B-Trees

Motivation for B-Trees

- So far we have assumed that we can store an entire data structure in main memory
- What if we have so much data that it won't fit?
- We will have to use disk storage but when this happens our time complexity fails
- The problem is that Big-Oh analysis assumes that all operations take roughly equal time
- This is not the case when disk access is involved

Motivation (cont.)

- Assume that a disk spins at 3600 RPM
- In 1 minute it makes 3600 revolutions, hence one revolution occurs in 1/60 of a second, or 16.7ms
- On average what we want is half way round this disk – it will take 8ms
- This sounds good until you realize that we get 120 disk accesses a second – the same time as 25 million instructions
- In other words, one disk access takes about the same time as 200,000 instructions
- It is worth executing lots of instructions to avoid a disk access

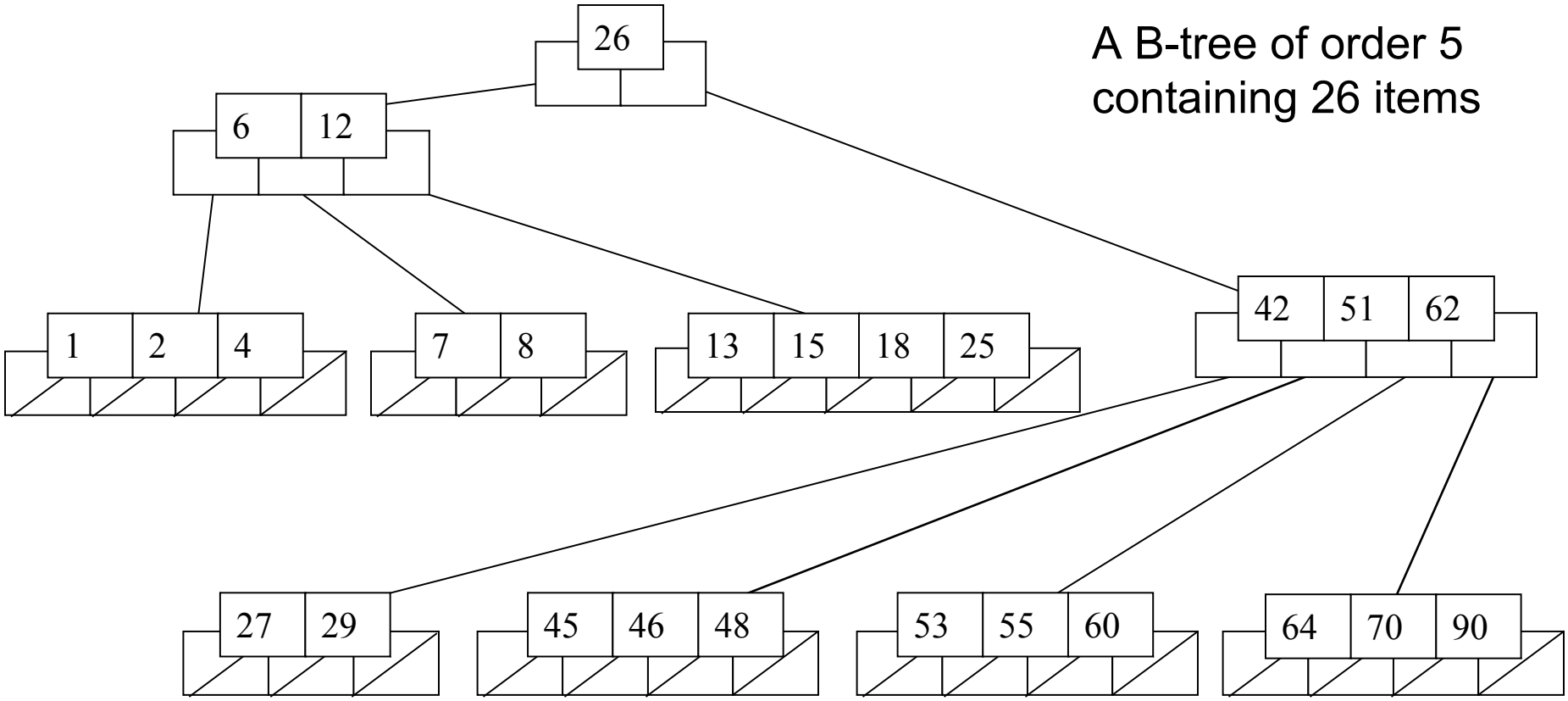
Motivation (cont.)

- Assume that we use an Binary tree to store all the details of people in Canada (about 32 million records)
- We still end up with a **very** deep tree with lots of different disk accesses; $\log_2 20{,}000{,}000$ is about 25, so this takes about 0.21 seconds (if there is only one user of the program)
- We know we can't improve on the log *n* for a binary tree
- But, the solution is to use more branches and thus less height!
- As branching increases, depth decreases

Definition of a B-tree

- 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
- The number m should always be odd

An example B-Tree

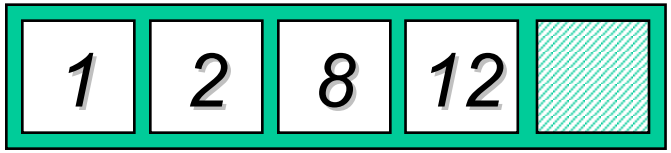


A B-tree of order 5
containing 26 items

Note that all the leaves are at the same level

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:

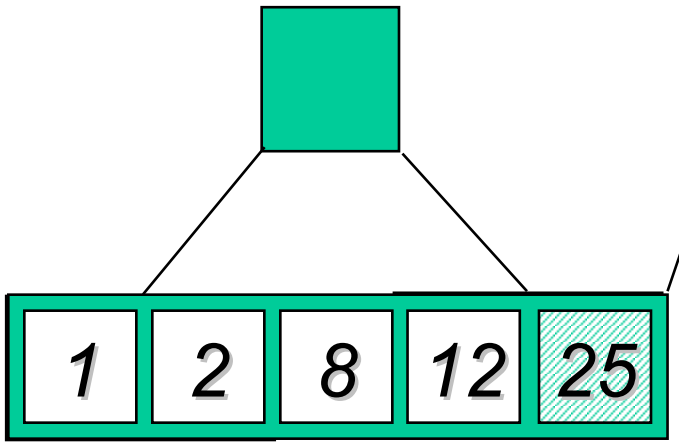


- To put the fifth item in the root would violate condition 5
- Therefore, when 25 arrives, pick the middle key to make a new root

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

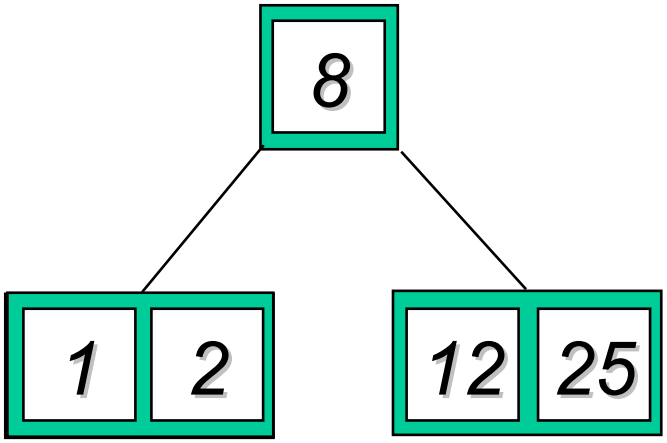
Constructing a B-tree

Add 25 to the tree

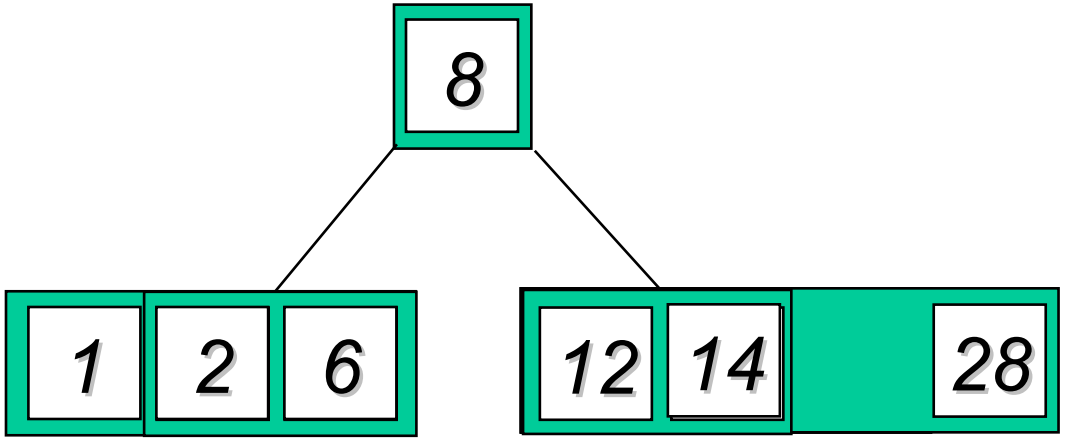


Exceeds Order.
Promote middle and split.

Constructing a B-tree (contd.)

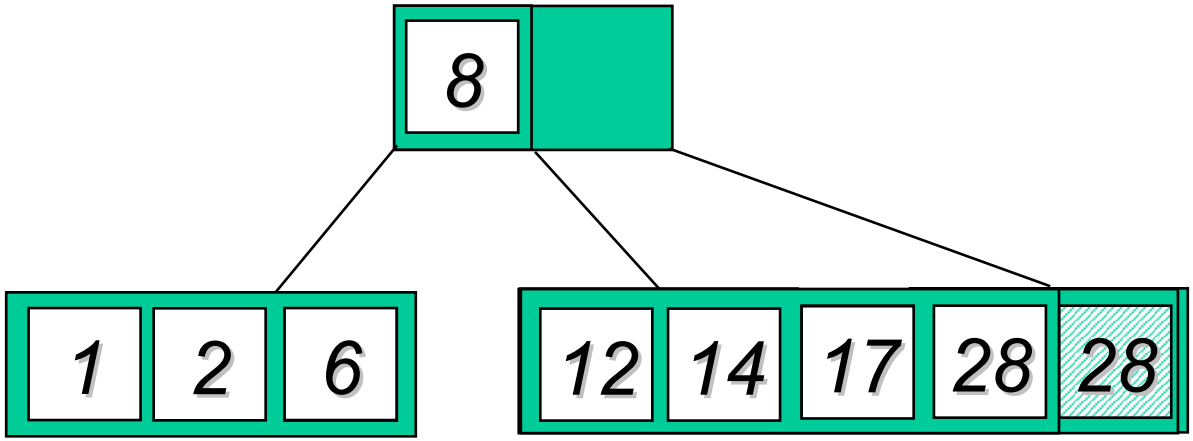


6, 14, 28 get added to the leaf nodes:



Constructing a B-tree (contd.)

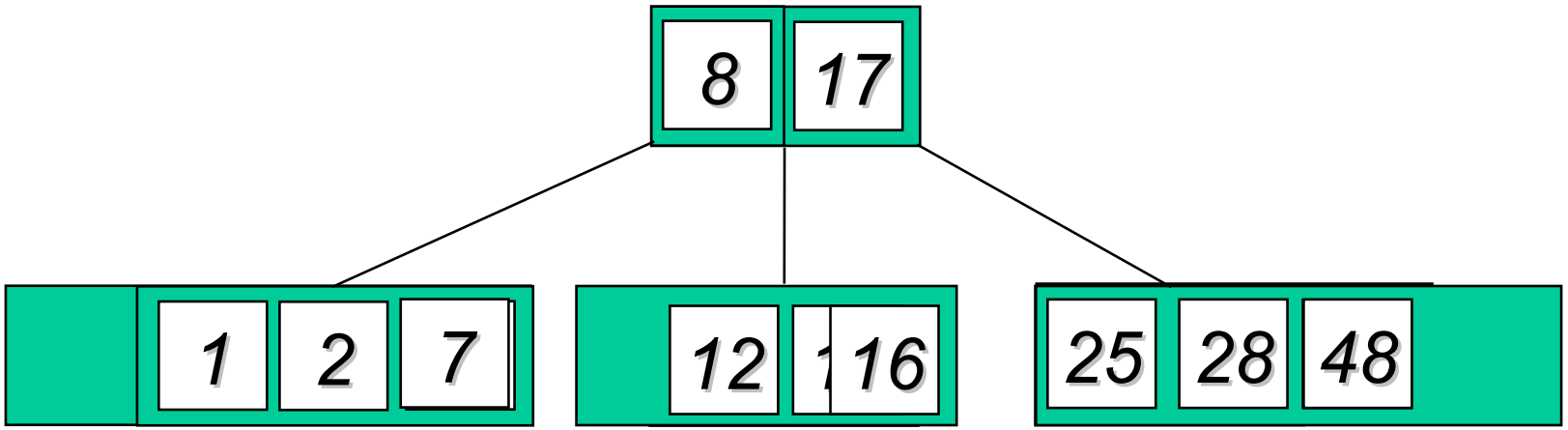
Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf



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

Constructing a B-tree (contd.)

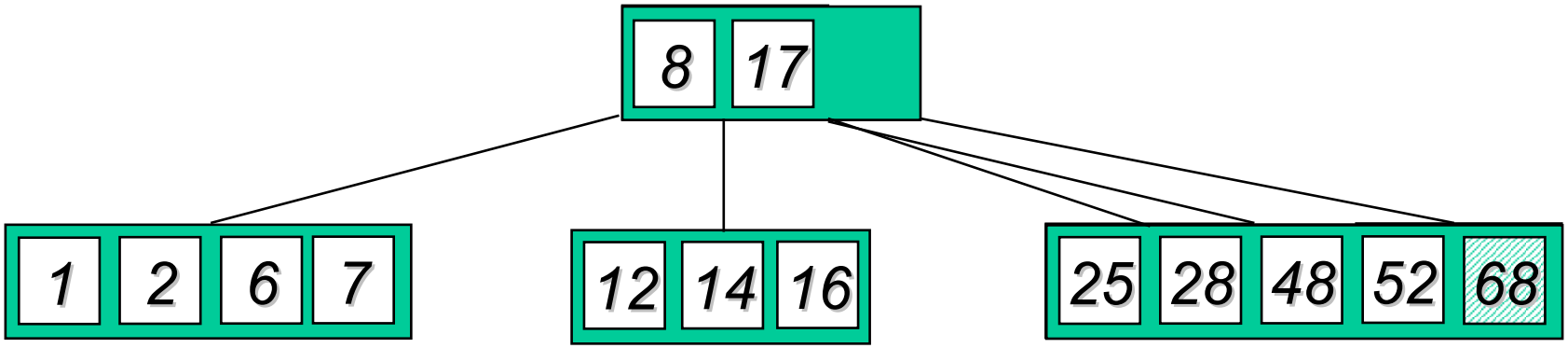
7, 52, 16, 48 get added to the leaf nodes



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

Constructing a B-tree (contd.)

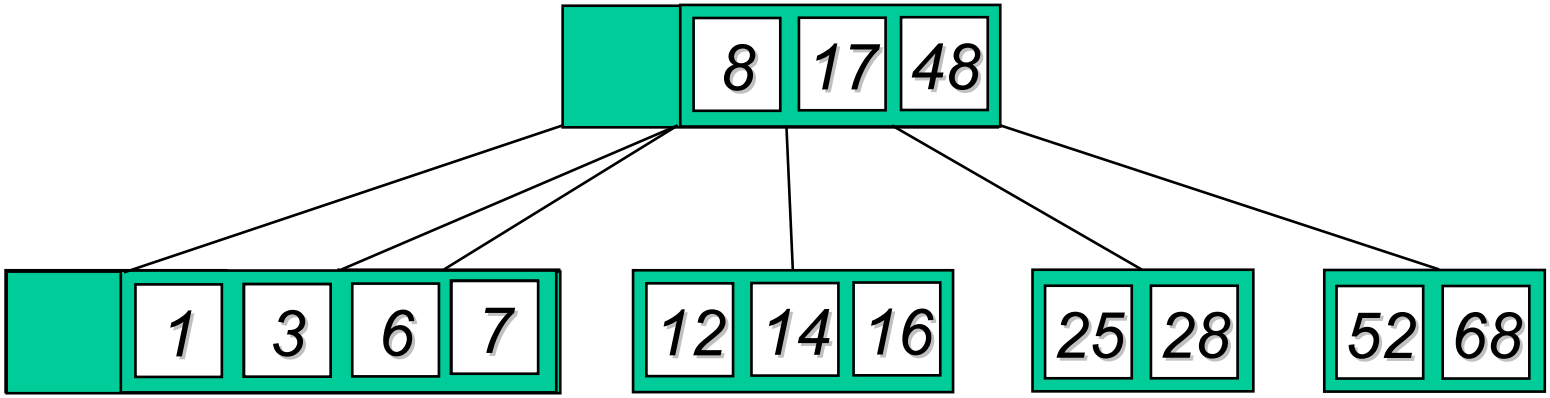
Adding 68 causes us to split the right most leaf, promoting 48 to the root



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

Constructing a B-tree (contd.)

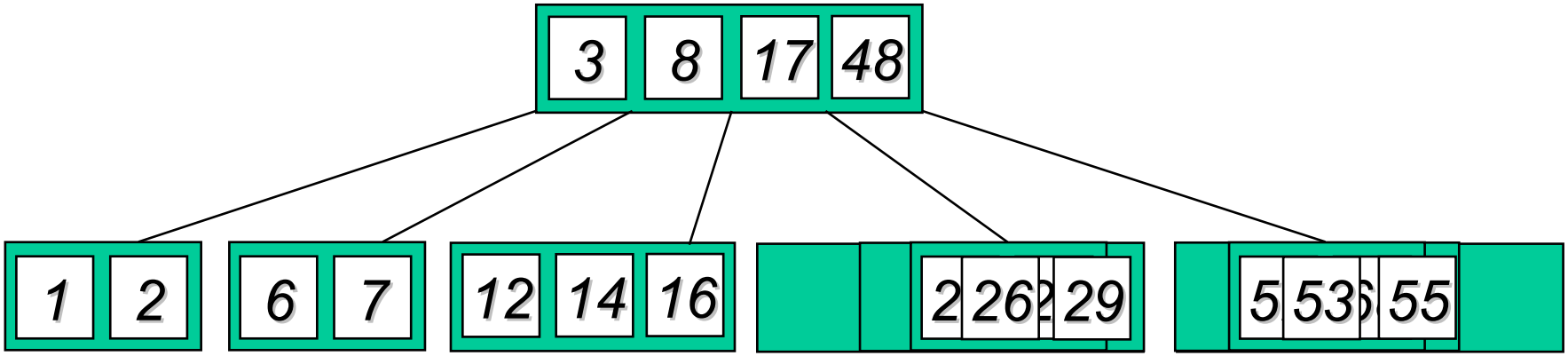
Adding 3 causes us to split the left most leaf



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

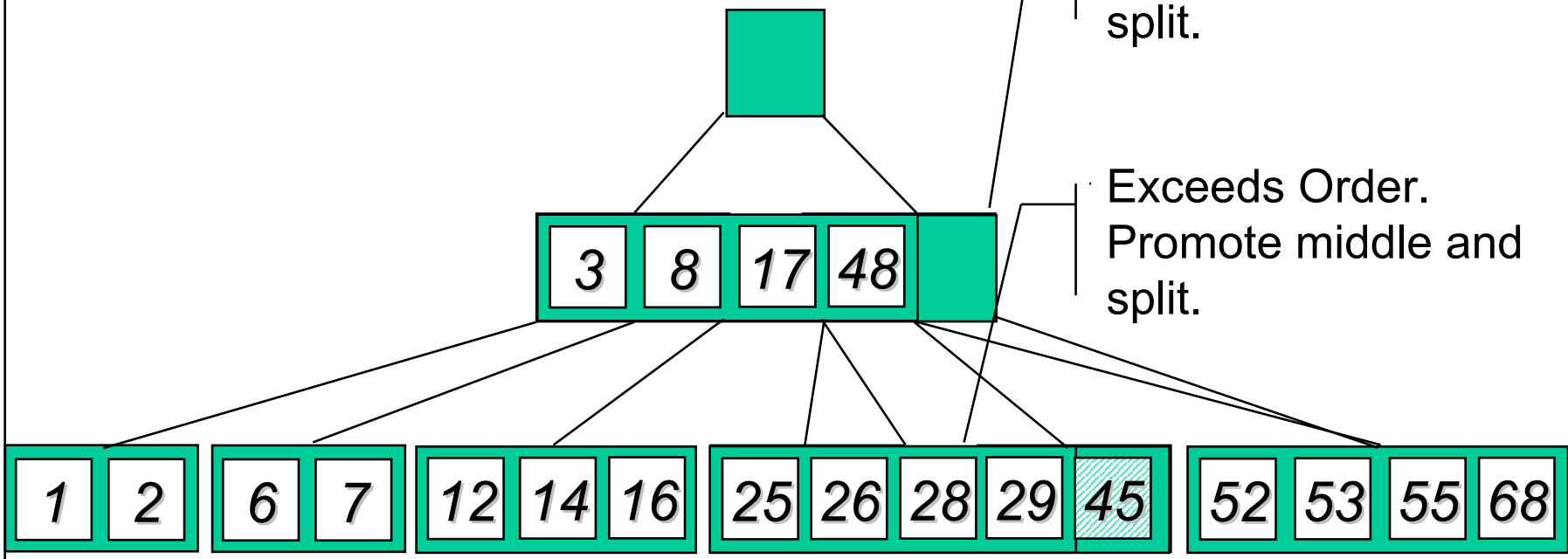
Constructing a B-tree (contd.)

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



Constructing a B-tree (contd.)

Add 45 increases the trees level



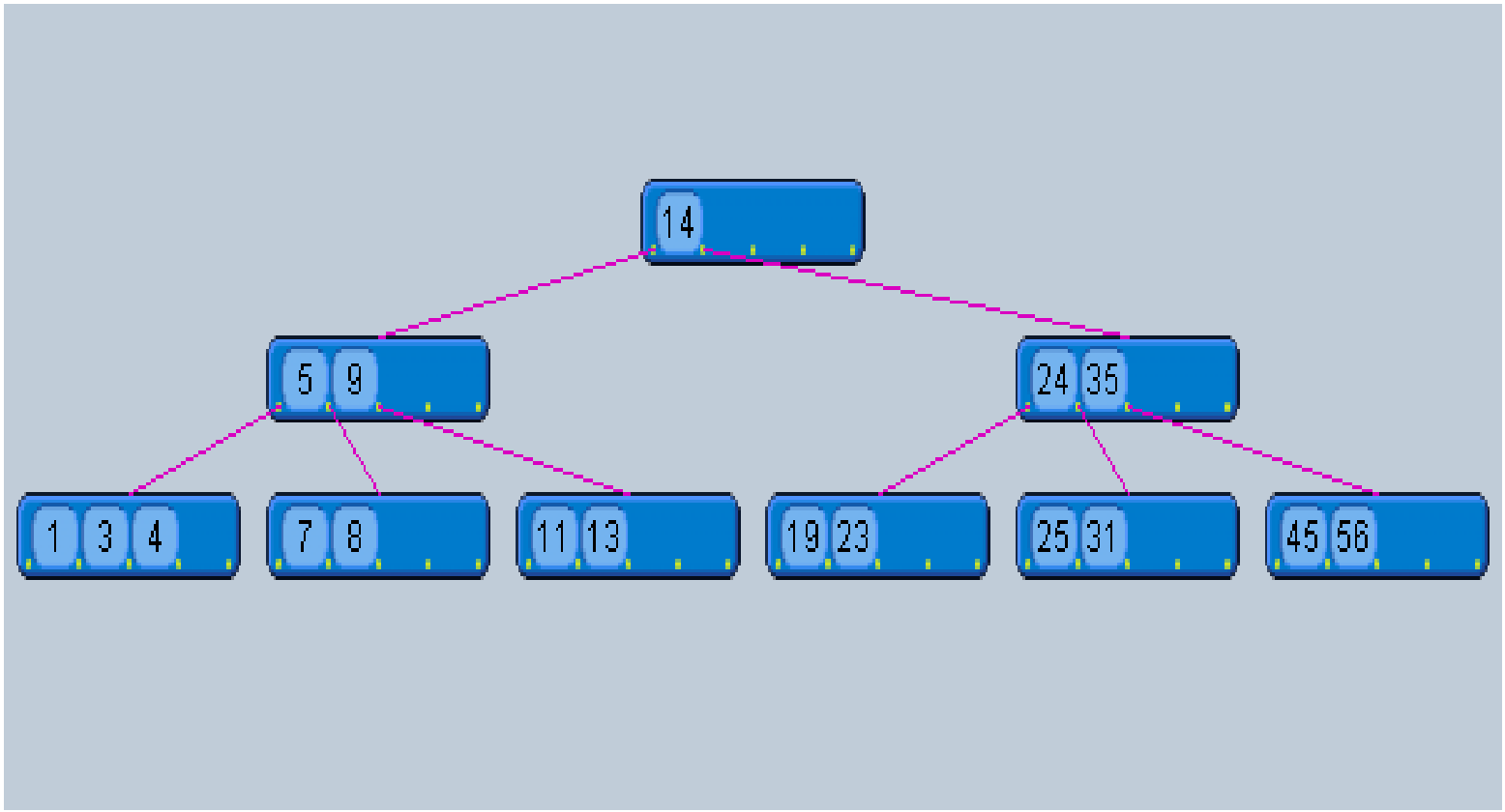
Inserting into a B-Tree

- 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

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

Answer to Exercise



Java Applet Source

Removal from a B-tree

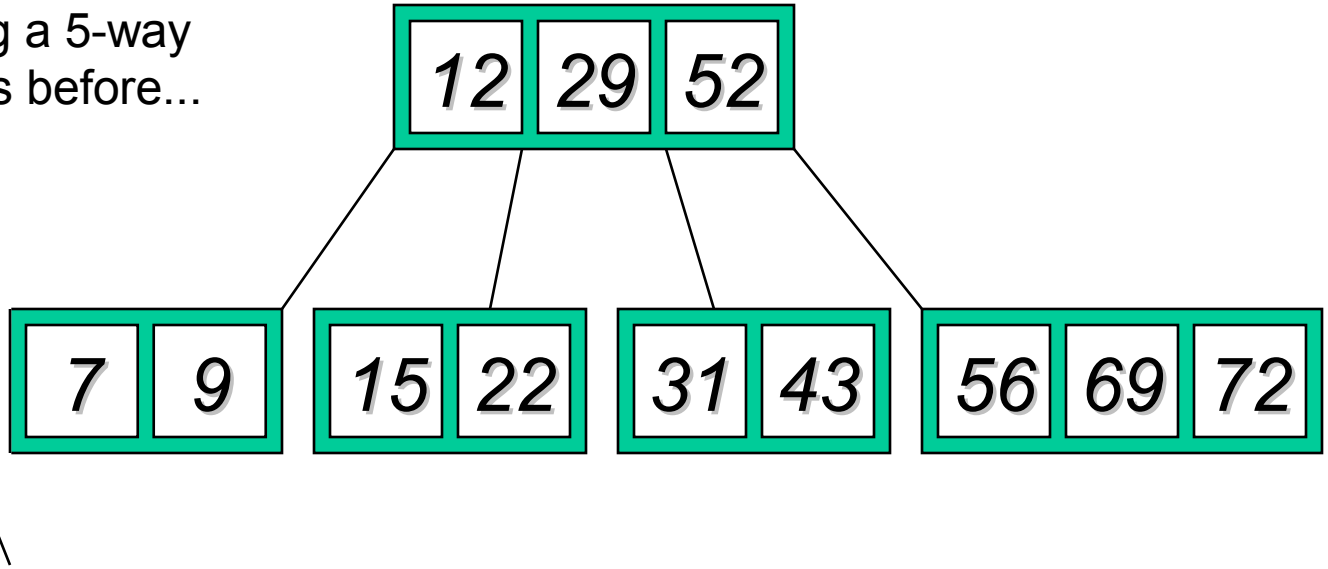
- During insertion, the key always goes *into* a *leaf*. For deletion we wish to remove *from* a leaf. There are three possible ways we can do this:
- 1 - If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted.
- 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 can we 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:
  - 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

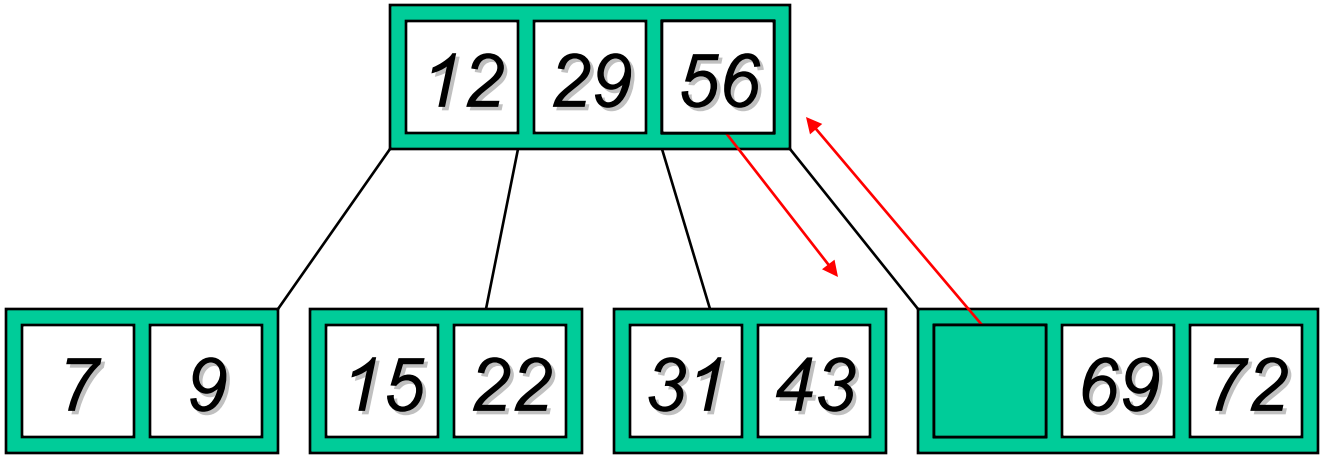
Assuming a 5-way B-Tree, as before...



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

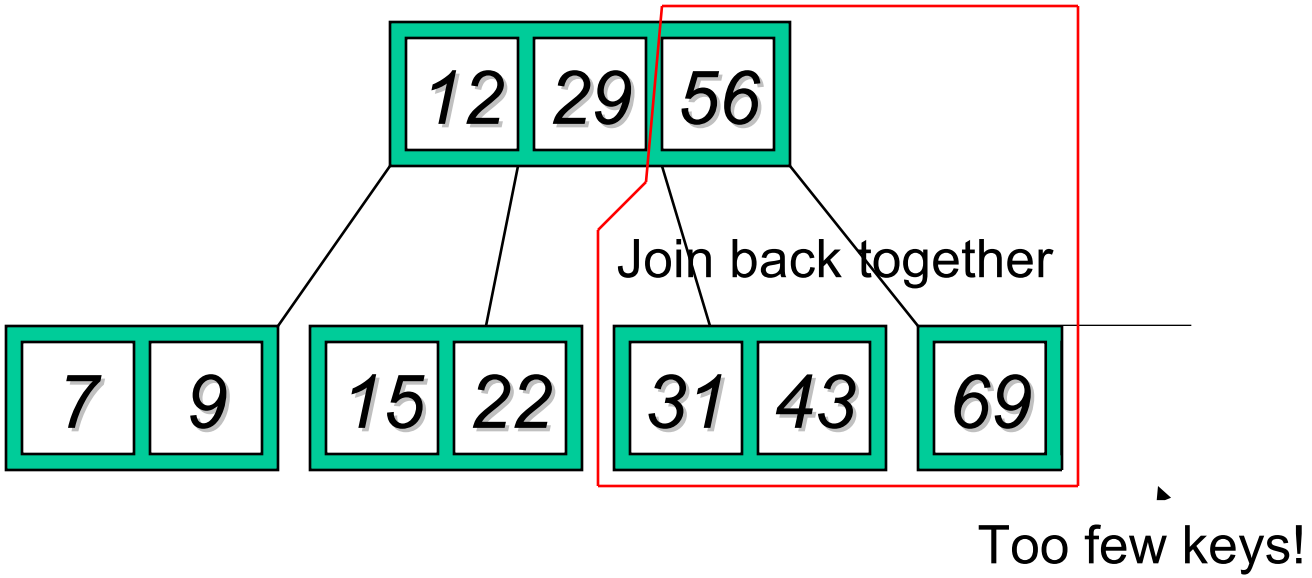
Note when printed: this slide is animated

Type #2: Simple non-leaf deletion



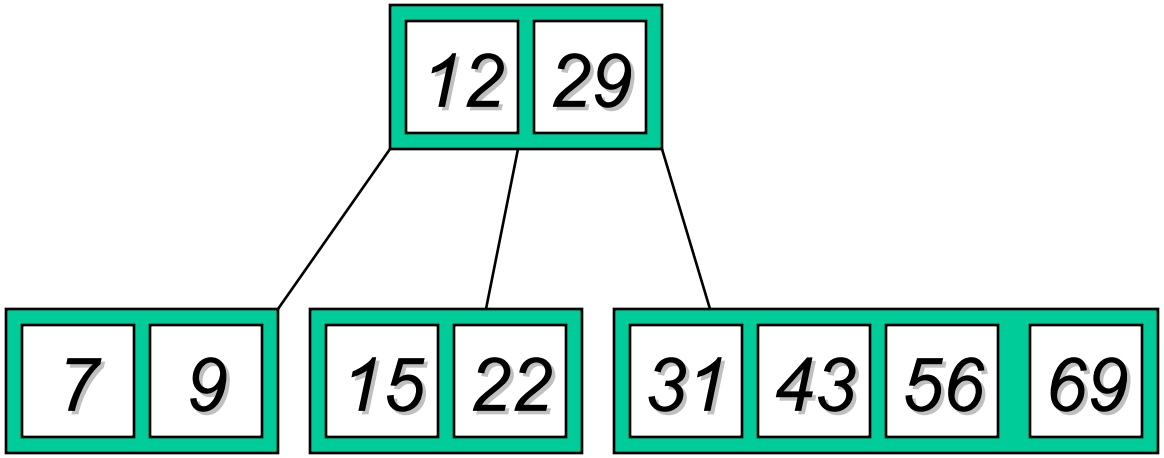
Note when printed: this slide is animated

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



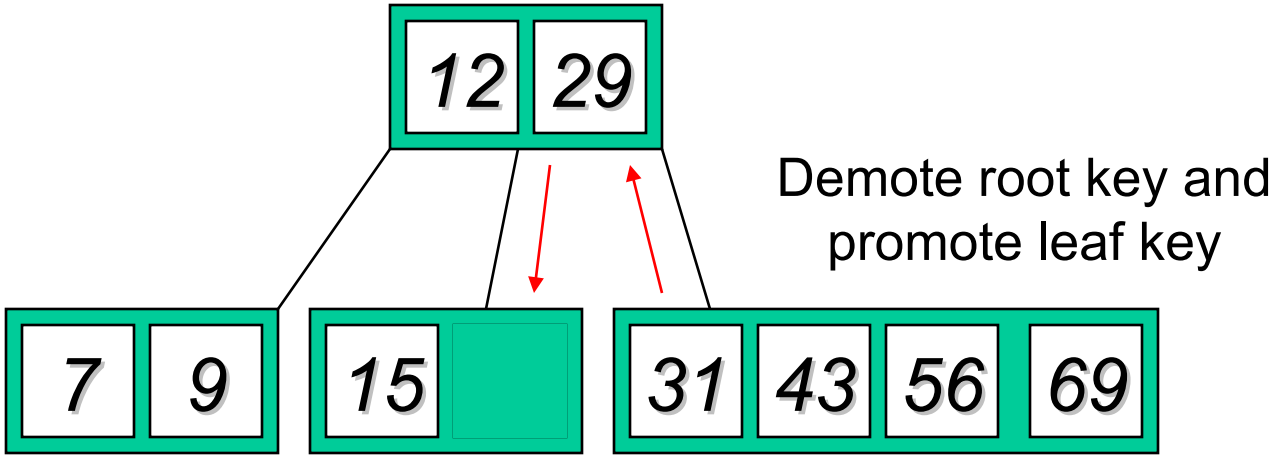
Note when printed: this slide is animated

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



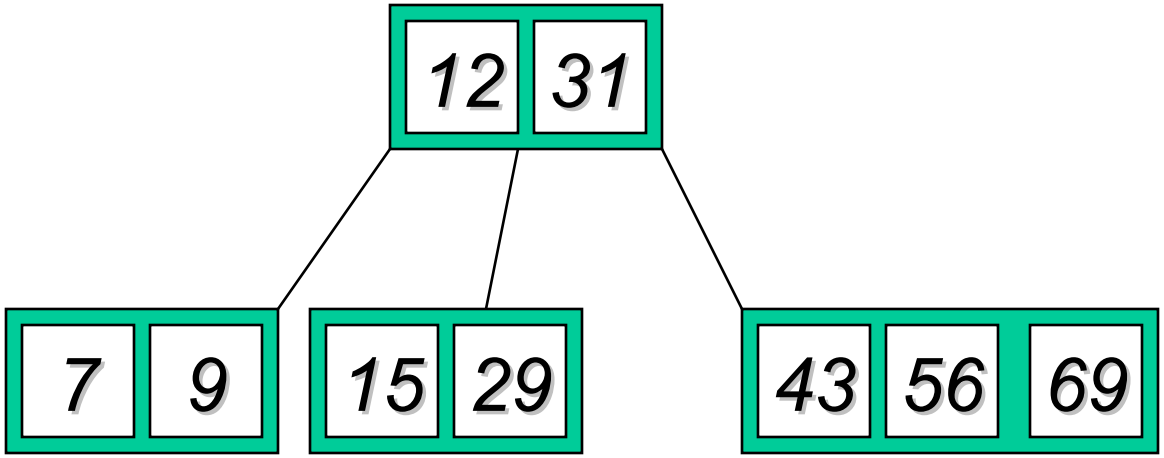
Note when printed: this slide is animated

Type #3: Enough siblings



Note when printed: this slide is animated

Type #3: Enough siblings



Note when printed: this slide is animated

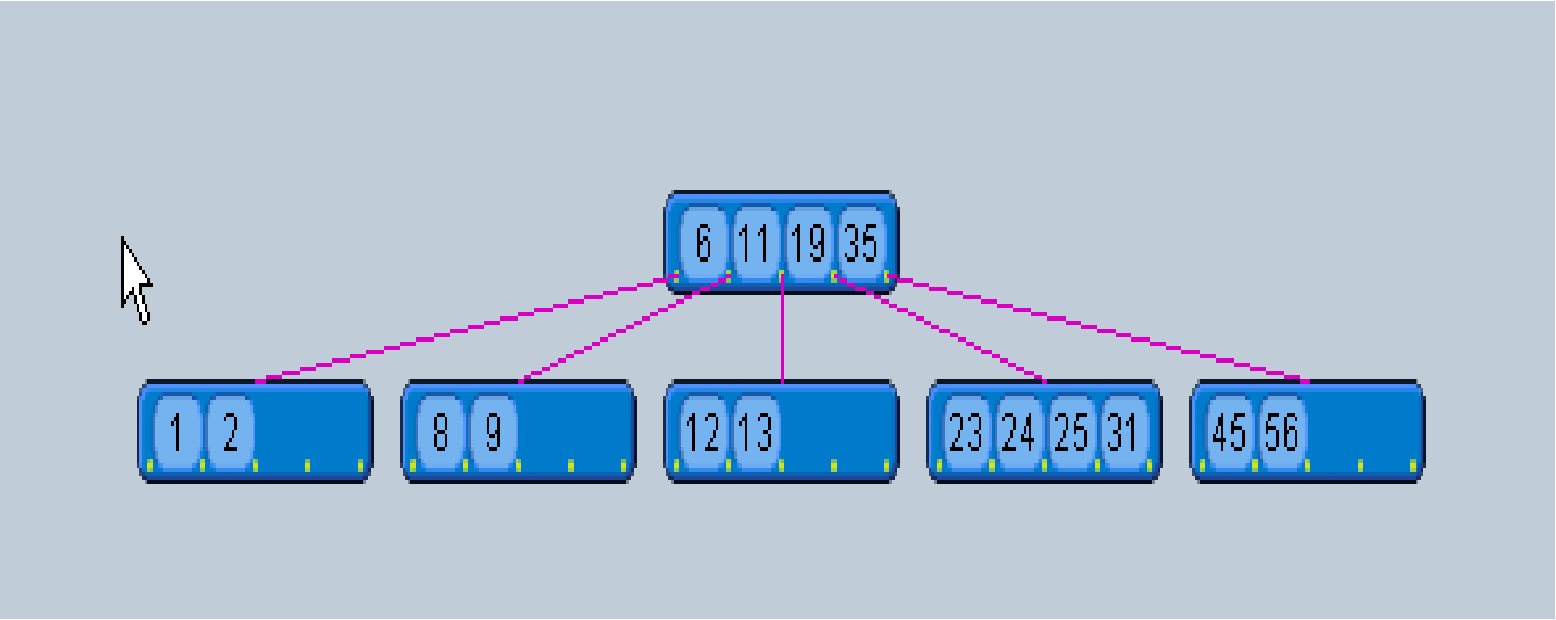
Exercise in Removal from a B-Tree

- Given 5-way 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

Answer to Exercise



Java Applet Source

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

B-Tree Assignment

- Rest of slides will talk about the code necessary for the implementation of a b-tree class