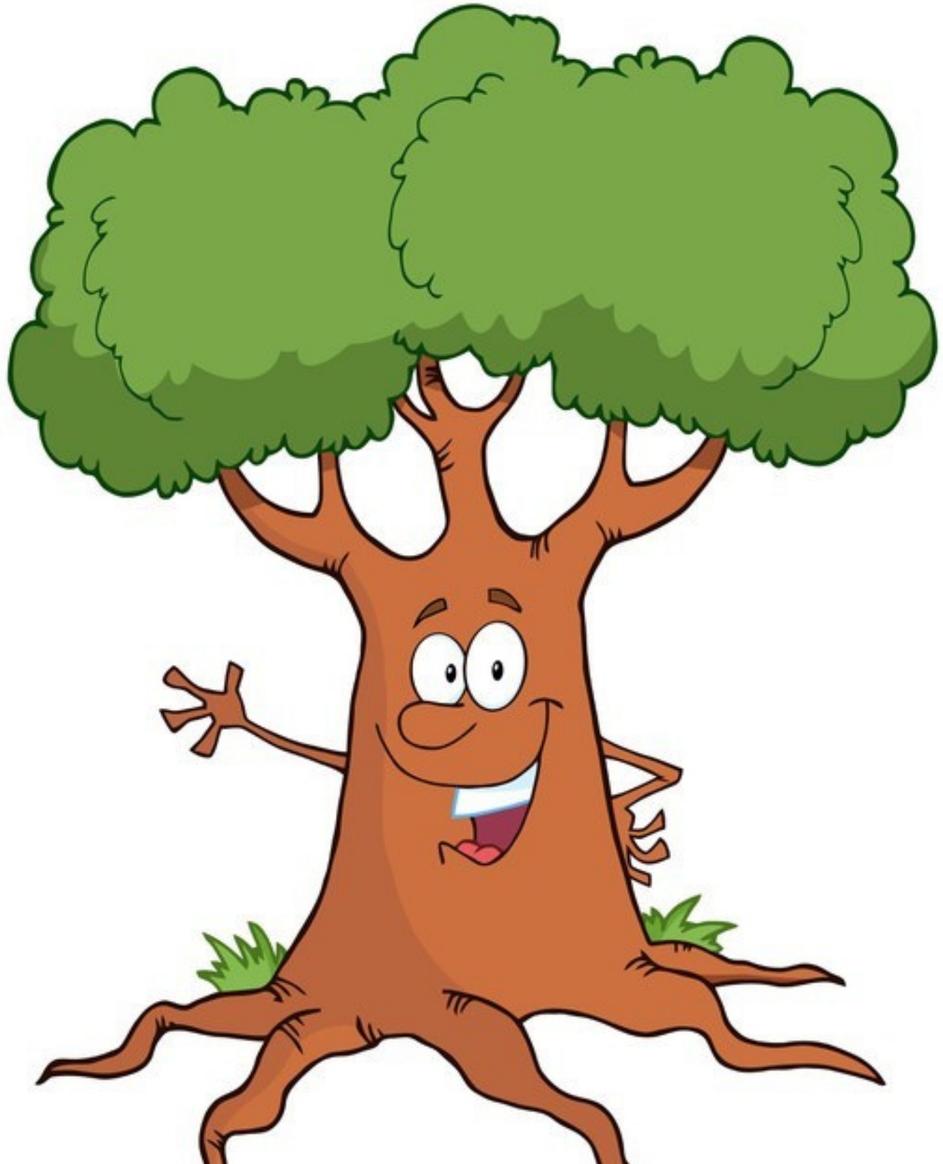
B-tree



Some standard texts...

- B-tree is a self-balancing tree data structure
- B-tree maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time
- B-tree generalizes binary search tree

There are already AVL and Red-Black trees.

Why do we need another data structure with these standard properties?

In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in the main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in the main memory.

When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to the main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses.

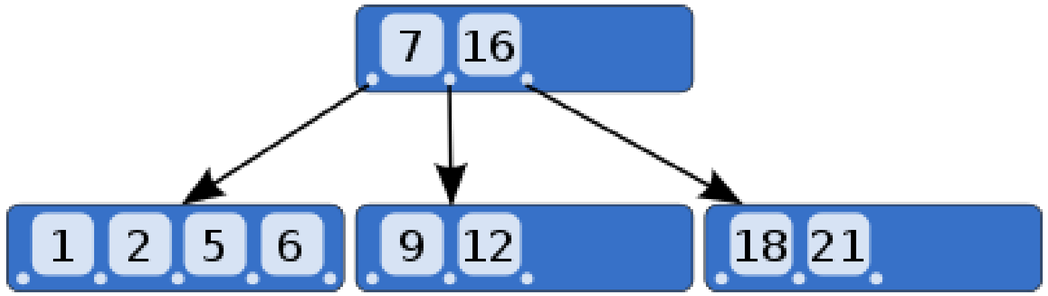
Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree.

The height of B-Trees is kept low by putting the maximum possible keys in a B-Tree node. Generally, the B-Tree node size is kept equal to the disk block size.

Definition

- Every node has at most m children.
- Every internal node except the root has at least ⌈m/2⌉ children.
- Every non-leaf node has at least two children.
- All leaves appear on the same level and carry no information.
- A non-leaf node with k children contains k−1 keys.

Each internal node's keys act as separation values which divide its subtrees. For example, if an internal node has 3 child nodes (or subtrees) then it must have 2 keys: a1 and a2. All values in the leftmost subtree will be less than a1, all values in the middle subtree will be between a1 and a2, and all values in the rightmost subtree will be greater than a2.



Why B?

Bayer and McCreight never explained what, if anything, the B stands for: Boeing, balanced, broad, bushy, and Bayer have been suggested.

McCreight has said that "the more you think about what the B in B-trees means, the better you understand B-trees."

Best case and worst case heights

Let $h \geq -1$ be the height of the classic B-tree. Let $n \geq 0$ be the number of entries in the tree. Let m be the maximum number of children a node can have. Each node can have at most $m-1$ keys.

It can be shown (by induction for example) that a B-tree of height h with all its nodes completely filled has $n = m^{h+1}-1$ entries. Hence, the best case height (i.e. the minimum height) of a B-tree is:

$$h_{\min} = \lceil \log_m (n+1) \rceil - 1$$

Let d be the minimum number of children an internal (non-root) node must have. For an ordinary B-tree, $d = \lceil m/2 \rceil$.

Comer (1979) and Cormen et al. (2001) give the worst case height (the maximum height) of a B-tree as

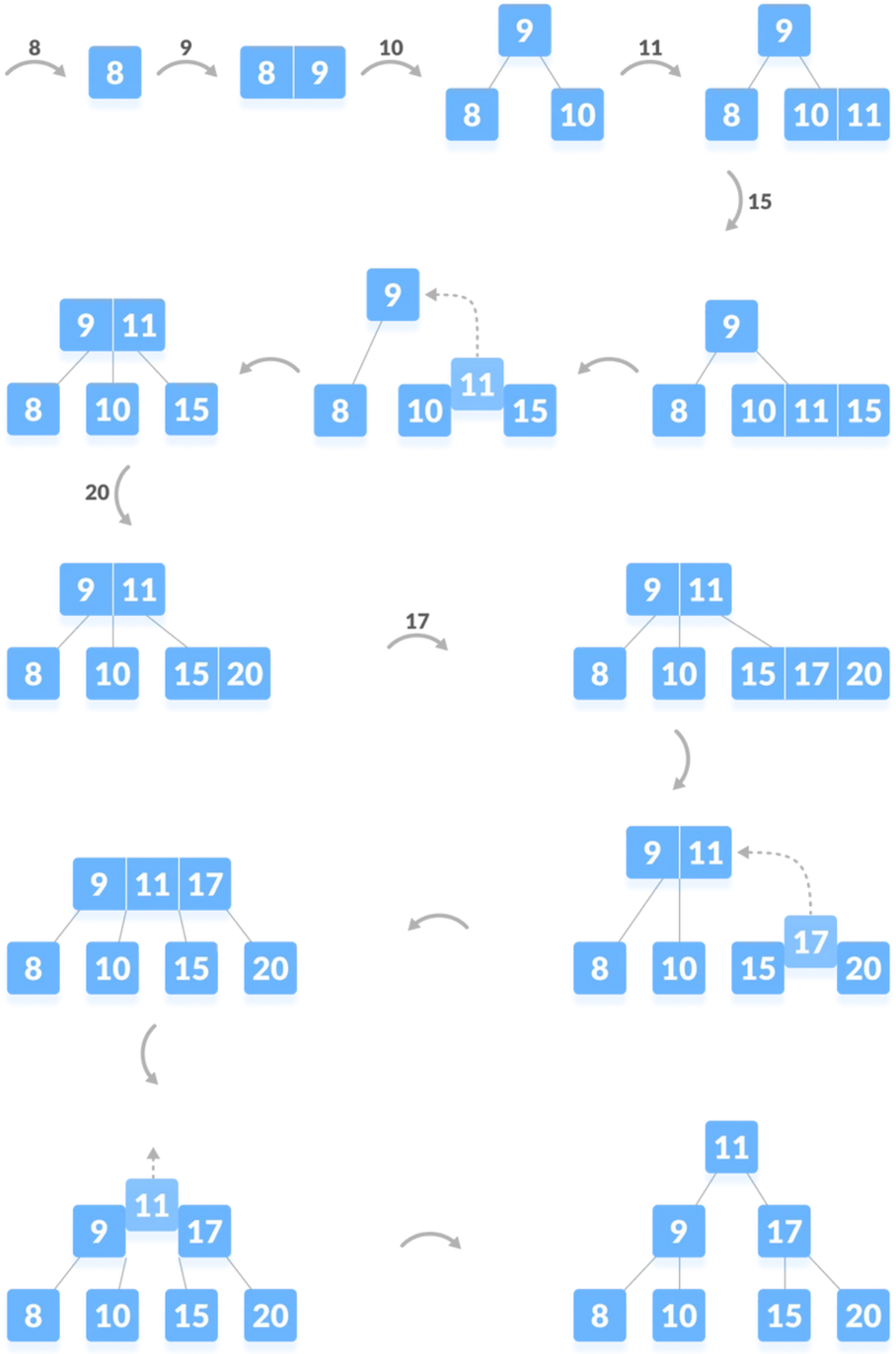
$$h_{\max} = \left\lfloor \log_d \frac{n+1}{2} \right\rfloor .$$

Insertion

Instertion

All insertions start at a leaf node. To insert a new element, search the tree to find the leaf node where the new element should be added. Insert the new element into that node with the following steps:

1. If the node contains fewer than the maximum allowed number of elements, then there is room for the new element. Insert the new element in the node, keeping the node's elements ordered.
2. Otherwise the node is full, evenly split it into two nodes so:
    a. A single median is chosen from among the leaf's elements and the new element that is being inserted.
    b. Values less than the median are put in the new left node and values greater than the median are put in the new right node, with the median acting as a separation value.
    c. The separation value is inserted in the node's parent, which may cause it to be split, and so on. If the node has no parent (i.e., the node was the root), create a new root above this node (increasing the height of the tree).

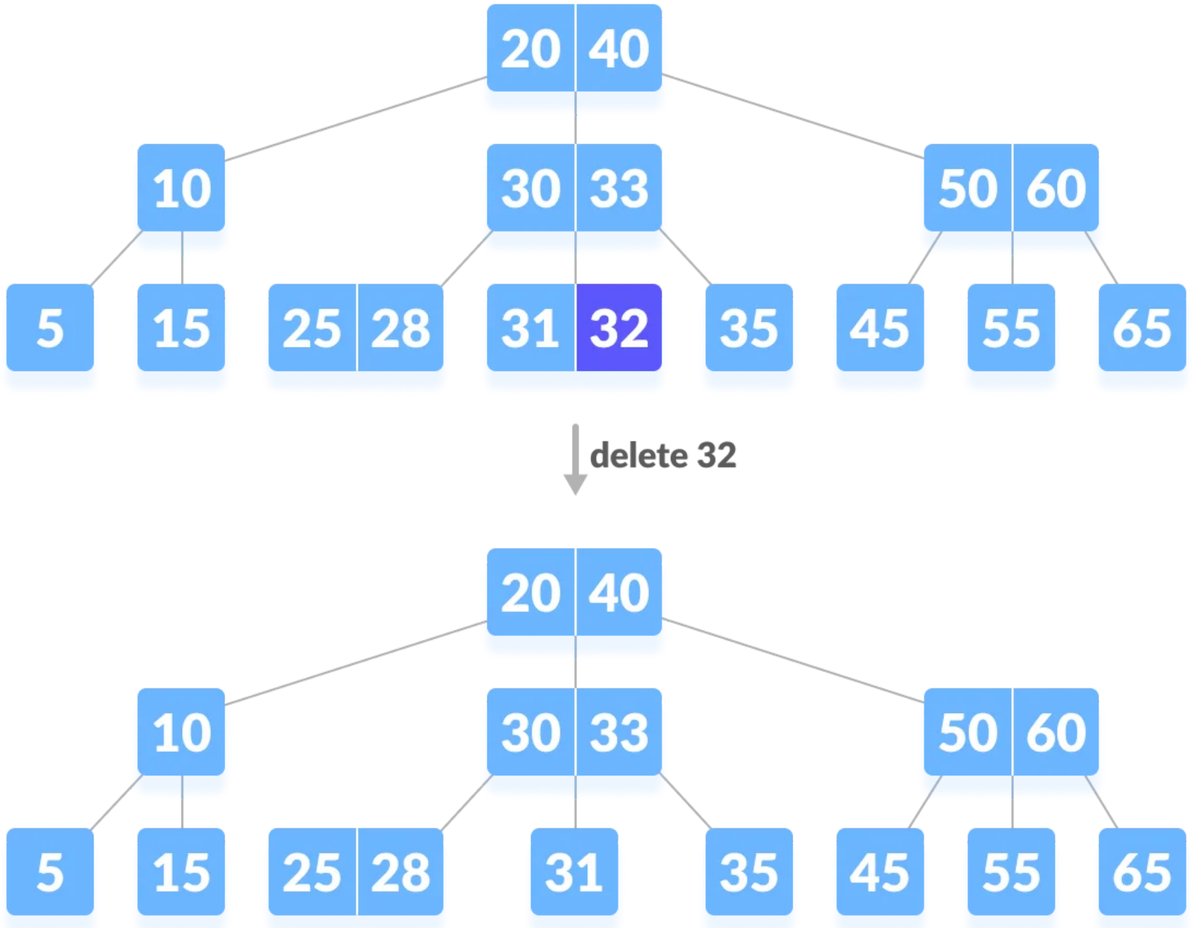

8
9
10
11
15
20
17

Deletion

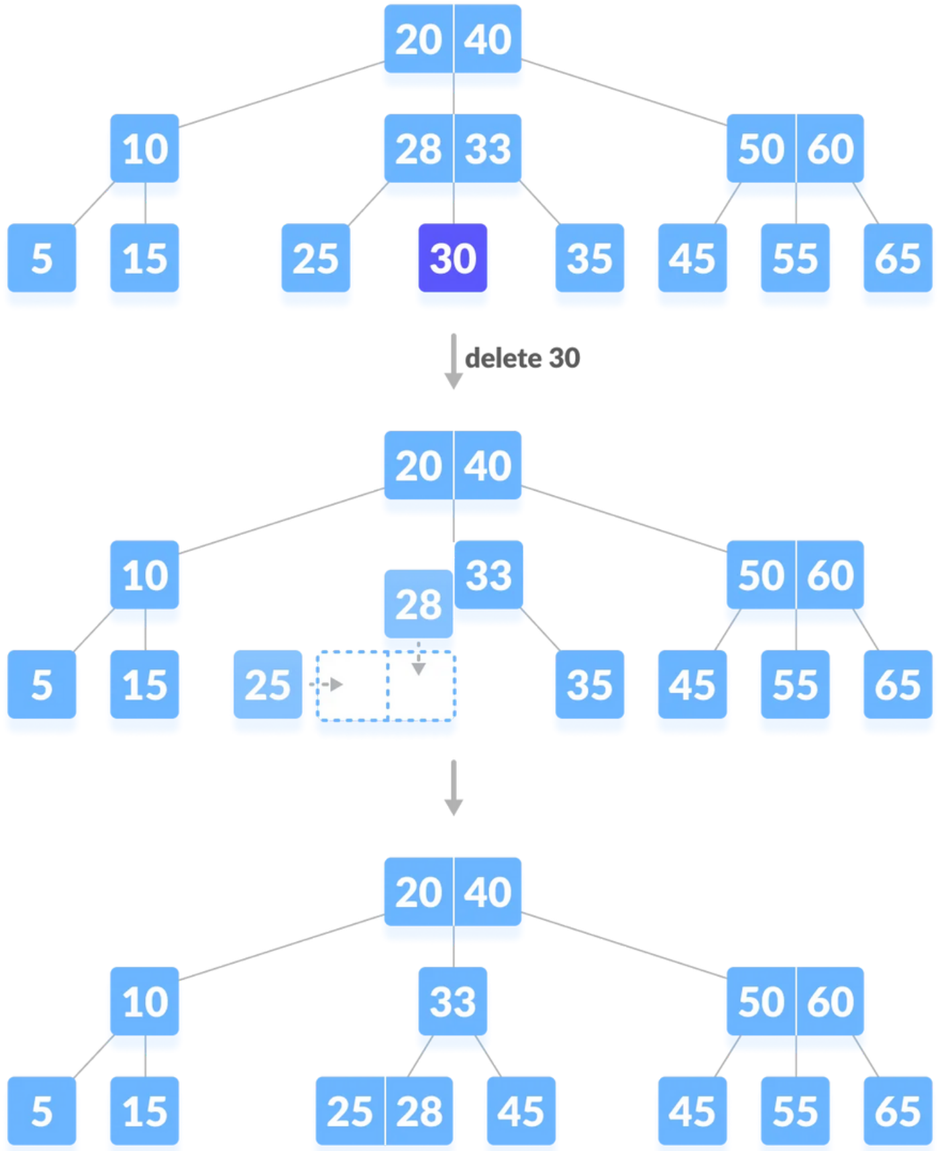
Case I

The key to be deleted lies in the leaf. There are two cases for it.

1. The deletion of the key does not violate the property of the minimum number of keys a node should hold.



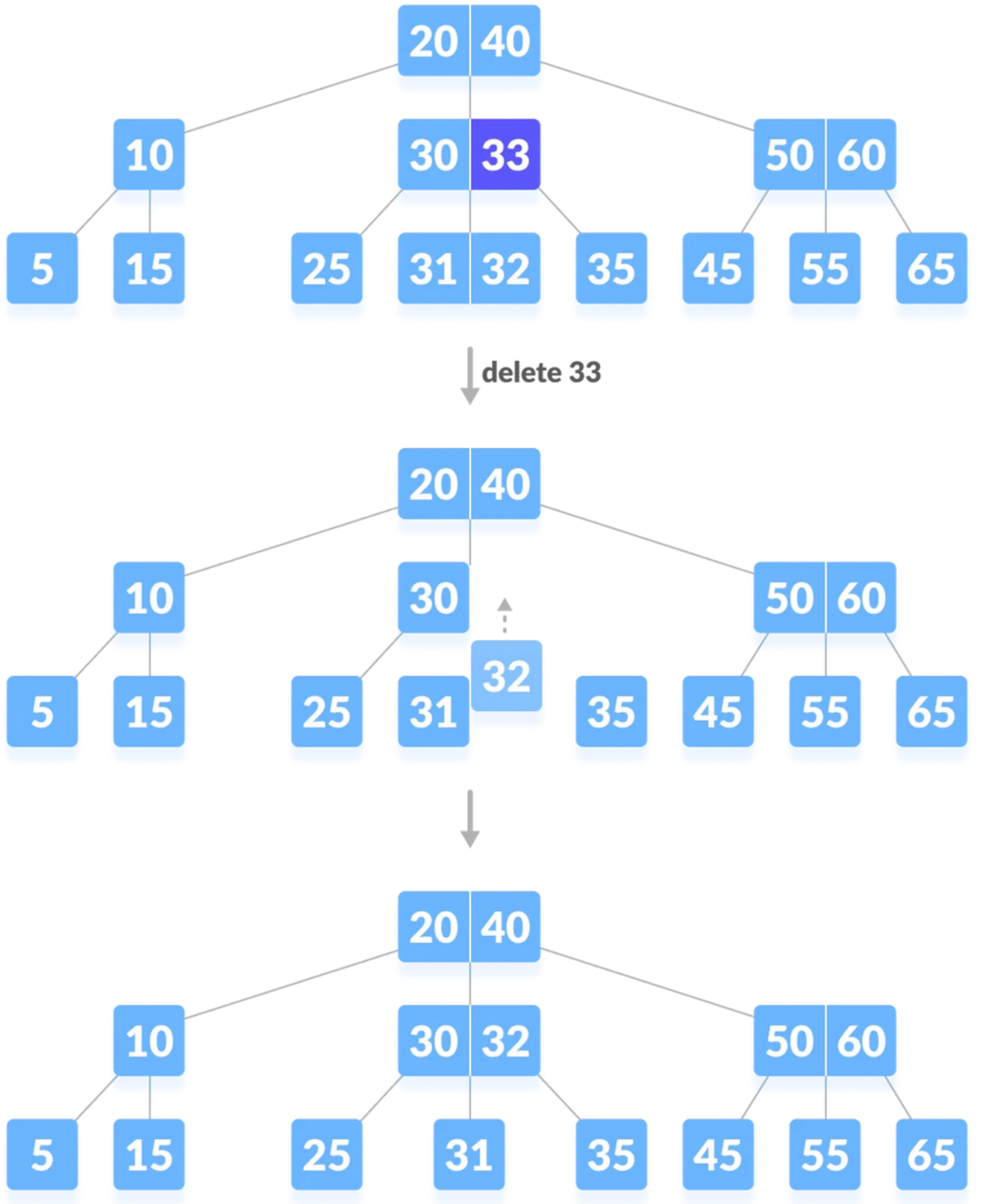
2'. If both the immediate sibling nodes already have a minimum number of keys, then merge the node with either the left sibling node or the right sibling node. This merging is done through the parent node.



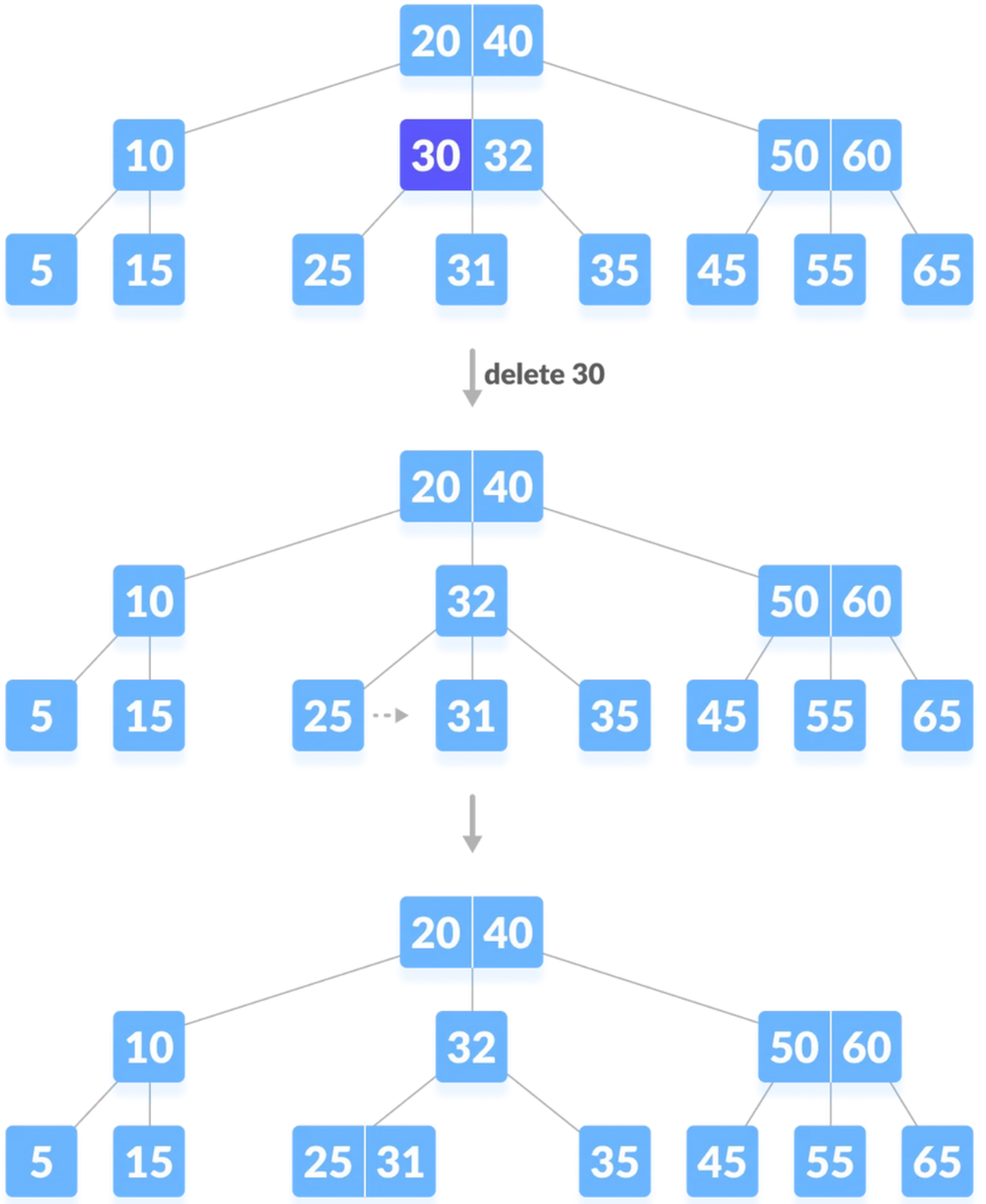
Case II

If the key to be deleted lies in the internal node, the following cases occur.

1. The internal node, which is deleted, is replaced by an inorder predecessor if the left child has more than the minimum number of keys.
2. The internal node, which is deleted, is replaced by an inorder successor if the right child has more than the minimum number of keys.



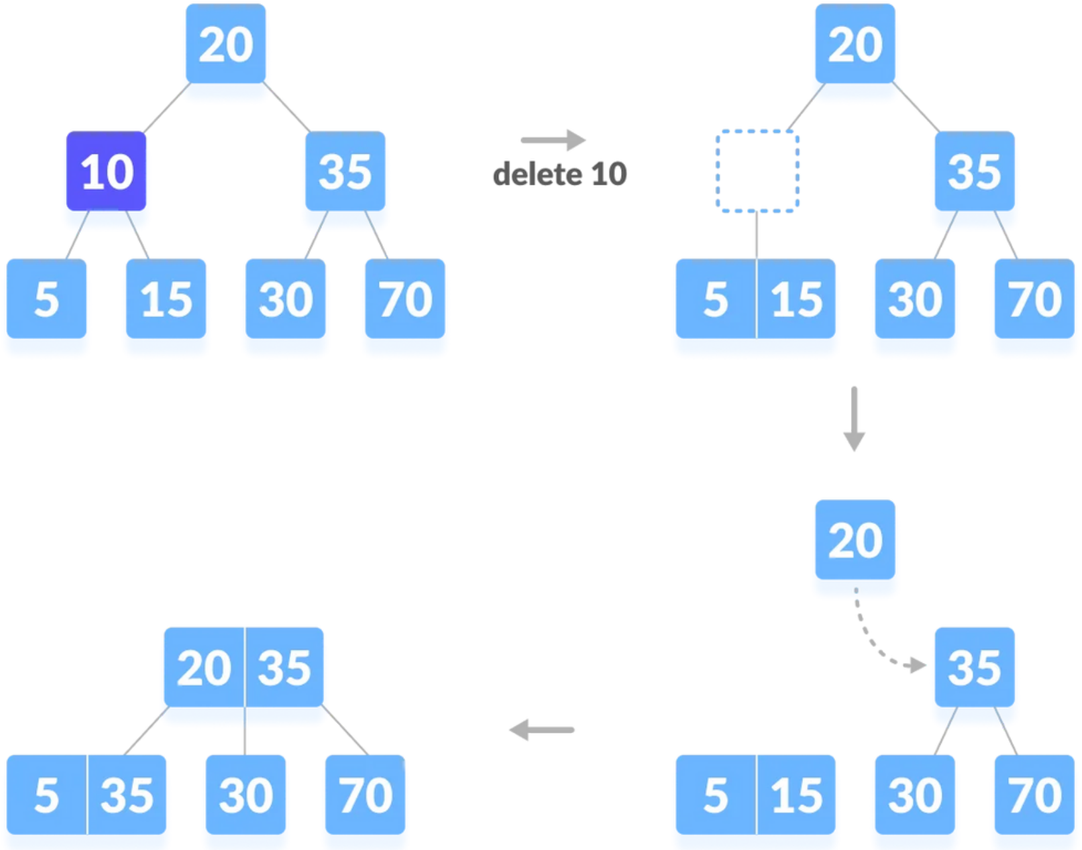
3. If either child has exactly a minimum number of keys then, merge the left and the right children.

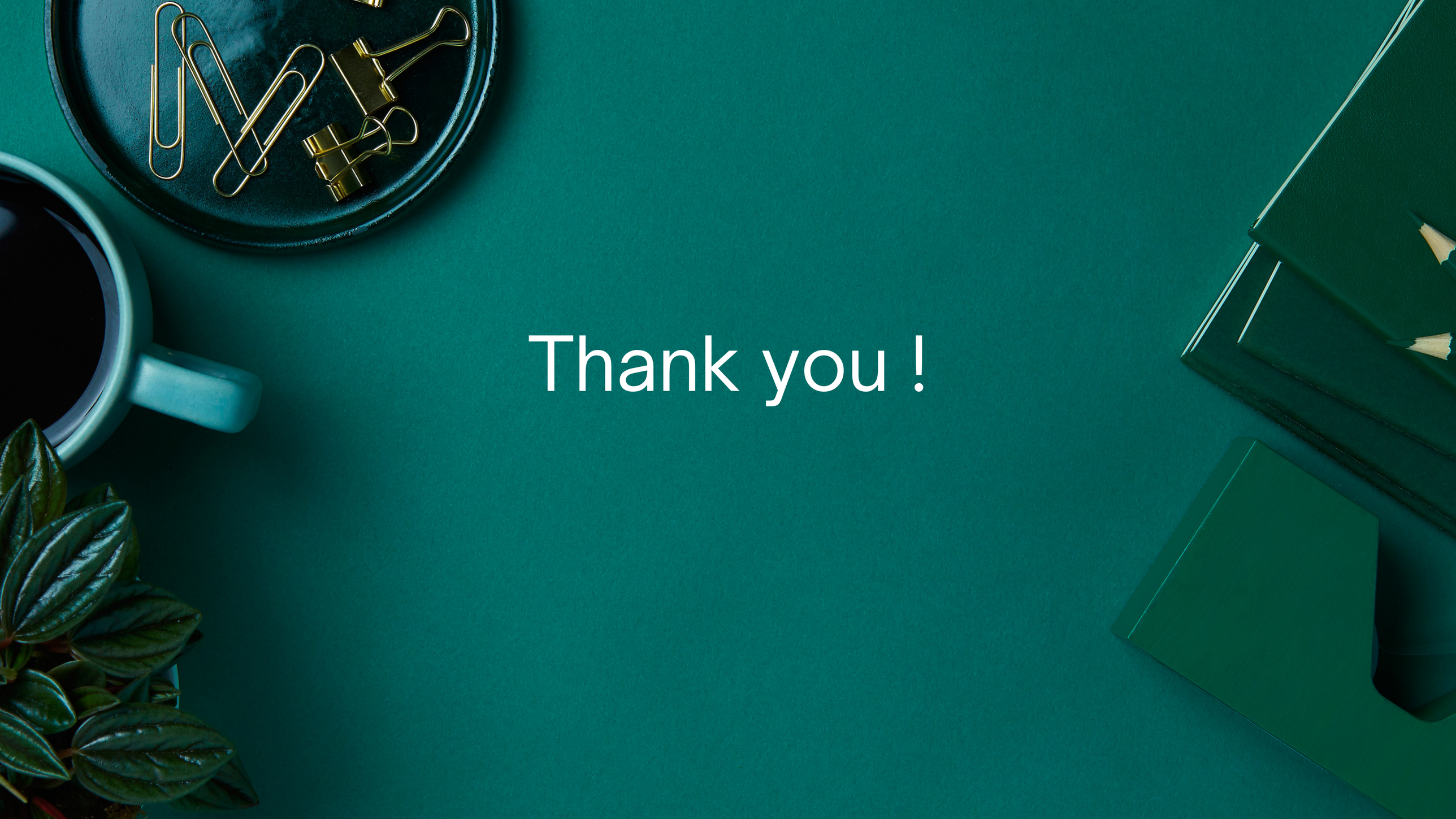


Case III

In this case, the height of the tree shrinks. If the target key lies in an internal node, and the deletion of the key leads to a fewer number of keys in the node (i.e. less than the minimum required), then look for the inorder predecessor and the inorder successor. If both the children contain a minimum number of keys then, borrowing cannot take place.

Again, look for the sibling to borrow a key. But, if the sibling also has only a minimum number of keys then, merge the node with the sibling along with the parent. Arrange the children accordingly (increasing order).




Thank you !