Priority Queue ADT

Queue ADT

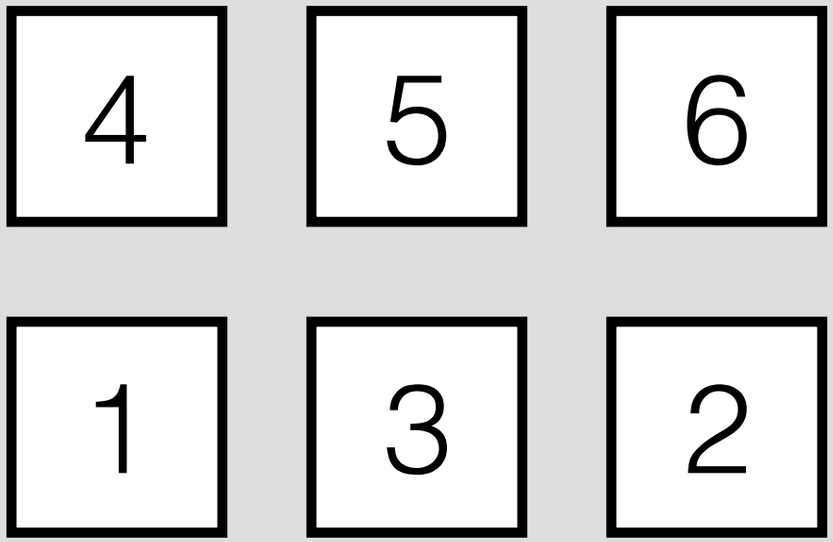
abstract data type

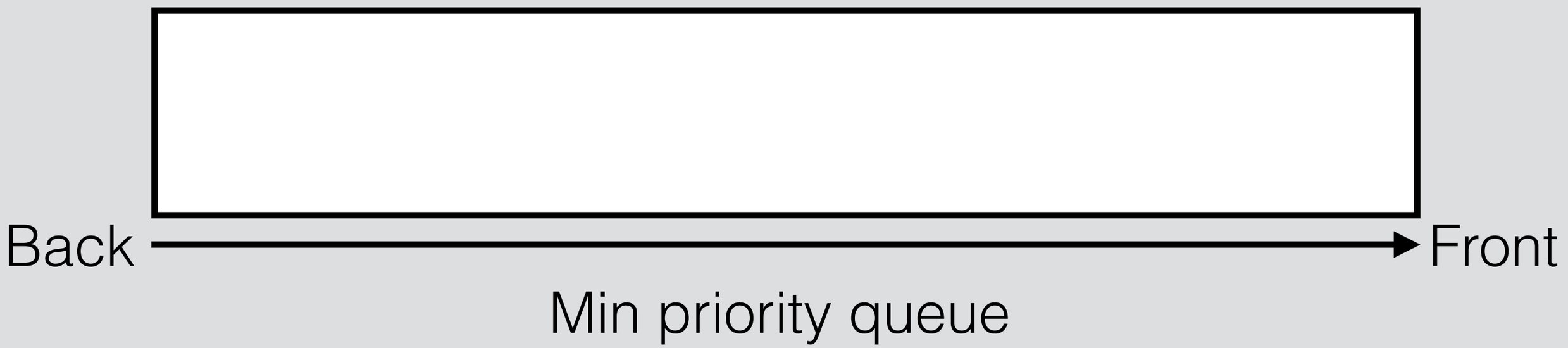
- **Data:** a sequence of elements.
- **Operation (push):** Add new element to back.
- **Operation (pop):** Remove element at front.
- **Operation (front):** Return element at front.
- **Operation (size):** Return the # element in the queue.

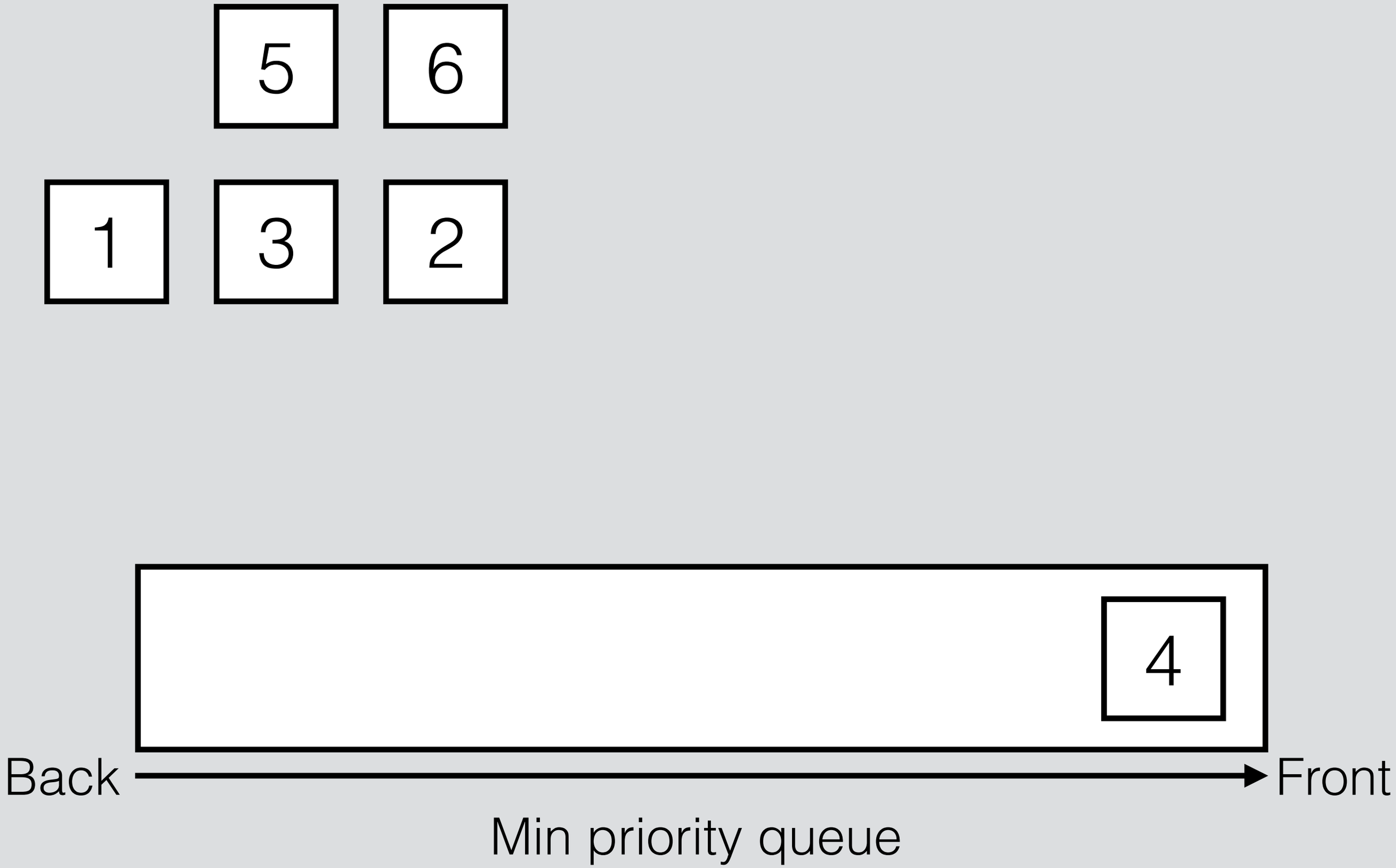
Min Priority Queue ADT

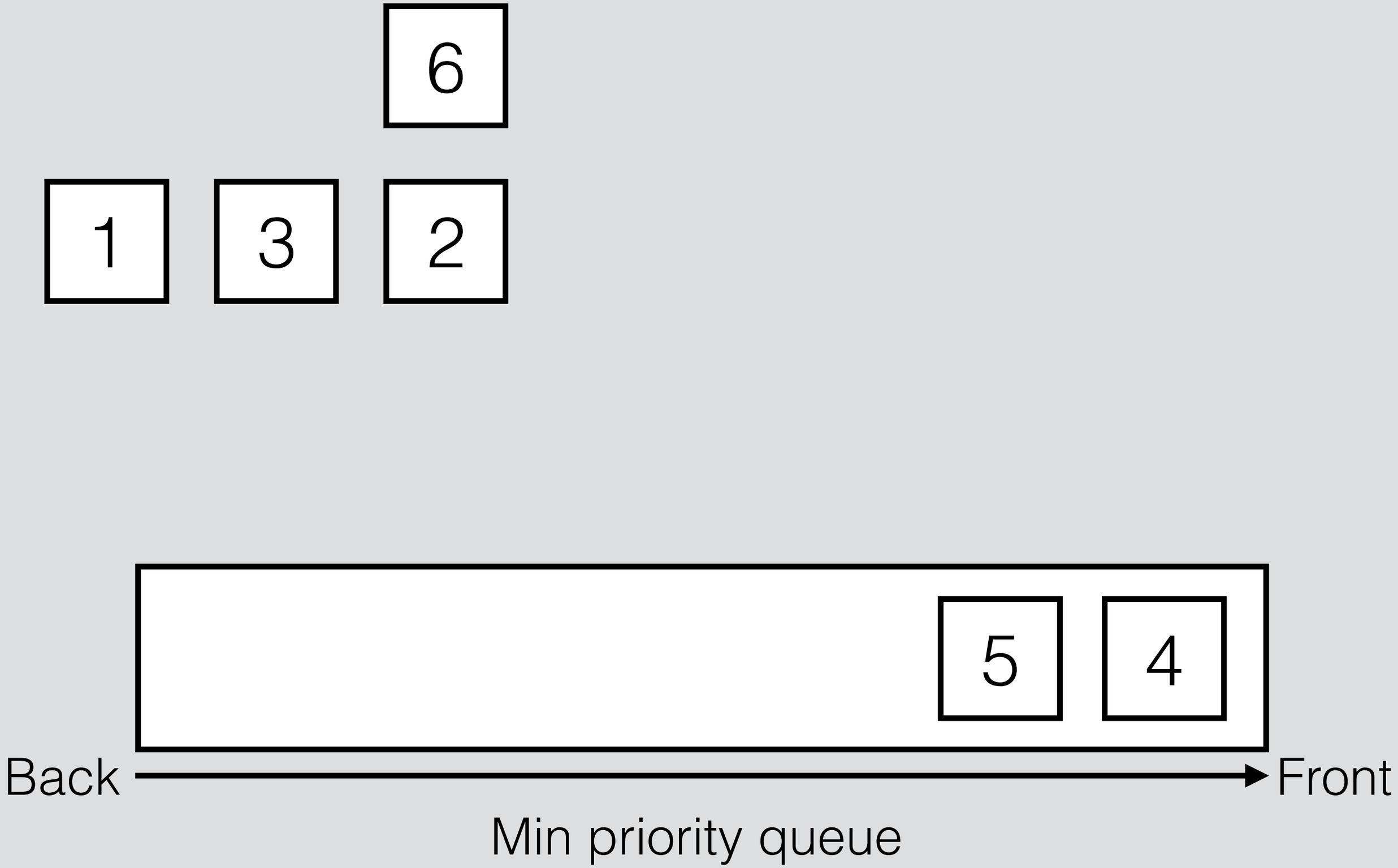
supports <

- **Data:** a sequence of (value, priority) pairs.
- **Operation (push):** Add new (value, priority) pair to queue.
- **Operation (pop):** Remove smallest-priority element.
- **Operation (front):** Return smallest-priority element.
- **Operation (size):** Return the # elements in the queue.

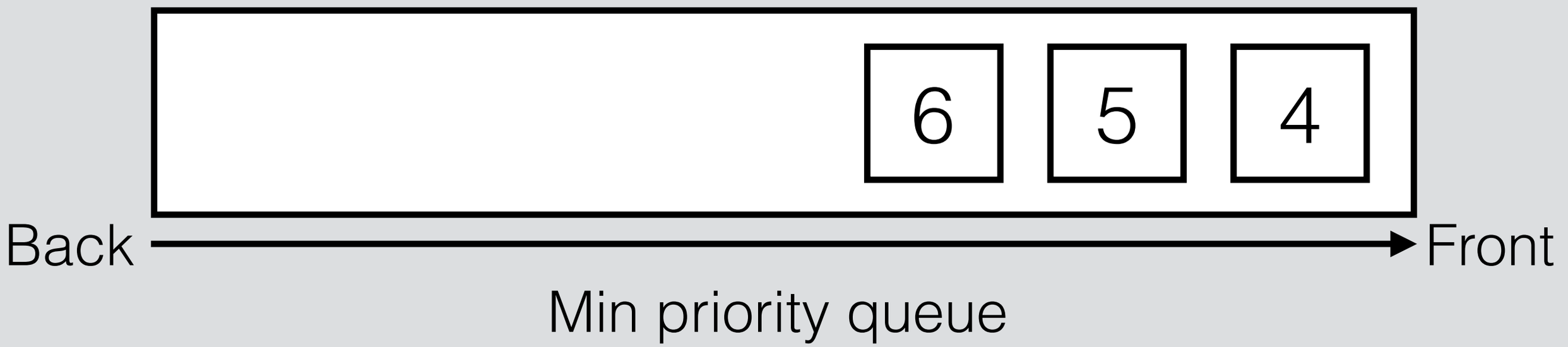

4
5
6
1
3
2


Back
Front
Min priority queue

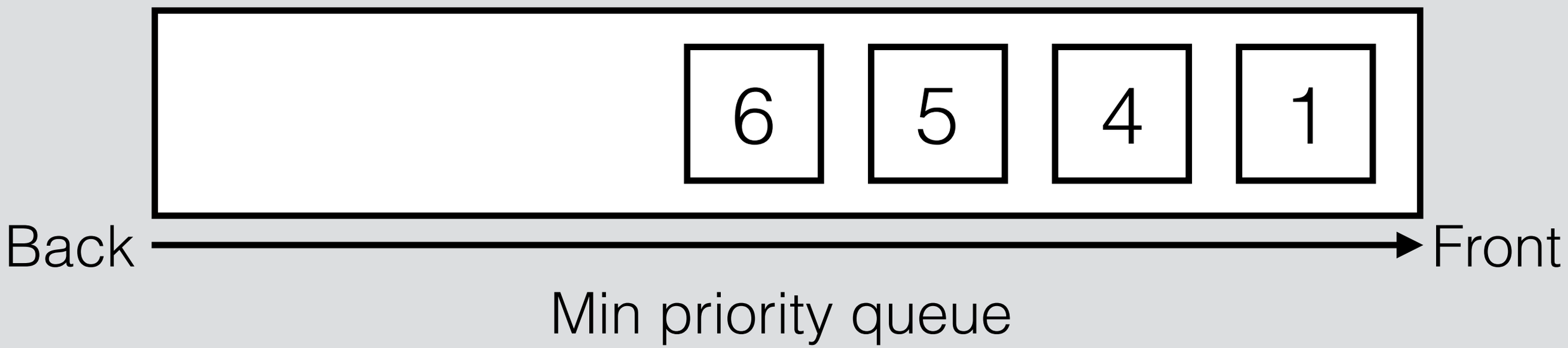

5
6
1
3
2
4
Back
Front
Min priority queue


6
1
3
2
5
4
Back
Front
Min priority queue


1
3
2

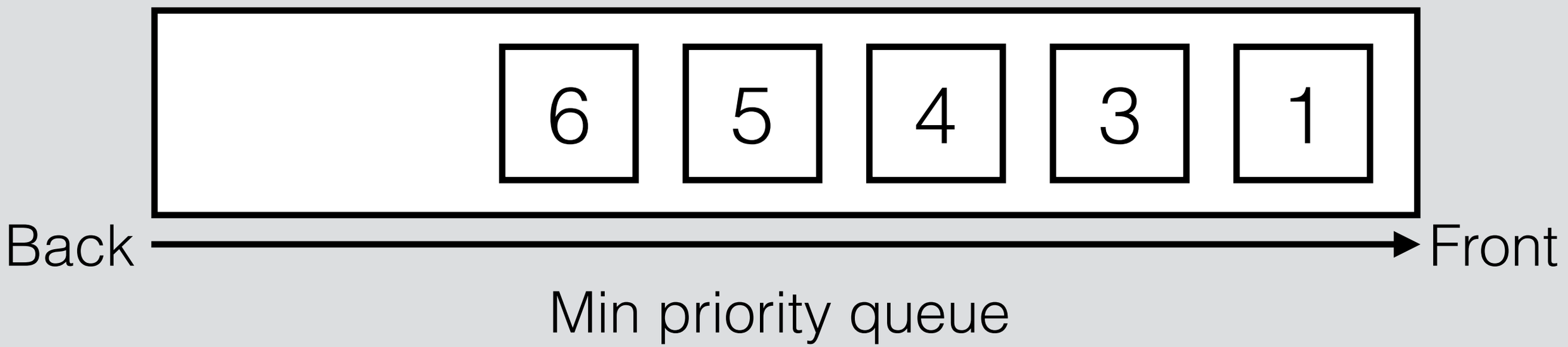

6
5
4
Back
Front
Min priority queue

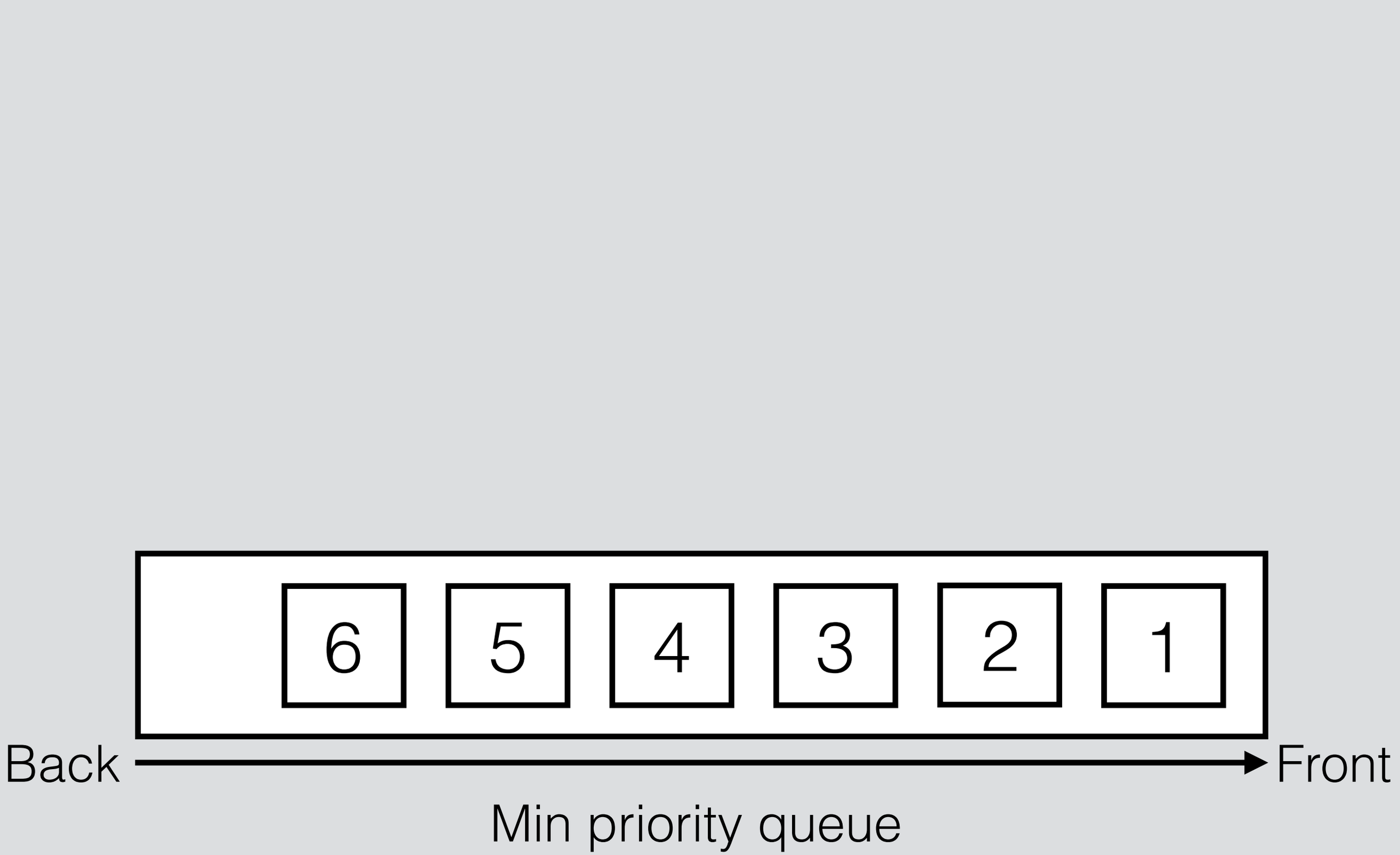
3 2

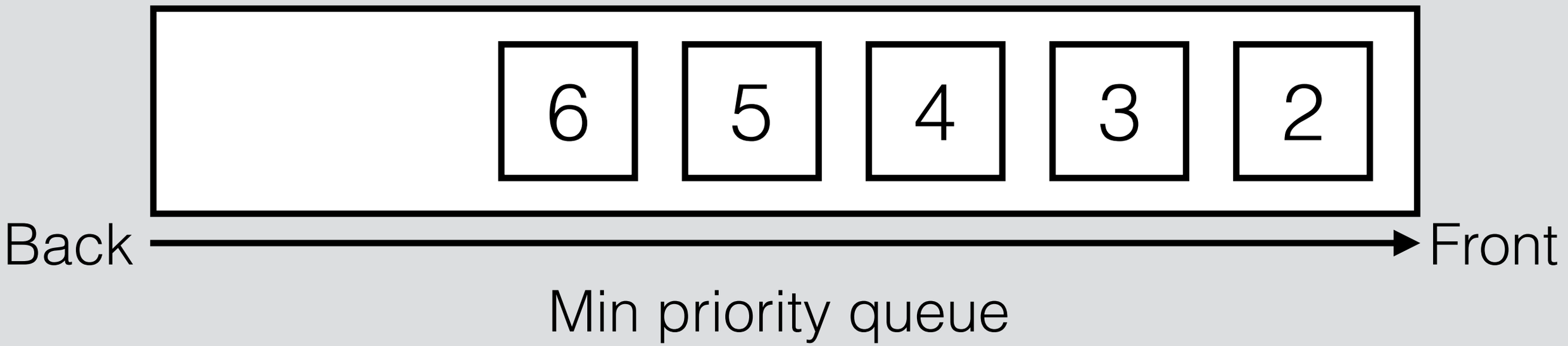


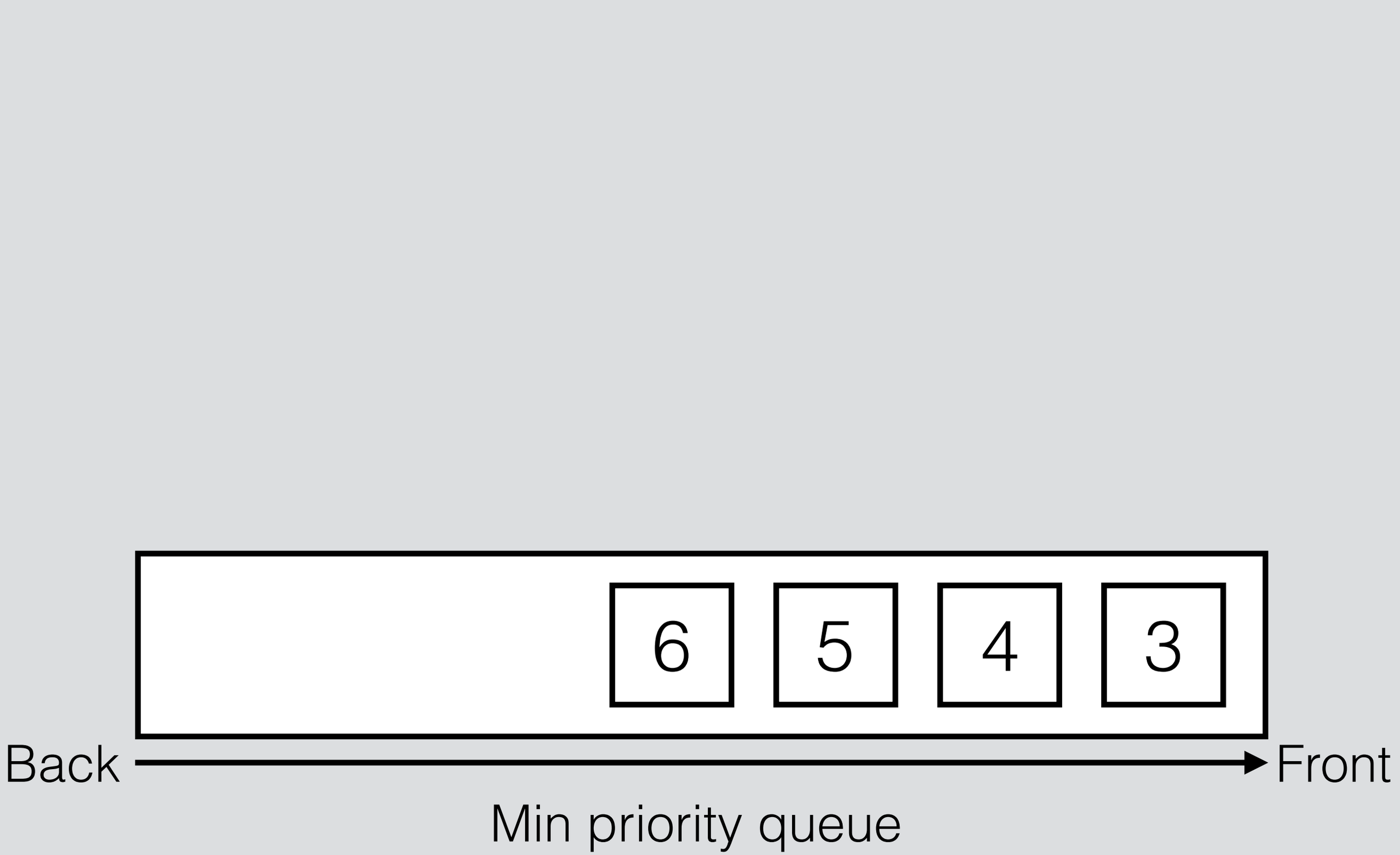
Min priority queue

2




6
5
4
3
2
1
Back
Front
Min priority queue


6
5
4
3
2
Back
Front
Min priority queue



Min priority queue

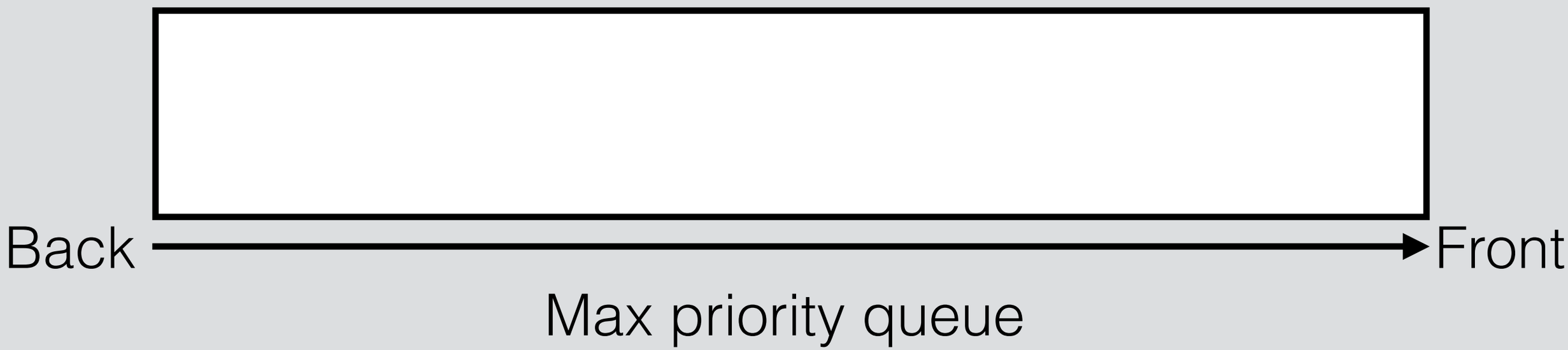
Max Priority Queue ADT

supports <

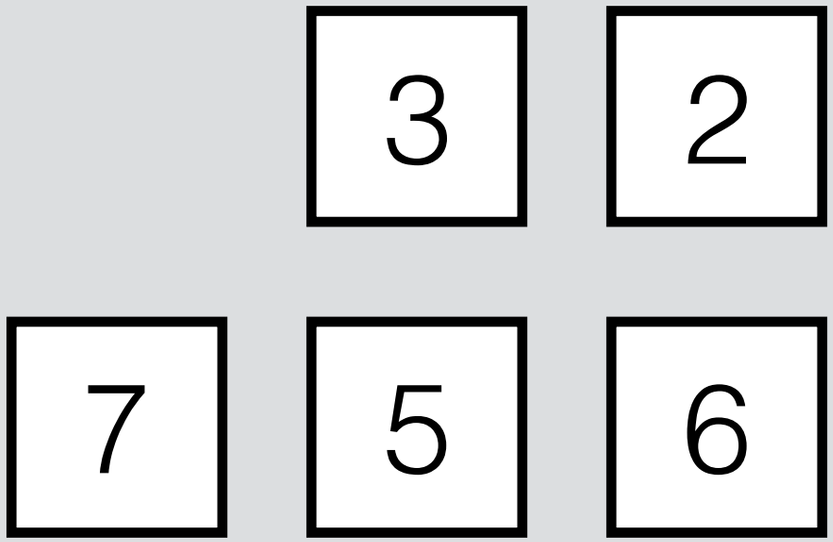
- **Data:** a sequence of (value, priority) elements.
- **Operation (push):** Add new element to queue.
- **Operation (pop):** Remove largest-priority element.
- **Operation (front):** Return largest-priority element.
- **Operation (size):** Return the # elements in the queue.

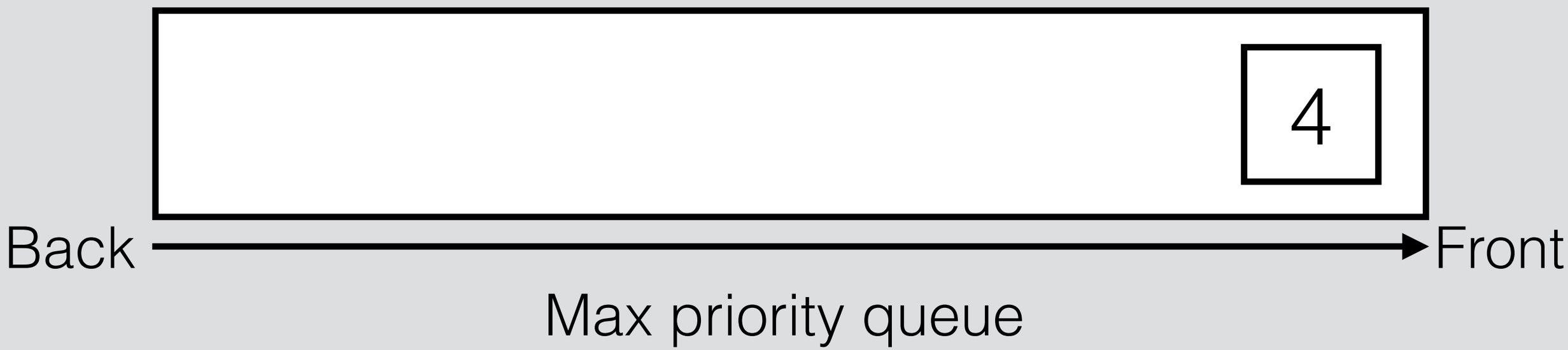
4 3 2

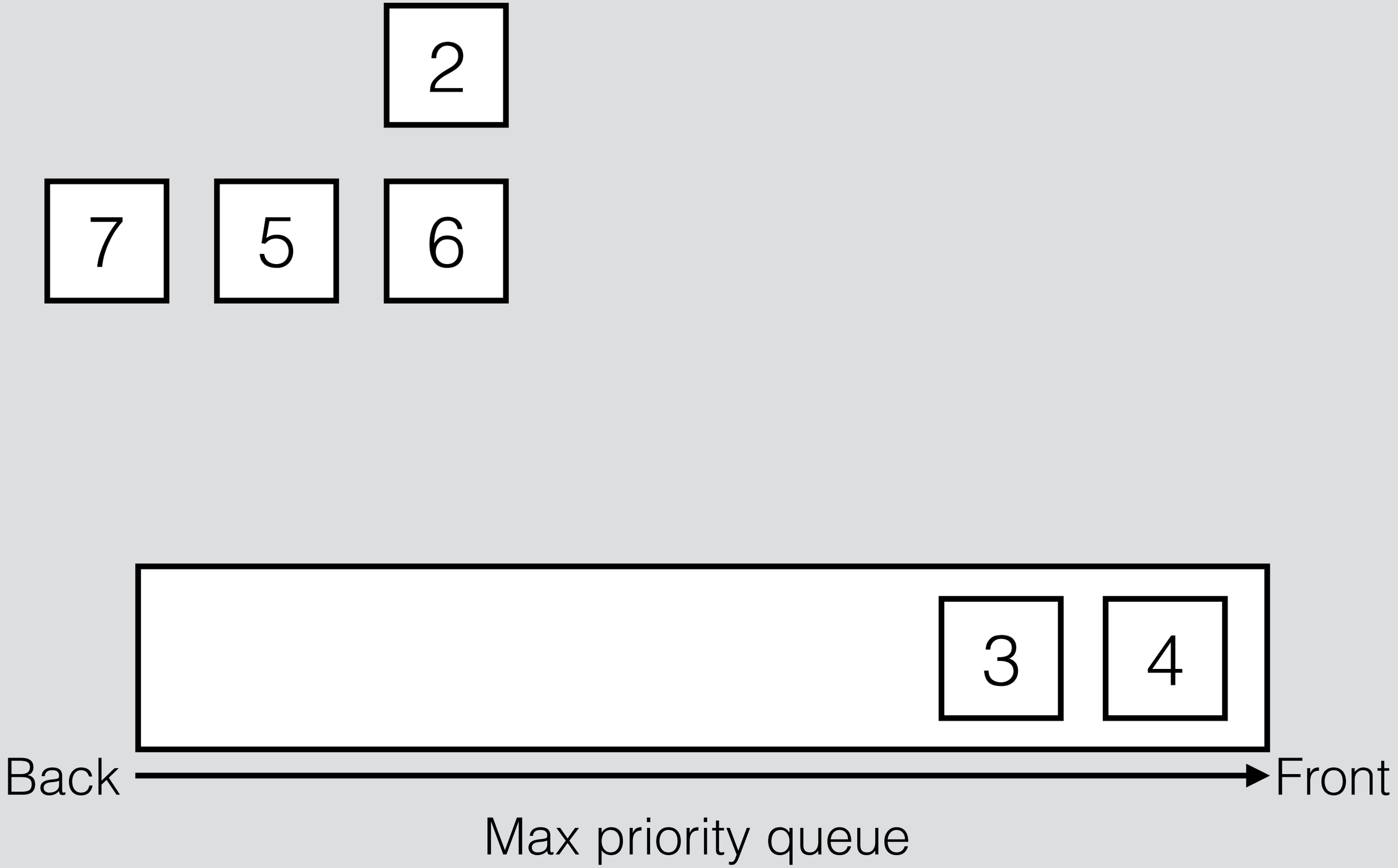
7 5 6



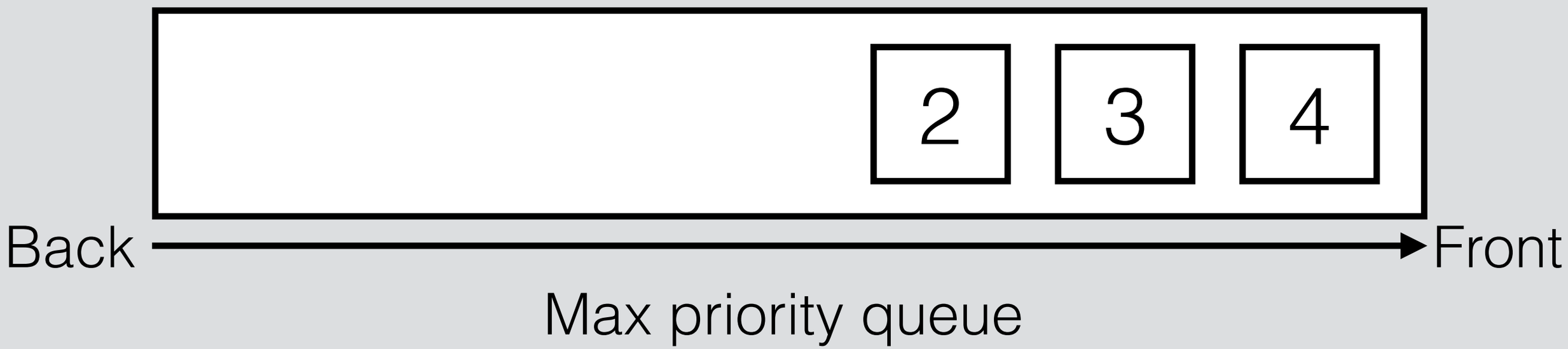
Max priority queue

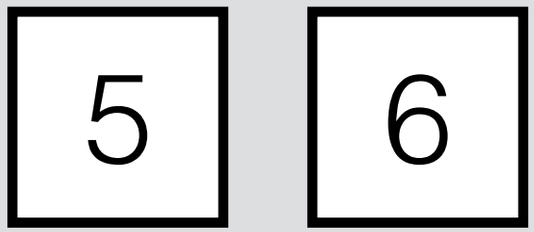

3
2
7
5
6

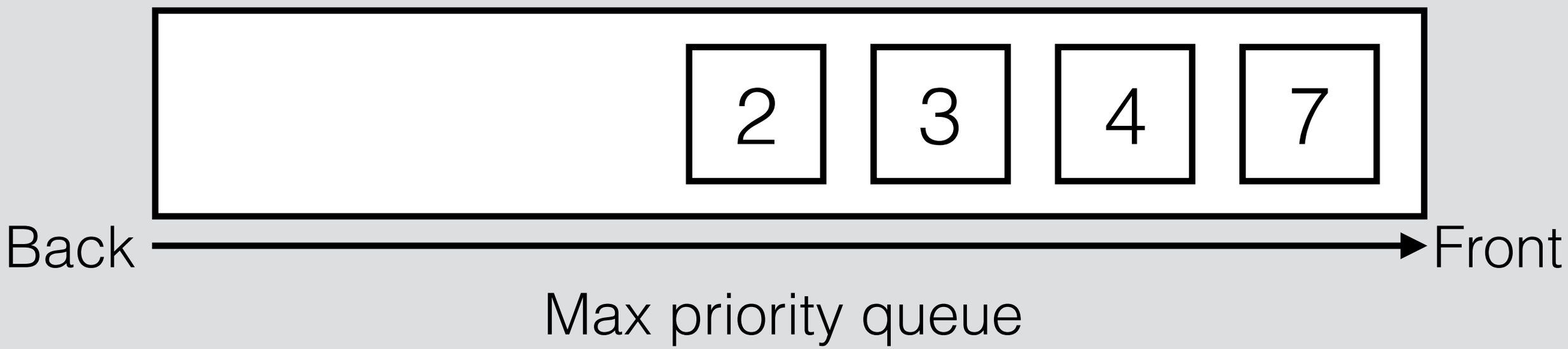

4
Back
Front
Max priority queue

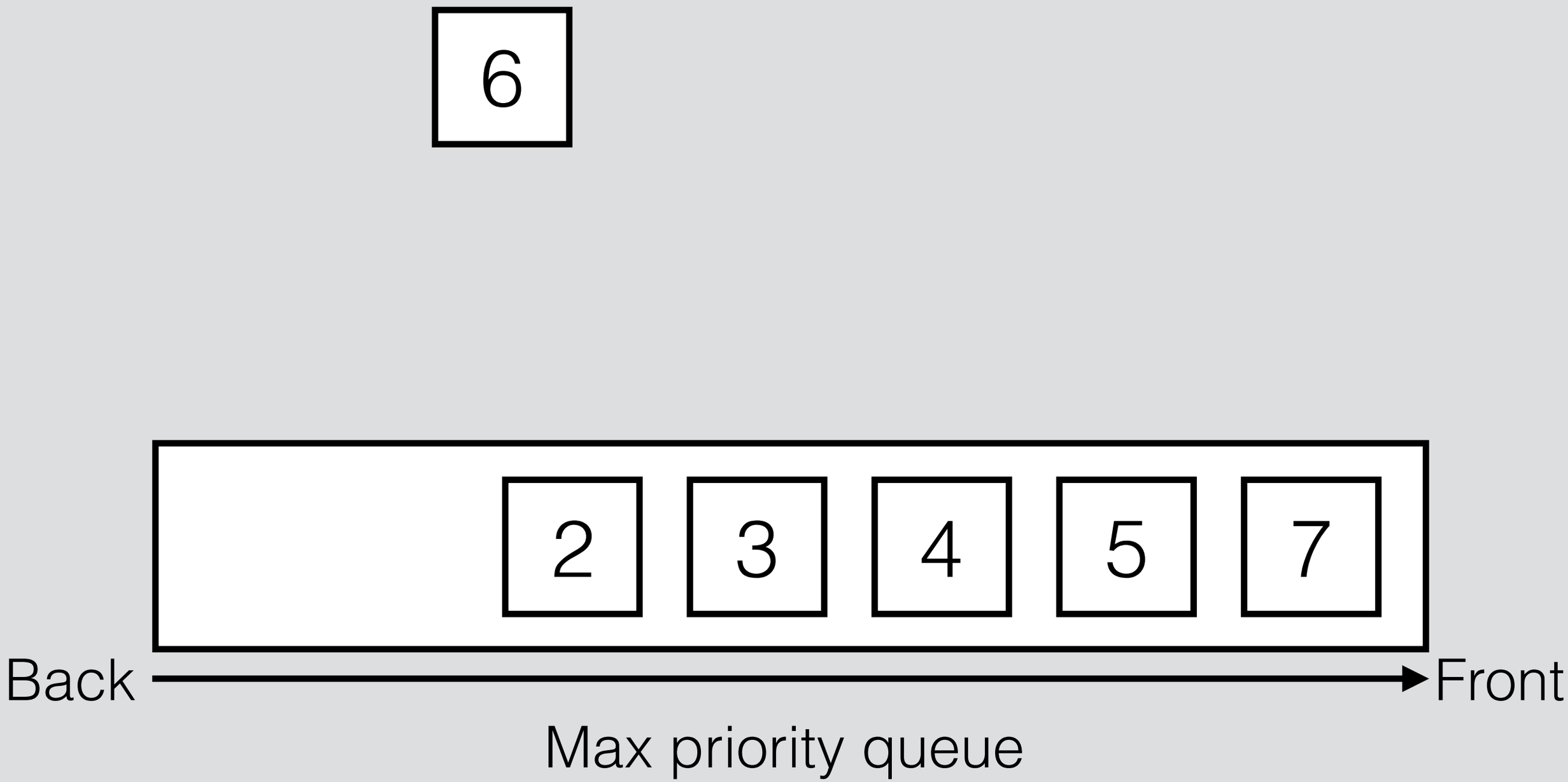

2
7
5
6
3
4
Back
Front
Max priority queue

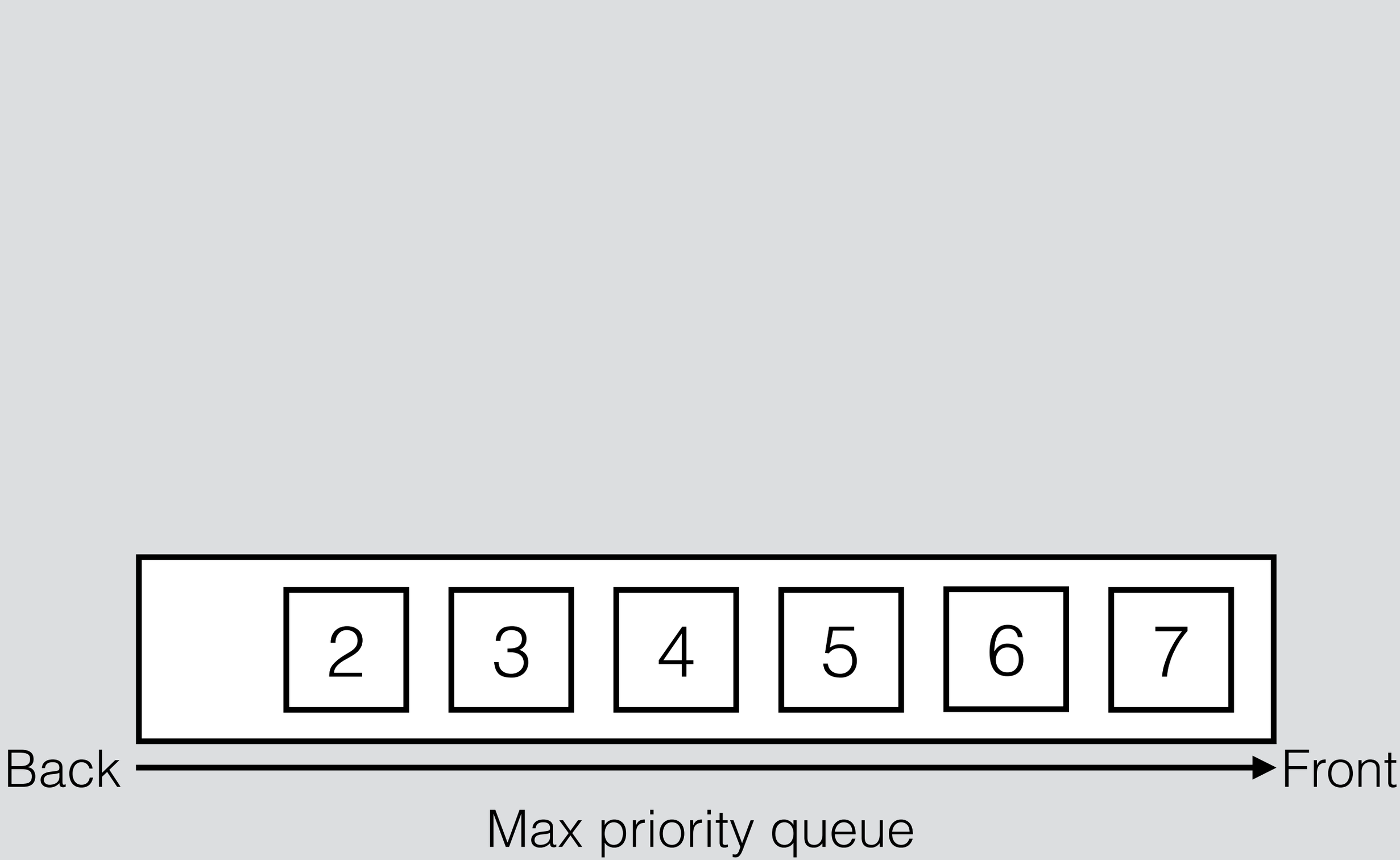

7
5
6

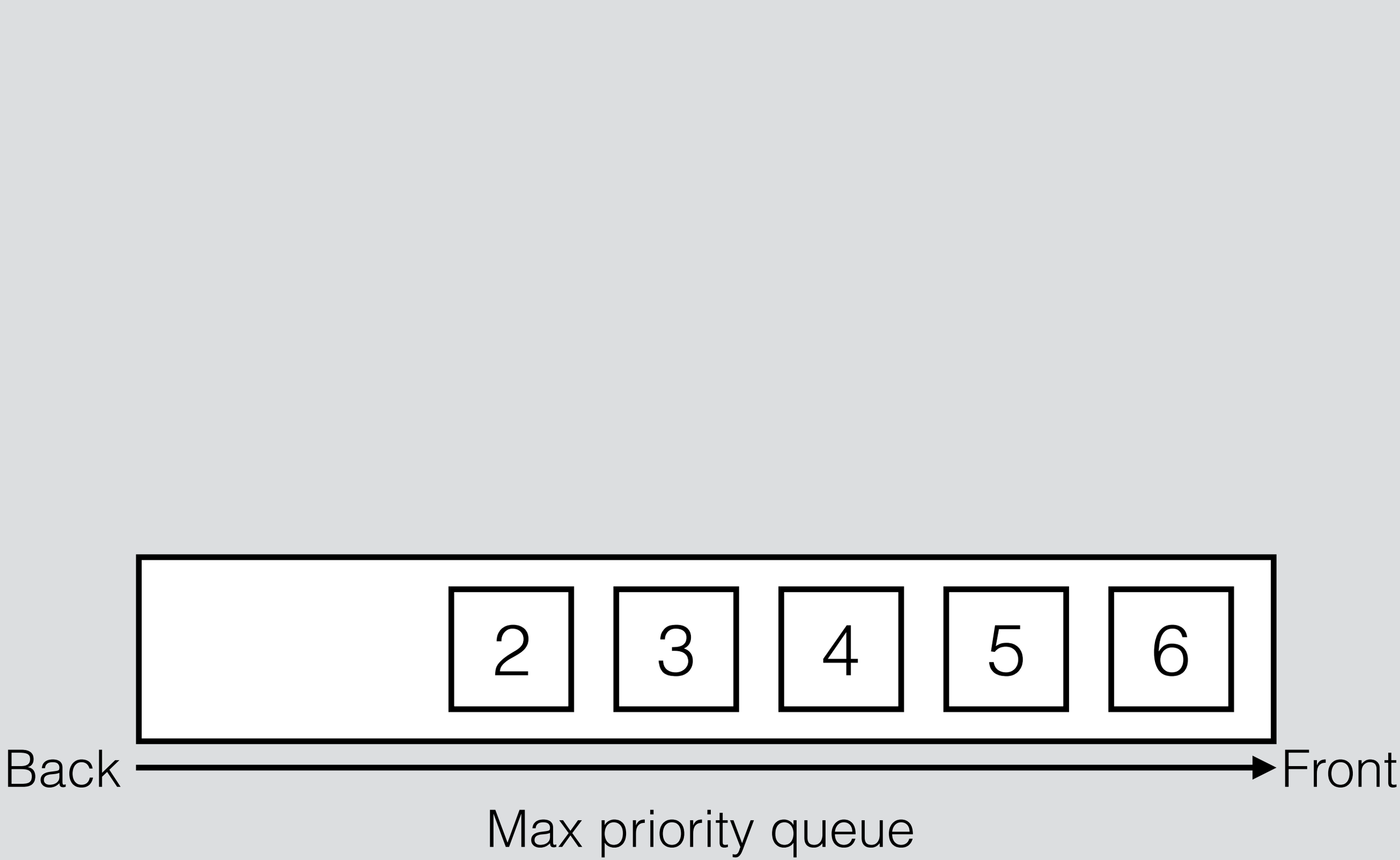

2
3
4
Back
Front
Max priority queue

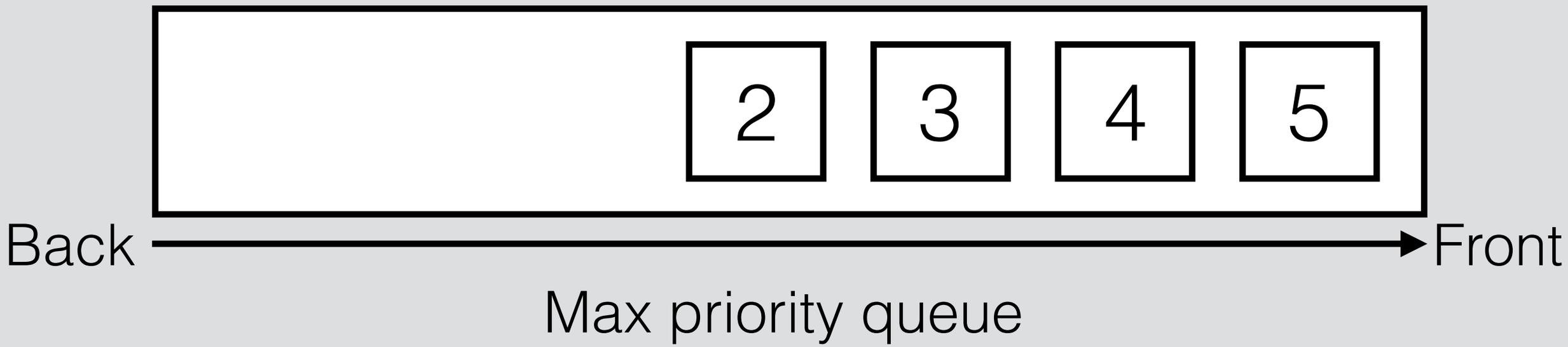

5
6


2
3
4
7
Back
Front
Max priority queue


6
2
3
4
5
7
Back
Front
Max priority queue


2
3
4
5
6
7
Back
Front
Max priority queue


2
3
4
5
6
Back
Front
Max priority queue


2
3
4
5
Back
Front
Max priority queue

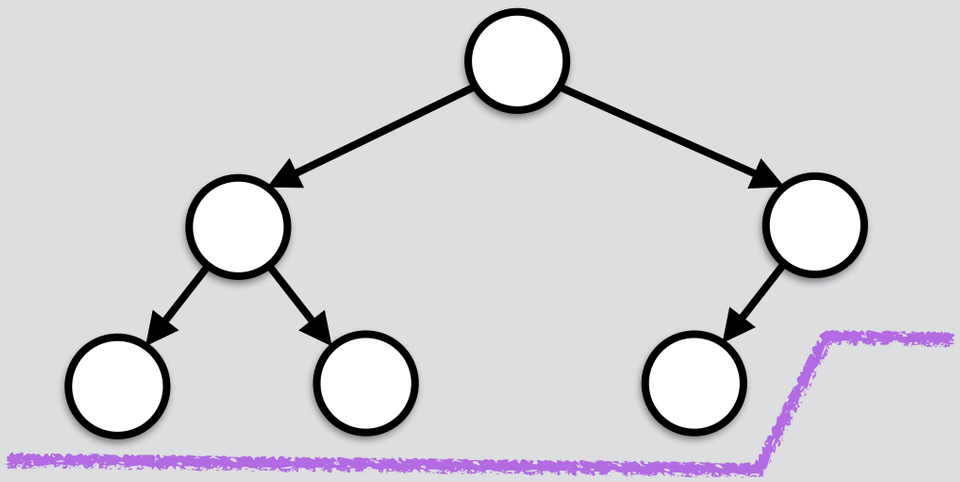
Min Priority Queue Worst-Case Running Times

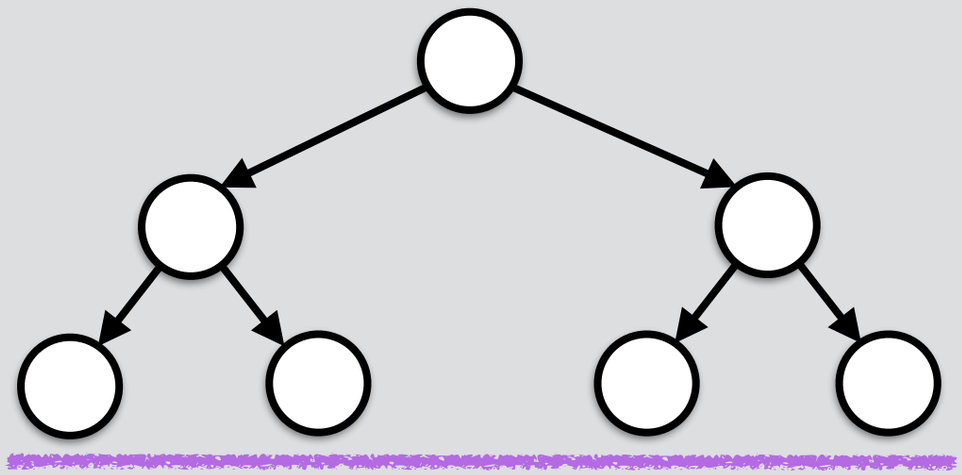
	Linked list:	AVL Tree:	Heap:
Operation (push):	Θ(n)	Θ(log(n))	Θ(log(n))
Operation (pop):	Θ(1)	Θ(log(n))	Θ(log(n))
Operation (front):	Θ(1)	Θ(1)	Θ(1)
Operation (size):	Θ(1)	Θ(1)	Θ(1)

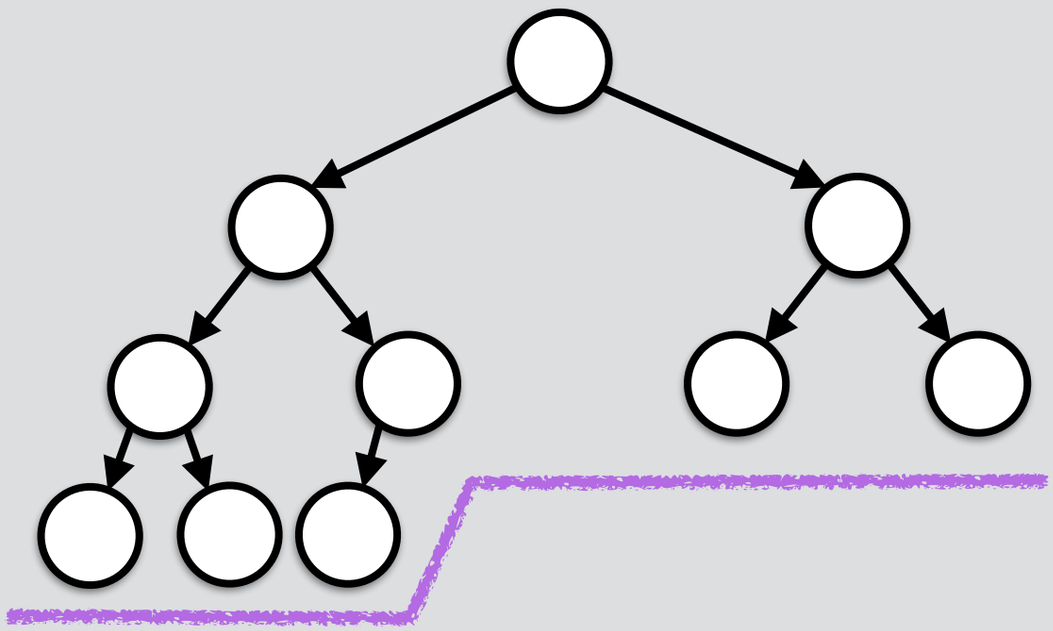
Heap Concept

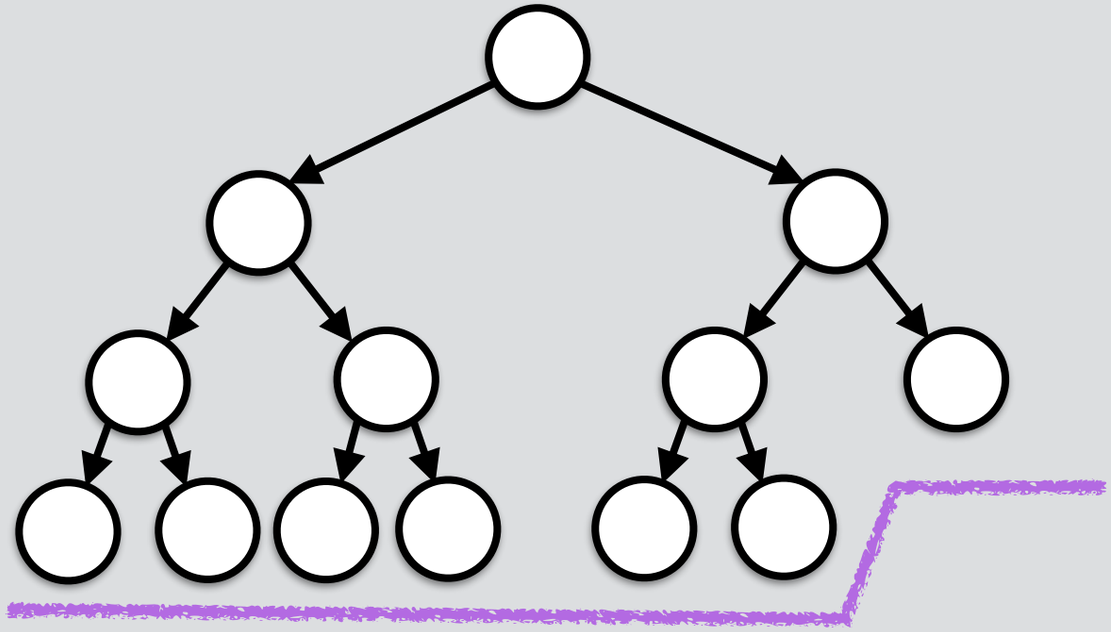
Complete trees

Only missing rightmost nodes in bottom level

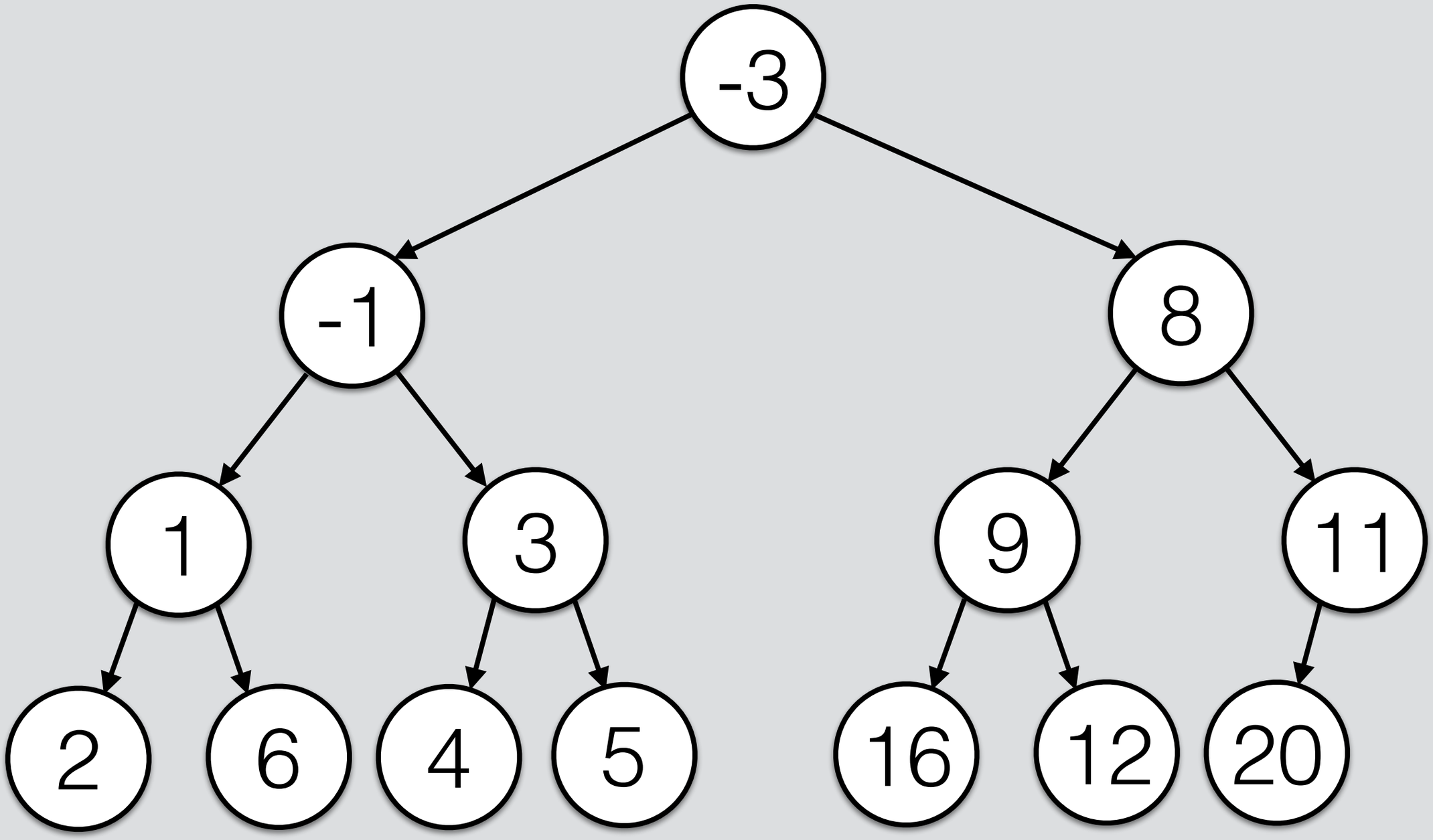




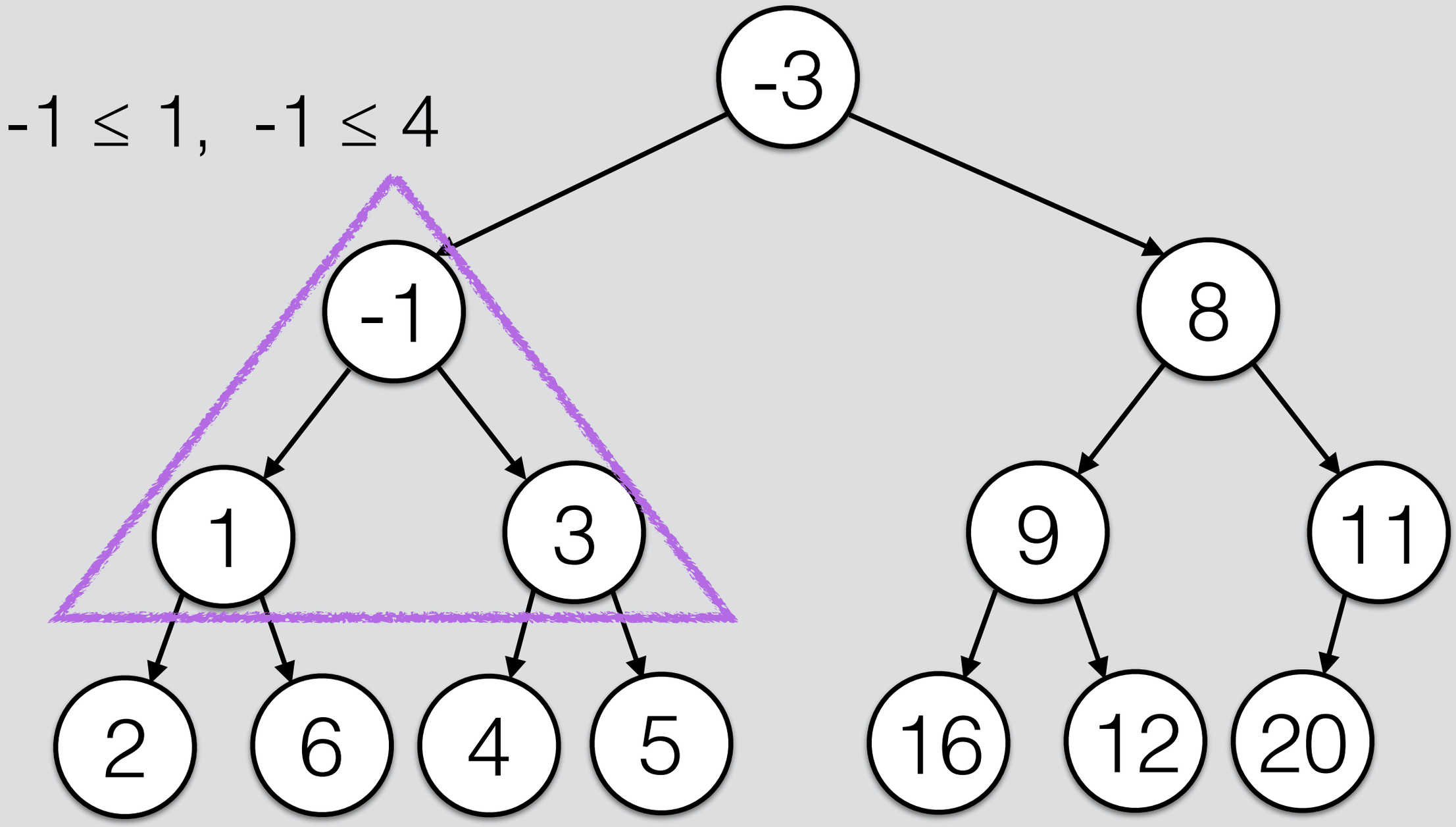




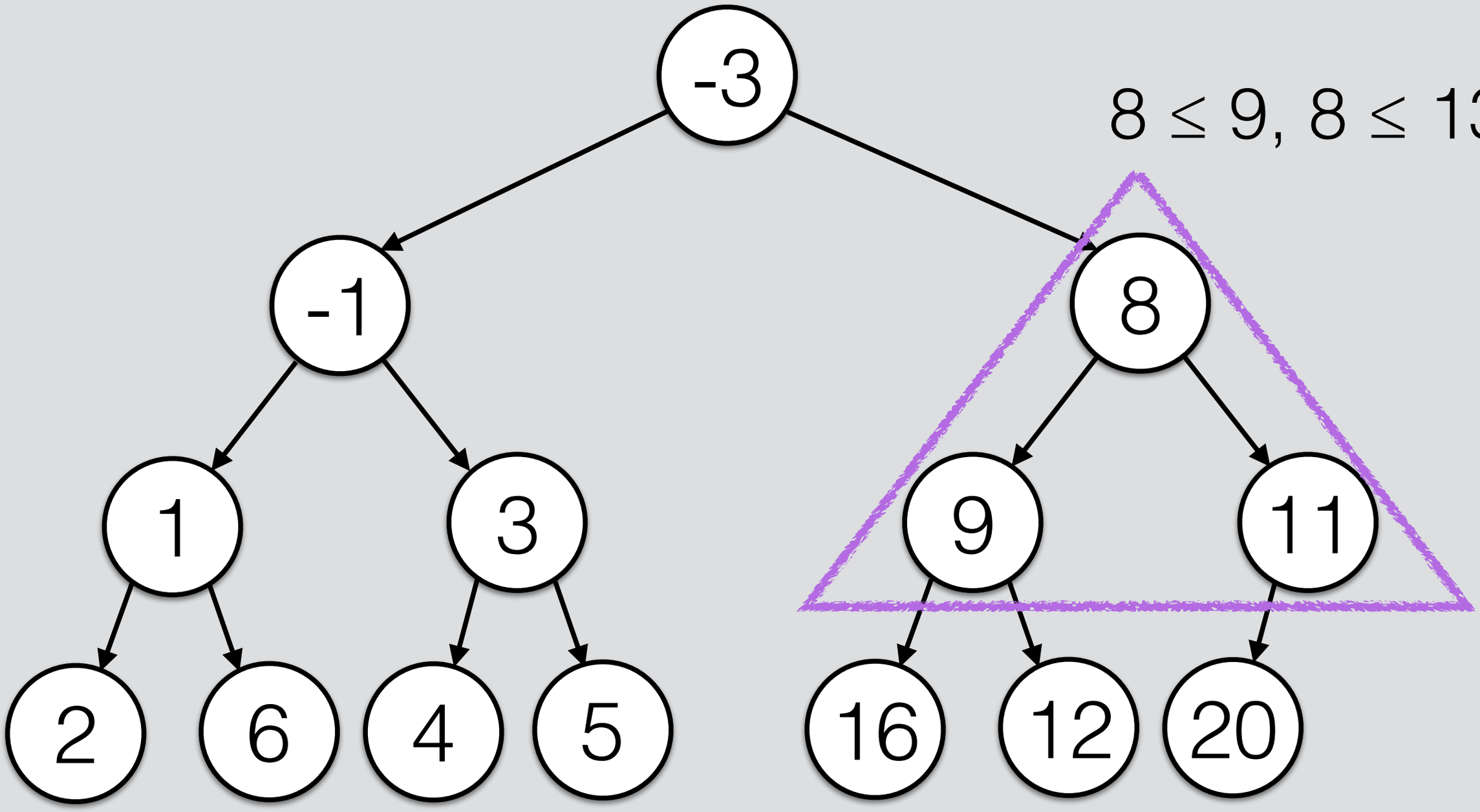
Min-Heap



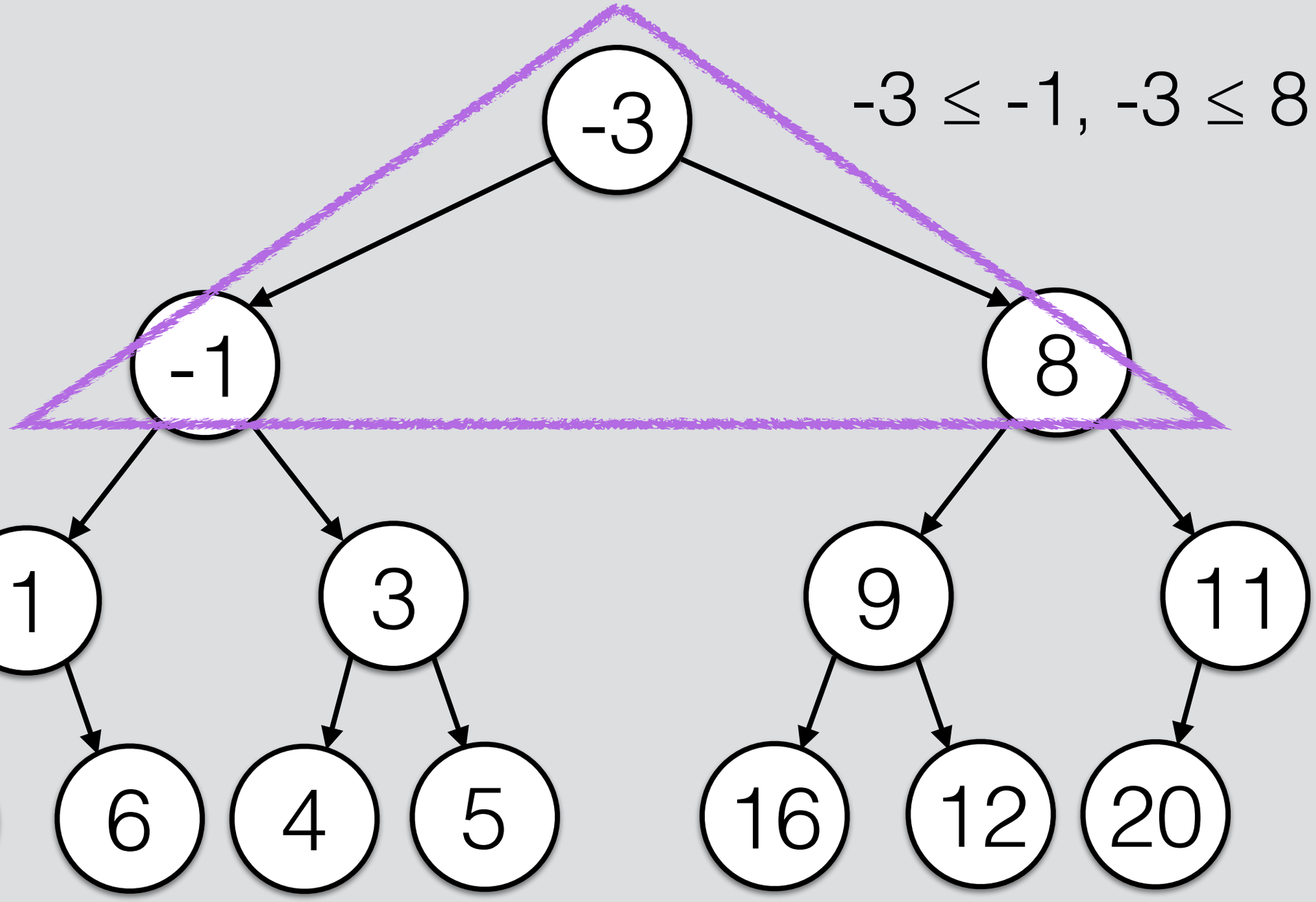
Min-Heap



Min-Heap

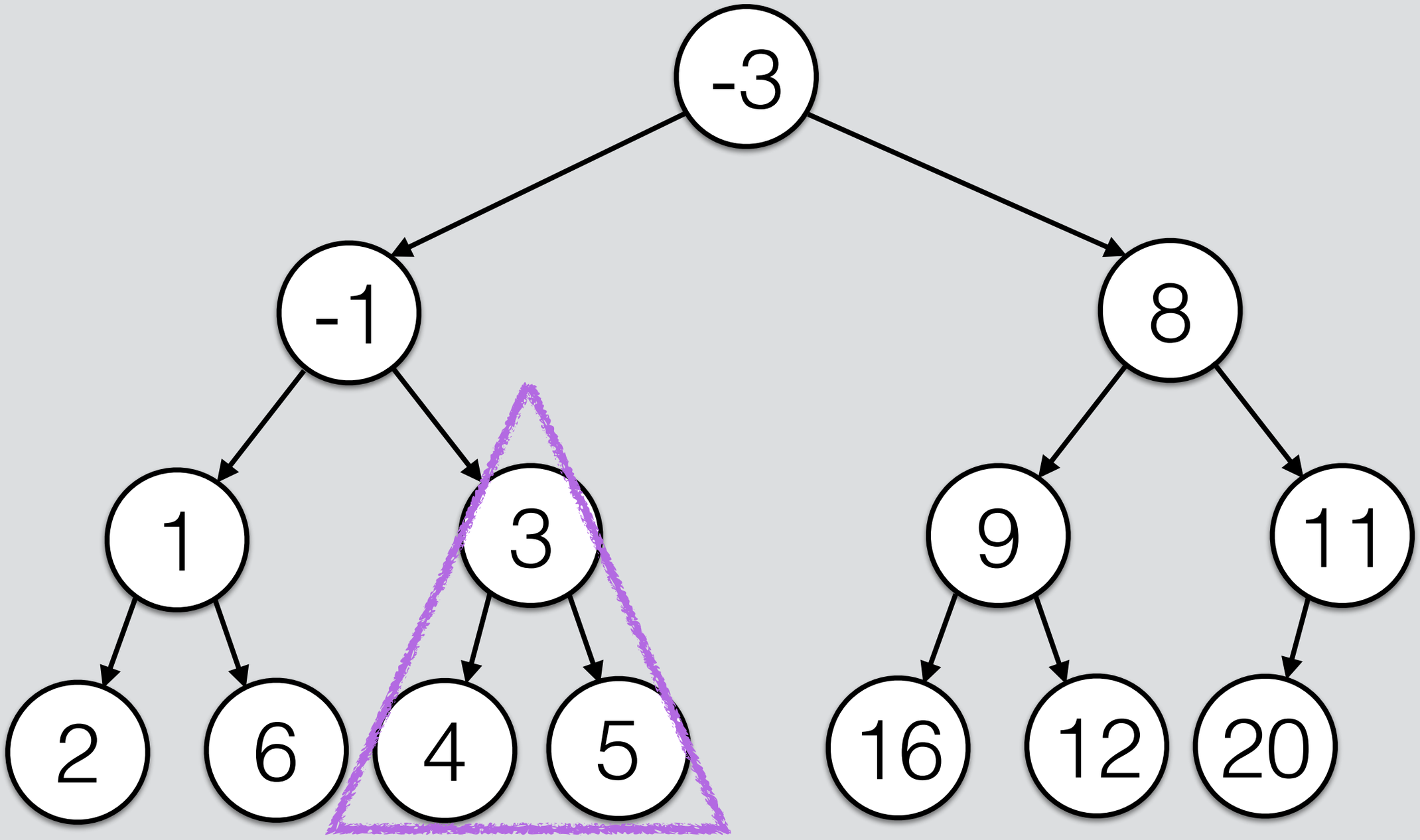


Min-Heap



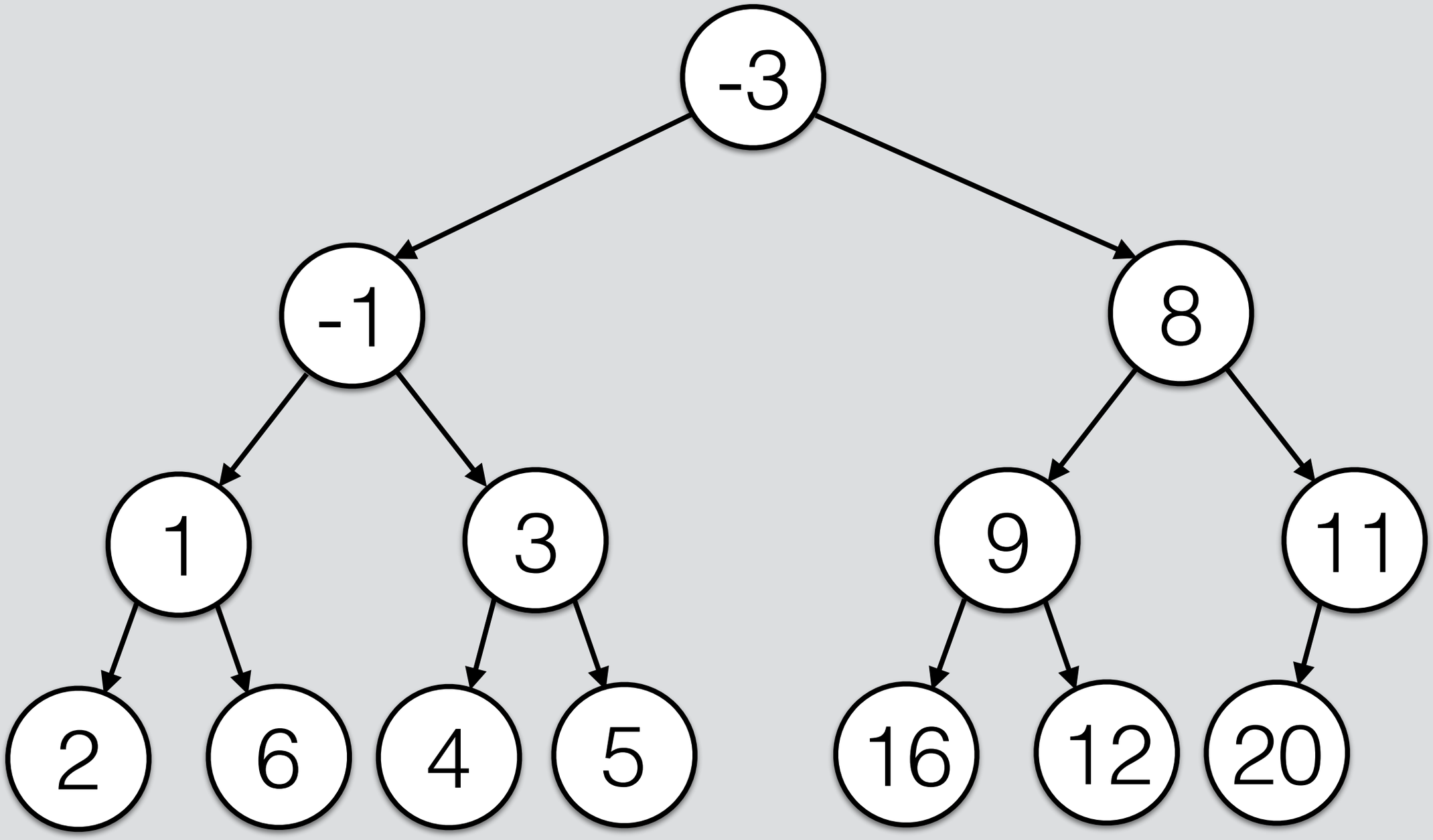
$-3 \leq -1,\ -3 \leq 8$

Min-Heap



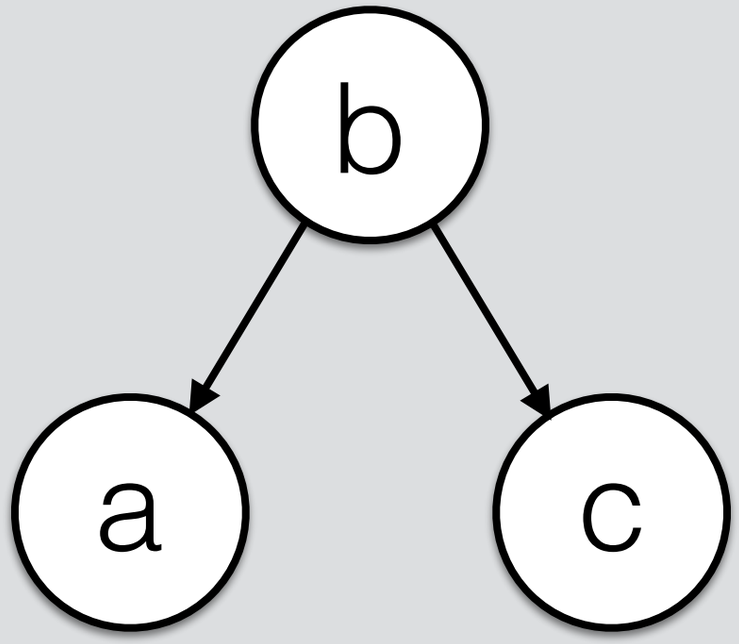
$3 \leq 4, \quad 3 \leq 5$

Min-Heap

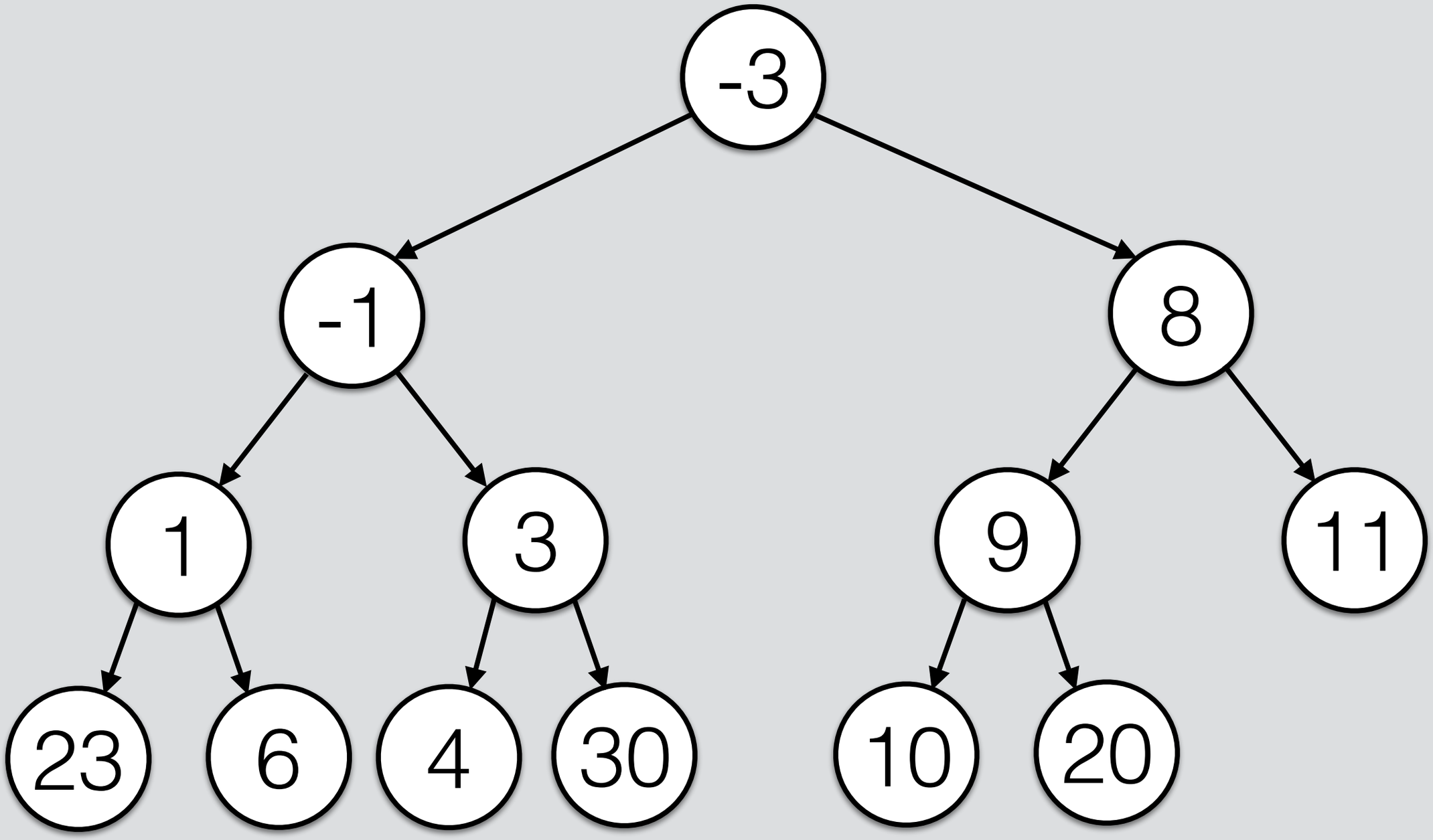


Min-heap rule

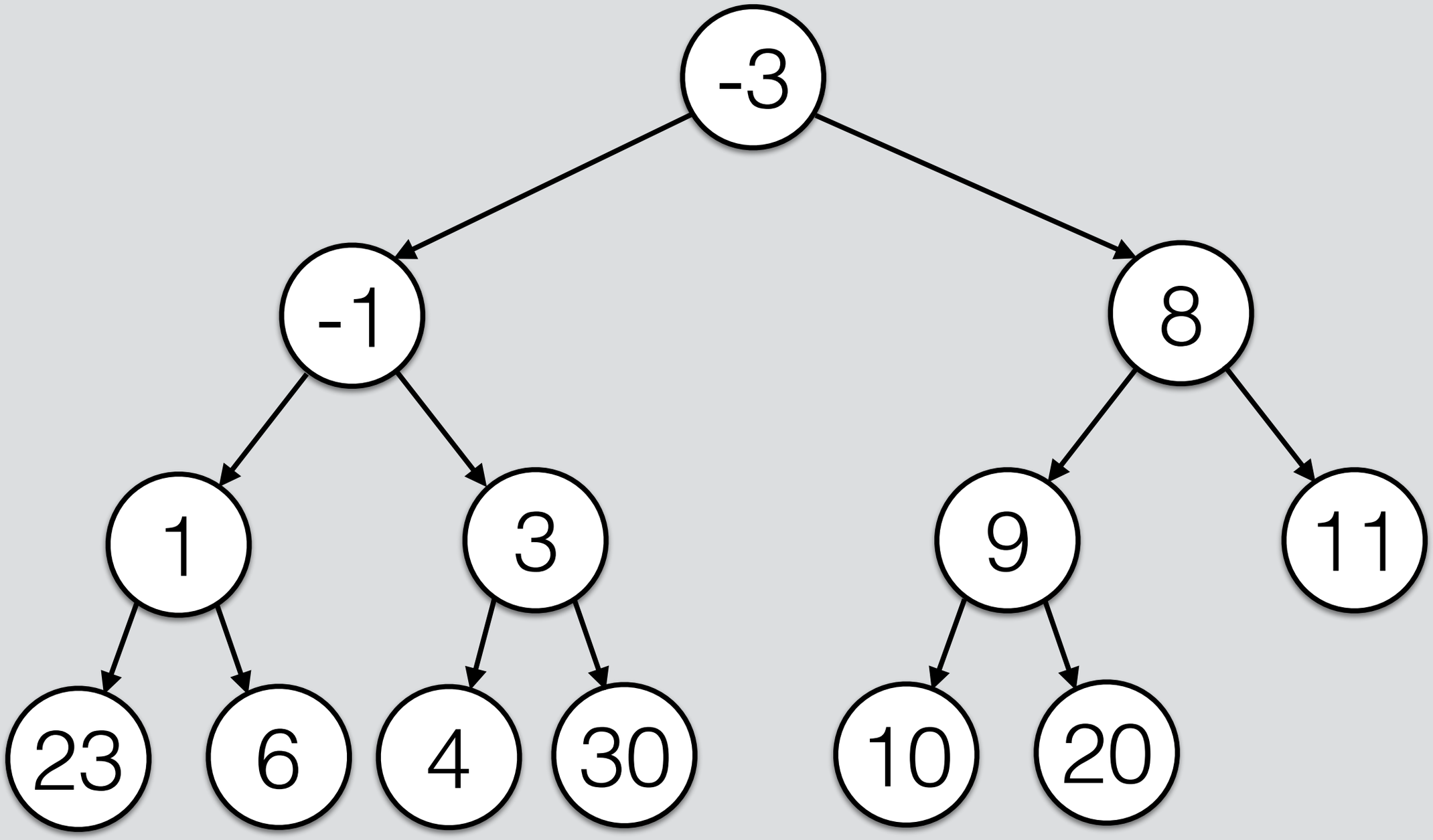
For every node in a min-heap, b ≤ a, b ≤ c.



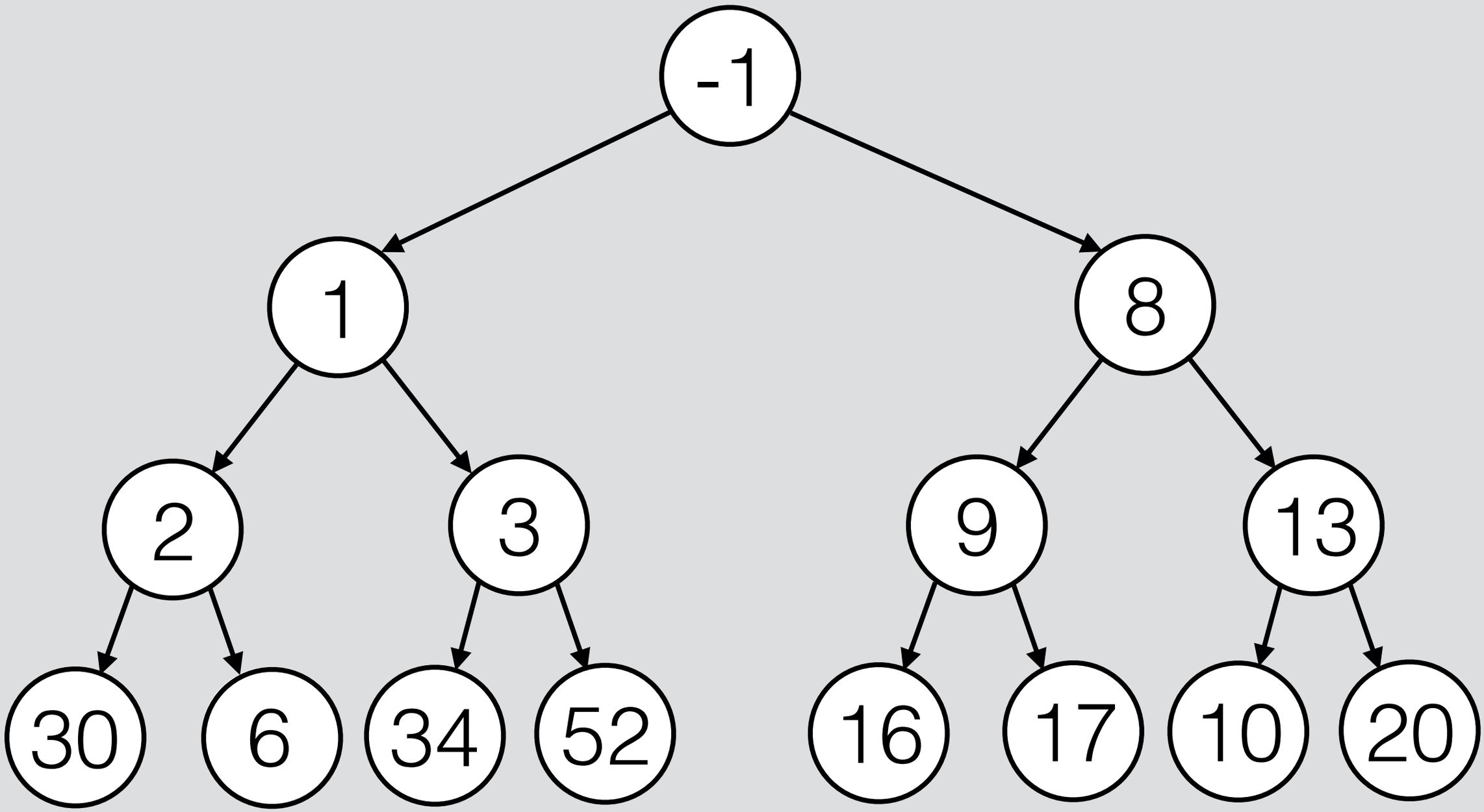
- Where is the smallest element in a min-heap?
- Where is the largest element in a min-heap?
- Is the min-heap of a set of elements unique?



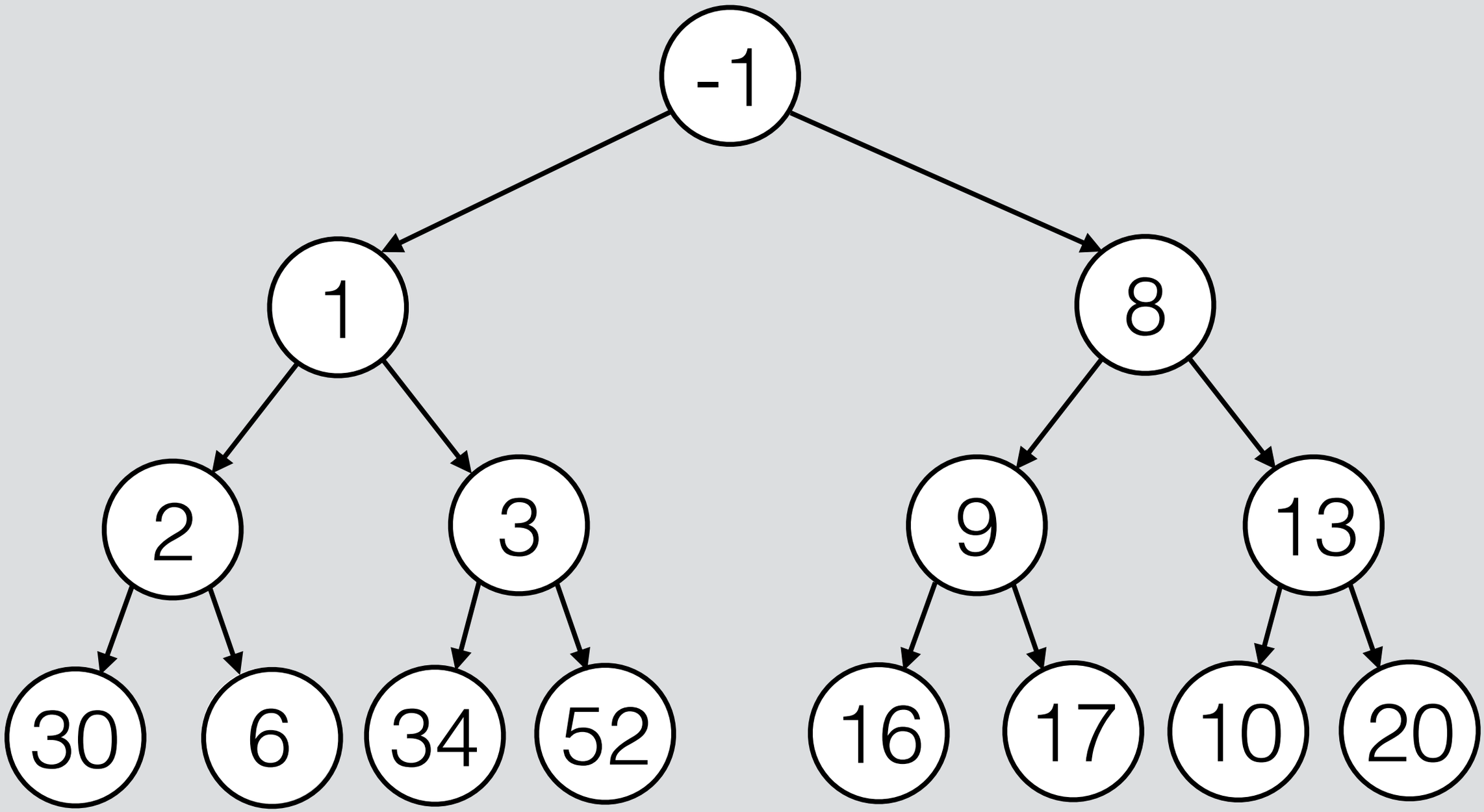
- Where is the smallest element in a min-heap? Root
- Where is the largest element in a min-heap? Leaf
- Is the min-heap of a set of elements unique? No



pop()

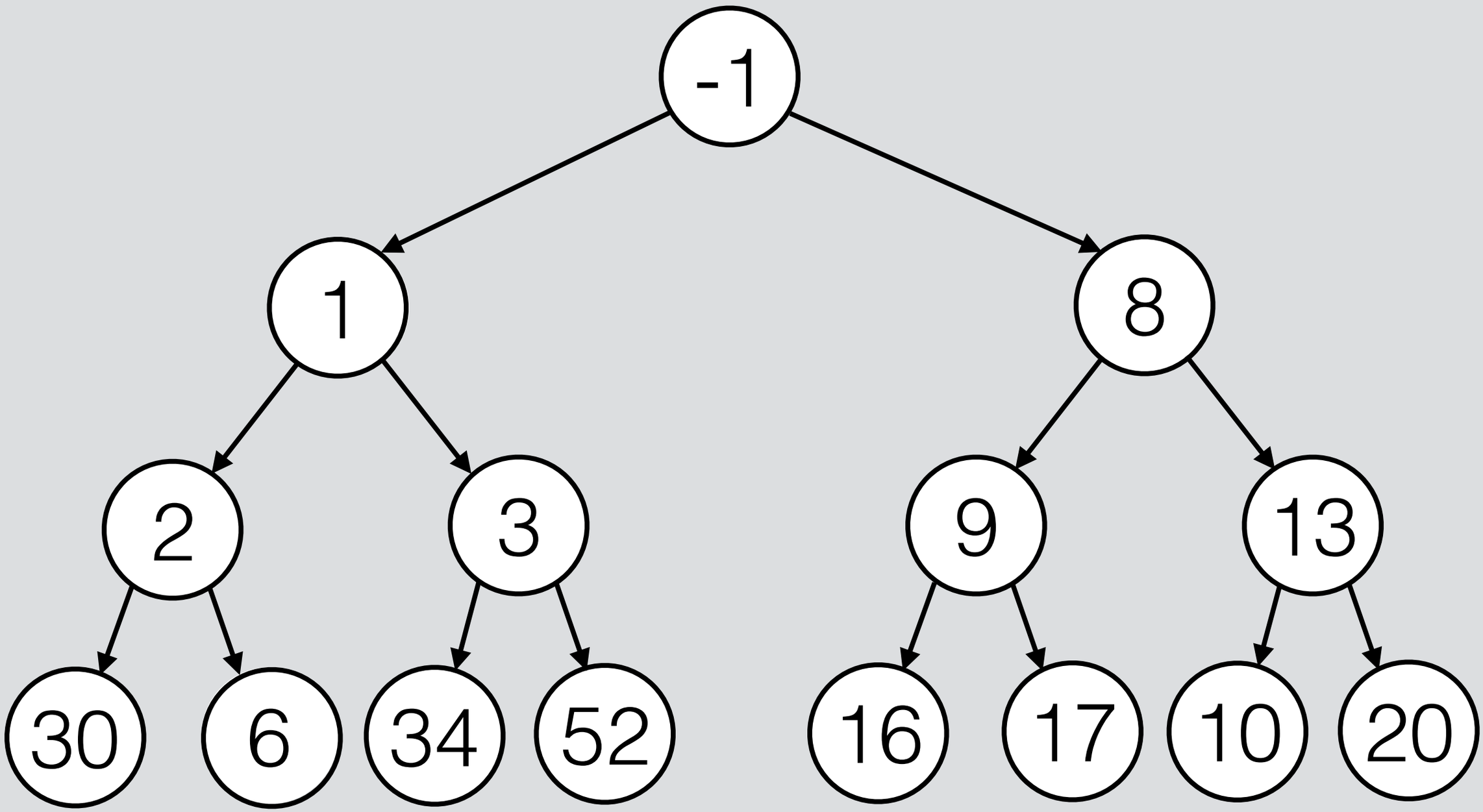

-1
1
8
2
3
9
13
30
6
34
52
16
17
10
20

pop();



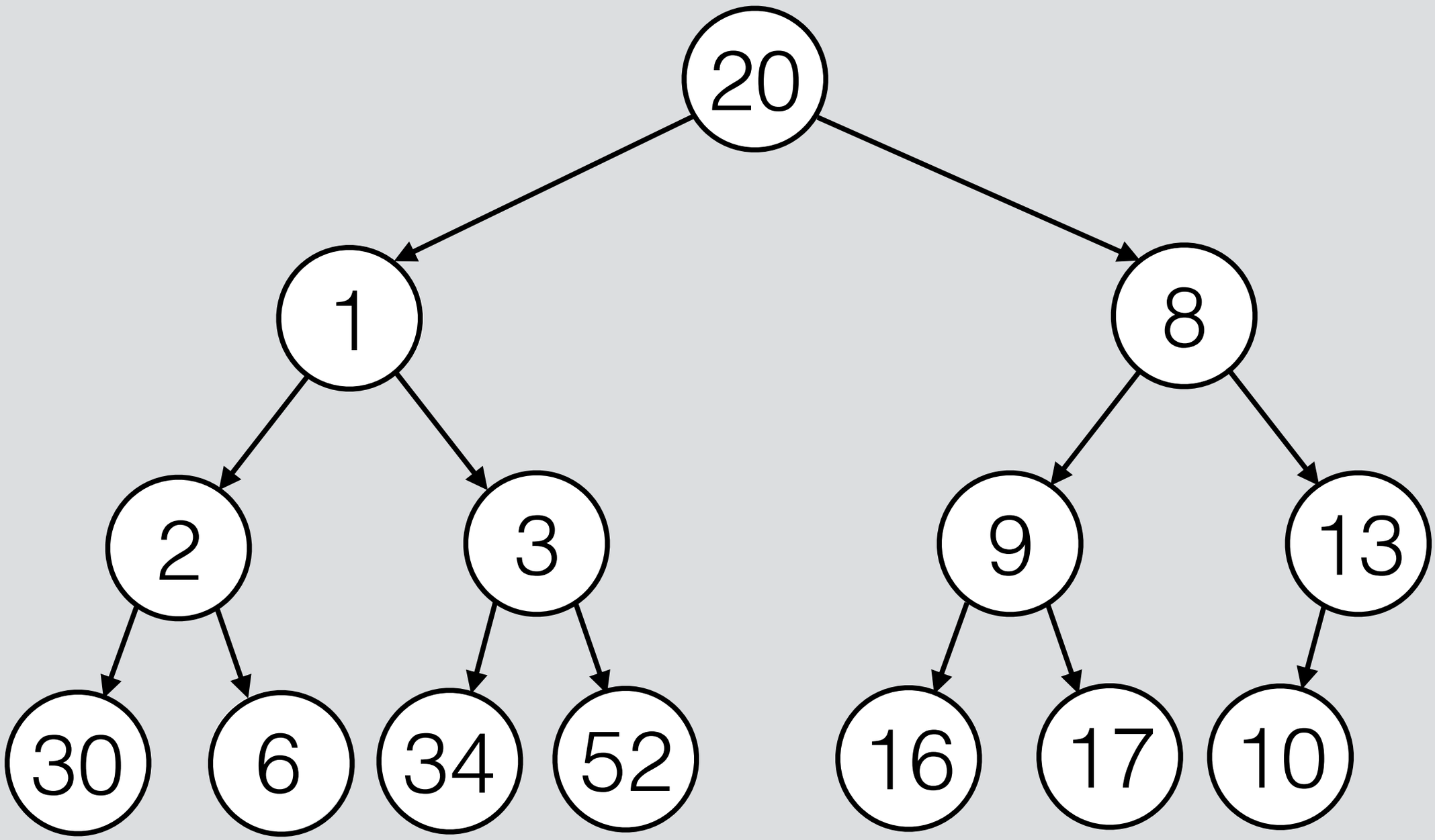
1. Replace root.

```
pop();
```



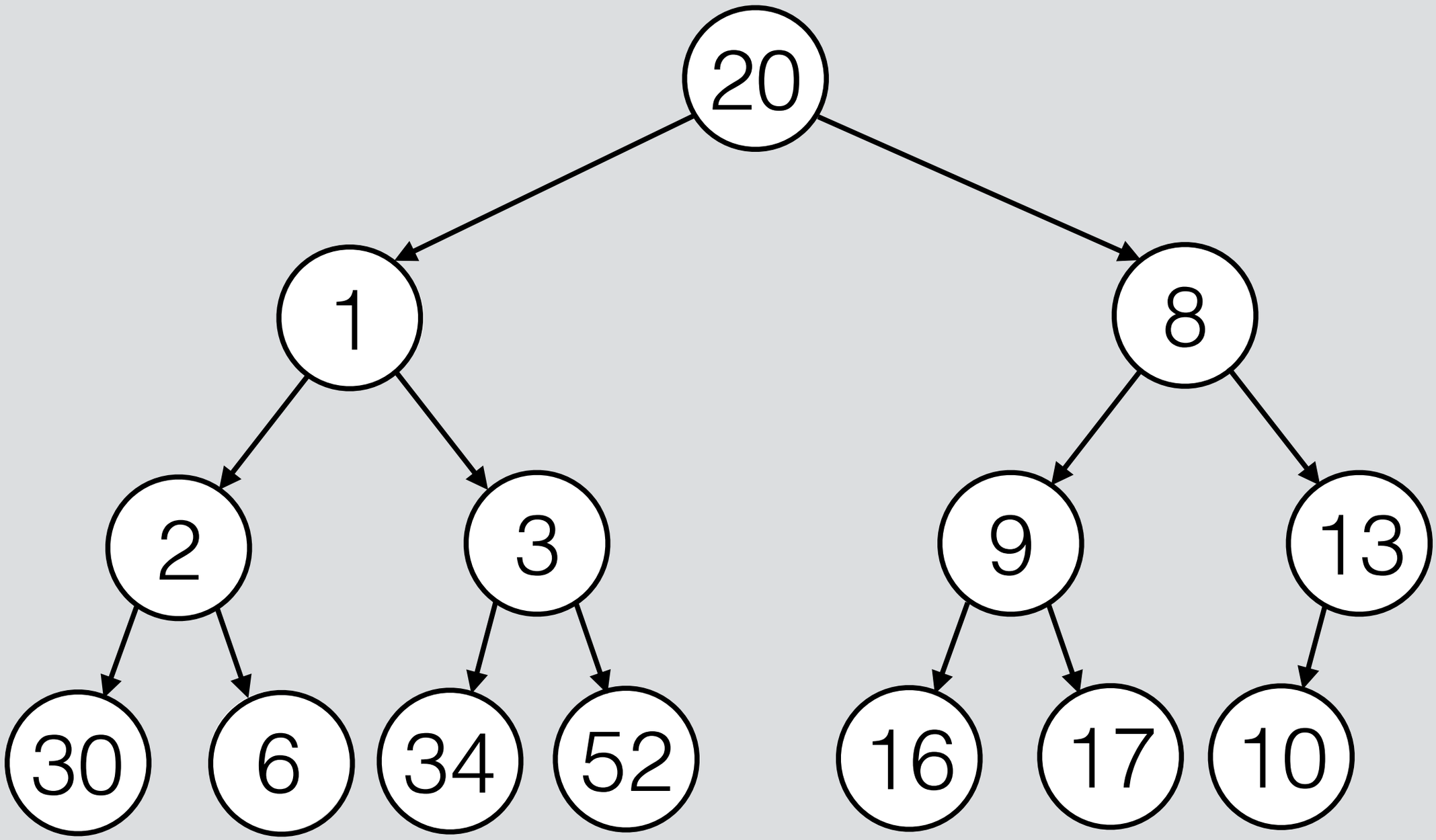
1. Replace root.

`pop();`



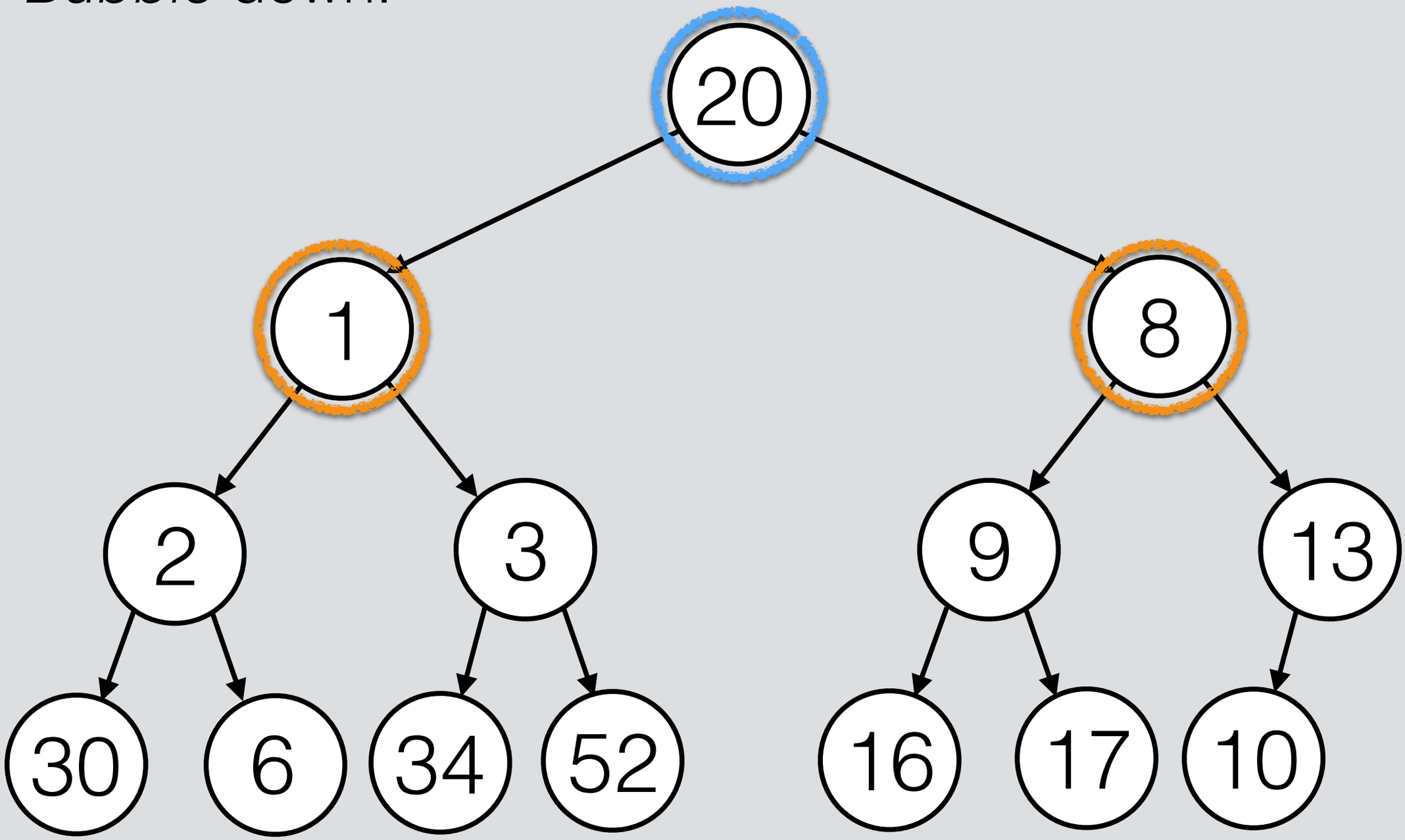
1. Replace root.
2. *Bubble* down.

`pop();`



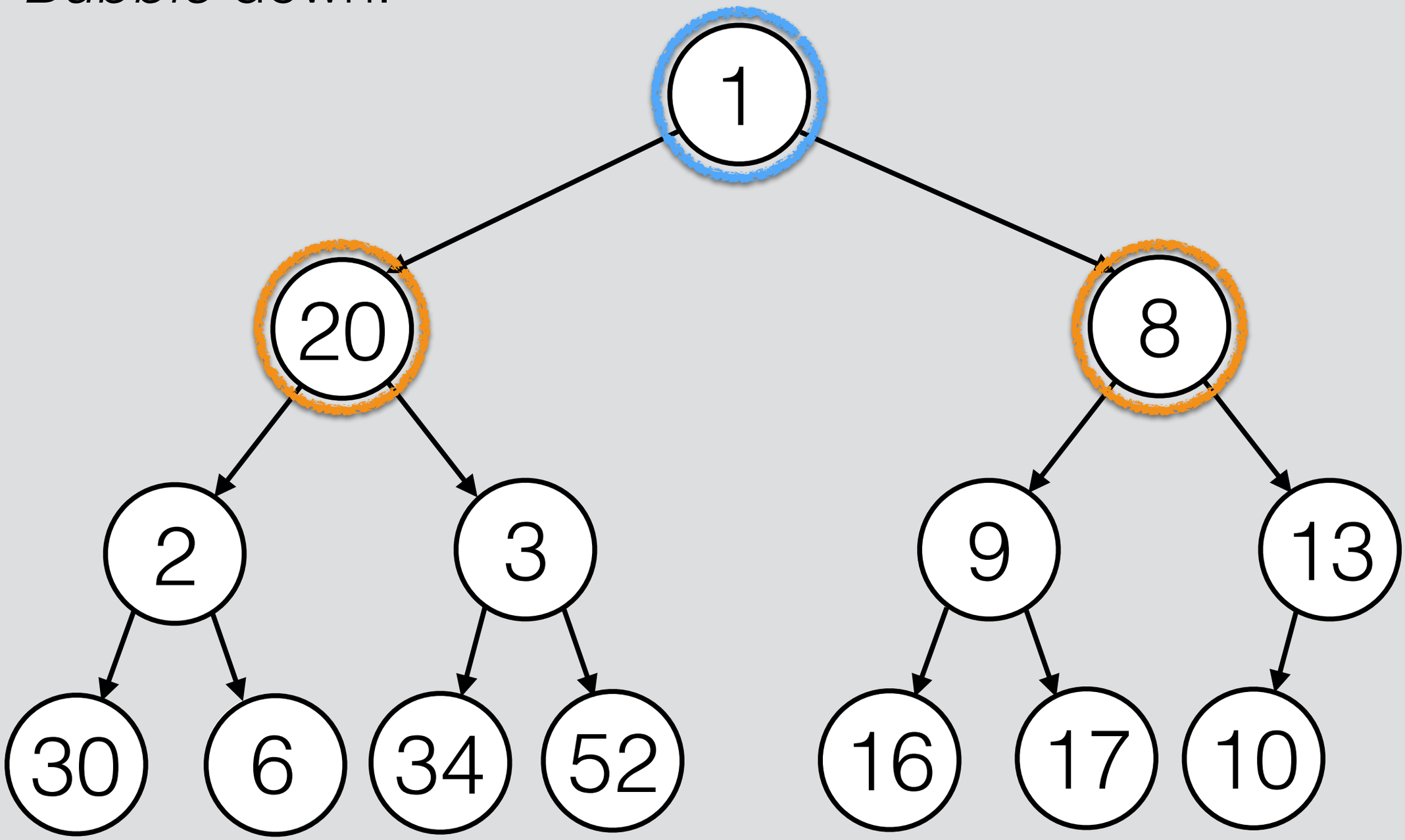
1. Replace root.
2. *Bubble* down.

`pop();`



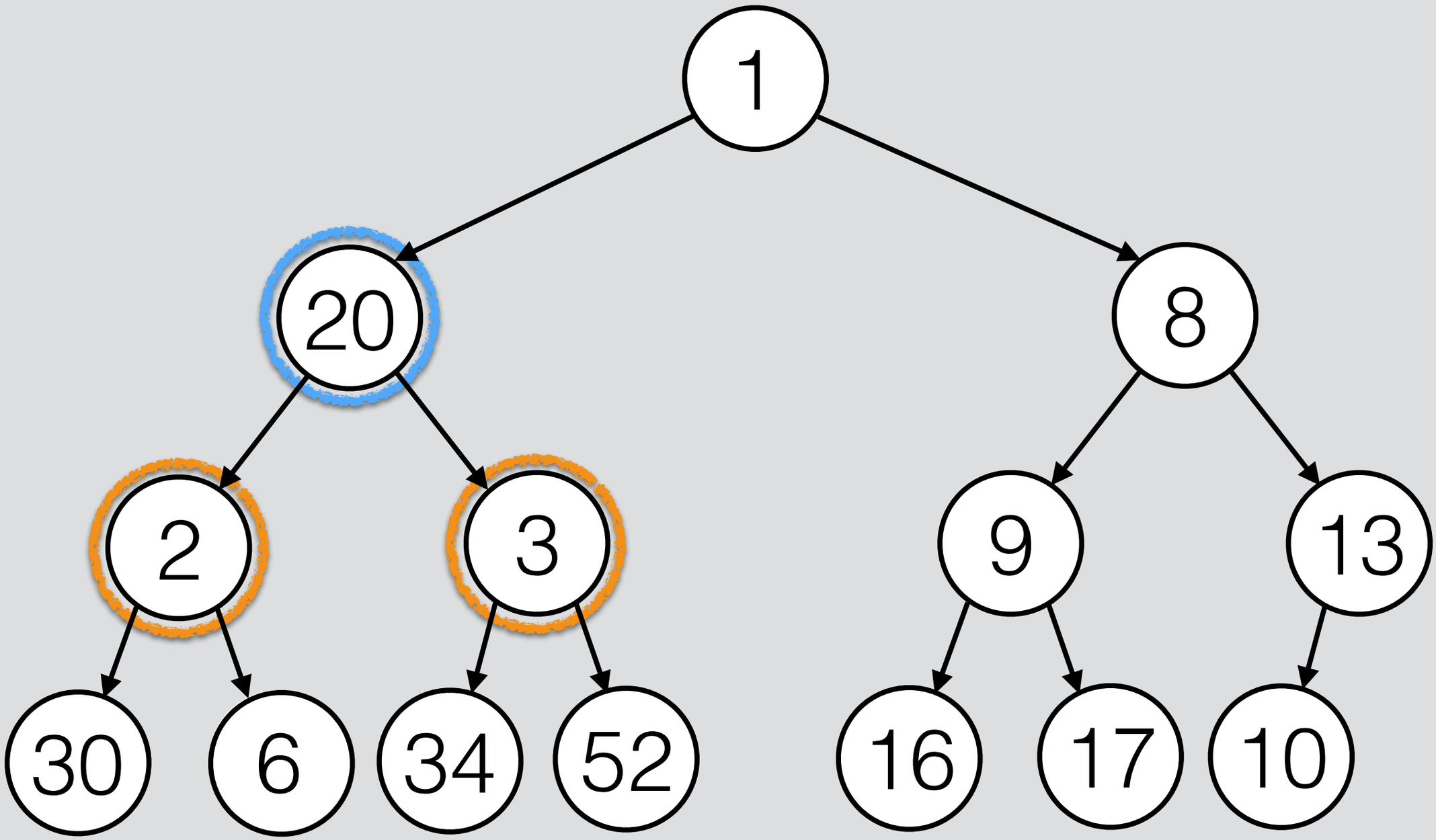
1. Replace root.
2. *Bubble* down.

`pop();`



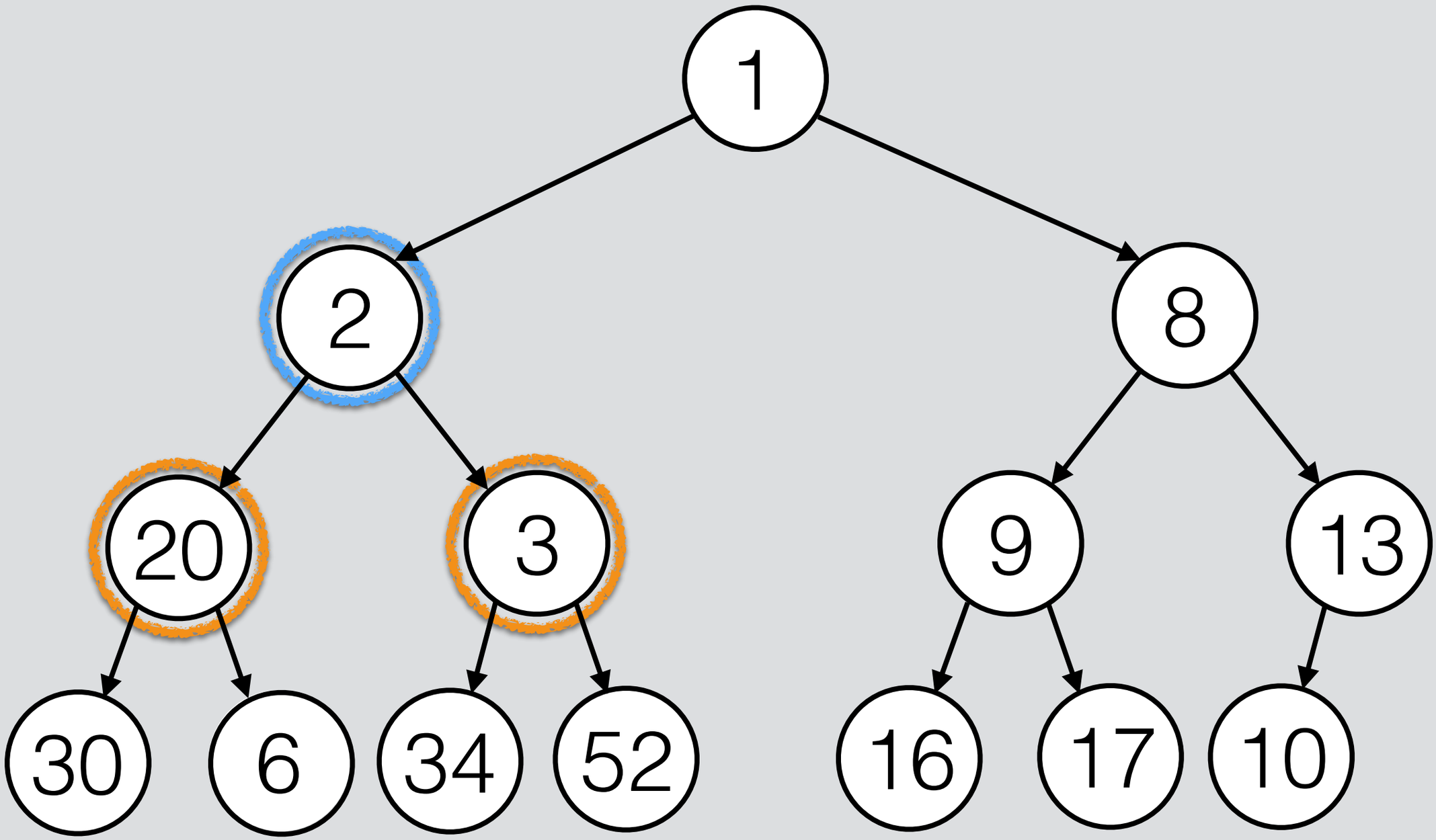
1. Replace root.
2. *Bubble* down.

`pop();`



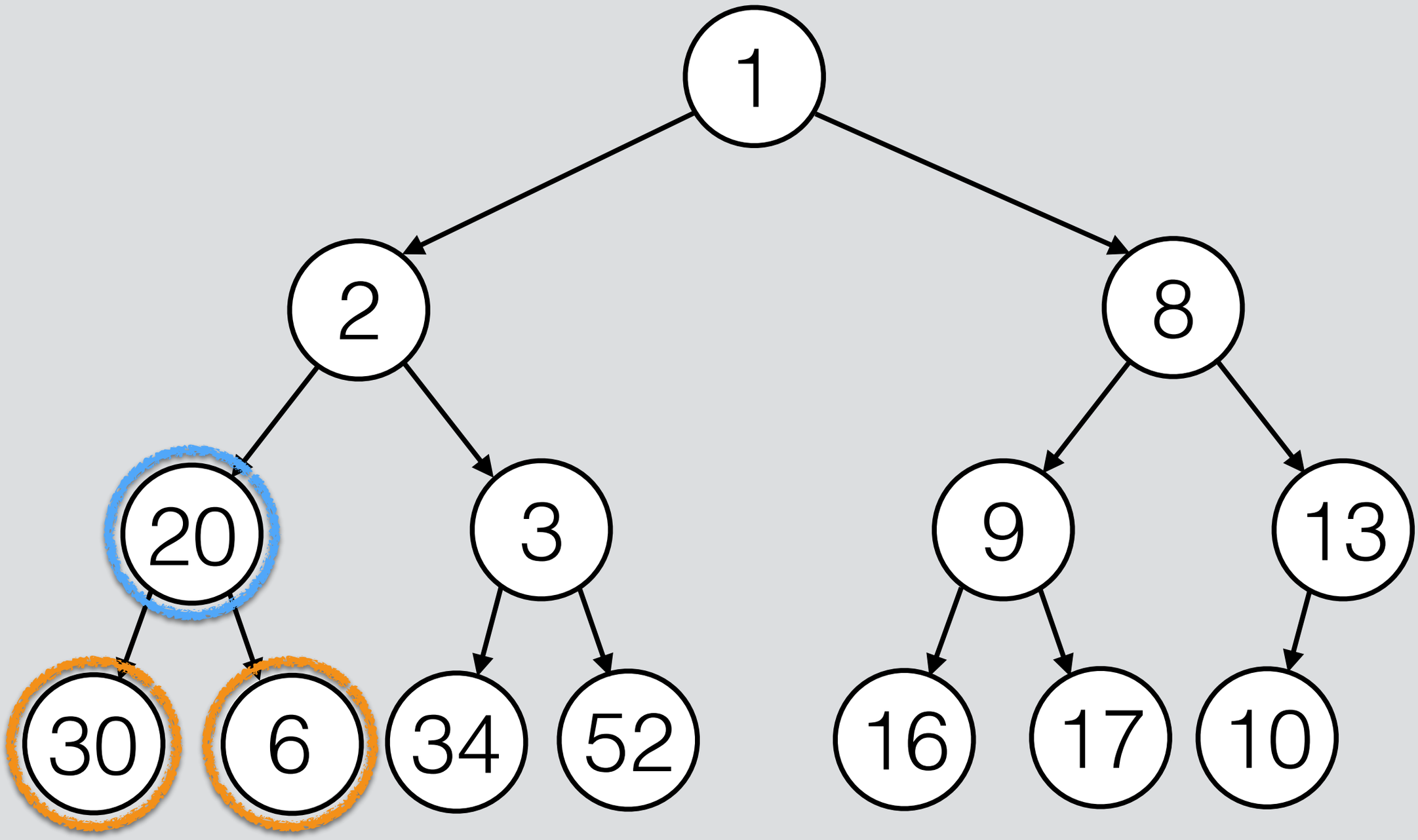
1. Replace root.
2. *Bubble* down.

`pop();`



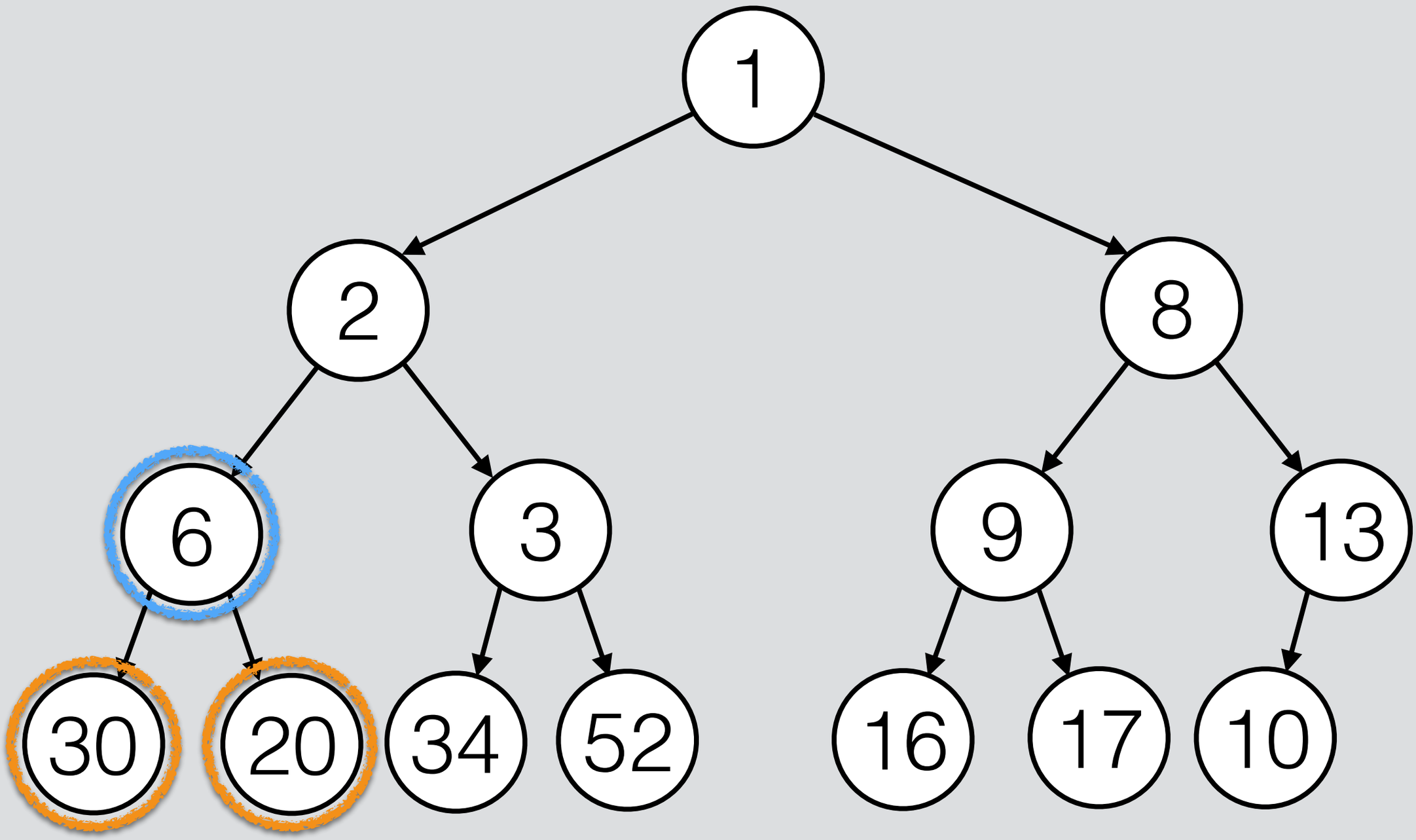
1. Replace root.
2. *Bubble* down.

`pop();`



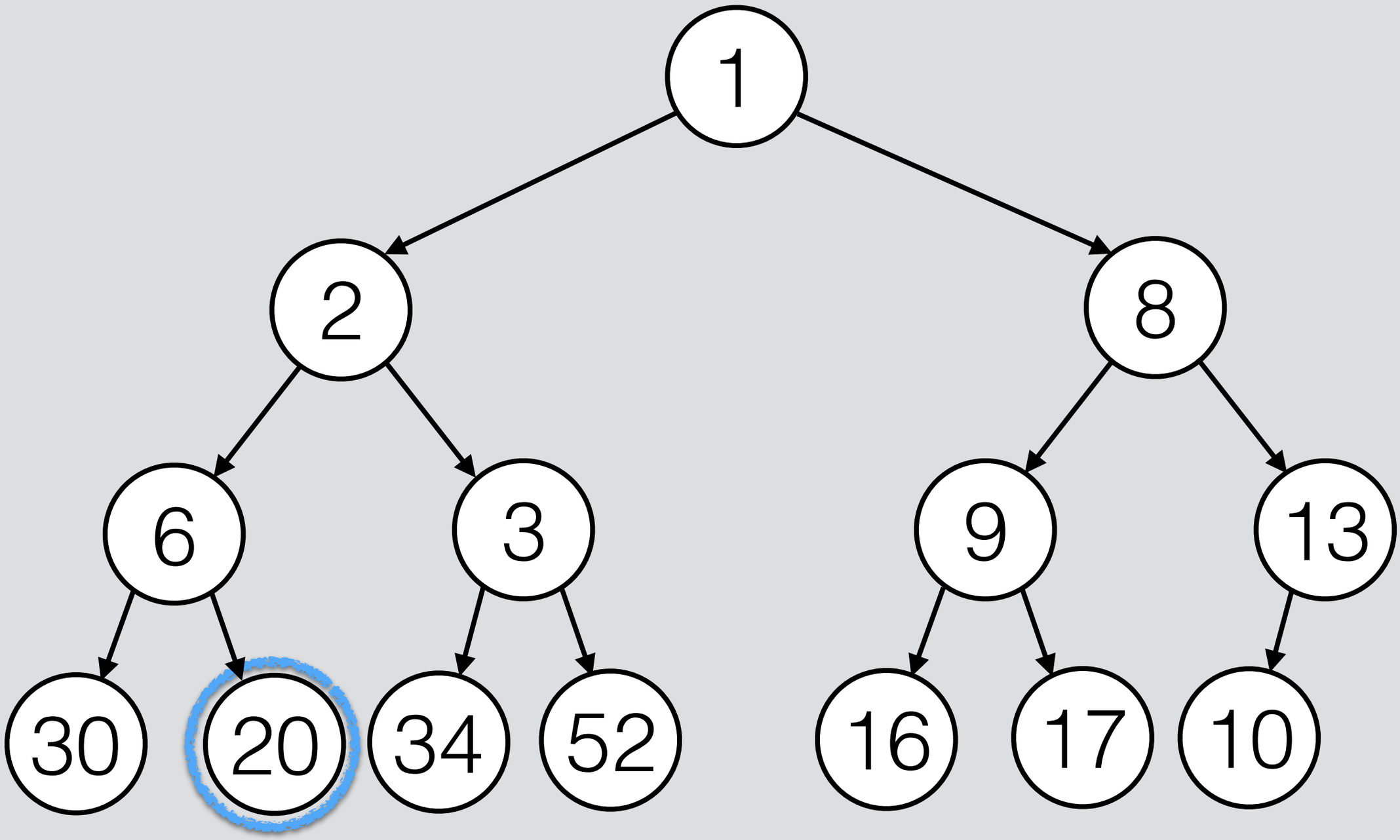
1. Replace root.
2. *Bubble* down.

`pop();`



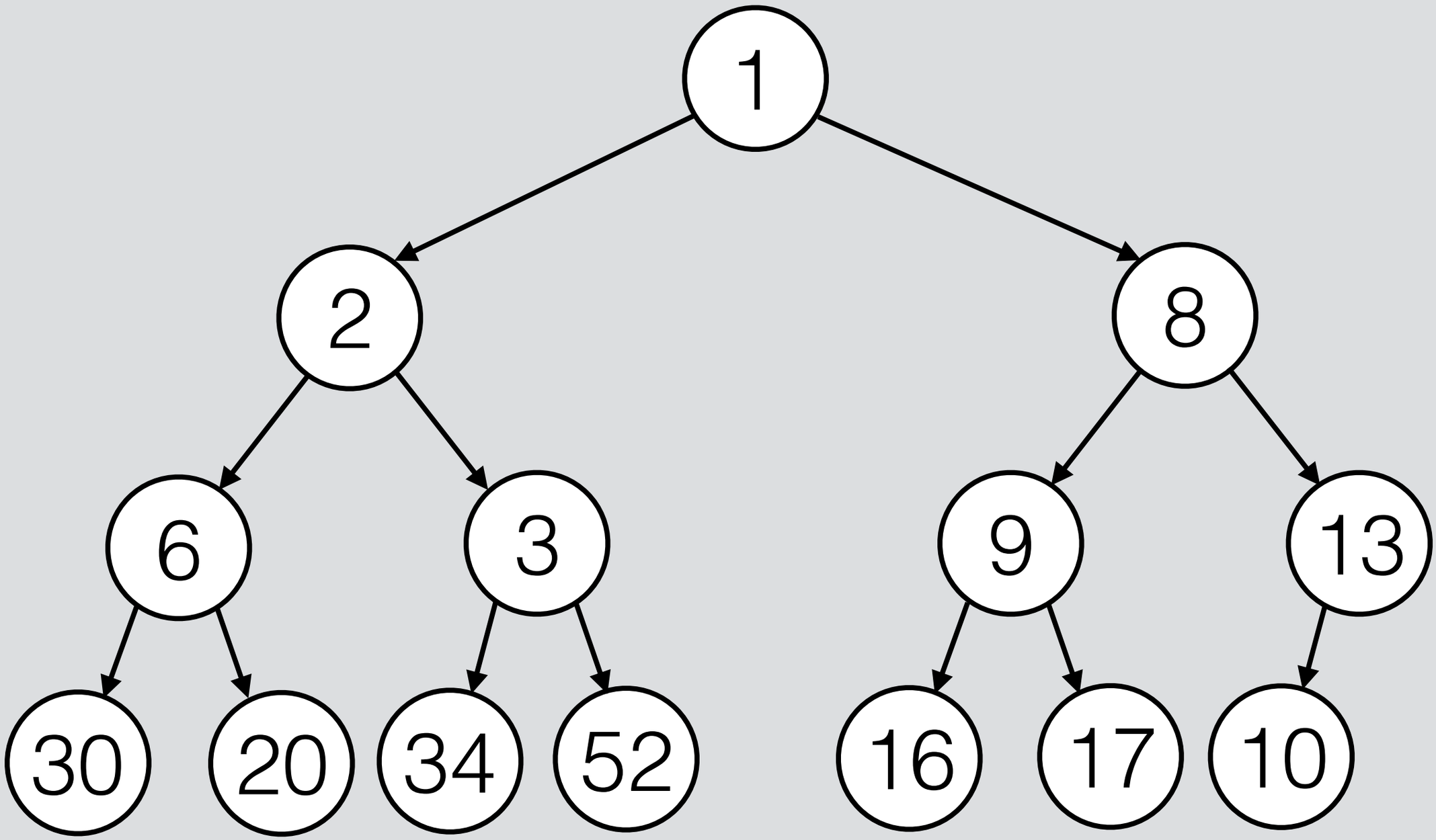
1. Replace root.
2. *Bubble* down.

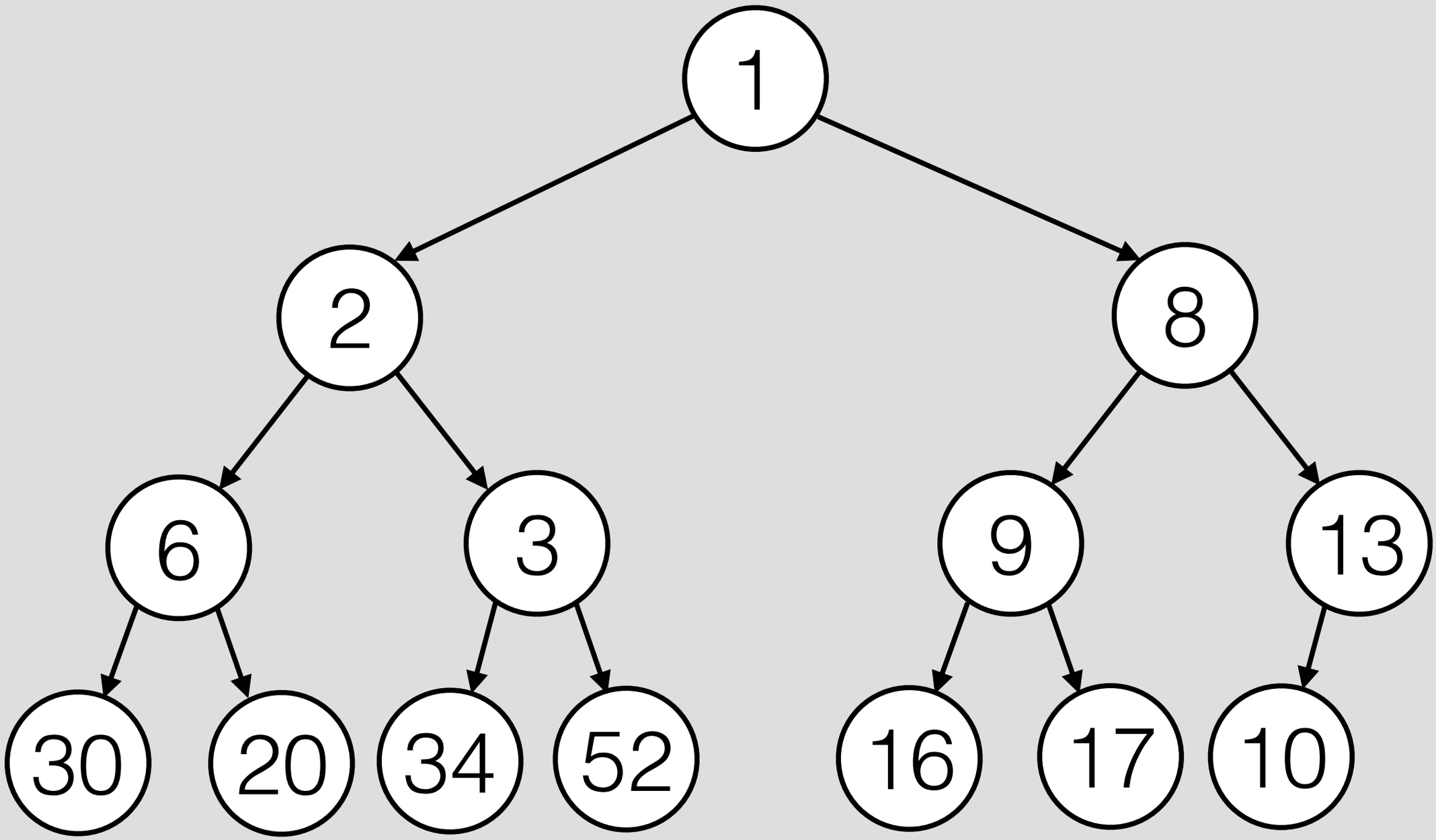
`pop();`

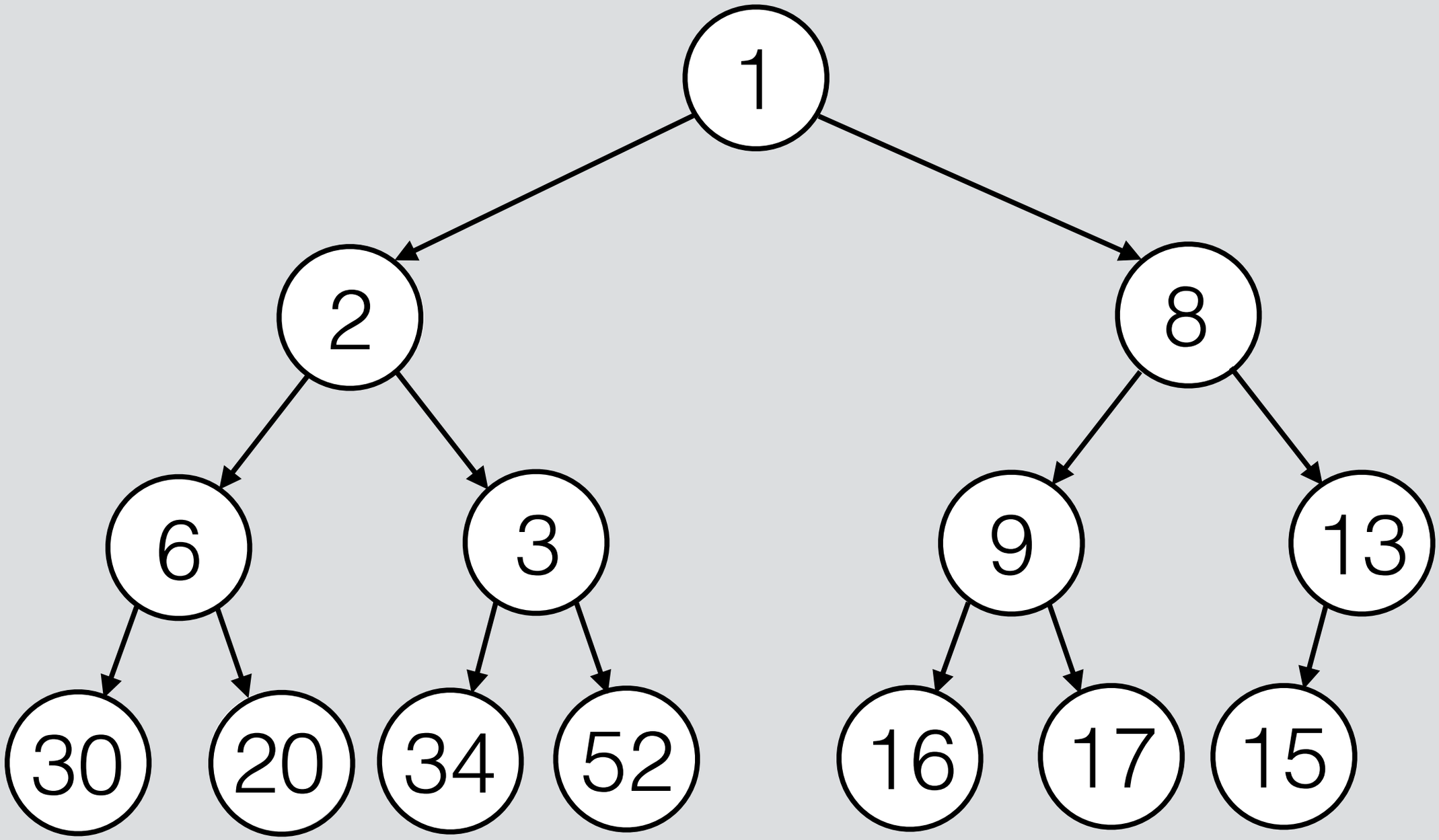


1. Replace root.
2. *Bubble* down.

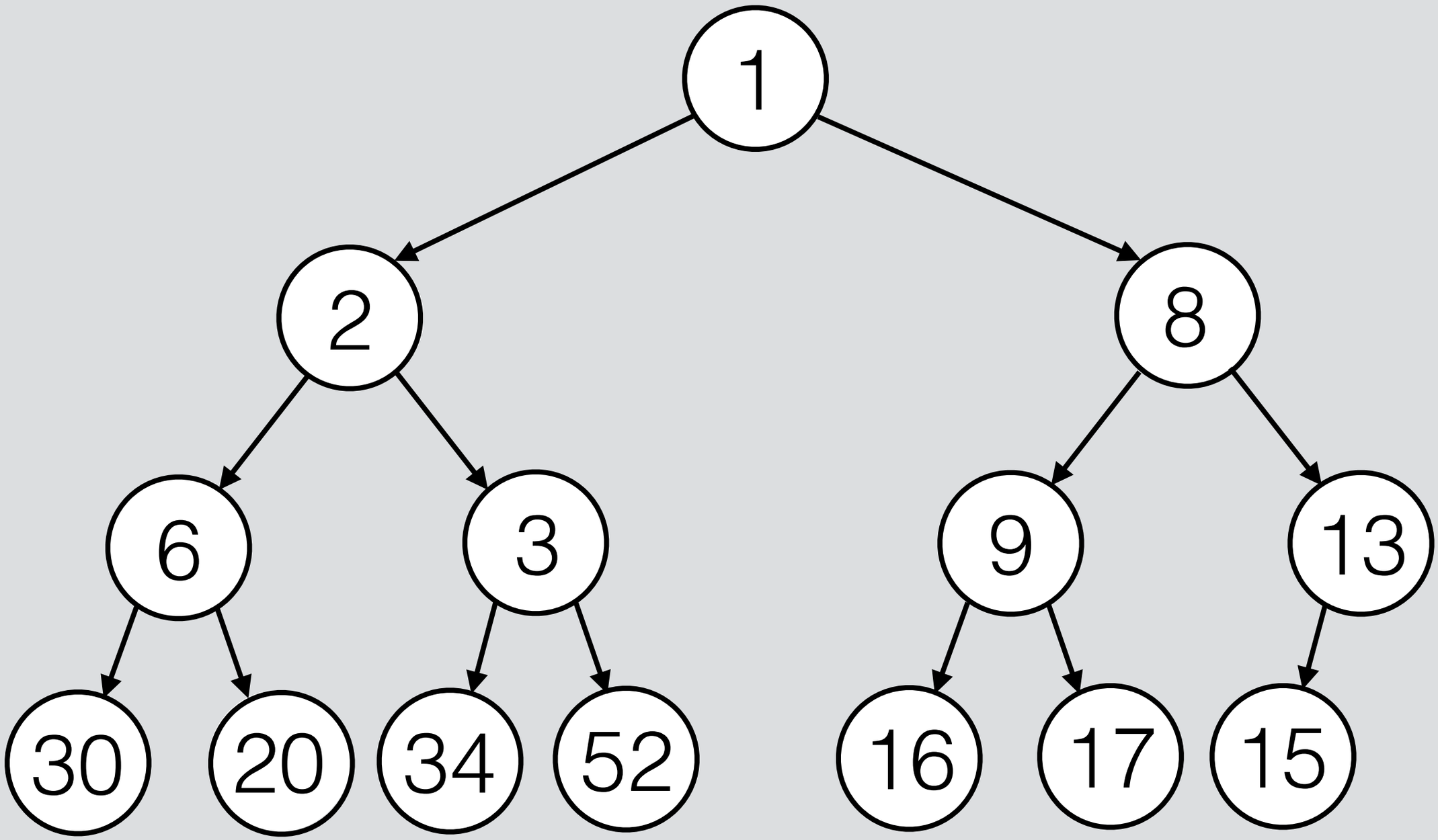
`pop();`




1
2
8
6
3
9
13
30
20
34
52
16
17
10

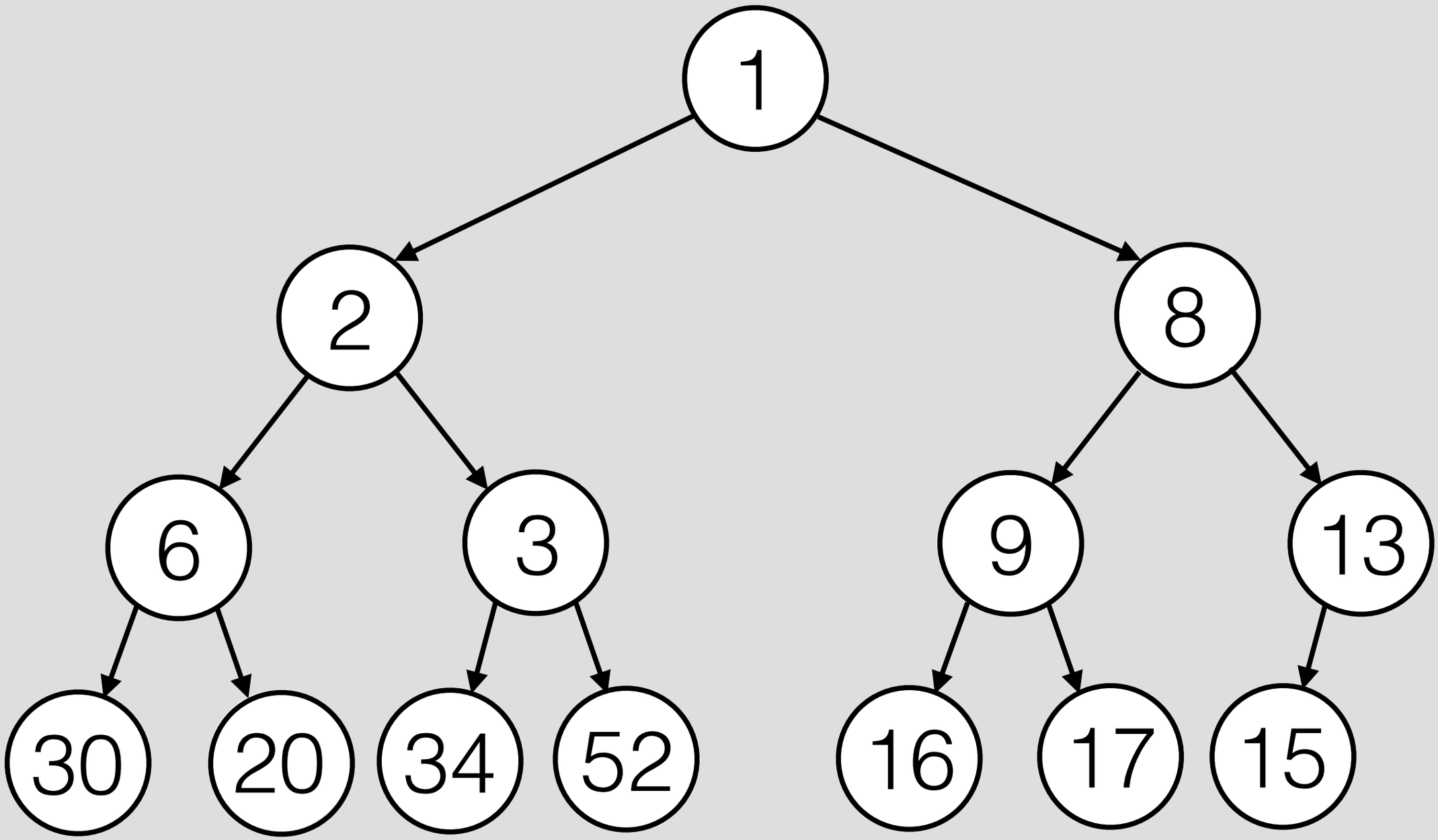

1
2
8
6
3
9
13
30
20
34
52
16
17
15

pop();



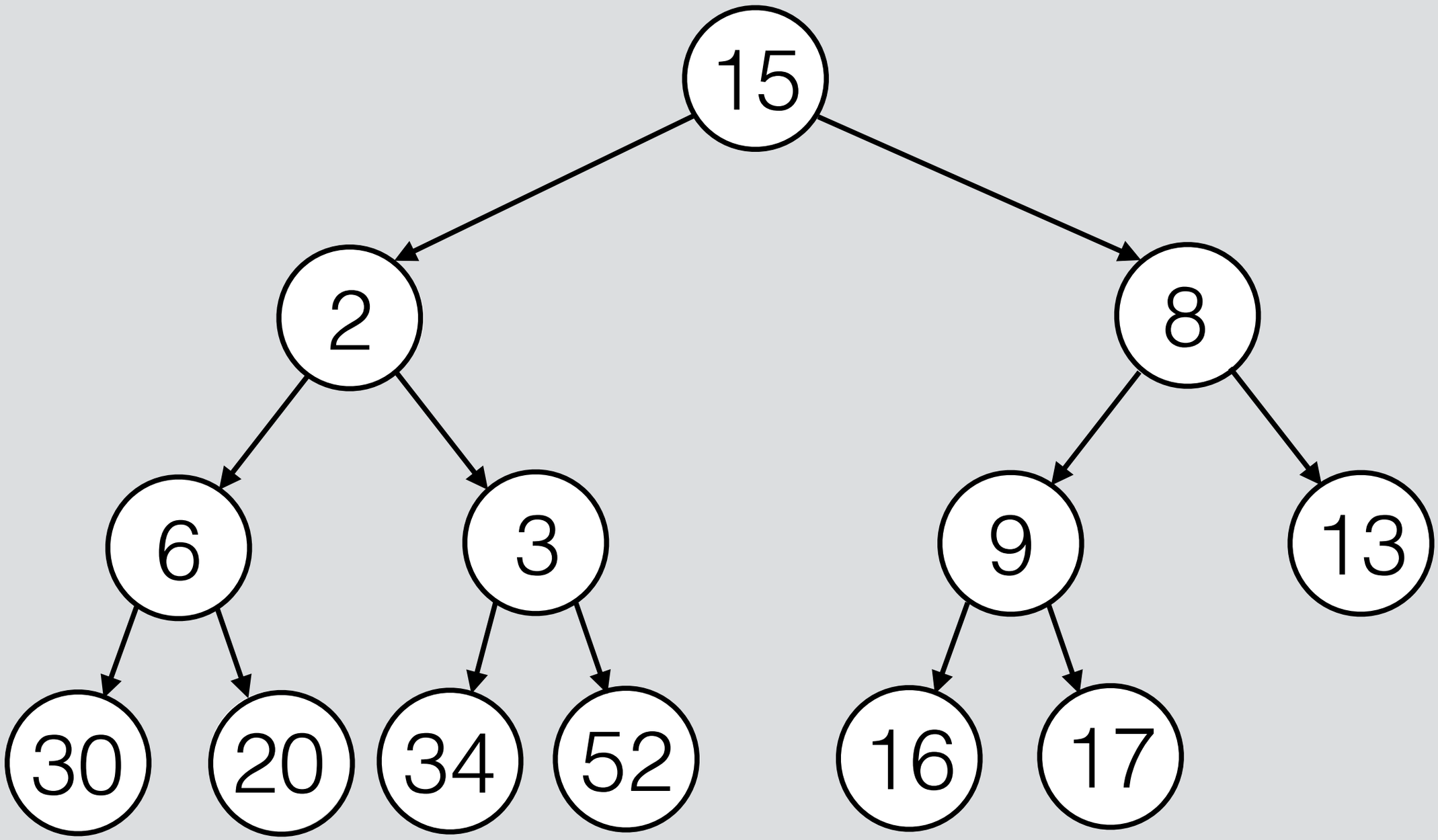
1. Replace root.

`pop();`



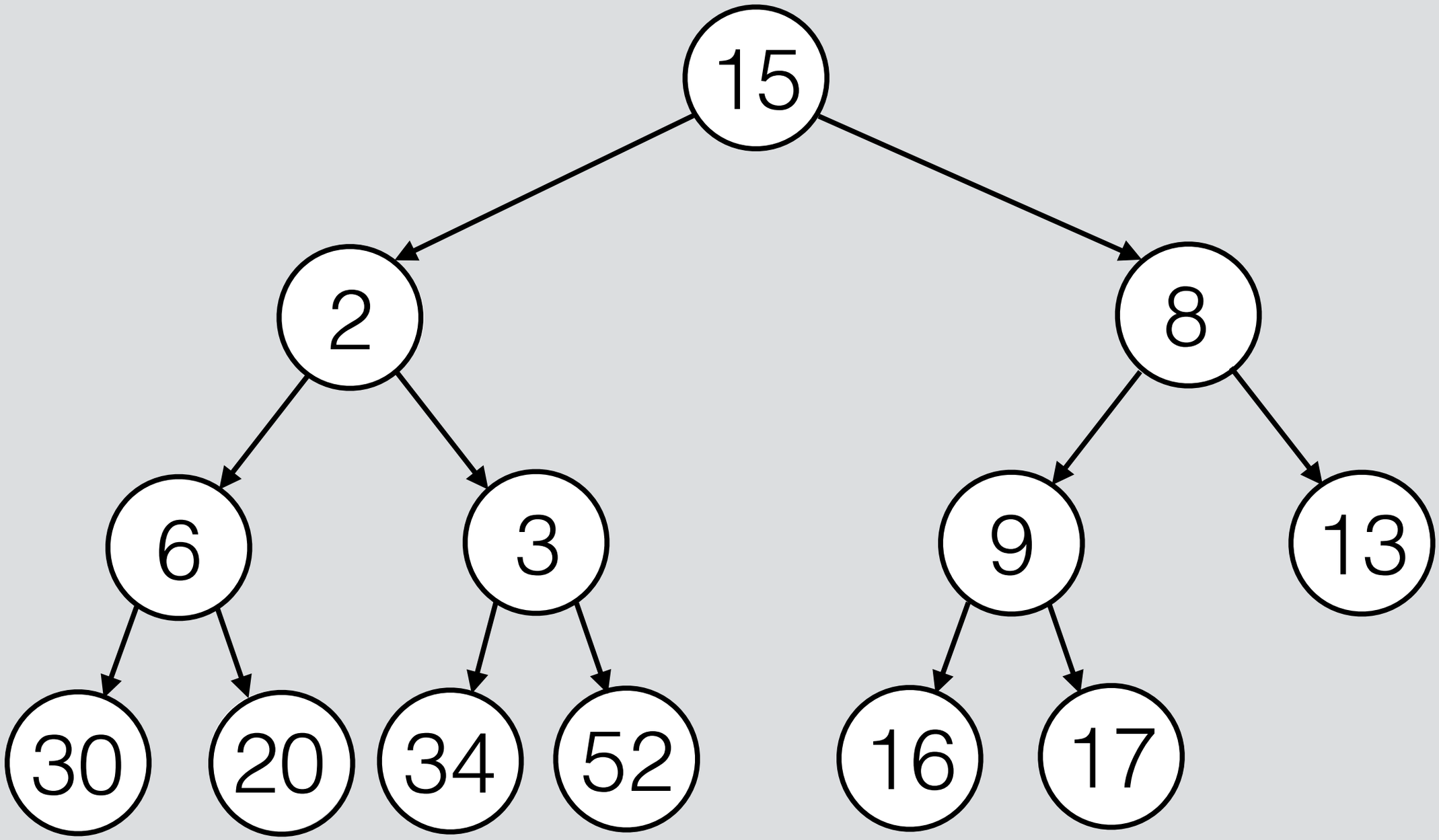
1. Replace root.

`pop();`



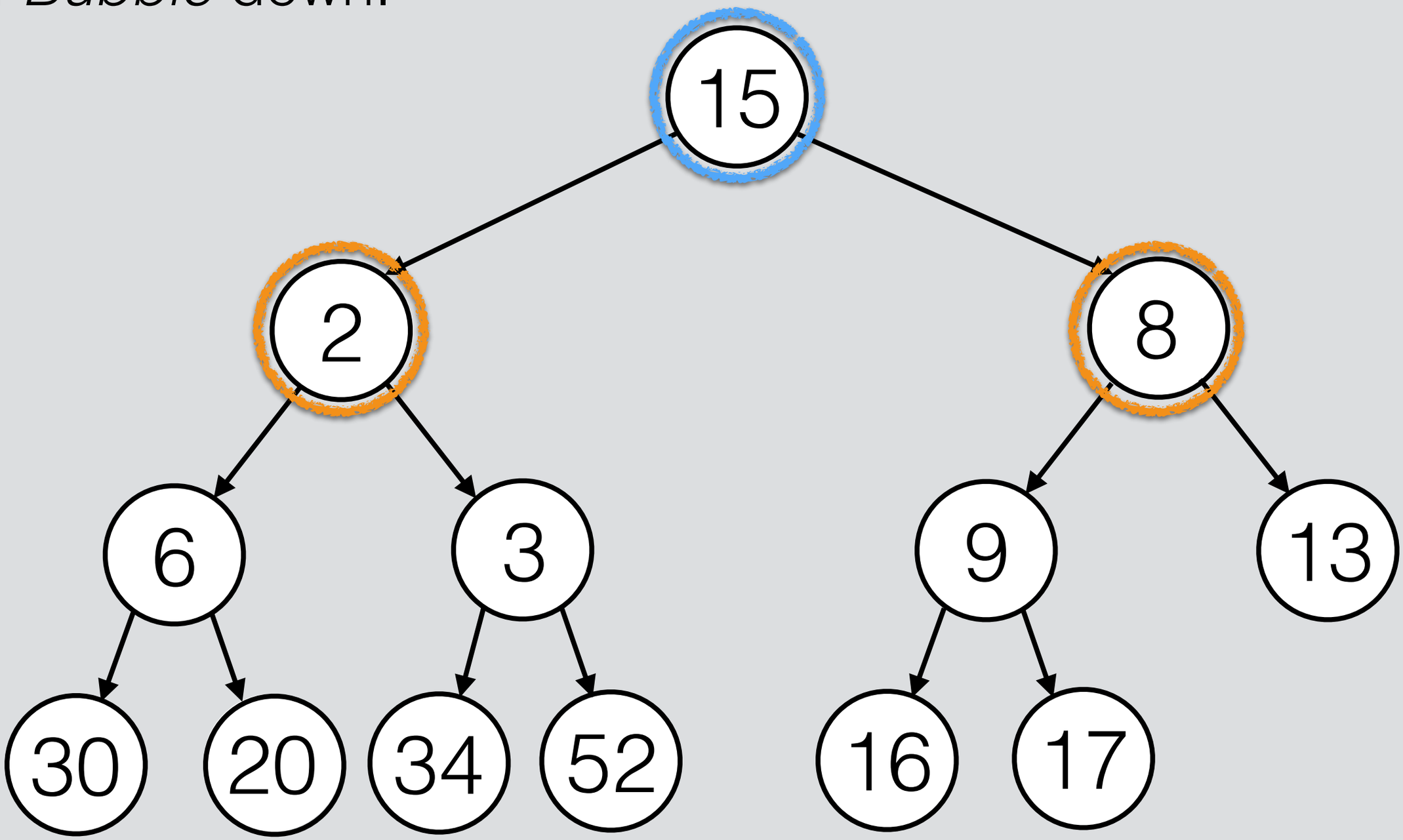
1. Replace root.
2. *Bubble* down.

`pop();`



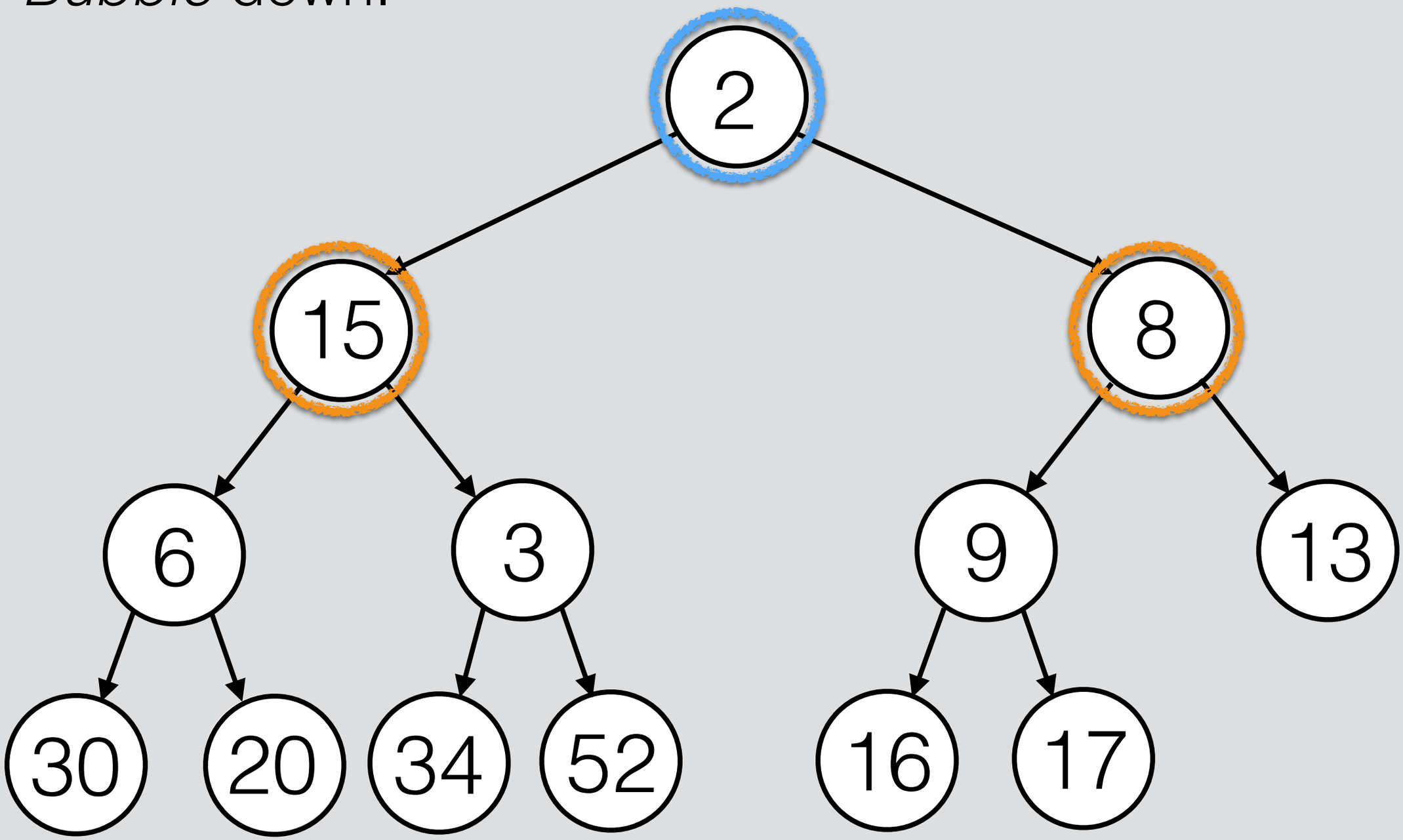
1. Replace root.
2. *Bubble* down.

`pop();`



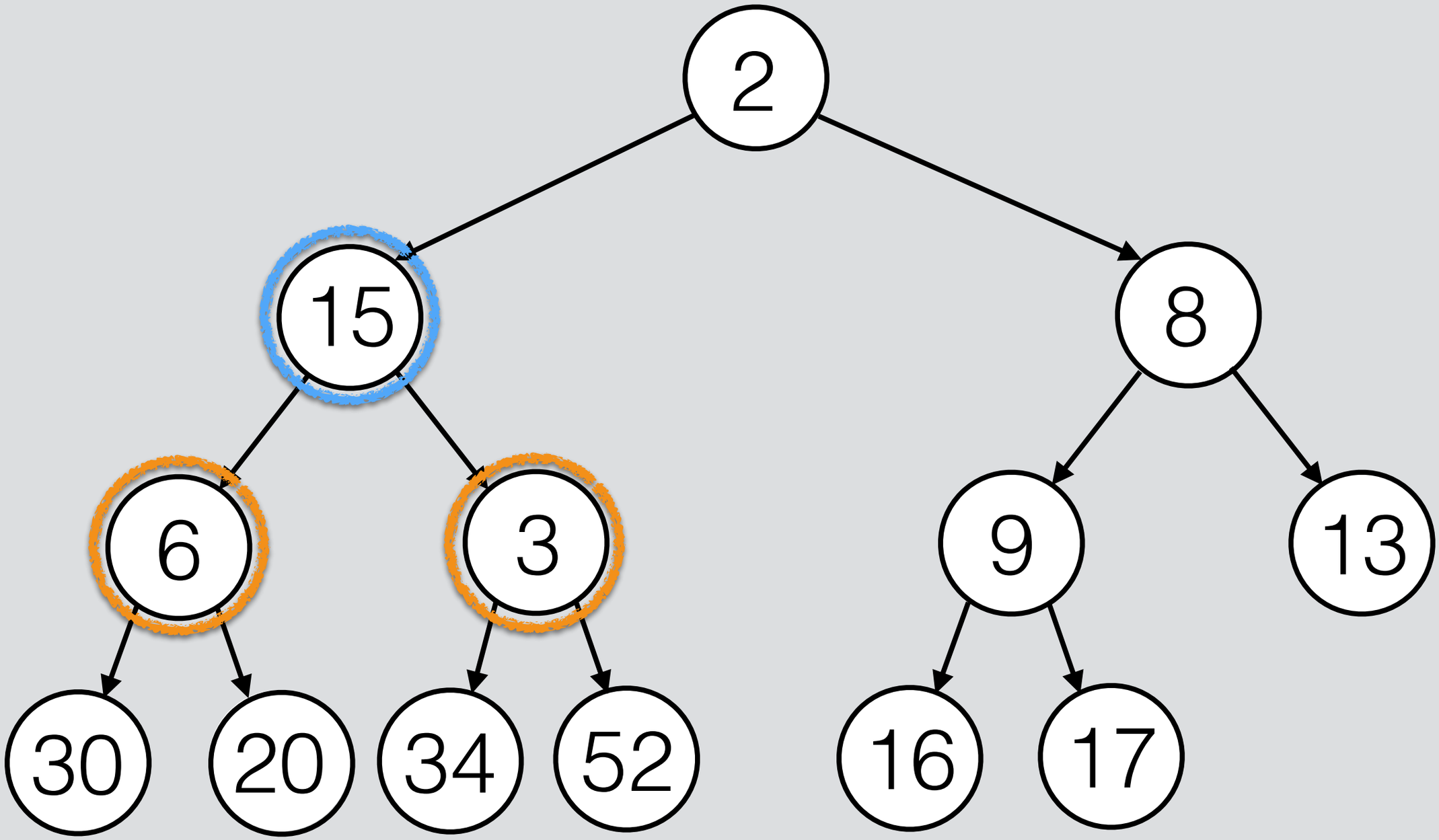
1. Replace root.
2. *Bubble* down.

`pop();`



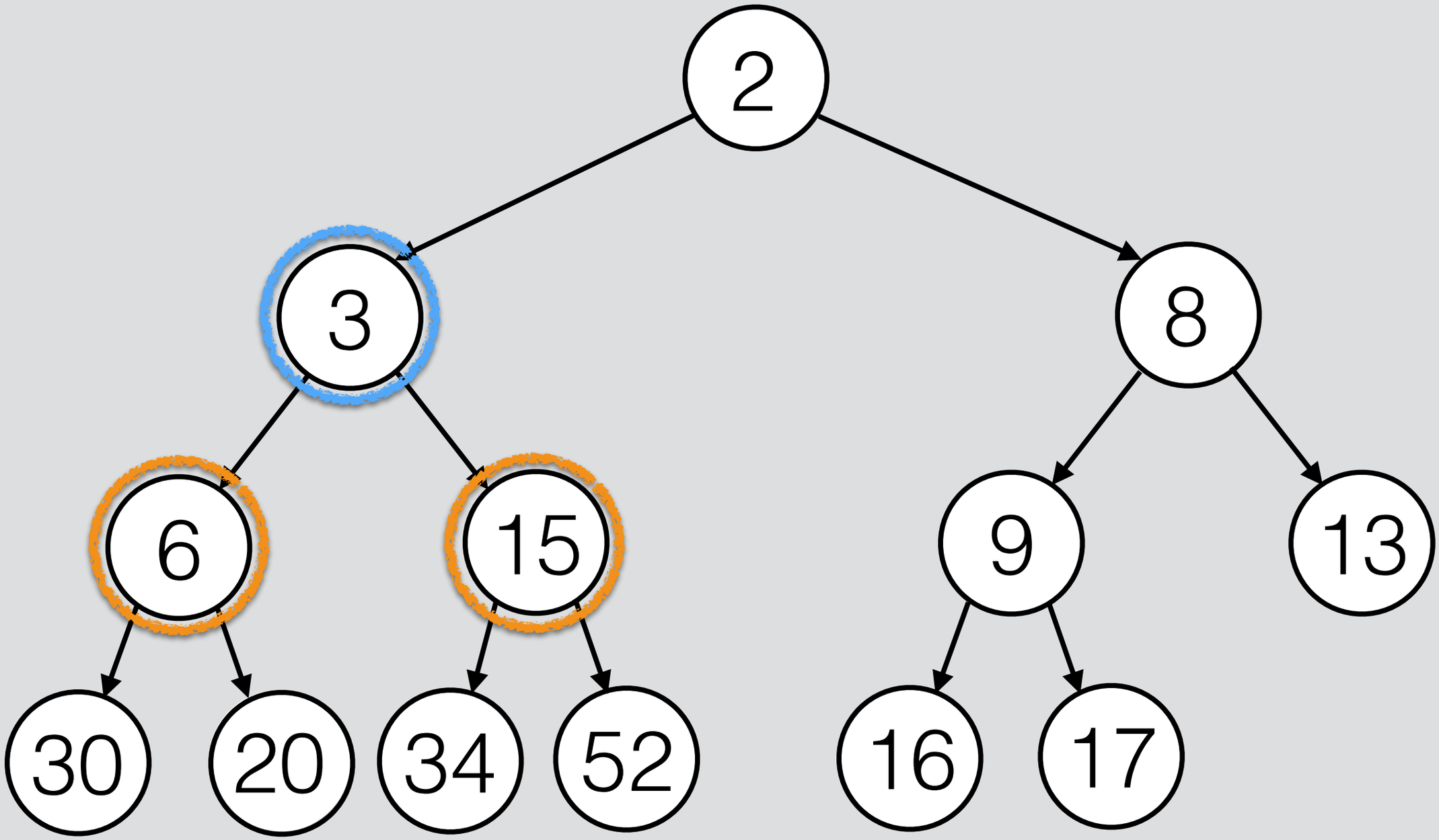
1. Replace root.
2. *Bubble* down.

`pop();`



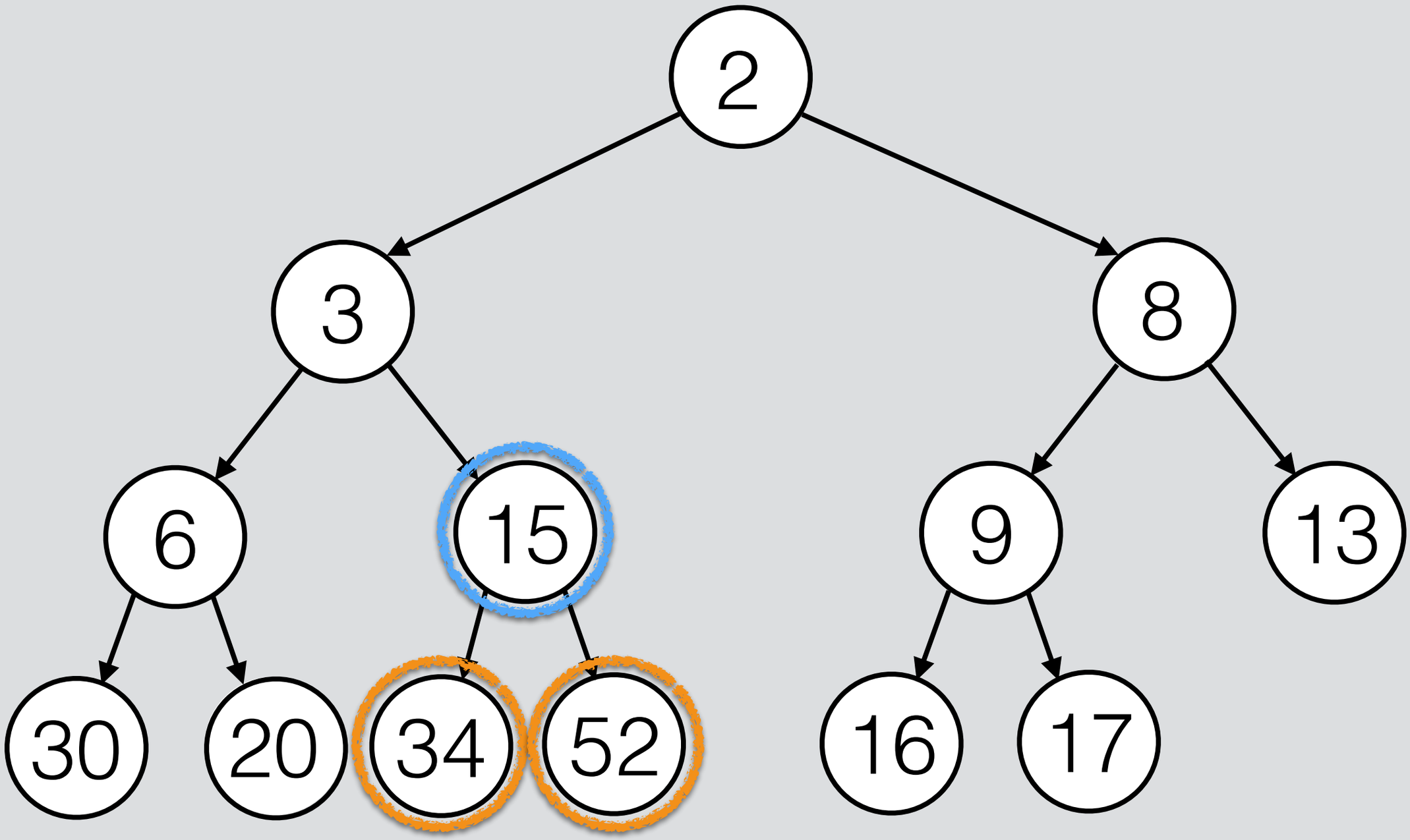
1. Replace root.
2. *Bubble* down.

`pop();`



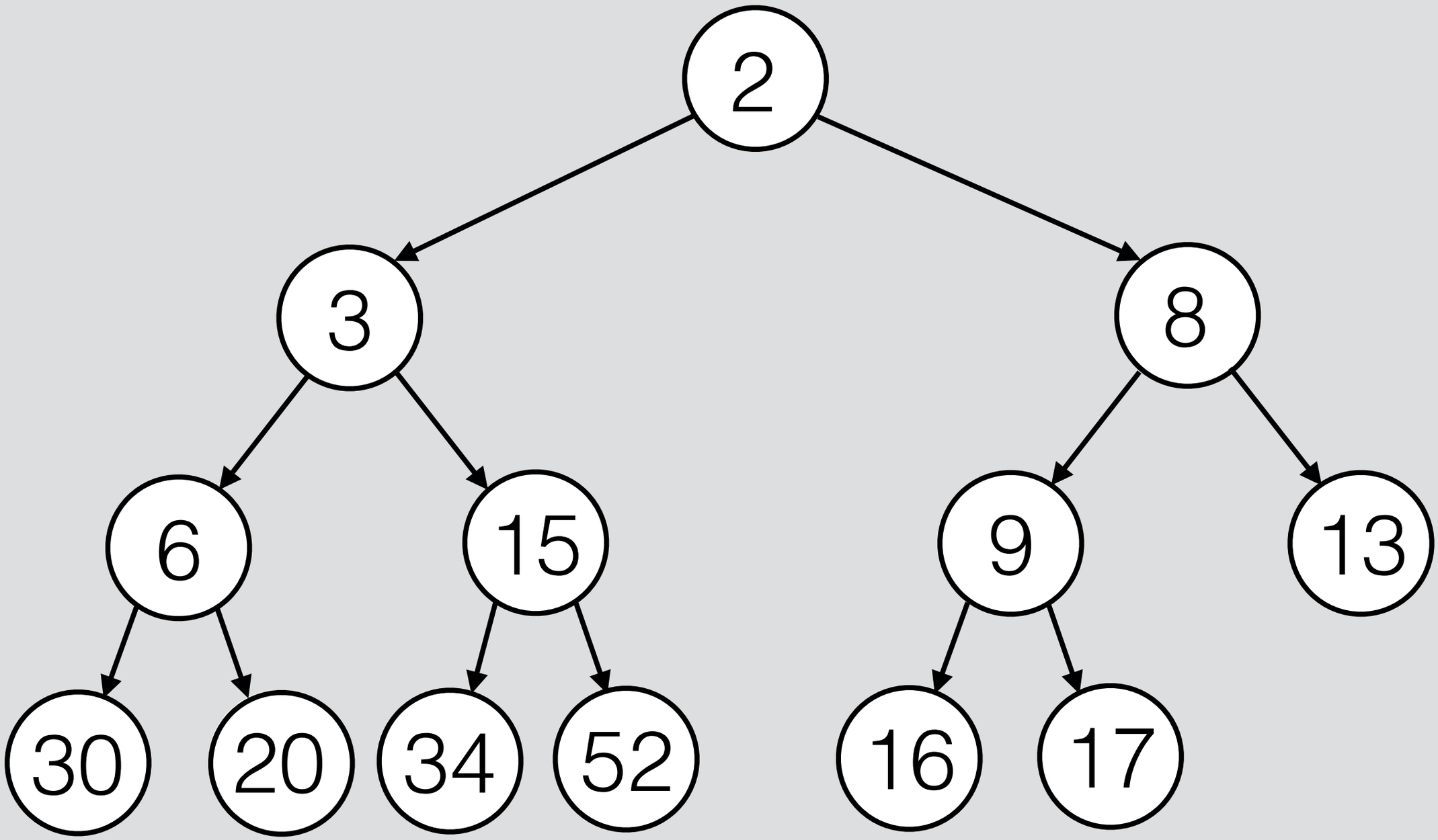
1. Replace root.
2. *Bubble* down.

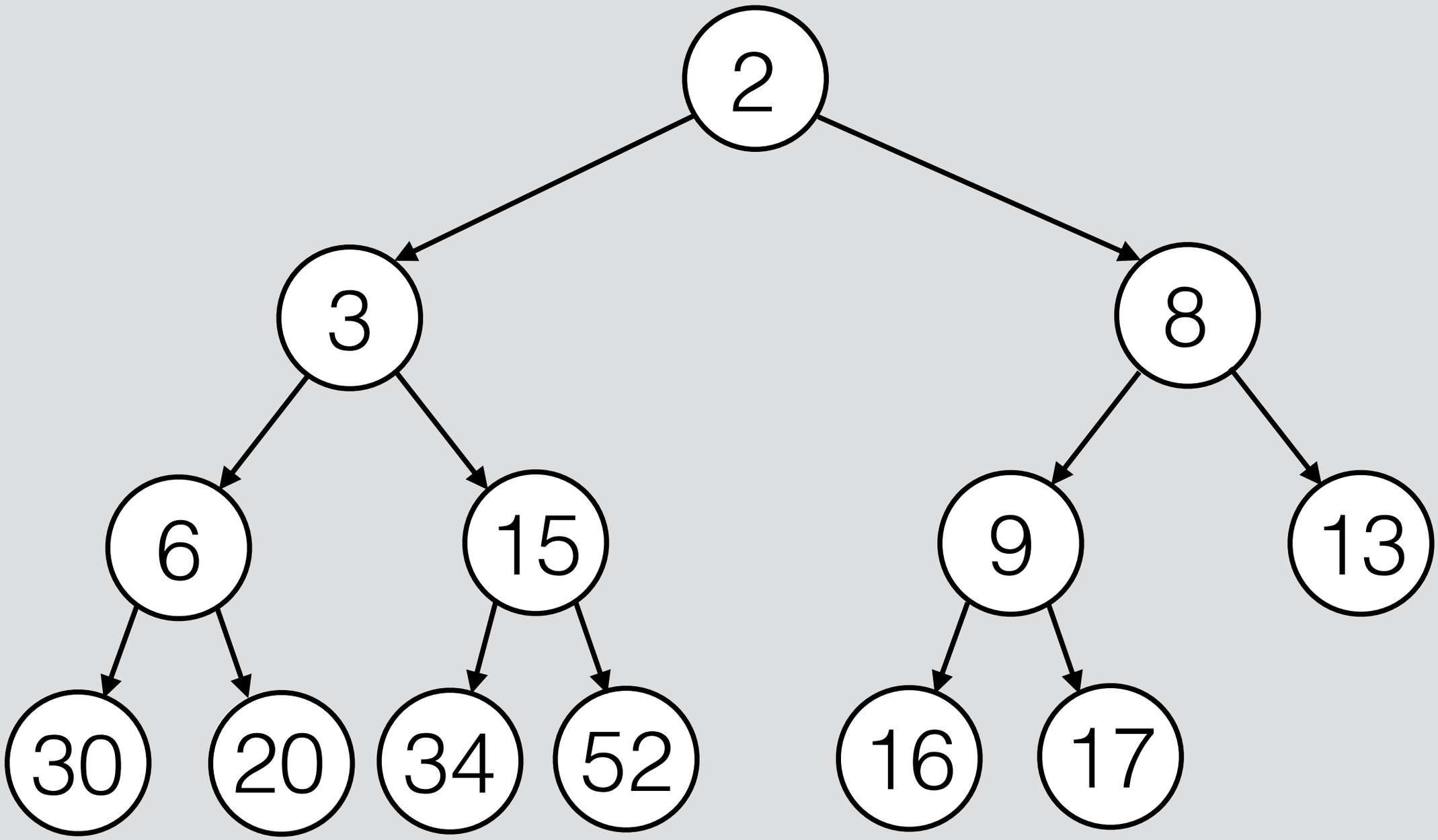
`pop();`



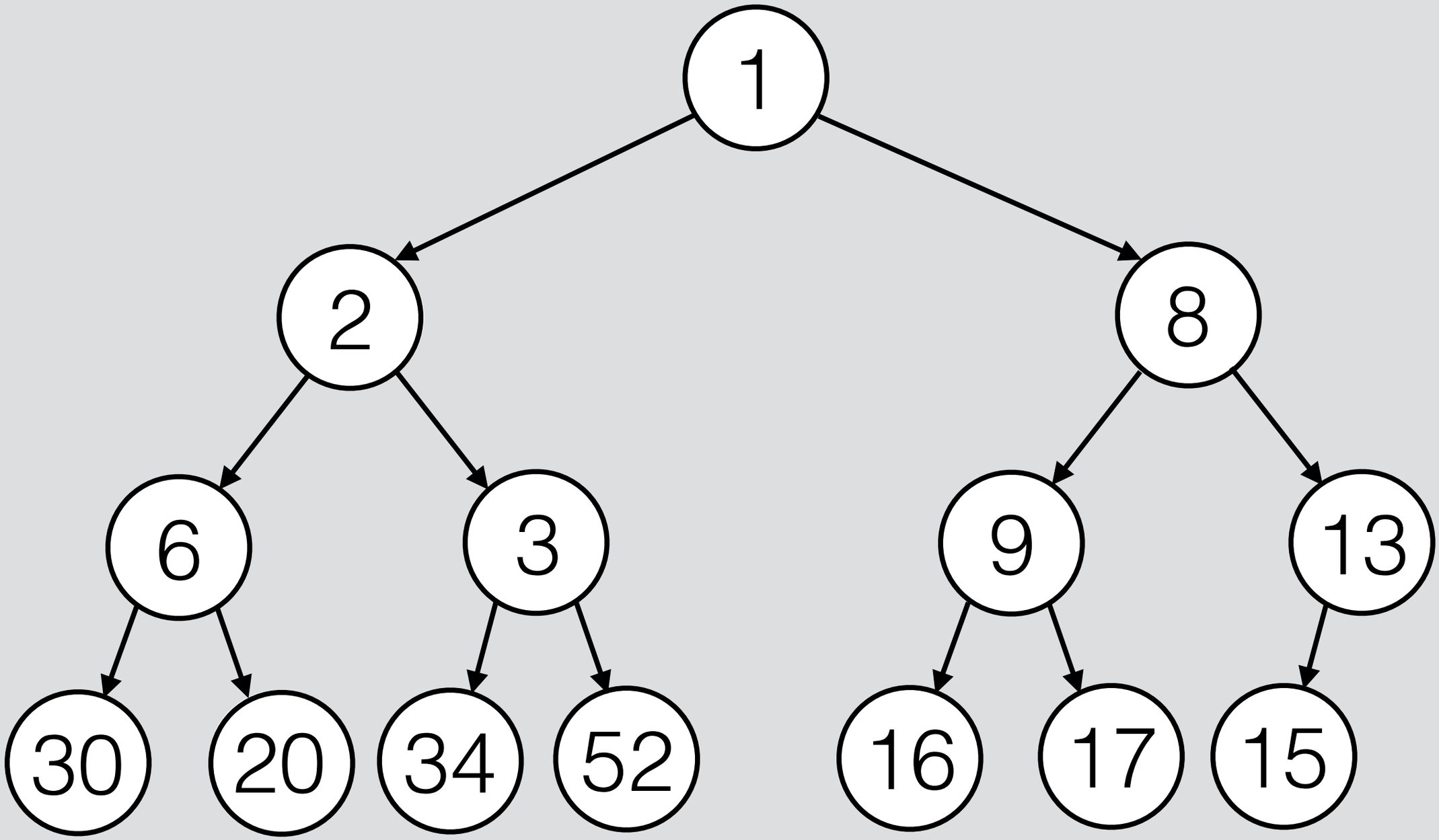
1. Replace root.
2. *Bubble* down.

`pop();`

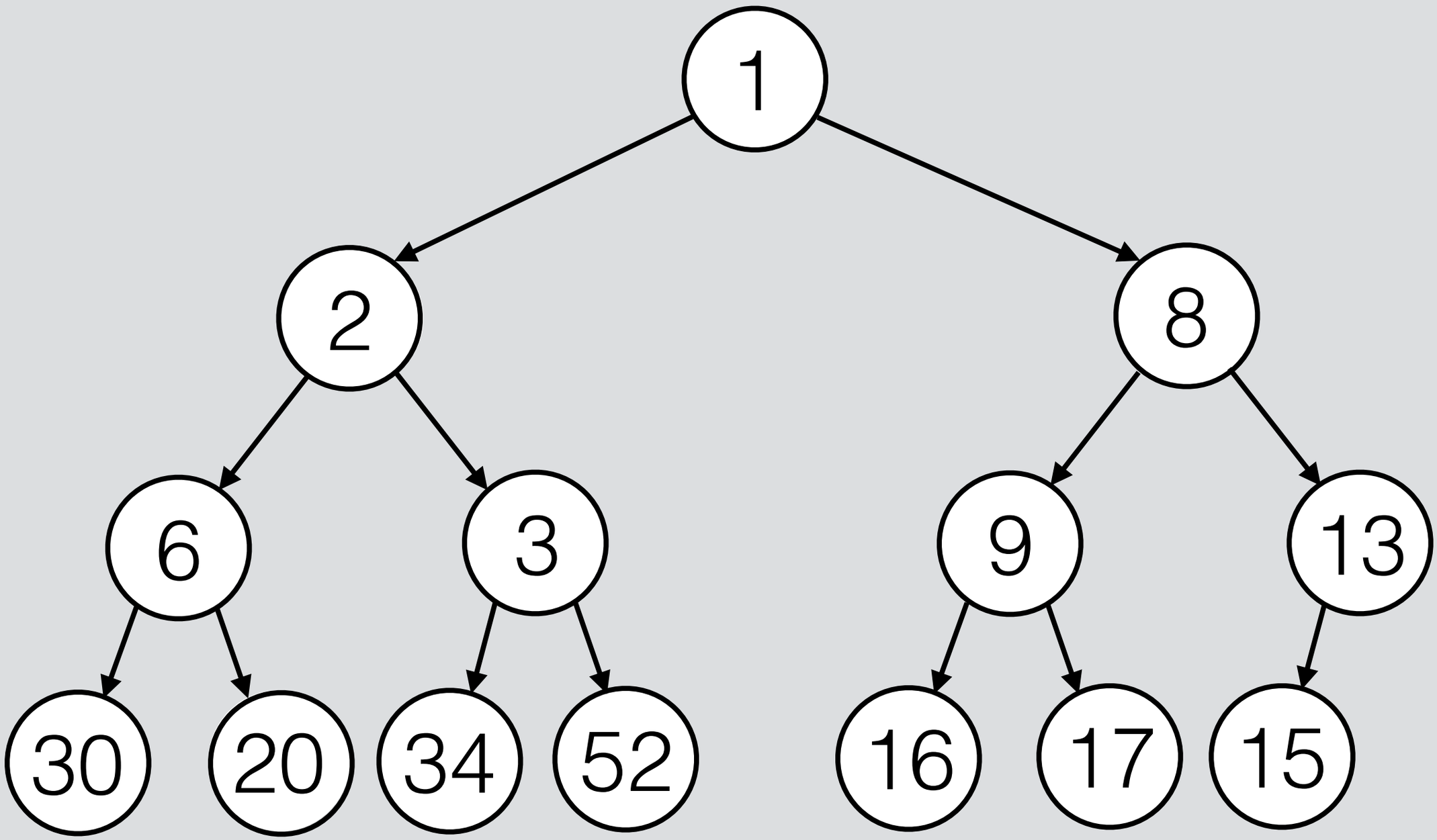



2
3
8
6
15
9
13
30
20
34
52
16
17

push()

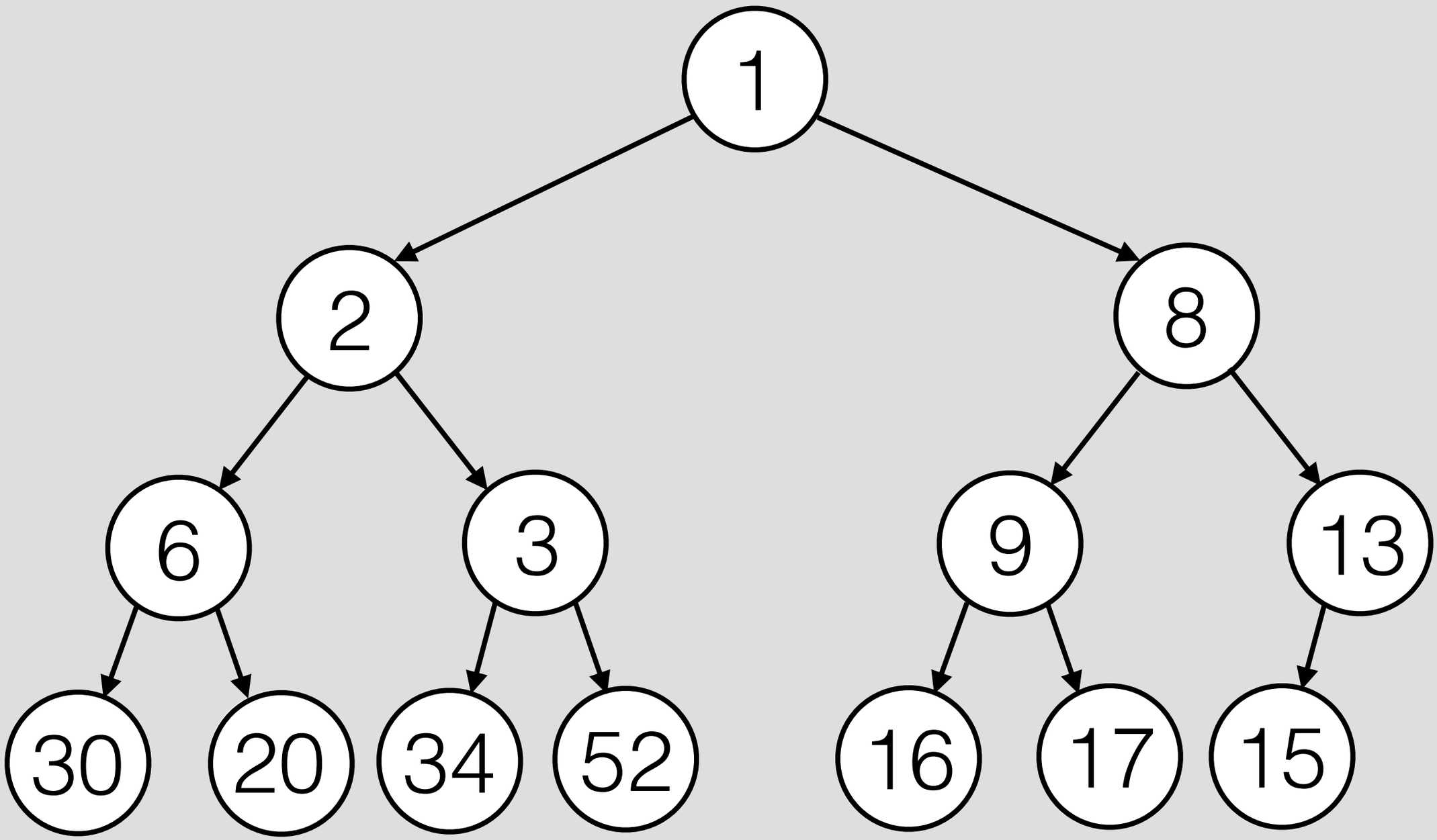

1
2
8
6
3
9
13
30
20
34
52
16
17
15

```
push(4, 4);
```



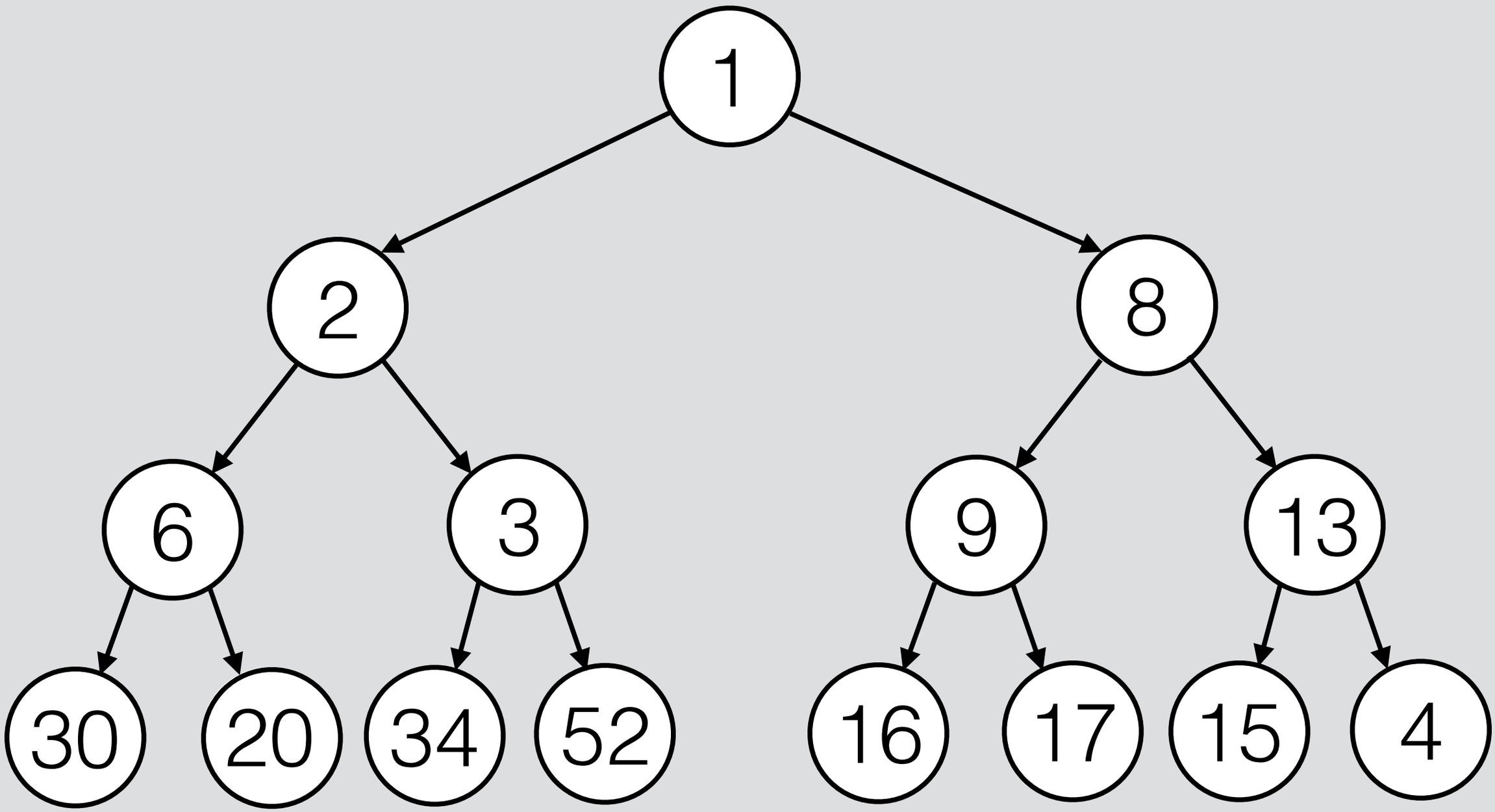
1. Insert at leaf.

```
push(4, 4);
```



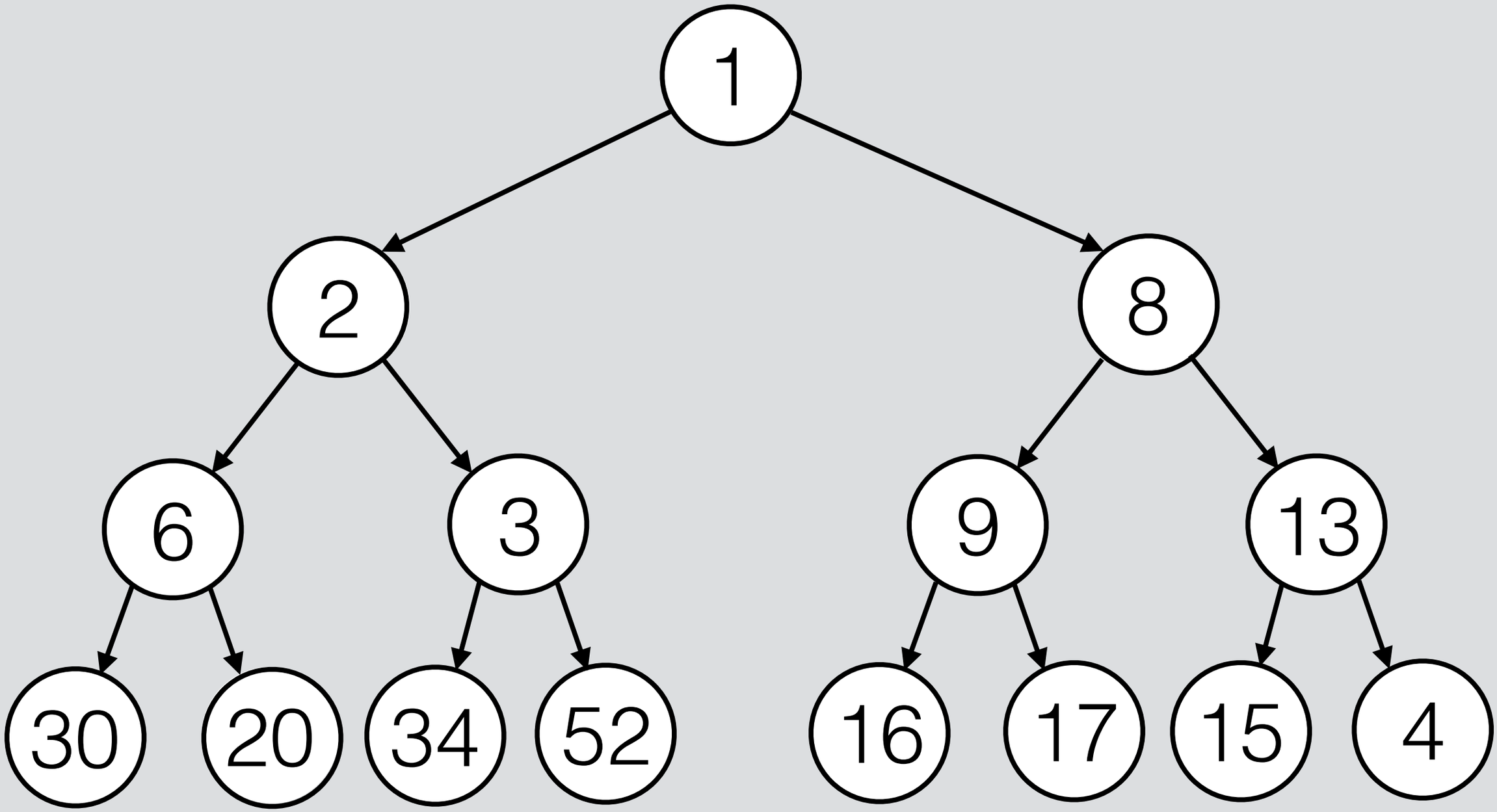
1. Insert at leaf.

```
push(4, 4);
```



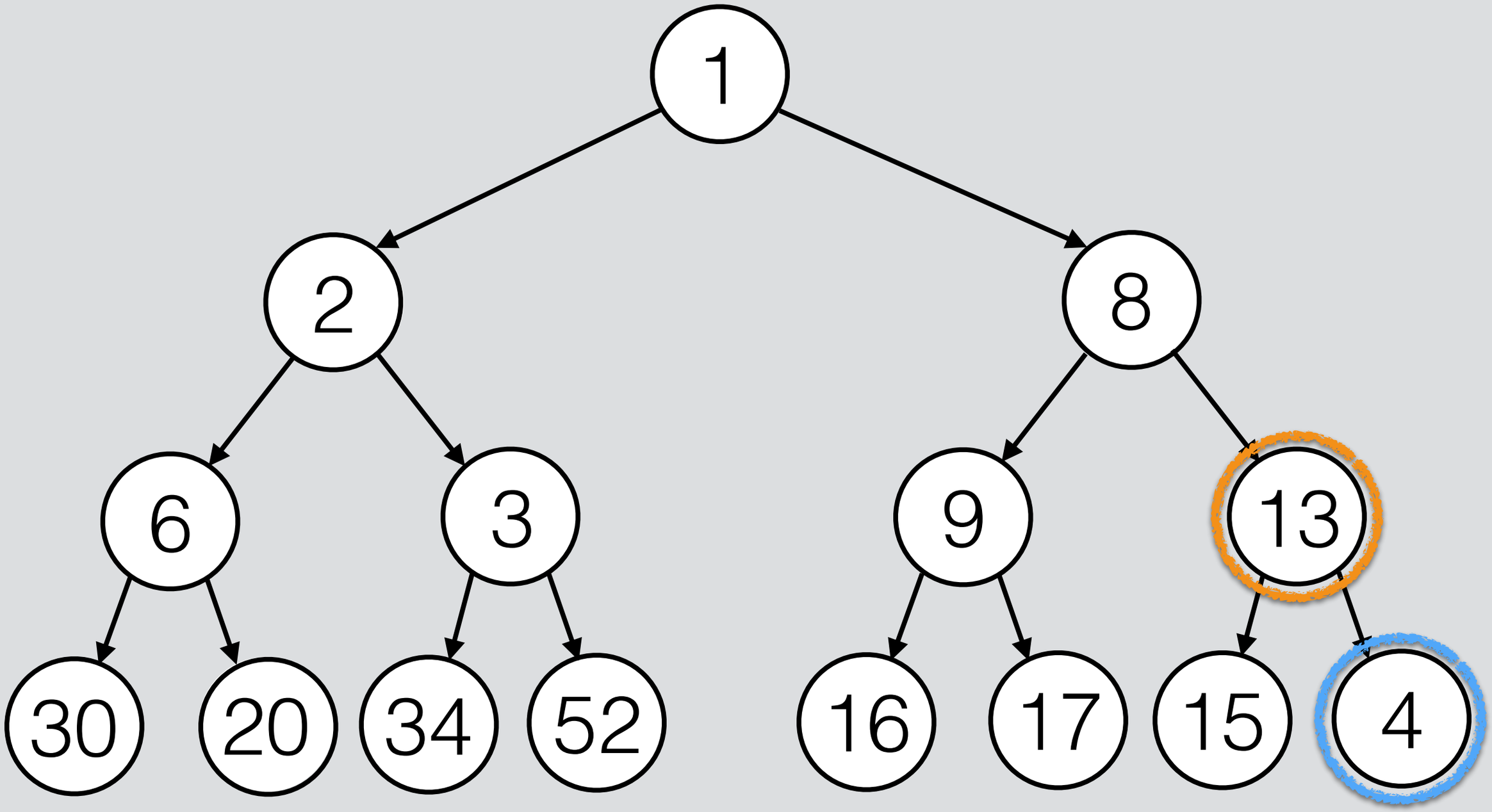
1. Insert at leaf.
2. *Bubble* up.

```
push(4, 4);
```



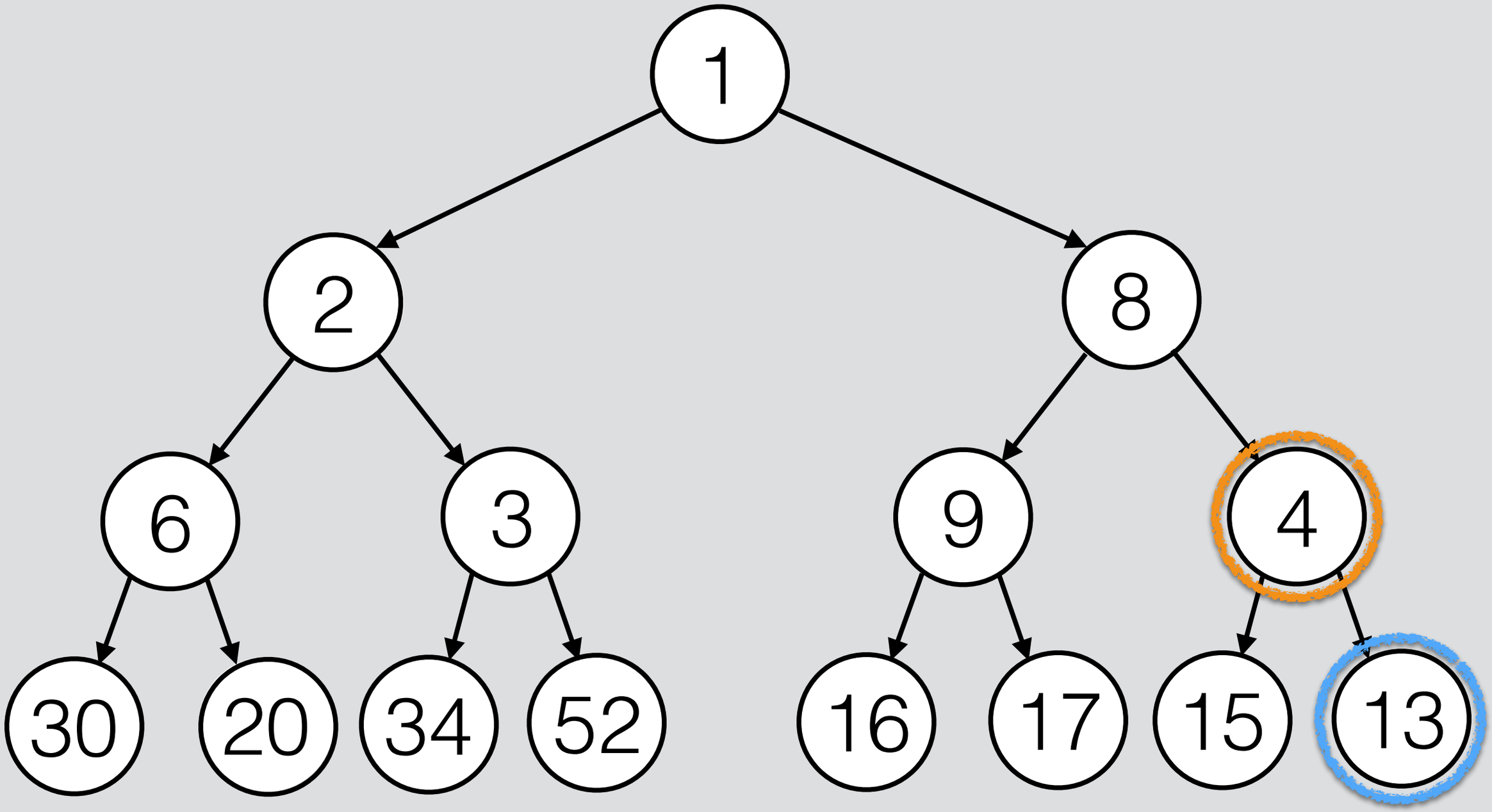
1. Insert at leaf.
2. *Bubble* up.

```
push(4, 4);
```



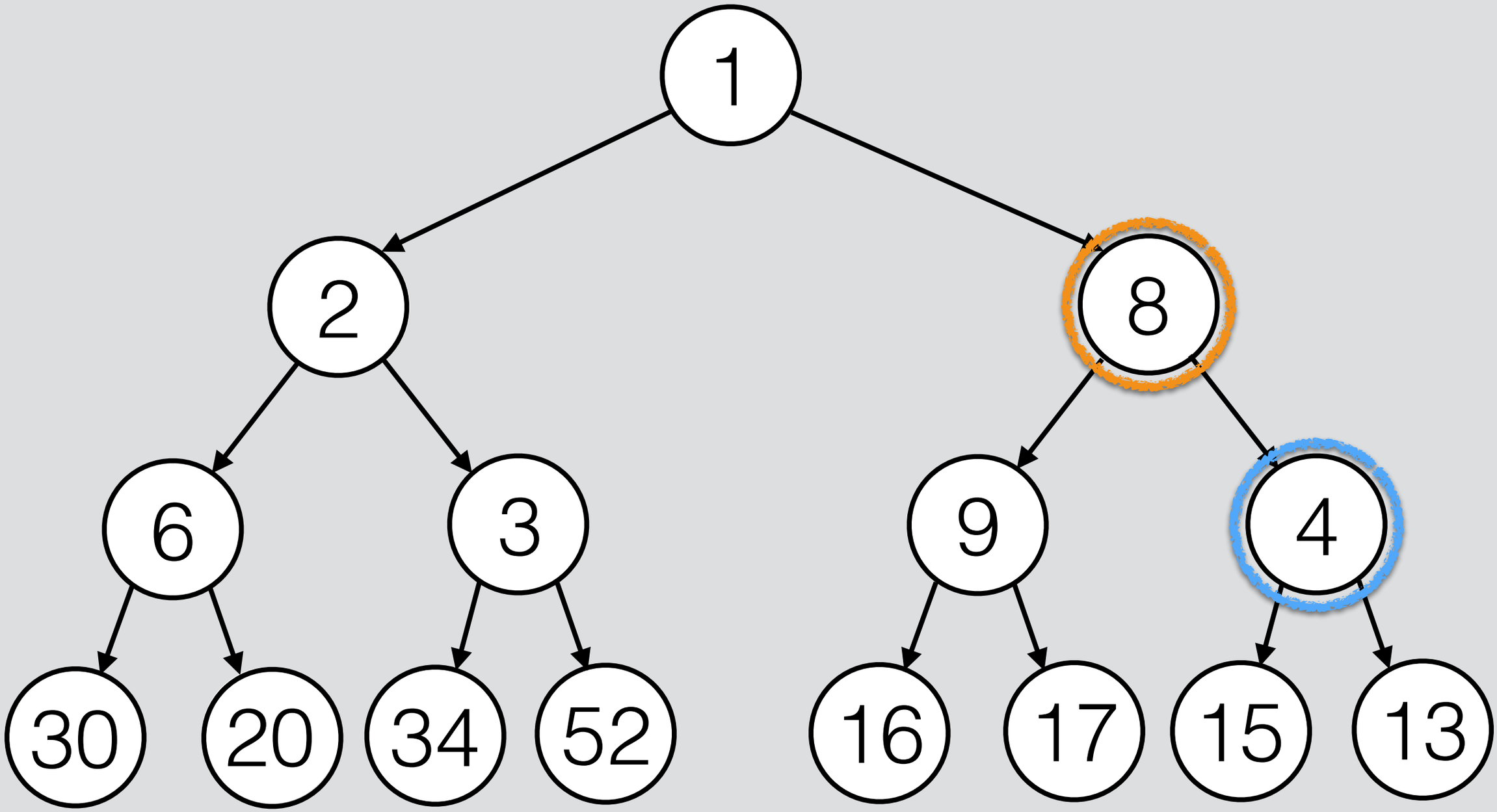
1. Insert at leaf.
2. *Bubble* up.

```
push(4, 4);
```



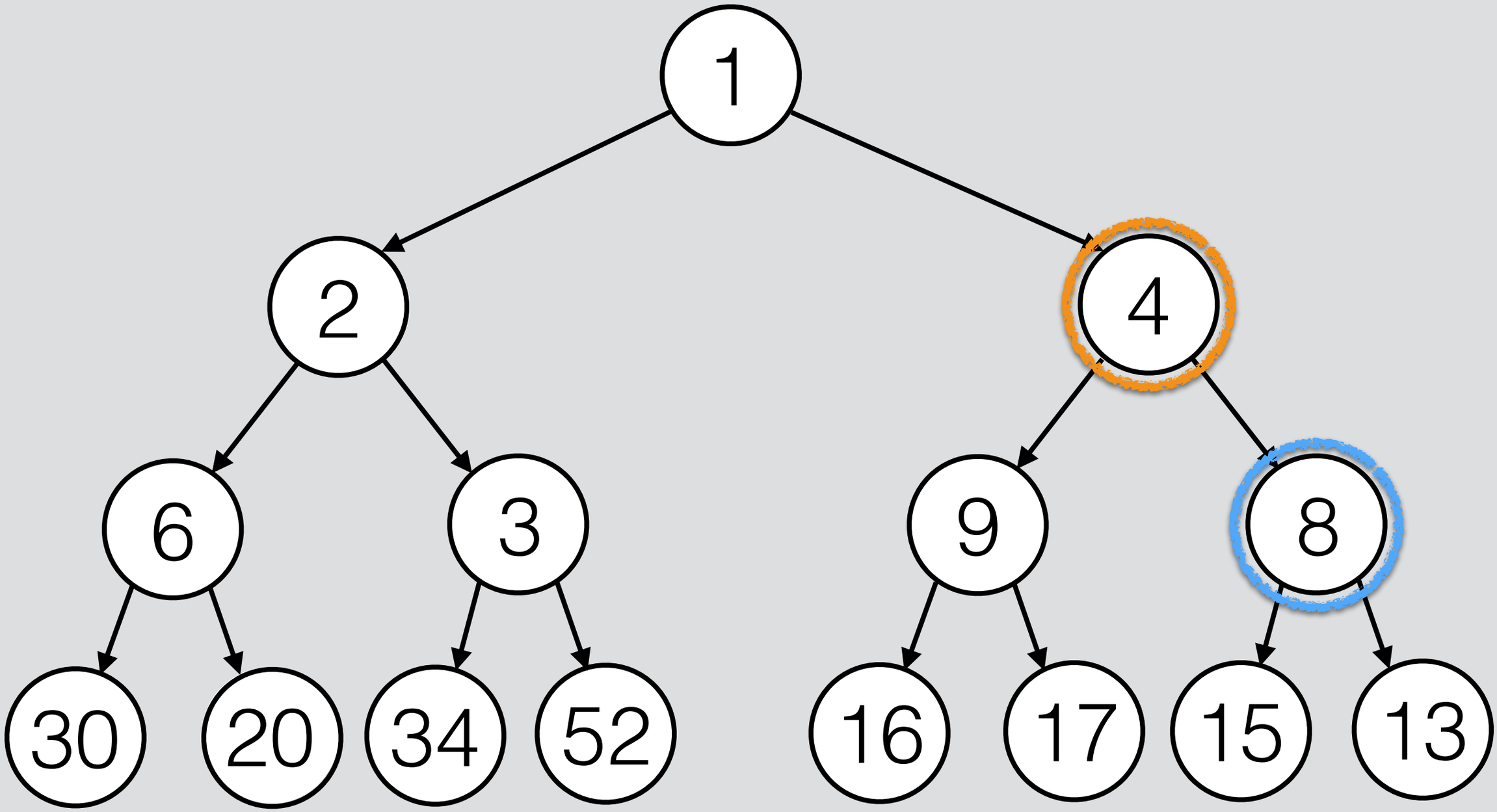
1. Insert at leaf.
2. *Bubble* up.

```
push(4, 4);
```



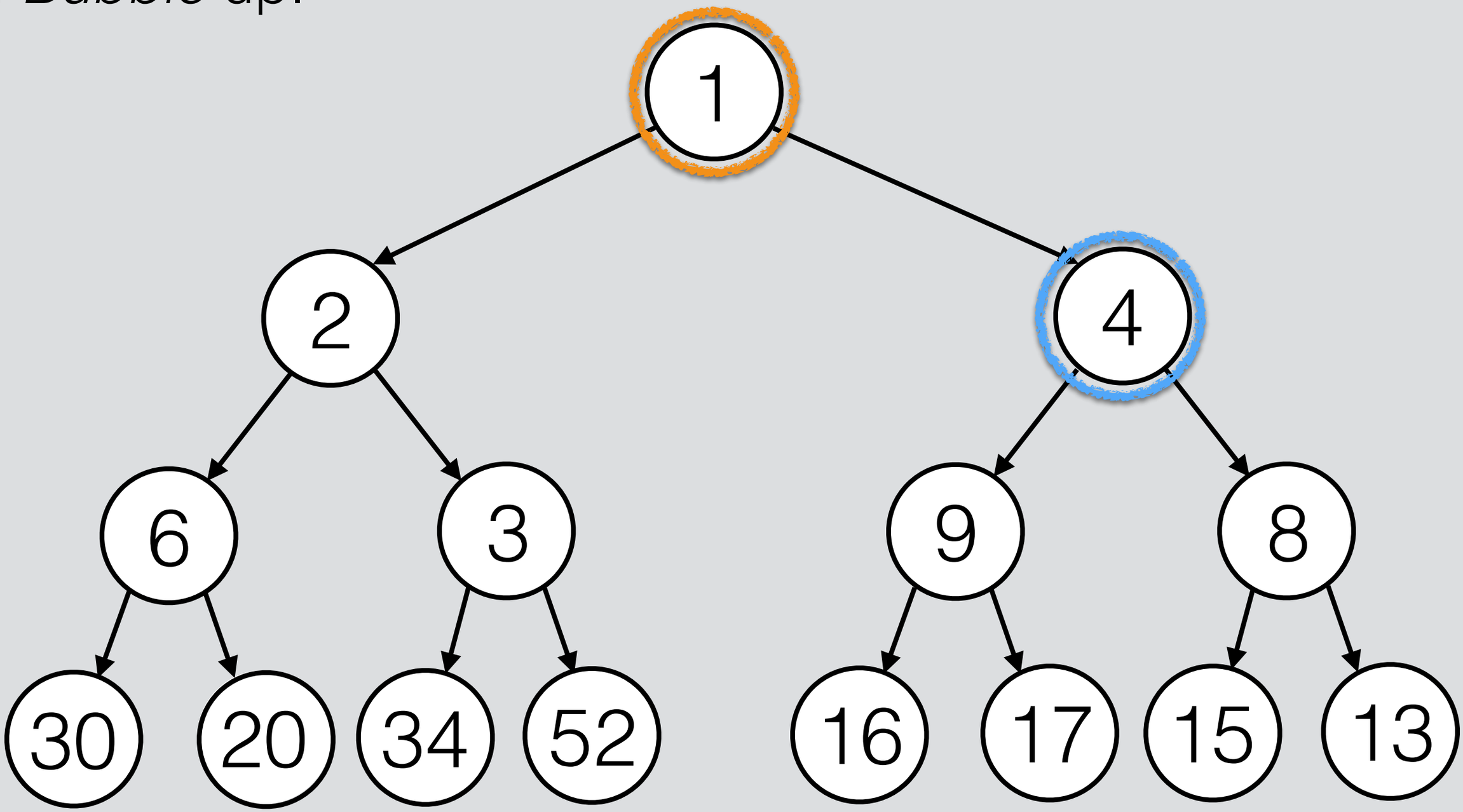
1. Insert at leaf.
2. *Bubble* up.

```
push(4, 4);
```



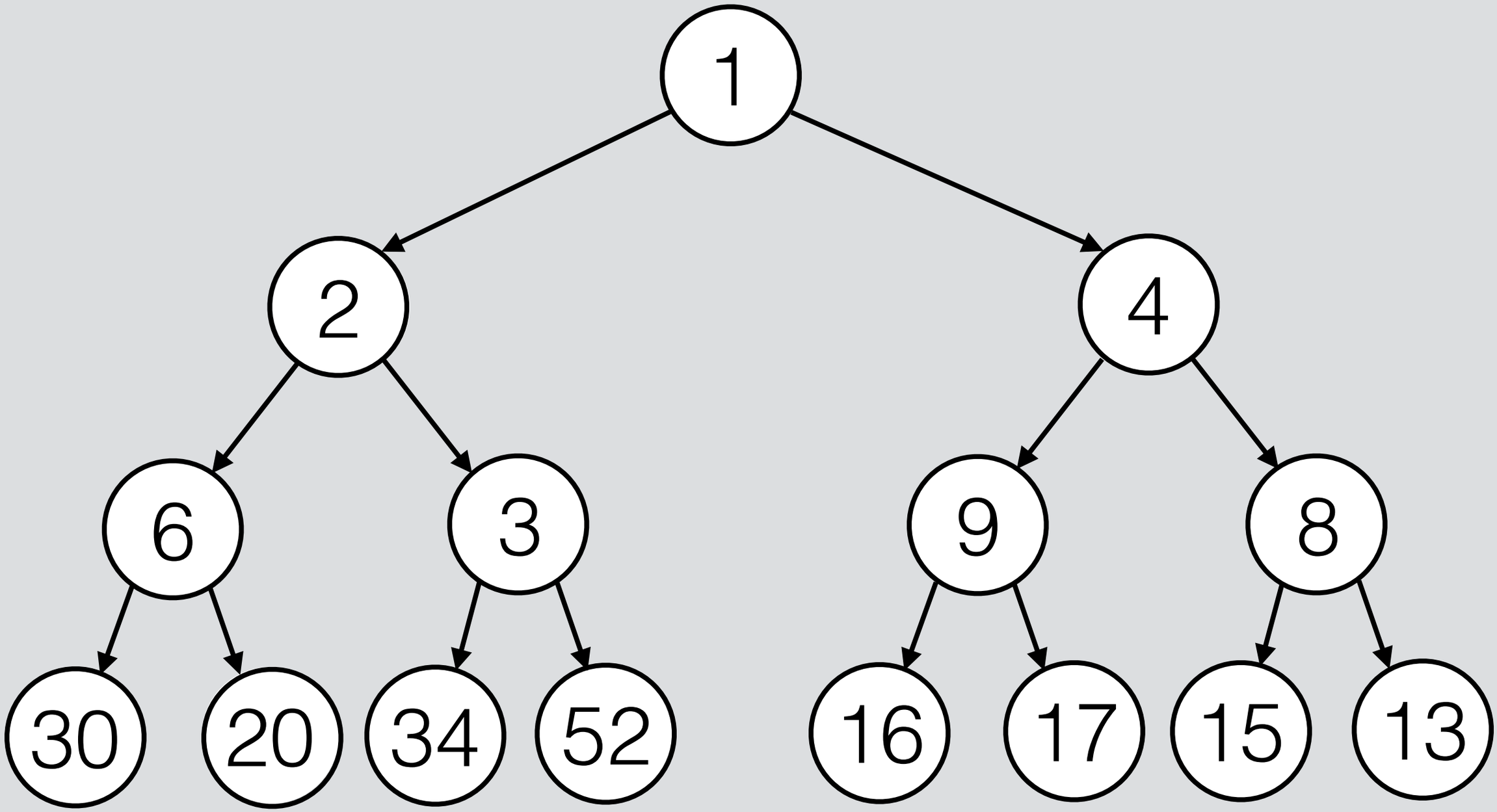
1. Insert at leaf.
2. *Bubble* up.

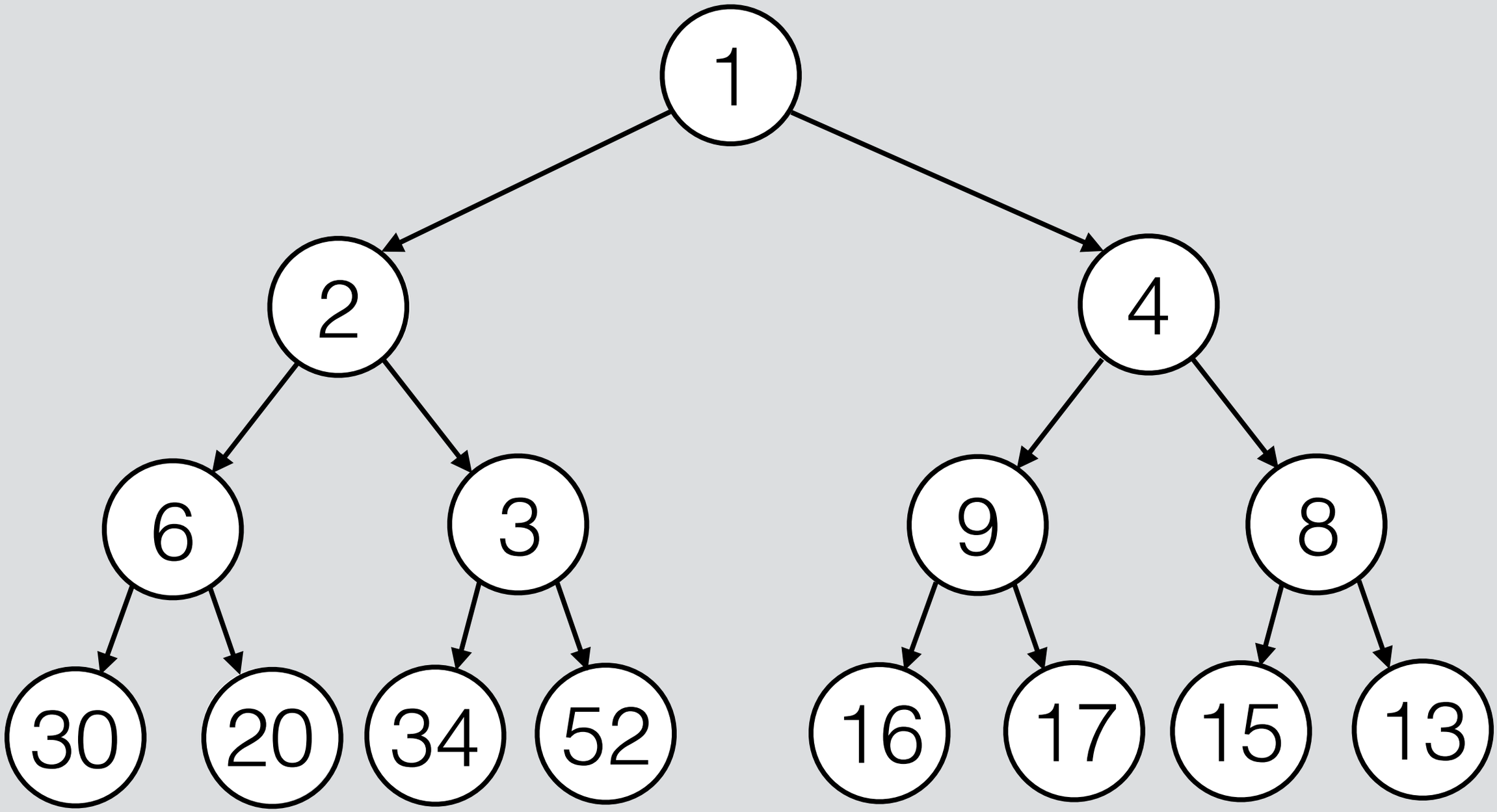
```
push(4, 4);
```

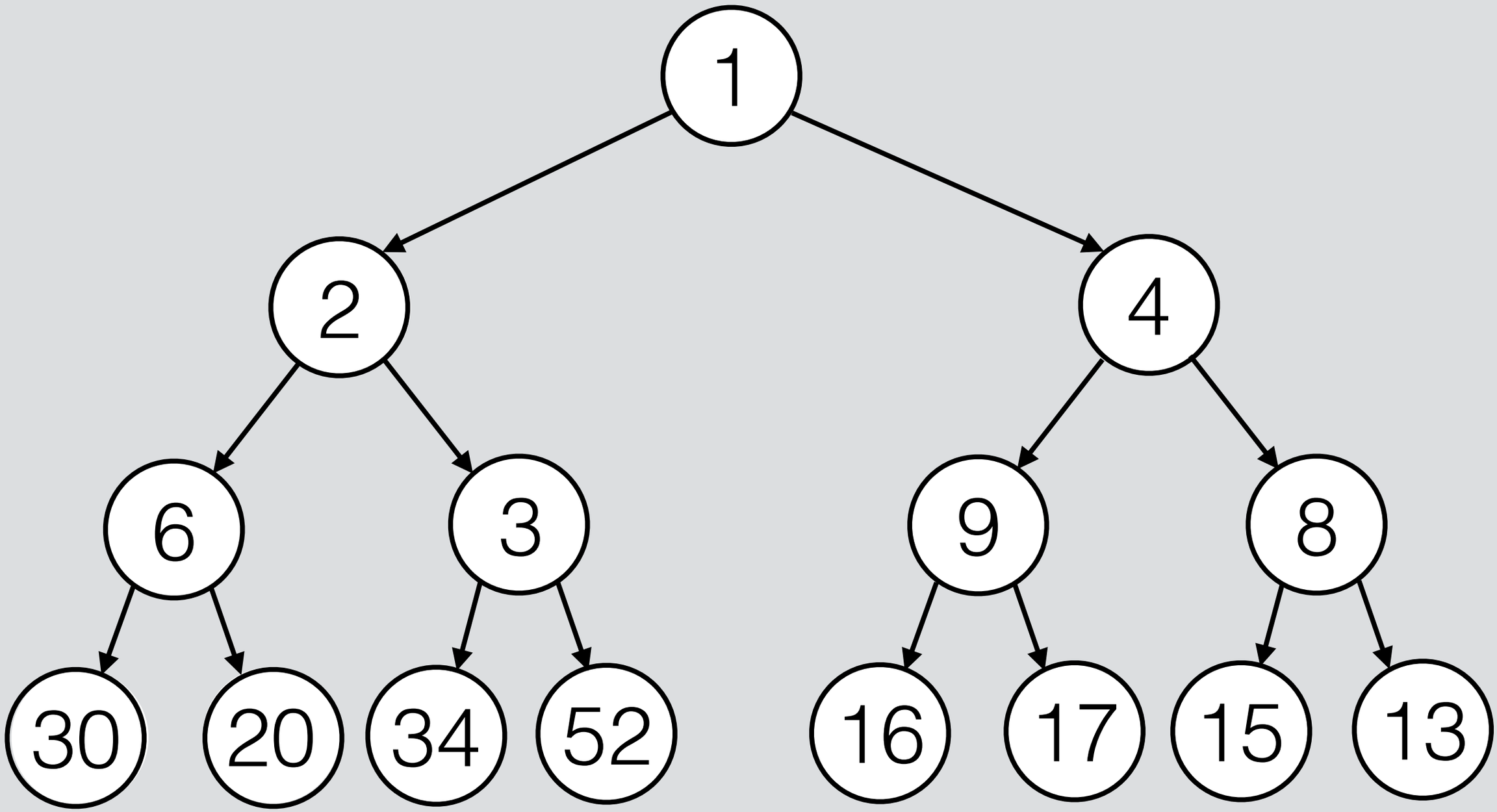


1. Insert at leaf.
2. *Bubble* up.

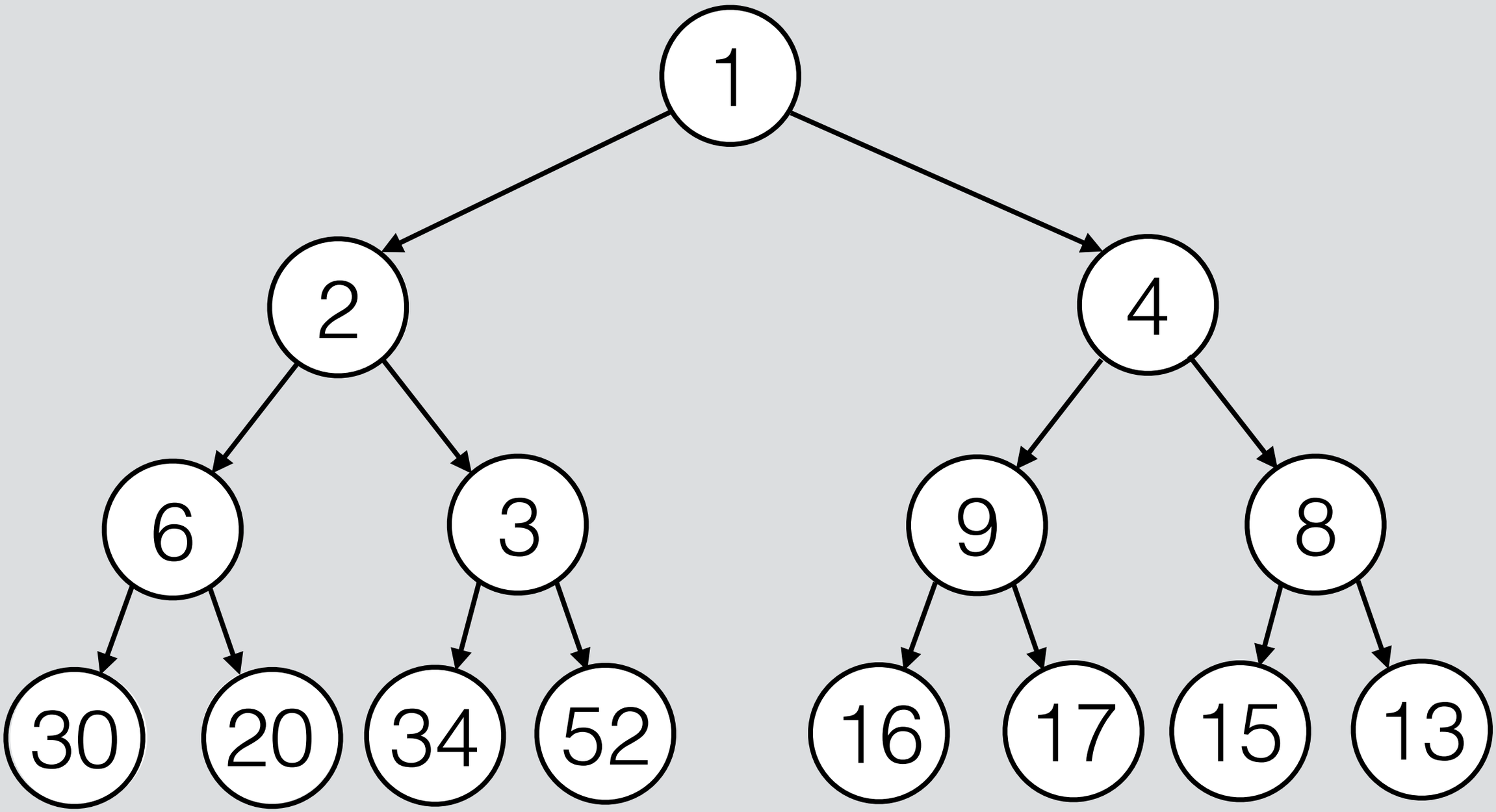
```
push(4, 4);
```




1
2
4
6
3
9
8
30
20
34
52
16
17
15
13

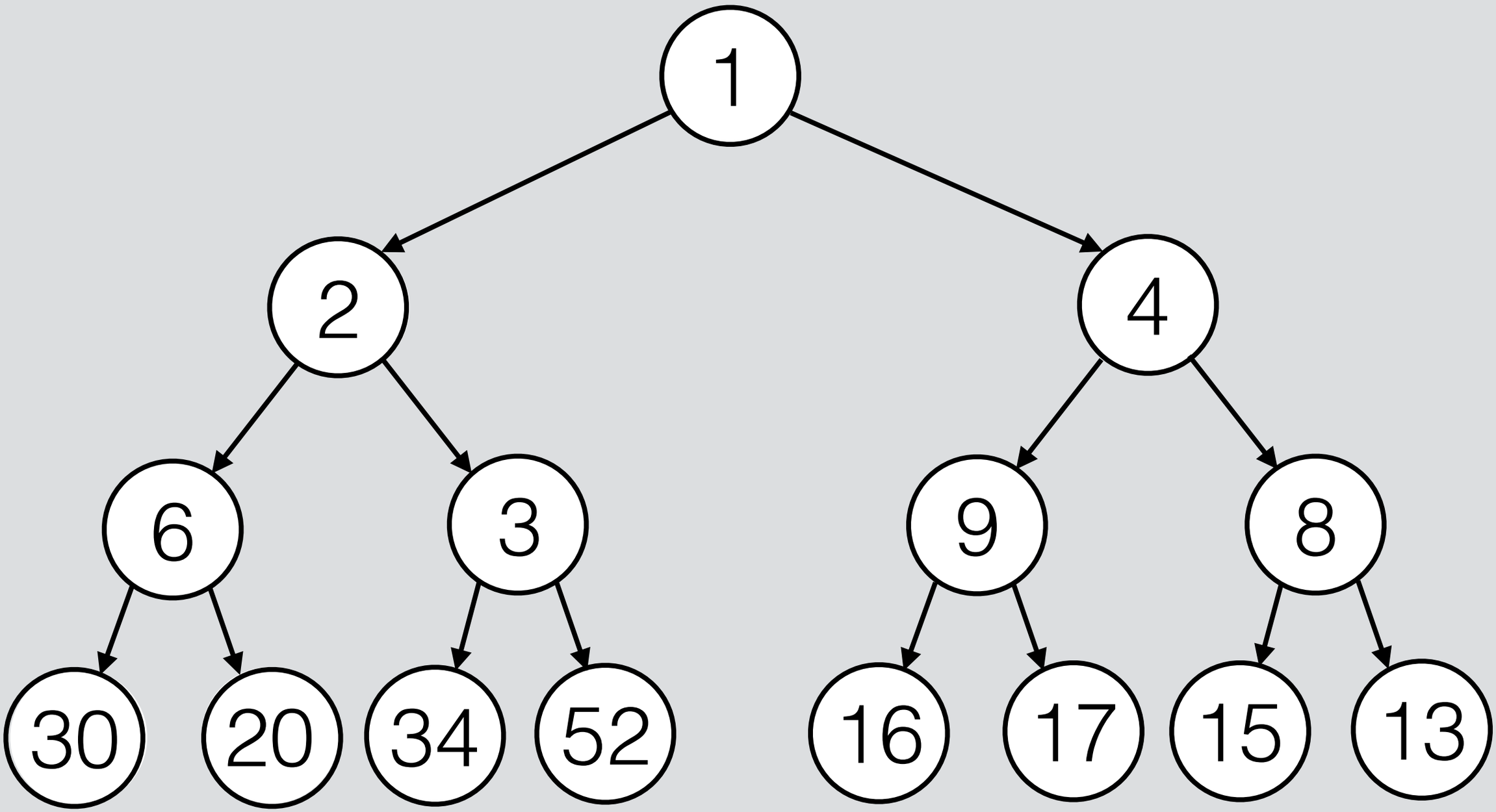

1
2
4
6
3
9
8
30
20
34
52
16
17
15
13

```
push(7, 7);
```



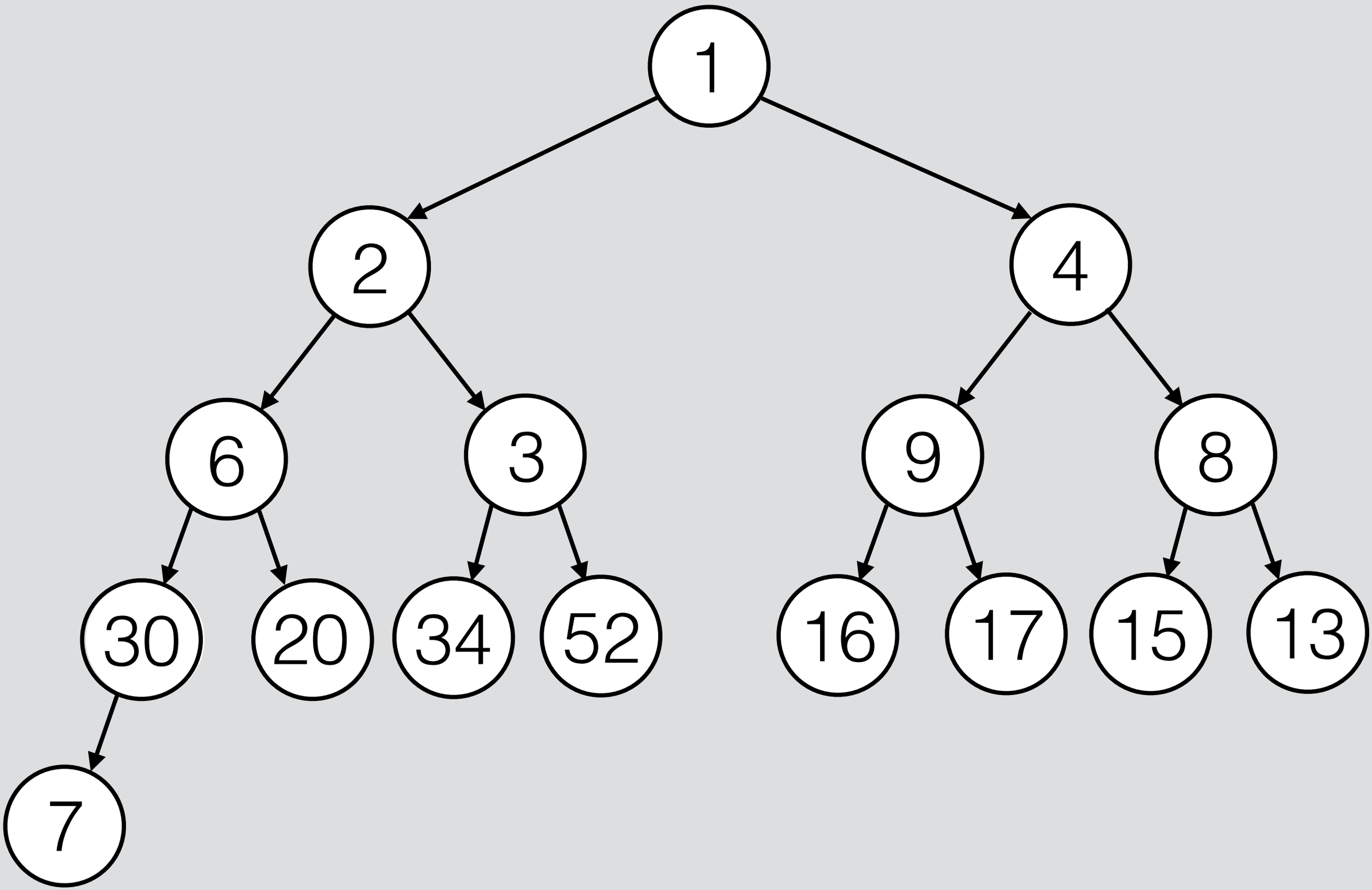
1. Insert at leaf.

```
push(7, 7);
```



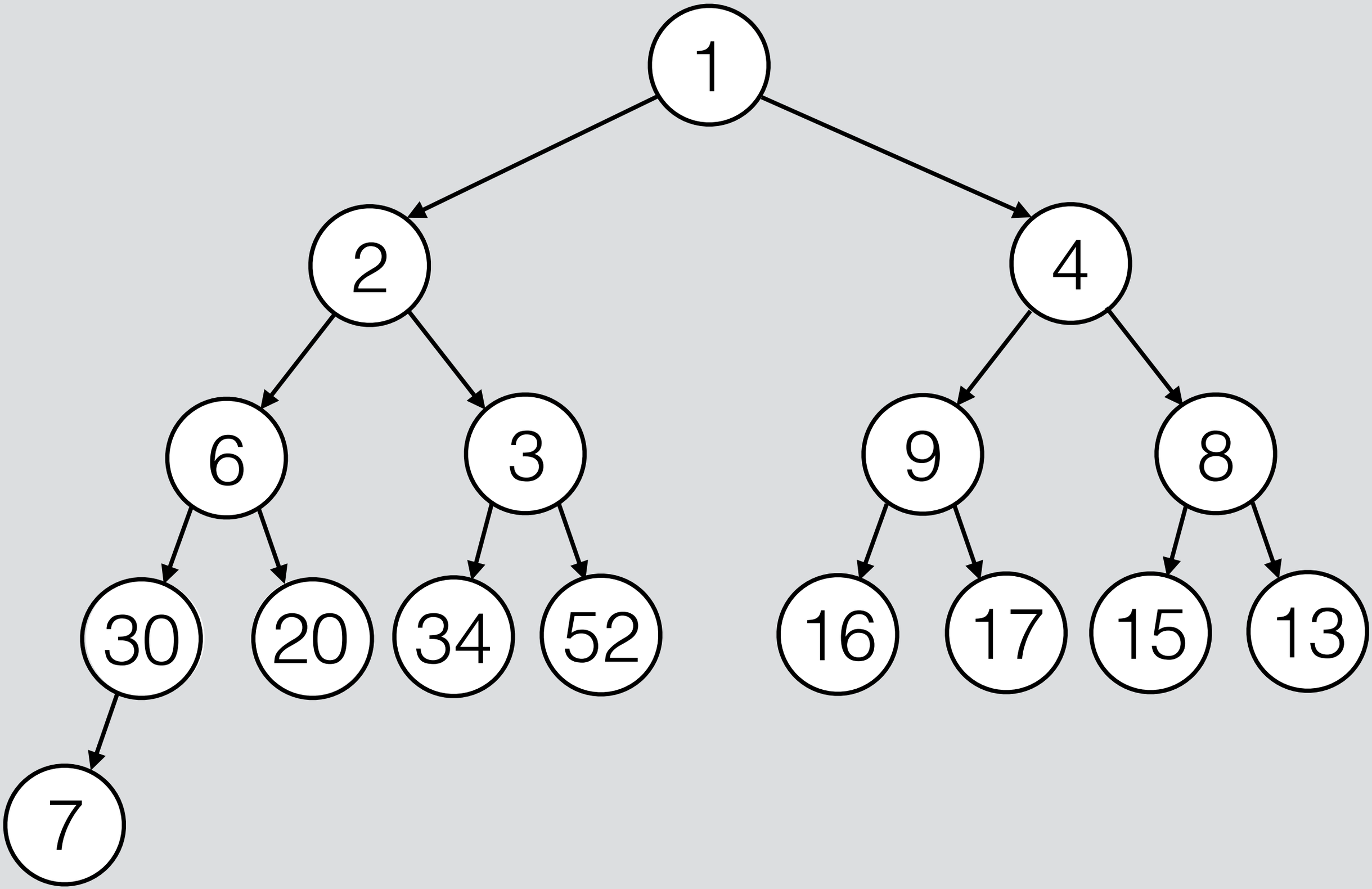
1. Insert at leaf.

```
push(7, 7);
```



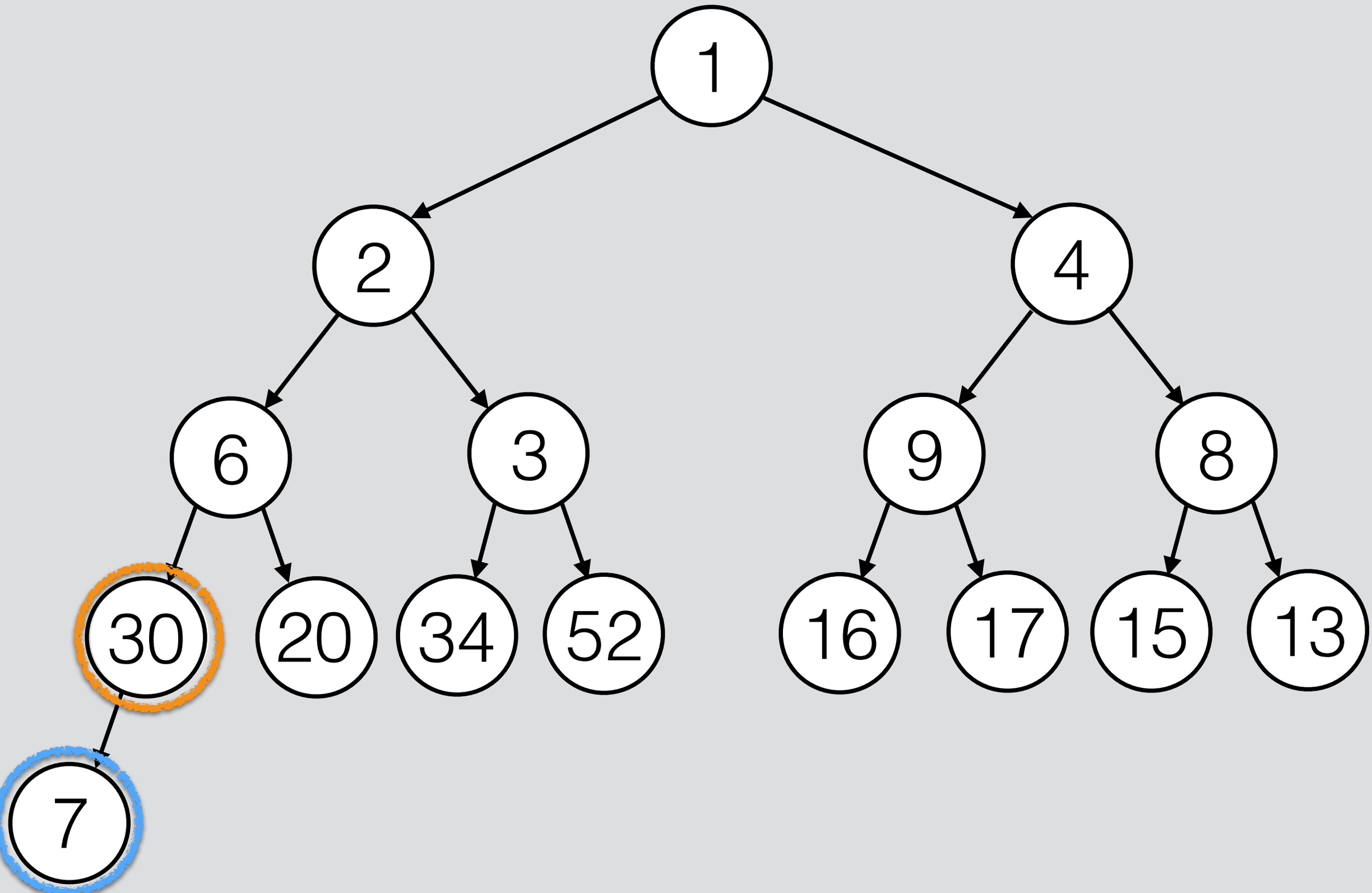
1. Insert at leaf.
2. *Bubble* up.

```
push(7, 7);
```



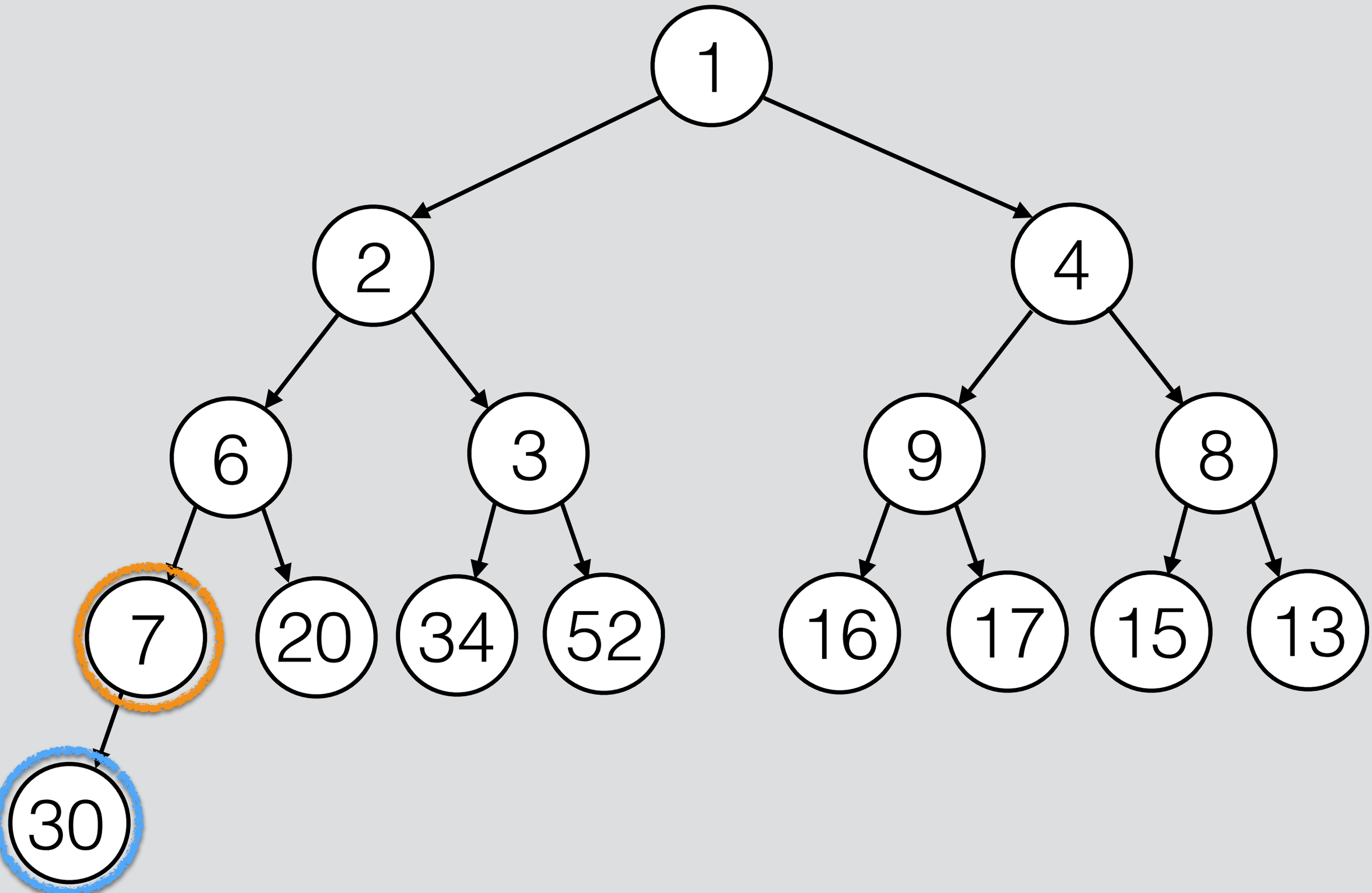
1. Insert at leaf.
2. *Bubble* up.

```
push(7, 7);
```



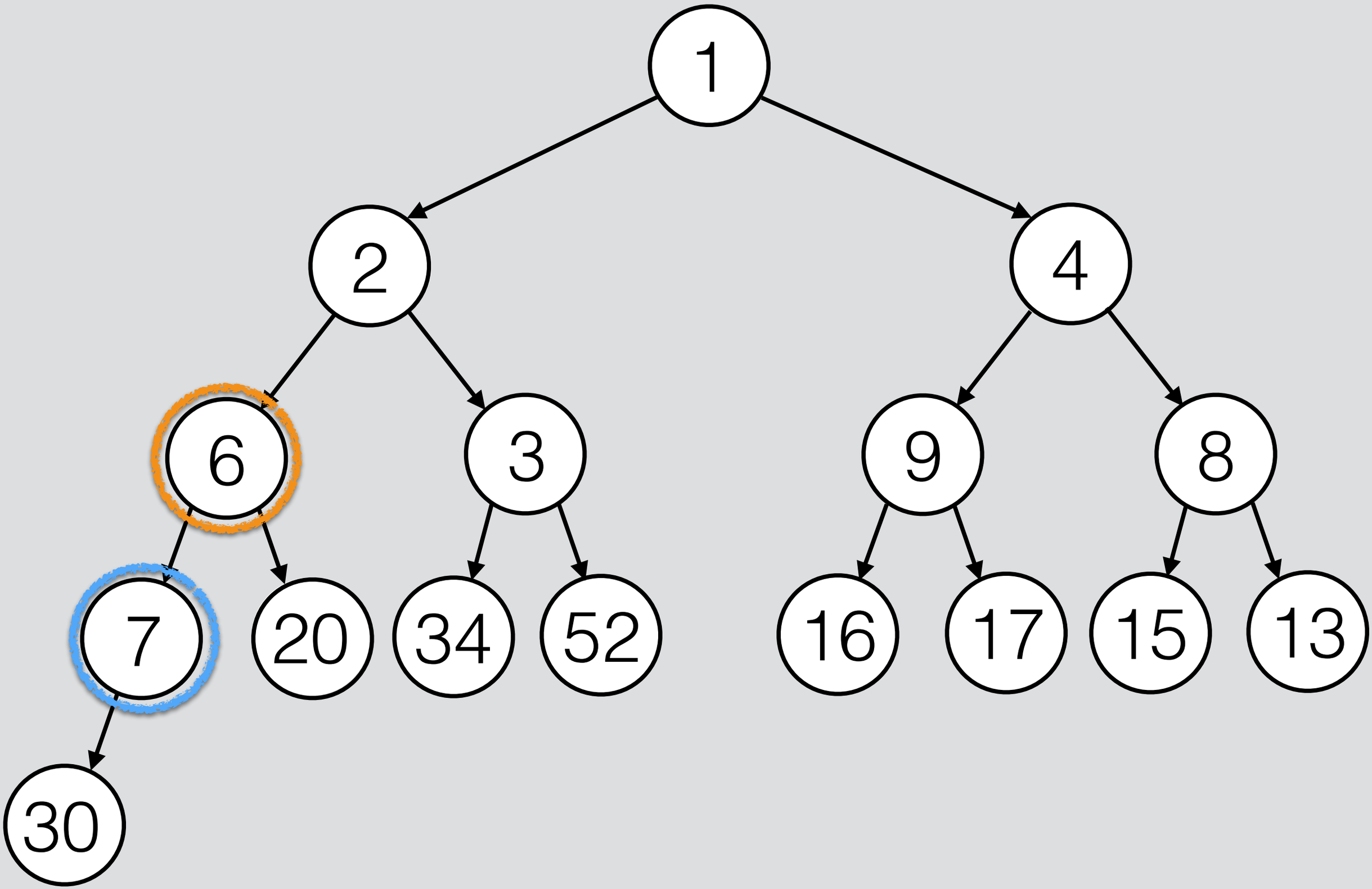
1. Insert at leaf.
2. *Bubble* up.

```
push(7, 7);
```



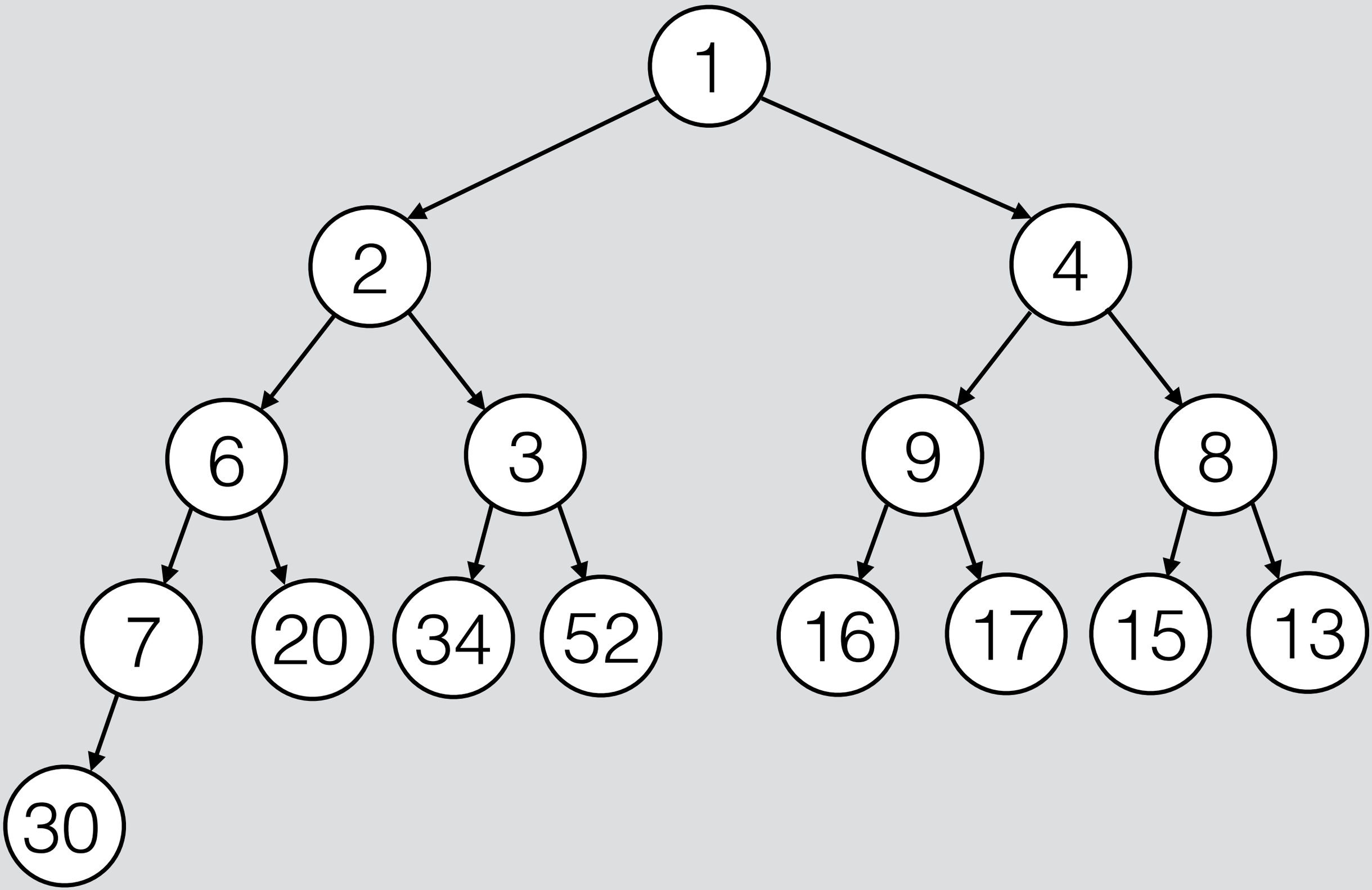
1. Insert at leaf.
2. *Bubble* up.

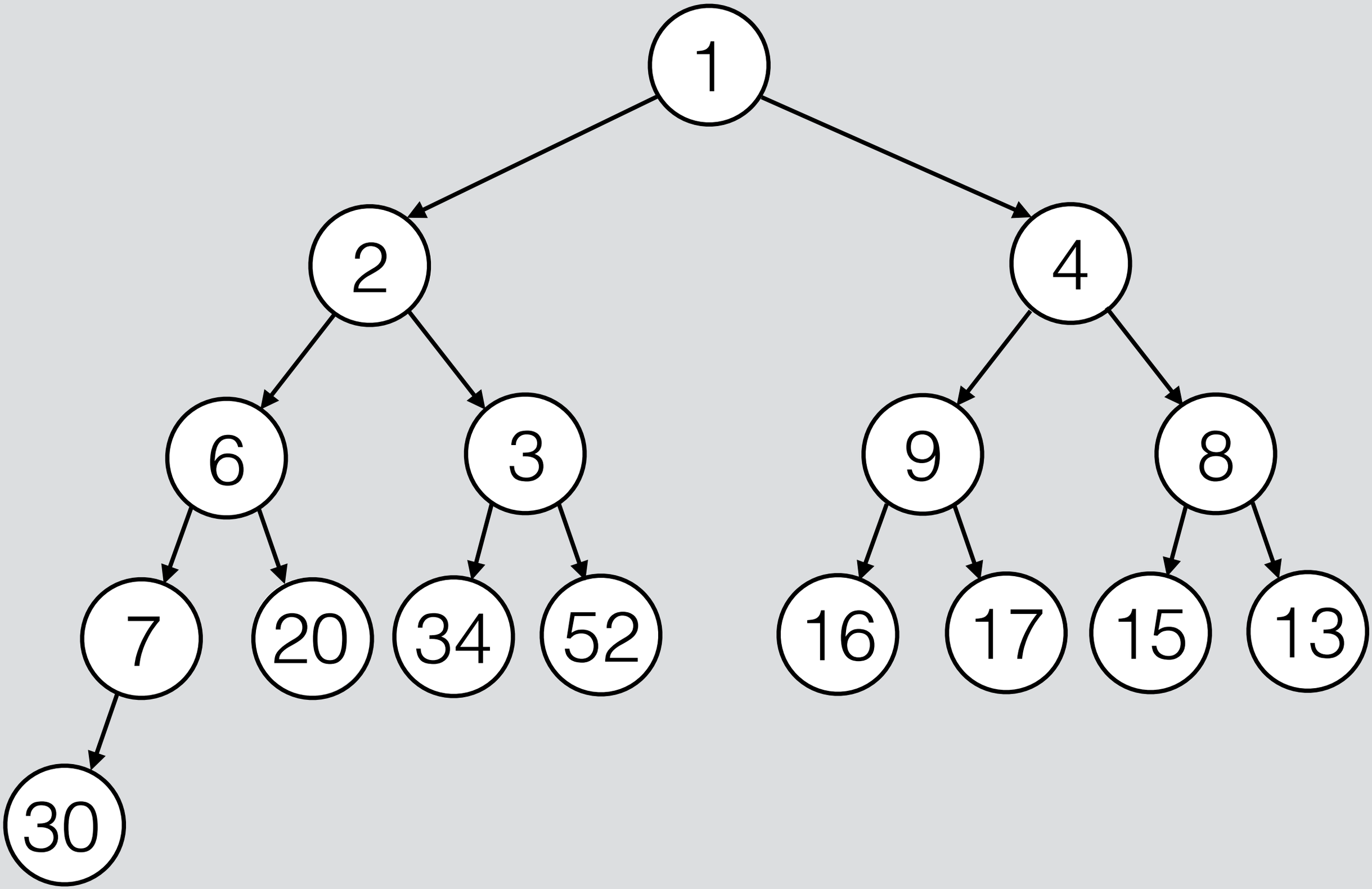
```
push(7, 7);
```



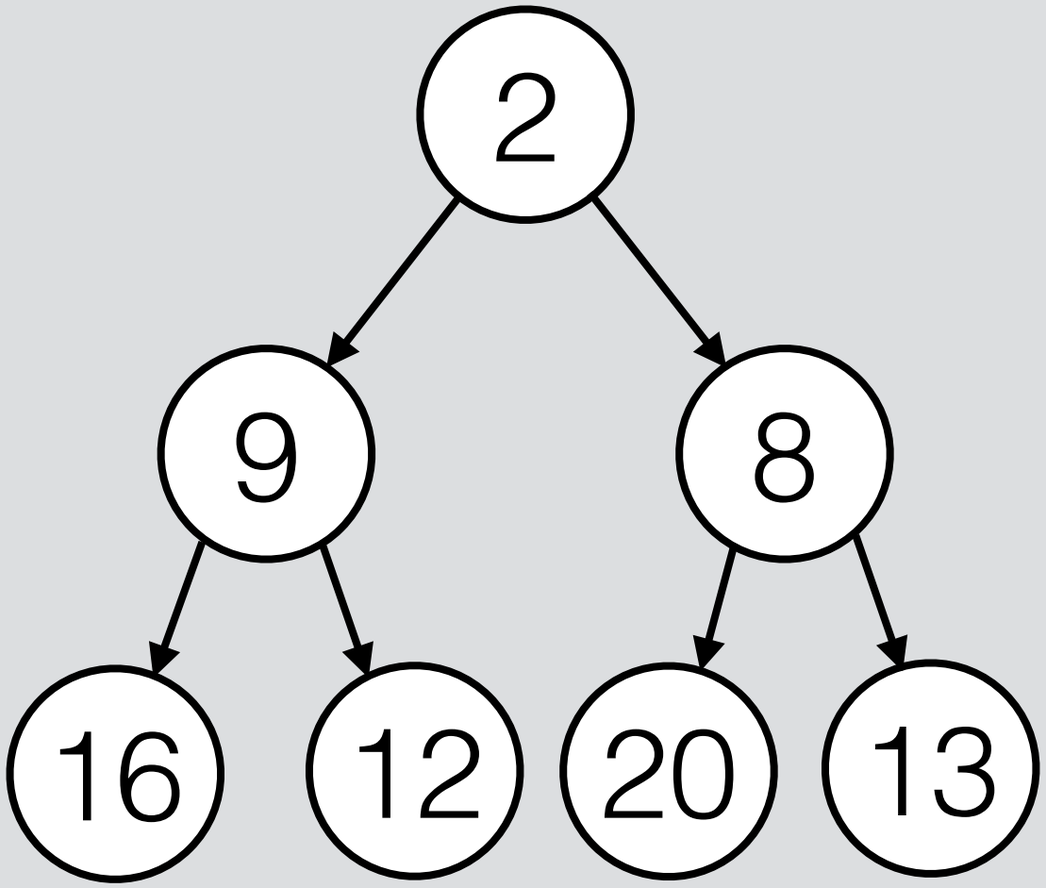
1. Insert at leaf.
2. *Bubble* up.

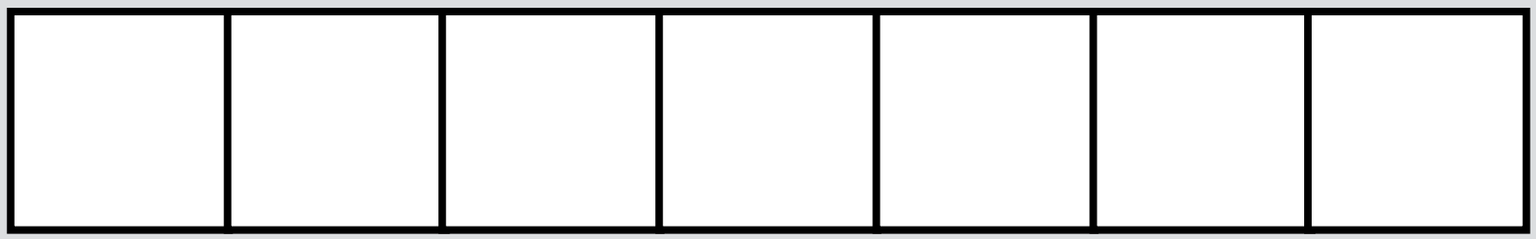
```
push(7, 7);
```

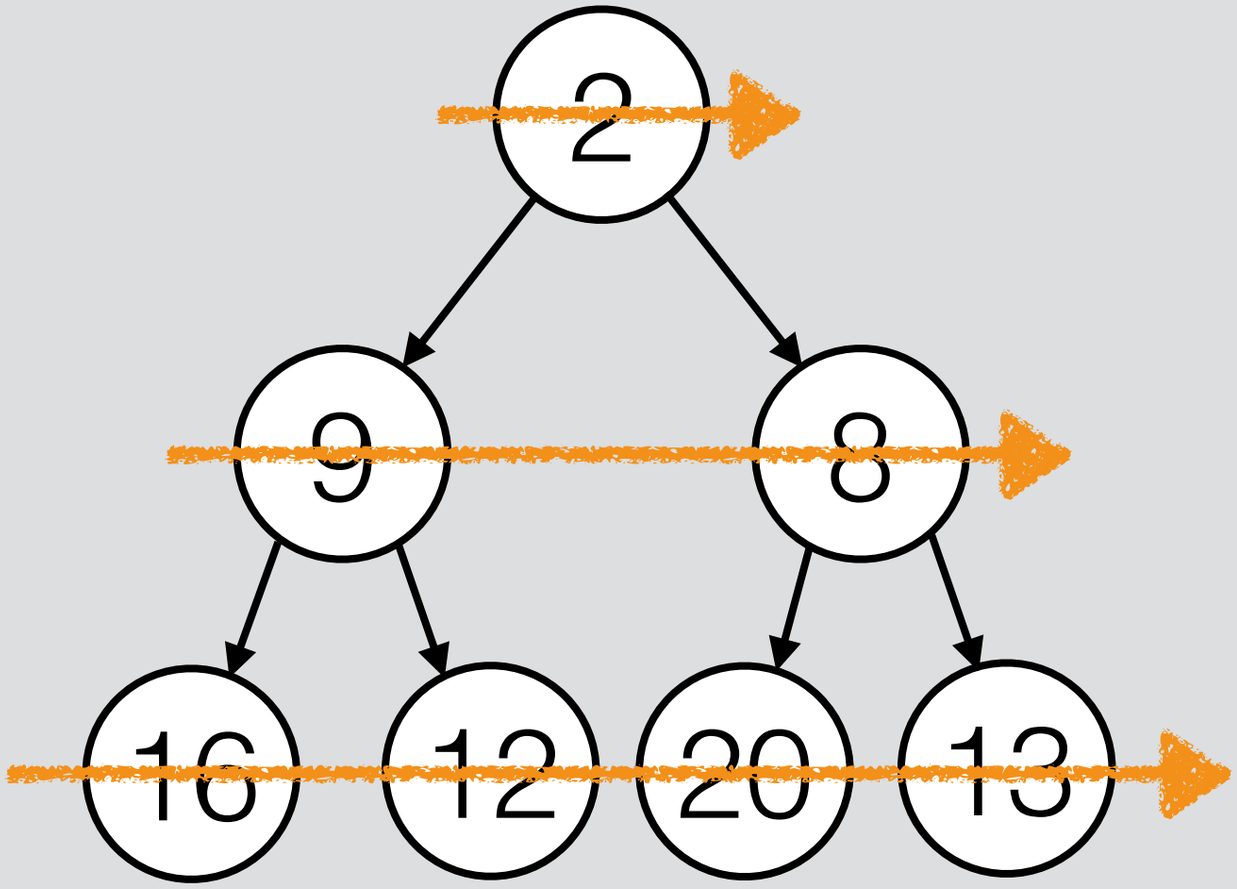


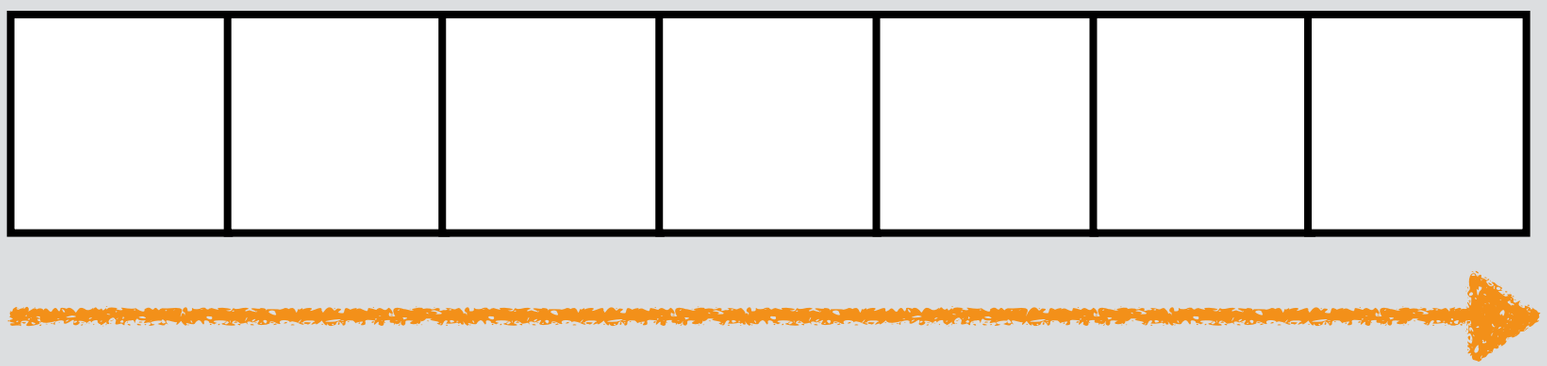

1
2
4
6
3
9
8
7
20
34
52
16
17
15
13
30

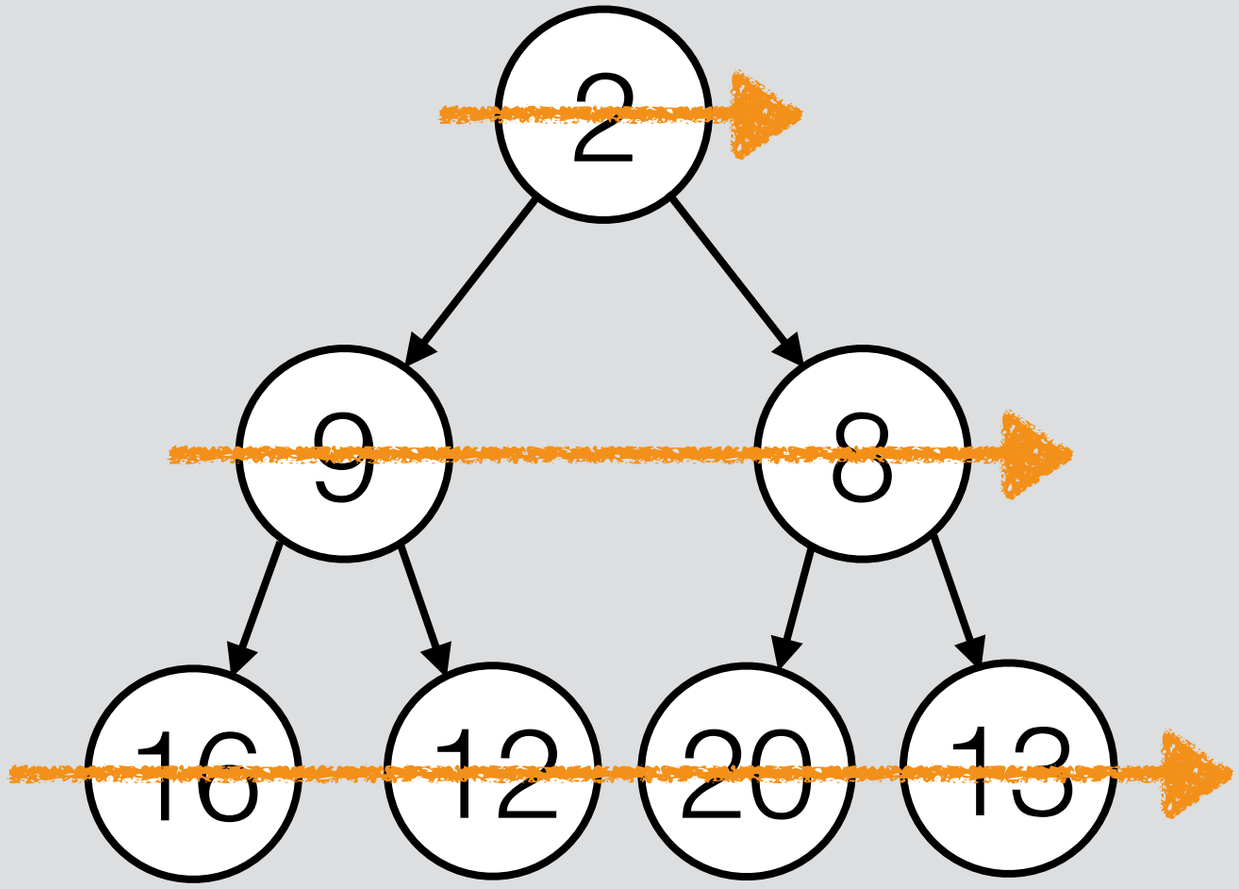
Heap Implementation

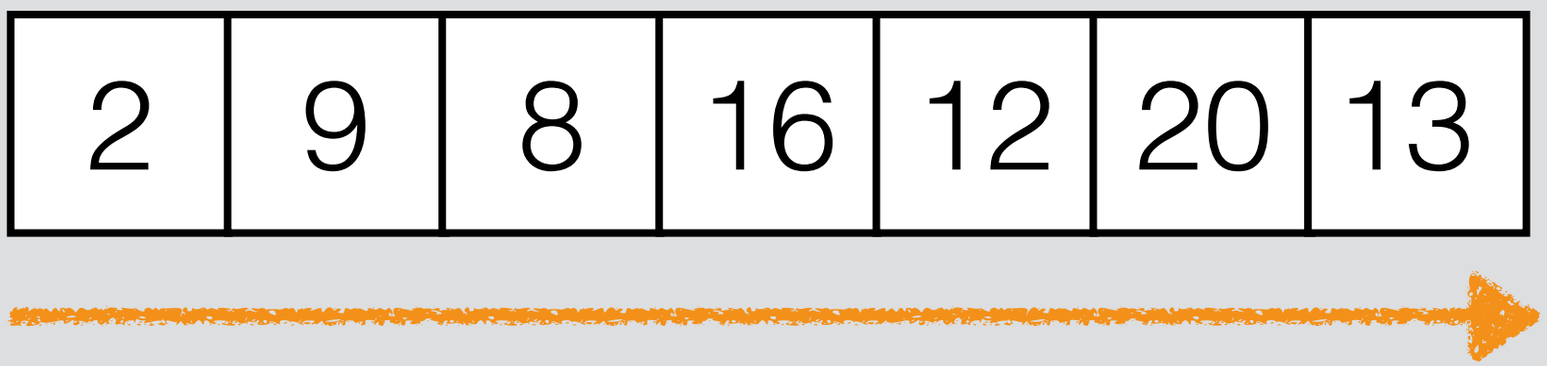

2
9
8
16
12
20
13

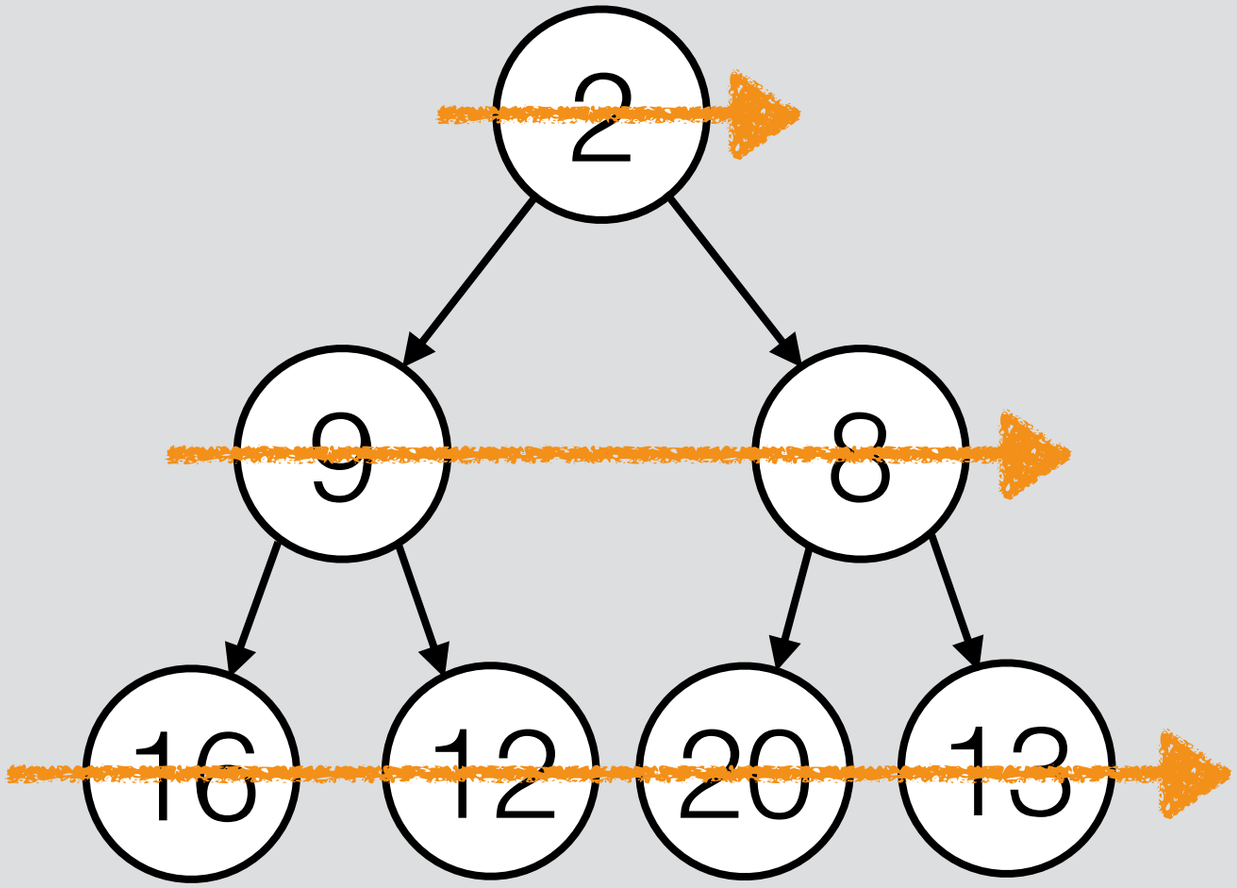


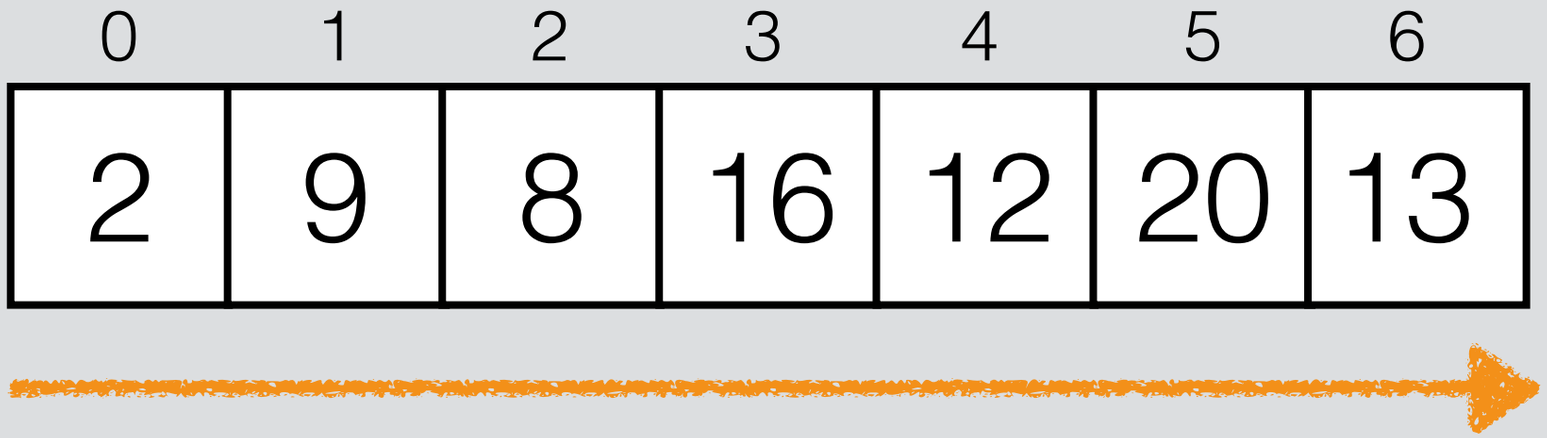

2
9
8
16
12
20
13




2
9
8
16
12
20
13

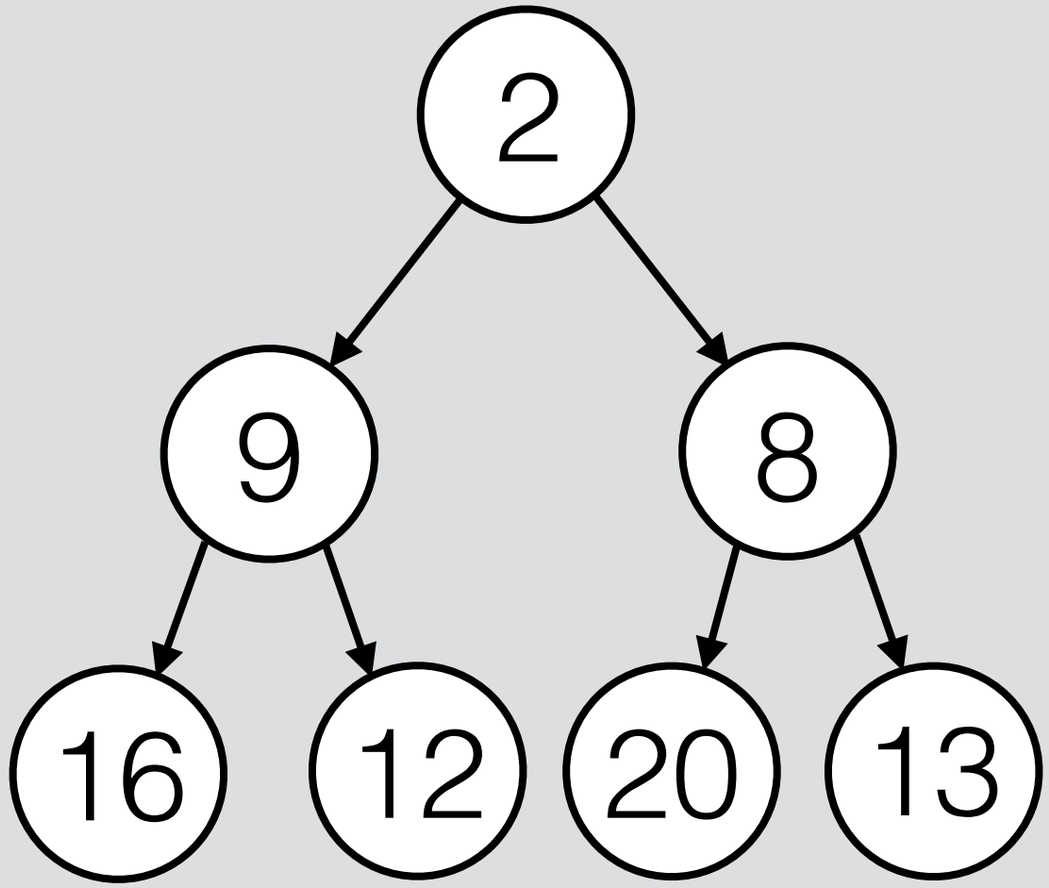

2
9
8
16
12
20
13

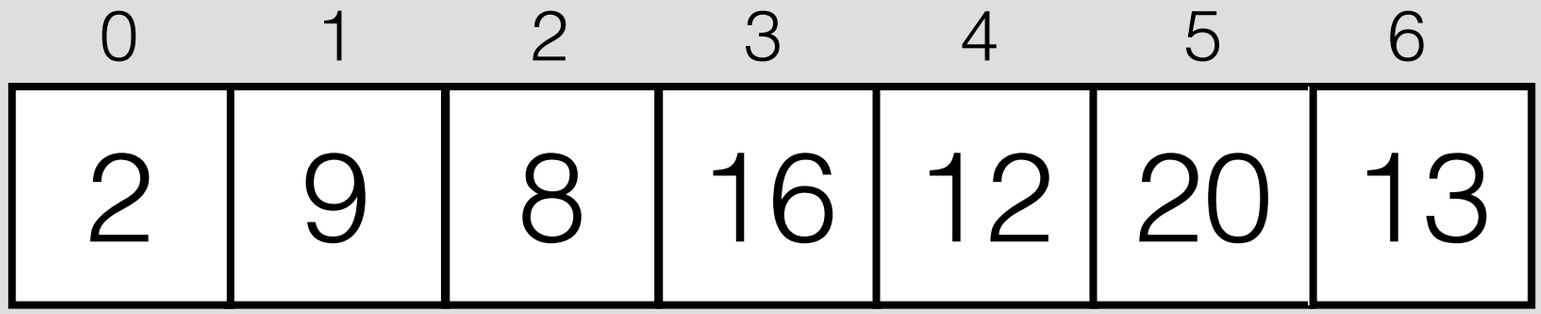




Parent: p
Children: 2p+1, 2p+2

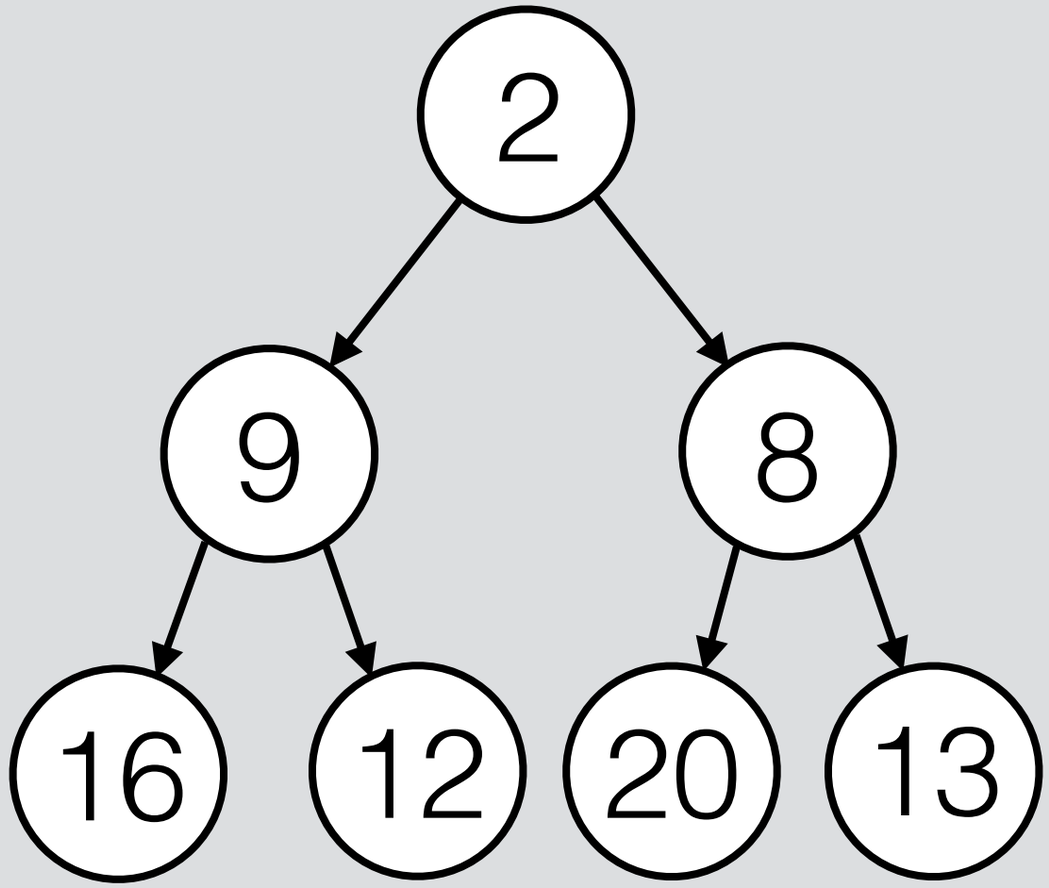
Parent: (c-1)/2
Child: c



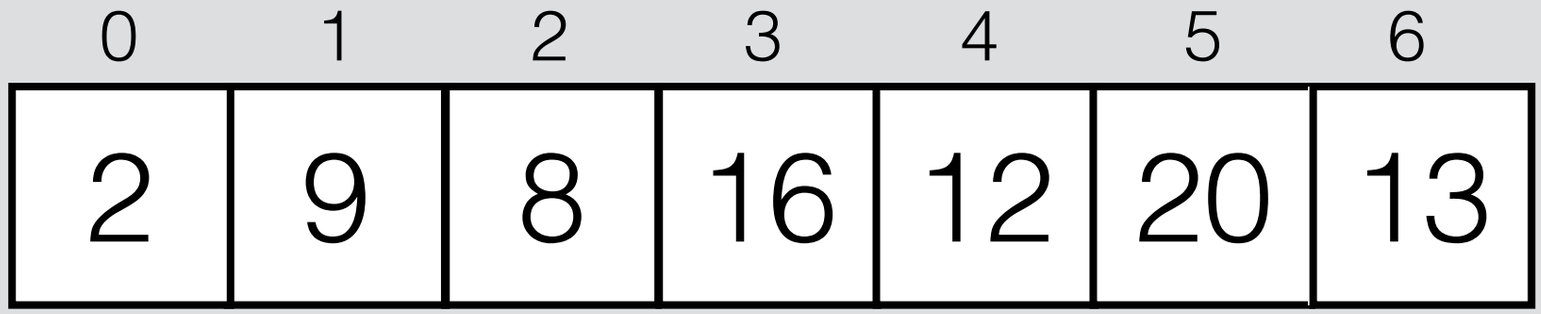


Parent: p
Children: 2p+1, 2p+2

Parent: (c-1)/2
Child: c

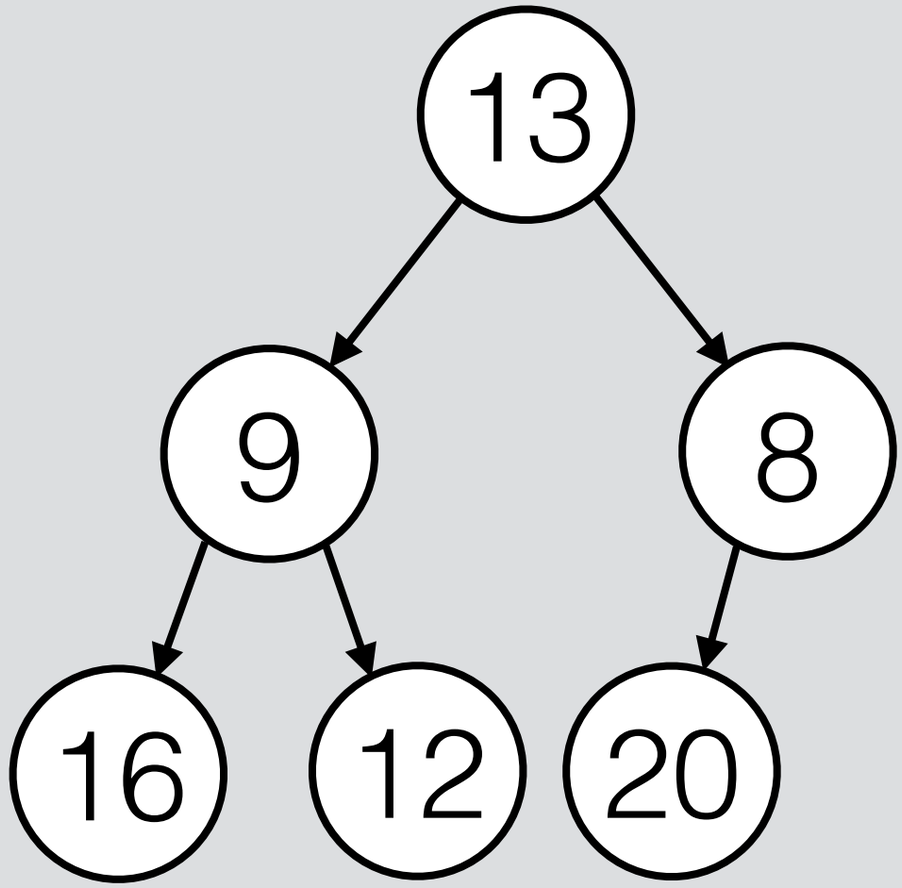


pop();

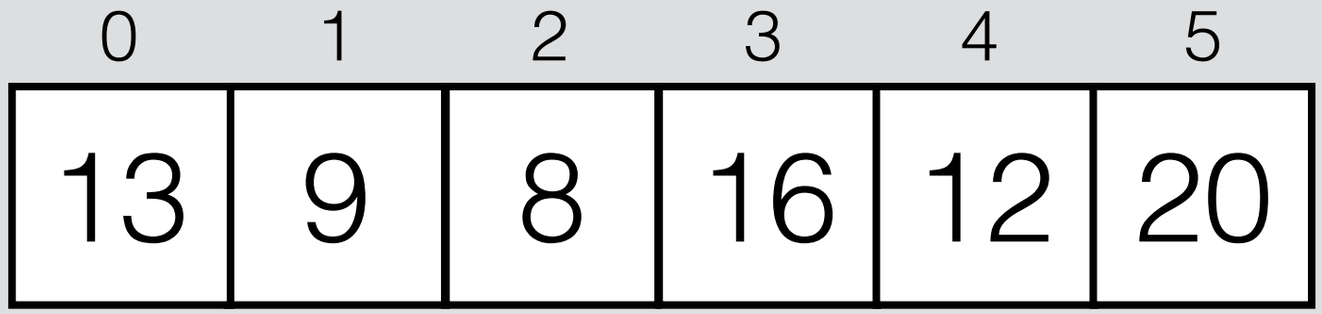


Parent: p
Children: 2p+1, 2p+2

Parent: (c-1)/2
Child: c

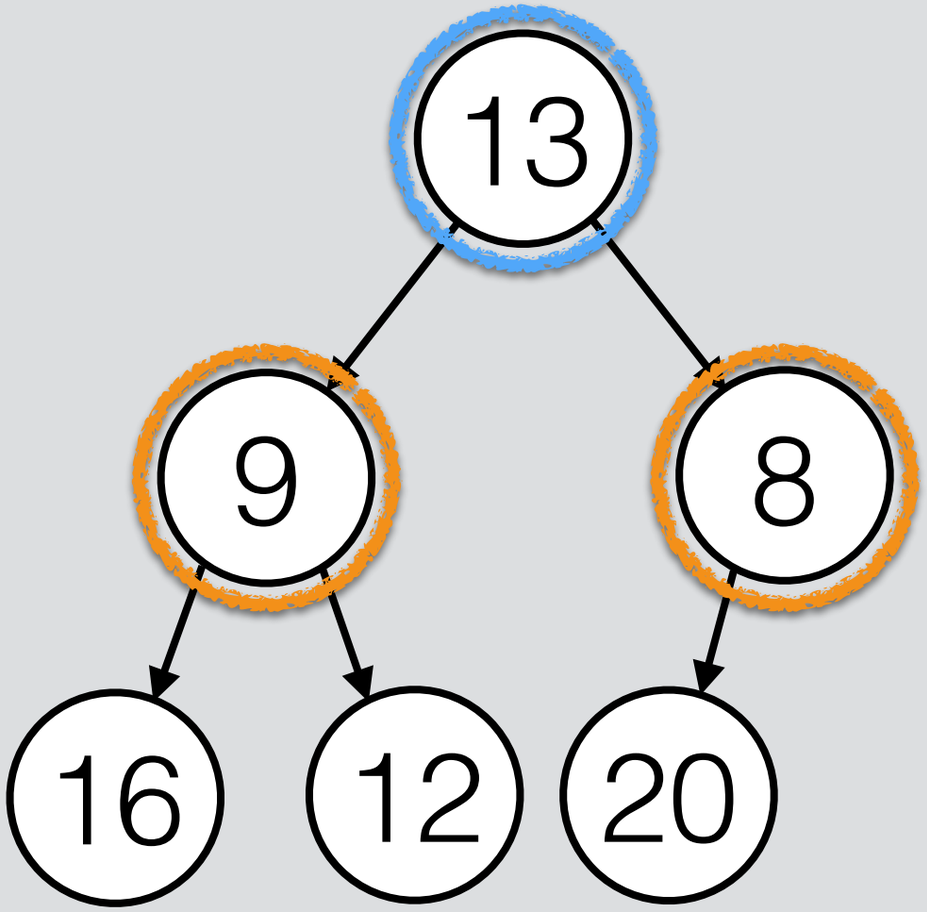



```
pop();
```

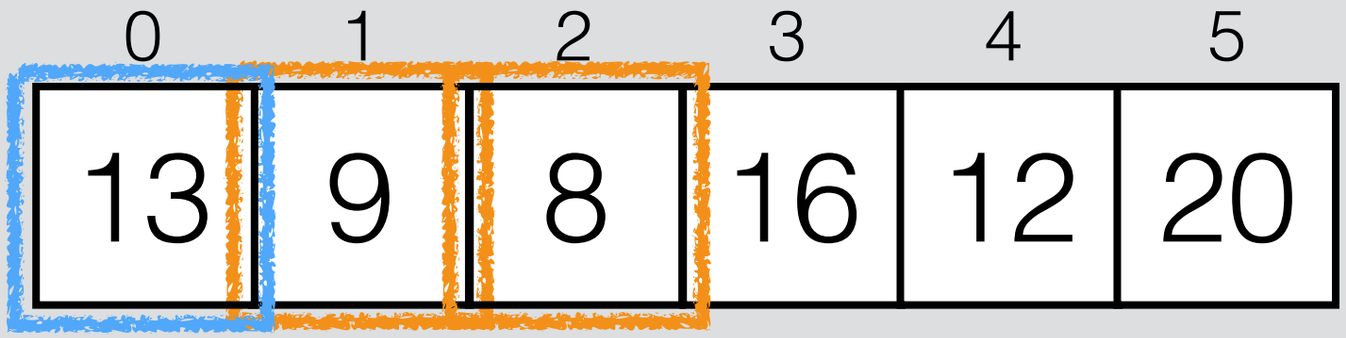


Parent: p
Children: 2p+1, 2p+2

Parent: (c-1)/2
Child: c

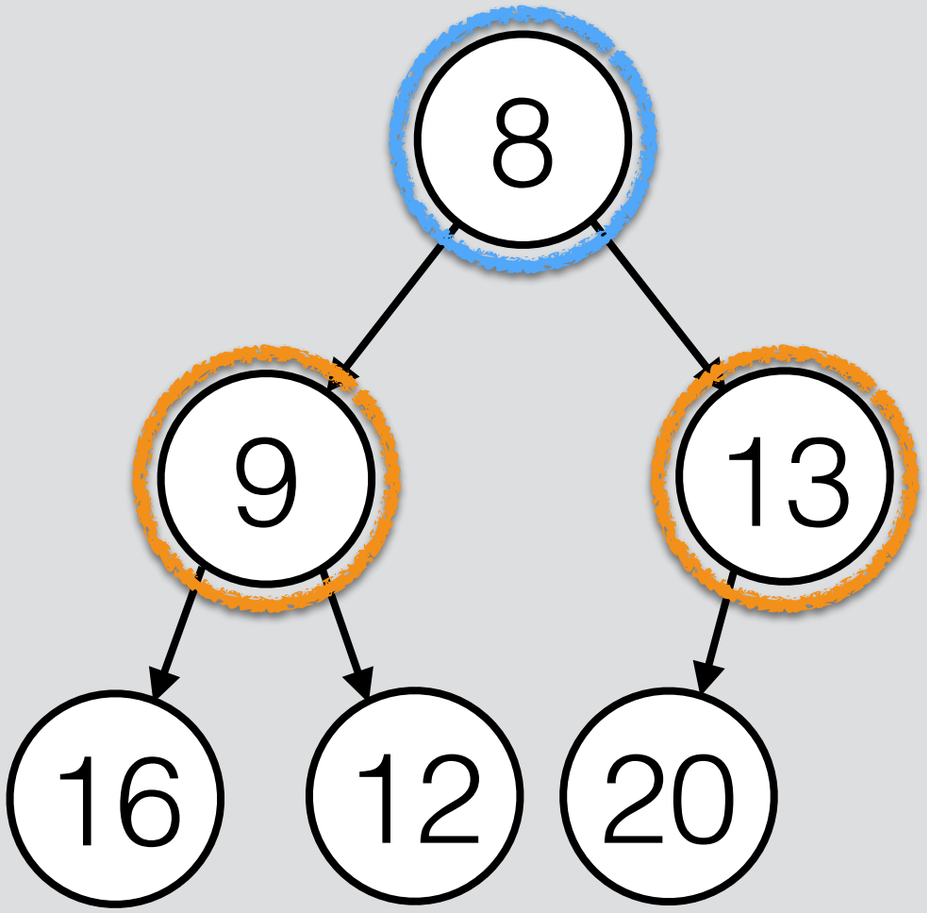


pop();

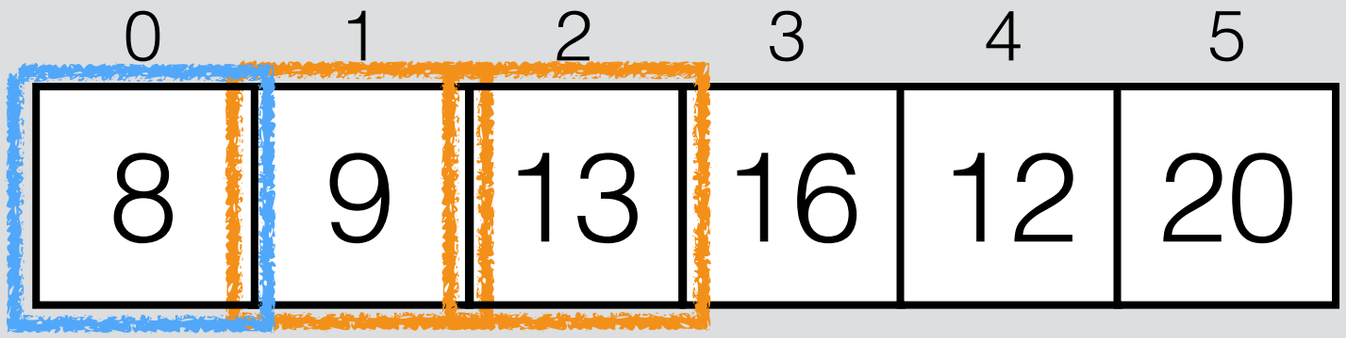


Parent: p
Children: 2p+1, 2p+2

Parent: (c-1)/2
Child: c

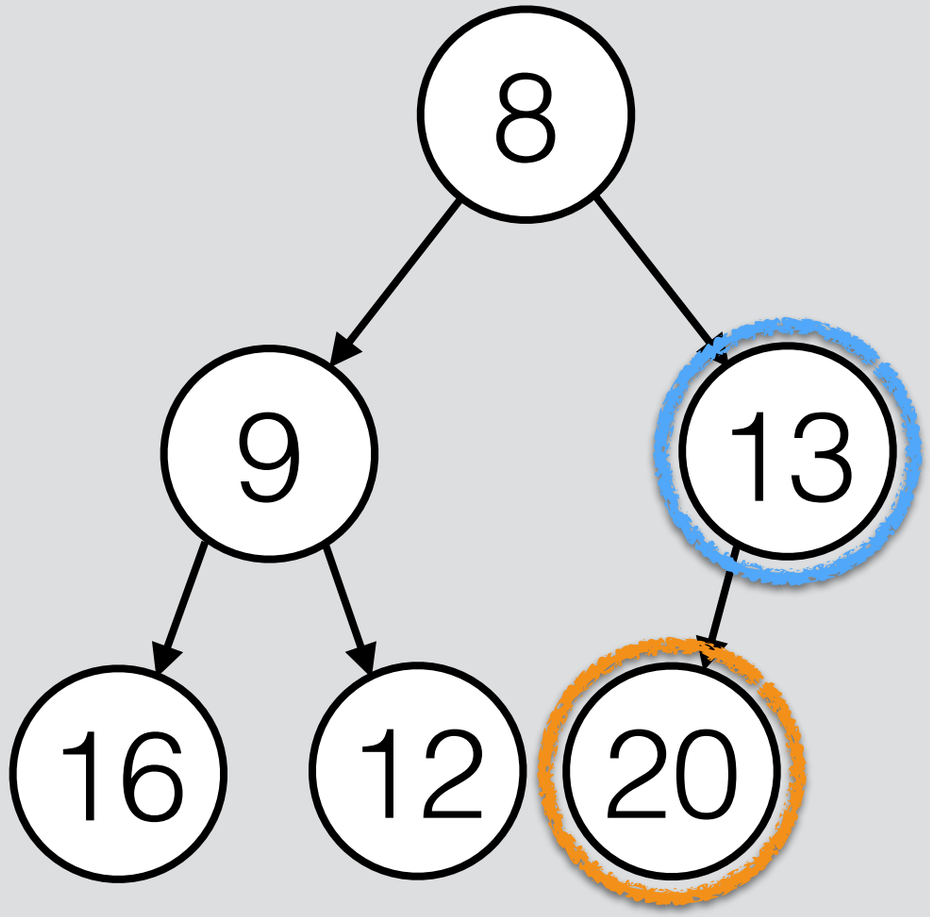


pop();

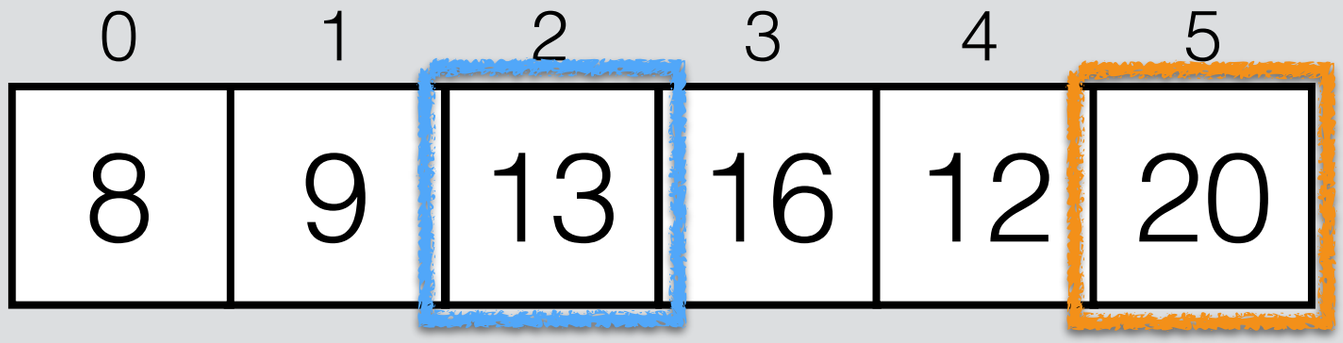


Parent: p
Children: 2p+1, 2p+2

Parent: (c-1)/2
Child: c

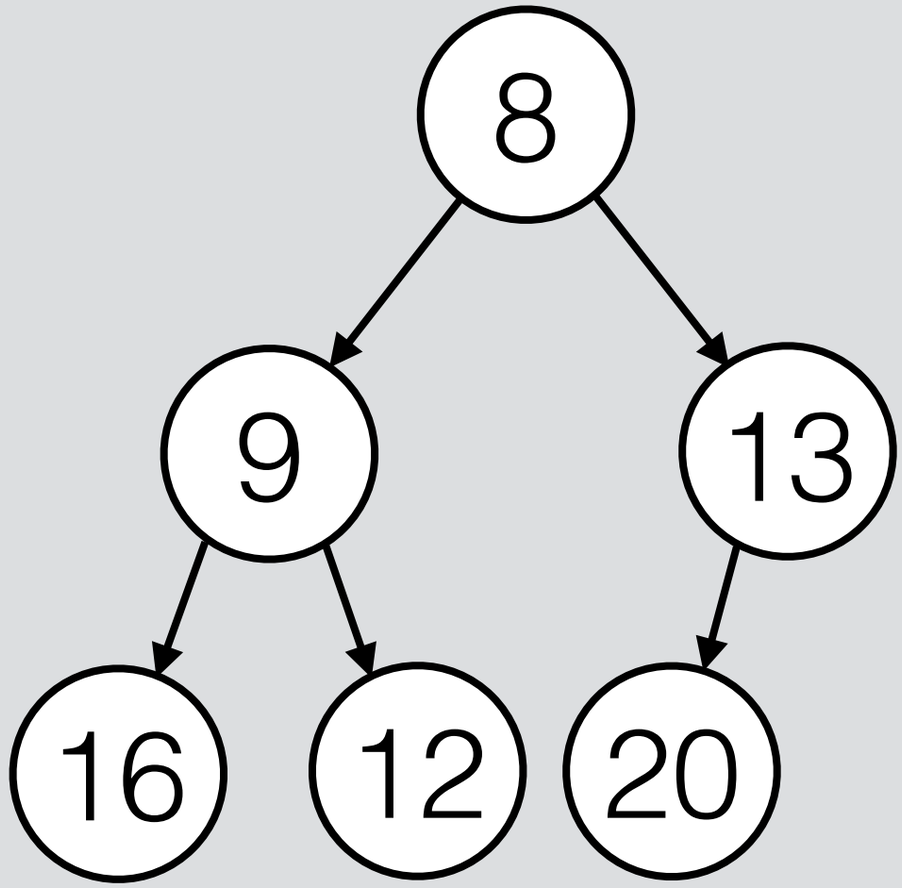


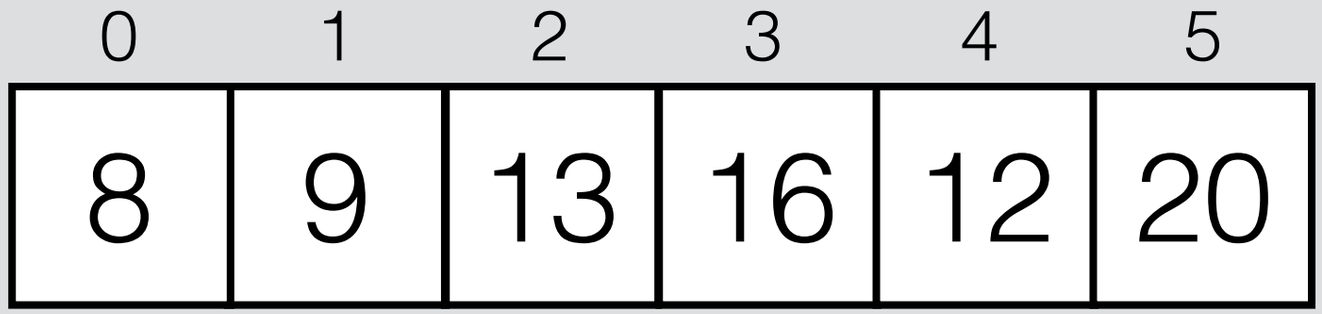
pop();



Parent: p
Children: 2p+1, 2p+2

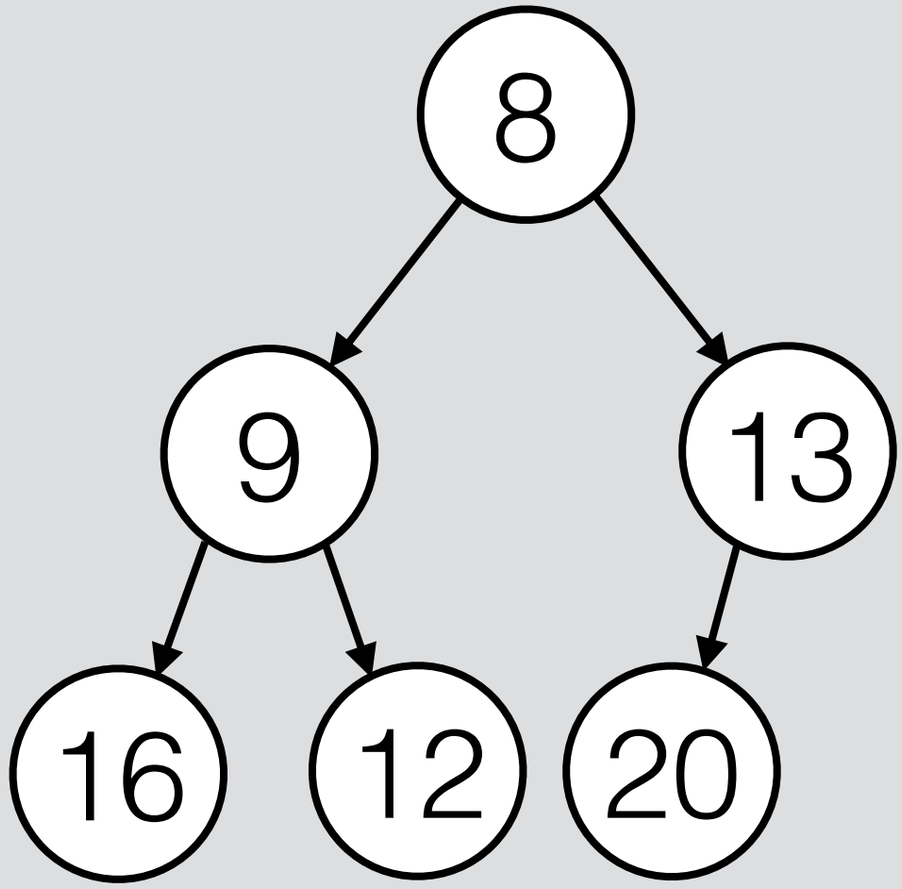
Parent: (c-1)/2
Child: c

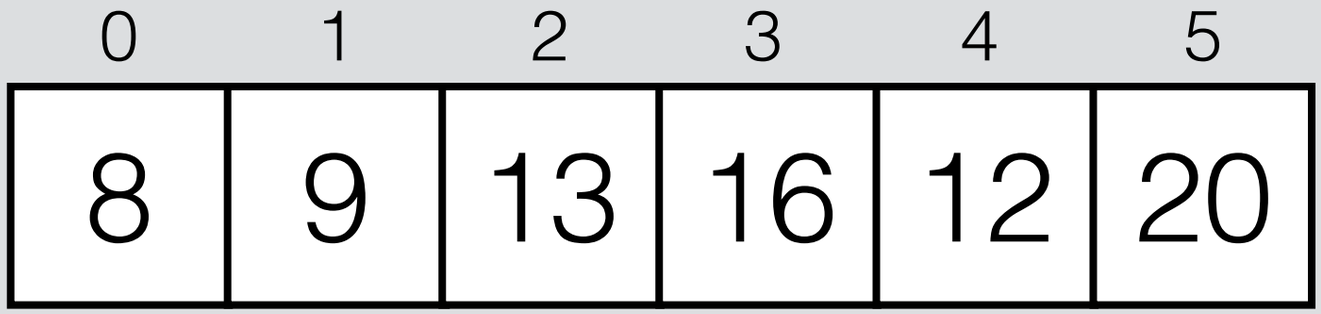




Parent: p
Children: 2p+1, 2p+2

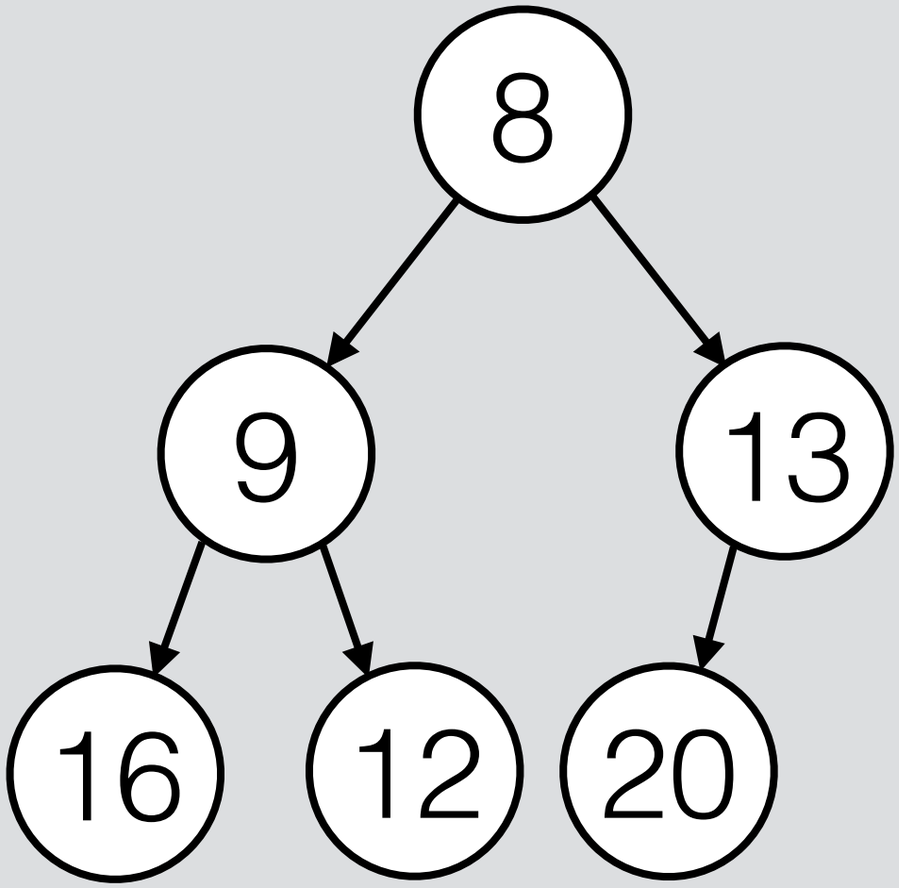
Parent: (c-1)/2
Child: c



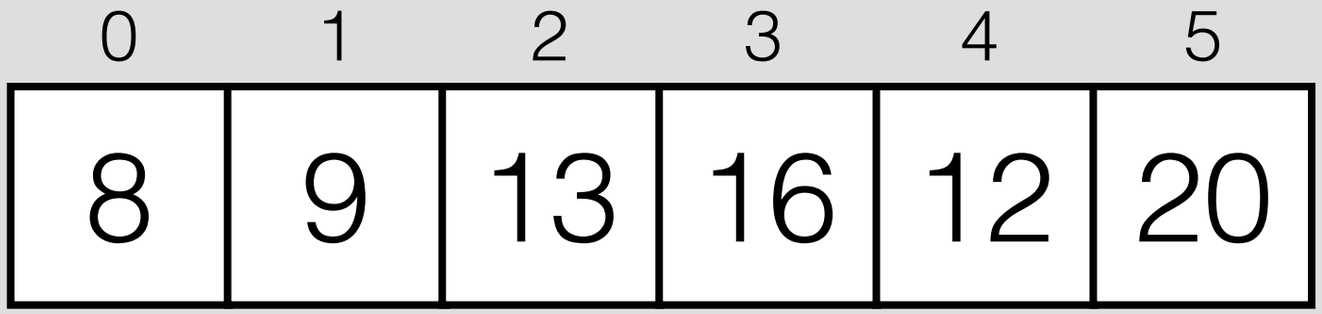


Parent: p
Children: 2p+1, 2p+2

Parent: (c-1)/2
Child: c

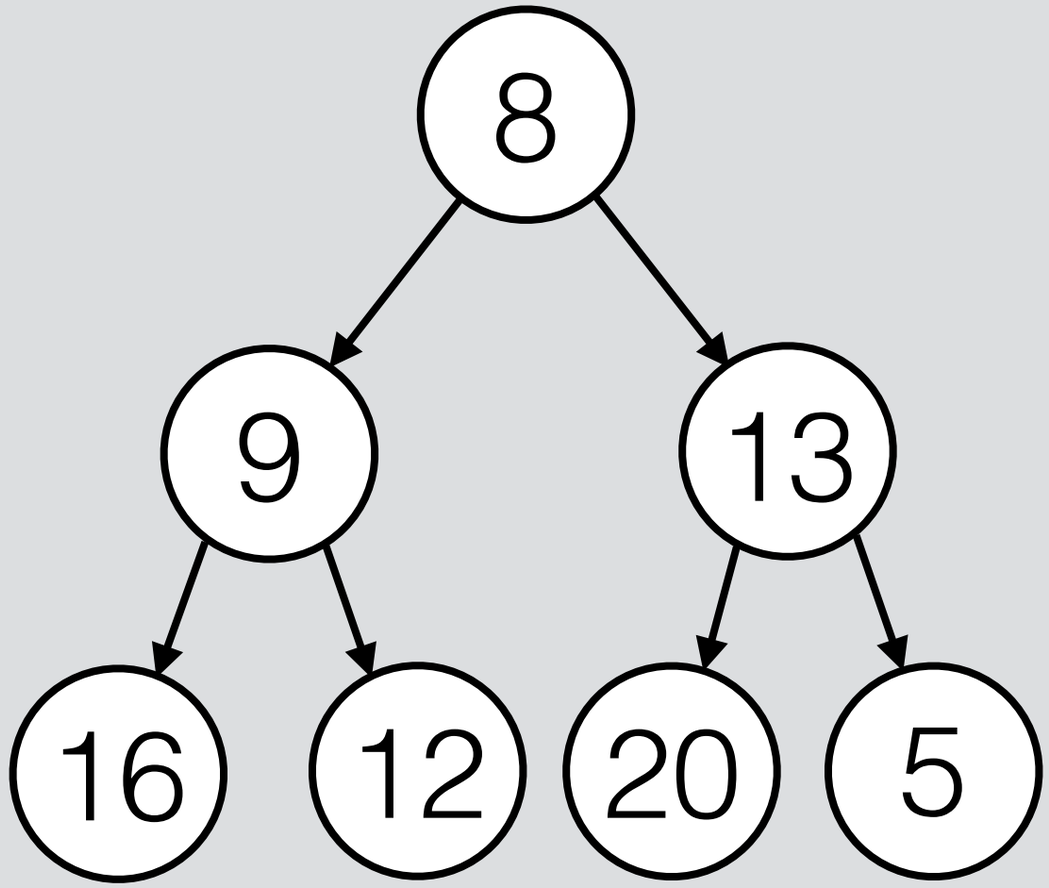



```
push(5);
```



Parent: p
Children: 2p+1, 2p+2

Parent: (c-1)/2
Child: c

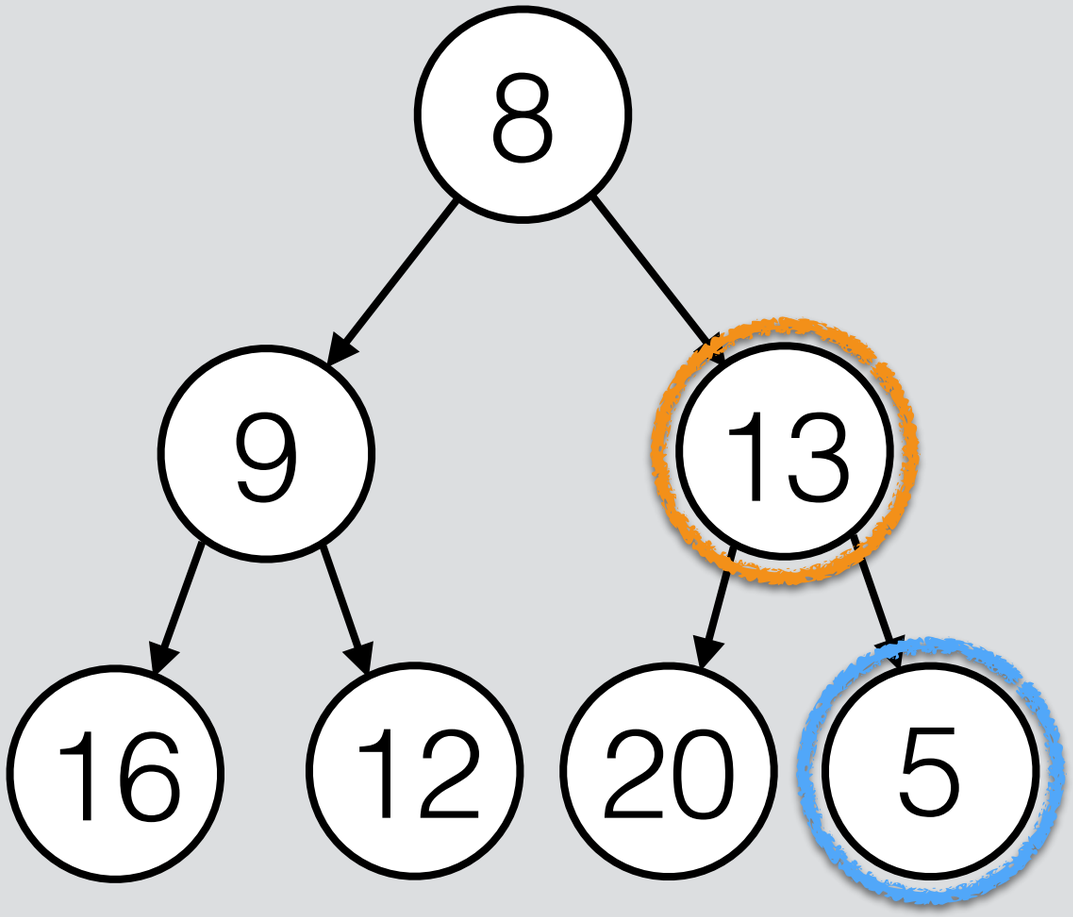



```
push(5);
```

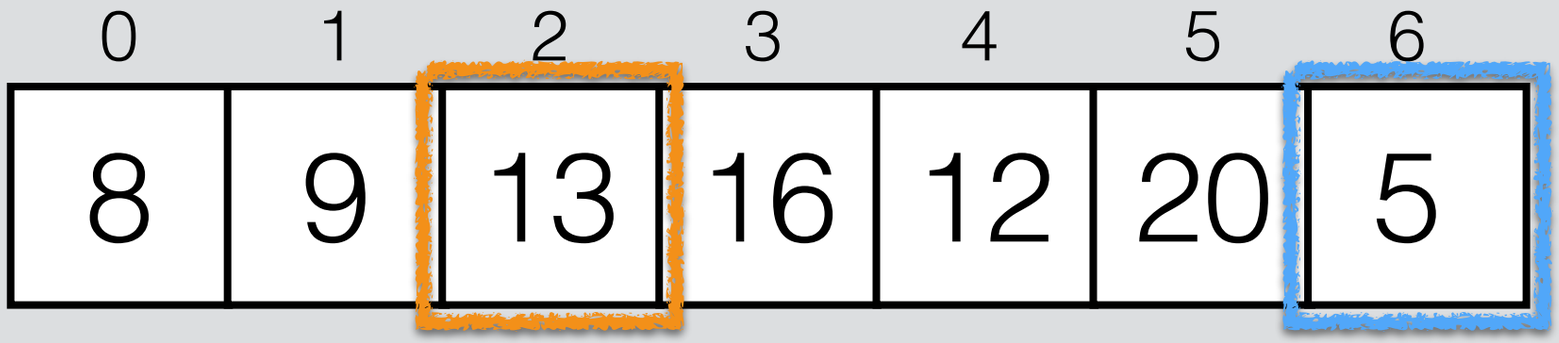
0	1	2	3	4	5	6
8	9	13	16	12	20	5

Parent: p
Children: 2p+1, 2p+2

Parent: (c-1)/2
Child: c

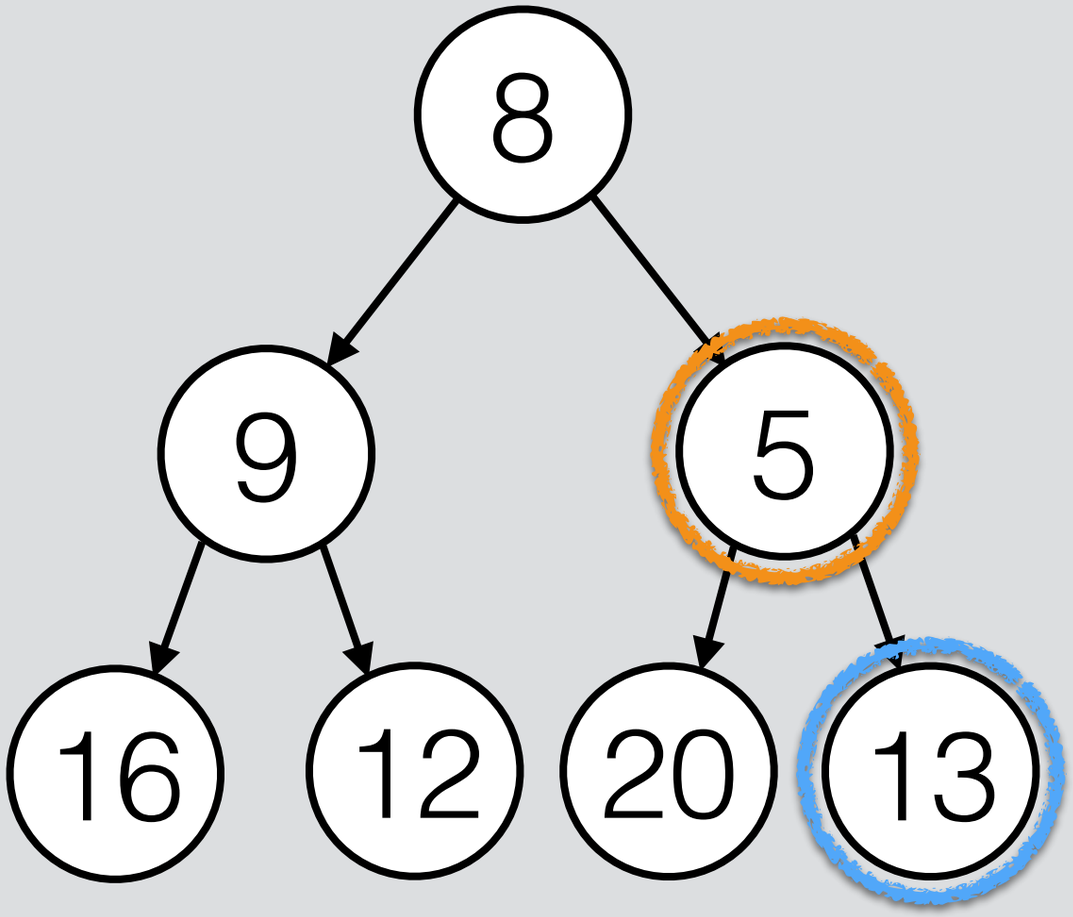



```
push(5);
```

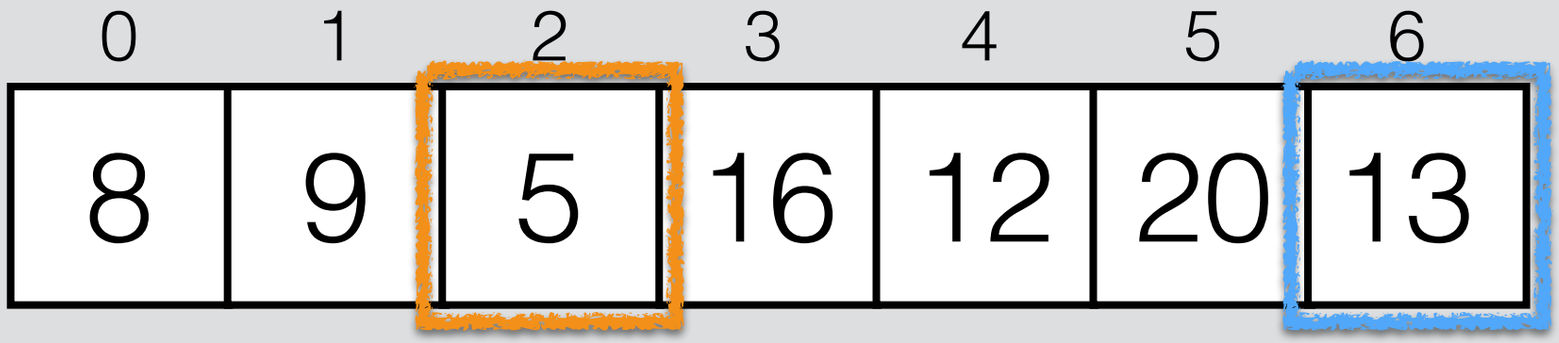


Parent: p
Children: 2p+1, 2p+2

Parent: (c-1)/2
Child: c

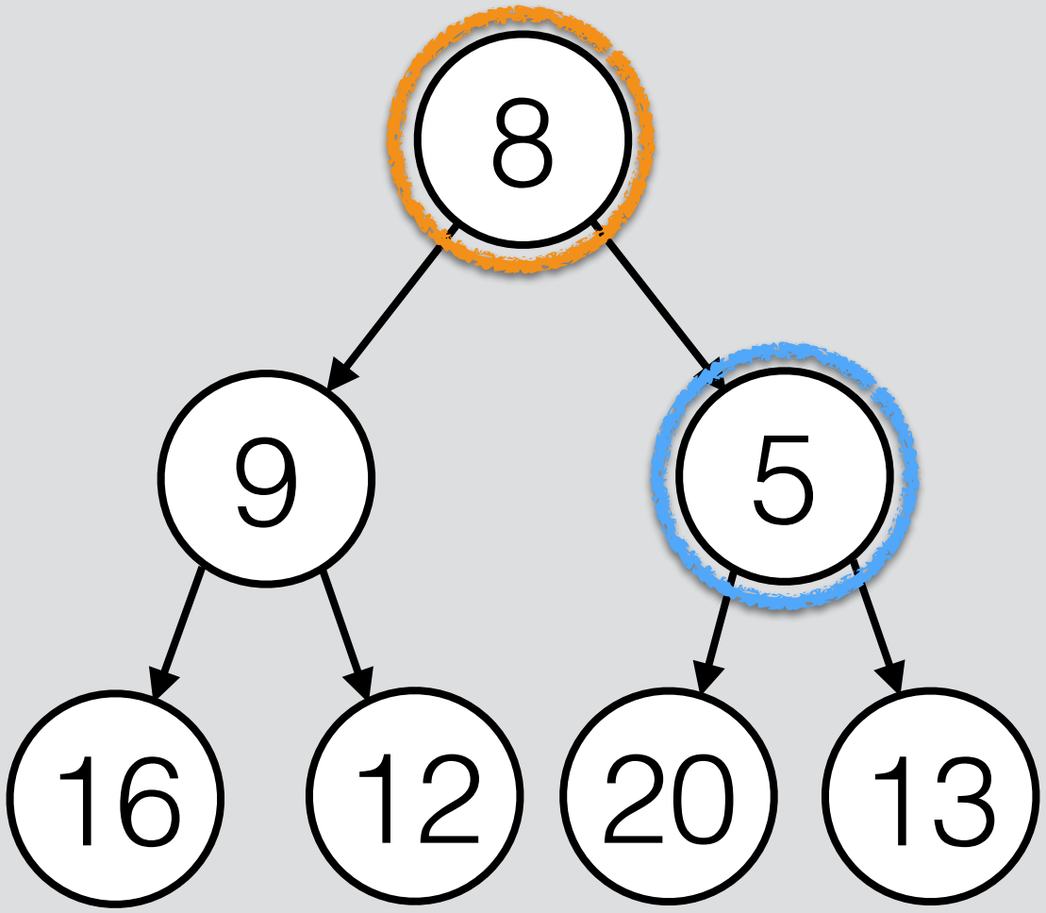



```
push(5);
```

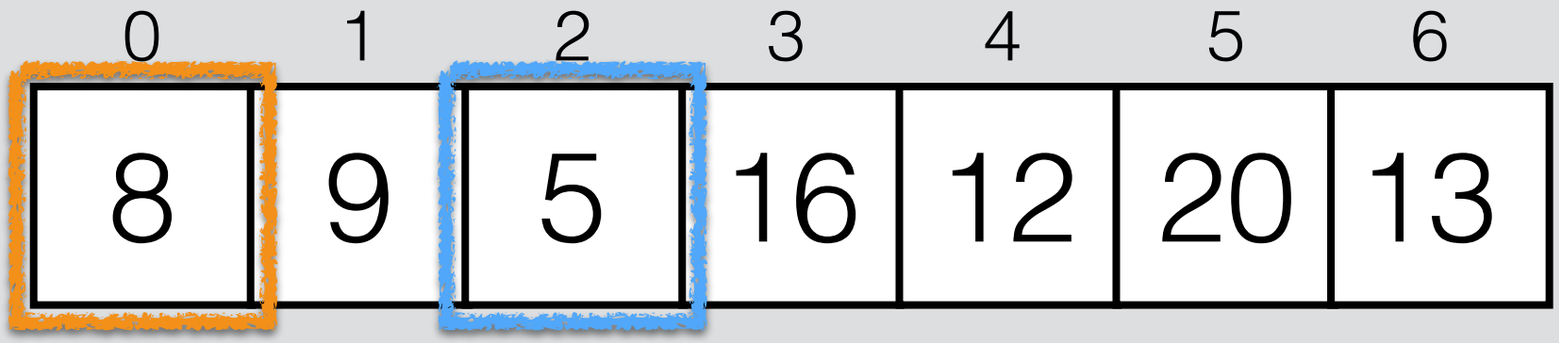


Parent: p
Children: 2p+1, 2p+2

Parent: (c-1)/2
Child: c

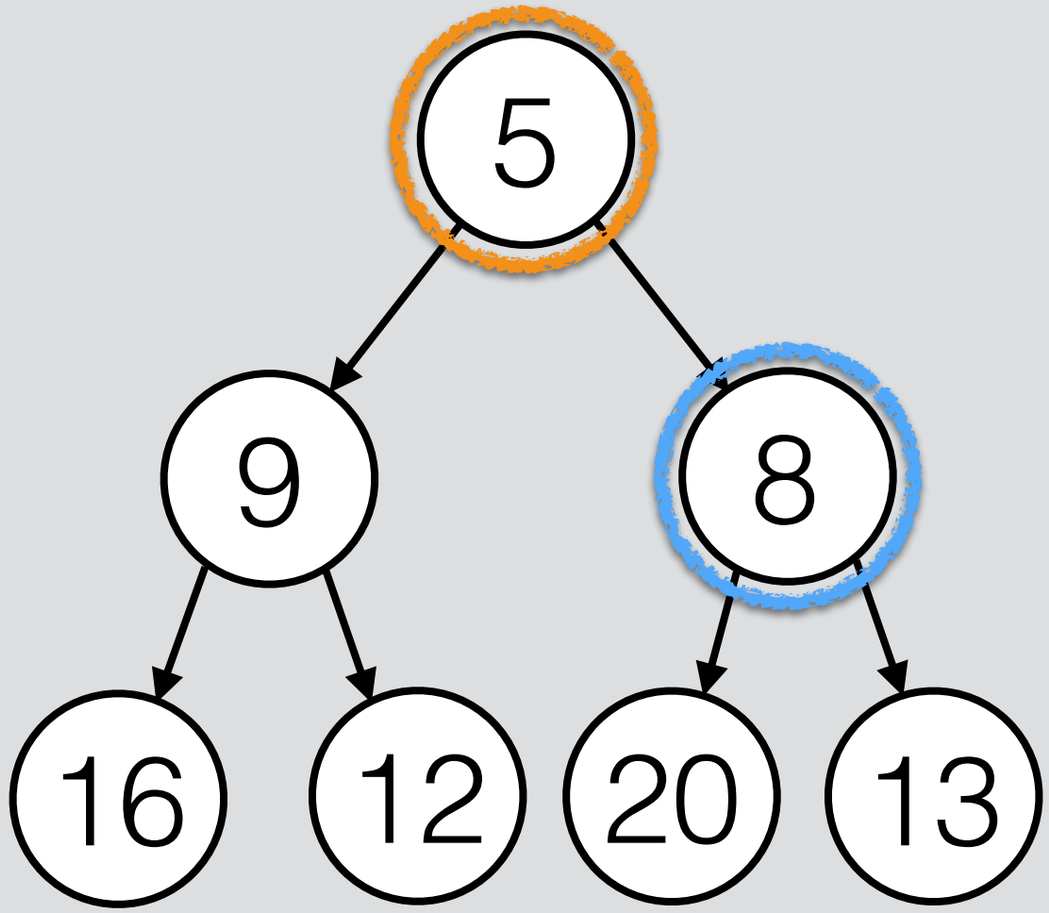



```
push(5);
```

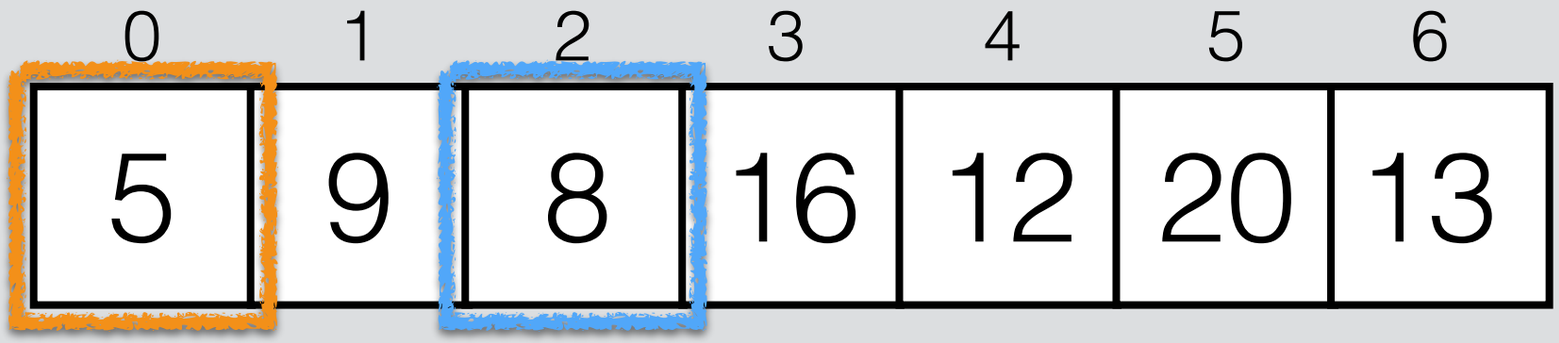


Parent: p
Children: 2p+1, 2p+2

Parent: (c-1)/2
Child: c

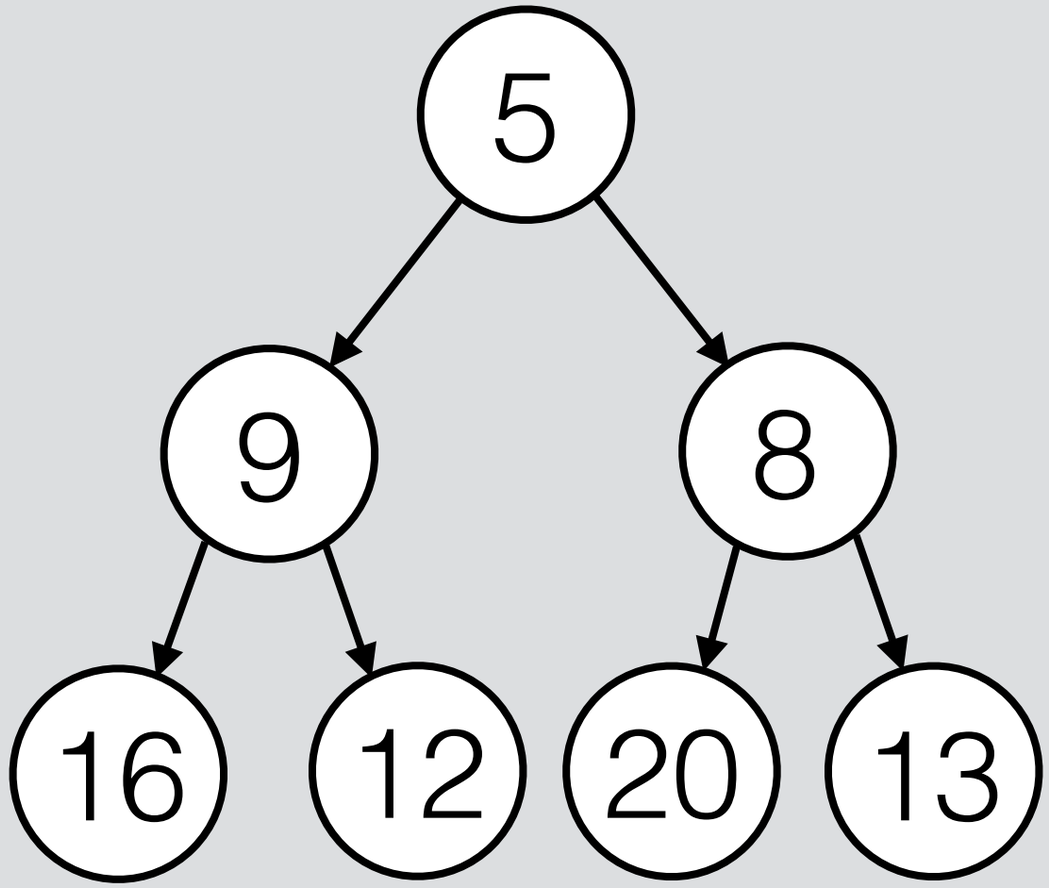



```
push(5);
```



Parent: p
Children: 2p+1, 2p+2

Parent: (c-1)/2
Child: c

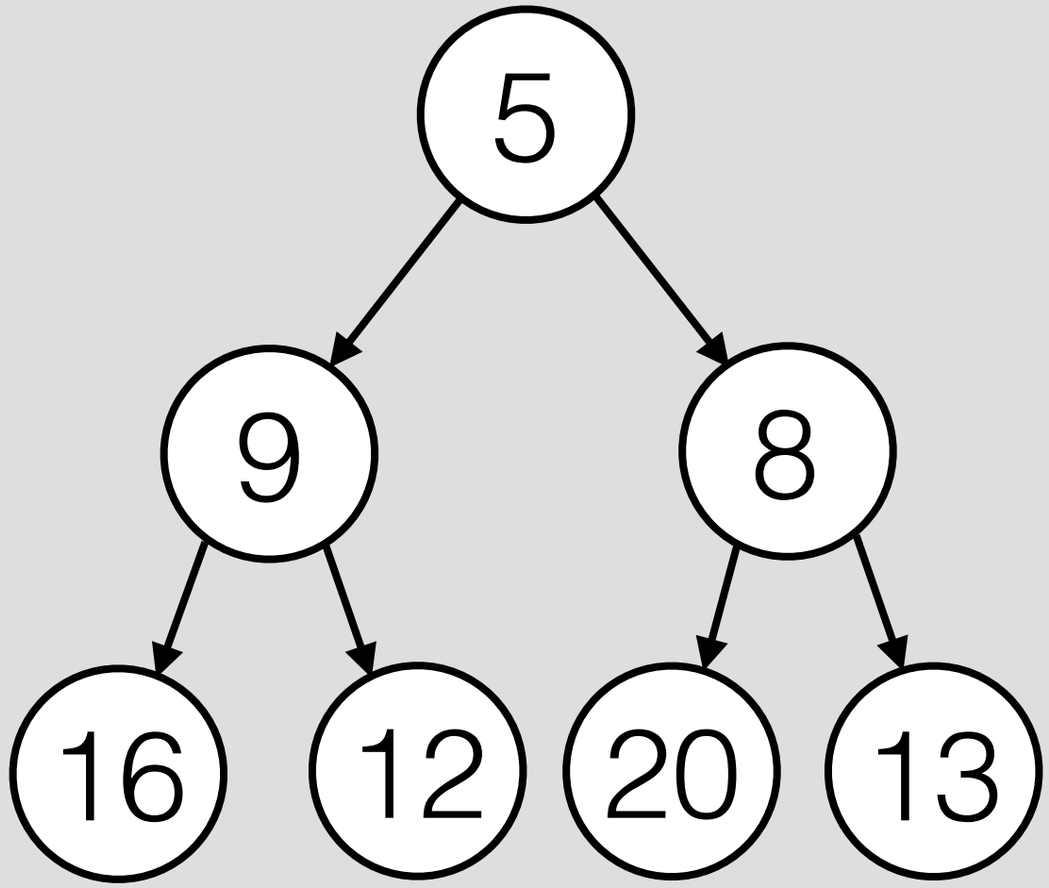


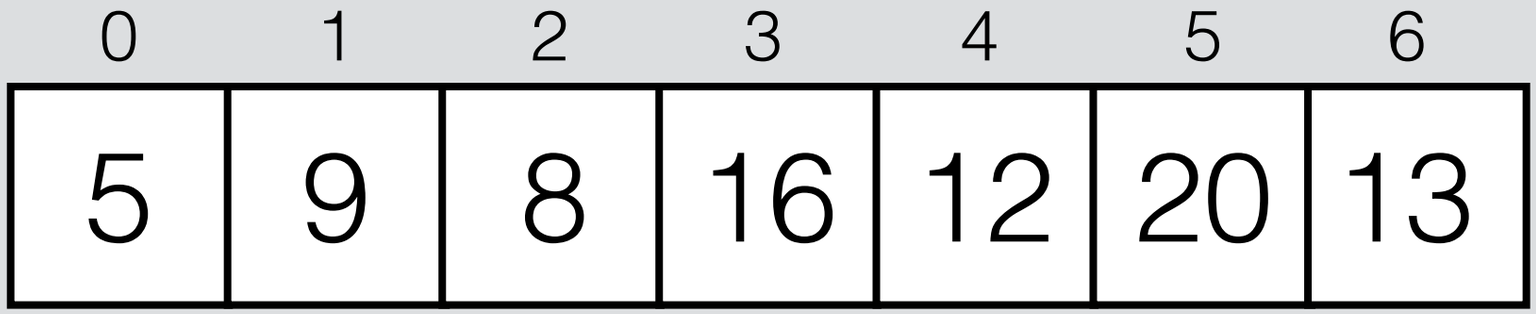

```
push(5);
```

0	1	2	3	4	5	6
5	9	8	16	12	20	13

Parent: p
Children: 2p+1, 2p+2

Parent: (c-1)/2
Child: c





Parent: p
Children: 2p+1, 2p+2

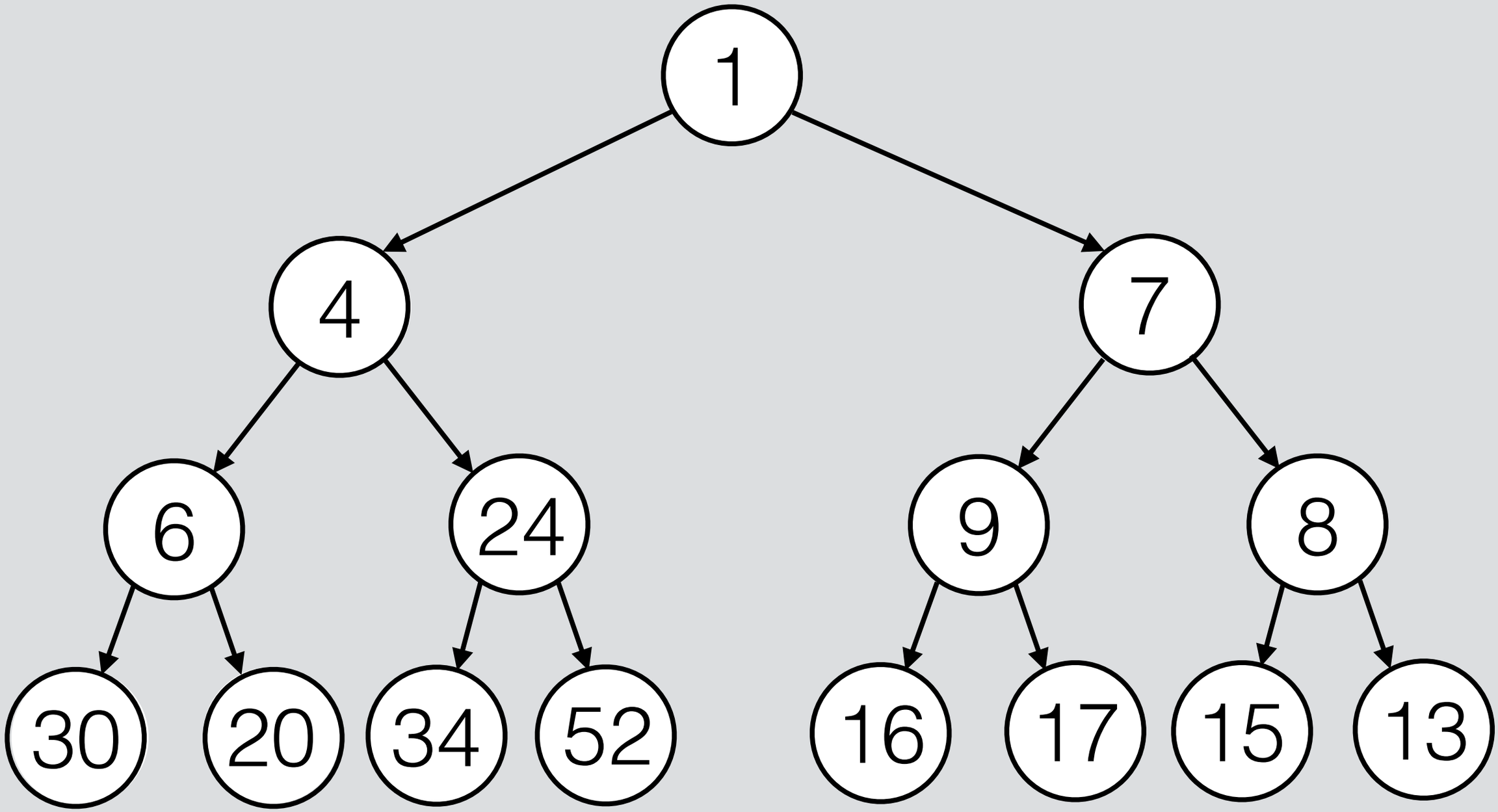
Parent: (c-1)/2
Child: c

decrease_key

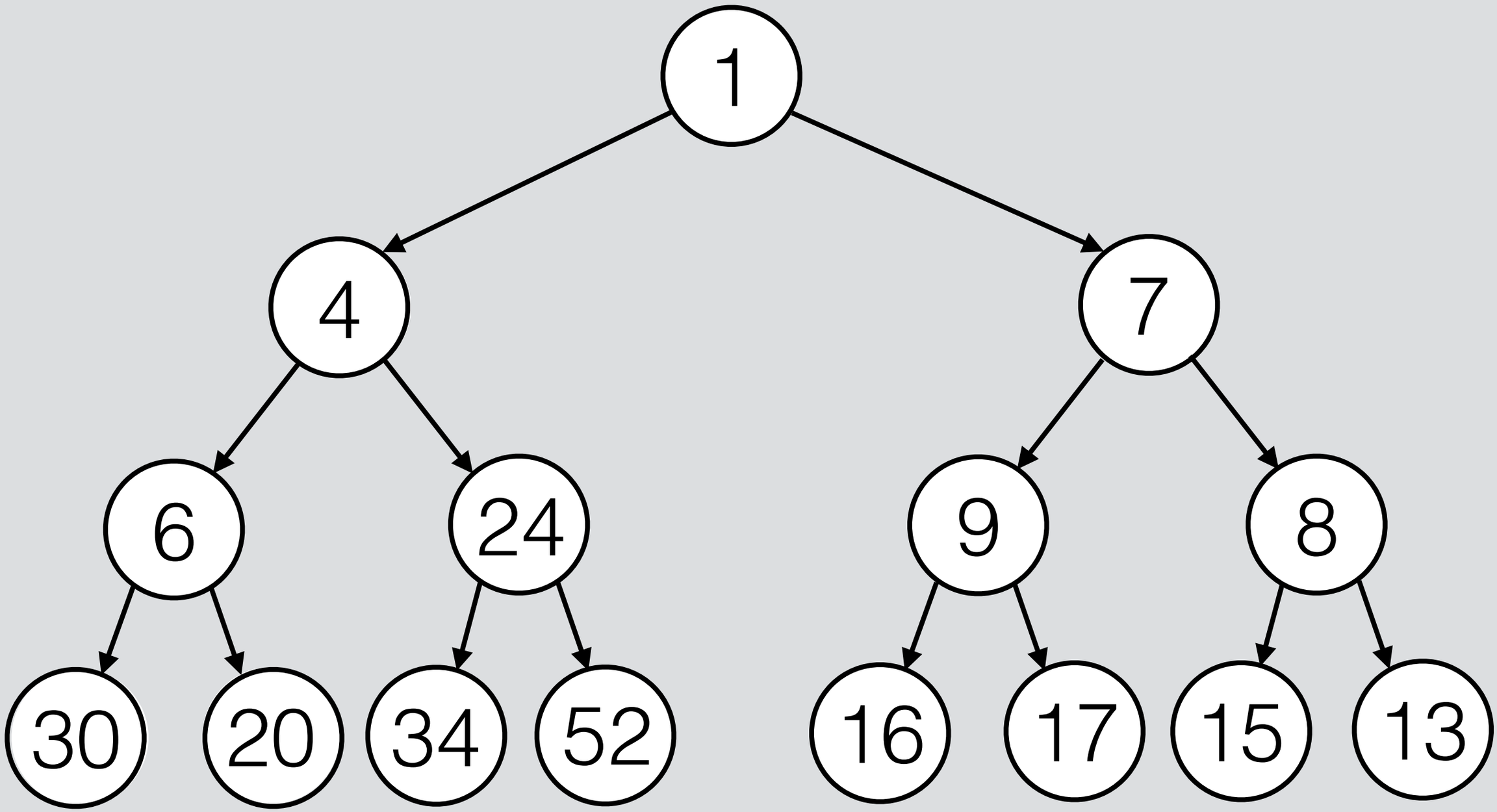
Min Priority Queue ADT

supports <

- **Data:** a sequence of (value, priority) pairs.
- **Operation (push):** Add new (value, priority) pair to queue.
- **Operation (pop):** Remove smallest-priority element.
- **Operation (front):** Return smallest-priority element.
- **Operation (size):** Return the # elements in the queue.
- **Operation (decrease_key):** Reduce priority of a value.

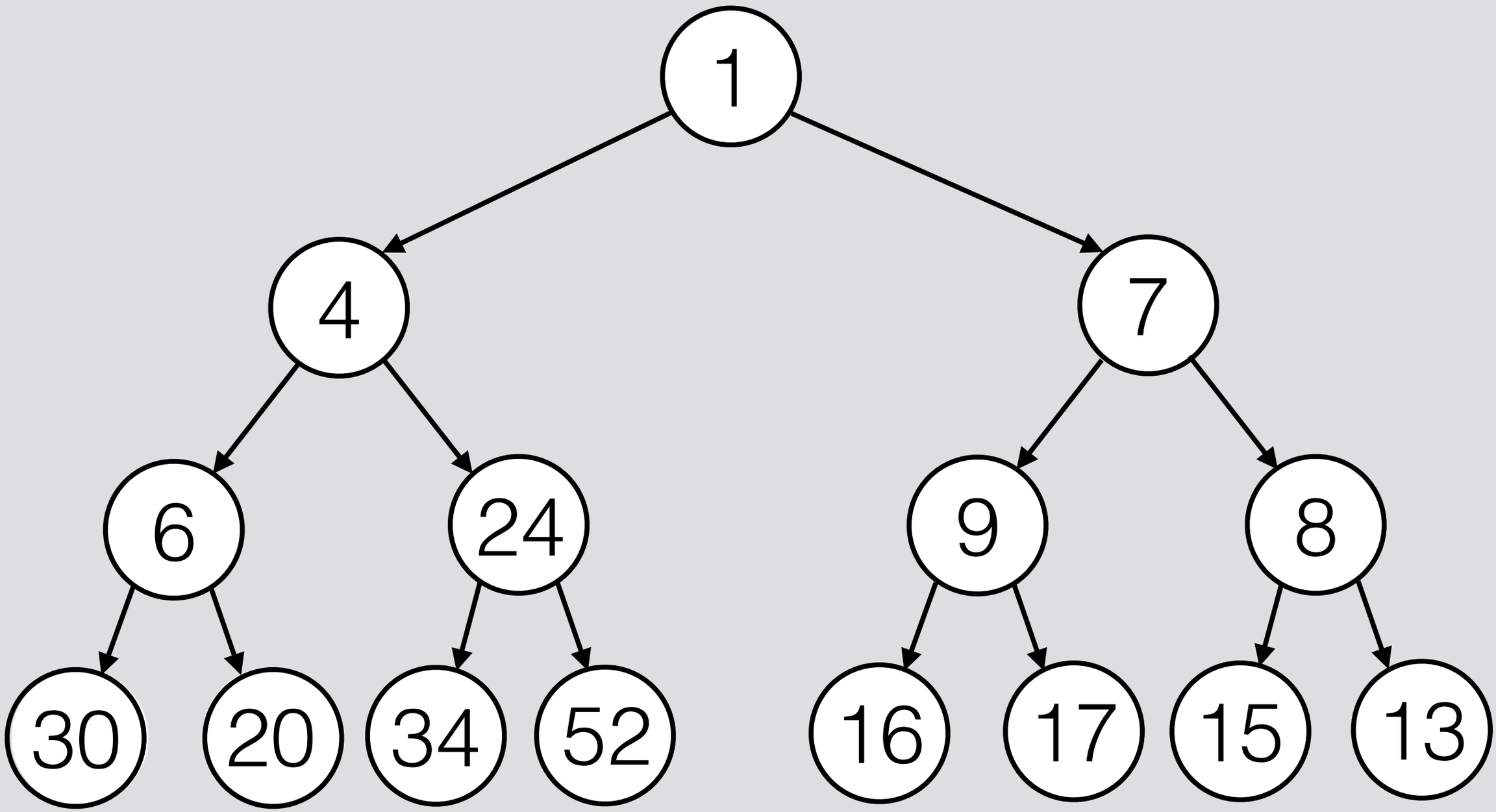

1
4
7
6
24
9
8
30
20
34
52
16
17
15
13

decrease_key(52, 3);



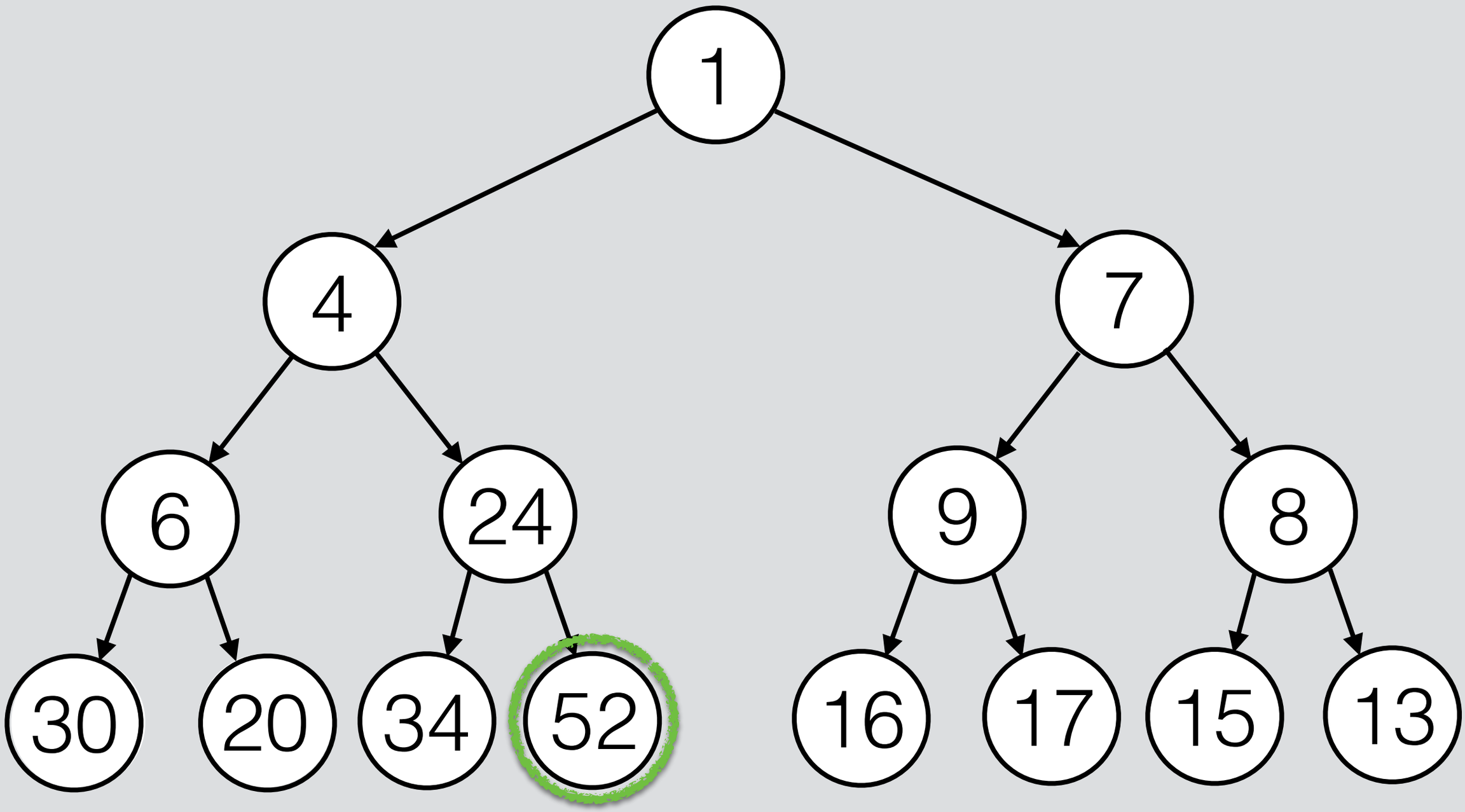
1. Find value.

```
decrease_key(52, 3);
```



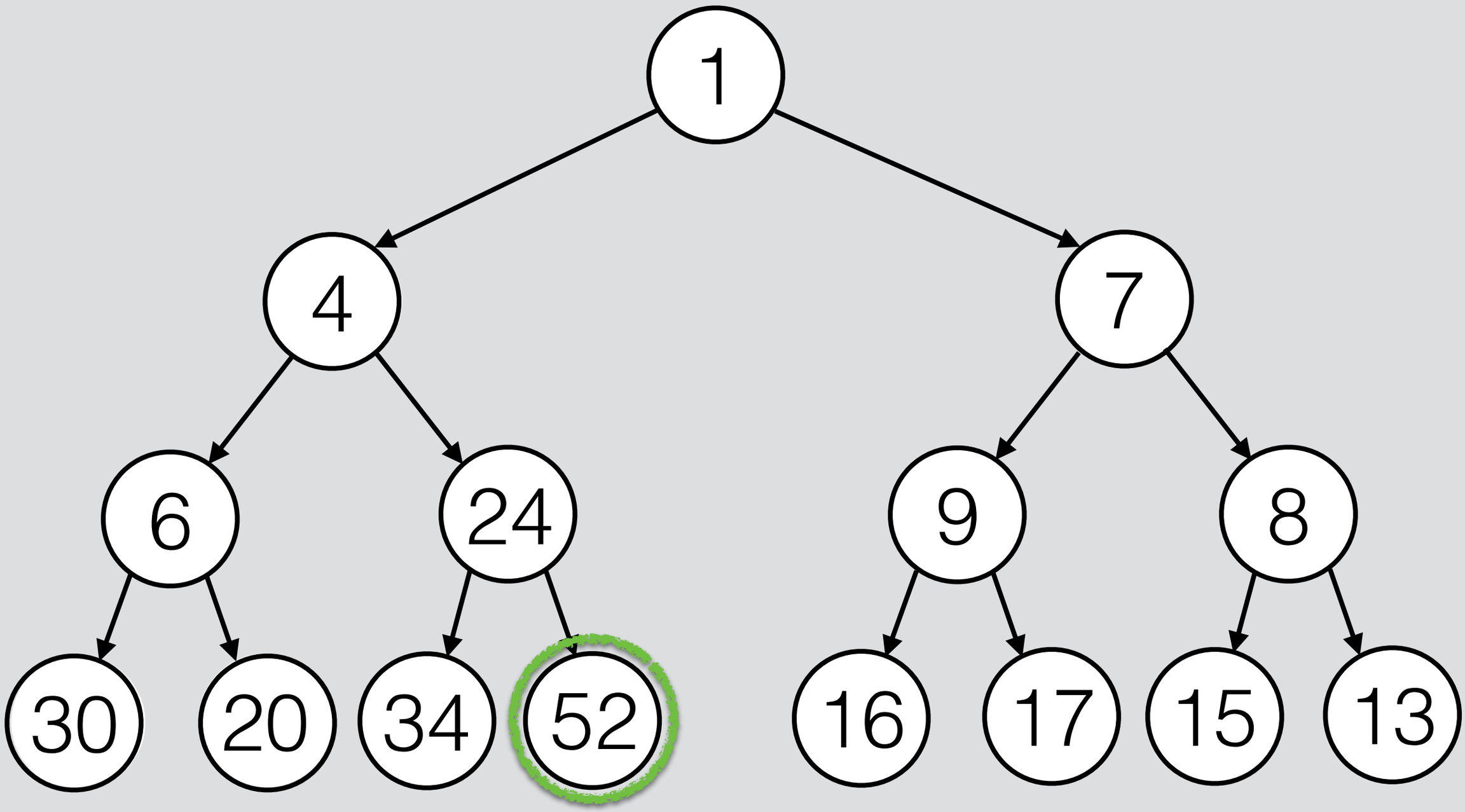
1. Find value.

```
decrease_key(52, 3);
```



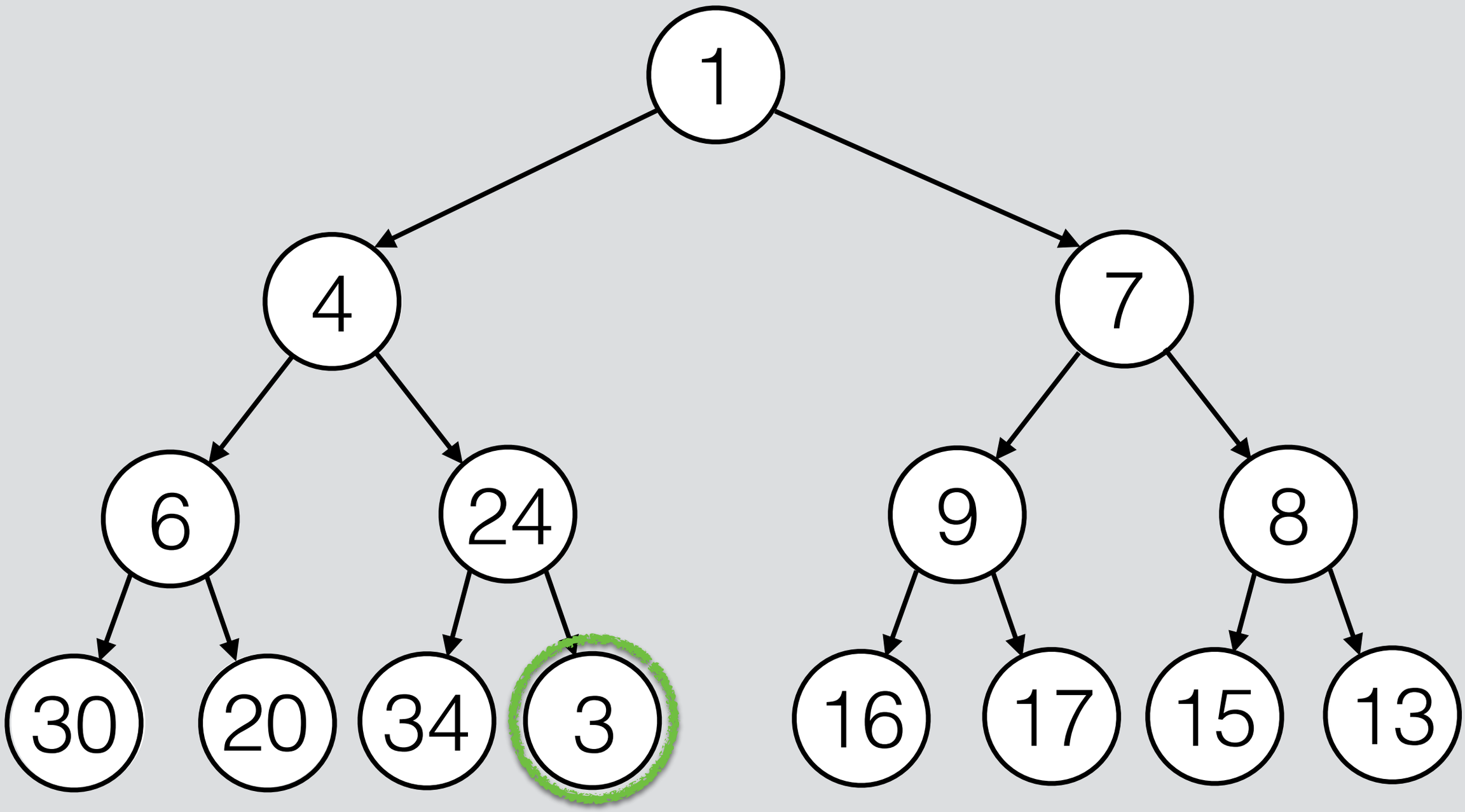
1. Find value.
2. Decrease priority.

```
decrease_key(52, 3);
```



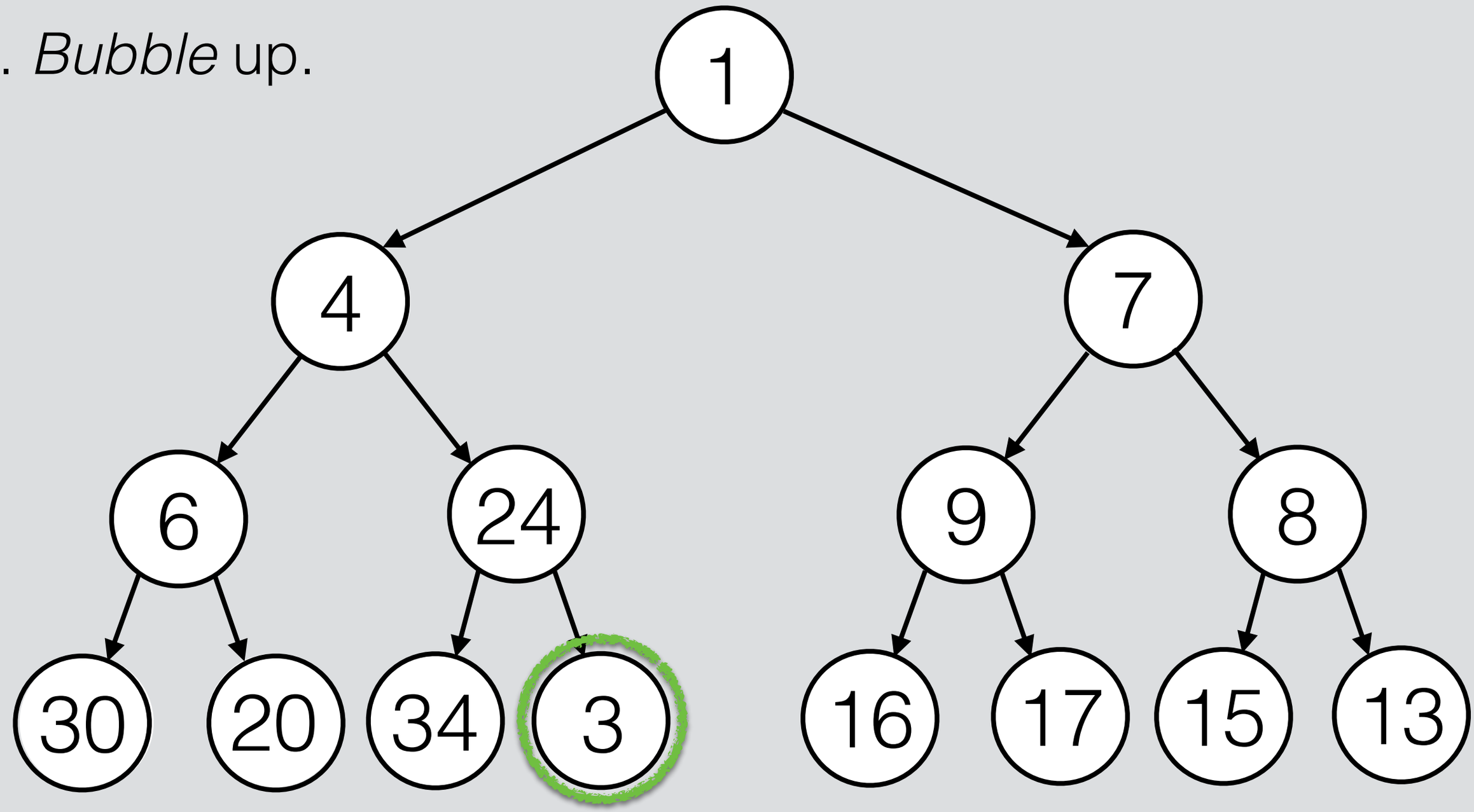
1. Find value.
2. Decrease priority.

```
decrease_key(52, 3);
```



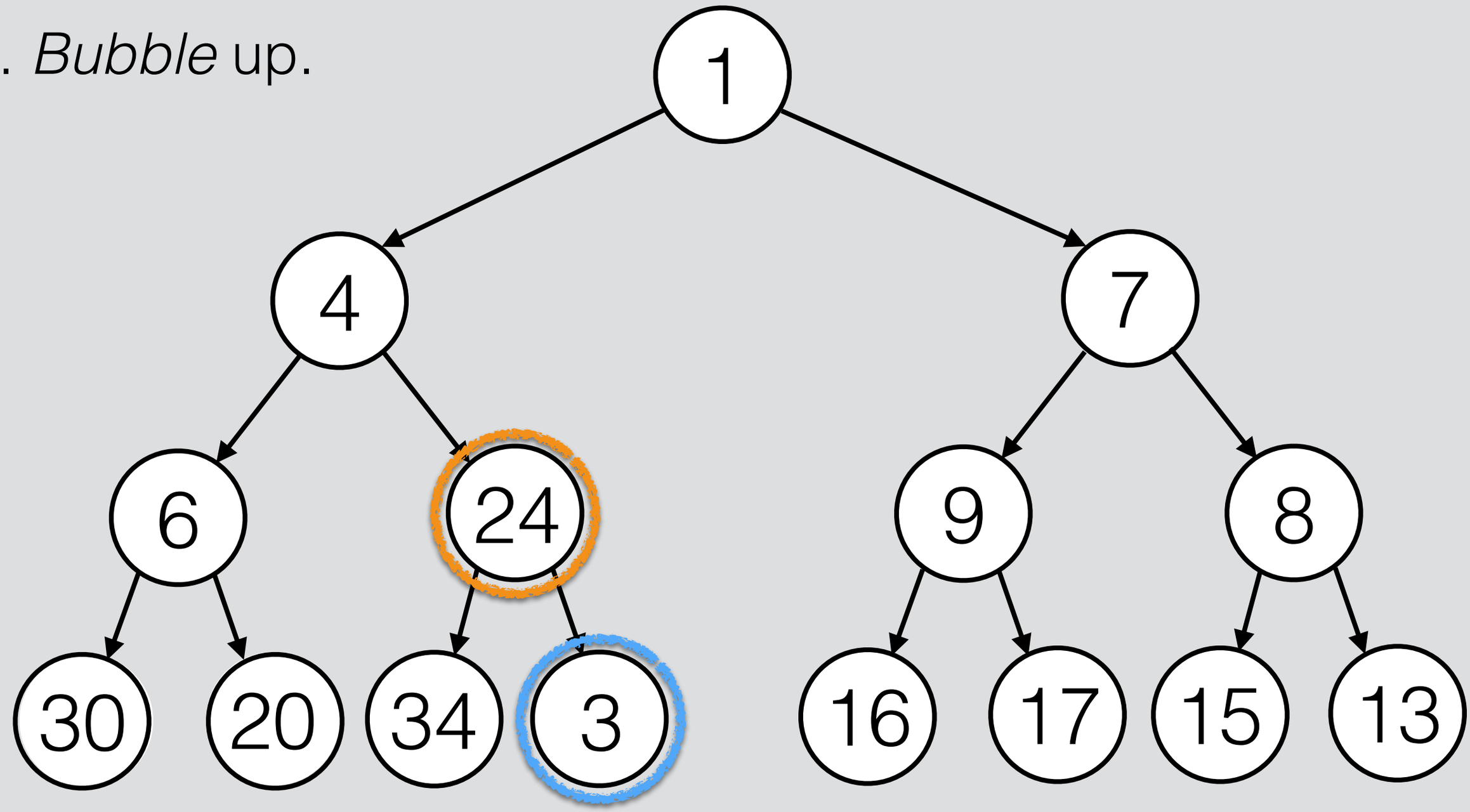
1. Find value.
2. Decrease priority.
3. *Bubble* up.

```
decrease_key(52, 3);
```



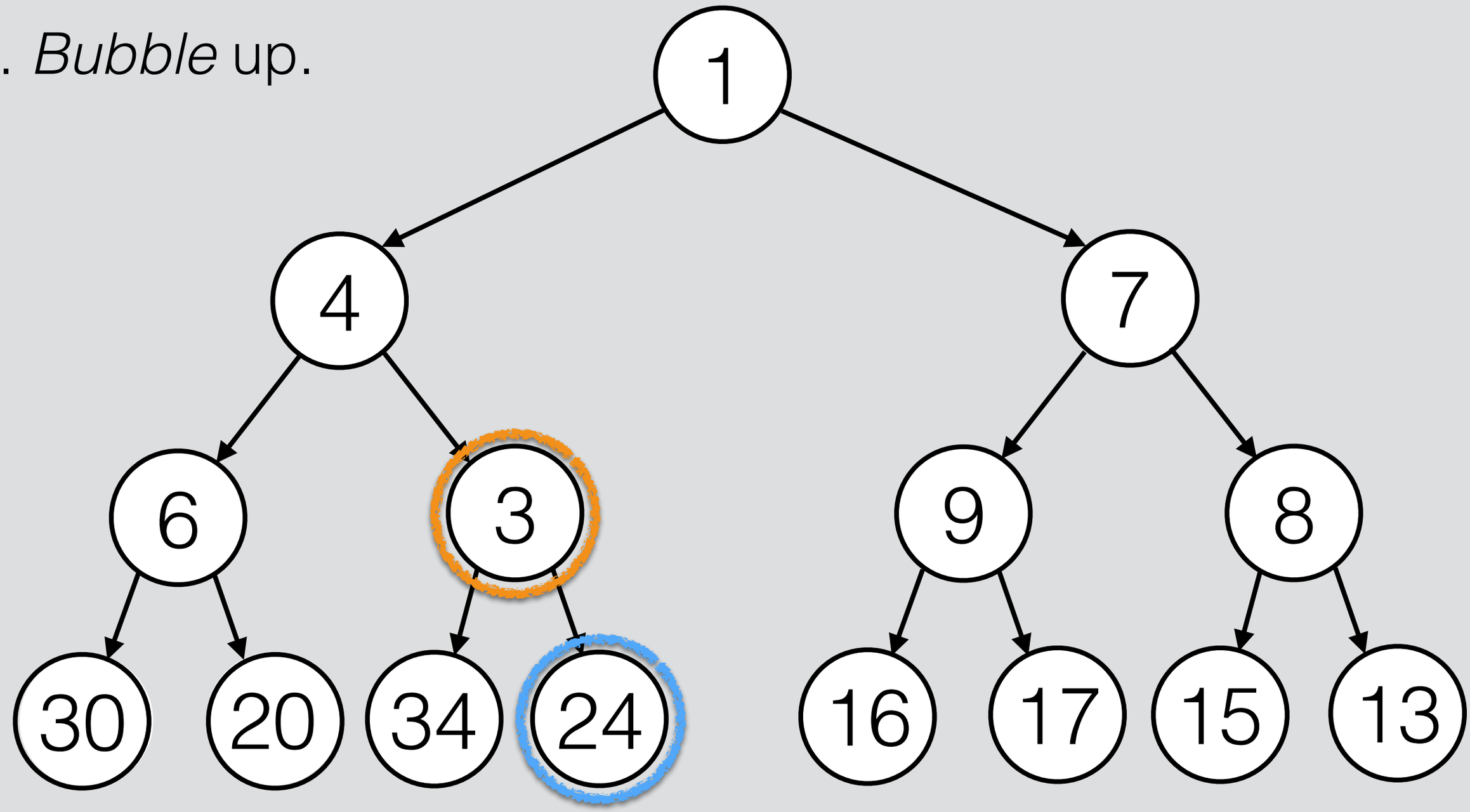
1. Find value.
2. Decrease priority.
3. *Bubble* up.

```
decrease_key(52, 3);
```



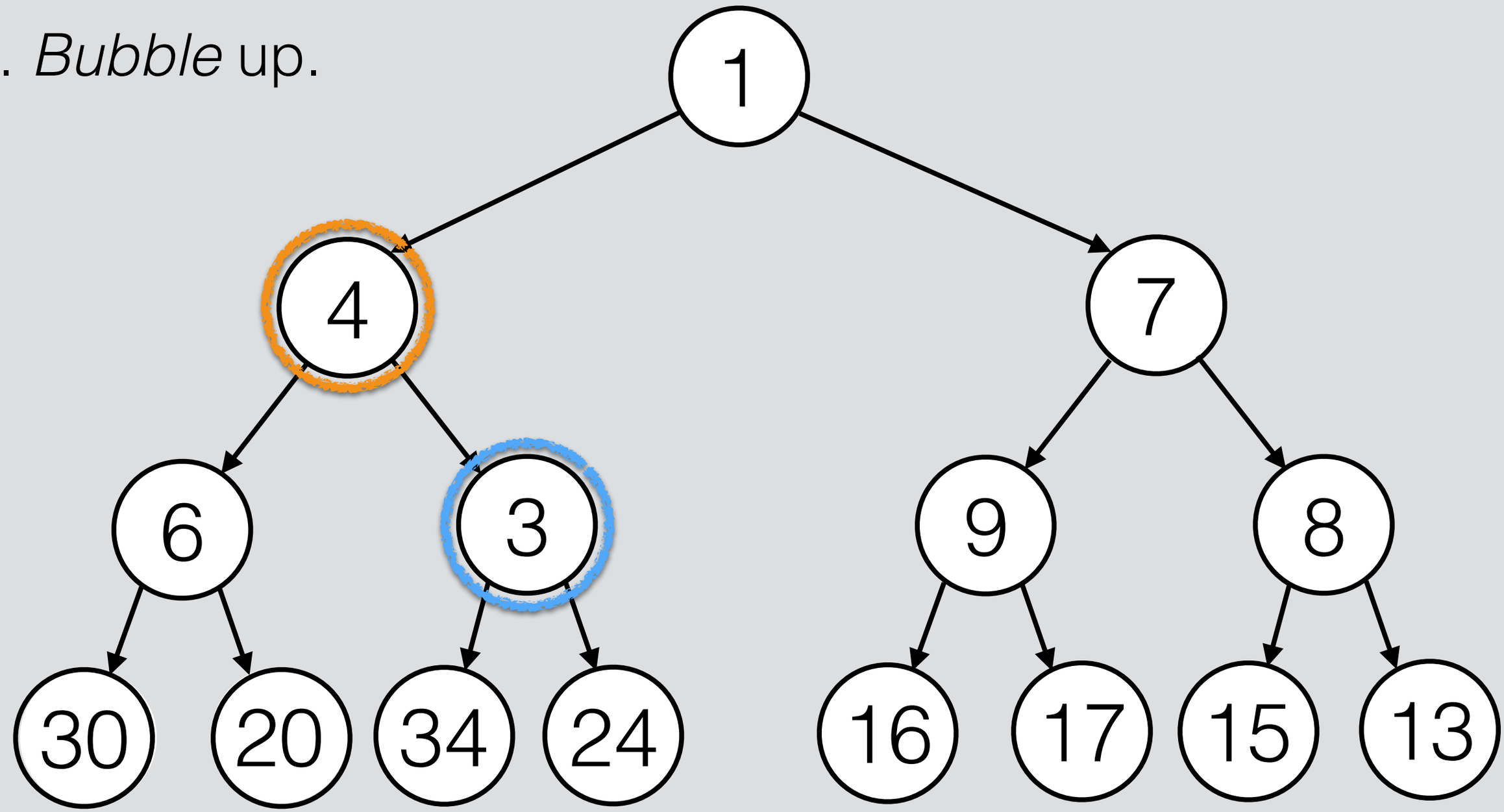
1. Find value.
2. Decrease priority.
3. *Bubble* up.

```
decrease_key(52, 3);
```



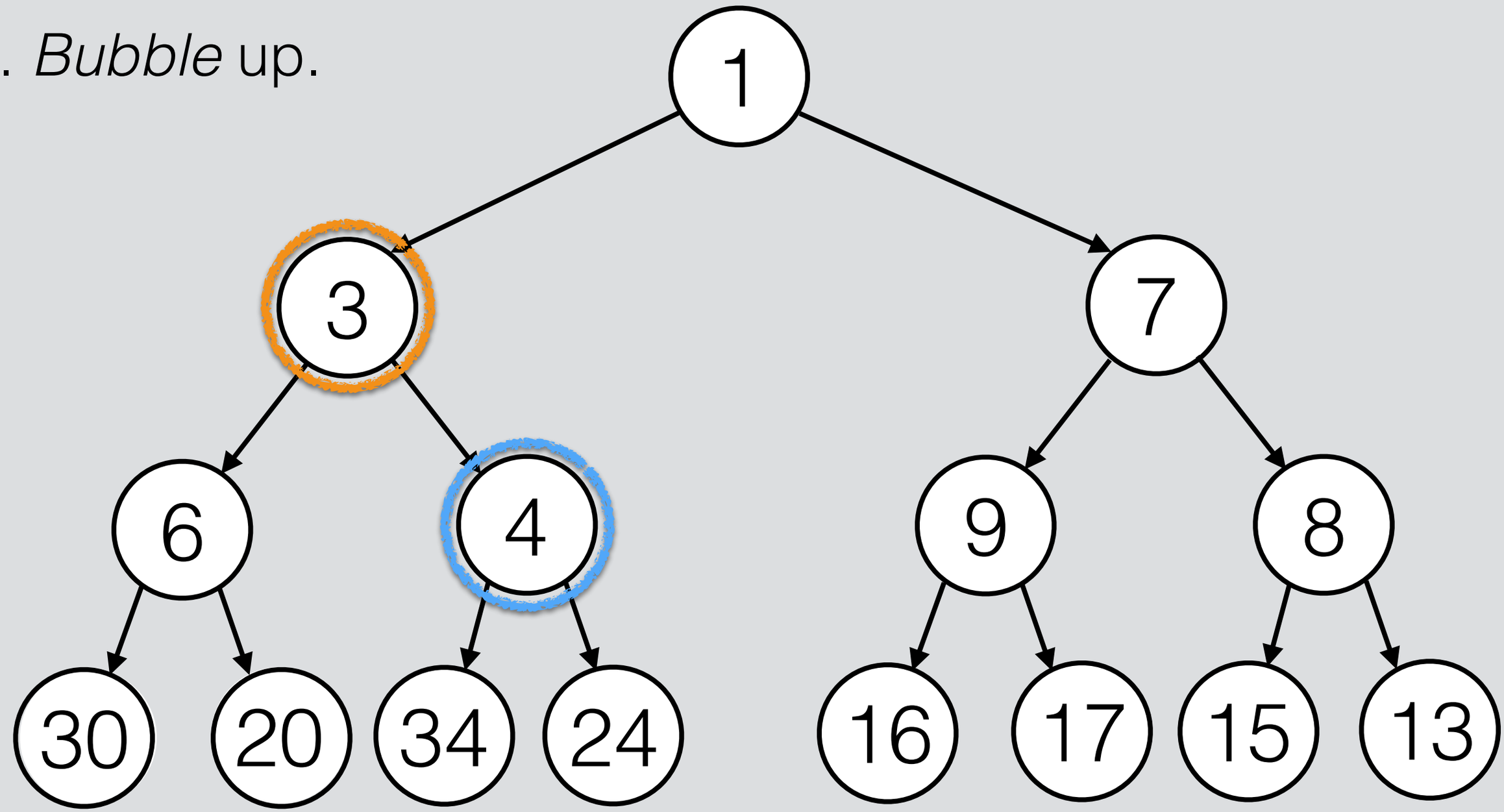
1. Find value.
2. Decrease priority.
3. *Bubble* up.

```
decrease_key(52, 3);
```



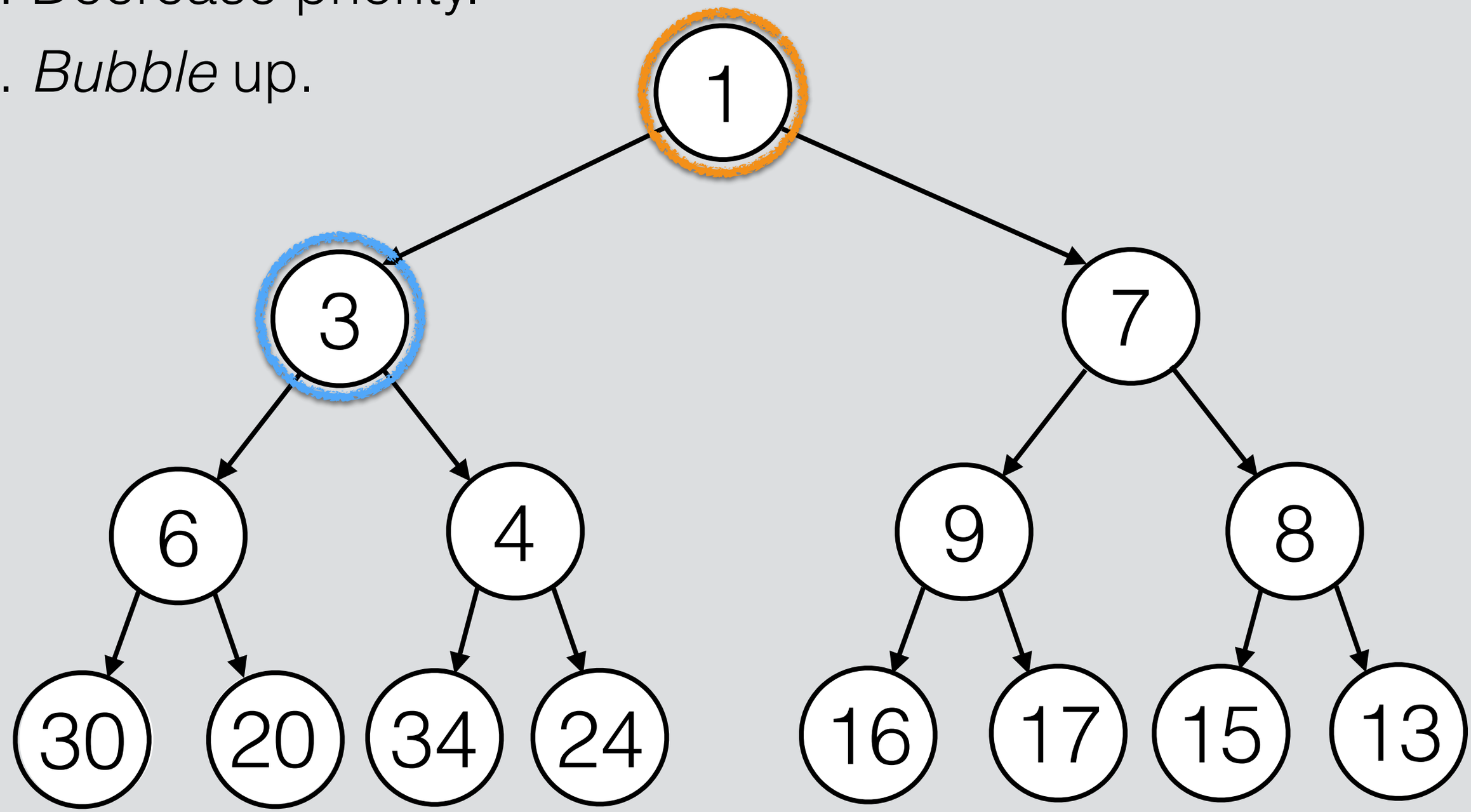
1. Find value.
2. Decrease priority.
3. *Bubble* up.

```
decrease_key(52, 3);
```



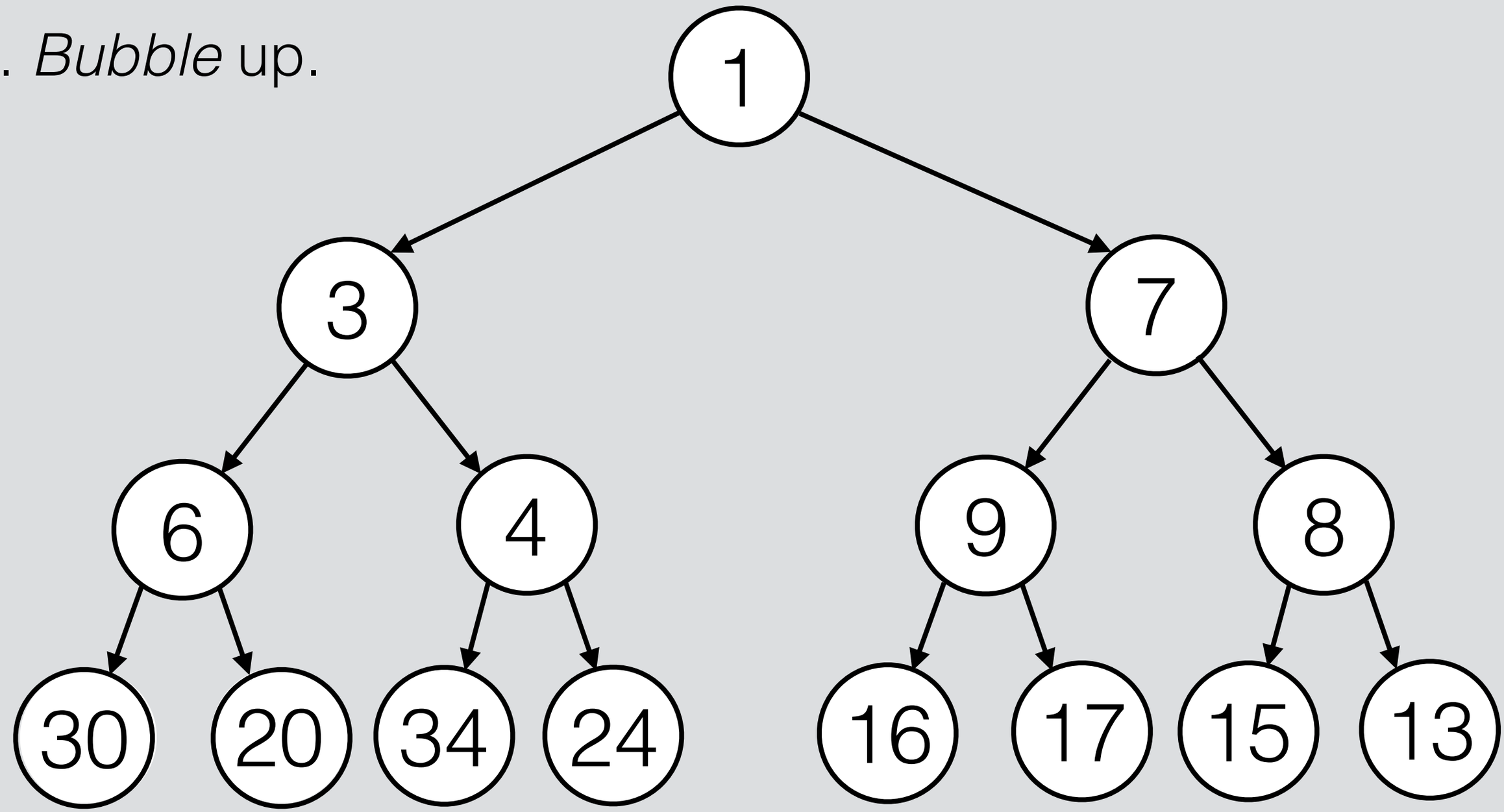
1. Find value.
2. Decrease priority.
3. *Bubble* up.

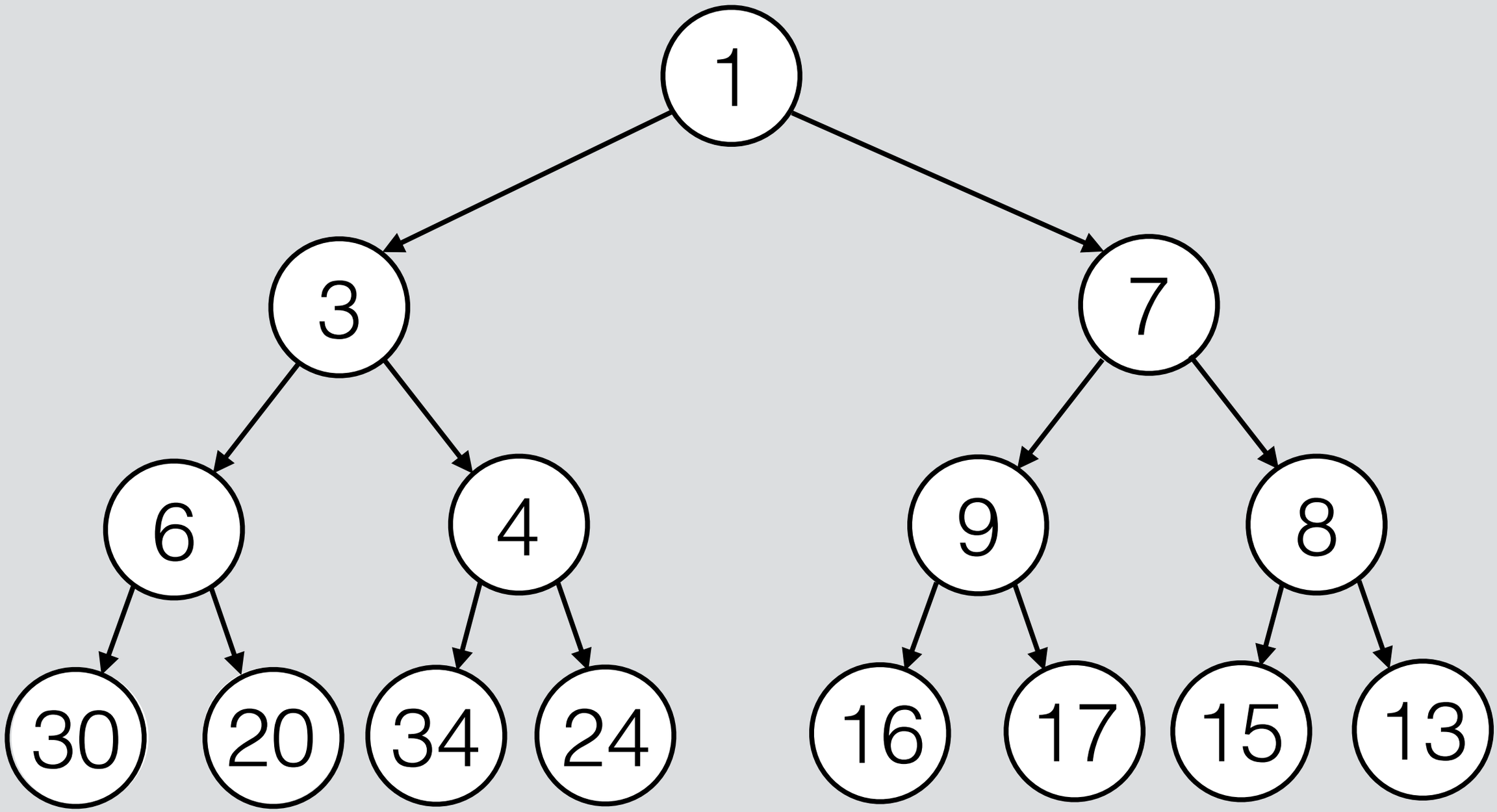
```
decrease_key(52, 3);
```



1. Find value.
2. Decrease priority.
3. *Bubble* up.

```
decrease_key(52, 3);
```




1
3
7
6
4
9
8
30
20
34
24
16
17
15
13

Min Priority Queue
Worst-Case Running Times

	Linked list:	AVL Tree:	Heap:
Operation (push):	Θ(n)	Θ(log(n))	Θ(log(n))
Operation (pop):	Θ(1)	Θ(log(n))	Θ(log(n))
Operation (front):	Θ(1)	Θ(1)	Θ(1)
Operation (size):	Θ(1)	Θ(1)	Θ(1)
Operation (decrease_key):	Θ(1)	Θ(log(n))	Θ(log(n))