

# Tournament Heaps

# Where We're Going

- ***Tournament Heaps (Today)***
  - A simple, flexible, and versatile priority queue.
- ***Lazy Tournament Heaps (Today)***
  - A powerful building block for designing more advanced data structures.
- ***Abdication Heaps (Tuesday)***
  - A heavyweight and theoretically excellent priority queue.

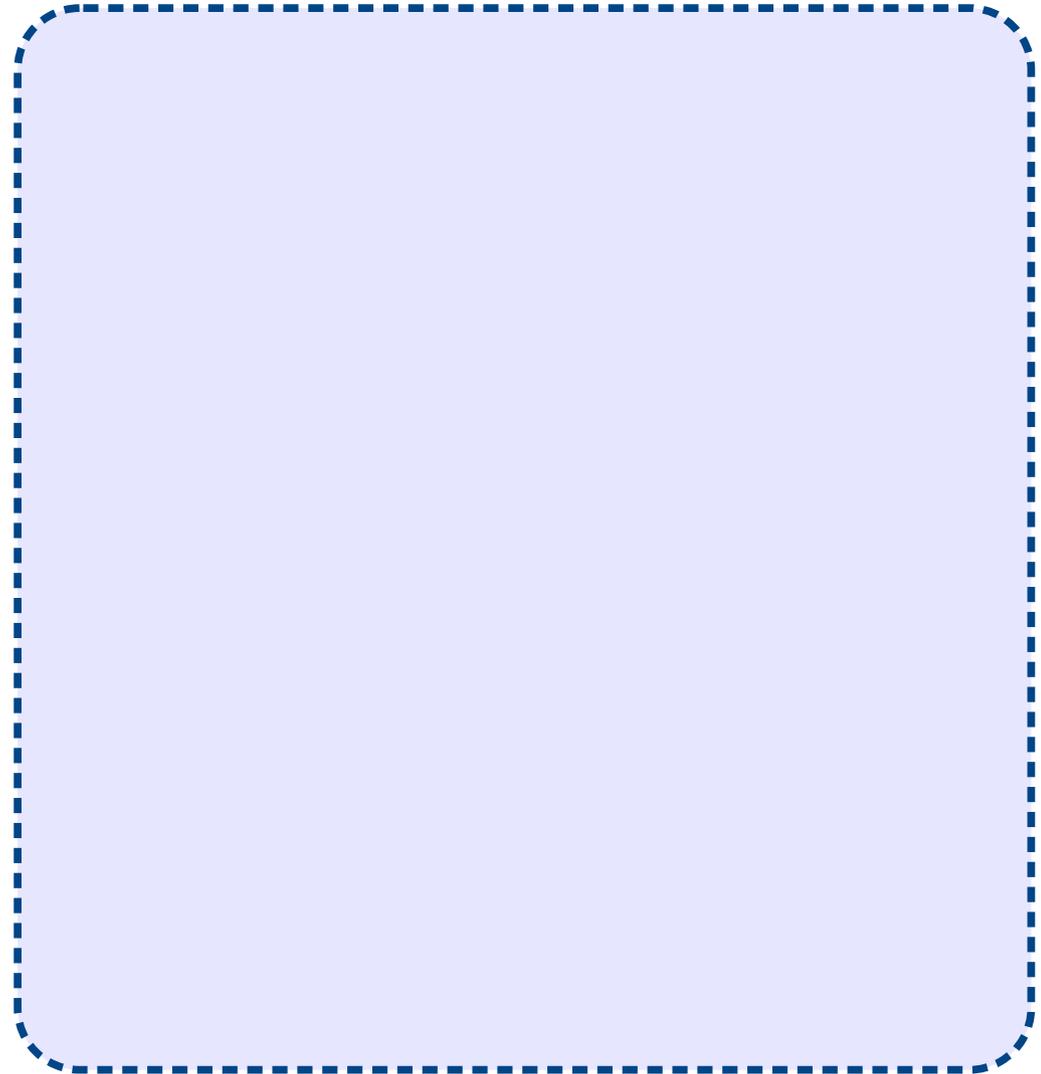
# ***Review:*** Priority Queues

# Priority Queues

- A **priority queue** is a data structure that supports these operations:
  - $pq.enqueue(v, k)$ , which enqueues element  $v$  with key  $k$ ;
  - $pq.find-min()$ , which returns the element with the least key; and
  - $pq.extract-min()$ , which removes and returns the element with the least key.
- They're useful as building blocks in a *bunch* of algorithms.

# Priority Queues

- A **priority queue** is a data structure that supports these operations:
  - $pq.enqueue(v, k)$ , which enqueues element  $v$  with key  $k$ ;
  - $pq.find-min()$ , which returns the element with the least key; and
  - $pq.extract-min()$ , which removes and returns the element with the least key.
- They're useful as building blocks in a *bunch* of algorithms.



# Priority Queues

- A **priority queue** is a data structure that supports these operations:
  - $pq.enqueue(v, k)$ , which enqueues element  $v$  with key  $k$ ;
  - $pq.find-min()$ , which returns the element with the least key; and
  - $pq.extract-min()$ , which removes and returns the element with the least key.
- They're useful as building blocks in a *bunch* of algorithms.

*Mt. Giluwe*

4,368

# Priority Queues

- A **priority queue** is a data structure that supports these operations:
  - $pq.enqueue(v, k)$ , which enqueues element  $v$  with key  $k$ ;
  - $pq.find-min()$ , which returns the element with the least key; and
  - $pq.extract-min()$ , which removes and returns the element with the least key.
- They're useful as building blocks in a *bunch* of algorithms.

*Mt. Giluwe*

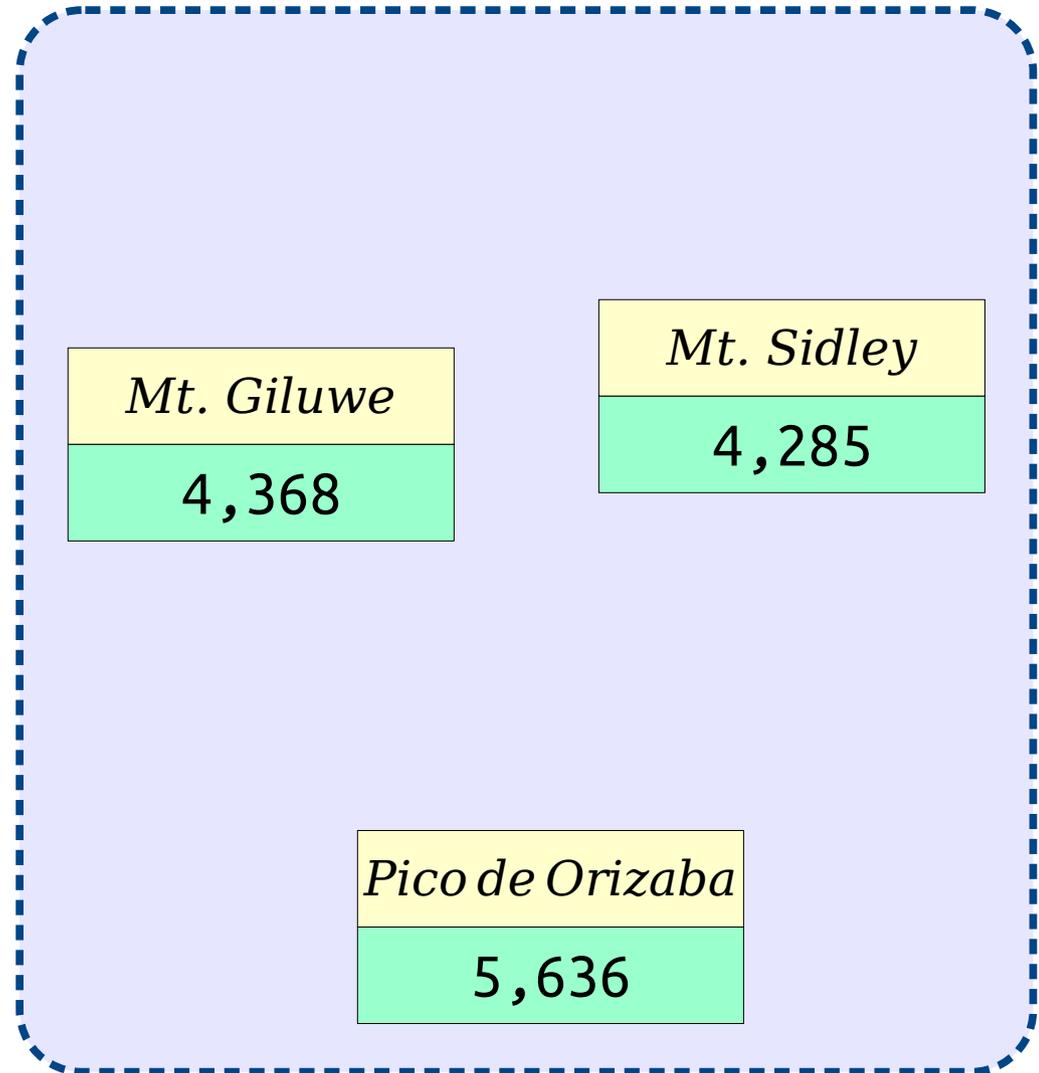
4,368

*Pico de Orizaba*

5,636

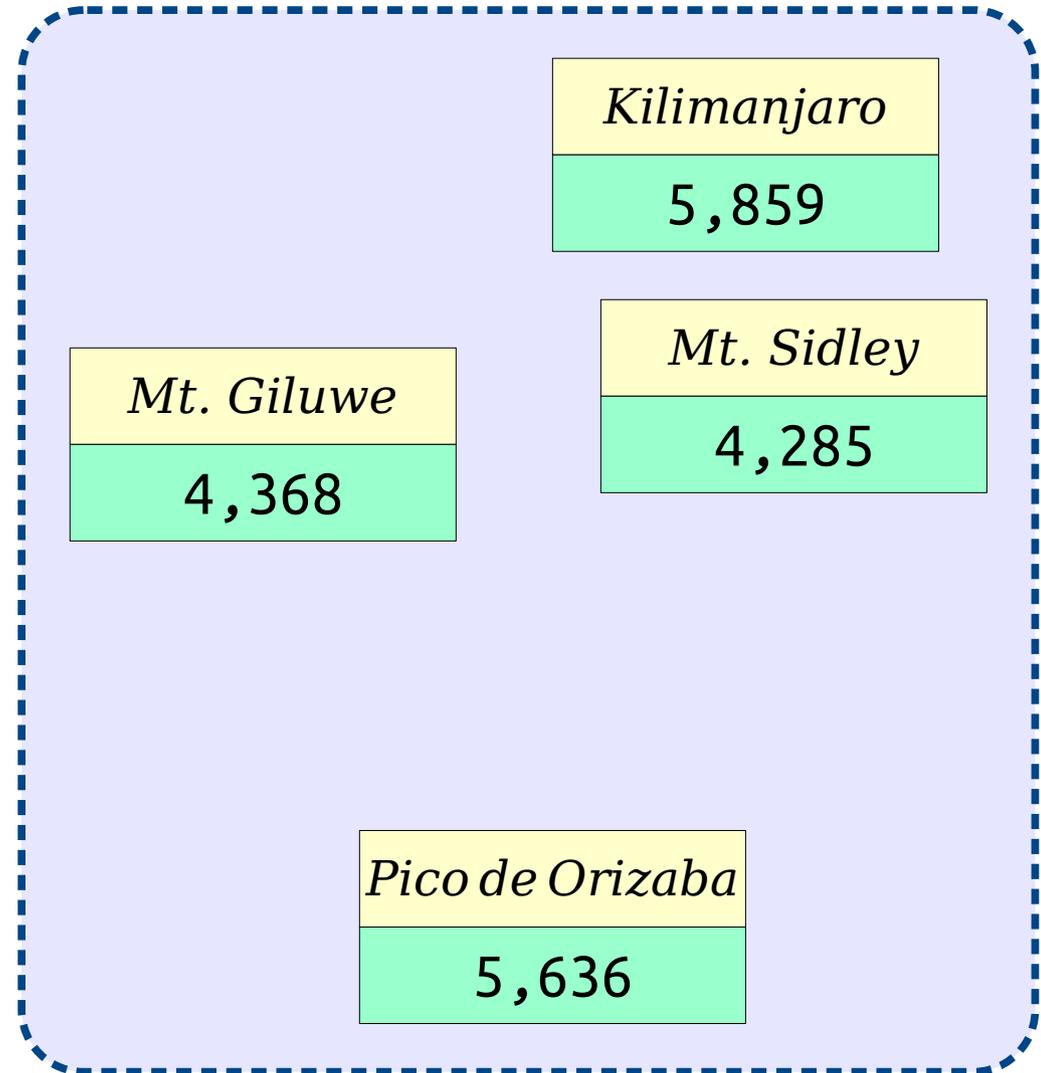
# Priority Queues

- A **priority queue** is a data structure that supports these operations:
  - $pq.enqueue(v, k)$ , which enqueues element  $v$  with key  $k$ ;
  - $pq.find-min()$ , which returns the element with the least key; and
  - $pq.extract-min()$ , which removes and returns the element with the least key.
- They're useful as building blocks in a *bunch* of algorithms.



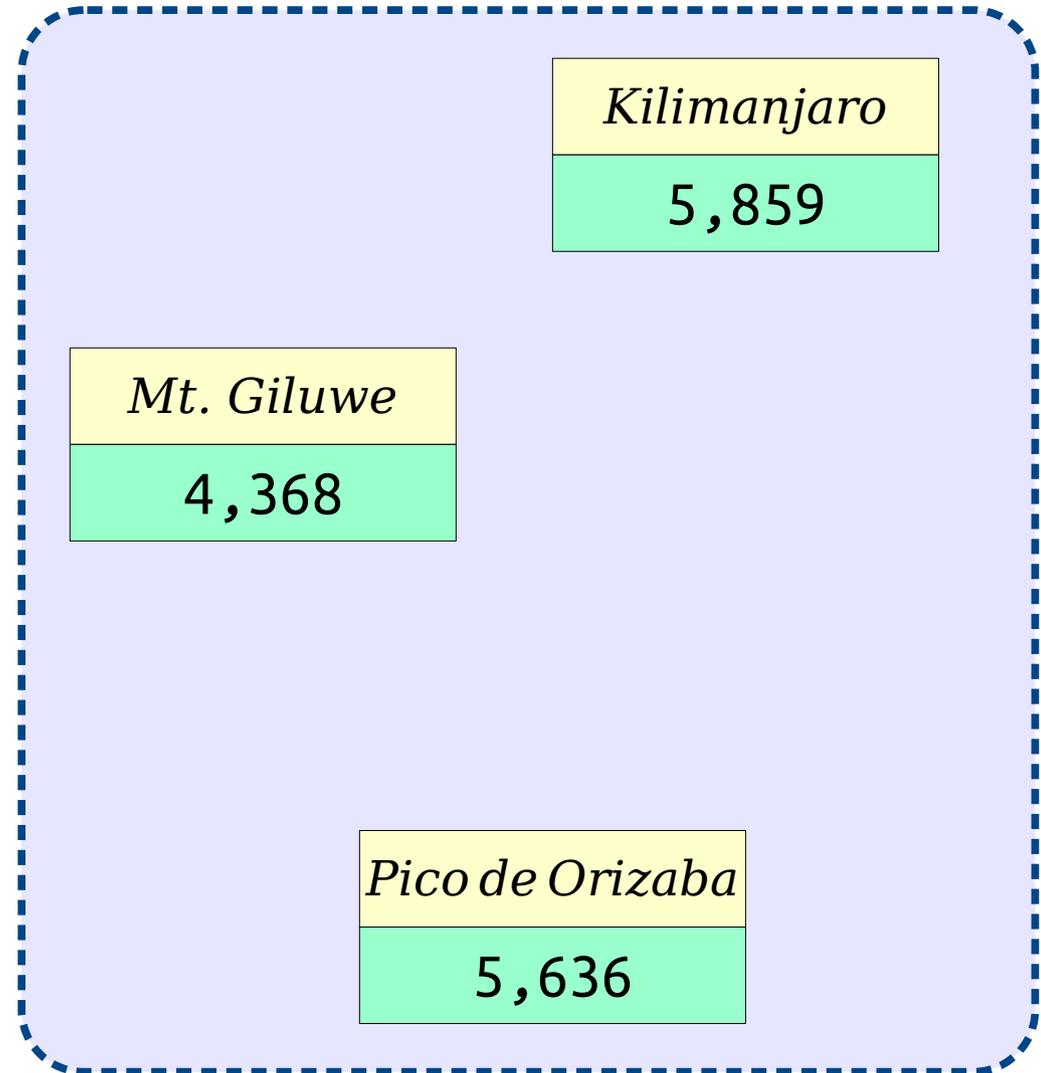
# Priority Queues

- A **priority queue** is a data structure that supports these operations:
  - $pq.enqueue(v, k)$ , which enqueues element  $v$  with key  $k$ ;
  - $pq.find-min()$ , which returns the element with the least key; and
  - $pq.extract-min()$ , which removes and returns the element with the least key.
- They're useful as building blocks in a *bunch* of algorithms.



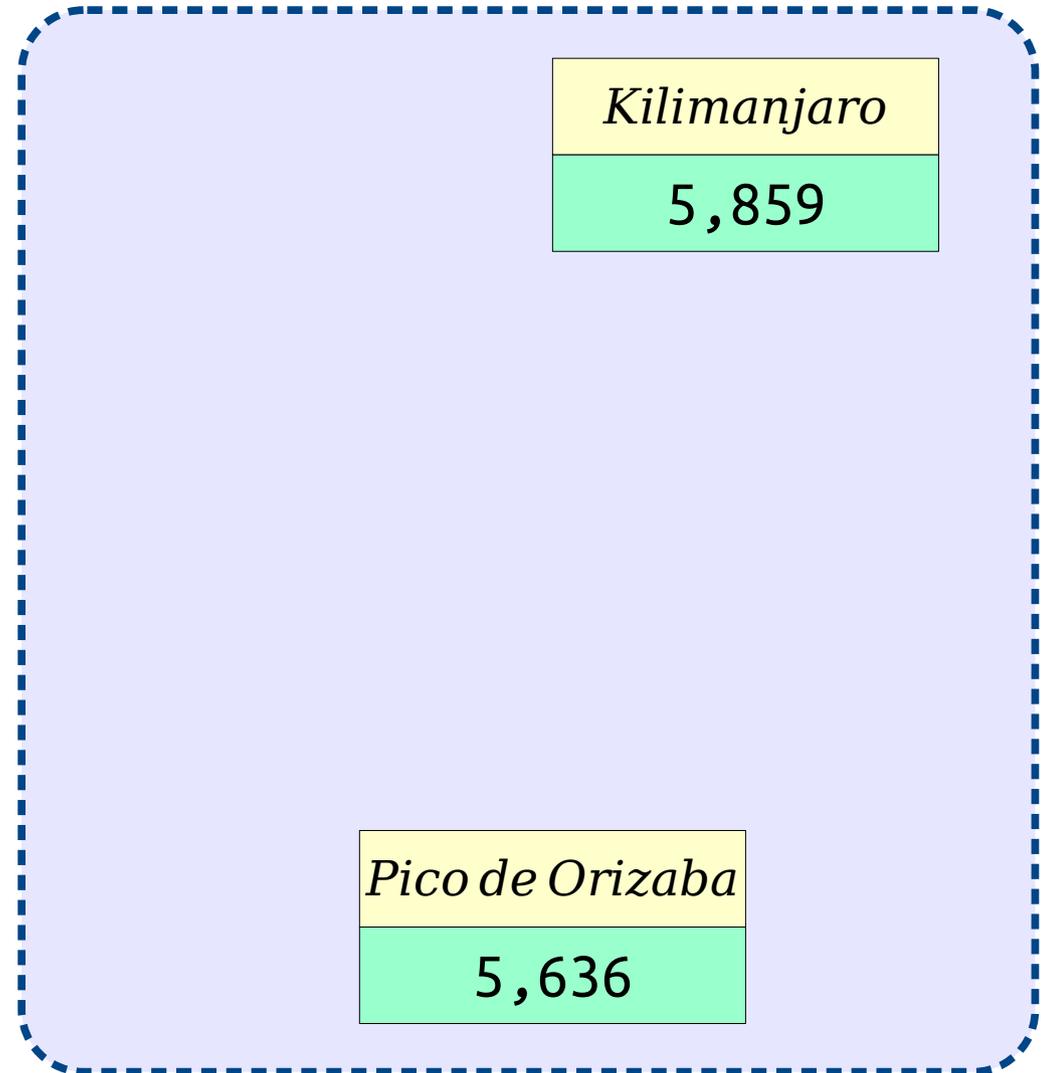
# Priority Queues

- A **priority queue** is a data structure that supports these operations:
  - $pq.enqueue(v, k)$ , which enqueues element  $v$  with key  $k$ ;
  - $pq.find-min()$ , which returns the element with the least key; and
  - $pq.extract-min()$ , which removes and returns the element with the least key.
- They're useful as building blocks in a *bunch* of algorithms.



# Priority Queues

- A **priority queue** is a data structure that supports these operations:
  - $pq.enqueue(v, k)$ , which enqueues element  $v$  with key  $k$ ;
  - $pq.find-min()$ , which returns the element with the least key; and
  - $pq.extract-min()$ , which removes and returns the element with the least key.
- They're useful as building blocks in a *bunch* of algorithms.



# Binary Heaps

- Priority queues are frequently implemented as **binary heaps**.
  - **enqueue** and **extract-min** run in time  $O(\log n)$ ; **find-min** runs in time  $O(1)$ .
- These heaps are surprisingly fast in practice. It's tough to beat their performance!
  - $d$ -ary heaps can outperform binary heaps for a well-tuned value of  $d$ , and otherwise only the **sequence heap** is known to specifically outperform this family.
  - (Is this information incorrect as of 2022? Let me know and I'll update it.)
- In that case, why do we need other heaps?

# Priority Queues in Practice

- Many graph algorithms directly rely on priority queues supporting extra operations:
  - ***meld***( $pq_1, pq_2$ ): Destroy  $pq_1$  and  $pq_2$  and combine their elements into a single priority queue. (*MSTs via Cheriton-Tarjan*)
  - $pq$ .***decrease-key***( $v, k'$ ): Given a pointer to element  $v$  already in the queue, lower its key to have new value  $k'$ . (*Shortest paths via Dijkstra, global min-cut via Stoer-Wagner*)
  - $pq$ .***add-to-all***( $\Delta k$ ): Add  $\Delta k$  to the keys of each element in the priority queue, typically used with ***meld***. (*Optimum branchings via Chu-Edmonds-Liu*)
- In lecture, we'll cover tournament heaps to efficiently support ***meld*** and abdication heaps to efficiently support ***meld*** and ***decrease-key***.
- You'll design a priority queue supporting ***meld*** and ***add-to-all*** on the next problem set.

# Priority Queues in Practice

Many graph algorithms directly rely on priority queues supporting extra operations:

- ***meld***( $pq_1, pq_2$ ): Destroy  $pq_1$  and  $pq_2$  and combine their elements into a single priority queue. (*MSTs via Cheriton-Tarjan*)

***pq.decrease-key***( $v, k'$ ): Given a pointer to element  $v$  already in the queue, lower its key to have new value  $k'$ . (*Shortest paths via Dijkstra, global min-cut via Stoer-Wagner*)

***pq.add-to-all***( $\Delta k$ ): Add  $\Delta k$  to the keys of each element in the priority queue, typically used with ***meld***. (*Optimum branchings via Chu-Edmonds-Liu*)

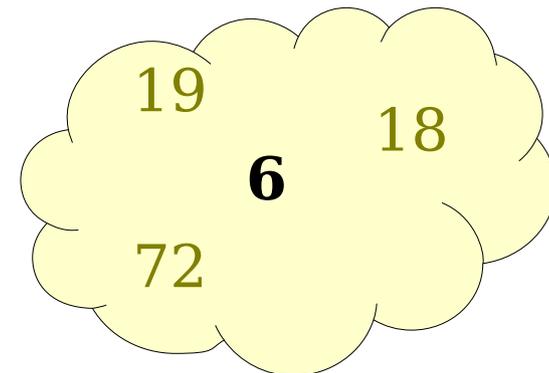
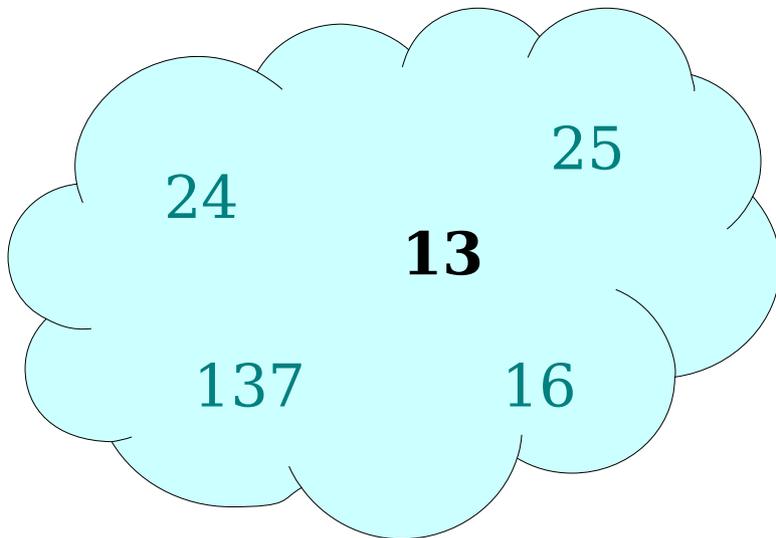
In lecture, we'll cover tournament heaps to efficiently support ***meld*** and abdication heaps to efficiently support ***meld*** and ***decrease-key***.

You'll design a priority queue supporting ***meld*** and ***add-to-all*** on the next problem set.

# Meldable Priority Queues

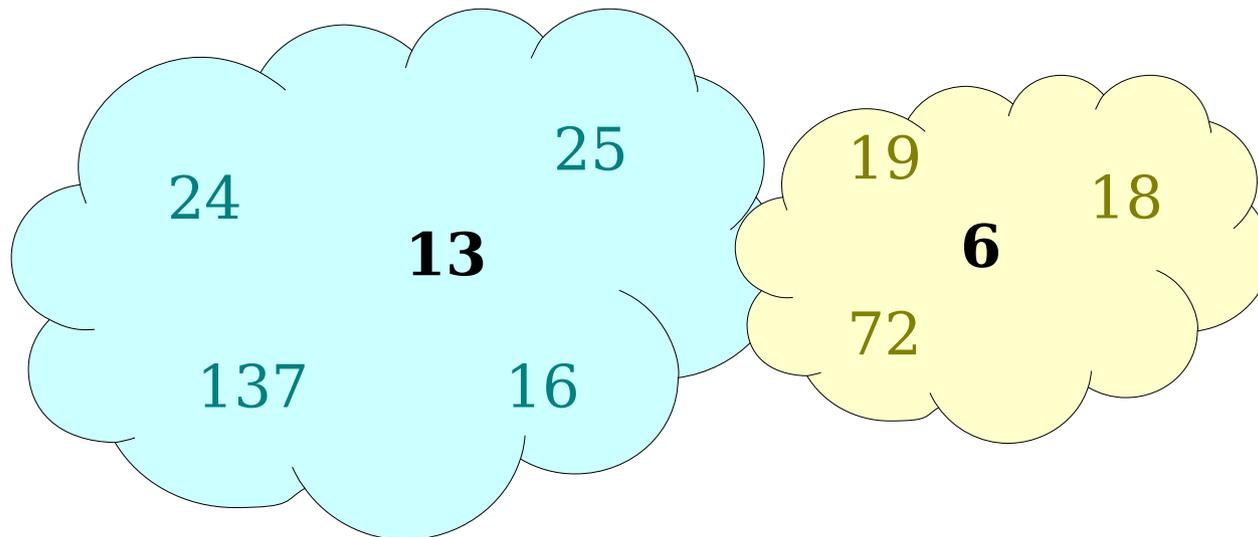
# Meldable Priority Queues

- A priority queue supporting the ***meld*** operation is called a ***meldable priority queue***.
- ***meld***( $pq_1$ ,  $pq_2$ ) destructively modifies  $pq_1$  and  $pq_2$  and produces a new priority queue containing all elements of  $pq_1$  and  $pq_2$ .



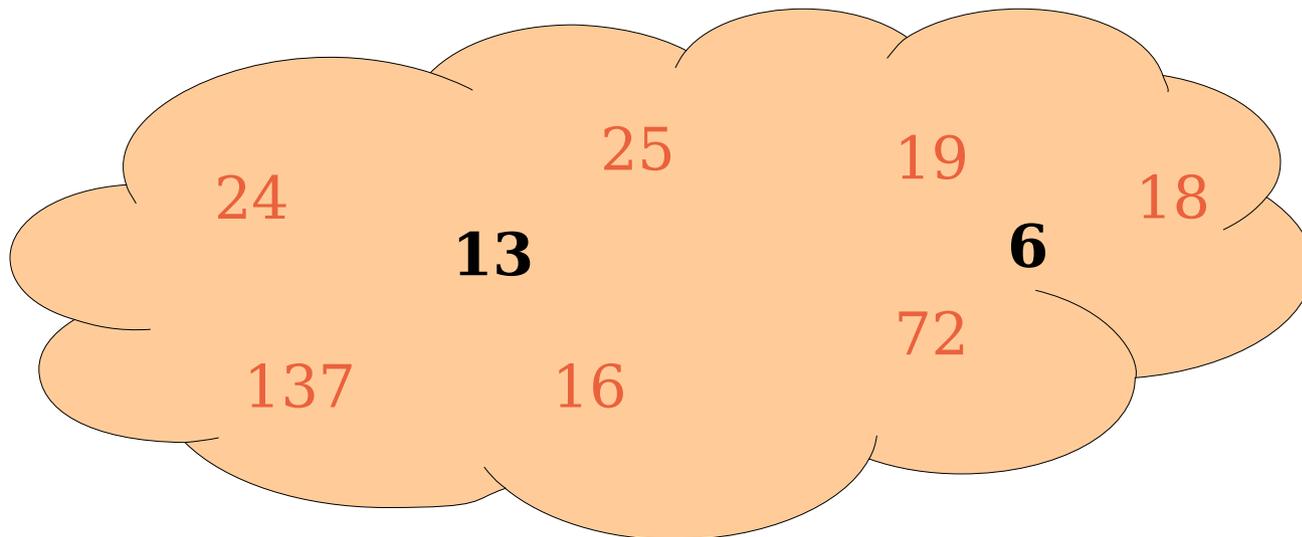
# Meldable Priority Queues

- A priority queue supporting the *meld* operation is called a *meldable priority queue*.
- *meld*( $pq_1, pq_2$ ) destructively modifies  $pq_1$  and  $pq_2$  and produces a new priority queue containing all elements of  $pq_1$  and  $pq_2$ .



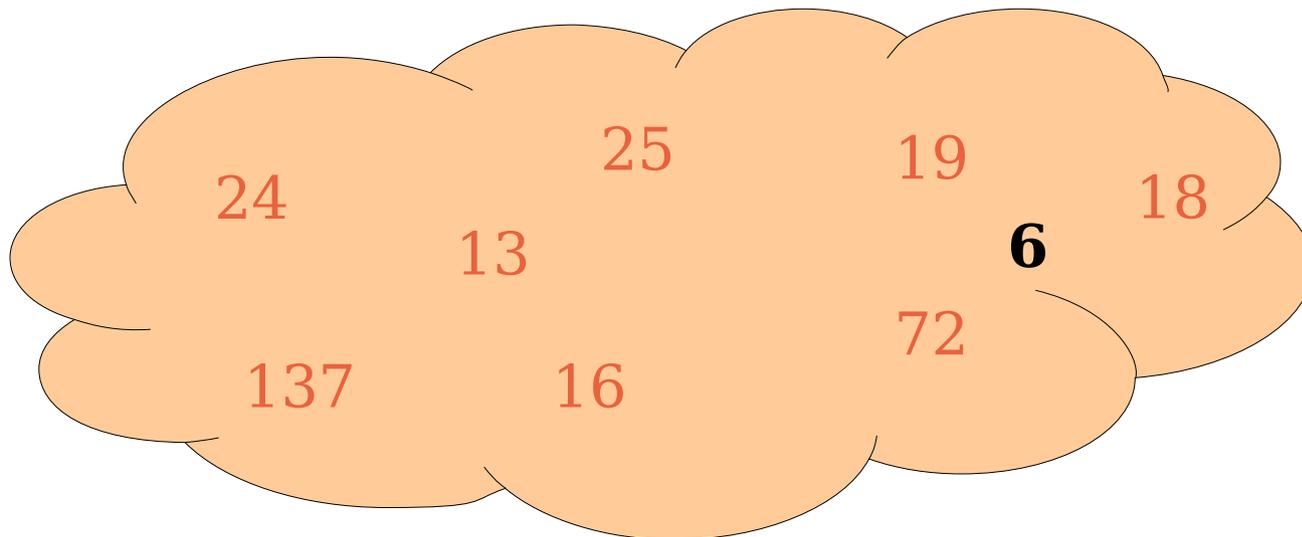
# Meldable Priority Queues

- A priority queue supporting the *meld* operation is called a *meldable priority queue*.
- *meld*( $pq_1, pq_2$ ) destructively modifies  $pq_1$  and  $pq_2$  and produces a new priority queue containing all elements of  $pq_1$  and  $pq_2$ .



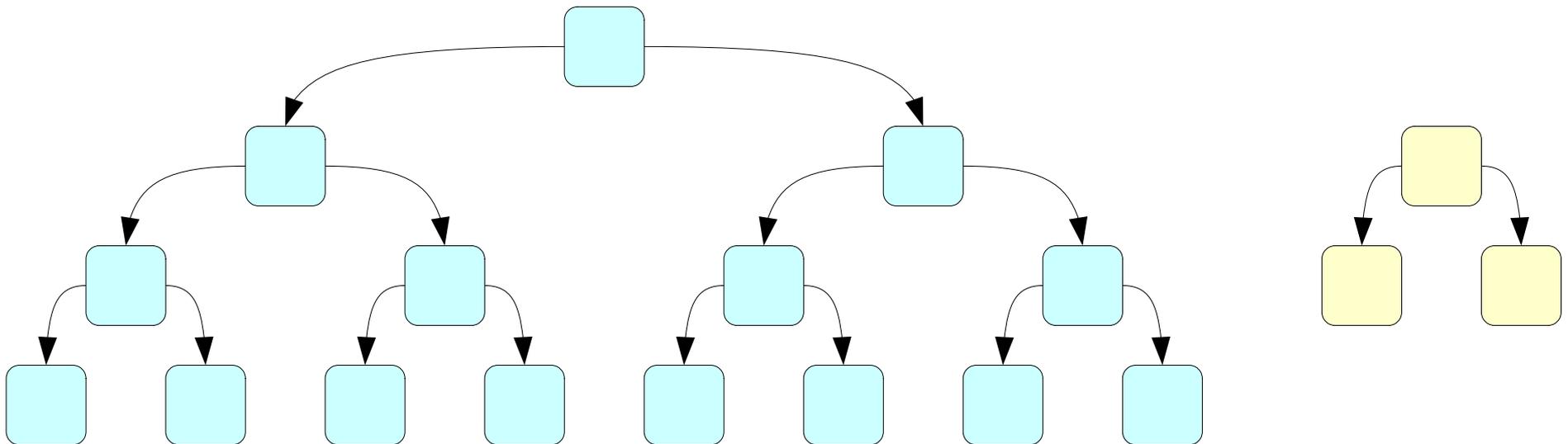
# Meldable Priority Queues

- A priority queue supporting the *meld* operation is called a *meldable priority queue*.
- *meld*( $pq_1, pq_2$ ) destructively modifies  $pq_1$  and  $pq_2$  and produces a new priority queue containing all elements of  $pq_1$  and  $pq_2$ .



# Efficiently Meldable Queues

- Standard binary heaps do not efficiently support *meld*.
- **Intuition**: Binary heaps are complete binary trees, and two complete binary trees cannot easily be linked to one another.



What things *can* be combined together  
efficiently?

# Adding Binary Numbers

- Given the binary representations of two numbers  $n$  and  $m$ , we can add those numbers in time  $O(\log m + \log n)$ .

***Intuition:***

Writing out  $n$  in any “reasonable” base requires  $\Theta(\log n)$  digits.

# Adding Binary Numbers

- Given the binary representations of two numbers  $n$  and  $m$ , we can add those numbers in time  $O(\log m + \log n)$ .

	1	0	1	1	0
+		1	1	1	1
<hr/>					

# Adding Binary Numbers

- Given the binary representations of two numbers  $n$  and  $m$ , we can add those numbers in time  $O(\log m + \log n)$ .

	1	0	1	1	0
+		1	1	1	1
<hr/>					

# Adding Binary Numbers

- Given the binary representations of two numbers  $n$  and  $m$ , we can add those numbers in time  $O(\log m + \log n)$ .

	1	0	1	1	0
+		1	1	1	1
<hr/>					
					1

# Adding Binary Numbers

- Given the binary representations of two numbers  $n$  and  $m$ , we can add those numbers in time  $O(\log m + \log n)$ .

	1	0	1	1	0
+		1	1	1	1
<hr/>					
					1

# Adding Binary Numbers

- Given the binary representations of two numbers  $n$  and  $m$ , we can add those numbers in time  $O(\log m + \log n)$ .

A diagram illustrating the addition of two binary numbers. The numbers are aligned by their least significant bits. The first number is 10110 and the second is 01111. A plus sign is to the left of the second number. A horizontal line is drawn under the numbers. The result 01101 is shown below the line. A carry of 1 is shown above the third column from the right. The fourth column from the right (the column with 1+1) is highlighted in yellow.

			1		
	1	0	1	1	0
+		1	1	1	1
<hr/>					
			0	1	

# Adding Binary Numbers

- Given the binary representations of two numbers  $n$  and  $m$ , we can add those numbers in time  $O(\log m + \log n)$ .

A diagram illustrating the addition of two binary numbers. The numbers are aligned vertically, with a plus sign to the left. The top number is 10110 and the bottom number is 01111. A horizontal line is drawn under the bottom number. The result, 10101, is shown below the line. A carry of 1 is shown above the third column from the right. The third column from the right (the third column from the left) is highlighted in yellow.

		1			
	1	0	1	1	0
+		1	1	1	1
<hr/>					
				0	1

# Adding Binary Numbers

- Given the binary representations of two numbers  $n$  and  $m$ , we can add those numbers in time  $O(\log m + \log n)$ .

		1	1		
	1	0	1	1	0
+		1	1	1	1
<hr/>					
			1	0	1

# Adding Binary Numbers

- Given the binary representations of two numbers  $n$  and  $m$ , we can add those numbers in time  $O(\log m + \log n)$ .

		1	1		
	1	0	1	1	0
+		1	1	1	1
			1	0	1

# Adding Binary Numbers

- Given the binary representations of two numbers  $n$  and  $m$ , we can add those numbers in time  $O(\log m + \log n)$ .

		1	1	1		
	1	0	1	1	0	
+		1	1	1	1	
<hr/>						
		0	1	0	1	

# Adding Binary Numbers

- Given the binary representations of two numbers  $n$  and  $m$ , we can add those numbers in time  $O(\log m + \log n)$ .

		1	1	1		
		1	0	1	1	0
+		1	1	1	1	
<hr/>						
		0	1	0	1	

# Adding Binary Numbers

- Given the binary representations of two numbers  $n$  and  $m$ , we can add those numbers in time  $O(\log m + \log n)$ .

	1	1	1	1		
	1	0	1	1	0	
+	1	1	1	1		
<hr/>						
	0	0	1	0	1	



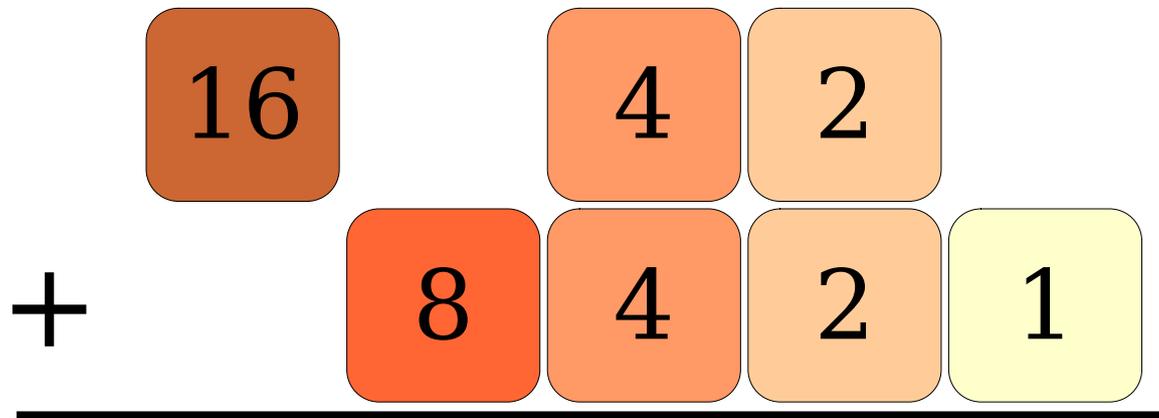
# A Different Intuition

- Represent  $n$  and  $m$  as a collection of “packets” whose sizes are powers of two.
- Adding together  $n$  and  $m$  can then be thought of as combining the packets together, eliminating duplicates

$$\begin{array}{rcccccc} & 1 & 0 & 1 & 1 & 0 \\ + & & 1 & 1 & 1 & 1 \\ \hline \end{array}$$

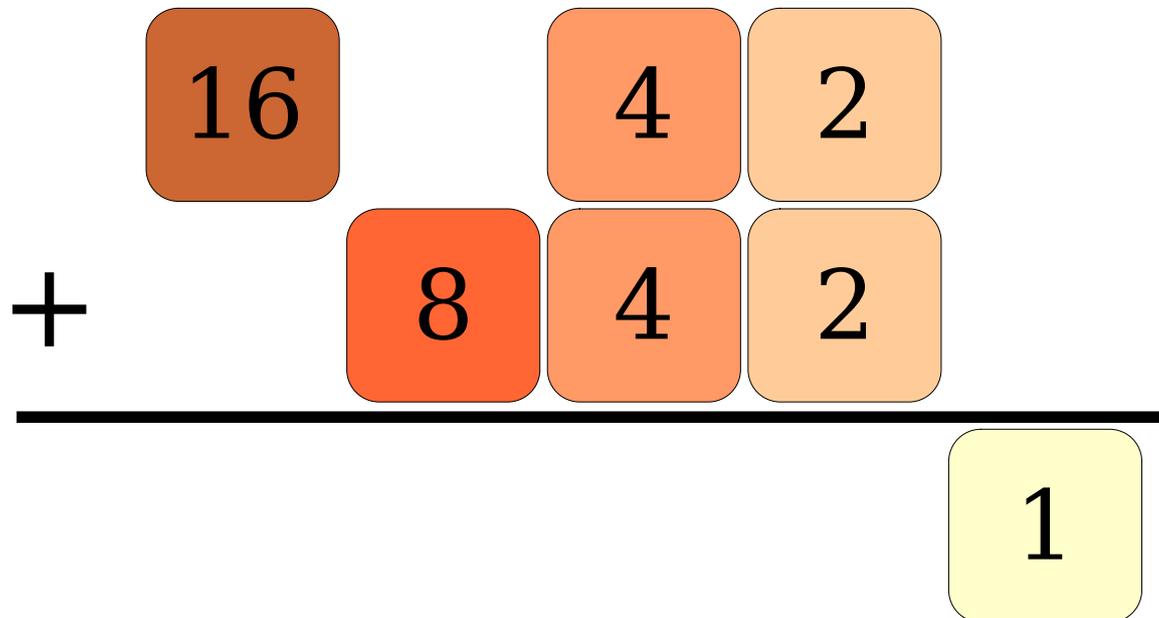
# A Different Intuition

- Represent  $n$  and  $m$  as a collection of “packets” whose sizes are powers of two.
- Adding together  $n$  and  $m$  can then be thought of as combining the packets together, eliminating duplicates



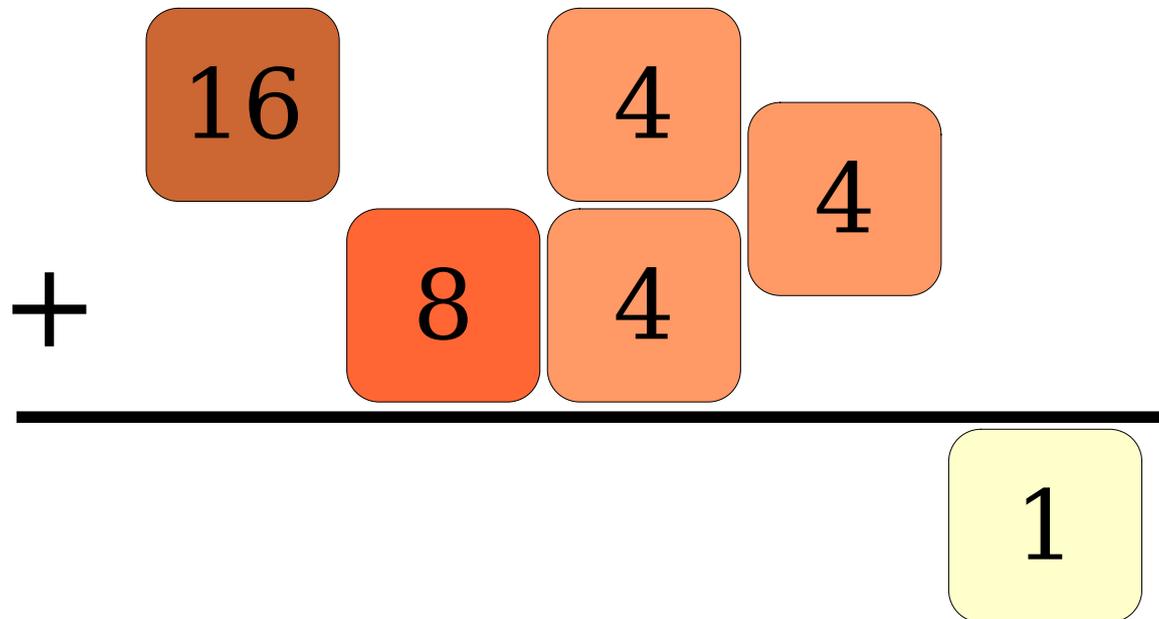
# A Different Intuition

- Represent  $n$  and  $m$  as a collection of “packets” whose sizes are powers of two.
- Adding together  $n$  and  $m$  can then be thought of as combining the packets together, eliminating duplicates



# A Different Intuition

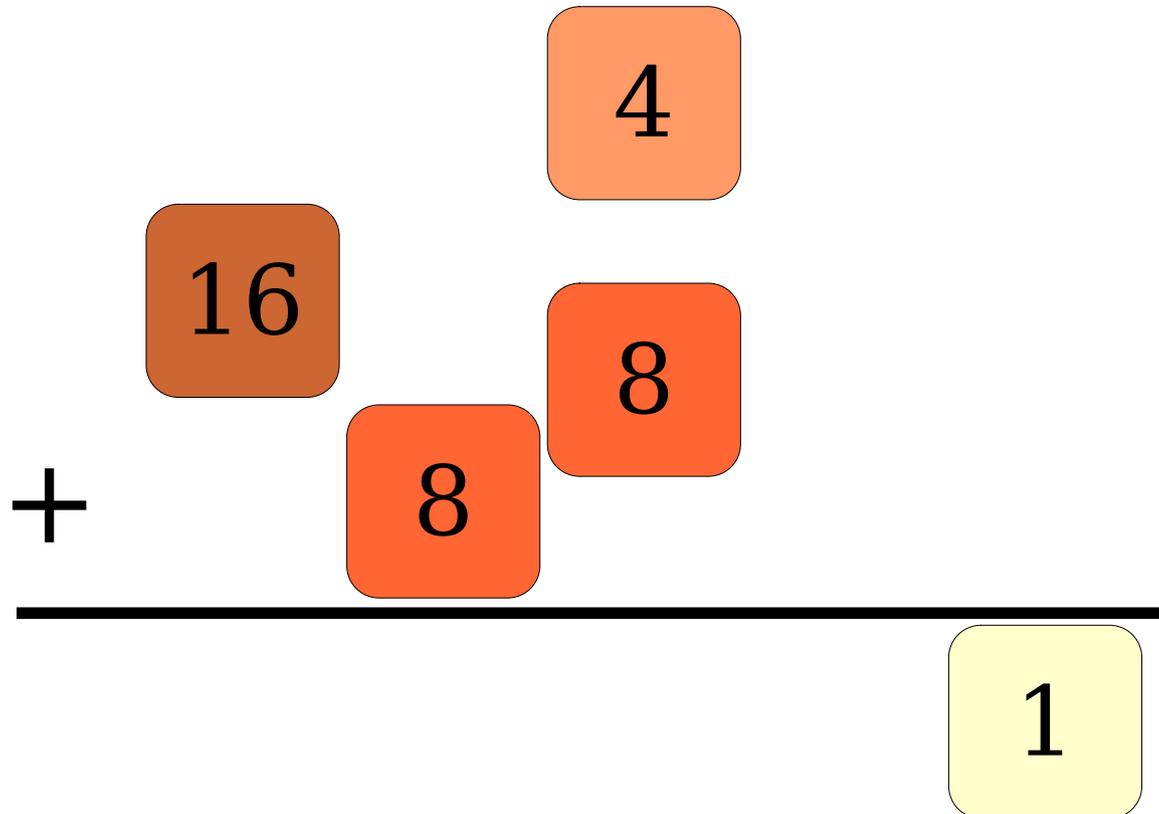
- Represent  $n$  and  $m$  as a collection of “packets” whose sizes are powers of two.
- Adding together  $n$  and  $m$  can then be thought of as combining the packets together, eliminating duplicates





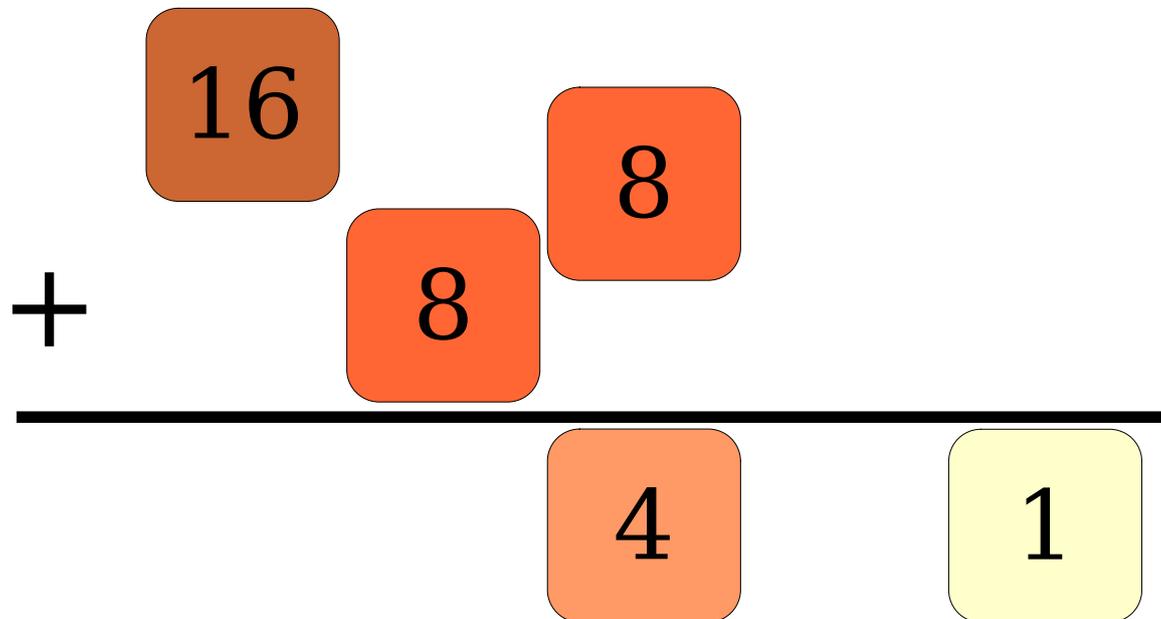
# A Different Intuition

- Represent  $n$  and  $m$  as a collection of “packets” whose sizes are powers of two.
- Adding together  $n$  and  $m$  can then be thought of as combining the packets together, eliminating duplicates



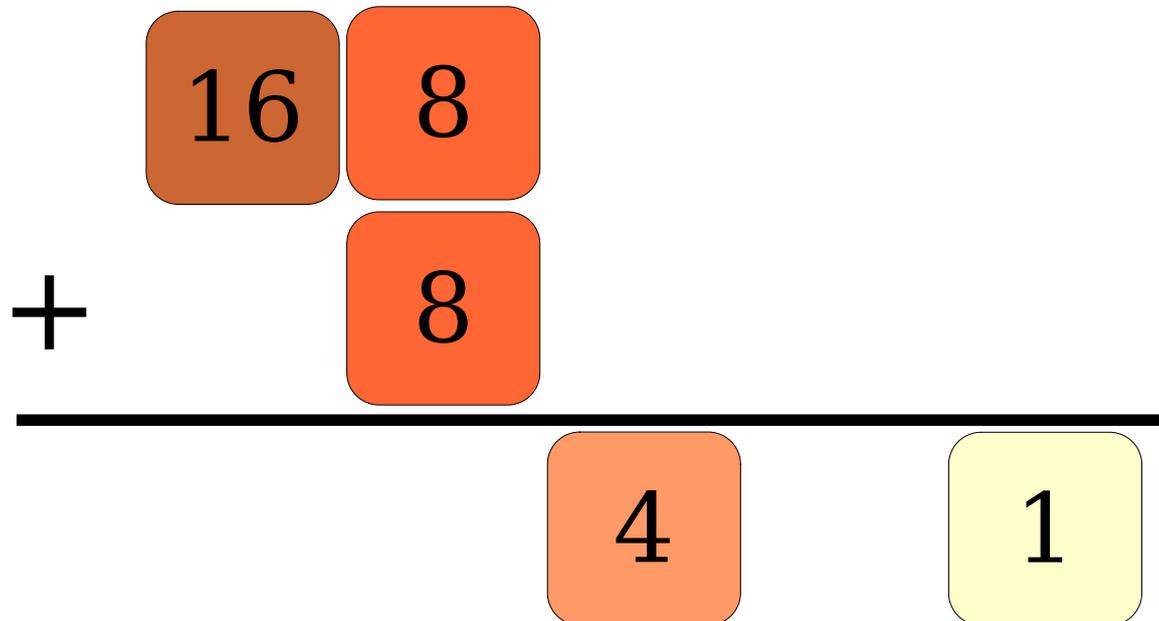
# A Different Intuition

- Represent  $n$  and  $m$  as a collection of “packets” whose sizes are powers of two.
- Adding together  $n$  and  $m$  can then be thought of as combining the packets together, eliminating duplicates



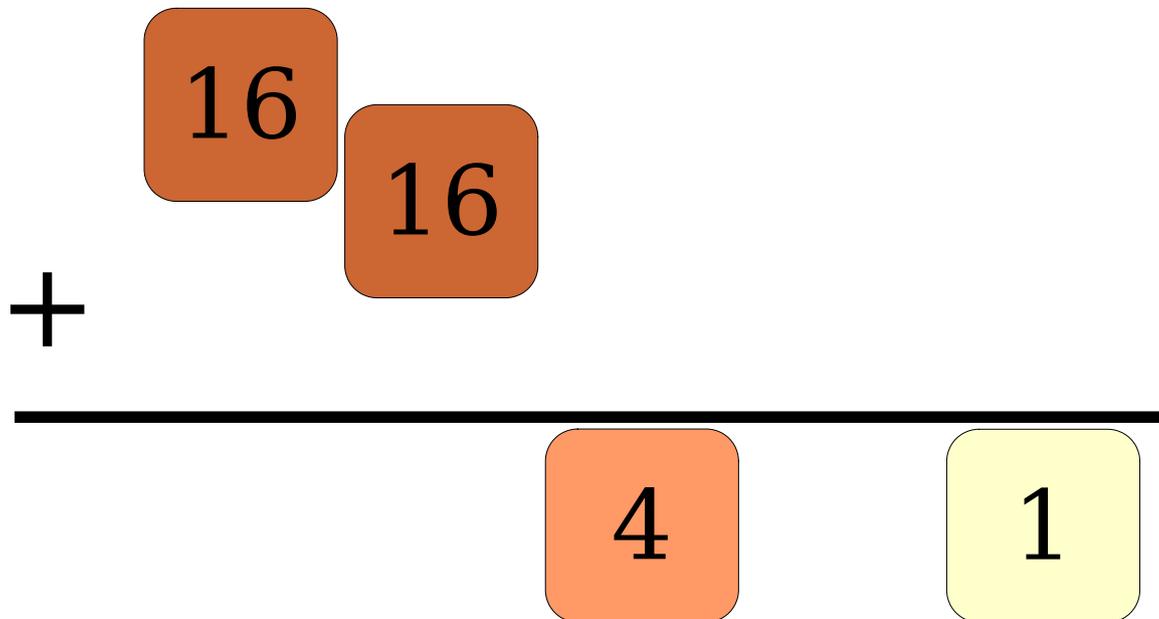
# A Different Intuition

- Represent  $n$  and  $m$  as a collection of “packets” whose sizes are powers of two.
- Adding together  $n$  and  $m$  can then be thought of as combining the packets together, eliminating duplicates



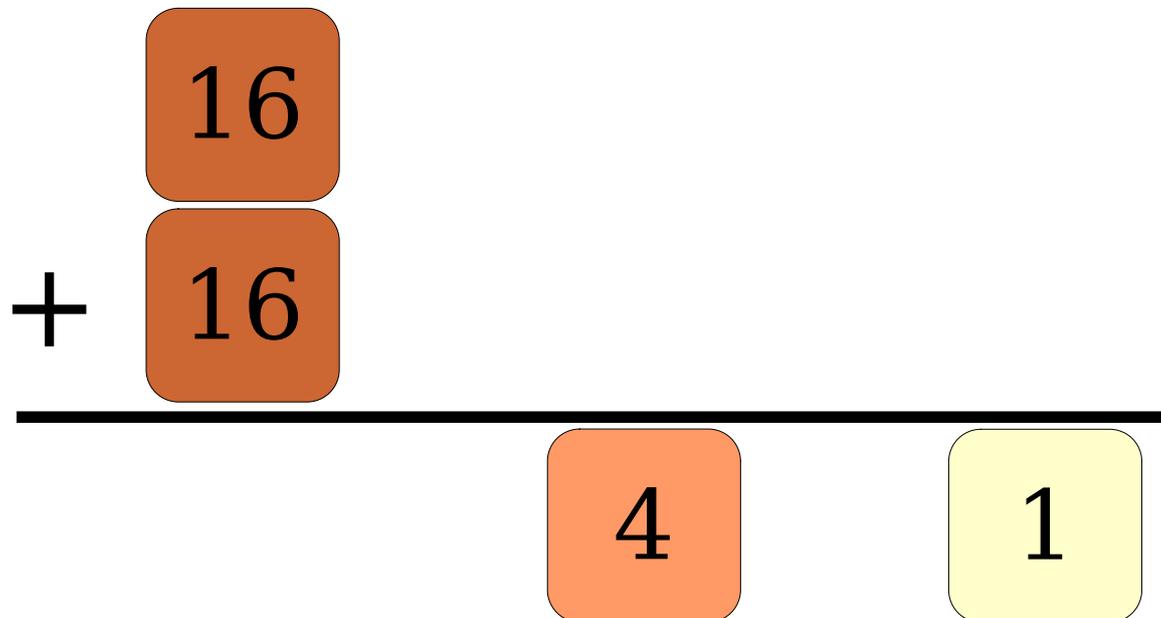
# A Different Intuition

- Represent  $n$  and  $m$  as a collection of “packets” whose sizes are powers of two.
- Adding together  $n$  and  $m$  can then be thought of as combining the packets together, eliminating duplicates



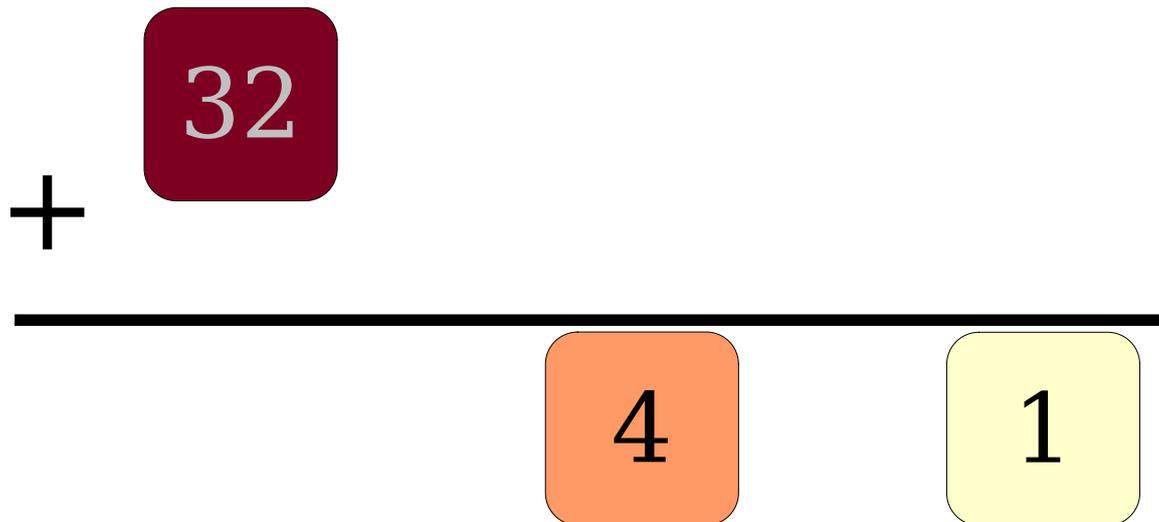
# A Different Intuition

- Represent  $n$  and  $m$  as a collection of “packets” whose sizes are powers of two.
- Adding together  $n$  and  $m$  can then be thought of as combining the packets together, eliminating duplicates



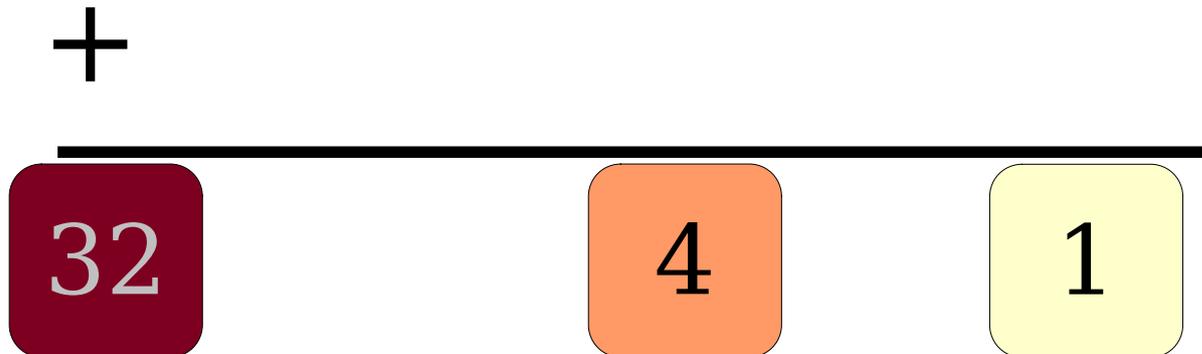
# A Different Intuition

- Represent  $n$  and  $m$  as a collection of “packets” whose sizes are powers of two.
- Adding together  $n$  and  $m$  can then be thought of as combining the packets together, eliminating duplicates



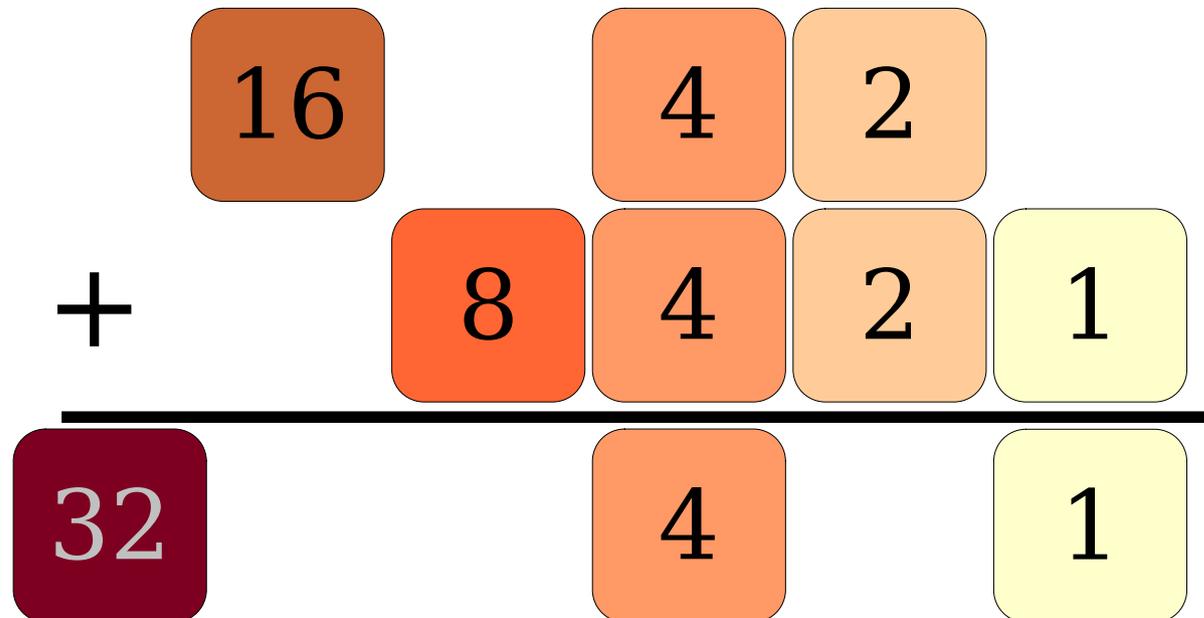
# A Different Intuition

- Represent  $n$  and  $m$  as a collection of “packets” whose sizes are powers of two.
- Adding together  $n$  and  $m$  can then be thought of as combining the packets together, eliminating duplicates



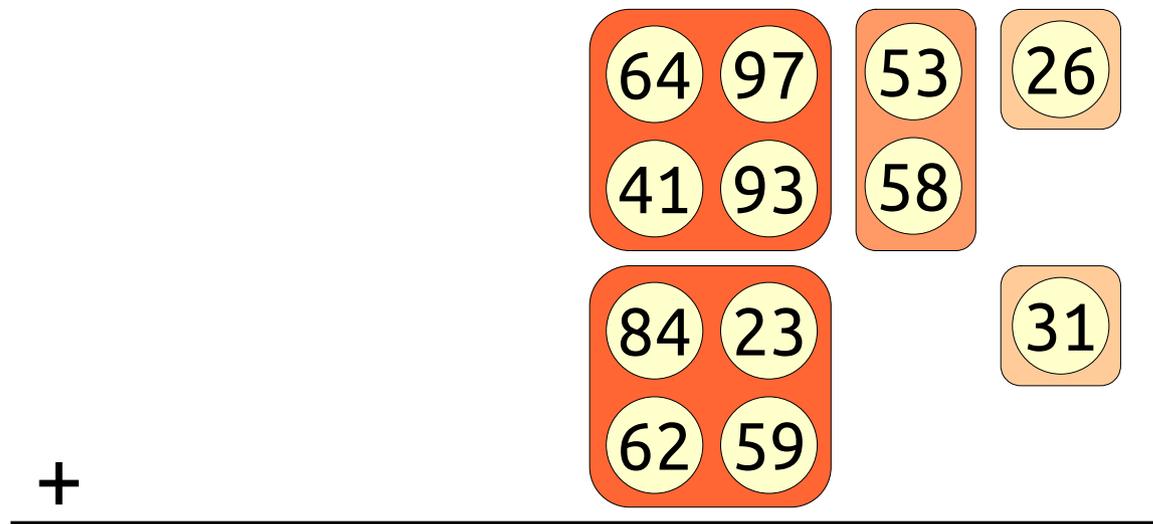
# A Different Intuition

- Represent  $n$  and  $m$  as a collection of “packets” whose sizes are powers of two.
- Adding together  $n$  and  $m$  can then be thought of as combining the packets together, eliminating duplicates



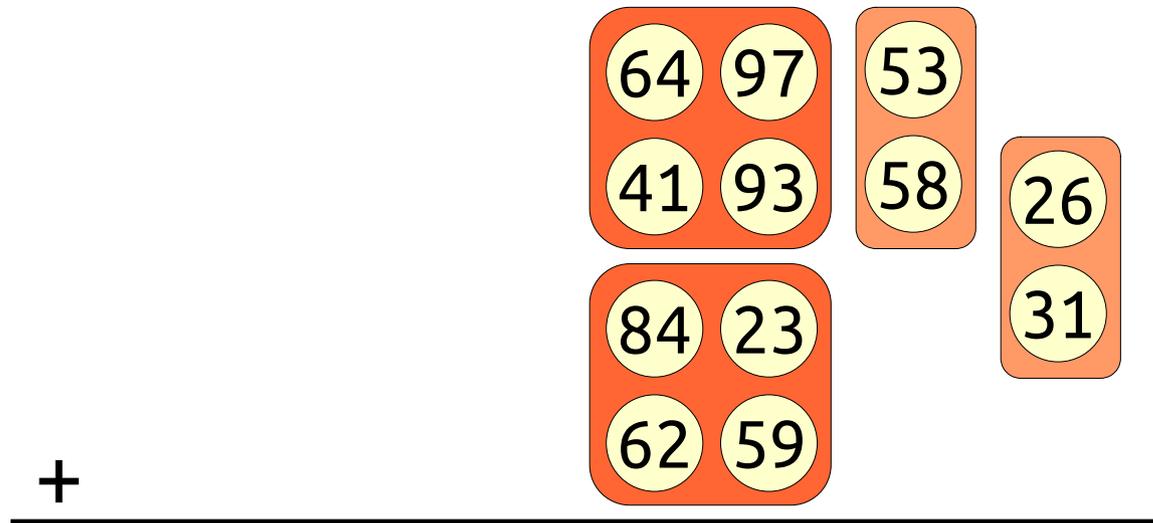
# Building a Priority Queue

- **Idea:** Store elements in “packets” whose sizes are powers of two and *meld* by “adding” groups of packets.



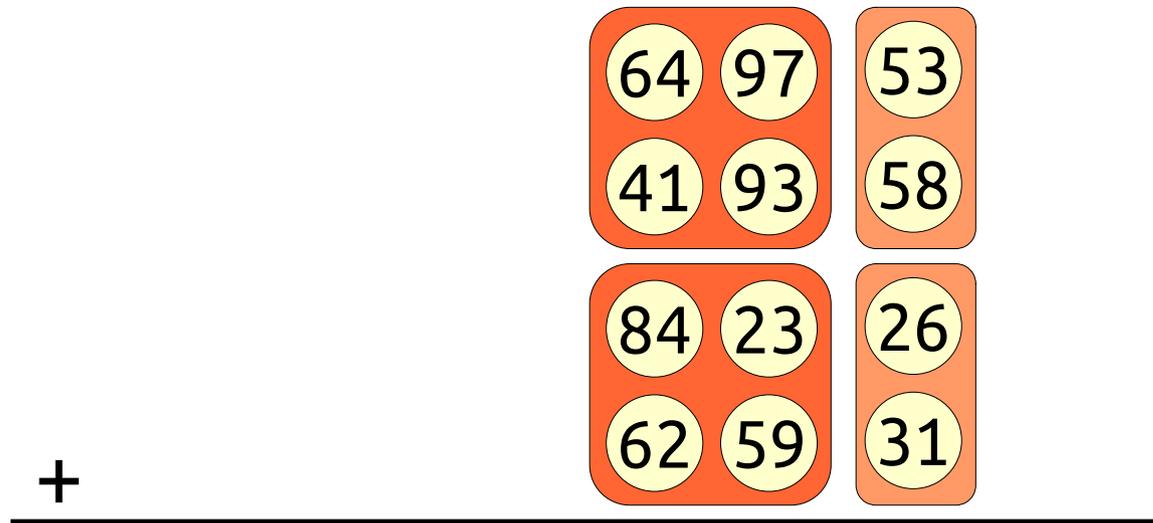
# Building a Priority Queue

- **Idea:** Store elements in “packets” whose sizes are powers of two and *meld* by “adding” groups of packets.



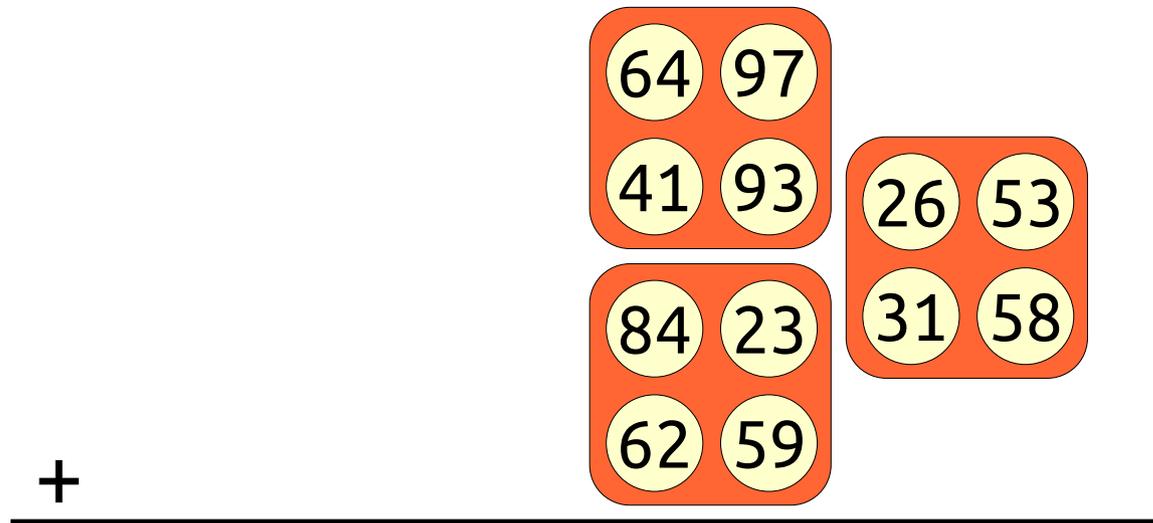
# Building a Priority Queue

- **Idea:** Store elements in “packets” whose sizes are powers of two and **meld** by “adding” groups of packets.



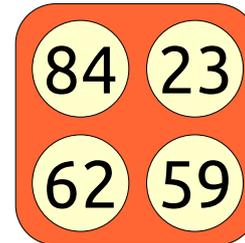
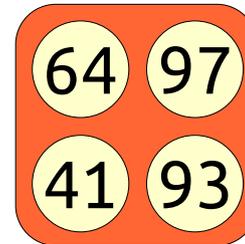
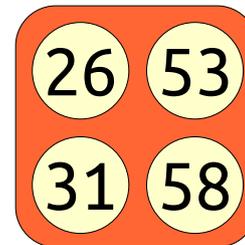
# Building a Priority Queue

- **Idea:** Store elements in “packets” whose sizes are powers of two and *meld* by “adding” groups of packets.



# Building a Priority Queue

- **Idea:** Store elements in “packets” whose sizes are powers of two and *meld* by “adding” groups of packets.

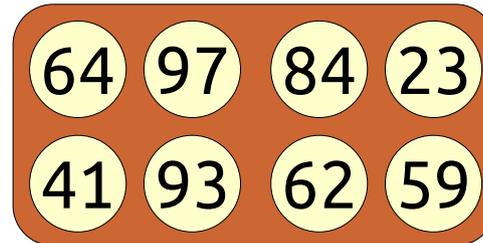
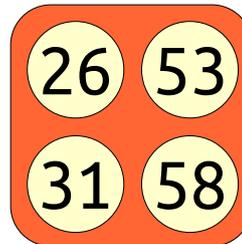


+

---

# Building a Priority Queue

- **Idea:** Store elements in “packets” whose sizes are powers of two and **meld** by “adding” groups of packets.

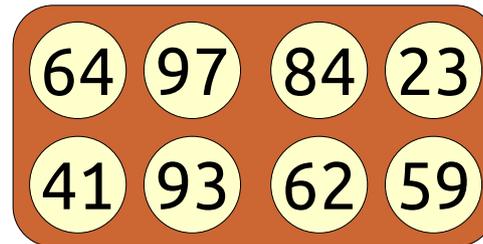


+

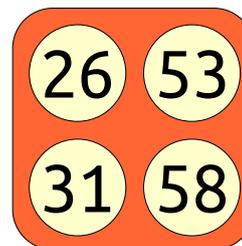
---

# Building a Priority Queue

- **Idea:** Store elements in “packets” whose sizes are powers of two and **meld** by “adding” groups of packets.

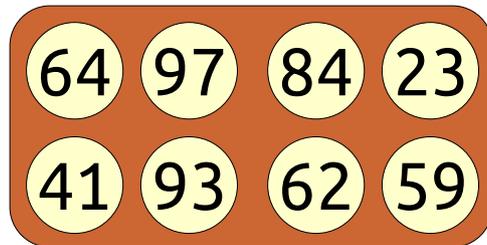


+

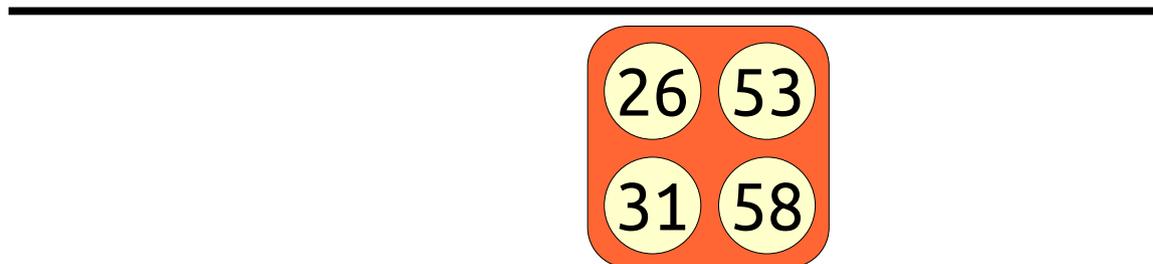


# Building a Priority Queue

- **Idea:** Store elements in “packets” whose sizes are powers of two and *meld* by “adding” groups of packets.

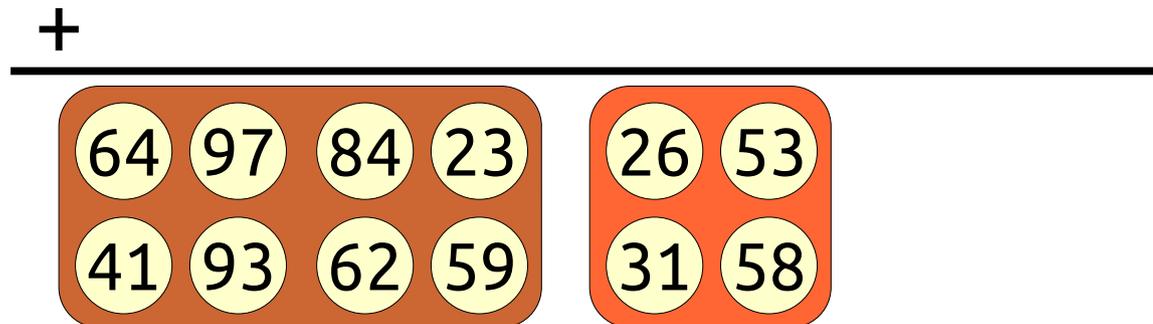


+



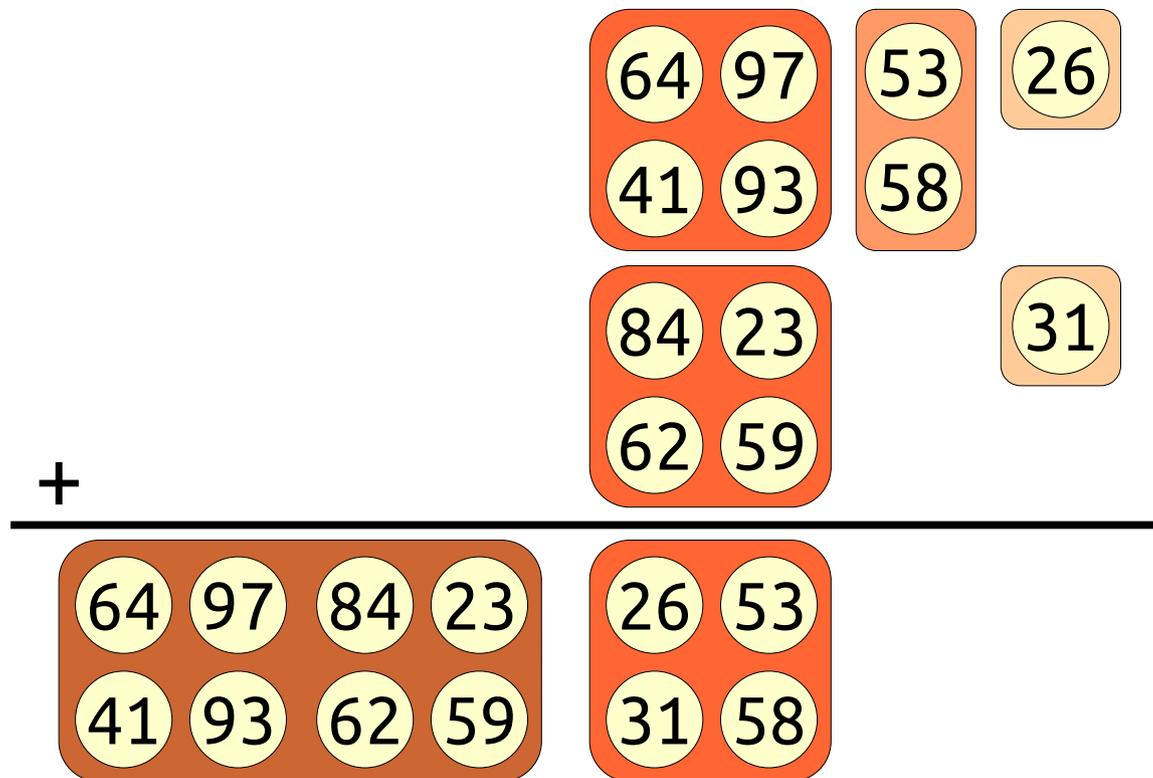
# Building a Priority Queue

- **Idea:** Store elements in “packets” whose sizes are powers of two and **meld** by “adding” groups of packets.



# Building a Priority Queue

- **Idea:** Store elements in “packets” whose sizes are powers of two and *meld* by “adding” groups of packets.

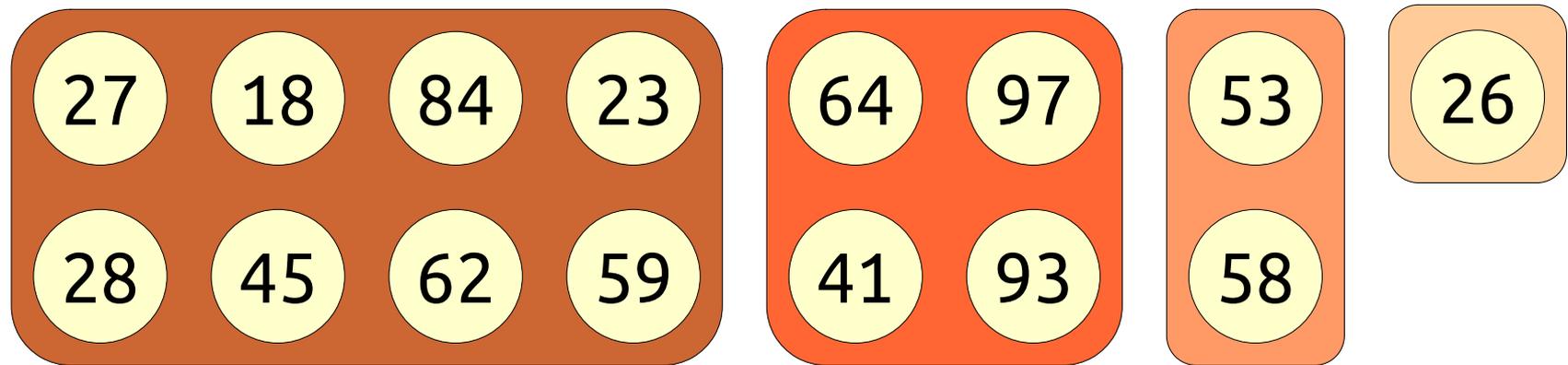


# Building a Priority Queue

- What properties must our packets have?

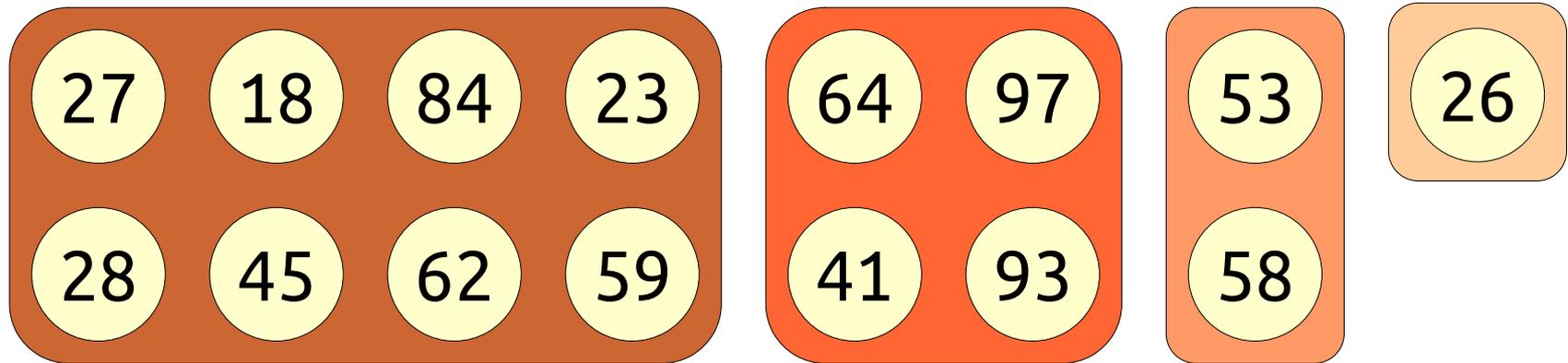
# Building a Priority Queue

- What properties must our packets have?
  - Sizes must be powers of two.



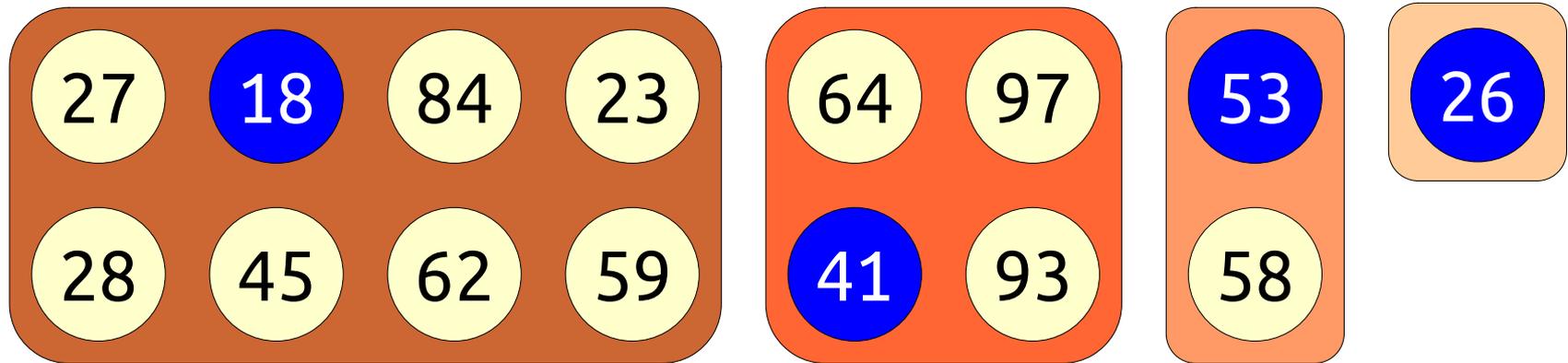
# Building a Priority Queue

- What properties must our packets have?
  - Sizes must be powers of two.
  - Can efficiently fuse packets of the same size.



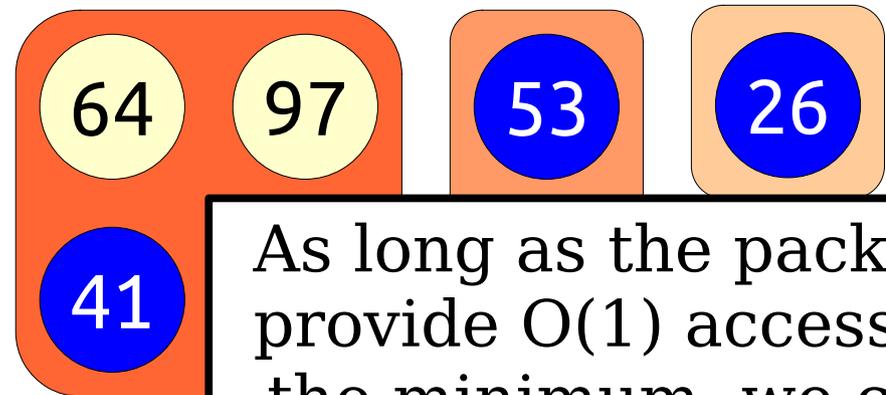
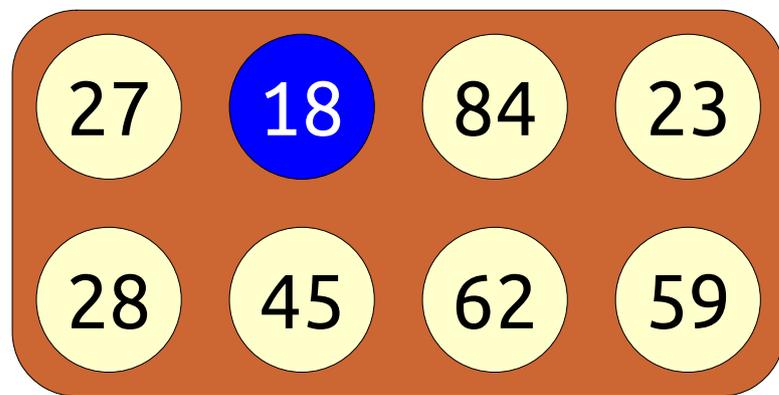
# Building a Priority Queue

- What properties must our packets have?
  - Sizes must be powers of two.
  - Can efficiently fuse packets of the same size.



# Building a Priority Queue

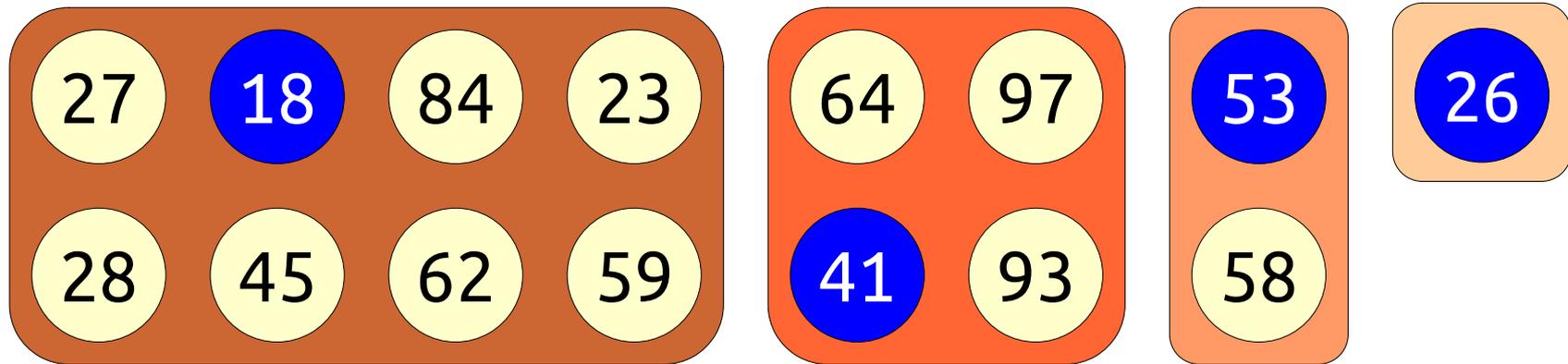
- What properties must our packets have?
  - Sizes must be powers of two.
  - Can efficiently fuse packets of the same size.



As long as the packets provide  $O(1)$  access to the minimum, we can execute *find-min* in time  $O(\log n)$ .

# Building a Priority Queue

- What properties must our packets have?
  - Sizes must be powers of two.
  - Can efficiently fuse packets of the same size.
  - Can efficiently find the minimum element of each packet.

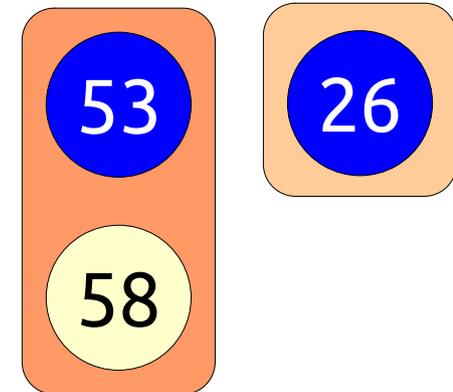
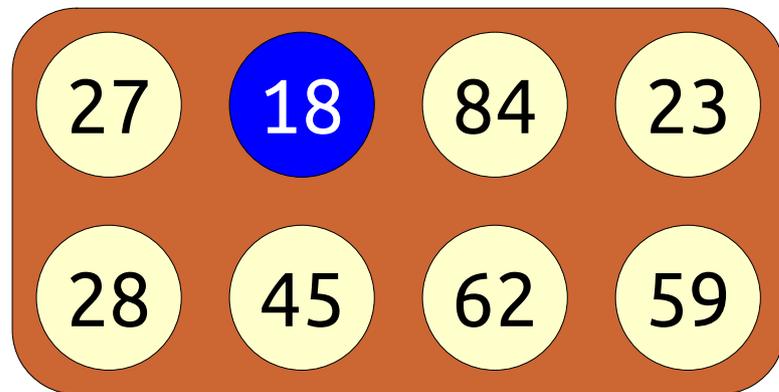


# Inserting into the Queue

- If we can efficiently meld two priority queues, we can efficiently enqueue elements to the queue.
- ***Idea:*** Meld together the queue and a new queue with a single packet.

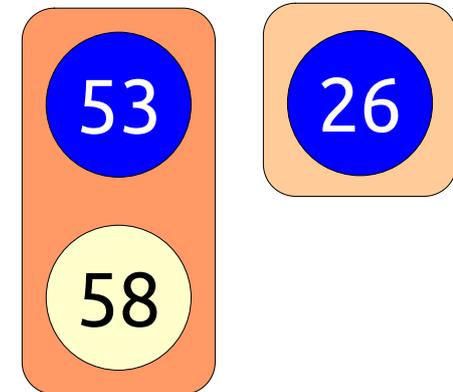
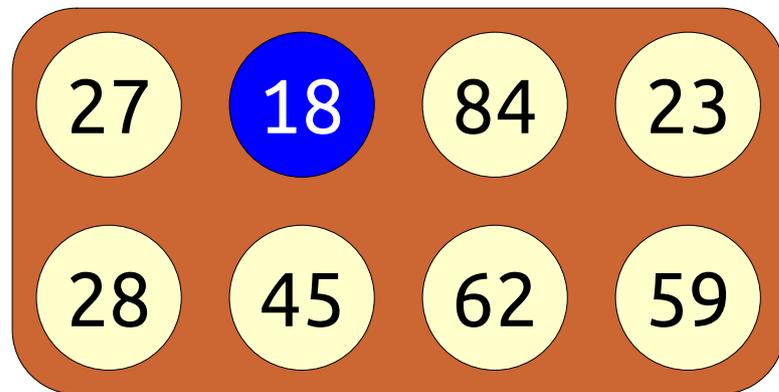
# Inserting into the Queue

- If we can efficiently meld two priority queues, we can efficiently enqueue elements to the queue.
- **Idea:** Meld together the queue and a new queue with a single packet.



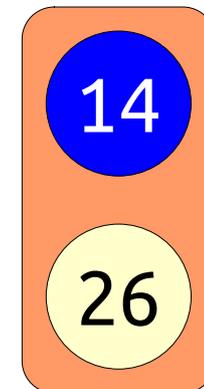
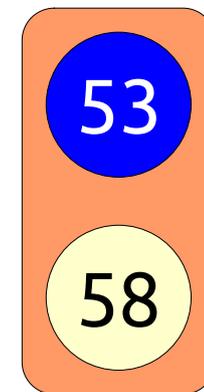
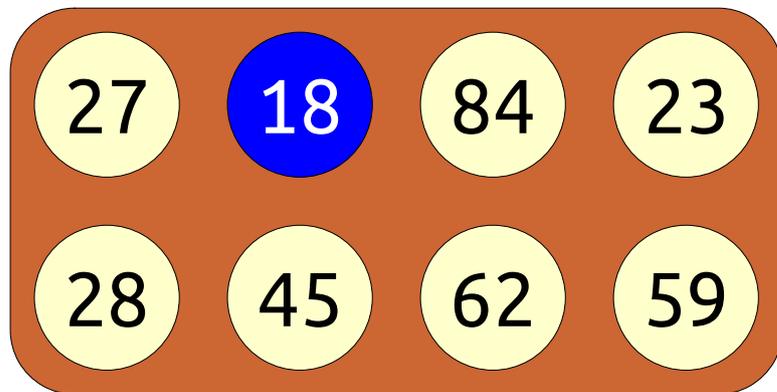
# Inserting into the Queue

- If we can efficiently meld two priority queues, we can efficiently enqueue elements to the queue.
- **Idea:** Meld together the queue and a new queue with a single packet.



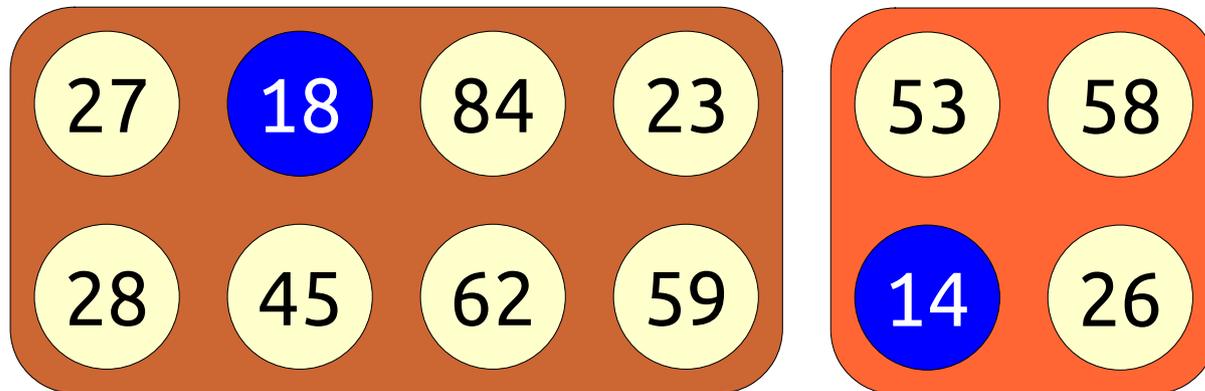
# Inserting into the Queue

- If we can efficiently meld two priority queues, we can efficiently enqueue elements to the queue.
- **Idea:** Meld together the queue and a new queue with a single packet.



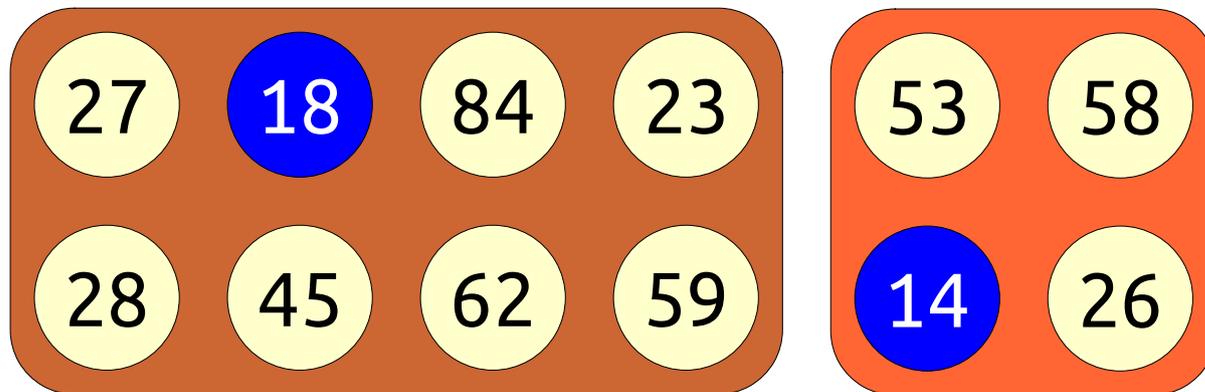
# Inserting into the Queue

- If we can efficiently meld two priority queues, we can efficiently enqueue elements to the queue.
- **Idea:** Meld together the queue and a new queue with a single packet.



# Inserting into the Queue

- If we can efficiently meld two priority queues, we can efficiently enqueue elements to the queue.
- **Idea:** Meld together the queue and a new queue with a single packet.



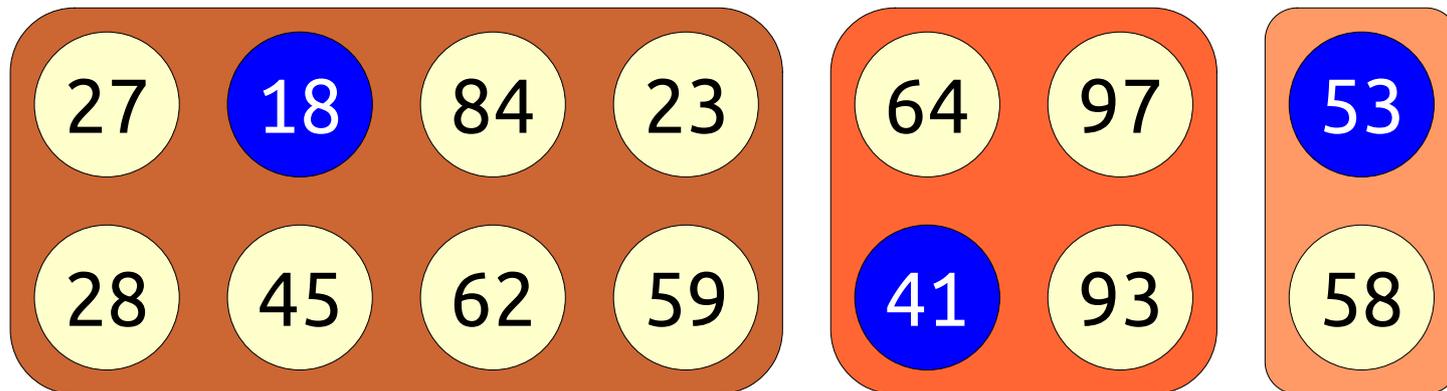
Time required:  
 $O(\log n)$  fuses.

# Deleting the Minimum

- Our analogy with arithmetic breaks down when we try to remove the minimum element.
- After losing an element, the packet will not necessarily hold a number of elements that is a power of two.

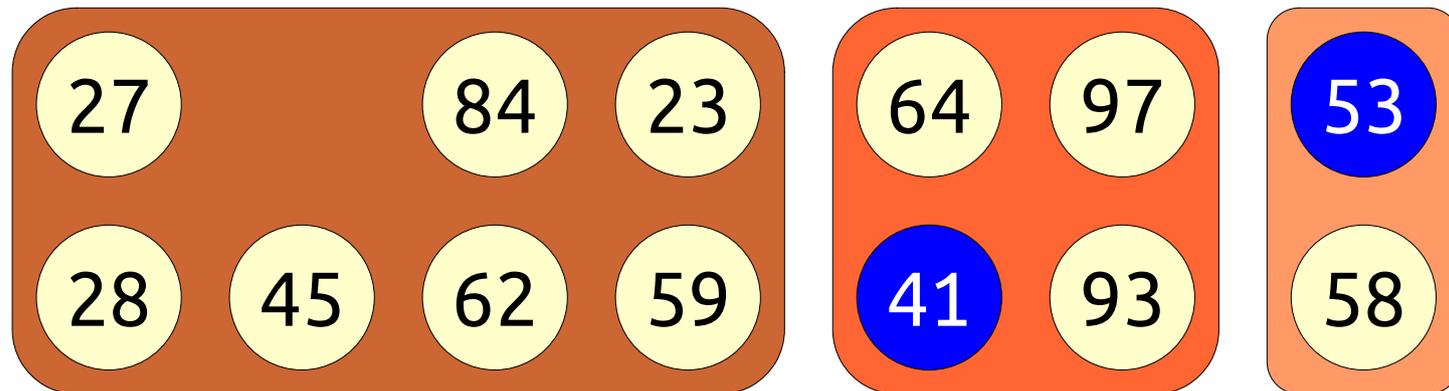
# Deleting the Minimum

- Our analogy with arithmetic breaks down when we try to remove the minimum element.
- After losing an element, the packet will not necessarily hold a number of elements that is a power of two.



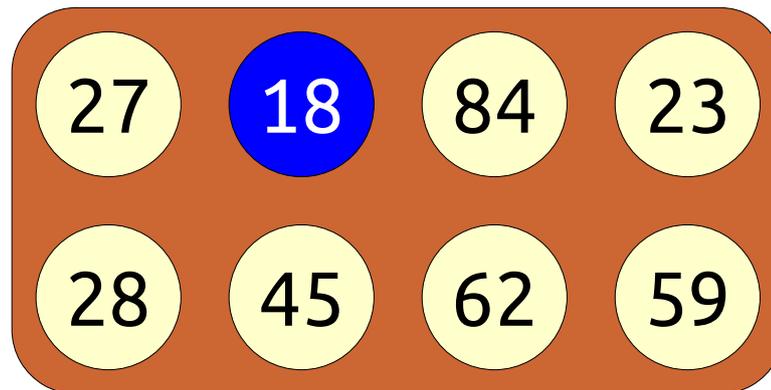
# Deleting the Minimum

- Our analogy with arithmetic breaks down when we try to remove the minimum element.
- After losing an element, the packet will not necessarily hold a number of elements that is a power of two.



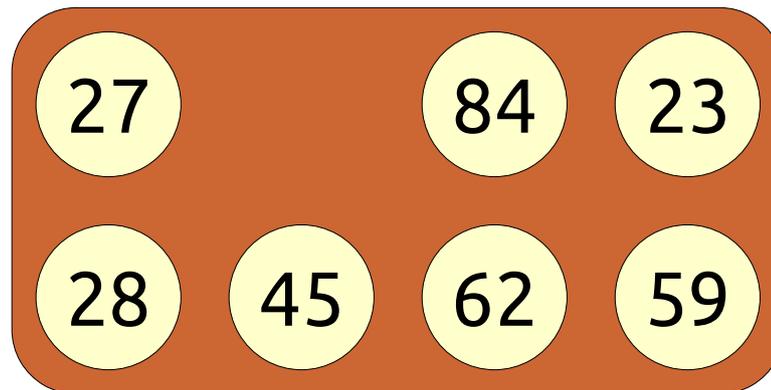
# Deleting the Minimum

- If we have a packet with  $2^k$  elements in it and remove a single element, we are left with  $2^k - 1$  remaining elements.



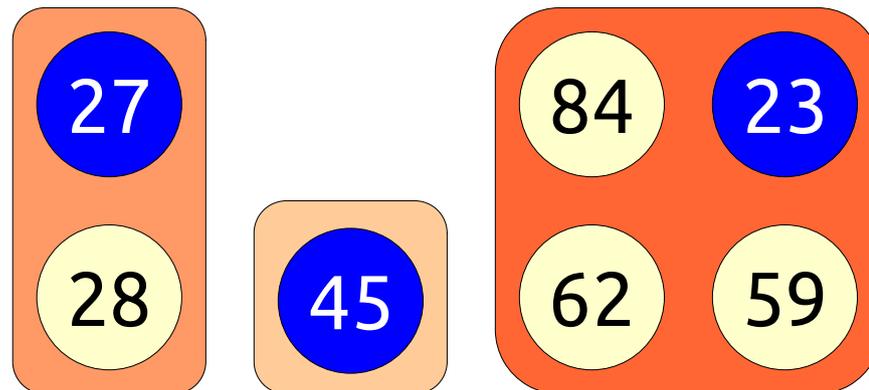
# Deleting the Minimum

- If we have a packet with  $2^k$  elements in it and remove a single element, we are left with  $2^k - 1$  remaining elements.



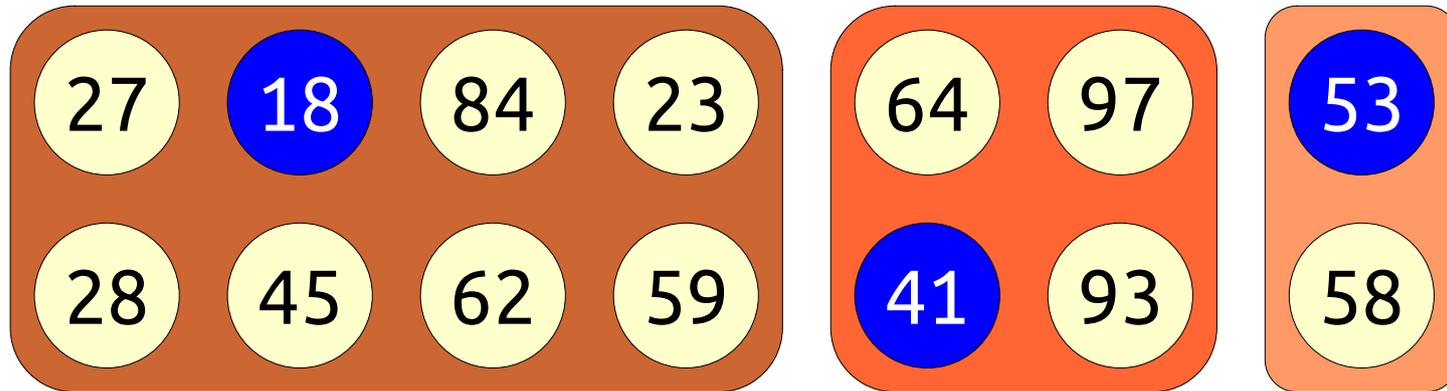
# Deleting the Minimum

- If we have a packet with  $2^k$  elements in it and remove a single element, we are left with  $2^k - 1$  remaining elements.
- **Fun fact:**  $2^k - 1 = 2^0 + 2^1 + 2^2 + \dots + 2^{k-1}$ .
- **Idea:** “Fracture” the packet into  $k$  smaller packets, then add them back in.



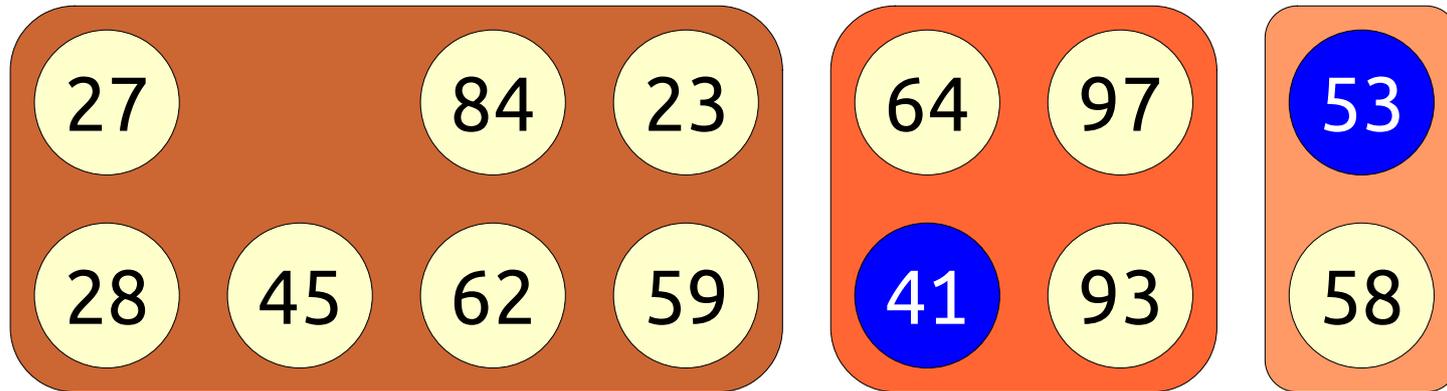
# Fracturing Packets

- We can *extract-min* by fracturing the packet containing the minimum and adding the fragments back in.



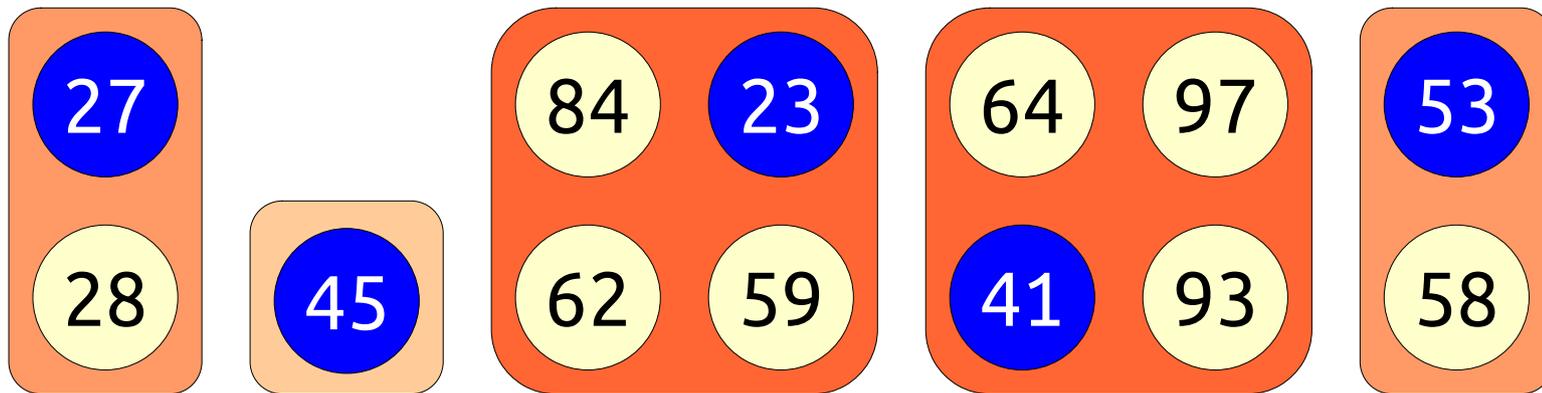
# Fracturing Packets

- We can *extract-min* by fracturing the packet containing the minimum and adding the fragments back in.



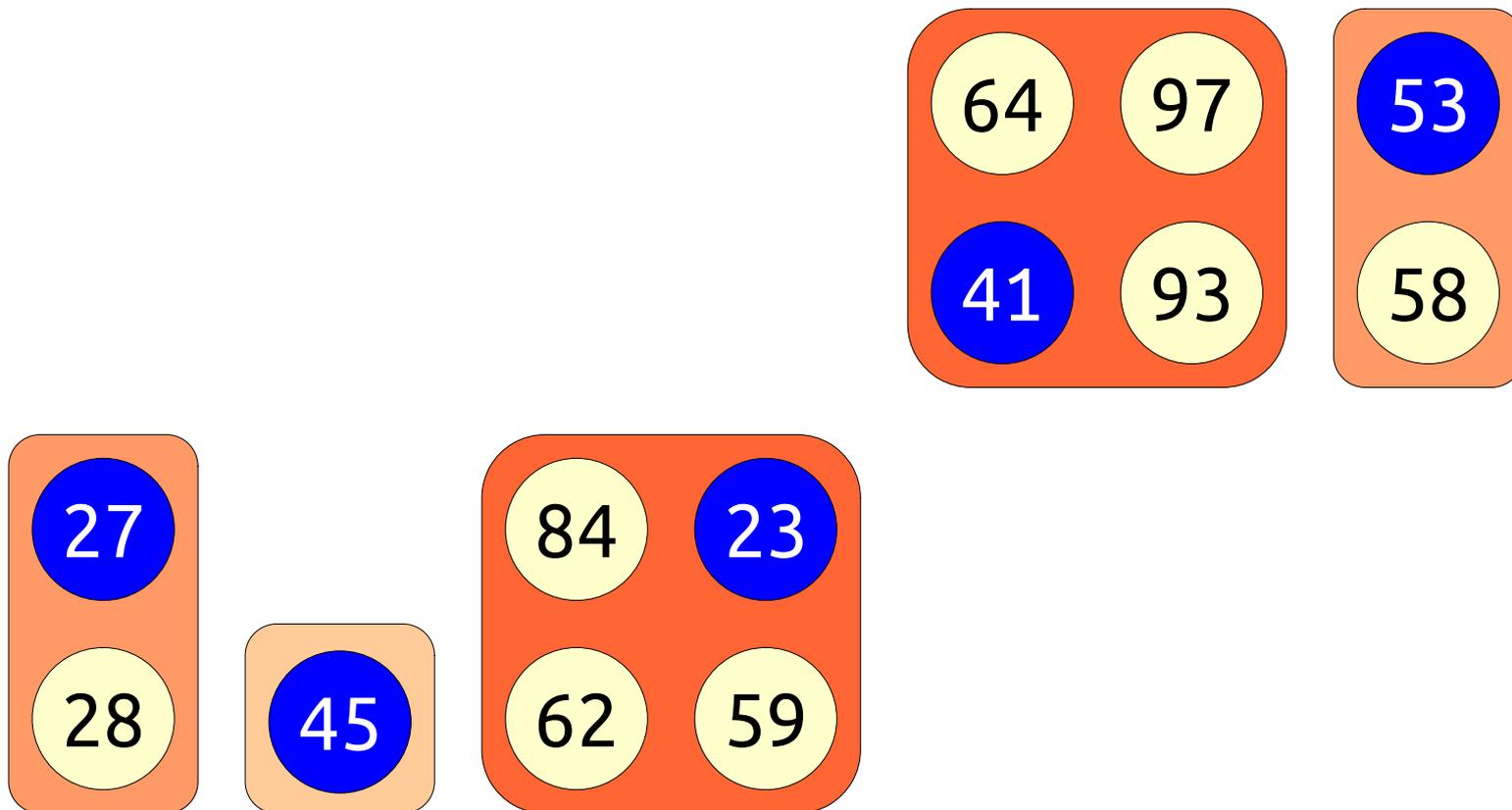
# Fracturing Packets

- We can *extract-min* by fracturing the packet containing the minimum and adding the fragments back in.



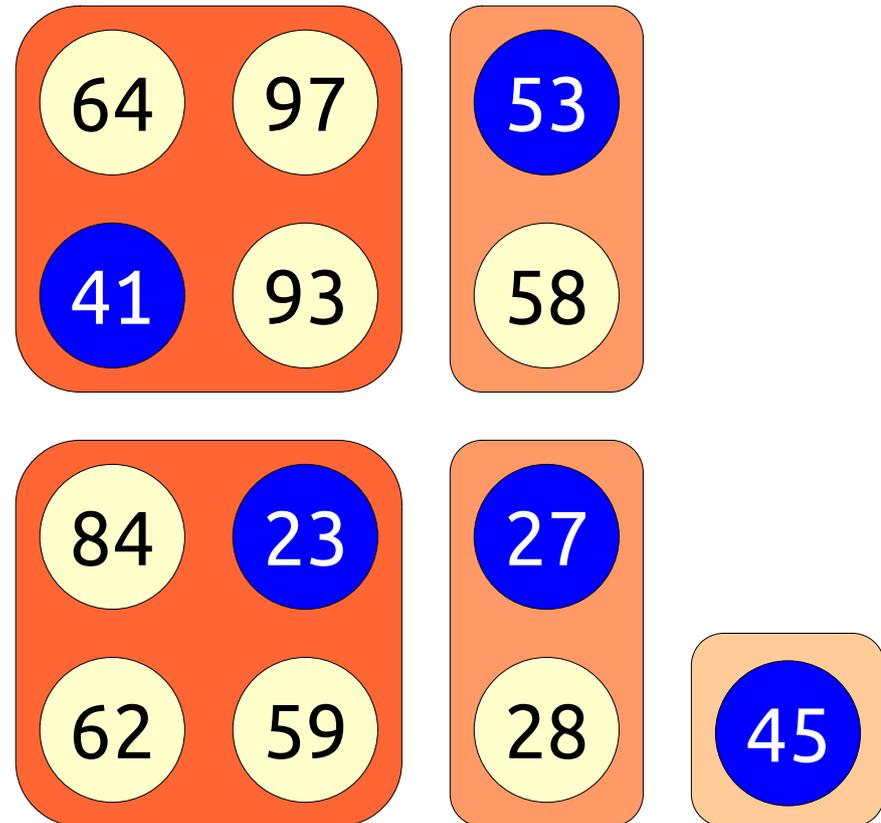
# Fracturing Packets

- We can *extract-min* by fracturing the packet containing the minimum and adding the fragments back in.



# Fracturing Packets

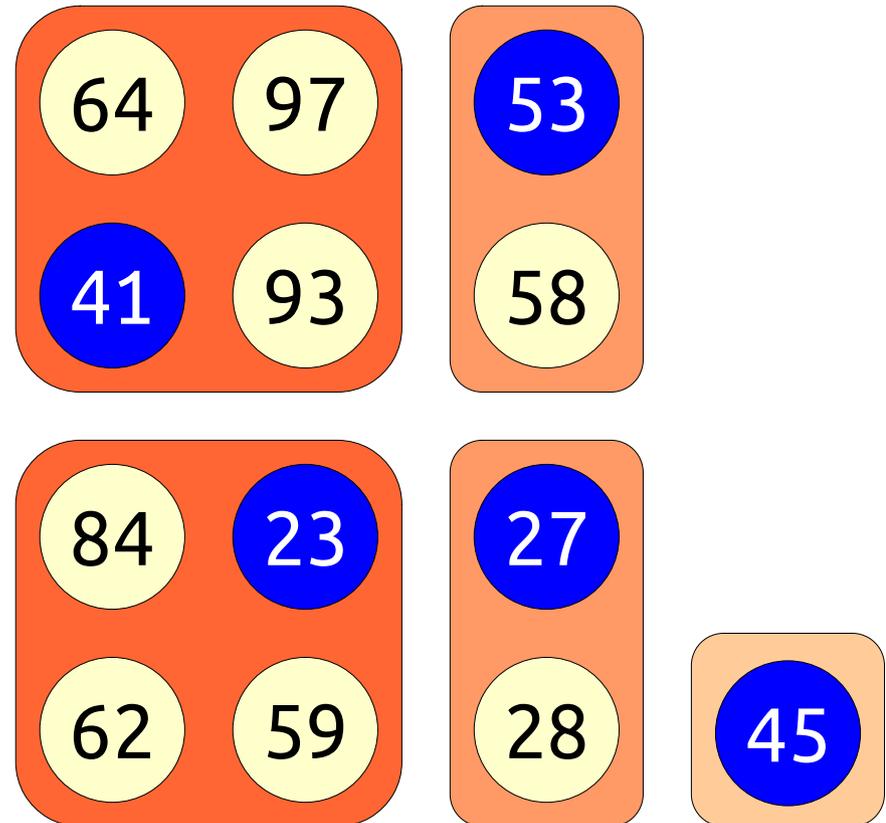
- We can *extract-min* by fracturing the packet containing the minimum and adding the fragments back in.



# Fracturing Packets

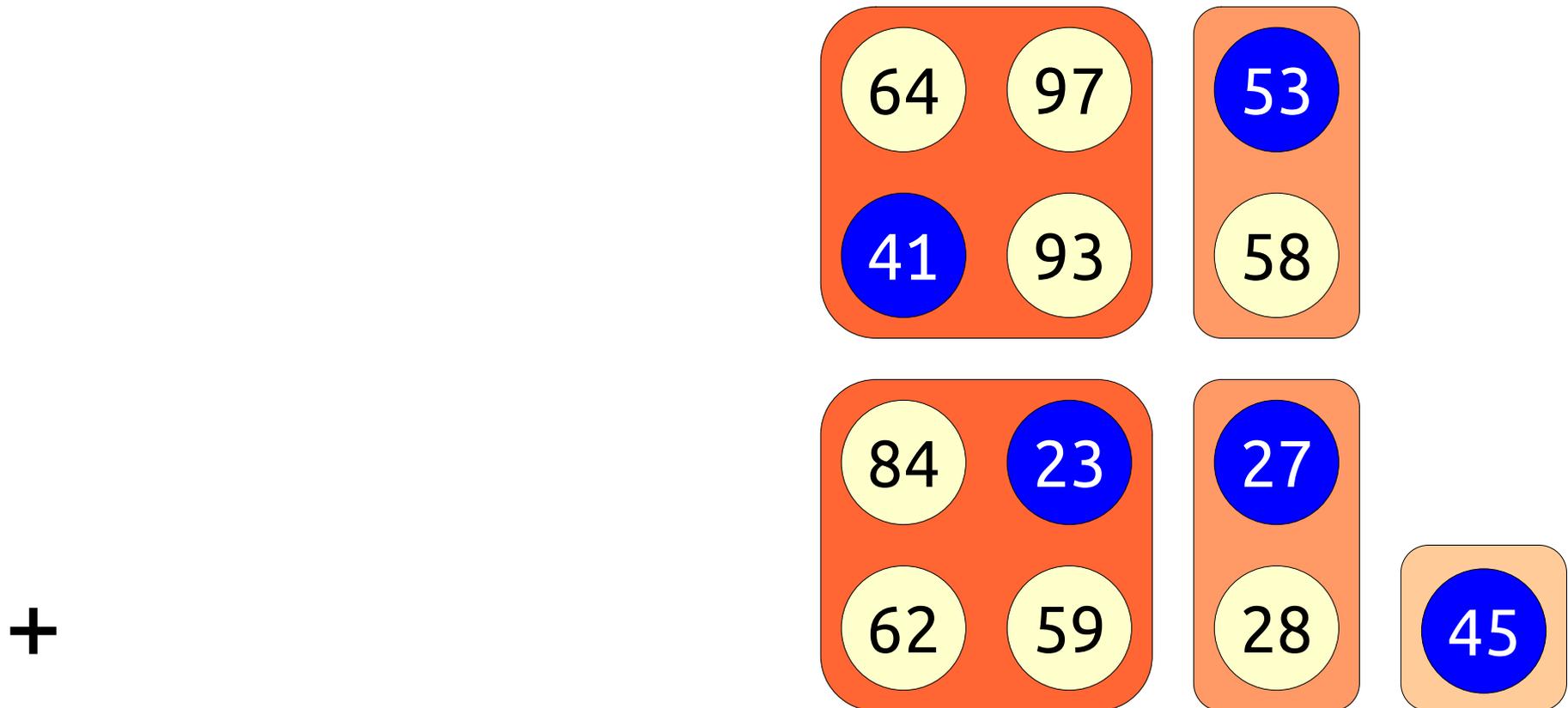
- We can *extract-min* by fracturing the packet containing the minimum and adding the fragments back in.

+



# Fracturing Packets

- We can *extract-min* by fracturing the packet containing the minimum and adding the fragments back in.
- Runtime is  $O(\log n)$  fuses in *meld*, plus fracture cost.



# Building a Priority Queue

- What properties must our packets have?
  - Size is a power of two.
  - Can efficiently fuse packets of the same size.
  - Can efficiently find the minimum element of each packet.
  - Can efficiently “fracture” a packet of  $2^k$  nodes into packets of  $2^0, 2^1, 2^2, 2^3, \dots, 2^{k-1}$  nodes.
- **Question:** How can we represent our packets to support the above operations efficiently?

Formulate a hypothesis!

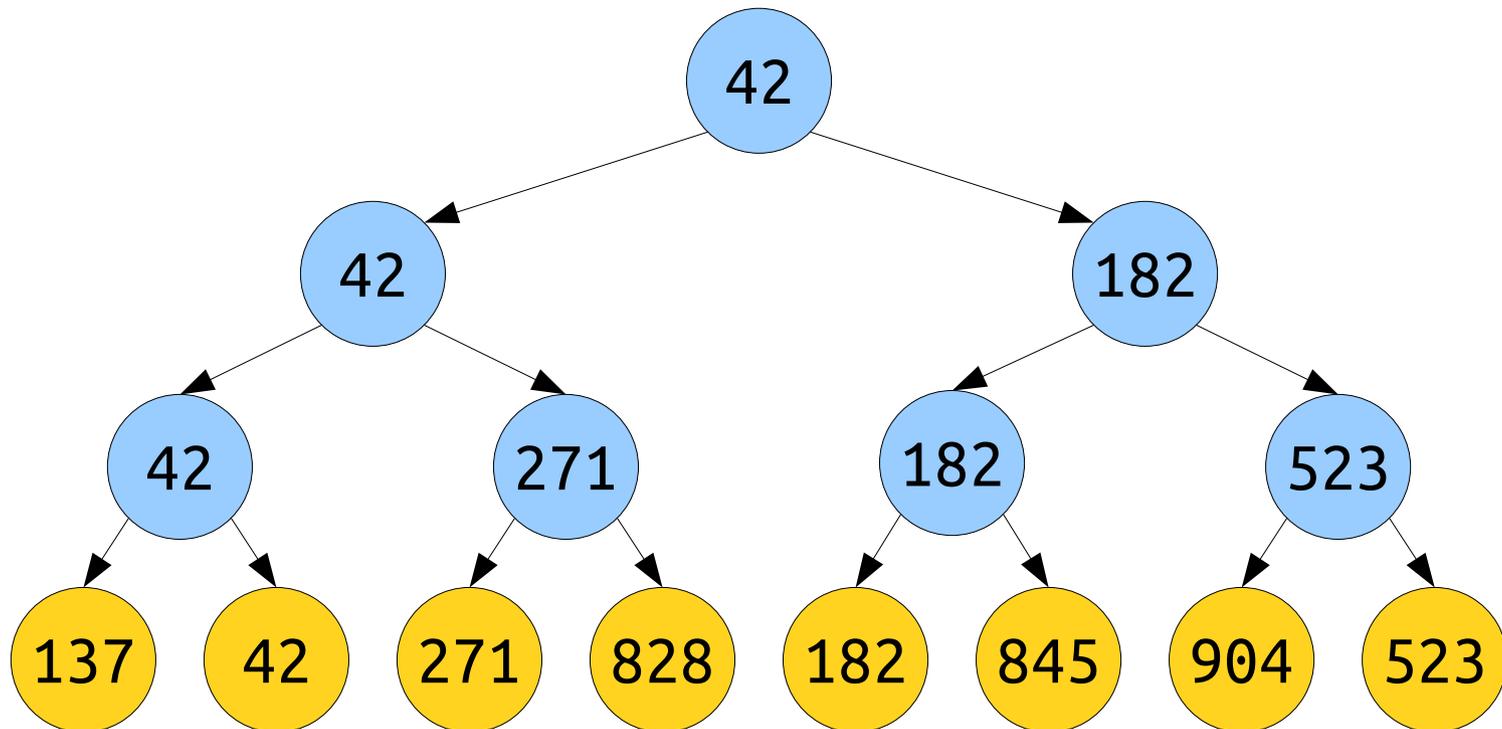
# Building a Priority Queue

- What properties must our packets have?
  - Size is a power of two.
  - Can efficiently fuse packets of the same size.
  - Can efficiently find the minimum element of each packet.
  - Can efficiently “fracture” a packet of  $2^k$  nodes into packets of  $2^0, 2^1, 2^2, 2^3, \dots, 2^{k-1}$  nodes.
- **Question:** How can we represent our packets to support the above operations efficiently?

Discuss with your  
neighbors!

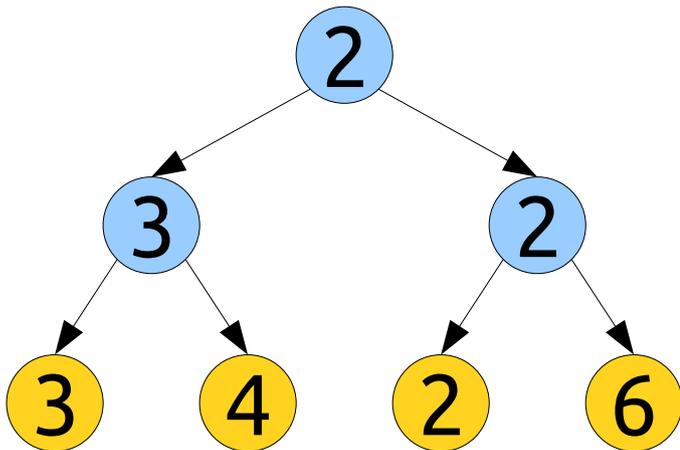
# Tournament Trees

- A ***tournament tree*** is a complete binary tree representing the result of a tournament.
- Each leaf represents a single “player” in the tournament.
- Each internal node shows the “winner” of the game played by those players.



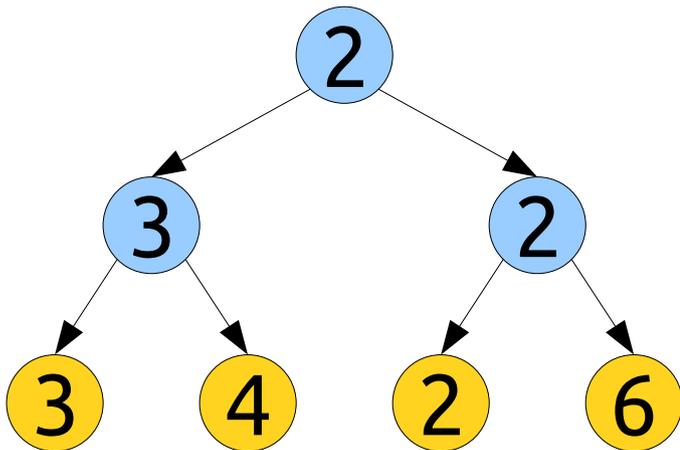
# Tournament Trees

- What properties must our packets have?
  - Size must be a power of two.
  - Can efficiently fuse packets of the same size.
  - Can efficiently find the minimum element of each packet.
  - Can efficiently “fracture” a packet of  $2^k$  nodes into packets of  $2^0, 2^1, 2^2, 2^3, \dots, 2^{k-1}$  nodes.



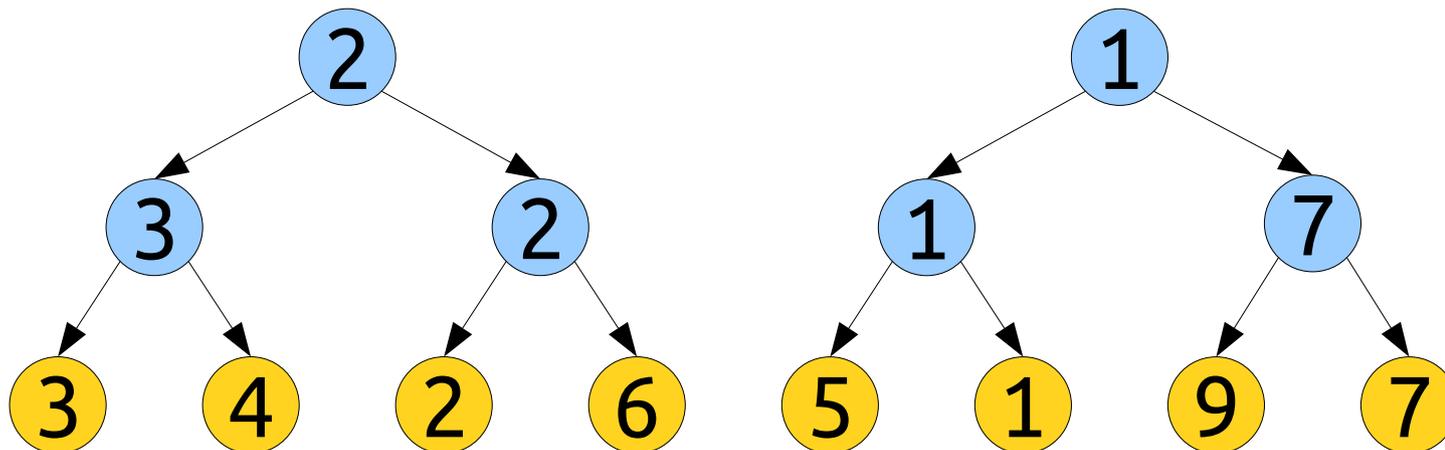
# Tournament Trees

- What properties must our packets have?
  - Size must be a power of two. ✓
  - Can efficiently fuse packets of the same size.
  - Can efficiently find the minimum element of each packet.
  - Can efficiently “fracture” a packet of  $2^k$  nodes into packets of  $2^0, 2^1, 2^2, 2^3, \dots, 2^{k-1}$  nodes.



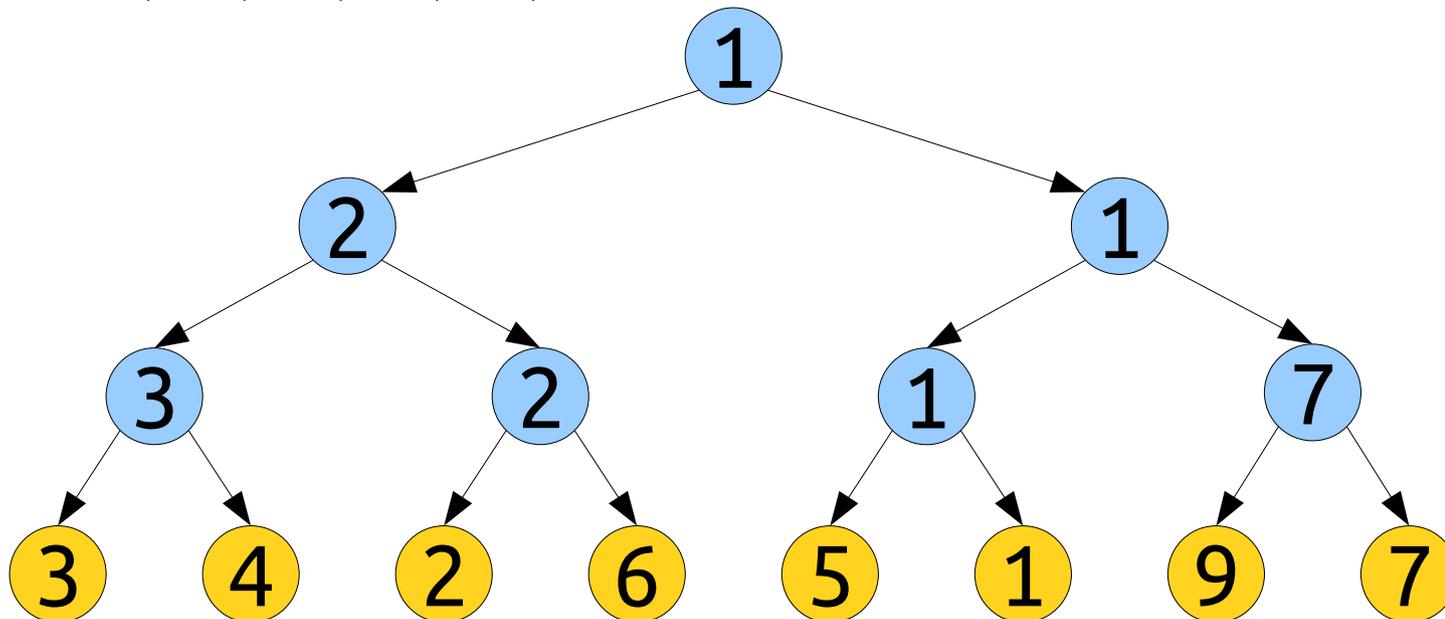
# Tournament Trees

- What properties must our packets have?
  - Size must be a power of two. ✓
  - Can efficiently fuse packets of the same size.
  - Can efficiently find the minimum element of each packet.
  - Can efficiently “fracture” a packet of  $2^k$  nodes into packets of  $2^0, 2^1, 2^2, 2^3, \dots, 2^{k-1}$  nodes.



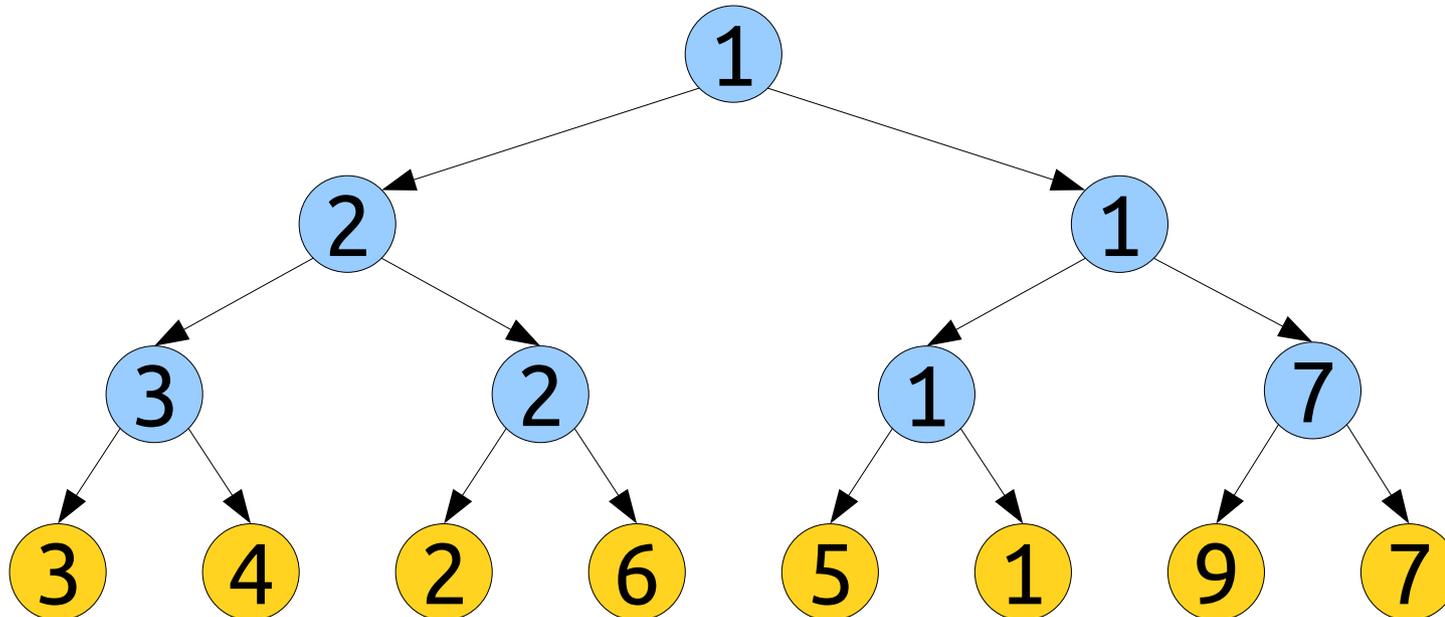
# Tournament Trees

- What properties must our packets have?
  - Size must be a power of two. ✓
  - Can efficiently fuse packets of the same size.
  - Can efficiently find the minimum element of each packet.
  - Can efficiently “fracture” a packet of  $2^k$  nodes into packets of  $2^0, 2^1, 2^2, 2^3, \dots, 2^{k-1}$  nodes.



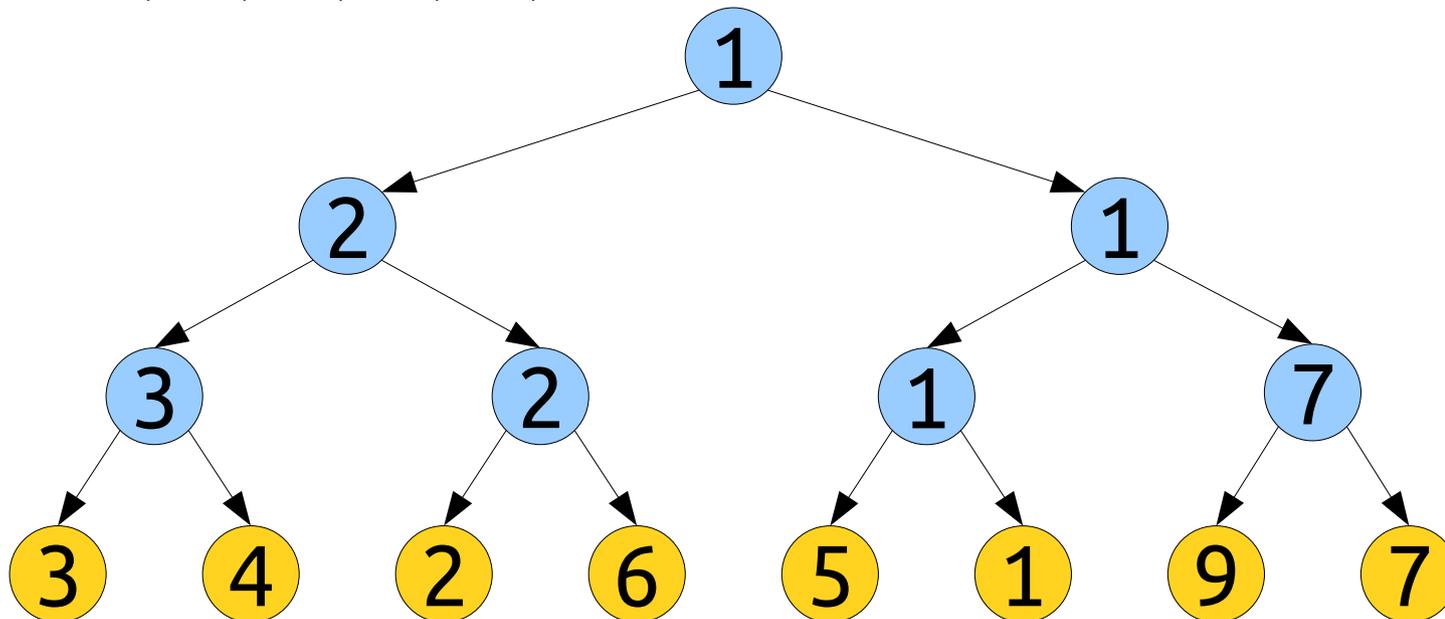
# Tournament Trees

- What properties must our packets have?
  - Size must be a power of two. ✓
  - Can efficiently fuse packets of the same size. ✓
  - Can efficiently find the minimum element of each packet.
  - Can efficiently “fracture” a packet of  $2^k$  nodes into packets of  $2^0, 2^1, 2^2, 2^3, \dots, 2^{k-1}$  nodes.



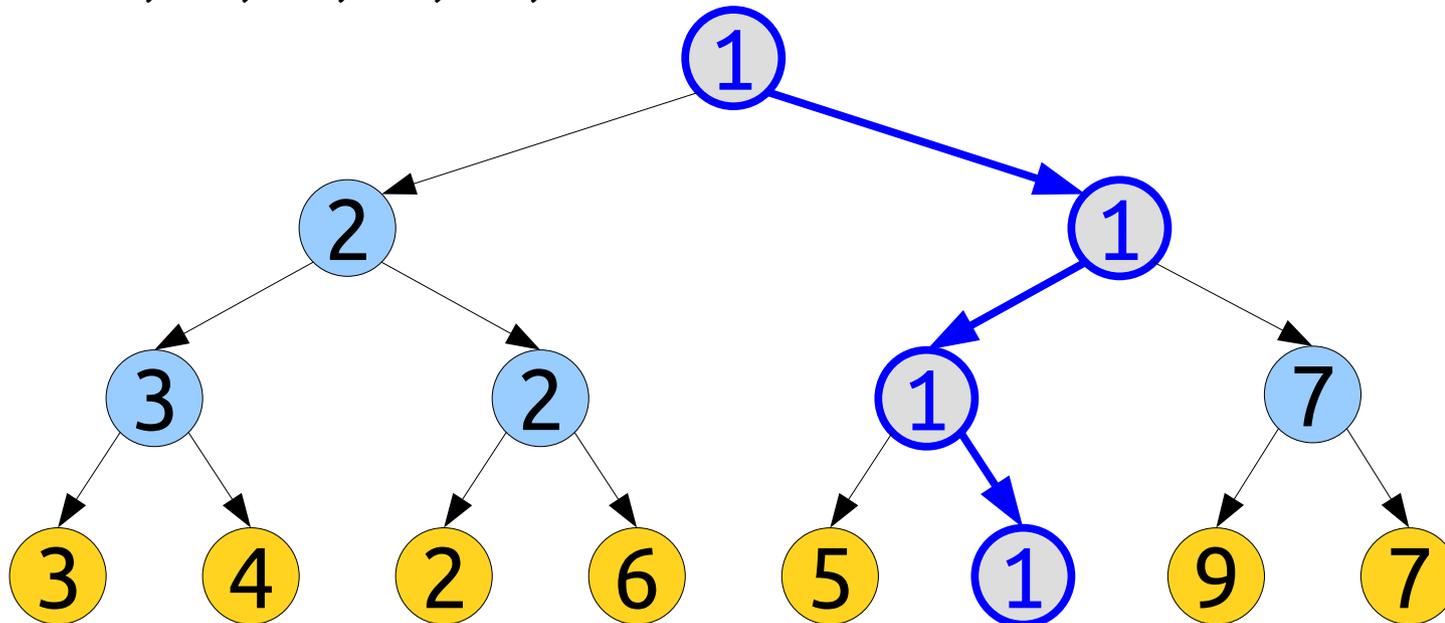
# Tournament Trees

- What properties must our packets have?
  - Size must be a power of two. ✓
  - Can efficiently fuse packets of the same size. ✓
  - Can efficiently find the minimum element of each packet. ✓
  - Can efficiently “fracture” a packet of  $2^k$  nodes into packets of  $2^0, 2^1, 2^2, 2^3, \dots, 2^{k-1}$  nodes.



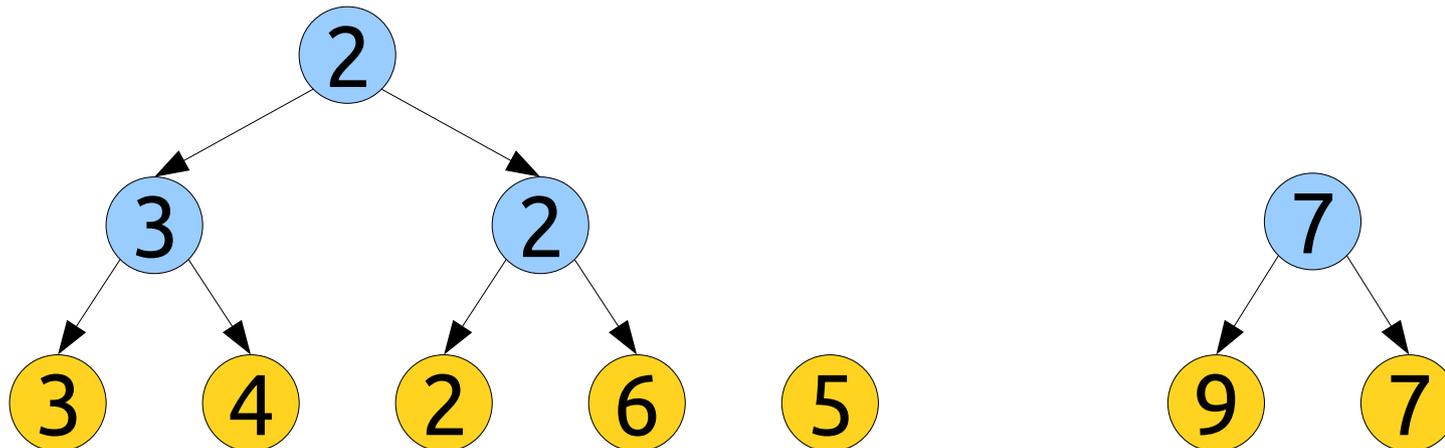
# Tournament Trees

- What properties must our packets have?
  - Size must be a power of two. ✓
  - Can efficiently fuse packets of the same size. ✓
  - Can efficiently find the minimum element of each packet. ✓
  - Can efficiently “fracture” a packet of  $2^k$  nodes into packets of  $2^0, 2^1, 2^2, 2^3, \dots, 2^{k-1}$  nodes.



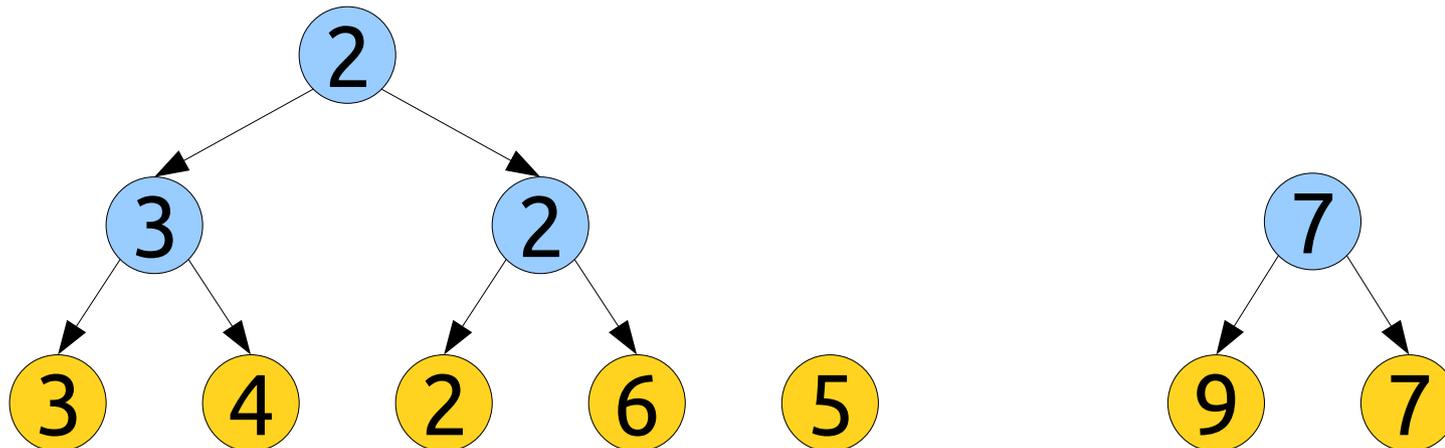
# Tournament Trees

- What properties must our packets have?
  - Size must be a power of two. ✓
  - Can efficiently fuse packets of the same size. ✓
  - Can efficiently find the minimum element of each packet. ✓
  - Can efficiently “fracture” a packet of  $2^k$  nodes into packets of  $2^0, 2^1, 2^2, 2^3, \dots, 2^{k-1}$  nodes.



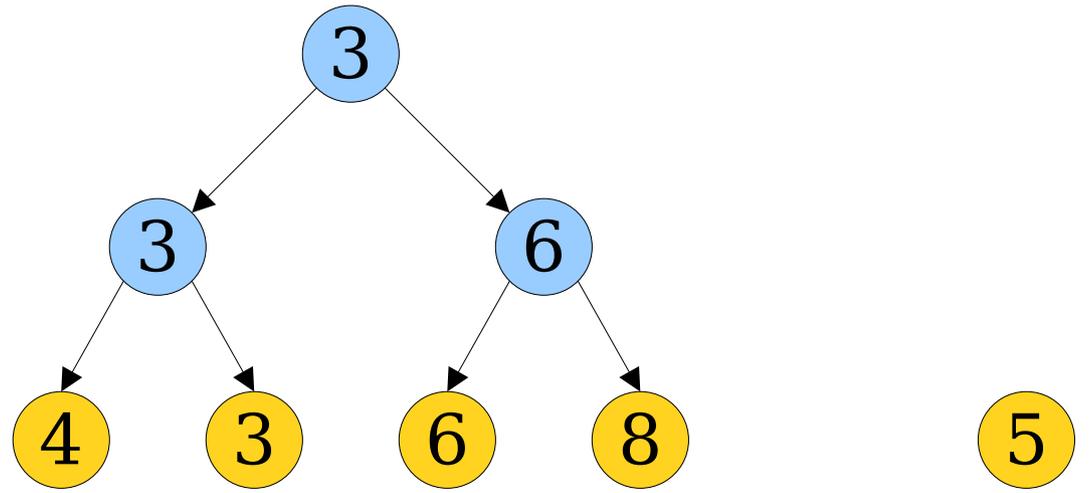
# Tournament Trees

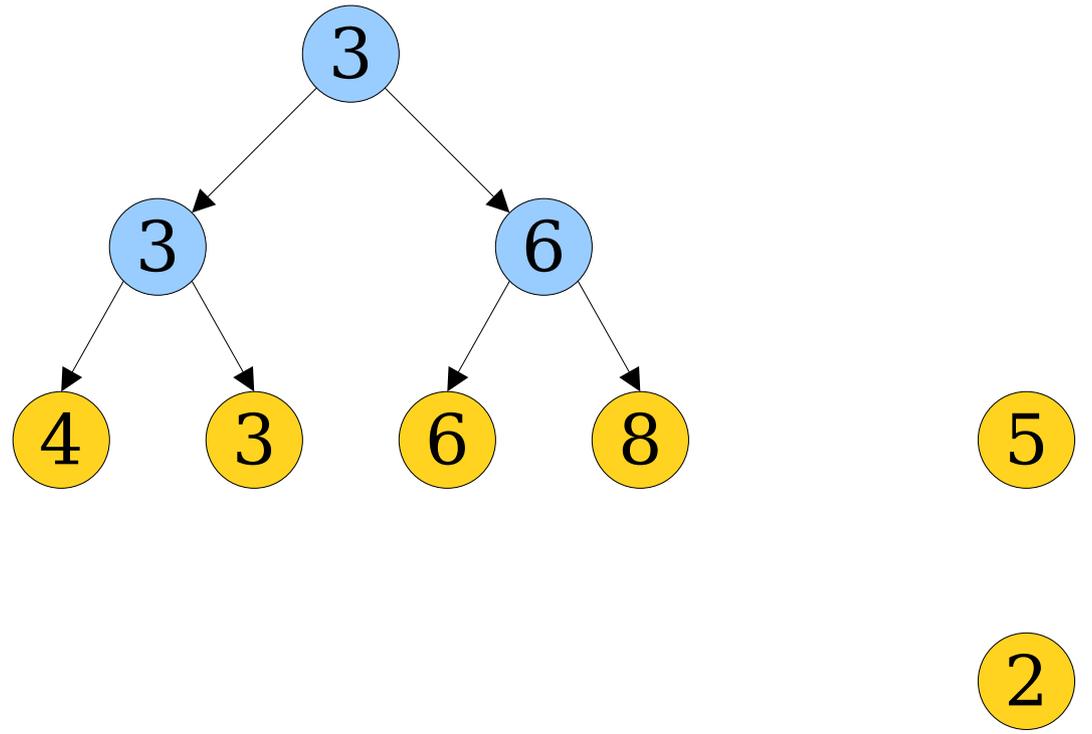
- What properties must our packets have?
  - Size must be a power of two. ✓
  - Can efficiently fuse packets of the same size. ✓
  - Can efficiently find the minimum element of each packet. ✓
  - Can efficiently “fracture” a packet of  $2^k$  nodes into packets of  $2^0, 2^1, 2^2, 2^3, \dots, 2^{k-1}$  nodes. ✓

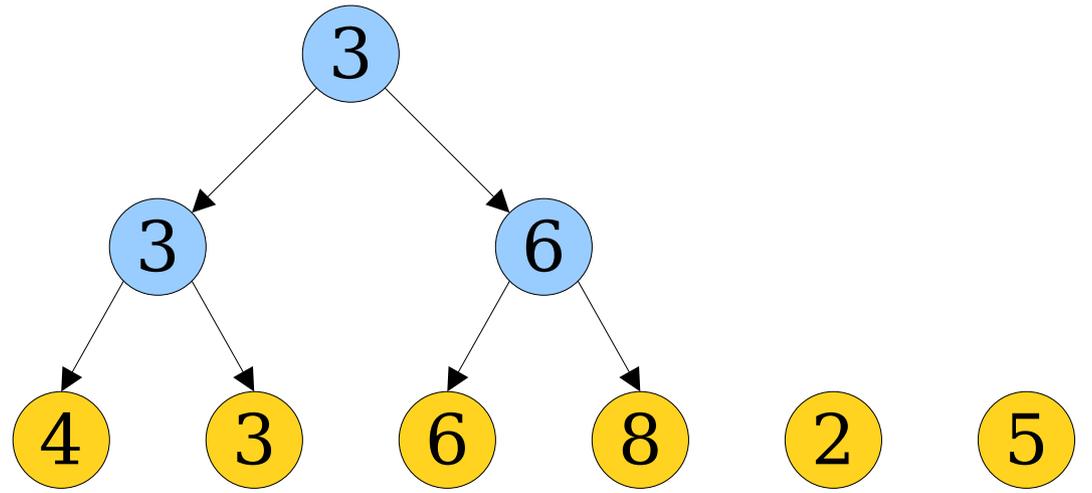


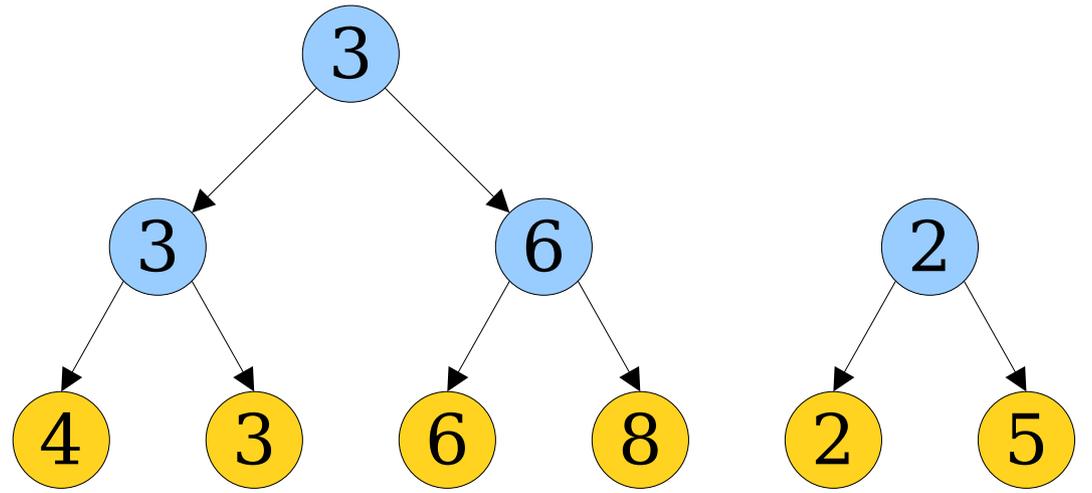
# The Tournament Heap

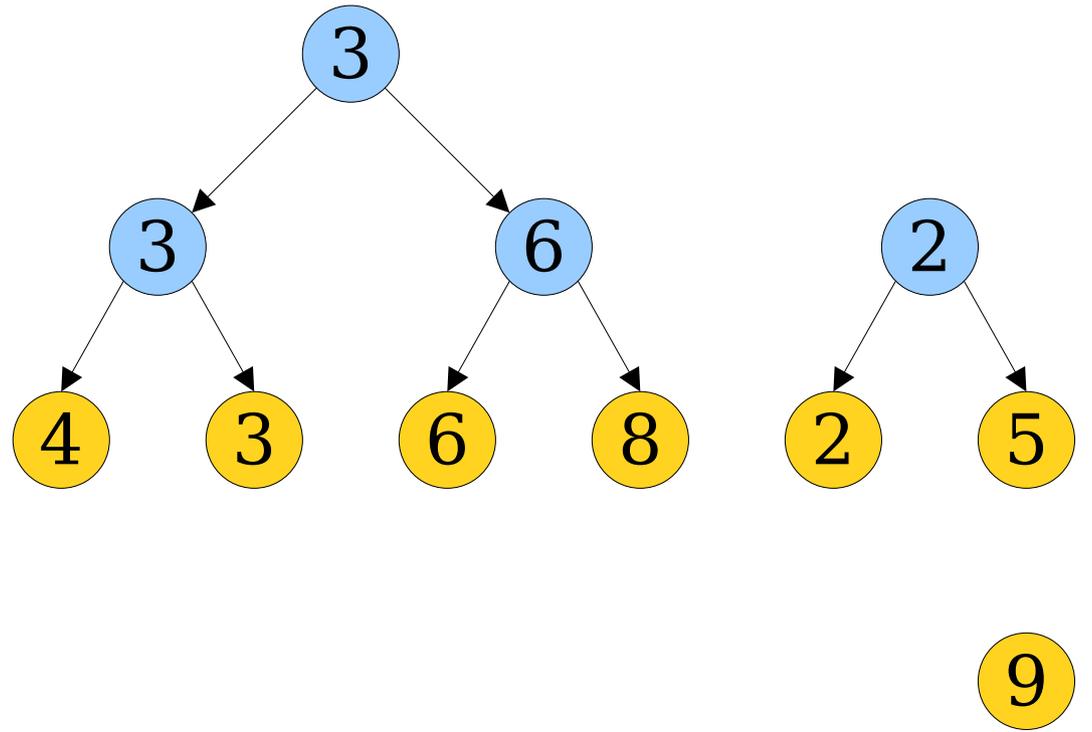
- A **tournament heap** is a collection of tournament trees stored in ascending order of size.
- Operations defined as follows:
  - **meld**( $pq_1, pq_2$ ): Use addition to combine all the trees.
    - Fuses  $O(\log n + \log m)$  trees. Cost:  $O(\log n + \log m)$ . Here, assume one tournament heap has  $n$  nodes, the other  $m$ .
  - $pq$ .**enqueue**( $v, k$ ): Meld  $pq$  and a singleton heap of  $(v, k)$ .
    - Total time:  $O(\log n)$ .
  - $pq$ .**find-min**(): Find the minimum of all tree roots.
    - Total time:  $O(\log n)$ .
  - $pq$ .**extract-min**(): Find the min, delete the tree root, then meld together the queue and the exposed children.
    - Total time:  $O(\log n)$ .

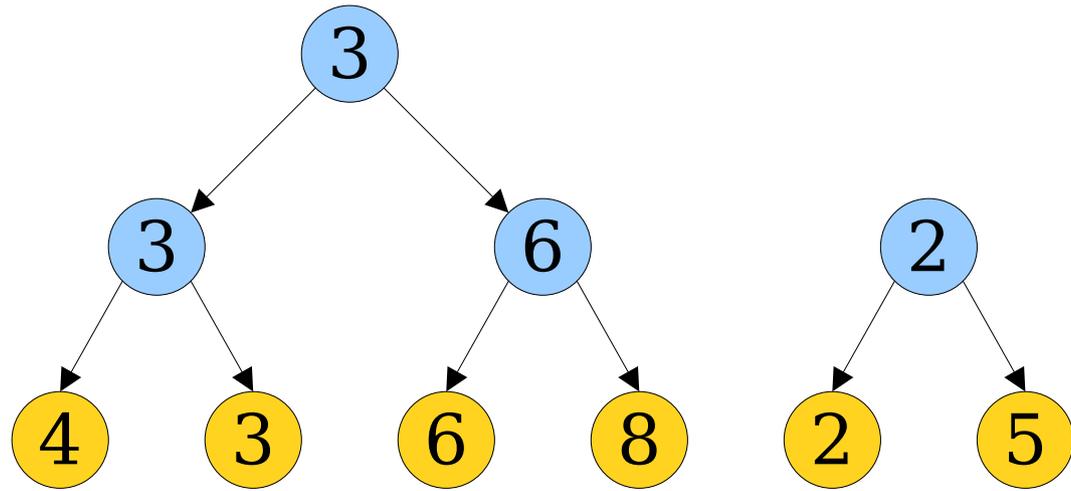




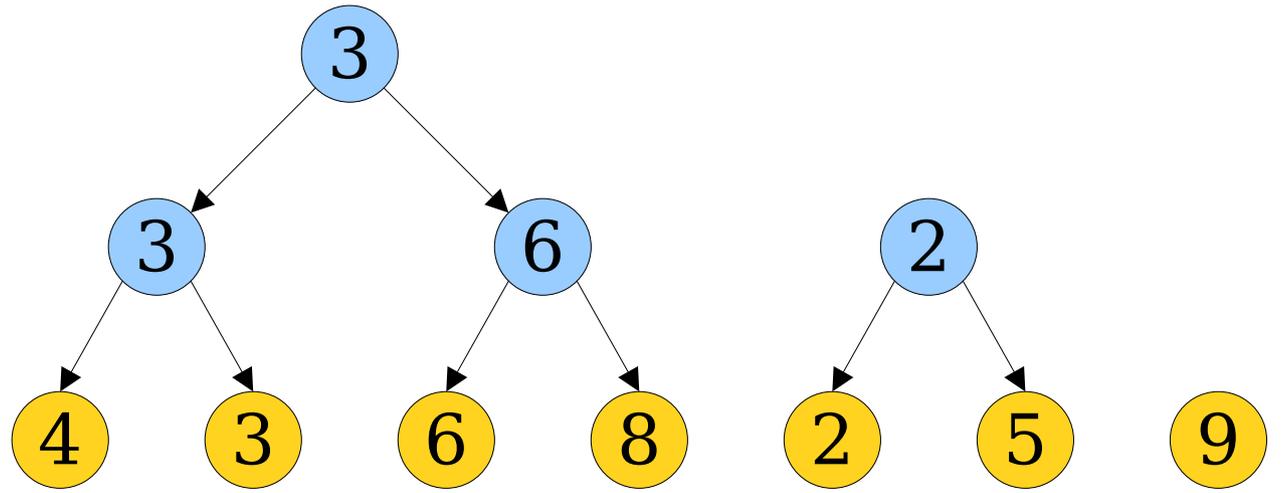


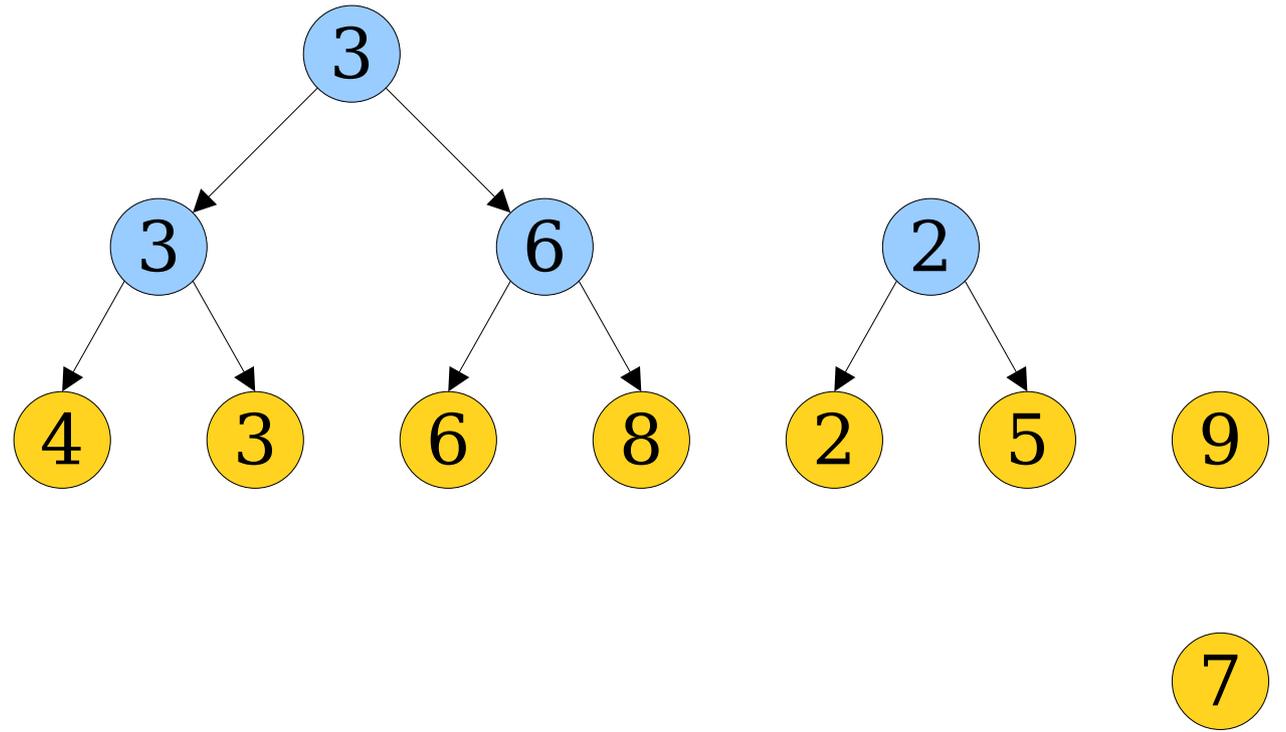


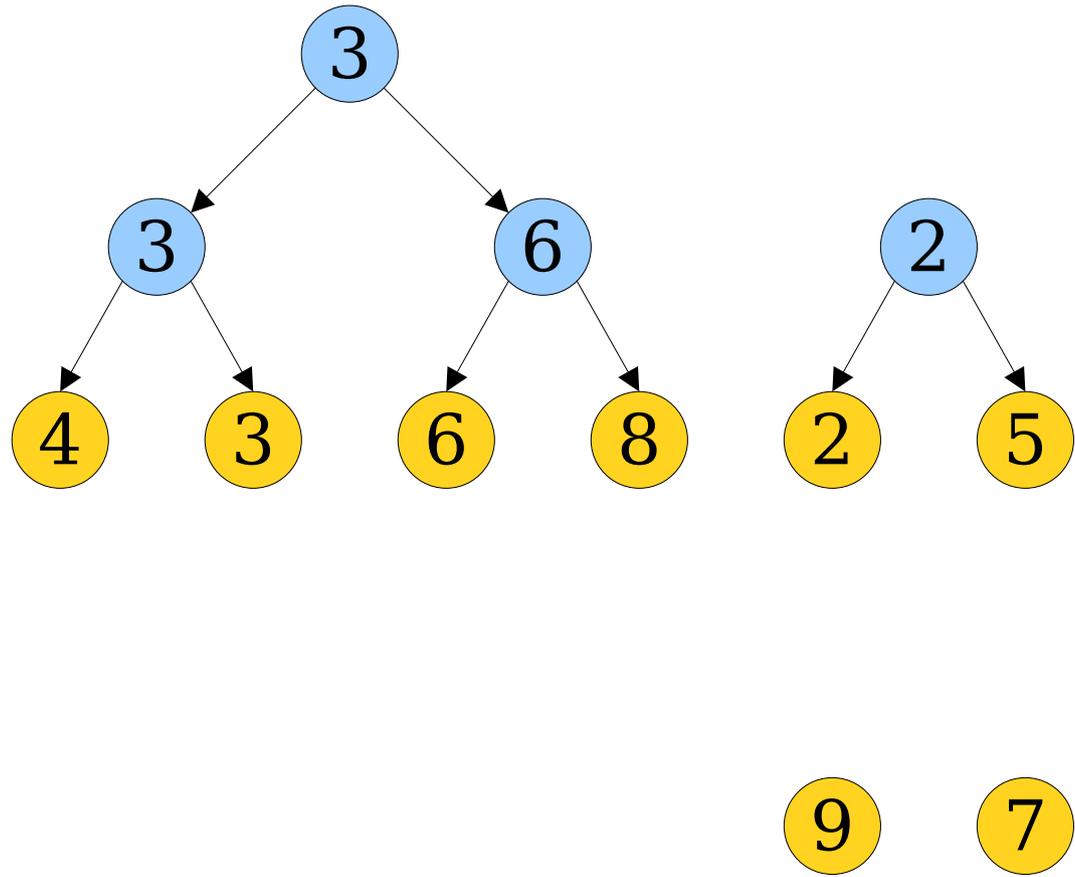


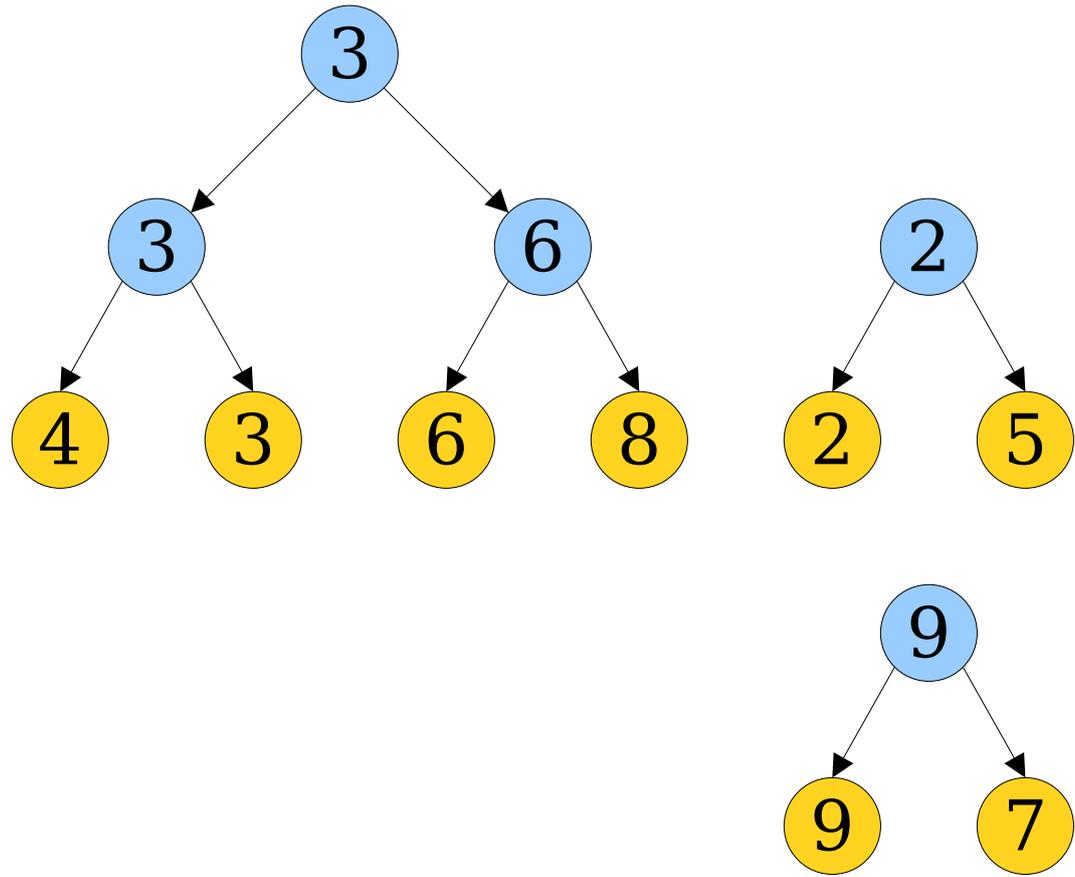


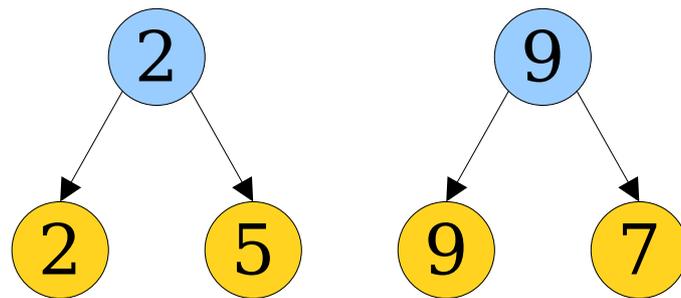
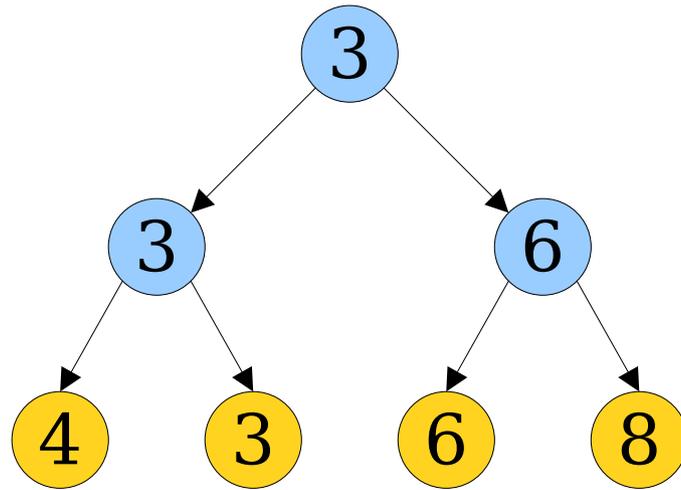
9

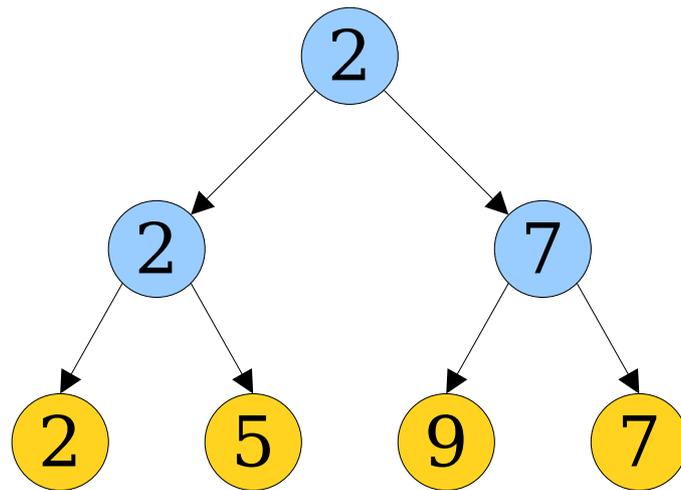
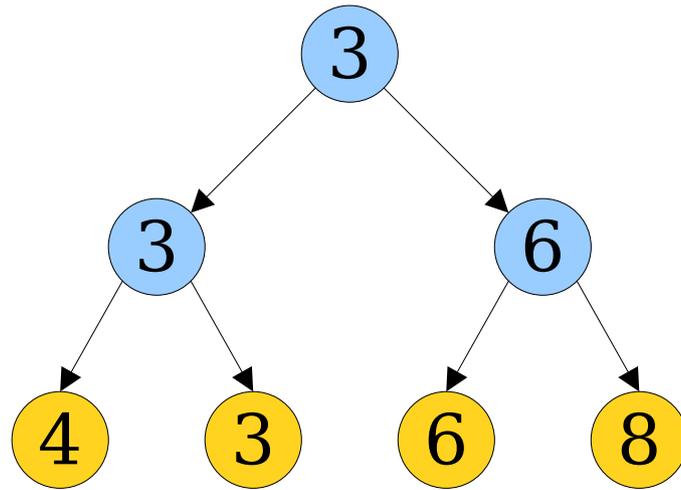


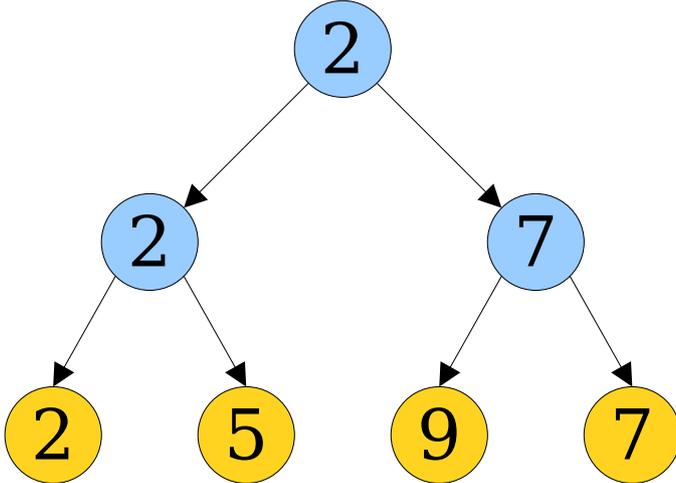
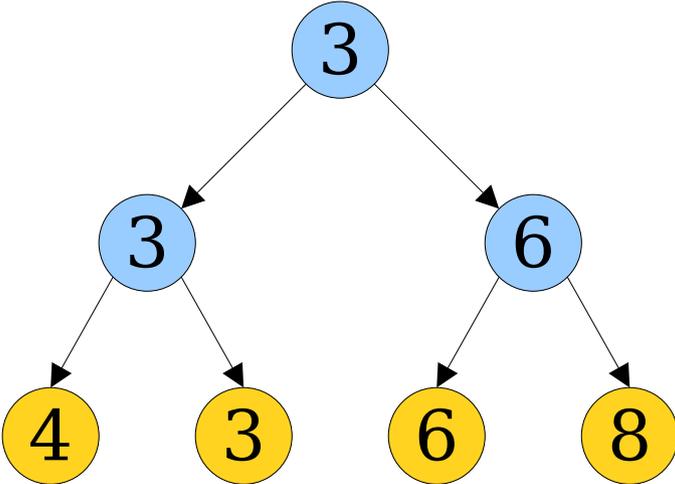


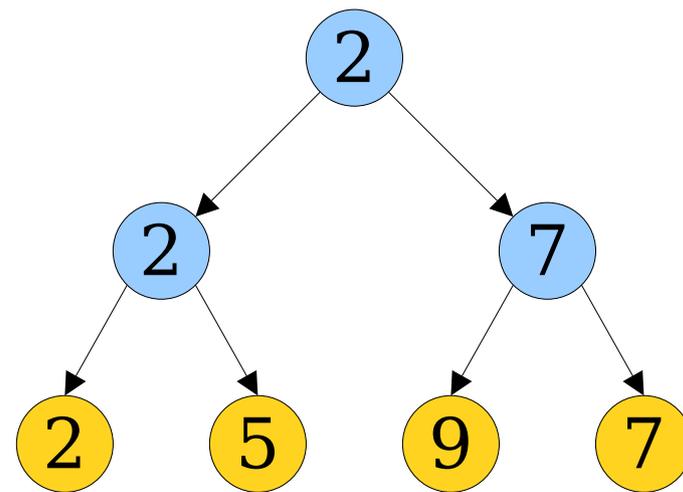
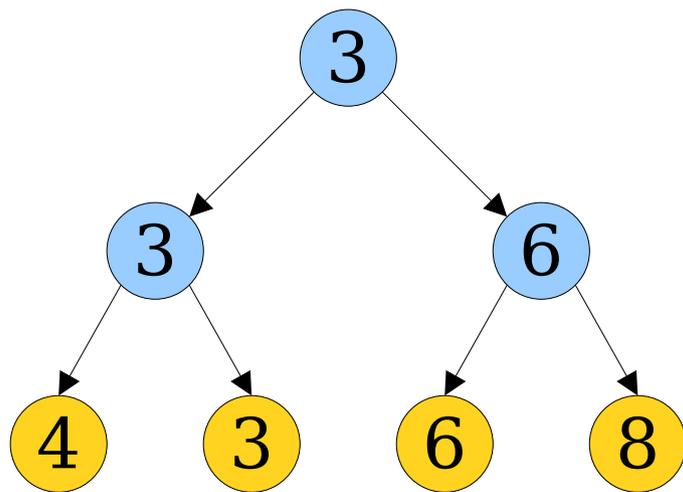


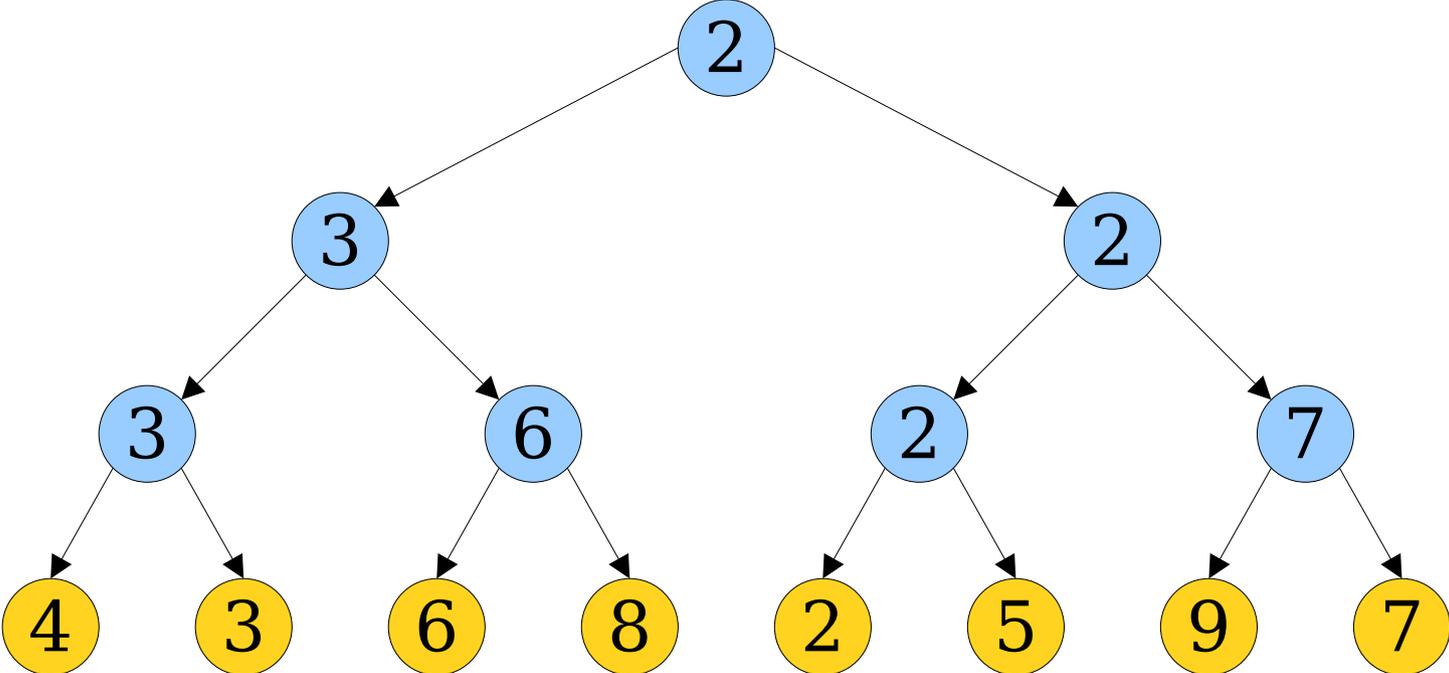


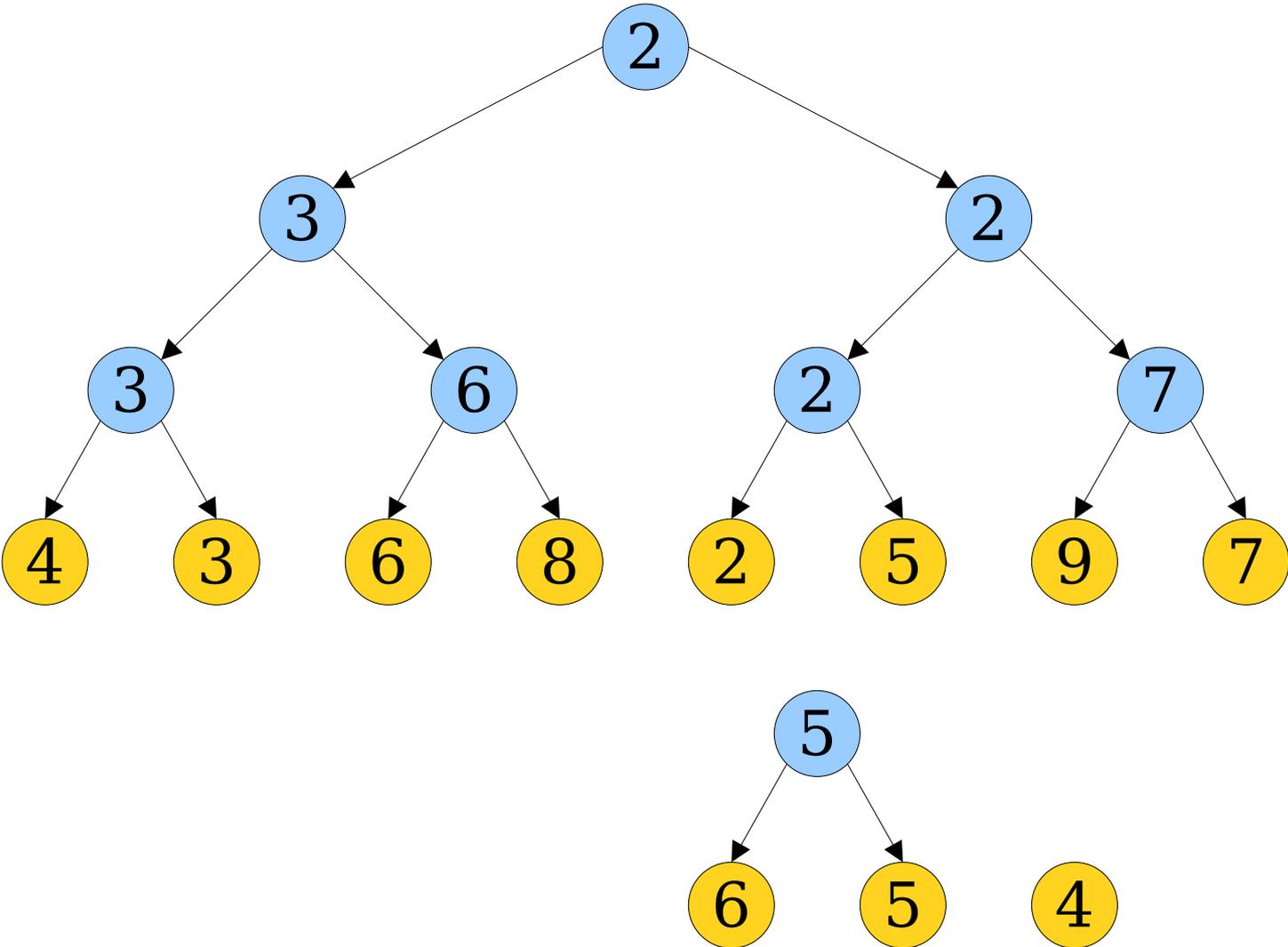


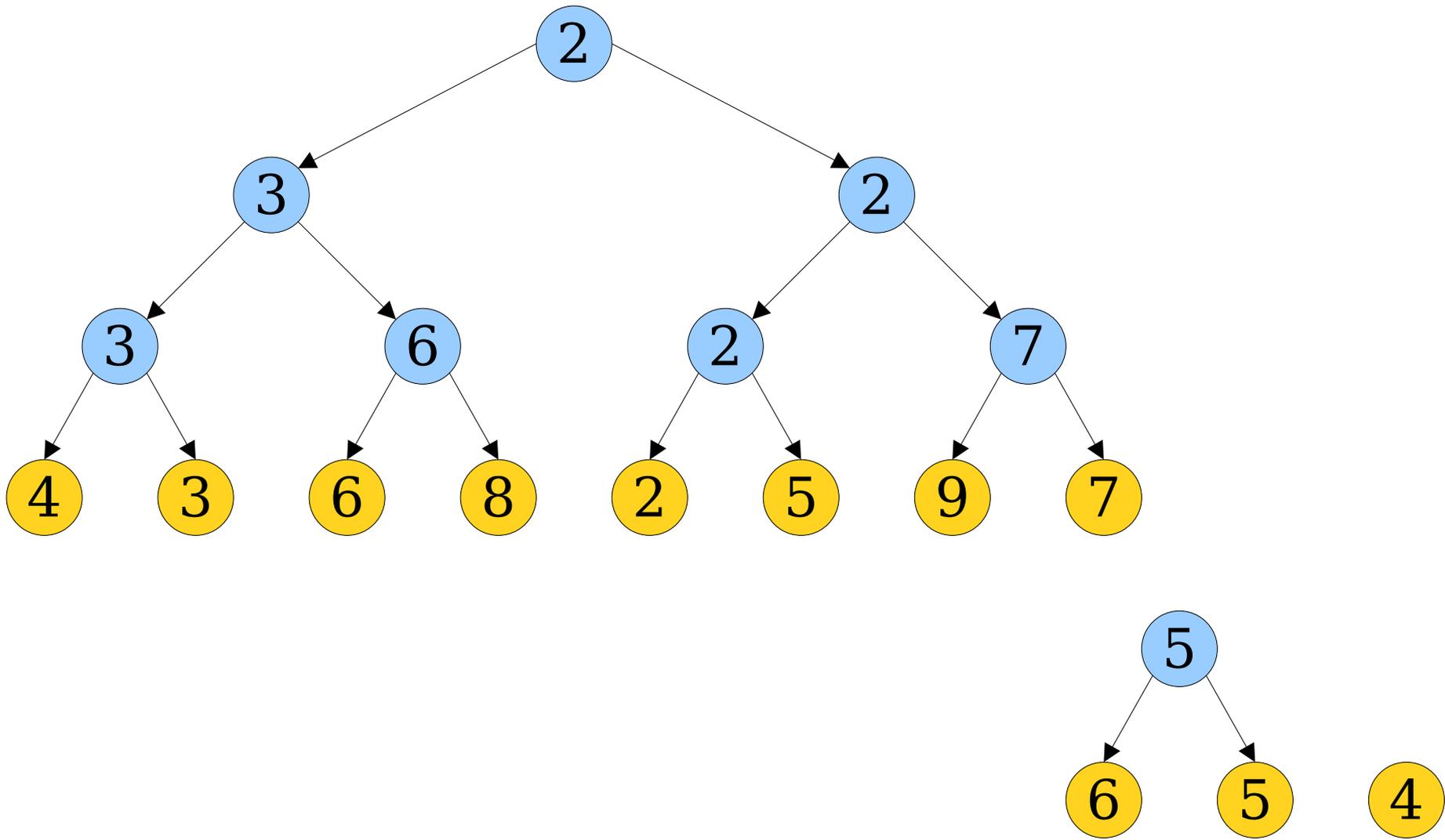


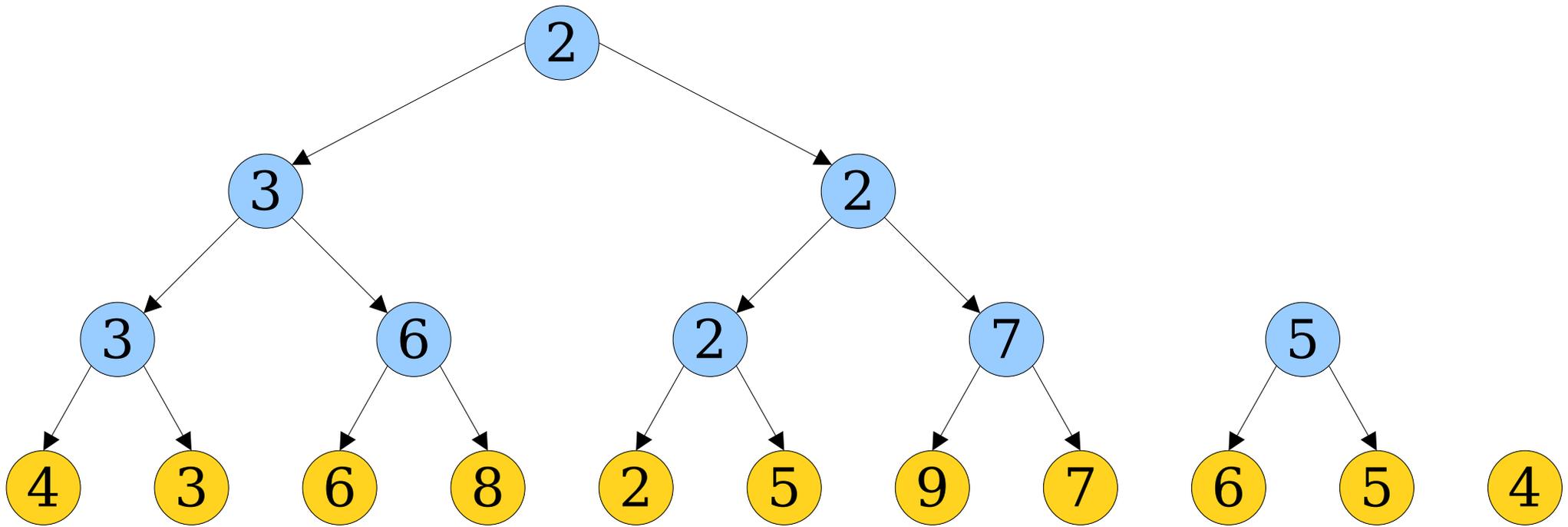


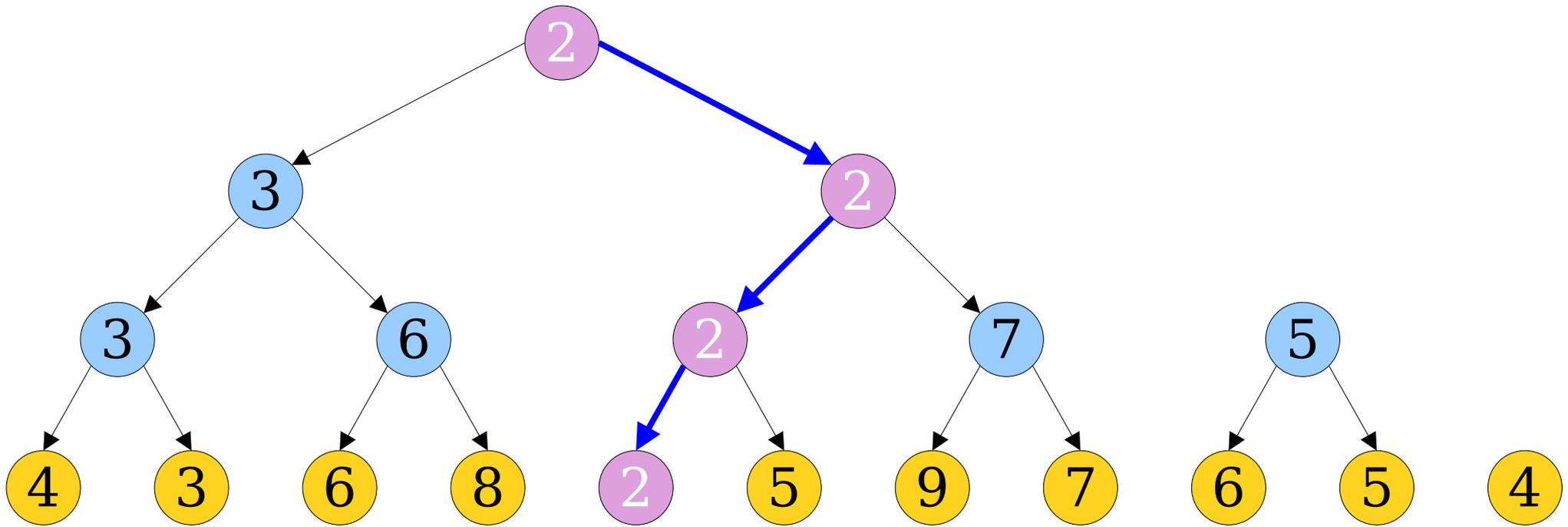


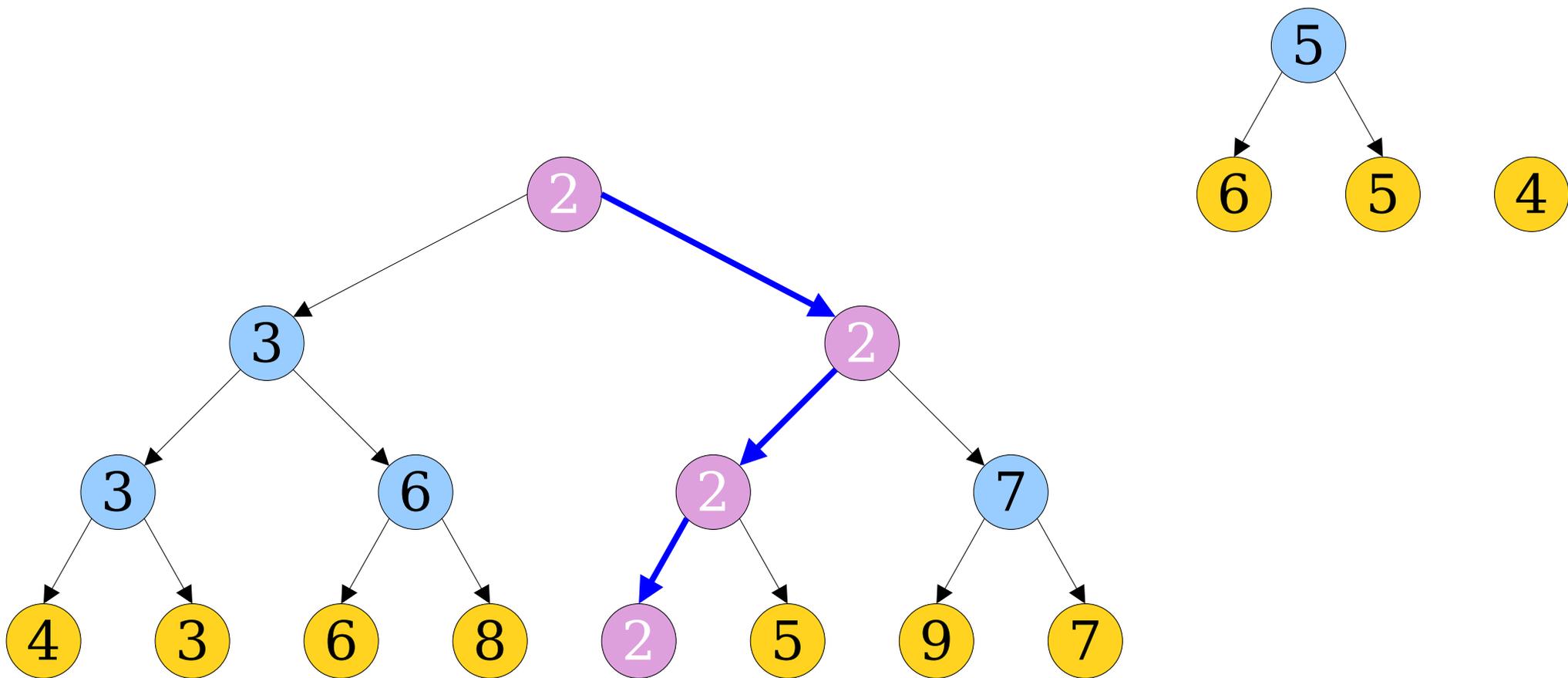


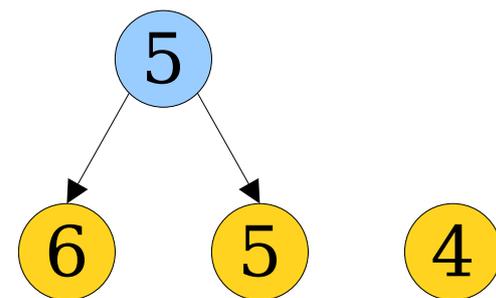
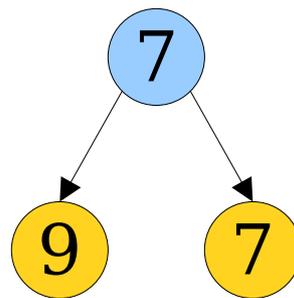
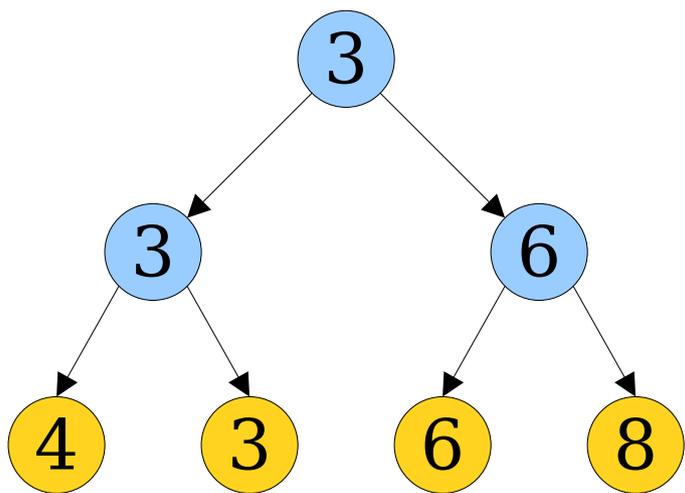


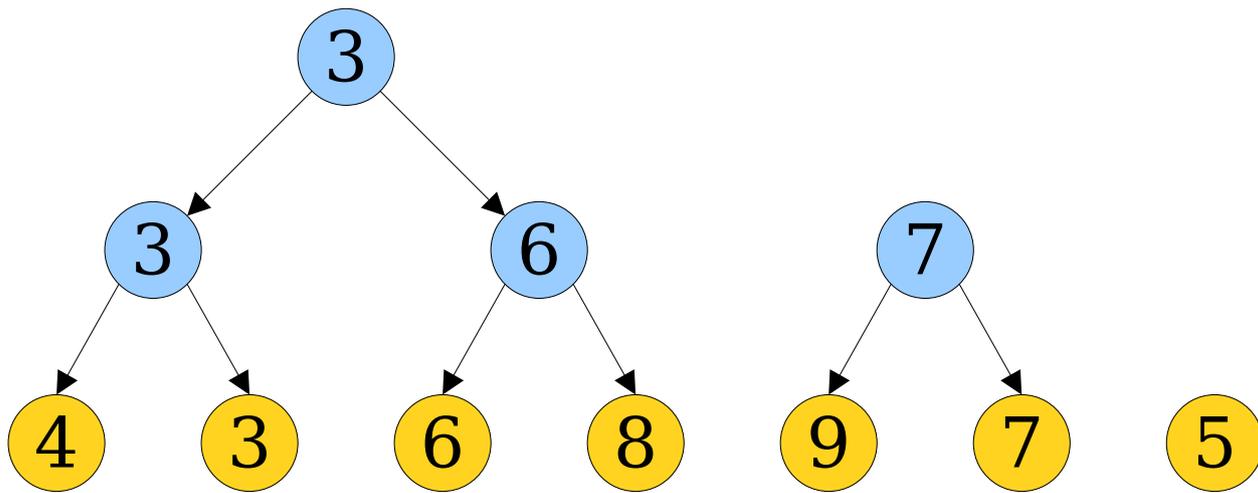
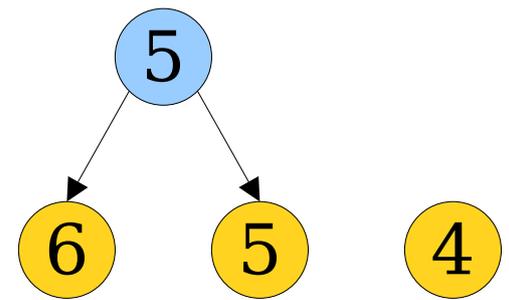


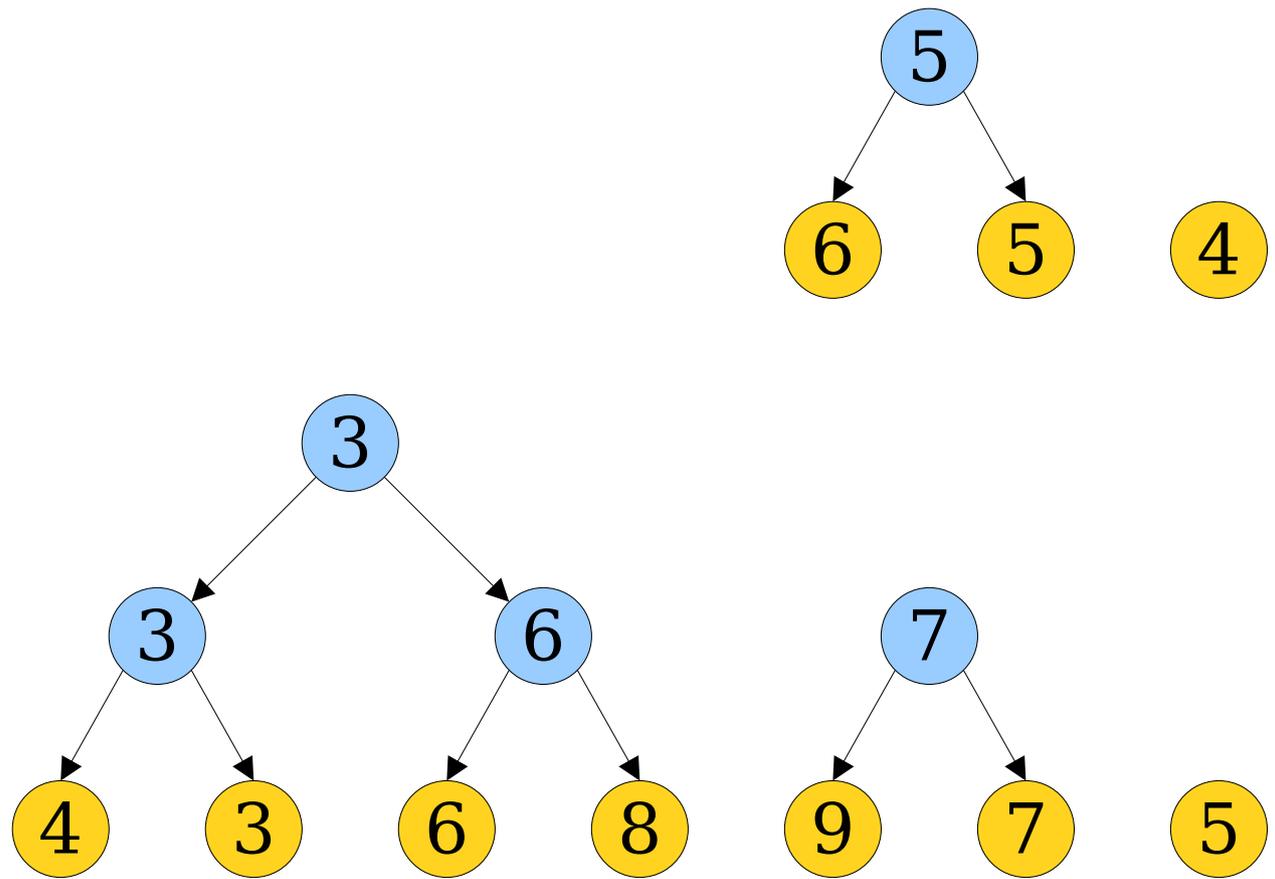


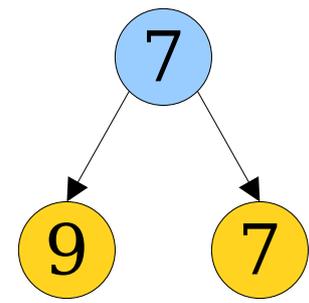
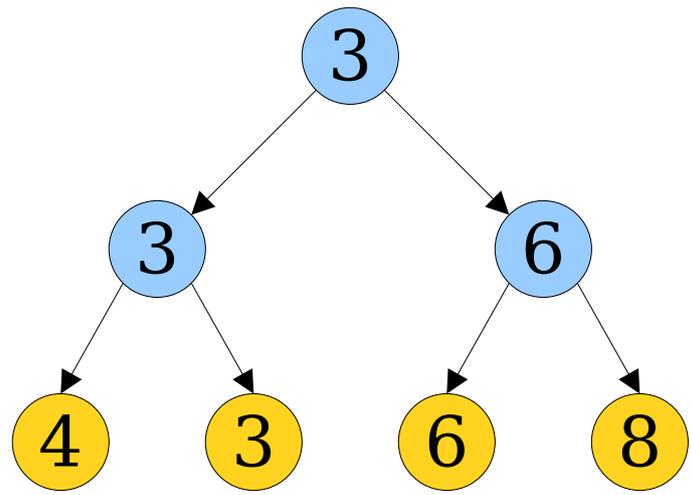
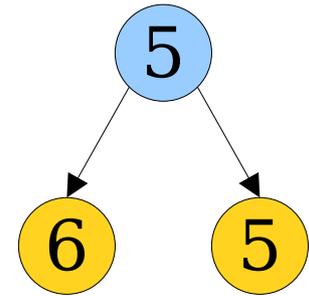
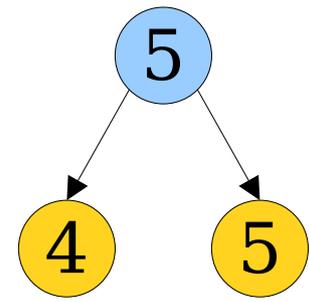


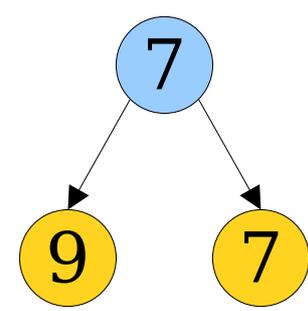
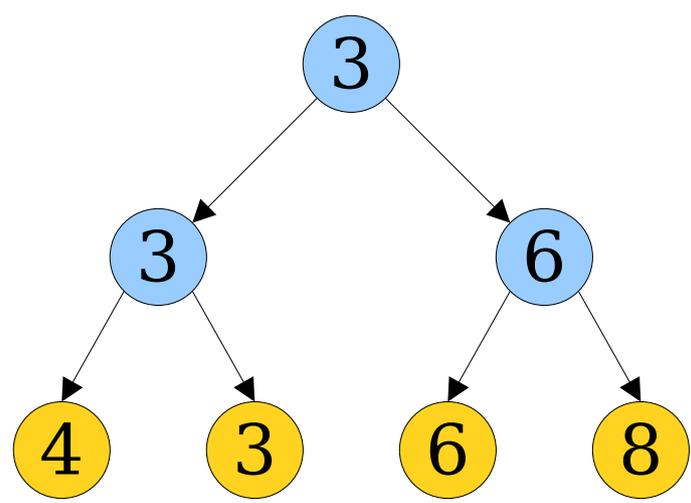
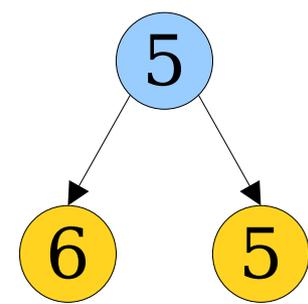
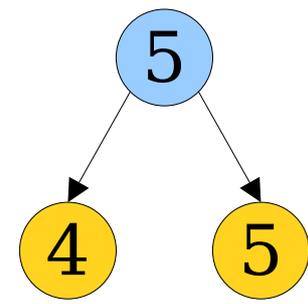


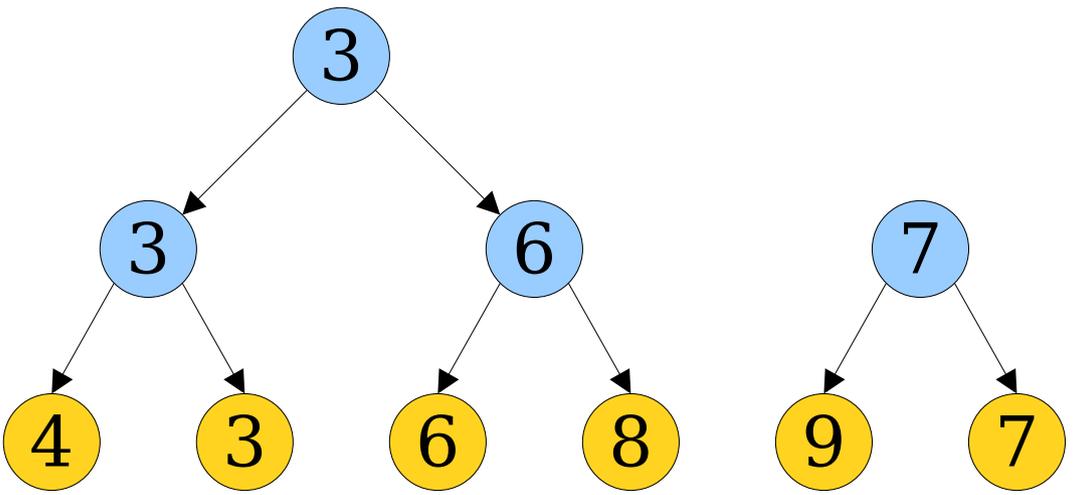
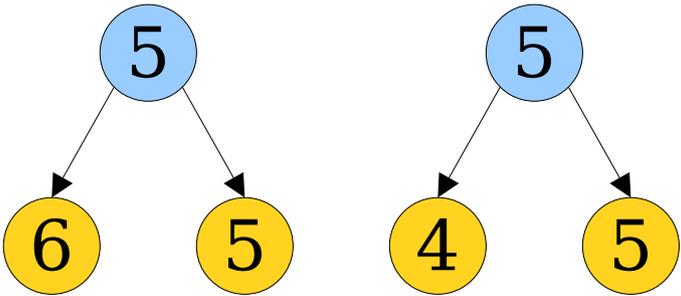


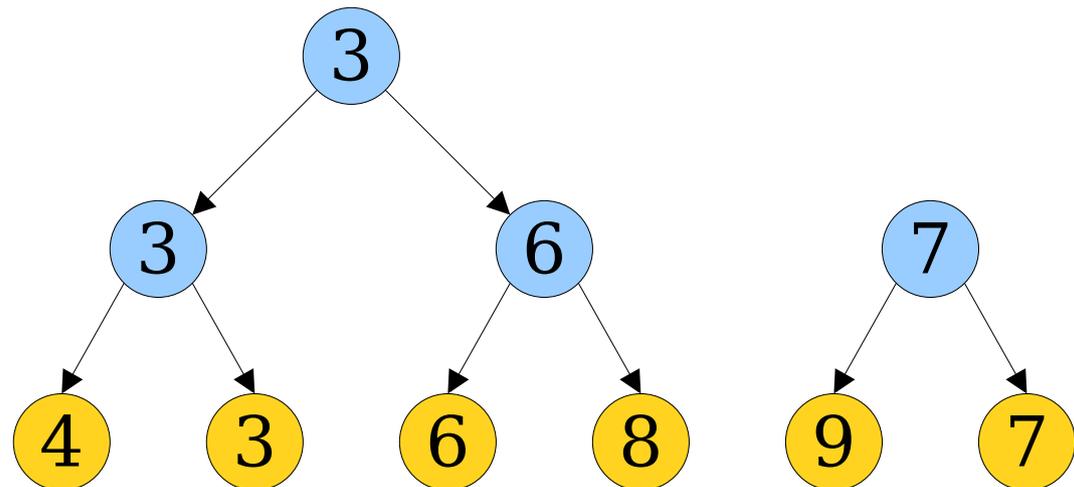
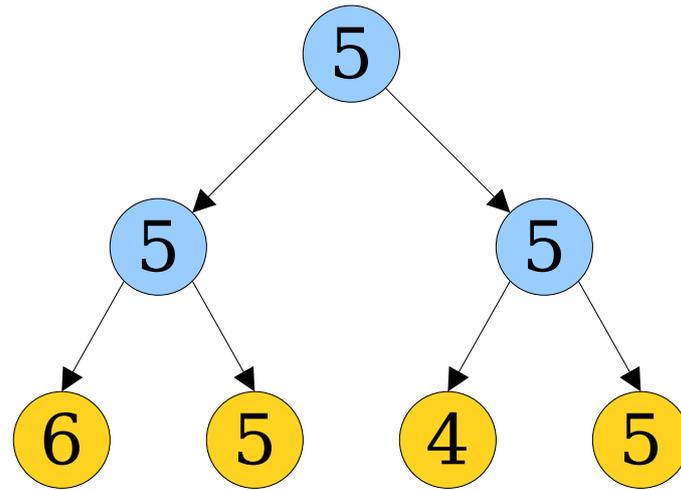


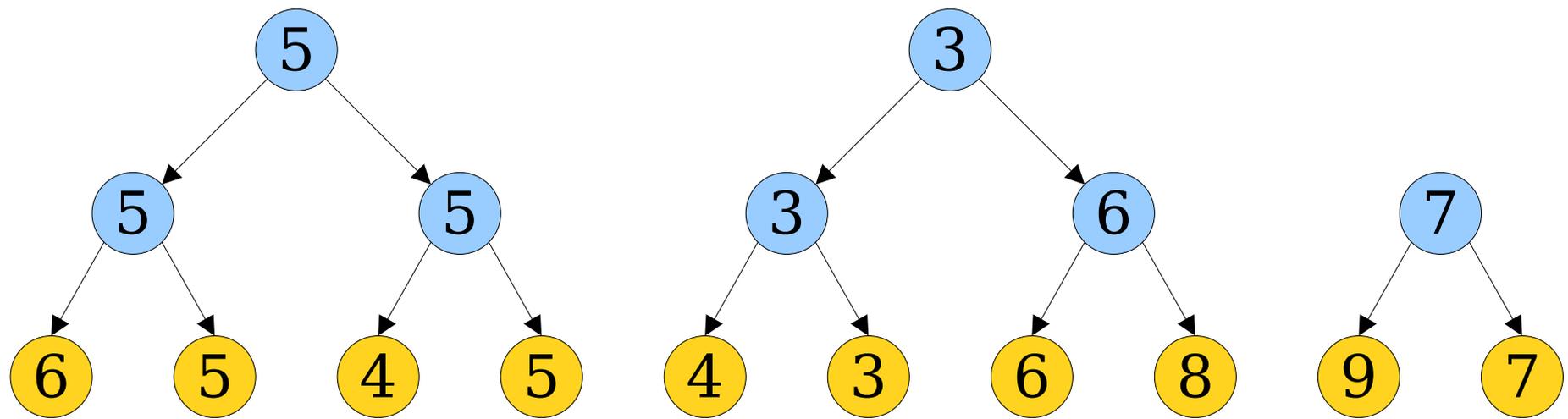


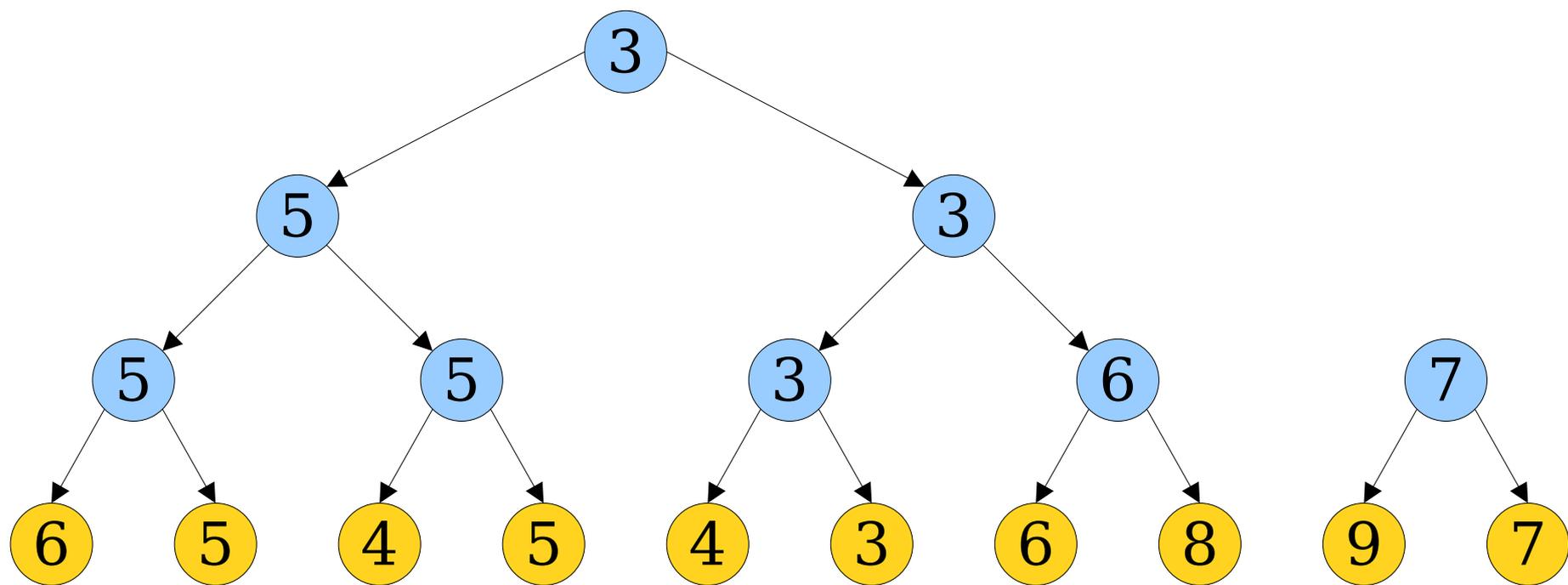












---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.

1

---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.

1

2

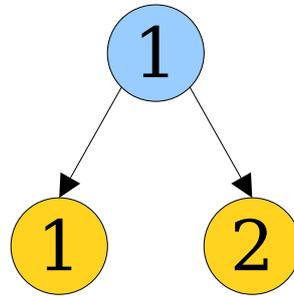
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



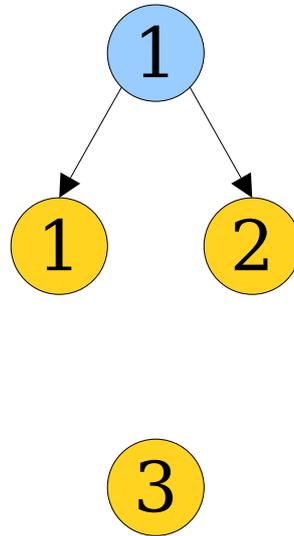
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



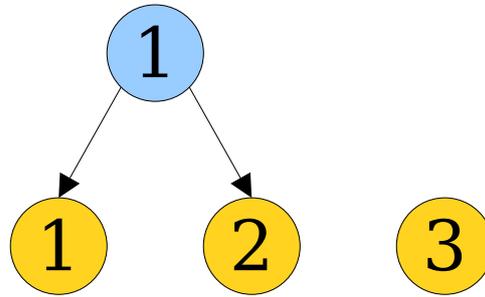
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



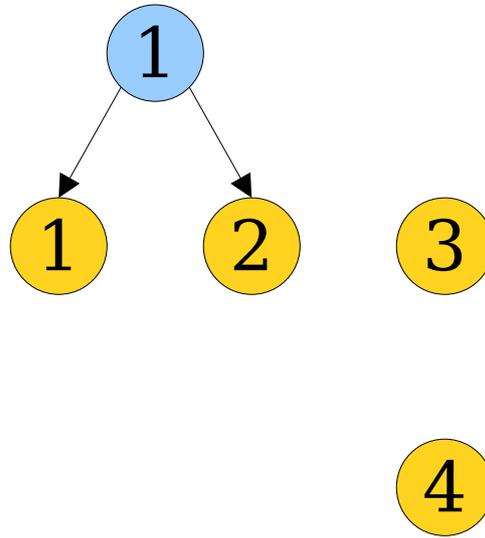
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



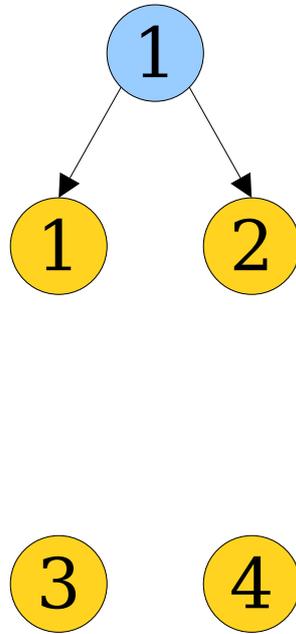
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



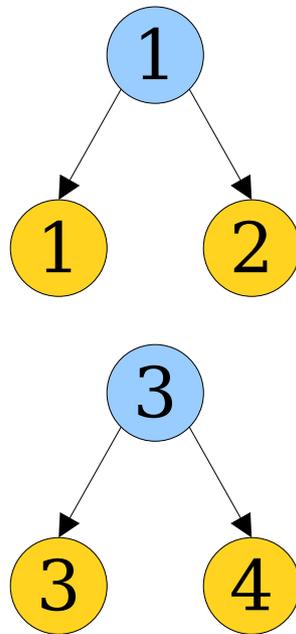
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



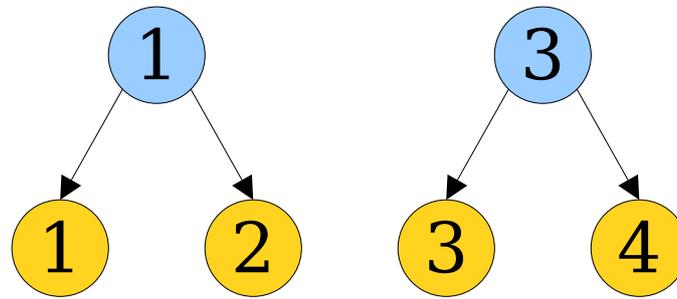
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



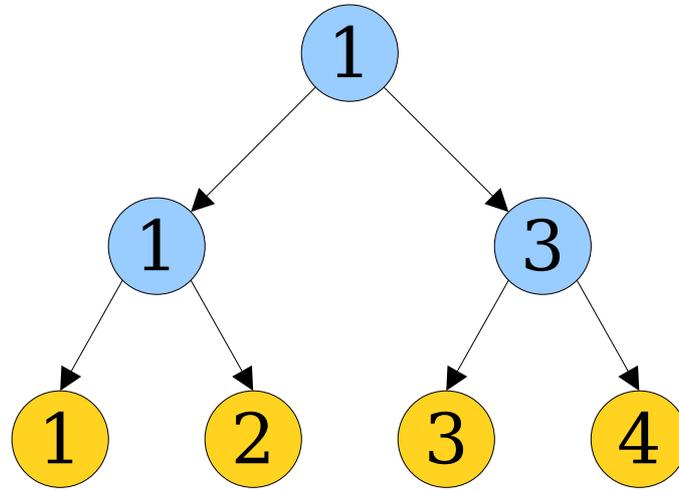
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



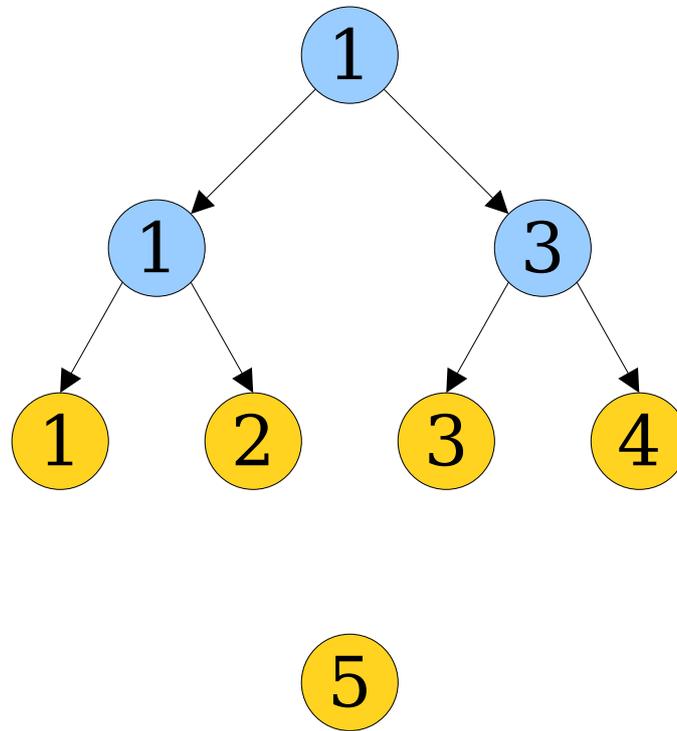
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



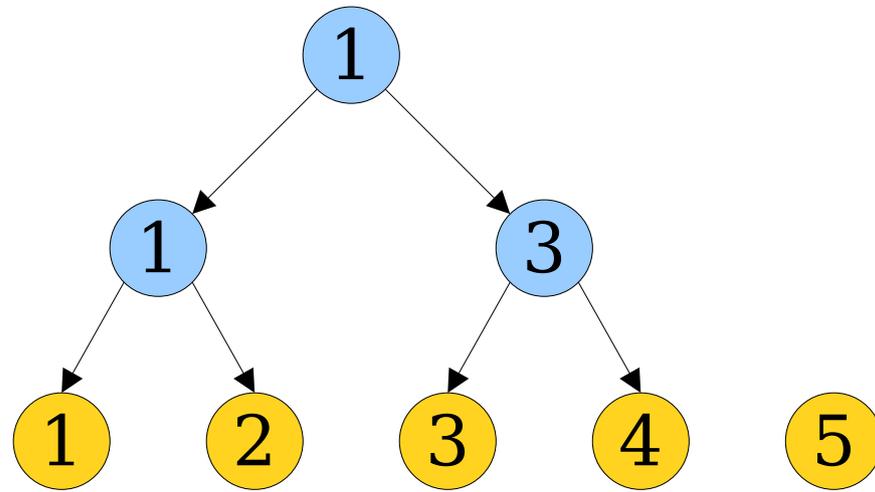
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



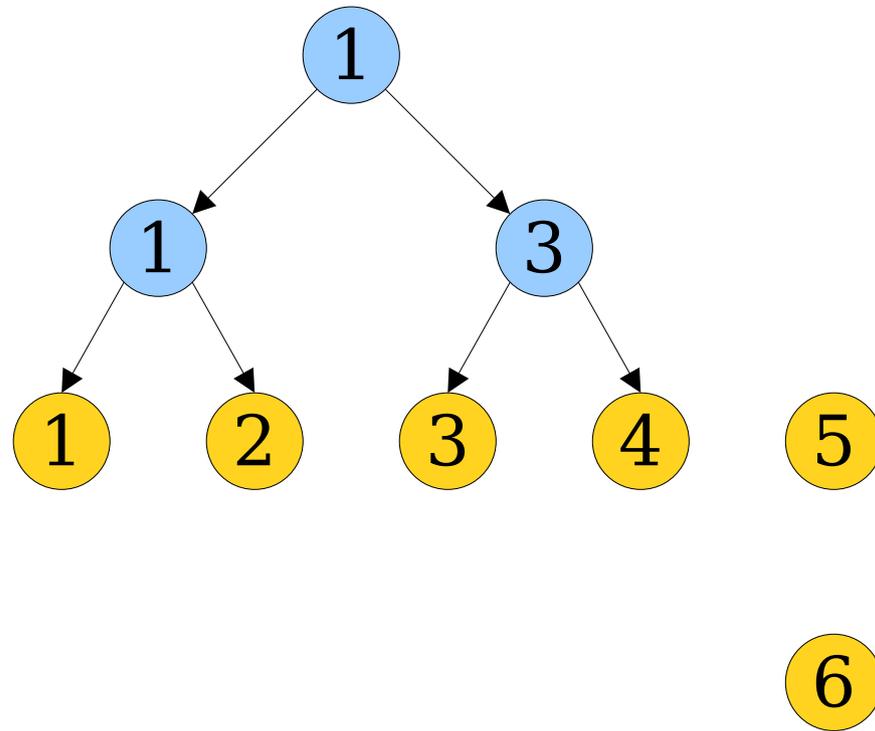
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



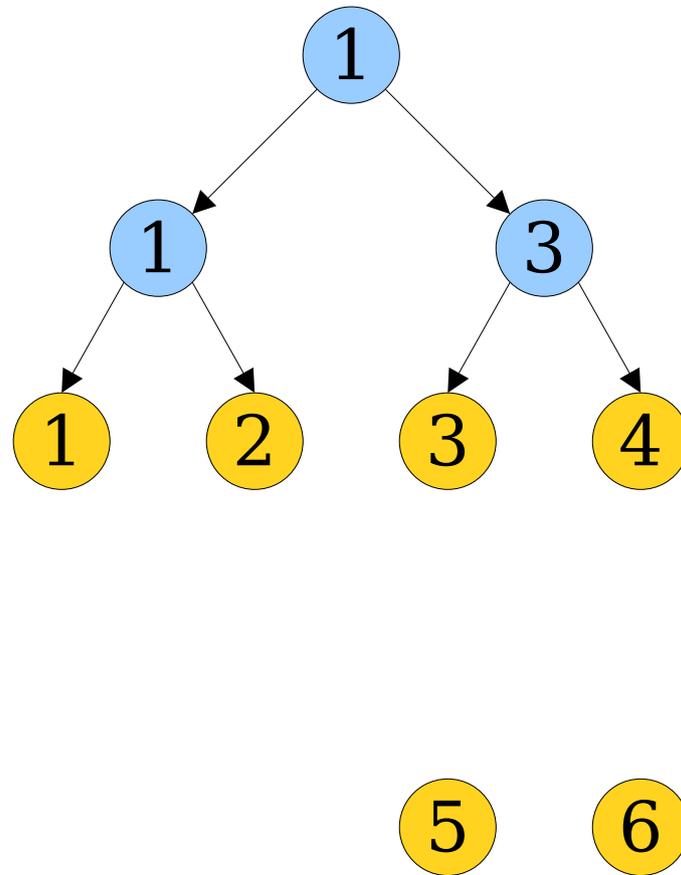
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



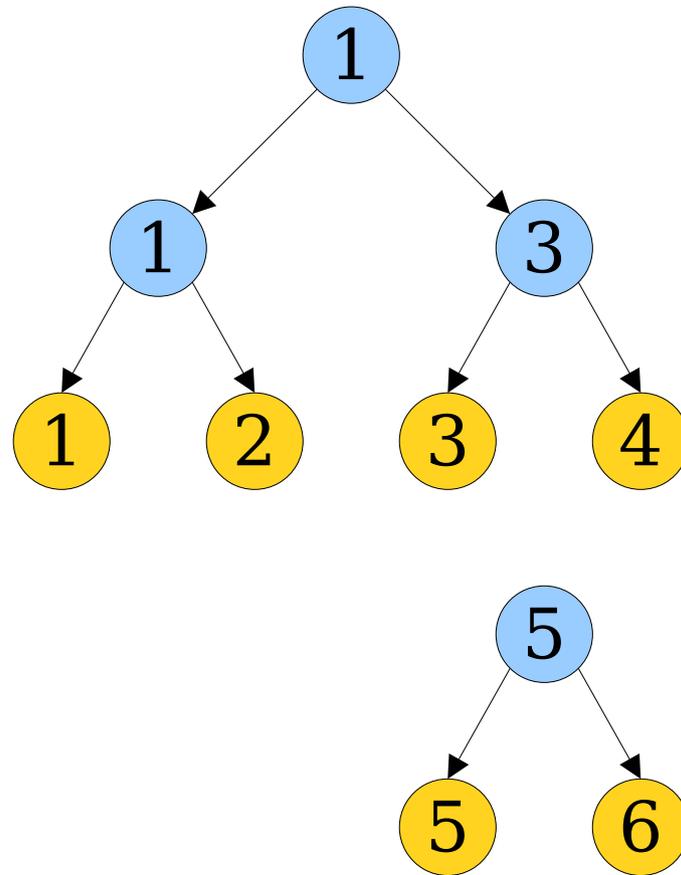
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



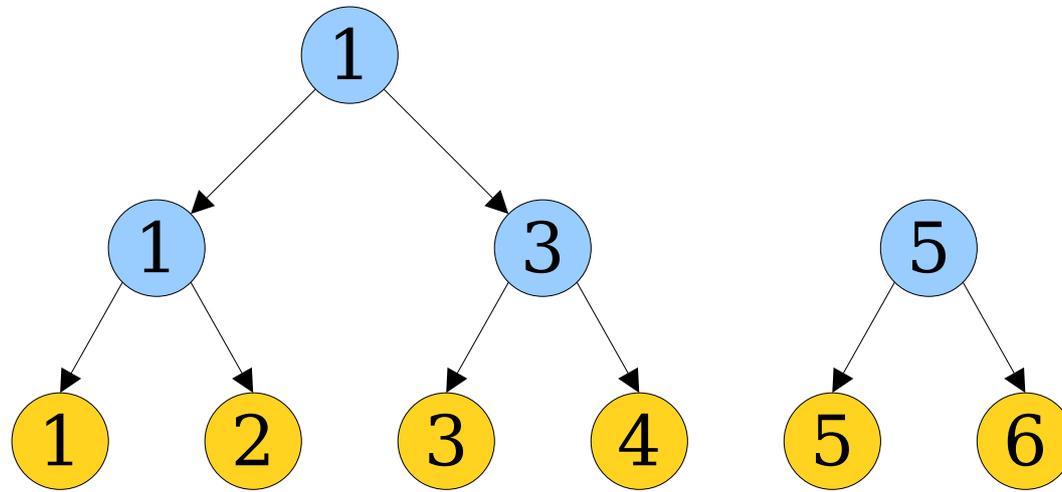
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



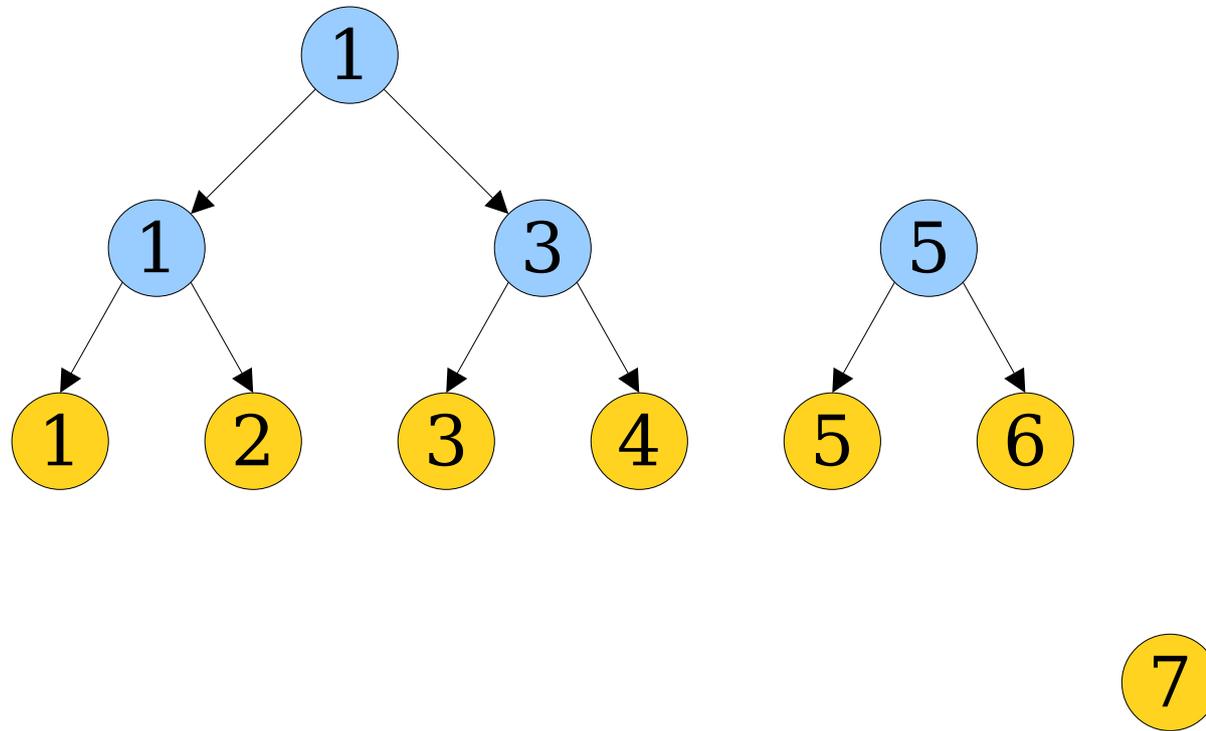
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



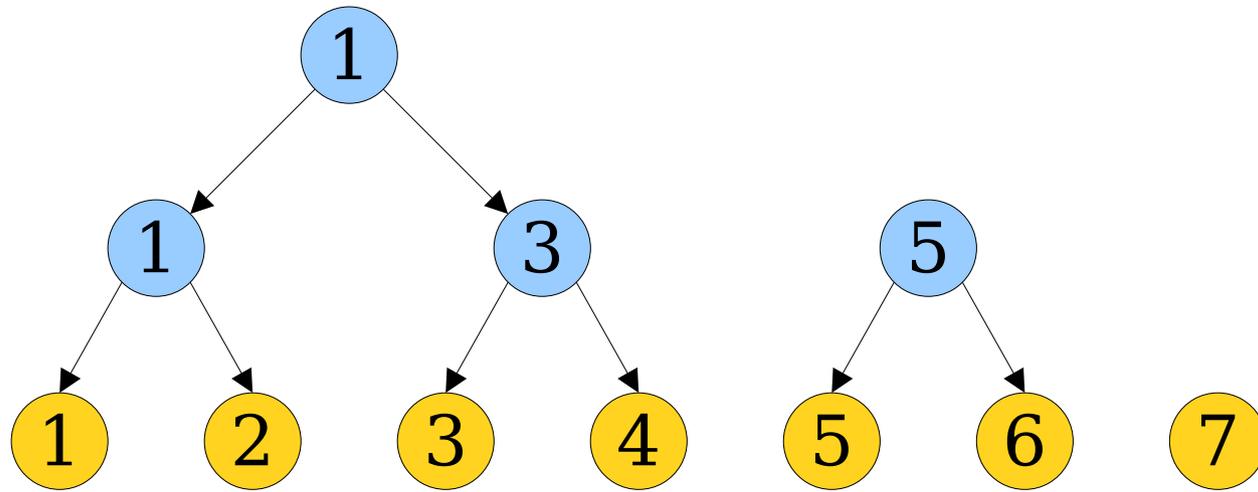
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



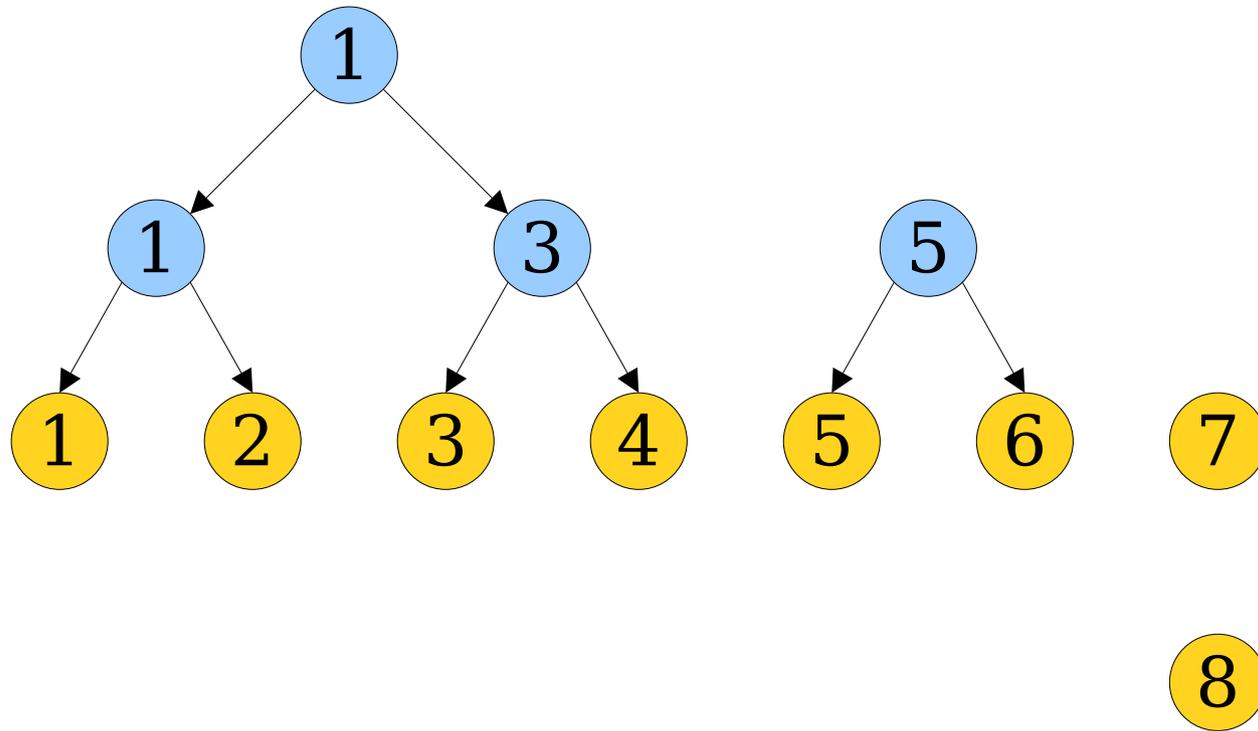
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



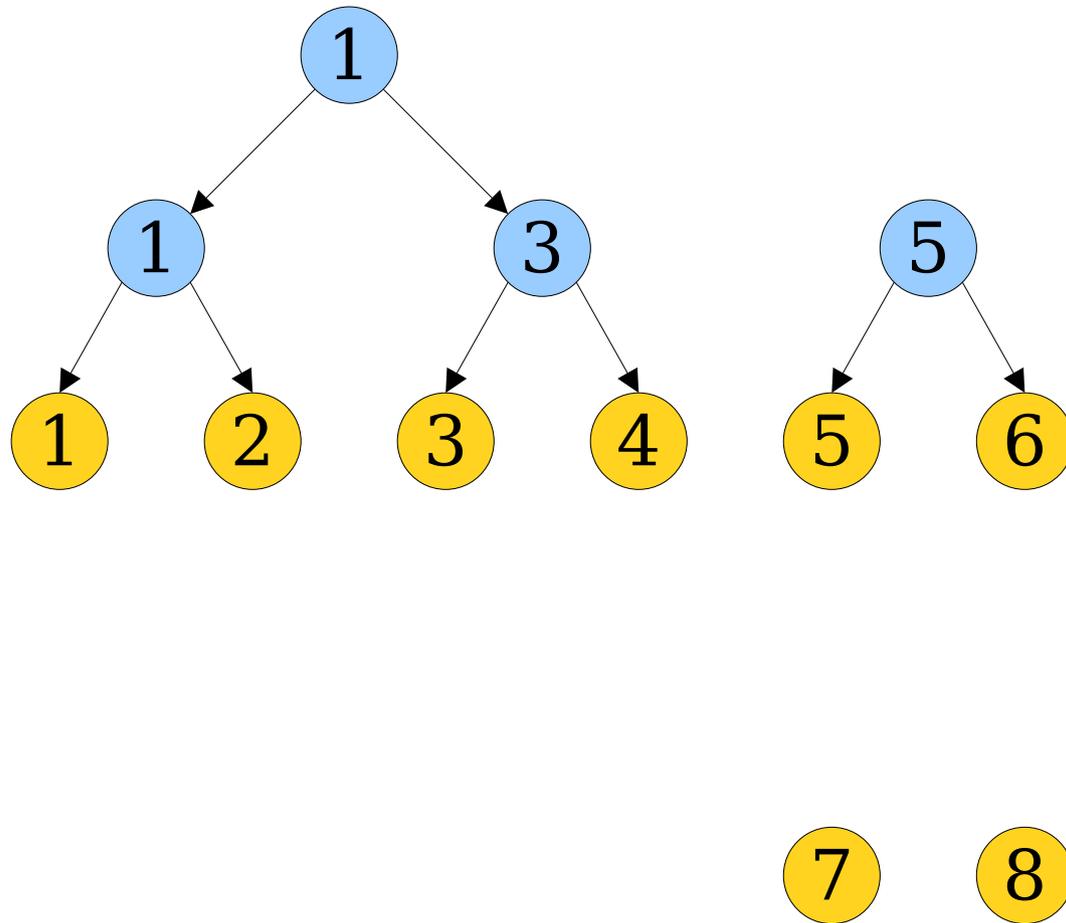
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



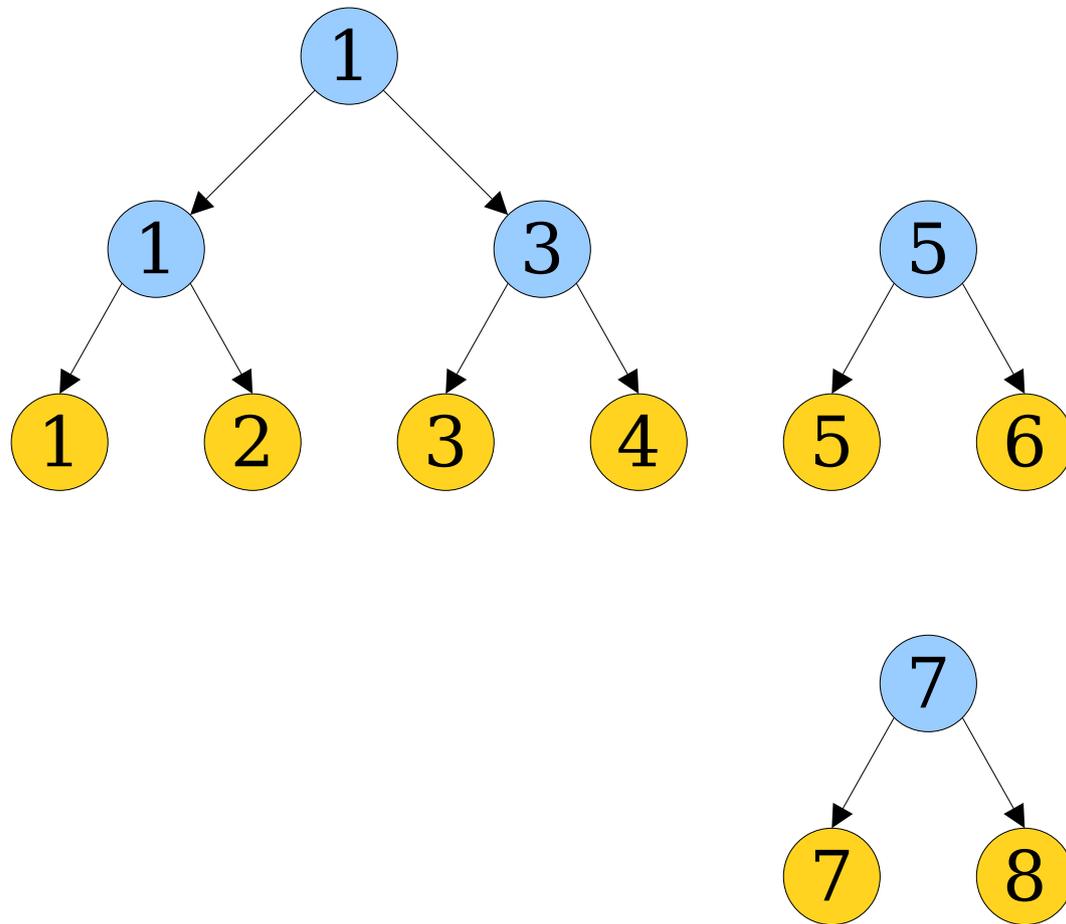
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



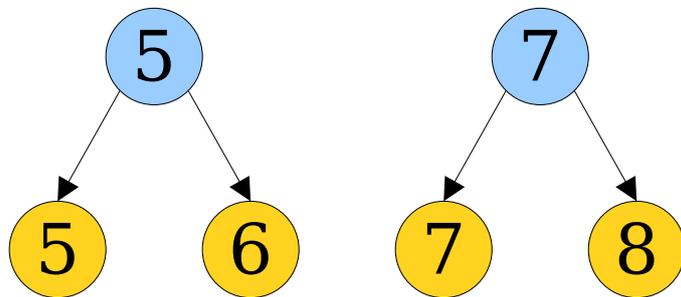
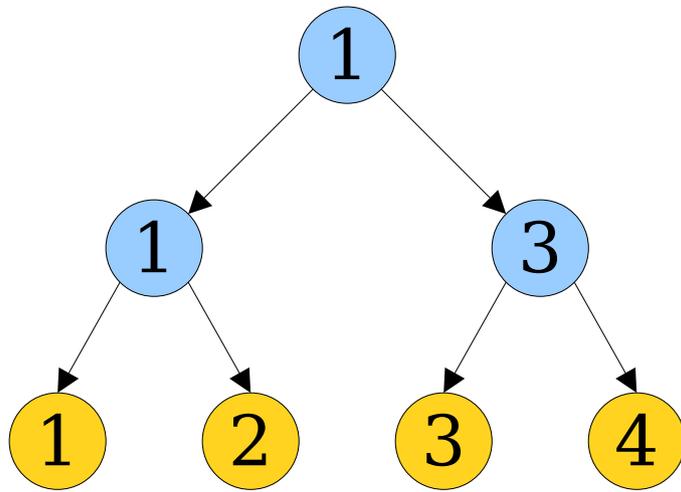
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



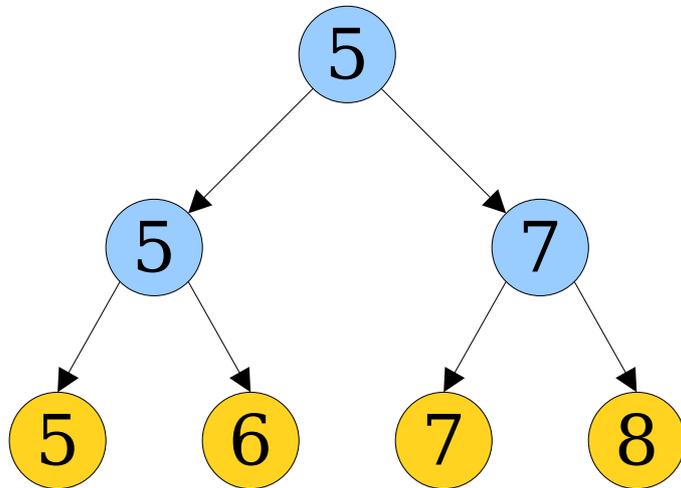
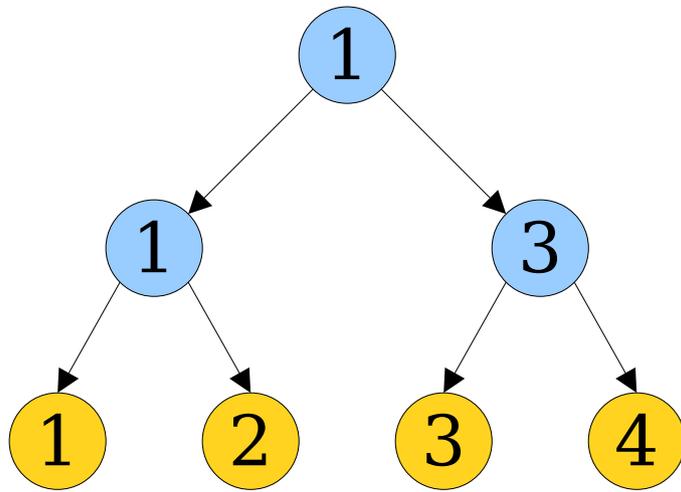
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



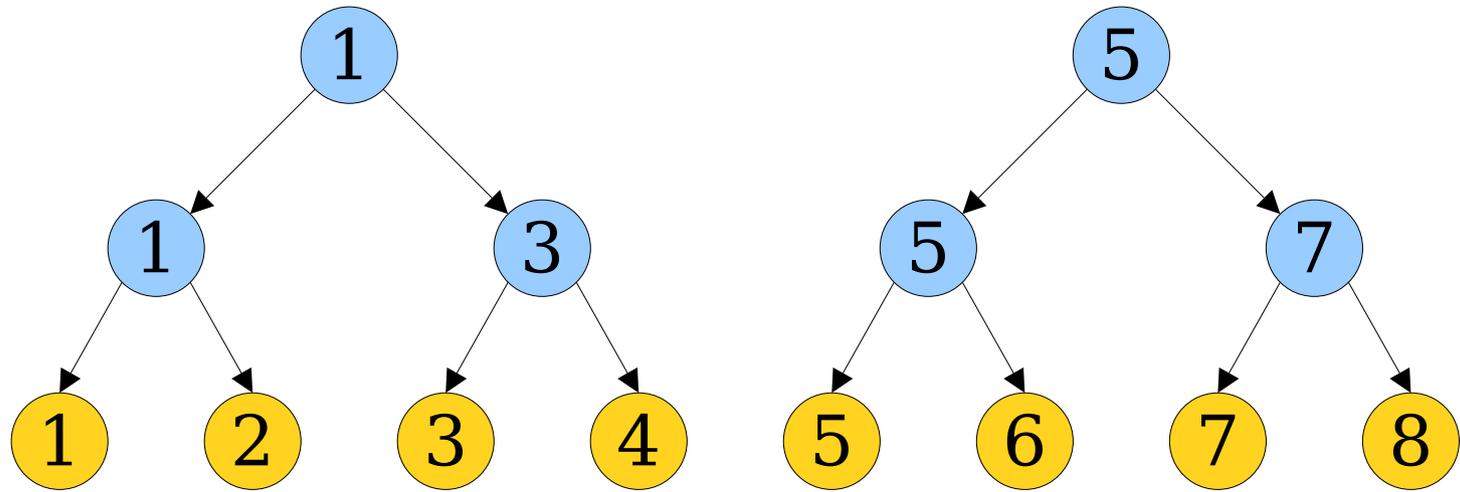
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



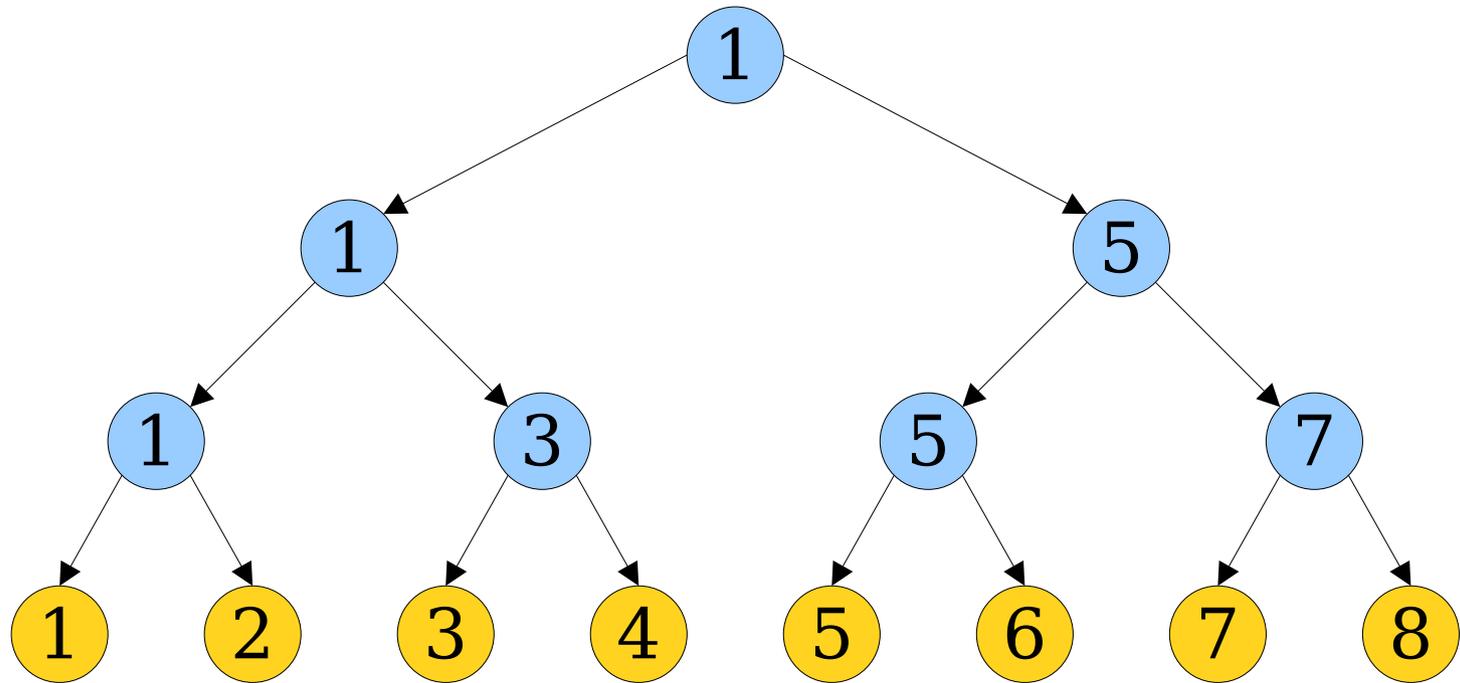
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



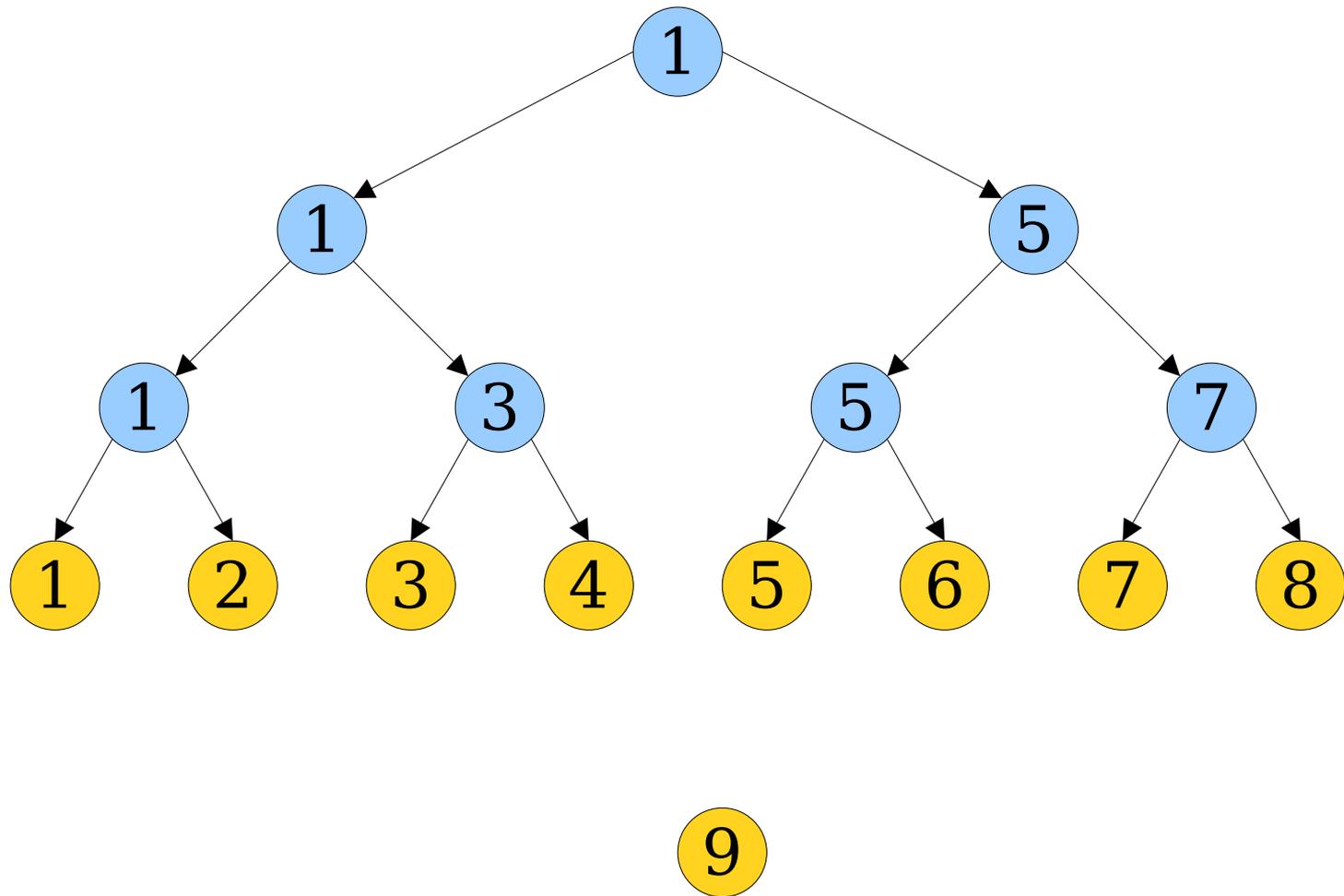
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



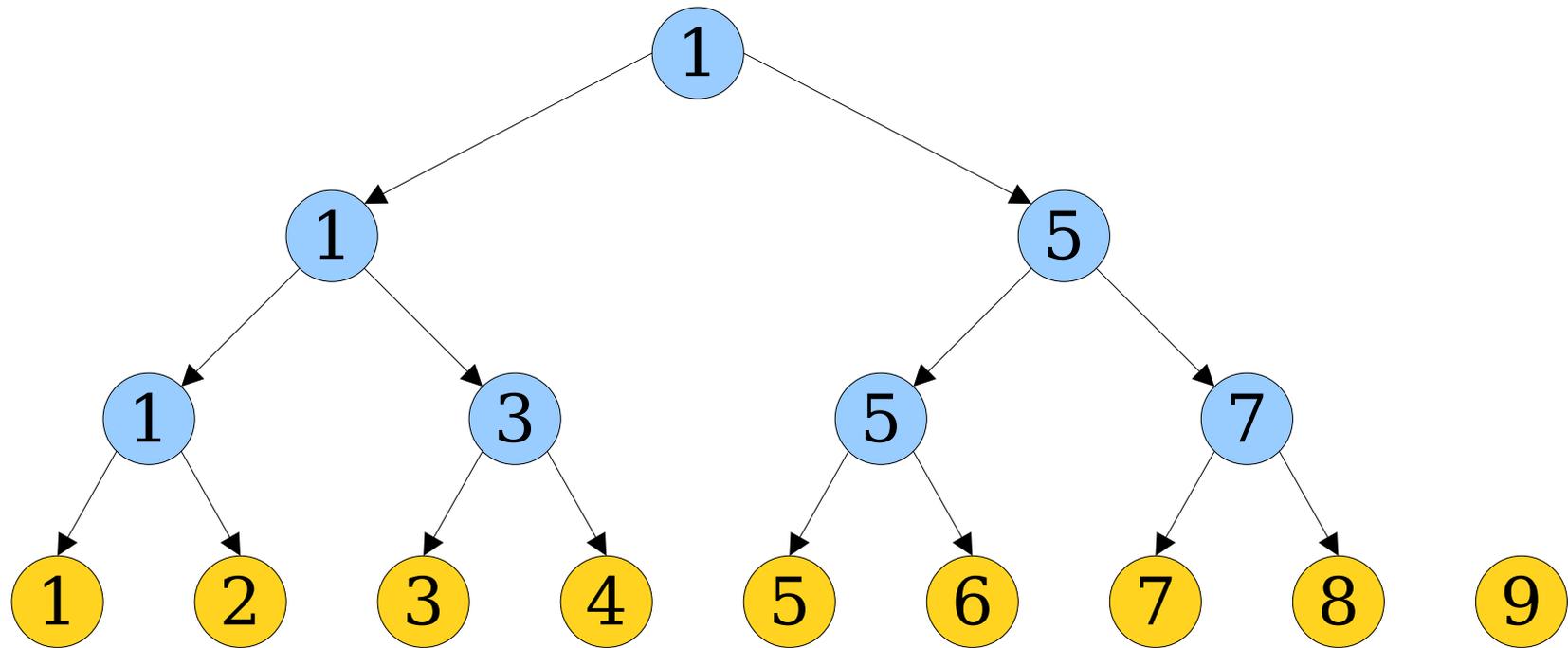
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



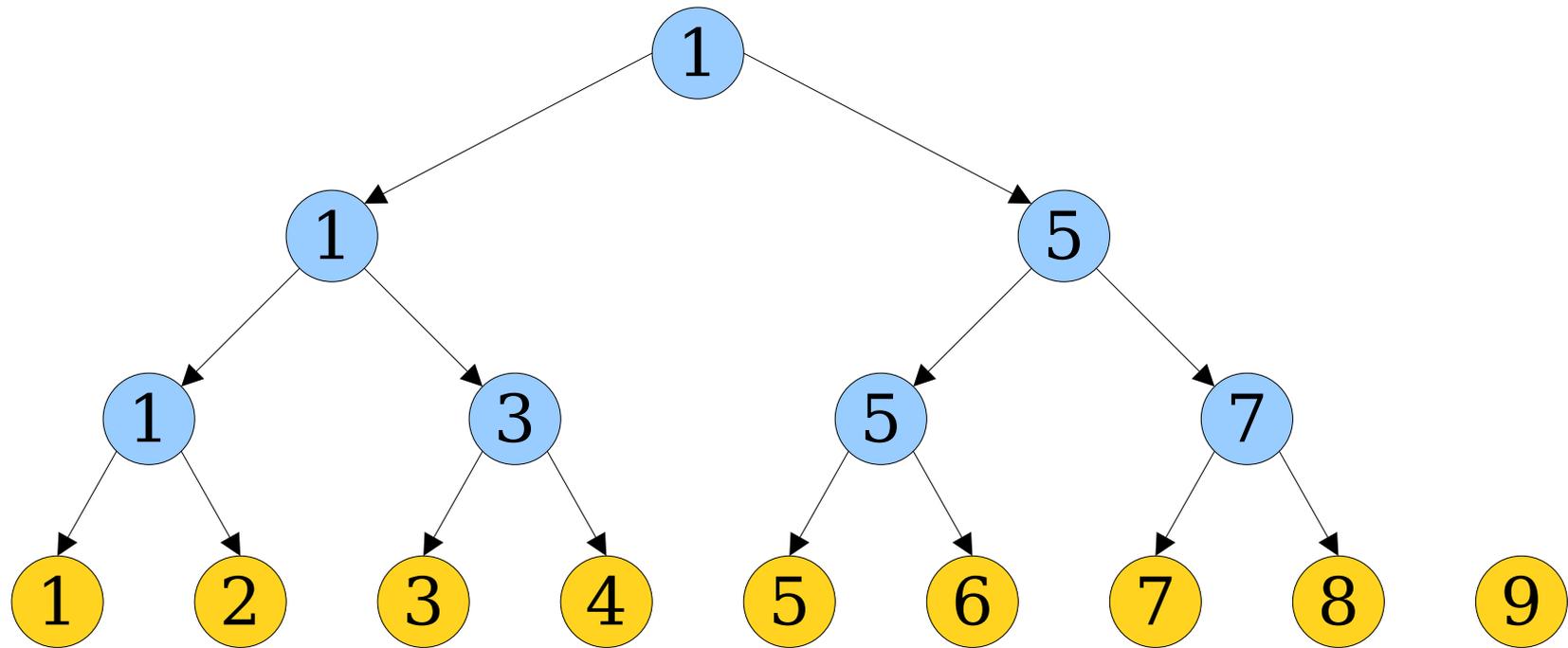
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



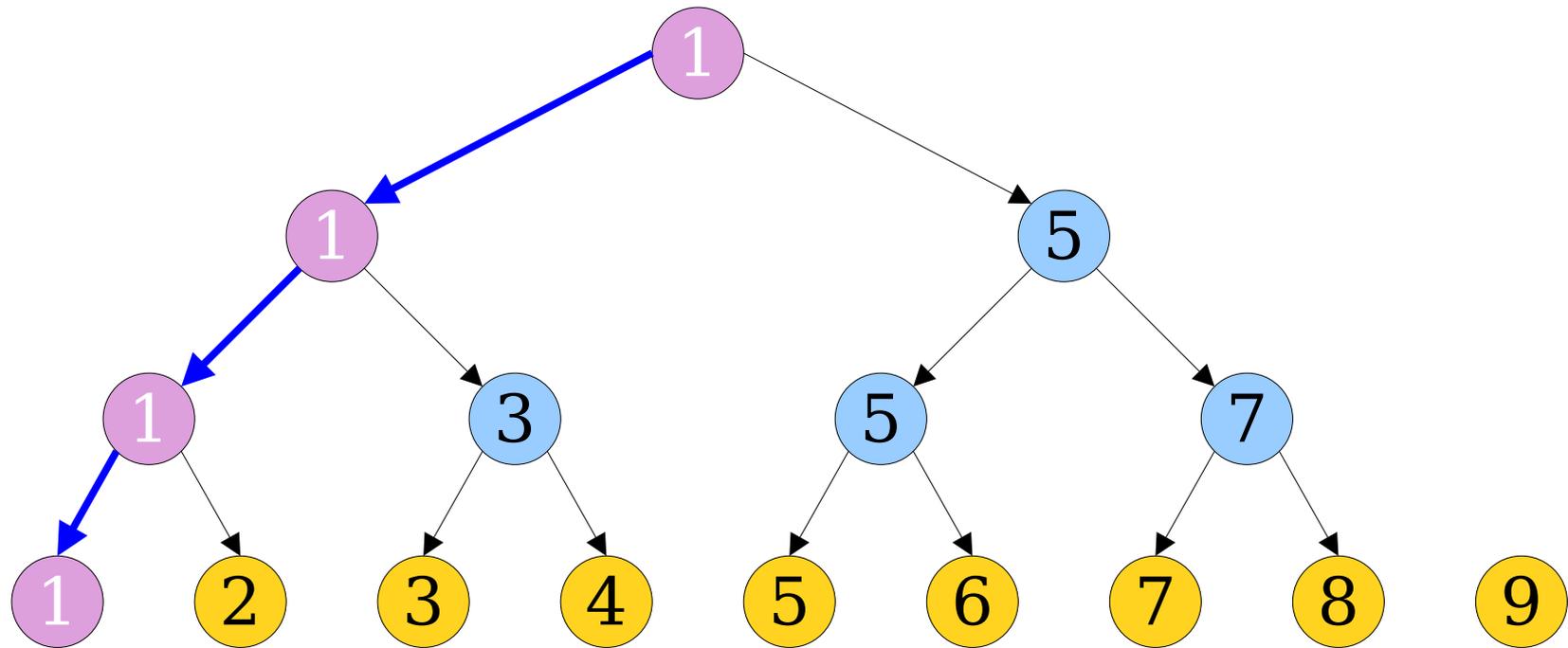
---

Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a tournament heap.



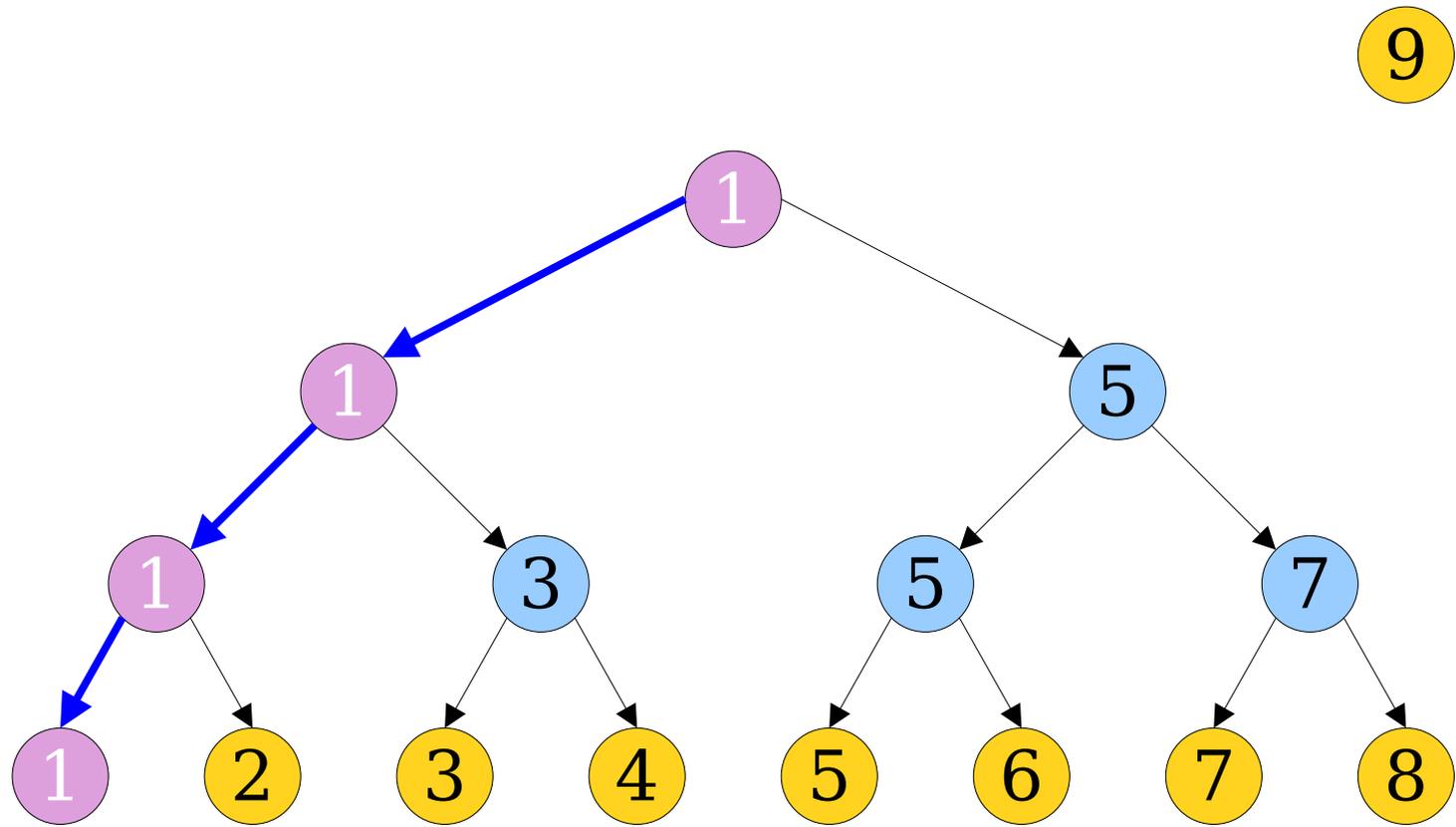
---

Draw what happens if we perform an *extract-min* on this tournament heap.

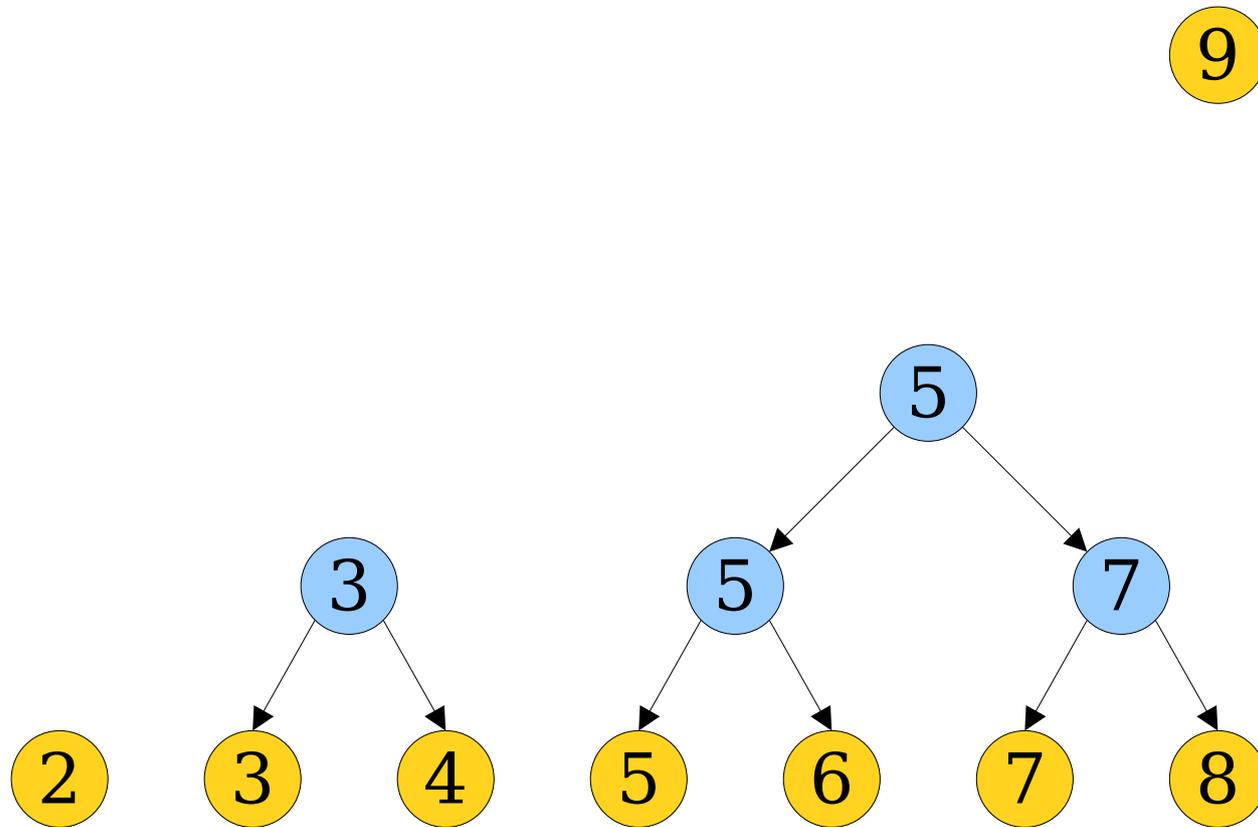


---

Draw what happens if we perform an *extract-min* on this tournament heap.

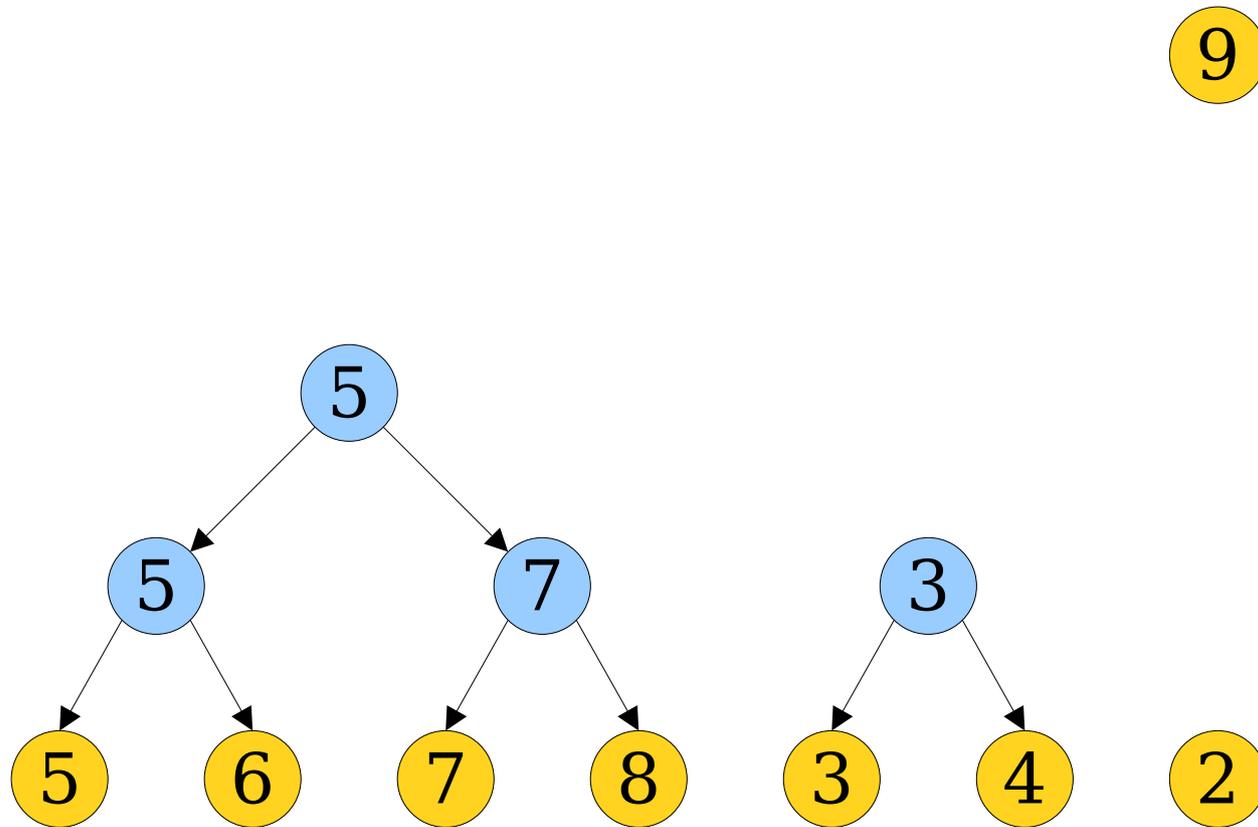


Draw what happens if we perform an *extract-min* on this tournament heap.



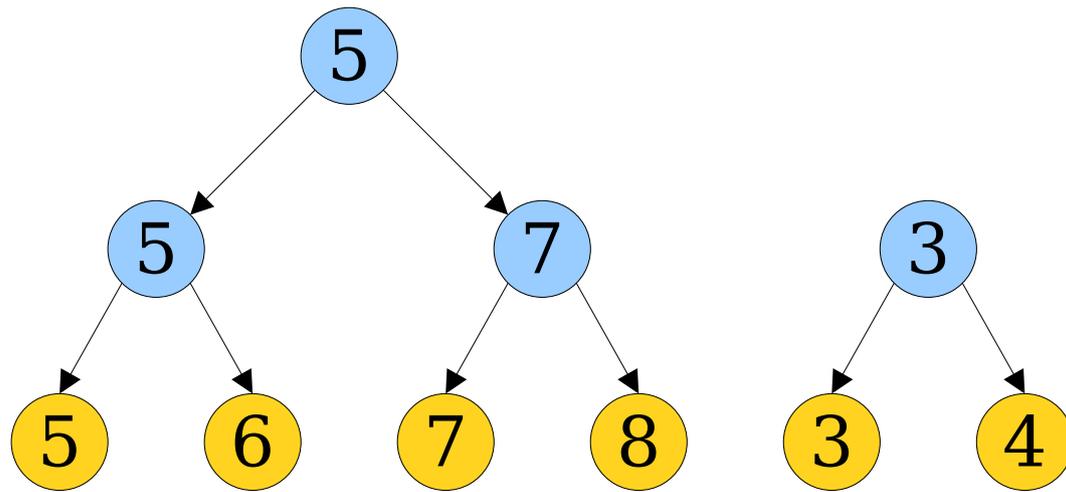
---

Draw what happens if we perform an *extract-min* on this tournament heap.



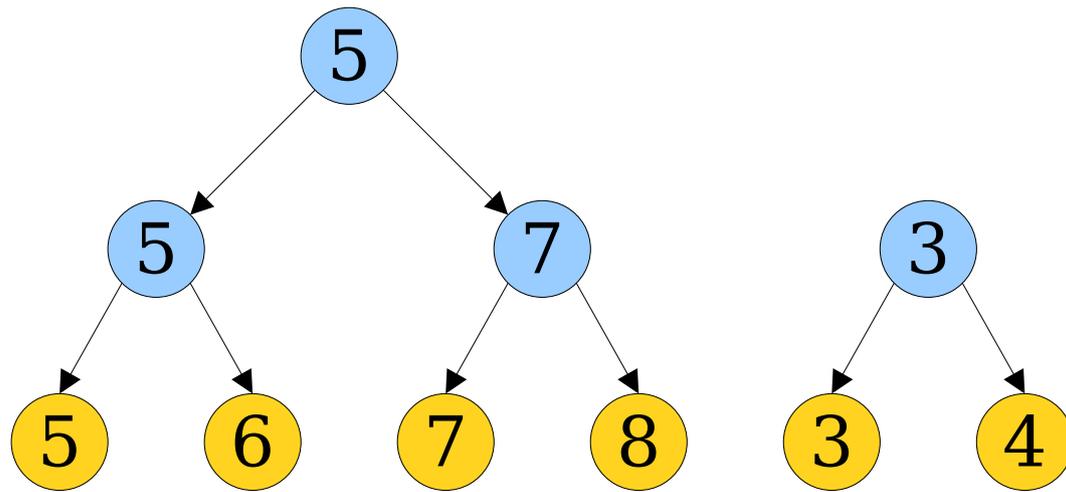
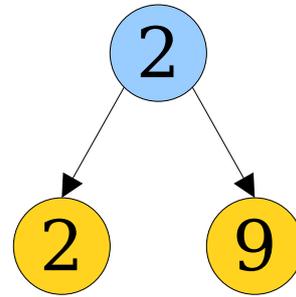
---

Draw what happens if we perform an *extract-min* on this tournament heap.



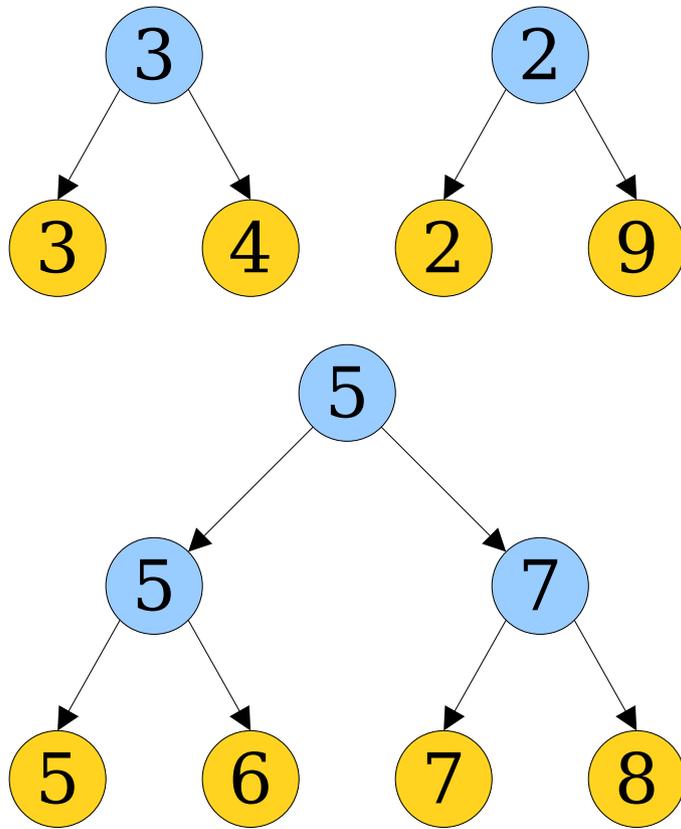
---

Draw what happens if we perform an *extract-min* on this tournament heap.



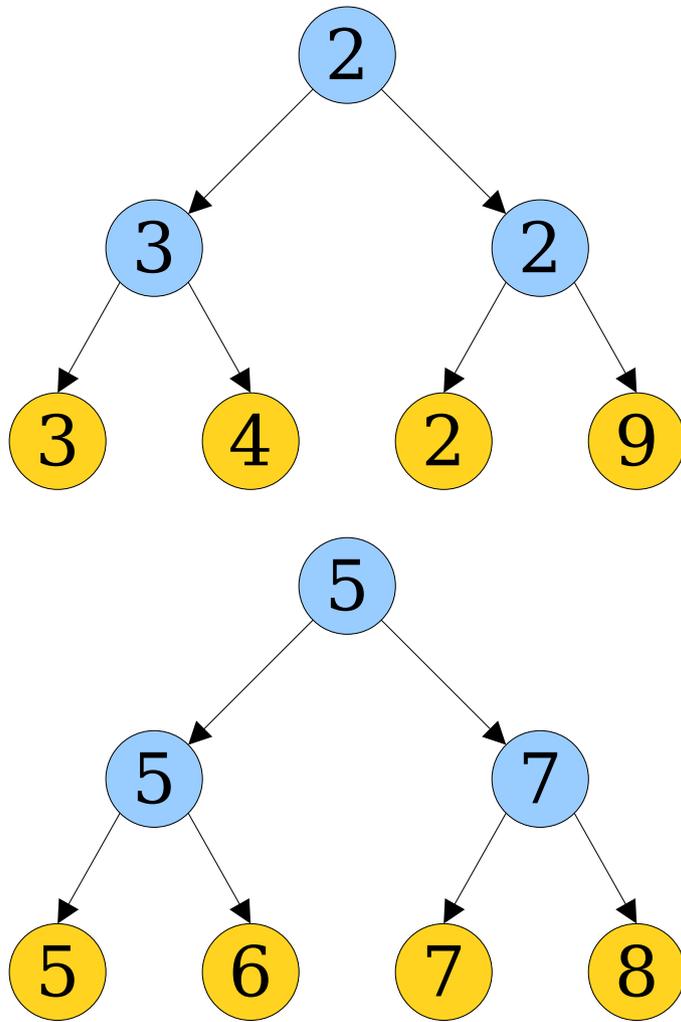
---

Draw what happens if we perform an *extract-min* on this tournament heap.



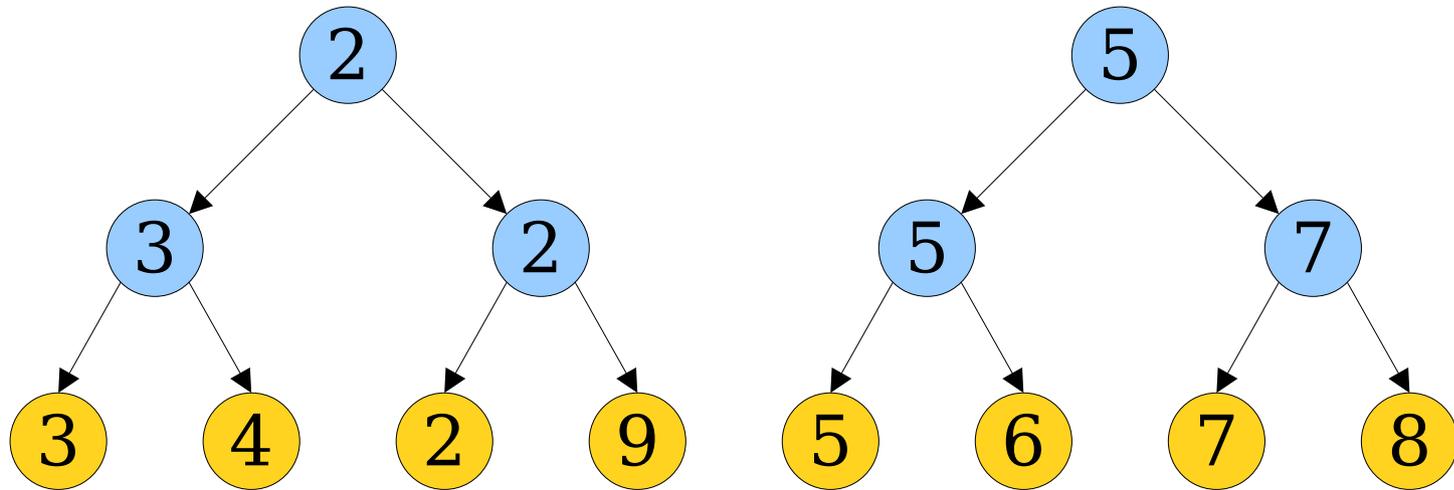
---

Draw what happens if we perform an *extract-min* on this tournament heap.



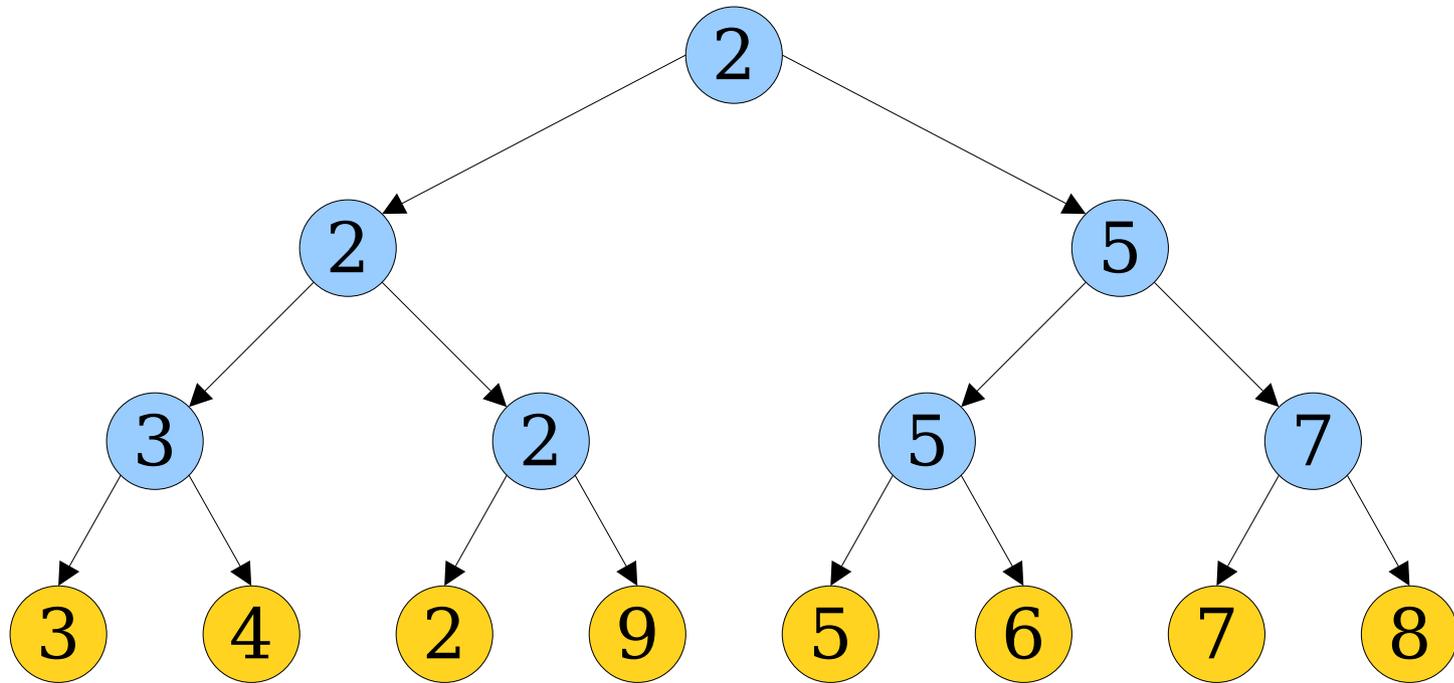
---

Draw what happens if we perform an *extract-min* on this tournament heap.



---

Draw what happens if we perform an *extract-min* on this tournament heap.



---

Draw what happens if we perform an *extract-min* on this tournament heap.

# Where We Stand

- Here's the current scorecard for the tournament heap.
- This is a fast, elegant, and clever data structure.
- **Question:** Can we do better?

## Tournament Heap

- **enqueue**:  $O(\log n)$
- **find-min**:  $O(\log n)$
- **extract-min**:  $O(\log n)$
- **meld**:  $O(\log m + \log n)$ .

# Where We Stand

- **Theorem:** No comparison-based priority queue structure can have *enqueue* and *extract-min* each take time  $o(\log n)$ .
- **Proof:** Suppose these operations each take time  $o(\log n)$ . Then we could sort  $n$  elements by perform  $n$  *enqueues* and then  $n$  *extract-mins* in time  $o(n \log n)$ . This is impossible with comparison-based algorithms. ■

## Tournament Heap

- *enqueue*:  $O(\log n)$
- *find-min*:  $O(\log n)$
- *extract-min*:  $O(\log n)$
- *meld*:  $O(\log m + \log n)$ .

# Where We Stand

- We can't make both *enqueue* and *extract-min* run in time  $o(\log n)$ .
- However, we could conceivably make one of them faster.
- **Question:** Which one should we prioritize?
- Probably *enqueue*, since we aren't guaranteed to have to remove all added items.
- **Goal:** Make *enqueue* take time  $O(1)$ .

## Tournament Heap

- *enqueue*:  $O(\log n)$
- *find-min*:  $O(\log n)$
- *extract-min*:  $O(\log n)$
- *meld*:  $O(\log m + \log n)$ .

# Where We Stand

- The *enqueue* operation is implemented in terms of *meld*.
- If we want *enqueue* to run in time  $O(1)$ , we'll need *meld* to take time  $O(1)$ .
- How could we accomplish this?

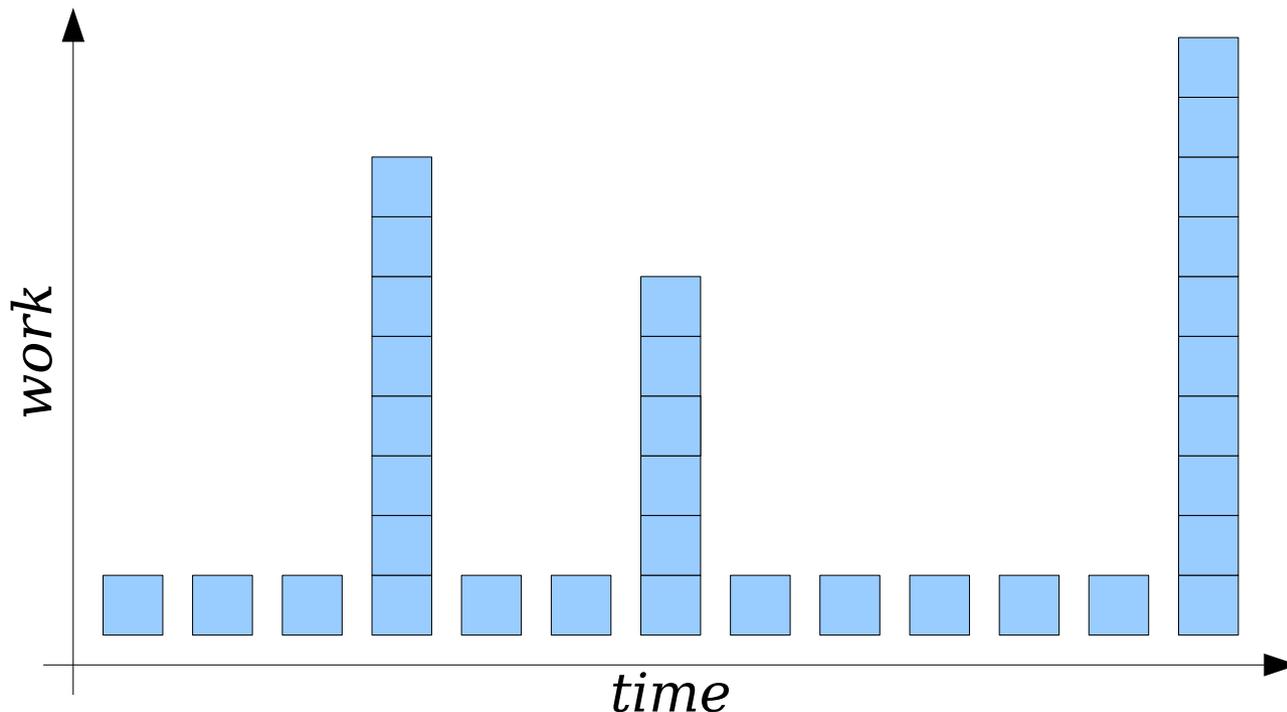
## Tournament Heap

- *enqueue*:  $O(\log n)$
- *find-min*:  $O(\log n)$
- *extract-min*:  $O(\log n)$
- *meld*:  $O(\log m + \log n)$ .

# Thinking With Amortization

# Refresher: Amortization

- In an amortized efficient data structure, some operations can take much longer than others, provided that previous operations didn't take too long to finish.
- Think dishwashers: you may have to do a big cleanup at some point, but that's because you did basically no work to wash all the dishes you placed in the dishwasher.

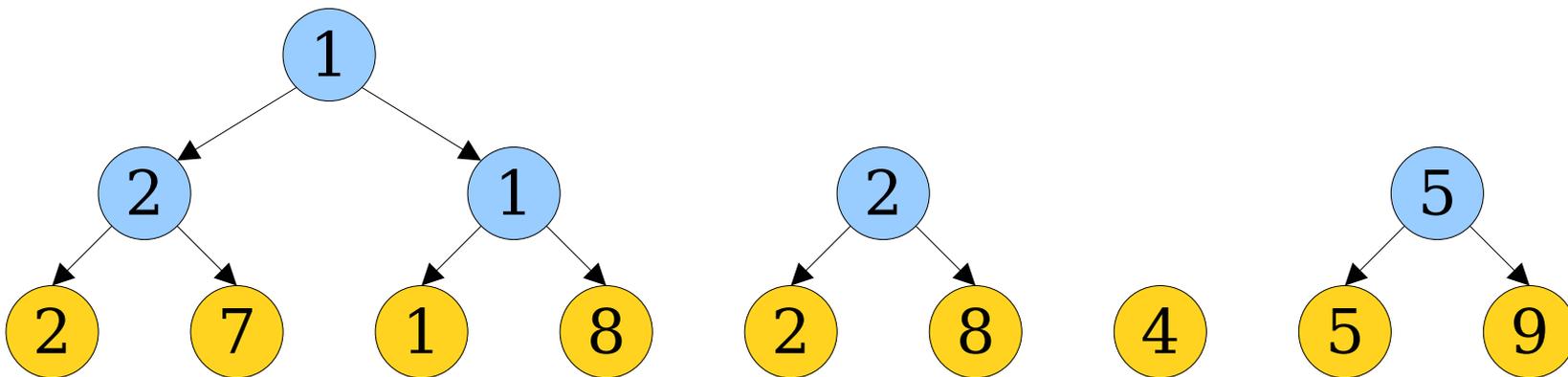


# Lazy Melding

- Consider the following lazy *melding* approach:

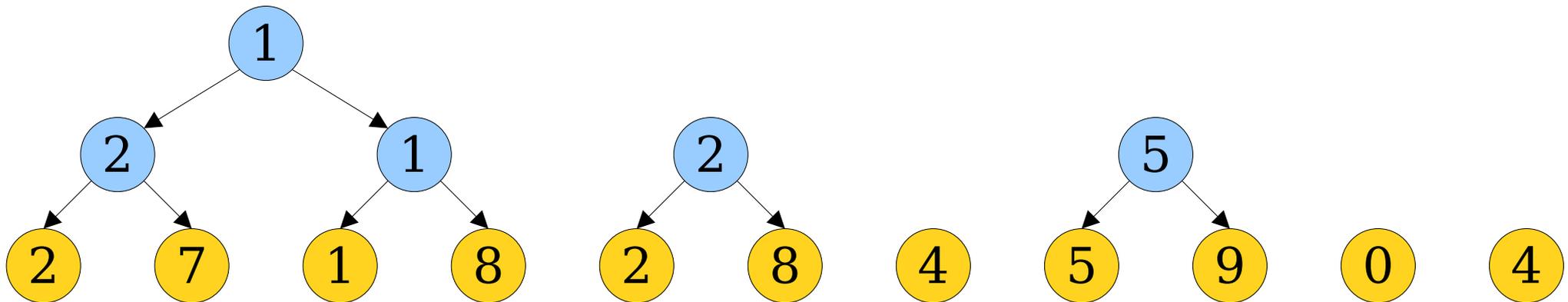
*To meld together two tournament heaps, just combine the two sets of trees together.*

- **Intuition:** Why do any work to organize keys if we're not going to do an *extract-min*? We'll worry about cleanup then.



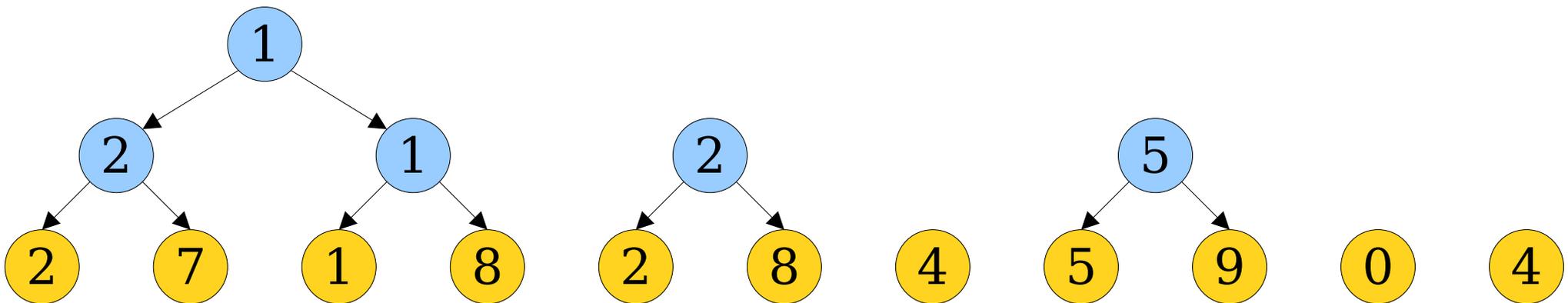
# Lazy Melding

- If we store our list of trees as circularly, doubly-linked lists, we can concatenate tree lists in time  $O(1)$ .
  - Cost of a *meld*:  $O(1)$ .
  - Cost of an *enqueue*:  $O(1)$ .
- If it sounds too good to be true, it probably is.



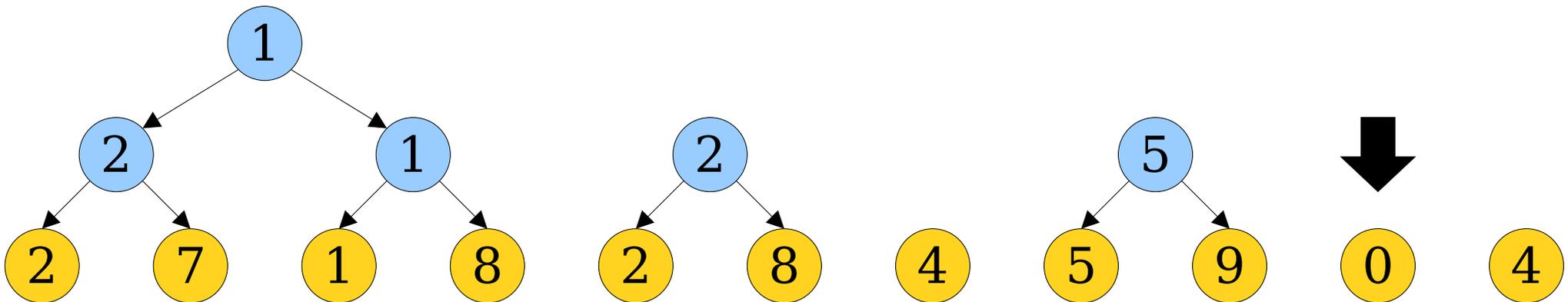
# Lazy Melding

- Imagine that we implement *extract-min* the same way as before:
  - Find the packet with the minimum.
  - “Fracture” that packet to expose smaller packets.
  - Meld those packets back in with the master list.
- What happens if we do this with lazy melding?



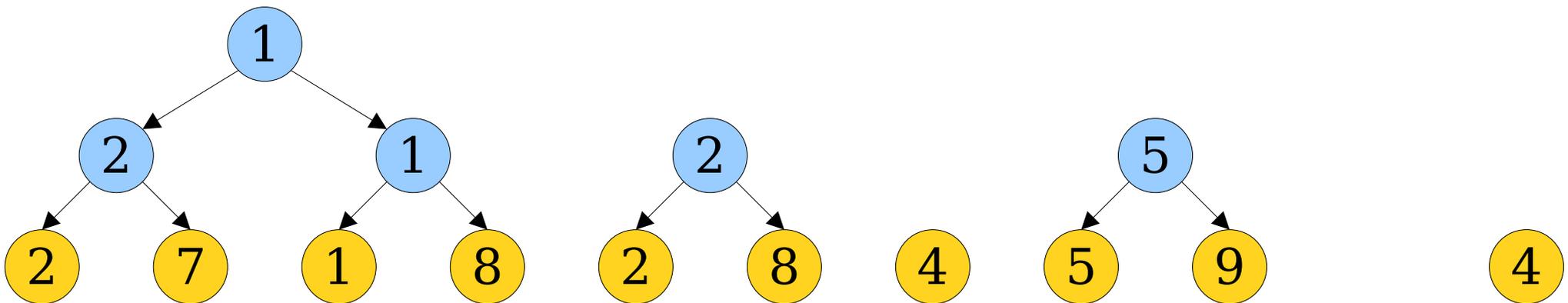
# Lazy Melding

- Imagine that we implement *extract-min* the same way as before:
  - Find the packet with the minimum.
  - “Fracture” that packet to expose smaller packets.
  - Meld those packets back in with the master list.
- What happens if we do this with lazy melding?



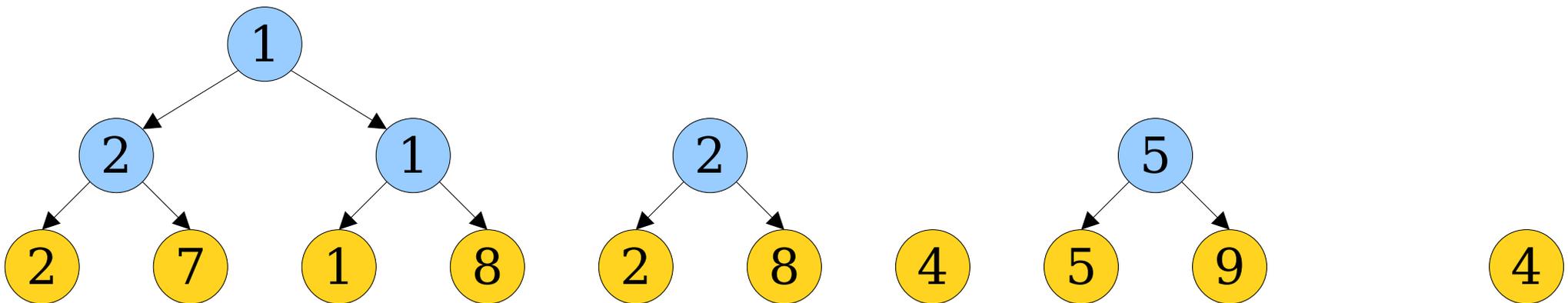
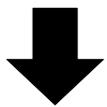
# Lazy Melding

- Imagine that we implement *extract-min* the same way as before:
  - Find the packet with the minimum.
  - “Fracture” that packet to expose smaller packets.
  - Meld those packets back in with the master list.
- What happens if we do this with lazy melding?



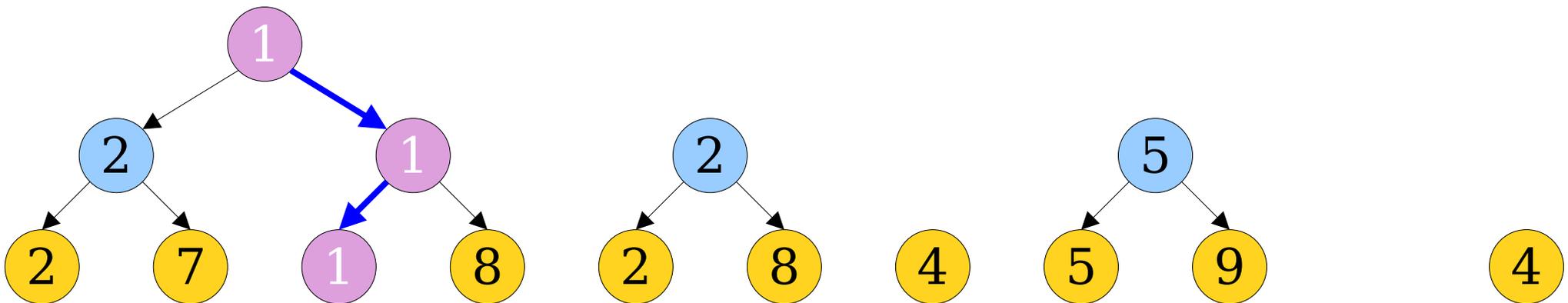
# Lazy Melding

- Imagine that we implement *extract-min* the same way as before:
  - Find the packet with the minimum.
  - “Fracture” that packet to expose smaller packets.
  - Meld those packets back in with the master list.
- What happens if we do this with lazy melding?



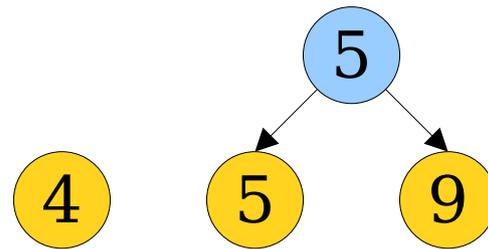
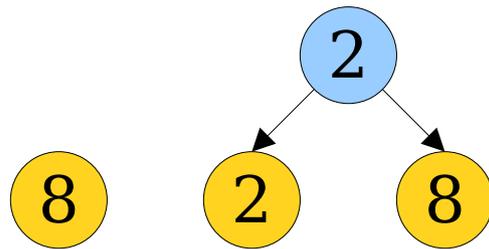
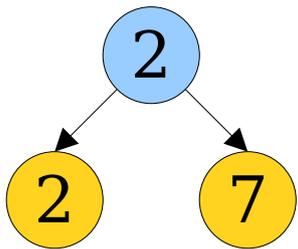
# Lazy Melding

- Imagine that we implement *extract-min* the same way as before:
  - Find the packet with the minimum.
  - “Fracture” that packet to expose smaller packets.
  - Meld those packets back in with the master list.
- What happens if we do this with lazy melding?



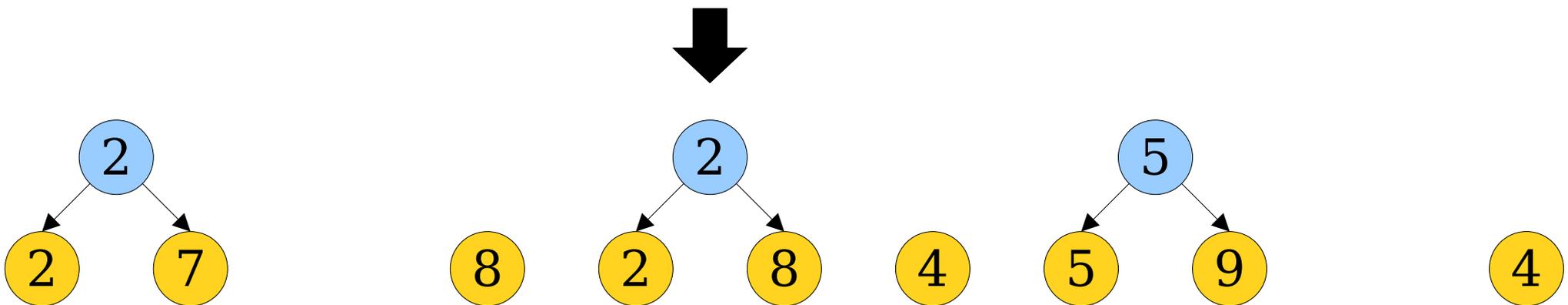
# Lazy Melding

- Imagine that we implement *extract-min* the same way as before:
  - Find the packet with the minimum.
  - “Fracture” that packet to expose smaller packets.
  - Meld those packets back in with the master list.
- What happens if we do this with lazy melding?



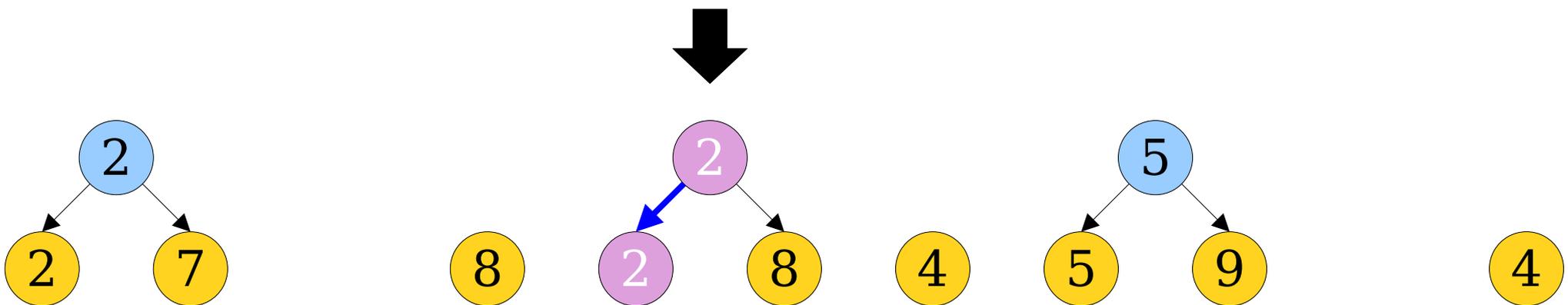
# Lazy Melding

- Imagine that we implement *extract-min* the same way as before:
  - Find the packet with the minimum.
  - “Fracture” that packet to expose smaller packets.
  - Meld those packets back in with the master list.
- What happens if we do this with lazy melding?



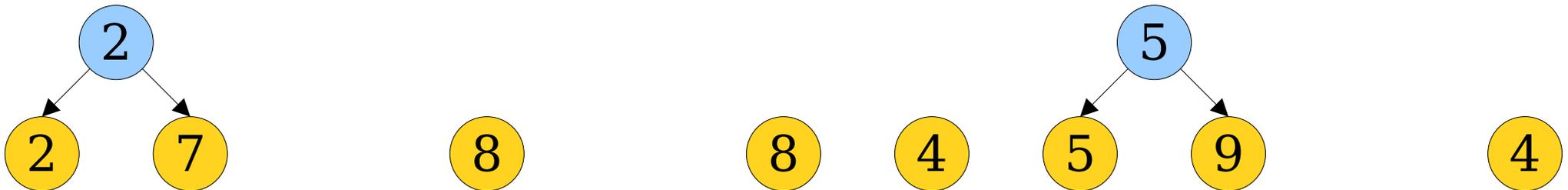
# Lazy Melding

- Imagine that we implement *extract-min* the same way as before:
  - Find the packet with the minimum.
  - “Fracture” that packet to expose smaller packets.
  - Meld those packets back in with the master list.
- What happens if we do this with lazy melding?



# Lazy Melding

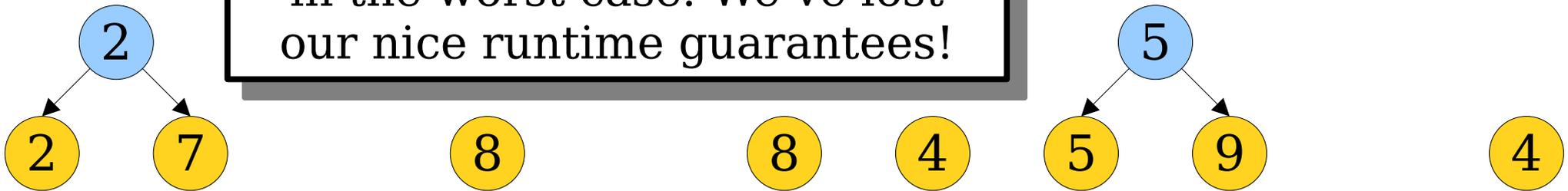
- Imagine that we implement *extract-min* the same way as before:
  - Find the packet with the minimum.
  - “Fracture” that packet to expose smaller packets.
  - Meld those packets back in with the master list.
- What happens if we do this with lazy melding?



# Lazy Melding

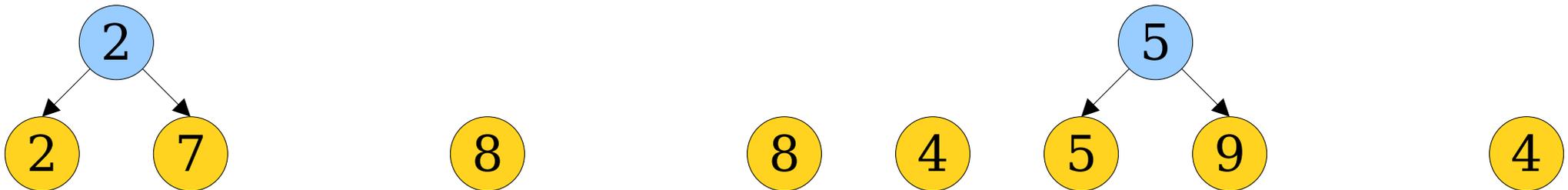
- Imagine that we implement *extract-min* the same way as before:
  - Find the packet with the minimum.
  - “Fracture” that packet to expose smaller packets.
  - Meld those packets back in with the master list.
- What happens if we do this with lazy melding?

Each pass of finding the minimum value takes time  $\Theta(n)$  in the worst case. We've lost our nice runtime guarantees!



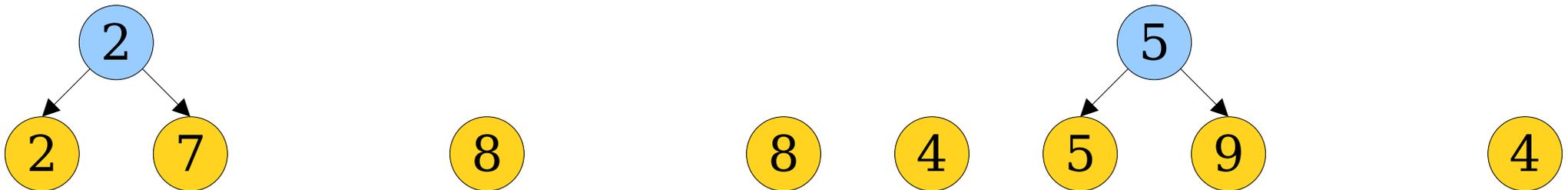
# Washing the Dishes

- Every *meld* (and *enqueue*) creates some “dirty dishes” (small trees) that we need to clean up later.
- If we never clean them up, then our *extract-min* will be too slow to be usable.
- **Idea:** Change *extract-min* to “wash the dishes” and make things look nice and pretty again.
- **Question:** What does “wash the dishes” mean here?



# Washing the Dishes

- With our eager *meld* (and *enqueue*) strategy, our priority queue never had more than one tree of each height.
- This kept the number of trees low, which is why each operation was so fast.
- **Idea:** After doing an *extract-min*, do a *coalesce step* to ensure there's at most one tree of each height. This gets us to where we would be if we had been doing cleanup as we go.



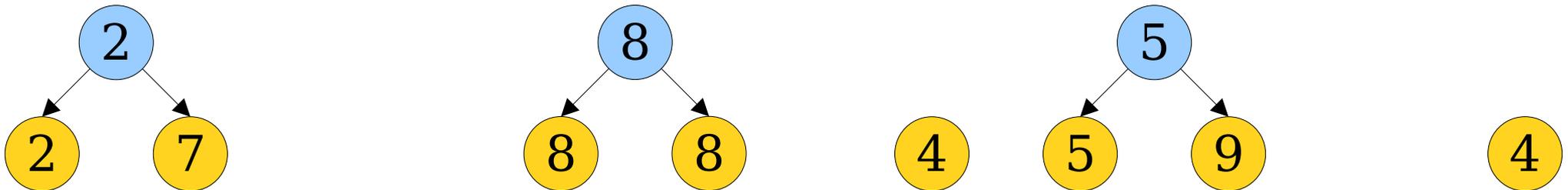
# Washing the Dishes

- With our eager *meld* (and *enqueue*) strategy, our priority queue never had more than one tree of each height.
- This kept the number of trees low, which is why each operation was so fast.
- **Idea:** After doing an *extract-min*, do a *coalesce step* to ensure there's at most one tree of each height. This gets us to where we would be if we had been doing cleanup as we go.



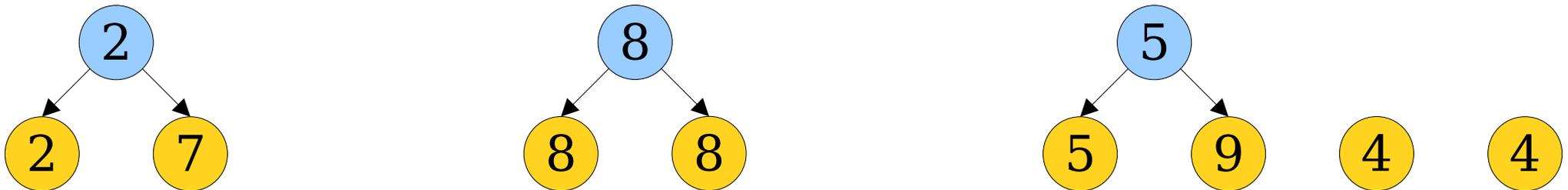
# Washing the Dishes

- With our eager *meld* (and *enqueue*) strategy, our priority queue never had more than one tree of each height.
- This kept the number of trees low, which is why each operation was so fast.
- **Idea:** After doing an *extract-min*, do a *coalesce step* to ensure there's at most one tree of each height. This gets us to where we would be if we had been doing cleanup as we go.



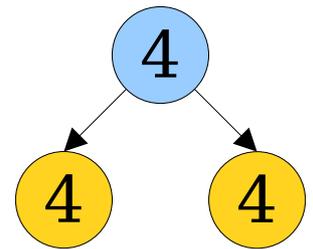
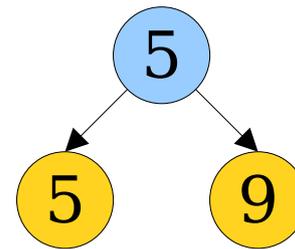
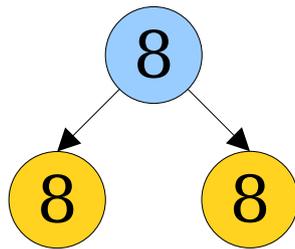
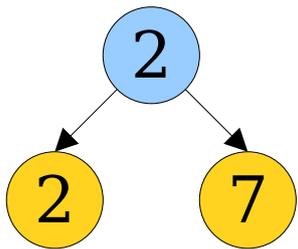
# Washing the Dishes

- With our eager *meld* (and *enqueue*) strategy, our priority queue never had more than one tree of each height.
- This kept the number of trees low, which is why each operation was so fast.
- **Idea:** After doing an *extract-min*, do a *coalesce step* to ensure there's at most one tree of each height. This gets us to where we would be if we had been doing cleanup as we go.



# Washing the Dishes

- With our eager *meld* (and *enqueue*) strategy, our priority queue never had more than one tree of each height.
- This kept the number of trees low, which is why each operation was so fast.
- **Idea:** After doing an *extract-min*, do a *coalesce step* to ensure there's at most one tree of each height. This gets us to where we would be if we had been doing cleanup as we go.



# Washing the Dishes

- With our eager *meld* (and *enqueue*) strategy, our priority queue never had more than one tree of each height.
- This kept the number of trees low, which is why each operation was so fast.
- **Idea:** After doing an *extract-min*, do a *coalesce step* to ensure there's at most one tree of each height. This gets us to where we would be if we had been doing cleanup as we go.



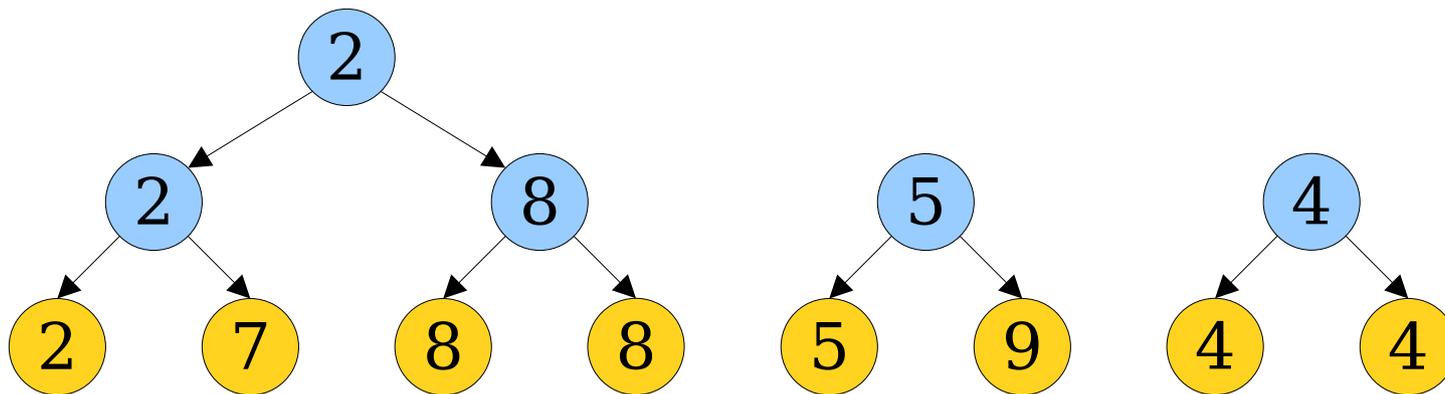
# Washing the Dishes

- With our eager *meld* (and *enqueue*) strategy, our priority queue never had more than one tree of each height.
- This kept the number of trees low, which is why each operation was so fast.
- **Idea:** After doing an *extract-min*, do a *coalesce step* to ensure there's at most one tree of each height. This gets us to where we would be if we had been doing cleanup as we go.



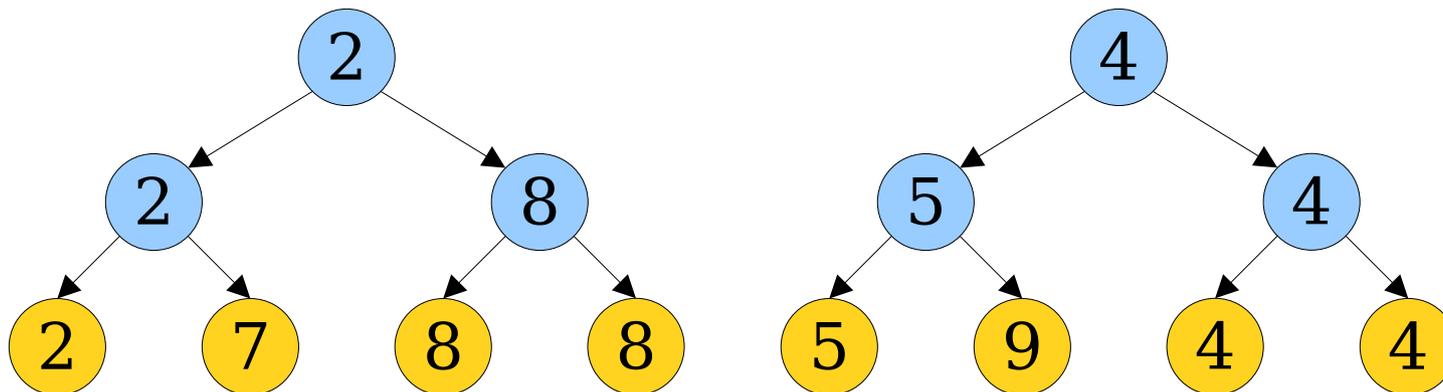
# Washing the Dishes

- With our eager *meld* (and *enqueue*) strategy, our priority queue never had more than one tree of each height.
- This kept the number of trees low, which is why each operation was so fast.
- **Idea:** After doing an *extract-min*, do a *coalesce step* to ensure there's at most one tree of each height. This gets us to where we would be if we had been doing cleanup as we go.



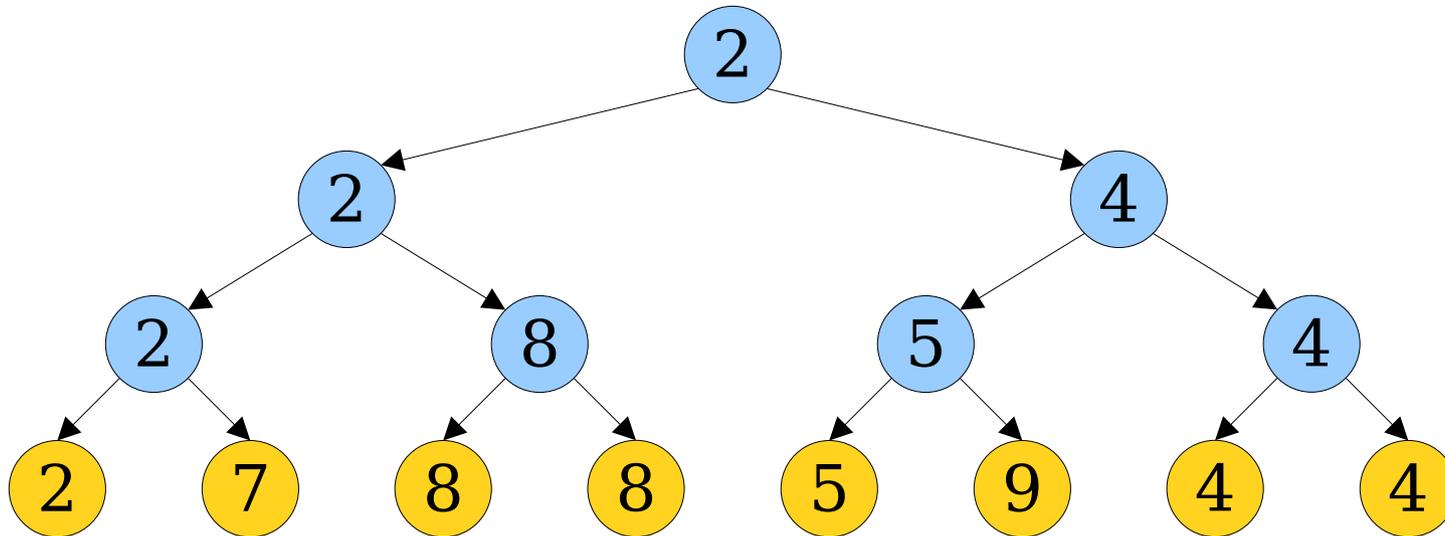
# Washing the Dishes

- With our eager *meld* (and *enqueue*) strategy, our priority queue never had more than one tree of each height.
- This kept the number of trees low, which is why each operation was so fast.
- **Idea:** After doing an *extract-min*, do a *coalesce step* to ensure there's at most one tree of each height. This gets us to where we would be if we had been doing cleanup as we go.



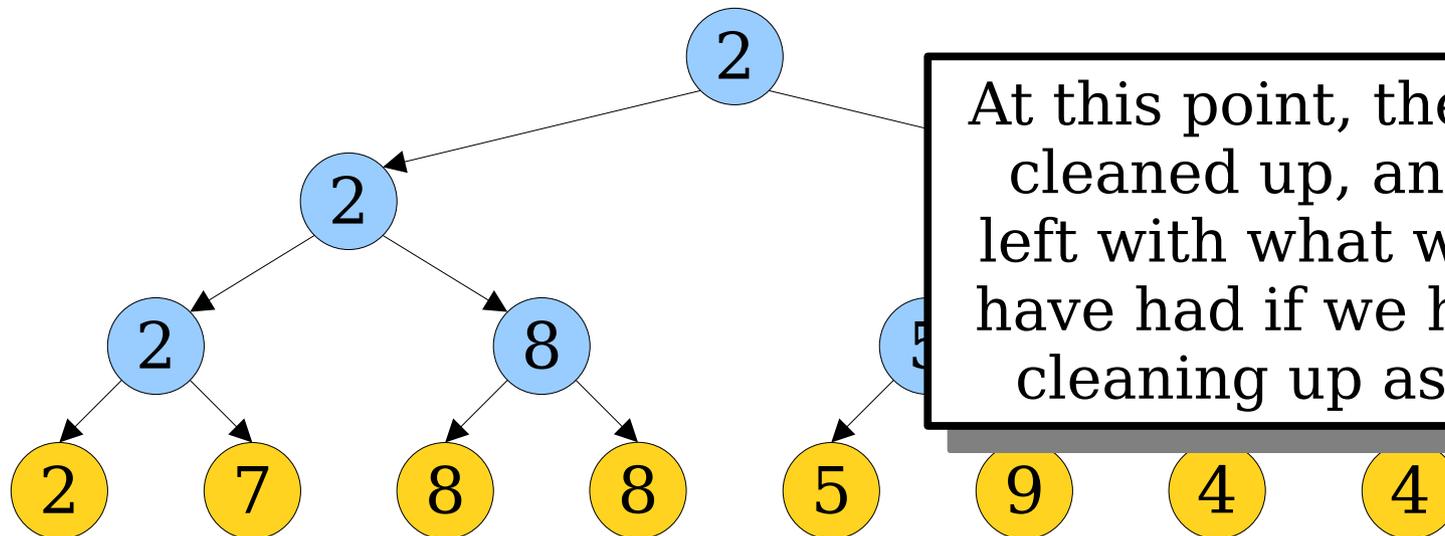
# Washing the Dishes

- With our eager *meld* (and *enqueue*) strategy, our priority queue never had more than one tree of each height.
- This kept the number of trees low, which is why each operation was so fast.
- **Idea:** After doing an *extract-min*, do a *coalesce step* to ensure there's at most one tree of each height. This gets us to where we would be if we had been doing cleanup as we go.



# Washing the Dishes

- With our eager *meld* (and *enqueue*) strategy, our priority queue never had more than one tree of each height.
- This kept the number of trees low, which is why each operation was so fast.
- **Idea:** After doing an *extract-min*, do a *coalesce step* to ensure there's at most one tree of each height. This gets us to where we would be if we had been doing cleanup as we go.

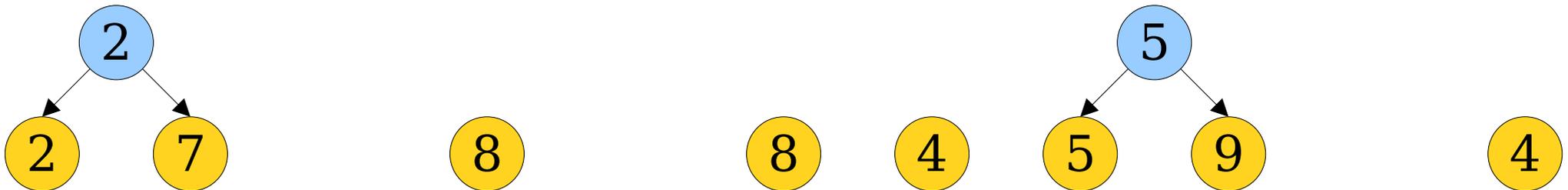


# Where We're Going

- A **lazy tournament heap** is a tournament heap, modified as follows:
  - The **meld** operation is lazy. It just combines the two groups of trees together.
  - After doing an **extract-min**, we do a **coalesce** to combine together trees until there's at most one tree of each height.
- Intuitively, we'd expect this to amortize away nicely, since the "mess" left by **meld** gets cleaned up later on by a future **extract-min**.
- Questions left to answer:
  - How do we efficiently implement the **coalesce** operation?
  - How efficient is this approach, in an amortized sense?

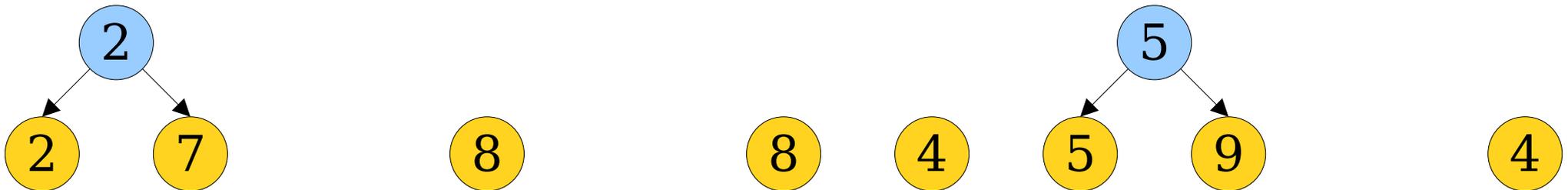
# Coalescing Trees

- The *coalesce* step repeatedly combines trees together until there's at most one tree of each height.
- How do we implement this so that it runs quickly?



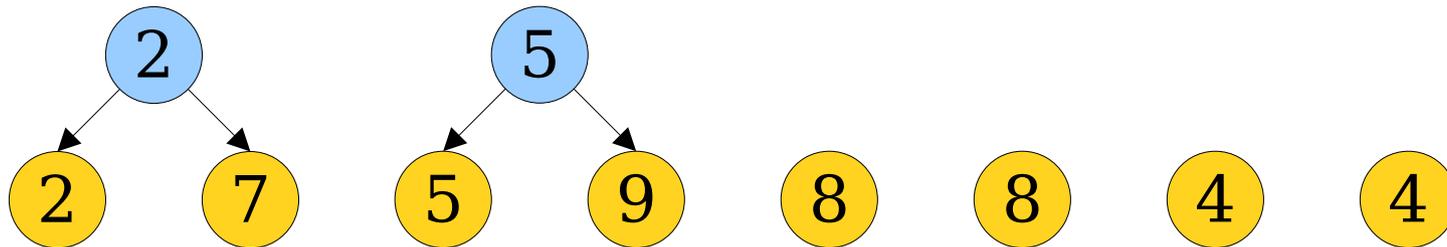
# Coalescing Trees

- **Observation:** This would be a *lot* easier to do if all the trees were sorted by height.



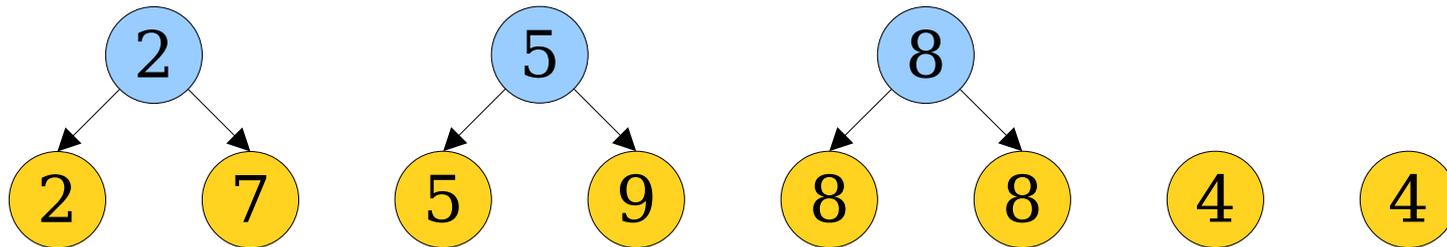
# Coalescing Trees

- **Observation:** This would be a *lot* easier to do if all the trees were sorted by height.



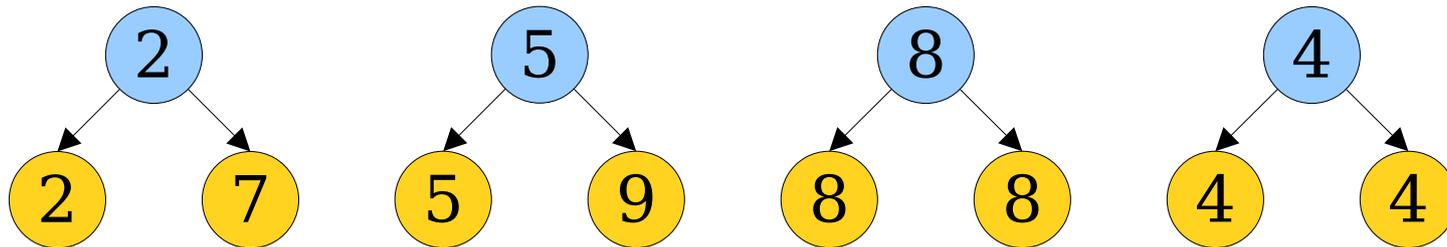
# Coalescing Trees

- **Observation:** This would be a *lot* easier to do if all the trees were sorted by height.



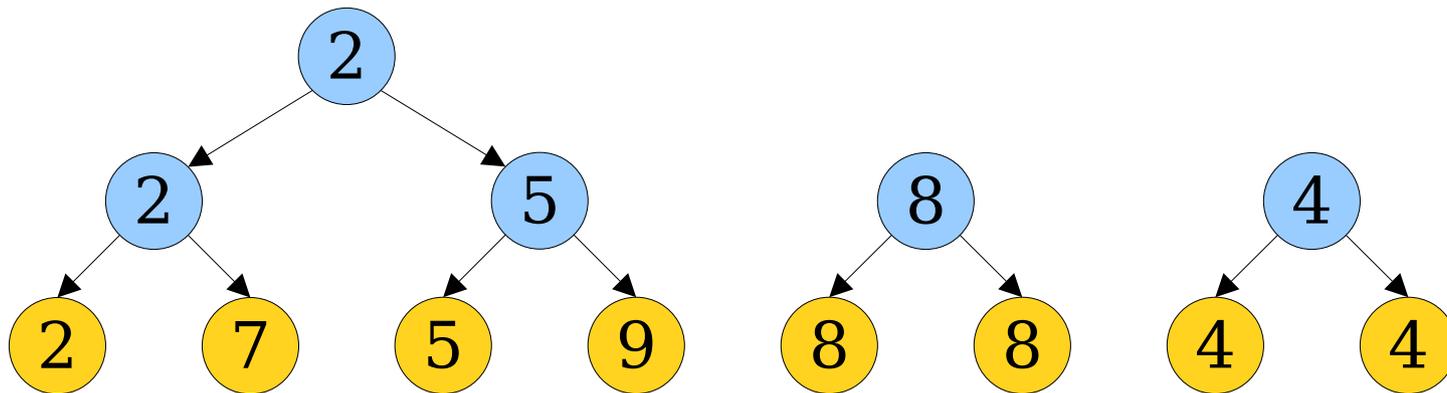
# Coalescing Trees

- **Observation:** This would be a *lot* easier to do if all the trees were sorted by height.



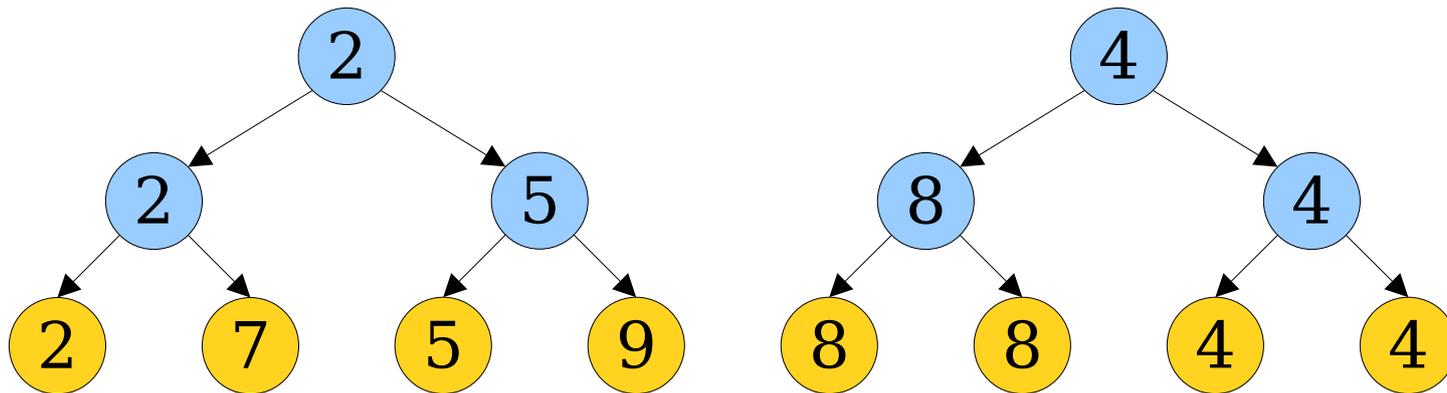
# Coalescing Trees

- **Observation:** This would be a *lot* easier to do if all the trees were sorted by height.



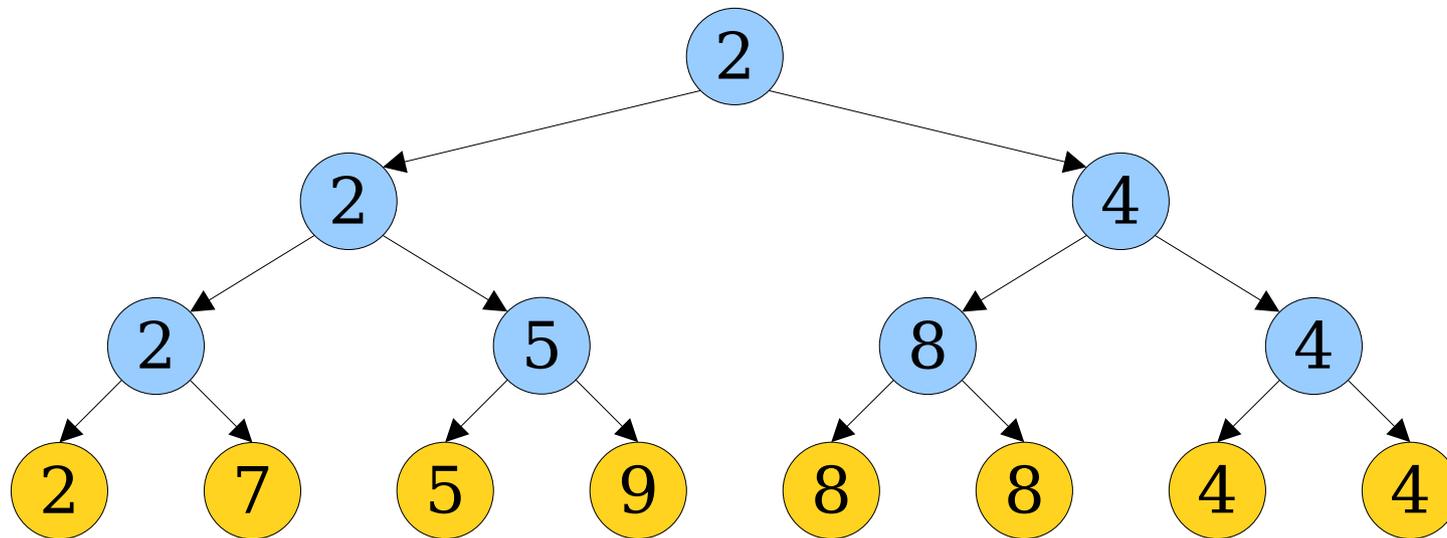
# Coalescing Trees

- **Observation:** This would be a *lot* easier to do if all the trees were sorted by height.



# Coalescing Trees

- **Observation:** This would be a *lot* easier to do if all the trees were sorted by height.

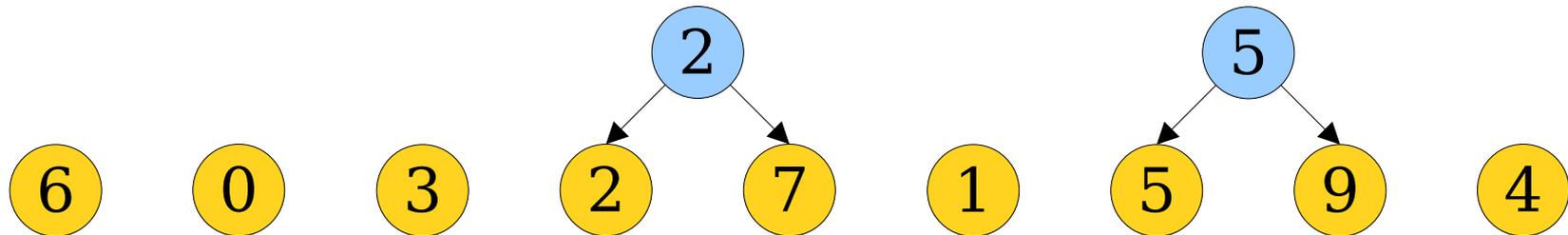


# Coalescing Trees

- **Observation:** This would be a *lot* easier to do if all the trees were sorted by height.
- We can sort our group of  $t$  trees by height in time  $O(t \log t)$  using a standard sorting algorithm.
- **Better idea:** All the sizes are small integers. Use counting sort!

# Coalescing Trees

- Here is a fast implementation of *coalesce*:
  - Distribute the trees into an array of buckets big enough to hold trees of heights 0, 1, 2, ...,  $\lceil \log_2 (n + 1) \rceil$ .
  - Start at bucket 0. While there's two or more trees in the bucket, fuse them and place the result one bucket higher.



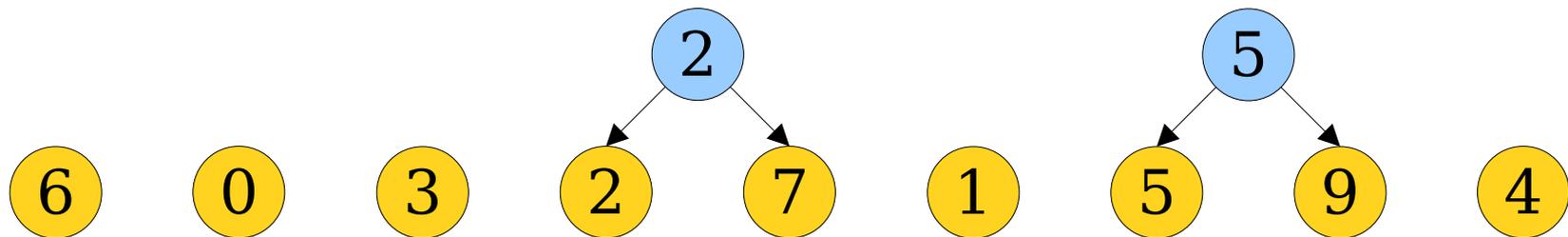
# Coalescing Trees

- Here is a fast implementation of *coalesce*:
  - Distribute the trees into an array of buckets big enough to hold trees of heights 0, 1, 2, ...,  $\lceil \log_2 (n + 1) \rceil$ .
  - Start at bucket 0. While there's two or more trees in the bucket, fuse them and place the result one bucket higher.

*Height 2*

*Height 1*

*Height 0*



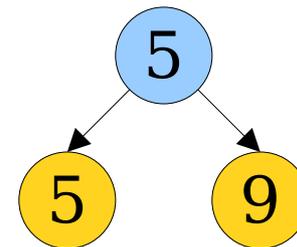
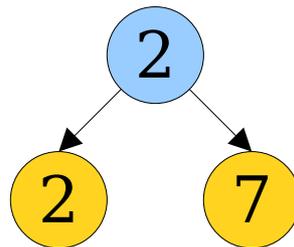
# Coalescing Trees

- Here is a fast implementation of *coalesce*:
  - Distribute the trees into an array of buckets big enough to hold trees of heights 0, 1, 2, ...,  $\lceil \log_2 (n + 1) \rceil$ .
  - Start at bucket 0. While there's two or more trees in the bucket, fuse them and place the result one bucket higher.

*Height 2*

*Height 1*

*Height 0*



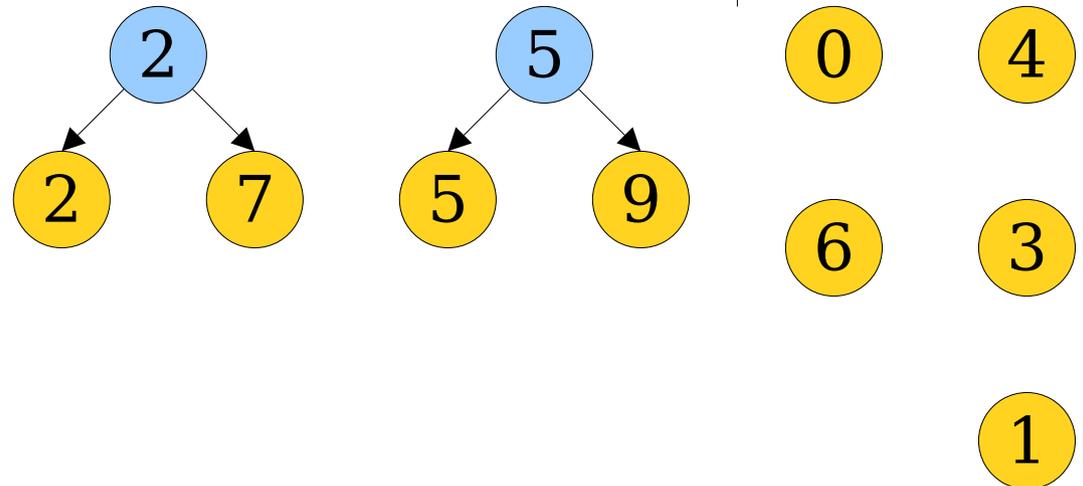
# Coalescing Trees

- Here is a fast implementation of *coalesce*:
  - Distribute the trees into an array of buckets big enough to hold trees of heights 0, 1, 2, ...,  $\lceil \log_2 (n + 1) \rceil$ .
  - Start at bucket 0. While there's two or more trees in the bucket, fuse them and place the result one bucket higher.

*Height 2*

*Height 1*

*Height 0*



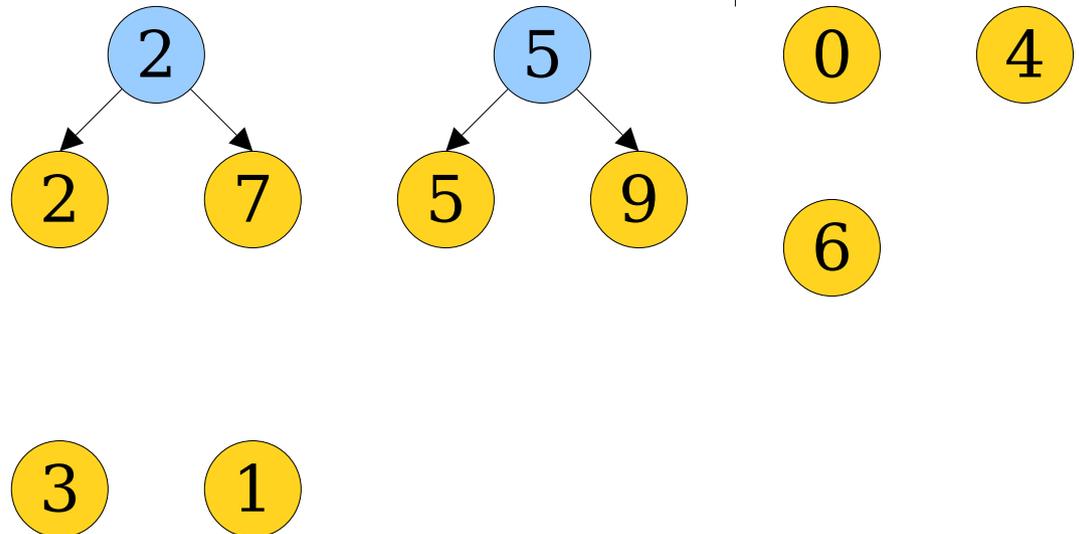
# Coalescing Trees

- Here is a fast implementation of *coalesce*:
  - Distribute the trees into an array of buckets big enough to hold trees of heights 0, 1, 2, ...,  $\lceil \log_2 (n + 1) \rceil$ .
  - Start at bucket 0. While there's two or more trees in the bucket, fuse them and place the result one bucket higher.

*Height 2*

*Height 1*

*Height 0*



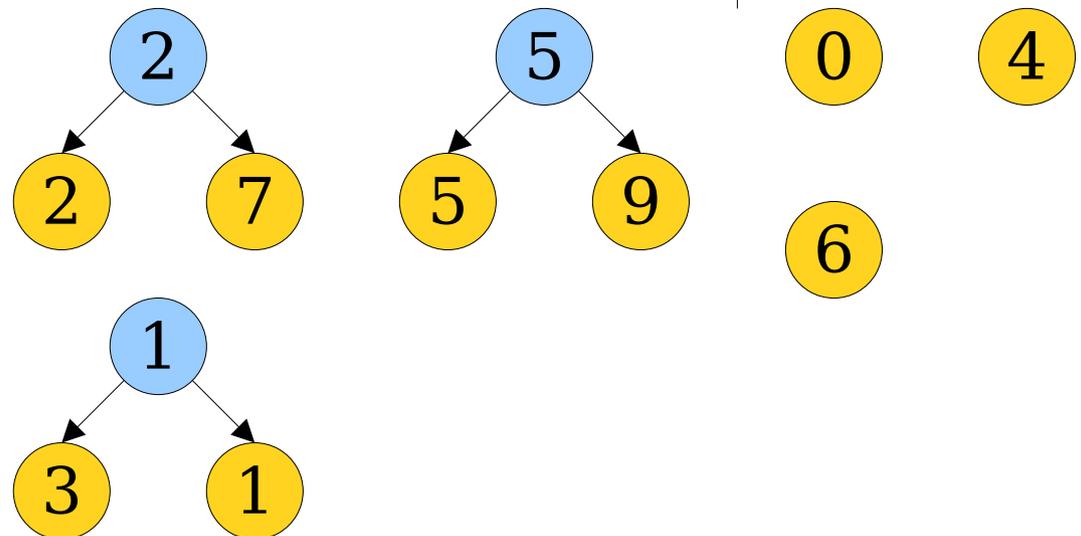
# Coalescing Trees

- Here is a fast implementation of *coalesce*:
  - Distribute the trees into an array of buckets big enough to hold trees of heights 0, 1, 2, ...,  $\lceil \log_2 (n + 1) \rceil$ .
  - Start at bucket 0. While there's two or more trees in the bucket, fuse them and place the result one bucket higher.

*Height 2*

*Height 1*

*Height 0*



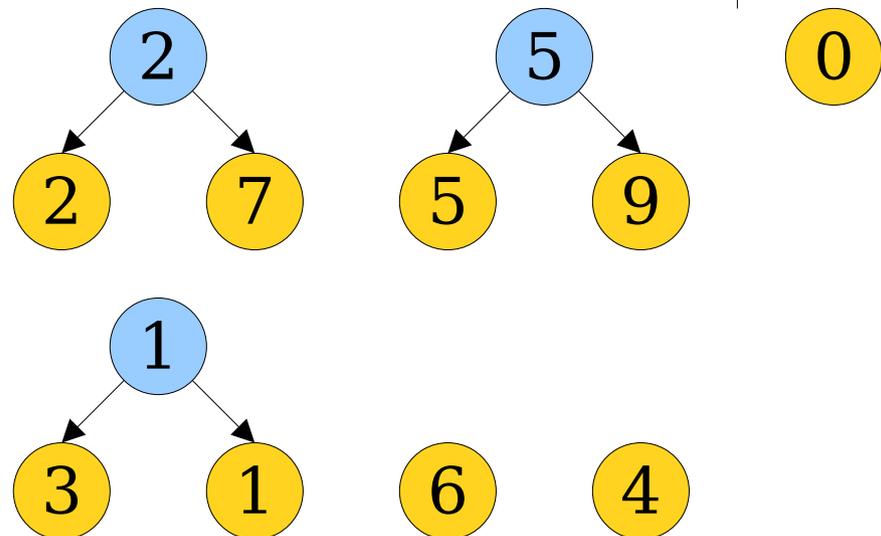
# Coalescing Trees

- Here is a fast implementation of *coalesce*:
  - Distribute the trees into an array of buckets big enough to hold trees of heights 0, 1, 2, ...,  $\lceil \log_2 (n + 1) \rceil$ .
  - Start at bucket 0. While there's two or more trees in the bucket, fuse them and place the result one bucket higher.

*Height 2*

*Height 1*

*Height 0*



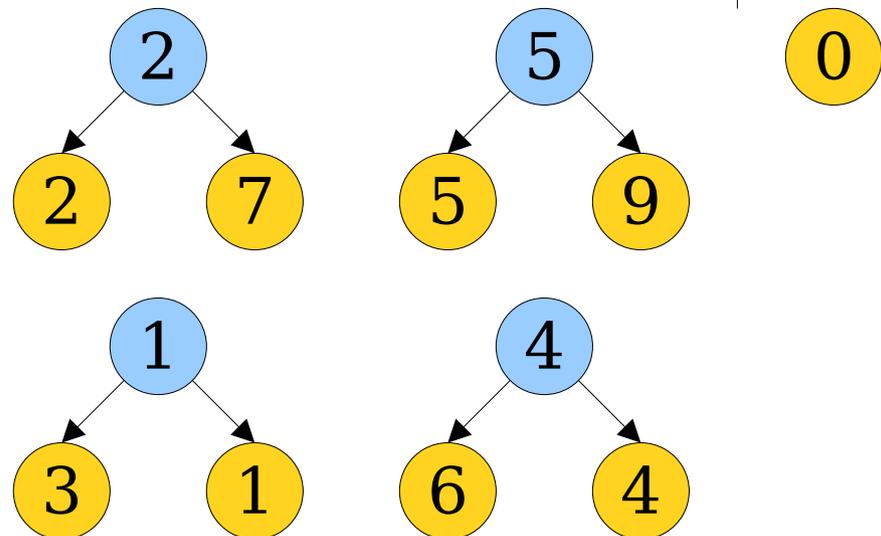
# Coalescing Trees

- Here is a fast implementation of *coalesce*:
  - Distribute the trees into an array of buckets big enough to hold trees of heights 0, 1, 2, ...,  $\lceil \log_2 (n + 1) \rceil$ .
  - Start at bucket 0. While there's two or more trees in the bucket, fuse them and place the result one bucket higher.

*Height 2*

*Height 1*

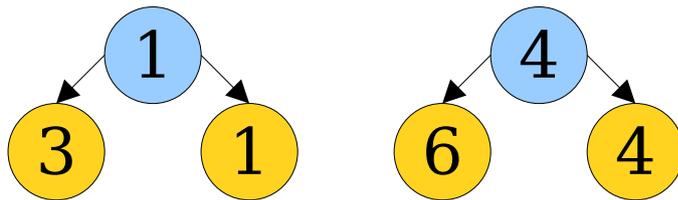
*Height 0*



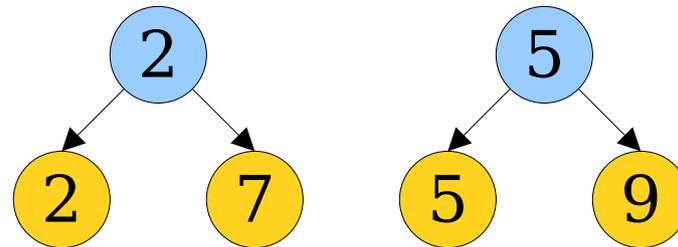
# Coalescing Trees

- Here is a fast implementation of *coalesce*:
  - Distribute the trees into an array of buckets big enough to hold trees of heights 0, 1, 2, ...,  $\lceil \log_2 (n + 1) \rceil$ .
  - Start at bucket 0. While there's two or more trees in the bucket, fuse them and place the result one bucket higher.

*Height 2*



*Height 1*

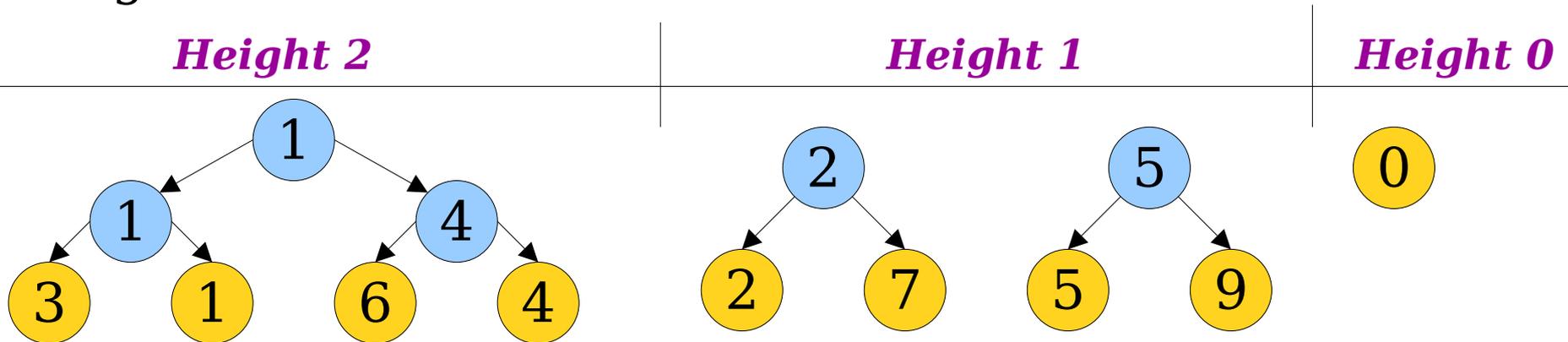


*Height 0*



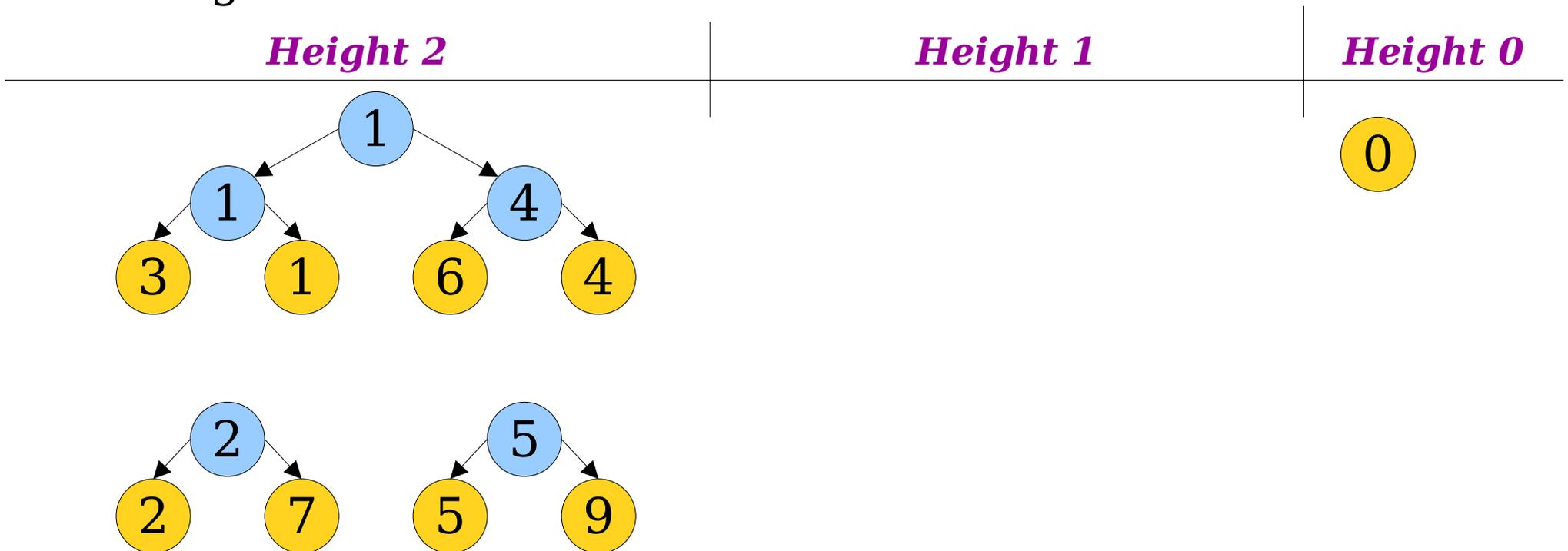
# Coalescing Trees

- Here is a fast implementation of *coalesce*:
  - Distribute the trees into an array of buckets big enough to hold trees of heights 0, 1, 2, ...,  $\lceil \log_2 (n + 1) \rceil$ .
  - Start at bucket 0. While there's two or more trees in the bucket, fuse them and place the result one bucket higher.



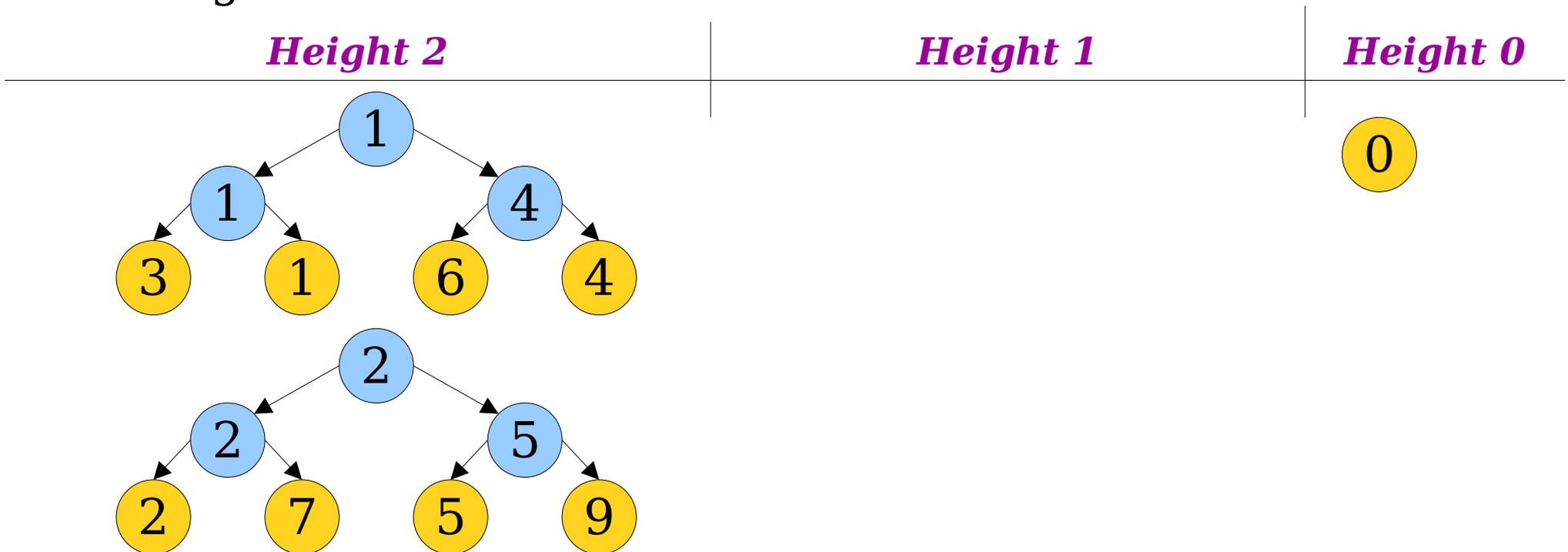
# Coalescing Trees

- Here is a fast implementation of *coalesce*:
  - Distribute the trees into an array of buckets big enough to hold trees of heights 0, 1, 2, ...,  $\lceil \log_2 (n + 1) \rceil$ .
  - Start at bucket 0. While there's two or more trees in the bucket, fuse them and place the result one bucket higher.



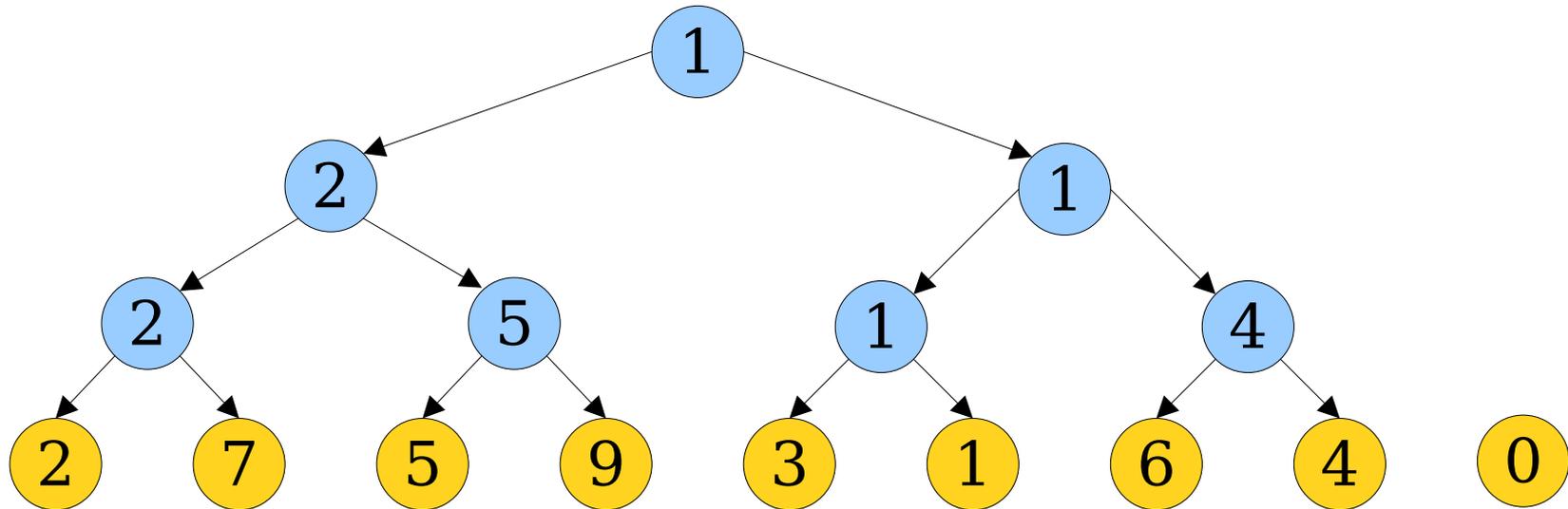
# Coalescing Trees

- Here is a fast implementation of *coalesce*:
  - Distribute the trees into an array of buckets big enough to hold trees of heights 0, 1, 2, ...,  $\lceil \log_2 (n + 1) \rceil$ .
  - Start at bucket 0. While there's two or more trees in the bucket, fuse them and place the result one bucket higher.



# Coalescing Trees

- Here is a fast implementation of *coalesce*:
  - Distribute the trees into an array of buckets big enough to hold trees of heights 0, 1, 2, ...,  $\lceil \log_2 (n + 1) \rceil$ .
  - Start at bucket 0. While there's two or more trees in the bucket, fuse them and place the result one bucket higher.



# Analyzing Coalesce

- **Claim:** Coalescing a group of  $t$  trees takes time  $O(t + \log n)$ .
  - Time to create the array of buckets:  $O(\log n)$ .
  - Time to distribute trees into buckets:  $O(t)$ .
  - Time to fuse trees:  $O(t + \log n)$ 
    - Number of fuses is  $O(t)$ , since each fuse decreases the number of trees by one. Cost per fuse is  $O(1)$ .
    - Need to iterate across  $O(\log n)$  buckets.
- Total work done:  **$O(t + \log n)$** .
- In the worst case, this is  $O(n)$ .

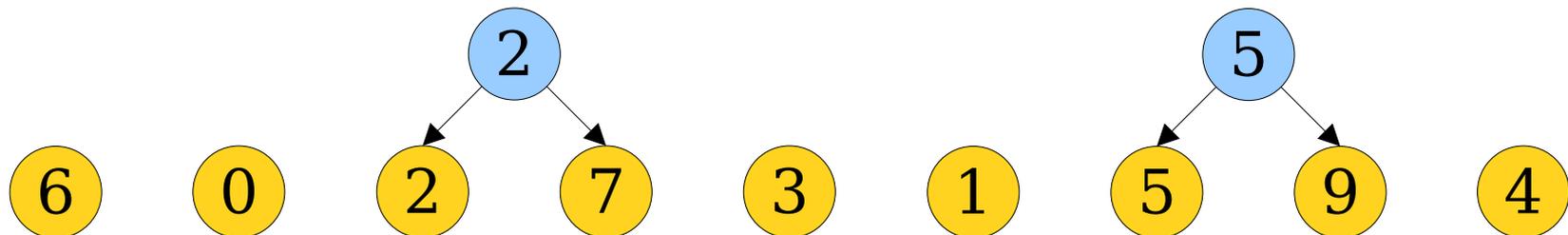
# The Story So Far

- A tournament heap with lazy melding has these worst-case time bounds:
  - ***enqueue***:  $O(1)$
  - ***meld***:  $O(1)$
  - ***find-min***:  $O(1)$
  - ***extract-min***:  $O(n)$ .
- But these are *worst-case* time bounds. Intuitively, things should nicely amortize away.
  - The number of trees grows slowly (one per ***enqueue***).
  - The number of trees drops quickly (at most one tree per order) after an ***extract-min***).

# An Amortized Analysis

- This is a great spot to use an amortized analysis by defining a potential function  $\Phi$ .
- In each case, the idea is to clearly mark what “messes” we need to clean up.
- In our case, each tree is a “mess,” since our future *coalesce* operation has to clean it up.

Set  $\Phi$  to the number of trees in the lazy tournament heap.



# An Amortized Analysis

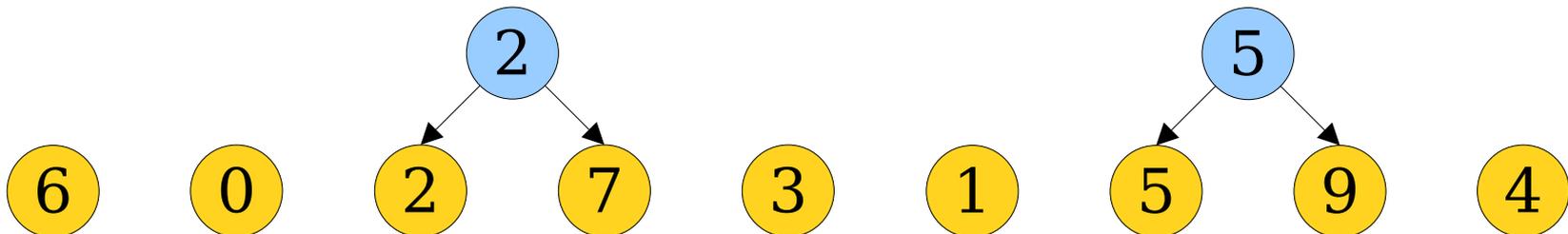
- **Recall:** We assign amortized costs as

$$\text{amortized-cost} = \text{real-cost} + k \cdot \Delta\Phi,$$

where  $\Delta\Phi = \Phi_{\text{after}} - \Phi_{\text{before}}$ .

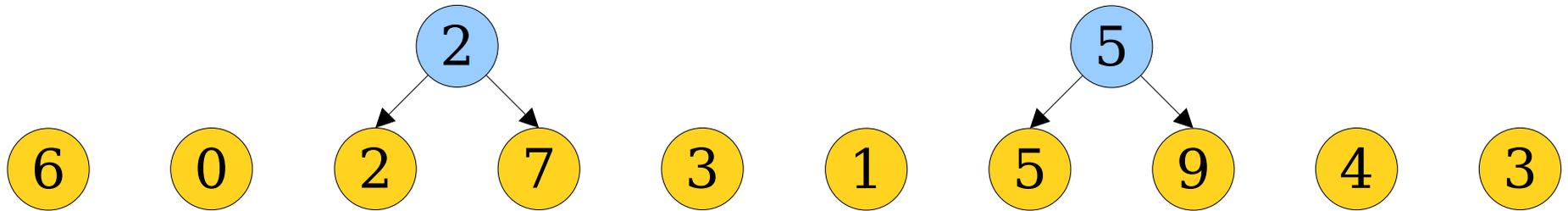
- Increasing  $\Phi$  (adding more trees) artificially boosts costs.
- Decreasing  $\Phi$  (removing trees) artificially lowers costs.
- Let's work out the amortized costs of each operation on a lazy tournament heap.

Set  $\Phi$  to the number of trees in the lazy tournament heap.



# Analyzing an Insertion

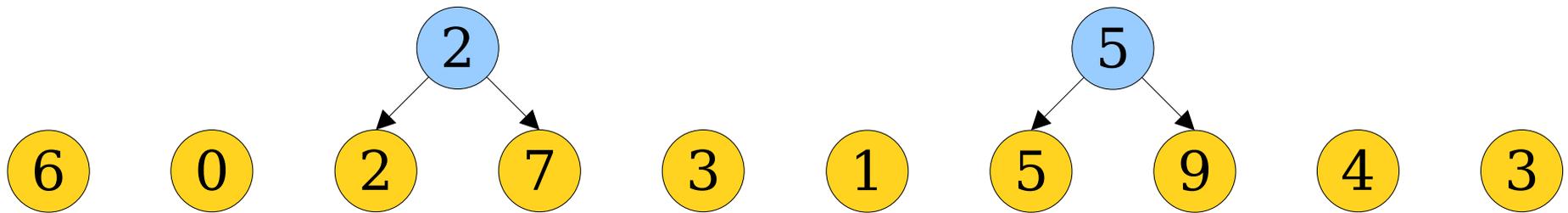
- To *enqueue* a key, we add a new tournament tree to the forest.



Set  $\Phi$  to the number of trees in the lazy tournament heap.

# Analyzing an Insertion

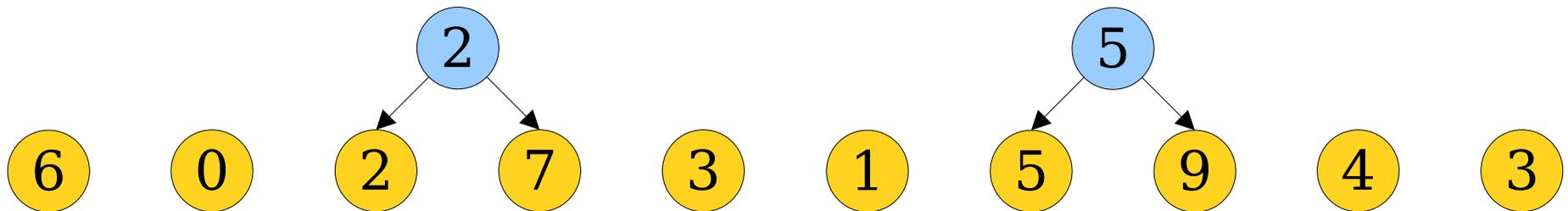
- To *enqueue* a key, we add a new tournament tree to the forest.
- Real cost:  $O(1)$ .  $\Delta\Phi$ :  $+1$
- Amortized cost:  **$O(1)$** .



Set  $\Phi$  to the number of trees in the lazy tournament heap.

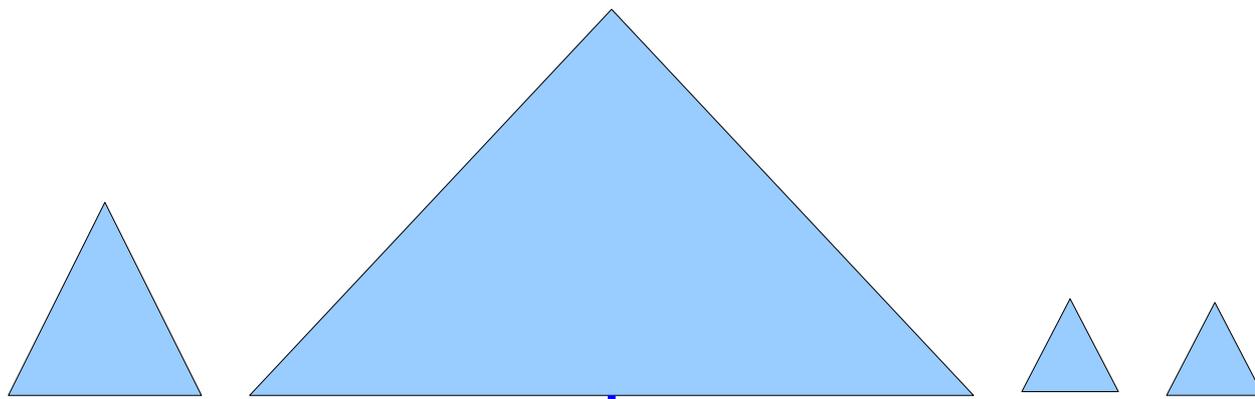
# Analyzing a Meld

- What is the amortized cost of *meld*?
- Real Cost:  $O(1)$ .
- $\Delta\Phi = 0$ .
  - No trees are created or destroyed.
- Amortized cost:  **$O(1)$** .



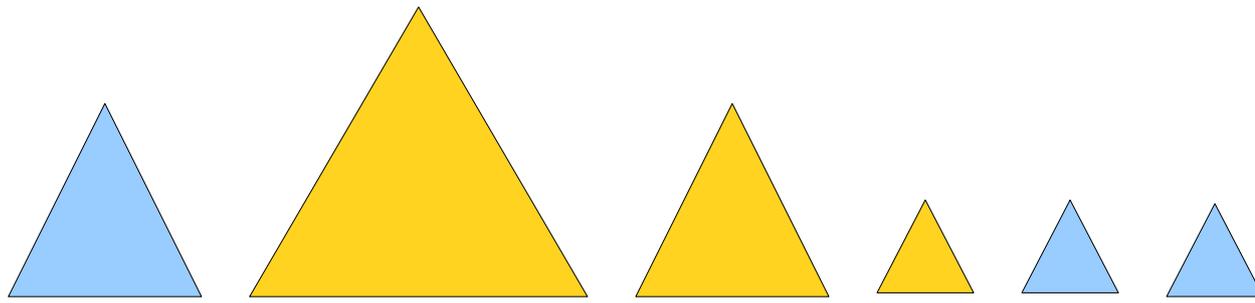
Set  $\Phi$  to the number of trees in the lazy tournament heap.

Analyzing *extract-min*



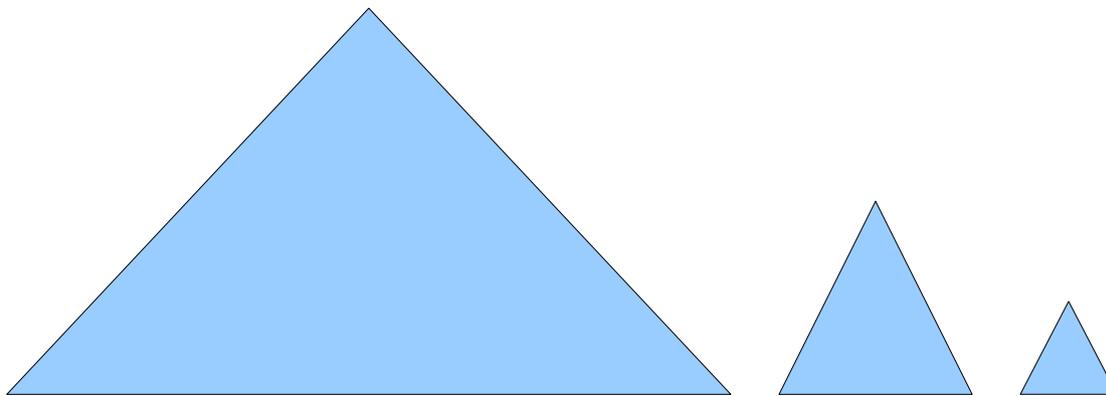
*Find tree with minimum key.*

Work:  $O(t)$   
 $\Phi = t$



*Remove min.  
 Add children to list of trees.*

Work:  $O(\log n)$

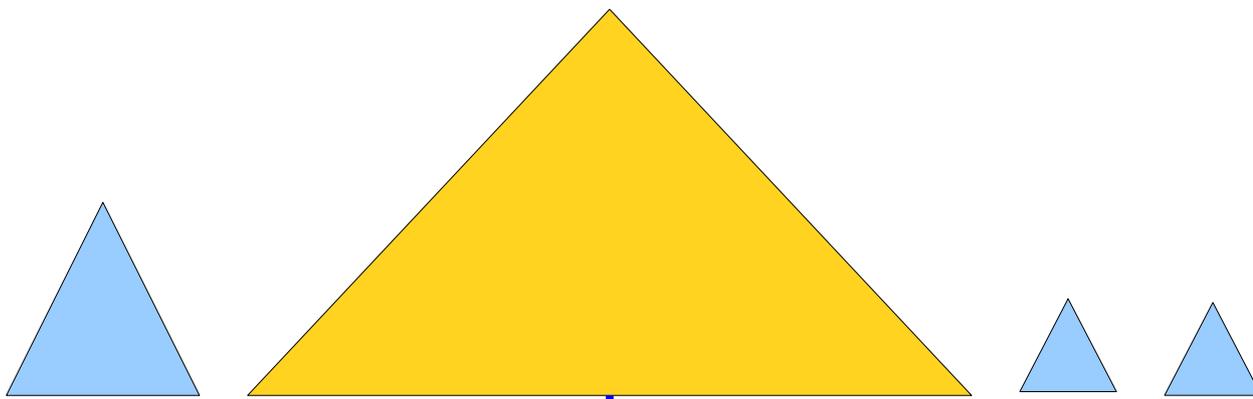


*Run the coalesce algorithm.*

Work:  $O(t + \log n)$   
 $\Phi = O(\log n)$

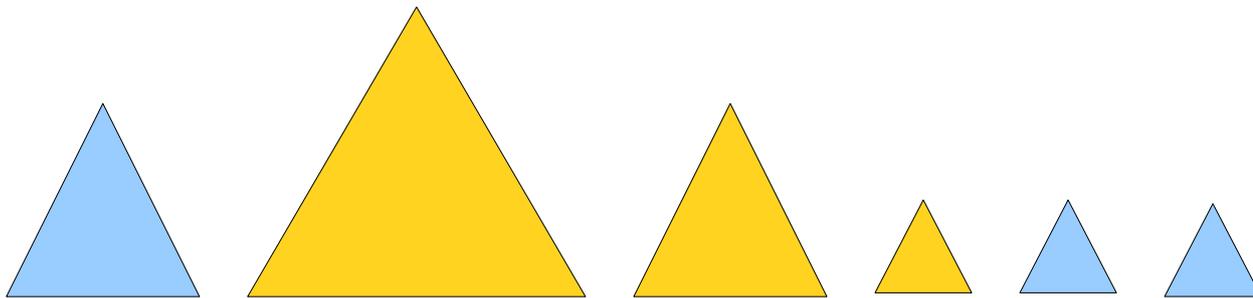
Work:  $O(t + \log n)$

$\Delta\Phi: O(-t + \log n)$



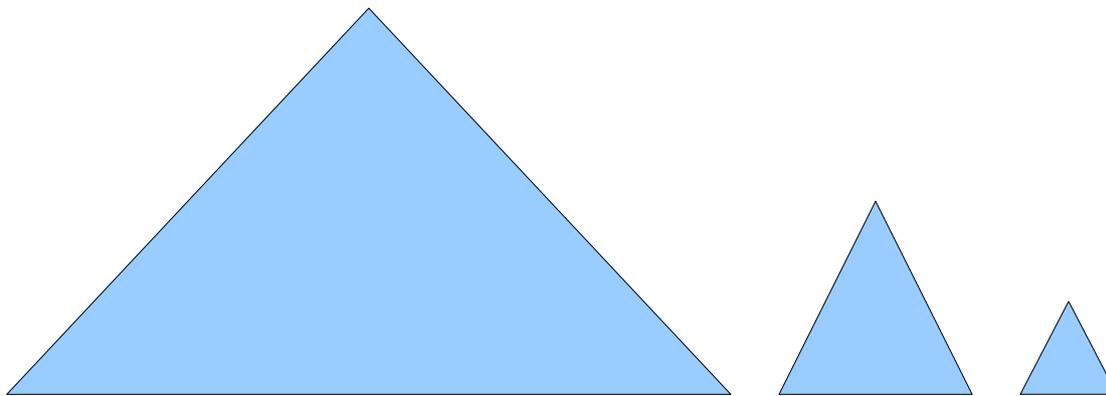
*Find tree with minimum key.*

Work:  $O(t)$   
 $\Phi = t$



*Remove min.  
Add children to  
list of trees.*

Work:  $O(\log n)$



*Run the coalesce  
algorithm.*

Work:  $O(t + \log n)$   
 $\Phi = O(\log n)$

Amortized cost:  **$O(\log n)$** .

# Analyzing Extract-Min

- Suppose we perform an *extract-min* on a lazy tournament heap with  $t$  trees in it.
- Initially, we fracture the tree containing the minimum. This increases the number of trees to  $t + O(\log n)$ .
- The runtime for coalescing these trees is  $O(t + \log n)$ .
- When we're done merging, there will be  $O(\log n)$  trees remaining, so  $\Delta\Phi = -t + O(\log n)$ .
- Amortized cost is

$$\begin{aligned} & O(t + \log n) + k \cdot (-t + O(\log n)) \\ &= O(t) - k \cdot t + k \cdot O(\log n) \\ &= O(\log n). \end{aligned}$$

# The Final Scorecard

- Here's the final scorecard for our lazy tournament heap.
- These are *great* runtimes! We can't improve upon this except by making ***extract-min*** worst-case efficient.
  - This is possible! Check out ***bootstrapped skew binomial heaps*** for details!

## Lazy Tournament Heap

- ***Insert***:  $O(1)$
- ***Find-Min***:  $O(1)$
- ***Extract-Min***:  $O(\log n)^*$
- ***Meld***:  $O(1)$

\* *amortized*

# Major Ideas from Today

- Isometries are a *great* way to design data structures.
  - Here, tournament heaps come from binary arithmetic.
- Designing for amortized efficiency is about building up messes slowly and rapidly cleaning them up.
  - Each individual *enqueue* isn't too bad, and a single *extract-min* fixes all the prior problems.

# Next Time

- ***The Need for decrease-key***
  - A powerful and versatile operation on priority queues.
- ***Abdication Heaps***
  - A variation on lazy tournament heaps with efficient ***decrease-key***.
- ***Analyzing Abdication Heaps***
  - A clever analysis.