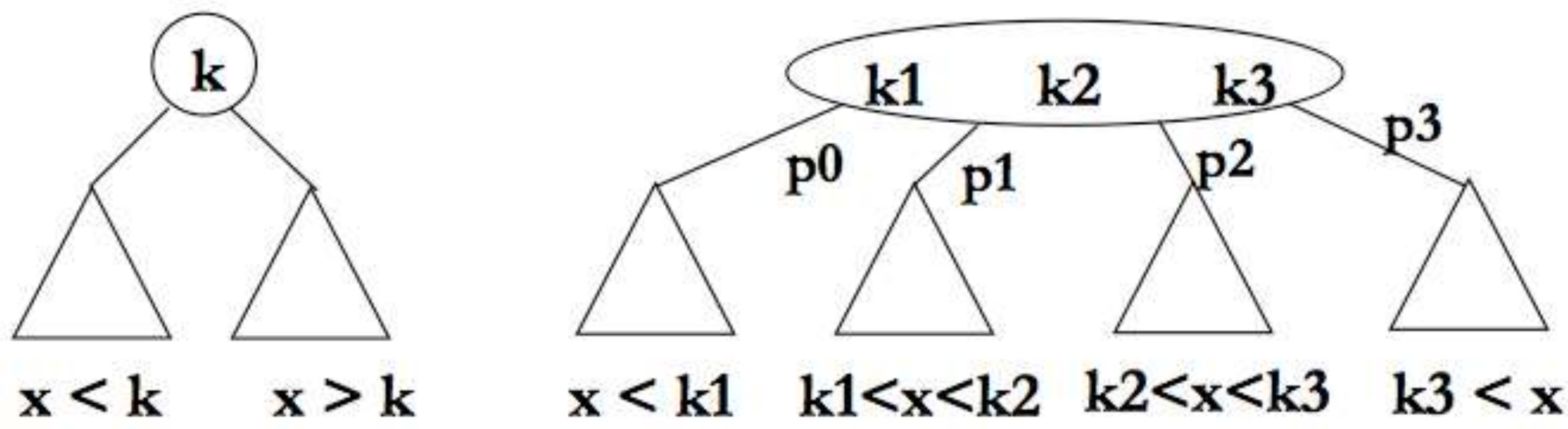
Lecture 8. B-Tree

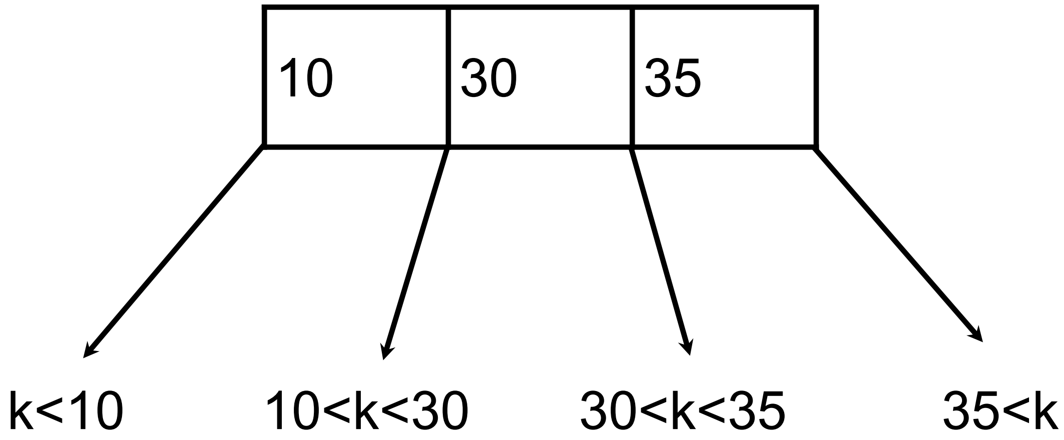
m-way search tree

- Binary Trees are not quite appropriate for data stored on disks
  - disk access is much slower than memory access
  - disk is partitioned into blocks (pages) and the access time of a word is the same as that of the entire block containing the word
  - we need to reduce the number of disk access

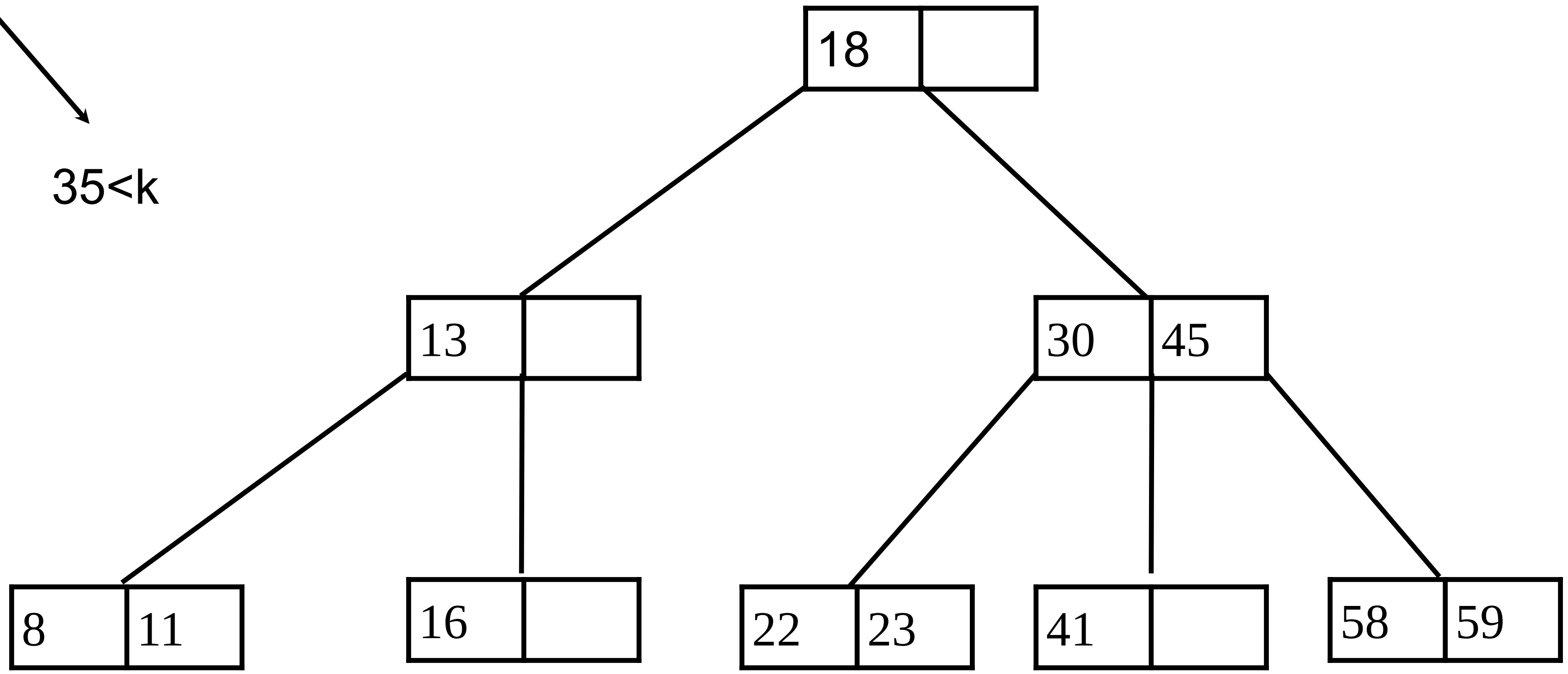
    → make each node of the tree wider (m-way search tree)



m-way search tree



m=4
(m: degree of node)



m=3 (2-3 tree)

B-Tree

- a B-tree of order m is an m-way search tree with the following properties
  - the root is either a leaf or has at least 2 children
  - all non-leaf nodes (except the root) have between $\lceil m/2 \rceil$ and m children
  - all leaves are at the same level
- for example,
  - when m=3, all internal nodes of B-tree have a degree of either 2 or 3 (2-3 tree)
  - when m=4, all internal nodes of B-tree have a degree of 2, 3, or 4 (2-3-4 tree)
- a B-tree of order 3 is 2-3 tree and a B-tree of order 4 is 2-3-4 tree
- a B-tree of order 2 is full binary tree

B-Tree

- a B-tree of height h
  - best case: the tree is splitting widely (m^h leaves)

$$h \le \log_m n = \frac{\log n}{\log m} = O(\log n)$$

  - worst case: the tree is splitting $\lceil m/2 \rceil$ ways

$$h \le \log_{\left\lceil \frac{m}{2} \right\rceil} n = \frac{\log n}{\log \left\lceil \frac{m}{2} \right\rceil} = O(\log n)$$

B-Tree: node structure

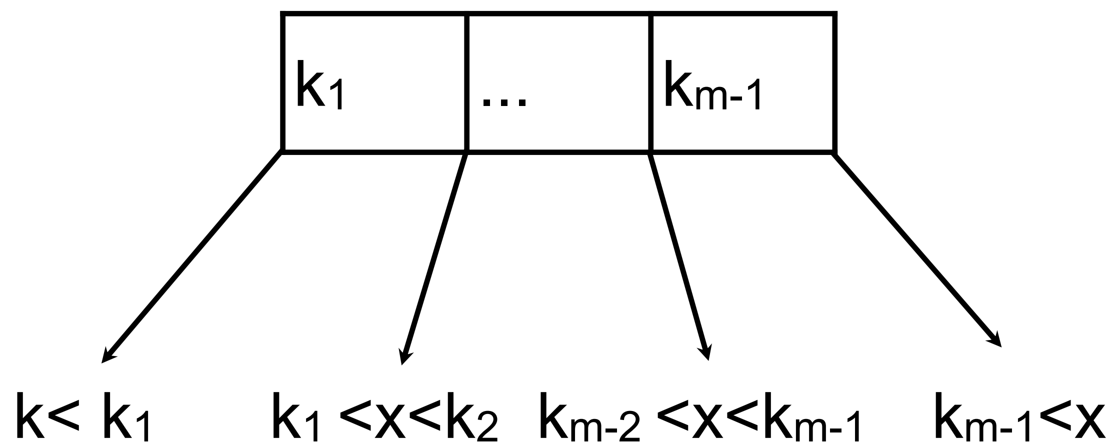
```
#define order 3

struct B_node {
   int    n_child;                /* number of children */
   B_node *child[order];   /* children pointers  */
   int   key[order-1];          /* keys  : to find  x  in array ,binary searching  -> O(log m)*/
}
```

search

- When we arrive an internal node with key $k_1 < k_2 ,... < k_{m-1}$, search for x in this list (either linearly or by binary search)
  - if you found x, you are done
  - otherwise, find the index i such that $k_i < x < k_{i+1}$ ($k_0 = -\infty$ and $k_m = \infty$), and recursively search the subtree pointed by p_i.

- Complexity = $\log m \cdot \log_m n = O(\log n)$

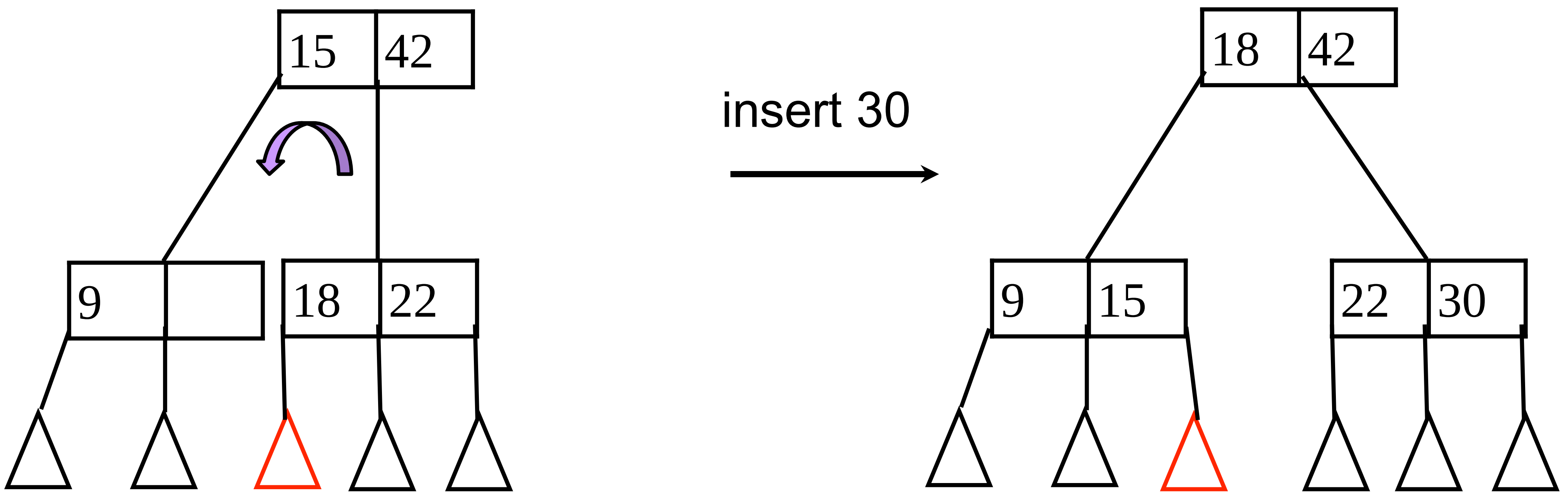


insertion

- find the appropriate leaf into which the node can be inserted
  - if the leaf is not full (< m-1 keys), insert it
  - if the node overflows, restore the balance
    - key rotation
    - node split

insertion

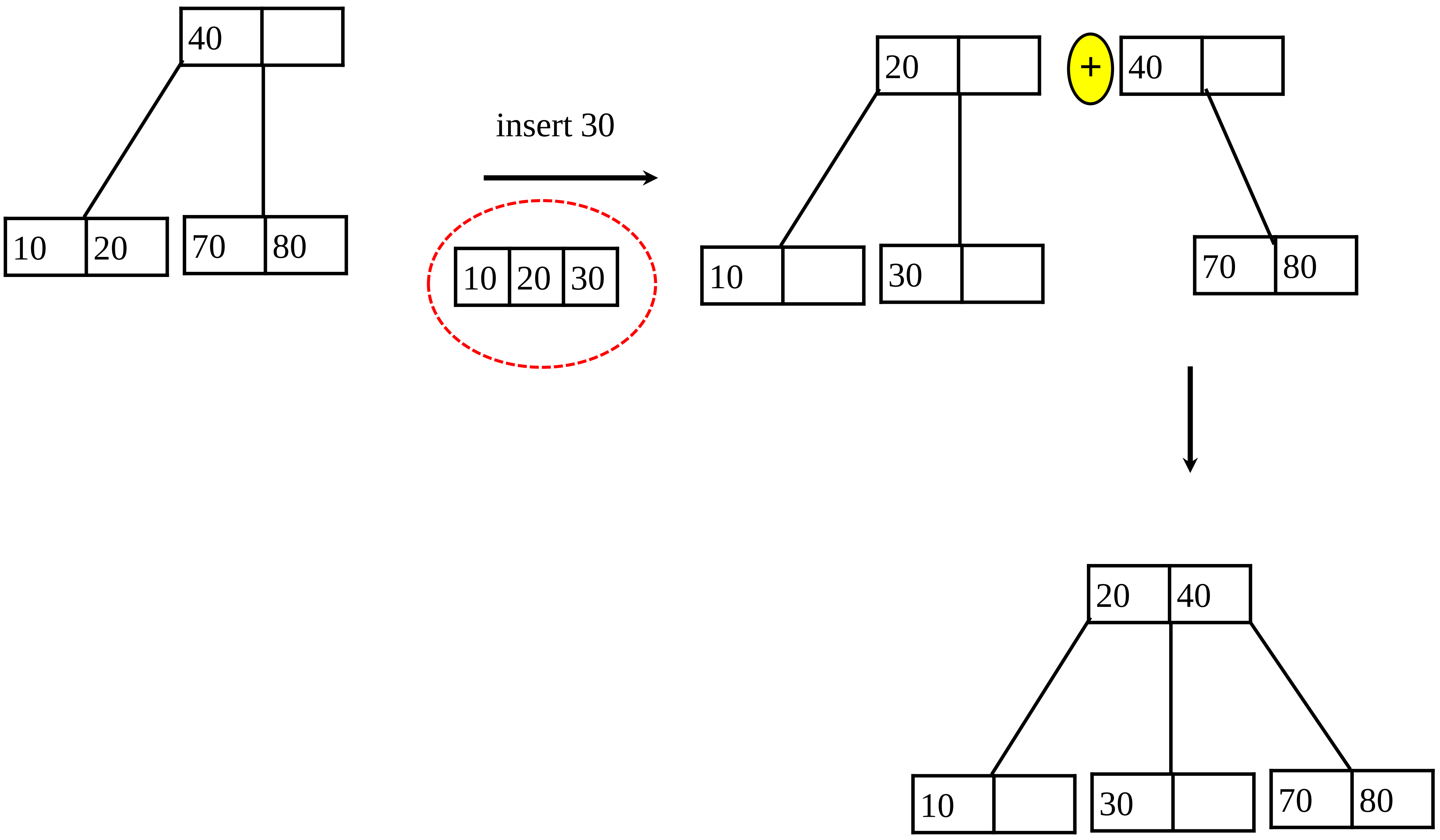
- key rotation: check for siblings for rotation



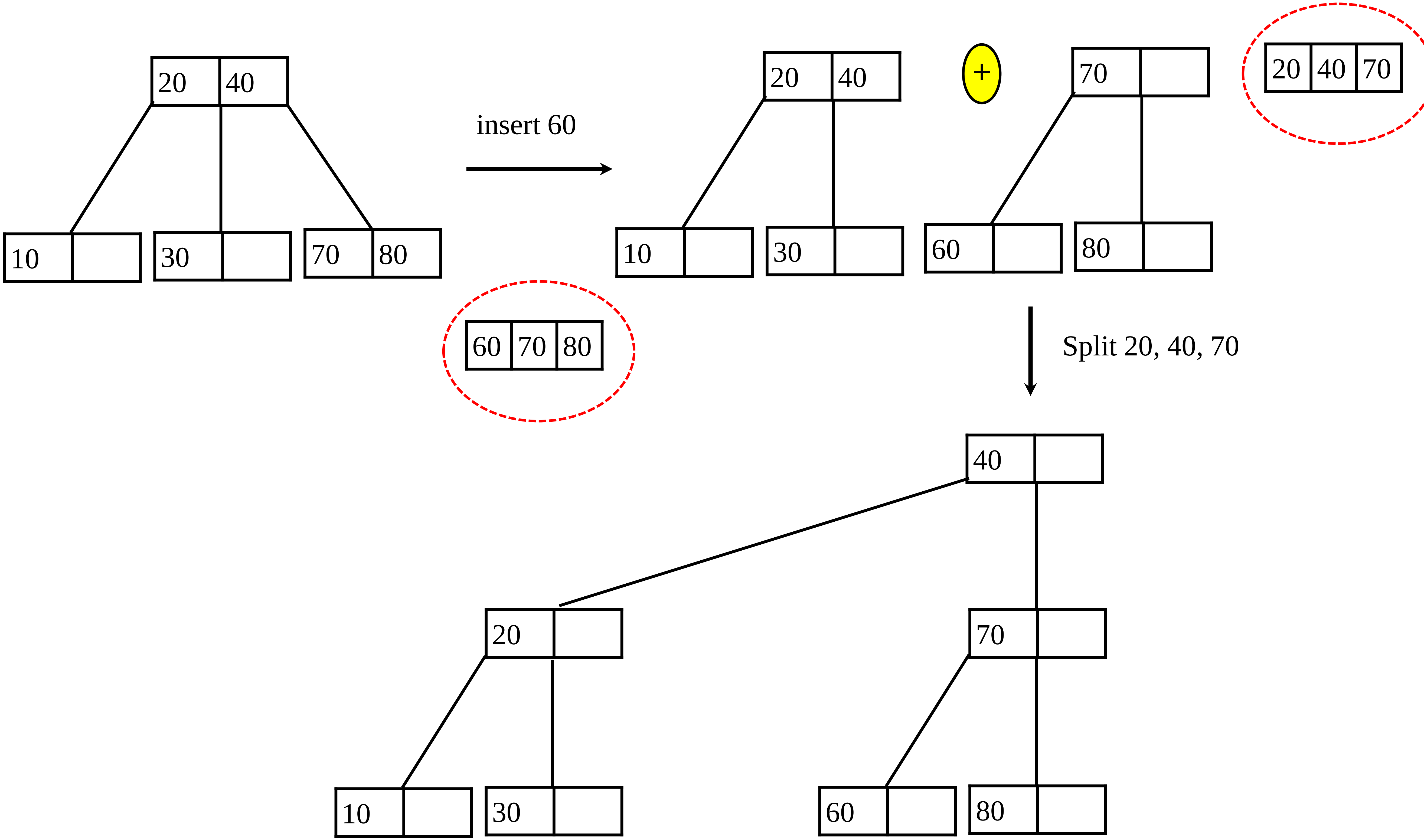
insertion

- node split
  - if we have a node with m keys after insertion, split the node into three groups
    - (a) a node with the keys from the first to $(\lceil m/2 \rceil - 1)^{th}$
    - (b) a node with the $\lceil m/2 \rceil^{th}$ key
    - (c) a node with the keys from $(\lceil m/2 \rceil + 1)^{th}$ to m^{th} keys
  - make (a) and (c) as new nodes and insert (b) to the parent
  - if the parent overflows, repeat the process
  - if the root overflows, create a new node with 2 children

insertion



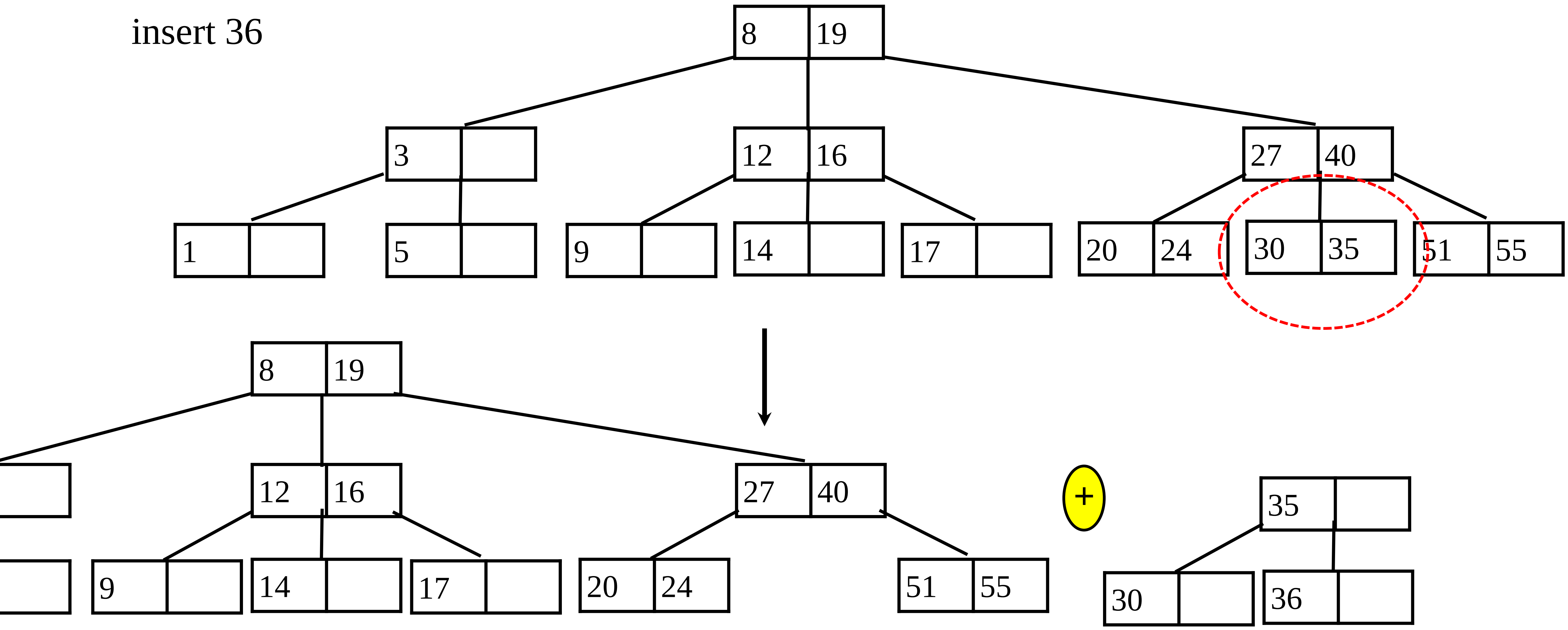
insertion

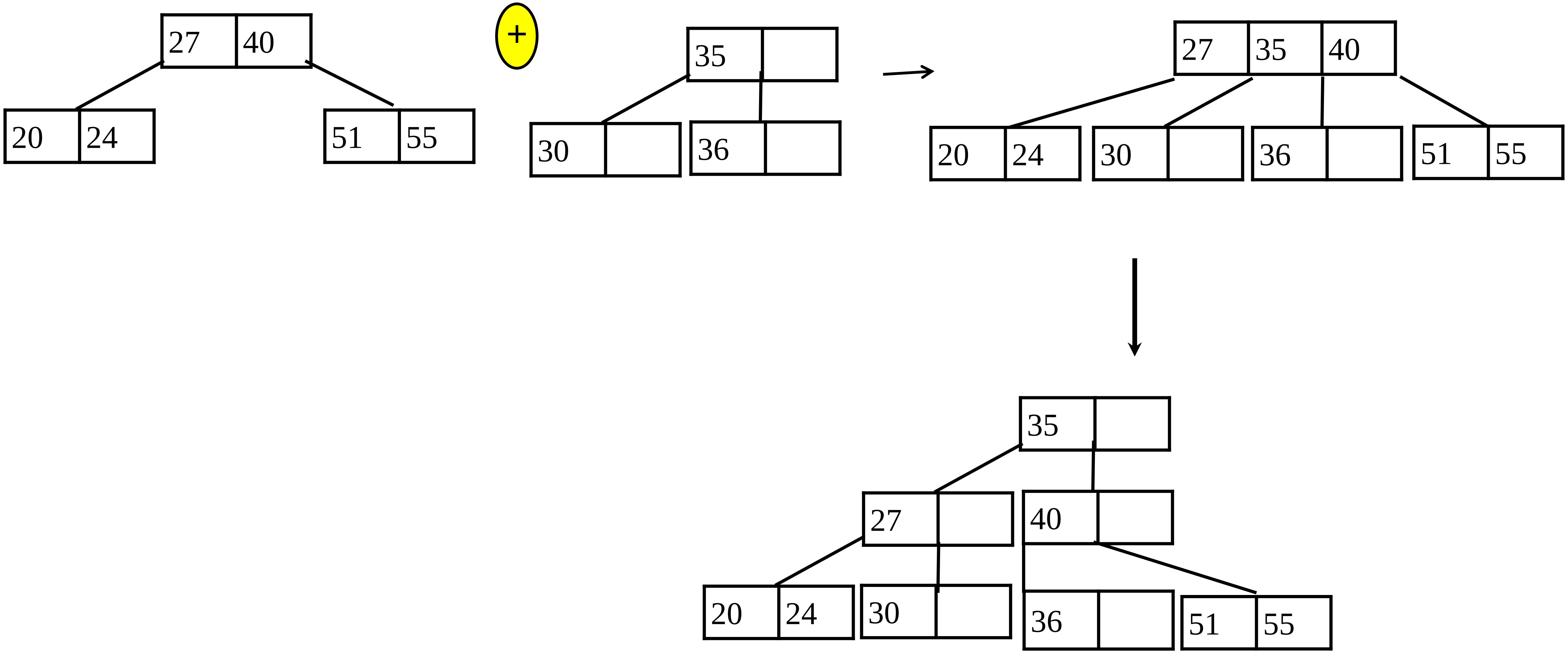


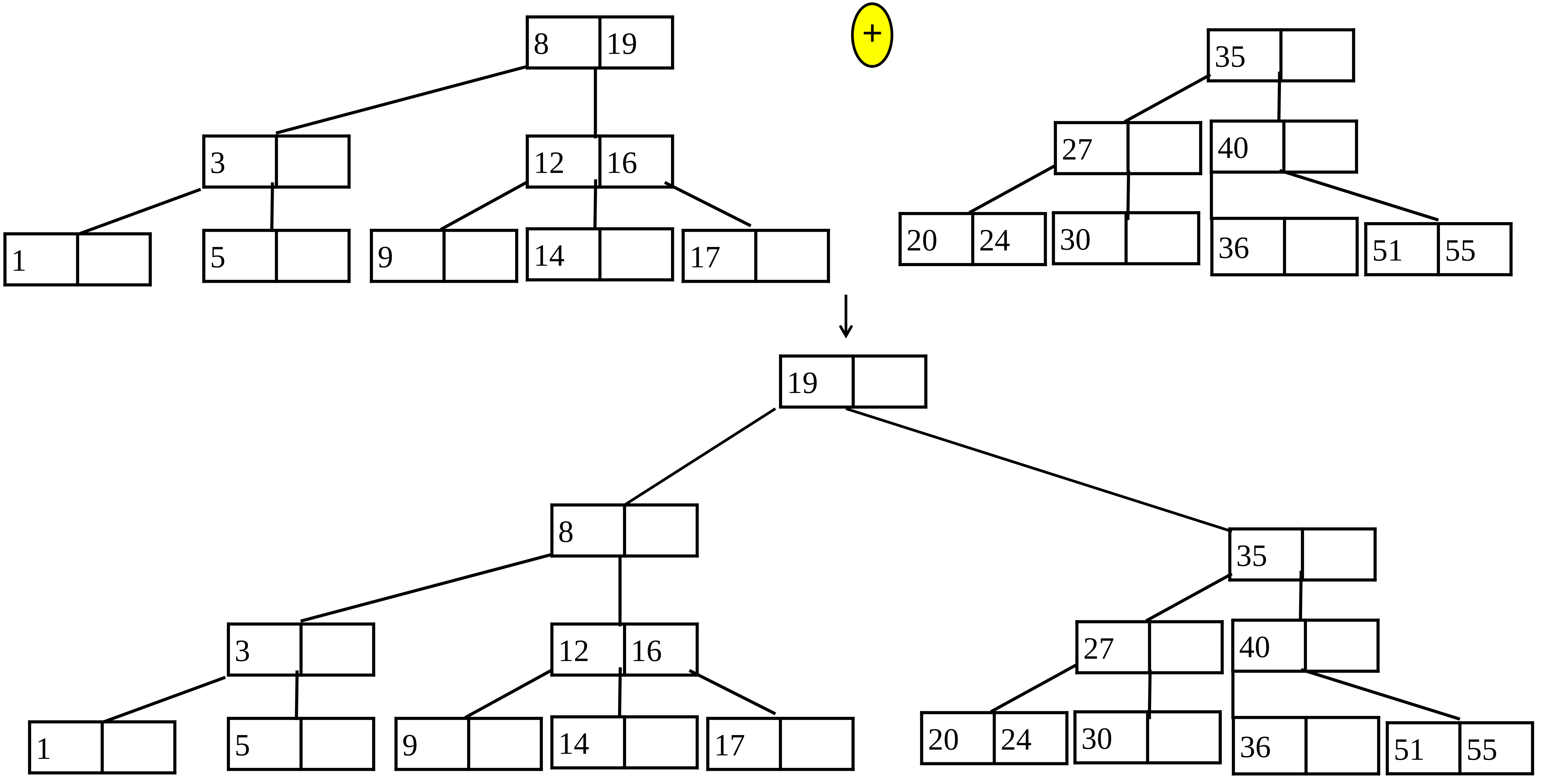
20
40
10
30
70
80
insert 60
60
70
80
20
40
10
30
+
70
60
80
20
40
70
Split 20, 40, 70
40
20
70
10
30
60
80

insertion

insert 36




27 40
20 24
51 55
+
35
30
36
27 35 40
20 24
30
36
51 55
35
27
40
20 24
30
36
51 55

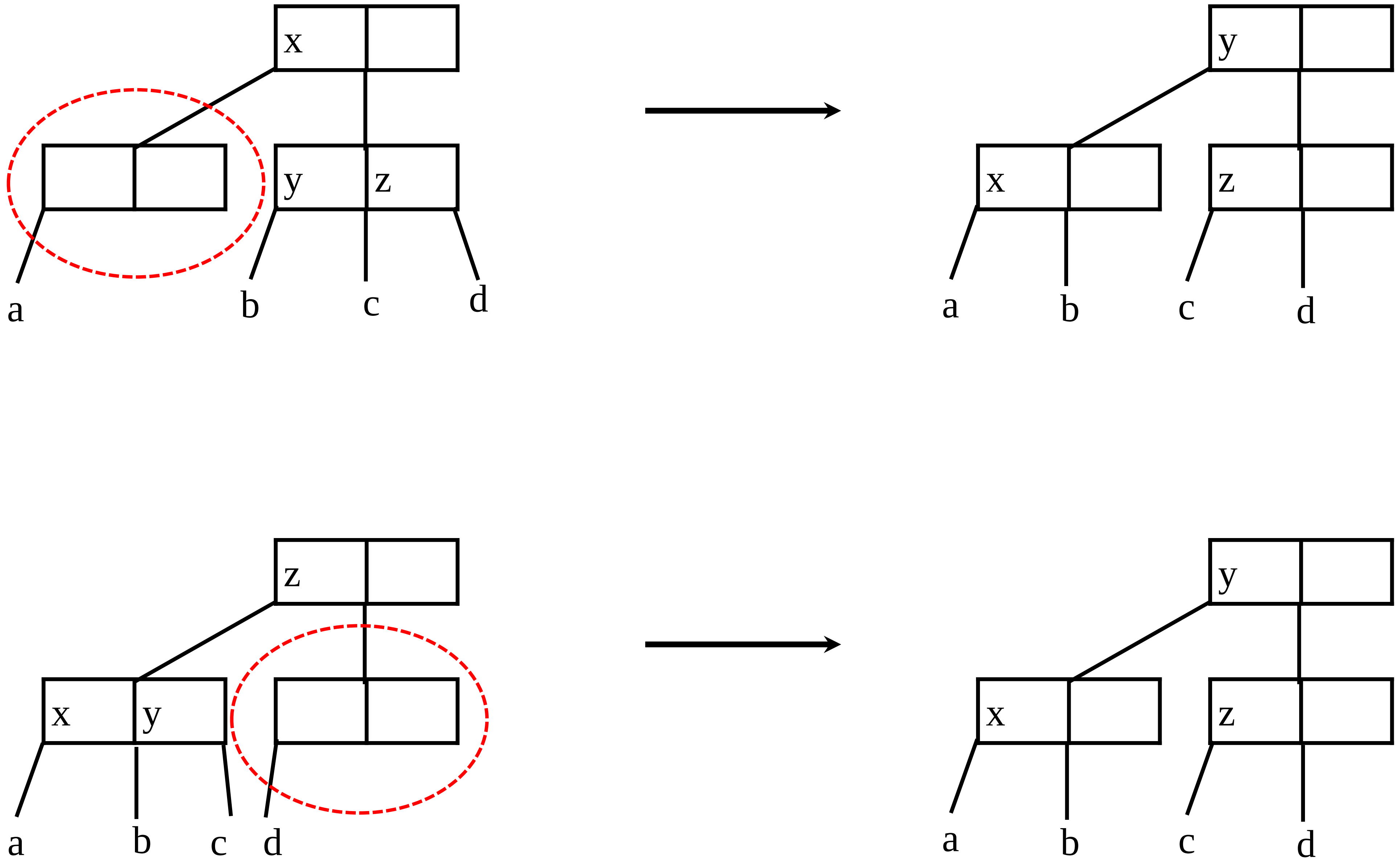

8 19
+
35
3
12 16
27
40
1
5
9
14
17
20 24
30
36
51 55
19
8
35
3
12 16
27
40
1
5
9
14
17
20 24
30
36
51 55

deletion

- find a suitable replacement which is the largest key in the left child (or the smallest in the right) and move it to fill the hole
  - key rotation
  - node merging

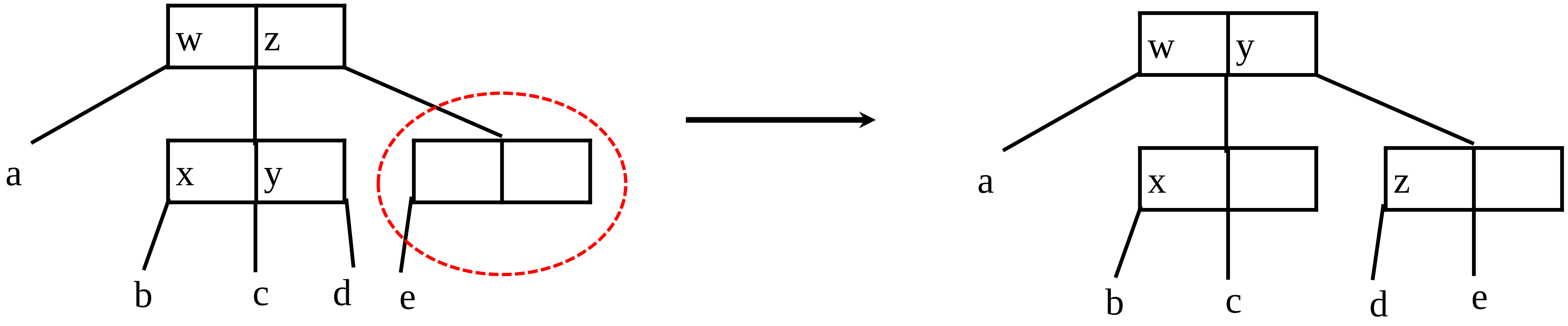
deletion

- key rotation



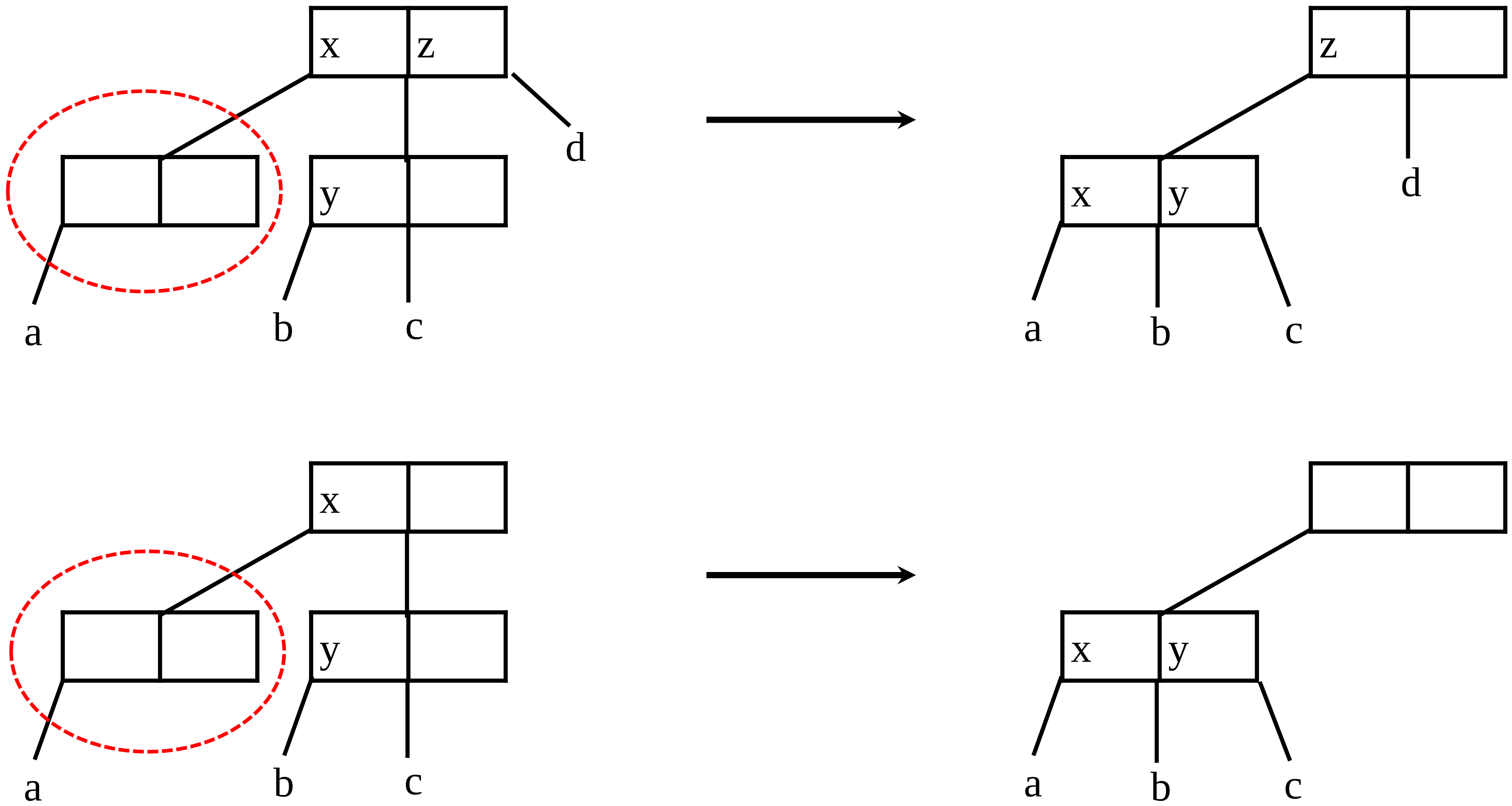
deletion

- key rotation

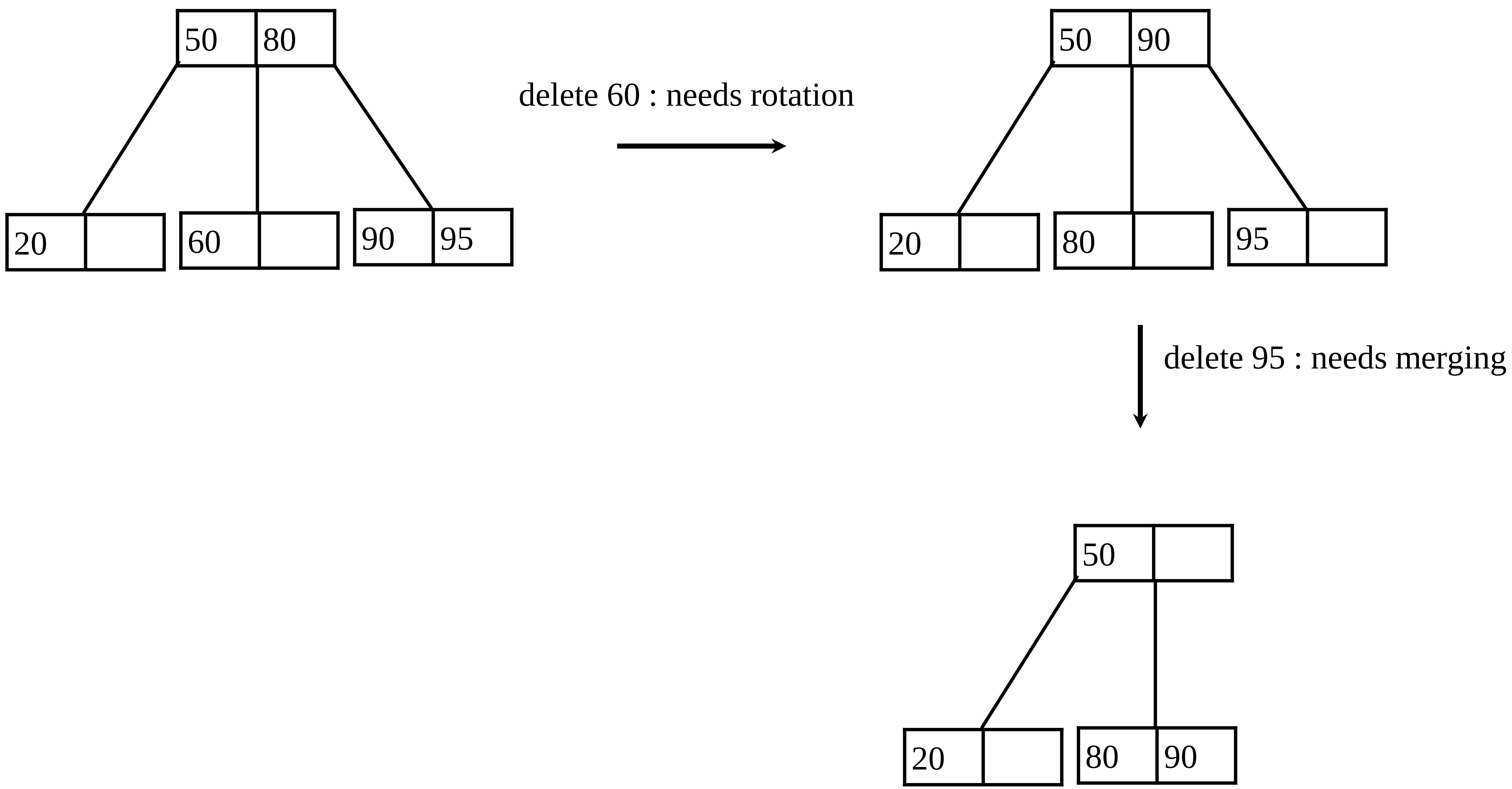


deletion

- node merging
  - no sibling that can be rotated
  - move down the intermediate node from the parent and put it in the new node
  - if this might cause underflow in the parent node, repeat the process

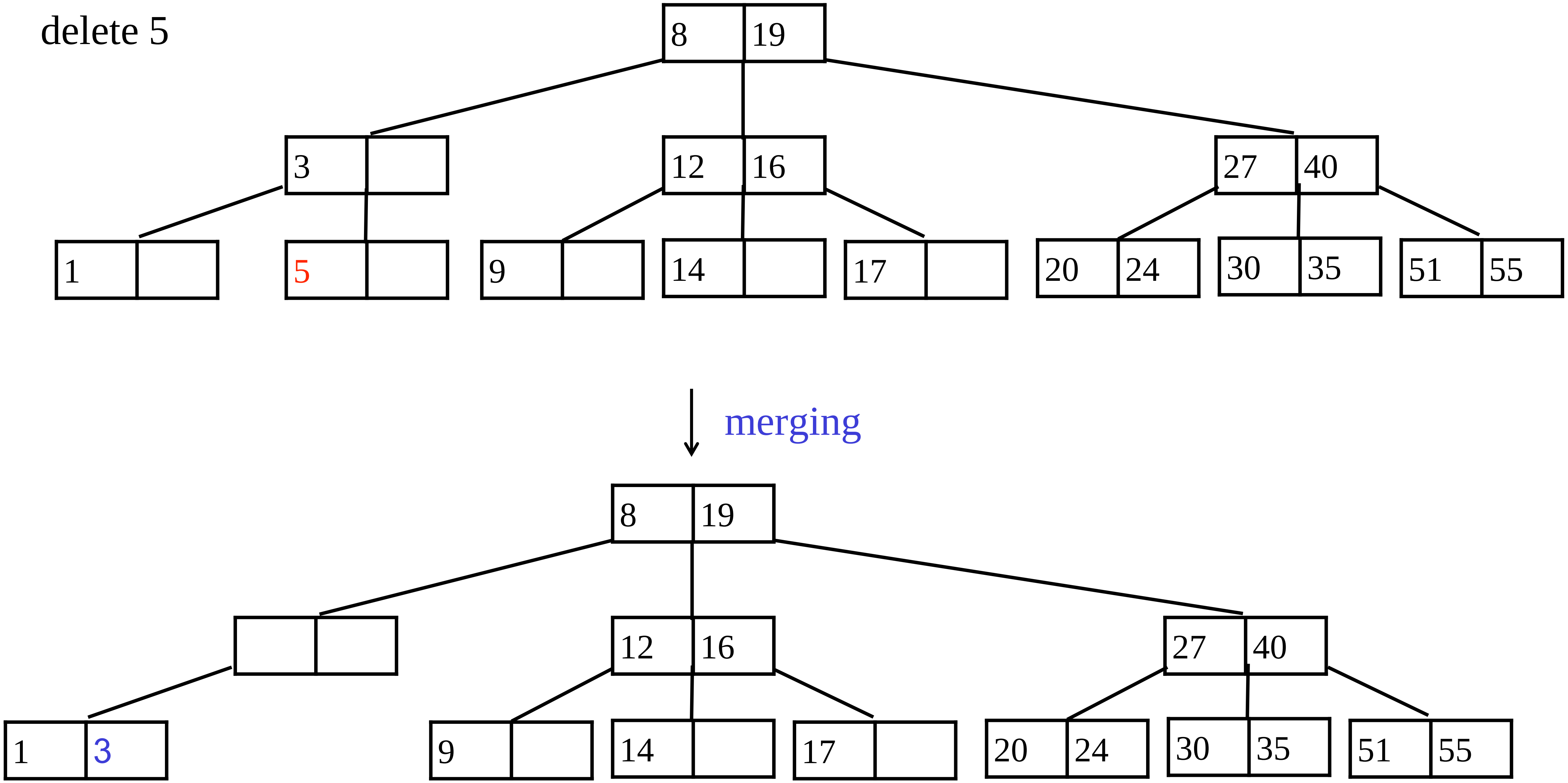


deletion

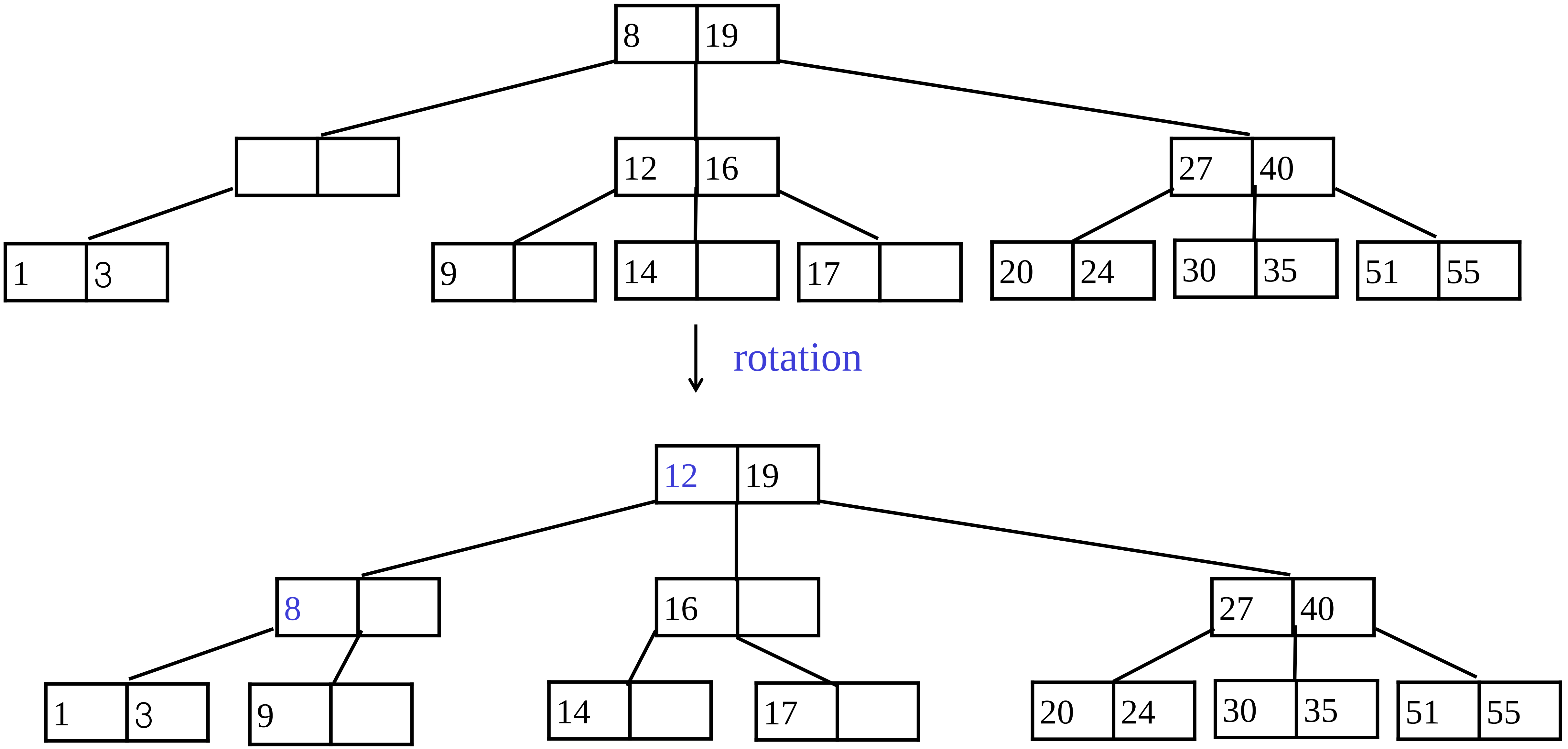

50
80
20
60
90
95
delete 60 : needs rotation
50
90
20
80
95
delete 95 : needs merging
50
20
80
90

deletion

delete 5



deletion



8
19
12
16
27
40
1
3
9
14
17
20
24
30
35
51
55
rotation
12
19
8
16
27
40
1
3
9
14
17
20
24
30
35
51
55

Use of B-tree in database system

- number of disk access is $O(\log_m n)$
- each disk access requires $O(\log m)$ overhead to determine the direction to branch, but this is done in main memory without a hard disk access, thus negligible.
- m can be determined as large as possible, but it must still be small enough so that an internal node can fit into one disk block.
- m is typically between 32 and 256.
- often one or two levels of internal nodes reside in main memory.