BBM371- Data Management

Lecture 8 - Tree Based Indexing

(B-Trees)

29.11.2018

Hash based indexing vs Tree based indexing

- Remember that hash based indexing is used for equality search and does not work for range search.

- Tree based indexing is used for range search (and also for equality search).

```
Select * from Ogrenci where Ogrenci.Age>40 and Ogrenci.Age<60
```

B-Tree

- **Rudolf Bayer** and **Ed McCreight** invented the B-tree while working at Boeing Research Labs in 1971 (Bayer & McCreight 1972), but they did not explain what, if anything, the B stands for. Douglas Comer explains:
  - The origin of "B-tree" has never been explained by the authors. As we shall see, "balanced," "broad," or "bushy" might apply. However, it seems appropriate to think of B-trees as "Bayer"-trees. (Comer 1979, p. 123)
    - Comer, Douglas (June 1979), "The Ubiquitous B-Tree", Computing Surveys 11 (2): 123–137, doi:10.1145/356770.356776, ISSN 0360-0300.

B-Tree

- B-tree is a specialized multiway tree designed especially for use on the disk

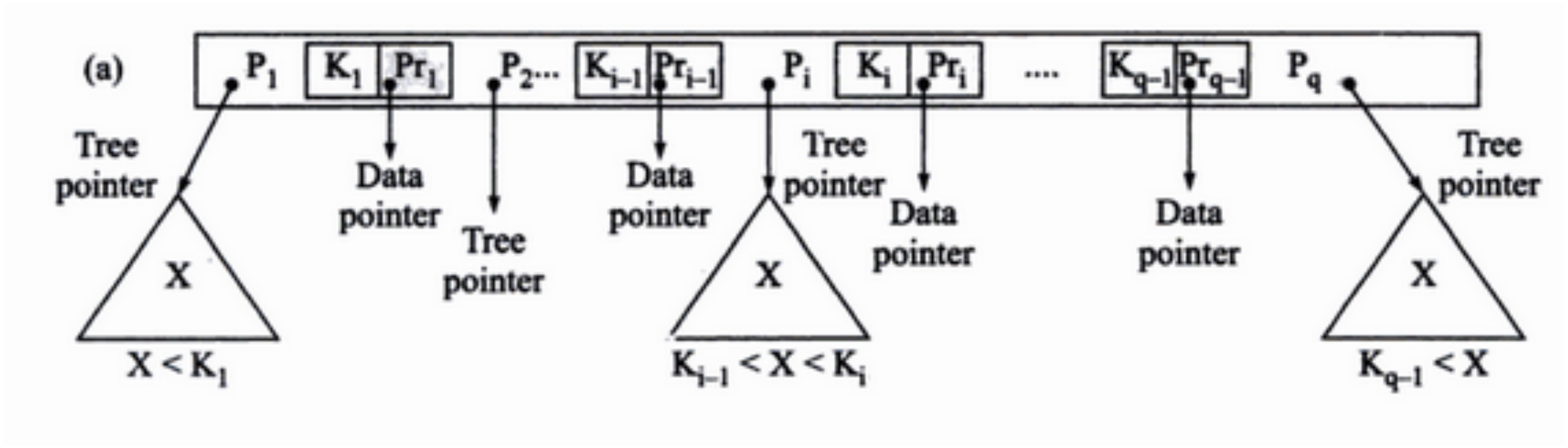
- B-Tree consists of a root node, intermediate nodes and leaf nodes containing the indexed field values in the leaf nodes of the tree.

B-Tree Characteristics

- In a B-tree each node may contain a large number of keys
- B-tree is designed to branch out in a large number of directions and to contain a lot of keys in each node so that the height of the tree is relatively small
- Tree is always balanced.
- Space wasted by deletion, if any, never becomes excessive

Node Structure in a B-Tree

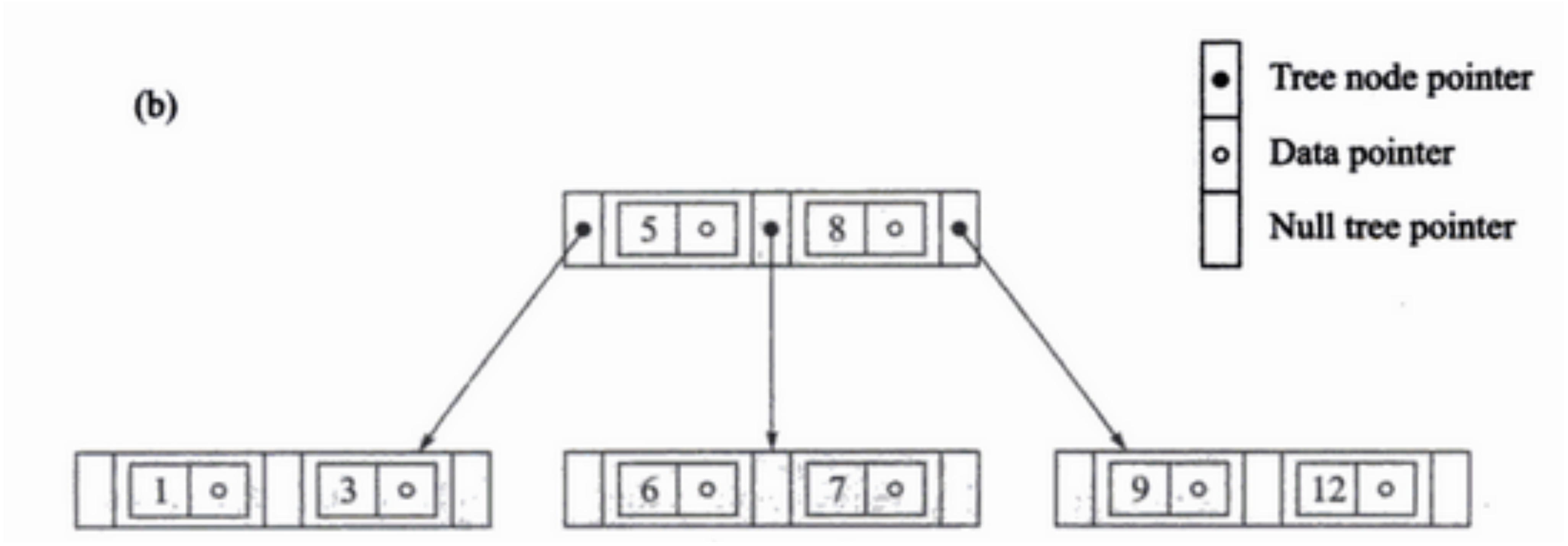
A node in a B-tree:



Each P is a tree pointer and each Pr is a data pointer.

Node Structure in a B-Tree

A B-Tree of min. order d=3:

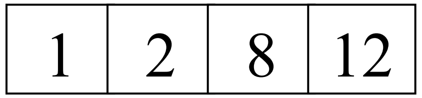


Aspects of B Tree

- Specs of min ‘d+1’ order B Tree:
  1. Internal nodes (except for root) has at least d, at most 2d keys
  2. Root (if it is not a leaf) has at least 2 children
  3. All leaf nodes are in the same level (balanced tree)
  4. Level increase and decreases are handled towards up (not down)
  5. An internal node has at least d keys and d+1 children

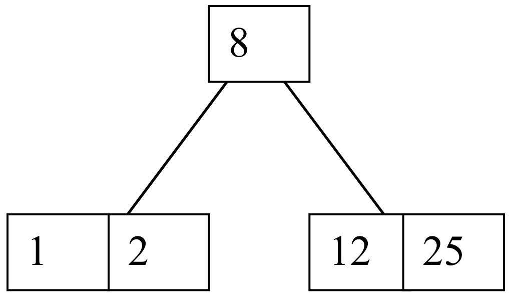
Constructing a B-tree

- Suppose we start with an empty B-tree and keys arrive in the following order:
  1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45
- We want to construct a B-tree of max order 5 (d=2).
- The first four items go into the root node:

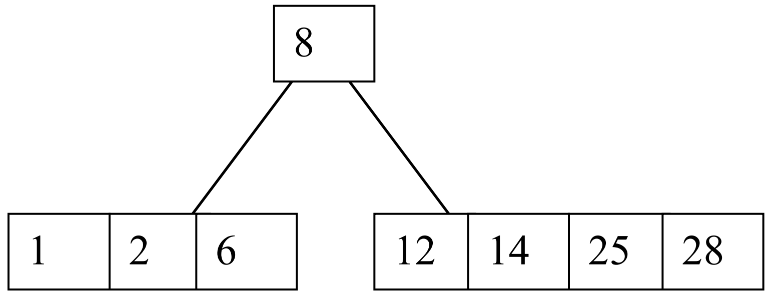


- Insertion of the fifth item in the root node violates condition 5
- Therefore, when 25 arrives, pick the middle key to make a new root

Constructing a B-tree (contd.)

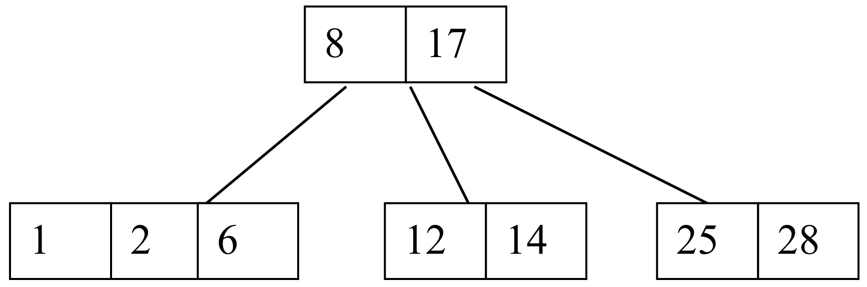


6, 14, 28 get added to the leaf nodes:

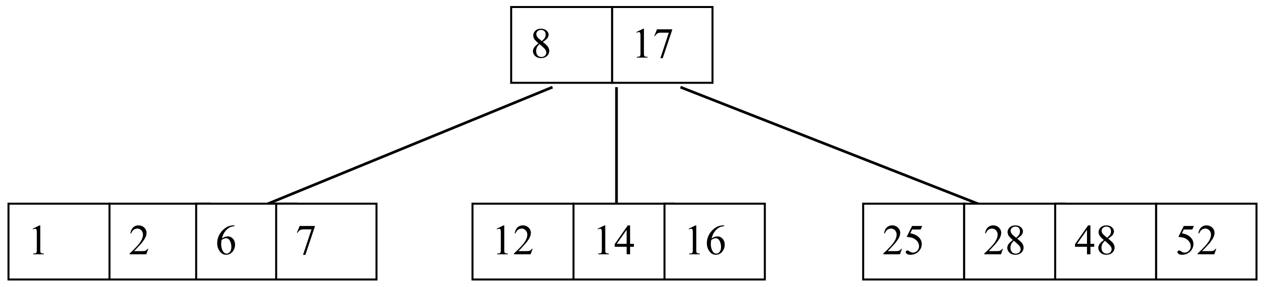


Constructing a B-tree (contd.)

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

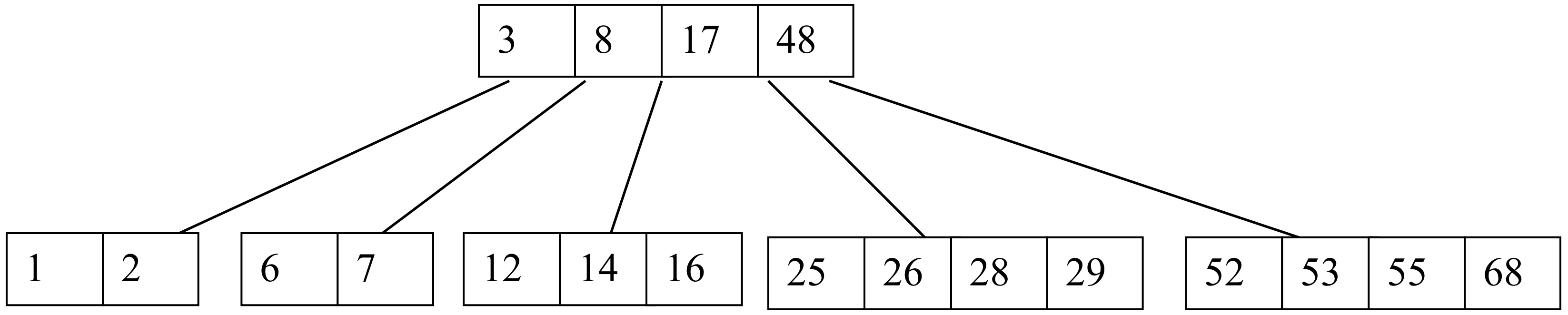


7, 52, 16, 48 get added to the leaf nodes

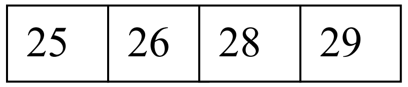


Constructing a B-tree (contd.)

Adding 68 causes us to split the rightmost 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

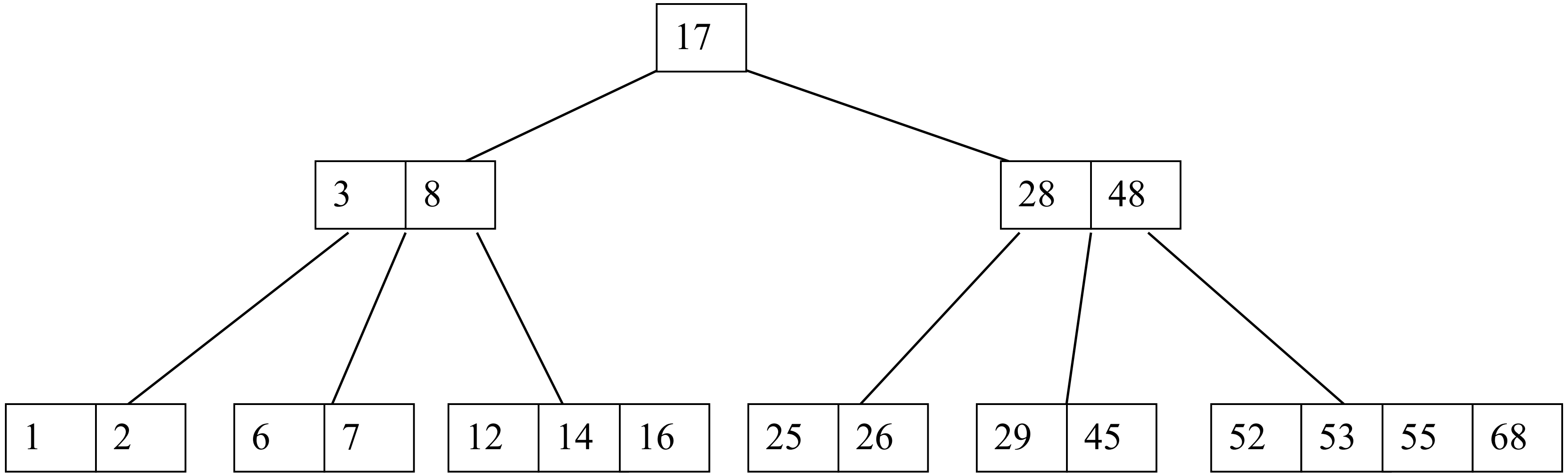


Adding 45 causes a split of



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

Constructing a B-tree (contd.)



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

Structure of a B-Tree Node

- **Bucket Factor**: number of records inserted into one node
- **Fan-out :** number of children sourced from one node
    - High fan-out makes the tree bushy and therefore leads to low height.
    - A high fan-out makes the tree more efficient.
- The size of each node is equal to the size of the page/block

Exercise in Inserting a B-Tree

- 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 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:

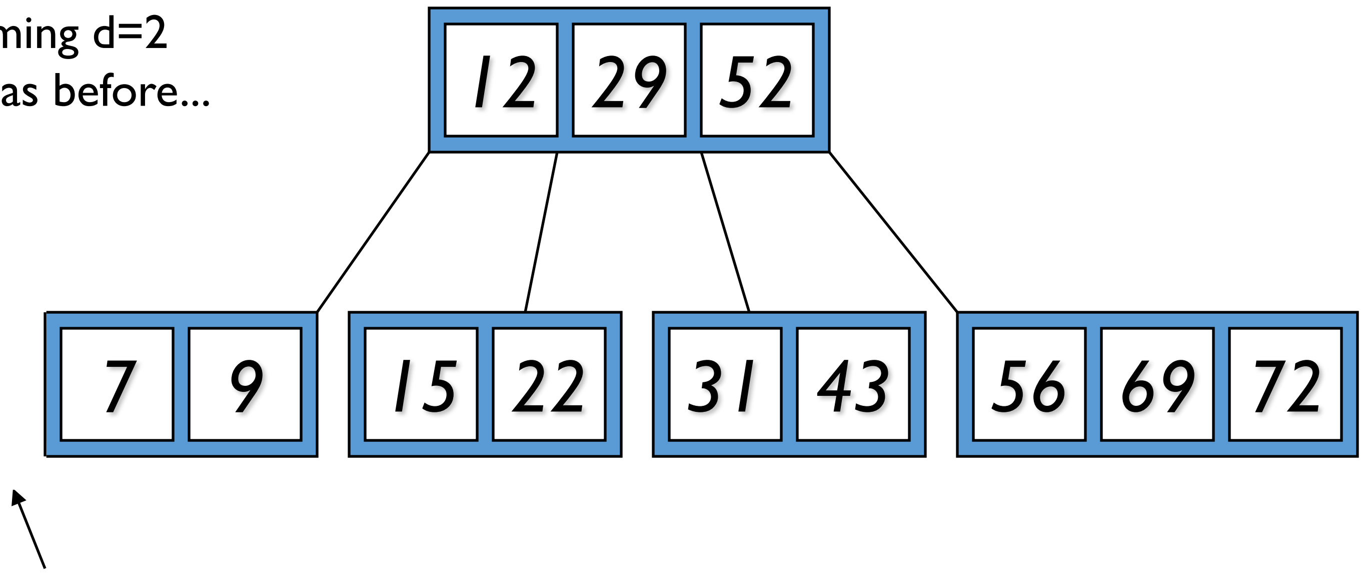
1. If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted.
2. If the key is *not* in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf -- in this case 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 child 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 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 d=2
B-Tree, as before...

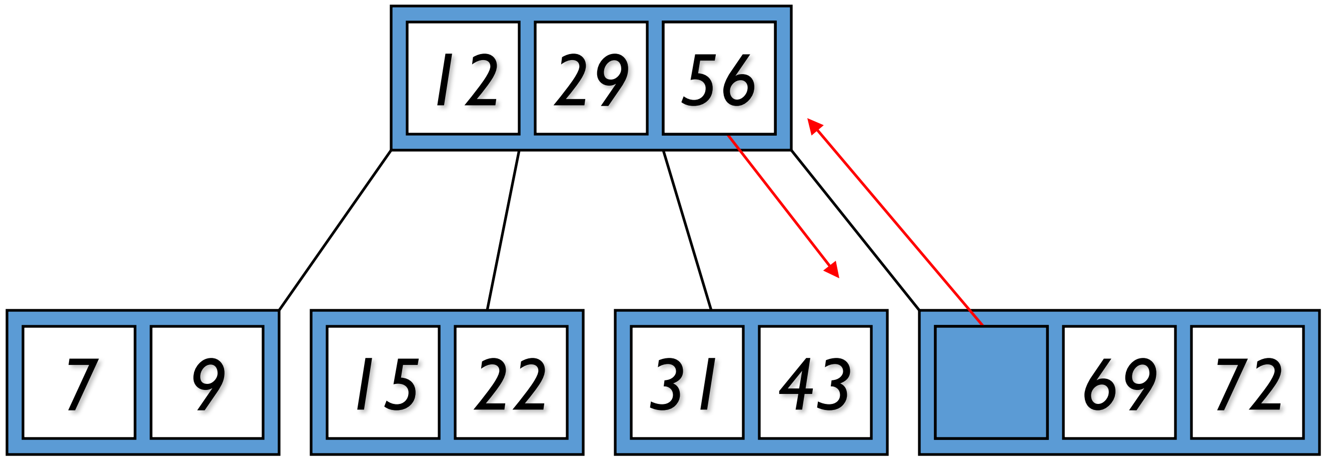


Delete 2: Since there are enough keys in the node, just delete it

Note when printed: this slide is animated

Type #2: Simple non-leaf deletion

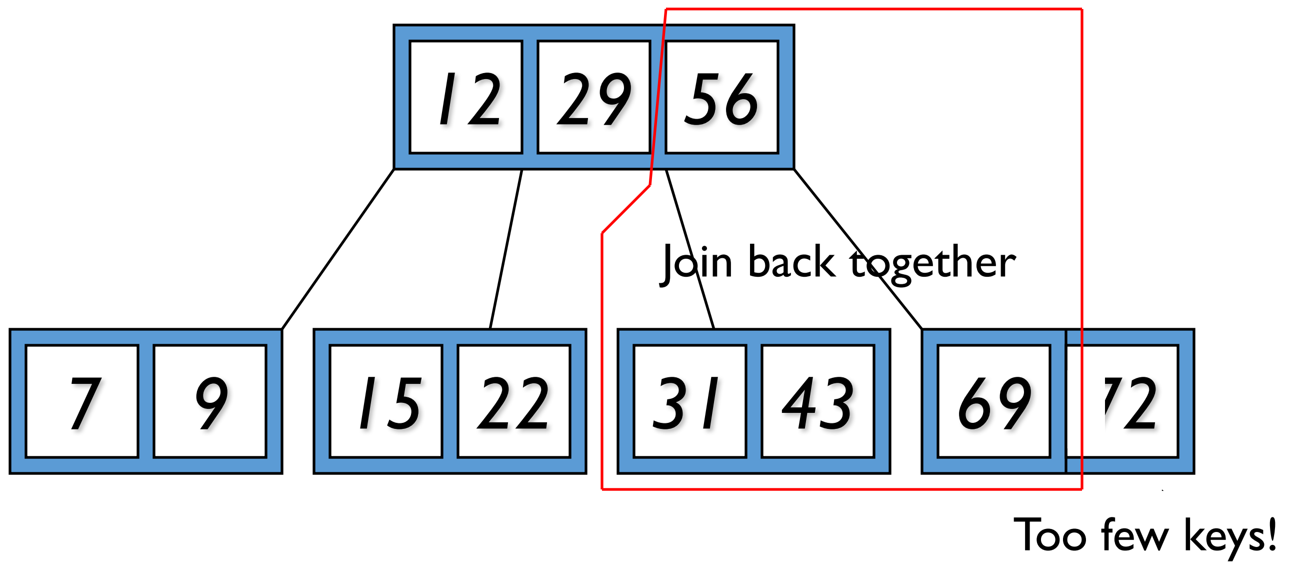
Delete 52:



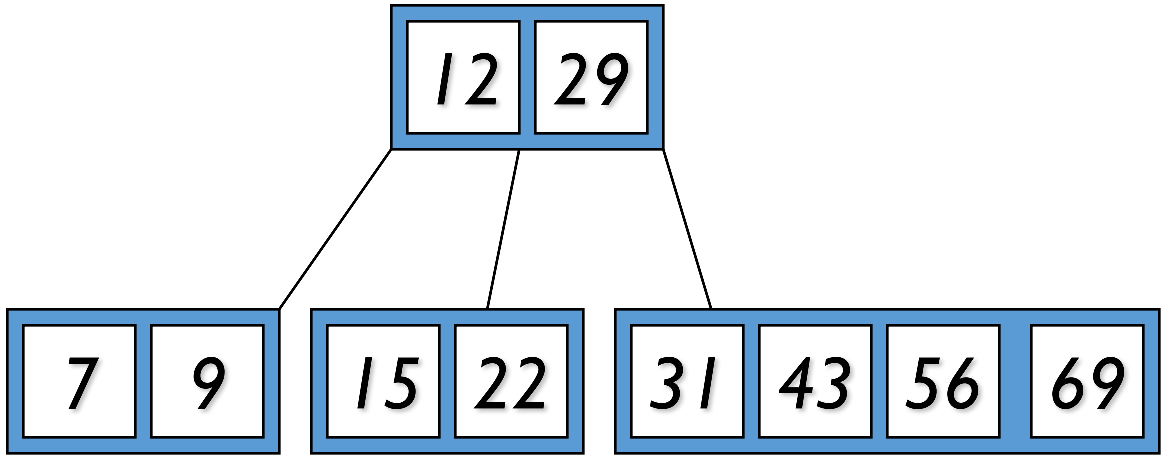
Note when printed: this slide is animated

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

Delete 72:

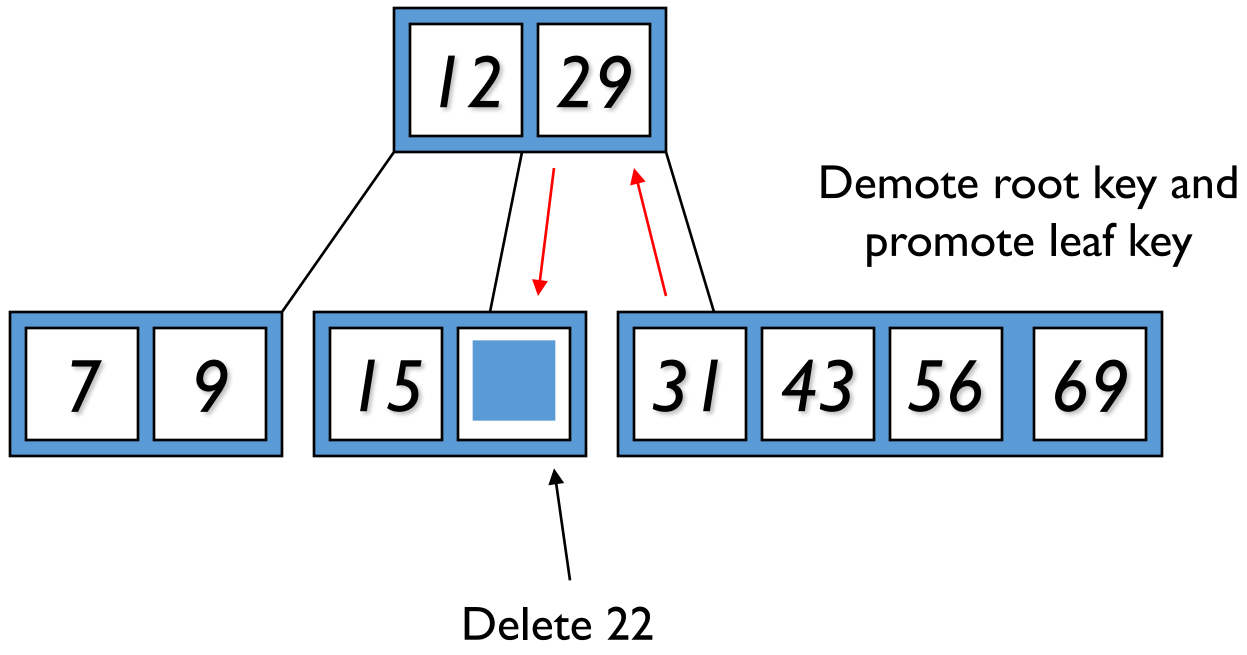


Type #4: Too few keys in node and its siblings



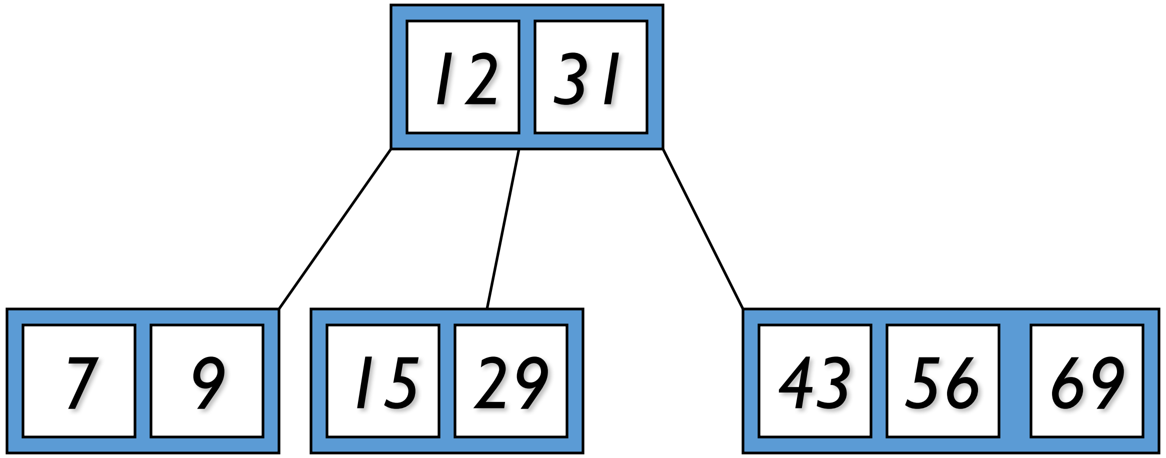
Note when printed: this slide is animated

Type #3: Enough siblings



Note when printed: this slide is animated

Type #3: Enough siblings



Note when printed: this slide is animated

Exercise for Removing from a B-Tree

- Given a max 5 order B-tree created by these data (last exercise):
- 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

Analysis of B-Trees

- The maximum number of items in a B-tree of max order m $(2d+1)$ 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

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

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

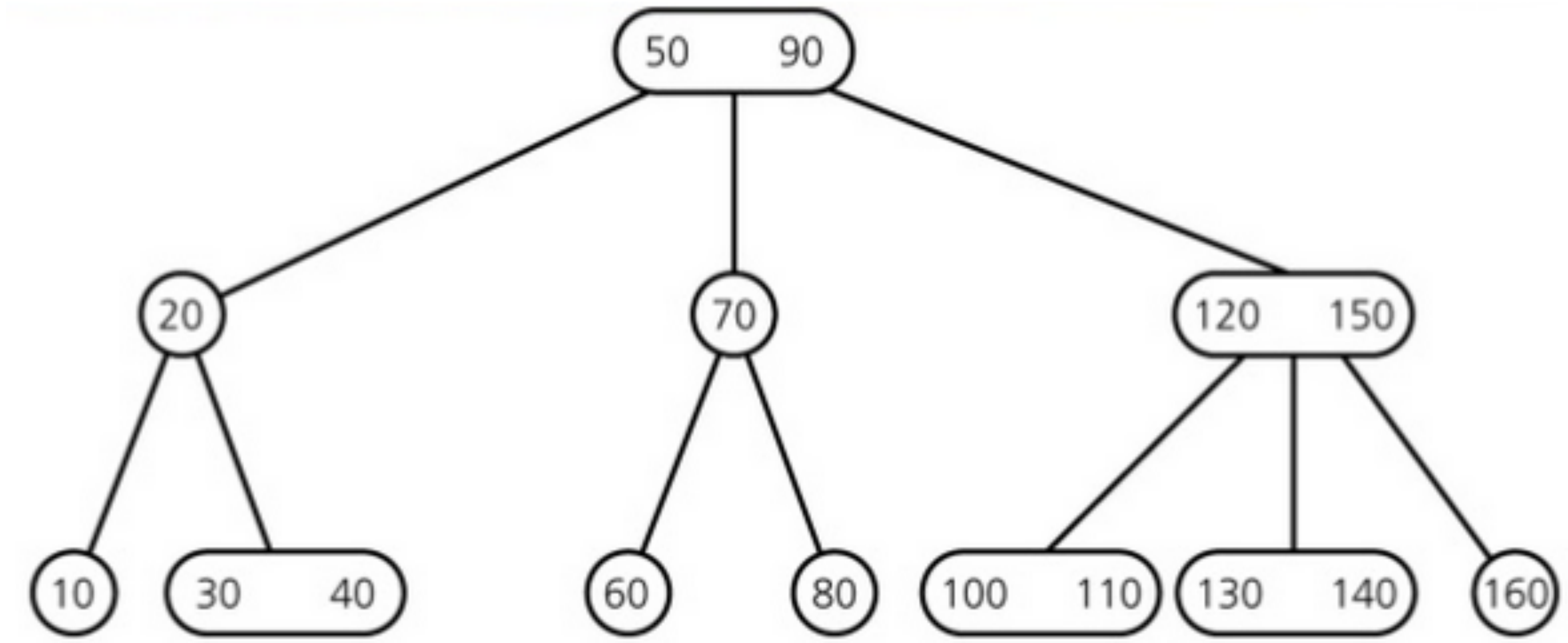
- When $m = 5$ and $h = 2$ this gives $5^3 - 1 = 124$

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 max 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)

2-3 Tree

- If we take *d=1*, 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


50 90
20
70
120 150
10
30 40
60
80
100 110
130 140
160