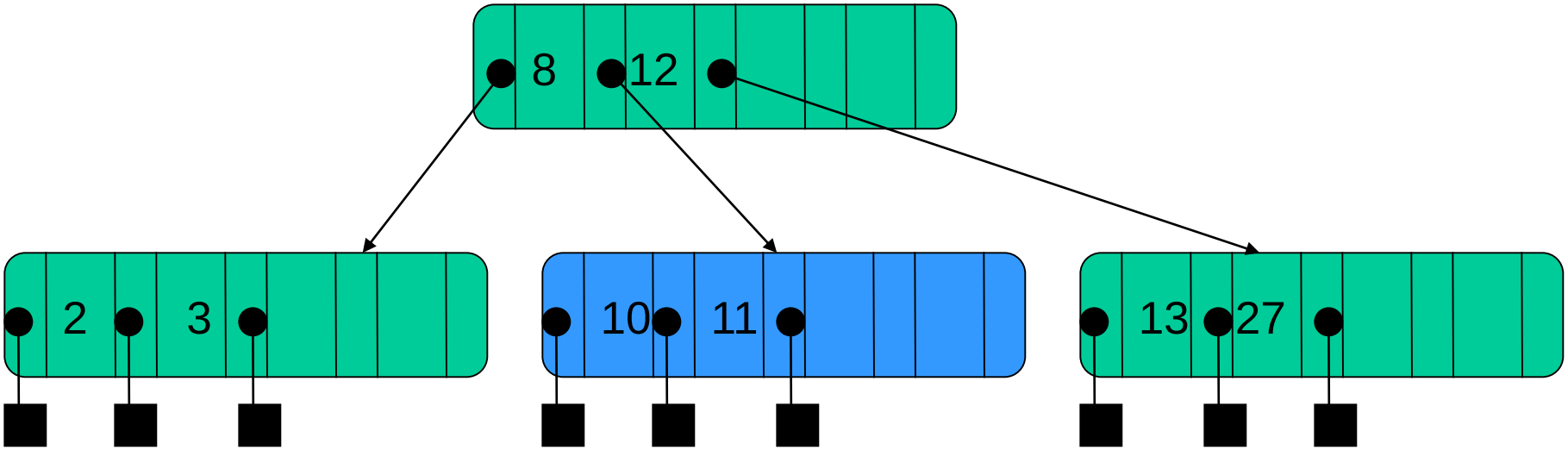


We fall out of the tree at the child to the right of key 10.

But there is no more room in the left child of the root to hold 11.

Therefore, we must *split* this node...

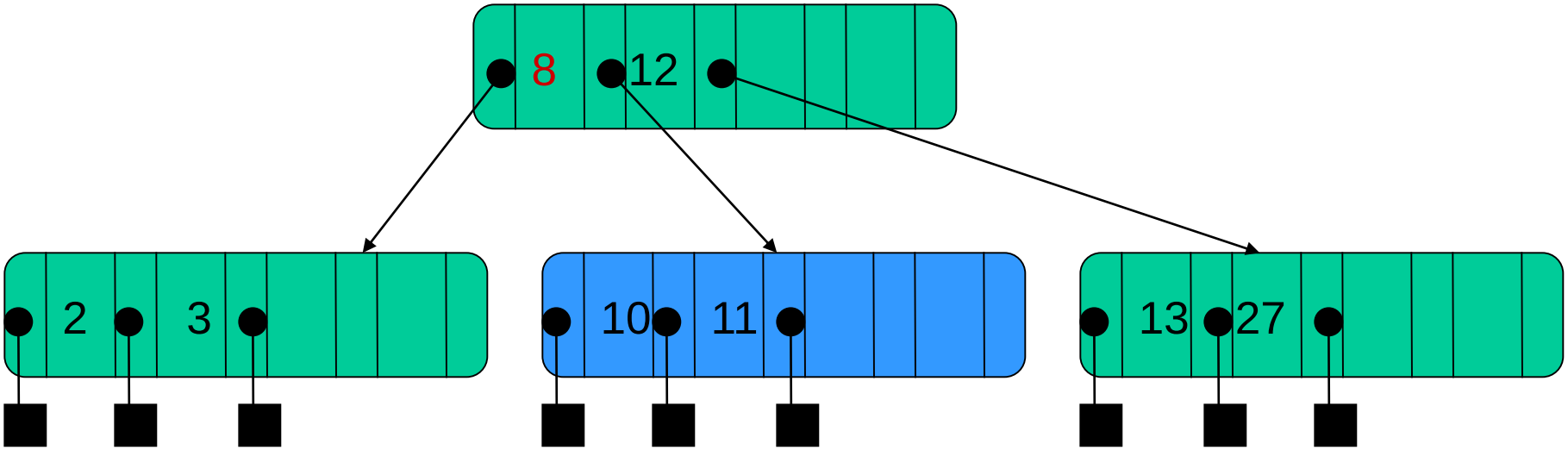
Insert 11 (Continued)



The m + 1 children are divided evenly between the old and new nodes.

The parent gets one new child. (If the parent become overfull, then it, too, will have to be split).

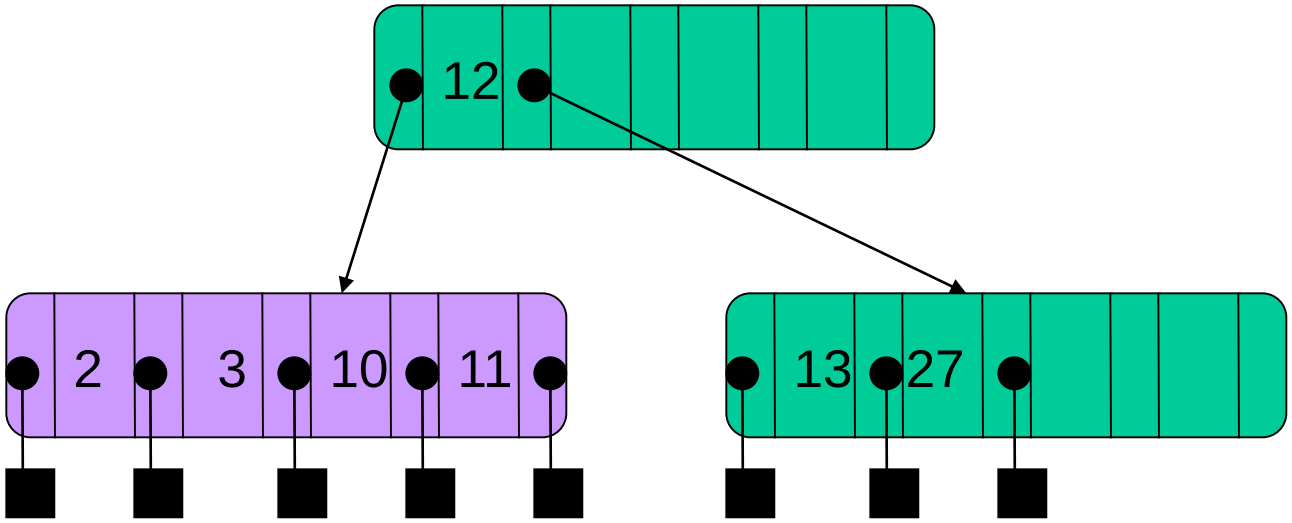
Remove 8



Removing 8 might force us to move another key up from one of the children. It could either be the 3 from the 1st child or the 10 from the second child.

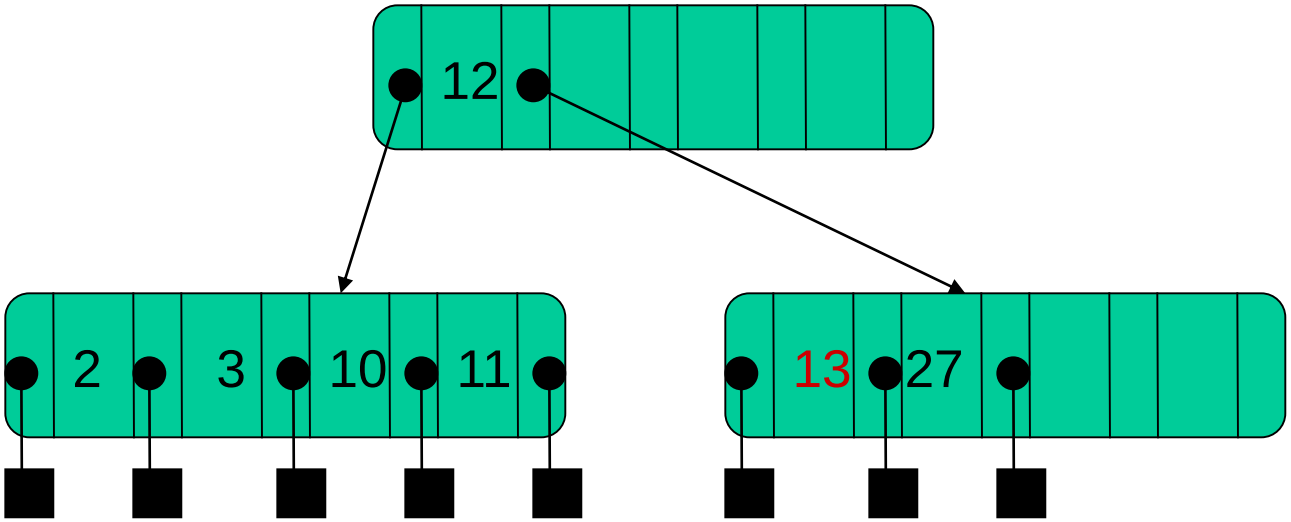
However, neither child has more than the minimum number of keys (2), so the two nodes will have to be merged. Nothing moves up.

Remove 8 (Continued)



The root contains one fewer key, and has one fewer child.

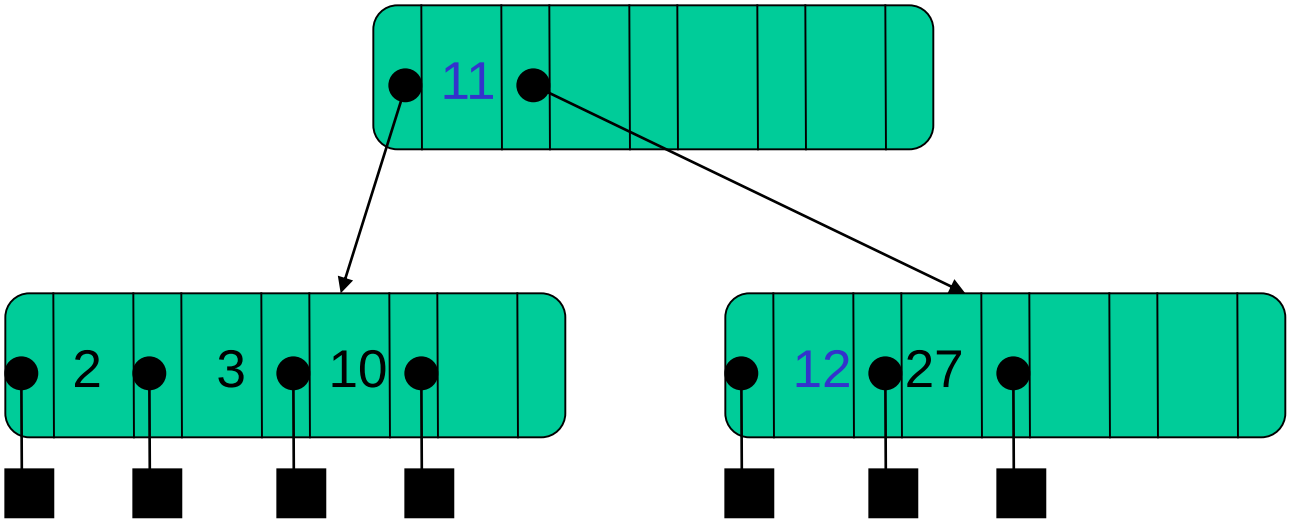
Remove 13



Removing 13 would cause the node containing it to become under-full

To fix this, we try to reassign one key from a sibling that has spares.

Remove 13 (Cont)

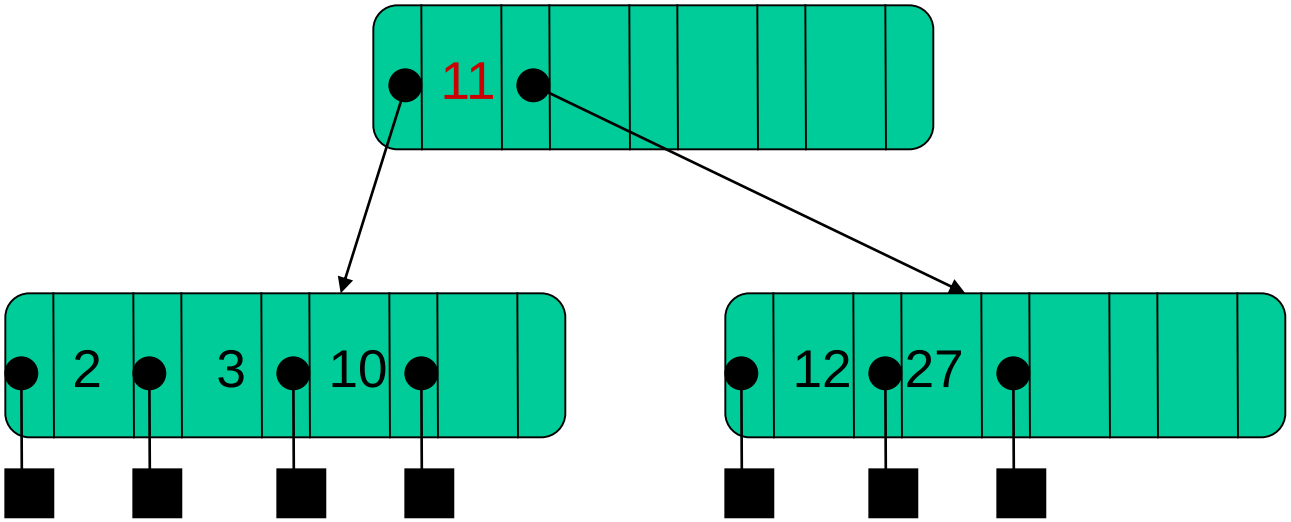


The 13 is replaced by the parent's key 12.

The parent's key 12 is replaced by the spare key 11 from the left sibling.

The sibling has one fewer element.

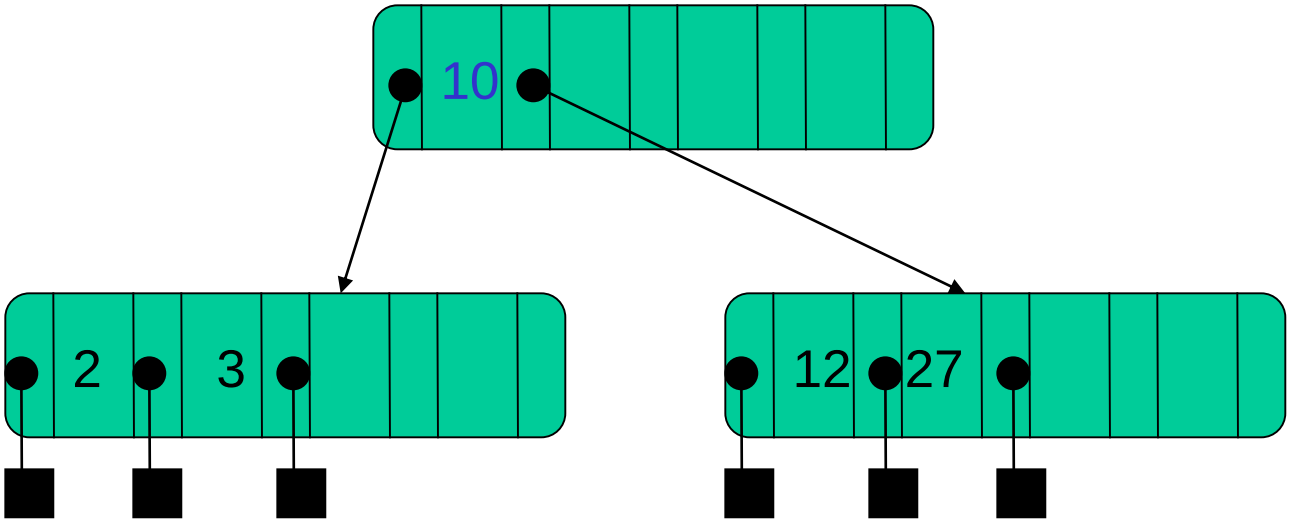
Remove 11



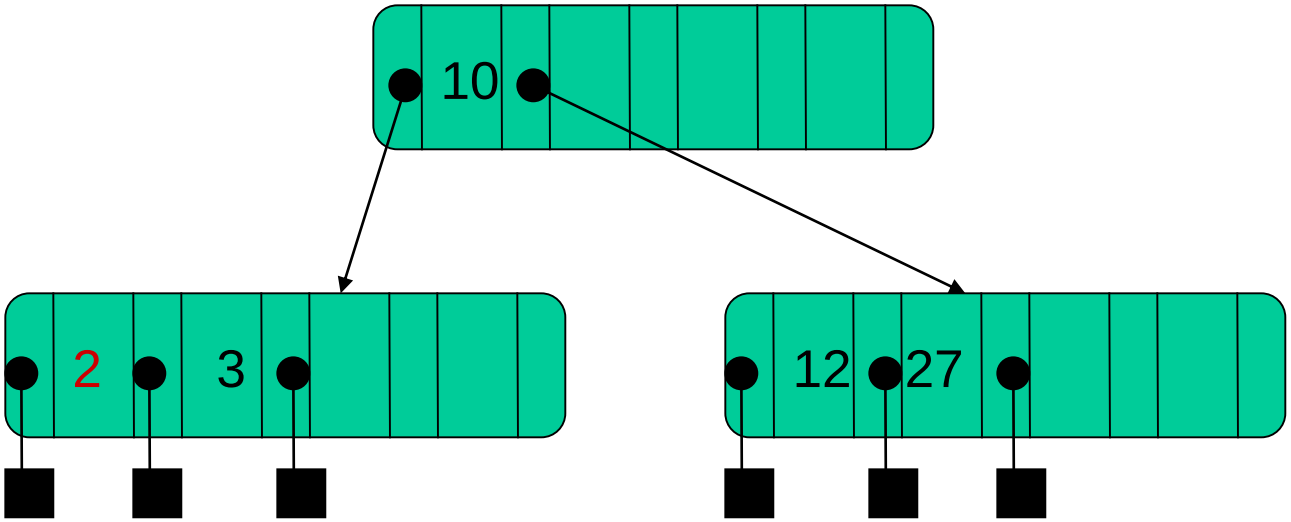
11 is in a non-leaf, so replace it by the value immediately preceding: 10.

10 is at leaf, and this node has spares, so just delete it there.

Remove 11 (Cont)



Remove 2

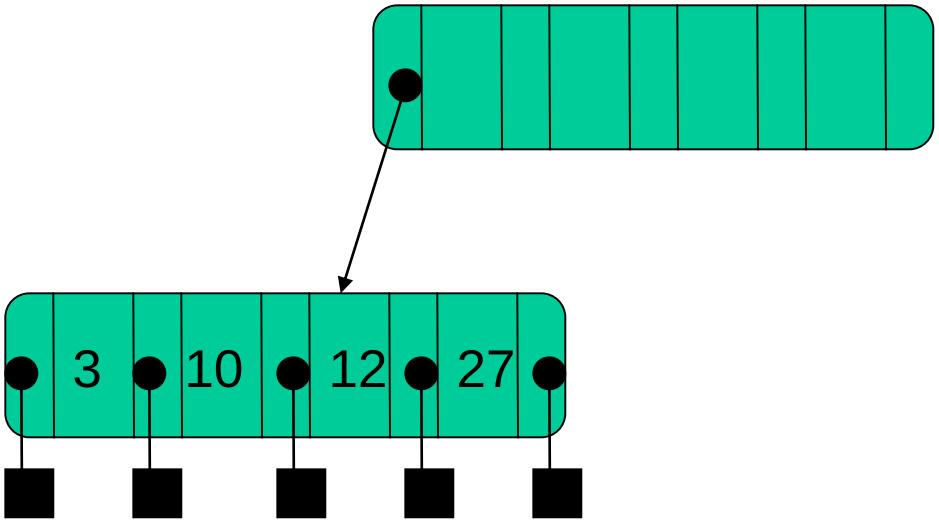


Although 2 is at leaf level, removing it leads to an underfull node.

The node has no left sibling. It does have a right sibling, but that node is at its minimum occupancy already.

Therefore, the node must be merged with its right sibling.

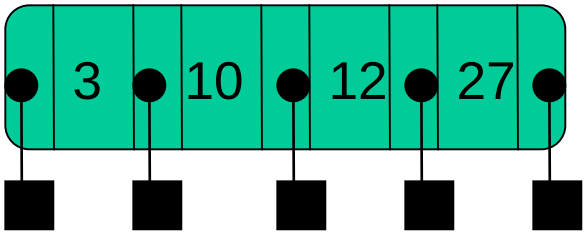
Remove 2 (Cont)



The result is illegal, because the root does not have at least 2 children.

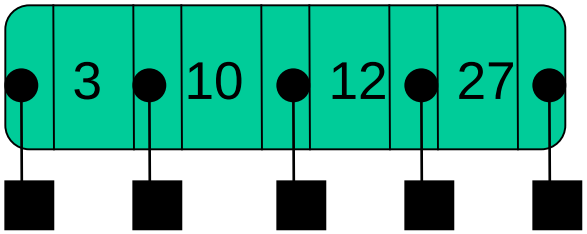
Therefore, we must remove the root, making its child the new root.

Remove 2 (Cont)



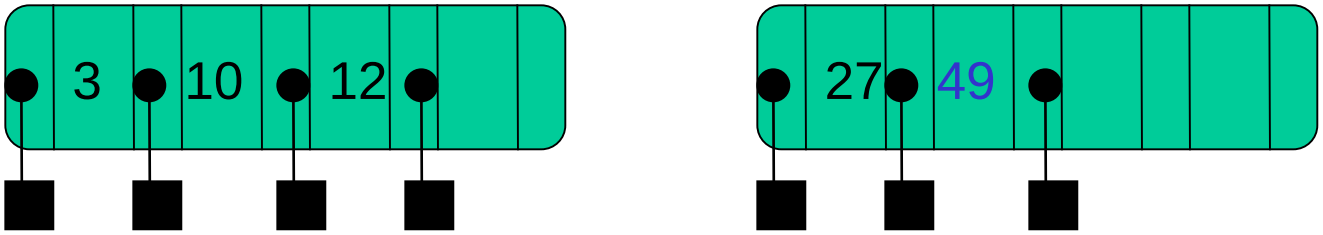
The new B-tree has only one node, the root.

Insert 49



Let's put an element into this B-tree.

Insert 49 (Cont)

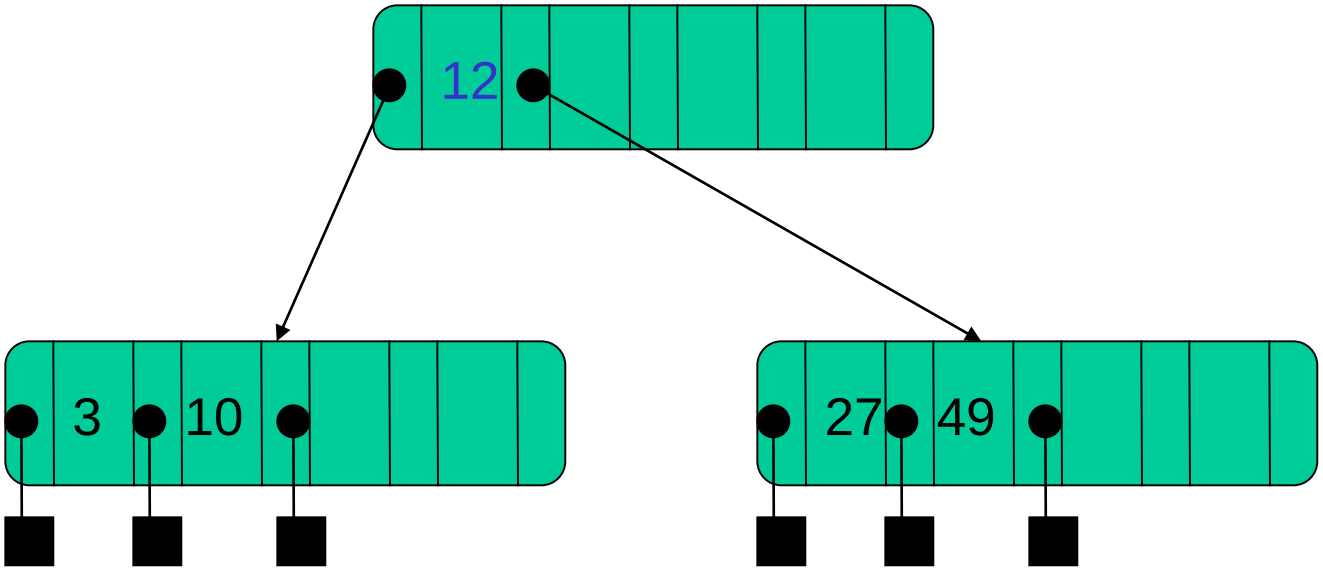


Adding this key make the node overfull, so it must be split into two.

But this node was the root.

So we must construct a new root, and make these its children.

Insert 49 (Cont)



The middle key (12) is moved up into the root.

The result is a B-tree with one more level.

B-Tree performance

Let h = height of the B-tree.

get(k): at most h disk accesses. O(h)

put(k): at most $3h + 1$ disk accesses. O(h)

remove(k): at most $3h$ disk accesses. O(h)

$h < \log_d (n + 1)/2 + 1$ where $d = \lceil m/2 \rceil$ (Sahni, p.641).

An important point is that the constant factors are relatively low.

m should be chosen so as to match the maximum node size to the block size on the disk.

Example: $m = 128$, $d = 64$, $n \approx 64^3 = 262144$, $h = 4$.

.

In practical applications, however, B-Trees of large order (e.g., $m = 128$) are more common than low-order B-Trees such as m=5.