ITM 517 Algorithm

Ja-Hee Kim

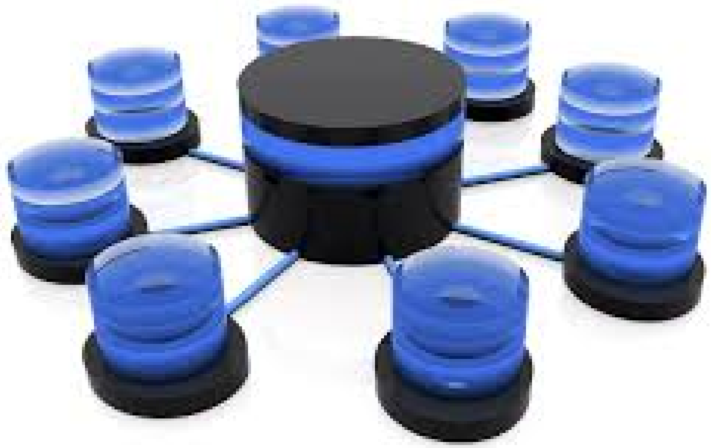
Tree

B tree

Visualization: https://www.cs.usfca.edu/~galles/visualization/BTree.html
Source code: https://www.geeksforgeeks.org/introduction-of-b-tree-2/

Motivation

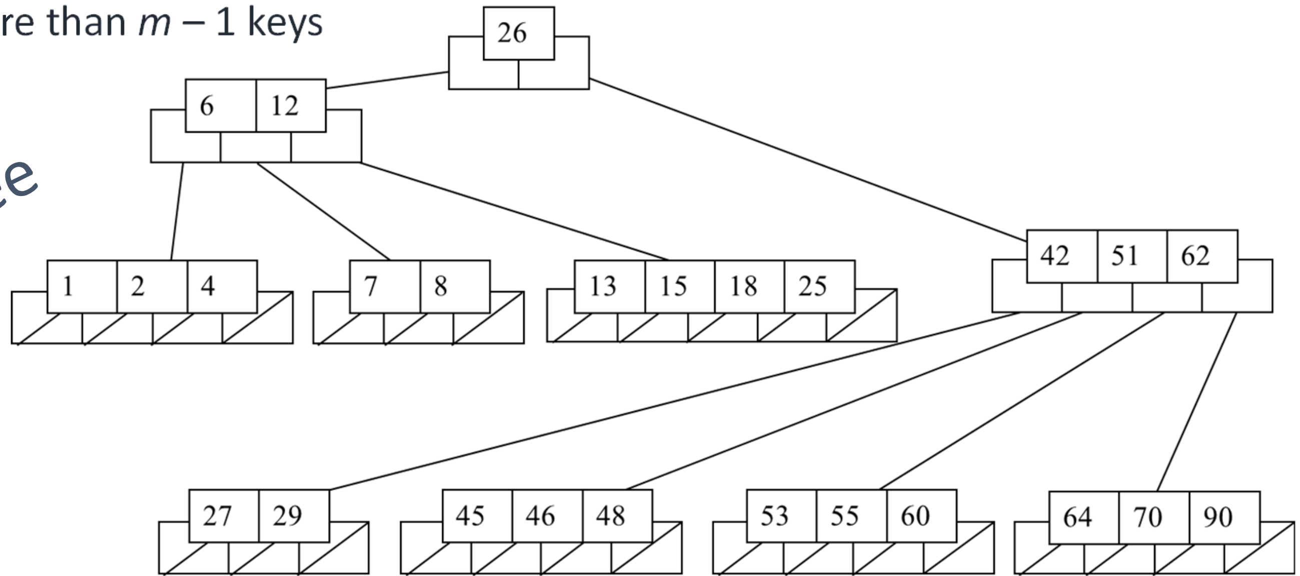
- Index structures for large datasets cannot be stored in main memory
- We end up with a very deep binary tree with lots of different disk accesses;
- But, the solution is to use more branches and thus reduce the height of the tree!
  - As branching increases, depth decreases



Definition

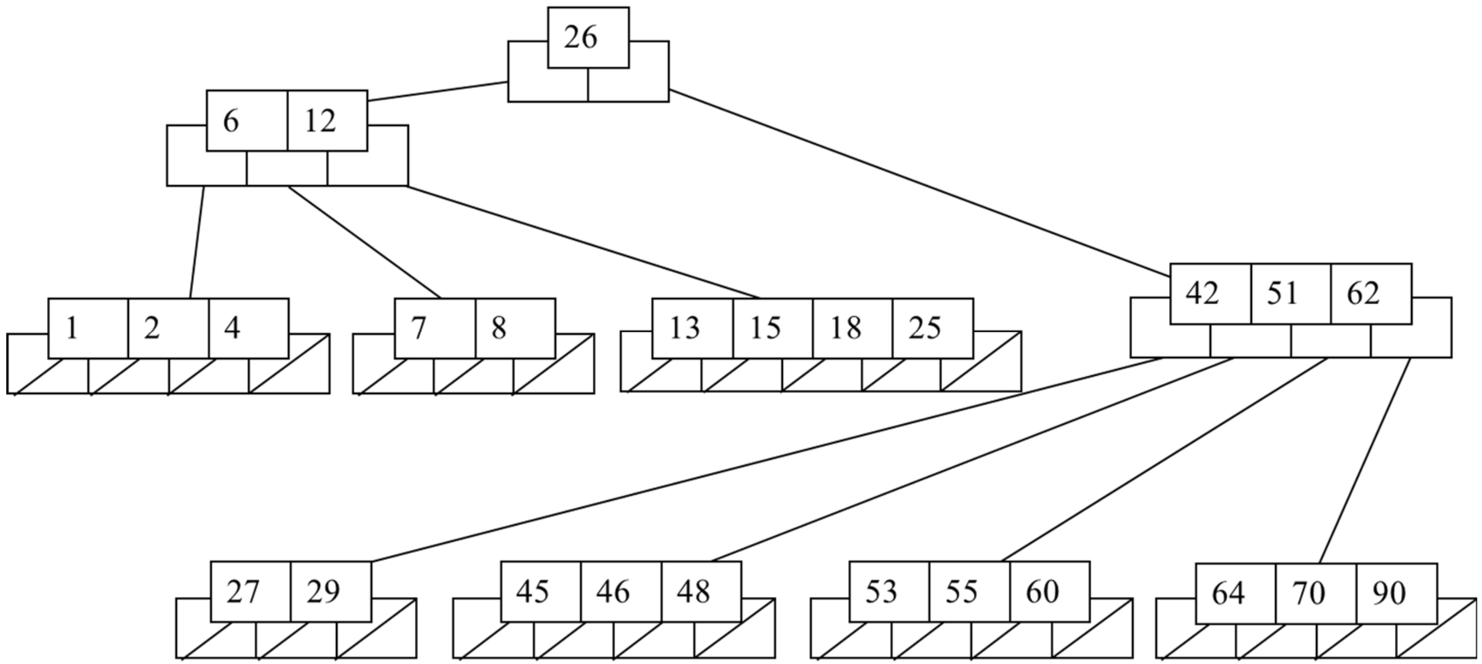
- A B-tree of order *m* is an *m*-way tree (i.e., a tree where each node may have up to *m* children) in which:
  1. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
  2. all leaves are on the same level
  3. all non-leaf nodes except the root have at least $\left\lceil \frac{m-1}{2} \right\rceil$-1 keys.
  4. All keys of a node are sorted in increasing order.
  5. Every node contains no more than $m - 1$ keys

5-way tree



Searching

- Search will start with the root.
- Check in which range the key is.
- Example: Finding 60.

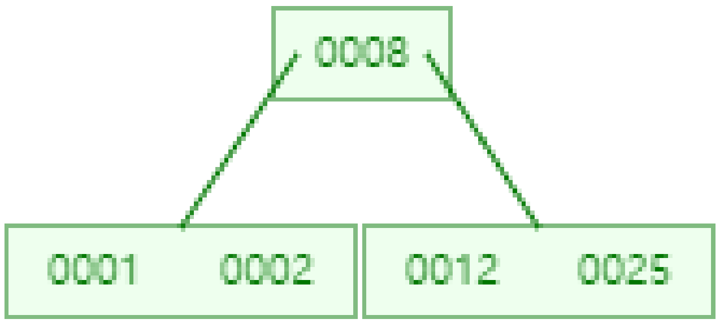


Constructing a B-tree

- 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

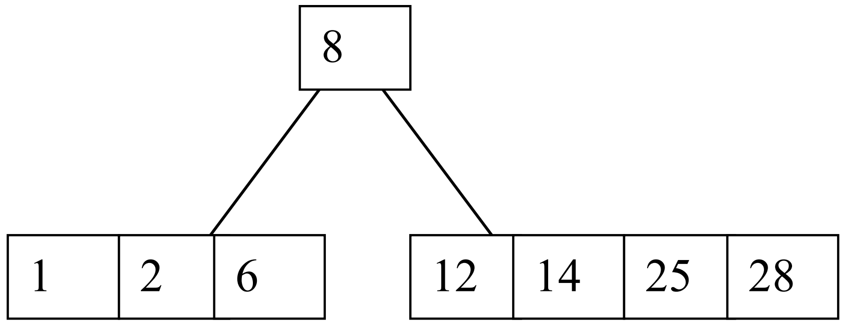
- To put the fifth item in the root would violate condition 5
- Therefore, when 25 arrives, pick the middle key to make a new root



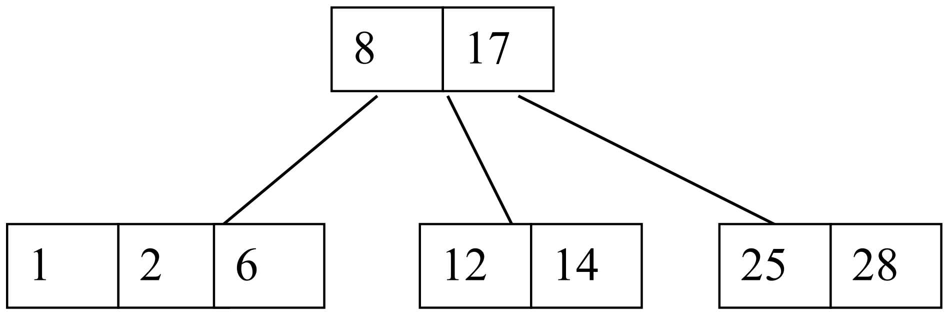
Constructing a B-tree (contd.)

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

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



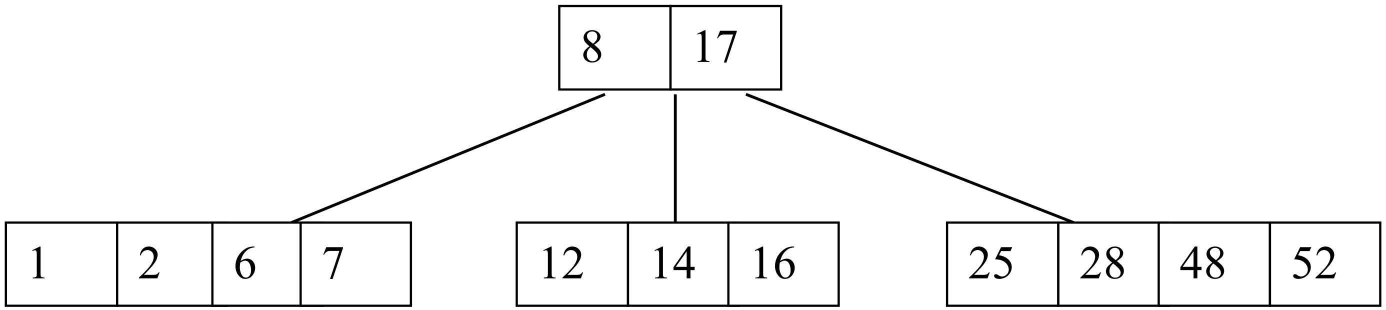
Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf



Constructing a B-tree (contd.)

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

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

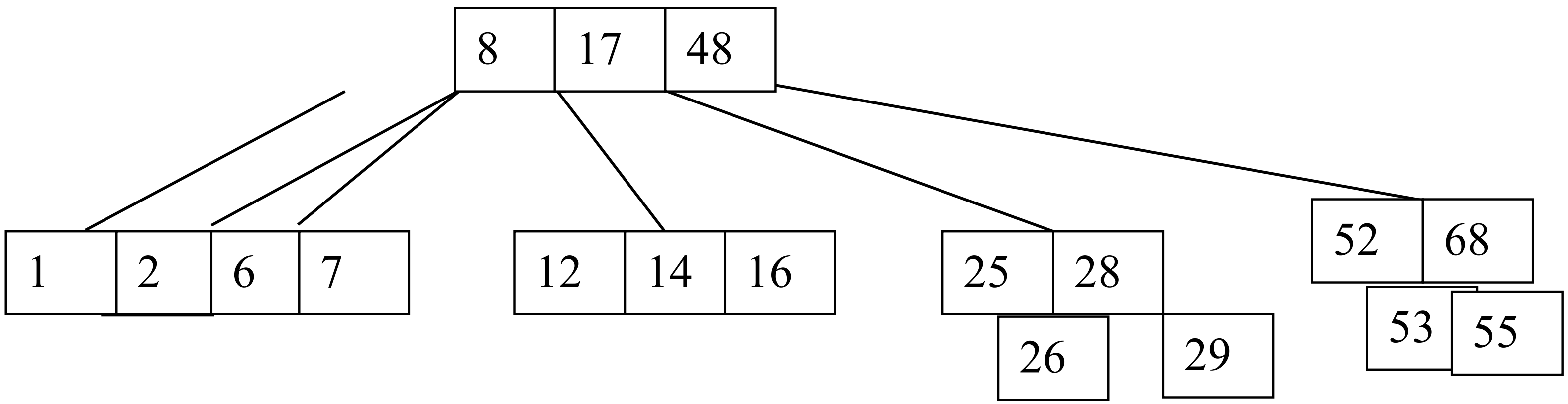


Adding 68 causes us to split the right most leaf, promoting 48 to the root

Constructing a B-tree (contd.)

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

adding 3 causes us to split the left most leaf, promoting 3 to the root;

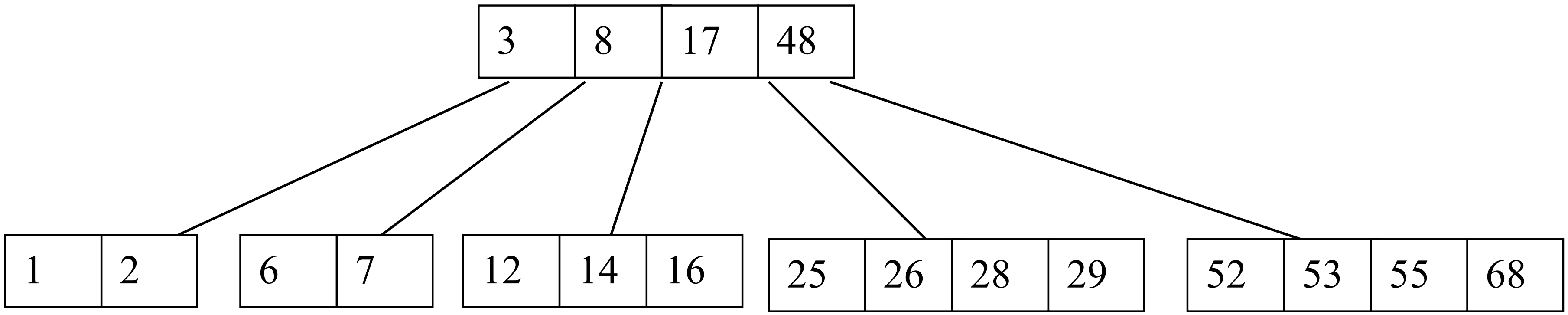


26, 29, 53, 55 then go into the leaves

Constructing a B-tree (contd.)

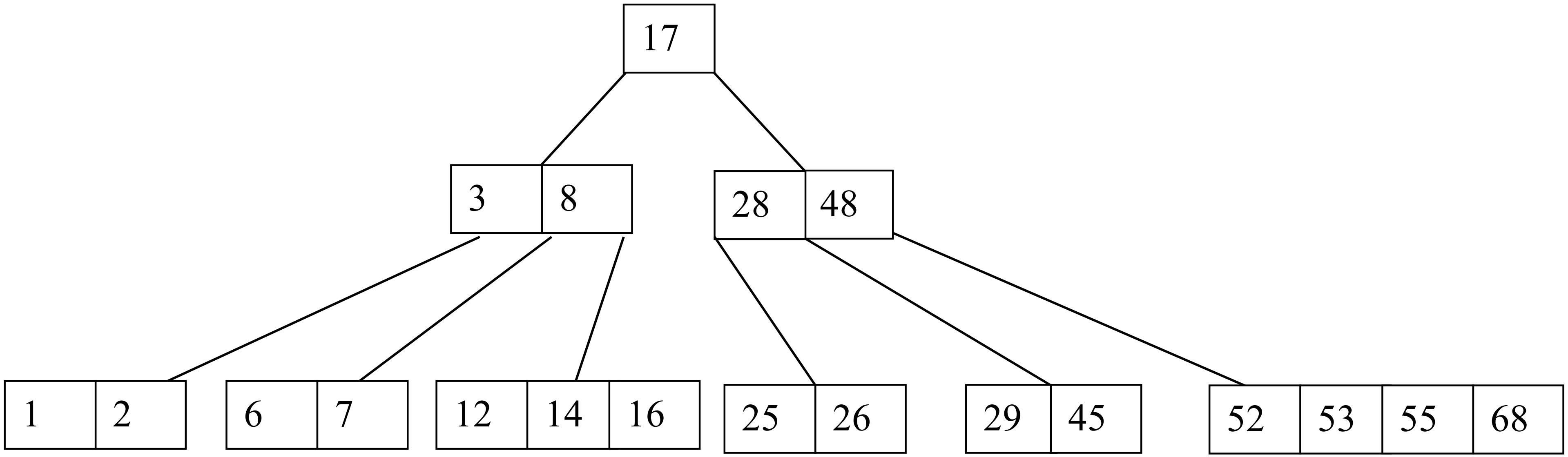
~~1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55~~ 45

Adding 45 causes a split of a leaf and promoting 28 to the root then causes the root to split



Final B-tree

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

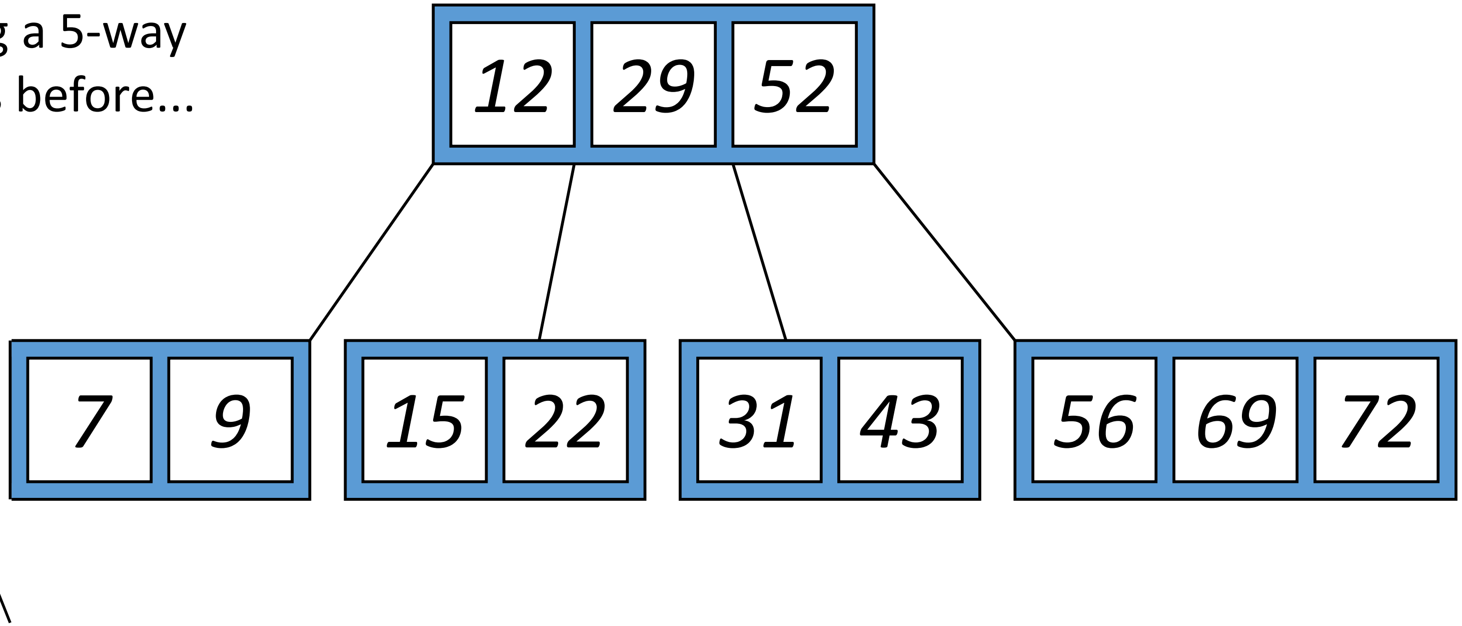


Inserting into a B-Tree

- 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

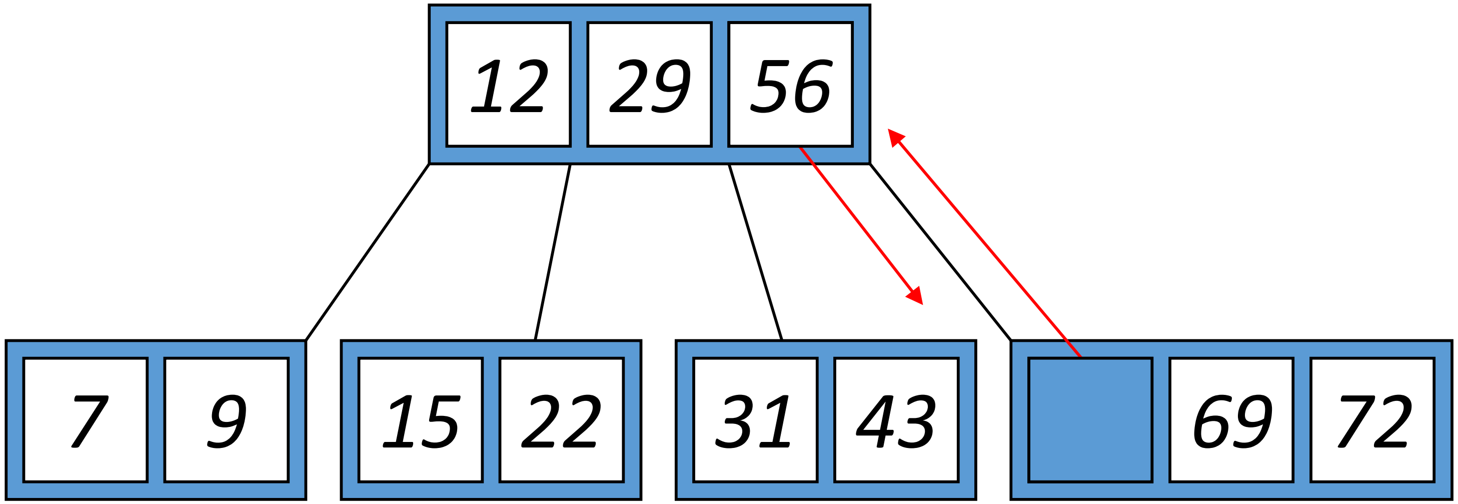
Case 1: Simple leaf deletion

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

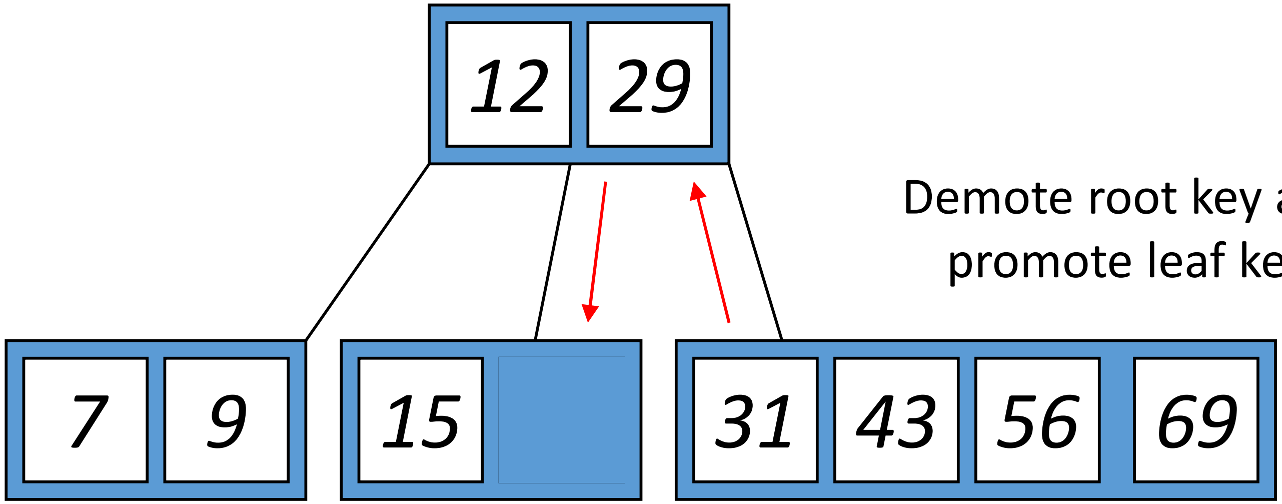


Delete 2: Since there are enough
keys in the node, just delete it

Case 2: Simple non-leaf deletion

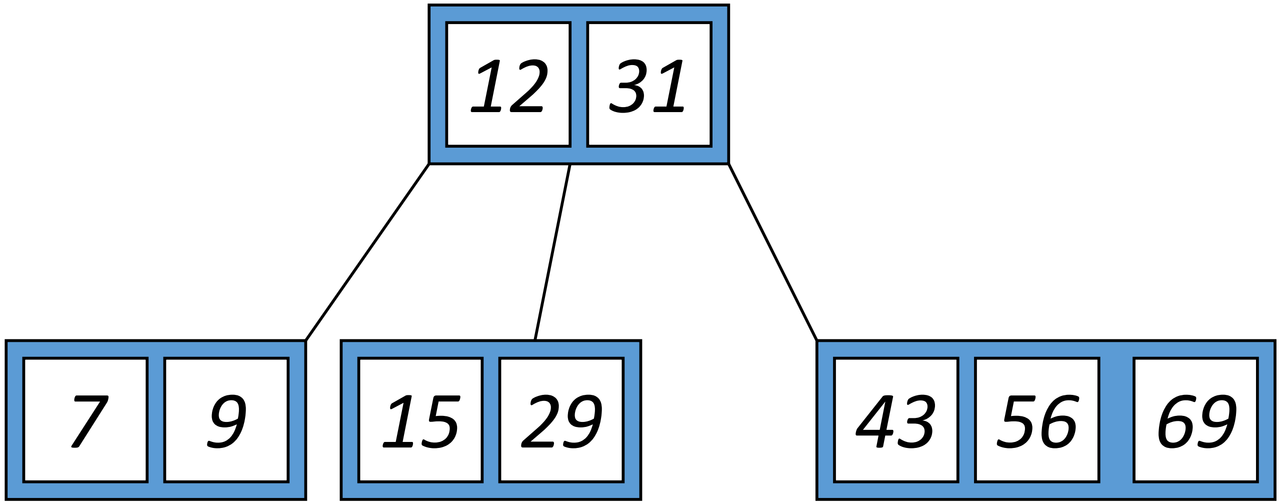


Case 3: Enough siblings

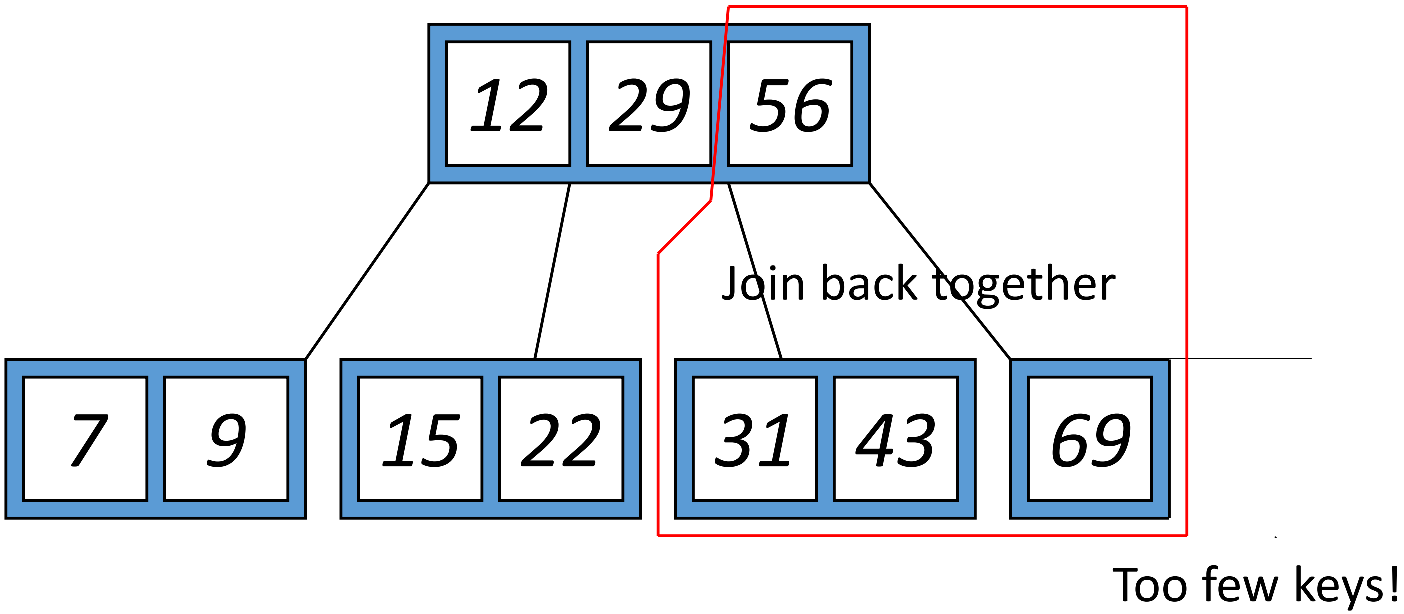


Demote root key and
promote leaf key

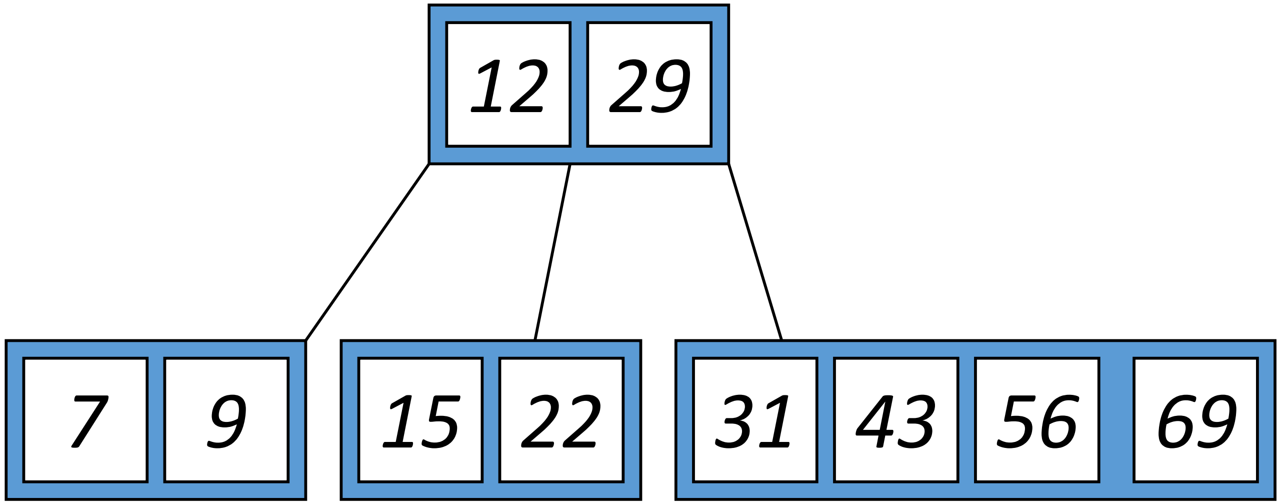
Case 3: Enough siblings



Case 4: Too few keys



Case 4: Too few keys



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:
- Case 1: the key is already in a leaf node
  - removing it
  - Case 1-1 leaf node to have too few keys
    - simply remove the key to be deleted.
- Case 2:the key is *not* in a leaf
  - it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf
  - 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:
  - Case 3: one of them has more than the min. number of keys
    - we can promote one of its keys to the parent and take the parent key into our lacking leaf
  - Case 4: neither of them has more than the min. number of keys
    - the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key)
    - 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

Analysis of B-Trees

- The maximum number of items in a B-tree of order m and height h:

root	$m-1$
level 1	$m(m-1)$
level 2	$m^2(m-1)$
. . .	
level h	$m^h(m-1)$

- So, the total number of items is

$$(1 + m + m^2 + m^3 + \ldots + m^h)(m-1) =$$

$$[(m^{h+1}-1)/ (m-1)]\ (m-1) = \boldsymbol{m^{h+1}-1}$$

- When $m = 5$ and $h = 2$ this gives $5^3 - 1 = 124$

Reasons for using B-Trees

- When searching tables held on disc, the cost of each disc transfer is high but doesn't depend much on the amount of data transferred, especially if consecutive items are transferred
  - If we use a B-tree of order 101, say, we can transfer each node in one disc read operation
  - A B-tree of order 101 and height 3 can hold $101^4 - 1$ items (approximately 100 million) and any item can be accessed with 3 disc reads (assuming we hold the root in memory)
- If we take $m = 3$, we get a **2-3 tree**, in which non-leaf nodes have two or three children (i.e., one or two keys)
  - B-Trees are always balanced (since the leaves are all at the same level), so 2-3 trees make a good type of balanced tree

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 (odd) 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

Thanks