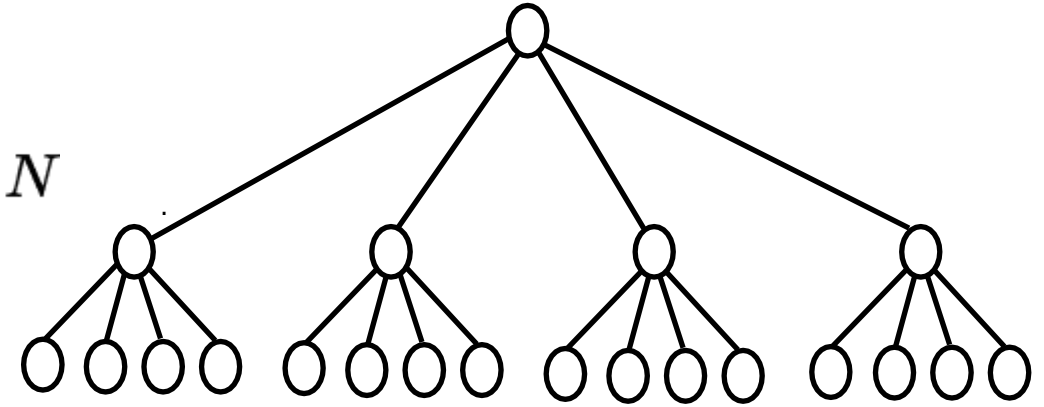
# B Trees

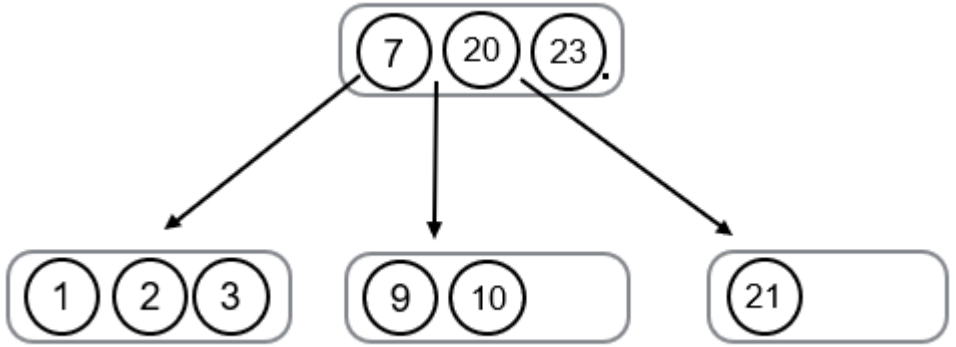
CS223: Data Structures

# M-way tree

Each node can have M children.
Height of a complete M-way tree is $\log_M N$



We can generalize binary search trees to M-way search trees.

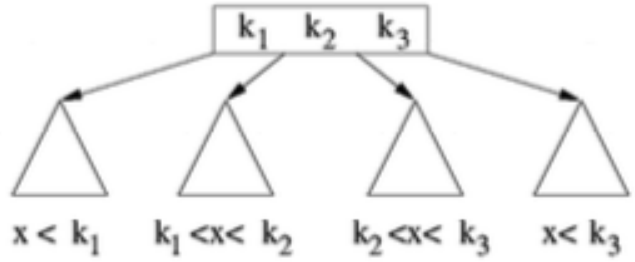


# Definition of a B-tree

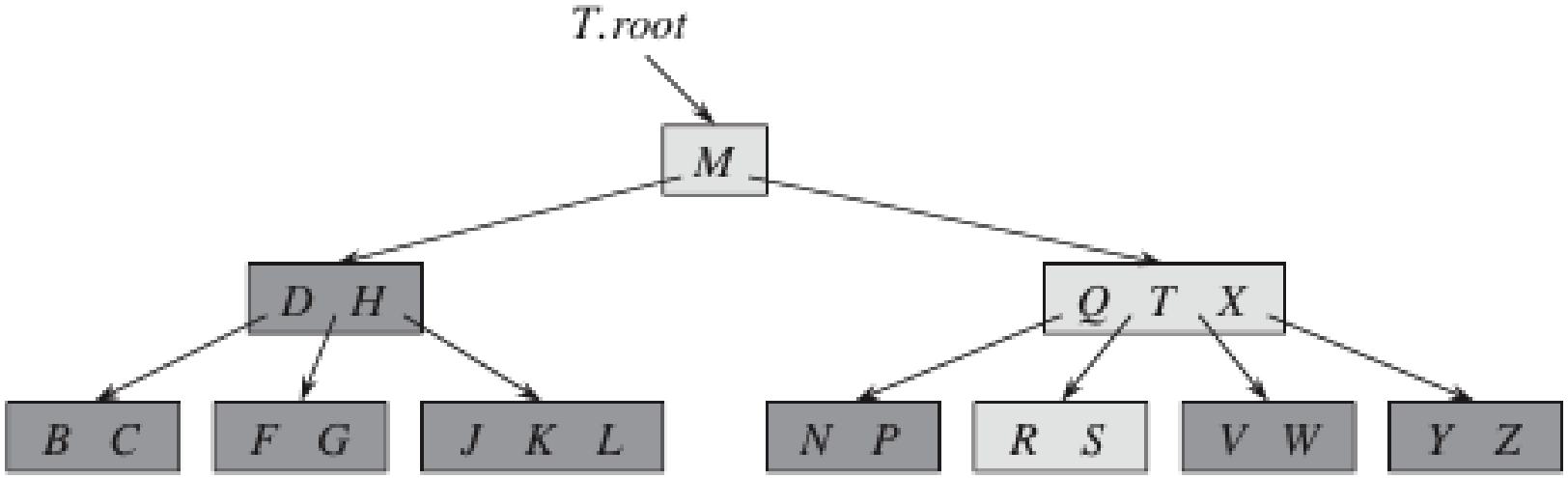
A B-Tree is a generalization of the 2-3 tree to M-way search trees.
A B-tree of order *m* is an *m*-way tree (i.e., a tree where each node may have up to *m* children) in which:

1. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
2. all leaves are on the same level
3. all non-leaf nodes except the root have at least $\lceil m / 2 \rceil$ children
4. the root is either a leaf node, or it has from two to *m* children
5. a leaf node contains no more than *m* – 1 keys
6. The maximum number of items in a B-tree of order *m* and height *h* = $\boldsymbol{m^{h+1} - 1}$
   When *m* = 5 and *h* = 2 this gives $5^3 - 1 = 124$

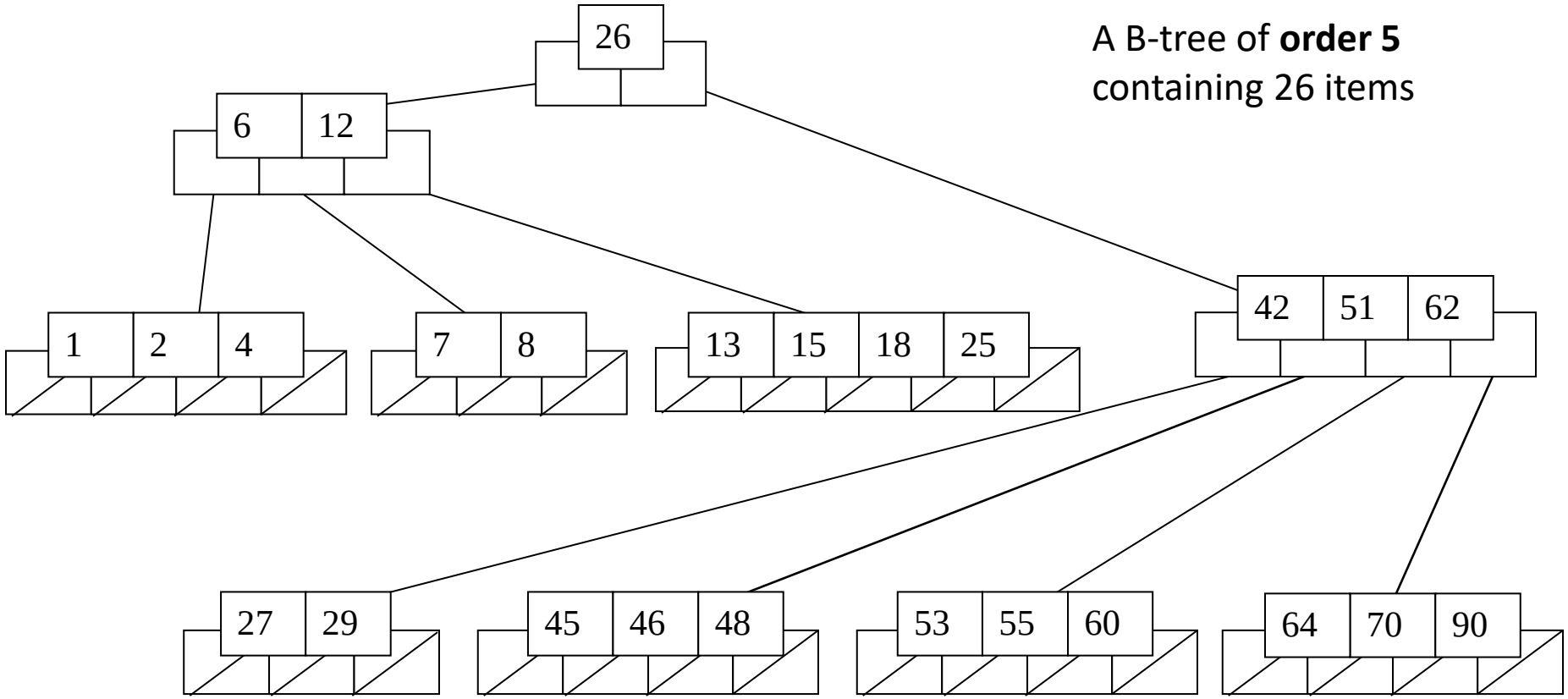
Every internal node (except for the root) has
$\lceil \frac{M}{2} \rceil \leq d \leq M$ children and contains $d - 1$ values.



# B-Trees: an example



# An example B-Tree




A B-tree of **order 5** containing 26 items

*Note that all the leaves are at the same level*

# Inserting into a B-Tree

- Attempt to insert the new key into a leaf
  - If this would result in that leaf becoming too big, split the leaf into two, promoting the middle key to the leaf's parent
  - If this would result in the parent becoming too big, split the parent into two, promoting the middle key
- This strategy might have to be repeated all the way to the top
- If necessary, the root is split in two and the middle key is promoted to a new root, making the tree one level higher

# B-Tree: Insert *X*

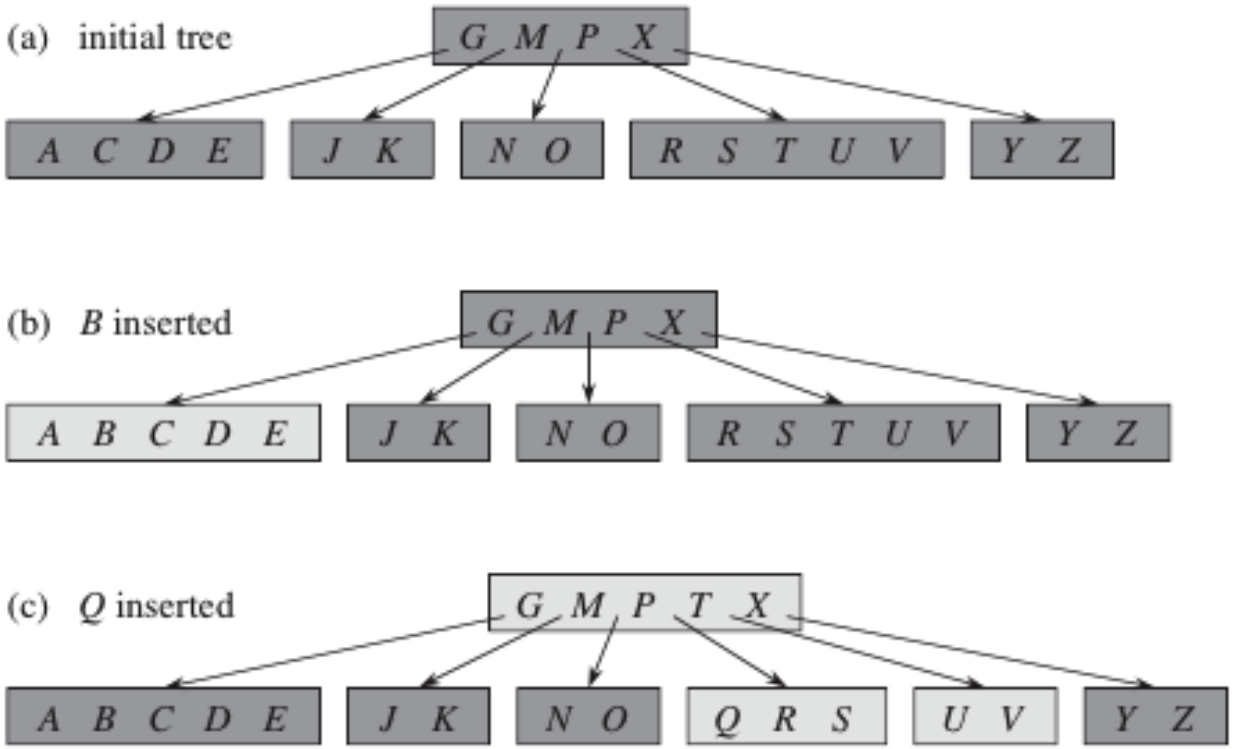
1. As in *M-way* tree find the leaf node to which *X* should be added
2. Add *X* to this node in the appropriate place among the values already there
3. Number of values in the node after adding the key:
   - Fewer than *2t-1:* done
   - Equal to *2t: overflowed*
4. Fix overflowed node

# Fix an Overflowed

1. Split the node into three parts, *M=2t*:
   - Left: the first *t* values, become a left child node
   - Middle: the middle value at position *t*, goes up to parent
   - Right: the last *t-1* values, become a right child node
2. Continue with the parent:
   - Until no overflow occurs in the parent
   - If the root overflows, split it too, and create a new root node

# B-Tree-Insert: Illustration

M = 6



# Constructing a B-tree

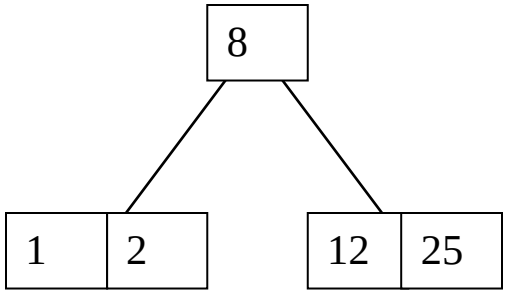
Suppose we start with an empty B-tree and keys arrive in the following order: 1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45
We want to construct a B-tree of order **5**.

The first four items go into the root:

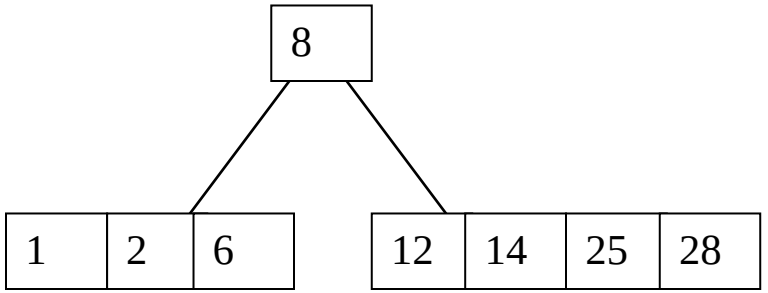
| 1 | 2 | 8 | 12 |
|---|---|---|---|

To put the fifth item in the root would violate condition 5
Therefore, when 25 arrives, pick the middle key to make a new root

# Constructing a B-tree (contd.)

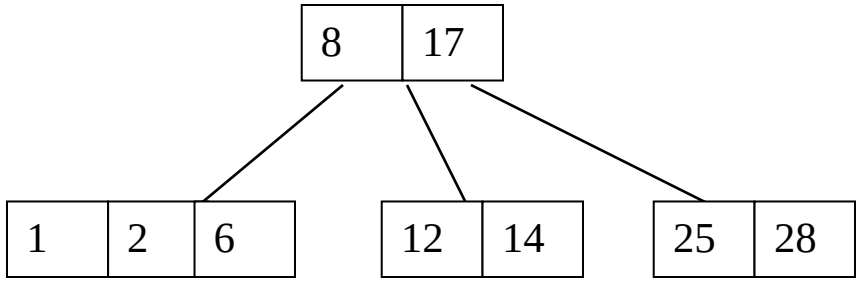



6, 14, 28 get added to the leaf nodes:

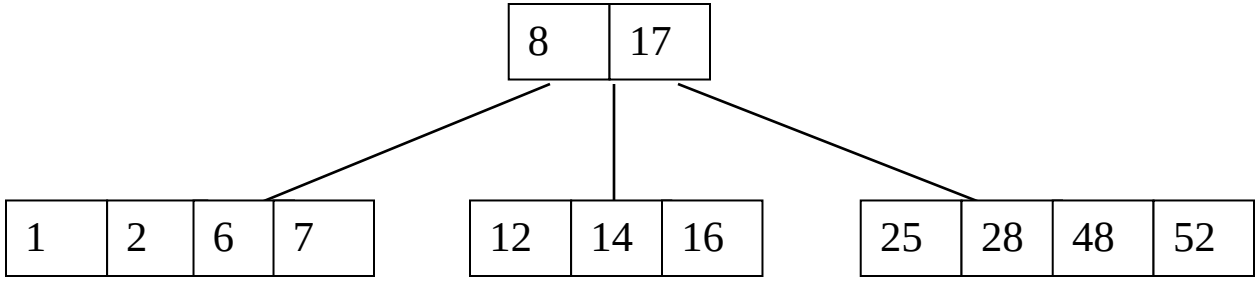


# Constructing a B-tree (contd.)

Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf

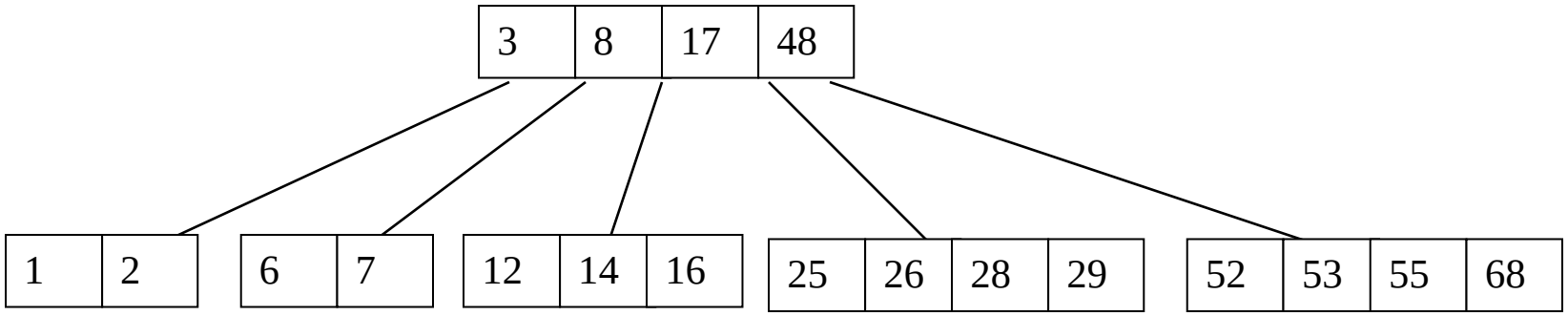



7, 52, 16, 48 get added to the leaf nodes

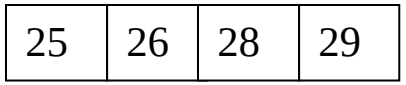


# Constructing a B-tree (contd.)

Adding 68 causes us to split the right most leaf, promoting 48 to the root, and adding 3 causes us to split the left most leaf, promoting 3 to the root; 26, 29, 53, 55 then go into the leaves

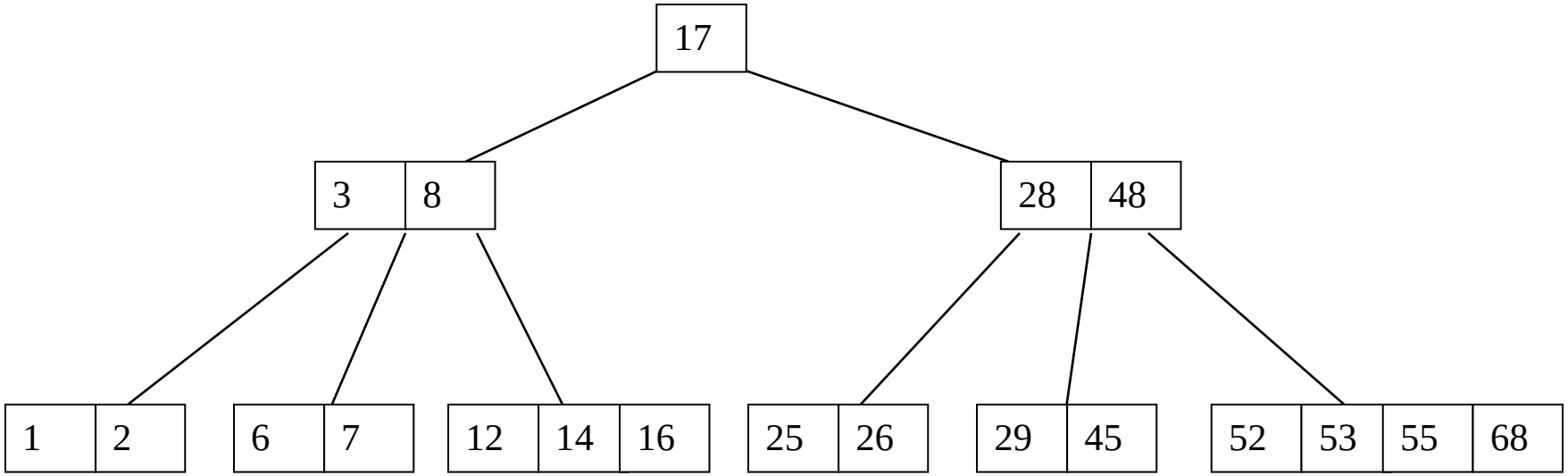



Adding 45 causes a split of



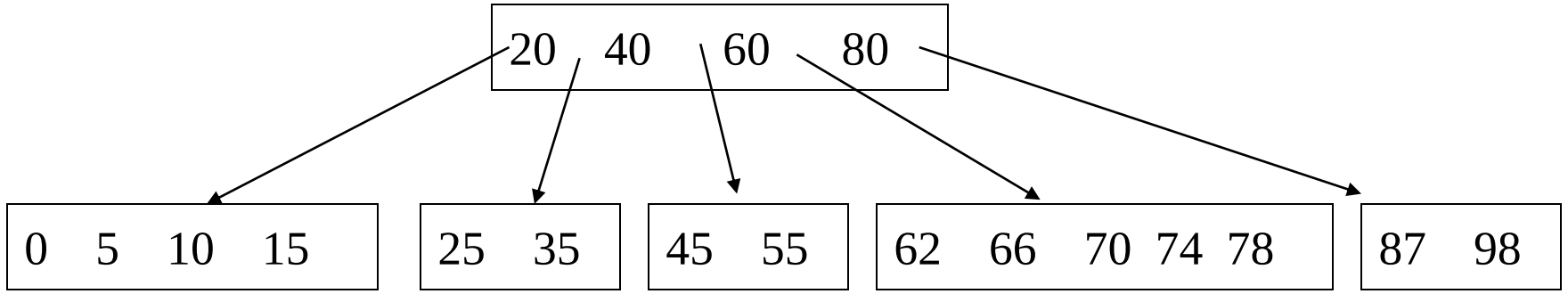

and promoting 28 to the root then causes the root to split

# Constructing a B-tree (contd.)

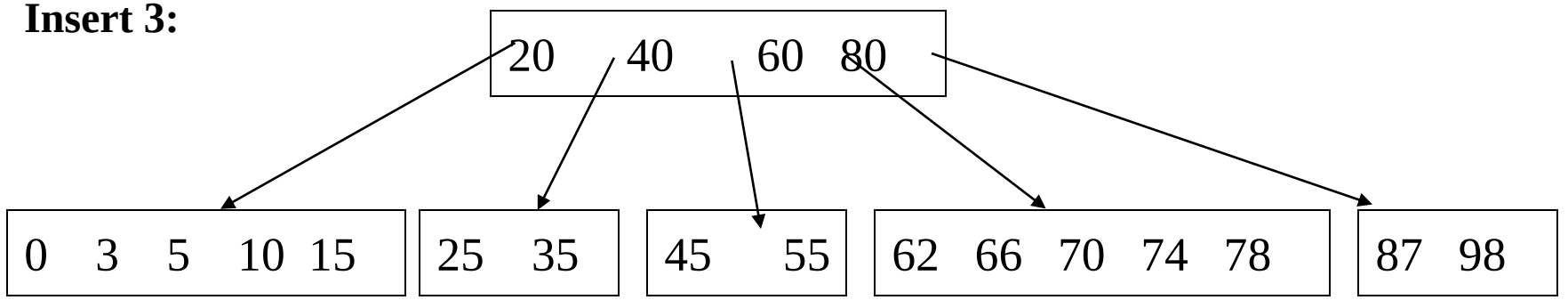


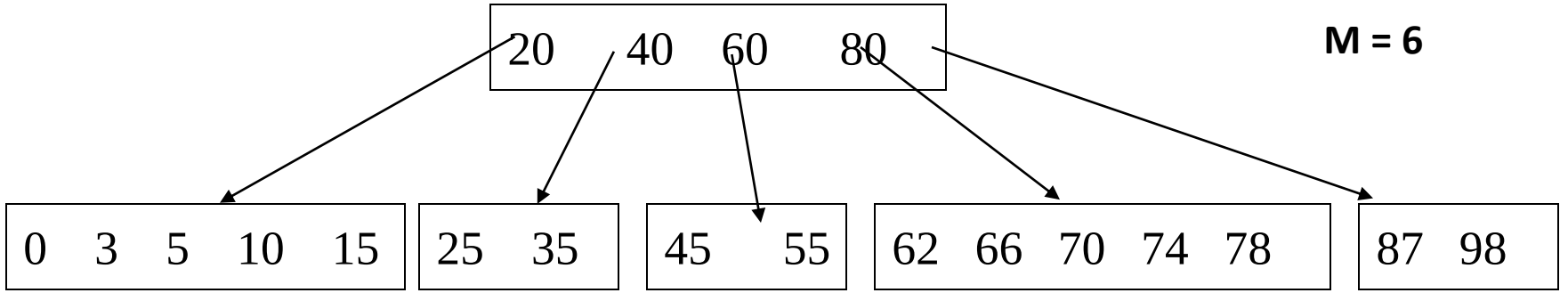
# Insert example

**M = 6**



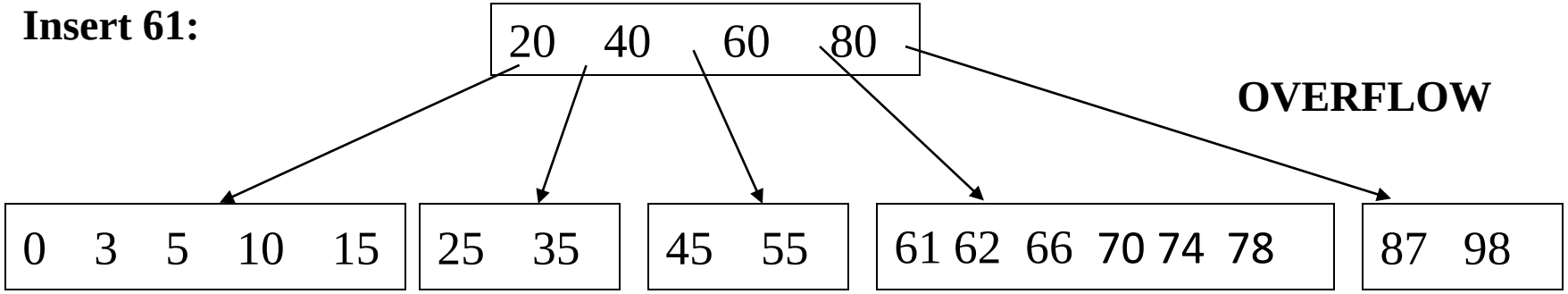

**Insert 3:**



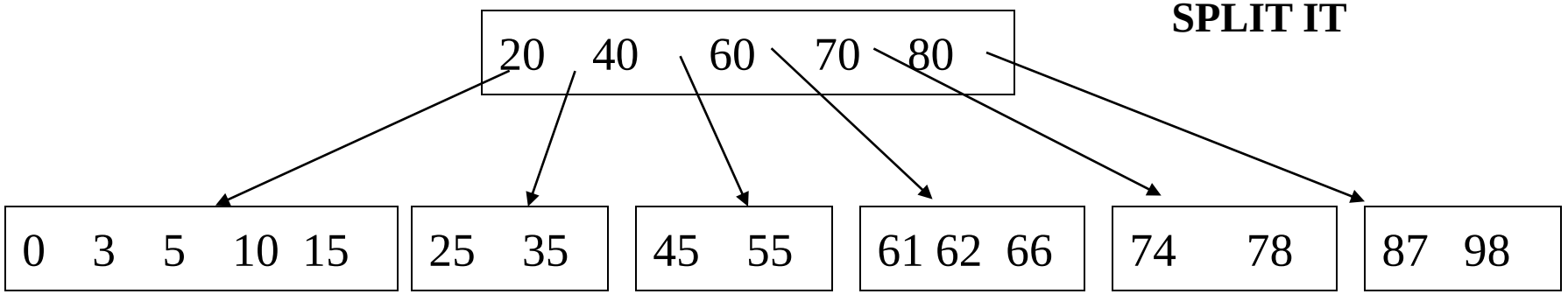



**M = 6**

**Insert 61:**



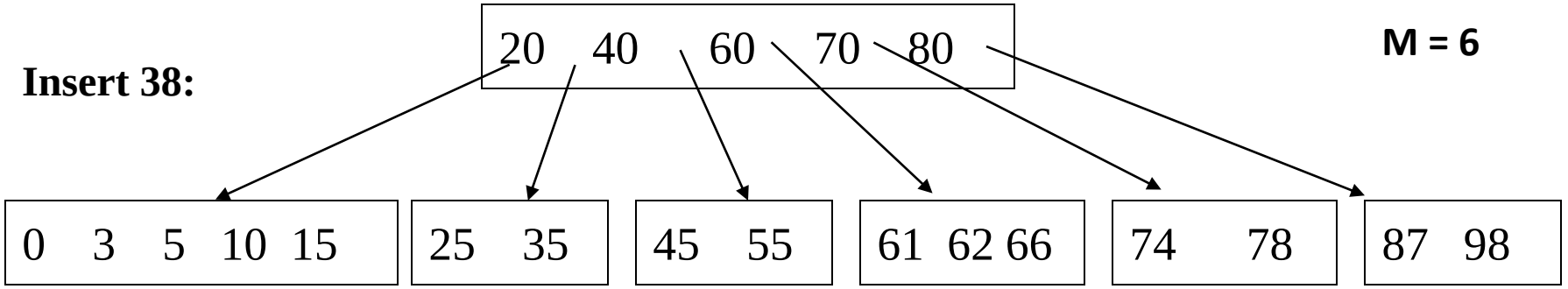

**OVERFLOW**

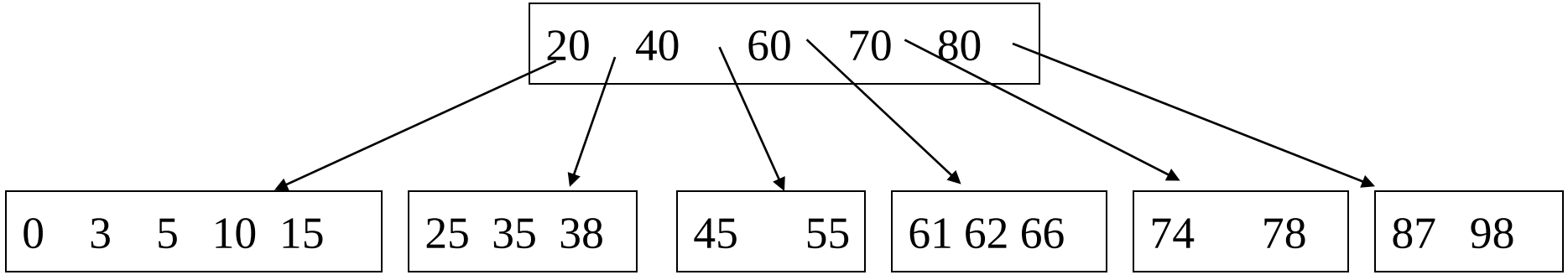



**SPLIT IT**

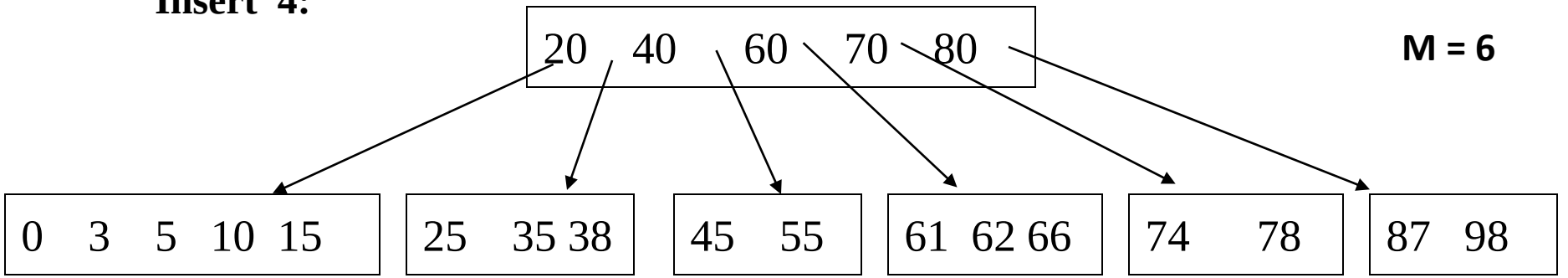
**Insert 38:**

M = 6


20 40 60 70 80
0 3 5 10 15
25 35
45 55
61 62 66
74 78
87 98

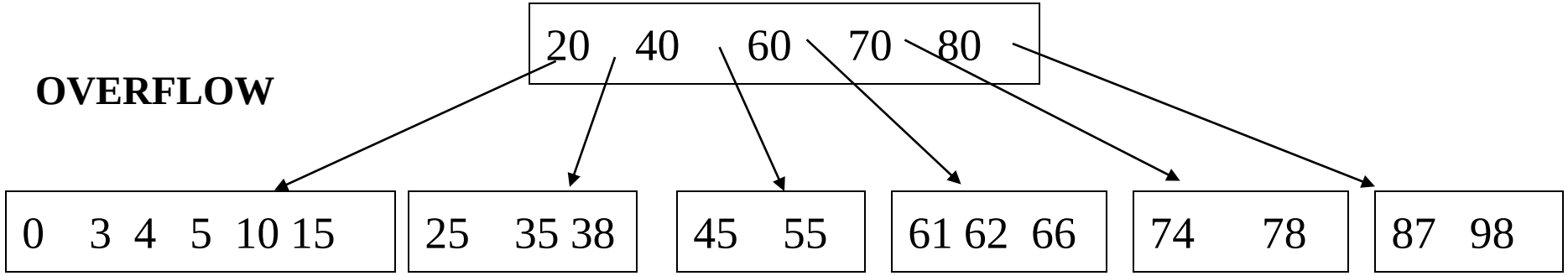

20 40 60 70 80
0 3 5 10 15
25 35 38
45 55
61 62 66
74 78
87 98

**Insert 4:**




**M = 6**

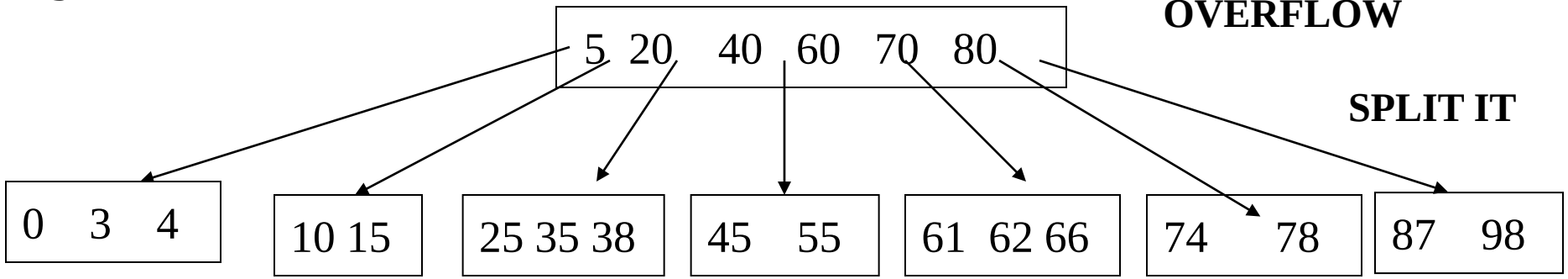
**OVERFLOW**

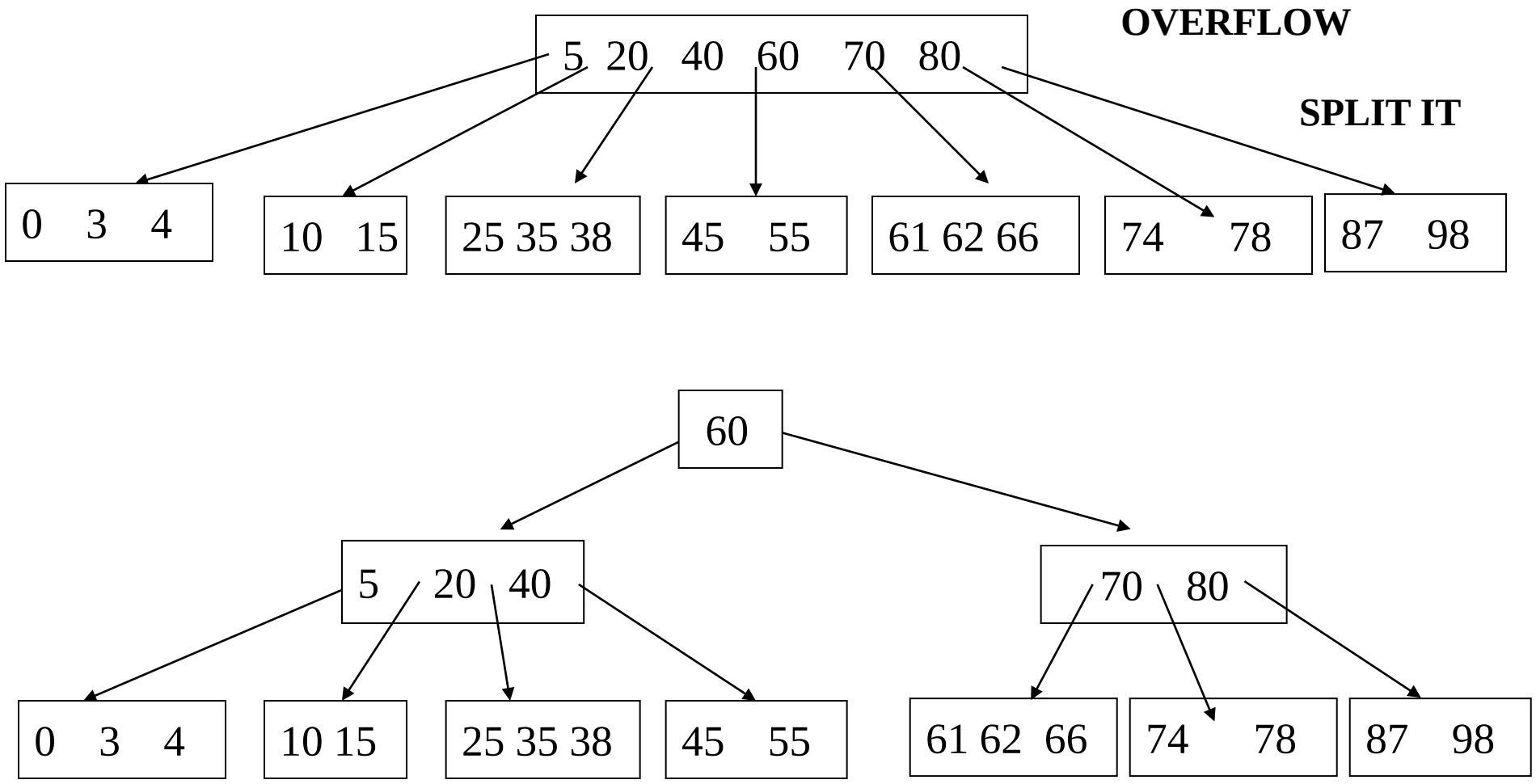



**SPLIT IT**

**OVERFLOW**

**SPLIT IT**




OVERFLOW
5 20 40 60 70 80
SPLIT IT
0 3 4
10 15
25 35 38
45 55
61 62 66
74 78
87 98
60
5 20 40
70 80
0 3 4
10 15
25 35 38
45 55
61 62 66
74 78
87 98

# Removal from a B-tree

During insertion, the key always goes *into* a *leaf*.
For deletion we wish to remove *from* a leaf. There are three possible ways we can do this:

1. If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted.
2. If the key is *not* in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf
   - in this case we can delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.

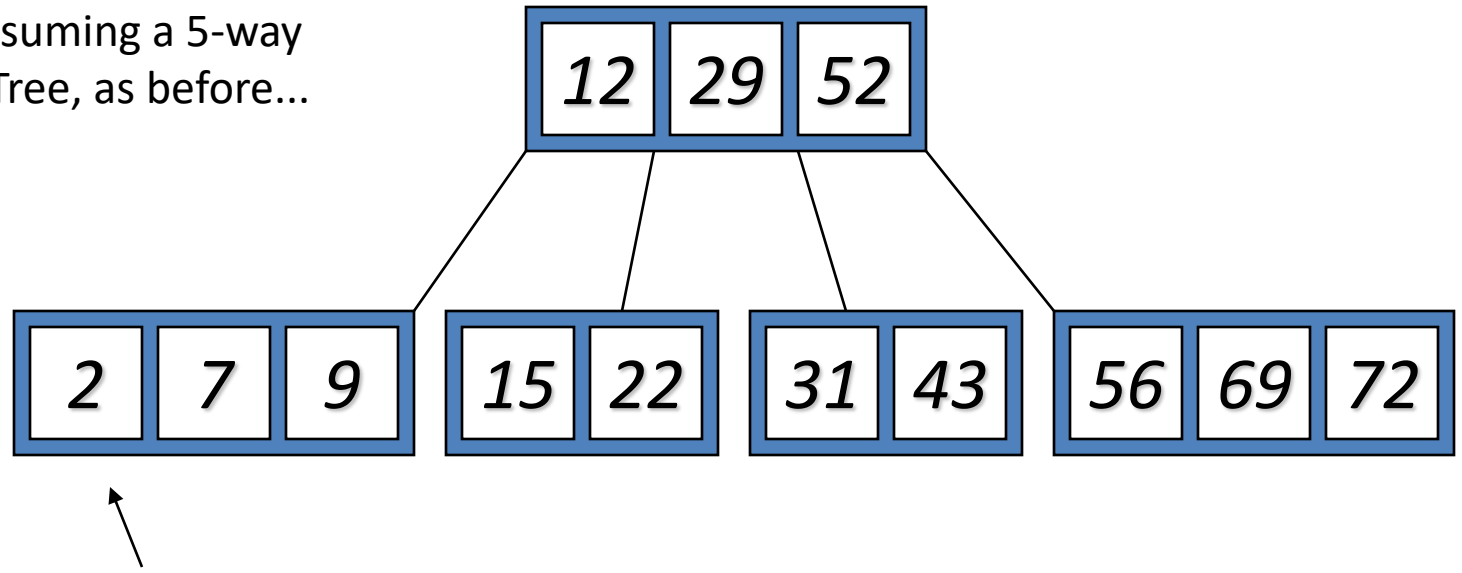
# Removal from a B-tree (2)

If (1) or (2) lead to a leaf node containing less than the minimum number of keys then we have to look at the siblings immediately adjacent to the leaf in question:

- if one of them has more than the min. number of keys then we can promote one of its keys to the parent and take the parent key into our lacking leaf
- if neither of them has more than the min. number of keys then the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key) and the new leaf will have the correct number of keys; if this step leave the parent with too few keys then we repeat the process up to the root itself, if required

# Type #1: Simple leaf deletion

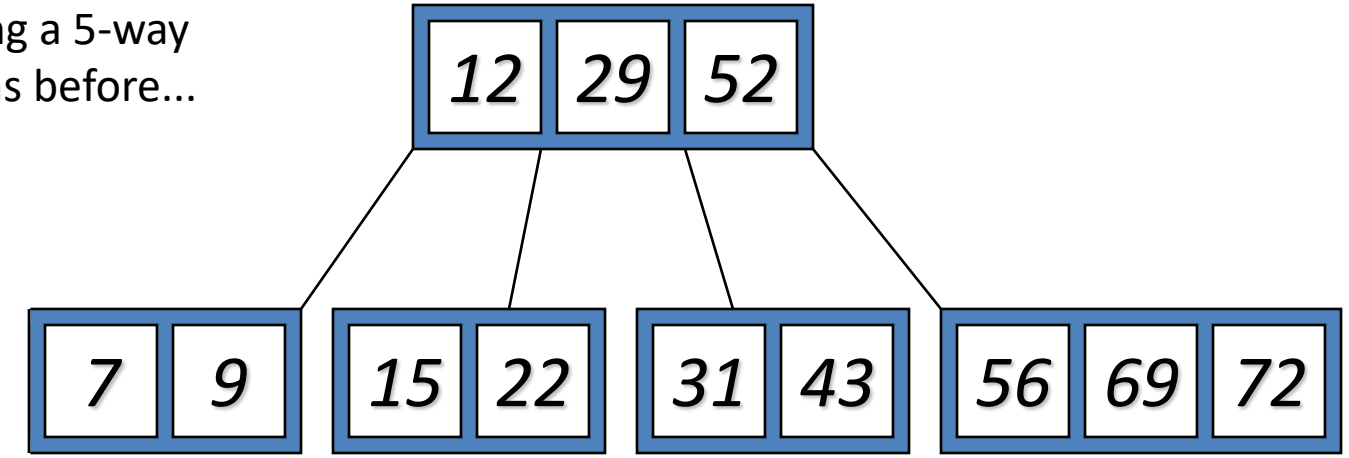
Assuming a 5-way B-Tree, as before...




Delete 2: Since there are enough keys in the node, just delete it

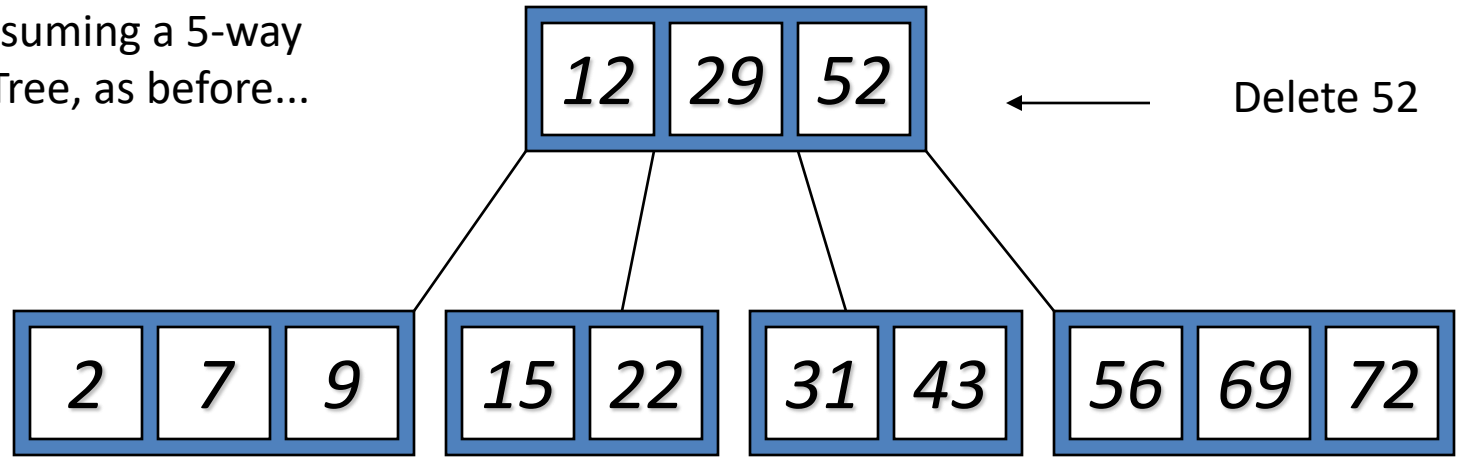
# Type #1: Simple leaf deletion

Assuming a 5-way B-Tree, as before...



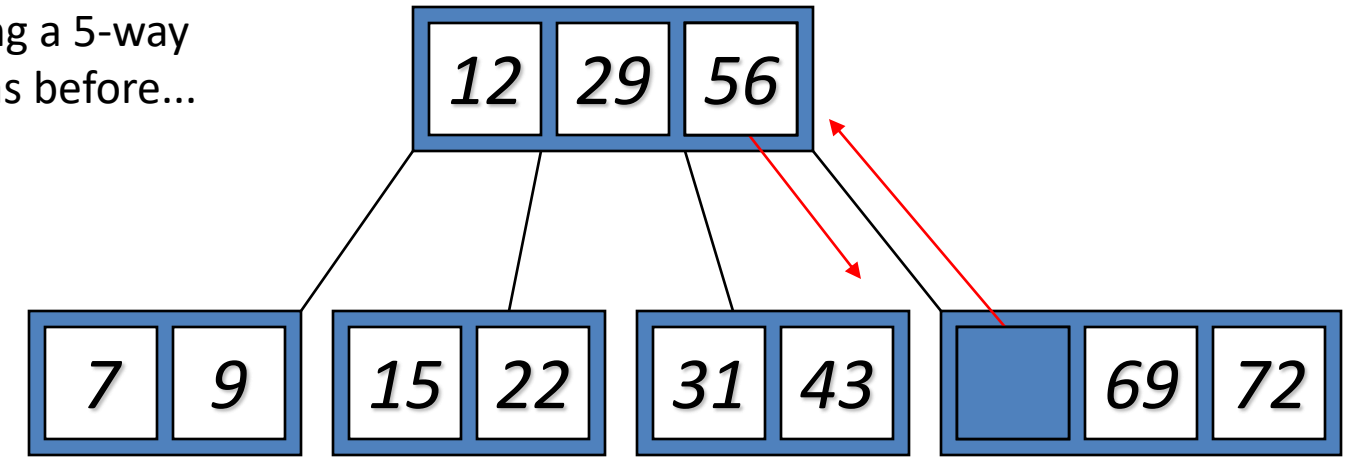
# Type #1: Simple leaf deletion

Assuming a 5-way
B-Tree, as before...



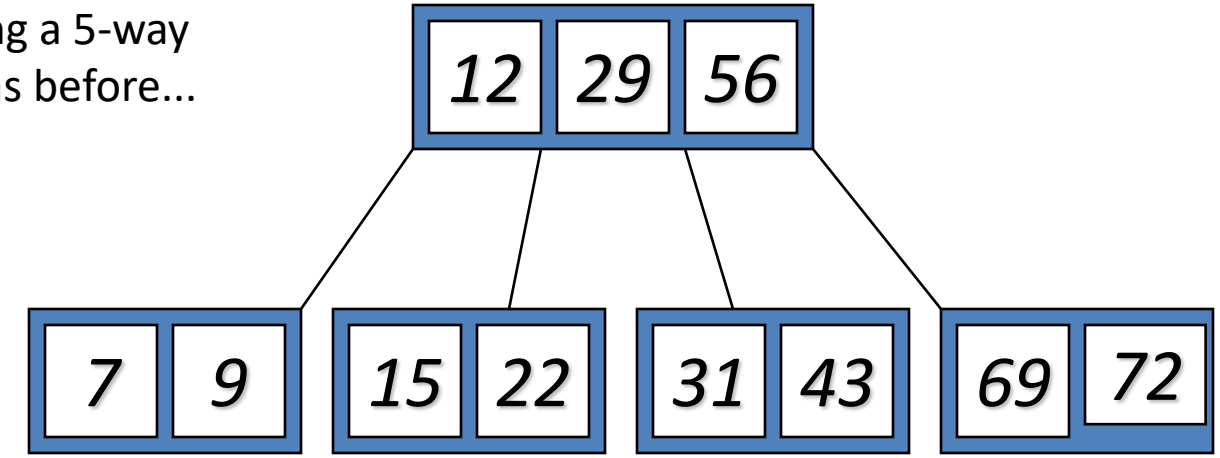
# Type #2: Simple non-leaf deletion

Assuming a 5-way
B-Tree, as before...



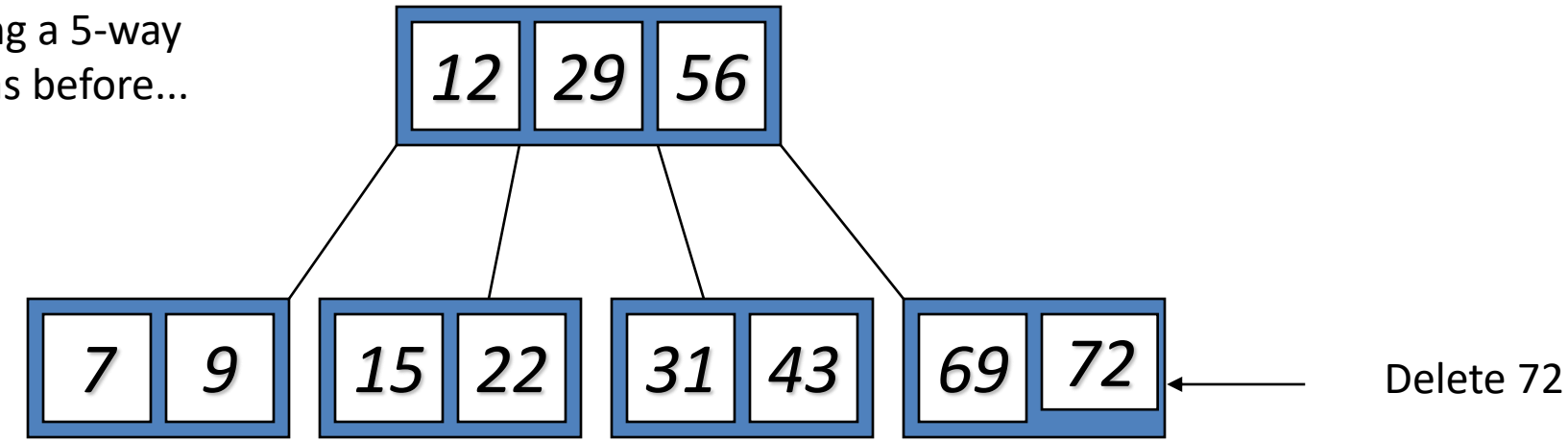
# Type #4: Too few keys in node and its siblings

Assuming a 5-way
B-Tree, as before...



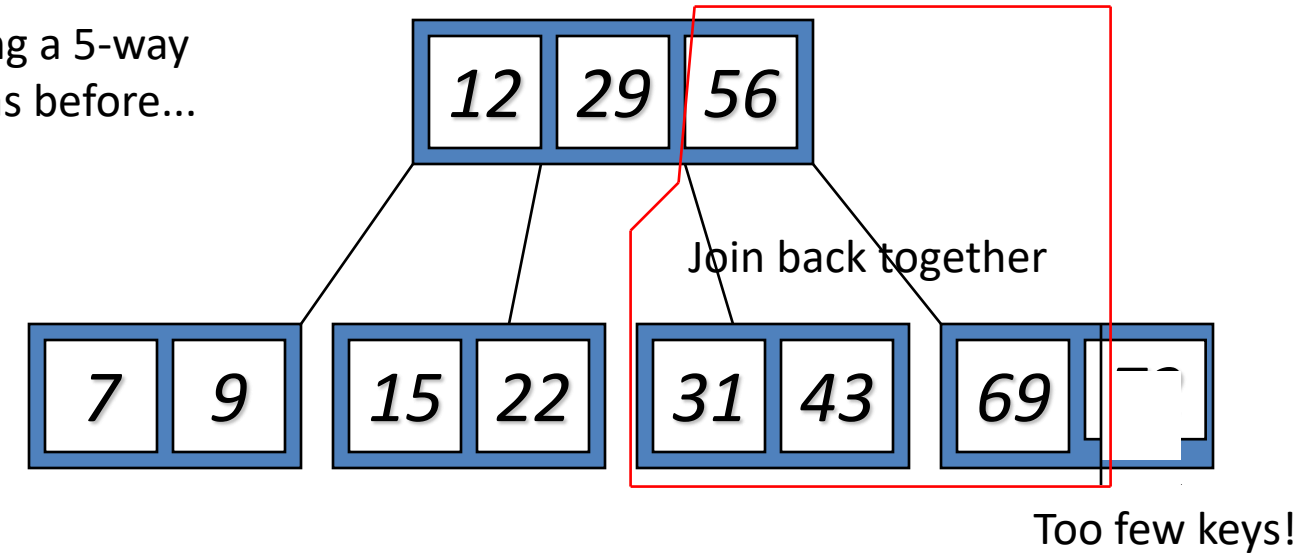
# Type #4: Too few keys in node and its siblings

Assuming a 5-way
B-Tree, as before...



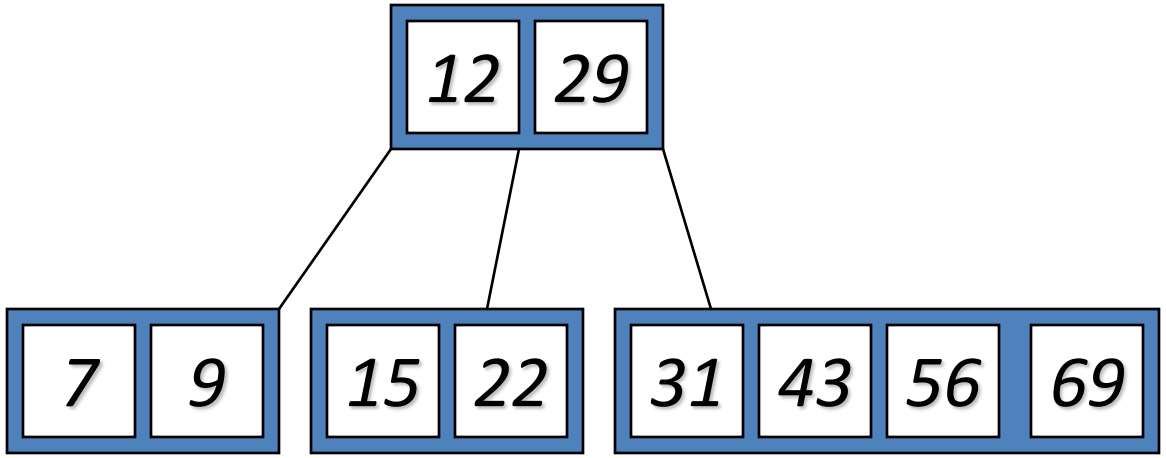
# Type #4: Too few keys in node and its siblings

Assuming a 5-way
B-Tree, as before...

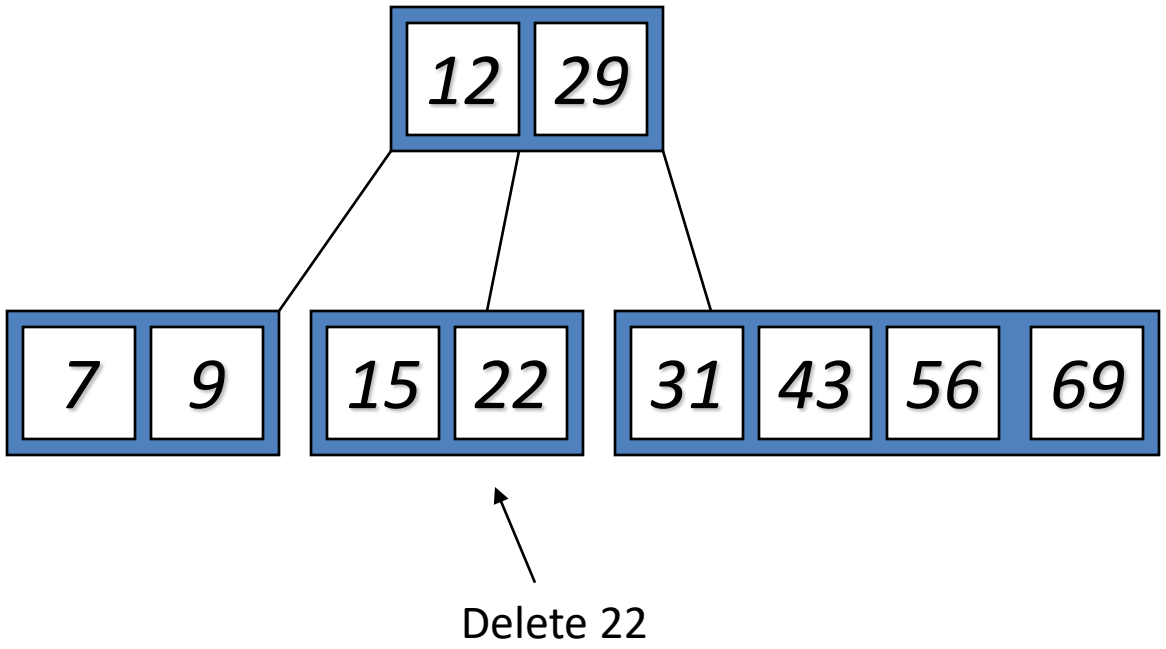



Too few keys!

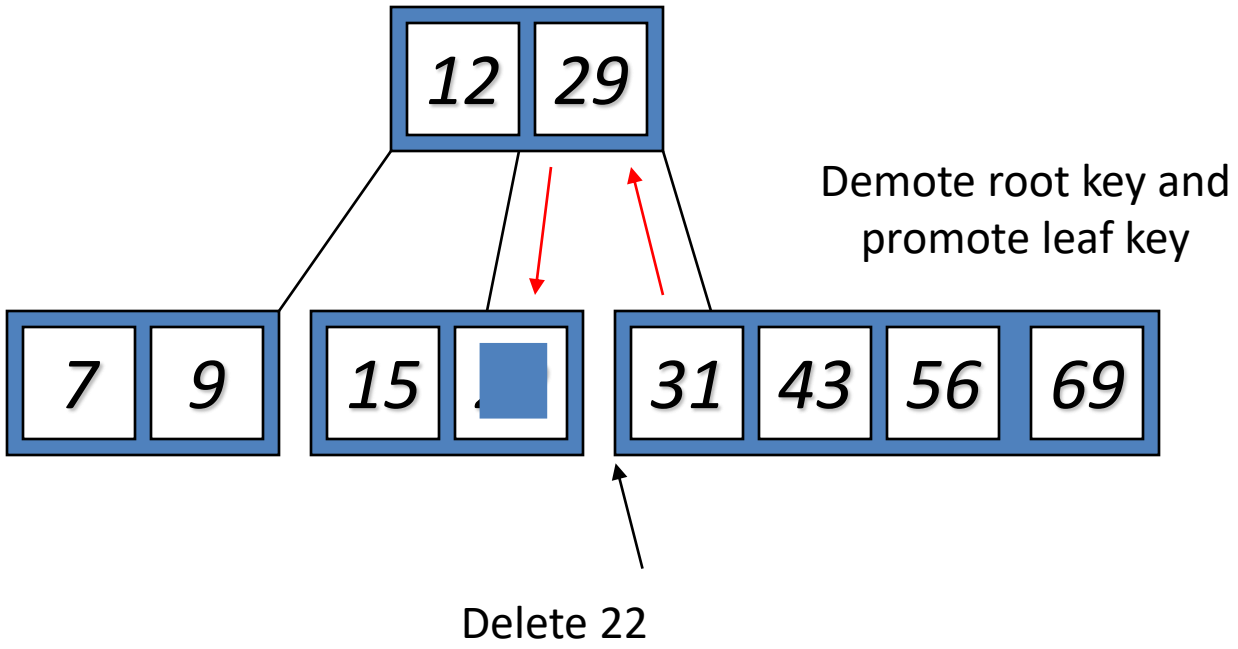
# Type #4: Too few keys in node and its siblings



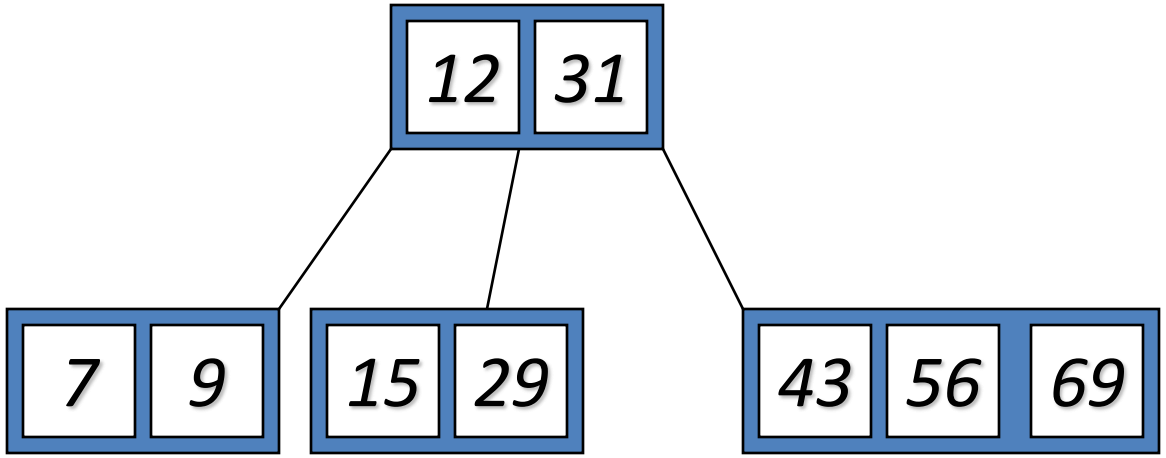
# Type #4: Too few keys in node and its siblings

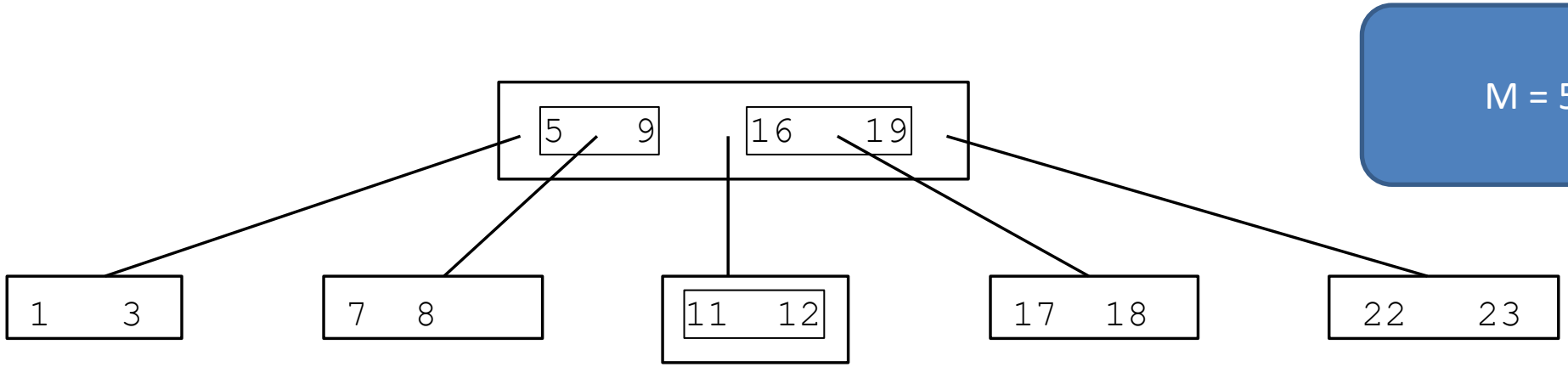


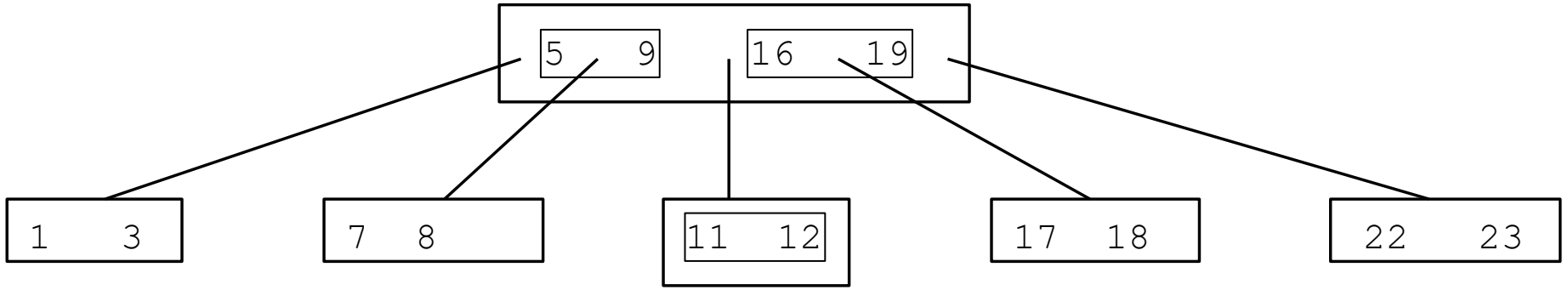
# Type #3: Enough siblings



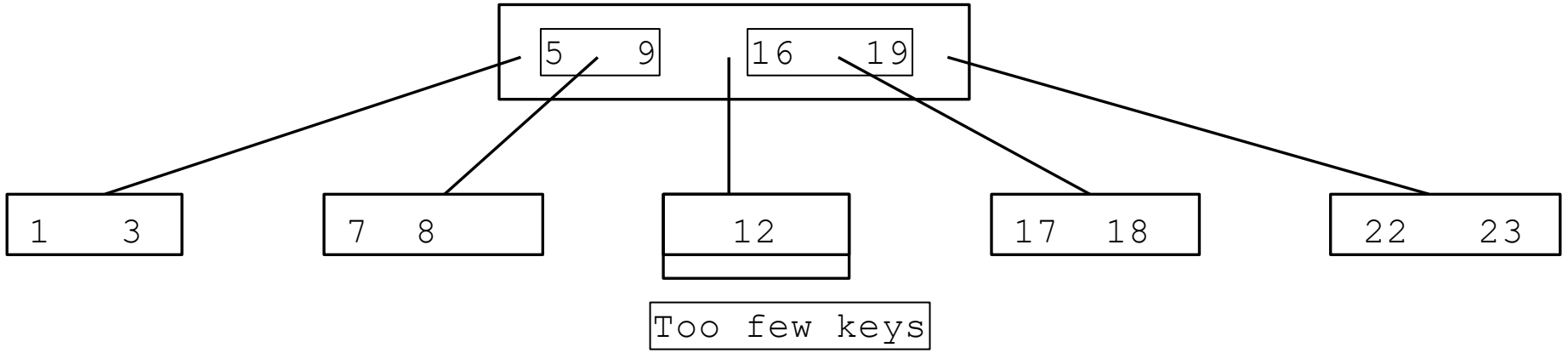
# Type #3: Enough siblings

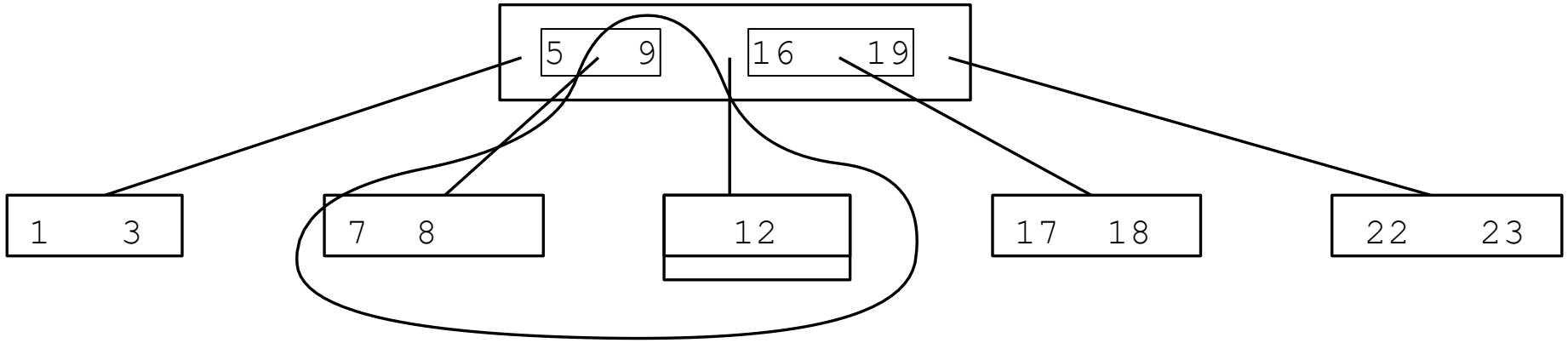



M = 5
5 9
16 19
1 3
7 8
11 12
17 18
22 23

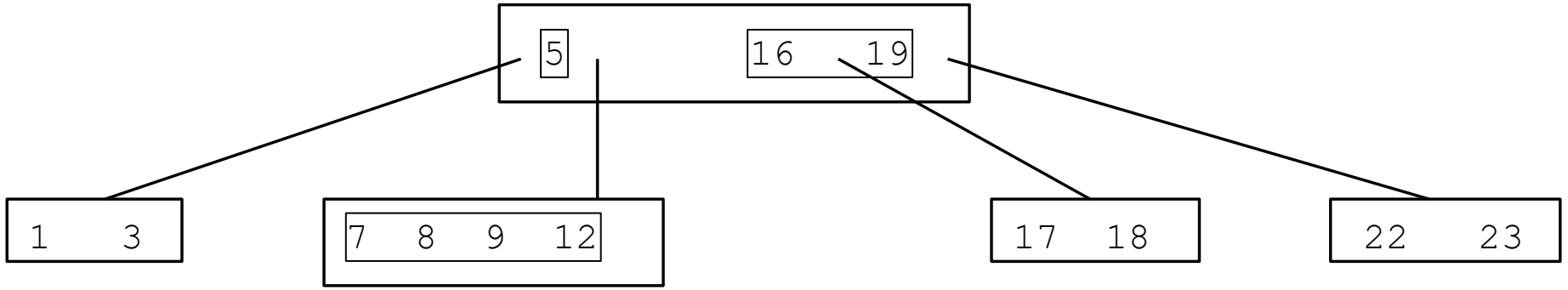


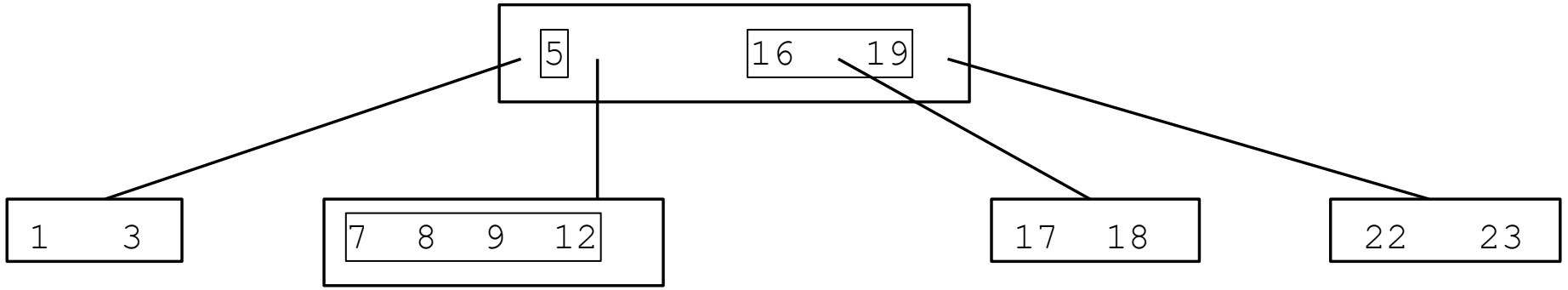

Delete the 11


5 9
16 19
1 3
7 8
12
17 18
22 23
Too few keys

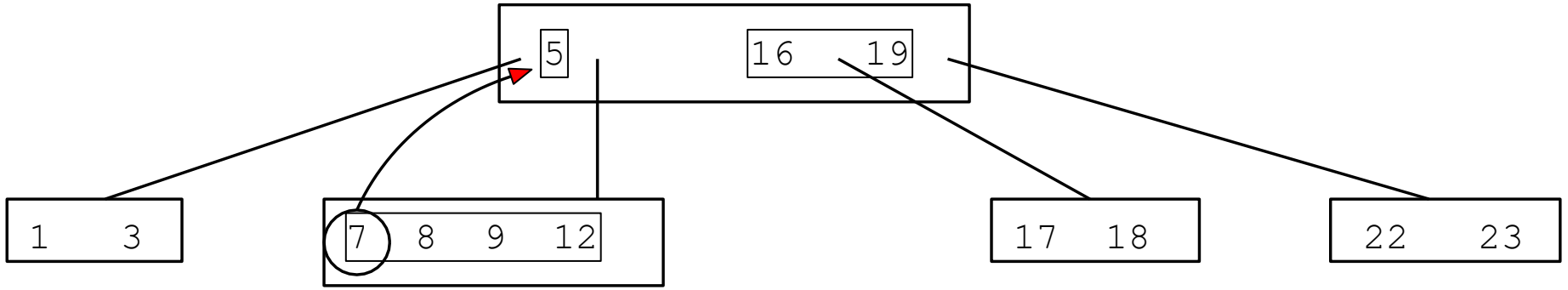



Merge with a sibling (pick the left sibling arbitrarily)

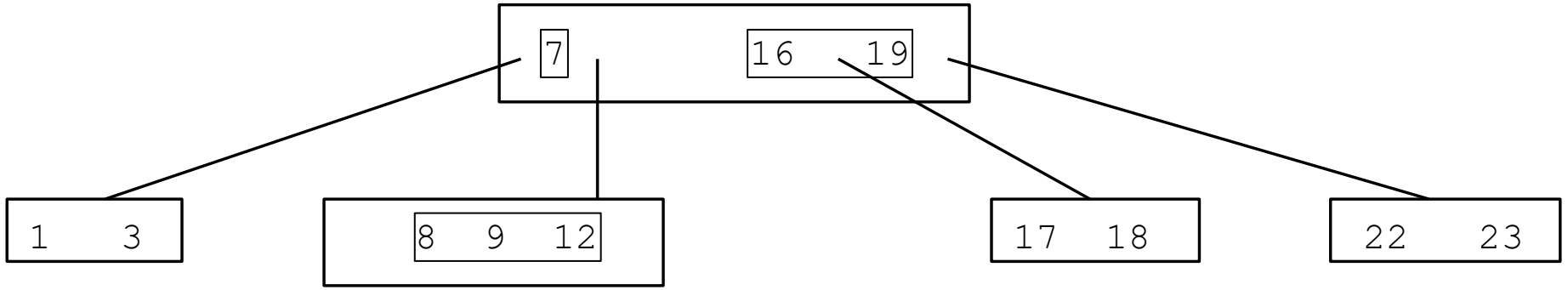

5
16 19
1 3
7 8 9 12
17 18
22 23

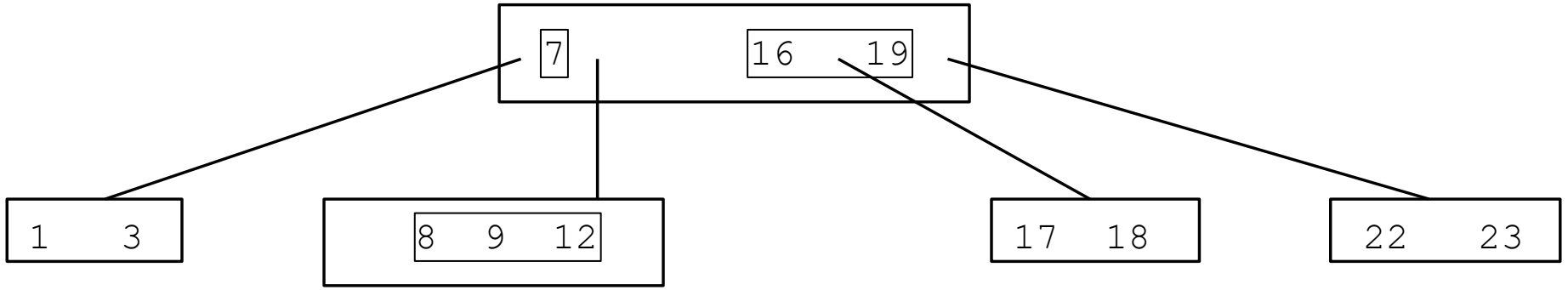



Remove the 5

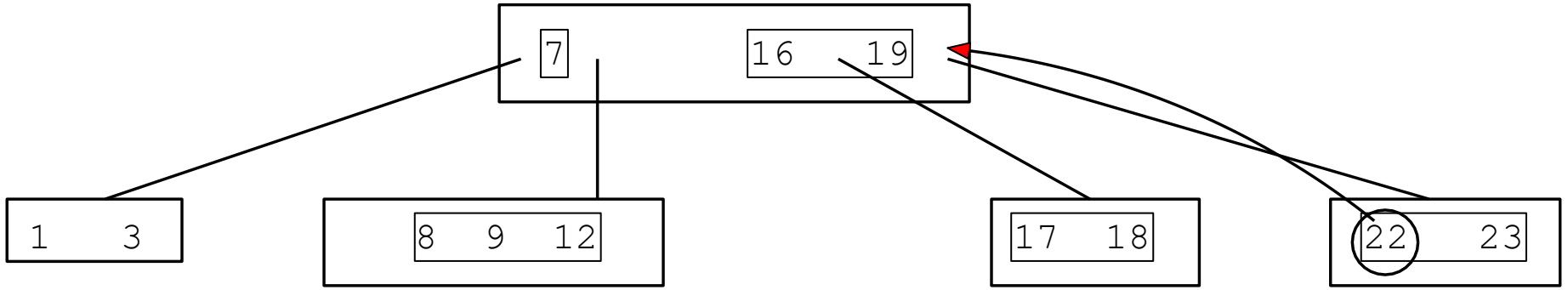



Remove the 5

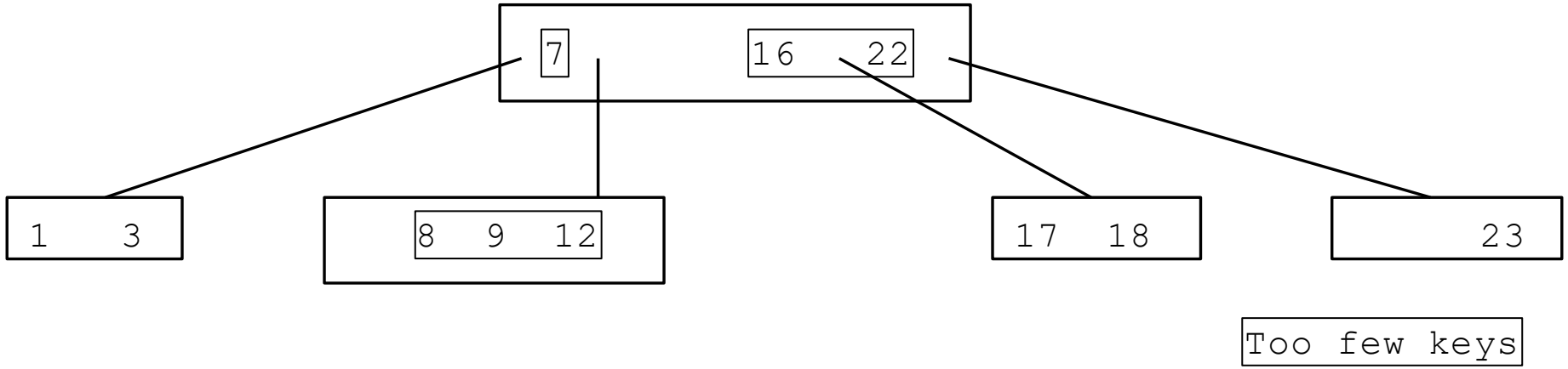

7
16 19
1 3
8 9 12
17 18
22 23

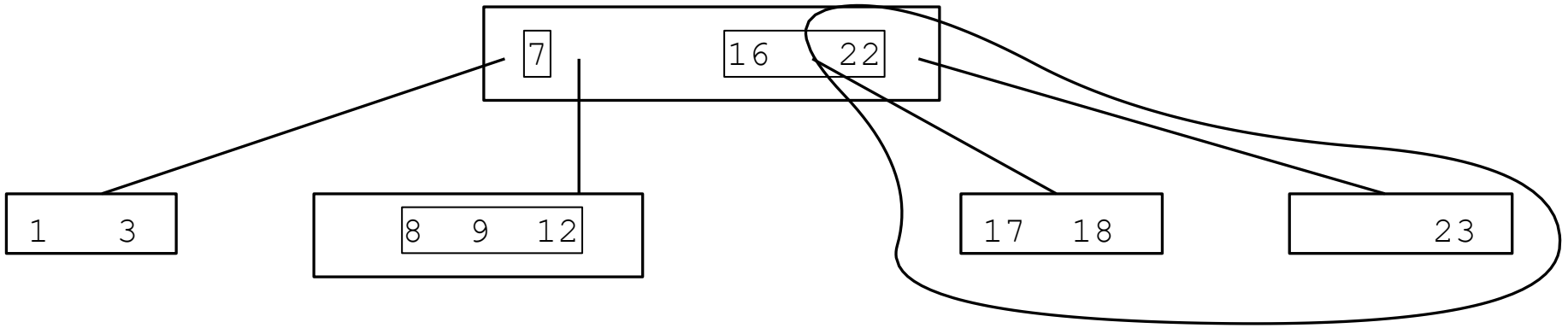



Remove the 19

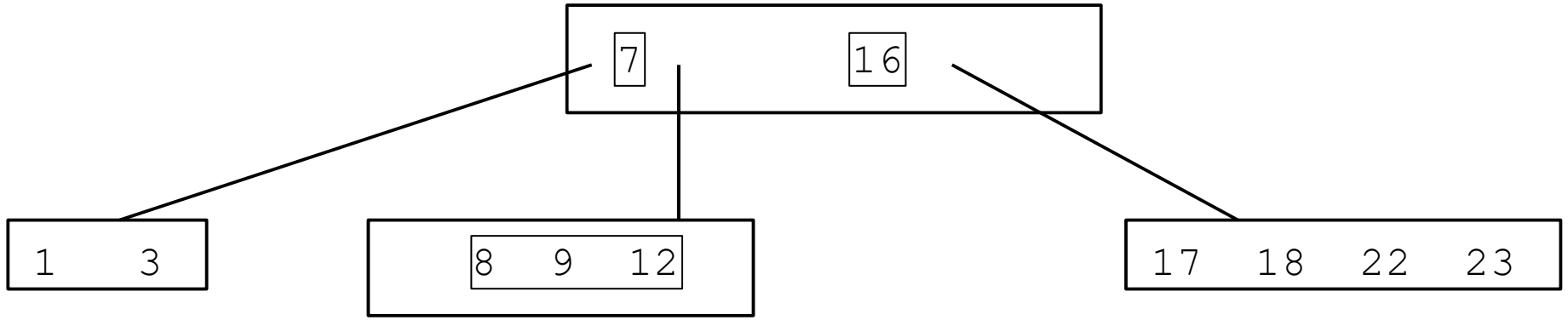



Remove the 19


7
16 22
1 3
8 9 12
17 18
23
Too few keys



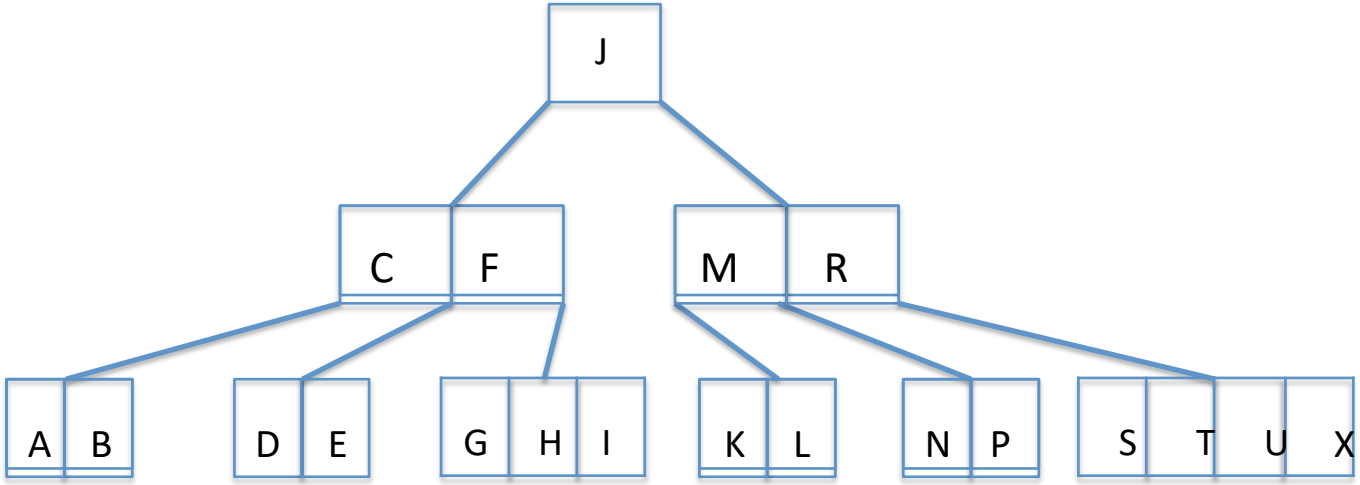

- Merge with left sibling


7
16
1 3
8 9 12
17 18 22 23

# Another Delete Example

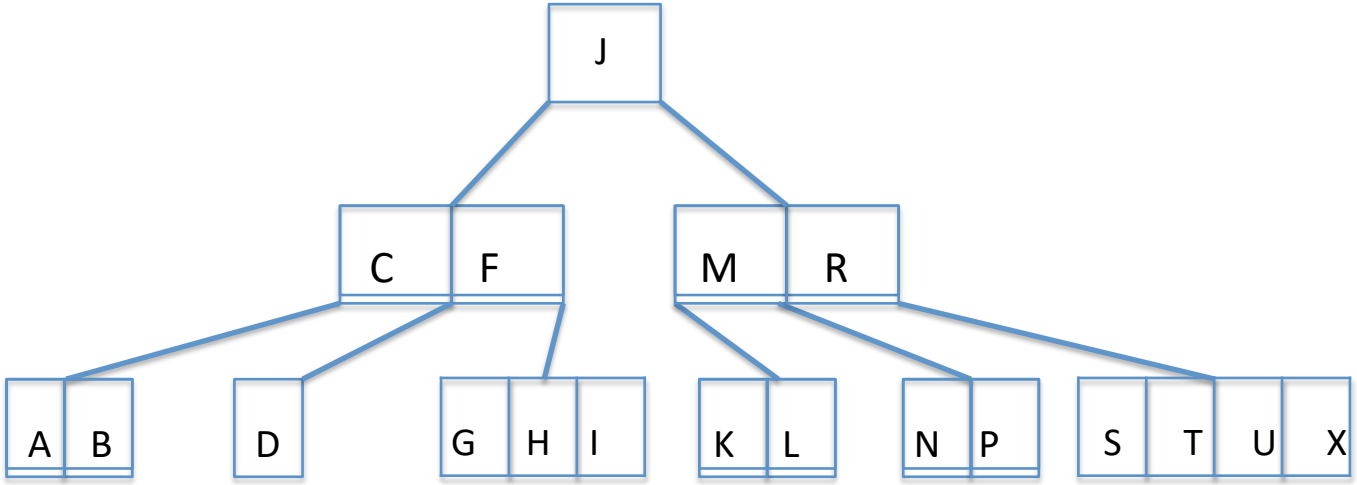
M = 5

- Delete E from leaf node



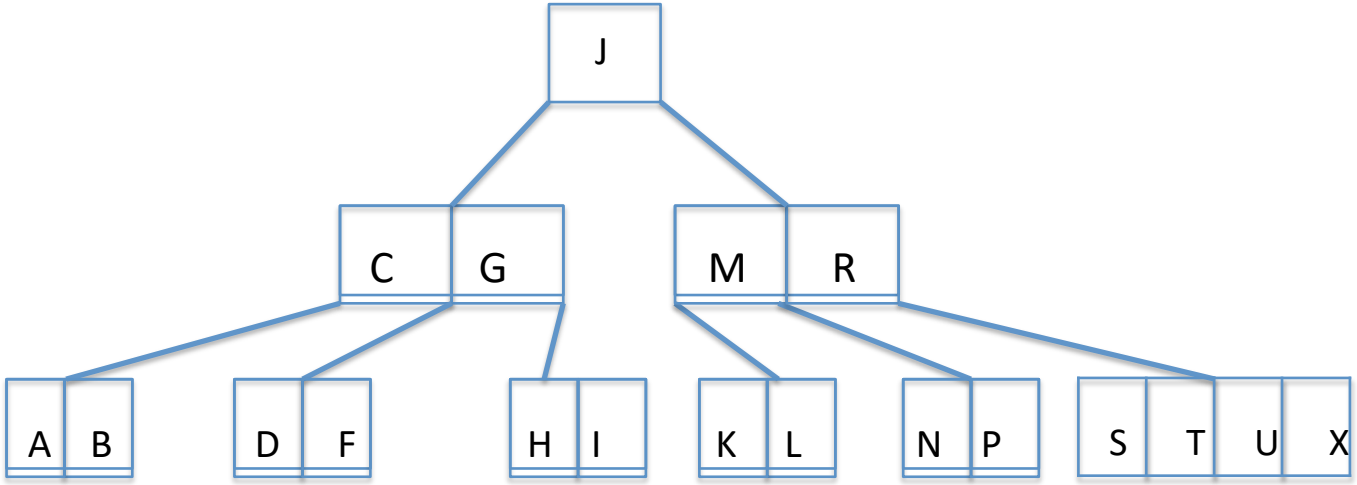
# Deleting Nodes

- Delete E



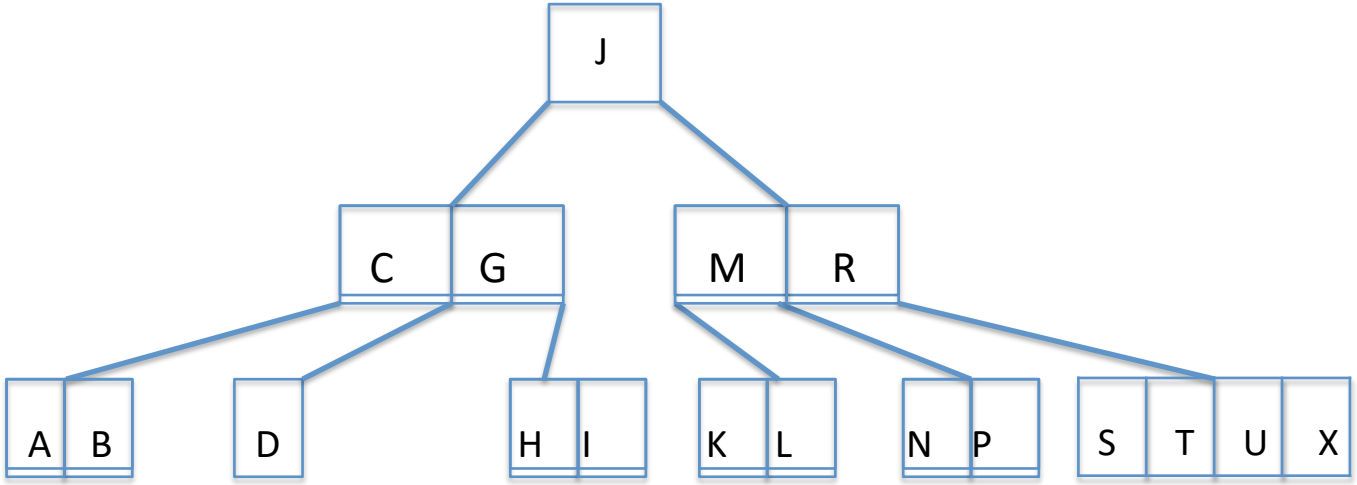
# Deleting Nodes

- Borrow from a neighbor



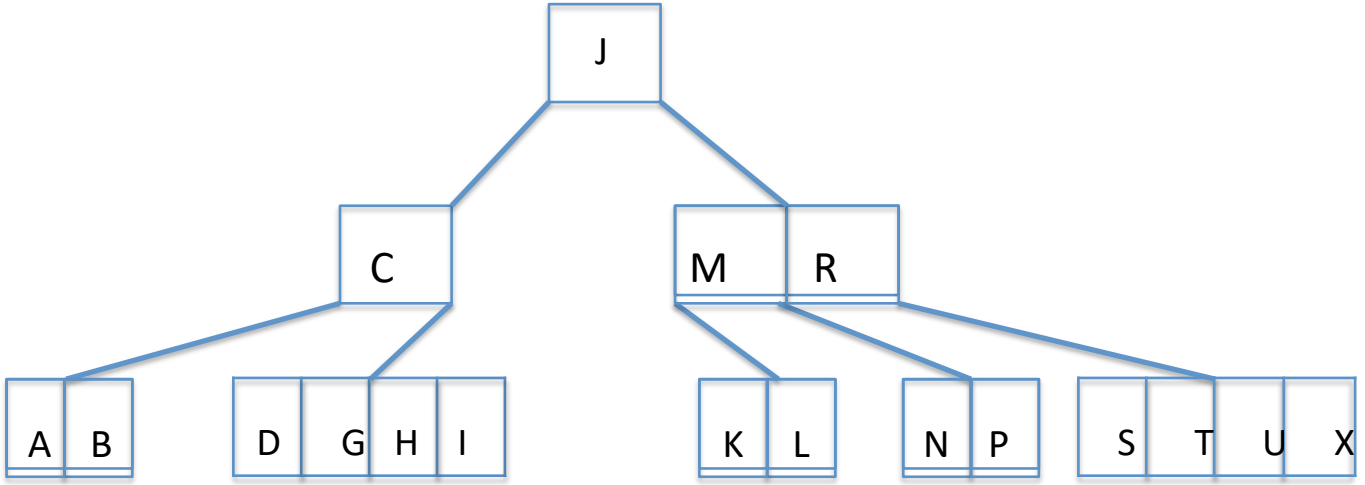
# Deleting Nodes

- Delete F — but can't borrow from a neighbor



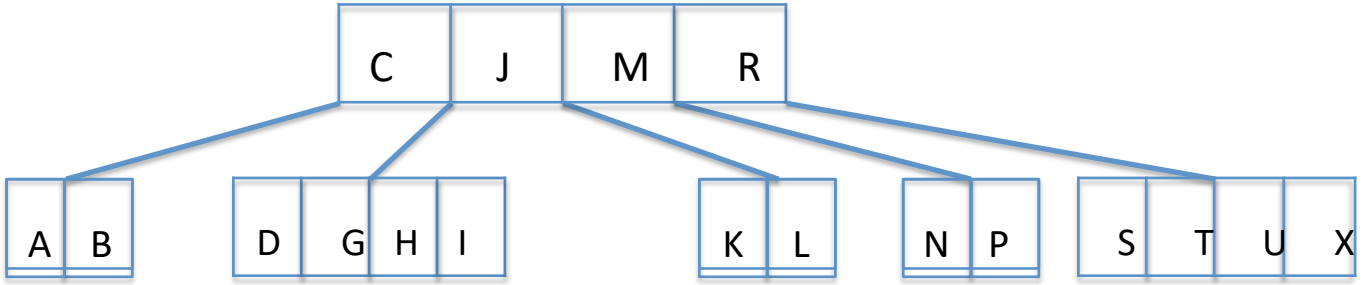
# Deleting Nodes

Combine and push the problem up one level



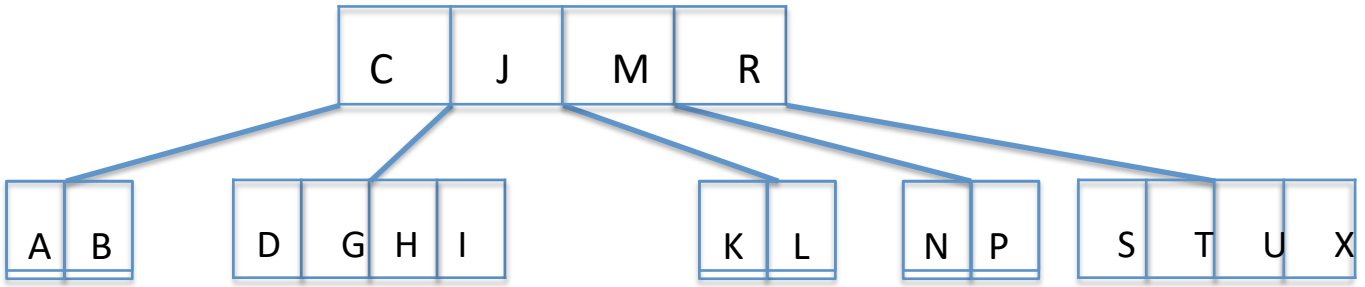
# Deleting Nodes

Can’t borrow so combine

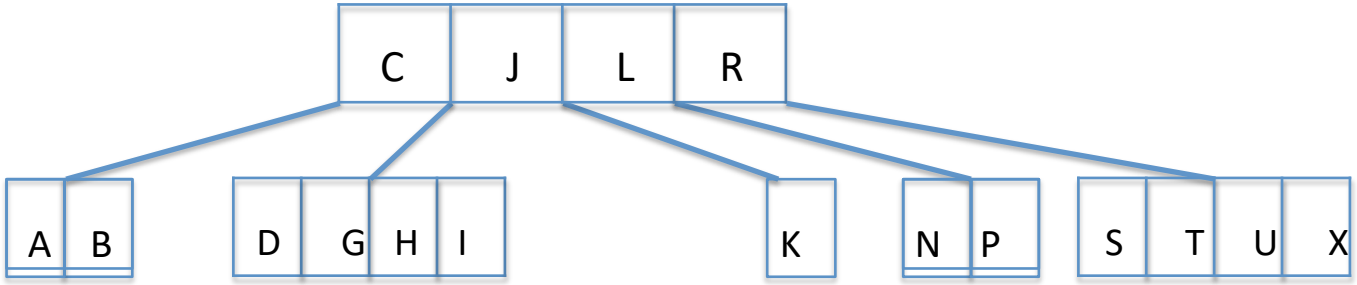


# Deleting Nodes

Delete M

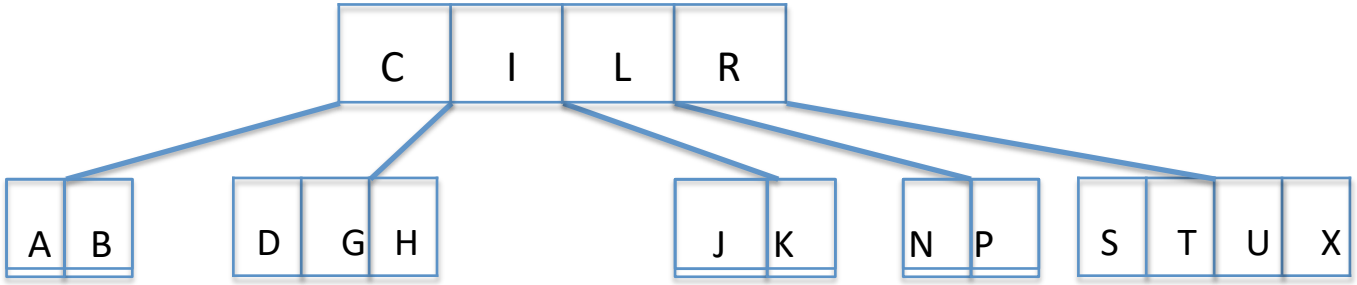


# Deleting Nodes



# Deleting Nodes

Borrow from a neighbor



# Comparing Trees

## Binary trees

- Can become *unbalanced* and *lose* their good time complexity (big O)
- AVL trees are strict binary trees that *overcome the balance problem*
- Heaps remain balanced but only *prioritise* (not order) the keys

## Multi-way trees

- B-Trees can be *m*-way, they can have any number of children
- One B-Tree, the 2-3 (or 3-way) B-Tree, *approximates* a permanently balanced binary tree, exchanging the AVL tree's balancing operations for insertion and (more complex) deletion operations