CS2040S
Data Structures and Algorithms
(e-learning edition)

Priority Queues and Heaps

Priority Queue

Maintain a set of prioritized objects:

- insert: add a new object with a specified priority
- extractMin: remove and return the object with minimum priority

Ex: Scheduling

- Find next task to do
- Earliest deadline first

Task	Due date
CS4234 PS8	March 31
Study for Exam	April 4
Wash clothes	April 6
See friends	May 12

Abstract Data Type

Priority Queue

interface	**IPriorityQueue<Key, Priority>**	
void	insert(Key k, Priority p)	*insert k with priority p*
Data	extractMin()	*remove key with minimum priority*
void	decreaseKey(Key k, Priority p)	*reduce the priority of key k to priority p*
boolean	contains(Key k)	*does the priority queue contain key k?*
boolean	isEmpty()	*is the priority queue empty?*

Notes:

Assume data items are unique.

Abstract Data Type

Max Priority Queue

interface	**IMaxPriorityQueue<Key, Priority>**	
void	insert(Key k, Priority p)	*insert k with priority p*
Data	extractMax()	*remove key with maximum priority*
void	increaseKey(Key k, Priority p)	*increase the priority of key k to priority p*
boolean	contains(Key k)	*does the priority queue contain key k?*
boolean	isEmpty()	*is the priority queue empty?*

Notes:

Assume data items are unique.

Priority Queue

Sorted array

- insert: O(n)
  - Find insertion location in array.
  - Move everything over.
- extractMax: O(1)
  - Return largest element in array

object	G	C	Y	Z	B	D	F	J	L
priority	2	7	9	13	22	26	29	31	45

Priority Queue

Unsorted array

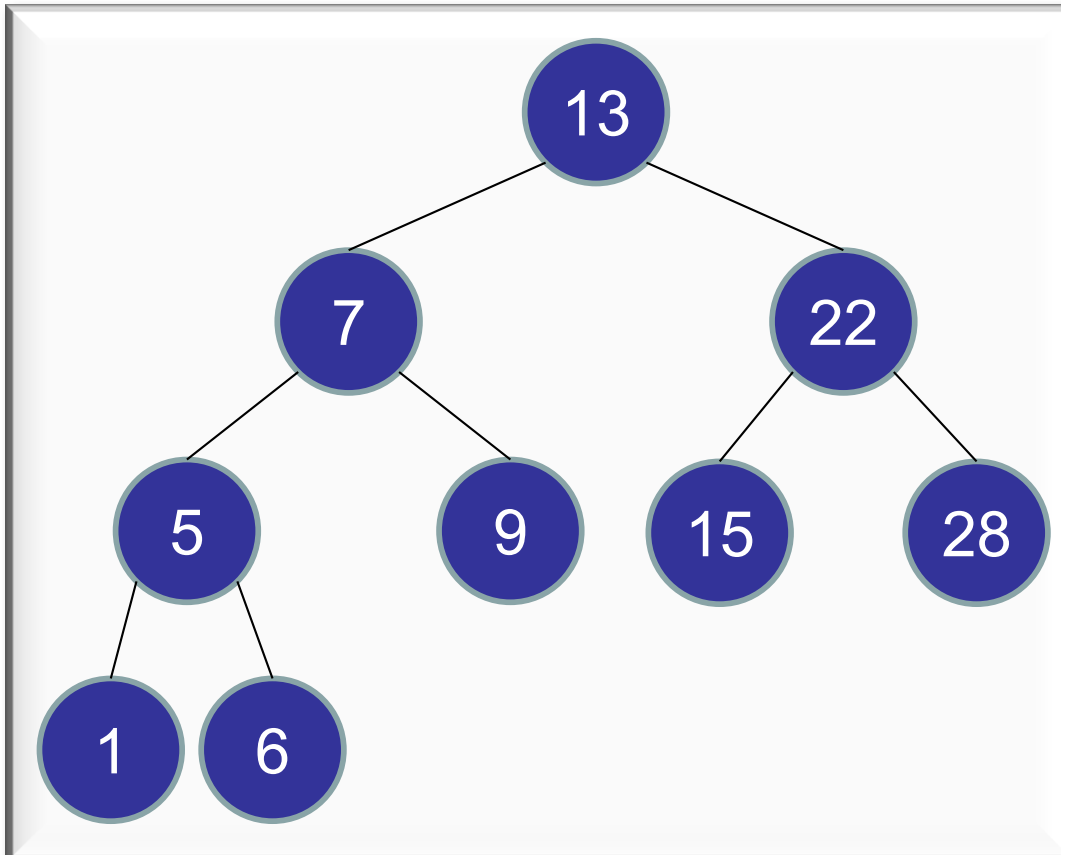
- insert: O(1)
  - Add object to end of list
- extractMax: O(n)
  - Search for largest element in array.
  - Remove and move everything over.

object	G	L	D	Z	B	J	F	C	Y
priority	2	45	26	13	22	31	29	7	9

Priority Queue

AVL Tree (indexed by priority)

- insert: O(log n)
  - Insert object in tree
- extractMax: O(log n)
  - Find maximum item.
  - Delete it from tree.



Priority Queue

Other operations:

- contains:
  - Look up key in hash table.
- decreaseKey:
  - Look up key in hash table.
  - Remove object from array/tree.
  - Re-insert object into array/tree.

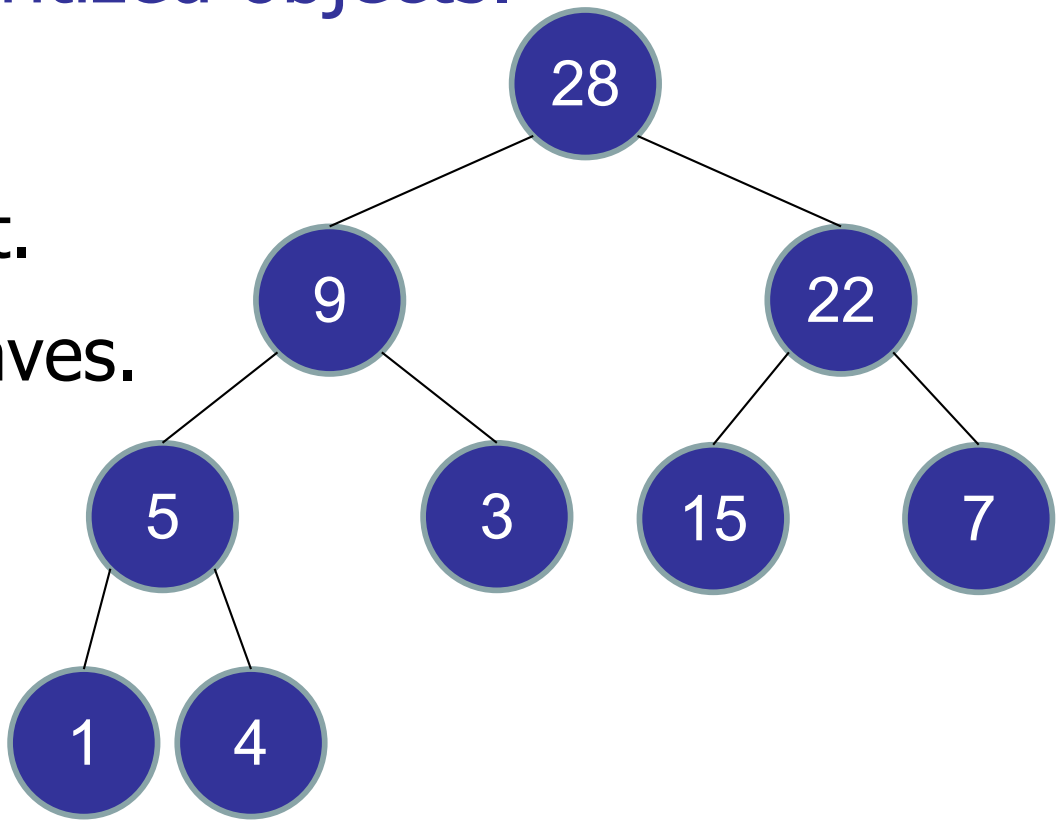
Hash table:

- Maps priorities to array slots or nodes in tree.

Heap

(aka Binary Heap or MaxHeap)

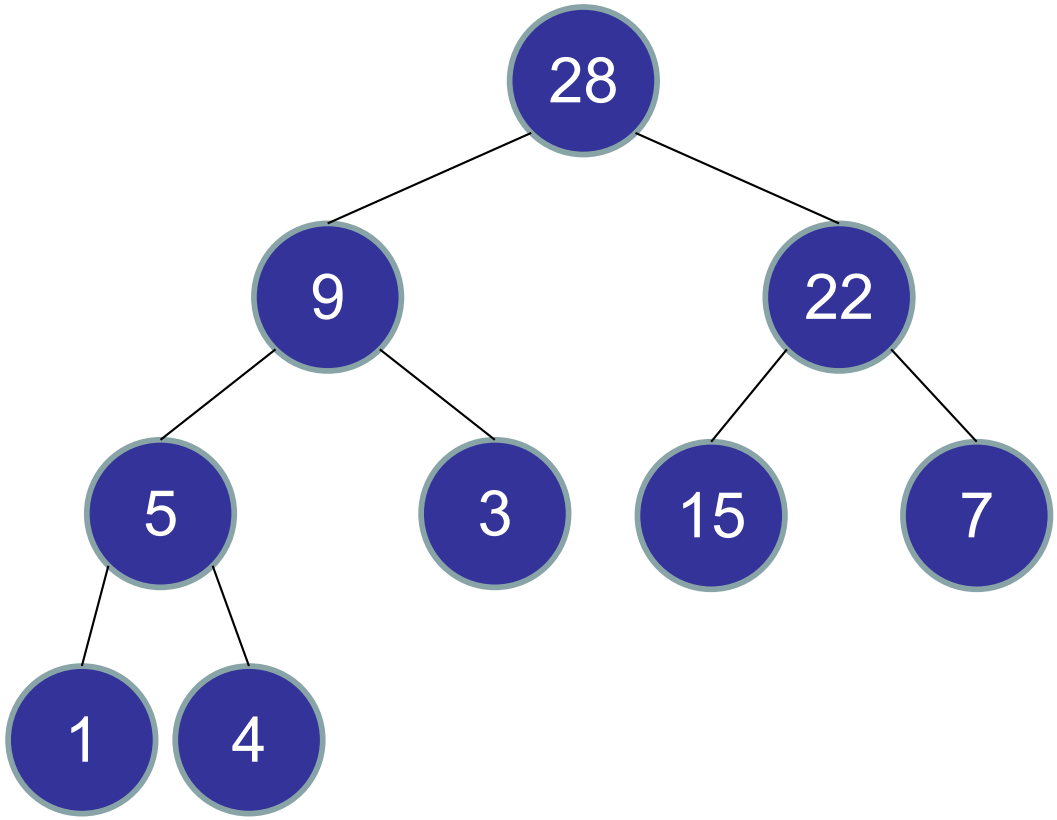
- Implements a Max Priority Queue
- Maintain a set of prioritized objects.
- Store items in a tree.
  - Biggest items at root.
  - Smallest items at leaves.



Two Properties of a Heap

1. Heap Ordering

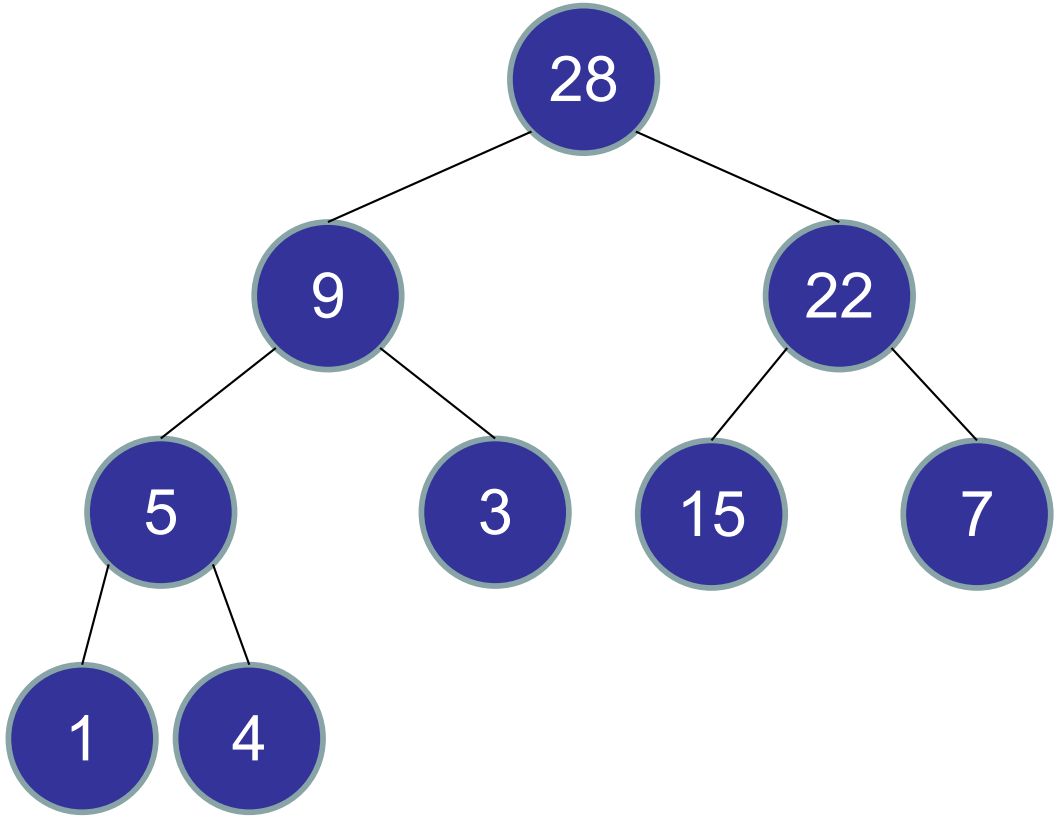
```
priority[parent] >= priority[child]
```



Note: not a binary search tree.

Two Properties of a Heap

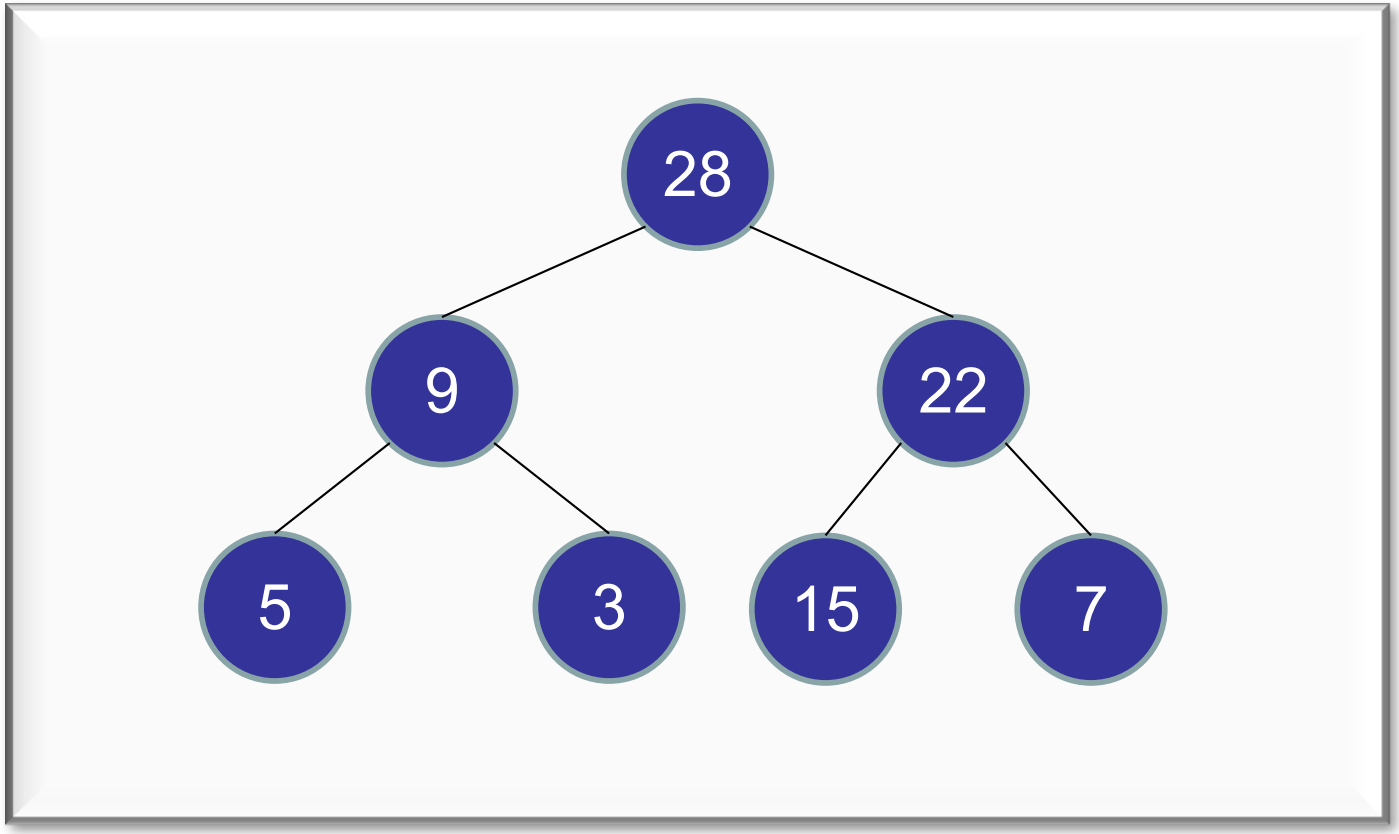
2. Complete binary tree
   - Every level is full, except possibly the last.
   - All nodes are as far left as possible.



Is it a heap?

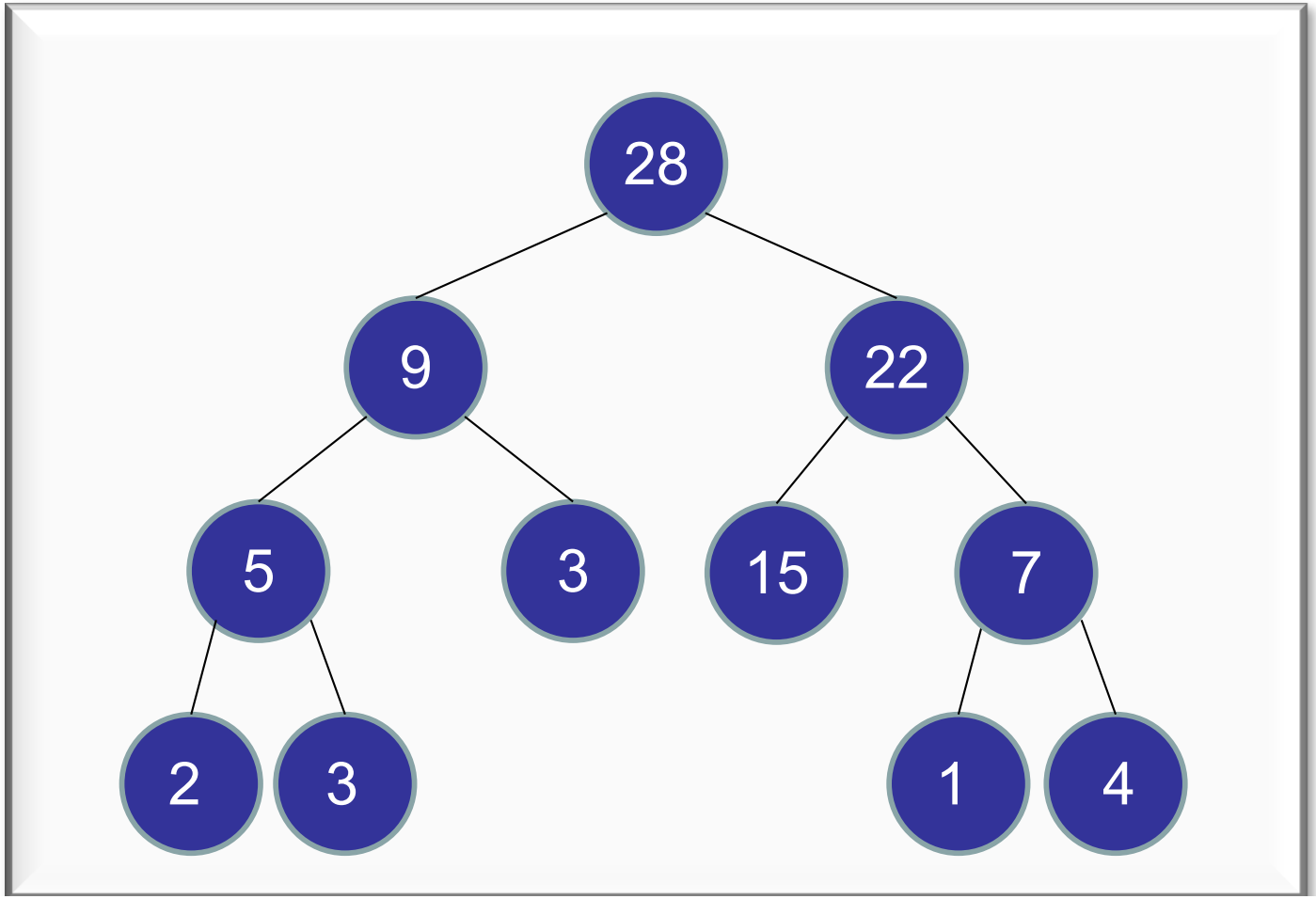
✓1. Yes

2. No.



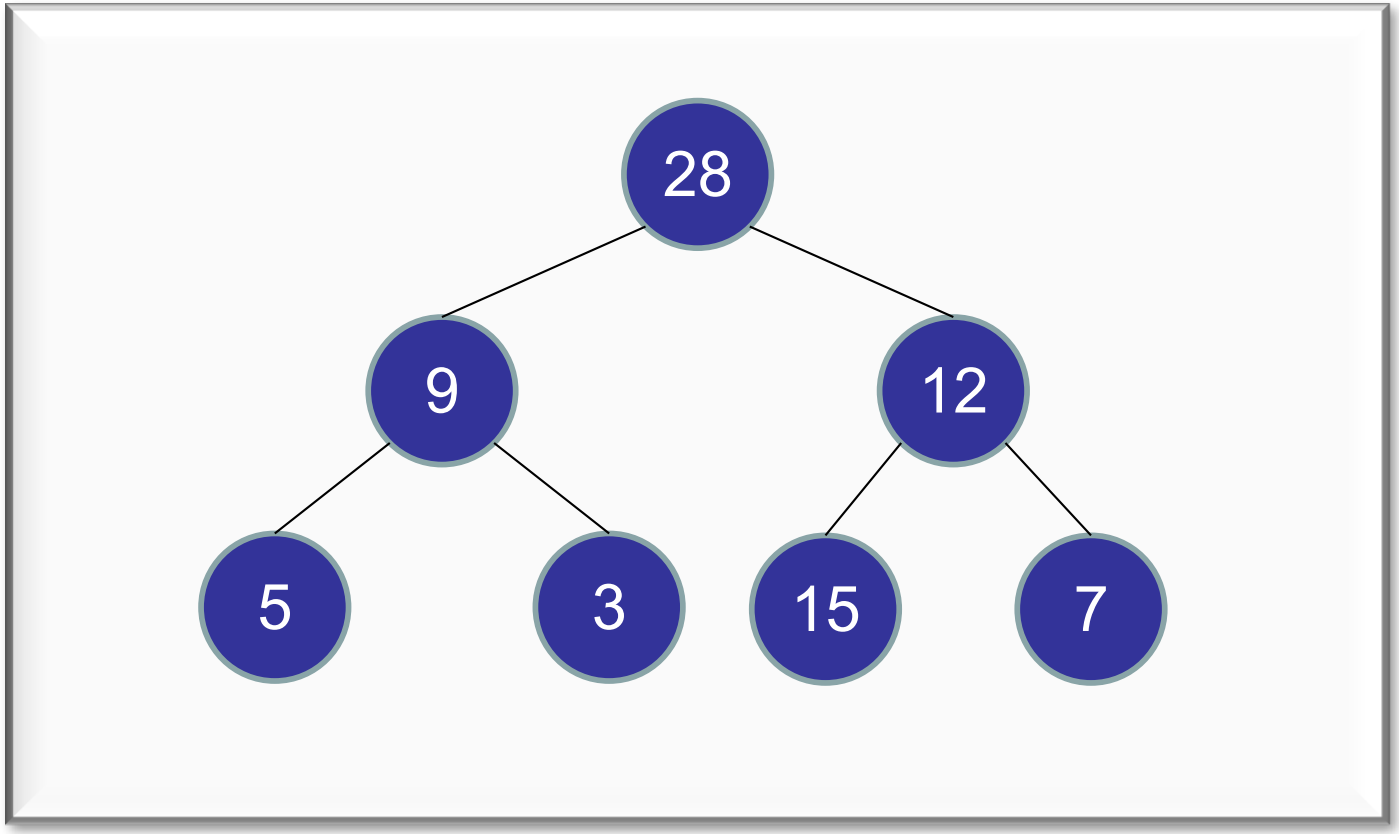
Is it a heap?

1. Yes
2. ✓ No.



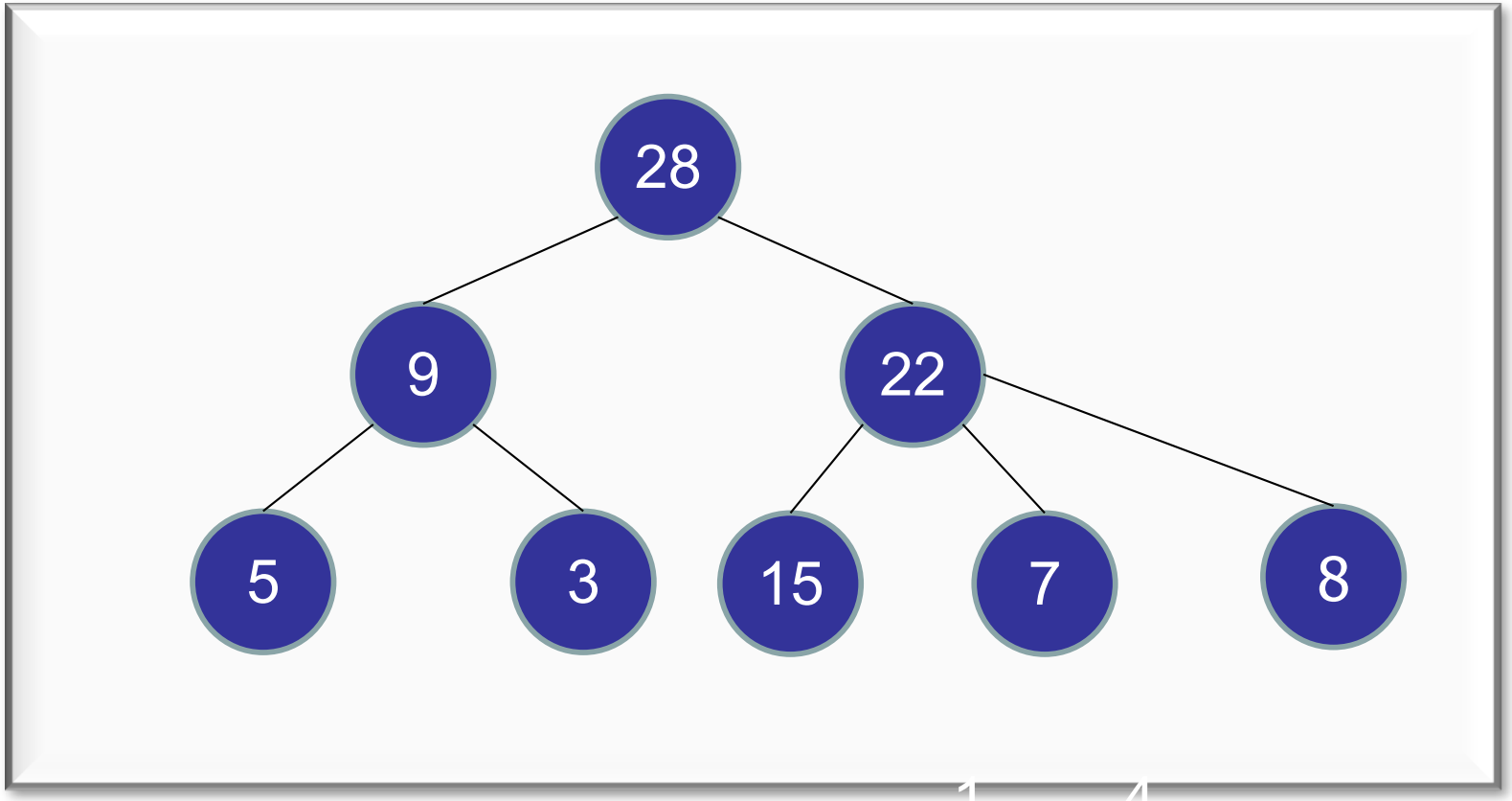
Is it a heap?

1. Yes
2. ✓ No.



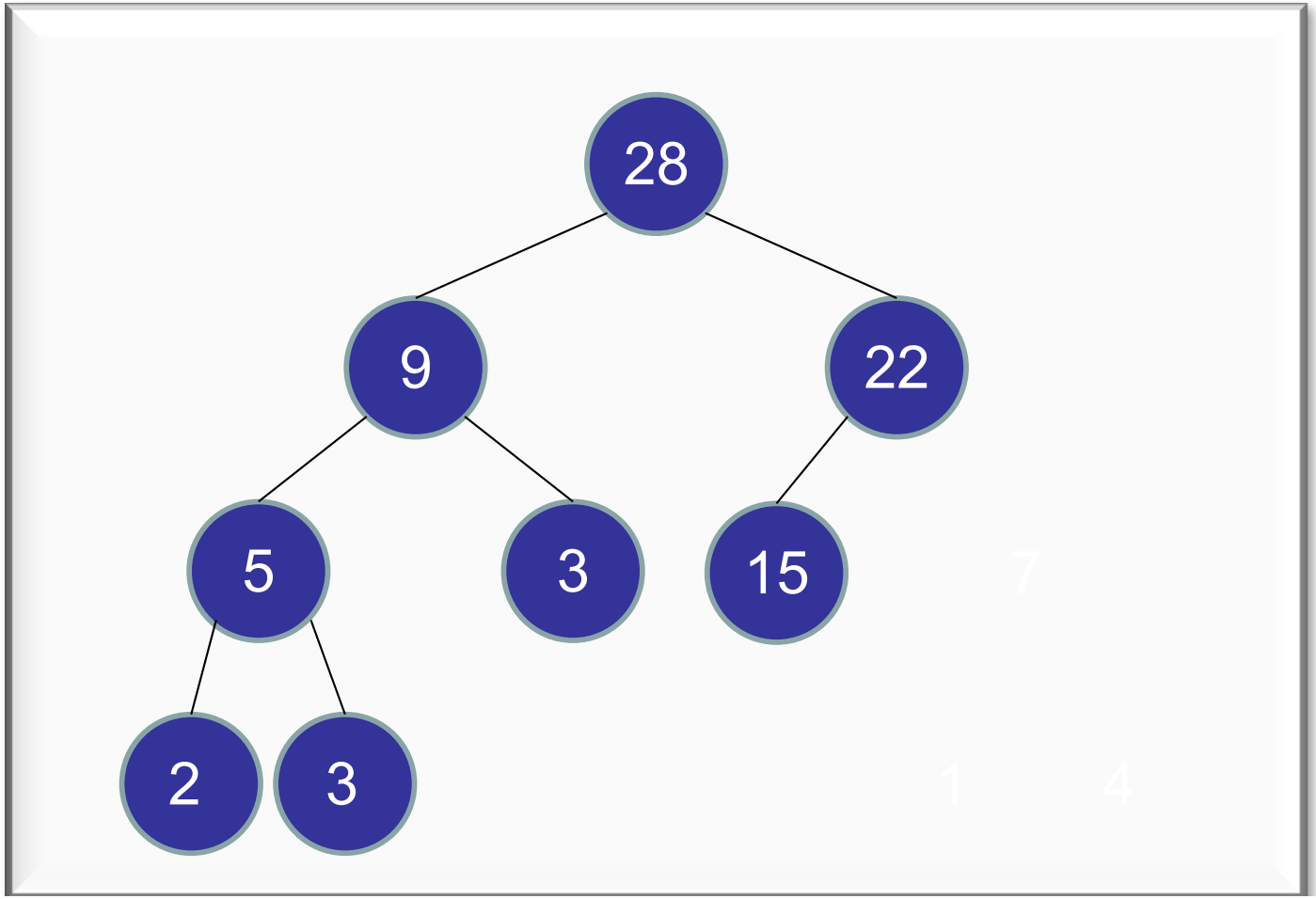
Is it a heap?

1. Yes
2. ✓ No.



Is it a heap?

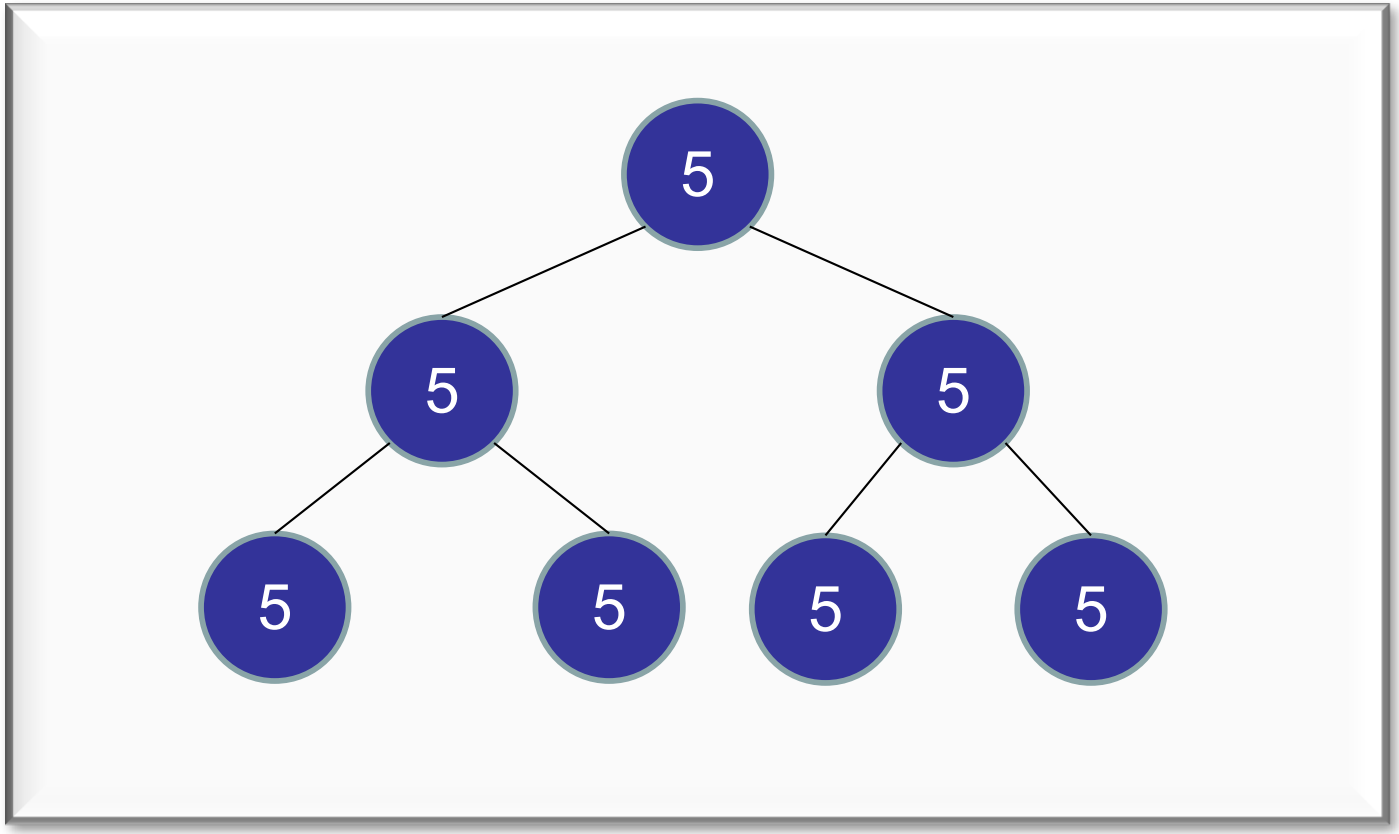
1. Yes
2. ✓ No.



Is it a heap?

✓1. Yes

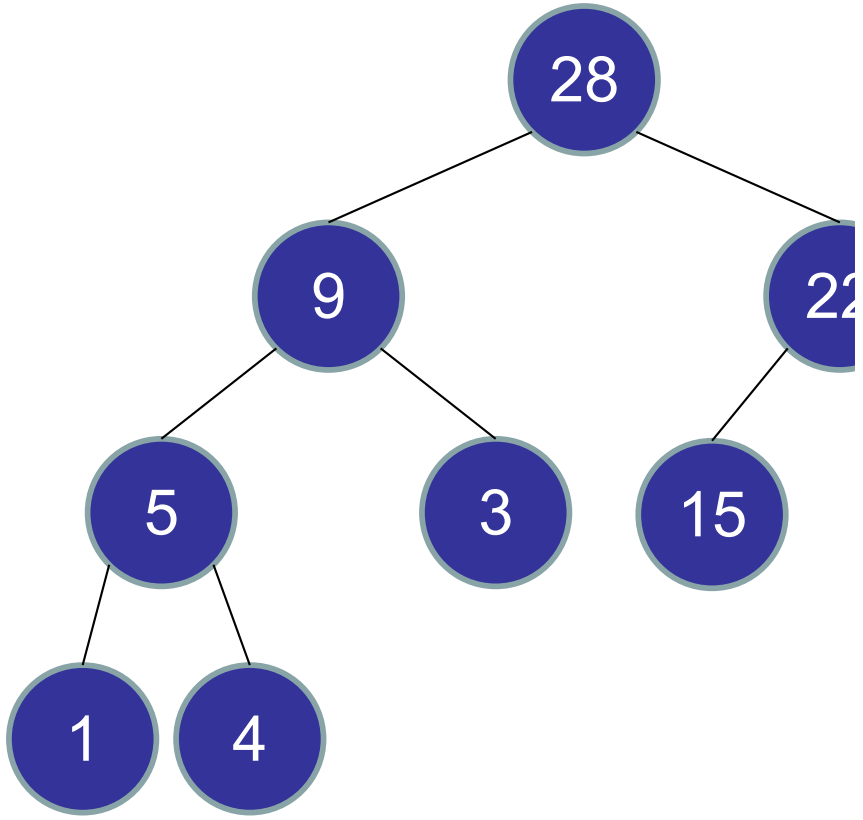
2. No.



Heap

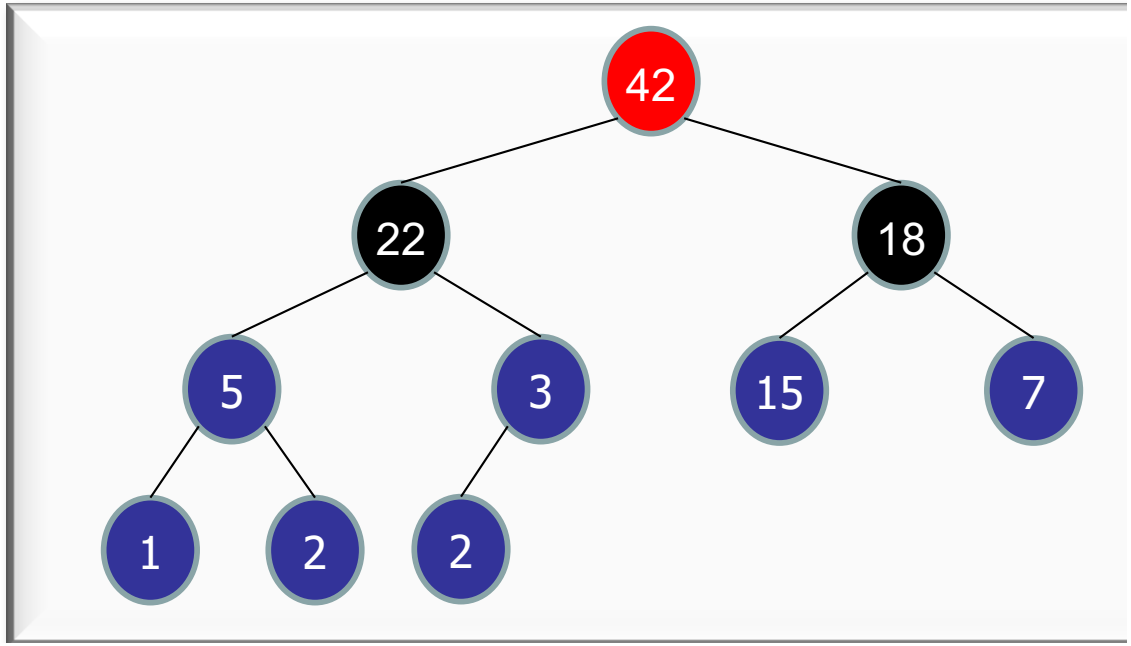
(aka Binary Heap or MaxHeap)

- Implements a Max Priority Queue
- Maintain a set of prioritized objects.
- Store items in a tree.
  - Biggest items at root.
  - Smallest items at leaves.
- Two properties:
  1. Heap Ordering
  2. Complete Binary Tree



What is the maximum height of a heap with n elements?

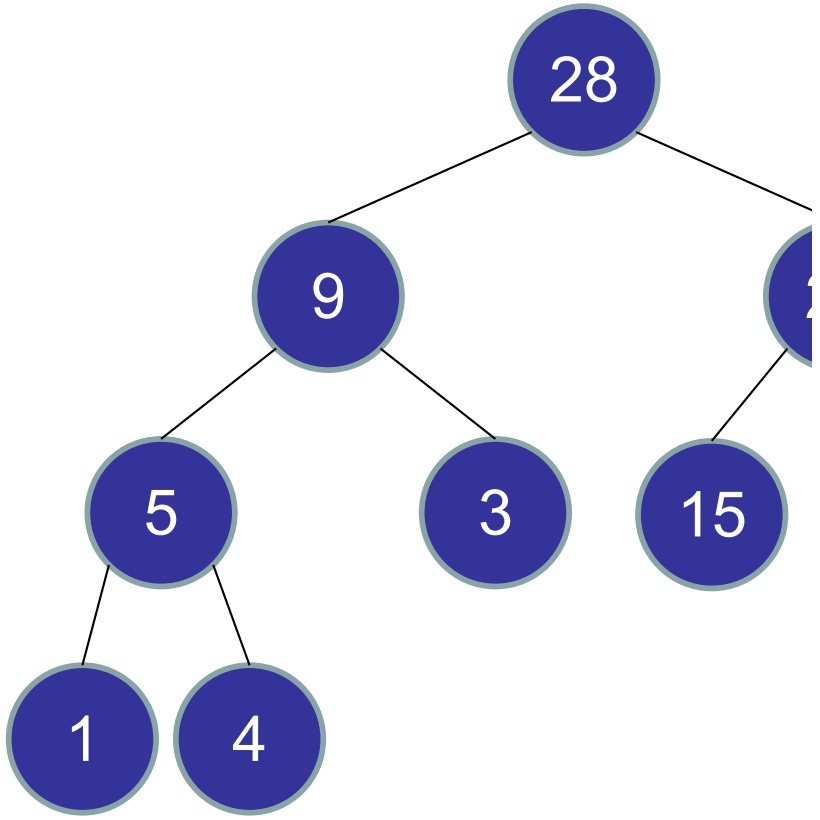
1. floor(log(n-1))
2. log(n)
3. ✓ floor(log n)
4. ceiling(log n)
5. ceiling(log(n+1))



Heap

(aka Binary Heap or MaxHeap)

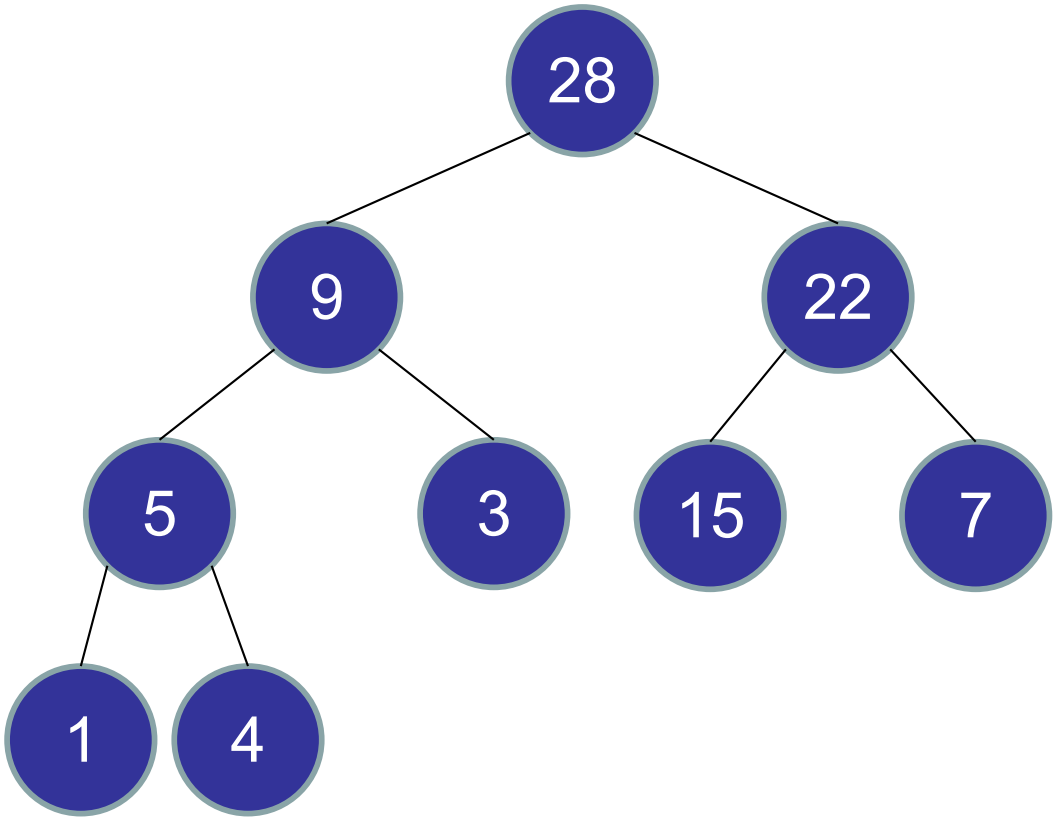
- Implements a Max Priority Queue
- Maintain a set of prioritized objects.
- Store items in a tree.
  - Biggest items at root.
  - Smallest items at leaves.
- Two properties:
  1. Heap Ordering
  2. Complete Binary Tree
- Height: O(log n)



Heap

Priority Queue Operations

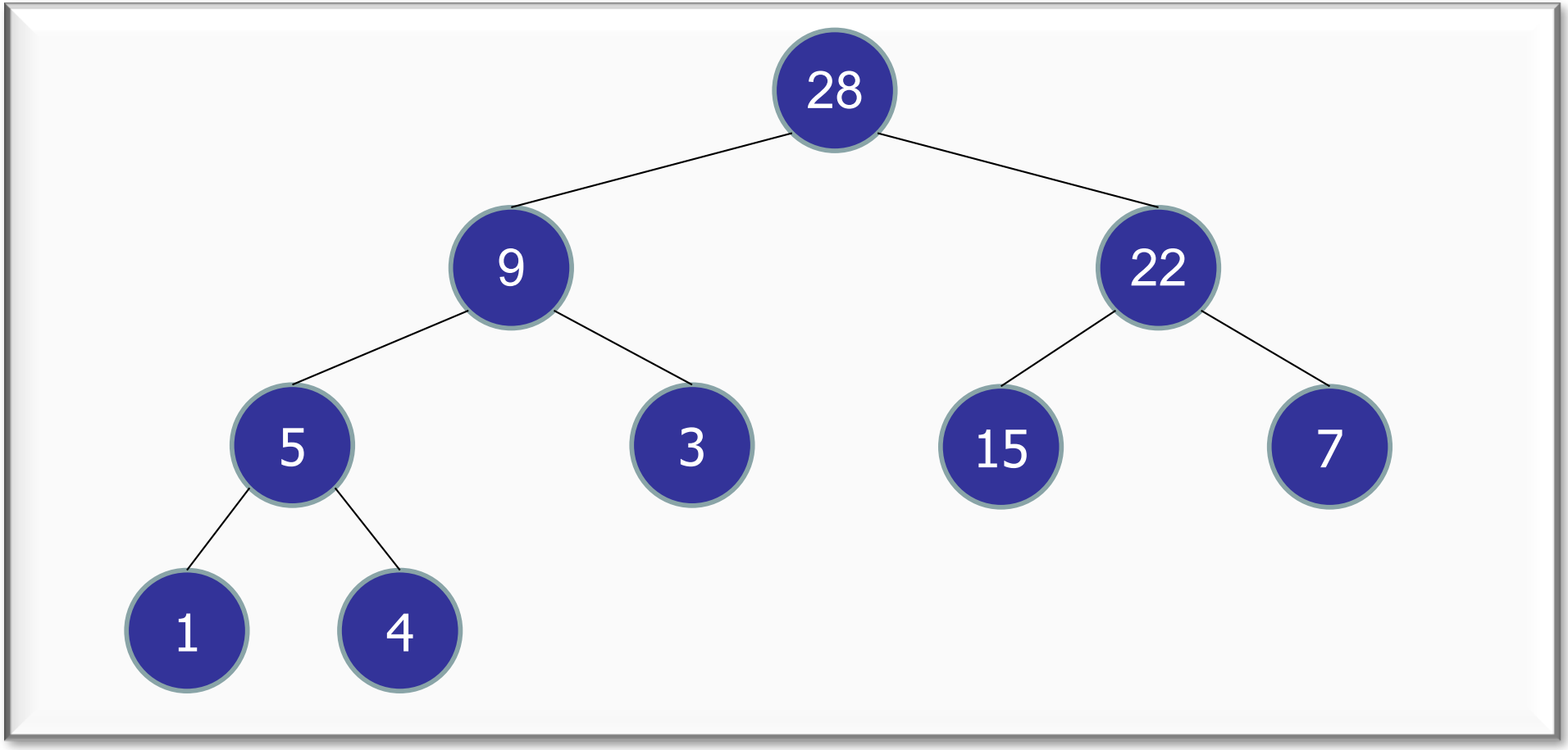
- insert
- extractMax
- increaseKey
- decreaseKey
- delete



Inserting in a Heap

`insert(25)`:

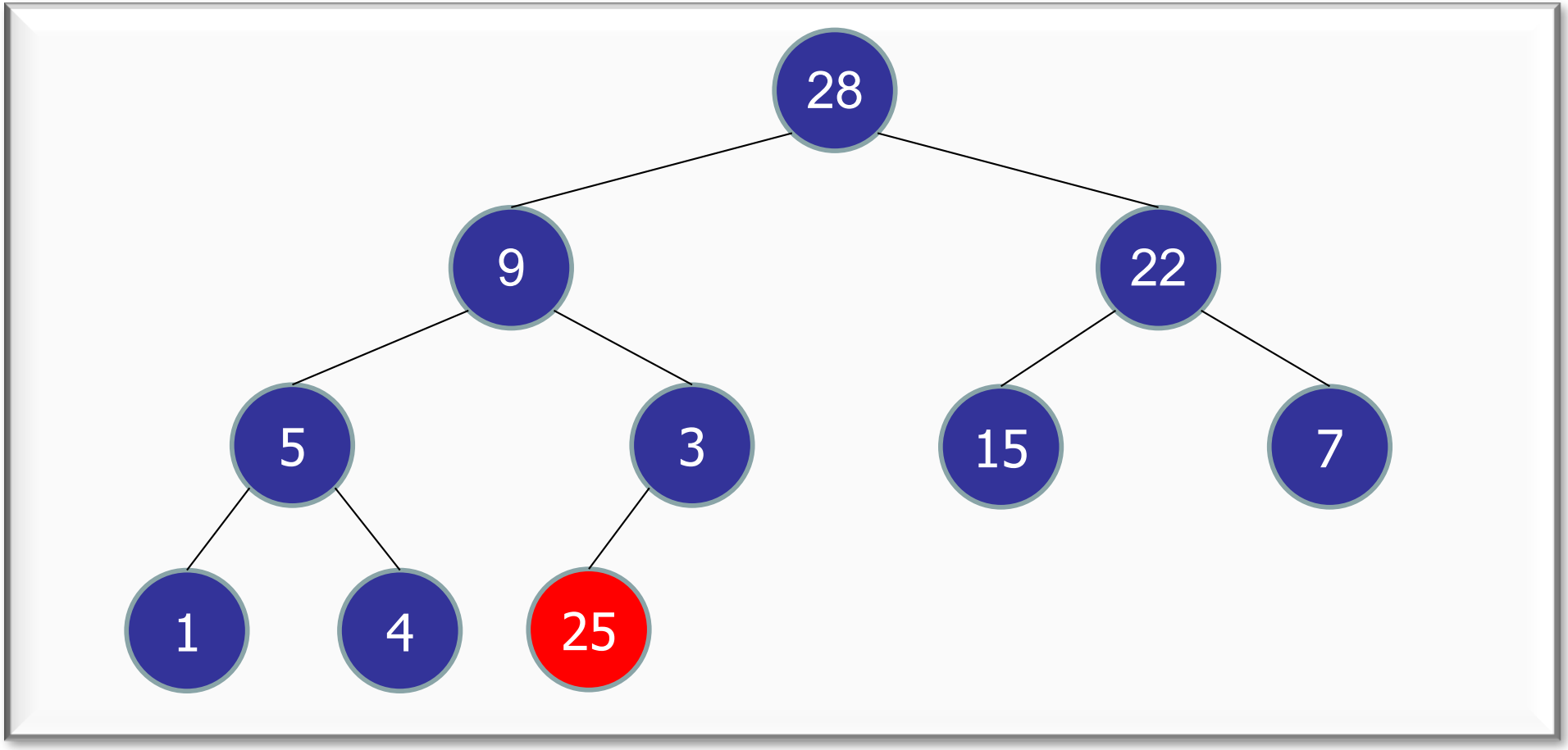
- Step one: add a new leaf with priority 25.



Inserting in a Heap

`insert(25)`:

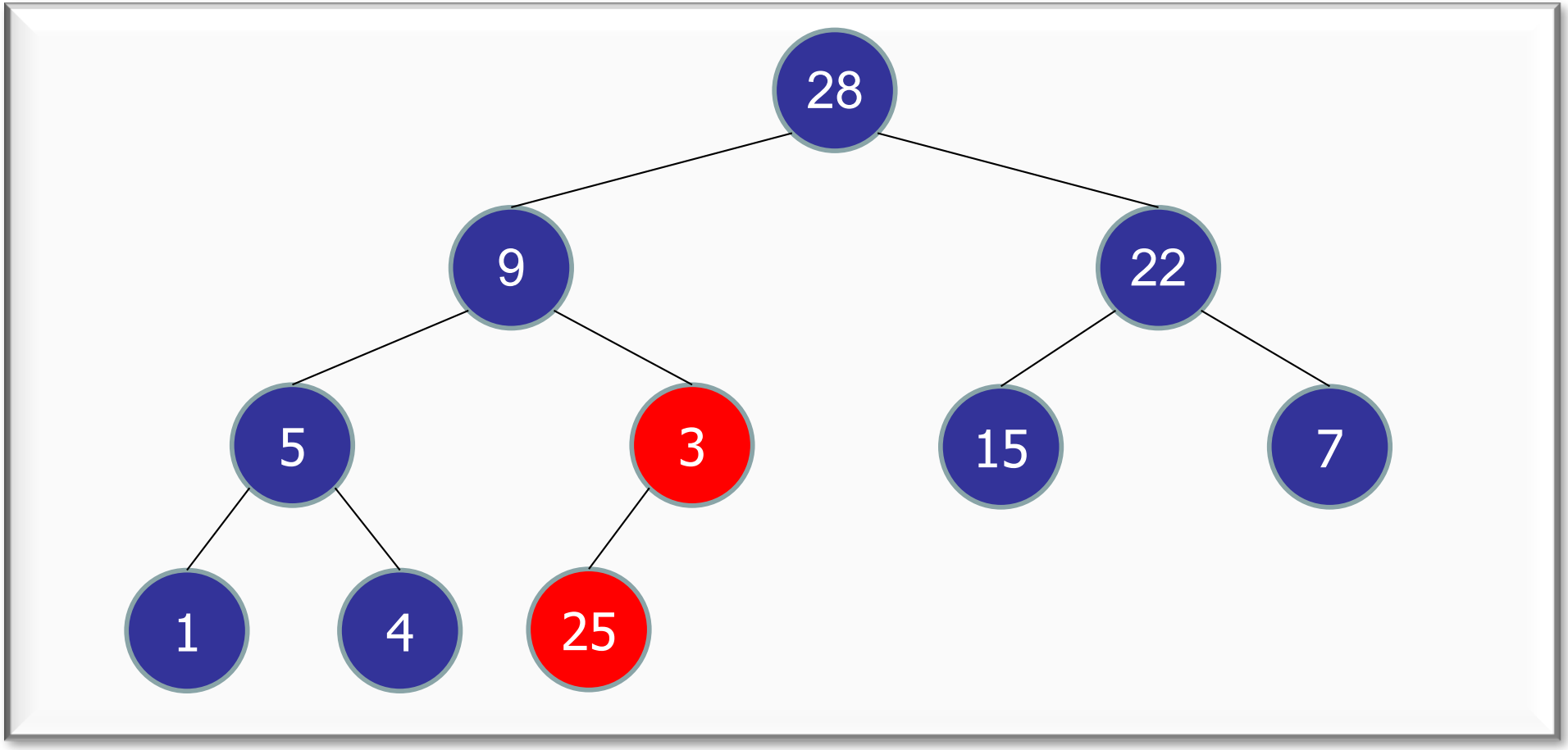
- Step one: add a new leaf with priority 25.



Inserting in a Heap

`insert(25)`:

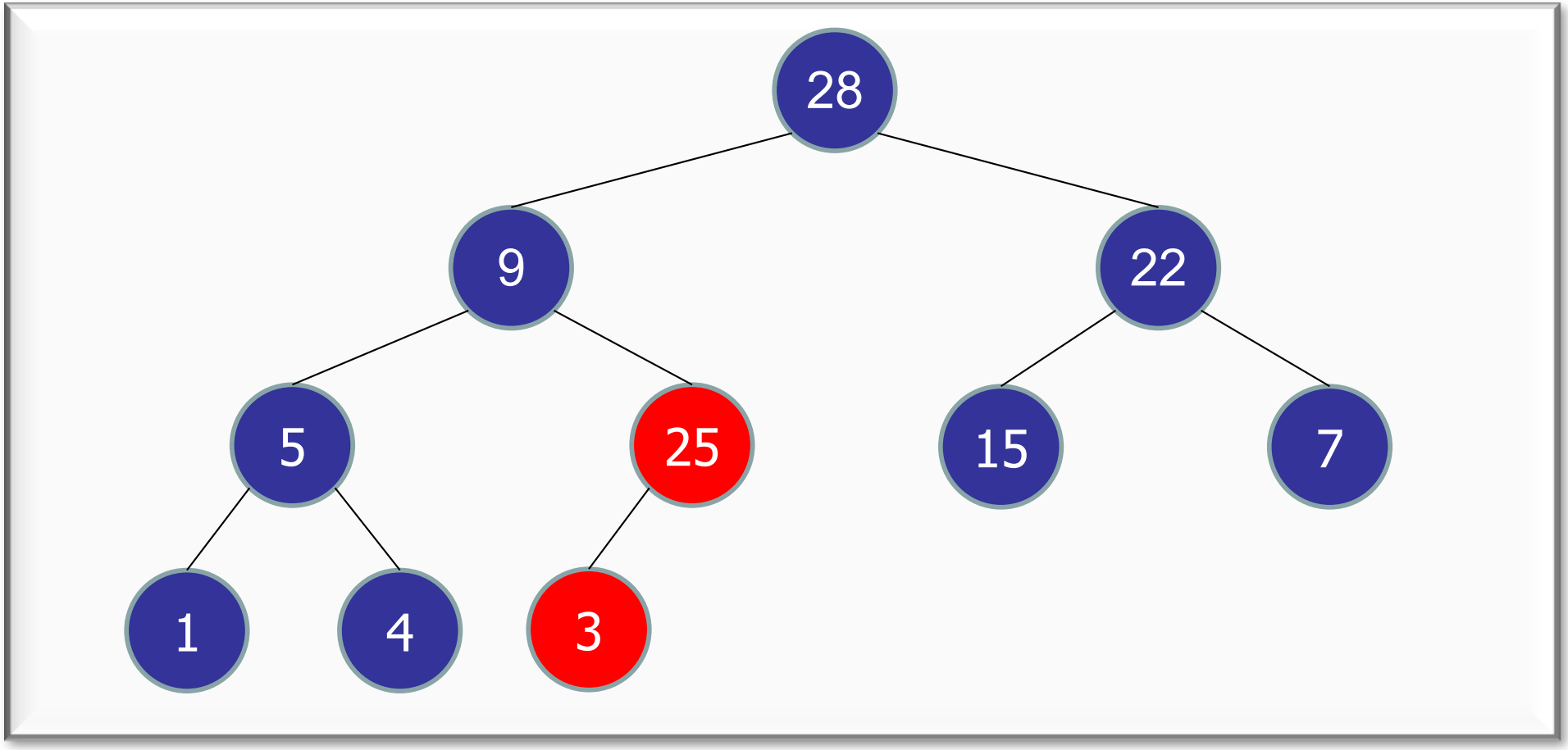
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

`insert(25)`:

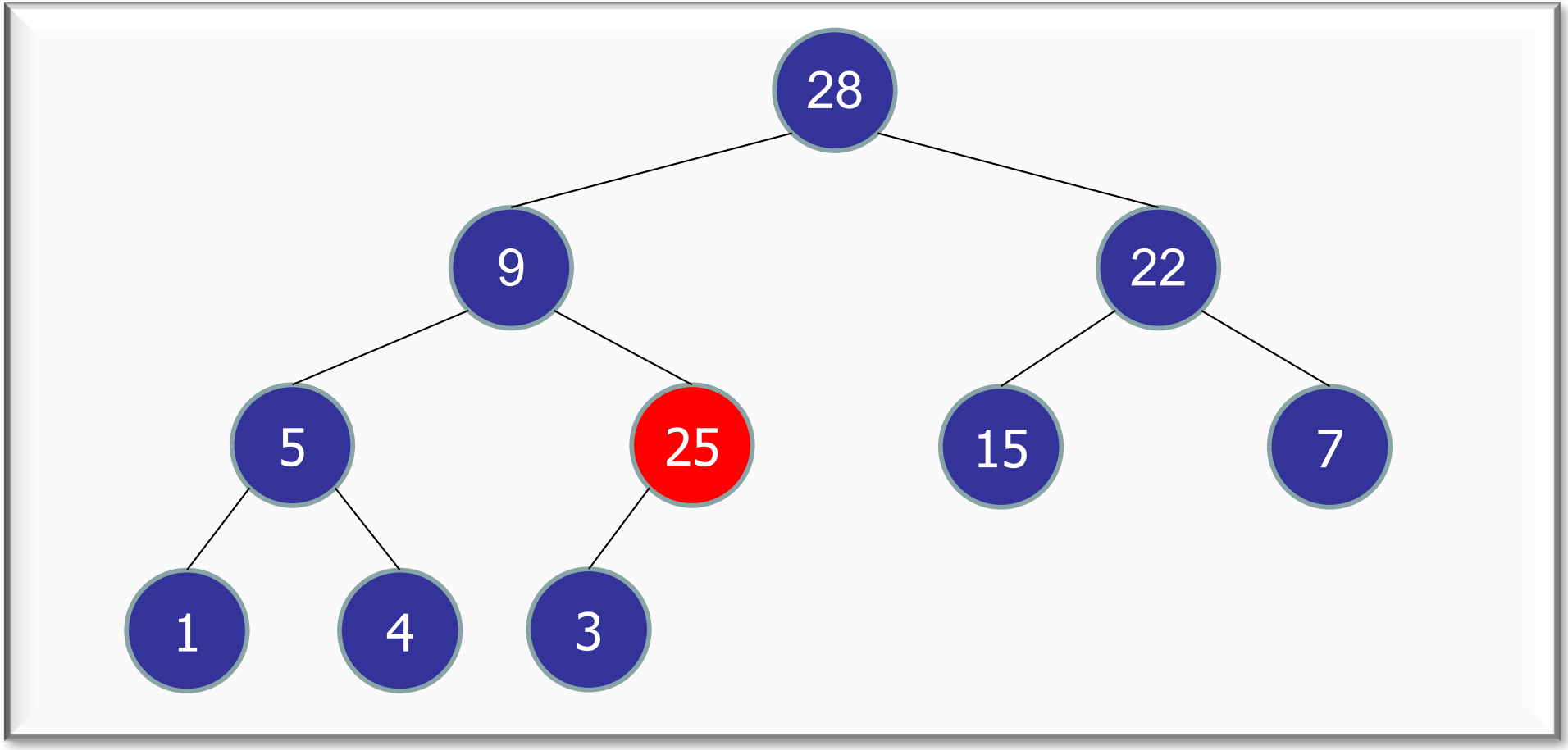
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

`insert(25)`:

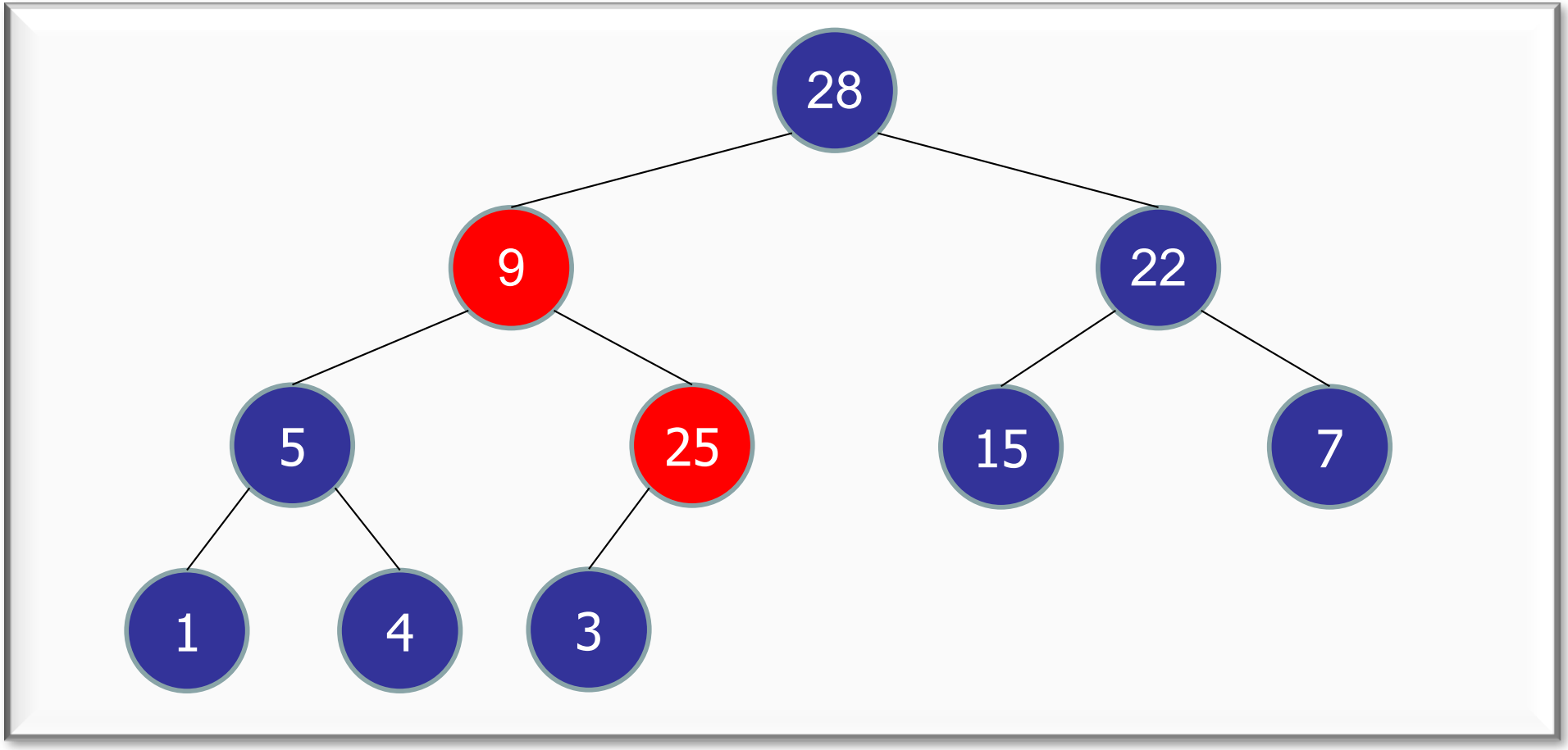
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

`insert(25)`:

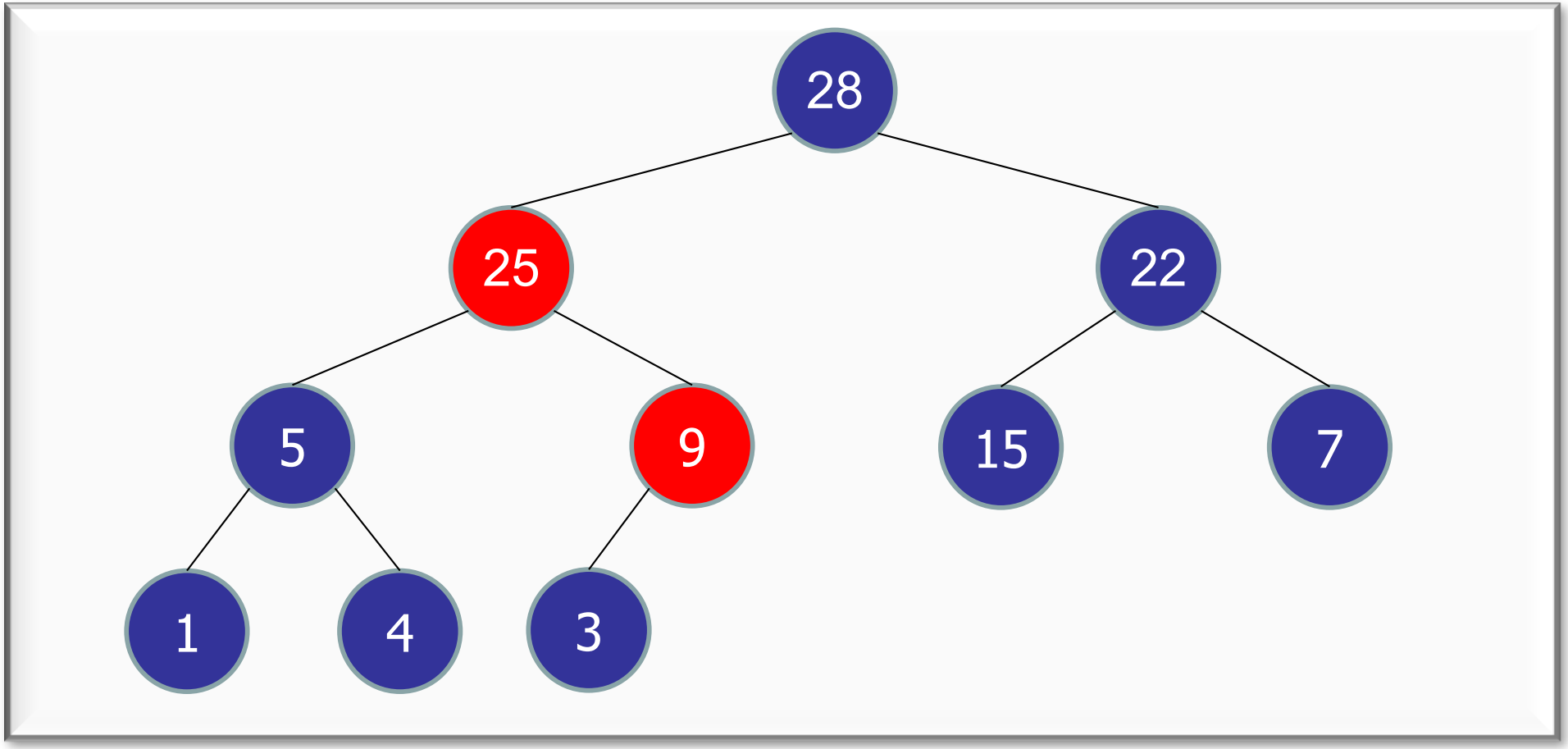
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

`insert(25)`:

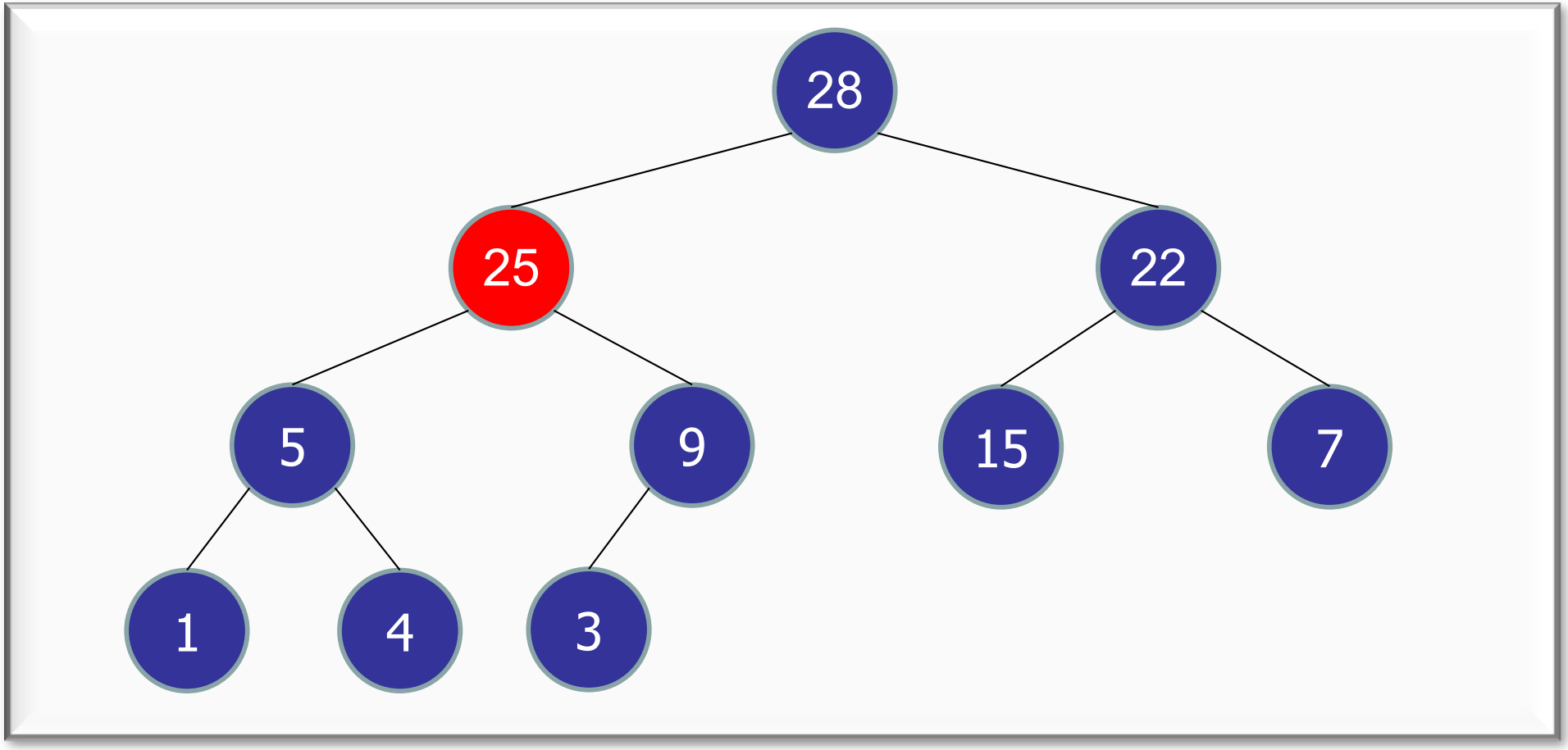
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

`insert(25)`:

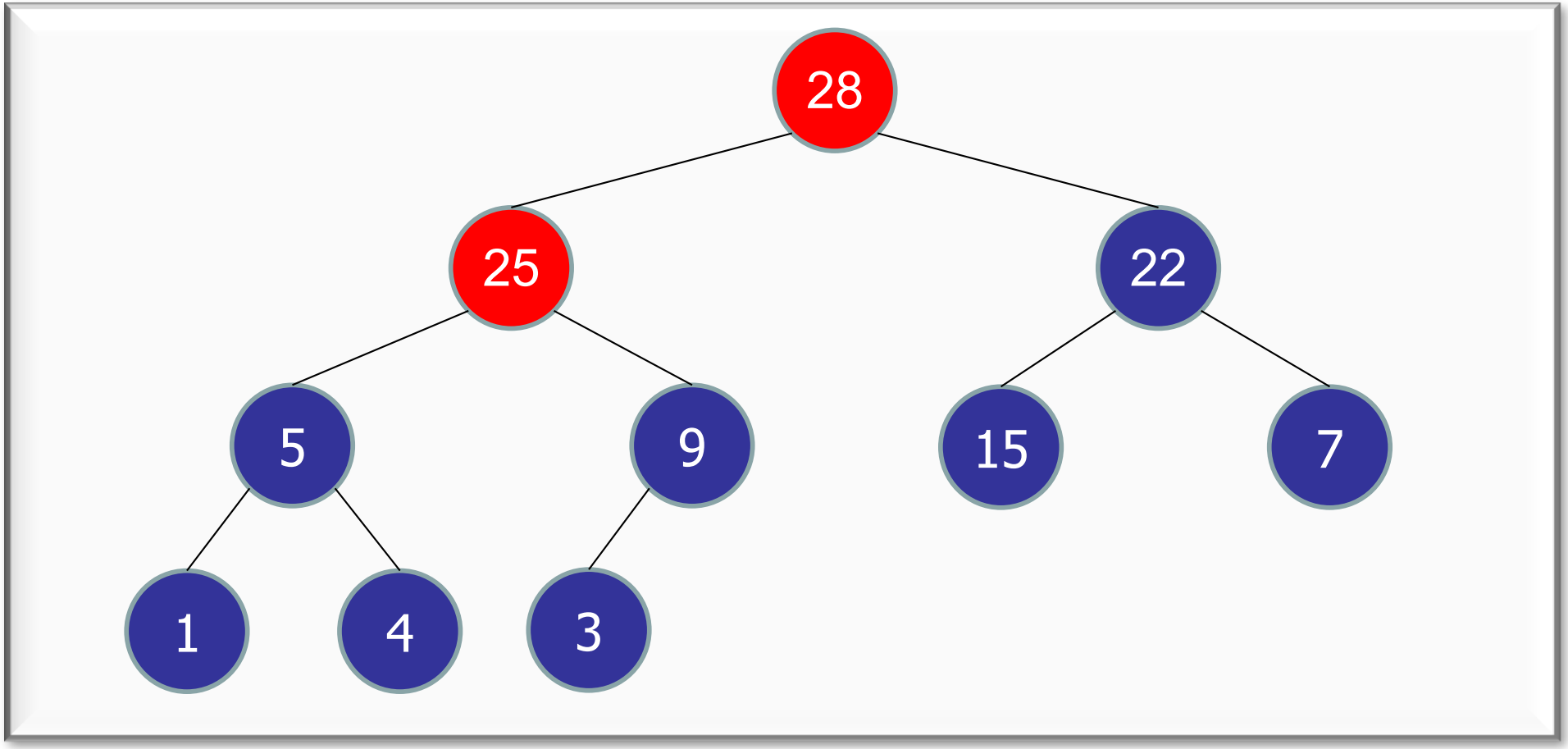
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

`insert(25)`:

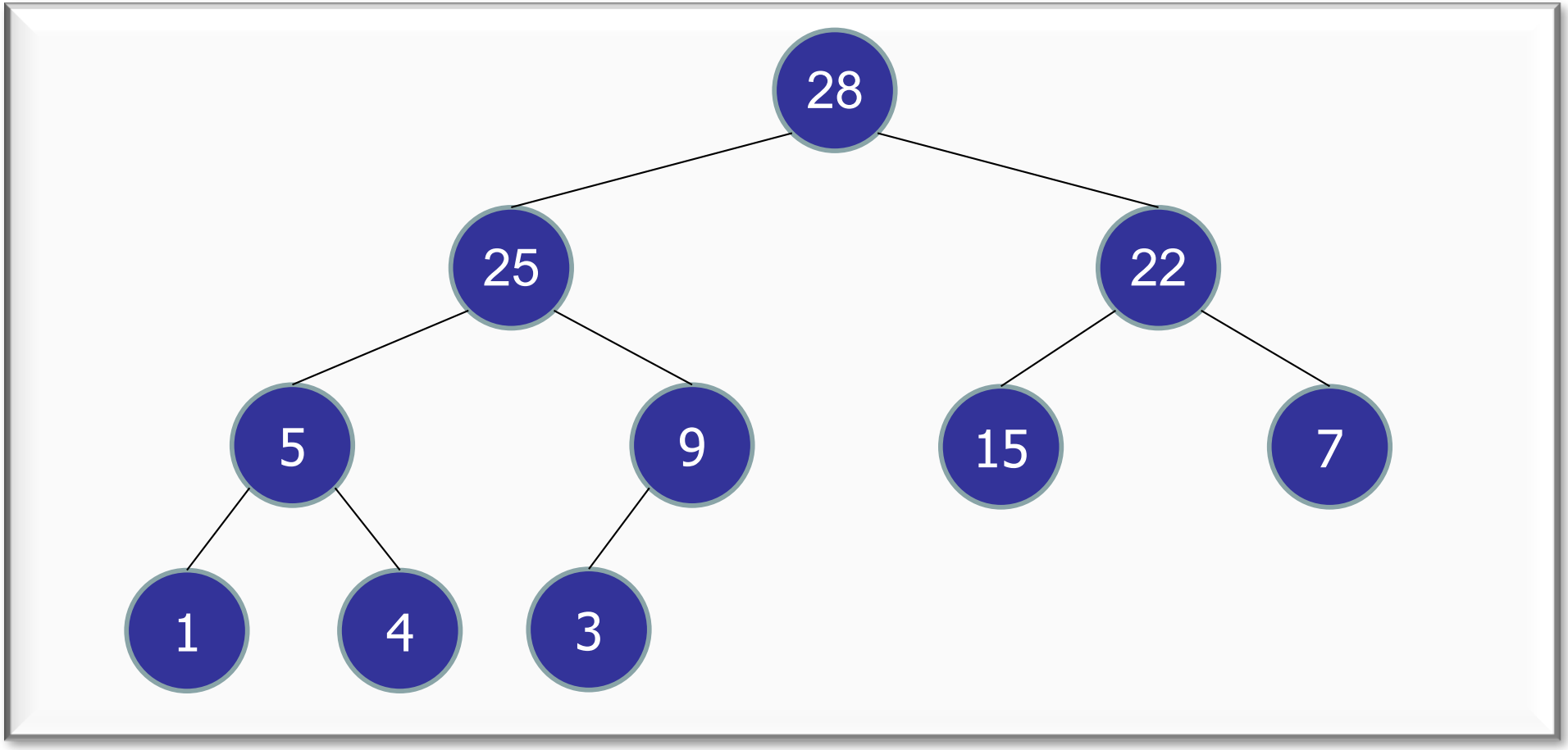
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

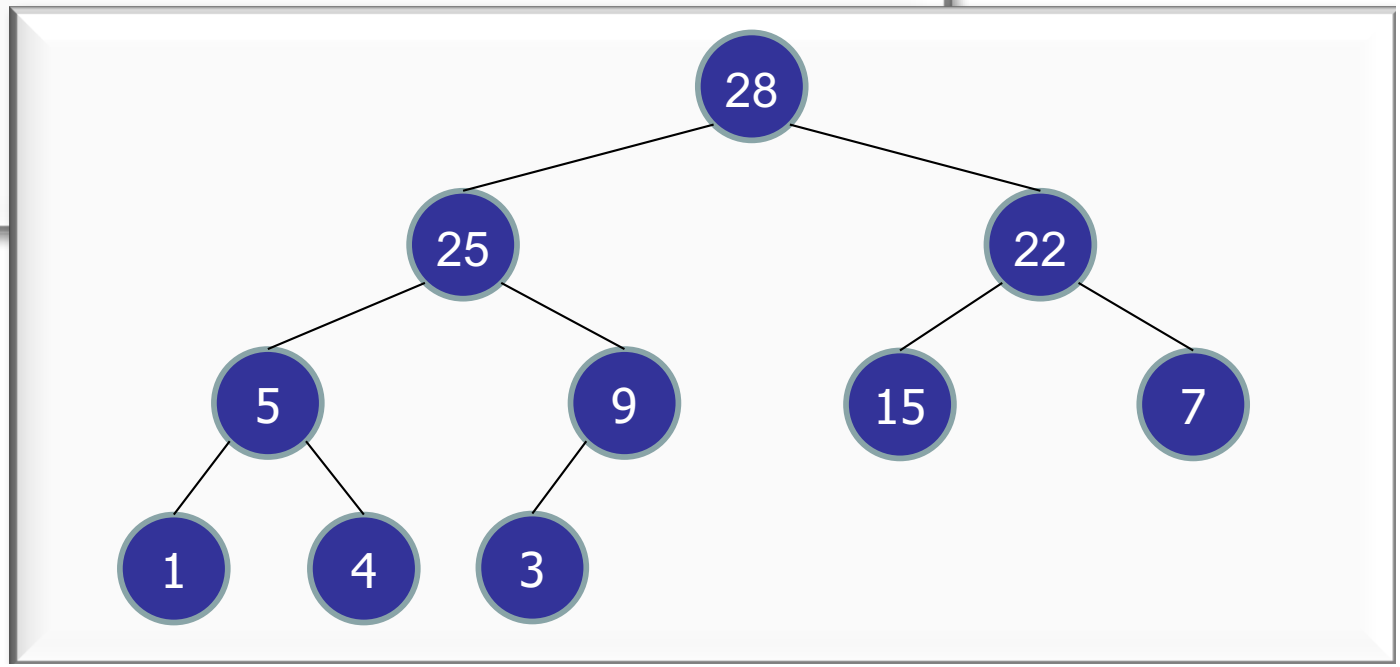
`insert(25)`:

- Step one: add a new leaf with priority 25.
- Step two: bubble up



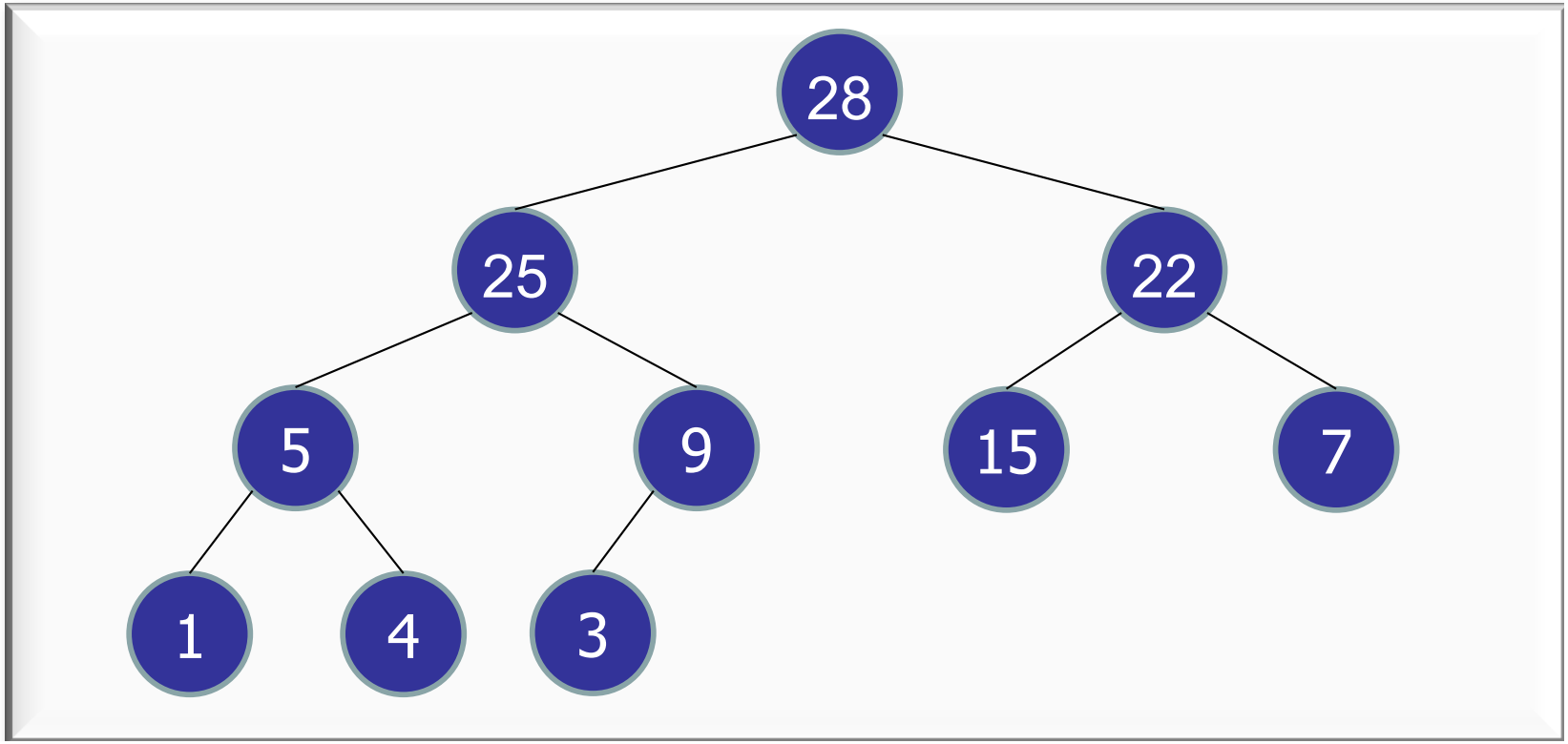
Inserting in a Heap

```
bubbleUp(Node v) {
   while (v != null) {
      if (priority(v) > priority(parent(v)))
            swap(v, parent(v));
      else return;
      v = parent(v);
   }
}
```



Inserting in a Heap

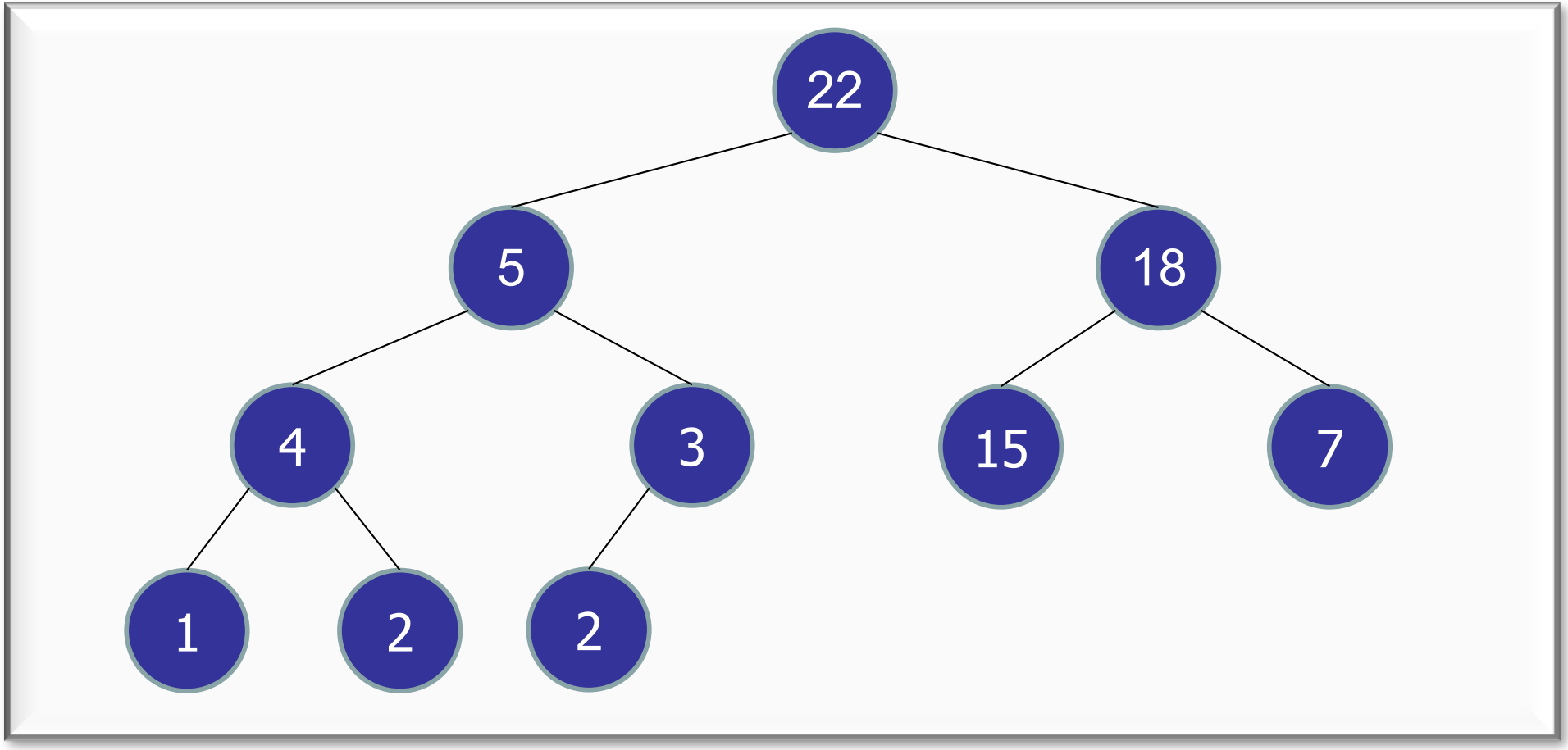
```
insert(Priority p, Key k) {
   Node v = m_completeTree.insert(p,k);
   bubbleUp(v);
}
```



Inserting in a Heap

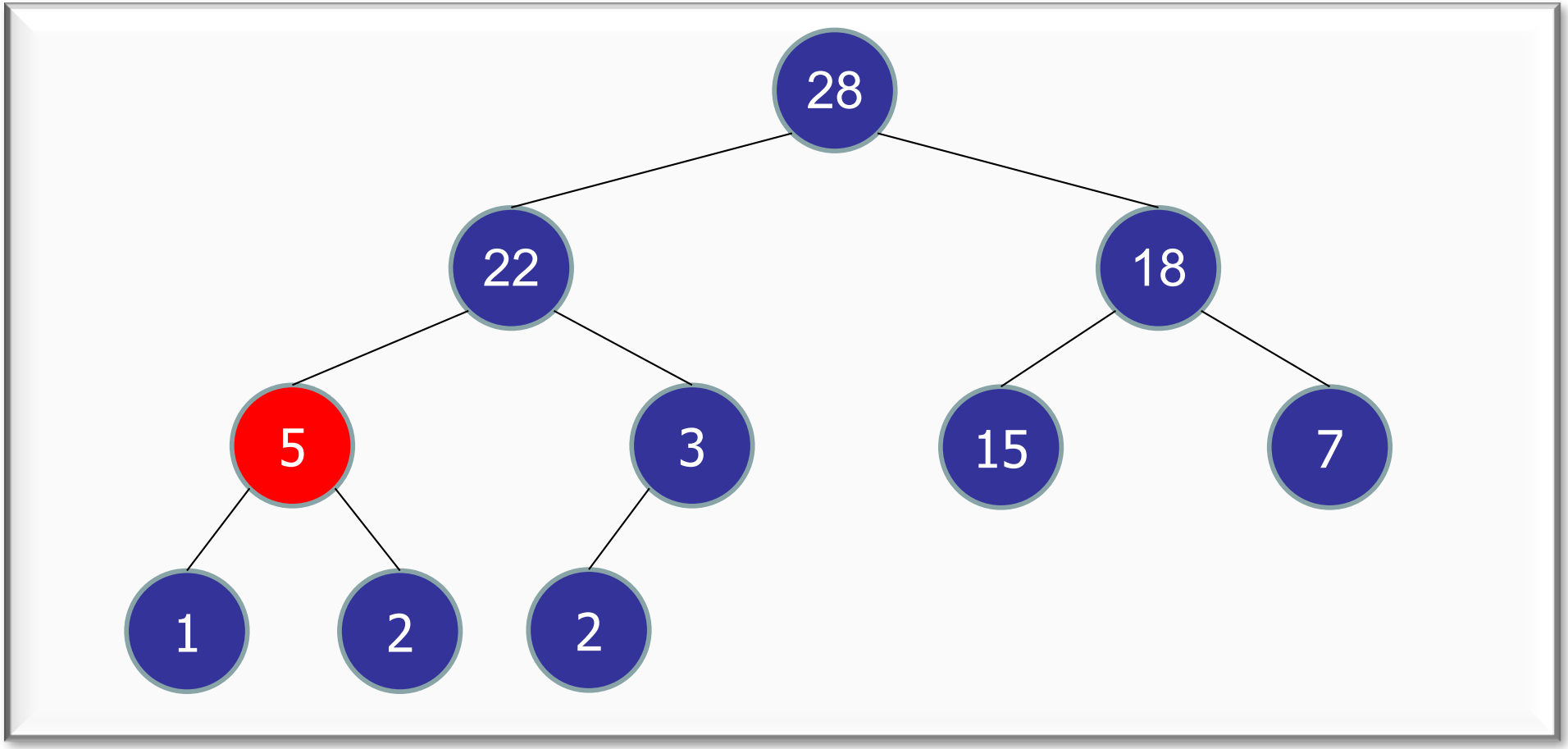
`insert(...)` :

- On completion, heap order is restored.
- Complete binary tree.



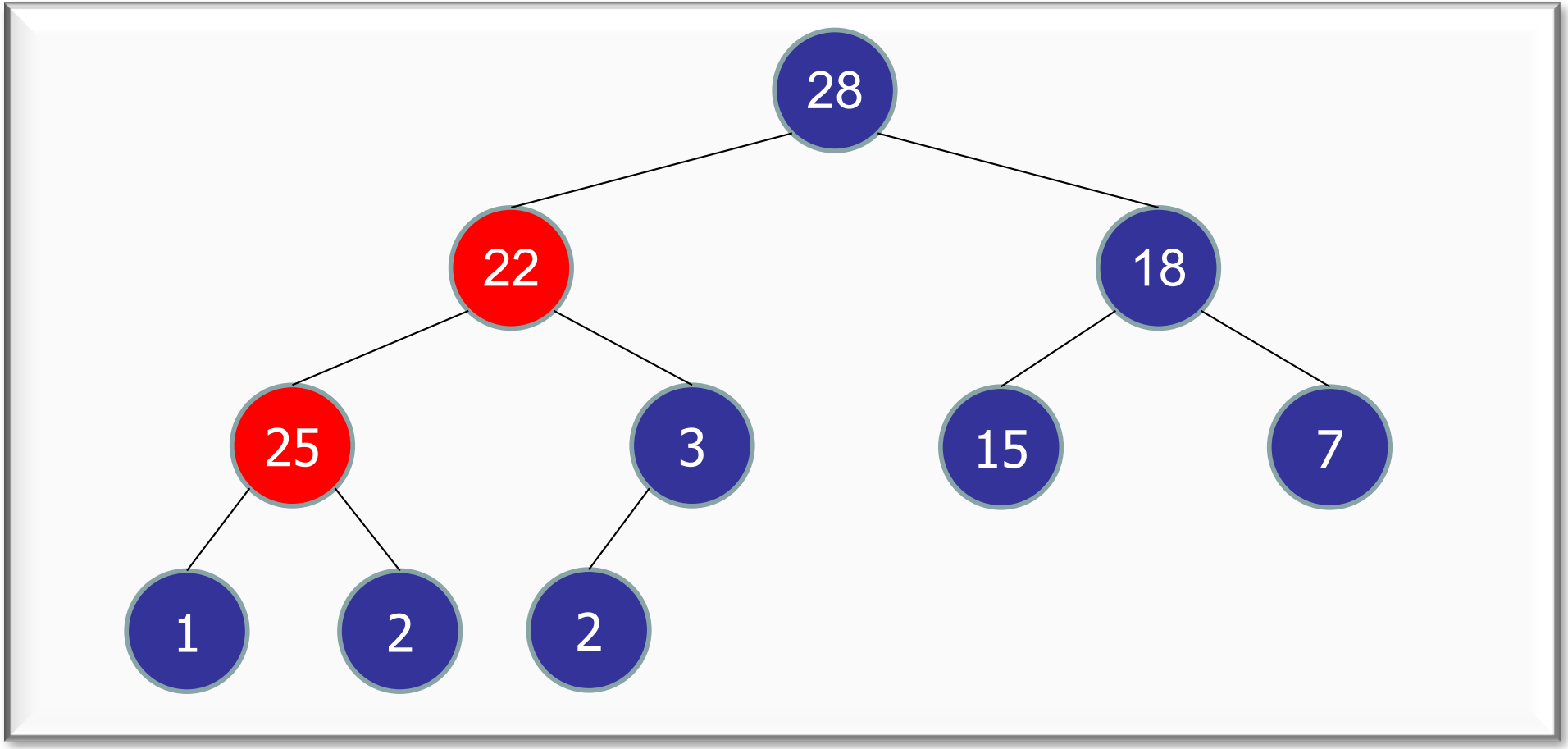
Inserting in a Heap

`increaseKey(5 → 25):`



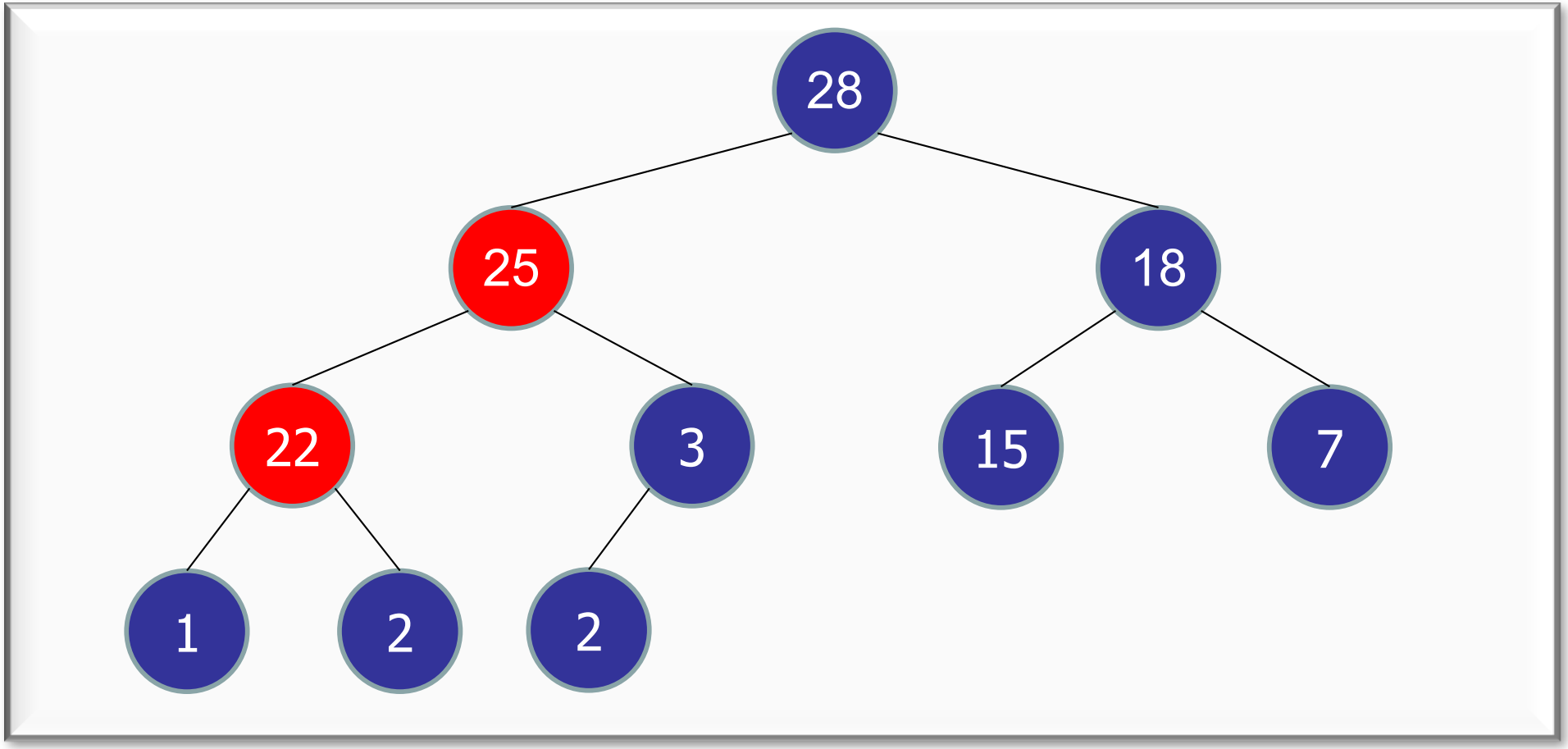
Inserting in a Heap

`increaseKey(5 → 25)`: bubbleUp(25)



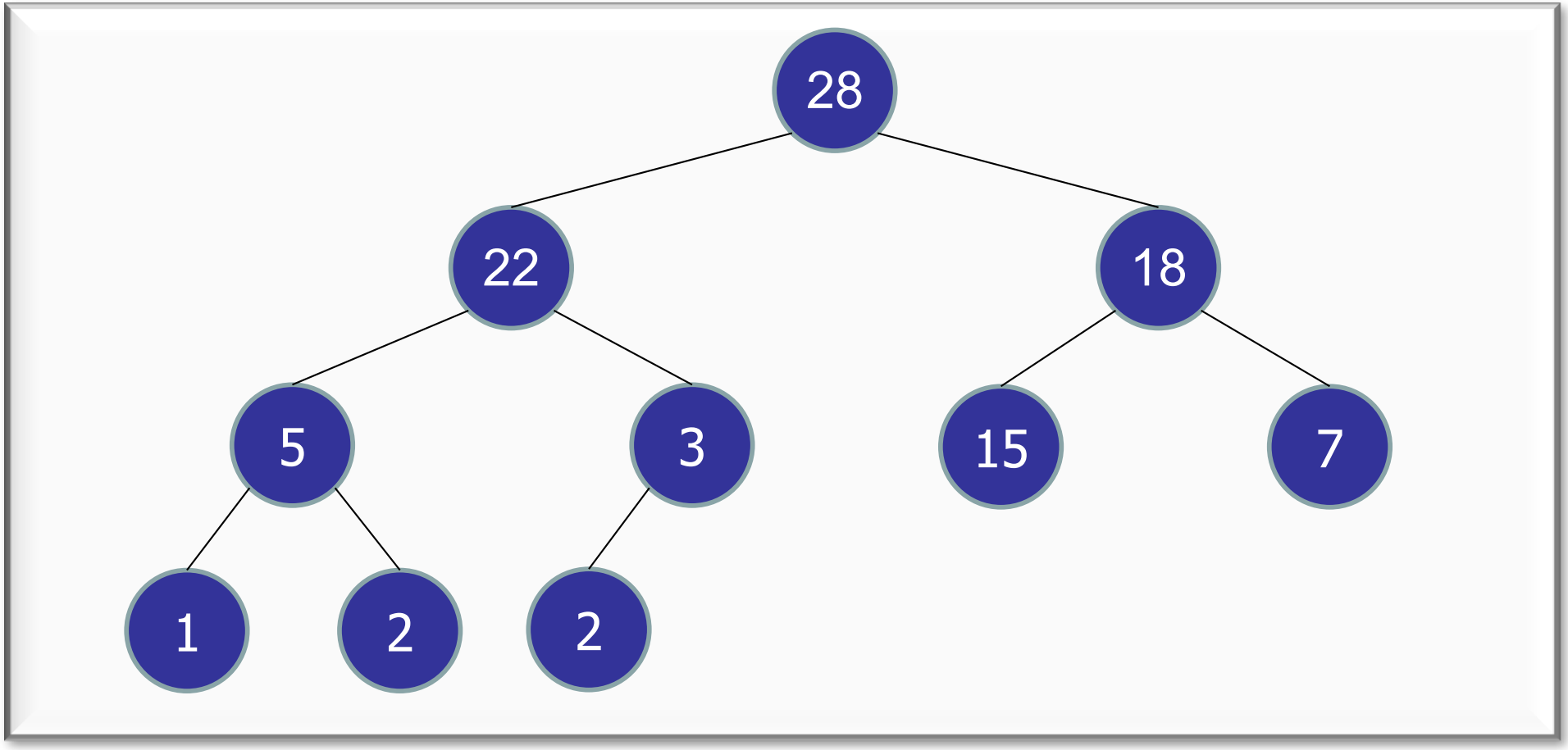
Inserting in a Heap

`increaseKey(5 → 25)`: bubbleUp(25)



Inserting in a Heap

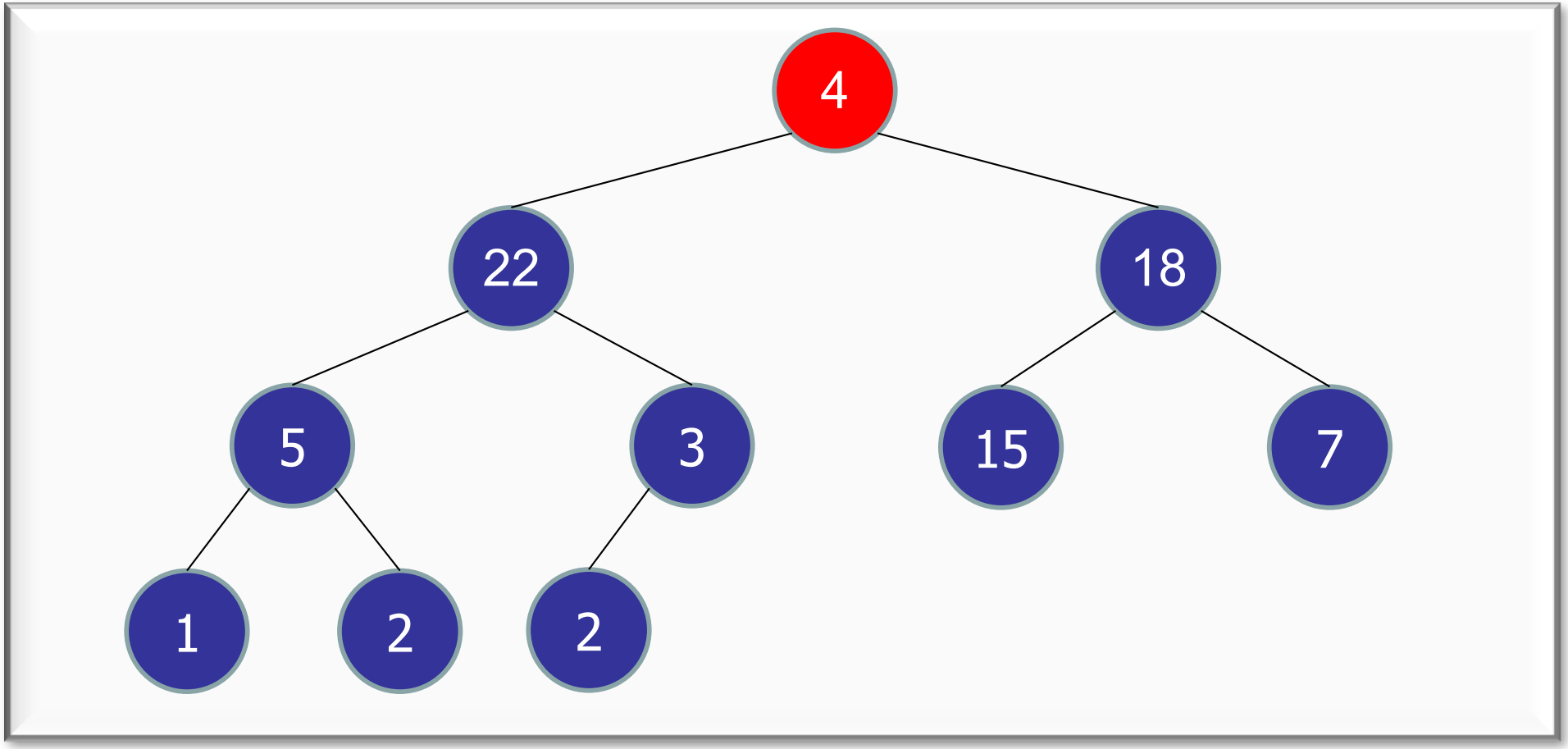
`decreaseKey(28 → 4):`



Inserting in a Heap

`decreaseKey(28 → 4)`:

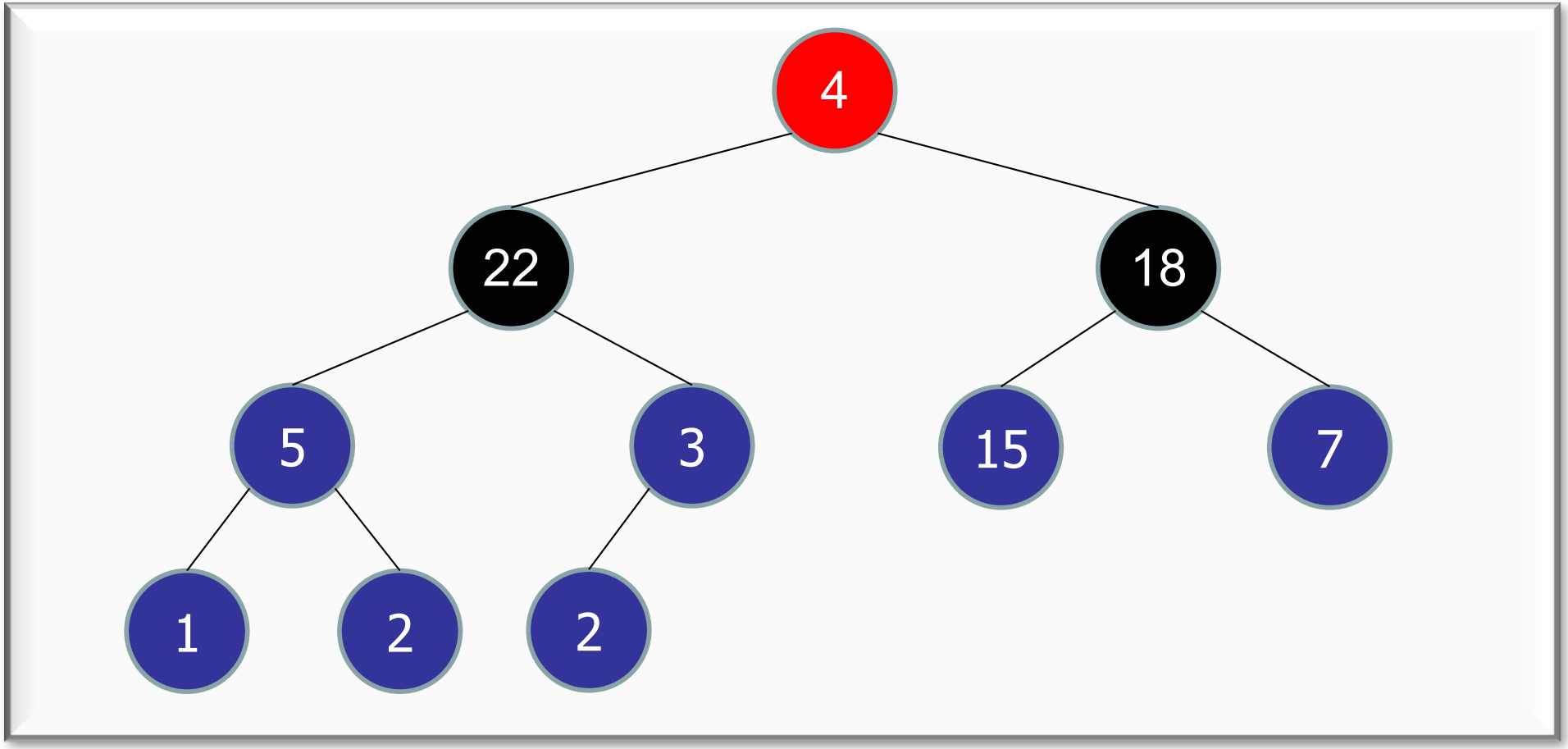
- Step 1: Update the priority



Inserting in a Heap

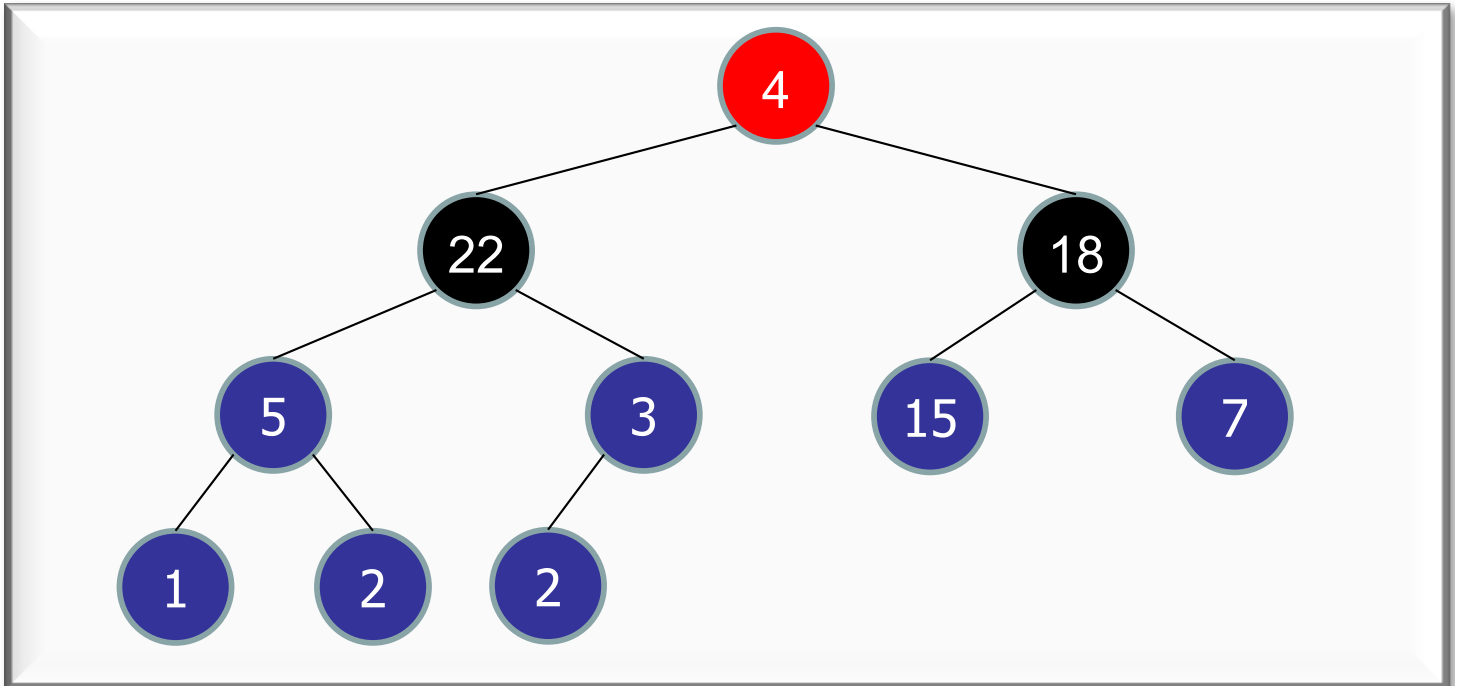
`decreaseKey(28 → 4)`:

- Step 1: Update the priority
- Step 2: bubbleDown(4)



Which way to bubbleDown?

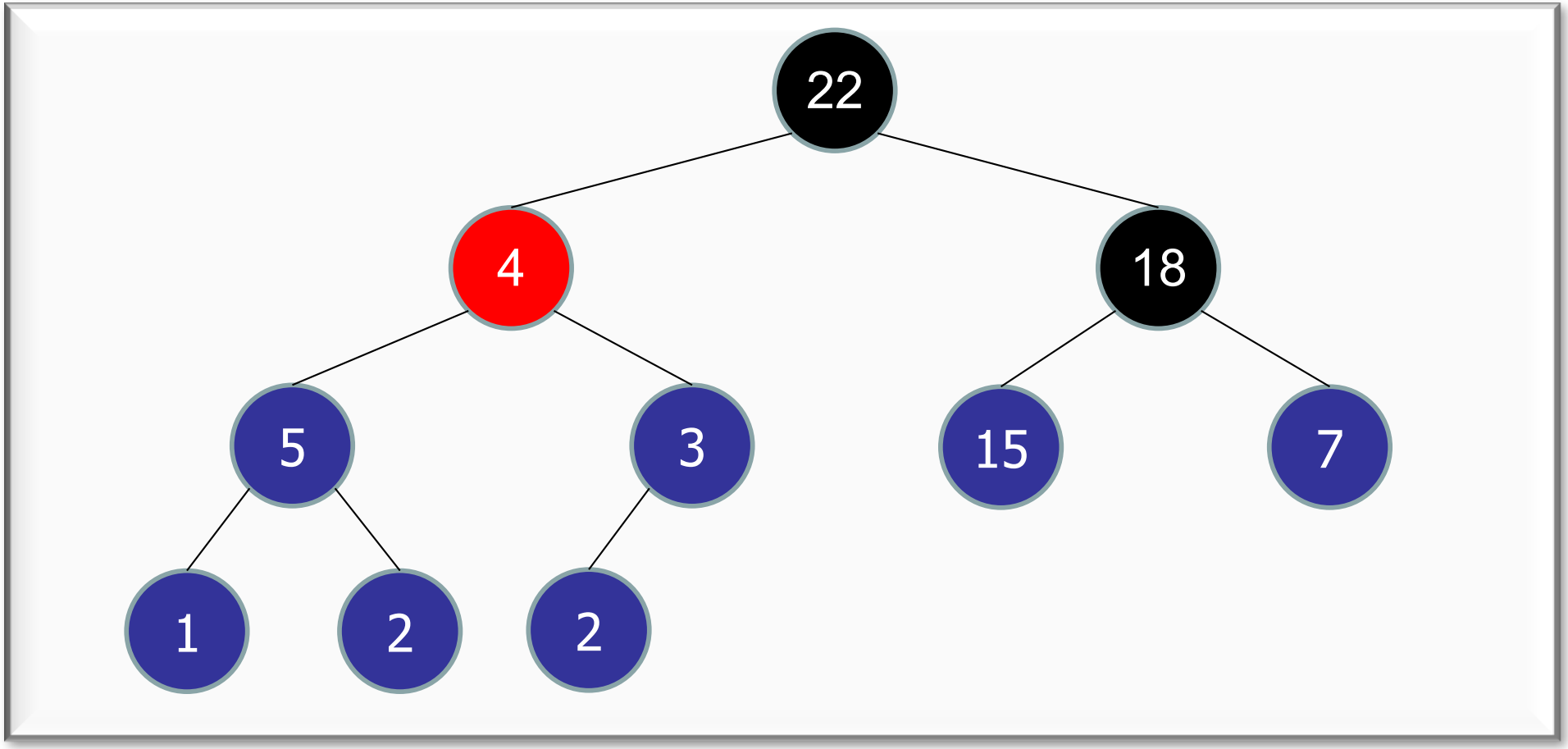
✓1. Left
2. Right
3. Does not matter



Inserting in a Heap

`decreaseKey(28 → 4)` :

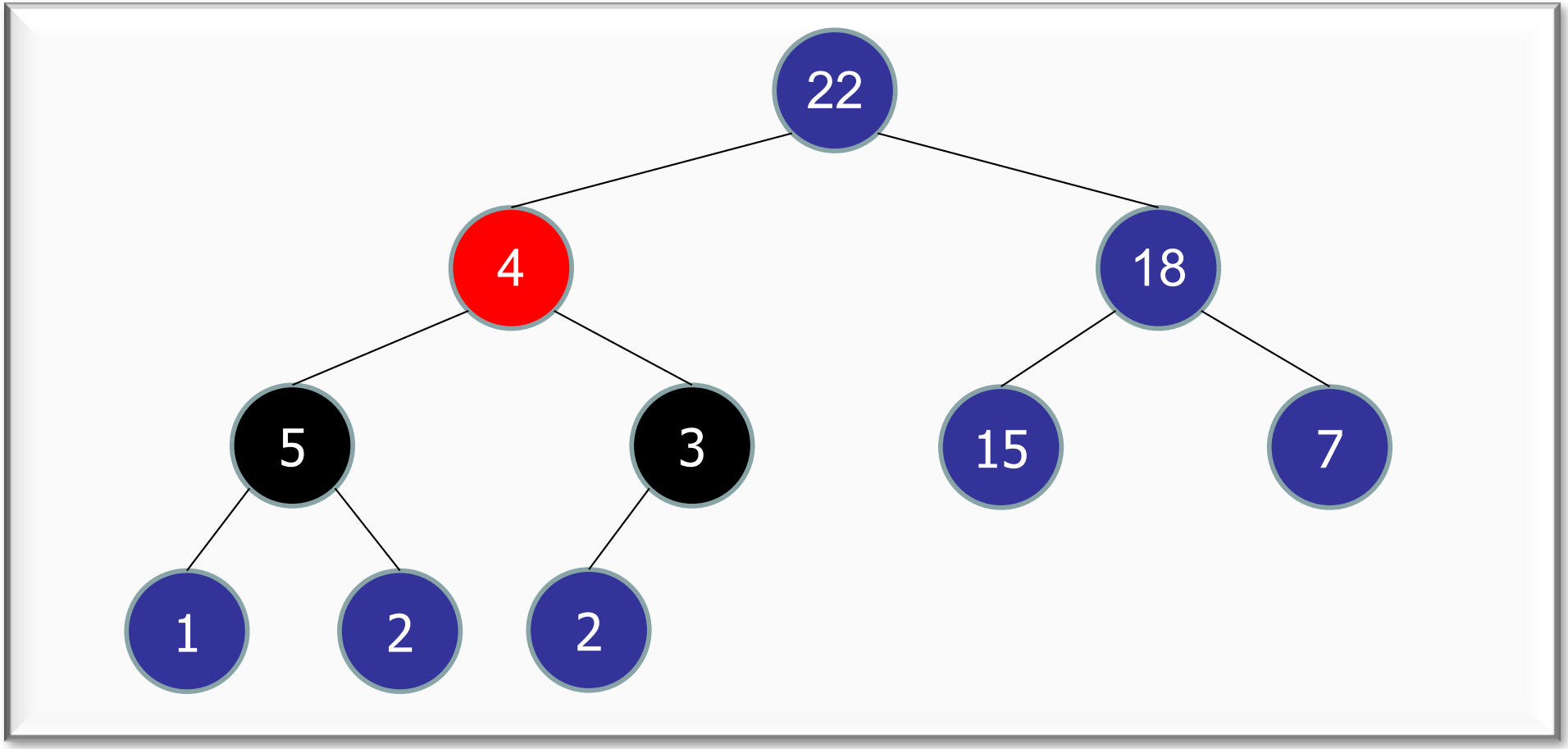
- Step 1: Update the priority
- Step 2: bubbleDown(4)



Inserting in a Heap

`decreaseKey(28 → 4)` :

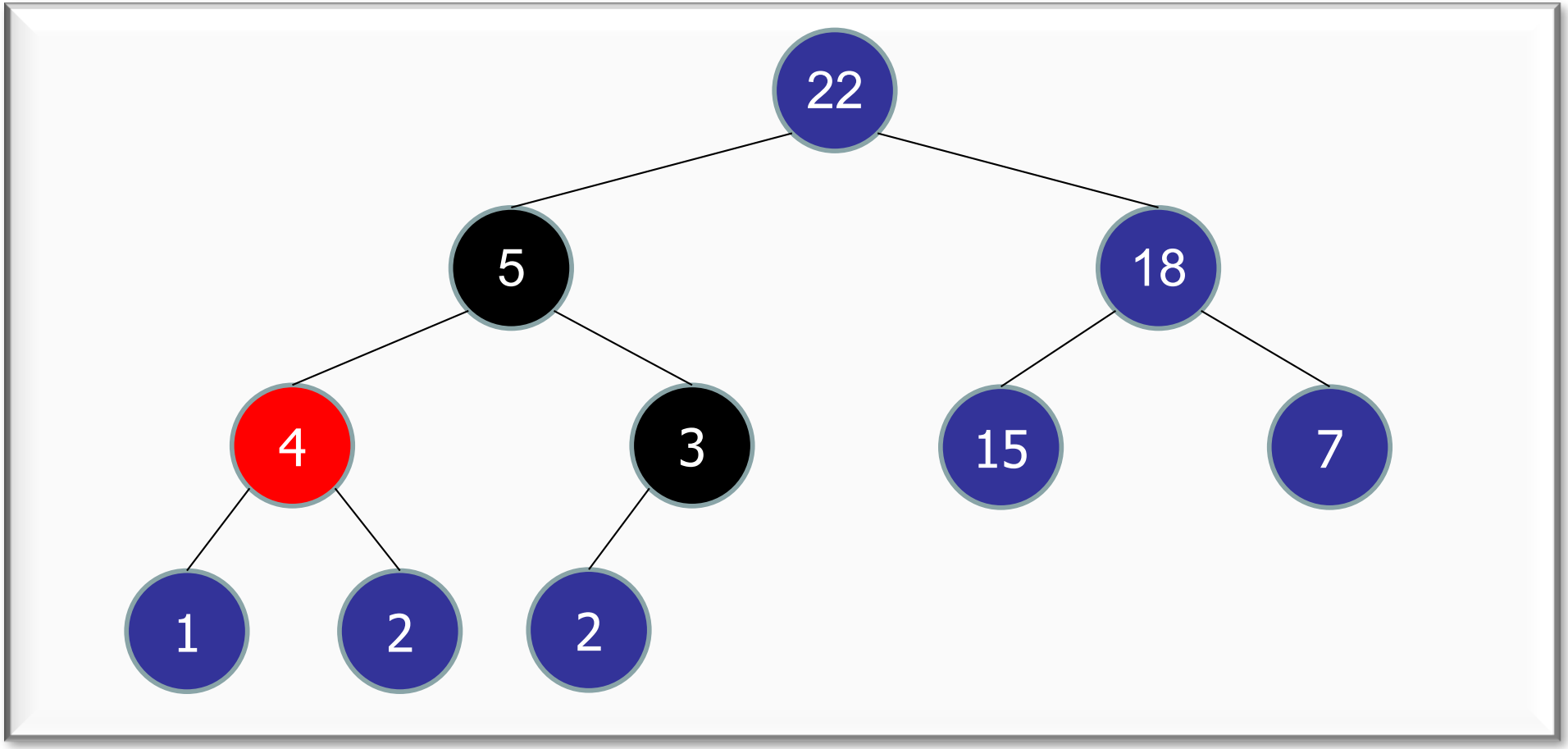
- Step 1: Update the priority
- Step 2: bubbleDown(4)



Inserting in a Heap

`decreaseKey(28 → 4)` :

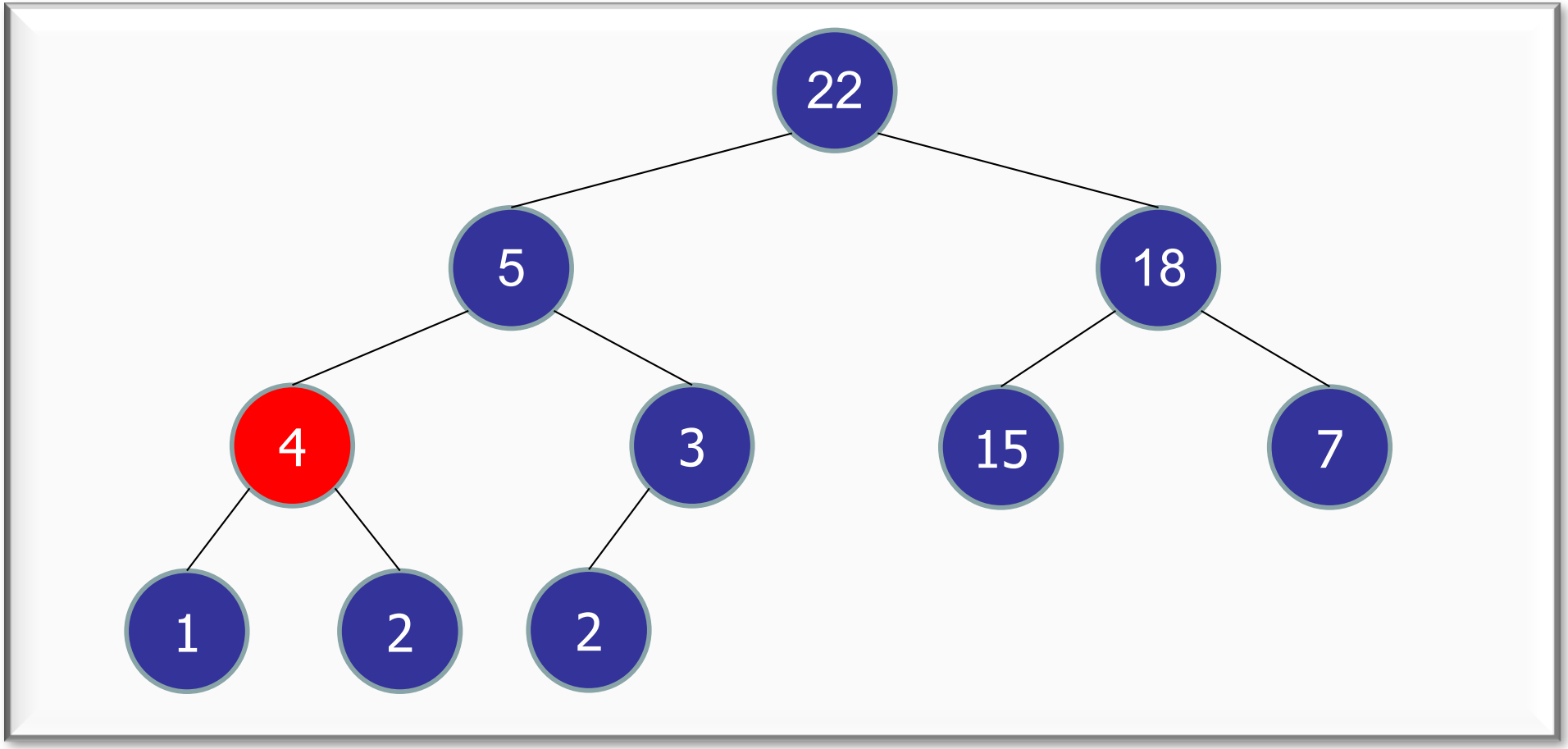
- Step 1: Update the priority
- Step 2: bubbleDown(4)



Inserting in a Heap

`decreaseKey(28 → 4)` :

- Step 1: Update the priority
- Step 2: bubbleDown(4)



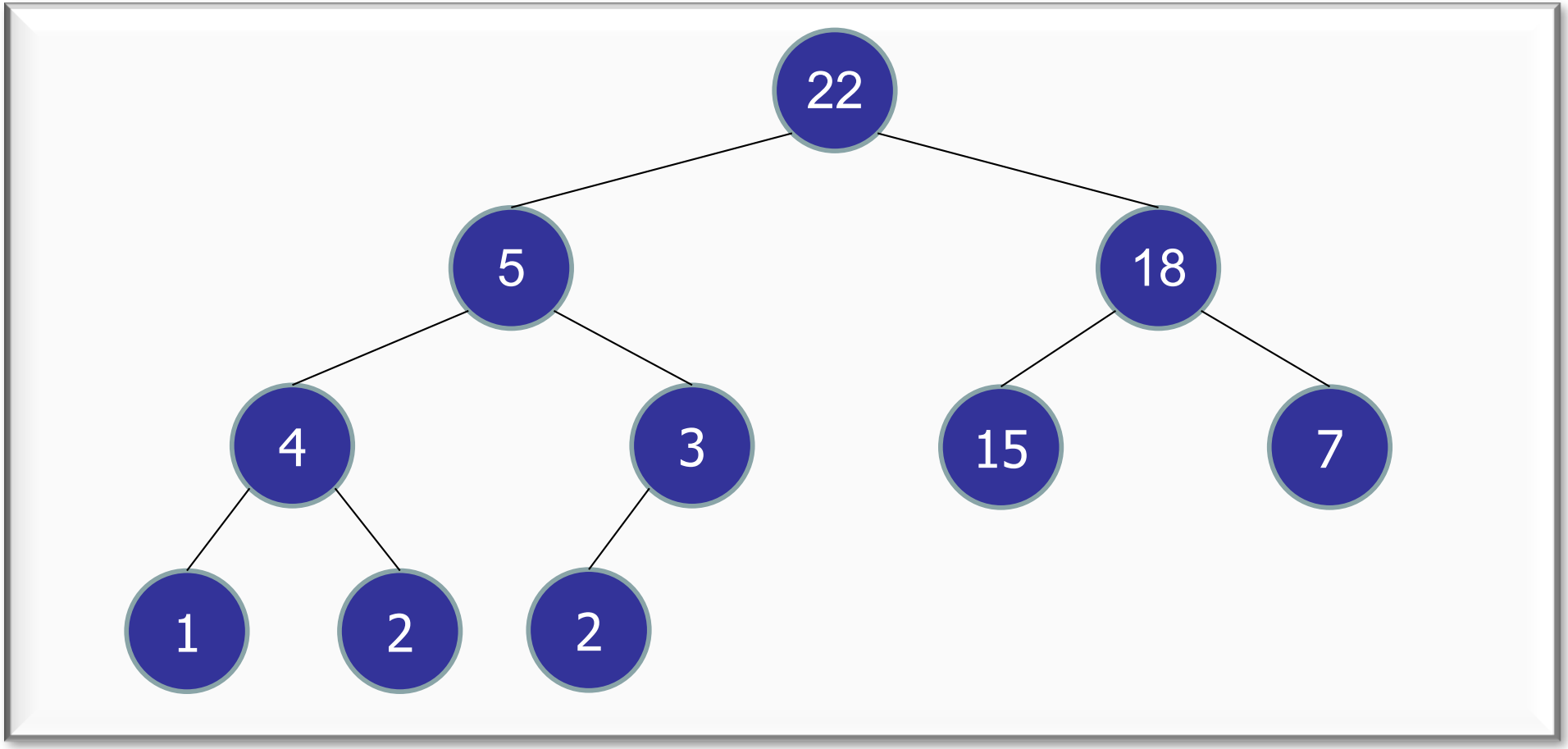
Inserting in a Heap

```
bubbleDown(Node v)
  while (!leaf(v)) {
     leftP = priority(left(v));
     rightP = priority(right(v));
     maxP = max(leftP, rightP, priority(v));
     if (leftP == max) {
           swap(v, left(v));
           v = left(v); }
     else if (rightP == max) {
           swap(v, right(v));
           v = right(v); }
     else return;
  }
```

Inserting in a Heap

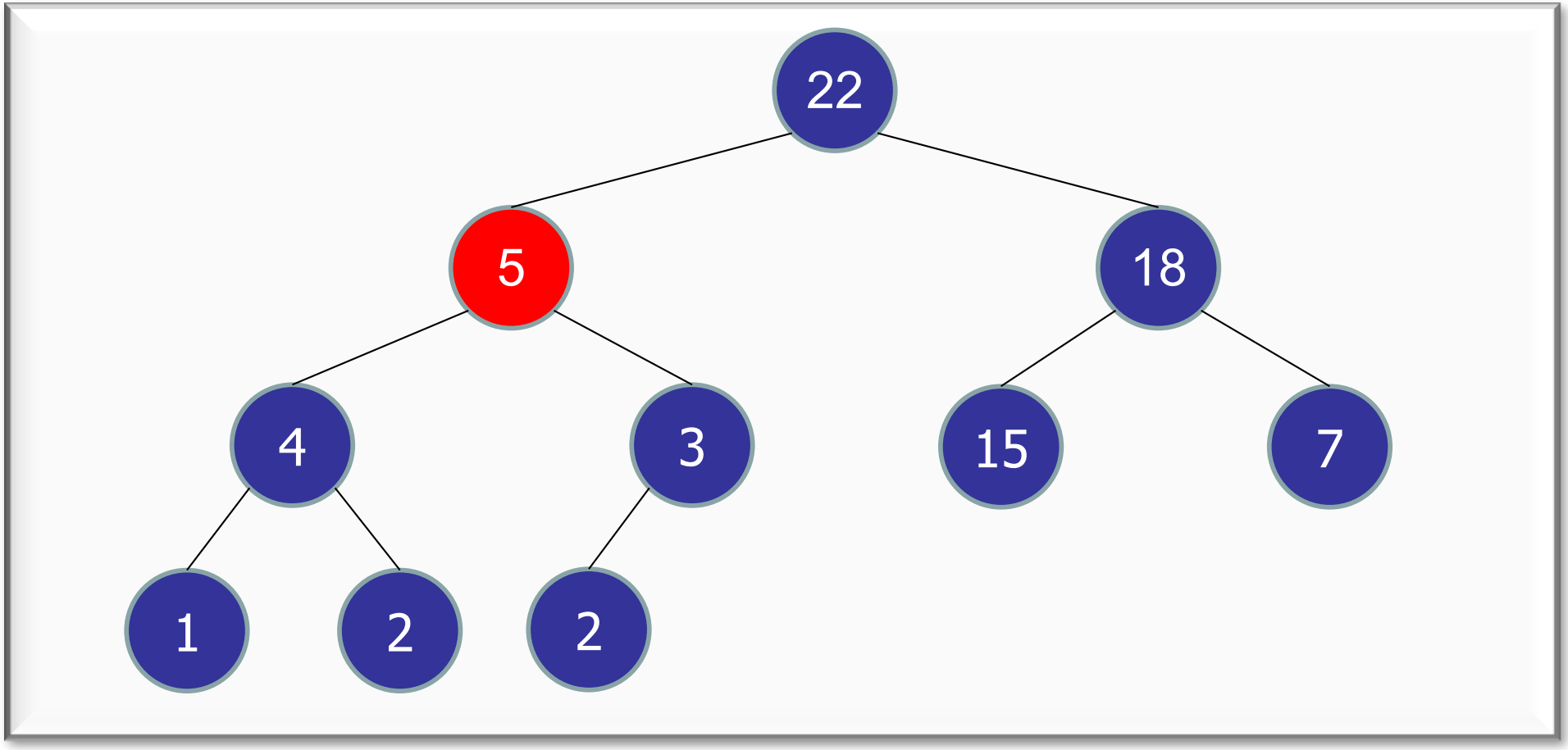
`decreaseKey(. . .)`:

- On completion, heap order is restored.
- Complete binary tree.



Inserting in a Heap

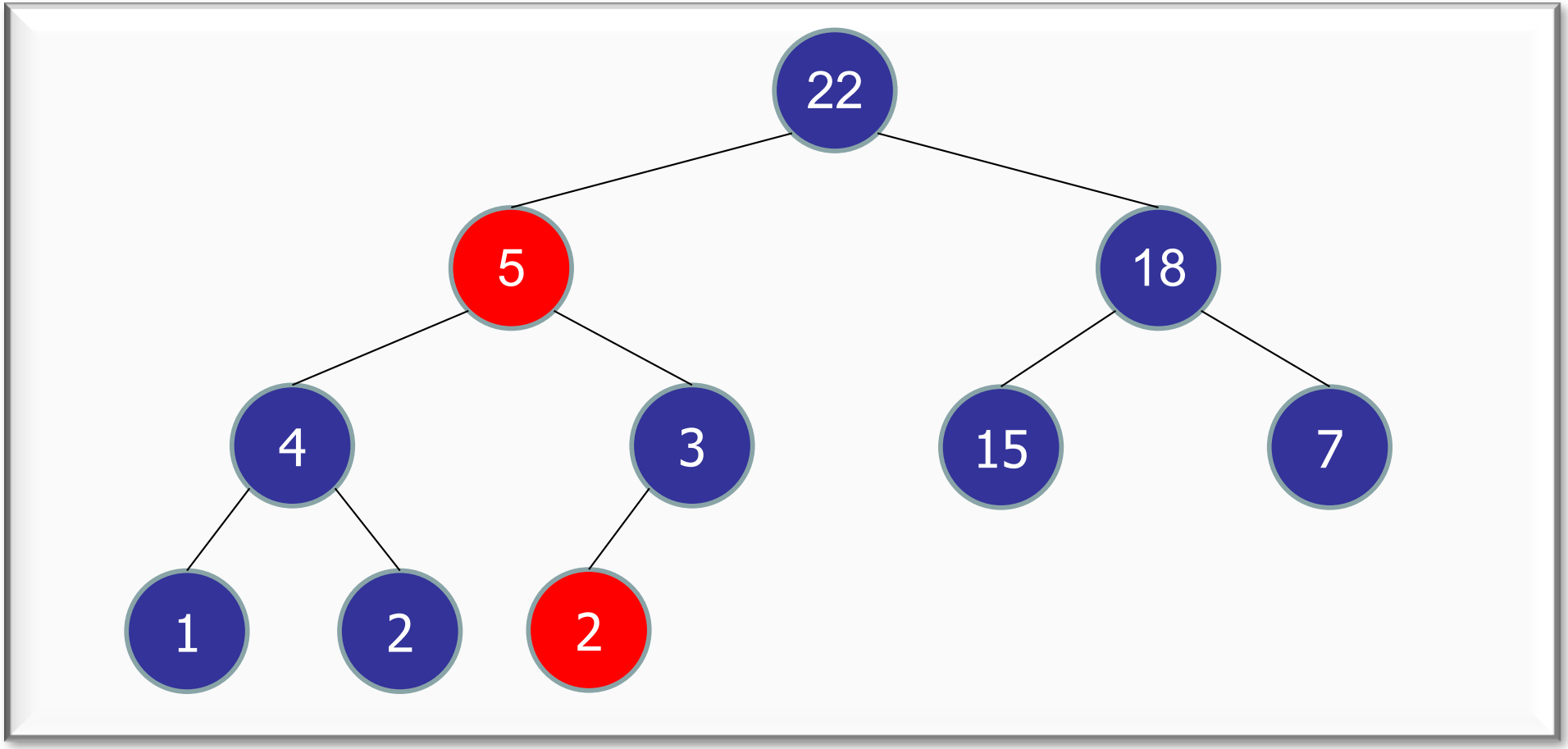
`delete(5)` :



Inserting in a Heap

`delete(5)` :

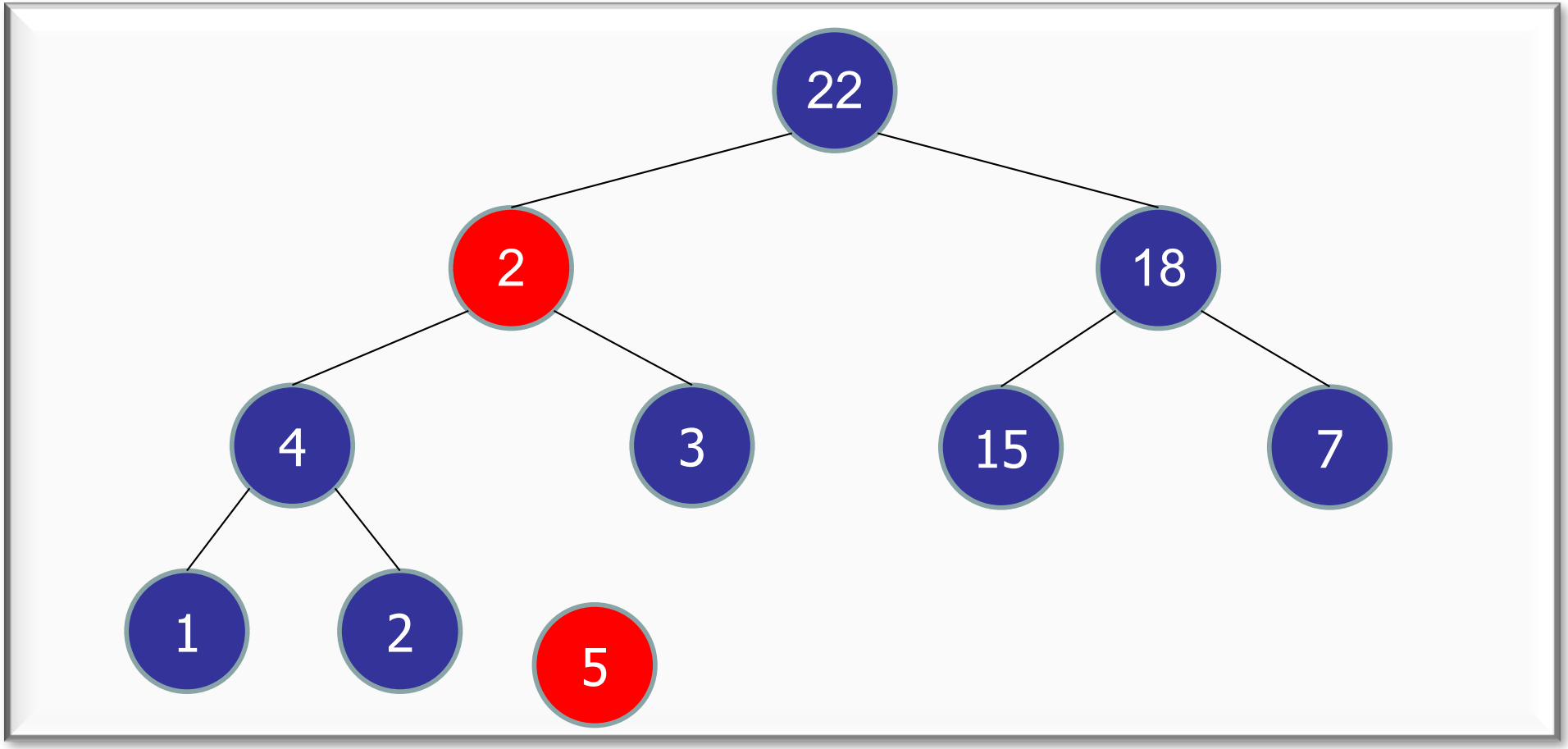
- swap(5, last())



Inserting in a Heap

`delete(5)` :

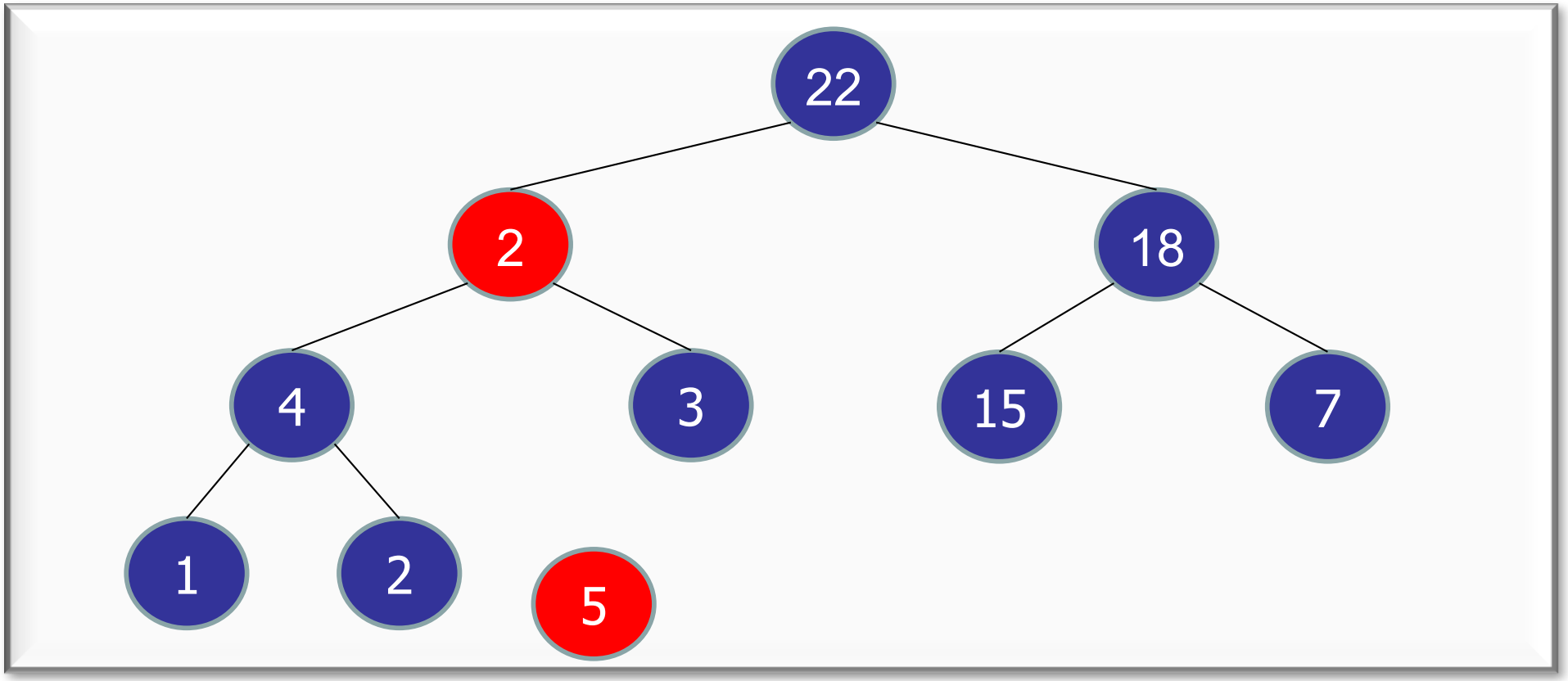
- swap(5, last())
- remove(last())



Inserting in a Heap

`delete(5)` :

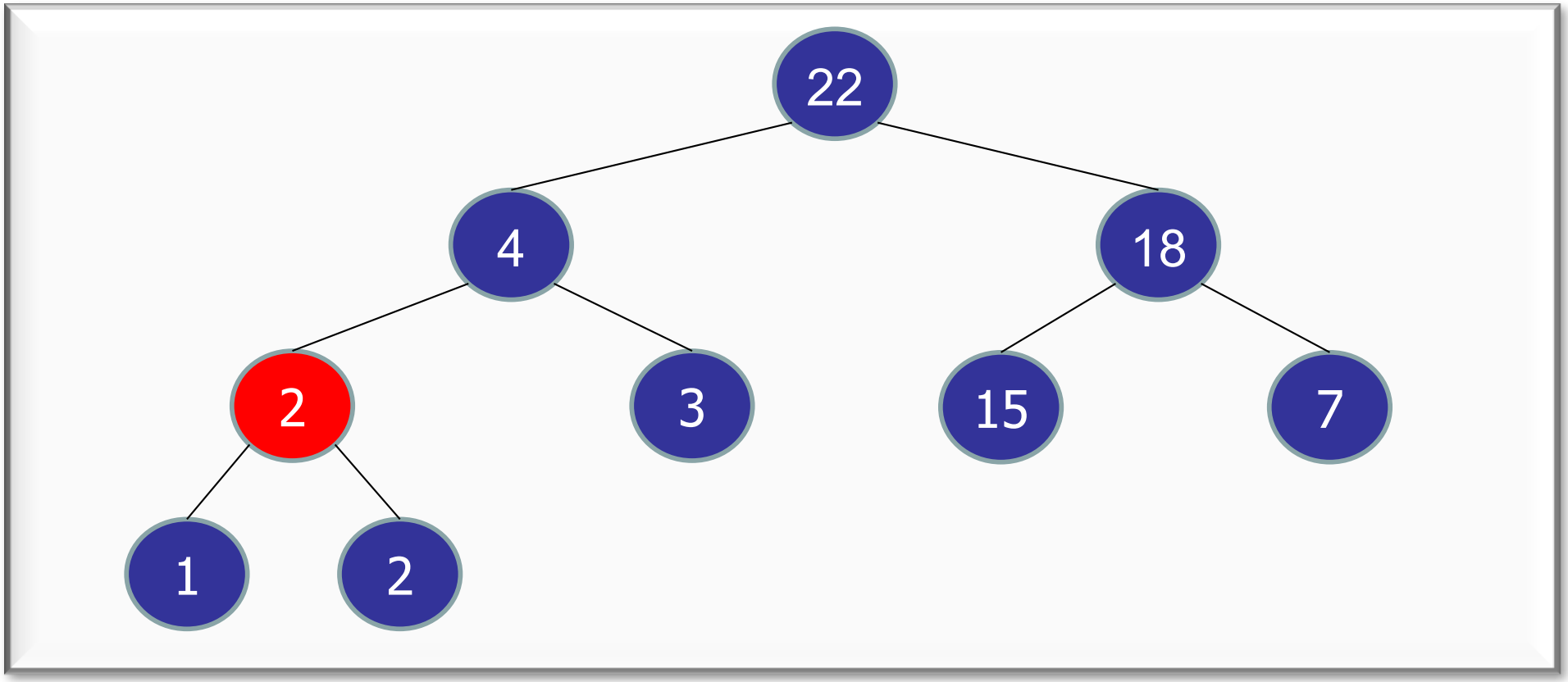
- swap(5, last())
- remove(last())
- bubbleDown(2)



Inserting in a Heap

`delete(5)` :

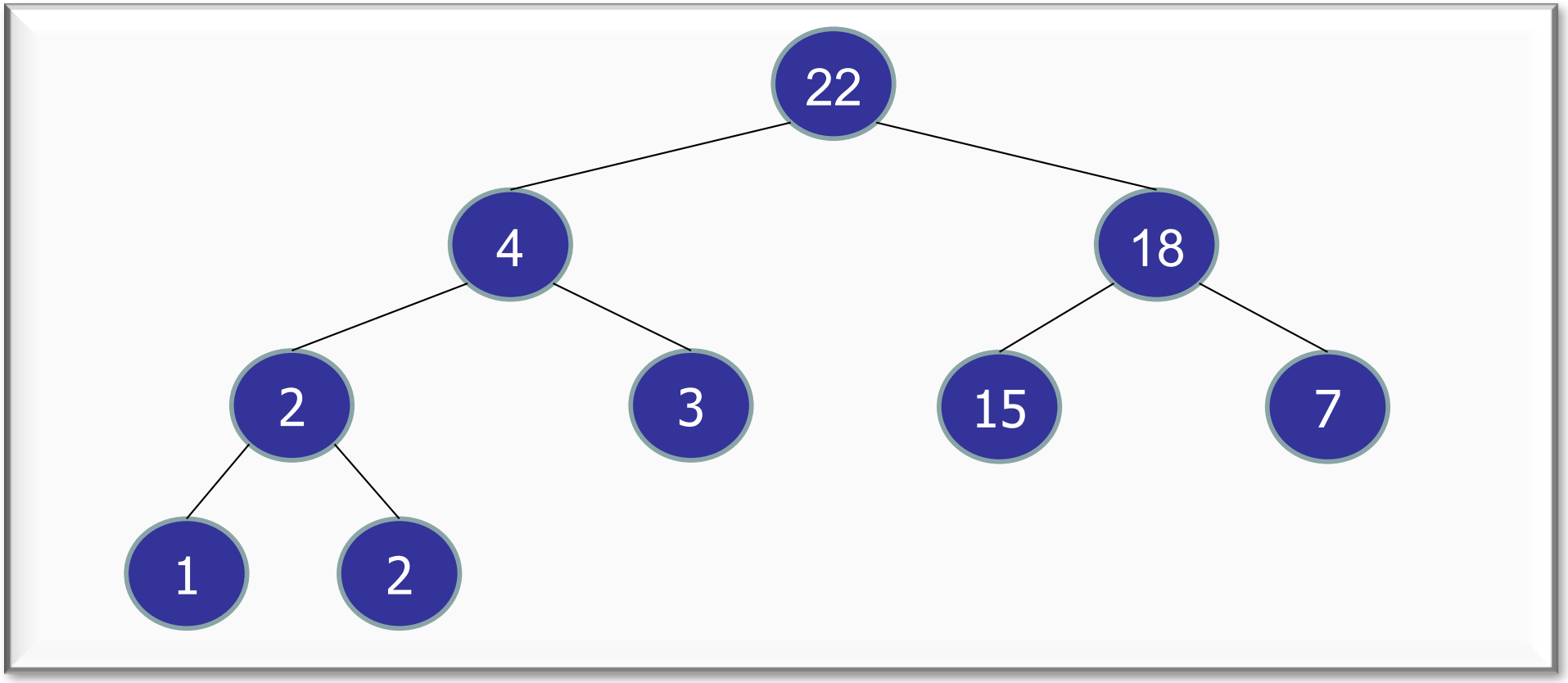
- swap(5, last())
- remove(last())
- bubbleDown(2)



Inserting in a Heap

`delete(5)` :

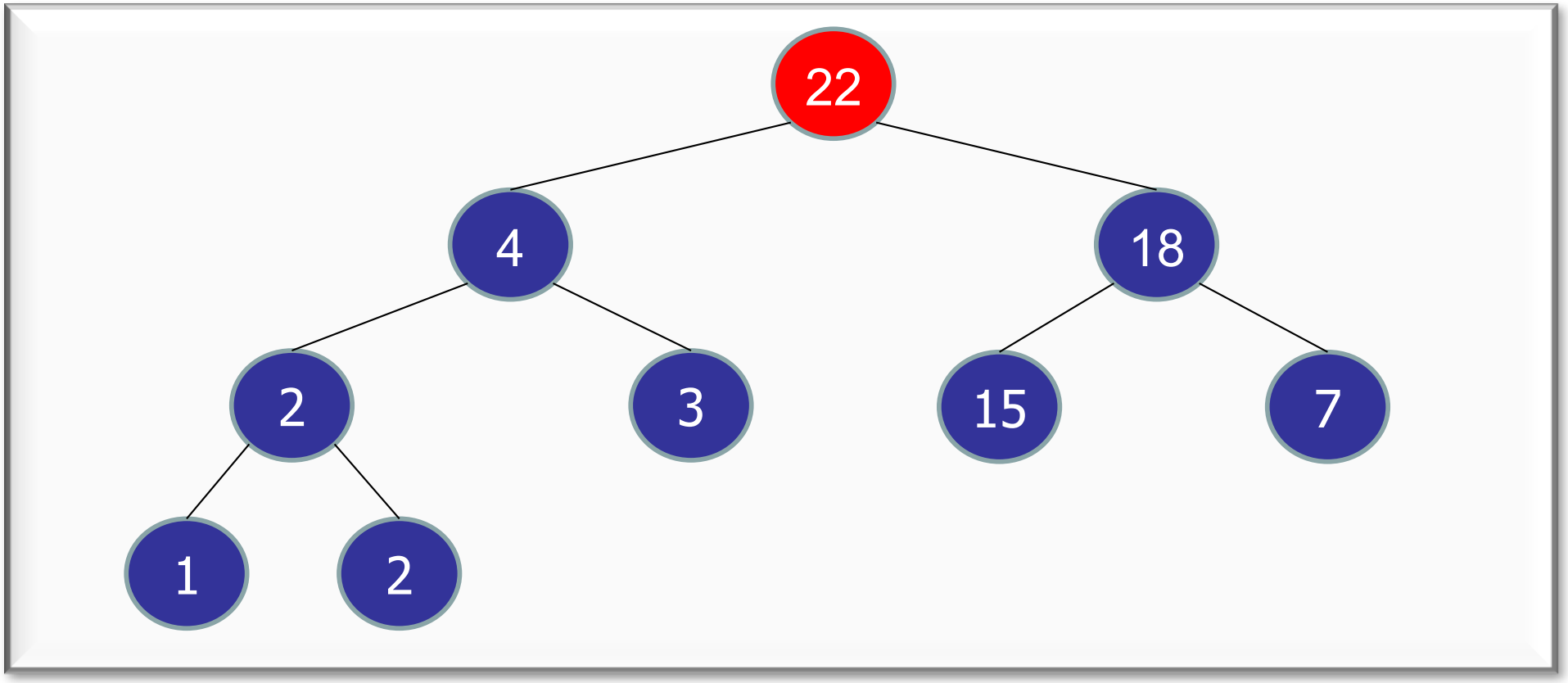
- swap(5, last())
- remove(last())
- bubbleDown(2)



Inserting in a Heap

`extractMax()` :

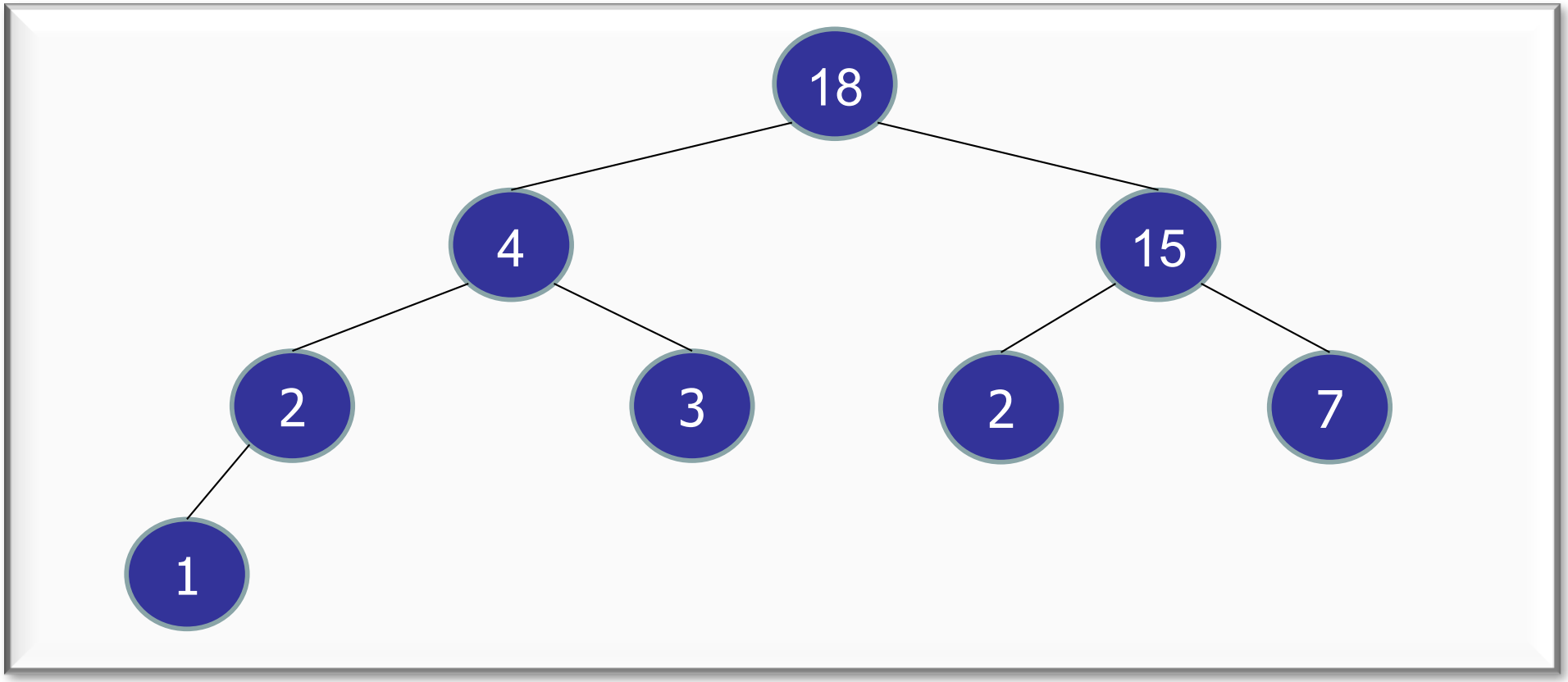
- Node v = root;
- delete(root);



Inserting in a Heap

`extractMax()` :

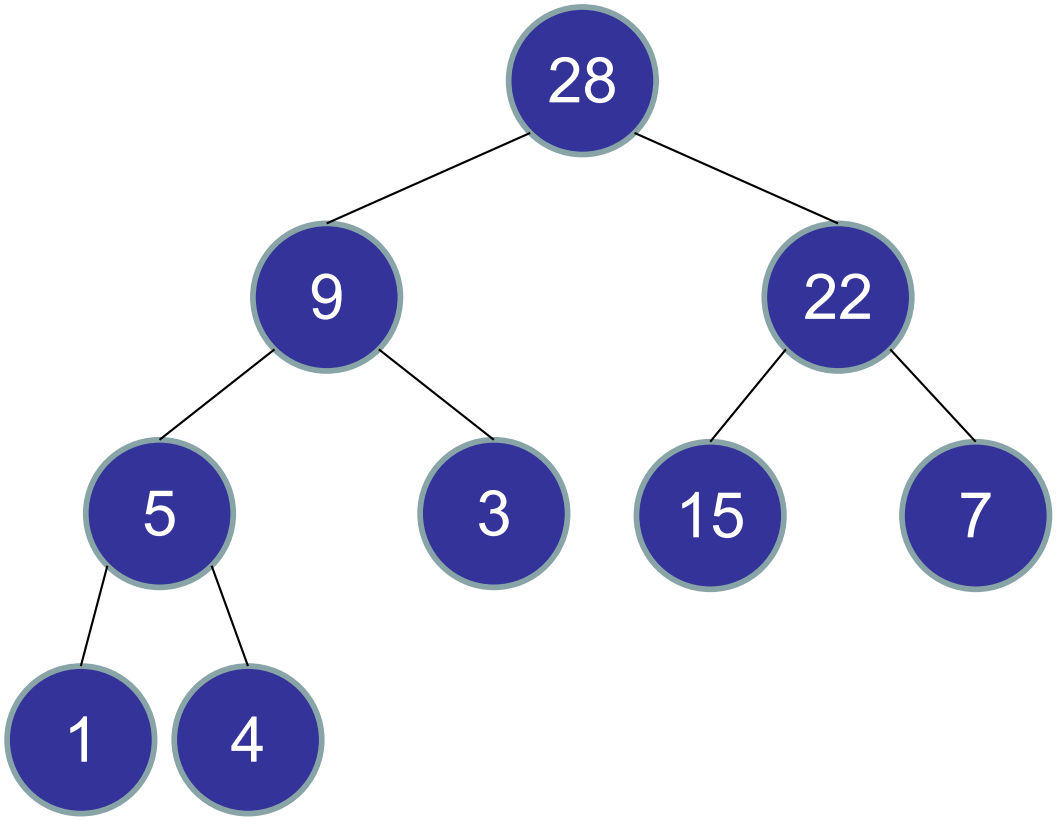
- Node v = root;
- delete(root);



(Max) Priority Queue

Heap Operations: O(log n)

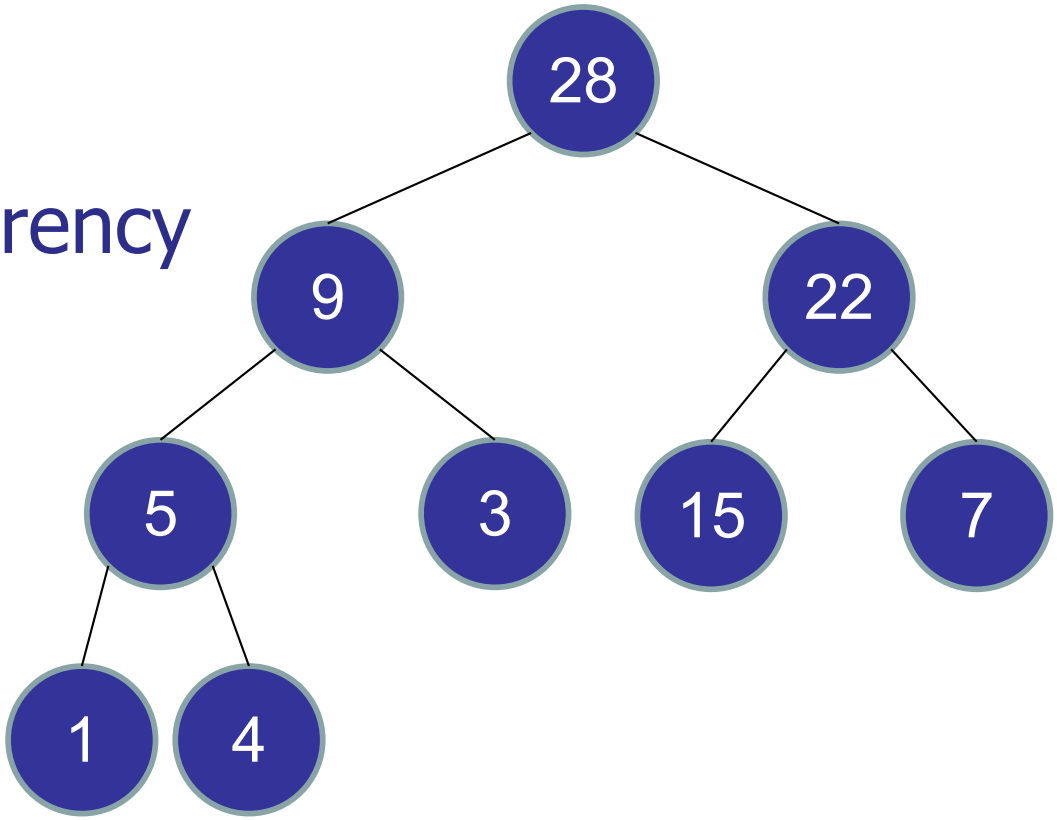
- insert
- extractMax
- increaseKey
- decreaseKey
- delete



(Max) Priority Queue

Heap vs. AVL Tree

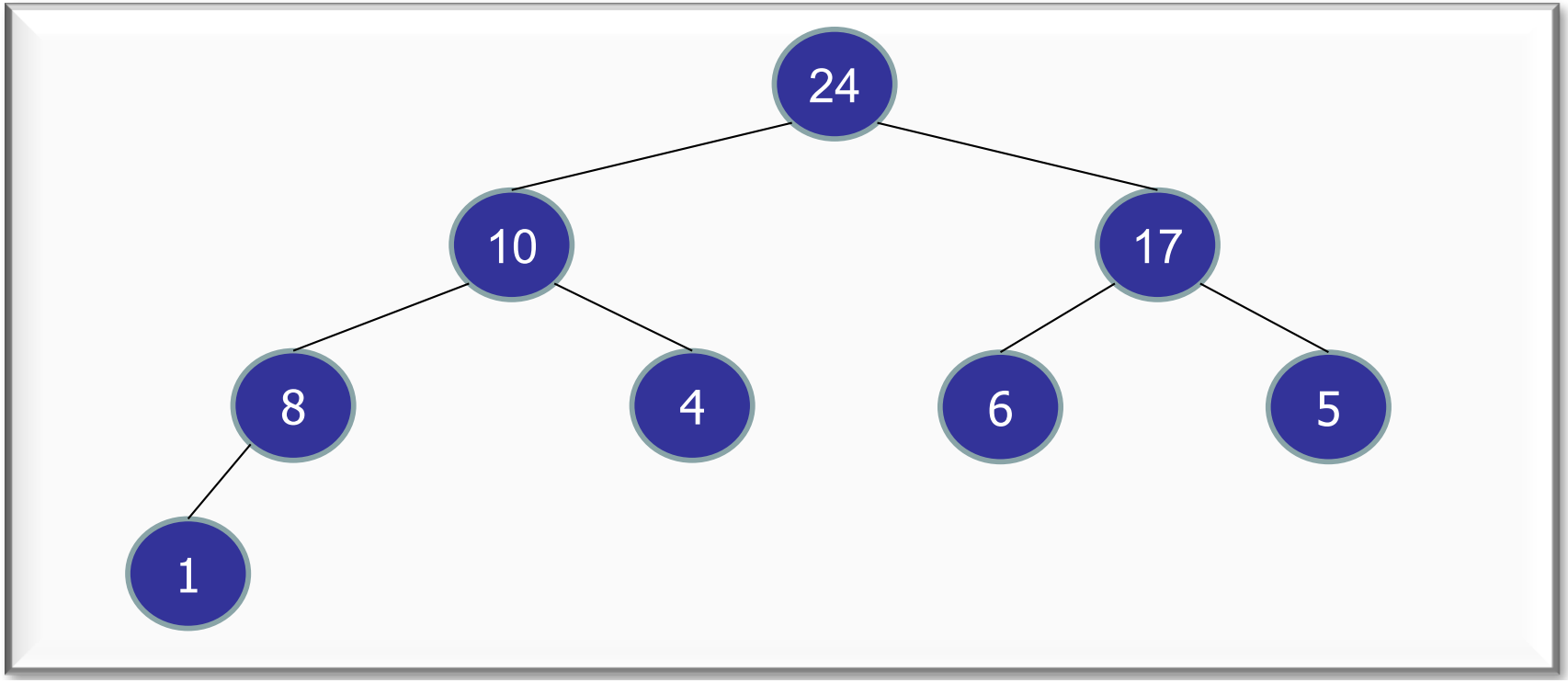
- Same cost for operations
- Slightly simpler
  - No rotations
- Slightly better concurrency



Store Tree in an Array

Map each node in complete binary tree into a slot in an array.

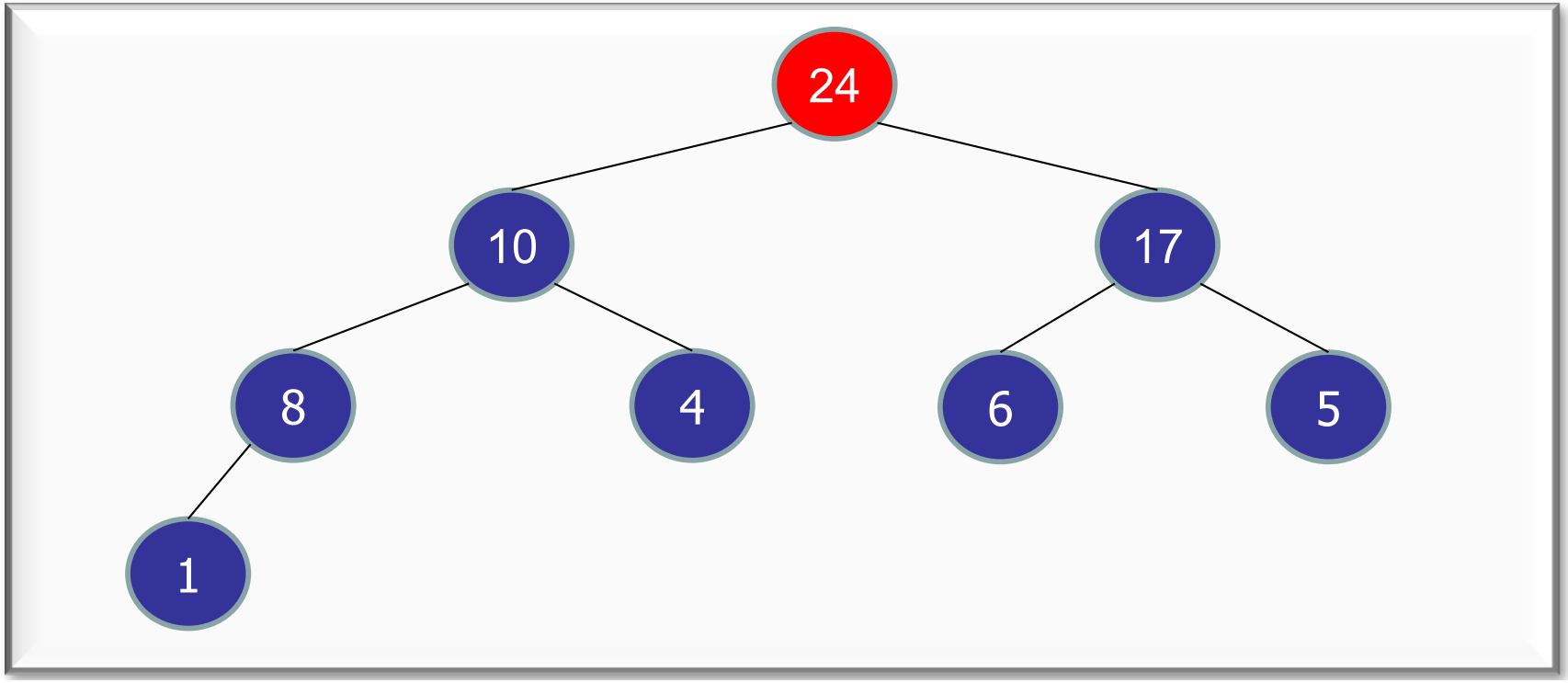
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	7	1	



Store Tree in an Array

Map each node in complete binary tree into a slot in an array.

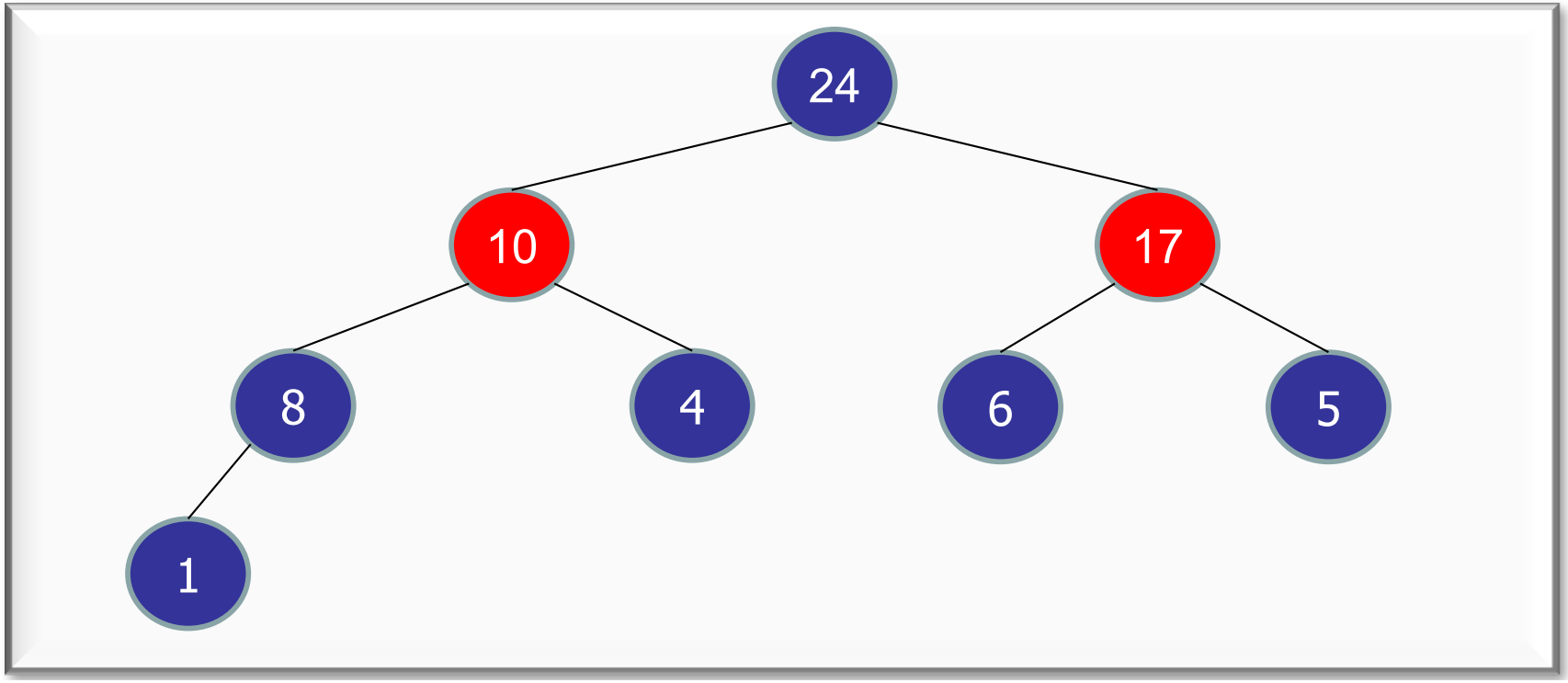
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	7	1	



Store Tree in an Array

Map each node in complete binary tree into a slot in an array.

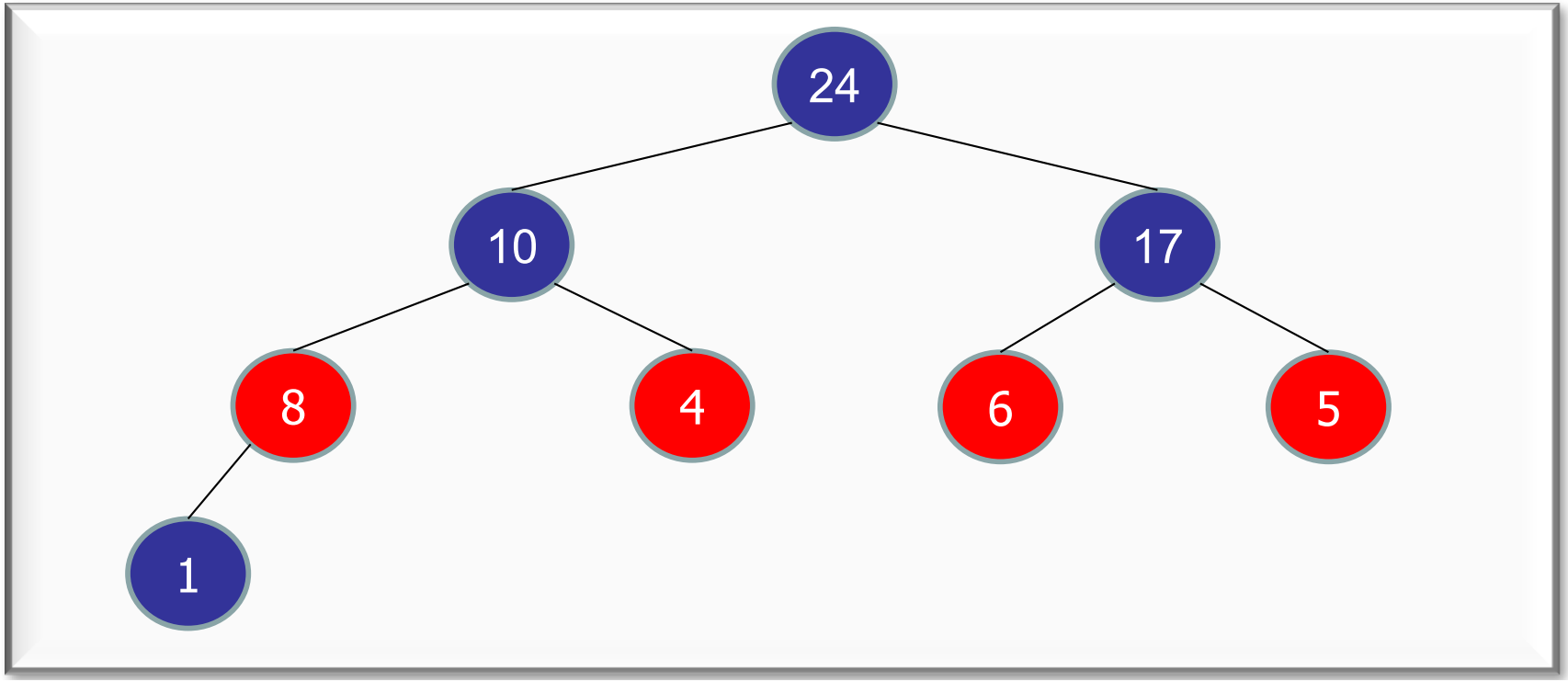
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	7	1	



Store Tree in an Array

Map each node in complete binary tree into a slot in an array.

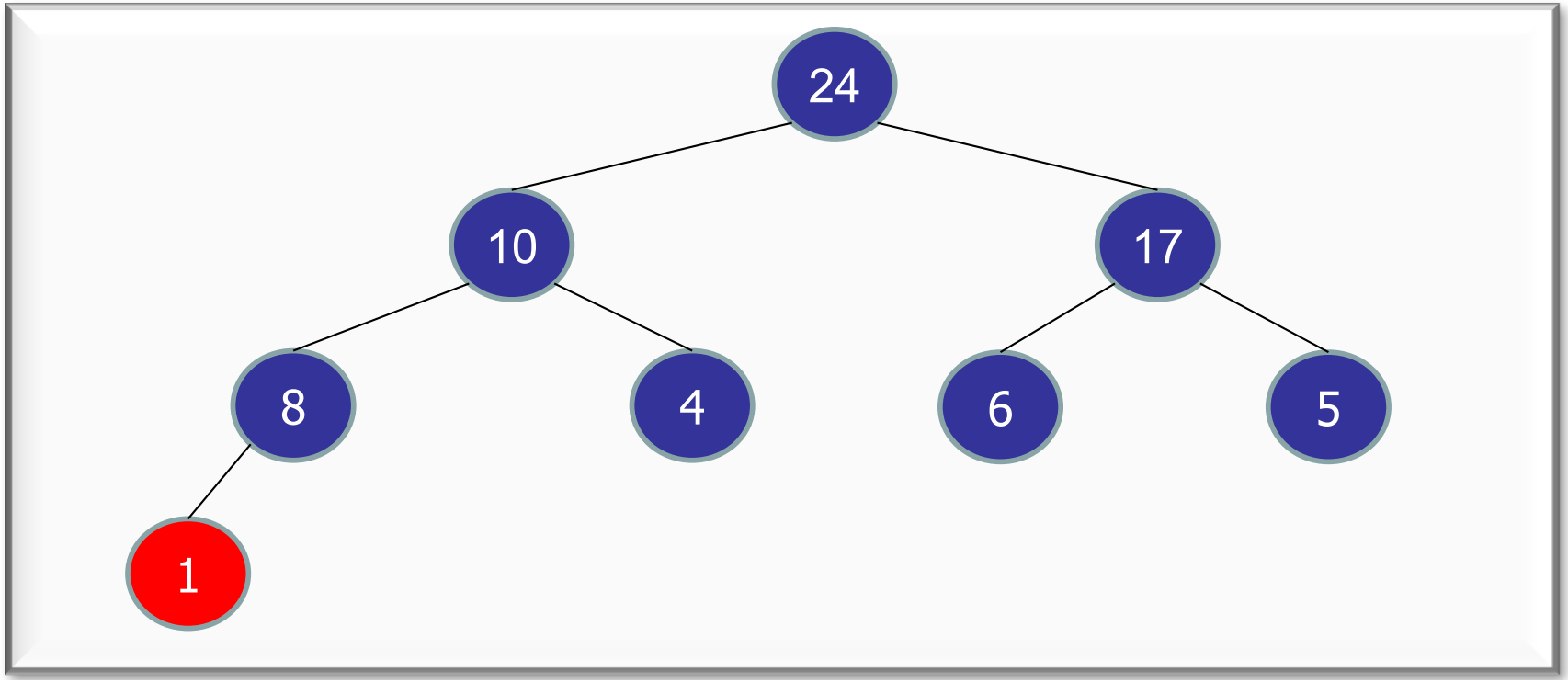
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	



Store Tree in an Array

Map each node in complete binary tree into a slot in an array.

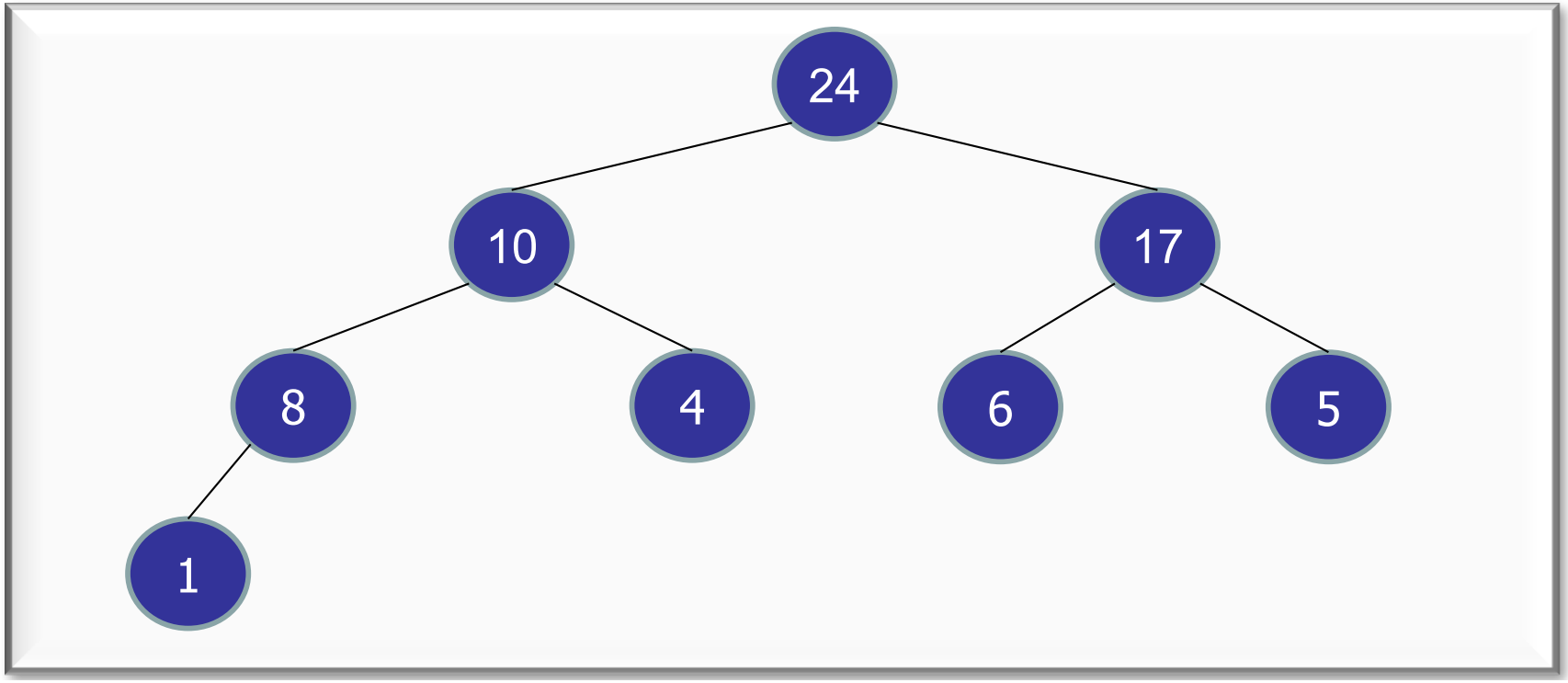
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	



Store Tree in an Array

`insert(15) :`

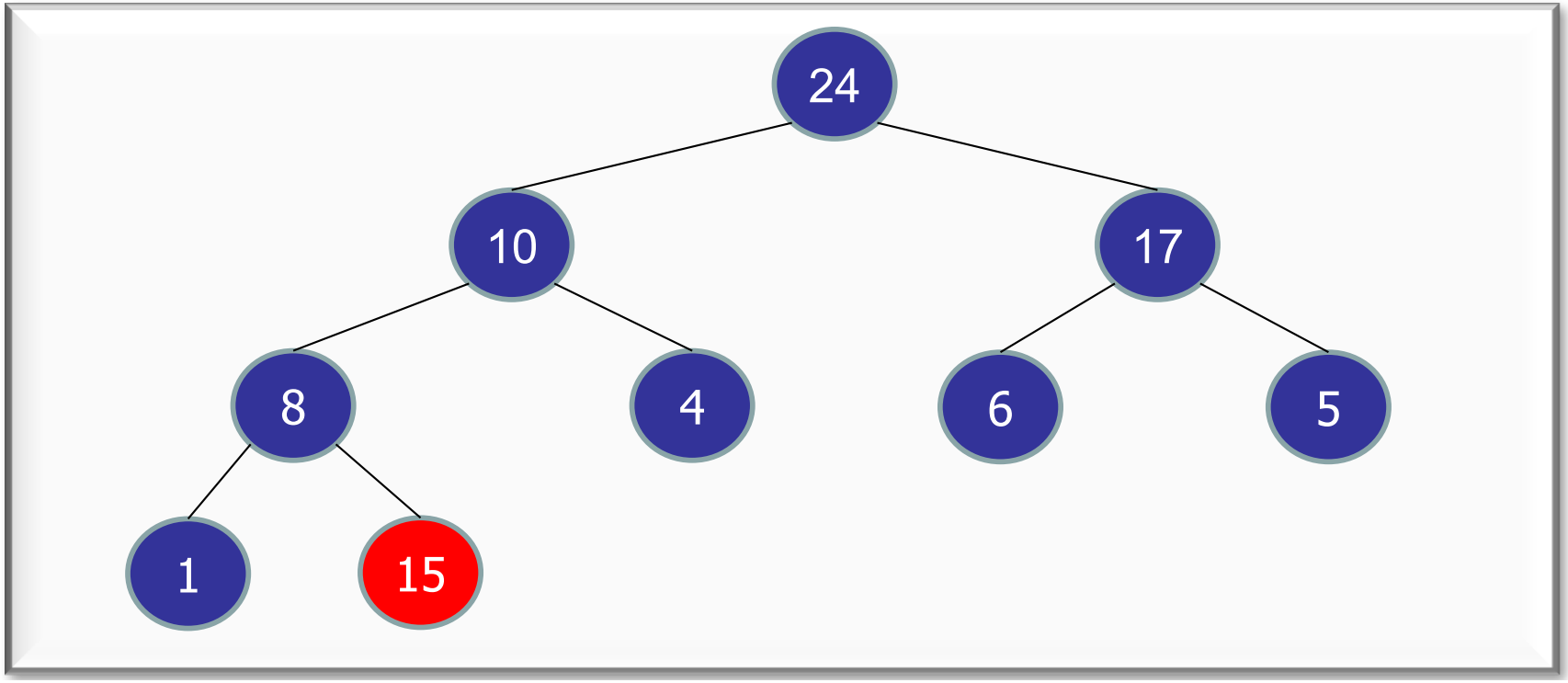
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	



Store Tree in an Array

`insert(15) :`

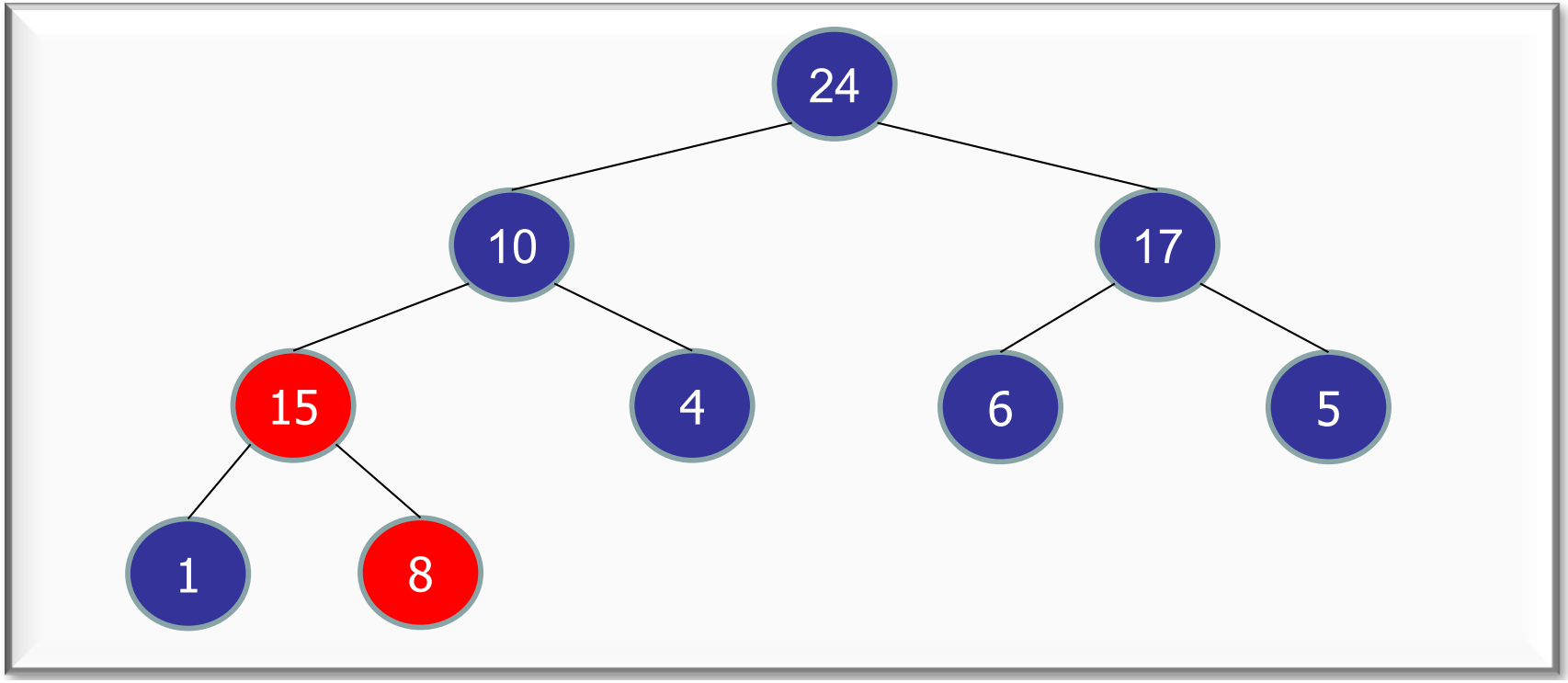
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	15



Store Tree in an Array

`insert(15) :`

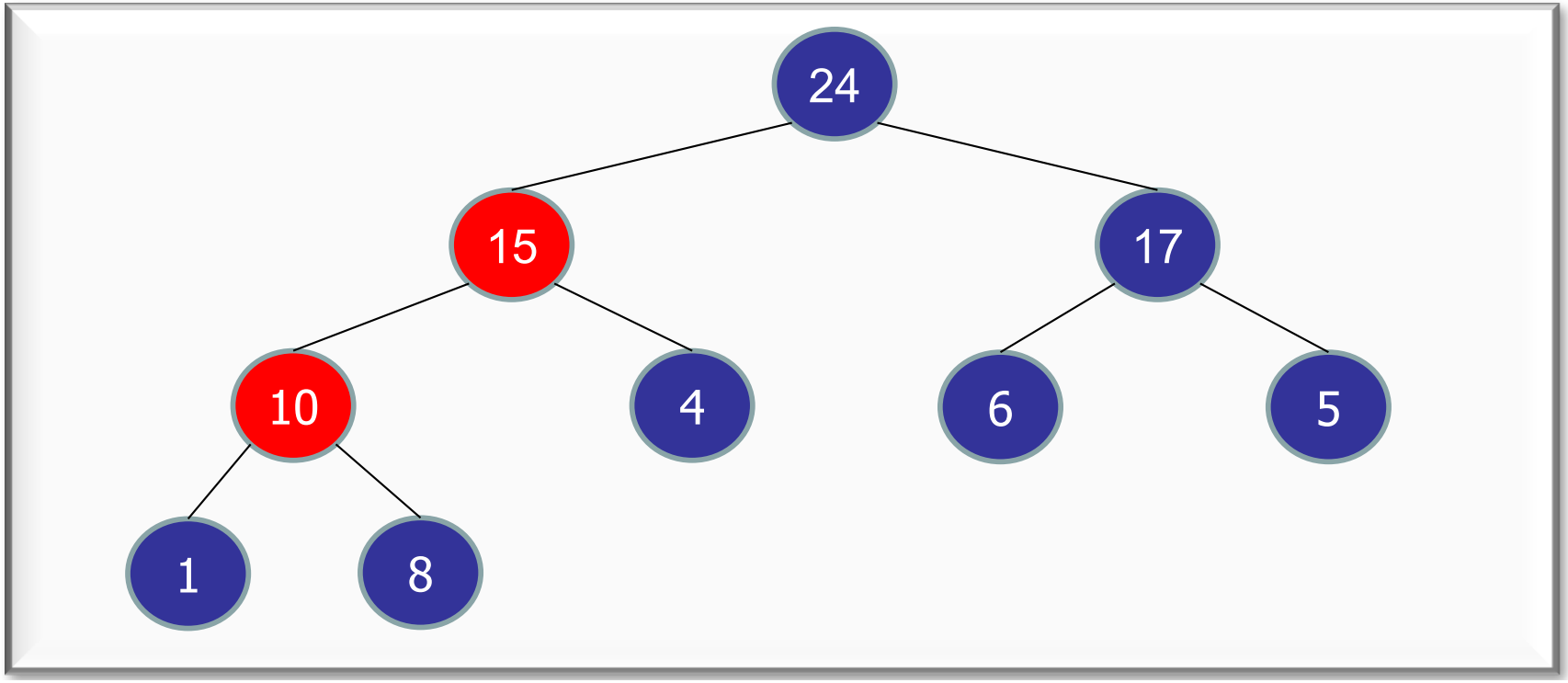
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	15	4	6	5	1	8



Store Tree in an Array

`insert(15) :`

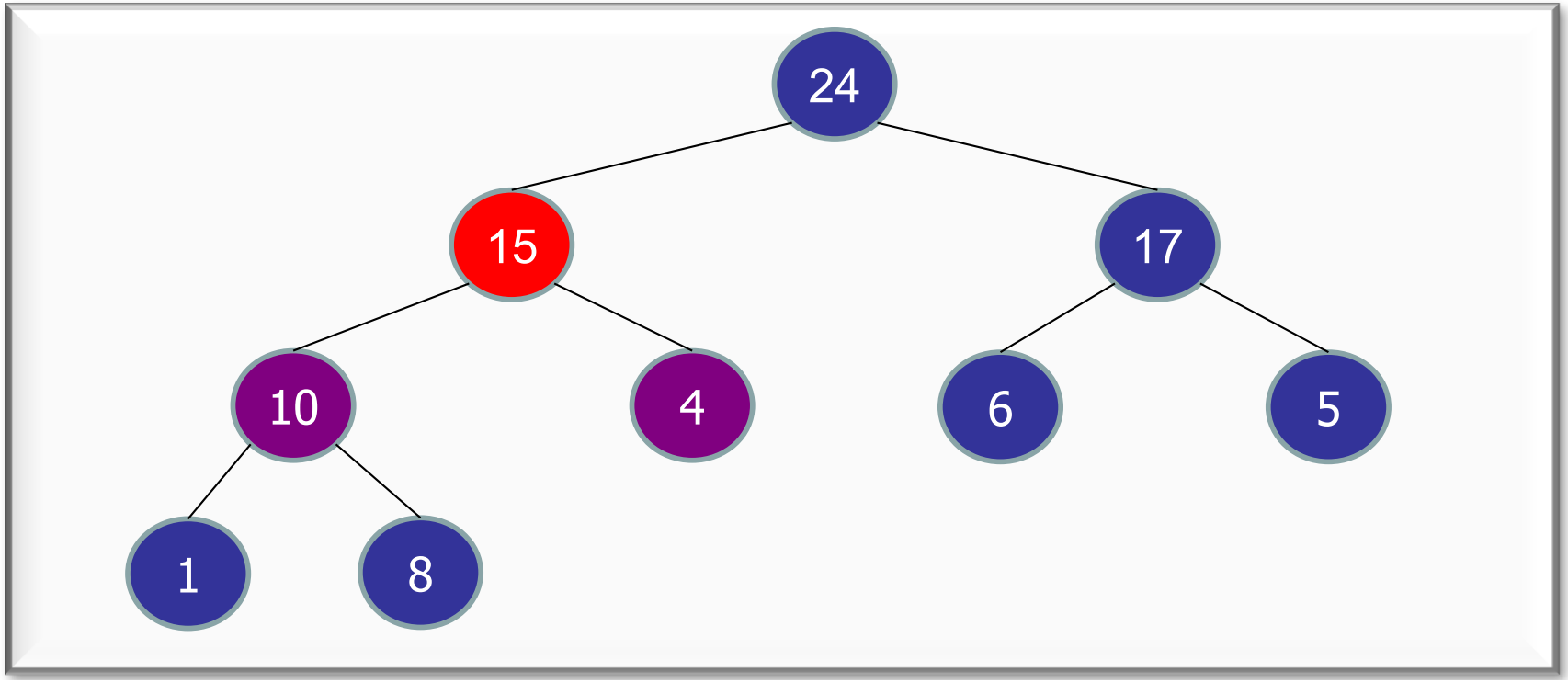
array slot	0	1	2	3	4	5	6	7	8
priority	24	15	17	10	4	6	5	1	8



Store Tree in an Array

```
left(x)  = 2x+1
right(x) = 2x+2
```

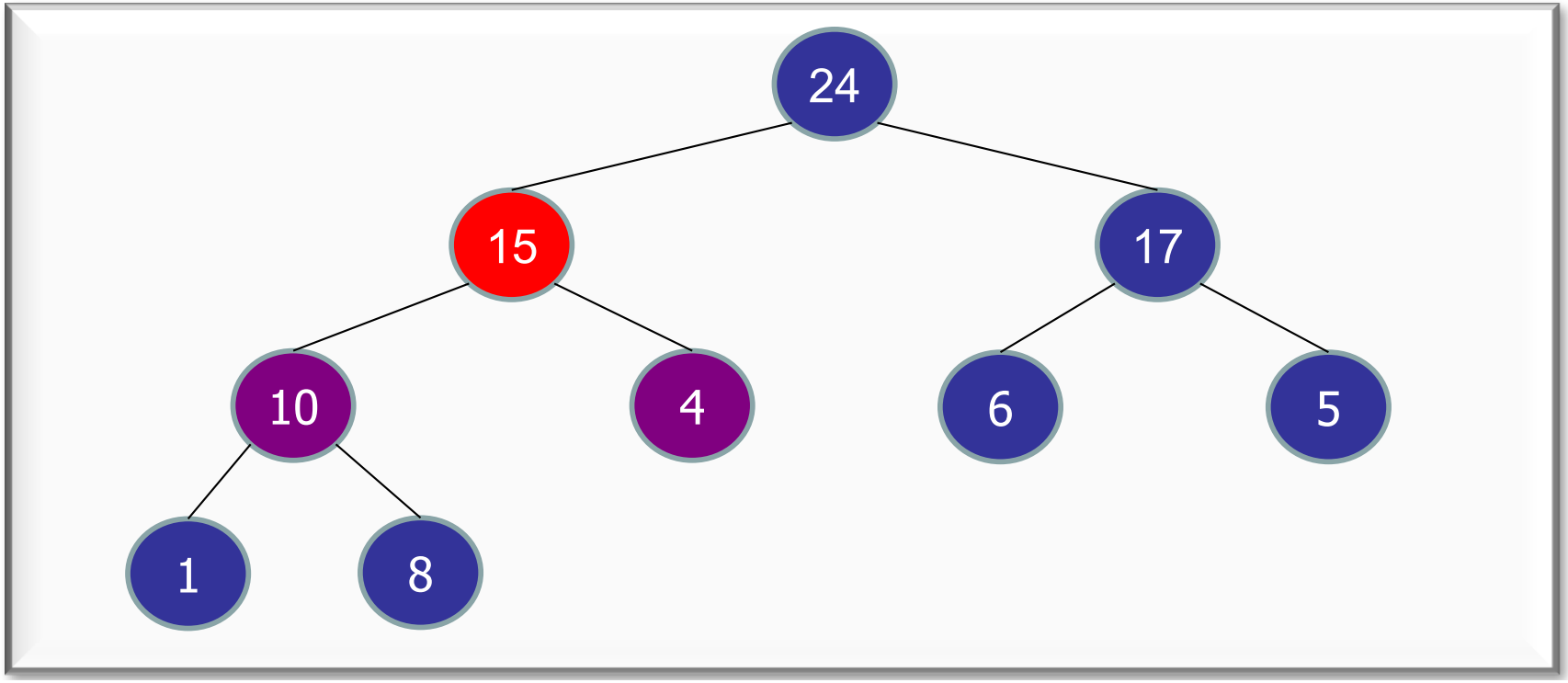
array slot	0	1	2	3	4	5	6	7	8
priority	24	15	17	10	4	6	5	1	8



Store Tree in an Array

```
parent(x)  = floor((x-1)/ 2)
```

array slot	0	1	2	3	4	5	6	7	8
priority	24	15	17	10	4	6	5	1	8



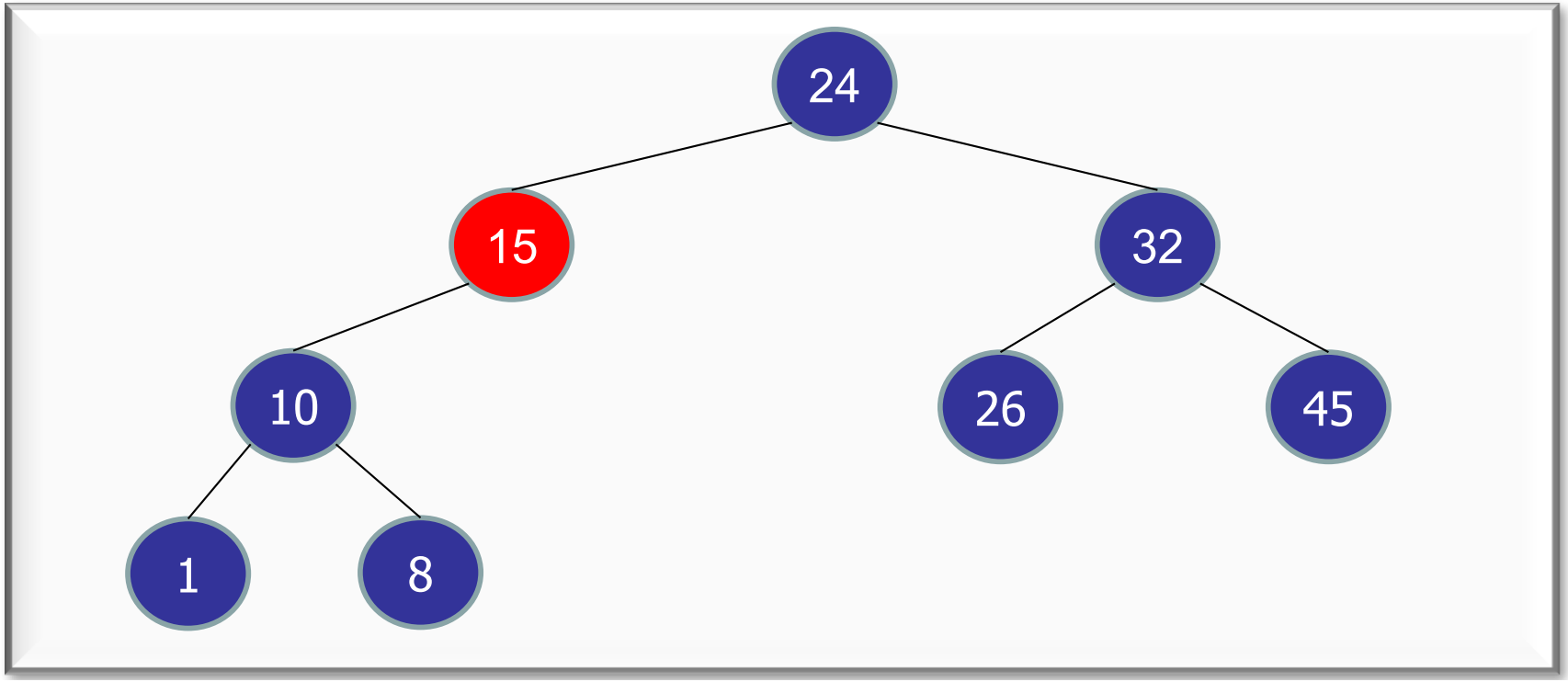
Why not store an AVL tree in an array?

1. Too much wasted space.
2. Too expensive to calculate left/right/parent.
3. ✓ Too slow to update.
4. You can store an AVL tree in an array.

Store Tree in an Array

```
right-rotate(15)
```

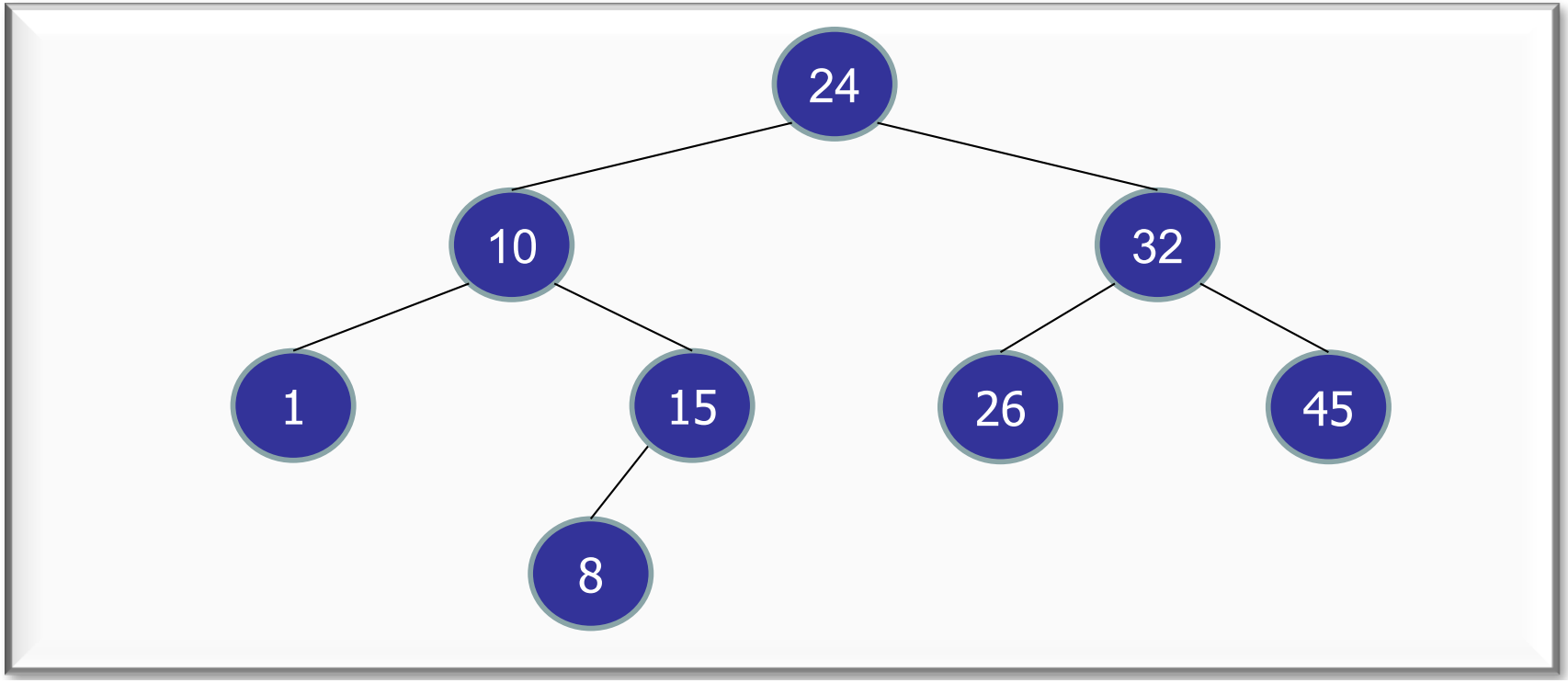
array slot	0	1	2	3	4	5	6	7	8
priority	24	15	32	10		26	45	1	8



Store Tree in an Array

```
right-rotate(15)
```

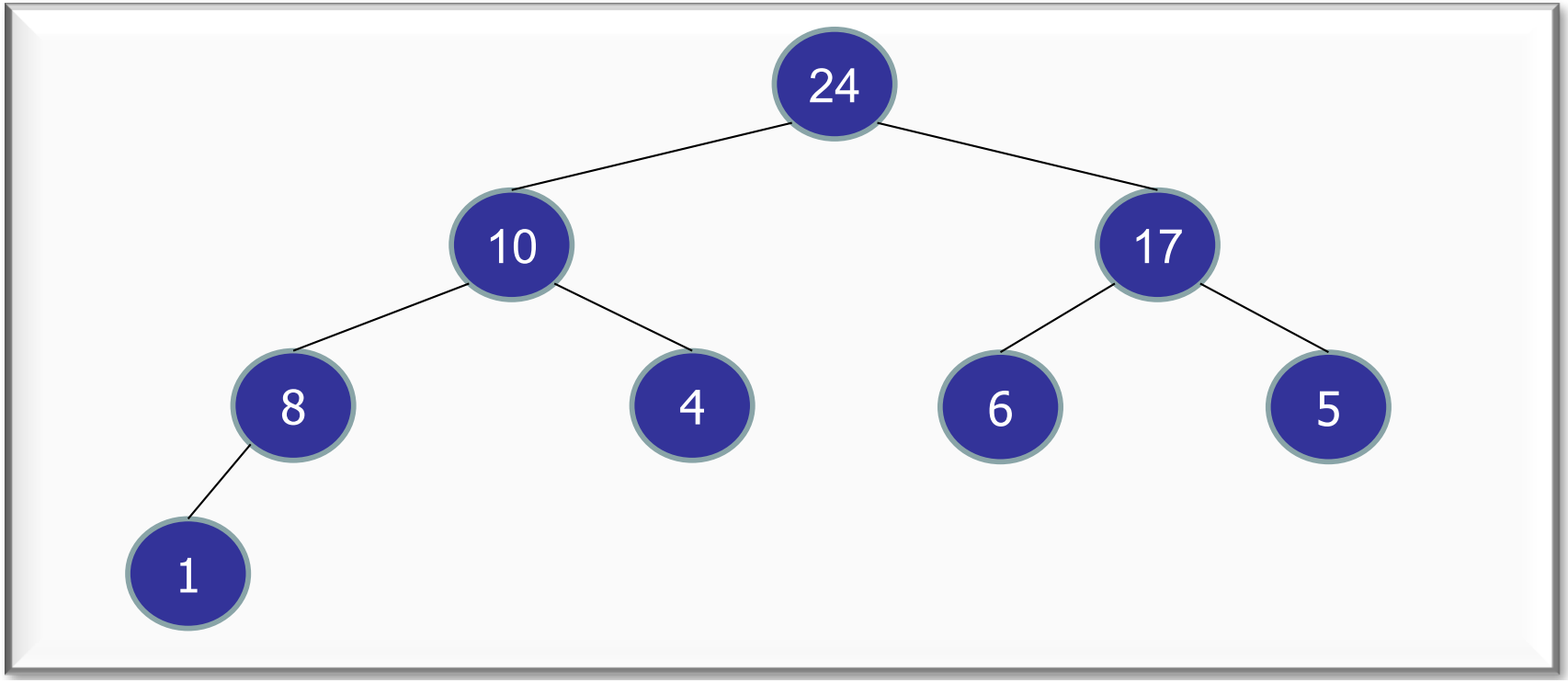
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	32	1	15	26	45	8	



Store Tree in an Array

Map each node in complete binary tree into a slot in an array.

array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	



HeapSort

Unsorted list:

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

HeapSort

Unsorted list:

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

Unsorted list ➔ Heap

array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	3

HeapSort

Unsorted list:

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

Unsorted list ➔ Heap

array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	3

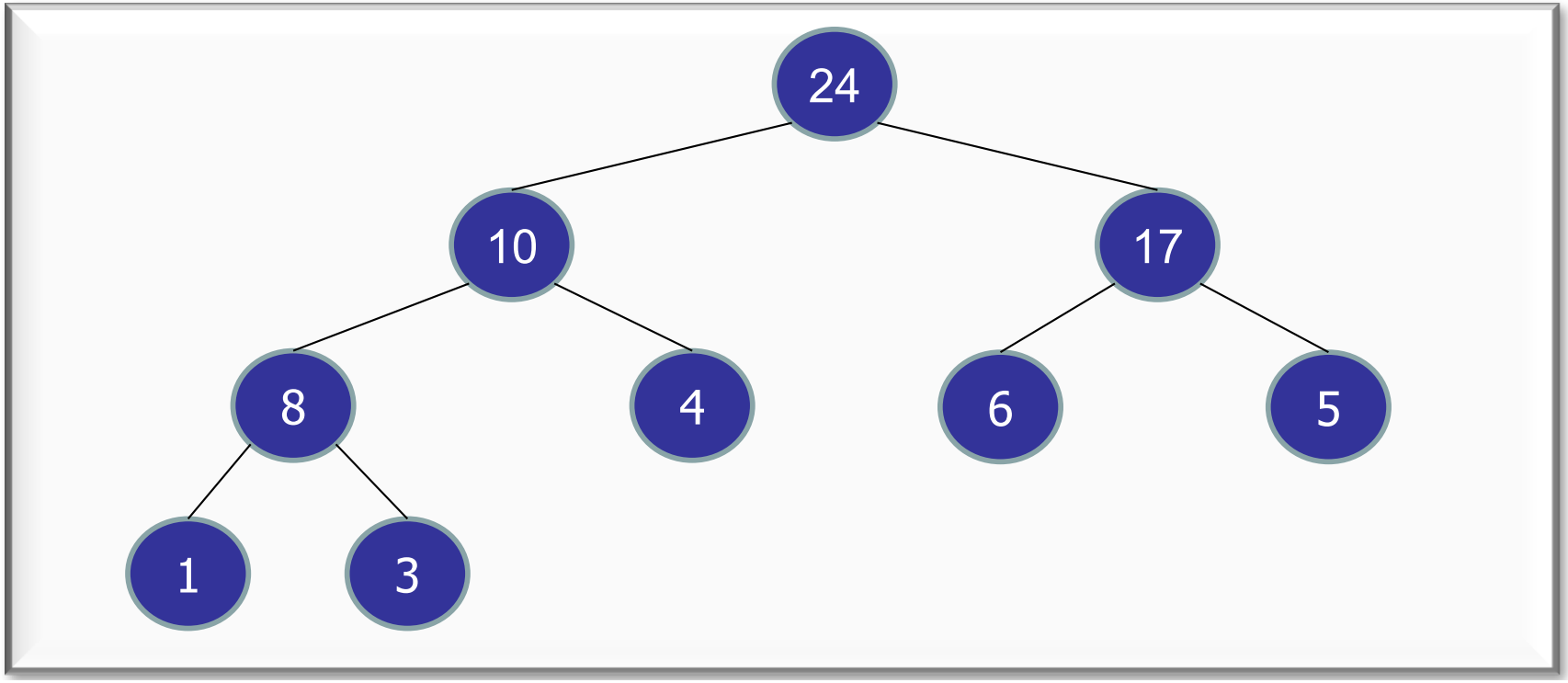
Heap ➔ Sorted list:

array slot	0	1	2	3	4	5	6	7	8
key	1	3	4	5	6	8	10	17	24

HeapSort

Heap ➔ Sorted list:

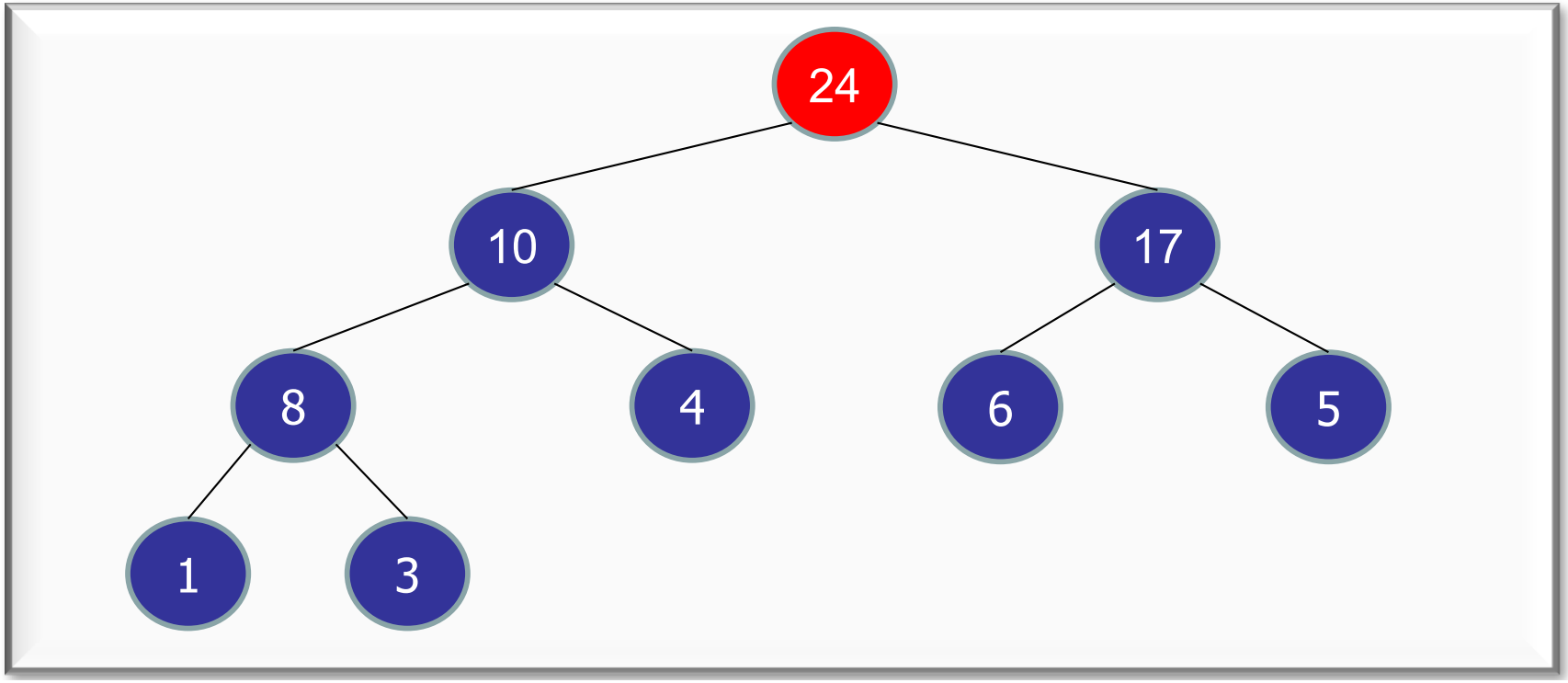
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	3



HeapSort

```
value = extractMax();
```

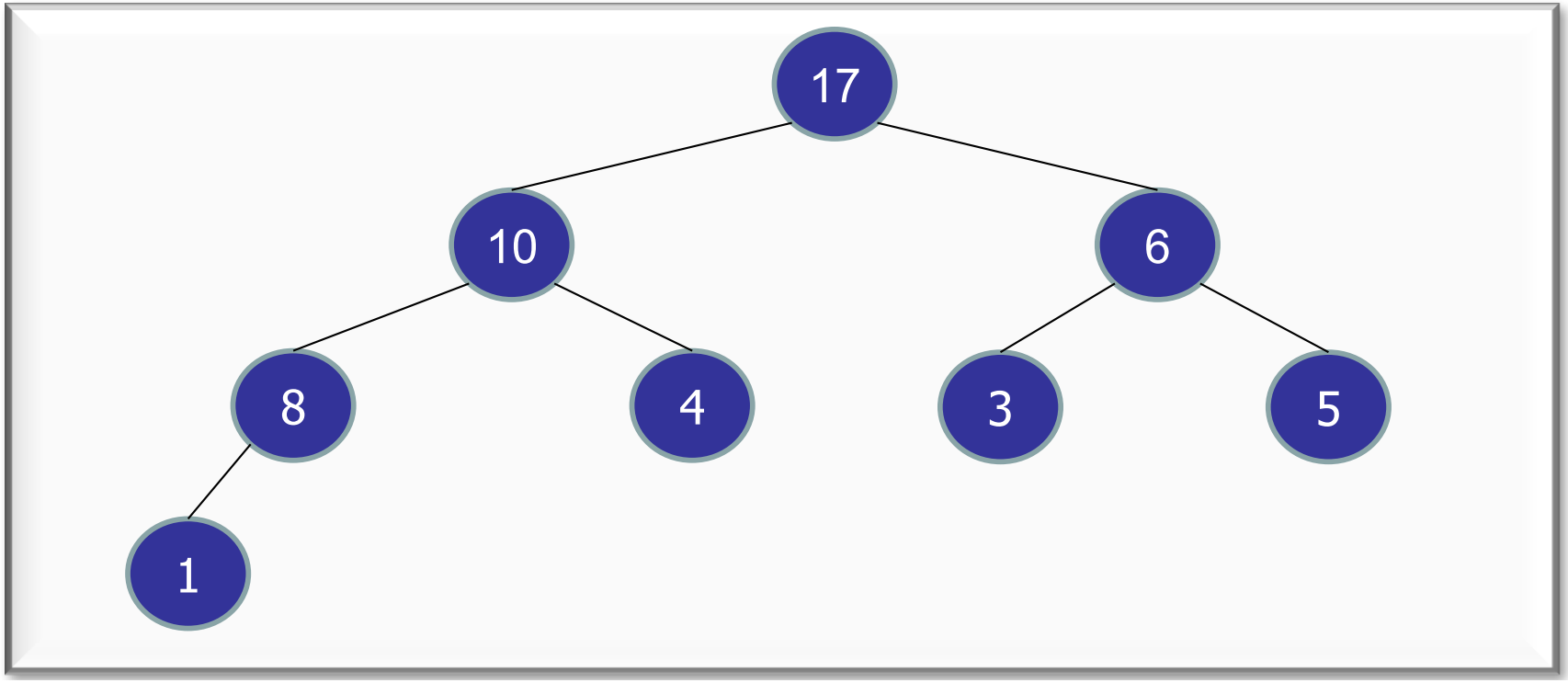
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	3



HeapSort

```
value = extractMax();
```

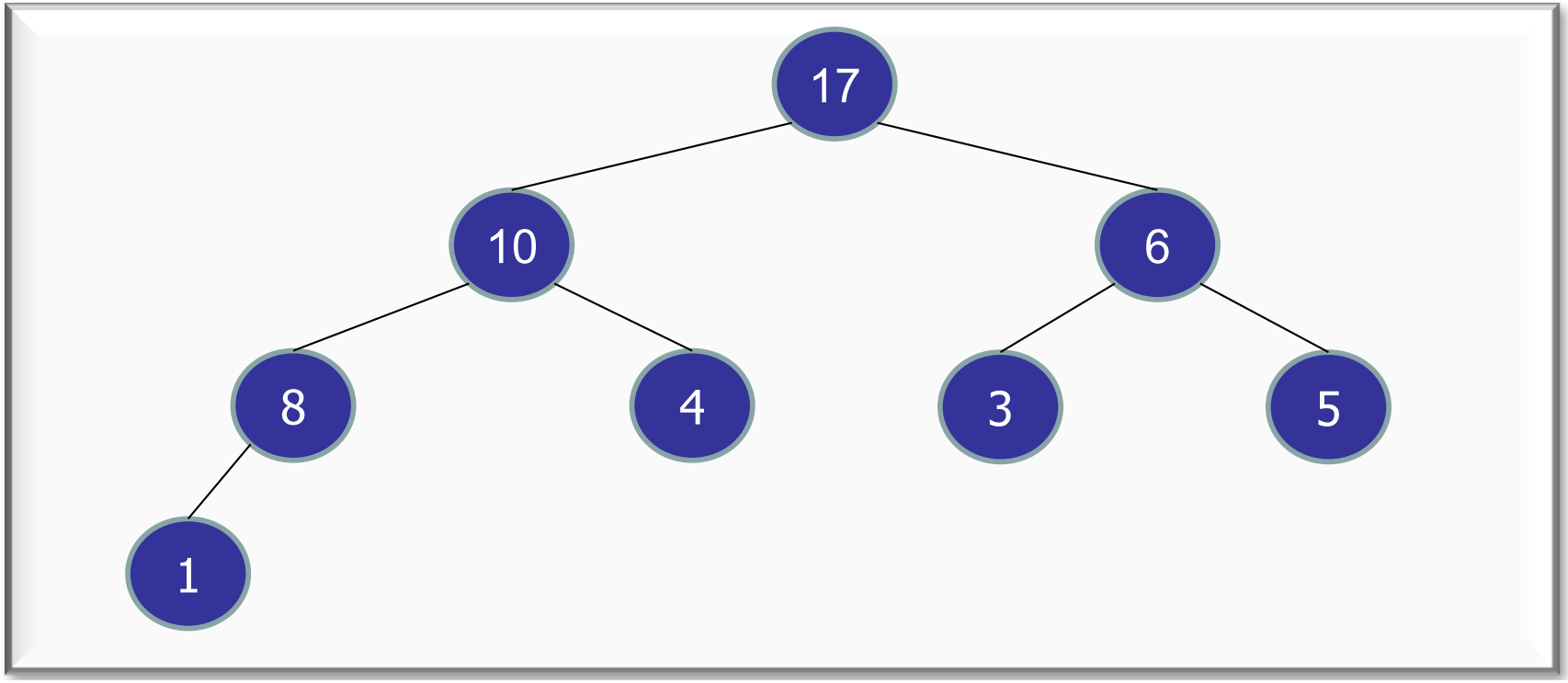
array slot	0	1	2	3	4	5	6	7	8
priority	17	10	6	8	4	3	5	1	



HeapSort

```
value = extractMax();
A[8] = value;
```

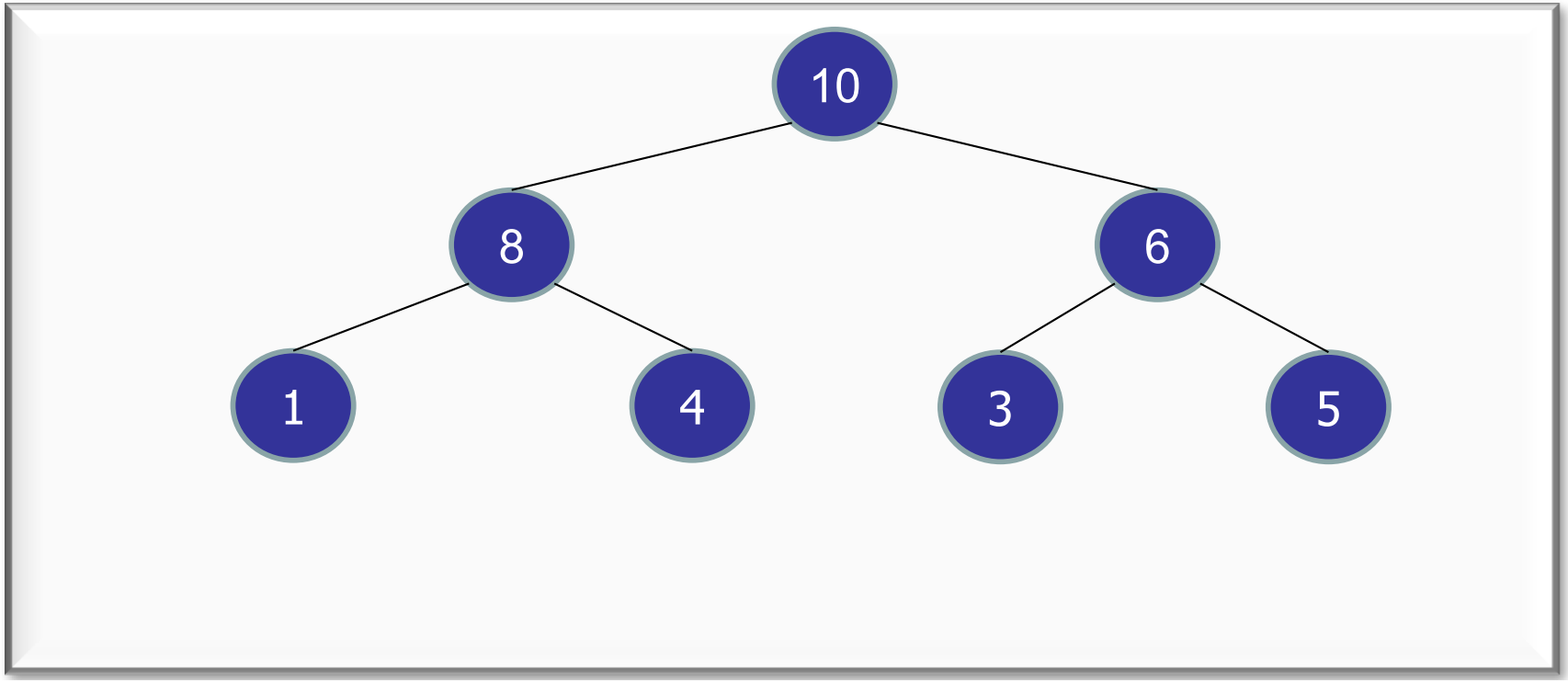
array slot	0	1	2	3	4	5	6	7	8
priority	17	10	6	8	4	3	5	1	24



HeapSort

```
value = extractMax();
A[7] = value;
```

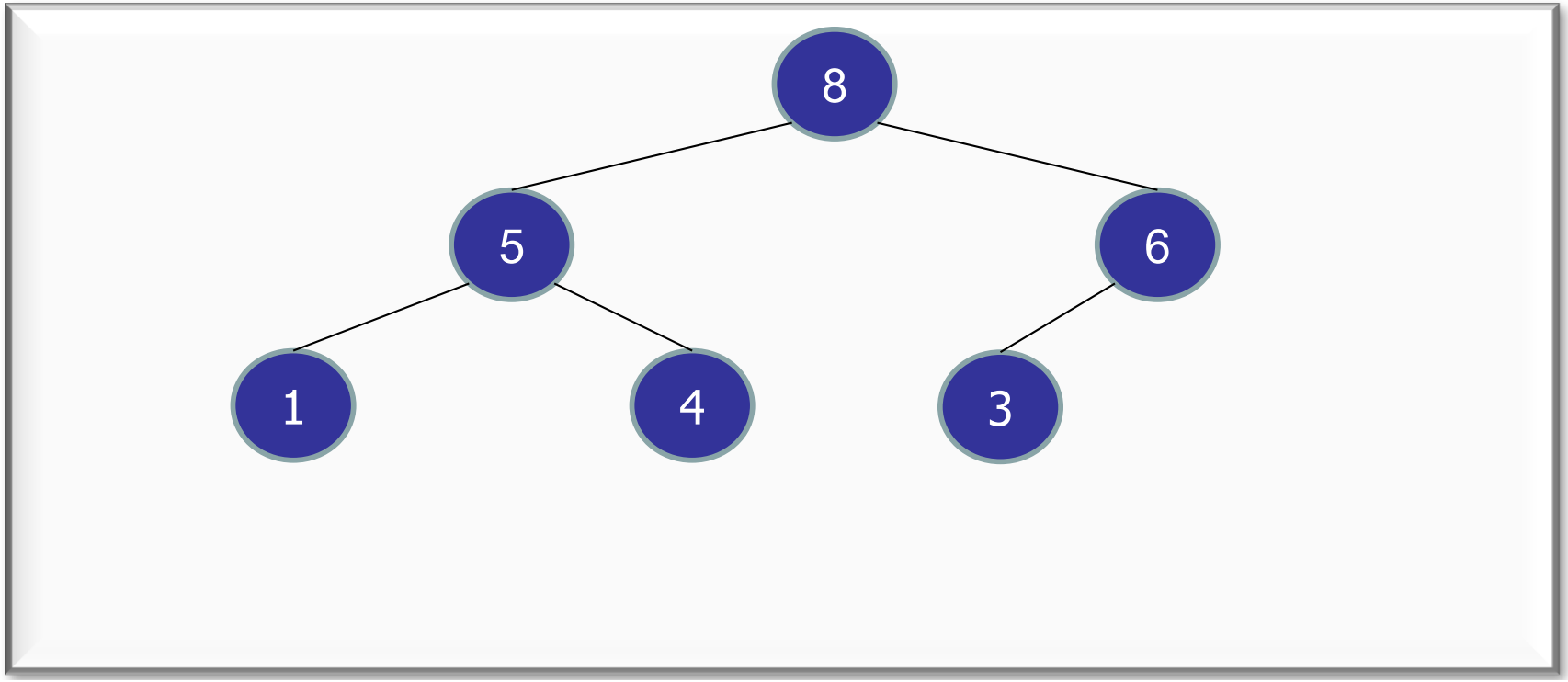
array slot	0	1	2	3	4	5	6	7	8
priority	10	8	6	1	4	3	5	17	24



HeapSort

```
value = extractMax();
A[6] = value;
```

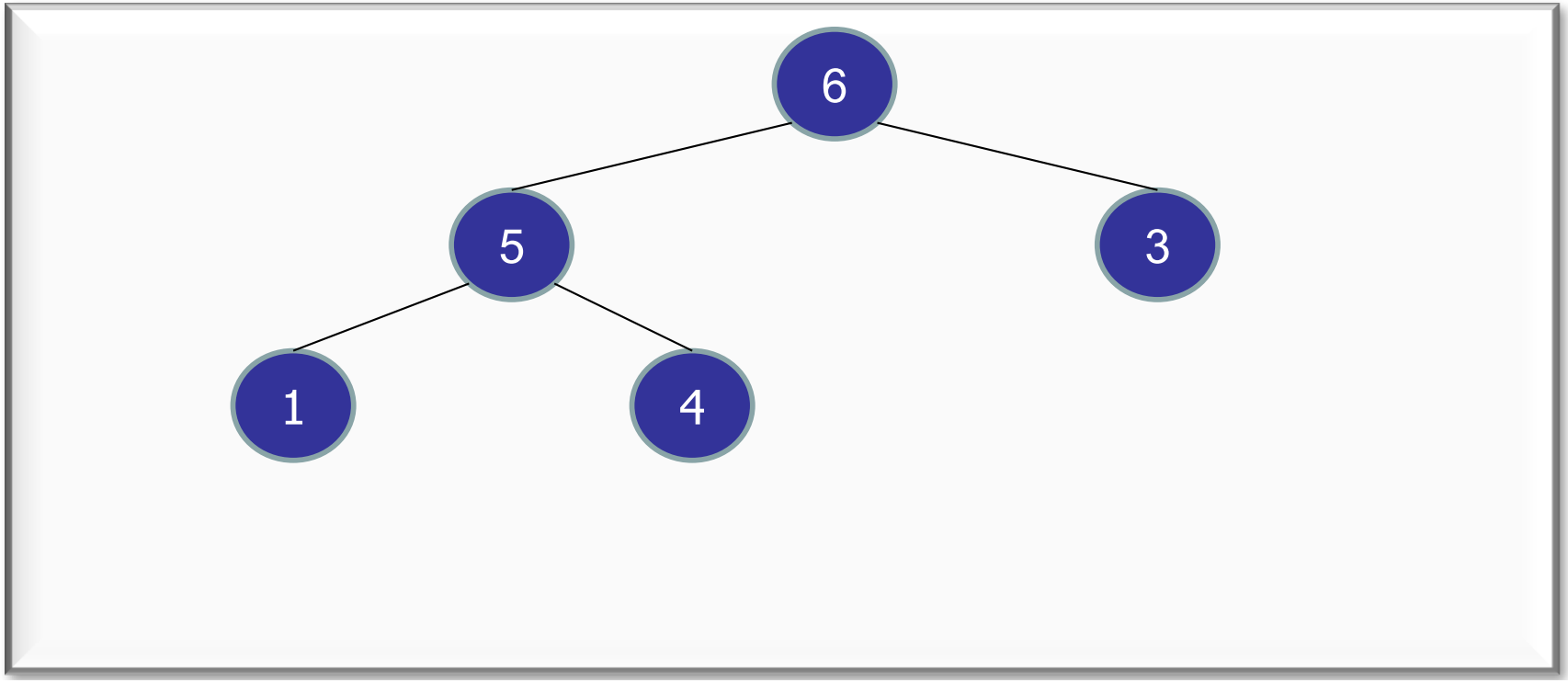
array slot	0	1	2	3	4	5	6	7	8
priority	8	5	6	1	4	3	10	17	24



HeapSort

```
value = extractMax();
A[5] = value;
```

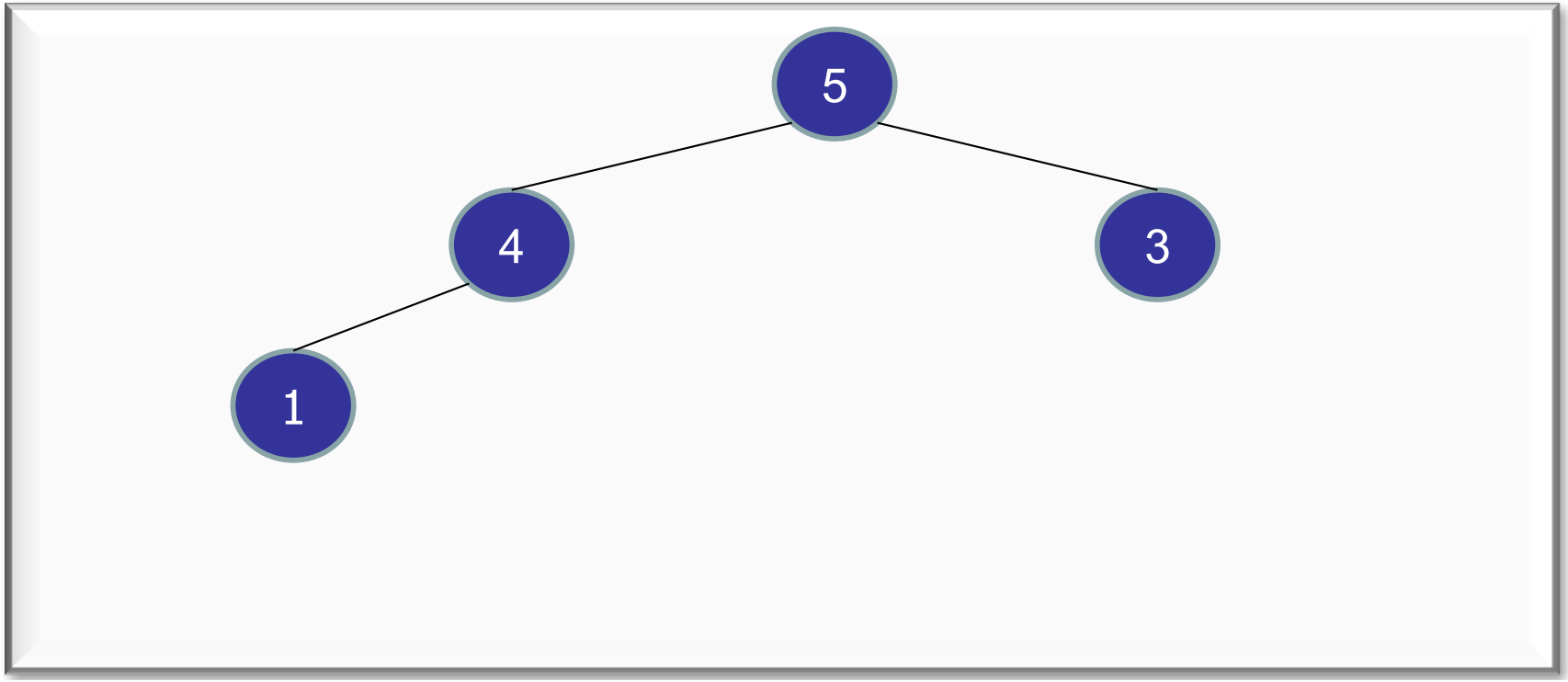
array slot	0	1	2	3	4	5	6	7	8
priority	6	5	3	1	4	8	10	17	24



HeapSort

```
value = extractMax();
A[4] = value;
```

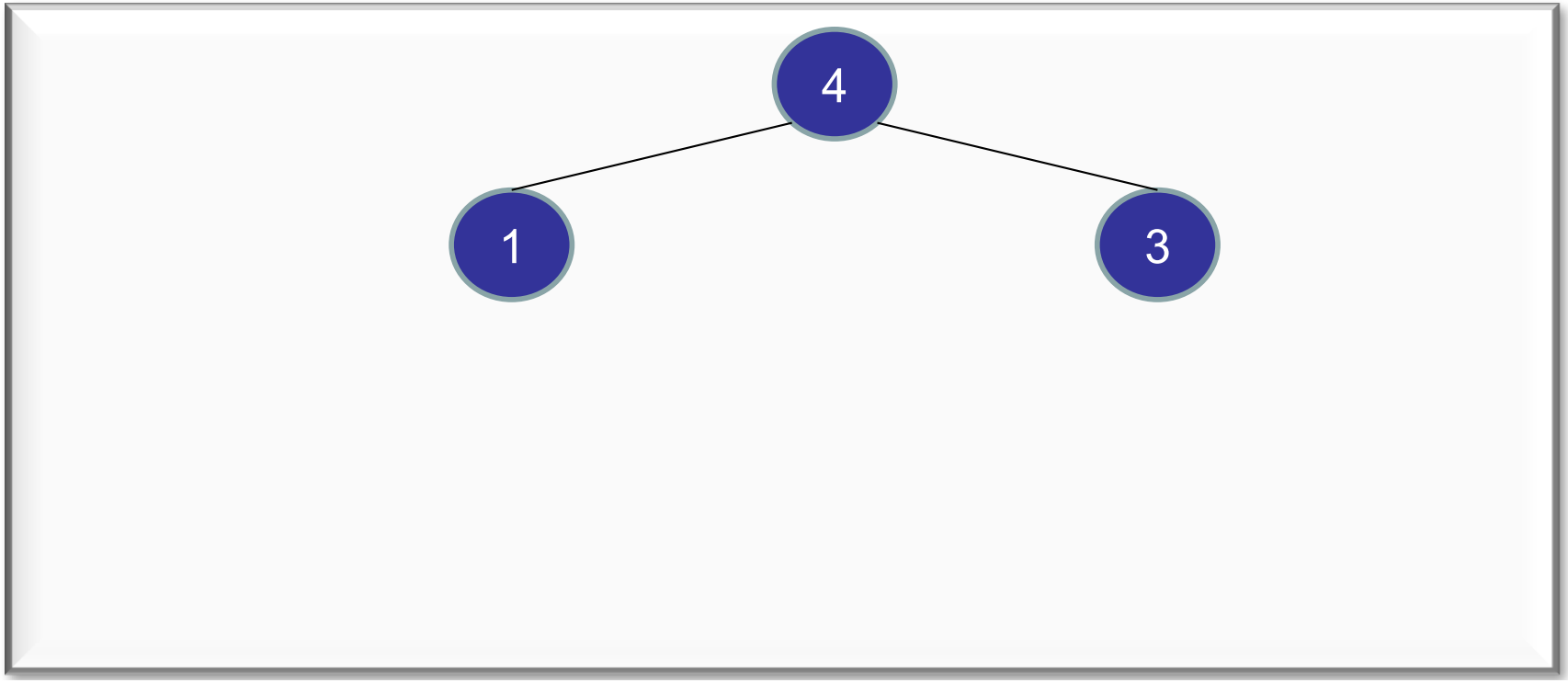
array slot	0	1	2	3	4	5	6	7	8
priority	5	4	3	1	6	8	10	17	24



HeapSort

```
value = extractMax();
A[3] = value;
```

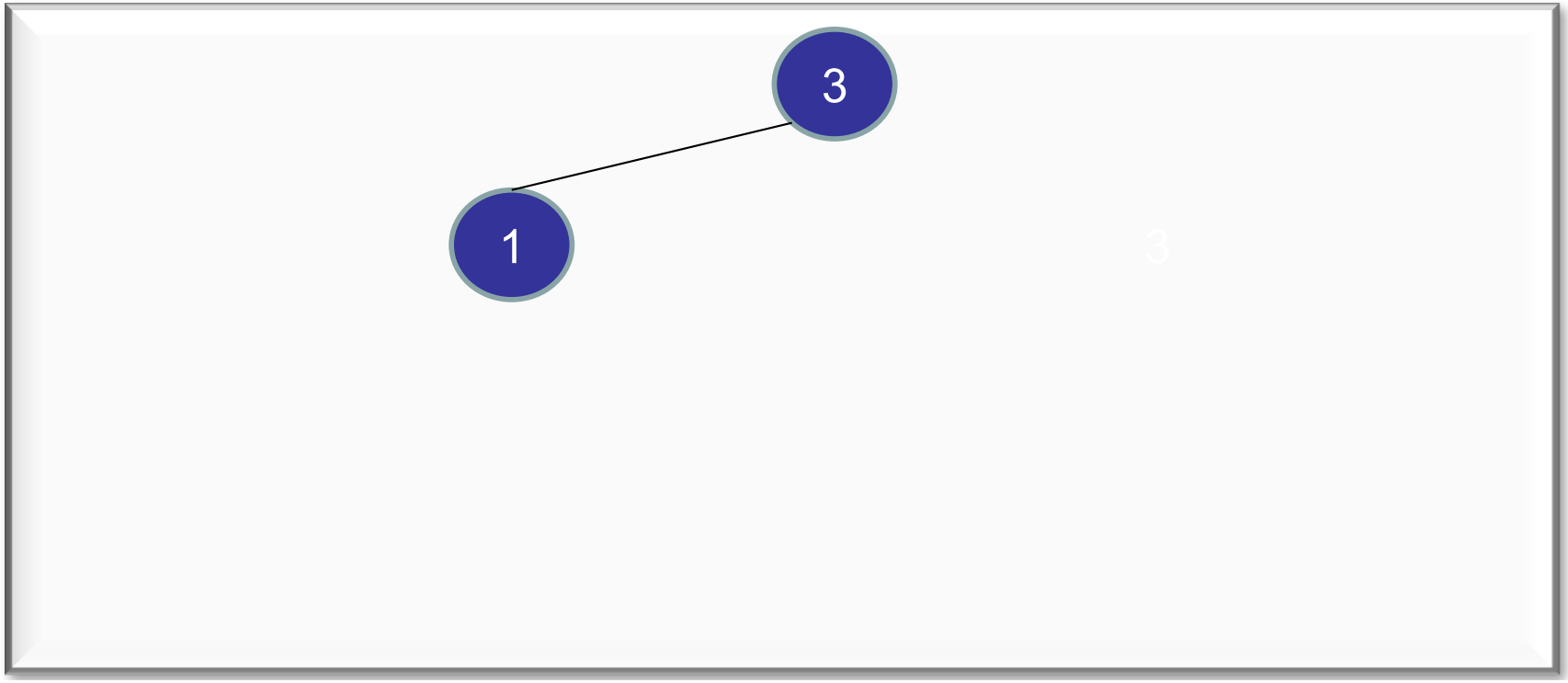
array slot	0	1	2	3	4	5	6	7	8
priority	4	1	3	5	6	8	10	17	24



HeapSort

```
value = extractMax();
A[2] = value;
```

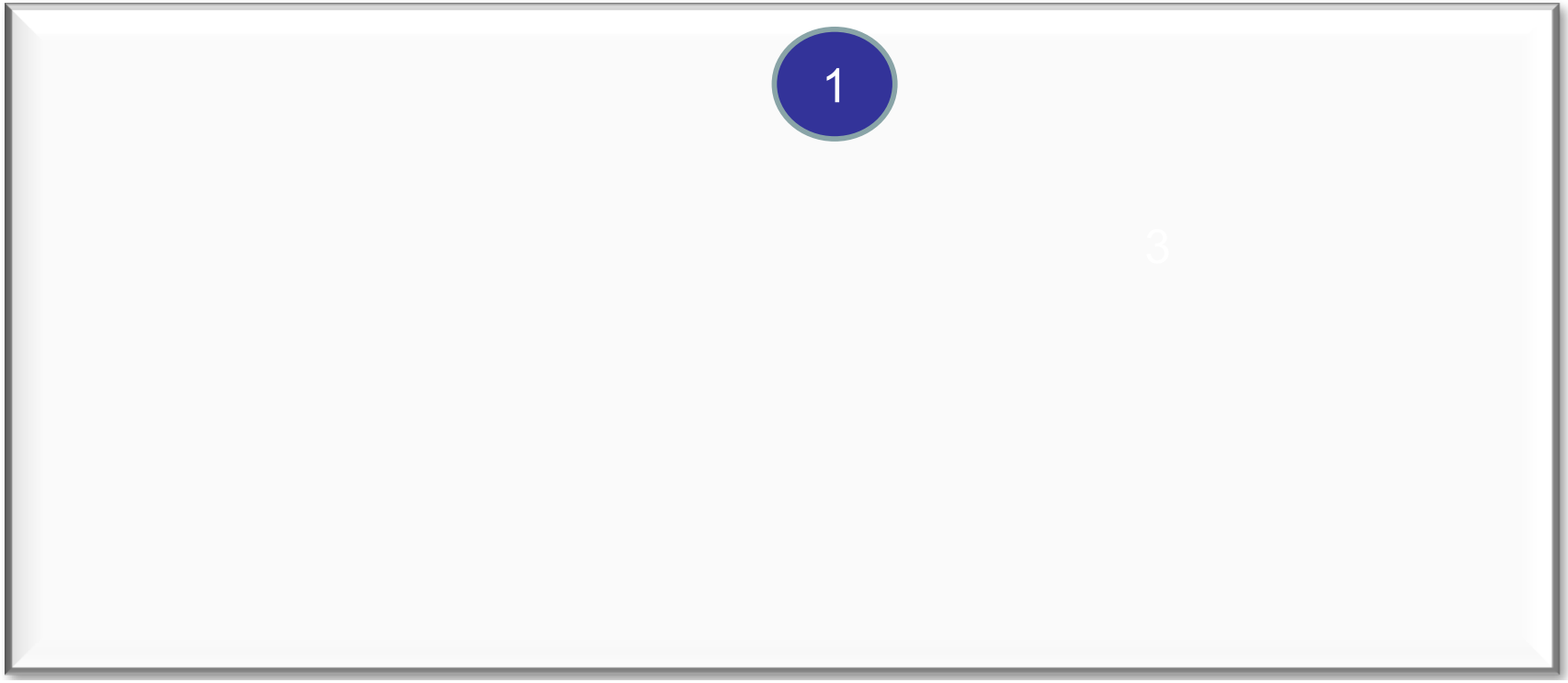
array slot	0	1	2	3	4	5	6	7	8
priority	3	1	4	5	6	8	10	17	24



HeapSort

```
value = extractMax();
A[1] = value;
```

array slot	0	1	2	3	4	5	6	7	8
priority	1	3	4	5	6	8	10	17	24



HeapSort

```
value = extractMax();
A[0] = value;
```

array slot	0	1	2	3	4	5	6	7	8
priority	1	3	4	5	6	8	10	17	24

HeapSort

Heap array ➔ Sorted list:

array slot	0	1	2	3	4	5	6	7	8
priority	1	3	4	5	6	8	10	17	24

```
// int[] A = array stored as a heap
for (int i=(n-1); i>=0; i--) {
      int value = extractMax(A);
      A[i] = value;
}
```

What is the running time for converting a heap into a sorted array?

1. O(log n)
2. O(n)
3. ✓ O(n log n)
4. $O(n^2)$
5. I have no idea.

HeapSort

Heap array ➔ Sorted list: O(n log n)

array slot	0	1	2	3	4	5	6	7	8
priority	1	3	4	5	6	8	10	17	24

```
// int[] A = array stored as a heap
for (int i=(n-1); i>=0; i--) {
      int value = extractMax(A); // O(log n)
      A[i] = value;
}
```

HeapSort

Unsorted list:

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

Unsorted list ➔ Heap

array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	3

HeapSort

Heapify: Unsorted list ➔ Heap:

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

```
// int[] A = array of unsorted integers
for (int i=0; i<n; i++) {
        int value = A[i];
        A[i] = EMPTY:
        heapInsert(value, A, 0, i);
}
```

HeapSort

Heapify: Unsorted list ➔ Heap: O(n log n)

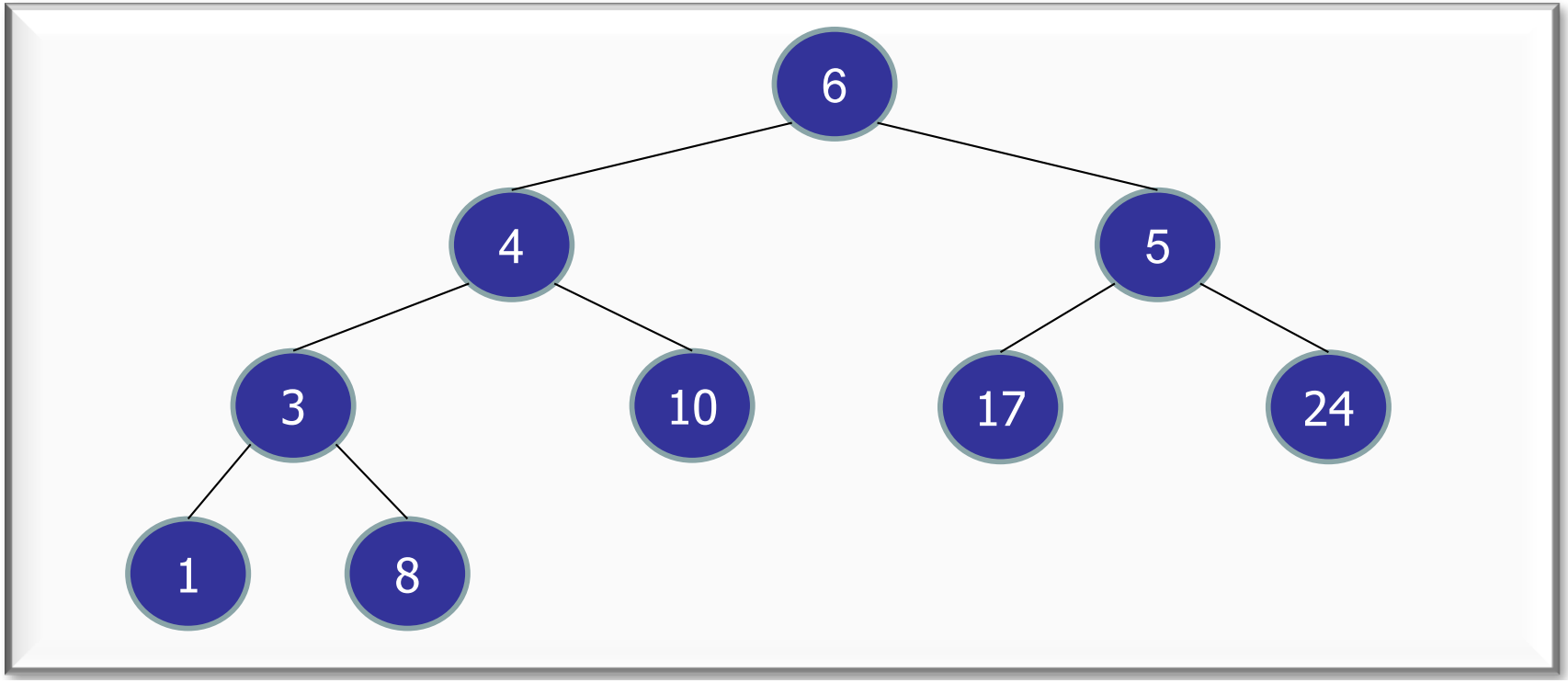
array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

```
// int[] A = array of unsorted integers
for (int i=0; i<n; i++) {
        int value = A[i];
        A[i] = EMPTY:
        heapInsert(value, A, 0, i); // O(log n)
}
```

HeapSort

Heapify v.2: Unsorted list ➔ Heap

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

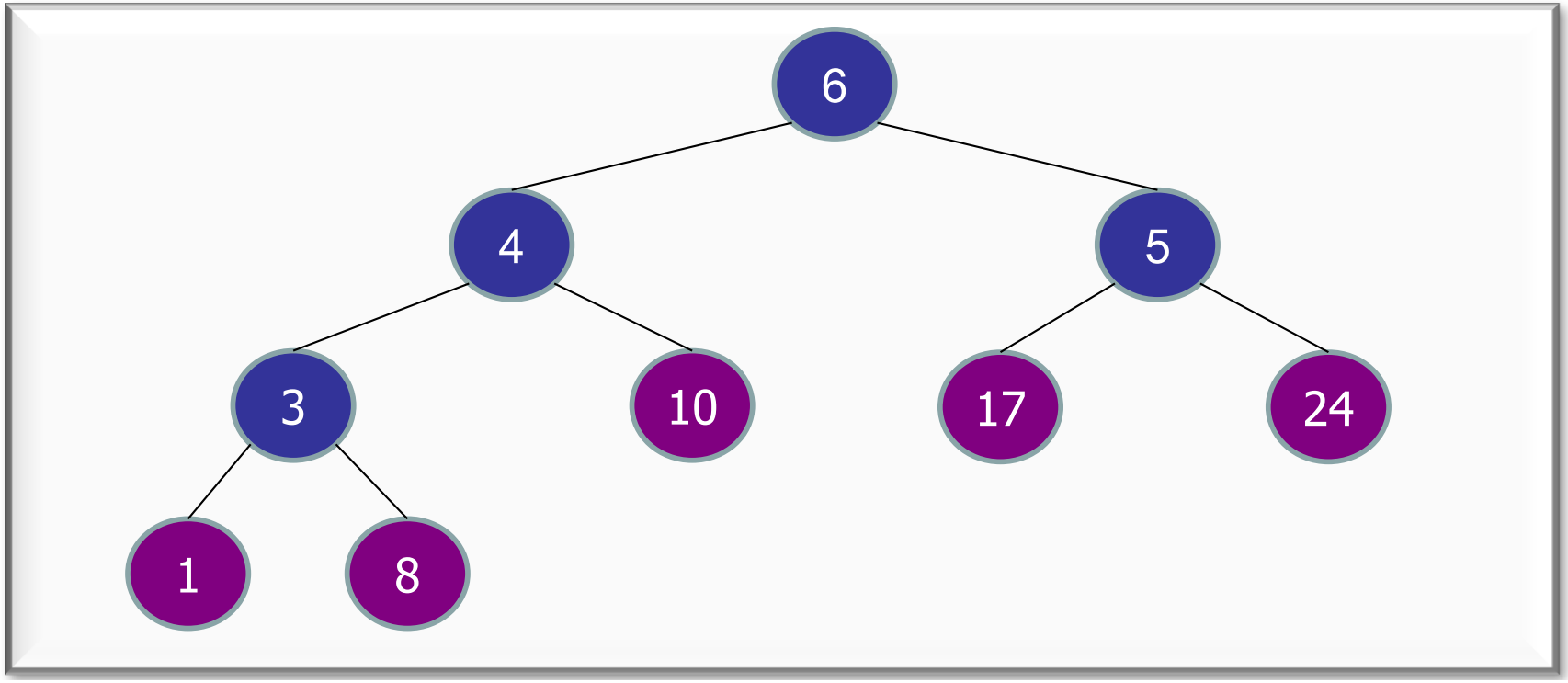


HeapSort

Idea: Recursion

Base case: each leaf is a heap.

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

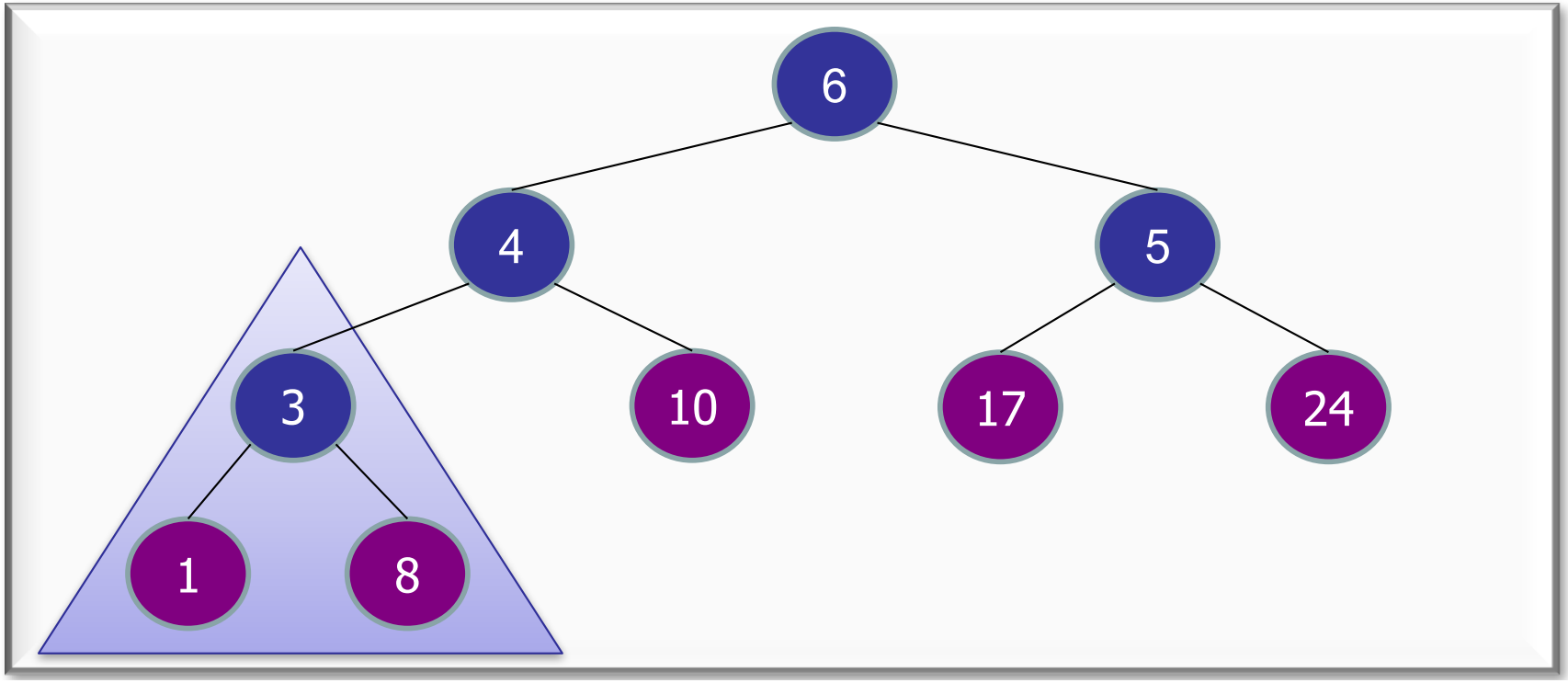


HeapSort

Idea: Recursion

Recursion: left + right are heaps.

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

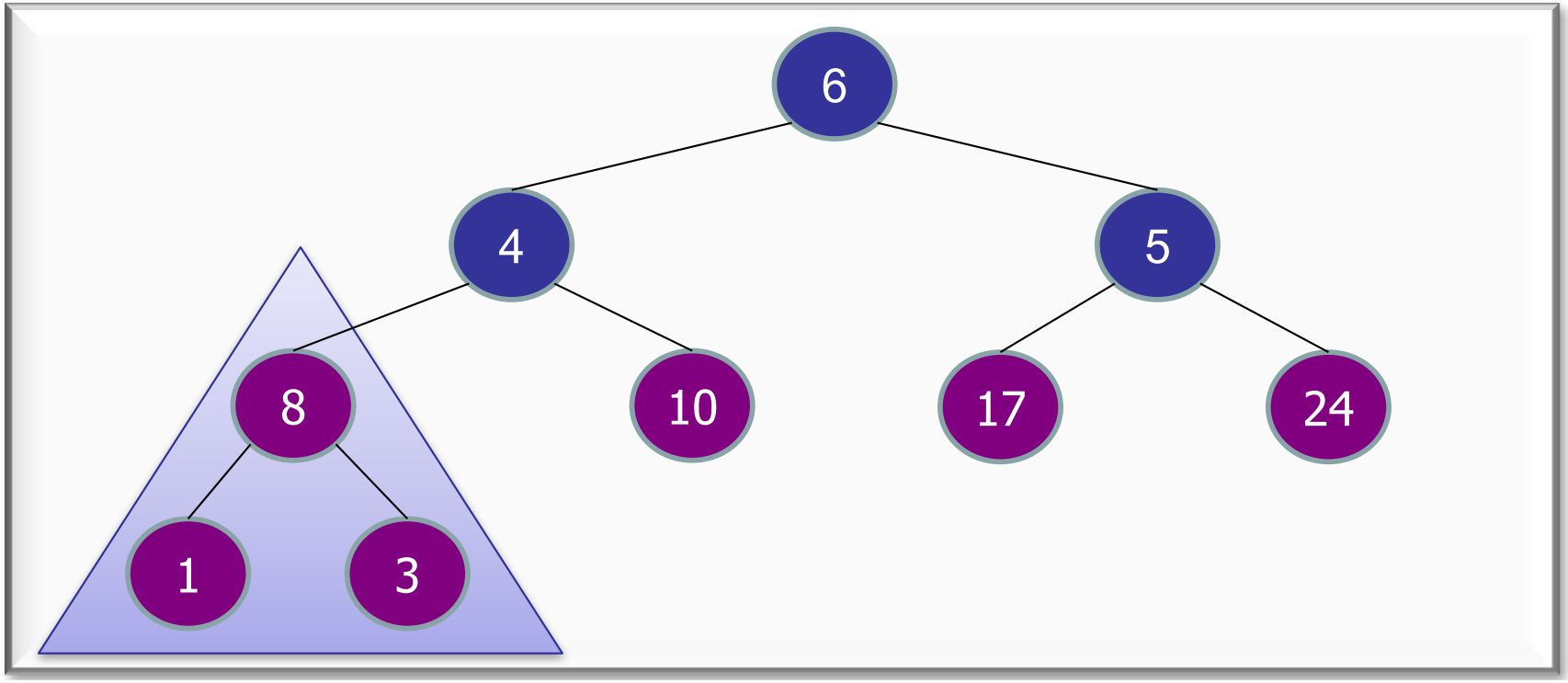


HeapSort

Idea: Recursion

Recursion: left + right are heaps.

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	8	10	17	24	1	3

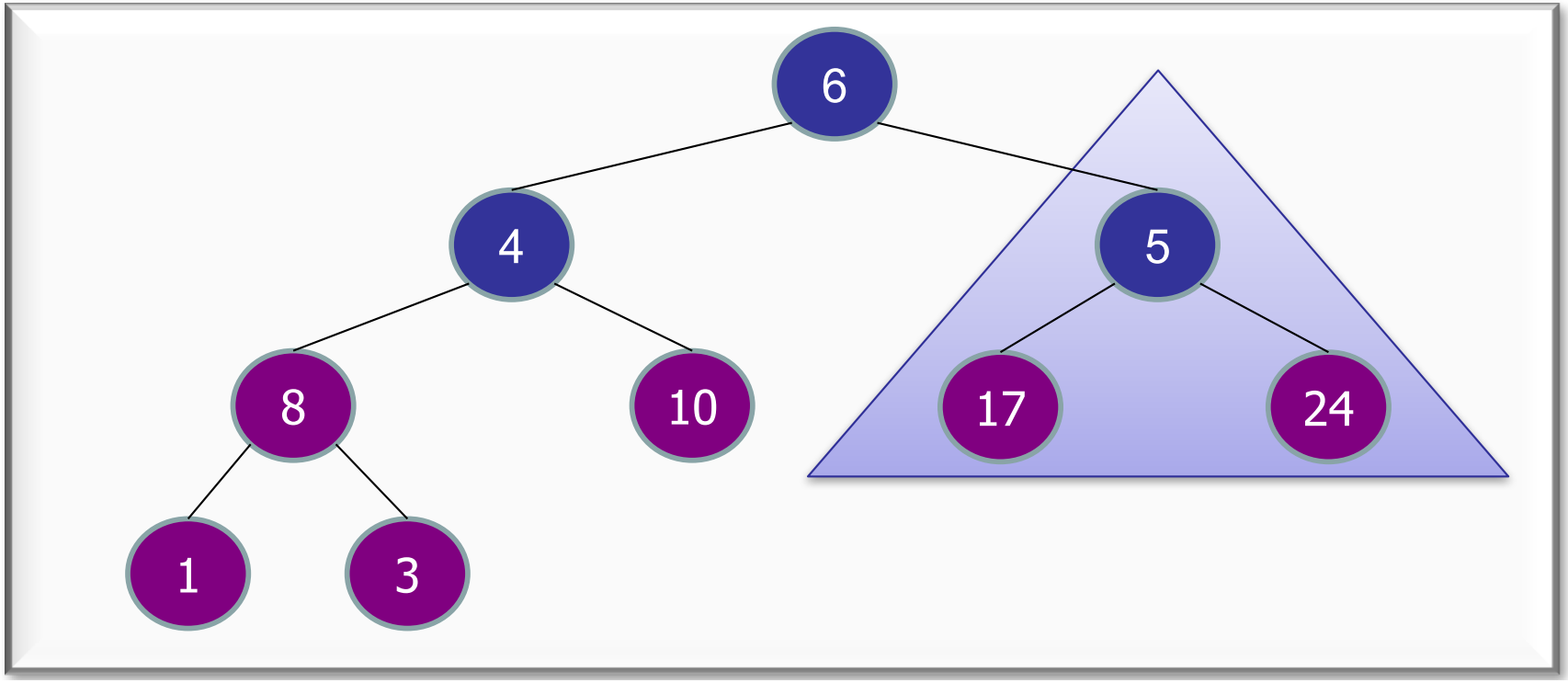


HeapSort

Idea: Recursion

Recursion: left + right are heaps.

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	8	10	17	24	1	3

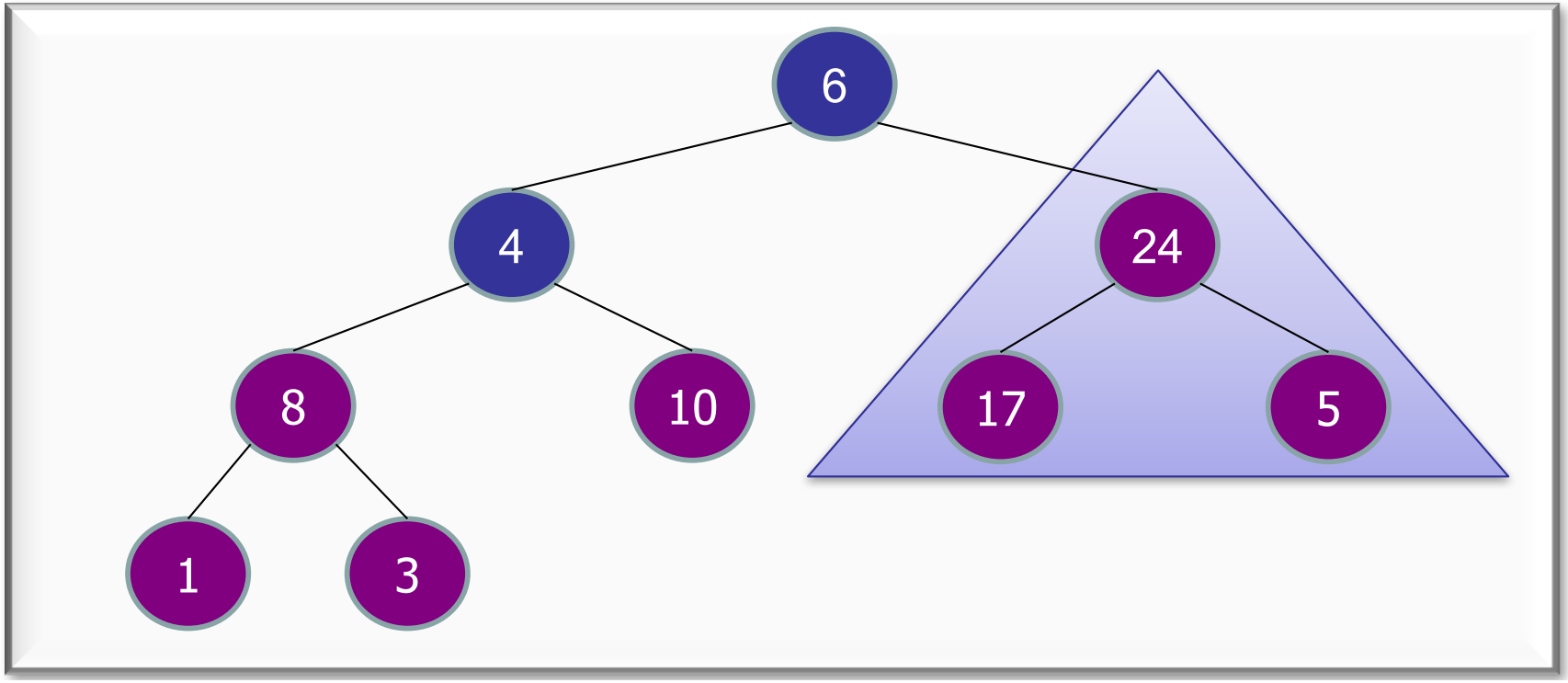


HeapSort

Idea: Recursion

Recursion: left + right are heaps.

array slot	0	1	2	3	4	5	6	7	8
key	6	4	24	8	10	17	5	1	3

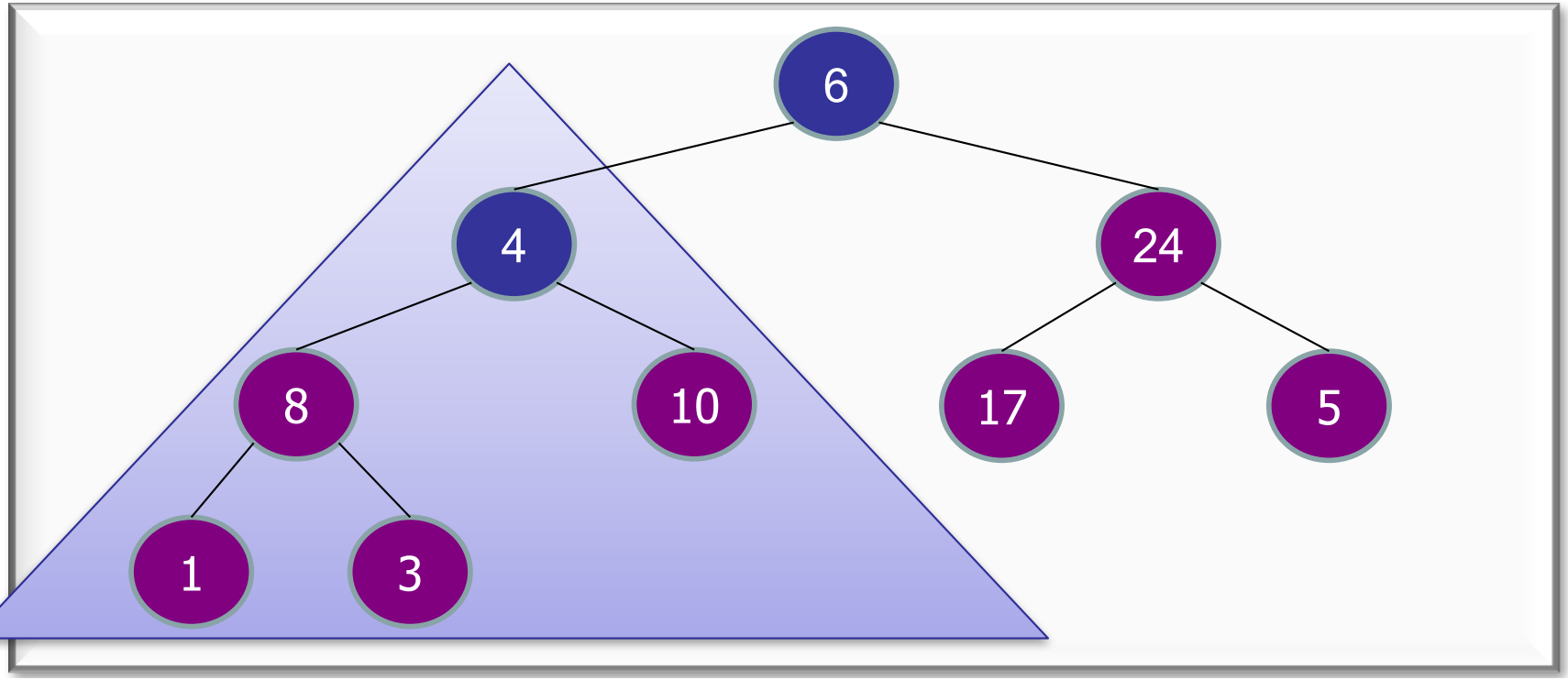


HeapSort

Idea: Recursion

Recursion: left + right are heaps.

array slot	0	1	2	3	4	5	6	7	8
key	6	4	24	8	10	17	5	1	3

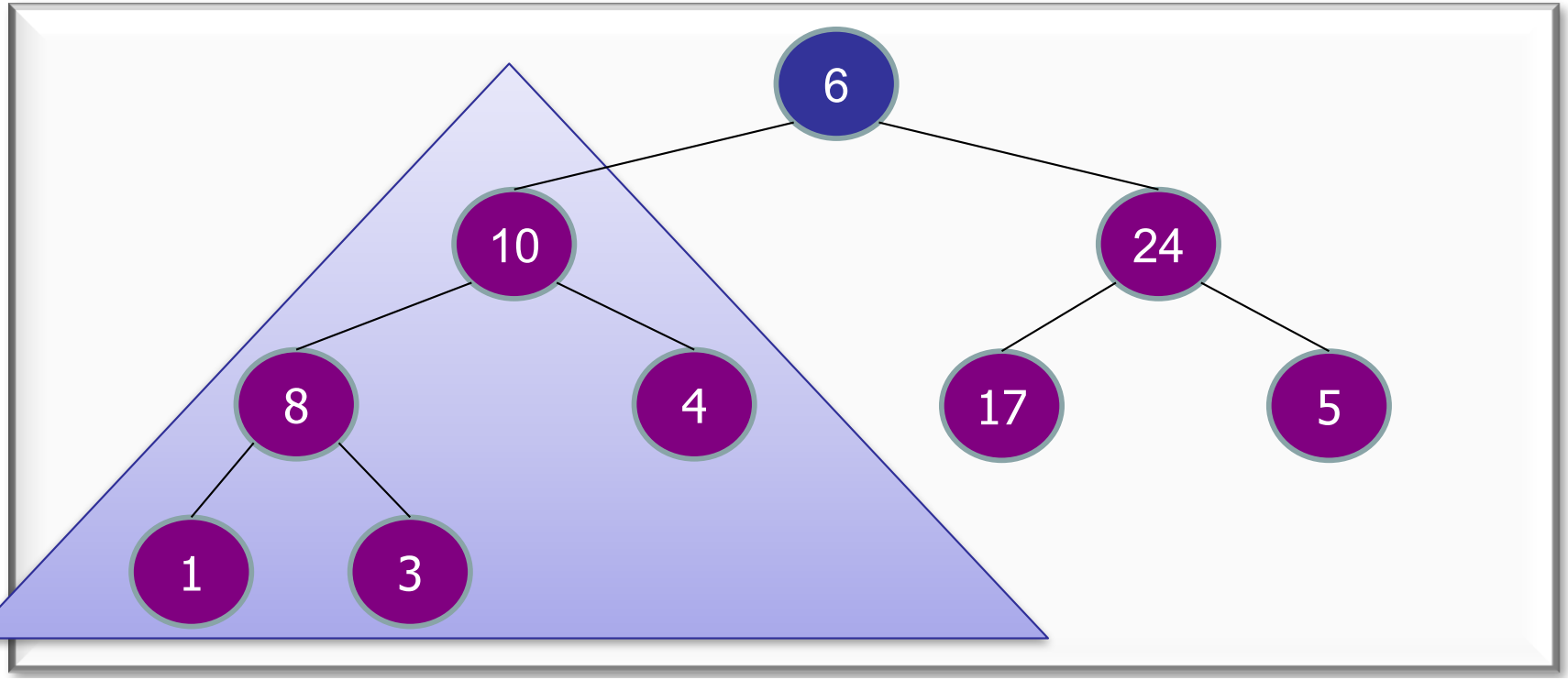


HeapSort

Idea: Recursion

Recursion: left + right are heaps.

array slot	0	1	2	3	4	5	6	7	8
key	6	10	24	8	4	17	5	1	3

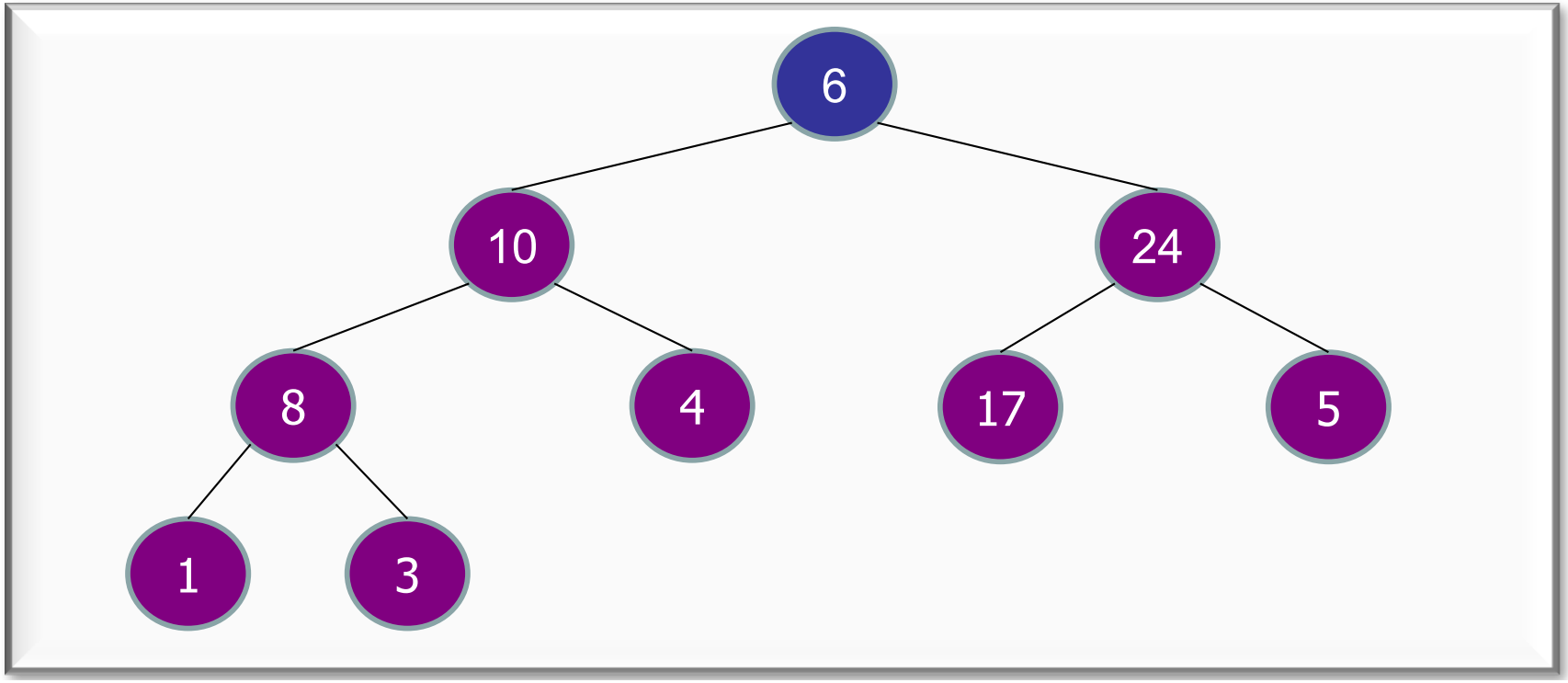


HeapSort

Idea: Recursion

Recursion: left + right are heaps.

array slot	0	1	2	3	4	5	6	7	8
key	6	10	24	8	4	17	5	1	3

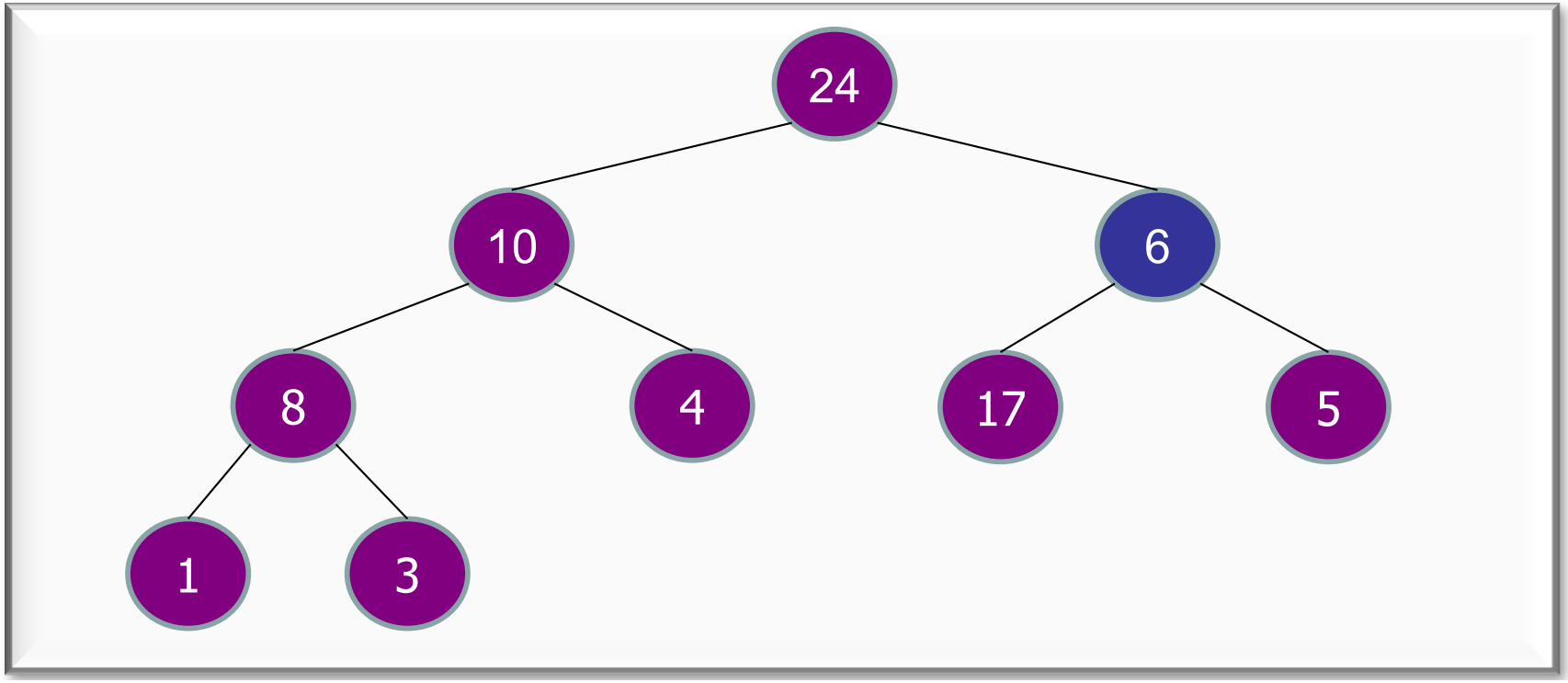


HeapSort

Idea: Recursion

Recursion: left + right are heaps.

array slot	0	1	2	3	4	5	6	7	8
key	24	10	6	8	4	17	5	1	3

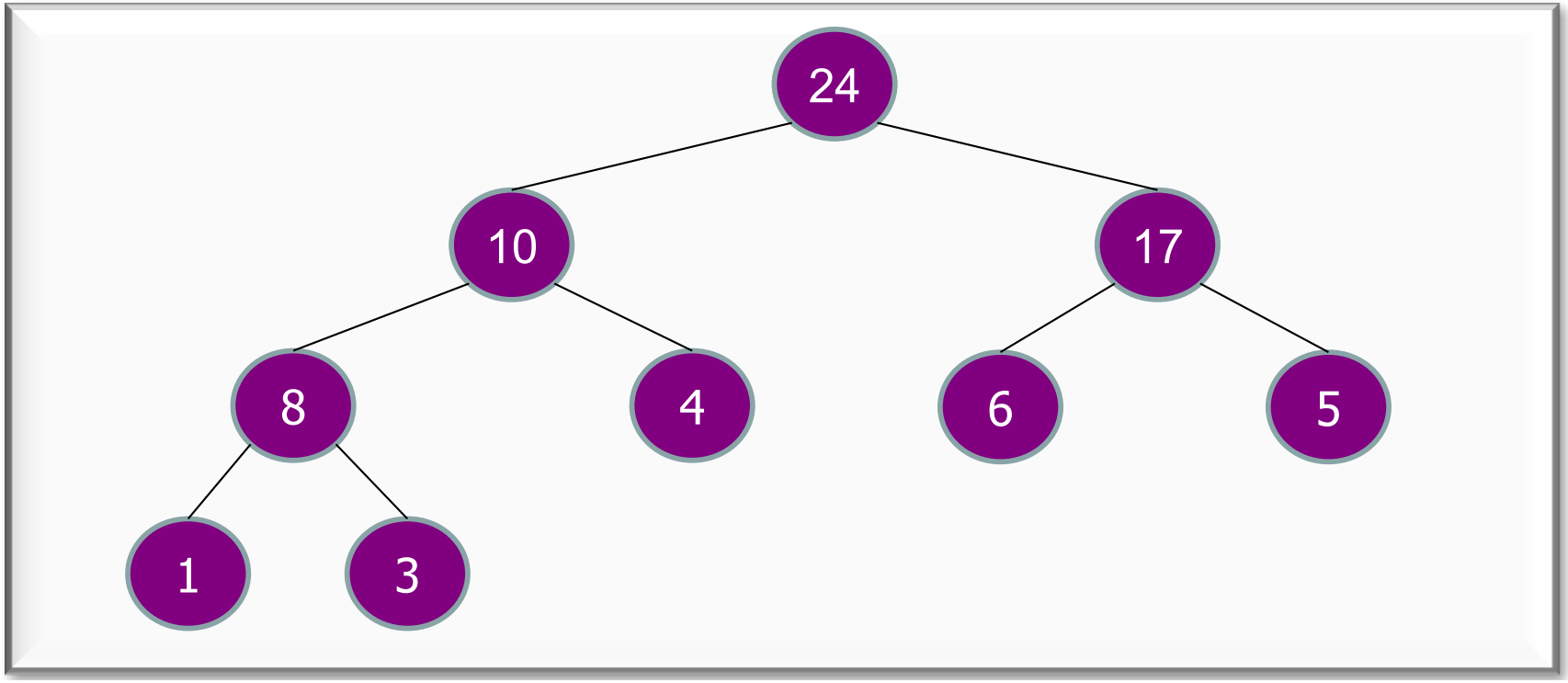


HeapSort

Idea: Recursion

Recursion: left + right are heaps.

array slot	0	1	2	3	4	5	6	7	8
key	24	10	17	8	4	6	5	1	3



HeapSort

Heapify v.2: Unsorted list ➔ Heap

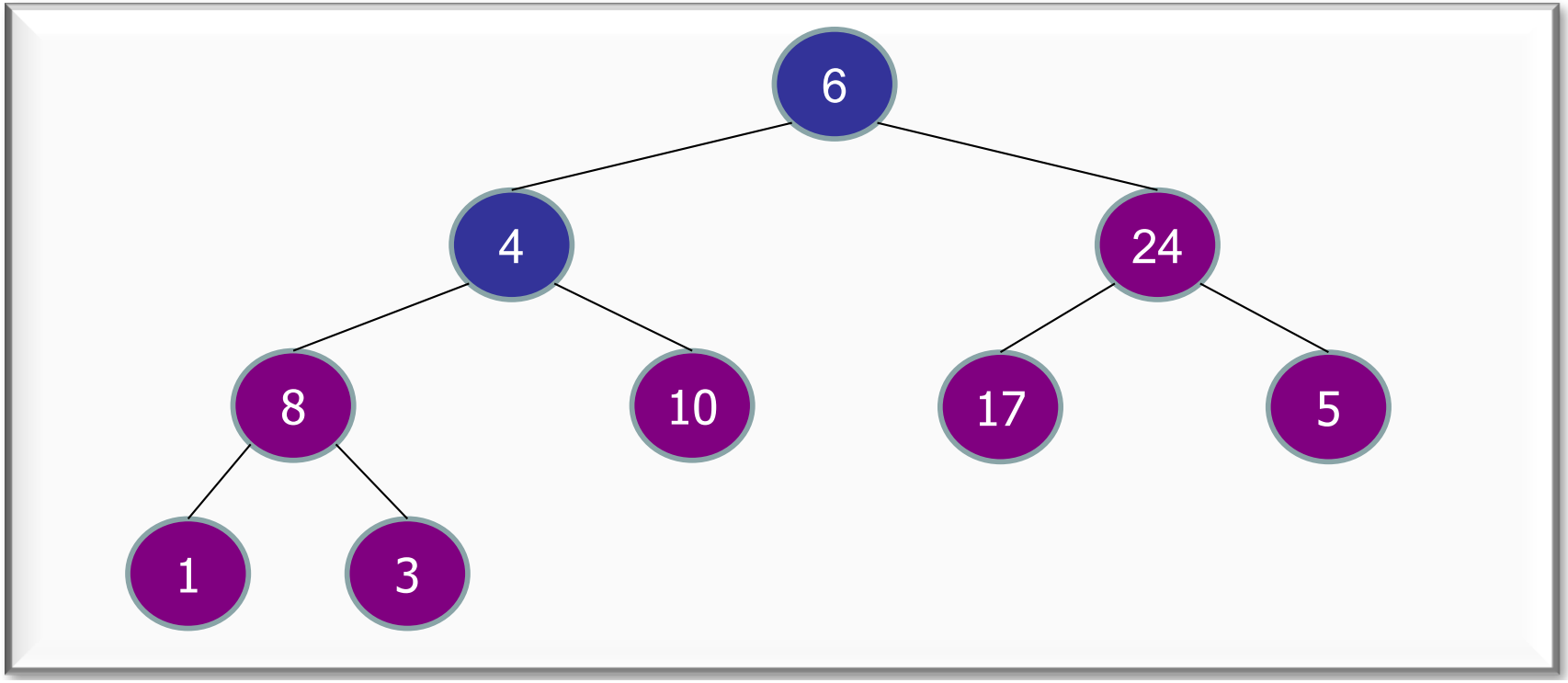
array slot	0	1	2	3	4	5	6	7	8
key	24	10	17	8	4	6	5	1	3

```
// int[] A = array of unsorted integers
for (int i=(n-1); i>=0; i--) {
      bubbleDown(i, A); // O(log n)
}
```

HeapSort

Observation: cost(bubbleDown) = height

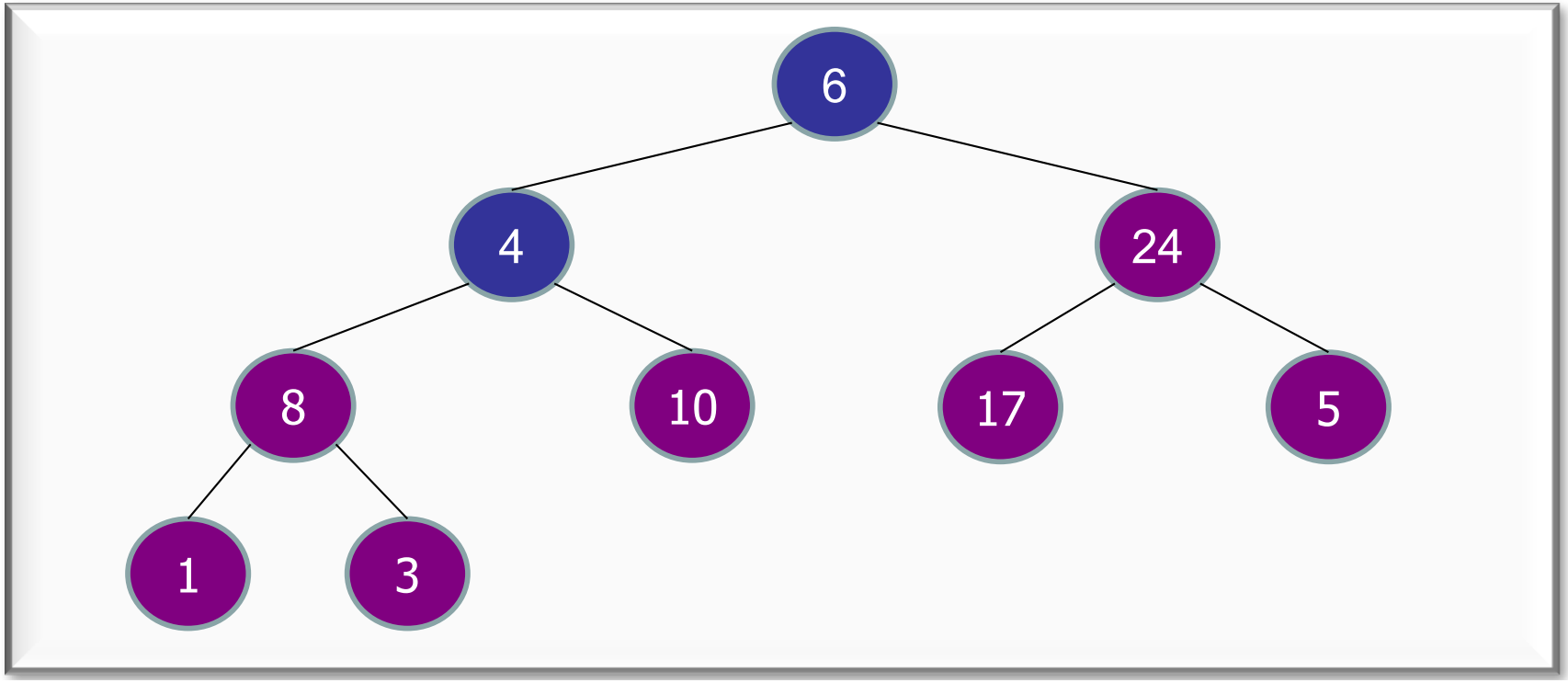
array slot	0	1	2	3	4	5	6	7	8
key	6	4	24	8	10	17	5	1	3



HeapSort

Observation: > n/2 nodes are leaves (height=0)

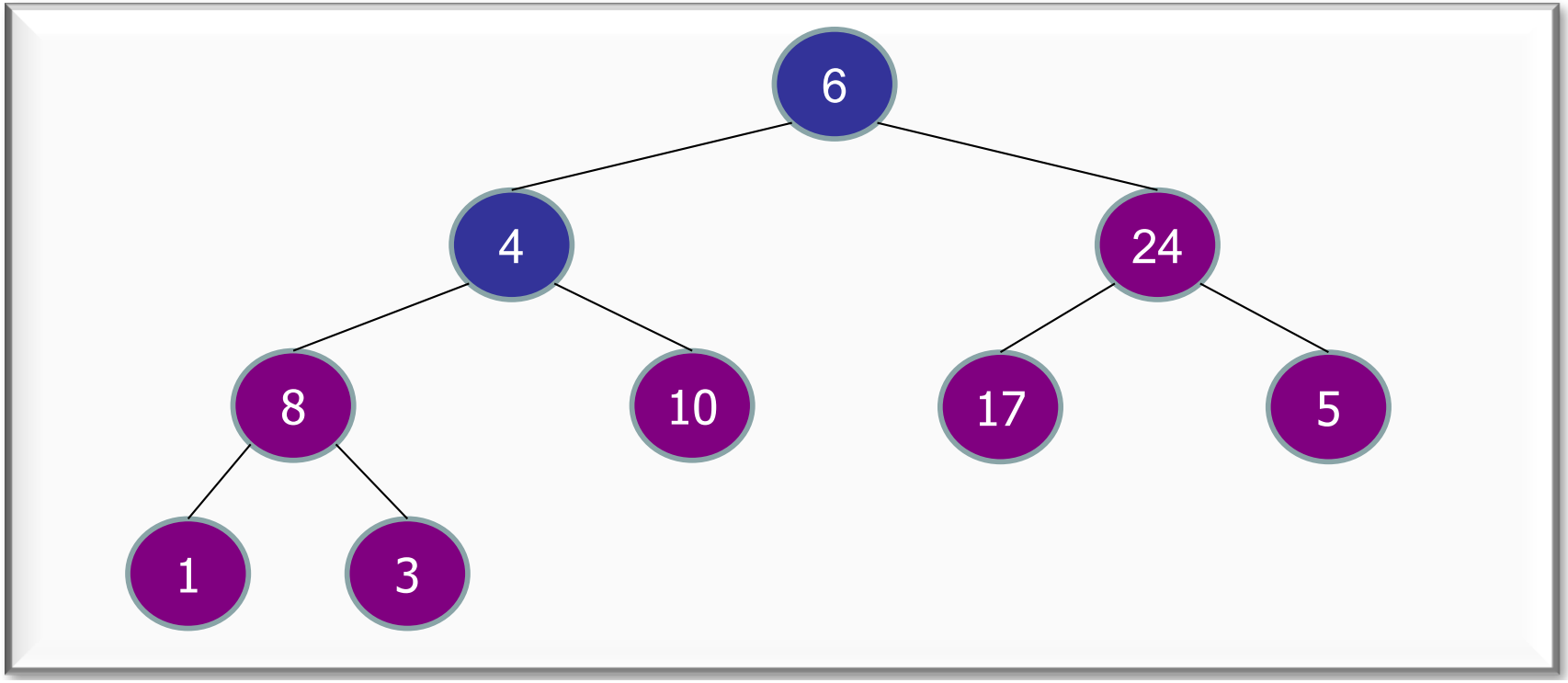
array slot	0	1	2	3	4	5	6	7	8
key	6	4	24	8	10	17	5	1	3



HeapSort

Observation: most nodes have small height!

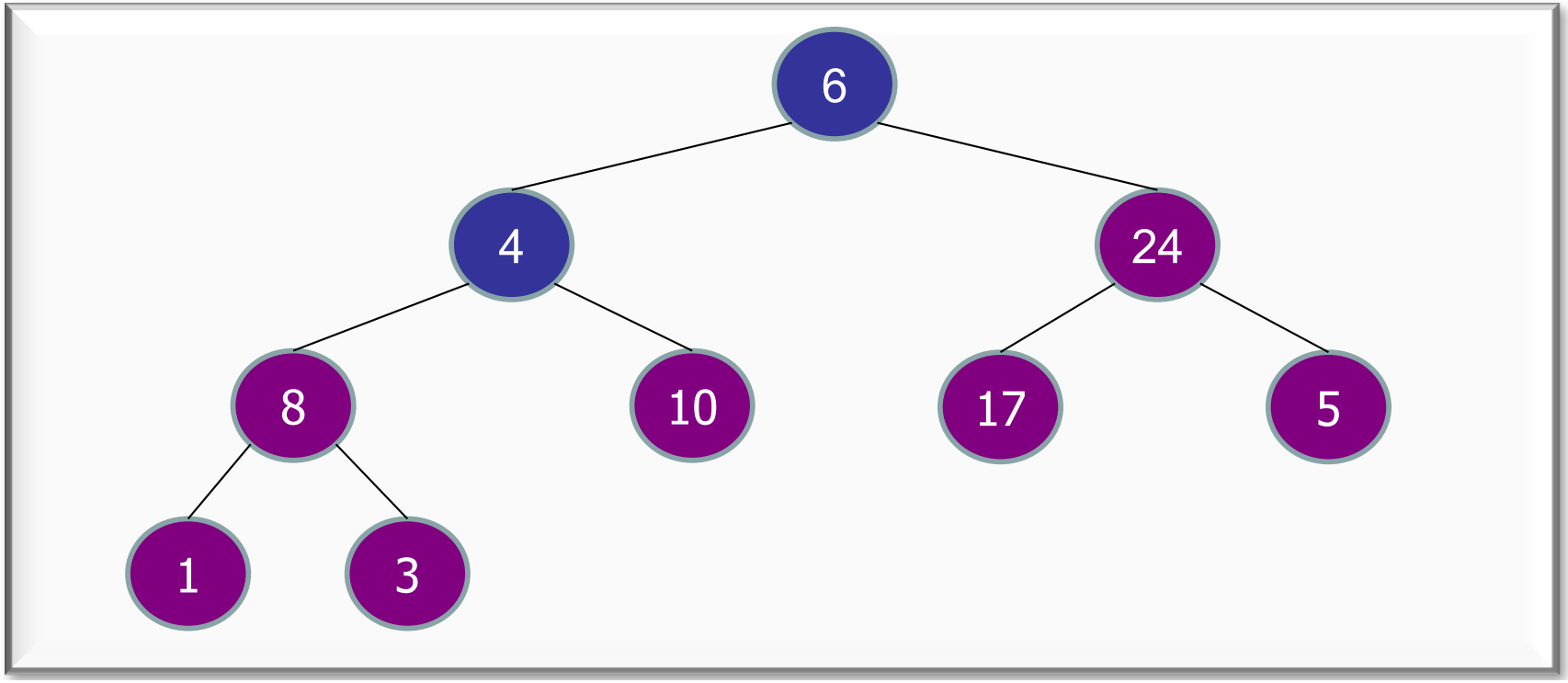
array slot	0	1	2	3	4	5	6	7	8
key	6	4	24	8	10	17	5	1	3



HeapSort

Cost of building a heap:

Height	0	1	2	3	...	⌊ log(n) ⌋
Number	⌈n/2⌉	⌈n/4⌉	⌈n/8⌉	⌈n/16⌉	...	1



HeapSort

Cost of building a heap:

Height	0	1	2	3	...	⌊ log(n) ⌋
Number	⌈n/2⌉	⌈n/4⌉	⌈n/8⌉	⌈n/16⌉	...	1

$$\sum_{h=0}^{h=\log(n)} \frac{n}{2^h} O(h)$$

cost for bubbling a node at level h

upper bound on number of nodes at level h

HeapSort

Cost of building a heap:

Height	0	1	2	3	...	⌊ log(n) ⌋
Number	⌈n/2⌉	⌈n/4⌉	⌈n/8⌉	⌈n/16⌉	...	1

$$\sum_{h=0}^{h=\log(n)} \frac{n}{2^h} O(h) \le cn\left(\frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \ldots\right)$$

$$\le cn\left(\frac{1/2}{(1-1/2)^2}\right) \le 2 \cdot O(n)$$

HeapSort

Heapify v.2: Unsorted list ➔ Heap: O(n)

array slot	0	1	2	3	4	5	6	7	8
key	24	10	17	8	4	6	5	1	3

```
// int[] A = array of unsorted integers
for (int i=(n-1); i>=0; i--) {
      bubbleDown(i, A); // O(height)
}
```

HeapSort

Unsorted list:

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

Unsorted list ➔ Heap: O(n)

array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	3

Heap array ➔ Sorted list: O(n log n)

array slot	0	1	2	3	4	5	6	7	8
key	1	3	4	5	6	8	10	17	24

HeapSort

Summary

- O(n log n) time *worst-case*
- In-place
- Fast:
  - Faster than MergeSort
  - A little slower than QuickSort.
- Deterministic: always completes in O(n log n)
- Unstable (Come up with an example!)
- Ternary (3-way) HeapSort is a little faster.

Where is the largest element in a max-heap?

1. Leftmost child
2. ✓ Root
3. Rightmost child
4. It depends
5. I forget.

Where is the smallest element in a max-heap?

1. Leftmost child
2. Root
3. Rightmost child
4. ✓ It depends
5. I forget.

Where is the cost of finding the successor of an arbitrary element in a heap?

1. $O(1)$
2. $O(\log n)$
3. ✓ $O(n)$
4. $O(n^2)$
5. I forget.

Let A be an array sorted from largest to smallest. Is A a max-heap?

✓1. Yes
2. No
3. Maybe
4. I don’t know.

How fast is HeapSort on a sorted array?

1. O(n)
2. ✓ O(n log n)
3. $O(n^2)$
4. It depends
5. I forget.

Priority Queue

Maintain a set of prioritized objects:

- insert: add a new object with a specified priority
- extractMin: remove and return the object with minimum priority

Ex: Scheduling

- Find next task to do
- Earliest deadline first

Task	Due date
CS4234 PS8	March 31
Study for Exam	April 4
Wash clothes	April 6
See friends	May 12