B-trees

Motivation

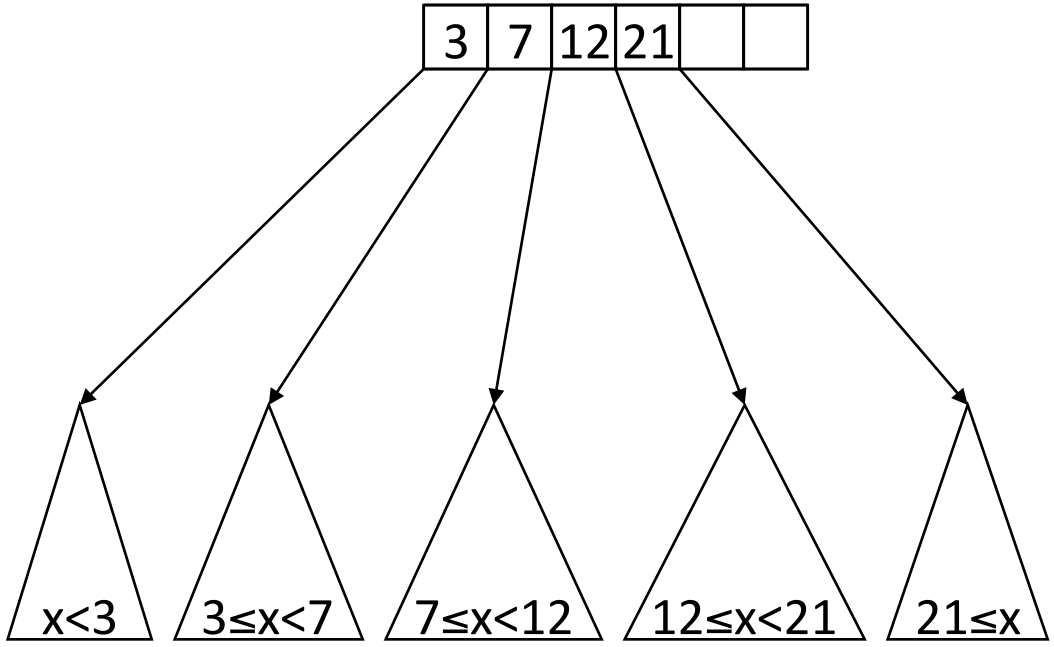
- Index structures for large datasets cannot be stored in main memory
- Storing it on disk requires different approach to efficiency (accessing a disk is limited by rpm specs, including mechanical factors)

- Now, assume we use an AVL tree to store about 20 million records
- We end up with a **very** deep binary tree with lots of different disk accesses; $\log_2 20{,}000{,}000$ is about 24, so this takes about 0.2 seconds

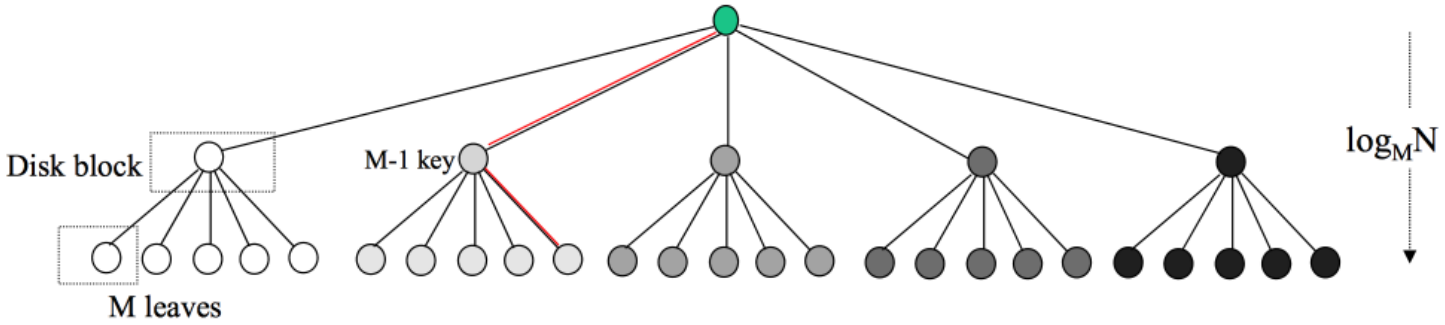
- We know we can't improve on the log *n* lower bound on search for a binary tree
- But, the solution is to use more branches and thus reduce the height of the tree
  - As branching increases, depth decreases

B-tree definition

- Each node has many keys
  - Sub-tree between two keys *x* and *y* contains values *v* such that $x \leq v < y$
  - binary search within a node to find correct subtree
- Each node takes one full {*page, block, line*} of memory



B-tree definition



- B-tree of order *m* is an *m*-way tree (i.e., a tree where each node may have up to *m* children, *m* being always odd) 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

construction

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

- Suppose we start with an empty B-tree and keys arrive in the following order:

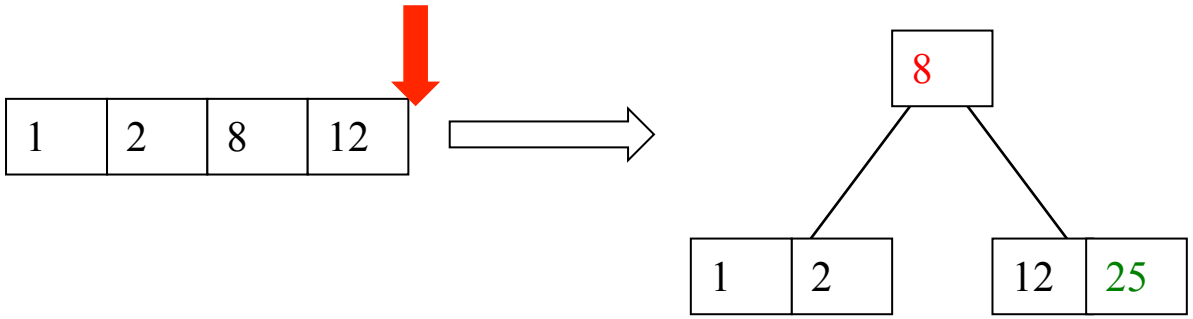
  1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45

- We want to construct a B-tree of order 5
- The first four items go into the root:

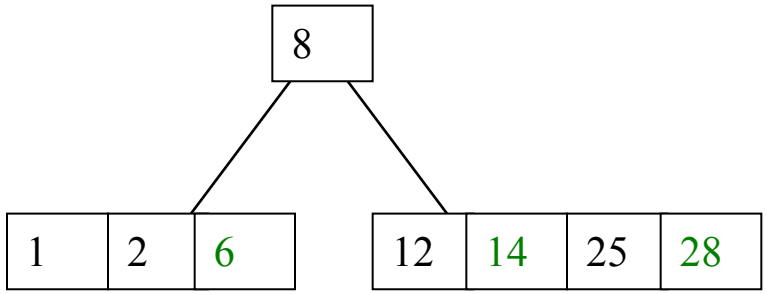
1	2	8	12

- 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

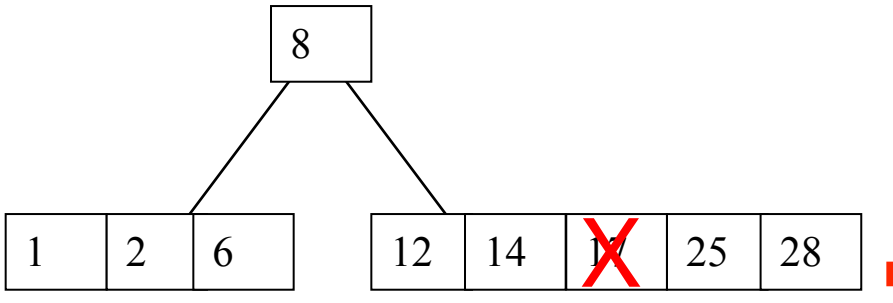
1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45



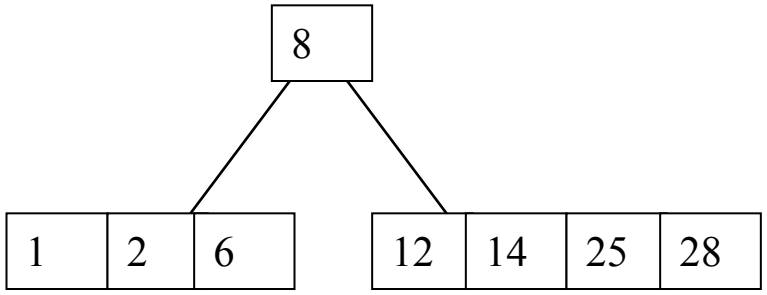
1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45



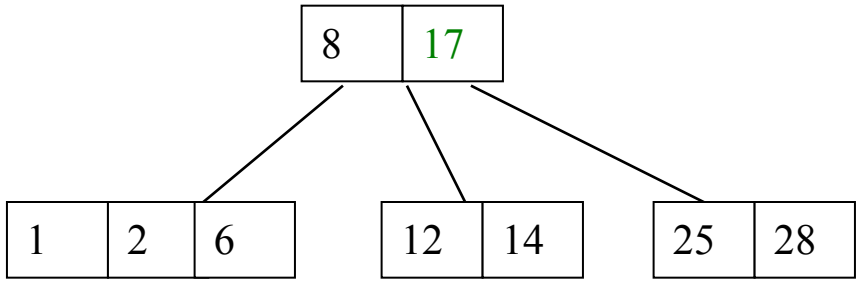
1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45



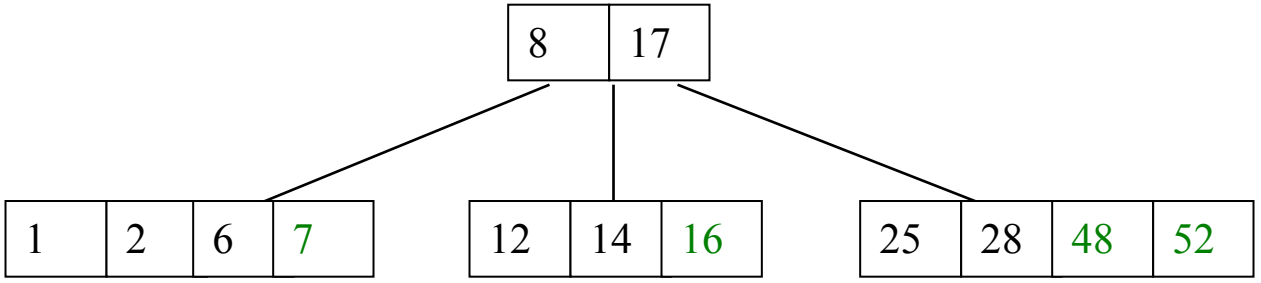
1. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
2. all leaves are on the same level
3. all non-leaf nodes except the root have at least $\lceil m / 2 \rceil$ children
4. the root is either a leaf node, or it has from two to m children
5. a leaf node contains no more than $m - 1$ keys

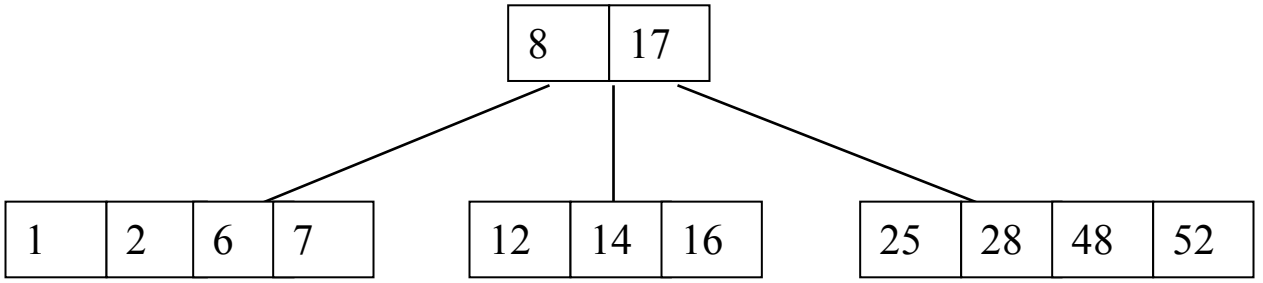


1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45

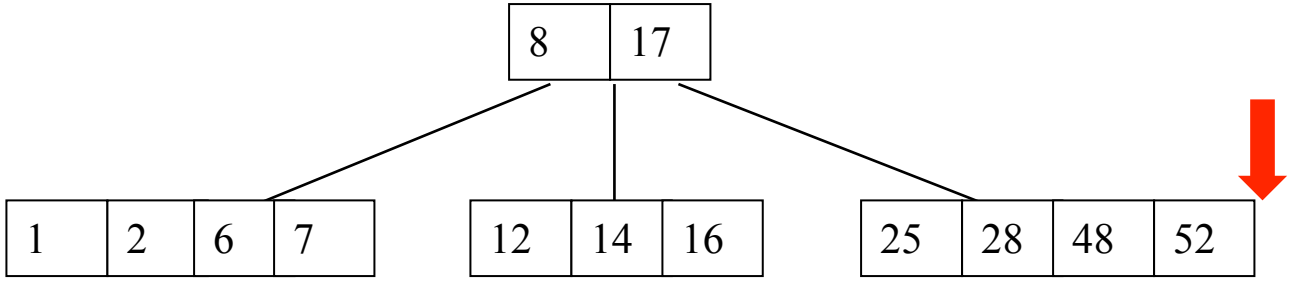


1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45

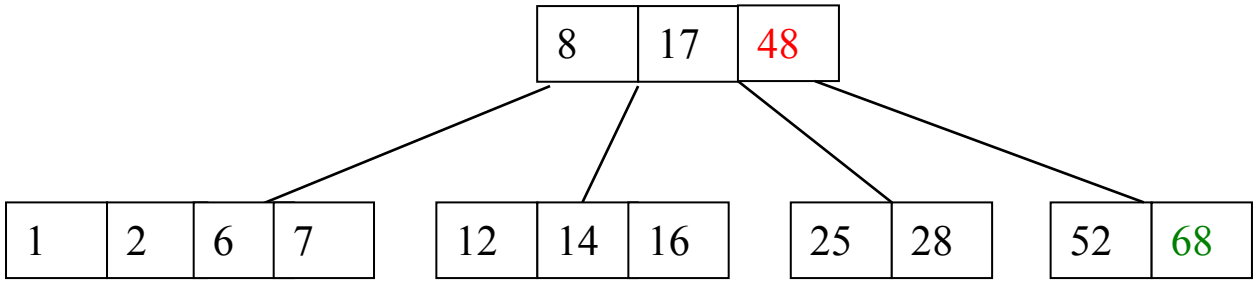


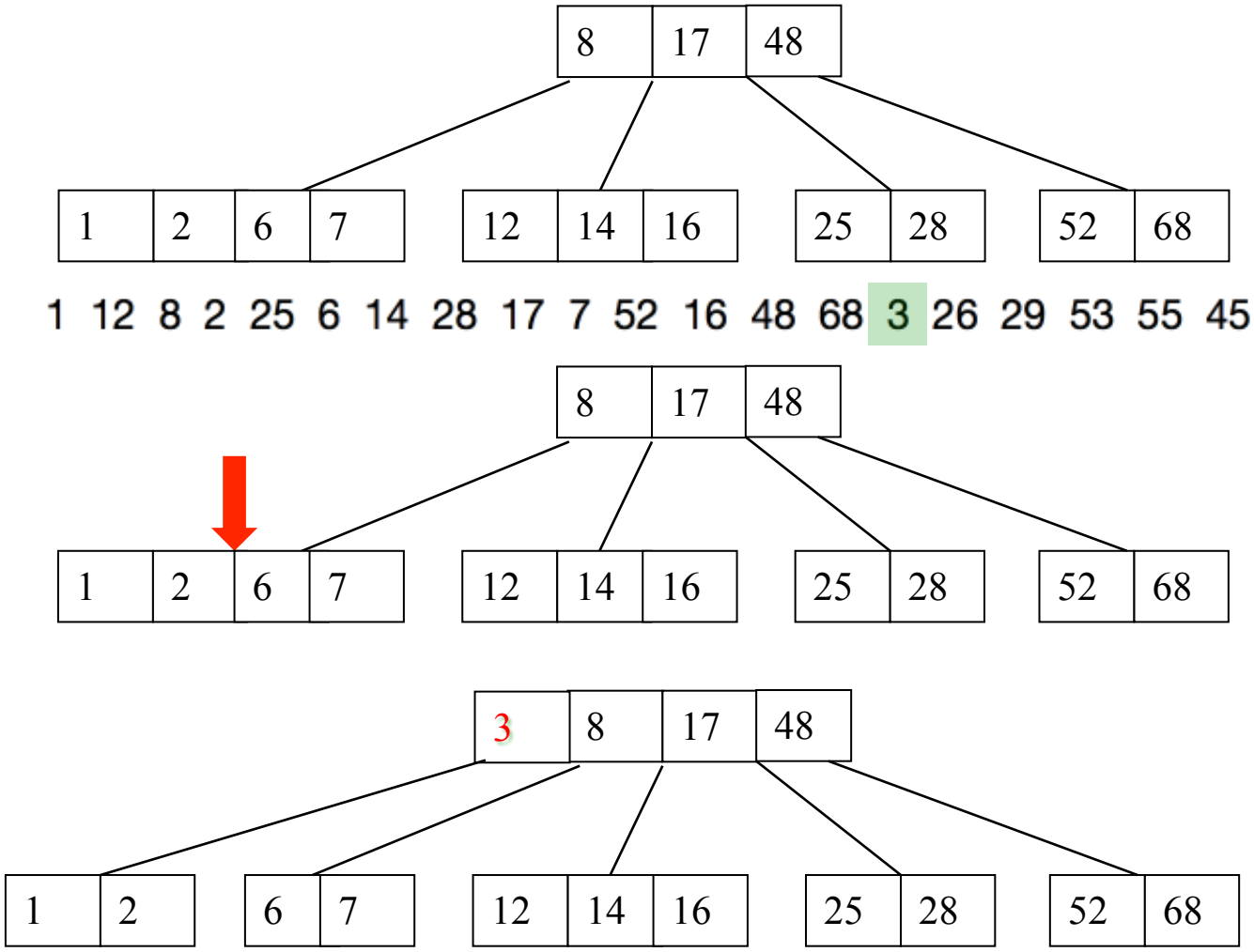


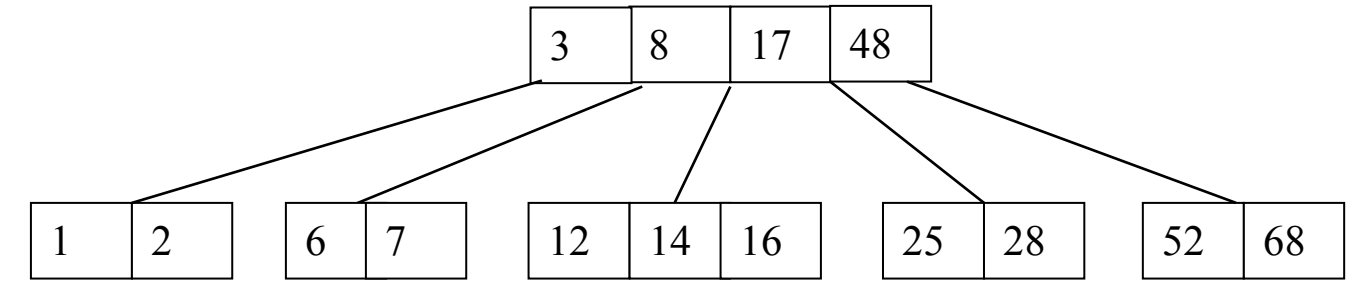
1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45



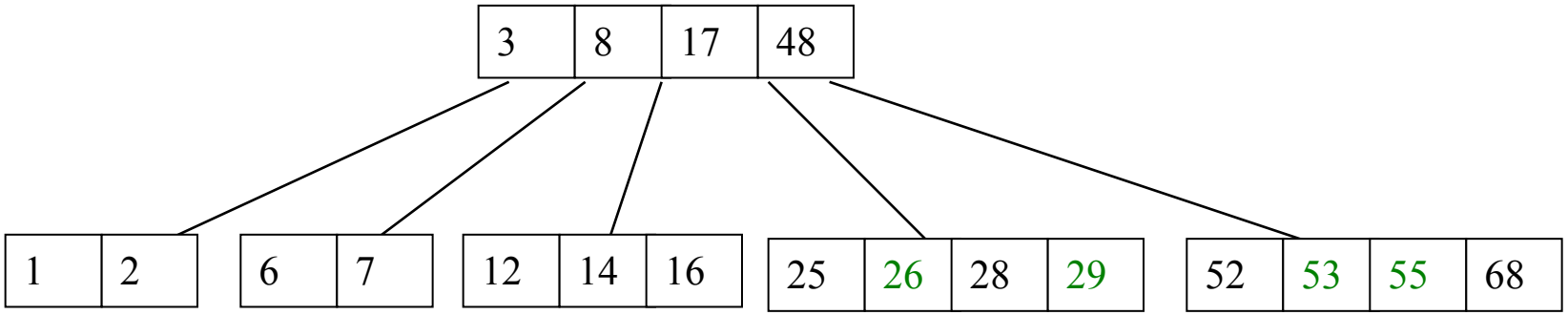
1. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
2. all leaves are on the same level
3. all non-leaf nodes except the root have at least $\lceil m / 2 \rceil$ children
4. the root is either a leaf node, or it has from two to m children
5. a leaf node contains no more than $m - 1$ keys



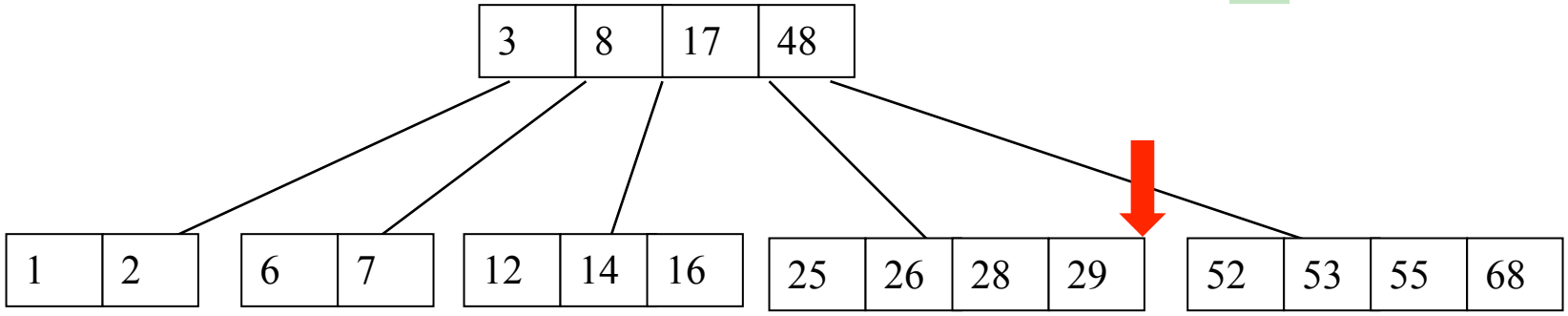

8 17 48
1 2 6 7
12 14 16
25 28
52 68
1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45
8 17 48
1 2 6 7
12 14 16
25 28
52 68
3 8 17 48
1 2
6 7
12 14 16
25 28
52 68

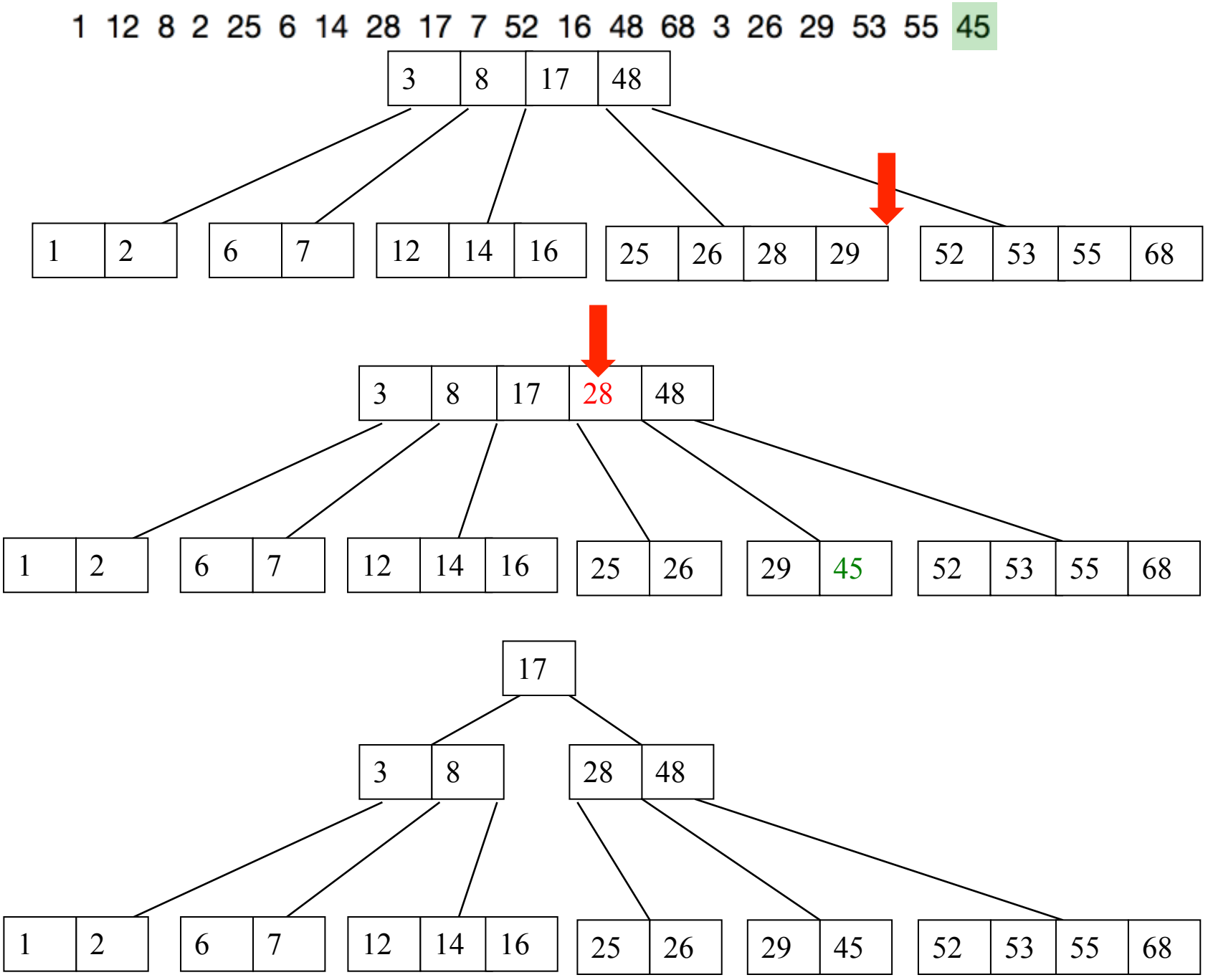


1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45

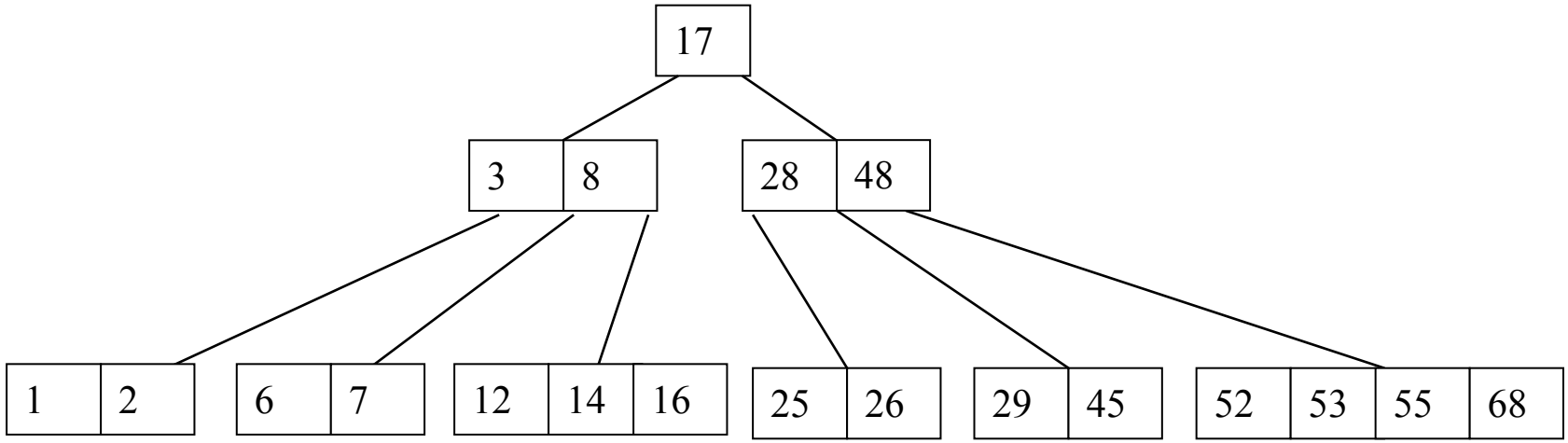


1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45




1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45
3 8 17 48
1 2
6 7
12 14 16
25 26 28 29
52 53 55 68
3 8 17 28 48
1 2
6 7
12 14 16
25 26
29 45
52 53 55 68
17
3 8
28 48
1 2
6 7
12 14 16
25 26
29 45
52 53 55 68

checking



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

insertion

- Attempt to insert the new key into a leaf
- If this results in that leaf becoming too big (>*m*-1), split the leaf into two, promoting the middle key to the leaf's parent
- If this results in the parent becoming too big (>*m*-1), split the parent into two, promoting the middle key
- This strategy should 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

exercise

Insert the following keys to a 5-way B-tree:

3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56

removal

During insertion, the key always goes *into* a *leaf*. For deletion we wish to remove *from* a leaf. The way we can do that:

1 - If the key is already in a leaf node, and removing it does not cause that leaf node to have too few keys (<floor[m/2]), then simply remove the key to be deleted.

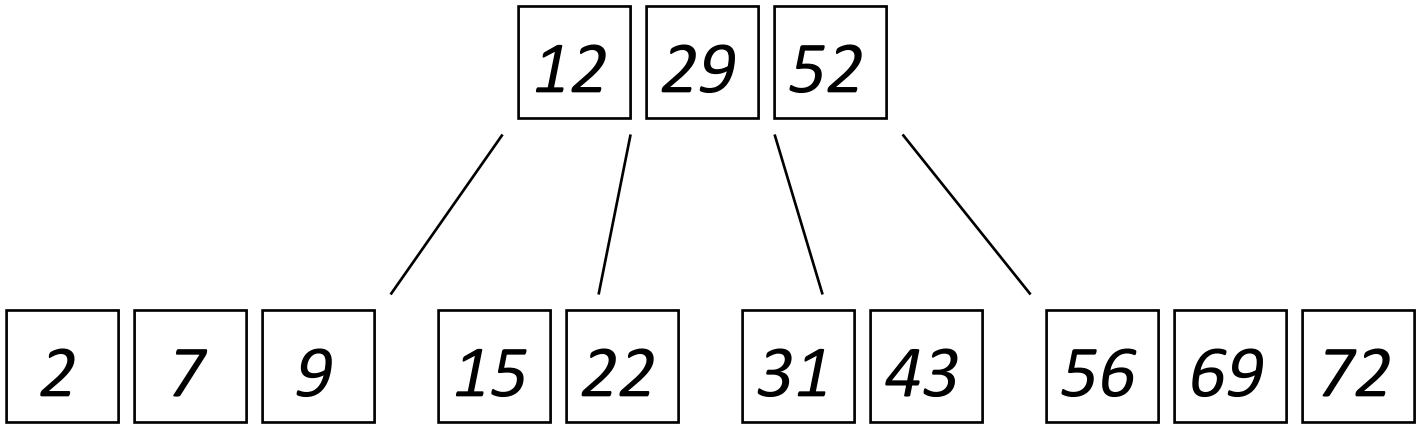
2 - If the key is *not* in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf; in this case we can delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.

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:

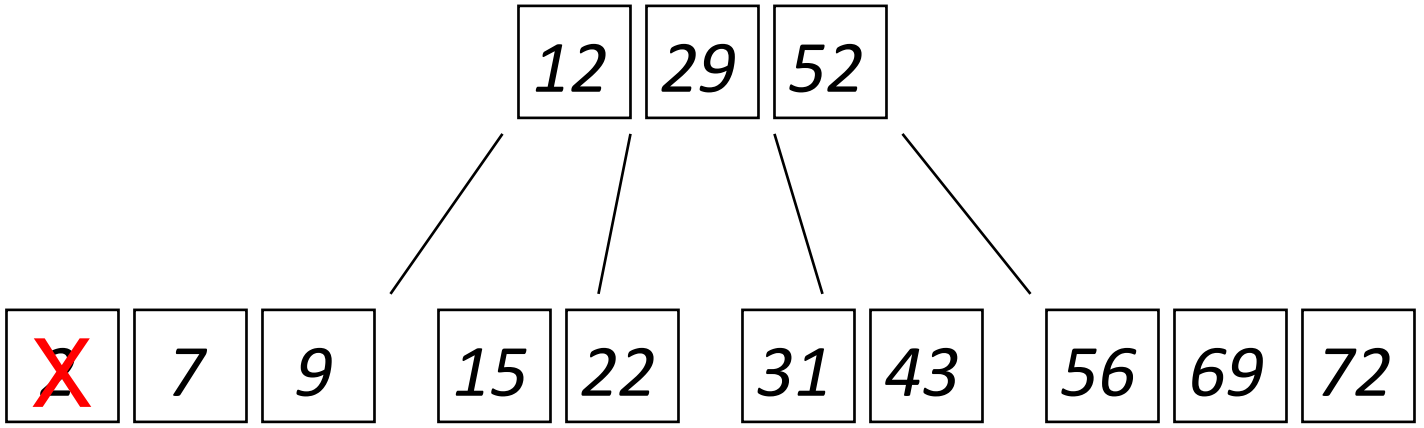
3 - if one of them has more than the min number of keys then we can promote one of its keys to the parent and take the parent key into our lacking leaf

4 - 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 leaves the parent with too few keys then we repeat the process up to the root itself, if needed

example

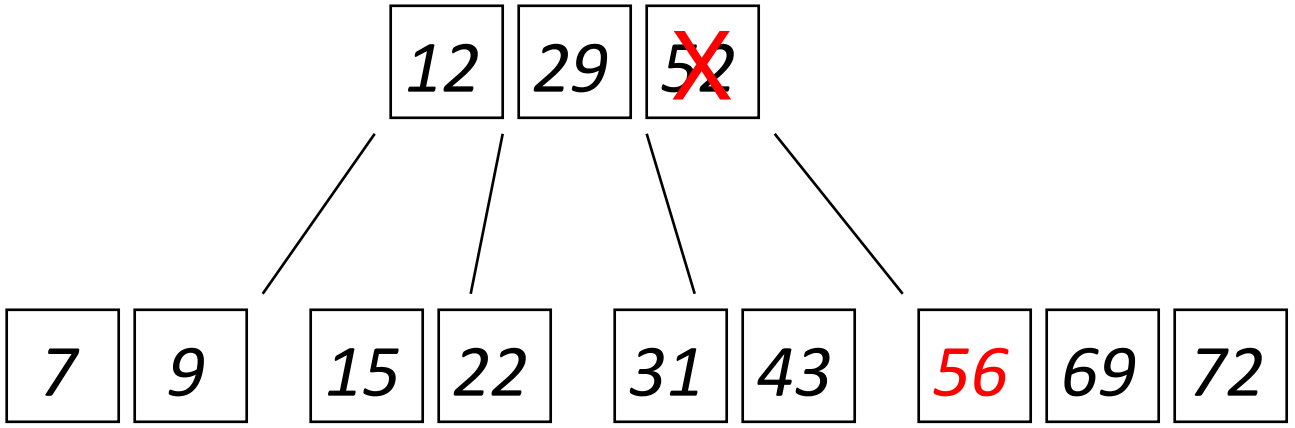


delete key = 2

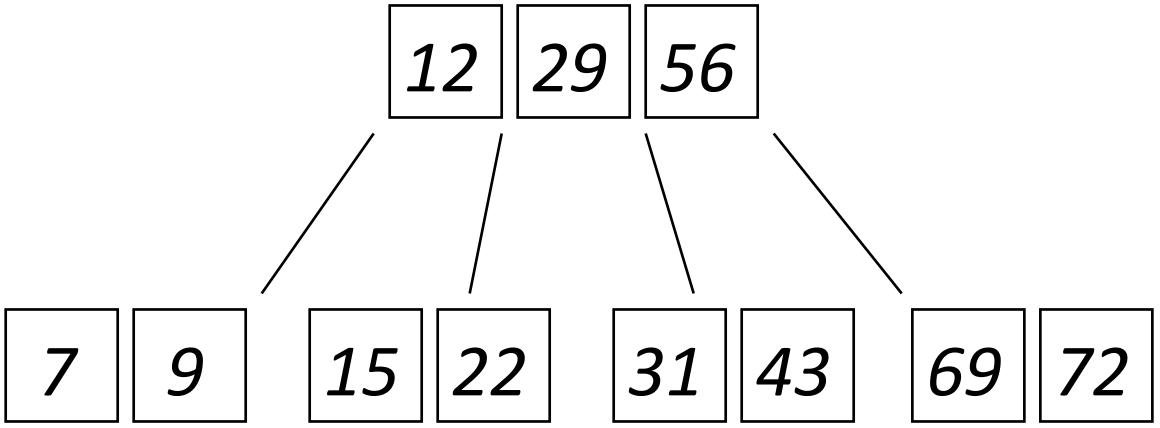


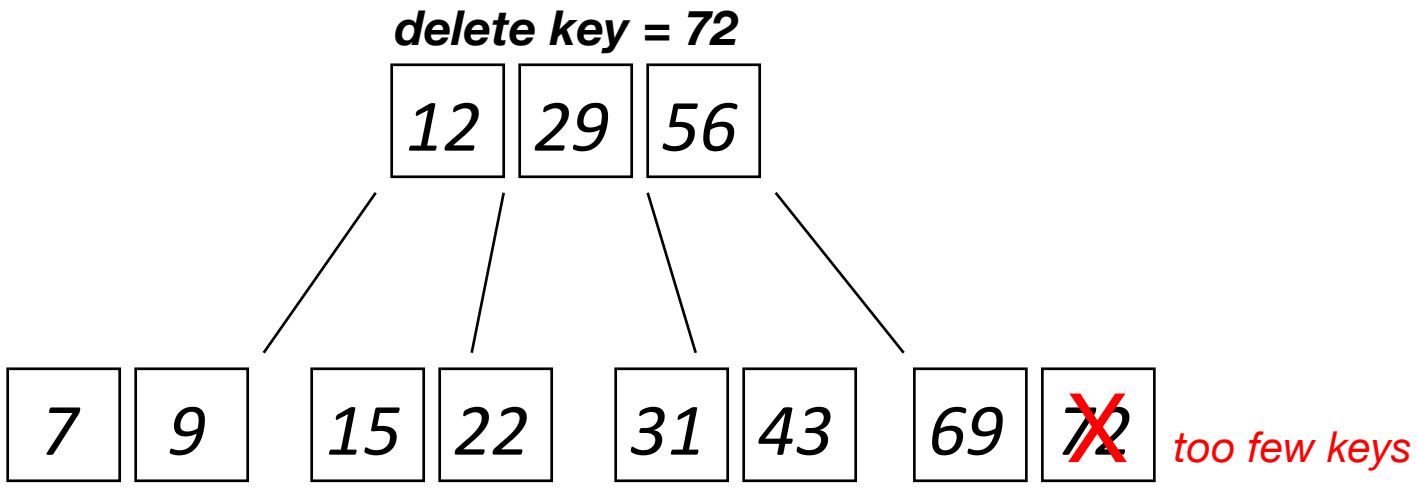
1 - If the key is already in a leaf node, and removing it does not cause that leaf node to have too few keys (<floor[m/2]), then simply remove the key to be deleted.

delete key = 52



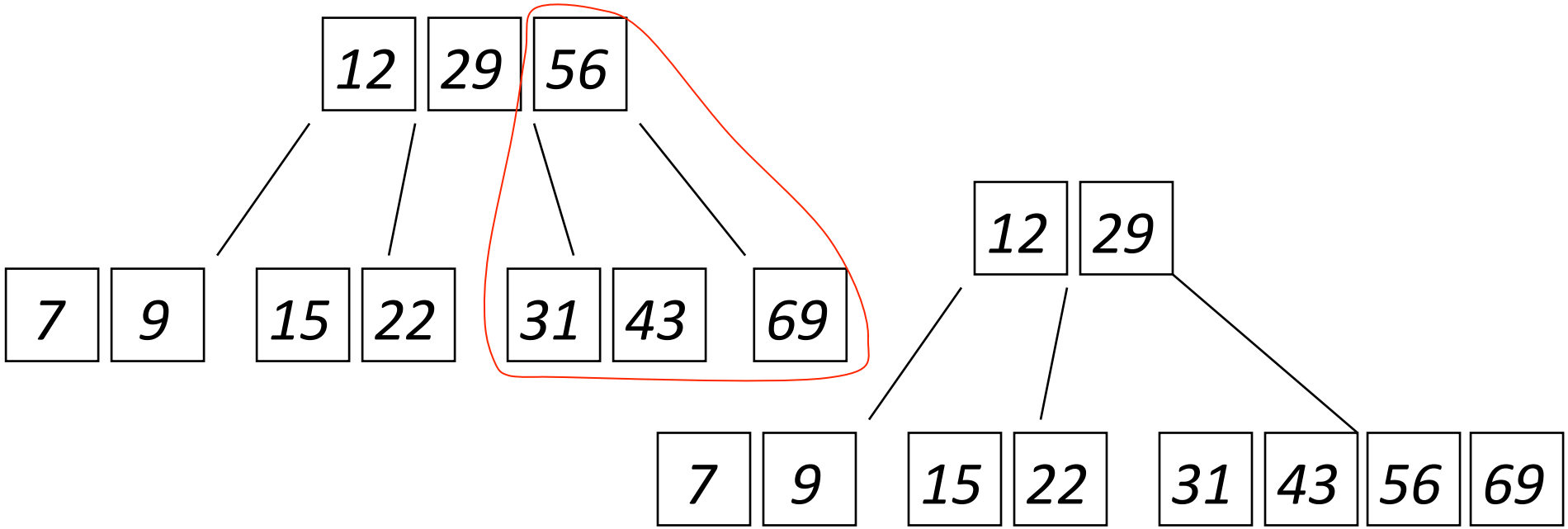
2 - If the key is not in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf; in this case we can delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.

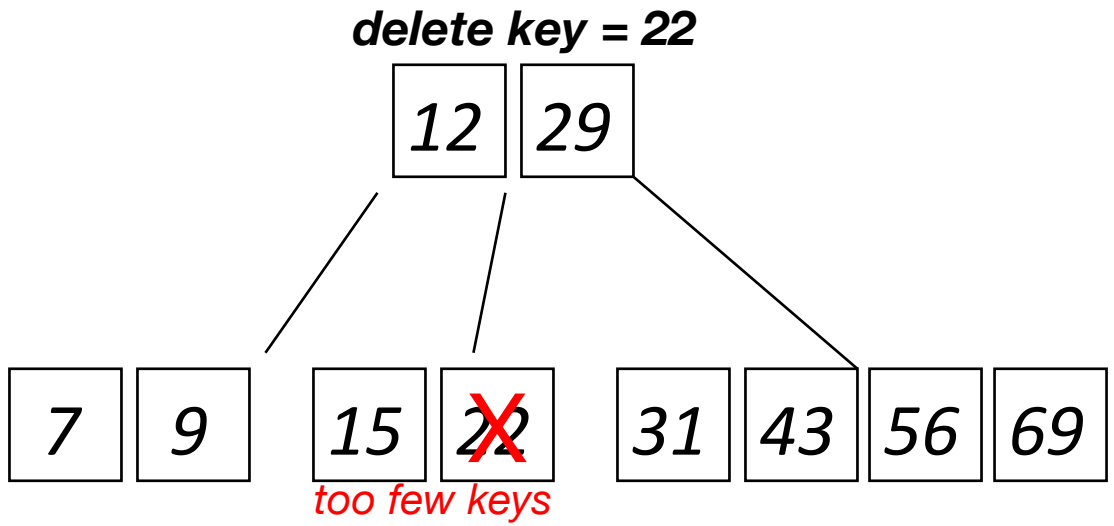




If (1) or (2) lead to a leaf node containing less than the minimum number of keys then we have to look at the siblings immediately adjacent to the leaf in question:

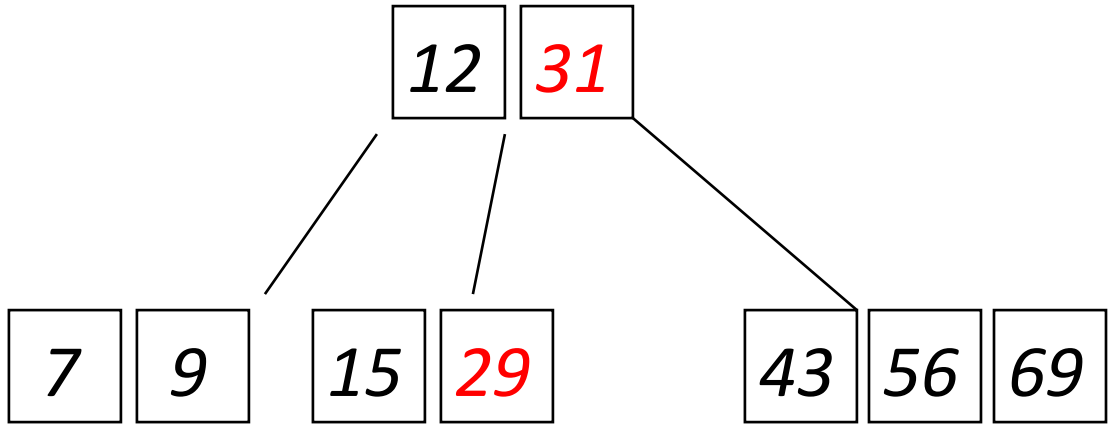
4 - if neither of them has more than the min number of keys then the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key) and the new leaf will have the correct number of keys; if this step leaves the parent with too few keys then we repeat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





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:

3 - 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



exercise

Given 5-way B-tree created by these data:

3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56

Add these further keys: 2, 6,12

Delete these keys: 4, 5, 7, 3, 14

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*

- Multi-way trees
  - B-Trees can be *m*-way, they can have any (odd) number of children
  - B-Trees can approximate a permanently balanced binary tree, exchanging the AVL tree balancing operations for insertion and more complex deletion operations