CS2040C
Data Structures and Algorithms

Welcome!

Roadmap

Part I: Priority Queues

- Binary Heaps
- HeapSort

Priority Queue ADT

Maintain a set of prioritized objects:

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

- Examples:
  - Event-driven simulation
    - customers in a line
  - Scheduling
  - Graph searching
  - Artificial intelligence
    - A* search

similar to queue just that the entire table will auto sort so that dequeue will always be the min/max

Task	Due date
HW	March 31
Study for Quiz 2	April 4
Wash clothes	April 6
See friends	May 12

Abstract Data Type

Min Priority Queue

`void`	`insert(Key k, Priority p)`	*insert k with priority p*
`Data`	`extractMin()` OR extractMax()	*remove key with minimum priority*
`void`	`decreaseKey(Key k, Priority p)` or increase	*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

`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

A

23

push array back

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

Priority Queue

Unsorted array

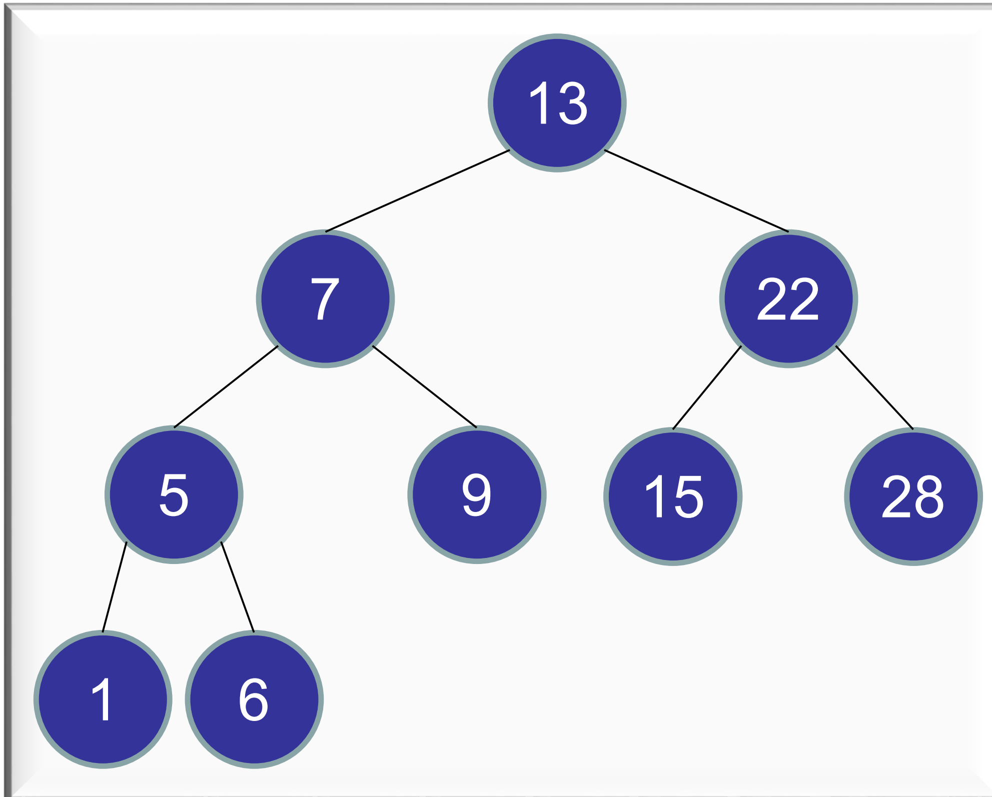
- insert: O(1)
  - Add object to end of list
- extractMax: O(n)
  - Search for largest element in array.
  - Remove and move everything over.

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.

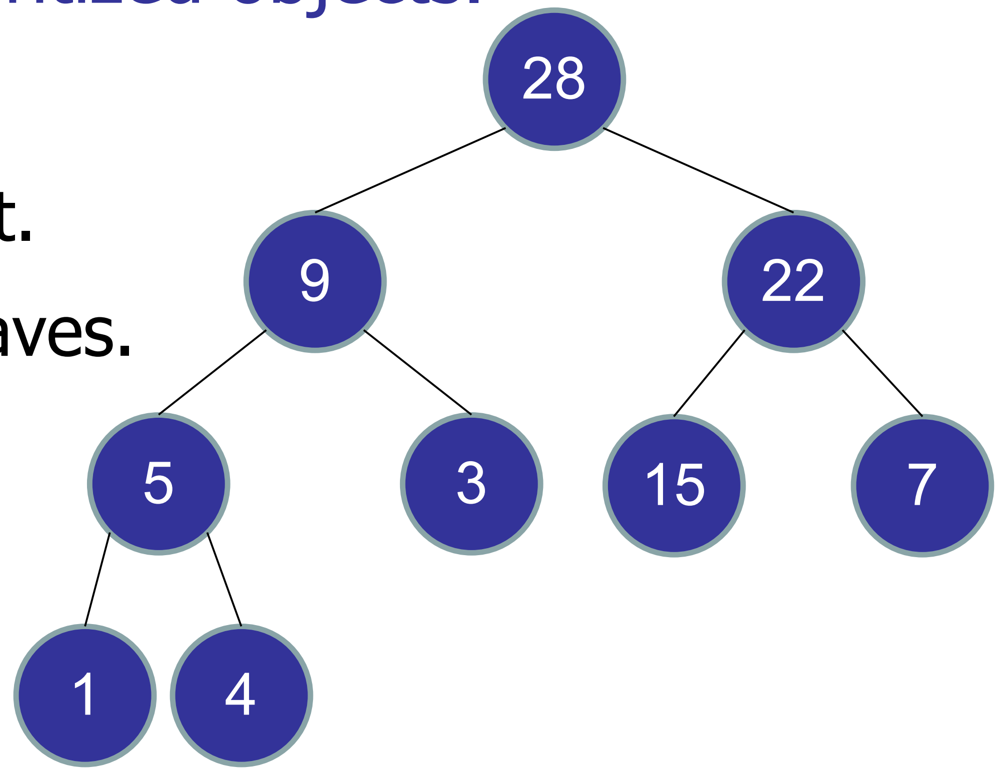
Hash table:

- Maps priorities to array slots or nodes in tree.

Heap

(aka Binary Heap or MaxHeap)

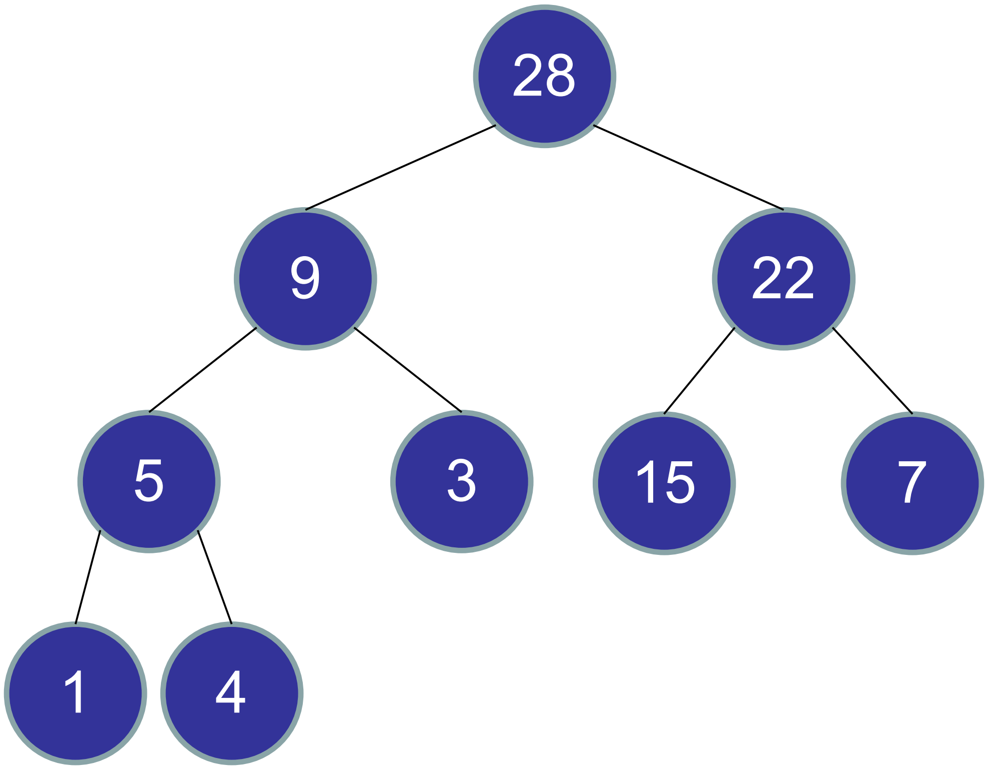
- Implements a Max Priority Queue
- Maintain a set of prioritized objects.
- Store items in a tree.
  - Biggest items at root.
  - Smallest items at leaves.



Two Properties of a Heap

1. Heap Ordering

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

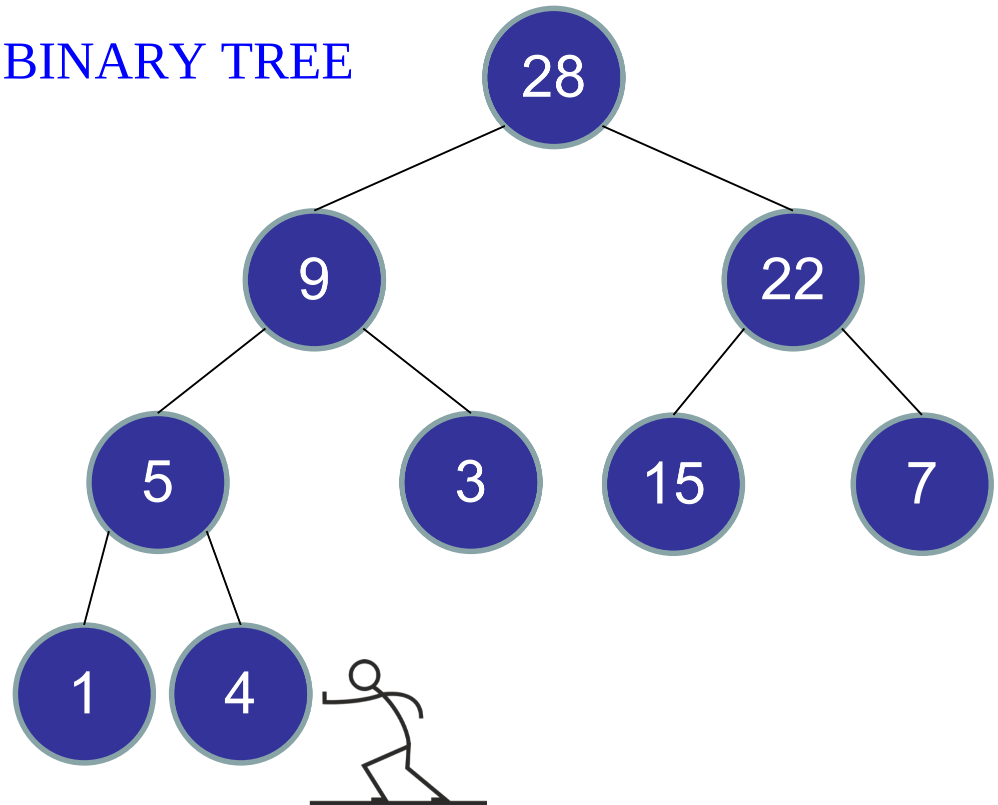


Note: not a binary **search** tree.

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.

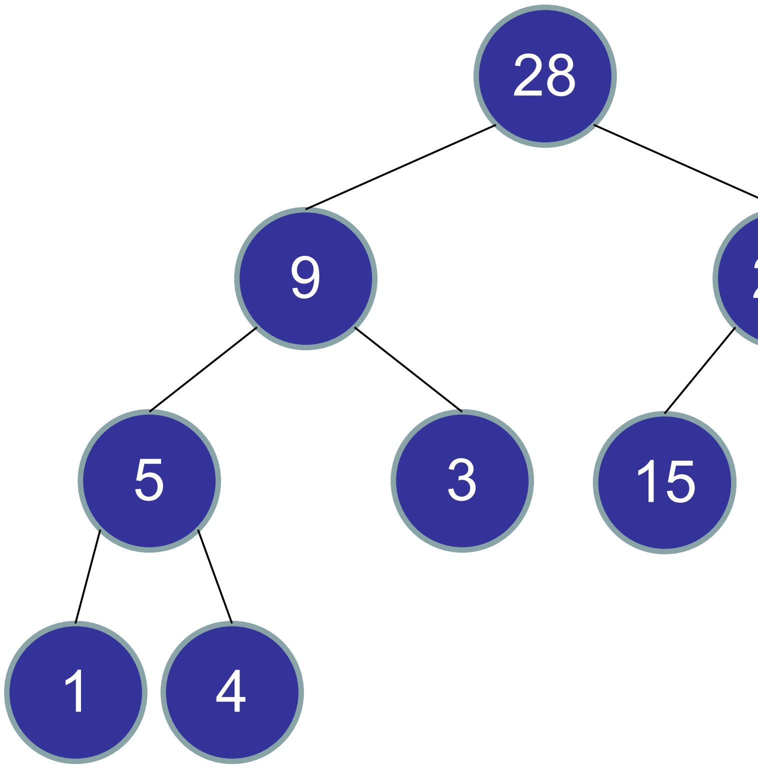
NOT A FULL BINARY TREE



Heap

(aka Binary Heap or MaxHeap)

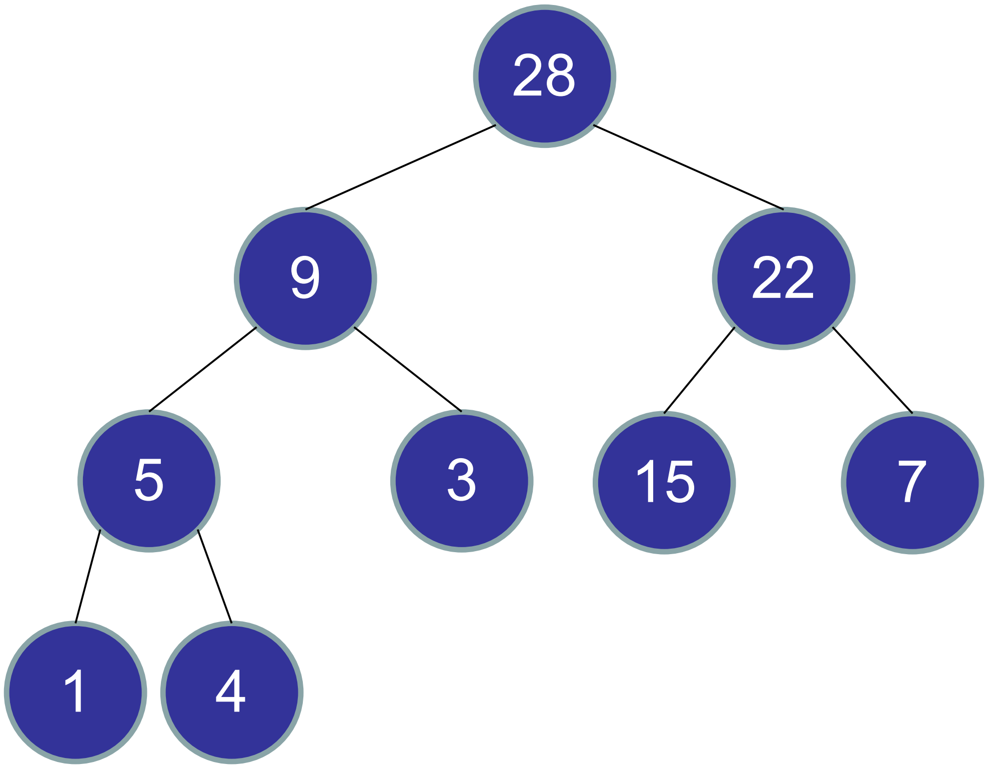
- Implements a Max Priority Queue
- Maintain a set of prioritized objects.
- Store items in a tree.
  - Biggest items at root.
  - Smallest items at leaves.
- Two properties:
  1. Heap Ordering
  2. Complete Binary Tree
- Height: O(log n)



Heap

Priority Queue Operations

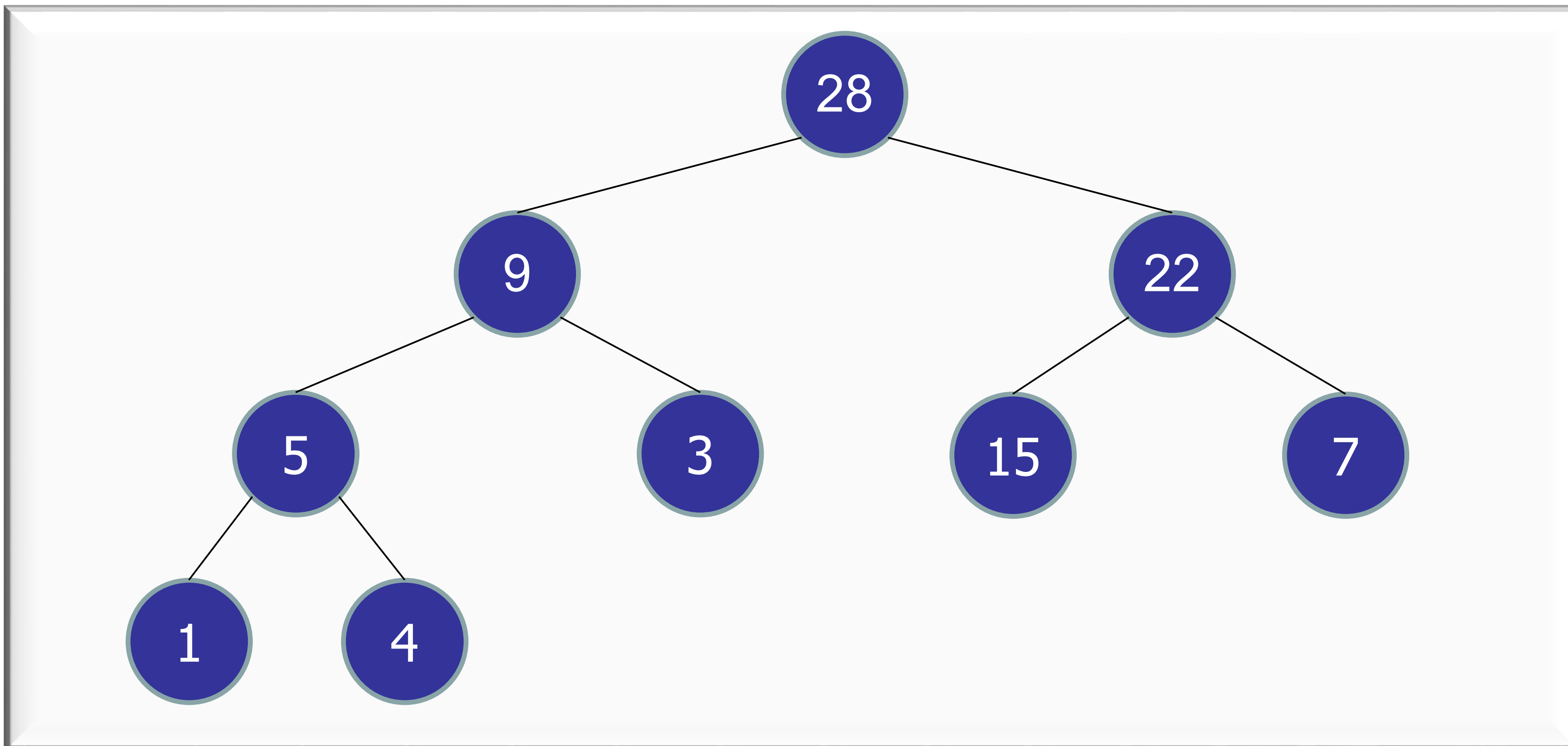
- insert
- extractMax
- increaseKey
- decreaseKey
- delete



Inserting in a Heap

`insert(25)`:

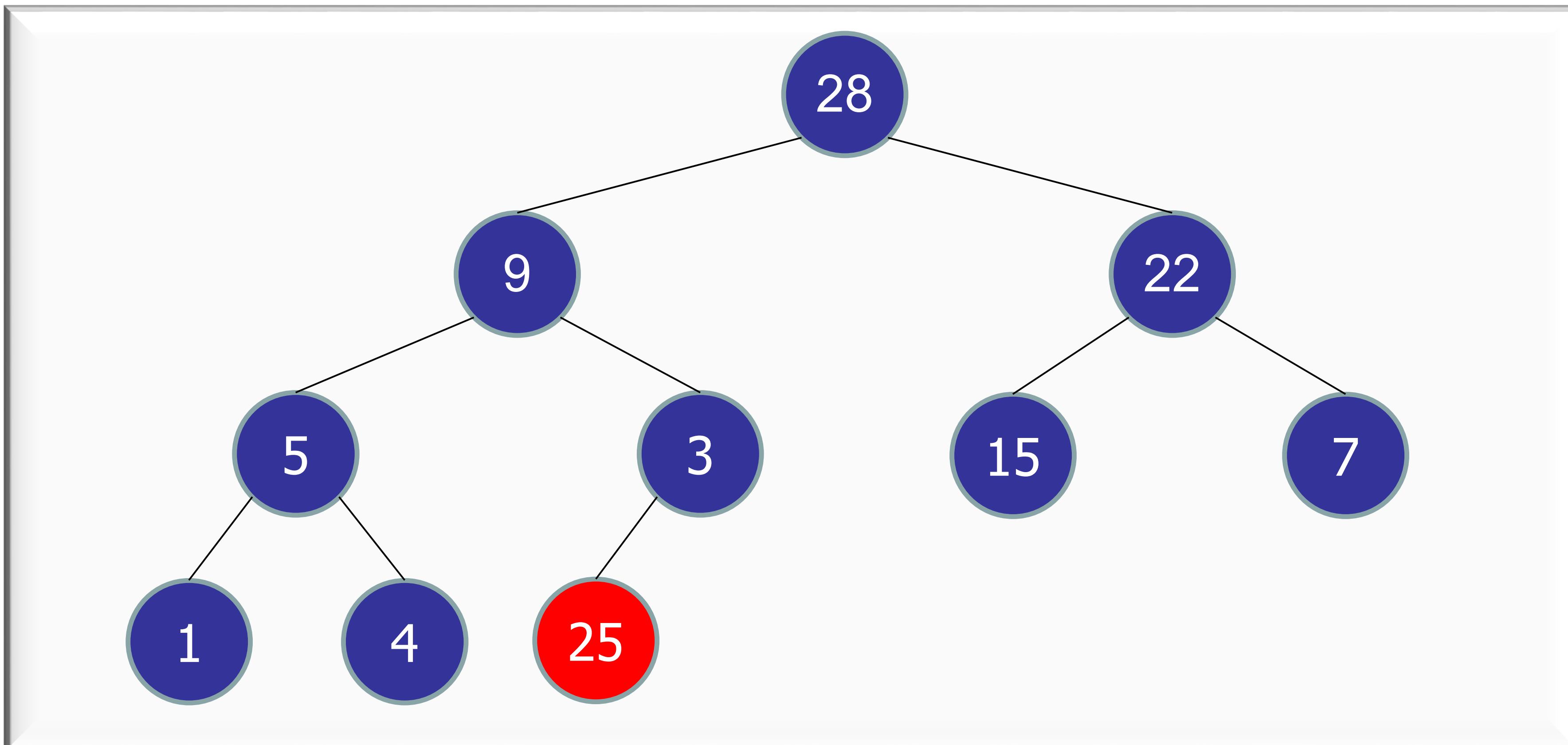
– Step one: add a new leaf with priority 25.



Inserting in a Heap

`insert(25)`:

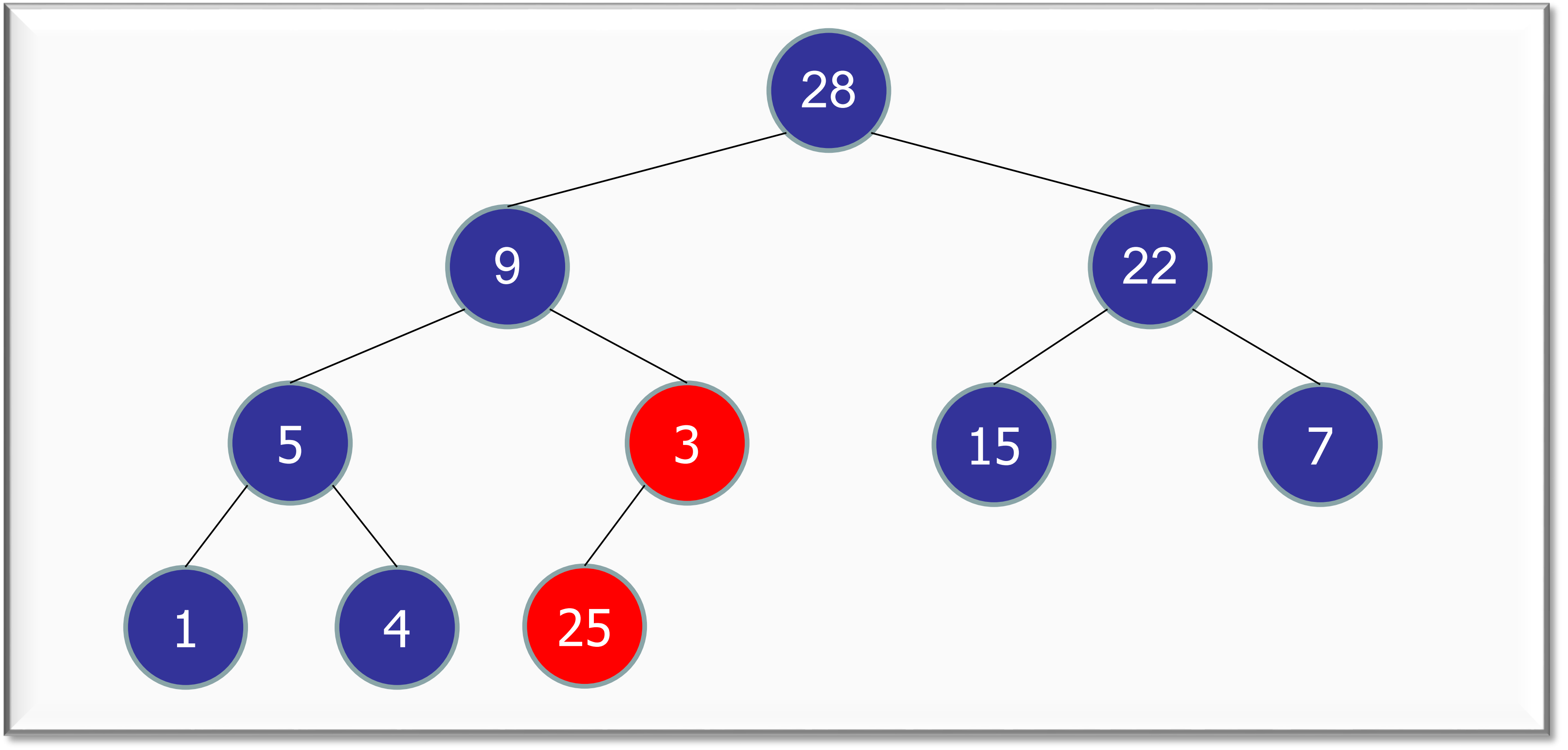
- Step one: add a new leaf with priority 25.



Inserting in a Heap

`insert(25)`:

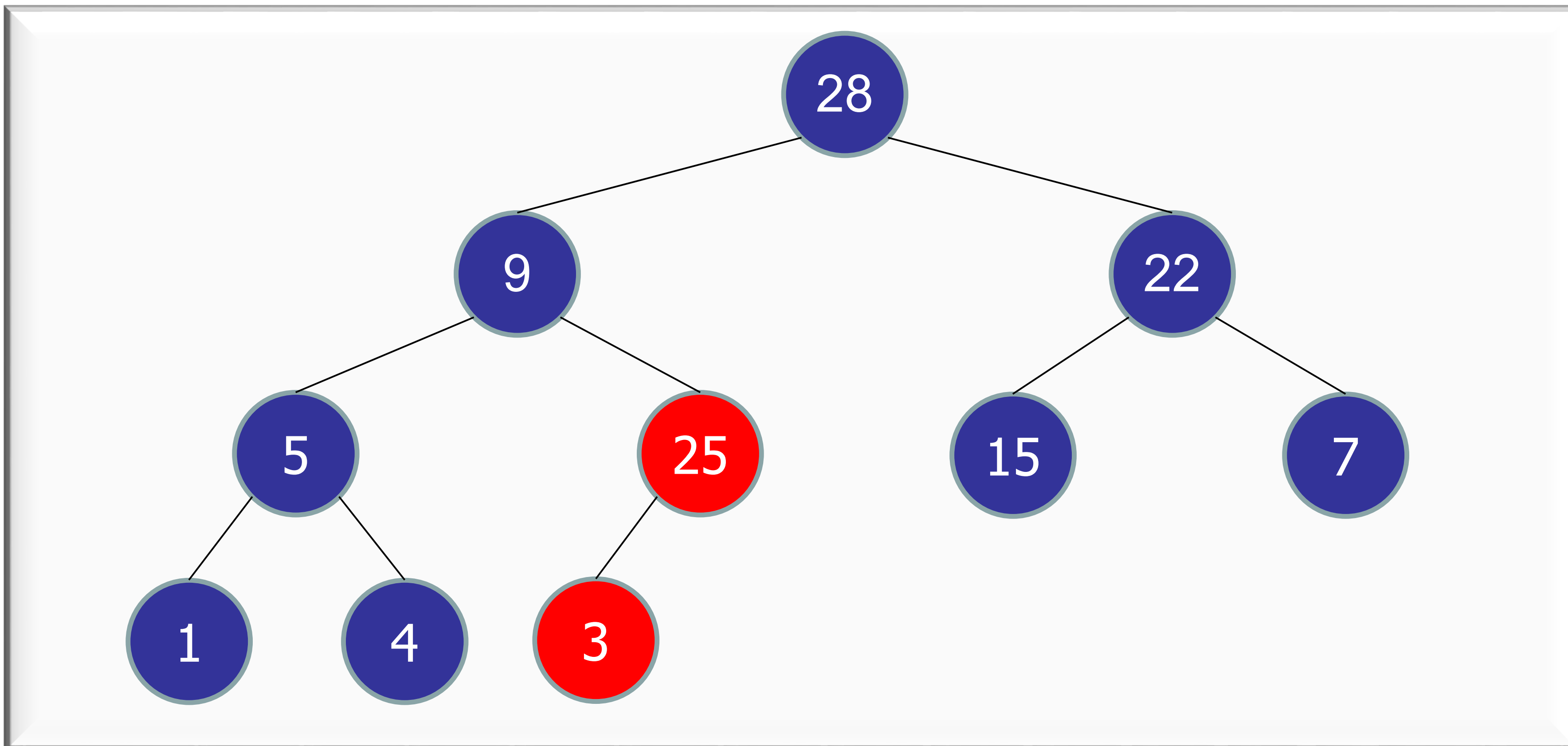
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

`insert(25)`:

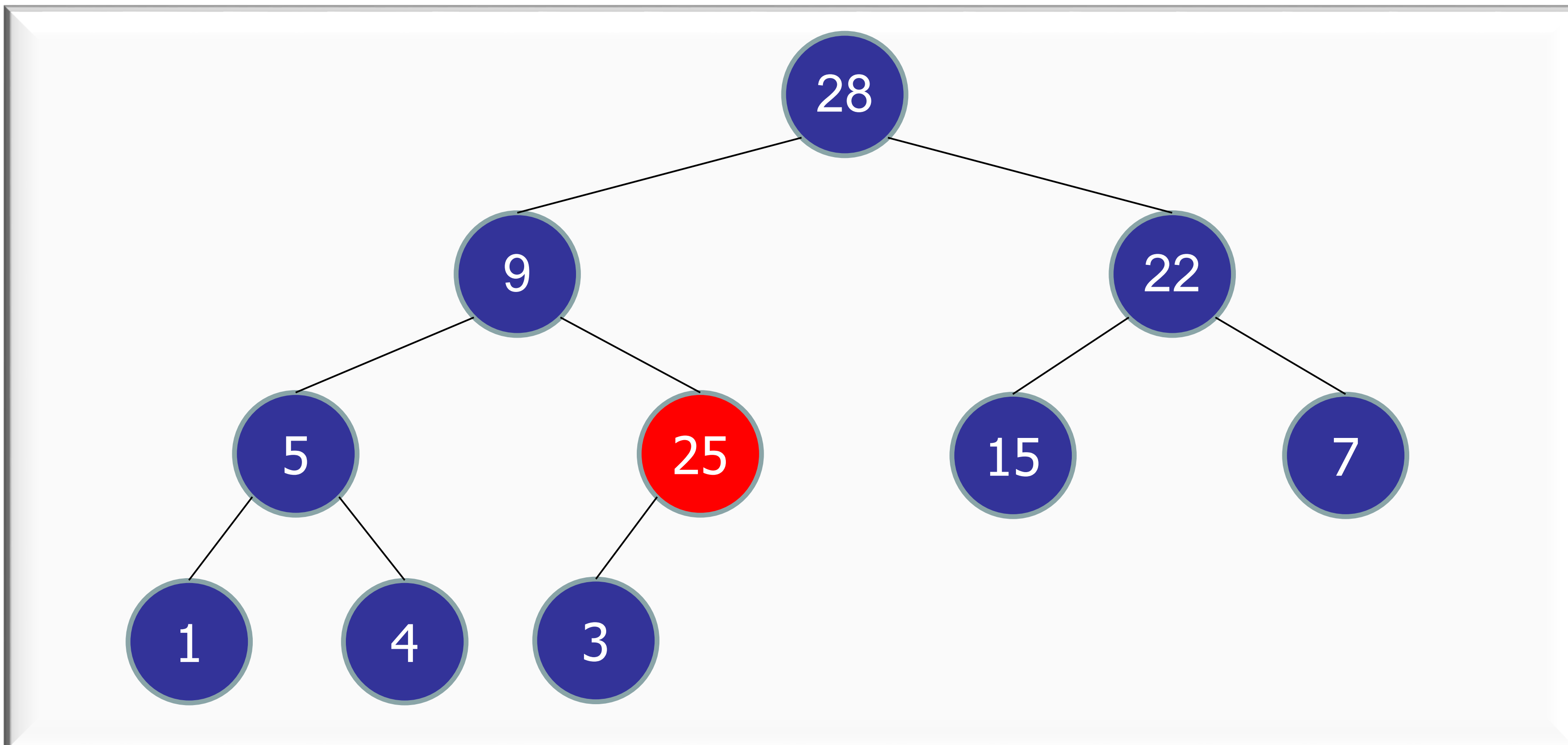
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

`insert(25)`:

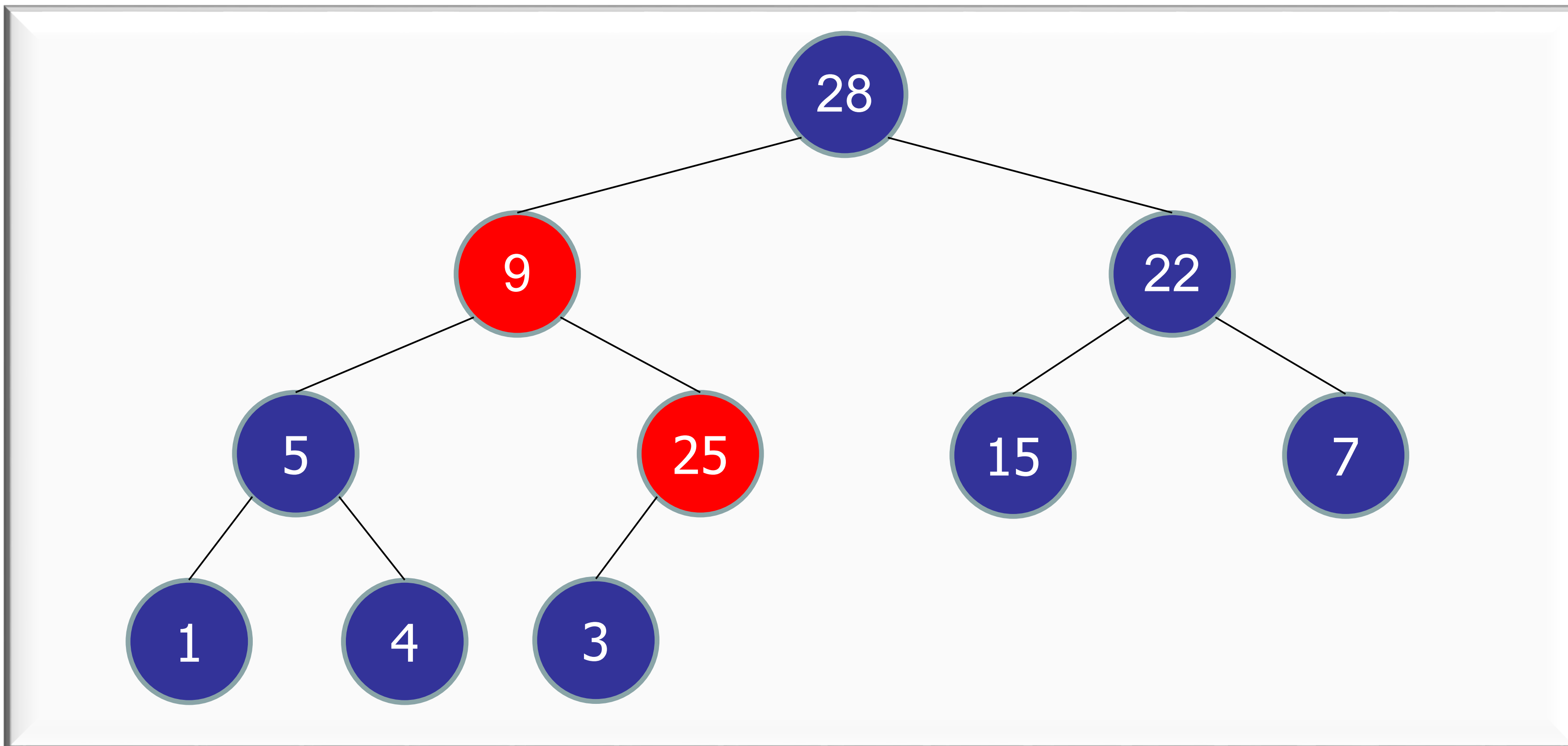
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

`insert(25)`:

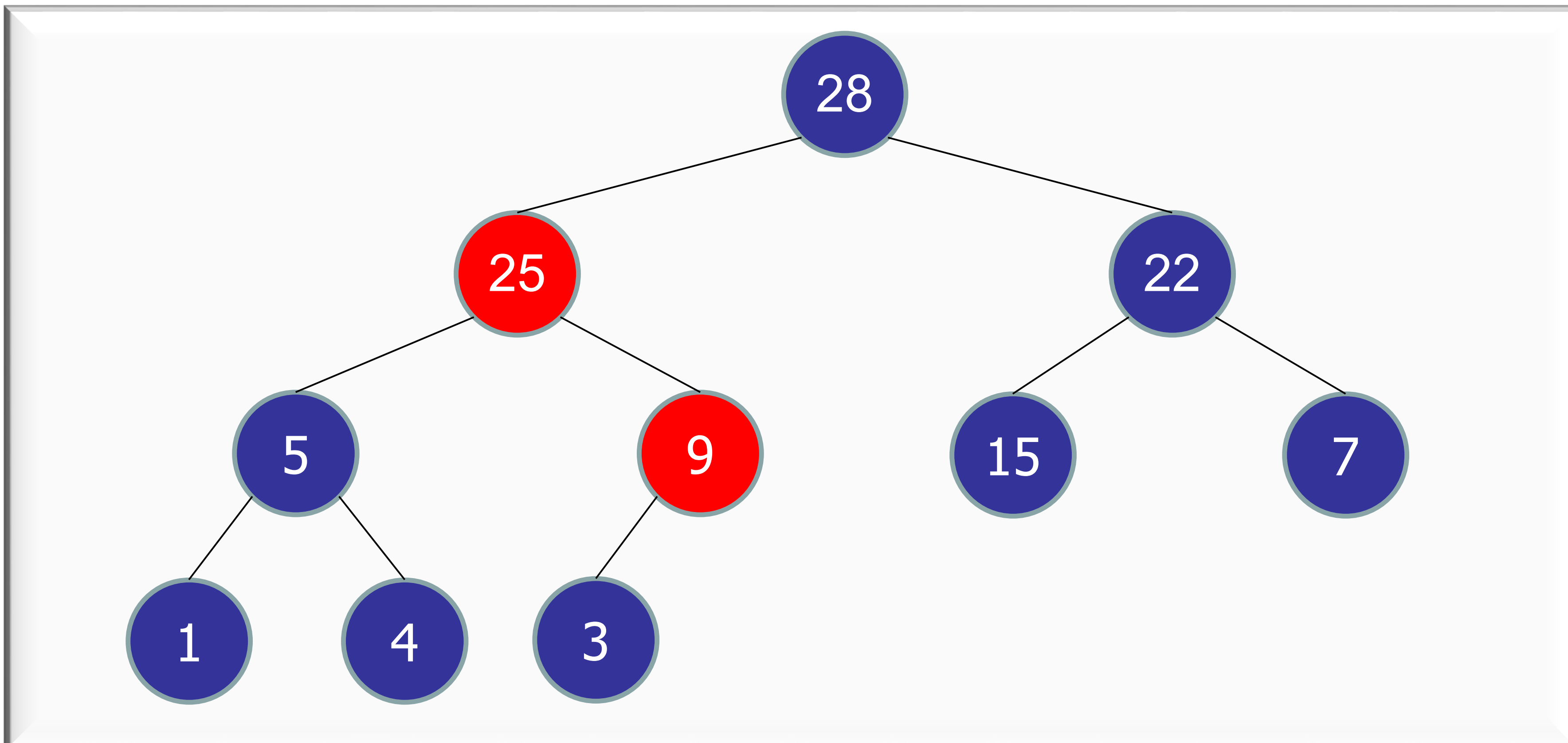
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

`insert(25)`:

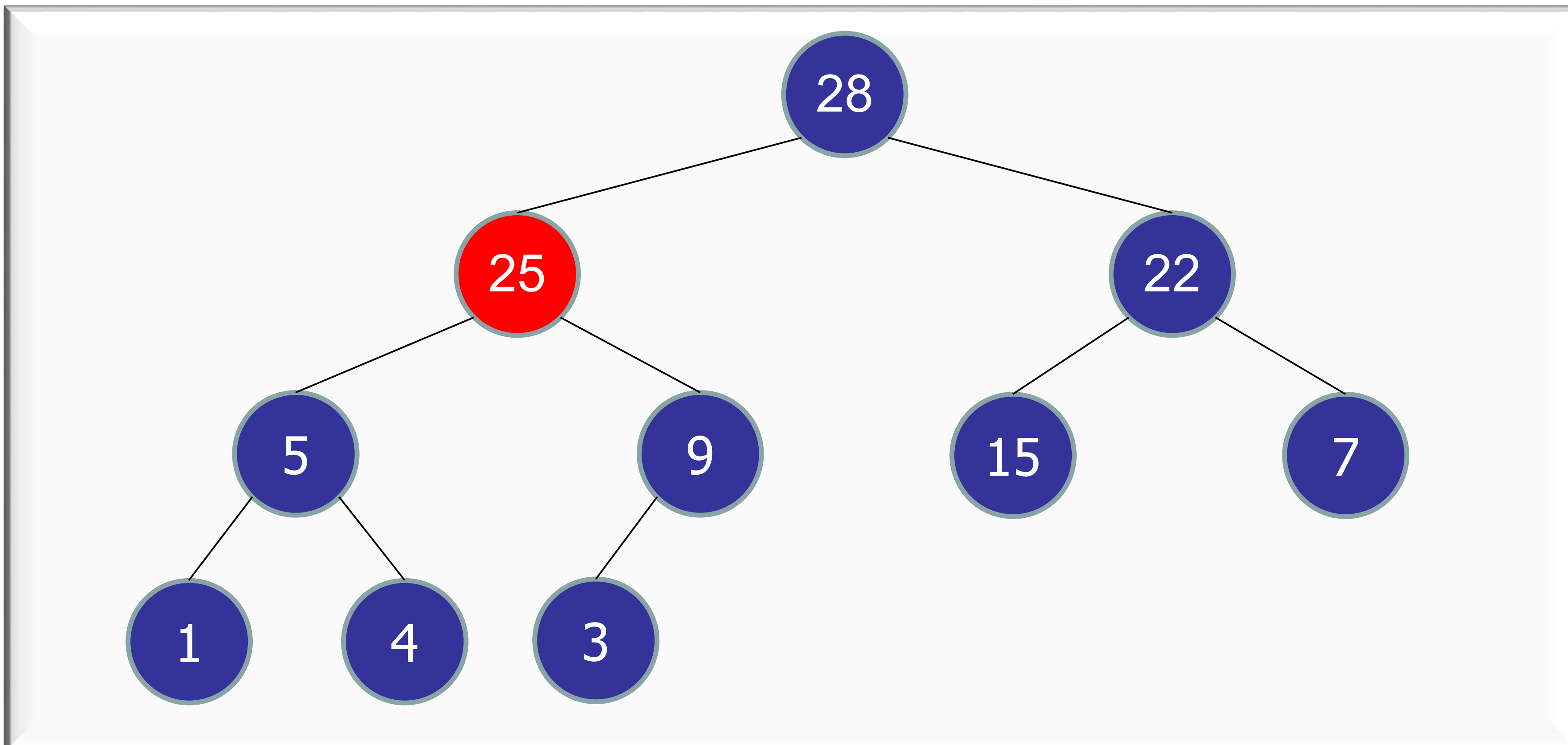
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

`insert(25)`:

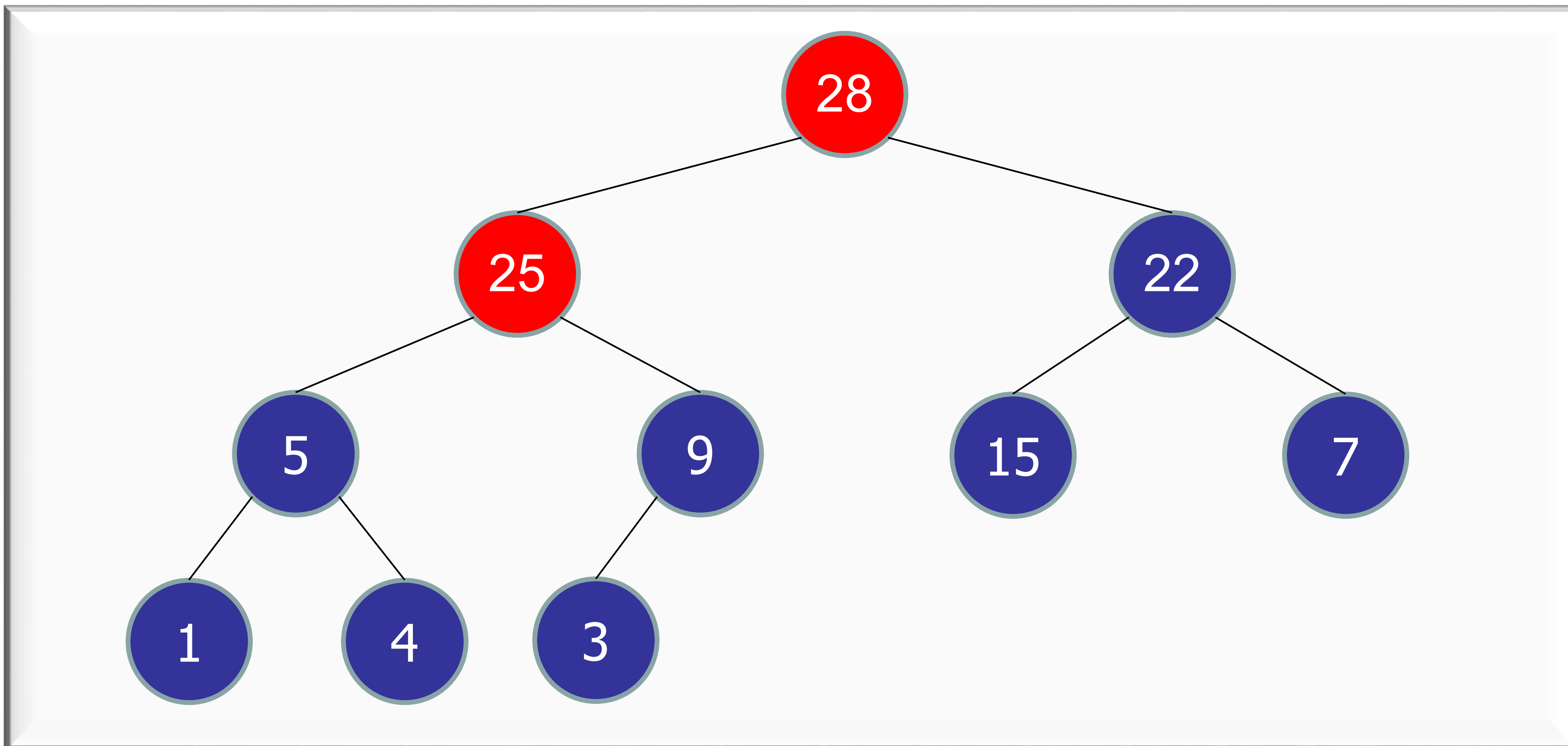
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

`insert(25)`:

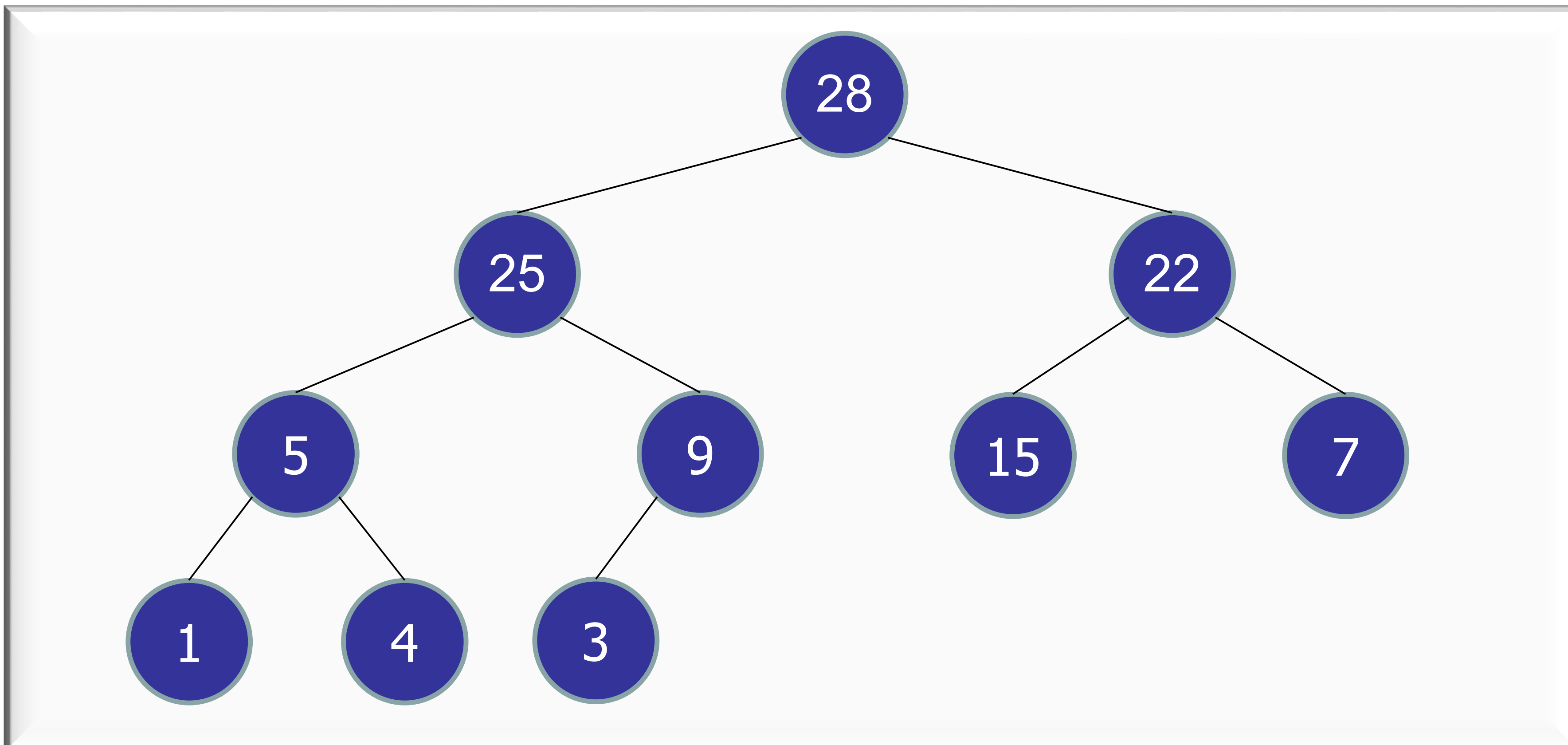
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

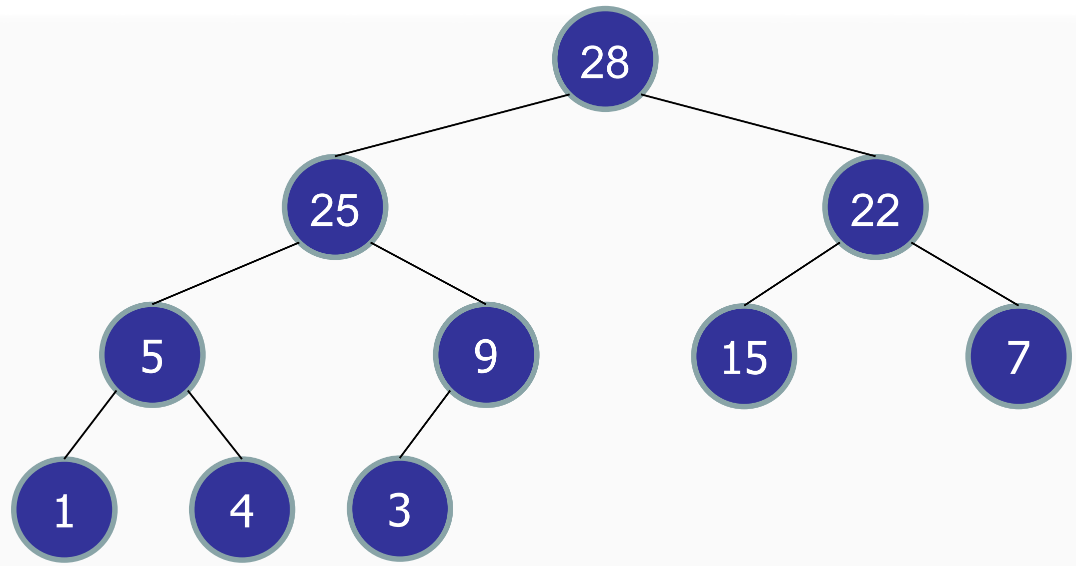
`insert(25)`:

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



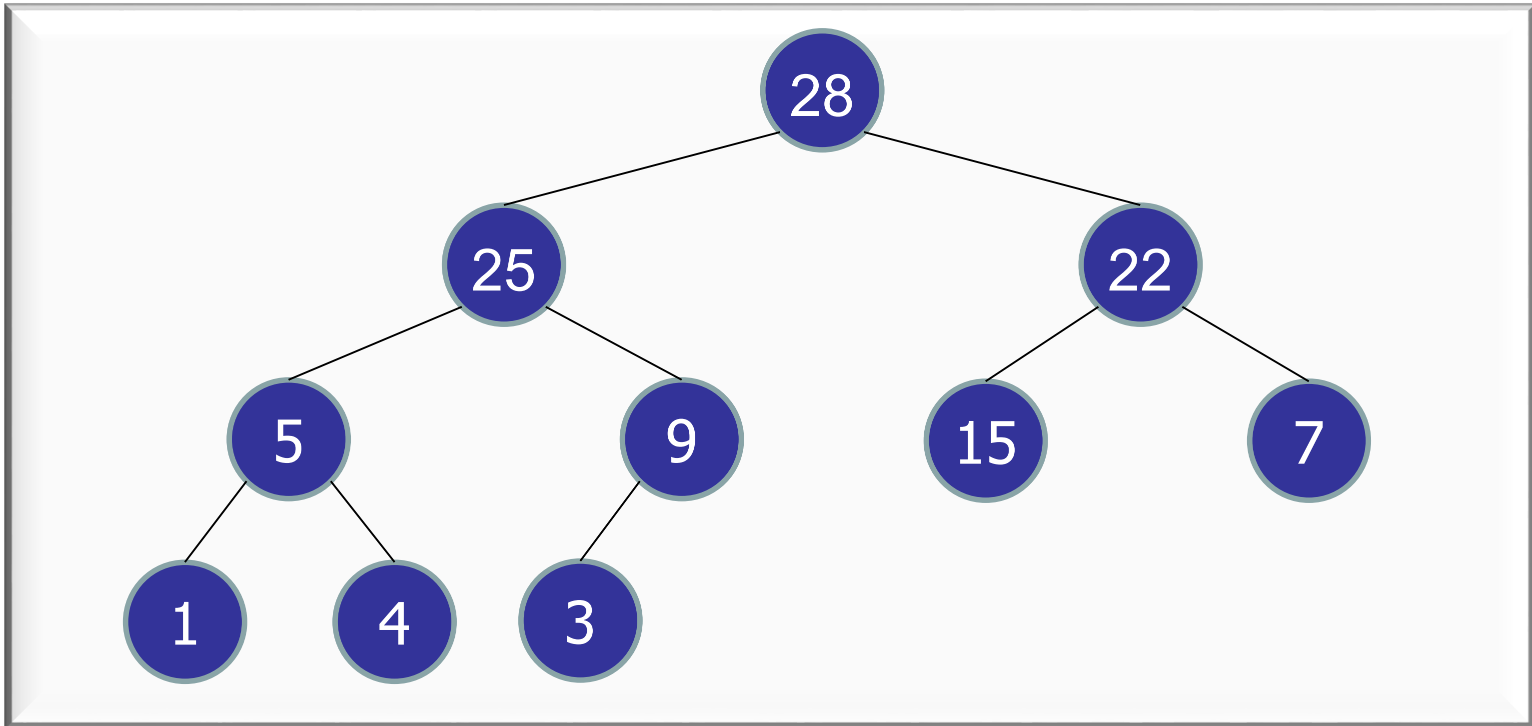
Inserting in a Heap

```
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

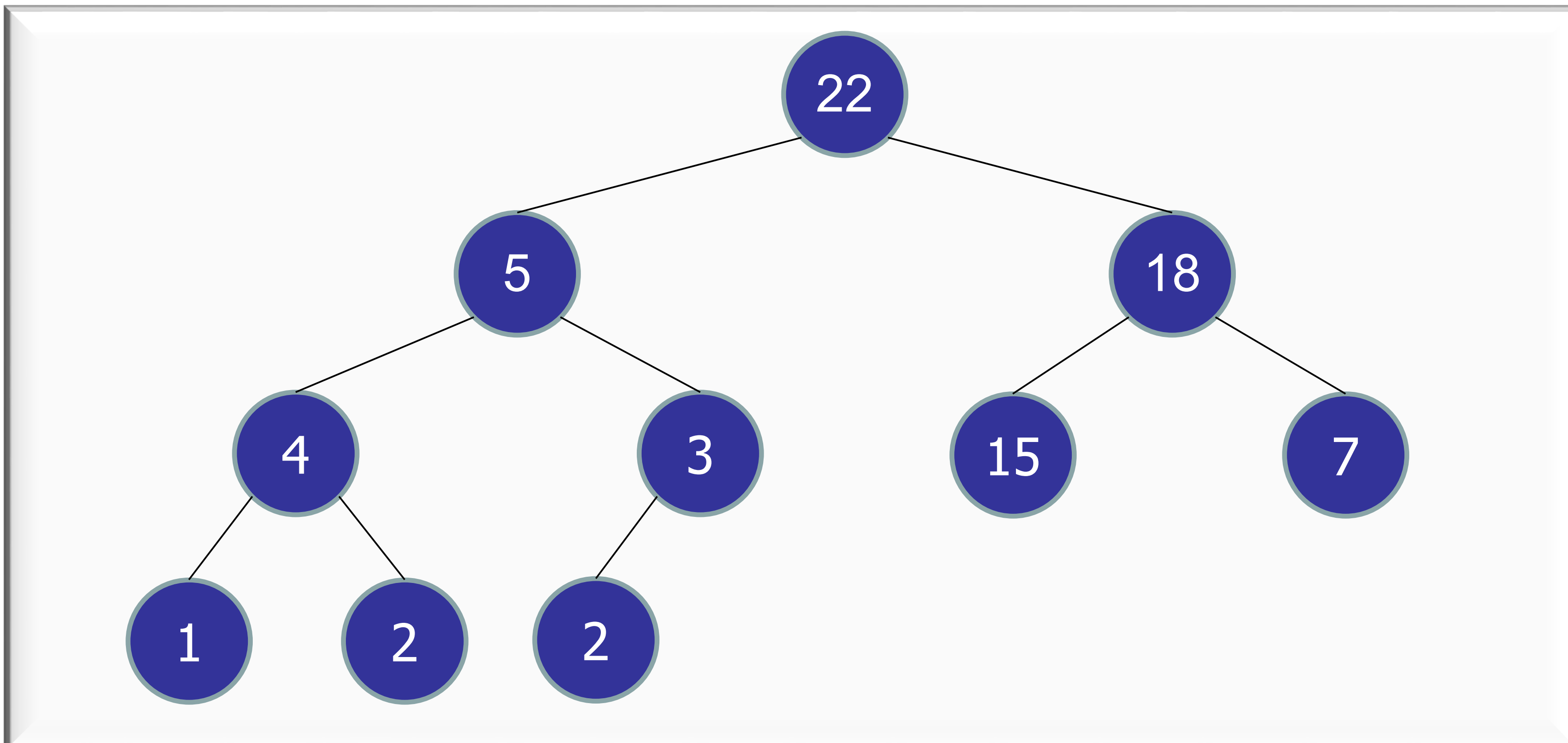
```
insert(Priority p, Key k) {
  Node v = m_completeTree.insert(p,k);
  bubbleUp(v);
}
```



Inserting in a Heap

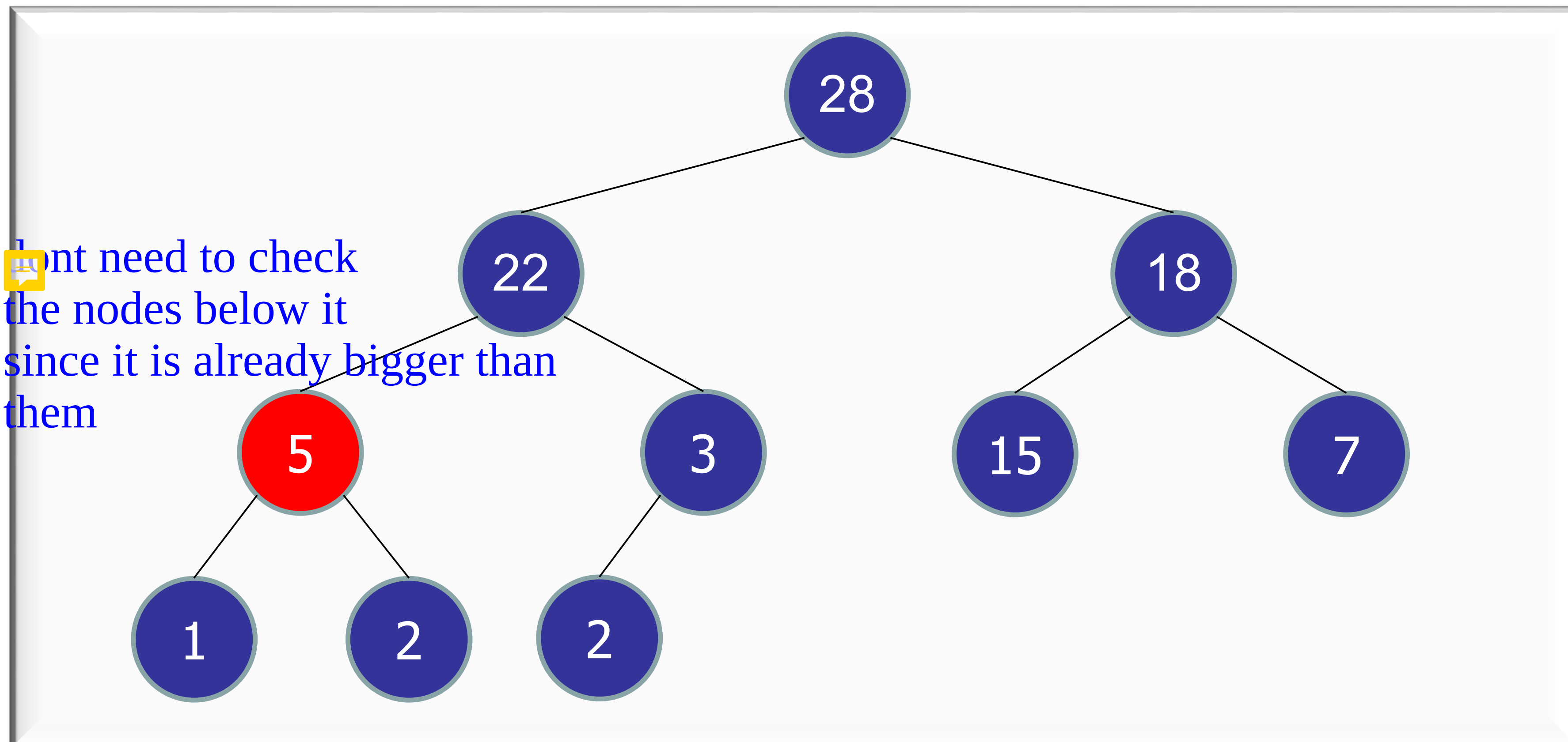
`insert(...)` :

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



Inserting in a Heap

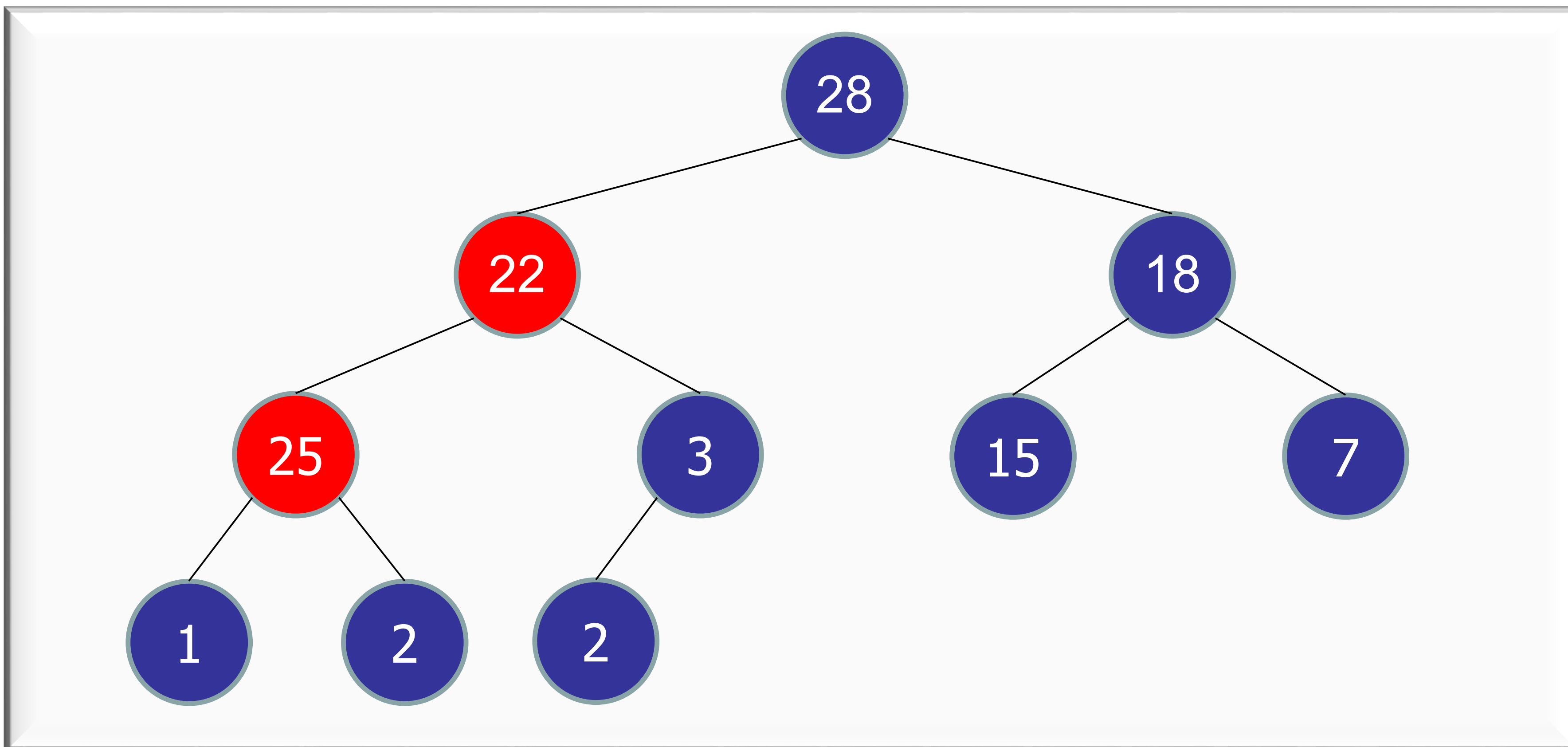
`increaseKey(5 → 25):`



dont need to check the nodes below it since it is already bigger than them

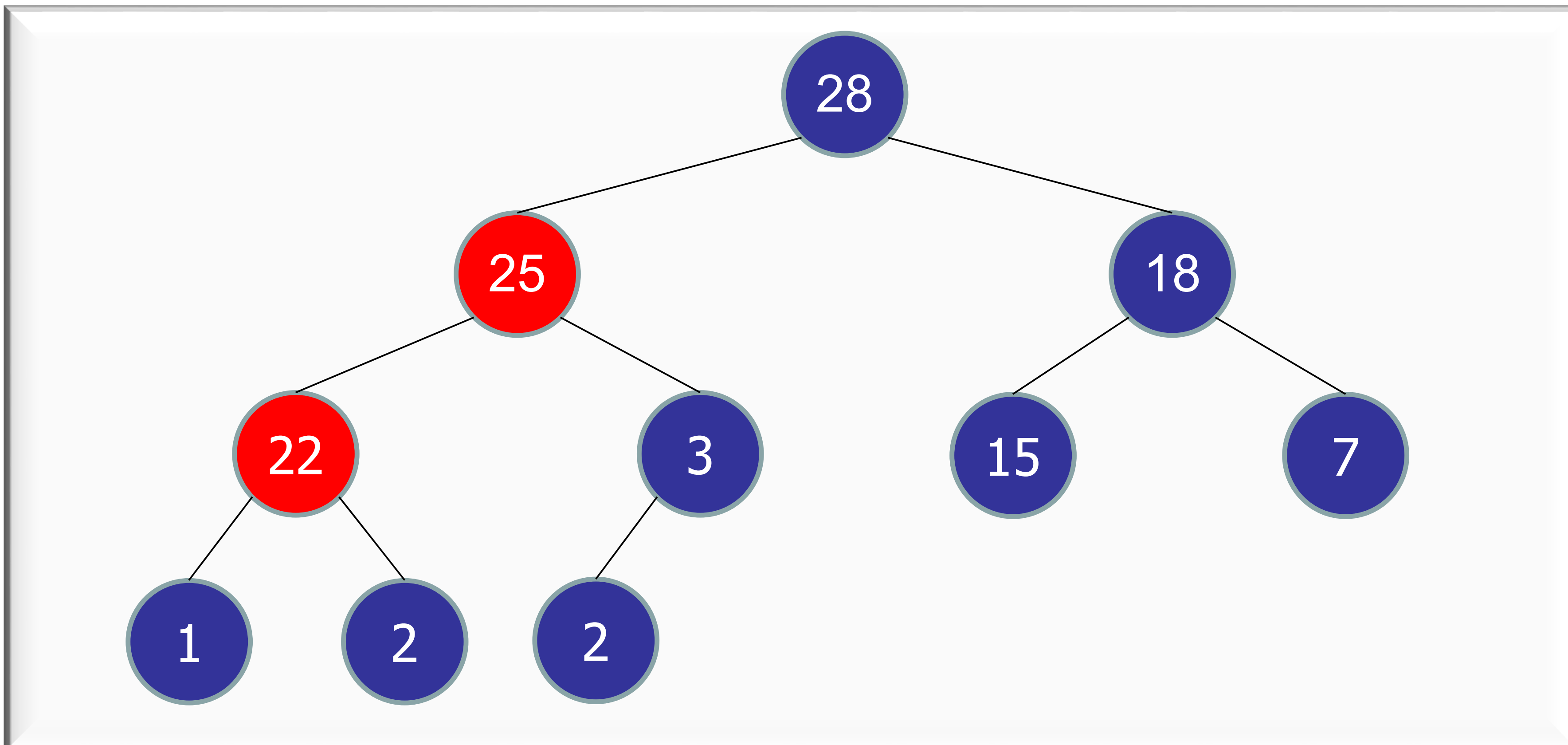
Inserting in a Heap

`increaseKey(5 → 25)`: bubbleUp(25)



Inserting in a Heap

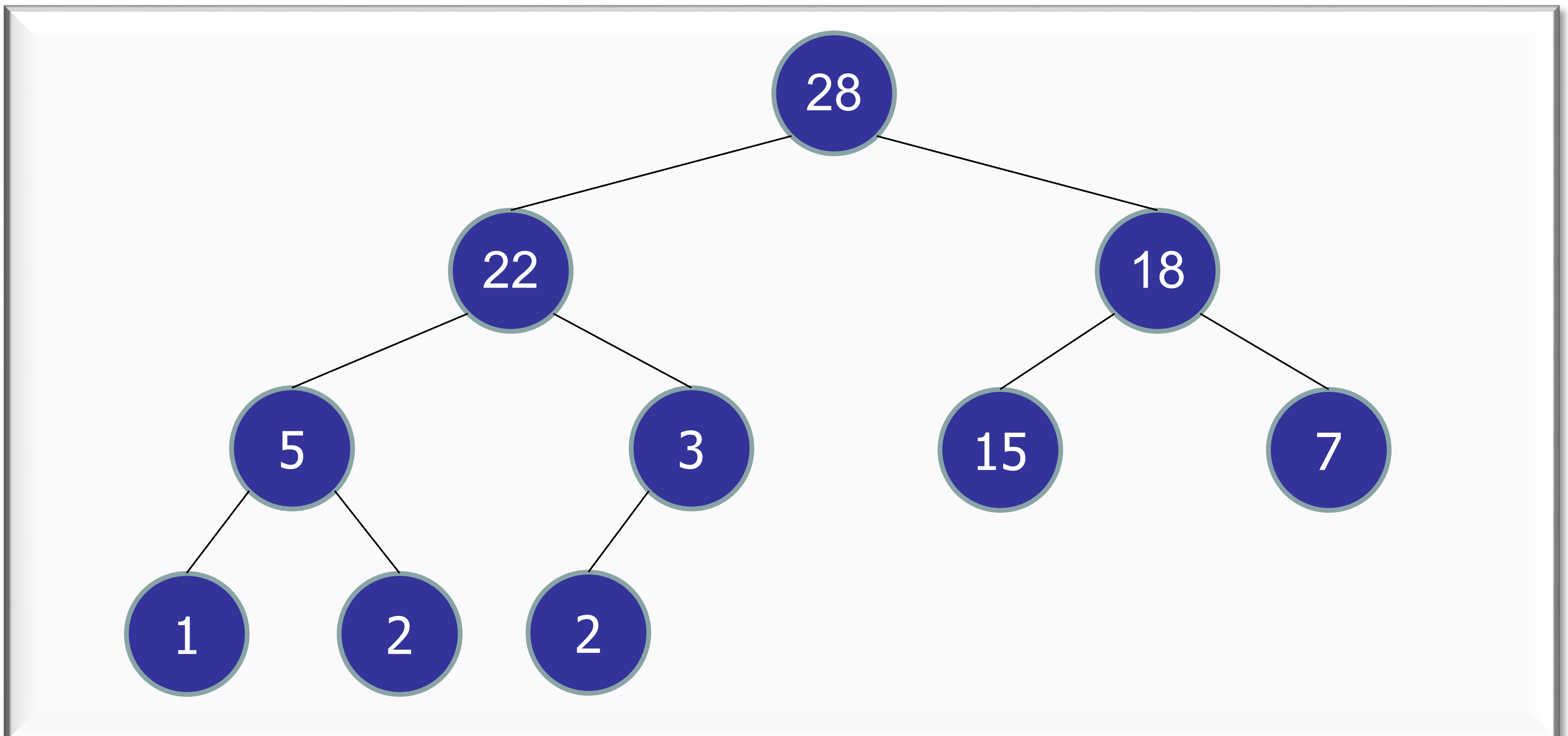
`increaseKey(5 → 25)`: bubbleUp(25)



Inserting in a Heap

`decreaseKey(28 → 4):`

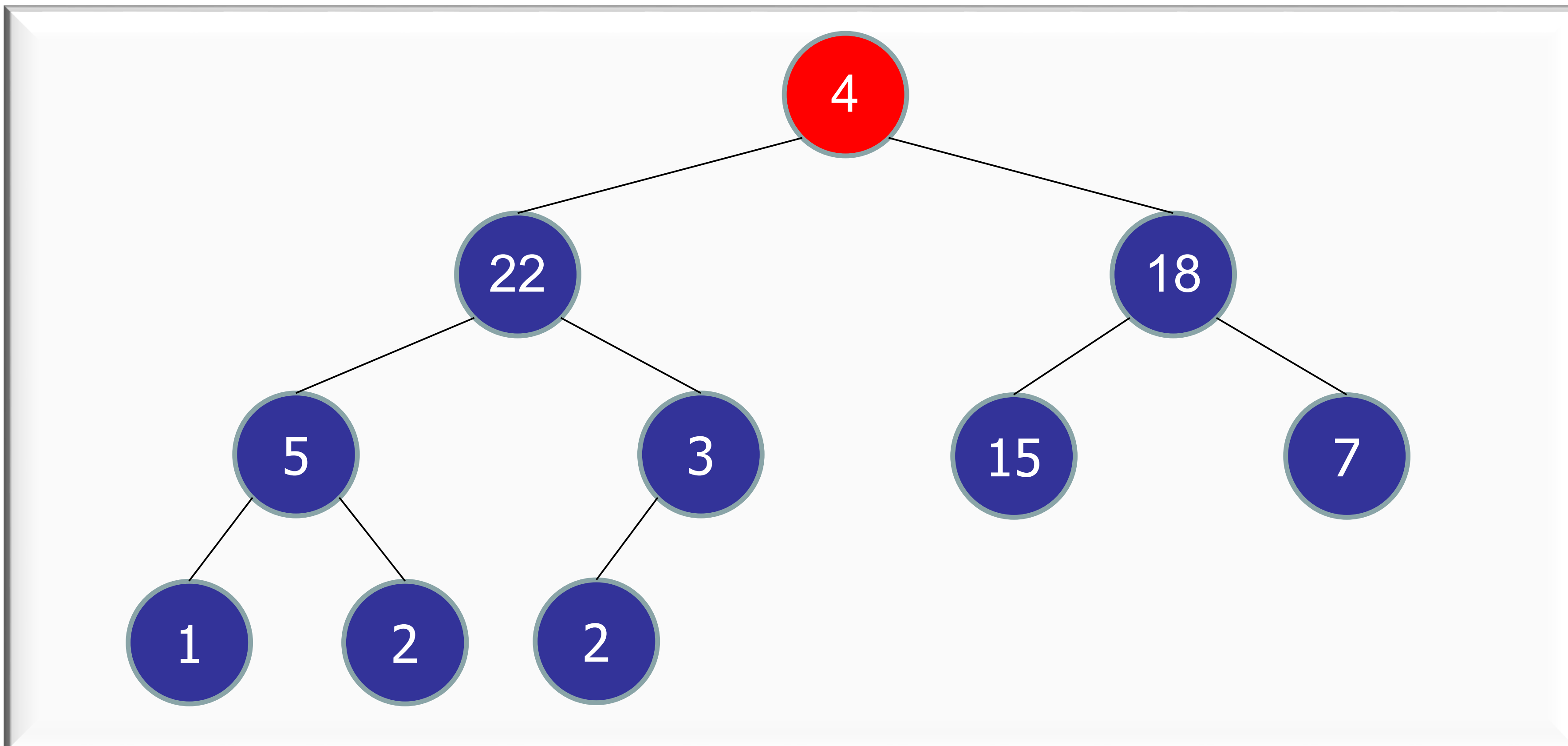
Only need to consider bubble down



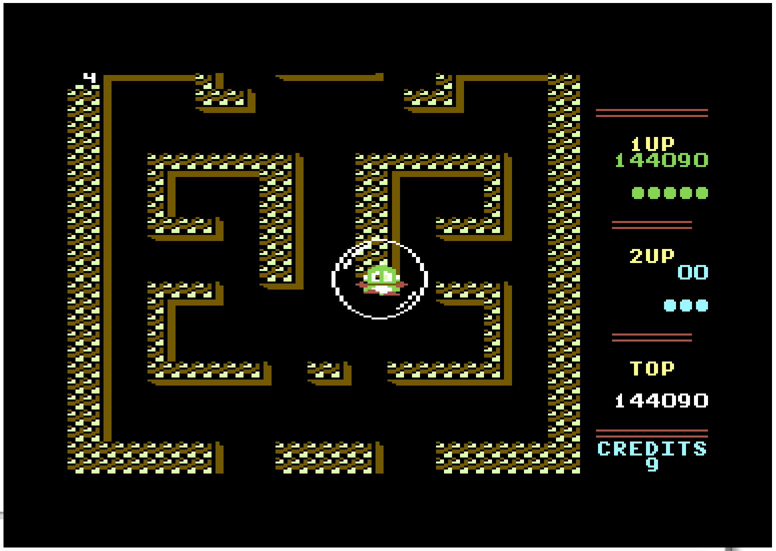
Inserting in a Heap

`decreaseKey(28 → 4)`:

- Step 1: Update the priority

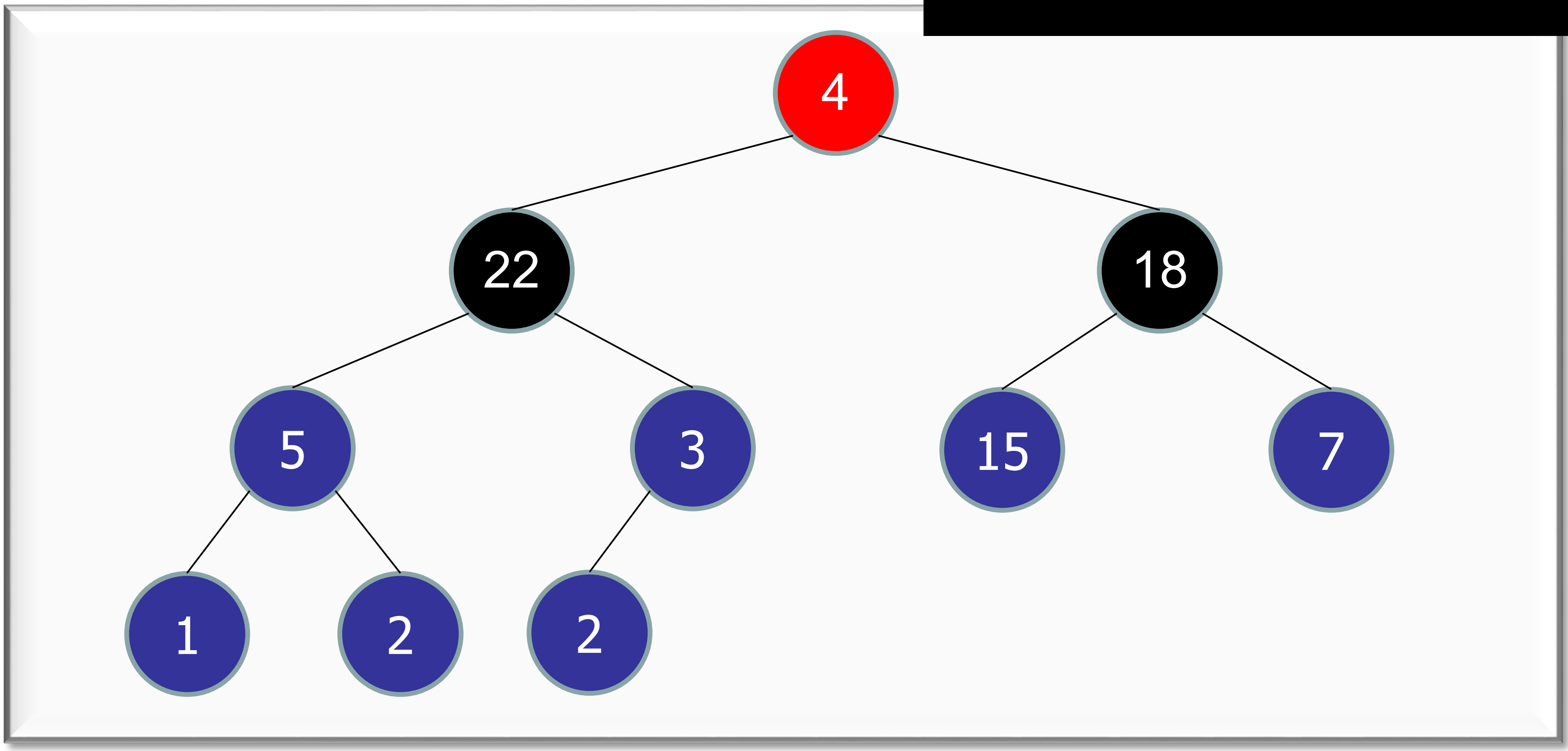


Inserting in a Heap



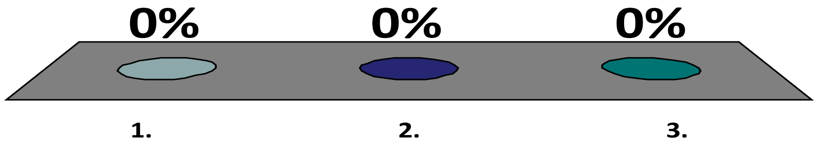
`decreaseKey(28 → 4):`

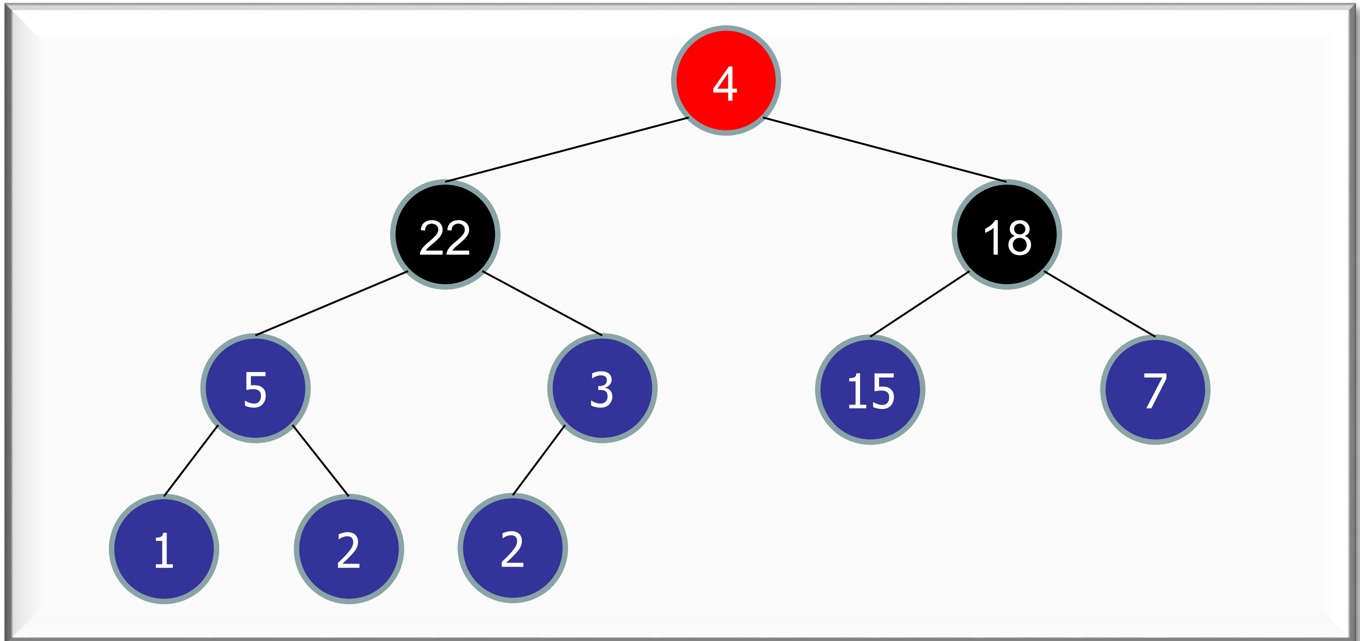
- Step 1: Update the priority
- Step 2: bubbleDown(4)



Which way to bubbleDown?

✓ 1. Larger child (22)
2. Smaller child (18)
3. Does not matter

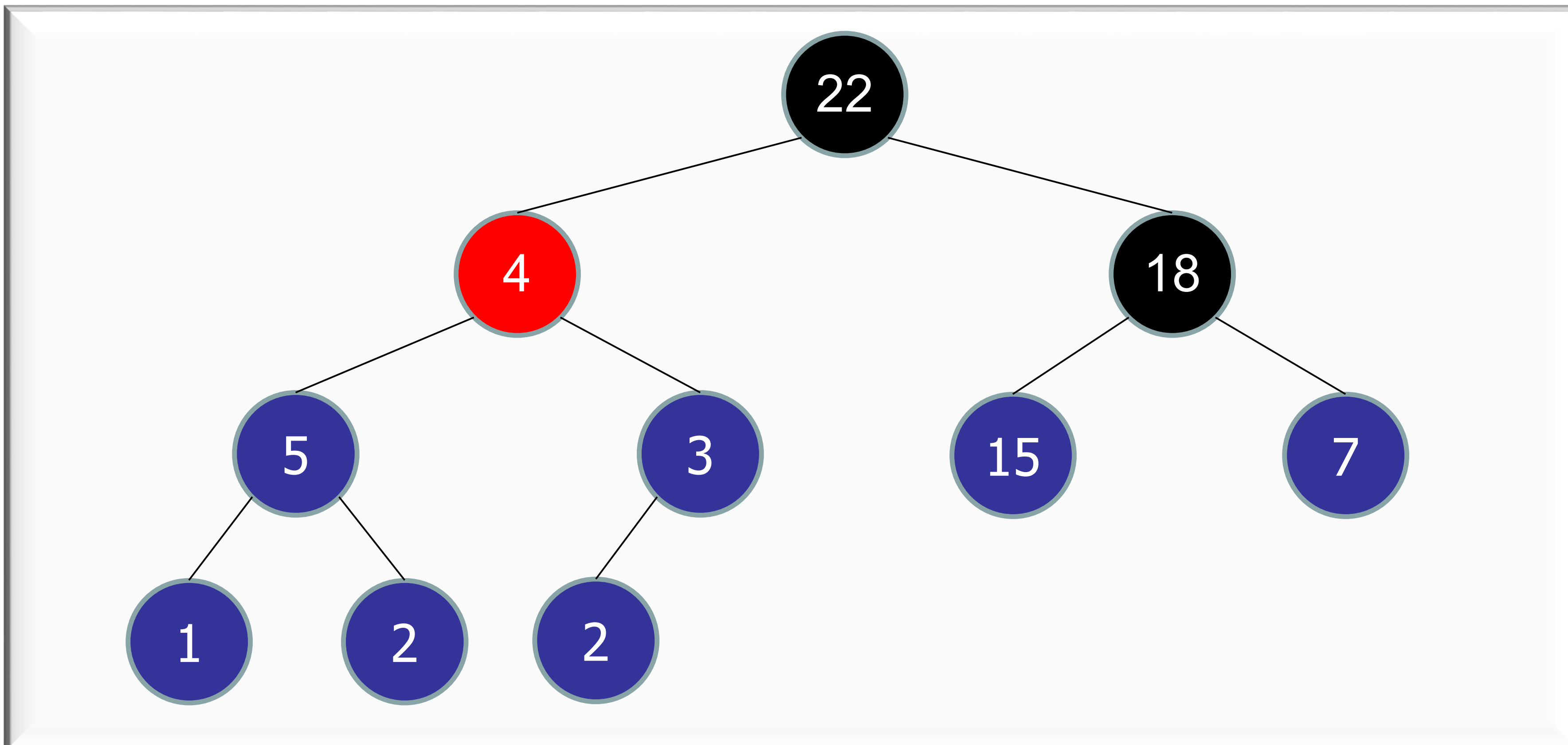




Inserting in a Heap

`decreaseKey(28 → 4)` :

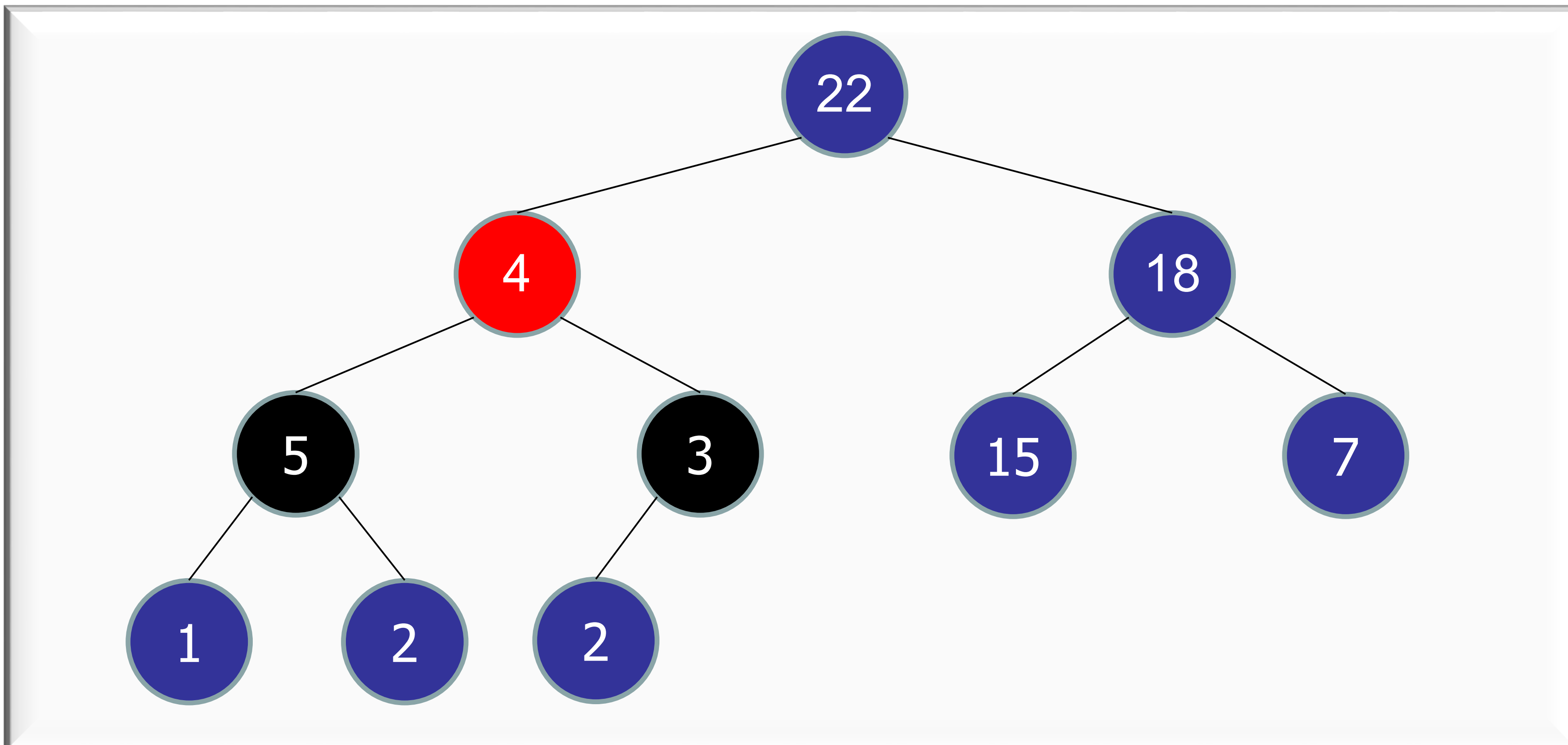
- Step 1: Update the priority
- Step 2: bubbleDown(4)



Inserting in a Heap

`decreaseKey(28 → 4)` :

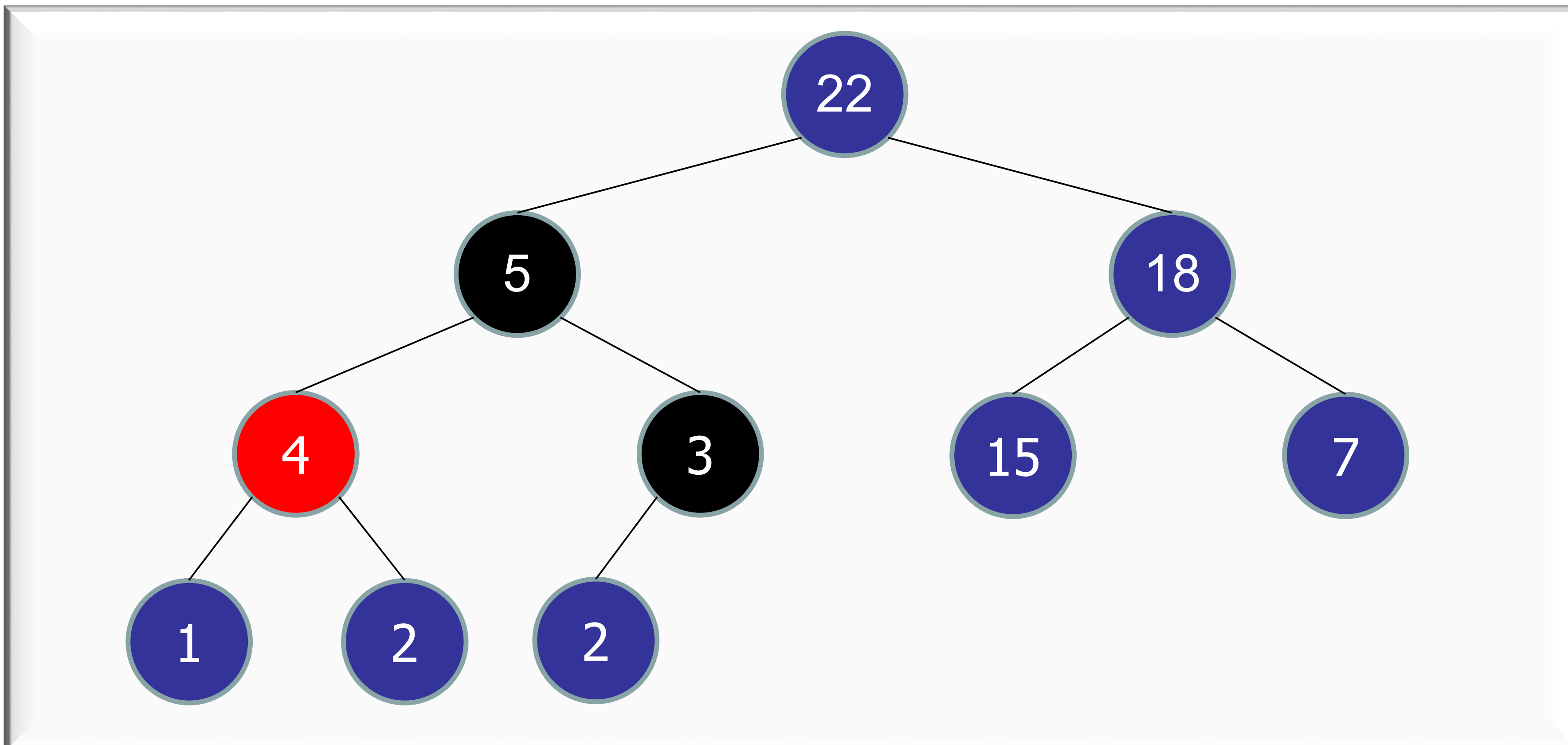
- Step 1: Update the priority
- Step 2: bubbleDown(4)



Inserting in a Heap

`decreaseKey(28 → 4)` :

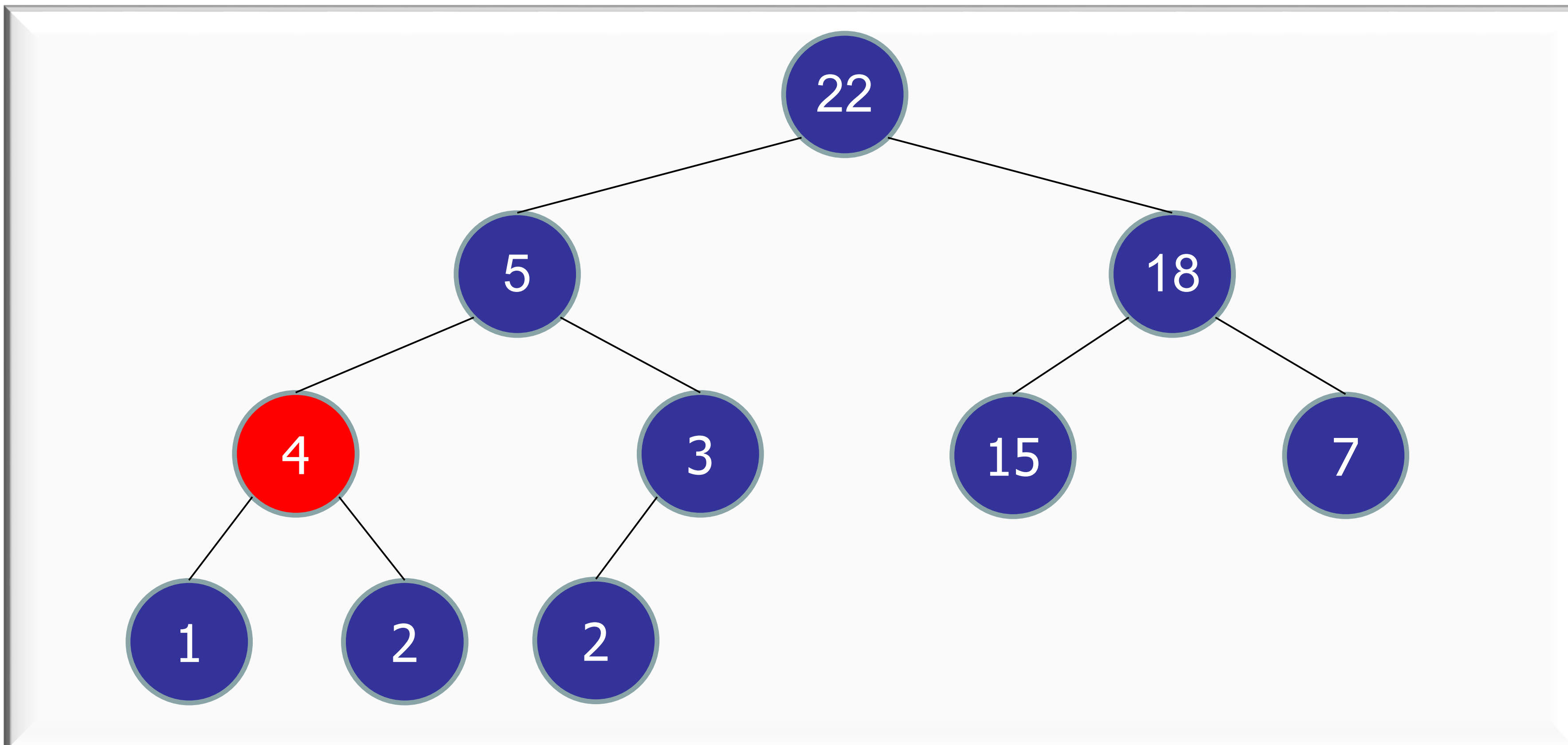
- Step 1: Update the priority
- Step 2: bubbleDown(4)



Inserting in a Heap

`decreaseKey(28 → 4)` :

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



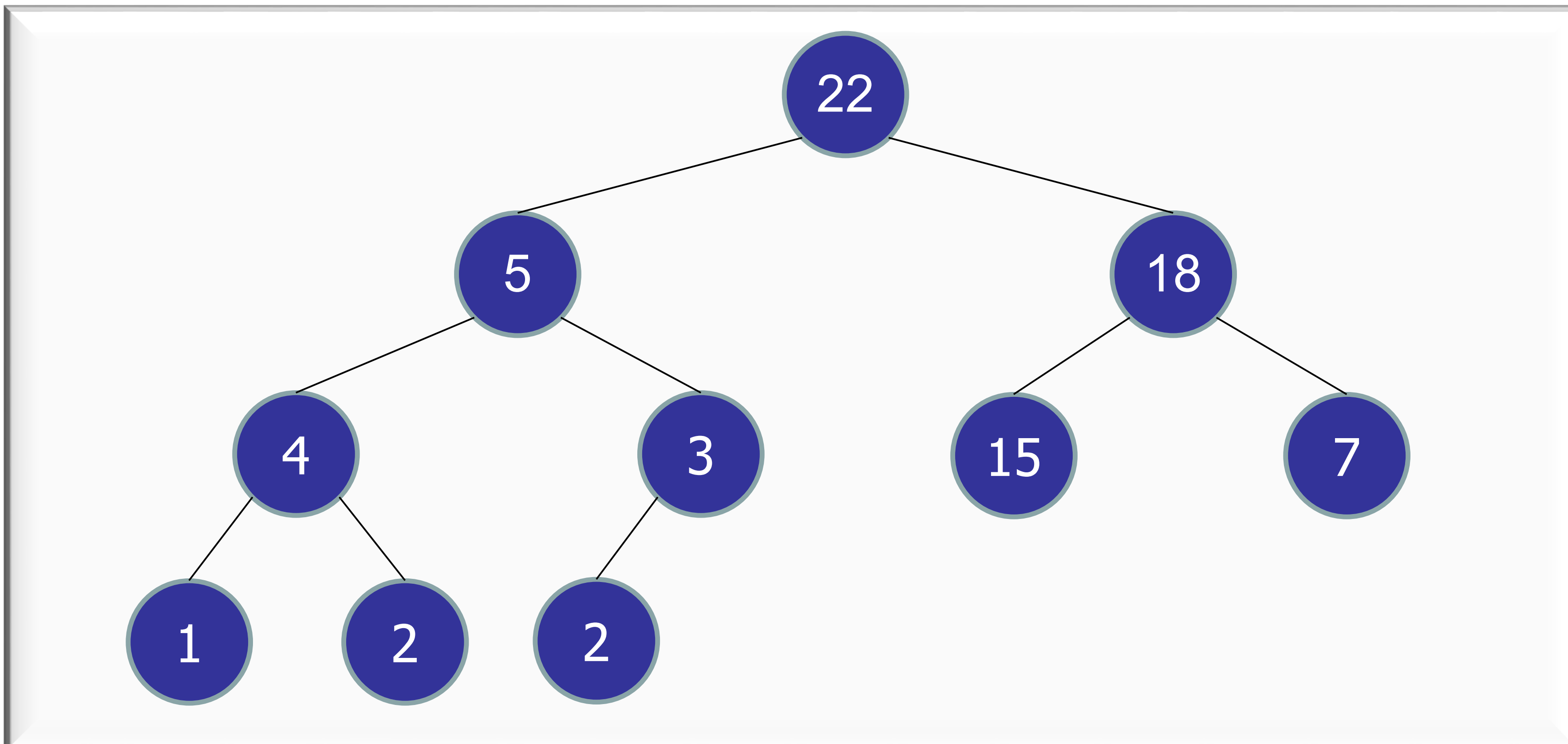
Inserting in a Heap

```
bubbleDown(Node v)
  while (!leaf(v)) {
     leftP = priority(left(v));
     rightP = priority(right(v));
     biggerChild = leftP > rightP ? left(v): right(v);
     if(priority(biggerChild) > priority(v))
     {
          swap(v,biggerChild);
          v = biggerChild;
     } else
          return;
  }
```

Inserting in a Heap

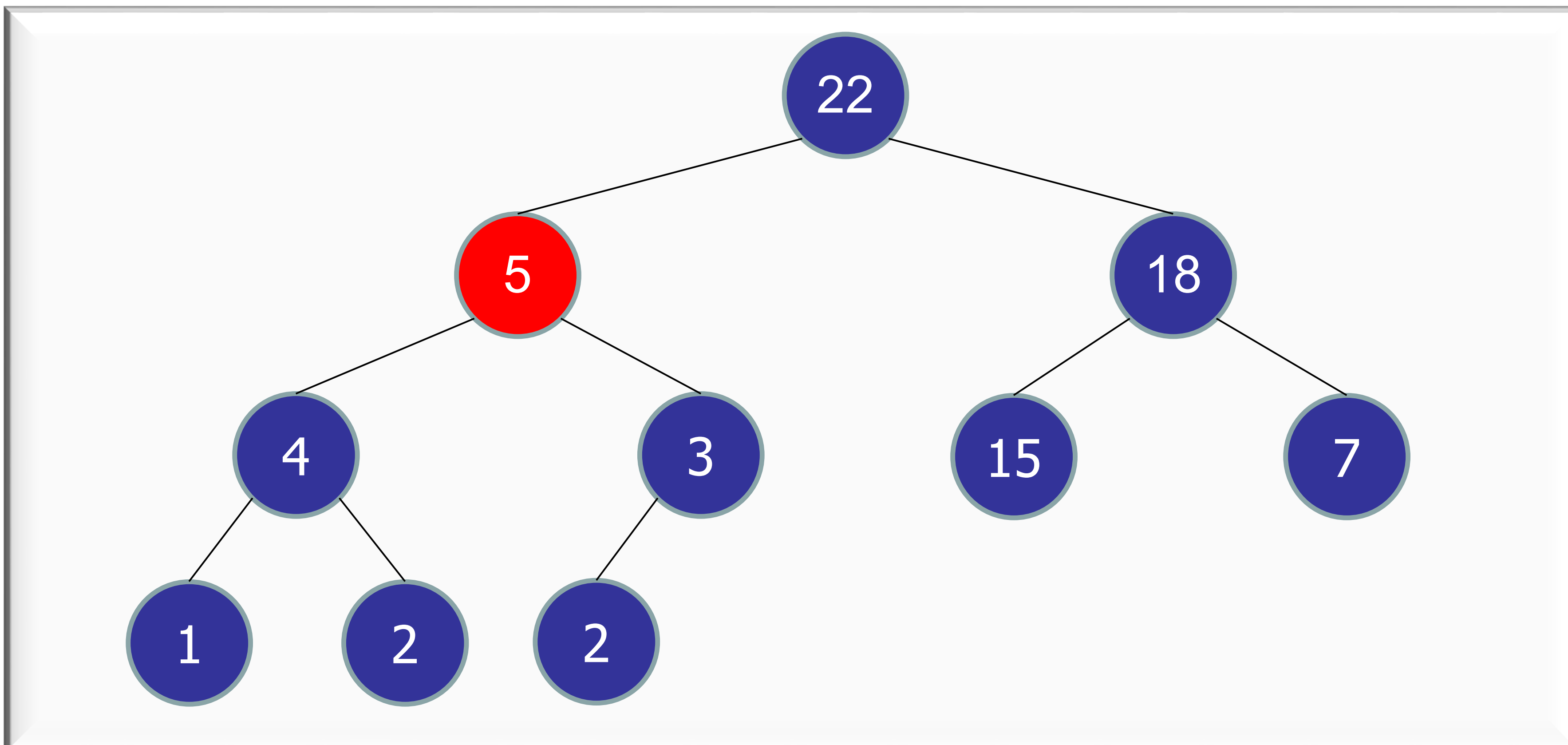
`decreaseKey(. . .)` :

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



Inserting in a Heap

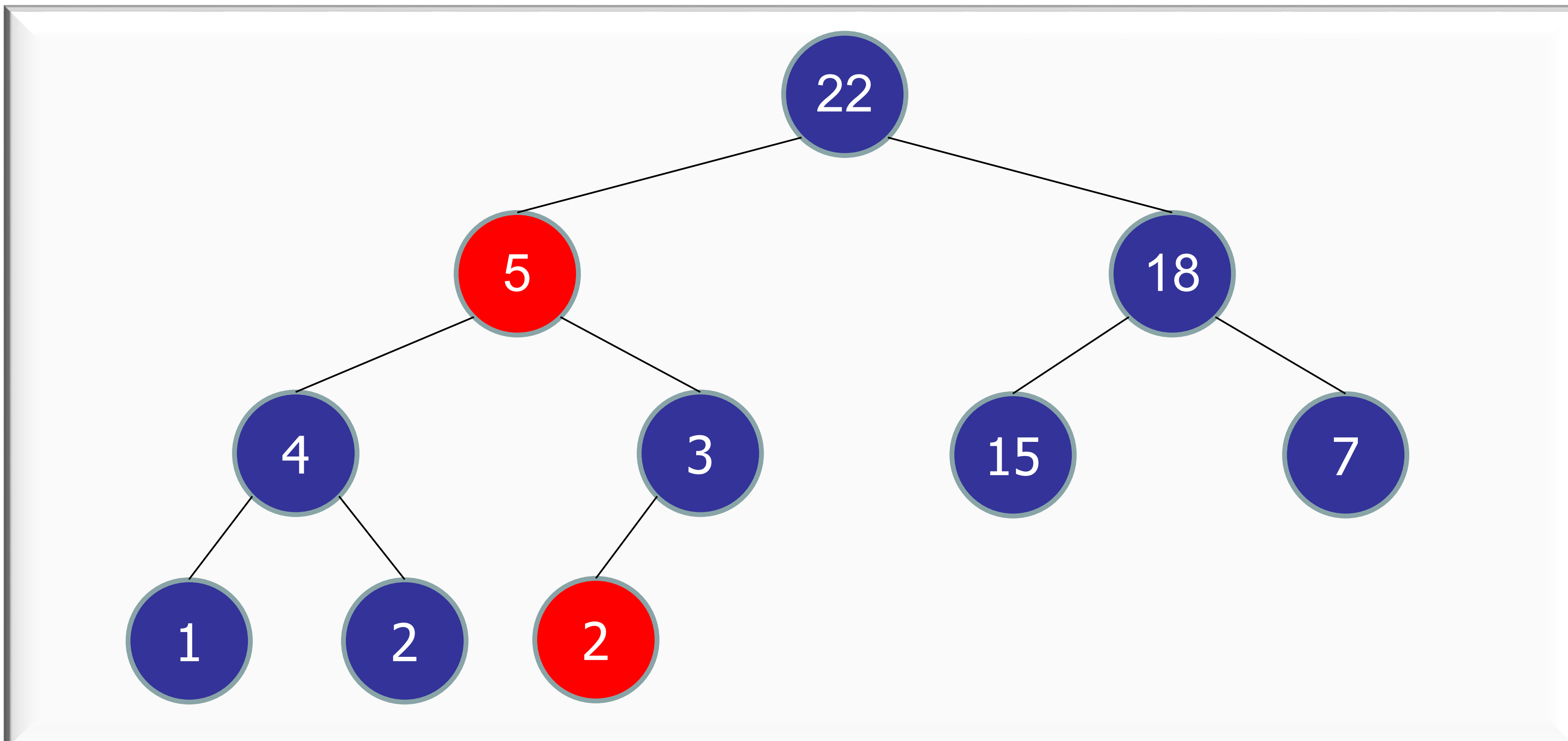
`delete(5)` :



Inserting in a Heap

`delete(5)` :

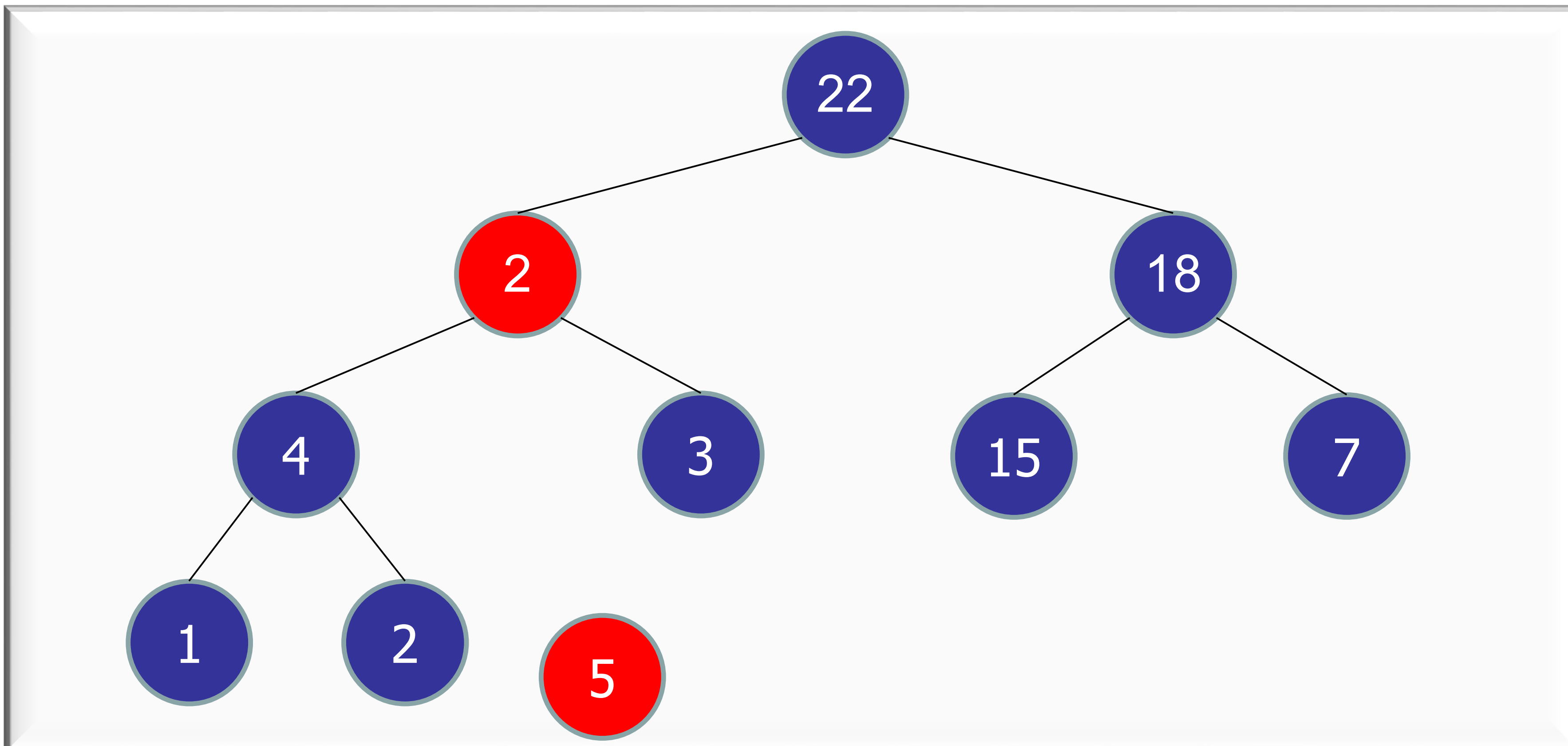
- `swap(5, last())`



Inserting in a Heap

`delete(5)` :

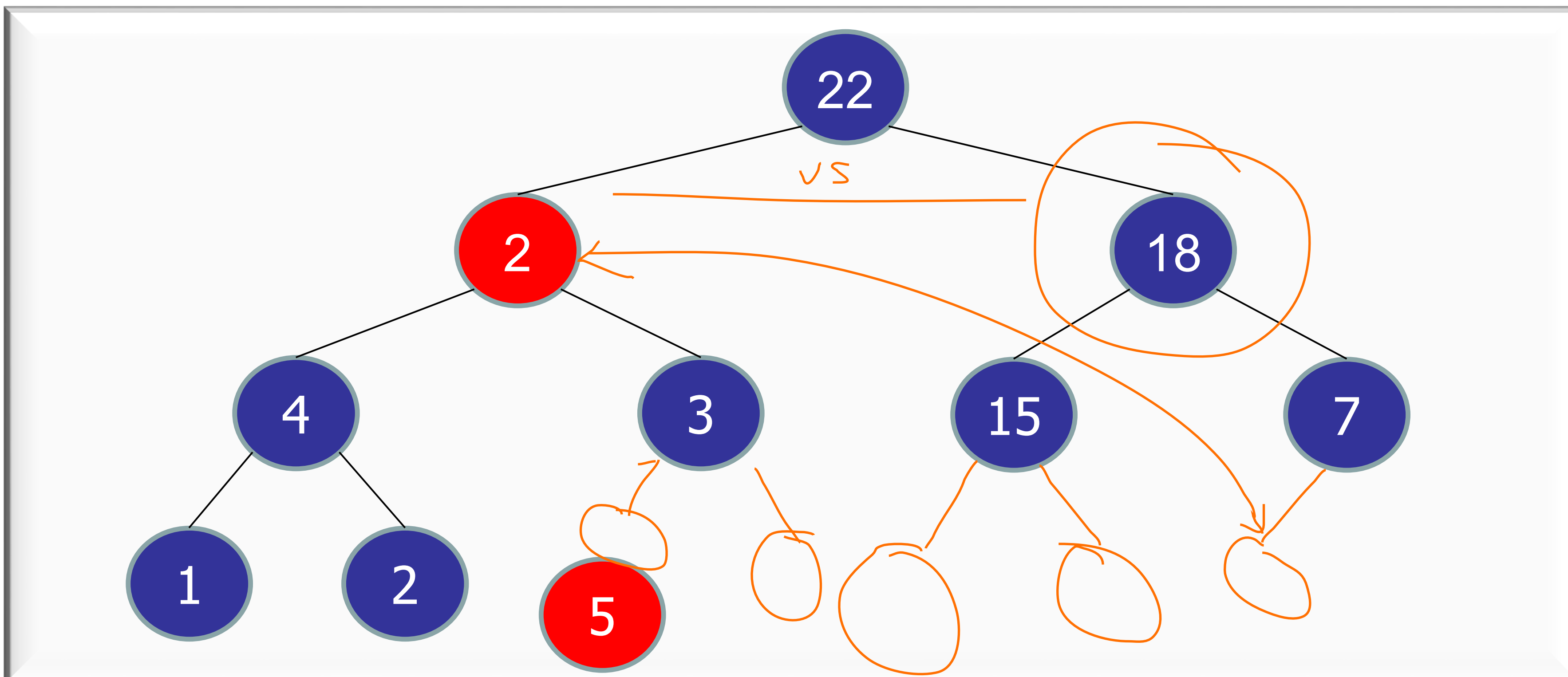
- `swap(5, last())`
- `remove(last())`



Inserting in a Heap

`delete(5)` :

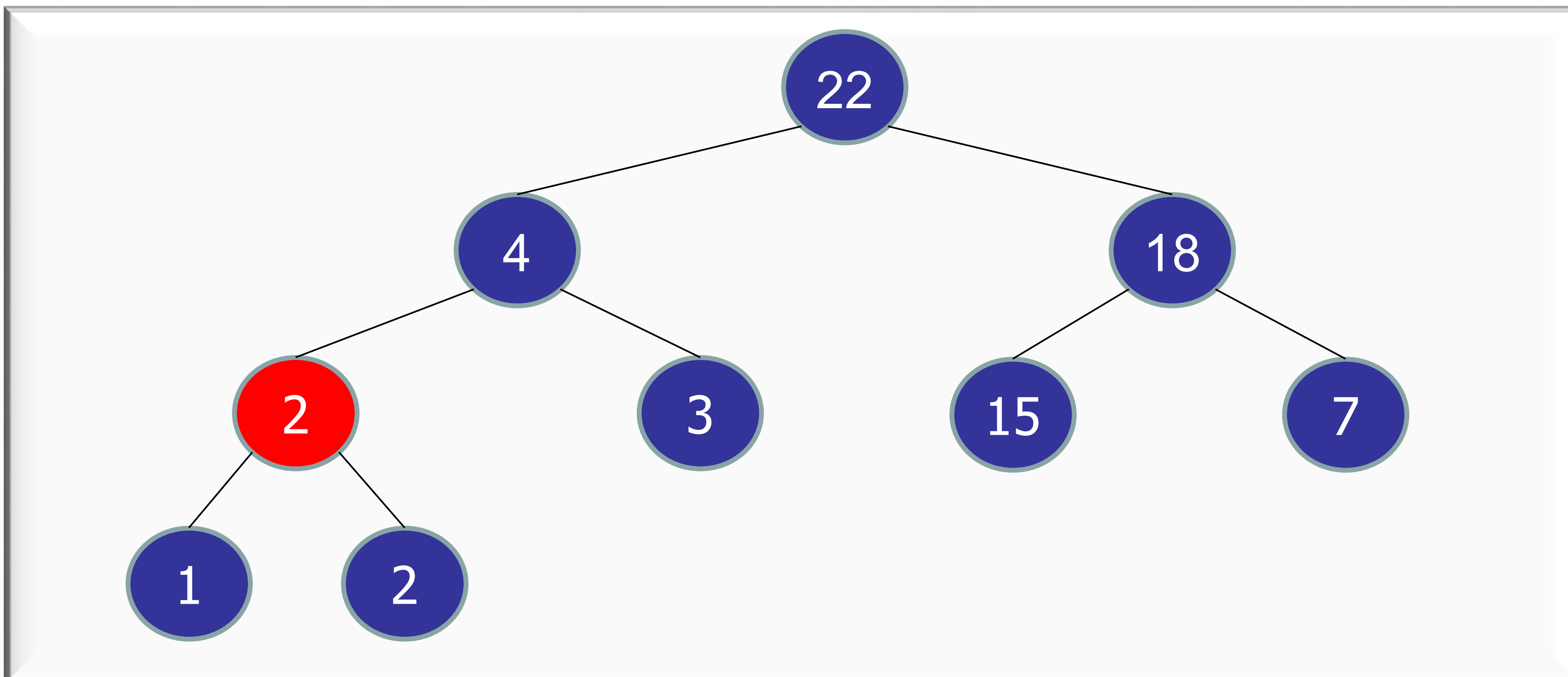
- `swap(5, last())`
- `remove(last())`
- `bubbleDown(2) // depending on if last() > deleted`



Inserting in a Heap

`delete(5)` :

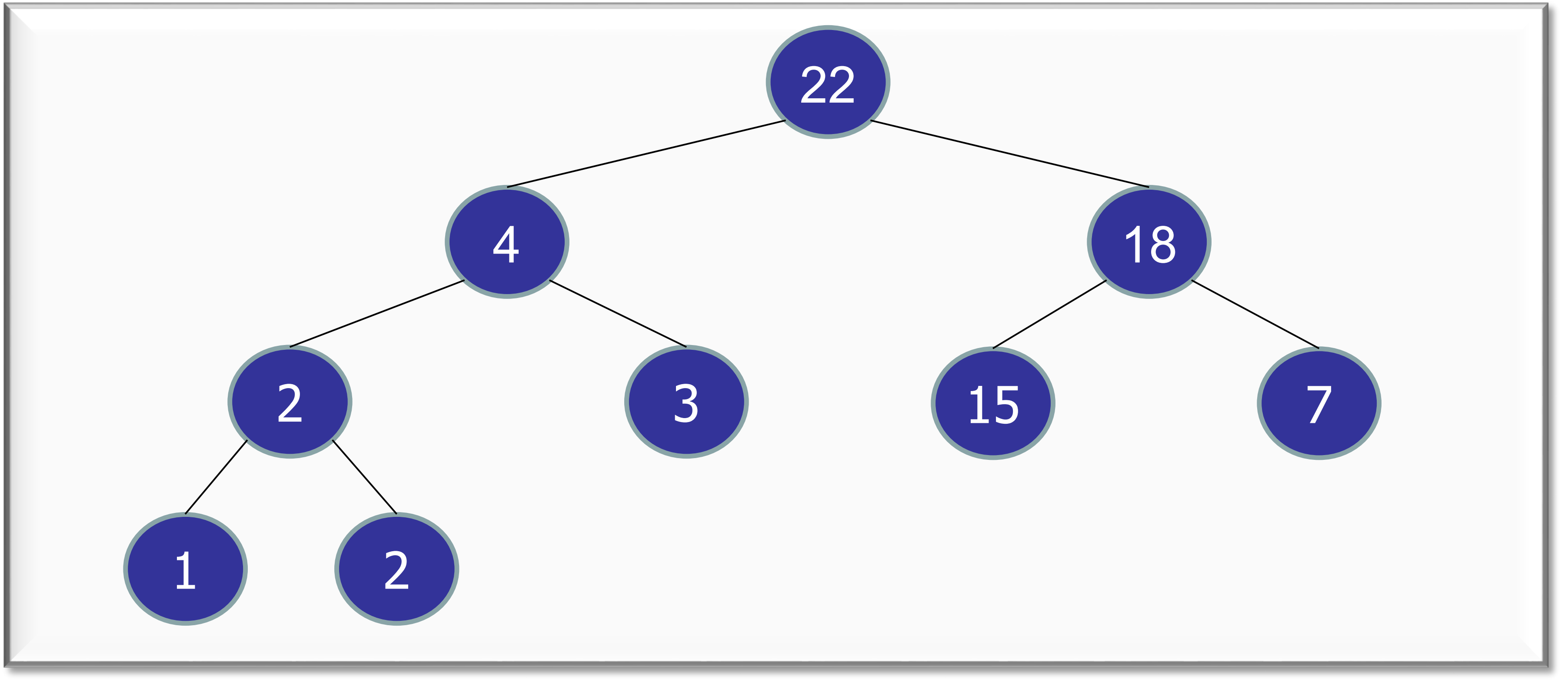
- `swap(5, last())`
- `remove(last())`
- `bubbleDown(2)`



Inserting in a Heap

`delete(5)` :

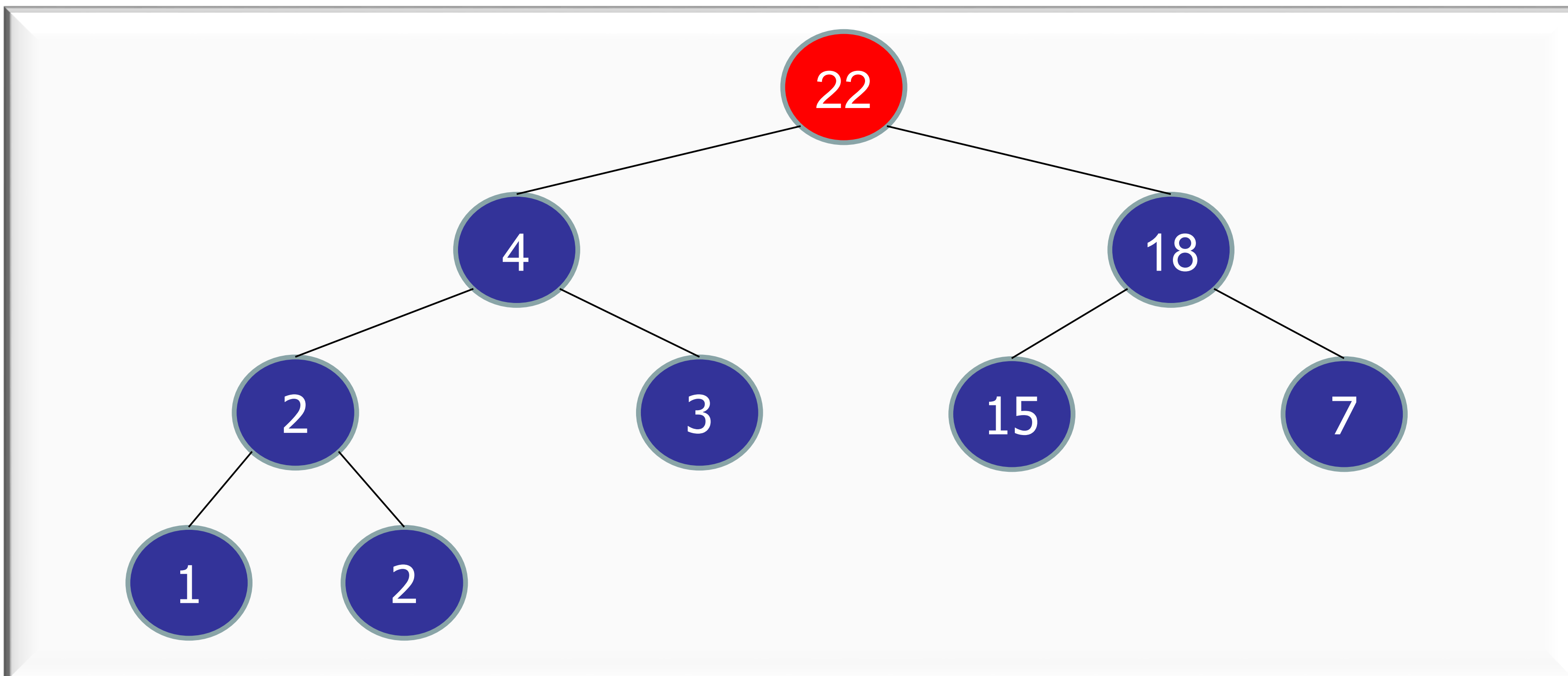
- `swap(5, last())`
- `remove(last())`
- `bubbleDown(2)`



Inserting in a Heap

`extractMax()` :

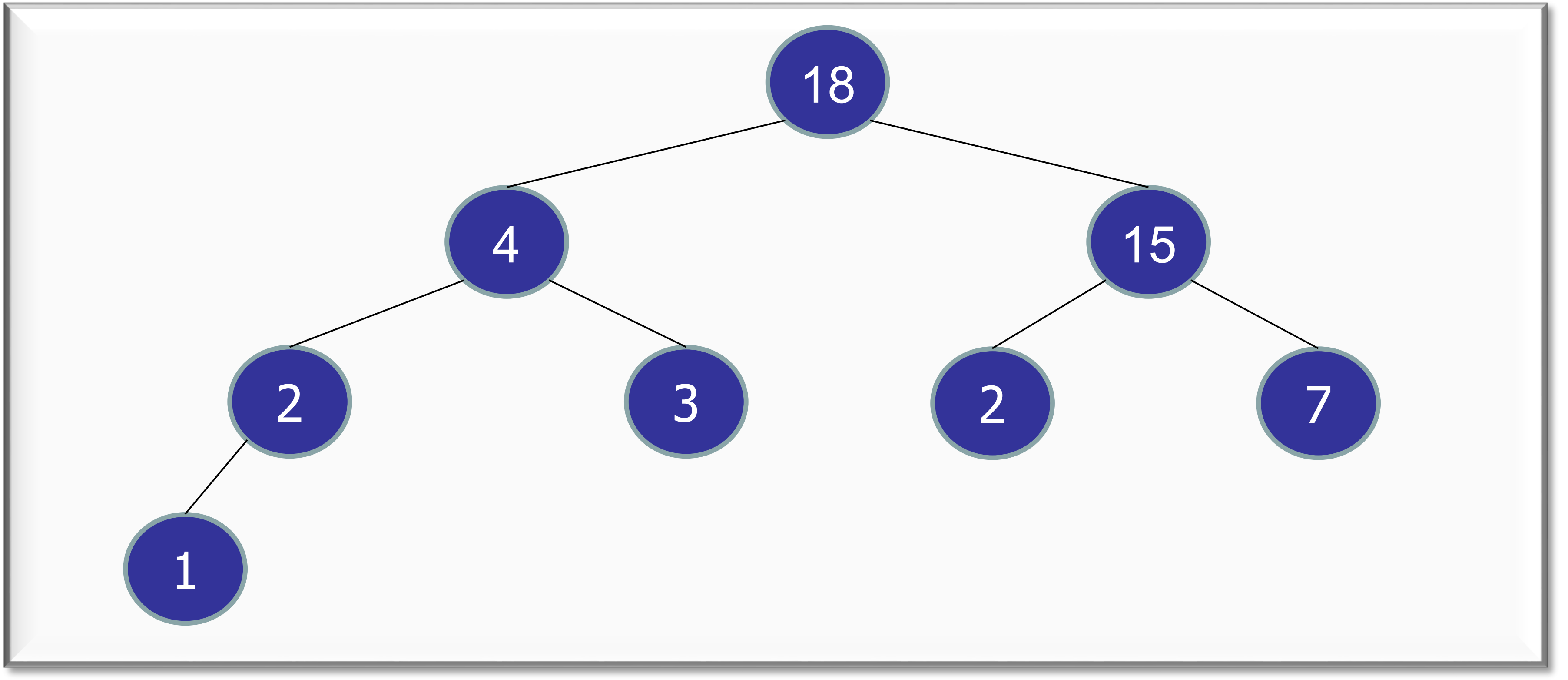
- `Node v = root;`
- `delete(root);`



Inserting in a Heap

`extractMax()` :

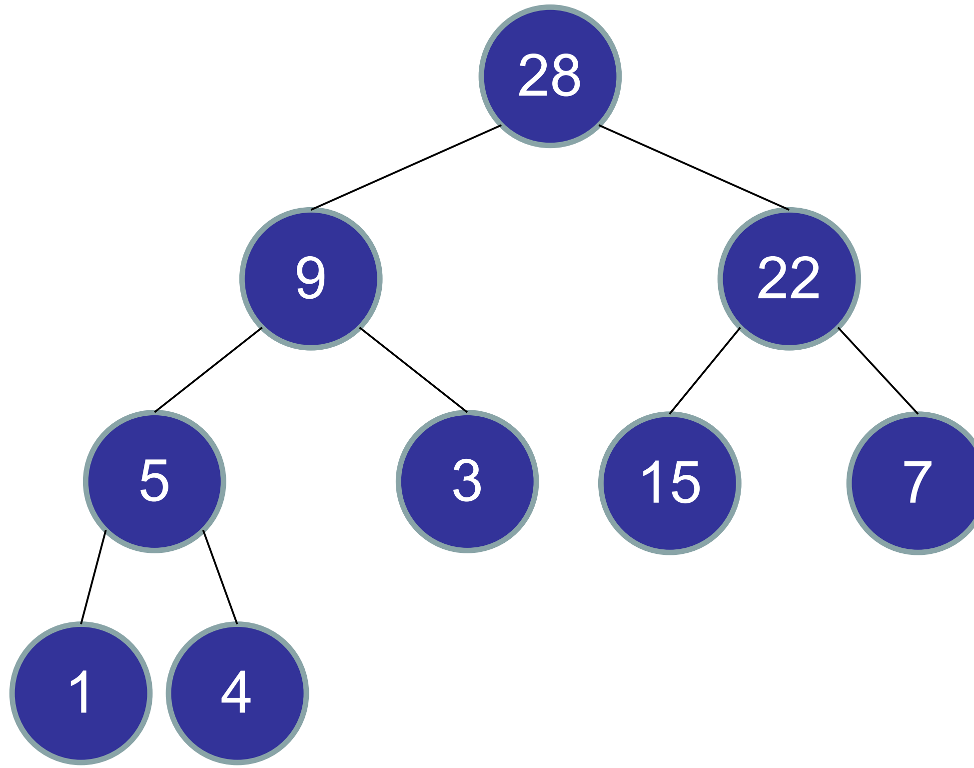
- `Node v = root;`
- `delete(root);`



(Max) Priority Queue

Heap Operations: O(log n)

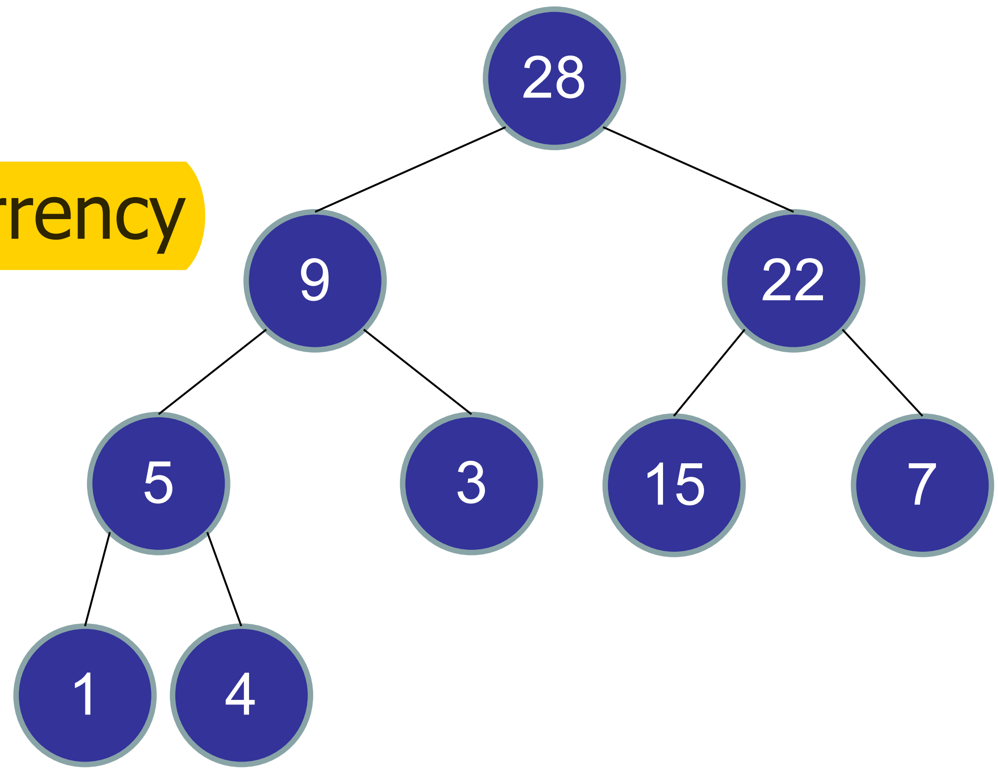
- insert
- extractMax
- increaseKey
- decreaseKey
- delete



(Max) Priority Queue

Heap vs. AVL Tree

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



How to store a tree?

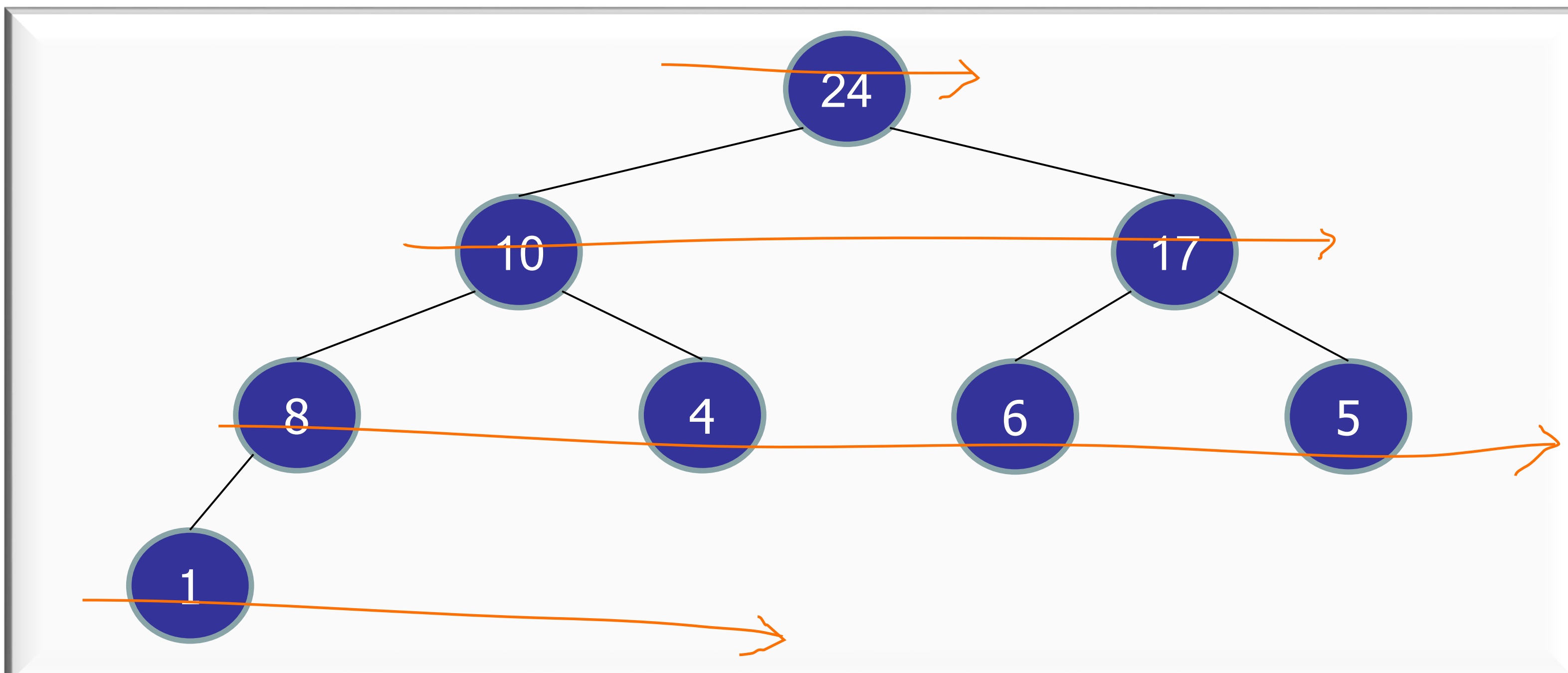
- Store in an array!

Store Tree in an Array

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

Level Order Traversal

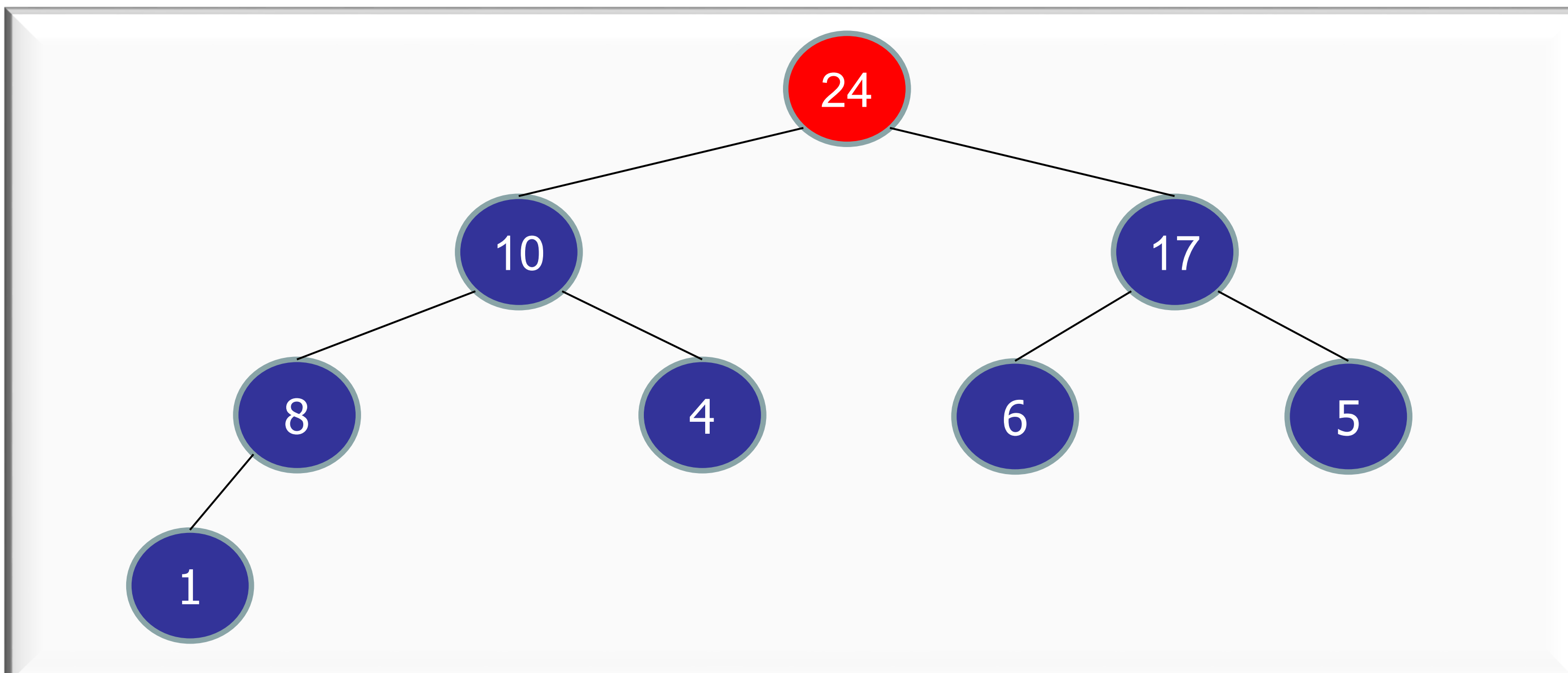
array slot	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
priority	**24**	**10**	**17**	**8**	**4**	**6**	**7**	**1**	



Store Tree in an Array

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

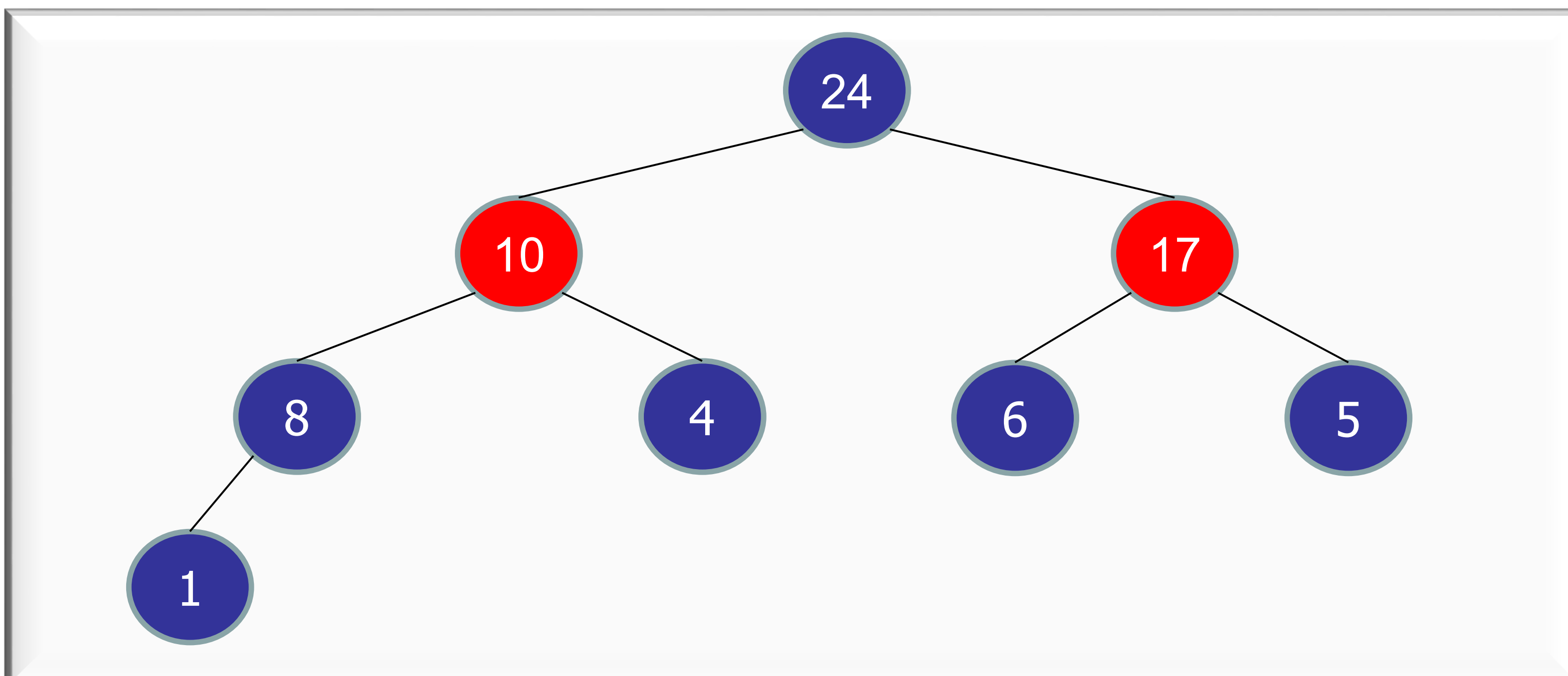
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	7	1	



Store Tree in an Array

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

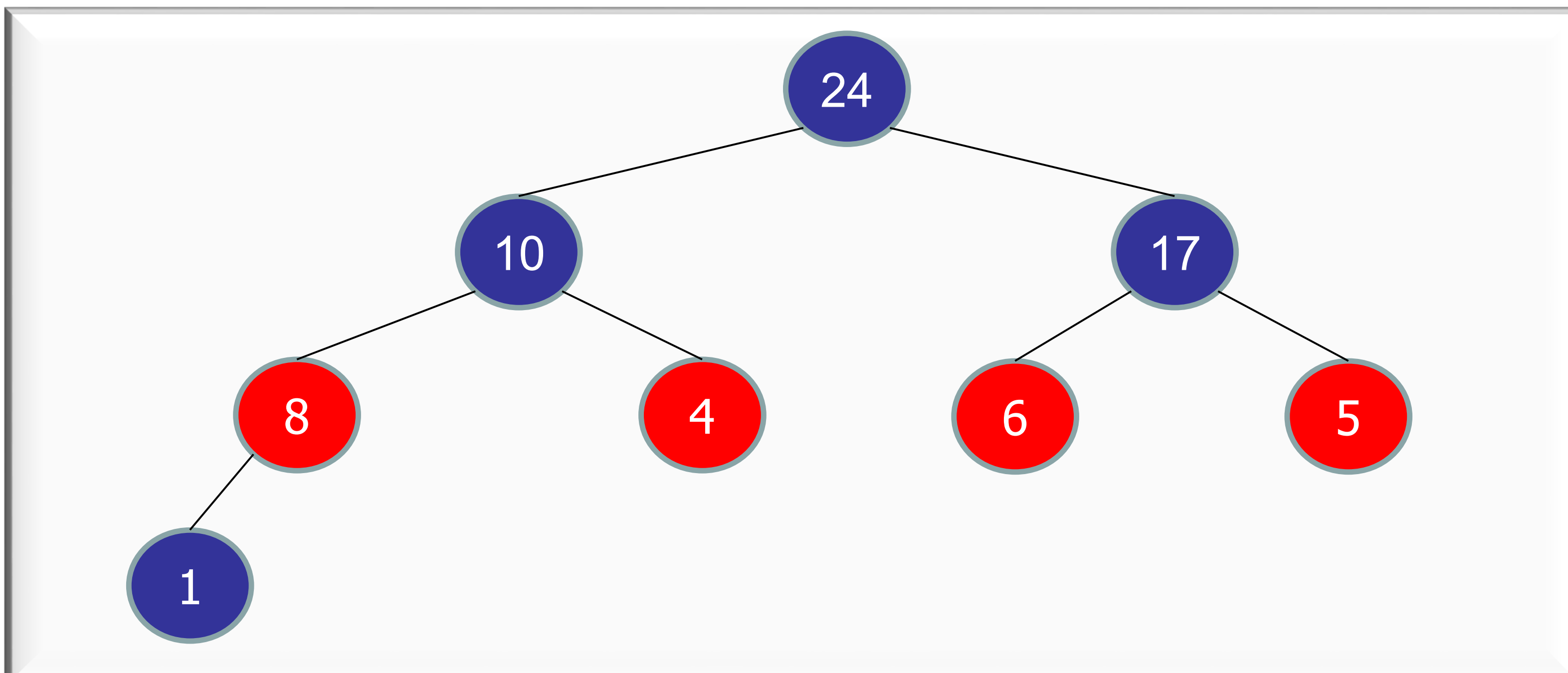
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	7	1	



Store Tree in an Array

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

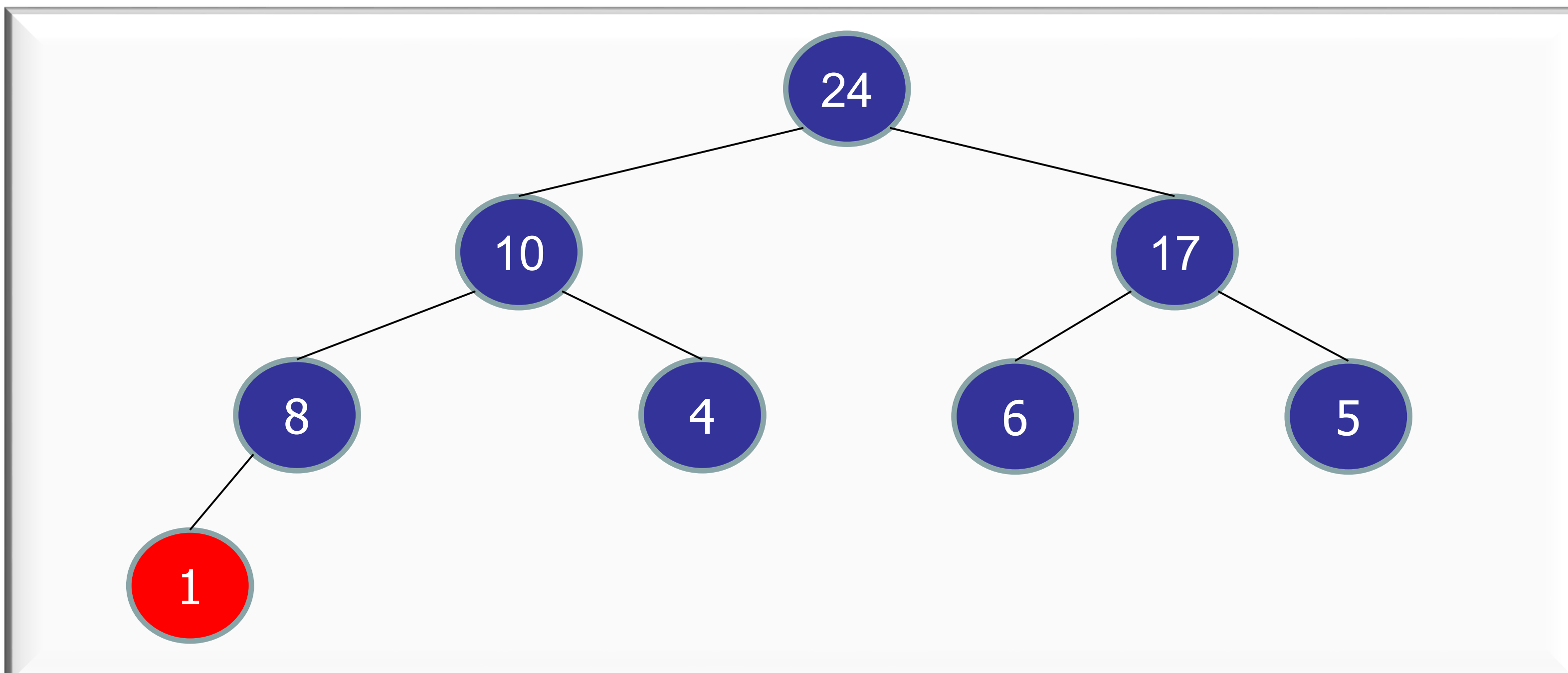
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	



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	$\boxed{1}$	

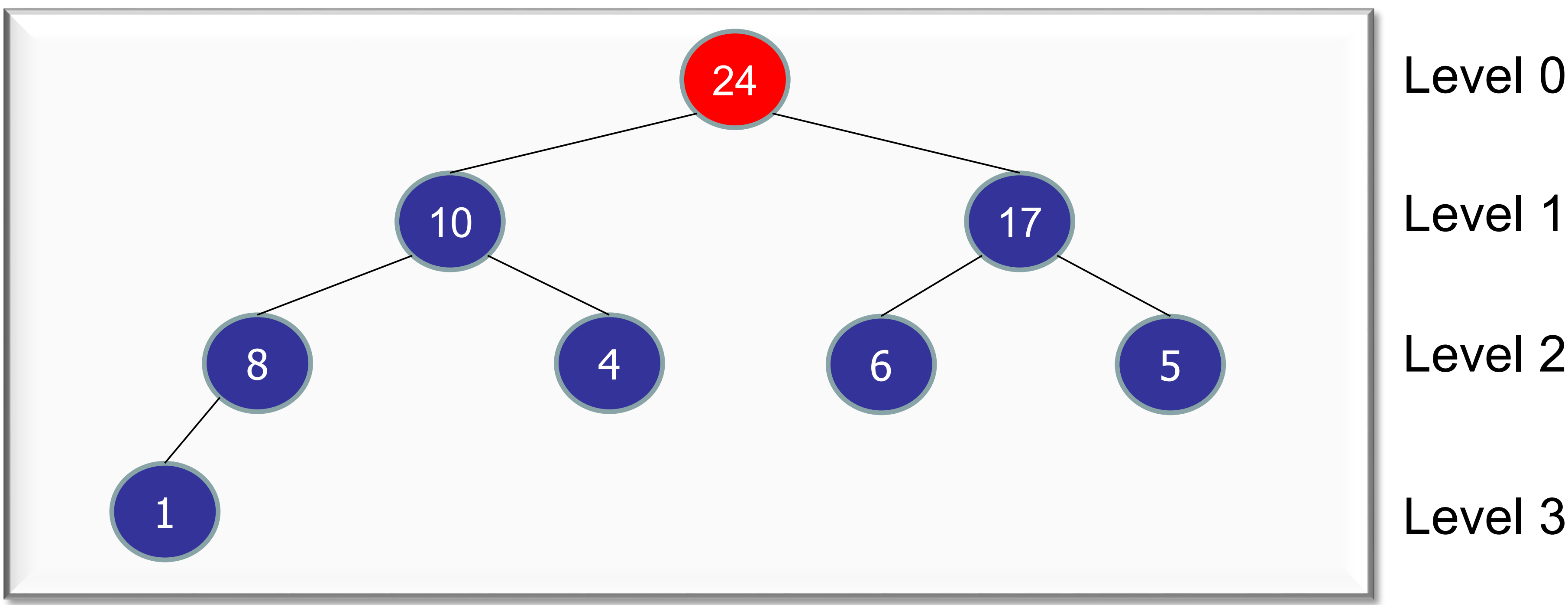


Store Tree in an Array

Each level i starts from the array index 2^i-1

Assuming the root is level 0 from top

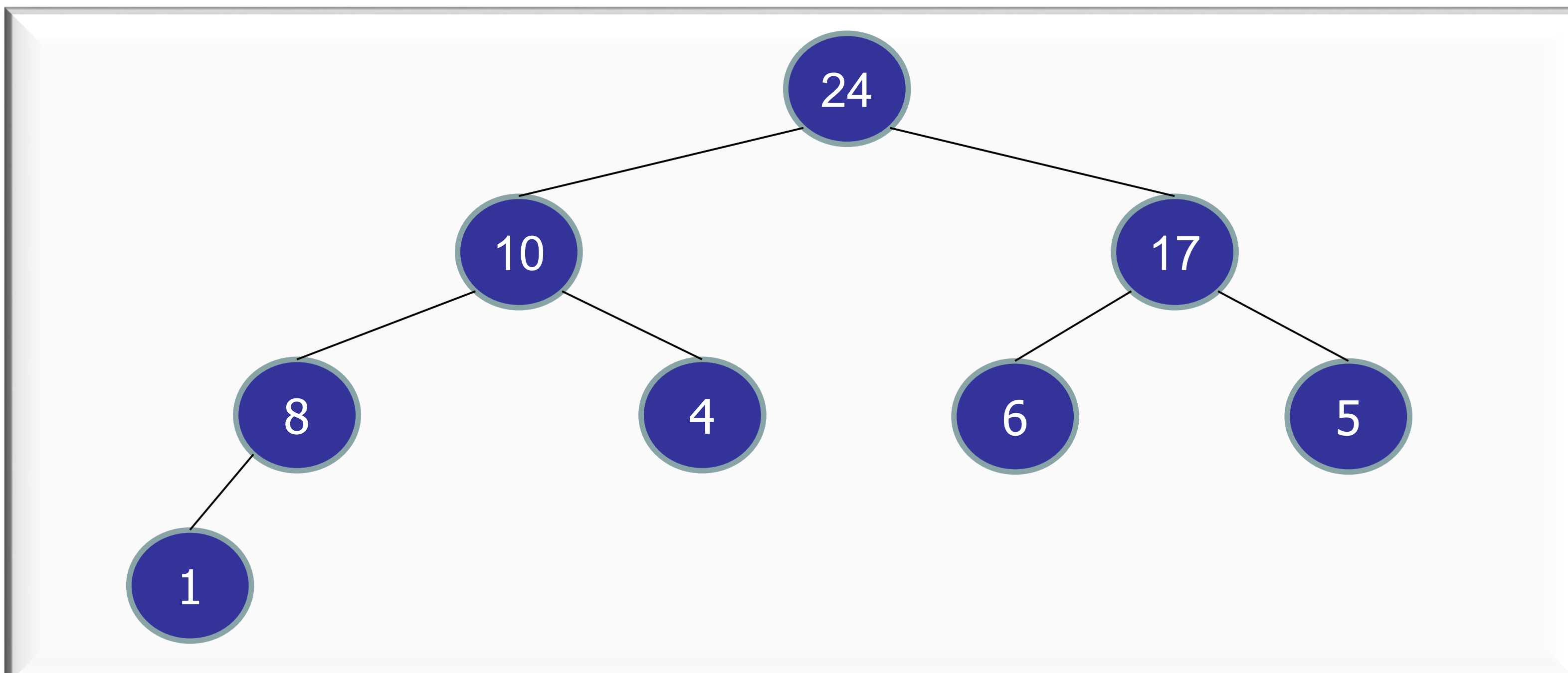
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	7	1	



Store Tree in an Array

`insert(15) :`

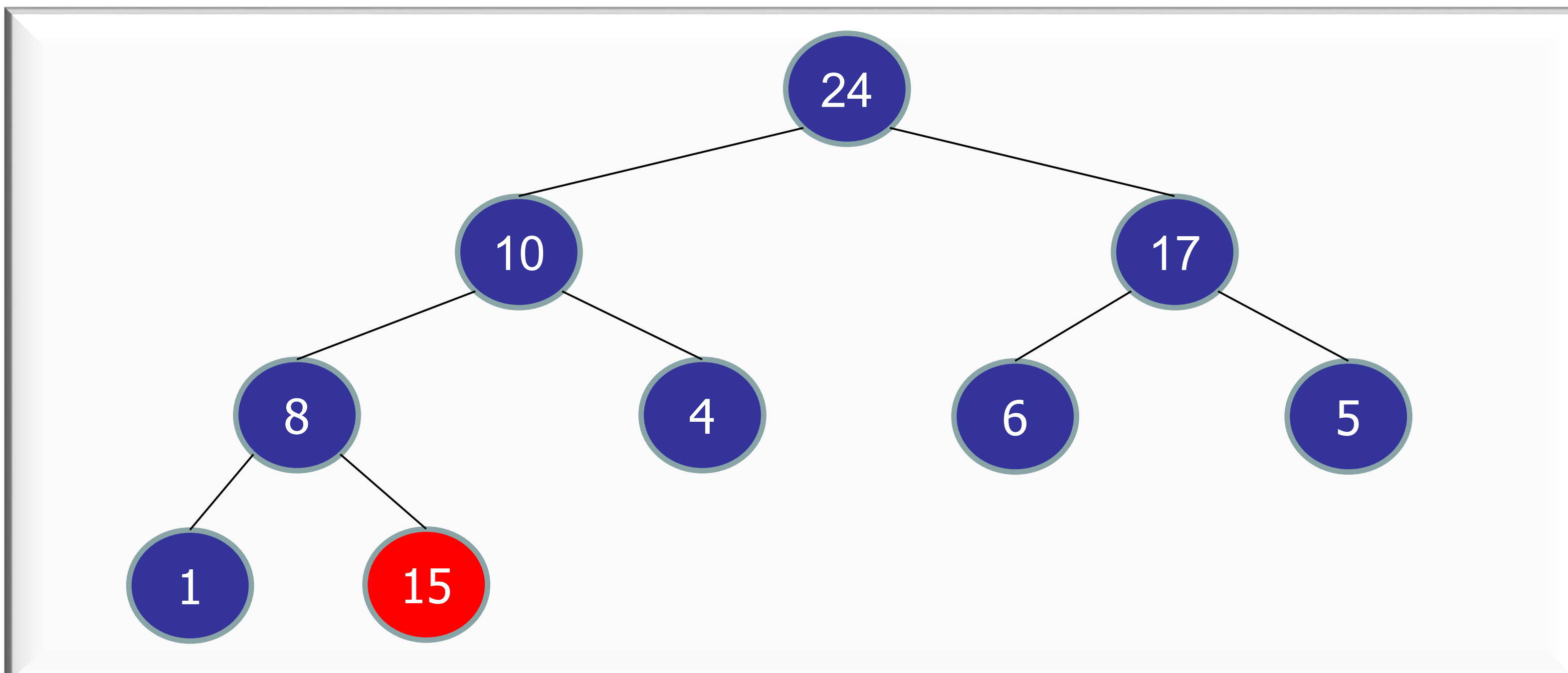
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	



Store Tree in an Array

`insert(15) :`

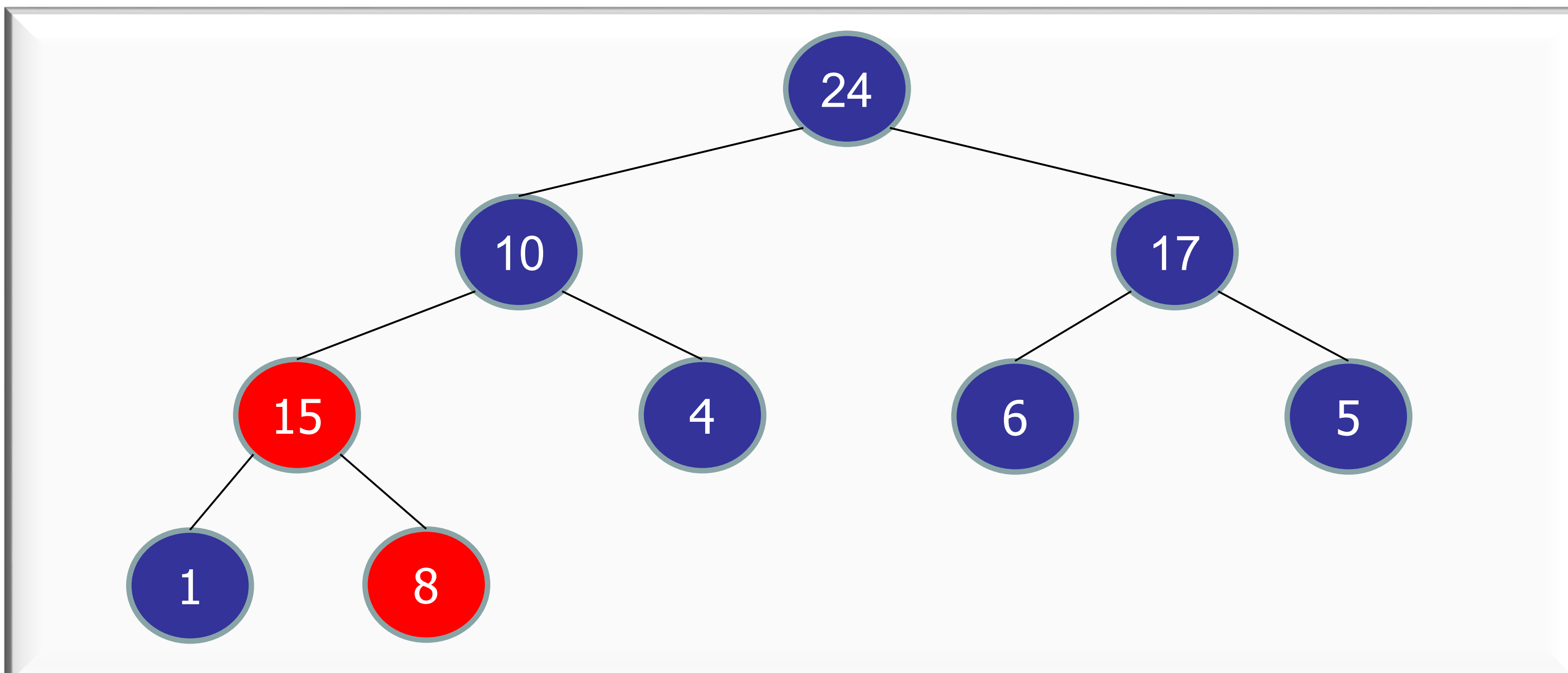
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	15



Store Tree in an Array

```
insert(15) :
```

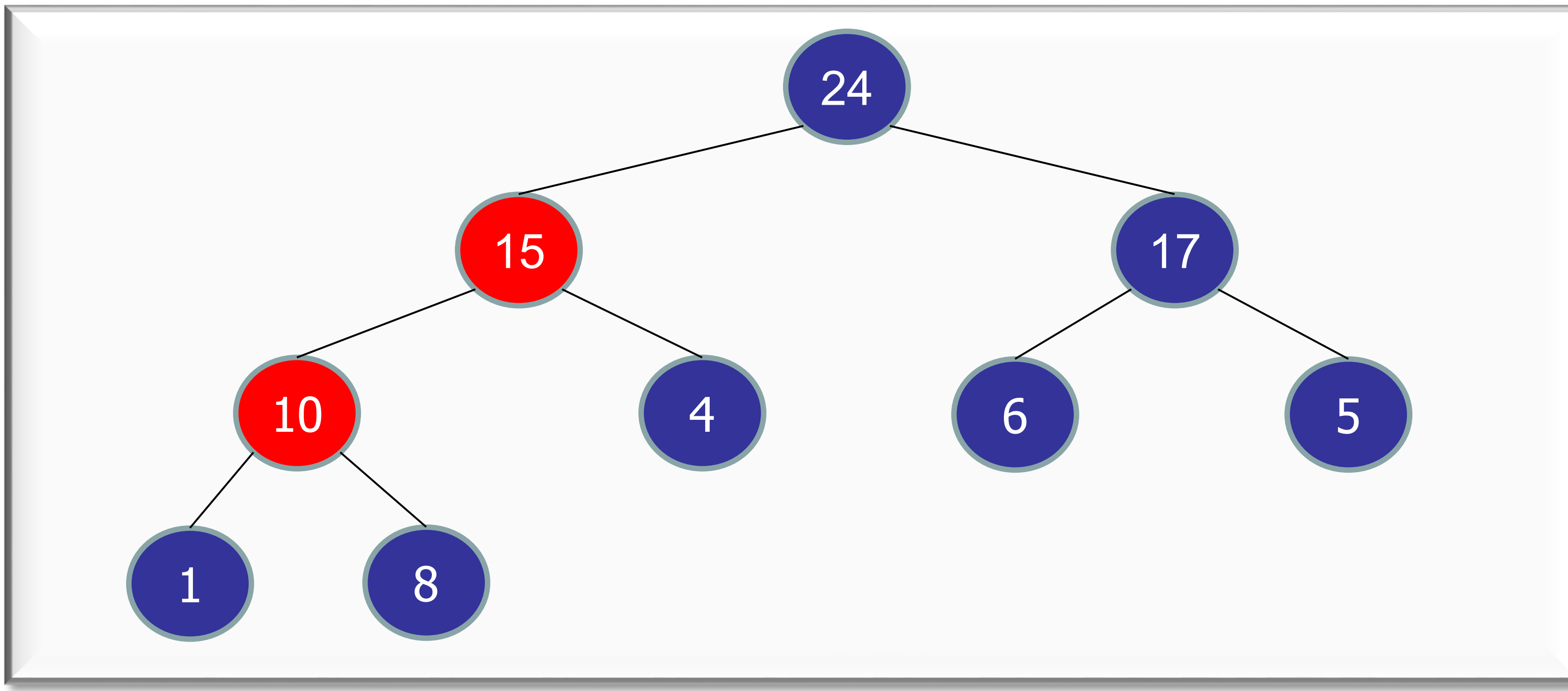
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	15	4	6	5	1	8



Store Tree in an Array

`insert(15) :`

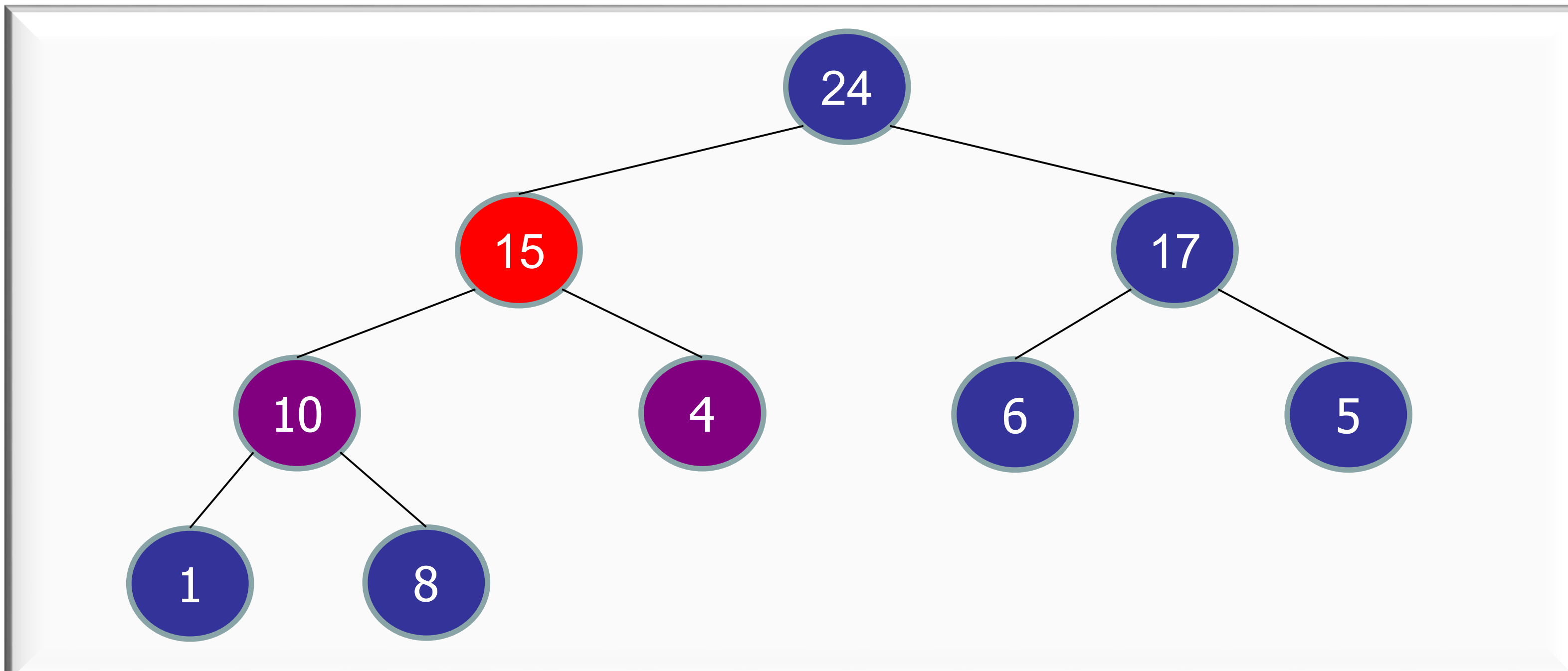
array slot	0	1	2	3	4	5	6	7	8
priority	24	15	17	10	4	6	5	1	8



Store Tree in an Array

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

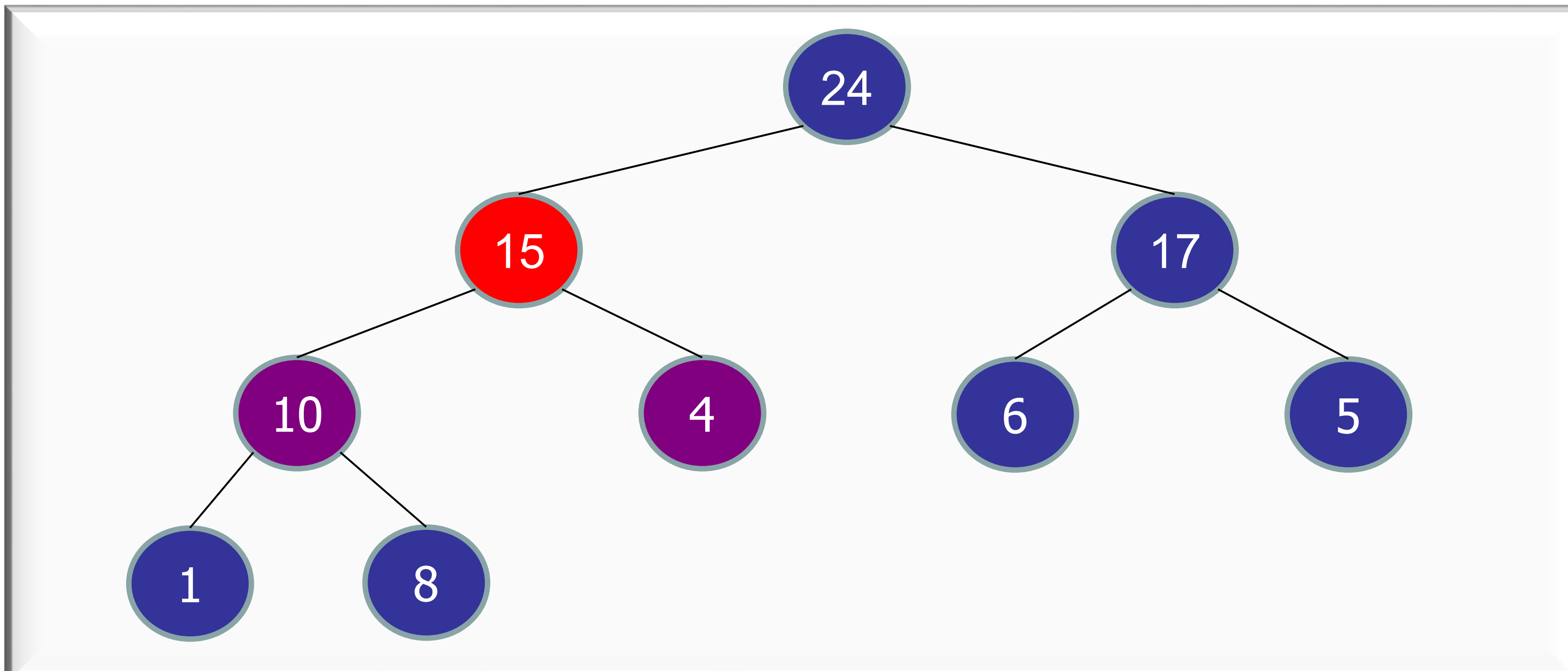
array slot	0	1	2	3	4	5	6	7	8
priority	24	15	17	10	4	6	5	1	8



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**

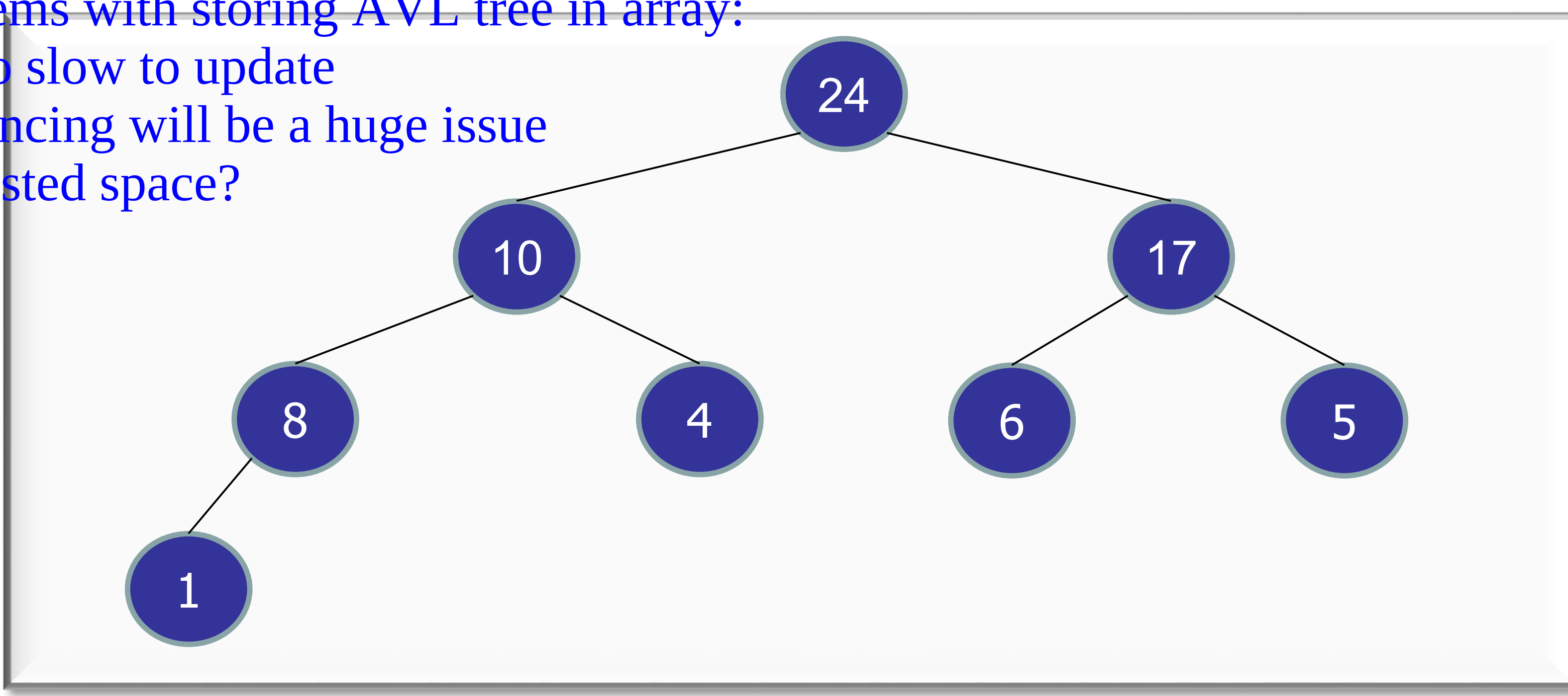


Store AVL Tree in an Array

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	

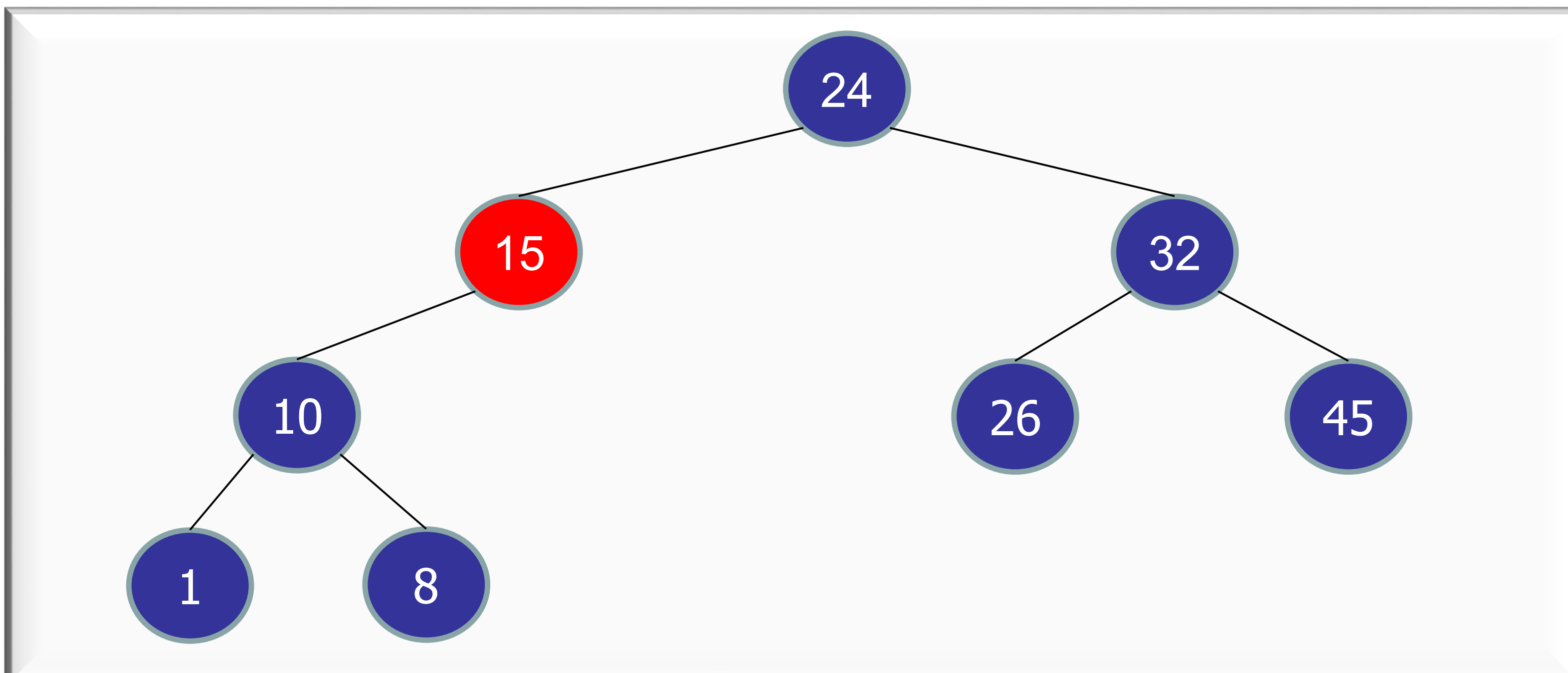
Problems with storing AVL tree in array:
1) Too slow to update
- Balancing will be a huge issue
2) Wasted space?



Store AVL Tree in an Array

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

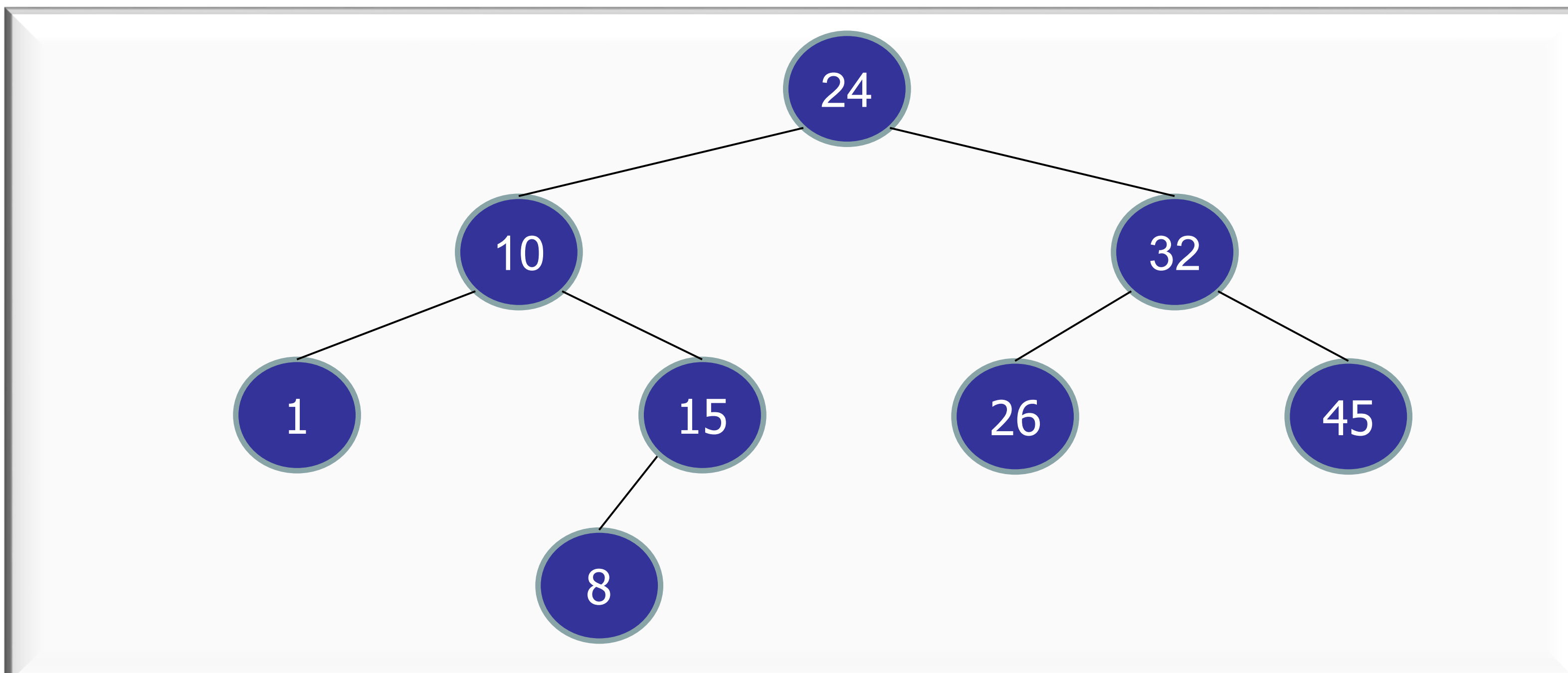
array slot	0	1	2	3	4	5	6	7	8
priority	24	15	32	10		26	45	1	8



Store AVL 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	



Let’s Sort things with heaps also!

- Heap sort!

Examples

- Bitter + Sweet = Bittersweet
- Living + Death = Living Death
- Beautiful + Tyrant = Beautiful Tyrant!
- Minor + Crisis = Minor Crisis
- Jumbo + Shrimp = Jumbo Shrimp
- Clearly + Confused = Clearly Confused
- Only + Choice = Only Choice
- Larger + Half = Larger Half
- Freezer + Burn = Freezer Burn
- Pretty + Ugly = Pretty Ugly

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

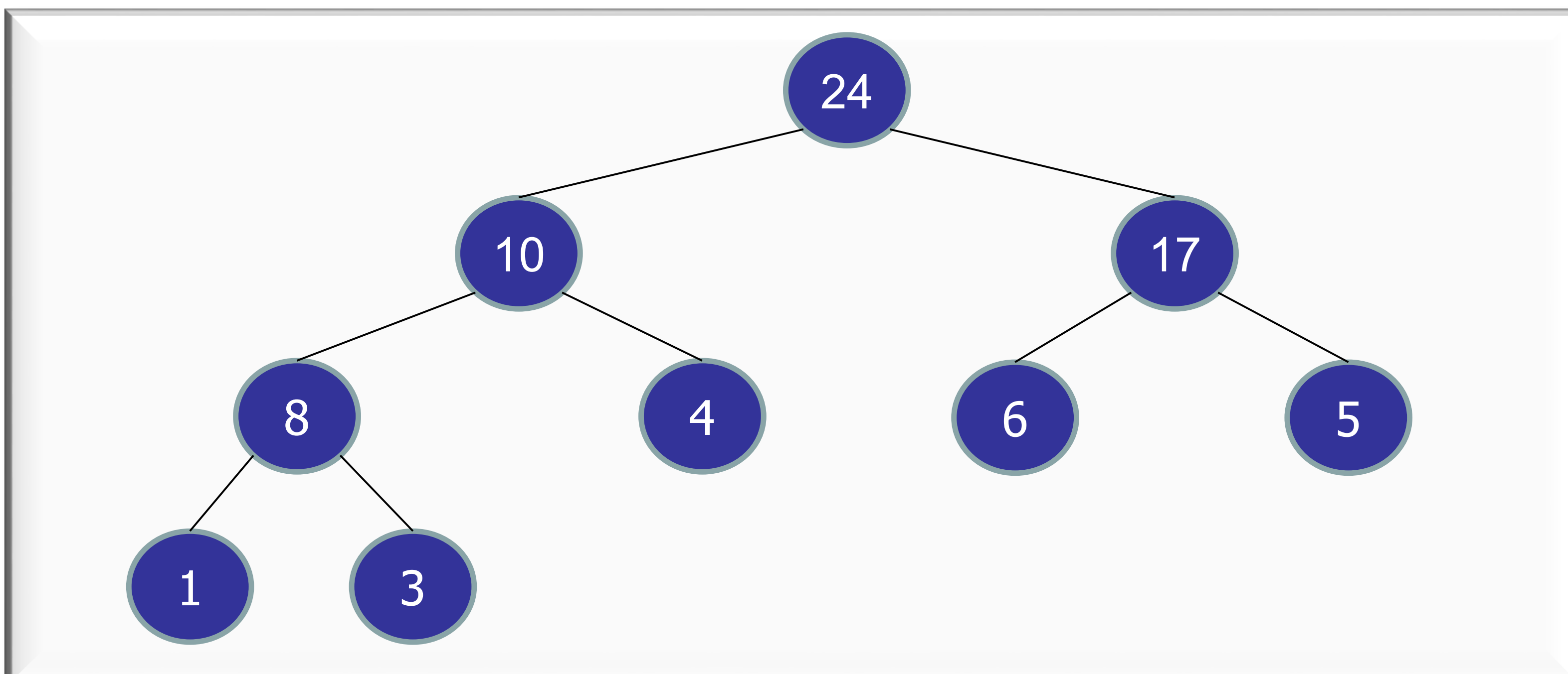
Heap → Sorted list:

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:

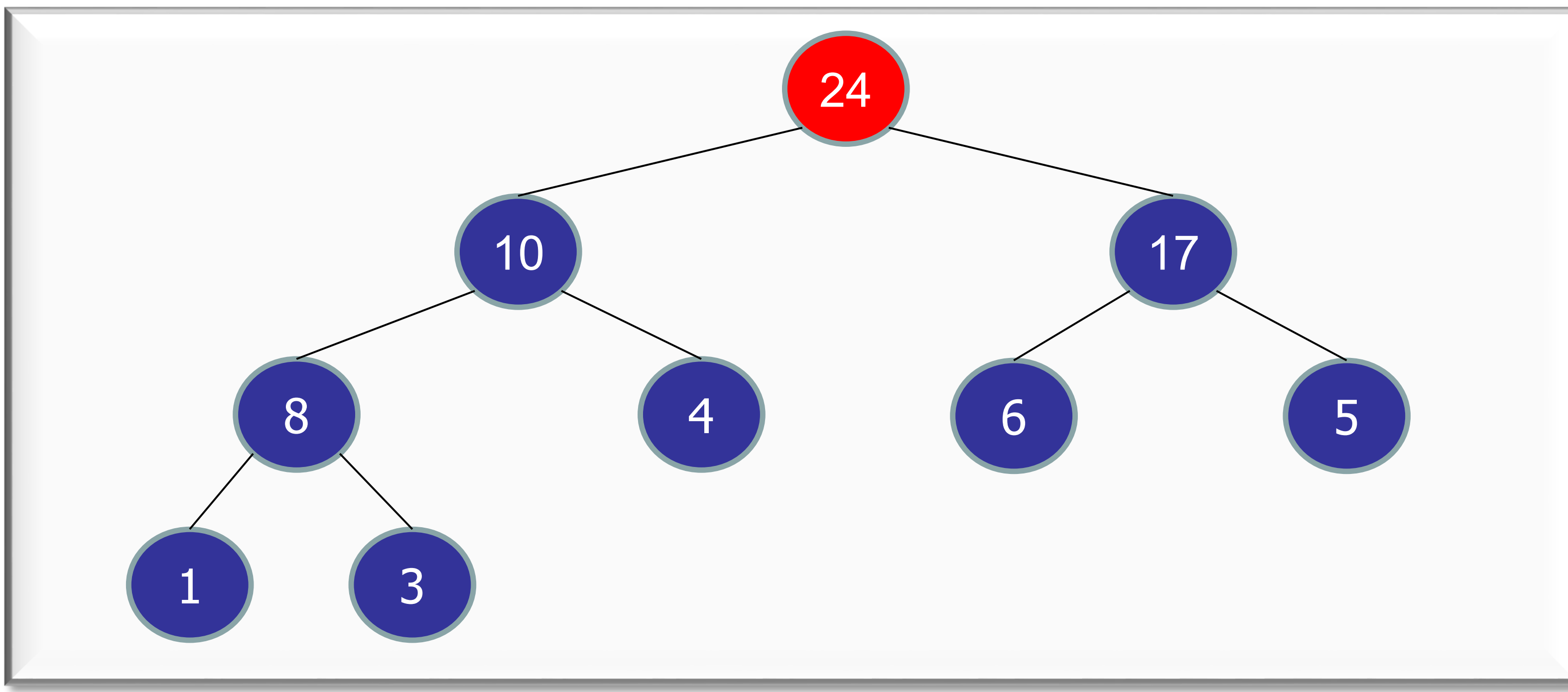
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	3



HeapSort

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

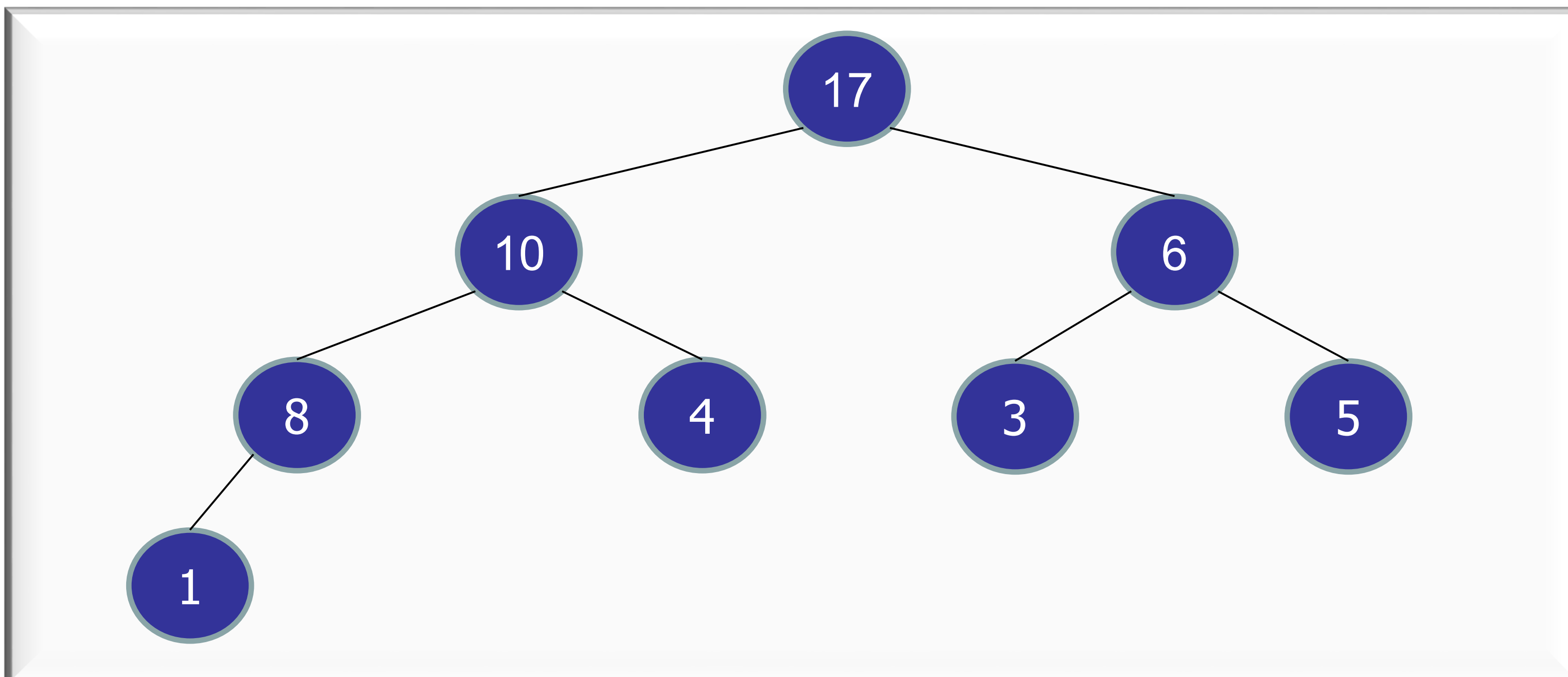
array slot	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
priority	**24**	**10**	**17**	**8**	**4**	**6**	**5**	**1**	**3**



HeapSort

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

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

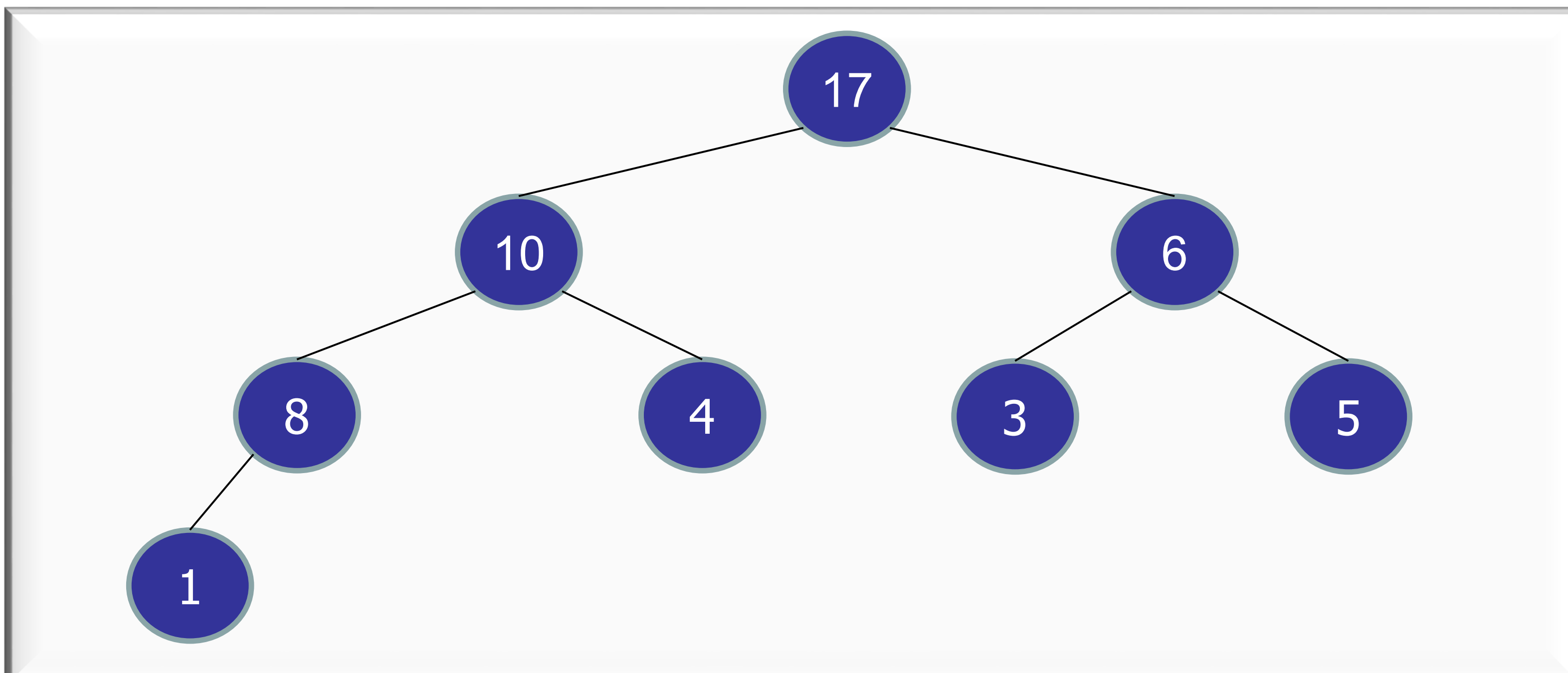


HeapSort

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

in place sorting

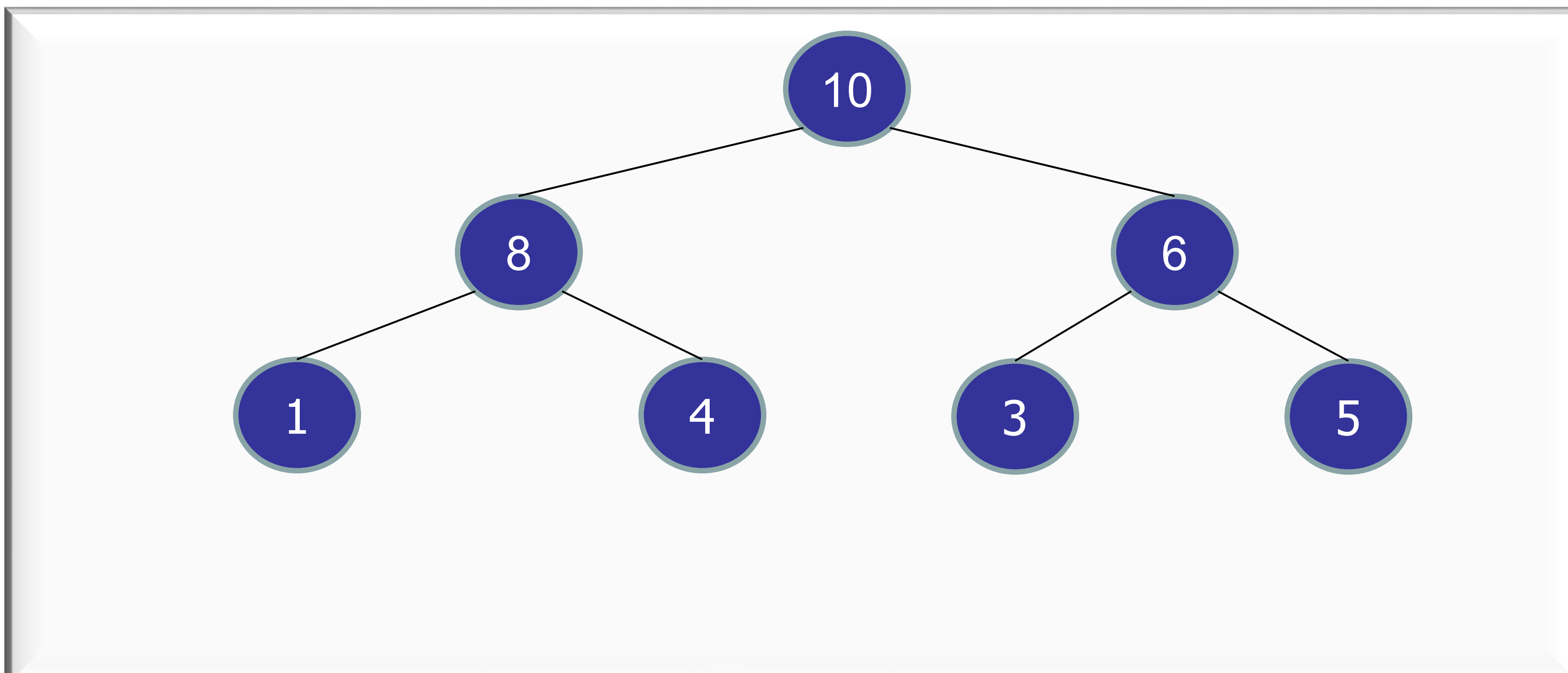
array slot	0	1	2	3	4	5	6	7	8
priority	17	10	6	8	4	3	5	1	24



HeapSort

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

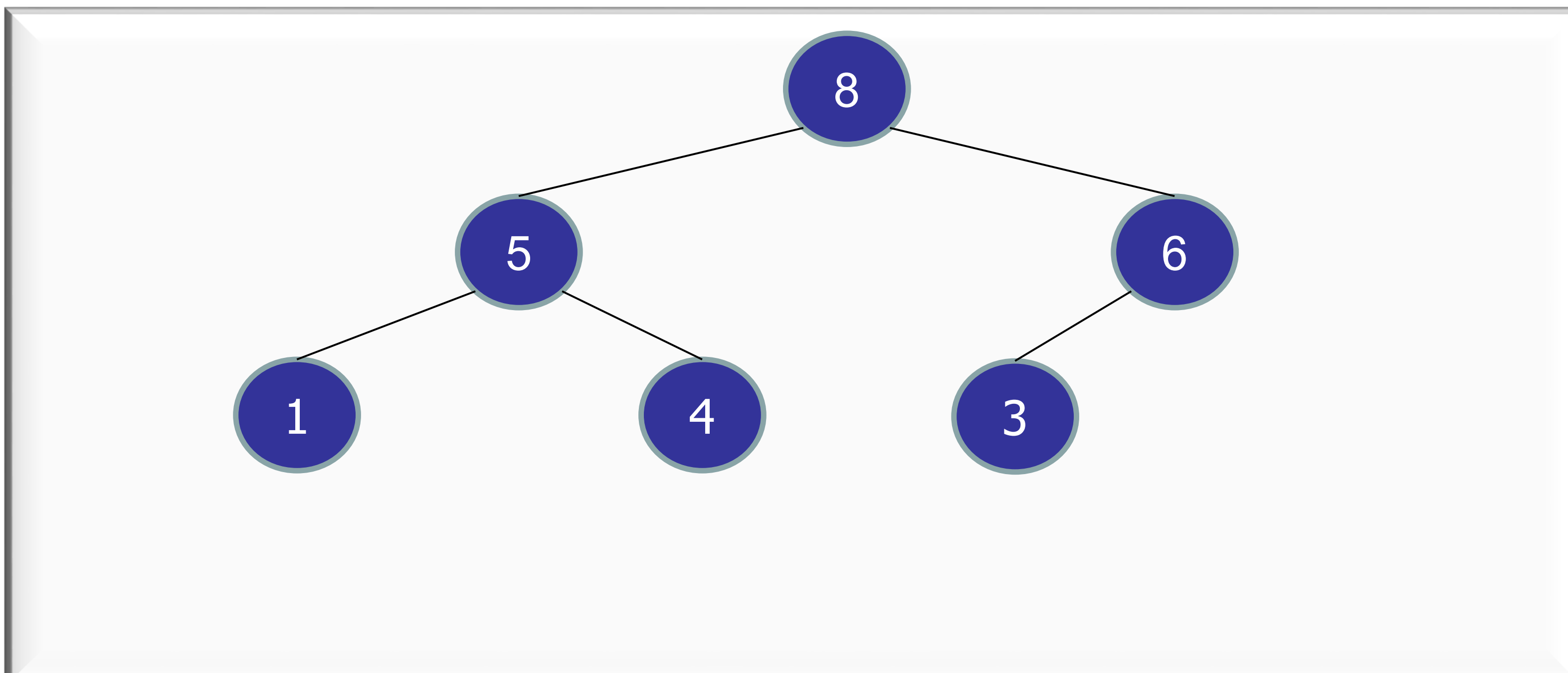
array slot	0	1	2	3	4	5	6	7	8
priority	10	8	6	1	4	3	5	17	24



HeapSort

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

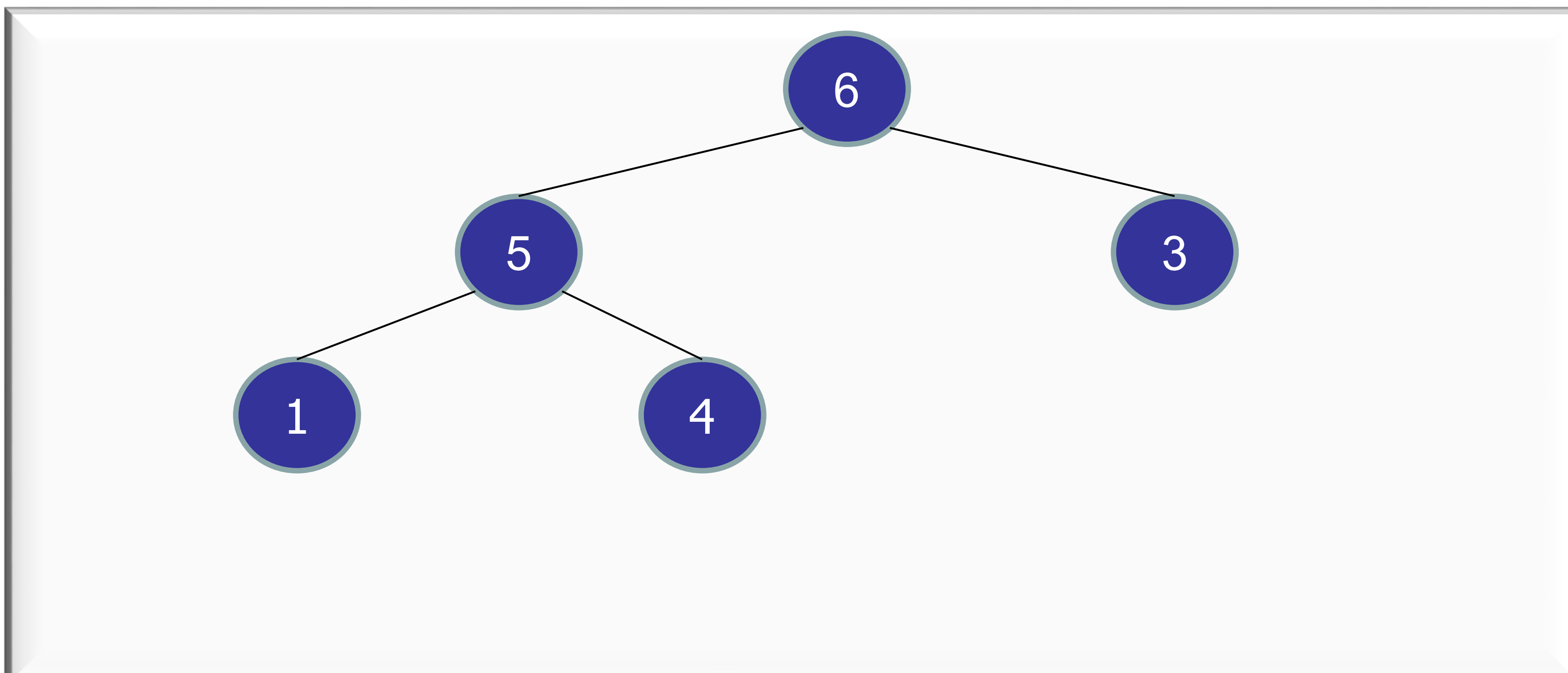
array slot	0	1	2	3	4	5	6	7	8
priority	8	5	6	1	4	3	10	17	24



HeapSort

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

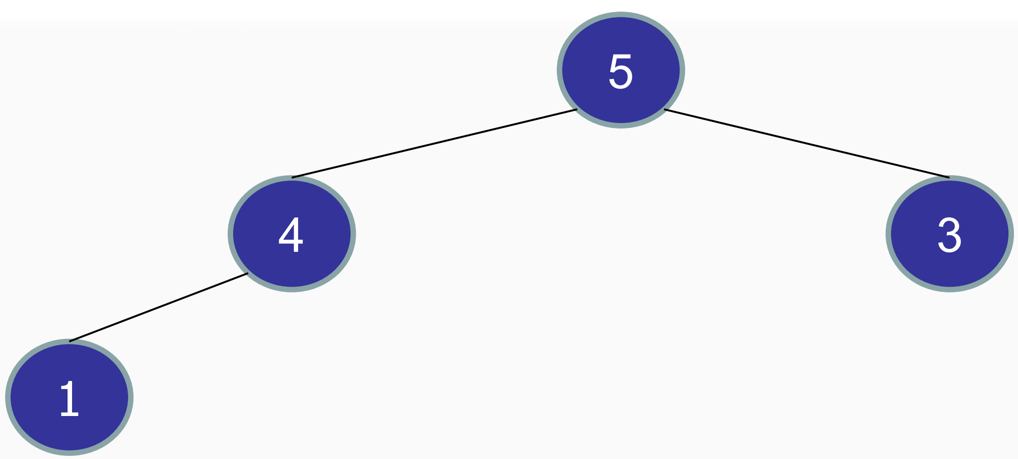
array slot	0	1	2	3	4	5	6	7	8
priority	6	5	3	1	4	8	10	17	24



HeapSort

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

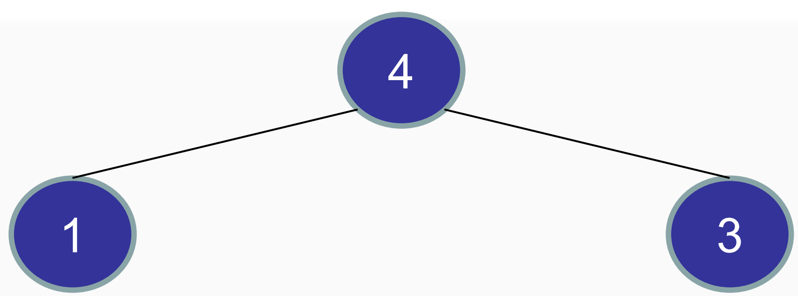
array slot	0	1	2	3	4	5	6	7	8
priority	5	4	3	1	6	8	10	17	24



HeapSort

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

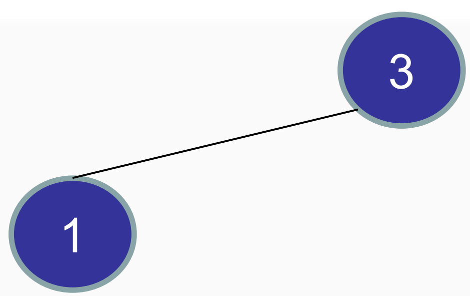
array slot	0	1	2	3	4	5	6	7	8
priority	4	1	3	5	6	8	10	17	24



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

∴ n log n

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

log(n)

O(n)

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

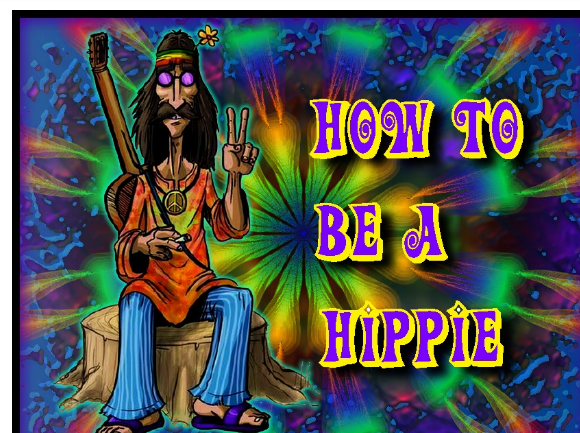
Unsorted list:

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

Heapify!



HeapSort

Heapify v.1: Unsorted list ➔ Heap

O(n log n)

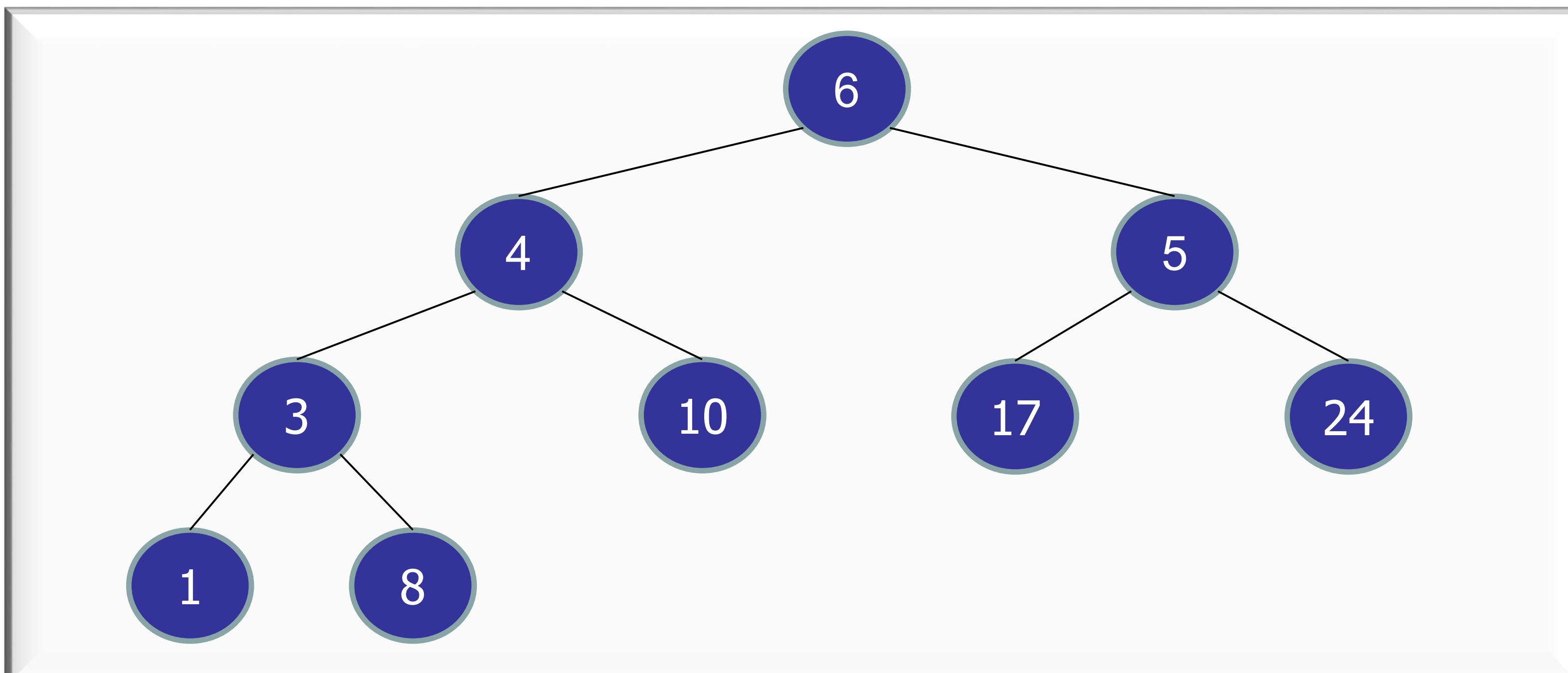
array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

```
// 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);}
```

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**

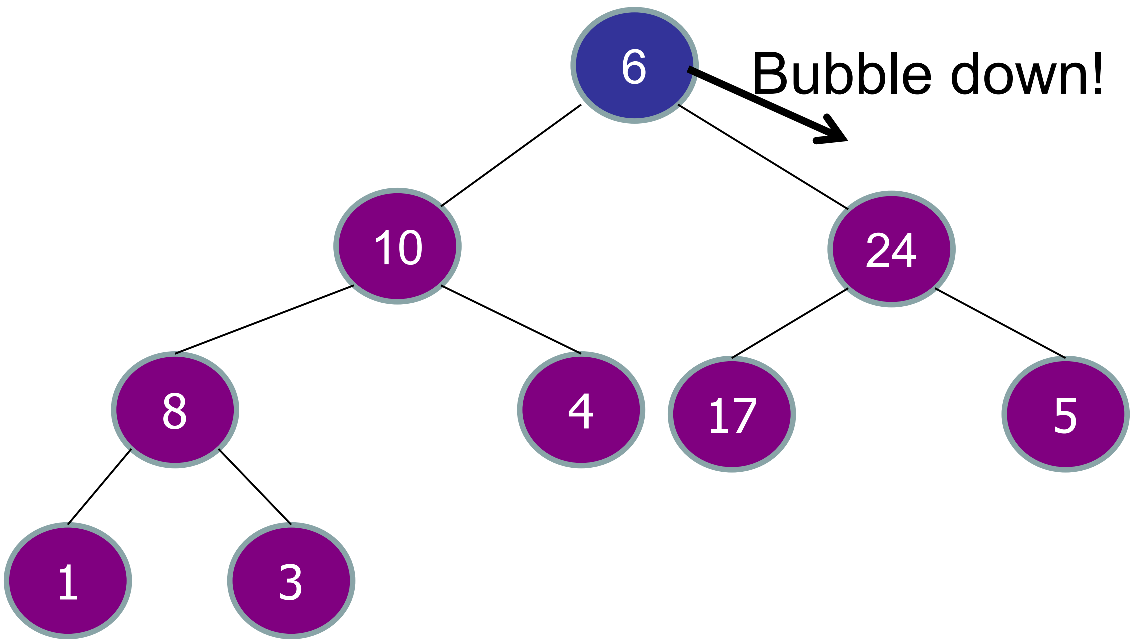


HeapSort

Heapify v.2: Unsorted list ➔ Heap

Idea: if you are given two heaps and one new node, how do you join all of them into <u>one single heap</u>?

- join them and bubble down the root

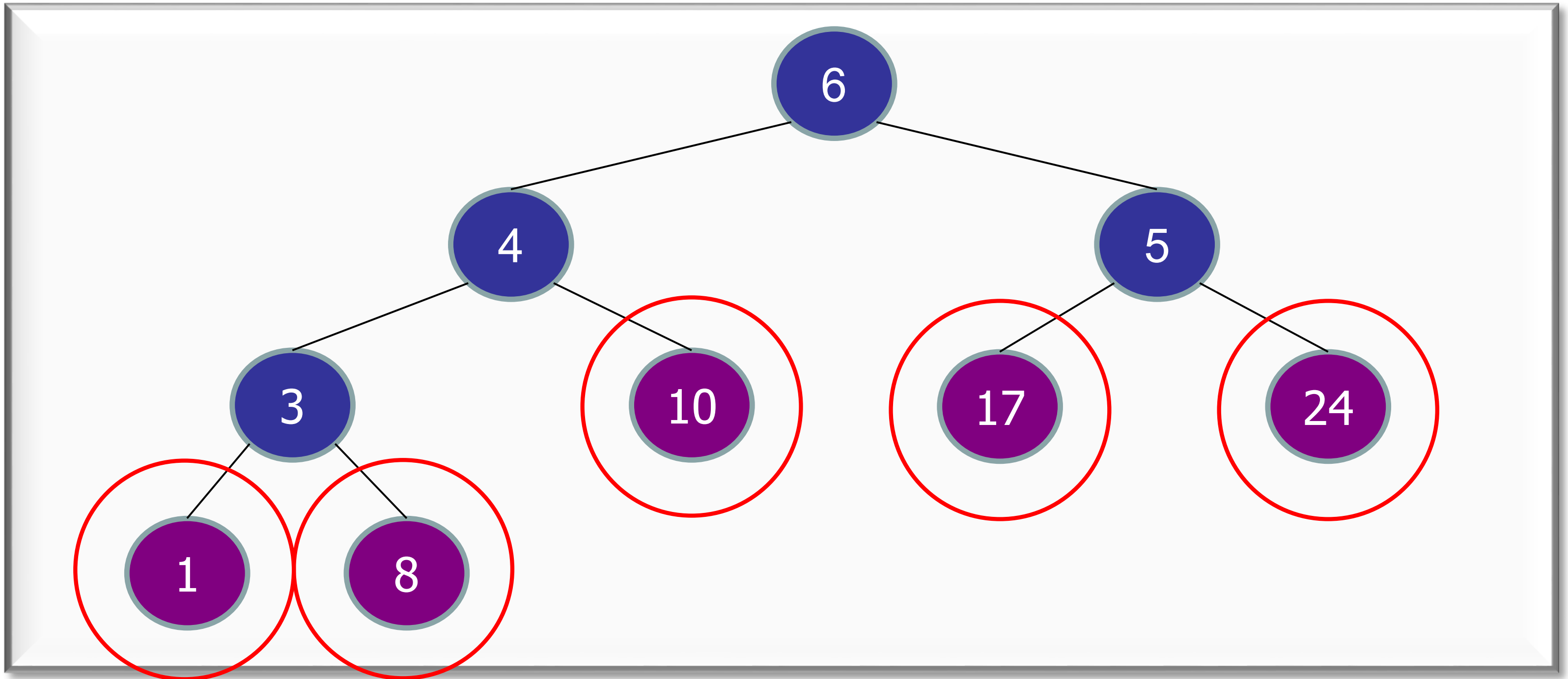


HeapSort

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

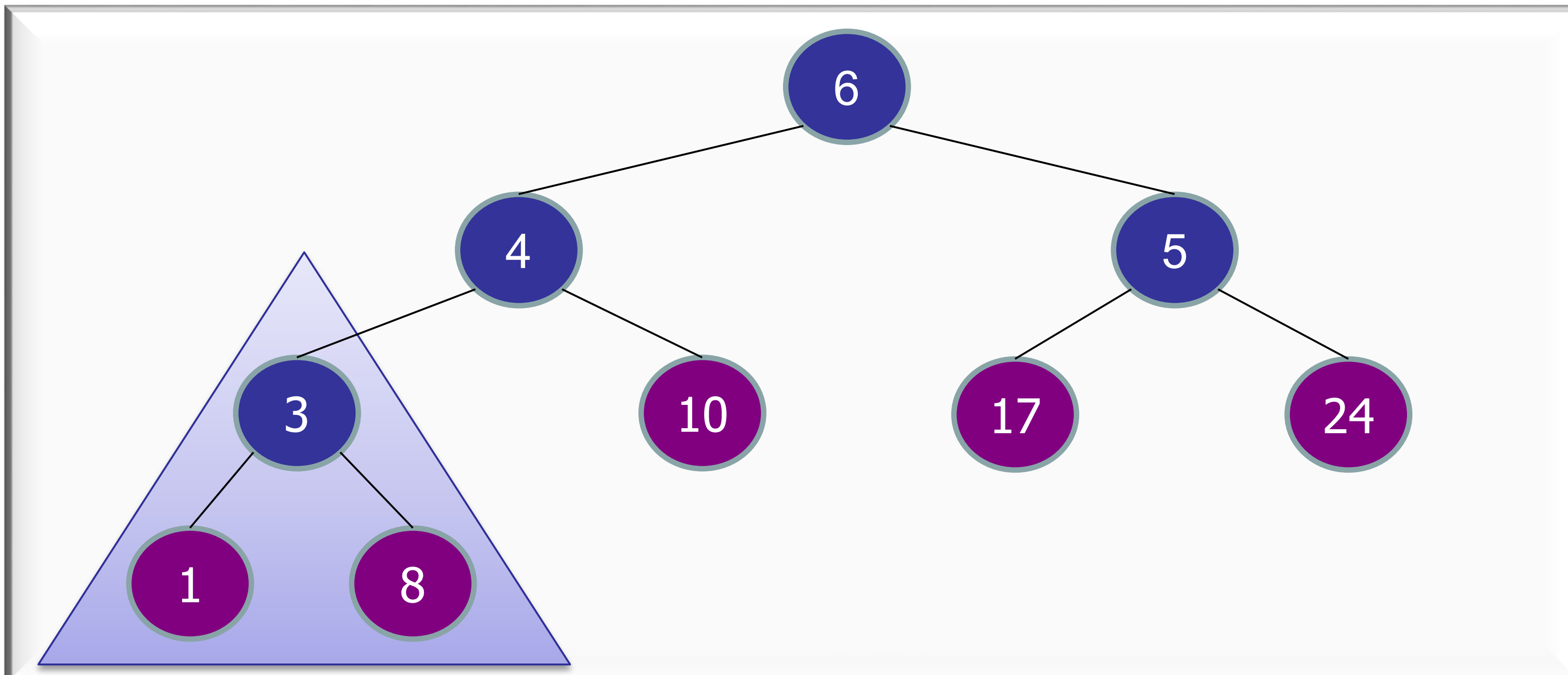


HeapSort

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

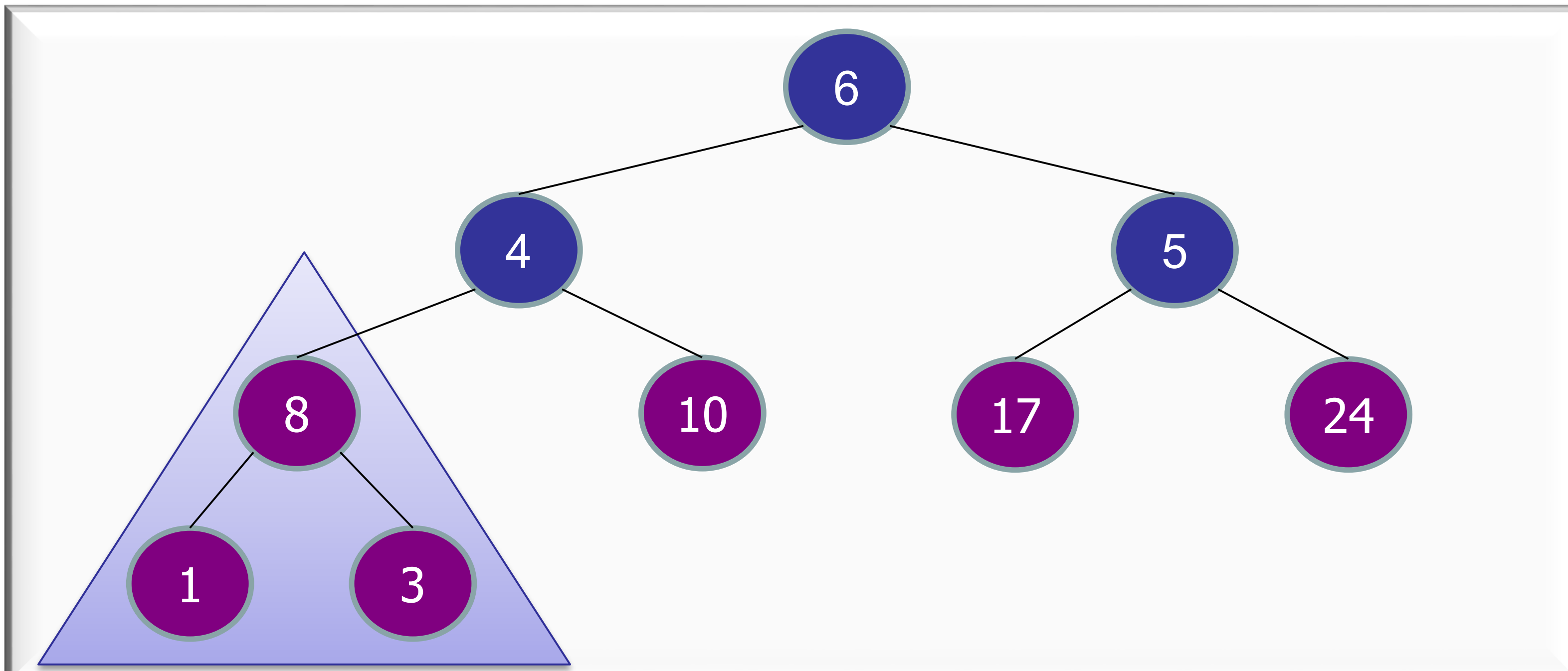


HeapSort

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

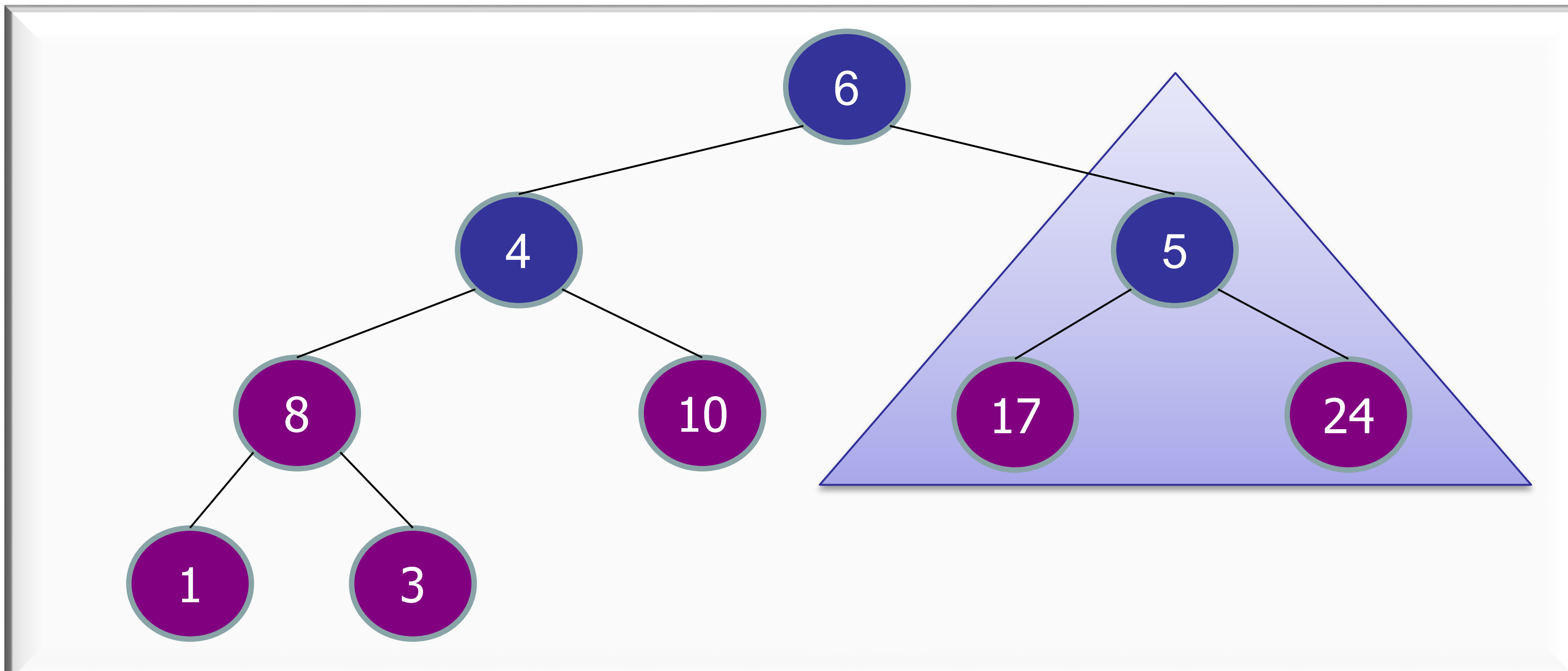


HeapSort

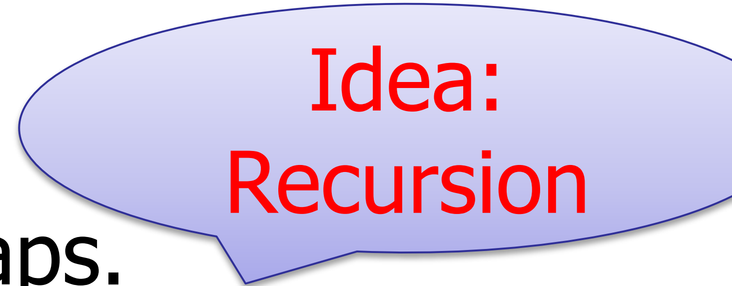
Idea: Recursion

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**

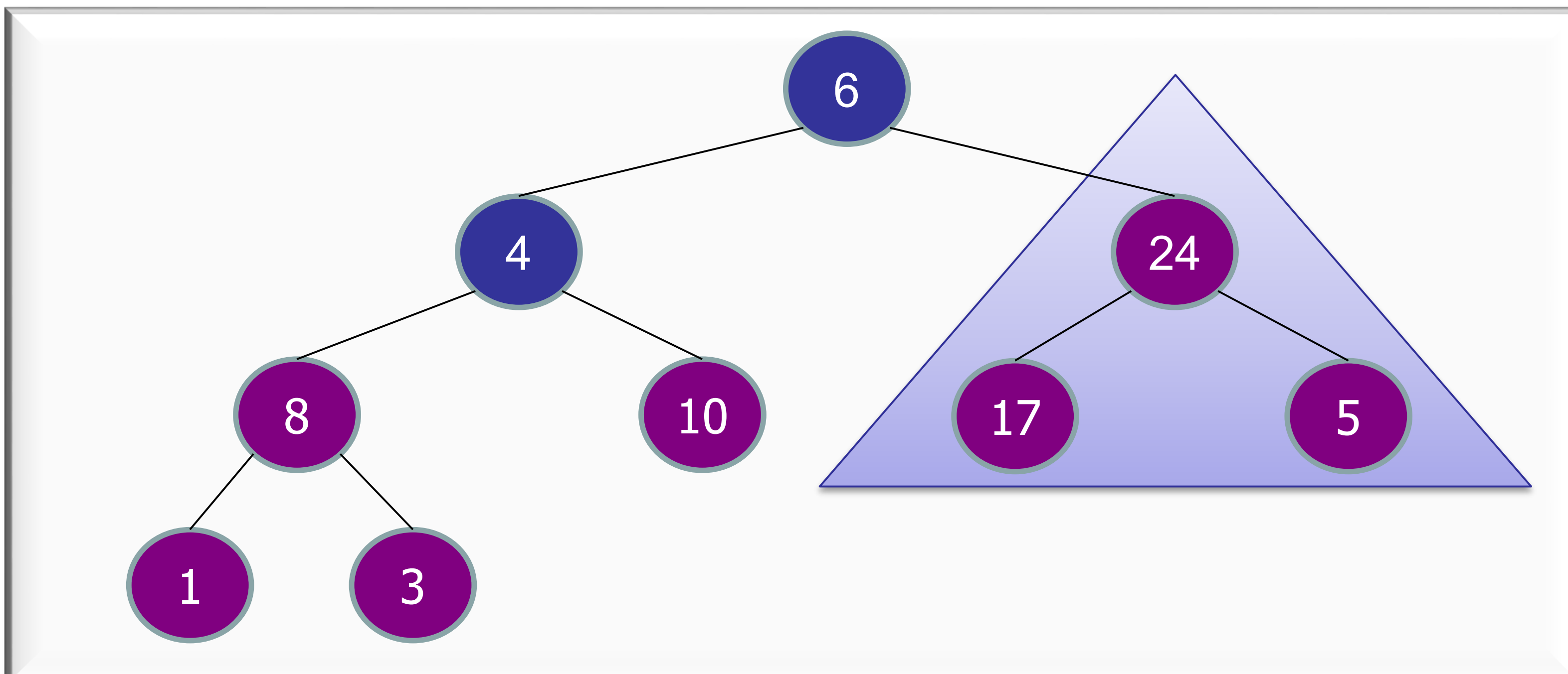


HeapSort



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

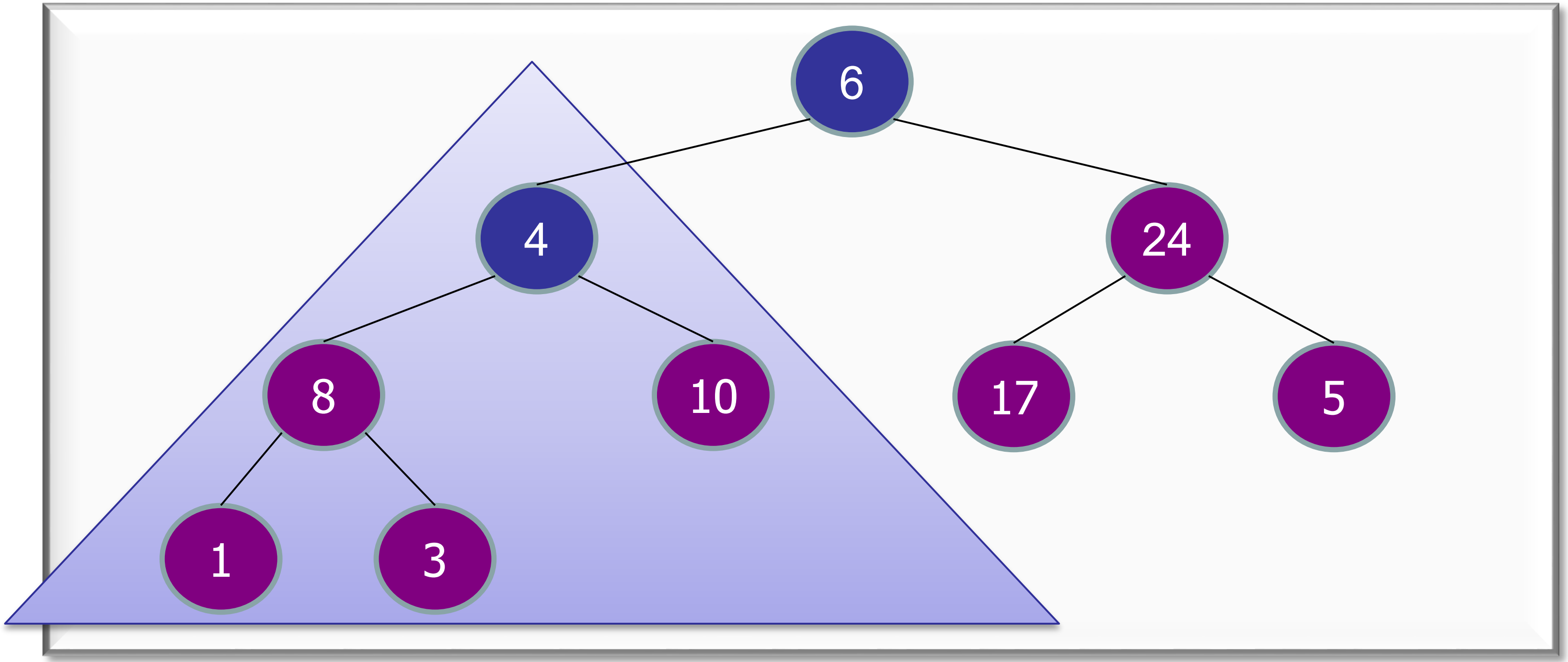


HeapSort

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

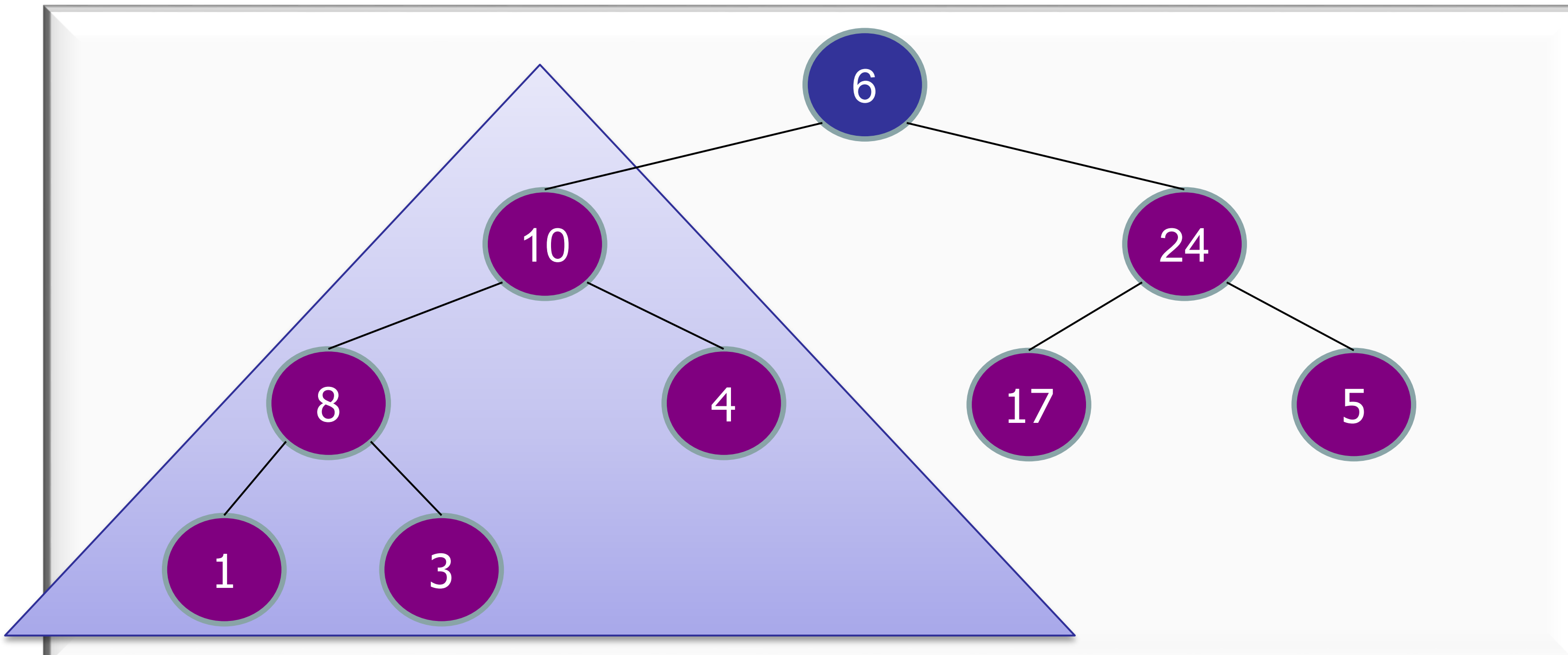


HeapSort

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

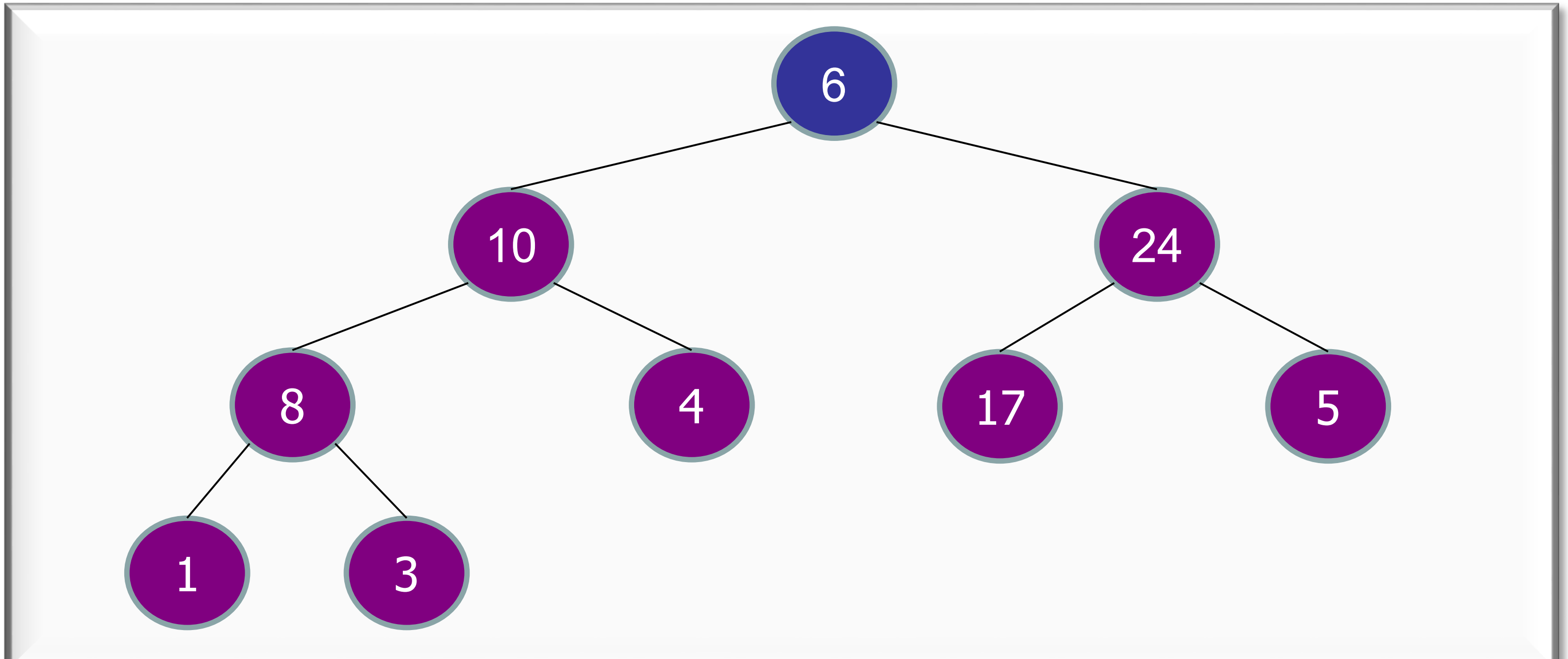


HeapSort

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**

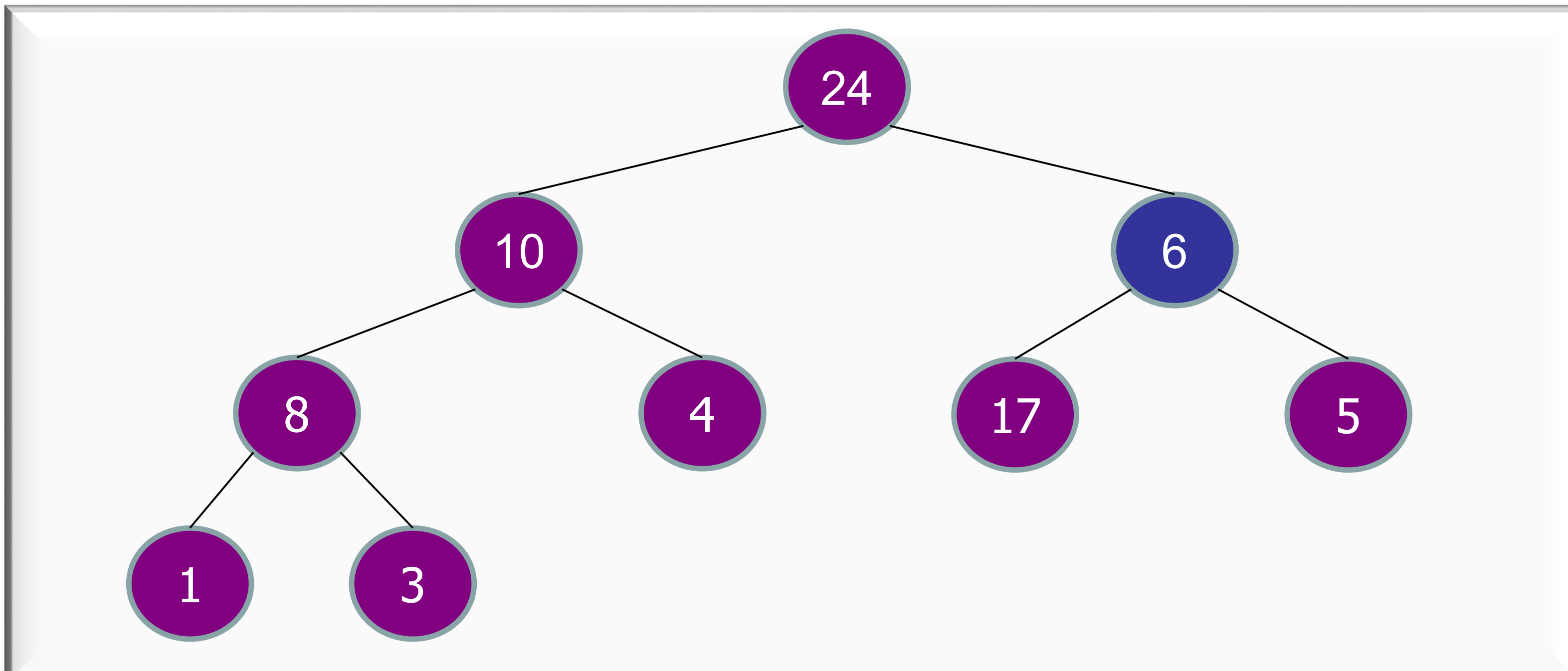


HeapSort

Idea: Recursion

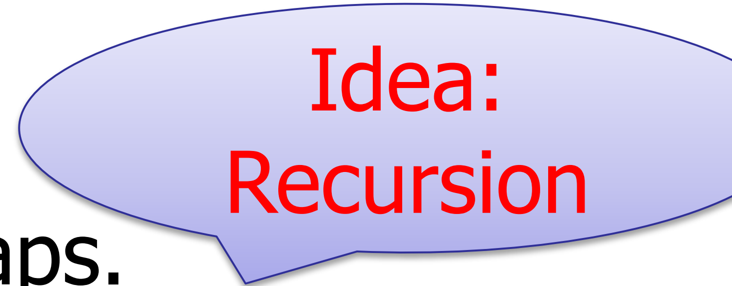
Recursion: left + right are heaps.

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

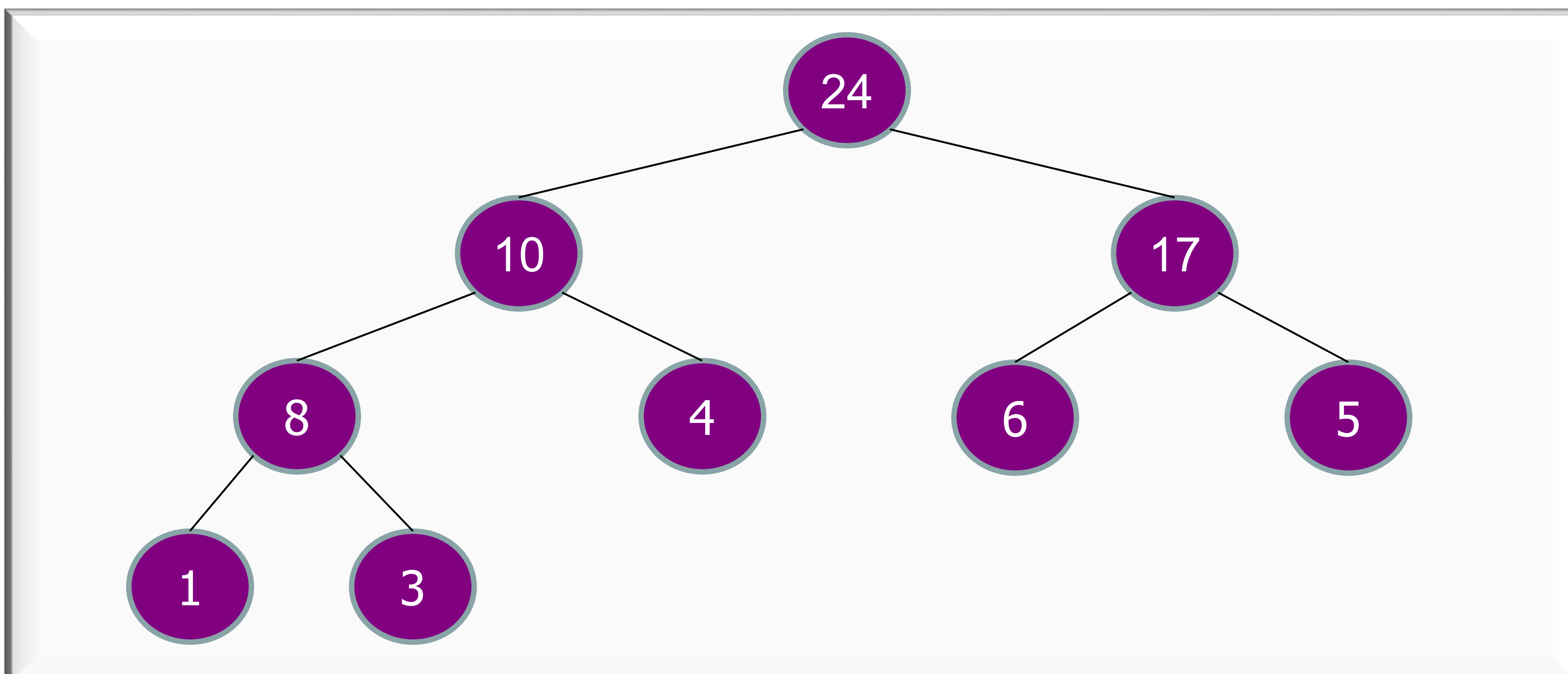


HeapSort

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



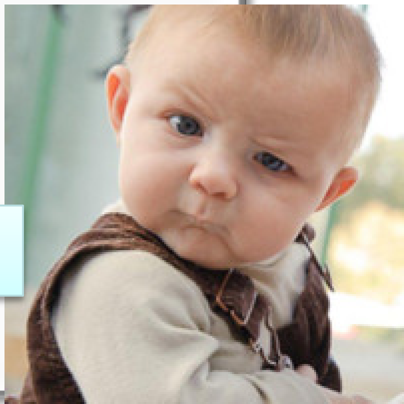
HeapSort

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)
}
```

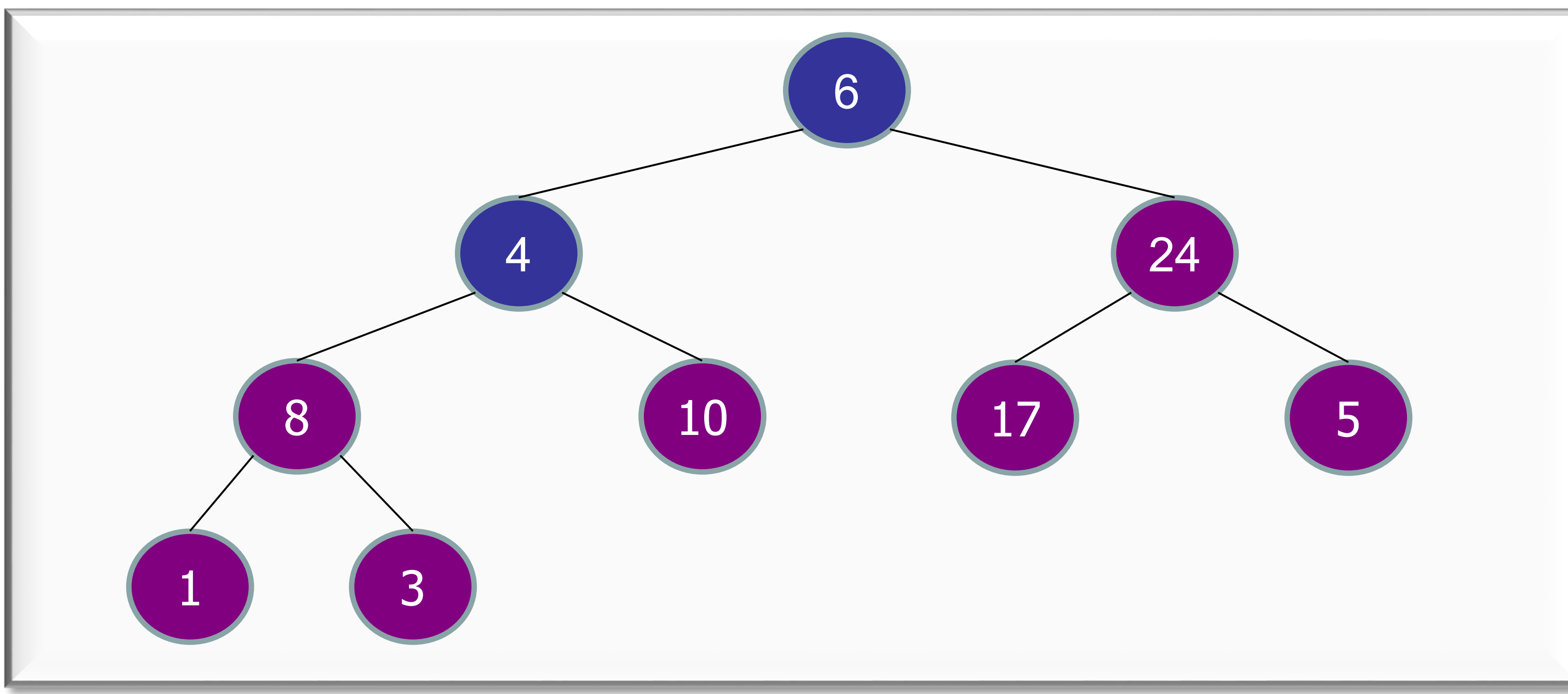
Is it better?!



HeapSort

Observation: cost(bubbleDown) = height

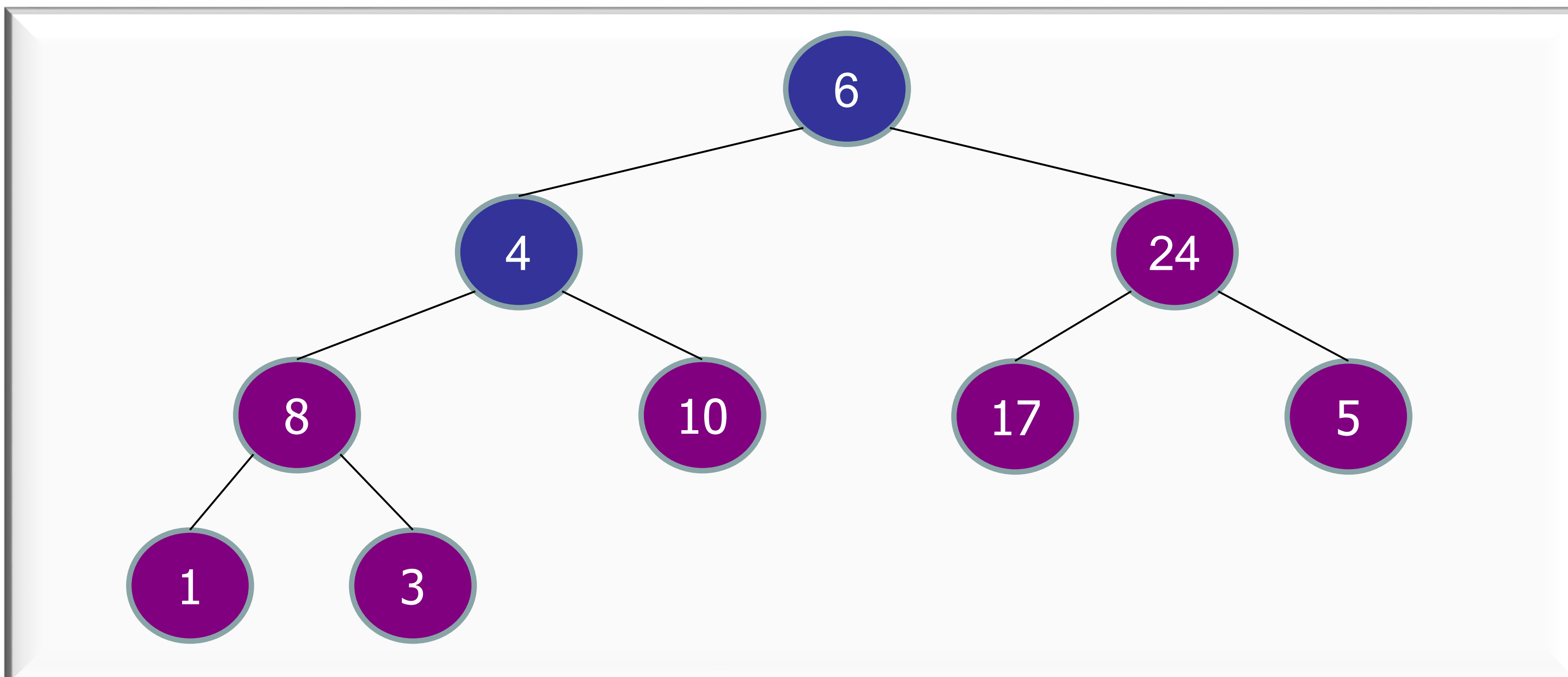
array slot	0	1	2	3	4	5	6	7	8
key	6	4	24	8	10	17	5	1	3



HeapSort

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

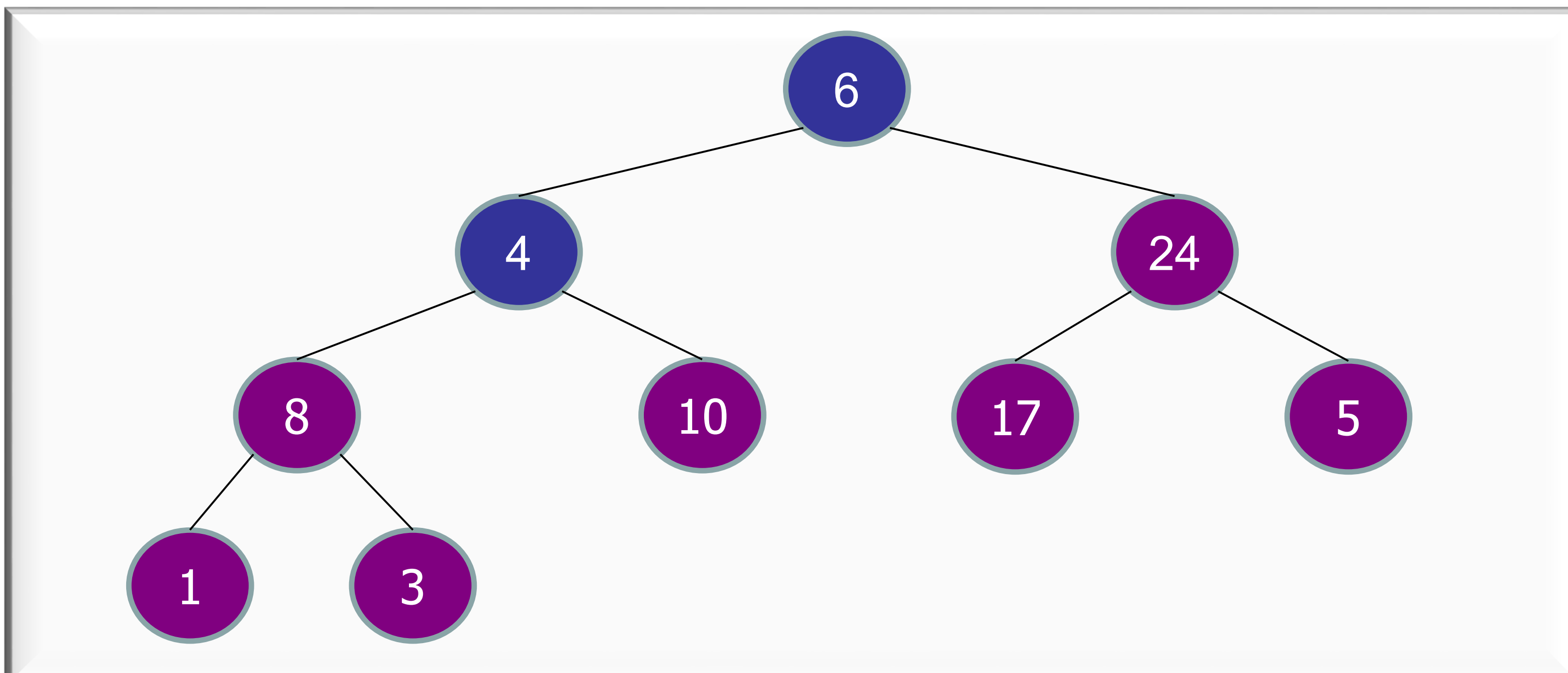
array slot	0	1	2	3	4	5	6	7	8
key	6	4	24	8	10	17	5	1	3



HeapSort

Observation: most nodes have small height!

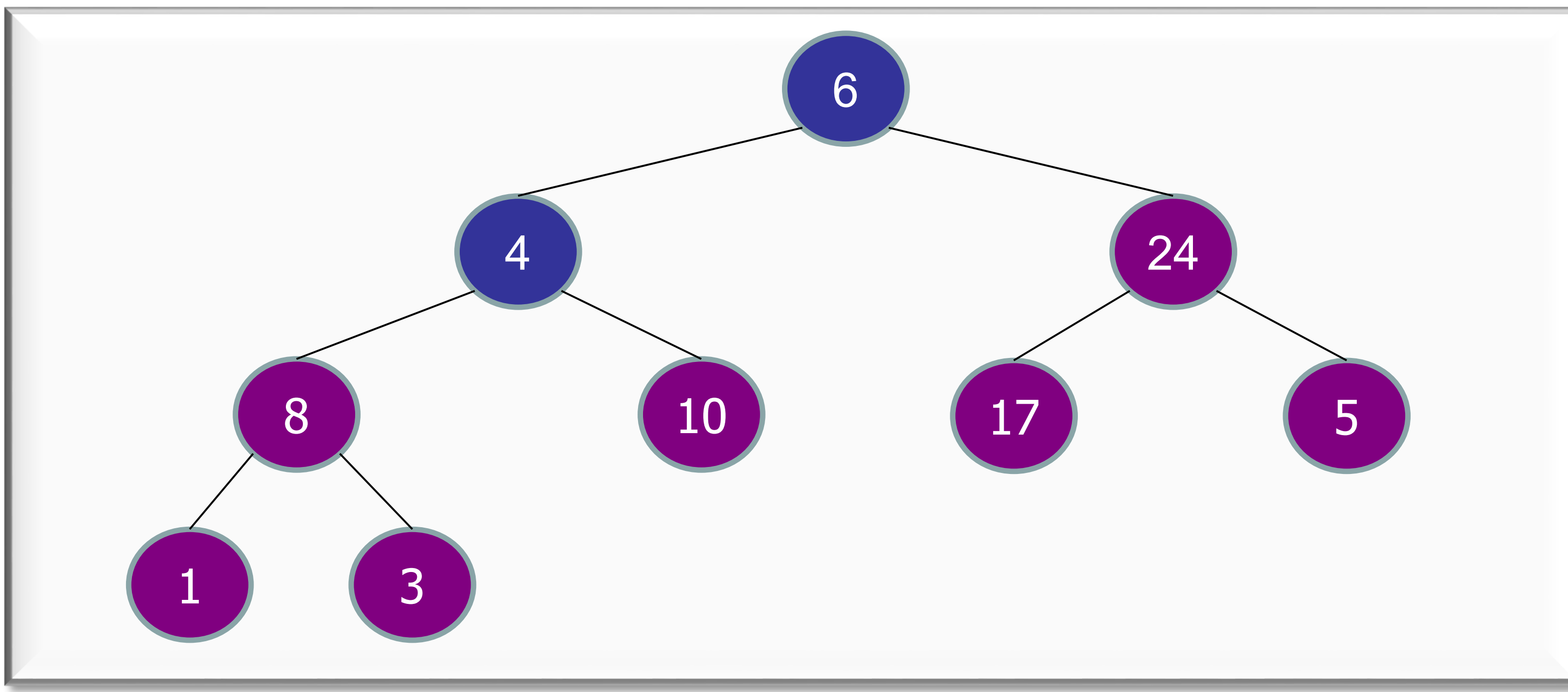
array slot	0	1	2	3	4	5	6	7	8
key	6	4	24	8	10	17	5	1	3



HeapSort

Cost of building a heap:

Height	**0**	**1**	**2**	**3**	**...**	**$\lfloor \log(n) \rfloor$**
Number	**$\lceil n/2 \rceil$**	**$\lceil n/4 \rceil$**	**$\lceil n/8 \rceil$**	**$\lceil n/16 \rceil$**	**...**	**1**



HeapSort

Cost of building a heap:

Height	0	1	2	3	…	$\lfloor \log(n) \rfloor$
Number	$\lceil n/2 \rceil$	$\lceil n/4 \rceil$	$\lceil n/8 \rceil$	$\lceil n/16 \rceil$	…	1

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

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**	**…**	$\lfloor \log(n) \rfloor$
Number	$\lceil n/2 \rceil$	$\lceil n/4 \rceil$	$\lceil n/8 \rceil$	$\lceil n/16 \rceil$	**…**	**1**

$$\sum_{h=0}^{\log n} \frac{n}{2^h} O(h) = cn \sum_{h=0}^{\log n} \frac{1}{2^h} O(h) = cn \cdot O\left(\sum_{h=0}^{\log n} \frac{h}{2^h}\right)$$

$$\sum_{h=0}^{\log n} \frac{h}{2^h} = ?$$

Geometric series

$$\sum_{h=0}^{\infty} x^h = \frac{1}{1-x} \quad \text{if } x < 1$$

Differentiate both sides

$$\sum_{h=0}^{\infty} h x^{h-1} = \frac{1}{(1-x)^2}$$

Multiply both sides by x

$$\sum_{h=0}^{\infty} h x^{h} = \frac{x}{(1-x)^2}$$

Put x = 1/2

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{0.5}{(1-0.5)^2} = 2$$

$$\sum_{h=0}^{\log n} \frac{h}{2^h} \le 2$$

HeapSort

Cost of building a heap:

Height	**0**	**1**	**2**	**3**	**…**	**$\lfloor \log(n) \rfloor$**
Number	**$\lceil n/2 \rceil$**	**$\lceil n/4 \rceil$**	**$\lceil n/8 \rceil$**	**$\lceil n/16 \rceil$**	**…**	**1**

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

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

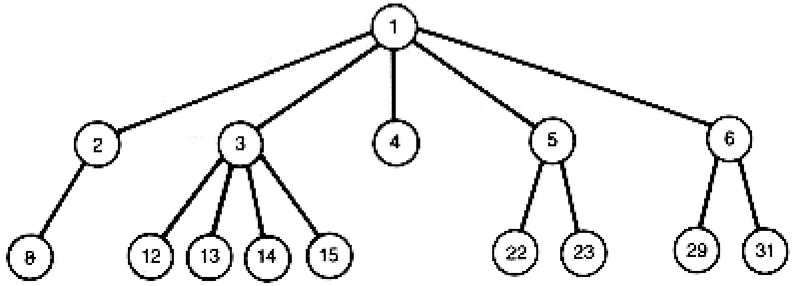
Unsorted list → Heap: $O(n)$

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.