BINARY HEAP

cs2420 | Introduction to Algorithms and Data Structures | Spring 2015

administrivia…

-assignment 10 is due on Thursday

-midterm grades out tomorrow

last time…

-a **hash table** is a general storage data structure

-insertion, deletion, and look-up are all **O(c)**

-like a stack, but not limited to top item

	Access	Insertion	Deletion	Notes
Hash Table	Constant	Constant	Constant	Magic?

-underlying data structure is just an array

-requires that all data types inserted have a hash function

-map the *hash value* to a valid index of the array using %

-use hash value to instantly look-up the index of any item

- -insertion, deletion, and search: **O(1)**
  - *-assuming the hash function is* **O(1)***!*

linear probing

-remember: *it is NOT required that two non-equal object have different hash values*

-because of this, it is possible for two different objects to has to the same index

- -this is called a **collision**

INSERT:

12, 15, 17, 46, 89, 90, 92

COLLISION! WHERE CAN WE PUT 92? (→ index 2)

array:	90		12			15	46	17		89
index:	0	1	2	3	4	5	6	7	8	9

clustering

-if an item's natural spot is taken, it goes in the next open spot, making a cluster for that hash

- -**clustering** happens because once there is a collision, there is a high probability that there will be more
- -this means that any item that hashes into the cluster will require several attempts to resolve the collision

-feedback loop:

- -the bigger the clusters are, the more likely they are to be hit
- -when a cluster gets hit, it gets bigger

quadratic probing

-quadratic probing attempts to deal with the clustering problem

-if `hash(item) = H`, and the cell at `H` is occupied:

- -try $H+1^2$
- -then $H+2^2$
- -then $H+3^2$
- -and so on…
- -wrap around to beginning of array if necessary

separate chaining

-why not make each spot in the array capable of holding more than one item?

- -use an array of linked lists
- -hash function selects index into array
- -called **separate chaining**

-for insertion, append the item to the end of the list

- -insertion is **O(1)** if we have what?

-searching is a linear scan through the list

- -fast if the list is short

a bit more on hash functions…

-ints have an obvious hash value

array:	90		12			15	46	17		89
index:	0	1	2	3	4	5	6	7	8	9

-what about Strings? Books? Shapes?…

-we must not overlook the requirement of a *good* hash functions

remember...

-hash functions take any item as input and produce an integer as output

-given the same input the function always returns the same output

-two different inputs MAY have the same hash value

thinking about `chars` and `Strings`

- ASCII defines an encoding for characters
  - ‘a’ = 97
  - ‘b’ = 98
  - …
  - ‘z’ = 122
  - ‘2’ = 50
  - …

- the `char` type is actually just a small integer
  - 8 bits instead of the usual 32

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

-a `String` is essentially an array of `char`

```
String s = "hello";
```

104	**101**	**108**	**108**	**111**

-Java hides these details

-how can we use this to create a hash function for `Strings`?

review

- **O(1)** for all major operations
  - assuming λ is managed

- linear probing
  - has clustering problems

- quadratic probing
  - has lesser clustering problems
  - requires $\lambda < 0.5$, and prime table size

- separate chaining
  - probably the easiest to implement, as well as the best performing

WHAT IS THE LOAD FACTOR λ FOR THE FOLLOWING HASH TABLE?

A) **4**
B) **6**
C) **0.4**
D) **0.5**
E) **0.6**

104	34		19	111	98		52		

USING LINEAR PROBING, IN WHAT INDEX WILL ITEM 93 BE ADDED?

A) **1**
B) **5**
C) **6**
D) **7**

array:	49		9	58	34				18	89
index:	0	1	2	3	4	5	6	7	8	9

USING QUADRATIC PROBING, IN WHAT INDEX WILL ITEM 22 BE ADDED?

A) **1**
B) **5**
C) **6**
D) **7**

array:	49		9	58	34				18	89
index:	0	1	2	3	4	5	6	7	8	9

recap

-i heart hash tables

- -collection structure with O(1) for major operations

-but!…

- -hash function must minimize collisions
    - *-should evenly distribute values across all possible integers*
- -collisions must be carefully dealt with
- -hash function runtime must be fast

- -no ordering
    - *-how do we find the smallest item in a hash table?*
    - *-in a BST?*

priority queues

- 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?

- 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)**

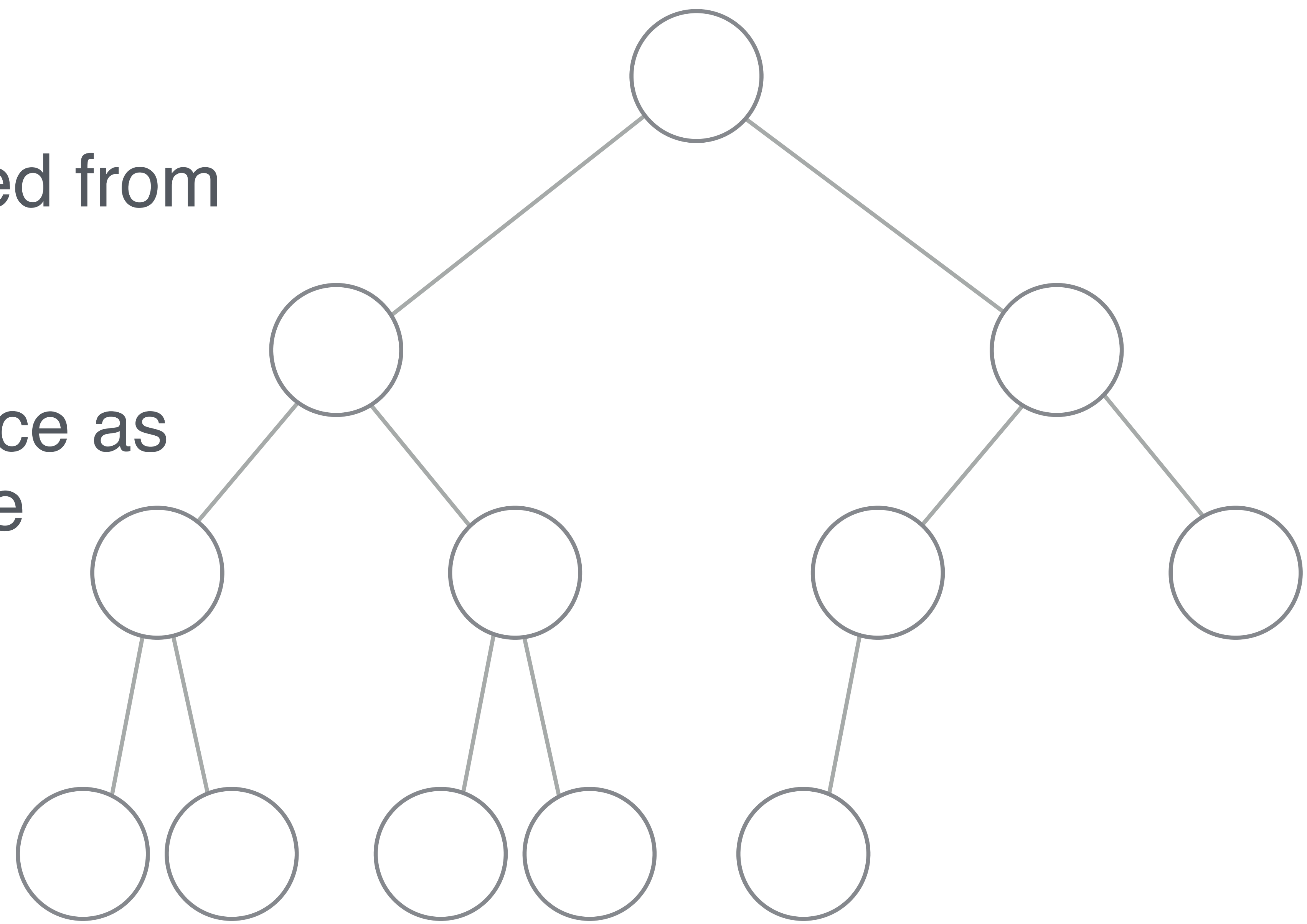
- option 3: a self-balancing BST
  - add: **O(logN)**
  - findMin: **O(logN)**
  - deleteMin: **O(logN)**

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

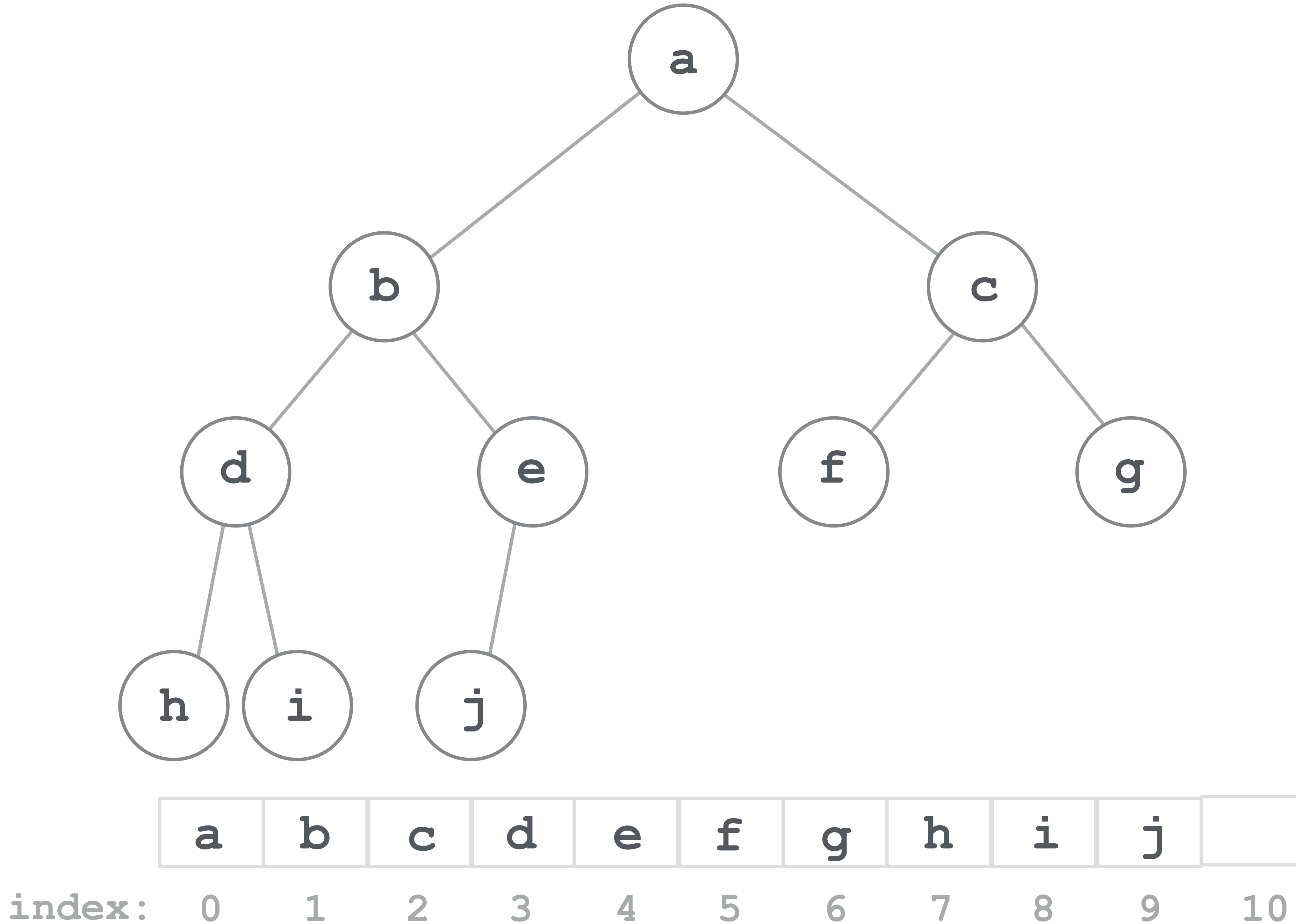


complete trees as an array

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

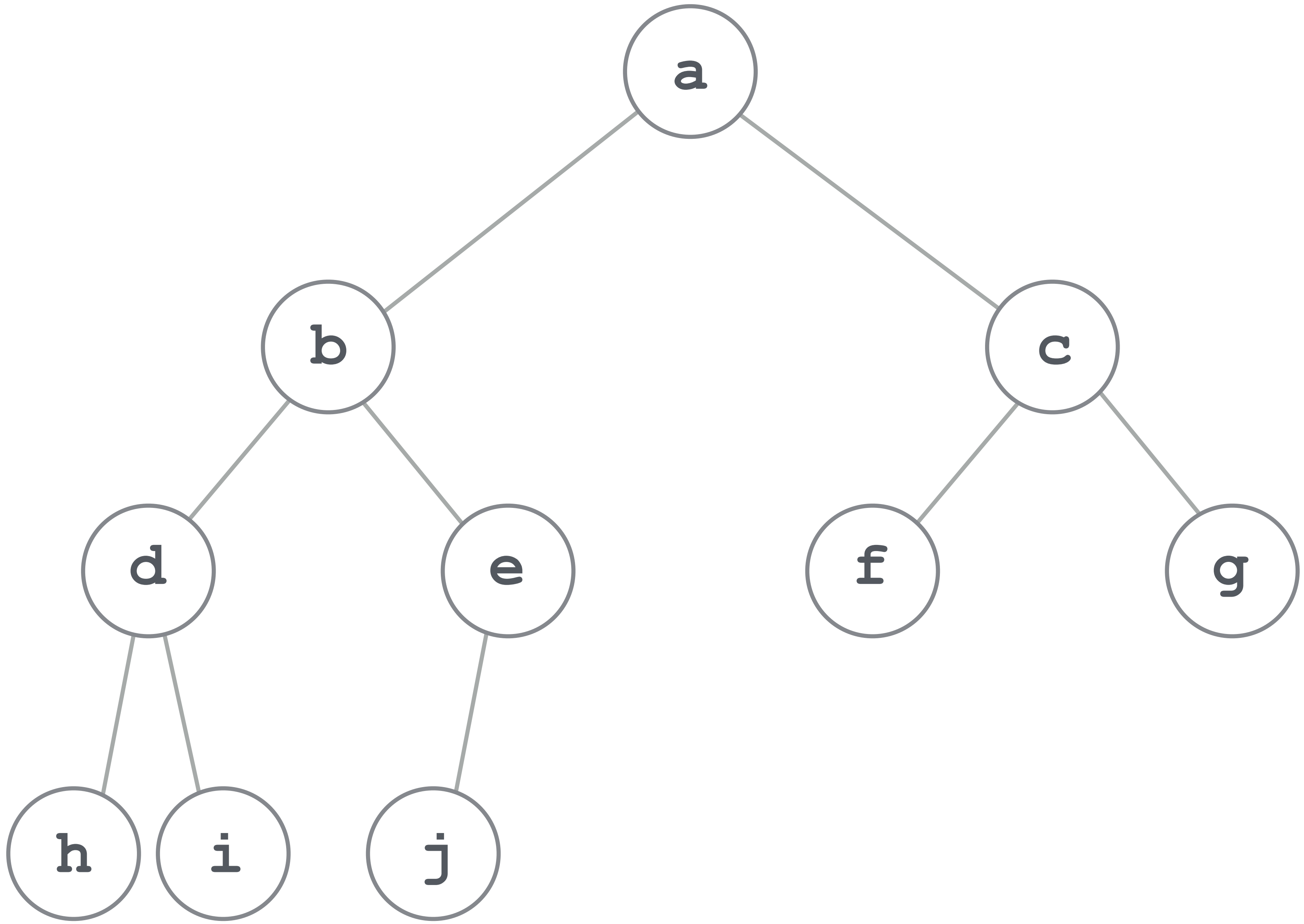
-the root goes at index 0, its left child at index 1, its right child at index 2

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



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

-how can we compute the index of *any* node’s parent?



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

-luckily, integer division automatically truncates

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

complete trees as an array

-keep track of a `currentSize` variable

- -holds the total number of nodes in the tree
- -the very last leaf of the bottom level will be at index `currentSize - 1`

-when computing the index of a child node, if that index is `>= currentSize`, then the child does not exist

traversal helper methods

```
int leftChildIndex(int i) {
  return (i*2) + 1;
}


int rightChildIndex(int i) {
  return (i*2) + 2;
}


int parentIndex(int i) {
  return (i-1) / 2;
}
```

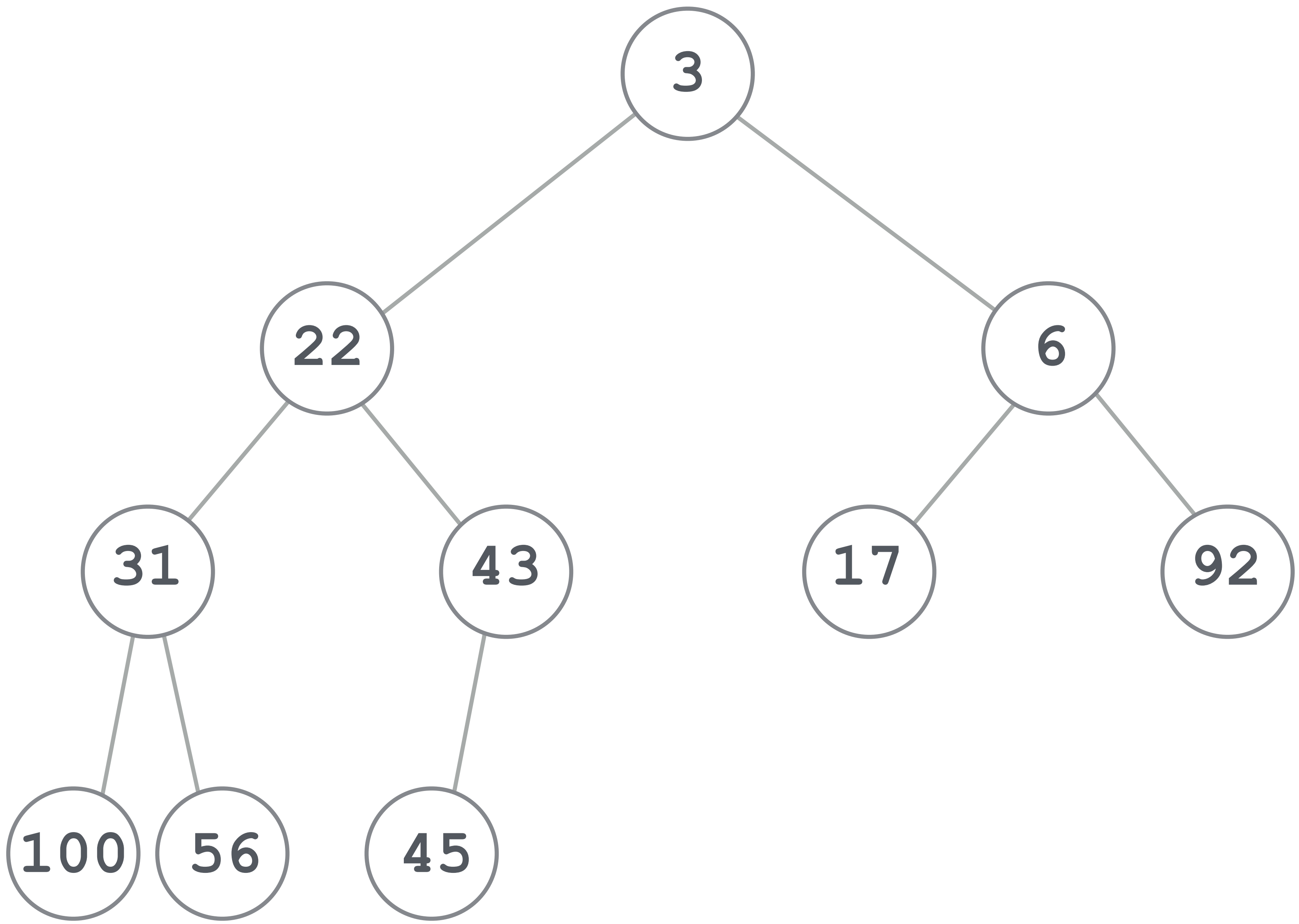
binary heap

-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**

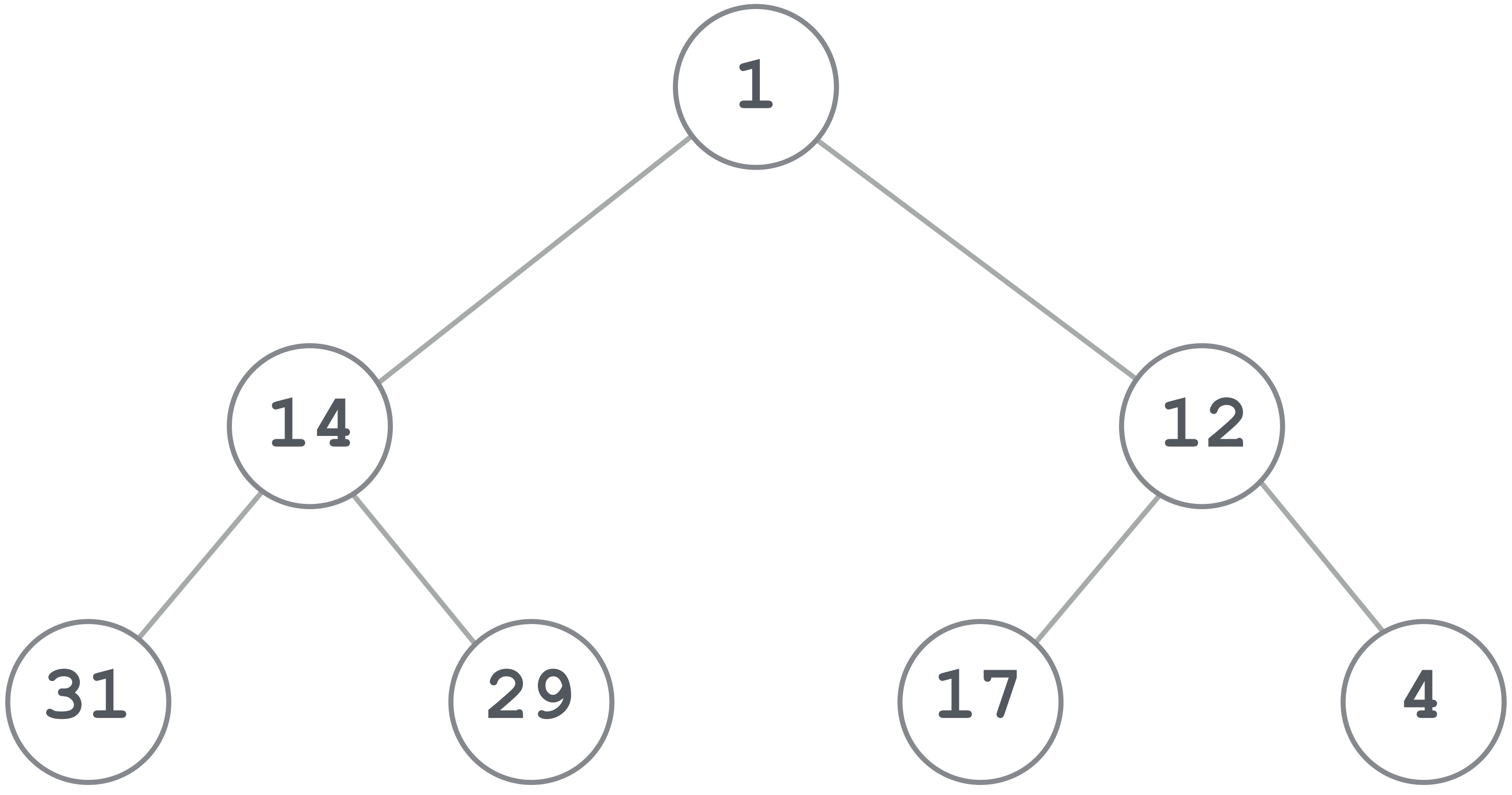
-a **max-heap** would have the opposite property



-order of children does not matter, only that they are greater than their parent

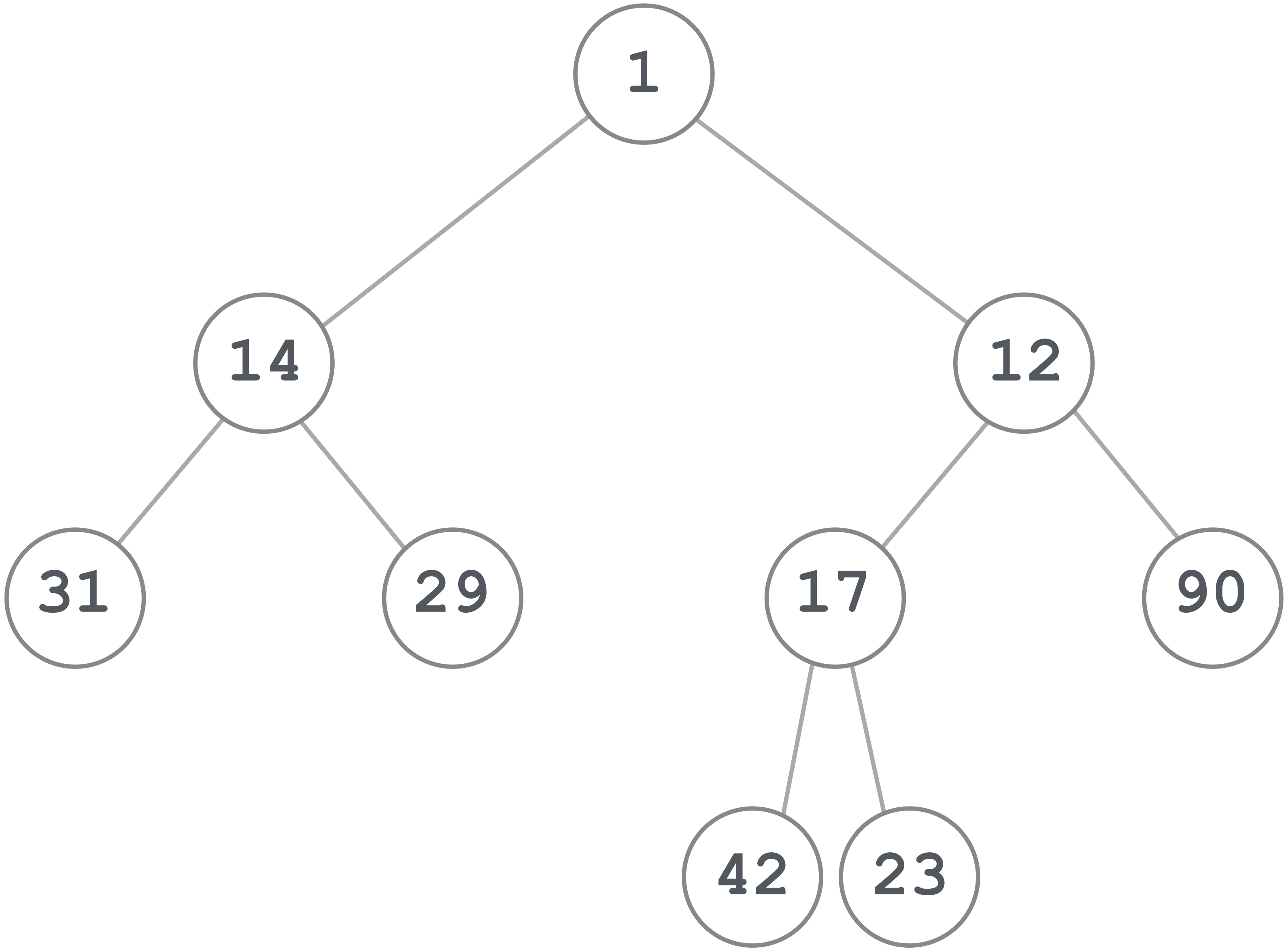
IS THIS A MIN-HEAP?

A) **yes**
B) **no**



IS THIS A MIN-HEAP?

A) **yes**
B) **no**



adding to a heap

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

- -structure and order

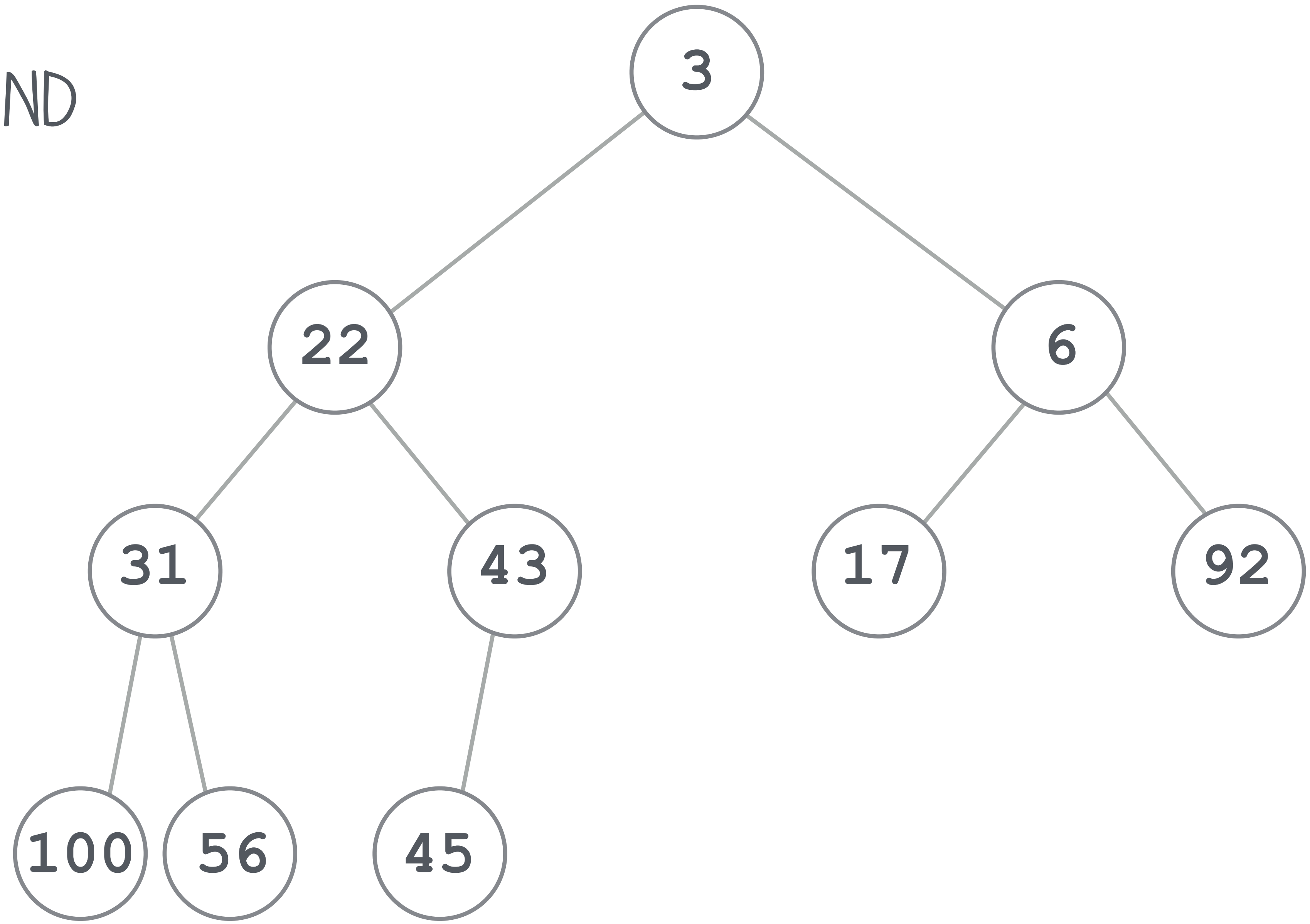
-deal with the structure property first… where can the new item go to maintain a complete tree?

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

- -swap upwards until > parent

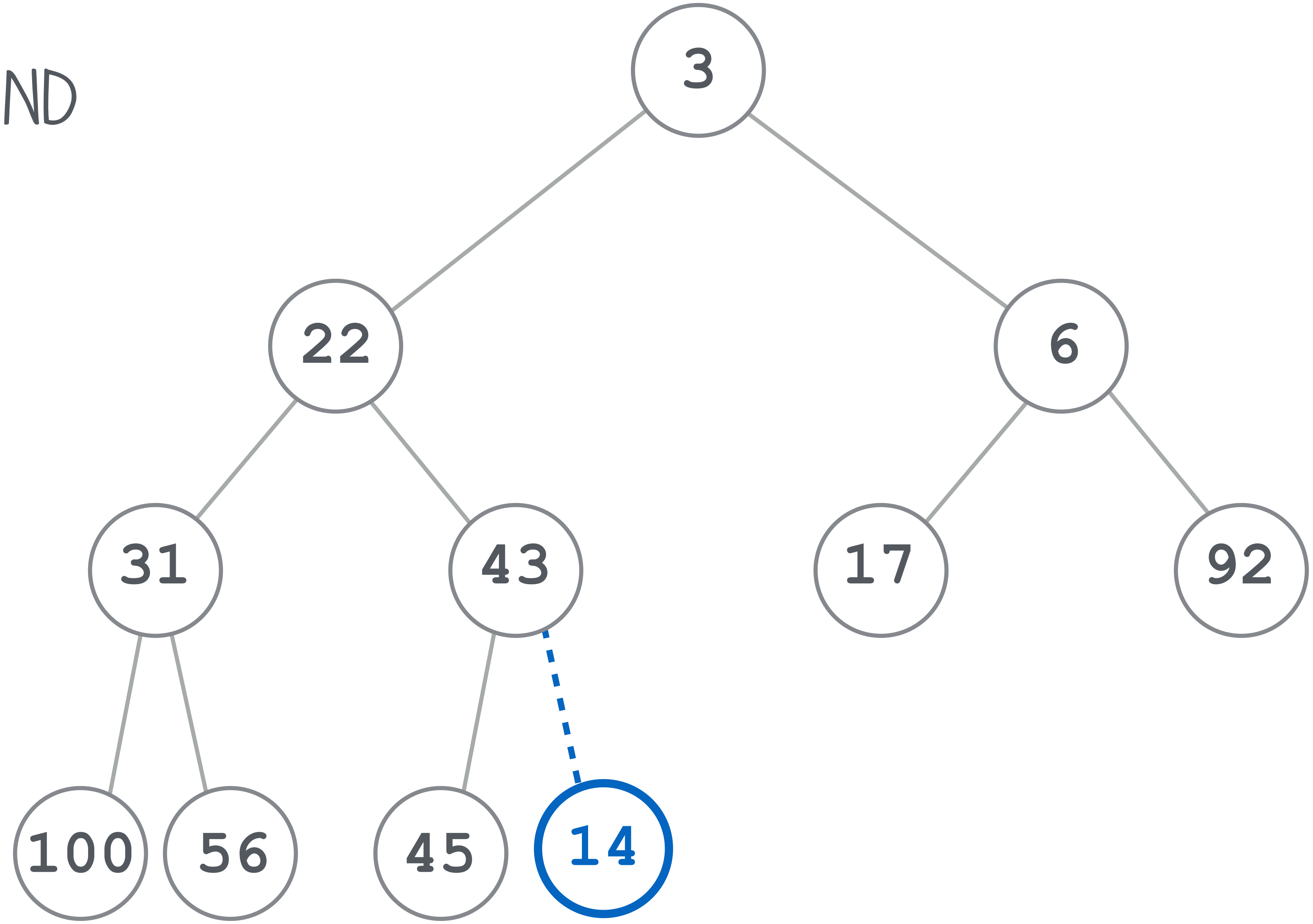
ADDING 14

PUT IT AT THE END OF THE TREE



ADDING 14

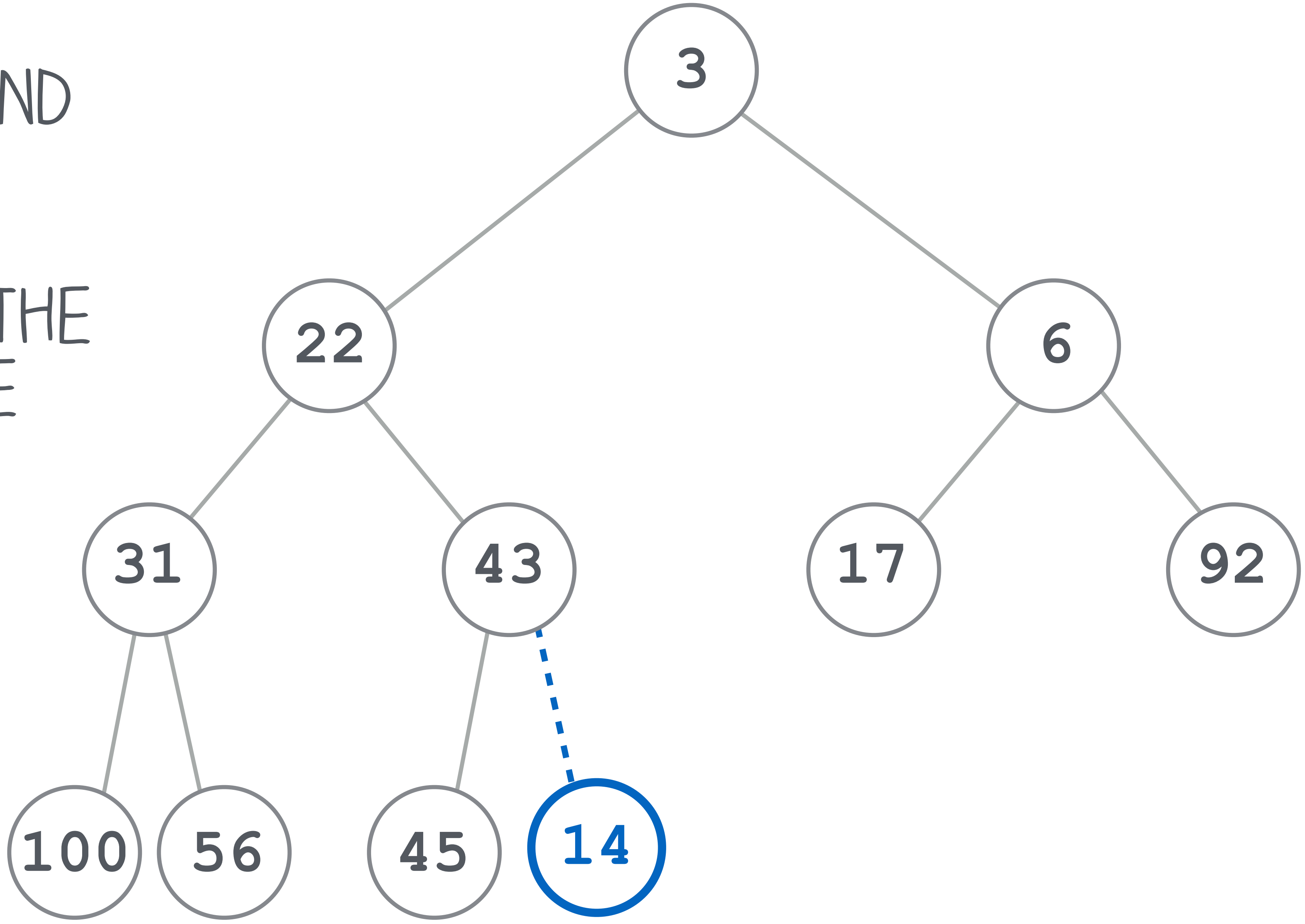
PUT IT AT THE END
OF THE TREE



ADDING 14

PUT IT AT THE END OF THE TREE

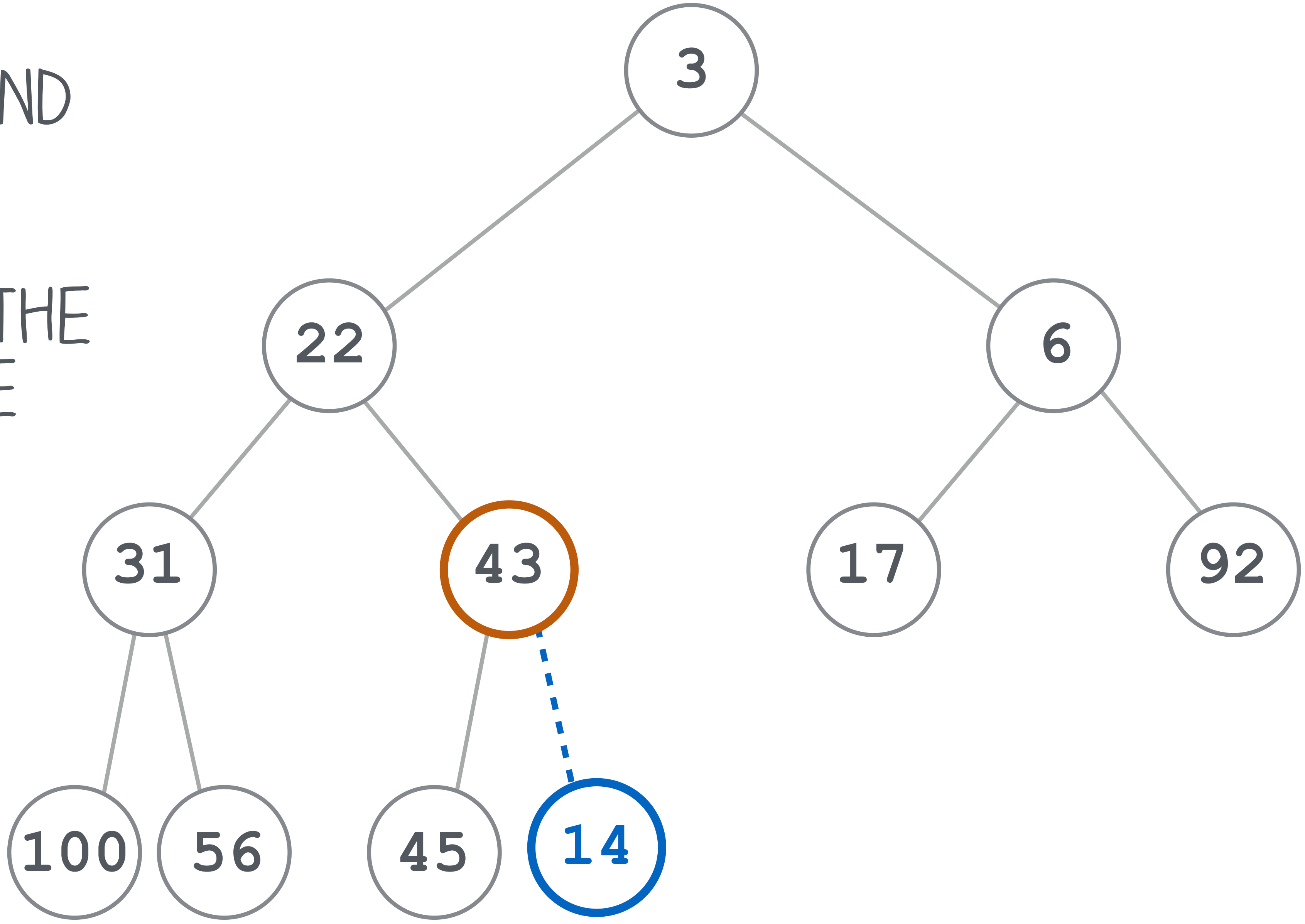
PERCOLATE UP THE TREE TO FIX THE ORDER



ADDING 14

PUT IT AT THE END OF THE TREE

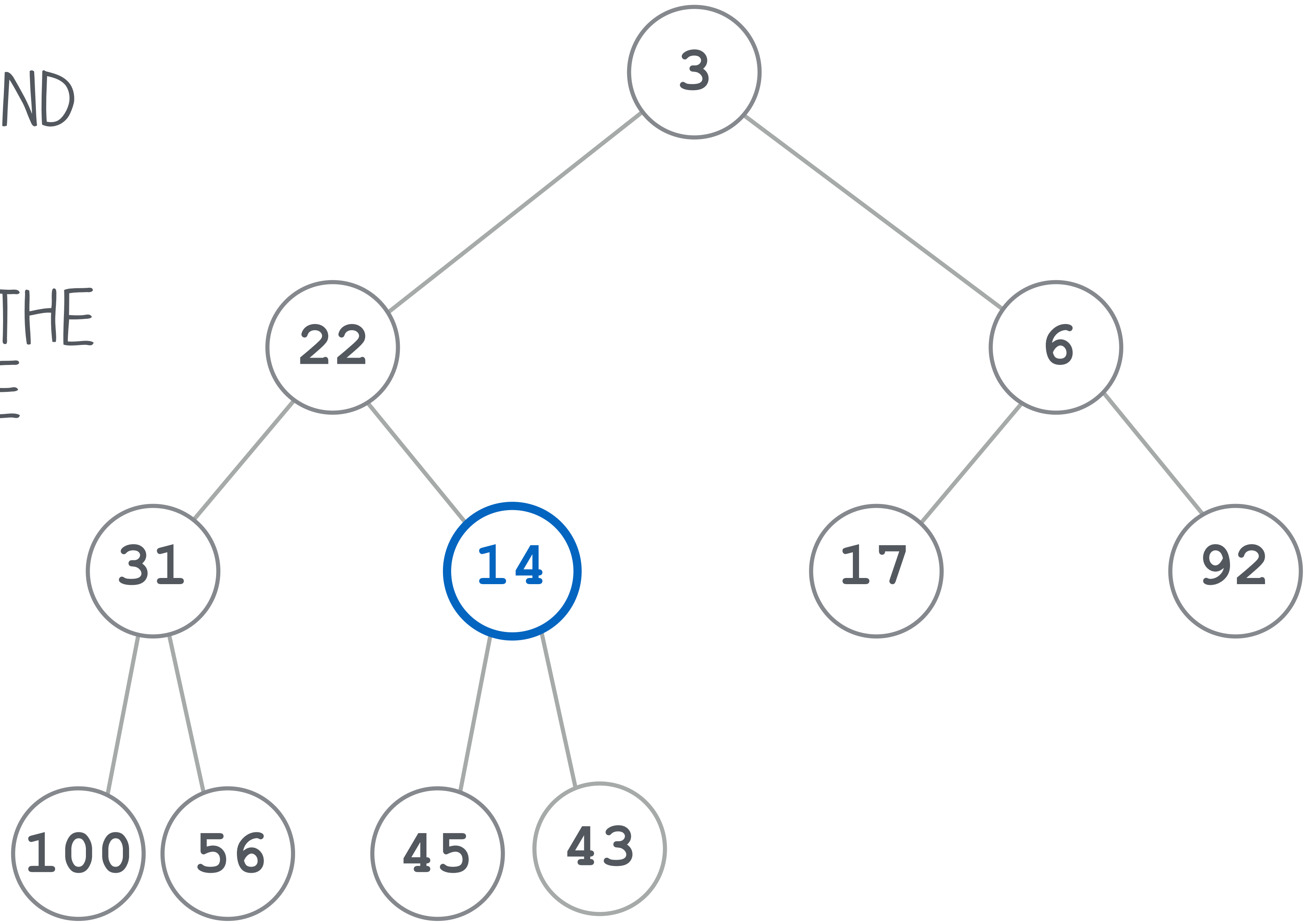
PERCOLATE UP THE TREE TO FIX THE ORDER



ADDING 14

PUT IT AT THE END OF THE TREE

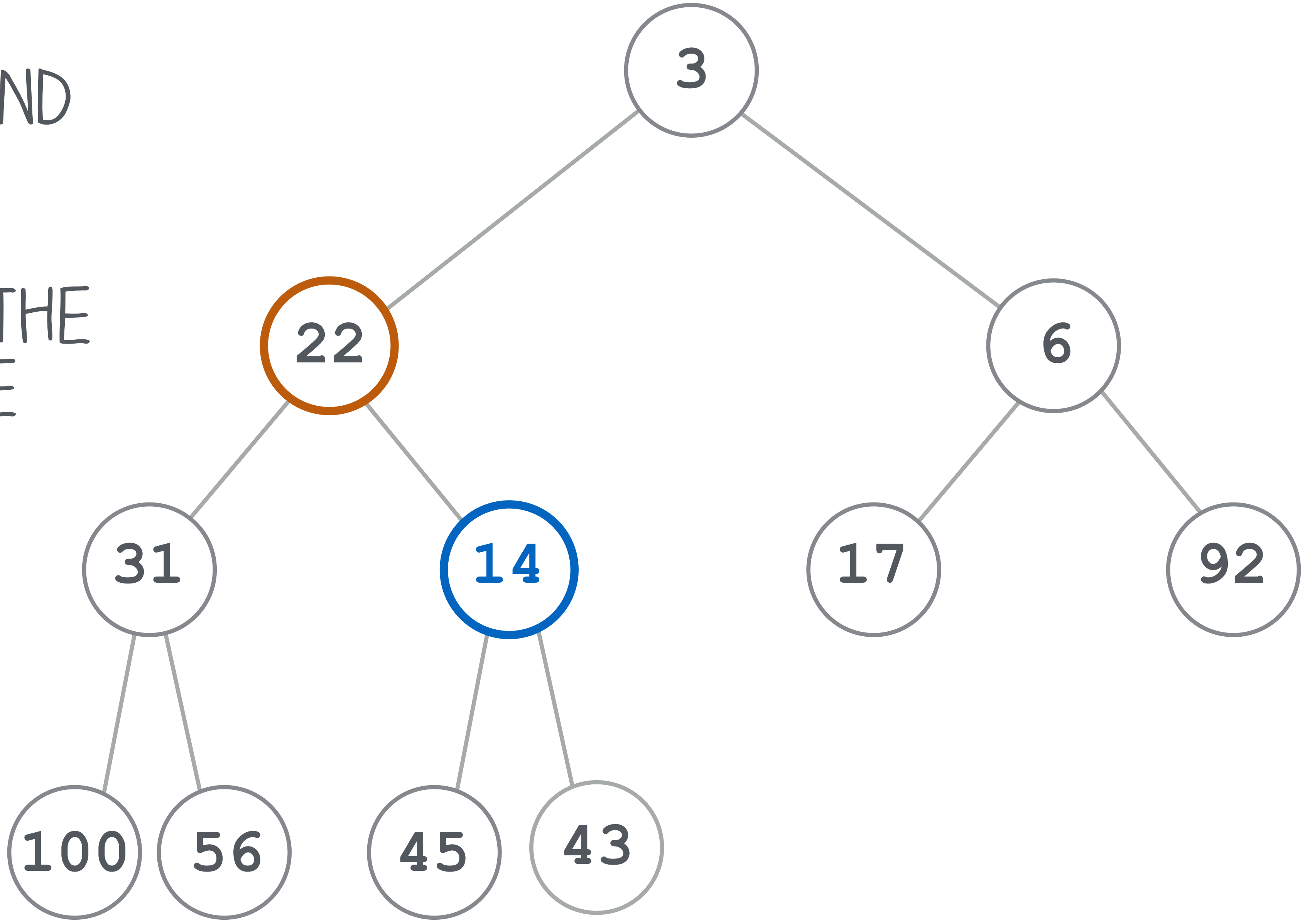
PERCOLATE UP THE TREE TO FIX THE ORDER



ADDING 14

PUT IT AT THE END OF THE TREE

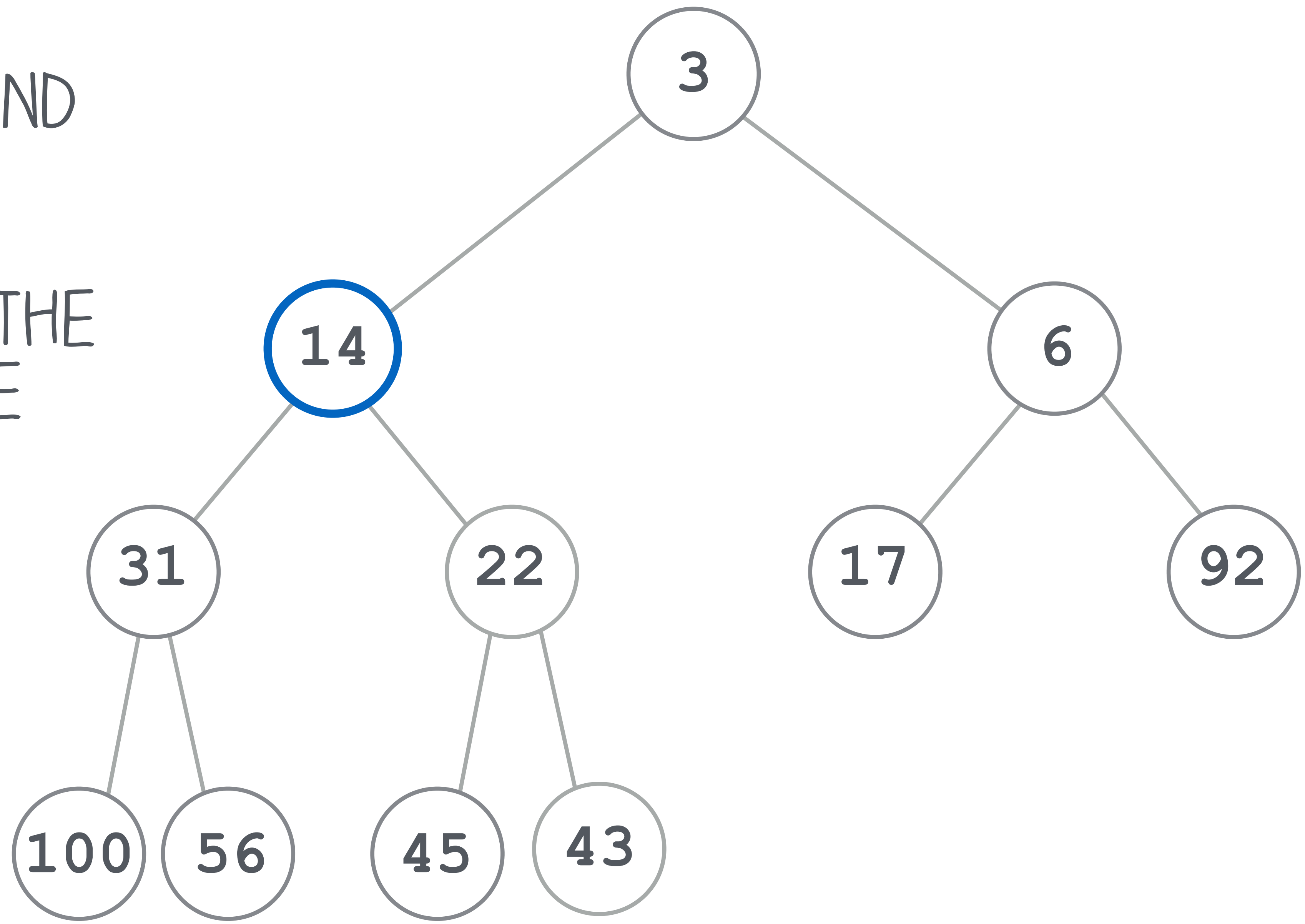
PERCOLATE UP THE TREE TO FIX THE ORDER



ADDING 14

PUT IT AT THE END OF THE TREE

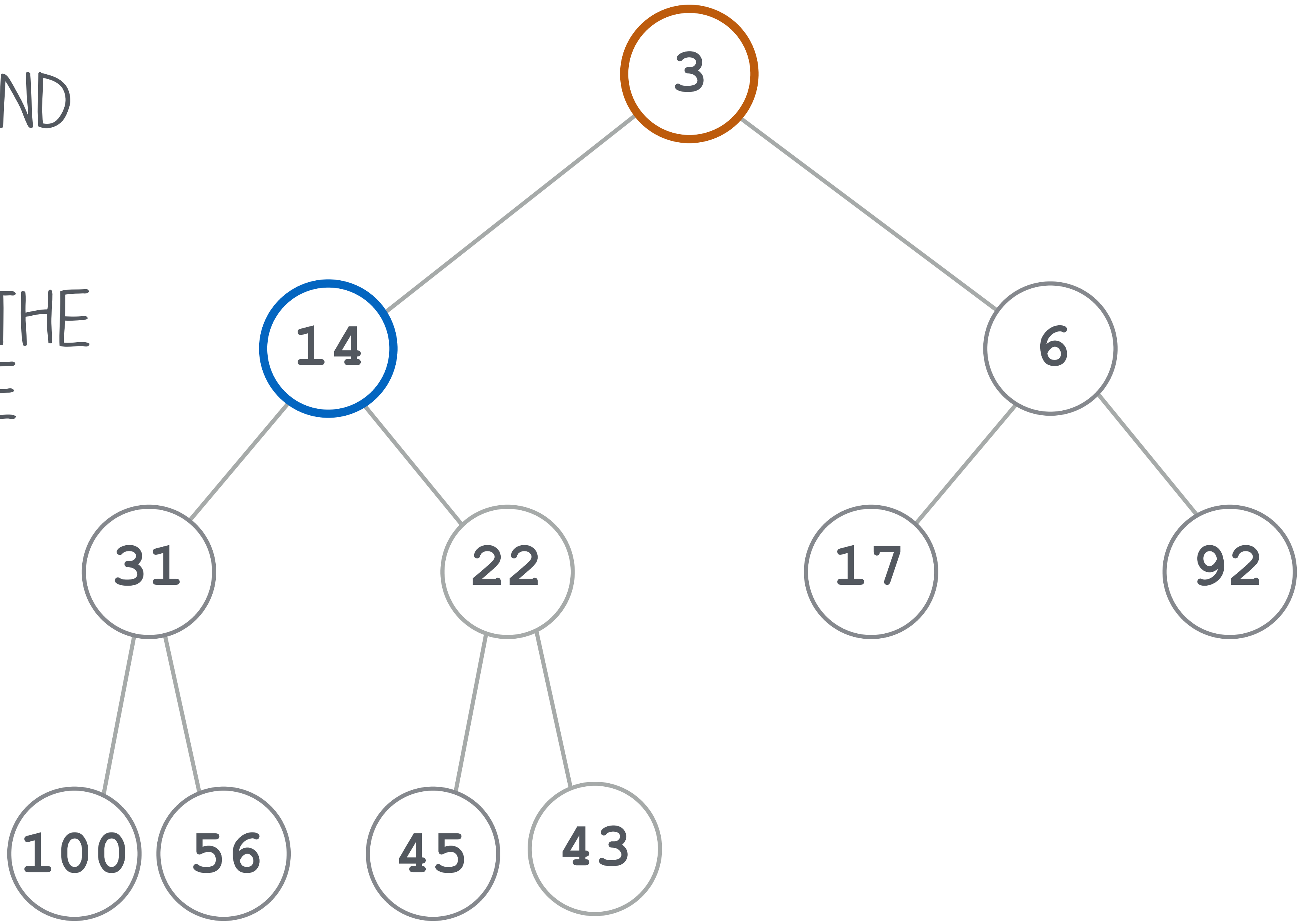
PERCOLATE UP THE TREE TO FIX THE ORDER



ADDING 14

PUT IT AT THE END OF THE TREE

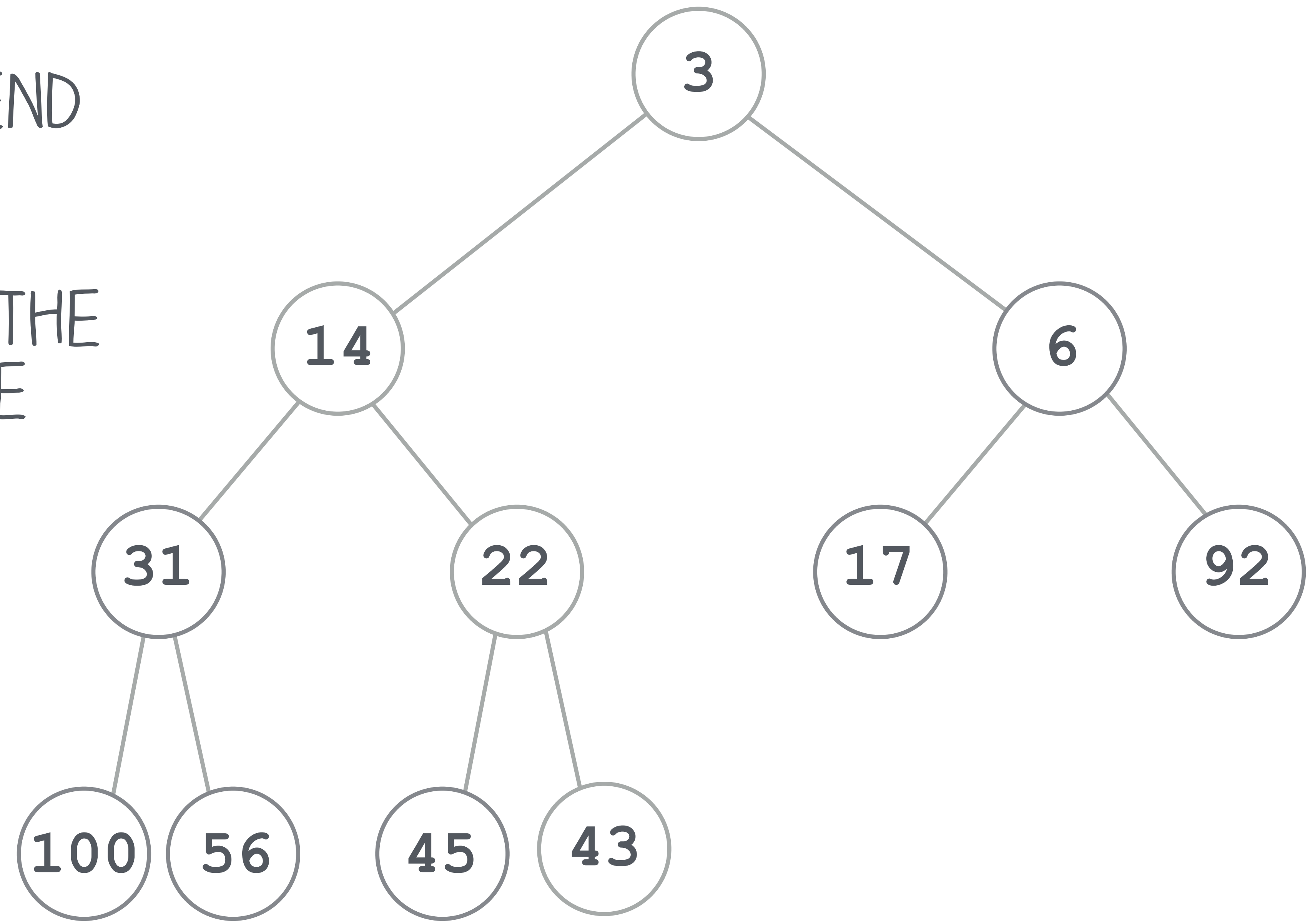
PERCOLATE UP THE TREE TO FIX THE ORDER



ADDING 14

PUT IT AT THE END OF THE TREE

PERCOLATE UP THE TREE TO FIX THE ORDER

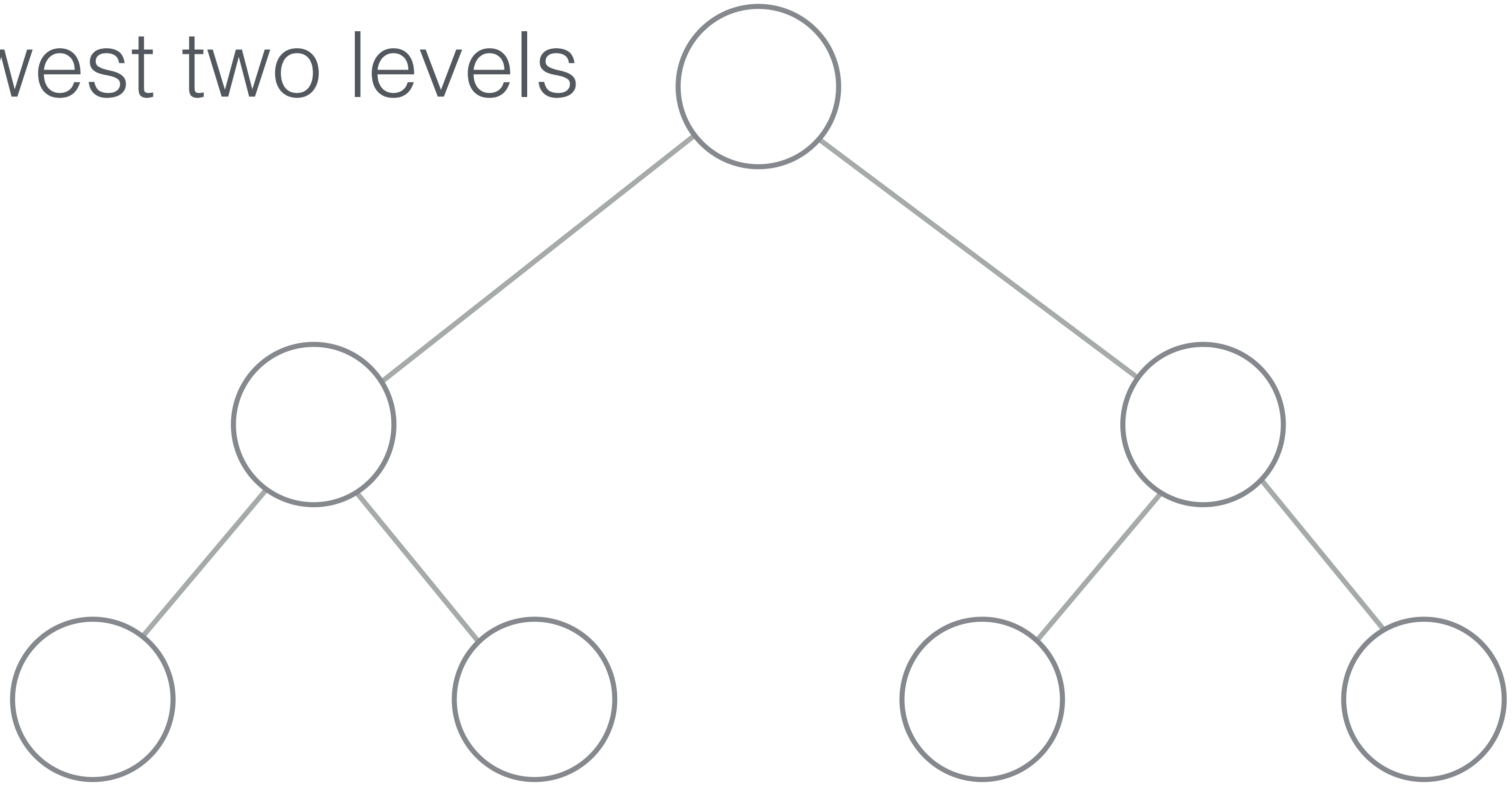


cost of add

-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



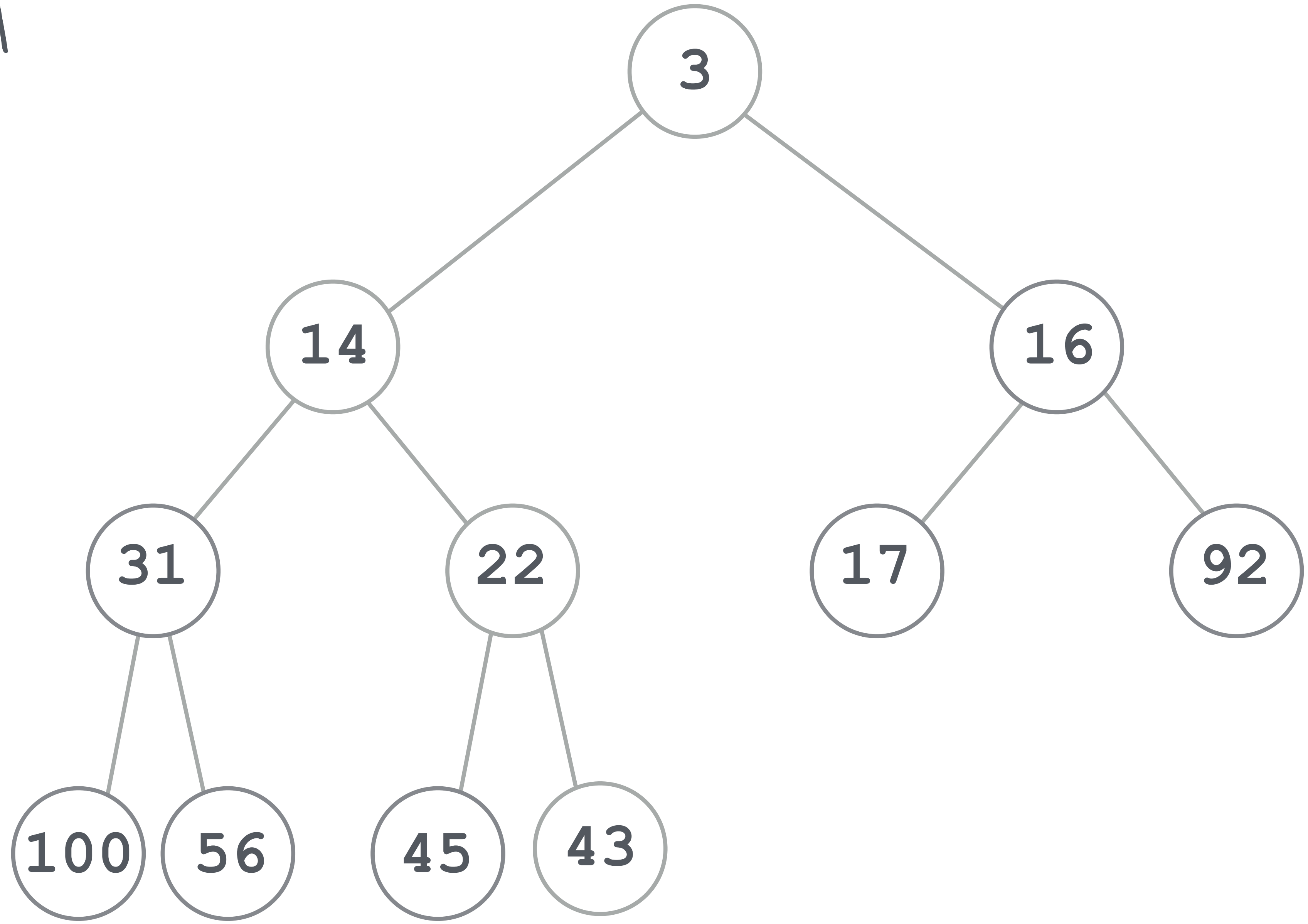
- if the new item is the smallest in the set, cost is **O(logN)**
  - must percolate up every level to the root
  - complete trees have logN levels
    - *is this the worst, average, or best case?*

- it has been shown that on average, 2.6 comparisons are needed for any N
  - thus, add terminates early, and average cost is **O(1)**

remove

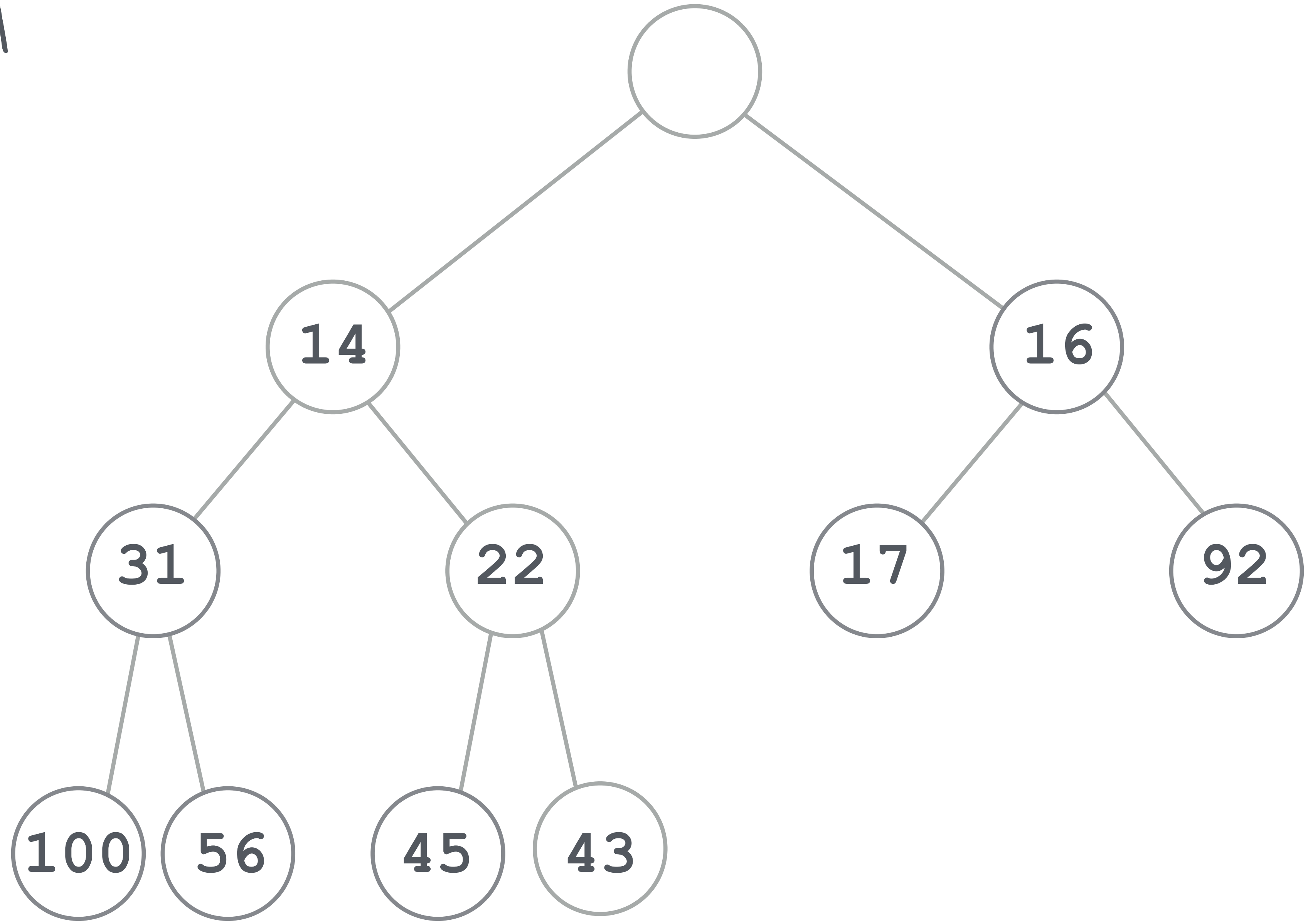
LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3



LET'S REMOVE THE SMALLEST ITEM

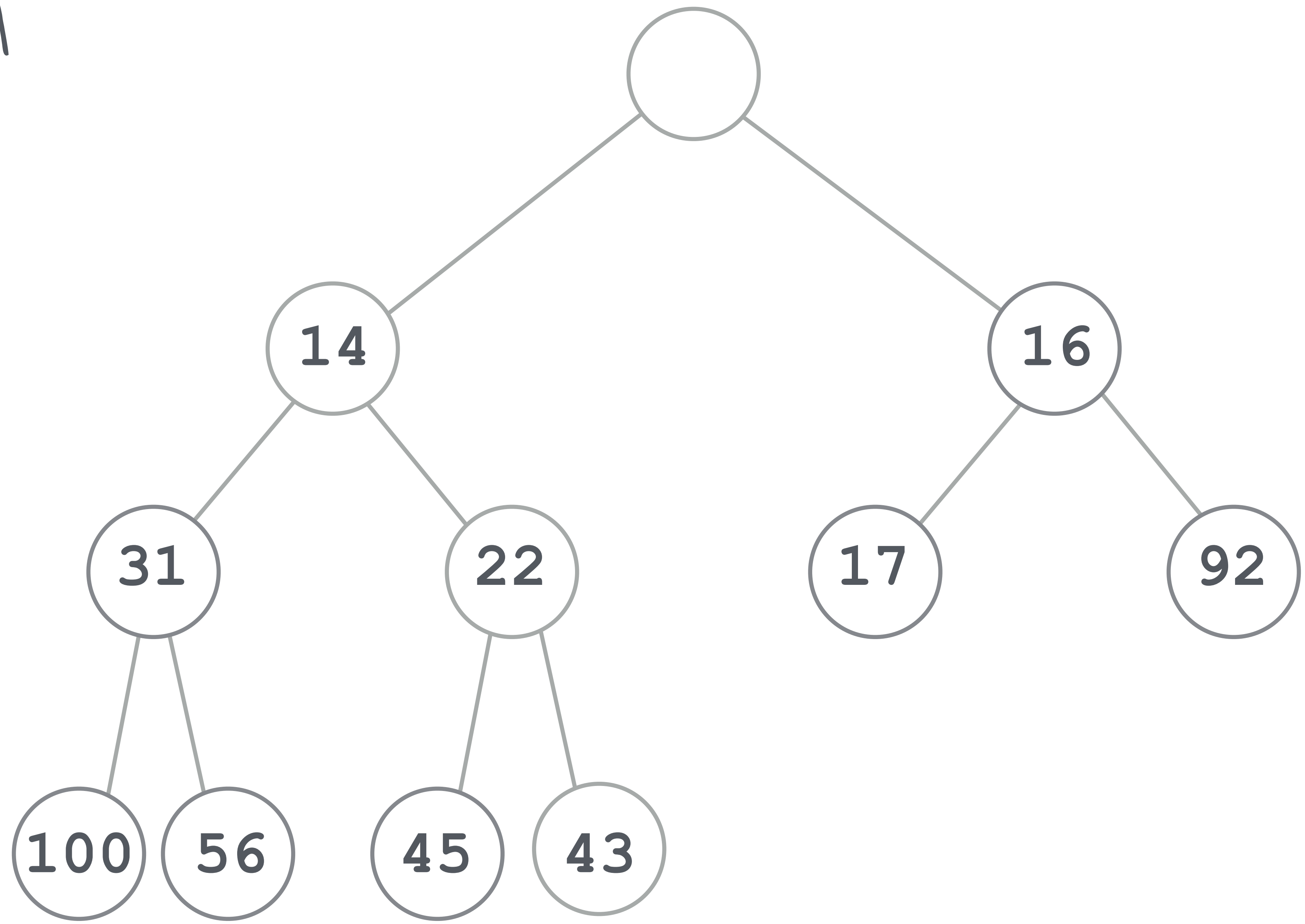
TAKE OUT 3



LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3

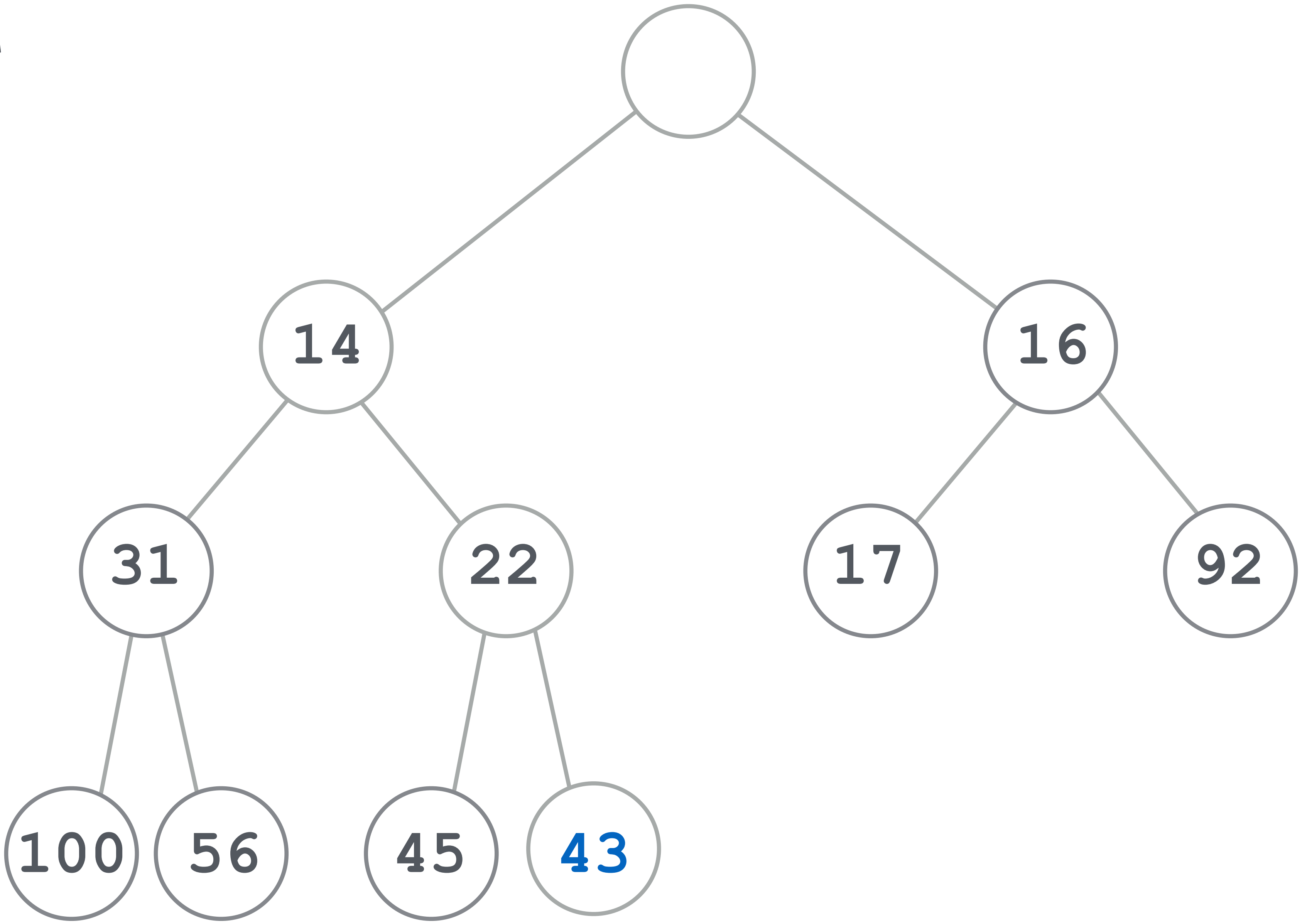
FILL WITH LAST ITEM ON LAST LEVEL. WHY?



LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3

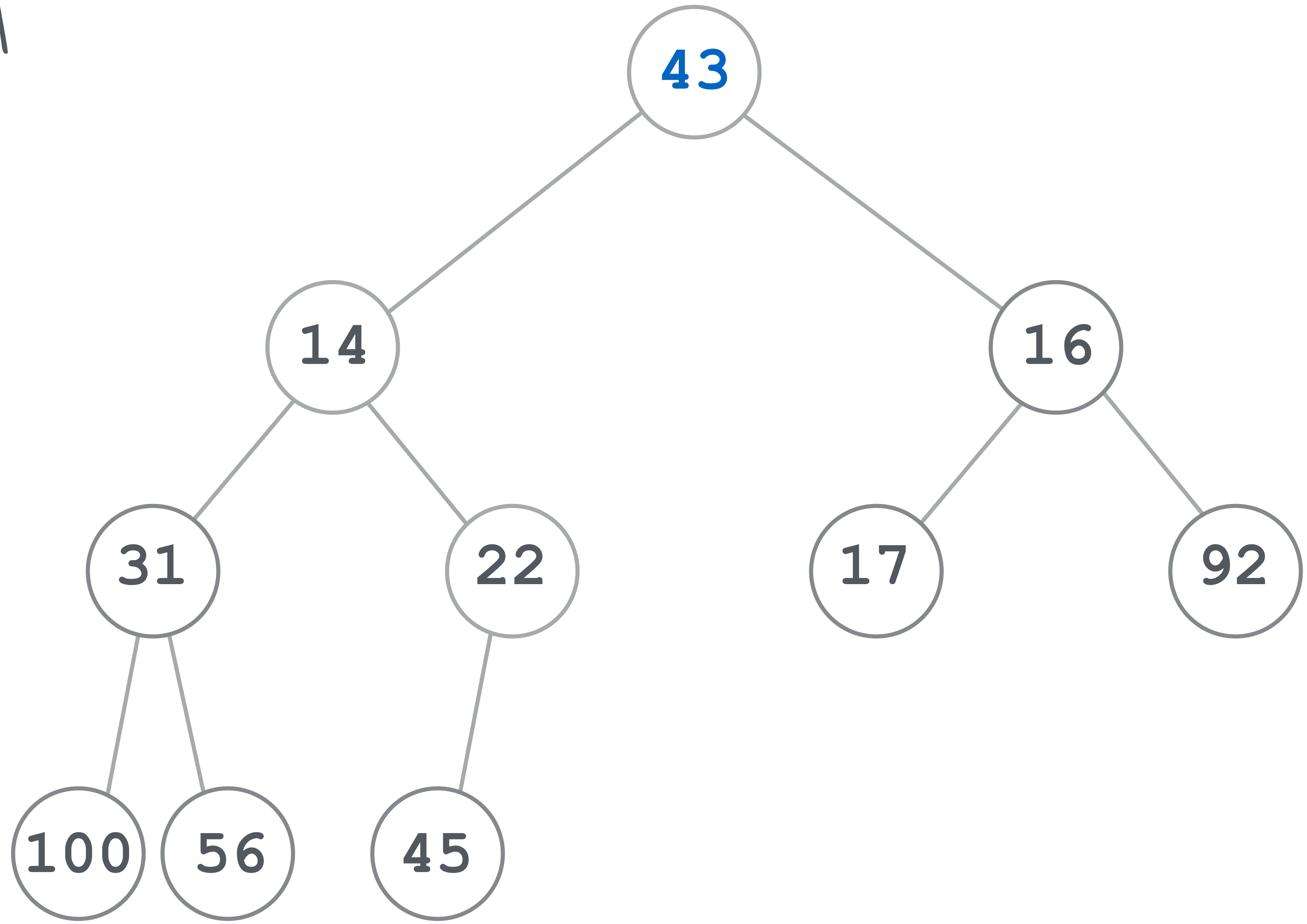
FILL WITH LAST ITEM ON LAST LEVEL. WHY?



LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3

FILL WITH LAST ITEM ON LAST LEVEL. WHY?

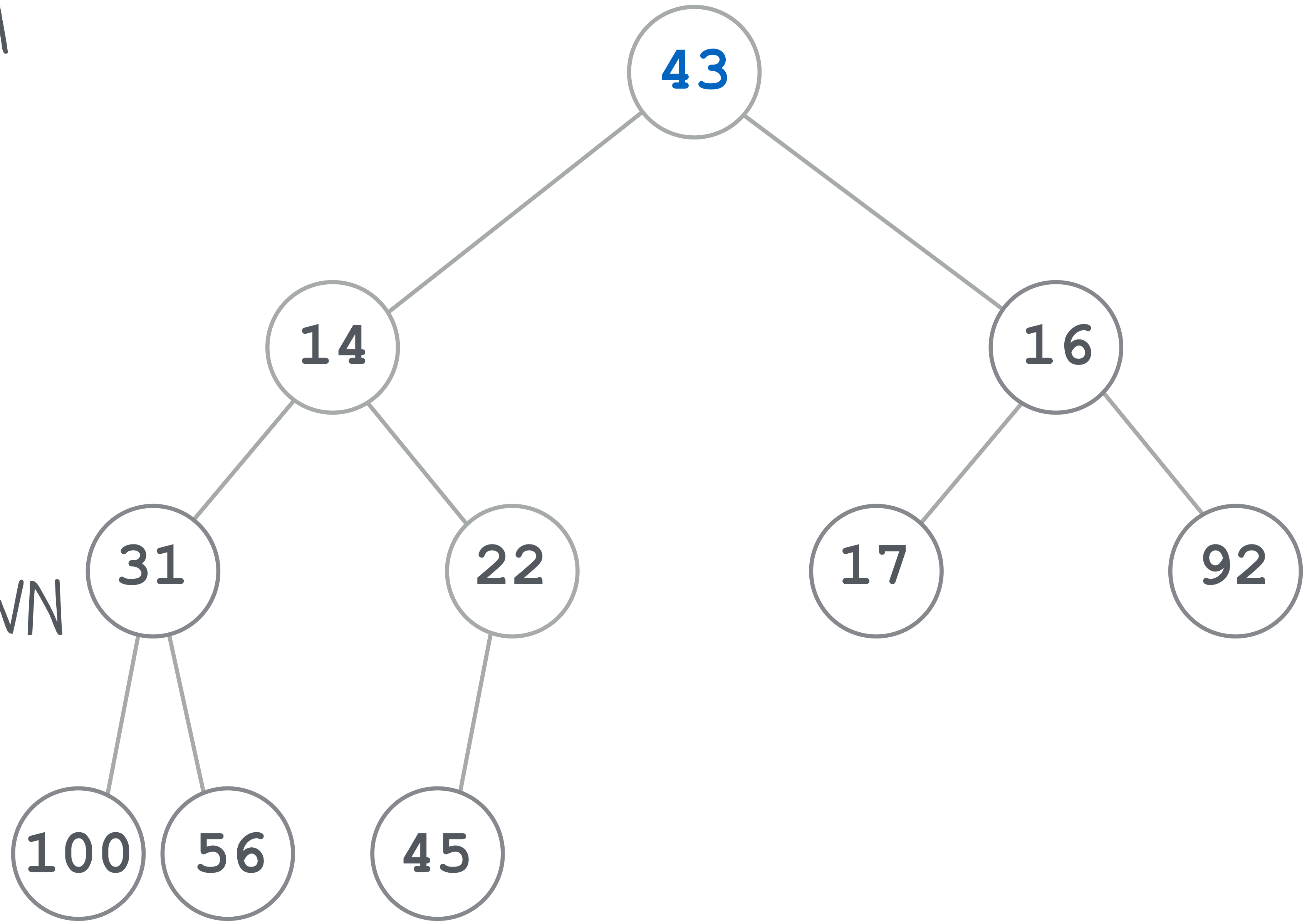


LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3

FILL WITH LAST ITEM ON LAST LEVEL. WHY?

PERCOLATE DOWN

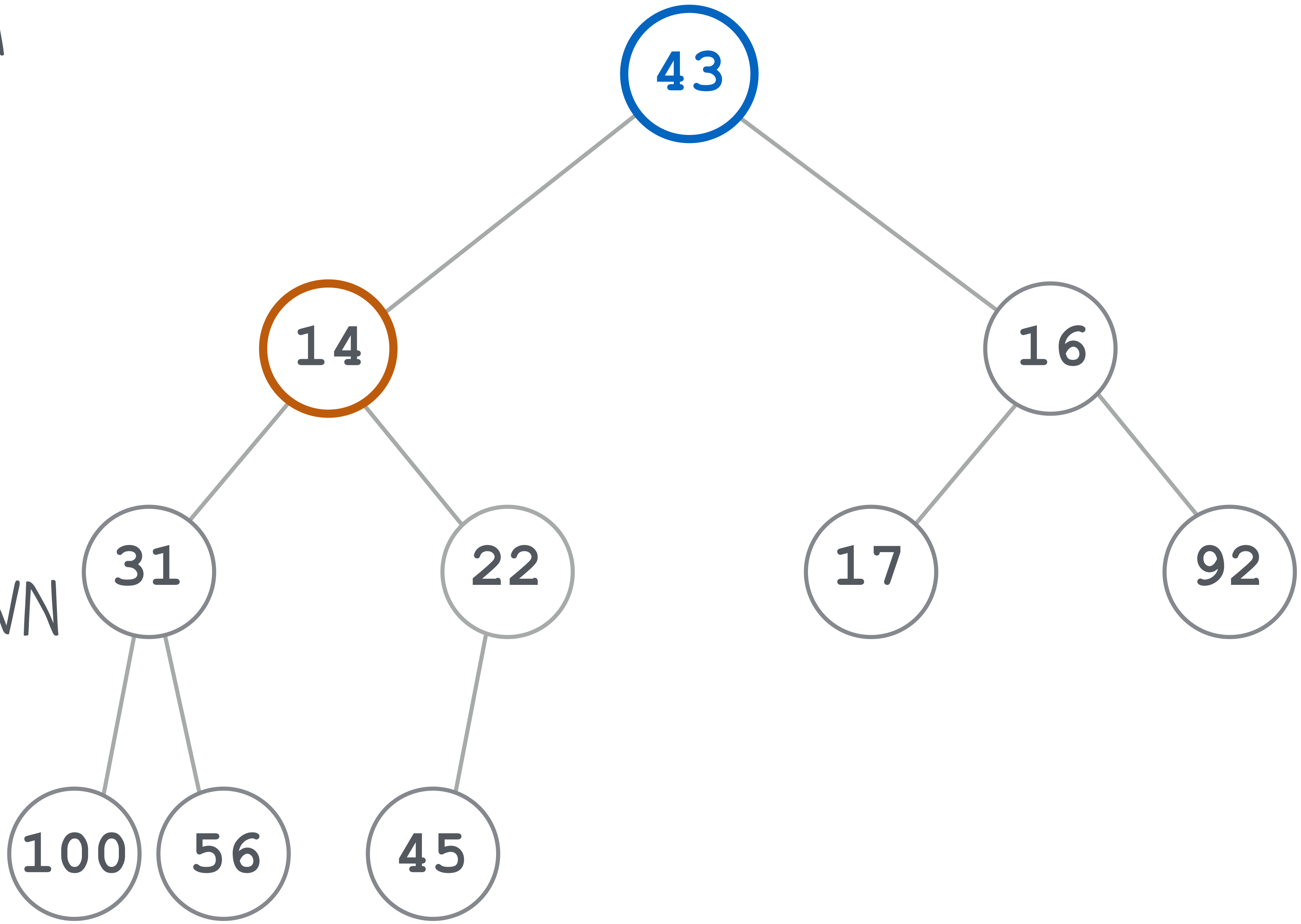


LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3

FILL WITH LAST ITEM ON LAST LEVEL. WHY?

PERCOLATE DOWN

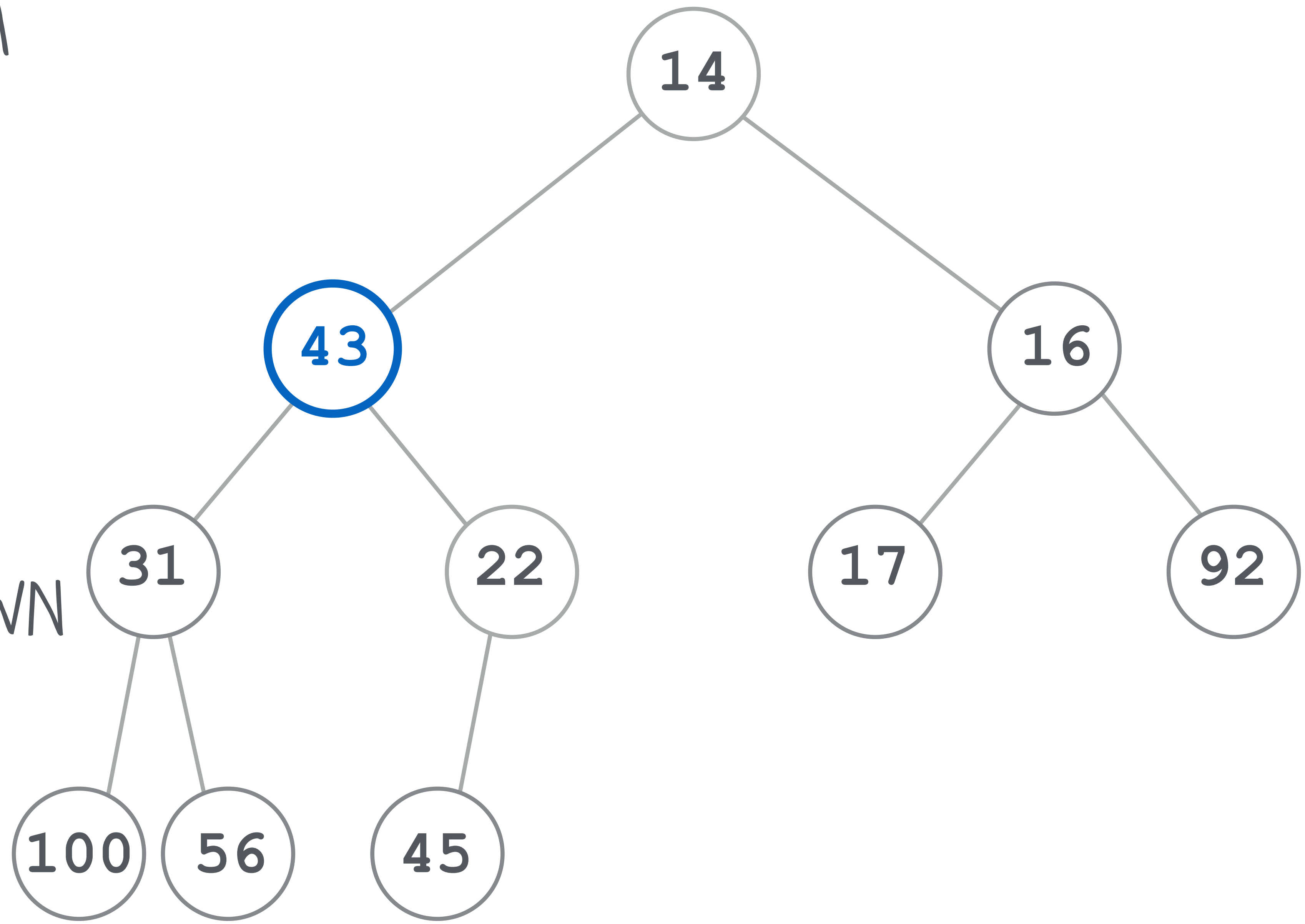


LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3

FILL WITH LAST ITEM ON LAST LEVEL. WHY?

PERCOLATE DOWN

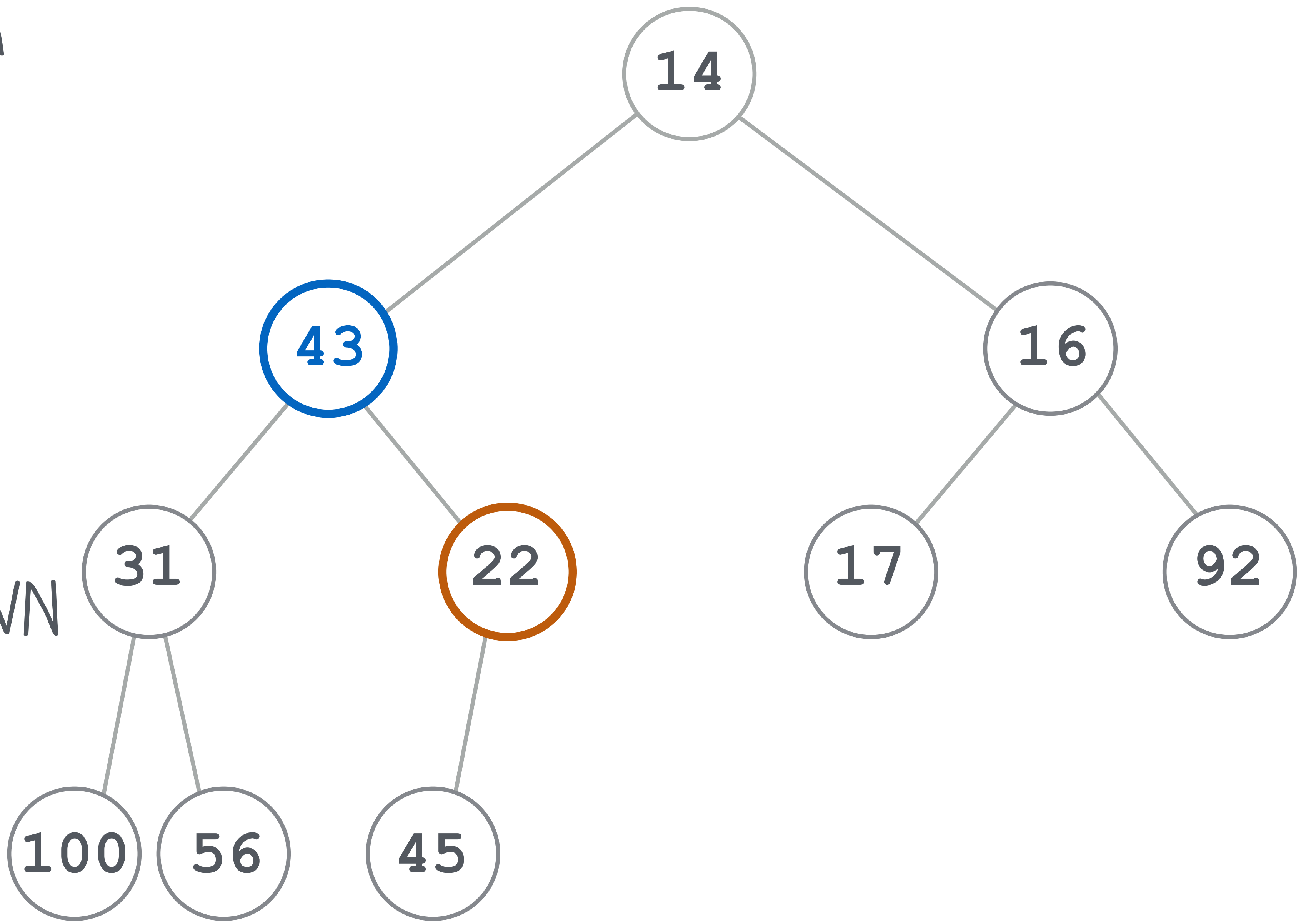


LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3

FILL WITH LAST ITEM ON LAST LEVEL. WHY?

PERCOLATE DOWN

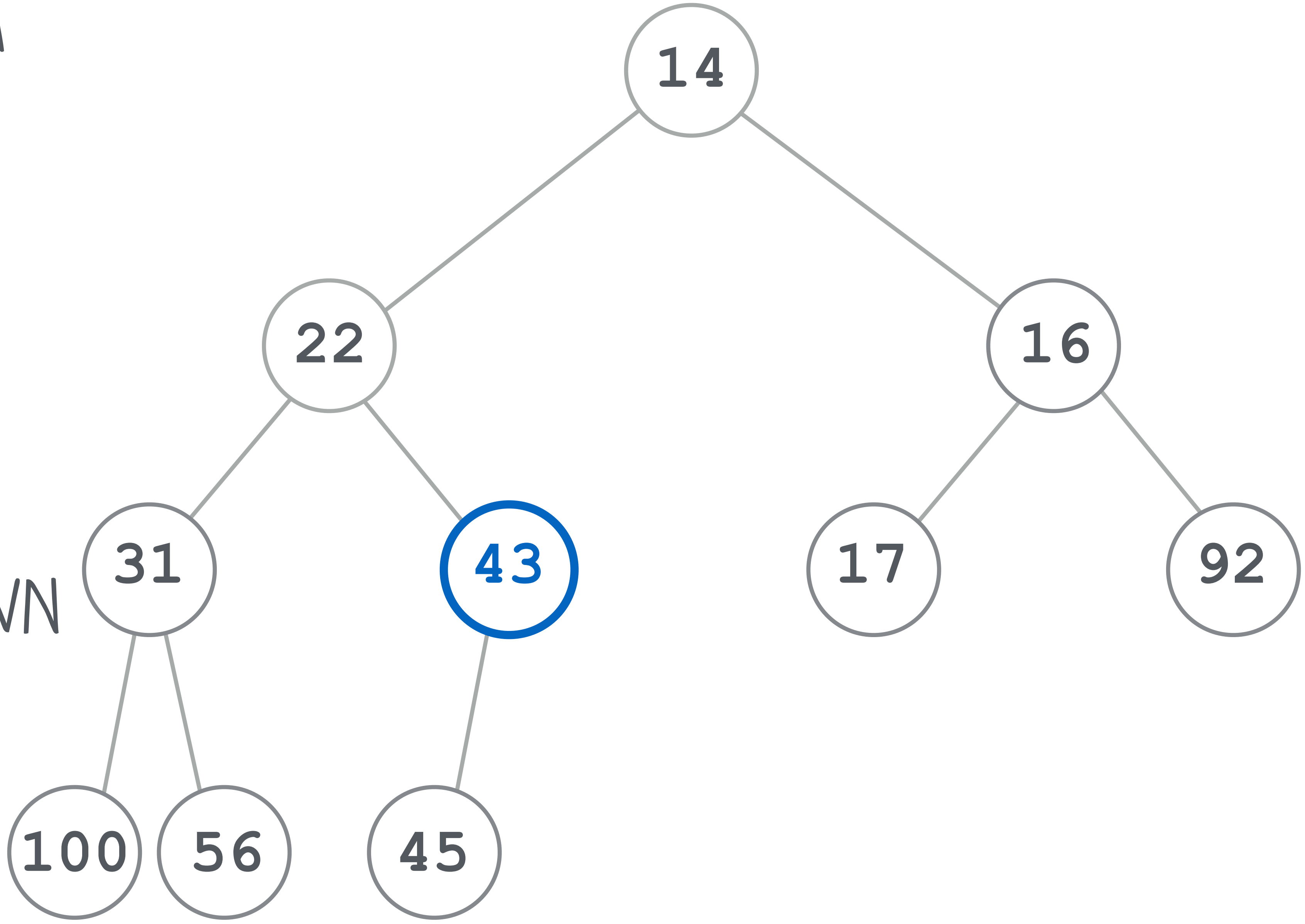


LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3

FILL WITH LAST ITEM ON LAST LEVEL. WHY?

PERCOLATE DOWN

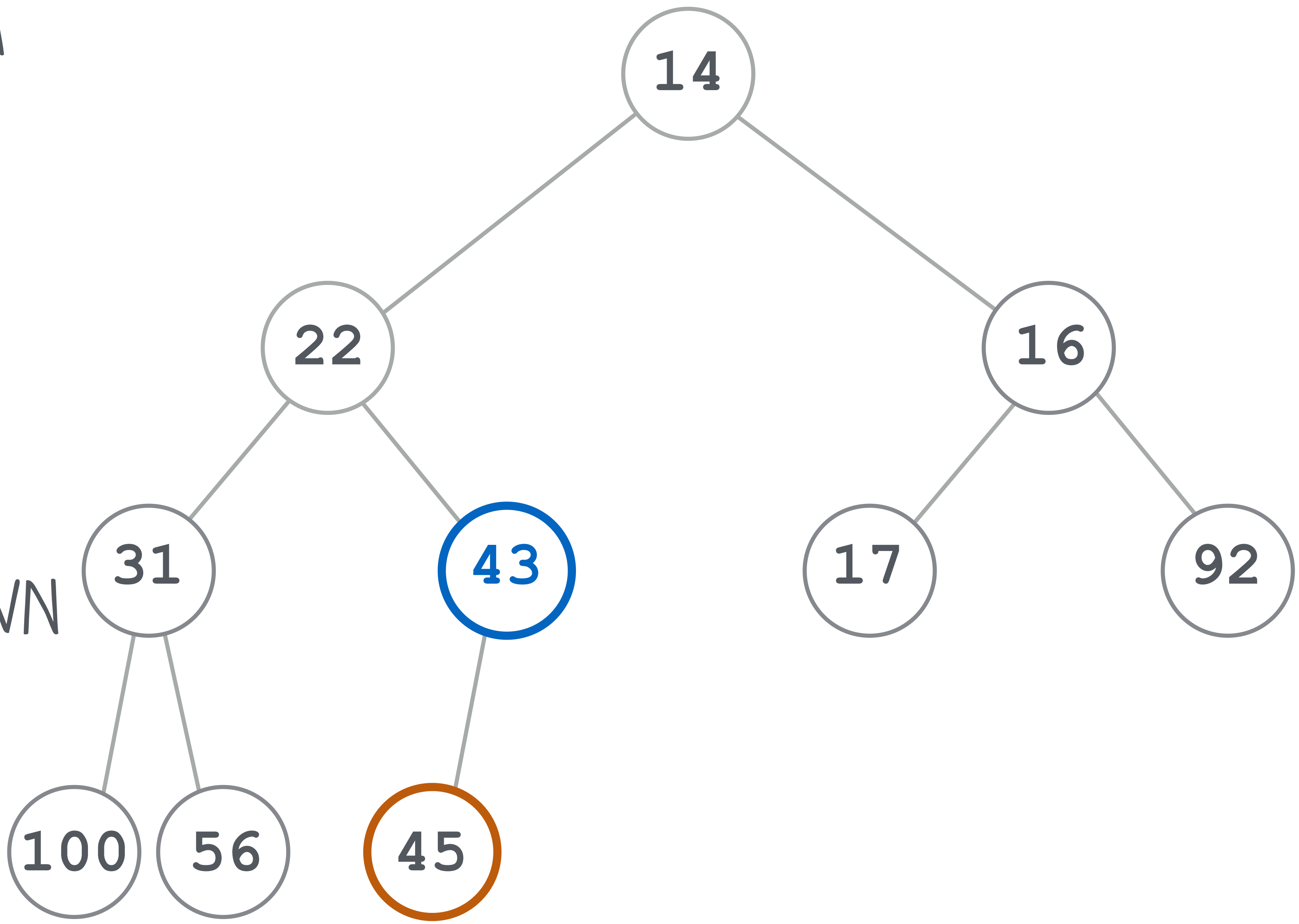


LET'S REMOVE THE SMALLEST ITEM

TAKE OUT 3

FILL WITH LAST ITEM ON LAST LEVEL. WHY?

PERCOLATE DOWN

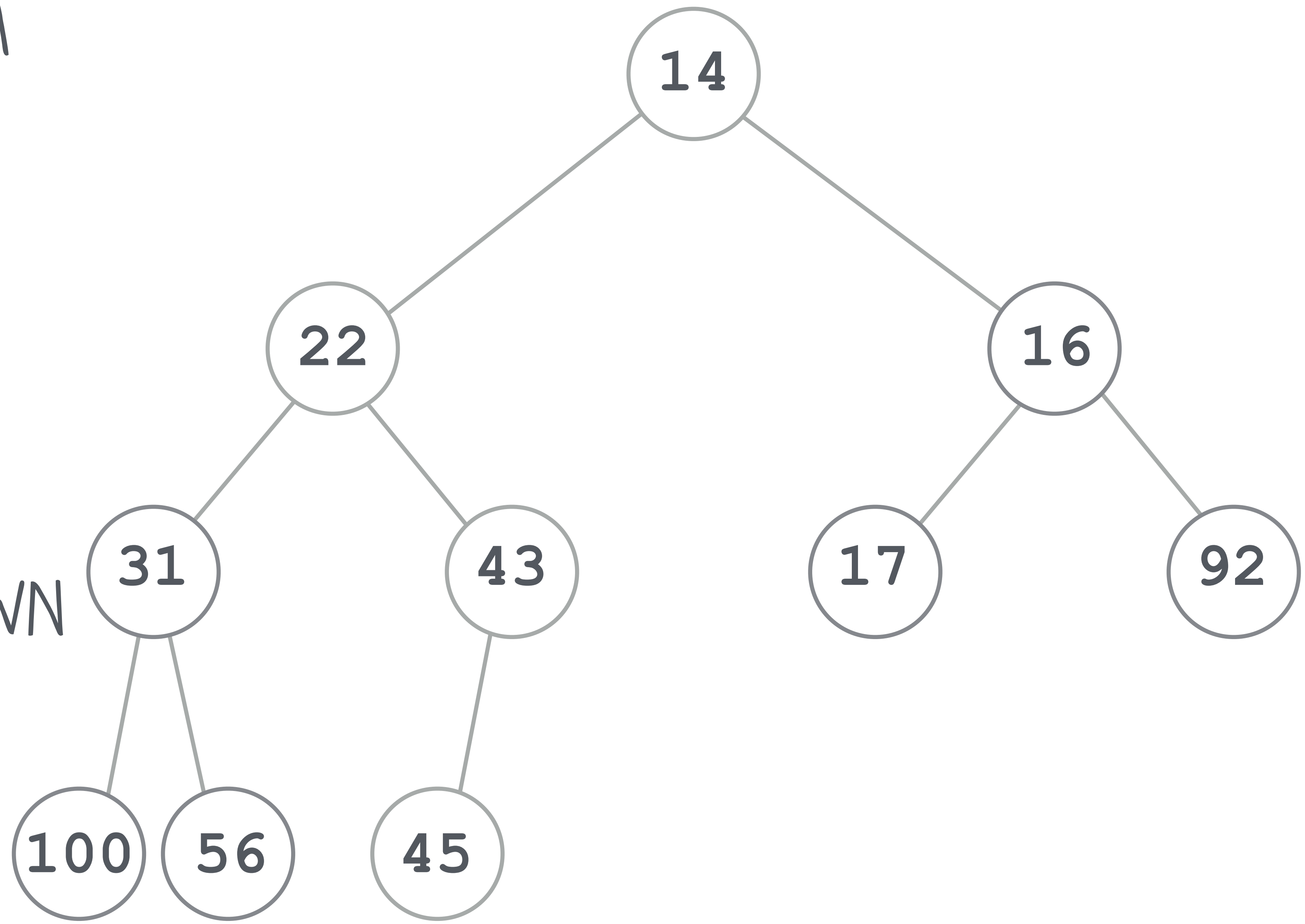


LET'S REMOVE THE SMALLEST ITEM

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?

recap

-priority queues can be implemented any number of ways

-a binary heap's main use is for implementing priority queues

-remember, the basic priority queue operations are:

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

-the average cases for a PQ implemented with a binary heap:

- -add
  - -**O(1):** *percolate up (average of 2.6 compares)*
- -findMin
  - - **O(1):** *just return the root*
- -deleteMin
  - - **O(logN):** *percolate down (rarely terminates before near the bottom of the tree)*

next time…

- reading
    - chapter 21 in book

- homework
    - assignment 10 due Thursday