CS2040S
Data Structures and Algorithms

Two Interesting Data Structures:
Heaps + Union-Find

Intermission (a break from graphs)

Part I: Implementing a Priority Queue

- Binary Heaps
- HeapSort

Part II: Disjoint Set

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

Intermission (a break from graphs)

Part I: Implementing a Priority Queue

- Binary Heaps
- HeapSort

Part II: Disjoint Set

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

"Tree" based structures…

Punctuated repetition…

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 valued priority

Ex: Scheduling

- Find next task to do
- Earliest deadline first

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

Abstract Data Type

Priority Queue

`interface IPriorityQueue<Key, Priority>`

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

Notes:

Assume data items are unique.

Abstract Data Type

Max Priority Queue

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

Notes:

Assume data items are unique.

Priority Queue

Sorted array

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

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

Priority Queue

Unsorted array

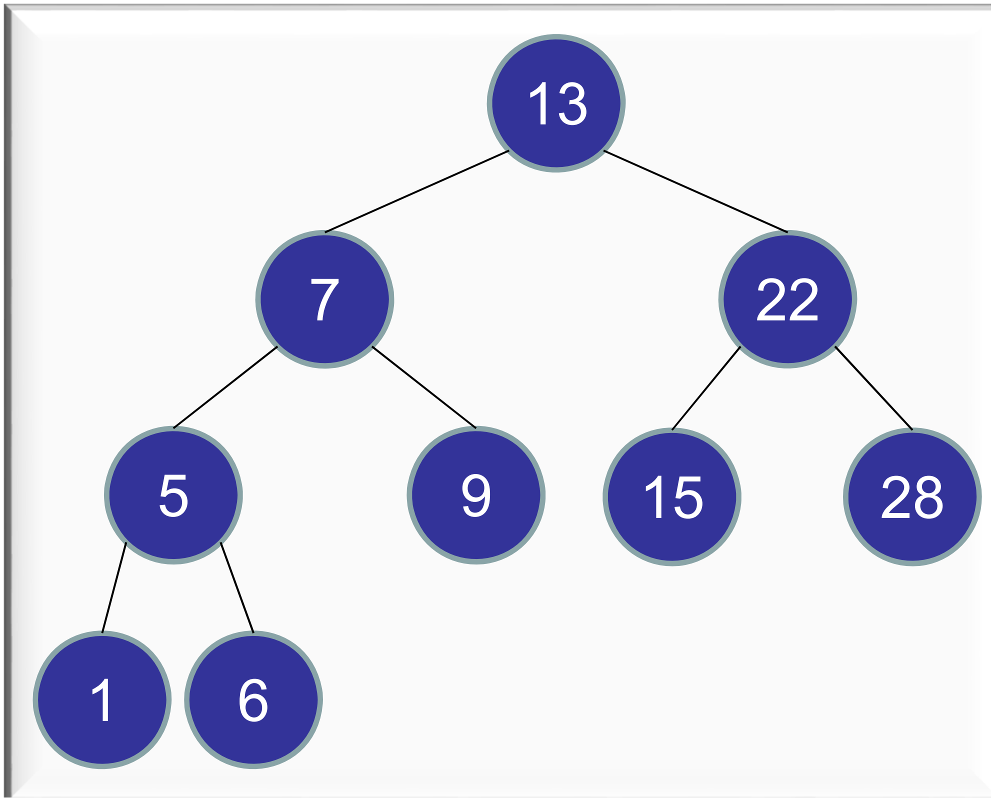
- insert: O(1)
  - Add object to end of list
- extractMax: O(n)
  - Search for largest element in array.
  - Remove and move everything over.

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

Priority Queue

AVL Tree (indexed by priority)

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



Priority Queue

Other operations:

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

Hash table:

- Maps priorities to array slots or nodes in tree.

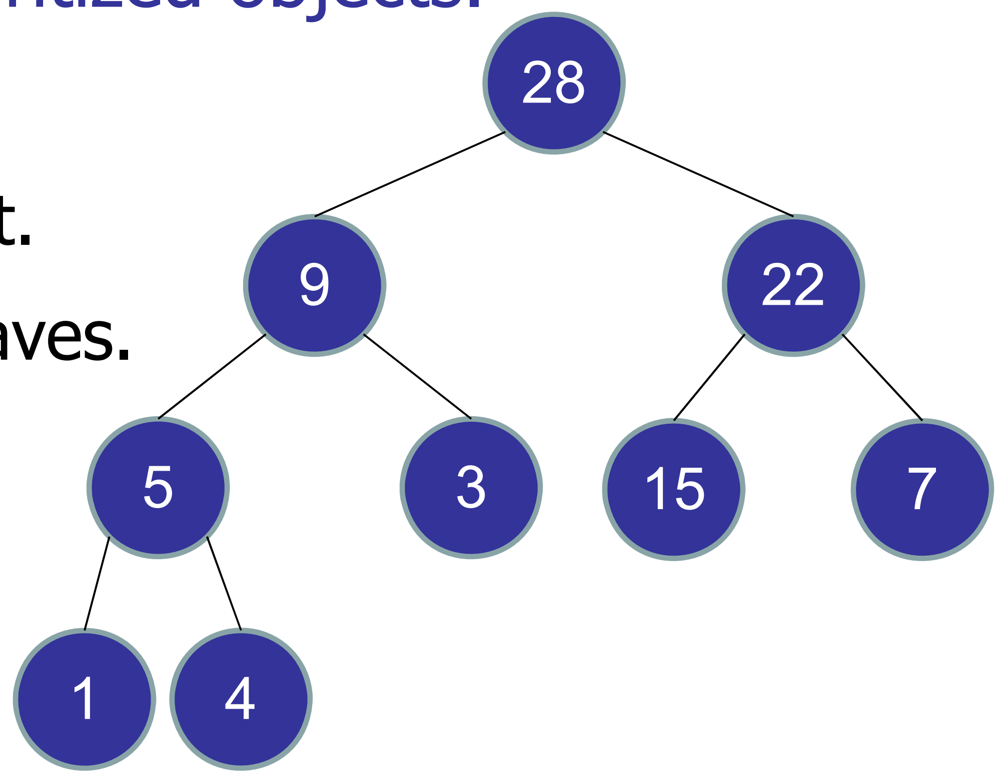
Dijkstra's Performance

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)

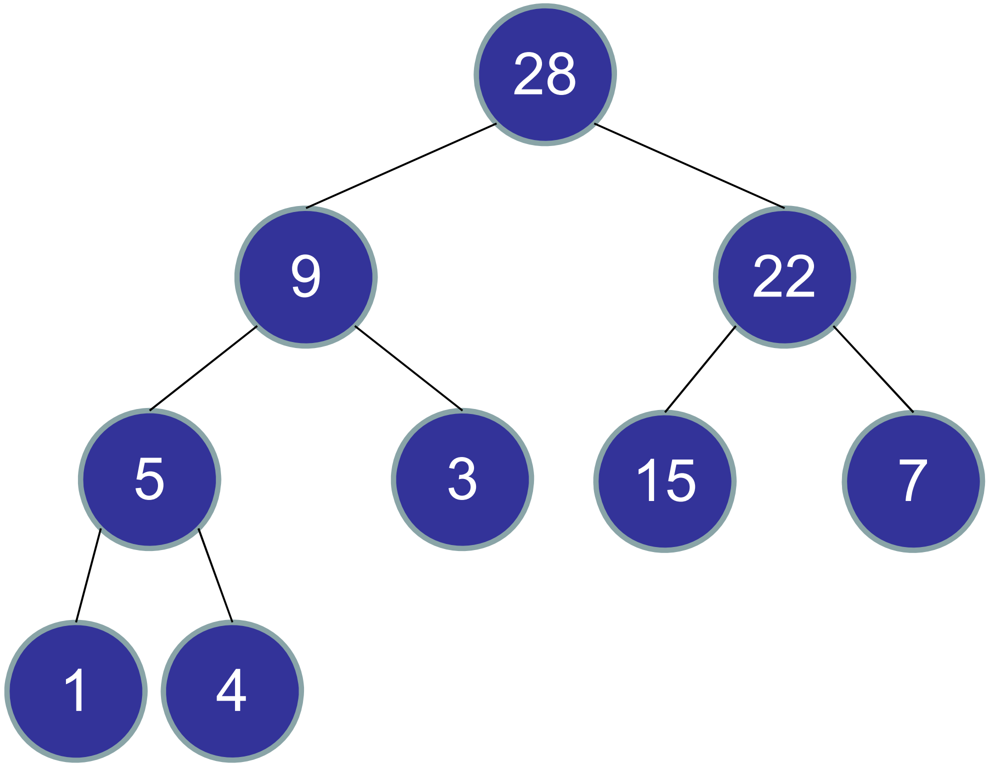
- Implements a Max Priority Queue
- Maintain a set of prioritized objects.
- Store items in a tree.
  - Biggest items at root.
  - Smallest items at leaves.



Two Properties of a Heap

1. Heap Ordering

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

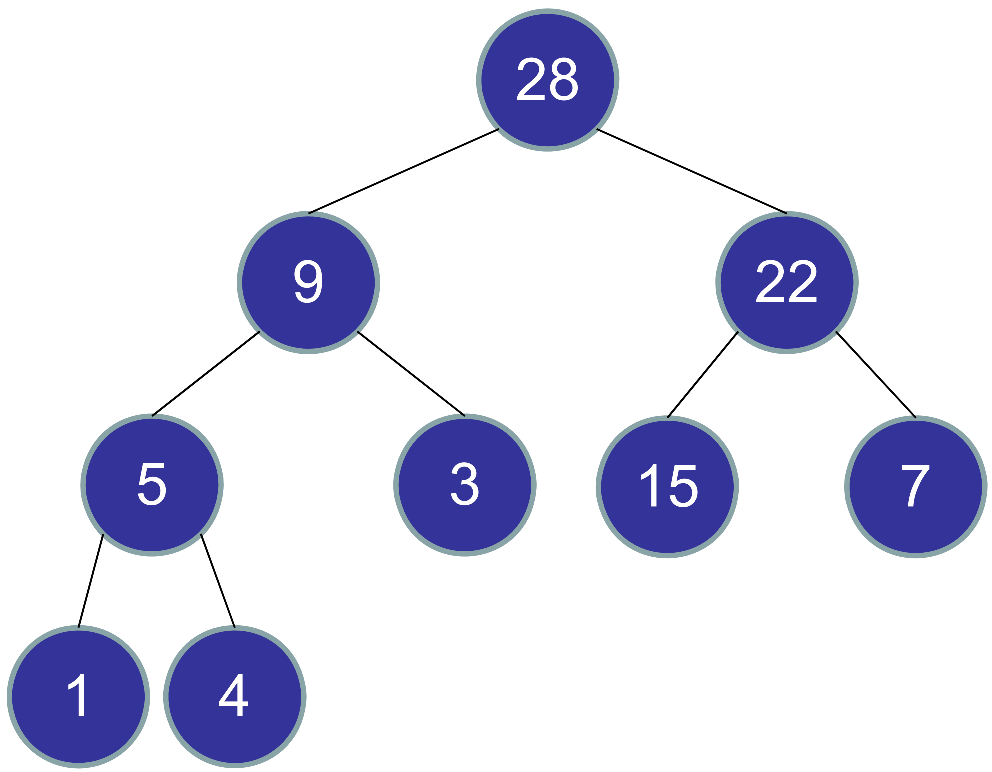


Note: not a binary search tree.

Two Properties of a Heap

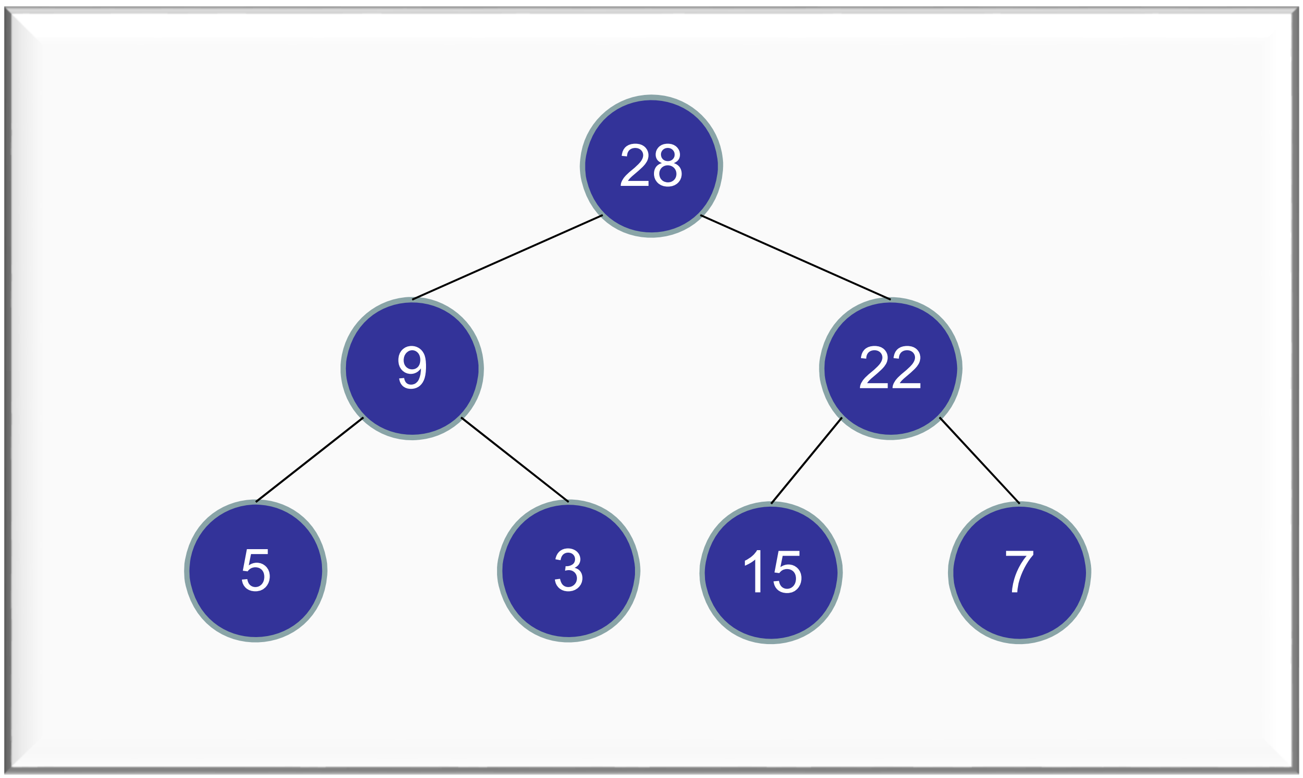
2. Complete binary tree

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



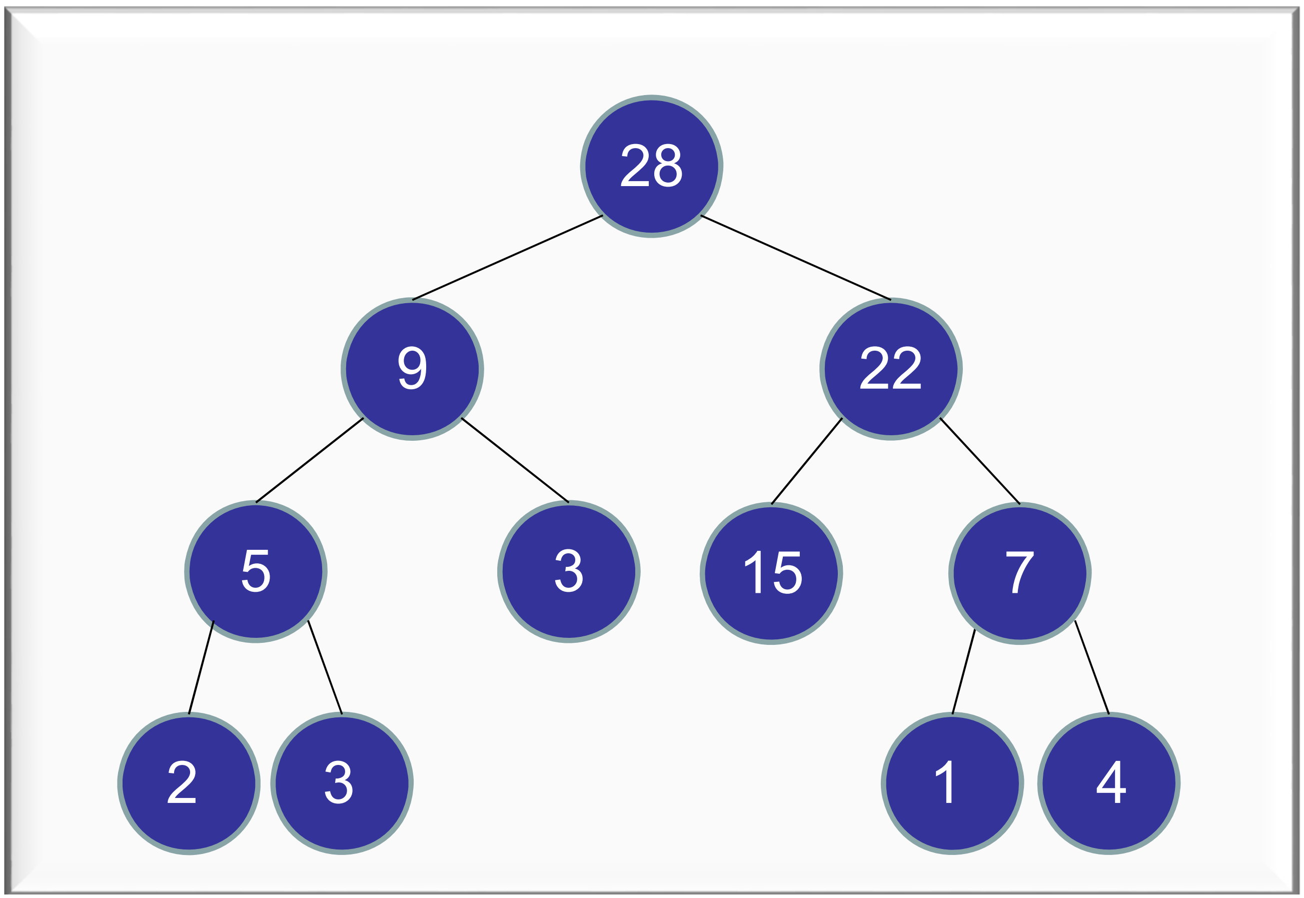
Is it a heap?

1. Yes
2. No.



Is it a heap?

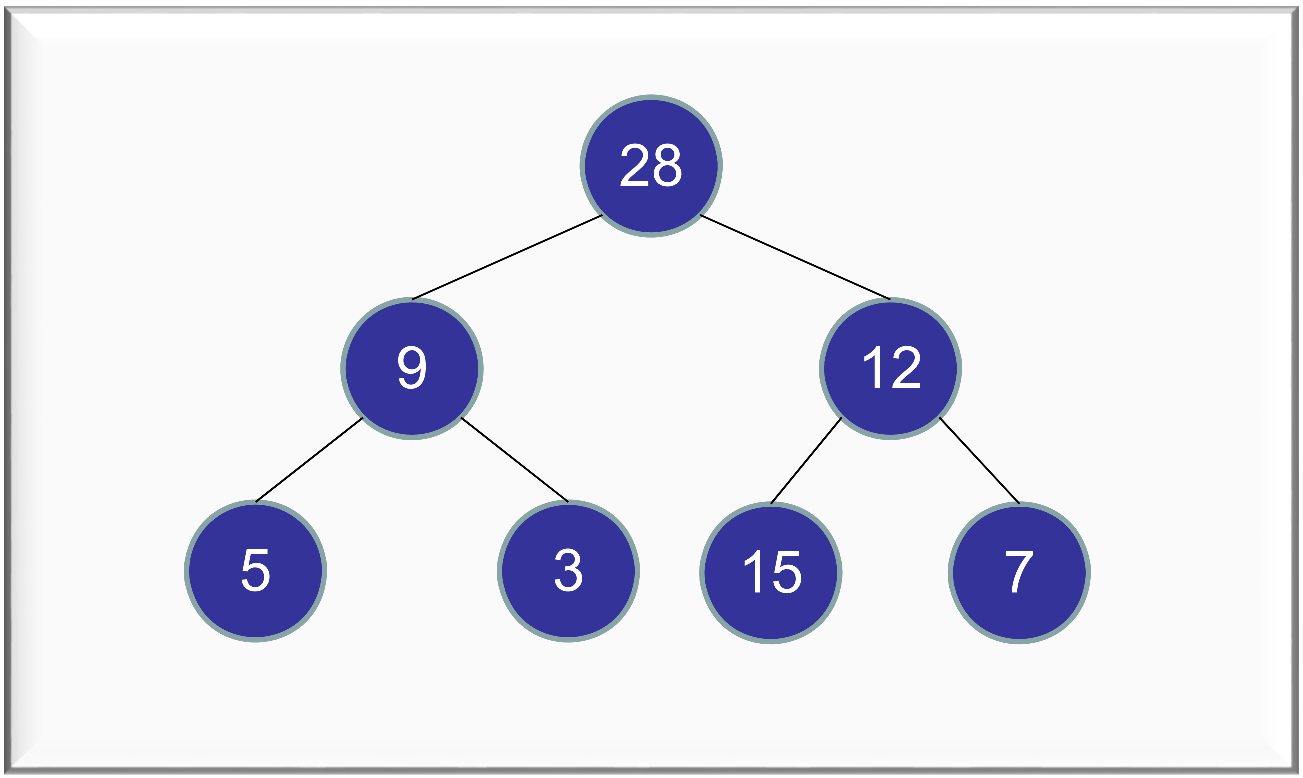
1. Yes
2. No.



Is it a heap?

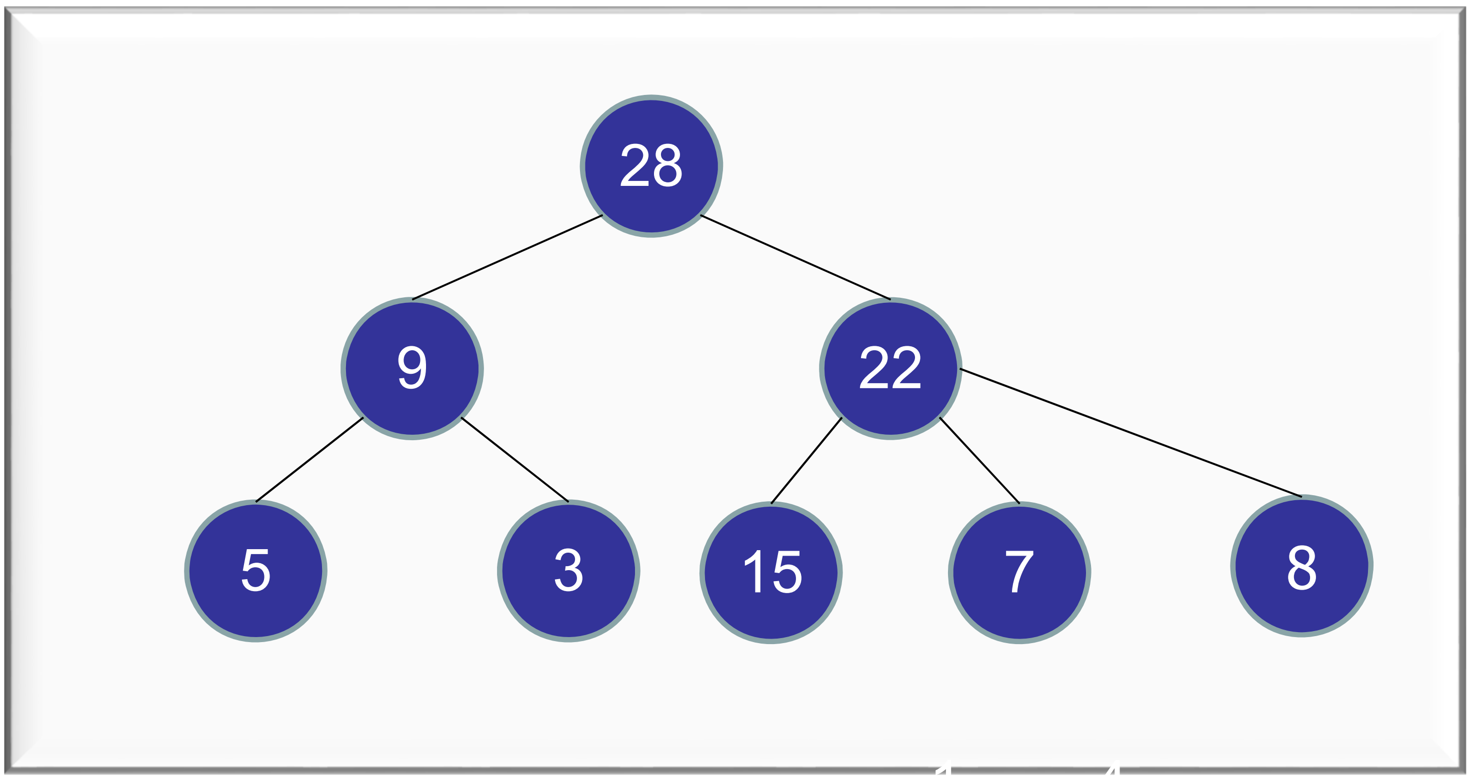
1. Yes
2. No.

Yes or No on Zoom.



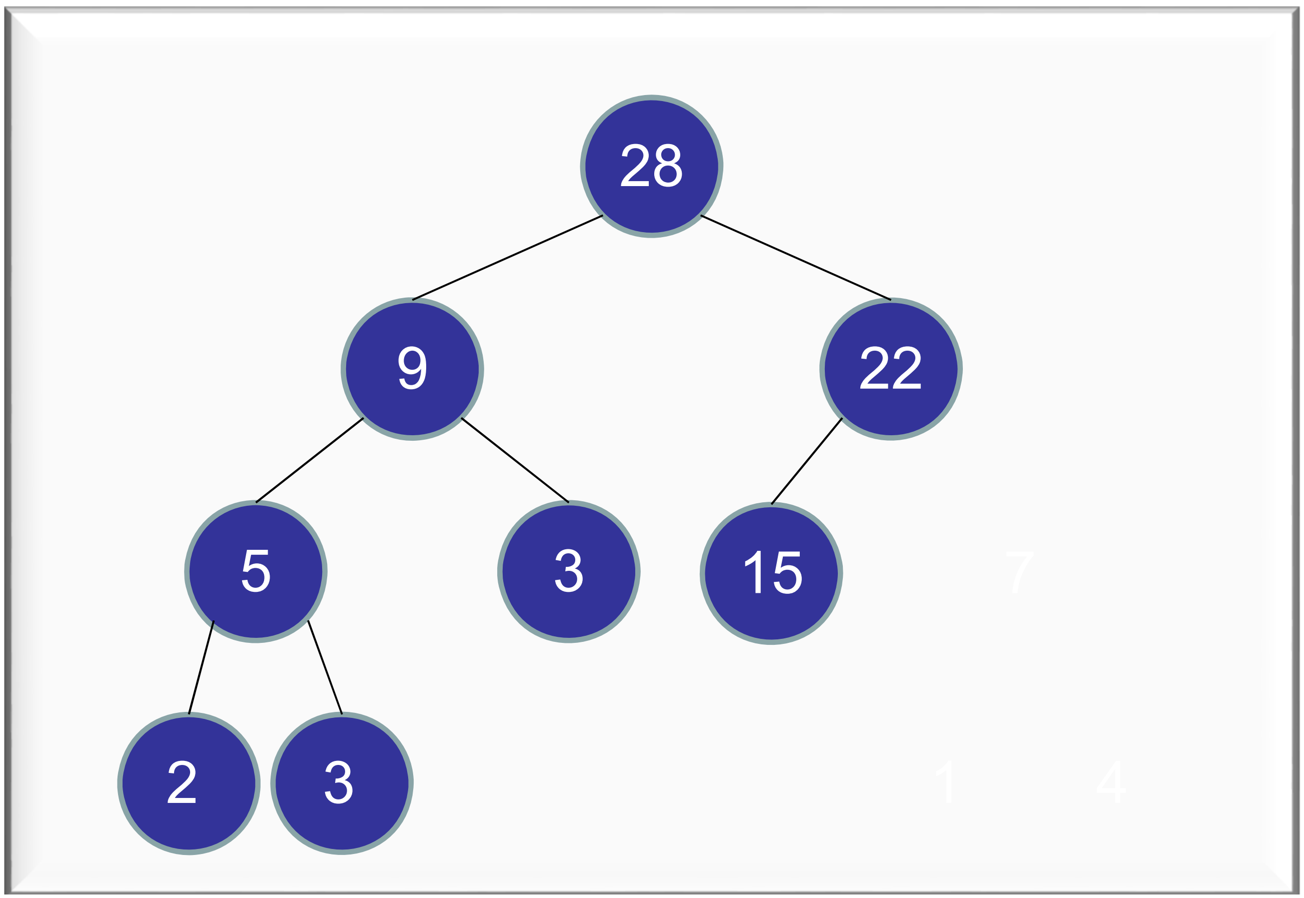
Is it a heap?

1. Yes
2. No.



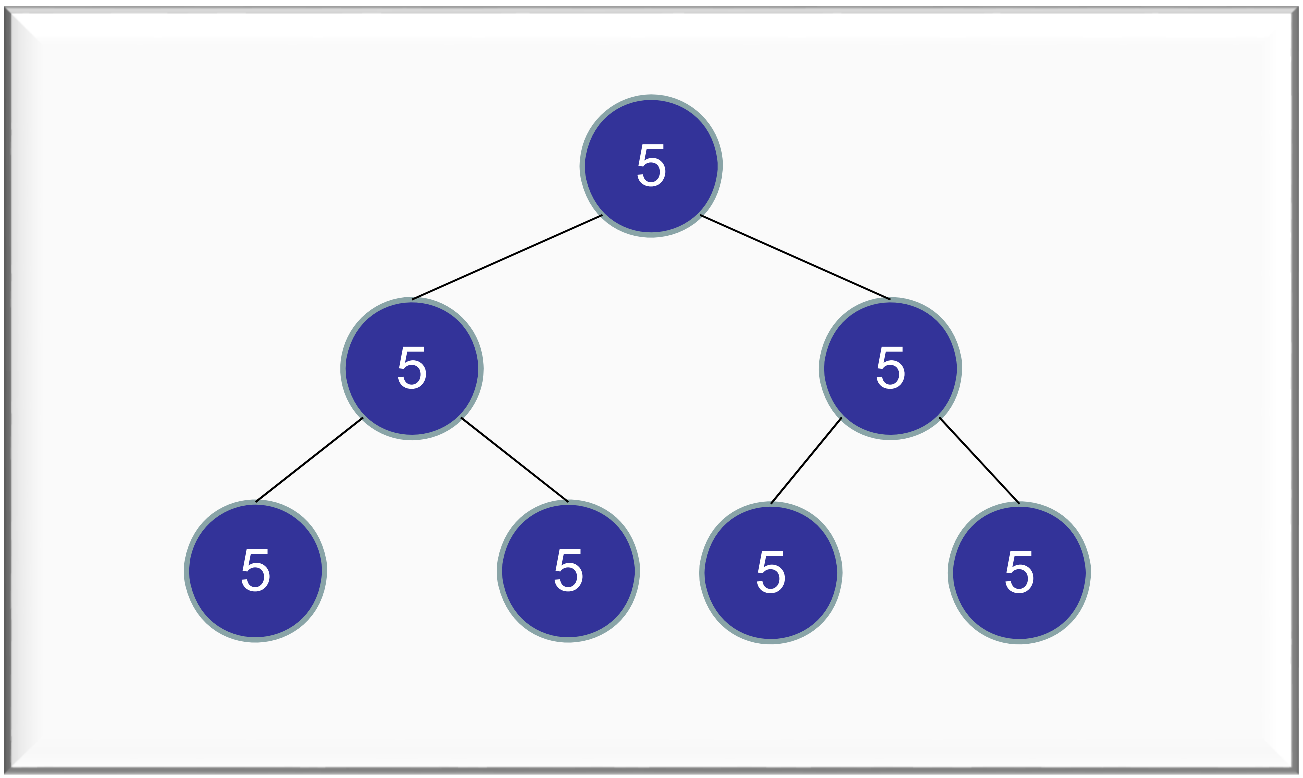
Is it a heap?

1. Yes
2. No.



Is it a heap?

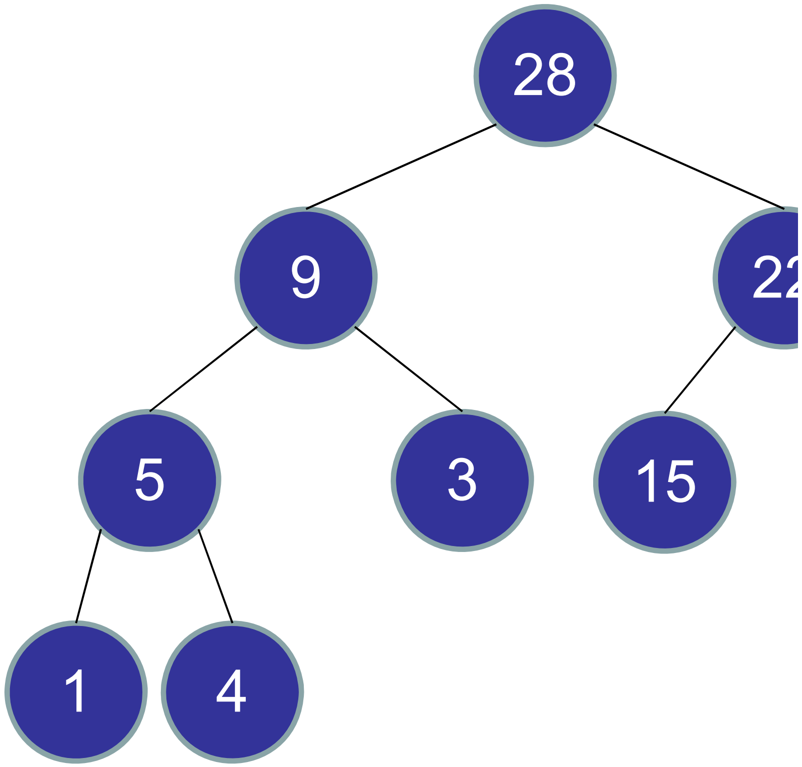
1. Yes
2. No.



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

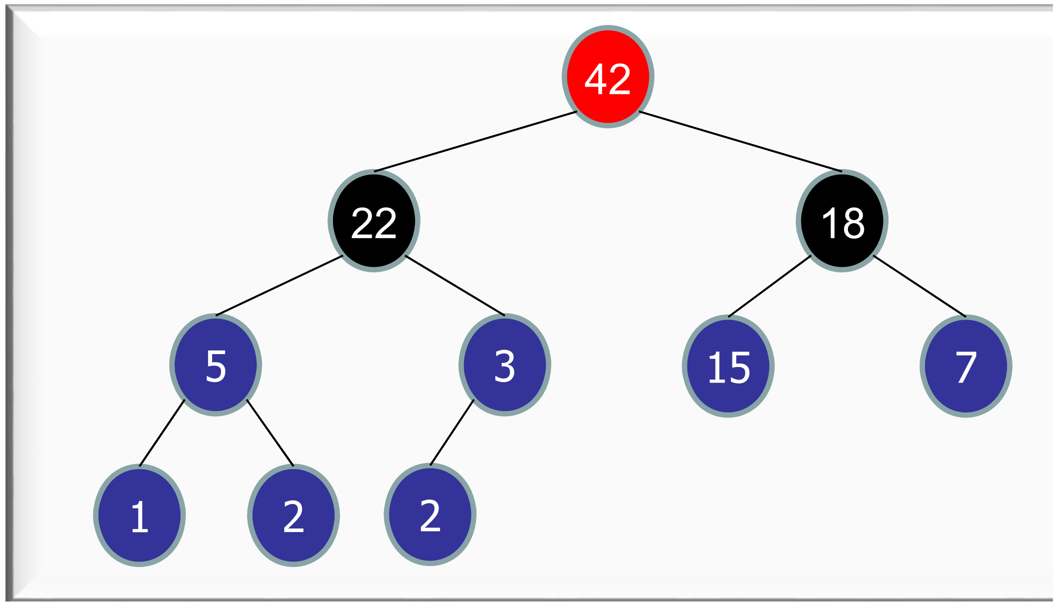


What is the maximum height of a heap with n elements?

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

ARCHIPELAGO

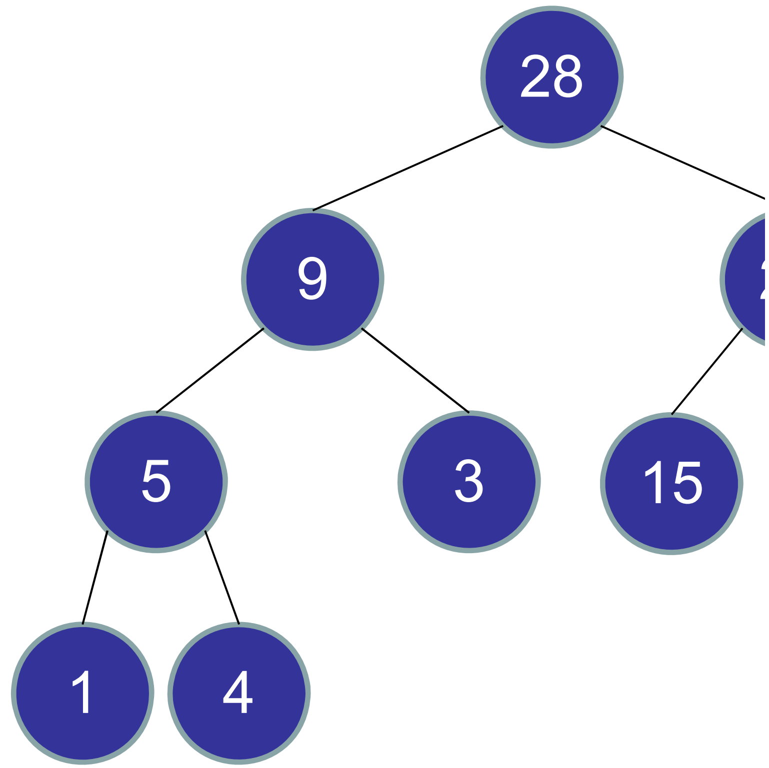
is open



Heap

(aka Binary Heap or MaxHeap)

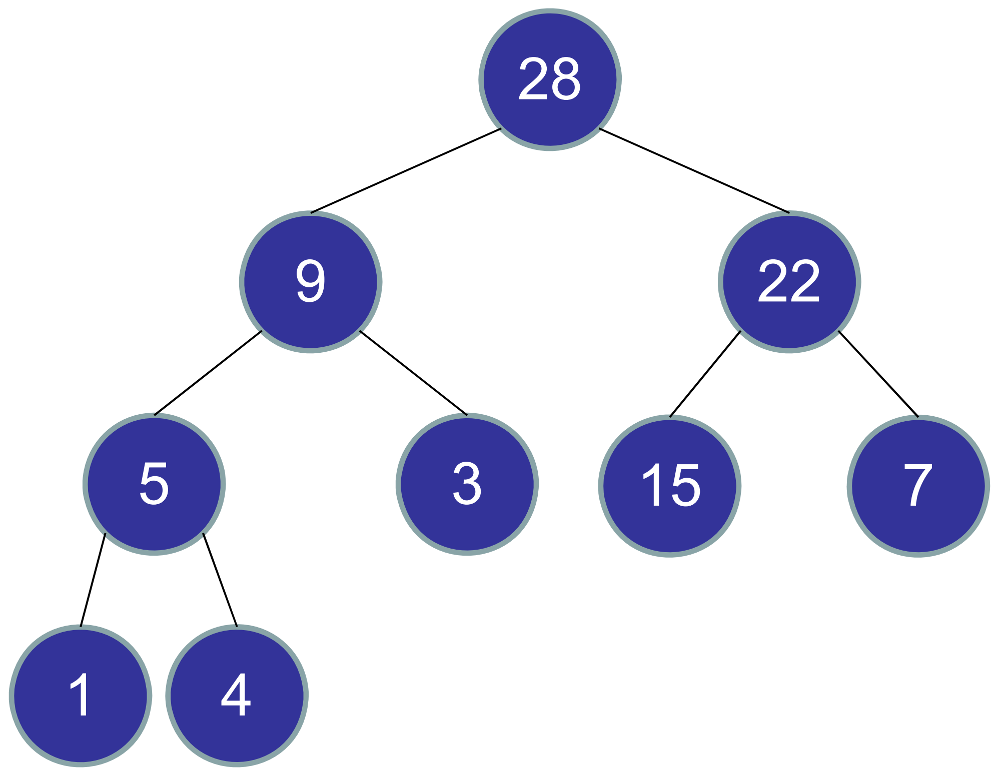
- Implements a Max Priority Queue
- Maintain a set of prioritized objects.
- Store items in a tree.
  - Biggest items at root.
  - Smallest items at leaves.
- Two properties:
  1. Heap Ordering
  2. Complete Binary Tree
- Height: O(log n)



Heap

Priority Queue Operations

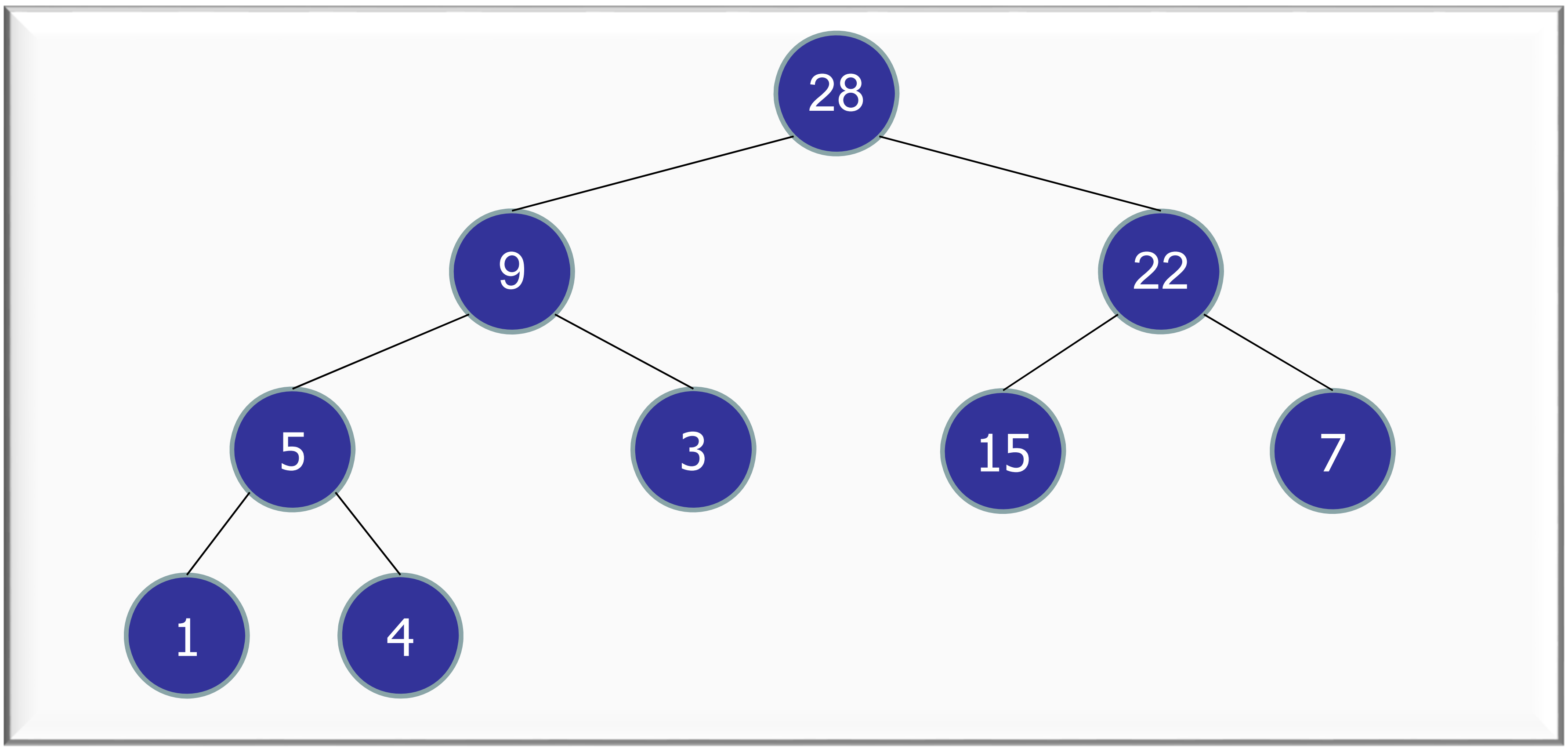
- insert
- extractMax
- increaseKey
- decreaseKey
- delete



Heap Operations

`insert(25)`:

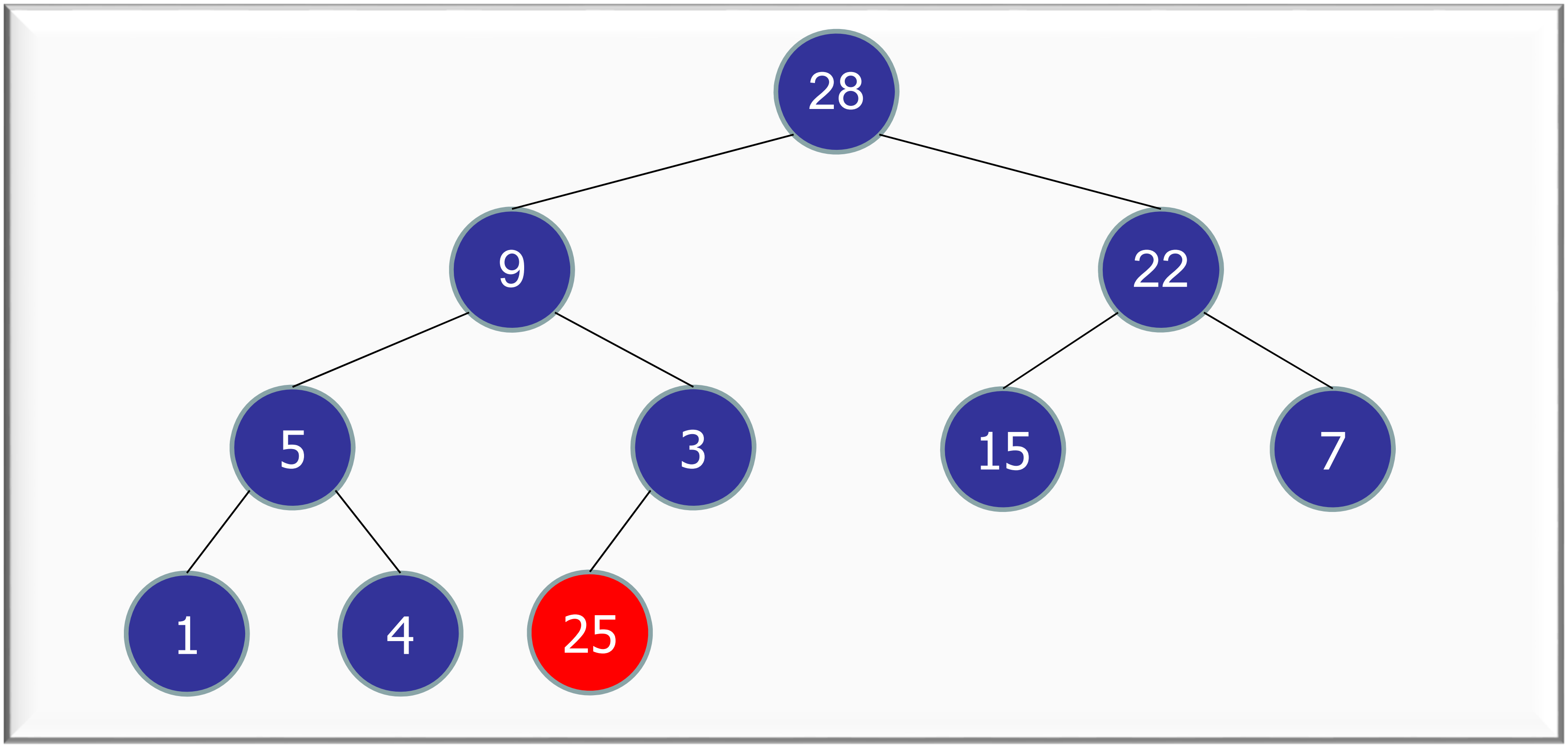
– Step one: add a new leaf with priority 25.



Heap Operations

`insert(25)`:

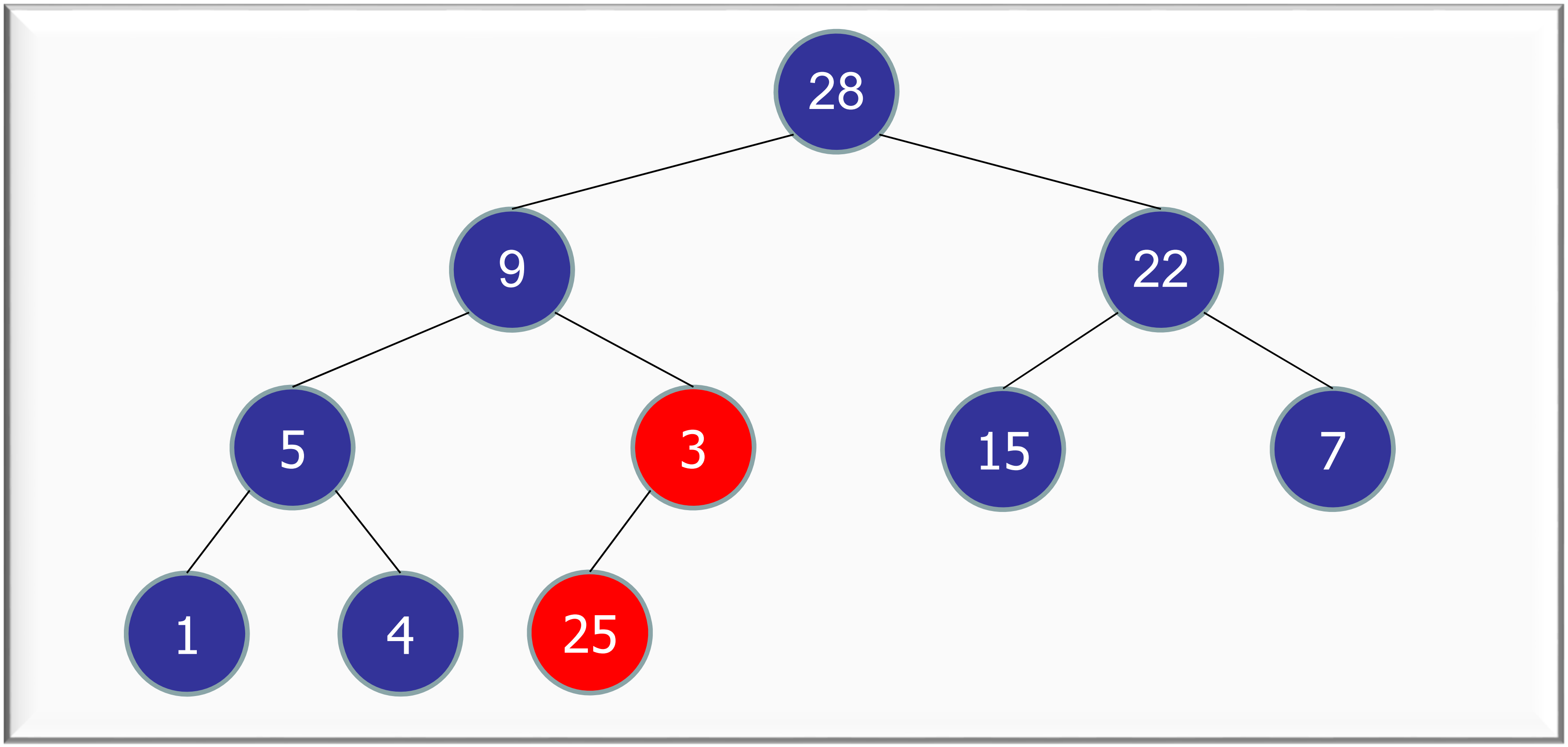
- Step one: add a new leaf with priority 25.



Heap Operations

`insert(25)`:

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

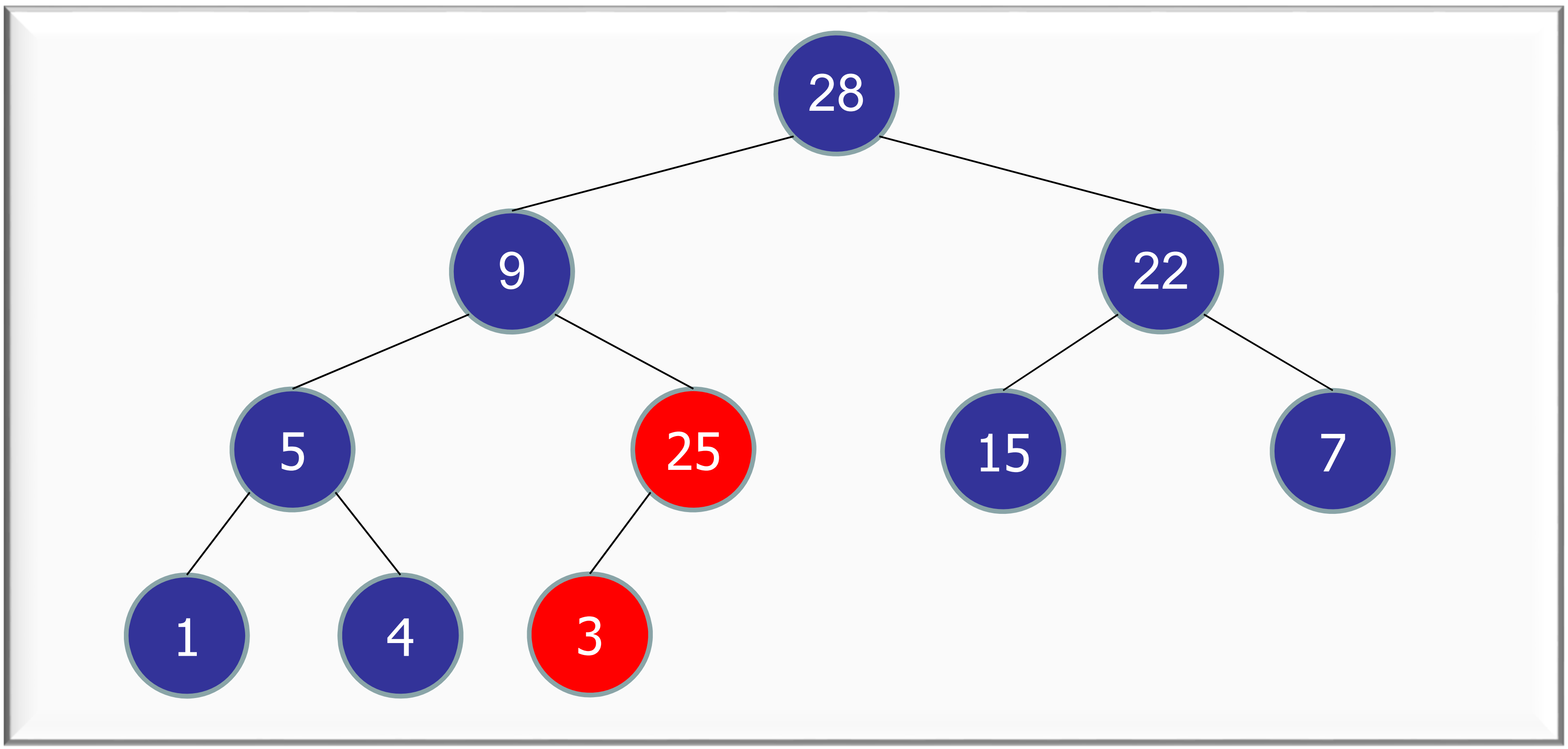


Heap Operations

`insert(25)`:

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

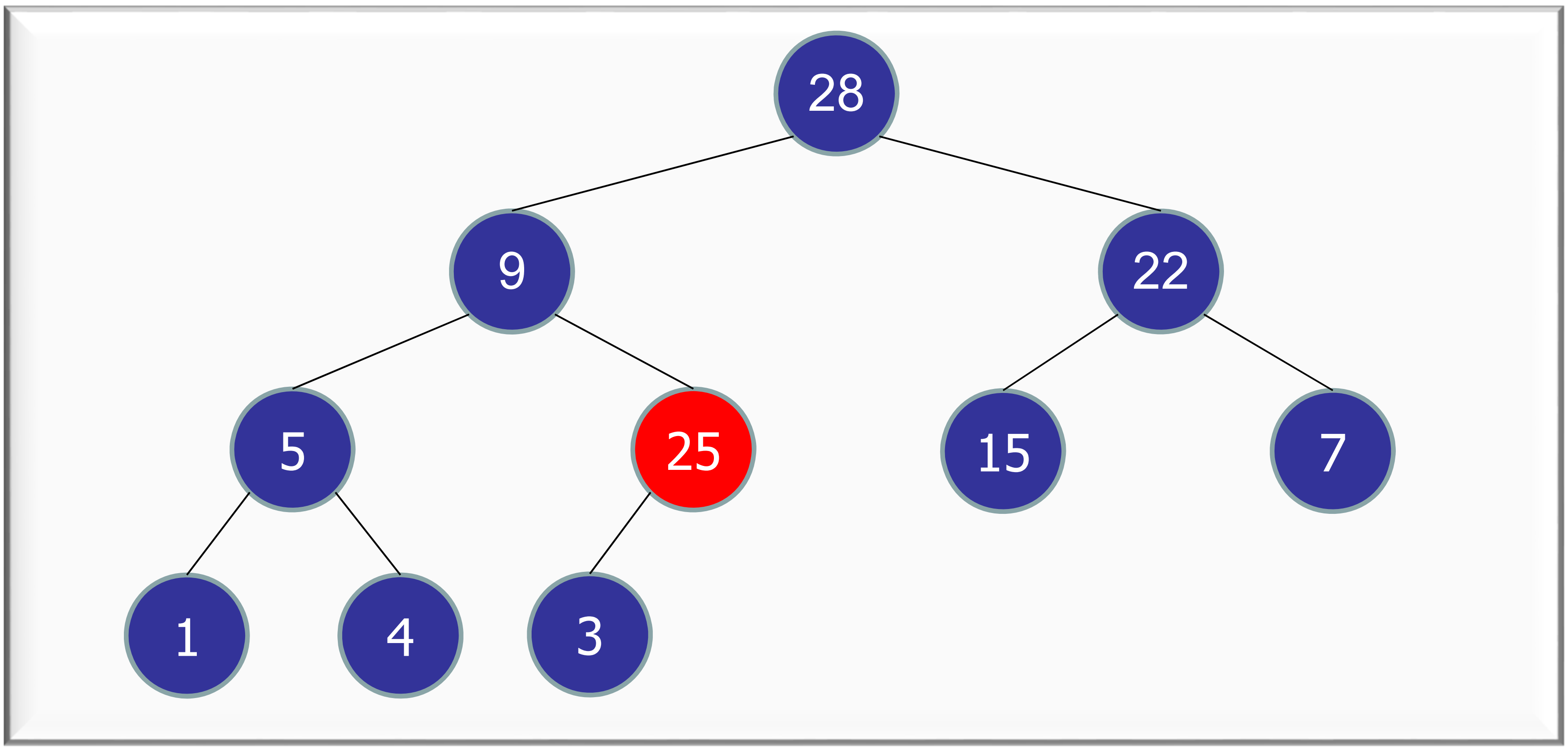
– Step two: bubble up



Heap Operations

`insert(25)`:

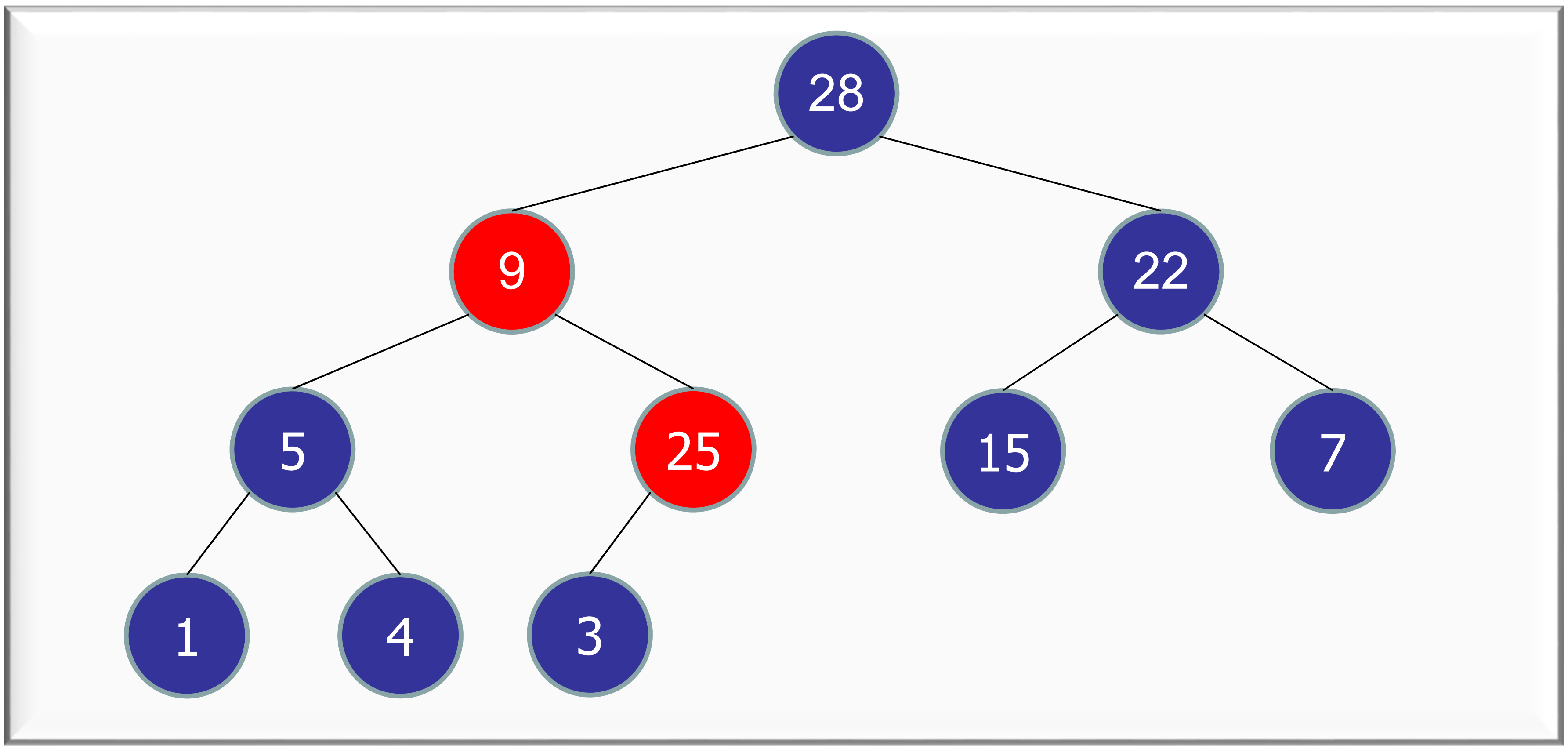
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Heap Operations

`insert(25)`:

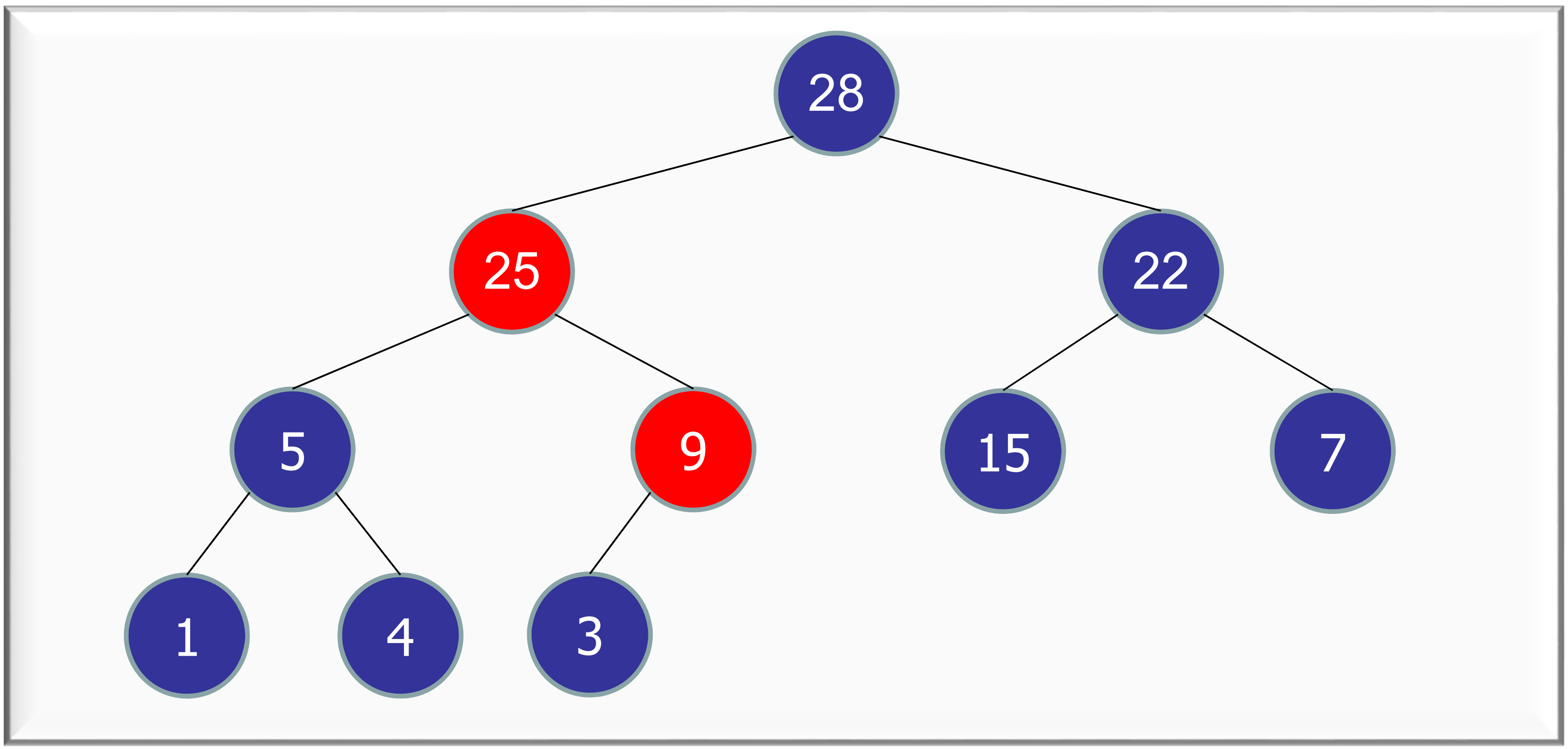
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Heap Operations

`insert(25)`:

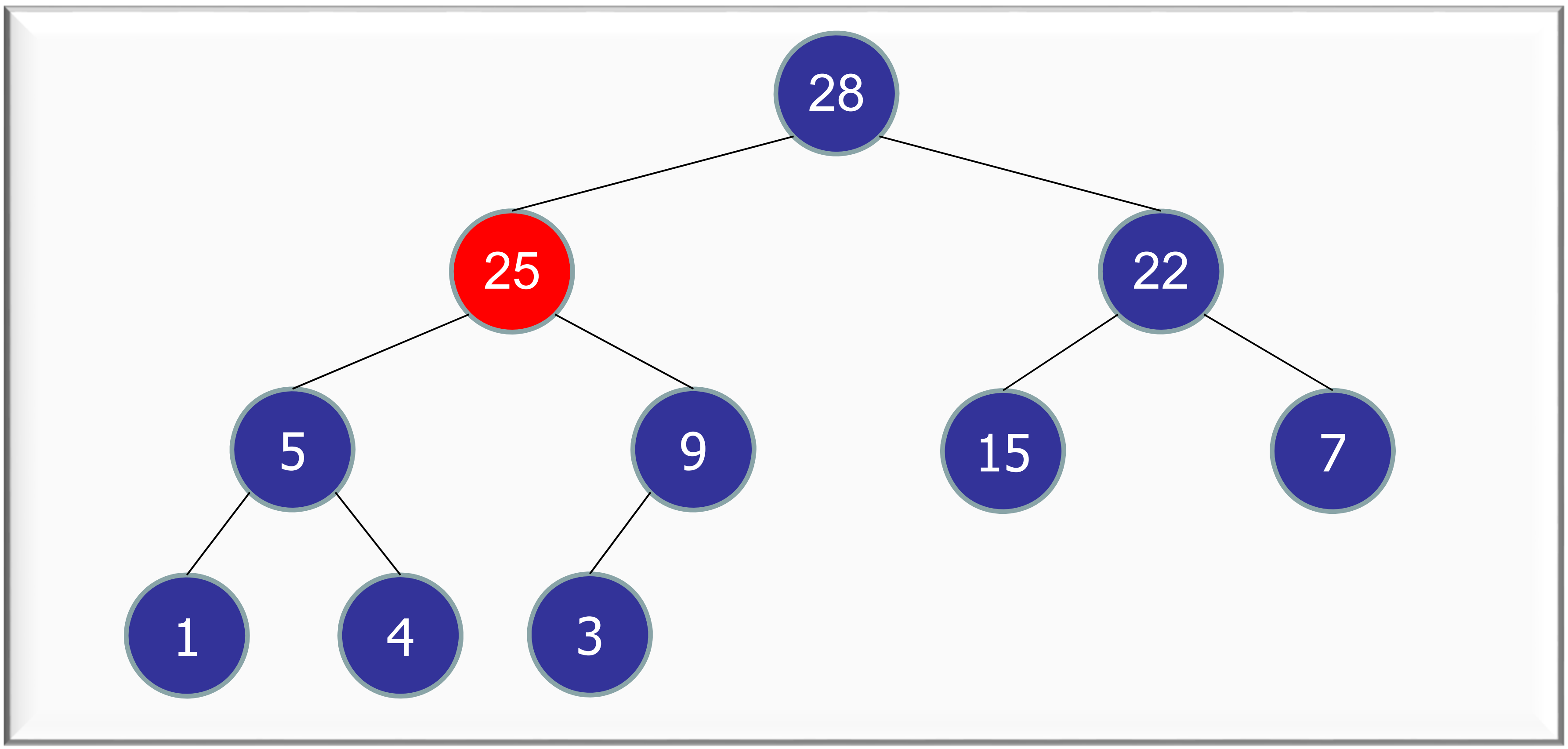
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Heap Operations

`insert(25)`:

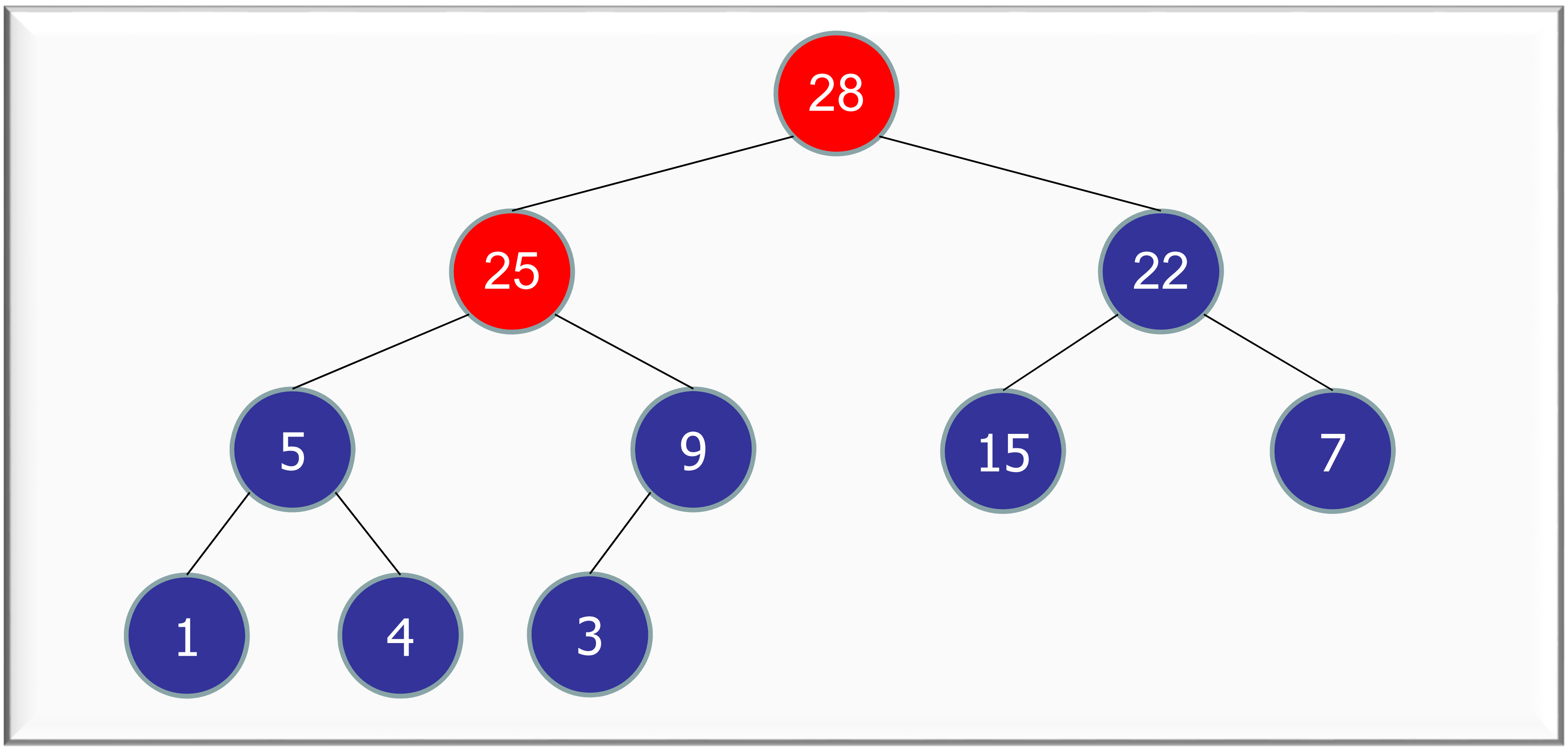
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Heap Operations

`insert(25)`:

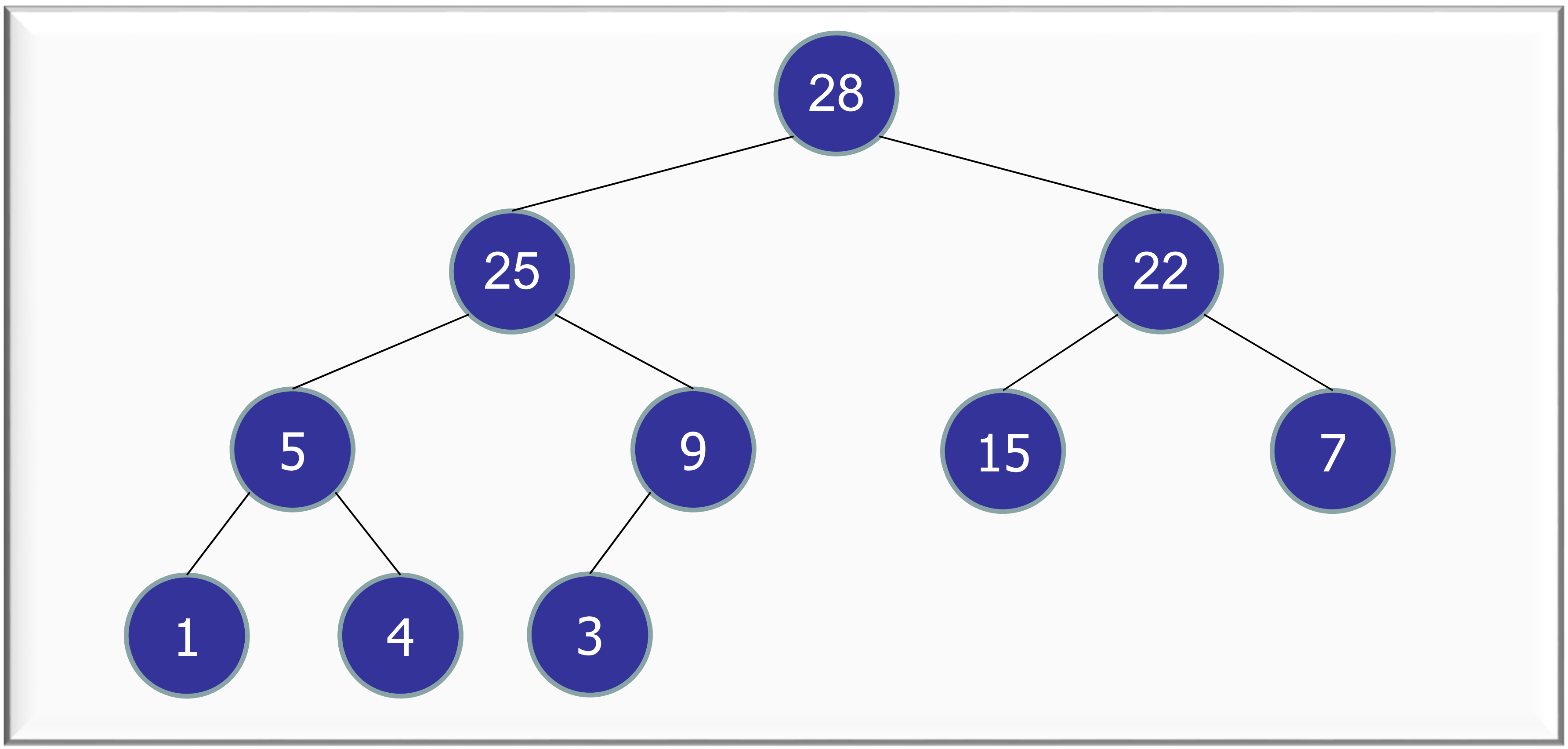
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Heap Operations

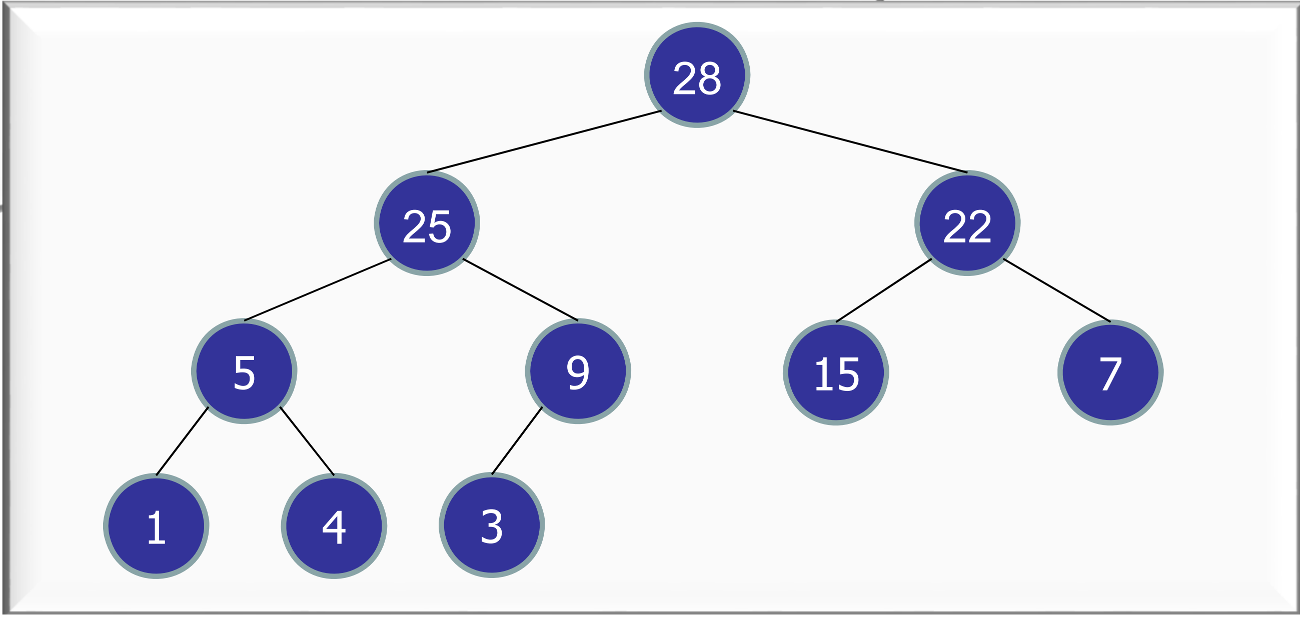
`insert(25)`:

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



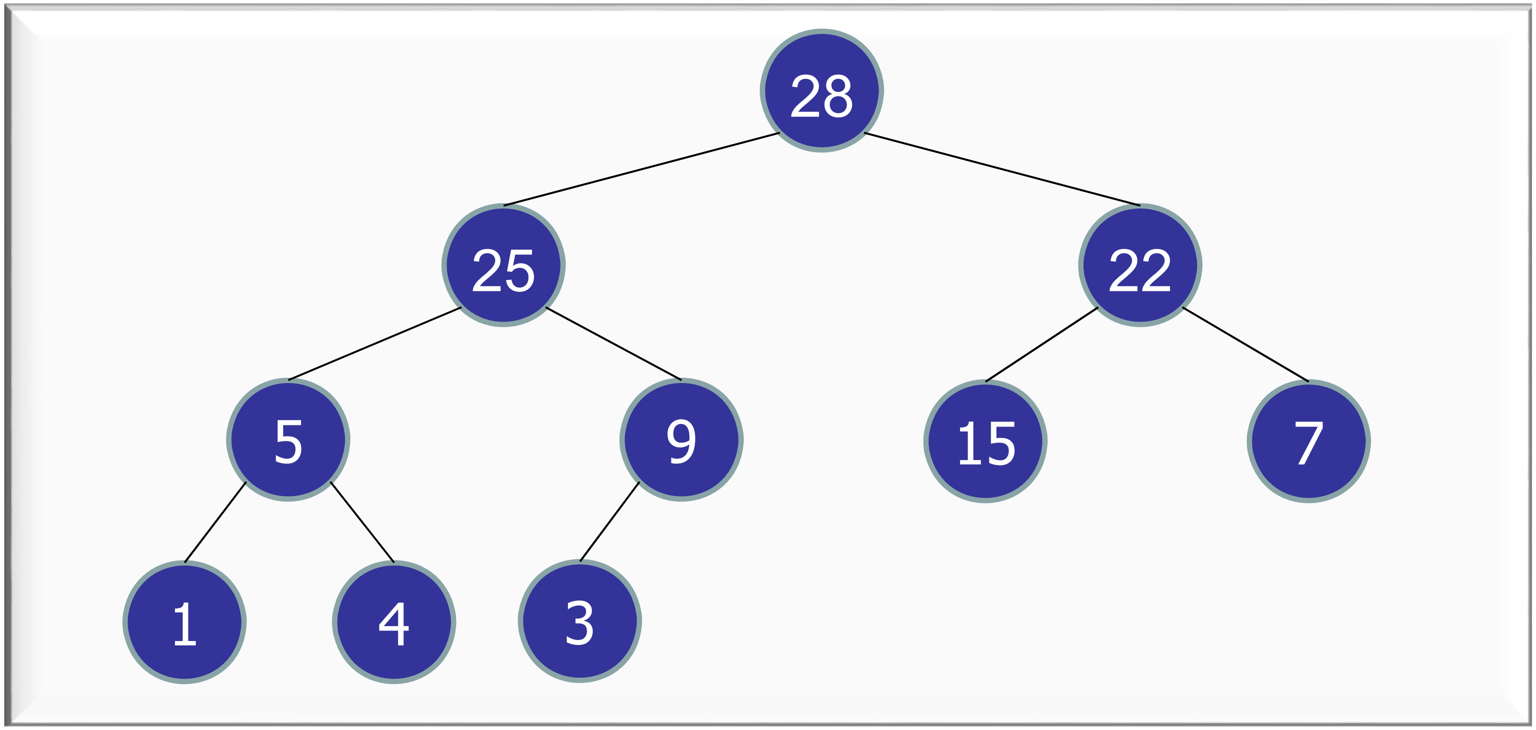
Heap Operations

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



Heap Operations

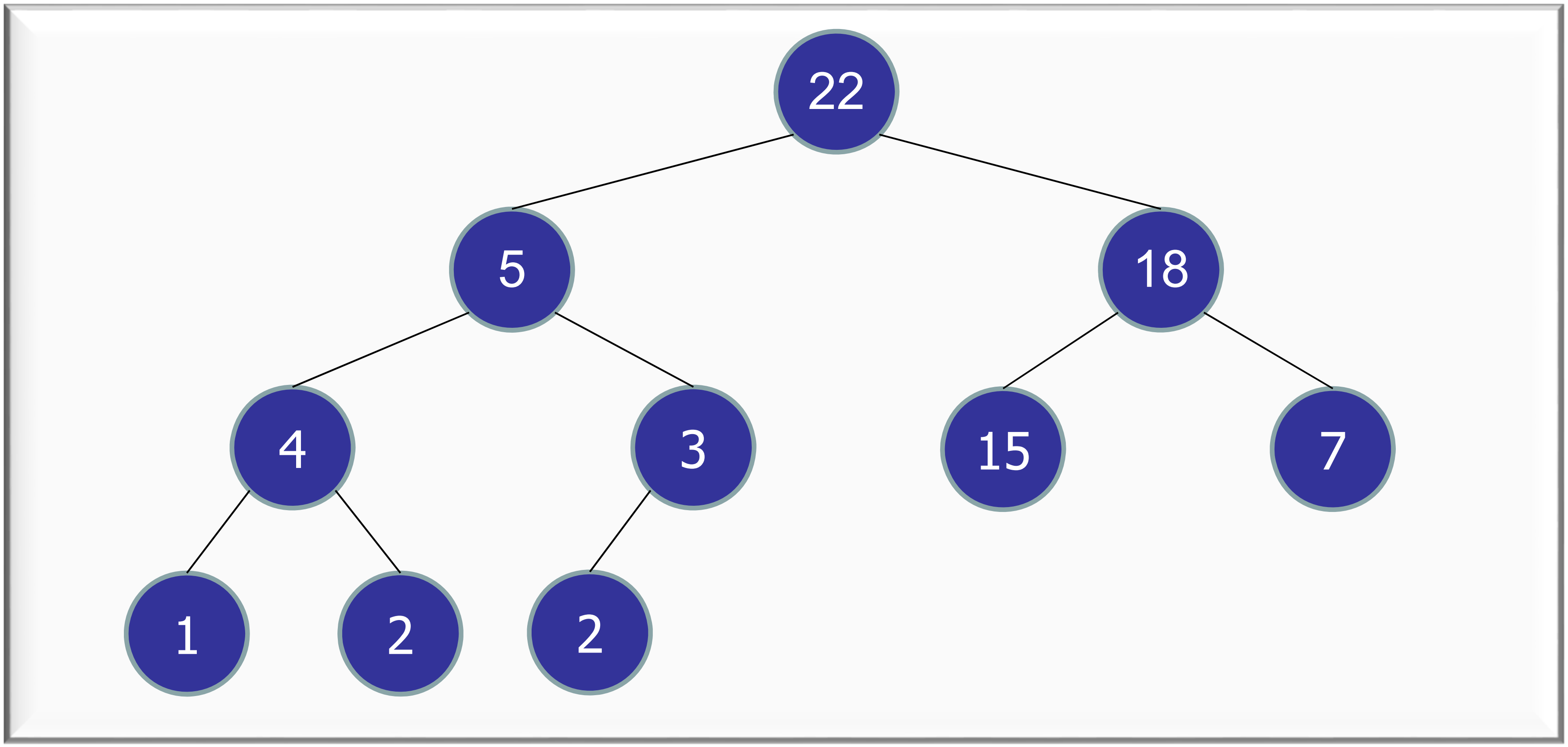
```
insert(Priority p, Key k) {
   Node v = completeTree.insert(p,k);
   bubbleUp(v);
}
```



Heap Operations

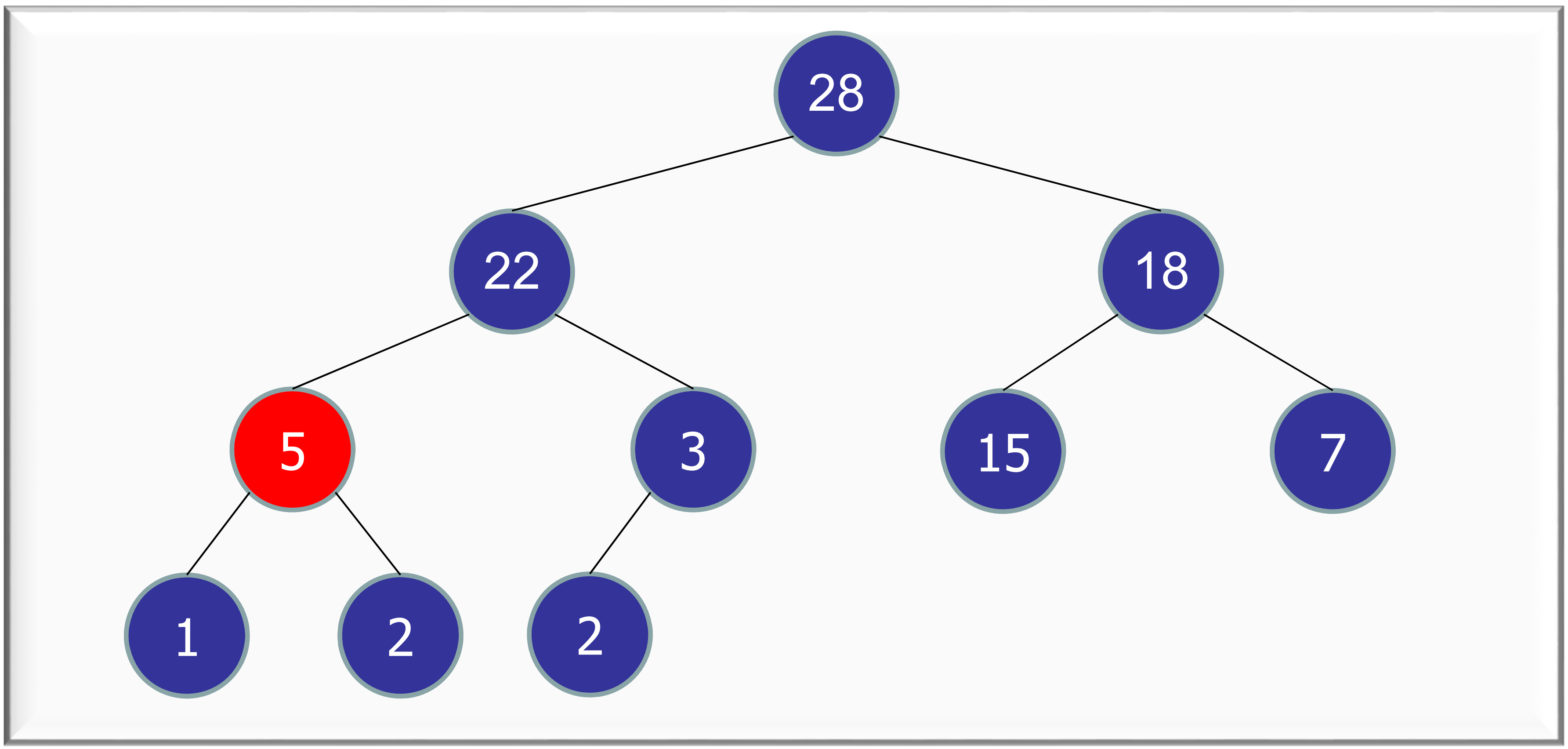
`insert(...)` :

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



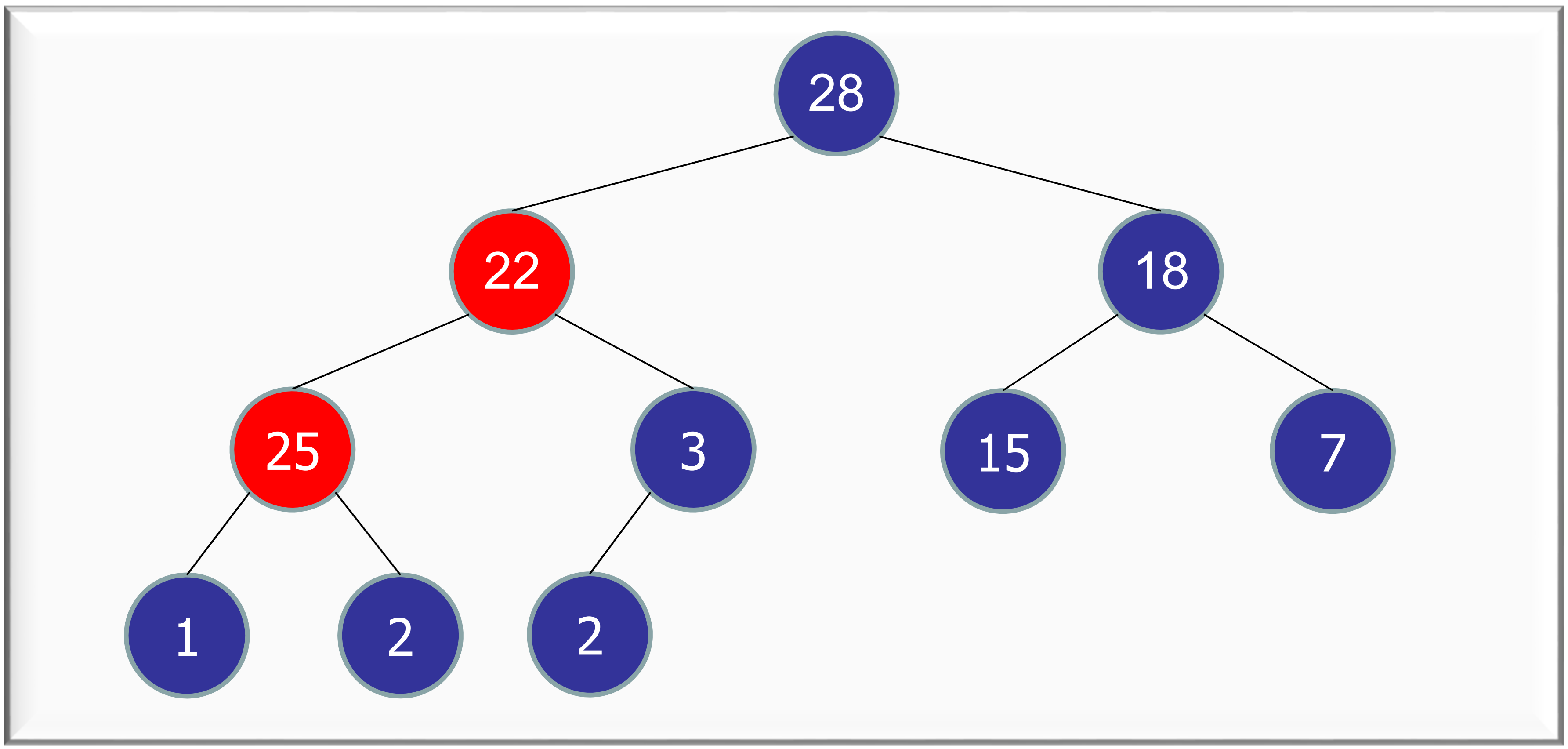
Heap Operations

`increaseKey(5 → 25):`



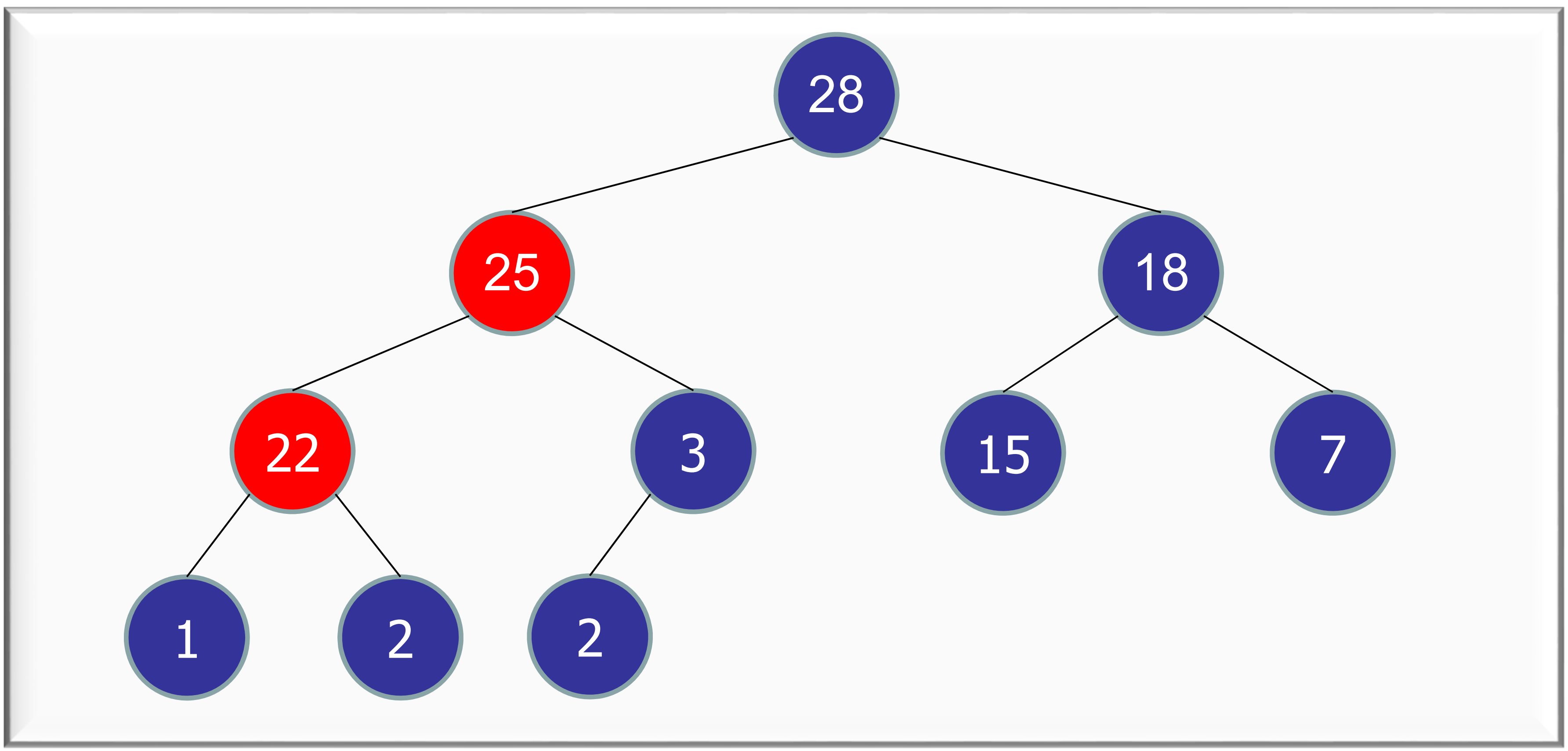
Heap Operations

`increaseKey(5 → 25)`: bubbleUp(25)



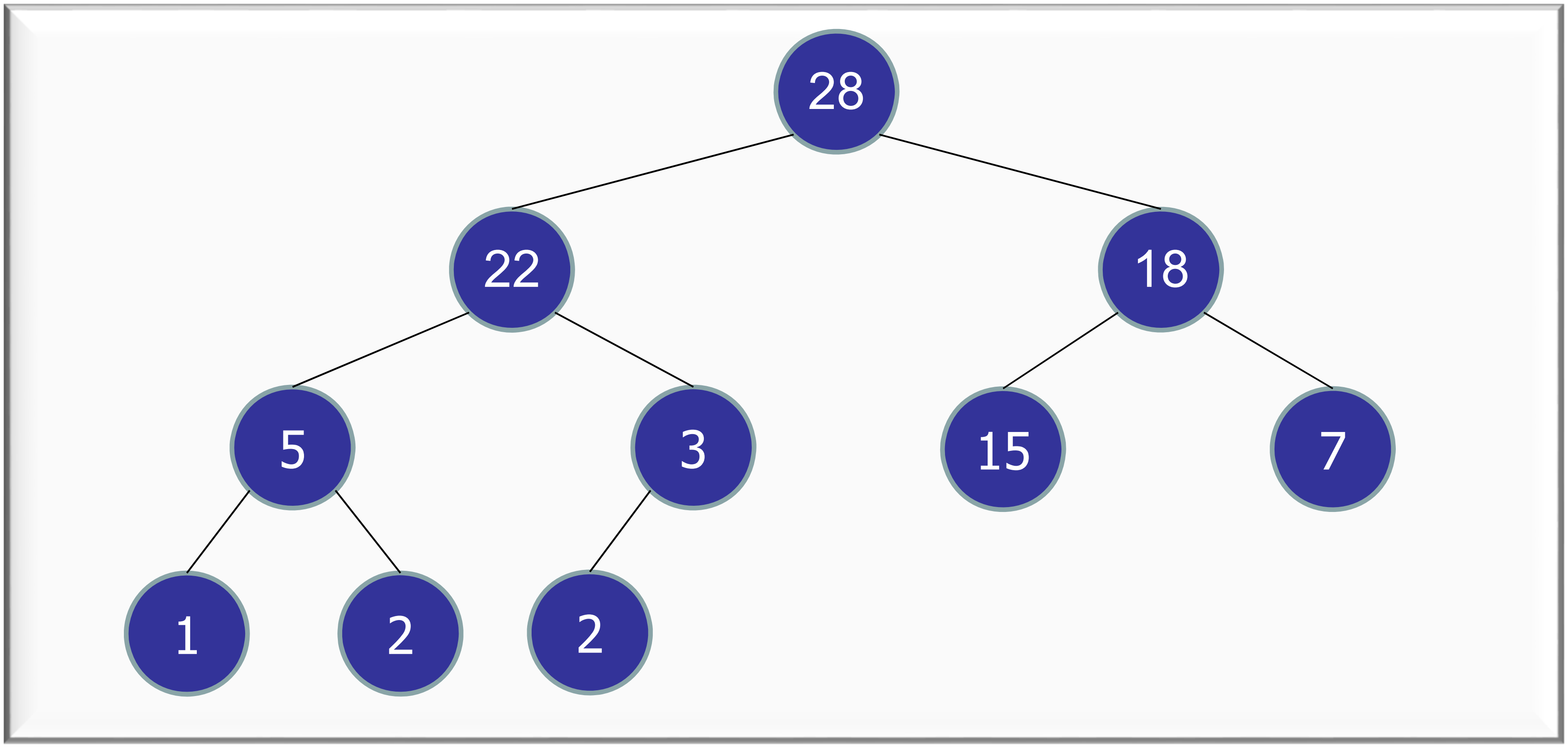
Heap Operations

`increaseKey(5 → 25)`: bubbleUp(25)



Heap Operations

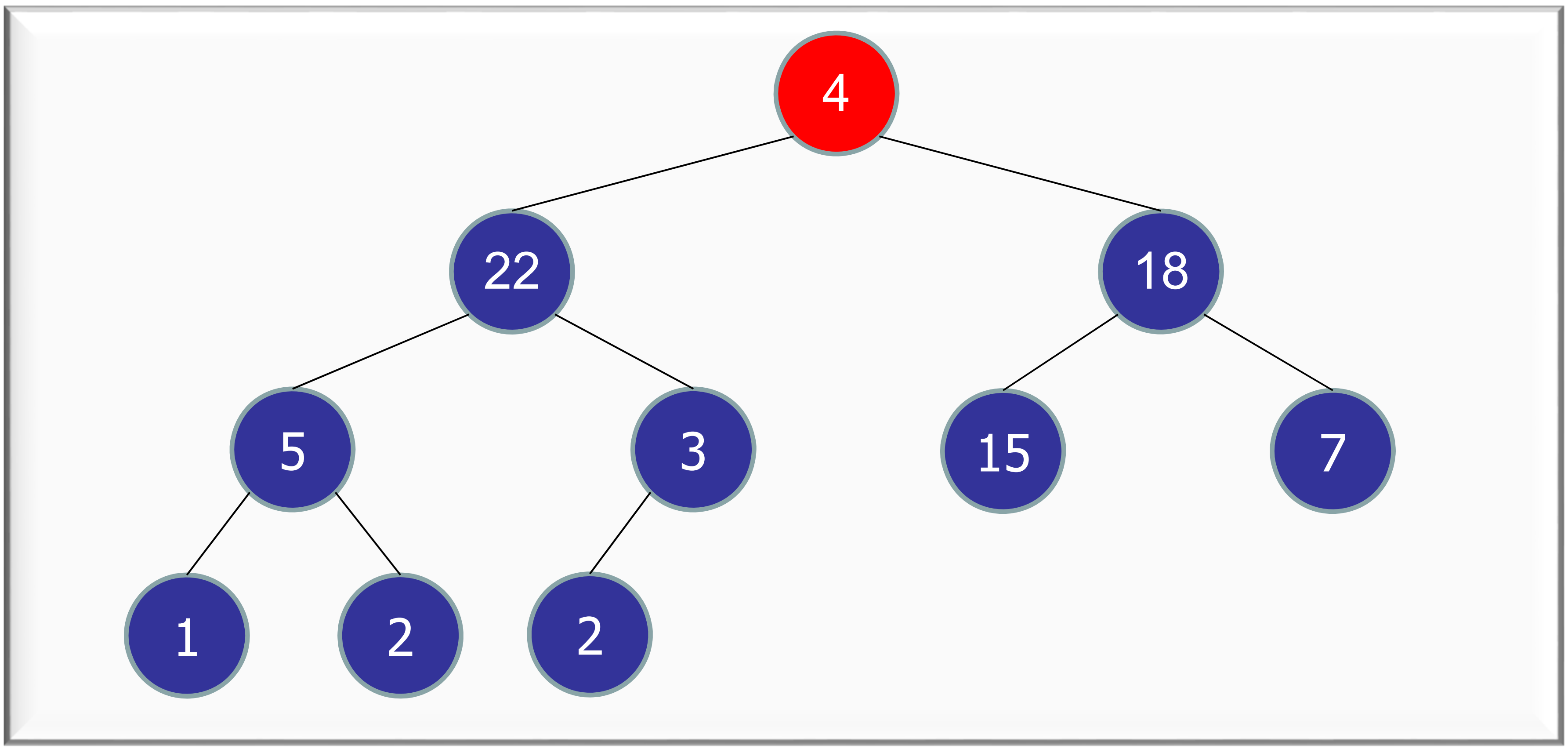
`decreaseKey(28 → 4)`:



Heap Operations

`decreaseKey(28 → 4)`:

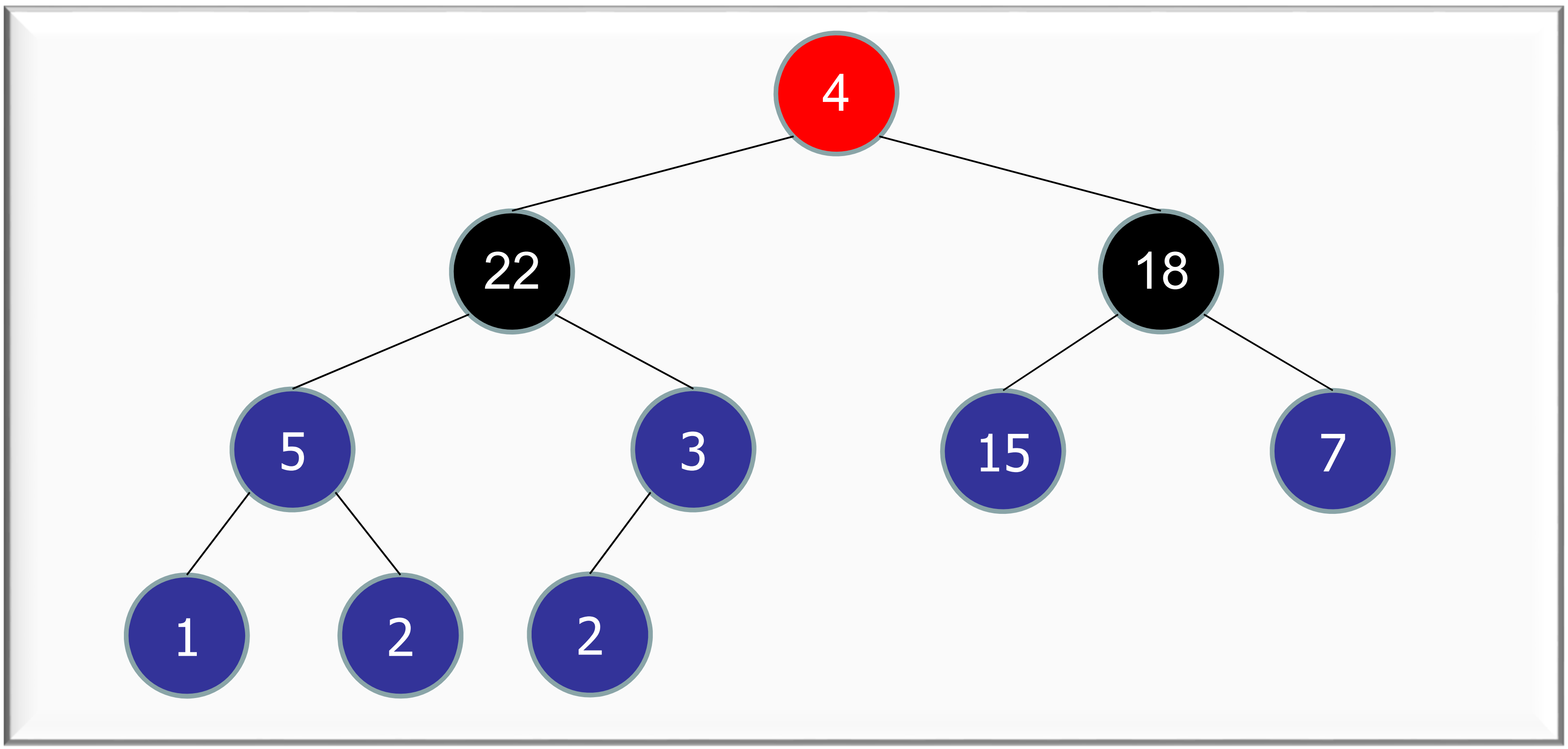
– Step 1: Update the priority



Heap Operations

`decreaseKey(28 → 4)`:

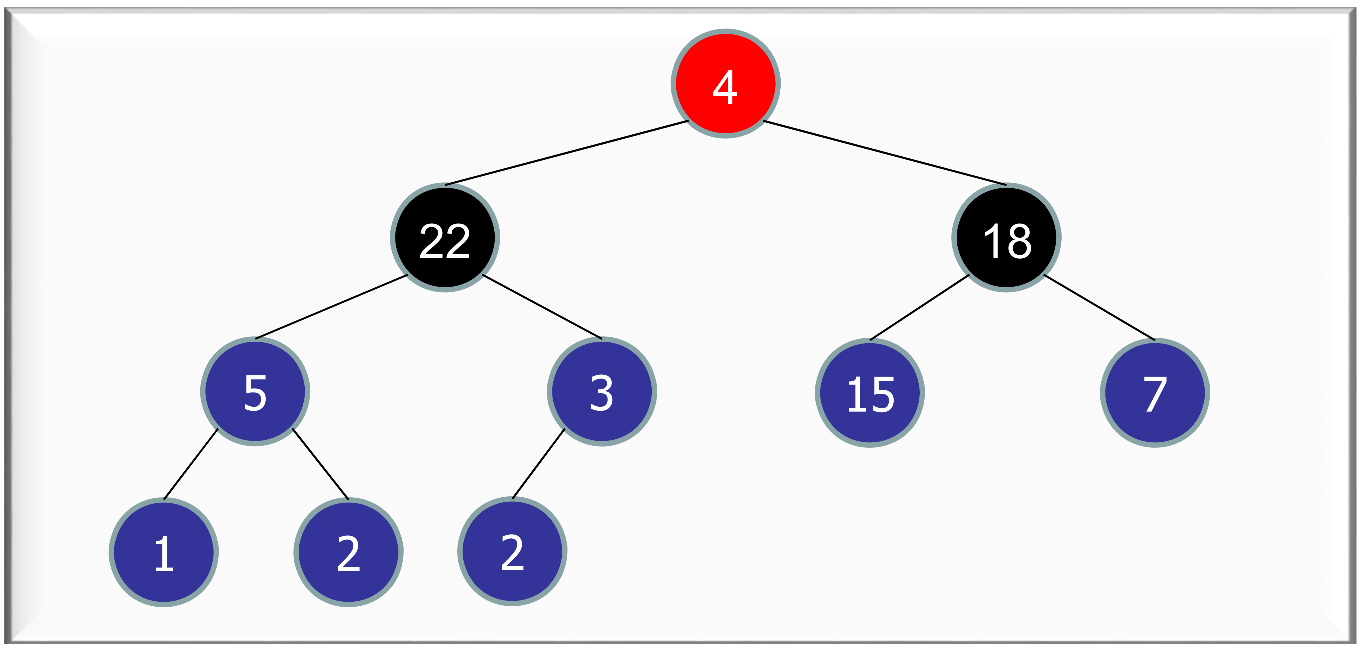
- Step 1: Update the priority
- Step 2: bubbleDown(4)



Which way to bubbleDown?

1. Left
2. Right
3. Does not matter

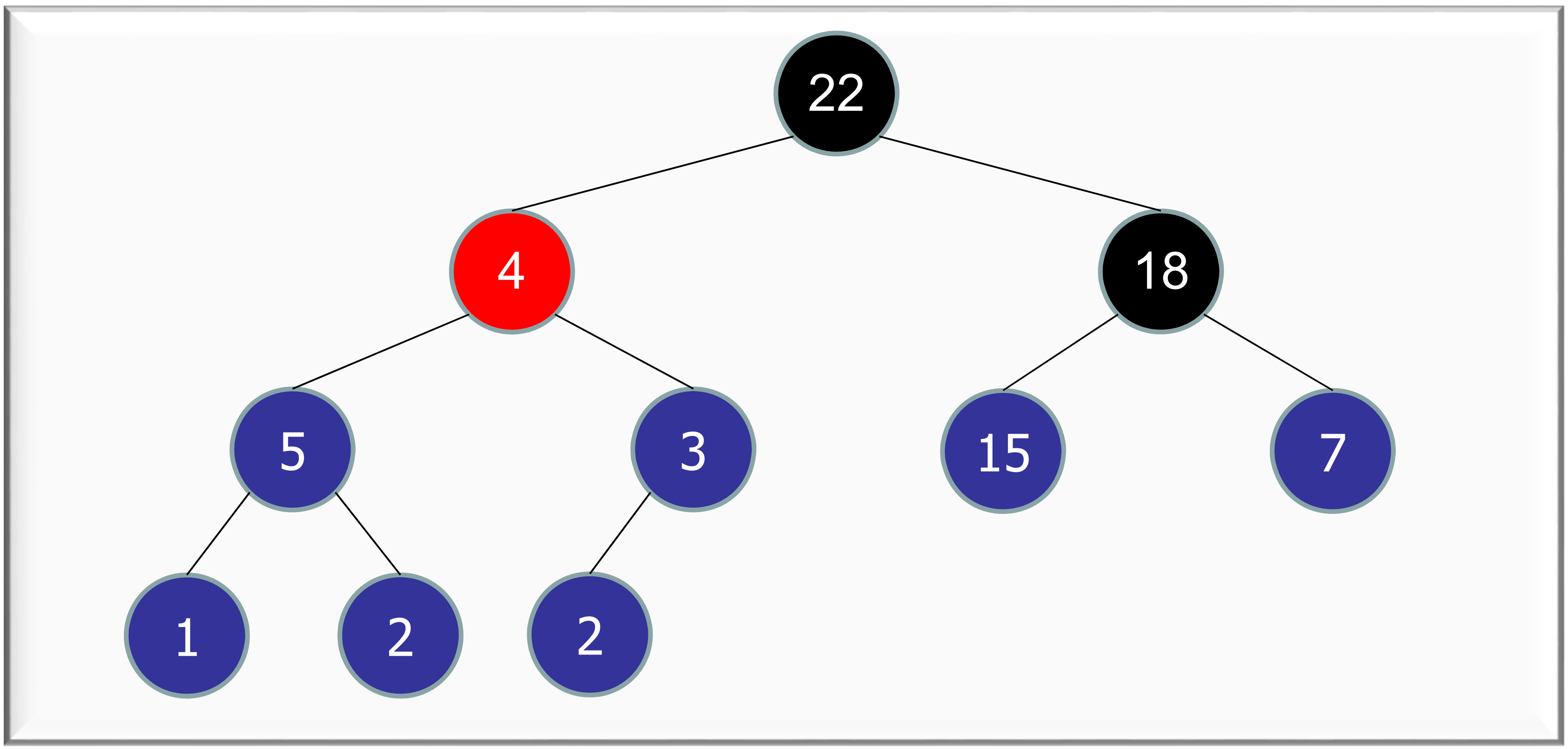
ARCHIPELAGO
is open



Heap Operations

`decreaseKey(28 → 4)` :

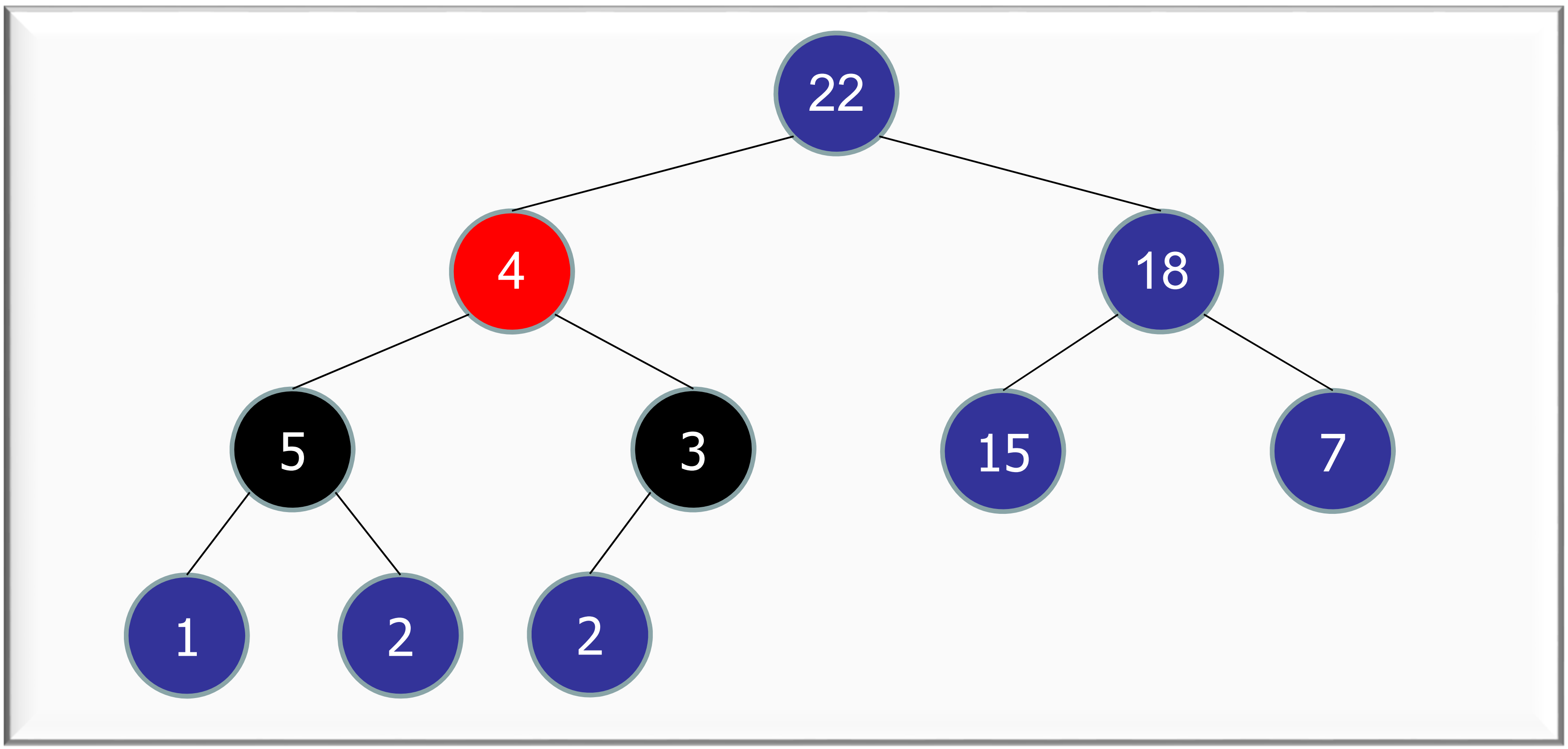
- Step 1: Update the priority
- Step 2: bubbleDown(4)



Heap Operations

`decreaseKey(28 → 4)` :

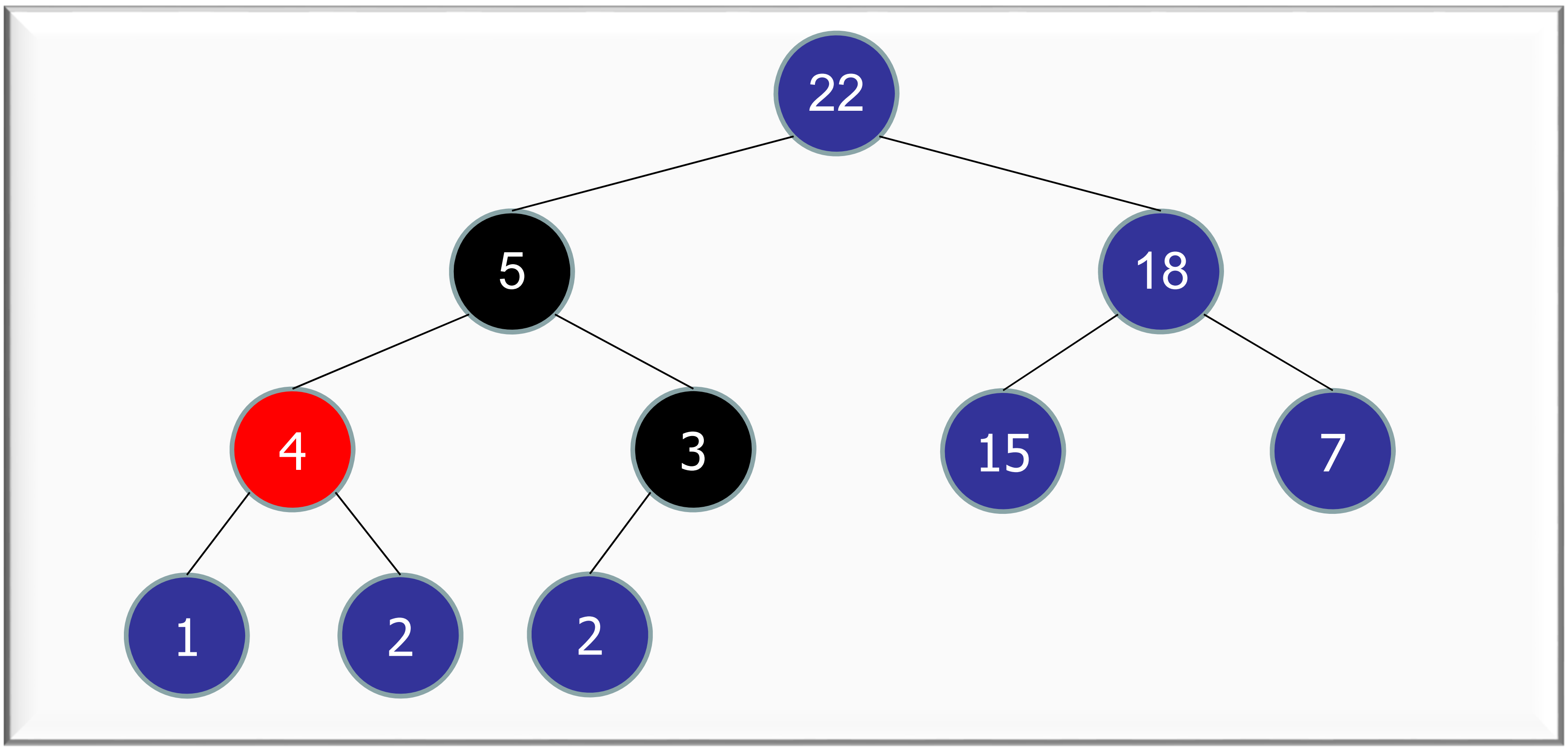
- Step 1: Update the priority
- Step 2: bubbleDown(4)



Heap Operations

`decreaseKey(28 → 4)` :

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

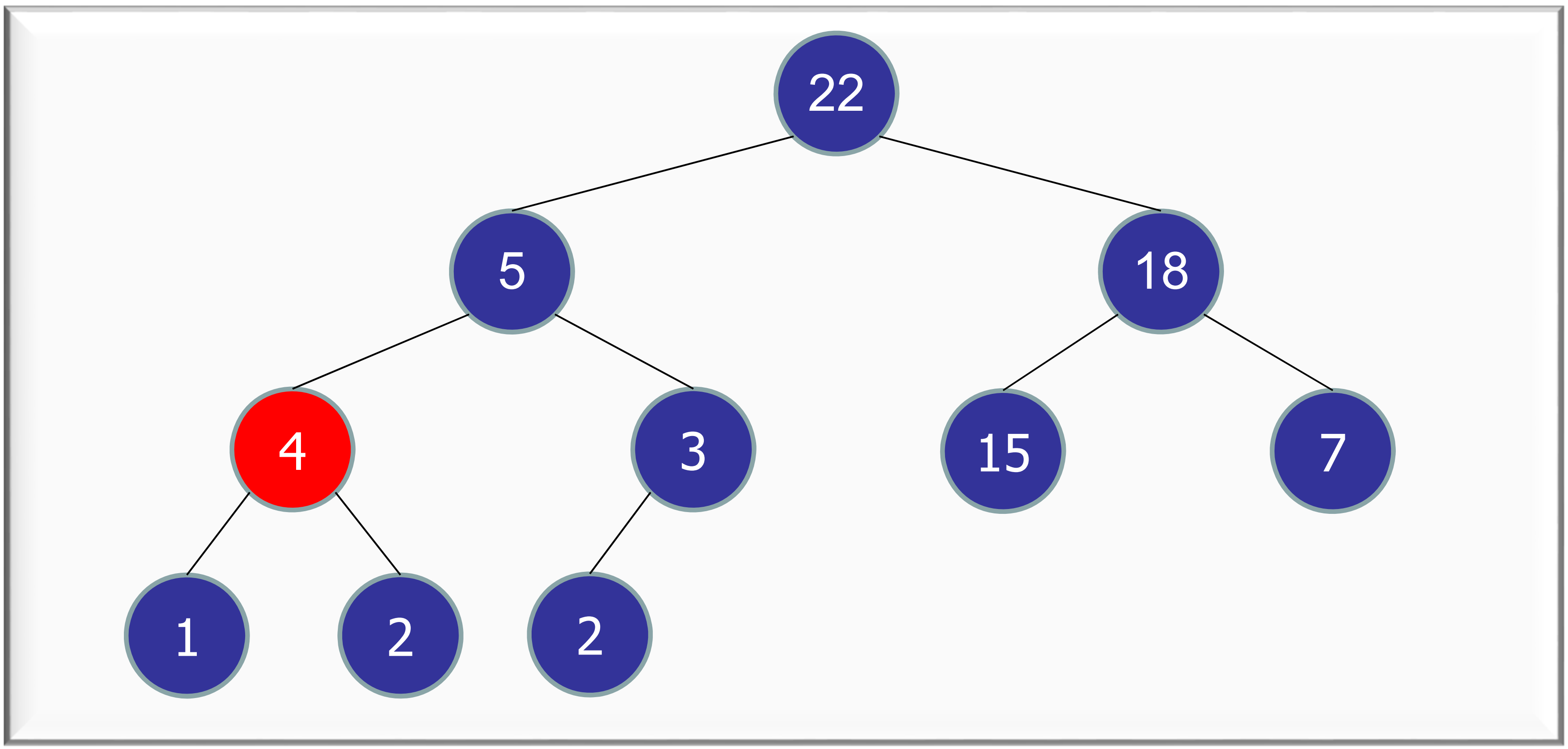


Heap Operations

`decreaseKey(28 → 4)` :

– Step 1: Update the priority

– Step 2: bubbleDown(4)



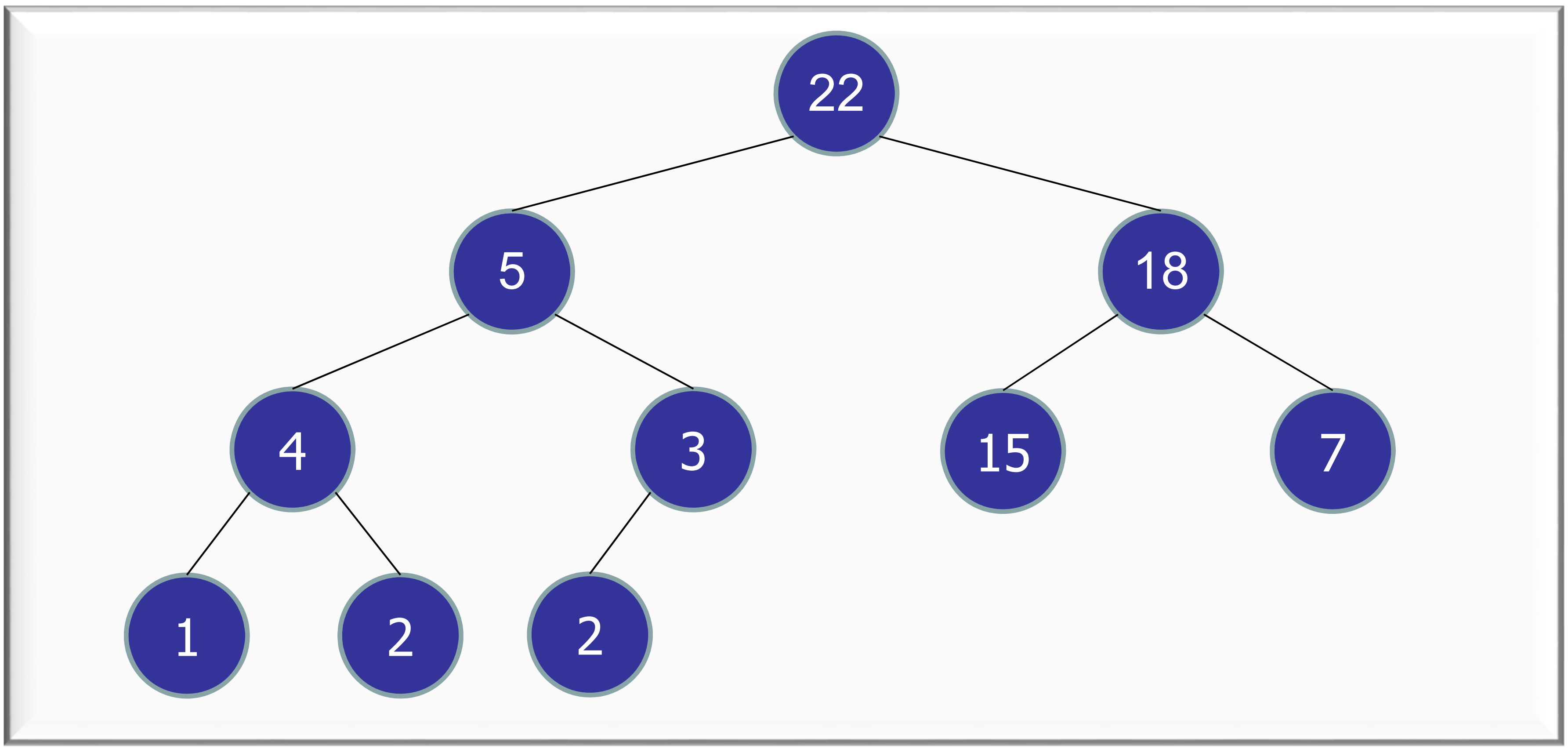
Heap Operations

```
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;
  }
```

Heap Operations

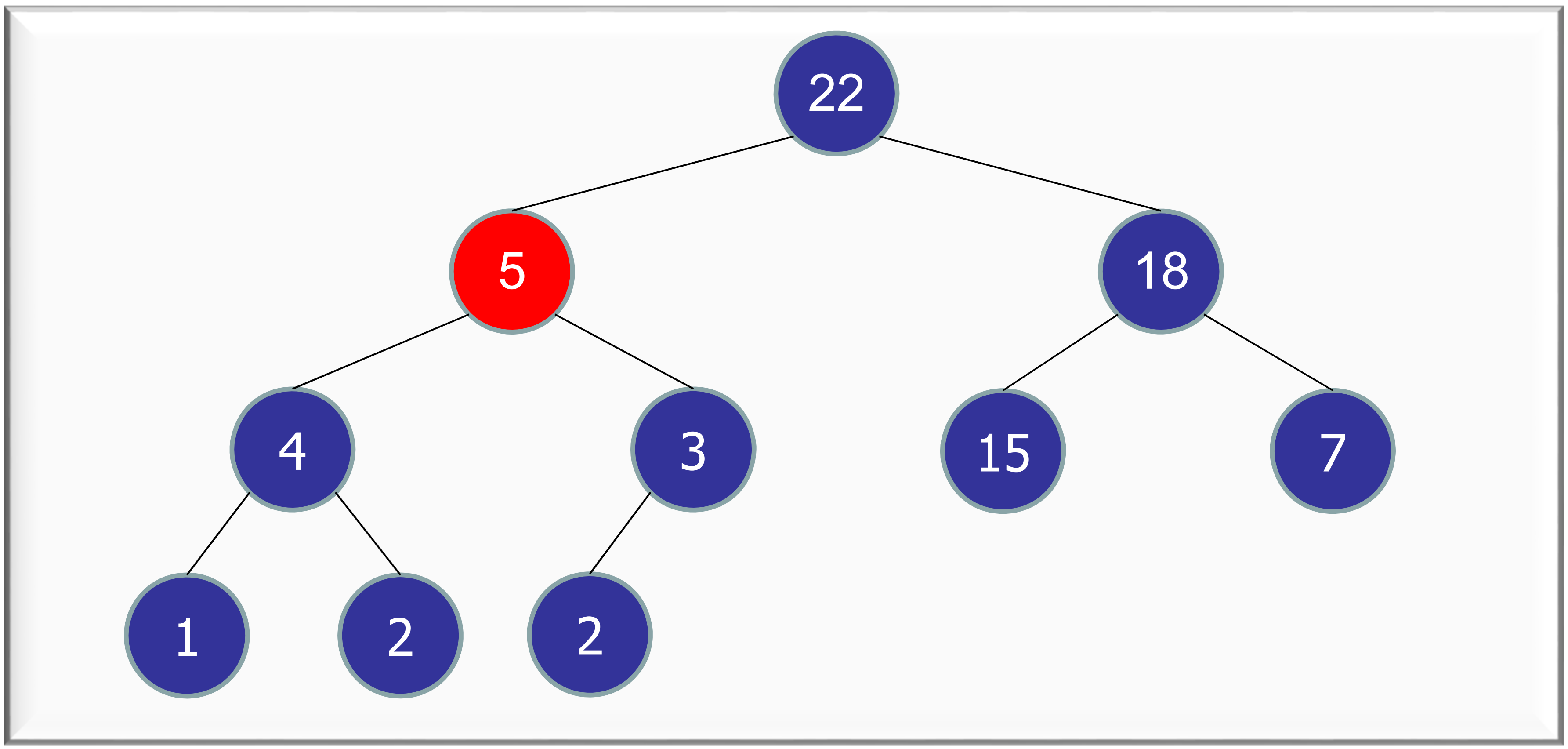
`decreaseKey(. . .)`:

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



Heap Operations

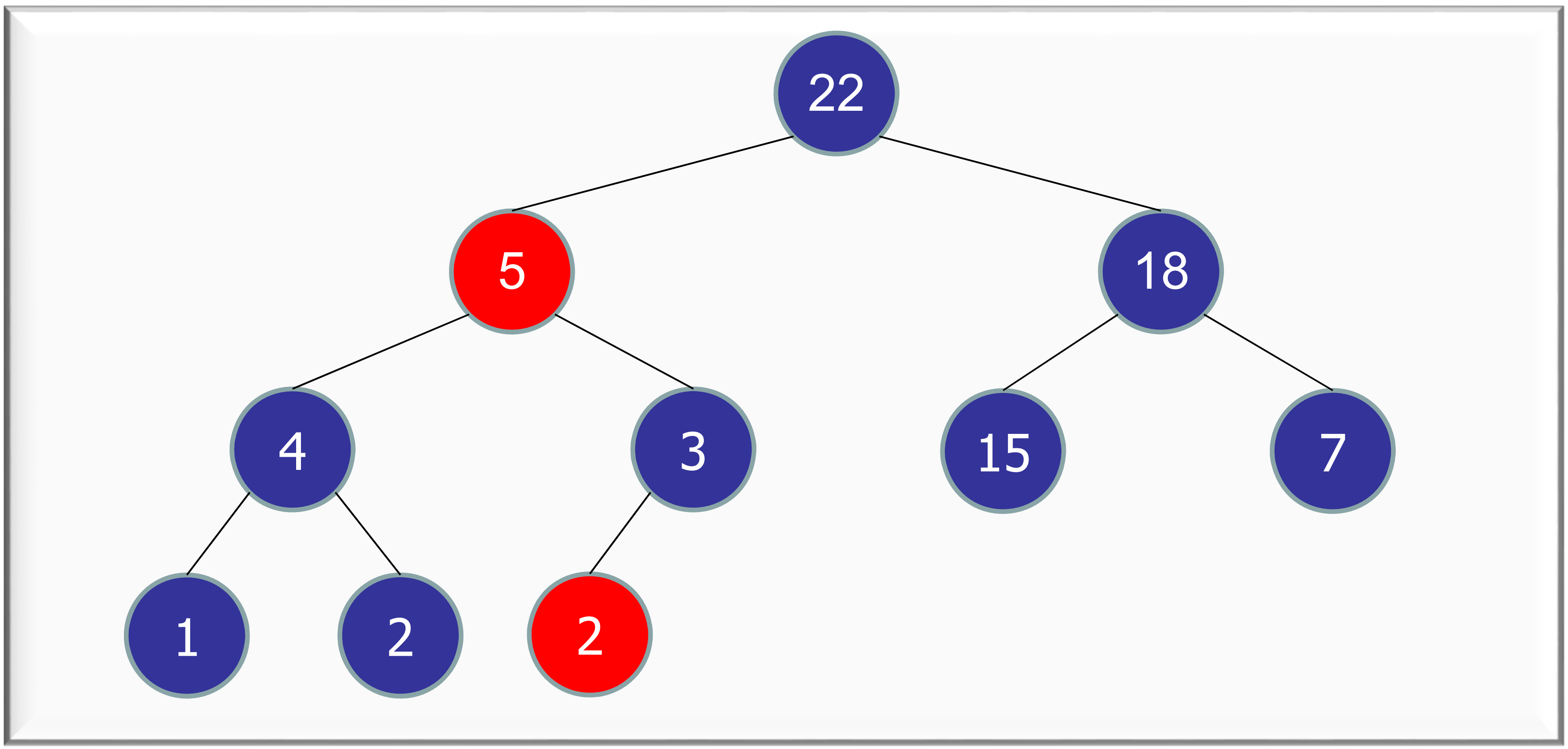
`delete(5)` :



Heap Operations

`delete(5)` :

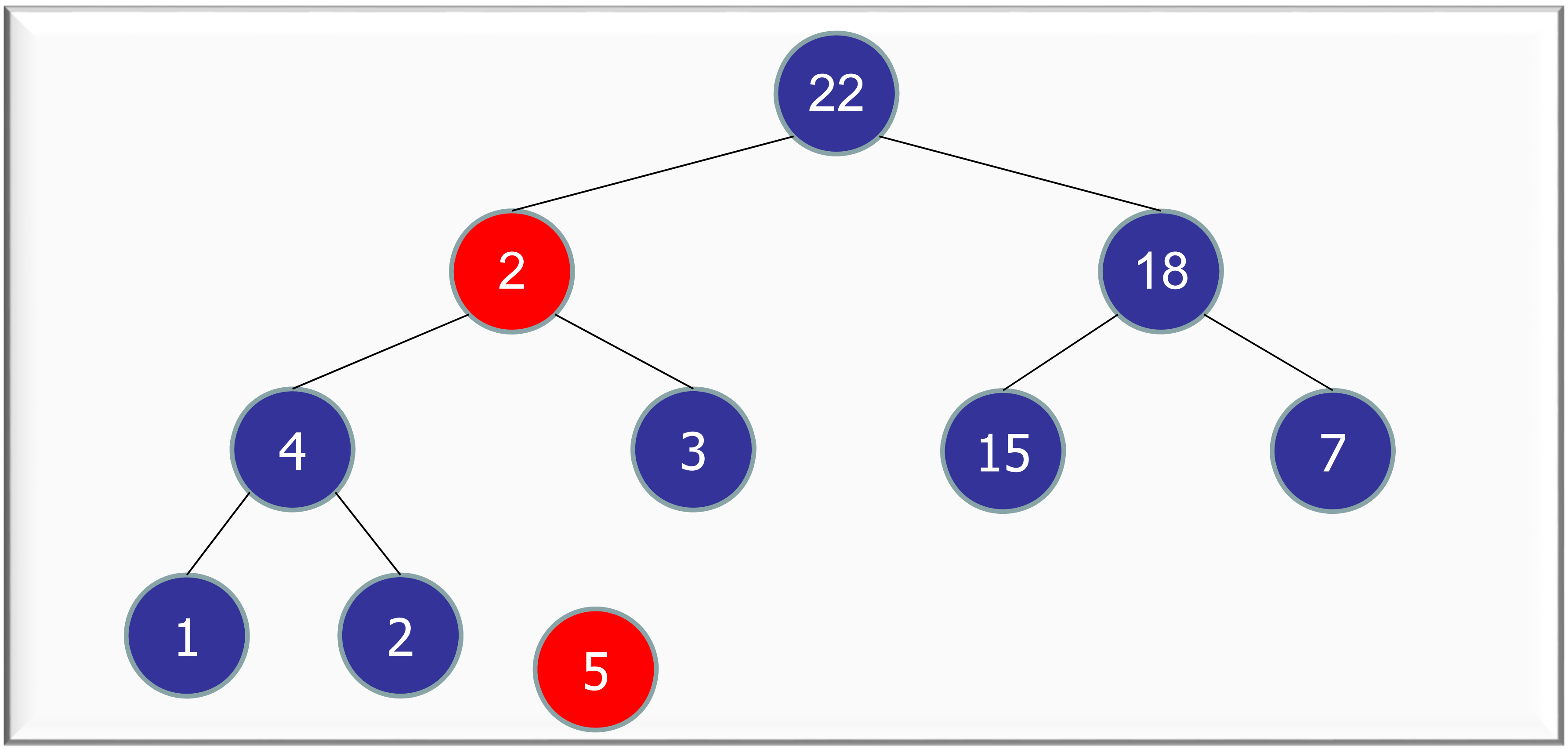
- swap(5, last())



Heap Operations

`delete(5)` :

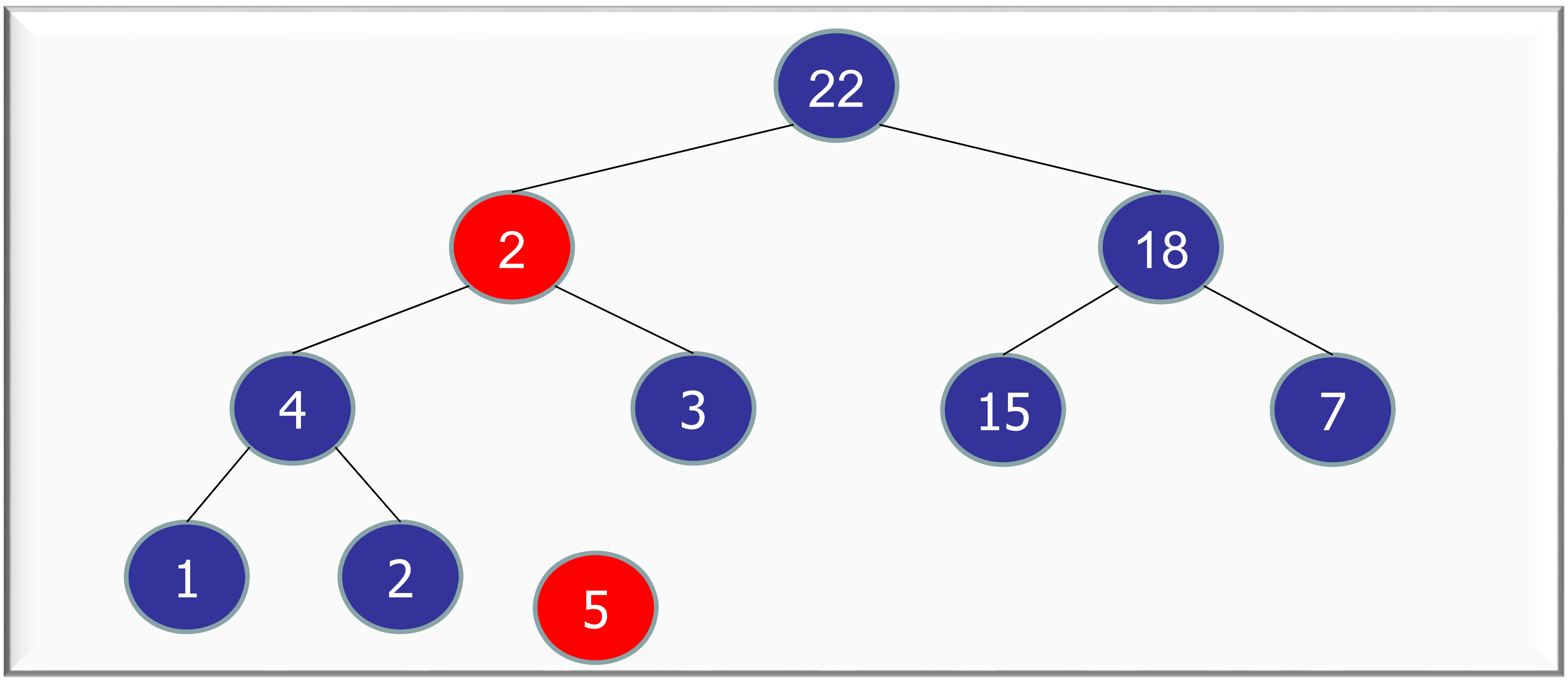
- swap(5, last())
- remove(last())



Heap Operations

`delete(5)` :

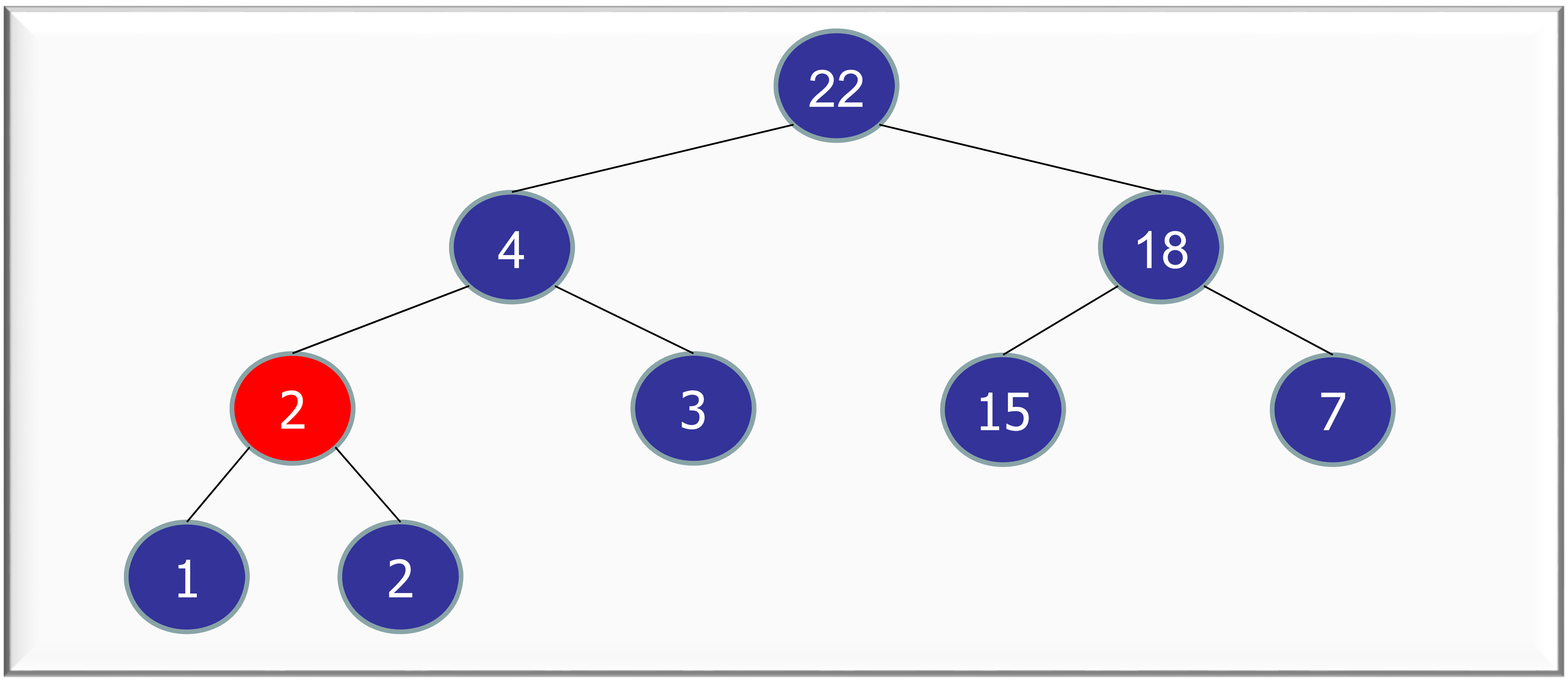
- swap(5, last())
- remove(last())
- bubbleDown(2)



Heap Operations

`delete(5)` :

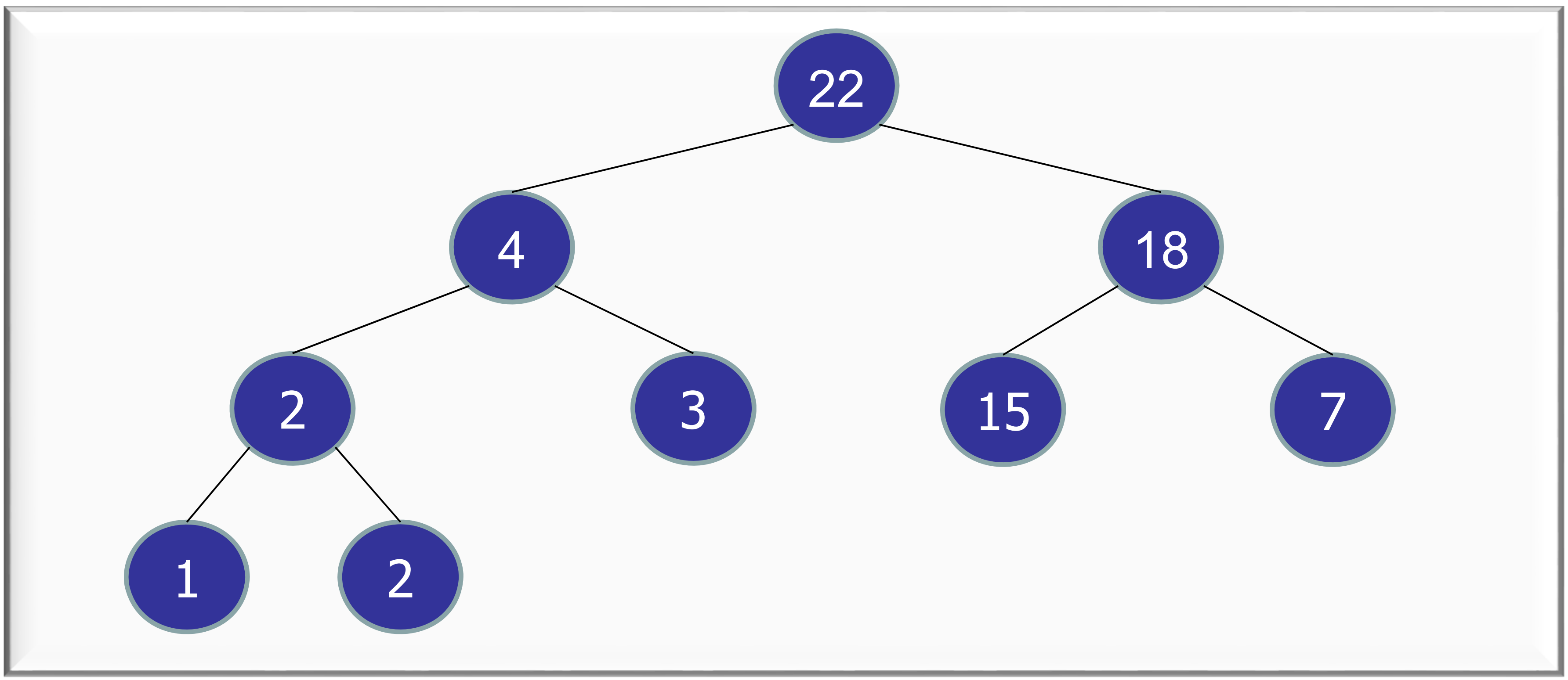
- swap(5, last())
- remove(last())
- bubbleDown(2)



Heap Operations

`delete(5)` :

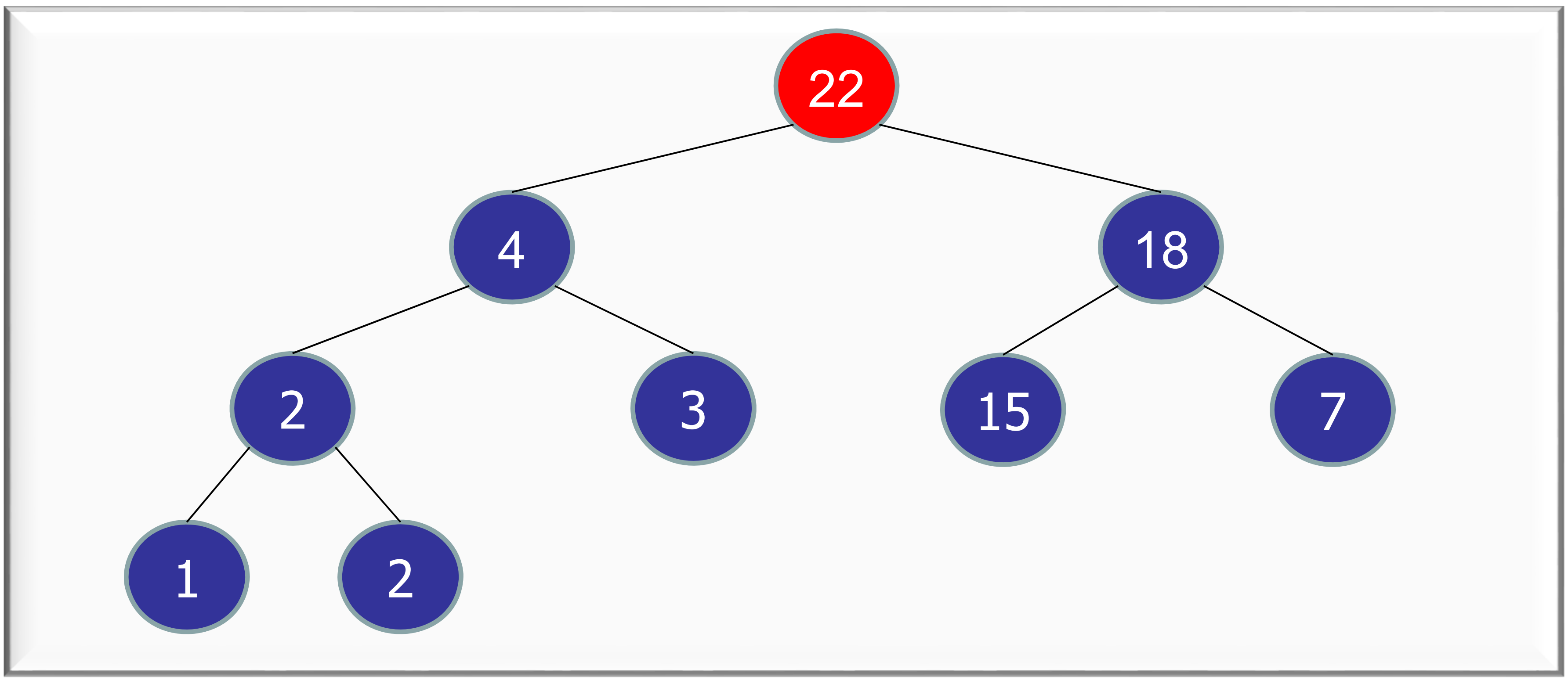
- swap(5, last())
- remove(last())
- bubbleDown(2)



Heap Operations

`extractMax()` :

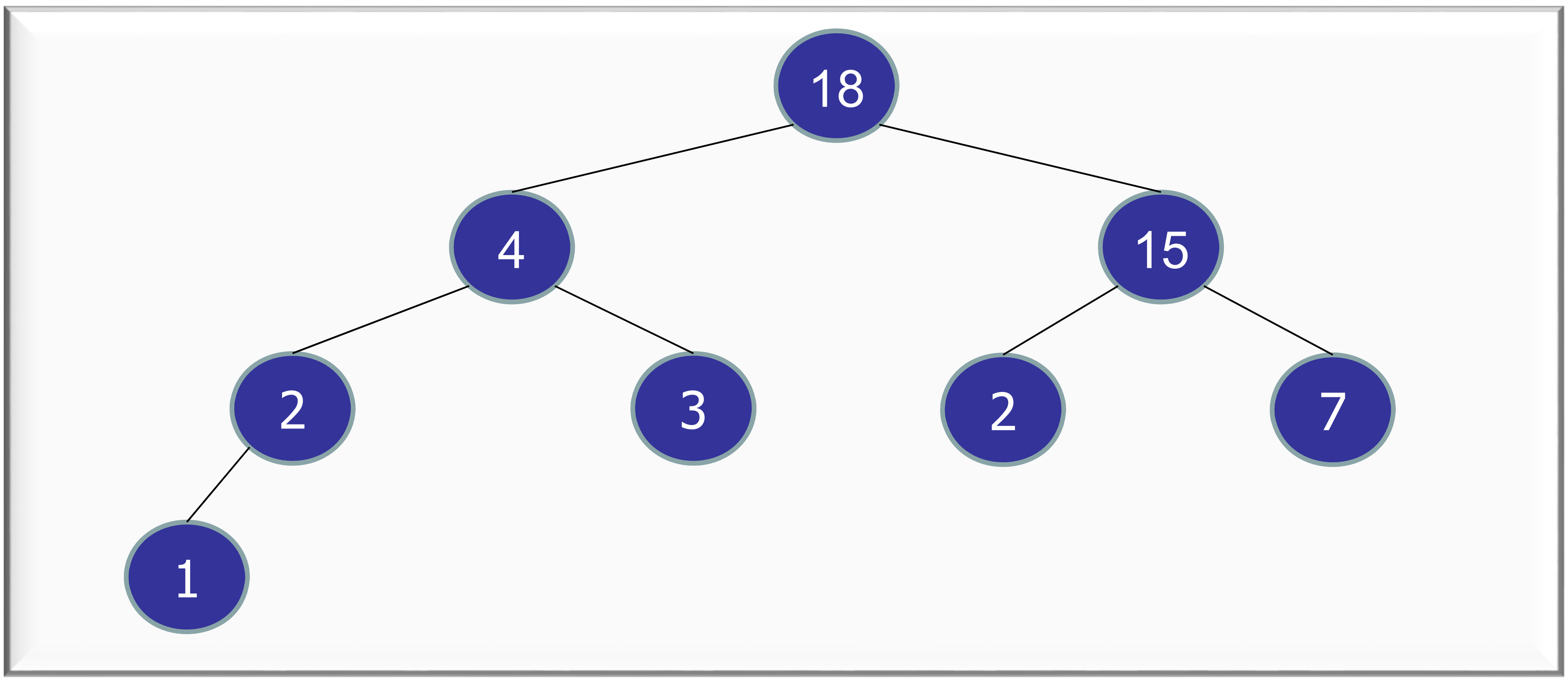
- Node v = root;
- delete(root);



Heap Operations

`extractMax()` :

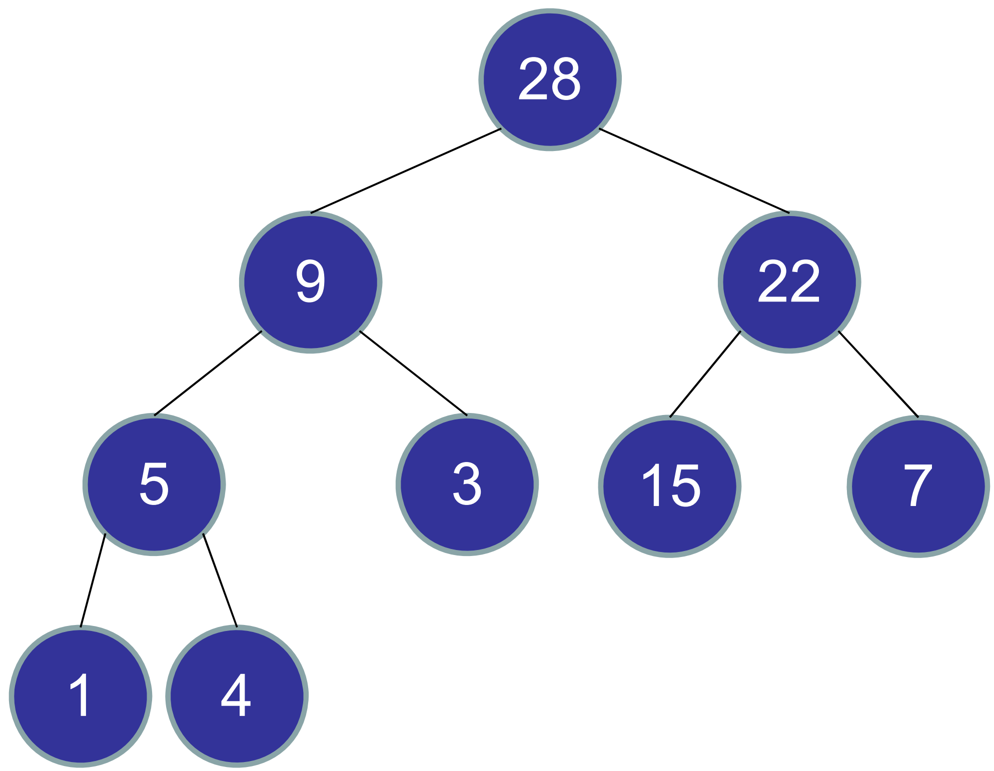
- Node v = root;
- delete(root);



(Max) Priority Queue

Heap Operations: O(log n)

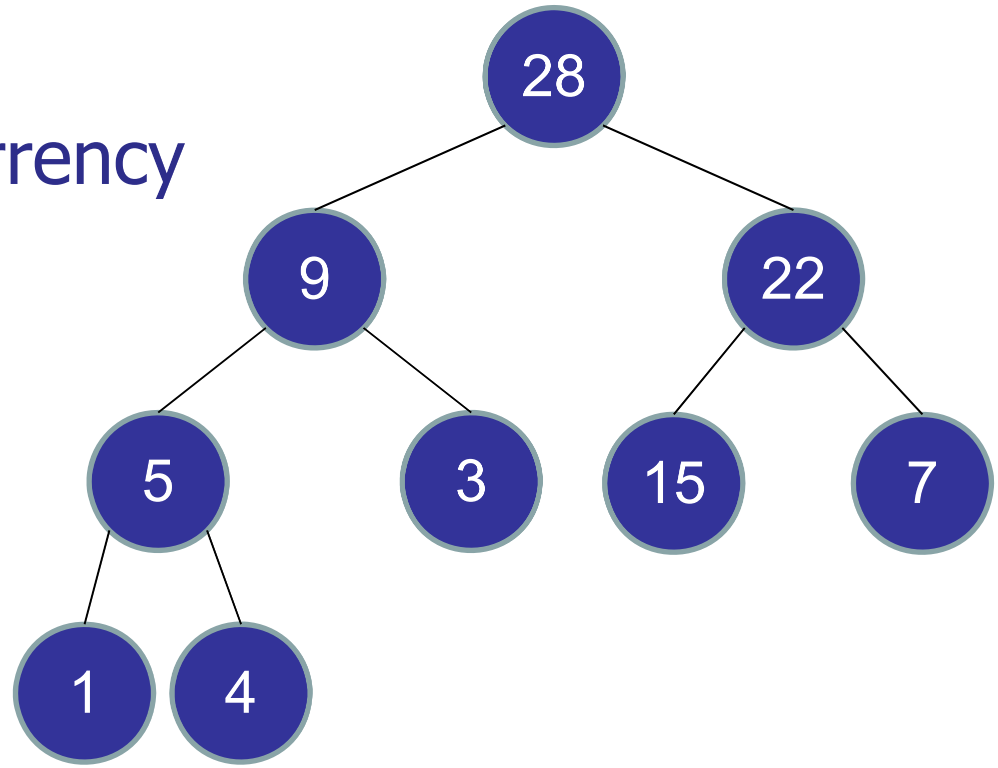
- insert
- extractMax
- increaseKey
- decreaseKey
- delete



(Max) Priority Queue

Heap vs. AVL Tree

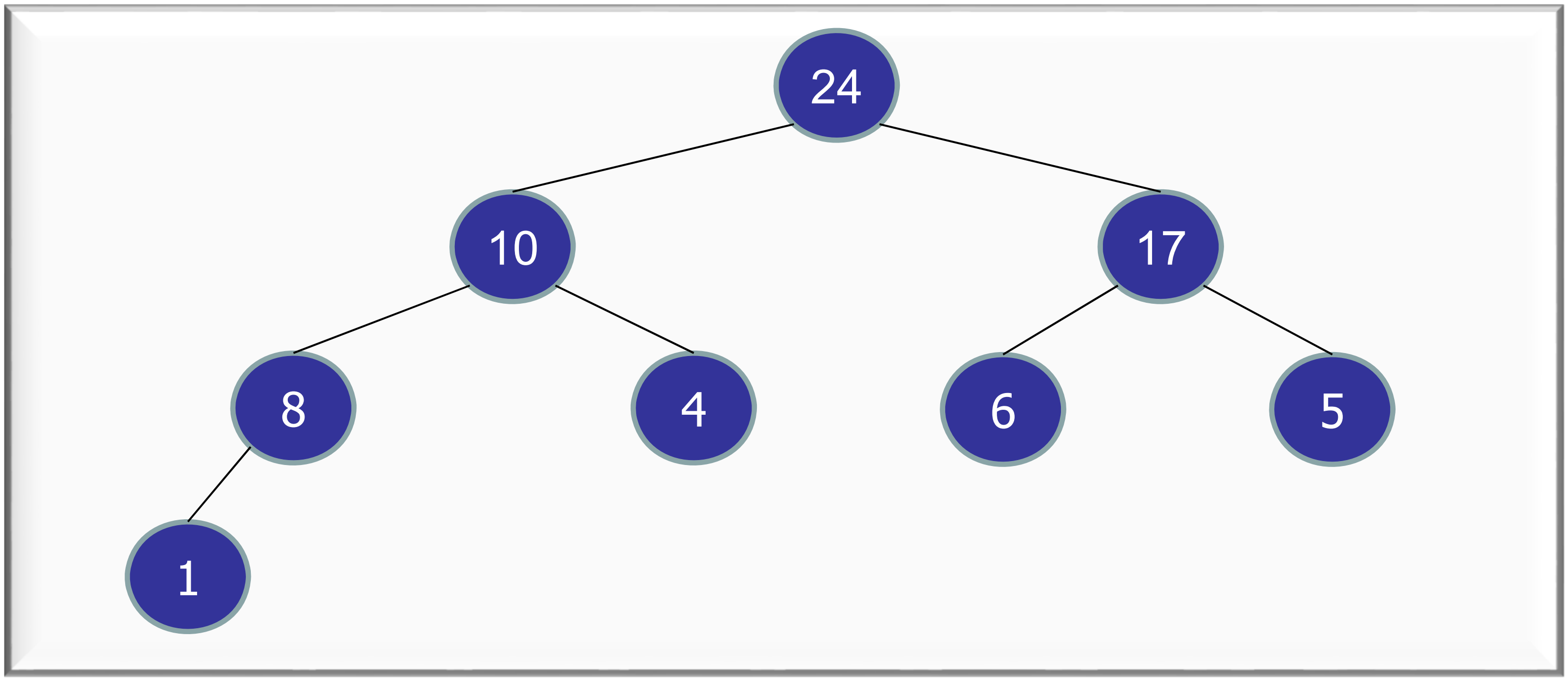
- Same asymptotic cost for operations
- Faster real cost (no constant factors!)
- Simpler: no rotations
- Slightly better concurrency



Store Tree in an Array

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

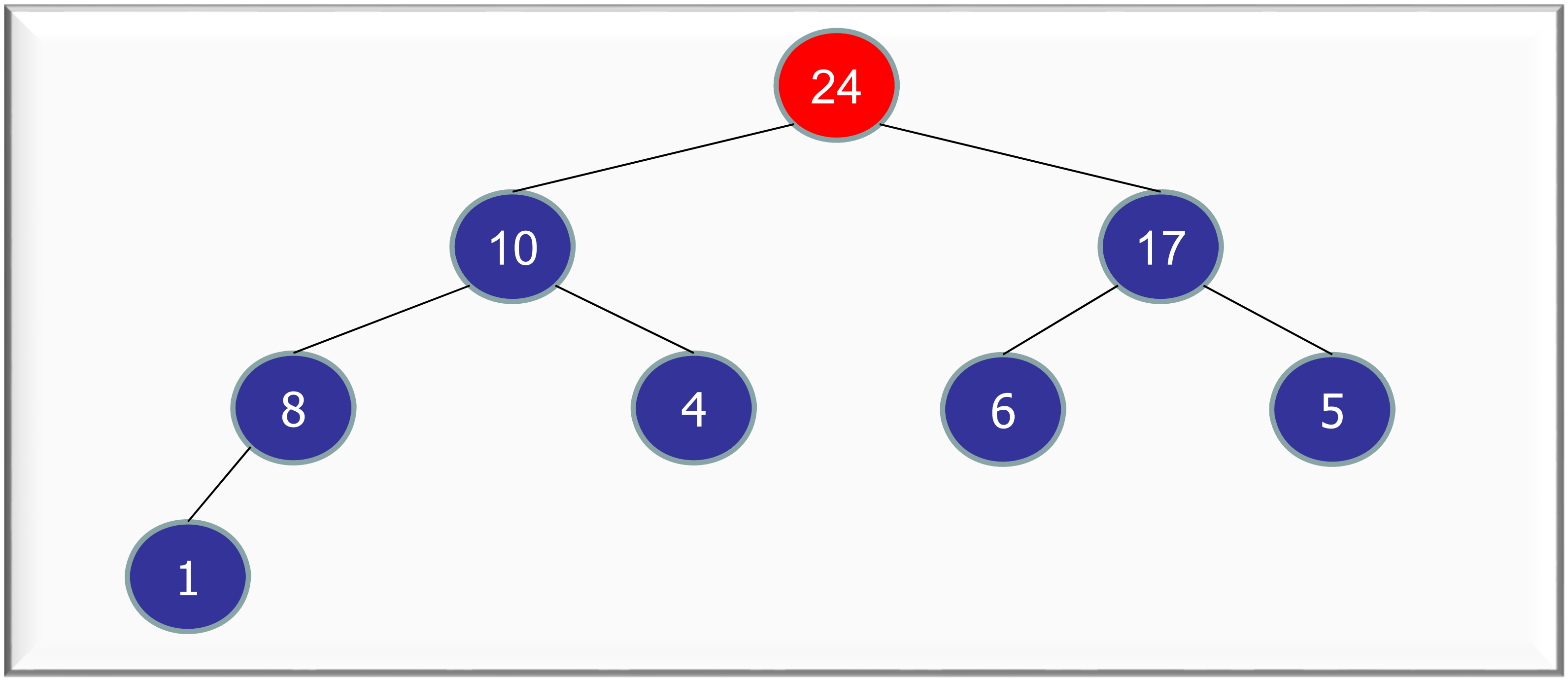
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	7	1	



Store Tree in an Array

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

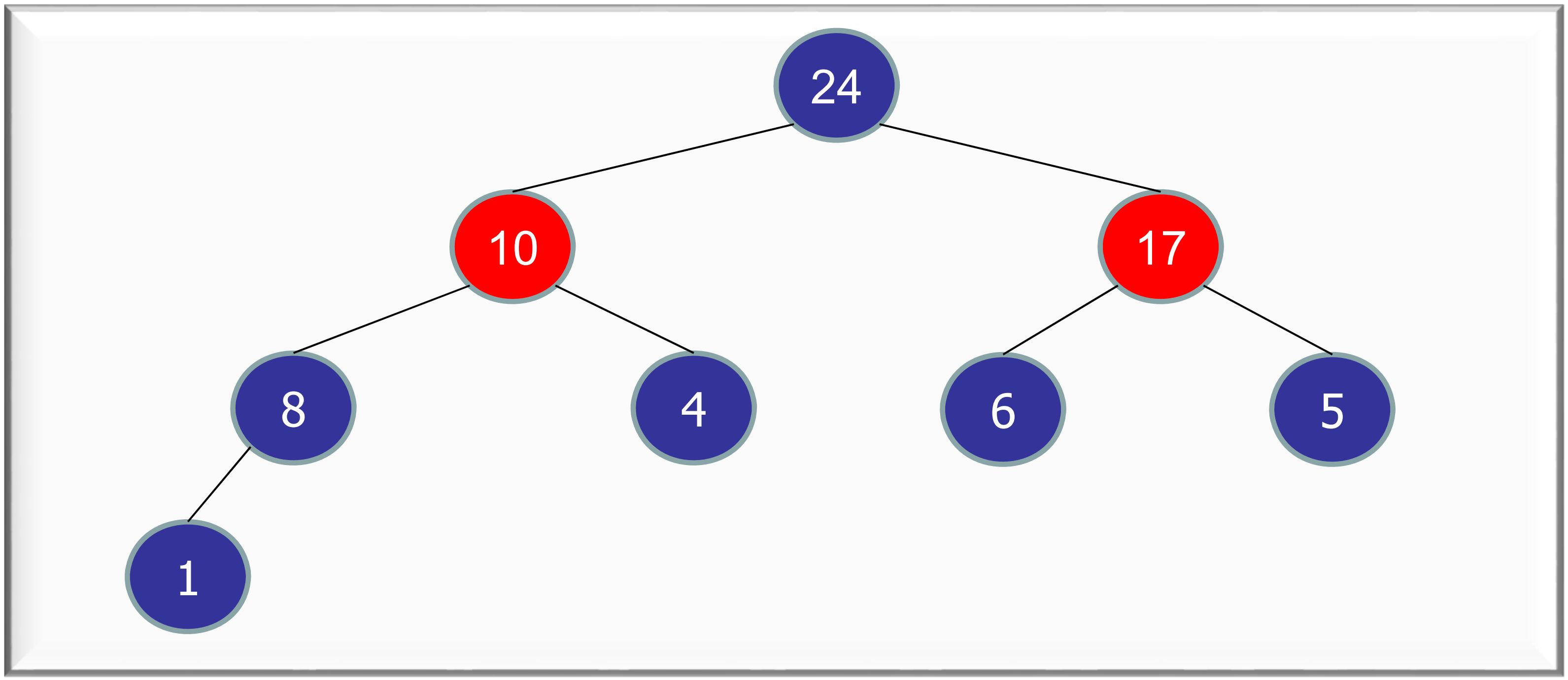
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	7	1	



Store Tree in an Array

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

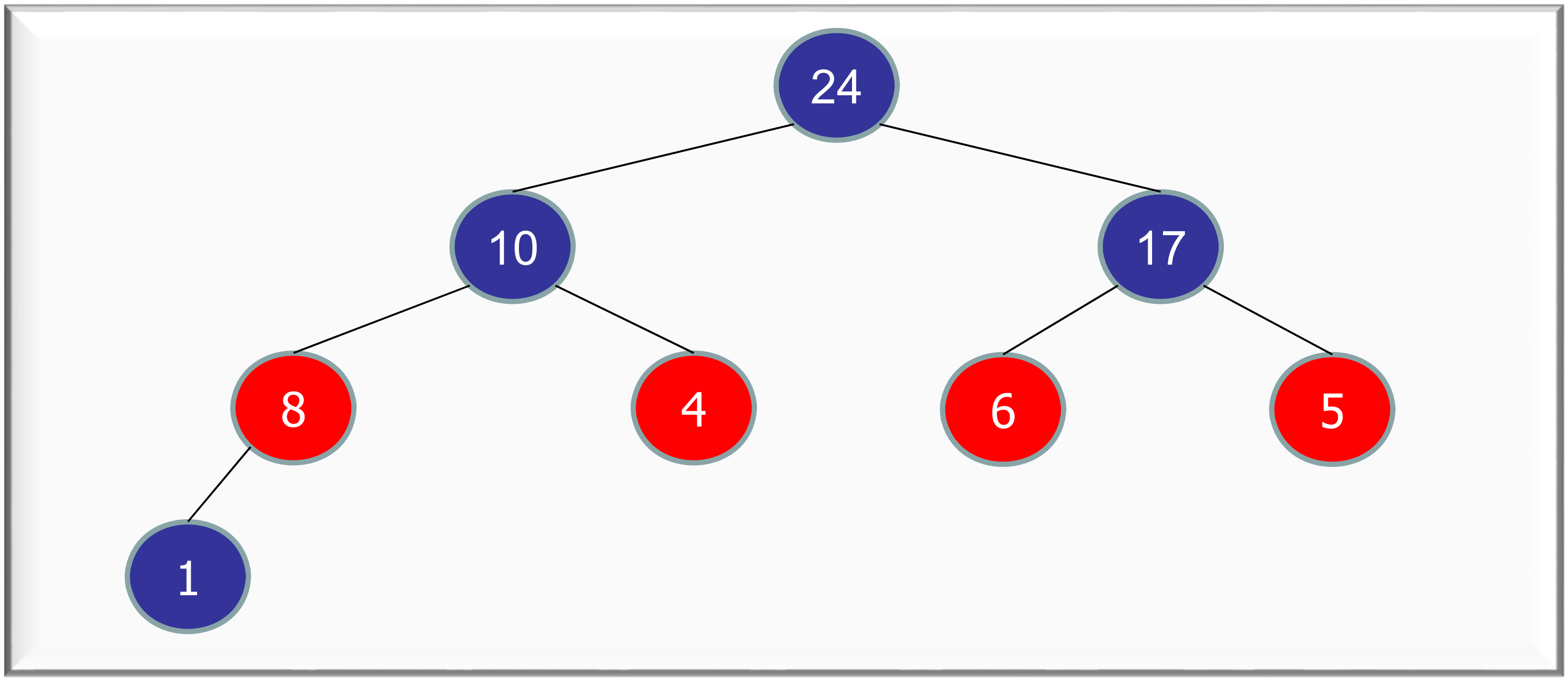
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	7	1	



Store Tree in an Array

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

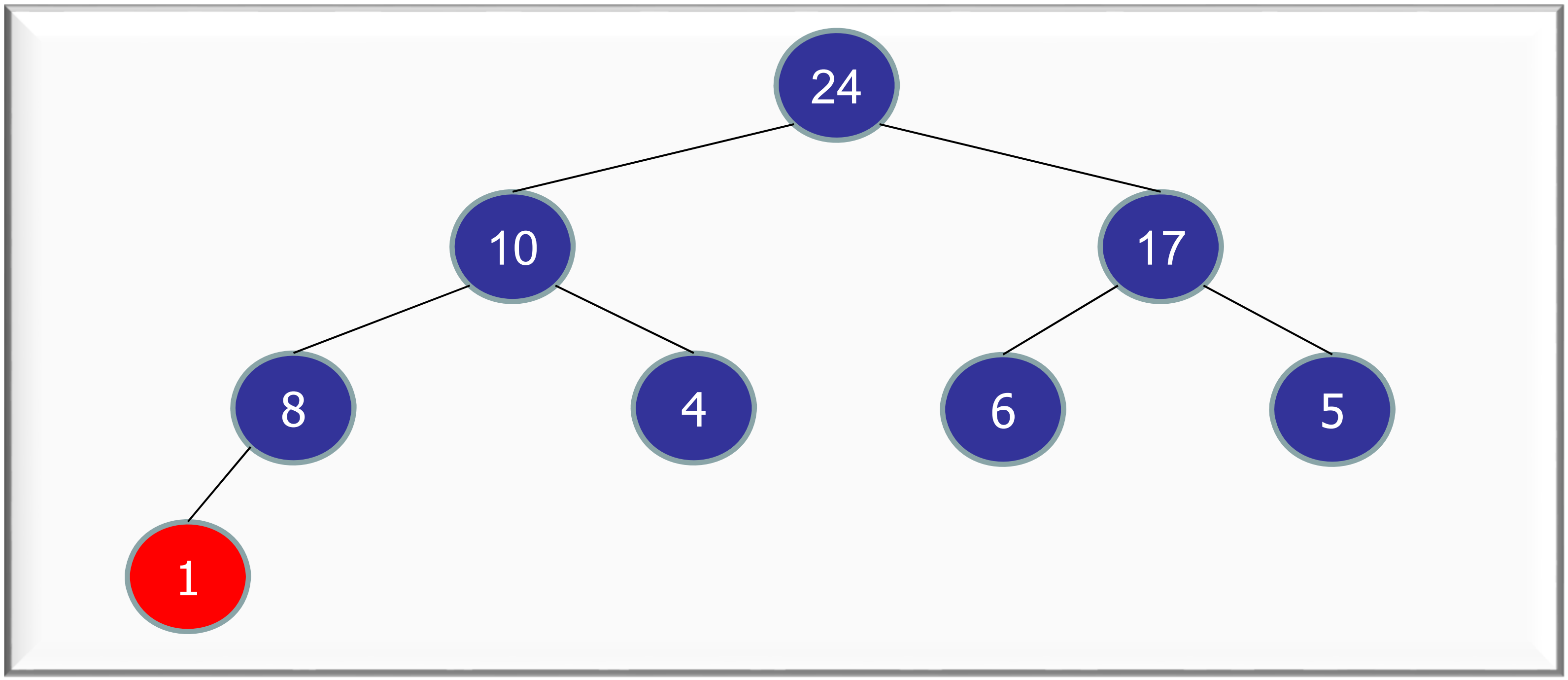
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	



Store Tree in an Array

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

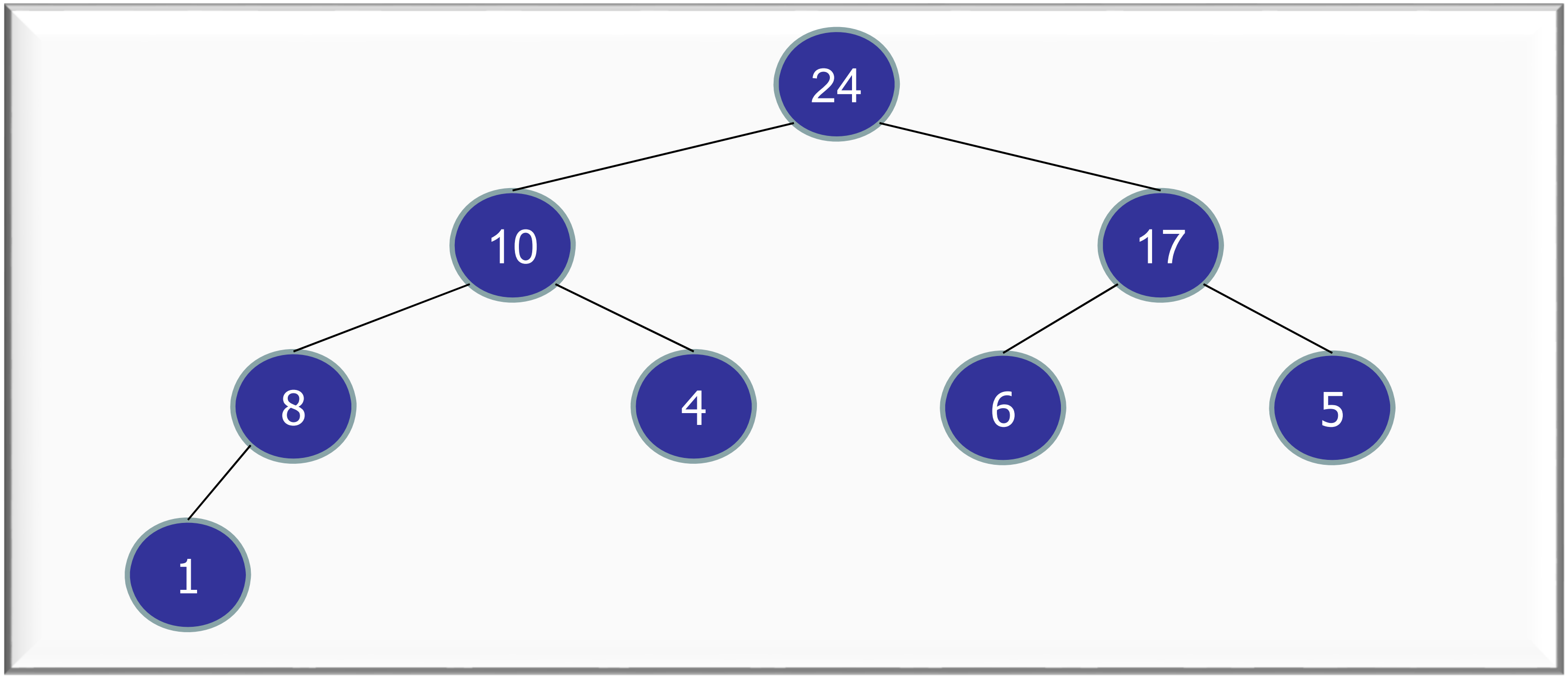
array slot	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
priority	**24**	**10**	**17**	**8**	**4**	**6**	**5**	**1**	



Store Tree in an Array

`insert(15) :`

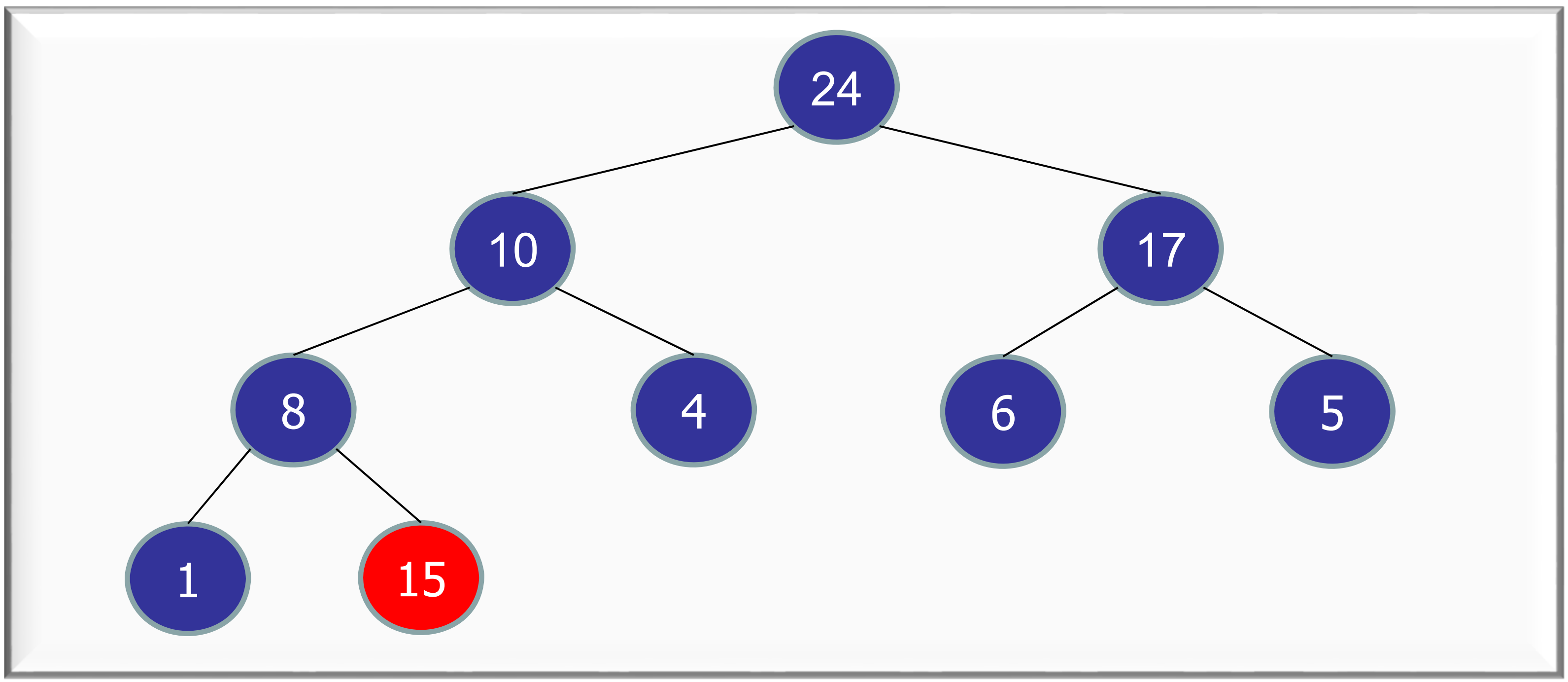
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	



Store Tree in an Array

`insert(15) :`

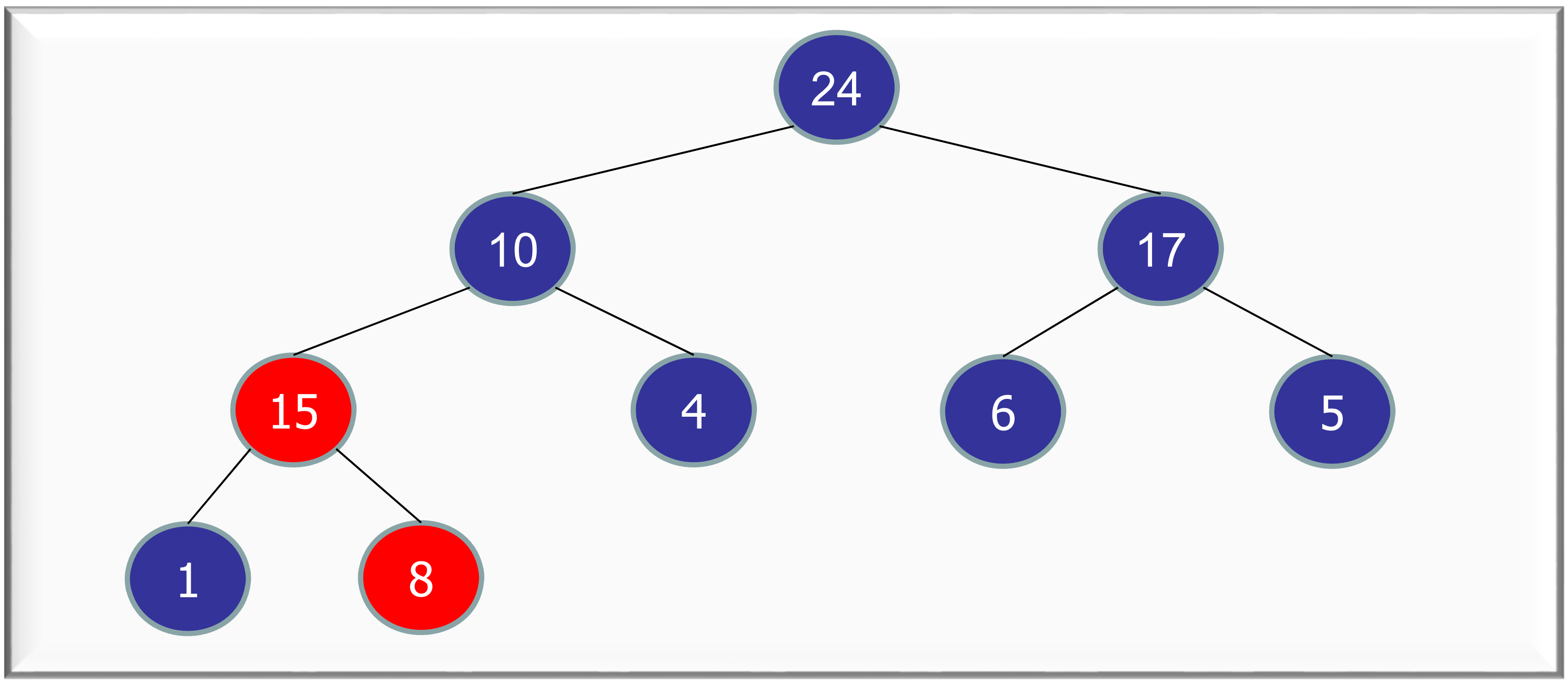
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	15



Store Tree in an Array

`insert(15) :`

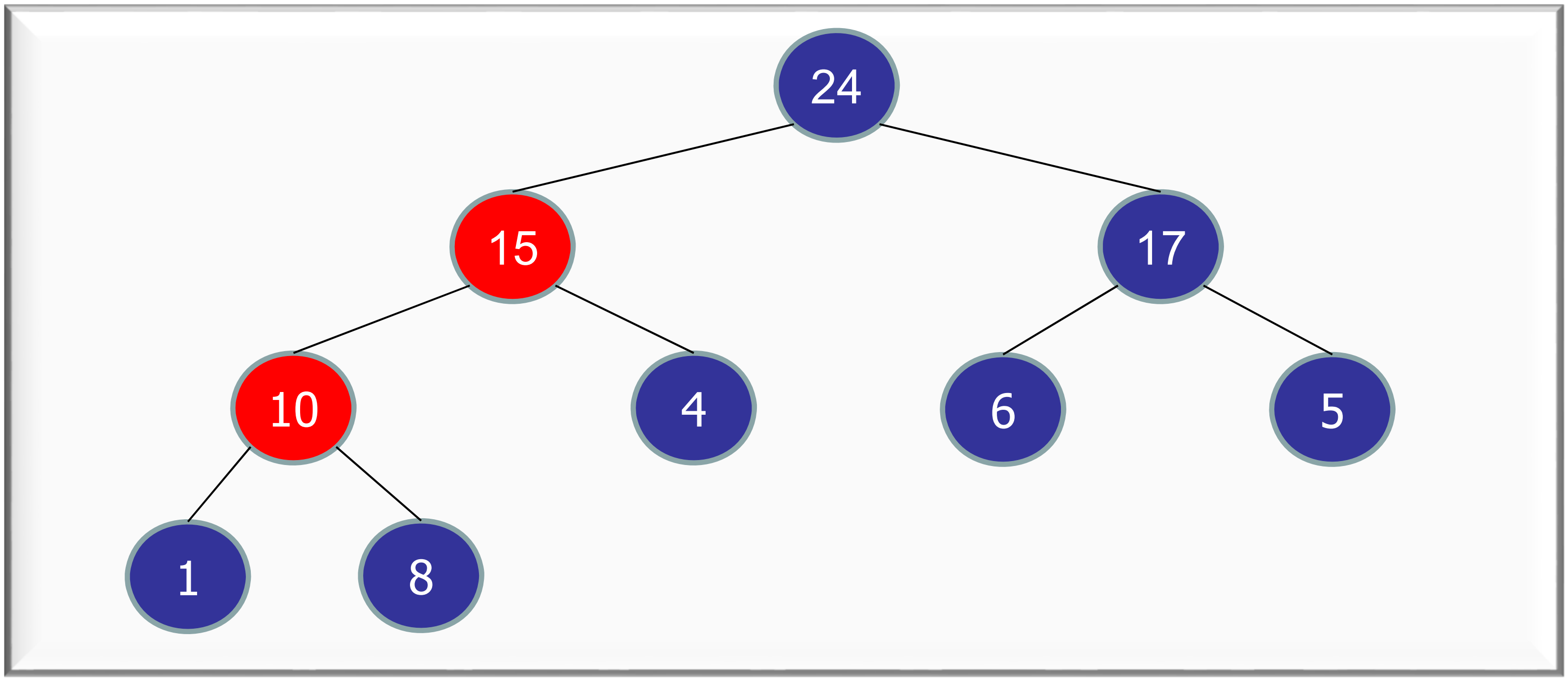
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	15	4	6	5	1	8



Store Tree in an Array

`insert(15) :`

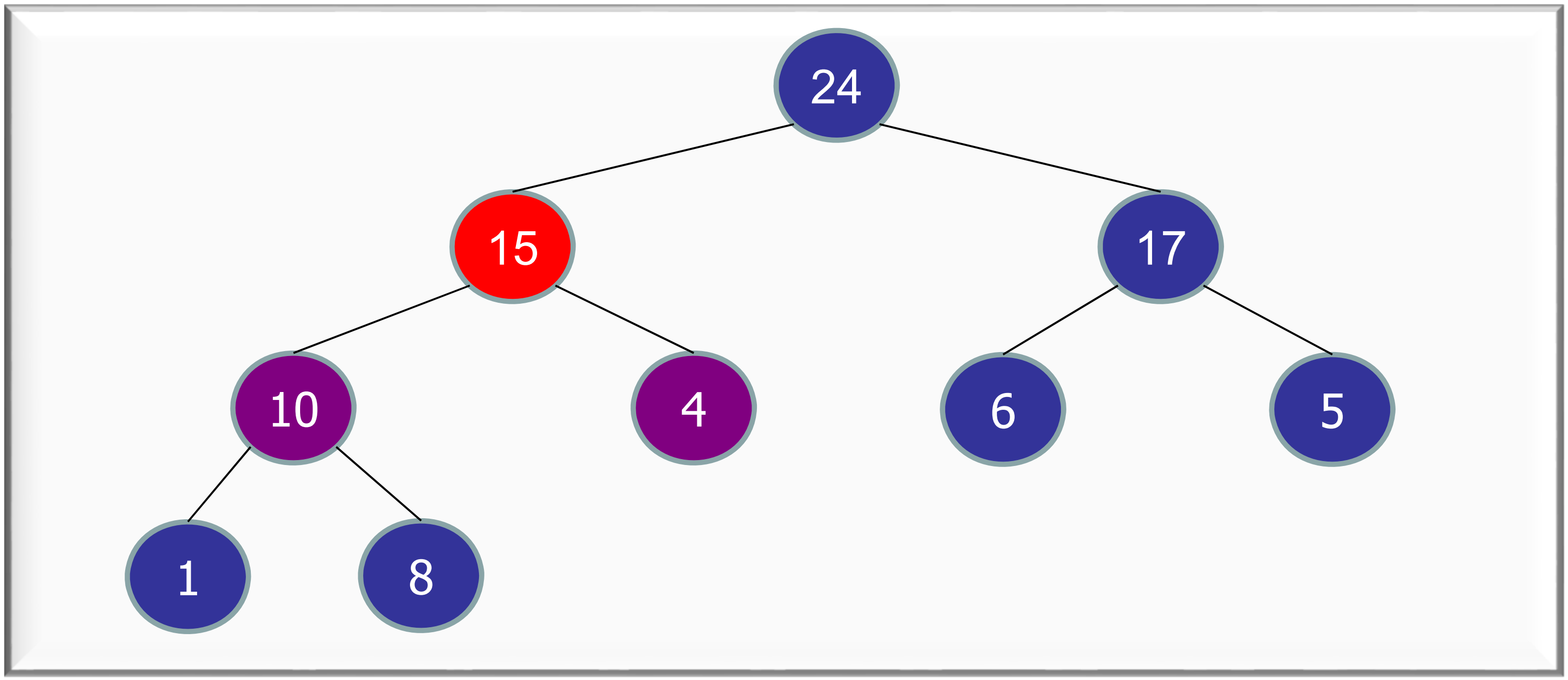
array slot	0	1	2	3	4	5	6	7	8
priority	24	15	17	10	4	6	5	1	8



Store Tree in an Array

```
left(x)  = ??
right(x) = ??
```

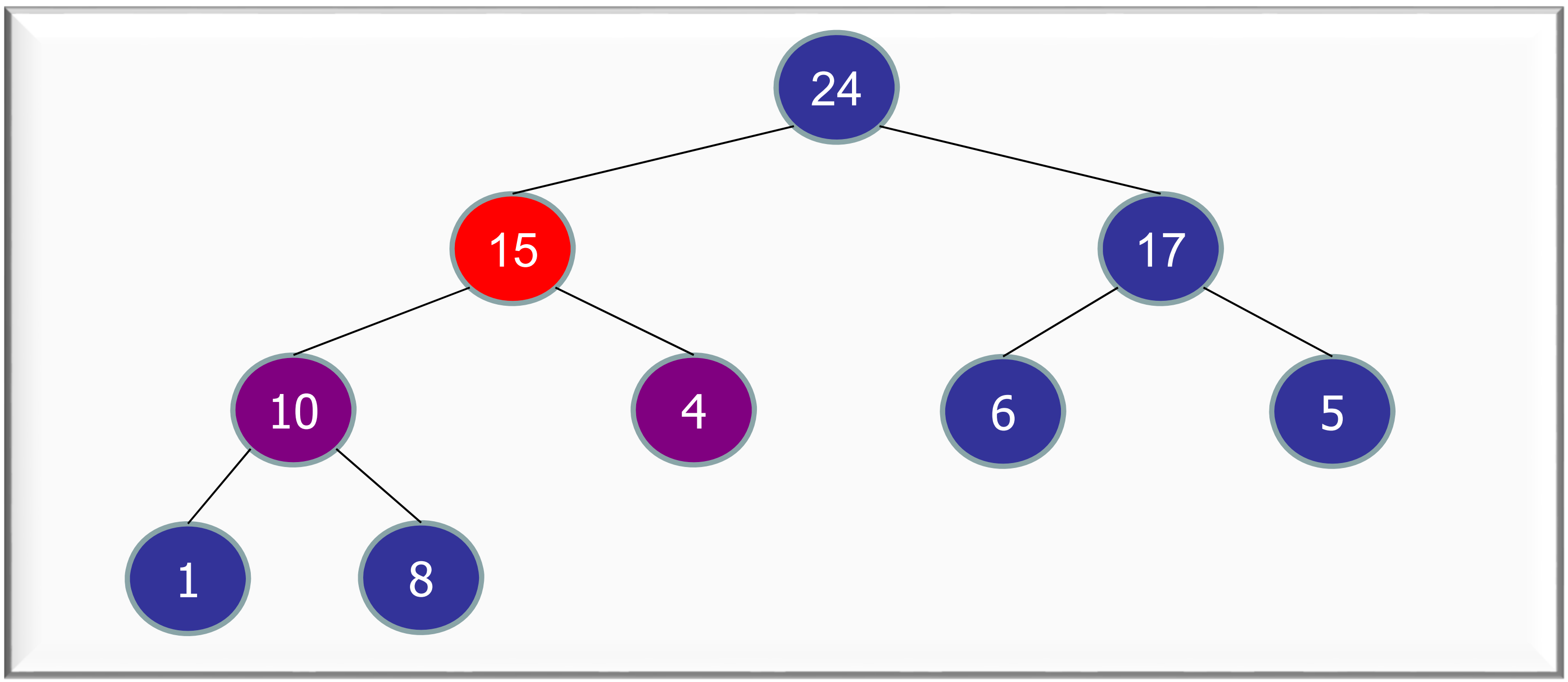
array slot	0	1	2	3	4	5	6	7	8
priority	24	15	17	10	4	6	5	1	8



Store Tree in an Array

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

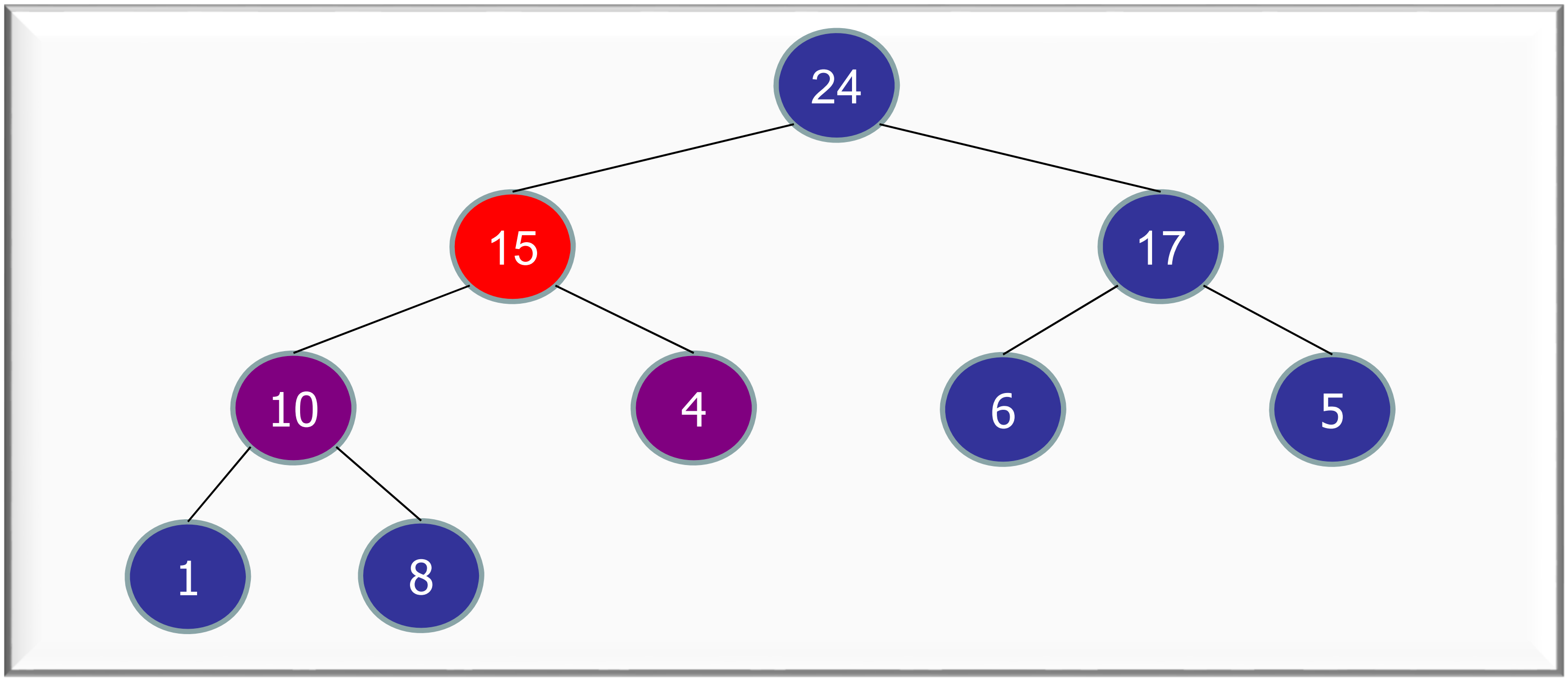
array slot	0	1	2	3	4	5	6	7	8
priority	24	15	17	10	4	6	5	1	8



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



Can we store an AVL tree in an array?

If so, how? If not, why not?

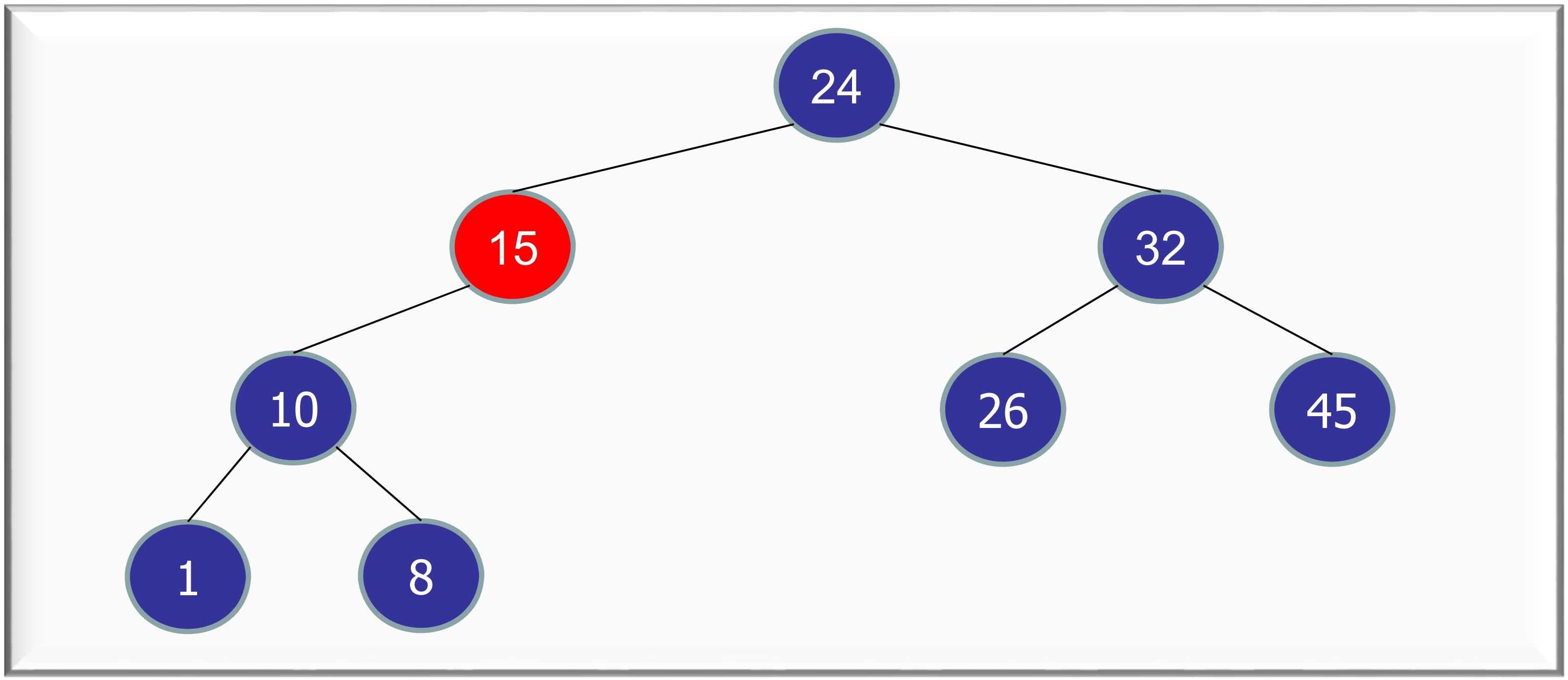
ARCHIPELAGO

is open

Store Tree in an Array

`right-rotate(15)`

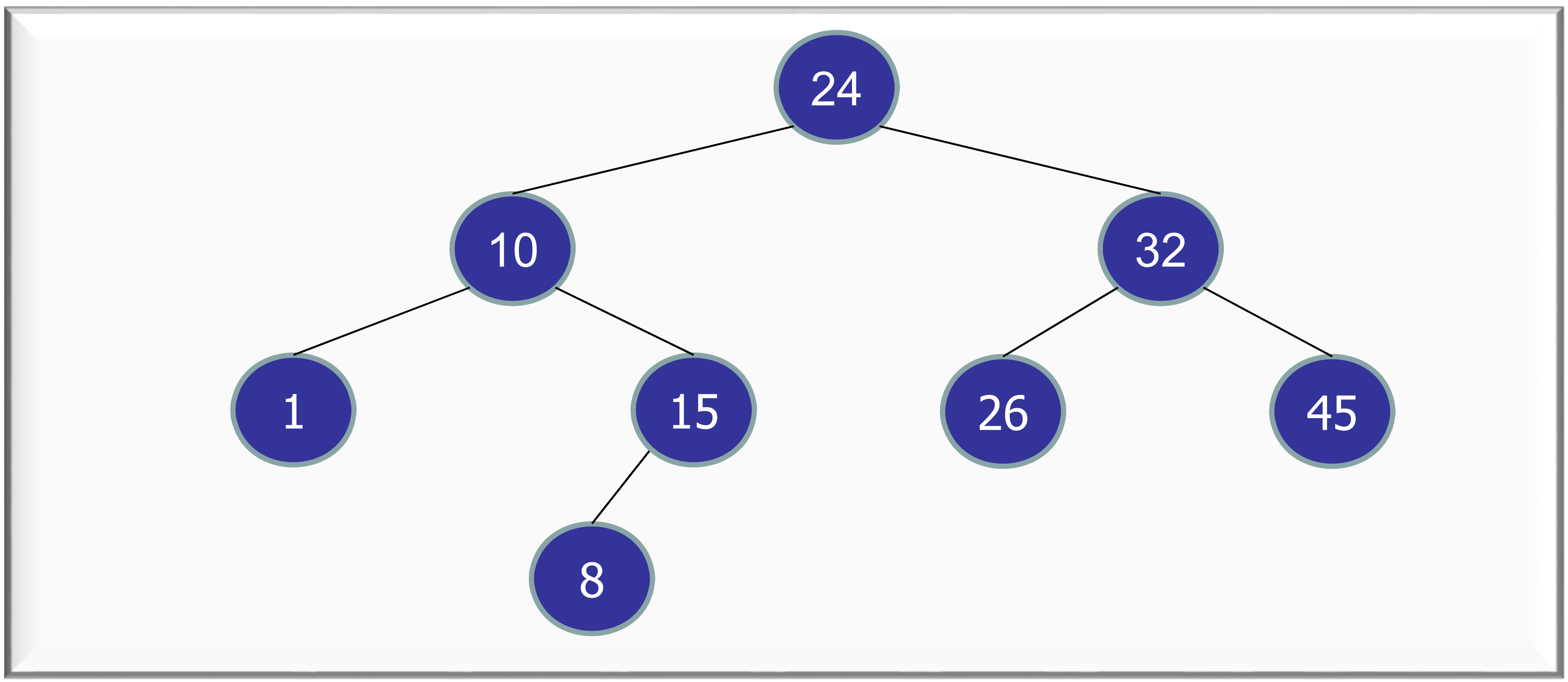
array slot	0	1	2	3	4	5	6	7	8
priority	24	15	32	10		26	45	1	8



Store Tree in an Array

`right-rotate(15)` : **not an O(1) operation!**

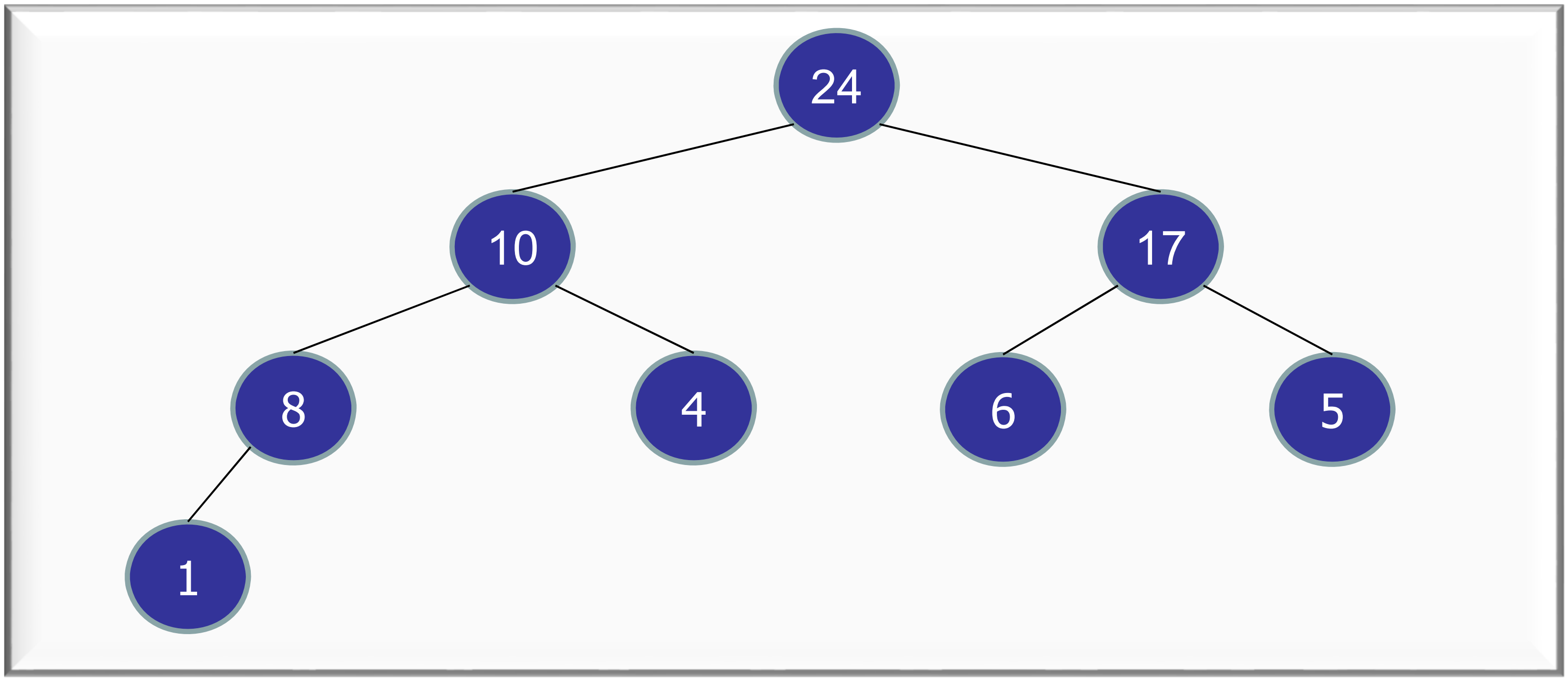
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	32	1	15	26	45	8	



Store Tree in an Array

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

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



HeapSort

Unsorted list:

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

HeapSort

Unsorted list:

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

Step 1. 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

Step 1. Unsorted list ➔ Heap

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

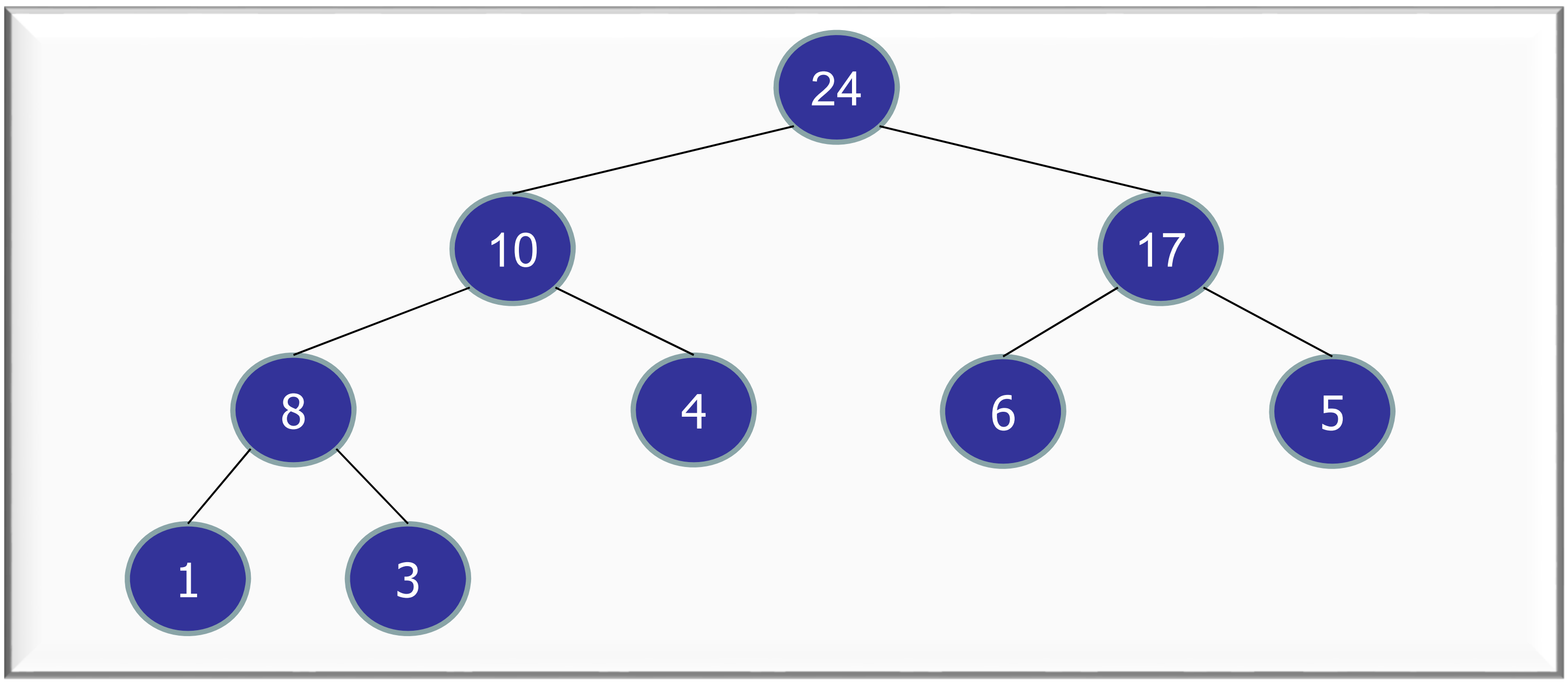
Step 2. Heap ➔ Sorted list:

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

HeapSort

Step 2. Heap ➔ Sorted list:

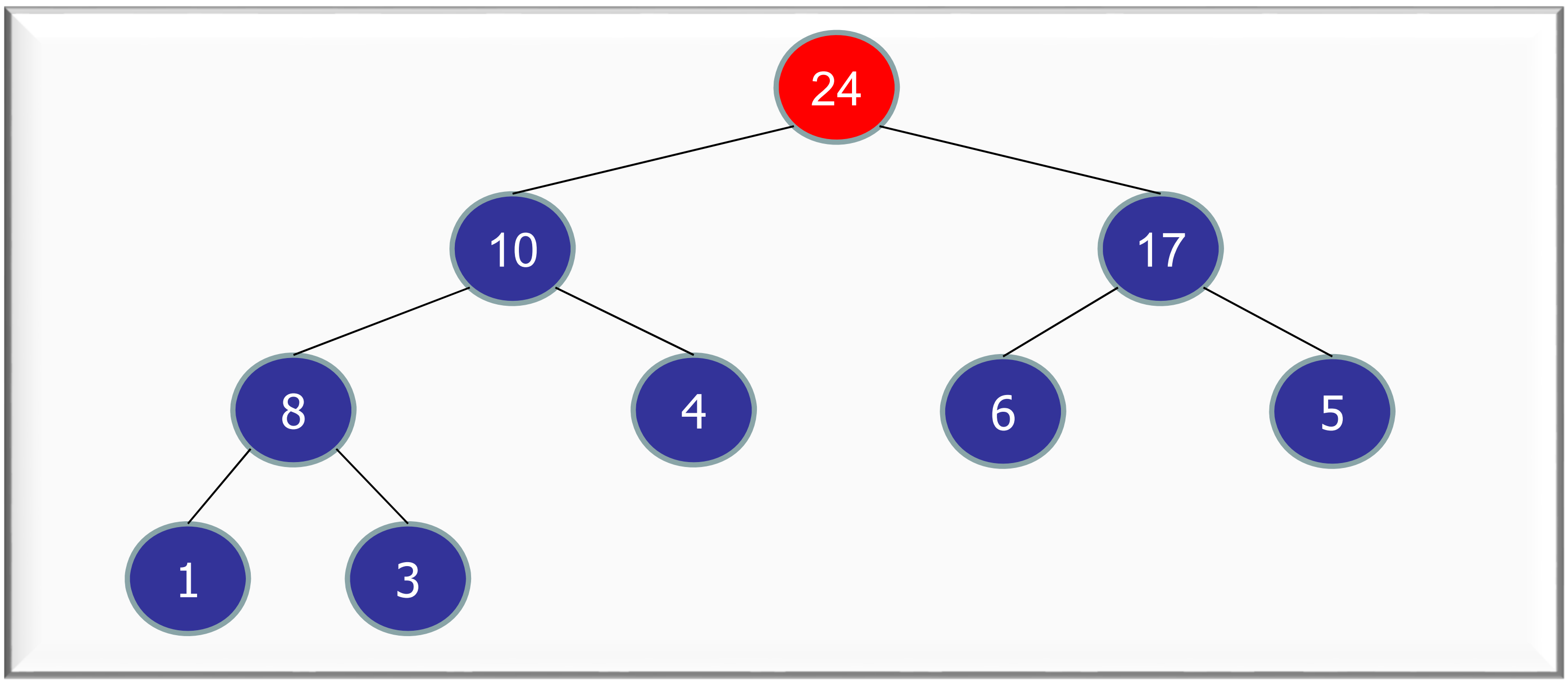
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	3



HeapSort

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

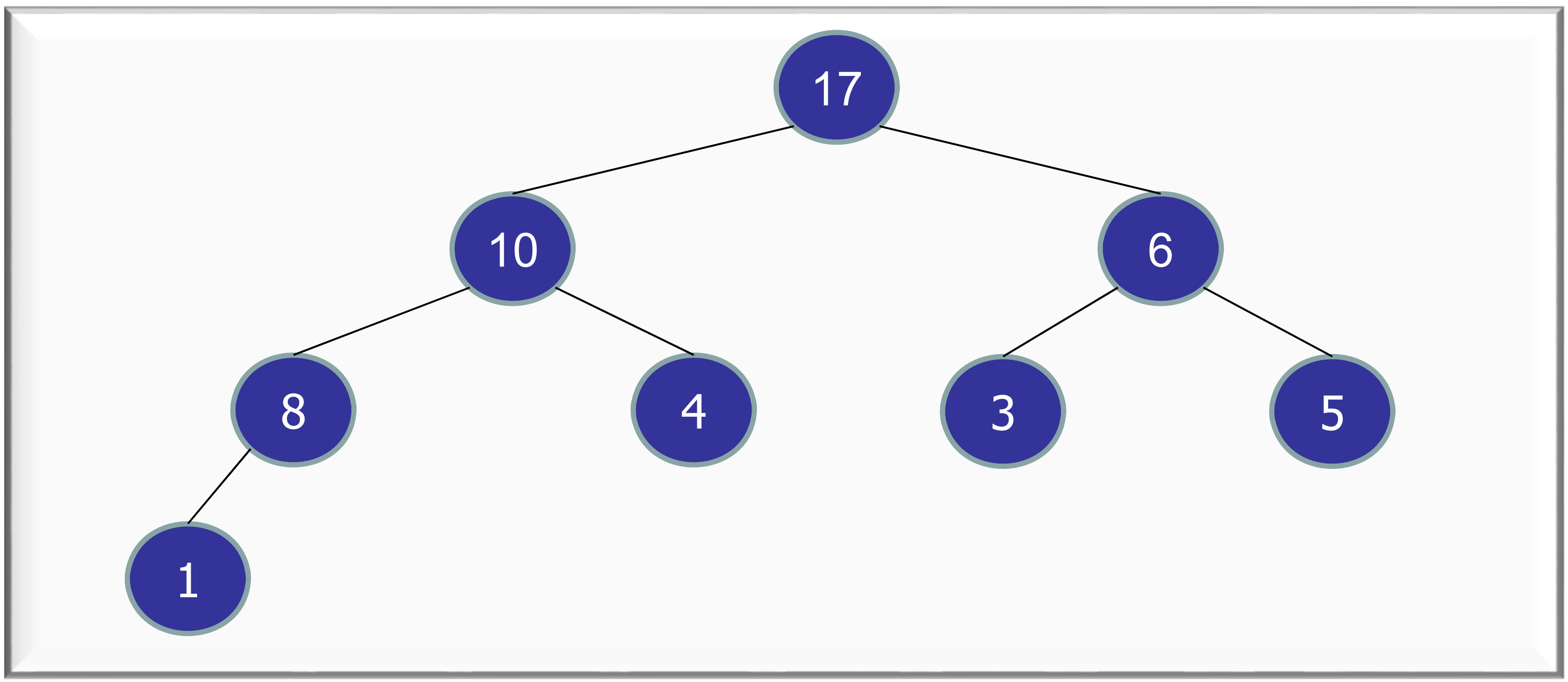
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	3



HeapSort

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

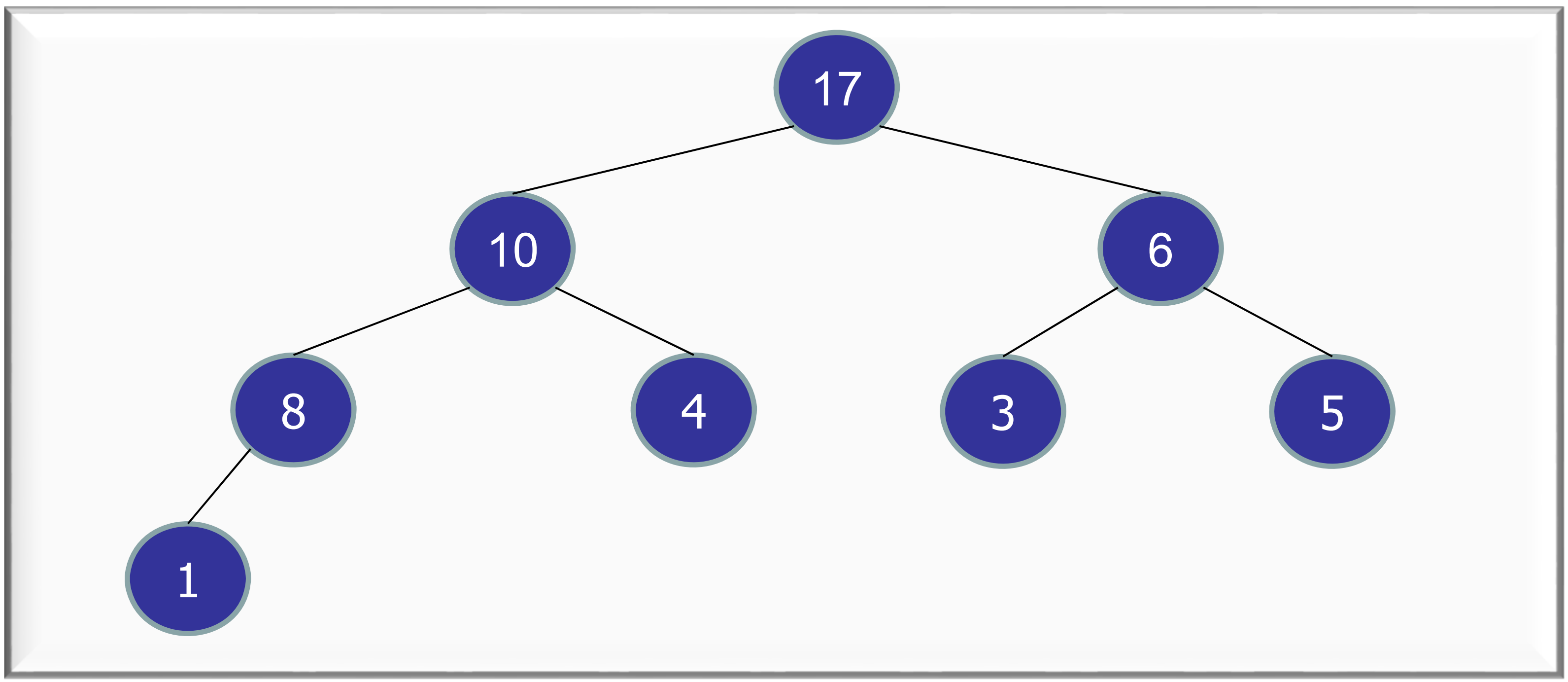
array slot	0	1	2	3	4	5	6	7	8
priority	17	10	6	8	4	3	5	1	



HeapSort

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

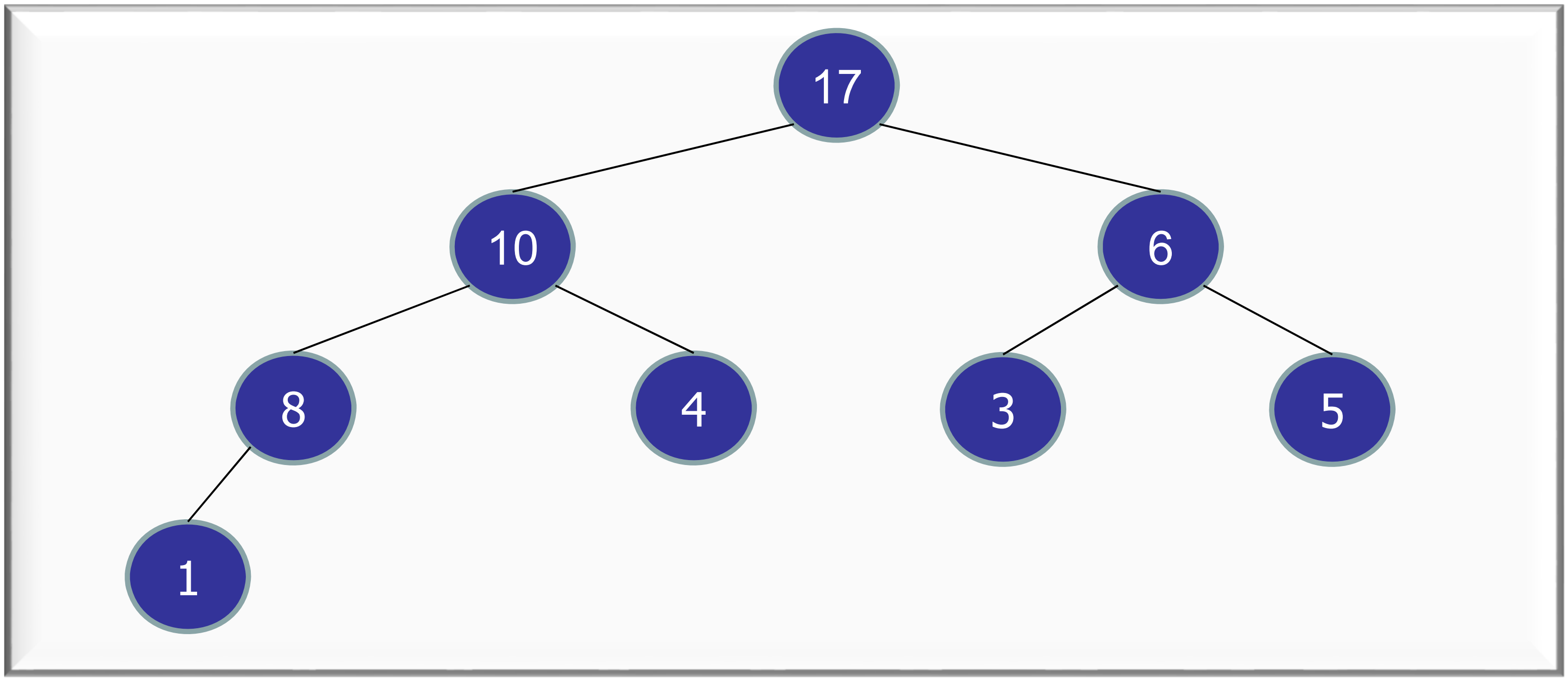
array slot	0	1	2	3	4	5	6	7	8
priority	17	10	6	8	4	3	5	1	24



HeapSort

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

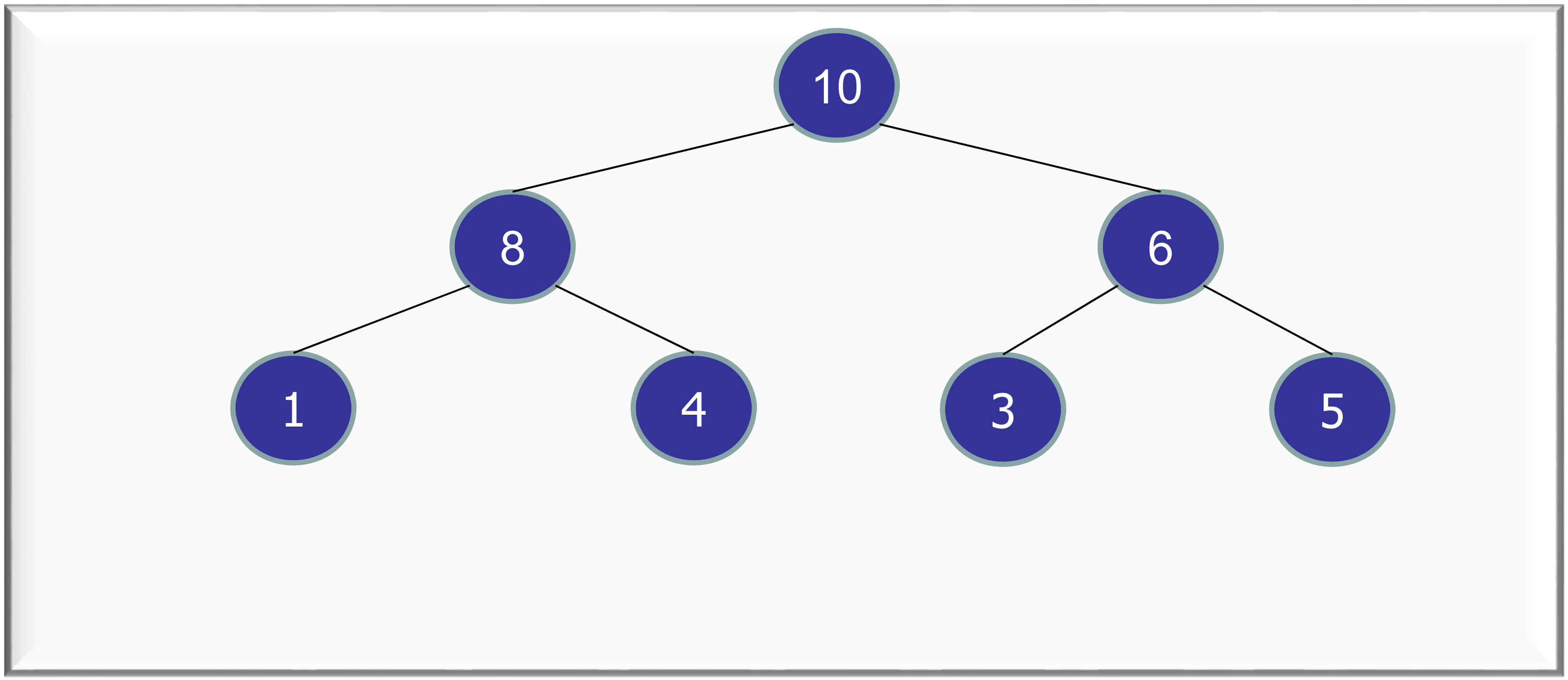
array slot	0	1	2	3	4	5	6	7	8
priority	17	10	6	8	4	3	5	1	24



HeapSort

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

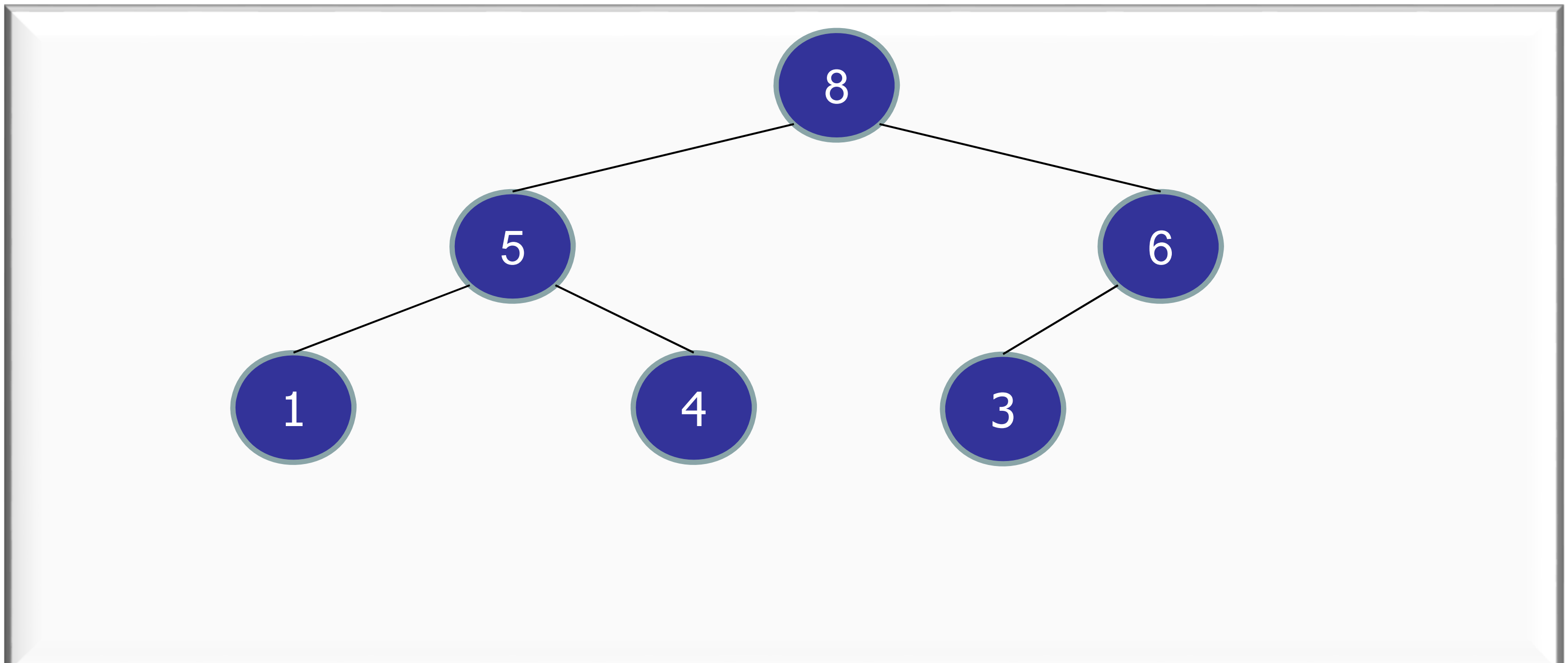
array slot	0	1	2	3	4	5	6	7	8
priority	10	8	6	1	4	3	5	17	24



HeapSort

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

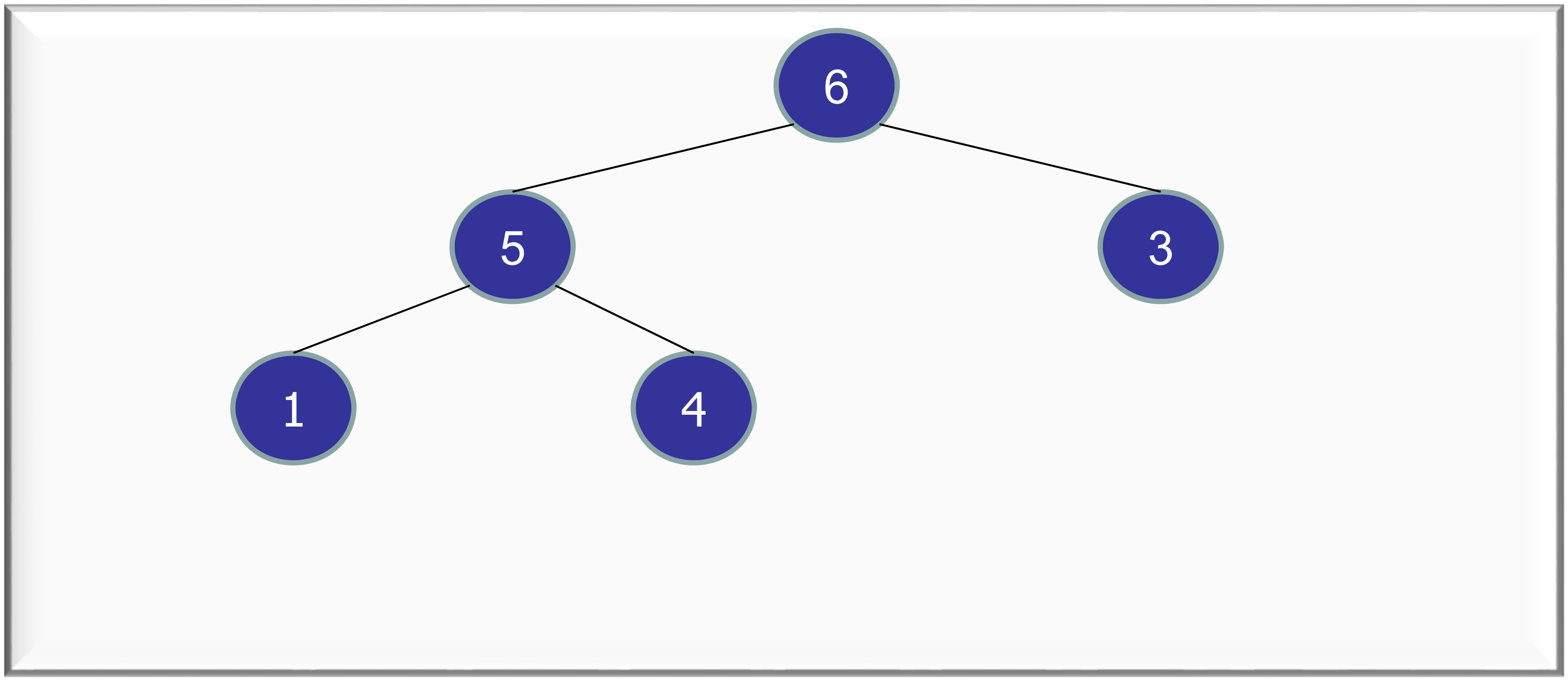
array slot	0	1	2	3	4	5	6	7	8
priority	8	5	6	1	4	3	10	17	24



HeapSort

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

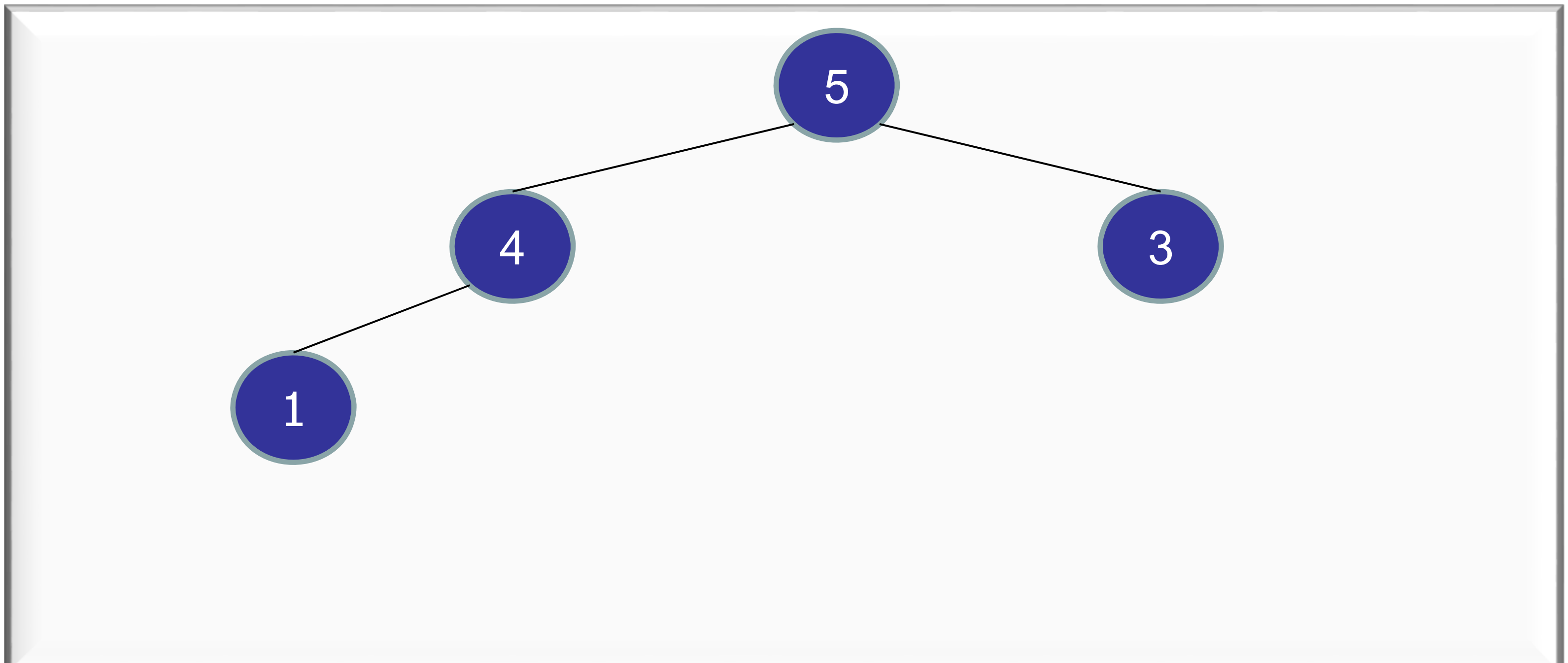
array slot	0	1	2	3	4	5	6	7	8
priority	6	5	3	1	4	8	10	17	24



HeapSort

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

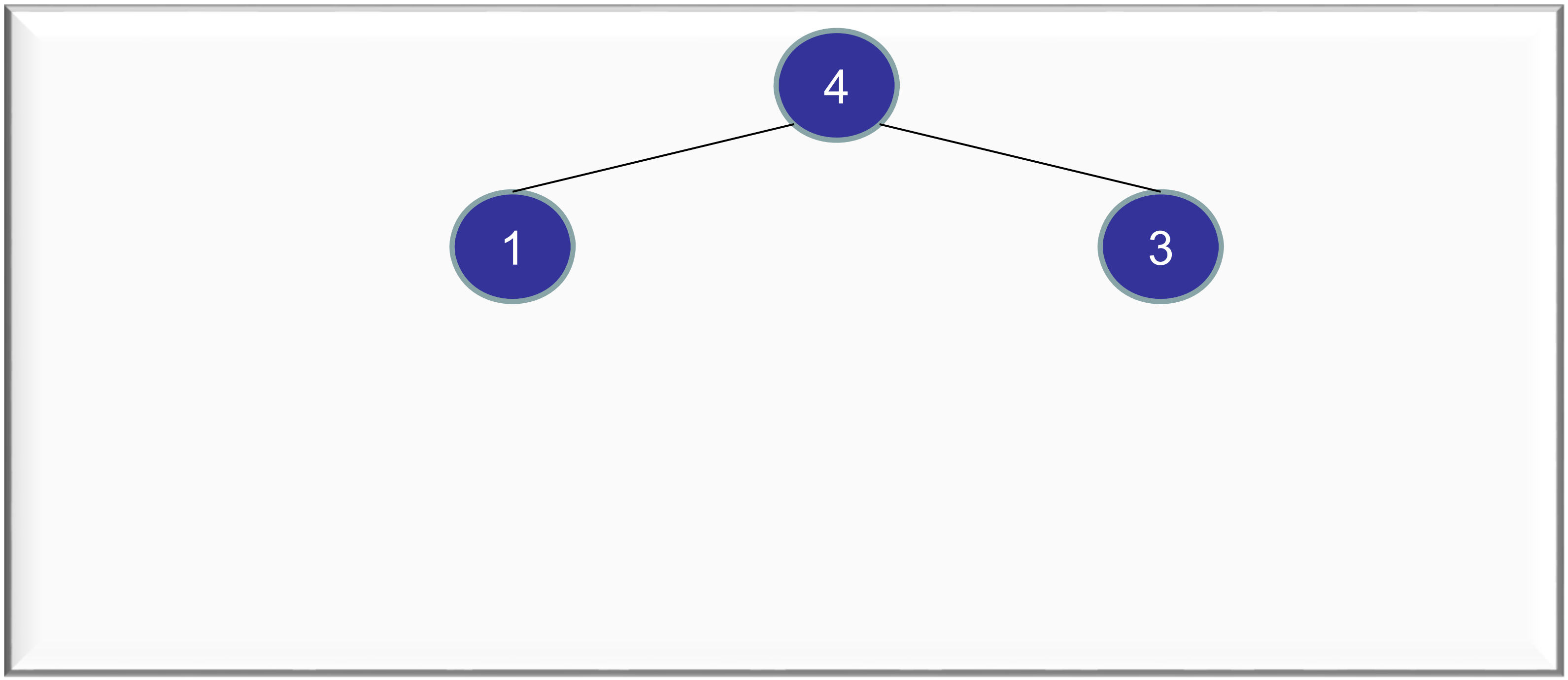
array slot	0	1	2	3	4	5	6	7	8
priority	5	4	3	1	6	8	10	17	24



HeapSort

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

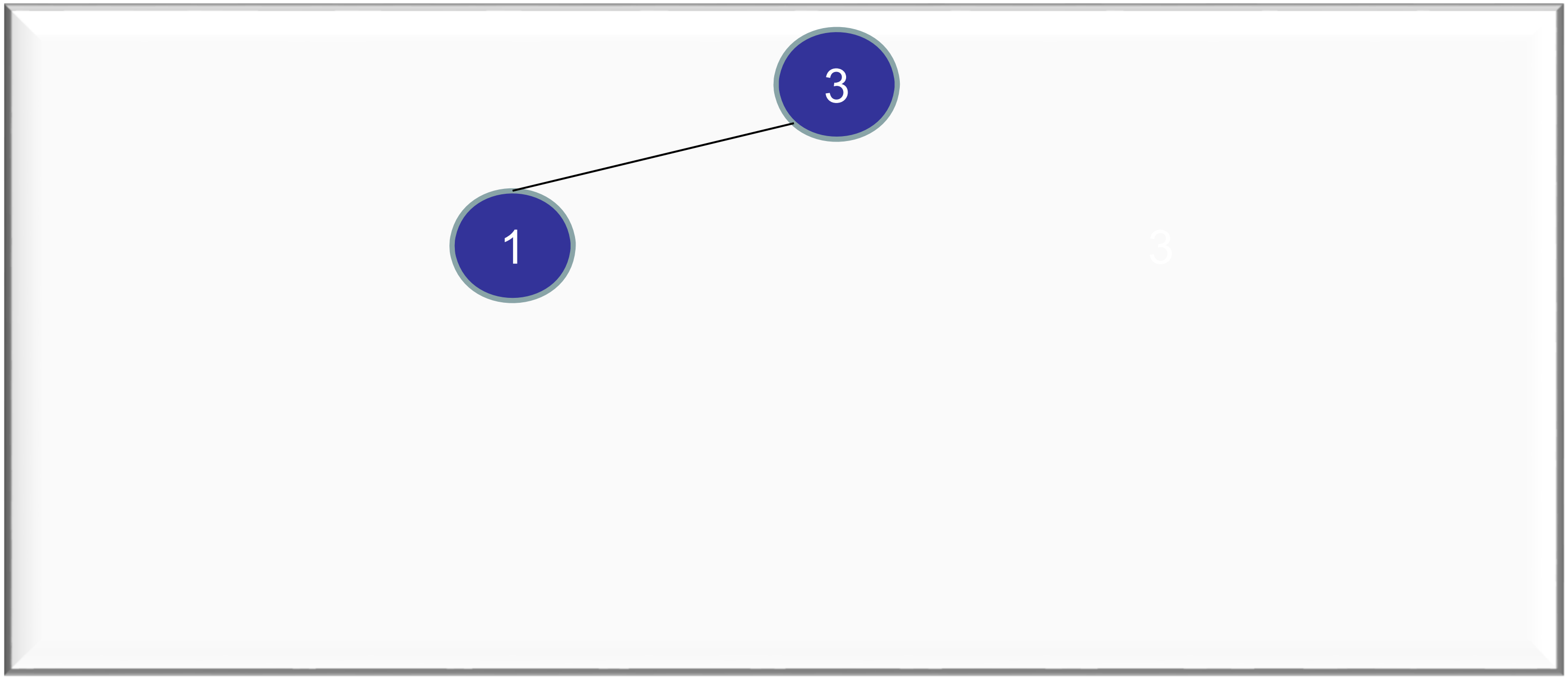
array slot	0	1	2	3	4	5	6	7	8
priority	4	1	3	5	6	8	10	17	24



HeapSort

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

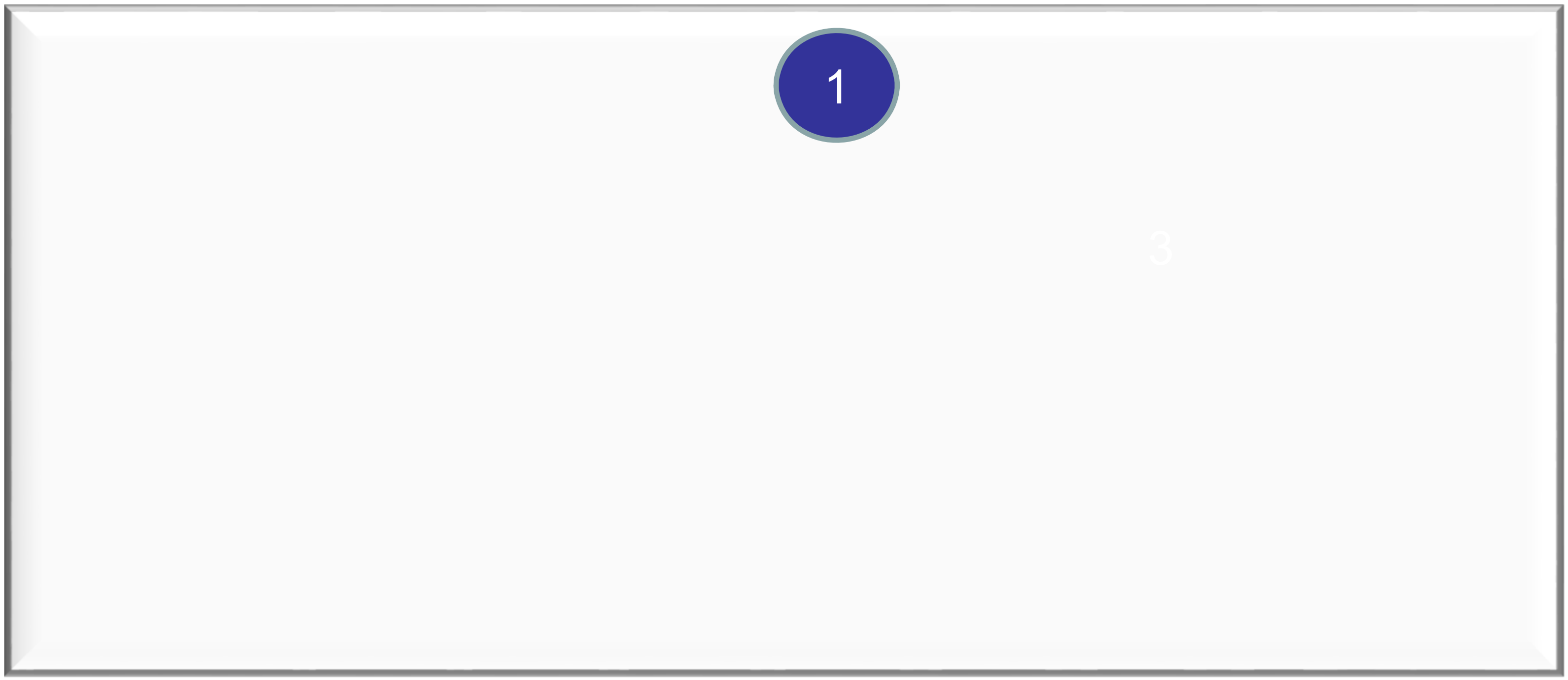
array slot	0	1	2	3	4	5	6	7	8
priority	3	1	4	5	6	8	10	17	24



HeapSort

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

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



HeapSort

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

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

HeapSort

Heap array ➔ Sorted list:

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

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

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

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

HeapSort

Heap array ➔ Sorted list: O(n log n)

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

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

HeapSort

Unsorted list:

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

Step 1. Unsorted list ➔ Heap

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

HeapSort

Heapify: Unsorted list → Heap:

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

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

HeapSort

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

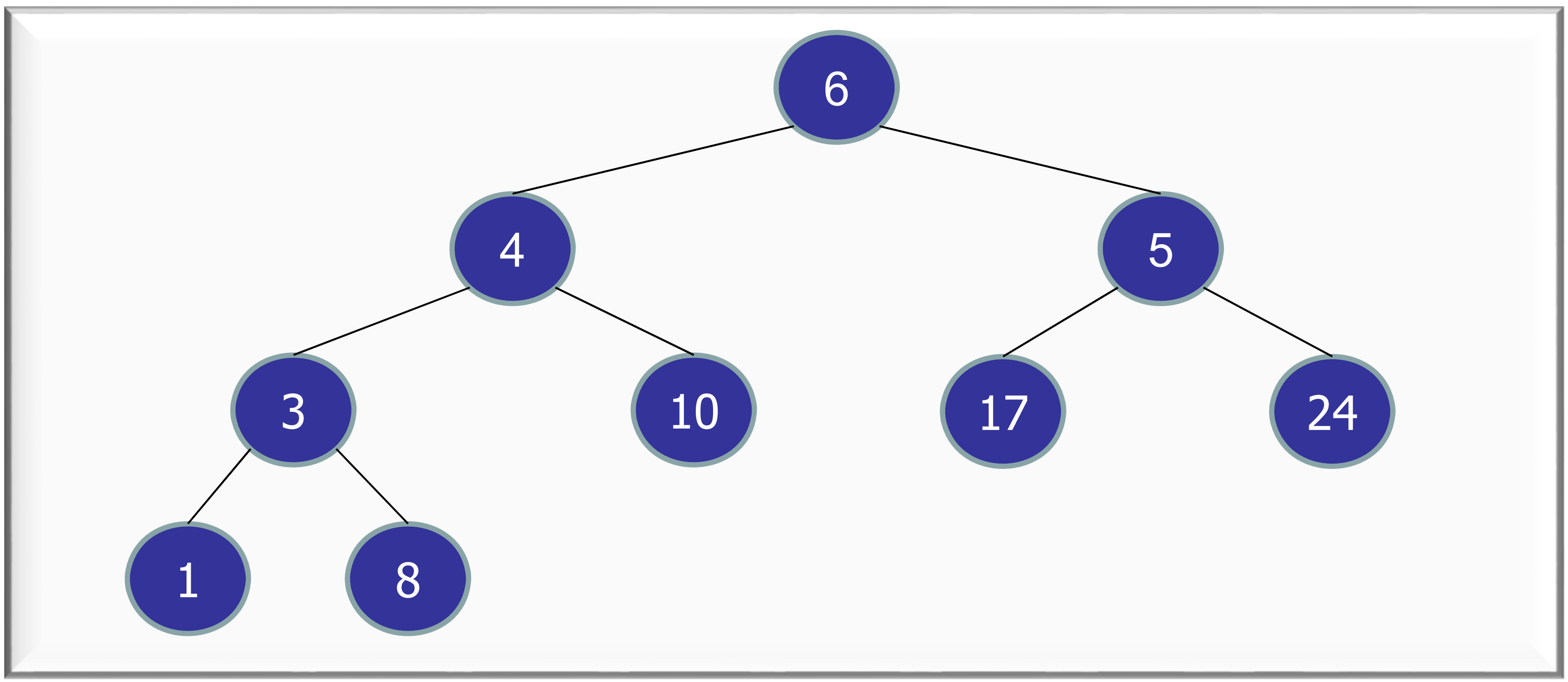
array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

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

HeapSort

Heapify v.2: Unsorted list → Heap

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

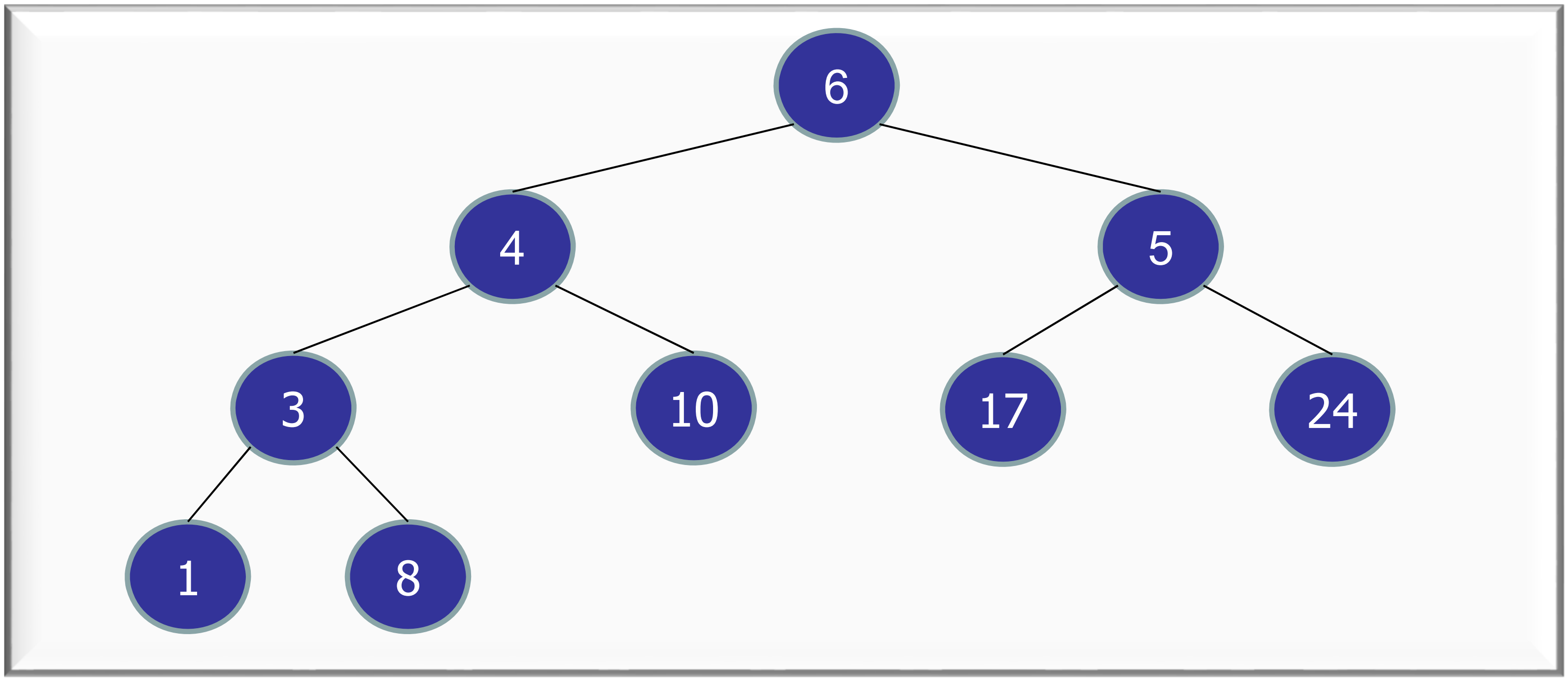


HeapSort

Idea: Recursion

Initially : Start with a complete tree.

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

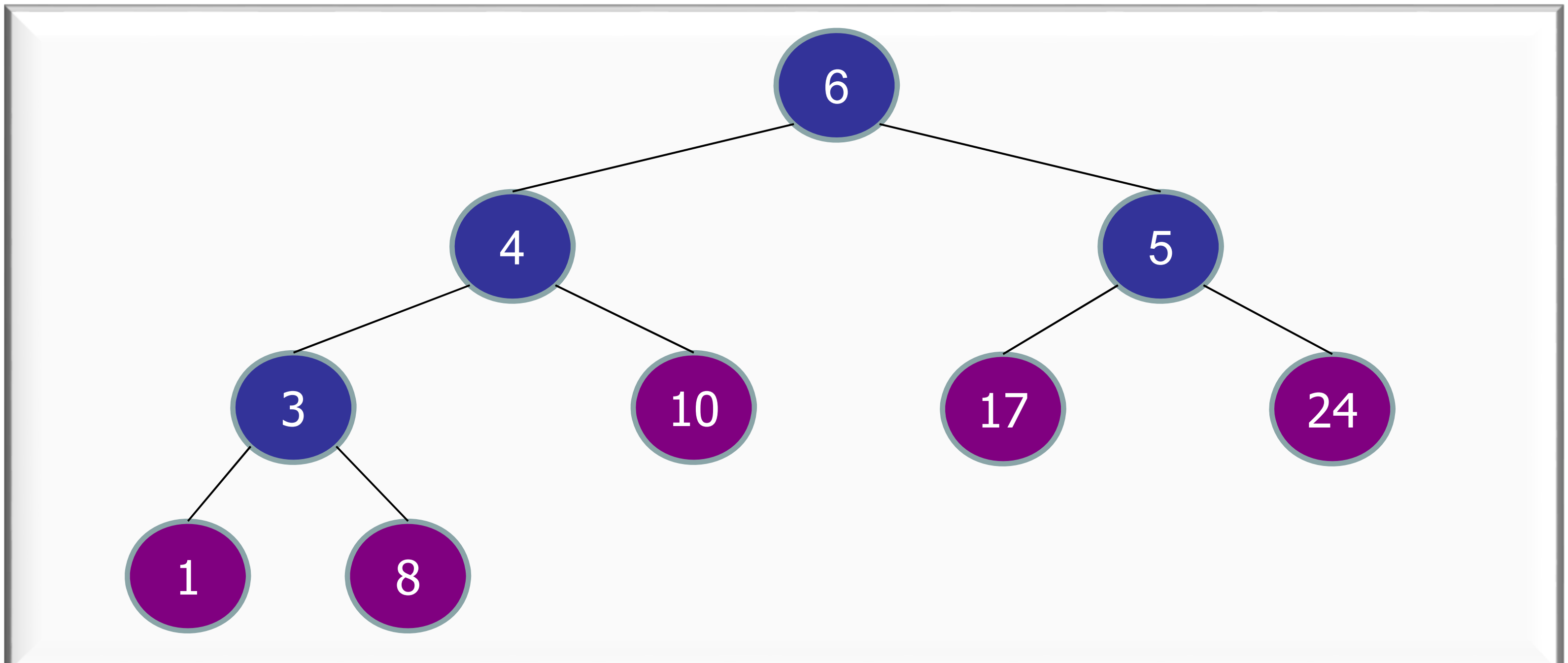


HeapSort

Idea: Recursion

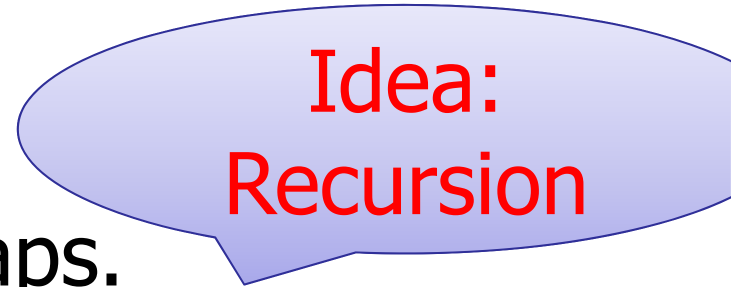
Base case: each leaf is a heap.

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

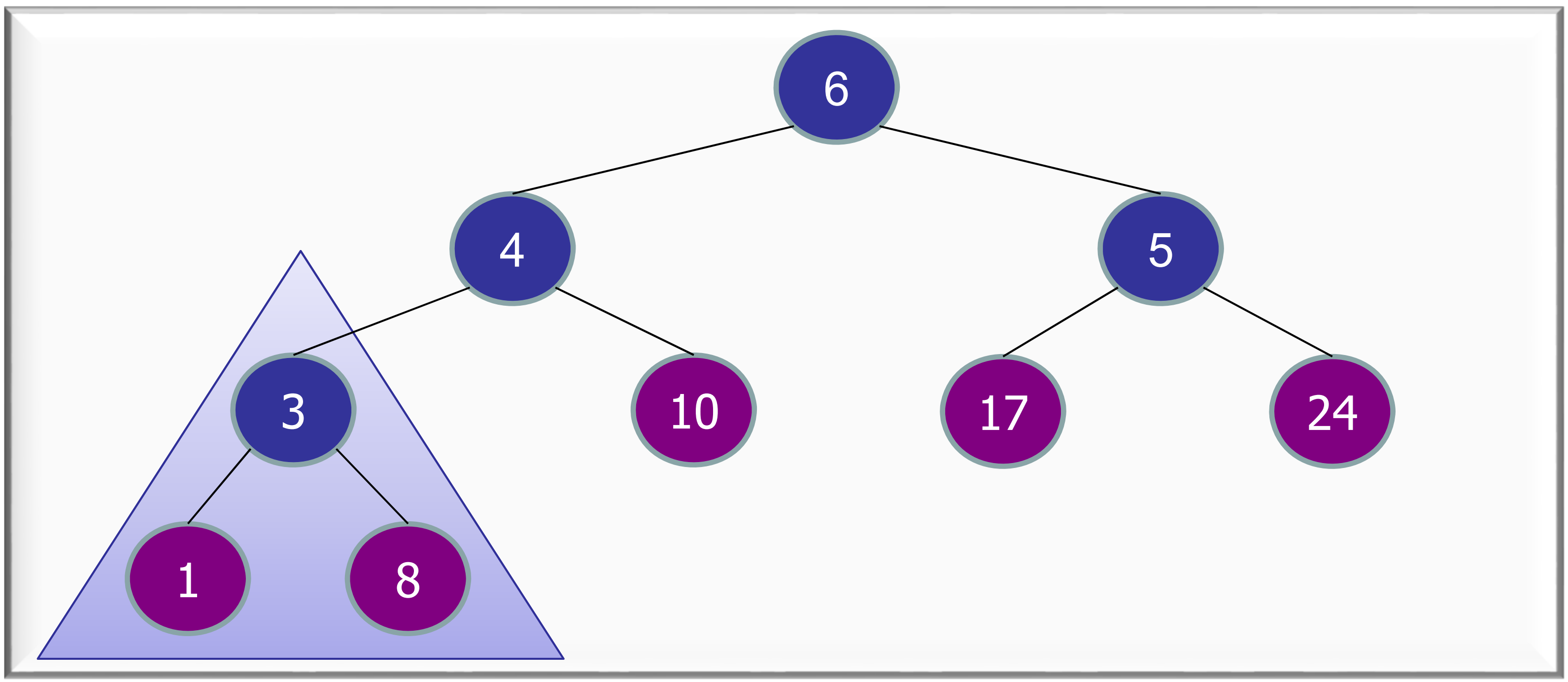


HeapSort

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

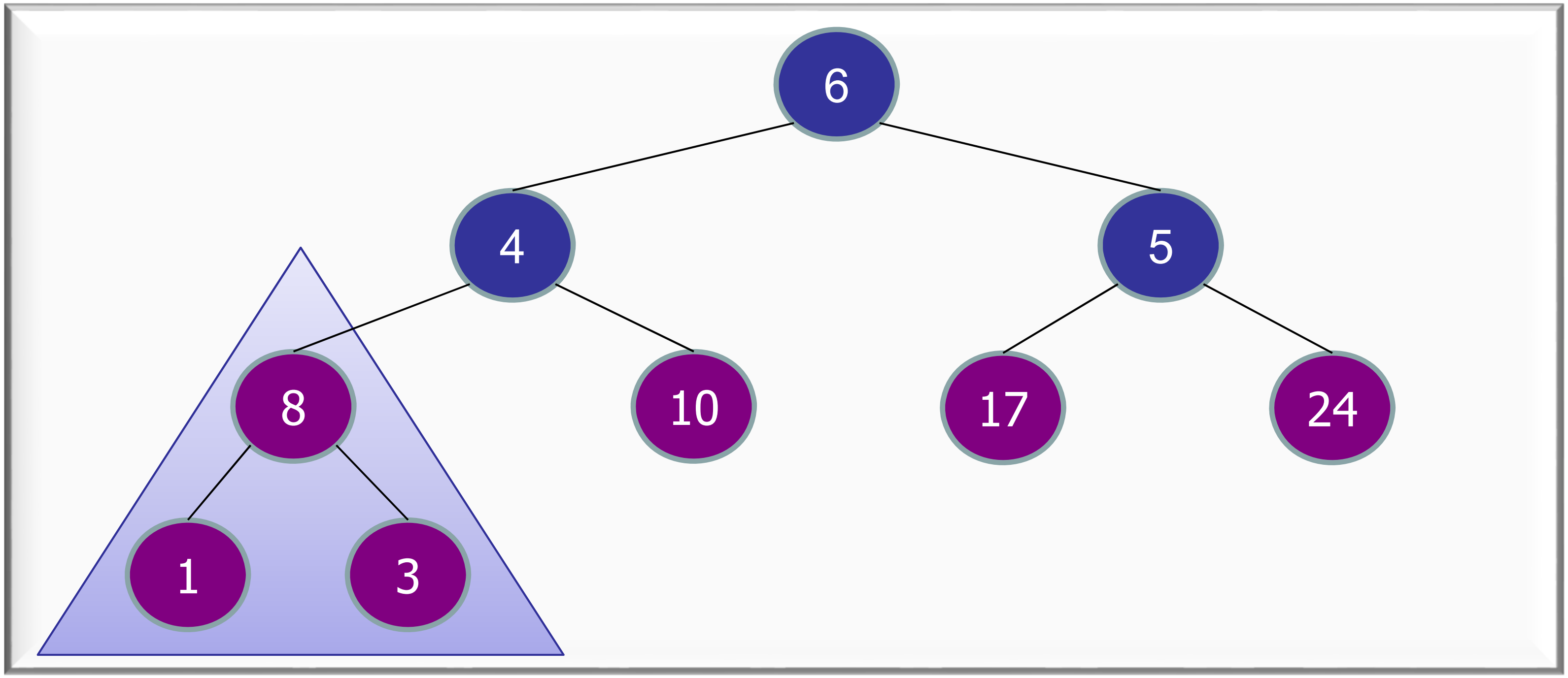


HeapSort

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

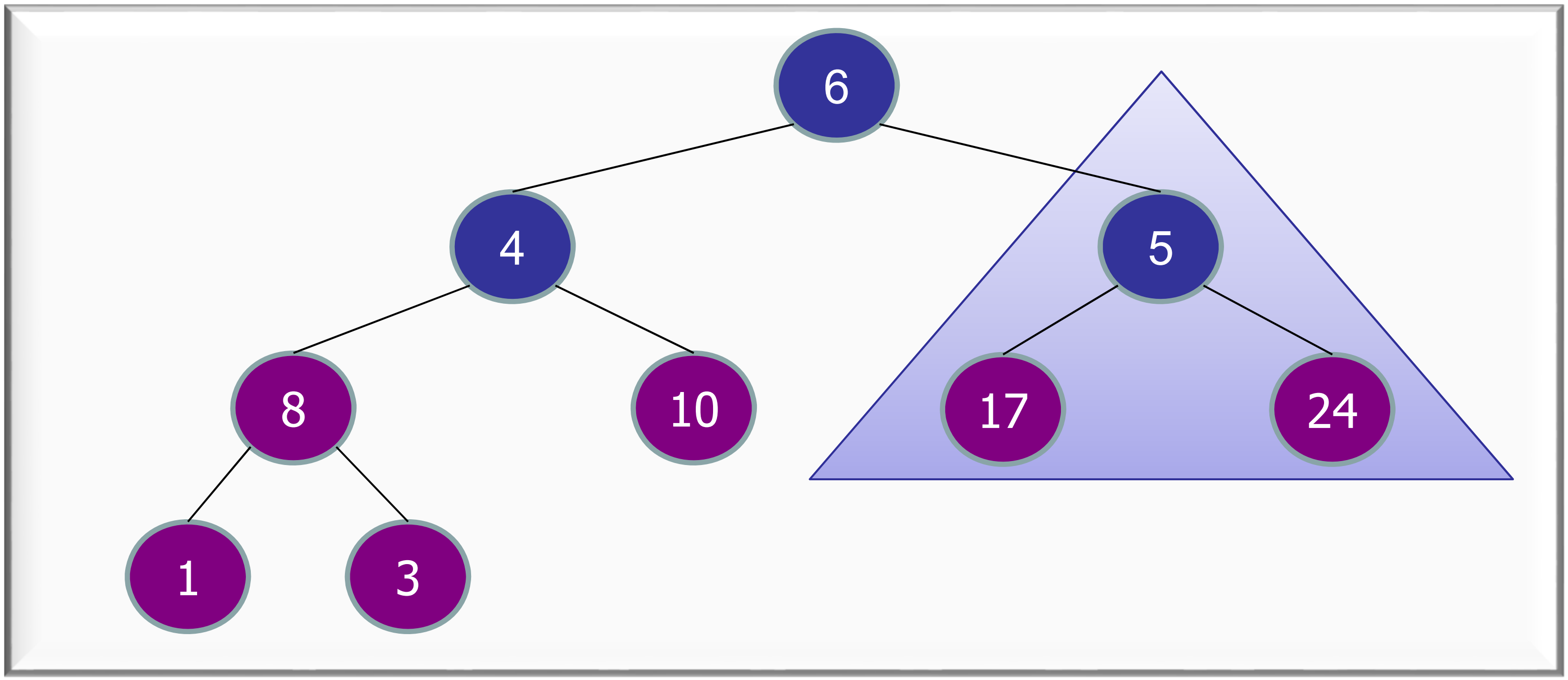


HeapSort

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

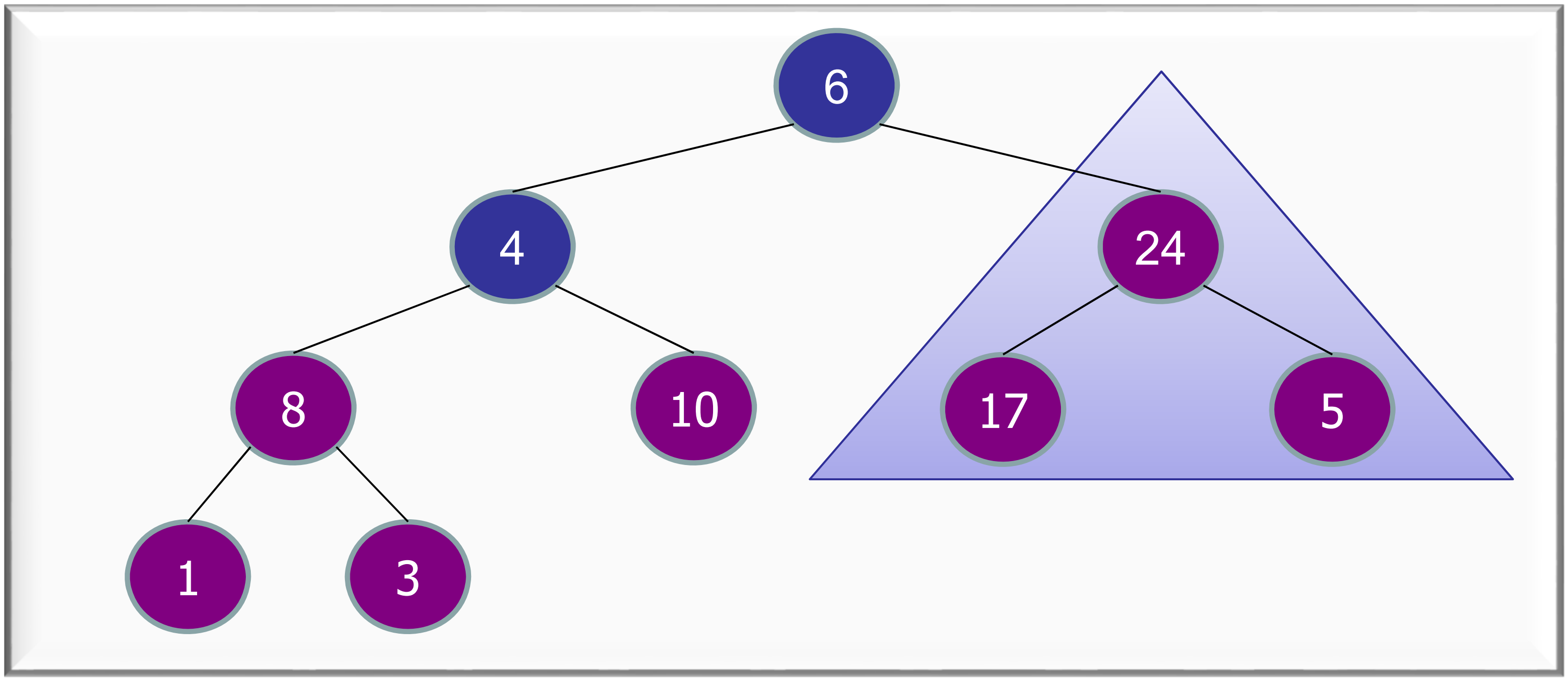


HeapSort

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

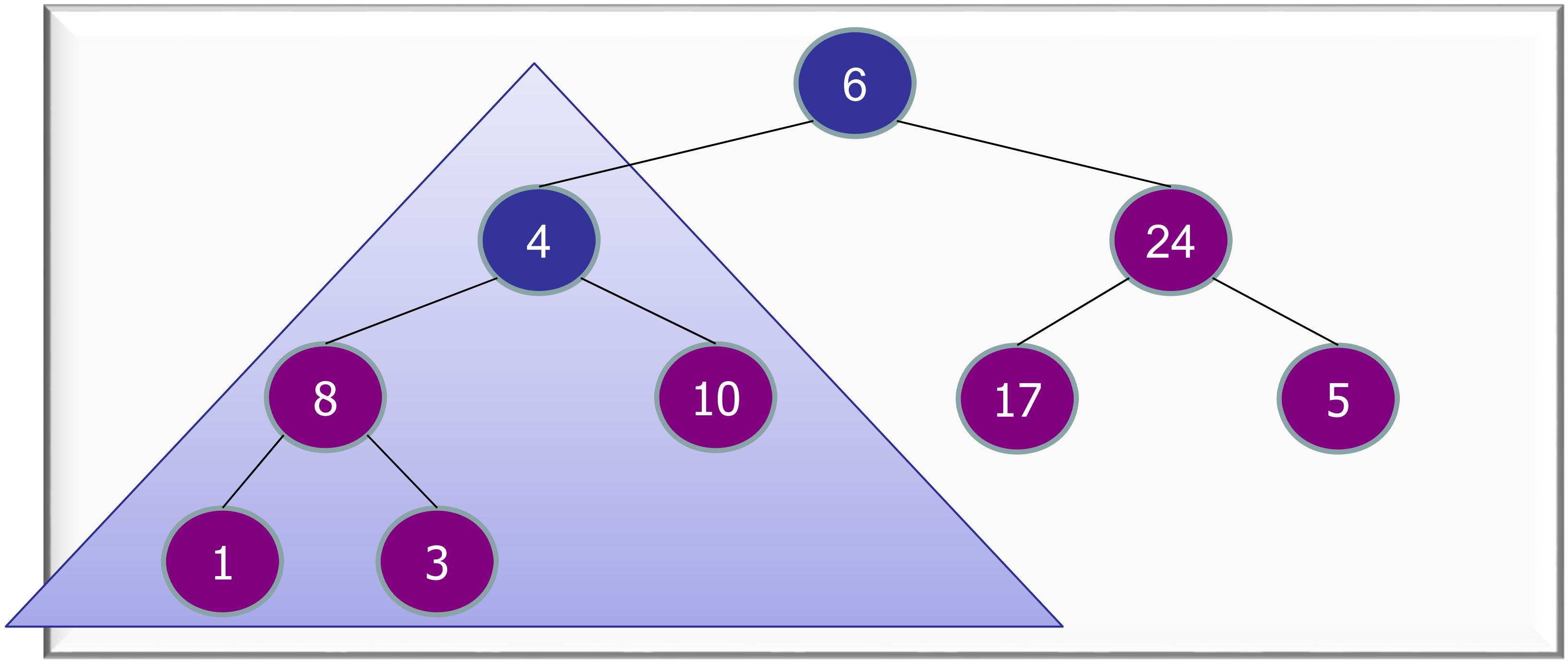


HeapSort

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

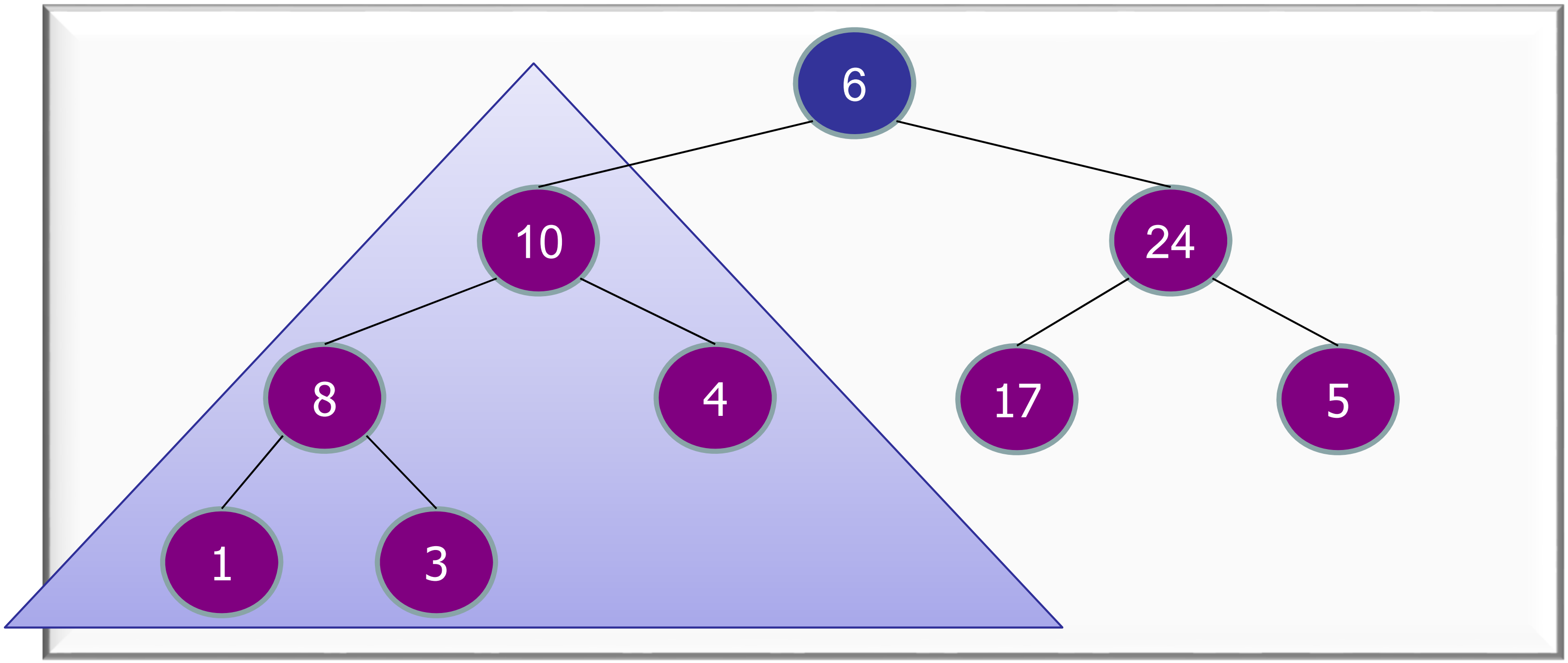


HeapSort

Idea: Recursion

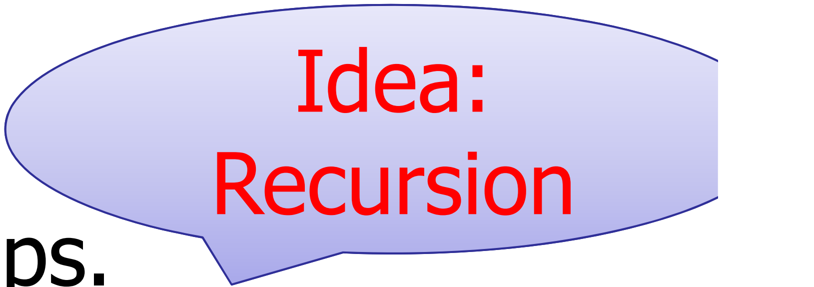
Recursion: left + right are heaps.

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

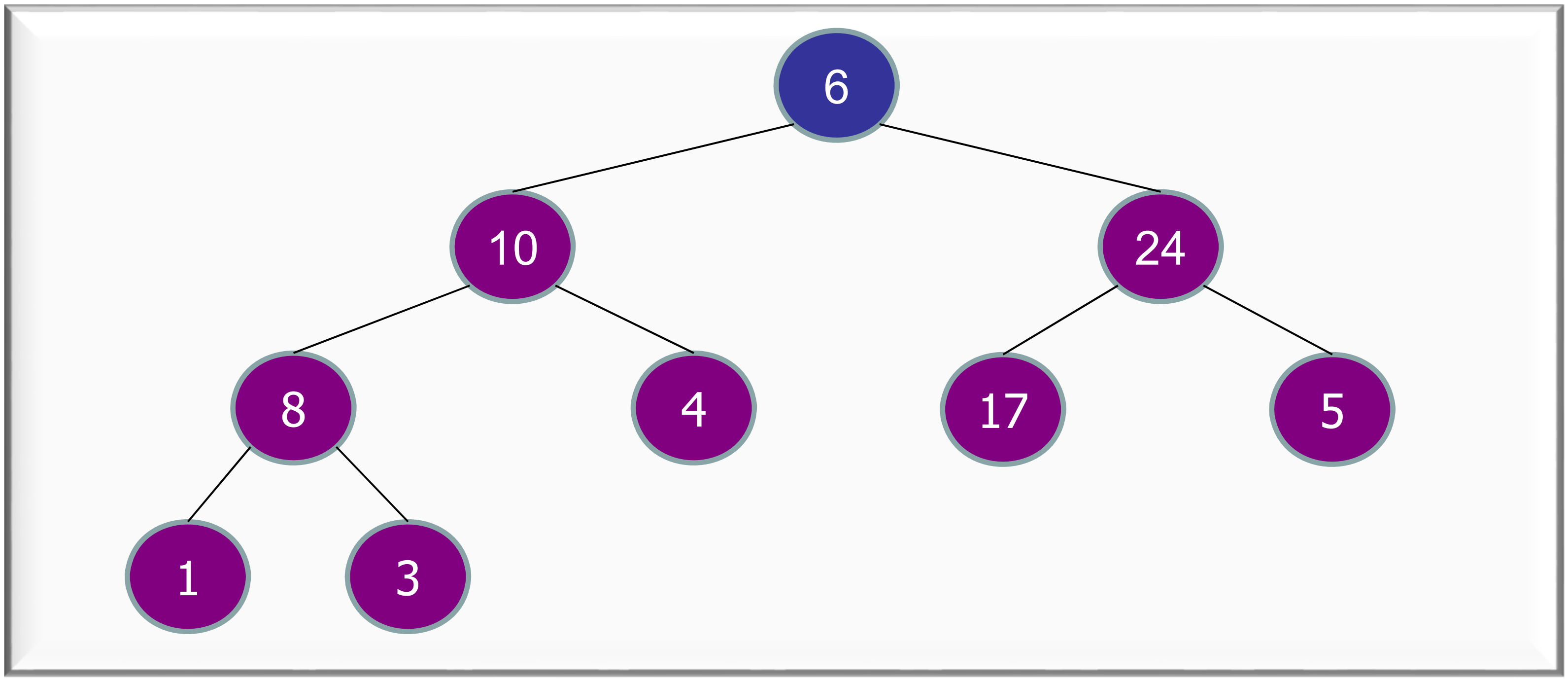


HeapSort

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**

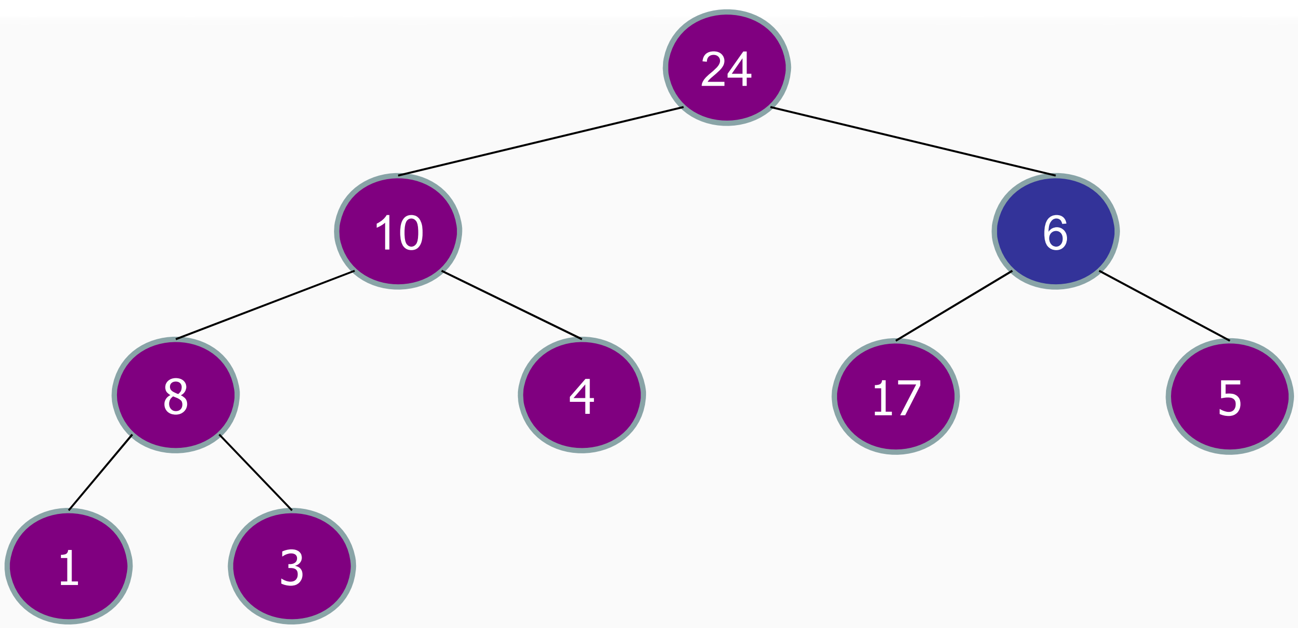


HeapSort

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

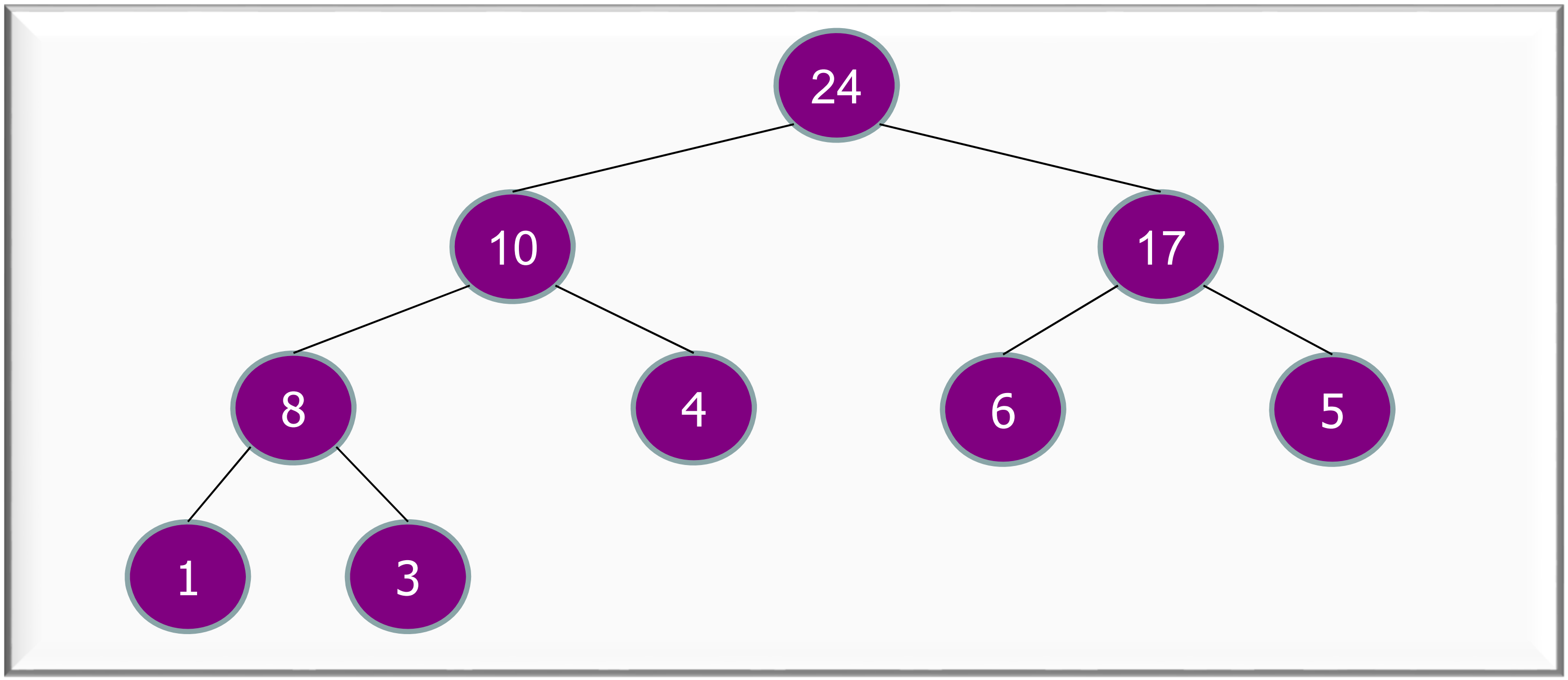


HeapSort

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

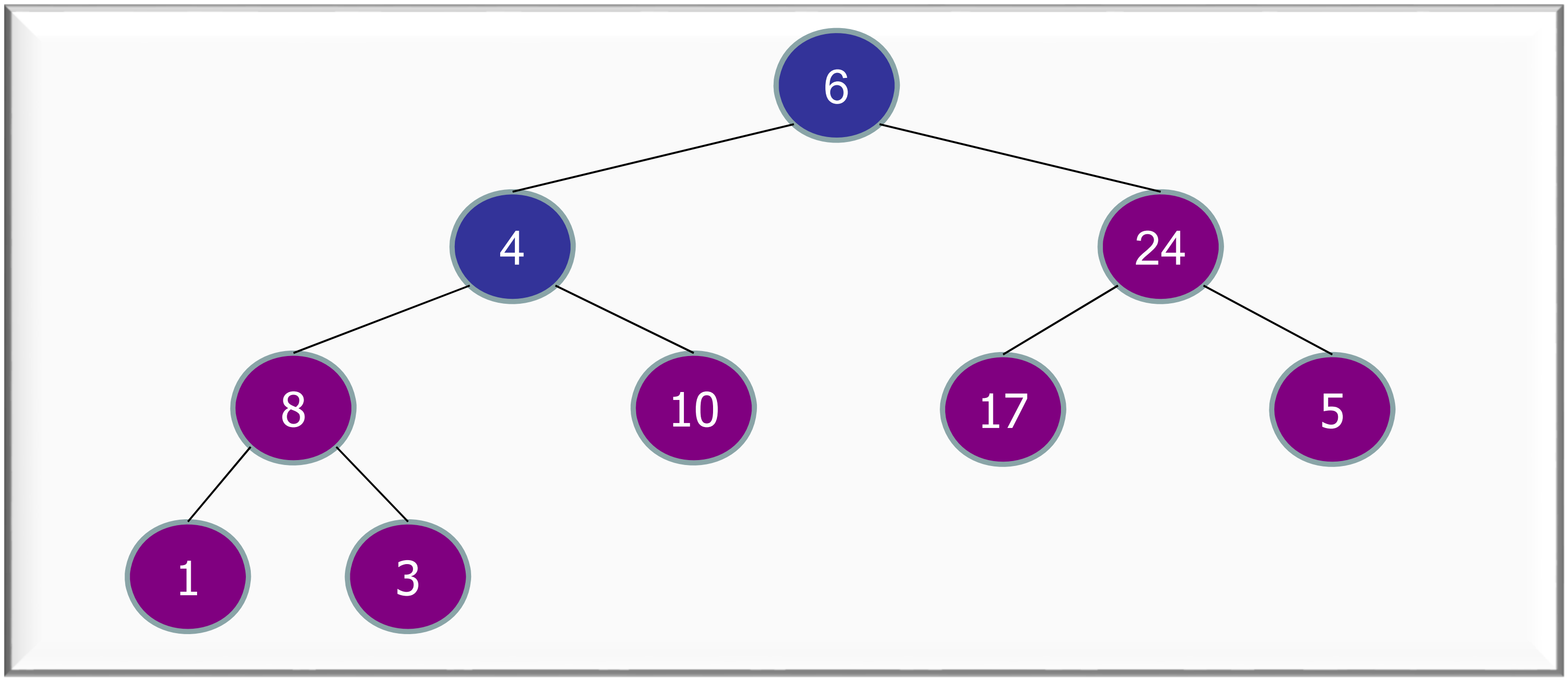
array slot	0	1	2	3	4	5	6	7	8
key	24	10	17	8	4	6	5	1	3

```
// 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

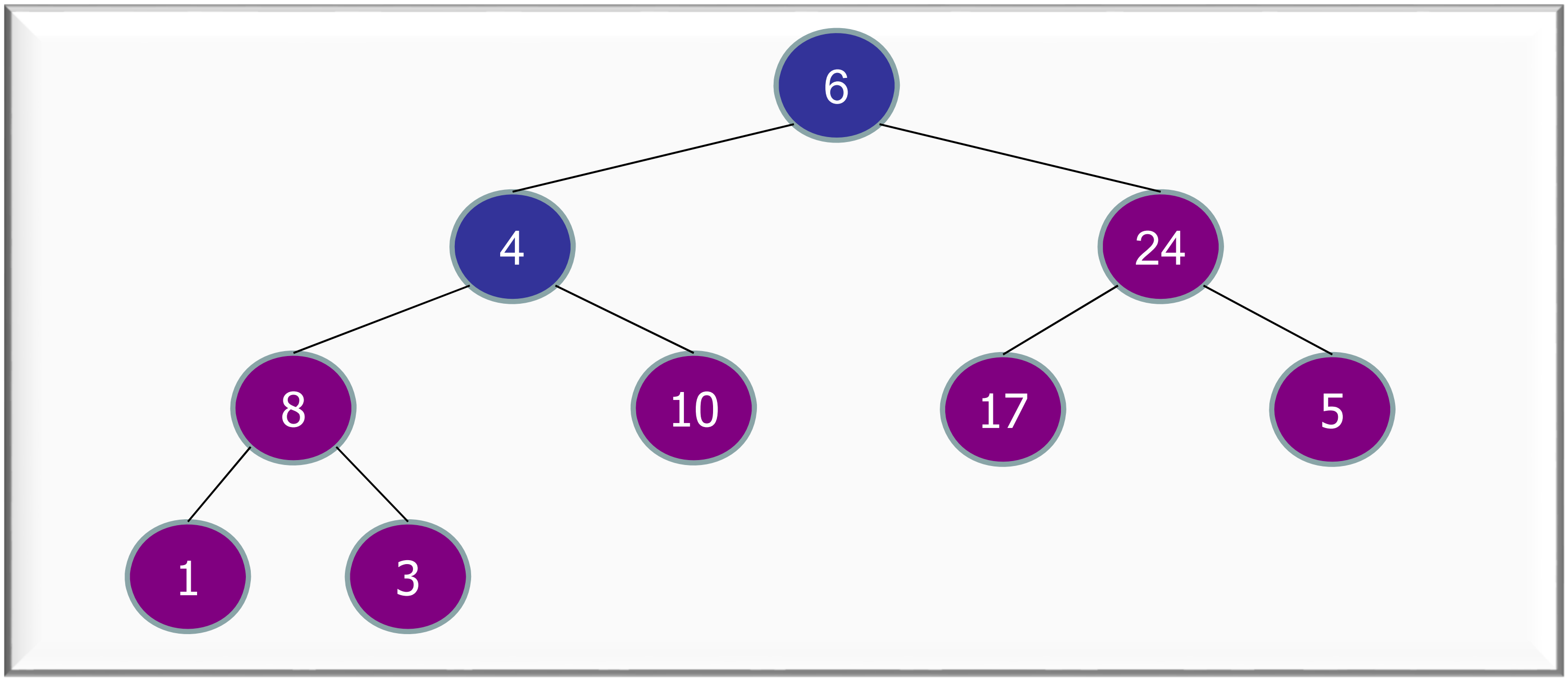
array slot	0	1	2	3	4	5	6	7	8
key	6	4	24	8	10	17	5	1	3



HeapSort

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

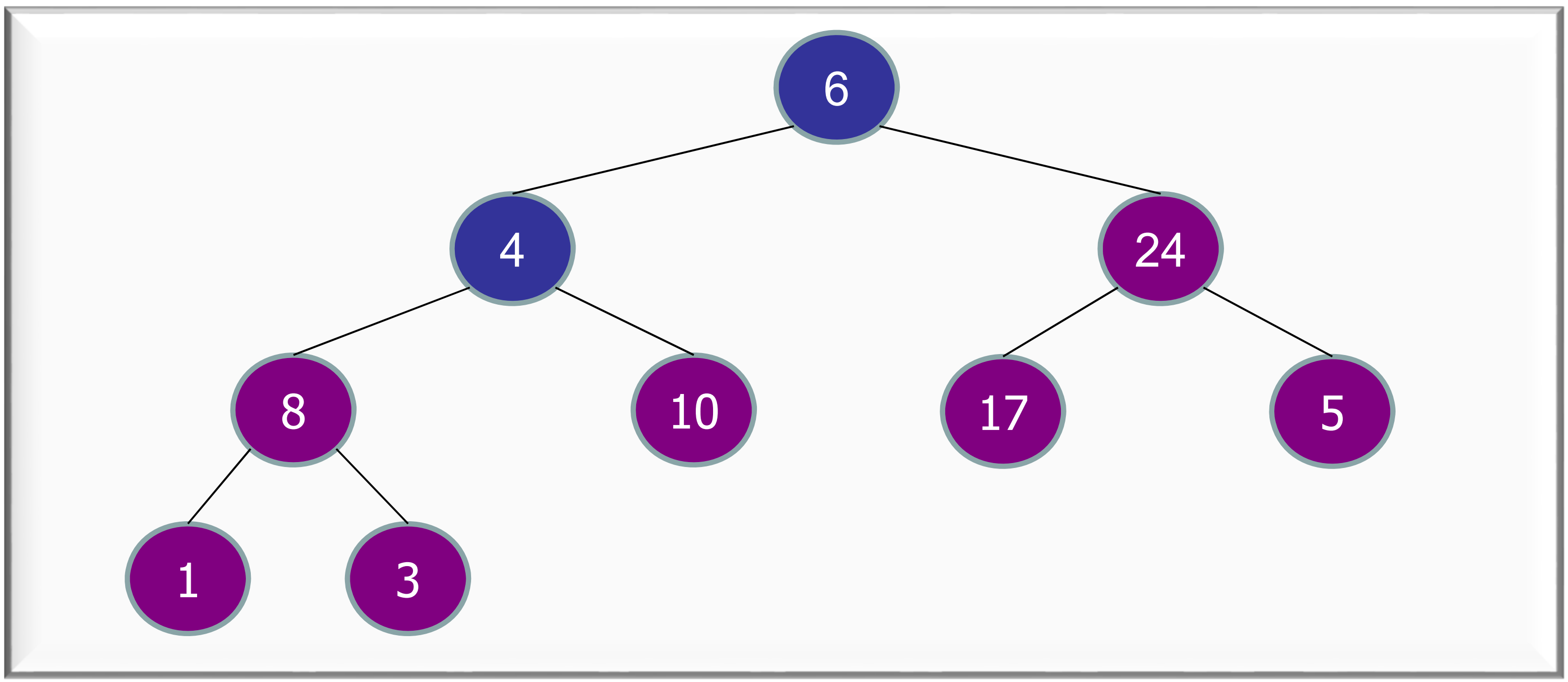
array slot	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
key	**6**	**4**	**24**	**8**	**10**	**17**	**5**	**1**	**3**



HeapSort

Observation: most nodes have small height!

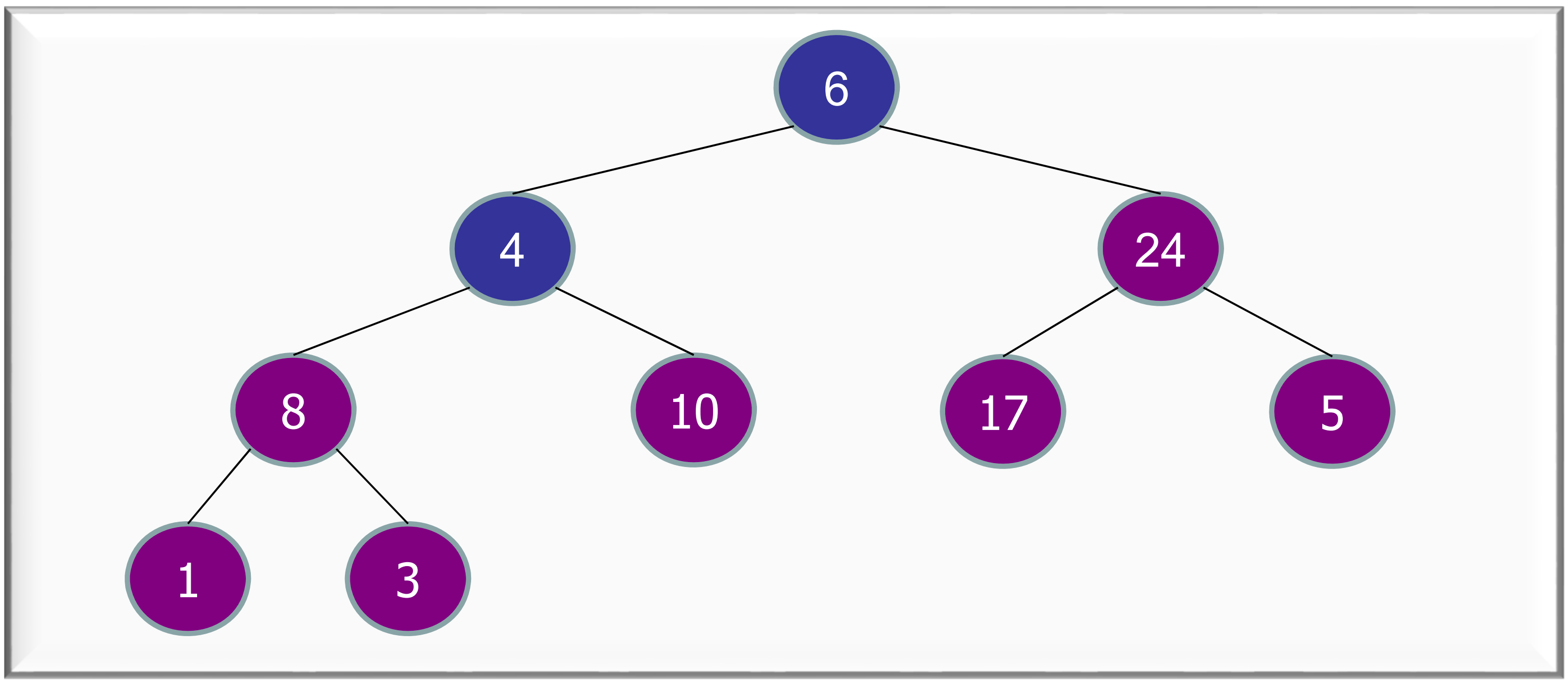
array slot	0	1	2	3	4	5	6	7	8
key	6	4	24	8	10	17	5	1	3



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

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

cost for bubbling a node at level h

upper bound on number of nodes at level h

HeapSort

Cost of building a heap:

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

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

$$\le cn\left(\frac{½}{(1 - ½)^2}\right) \le 2 \cdot O(n)$$

HeapSort

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

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

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

HeapSort

Unsorted list:

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

Step 1. 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

Step 2. 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: only need n space!
- 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.

Intermission:

Part I: Implementing a Priority Queue

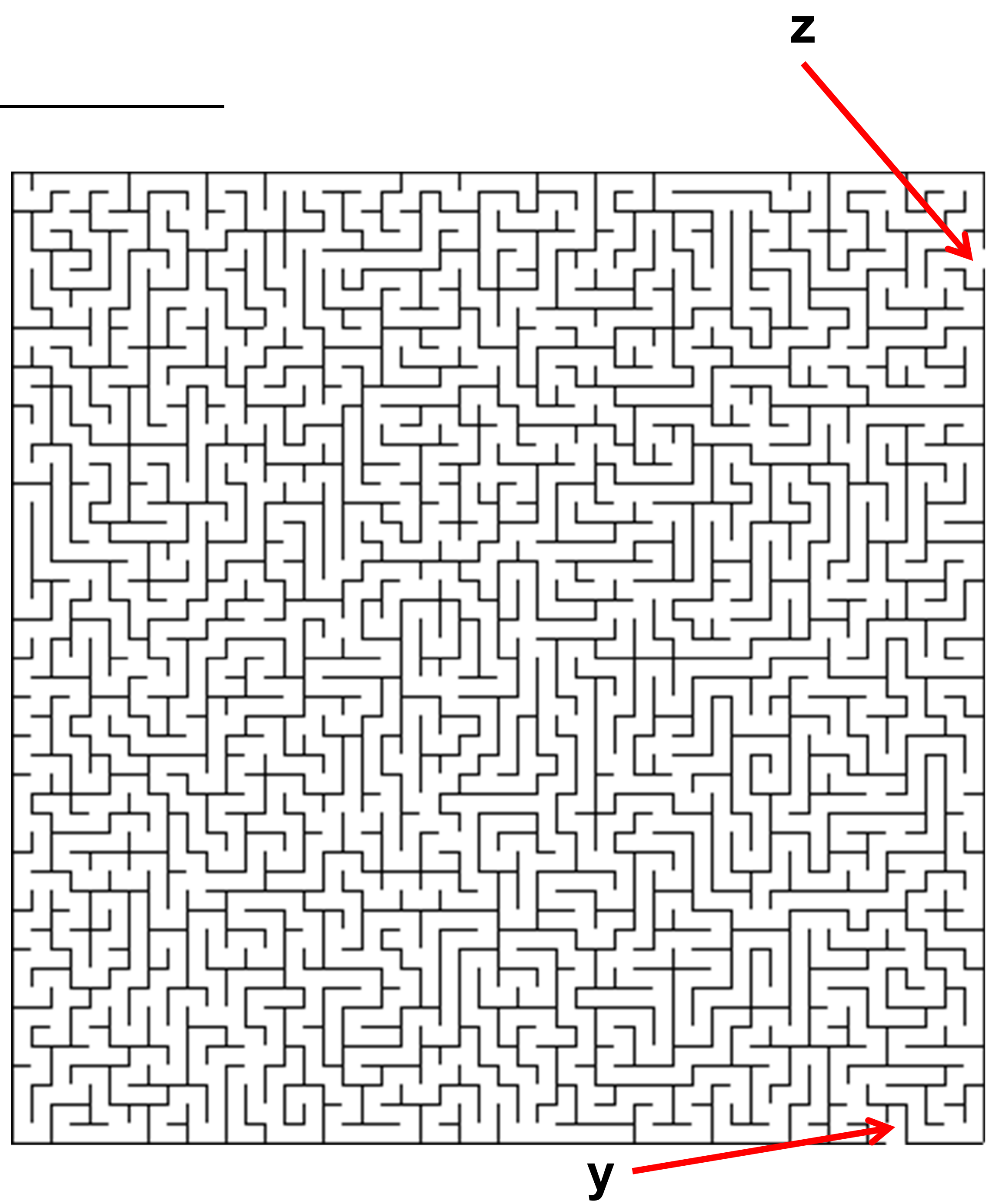
- Binary Heaps
- HeapSort

Part II: Disjoint Set

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

Mazes

Is there any route from y to z?

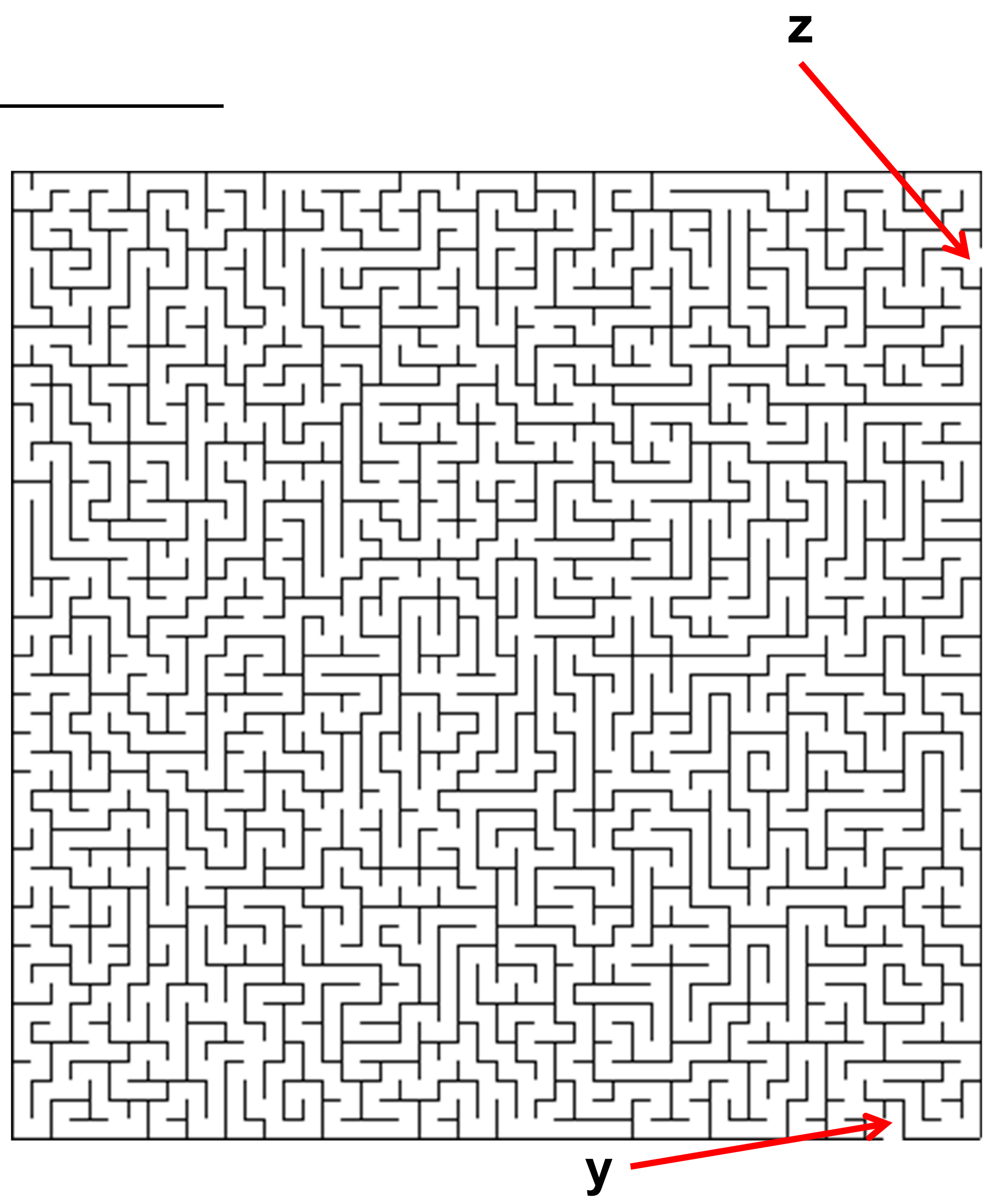


Mazes

Is there any route from y to z?

Either BFS or DFS takes time:

O(E + V)



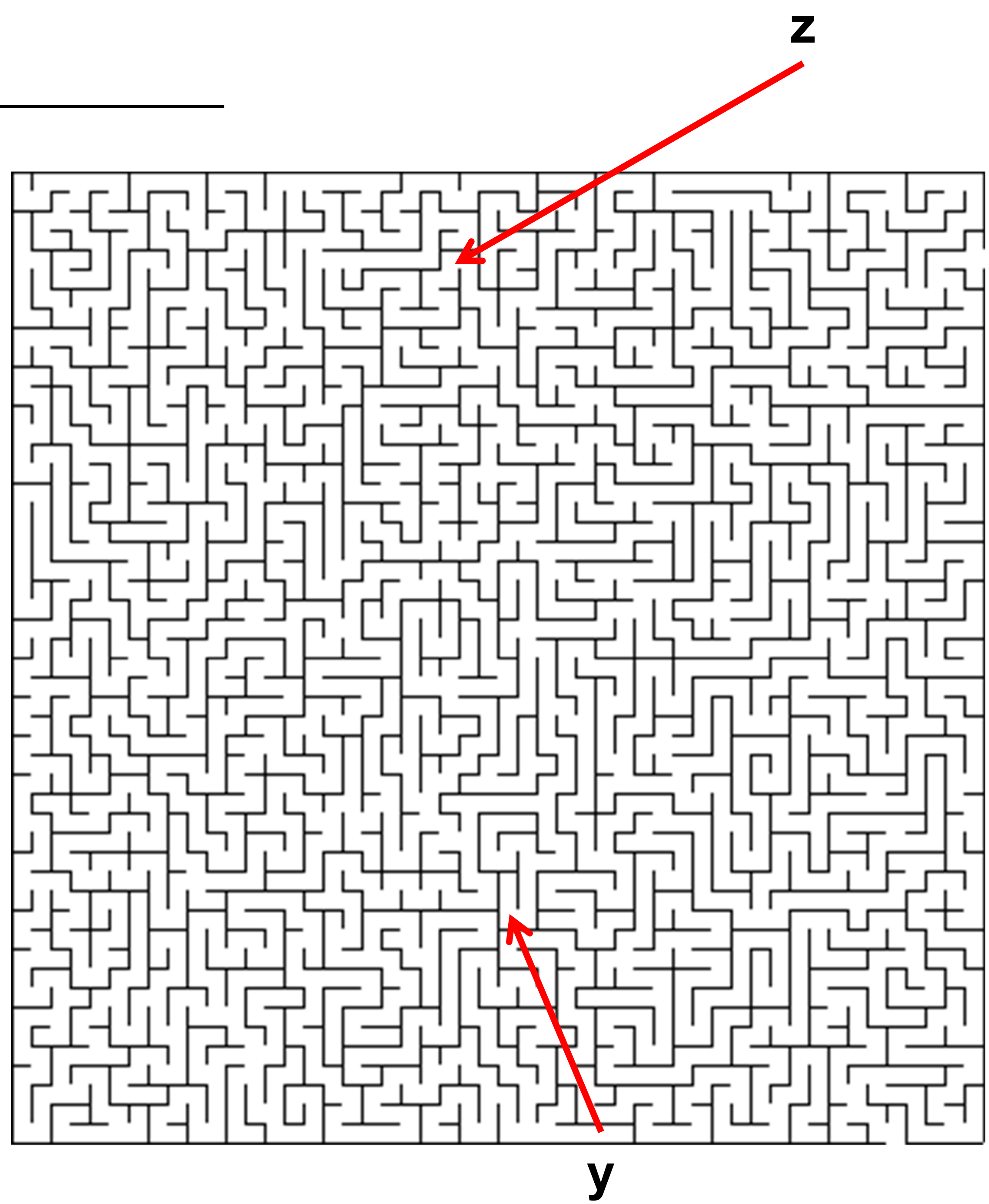
Mazes

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.



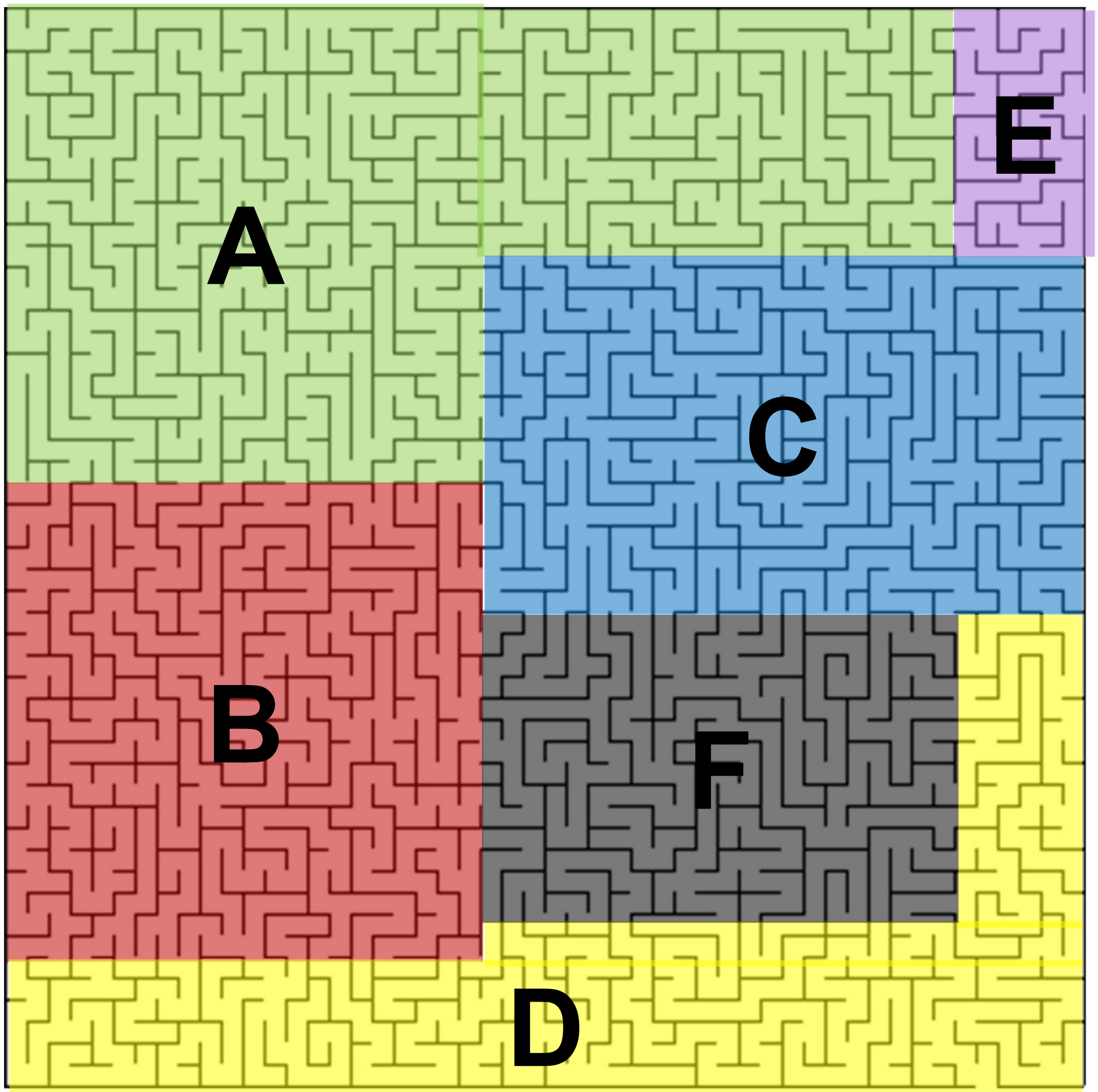
Mazes

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.



Dynamic Mazes

Preprocess:

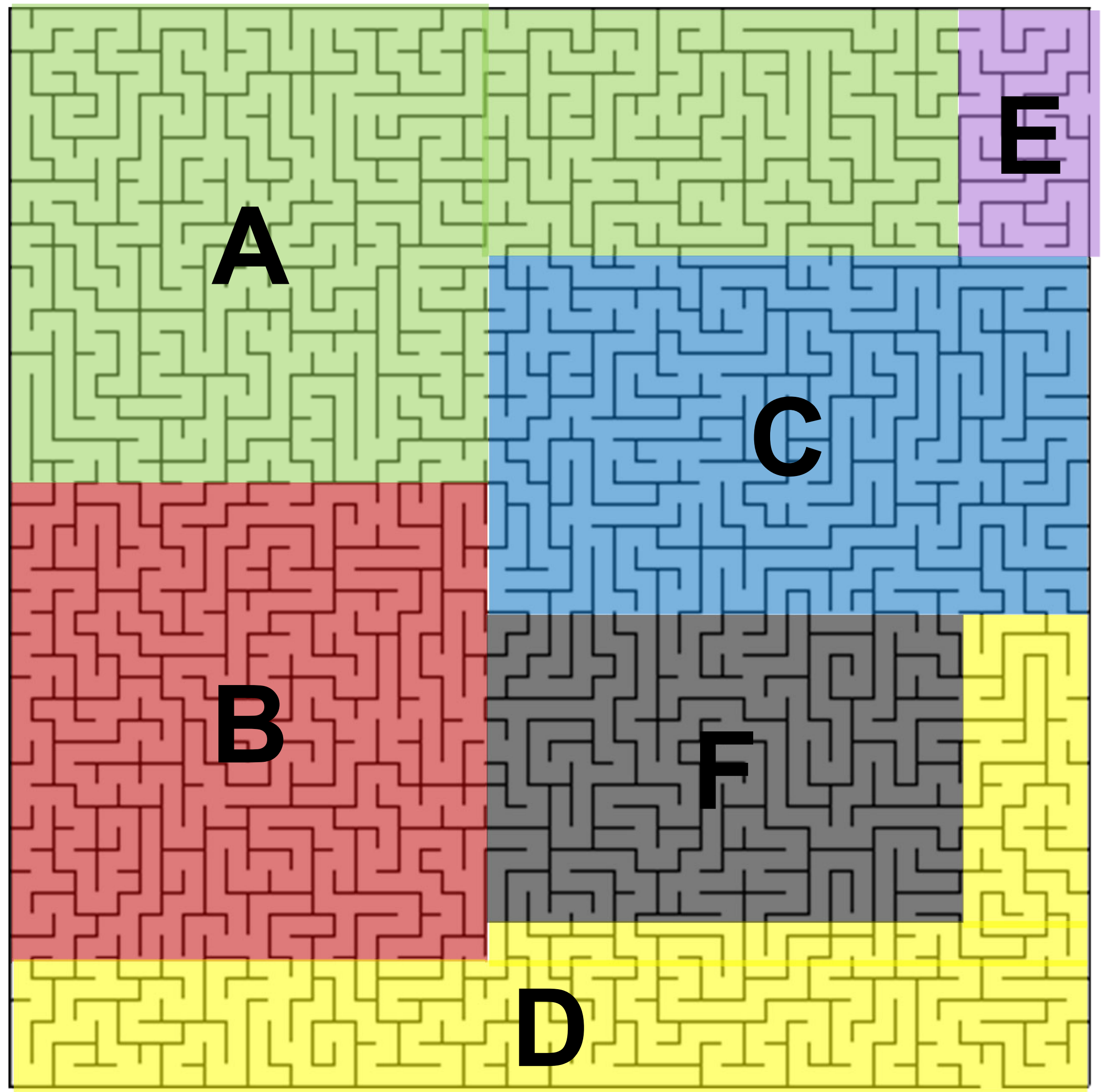
Prepare to answer queries.

destroyWall(x, y):

Remove walls from the maze using your superpowers.

isConnected(y, z):

Answer connectivity queries.



Dynamic Mazes

Preprocess:

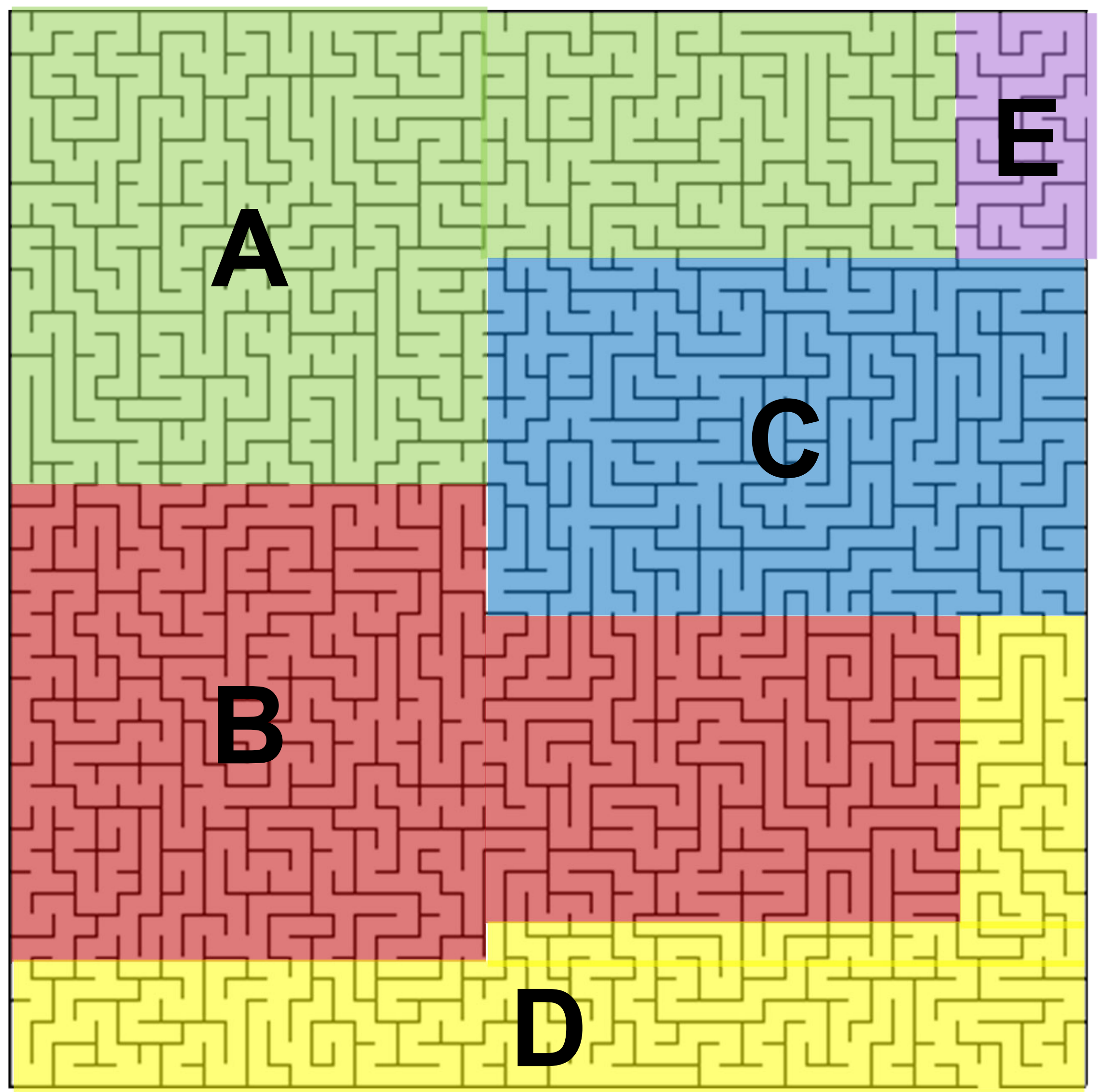
Prepare to answer queries.

destroyWall(x, y):

Remove walls from the maze using your superpowers.

isConnected(y, z):

Answer connectivity queries.



Dynamic Mazes

Preprocess:

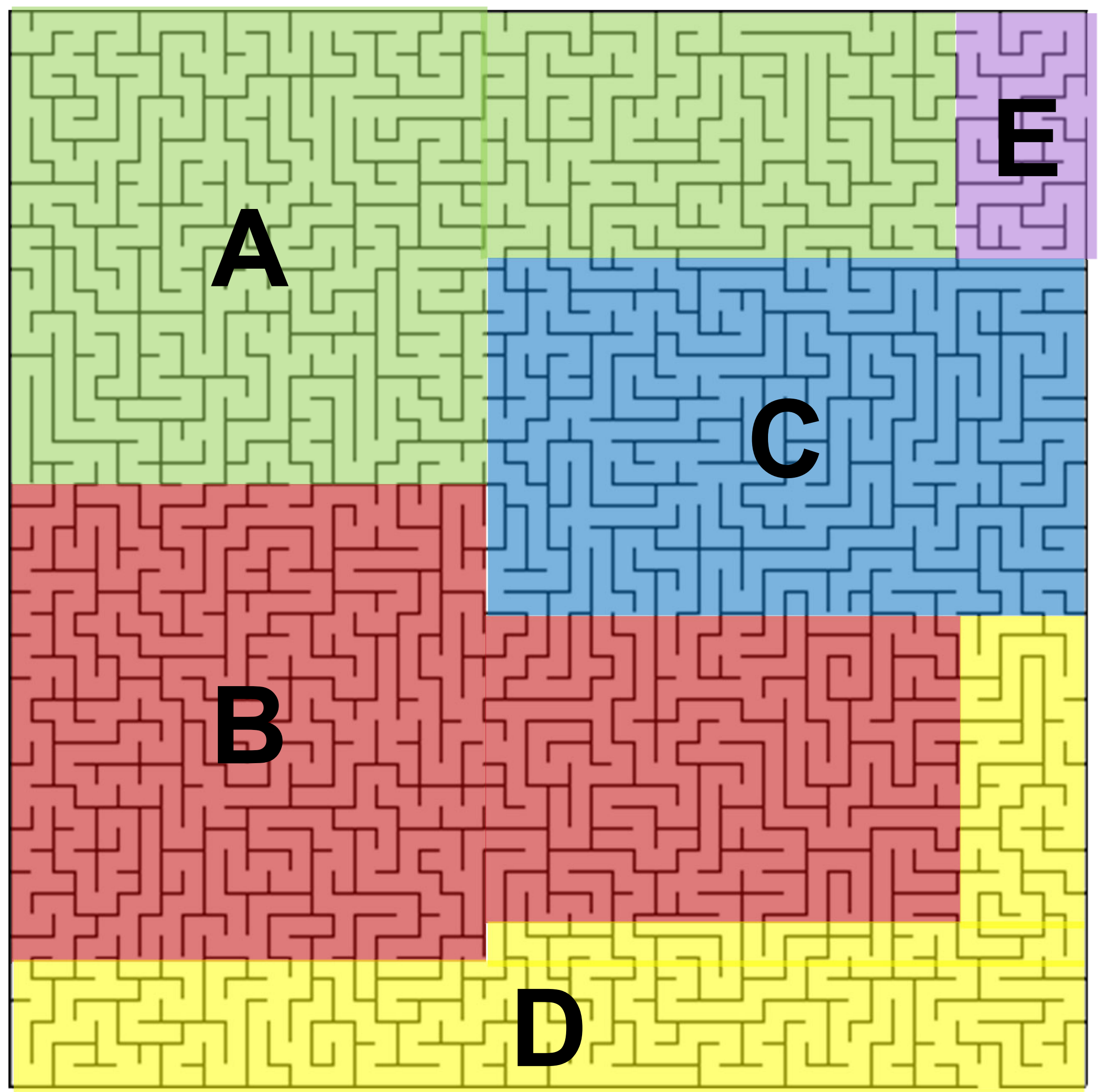
Prepare to answer queries.

Union(x, y):

Remove walls from the maze using your superpowers.

isConnected(y, z):

Answer connectivity queries.

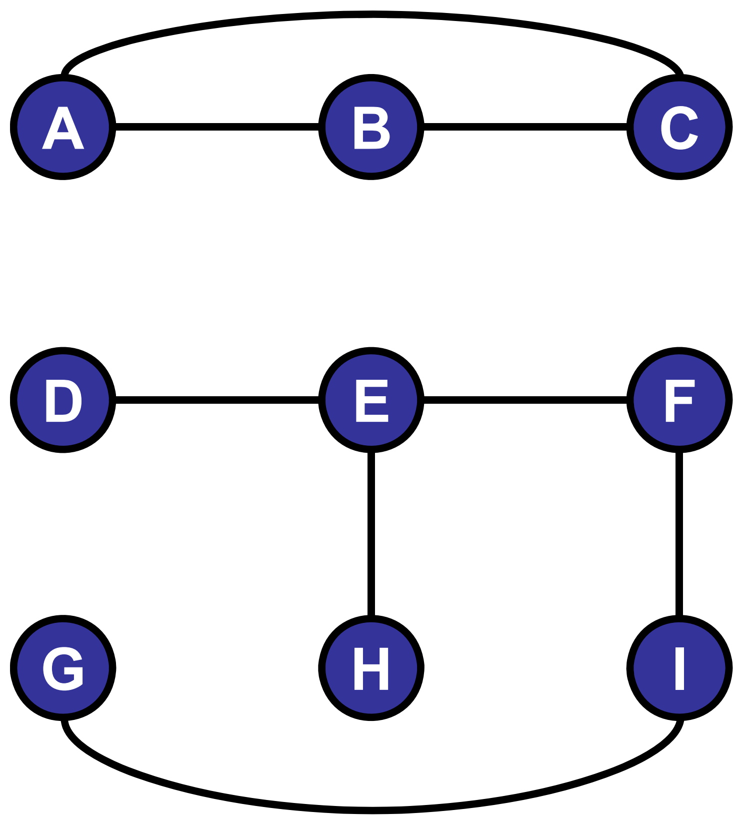


Dynamic Connectivity

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:

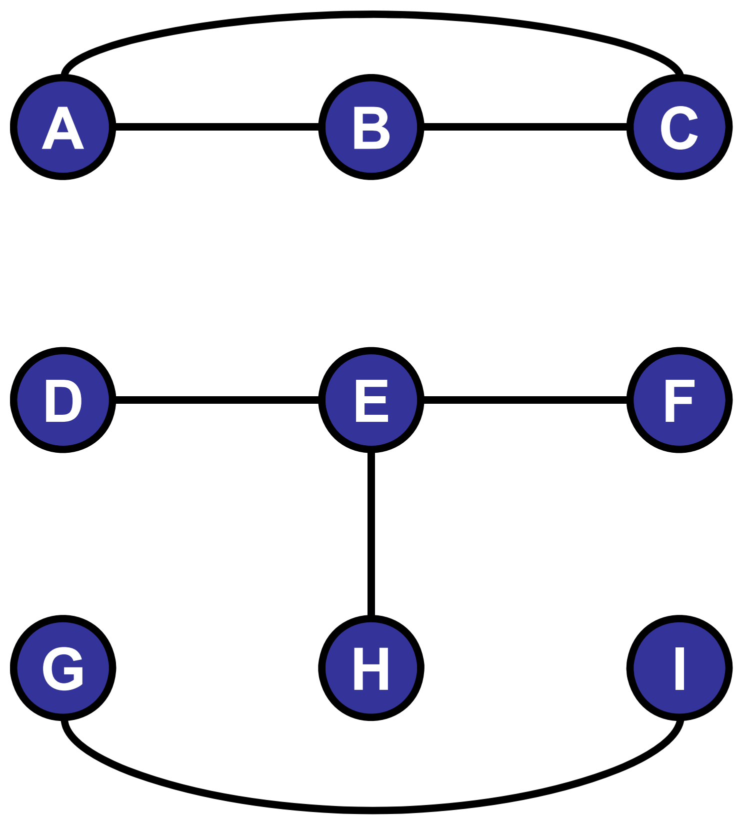
- Union: connect two objects
- Find: is there a path connecting the two objects?

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.



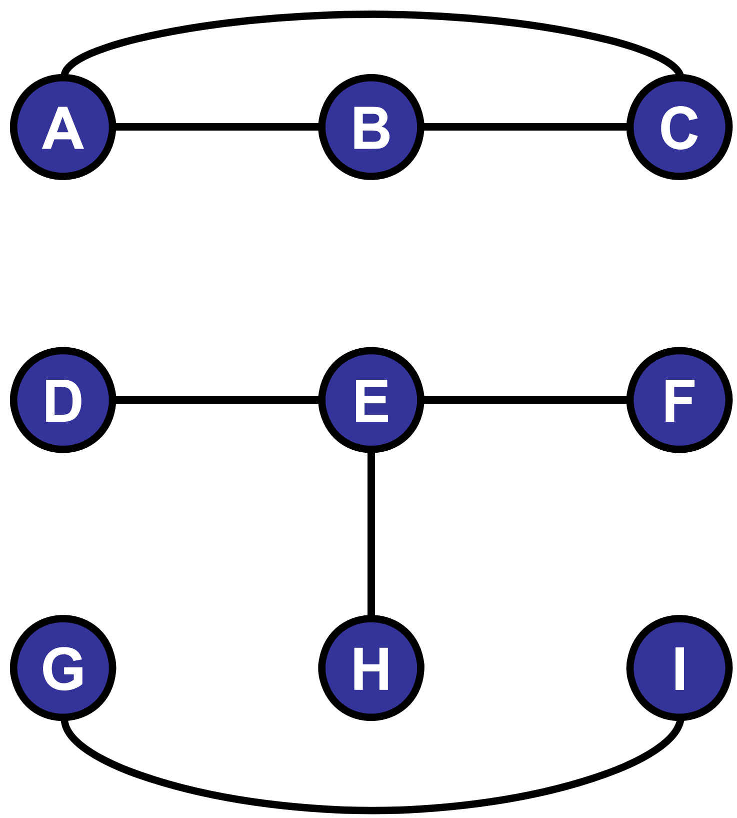
Dynamic Connectivity

Given a set of objects:

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

Maintain sets of nodes:

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



Dynamic Connectivity

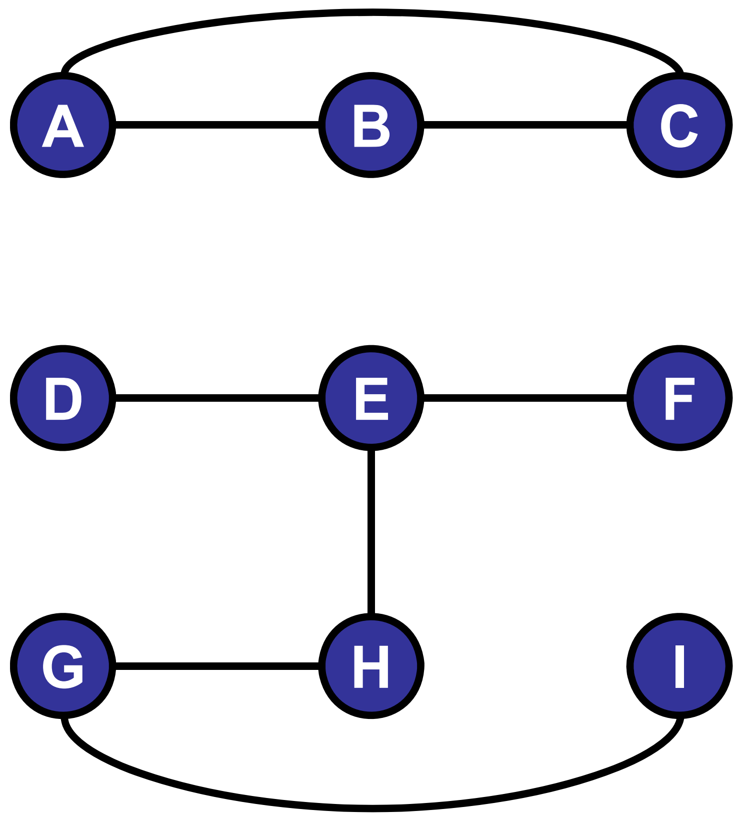
Given a set of objects:

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

Maintain sets of nodes:

{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

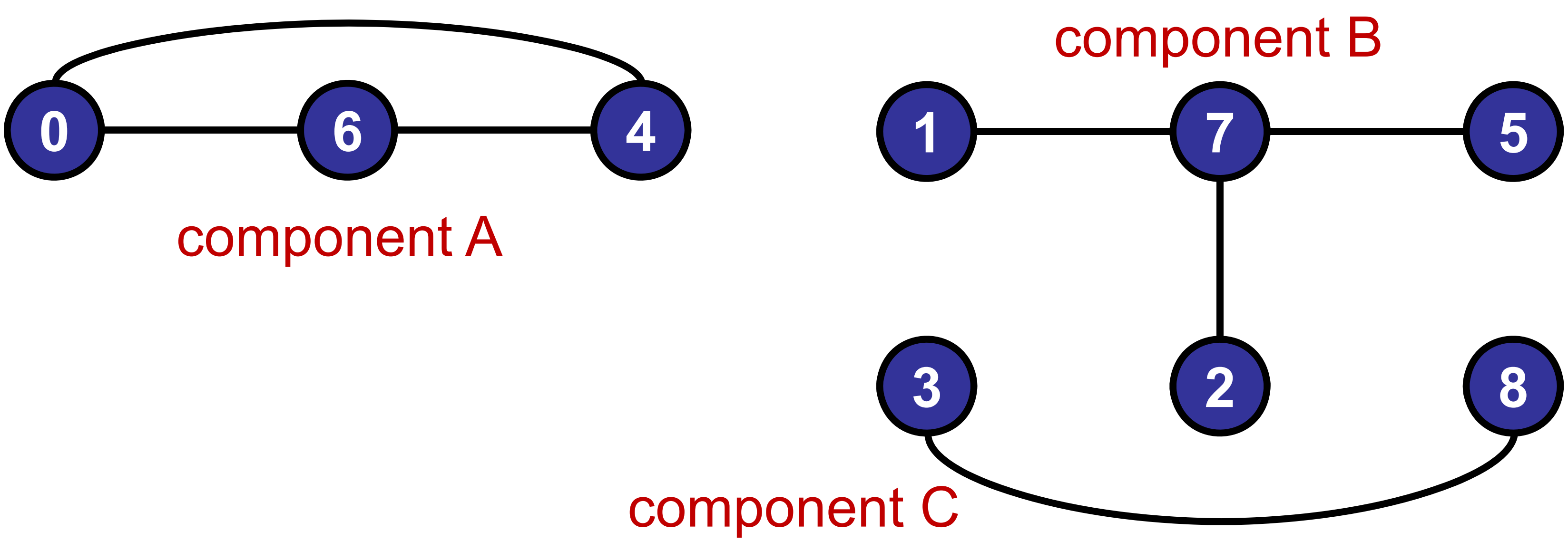
- Problem: Dynamic Connectivity
- Algorithm: Quick-Find
- Algorithm: Quick-Union
- Optimizations

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



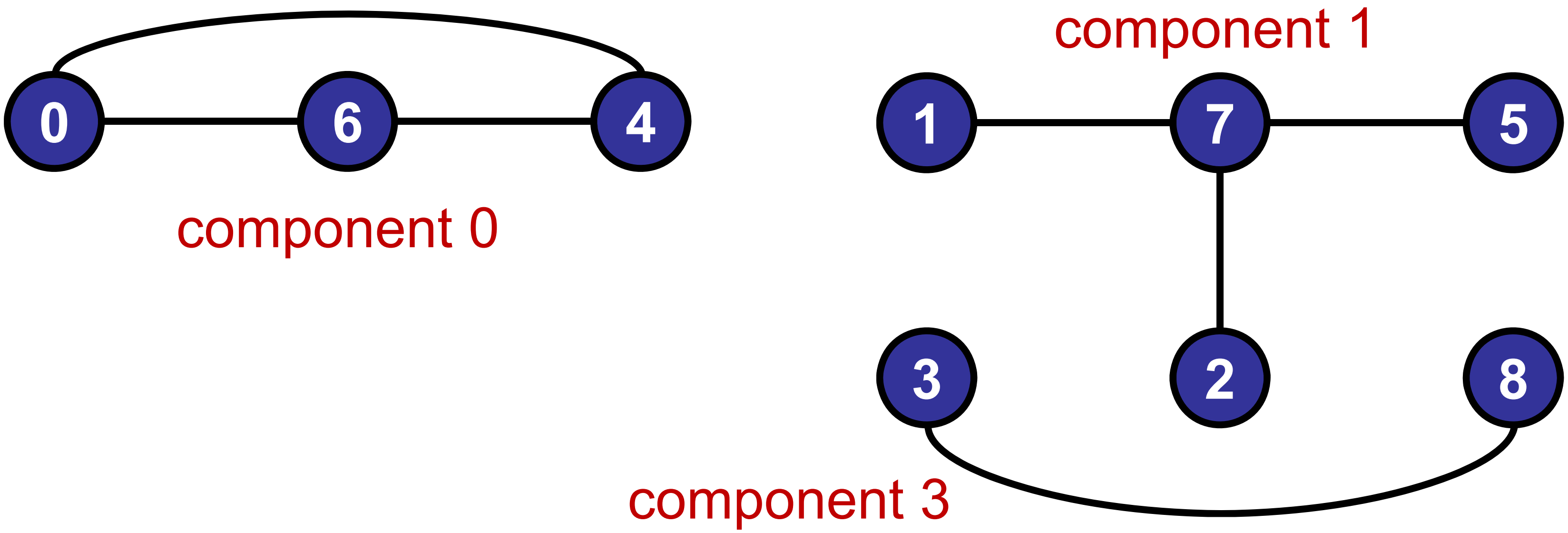
Quick Find

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



If objects are **not** integers, how could we convert them to integers?

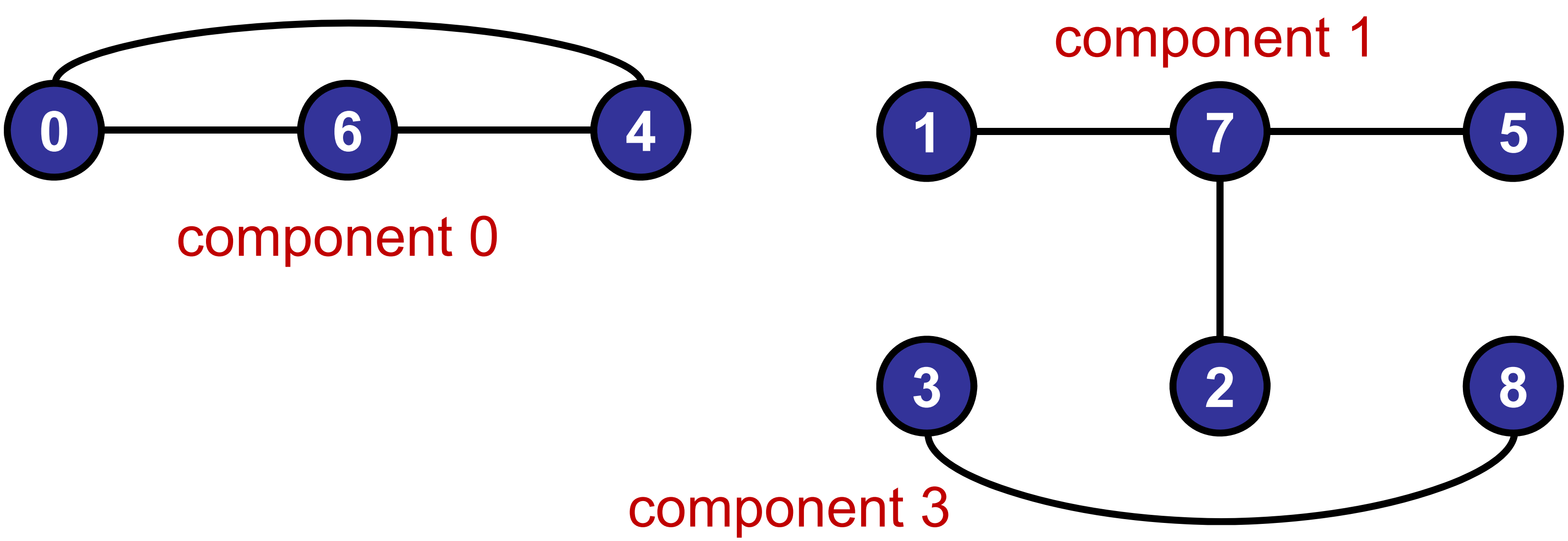
1. Binary search tree
2. Hash function
3. Hash table + chaining
4. Hash table + open addressing
5. Bloom filter
6. Priority queue

Quick Find

Data structure:

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

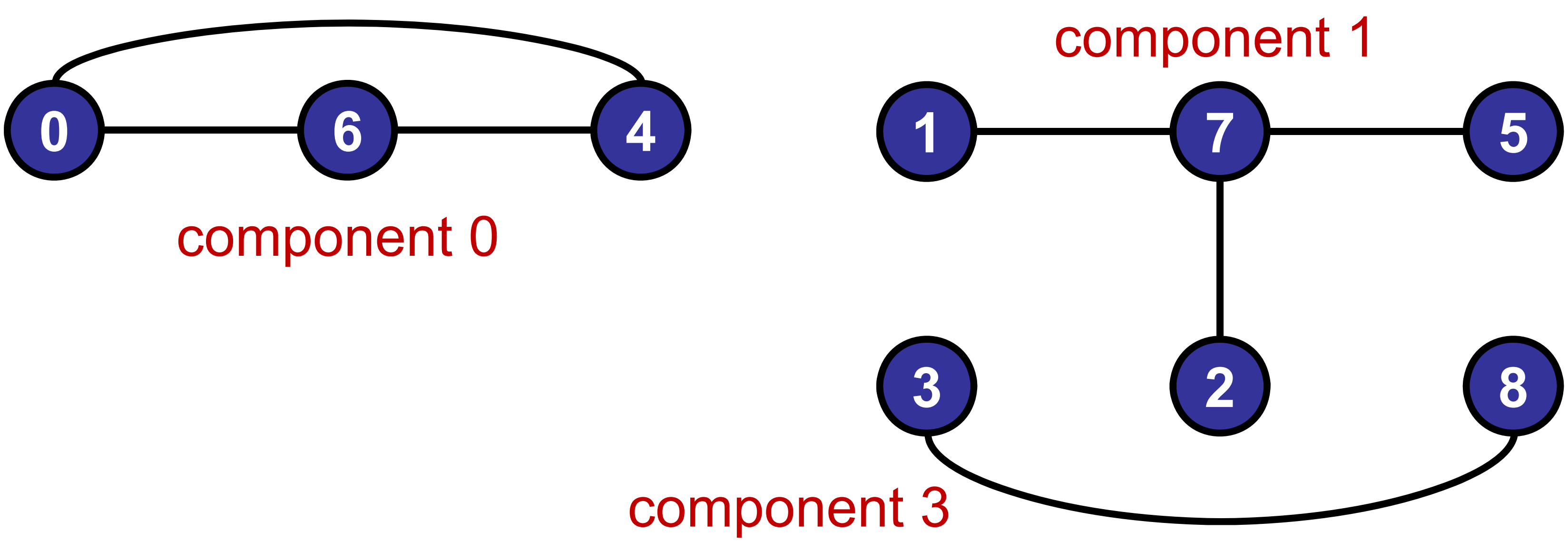
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	1	3	0	1	0	1	3



Quick Find

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

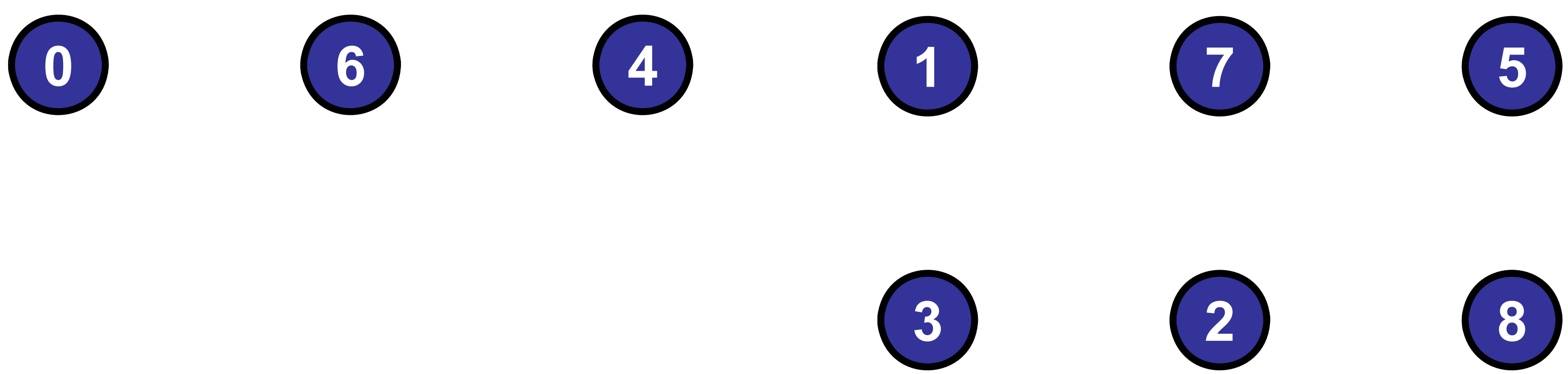
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	1	3	0	1	0	1	3



Quick Find

Initial state of data structure:

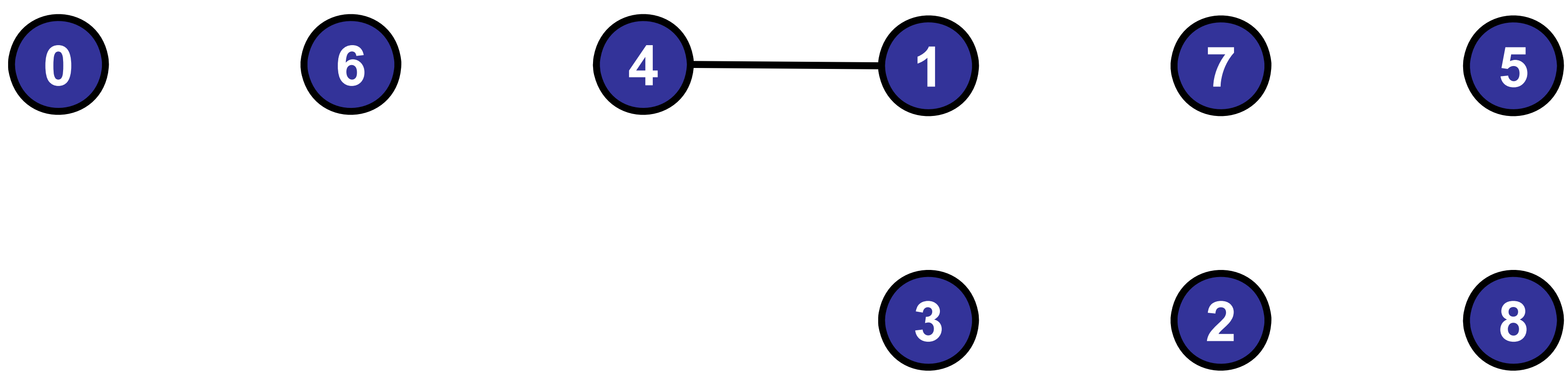
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	2	3	4	5	6	7	8



Quick Find

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

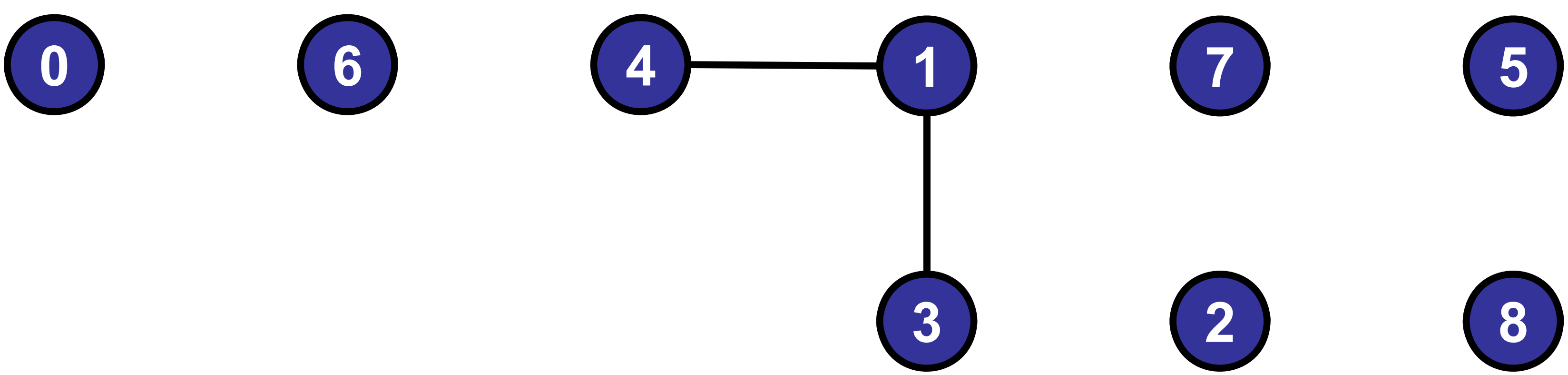
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	2	3	1	5	6	7	8



Quick Find

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

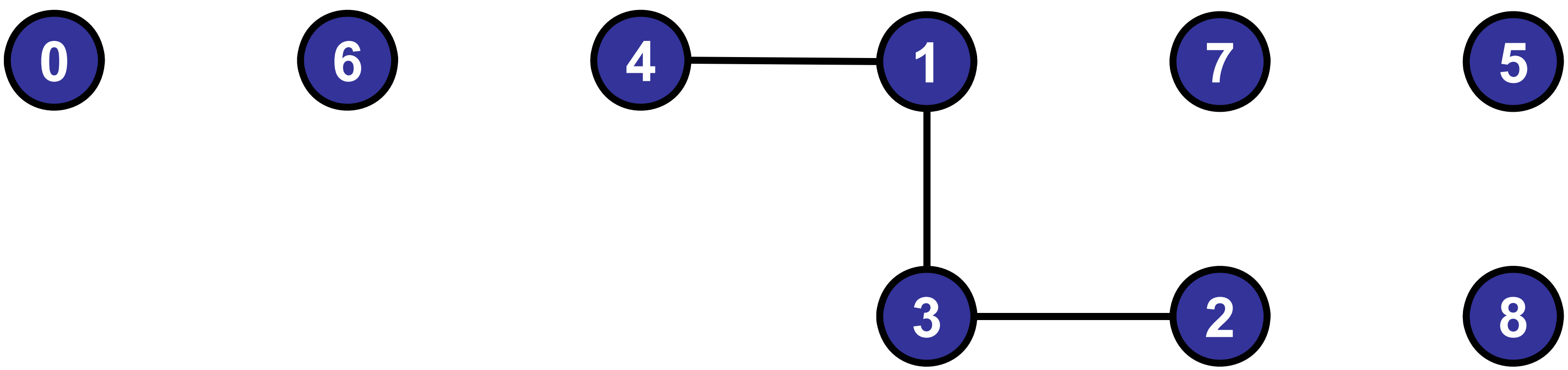
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	2	1	1	5	6	7	8



Quick Find

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

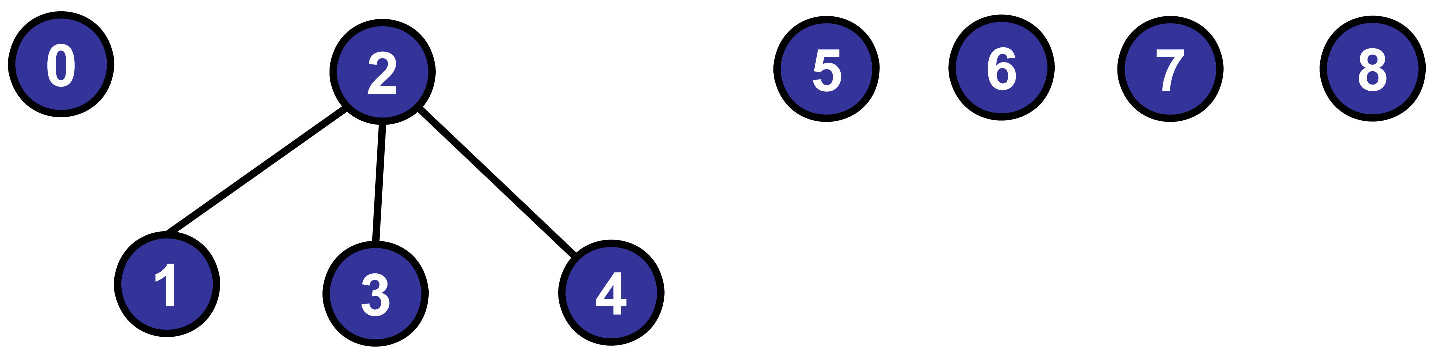
object	0	1	2	3	4	5	6	7	8
component identifier	0	2	2	2	2	5	6	7	8



Quick Find

Flat trees:

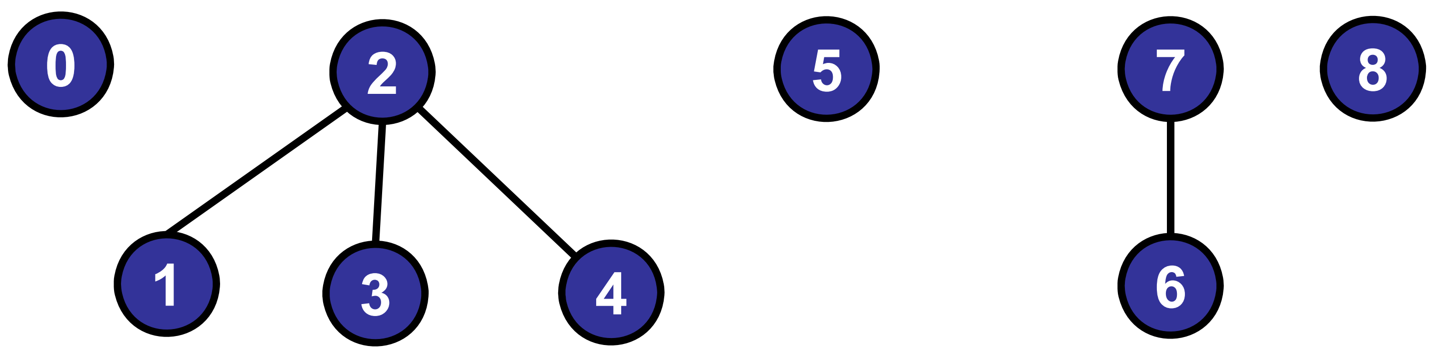
object	0	1	2	3	4	5	6	7	8
component identifier	0	2	2	2	2	5	6	7	8



Quick Find

Flat trees:

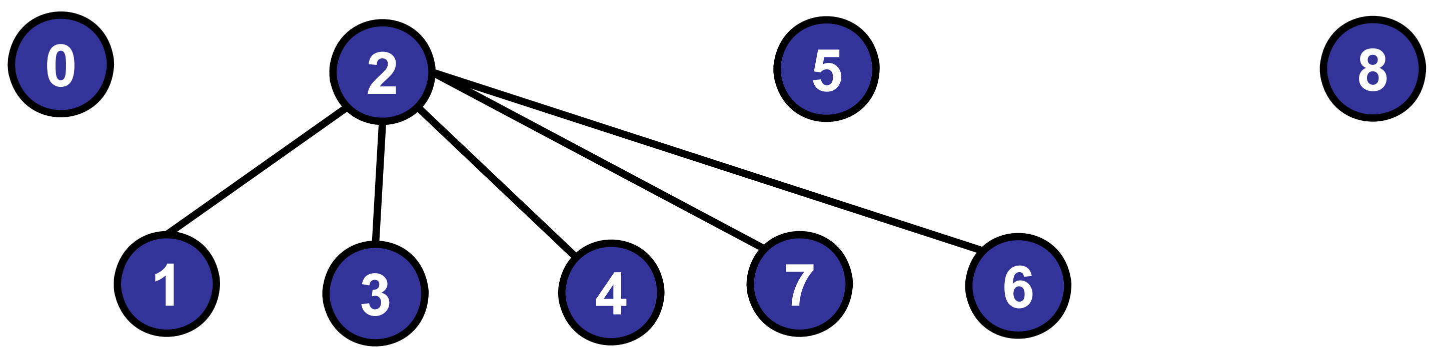
object	0	1	2	3	4	5	6	7	8
component identifier	0	2	2	2	2	5	7	7	8



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



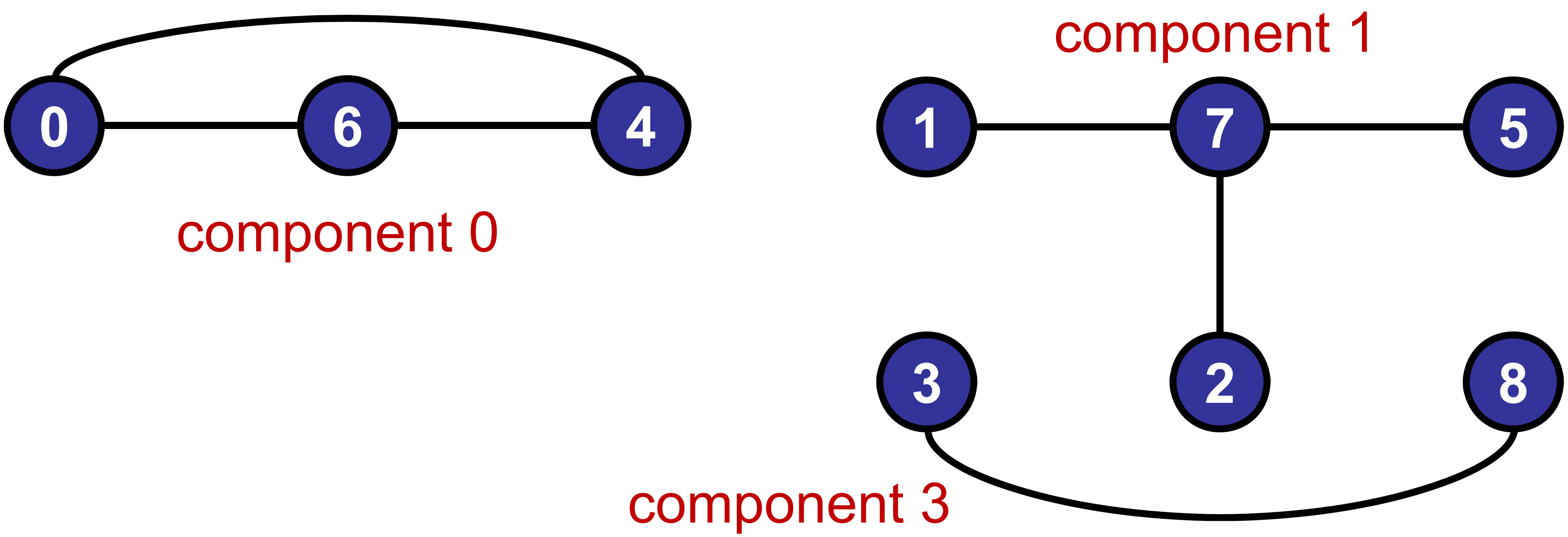
Running time of (Find, Union):

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]);
```

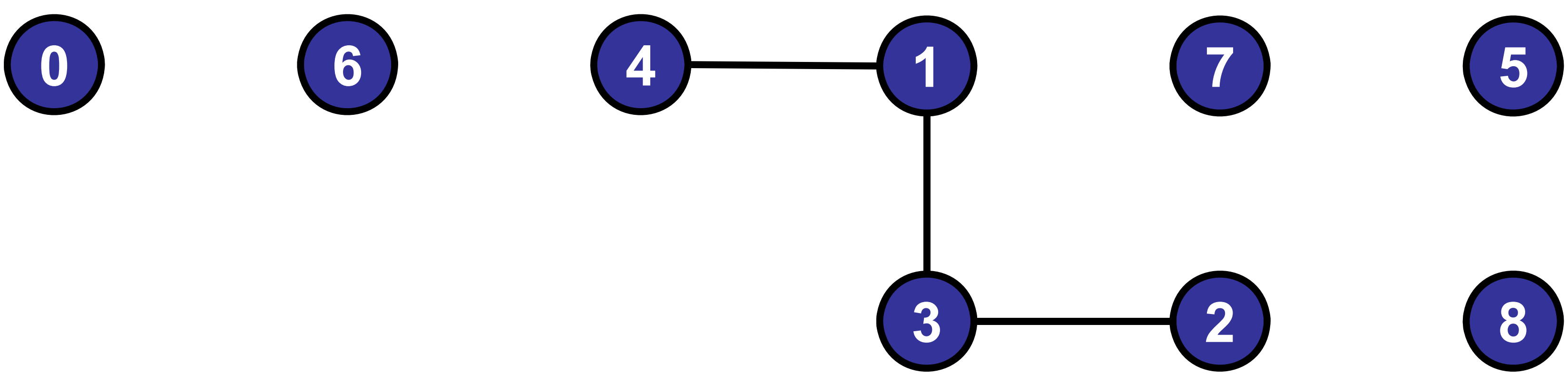
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	1	3	0	1	0	1	3



Quick Find

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

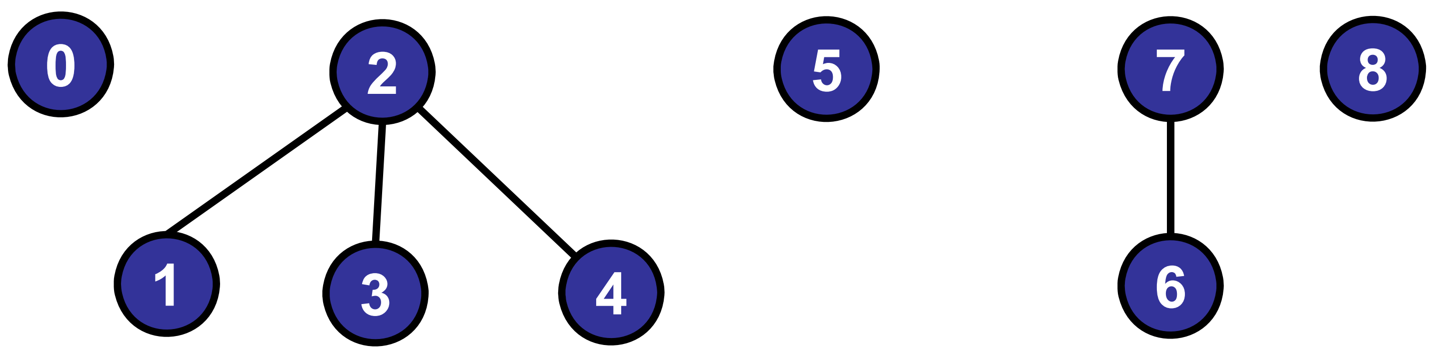
Disjoint Set

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

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

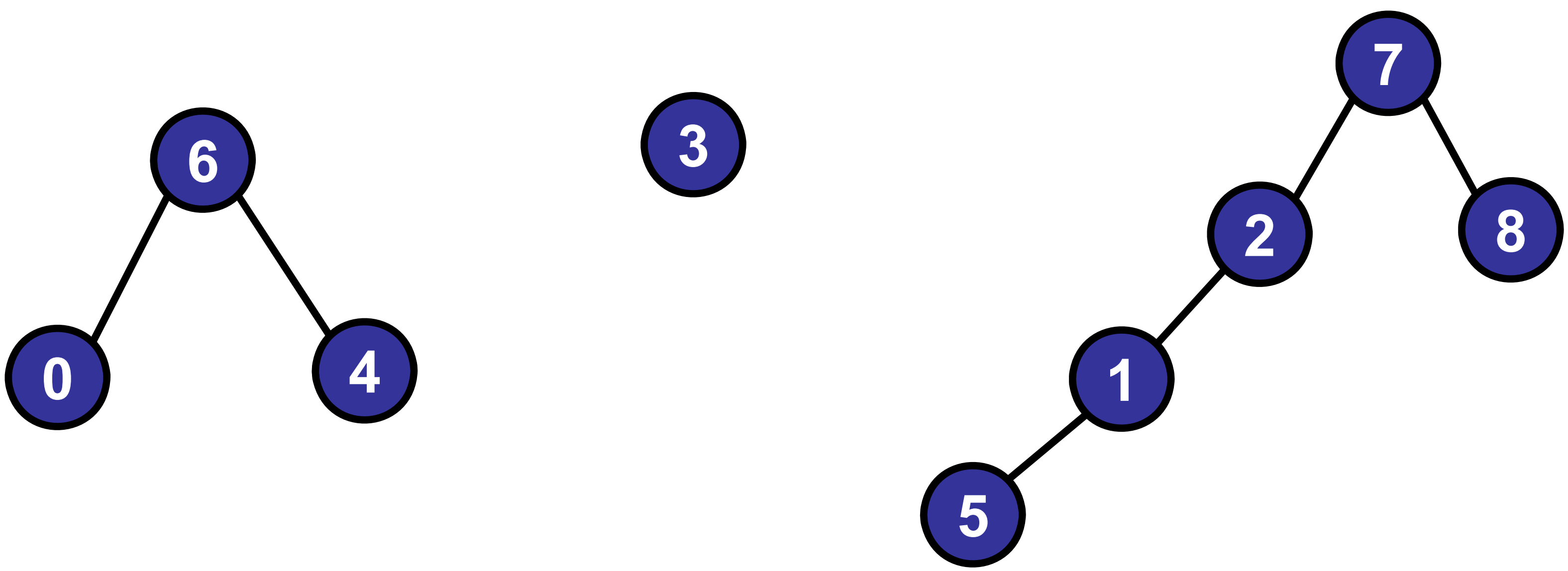


Quick Union

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

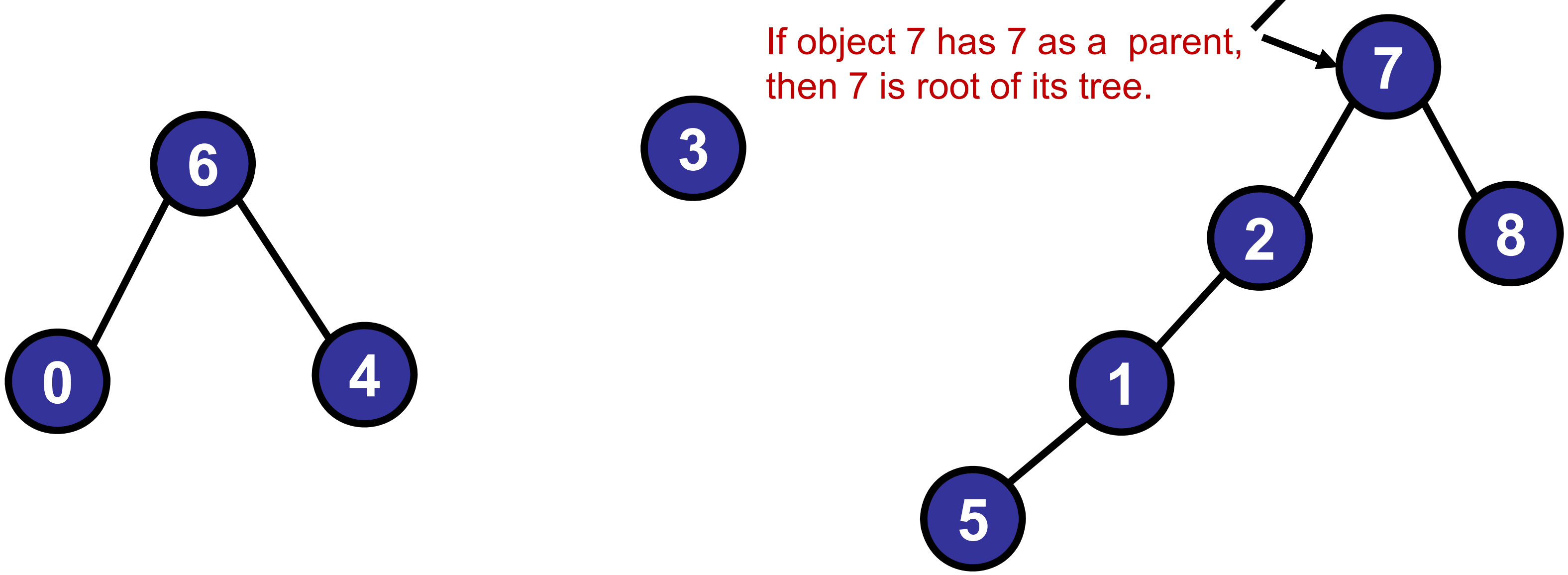


Quick Union

Data structure:

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

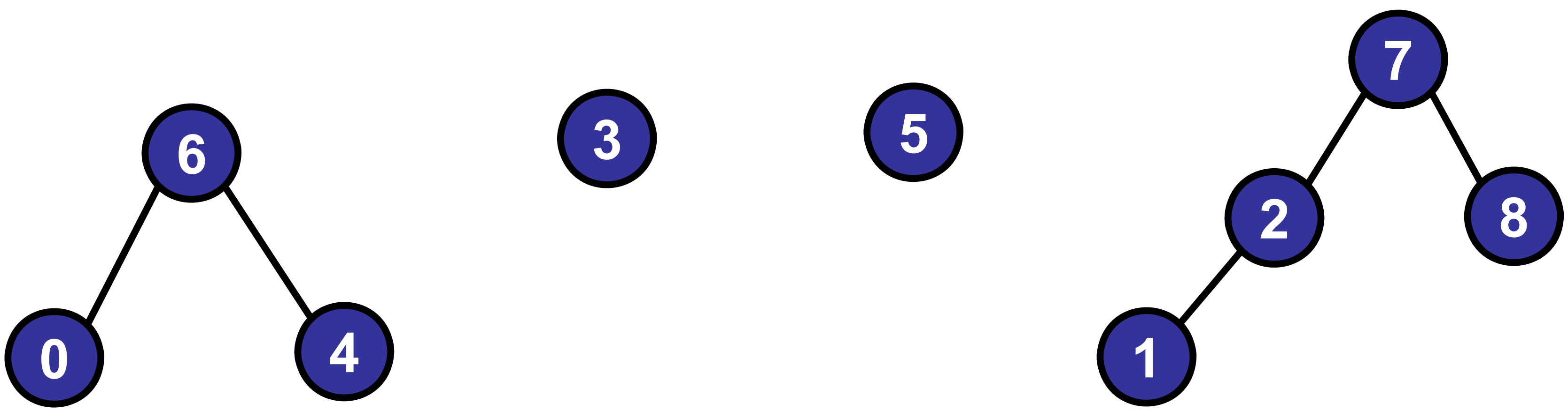
object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



Quick Union

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

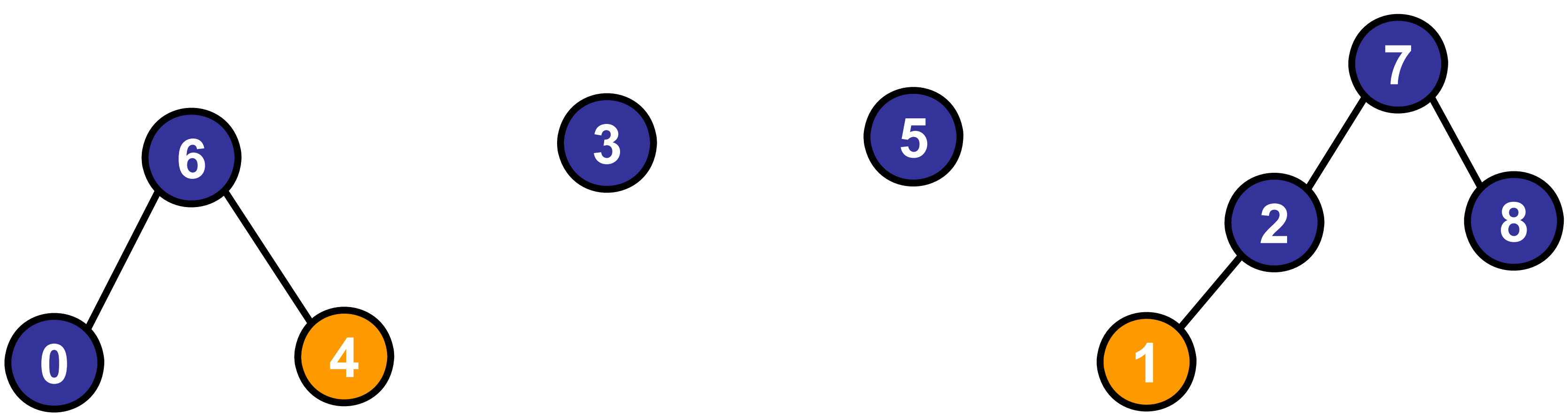
object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



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



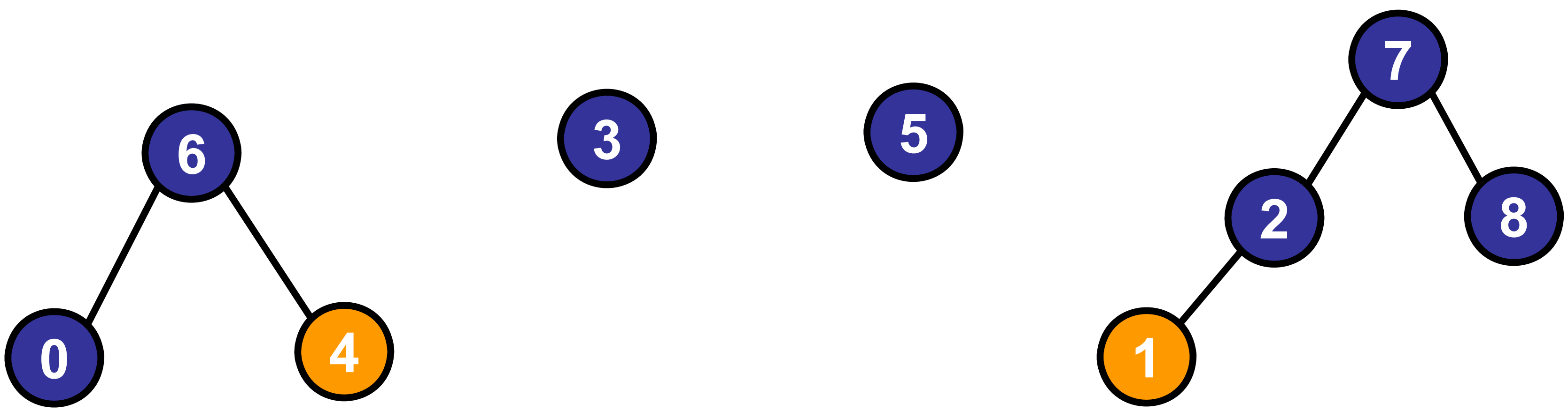
Quick Union

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

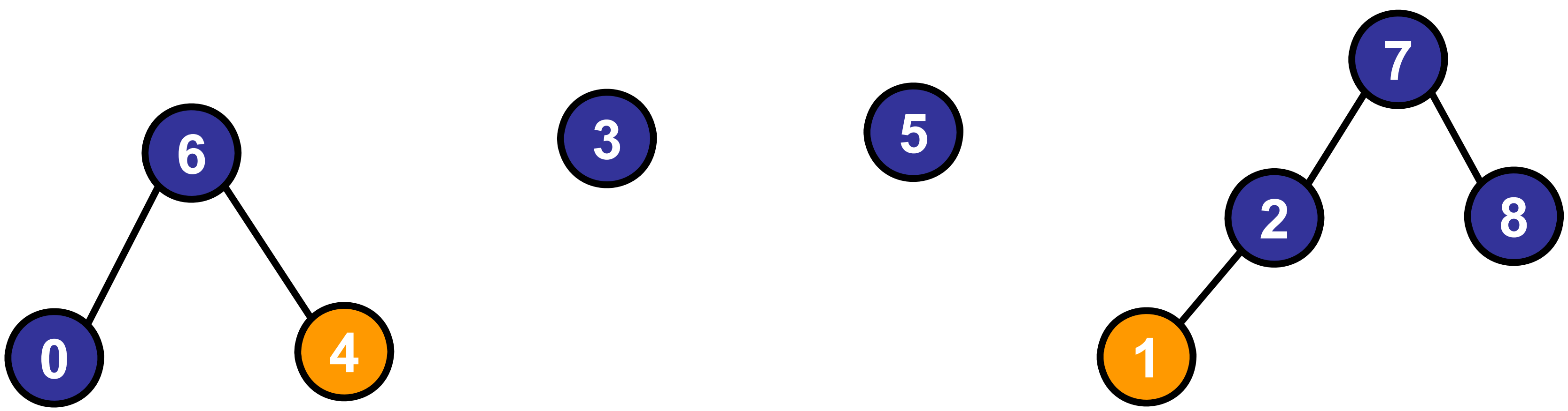
Example: `find(4, 1)`

4 → 6 → 6

1 → 2 → 7 → 7

return (6 == 7) → **false**

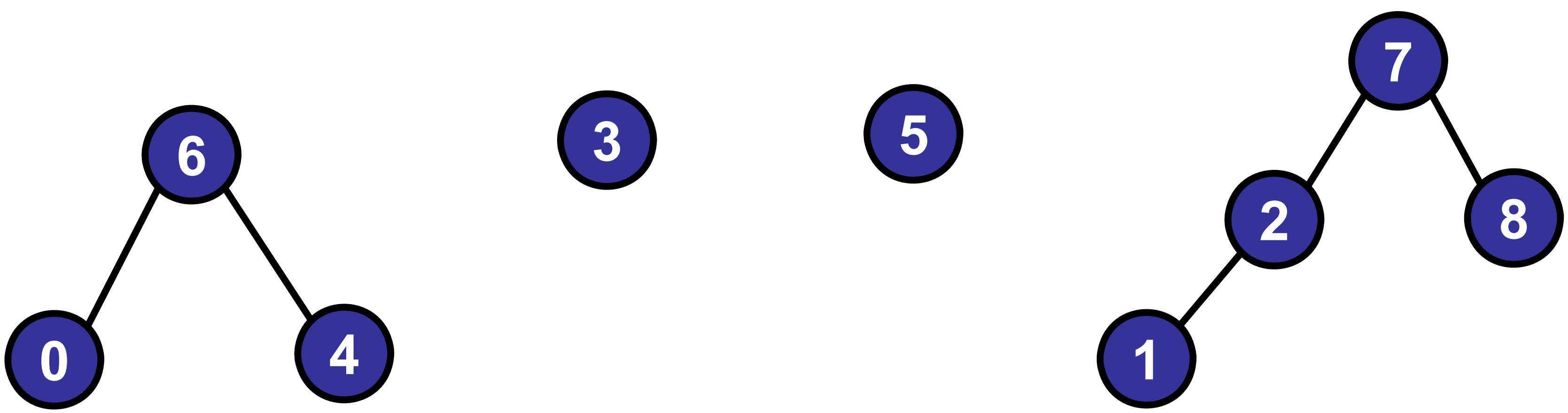
object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



Quick Union

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

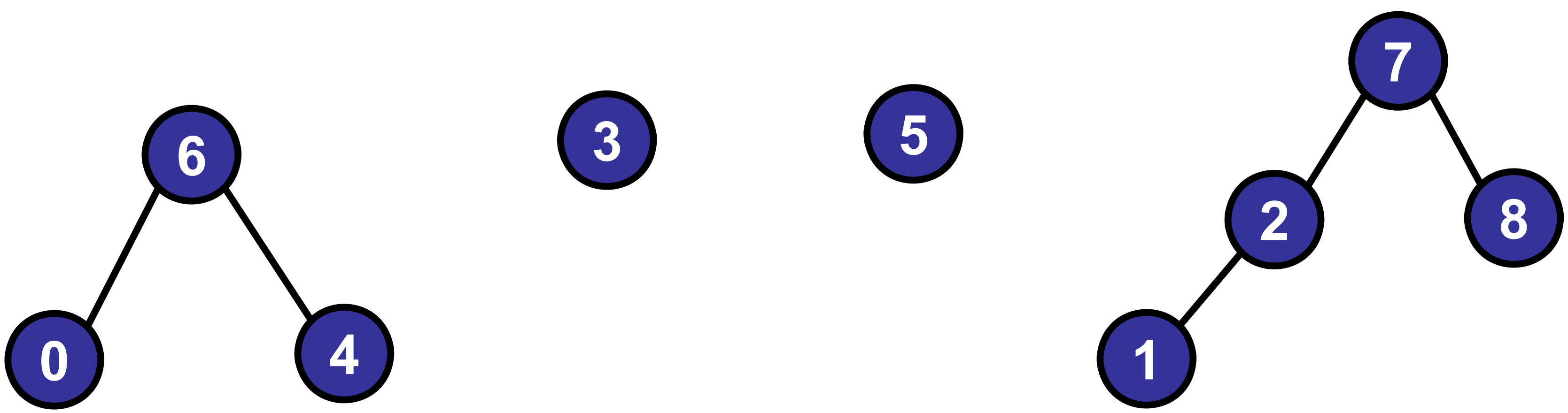
object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



Quick Union

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

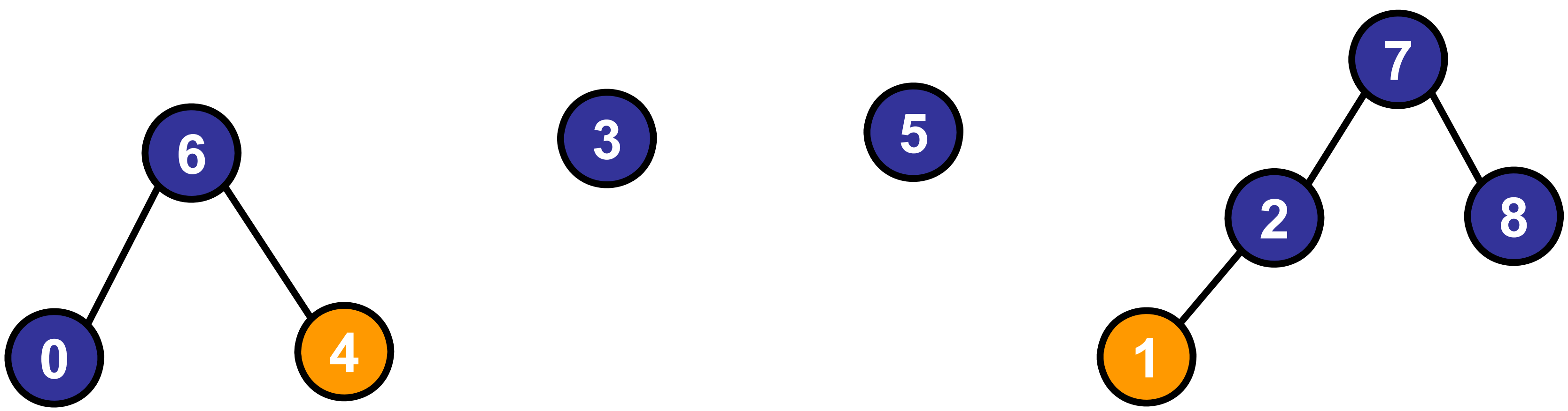
object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



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



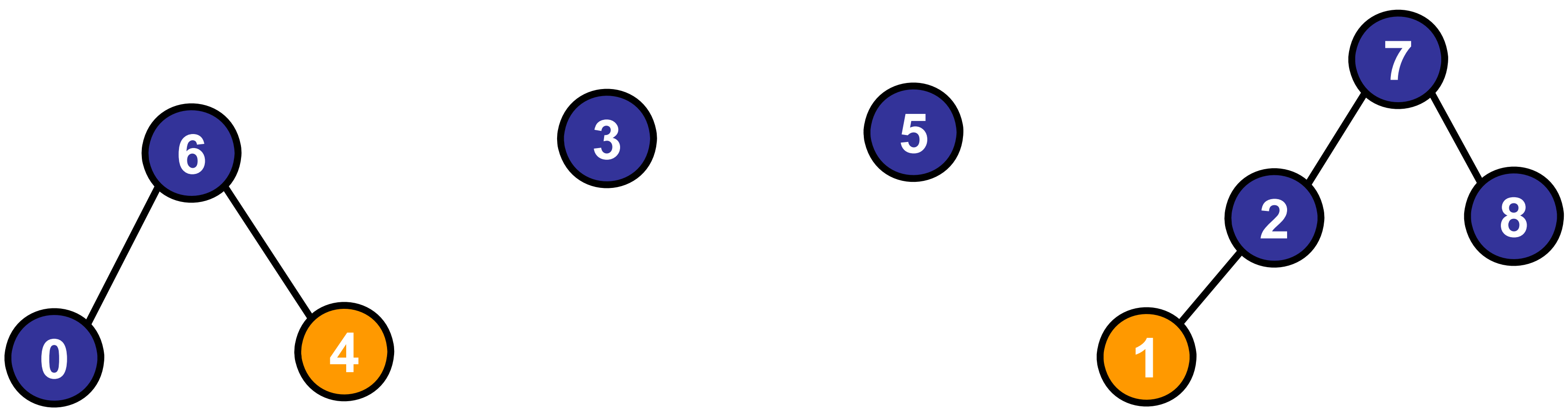
Quick Union

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

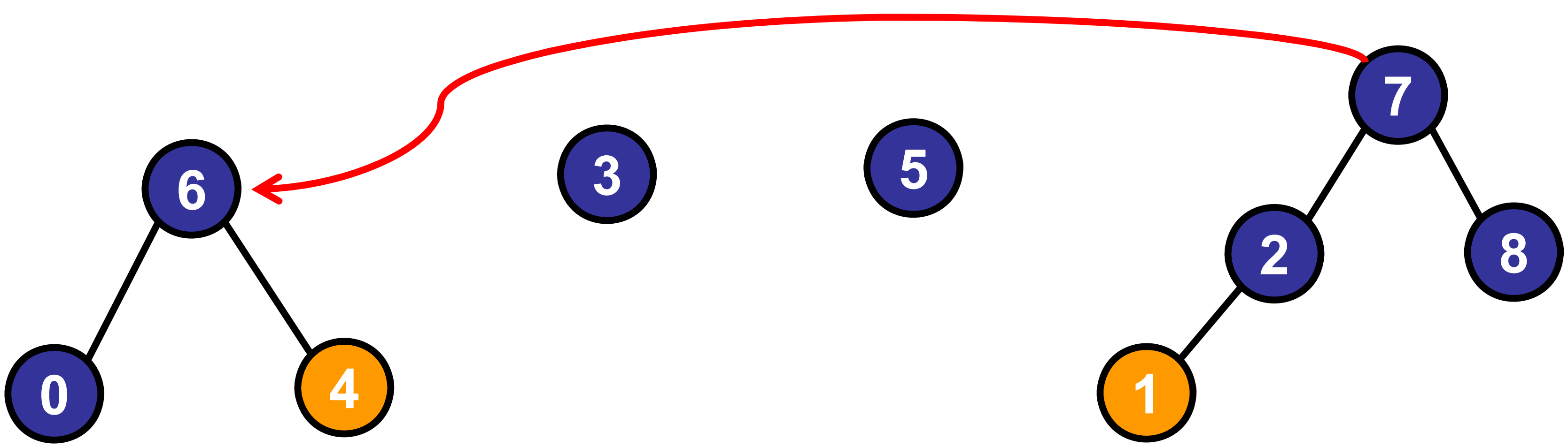
Example: union(1, 4)

```
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

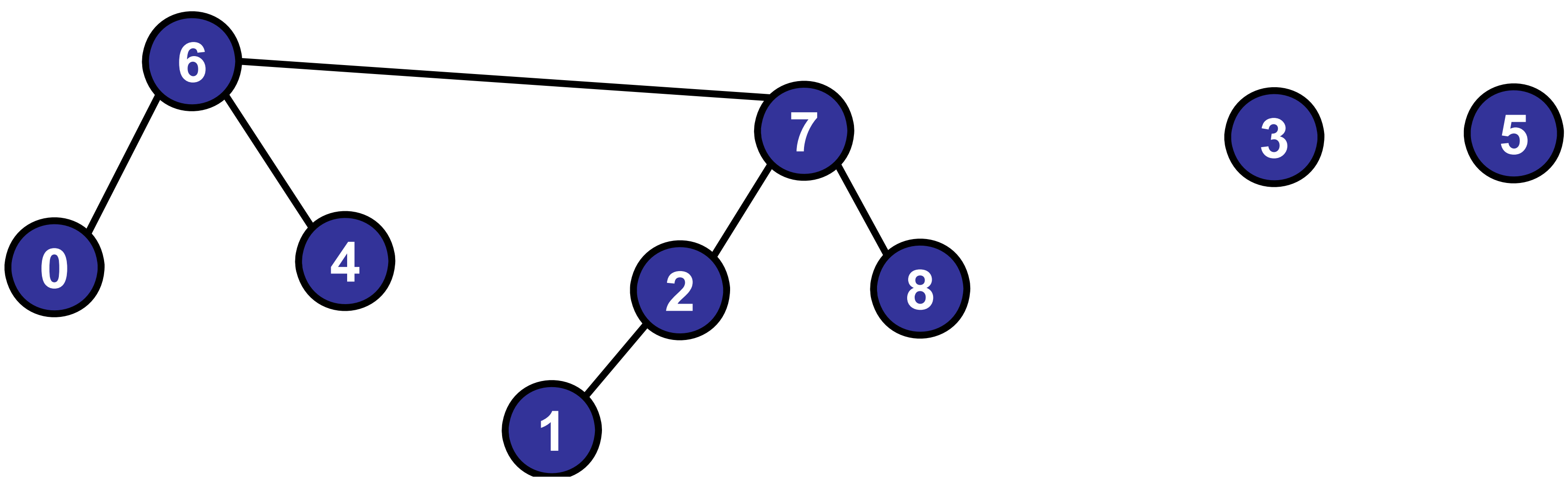
Example: union(1, 4)

```
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



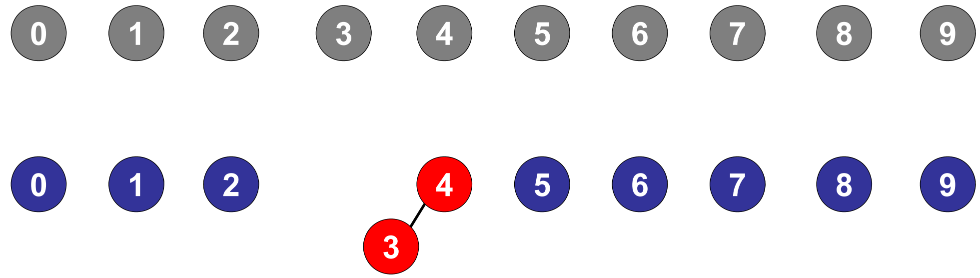
Example:

P	0	1	2	3	4	5	6	7	8	9

0 1 2 3 4 5 6 7 8 9

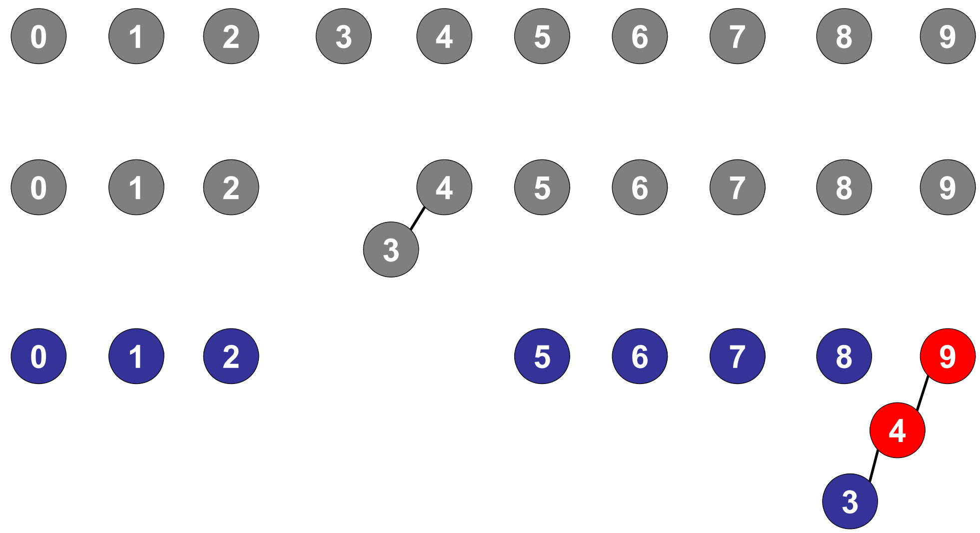
Example:

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



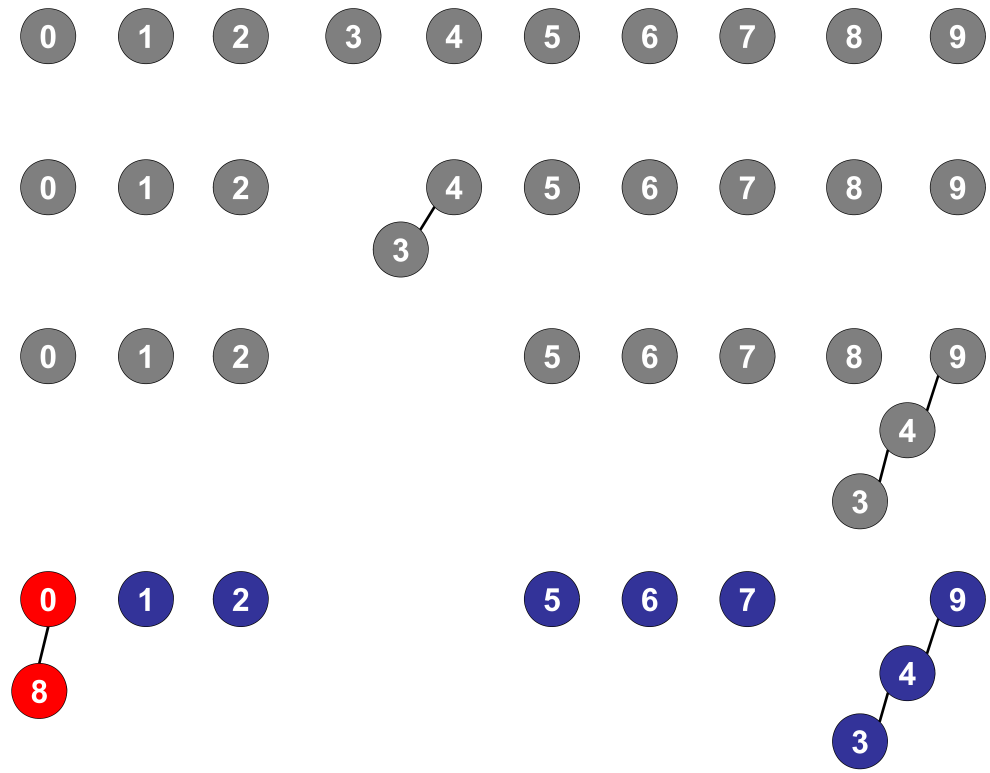
Example:

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



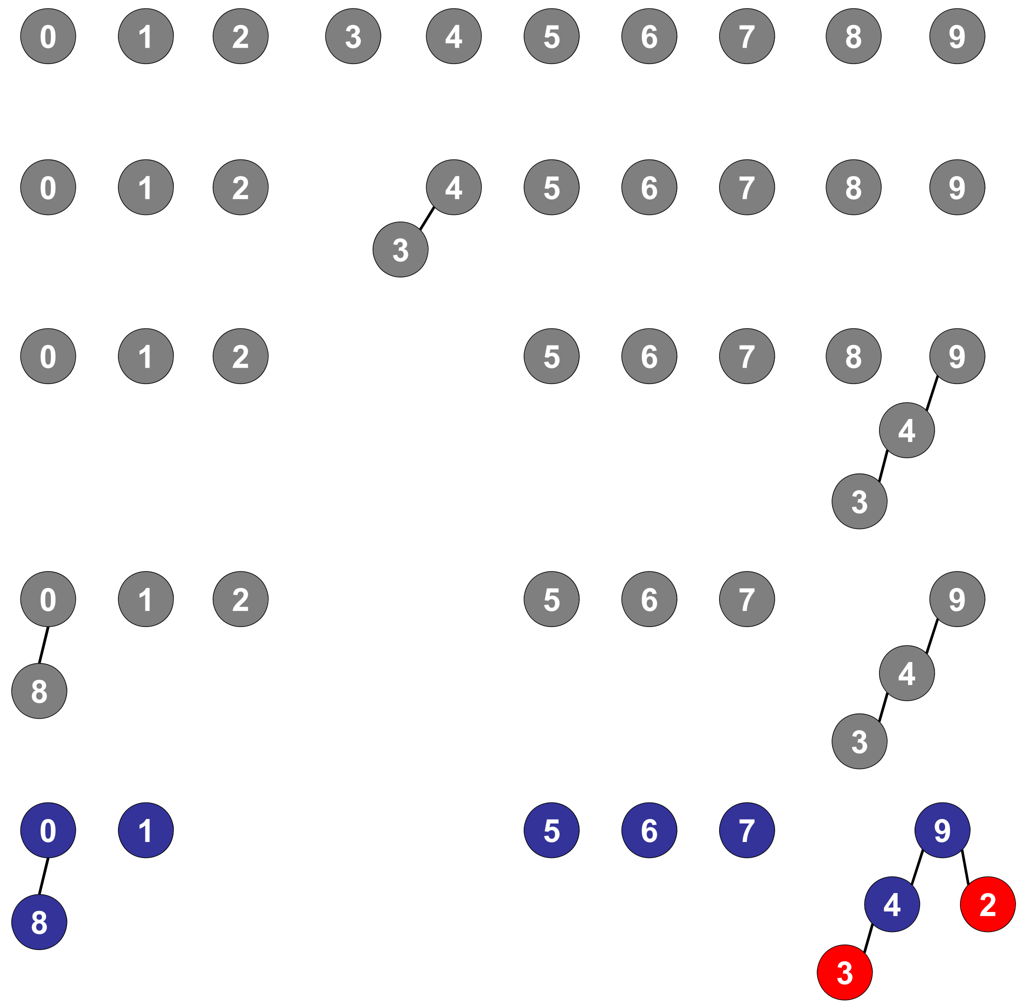
Example:

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



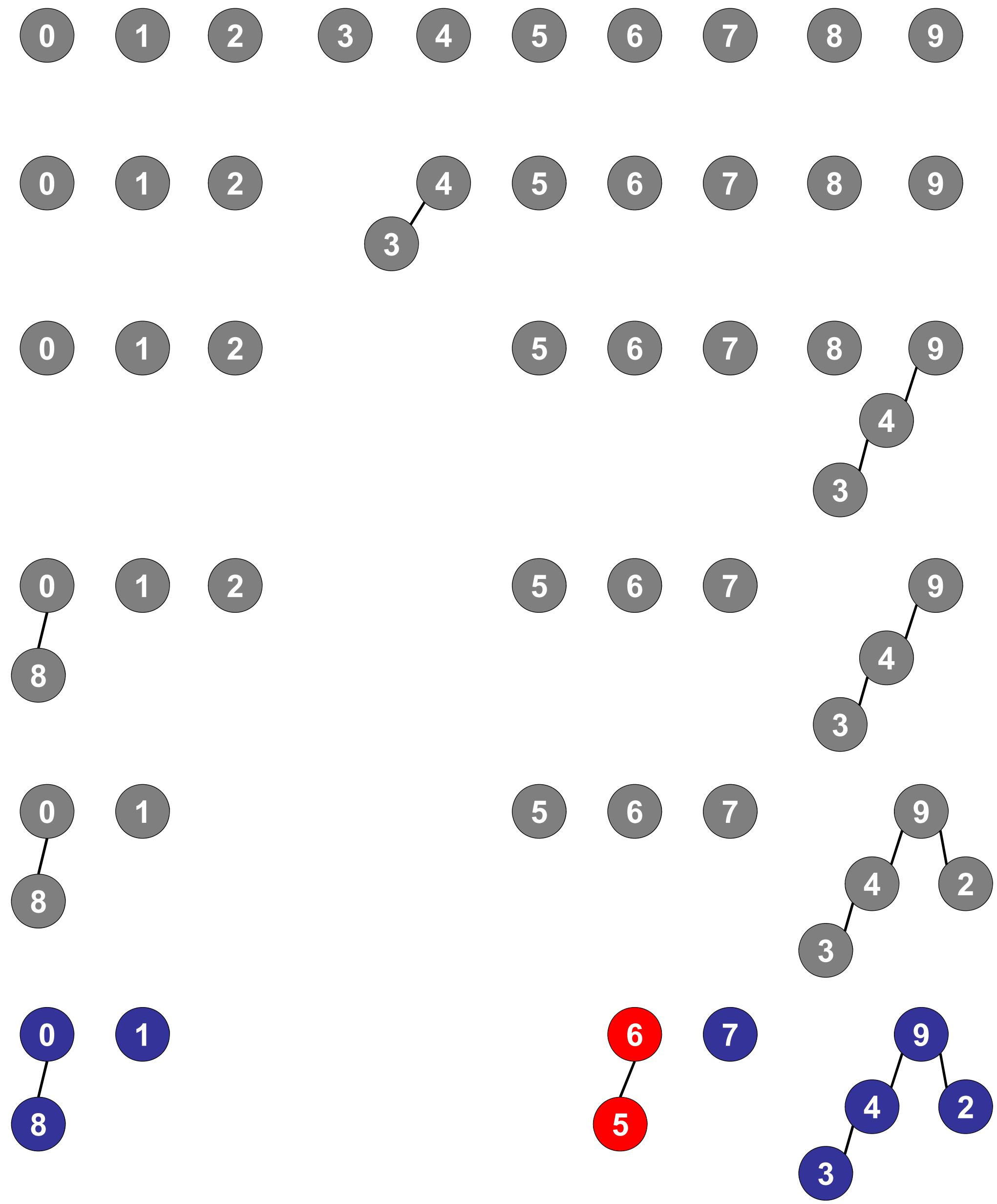
Example:

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

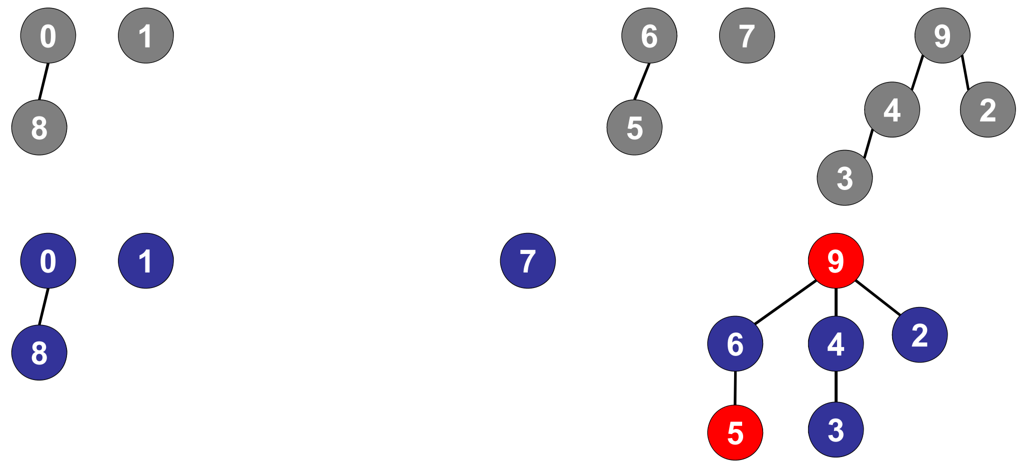


Example:

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

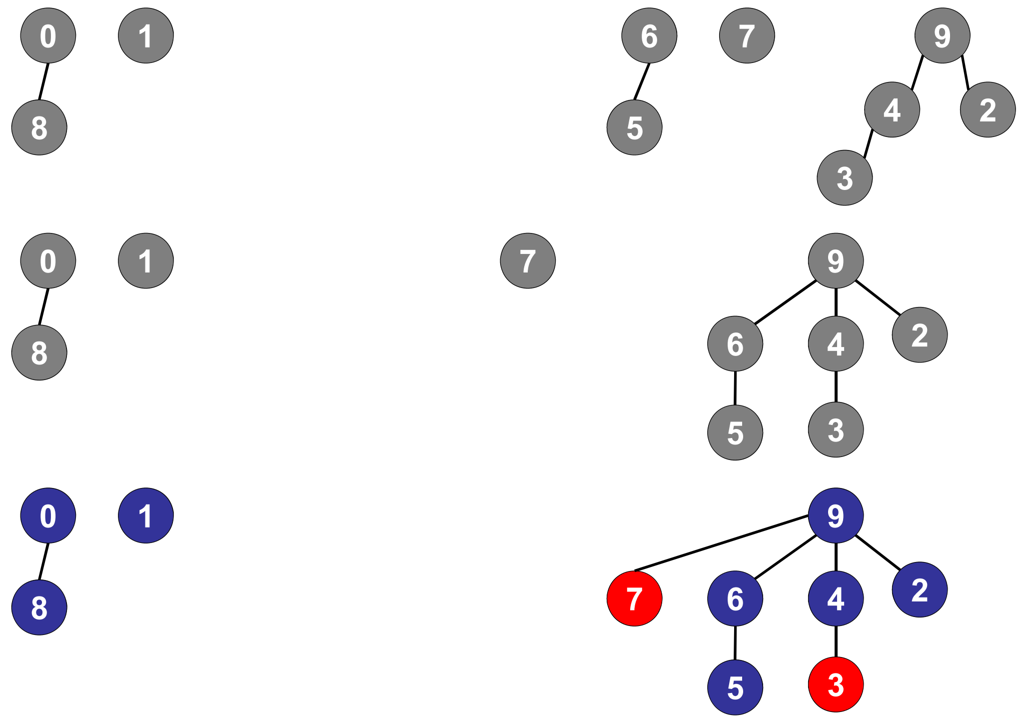
Example:

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



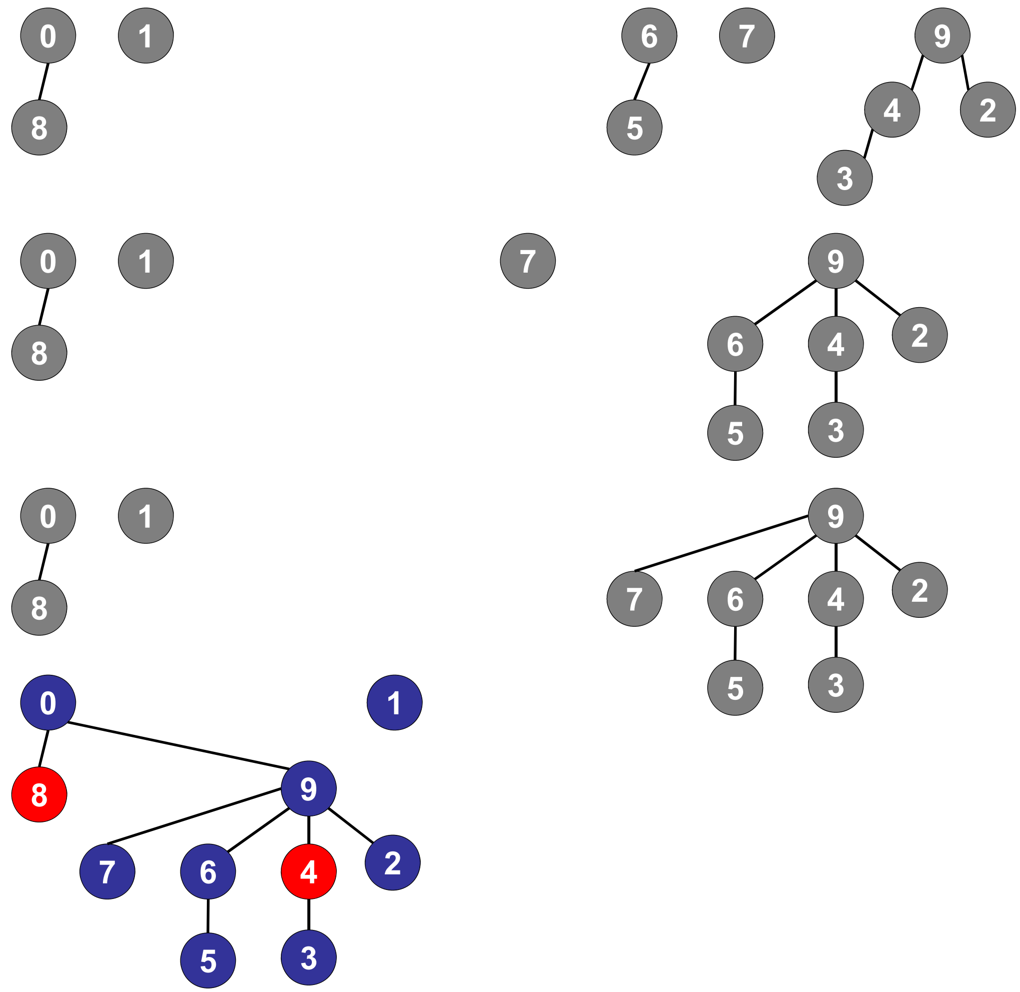
Example:

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



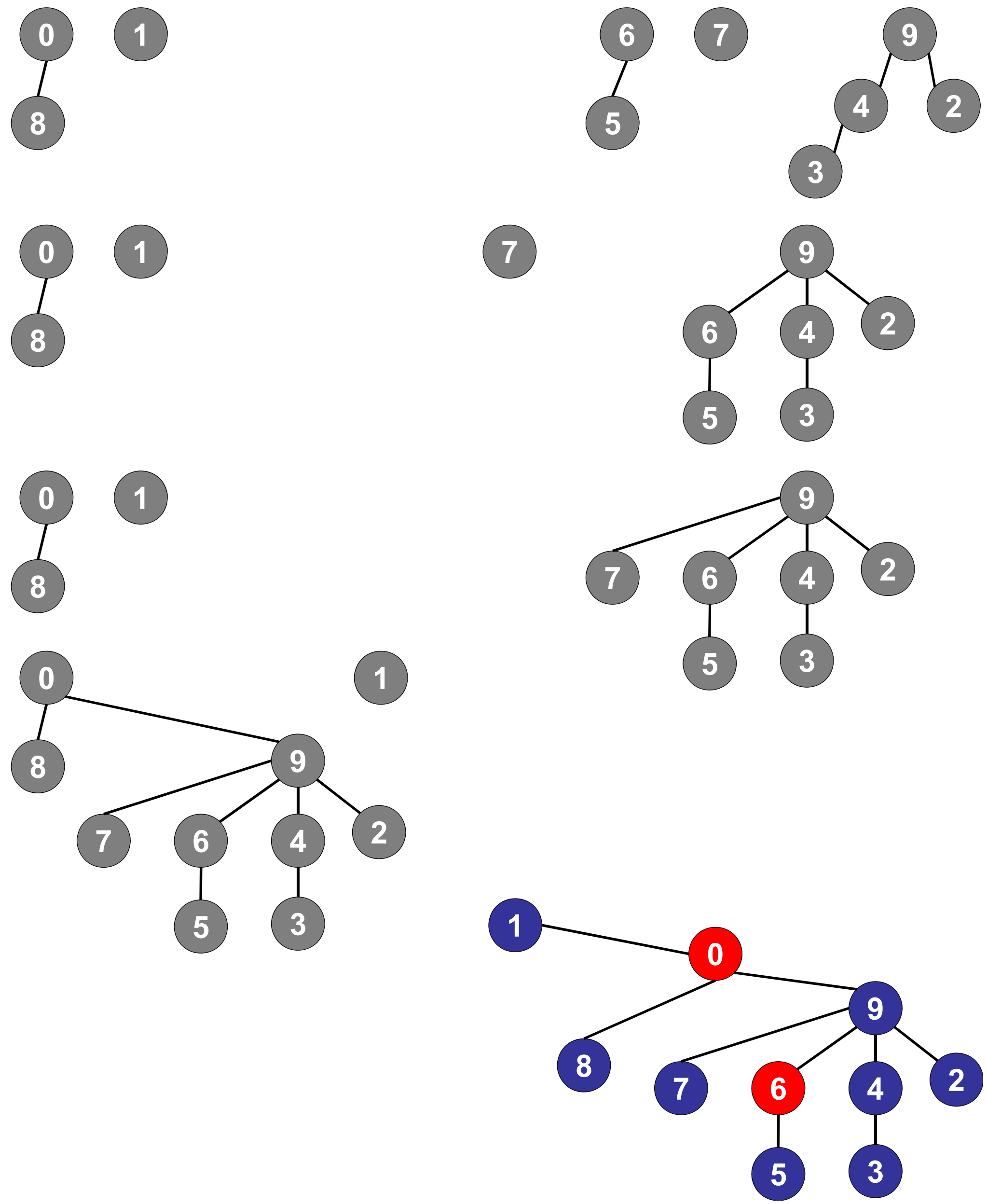
Example:

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:

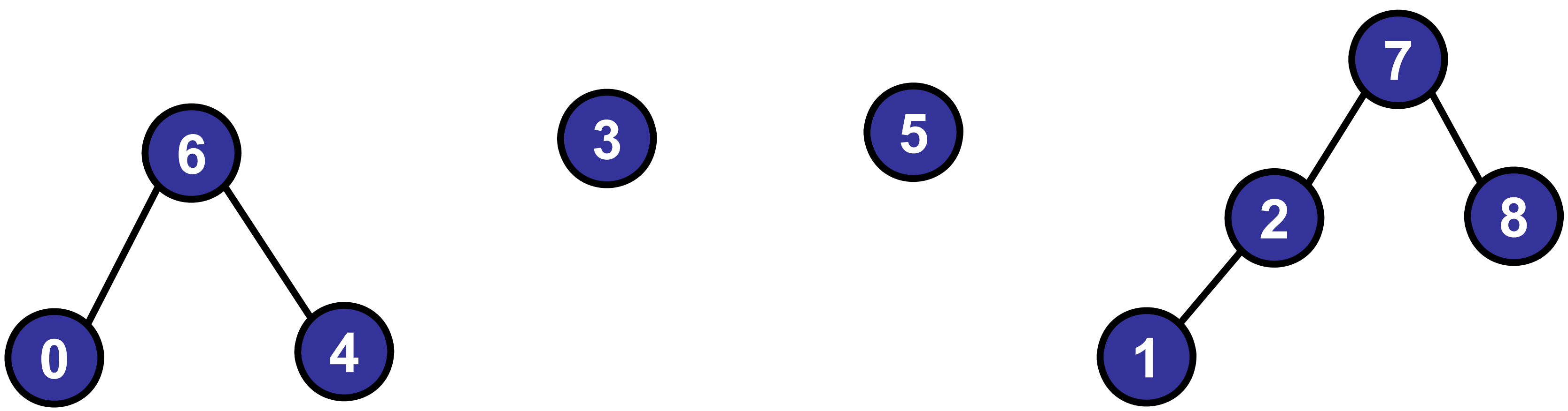
P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9
5-9	0	1	9	4	9	6	9	7	0	9
7-3	0	1	9	4	9	6	9	9	0	9
4-8	0	1	9	4	9	6	9	9	0	0
6-1	1	1	9	4	9	6	9	9	0	0



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



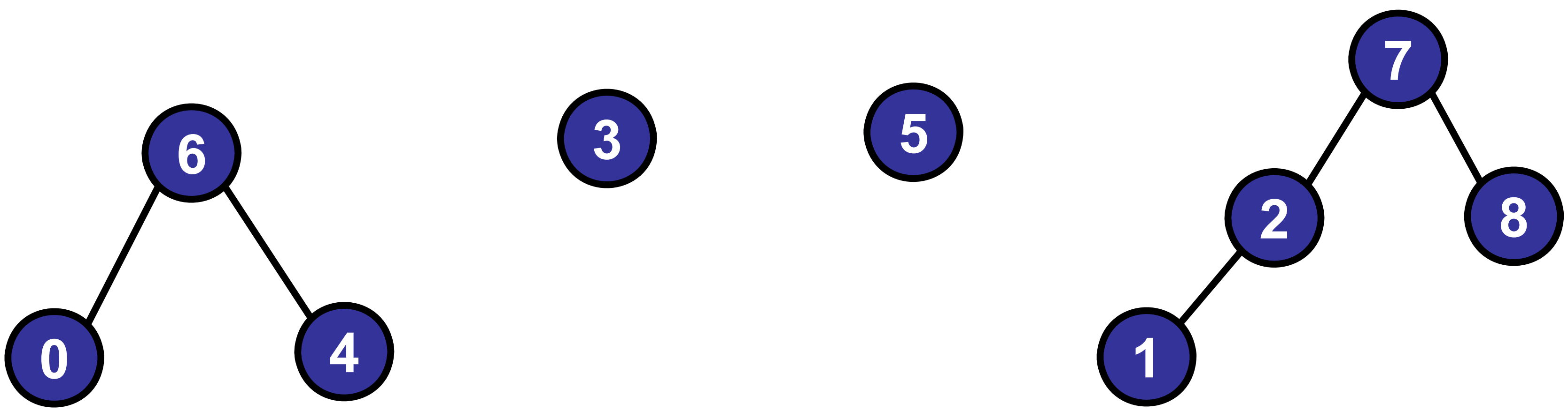
Running time of (Find, Union):

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);
```

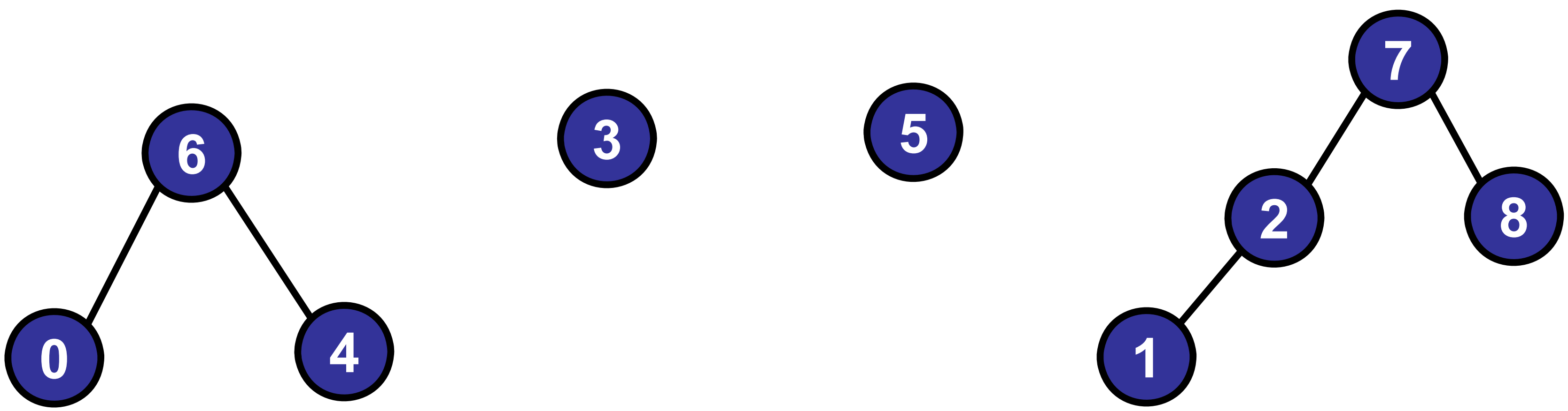
object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



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:

- 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

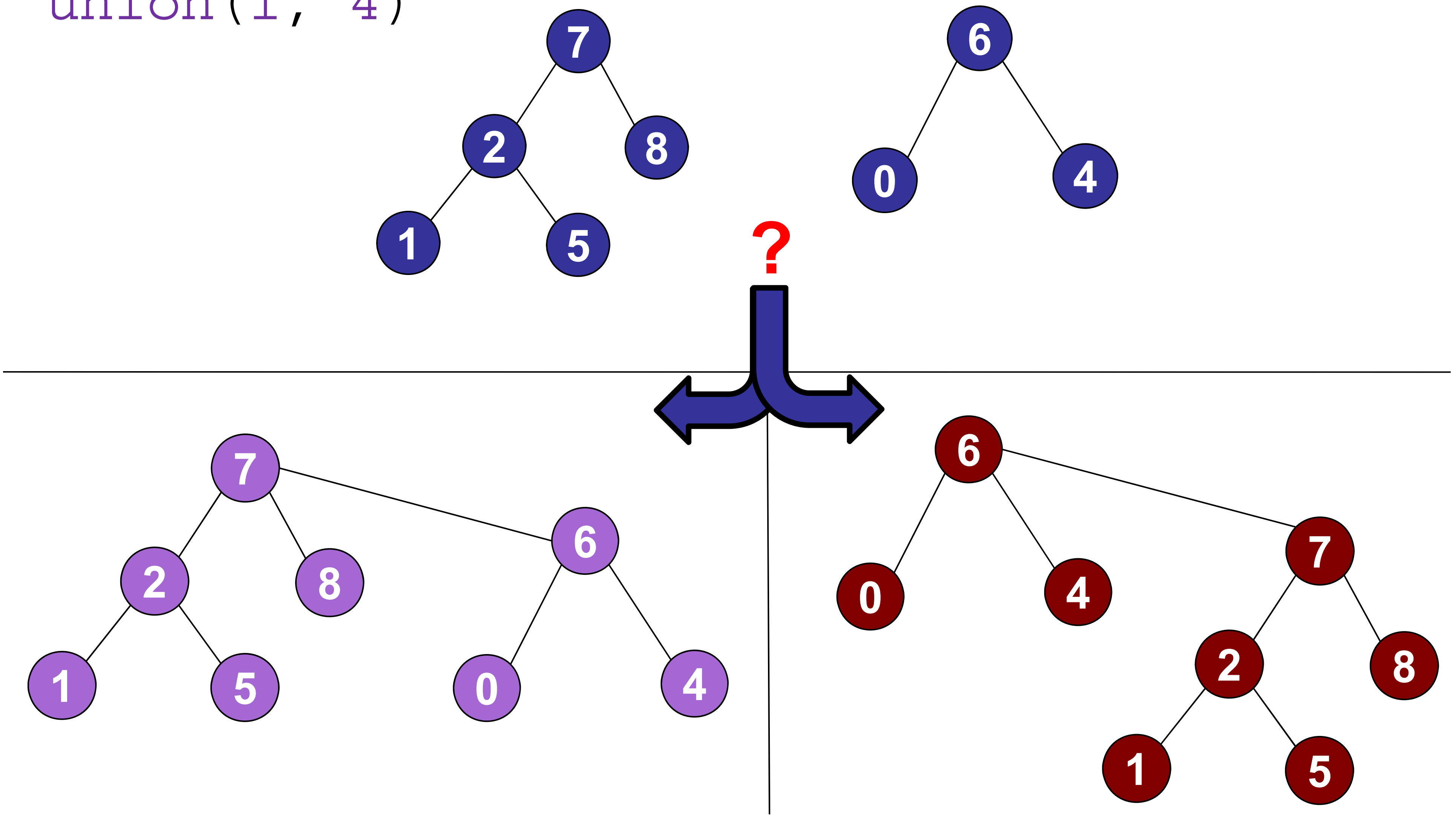
Part II: Disjoint Set

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

Weighted Union

Question: which tree should you make the root?

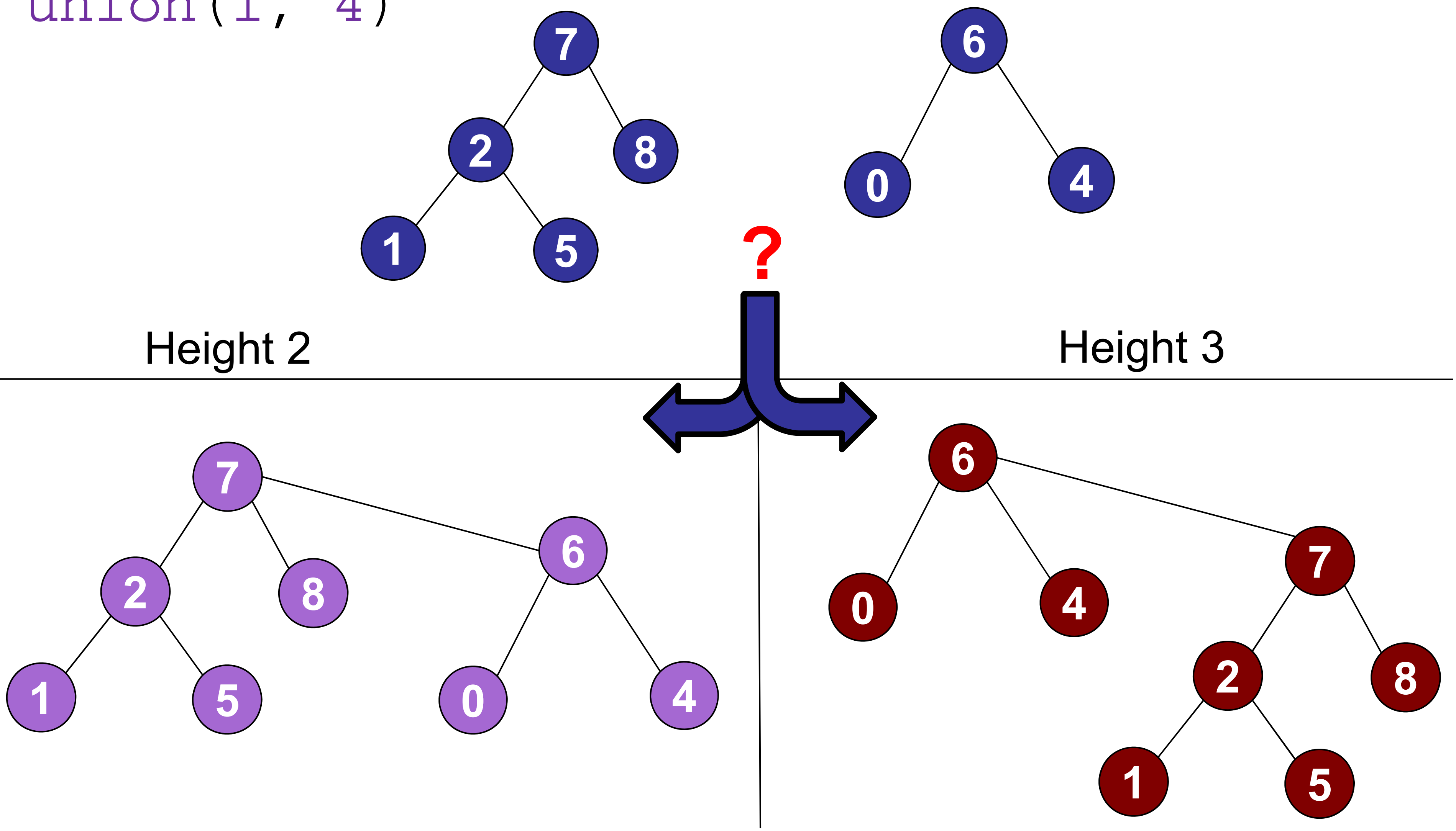
`union(1, 4)`



Weighted Union

Question: which tree should you make the root?

`union(1, 4)`



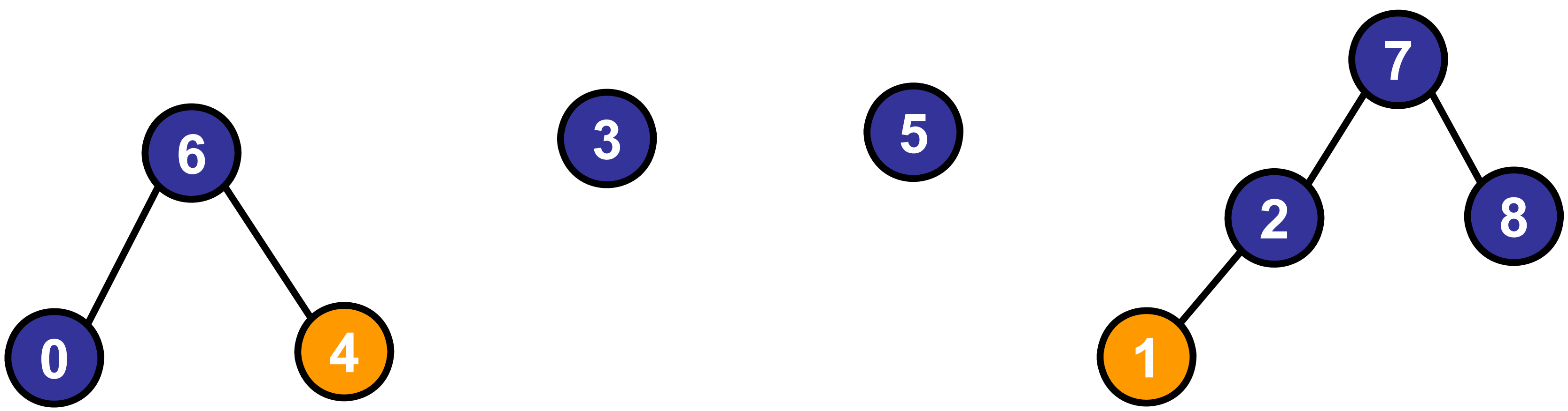
Weighted Union

```
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)`

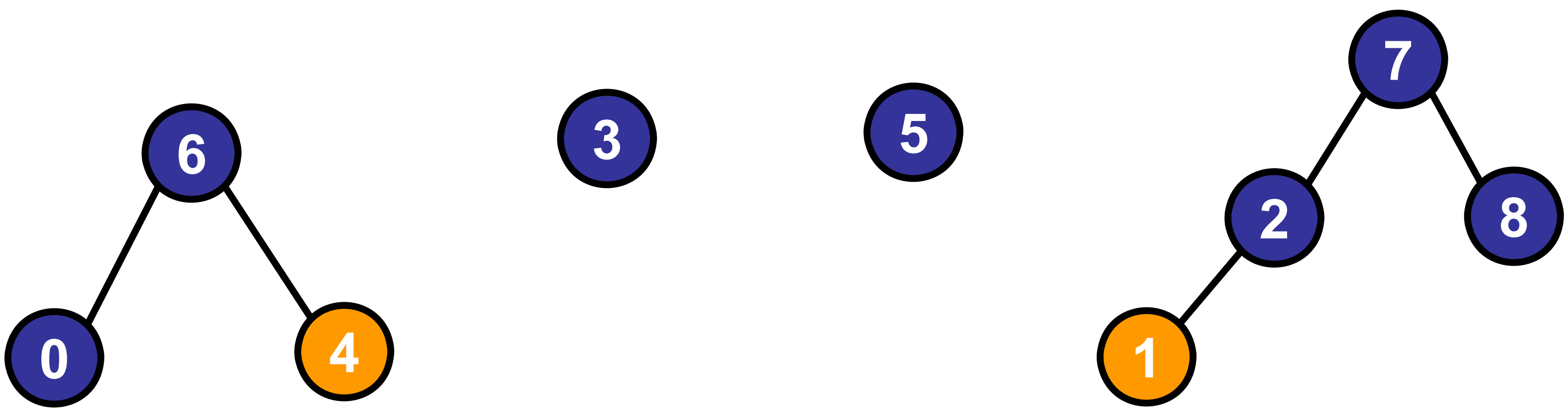
object	0	1	2	3	4	5	6	7	8
size	1	1	2	1	1	1	3	4	1
parent	6	2	7	3	6	1	6	7	7



Weighted Union

`union(1, 4)`

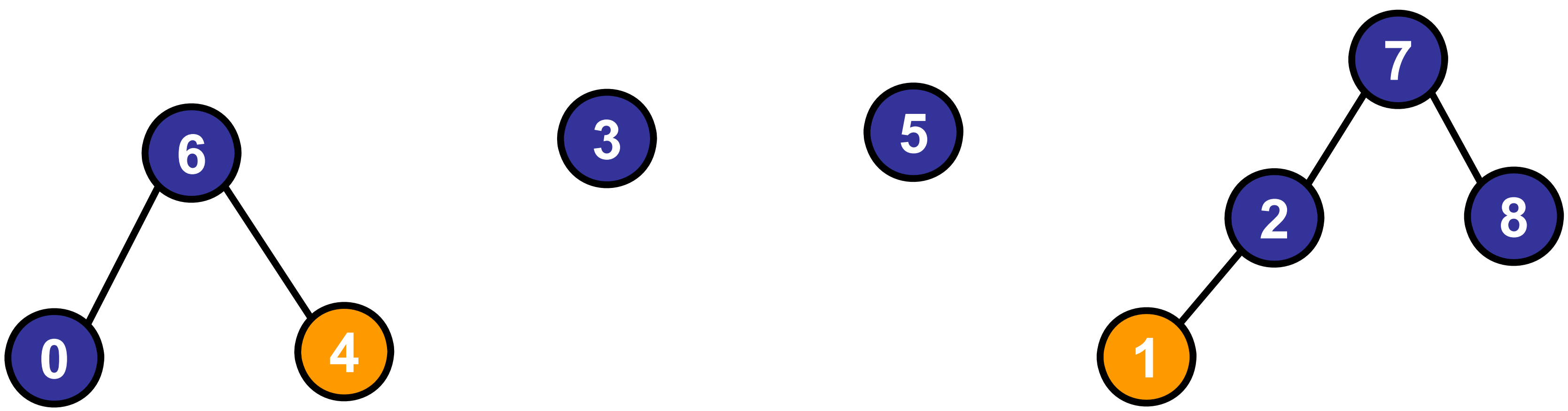
object	0	1	2	3	4	5	6	7	8
size	1	1	2	1	1	1	3	4	1
parent	6	2	7	3	6	1	6	7	7



Weighted Union

`union(1, 4)`

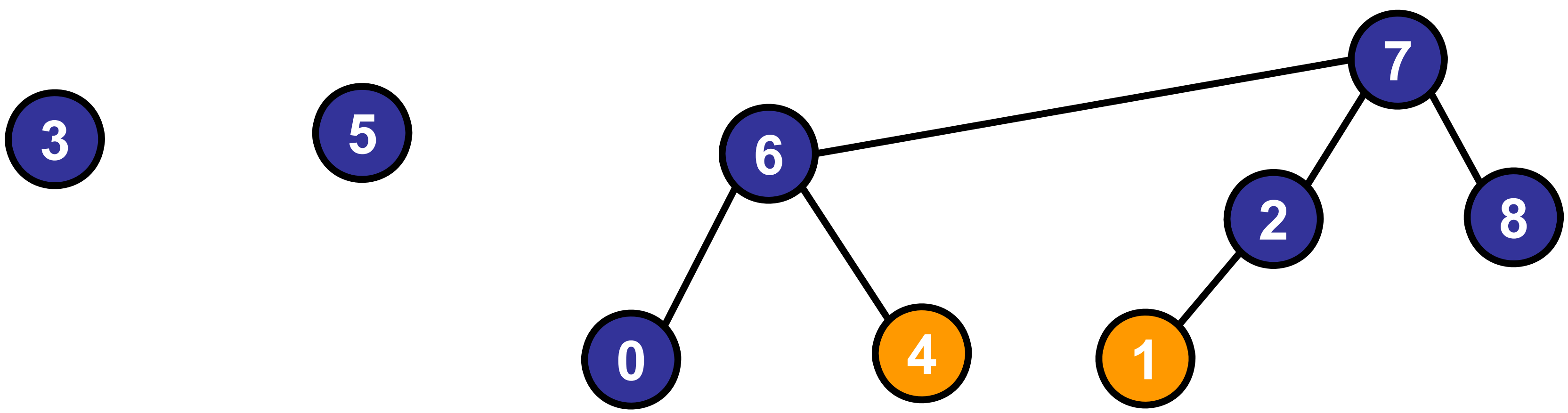
object	0	1	2	3	4	5	6	7	8
size	1	1	2	1	1	1	3	4	1
parent	6	2	7	3	6	1	6	7	7



Weighted Union

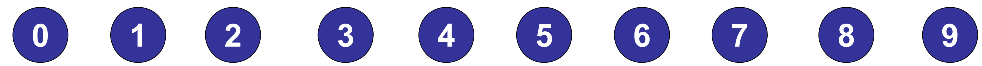
`union(1, 4)`

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



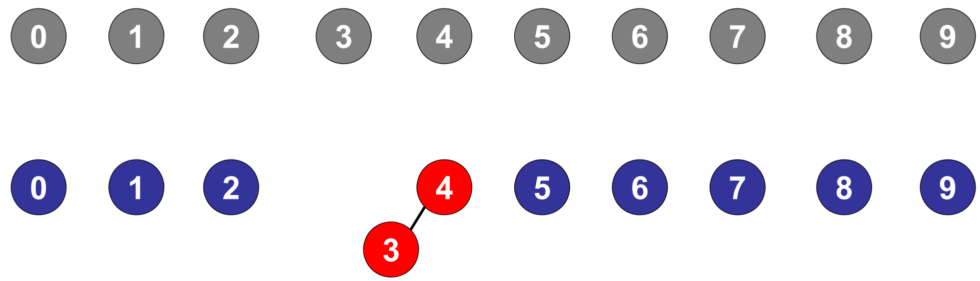
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9



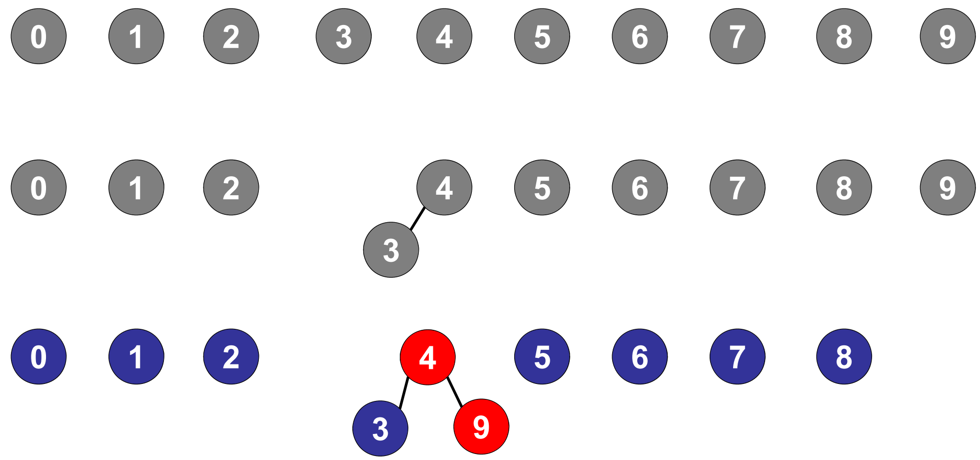
Example: Weighted Union

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



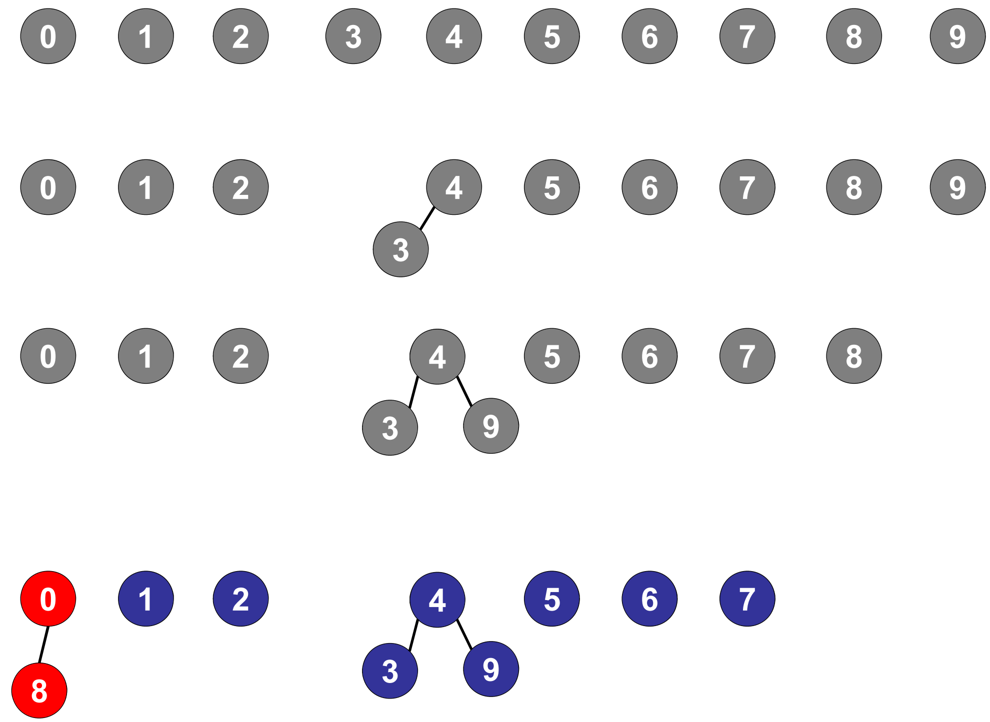
Example: Weighted Union

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



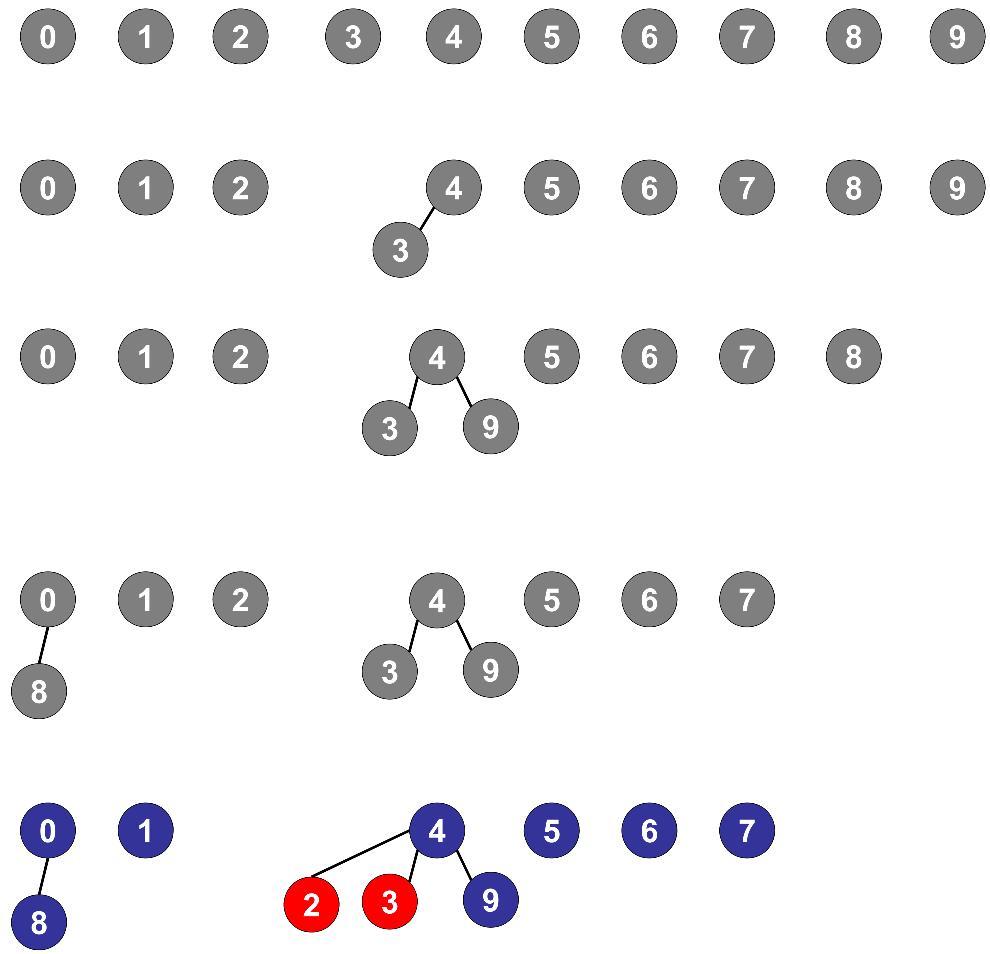
Example: Weighted Union

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



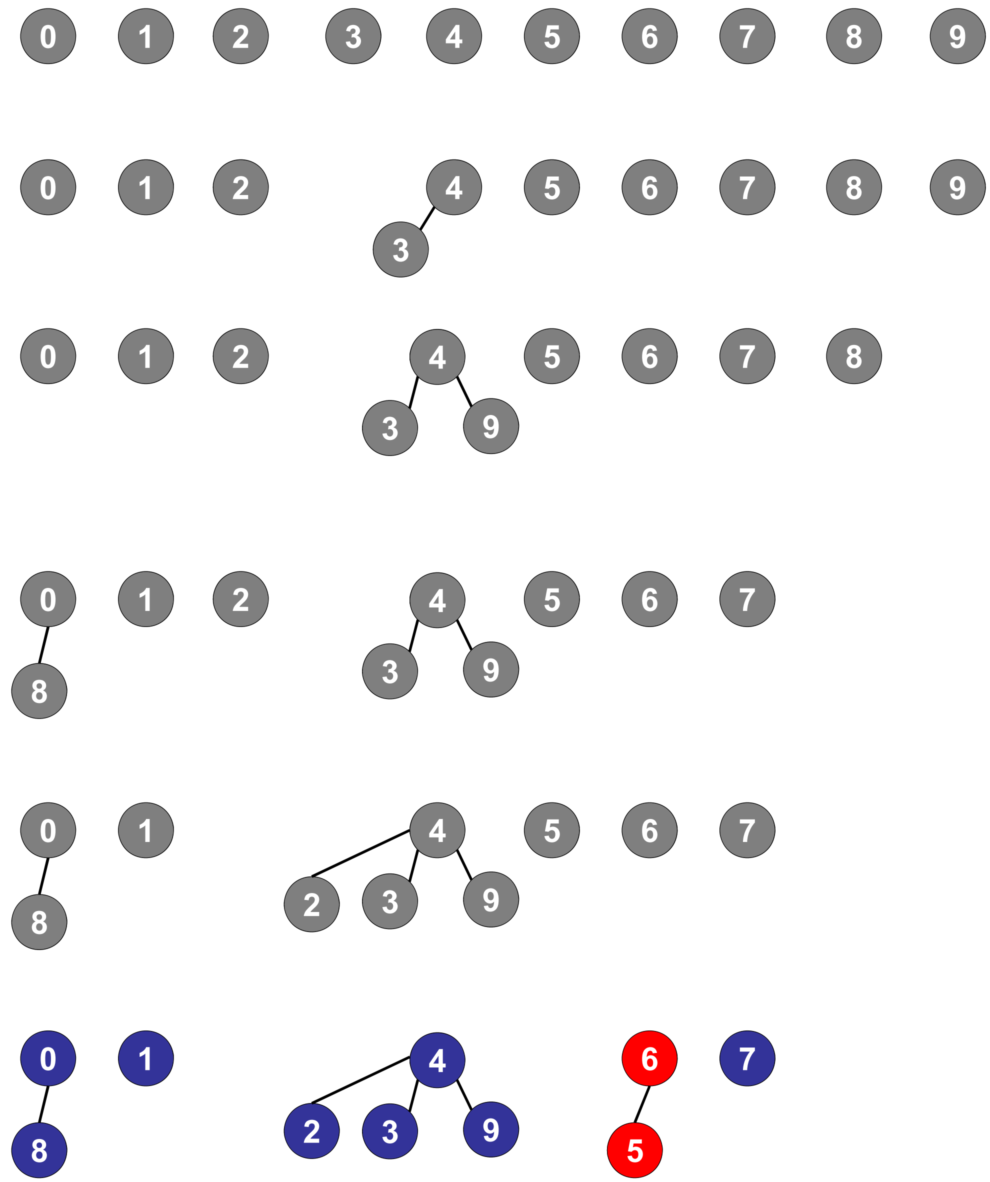
Example: Weighted Union

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



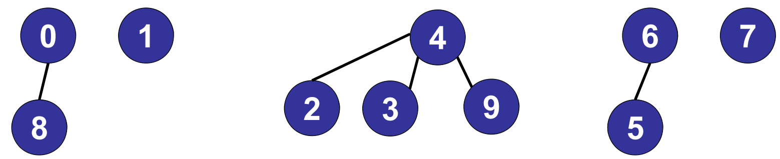
Example: Weighted Union

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



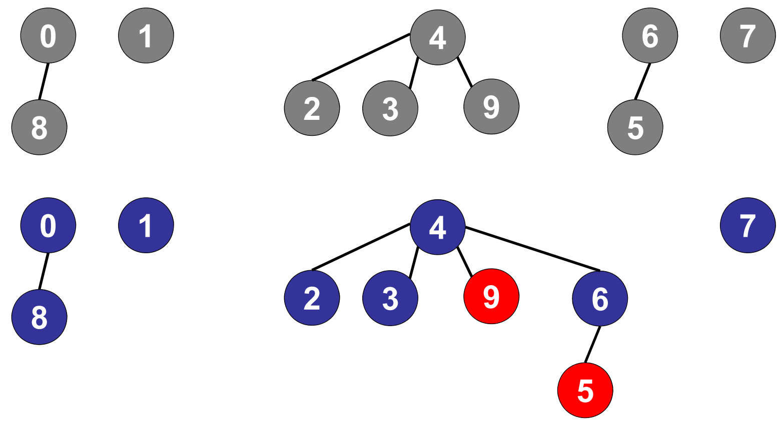
Example: Weighted Union

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



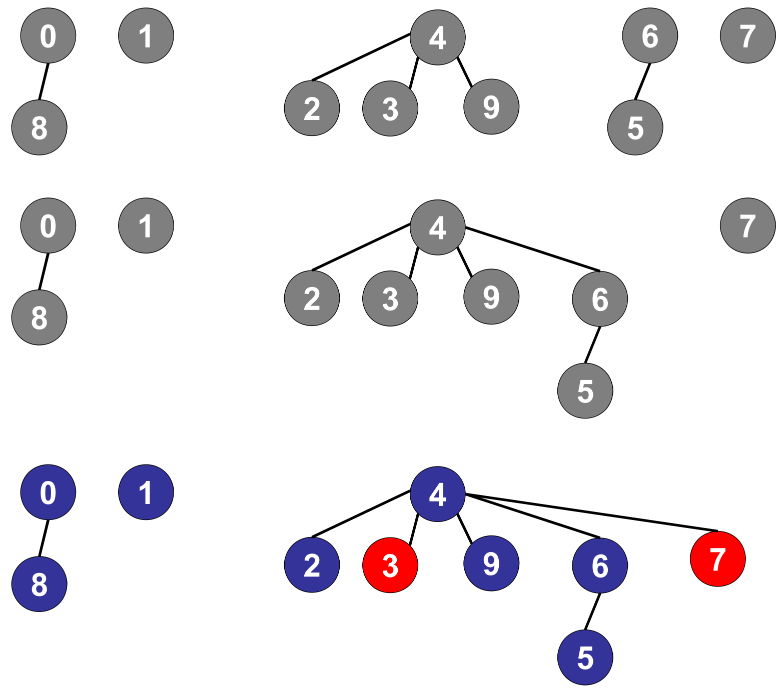
Example: Weighted Union

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



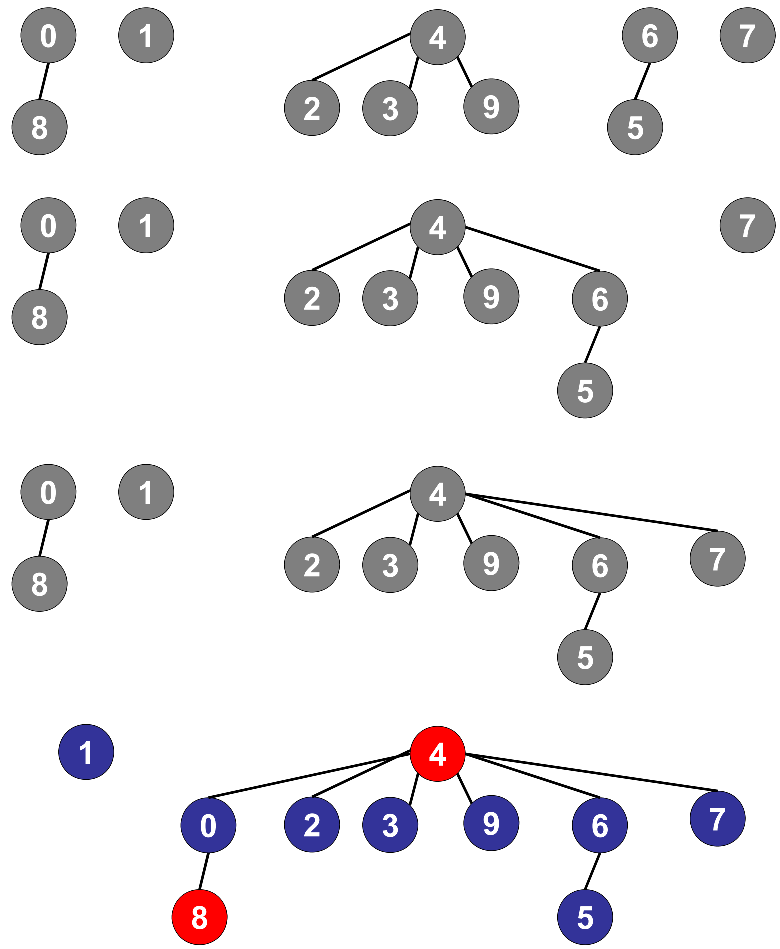
Example: Weighted Union

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



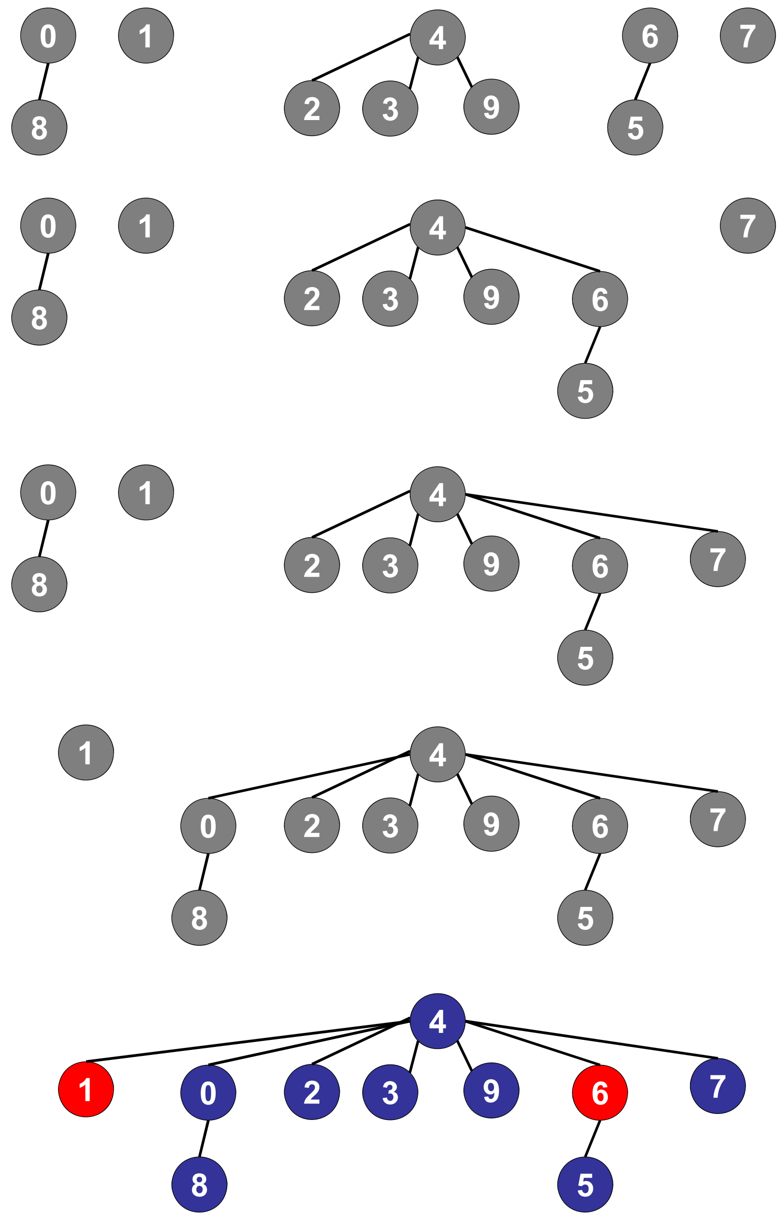
Example: Weighted Union

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



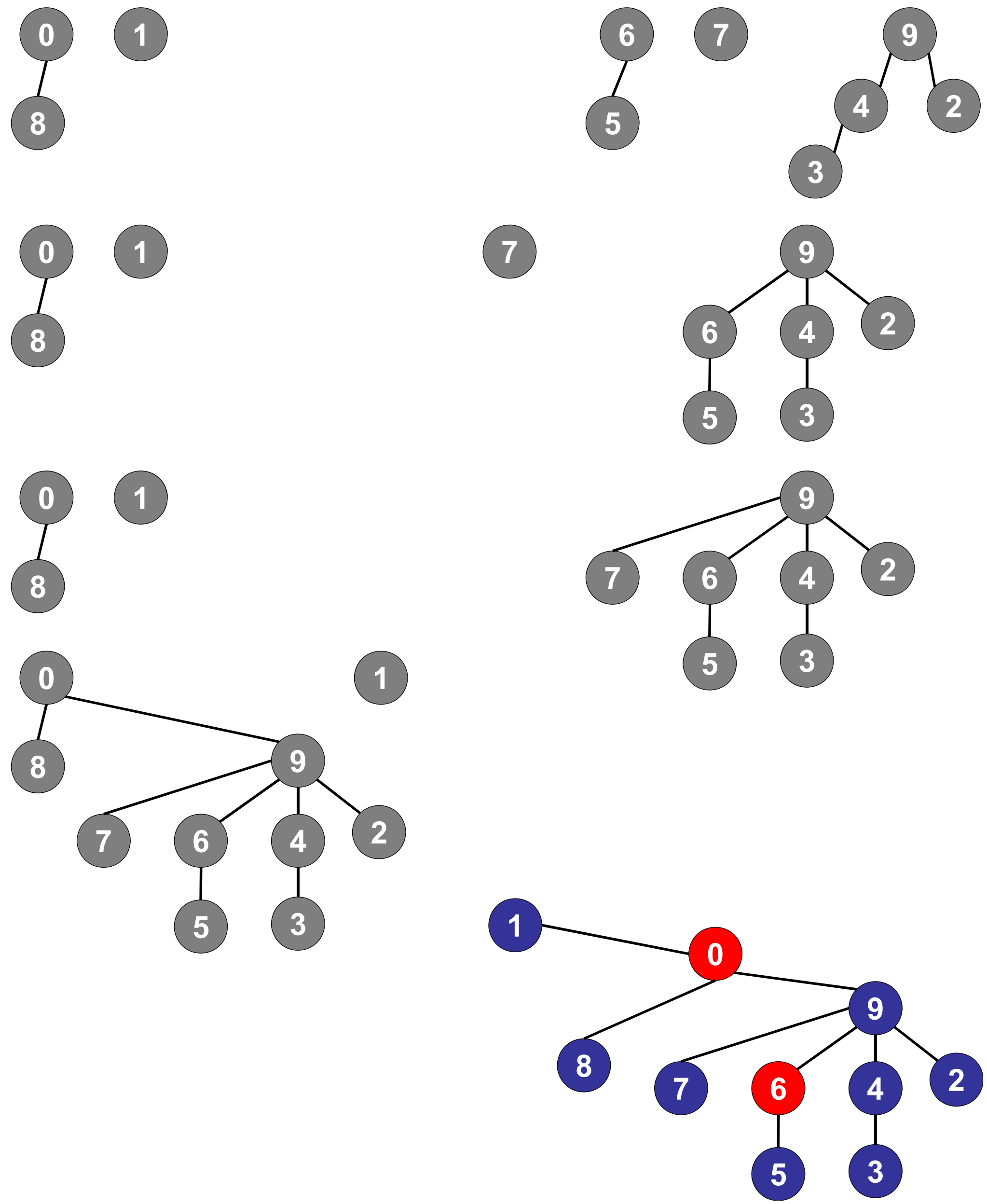
Example: Weighted Union

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



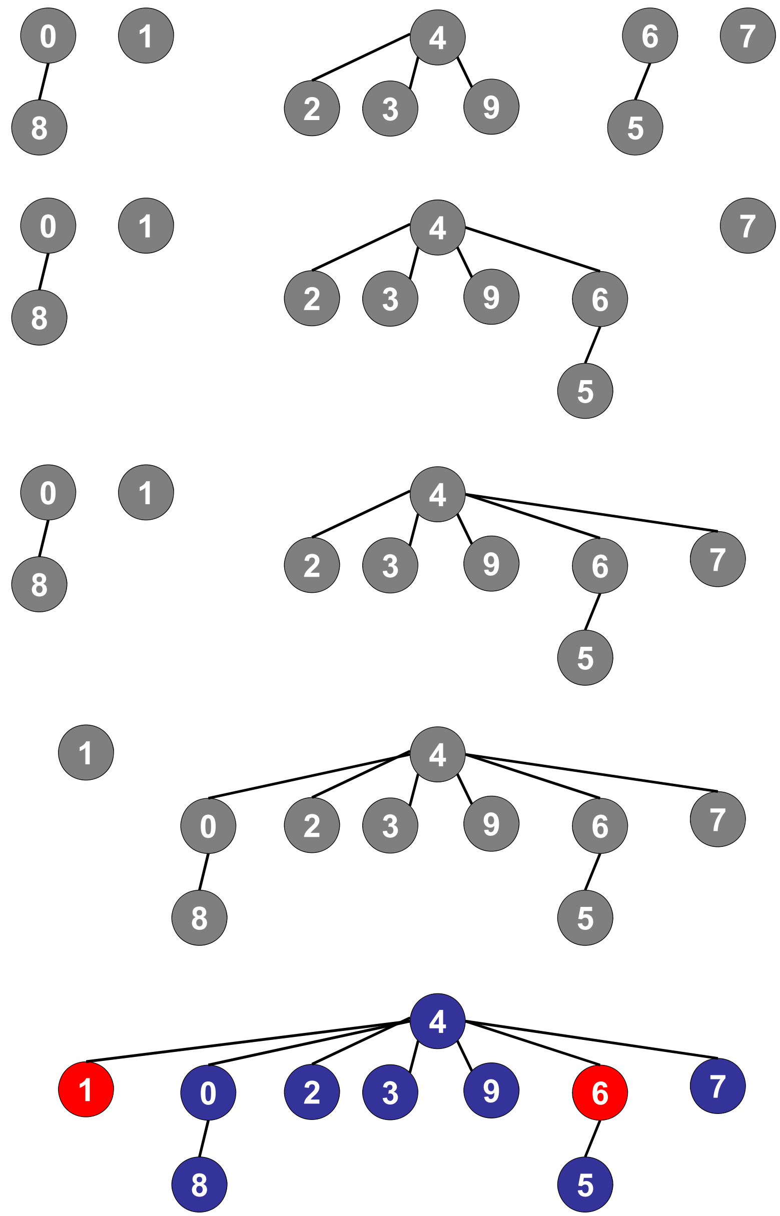
Example: (Unweighted) Quick Union

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?

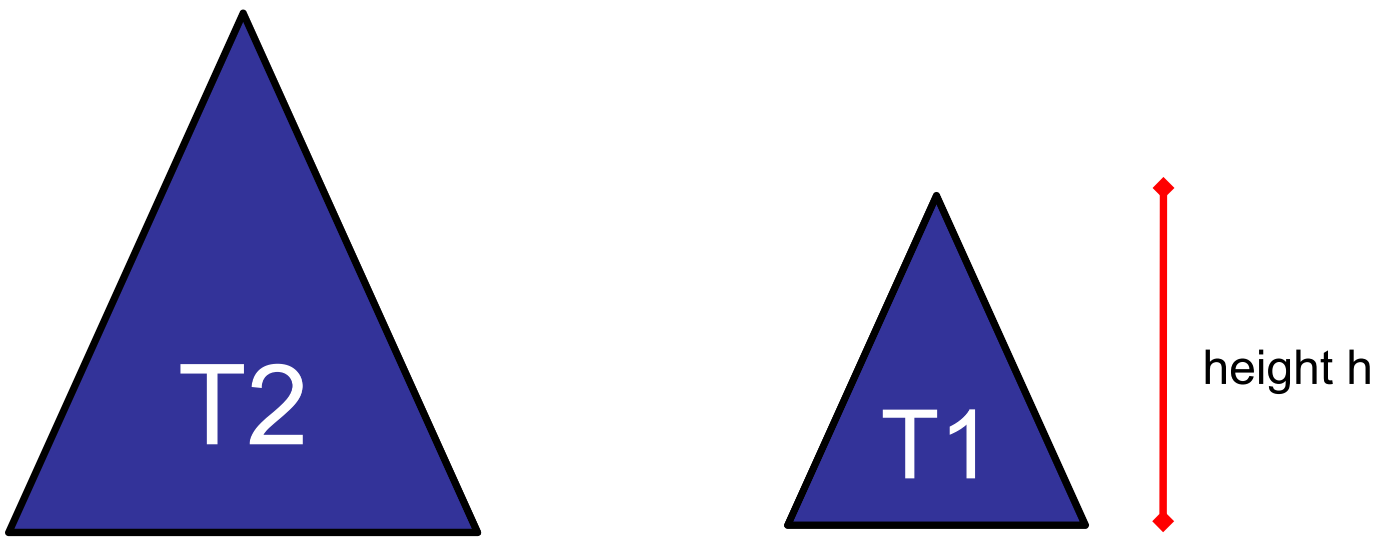
1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n \log n)$
5. $O(n^2)$
6. None of the above.

Weighted Union

Analysis:

- Tree T1 is merged with Tree T2.
- When does the depth of a node in T1 increase?

Only if: size(T2) >= size(T1) ➔ link T1 to T2 ➔ T1 is one level deeper

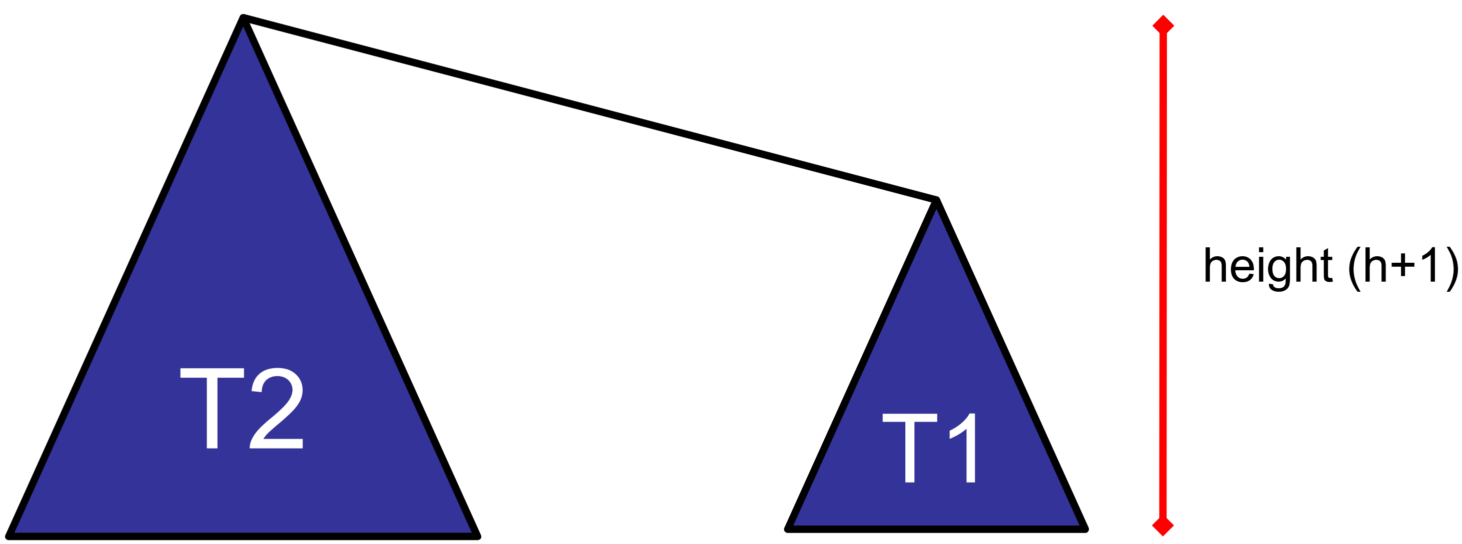


Weighted Union

Analysis:

- Tree T1 is merged with Tree T2.
- When does the depth of a node in T1 increase?

Only if: size(T2) >= size(T1) ➔ T1 is one level deeper

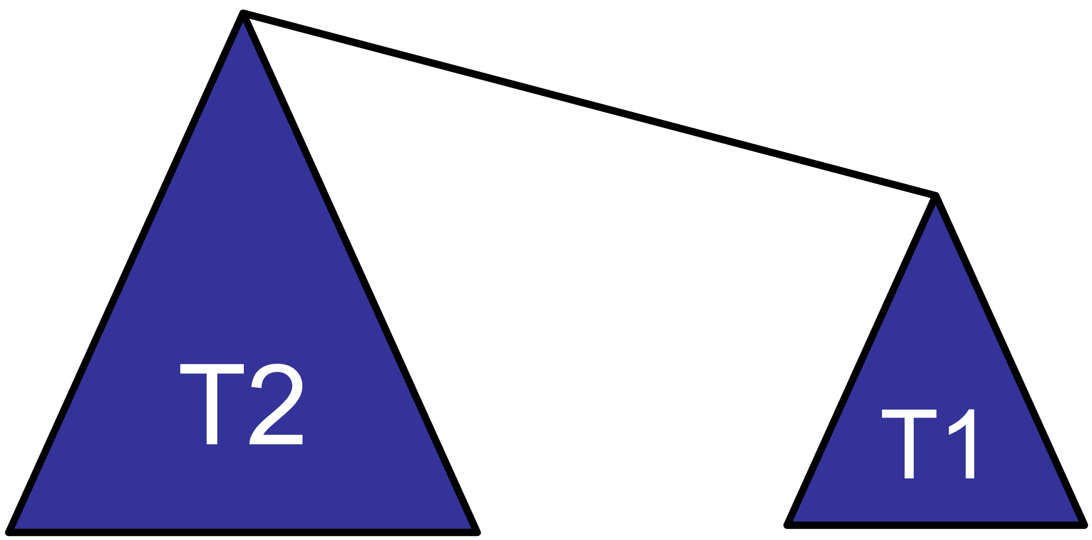


Weighted Union

Analysis:

- Tree T1 is merged with Tree T2.
- When does the depth increase?

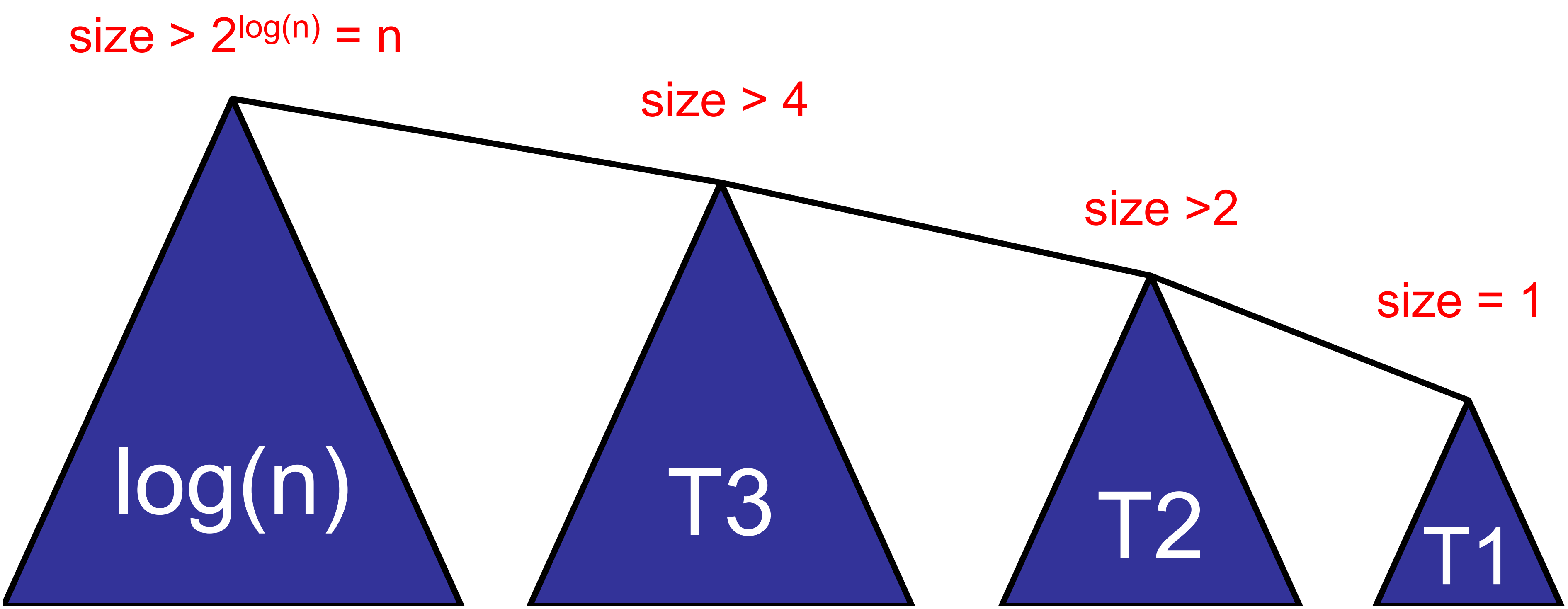
size(T1 + T2) > 2size(T1):



Weighted Union

Assume T1 is merged with a tree of height log(n).

size(Tj + Tk) > 2size(Tk):



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

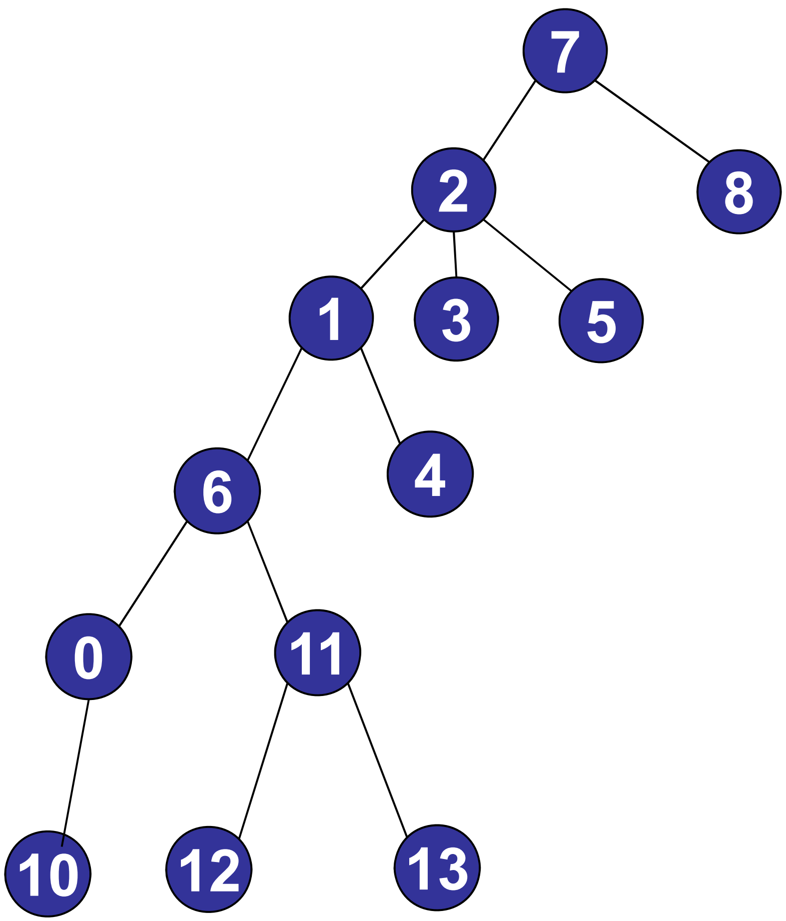
Weighted-union is faster:

- 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)

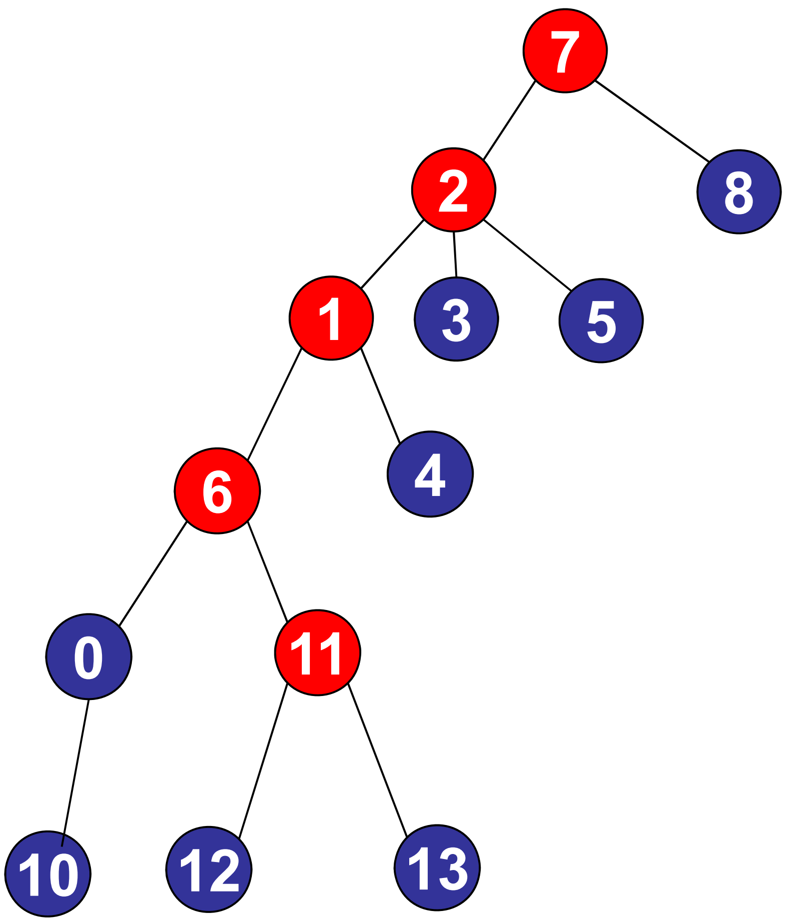
Path Compression

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



Path Compression

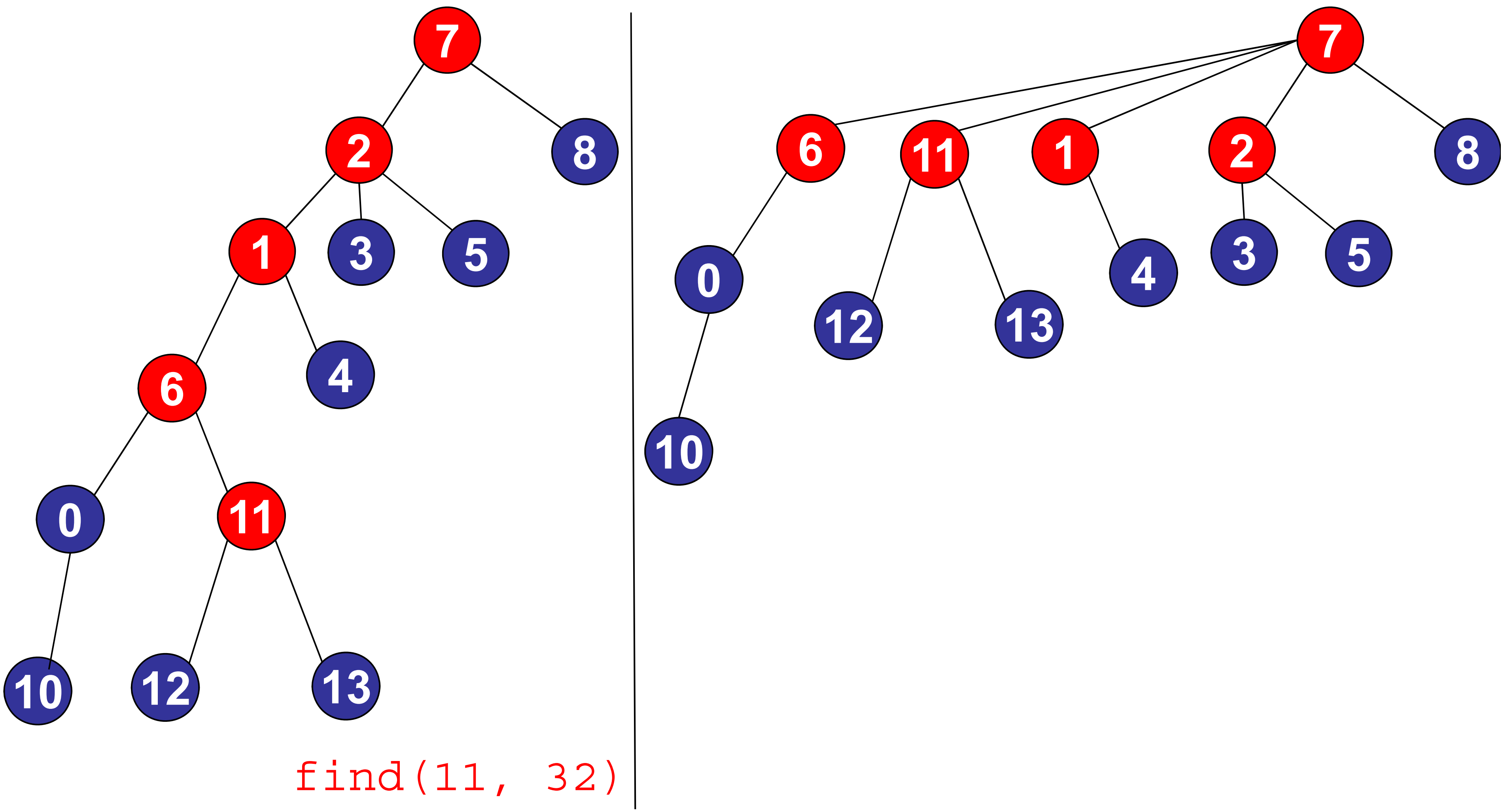
After finding the root: set the parent of each traversed node to the root.



find(11, 32)

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;
}
```

OR: 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:

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.

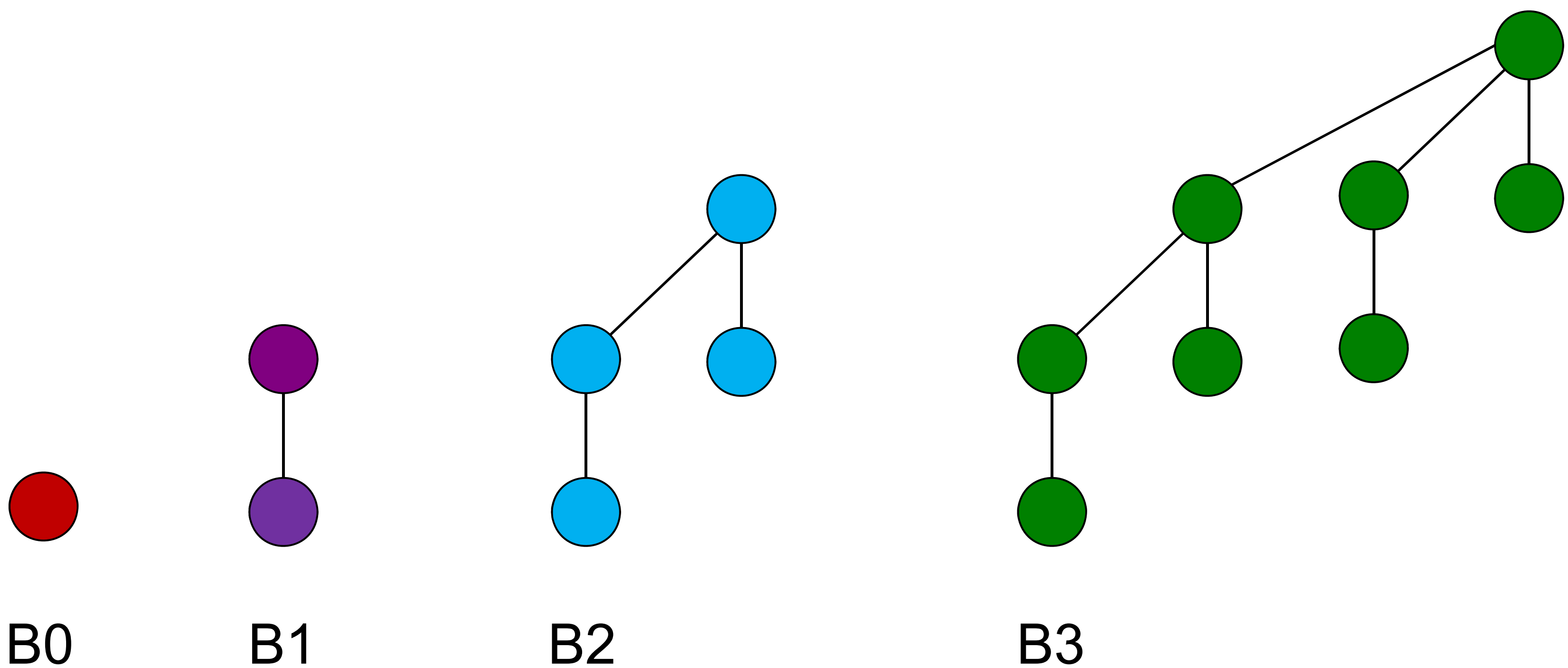
	find	**union**
quick-find	O(1)	O(n)
quick-union	O(n)	O(n)
weighted-union	O(log n)	O(log n)
weighted-union with path-compression	$\alpha(m, n)$	$\alpha(m, n)$

Union-Find Summary

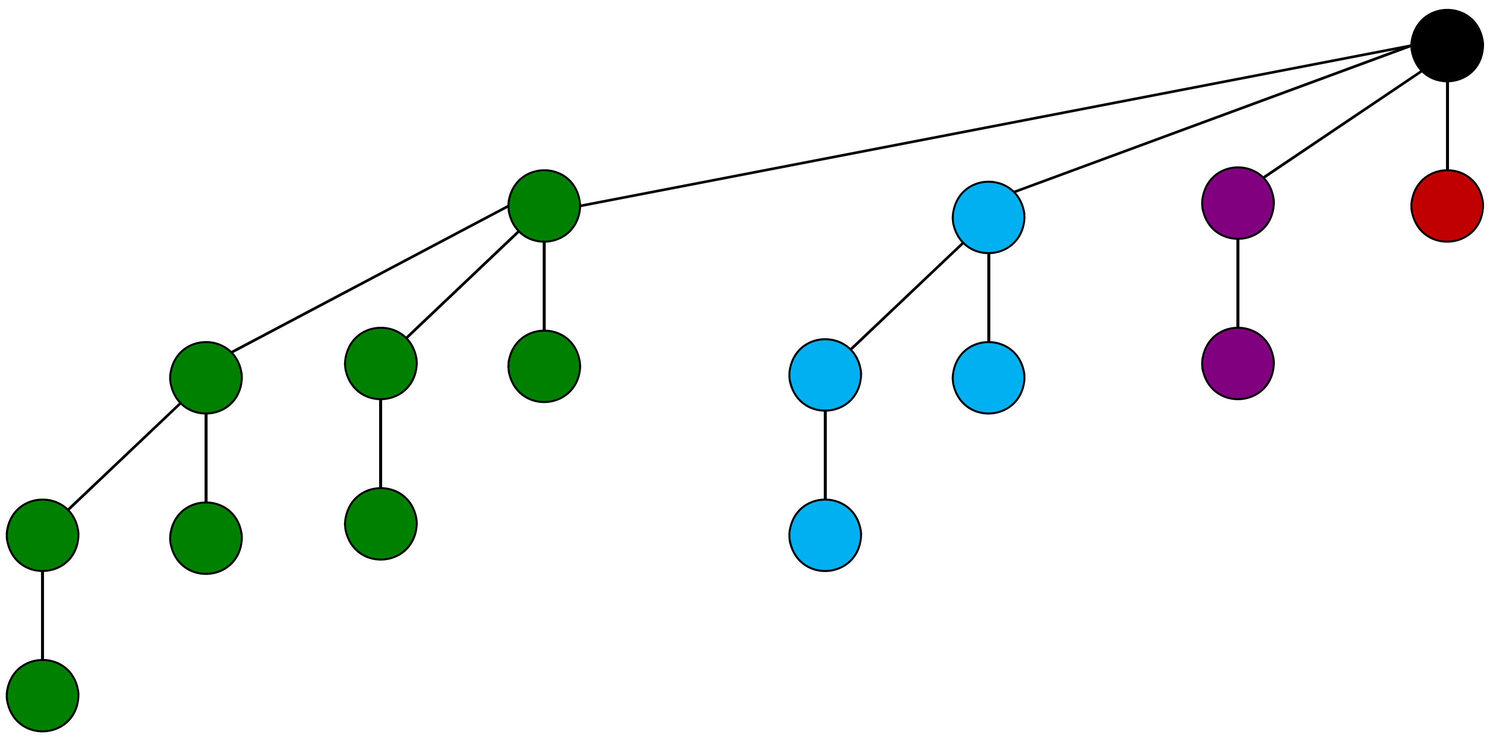
Path Compression **without** weighted union?

	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	α(m, n)	α(m, n)

Binomial Trees:

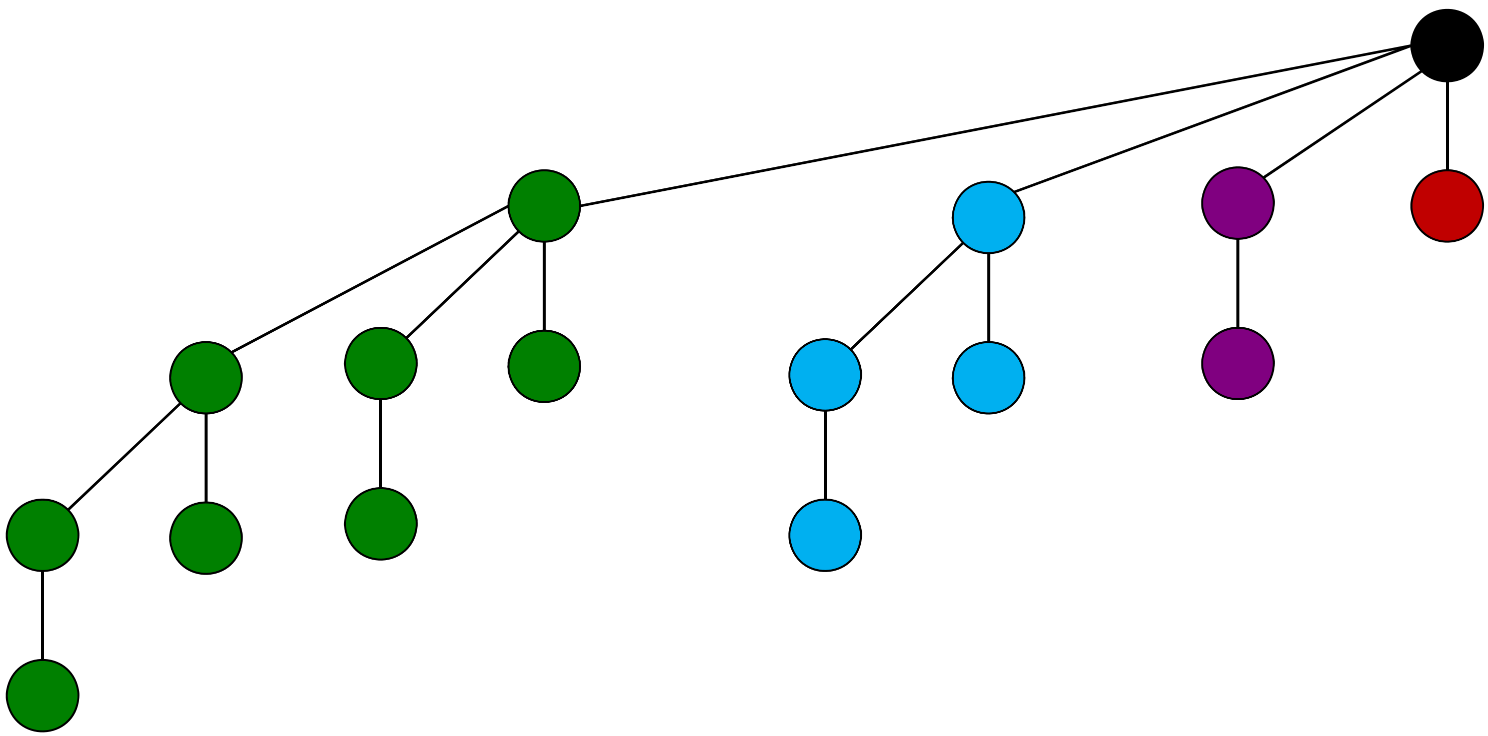


Binomial Trees:



B4 = (root + B0 + B1 + B2 + B3) = (B3 + B3)

Binomial Trees:



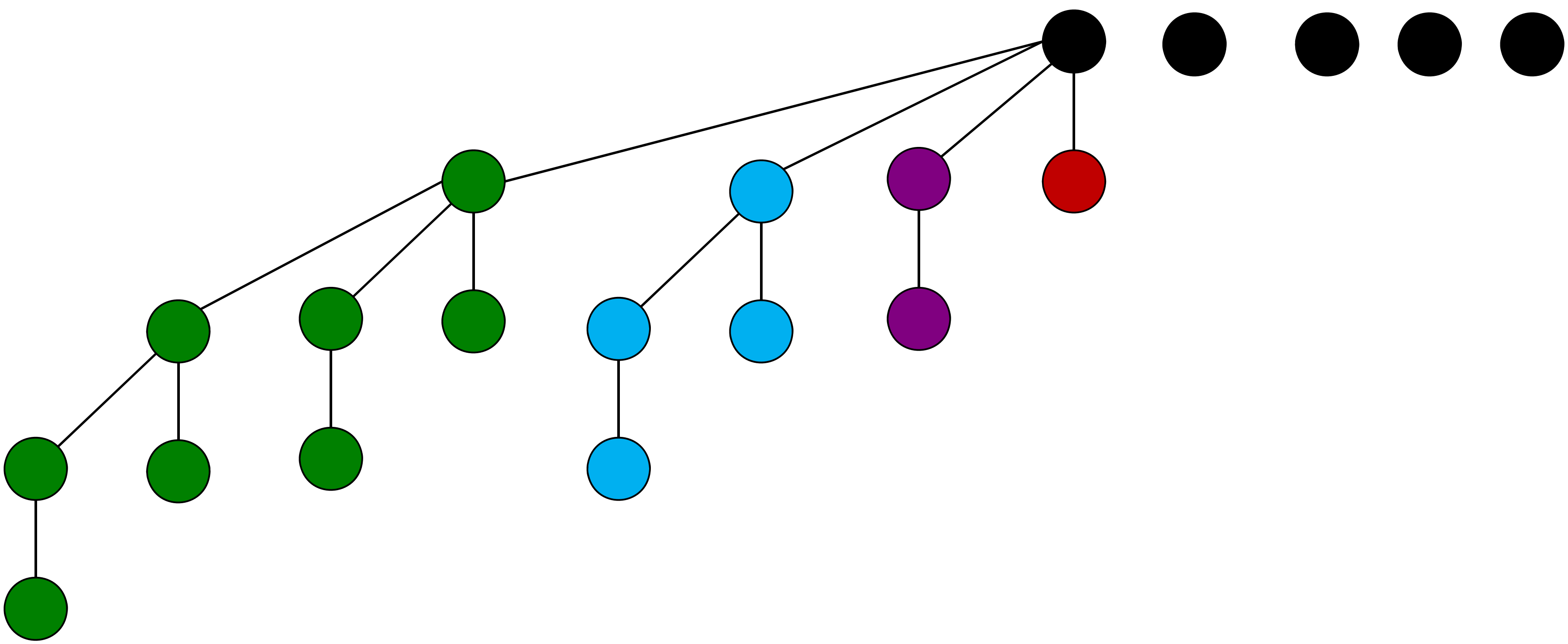
size(Bk) = $\Theta(2^k)$

height(Bk) = k-1

Union Find Example:

Step 1: Build Binomial tree using union operations.

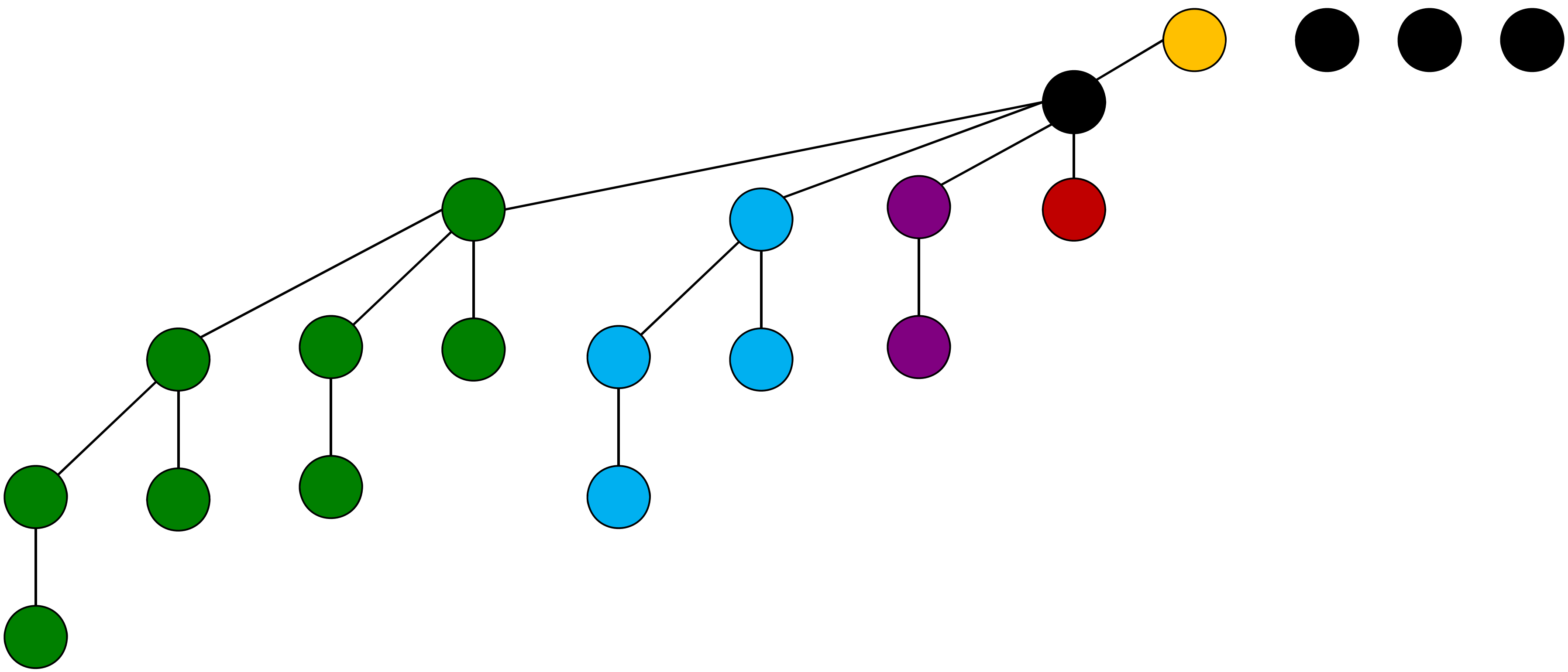
- Leave some extra objects free.



Union Find Example:

Step 1: Build Binomial tree using union operations.

Step 2: Union: create new root [O(1)]

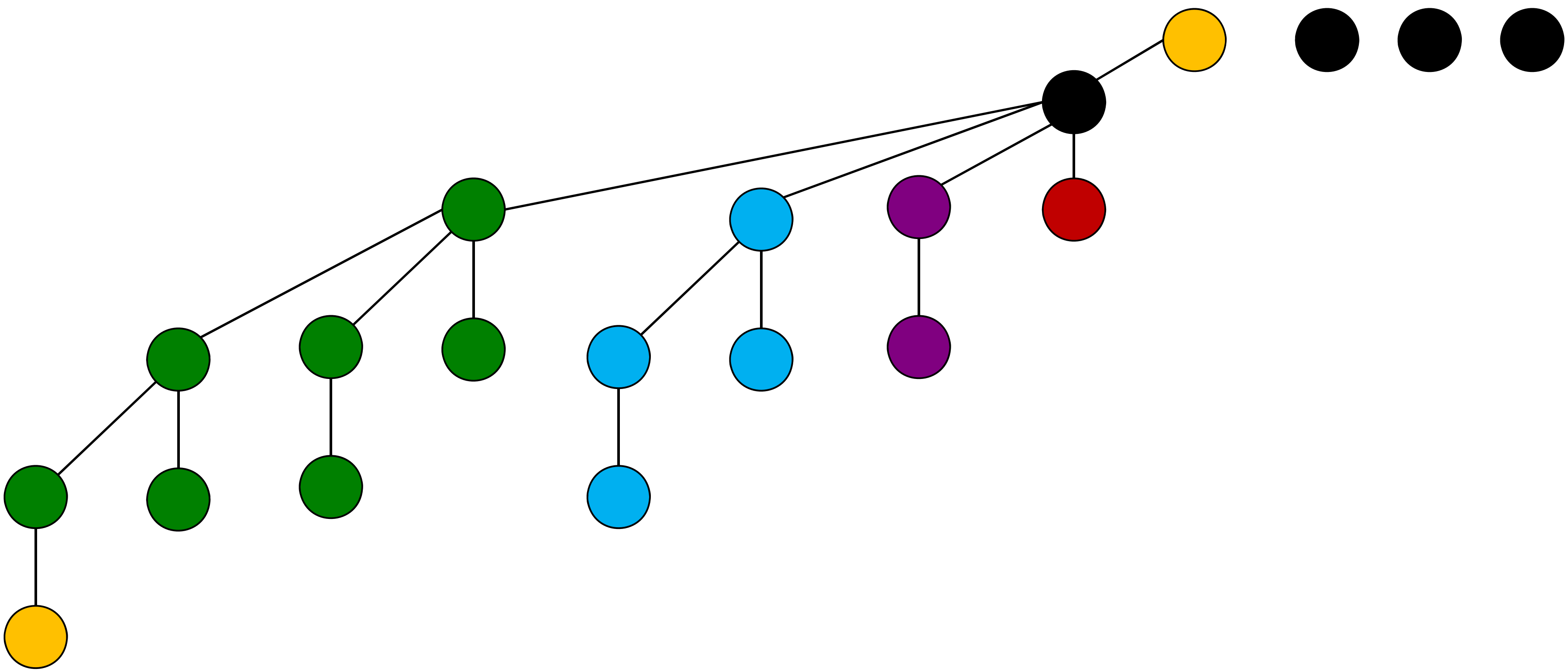


Union Find Example:

Step 1: Build Binomial tree using union operations.

Step 2: Union: create new root [O(1)]

Step 3: Find deepest leaf [O(log n)]



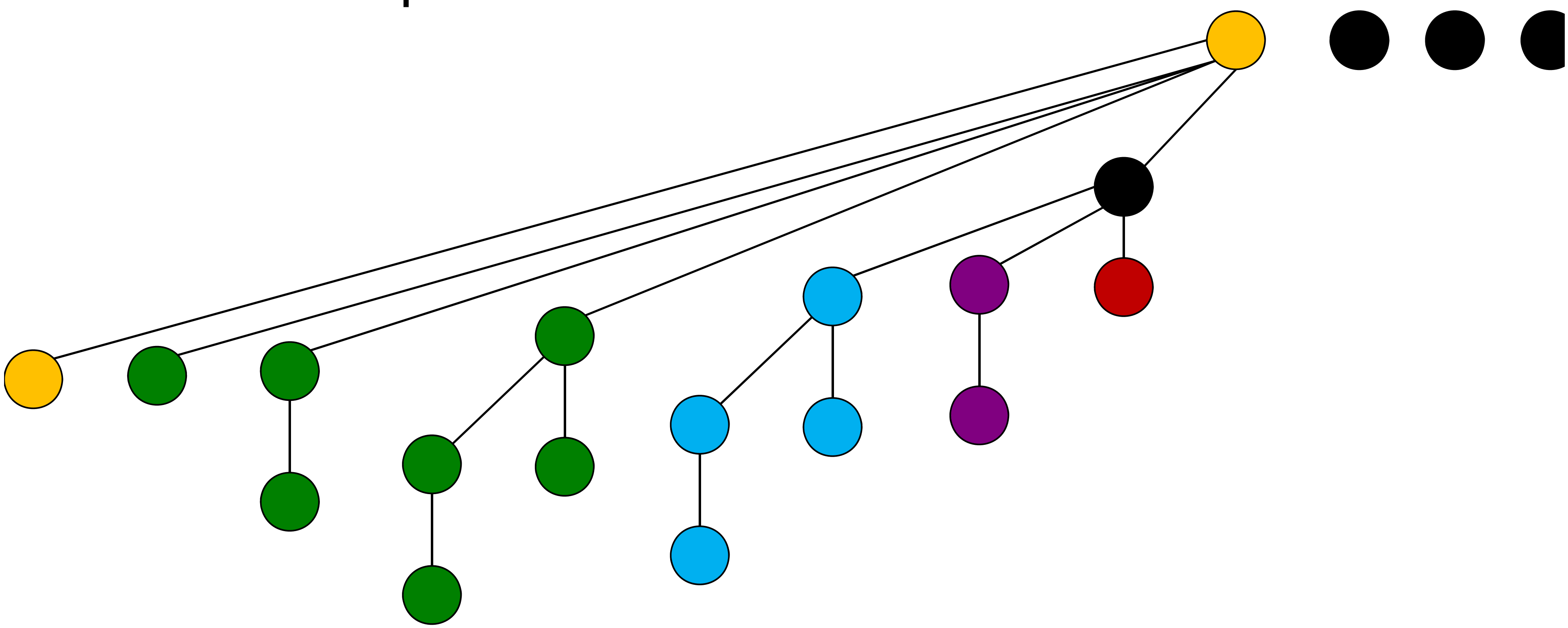
Union Find Example:

Step 1: Build Binomial tree using union operations.

Step 2: Union: create new root [O(1)]

Step 3: Find deepest leaf [O(log n)]

- Path compression…



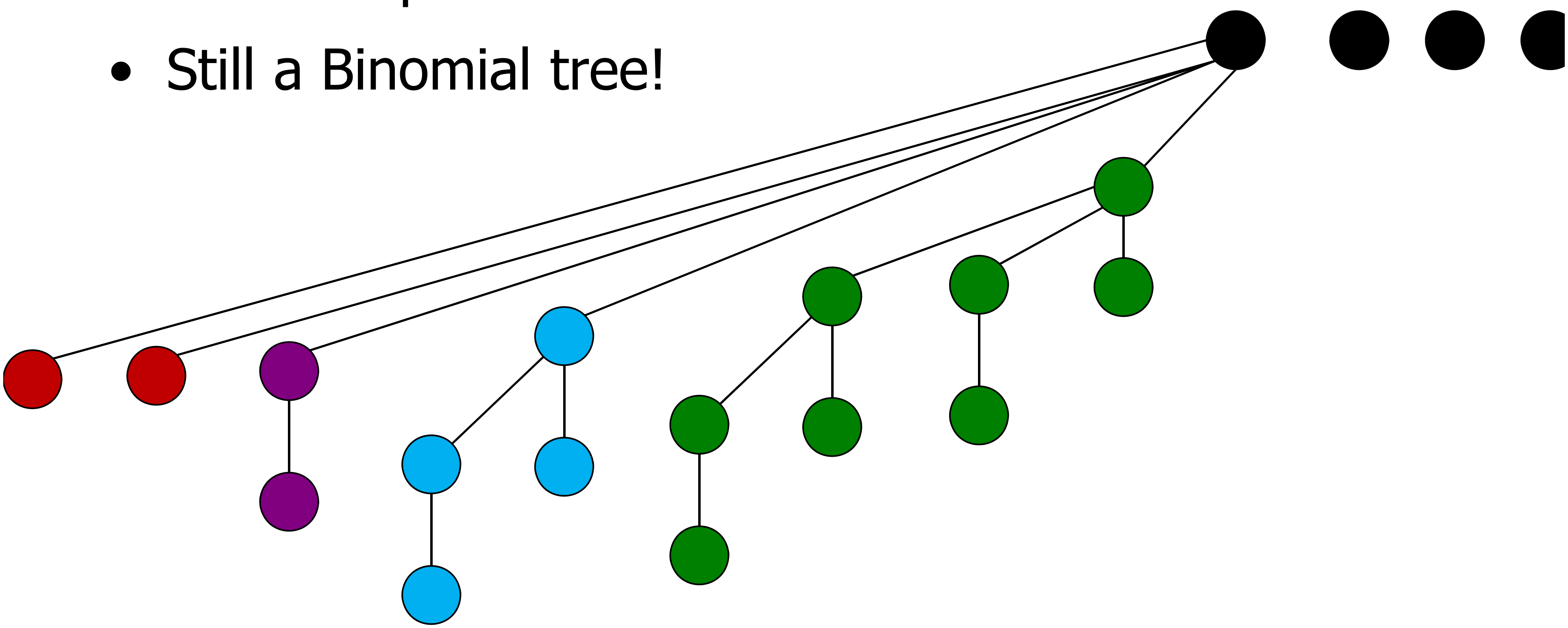
Union Find Example:

Step 1: Build Binomial tree using union operations.

Step 2: Union: create new root [O(1)]

Step 3: Find deepest leaf [O(log n)]

- Path compression…
- Still a Binomial tree!



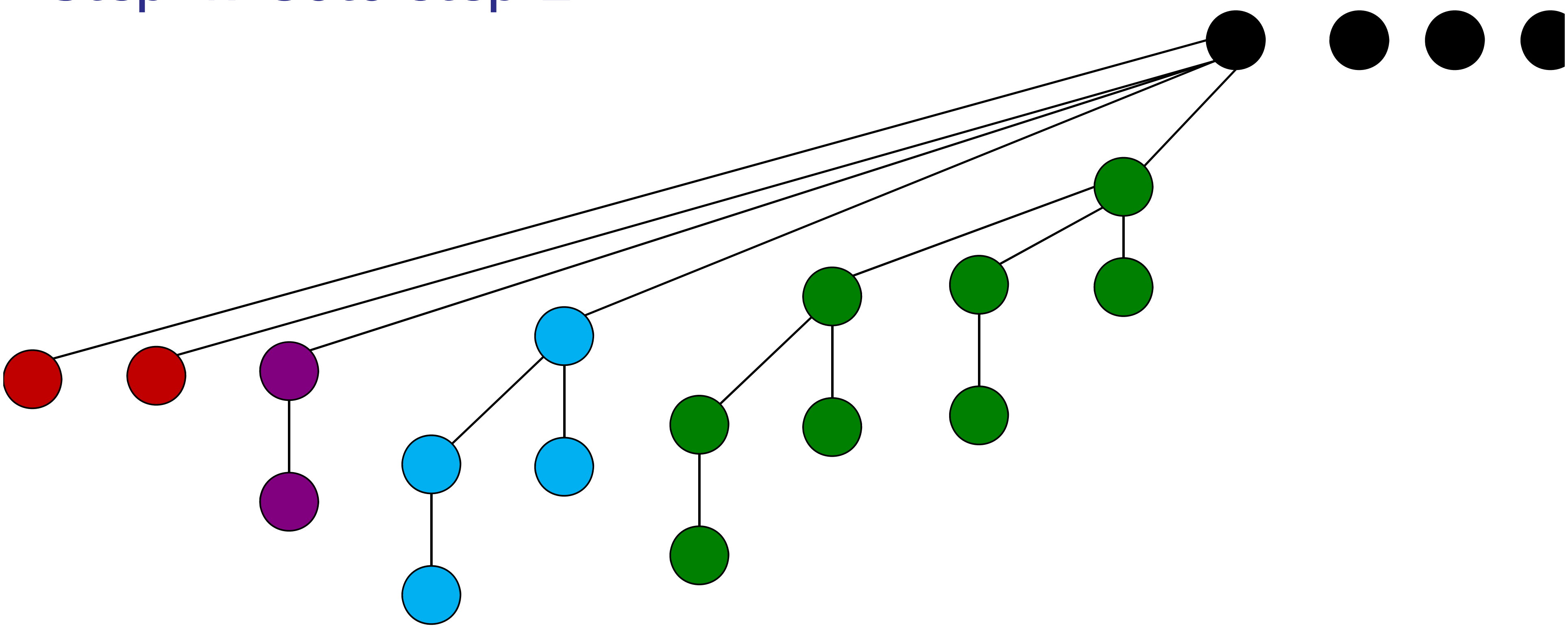
Union Find Example:

Step 1: Build Binomial tree using union operations.

Step 2: Union: create new root [O(1)]

Step 3: Find deepest leaf [O(log n)]

Step 4: Goto step 2.



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 results: 2013--present

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.

Applications

Many applications:

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

Applications

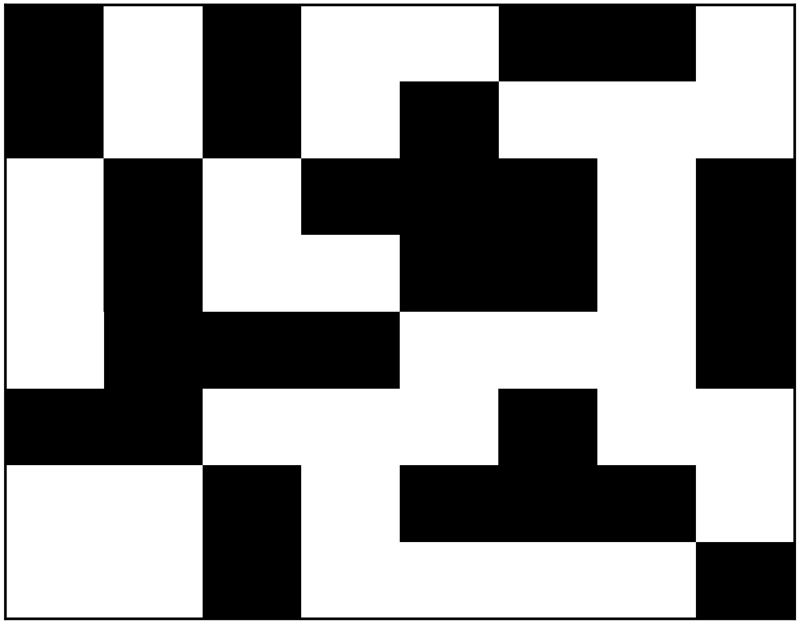
Many applications:

- Programming languages
  - Hinley-Milner polymorphic type inference
  - Equivalence of finite state automata
  - Image processing in Matlab
- Physics:
  - Hoshen-Kopelman algorithm
  - Percolation
  - Conductance / insulation

Percolation

Physical system:

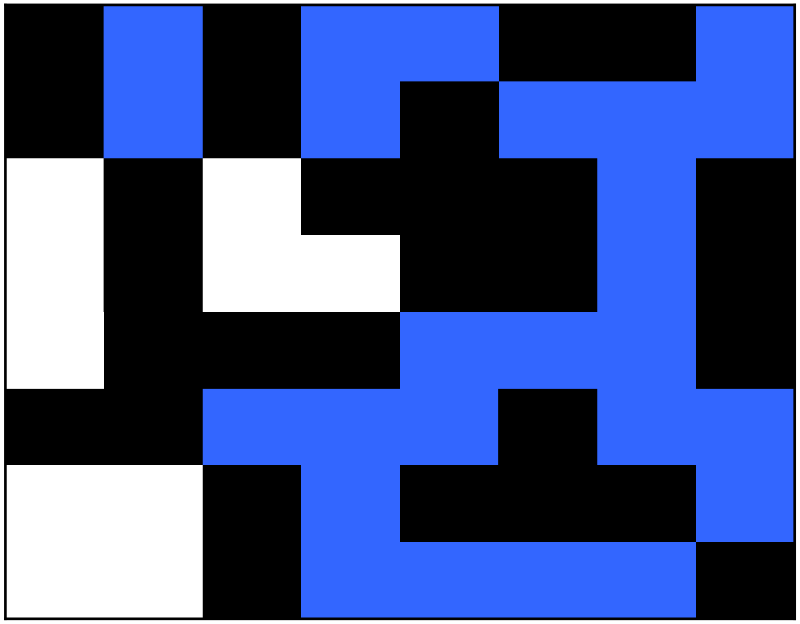
- n-by-n grid
- Each site open with probability p
- Are the top and bottom connected?



Percolation

Physical system:

- n-by-n grid
- Each site open with probability p
- Are the top and bottom connected?



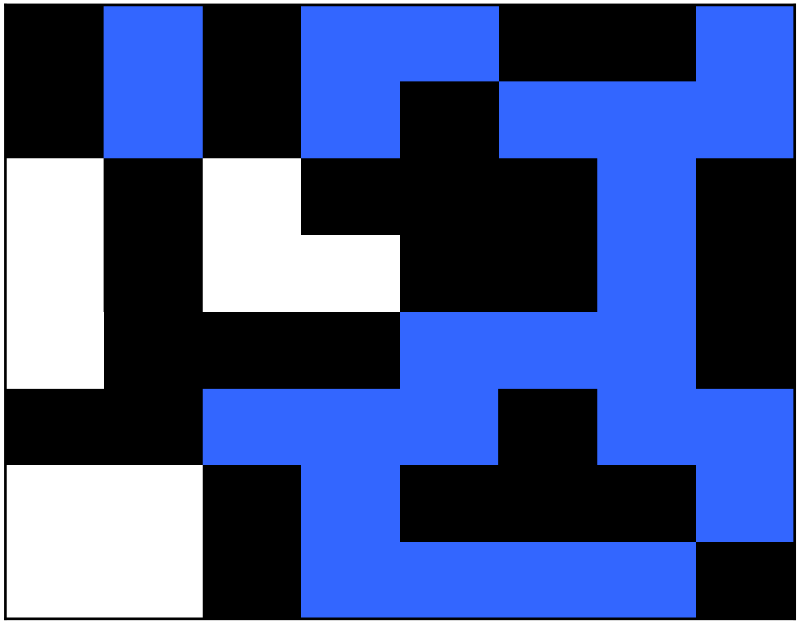
Percolation

Physical system:

– Sharp threshold p*:

Via simulation: p* = 0.592746

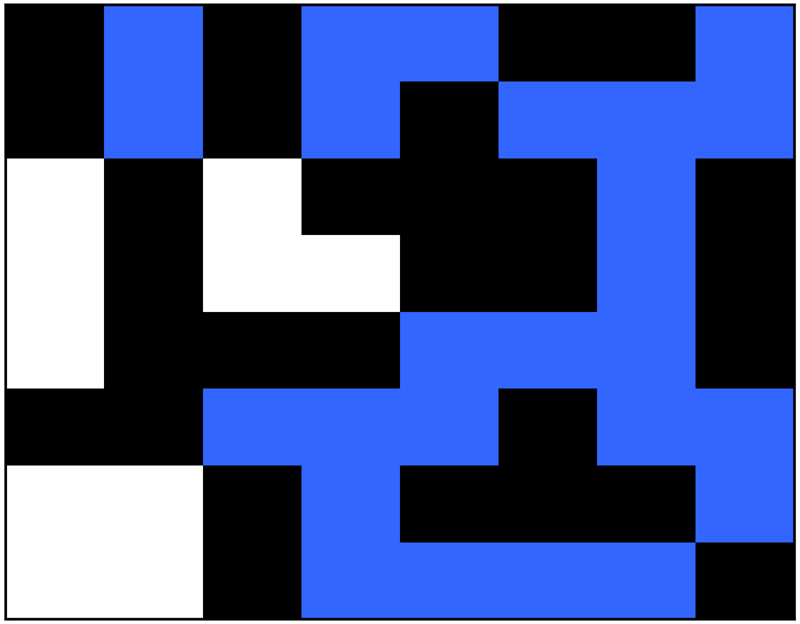
- p>p* : percolates
- p<p* : does not percolate



Percolation

Simulation:

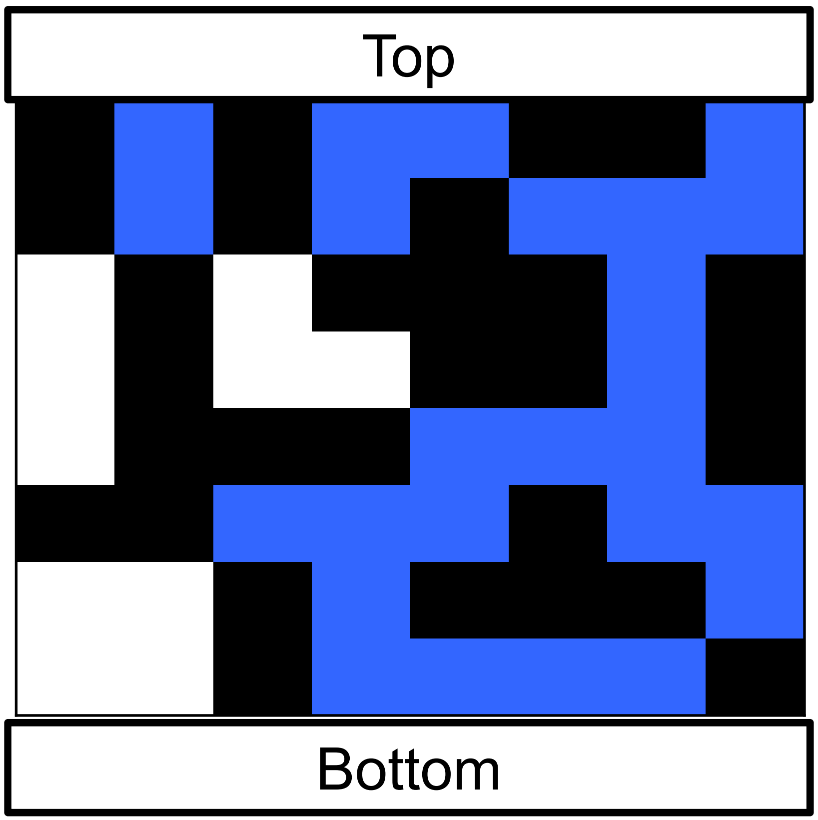
- Add each site to union-find object.
- Connect all open sites to neighboring open sites.
- For every pair on the top/bottom row, check if connected.



Percolation

Slightly better:

- Create virtual top and bottom.
- Only check if top and bottom are connected.



Intermission (a break from graphs)

Part I: Implementing a Priority Queue

- Binary Heaps
- HeapSort

Part II: Disjoint Set

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