Data Structures in Java

Lecture 11: B-Trees.

10/14/2015

Daniel Bauer

Homework, Midterm etc.

- Homework 3 is out! Due: Friday October 23rd.
  Jarvis tests in preparation.

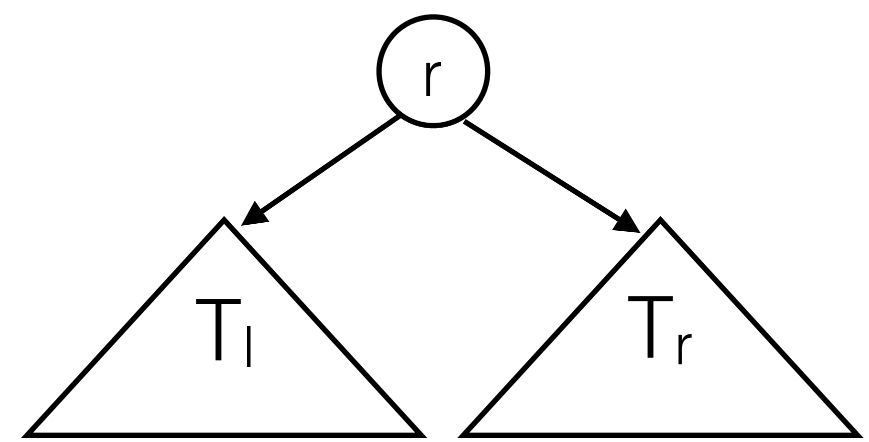
- Homework 2 grading is almost done.

  - Make sure to only submit .pdf and .txt (or Github markdown .md) for theory. Put the the main directory for each homework
    `homework-<youruni>/3/` and not
    `homework-<uni>/3/src/`

- Sample questions for Midterm to be released this weekend.

Review: Binary Search Trees

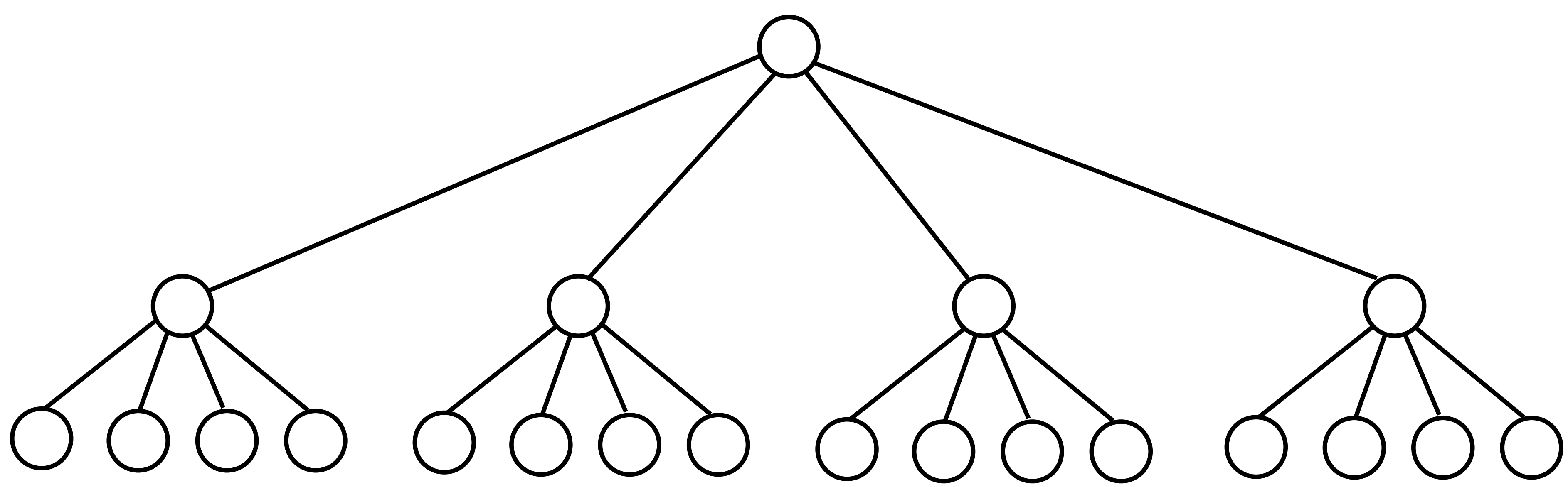
- BST property:
  - For all nodes *s* in T_l, $s_{item} < r_{item}$.
  - *F*or all nodes *t* in T_l, $t_{item} > r_{item}$.



- To keep BST operations (search/insert/delete/findMin/findMax) efficient, we need to maintain a **balanced tree:**
  - height of the tree should be close to log(N).
  - Example: AVL balancing condition, height difference between left and right subtree is at most 1.

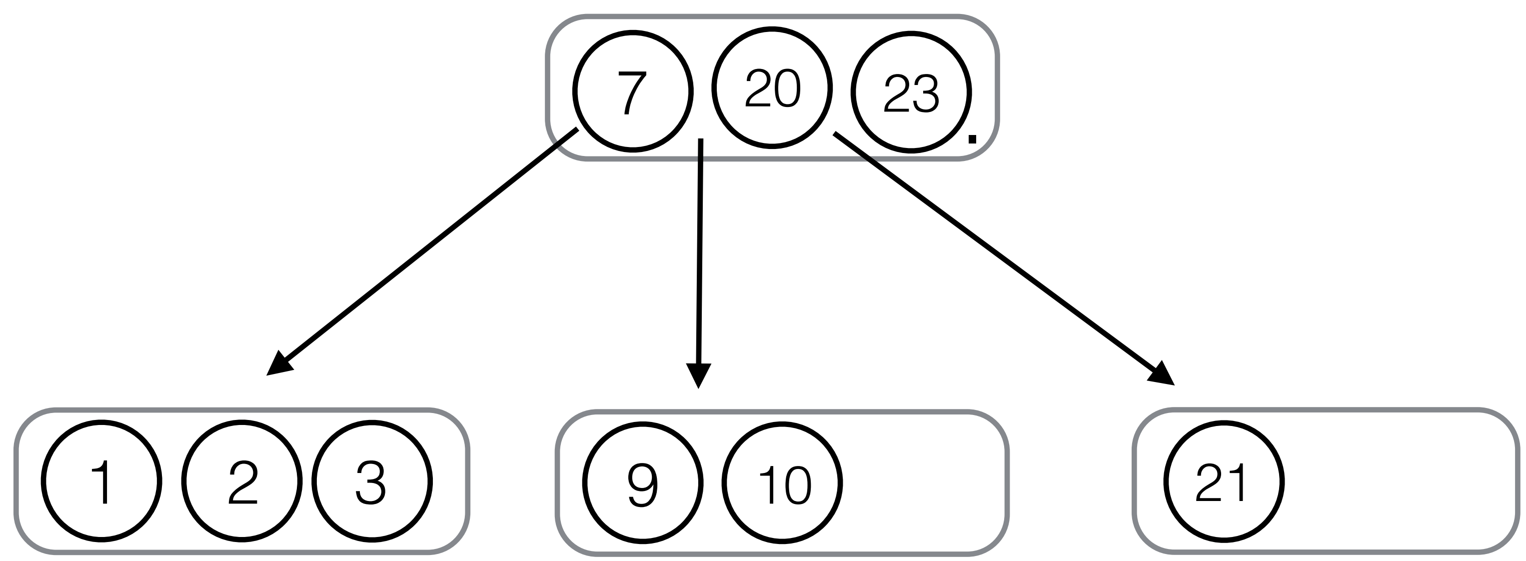
M-ary Trees

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



M-ary Search Tree

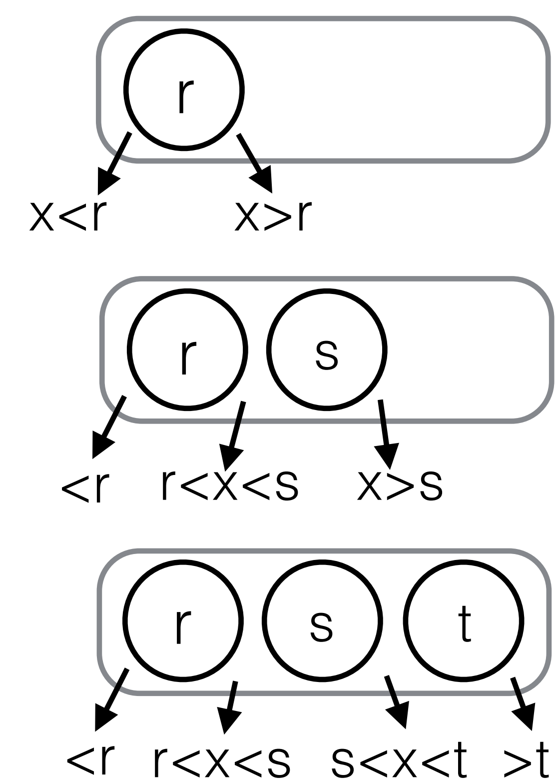
- We can generalize binary search trees to M-ary search trees.



4-ary search tree:
Nodes have 1,2, or 3 data items and 0 to 4 children.

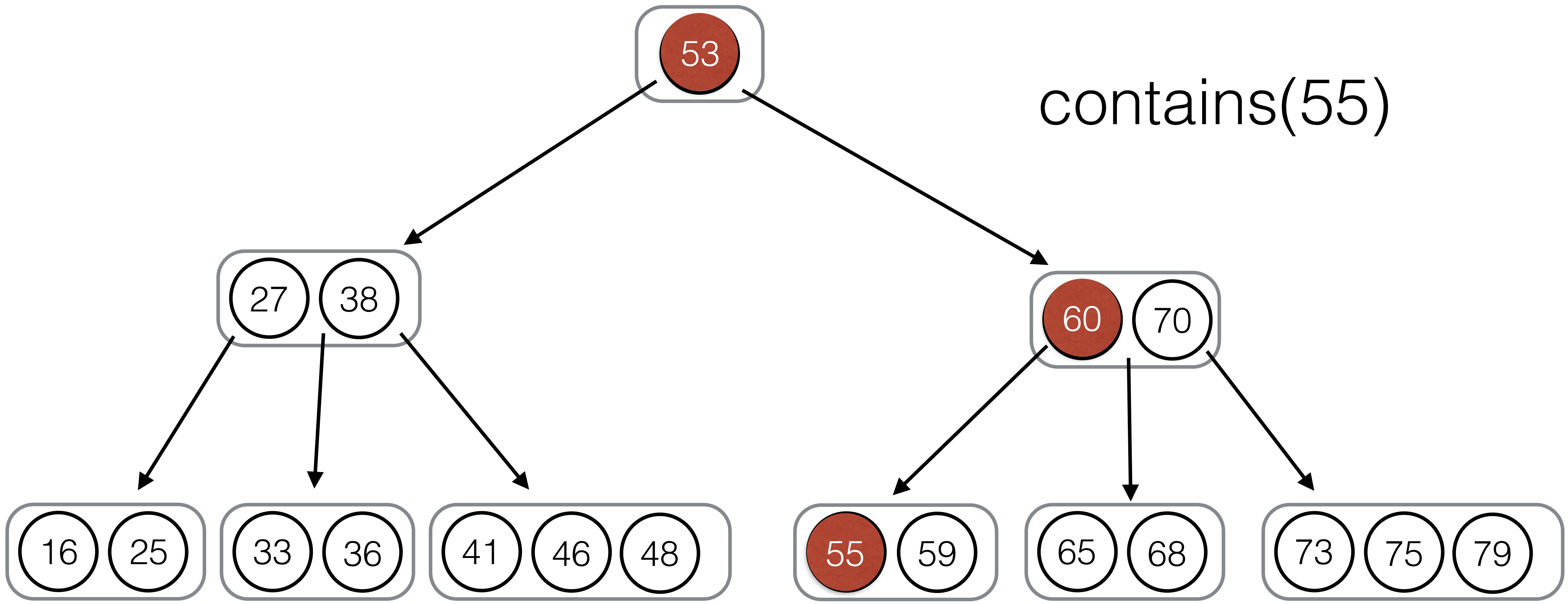
2-3-4 Trees

- A 2-3-4 Tree is a balanced 4-Ary search tree.
- Three types of internal nodes:
  - a 2-node has 1 item and 2 children.
  - a 3-node has 2 item and 3 children.
  - a 4-node has 3 item and 4 children.



- Balance condition:
  All leaves have the same depth.
  (height of the left and right subtree is always identical)

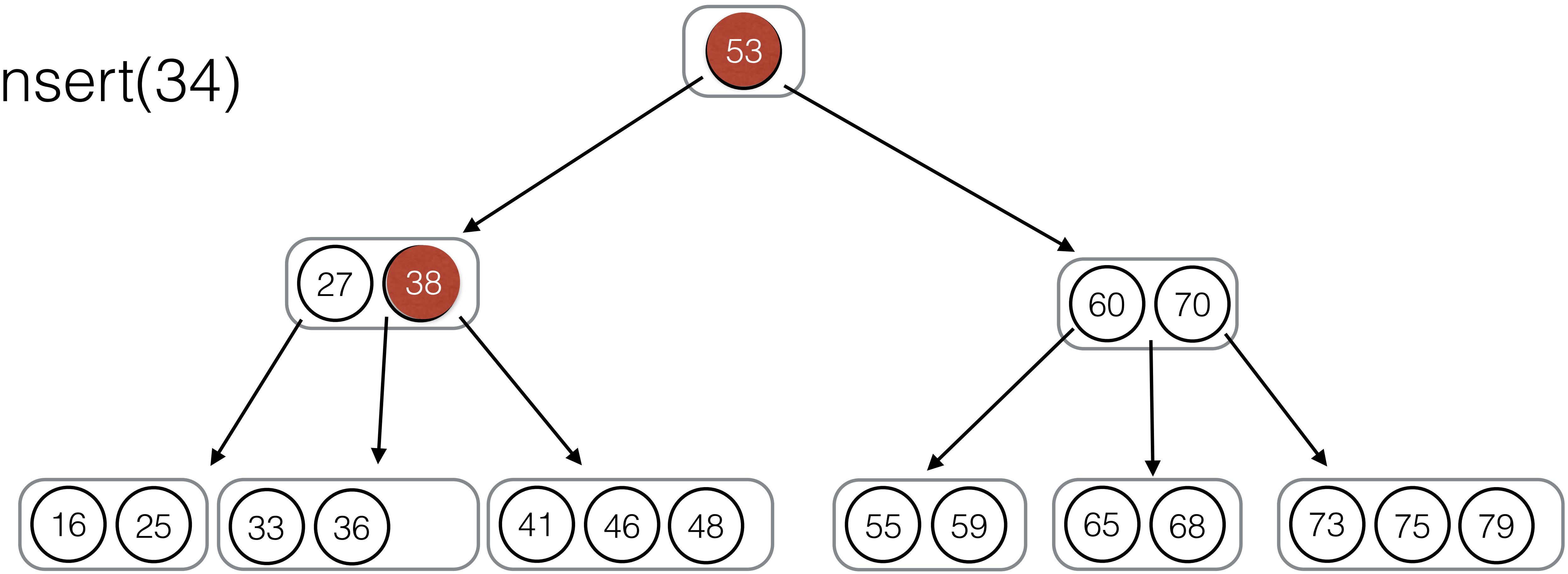
`contains` in a 2-3-4 Tree



- At each level try to find the item: 2 steps = O(c)
- If not found, follow reference down the tree. There are at most O(height(T)) = O(log N) references.

`insert` into a 2-3-4 Tree

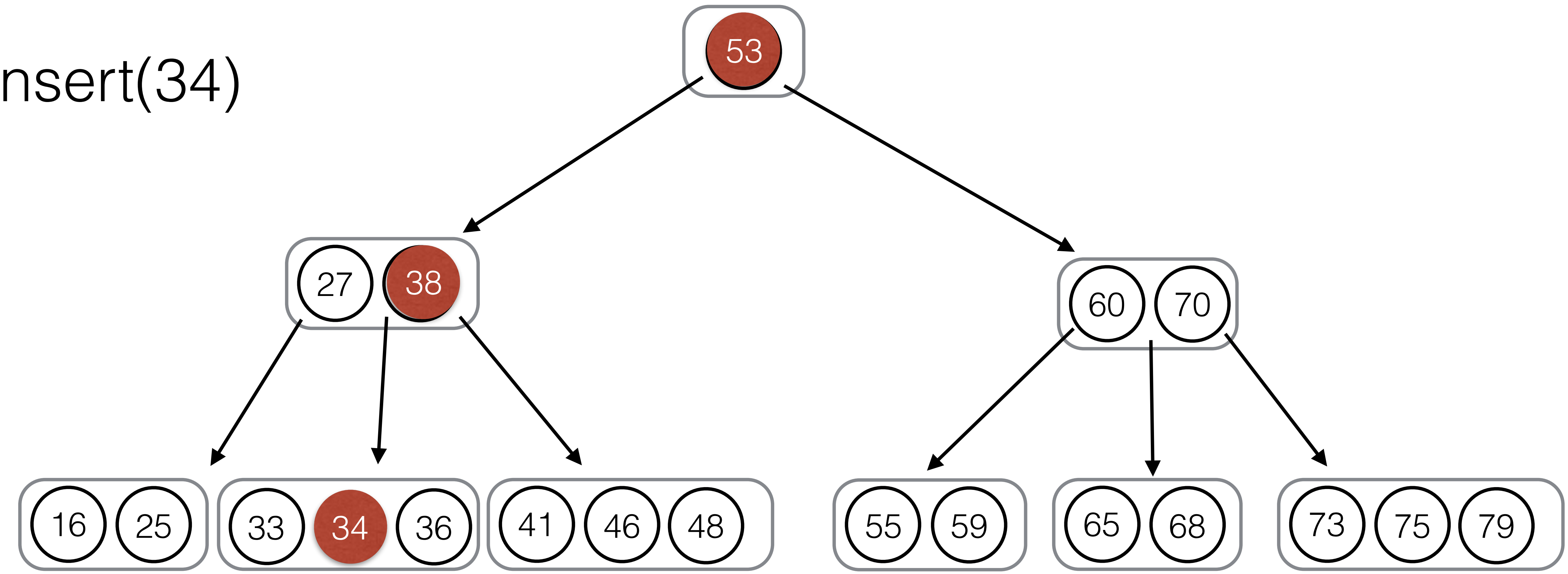
insert(34)



- Follow the same steps as contains.
- If X is found, do nothing.
- If there is still space in the leaf that should contain X, add it.

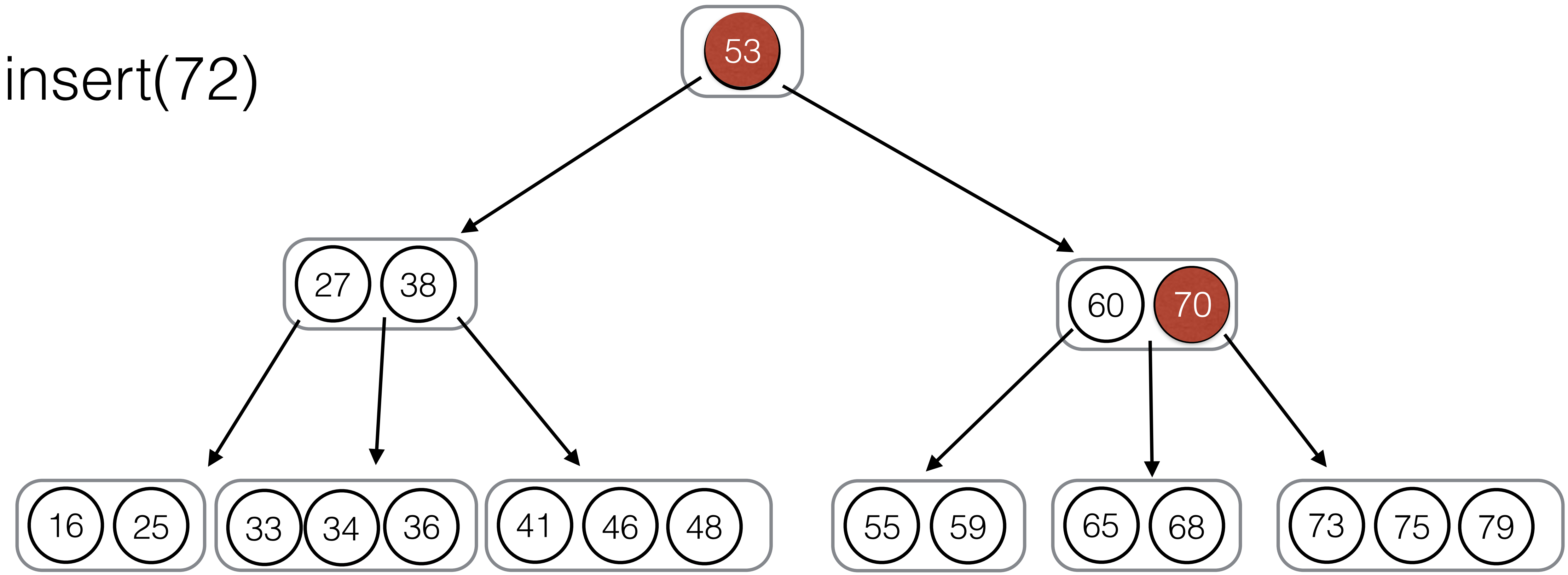
`insert` into a 2-3-4 Tree

insert(34)



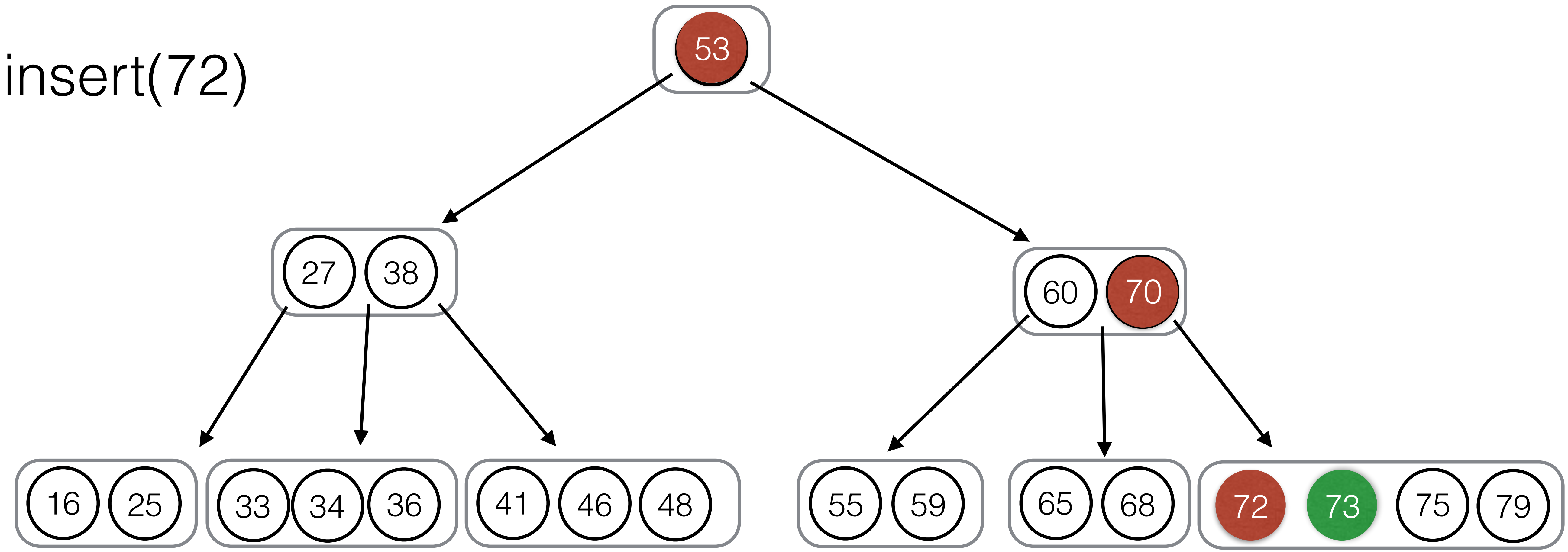
- Follow the same steps as contains.
- If X is found, do nothing.
- If there is still space in the leaf that should contain X, add it.
- **What if the leaf is full?**

`insert:` splitting nodes



- If the leaf is full, evenly split it into two nodes.
  - choose median *m* of values.
  - left node contains items < *m*, right node contains items >*m*.
  - add median items to parent, keep references to new nodes left and right of it.

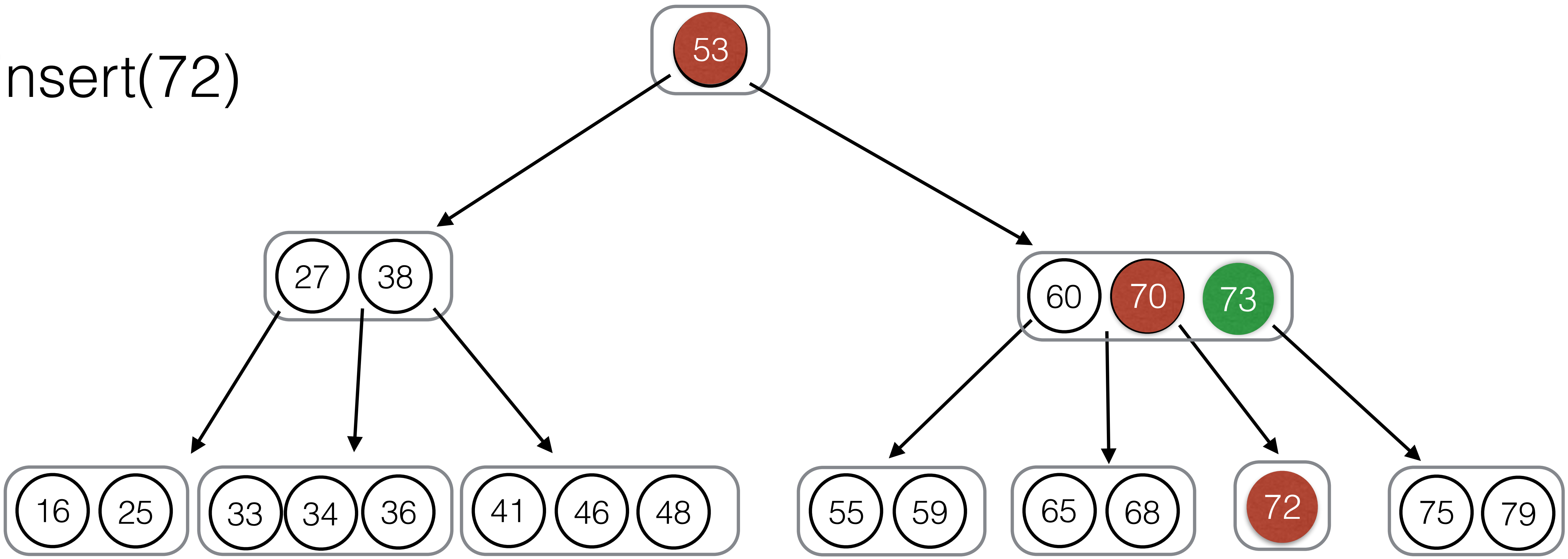
`insert`: splitting nodes



- If the leaf is full, evenly split it into two nodes.
  - choose median *m* of values.
  - left node contains items < *m*, right node contains items >*m*.
  - add median items to parent, keep references to new nodes left and right of it.

`insert`: splitting nodes

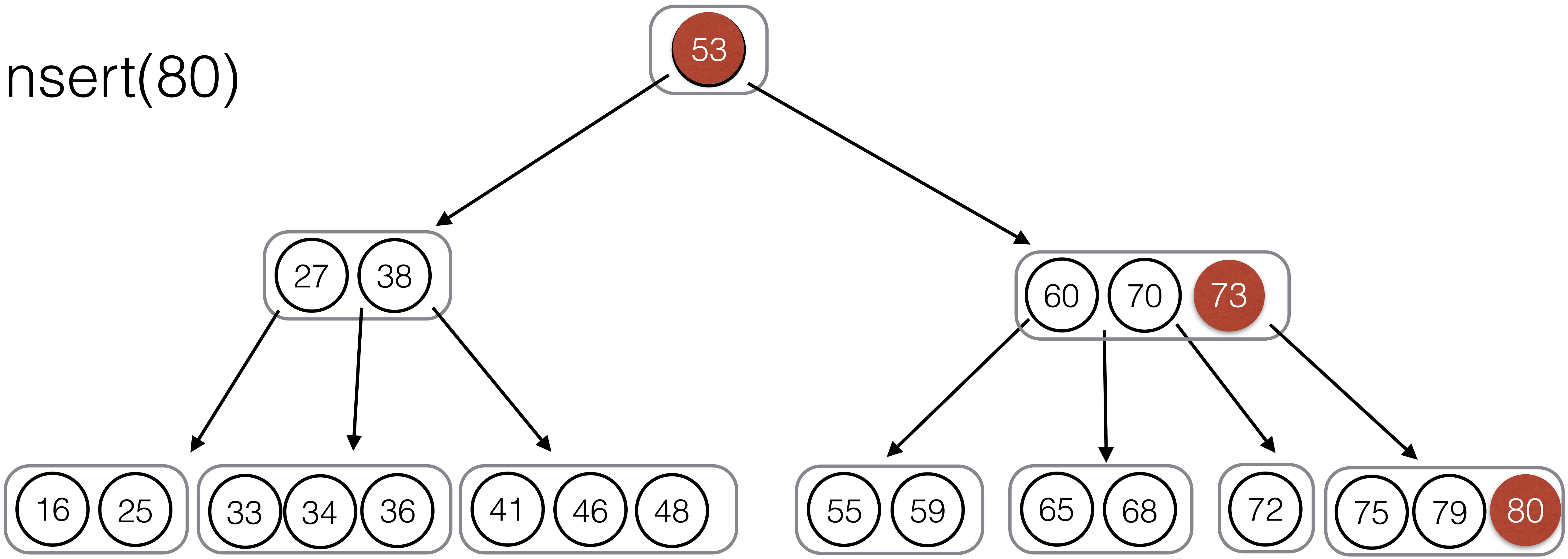
insert(72)



- If the leaf is full, evenly split it into two nodes.
  - choose median *m* of values.
  - left node contains items < *m*, right node contains items >*m*.
  - add median items to parent, keep references to new nodes left and right of it.

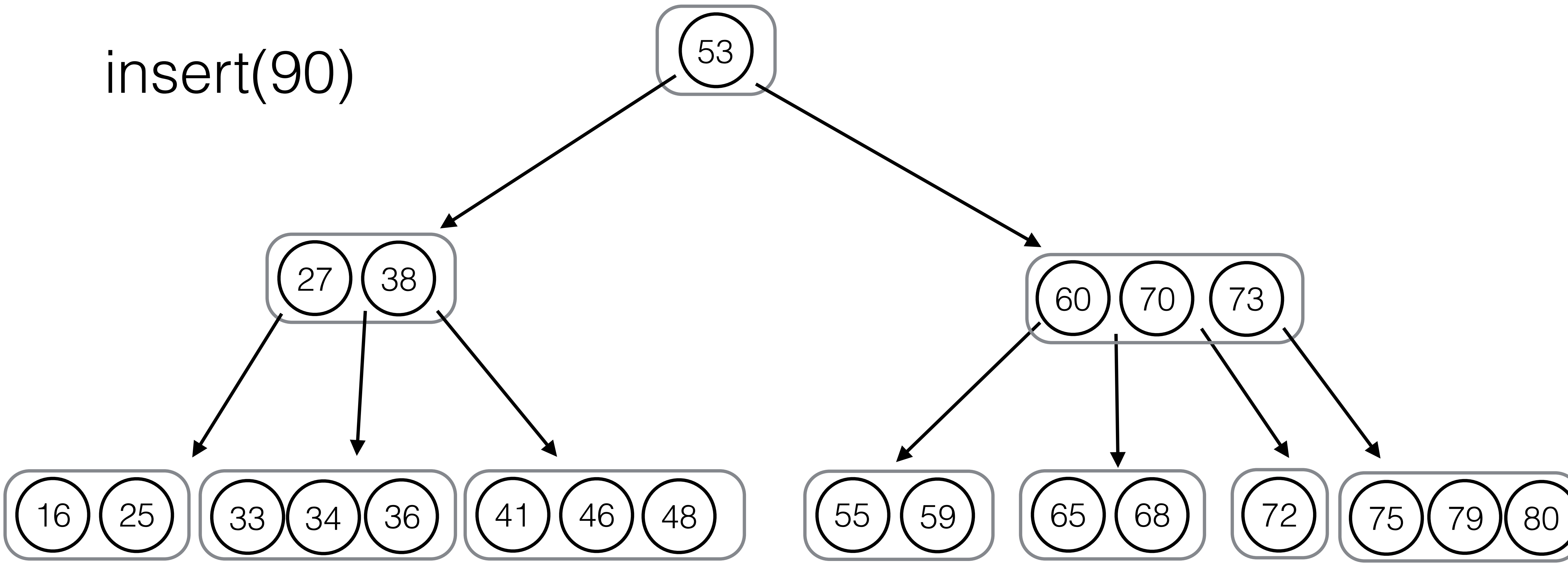
`insert`: splitting nodes

insert(80)



`insert`: splitting nodes

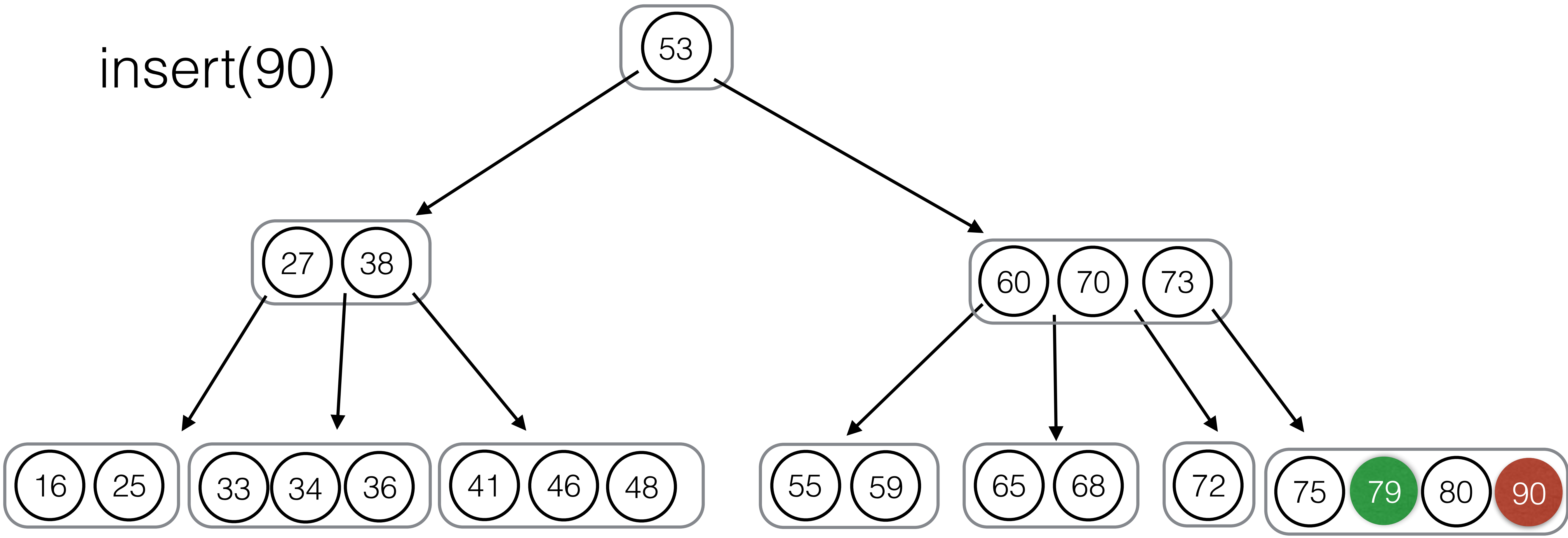
insert(90)



- If parent is also full, continue to split the parent until space can be found.
- If root is full, create a new root with old root as a single child.
- At most we need one pass down the tree and one pass up, so insertion is O(log N).

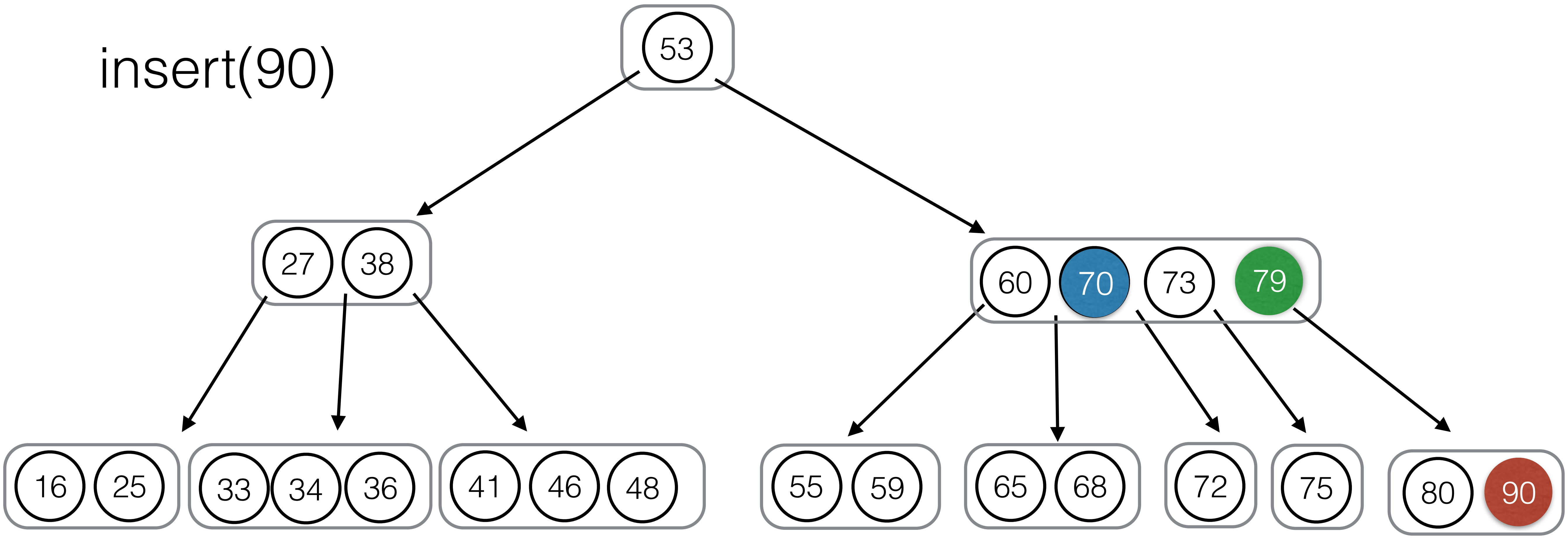
`insert`: splitting nodes

insert(90)



- If parent is also full, continue to split the parent until space can be found.
- If root is full, create a new root with old root as a single child.
- At most we need one pass down the tree and one pass up, so insertion is O(log N).

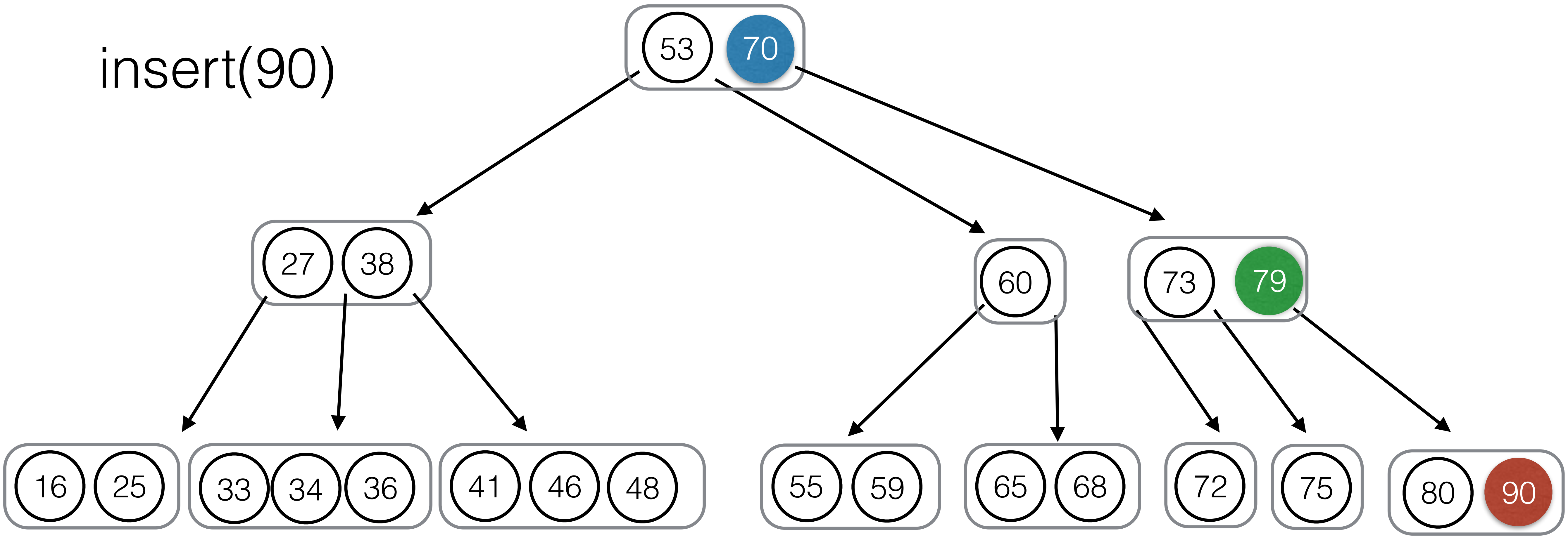
`insert`: splitting nodes



- If parent is also full, continue to split the parent until space can be found.
- If root is full, create a new root with old root as a single child.
- At most we need one pass down the tree and one pass up, so insertion is O(log N).

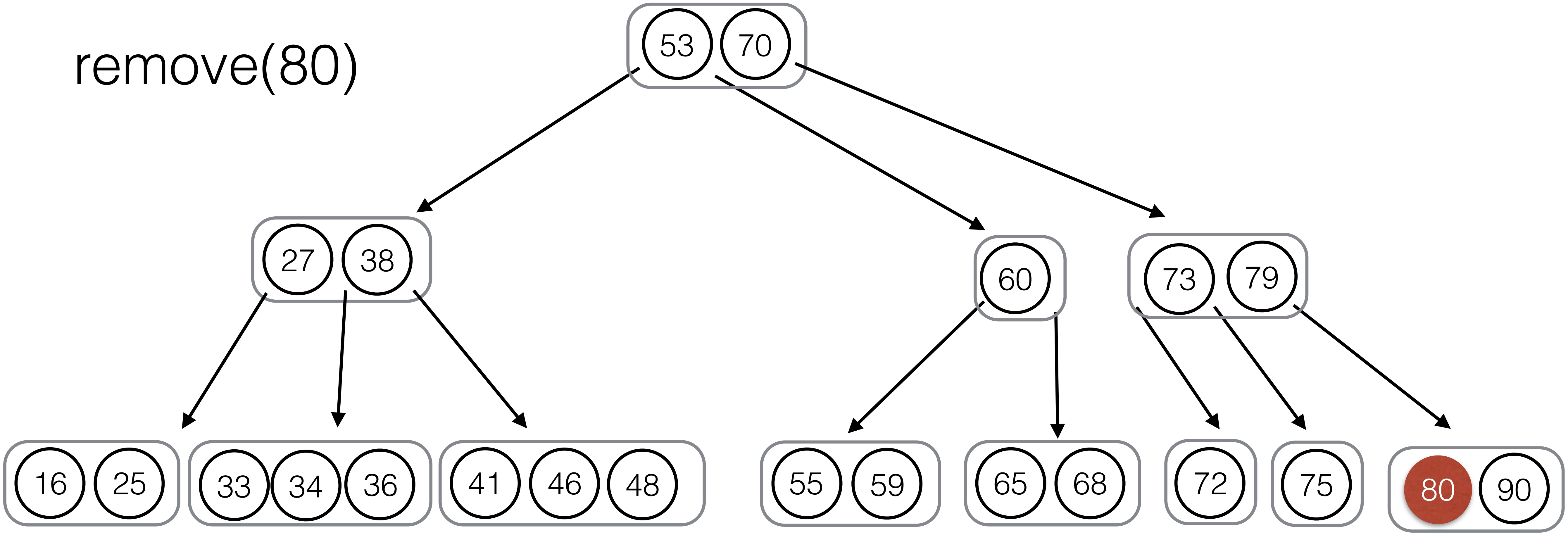
`insert`: splitting nodes

insert(90)



- If parent is also full, continue to split the parent until space can be found.
- If root is full, create a new root with old root as a single child.
- At most we need one pass down the tree and one pass up, so insertion is O(log N).

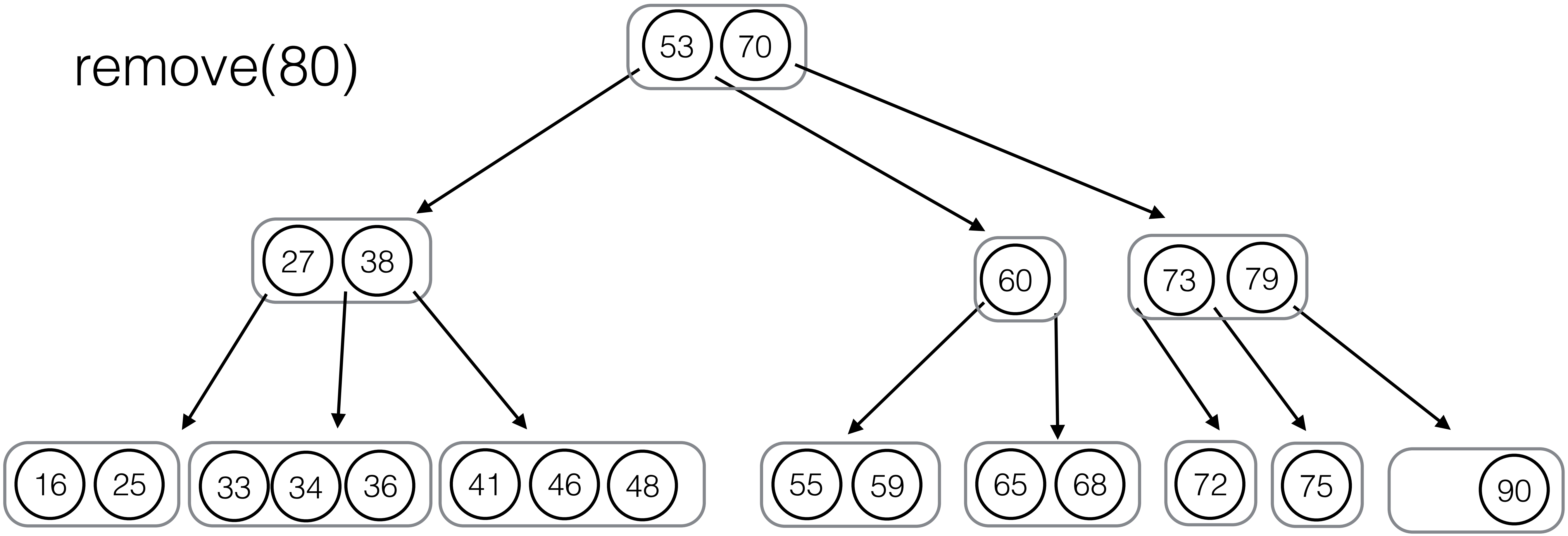
`remove` from a 2-3-4 tree



- Item in a 3- or 4-leaf can just be removed.

`remove` from a 2-3-4 tree

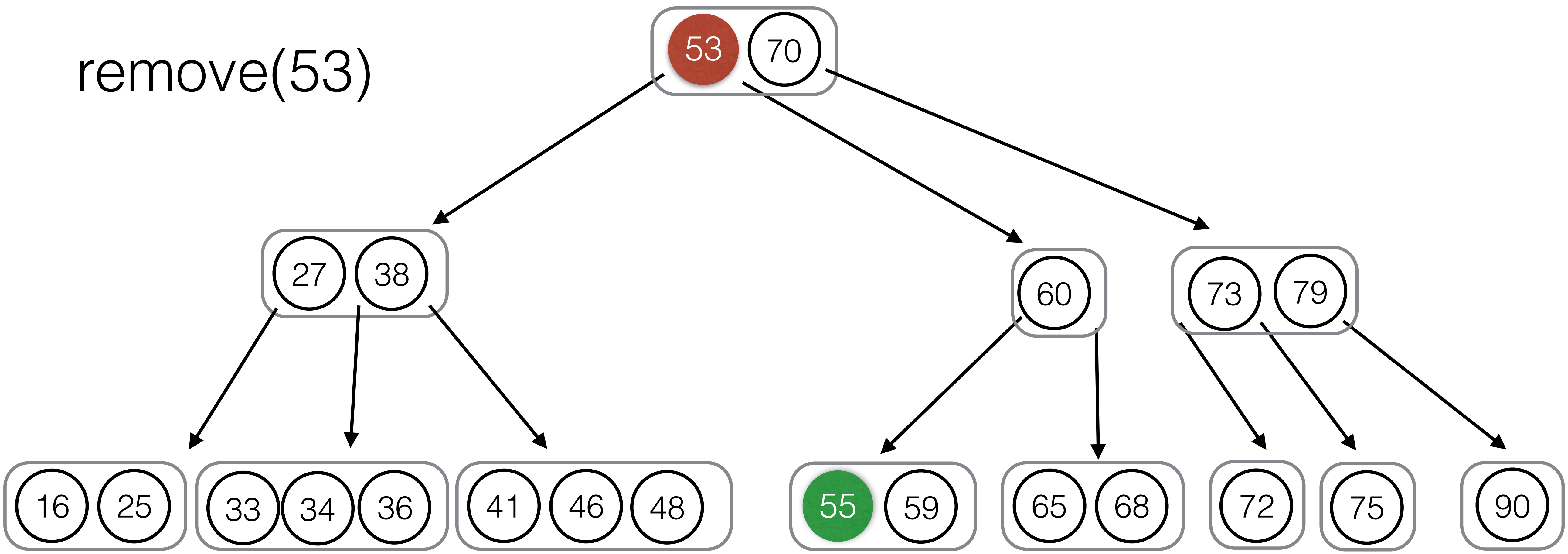
remove(80)



- Item in a 3- or 4-leaf can just be removed.

`remove` from a 2-3-4 tree

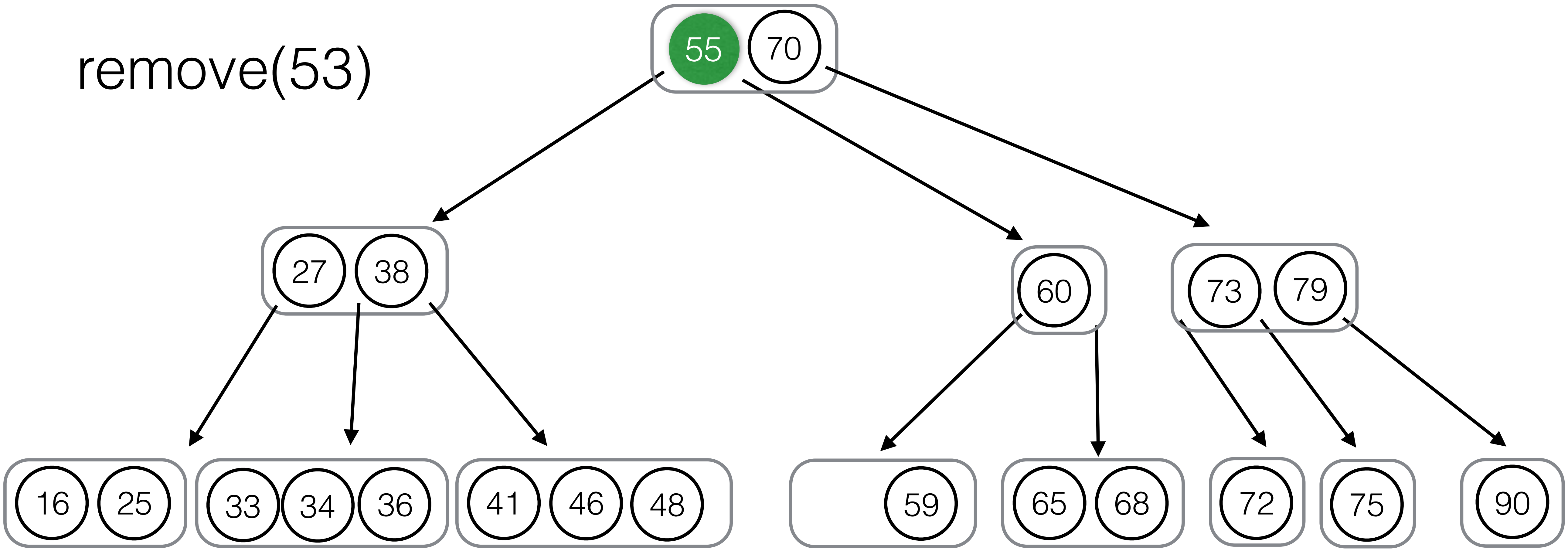
remove(53)



- Removal of an item v from internal node:
  - Continue down the tree to find the leaf with the next highest item *w*. Replace *v* with *w*. Remove *w* from its original position recursively.

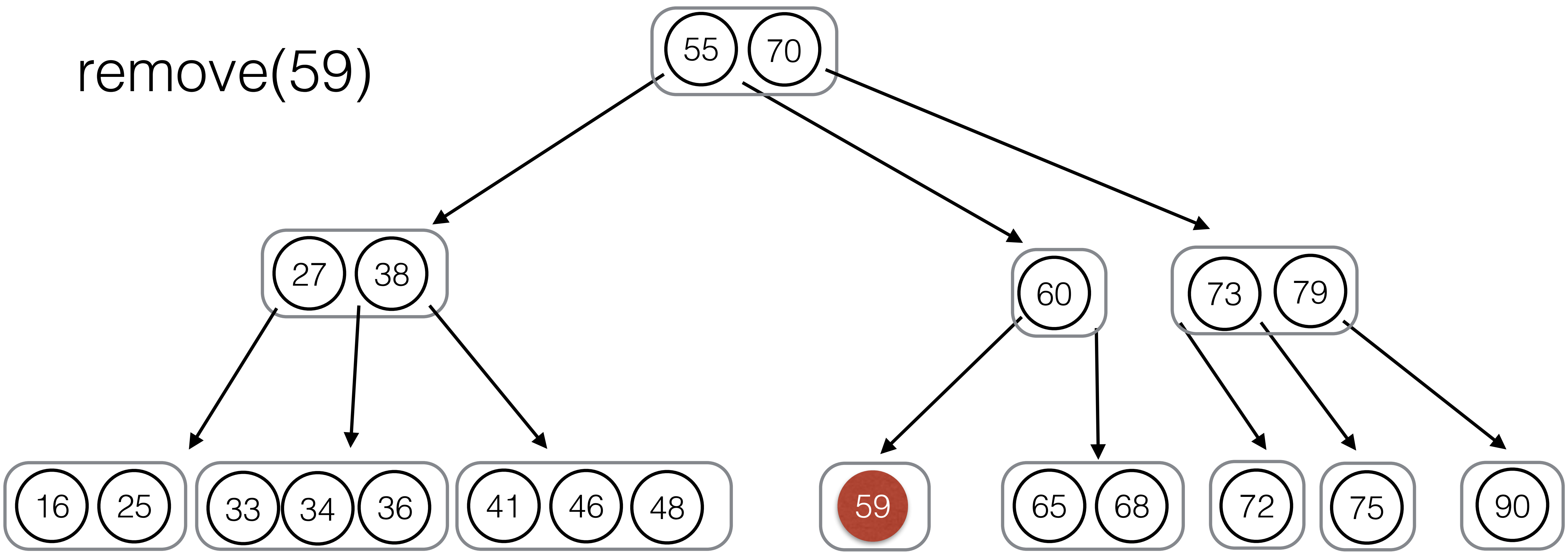
`remove` from a 2-3-4 tree

remove(53)



- Removal of an item v from internal node:
  - Continue down the tree to find the leaf with the next highest item *w*. Replace *v* with *w*. Remove *w* from its original position recursively.

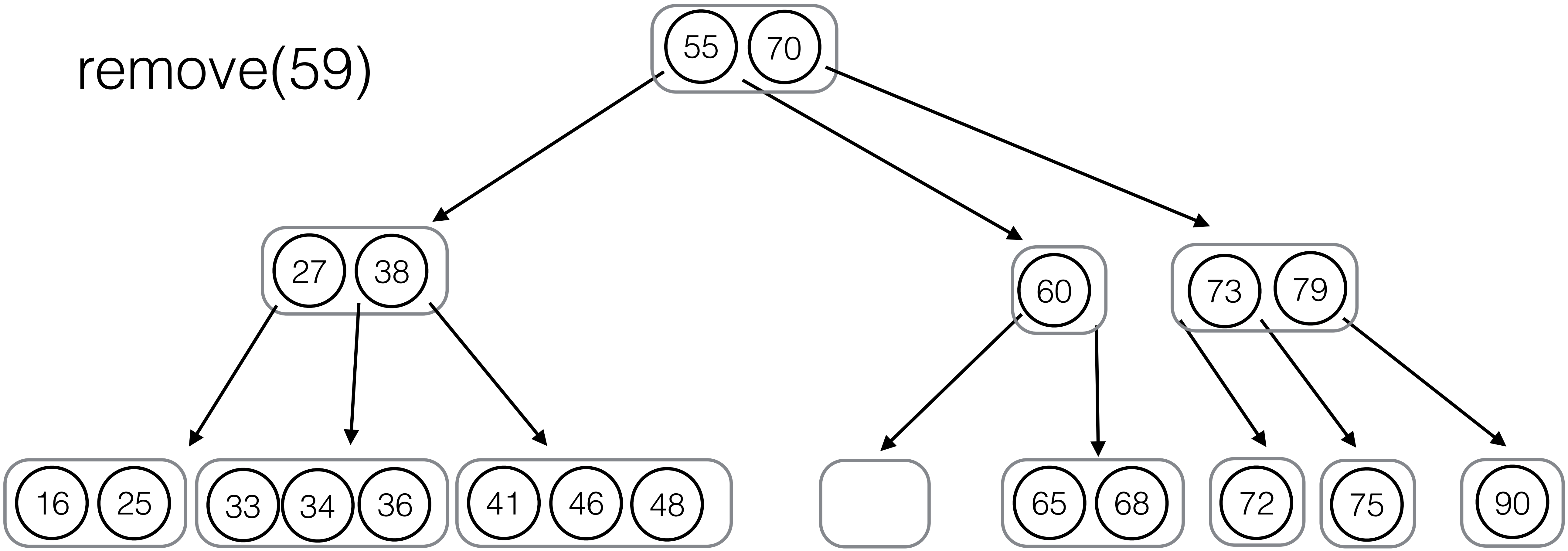
`remove` from a 2-3-4 tree



- Removal of an item form a leaf 2-node *t:*
  - We cannot simply remove *t* because the parent would not be well formed.
  - Move down an item from the parent of t. Replenish the parent by moving item from one of *t's* siblings.

`remove` from a 2-3-4 tree

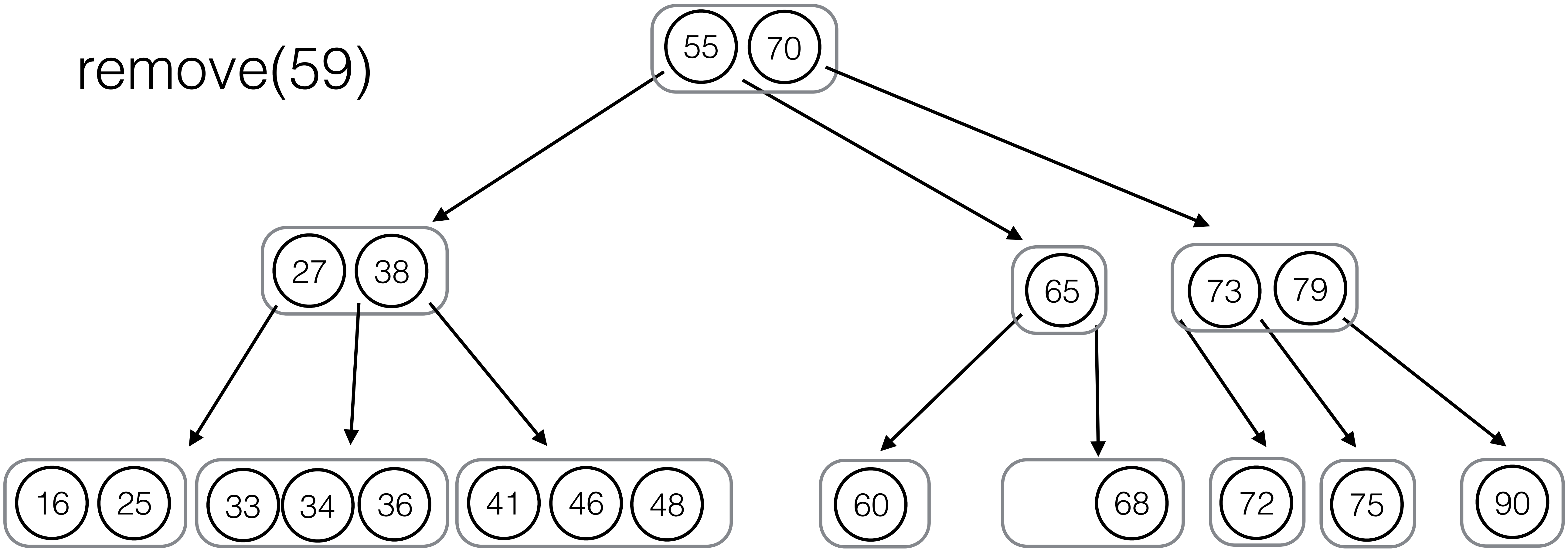
remove(59)



- Removal of an item form a leaf 2-node *t:*
  - We cannot simply remove *t* because the parent would not be well formed.
  - Move down an item from the parent of t. Replenish the parent by moving item from one of *t's* siblings.

`remove` from a 2-3-4 tree

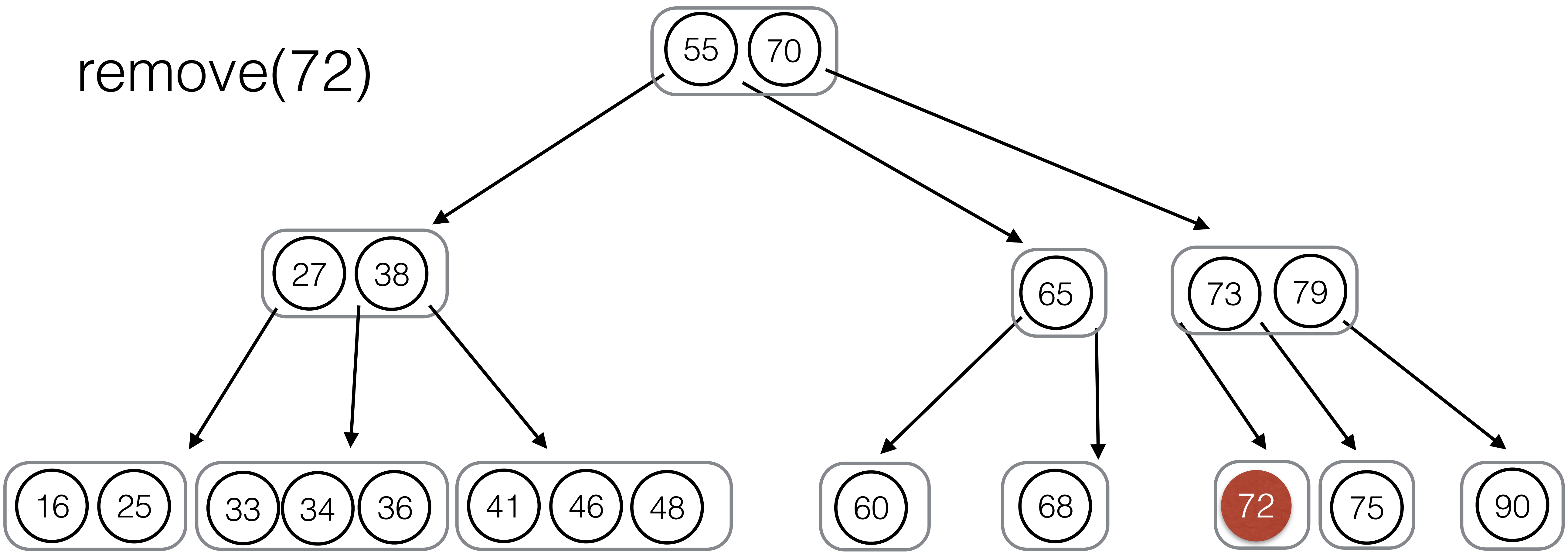
remove(59)



- Removal of an item form a leaf 2-node *t:*
  - We cannot simply remove *t* because the parent would not be well formed.
  - Move down an item from the parent of t. Replenish the parent by moving item from one of *t's* siblings.

What if no sibling is a 3 or 4 node?

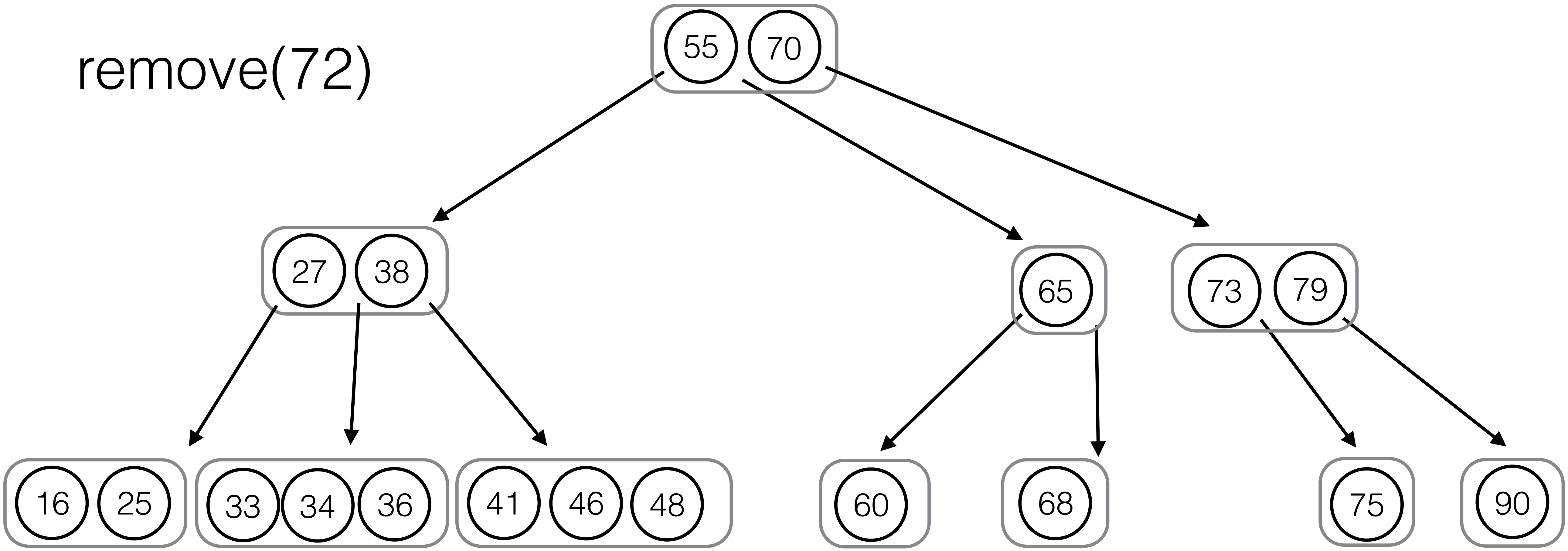
`remove` from a 2-3-4 tree



- Removal of a an item in a leaf 2-node that has no 3- or 4-node siblings:
  - **Fuse** the sibling node with one of the parent nodes.

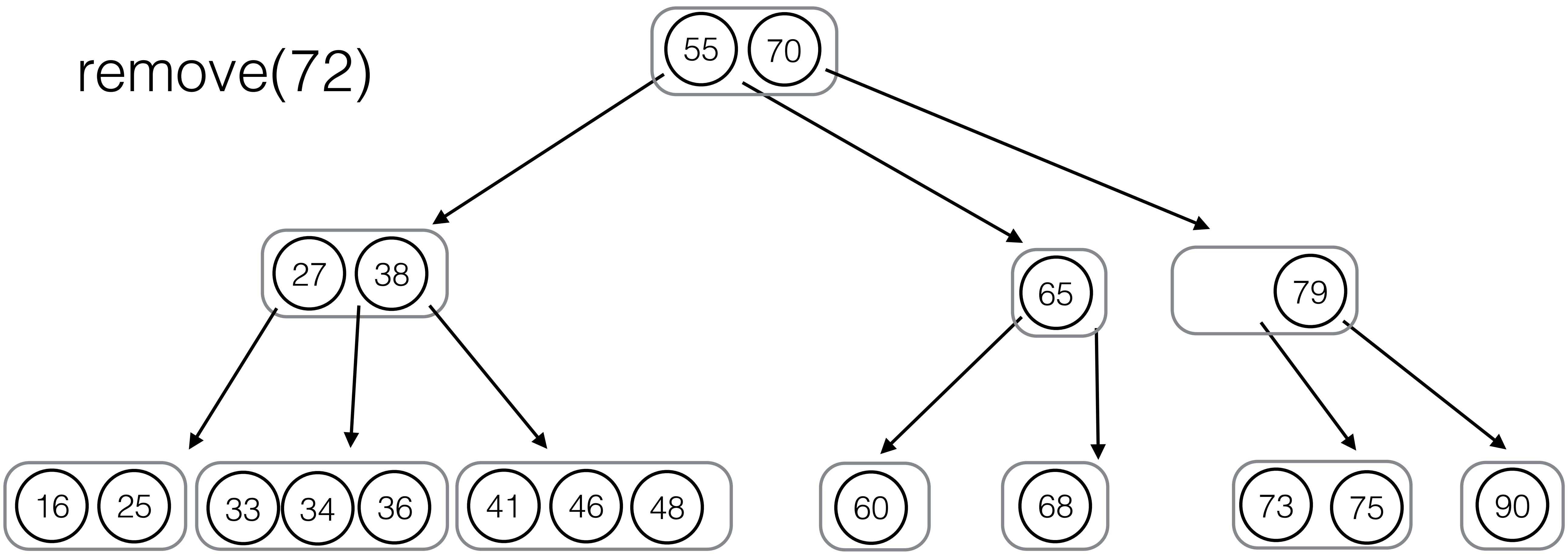
`remove` from a 2-3-4 tree

remove(72)



- Removal of a an item in a leaf 2-node that has no 3- or 4-node siblings:
  - **Fuse** the sibling node with one of the parent nodes.

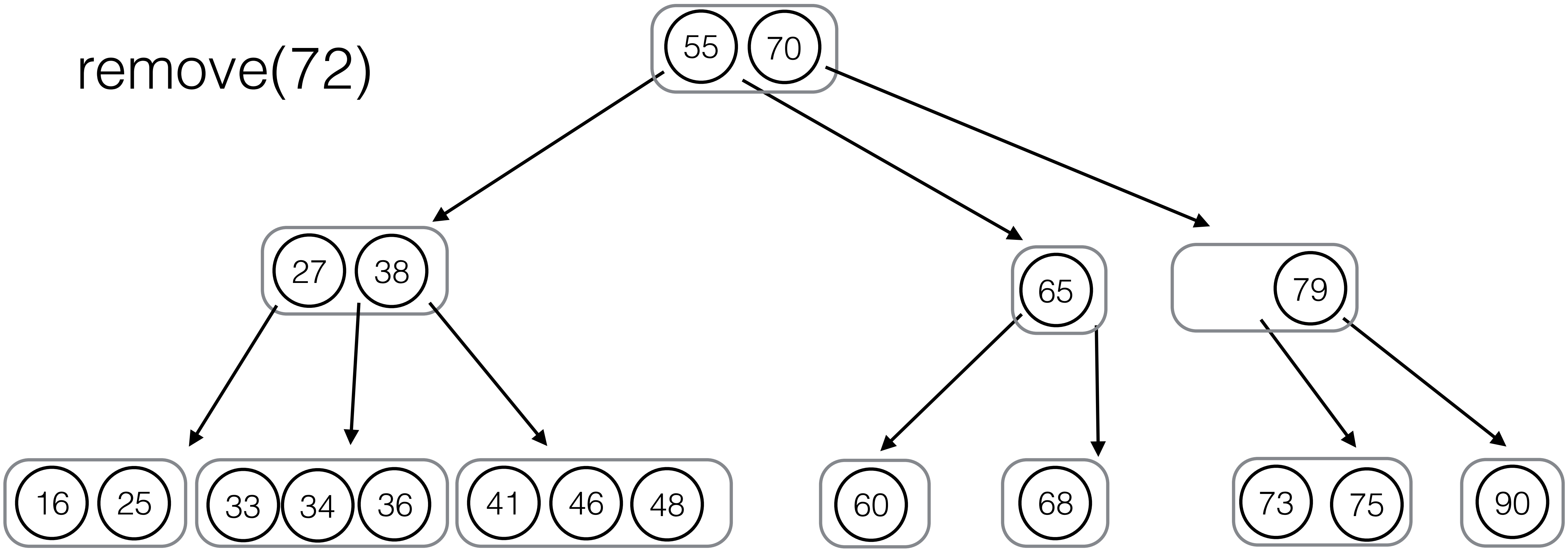
`remove` from a 2-3-4 tree



- Removal of a an item in a leaf 2-node that has no 3- or 4-node siblings:
  - **Fuse** the sibling node with one of the parent nodes.

`remove` from a 2-3-4 tree

remove(72)

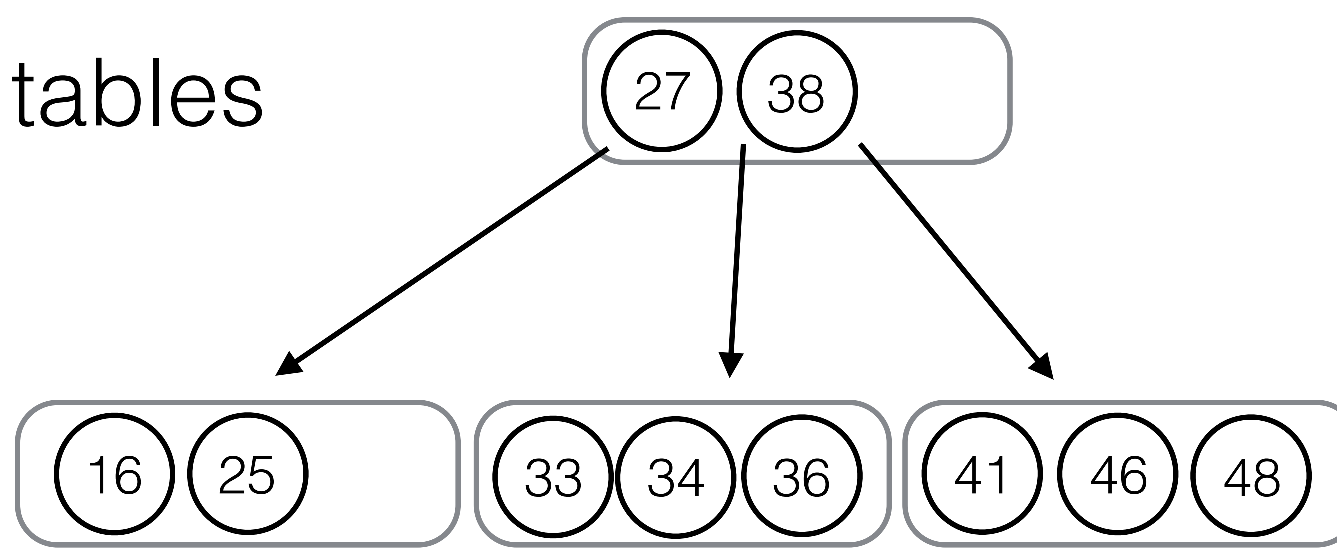


- Removal of a an item in a leaf 2-node that has no 3- or 4-node siblings:
  - **Fuse** the sibling node with one of the parent nodes.

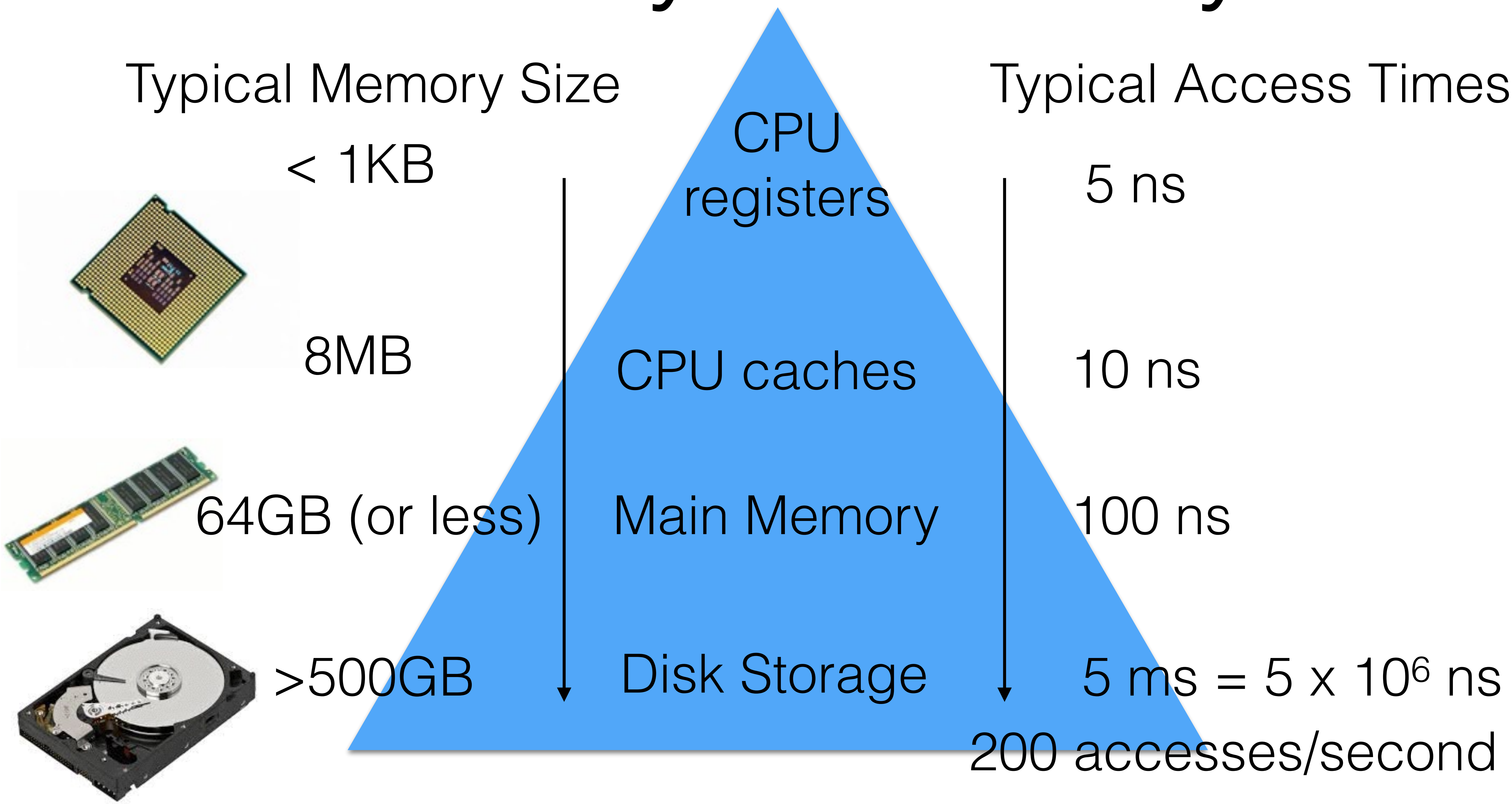
All modifications to fix the tree are local and therefore O(c).
Remove runs in O(log N).

B-Trees

- A B-Tree is a generalization of the 2-3-4 tree to M-ary search trees.
- Every internal node (except for the root) has $\lceil \frac{M}{2} \rceil \leq d \leq M$ children and contains $\boldsymbol{d-1}$ values.
- All leaves contain $\lceil \frac{L}{2} \rceil \leq d \leq L$ values (usually L=M-1)
- All leaves have the same depth.
- Often used to store large tables on hard disk drives. (databases, file systems)



Memory Hierarchy



Memory access is **much** faster than disk access.

Large BST on Disk (1)

- Assume we have a very large database table, represented as a binary search tree:
  - 10 million items, 256 bytes each.
  - 6 disk accesses per second (shared system).
- Assume no caching, every lookup requires disk access.

Large BST on Disk (2)

- Disk access time for finding a node in an unbalanced BST:

  - depth of searched node is *N* in the **worst case**:
    - 10 million items -> 10 million disk accesses
    - 10 million / 6 accesses per second ≈ 19 days!

  - **Expected** depth is *1.38 log N*
    - *1.38* $\log_2 10 \times 10^6$ *items ≈ 32 disk accesses*
    - *32 / 6 accesses per second ≈ 5 seconds*

Large BST on Disk (2)

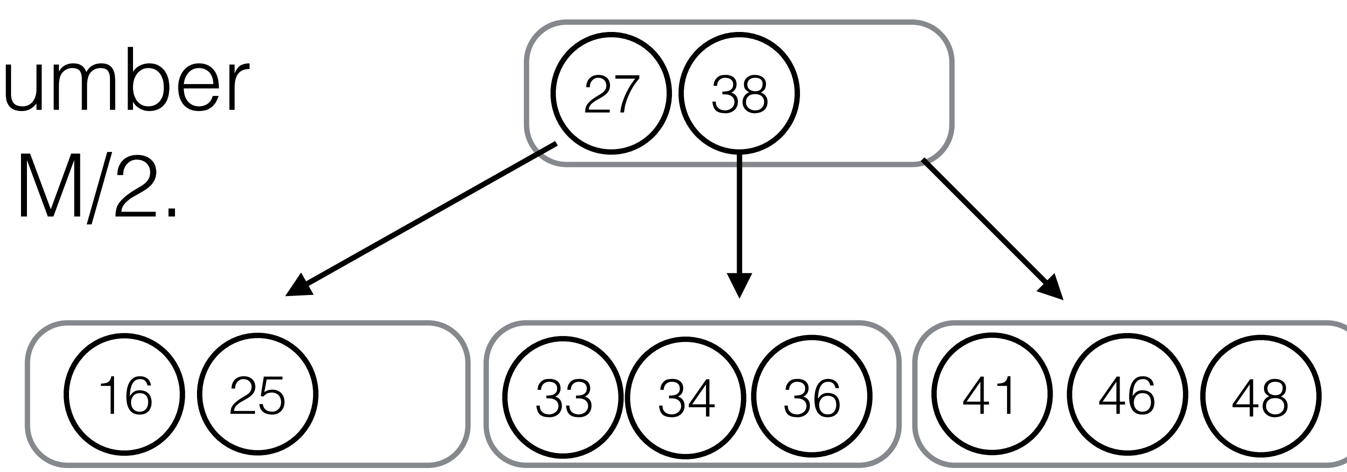
- Even for AVL Tree the worst case and average case will be around log N.
- *A*bout 24 disk accesses in 4 sec.

Storing B-Trees on Disk

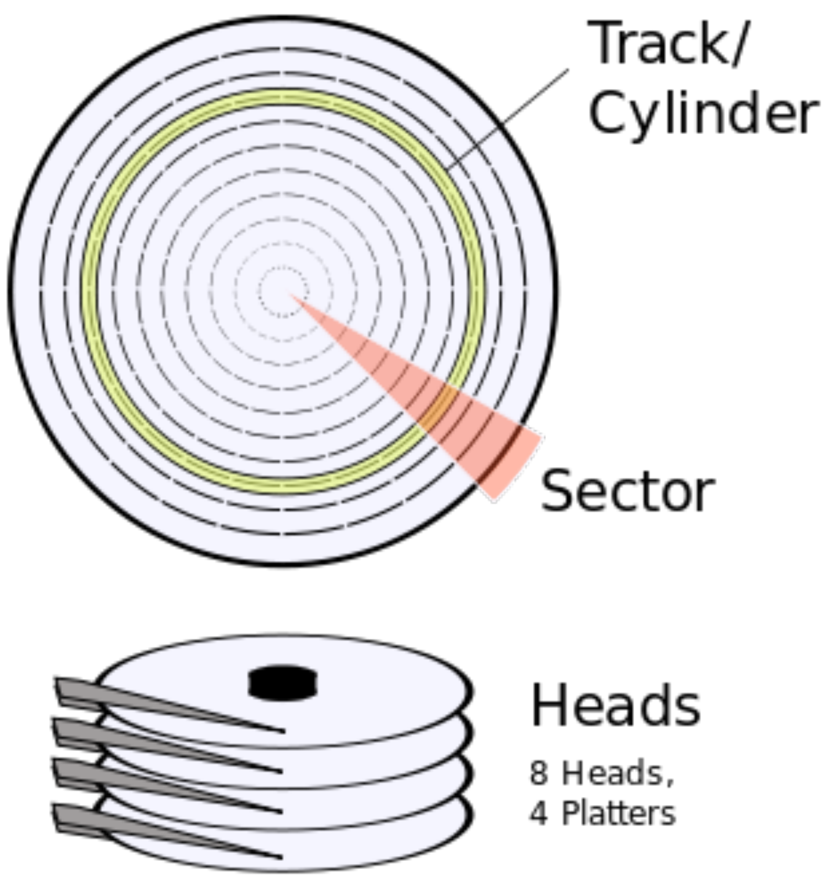
- We can use B-Trees to reduce the number of disk accesses. Basic idea:

  - Read an entire B-Tree node (containing M items) into memory in *single disk access*. Find the next reference using binary search.

  - Worst case height of the B-Tree is about $\log_{\frac{M}{2}} N$ because the minimum number of items in each node is M/2.



Hard Disk Drive Layout



- A **sector** is the minimal unit of data that can be read from the disk.

- Typical **physical sector size**:
512 byte (modern drives: 4096 byte)

- **Blocks** are logical units of adjacent sectors (defined by the operating system).
Typical block sizes are
1KB, 2KB, 4KB, 8KB.

Estimating the ideal M for a B-Tree

(M-1)*256 bytes

…

M * 8 bytes

- Assume 8KB= 8,192 byte block size.
- Every data item is 256 byte.
- An M-ary B-Tree contains at most *M-1* data items + M block addresses of other trees (a 8 byte pointer each).
- How big can we make the nodes?

$$(M-1)\cdot \mathbf{256\ byte} + M \cdot \mathbf{8\ byte} = \mathbf{8,192\ byte}$$

$$M = \mathbf{32}$$

Calculating Access Time

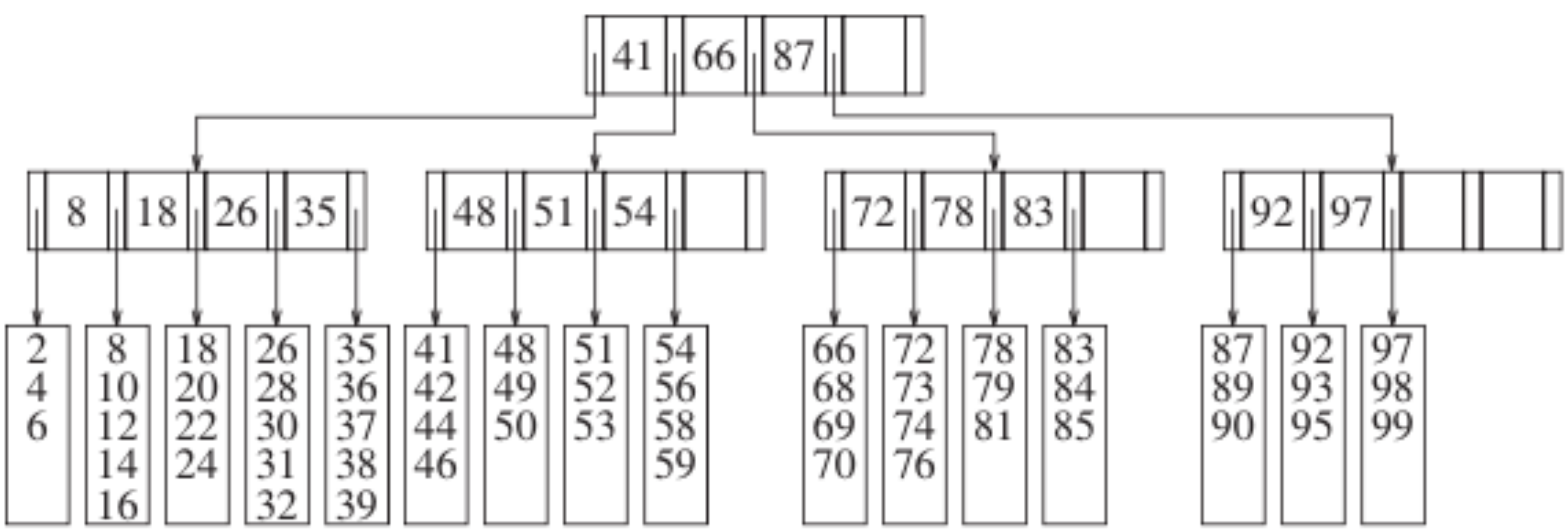
- We representing 10,000,000 items in a B-Tree with M=32
- The tree has a worst-case height of $log_{\frac{M}{2}} N$

$$log_{\frac{32}{2}} 10,000,000 \approx 6$$

- Worst-case time to find an item is
  6 accesses / 6 disk accesses per second = 1 *second*

B+ Trees

- Only leafs store full (key, value) pairs.
- Internal nodes only contain keys to help find the right leaf.
- Insert/removal only at leafs (slightly simpler, see book).



Weiss, Data Structures and Algorithms in Java, 3rd Ed.

B+ Trees on Disk

- Assume keys are 32 bytes.

$$(M-1)\cdot\mathbf{32\ byte}+M\cdot\mathbf{8\ byte}=\mathbf{8,192\ byte}$$

- We can fit at most M=205 keys in each node.

- Worst case time for 1 million keys:

$$\mathbf{log}_{\frac{205}{2}}\ \mathbf{10,000,000=3}$$

  - 3 accesses / 6 seconds per access = .5 seconds