CS2020
Data Structures and Algorithms

Welcome!

Roadmap

Part I: Priority Queues

- Binary Heaps
- HeapSort

Part II: Disjoint Set

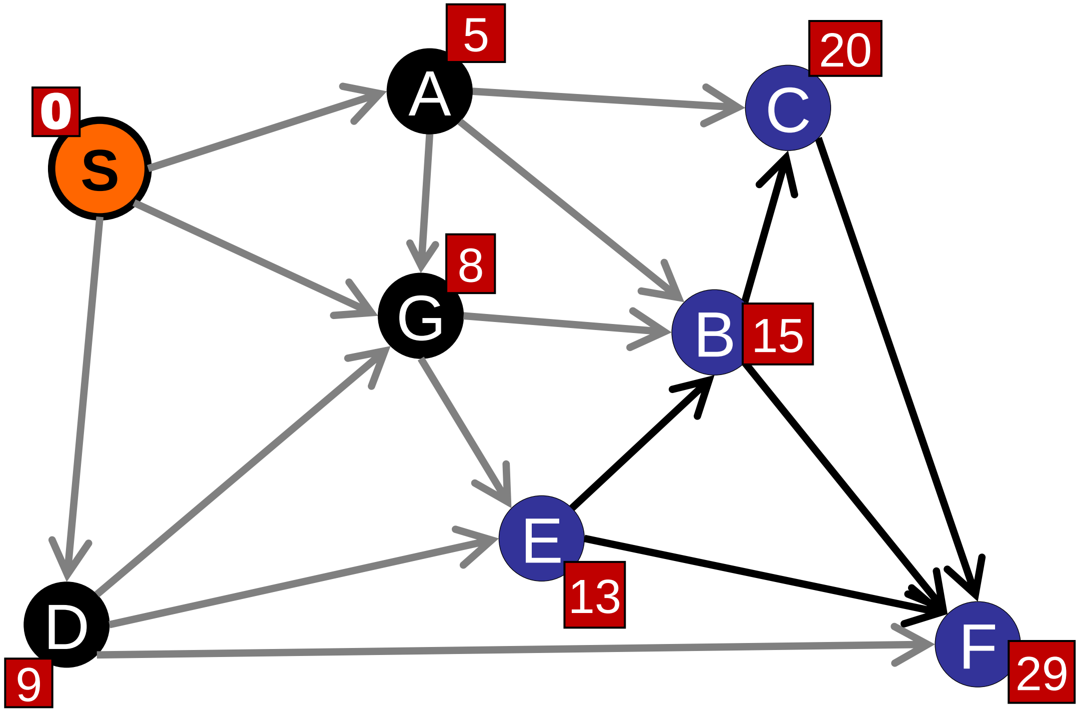
- Problem: Dynamic Connectivity
- Algorithm: Union-Find
- Applications

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: Dijkstra's Algorithm:



Vertex	Dist.
E	13
B	15
C	20
F	29

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
CS2020 PS6	March 31
Study for Quiz 2	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

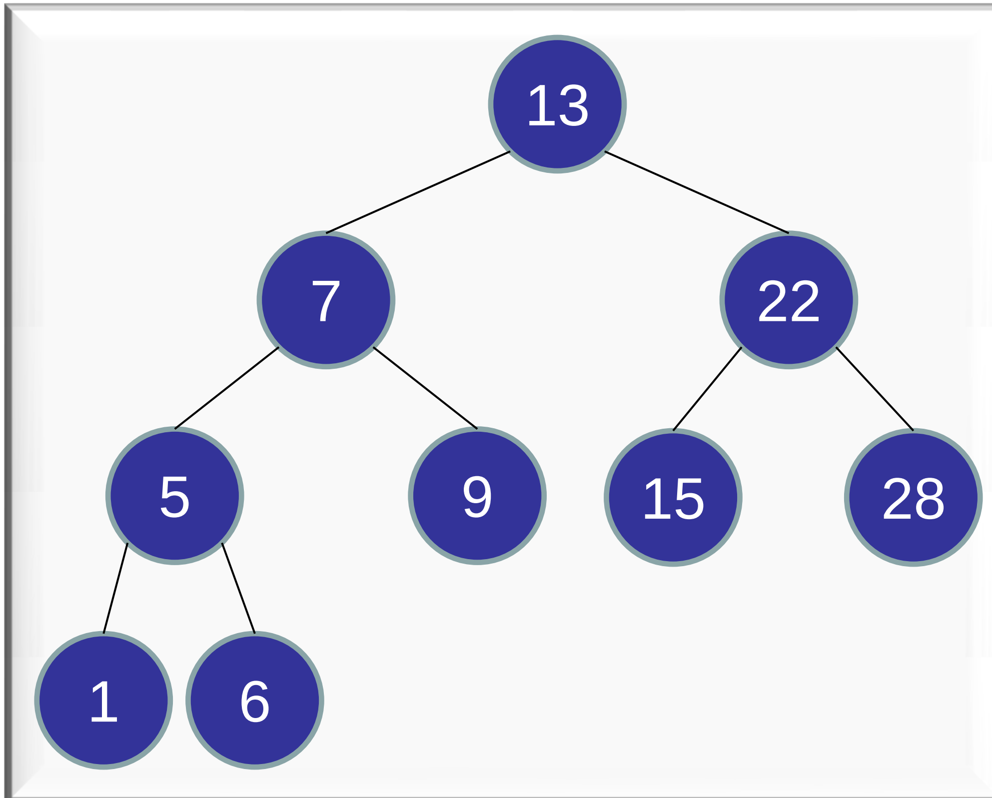
- insert: O(1)
  - Add object to end of list
- extractMax: O(n)
  - Search for largest element in array.
  - Remove and move everything over.

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.

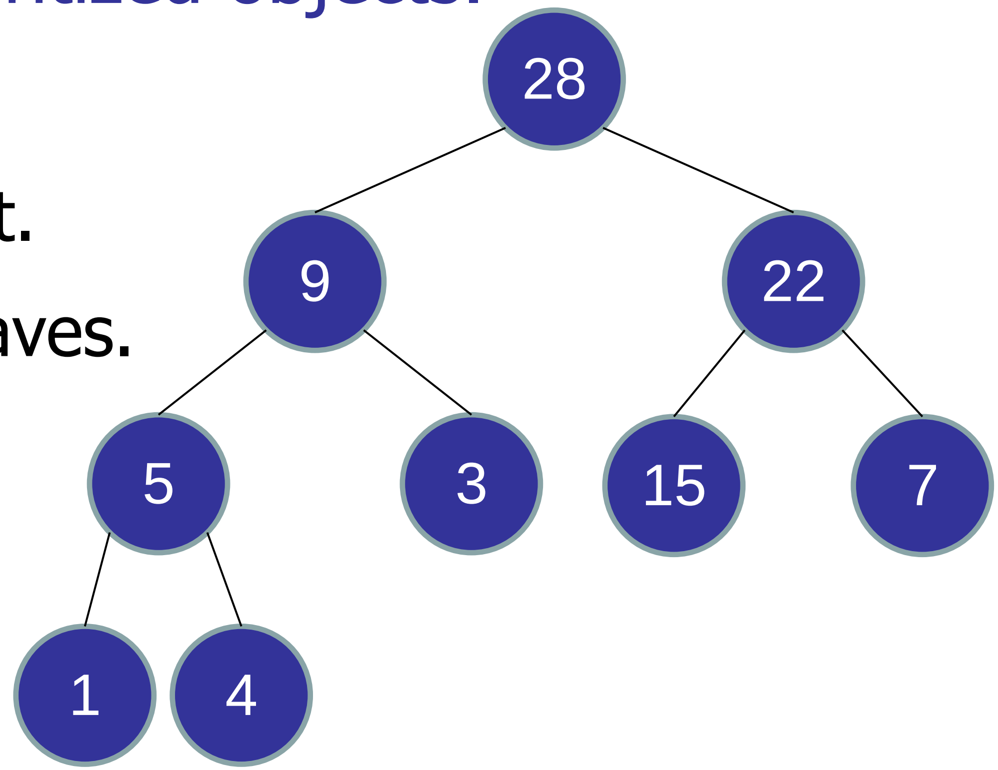
Dijkstra's Performance

PQ Implementation	insert	deleteMin	decreaseKey	Total
Unsorted Array	1	V	1	**$O(V^2)$**
Sorted Array	V	1	V	**O(EV)**
AVL Tree	log V	log V	log V	**O(E log V)**
Fibonacci Heap	1	log V	1	**O(E + V log V)**

Heap

(aka Binary Heap or MaxHeap)

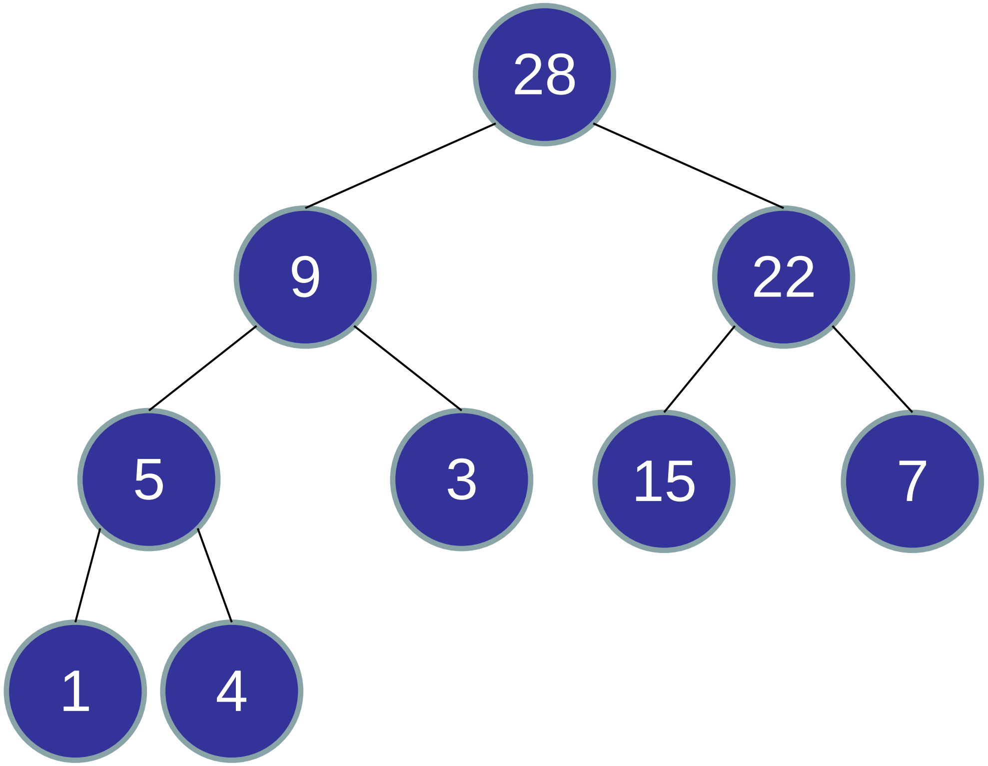
- Implements a Max Priority Queue
- Maintain a set of prioritized objects.
- Store items in a tree.
    - Biggest items at root.
    - Smallest items at leaves.



Two Properties of a Heap

1. Heap Ordering

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

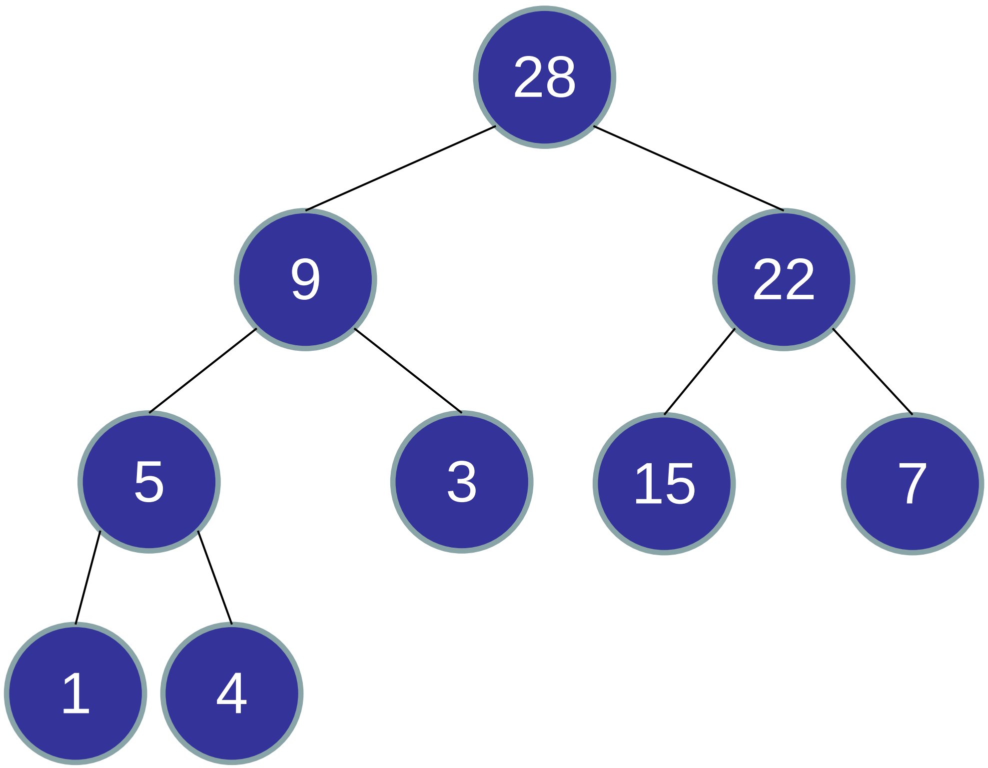


Note: not a binary search tree.

Two Properties of a Heap

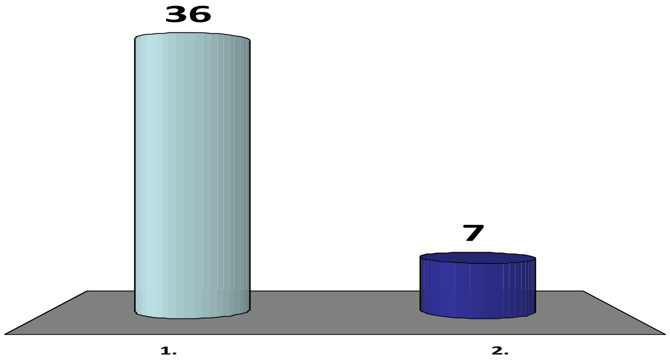
2. Complete binary tree

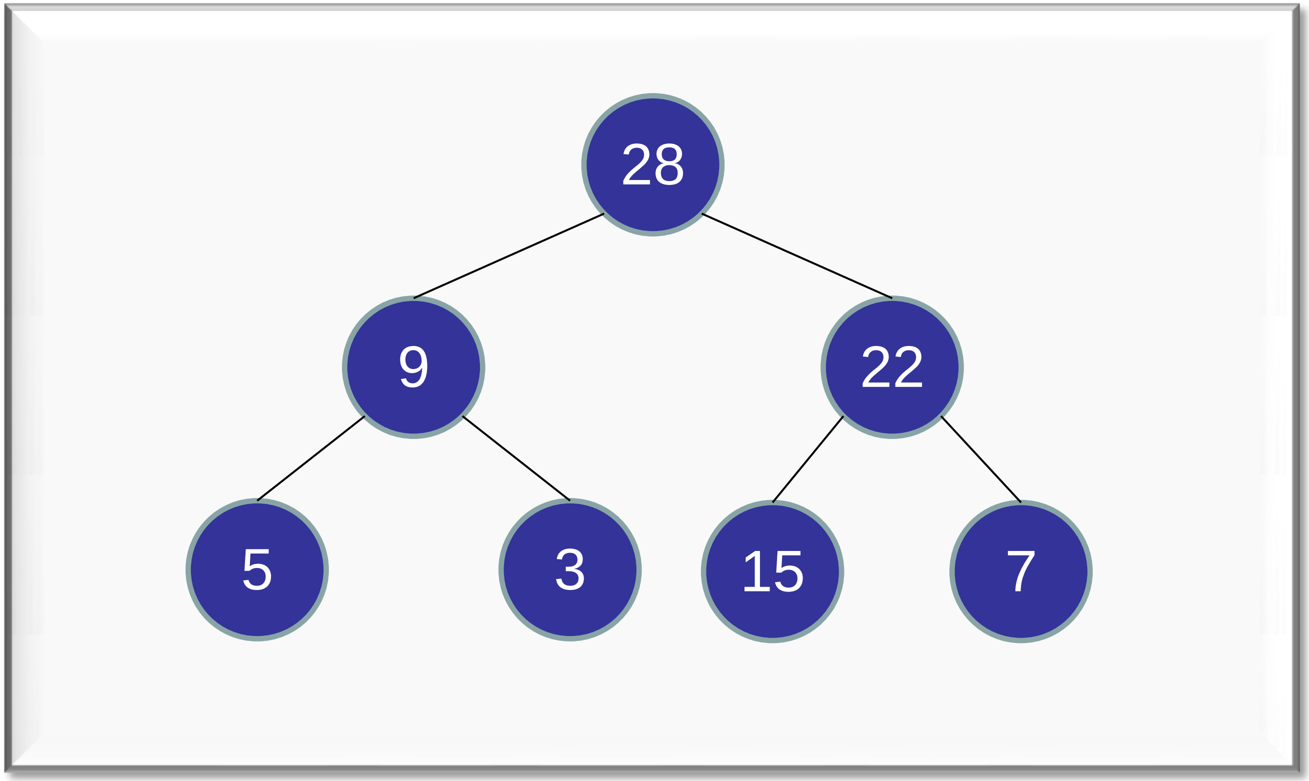
- Every level is full, except possibly the last.
- All nodes are as far left as possible.



Is it a heap?

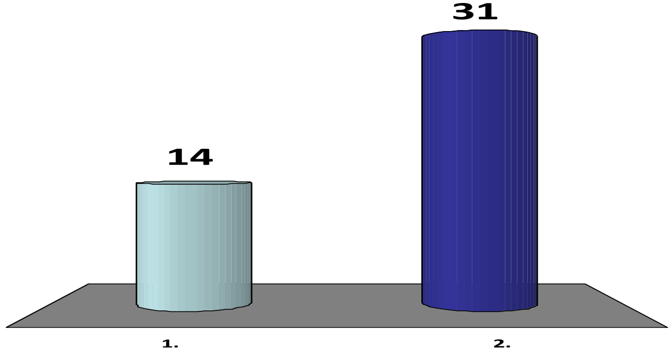
1. Yes
2. No.

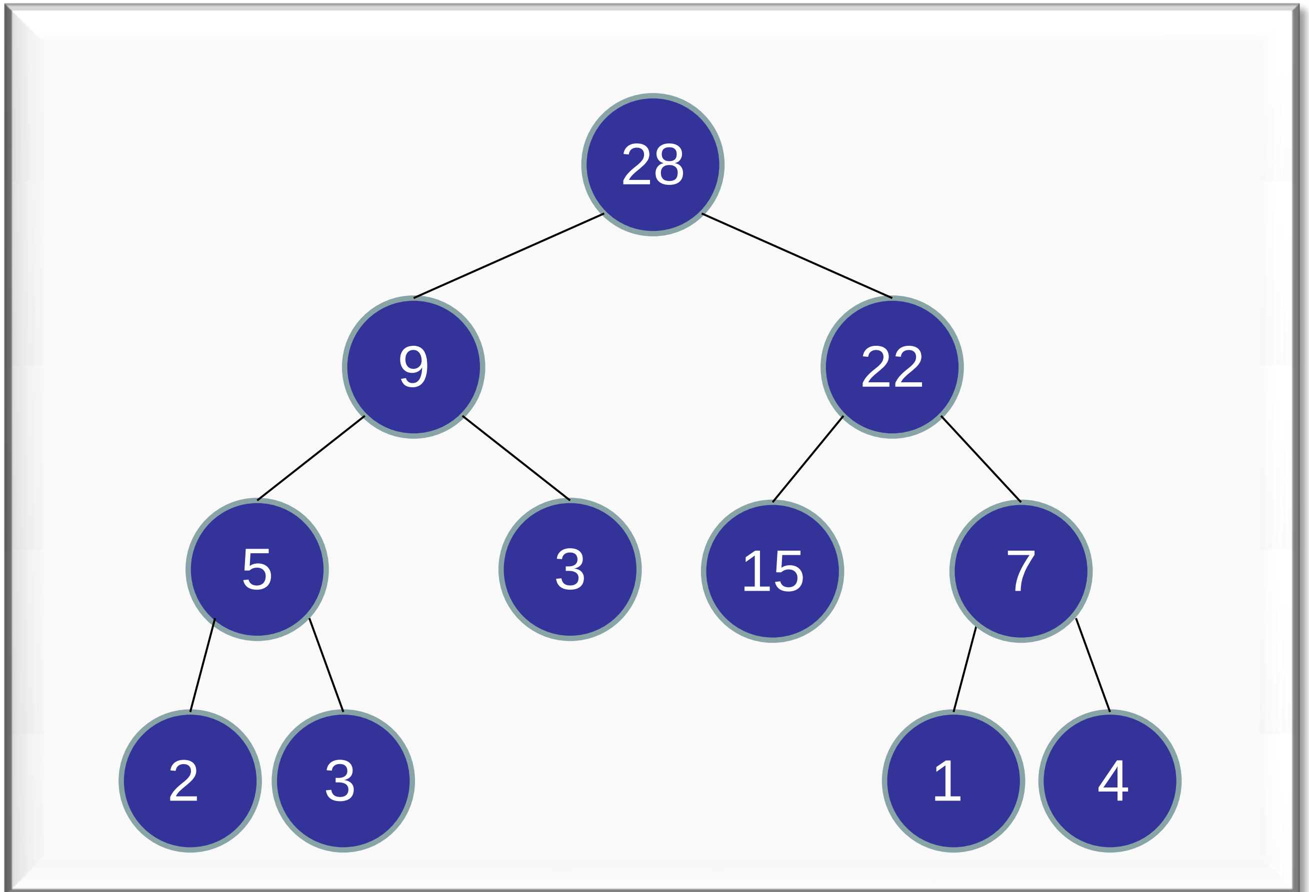




Is it a heap?

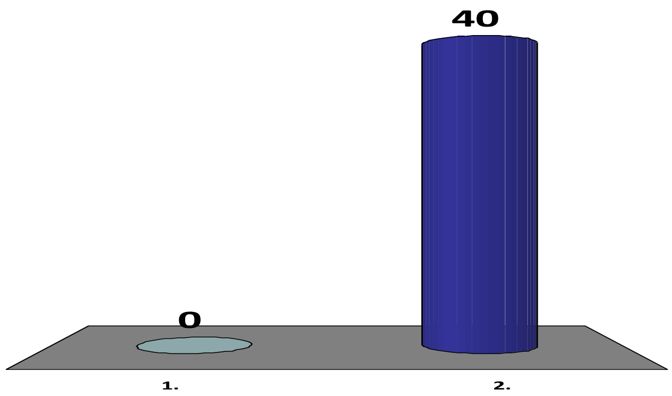
1. Yes
2. No.

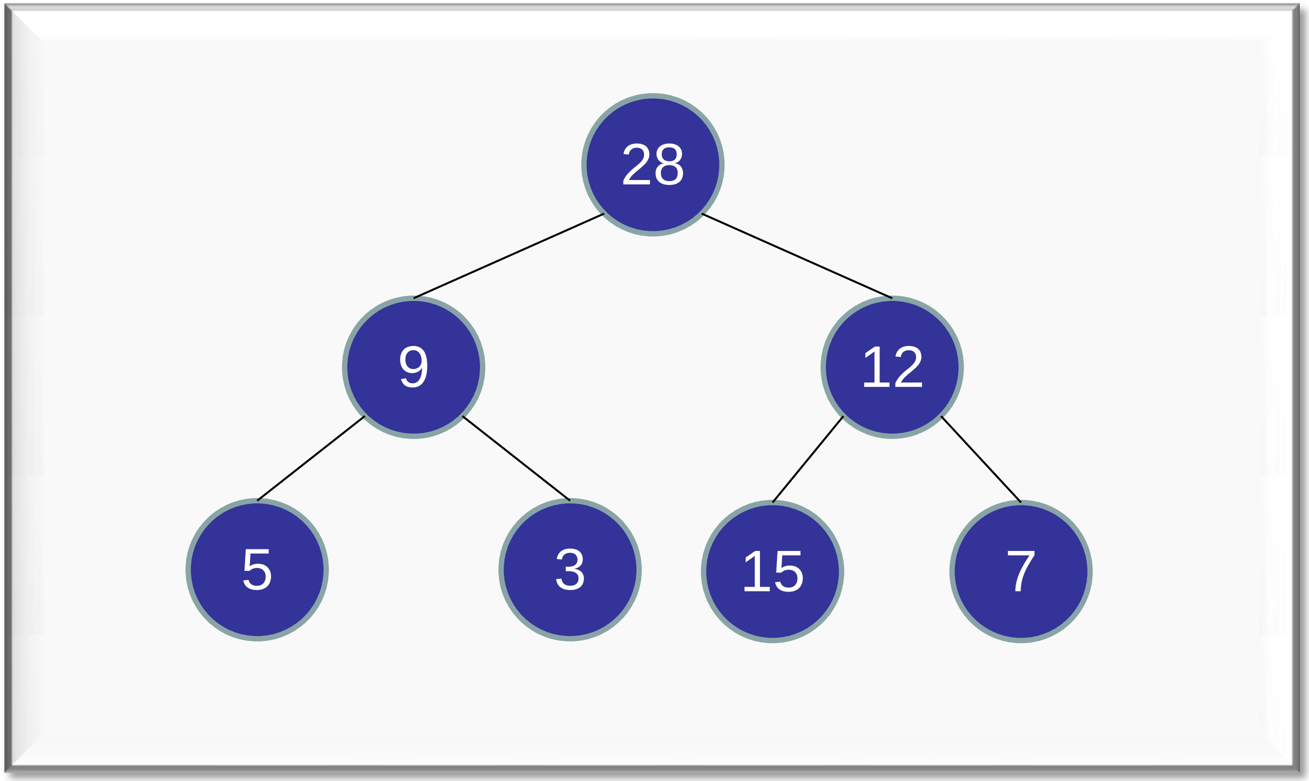




Is it a heap?

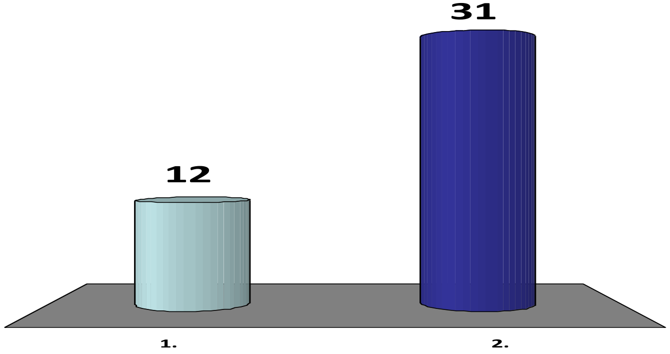
1. Yes
2. No.

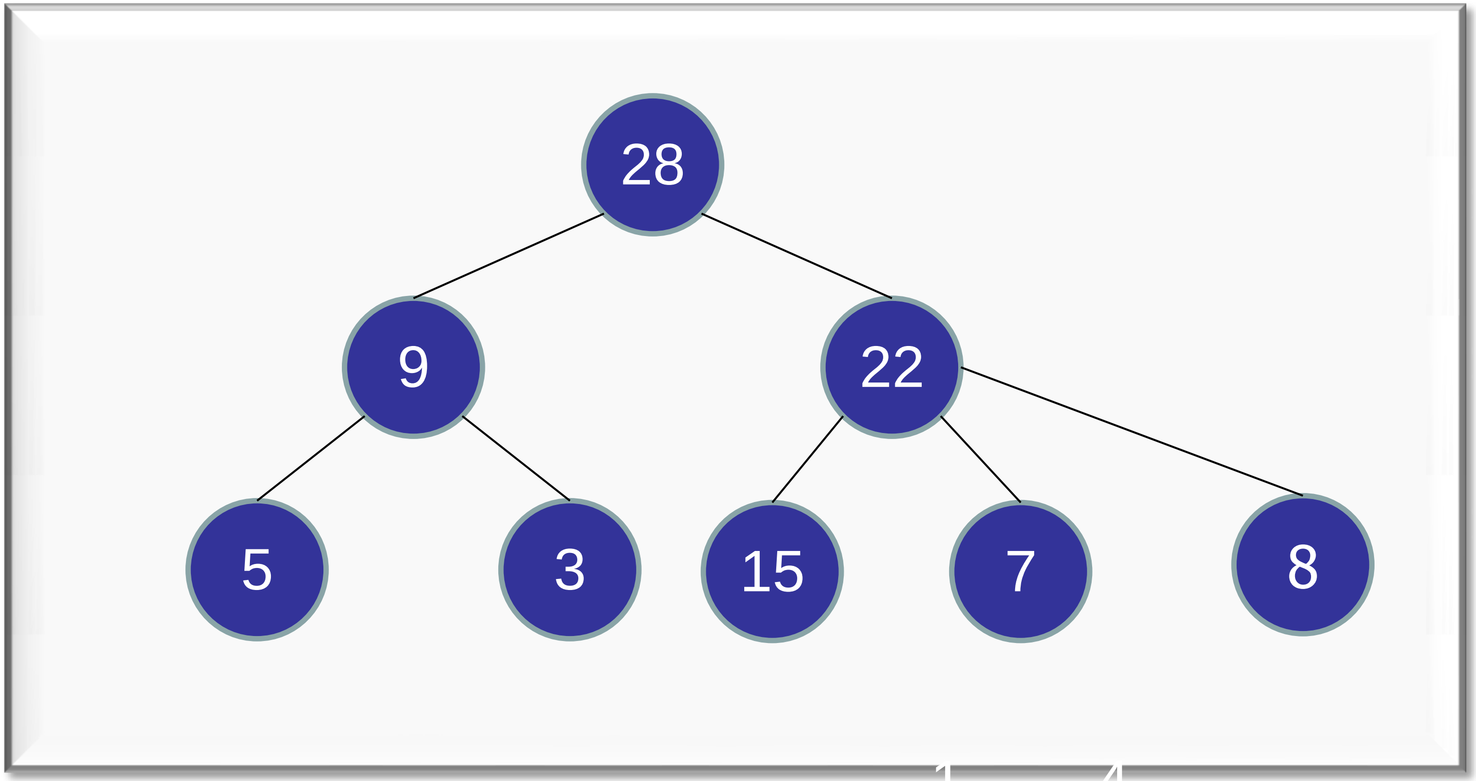




Is it a heap?

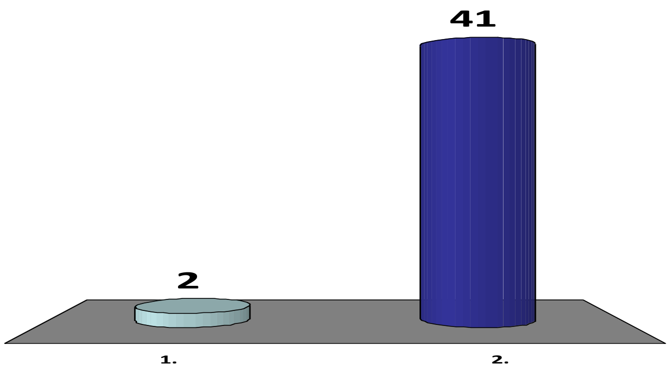
1. Yes
2. No.

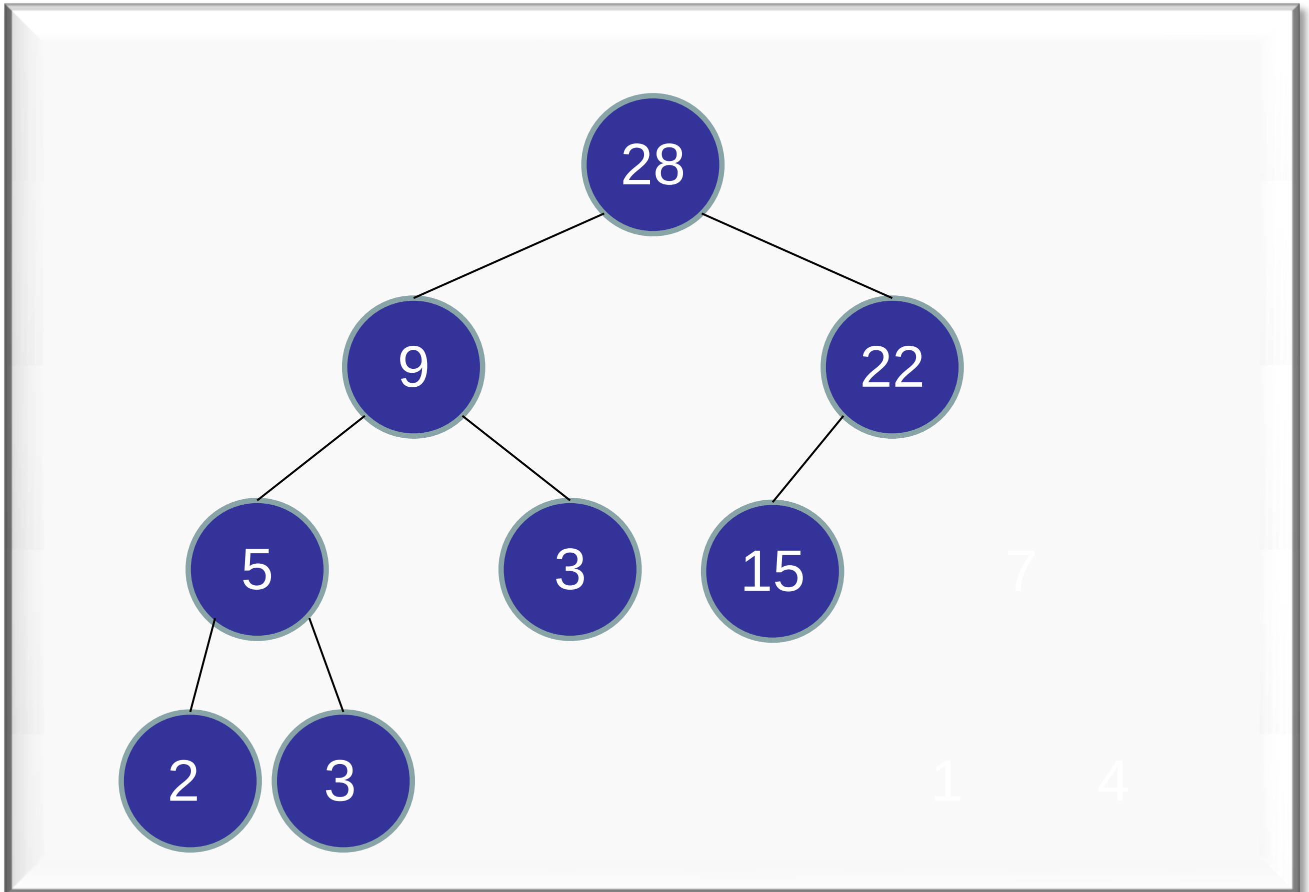




Is it a heap?

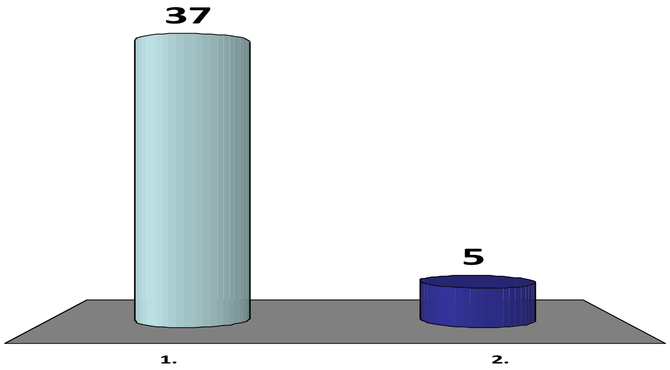
1. Yes
2. No.

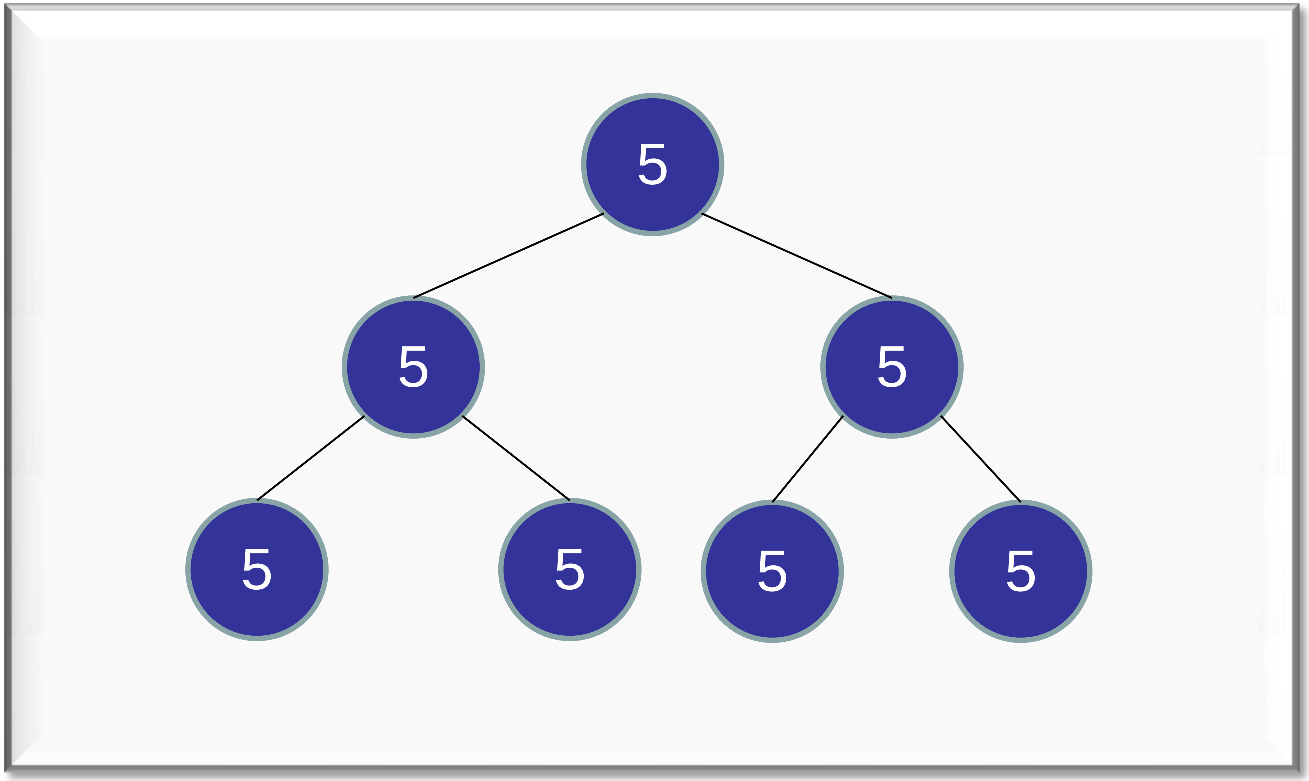




Is it a heap?

1. Yes ✓
2. No.

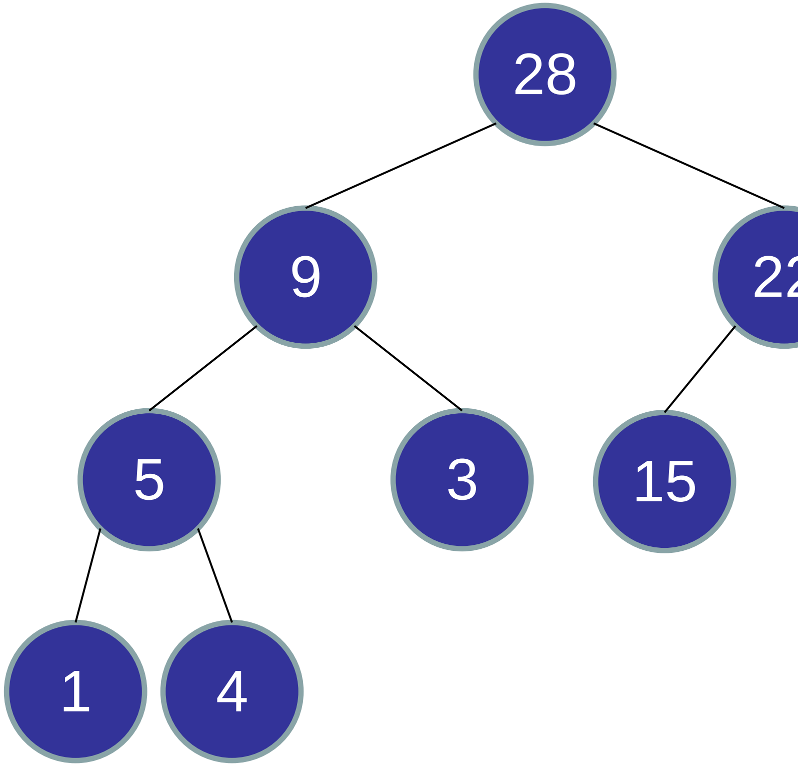




Heap

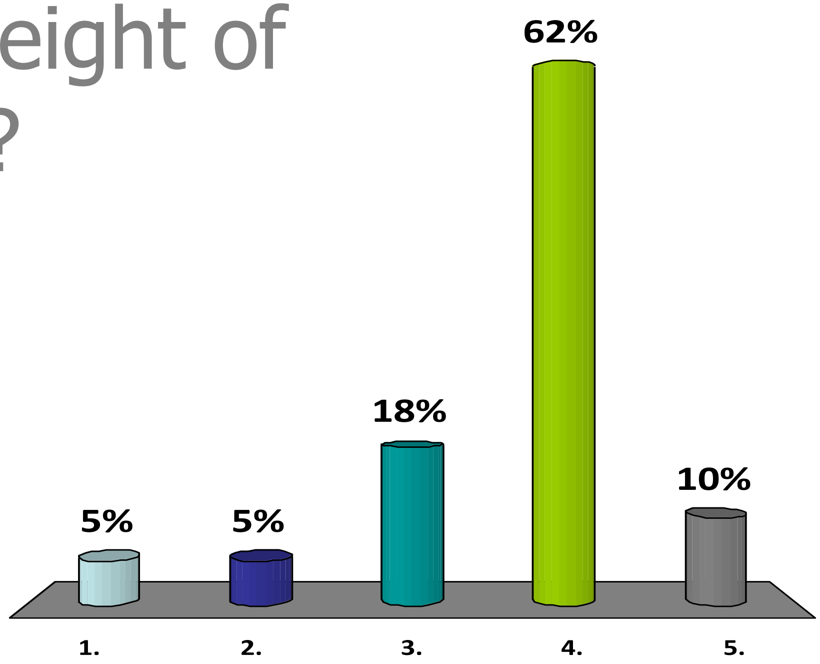
(aka Binary Heap or MaxHeap)

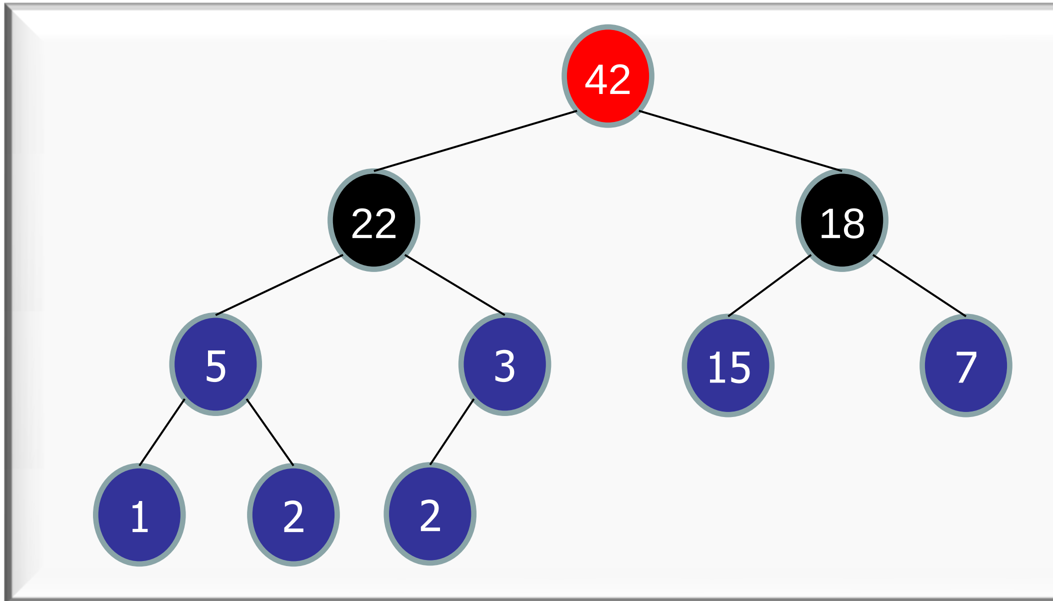
- Implements a Max Priority Queue
- Maintain a set of prioritized objects.
- Store items in a tree.
  - Biggest items at root.
  - Smallest items at leaves.
- Two properties:
  1. Heap Ordering
  2. Complete Binary Tree



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))

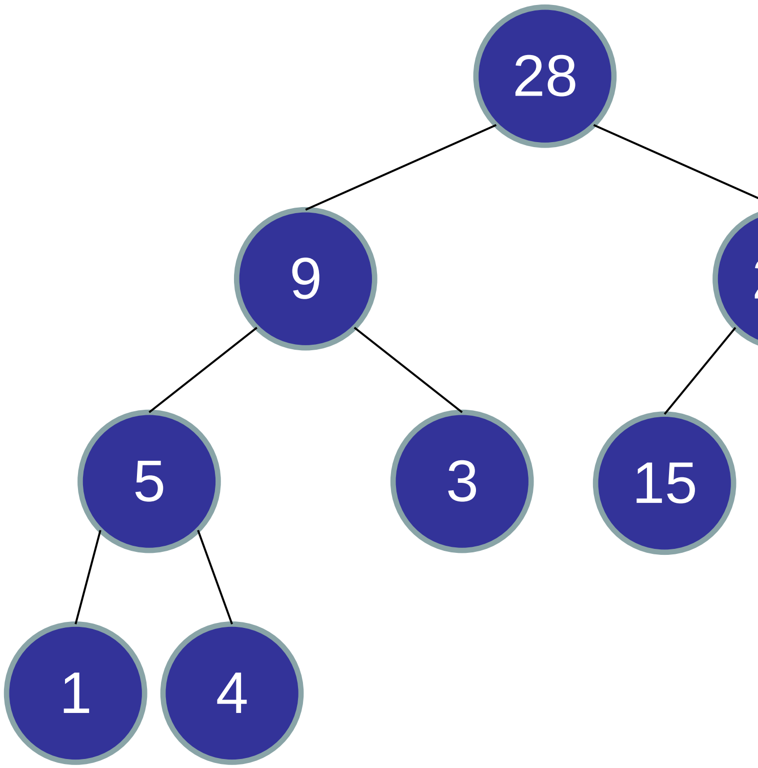




Heap

(aka Binary Heap or MaxHeap)

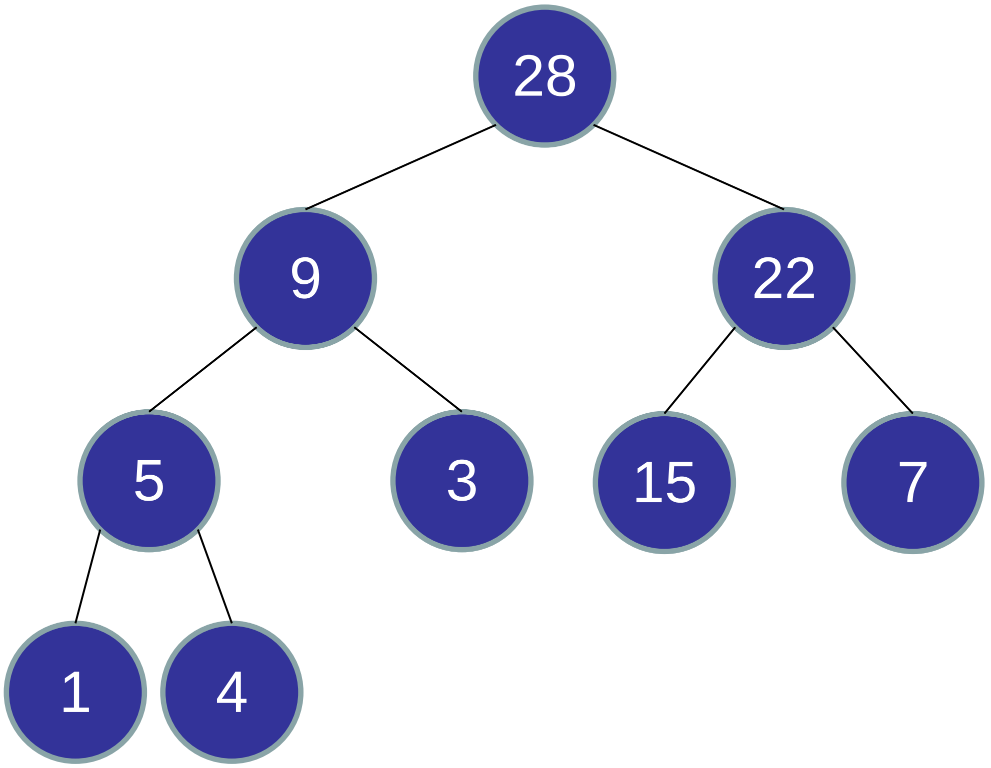
- Implements a Max Priority Queue
- Maintain a set of prioritized objects.
- Store items in a tree.
  - Biggest items at root.
  - Smallest items at leaves.
- Two properties:
  1. Heap Ordering
  2. Complete Binary Tree
- Height: O(log n)



Heap

Priority Queue Operations

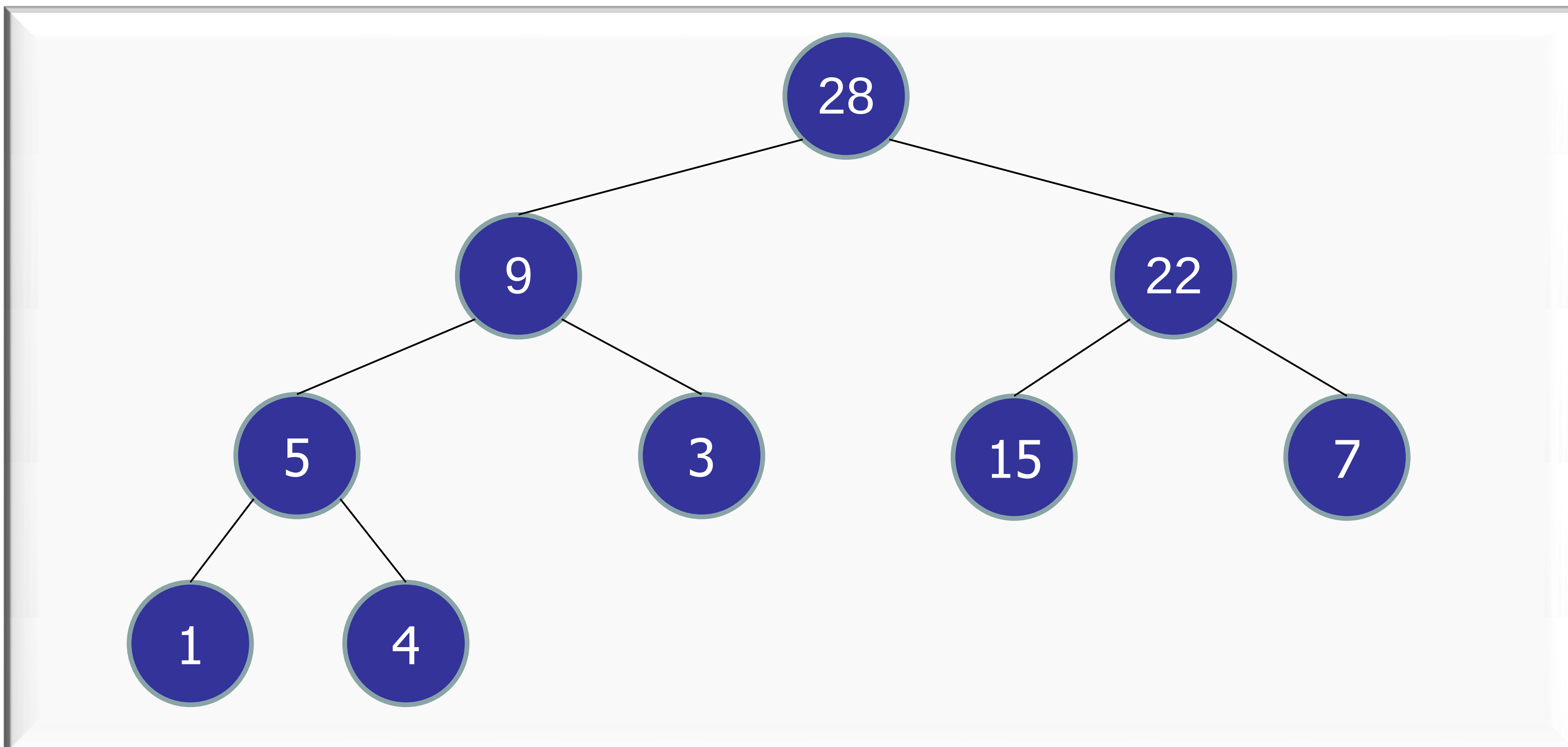
- insert
- extractMax
- increaseKey
- decreaseKey
- delete



Inserting in a Heap

`insert(25)`:

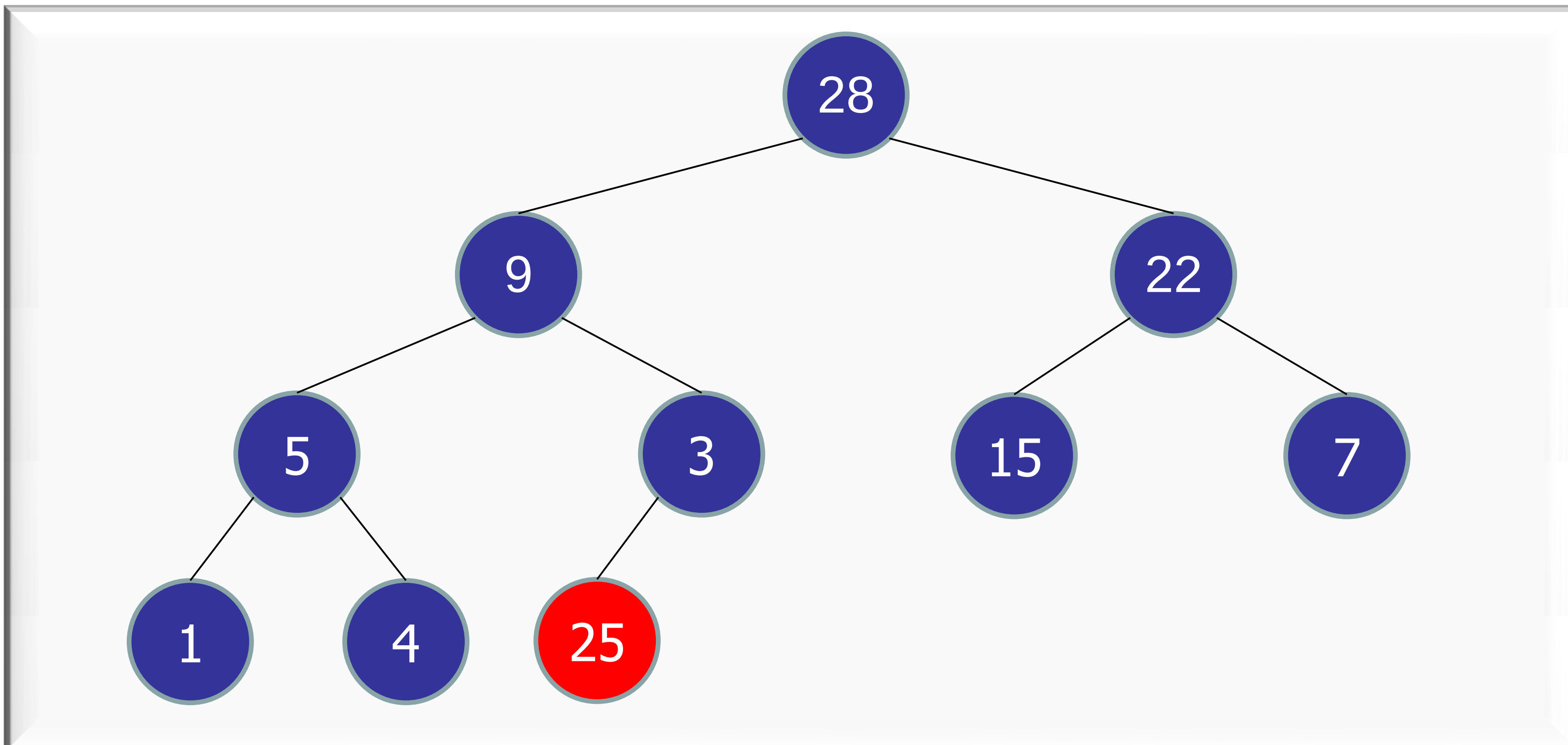
– Step one: add a new leaf with priority 25.



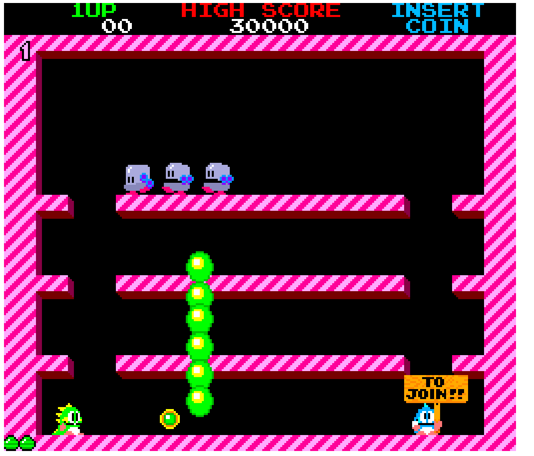
Inserting in a Heap

`insert(25)`:

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

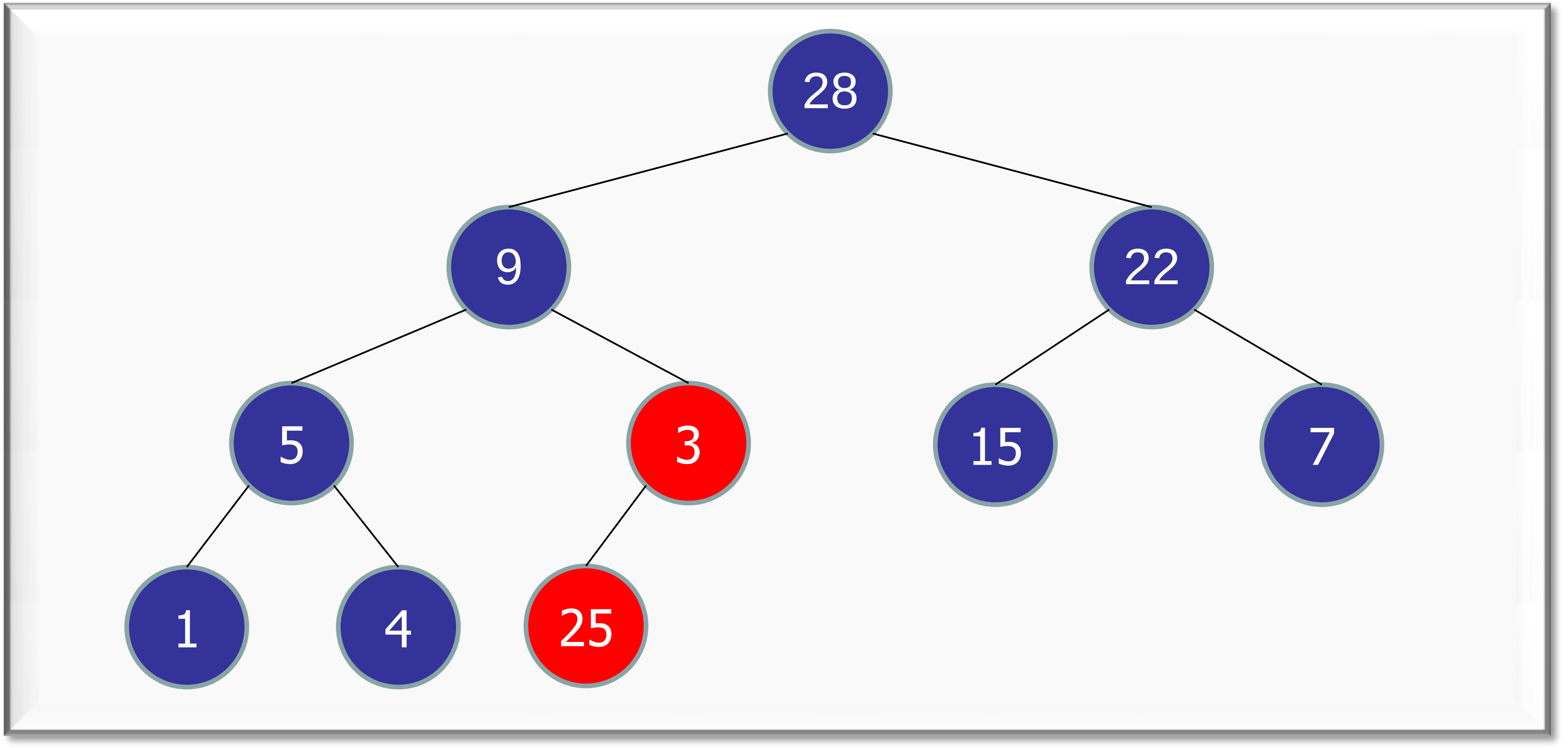


Inserting in a Heap



`insert(25)`:

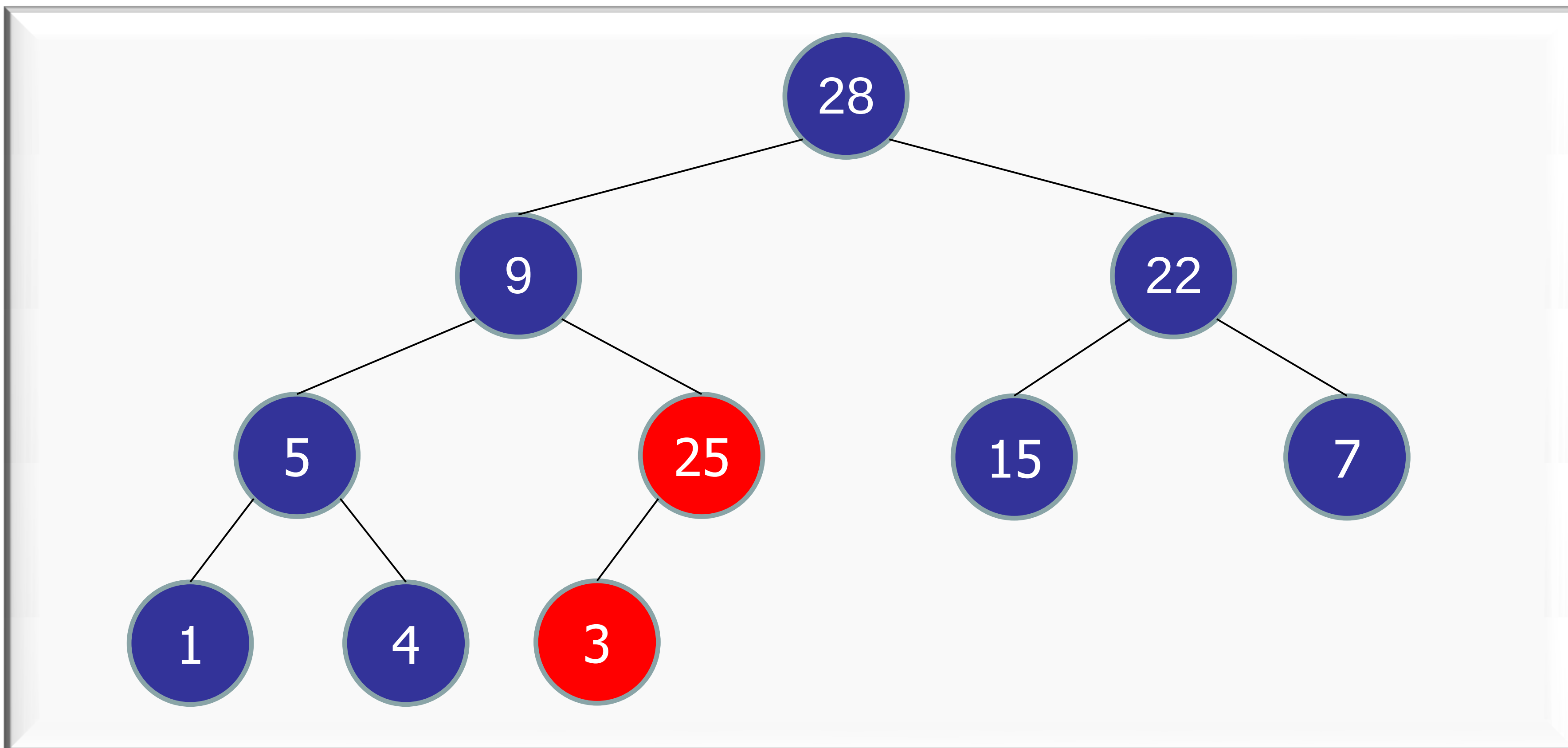
– Step one: add a new leaf with priority 25.
– Step two: bubble up



Inserting in a Heap

`insert(25)`:

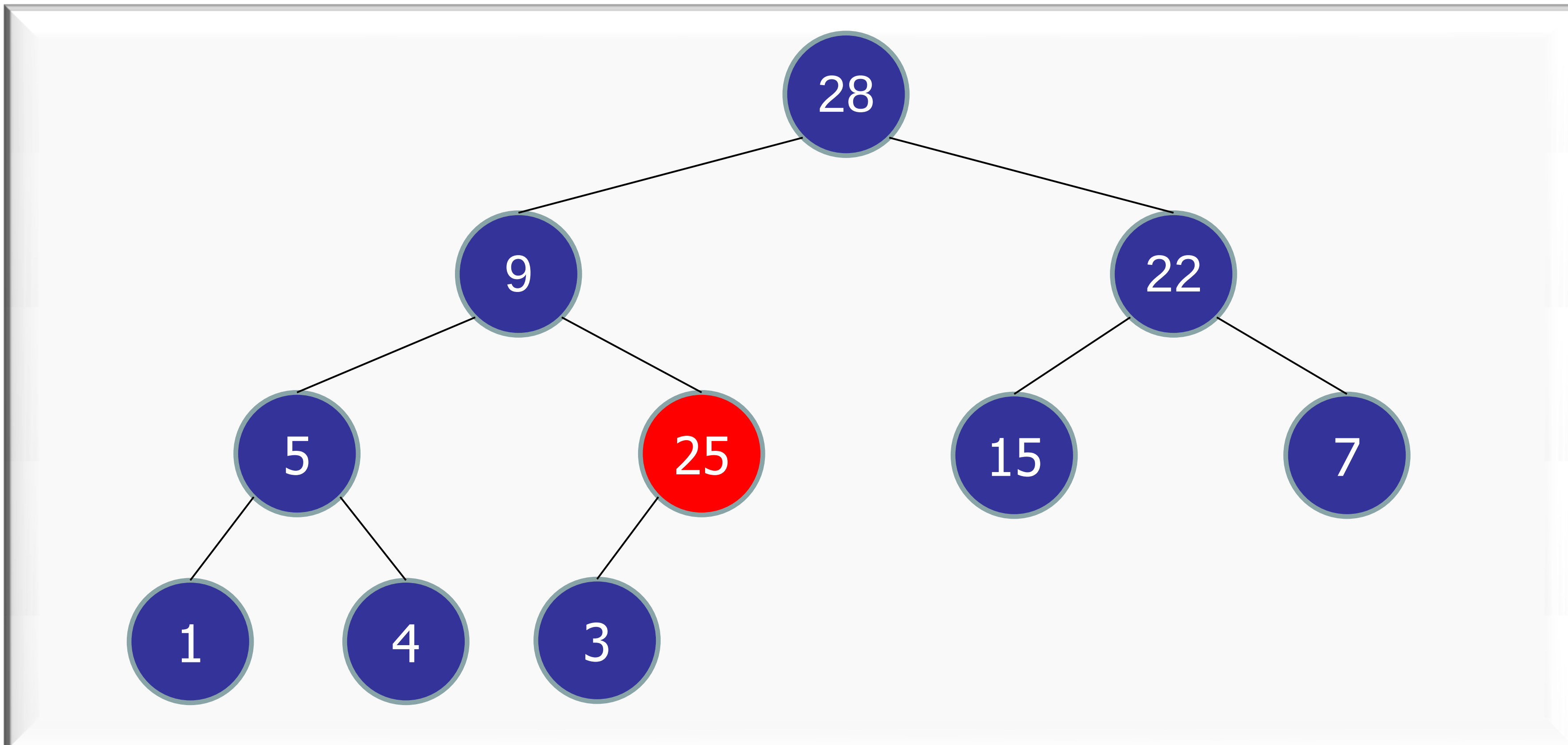
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

`insert(25)`:

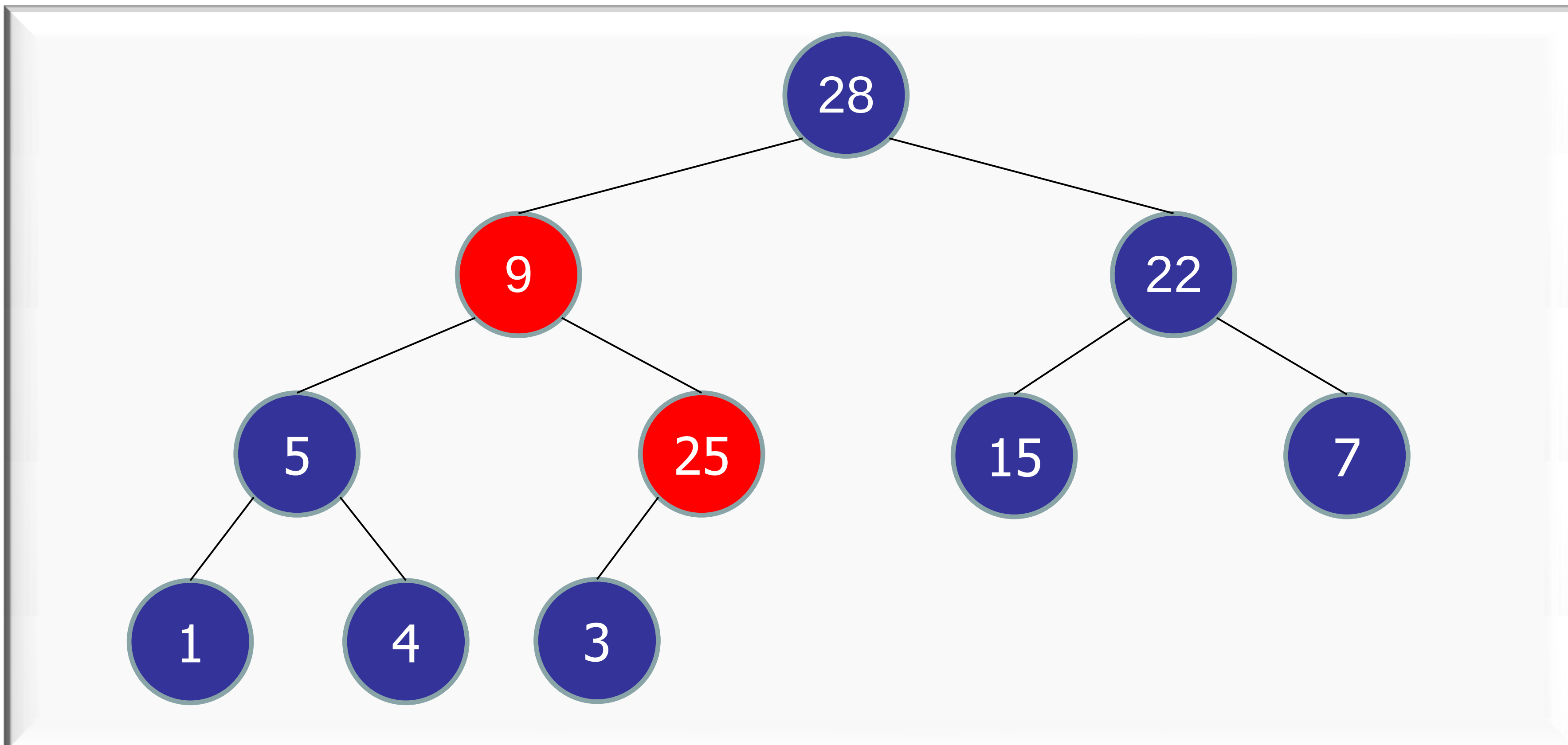
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

`insert(25)`:

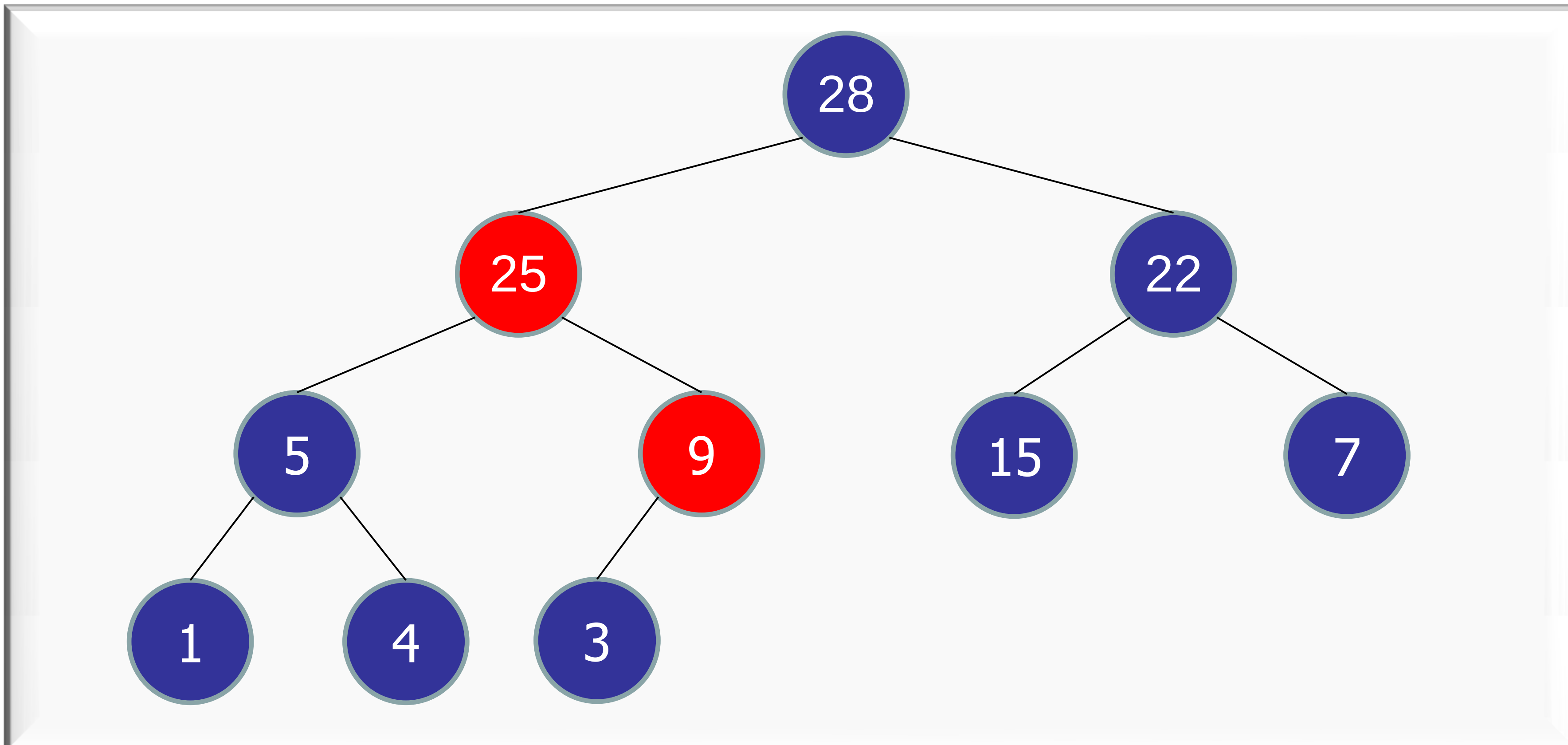
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

`insert(25)`:

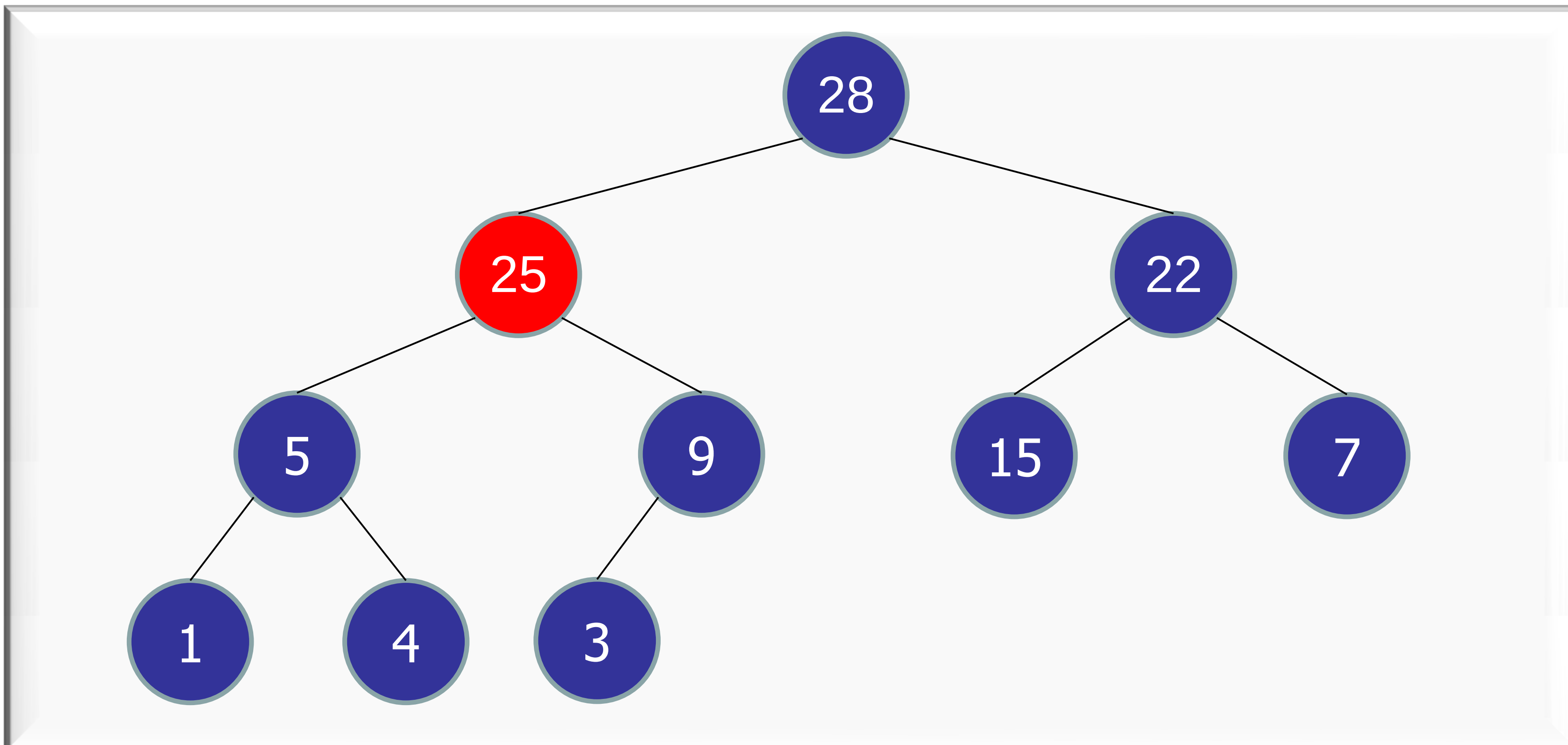
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

`insert(25)`:

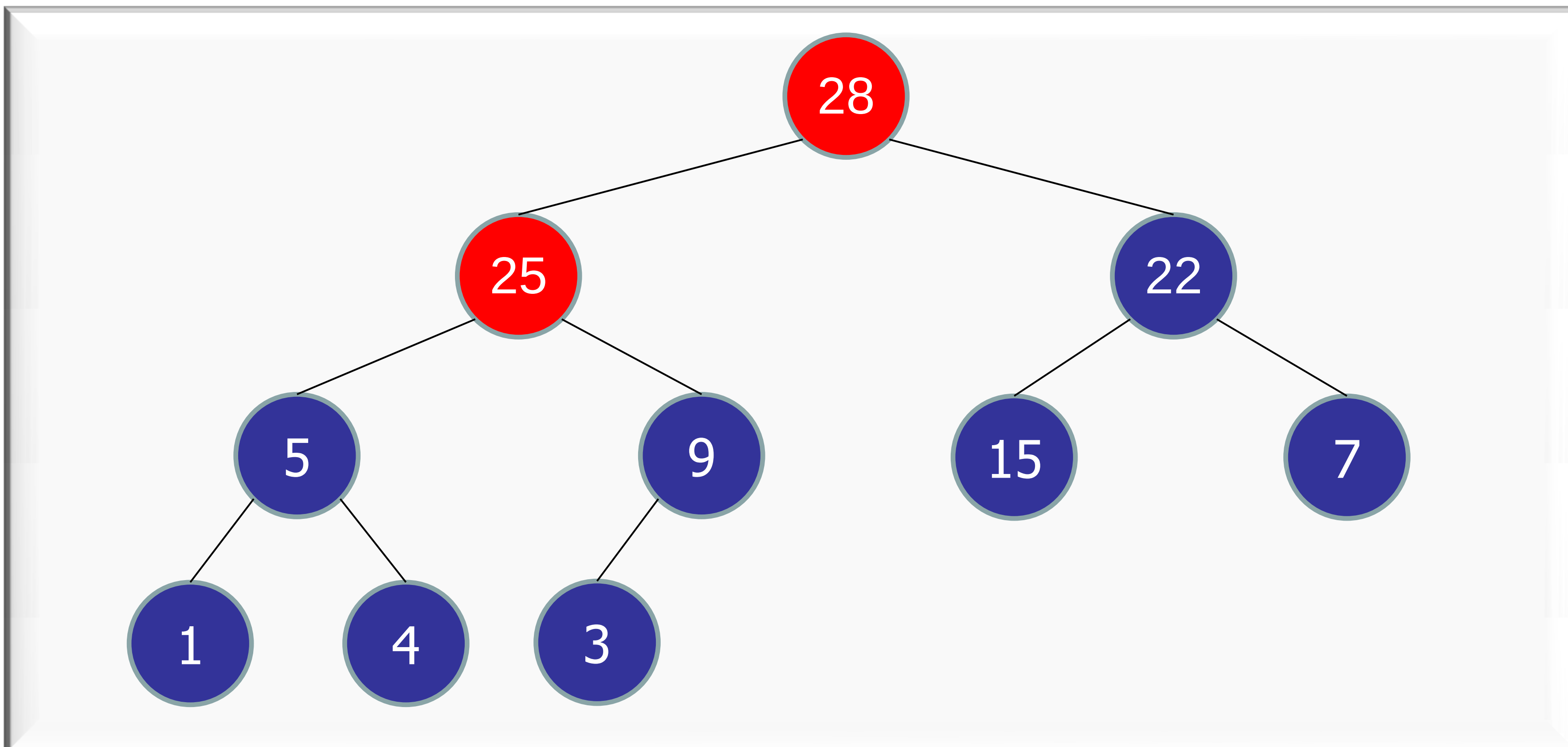
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

`insert(25)`:

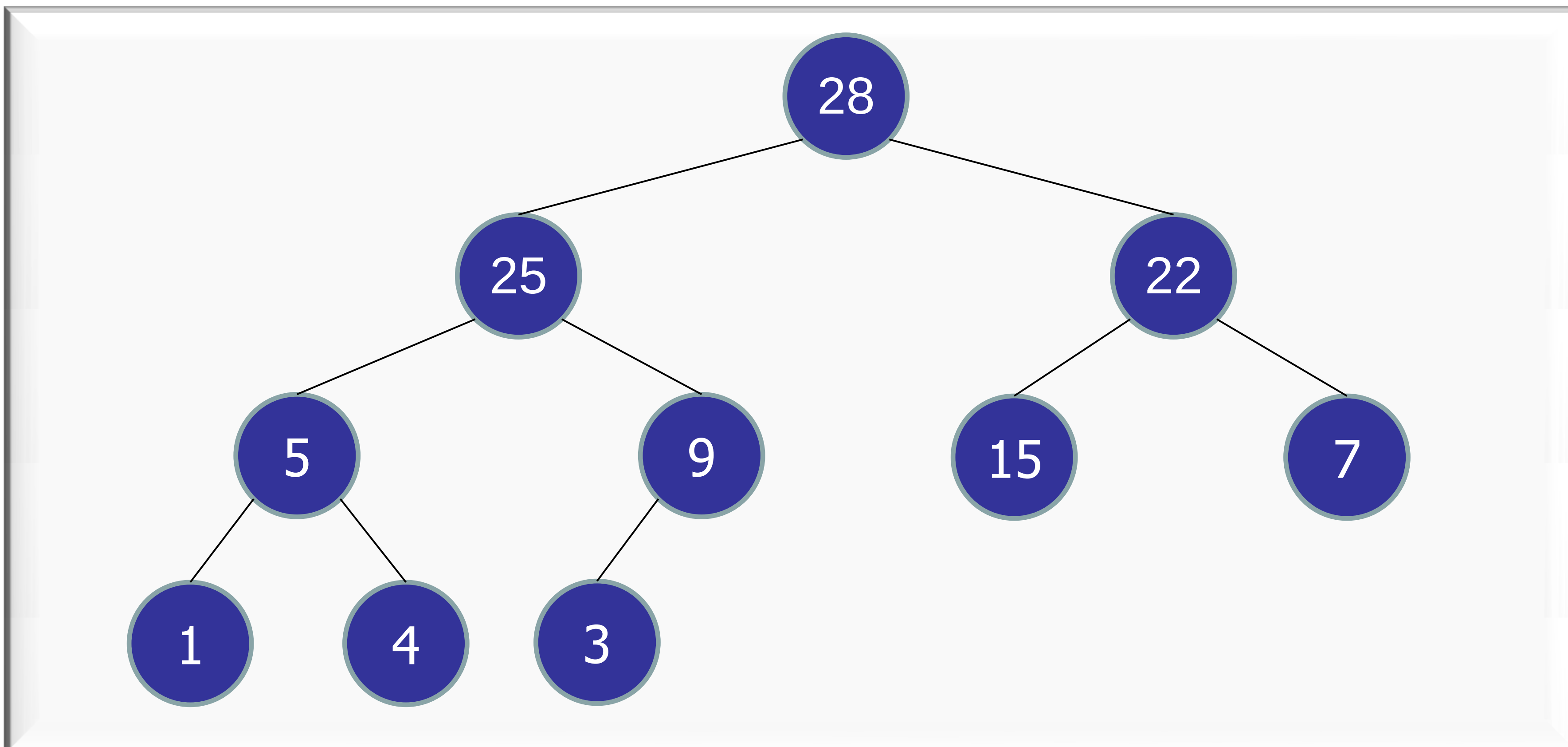
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

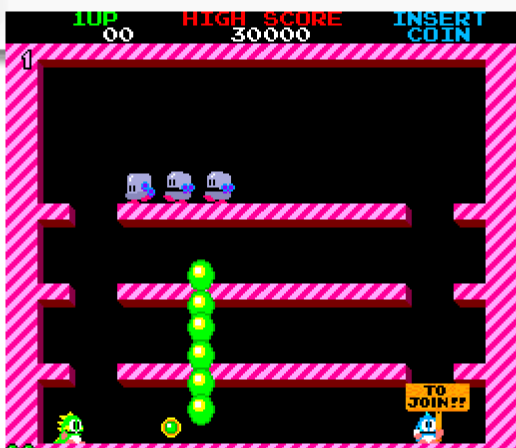
`insert(25)`:

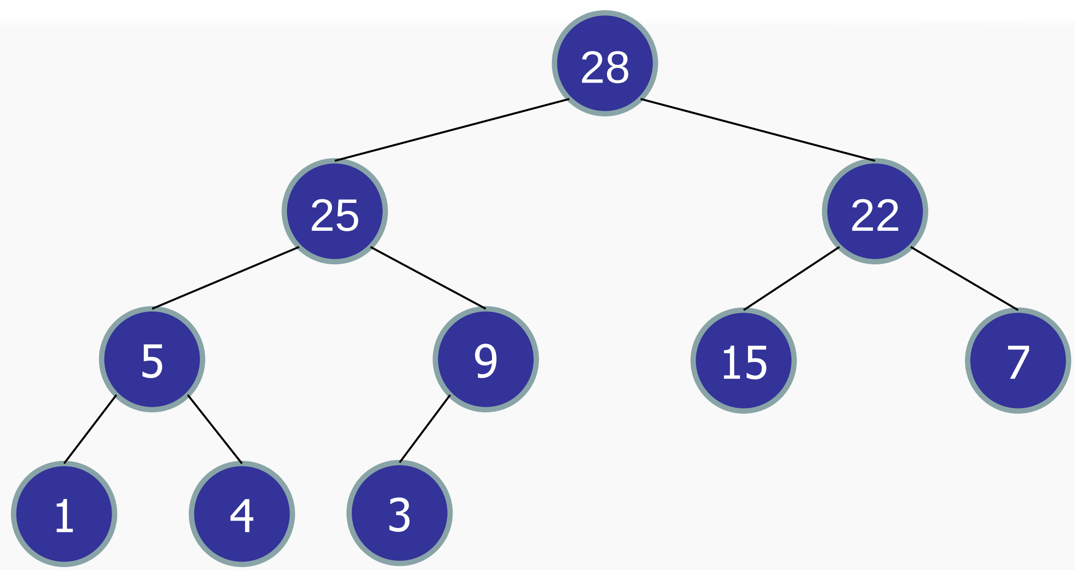
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Inserting in a Heap

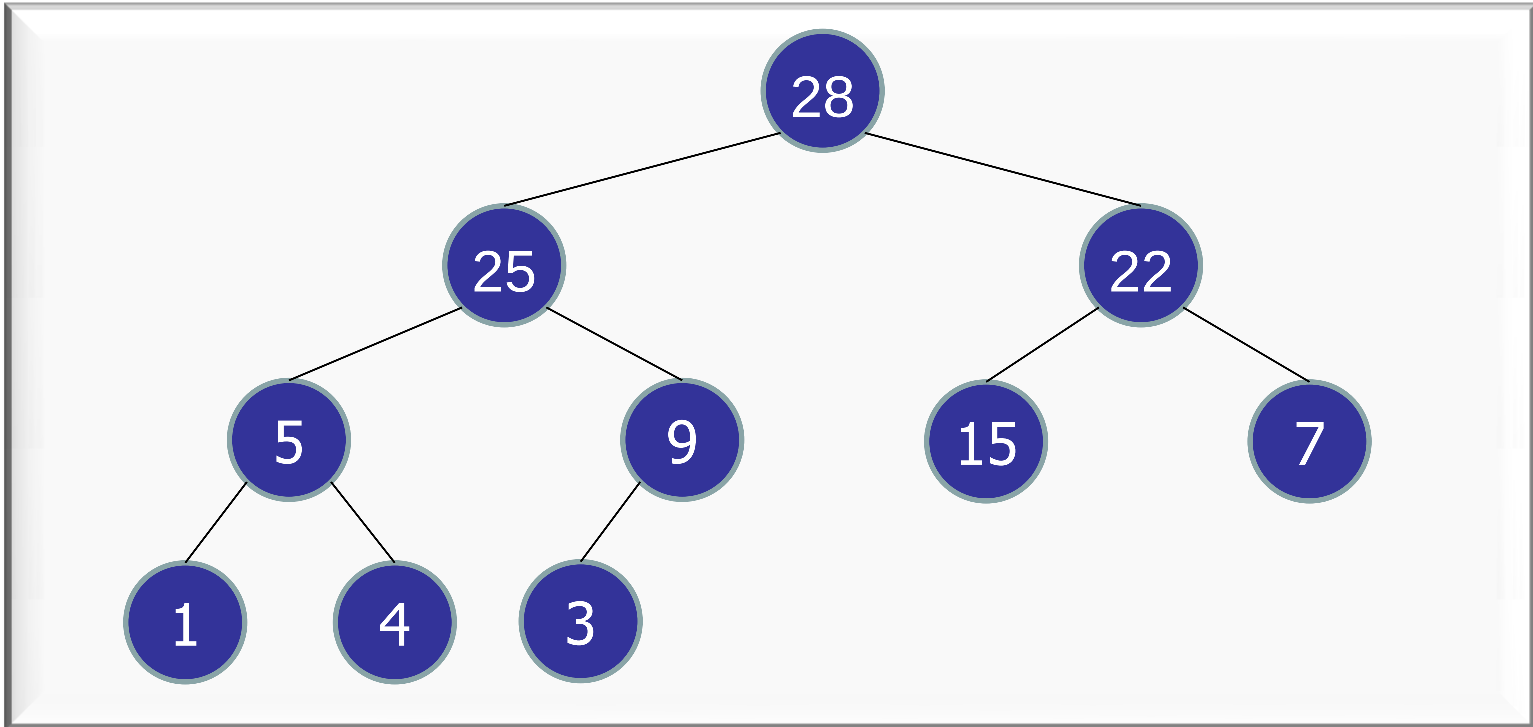
```
bubbleUp(Node v) {
  while (v != null) {
    if (priority(v) > priority(parent(v)))
        swap(v, parent(v));
    else return;
    v = parent(v);
  }
}
```





Inserting in a Heap

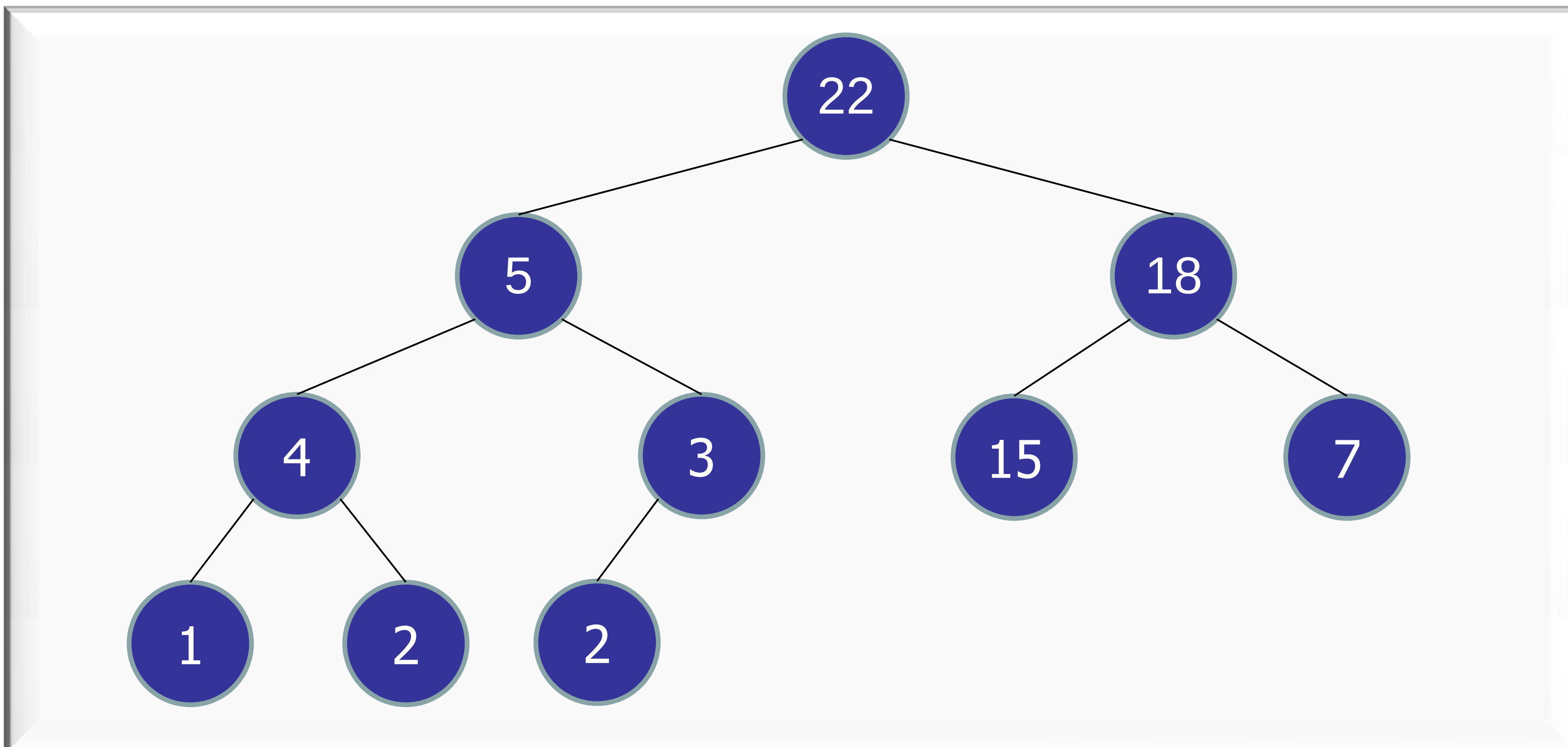
```
insert(Priority p, Key k) {
  Node v = m_completeTree.insert(p,k);
  bubbleUp(v);
}
```



Inserting in a Heap

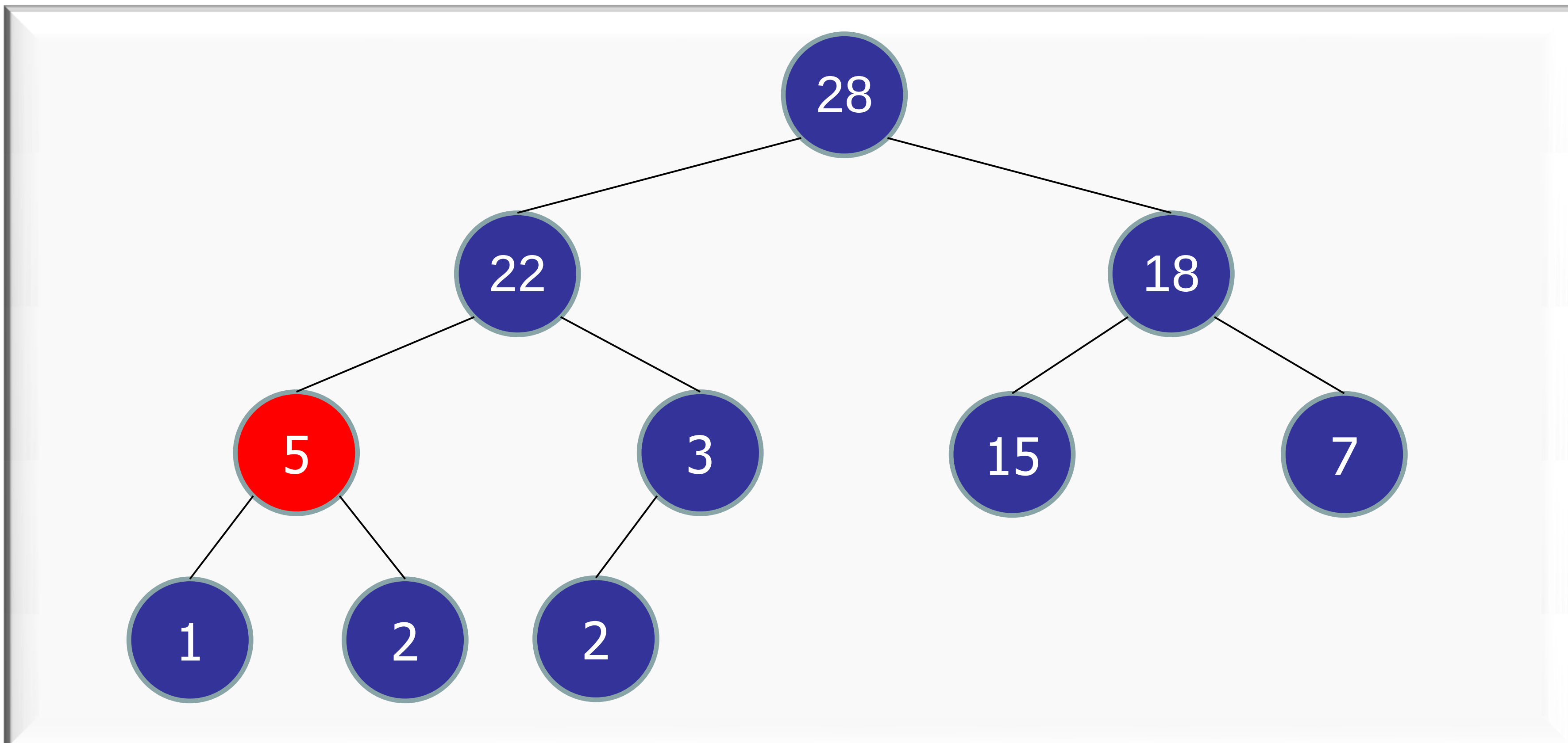
`insert(...)` :

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



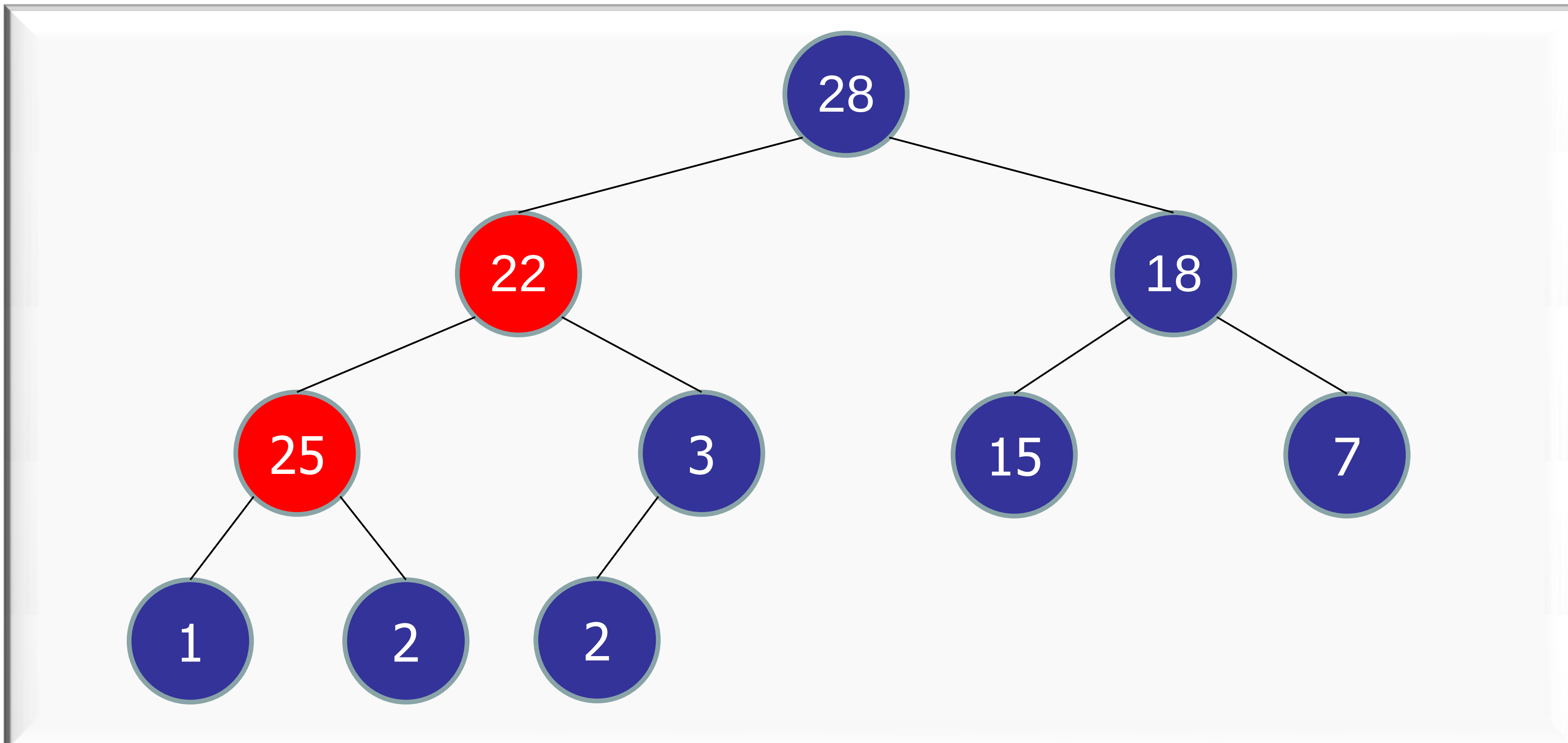
Inserting in a Heap

`increaseKey(5 → 25):`



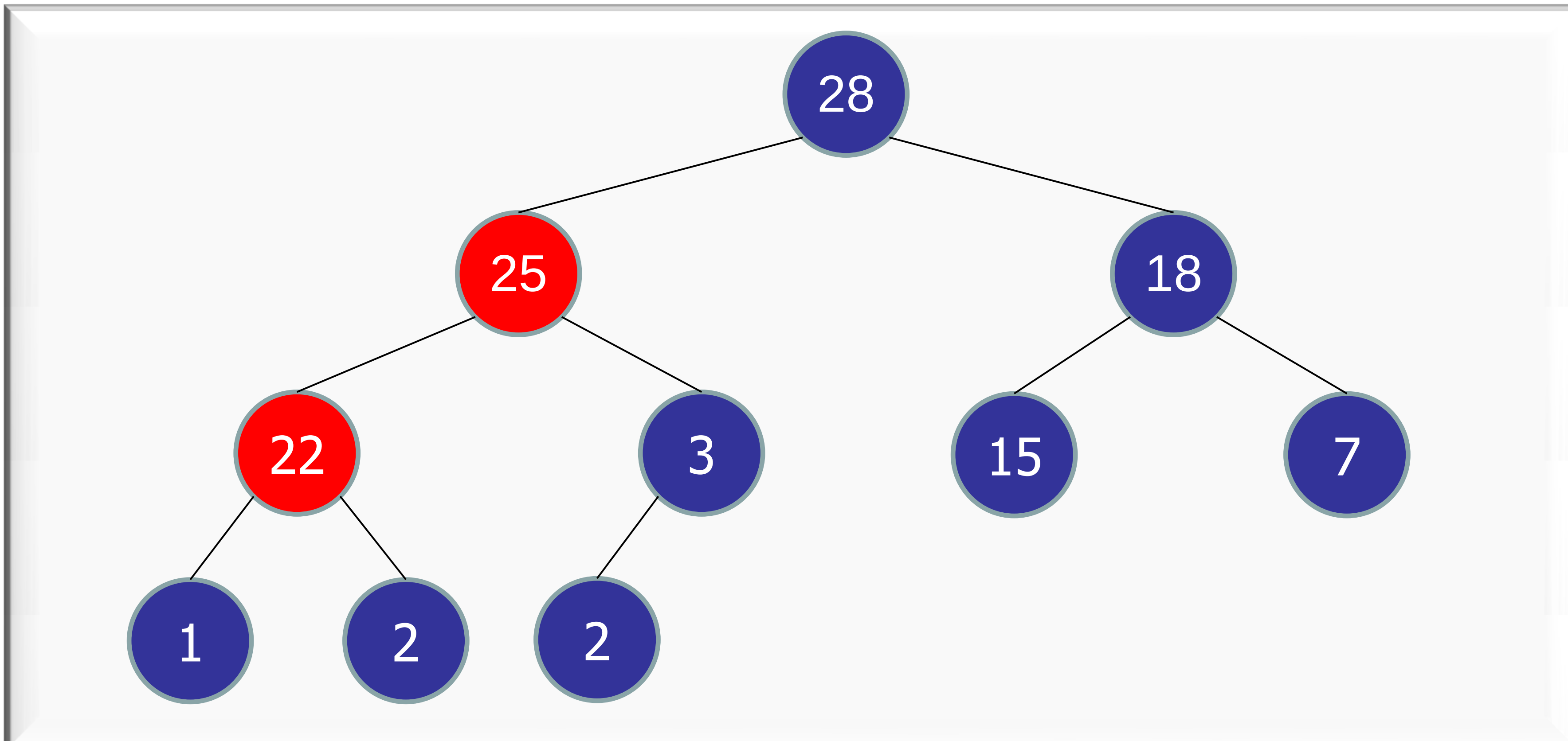
Inserting in a Heap

`increaseKey(5 → 25)`: bubbleUp(25)



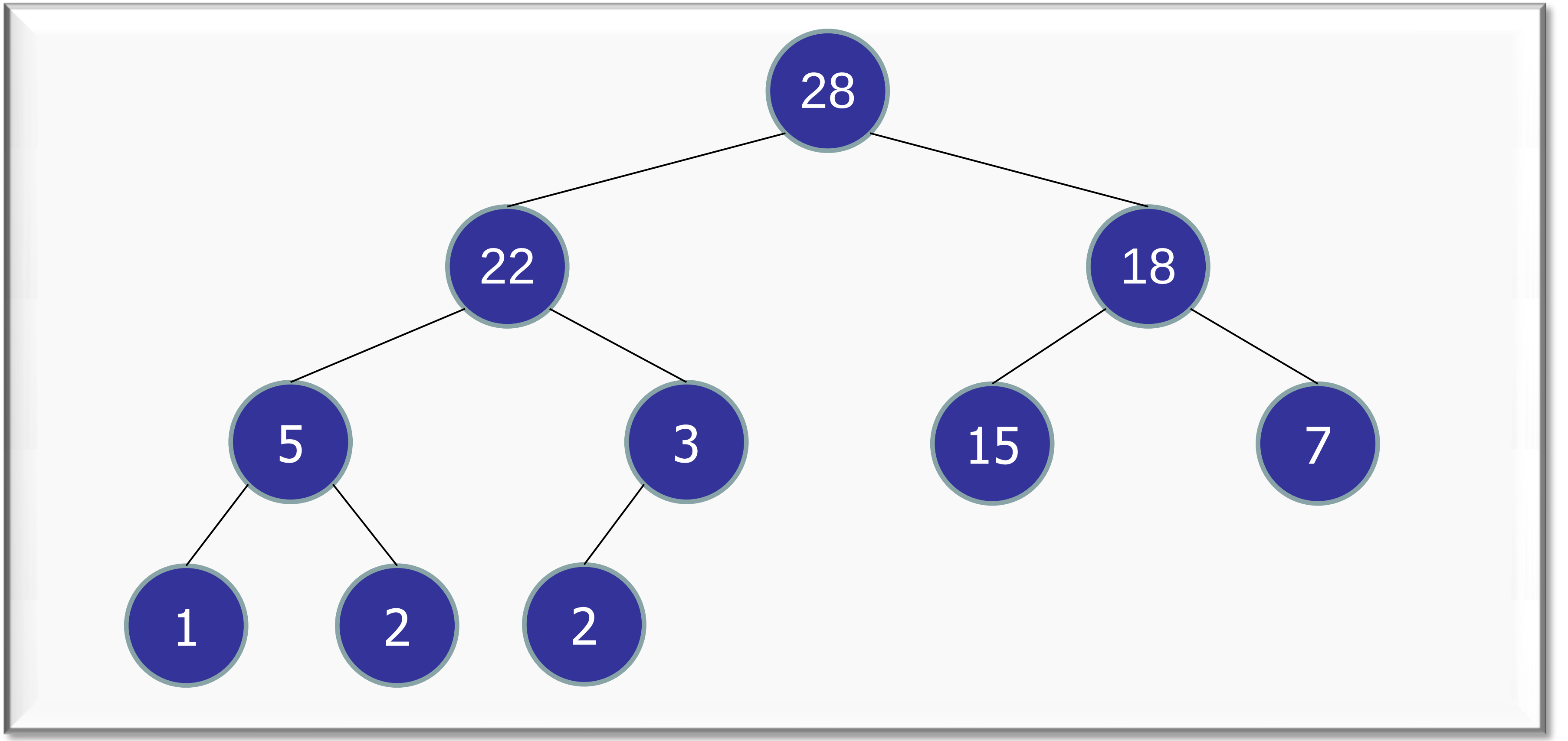
Inserting in a Heap

`increaseKey(5 → 25)`: bubbleUp(25)



Inserting in a Heap

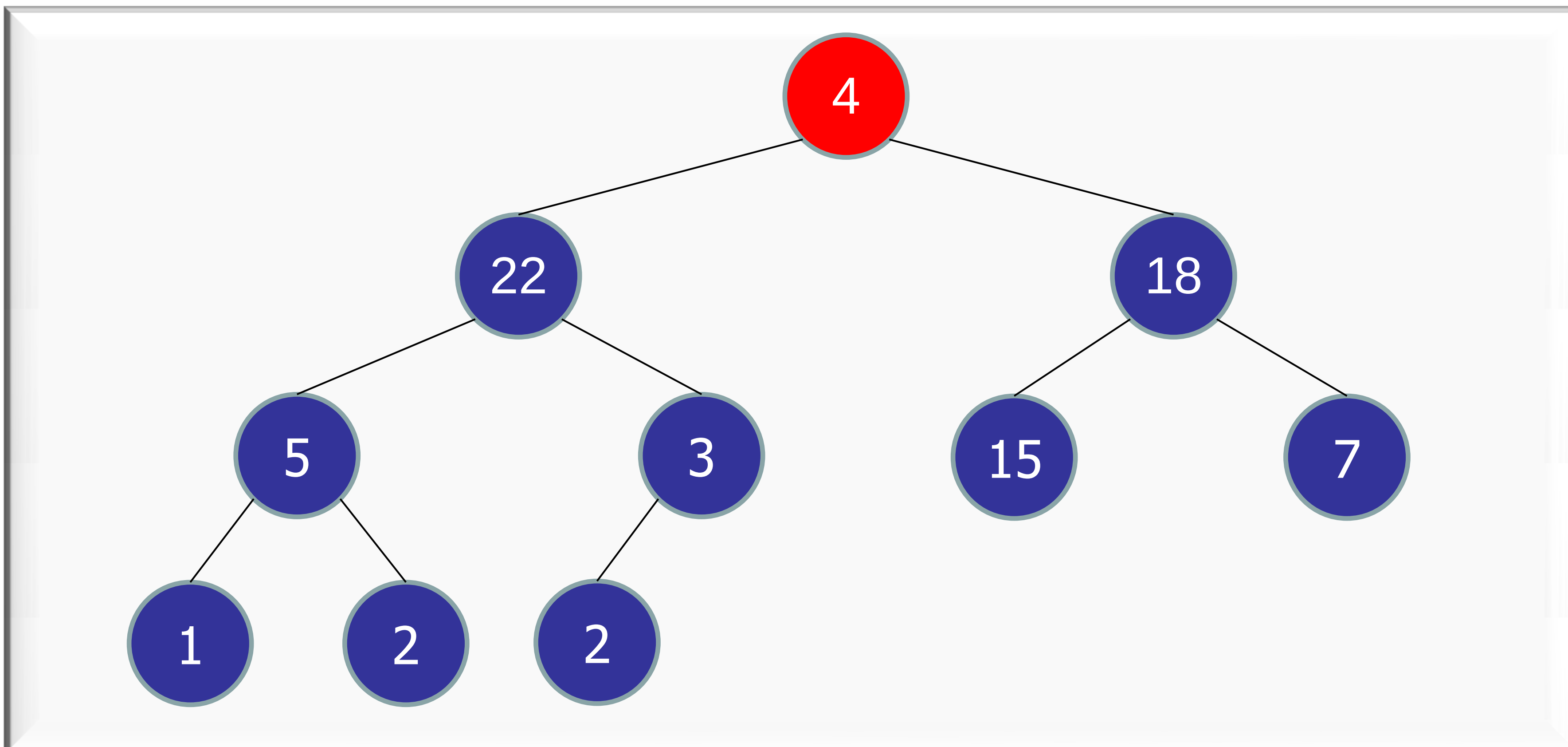
`decreaseKey(28 → 4):`



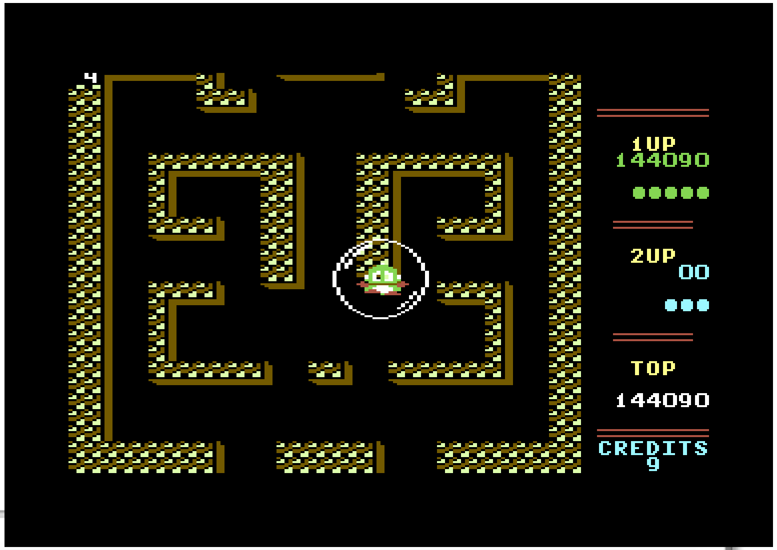
Inserting in a Heap

`decreaseKey(28 → 4)`:

- Step 1: Update the priority

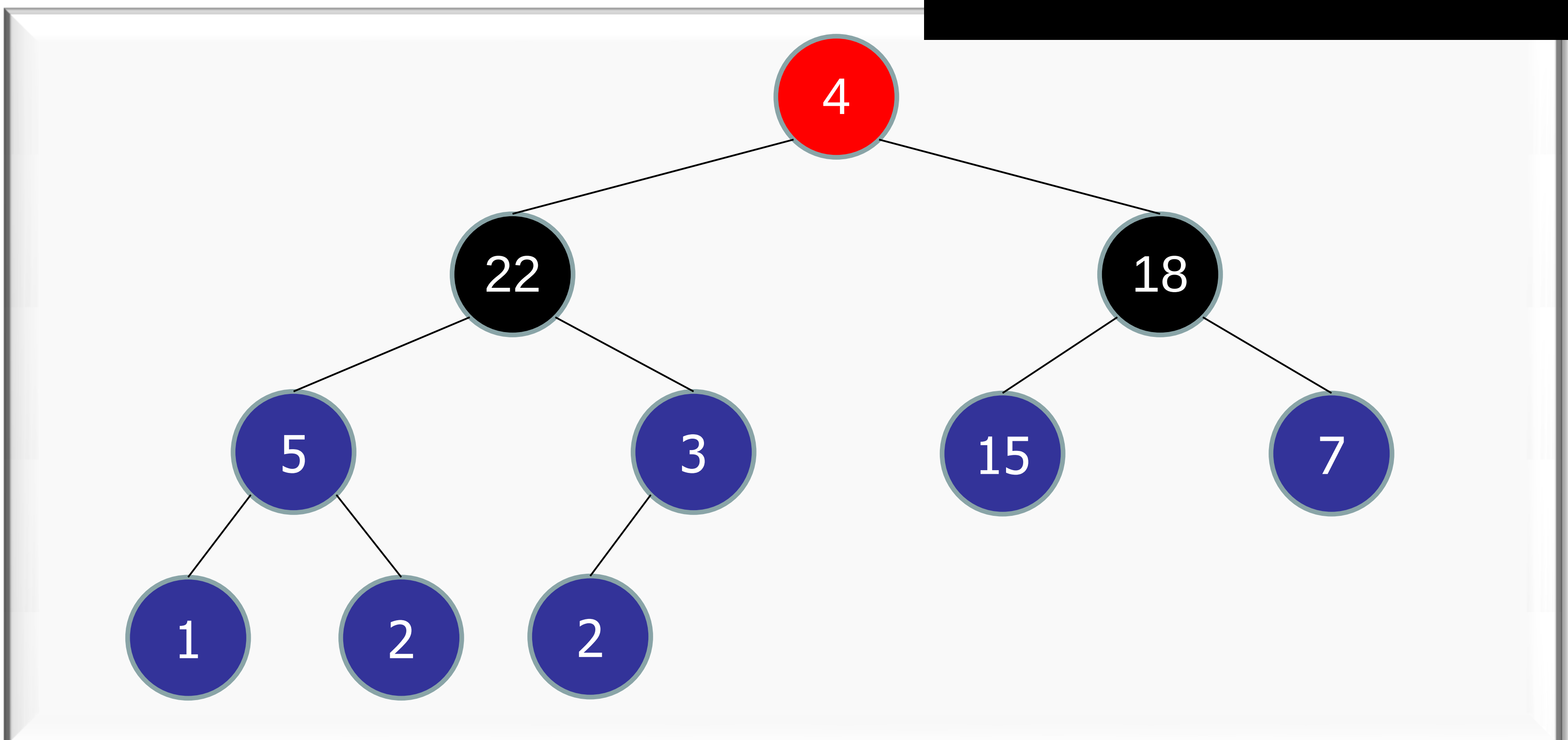


Inserting in a Heap



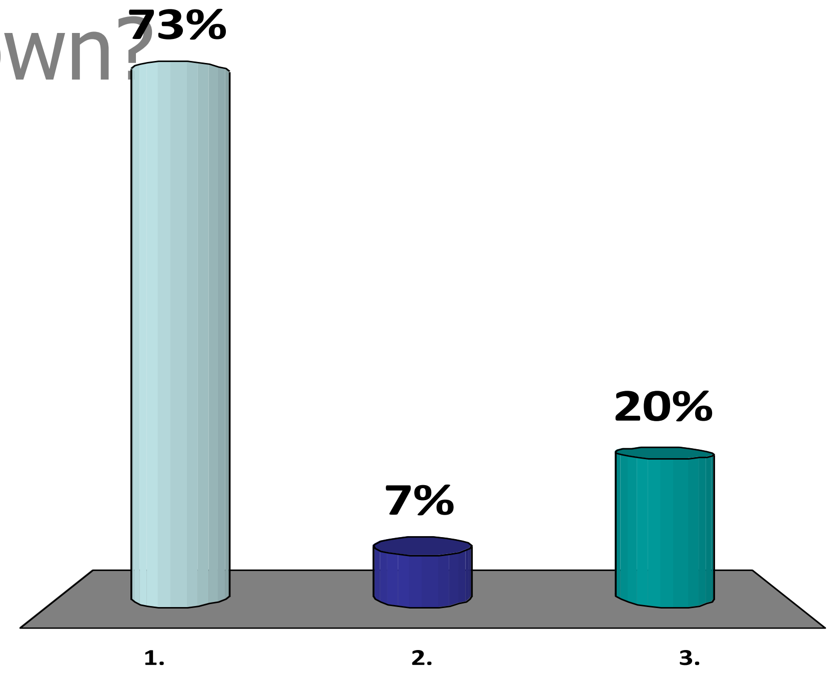
`decreaseKey(28 → 4):`

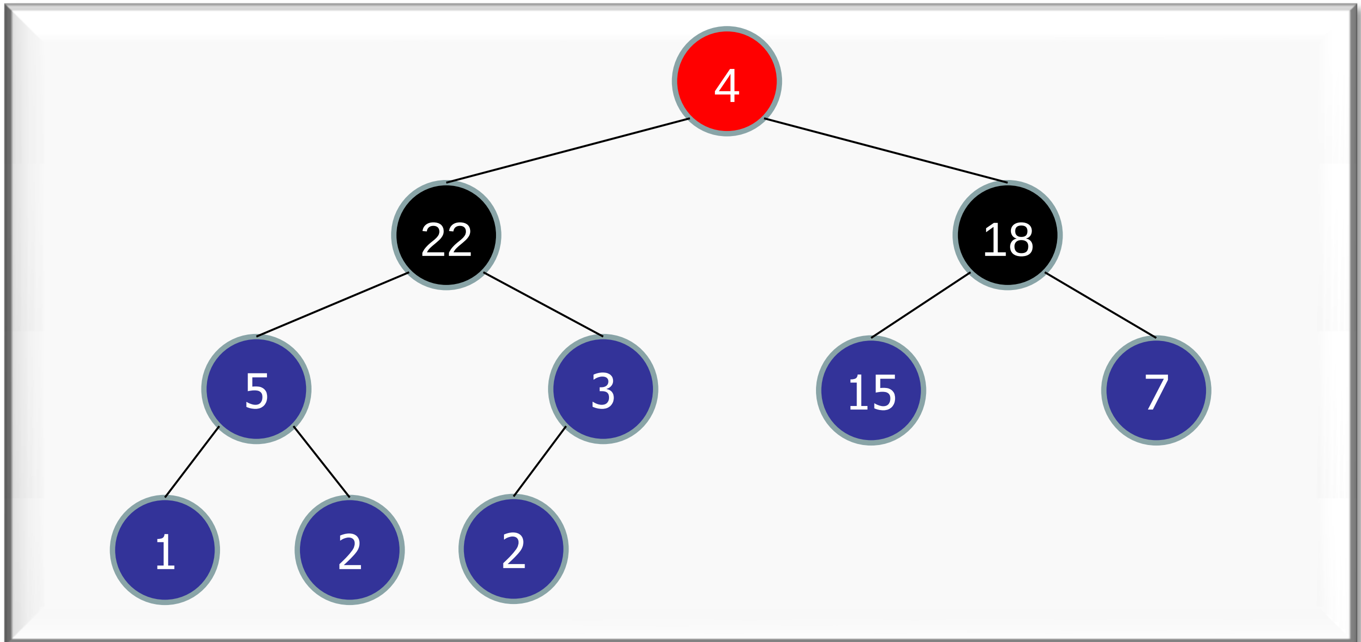
- Step 1: Update the priority
- Step 2: bubbleDown(4)



Which way to bubbleDown?

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

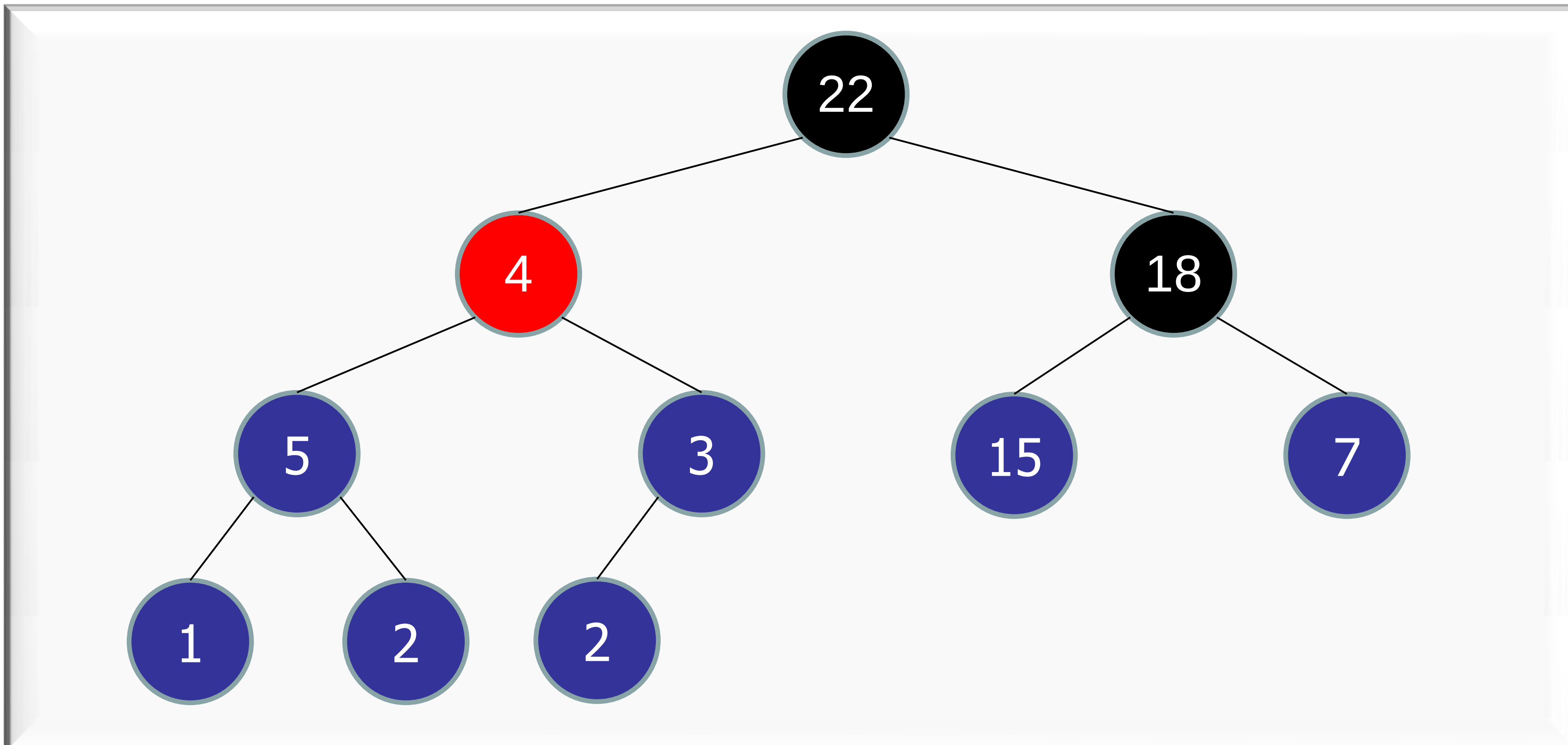




Inserting in a Heap

`decreaseKey(28 → 4)` :

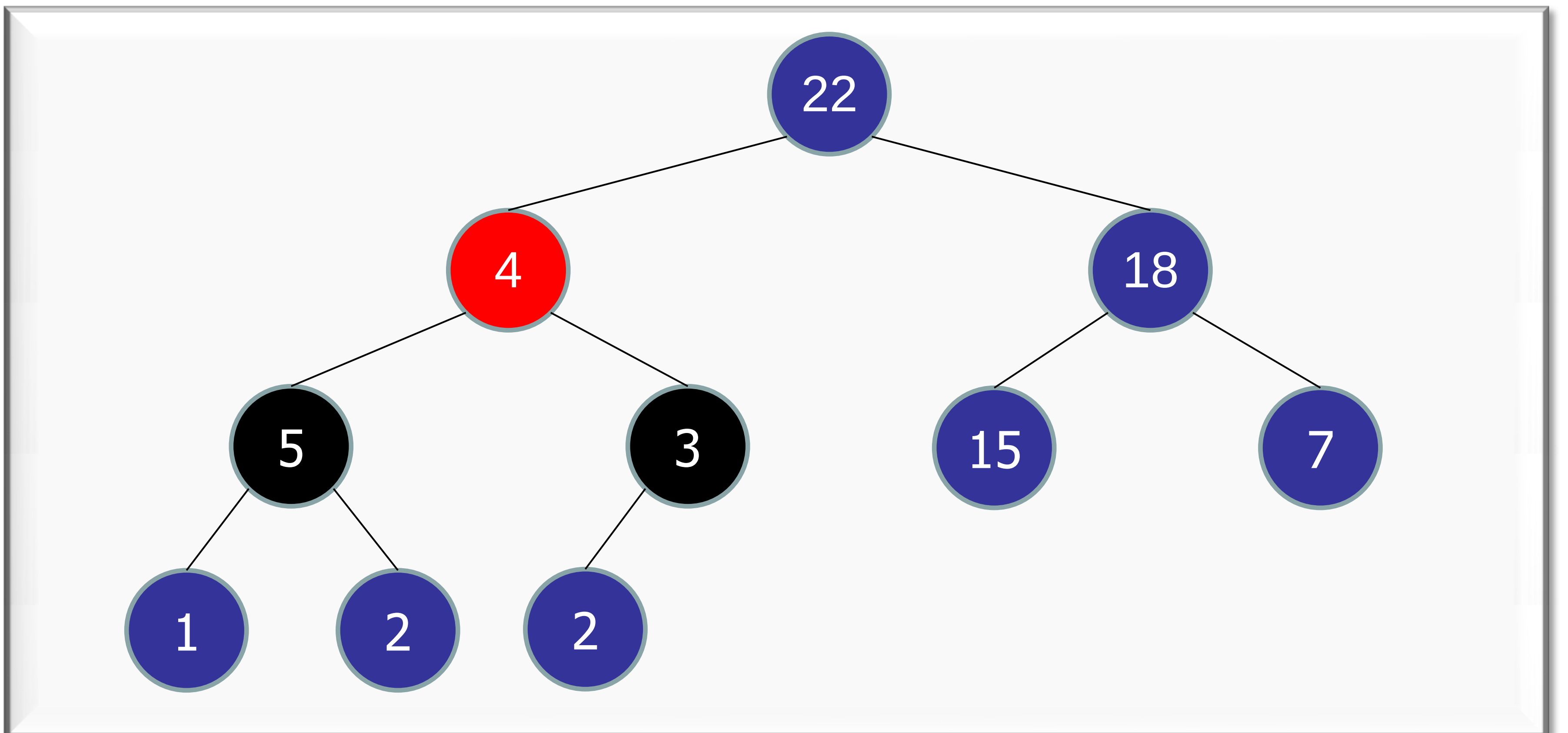
- Step 1: Update the priority
- Step 2: bubbleDown(4)



Inserting in a Heap

`decreaseKey(28 → 4)` :

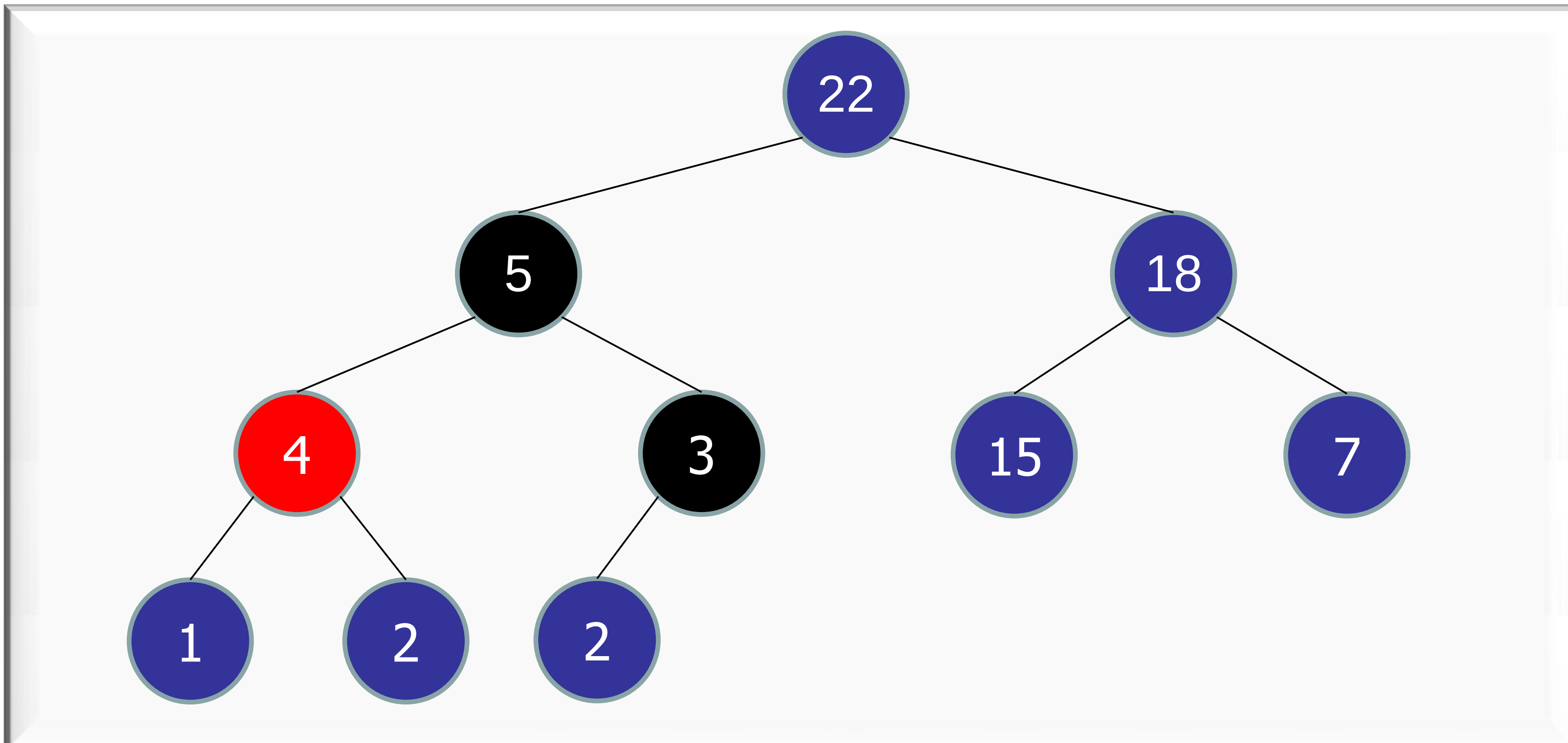
- Step 1: Update the priority
- Step 2: bubbleDown(4)



Inserting in a Heap

`decreaseKey(28 → 4)` :

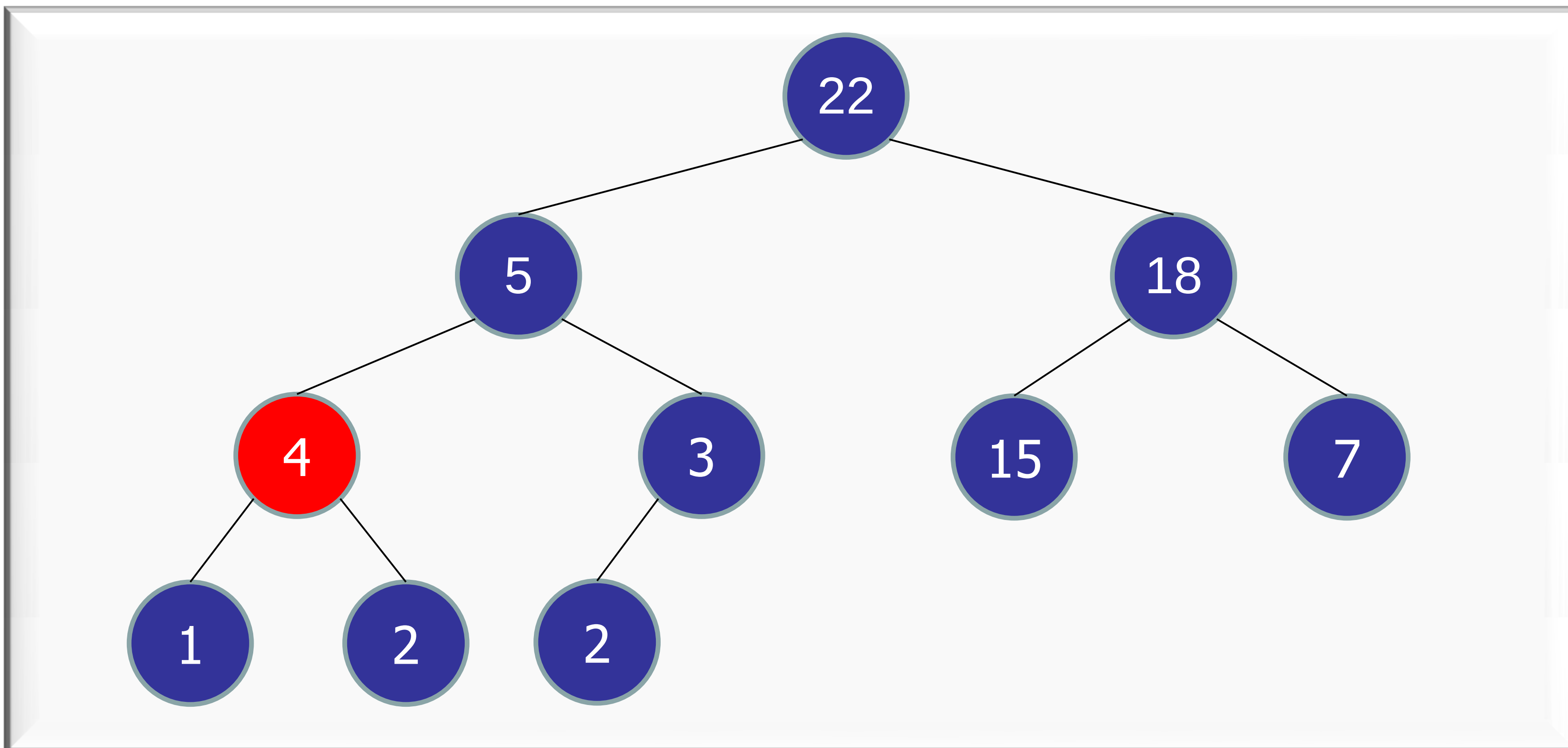
- Step 1: Update the priority
- Step 2: bubbleDown(4)



Inserting in a Heap

`decreaseKey(28 → 4)` :

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



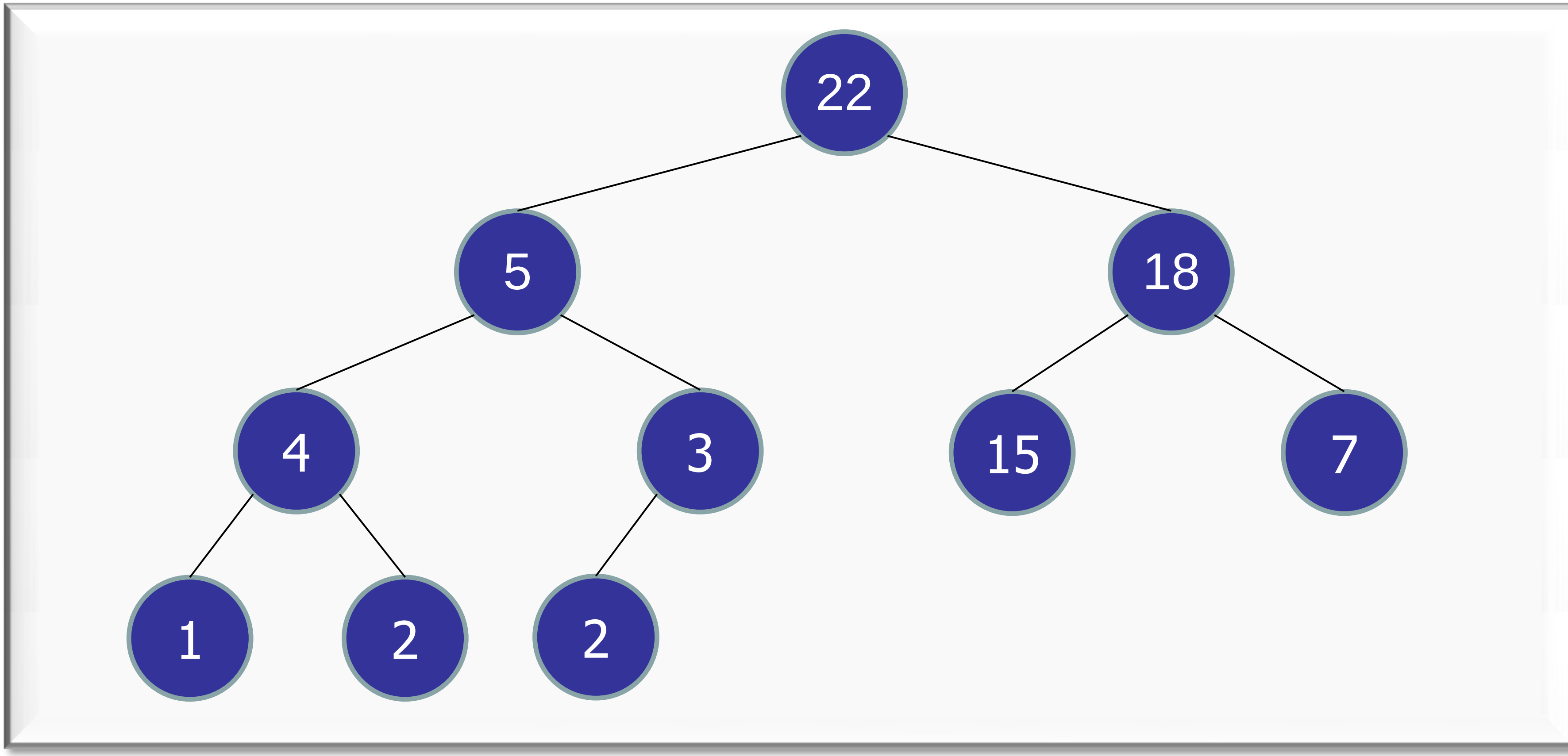
Inserting in a Heap

```
bubbleDown(Node v)
  while (!leaf(v)) {
    leftP = priority(left(v));
    rightP = priority(right(v));
    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

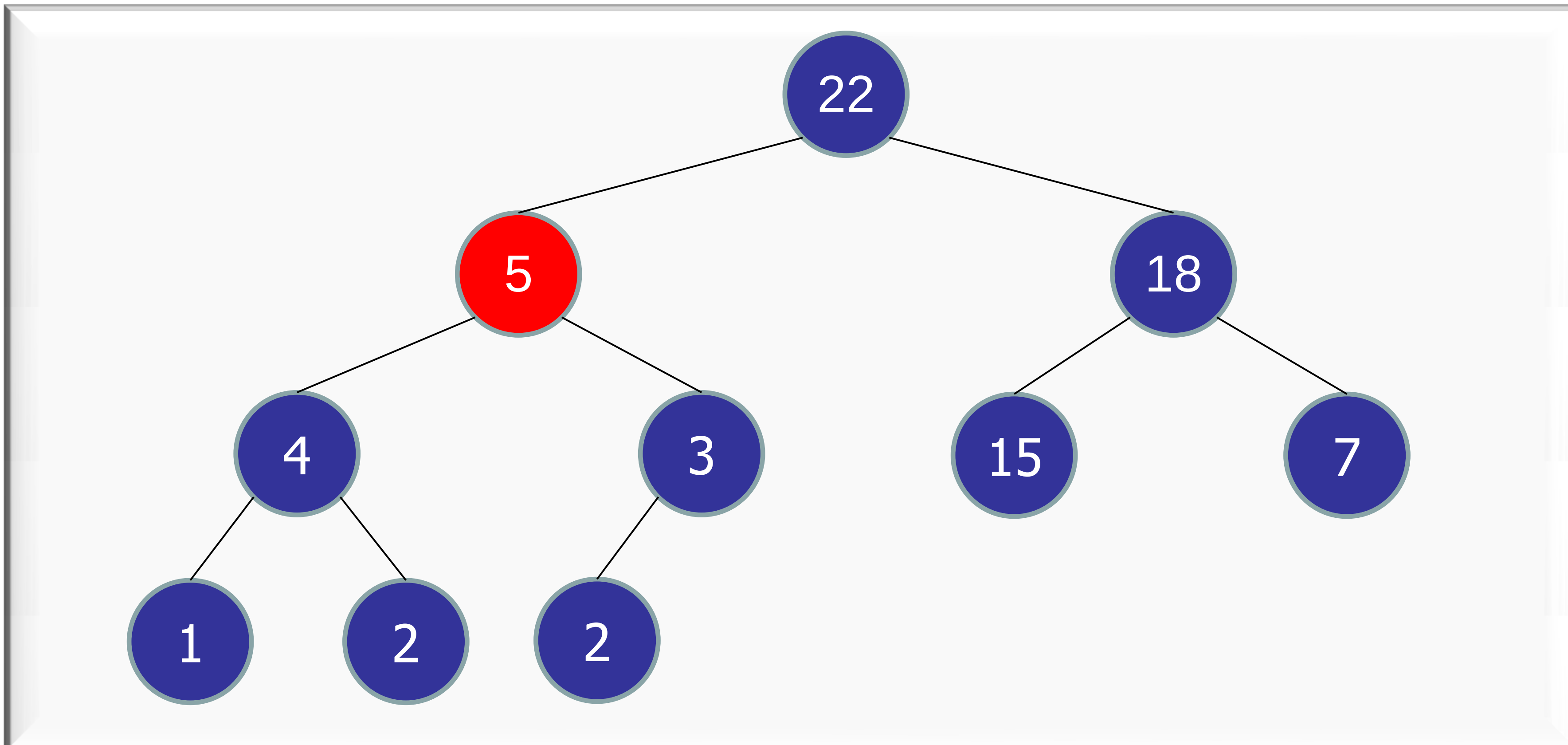
`decreaseKey(. . .)` :

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



Inserting in a Heap

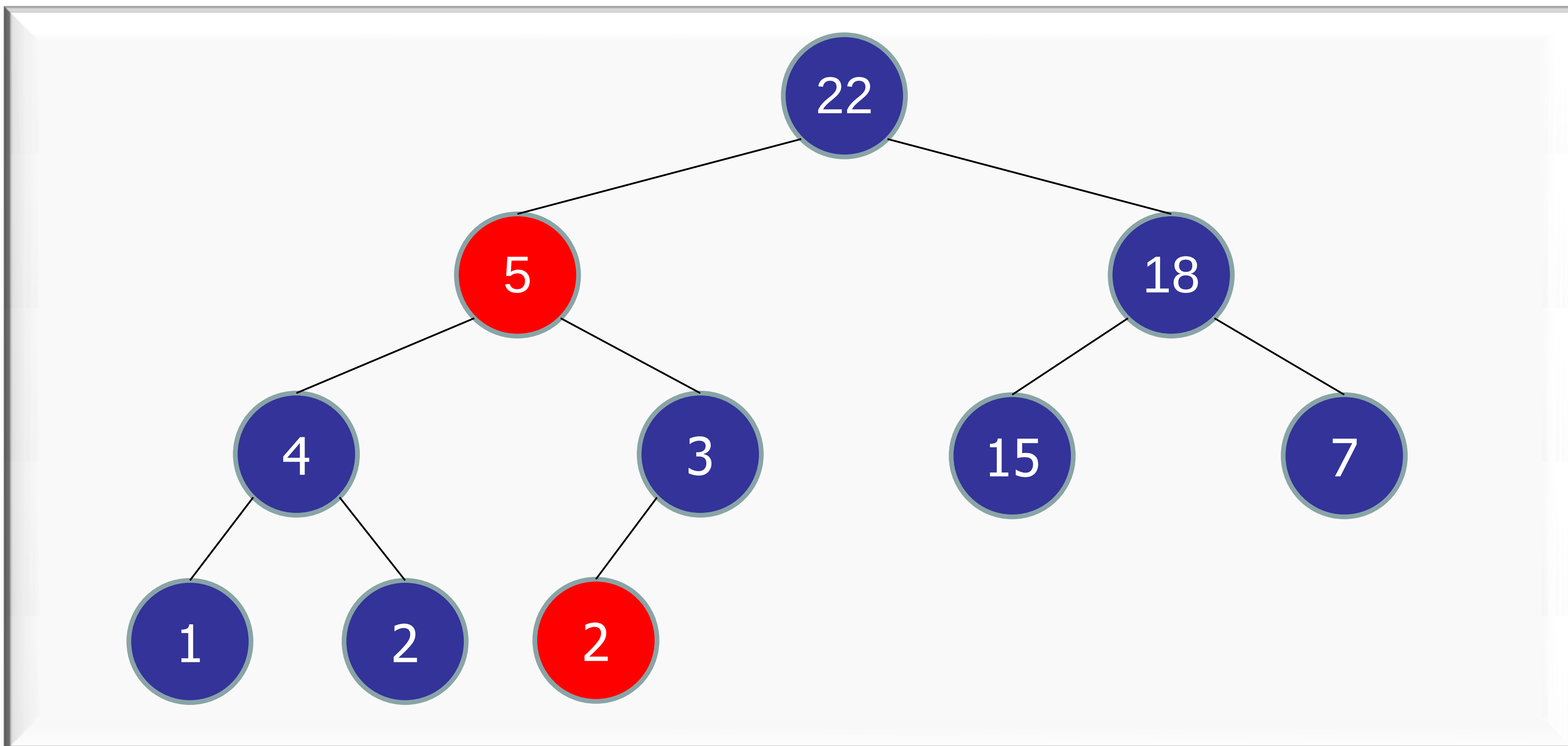
`delete(5)` :



Inserting in a Heap

`delete(5)` :

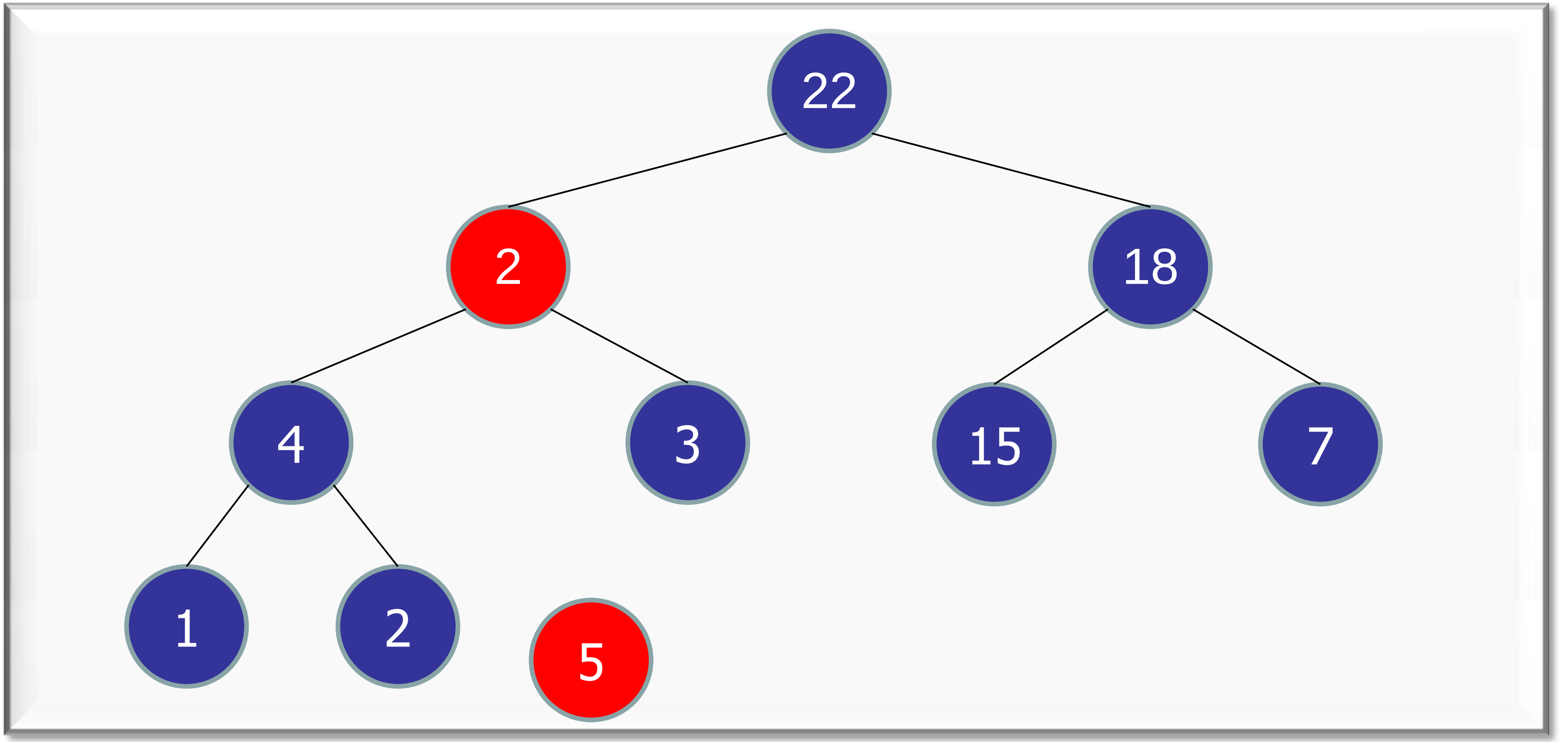
- `swap(5, last())`



Inserting in a Heap

`delete(5)` :

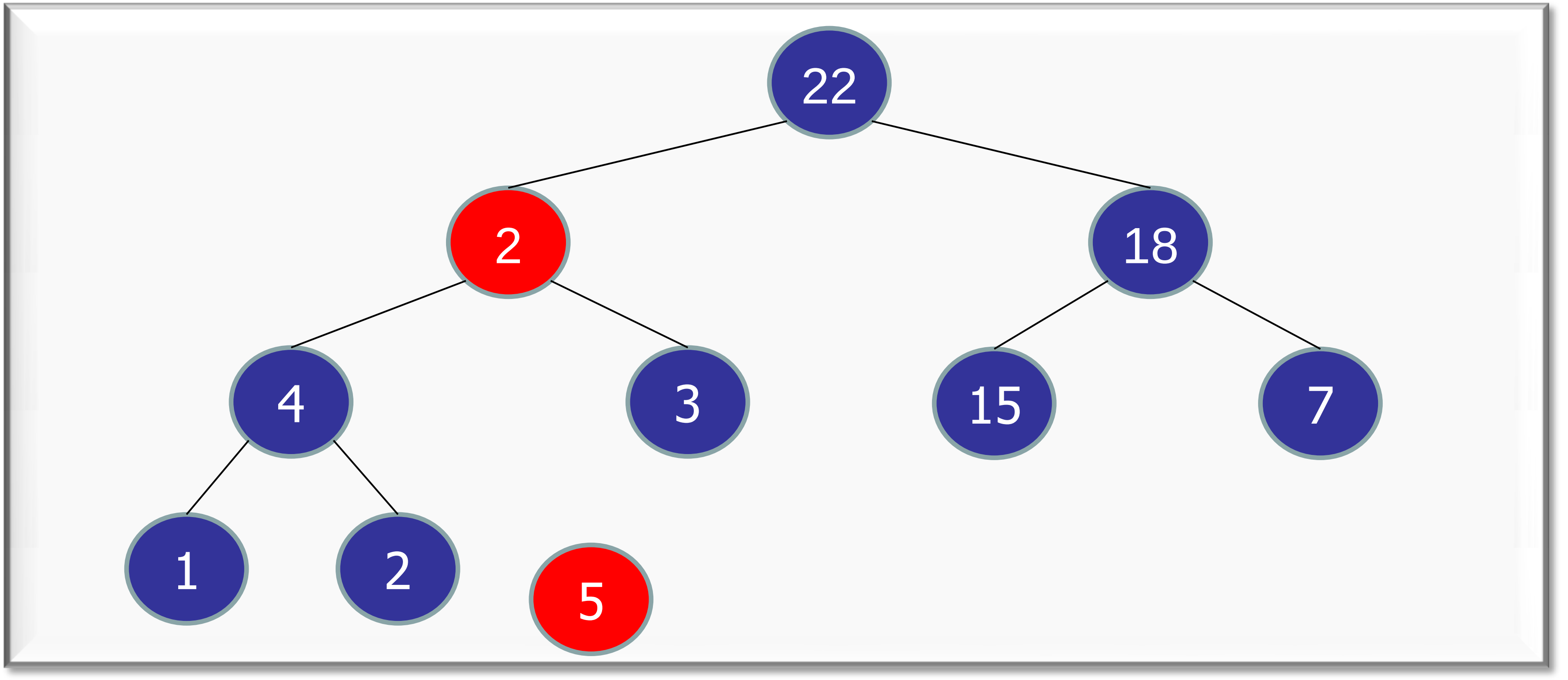
- `swap(5, last())`
- `remove(last())`



Inserting in a Heap

`delete(5)` :

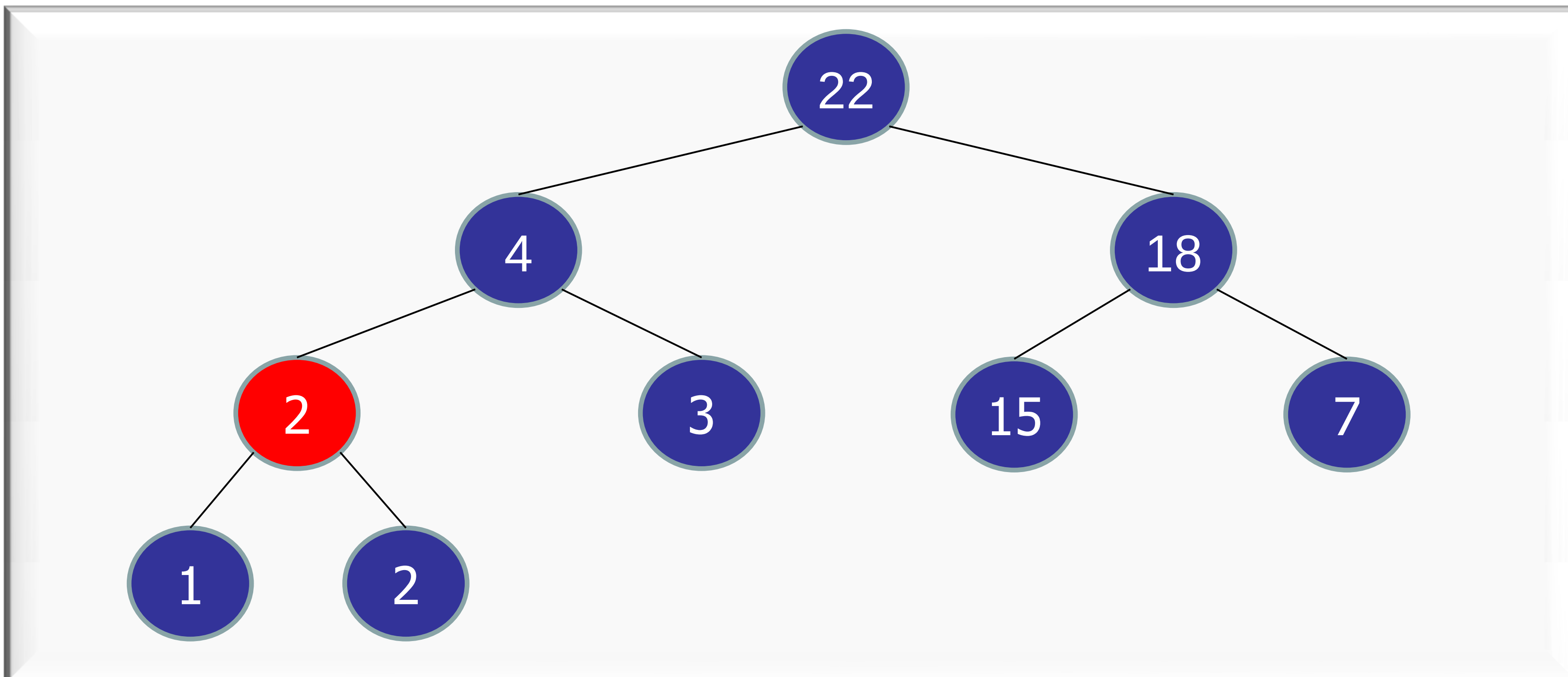
- `swap(5, last())`
- `remove(last())`
- `bubbleDown(2) // depending on if last() > deleted`



Inserting in a Heap

`delete(5)` :

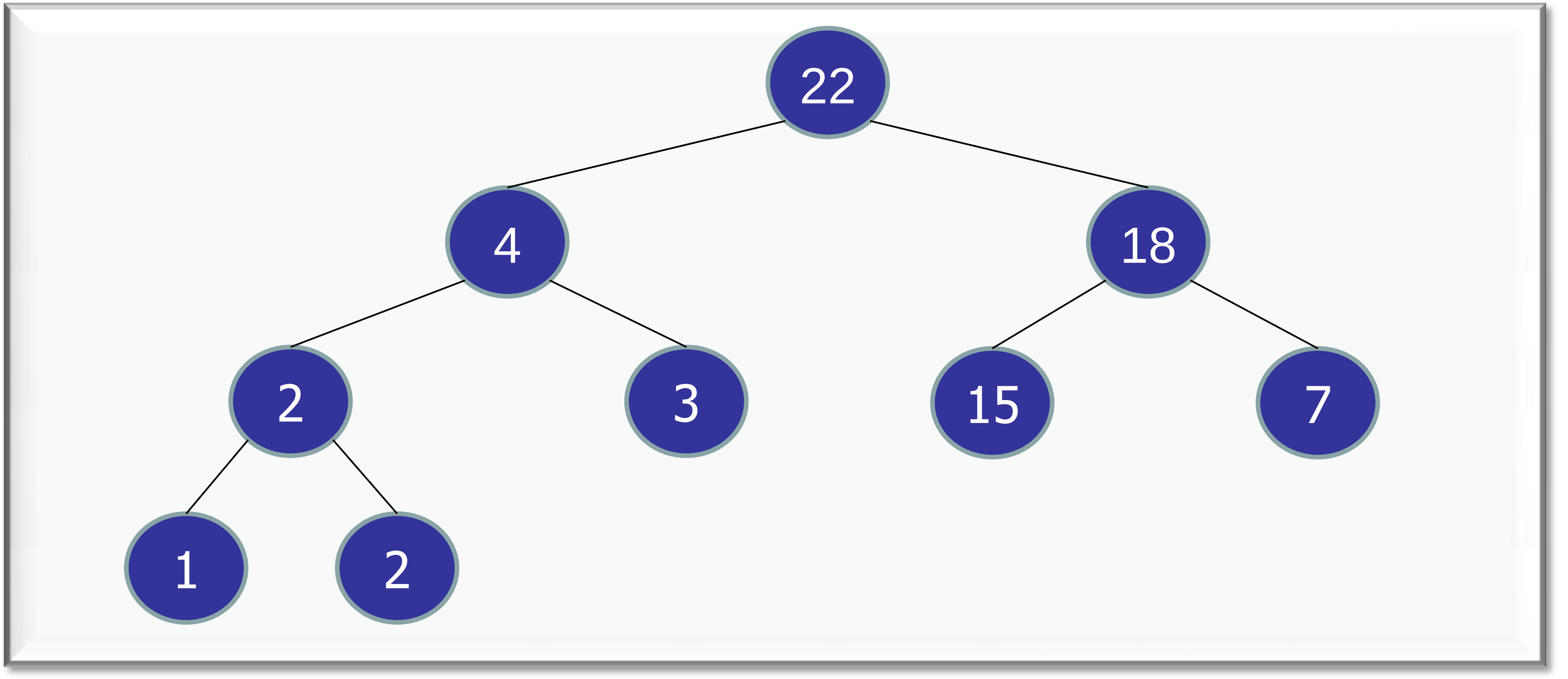
- `swap(5, last())`
- `remove(last())`
- `bubbleDown(2)`



Inserting in a Heap

`delete(5)` :

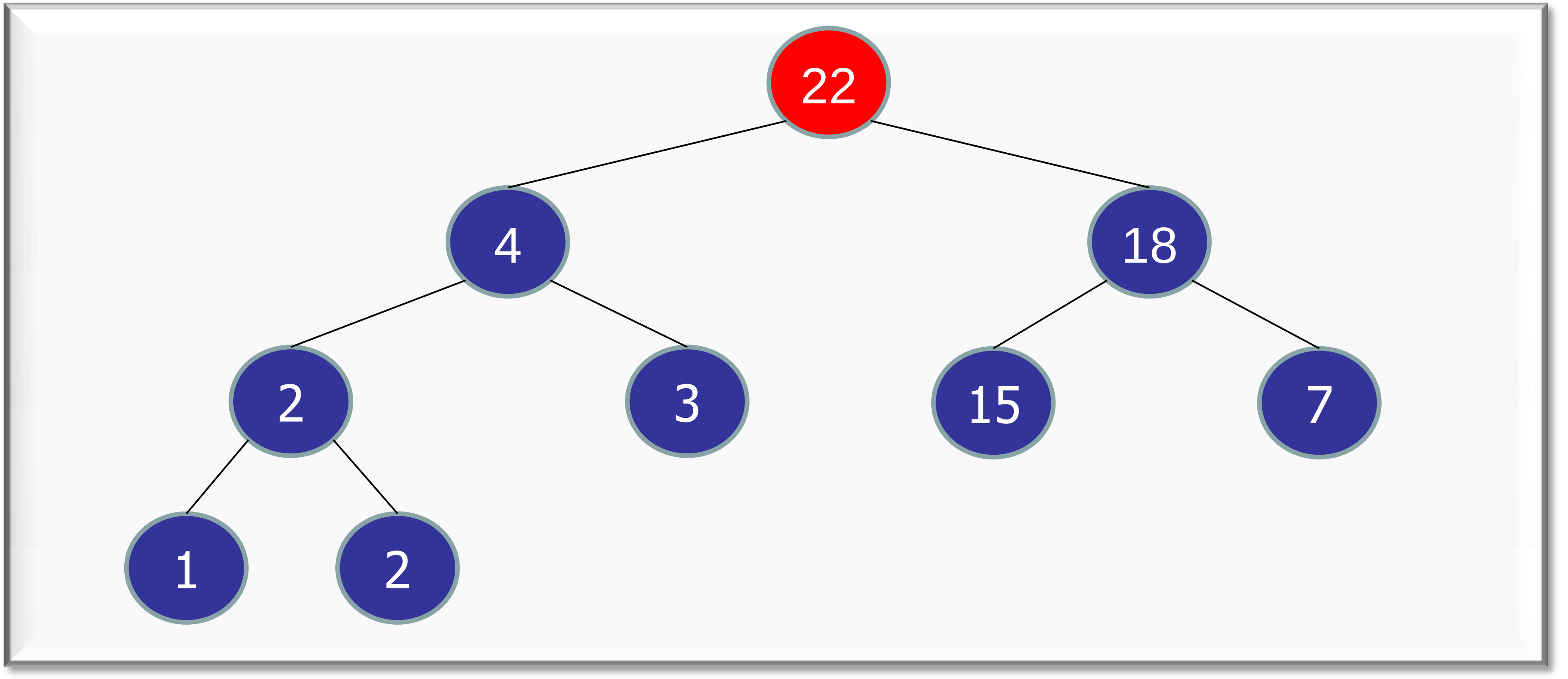
- `swap(5, last())`
- `remove(last())`
- `bubbleDown(2)`



Inserting in a Heap

`extractMax()` :

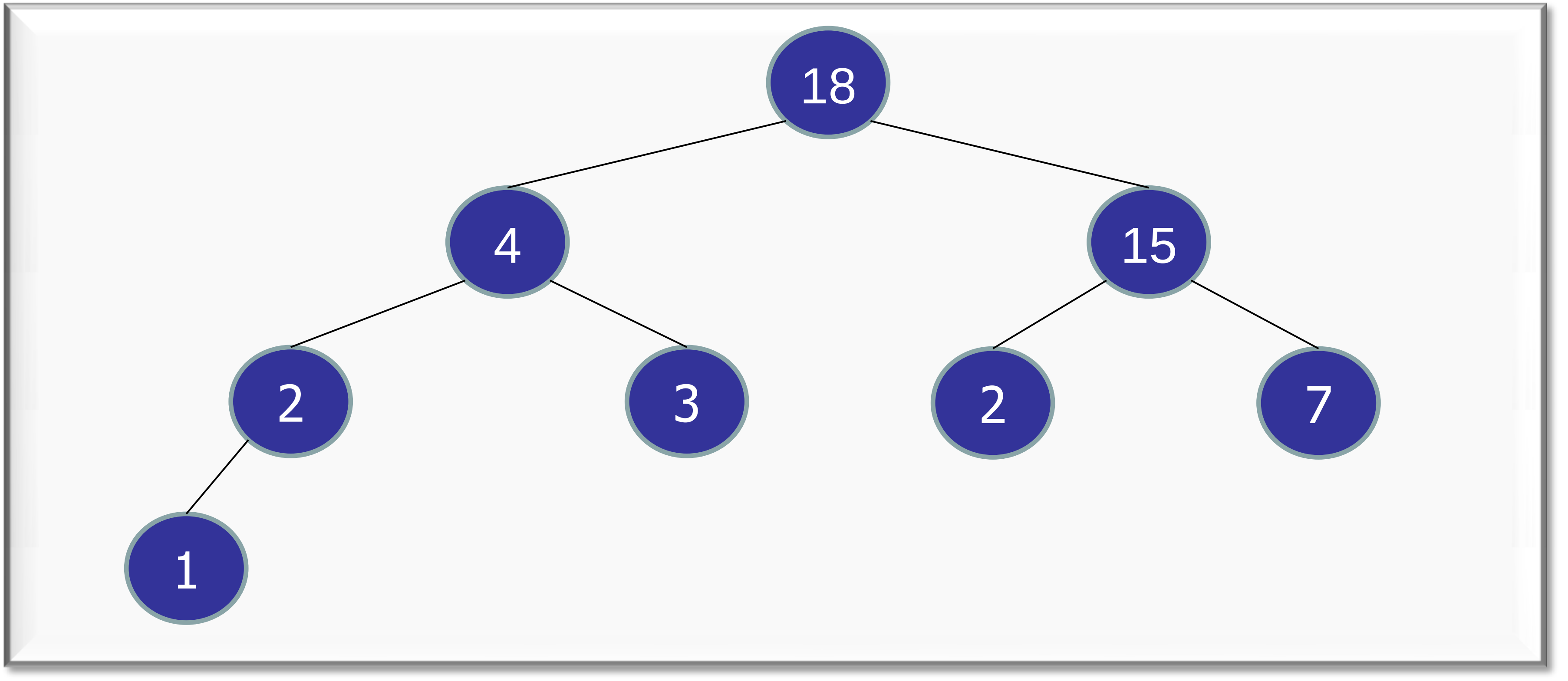
- `Node v = root;`
- `delete(root);`



Inserting in a Heap

`extractMax()` :

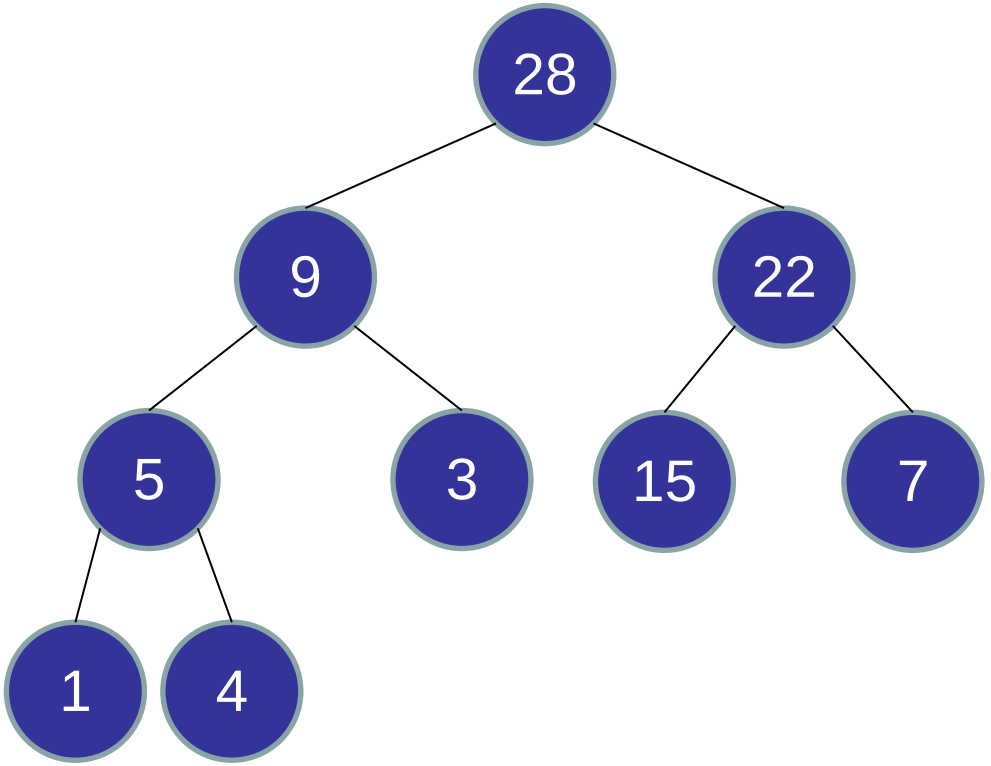
- `Node v = root;`
- `delete(root);`



(Max) Priority Queue

Heap Operations: O(log n)

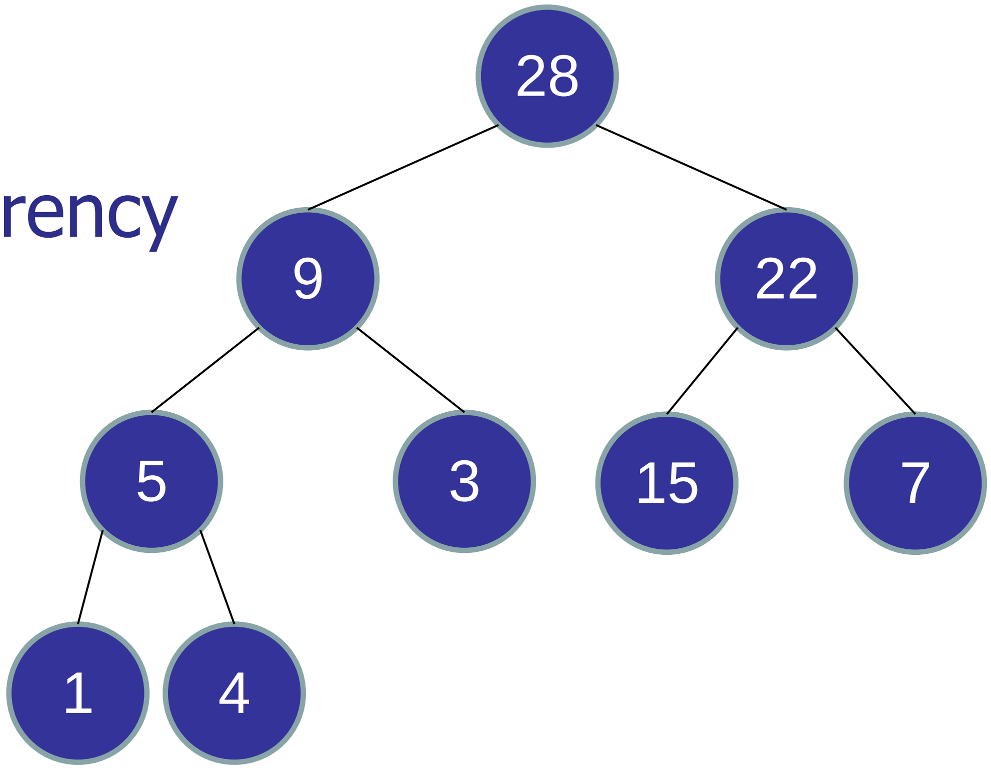
- insert
- extractMax
- increaseKey
- decreaseKey
- delete



(Max) Priority Queue

Heap vs. AVL Tree

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



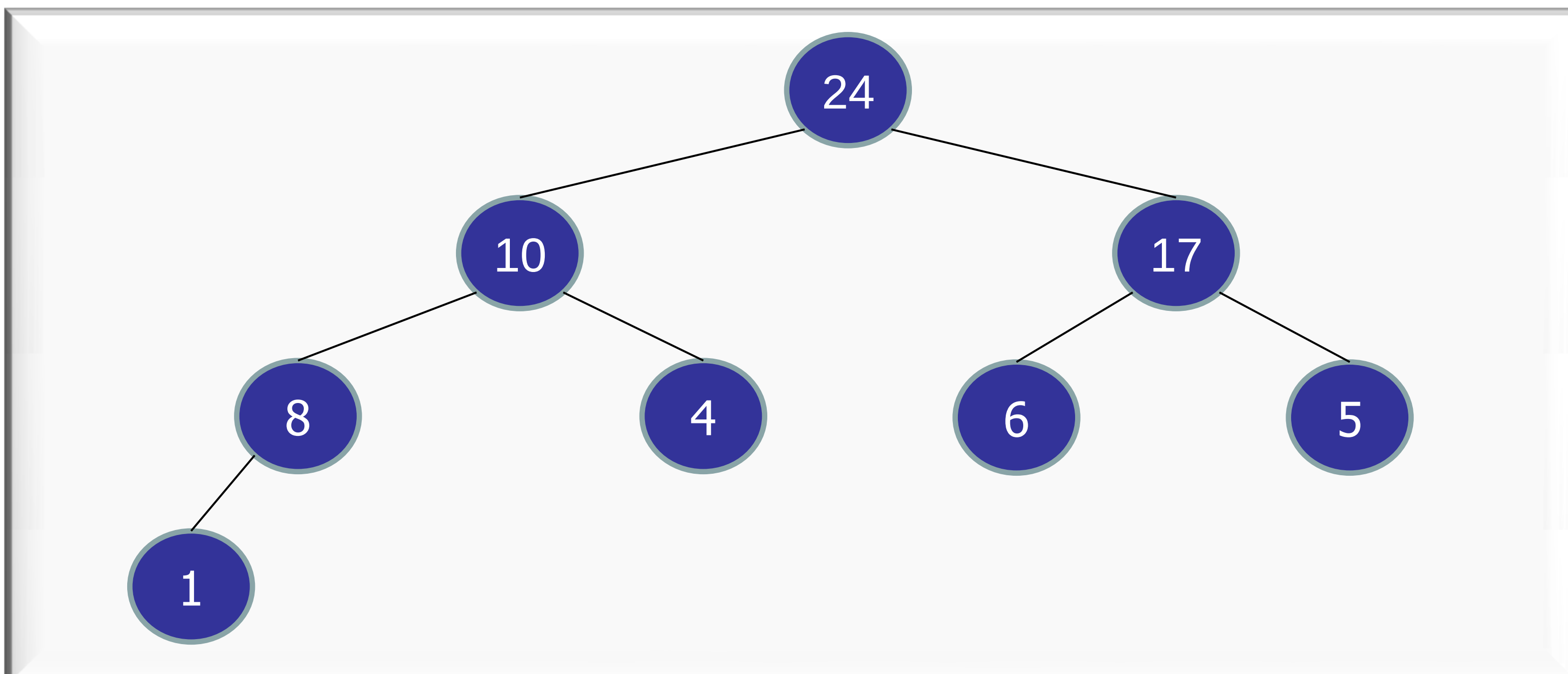
How to keep a tree?

- Store in an array!

Store Tree in an Array

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

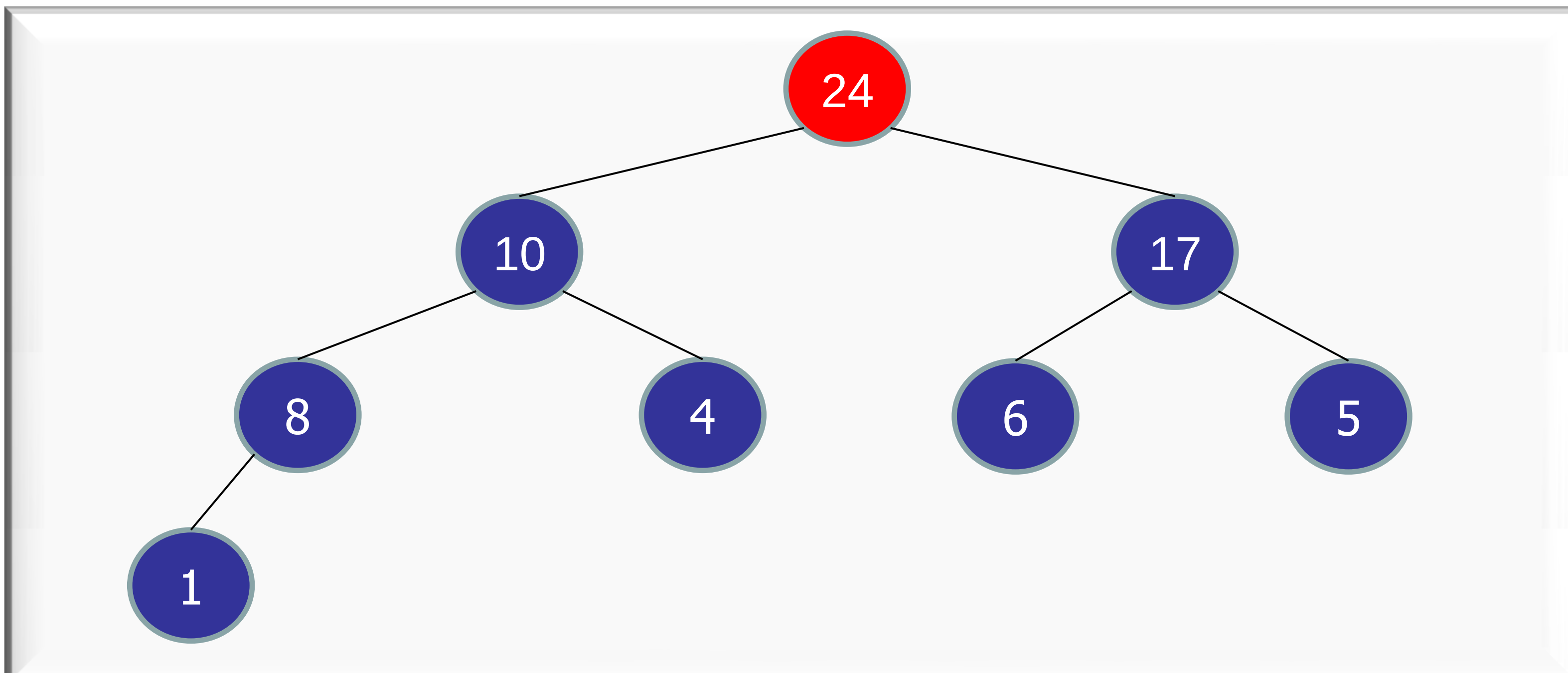
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	7	1	



Store Tree in an Array

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

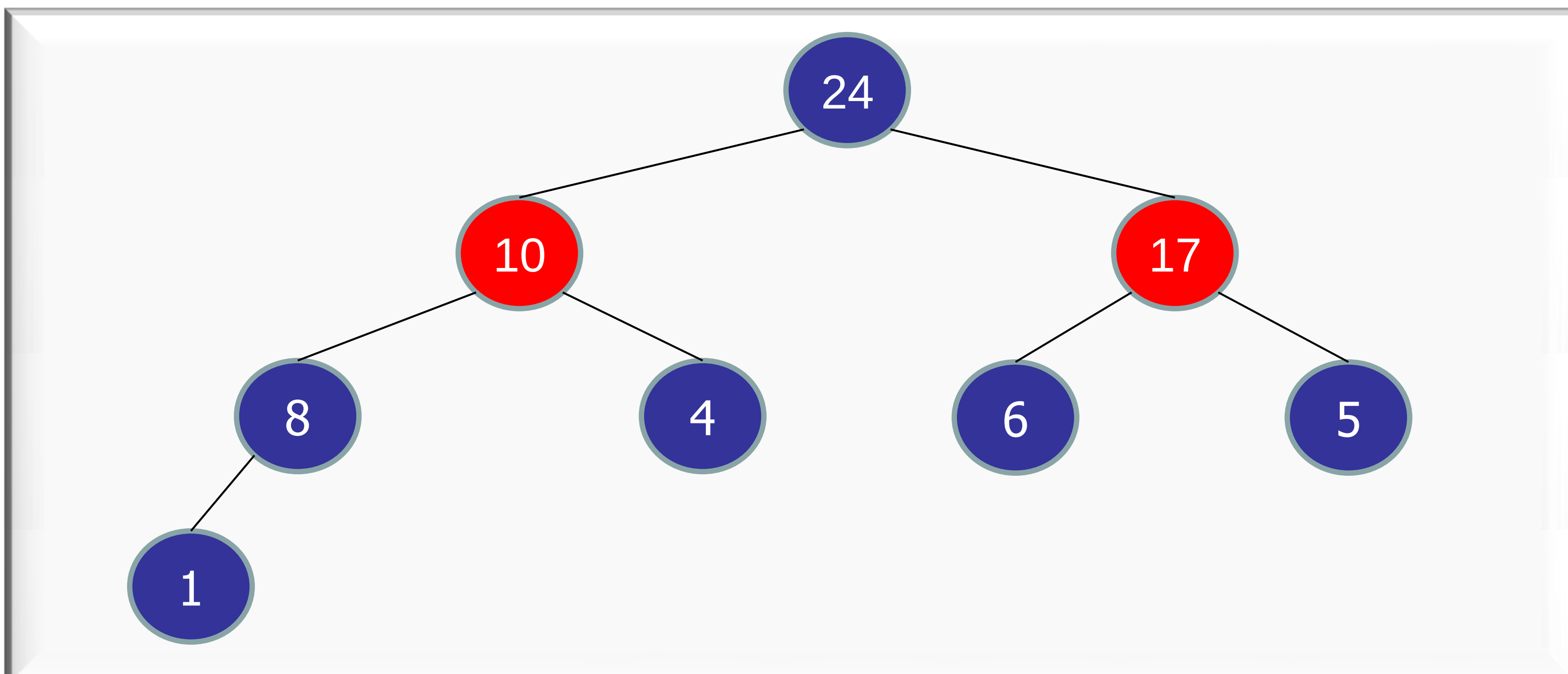
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	7	1	



Store Tree in an Array

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

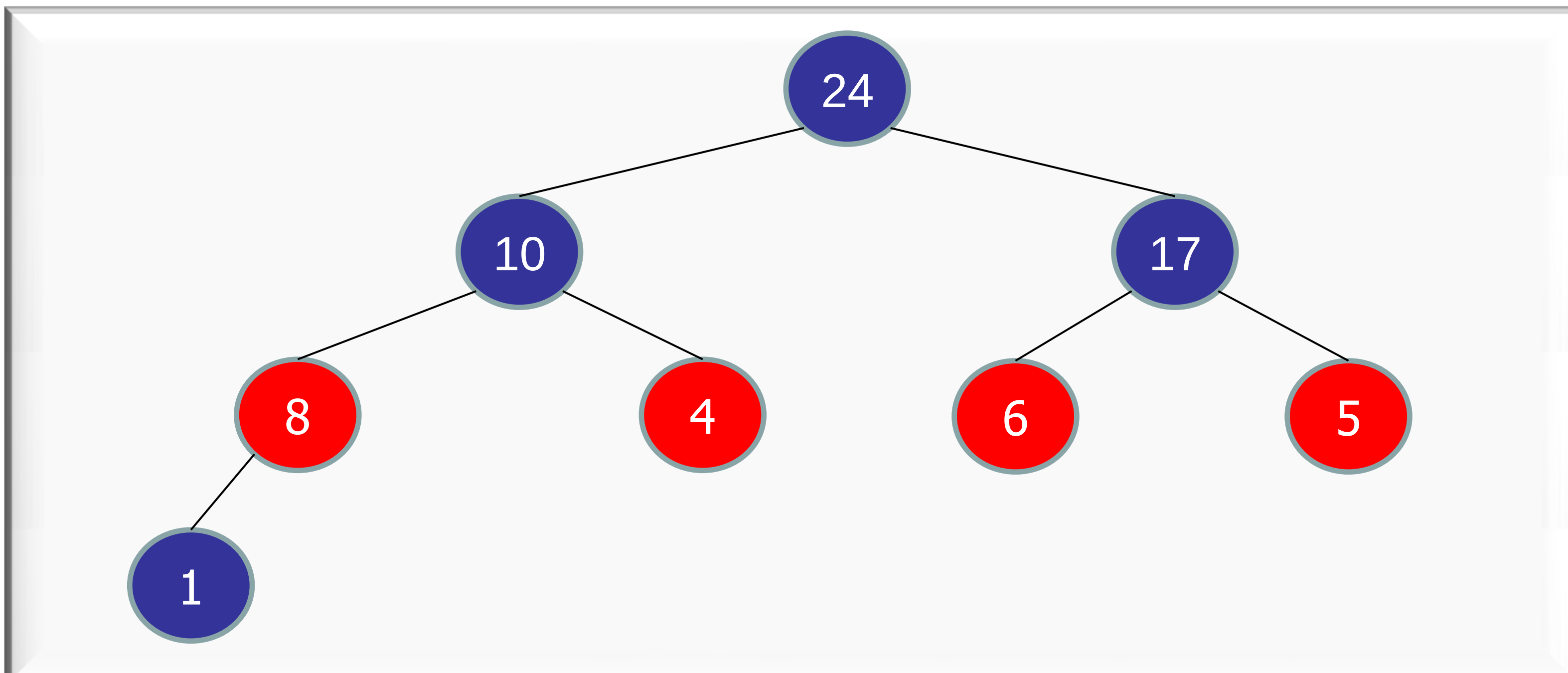
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	7	1	



Store Tree in an Array

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

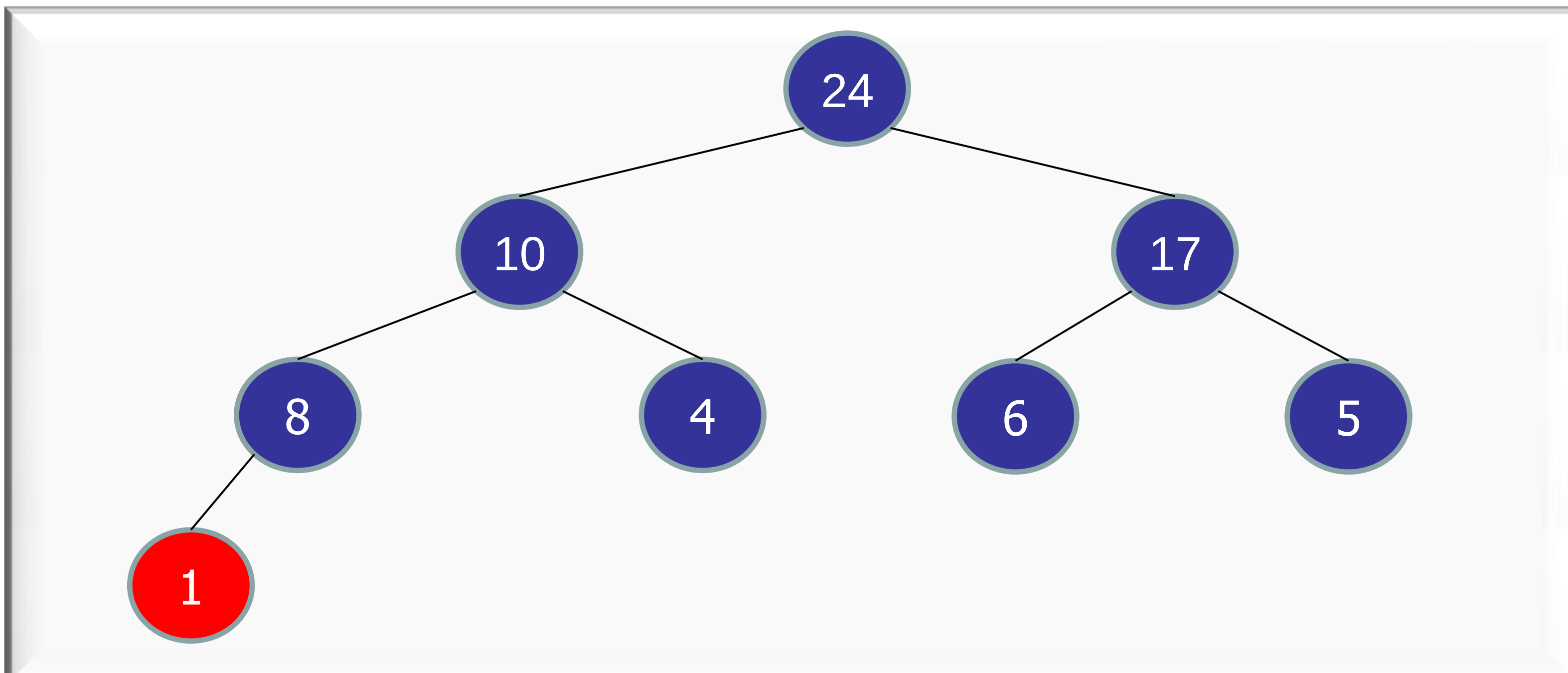
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	



Store Tree in an Array

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

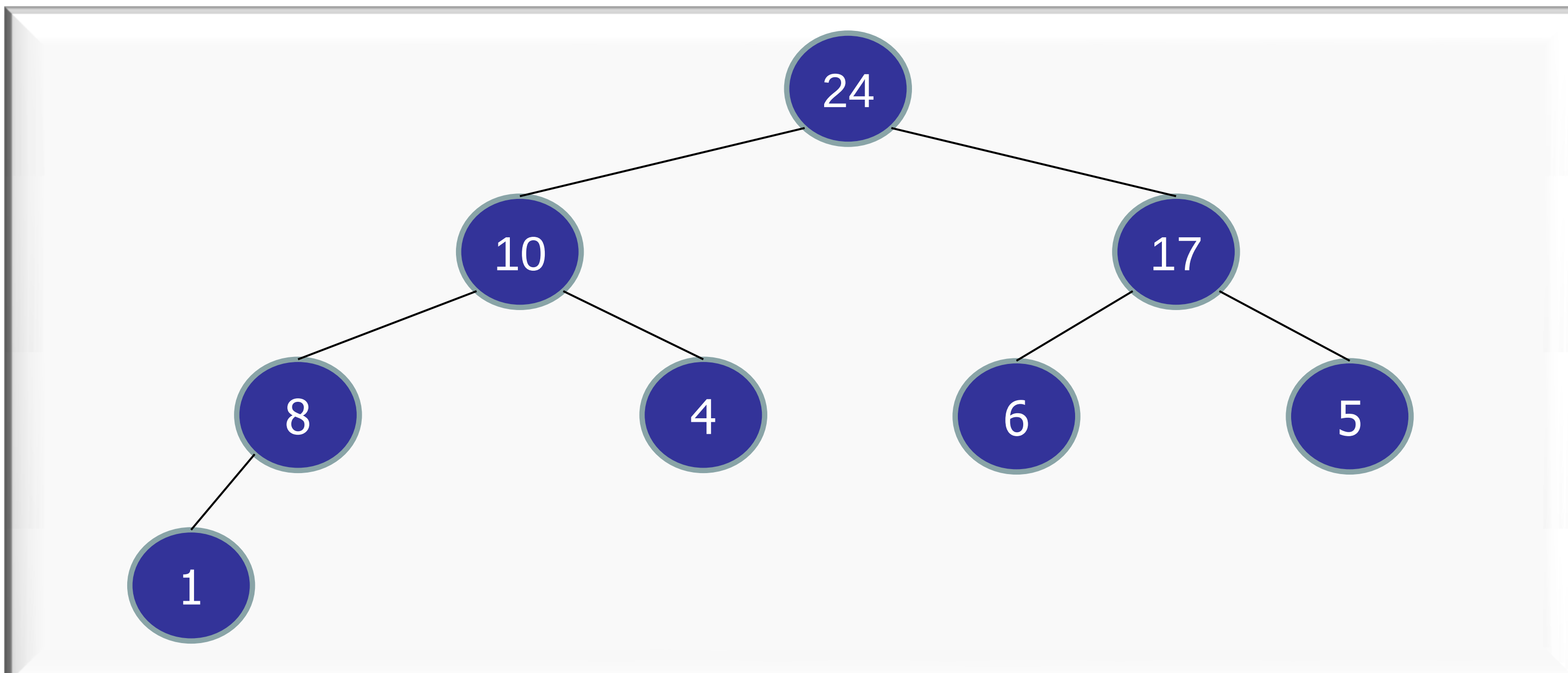
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	



Store Tree in an Array

```
insert(15) :
```

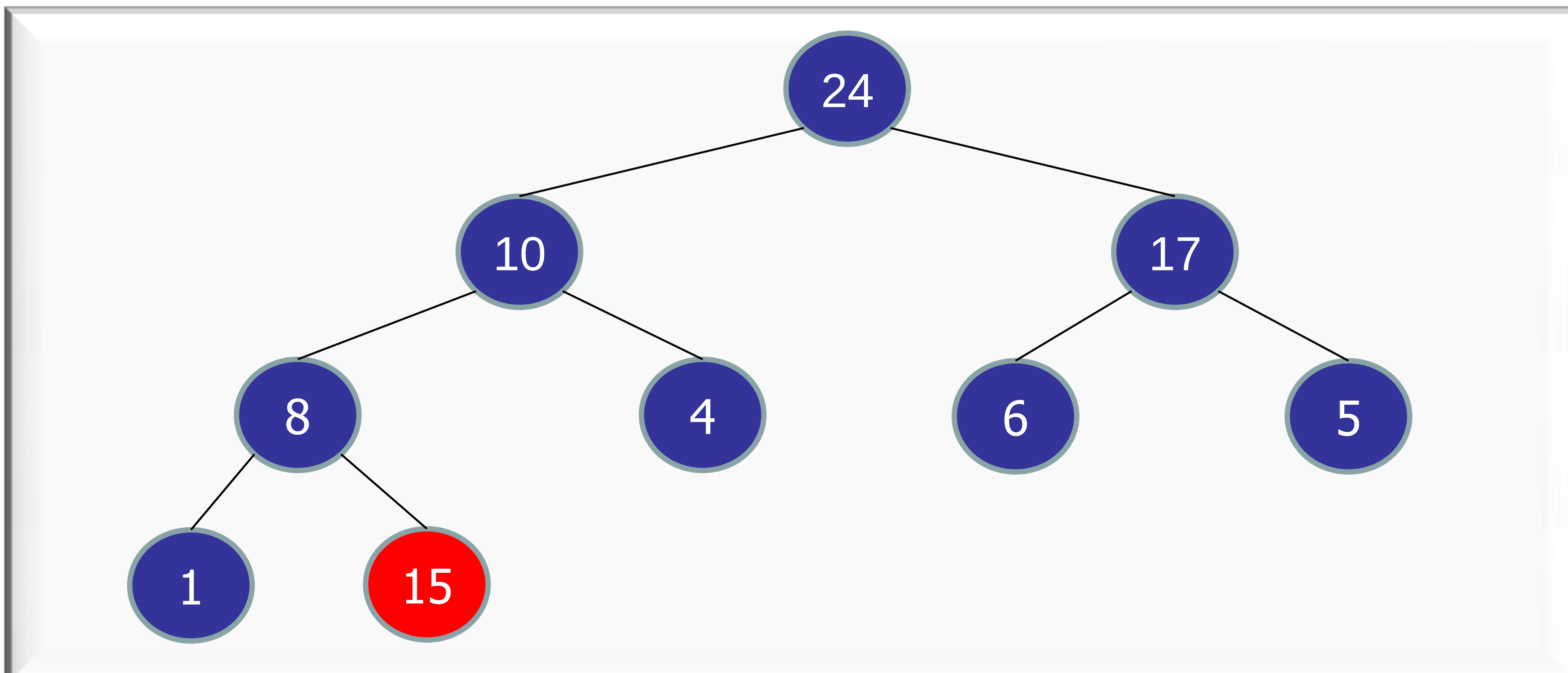
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	



Store Tree in an Array

`insert(15) :`

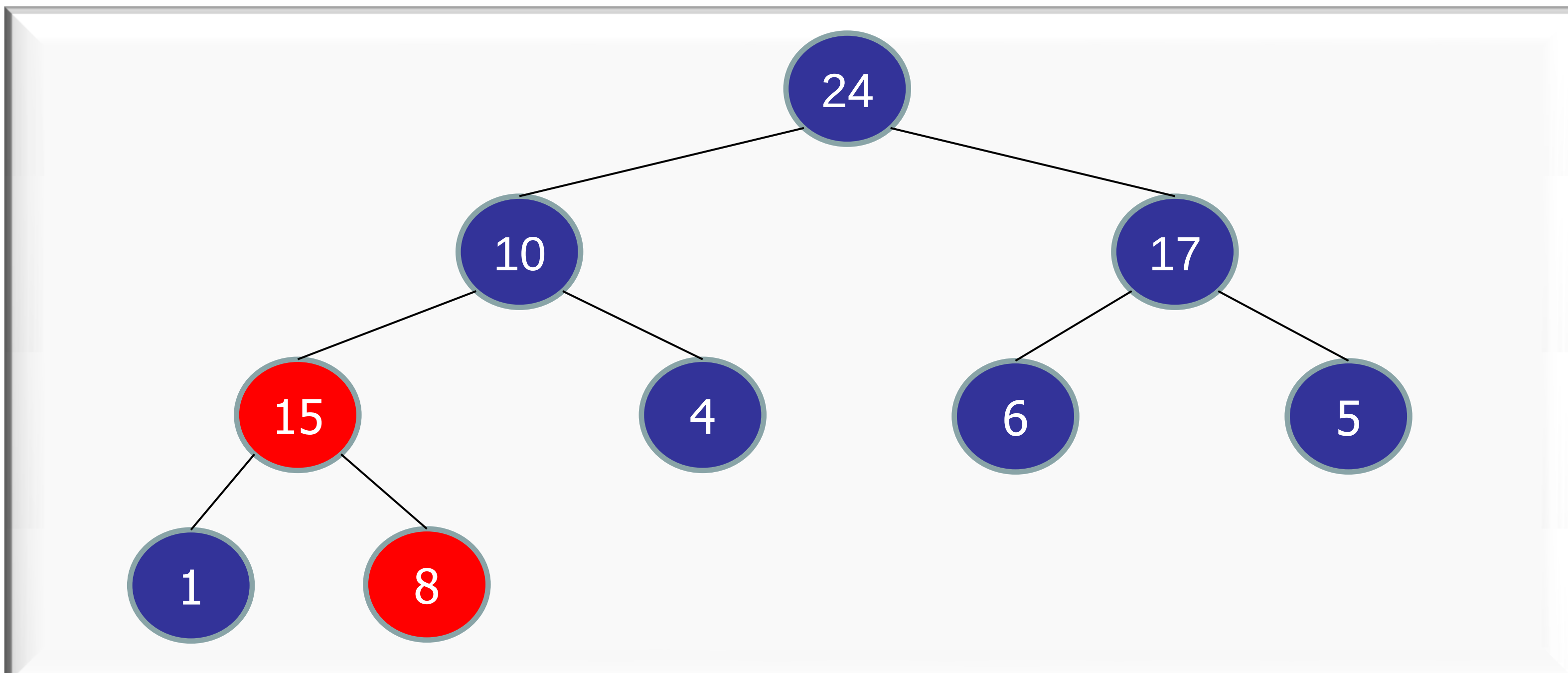
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	15



Store Tree in an Array

`insert(15) :`

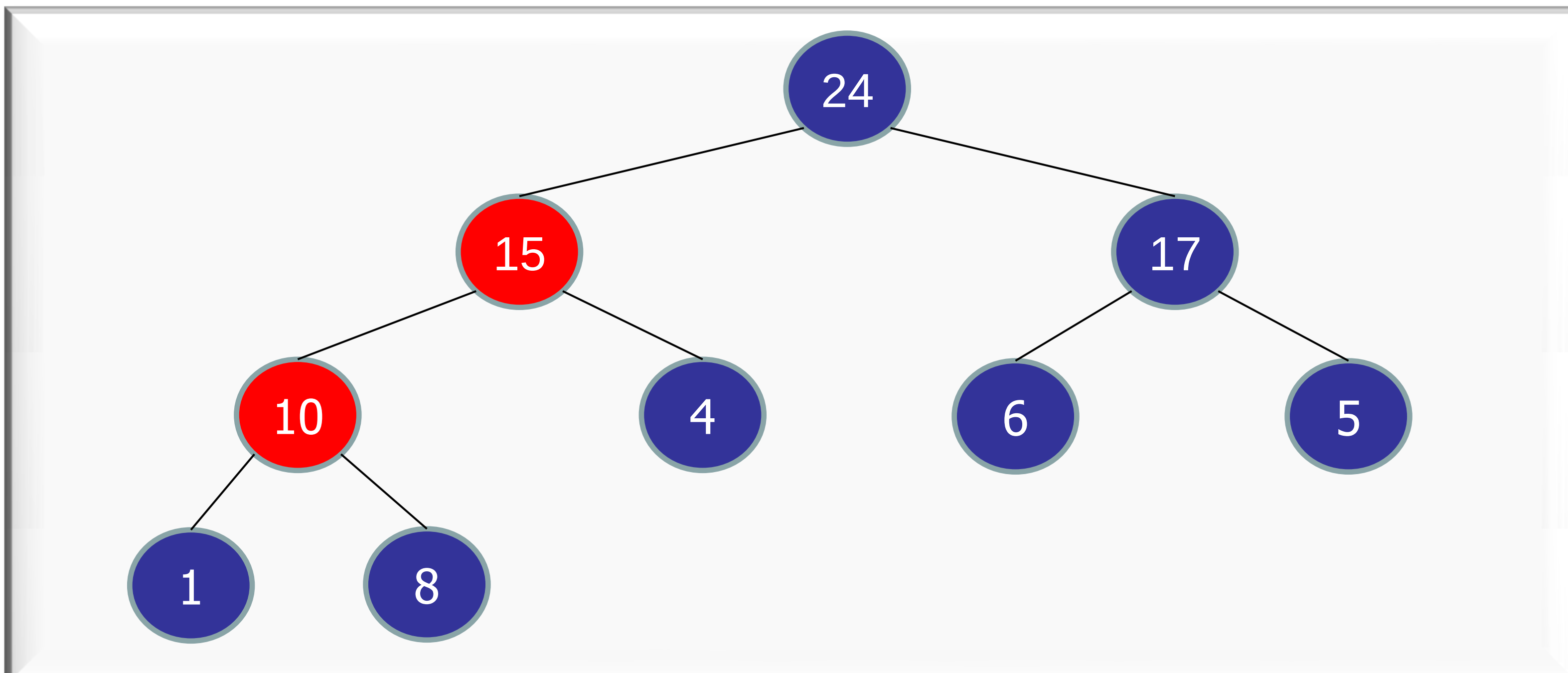
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	15	4	6	5	1	8



Store Tree in an Array

`insert(15) :`

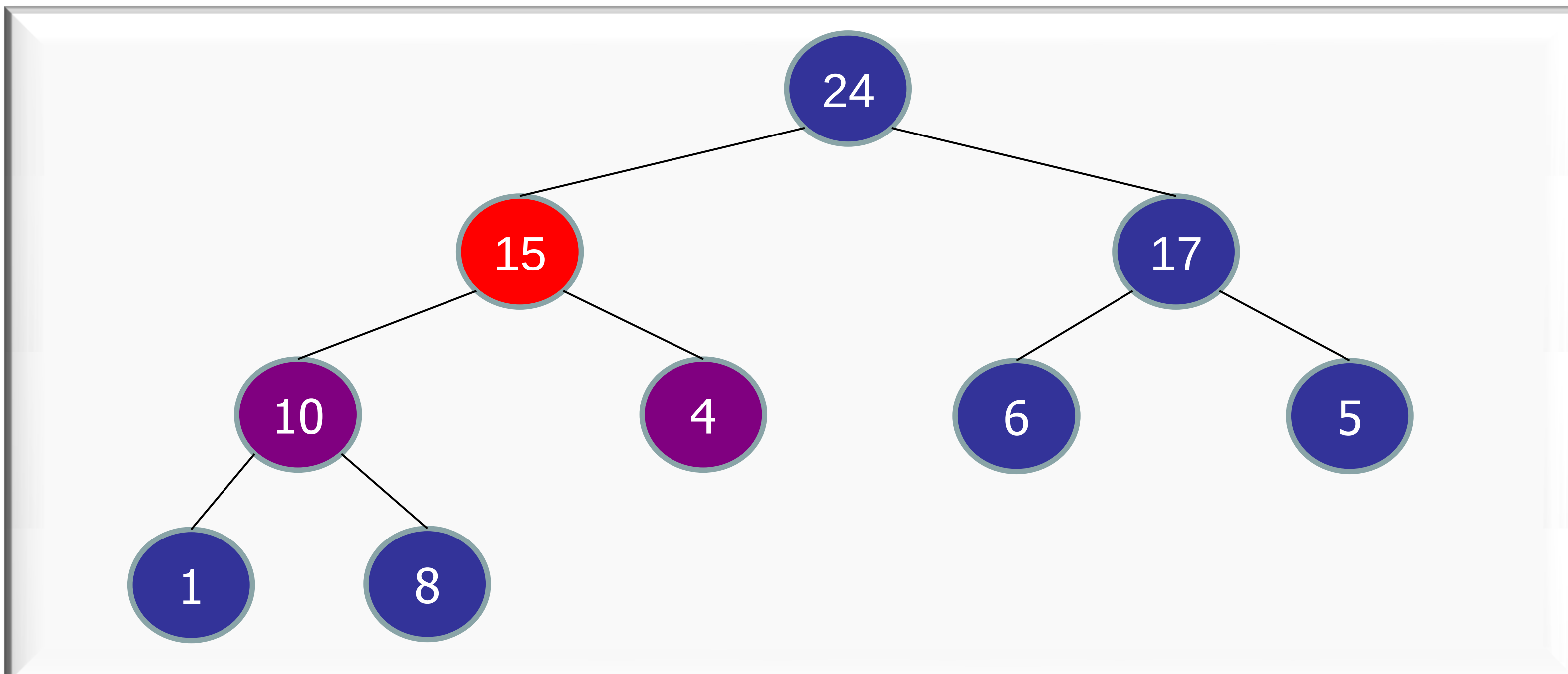
array slot	0	1	2	3	4	5	6	7	8
priority	24	15	17	10	4	6	5	1	8



Store Tree in an Array

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

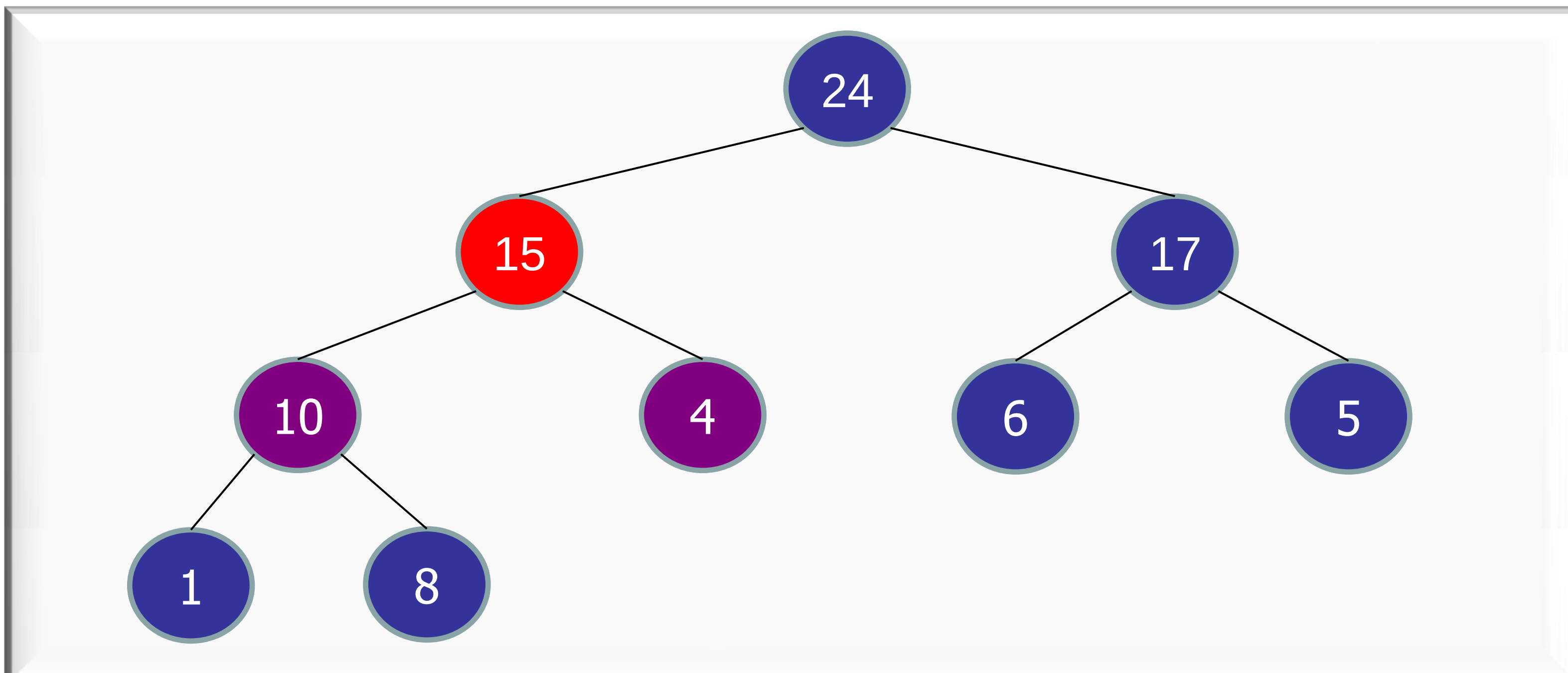
array slot	0	1	2	3	4	5	6	7	8
priority	24	15	17	10	4	6	5	1	8



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**



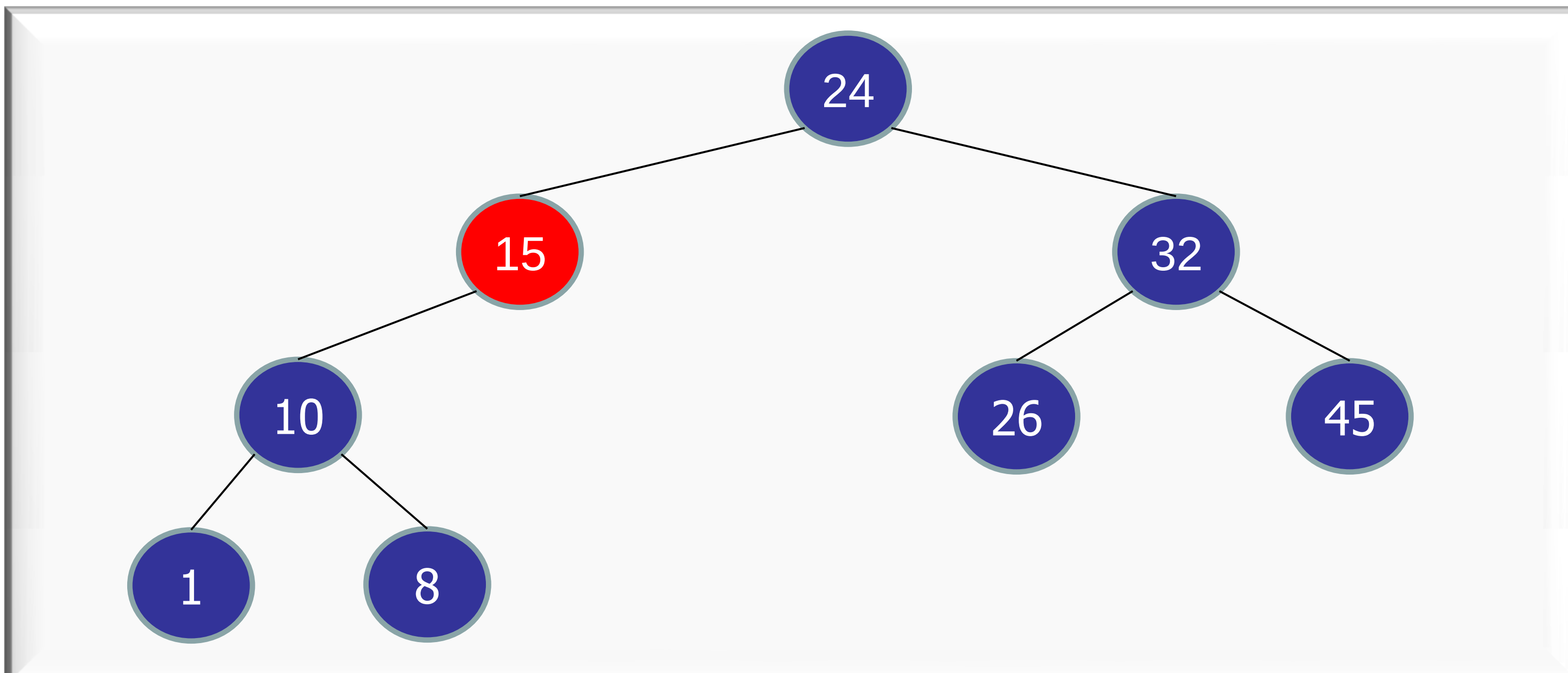
Why not store an AVL tree in an array?

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 AVL Tree in an Array

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

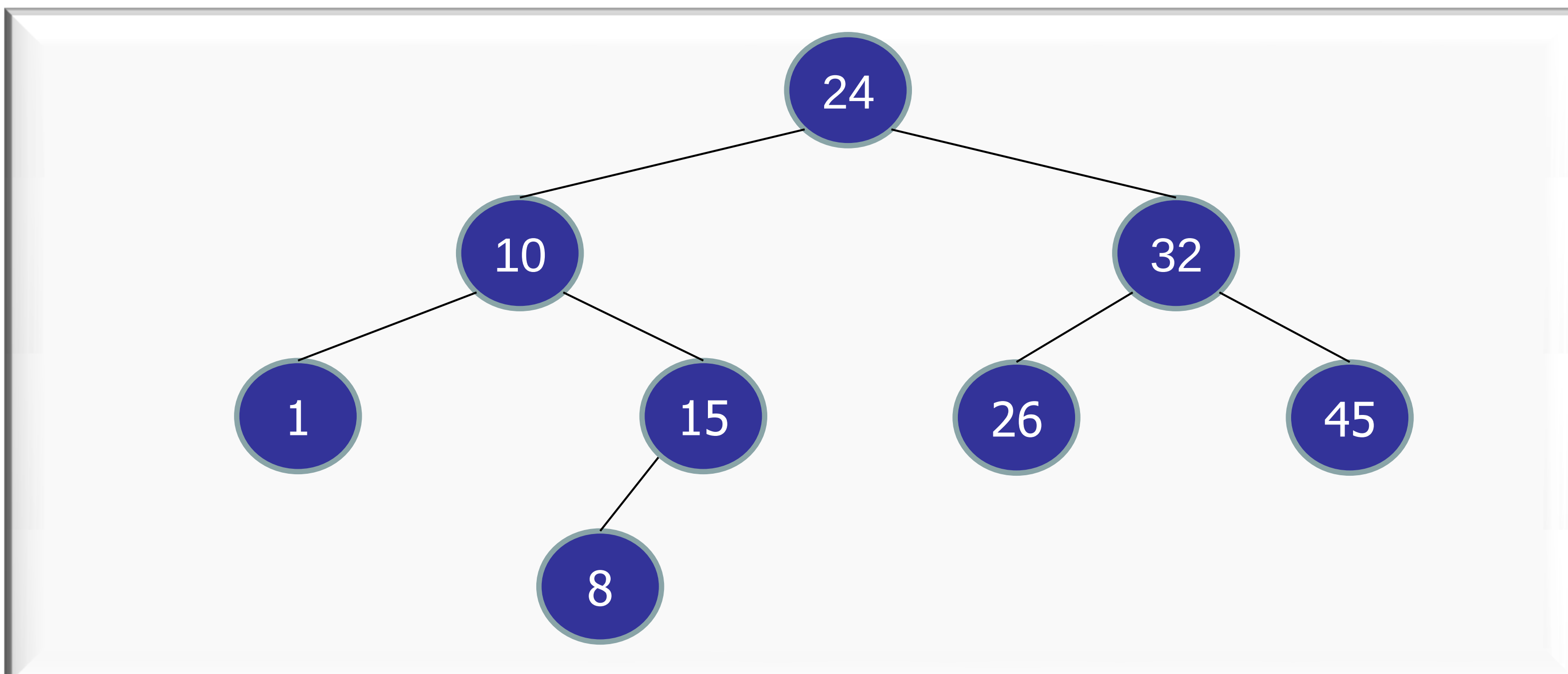
array slot	0	1	2	3	4	5	6	7	8
priority	24	15	32	10		26	45	1	8



Store AVL Tree in an Array

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

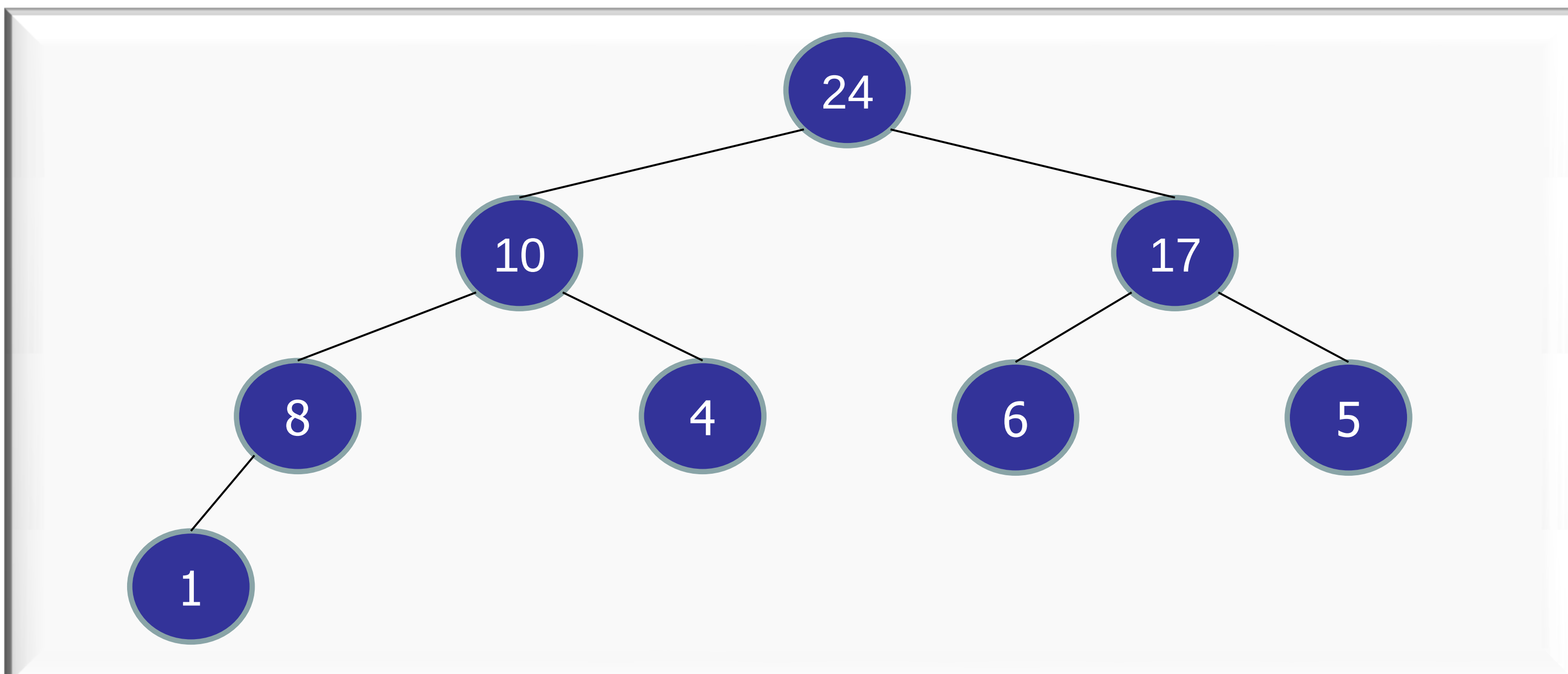
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	32	1	15	26	45	8	



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	



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

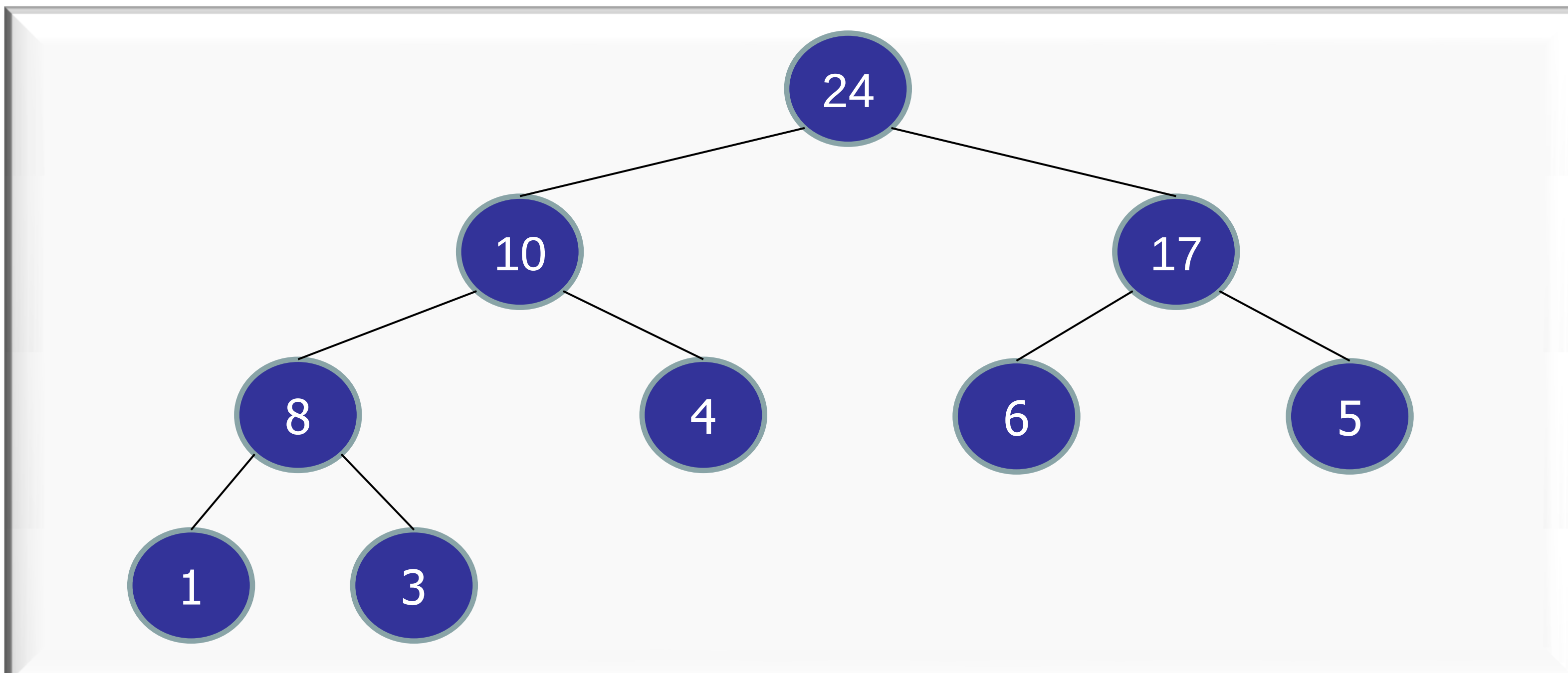
Heap → Sorted list:

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:

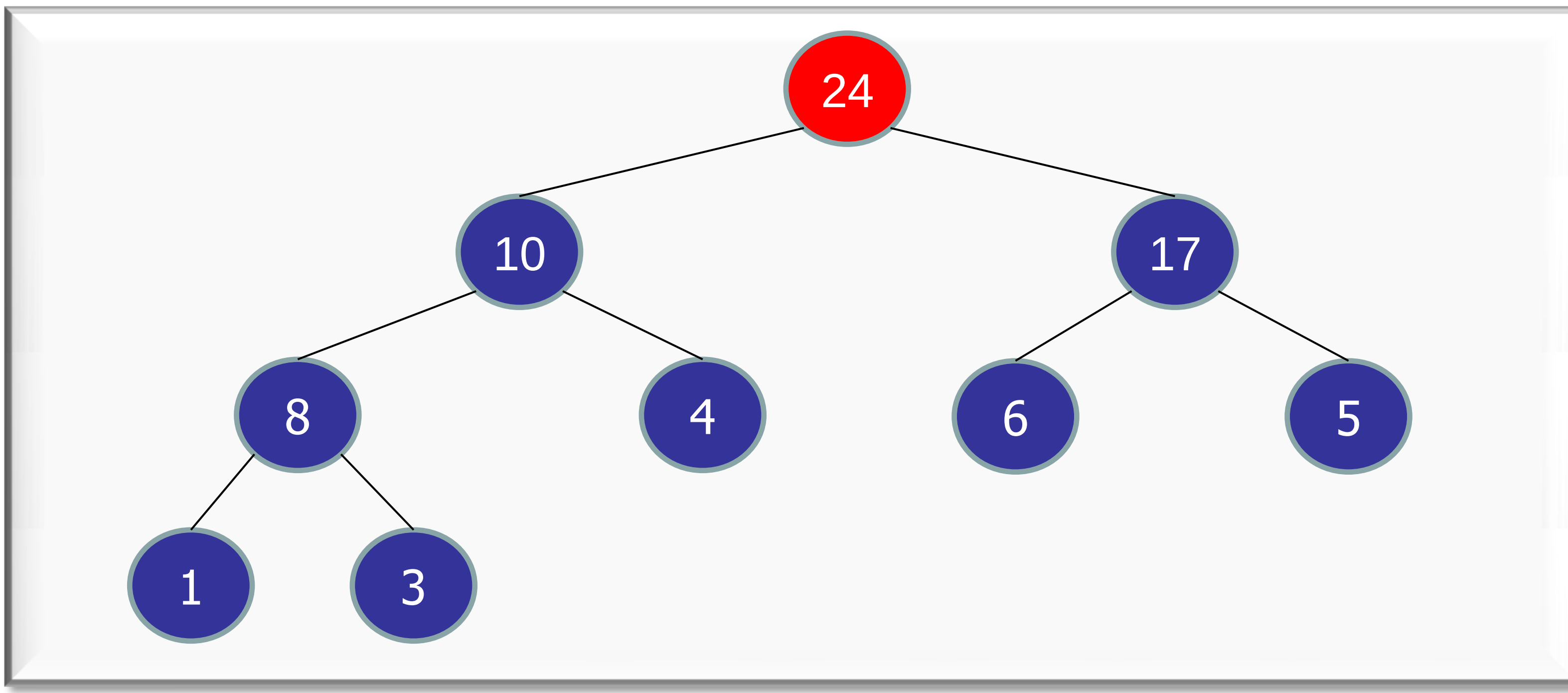
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	3



HeapSort

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

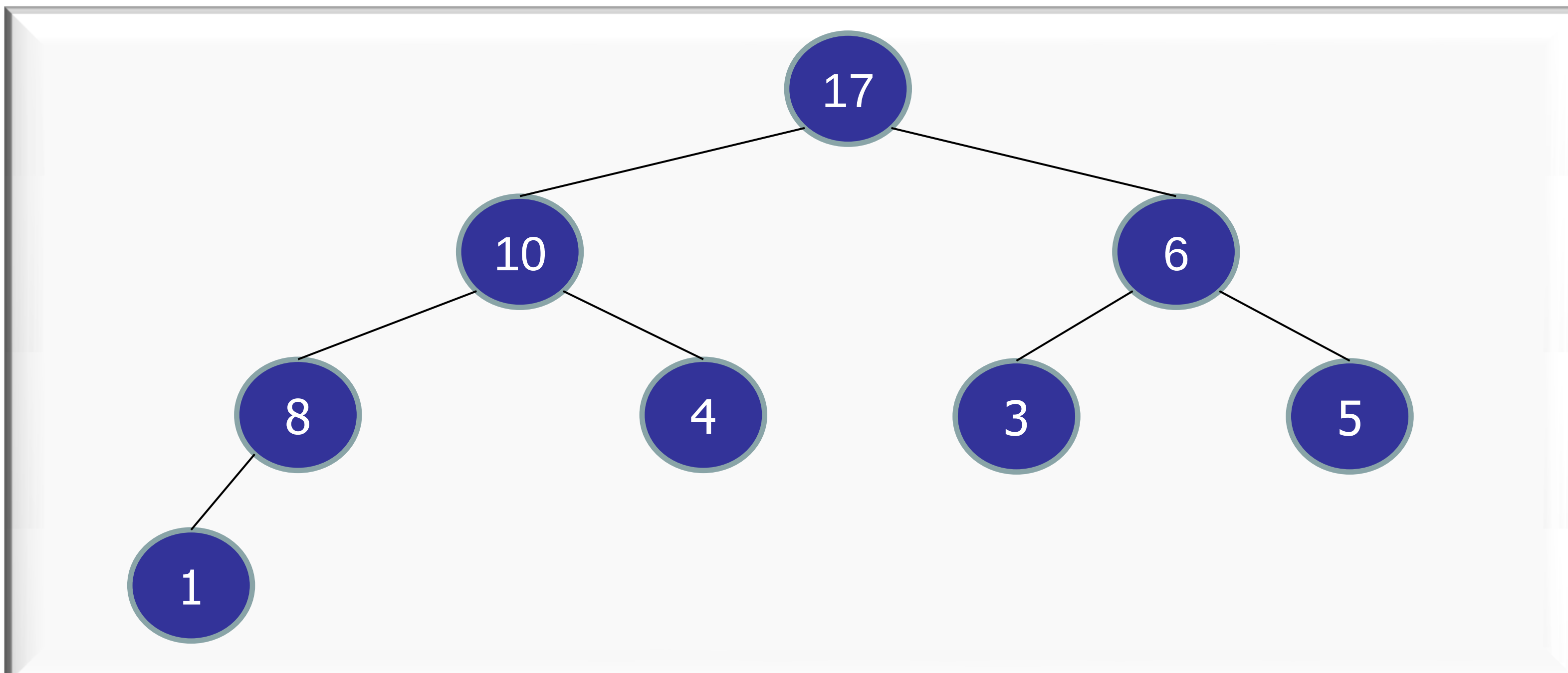
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	3



HeapSort

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

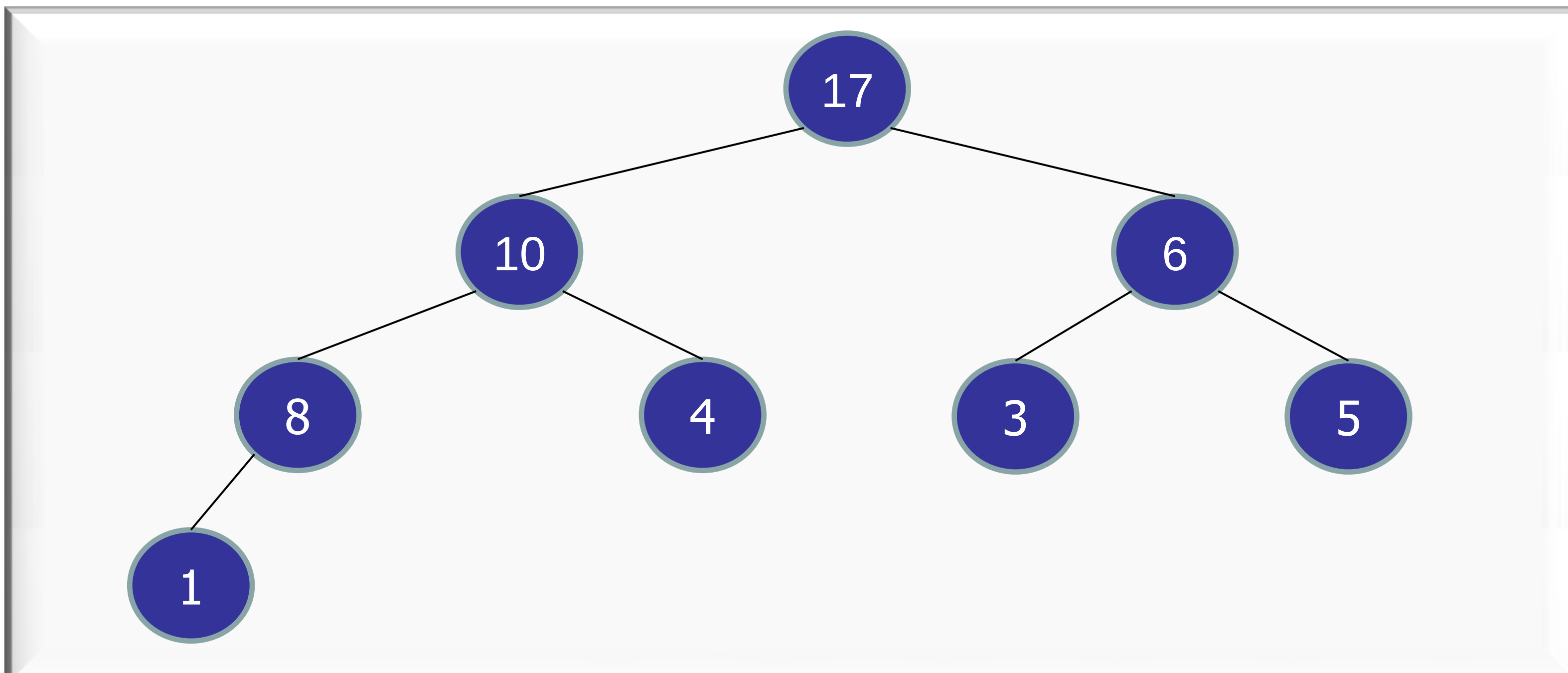
array slot	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
priority	**17**	**10**	**6**	**8**	**4**	**3**	**5**	**1**	



HeapSort

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

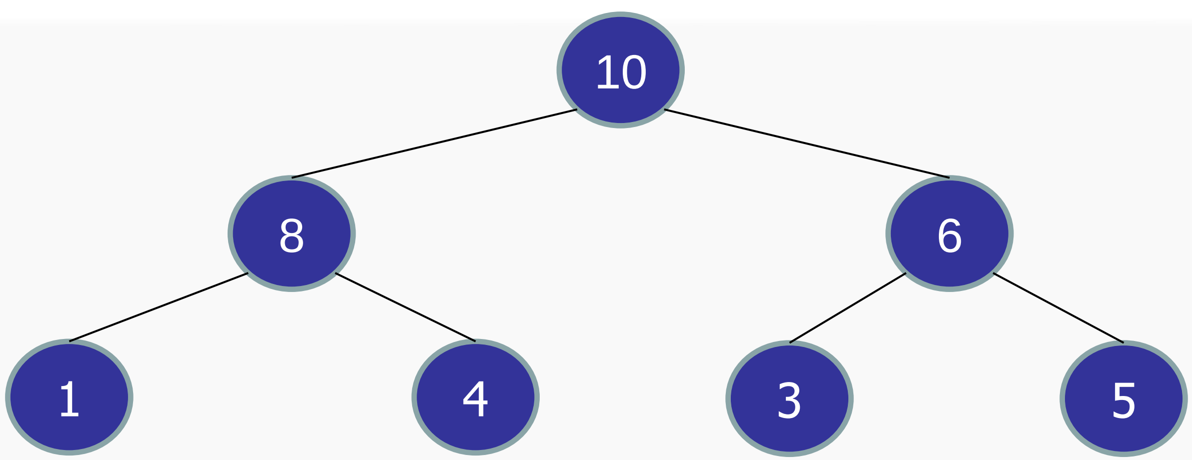
array slot	0	1	2	3	4	5	6	7	8
priority	17	10	6	8	4	3	5	1	24



HeapSort

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

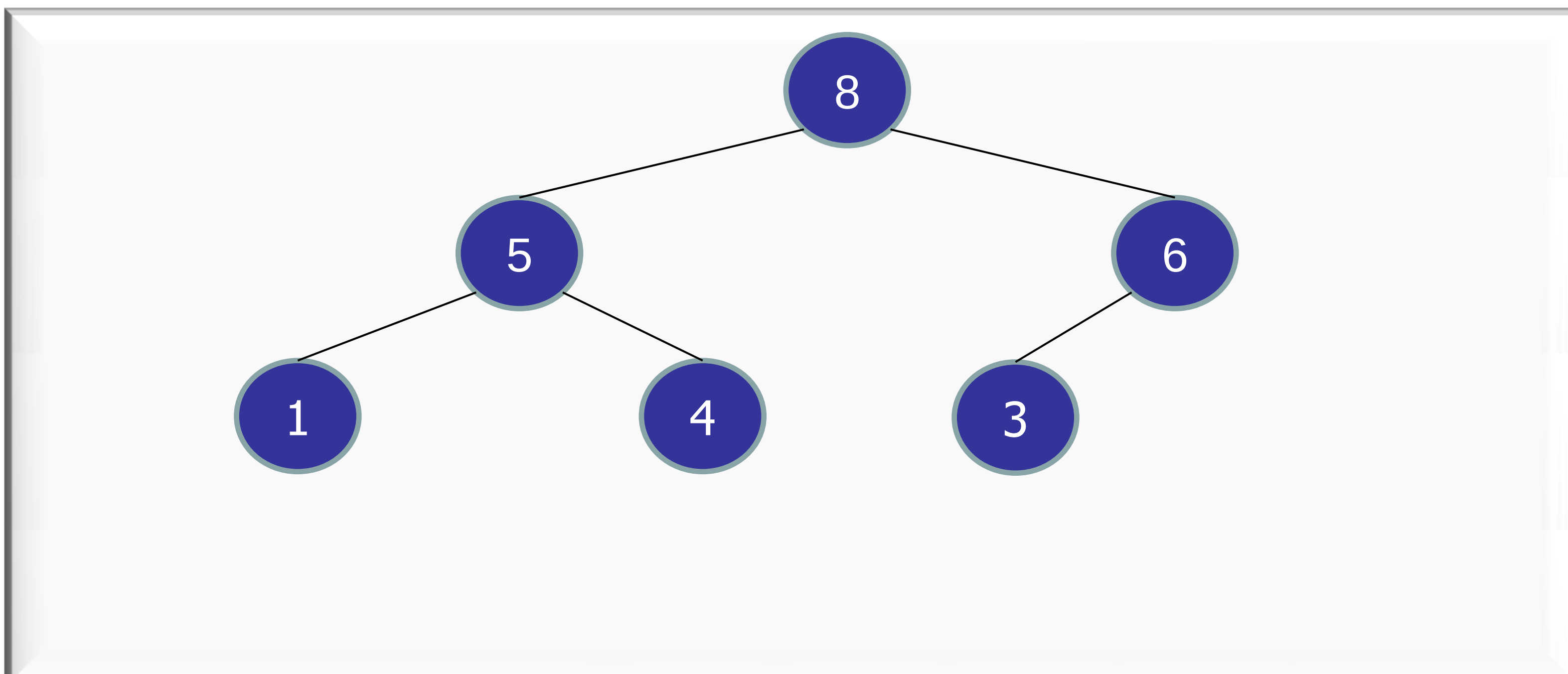
array slot	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
priority	**10**	**8**	**6**	**1**	**4**	**3**	**5**	**17**	**24**



HeapSort

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

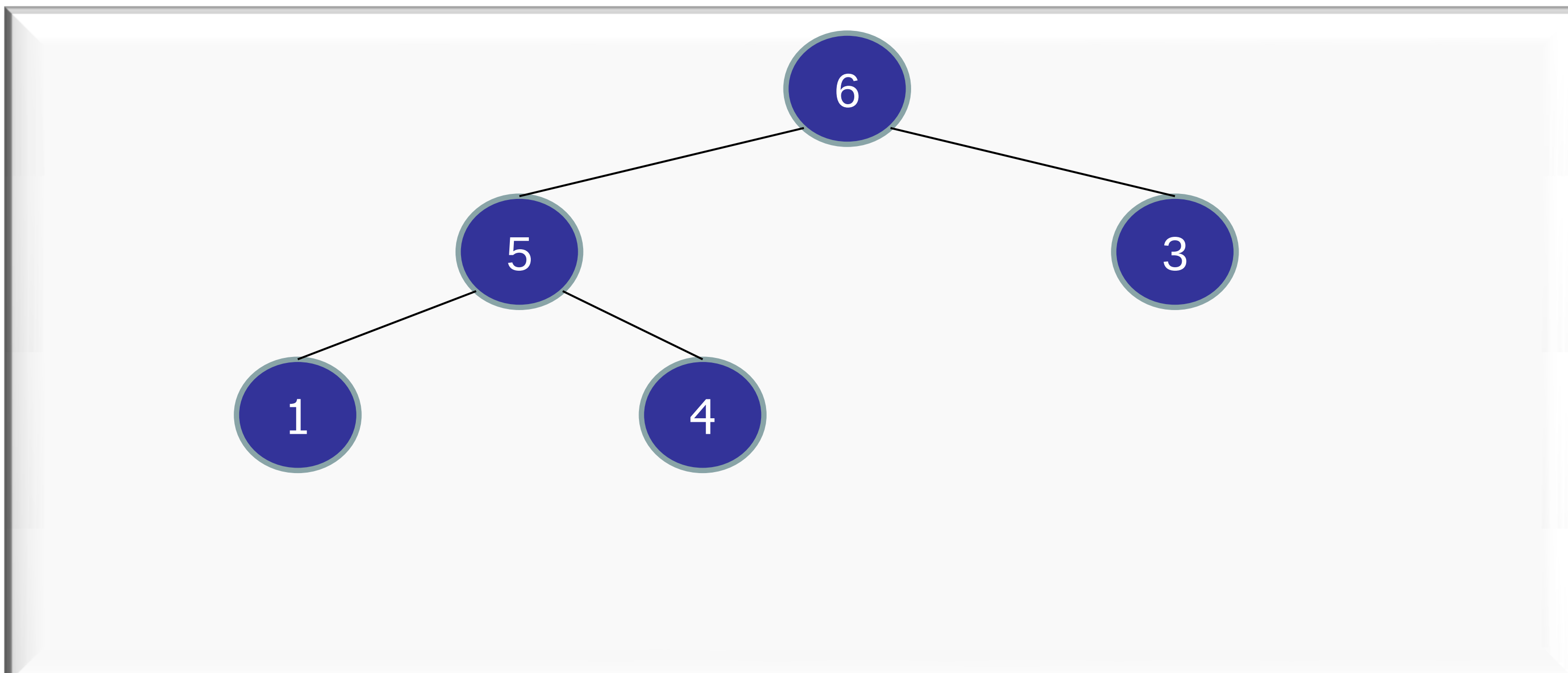
array slot	0	1	2	3	4	5	6	7	8
priority	8	5	6	1	4	3	10	17	24



HeapSort

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

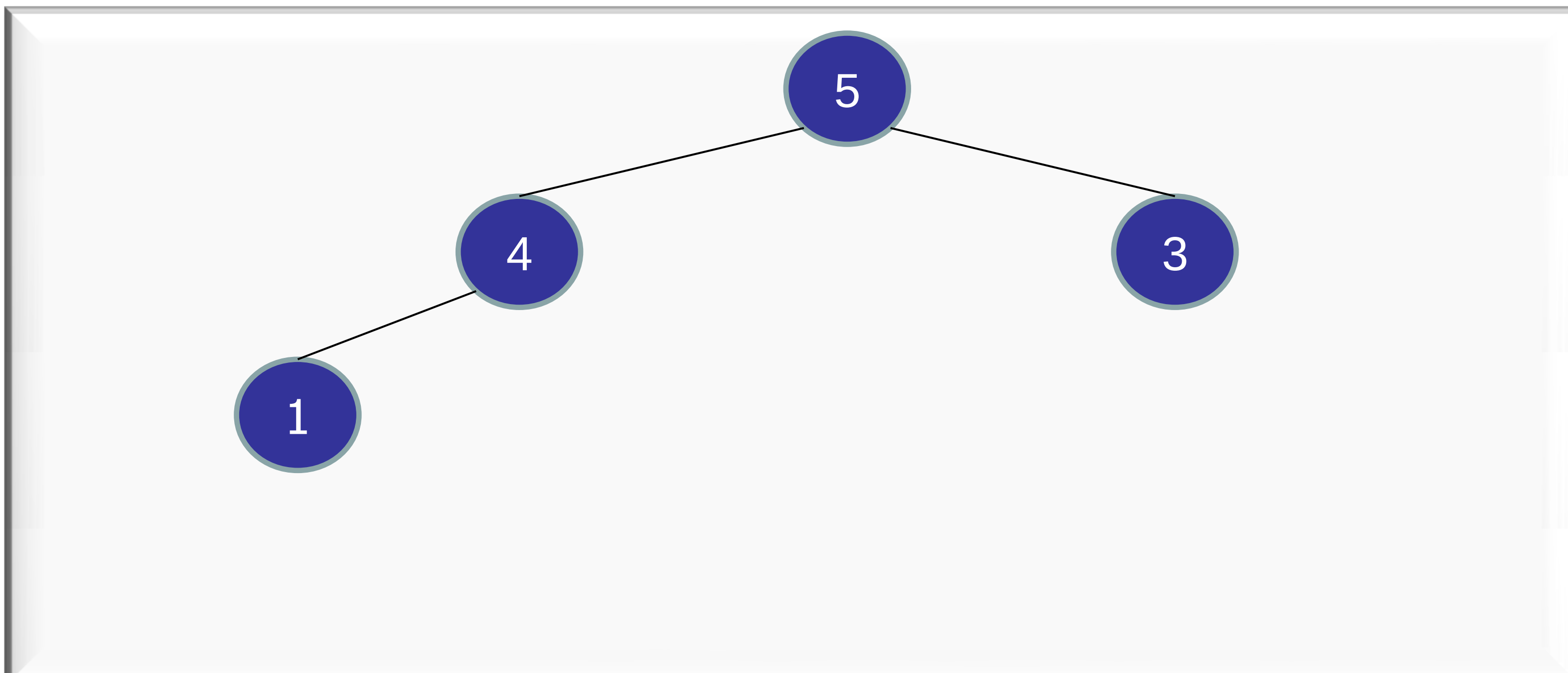
array slot	0	1	2	3	4	5	6	7	8
priority	6	5	3	1	4	8	10	17	24



HeapSort

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

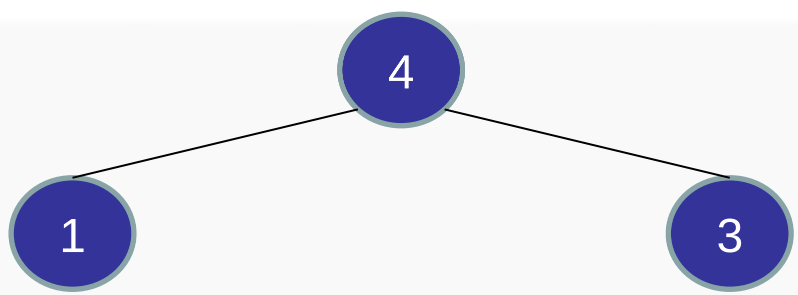
array slot	0	1	2	3	4	5	6	7	8
priority	5	4	3	1	6	8	10	17	24



HeapSort

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

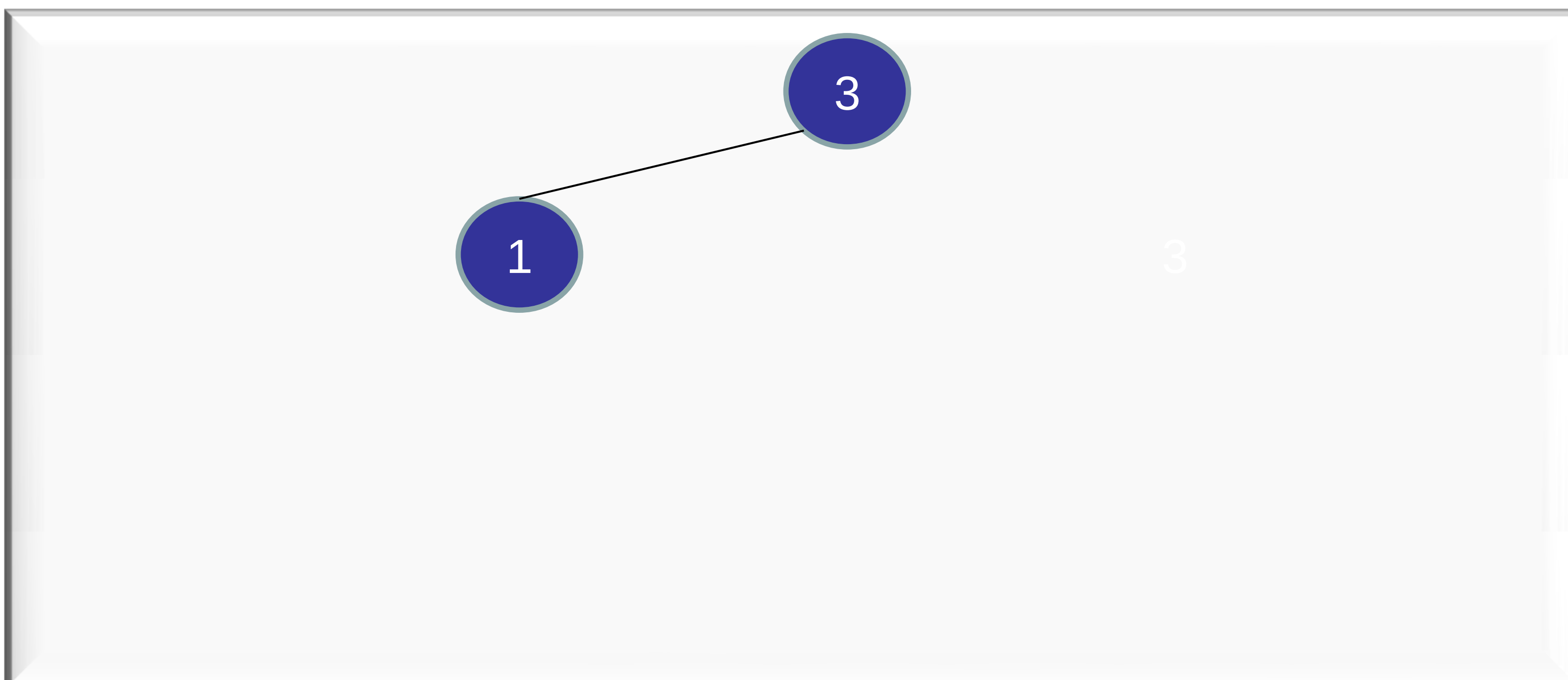
array slot	0	1	2	3	4	5	6	7	8
priority	4	1	3	5	6	8	10	17	24



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)

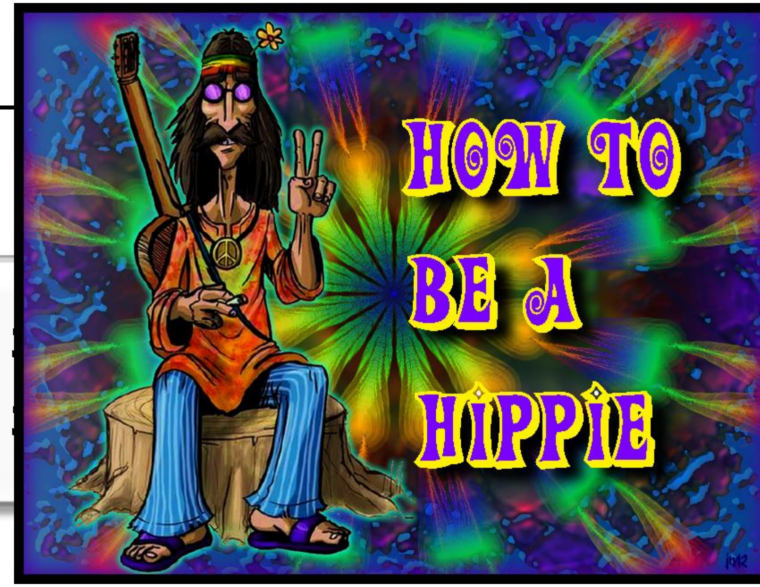
array slot	0	1	2	3	4	5	6	7	8
priority	1	3	4	5	6	8	10	17	24

```
// 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	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	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: 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)

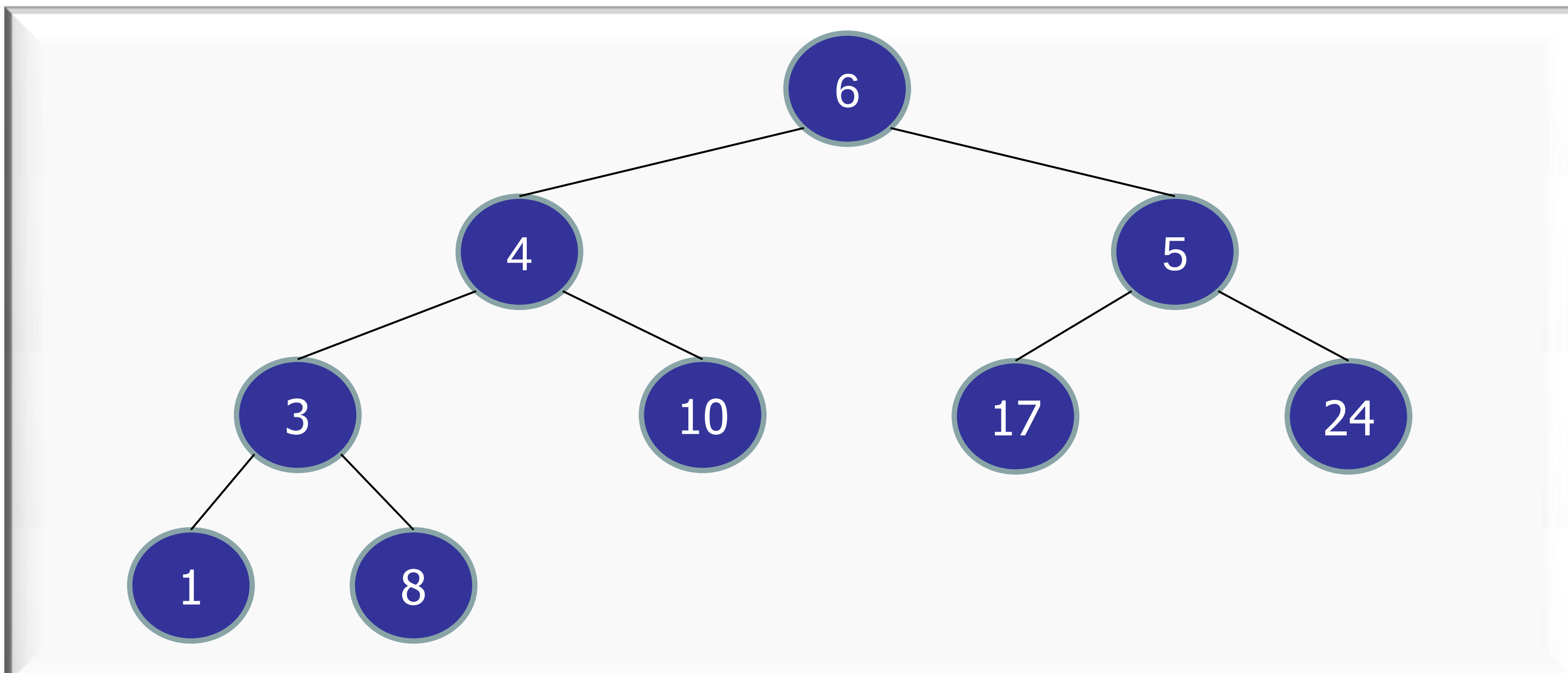
array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

```
// 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

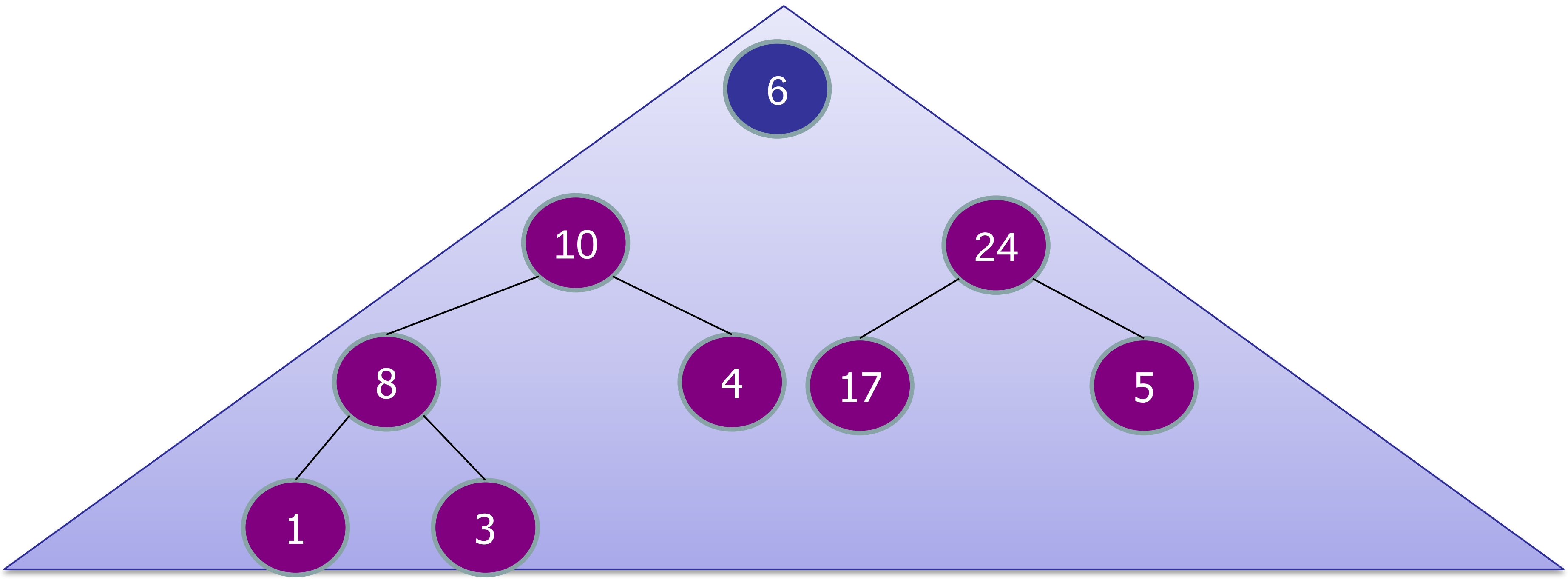
array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8



HeapSort

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>?

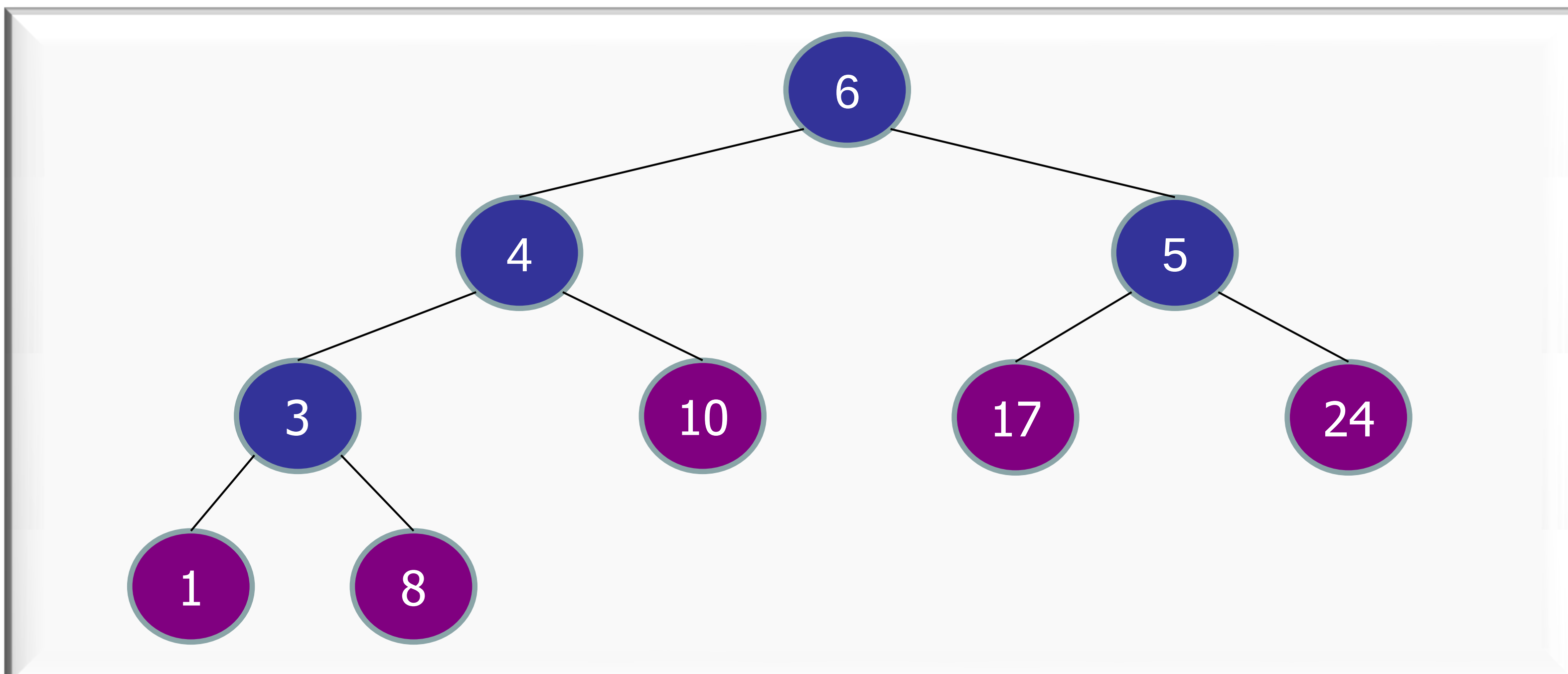


HeapSort

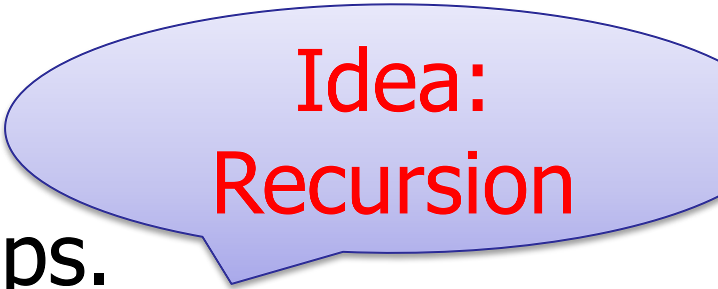
Idea: Recursion

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

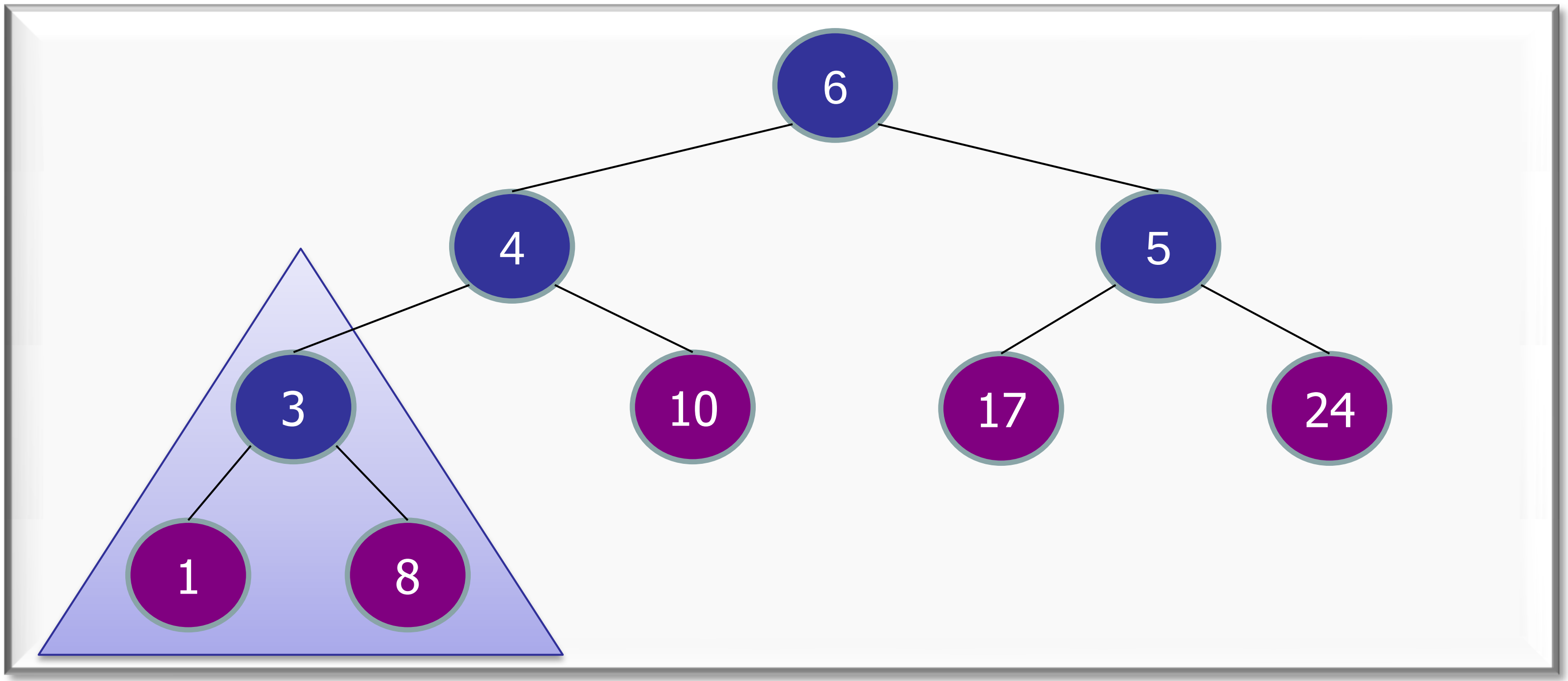


HeapSort



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

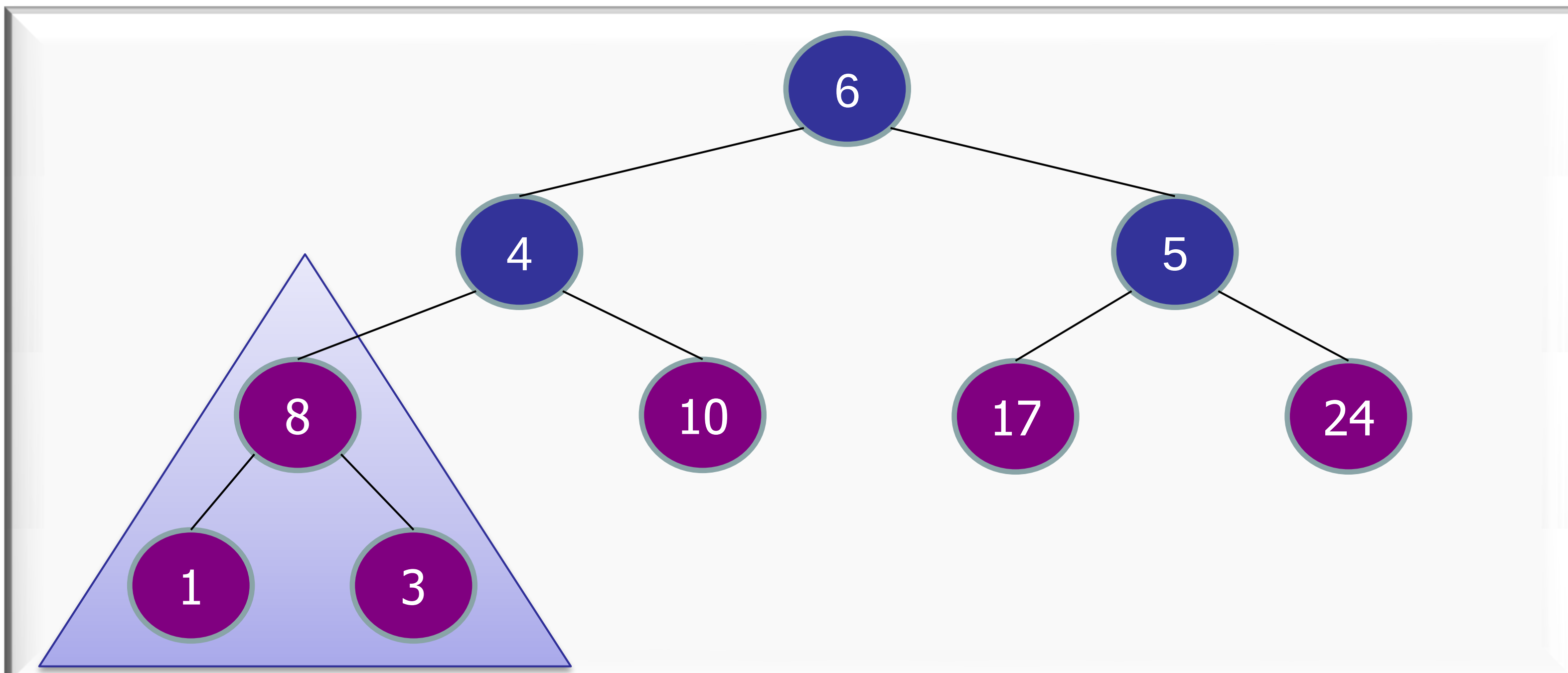


HeapSort

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**

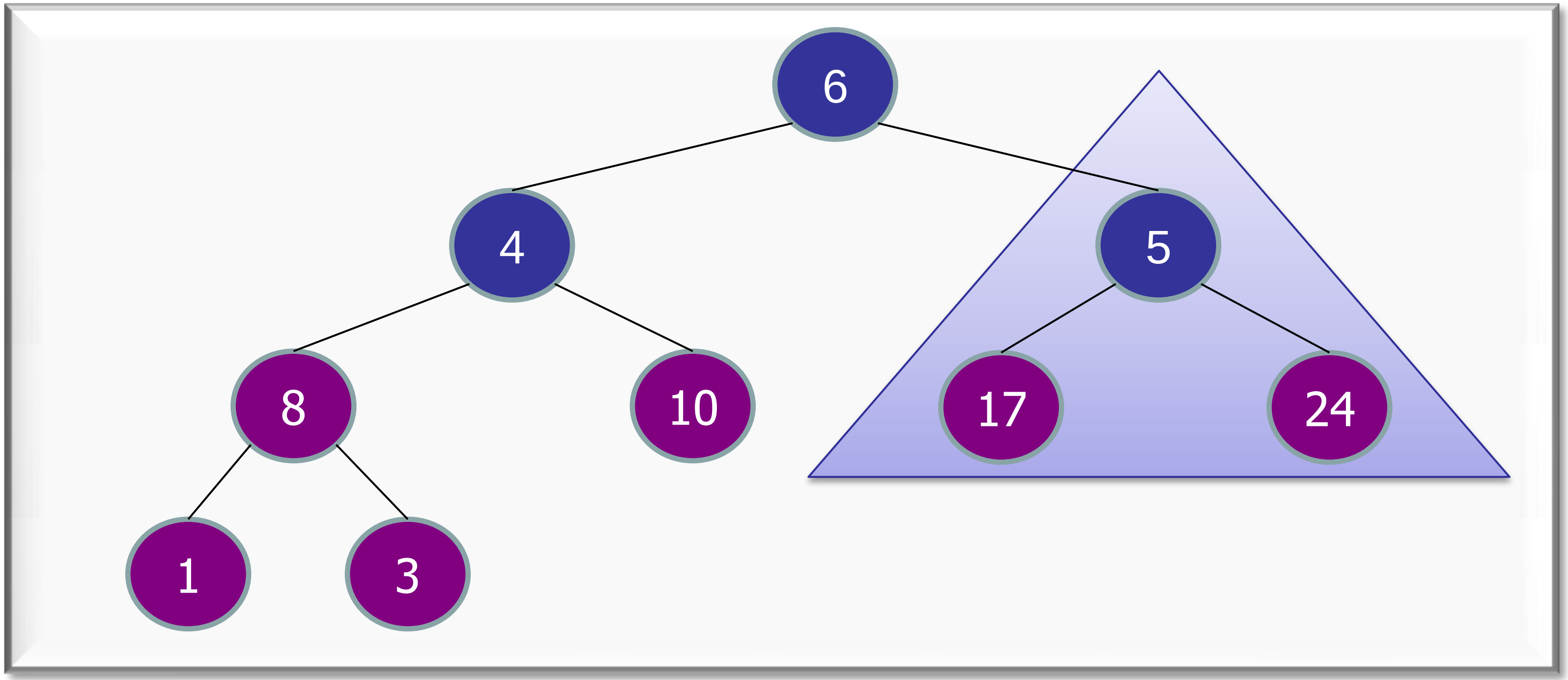


HeapSort

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

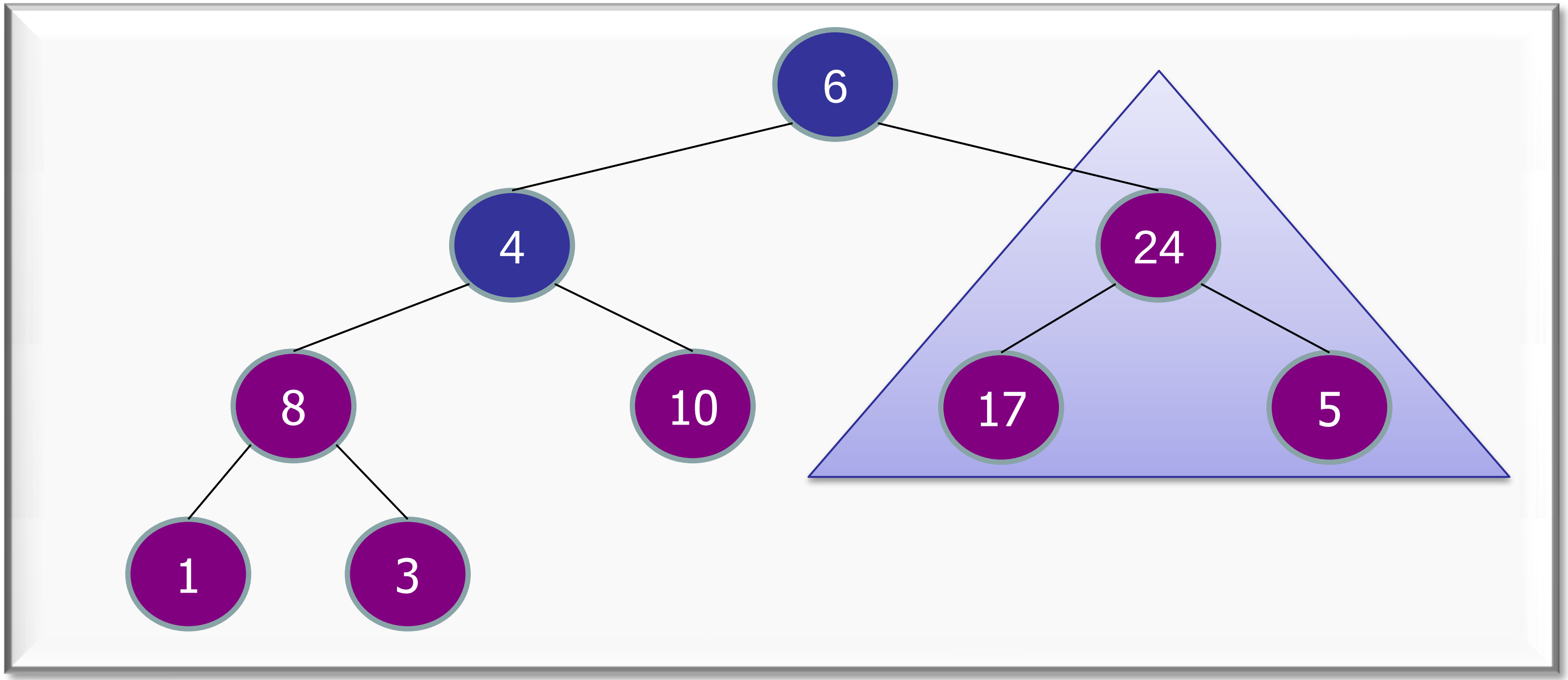


HeapSort

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

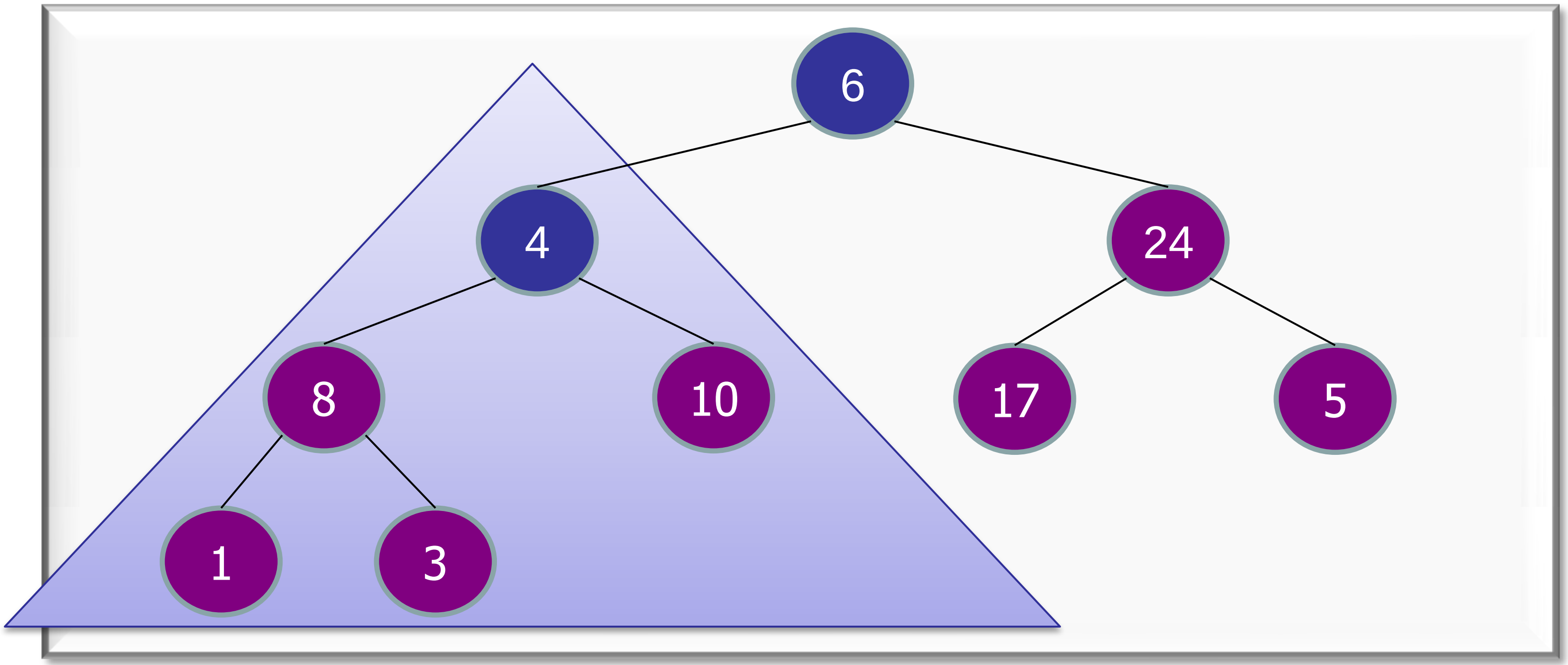


HeapSort

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

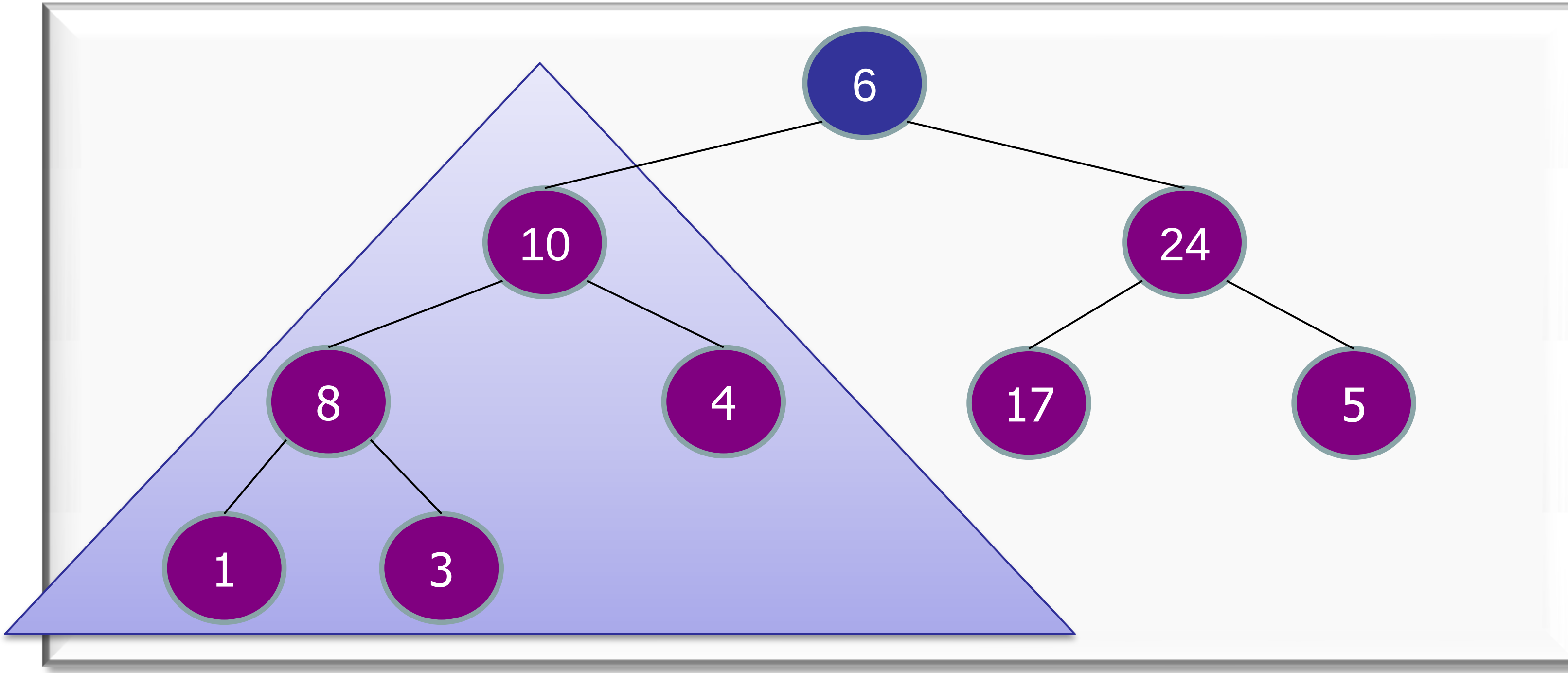


HeapSort

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**

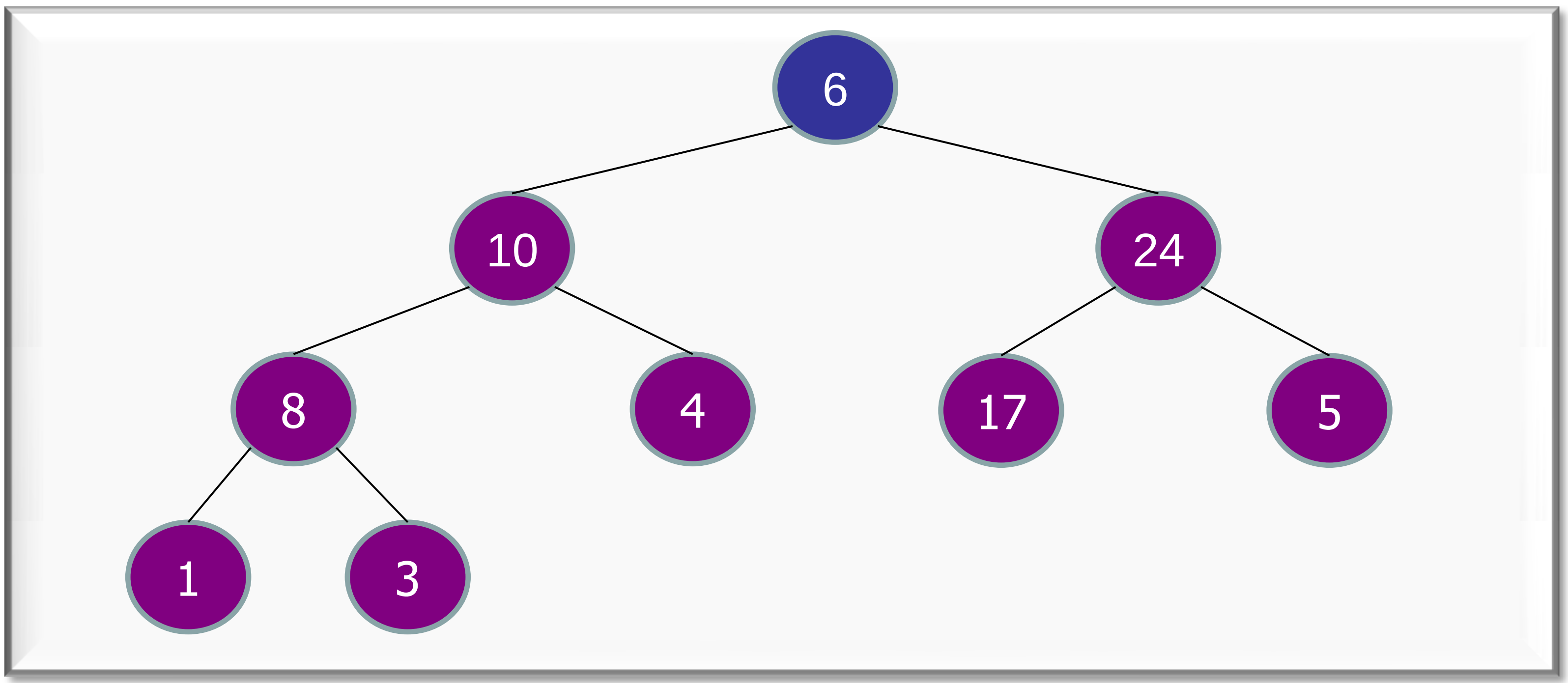


HeapSort

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

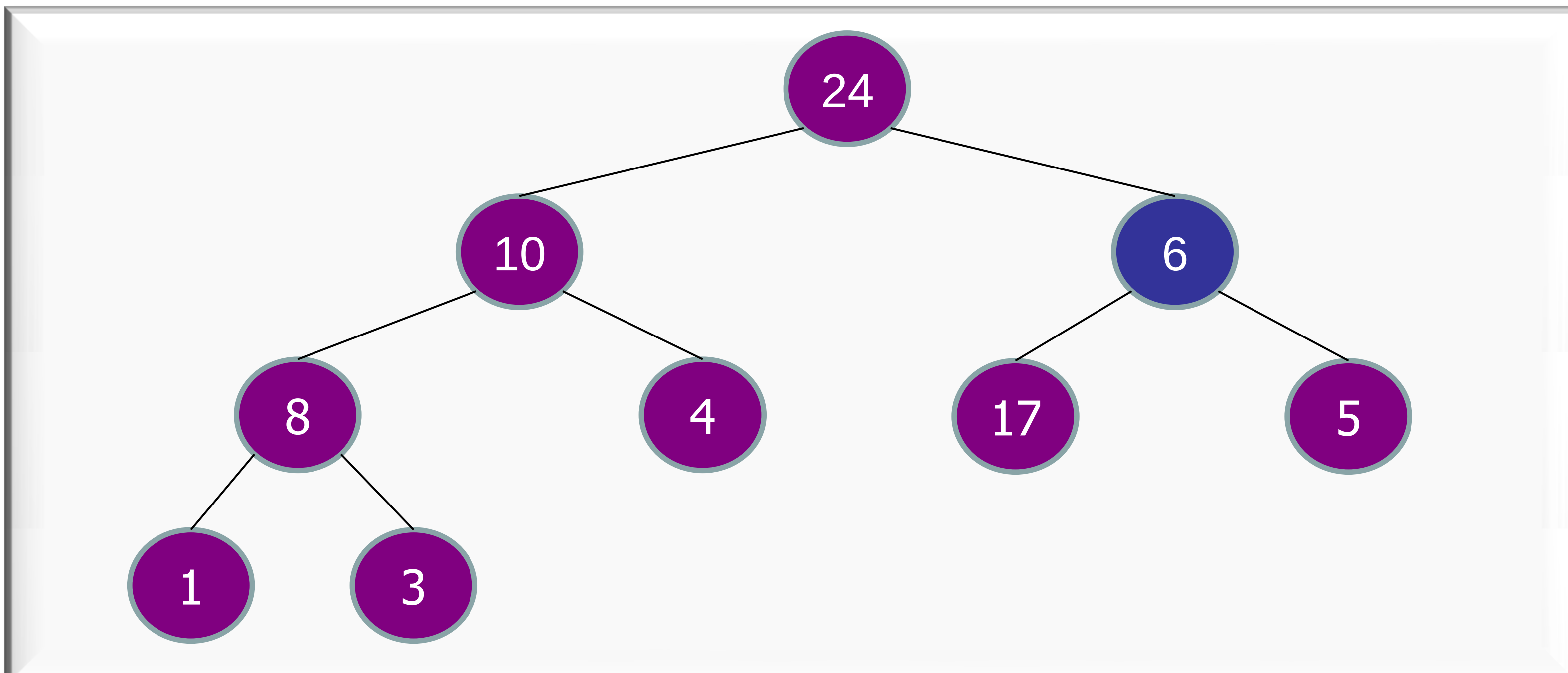


HeapSort

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

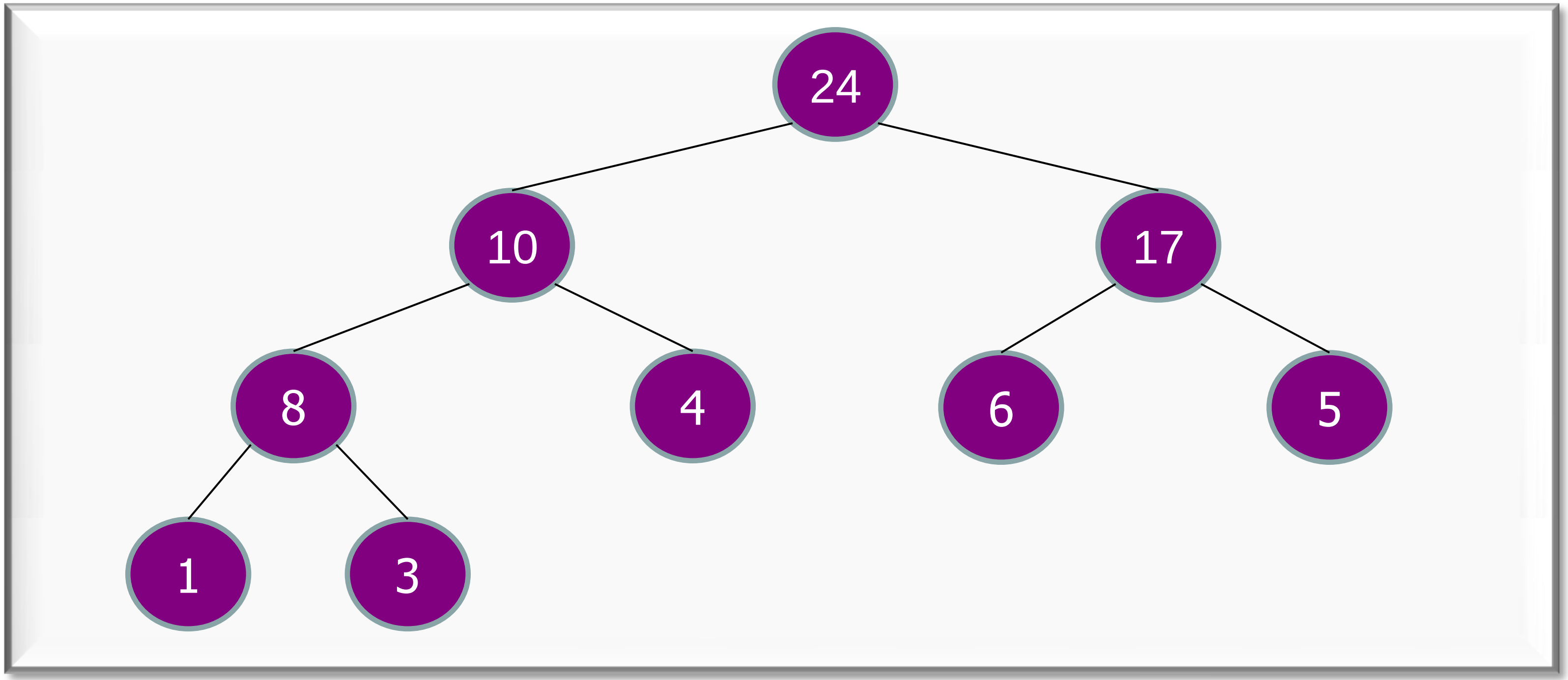


HeapSort

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

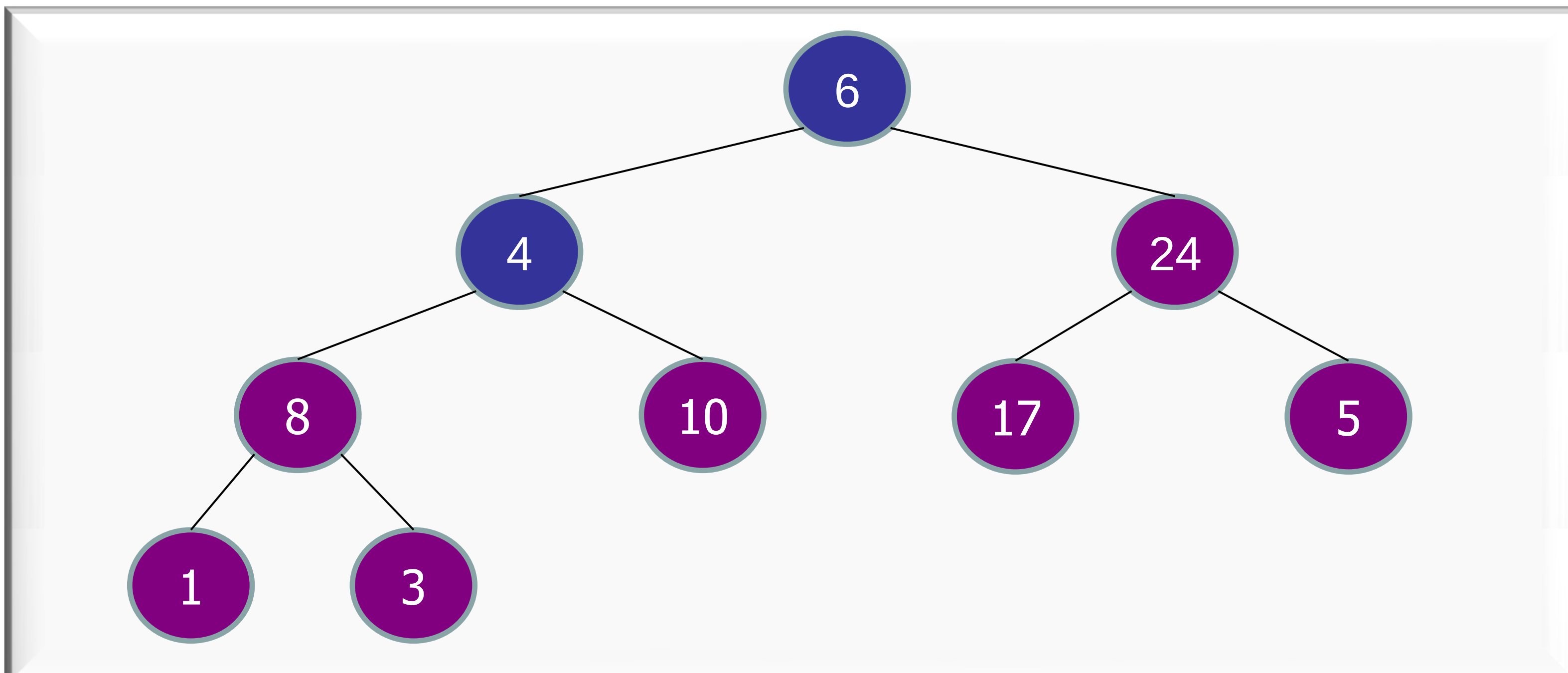
array slot	0	1	2	3	4	5	6	7	8
key	24	10	17	8	4	6	5	1	3

```
// 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

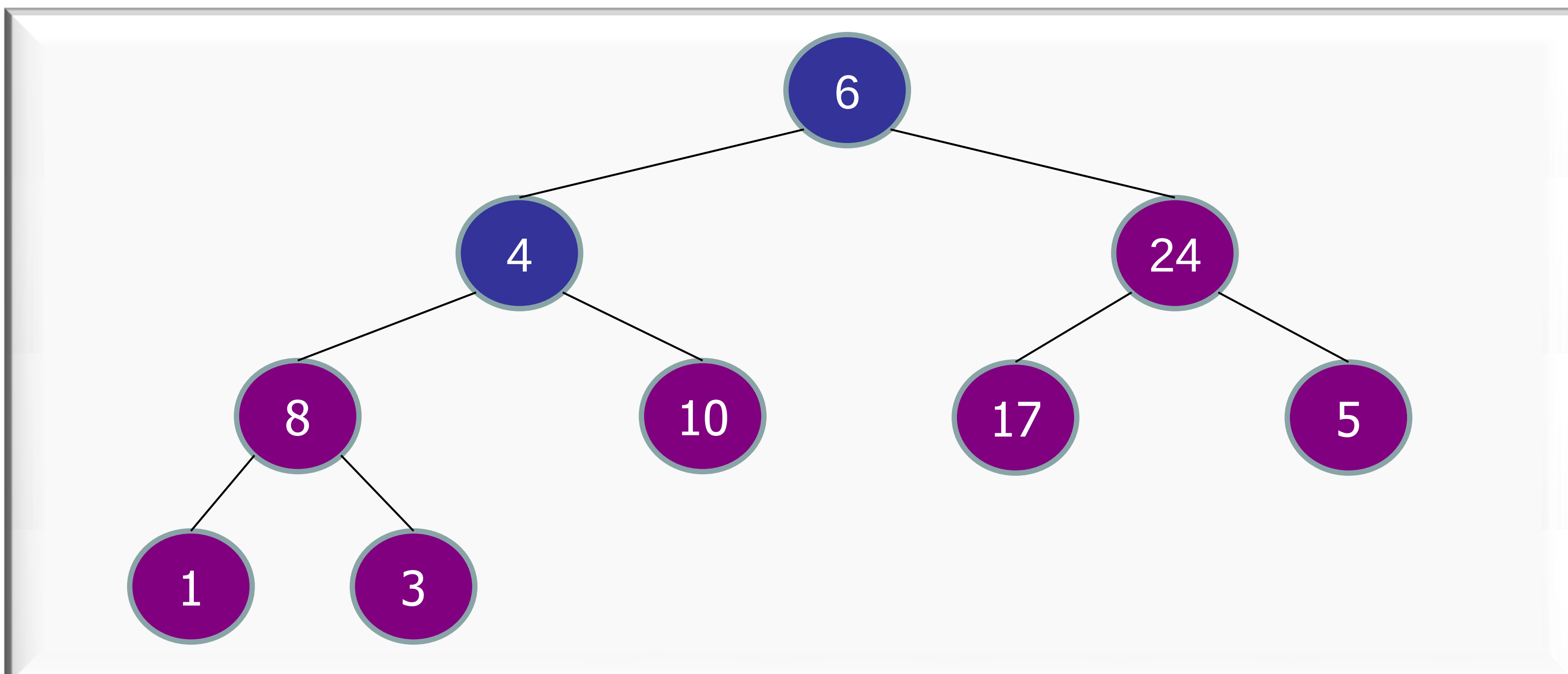
array slot	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
key	**6**	**4**	**24**	**8**	**10**	**17**	**5**	**1**	**3**



HeapSort

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

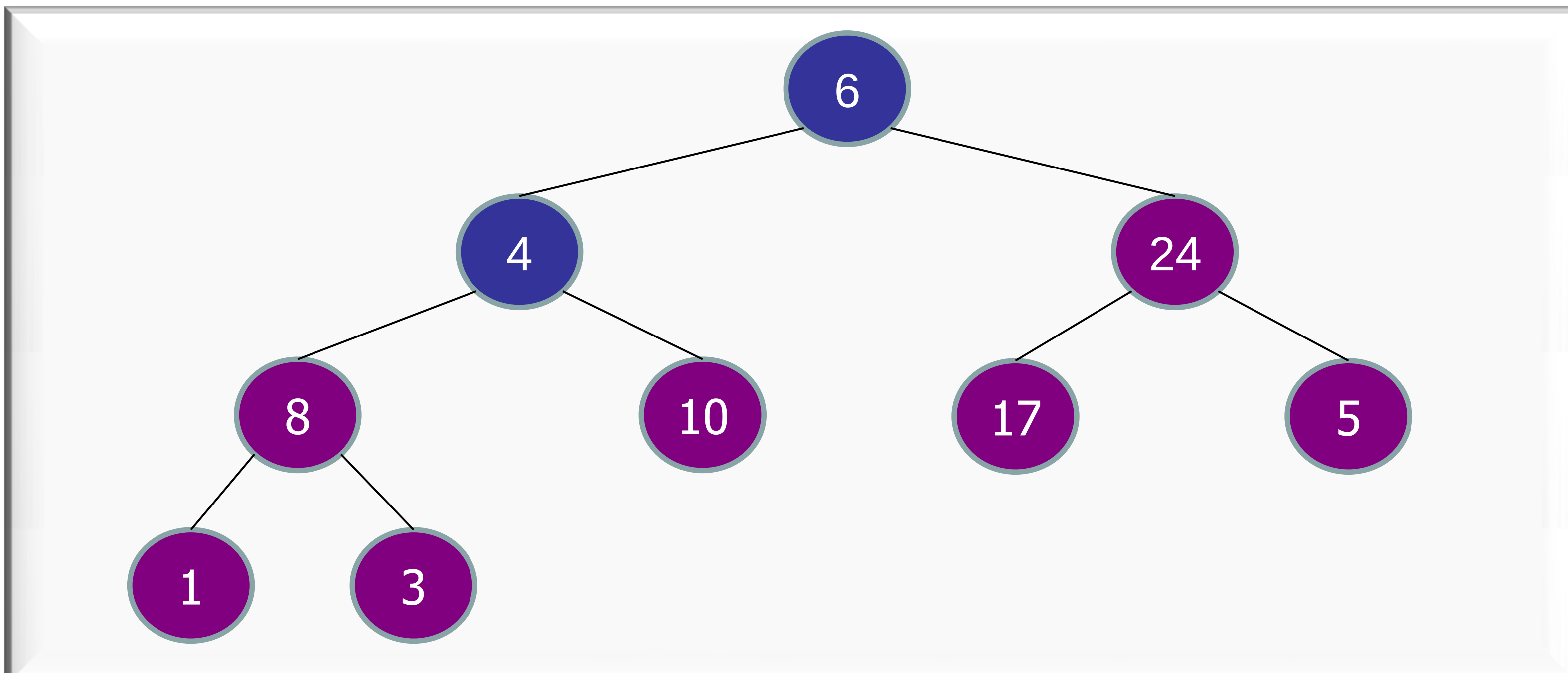
array slot	0	1	2	3	4	5	6	7	8
key	6	4	24	8	10	17	5	1	3



HeapSort

Observation: most nodes have small height!

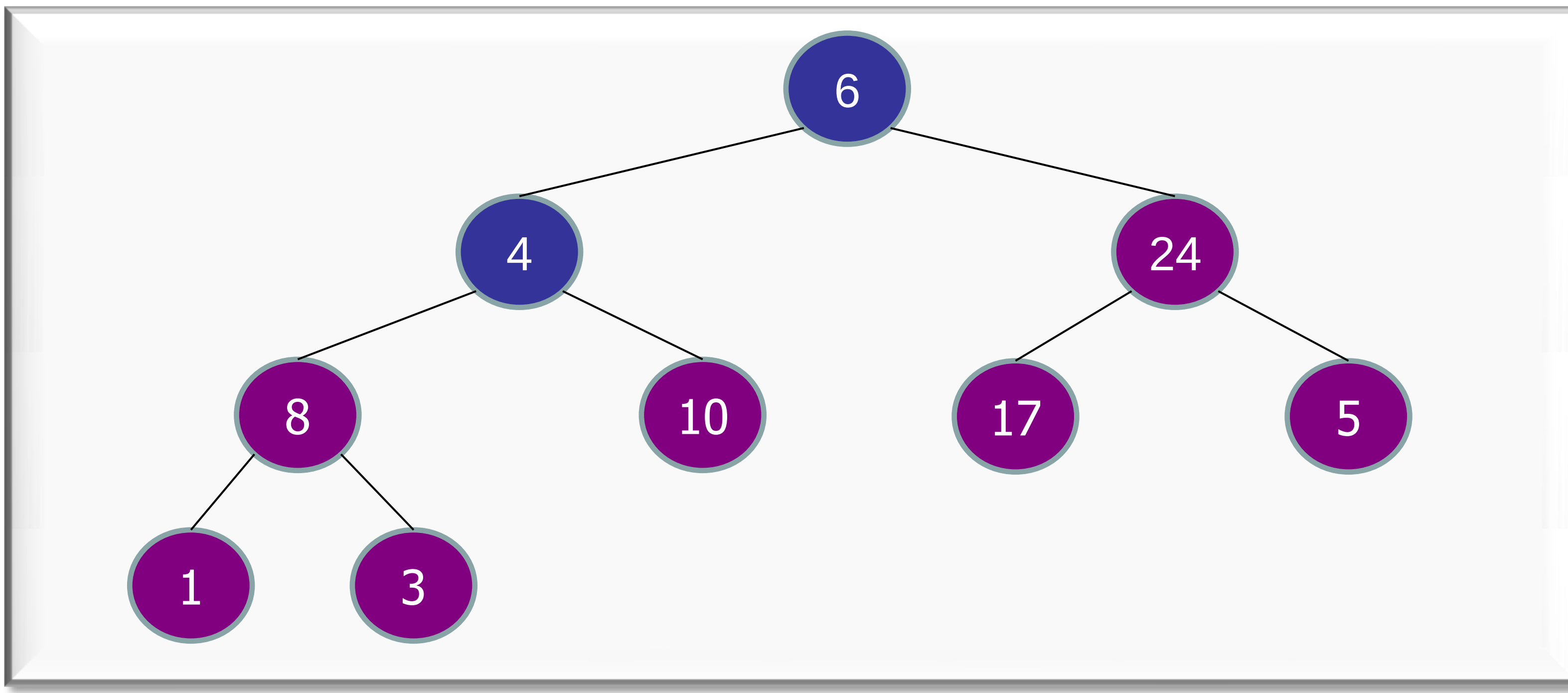
array slot	0	1	2	3	4	5	6	7	8
key	6	4	24	8	10	17	5	1	3



HeapSort

Cost of building a heap:

Height	**0**	**1**	**2**	**3**	**…**	**$\lfloor \log(n) \rfloor$**
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**	**...**	**⌊ log(n) ⌋**
Number	**⌈n/2⌉**	**⌈n/4⌉**	**⌈n/8⌉**	**⌈n/16⌉**	**...**	**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**	**...**	**⌊ log(n) ⌋**
Number	**⌈n/2⌉**	**⌈n/4⌉**	**⌈n/8⌉**	**⌈n/16⌉**	**...**	**1**

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

$$\leq cn\left(\frac{0.5}{(1-0.5)^2}\right) \leq 2O(n)$$

?

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

$$\downarrow$$

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

Geometric series

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

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

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

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

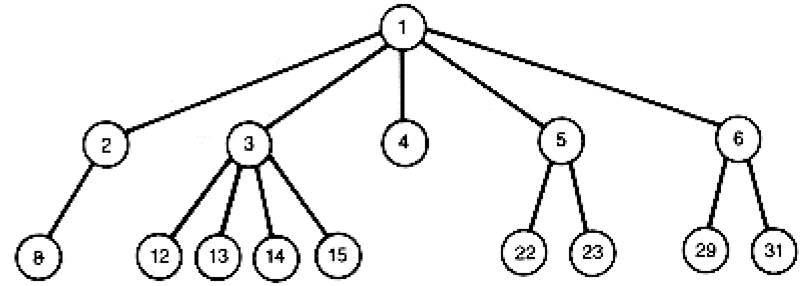
Unsorted list → Heap: O(n)

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.

Roadmap

Part I: Priority Queues

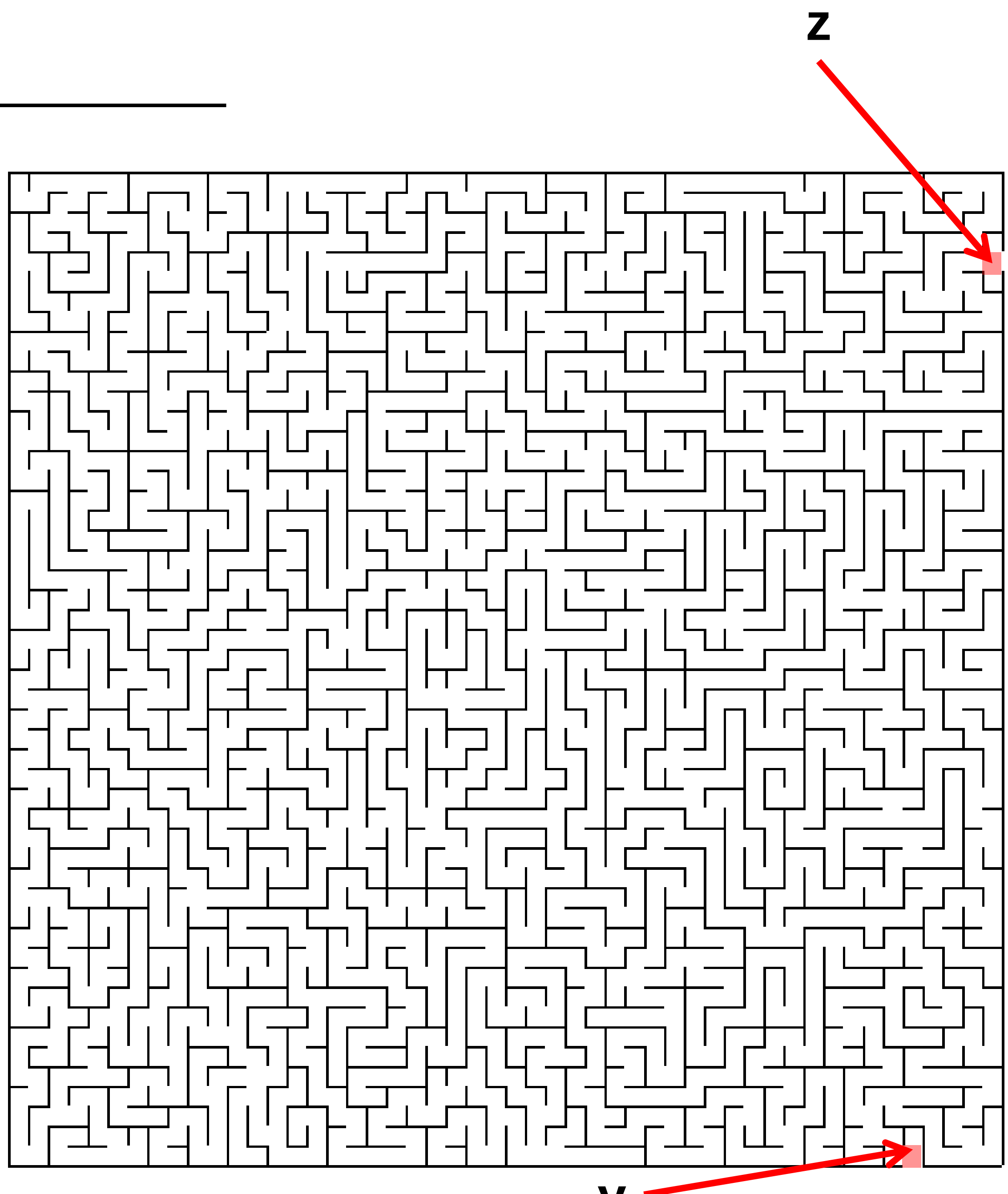
- Binary Heaps
- HeapSort

Part II: Disjoint Set

- Problem: Dynamic Connectivity
- Algorithm: Union-Find
- Applications

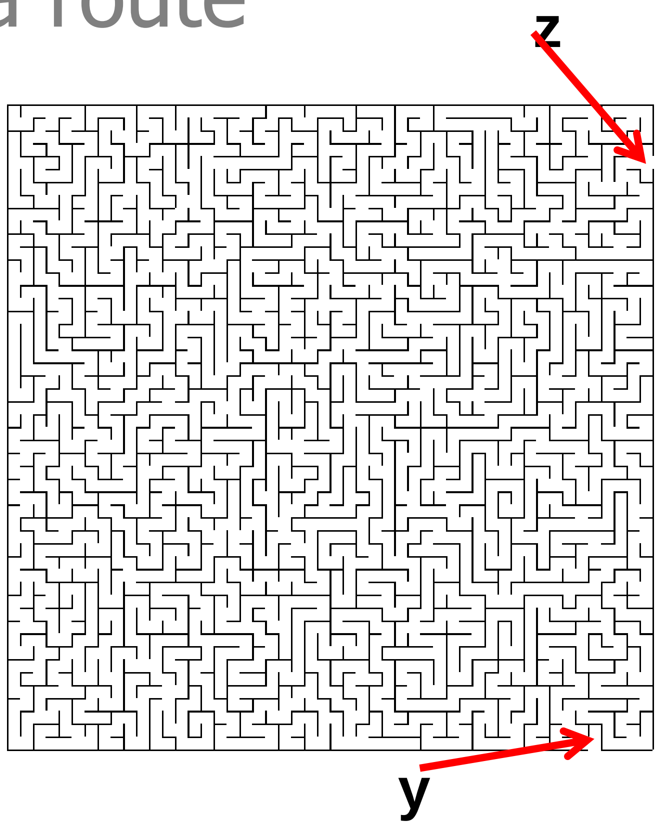
Mazes

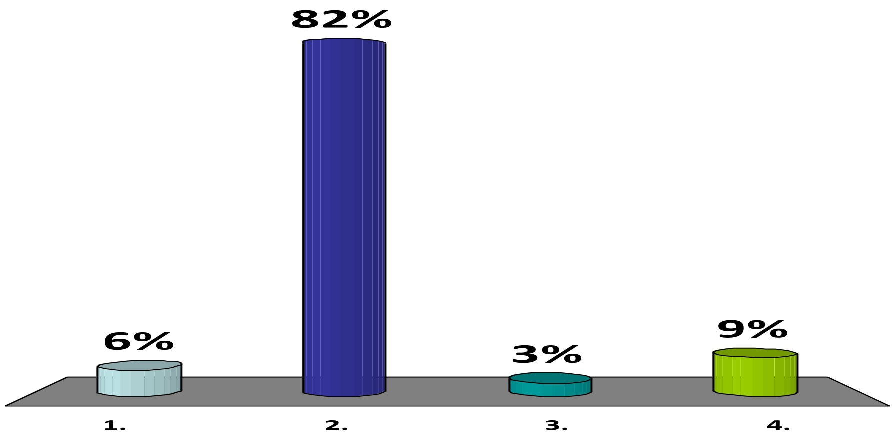
Is there any route from y to z?



Best way to find if there is a route from Y to Z?

1. Breadth-first search
2. Depth-first search
3. Topological sort
4. Quicksort





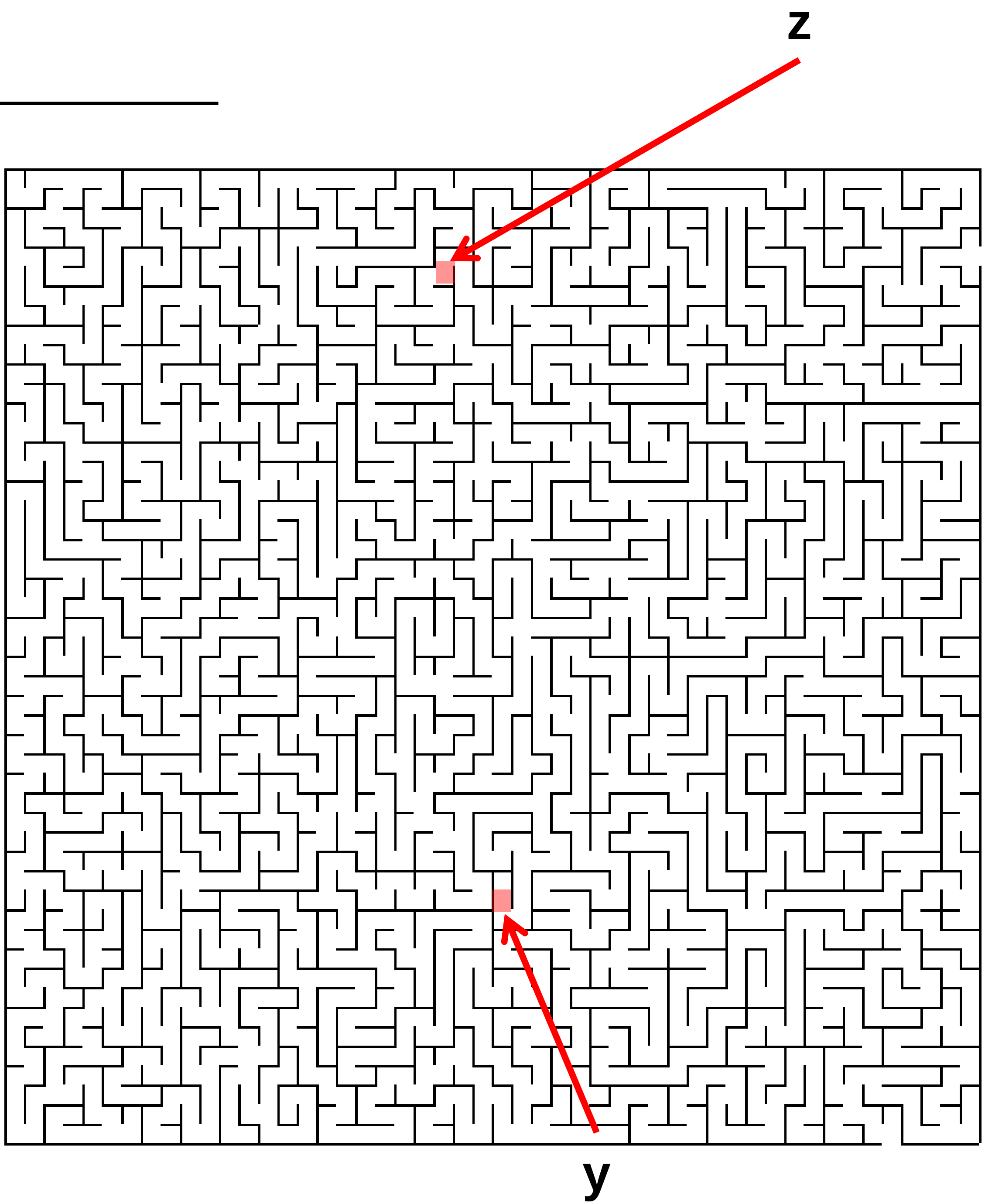
Mazes

Two steps:

1. Pre-process maze
2. Answer queries

isConnected(y,z) :

Returns true if there is a path from A to B, and false otherwise.



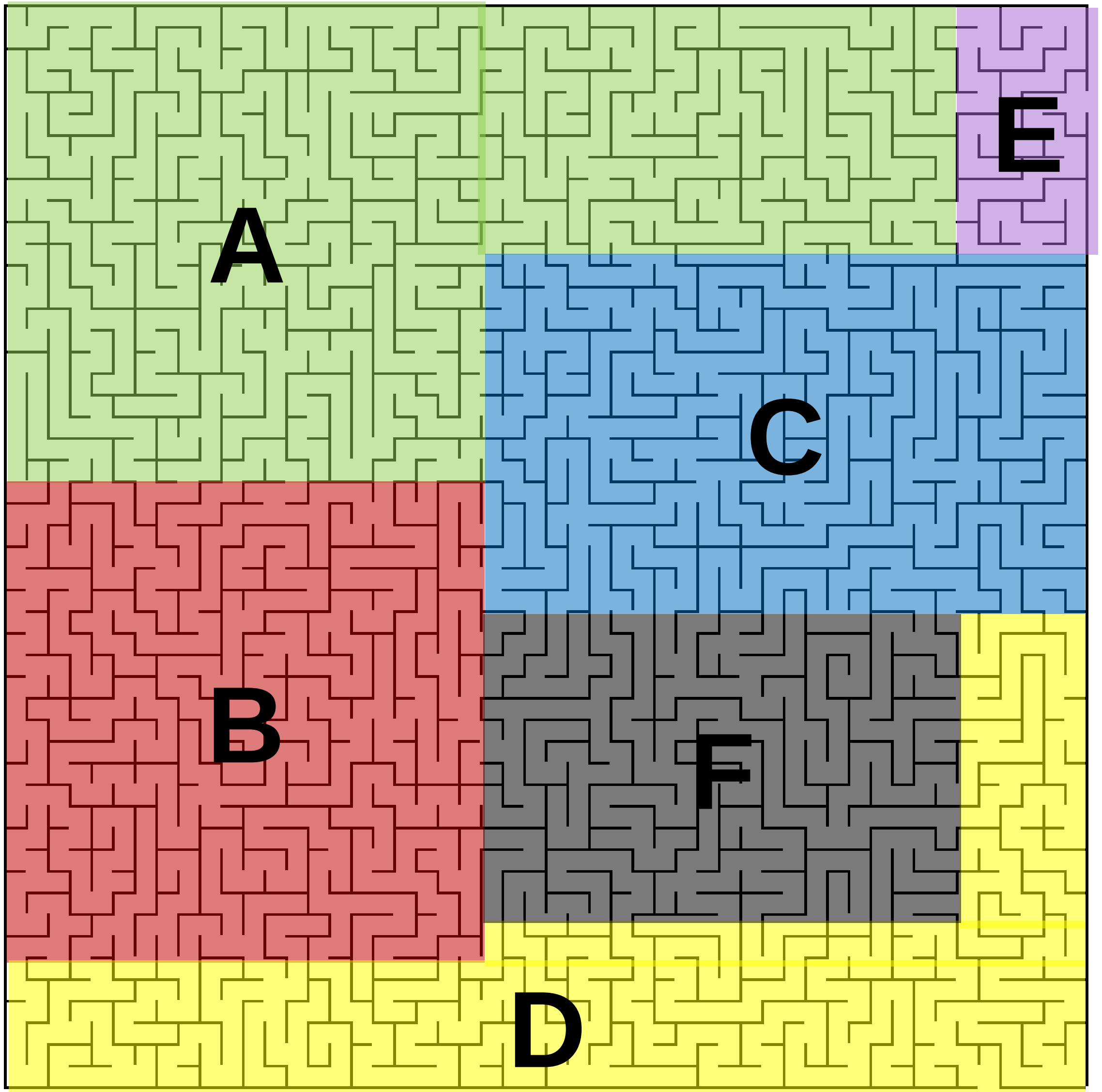
Mazes

Preprocess:

Identify connected components. Label each location with its component number.

isConnected(y,z) :

Returns true if A and B are in the same connected component.



Mazes

Preprocess:

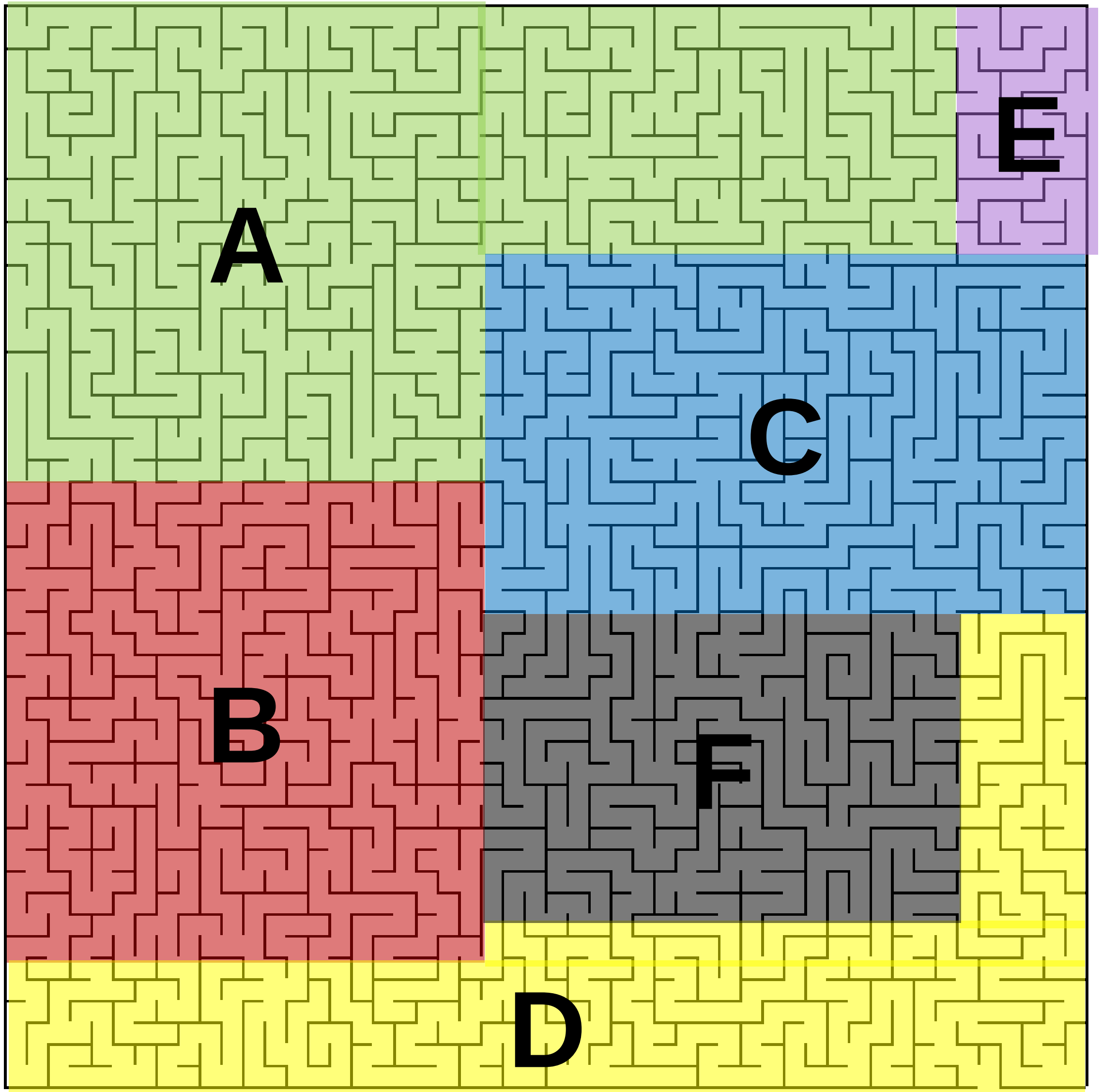
Prepare to answer queries.

destroyWall(x):

Remove walls from the maze using your superpowers.

isConnected(y, z):

Answer connectivity queries.



Mazes

Preprocess:

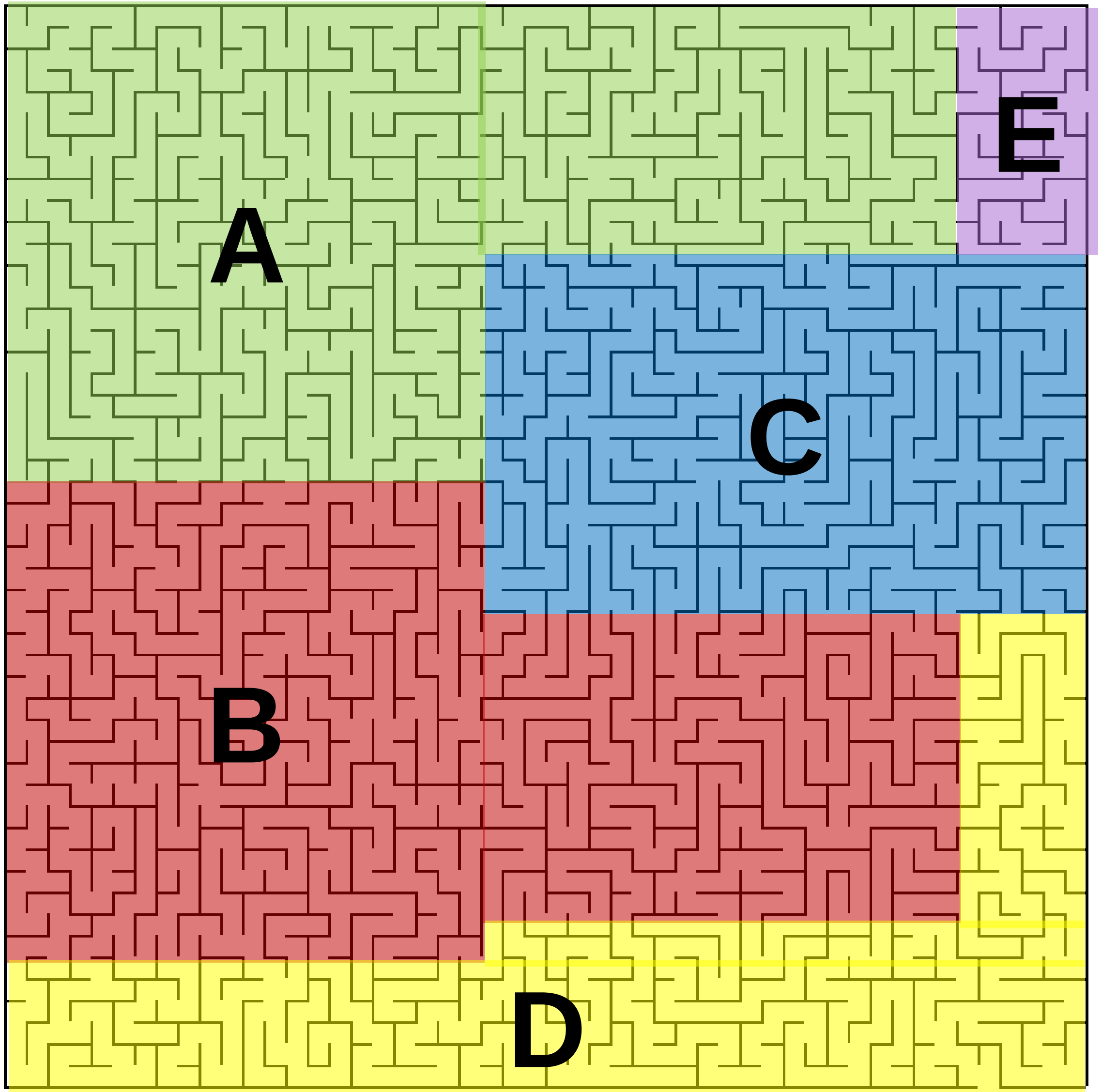
Prepare to answer queries.

destroyWall(x):

Remove walls from the maze using your superpowers.

isConnected(y, z):

Answer connectivity queries.

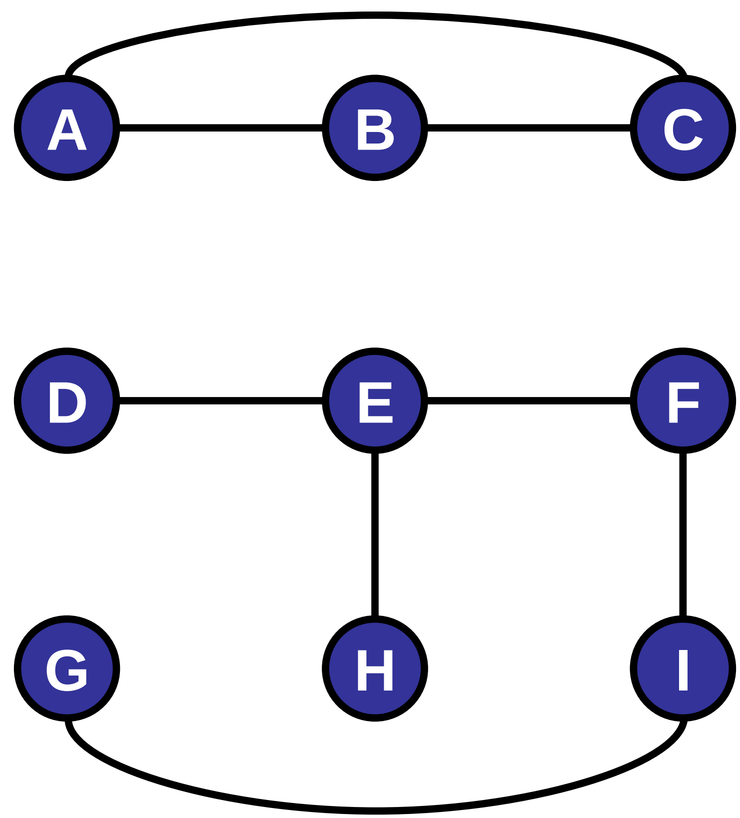


Dynamic Connectivity

Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

union(E, F)
union(I, G)
union(D, E)
union(B, A)
find(G, D) = **false**
find(D, F) = **true**
union(B, C)
union(H, E)
union(A, C)
union(F, I)
find(G, D) = **true**



Dynamic Connectivity

Given a set of objects:

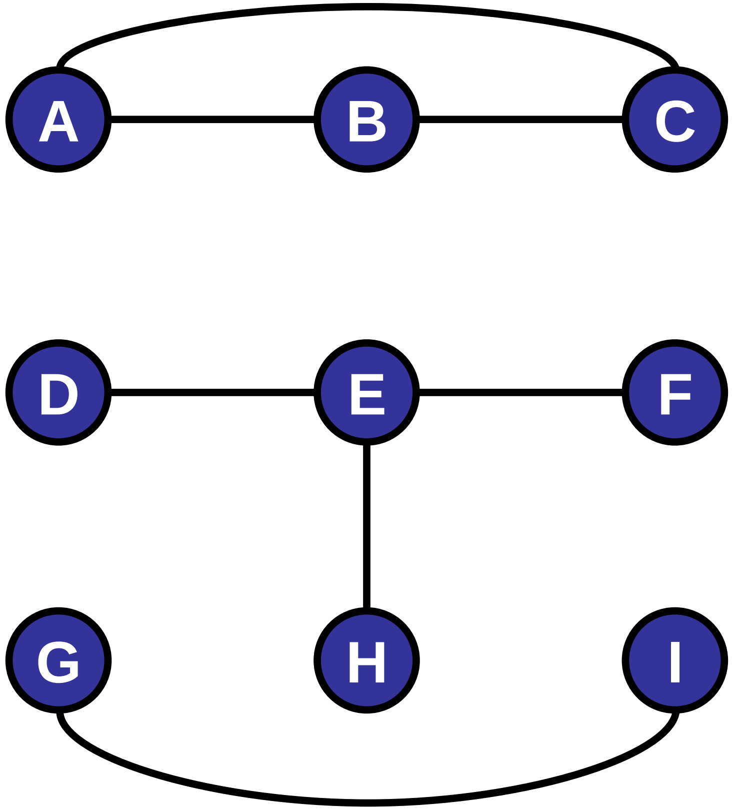
- Union: connect two objects
- Find: is there a path connecting the two objects?

Transitivity

- If **p** is connected to **q** and if **q** is connected to **r**, then **p** is connected to **r**.

Connected components:

- Maximal set of mutually connected objects.



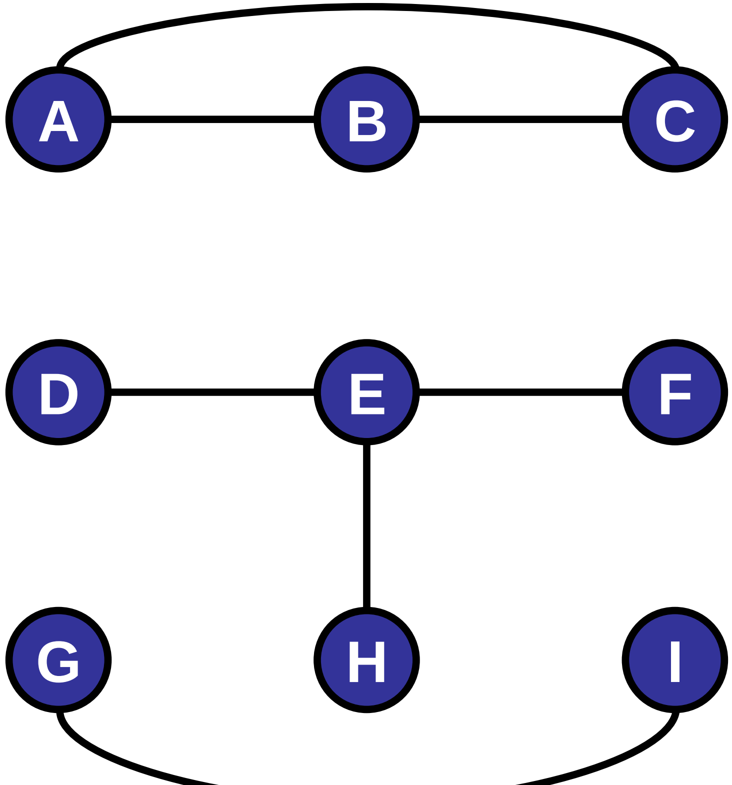
Dynamic Connectivity

Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

Maintain sets of connected components:

{A, B, C}
{D, E, F, H}
{G, I}



Dynamic Connectivity

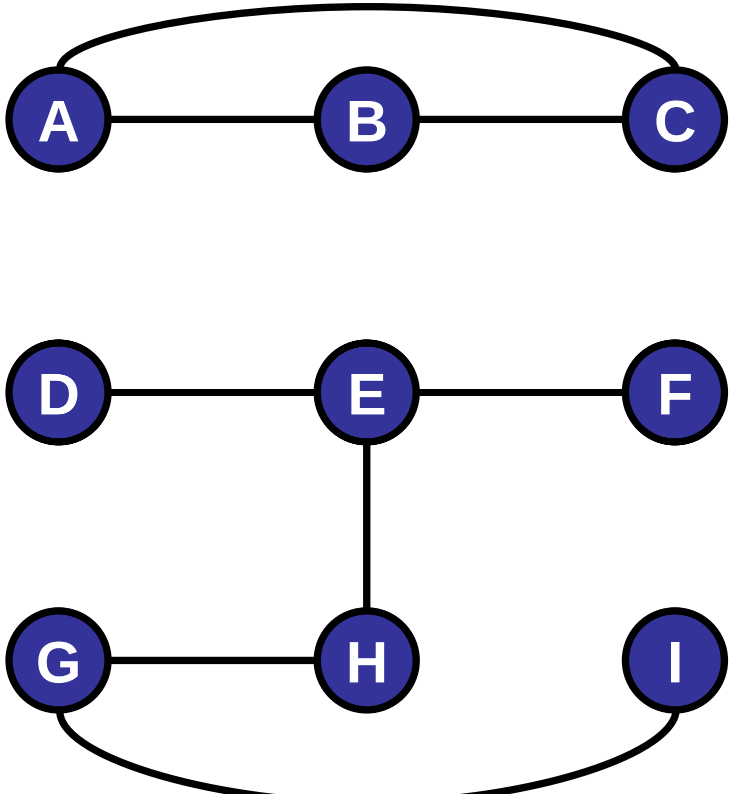
Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

Maintain sets of connected components:

{A, B, C}

{D, E, F, H, G, I}



Abstract Data Type

Disjoint Set (Union-Find)

```
public interface  DisjointSet<Key>
```

	`DisjointSet(int N)`	*constructor: N objects*
`boolean`	`find(Key p, Key q)`	*are p and q in the same set?*
`void`	`union(Key p, Key q)`	*replace sets containing p and q with their union*

Roadmap

Part II: Disjoint Set

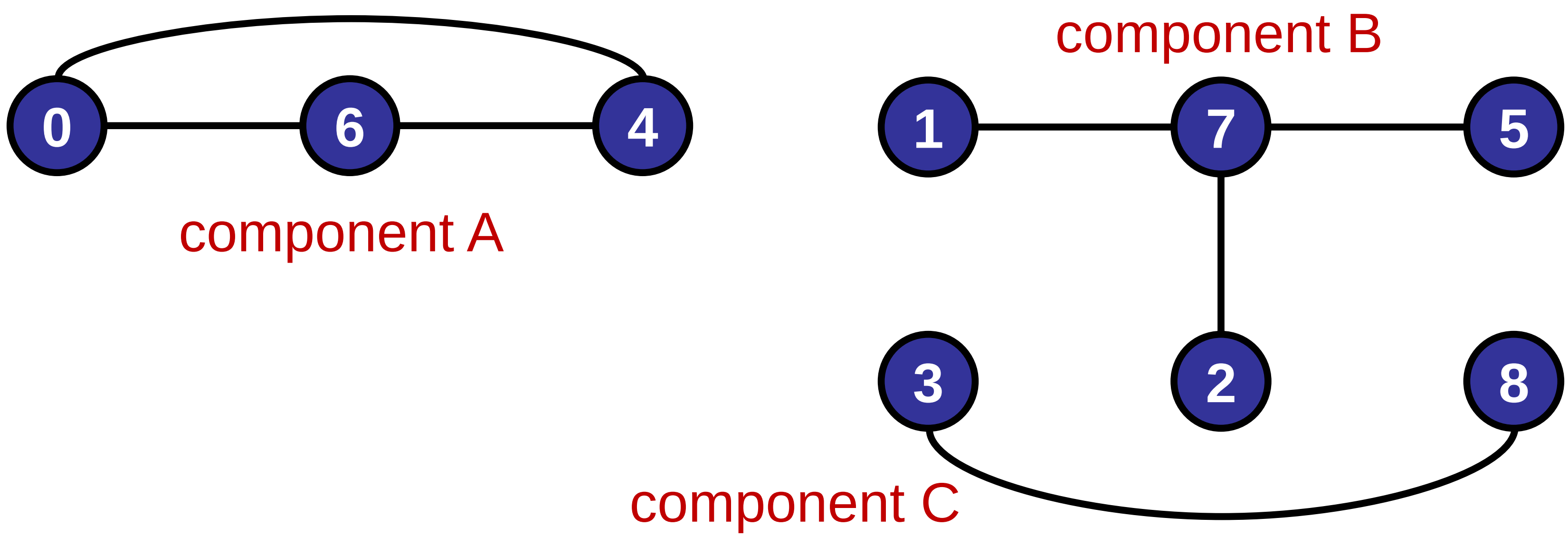
- Problem: Dynamic Connectivity
- Algorithm: Quick-Find
- Algorithm: Quick-Union
- Optimizations
- Applications

Quick Find

Data structure:

- Array: componentId
- Two objects are connected if they have the same component identifier.

object	0	1	2	3	4	5	6	7	8
component identifier	A	B	B	C	A	B	A	B	C



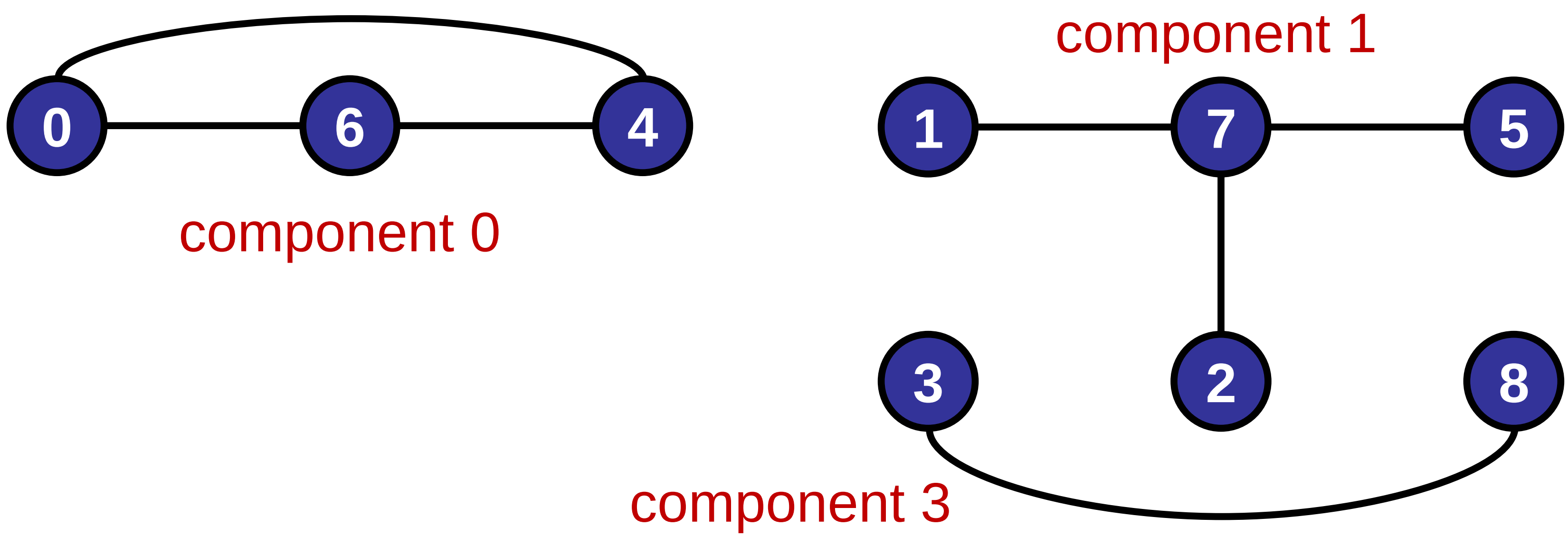
Quick Find

Assume objects are integers

Data structure:

- Integer array: int[] componentId
- Two objects are connected if they have the same component identifier.

object	0	1	2	3	4	5	6	7	8
component identifier	0	1	1	3	0	1	0	1	3

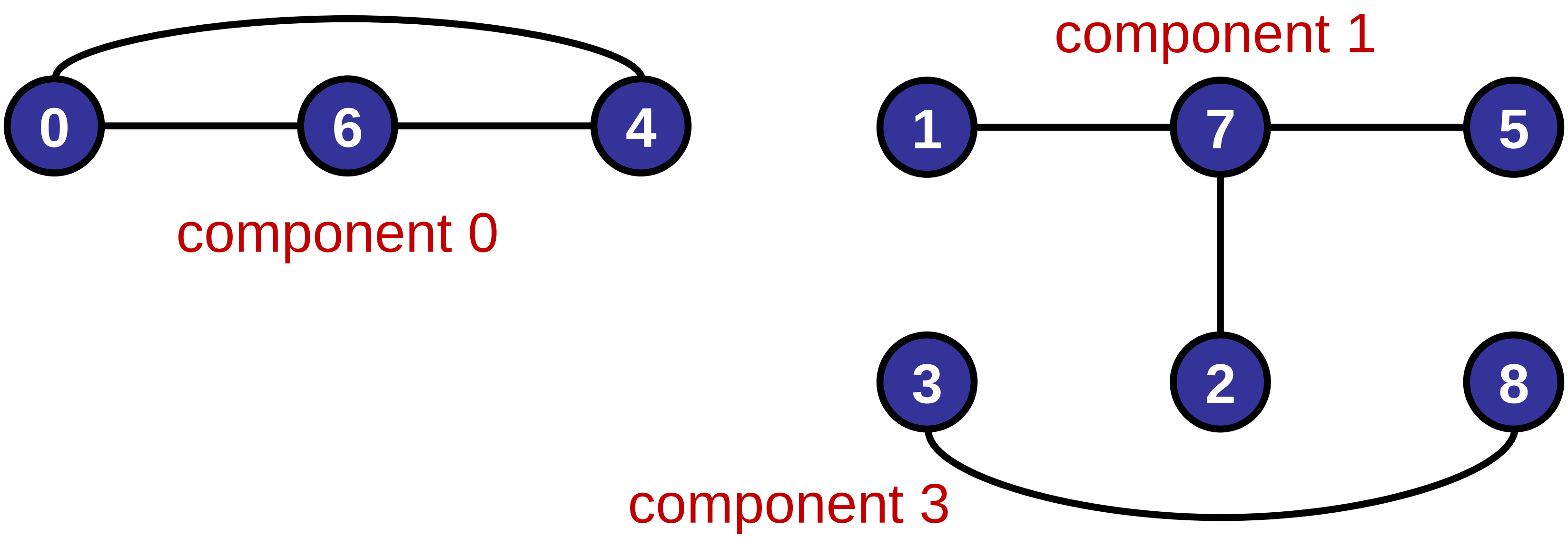


Quick Find

Data structure:

- Integer array: int[] componentId
- Two objects are connected if they have the same component identifier.

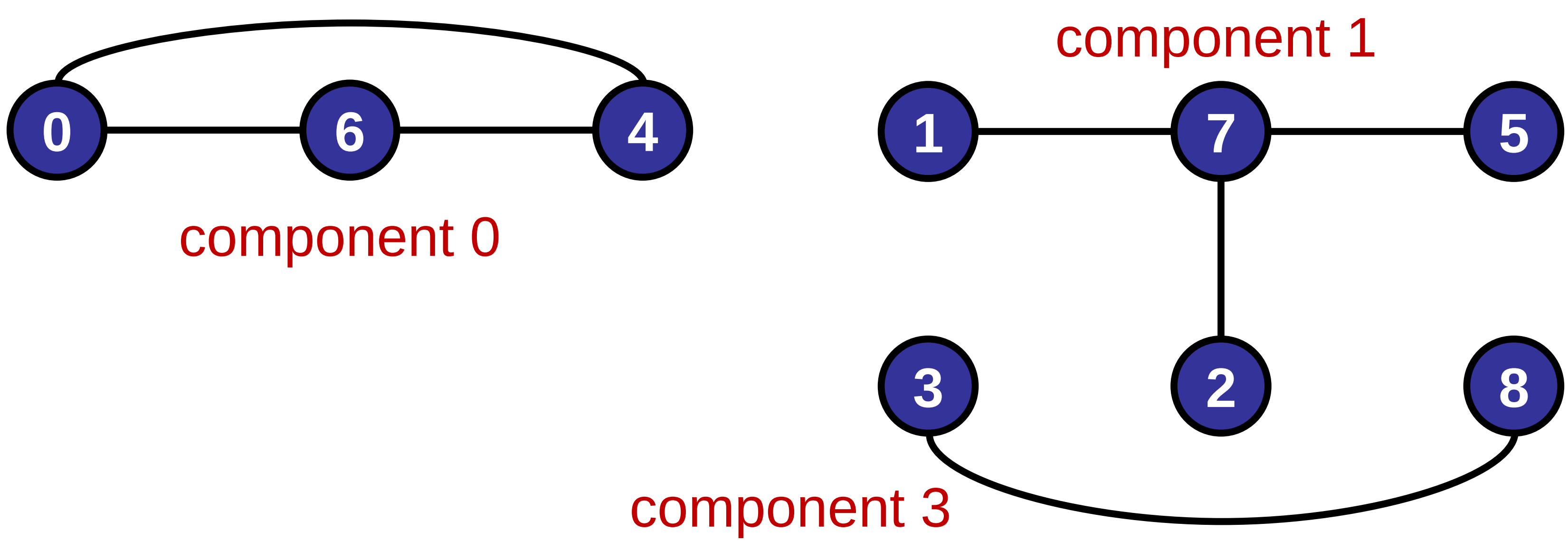
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	1	3	0	1	0	1	3



Quick Find

```
find(int p, int q)
  return(componentId[p] == componentId[q]);
```

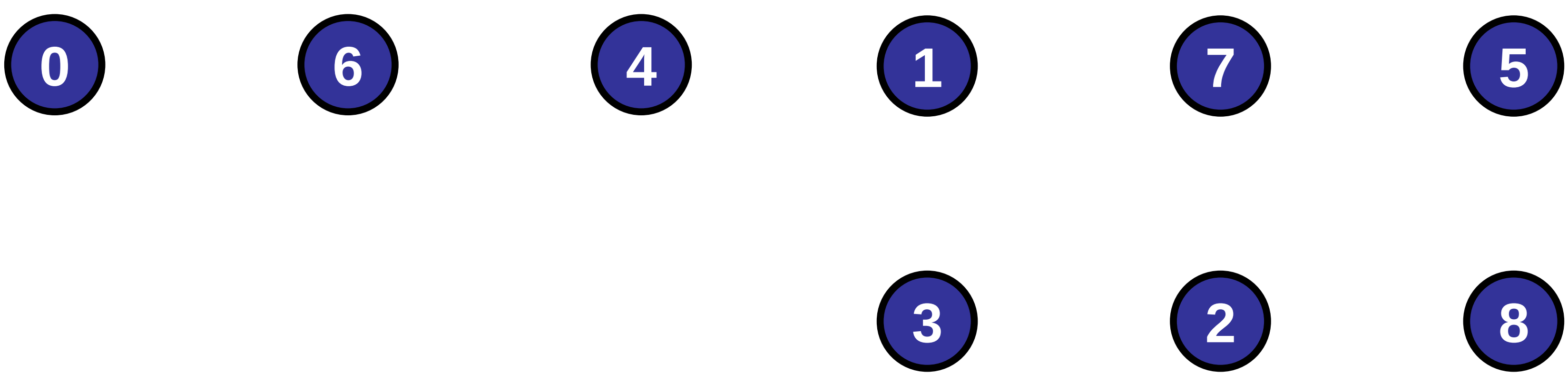
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	1	3	0	1	0	1	3



Quick Find

Initial state of data structure:

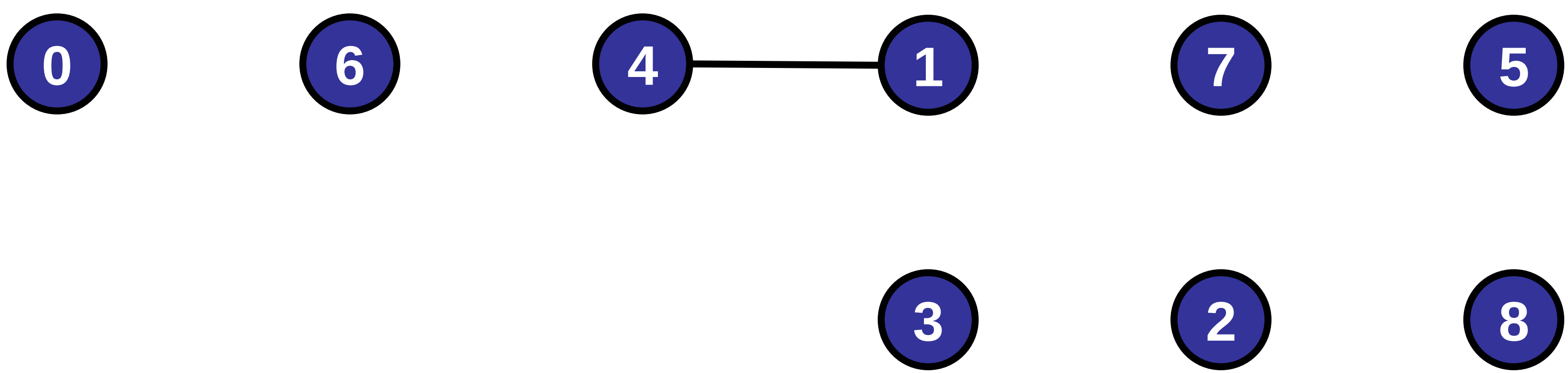
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	2	3	4	5	6	7	8



Quick Find

```
union(int p, int q)
   for (int i=0; i<componentId.length; i++)
           if (componentId[i] == componentId[q])
                componentId[i] = componentId[p];
```

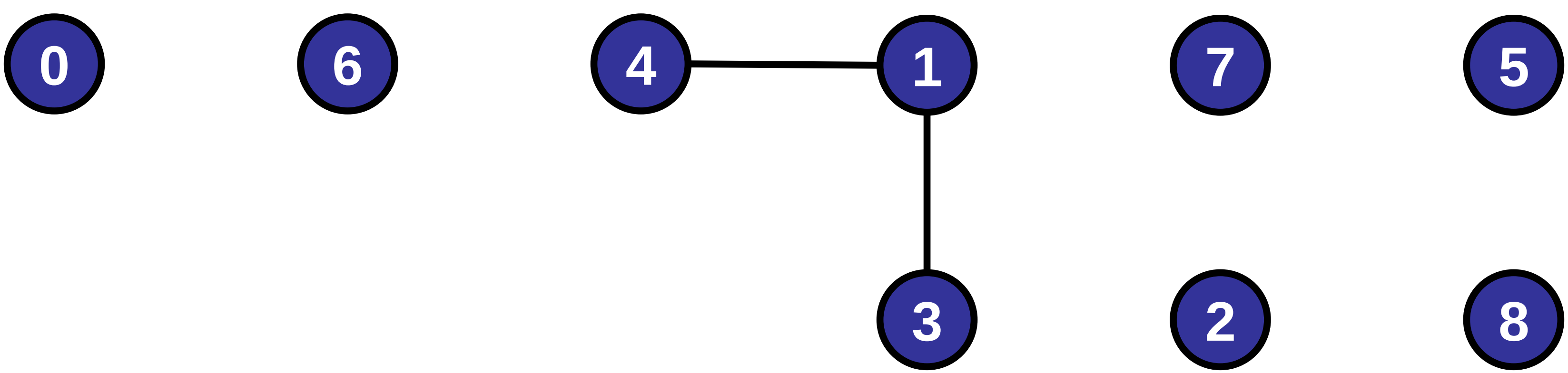
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	2	3	1	5	6	7	8



Quick Find

```
union(int p, int q)
  for (int i=0; i<componentId.length; i++)
        if (componentId[i] == componentId[q])
             componentId[i] = componentId[p];
```

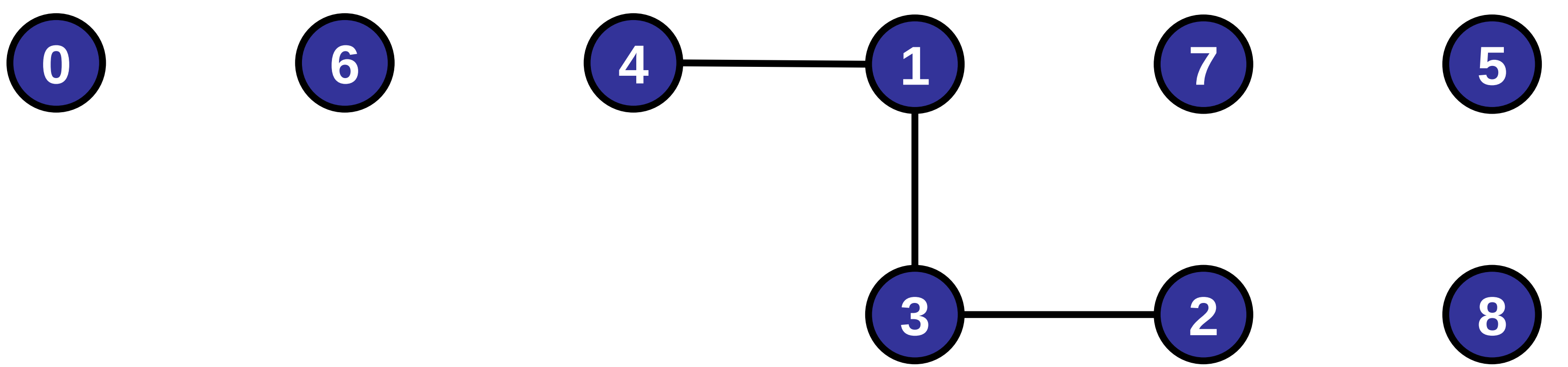
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	2	1	1	5	6	7	8



Quick Find

```
union(int p, int q)
   for (int i=0; i<componentId.length; i++)
         if (componentId[i] == componentId[q])
              componentId[i] = componentId[p];
```

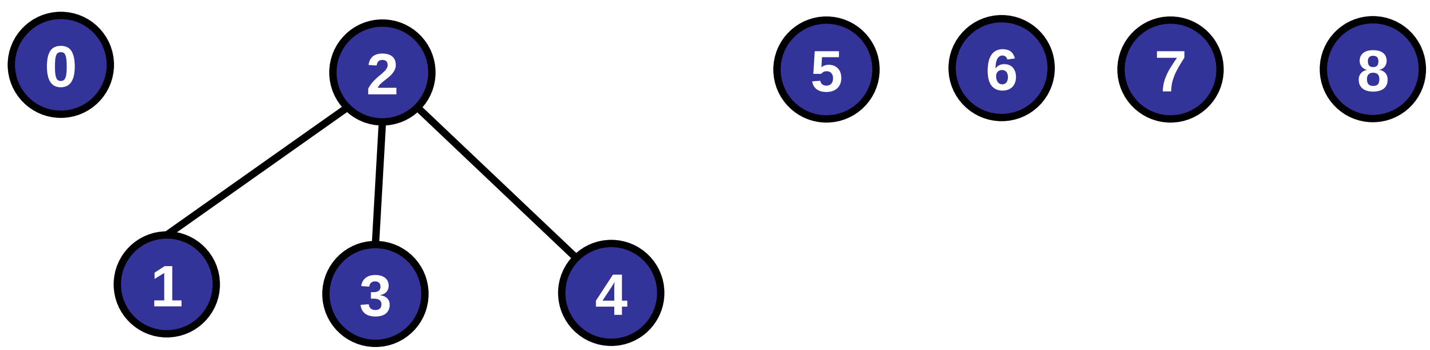
object	0	1	2	3	4	5	6	7	8
component identifier	0	2	2	2	2	5	6	7	8



Quick Find

Flat trees:

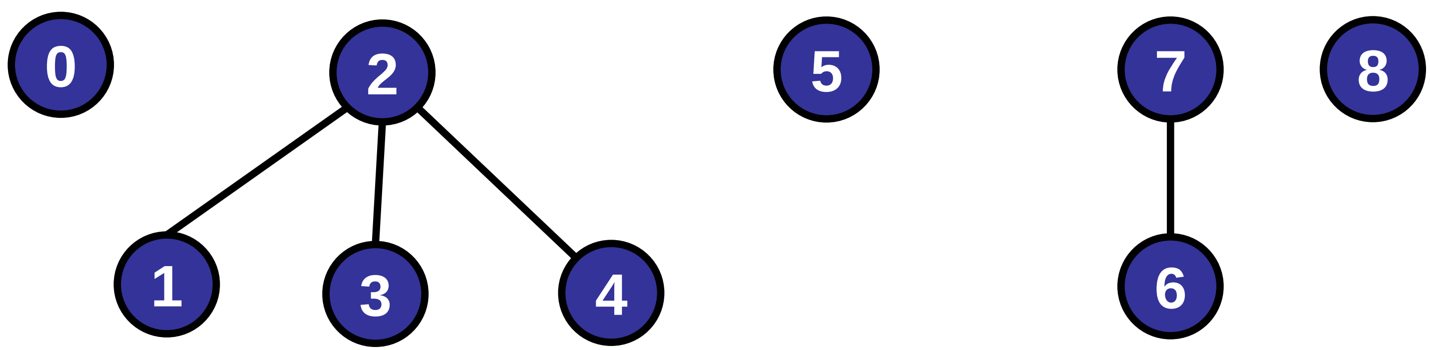
object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
component identifier	**0**	**2**	**2**	**2**	**2**	**5**	**6**	**7**	**8**



Quick Find

`Flat trees:`

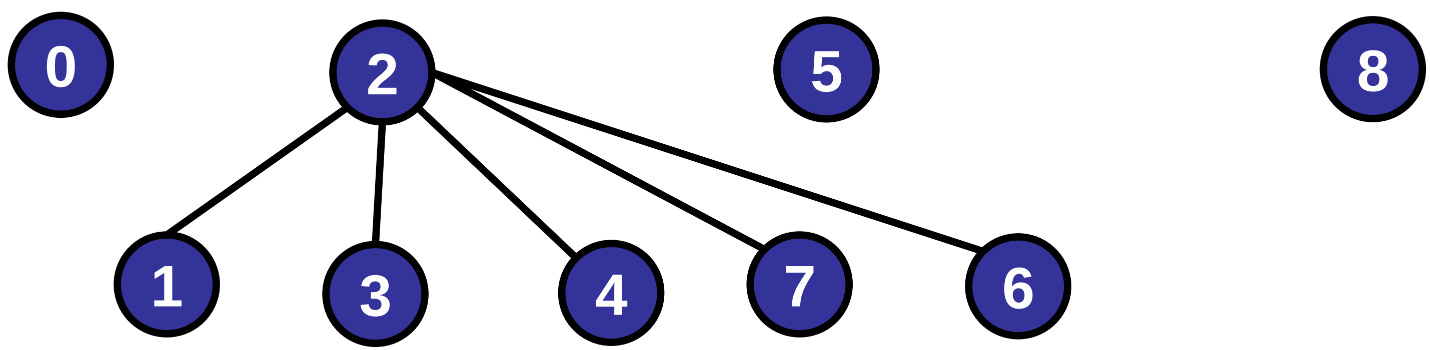
object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
component identifier	**0**	**2**	**2**	**2**	**2**	**5**	**7**	**7**	**8**



Quick Find

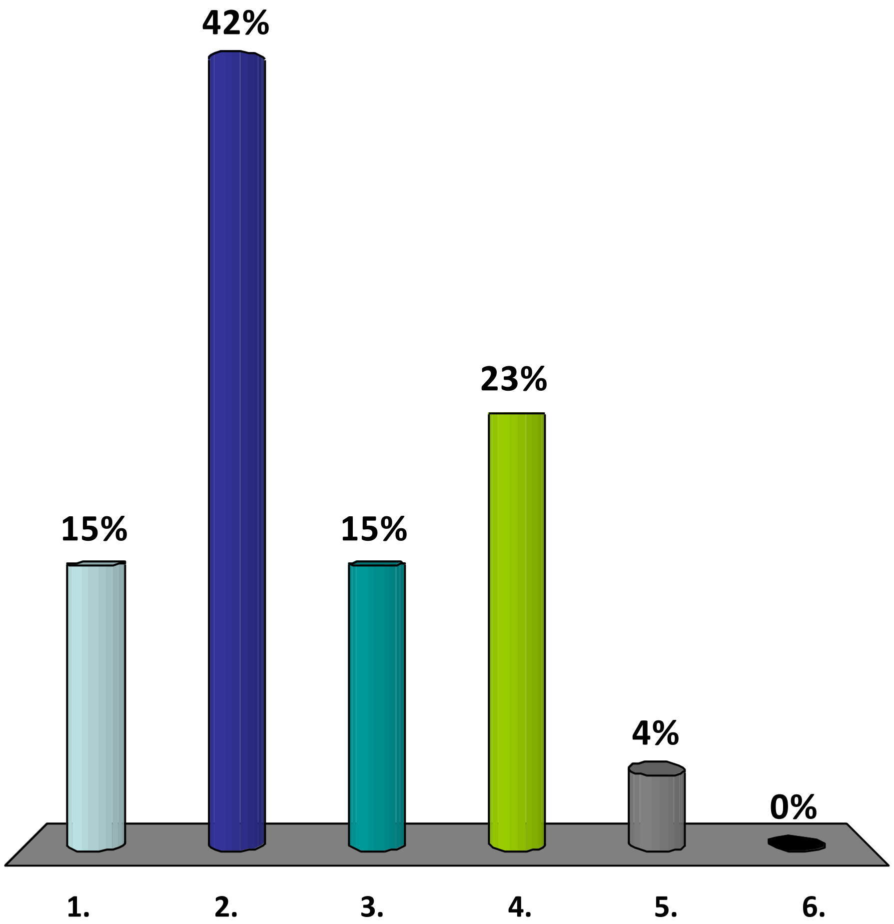
Flat trees:

object	0	1	2	3	4	5	6	7	8
component identifier	0	2	2	2	2	5	2	2	8



Running time of (Find, Union):

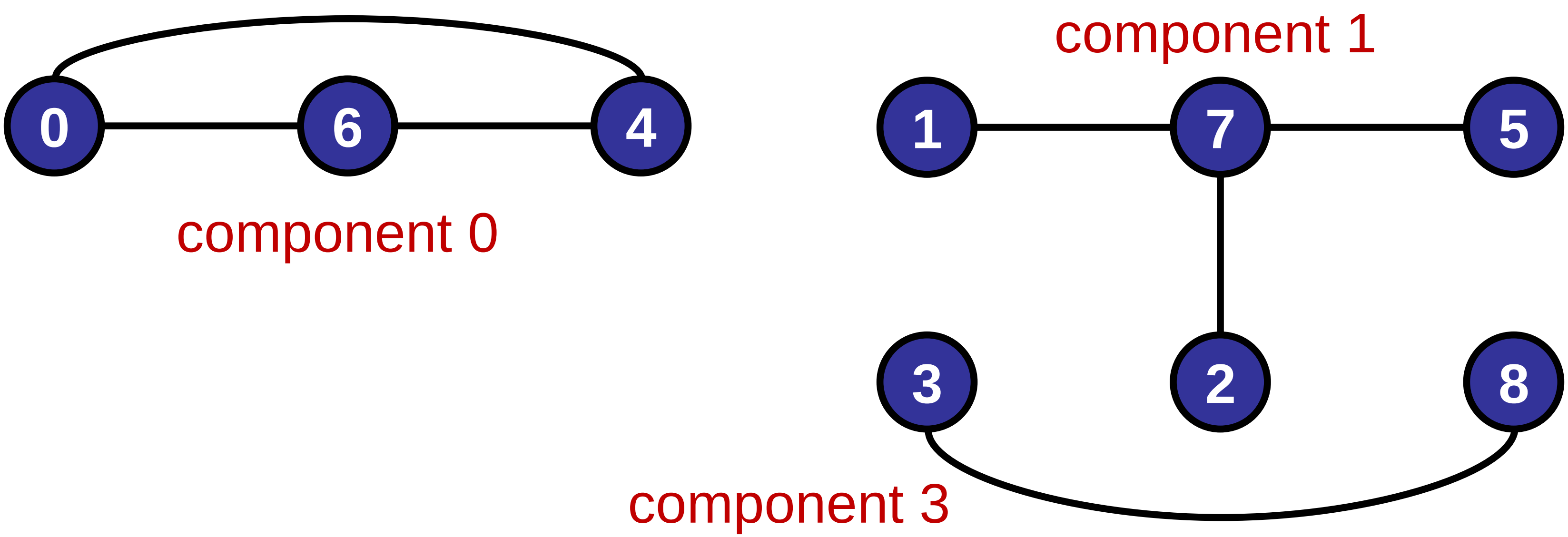
1. O(1), O(1)
2. O(1), O(n)
3. O(n), O(1)
4. O(n), O(n)
5. O(log n), O(log n)
6. None of the above.



Quick Find

```
find(int p, int q)
  return(componentId[p] == componentId[q]);
```

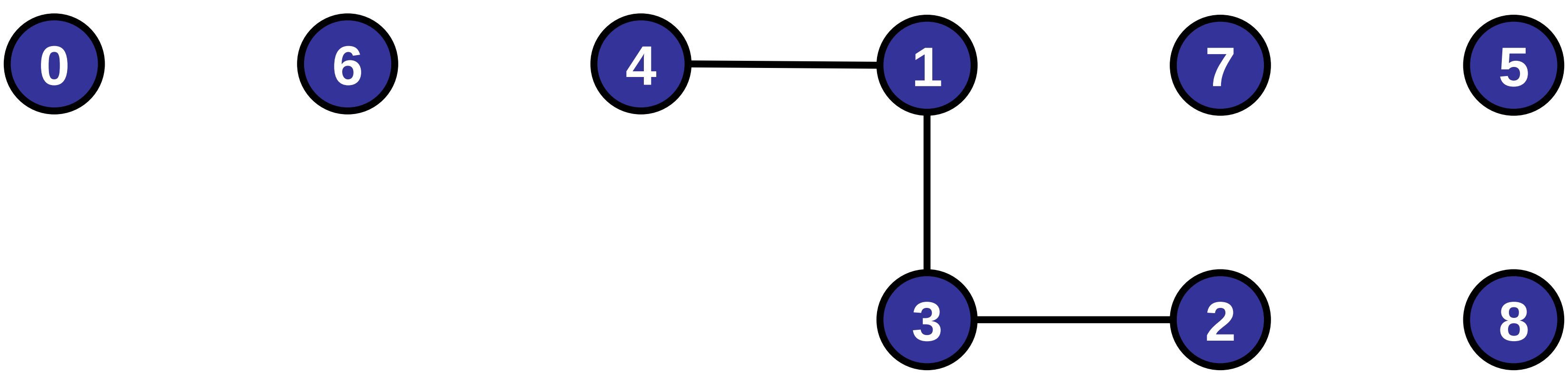
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	1	3	0	1	0	1	3



Quick Find

```
union(int p, int q)
   for (inti=0; i<componentId.length; i++)
           if (componentId[i] == componentId[q])
                componentId[i] = componentId[p];
```

object	0	1	2	3	4	5	6	7	8
component identifier	0	2	2	2	2	5	6	7	8



Roadmap

Part II: Disjoint Set

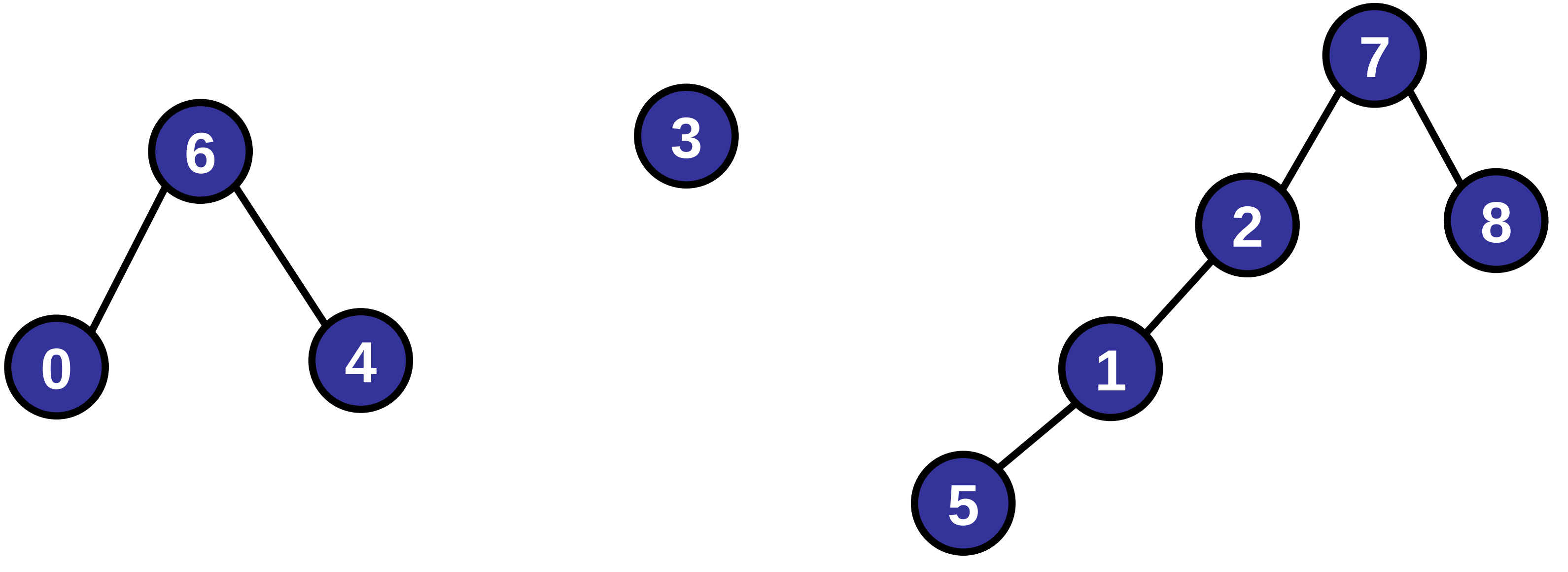
- Problem: Dynamic Connectivity
- Algorithm: Quick-Find
- Algorithm: Quick-Union
- Optimizations
- Applications

Quick Union

Data structure:

- Integer array: int[] parent
- Two objects are connected if they are part of the same tree.

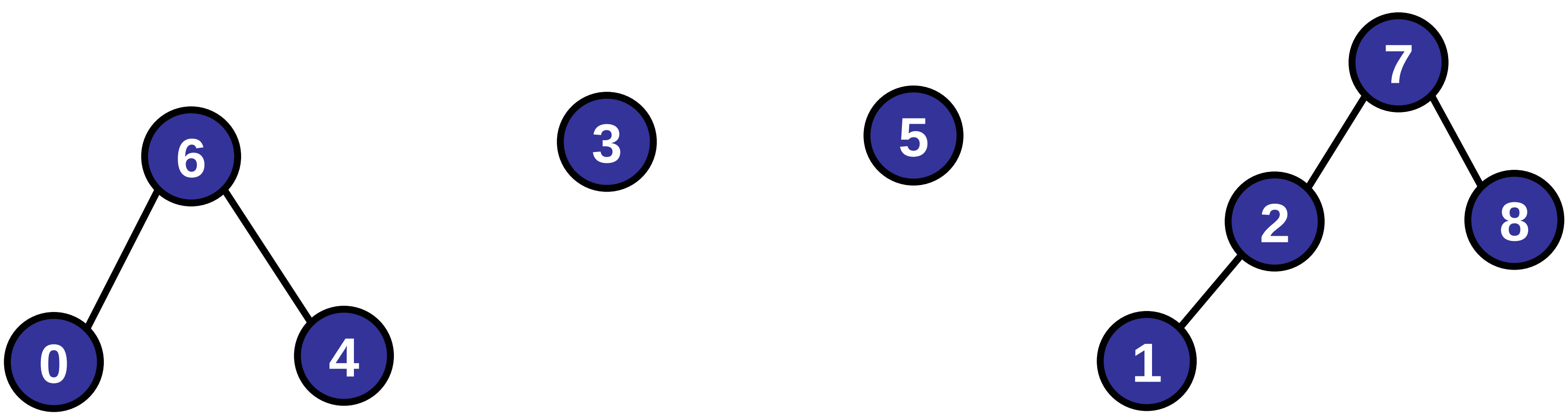
object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



Quick Union

```
find(int p, int q)
   while (parent[p] != p) p = parent[p];
   while (parent[q] != q) q = parent[q];
   return (p == q);
```

object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7

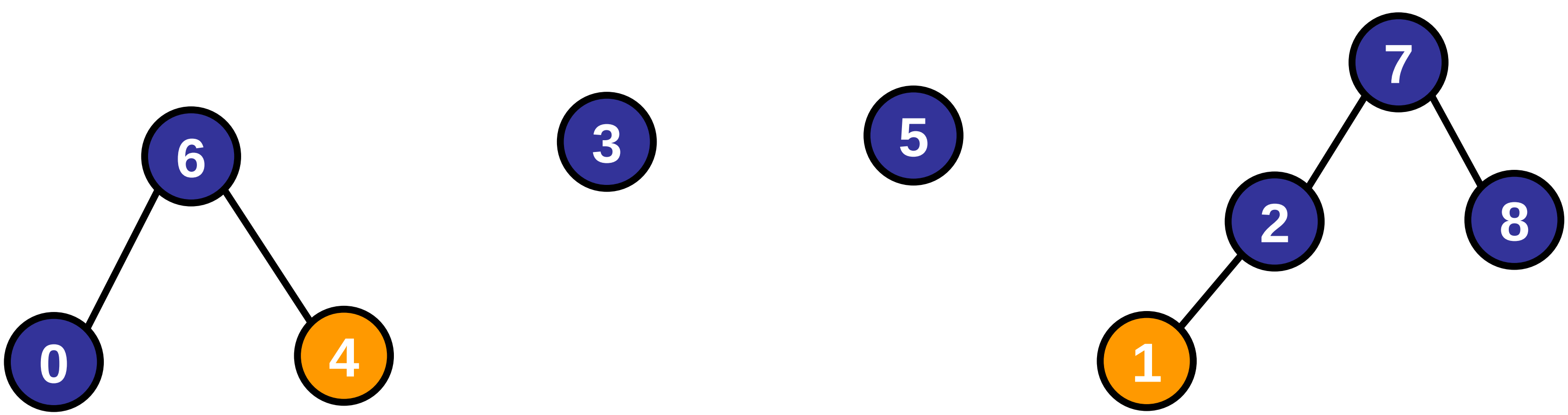


Quick Union

Example: `find(4, 1)`

`4 → 6 → 6;`

object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



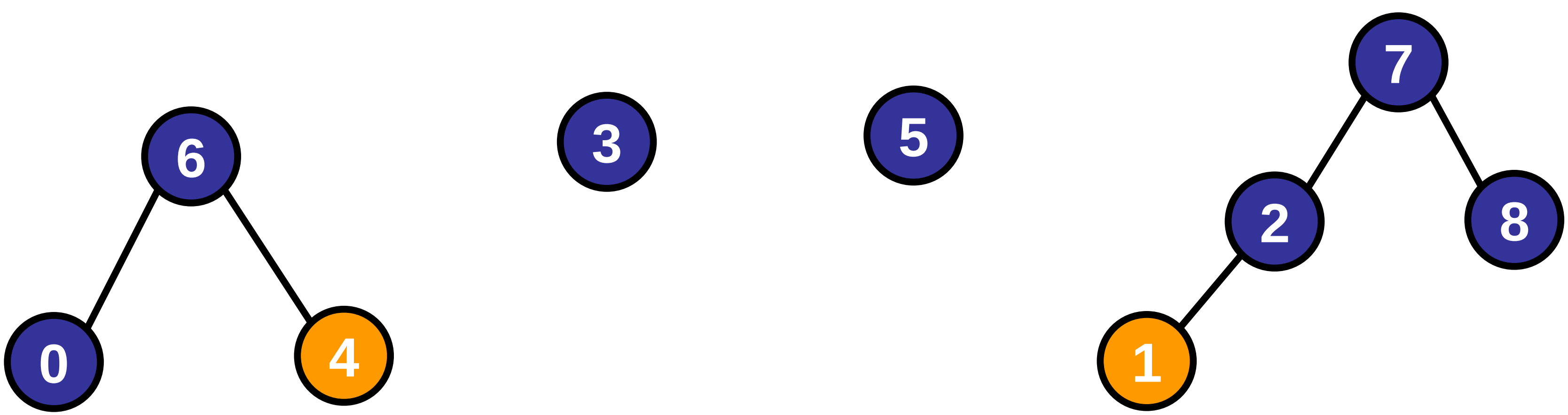
Quick Union

Example: find(4, 1)

4 → 6 → 6

1 → 2 → 7 → 7

object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7

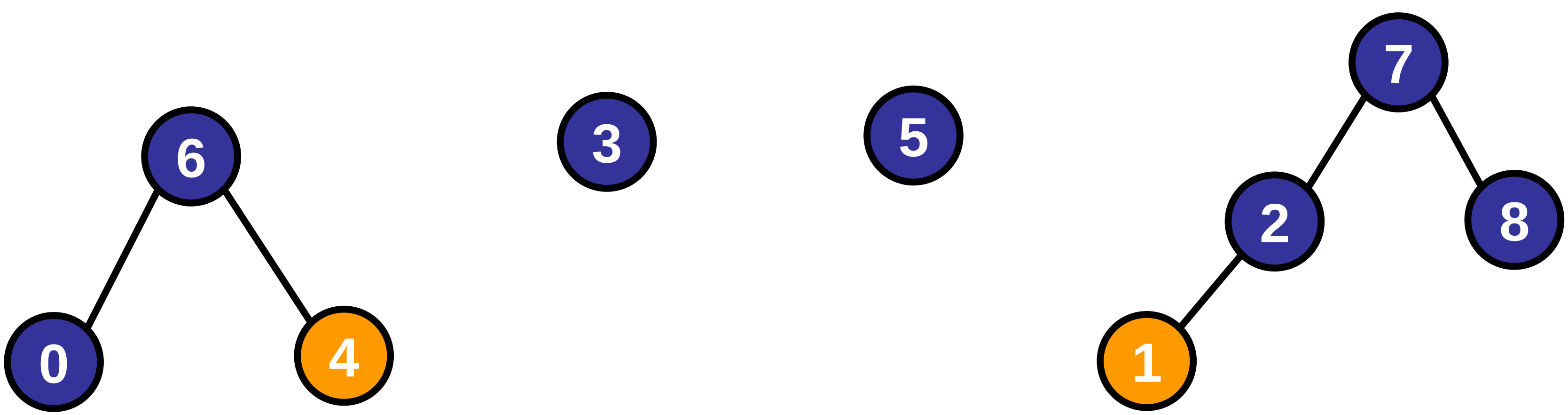


Quick Union

Example: `find(4, 1)`

```
4 → 6 → 6
1 → 2 → 7 → 7
return (6 == 7) → false
```

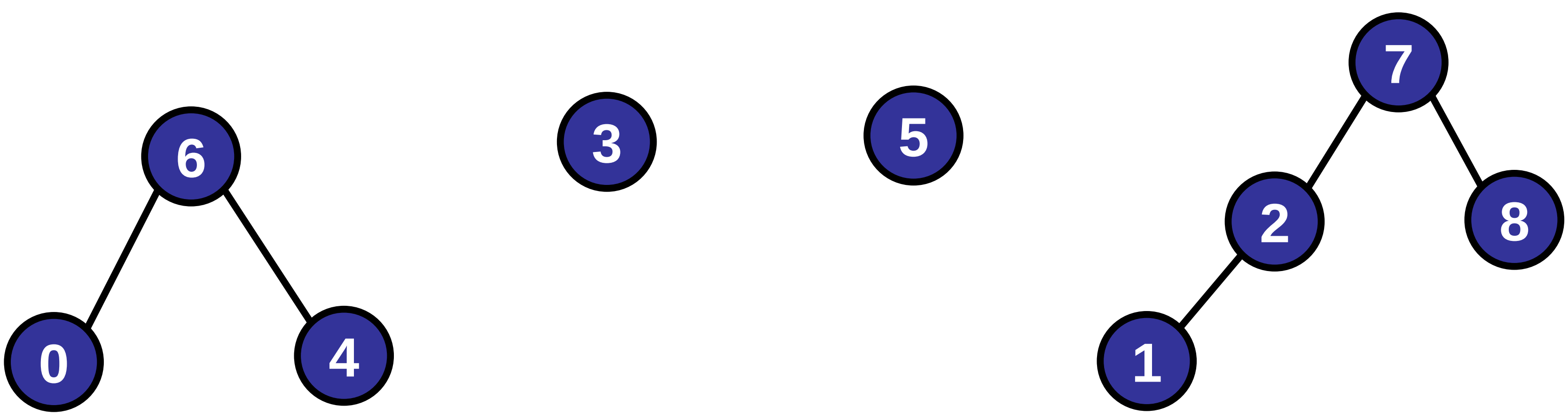
object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
parent	**6**	**2**	**7**	**3**	**6**	**1**	**6**	**7**	**7**



Quick Union

```
find(int p, int q)
   while (parent[p] != p) p = parent[p];
   while (parent[q] != q) q =parent[q];
   return (p == q);
```

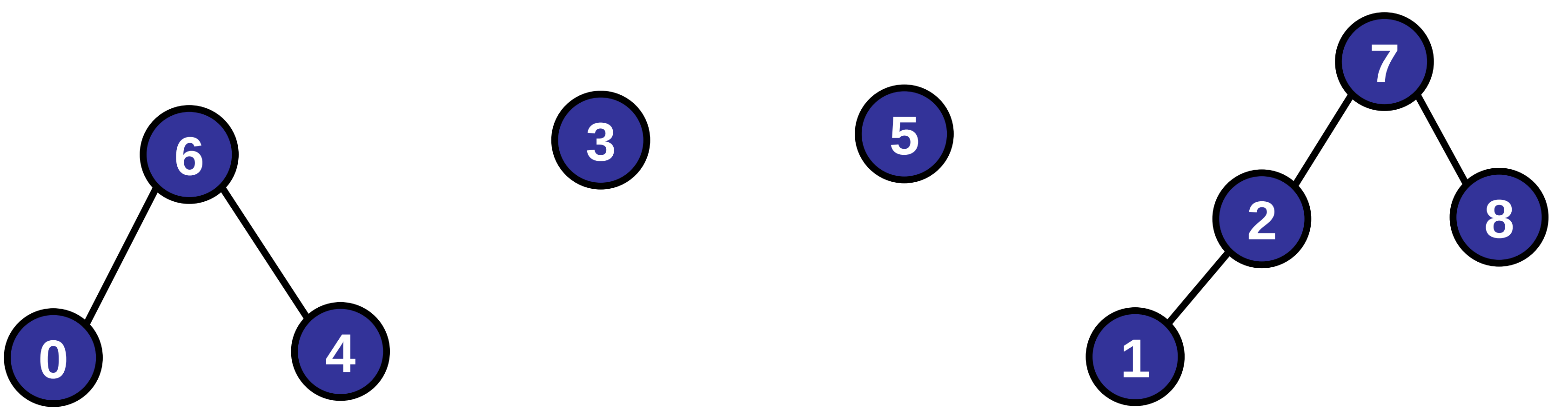
object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



Quick Union

```
union(int p, int q)
   while (parent[p] != p) p = parent[p];
   while (parent[q] != q) q= parent[q];
   parent[p] = q;
```

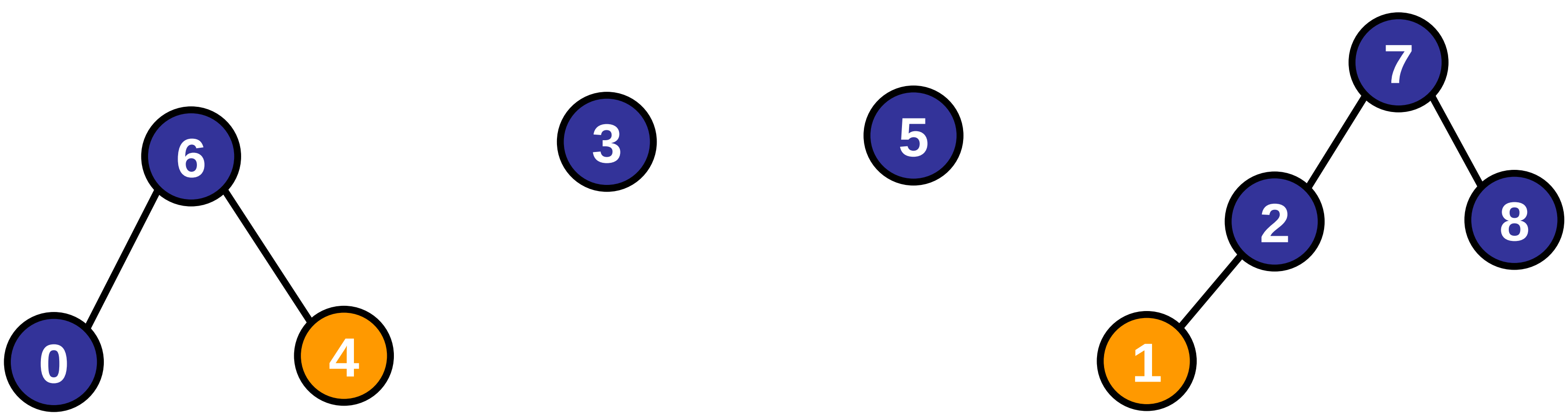
object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
parent	**6**	**2**	**7**	**3**	**6**	**1**	**6**	**7**	**7**



Quick Union

`Example: union(1, 4)`

object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



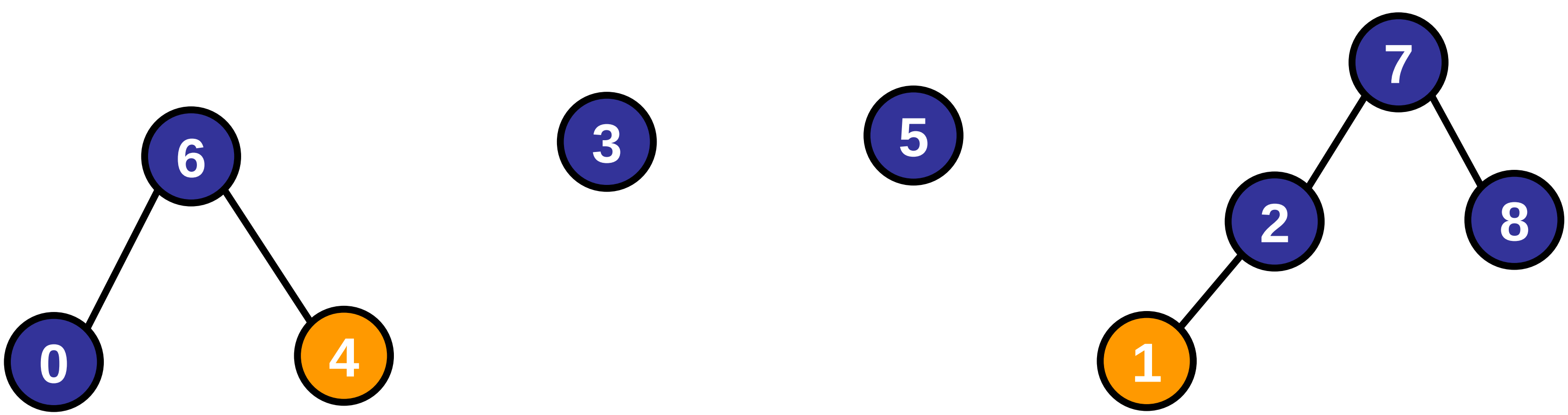
Quick Union

Example: **union**(1, 4)

4 → 6 → **6**

1 → 2 → 7 → **7**

object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



Quick Union

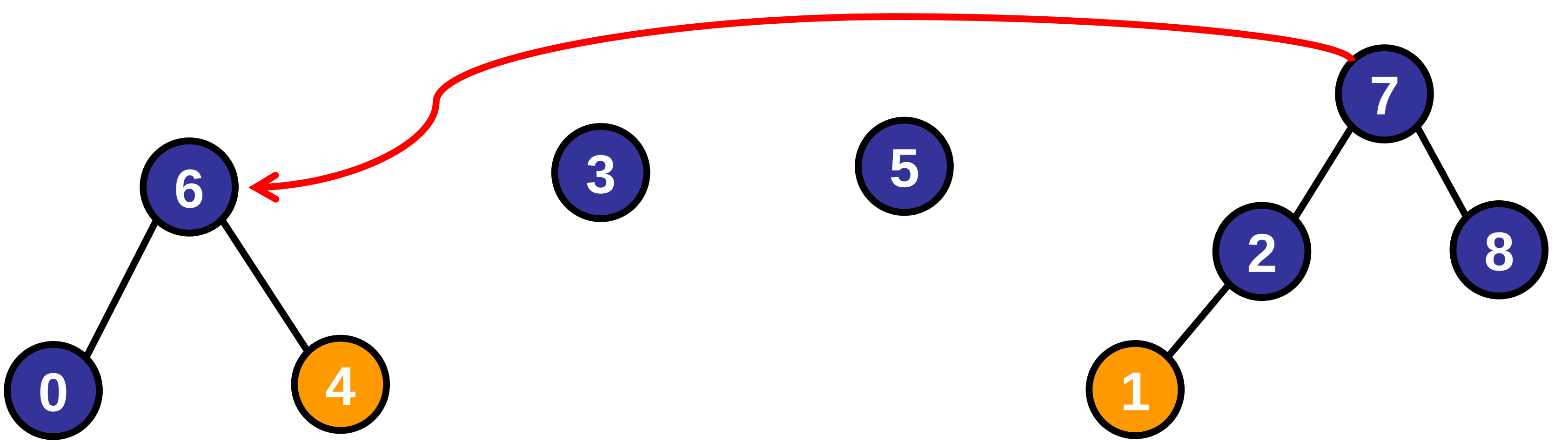
Example: `union(1, 4)`

4 → 6 → 6

1 → 2 → 7 → 7

`parent[7] = 6;`

object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
parent	**6**	**2**	**7**	**3**	**6**	**1**	**6**	**6**	**7**

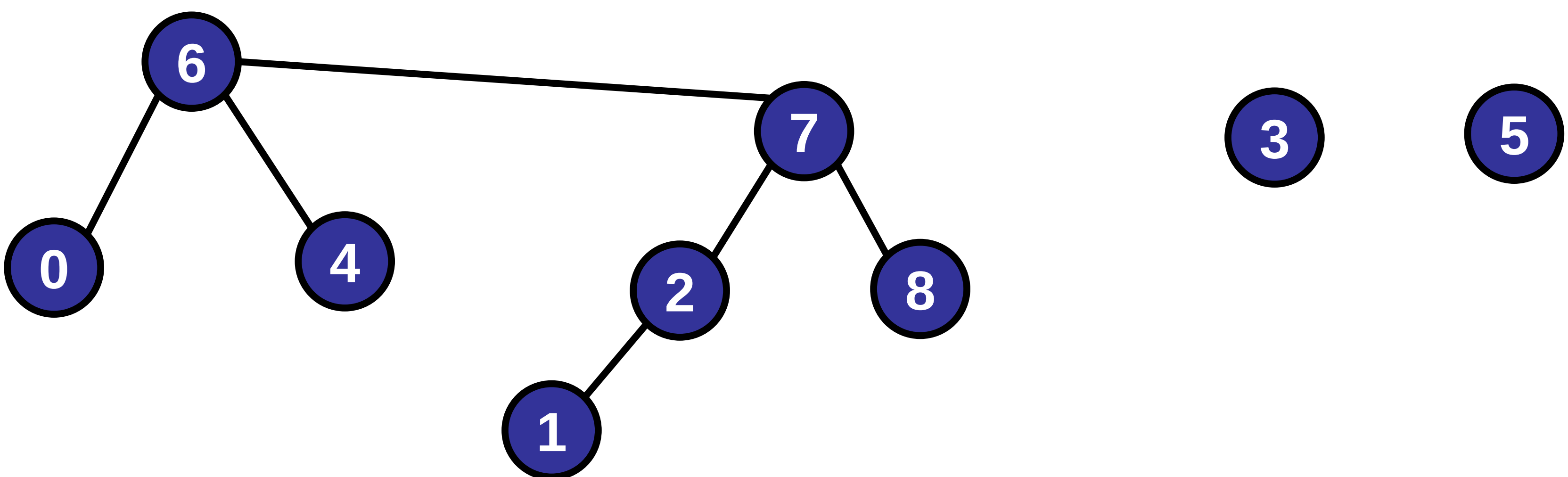


Quick Union

Example: union(1, 4)

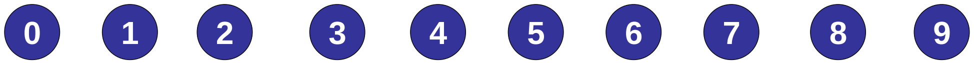
```
4 → 6 → 6
1 → 2 → 7 → 7
parent[7] = 6;
```

object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	6	7



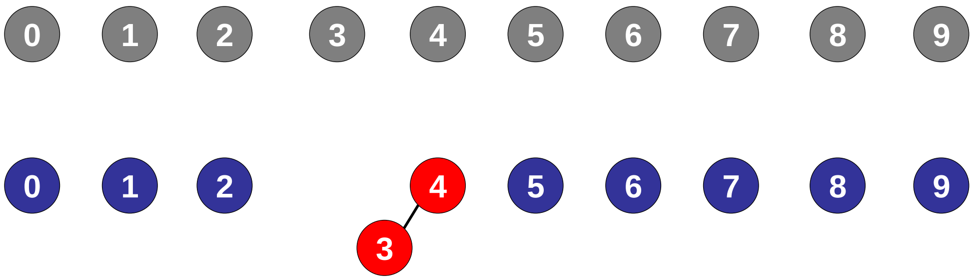
Example:

P	0	1	2	3	4	5	6	7	8	9



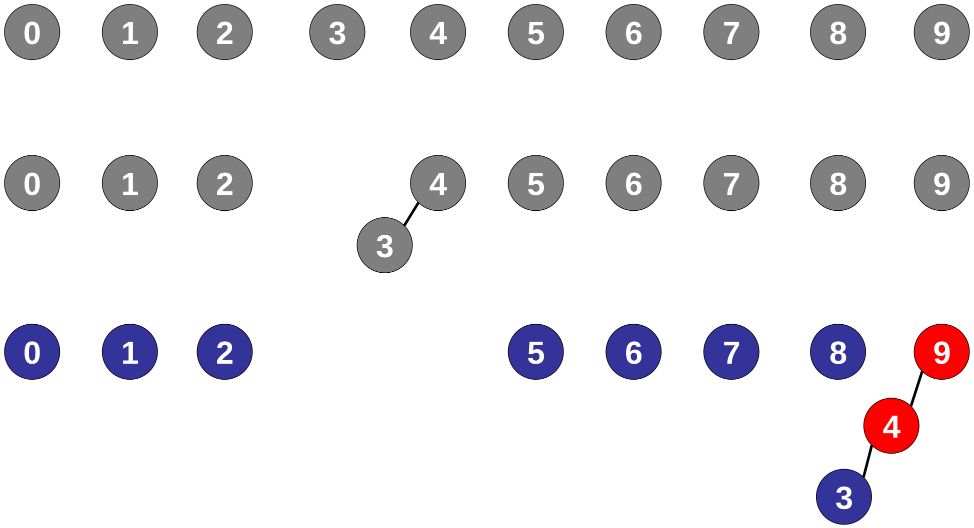
Example:

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9



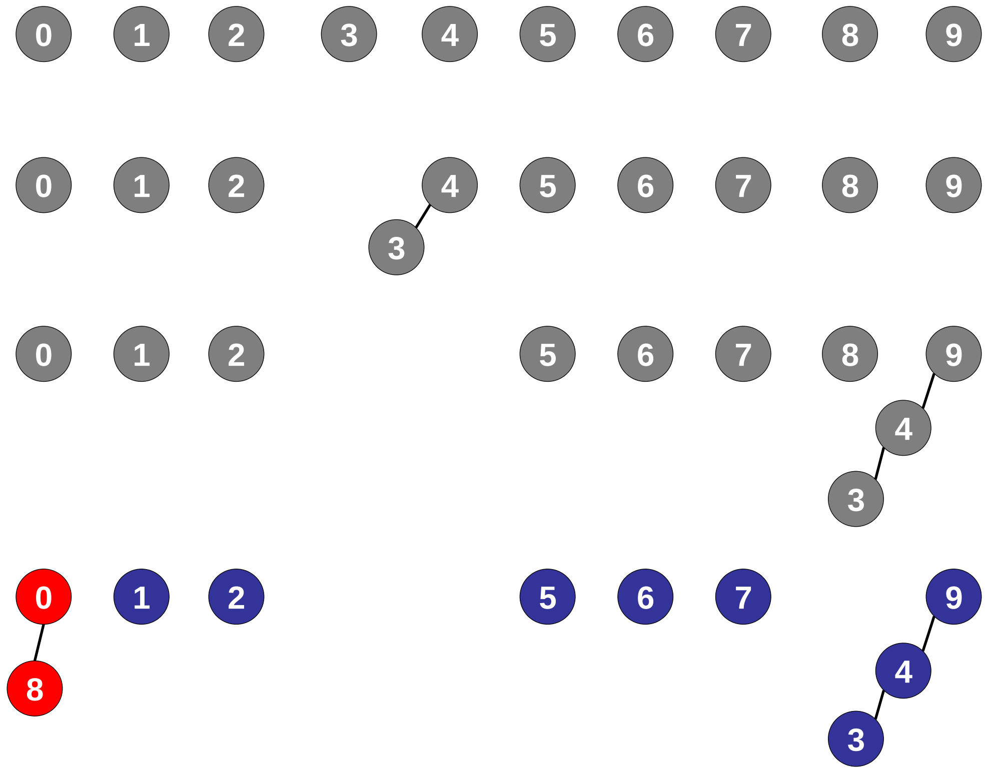
Example:

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9



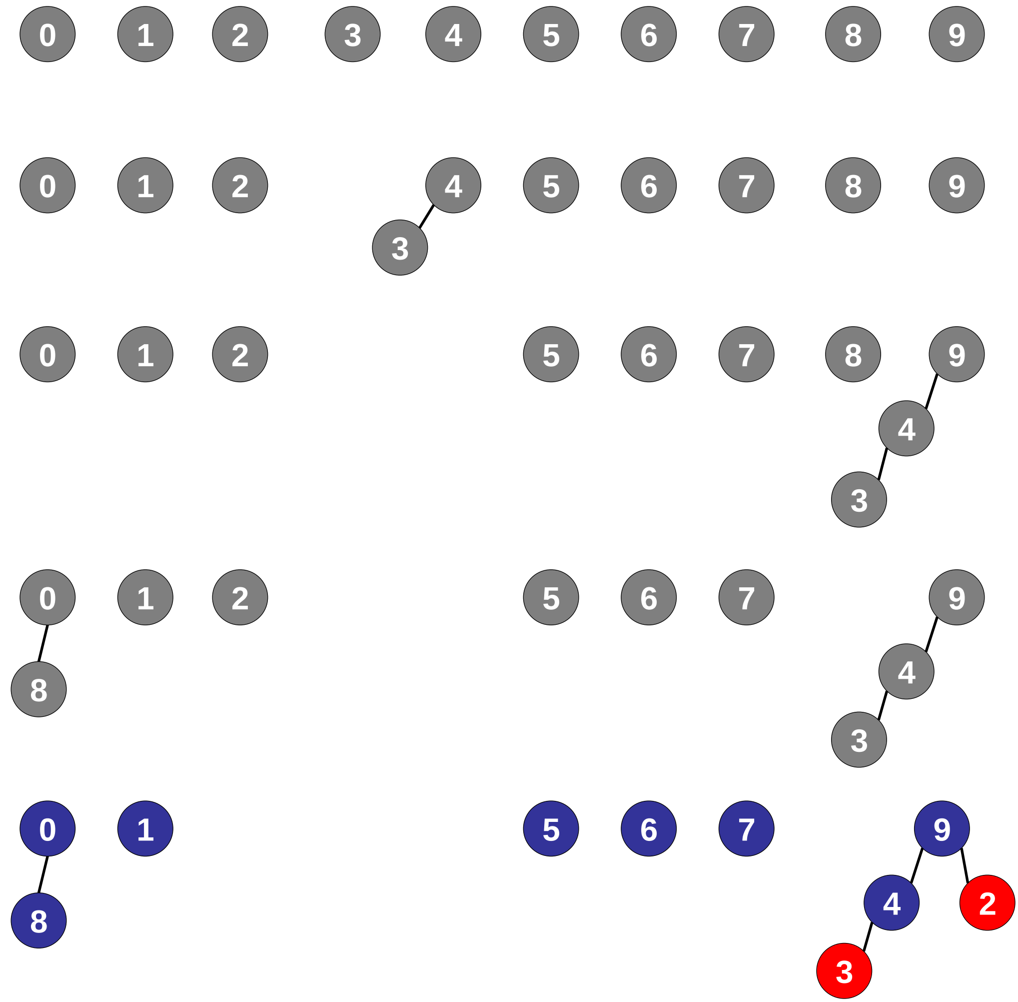
Example:

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9



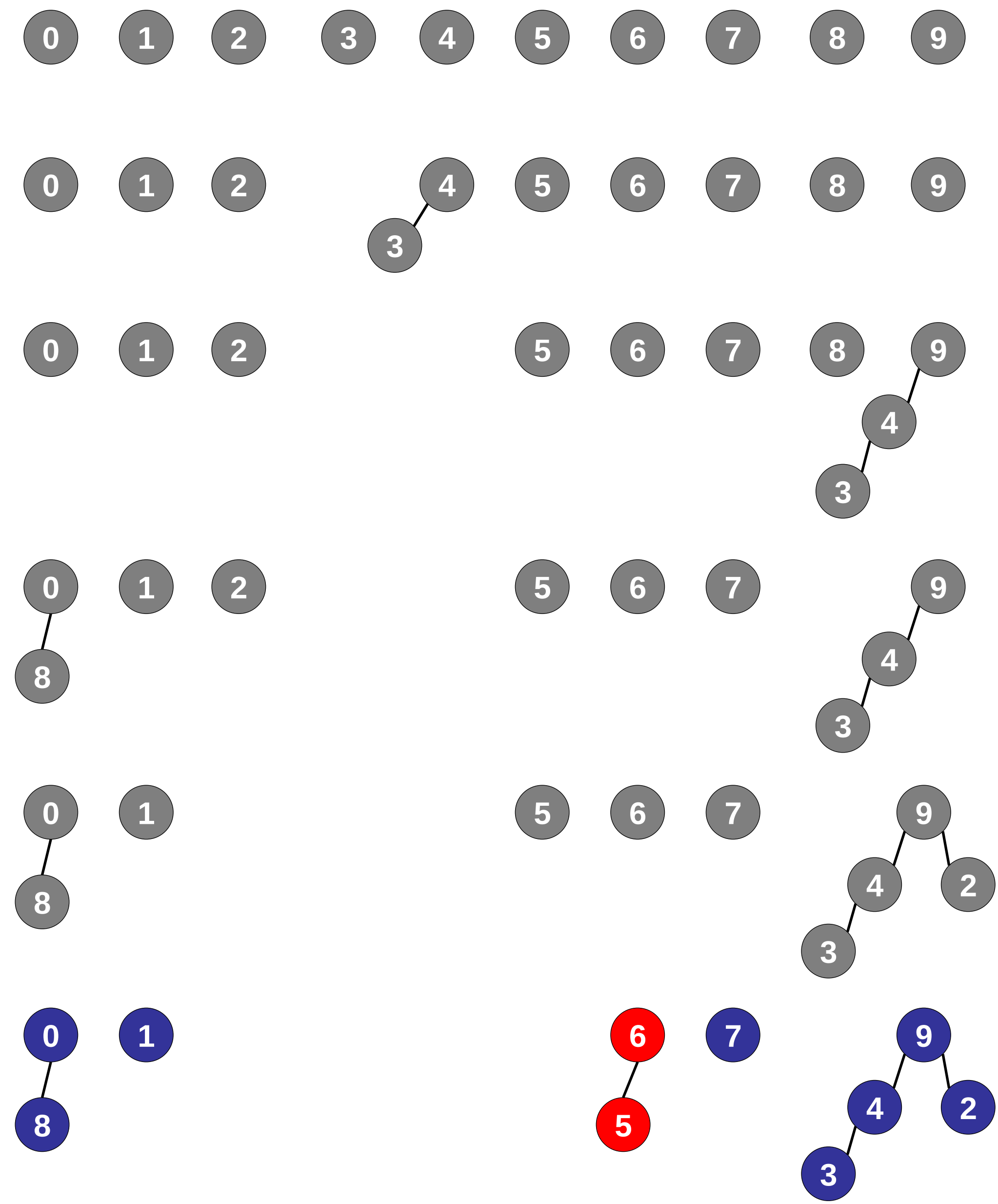
Example:

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9



Example:

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9

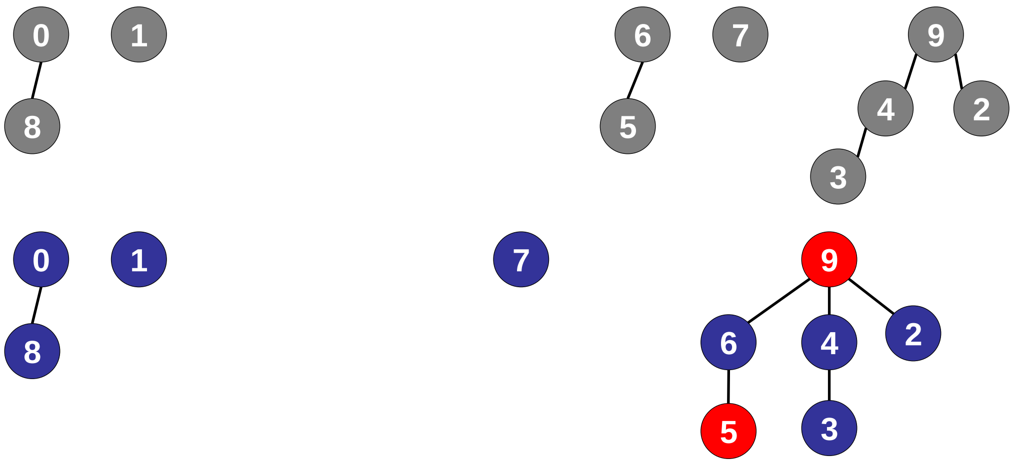


Example:

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9

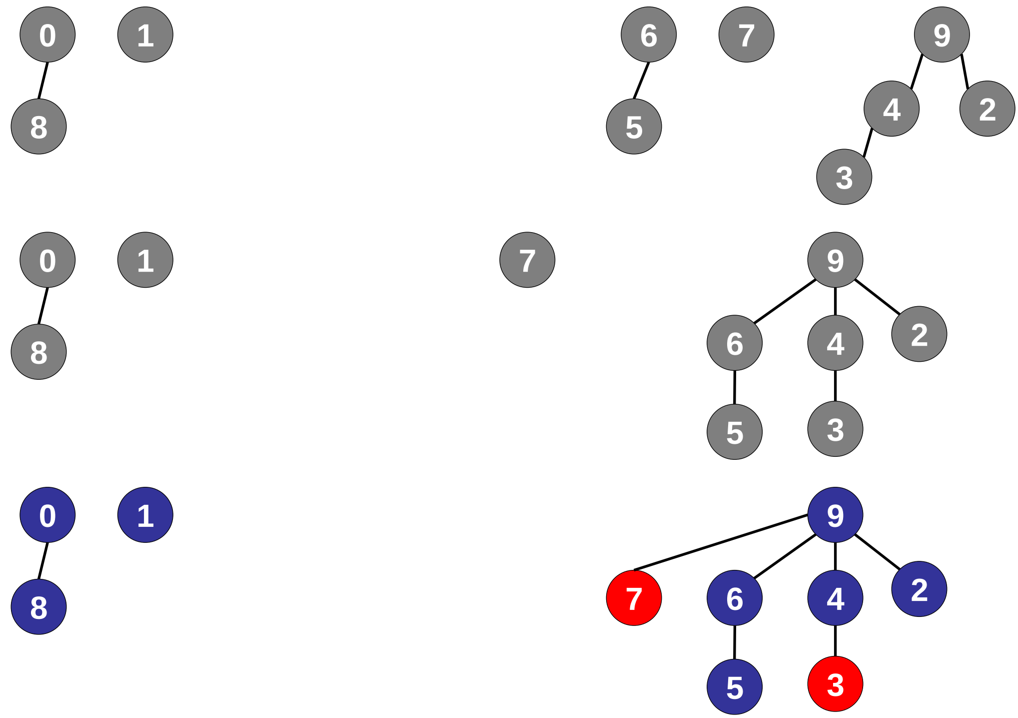
Example:

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9
5-9	0	1	9	4	9	6	9	7	0	9



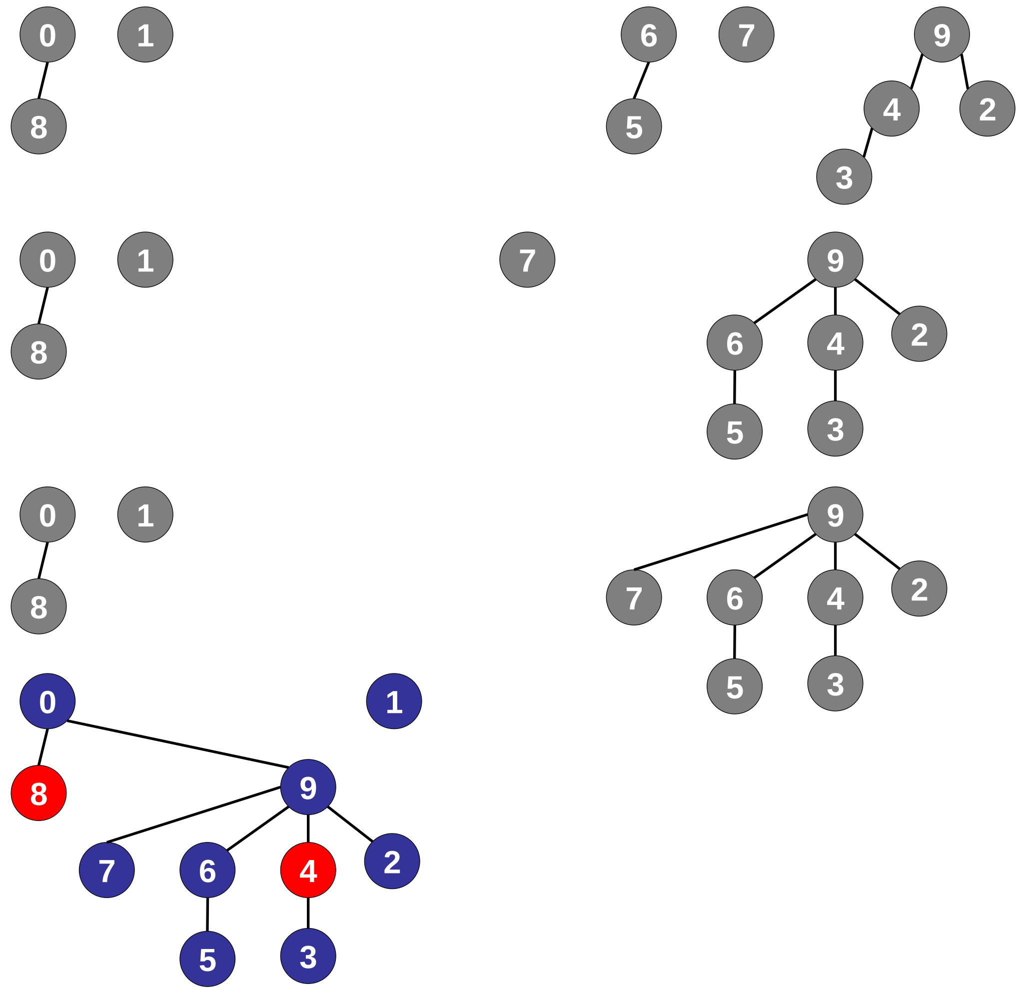
Example:

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9
5-9	0	1	9	4	9	6	9	7	0	9
7-3	0	1	9	4	9	6	9	9	0	9



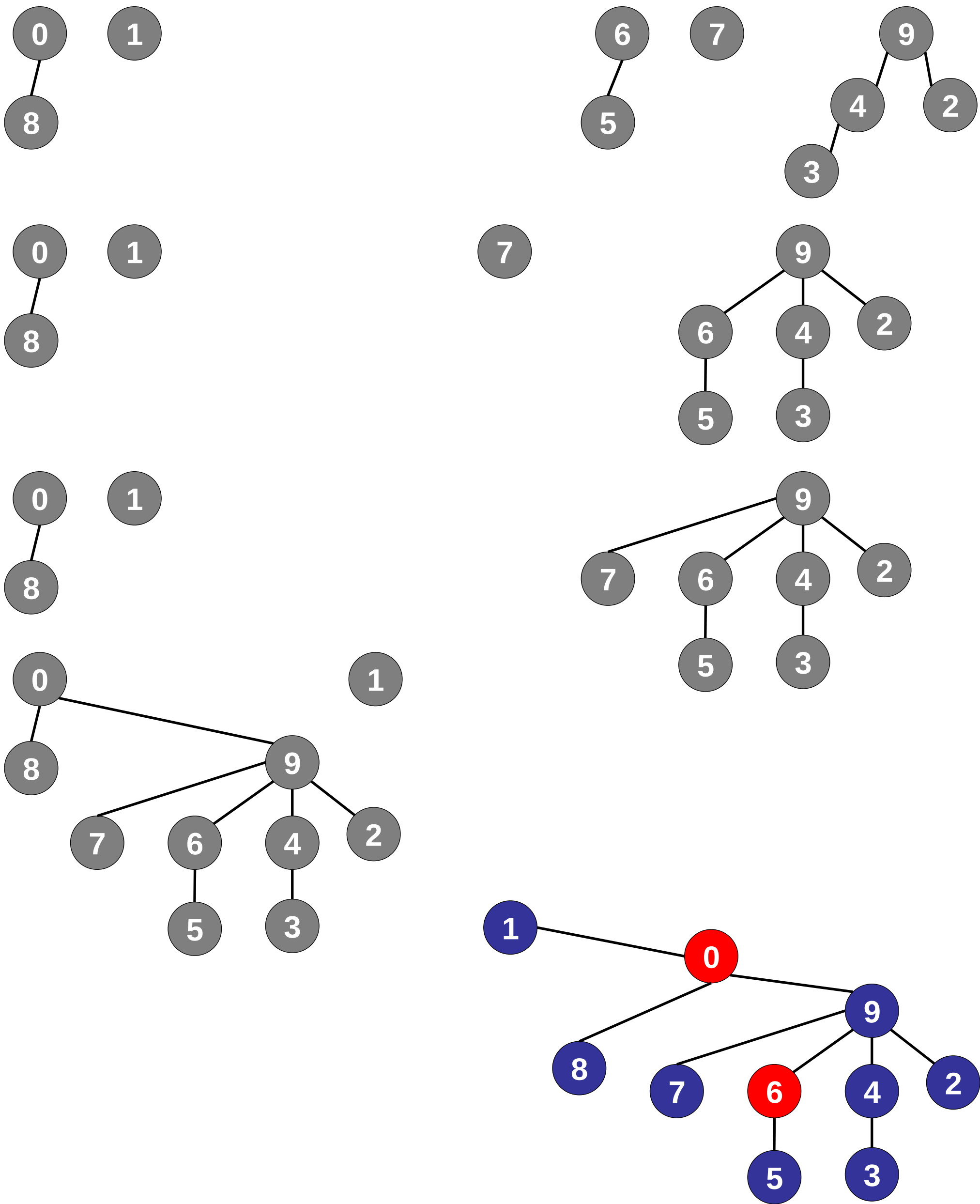
Example:

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9
5-9	0	1	9	4	9	6	9	7	0	9
7-3	0	1	9	4	9	6	9	9	0	9
4-8	0	1	9	4	9	6	9	9	0	0



Example:

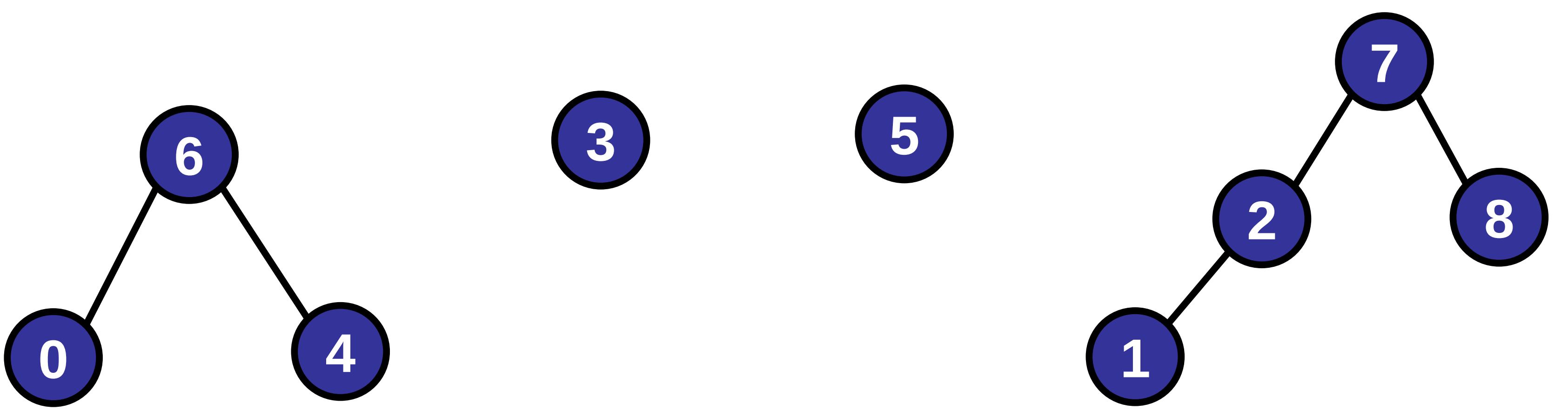
P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9
5-9	0	1	9	4	9	6	9	7	0	9
7-3	0	1	9	4	9	6	9	9	0	9
4-8	0	1	9	4	9	6	9	9	0	0
6-1	1	1	9	4	9	6	9	9	0	0



Quick Union

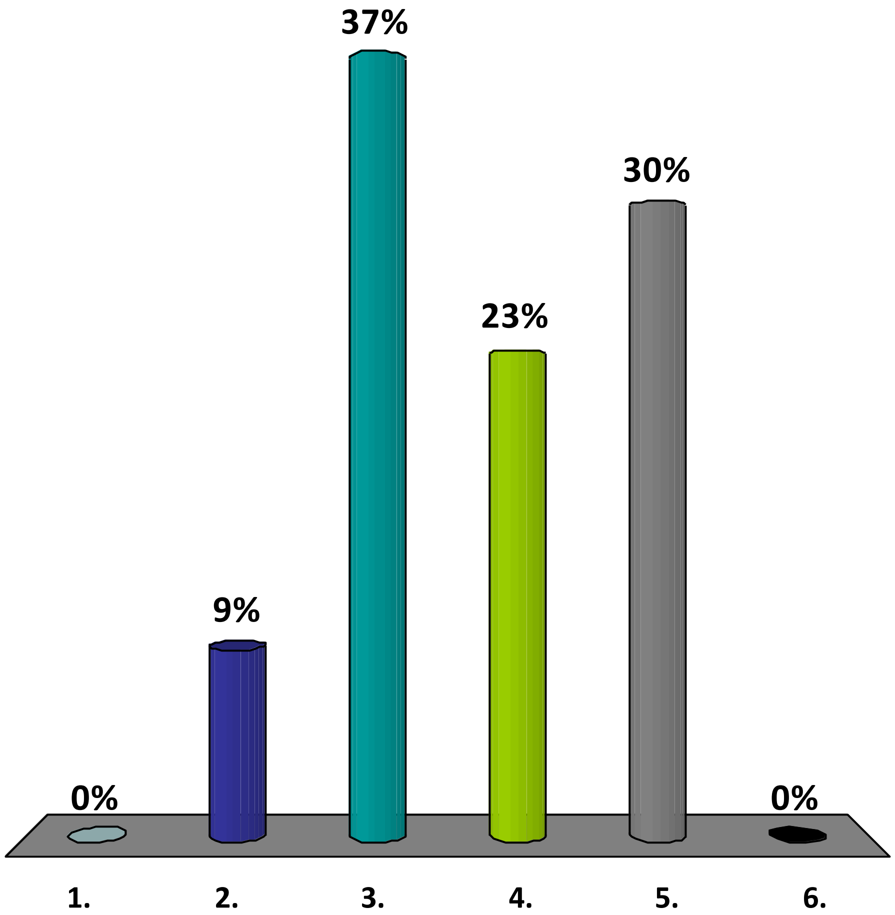
```
union(int p, int q)
   while (parent[p] != p) p = parent[p];
   while (parent[q] != q) q = parent[q];
   parent[p] = q;
```

object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
parent	**6**	**2**	**7**	**3**	**6**	**1**	**6**	**7**	**7**



Running time of (Find, Union):

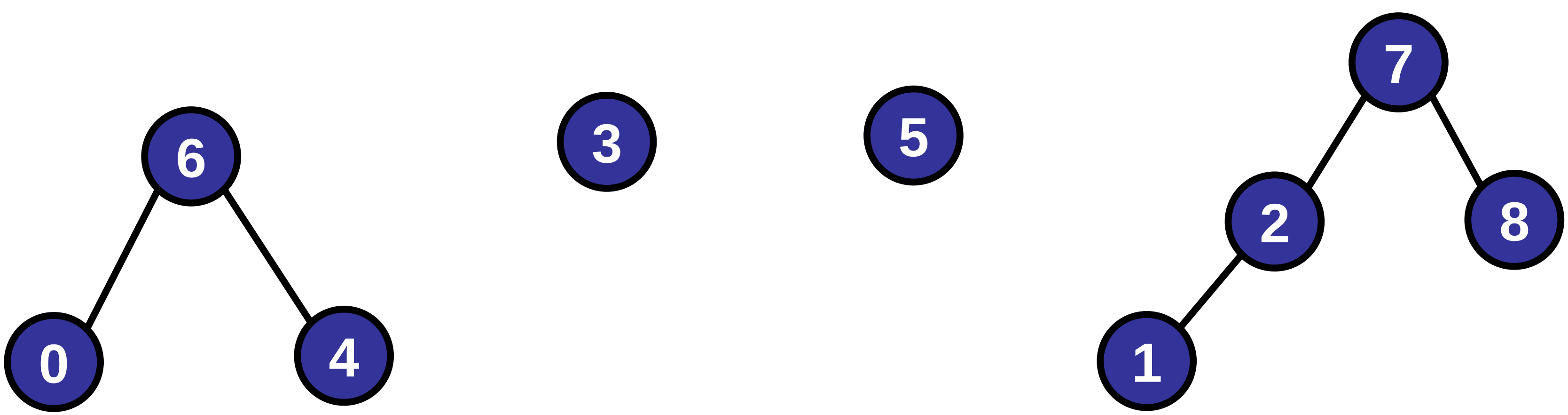
1. O(1), O(1)
2. O(1), O(n)
3. O(n), O(1)
4. O(n), O(n)
5. O(log n), O(log n)
6. None of the above.



Quick Union

```
find(int p, int q)
   while (parent[p] != p) p = parent[p];
   while (parent[q] != q) q = parent[q];
   return (p == q);
```

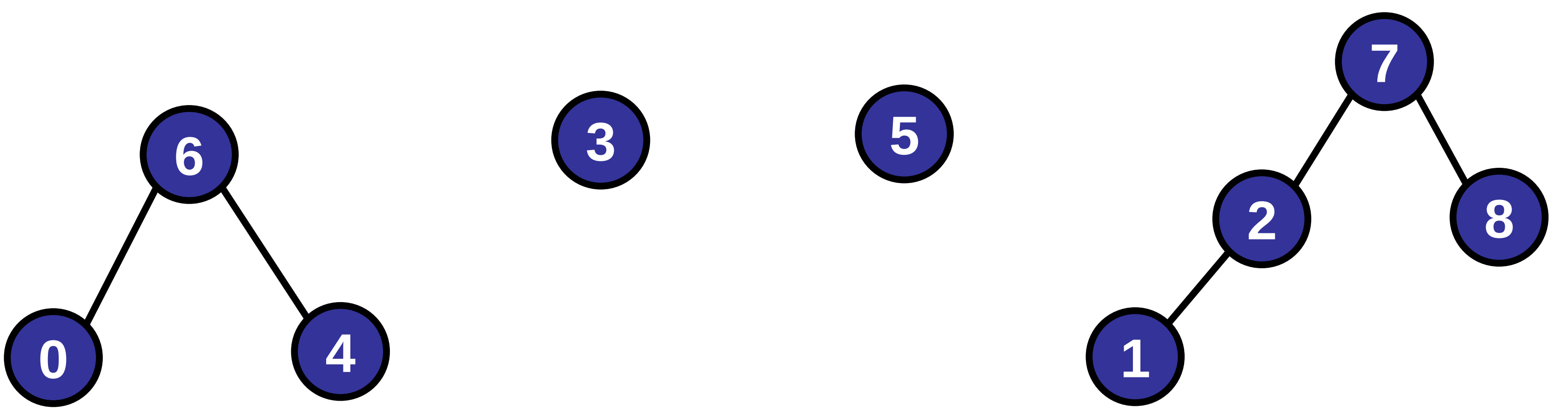
object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



Quick Union

```
union(int p, int q)
  while (parent[p] != p) p = parent[p];
  while (parent[q] != q) q = parent[q];
  parent[p] = q;
```

object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
parent	**6**	**2**	**7**	**3**	**6**	**1**	**6**	**7**	**7**



Union-Find Summary

Quick-find is slow:

- Find is fast
- Union is expensive
- Tree is flat

Quick-union is slow:

- Trees too tall (i.e., unbalanced)
- Union ***and*** find are expensive.

	find	union
quick-find	O(1)	O(n)
quick-union	O(n)	O(n)

Roadmap

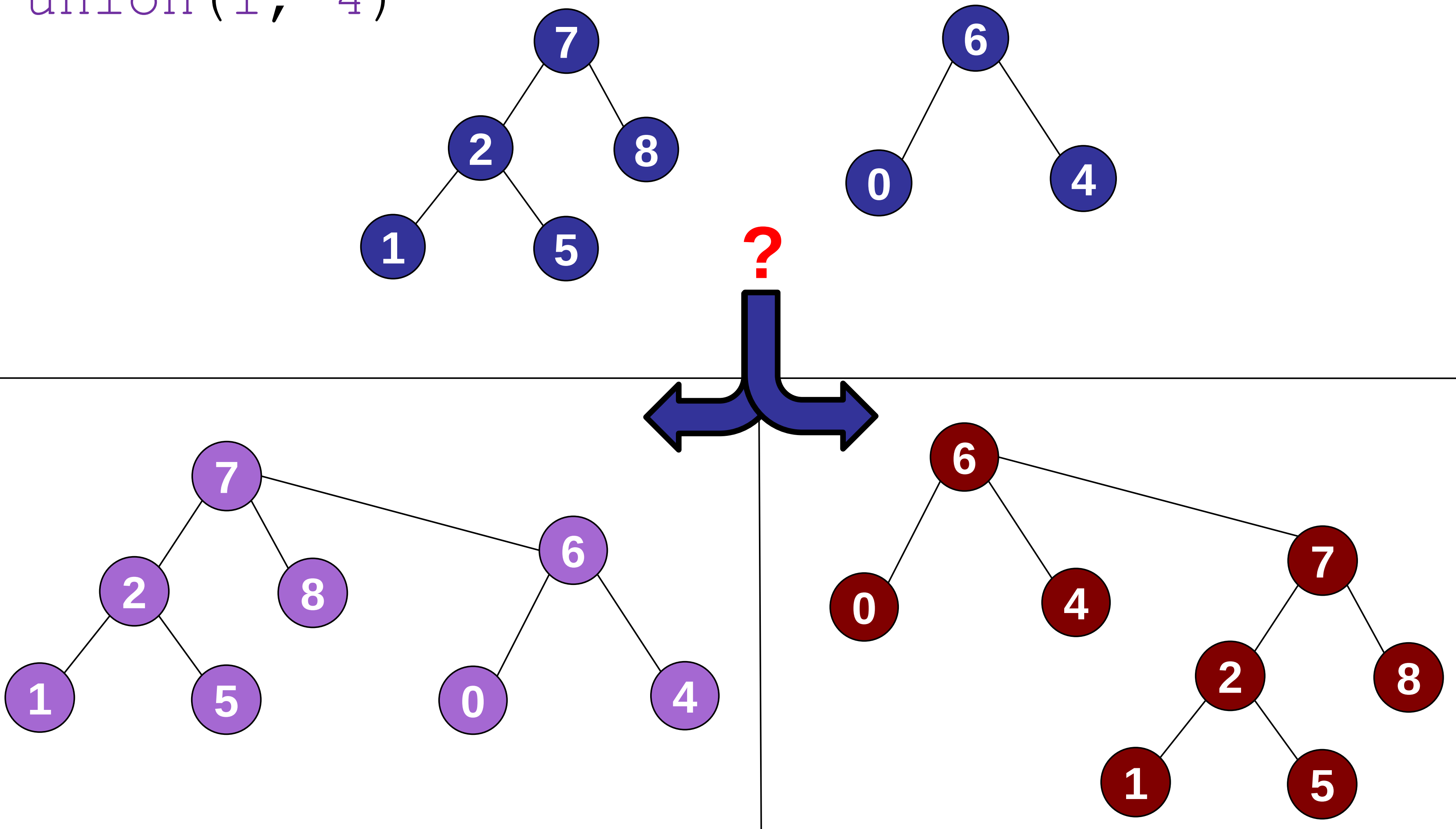
Part II: Disjoint Set

- Problem: Dynamic Connectivity
- Algorithm: Quick-Find
- Algorithm: Quick-Union
- Optimizations
- Applications

Weighted Union

Question: which tree should you make the root?

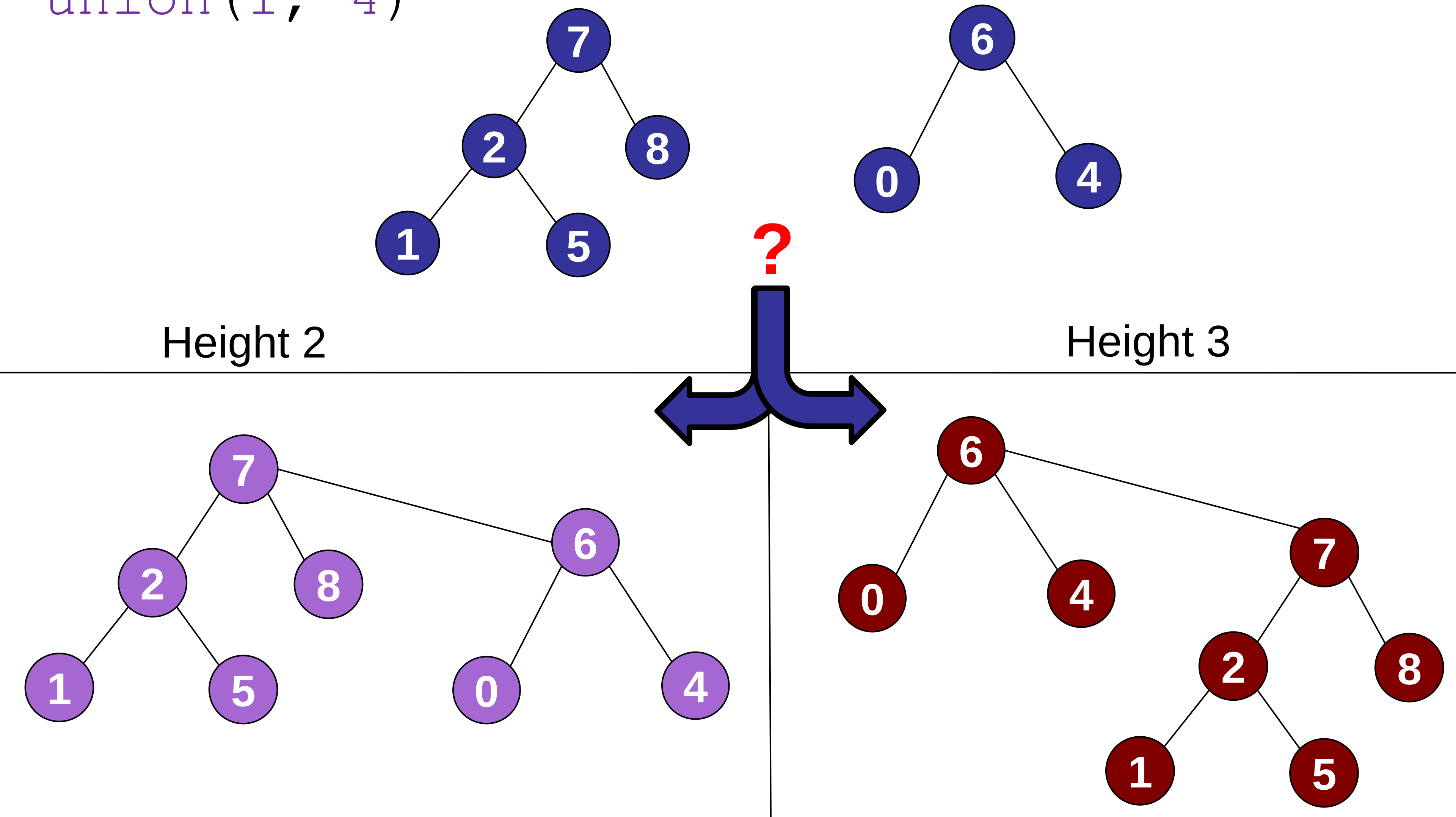
`union(1, 4)`



Weighted Union

Question: which tree should you make the root?

`union(1, 4)`



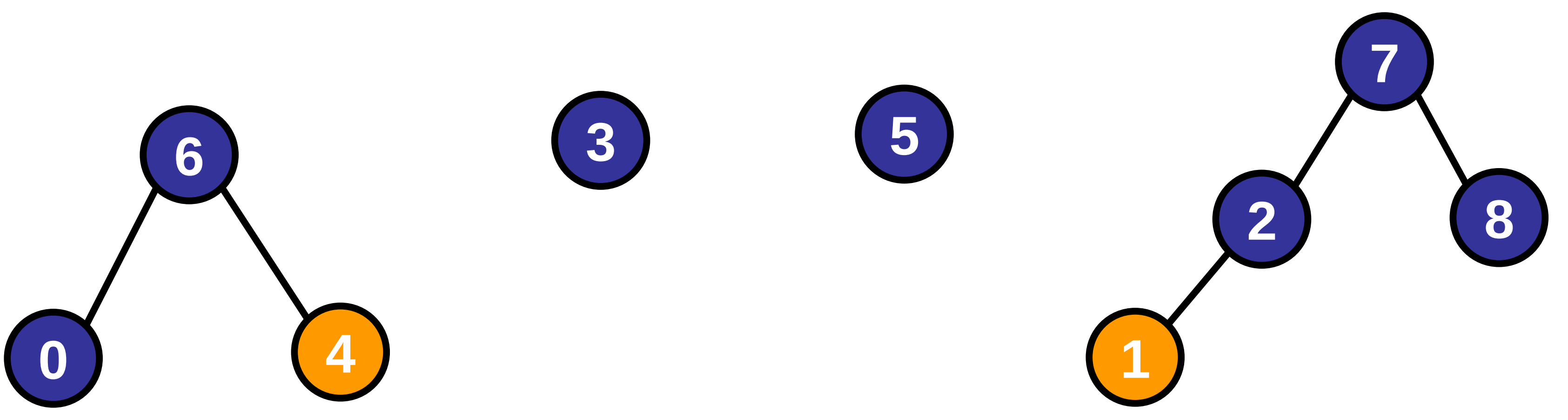
Weighted Union

```
union(int p, int q)
   while (parent[p] !=p) p = parent[p];
   while (parent[q] !=q) q = parent[q];
   if (size[p] > size[q] {
           parent[q] = p;   // Link q to p
           size[p] = size[p] + size[q];
   }
   else {
           parent[p] = q; // Link p to q
           size[q] = size[p] + size[q];
   }
```

Weighted Union

`union(1, 4)`

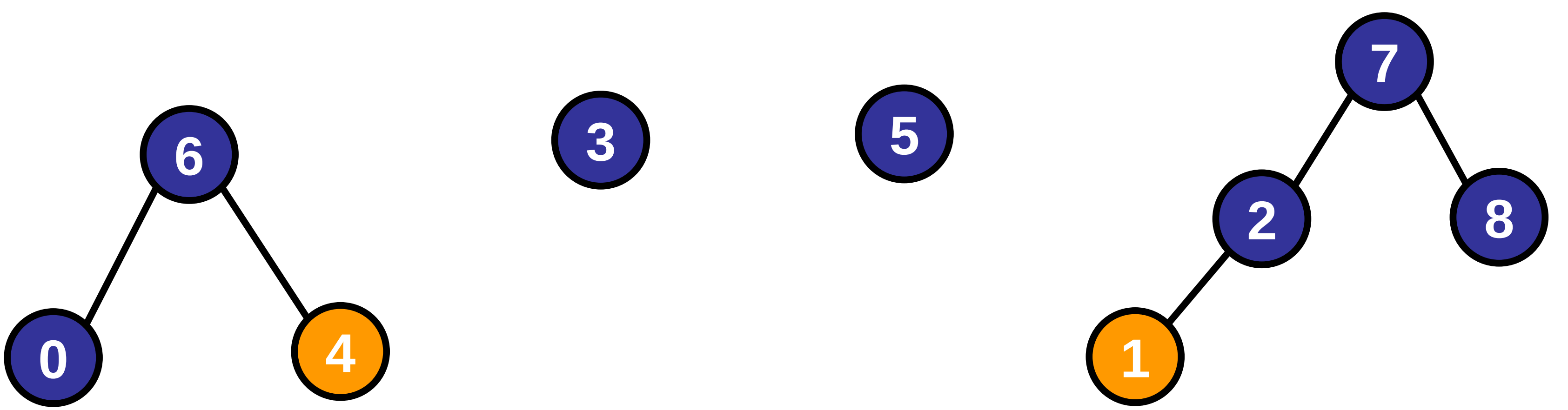
object	0	1	2	3	4	5	6	7	8
size	1	1	2	1	1	1	3	4	1
parent	6	2	7	3	6	1	6	7	7



Weighted Union

`union(1, 4)`

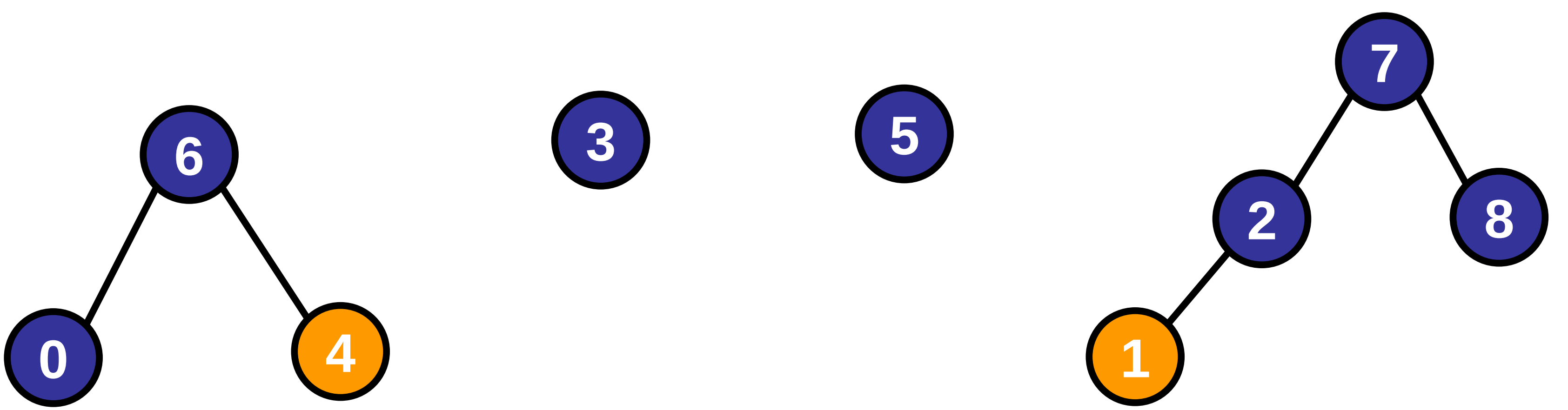
object	0	1	2	3	4	5	6	7	8
size	1	1	2	1	1	1	3	4	1
parent	6	2	7	3	6	1	6	7	7



Weighted Union

```
union(1, 4)
```

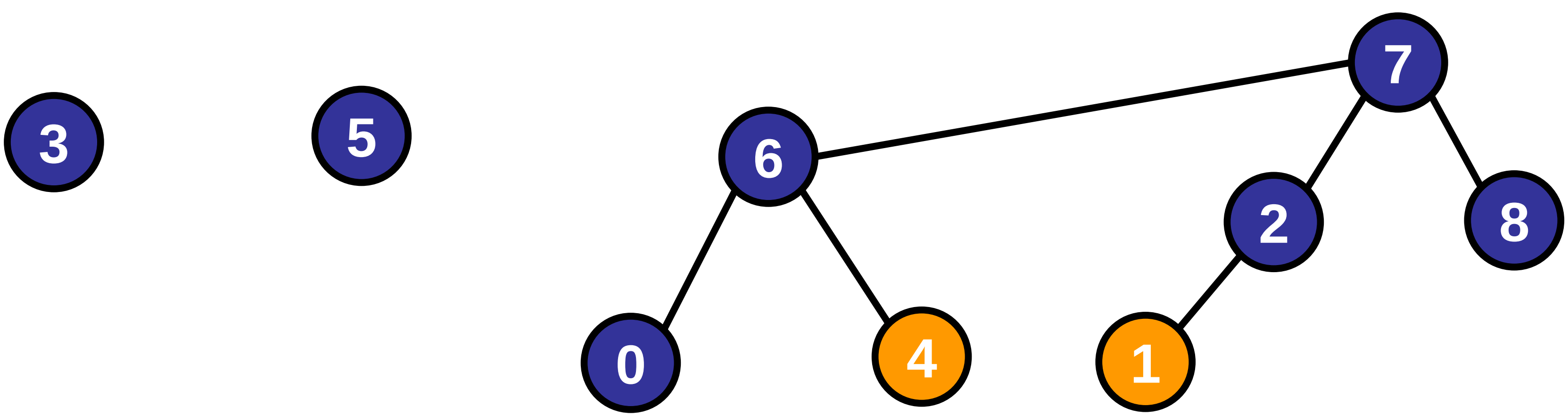
object	0	1	2	3	4	5	6	7	8
size	1	1	2	1	1	1	3	4	1
parent	6	2	7	3	6	1	6	7	7



Weighted Union

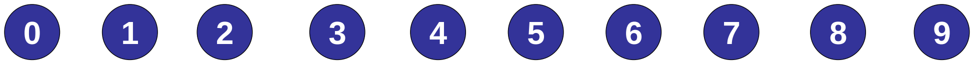
```
union(1, 4)
```

object	0	1	2	3	4	5	6	7	8
size	1	1	2	1	1	1	3	7	1
parent	6	2	7	3	6	1	6	7	7



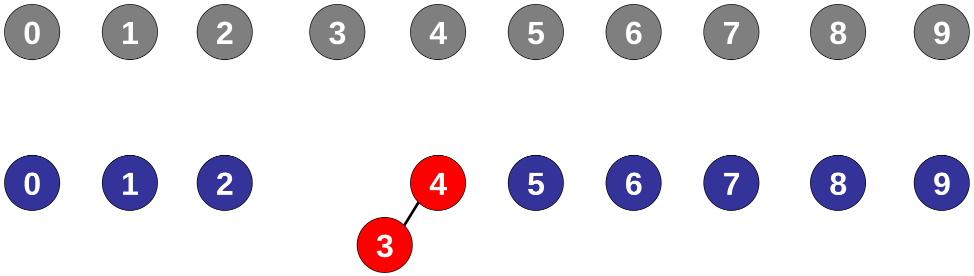
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9



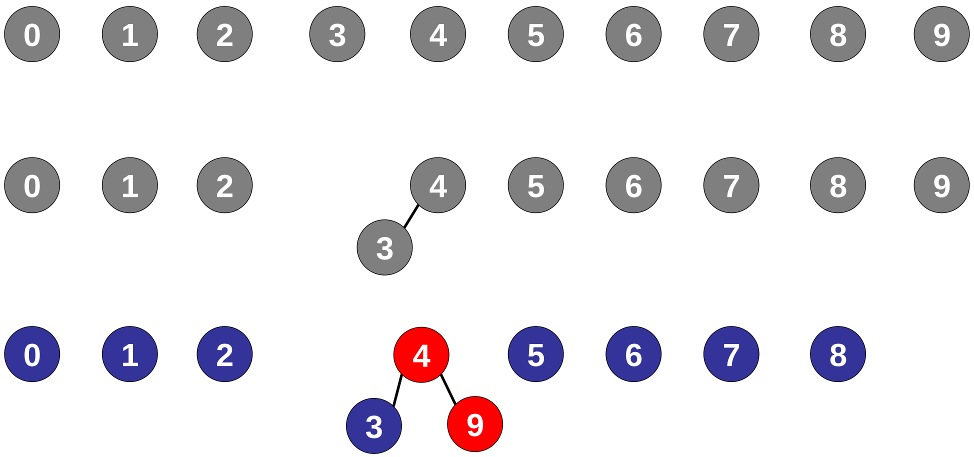
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9



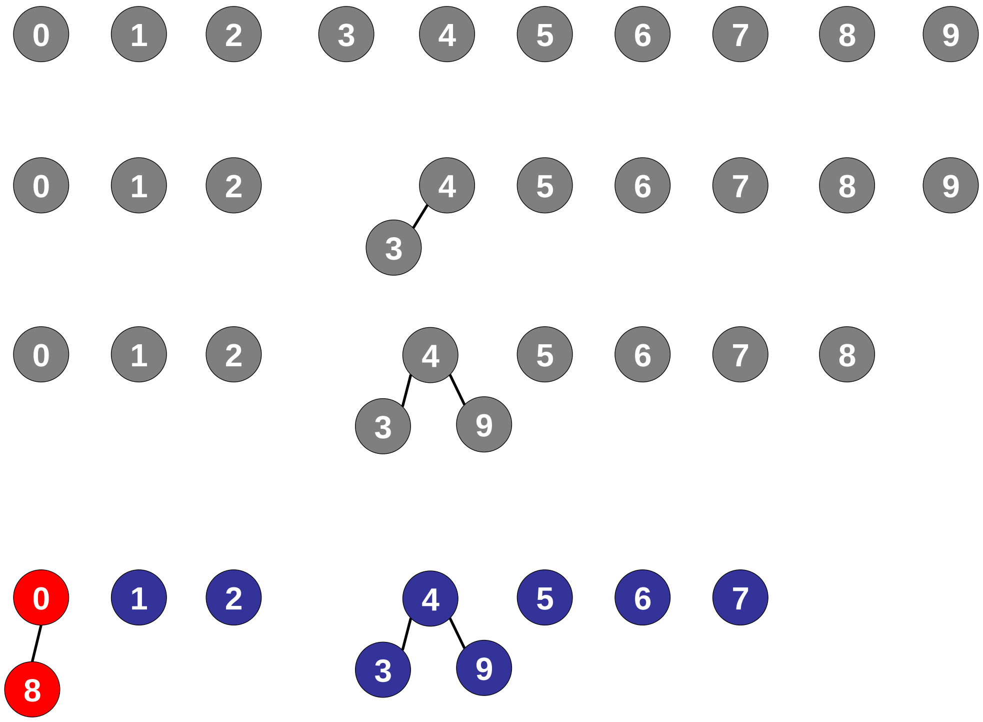
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	4	5	6	7	8	4



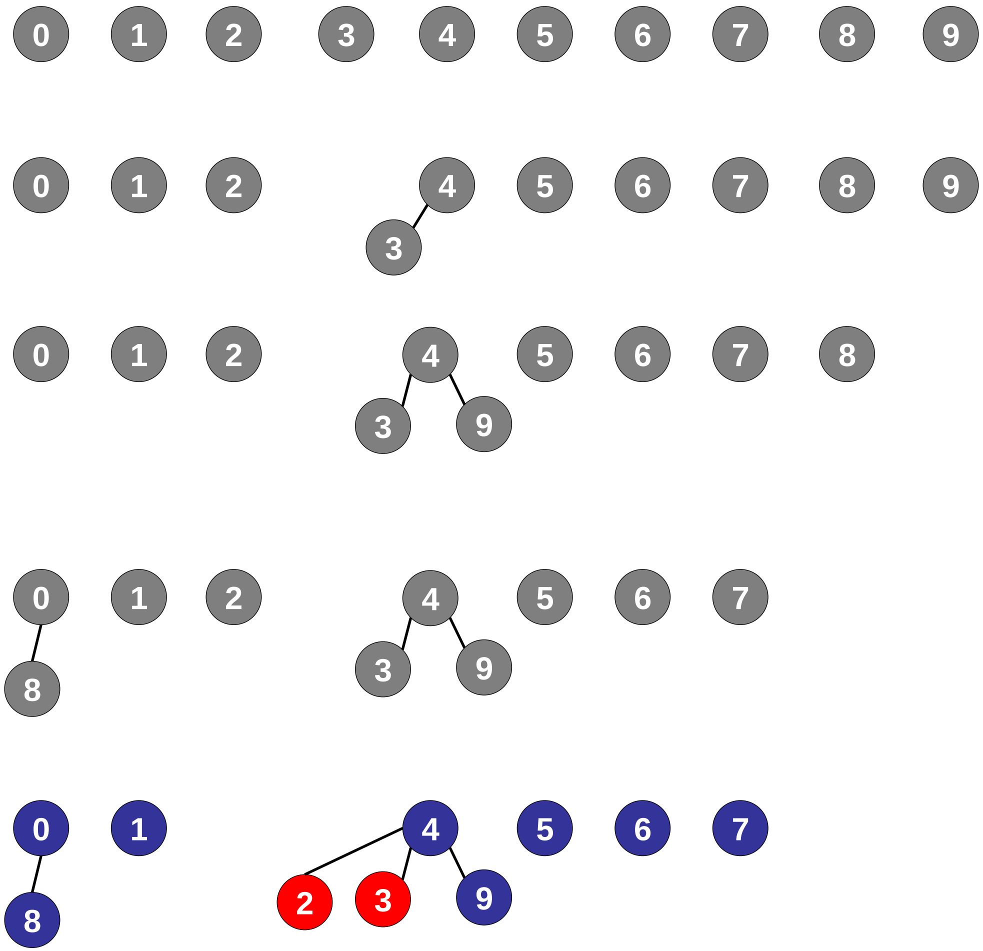
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	4	5	6	7	8	4
8-0	0	1	2	4	4	5	6	7	0	4



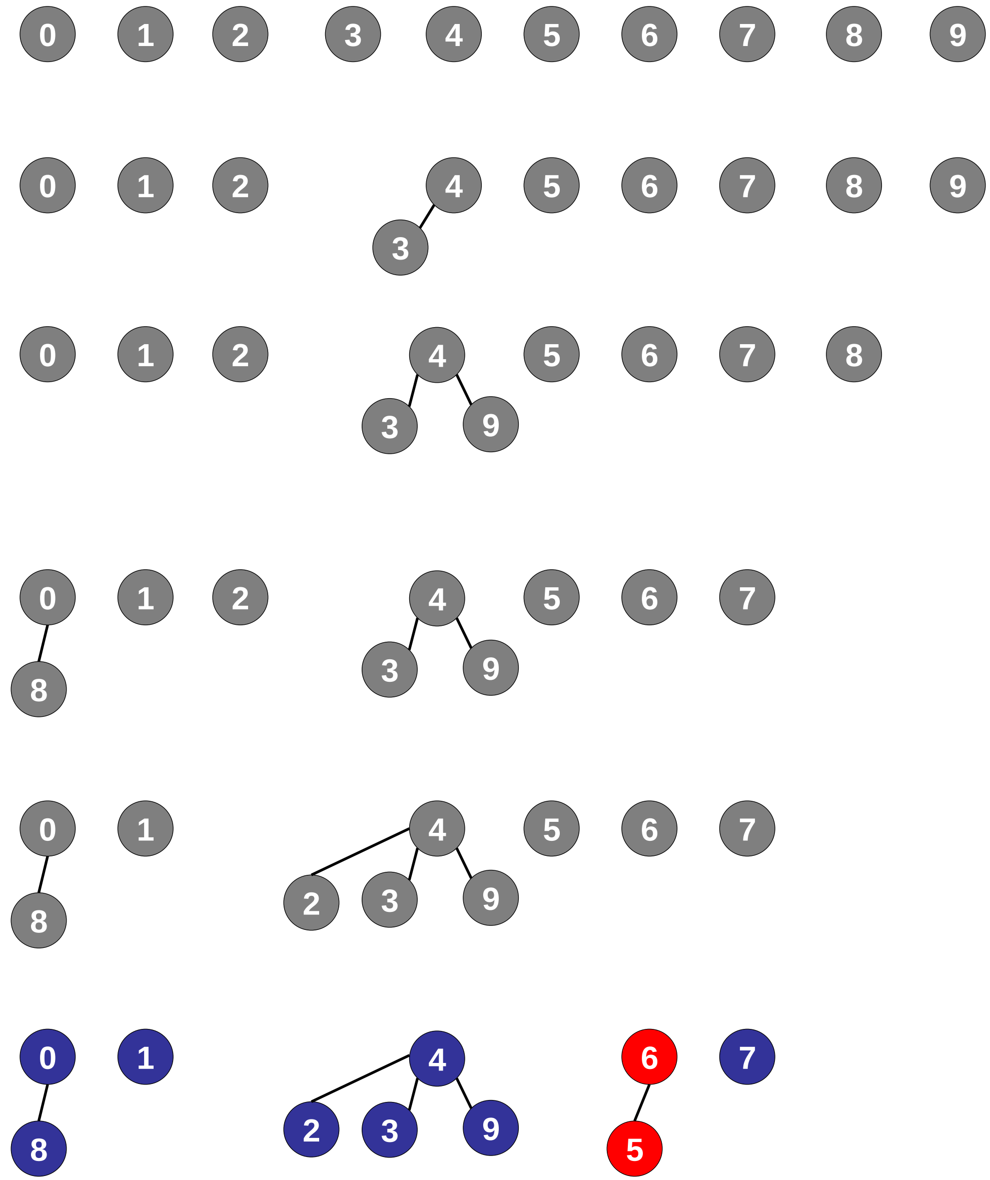
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	4	5	6	7	8	4
8-0	0	1	2	4	4	5	6	7	0	4
2-3	0	1	4	4	4	5	6	7	0	4



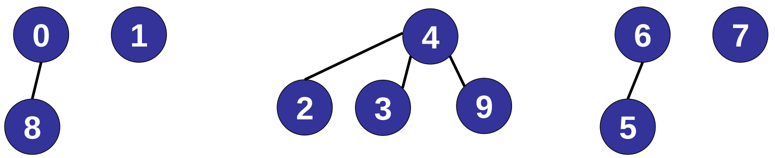
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	4	5	6	7	8	4
8-0	0	1	2	4	4	5	6	7	0	4
2-3	0	1	4	4	4	5	6	7	0	4
5-6	0	1	4	4	4	6	6	7	0	4



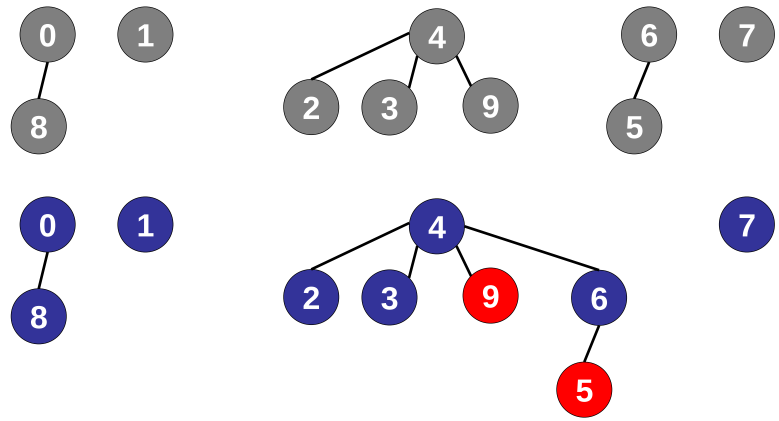
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	4	5	6	7	8	4
8-0	0	1	2	4	4	5	6	7	0	4
2-3	0	1	4	4	4	5	6	7	0	4
5-6	0	1	4	4	4	6	6	7	0	4



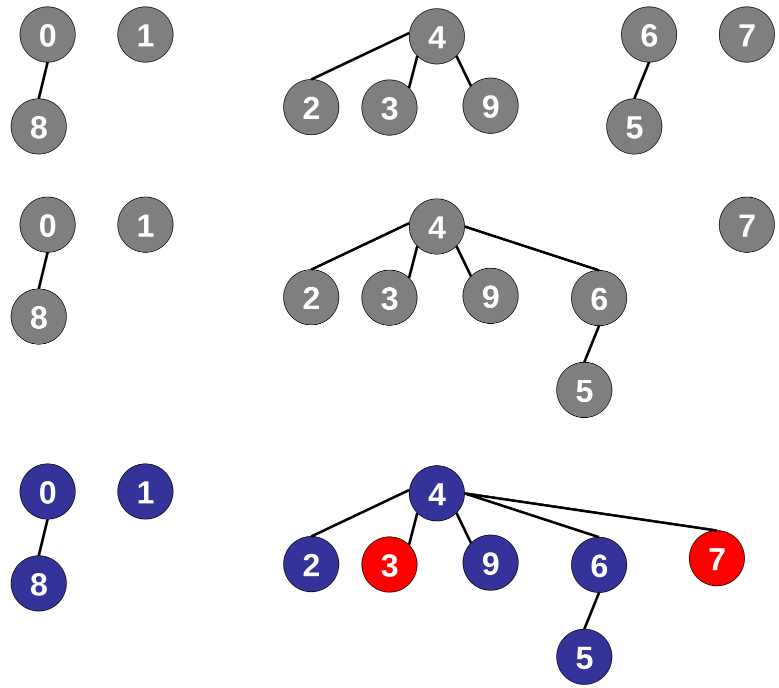
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	4	5	6	7	8	4
8-0	0	1	2	4	4	5	6	7	0	4
2-3	0	1	4	4	4	5	6	7	0	4
5-6	0	1	4	4	4	6	6	7	0	4
5-9	0	1	4	4	4	6	4	7	0	4



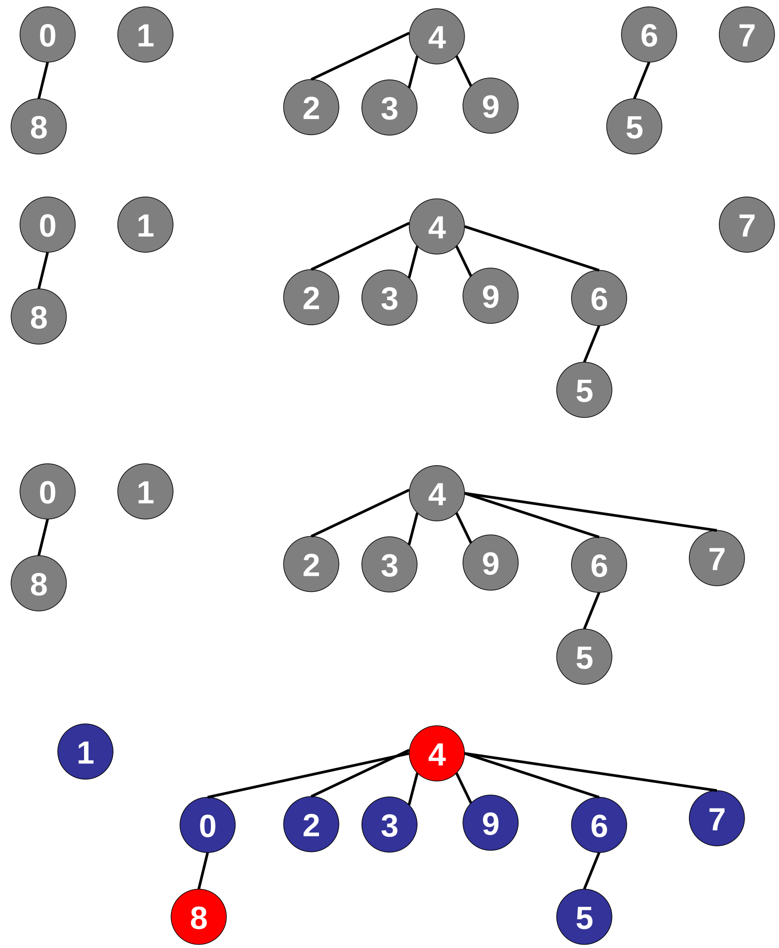
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	4	5	6	7	8	4
8-0	0	1	2	4	4	5	6	7	0	4
2-3	0	1	4	4	4	5	6	7	0	4
5-6	0	1	4	4	4	6	6	7	0	4
5-9	0	1	4	4	4	6	4	7	0	4
7-3	0	1	4	4	4	6	4	4	0	4



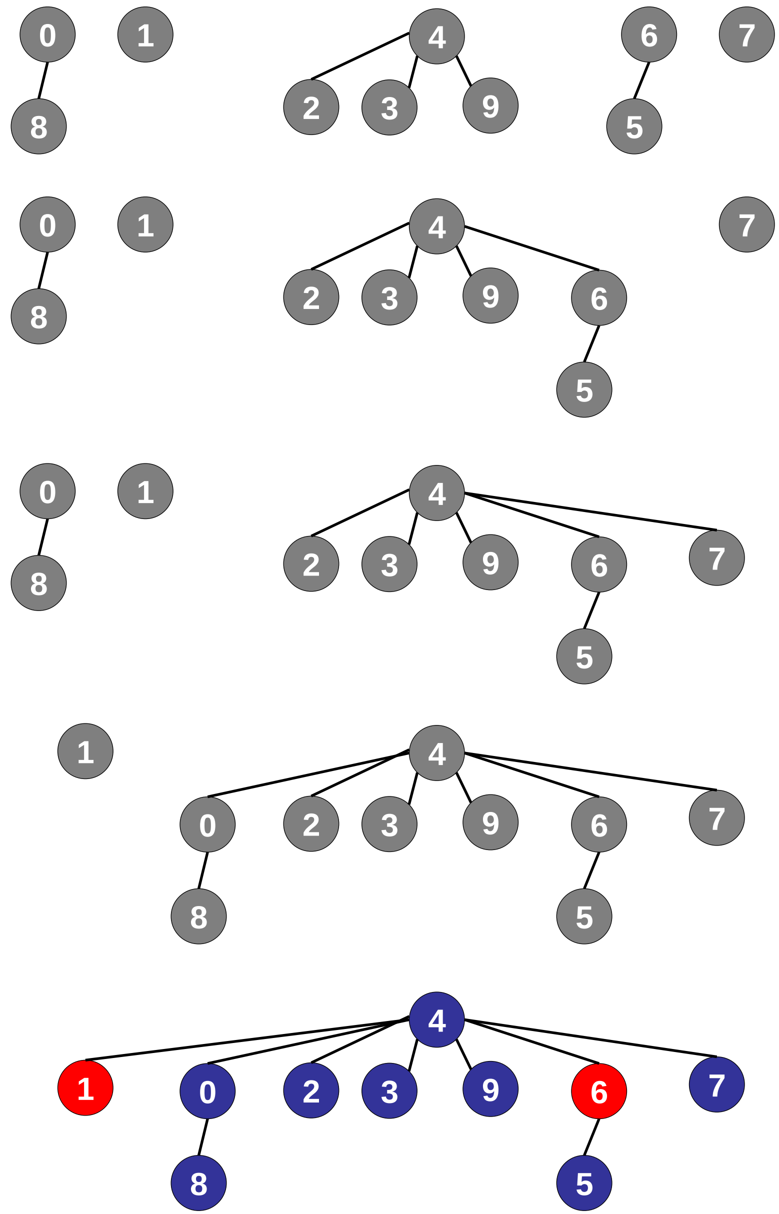
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	4	5	6	7	8	4
8-0	0	1	2	4	4	5	6	7	0	4
2-3	0	1	4	4	4	5	6	7	0	4
5-6	0	1	4	4	4	6	6	7	0	4
5-9	0	1	4	4	4	6	4	7	0	4
7-3	0	1	4	4	4	6	4	4	0	4
4-8	4	1	4	4	4	6	4	4	0	4



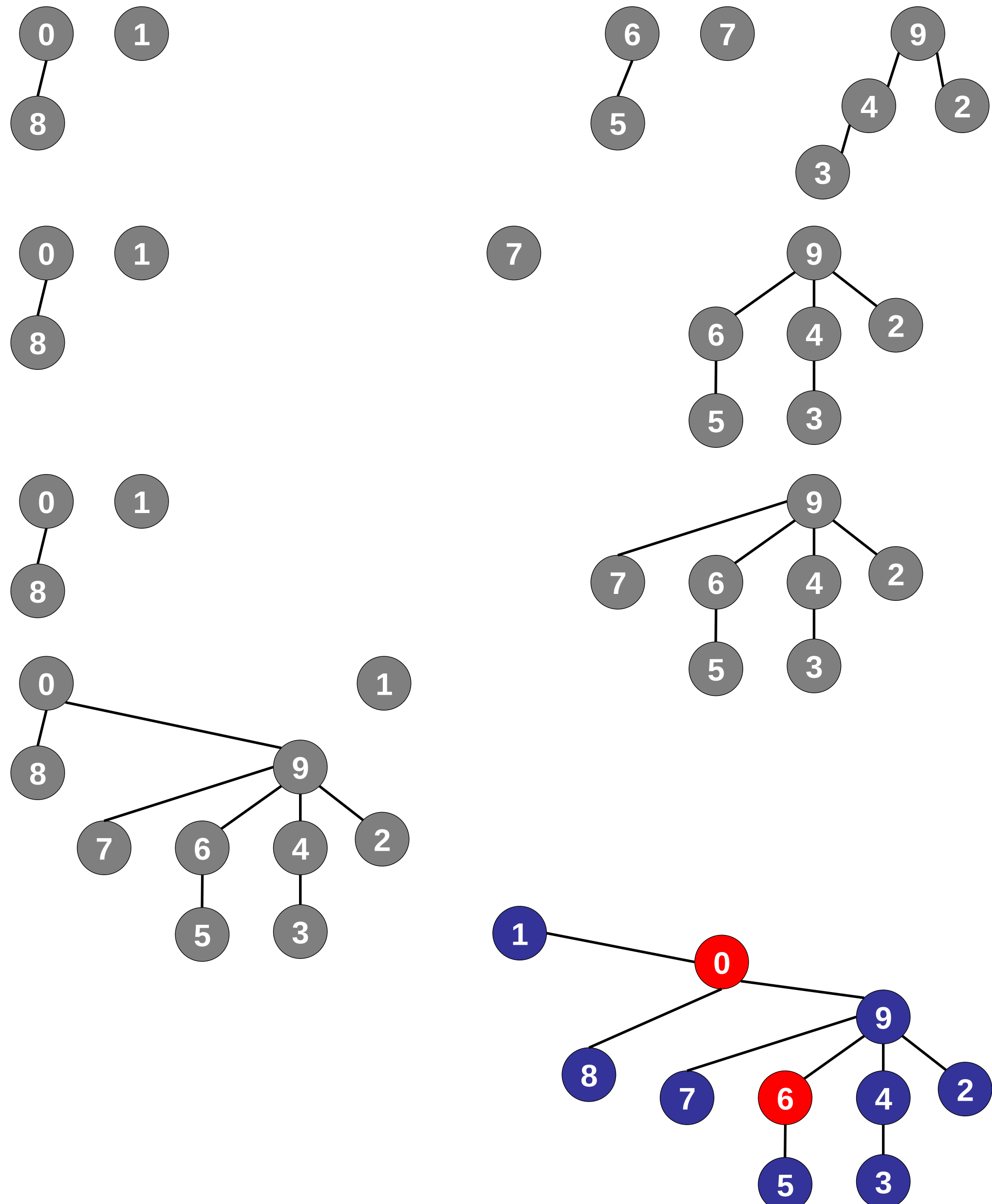
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	4	5	6	7	8	4
8-0	0	1	2	4	4	5	6	7	0	4
2-3	0	1	4	4	4	5	6	7	0	4
5-6	0	1	4	4	4	6	6	7	0	4
5-9	0	1	4	4	4	6	4	7	0	4
7-3	0	1	4	4	4	6	4	4	0	4
4-8	4	1	4	4	4	6	4	4	0	4
6-1	4	4	4	4	4	6	4	4	0	4



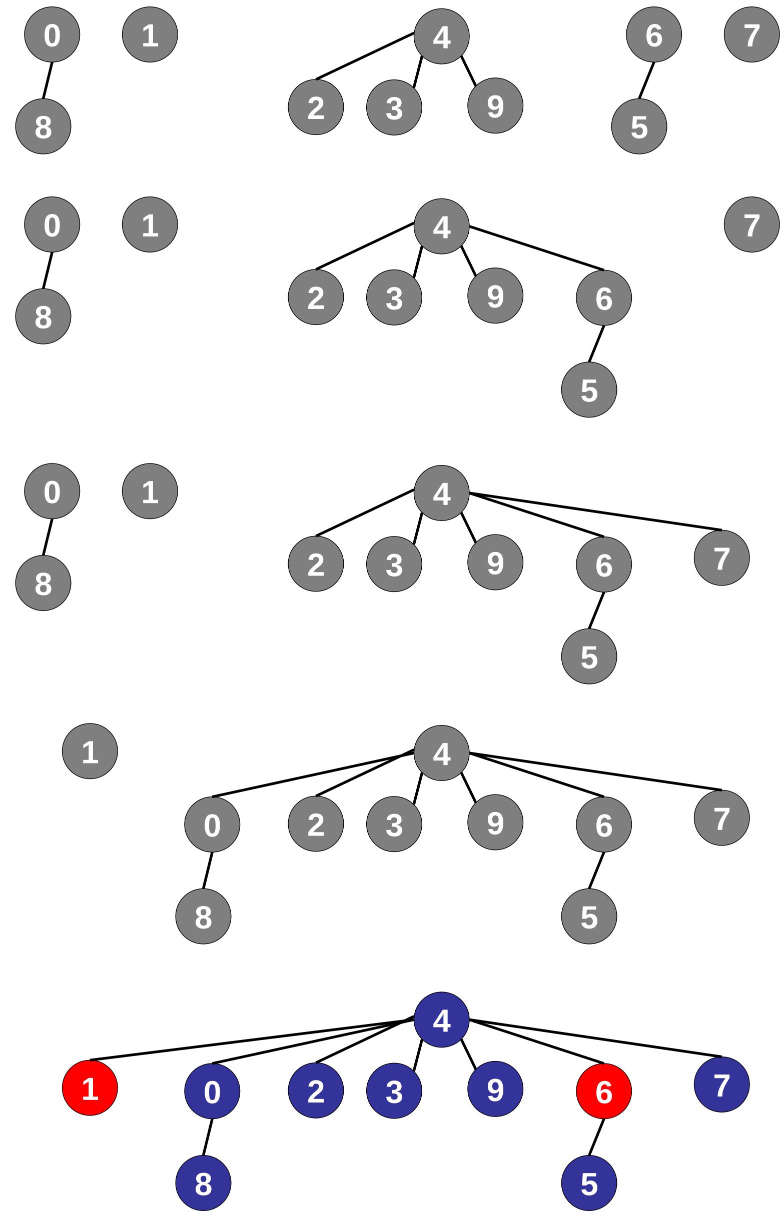
Example: (Unweighted) Quick Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9
5-9	0	1	9	4	9	6	9	7	0	9
7-3	0	1	9	4	9	6	9	9	0	9
4-8	0	1	9	4	9	6	9	9	0	0
6-1	1	1	9	4	9	6	9	9	0	0



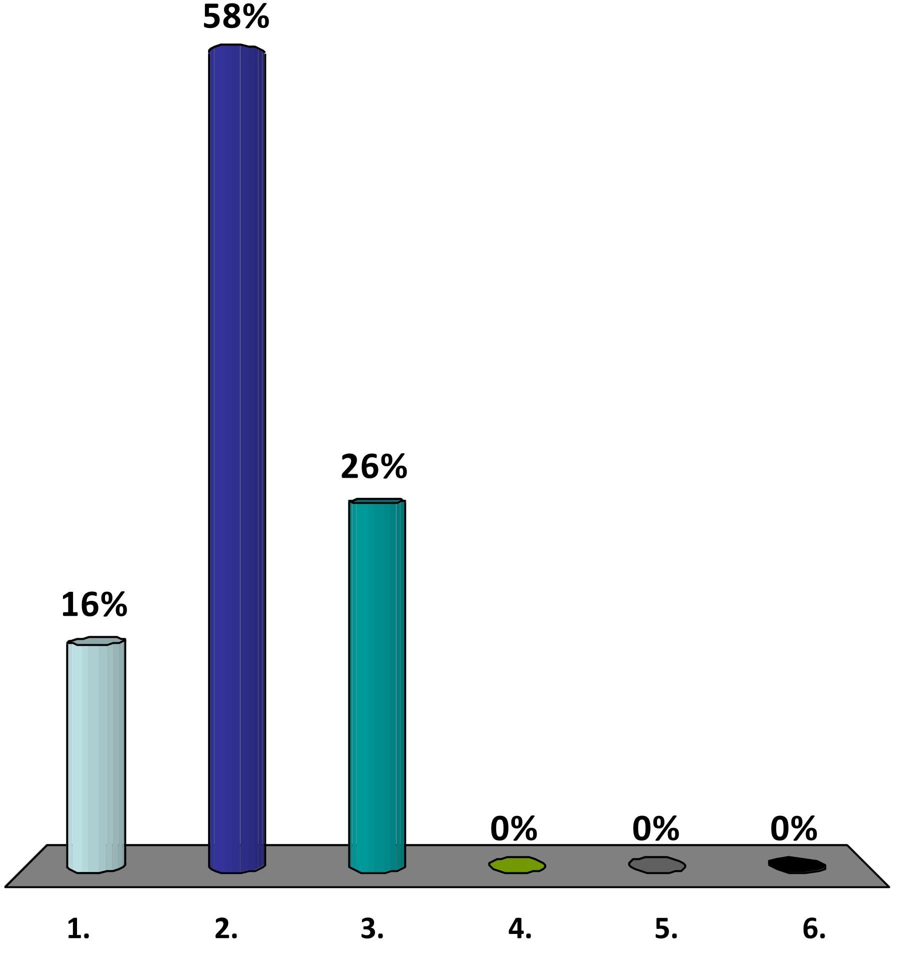
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	4	5	6	7	8	4
8-0	0	1	2	4	4	5	6	7	0	4
2-3	0	1	4	4	4	5	6	7	0	4
5-6	0	1	4	4	4	6	6	7	0	4
5-9	0	1	4	4	4	6	4	7	0	4
7-3	0	1	4	4	4	6	4	4	0	4
4-8	4	1	4	4	4	6	4	4	0	4
6-1	4	4	4	4	4	6	4	4	0	4



Maximum depth of tree?

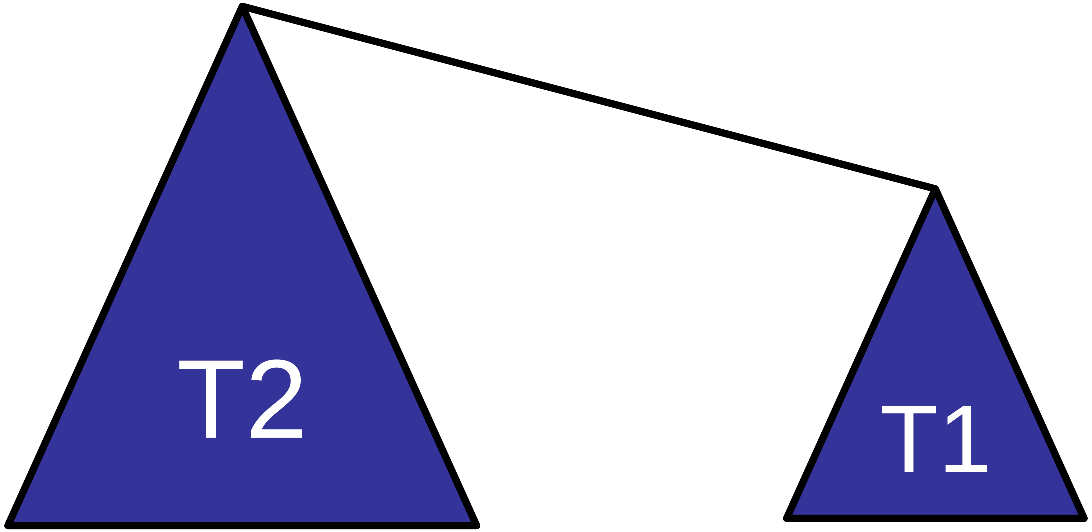
1. O(1)
2. O(log n)
3. O(n)
4. O(n log n)
5. O(n^2)
6. None of the above.



Weighted Union

Analysis:

– Base case: tree of height 0 contains 1 object.



Weighted Union

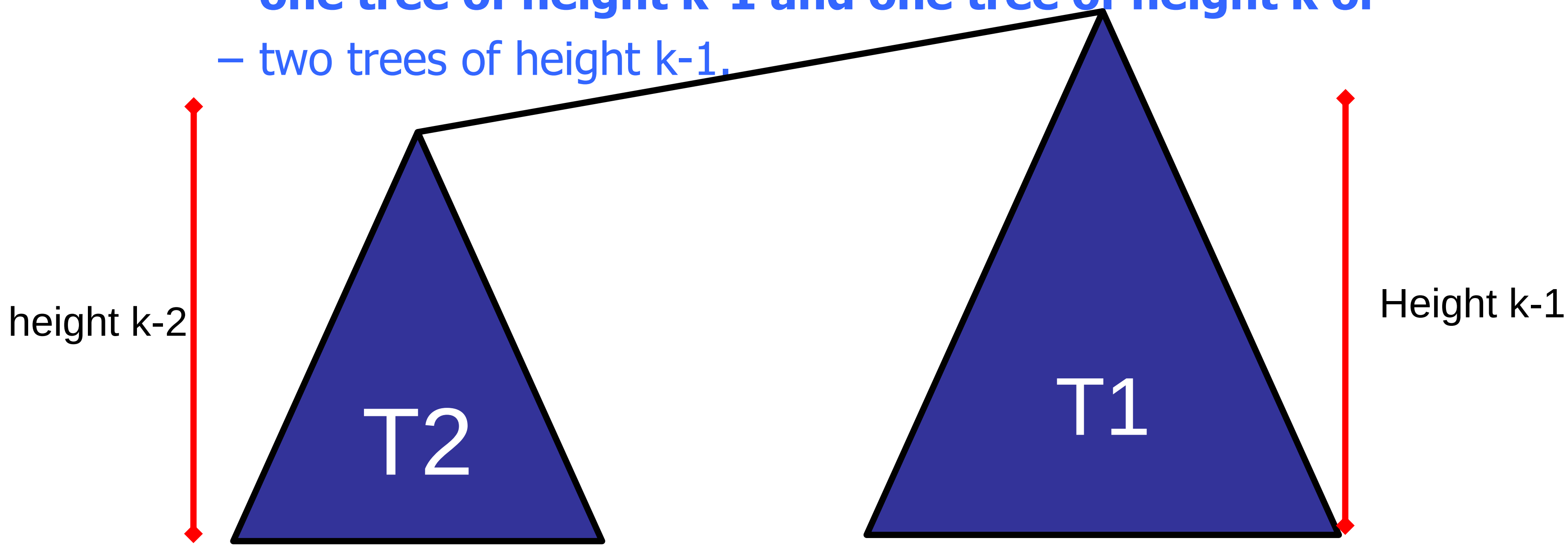
Analysis:

- Base case: tree of height 0 contains 1 object.
- Induction:
  - Induction: a tree of height k-1 contains at least $2^{(k-1)}$ objects.
  - A tree of height k is built from
    - one tree of height k-1 and one tree of height k or
    - two trees of height k-1.

Weighted Union

Analysis:

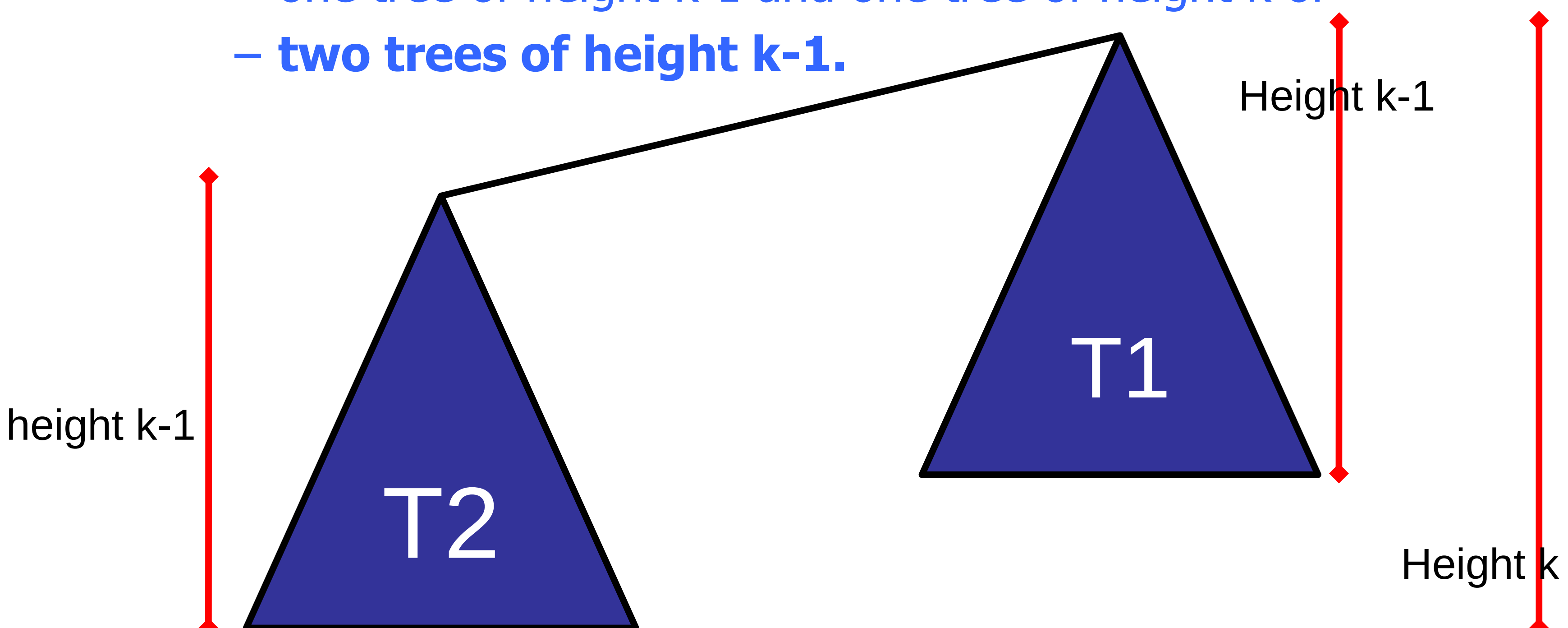
- Base case: tree of height 0 contains 1 object.
- Induction:
  - A tree of height k is built from
    - **one tree of height k-1 and one tree of height k or**
    - two trees of height k-1.



Weighted Union

Analysis:

- Base case: tree of height 0 contains 1 object.
- Induction:
  - A tree of height k is built from
    - one tree of height k-1 and one tree of height k or
    - **two trees of height k-1.**



Weighted Union

Analysis:

- Base case: tree of height 0 contains 1 object.
- Induction:
  - Tree of height k is built from two trees of height k-1.
  - Induction: a tree of height k-1 contains at least $2^{(k-1)}$ objects.
  - Conclusion: a tree of height k contains 2^k objects.

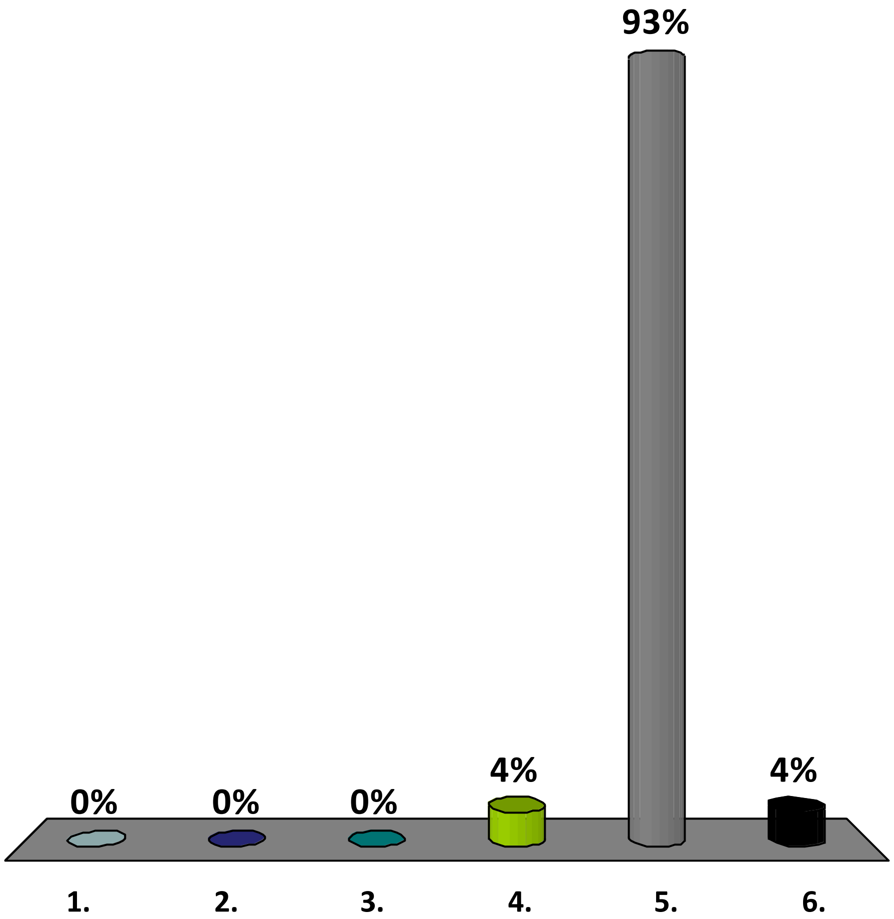
Weighted Union

Analysis:

- Base case: tree of height 0 contains 1 object.
- Induction:
  - Tree of height k is built from two trees of height k-1.
  - Induction: a tree of height k-1 contains at least $2^{(k-1)}$ objects.
  - Conclusion: a tree of height k contains 2^k objects.
- Conclusion:
  - Each tree is of height O(log n)

Running time of (Find, Union):

1. O(1), O(1)
2. O(1), O(n)
3. O(n), O(1)
4. O(n), O(n)
5. O(log n), O(log n)
6. None of the above.



Weighted Union

```
union(int p, int q) {
   while (parent[p] !=p) p = parent[p];
   while (parent[q] !=q) q = parent[q];
   if (size[p] > size[q] {
           parent[q] = p;    // Link q to p
           size[p] = size[p] + size[q];
   }
   else {
           parent[p] = q; // Link p to q
           size[q] = size[p] + size[q];
   }
}
```

Union-Find Summary

Quick-find and Quick-union are slow:

- Union and/or find is expensive
- Quick-union: tree is too deep

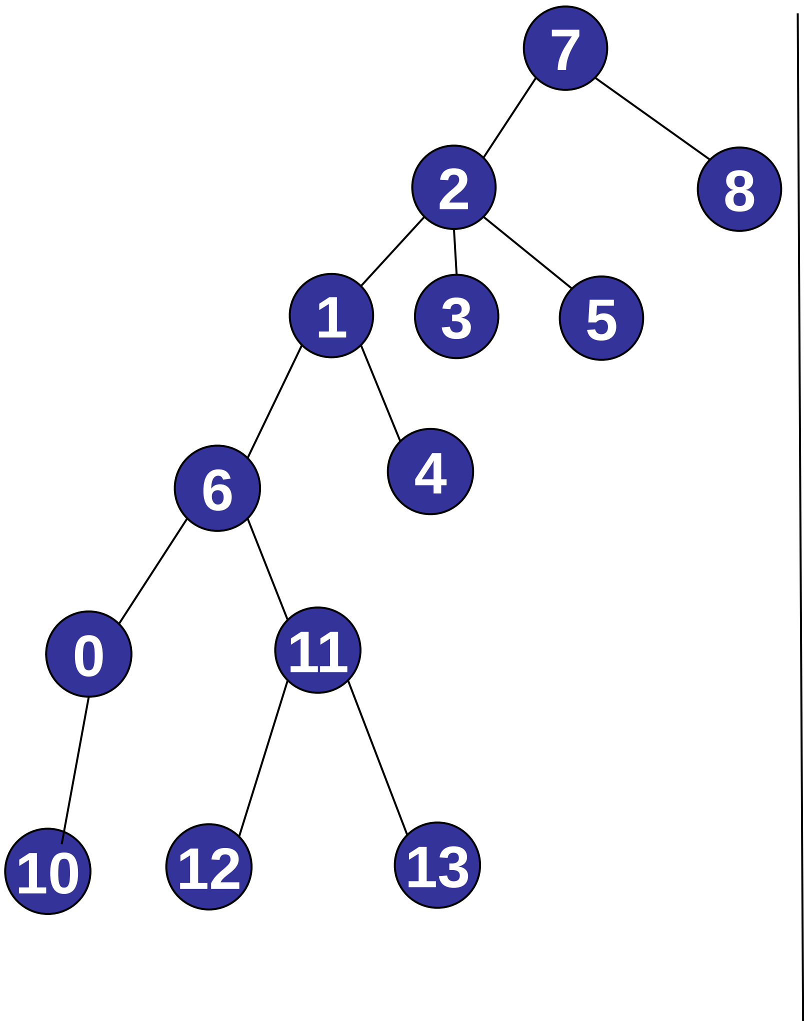
Weighted-union is faster:

- Trees too balanced: O(log n)
- Union *and* find are O(log n)

	find	union
quick-find	O(1)	O(n)
quick-union	O(n)	O(n)
weighted-union	O(log n)	O(log n)

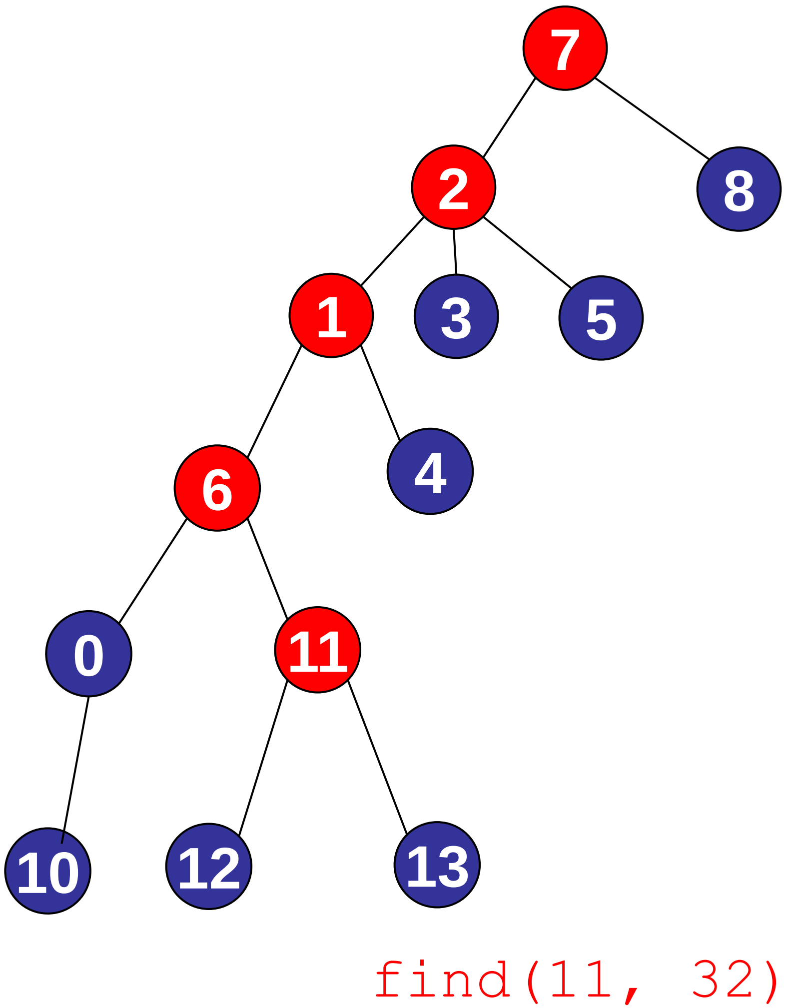
Path Compression

After finding the root: set the parent of each traversed node to the root.



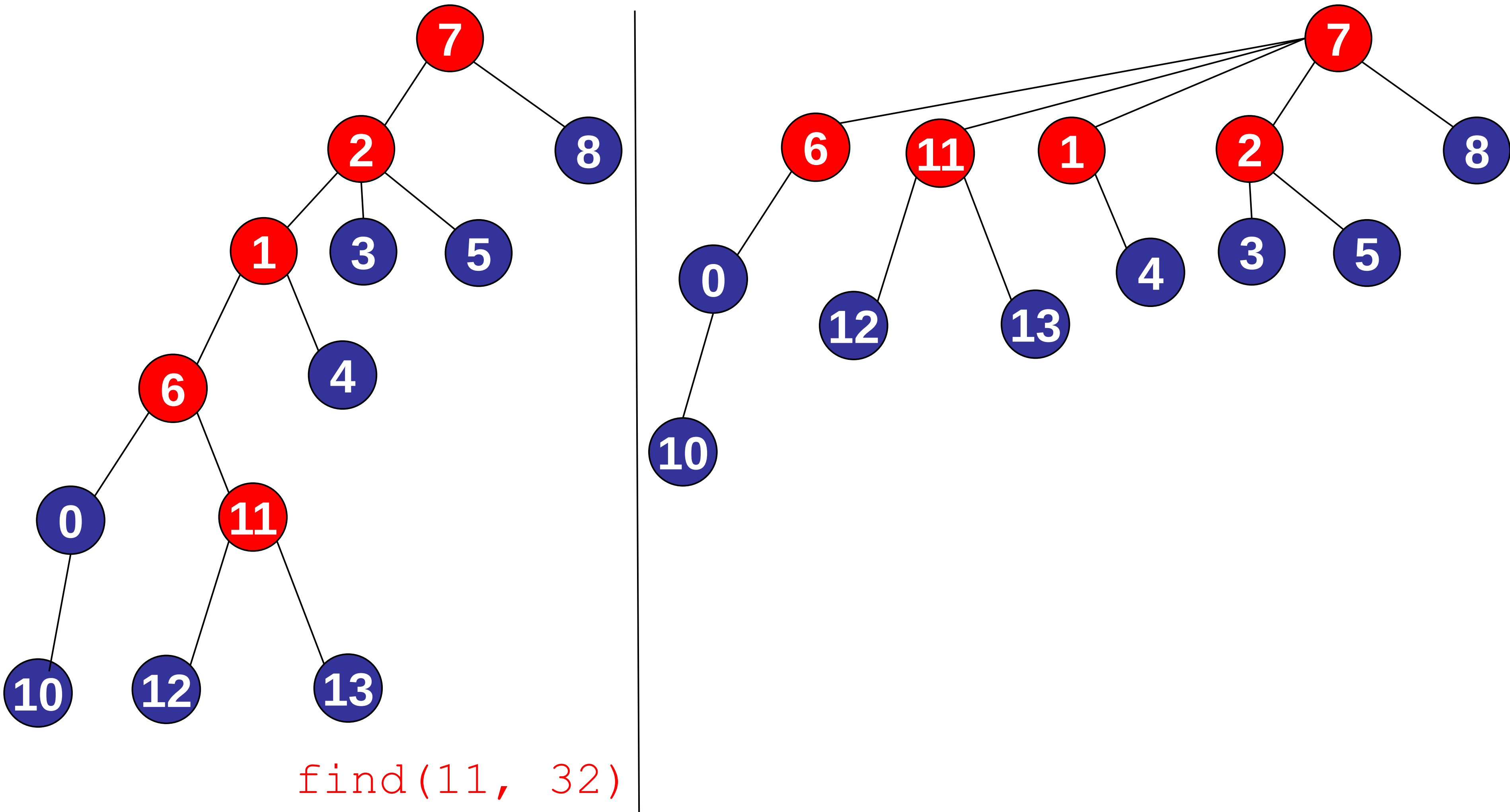
Path Compression

After finding the root: set the parent of each traversed node to the root.



Path Compression

After finding the root: set the parent of each traversed node to the root.



Path Compression

```
findRoot(int p) {
   root = p;
   while (parent[root] != root) root = parent[root];
   return root;
}
```

Path Compression

```
findRoot(int p) {
  root = p;
  while (parent[root] != root) root = parent[root];
  while (parent[p] != p) {
        temp = parent[p];
        parent[p] = root;
        p = temp;
  }
  return root;
}
```

Alternative Path Compression

```
findRoot(int p) {
   root = p;
   while (parent[root] != root) {
         parent[root] = parent[parent[root]];
         root = parent[root];
   }
   return root;
}
```

Make every other node in the path point to its grandparent!

- Simple
- Works as well!

Weight Union with Path Compression

Theorem: [Tarjan 1975]

Starting from empty, any sequence of m union/find operations on n objects takes: $O(n + m\alpha(m, n))$time.

Weight Union with Path Compression

Theorem: [Tarjan 1975]

Starting from empty, any sequence of m union/find operations on n objects takes: $O(n + m\alpha(m, n))$time.

Inverse Ackermann function: always ≤ 5 in this universe.

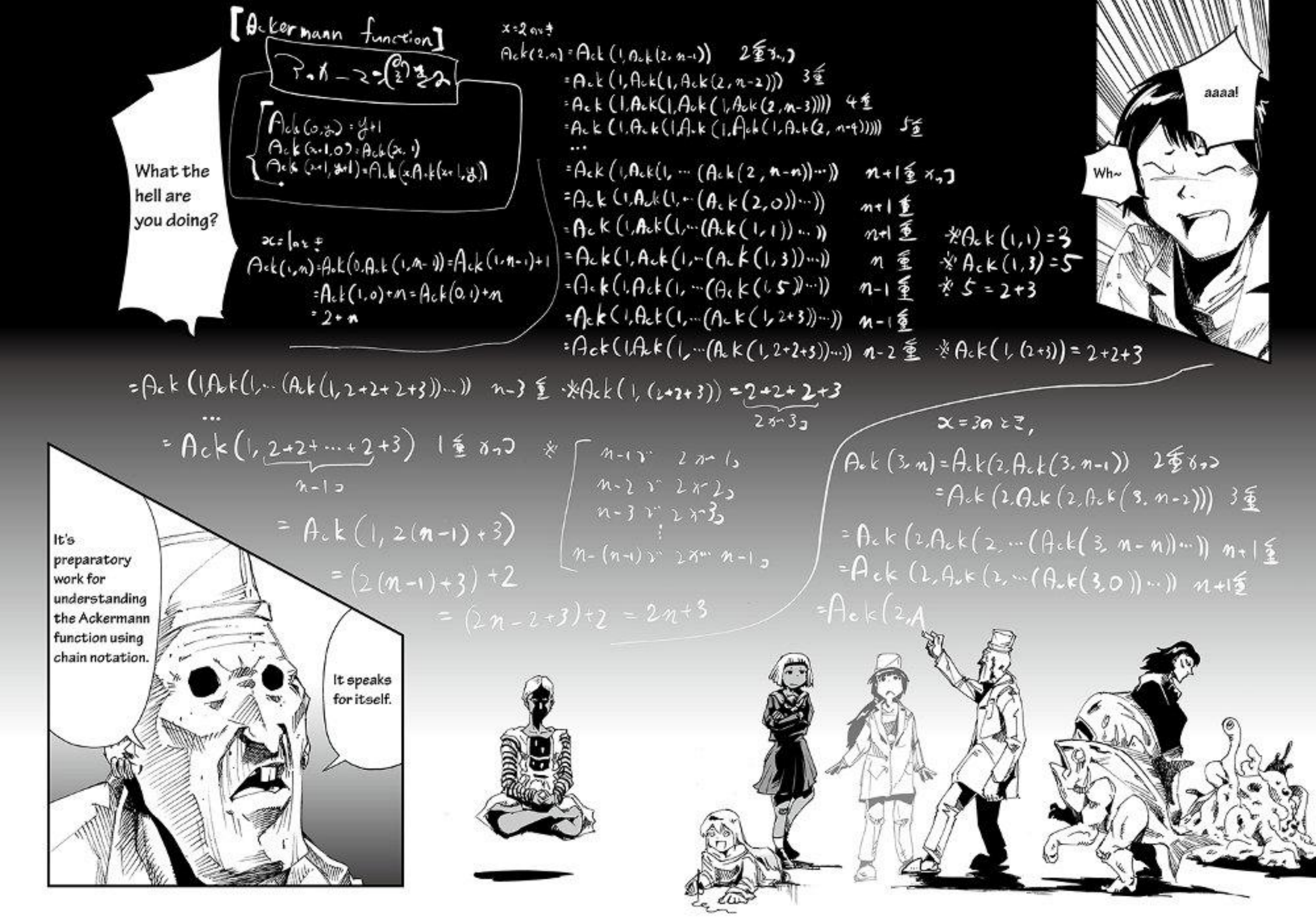
n	α(n, n)
4	0
8	1
32	2
8,192	3
2^{65533}	4

Weight Union with Path Compression

Theorem: [Tarjan 1975]

Starting from empty, any sequence of m union/find operations on n objects takes: $O(n + m\alpha(m, n))$ time.

Proof:


aaaa!
Wh~
What the hell are you doing?
[Ackermann function]
Ack(0,y)=y+1
Ack(x+1,0)=Ack(x,1)
Ack(x+1,y+1)=Ack(x,Ack(x+1,y))
Ack(1,n)=Ack(0,Ack(1,n-1))=Ack(1,n-1)+1
=Ack(1,0)+n=Ack(0,1)+n
=2+n
=Ack(1,2(n-1)+3)
=(2(n-1)+3)+2
=(2n-2+3)+2 = 2n+3
It speaks for itself.
It's preparatory work for understanding the Ackermann function using chain notation.

Weight Union with Path Compression

Theorem: [Tarjan 1975]

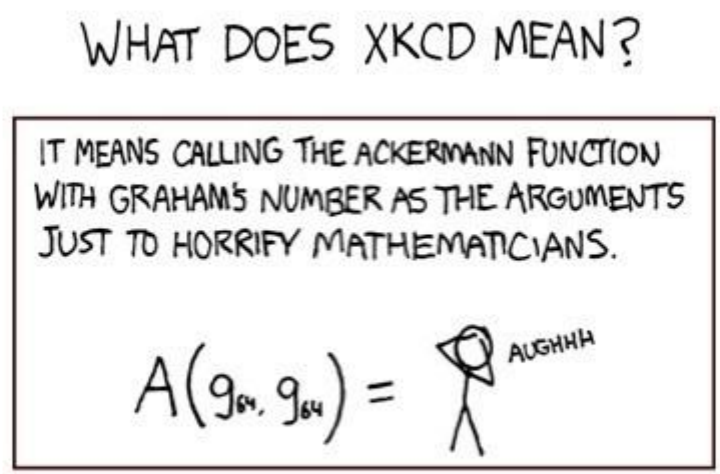
Starting from empty, any sequence of m union/find operations on n objects takes: $O(n + m\alpha(m, n))$ time.

Proof:

– Very difficult.

– Algorithm:

- very simple to implement.



Weight Union with Path Compression

Theorem: [Tarjan 1975]

Starting from empty, any sequence of m union/find operations on n objects takes: $O(n + m\alpha(m, n))$time.

Proof:

– Very difficult.

– Algorithm: very simple to implement.

Can we do better? No!

– Proof: impossible to achieve linear time.

Union-Find Summary

Weighted-union is faster:

- Trees are flat: O(log n)
- Union *and* find are O(log n)

Weighted Union + Path Compression is very fast:

- Trees very flat.
- On average, almost linear performance per operation.

	find	**union**
quick-find	O(1)	O(n)
quick-union	O(n)	O(n)
weighted-union	O(log n)	O(log n)
weighted-union with path-compression	$\alpha(m, n)$	$\alpha(m, n)$

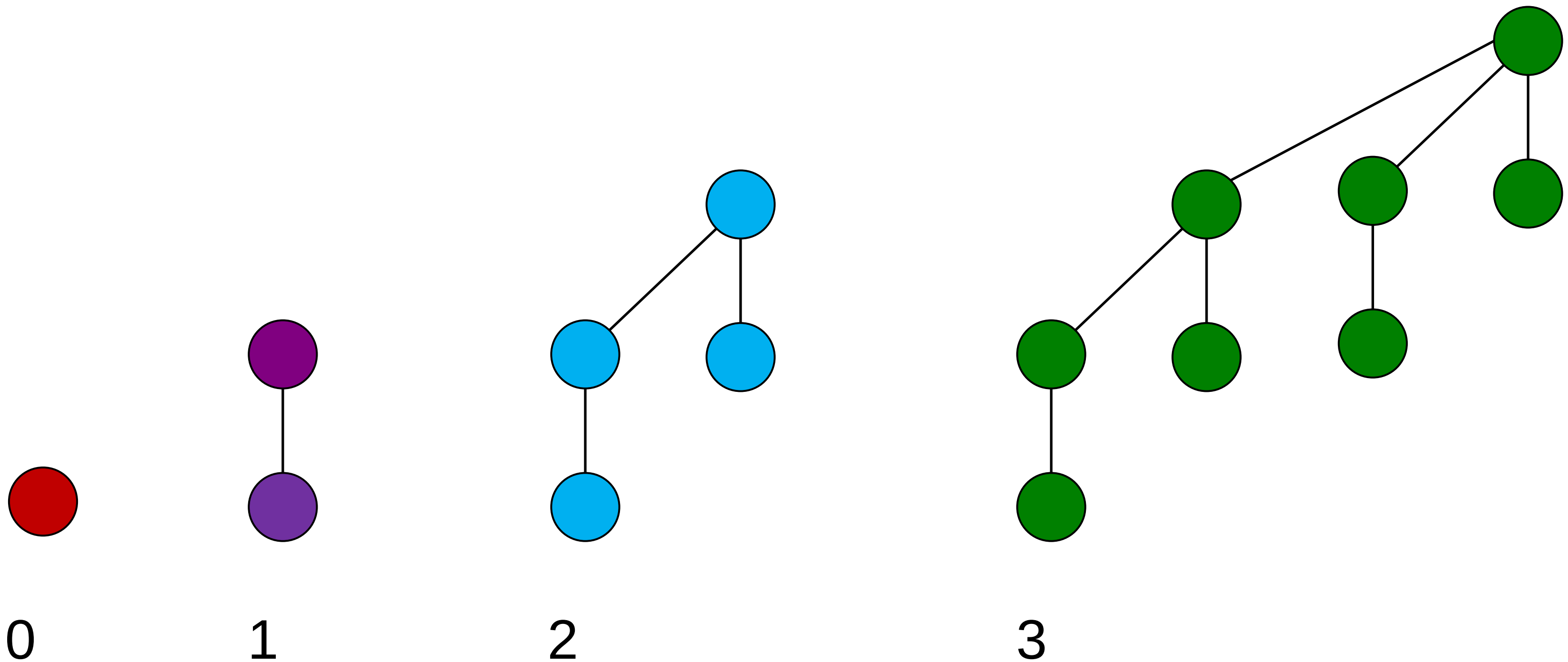
Union-Find Summary

Path Compression **without** weighted union?

	find	**union**
quick-find	O(1)	O(n)
quick-union	O(n)	O(n)
weighted-union	O(log n)	O(log n)
path compression	O(log n)	O(log n)
weighted-union with path-compression	$\alpha(m, n)$	$\alpha(m, n)$

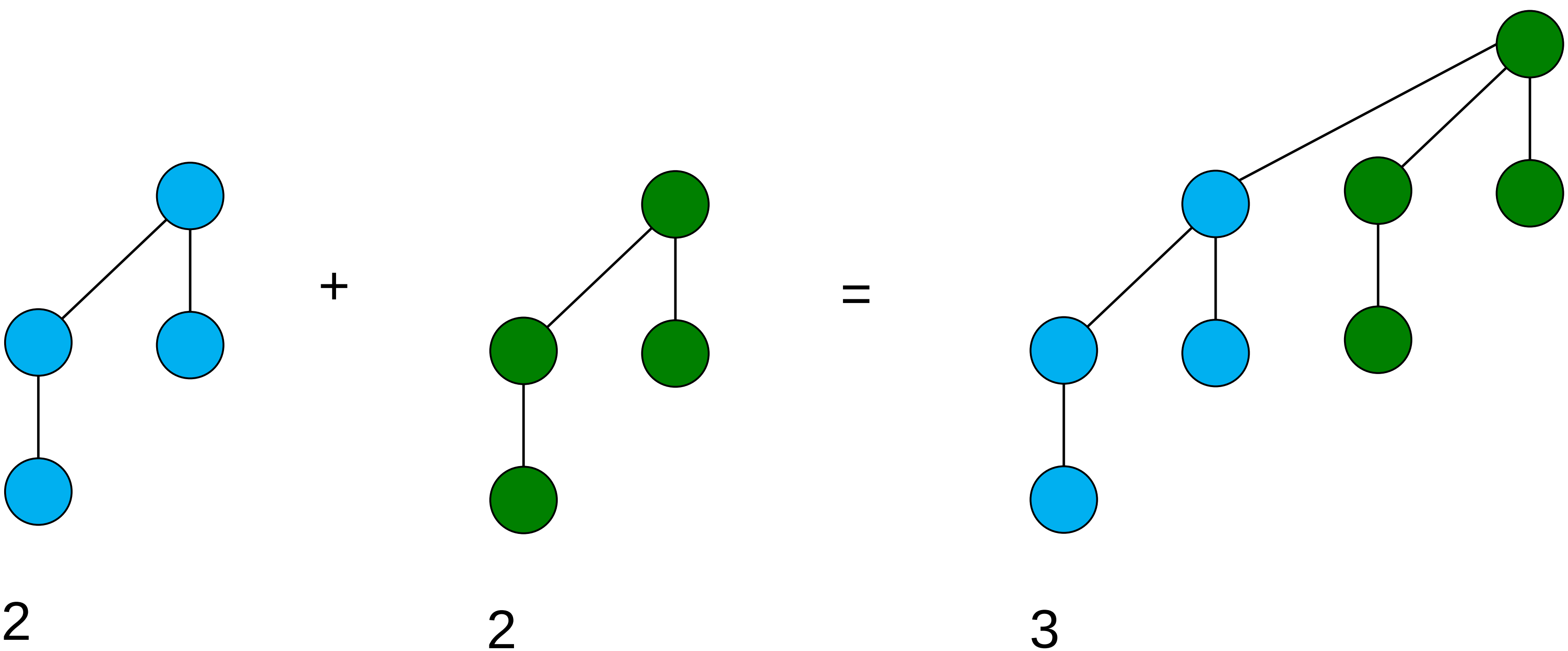
Binomial Trees:

Order k binomial trees



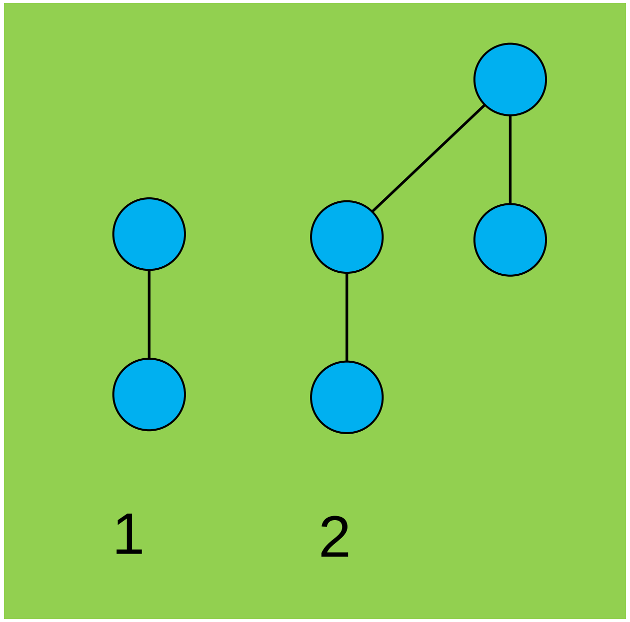
Binomial Trees:

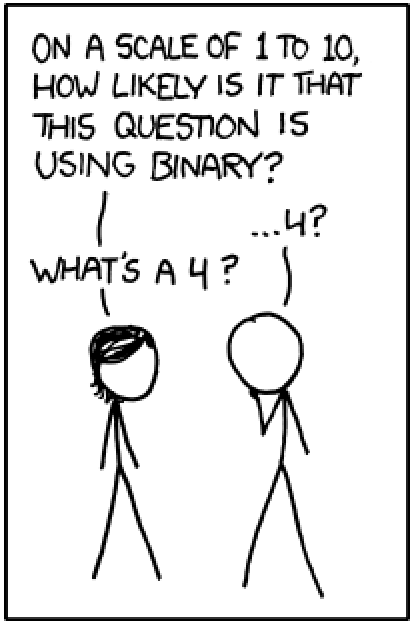
Two order k binomial trees can be merged into one k+1 binomial tree



Binomial Heap: To store n items

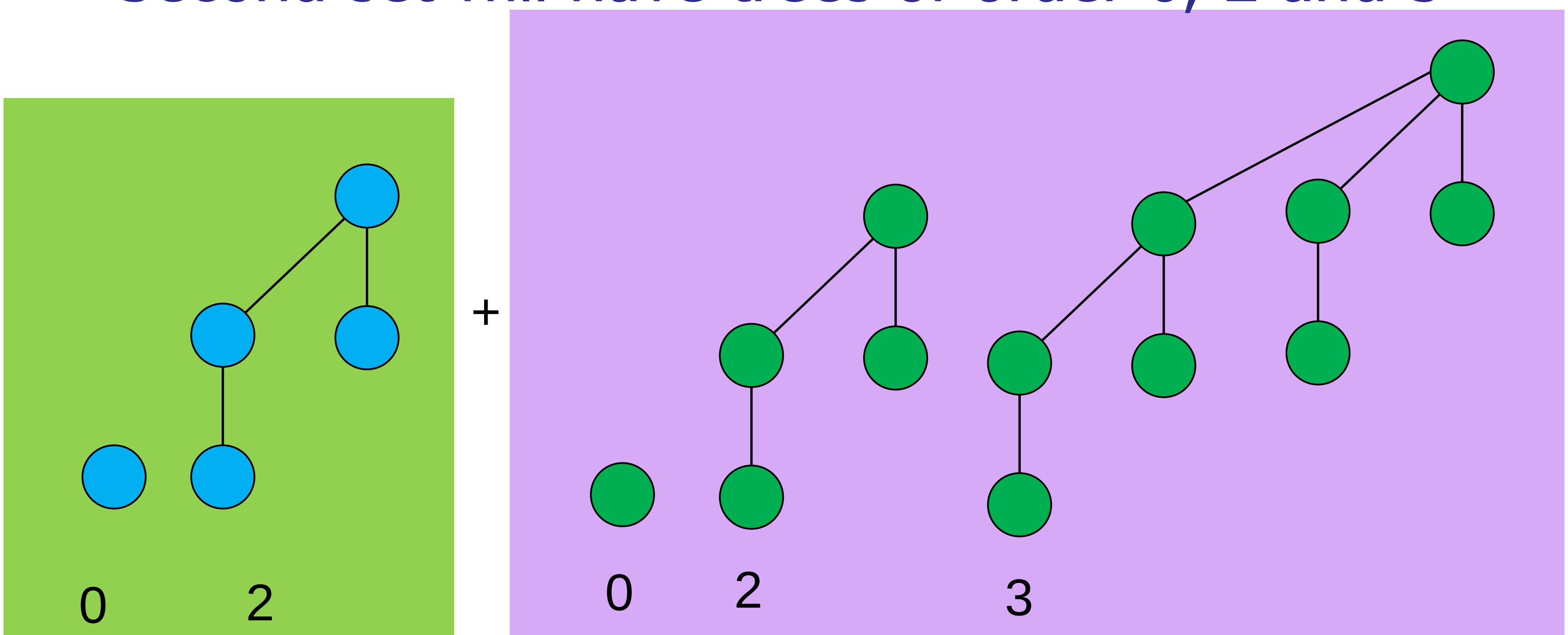
- Convert n into binary
- The number of binomial trees we need is the number of "1" of n in binary form
- E.g. if we have 6 items in ONE set, we need two binomial trees of order 1 and 2
  - $6_{(dec)} = 110_{(bin)}$





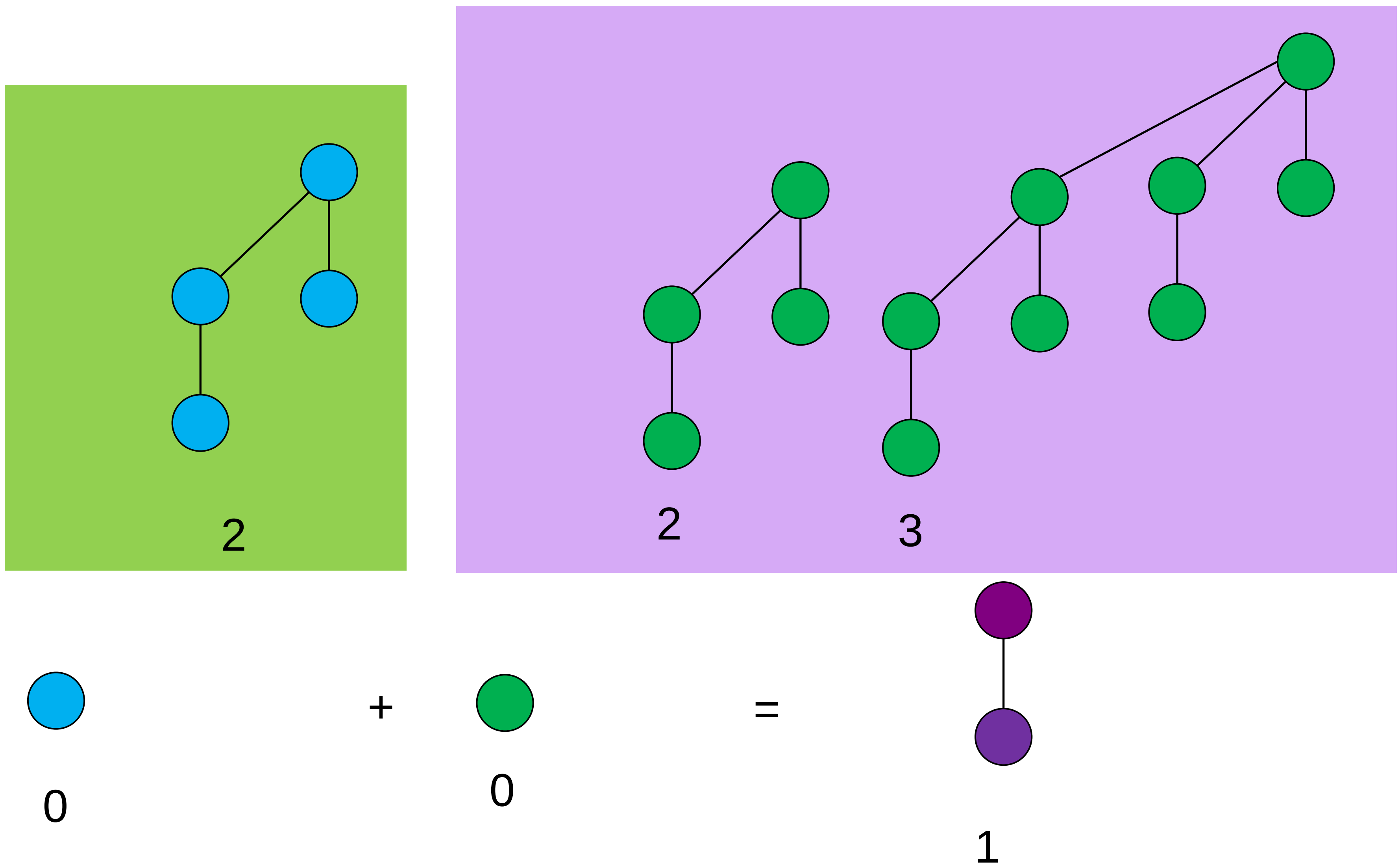
Union of Two Binomial Heap

- So it’s like adding two binary numbers
- E.g. two sets with cardinalities 5 and 13
  - First set will have trees of order 0 and 2
  - Second set will have trees of order 0, 2 and 3



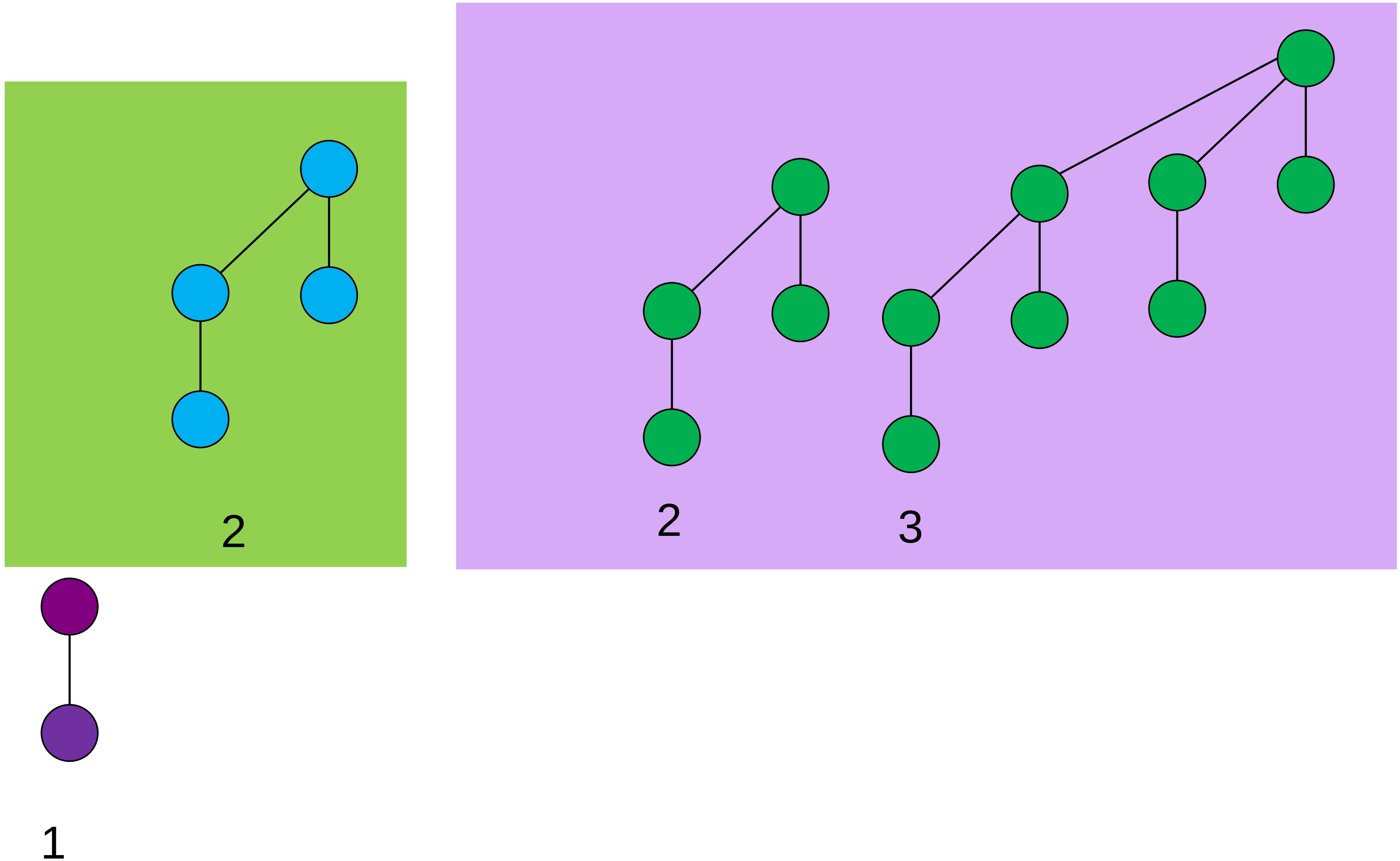
Union of Two Binomial Heap

- Merge from lowest order first



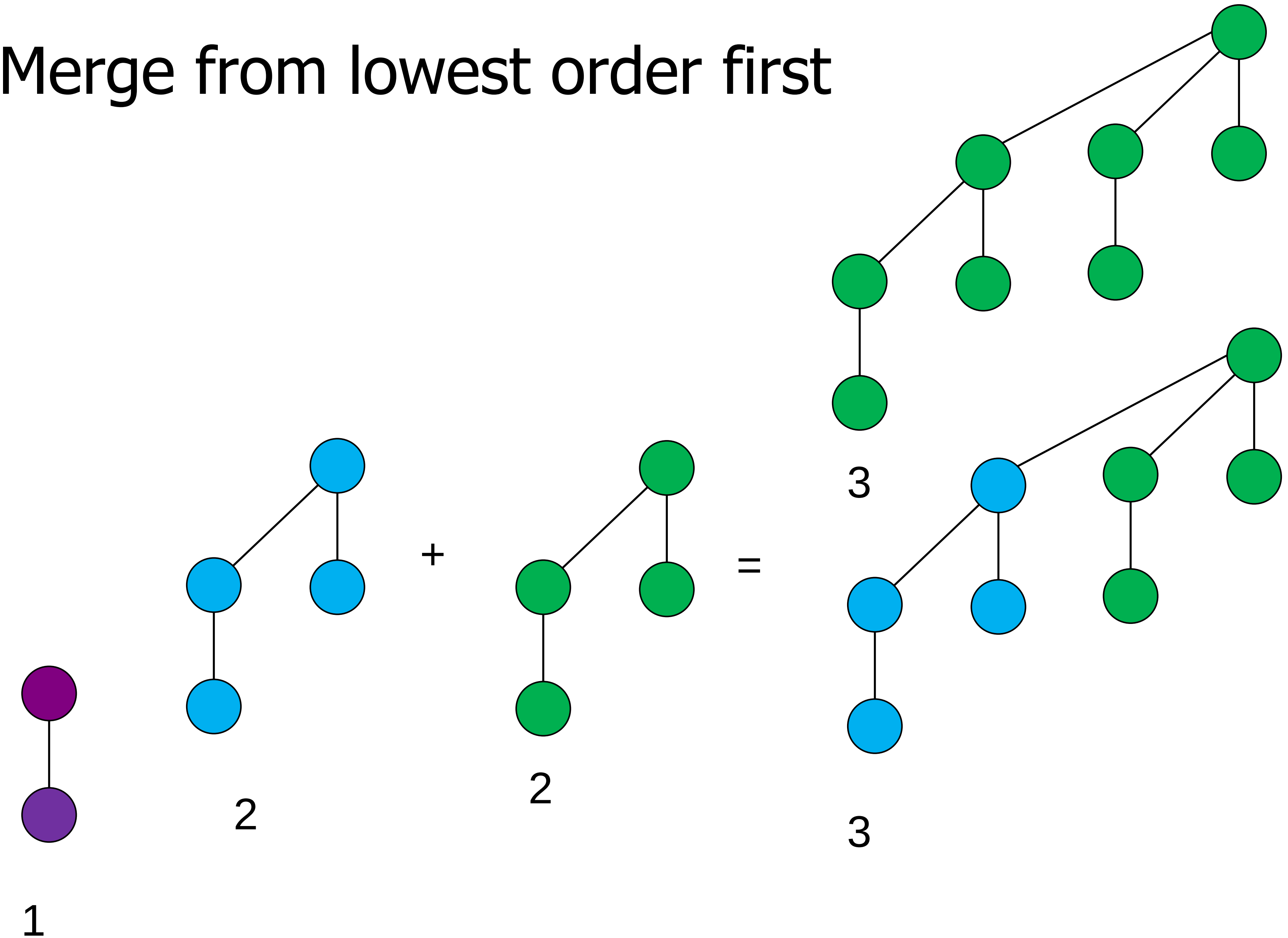
Union of Two Binomial Heap

- Merge from lowest order first



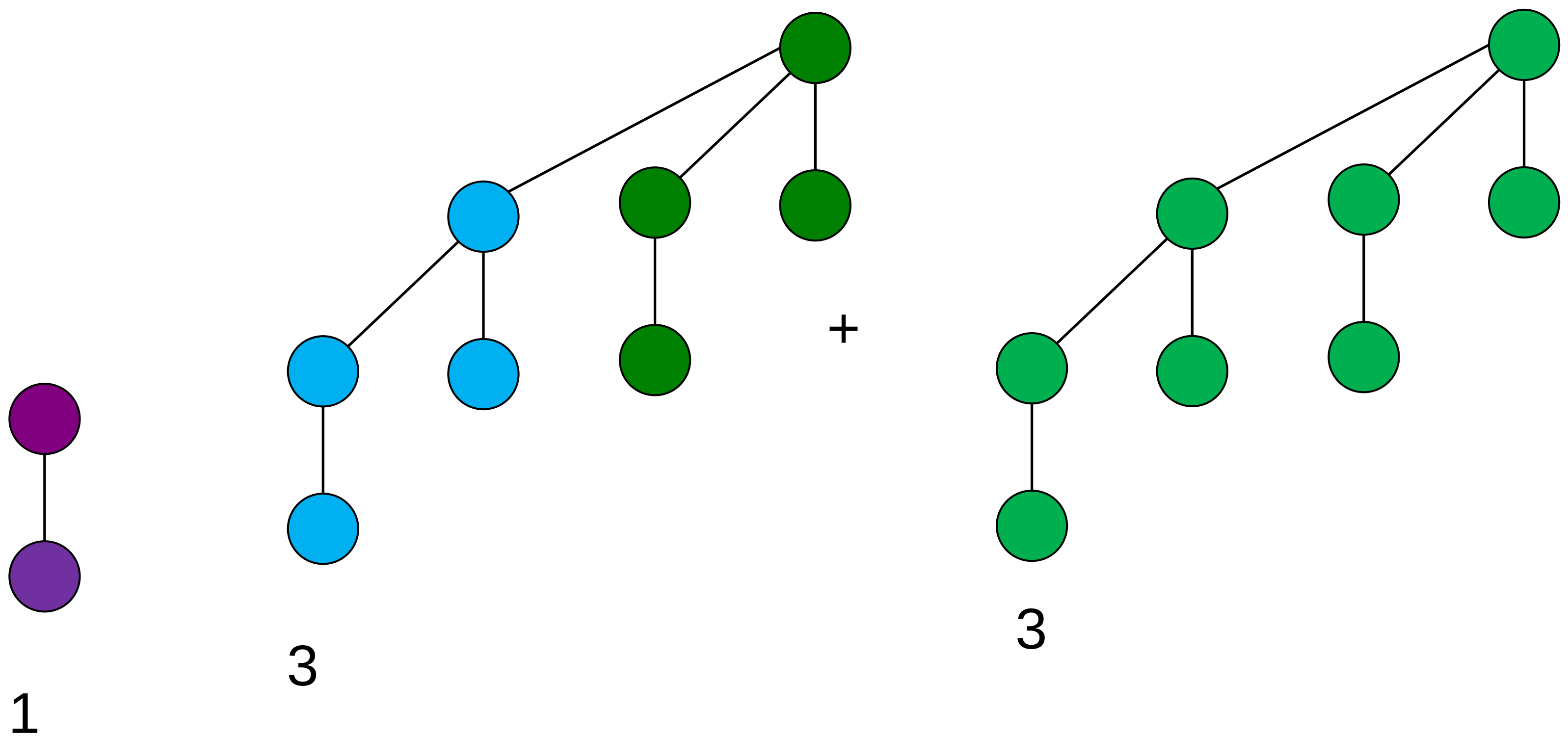
Union of Two Binomial Heap

- Merge from lowest order first



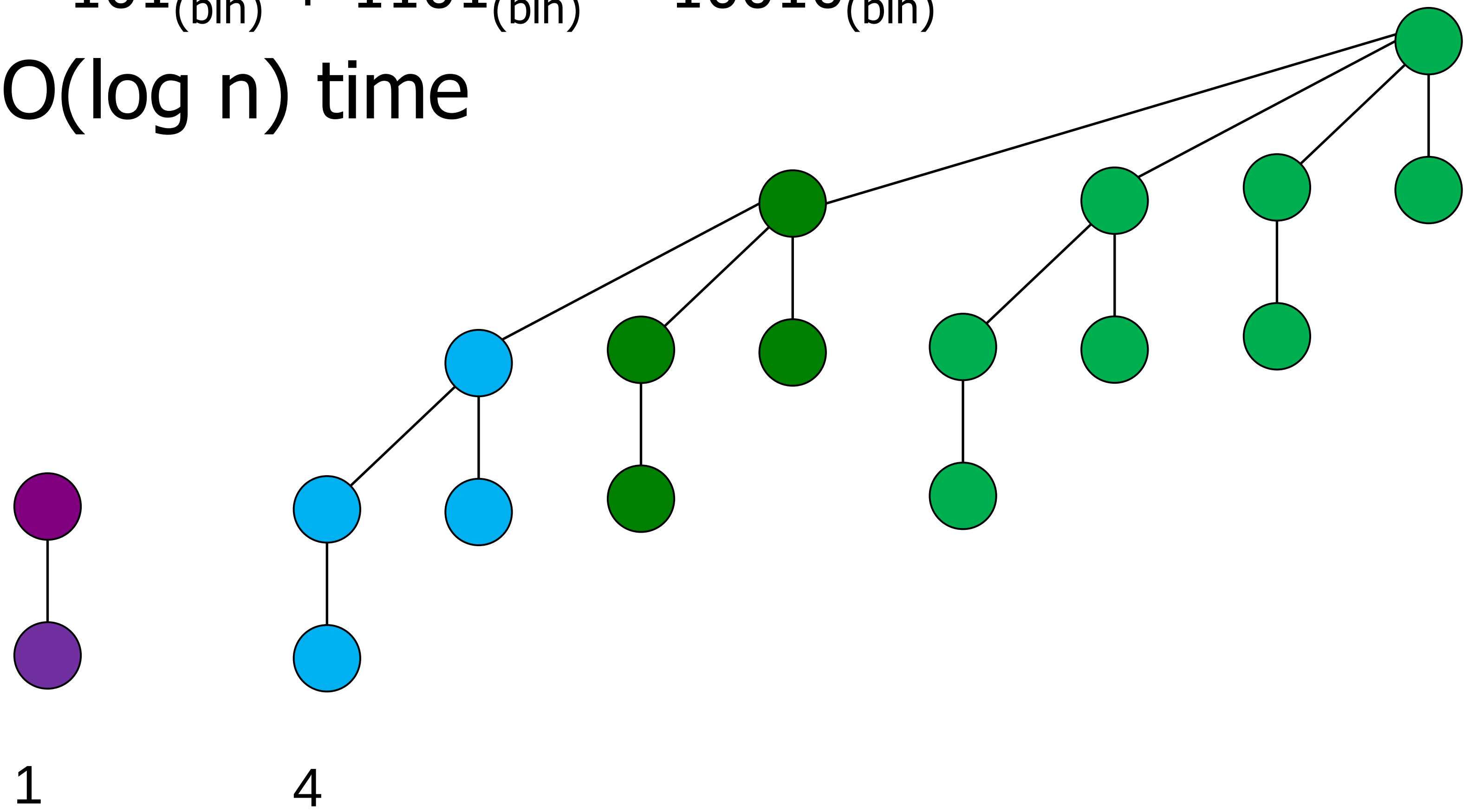
Union of Two Binomial Heap

- Merge from lowest order first



Union of Two Binomial Heap

- Merge from lowest order first
  - $5_{(dec)} + 13_{(dec)} = 18_{(dec)}$
  - $101_{(bin)} + 1101_{(bin)} = 10010_{(bin)}$
- O(log n) time



Union-Find Summary

Path Compression **without** weighted union?

	find	union
quick-find	O(1)	O(n)
quick-union	O(n)	O(n)
weighted-union	O(log n)	O(log n)
path compression	O(log n)	O(log n)
weighted-union with path-compression	$\alpha(m, n)$	$\alpha(m, n)$

Union-Find Summary

What about Union-Split-Find?

- Insert and delete edges.
- New result: 2013!!

Dynamic graph connectivity in polylogarithmic worst case time

Bruce M. Kapron * Valerie King * Ben Mountjoy *

Abstract

The dynamic graph connectivity problem is the following: given a graph on a fixed set of n nodes which is undergoing a sequence of edge insertions and deletions, answer queries of the form $q(a, b)$: "Is there a path between nodes a and b?" While data structures for this problem with polylogarithmic *amortized* time per operation have been known since the mid-1990's, these data structures have $\Theta(n)$ worst case time. In fact, no previously known solution has worst case time per operation which is $o(\sqrt{n})$.

We present a solution with worst case times $O(\log^4 n)$ per edge insertion, $O(\log^5 n)$ per edge deletion, and $O(\log n / \log\log n)$ per query. The answer to each query is correct if the answer is "yes" and is correct with high probability if the answer is "no". The data structure is based on a simple novel idea which can be used to quickly identify an edge in a cutset.

Our technique can be used to simplify and significantly

Though the problem of improving the worst case update time from $O(\sqrt{n})$ has been posed in the literature many times, there has been no improvement since 1985. In the words of Pătraşcu and Thorup, it is "perhaps the most fundamental challenge in dynamic graph algorithms today" [11].

Nearly every dynamic connectivity data structure maintains a spanning forest F. Dealing with edge insertions is relatively easy. The challenge is to find a replacement edge when a tree edge is deleted, splitting a tree into two subtrees. A replacement edge is an edge reconnecting the two subtrees, or, in other words, in the cutset of the cut $(T, V \setminus T)$ where T is one of the subtrees. An edge with both endpoints in the same subtree we call *internal* to the tree.

Roadmap

Part I: Priority Queues

- Binary Heaps
- HeapSort

Part II: Disjoint Set

- Problem: Dynamic Connectivity
- Algorithm: Union-Find
- Applications

Applications

Many applications:

- Mazes
  - Are two locations connected?

- Games:
  - Can you get from one state to another?

Applications

Many applications:

- Networks
  - Are two locations connected?
- Least-common-ancestor:
  - Which node in a tree network is the closest ancestor?

Applications

Many applications:

- Programming languages
  - Hinley-Milner polymorphic type inference
  - Equivalence of finite state automata
  - Image processing in Matlab

- Physics:
  - Hoshen-Kopelman algorithm
  - Percolation
  - Conductance / insulation

Applications

Many applications:

- Topology in Molecular Design

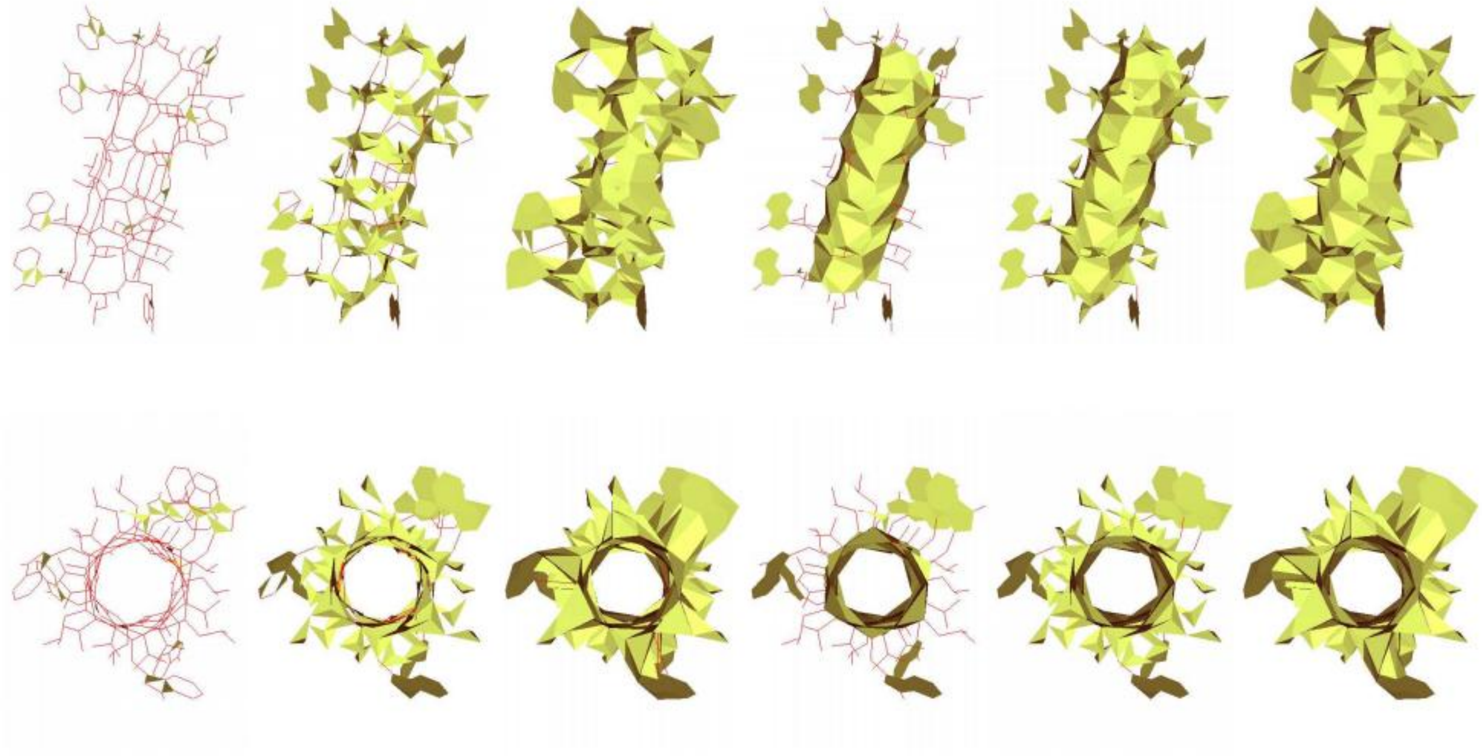


Figure 19. Side and top views of complexes $K_{715}, K_{1431}, K_{2682}$ of Gramicidin A are shown in the left three columns. The corresponding 2688-persistent complexes are shown on the right.