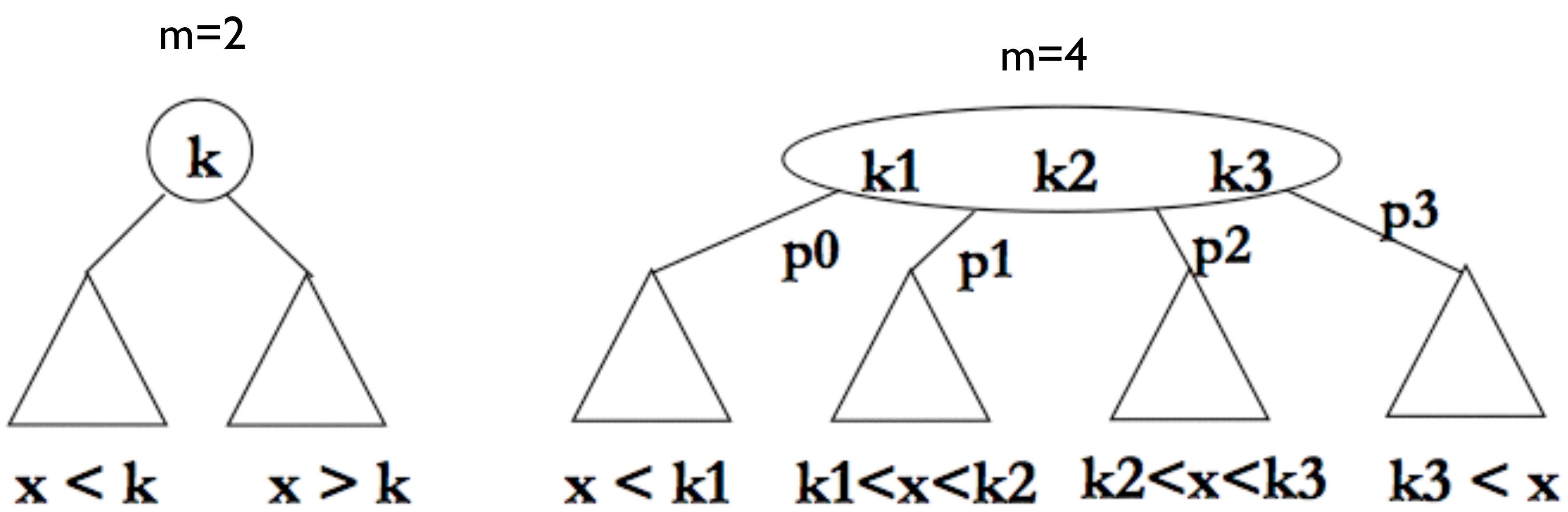
Data Structure:

B-Tree

m-way search tree

- Binary trees are not quite appropriate for data stored on disks
  - we assumed all data is kept in main memory
  - what if the data is kept in external disk?
    - 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

- In a tree of degree m and height h
  - the maximum number of nodes is $(m^h - 1) / (m-1)$

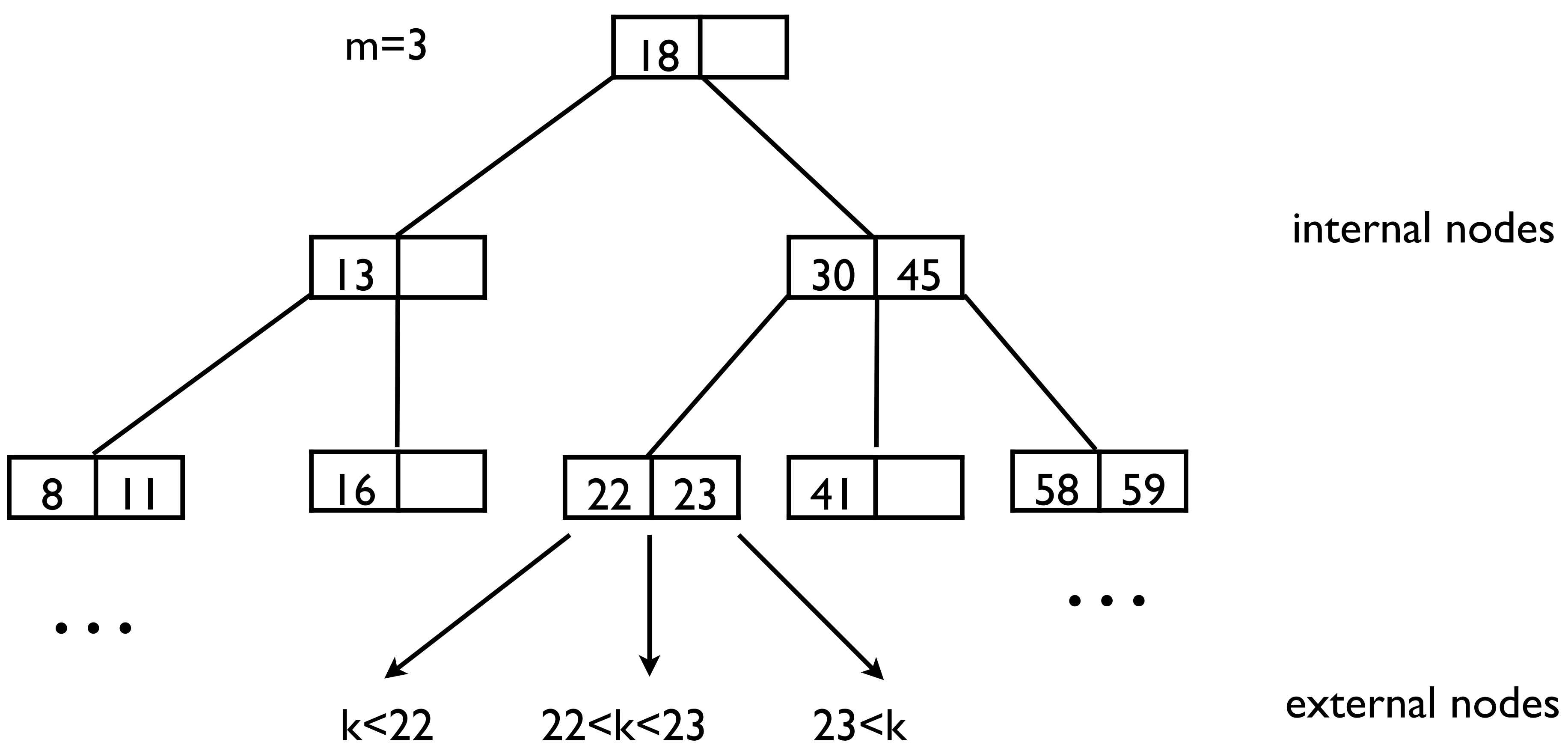
    $(m^0 + m^1 + m^2 + \ldots m^{h-1})$
  - the maximum number of elements in an m-way tree of height h is $m^h - 1$

    (since each node has at most m-1 elements)

- a binary tree with h=3 has 7 elements in the tree
- a 200-way tree with h=3 has $200^3 - 1 = 8 * 10^6 - 1$ nodes

B-Tree

- a B-tree of order m is an m-way search tree with the following properties
  - the root has at least 2 children
  - each node has upto m-1 keys
  - all external nodes are at the same level (perfectly balanced)
  - all internal nodes (except the root) have between $\lceil m/2 \rceil$ and m children
    - 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)



B-Tree

- a B-tree of height h
  - best case: the tree is splitting widely

    n = m^h -1

    h = $\lceil \log_m (n+1) \rceil$

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

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

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

B-Tree: node structure

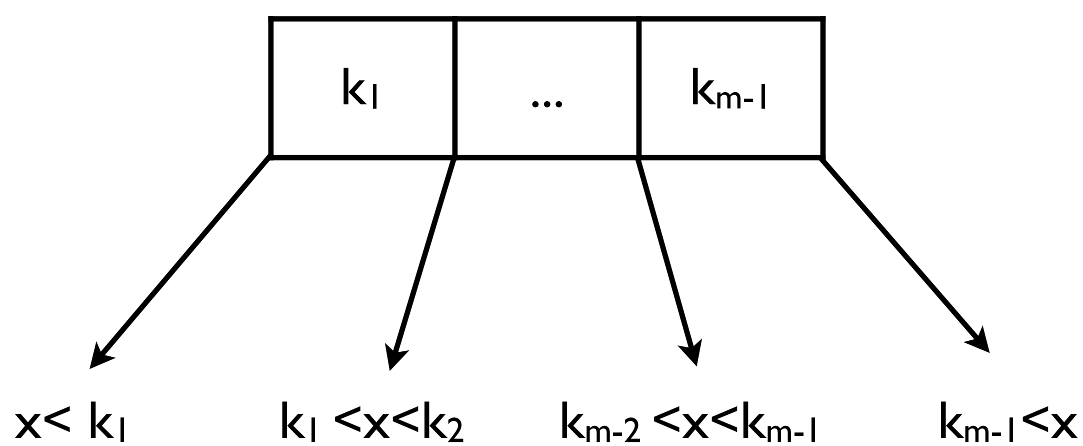
```
#define order 3

struct B_node {
        int    order;              /* number of children */
        B_node *child[order];   /* children pointers  */
        int   key[order-1];         /* keys              */
}
```

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 $k_i < x < k_{i+1}$, and recursively search the subtree pointed by p_i.

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

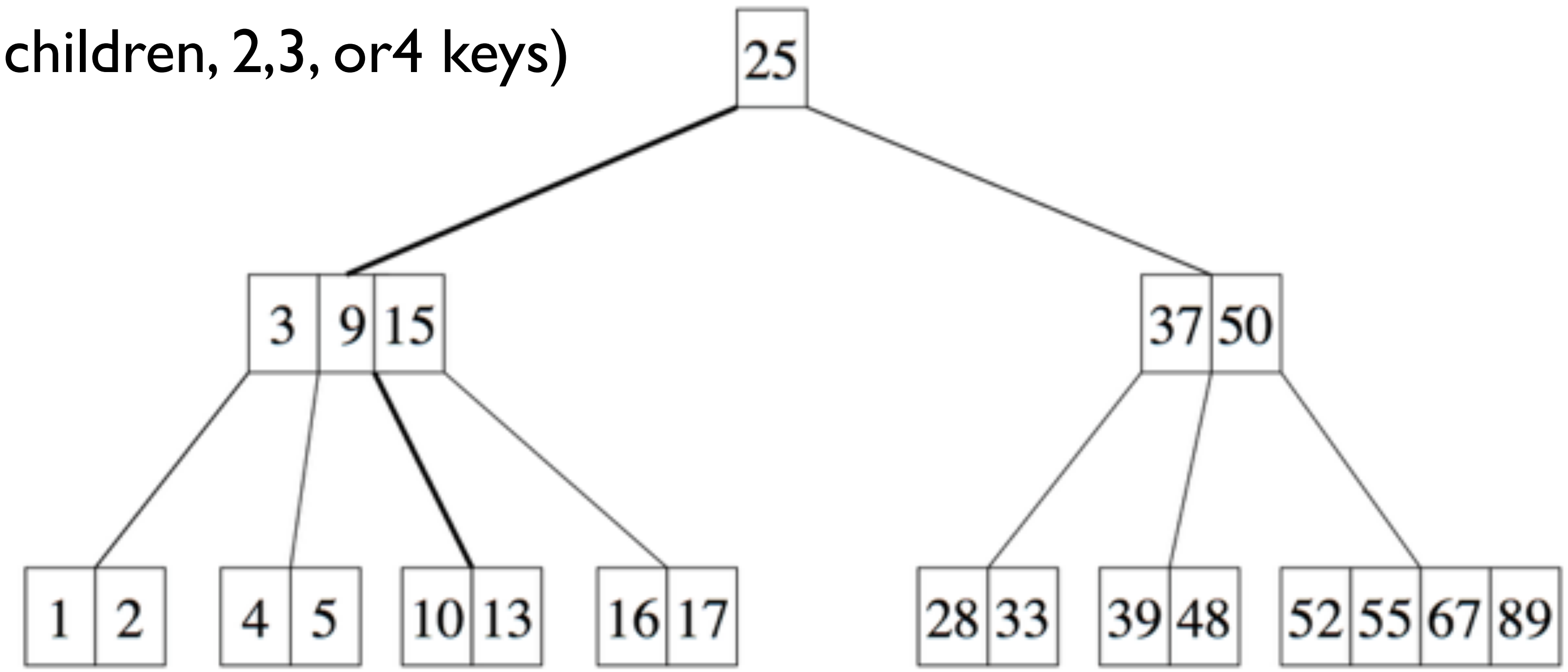


insertion

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

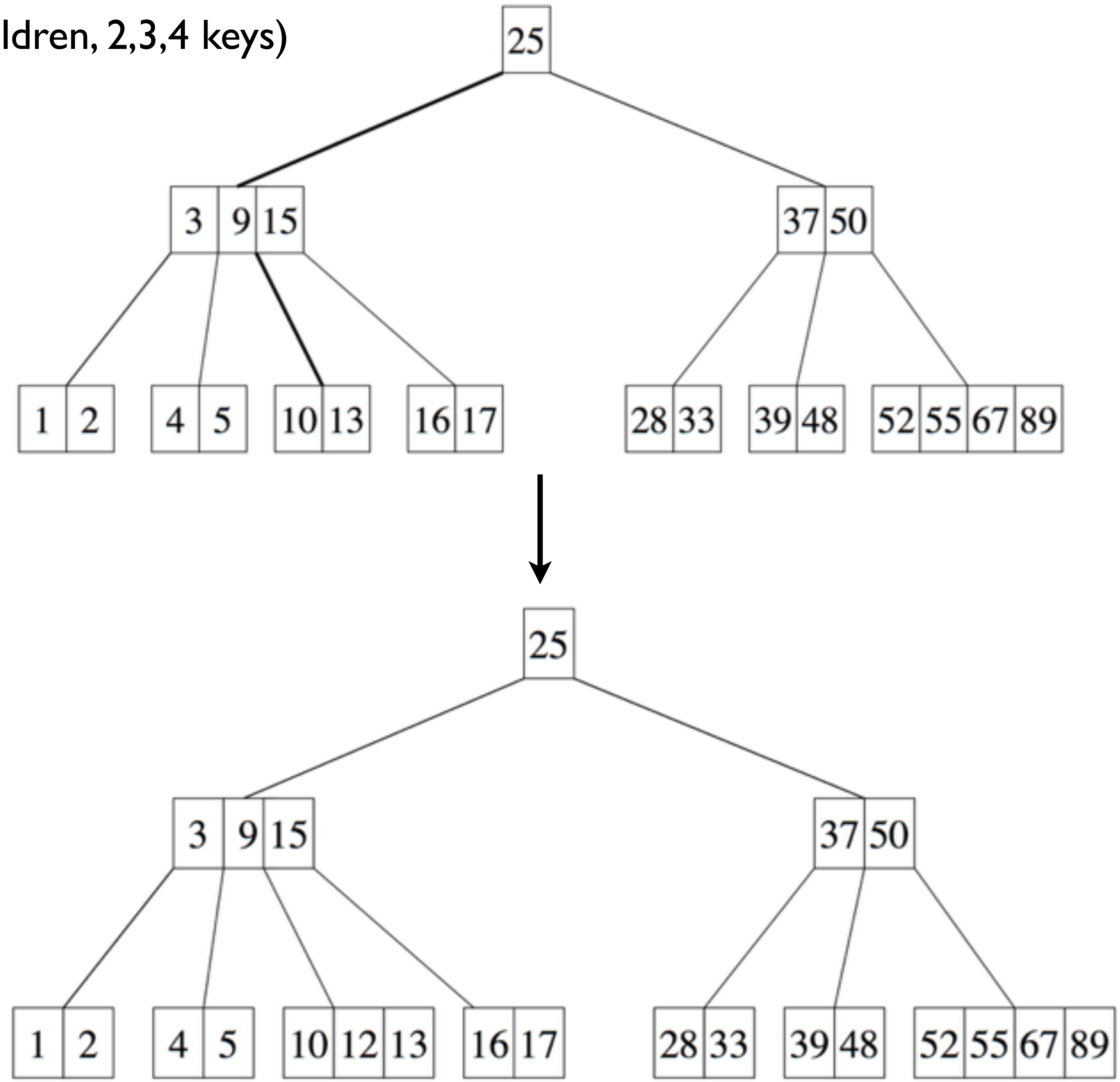
insertion

m= 5 (3,4,or 5 children, 2,3, or4 keys)
insert 12



insertion

m= 5 (3,4,5 children, 2,3,4 keys)
insert 12

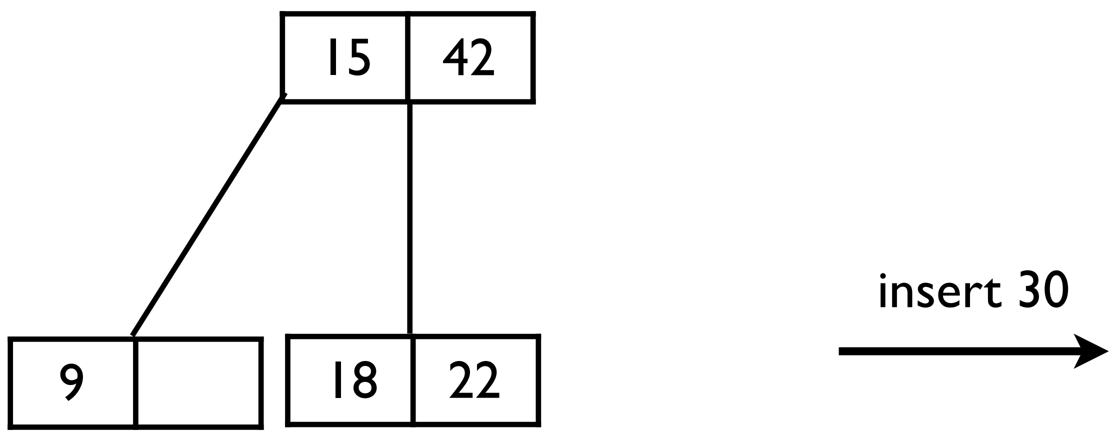


insertion

- 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 (if there is a space in the sibling node)
    - node split

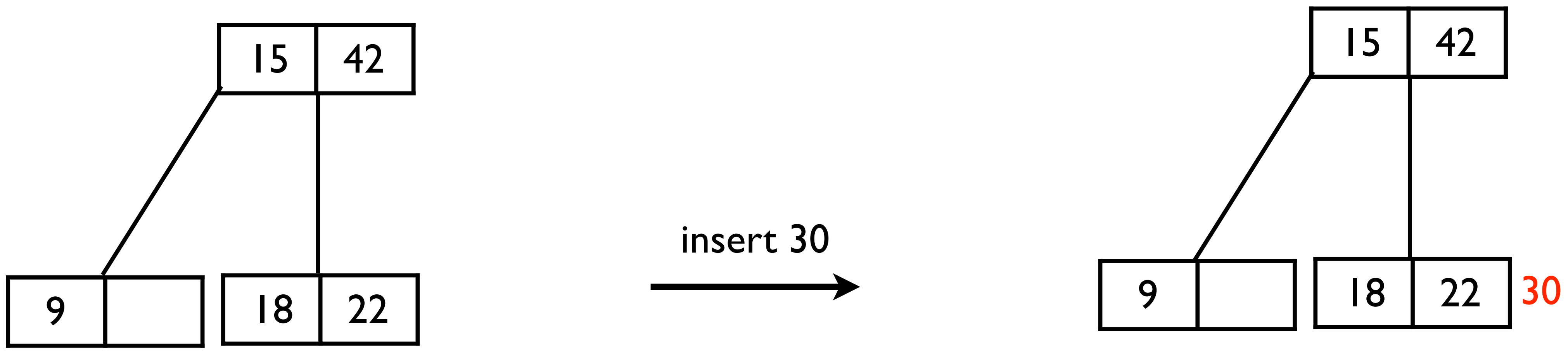
insertion

- key rotation: check for siblings for rotation into the B-tree of m=3



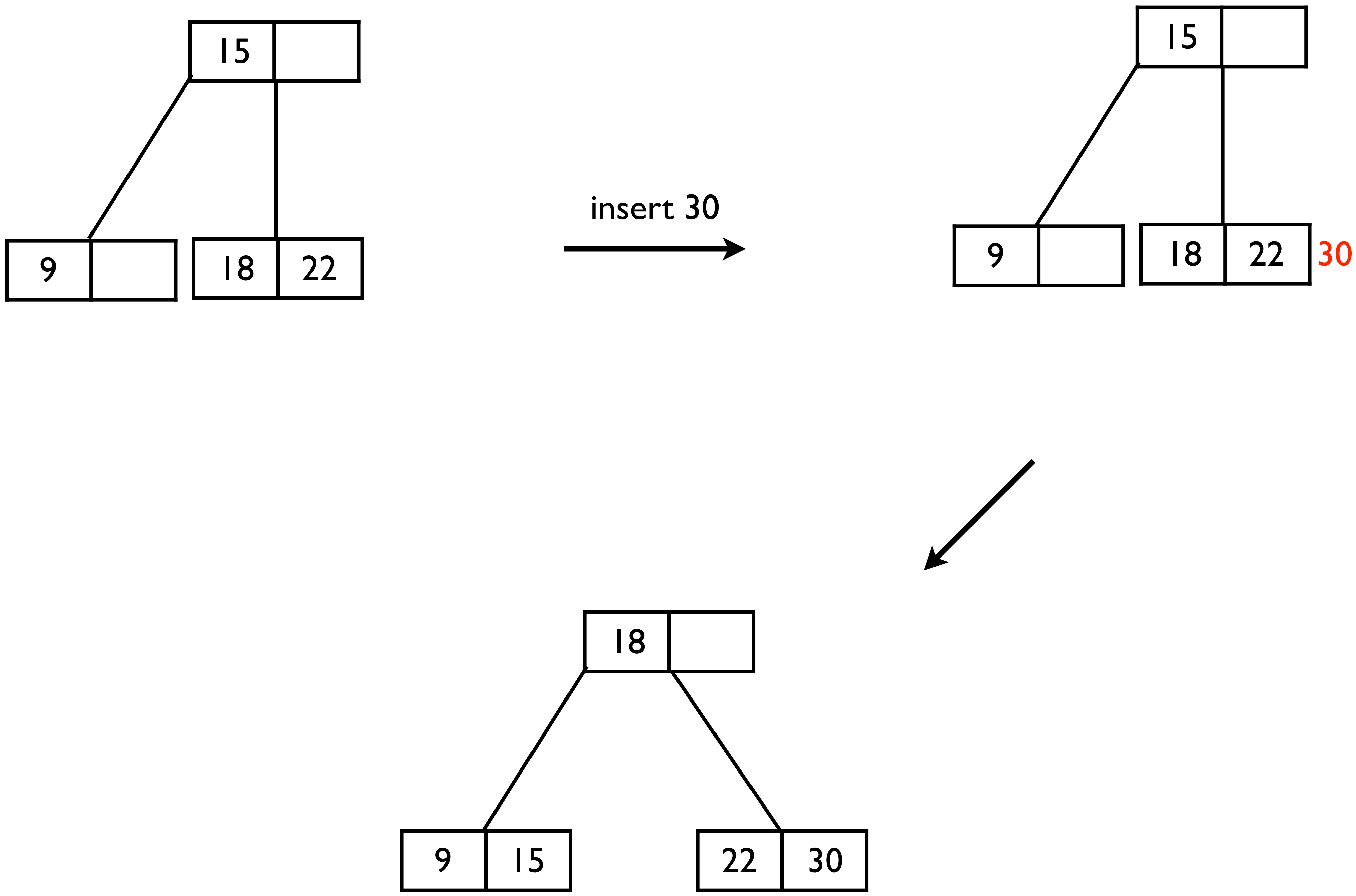
insertion

- key rotation: check for siblings for rotation into the B-tree of m=3

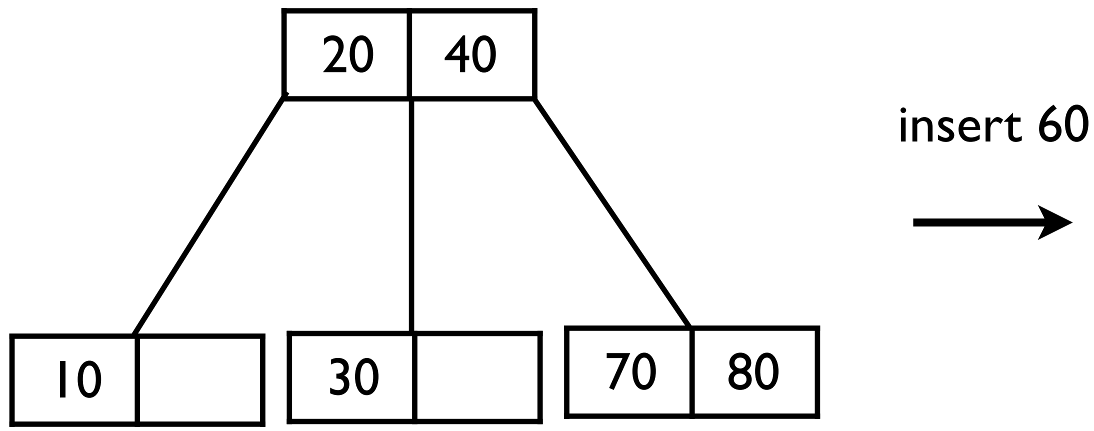


insertion

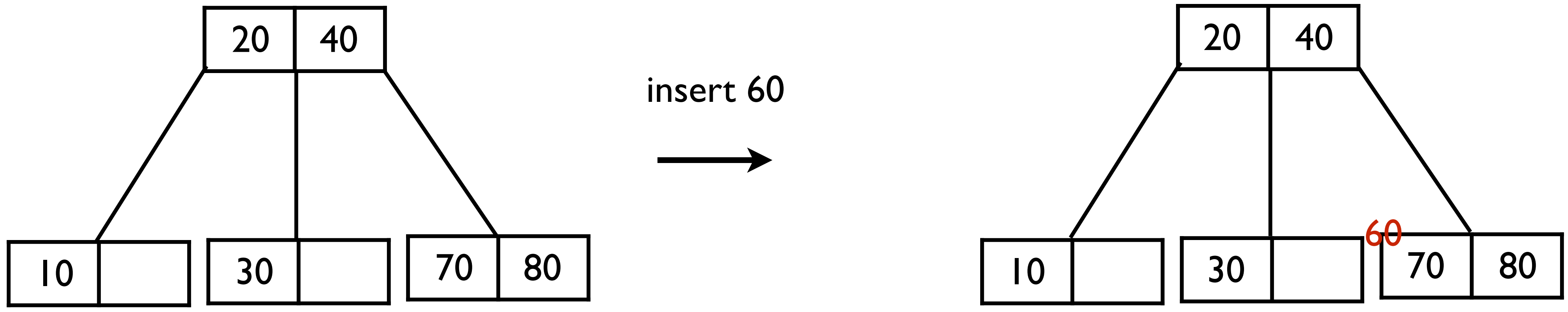
- key rotation: check for siblings for rotation into the B-tree of m=3



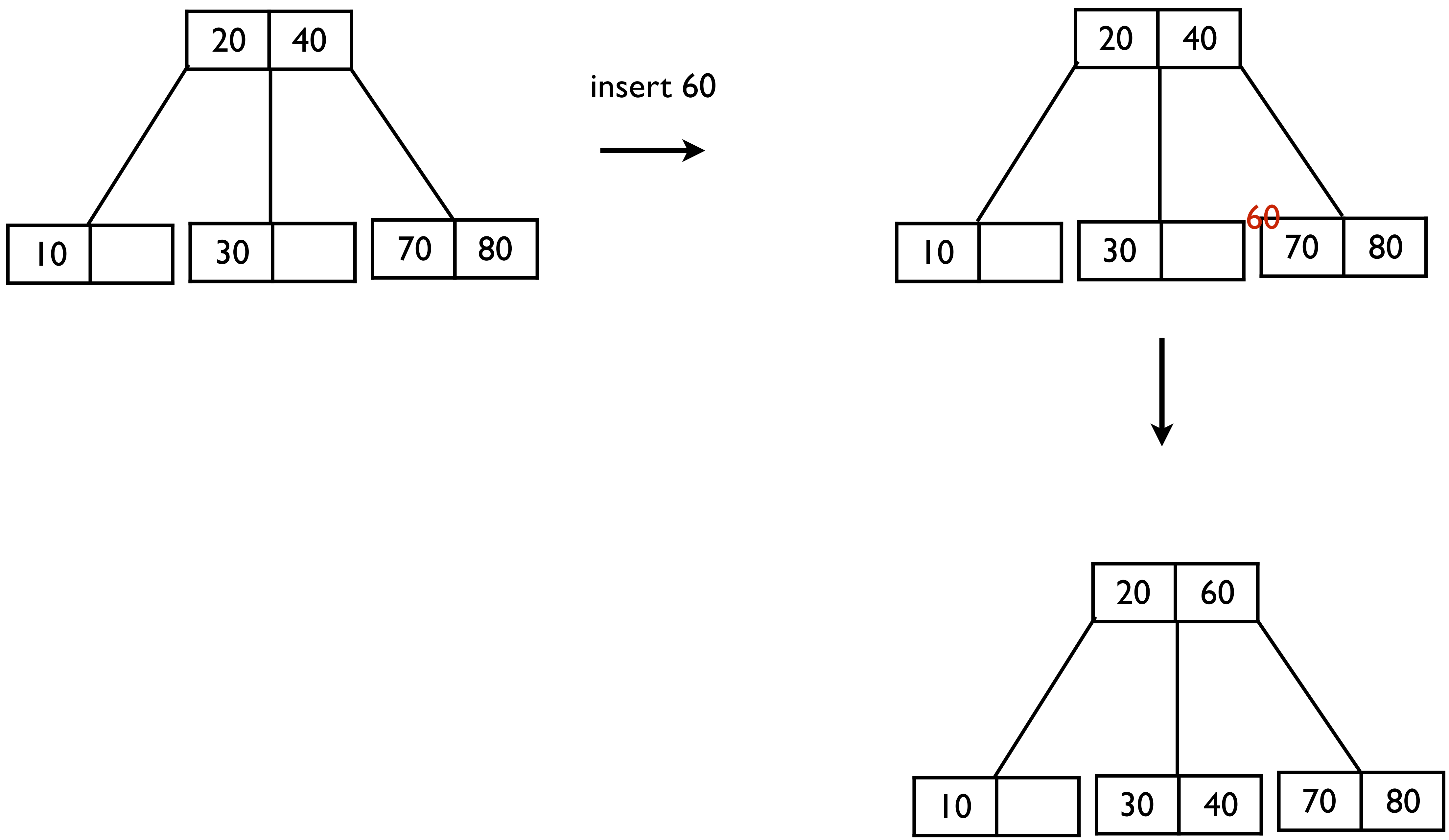
insertion



insertion



insertion

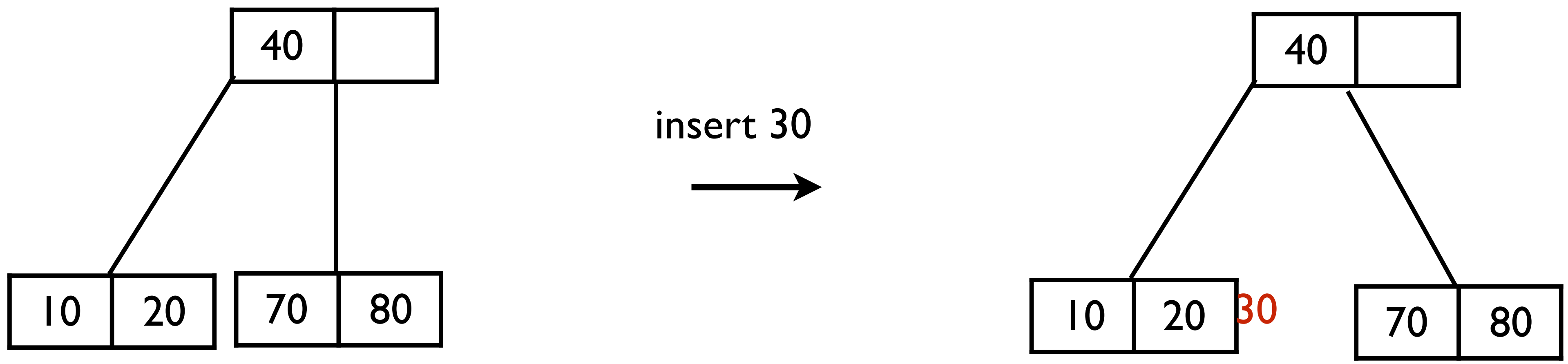


insertion

- node split
  - if we have a node with m keys after insertion (overflow), split the node into three groups
    - (a) a node with the keys smaller than the middle key
    - (b) a node with the middle key
    - (c) a node with the keys greater than the middle key
  - make (a) and (c) as new nodes and push (b) to the parent
  - if the parent overflows, repeat the process
  - if the root overflows, create a new node with 2 children

insertion

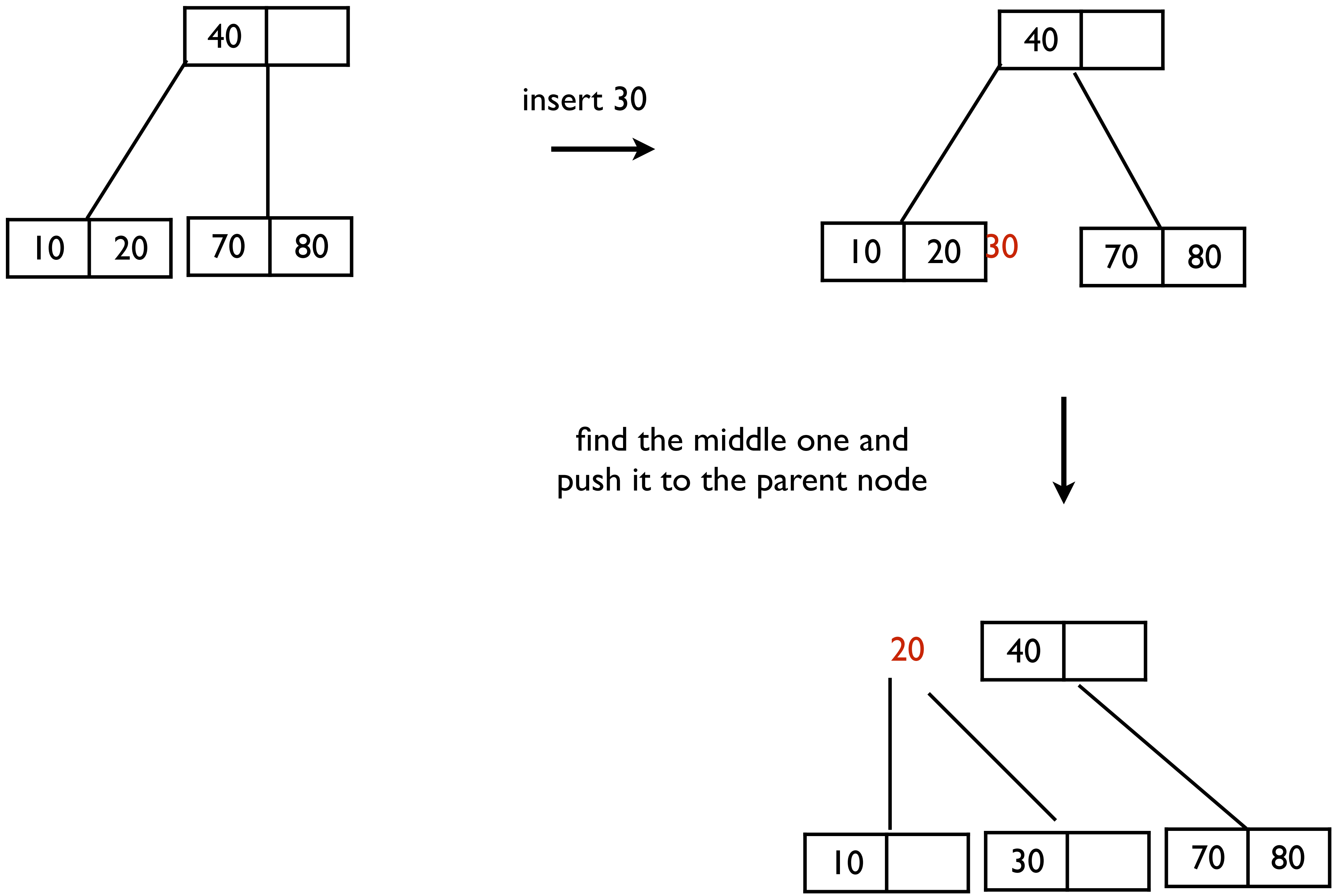
m= 3 (2,3 children, 1,2 keys)
insert 30



insertion

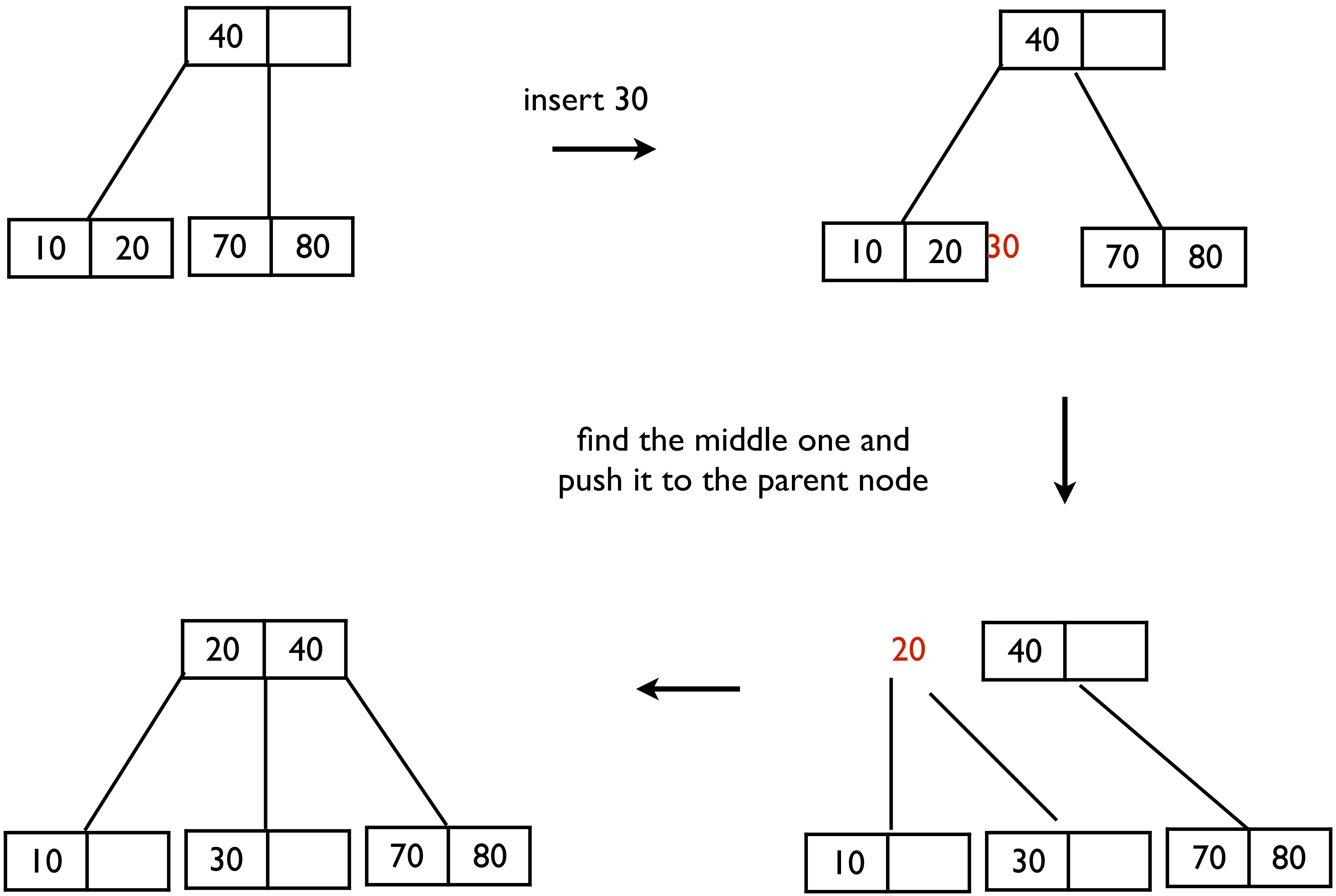
m= 3 (2,3 children, 1,2 keys)

insert 30



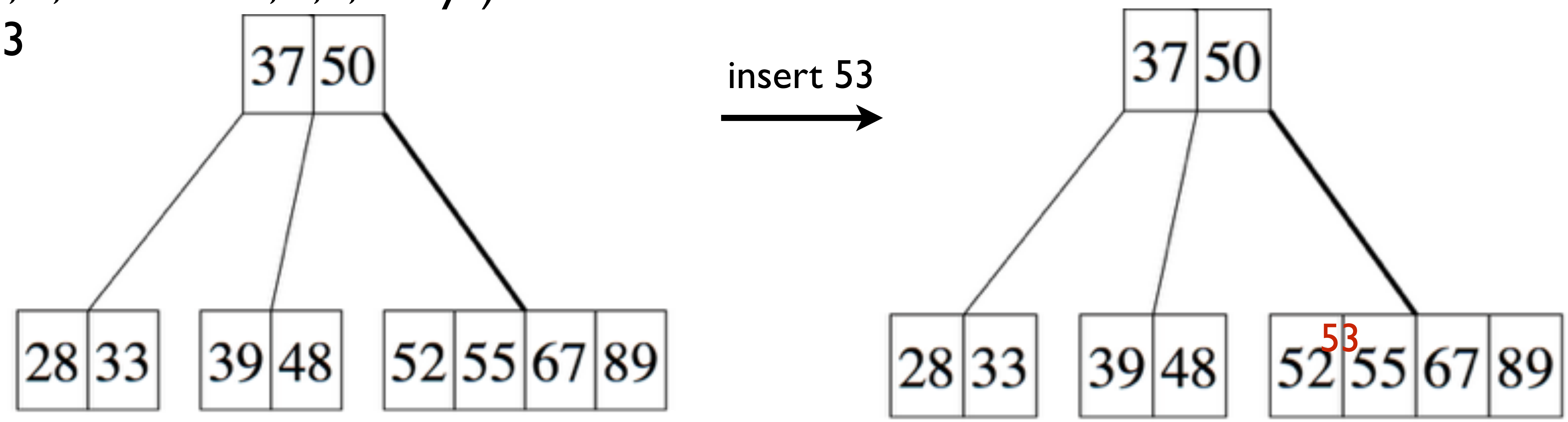
insertion

m= 3 (2,3 children, 1,2 keys)
insert 30



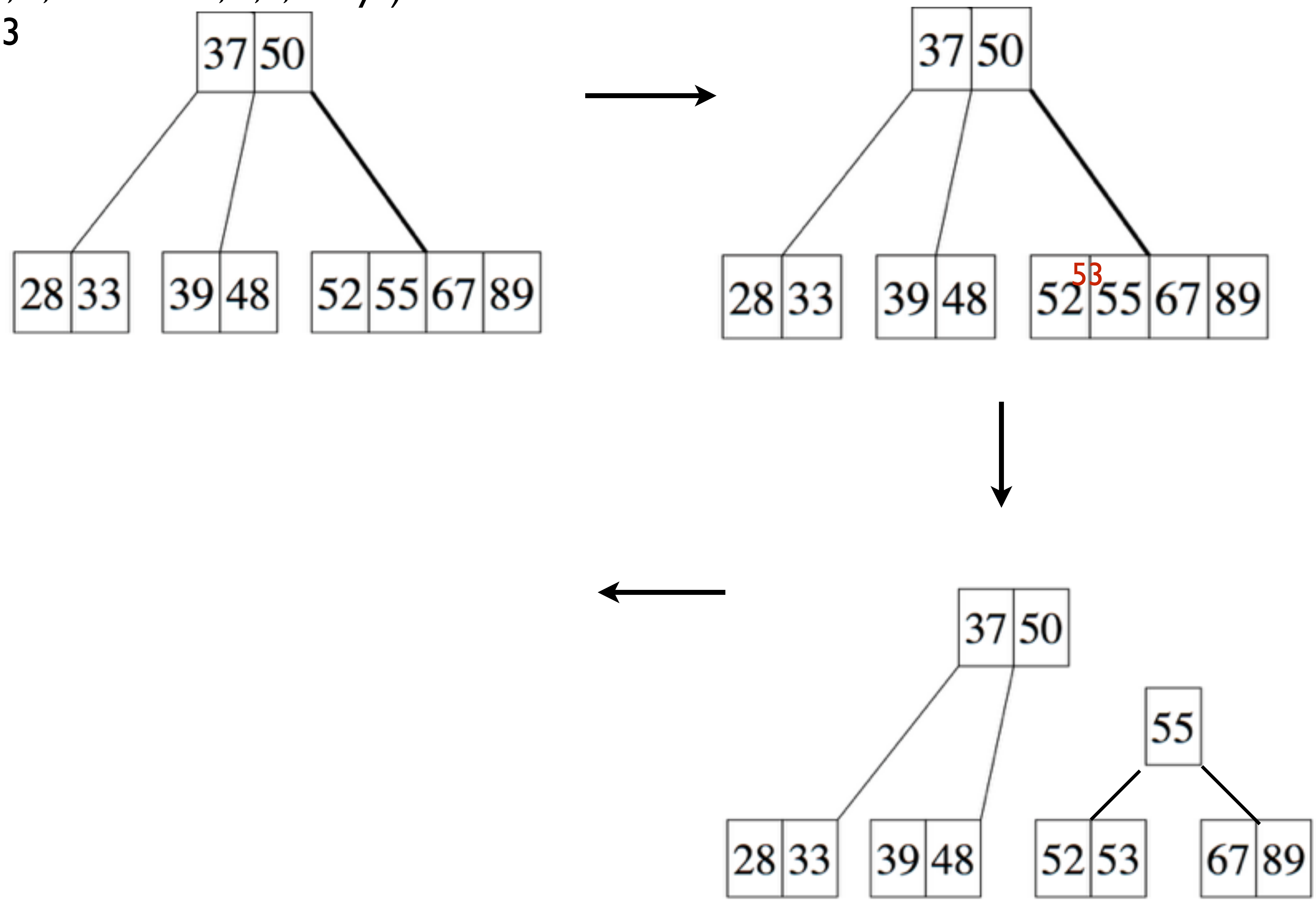
insertion

m= 5 (3, 4, 5 children, 2,3,4 keys)
insert 53



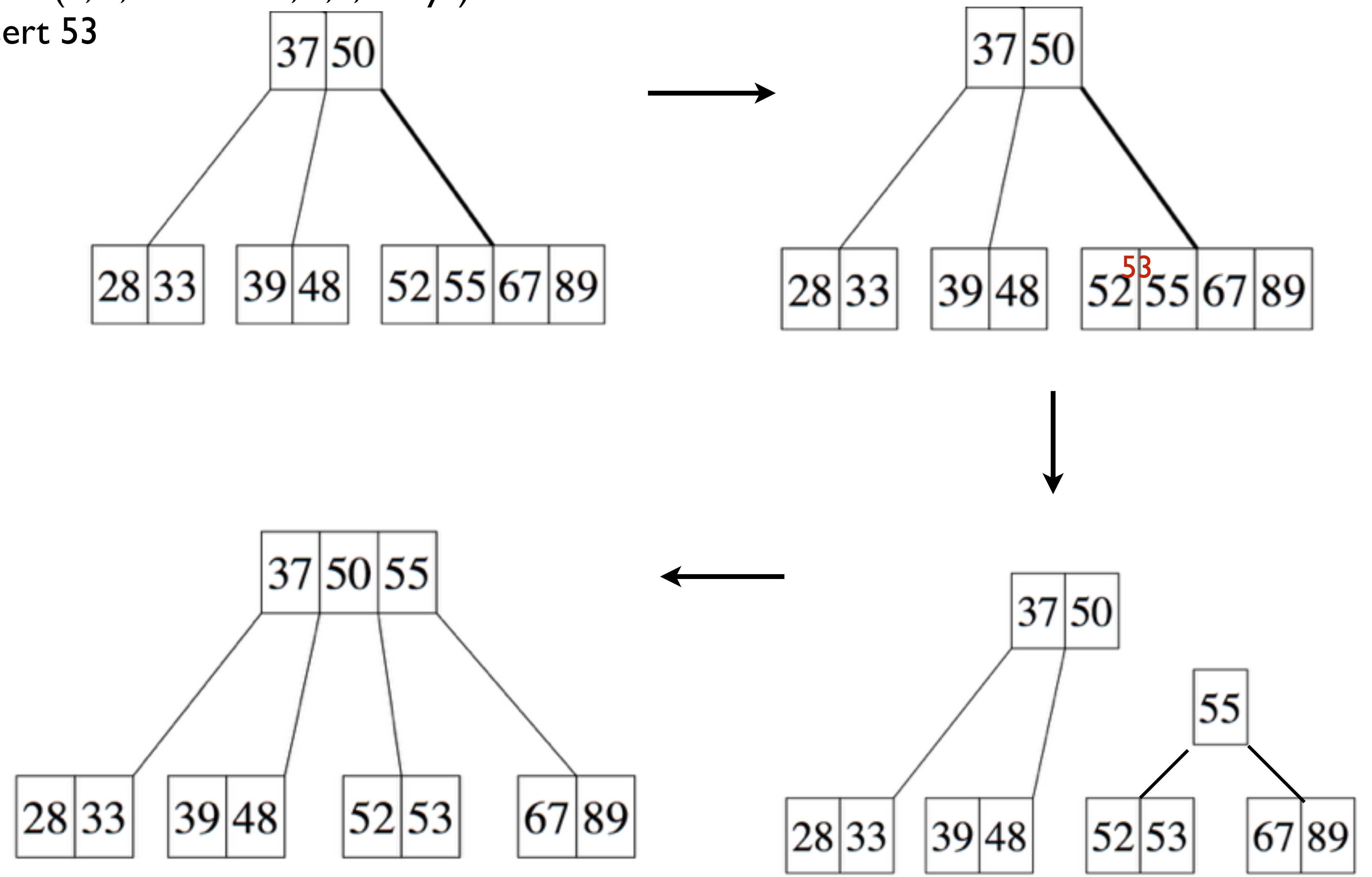
insertion

m= 5 (3, 4, 5 children, 2,3,4 keys)
insert 53



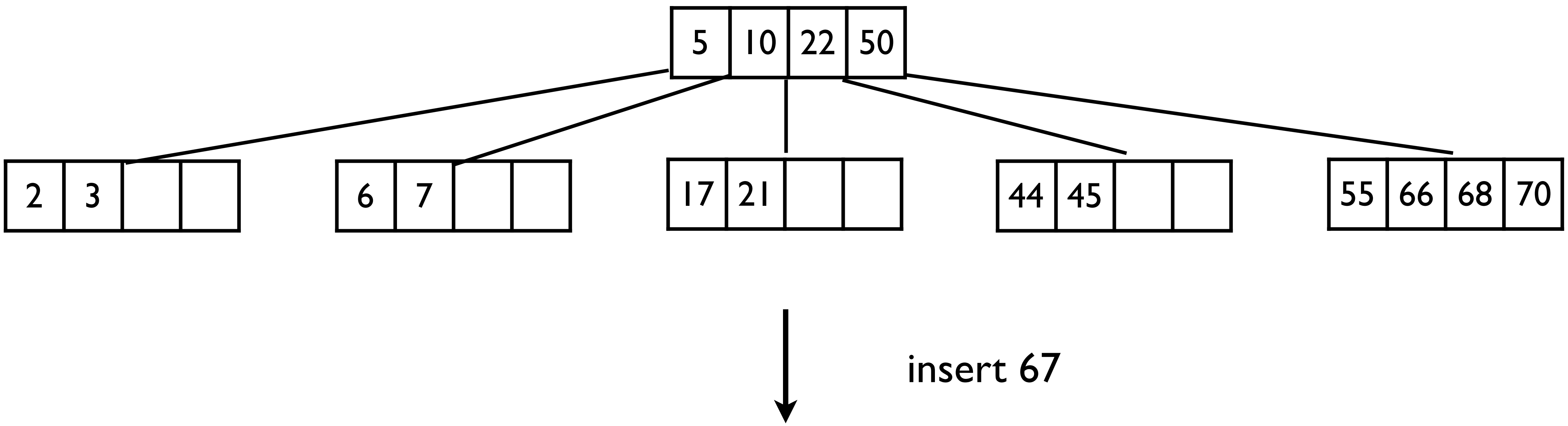
insertion

m= 5 (3, 4, 5 children, 2,3,4 keys)
insert 53



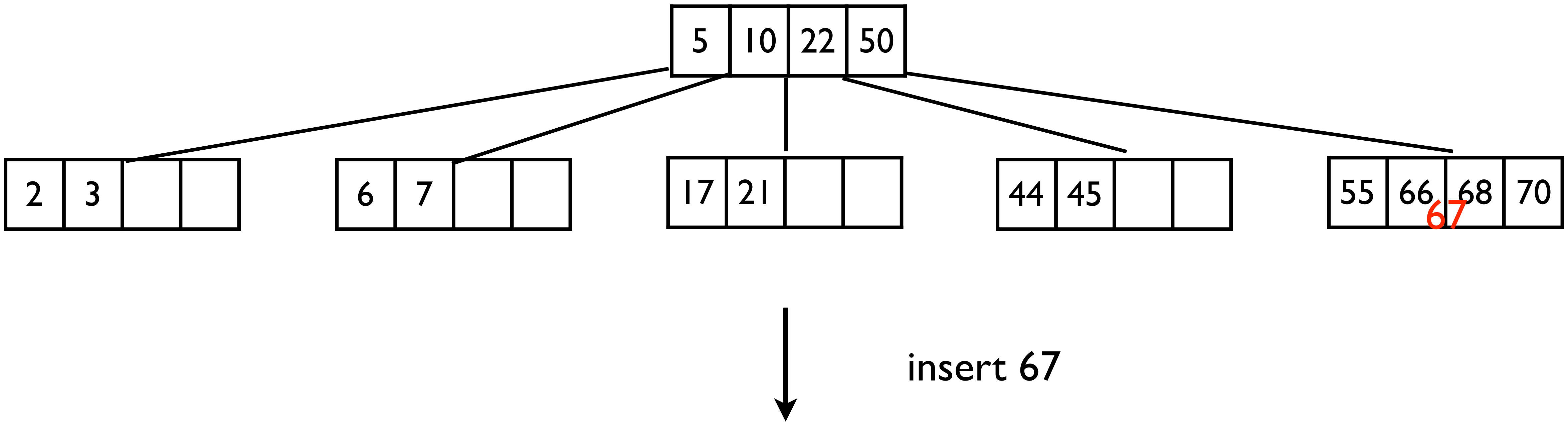
insertion

m= 5 (3, 4, 5 children, 2,3,4 keys)



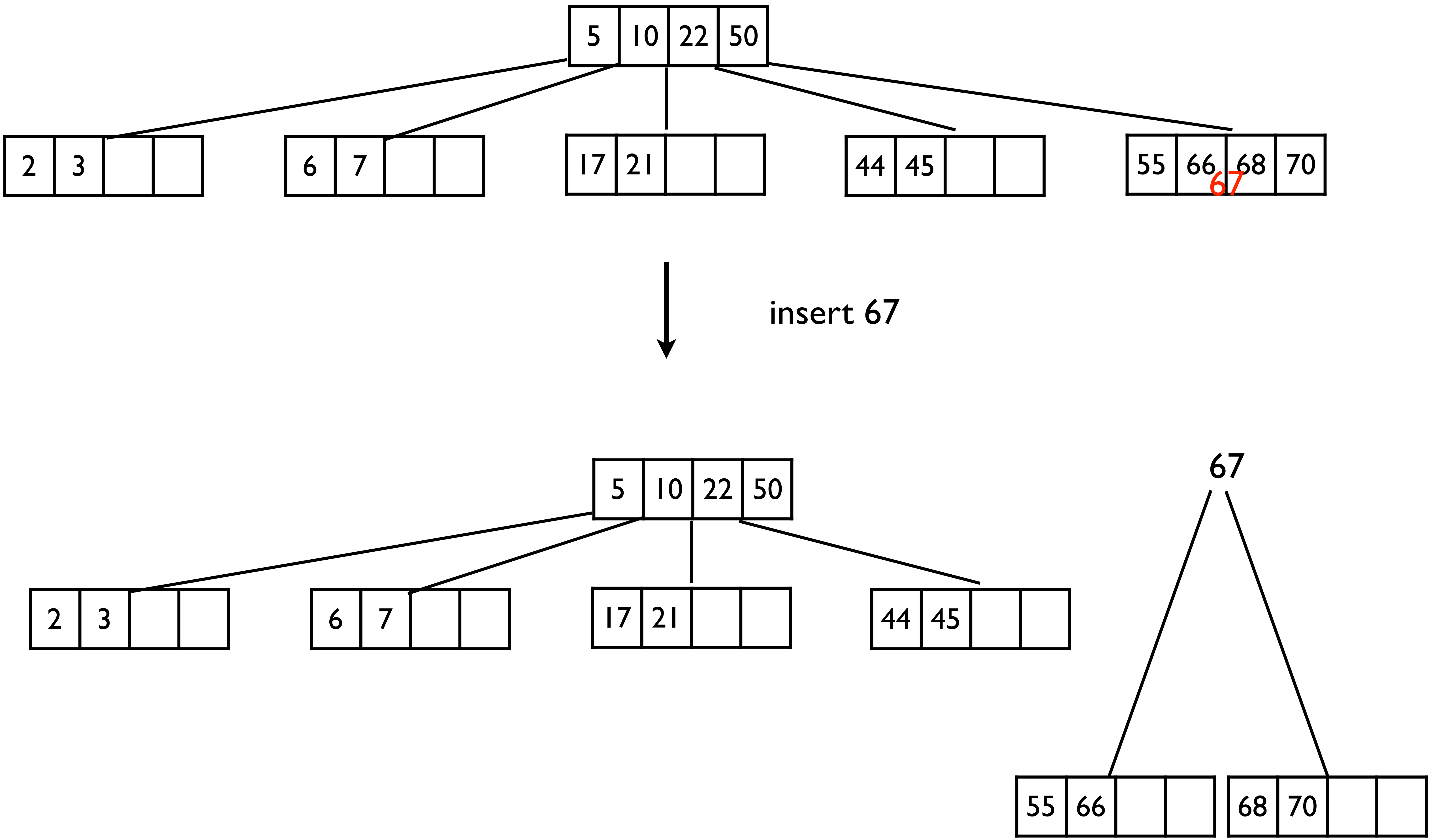
insertion

m= 5 (3, 4, 5 children, 2,3,4 keys)



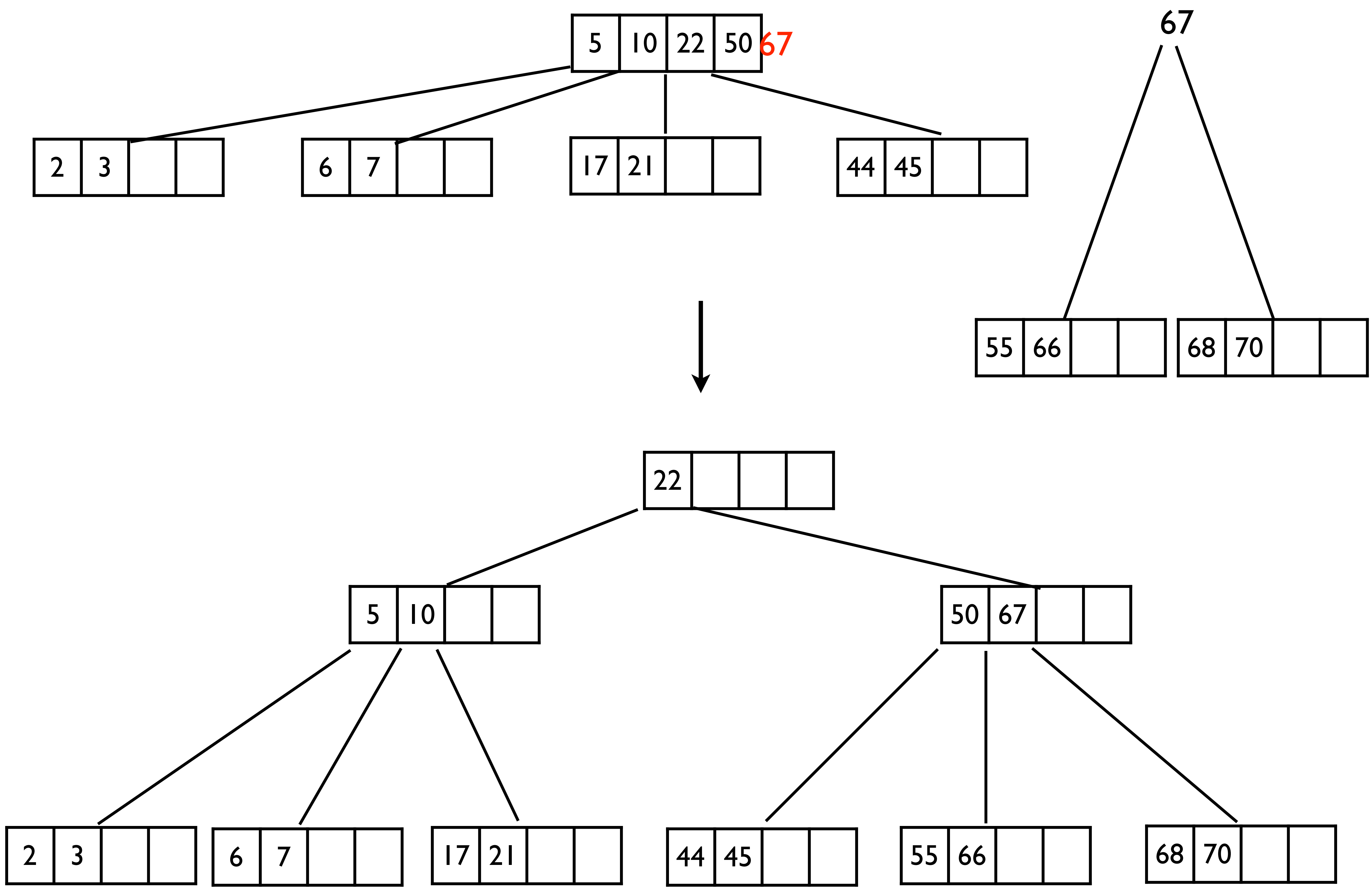
insertion

m= 5 (3, 4, 5 children, 2,3,4 keys)



insertion

m= 5 (3, 4, 5 children, 2,3,4 keys)



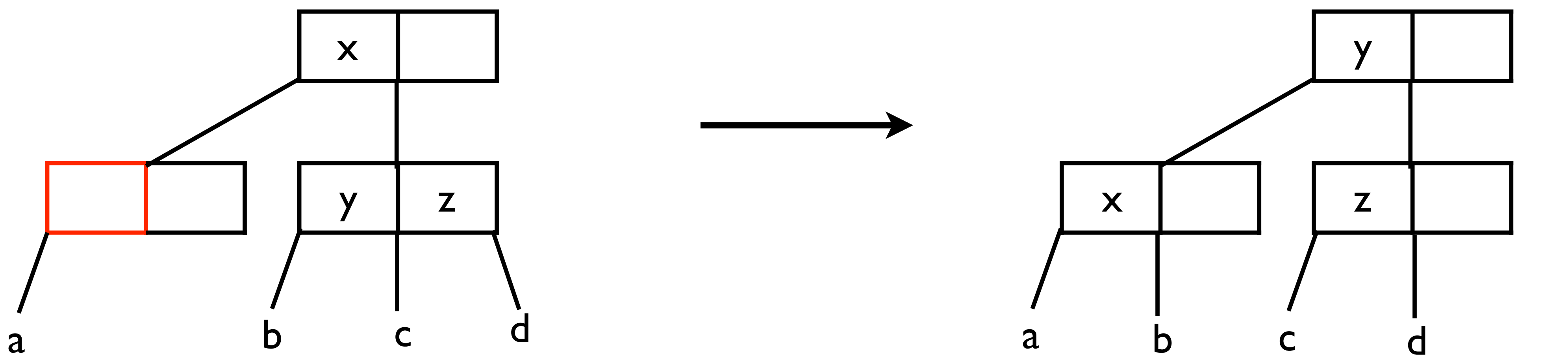
deletion

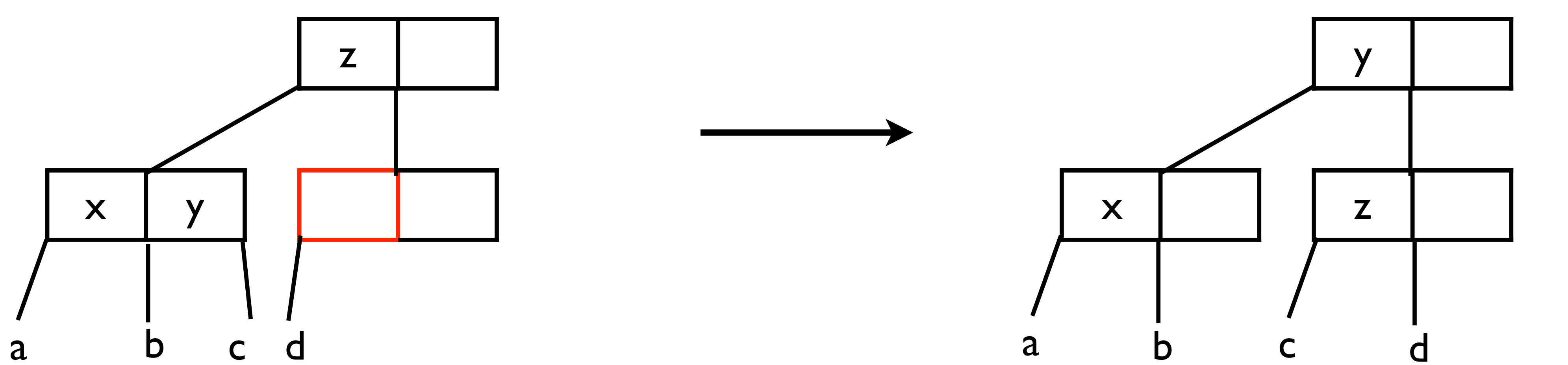
- find a suitable replacement which is the largest key in the left child (or the smallest in the right) and move it to fill the hole
  - key rotation
  - node merging

deletion

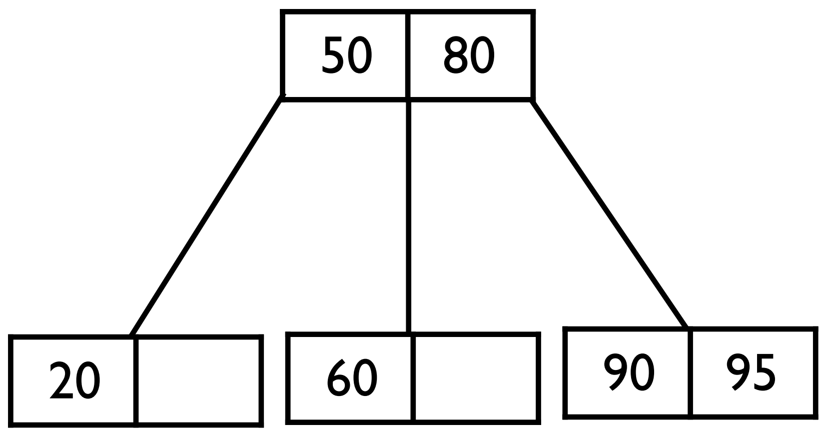
- key rotation: check for siblings for rotation

m= 3 (2,3 children, 1,2 keys)



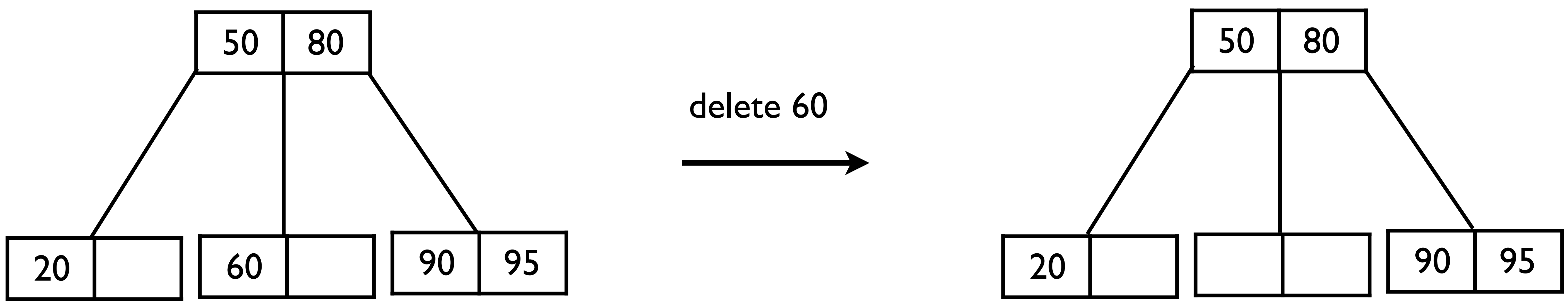


deletion

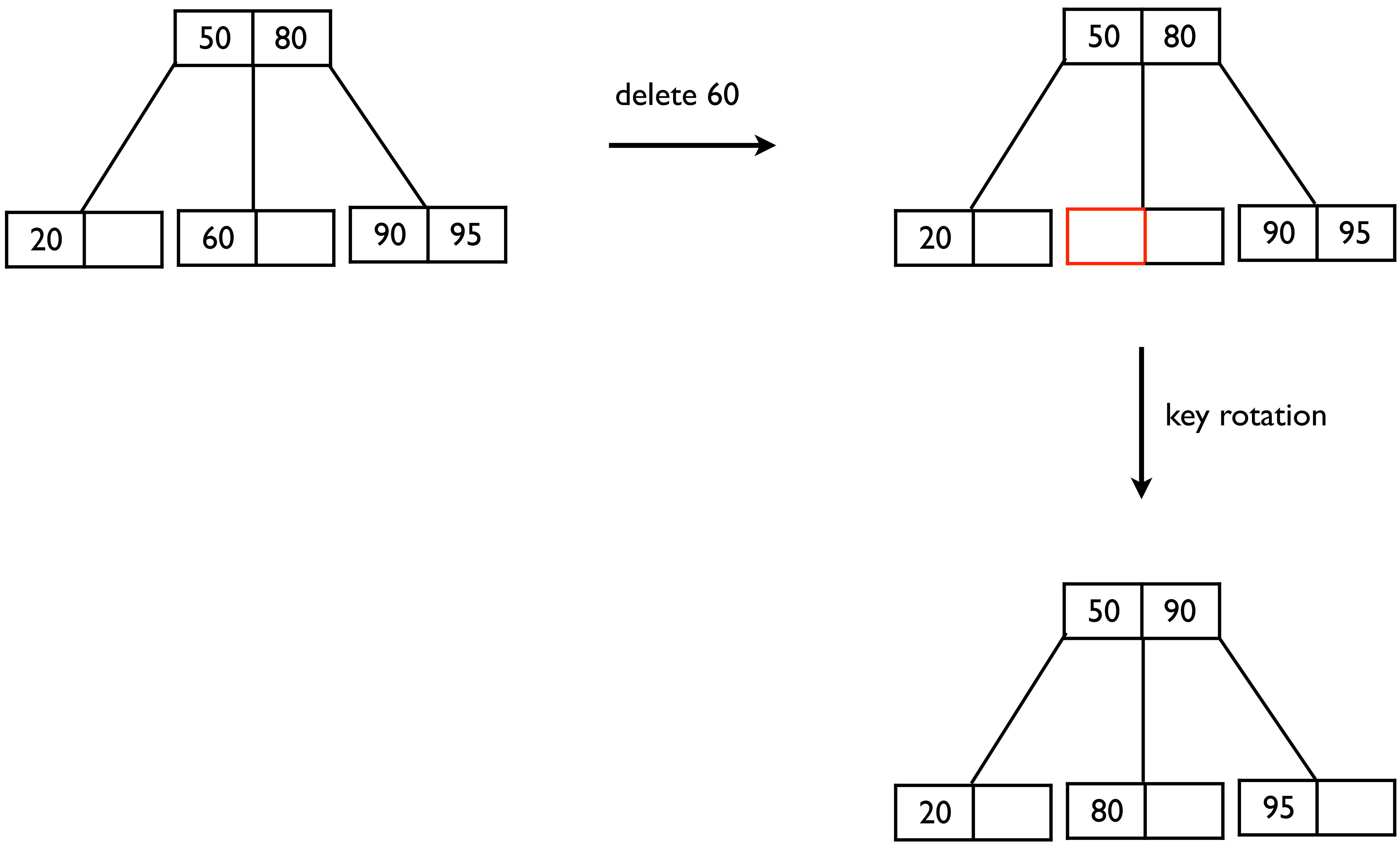


delete 60

deletion



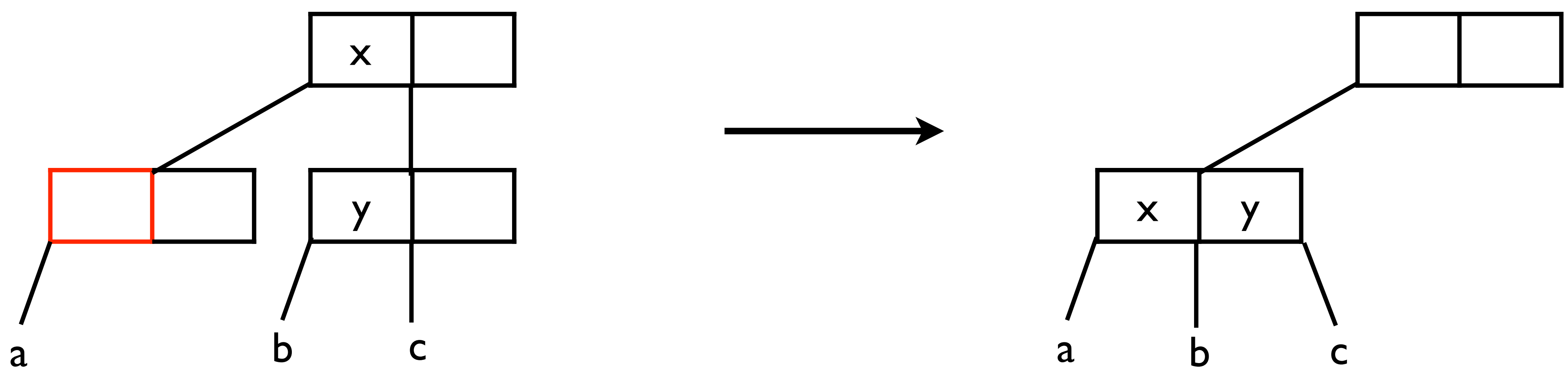
deletion

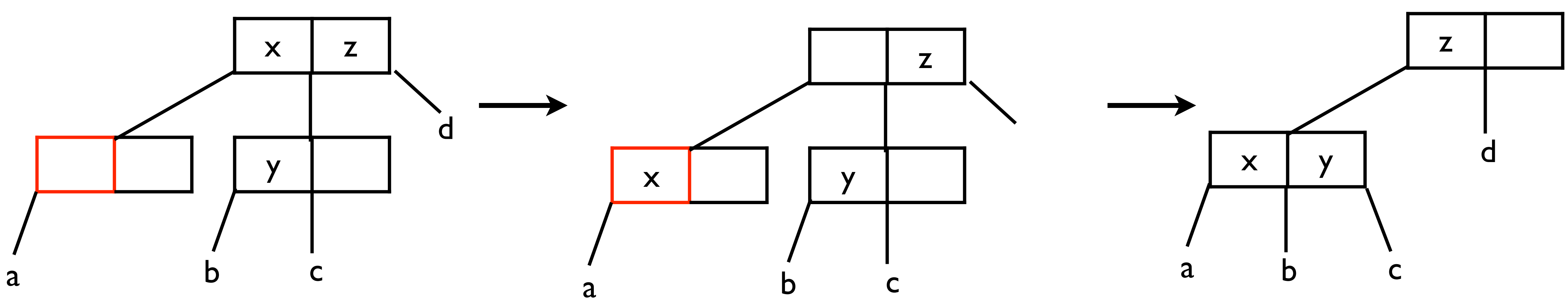


50
80
20
60
90
95
delete 60
50
80
20
90
95
key rotation
50
90
20
80
95

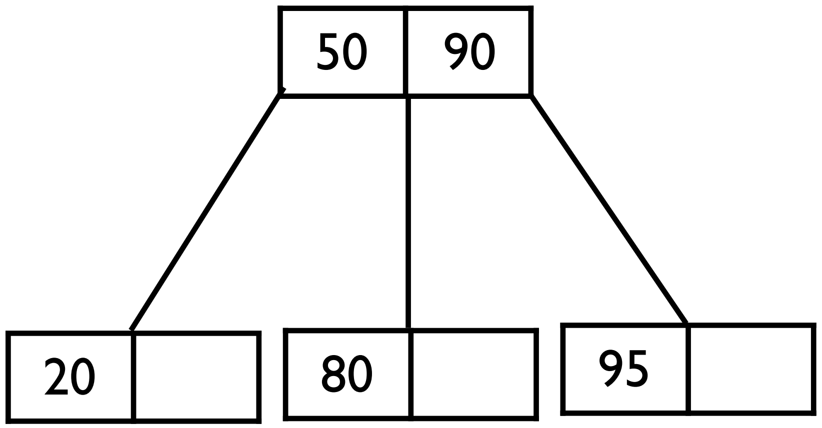
deletion

- 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



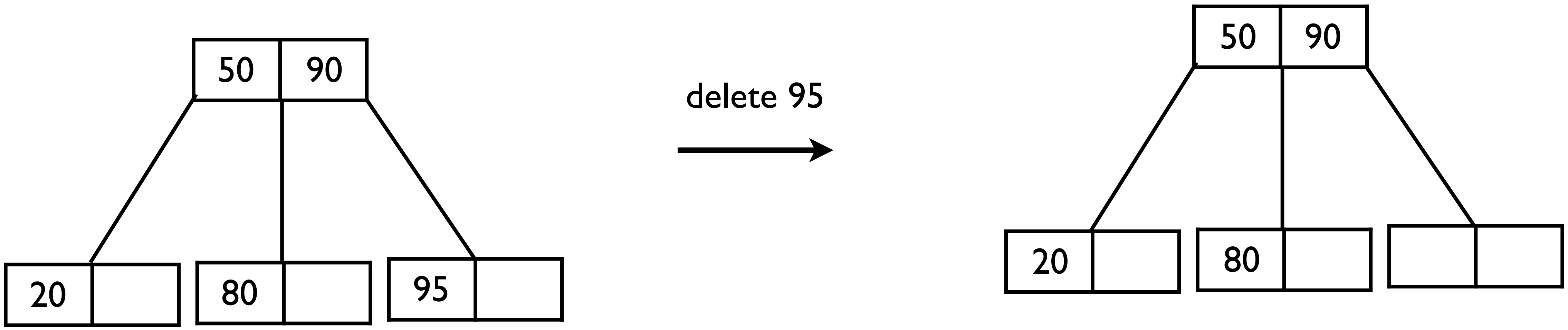


deletion



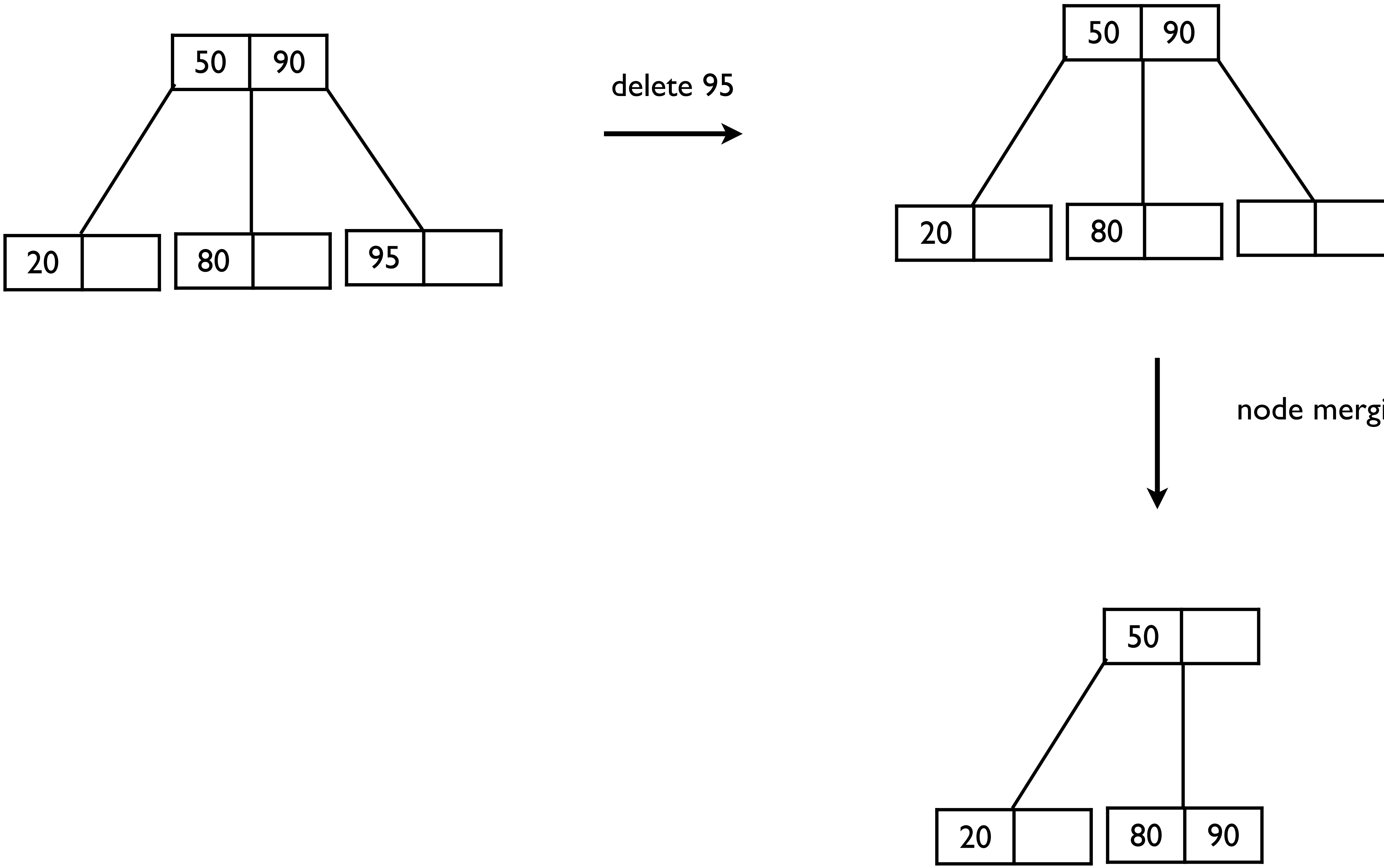
delete 95

deletion



deletion

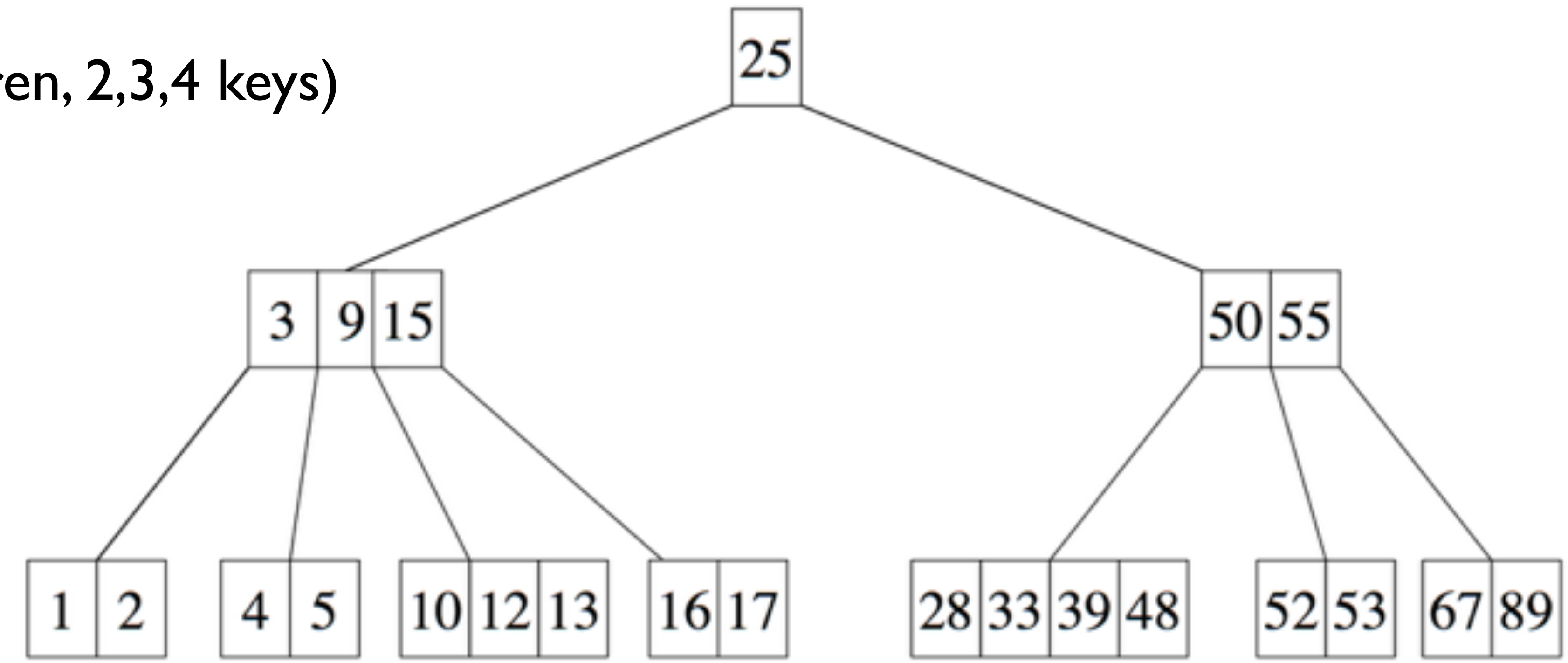
m= 3 (2,3 children, 1,2 keys)



deletion

m= 5 (3, 4, 5 children, 2,3,4 keys)

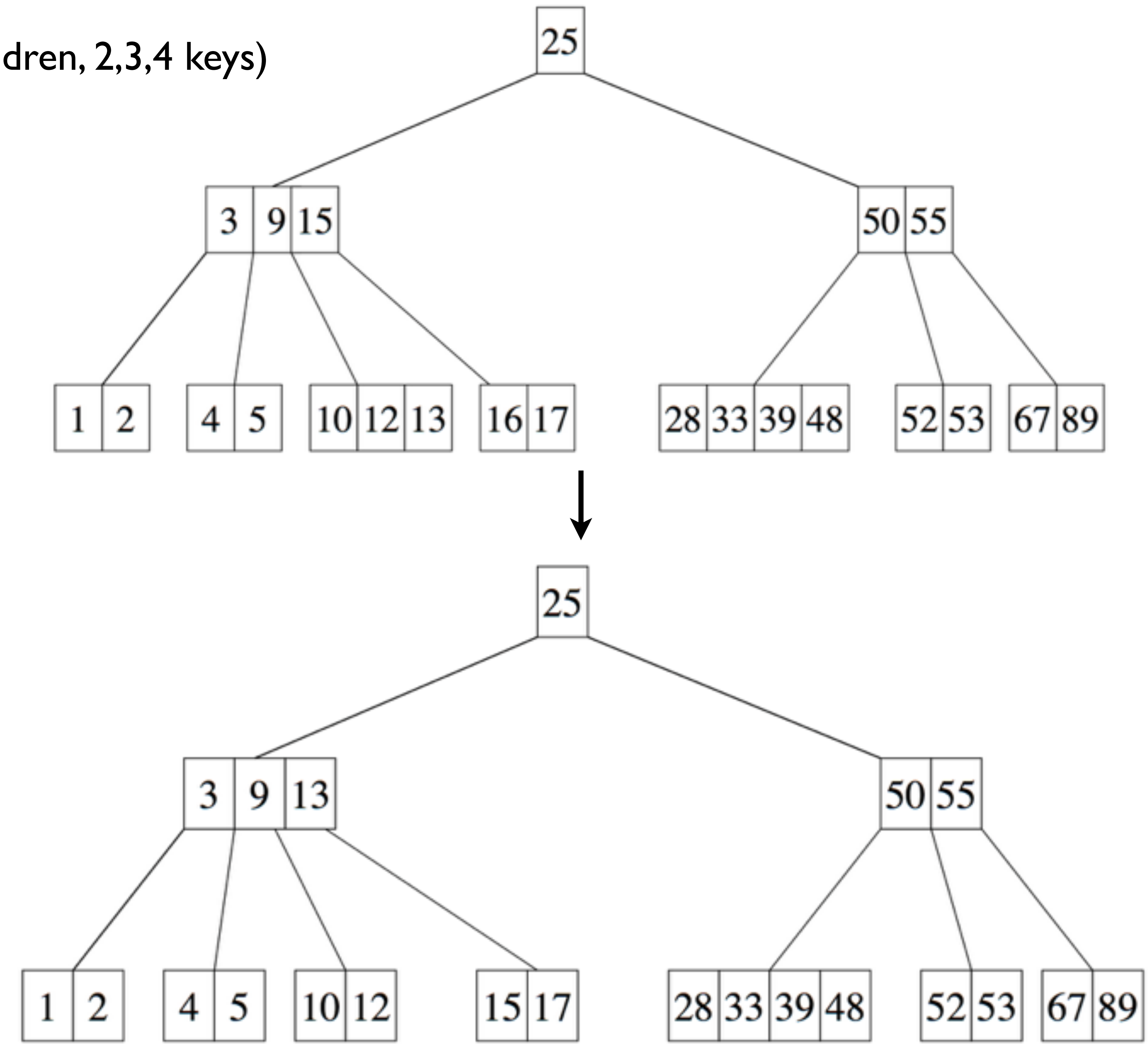
delete 16



deletion

m= 5 (3, 4, 5 children, 2,3,4 keys)

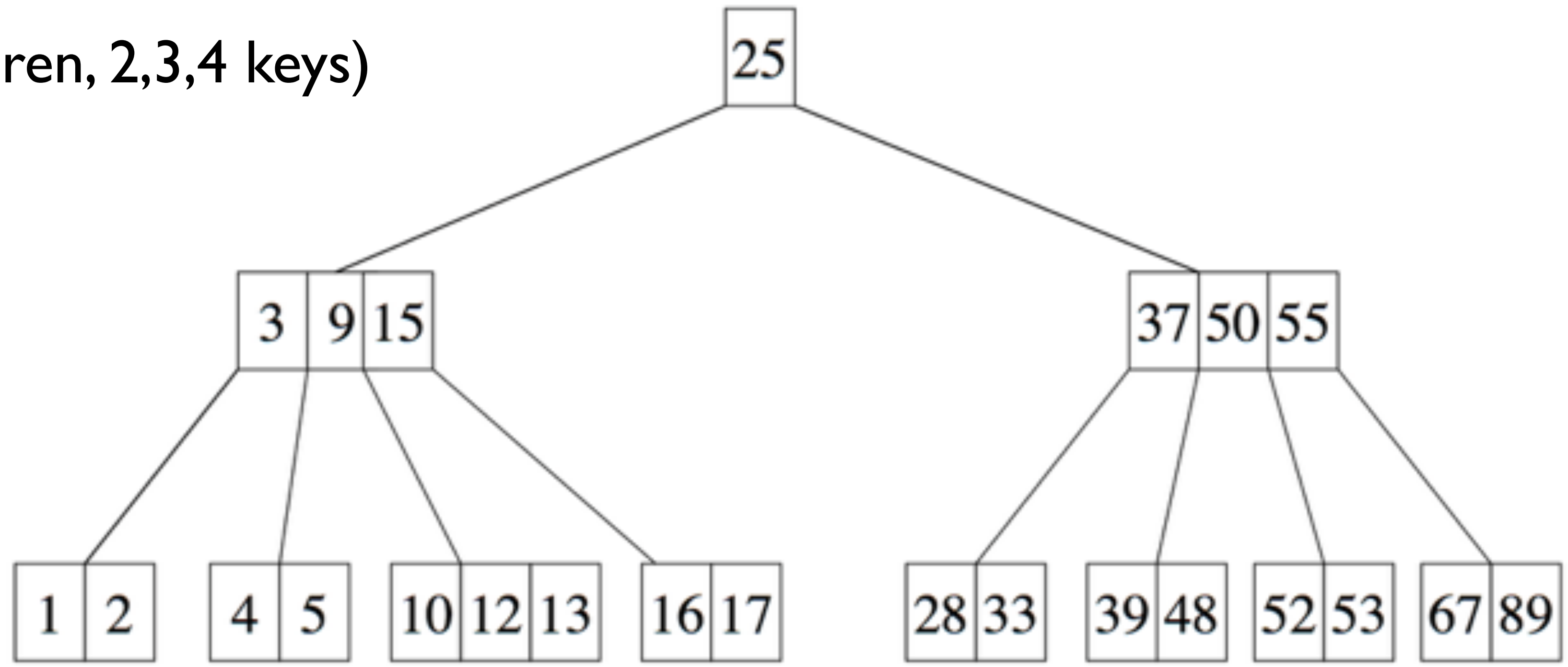
delete 16



deletion

m= 5 (3, 4, 5 children, 2,3,4 keys)

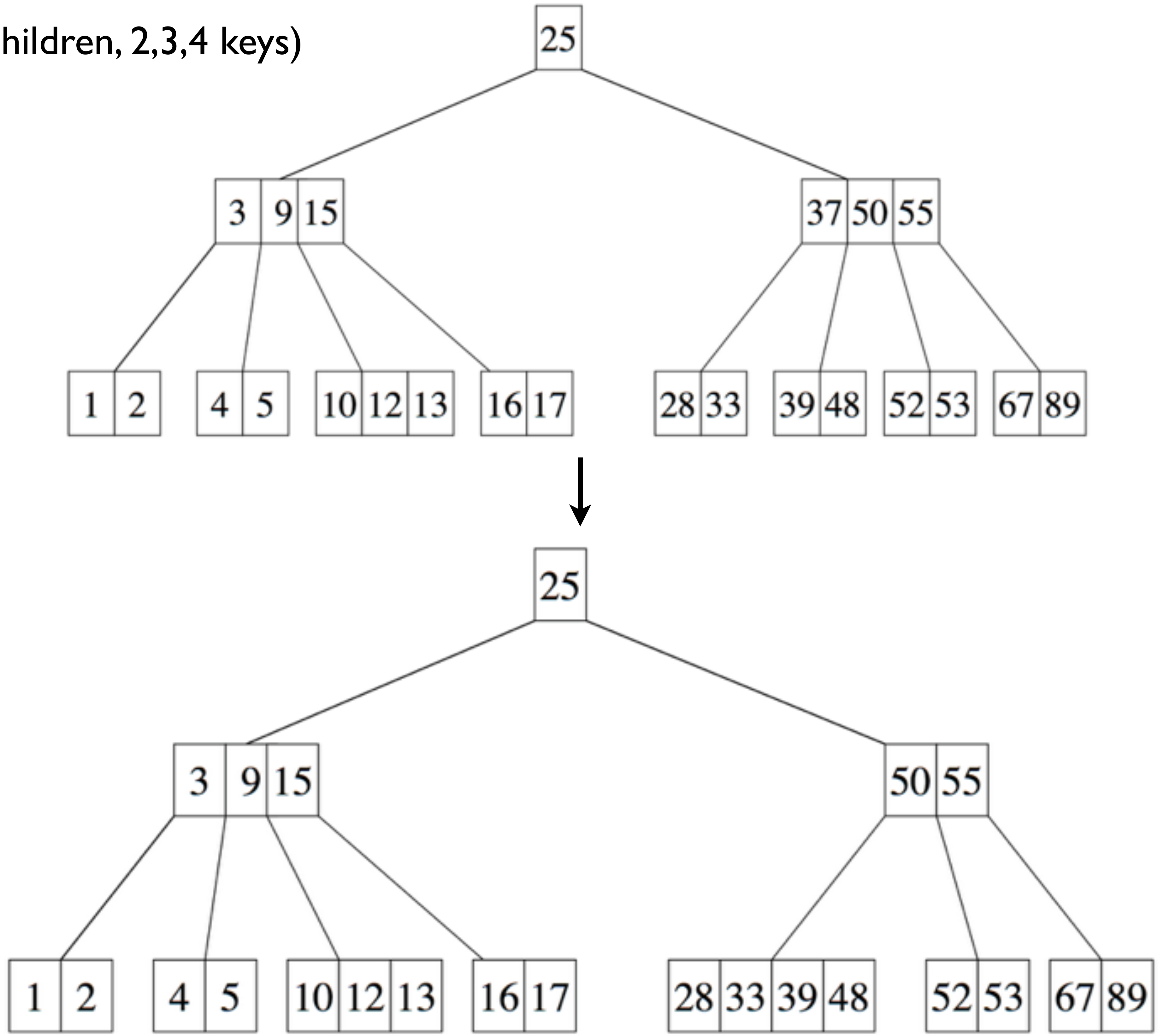
delete 37



deletion

m= 5 (3, 4, 5 children, 2,3,4 keys)

delete 37



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.