BINARY HEAPS 2

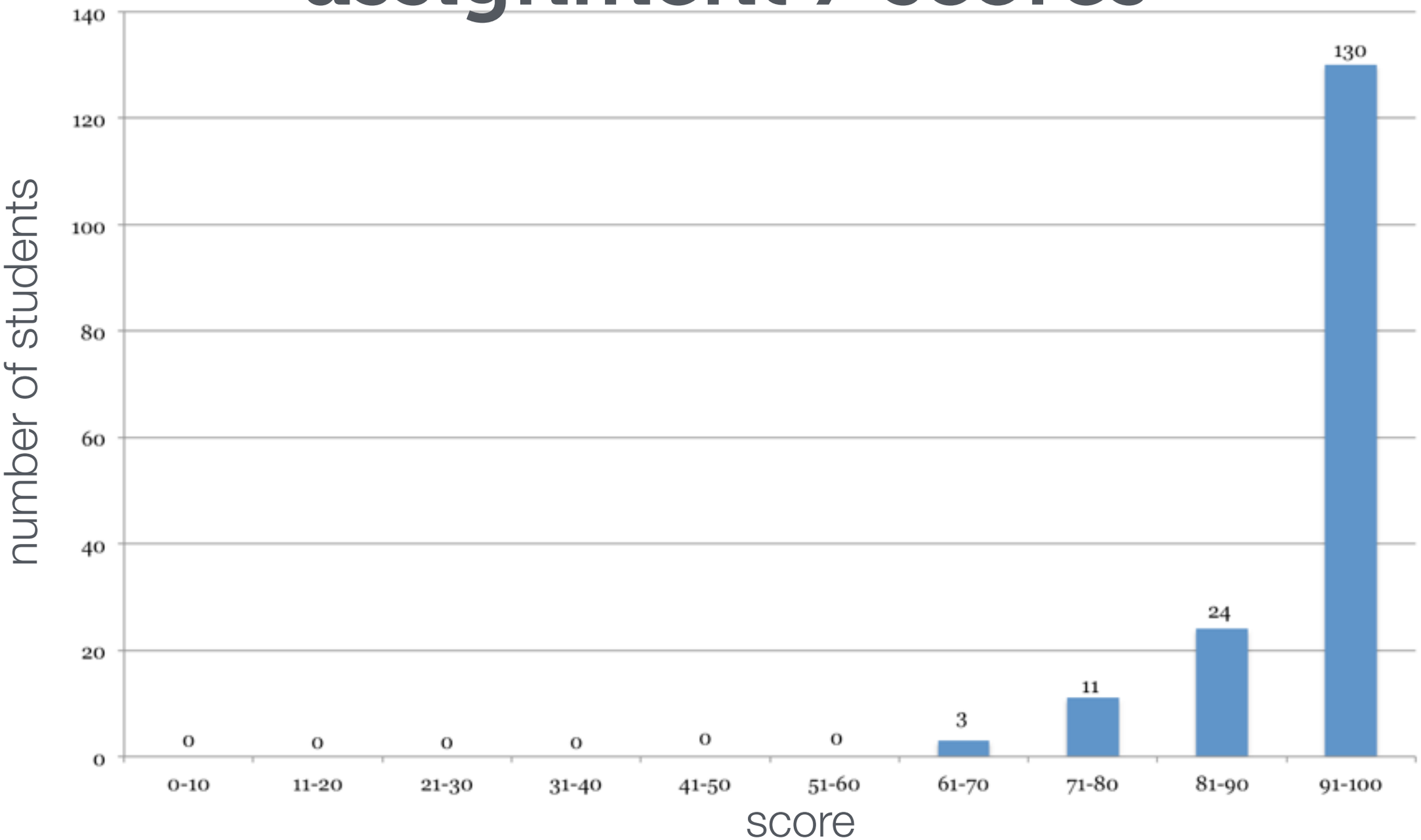
cs2420 | Introduction to Algorithms and Data Structures | Spring 2015

administrivia…

-assignment 10 is due tonight

-assignment 11 is up, due next Thursday

assignment 7 scores



let's chat about ethics...

```
// djb2 hash function
unsigned long hash(unsigned char *str)
{
    unsigned long hash = 5381;
    int c;

    while (c = *str++)
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

    return hash;
}
```

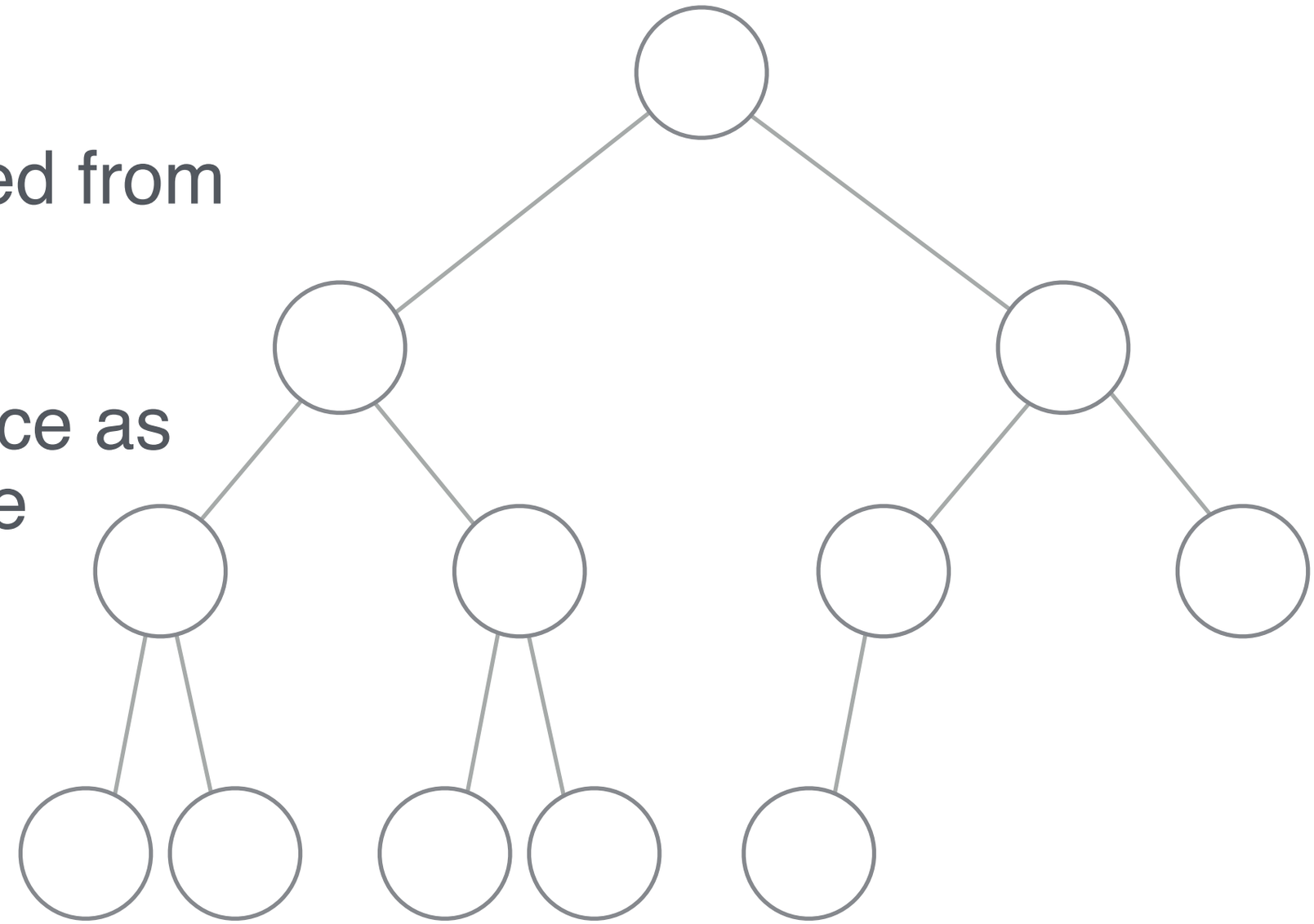
last time…

complete trees

-a **complete binary tree** has its levels completely filled, with the possible exception of the bottom level

-bottom level is filled from left to right

-each level has twice as many nodes as the previous level

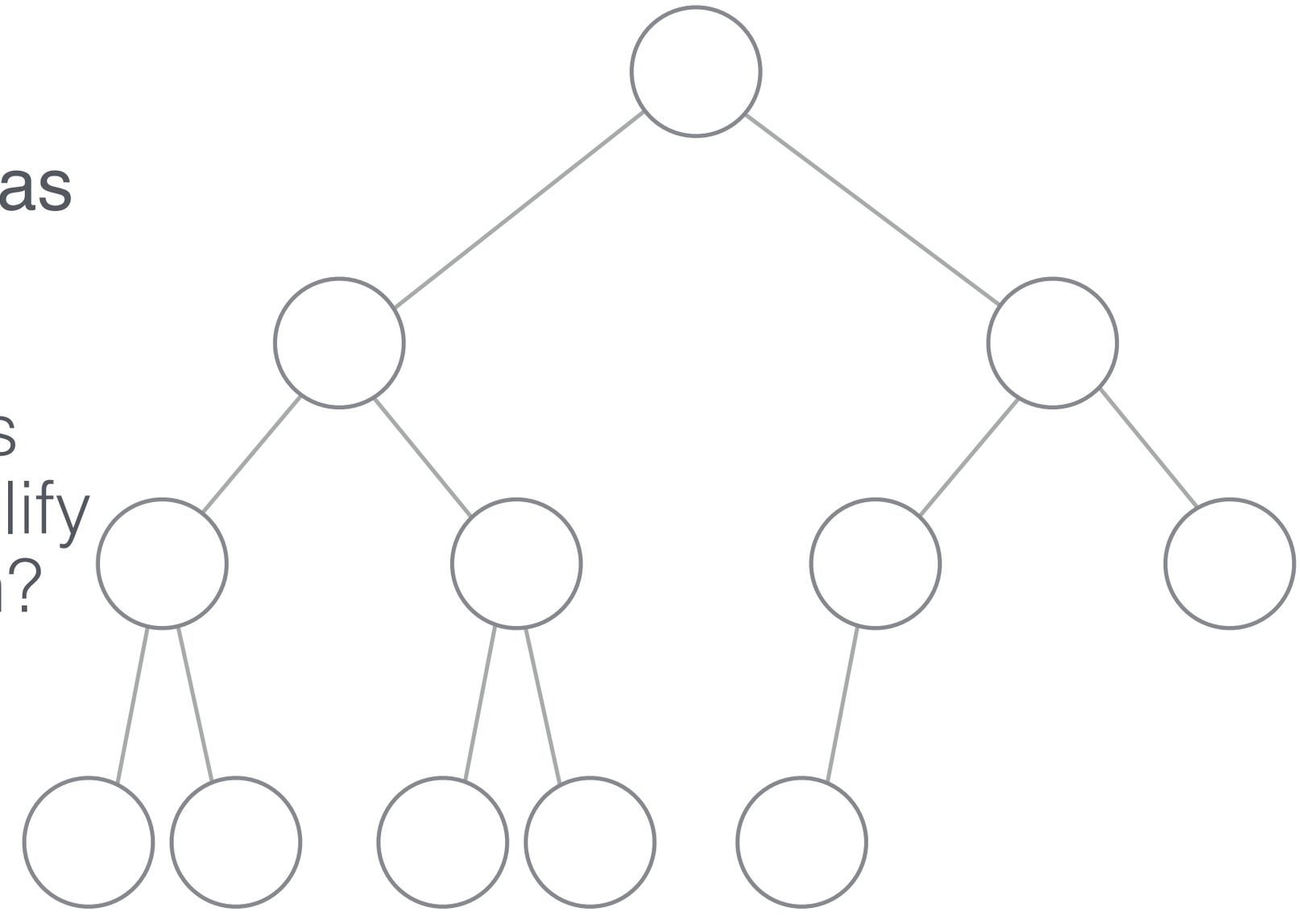


-a N-node complete tree has at most (logN) height

-operations are thus *at most* **O(logN)**

-each level has twice as many nodes as the previous one

  -how do we use this knowledge to simplify the implementation?

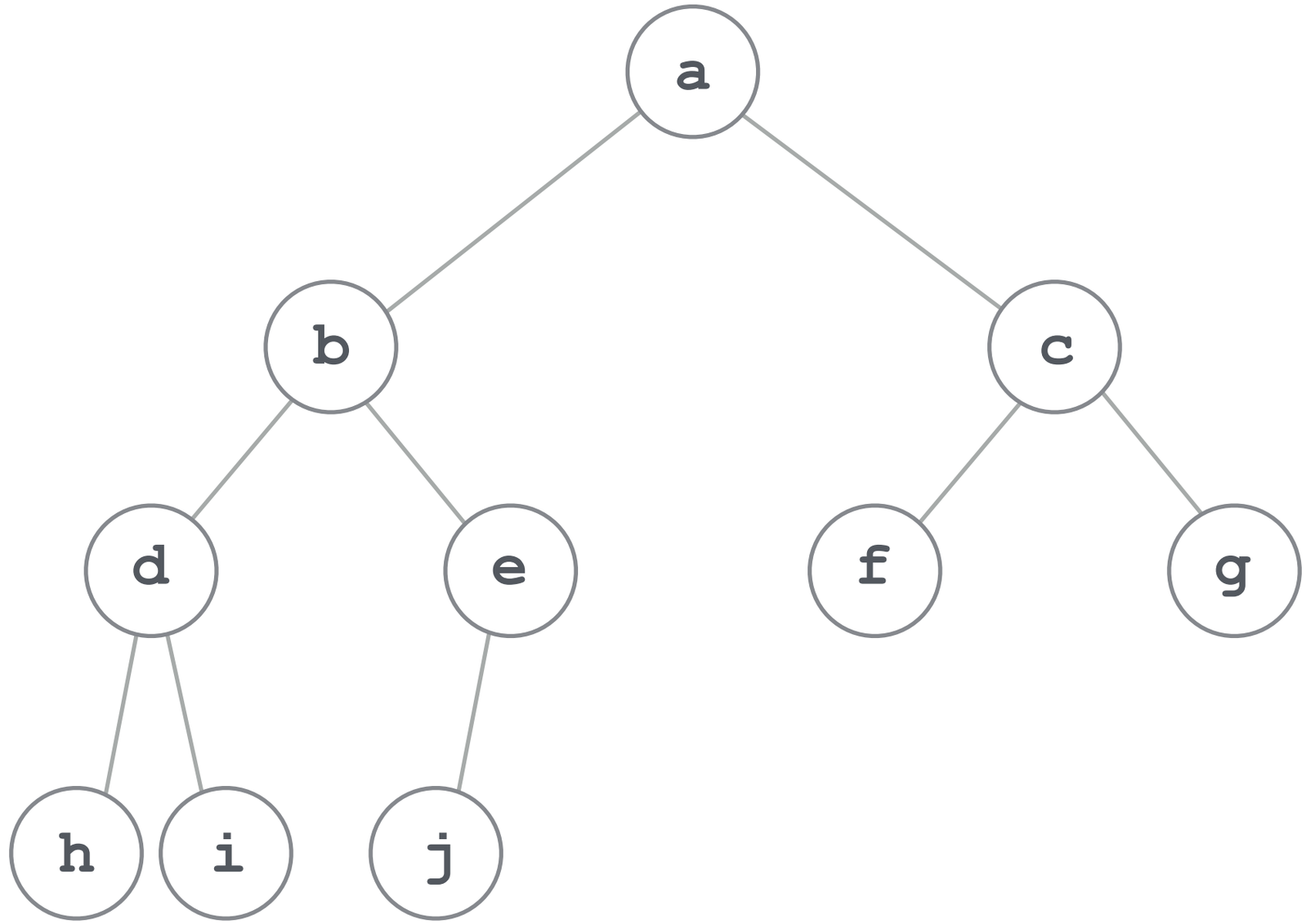


complete trees as an array

-if we are guaranteed that tree is complete, we can implement it as an array instead of a linked structure

-the root goes at index 0, its left child at index 1, its right child at index 2

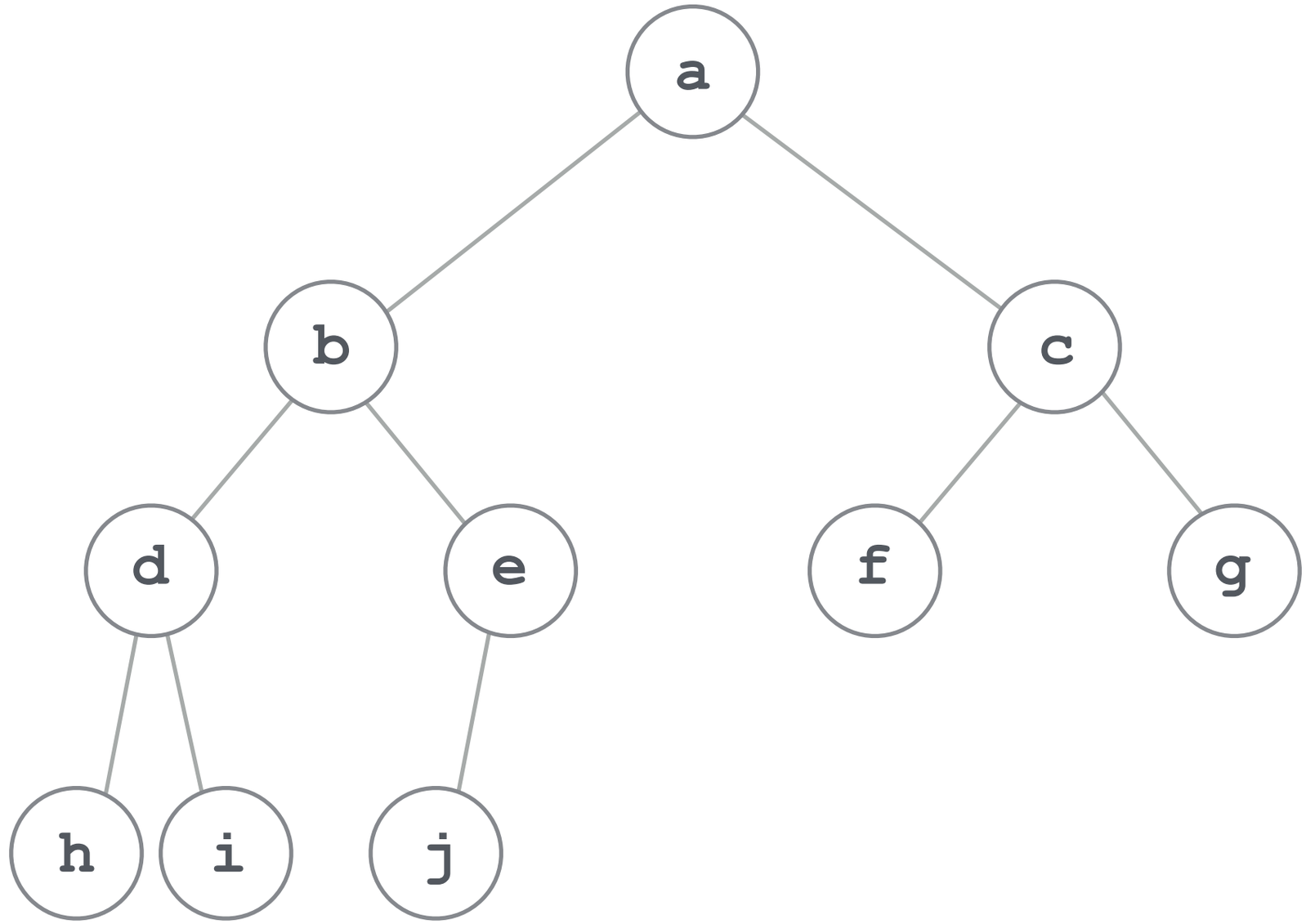
-for any node at index i, it two children are at index `(i*2) + 1` and `(i*2) + 2`



	a	b	c	d	e	f	g	h	i	j	
index:	0	1	2	3	4	5	6	7	8	9	10

-for example, d’s children start at (3*2) + 1

-how do we that f has no children?



	a	b	c	d	e	f	g	h	i	j	
index:	0	1	2	3	4	5	6	7	8	9	10

-any node’s parent is at index (i-1) / 2

remember the priority queue?

-a **priority queue** is a data structure in which access is limited to the minimum item in the set

- `-add`
- `-findMin`
- `-deleteMin`

-add location is unspecified, so long as the the above is always enforced

-what are our options for implementing this?

binary heap

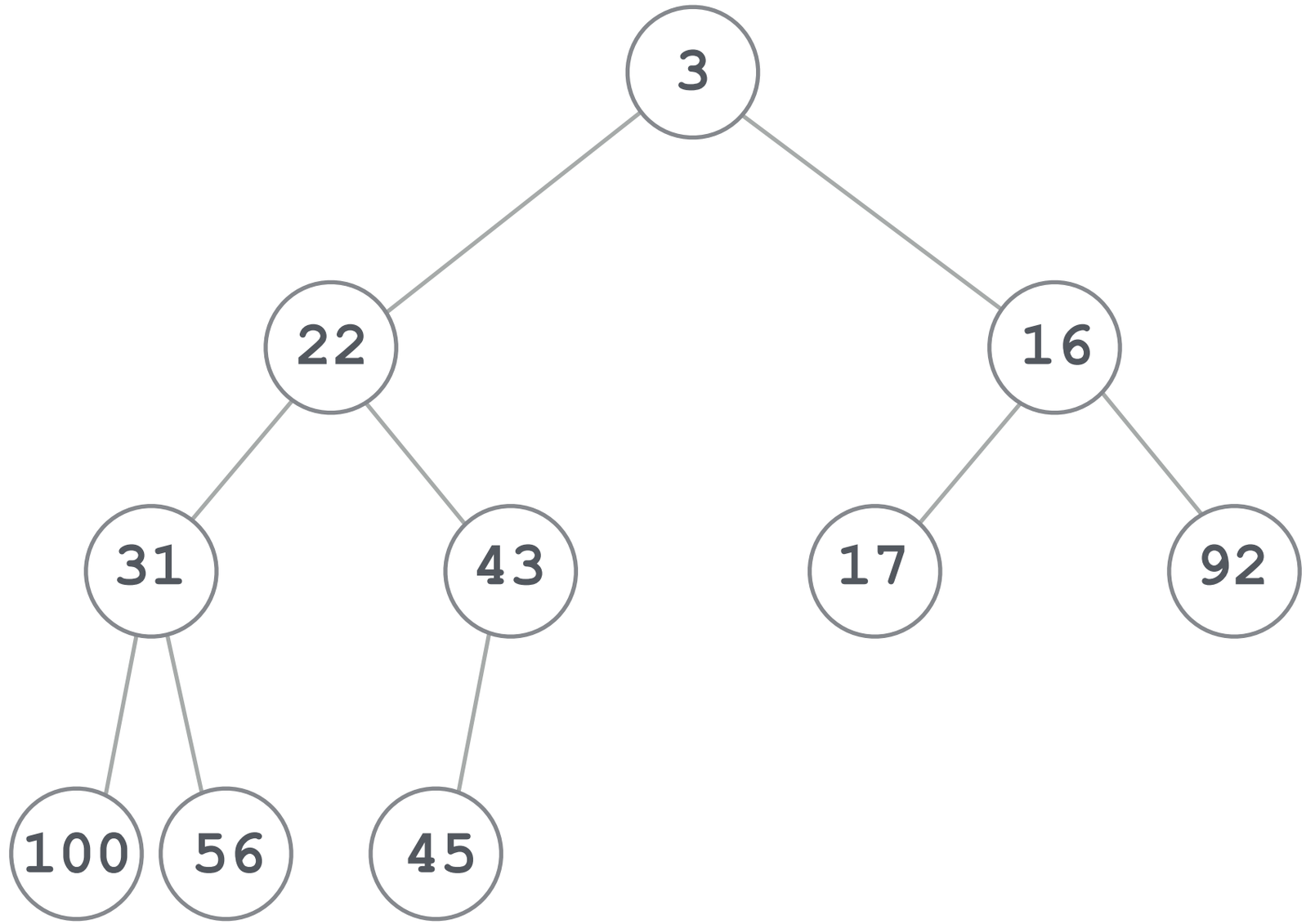
-a **binary heap** is a binary tree with two special properties

-*structure:* it is a complete tree

-*order:* the data in any node is less than or equal to the data of its children

-this is also called a **min-heap**

-a **max-heap** would have the opposite property



-where is the smallest item?

adding to a heap

-we must be careful to maintain the two properties when adding to a heap

  -structure and order

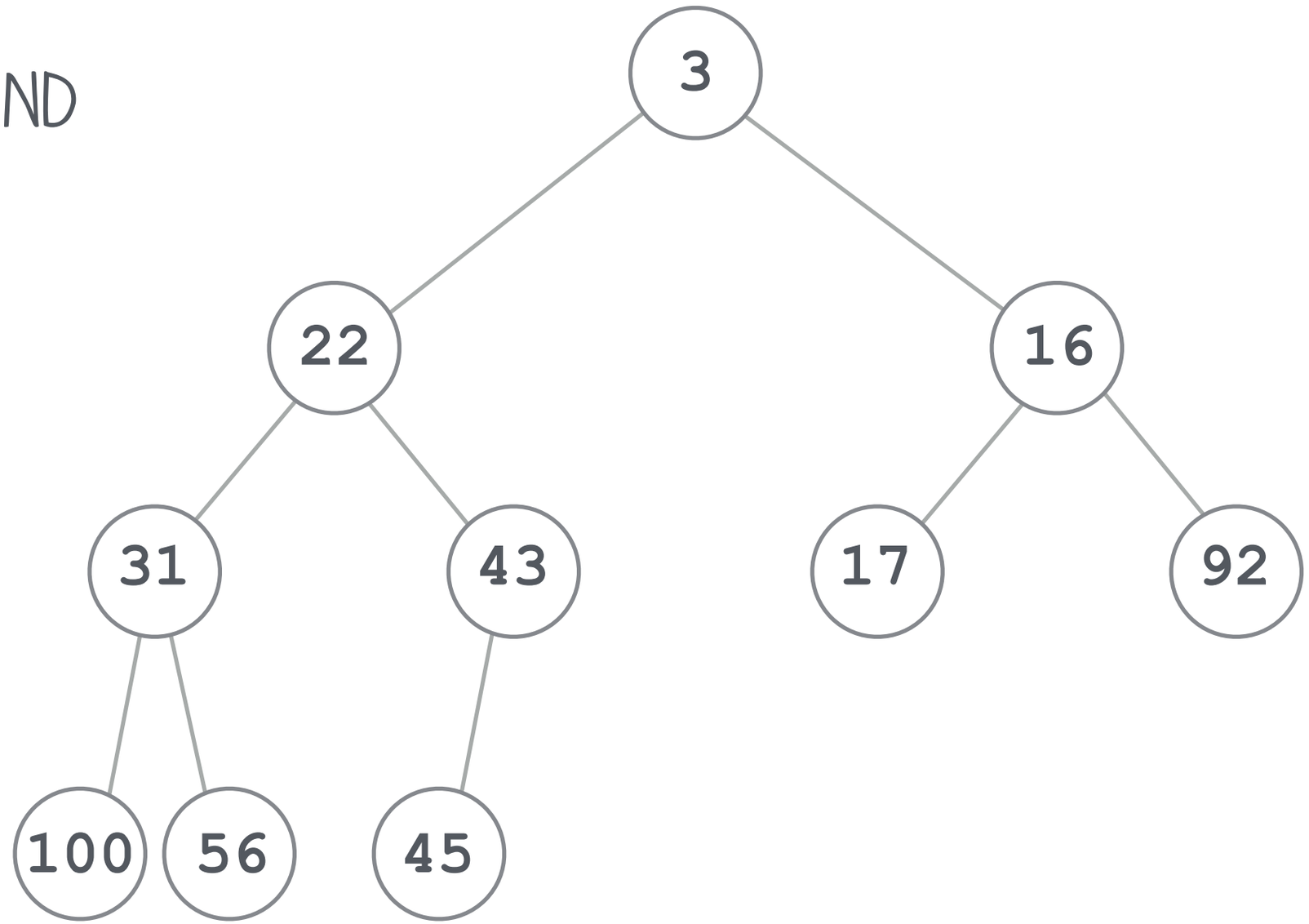
-deal with the structure property first… where can the new item go to maintain a complete tree?

-then, *percolate* the item upward until the order property is restored

  -swap upwards until > parent

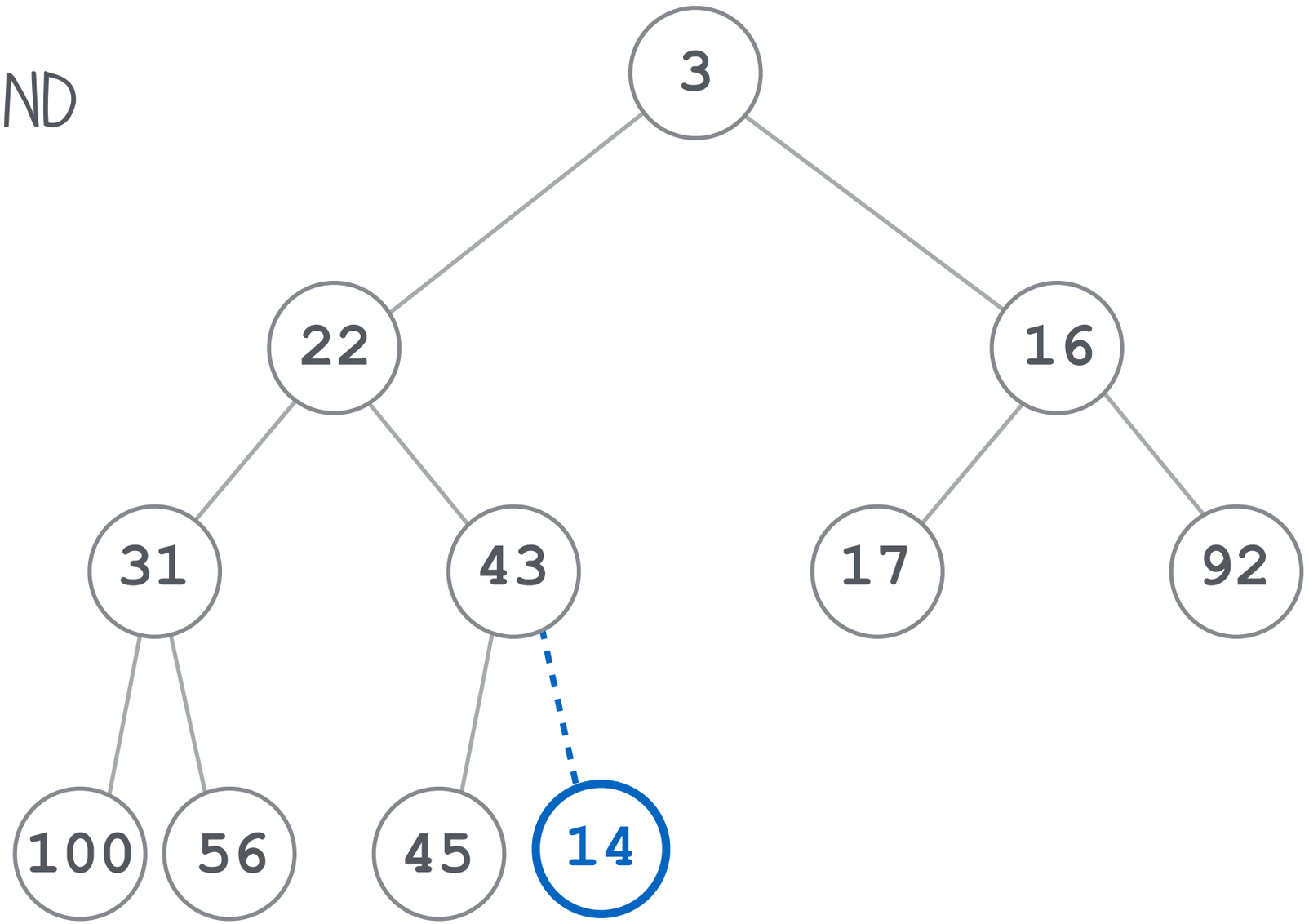
ADDING 14

PUT IT AT THE END
OF THE TREE



ADDING 14

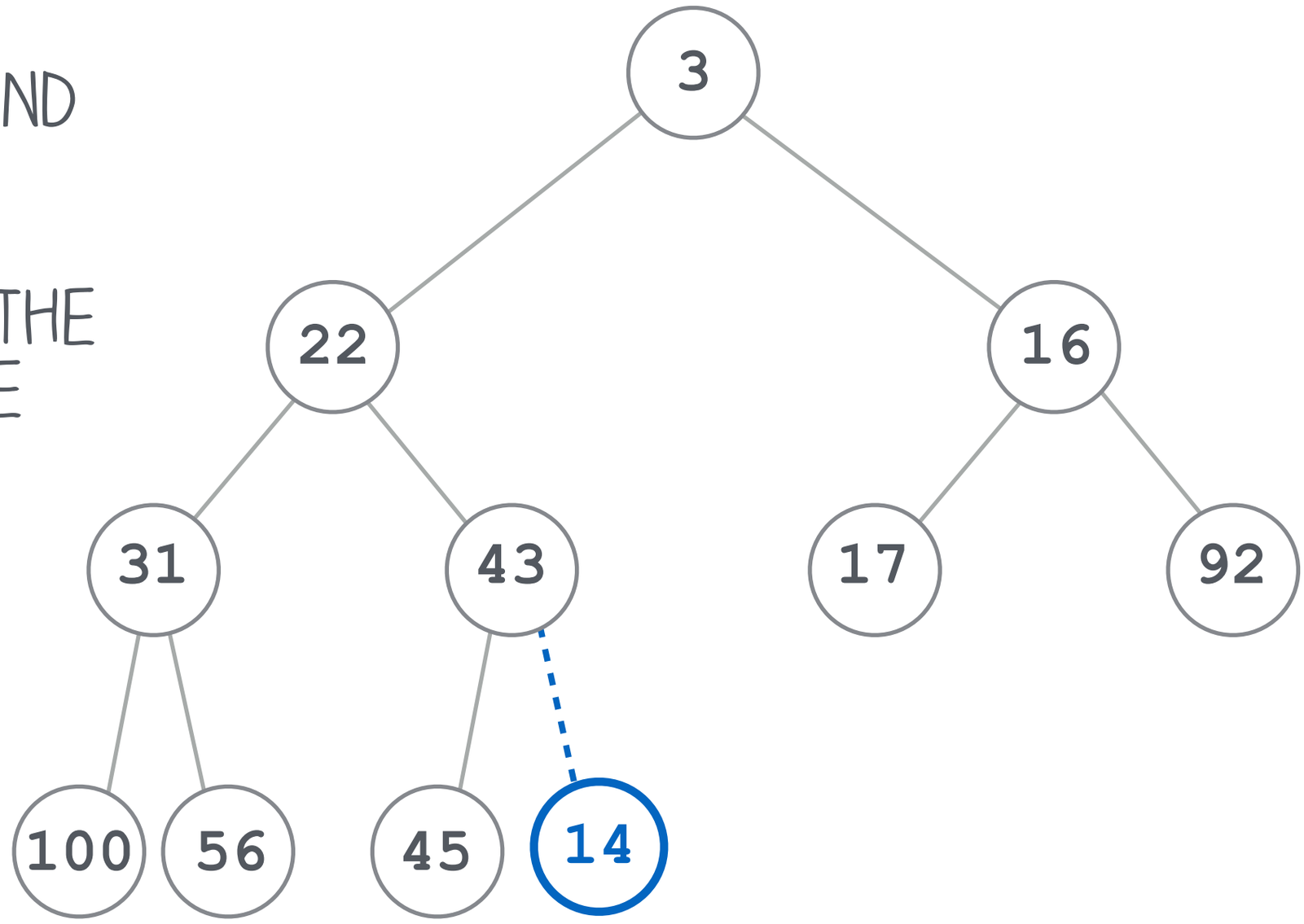
PUT IT AT THE END
OF THE TREE



ADDING 14

PUT IT AT THE END OF THE TREE

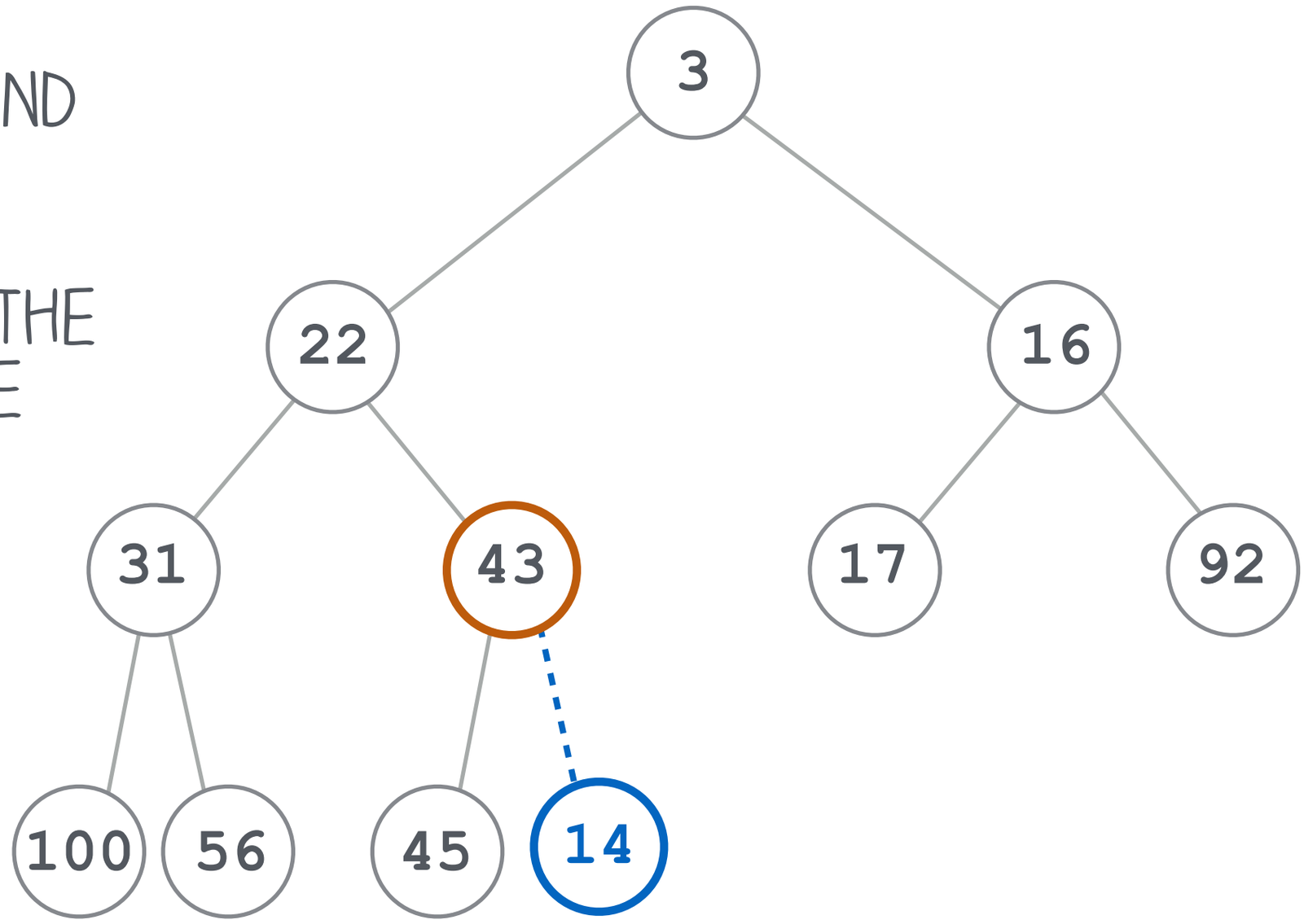
PERCOLATE UP THE TREE TO FIX THE ORDER



ADDING 14

PUT IT AT THE END OF THE TREE

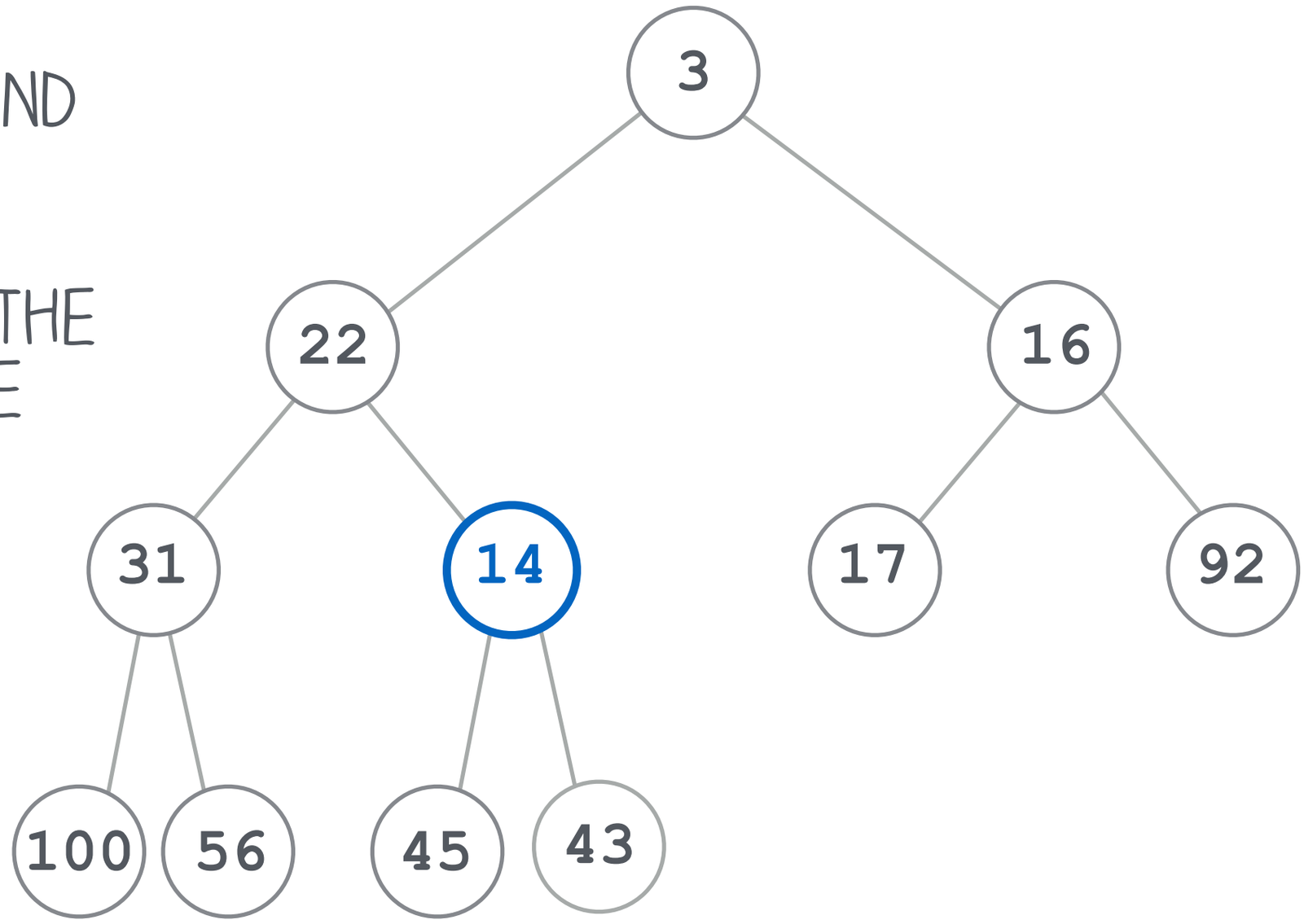
PERCOLATE UP THE TREE TO FIX THE ORDER



ADDING 14

PUT IT AT THE END OF THE TREE

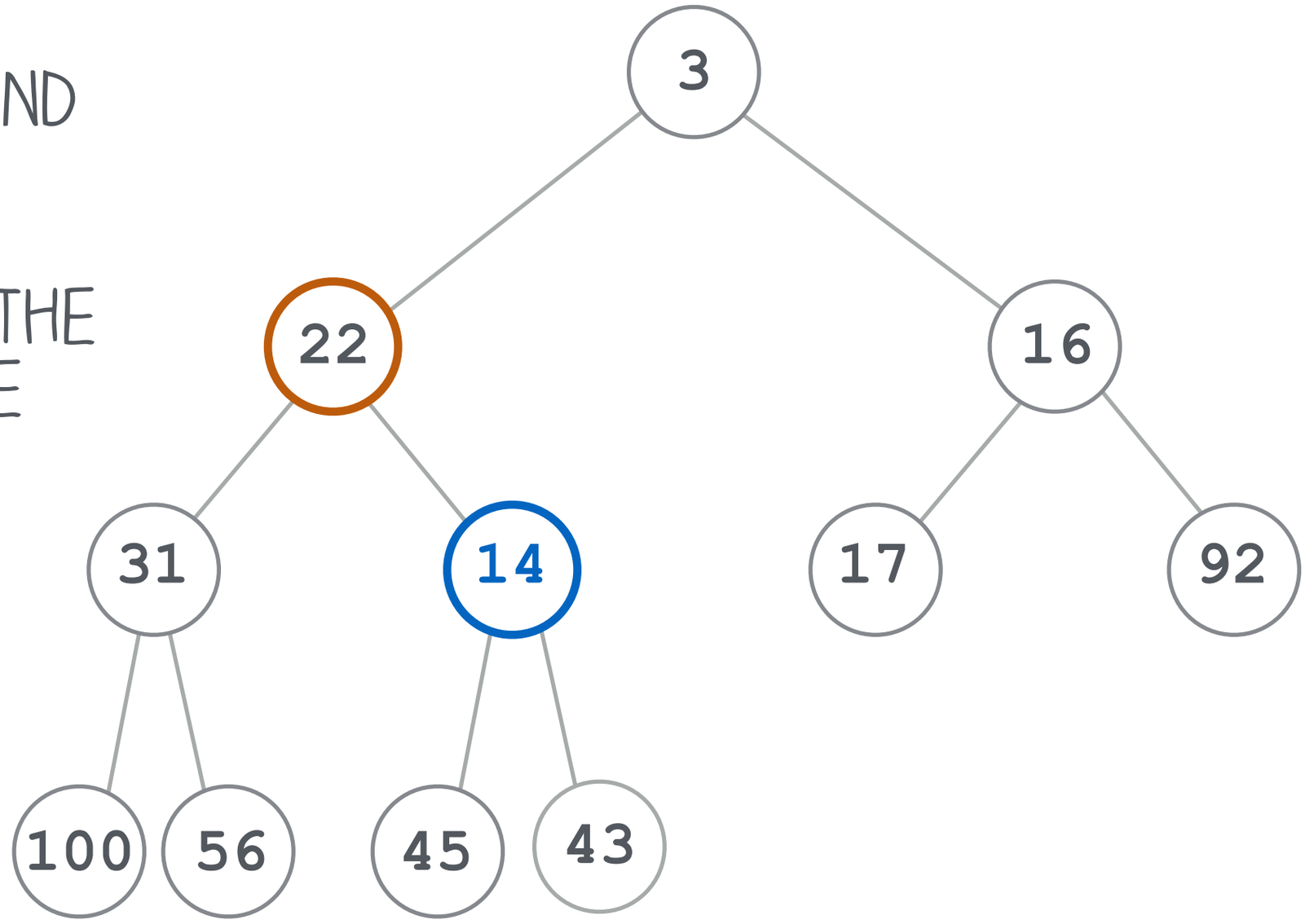
PERCOLATE UP THE TREE TO FIX THE ORDER



ADDING 14

PUT IT AT THE END OF THE TREE

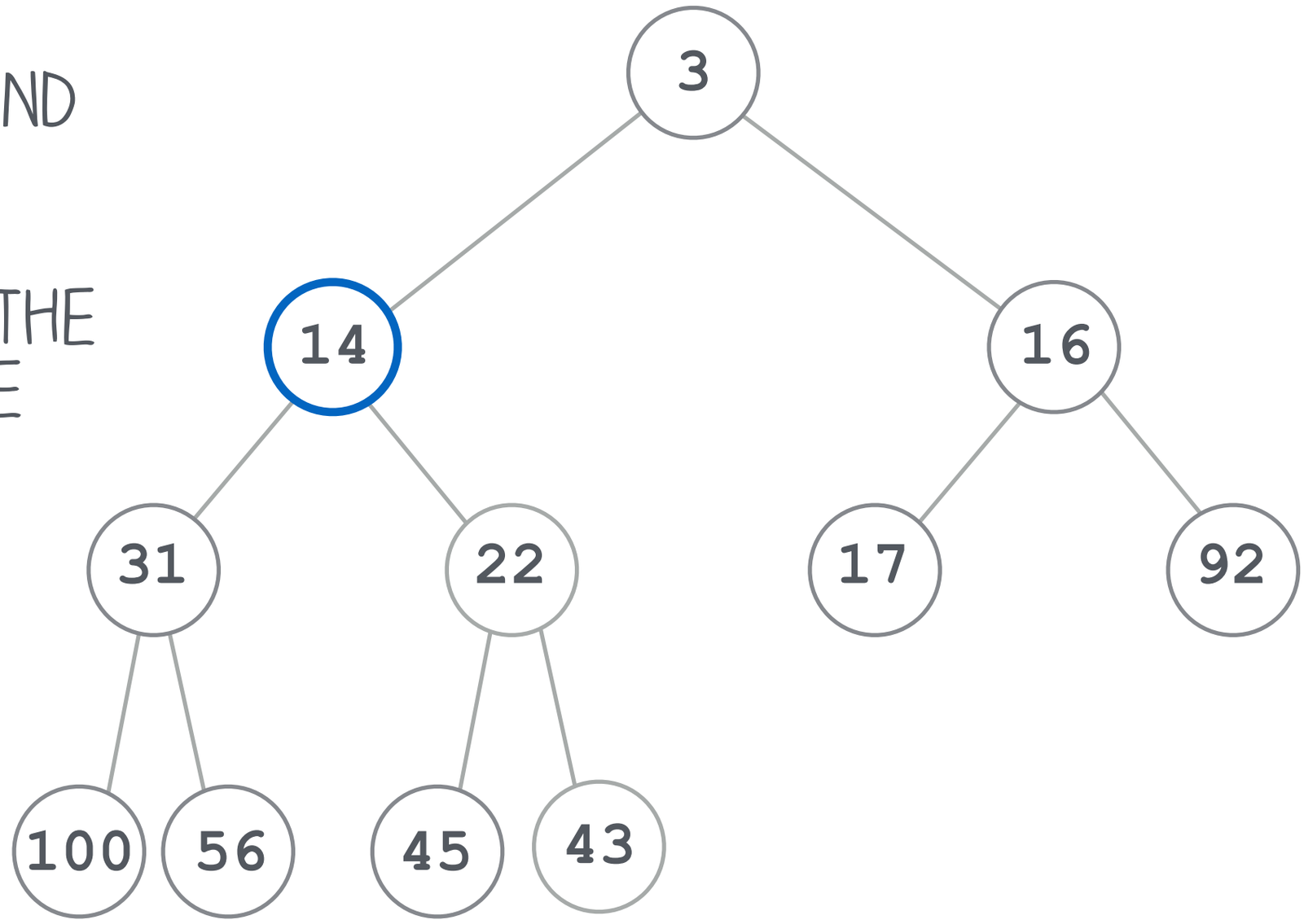
PERCOLATE UP THE TREE TO FIX THE ORDER



ADDING 14

PUT IT AT THE END OF THE TREE

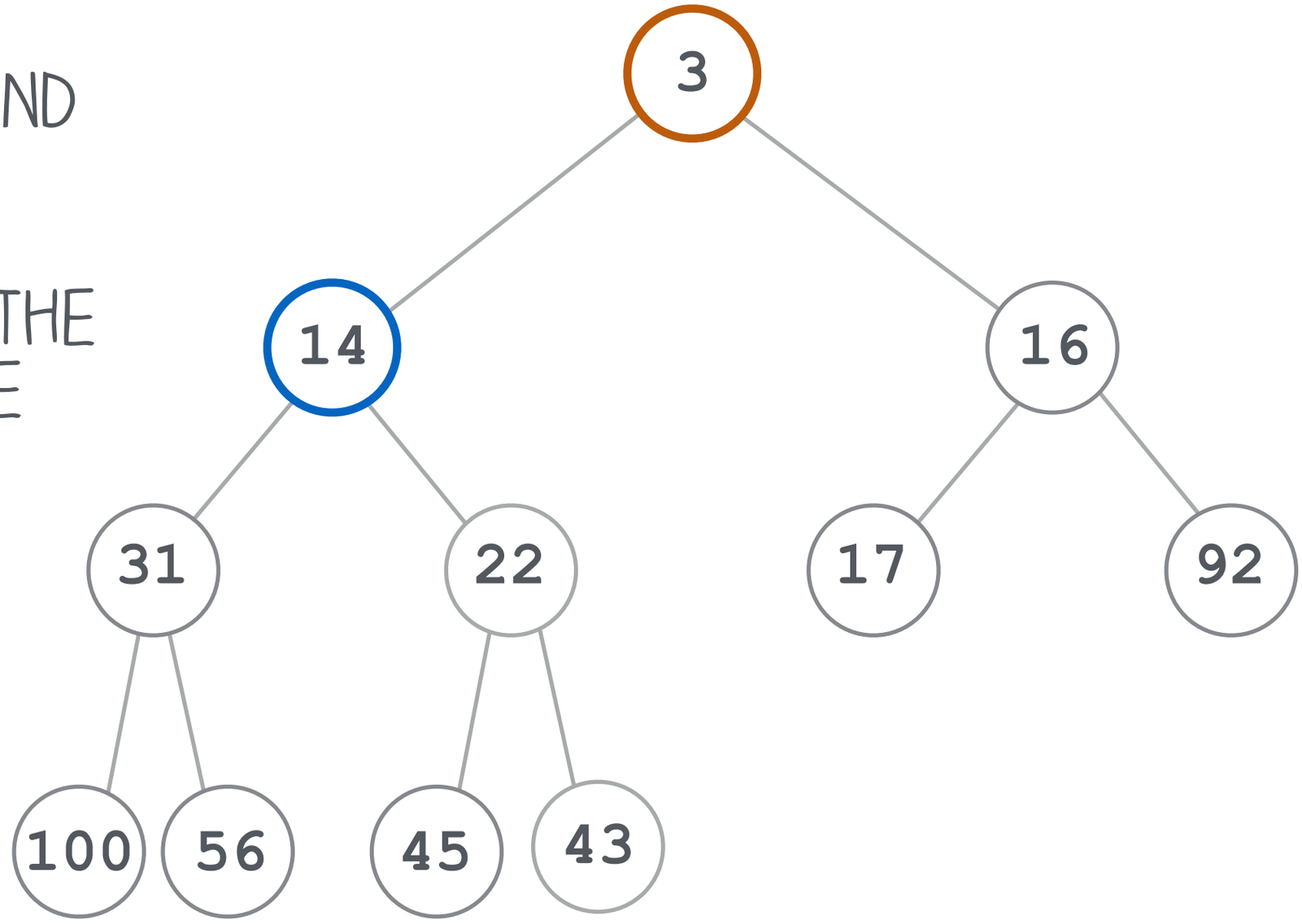
PERCOLATE UP THE TREE TO FIX THE ORDER



ADDING 14

PUT IT AT THE END OF THE TREE

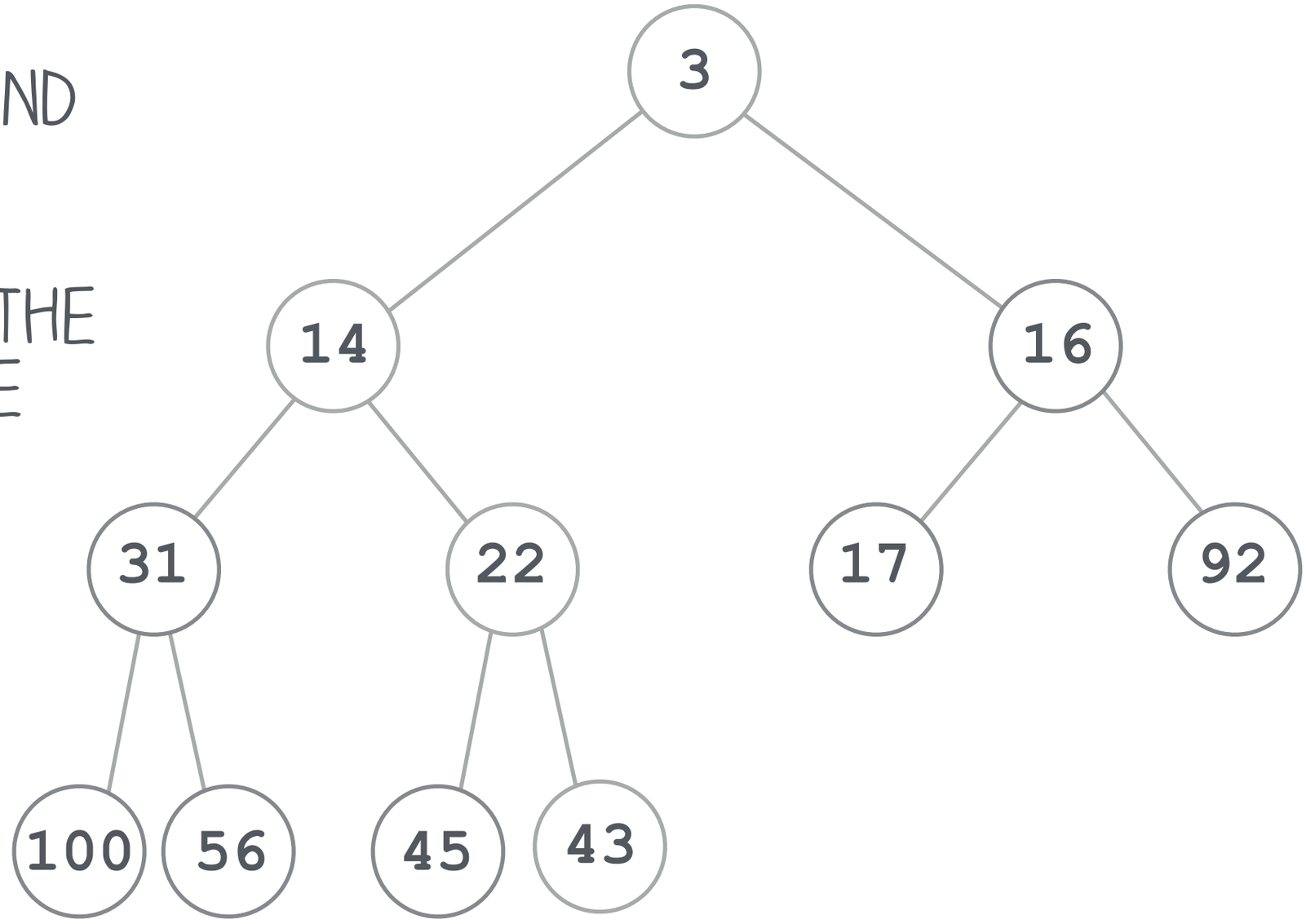
PERCOLATE UP THE TREE TO FIX THE ORDER



ADDING 14

PUT IT AT THE END OF THE TREE

PERCOLATE UP THE TREE TO FIX THE ORDER



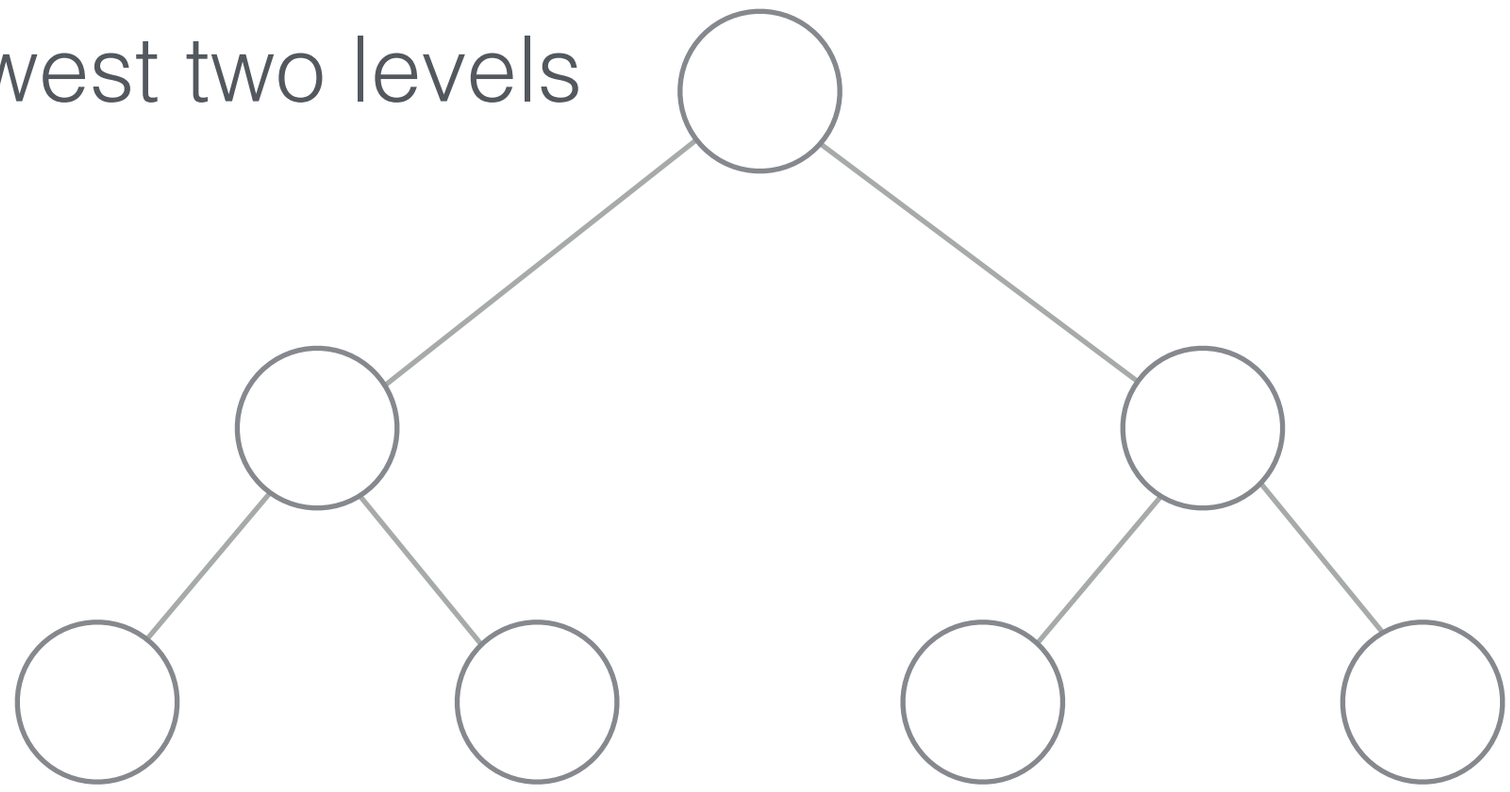
cost of add

-percolate up until smaller than all nodes below it…

-how many nodes are there on each level (in terms of N)?

-about half on the lowest level

-about 3/4 in the lowest two levels



-if the new item is the smallest in the set, cost is **O(logN)**

-must percolate up every level to the root

-complete trees have logN levels

-is this the worst, average, or best case?

-it has been shown that on average, 1.6 comparisons are needed for any N

-thus, add terminates early, and average cost is **O(1)**

-you may find conflicting information on the average cost value

-some sources may say O(logN)

-it depends on the level of the analysis and how tight of a bound we want

-don’t always believe what you read, test for yourself!

remove from a heap

-priority queues only support removing the smallest item

-in heap this is always what?

-remove and return the root

-we have a hole at the top, structure property is violated

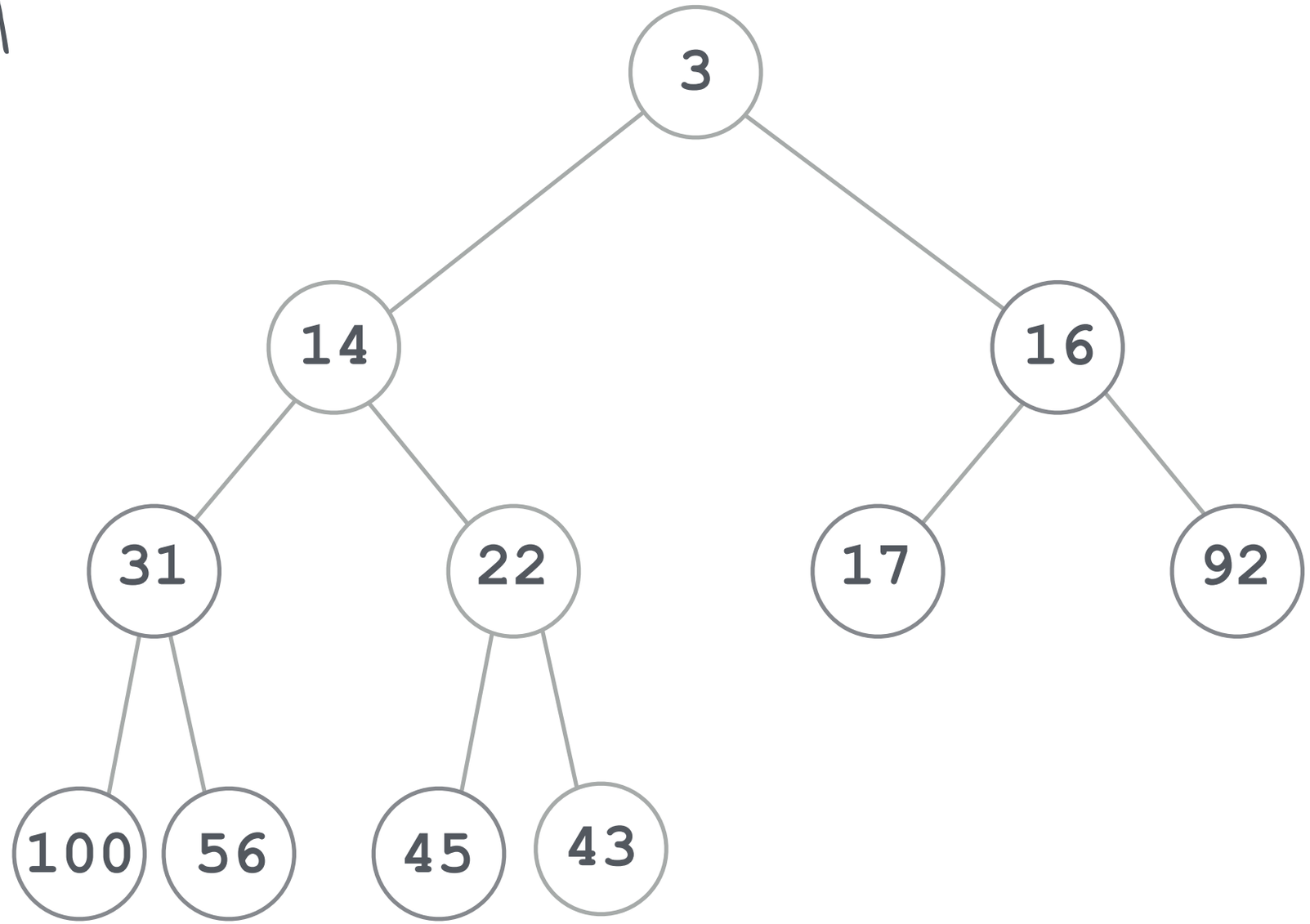
-fill the whole with another item in the tree

-which one?

-percolate down

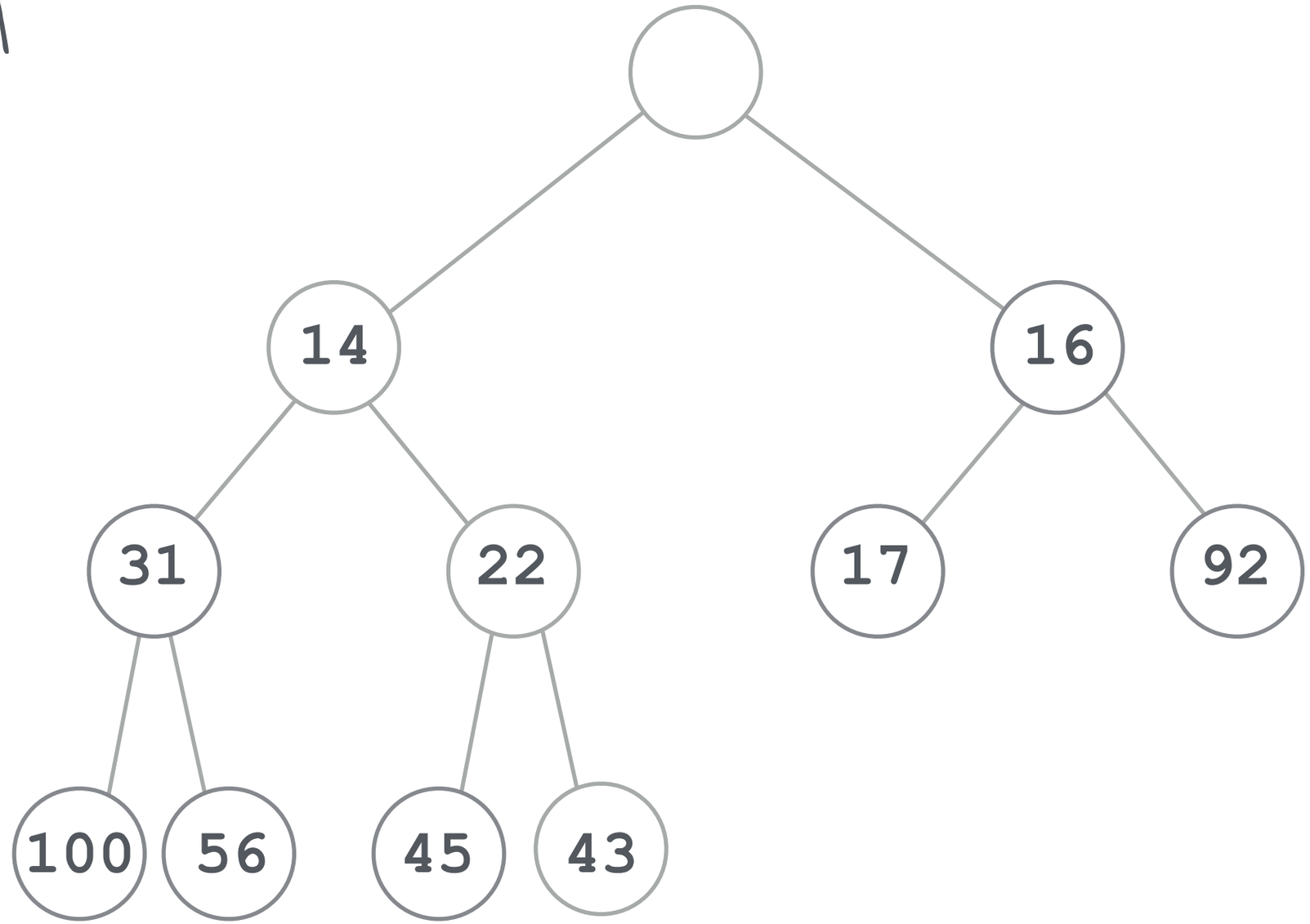
LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3



LET'S REMOVE THE SMALLEST ITEM

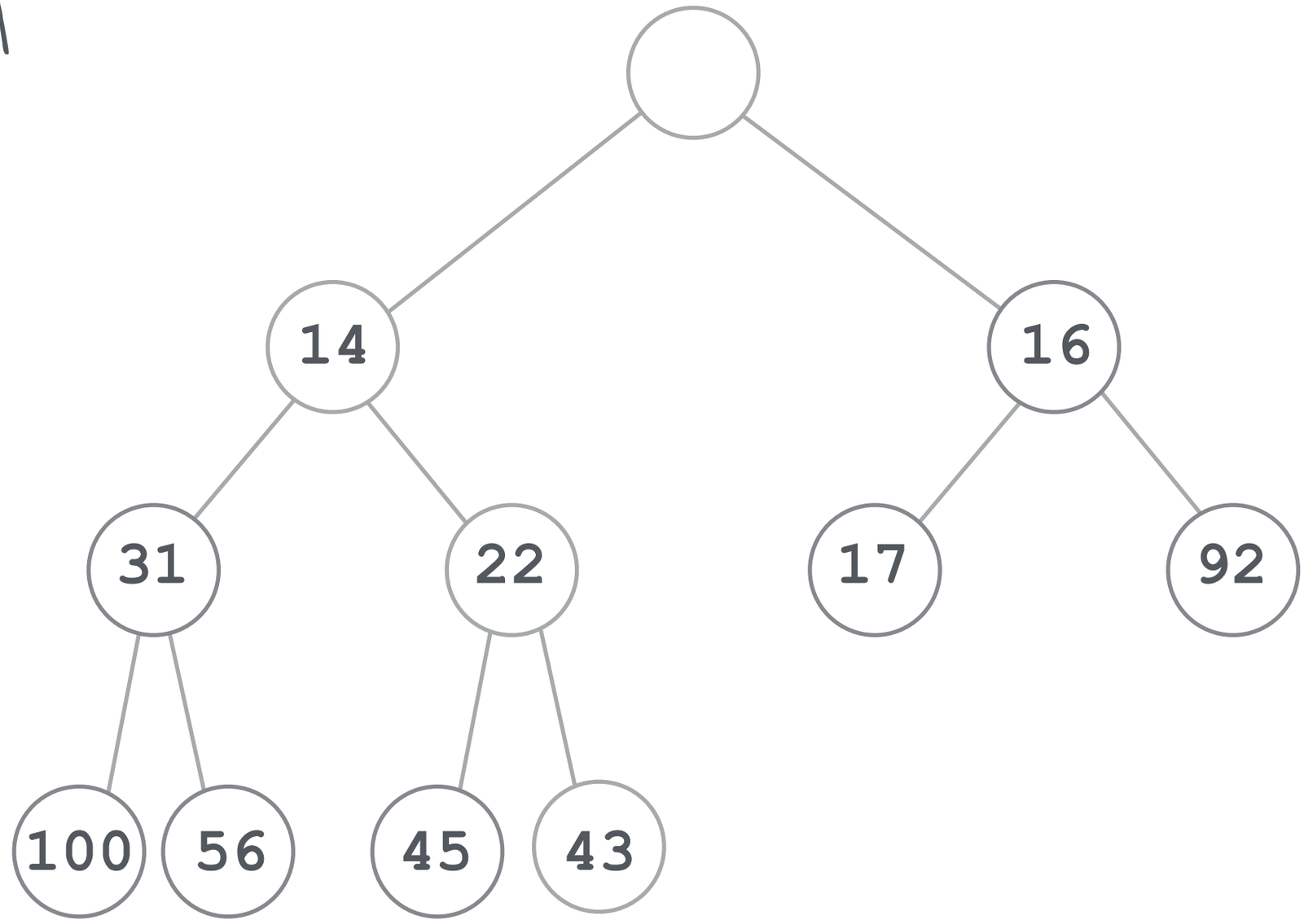
TAKE OUT 3



LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3

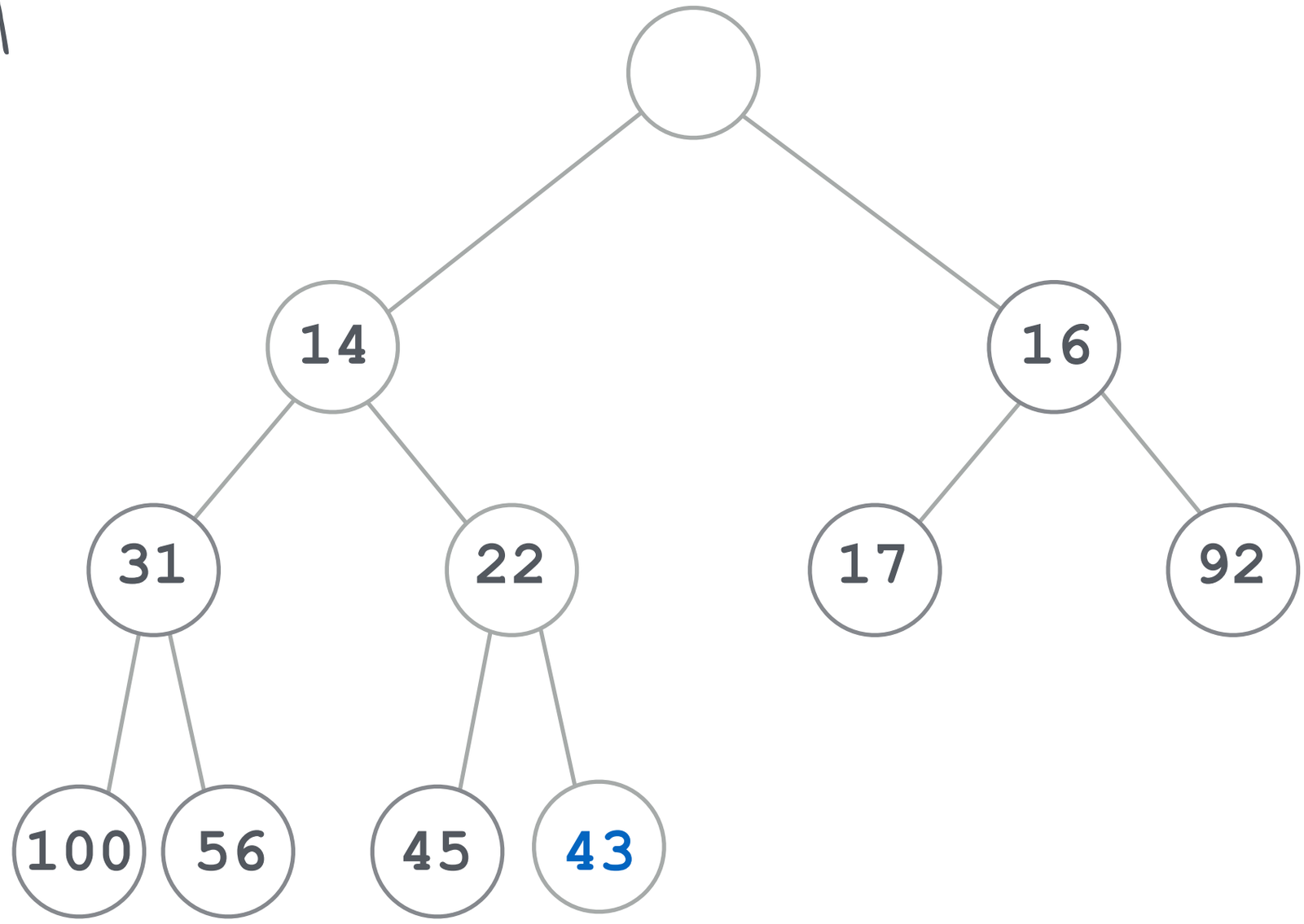
FILL WITH LAST ITEM ON LAST LEVEL. WHY?



LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3

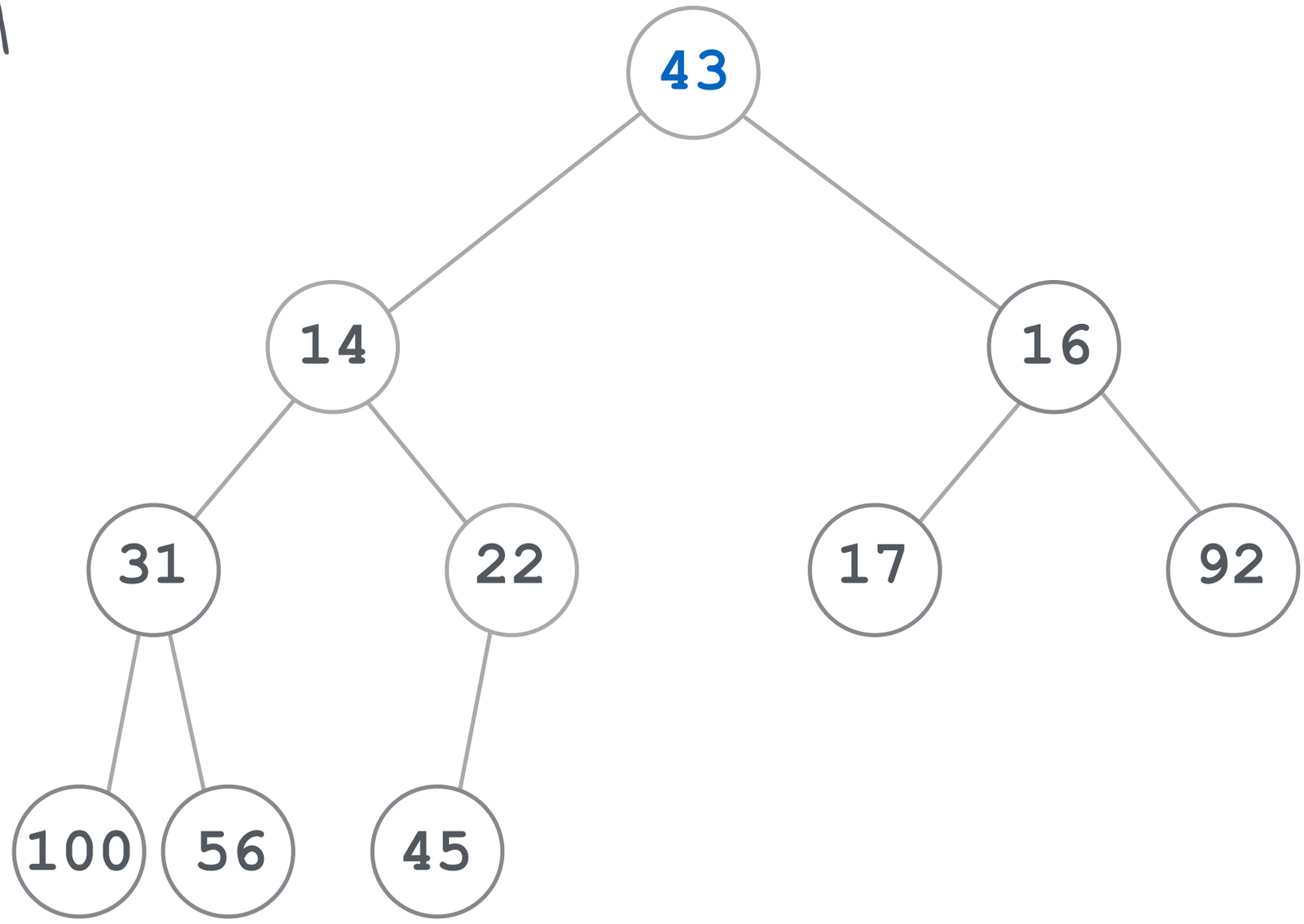
FILL WITH LAST ITEM ON LAST LEVEL. WHY?



LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3

FILL WITH LAST ITEM ON LAST LEVEL. WHY?

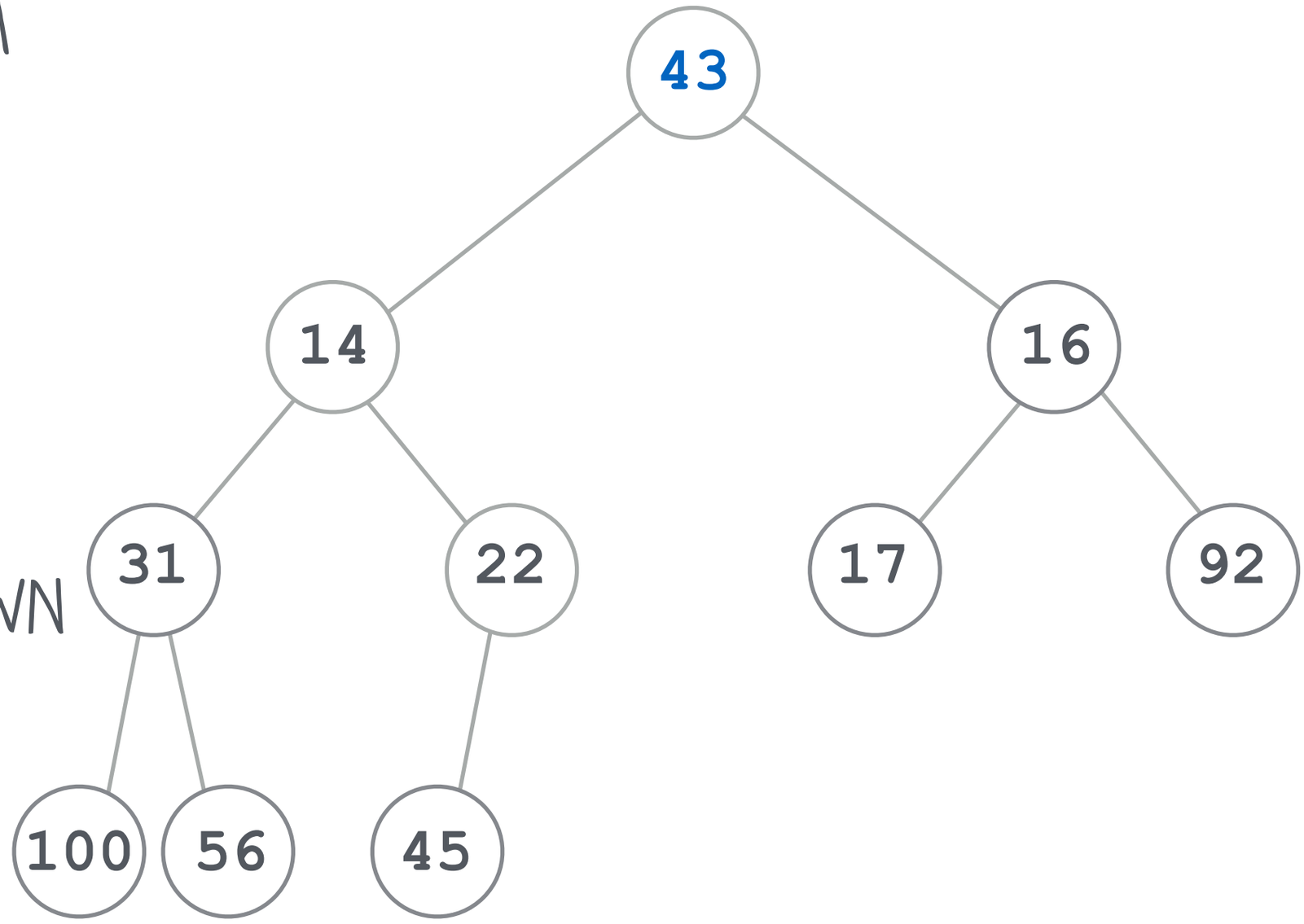


LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3

FILL WITH LAST ITEM ON LAST LEVEL. WHY?

PERCOLATE DOWN

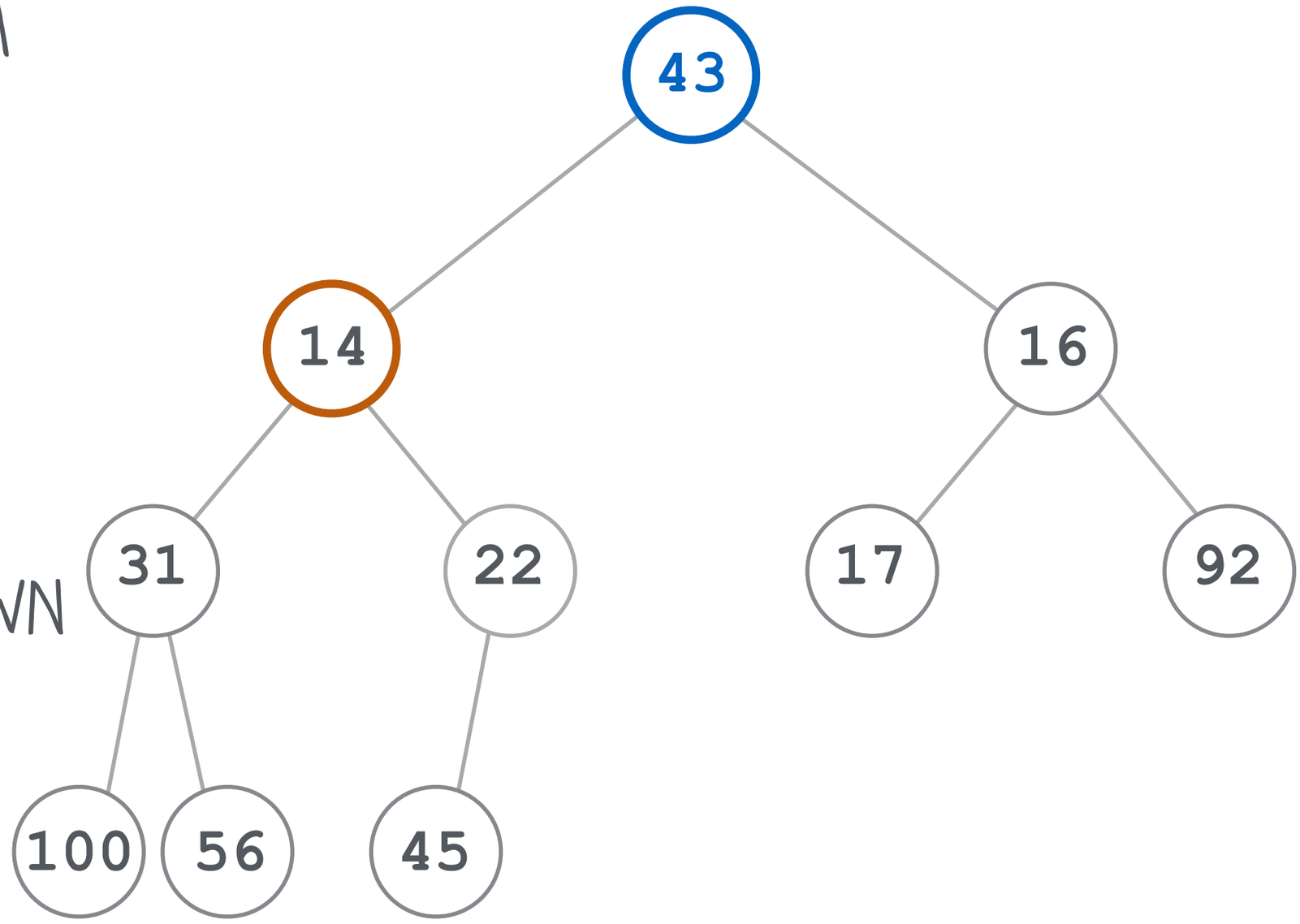


LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3

FILL WITH LAST ITEM ON LAST LEVEL. WHY?

PERCOLATE DOWN

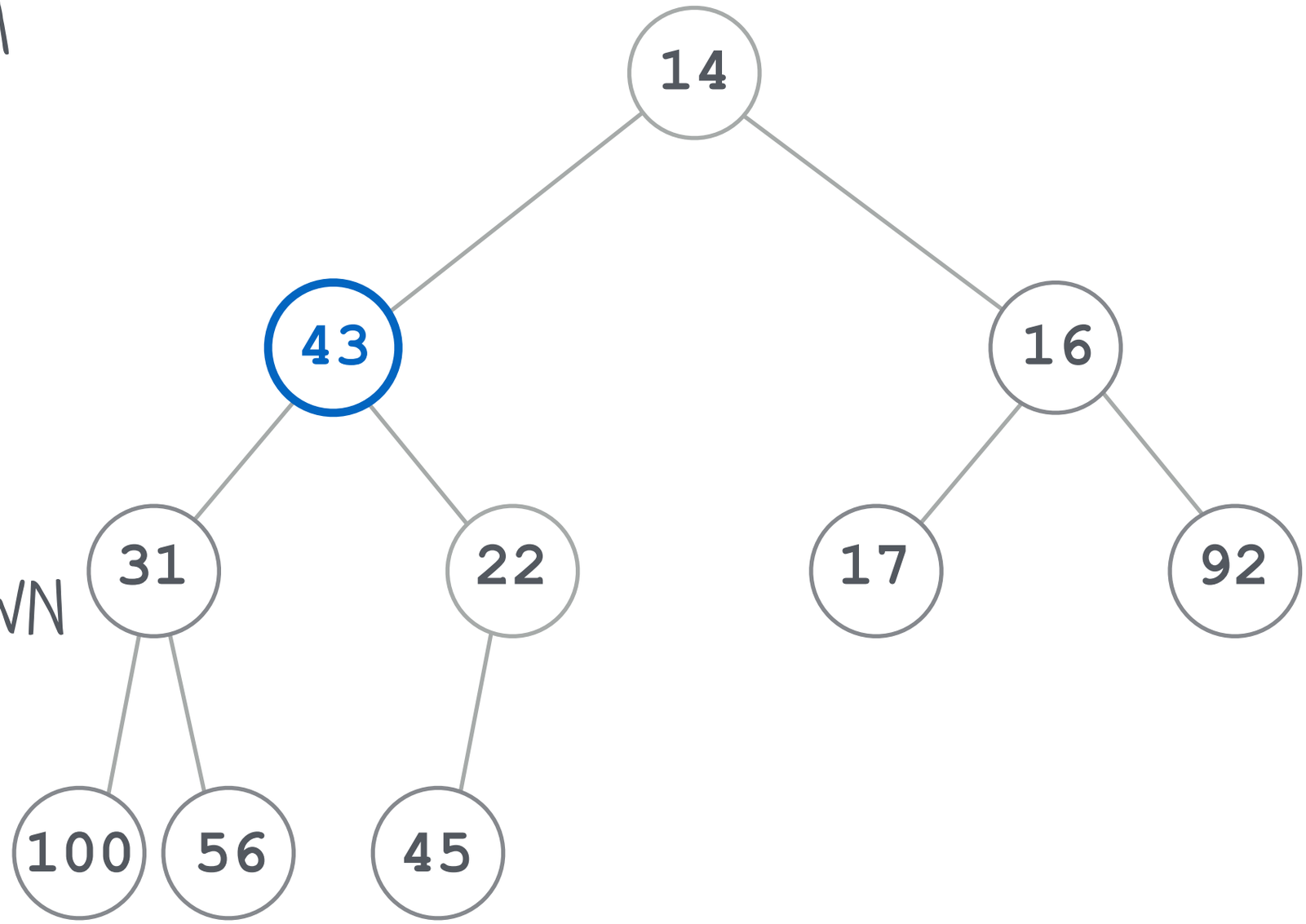


LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3

FILL WITH LAST ITEM ON LAST LEVEL. WHY?

PERCOLATE DOWN

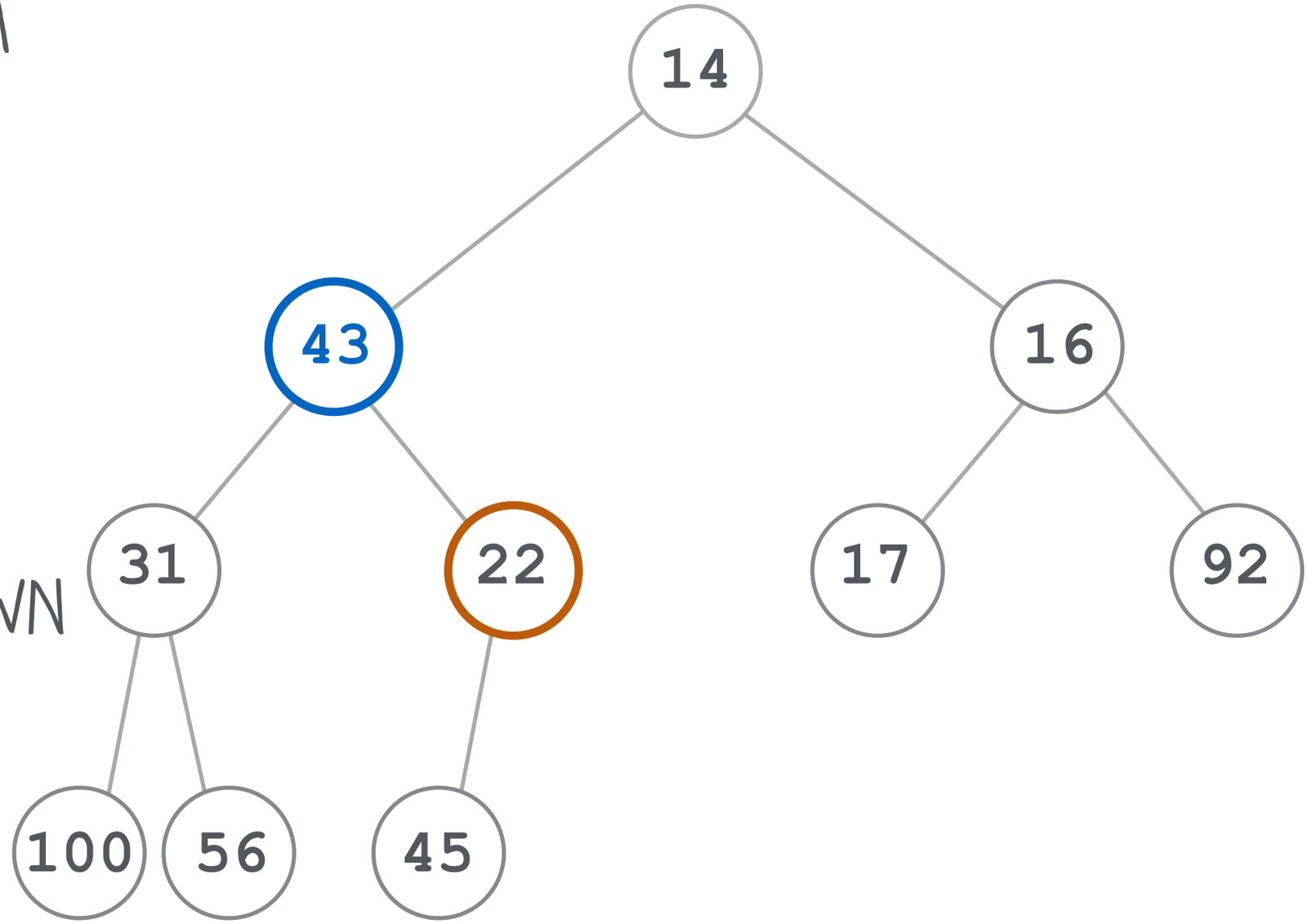


LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3

FILL WITH LAST ITEM ON LAST LEVEL. WHY?

PERCOLATE DOWN

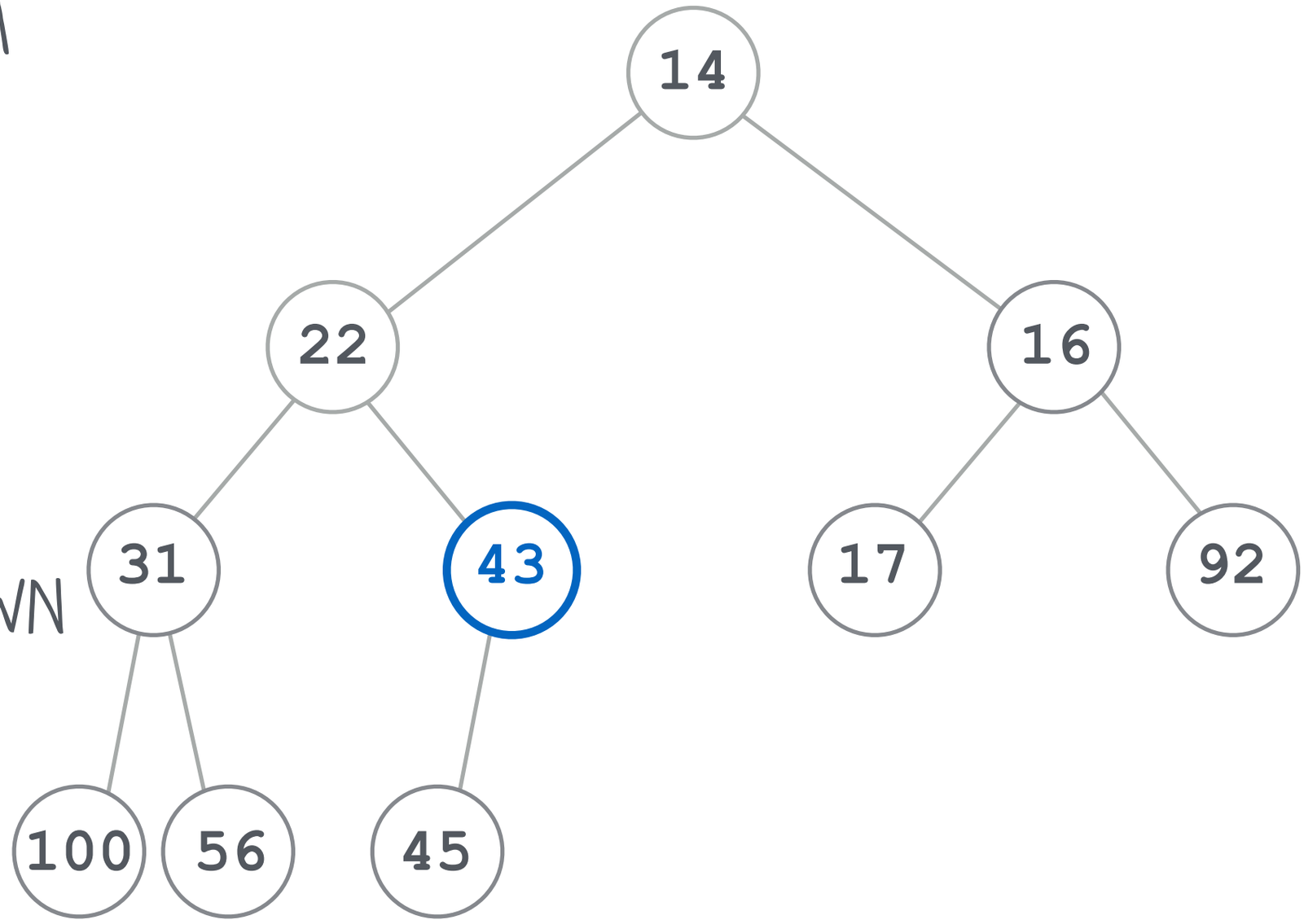


LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3

FILL WITH LAST ITEM ON LAST LEVEL. WHY?

PERCOLATE DOWN

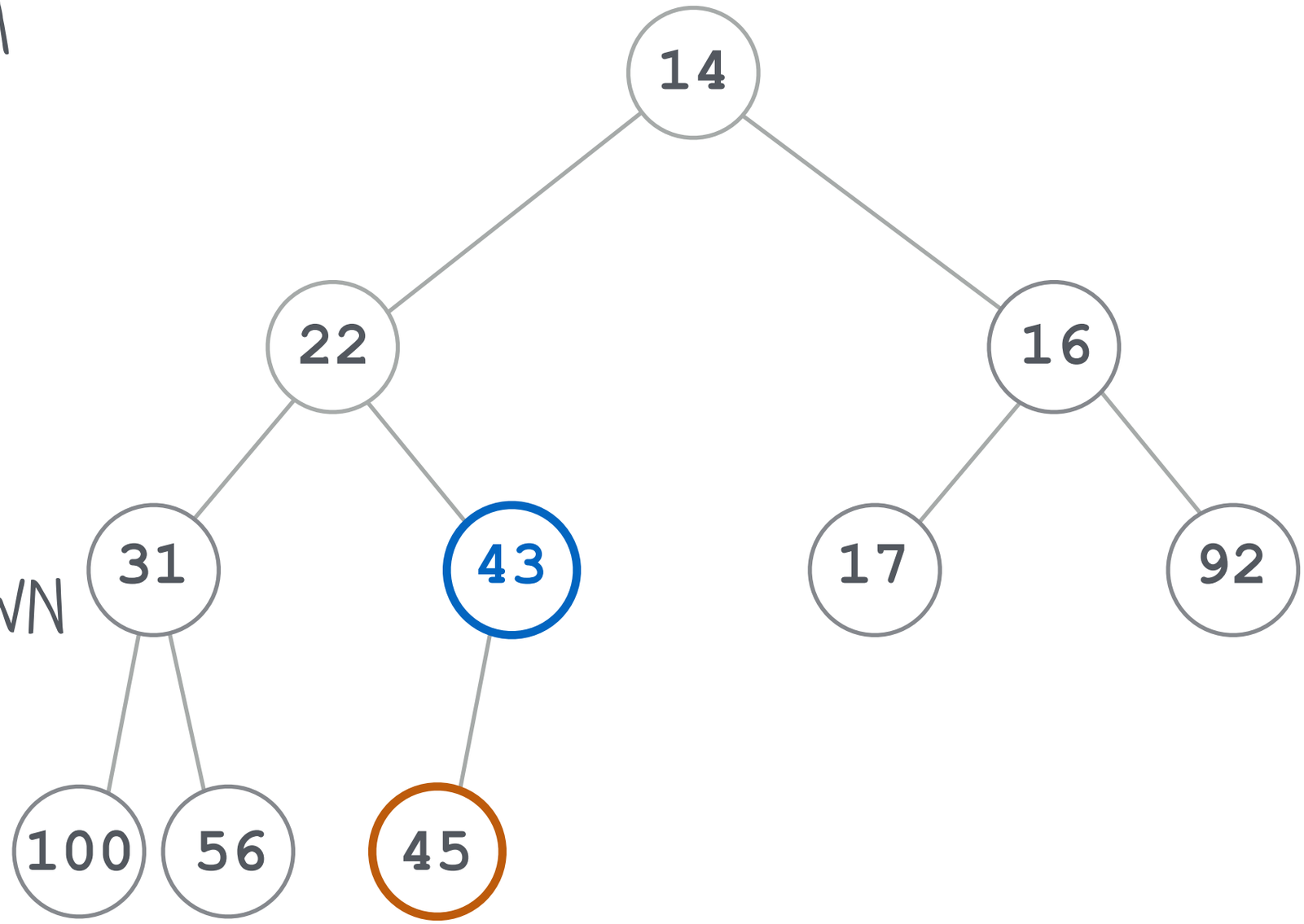


LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3

FILL WITH LAST ITEM ON LAST LEVEL. WHY?

PERCOLATE DOWN

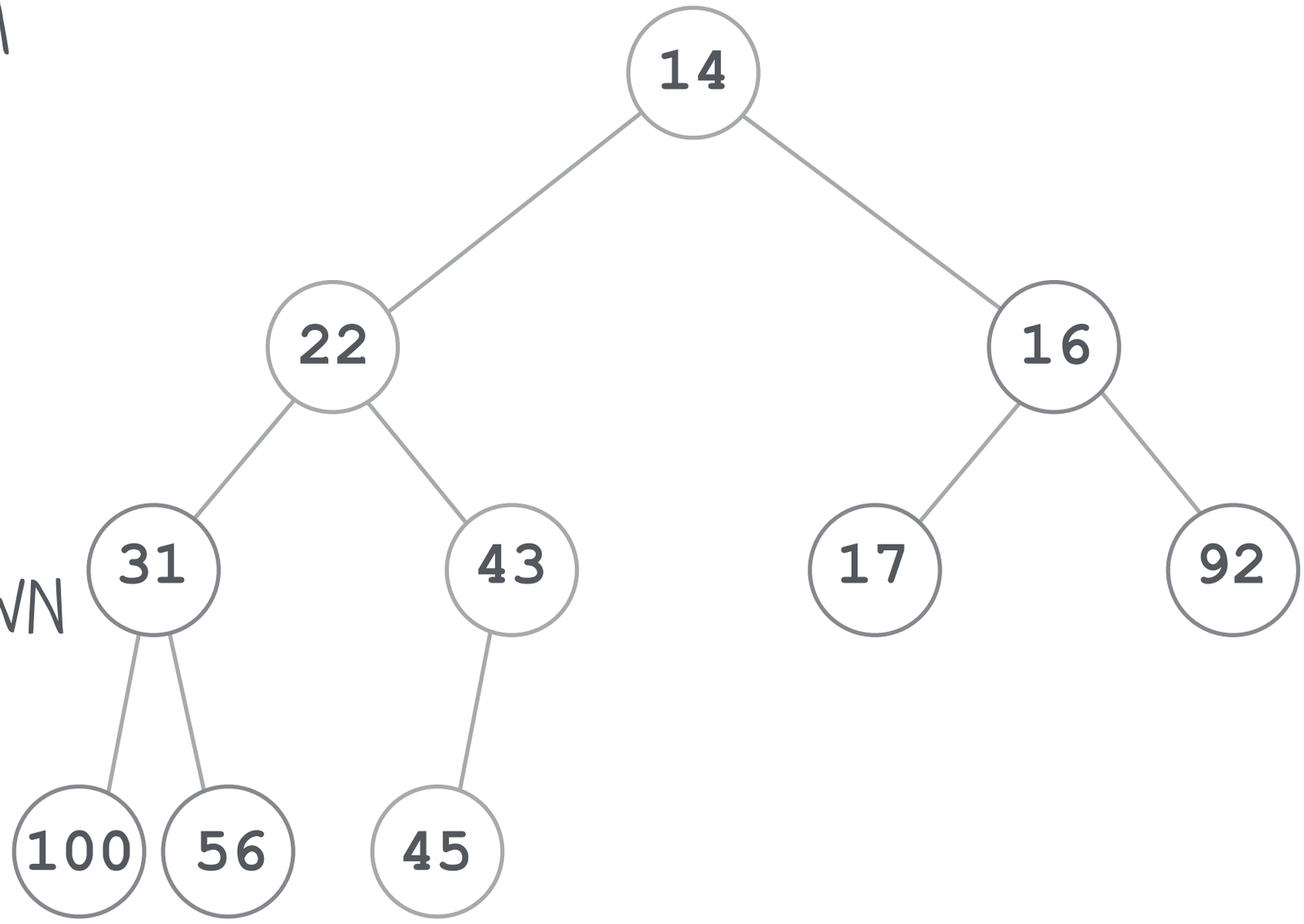


LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3

FILL WITH LAST ITEM ON LAST LEVEL. WHY?

PERCOLATE DOWN



cost of remove

-worst case is **O(logN)**

  -percolating down to the bottom level

-average case is also **O(logN)**

-rarely terminates more than 1-2 levels from the bottom… why?

back to the priority queue…

-option 1: a linked list

- -add: **O(1)**
- -findMin: **O(N)**
- -deleteMin: **O(N)** (including finding)

-option 2: a sorted linked list

- -add: **O(N)**
- -findMin: **O(1)**
- -deleteMin: **O(1)**

-option 3: a self-balancing BST

- -add: **O(logN)**
- -findMin: **O(logN)**
- -deleteMin: **O(logN)**

-option 4: a binary heap

- -add: **O(1)** (percolate up average of 1.6 swaps)
- -findMin: **O(1)** (just access the root)
- -deleteMin: **O(logN)** (percolate down but rarely terminates before bottom)

today…

-min-max heaps

-add

-delete max

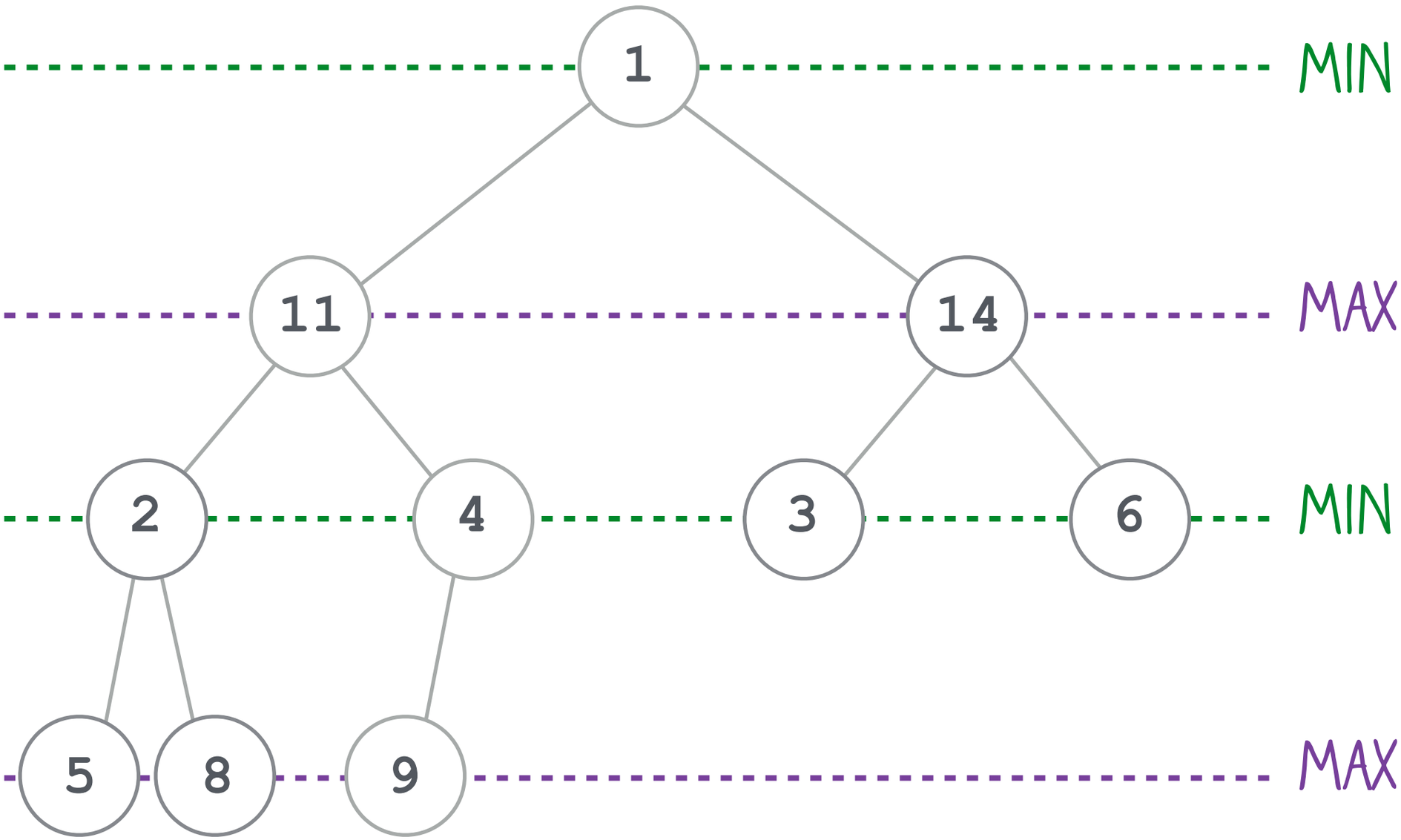
-delete min

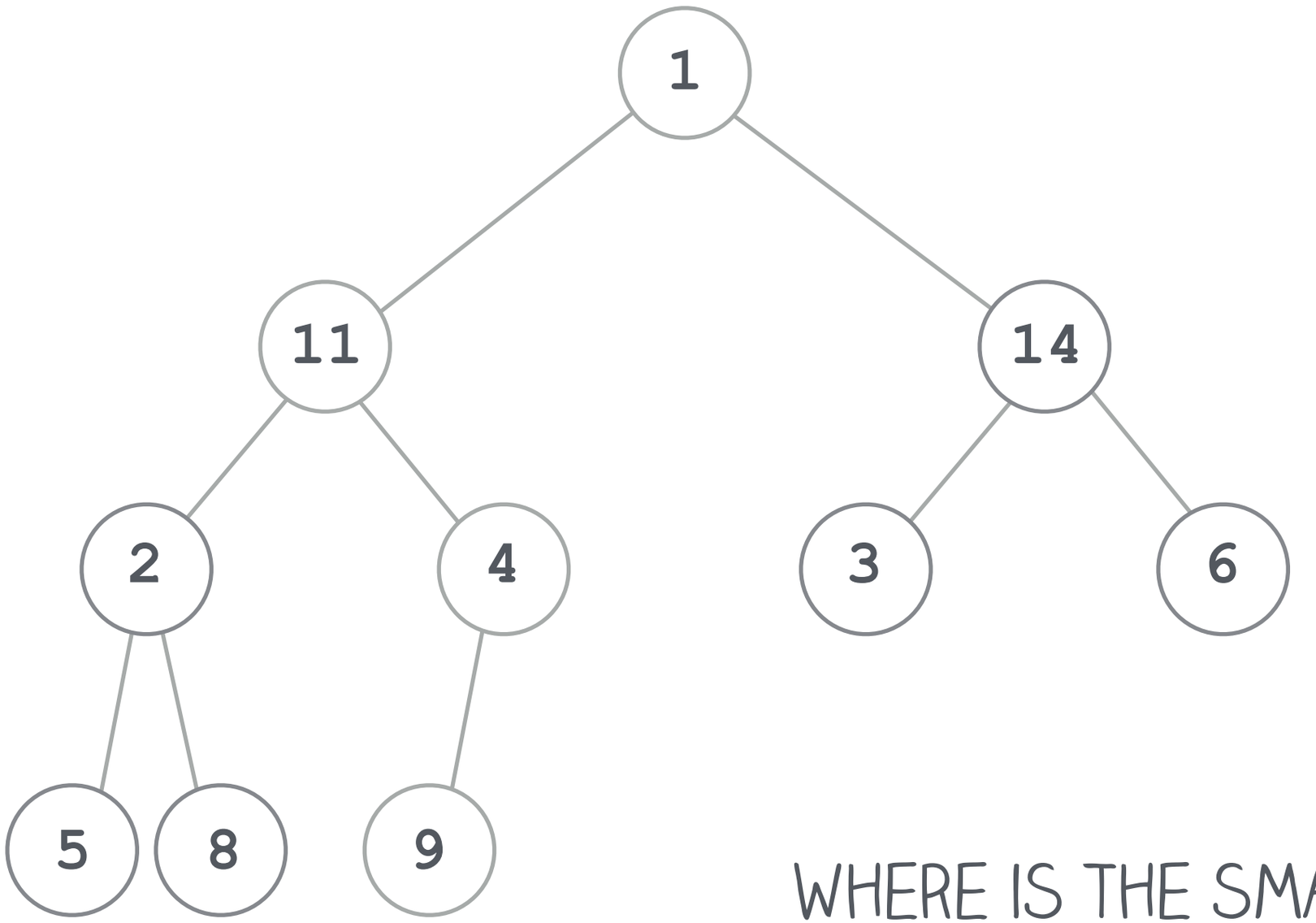
-delete algorithm

min-max heaps

- a **min-max heap** further extends the heap order property
  - for any node E at *even* depth, E is the minimum element in its subtree
  - for any node O at *odd* depth, O is the maximum element in its subtree

- the root is considered to be at even depth (zero)


1
MIN
11
14
MAX
2
4
3
6
MIN
5
8
9
MAX



WHERE IS THE SMALLEST ITEM?

WHERE IS THE LARGEST ITEM?

add

-AGAIN, we must ensure the heap property structure

-must be a complete tree

-add an item to the next open leaf node

-THEN, restore order with its parent

-does it belong on a min level or a max level?

-swap if necessary

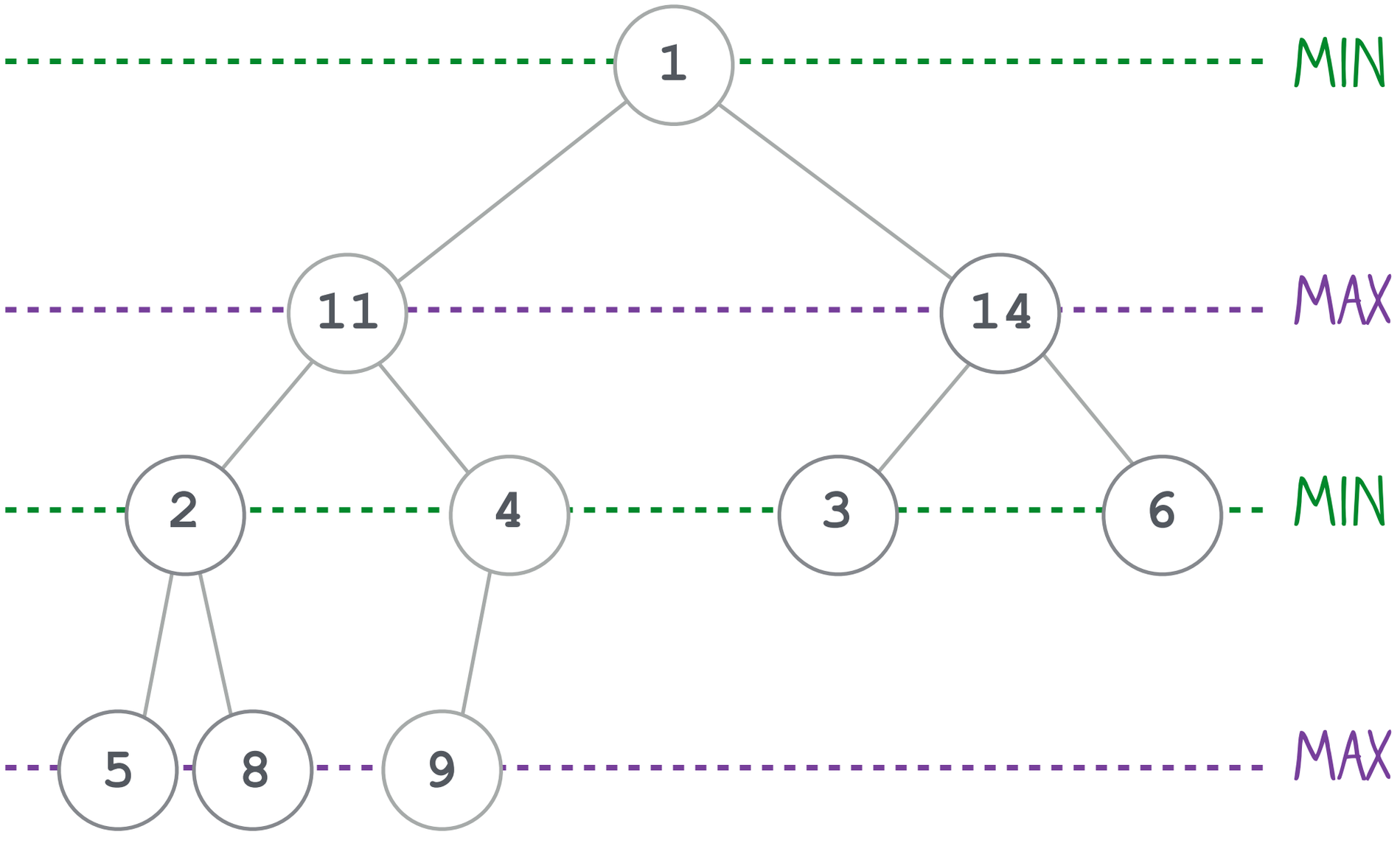
-the new location determines if it is a min or max node

-percolate up the appropriate levels

-if new item is a max node, percolate up max levels

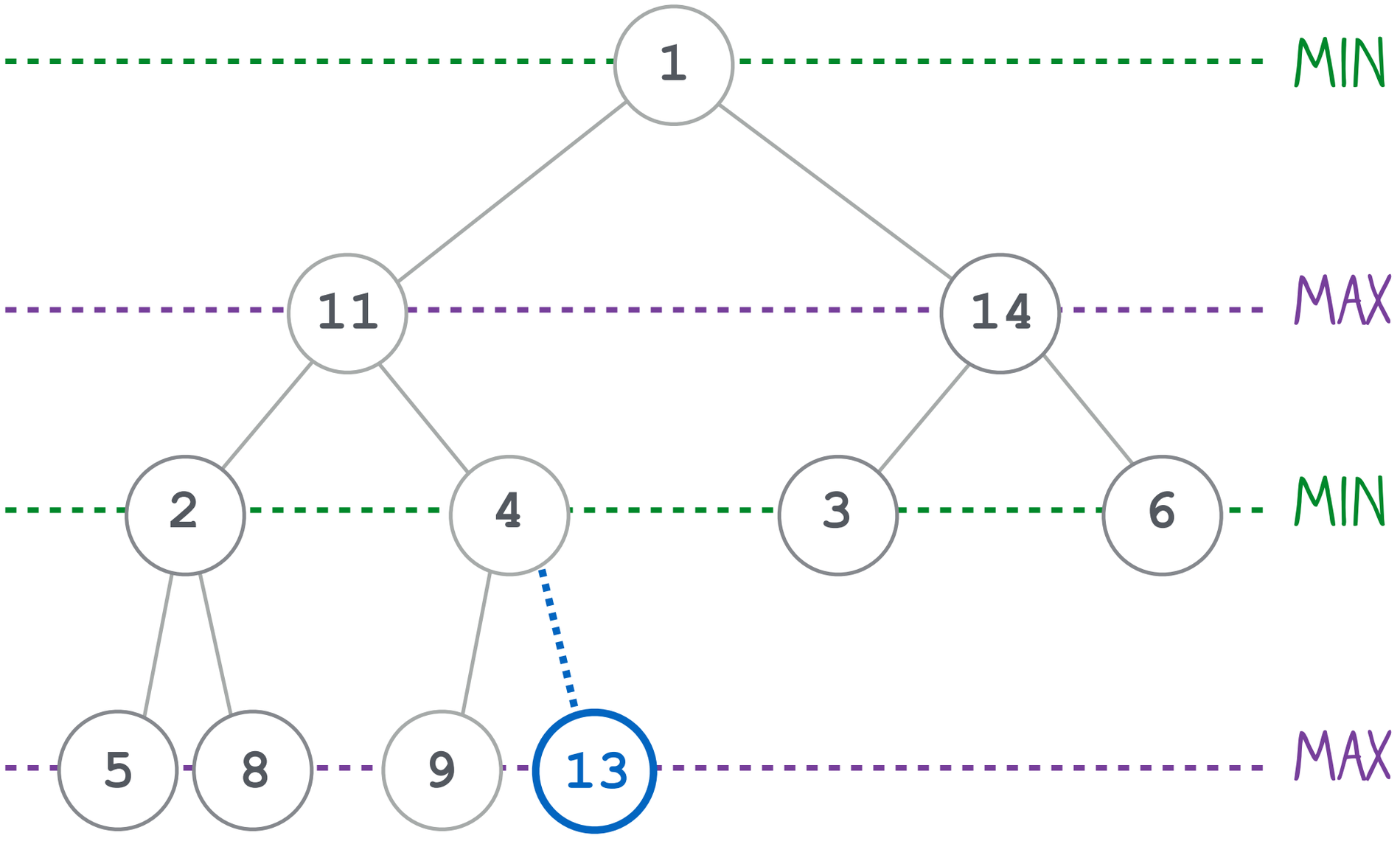
-else, percolate up min levels

WANT TO ADD 13



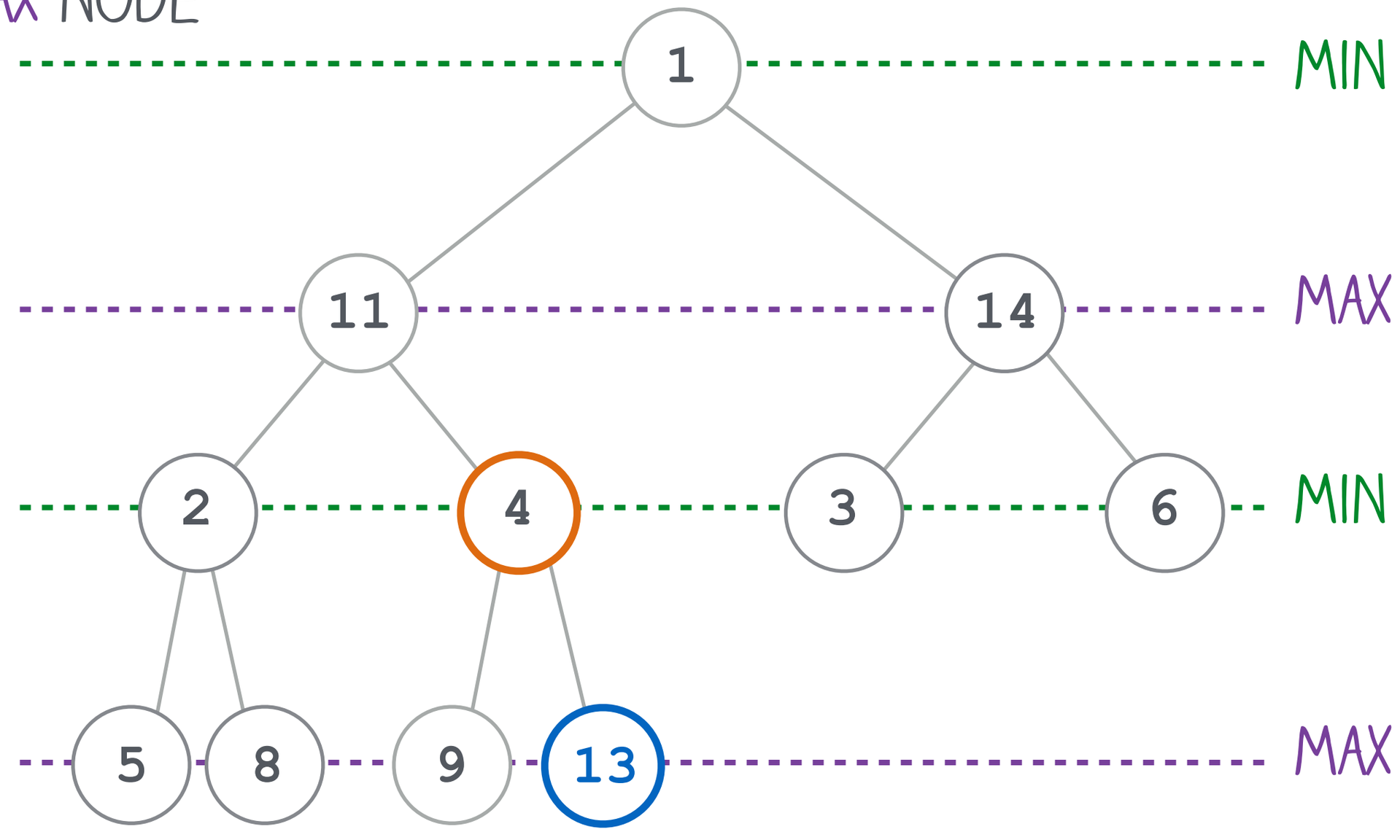
WANT TO ADD 13

ADD TO FIRST OPEN SPACE



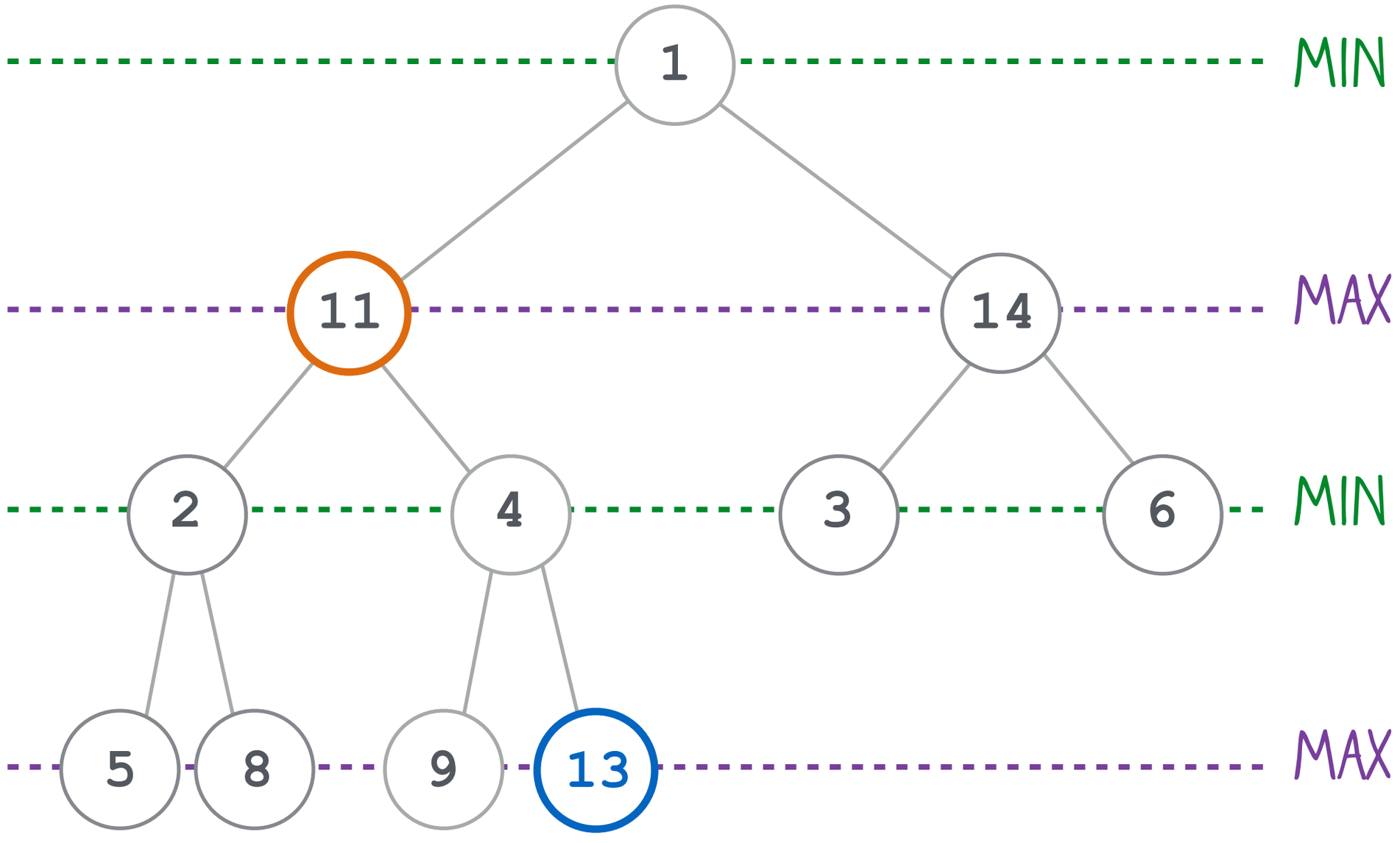
PARENT ON A MIN LEVEL, AND 13 IS GREATER THAN PARENT, SO NO SWAP NECESSARY

13 IS NOW A MAX NODE



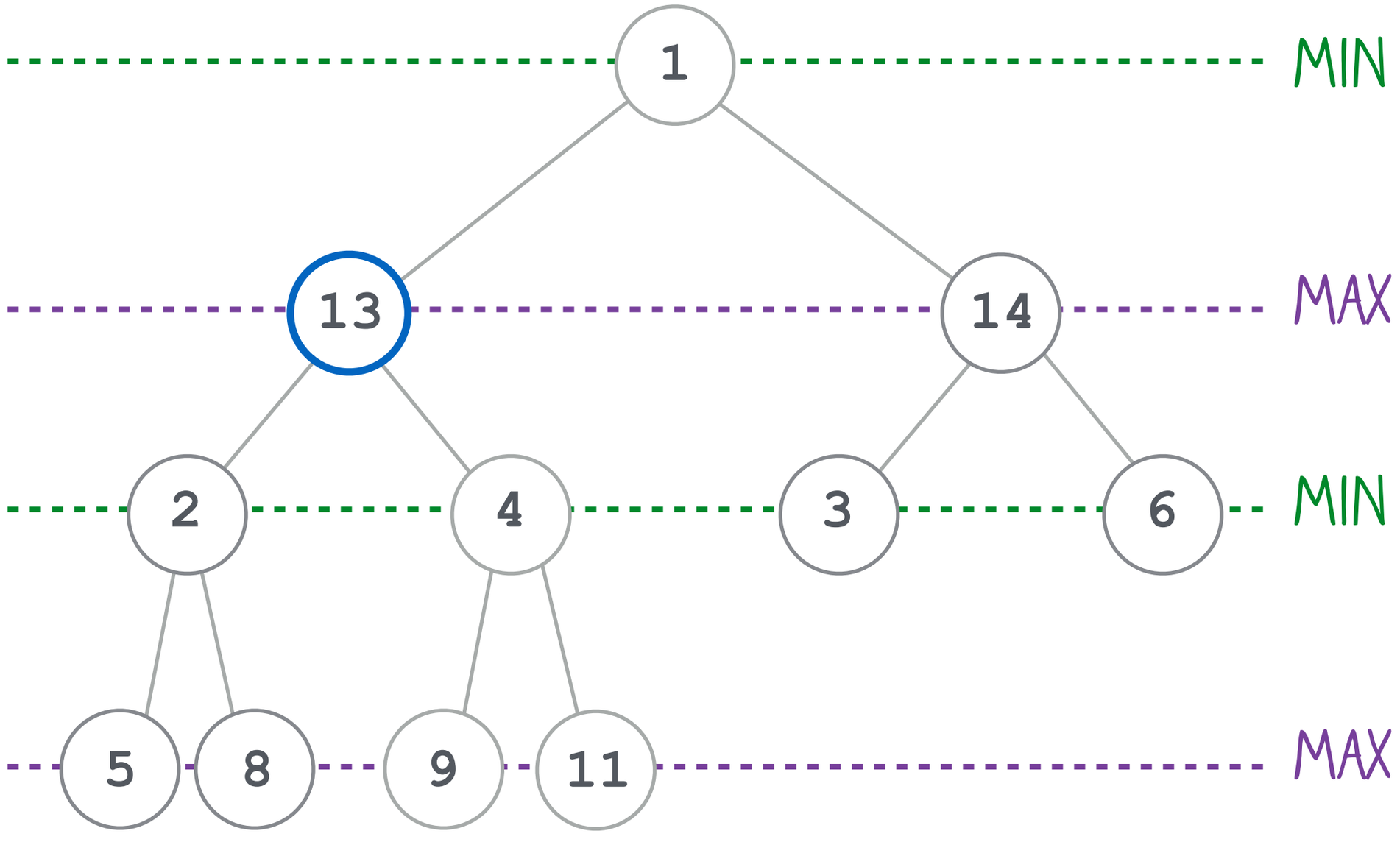
PERCOLATE UP THE MAX LEVELS

COMPARE TO GRANDPARENT



PERCOLATE UP THE MAX LEVELS

COMPARE TO GRANDPARENT

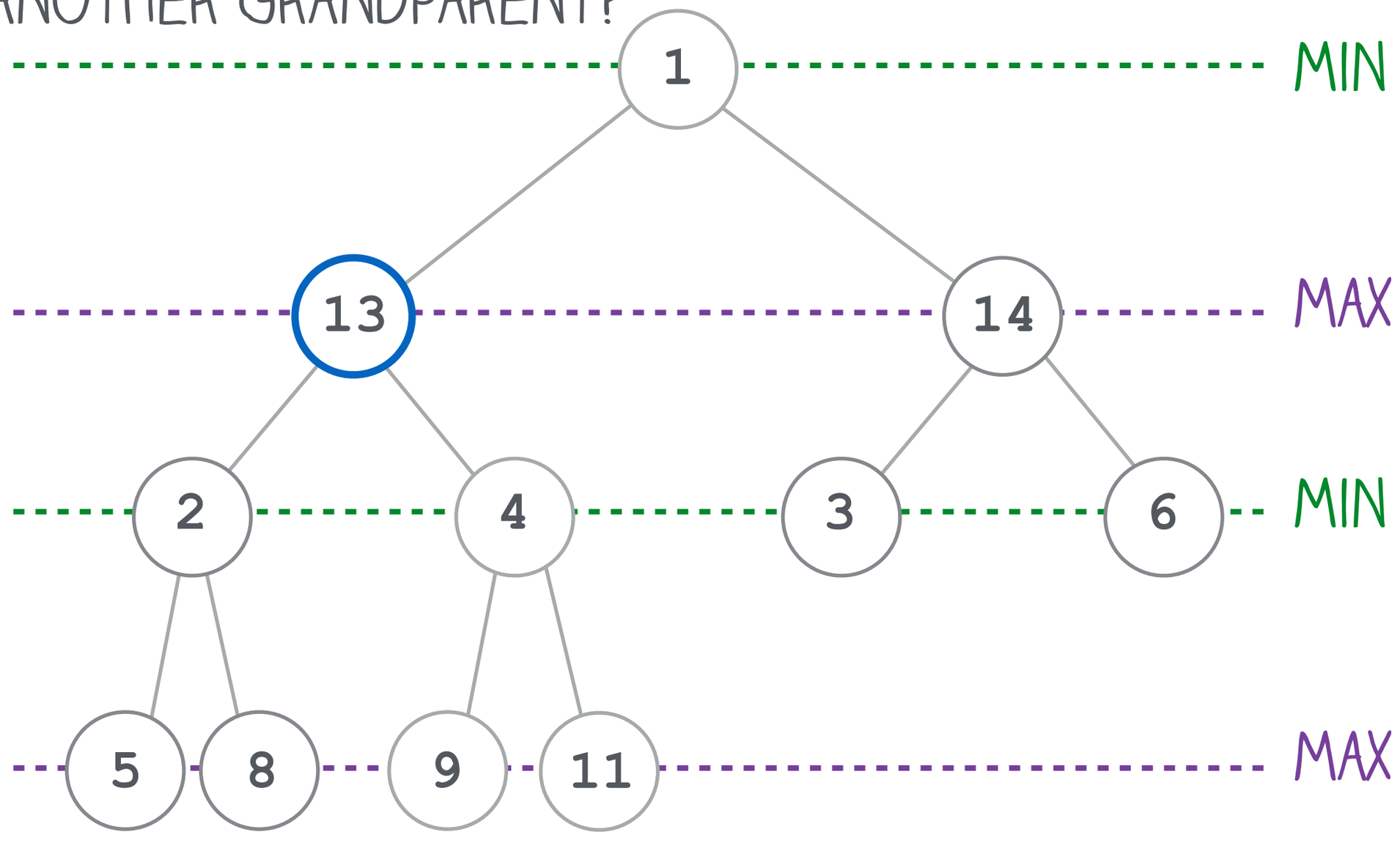


PERCOLATE UP THE MAX LEVELS

COMPARE TO GRANDPARENT

DOES 13 HAVE ANOTHER GRANDPARENT?

DONE!



-if the parent is on a **min** level, new node must be greater than the parent

-if the parent is on a **max** level, new node must be less than the parent

-percolate up like normal, except skip every other level

delete max

-max node is one of the two children of the root

-replace max node with the last leaf node in the tree

-preserve structure property!

-restore order with the new node's children

-if any child is larger, swap

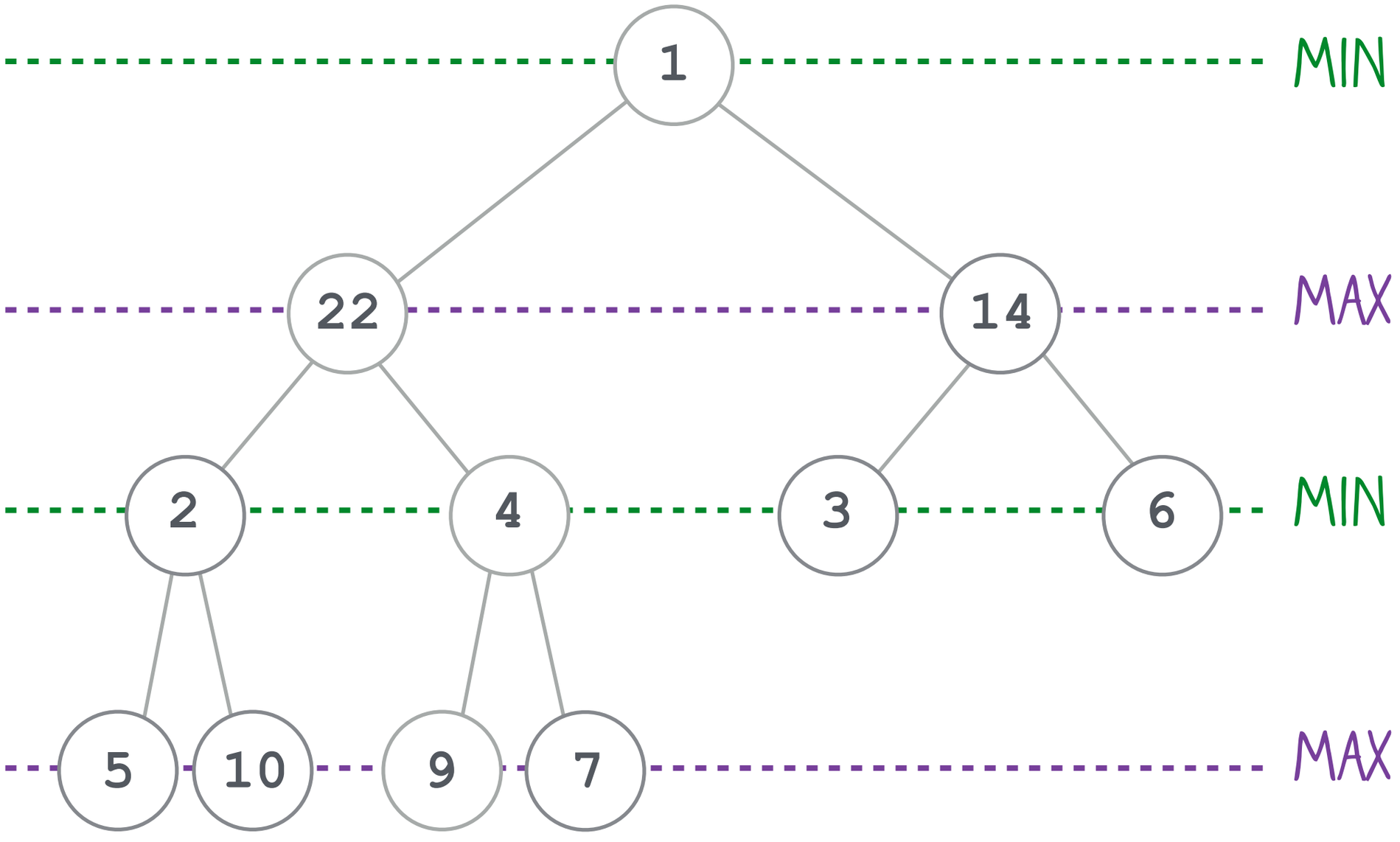
-percolate swapped child down the max levels

-if no child was larger, percolate the new node down the max levels

-if the node reaches the second to last level of tree, may require one more swap with direct children

WANT TO DELETE THE MAX

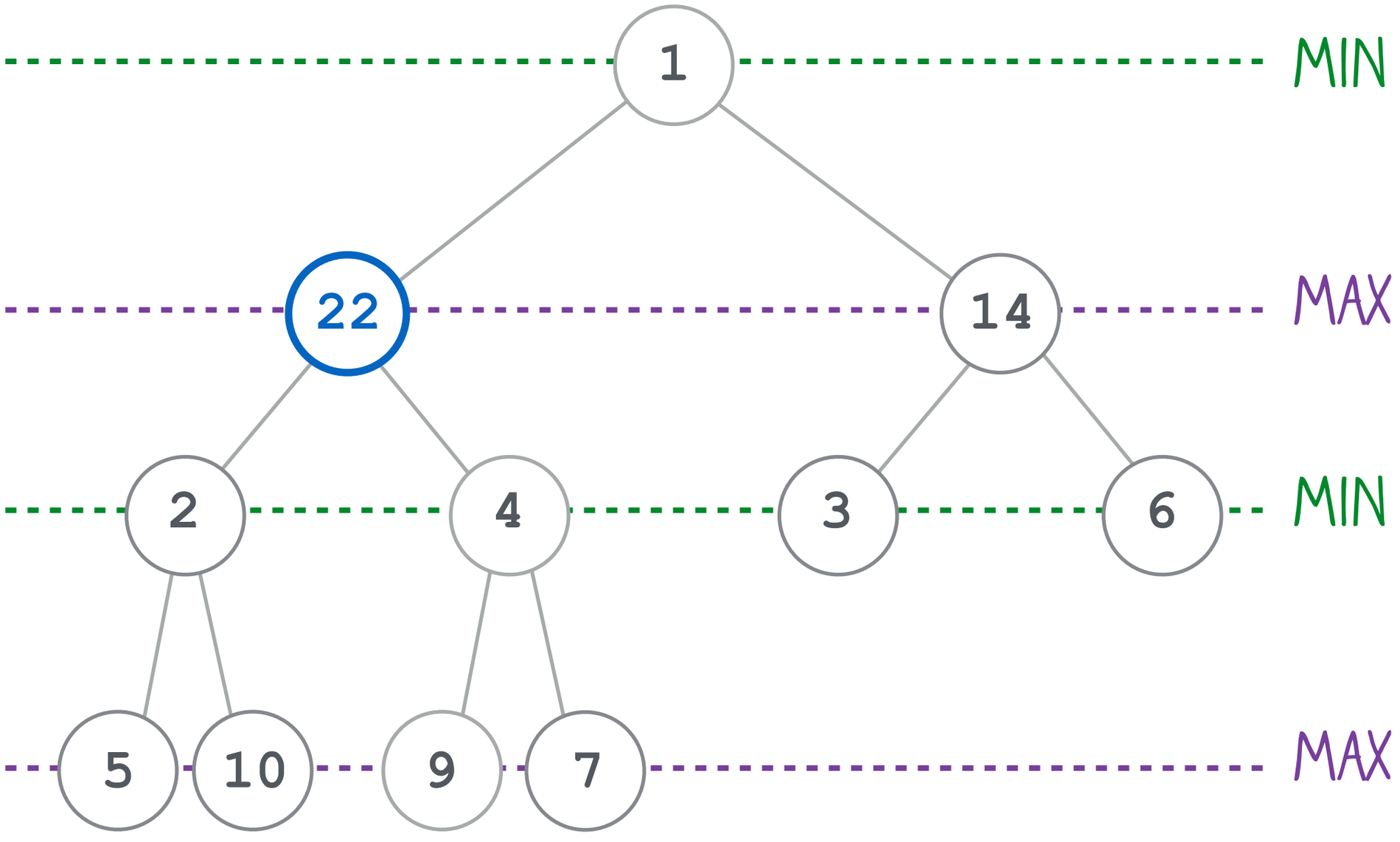
COMPARE CHILDREN OF ROOT



WANT TO DELETE THE MAX

COMPARE CHILDREN OF ROOT

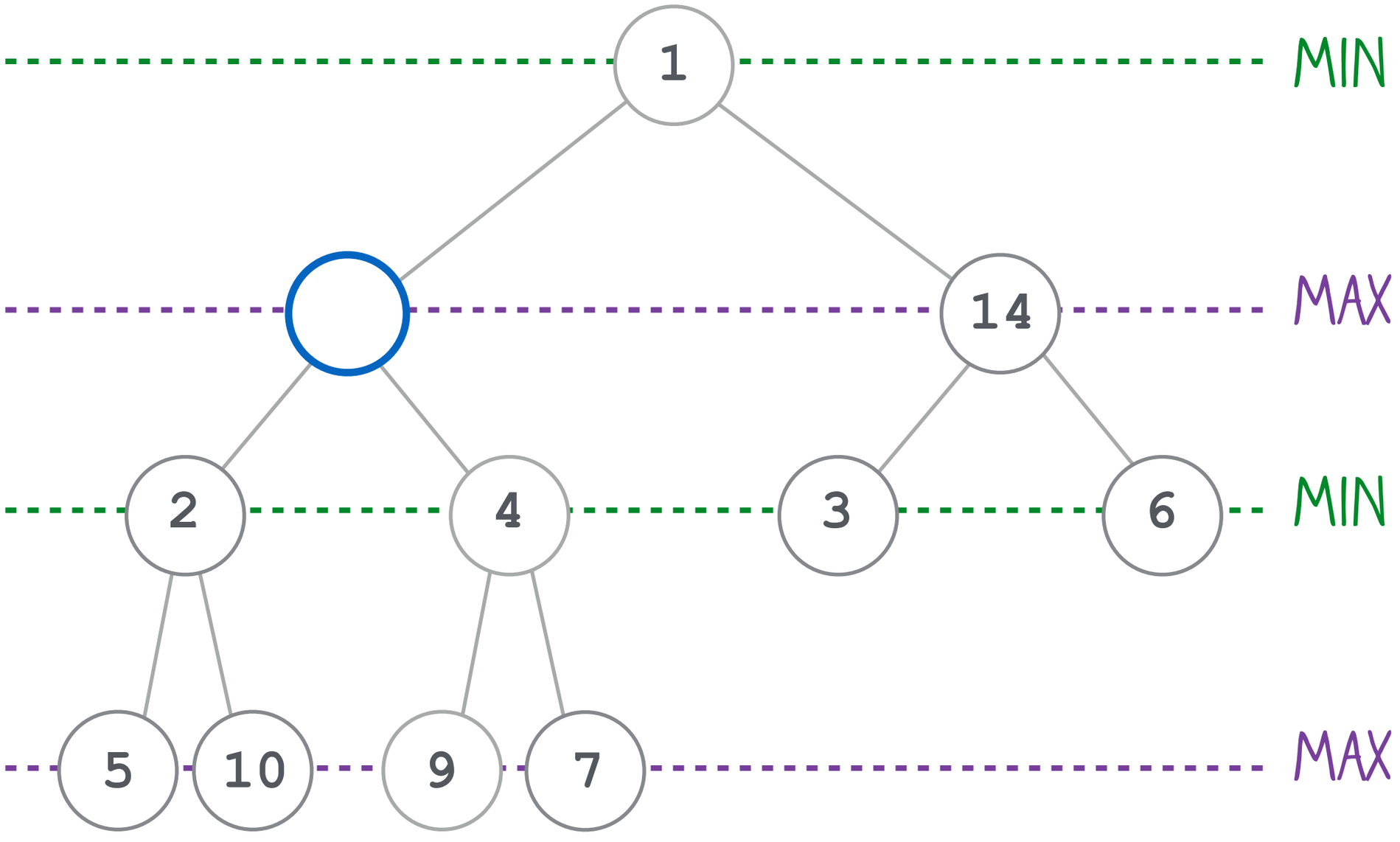
22 IS MAX



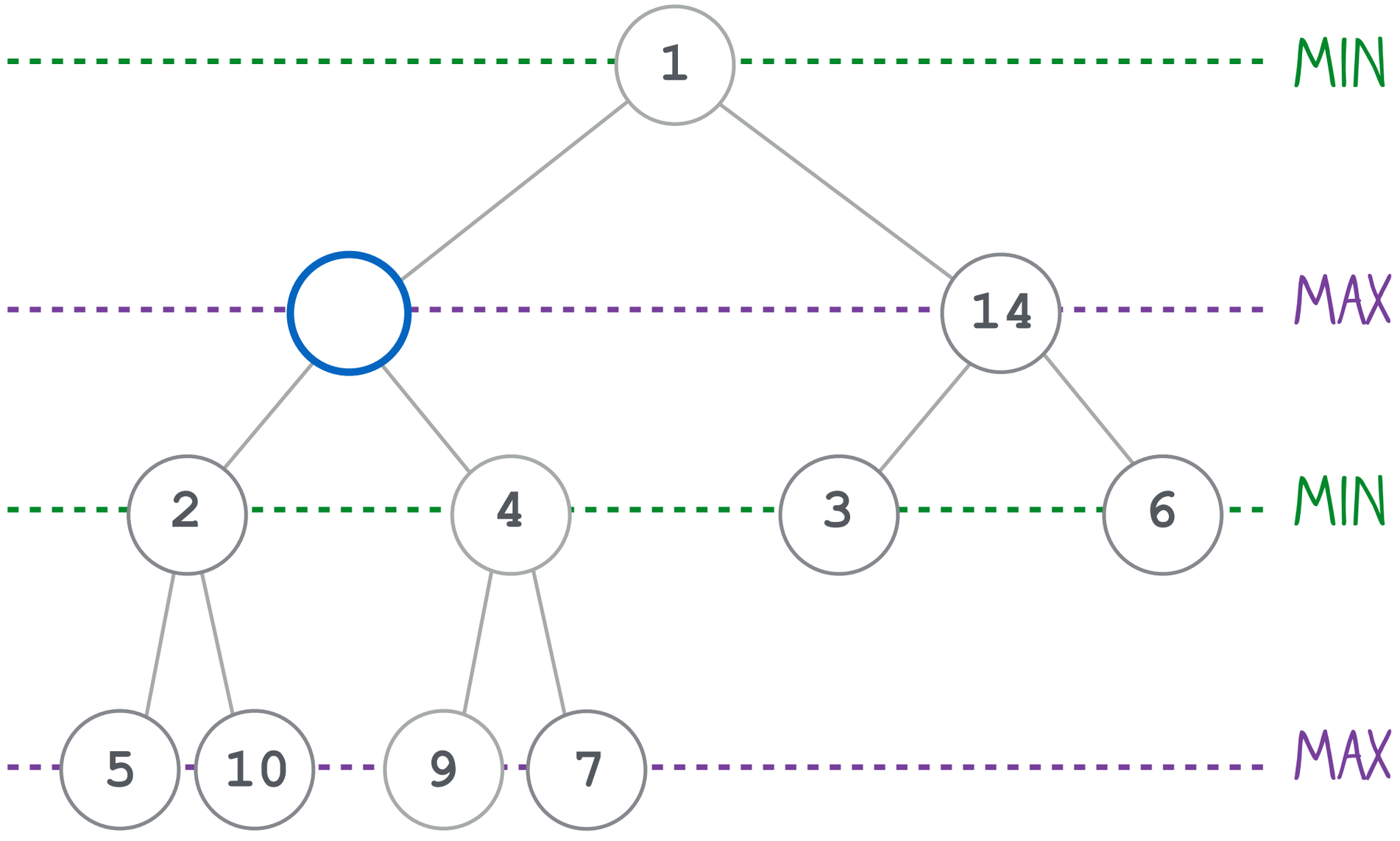
WANT TO DELETE THE MAX

COMPARE CHILDREN OF ROOT

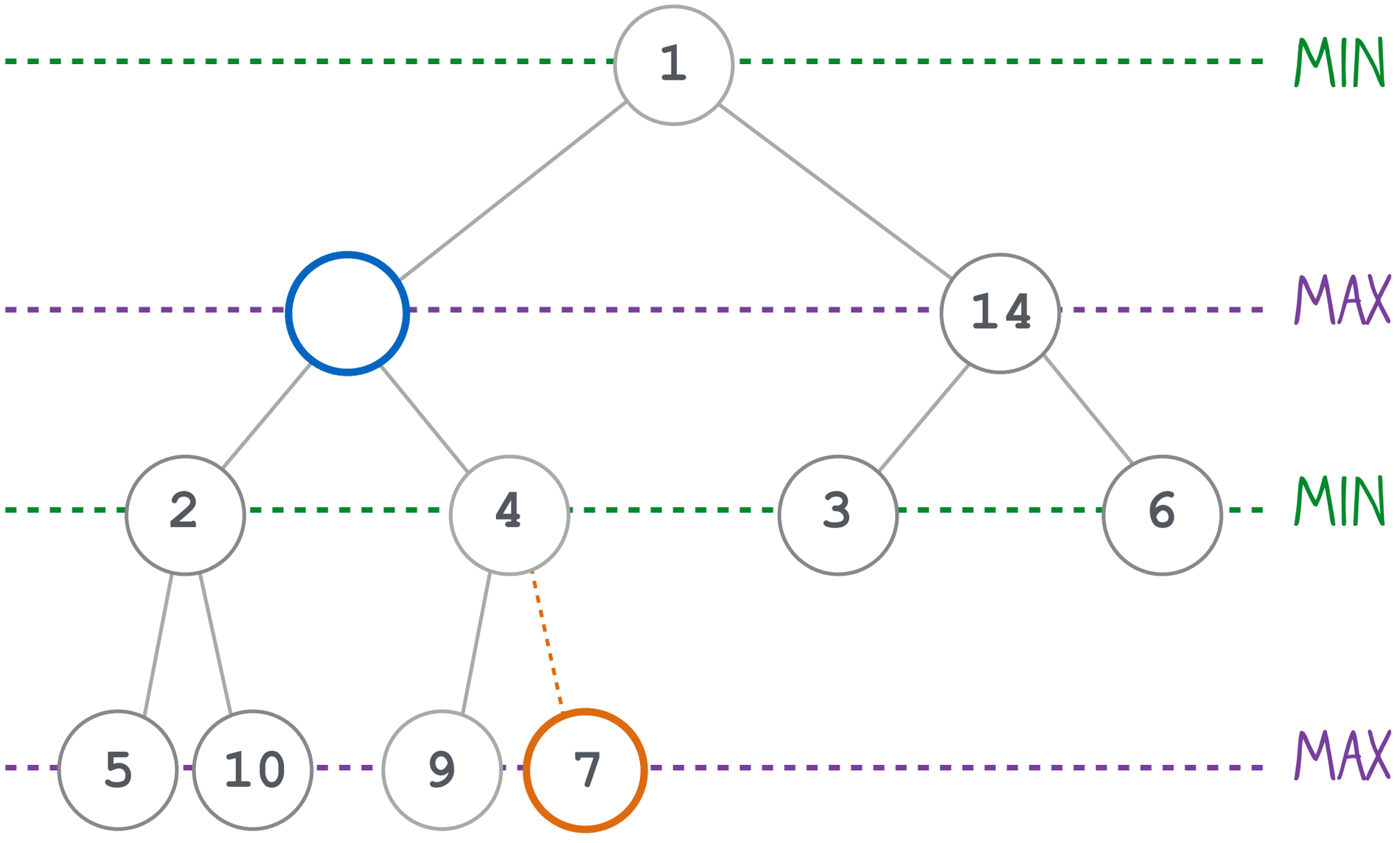
22 IS MAX



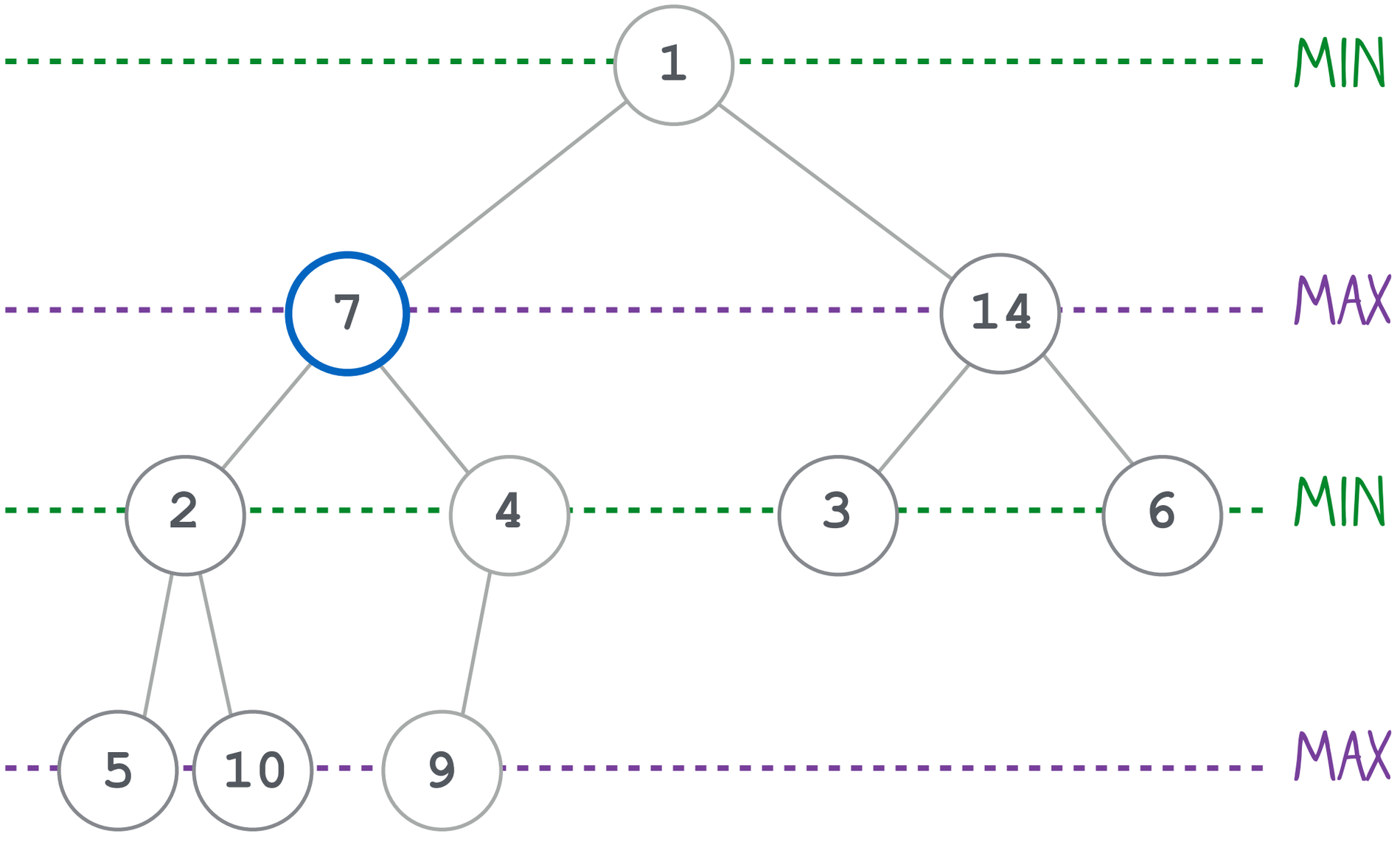
REPLACE WITH LAST LEAF NODE



REPLACE WITH LAST LEAF NODE

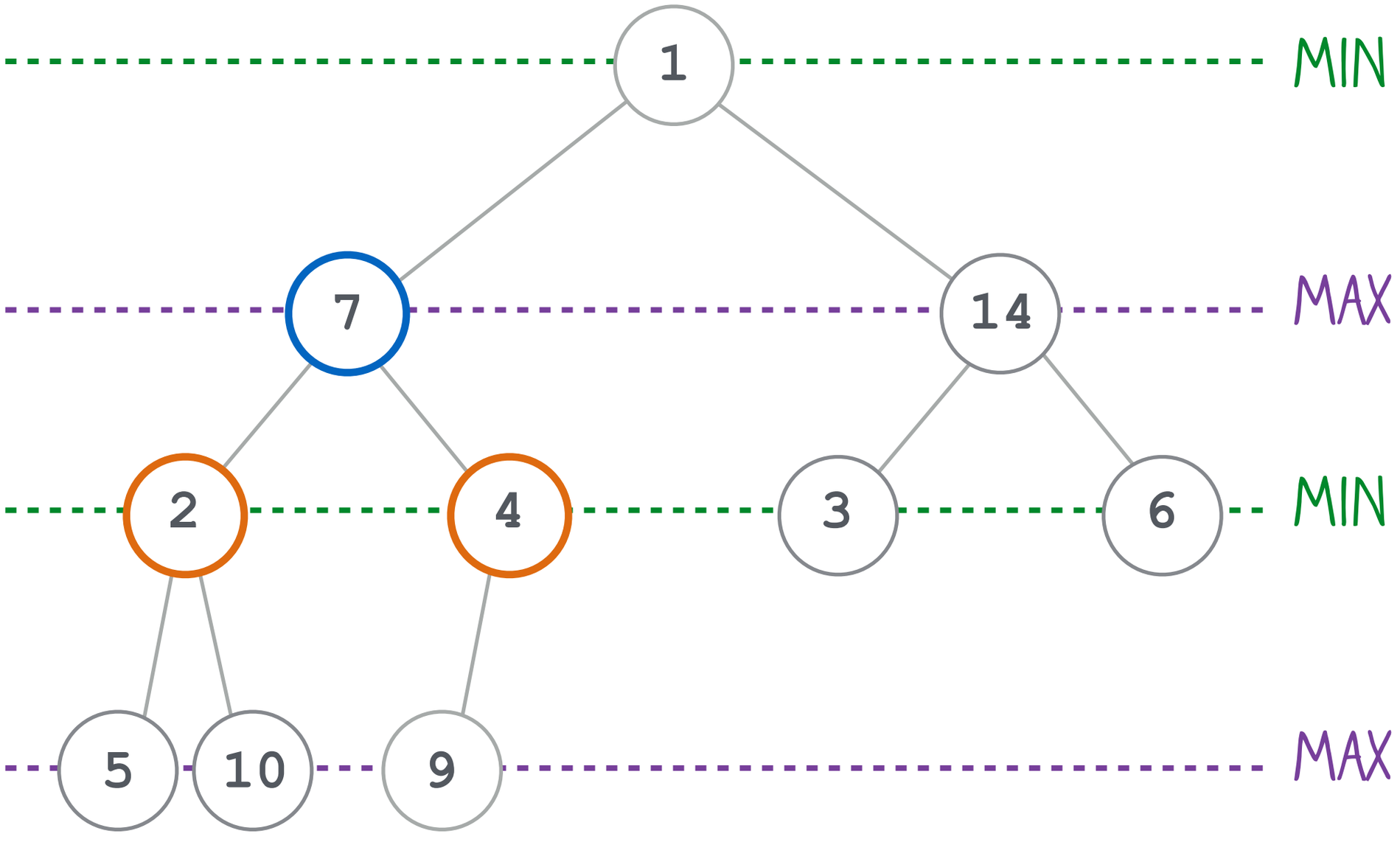


REPLACE WITH LAST LEAF NODE



RESTORE ORDER WITH CHILDREN

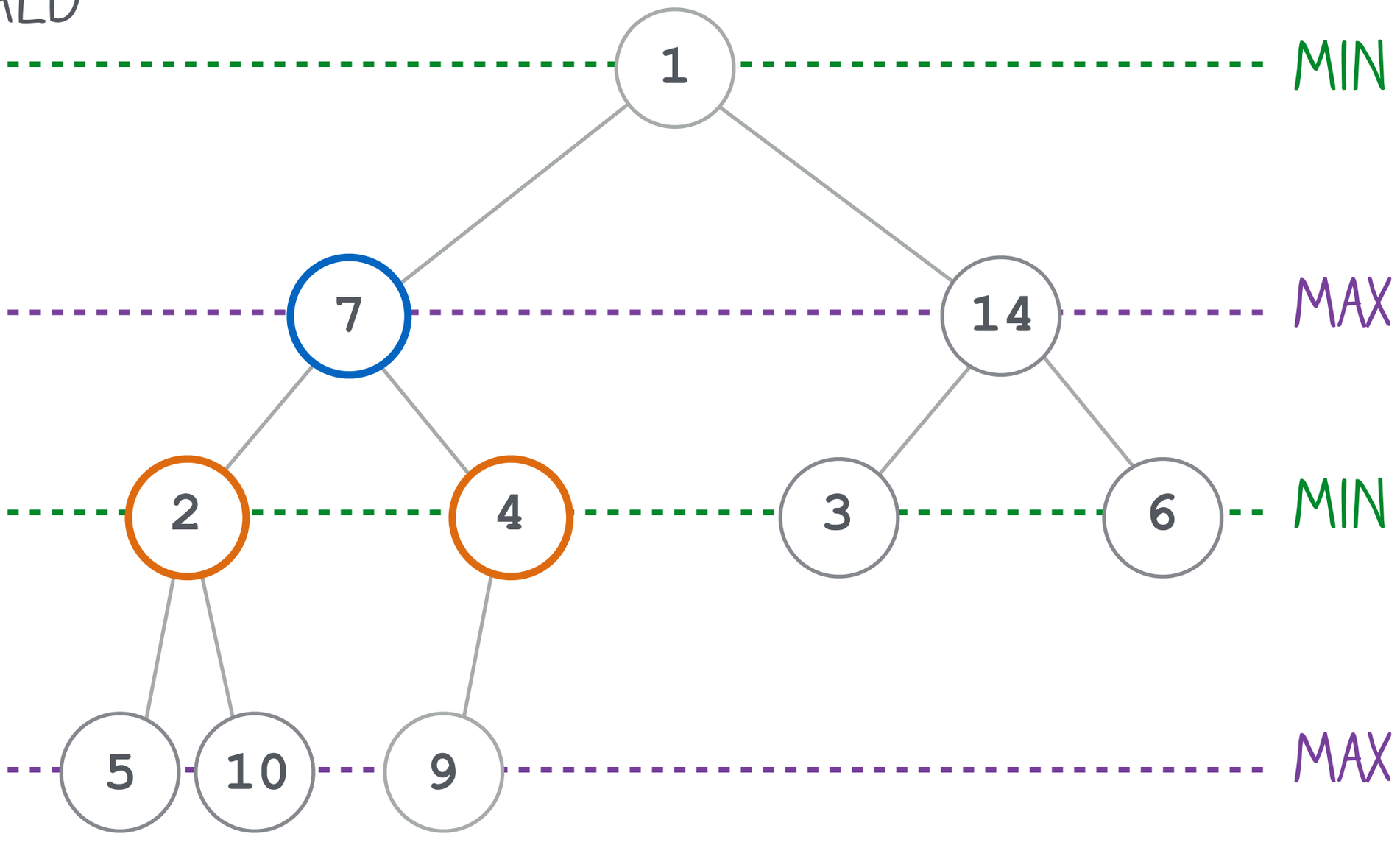
7 IS ON A MAX LEVEL, SO MUST BE > THAN CHILDREN



RESTORE ORDER WITH CHILDREN

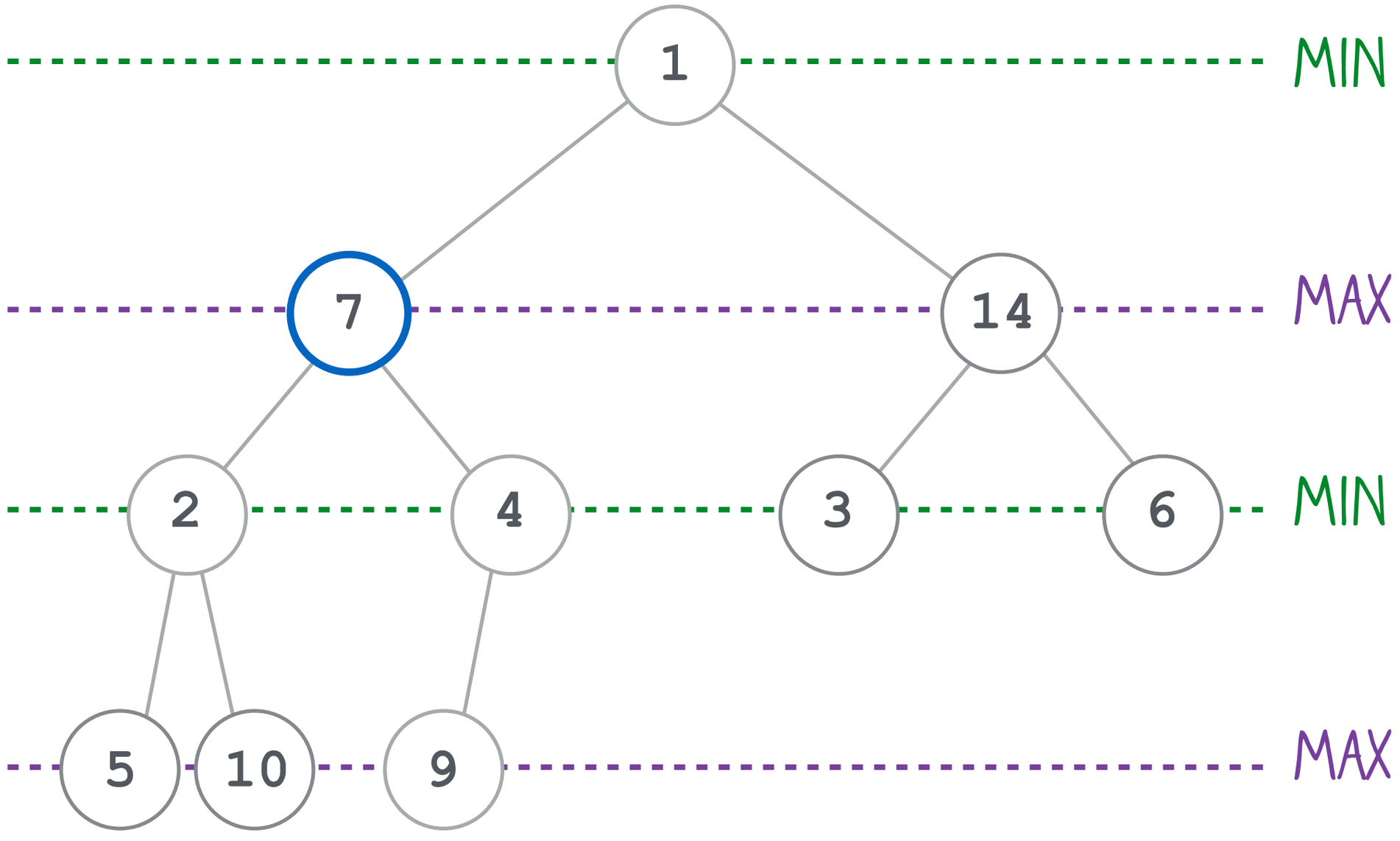
7 IS ON A MAX LEVEL, SO MUST BE > THAN CHILDREN

NO SWAP REQUIRED



7 IS NOW A MAX NODE

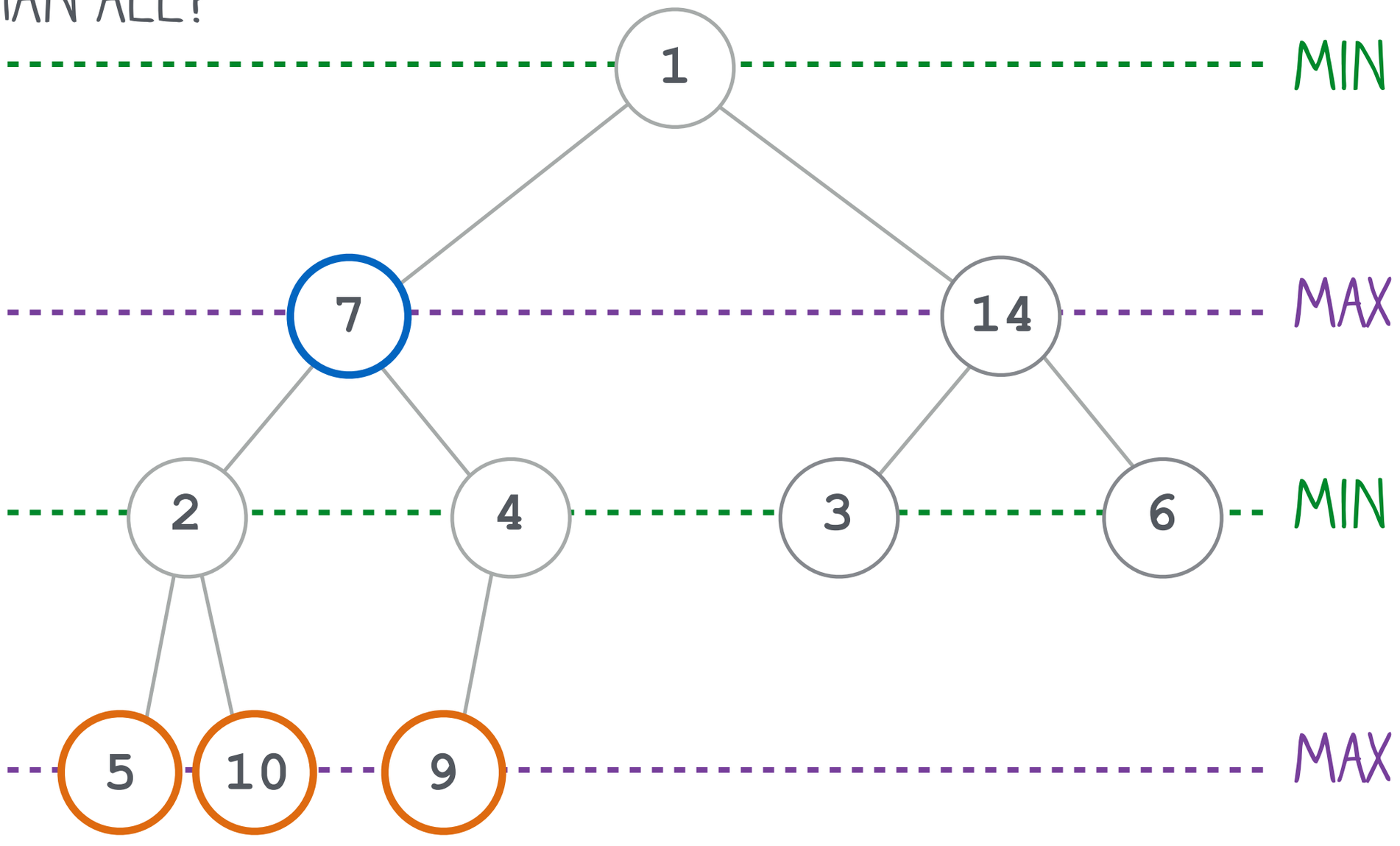
PERCOLATE DOWN MAX LEVELS



COMPARE TO ALL 4 GRANDCHILDREN

(OR, 3 IN THIS CASE)

IS 7 GREATER THAN ALL?

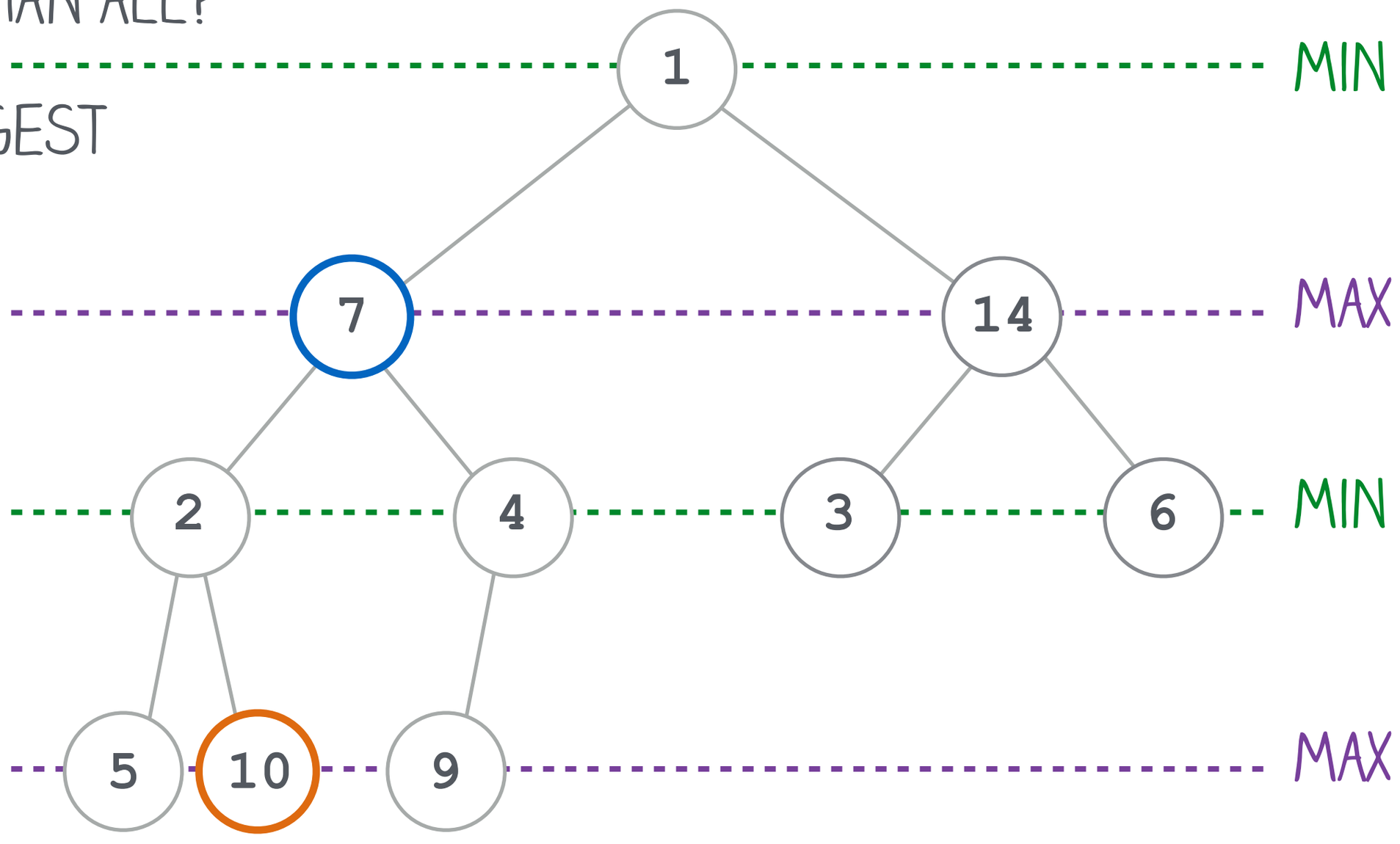


COMPARE TO ALL 4 GRANDCHILDREN

(OR, 3 IN THIS CASE)

IS 7 GREATER THAN ALL?

SWAP WITH LARGEST

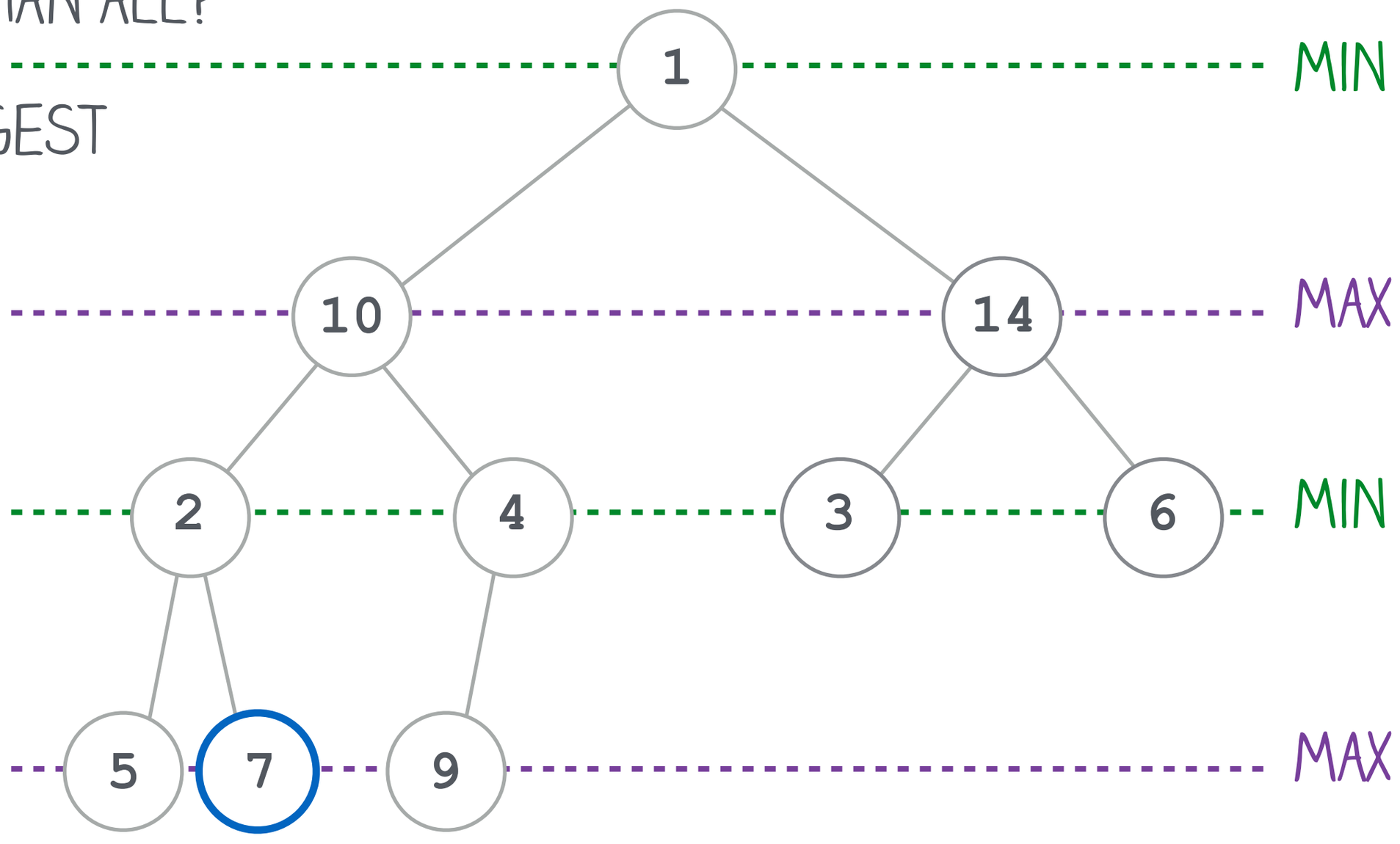


COMPARE TO ALL 4 GRANDCHILDREN

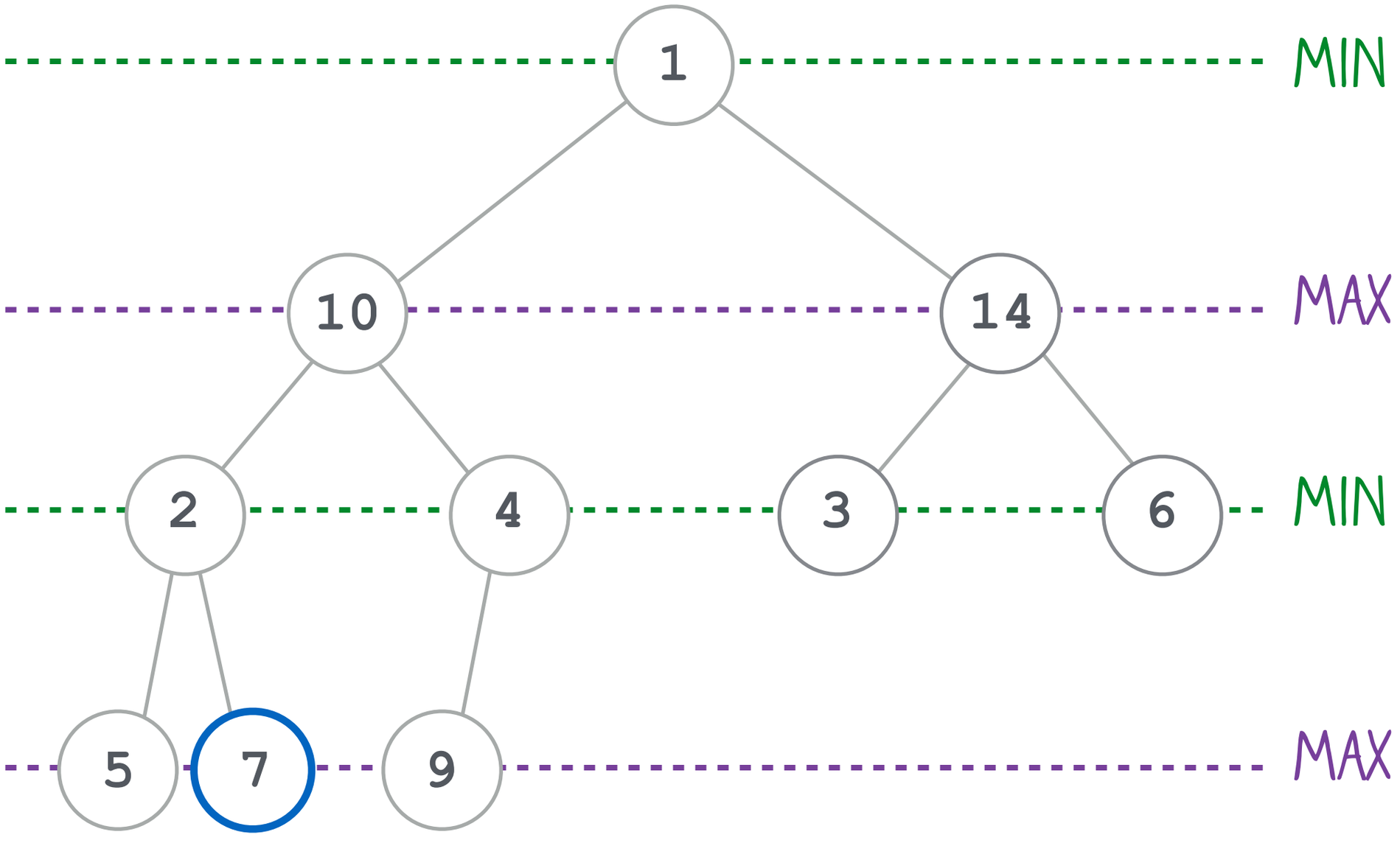
(OR, 3 IN THIS CASE)

IS 7 GREATER THAN ALL?

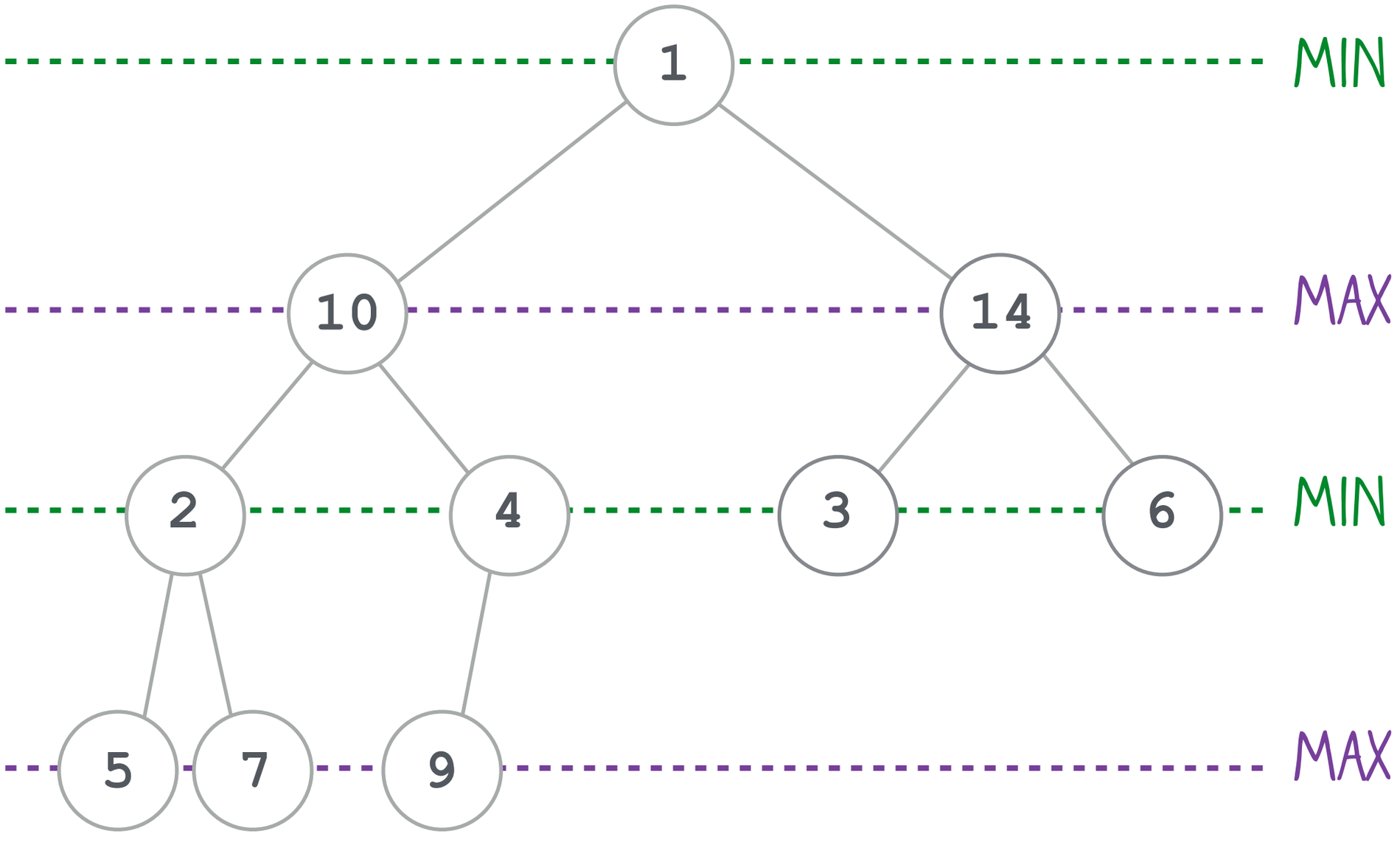
SWAP WITH LARGEST



DOES 7 HAVE MORE GRANDCHILDREN?



DONE!



-if 7 had been less than one of its DIRECT children we would have swapped them

-we would have then percolated that child down the max levels instead of 7

-very similar to a regular min-heap, except we skip every other level when percolating

deleting the min

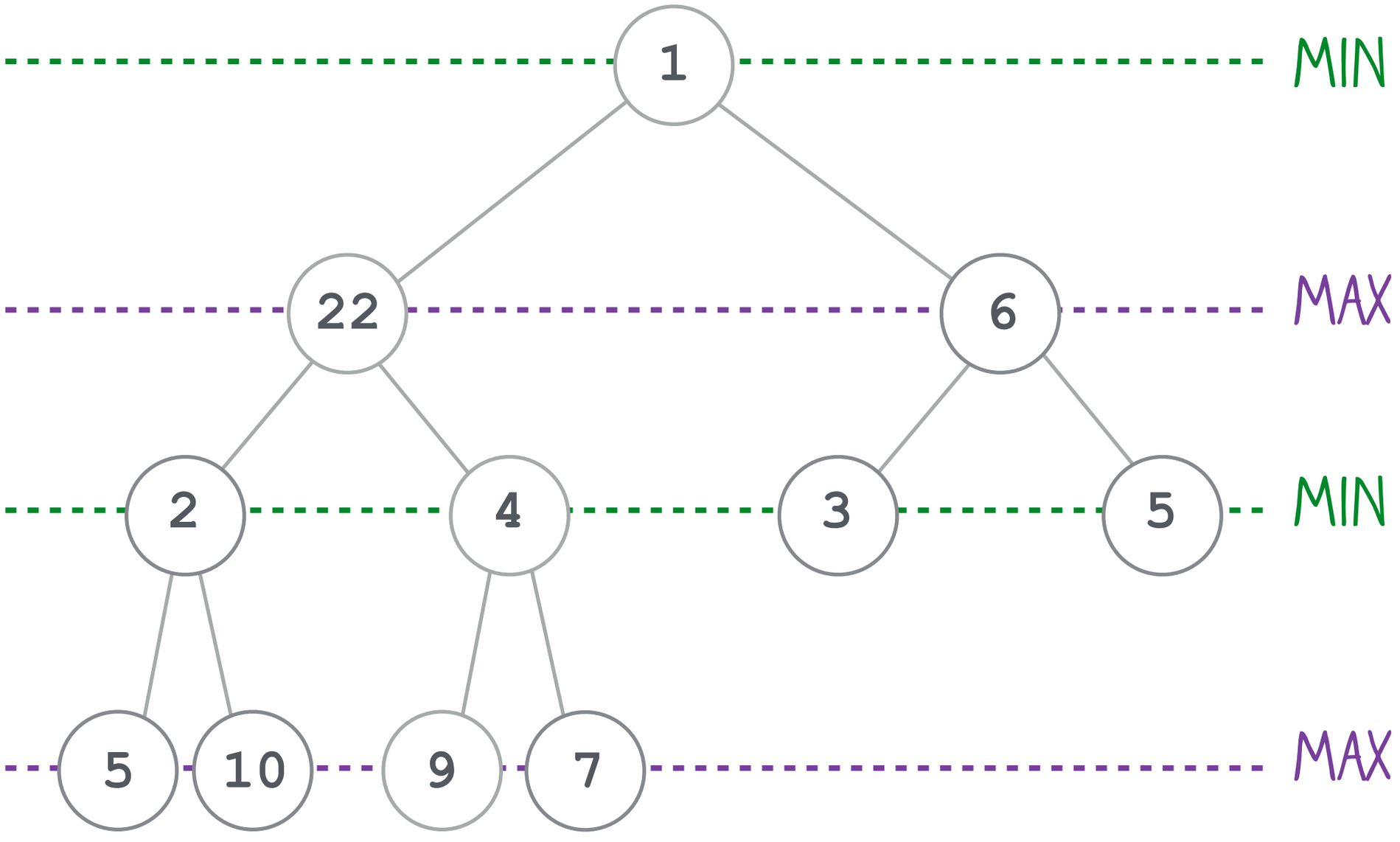
-the min node is always the ___

-replace it with last leaf node

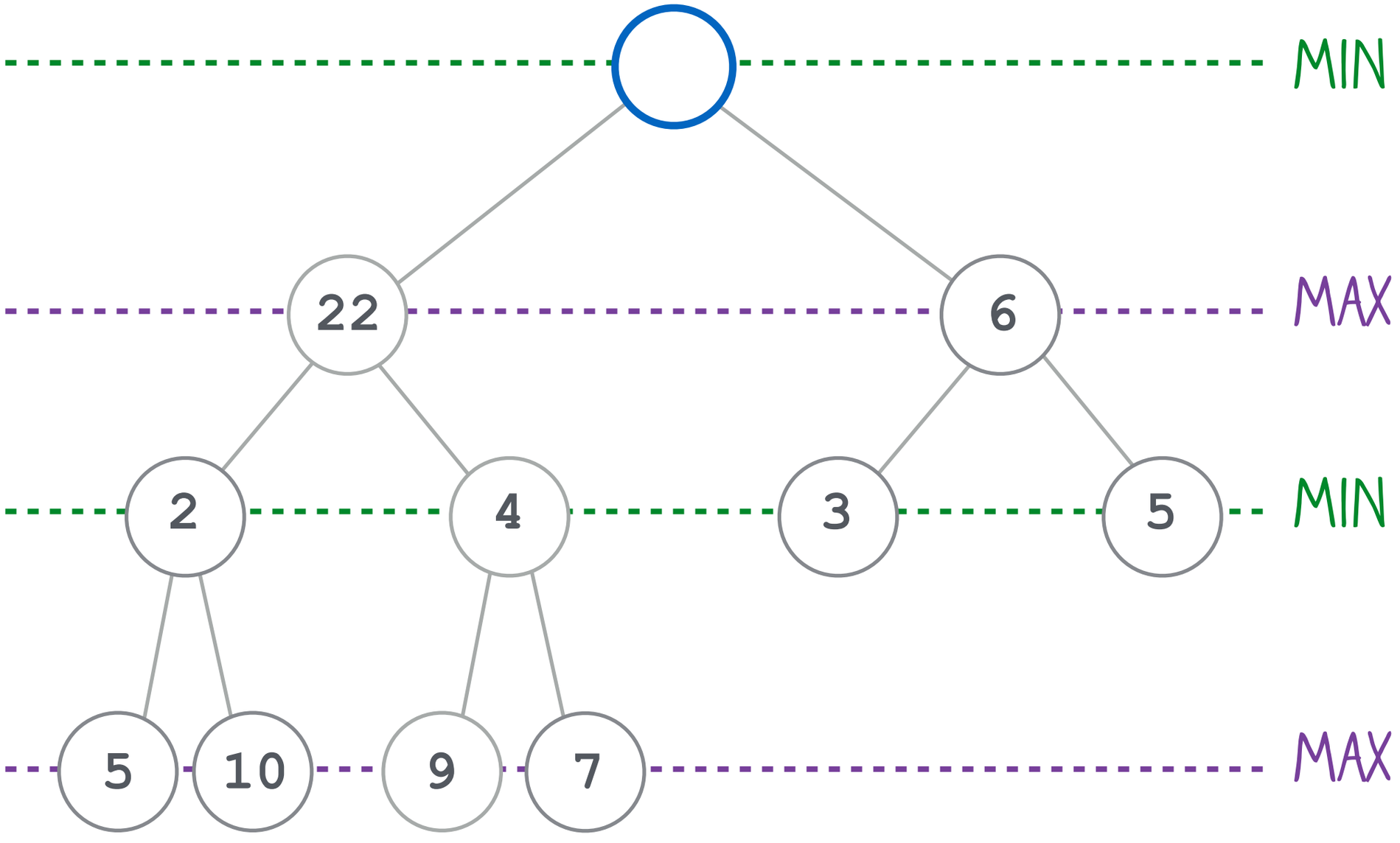
-restore order with direct children

-then, percolate new root down the min levels

WANT TO DELETE THE MIN

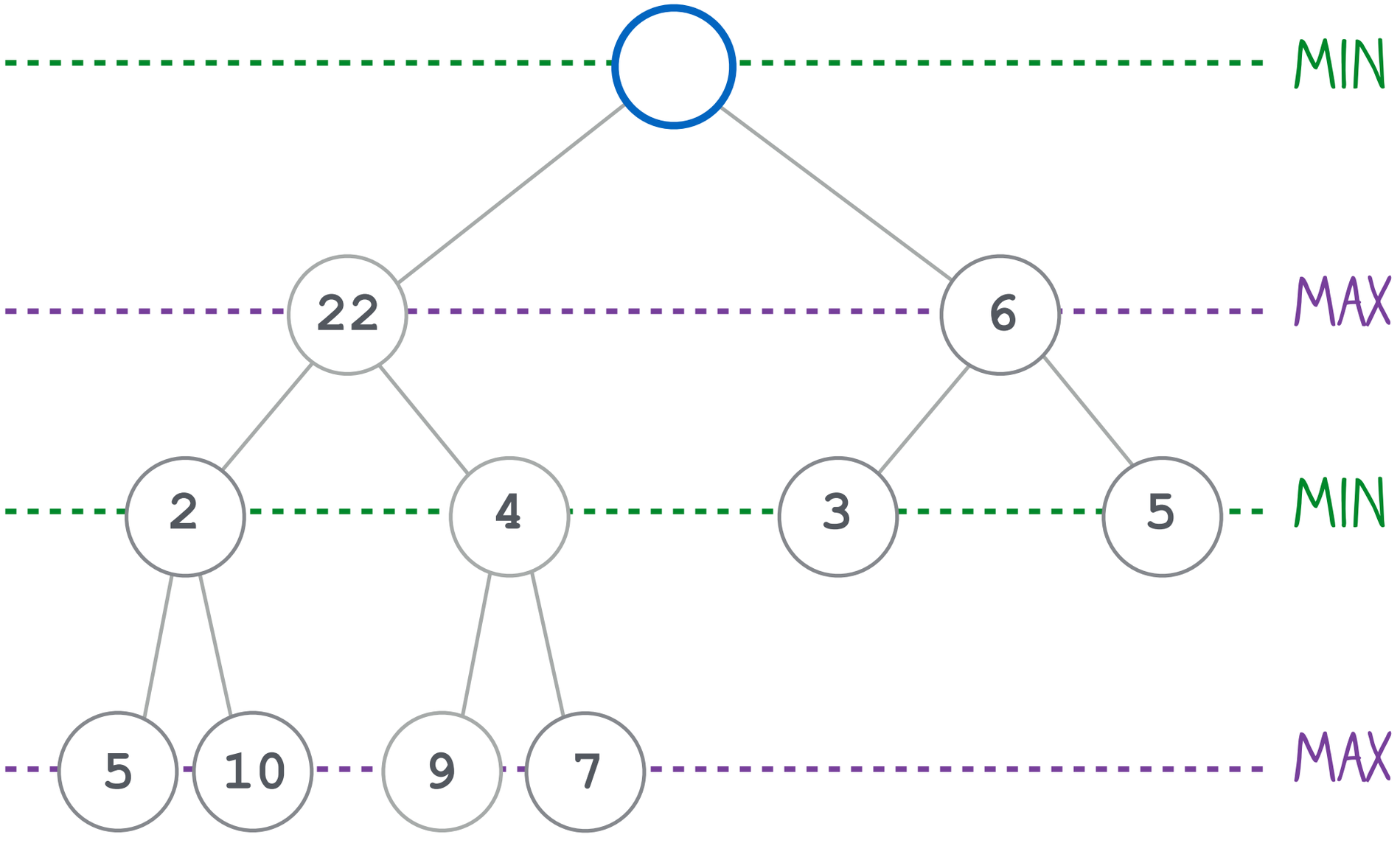


WANT TO DELETE THE MIN



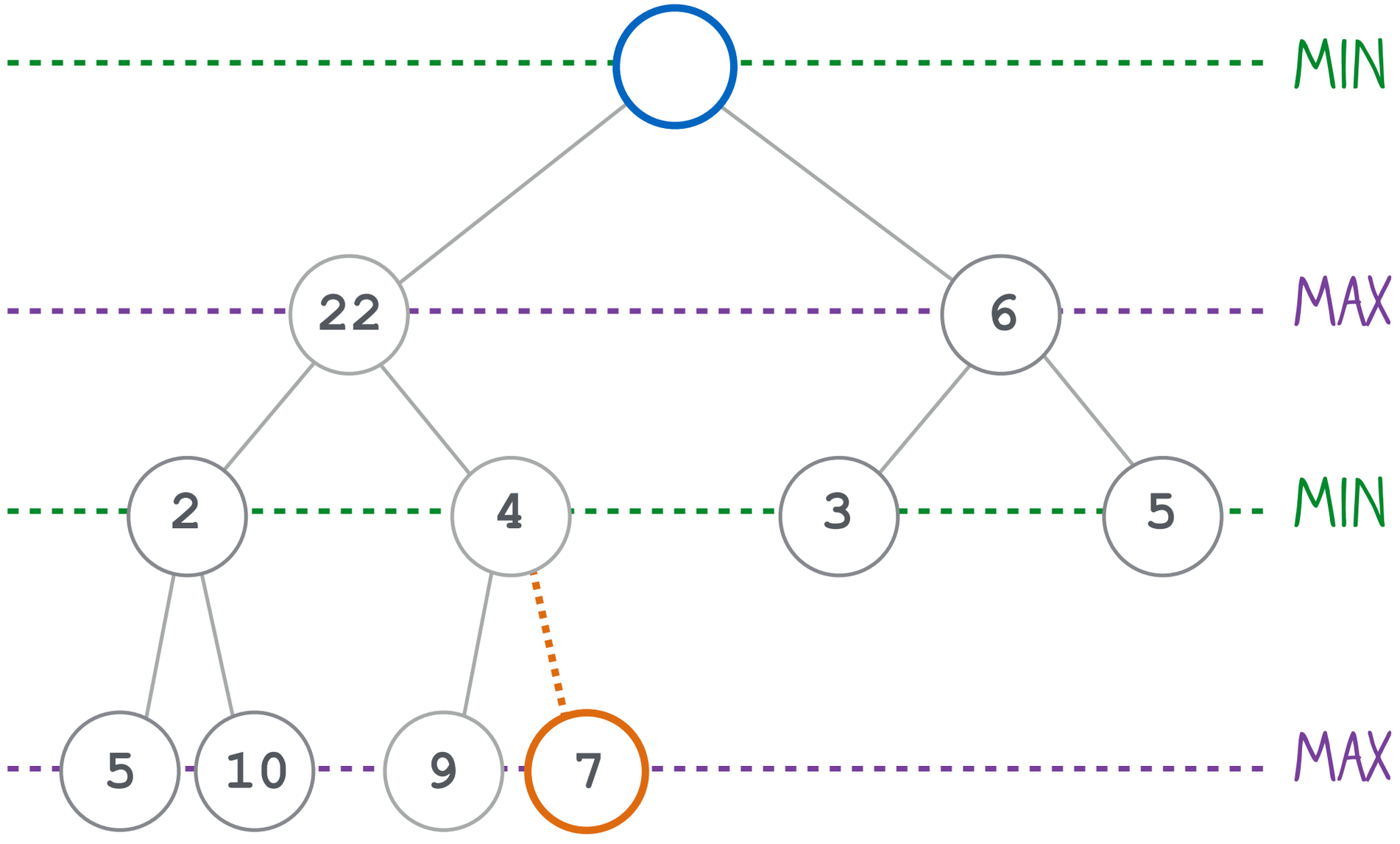
WANT TO DELETE THE MIN

REPLACE WITH THE LAST LEAF NODE



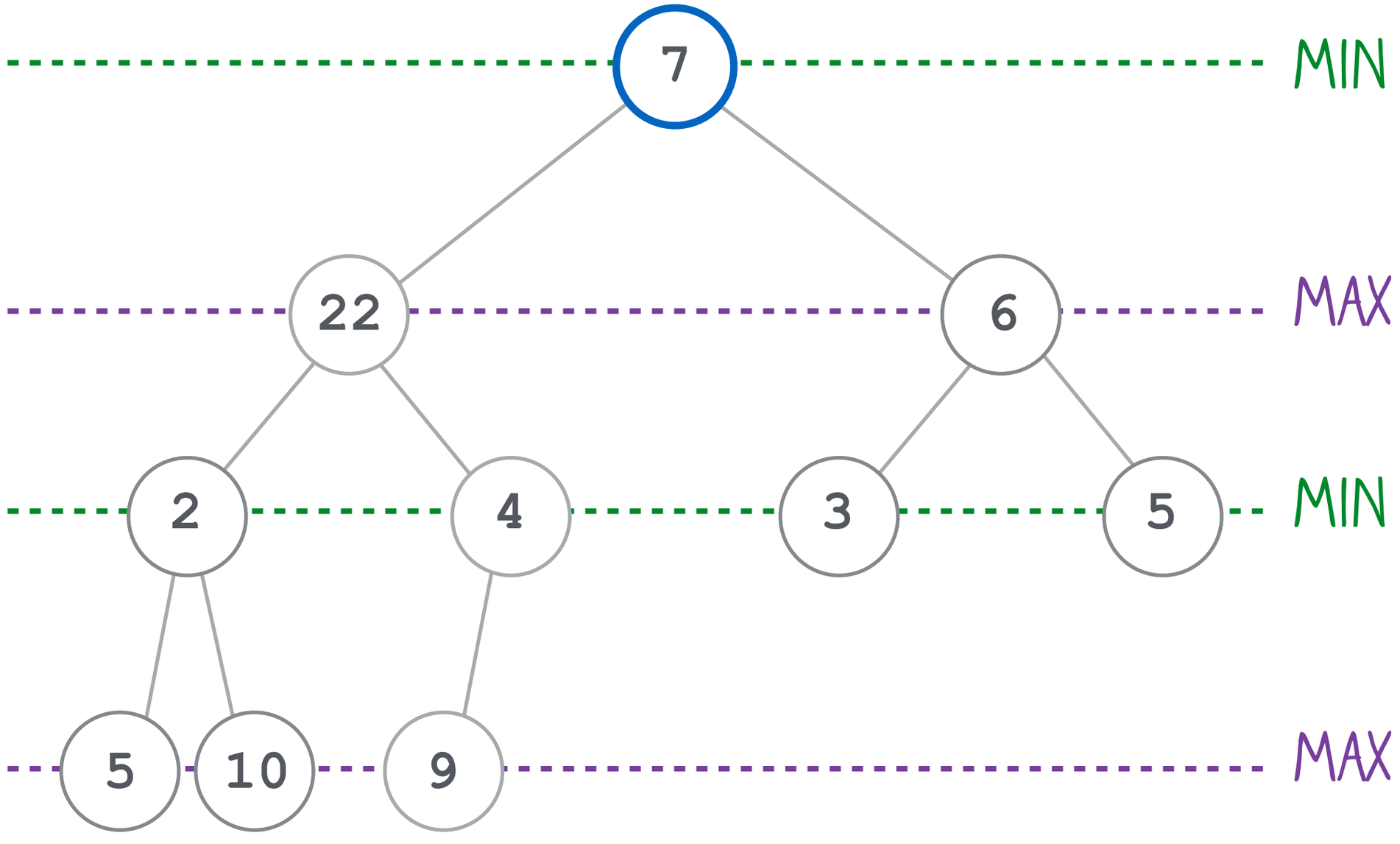
WANT TO DELETE THE MIN

REPLACE WITH THE LAST LEAF NODE

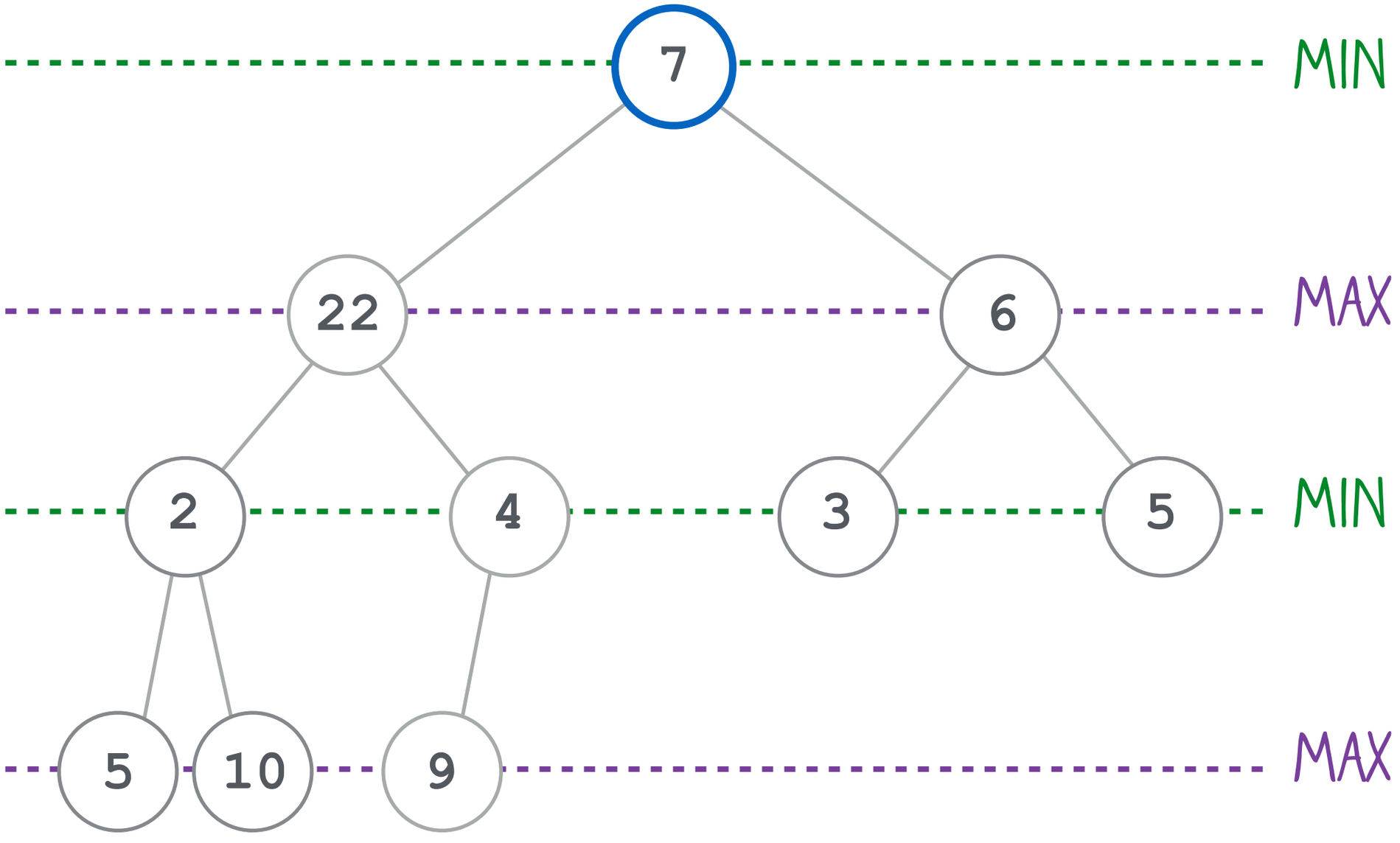


WANT TO DELETE THE MIN

REPLACE WITH THE LAST LEAF NODE

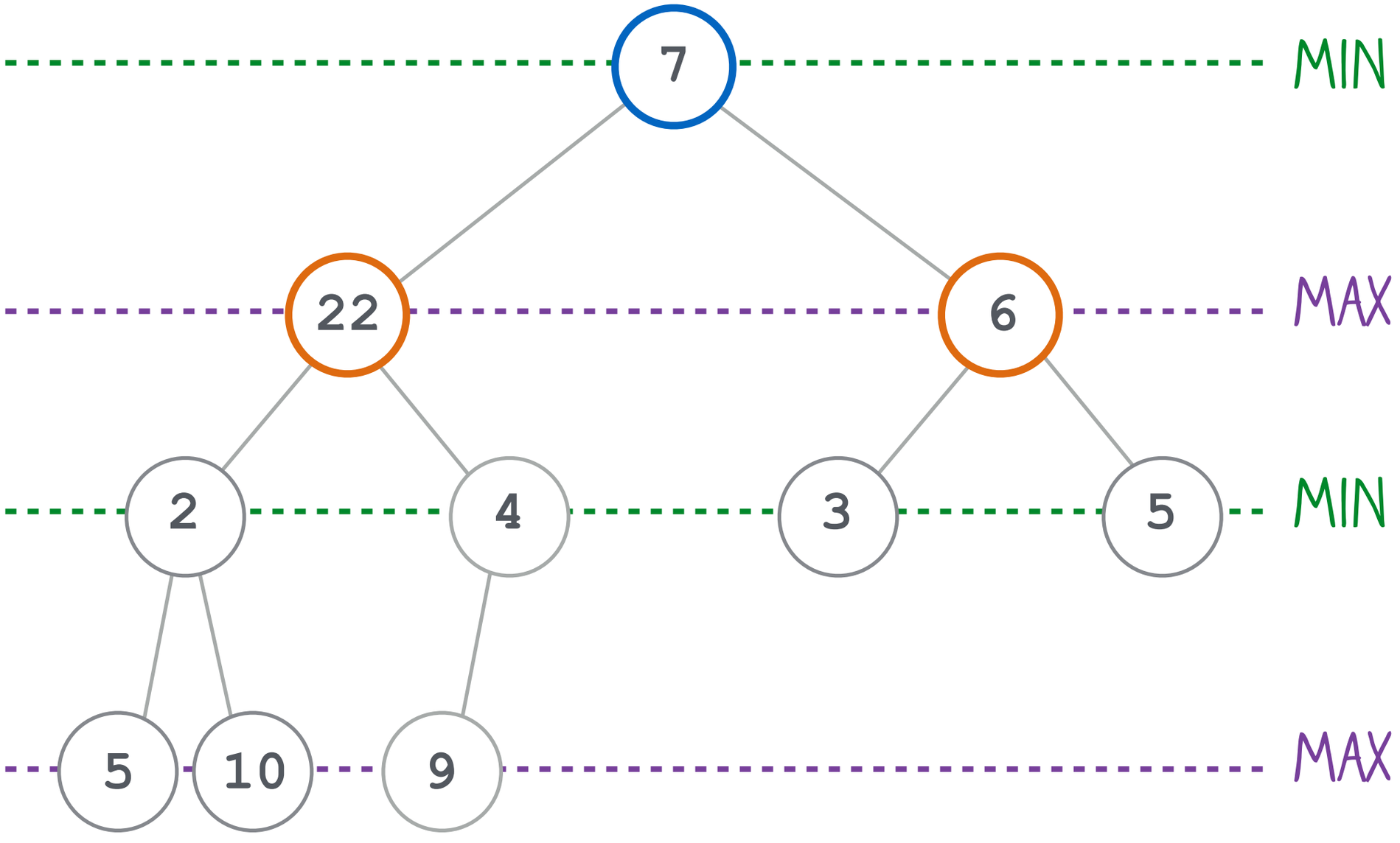


RESTORE ORDER WITH DIRECT CHILDREN



RESTORE ORDER WITH DIRECT CHILDREN

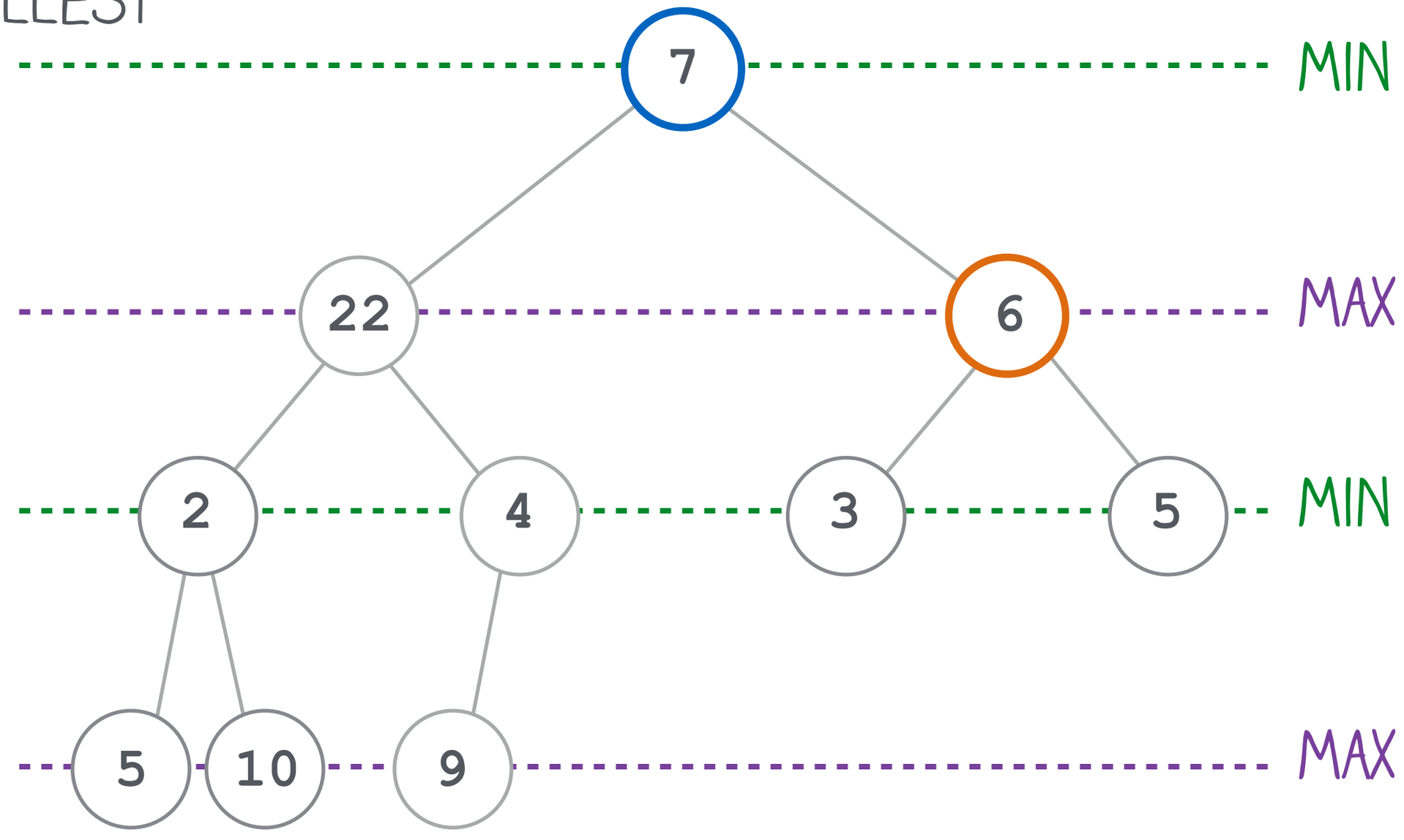
IS 7 < BOTH CHILDREN?



RESTORE ORDER WITH DIRECT CHILDREN

IS 7 < BOTH CHILDREN?

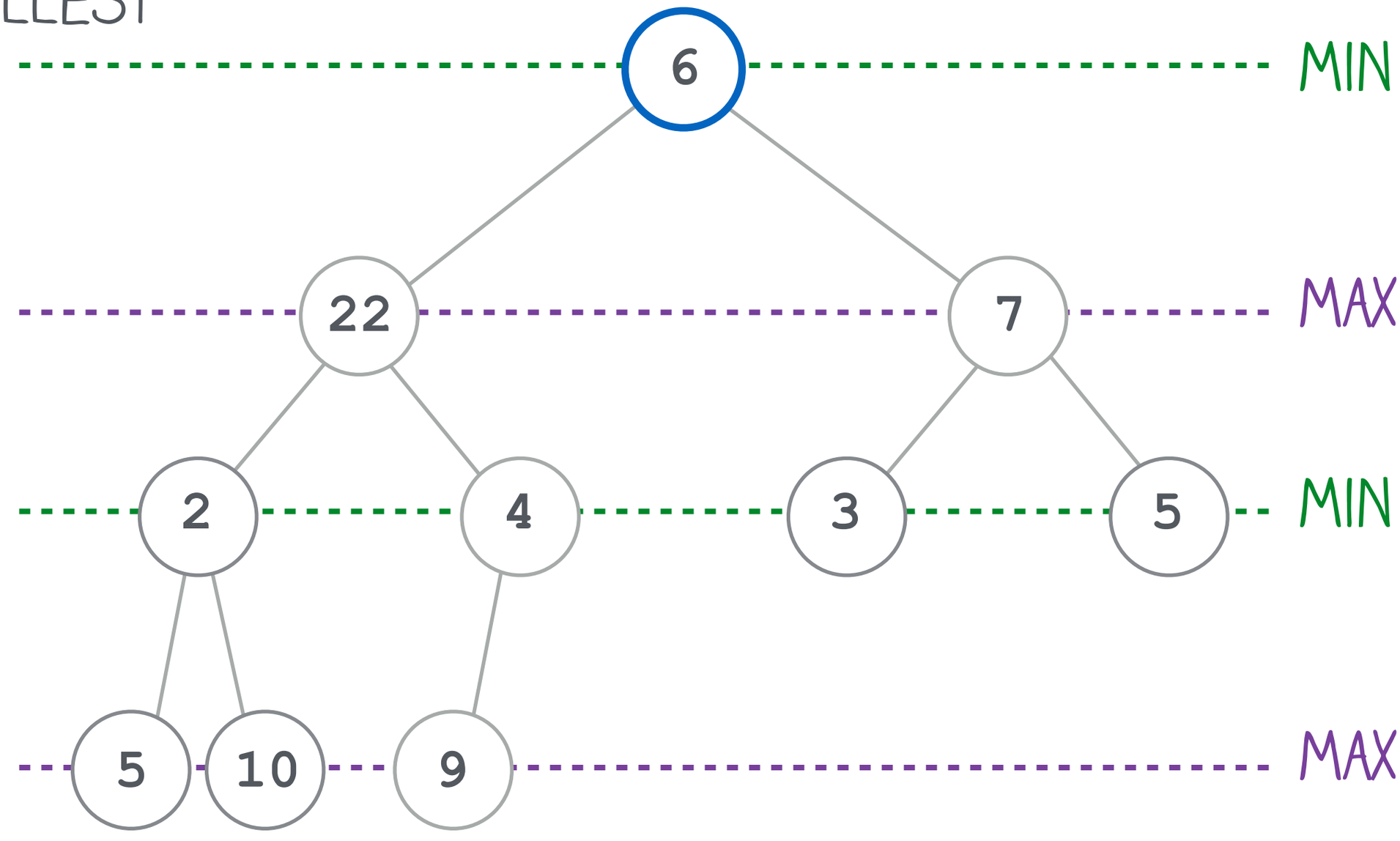
SWAP WITH SMALLEST



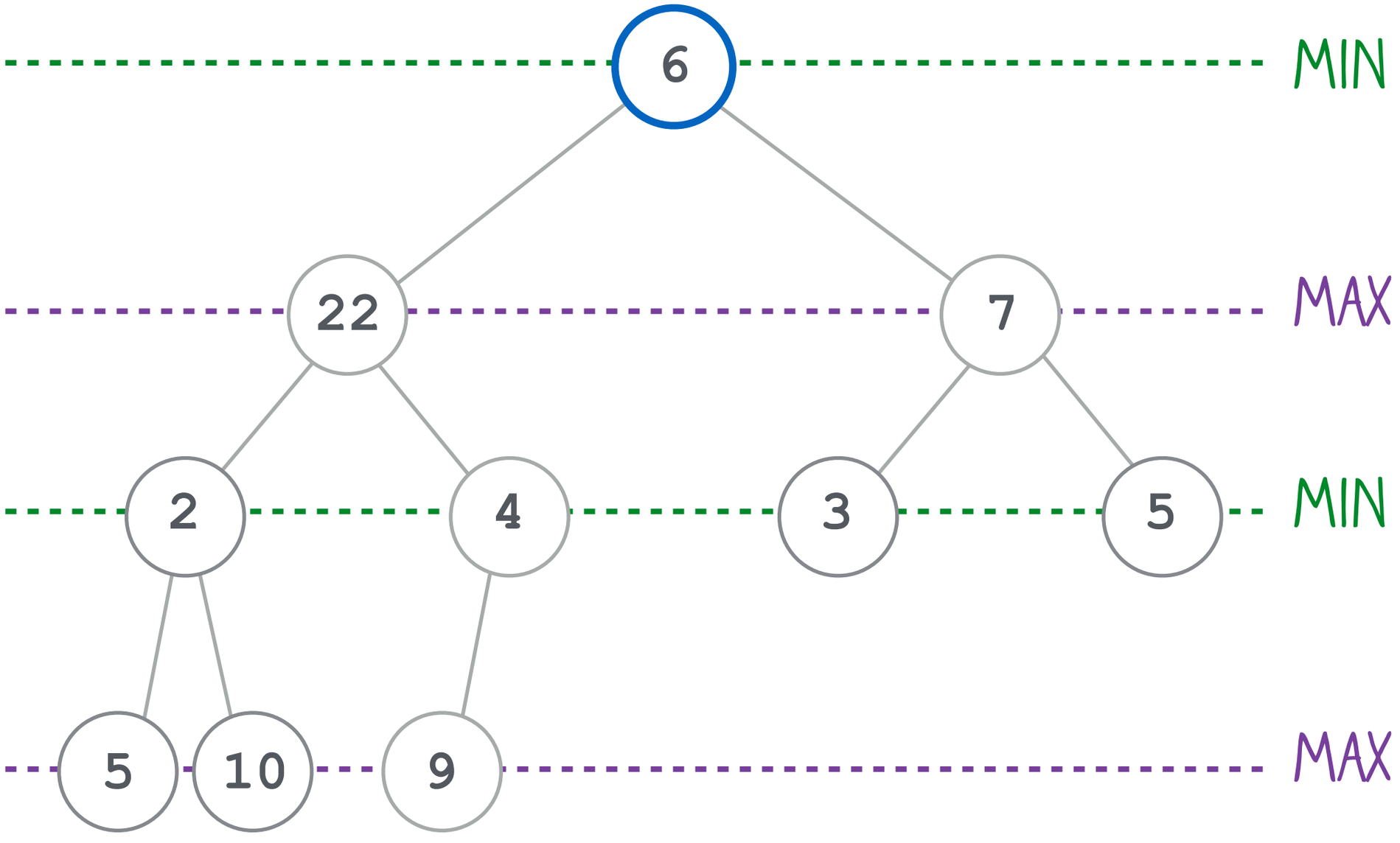
RESTORE ORDER WITH DIRECT CHILDREN

IS 7 < BOTH CHILDREN?

SWAP WITH SMALLEST



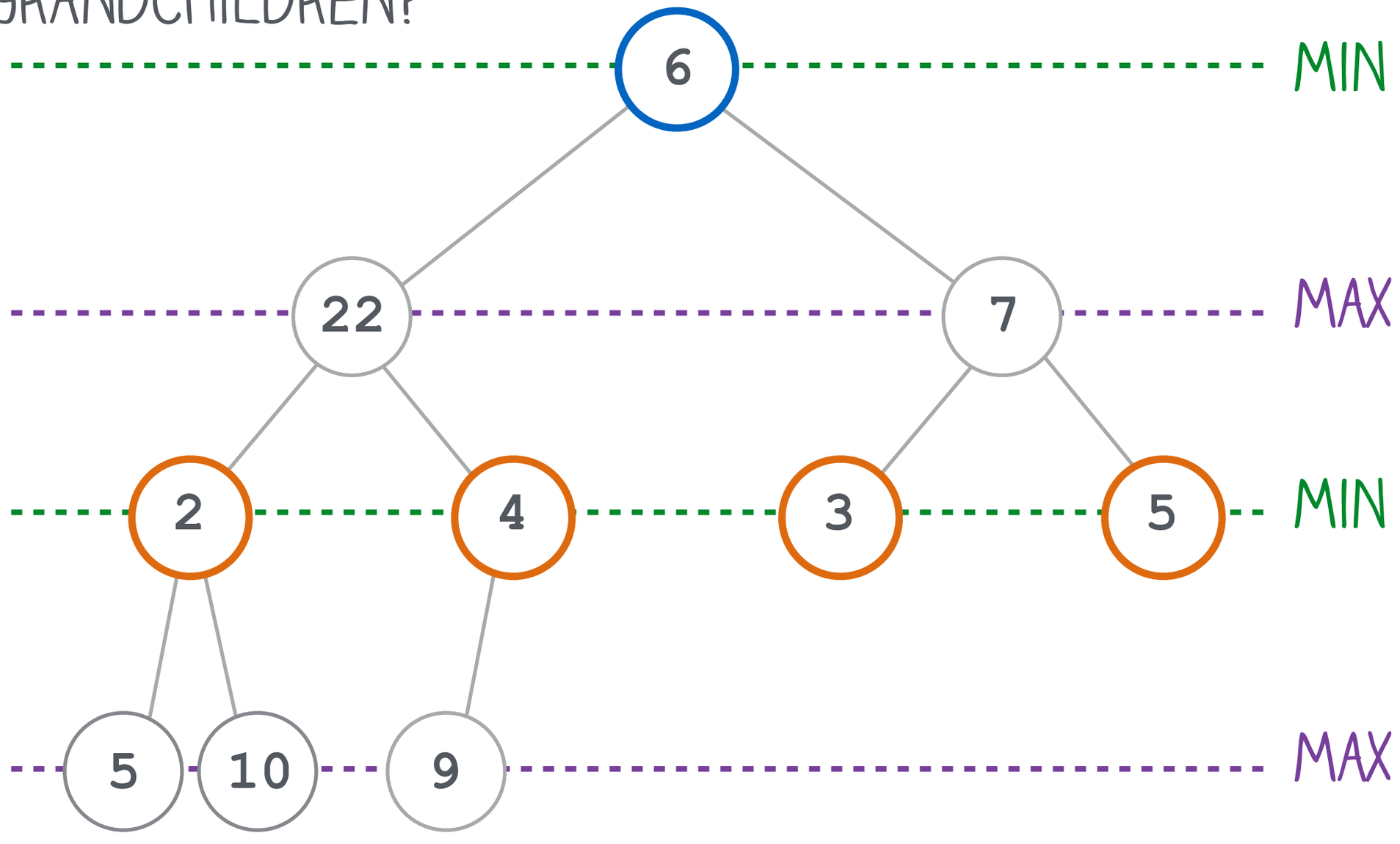
PERCOLATE 6 DOWN MIN LEVELS



PERCOLATE 6 DOWN MIN LEVELS

COMPARE TO ALL GRANDCHILDREN

IS 6 < ALL THE GRANDCHILDREN?

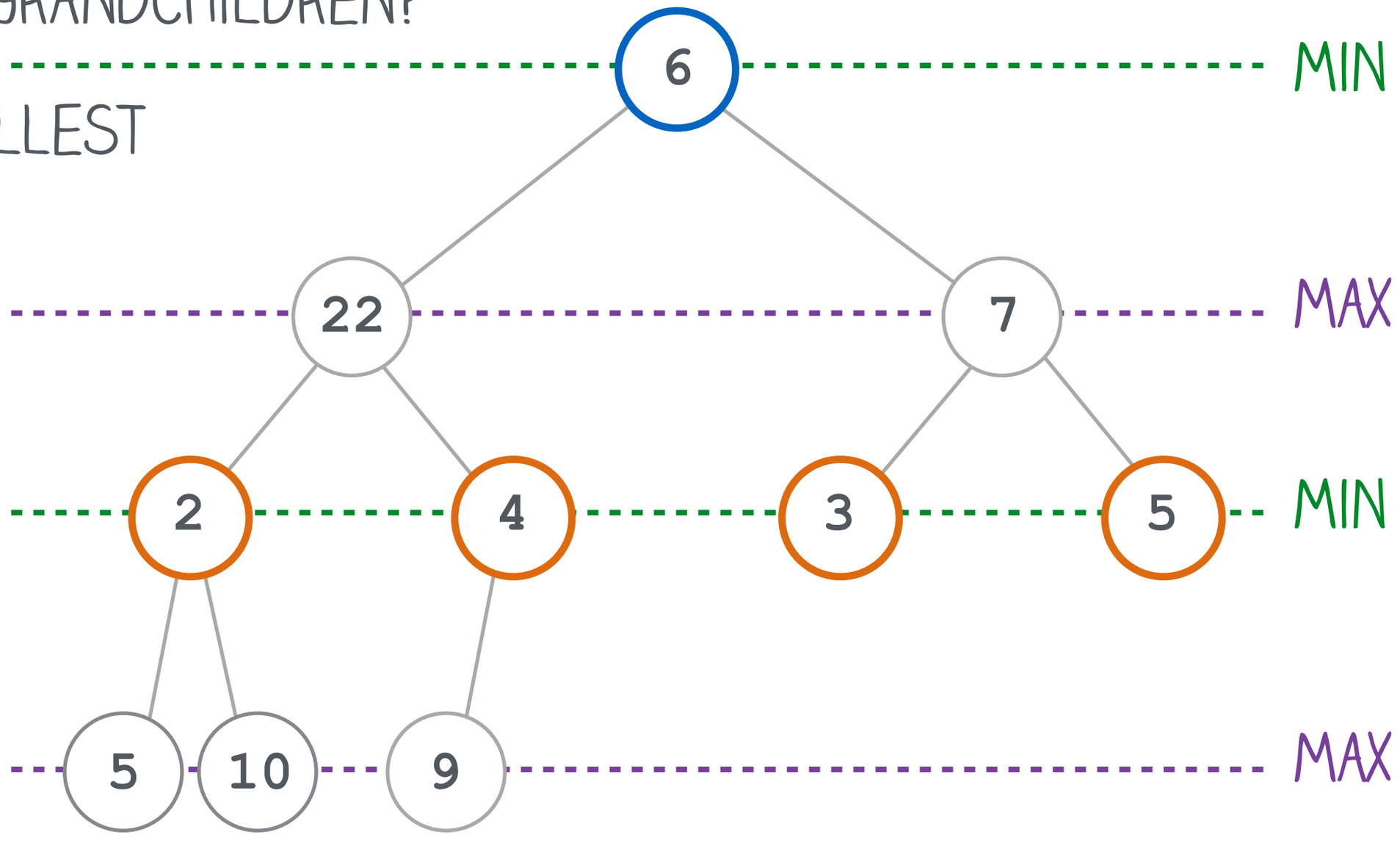


PERCOLATE 6 DOWN MIN LEVELS

COMPARE TO ALL GRANDCHILDREN

IS 6 < ALL THE GRANDCHILDREN?

SWAP WITH SMALLEST

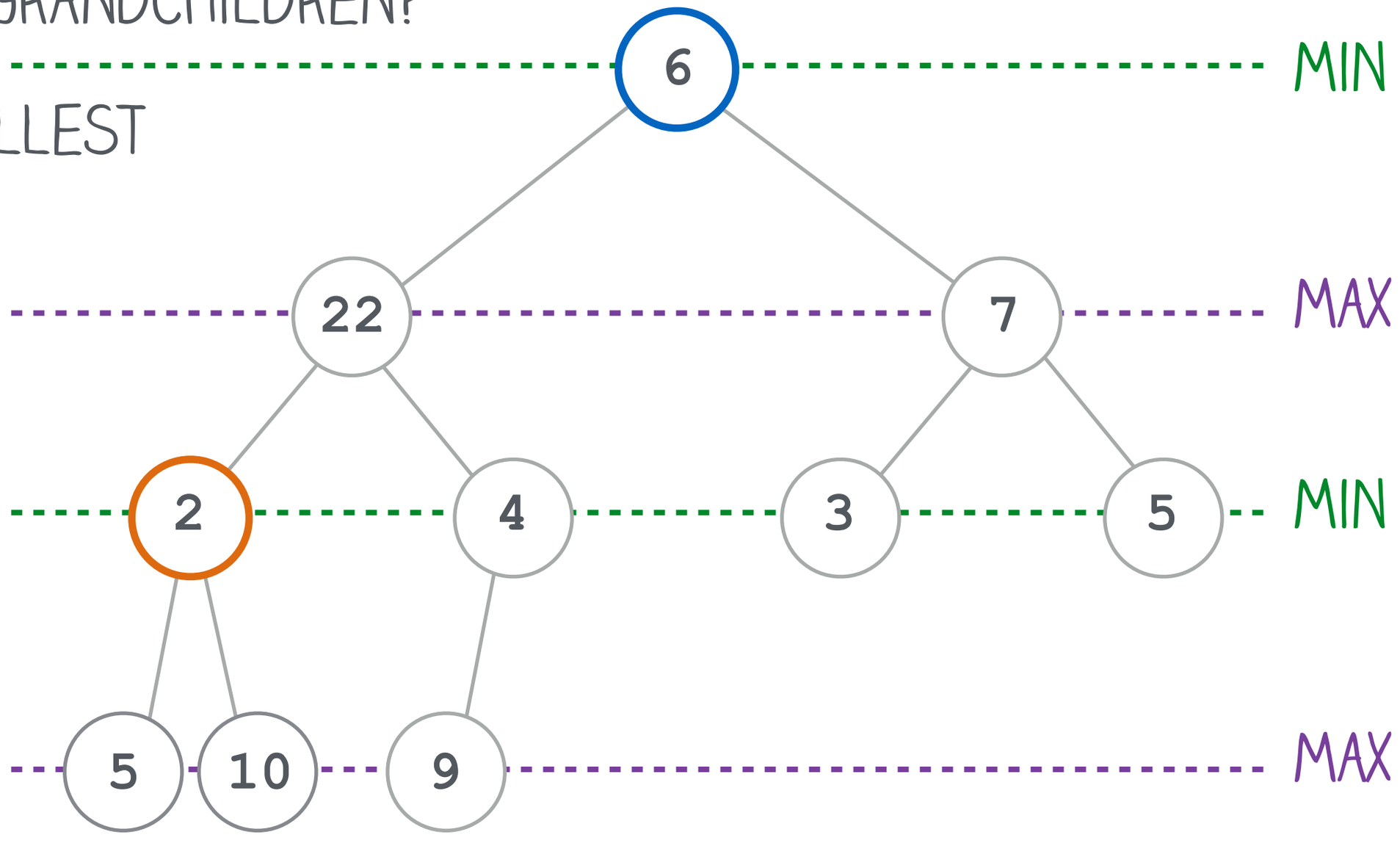


PERCOLATE 6 DOWN MIN LEVELS

COMPARE TO ALL GRANDCHILDREN

IS 6 < ALL THE GRANDCHILDREN?

SWAP WITH SMALLEST

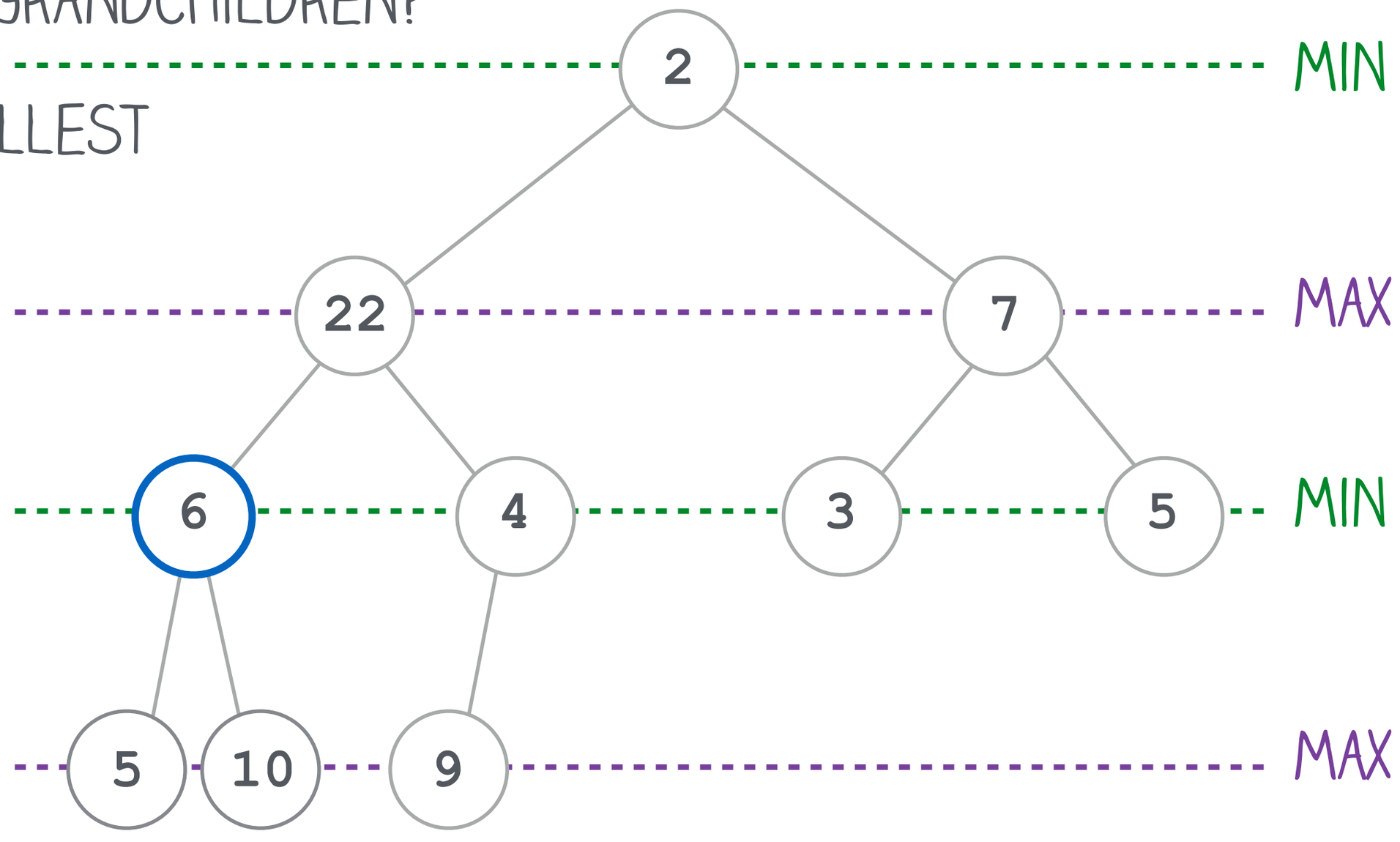


PERCOLATE 6 DOWN MIN LEVELS

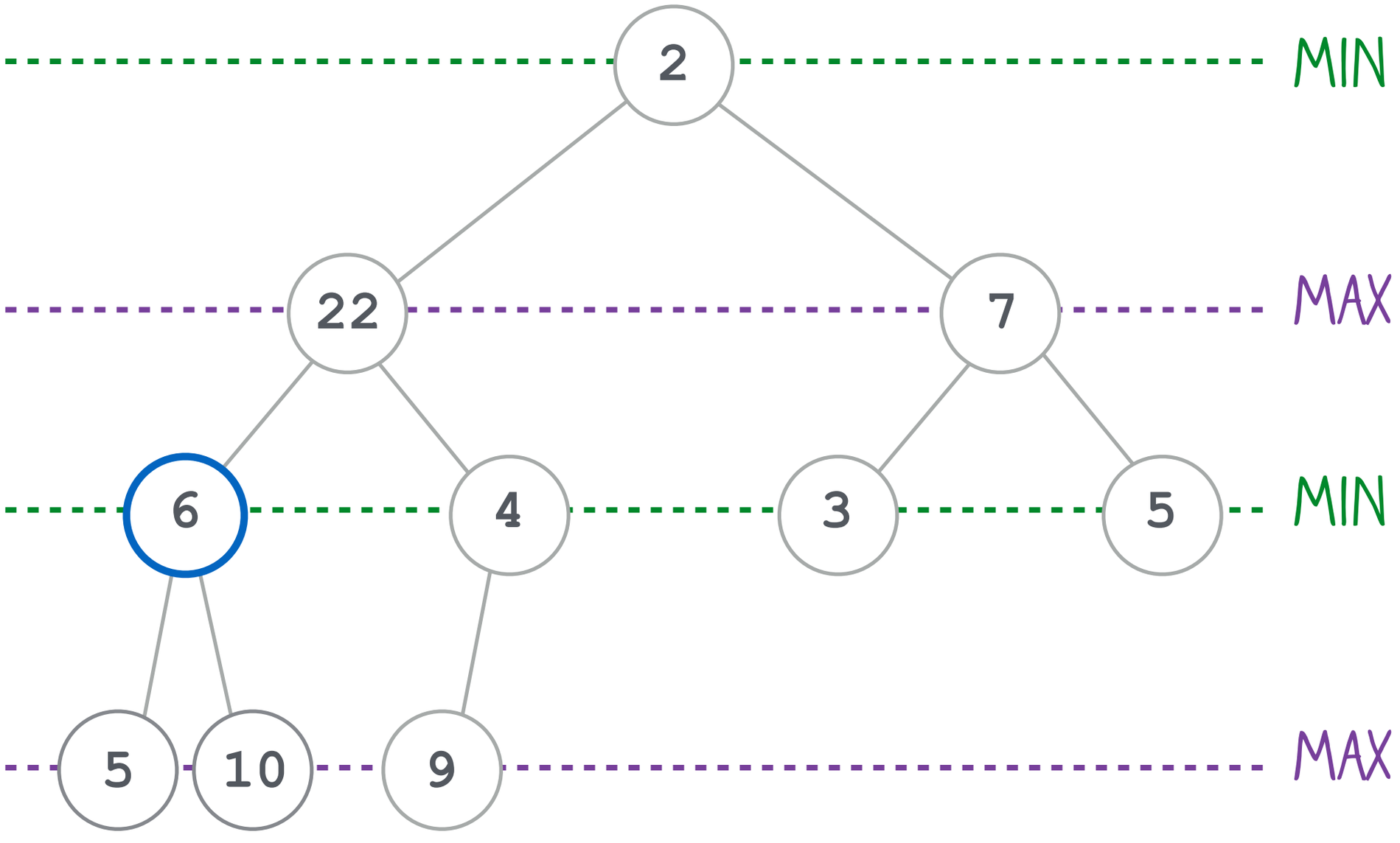
COMPARE TO ALL GRANDCHILDREN

IS 6 < ALL THE GRANDCHILDREN?

SWAP WITH SMALLEST

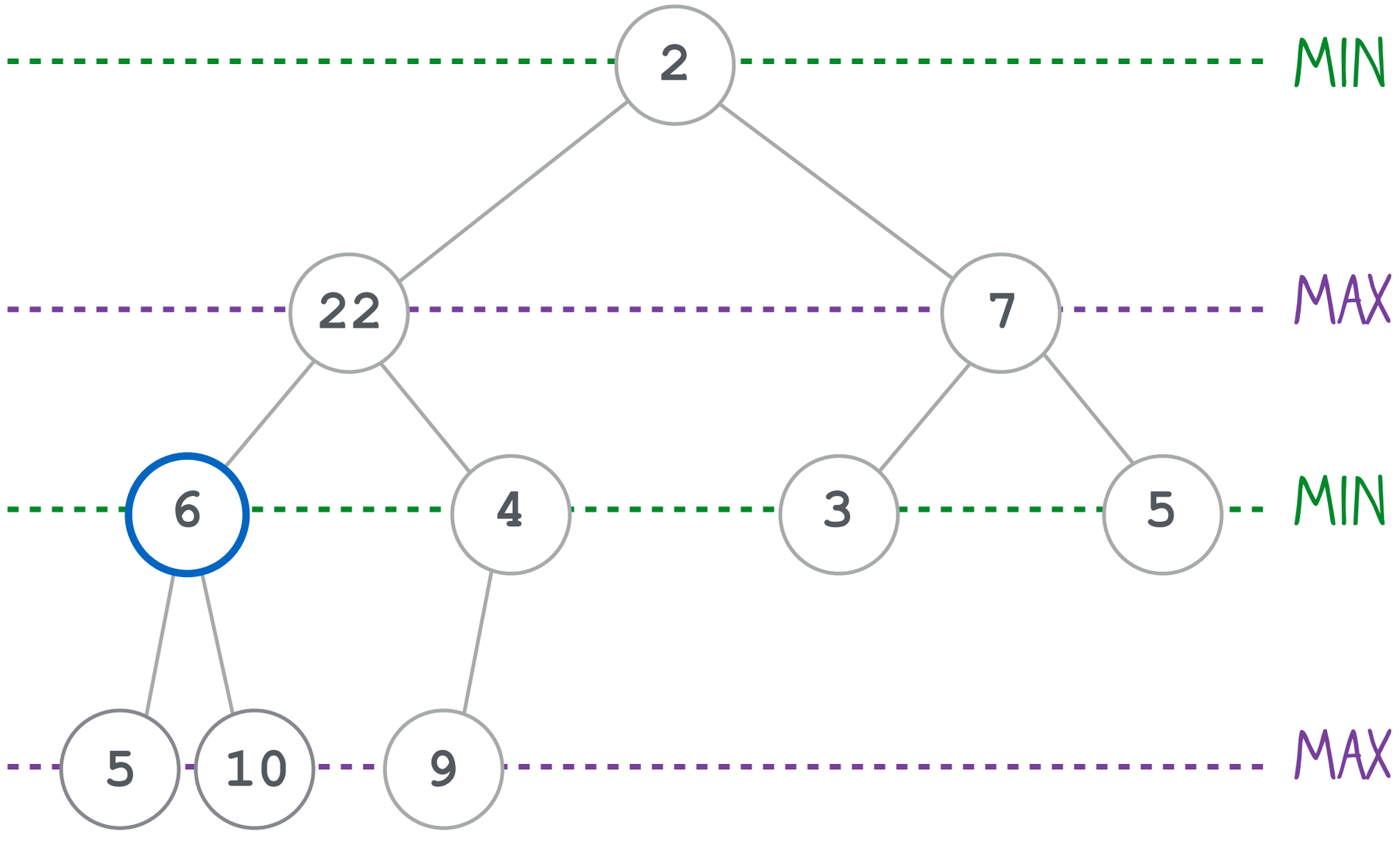


DOES 6 HAVE GRANDCHILDREN?



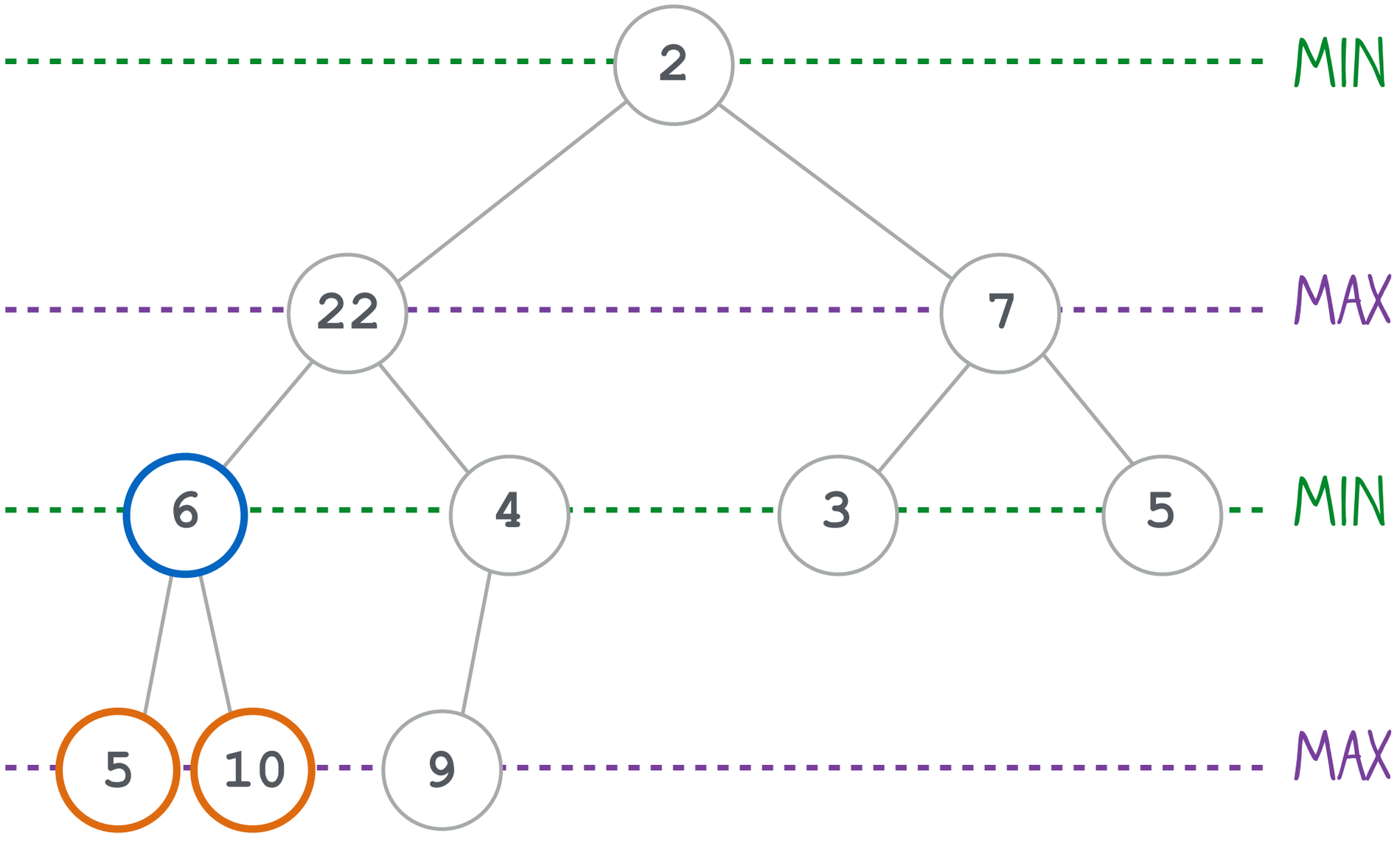
6 HAS NO GRANDCHILDREN, BUT IS NOT A LEAF NODE

COMPARE WITH DIRECT CHILDREN TO ENSURE ORDER PROPERTY



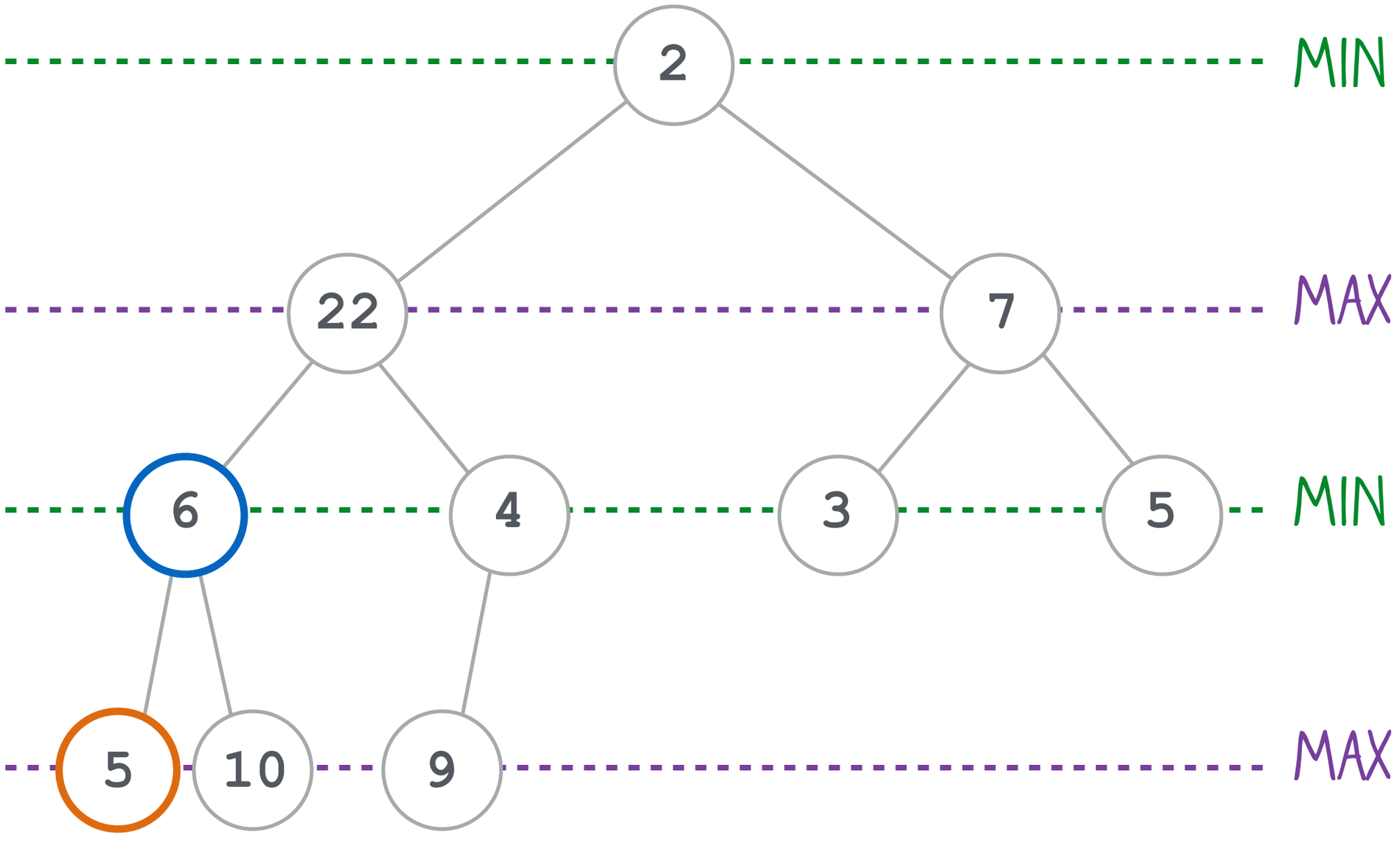
6 IS ON A MIN LEVEL, SO IT MUST BE SMALLER THAN CHILDREN

SWAP WITH SMALLEST



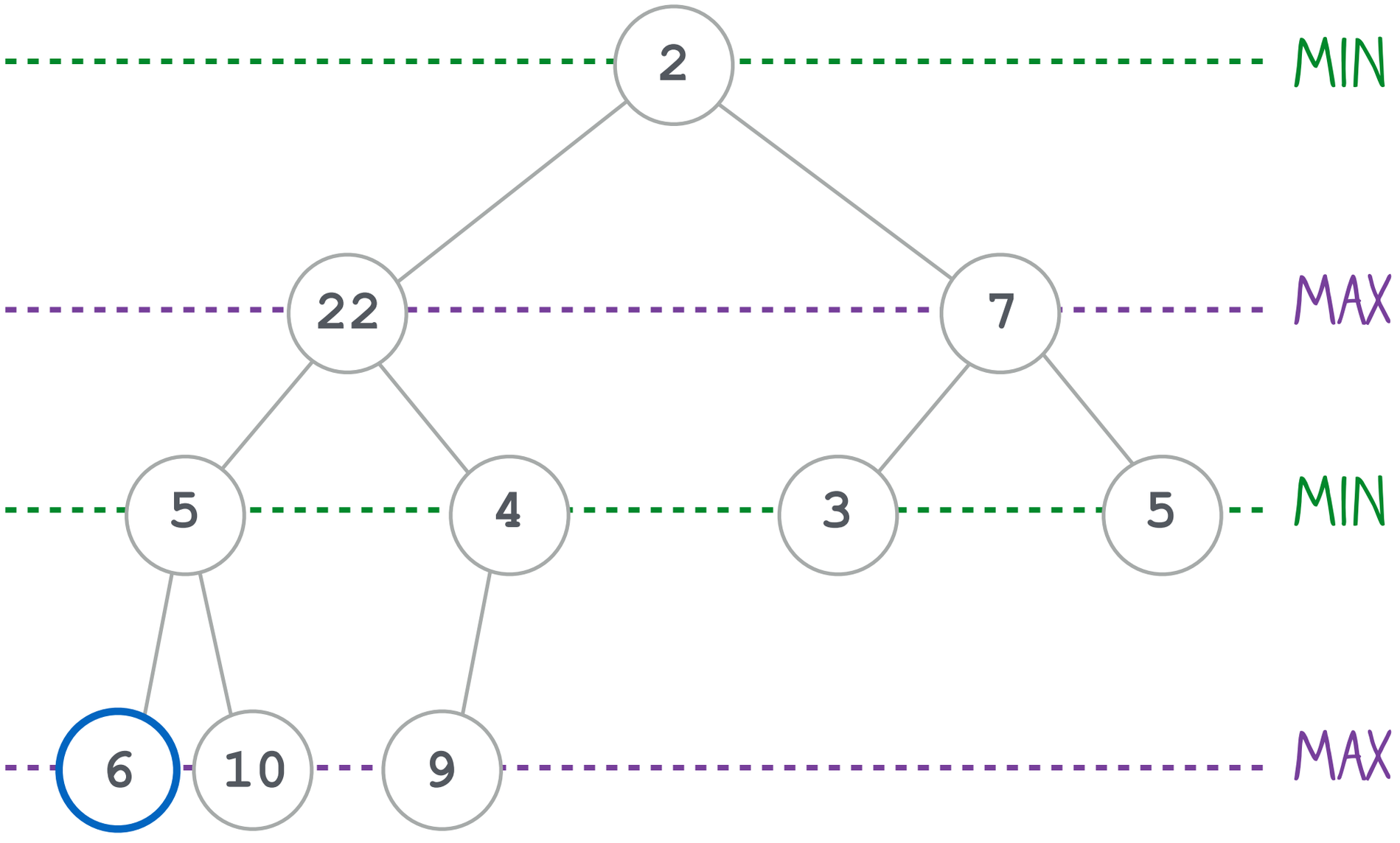
6 IS ON A MIN LEVEL, SO IT MUST BE SMALLER THAN CHILDREN

SWAP WITH SMALLEST



6 IS ON A MIN LEVEL, SO IT MUST BE SMALLER THAN CHILDREN

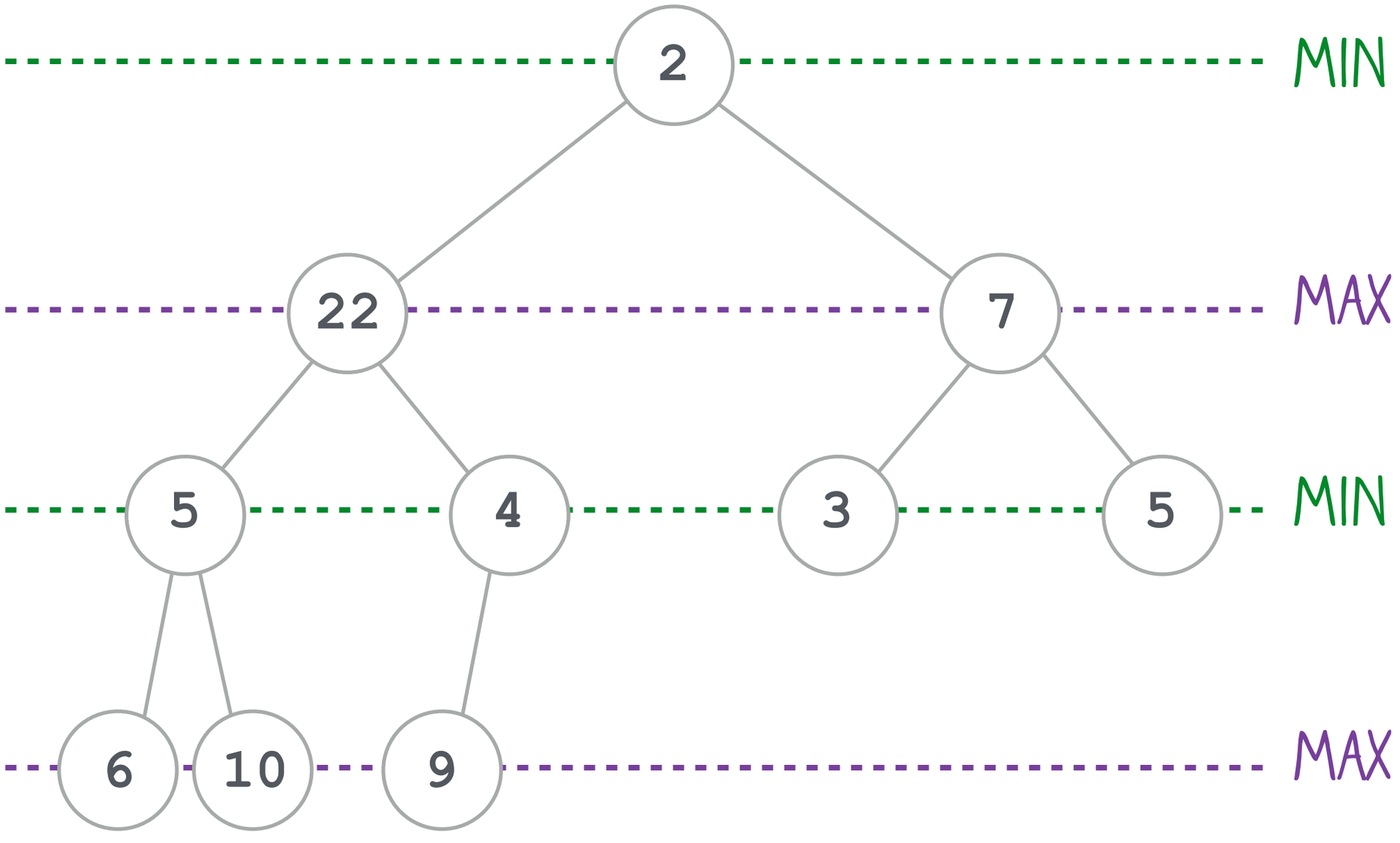
SWAP WITH SMALLEST



6 IS ON A MIN LEVEL, SO IT MUST BE SMALLER THAN CHILDREN

SWAP WITH SMALLEST

DONE!



delete algorithm

delete max (min is analogous)

1. locate node X (node containing max item)

2. replace X with last node in tree (last index in array!)

3. determine if new X is violating order property with direct children
   - if so, swap contents of X with the largest child

4. percolate new item X down max levels

5. if lowest max level reached, restore order with lowest min level (if applicable)

doubly-ended priority queue

-add

-findMin

-findMax

-deleteMin

-deleteMax

-min AND max items can be found in constant time with a min-max heap

-BUT… lots of special cases

  -tutorial on website with diagrams and step-by-step explanations

next time…

-reading

  -chapter 12 in book

-homework

  -assignment 10 due tonight

  -assignment 11 is out