Binomial Heaps

Outline for this Week

- ***Binomial Heaps (Today)***
  - A simple, flexible, and versatile priority queue.
- ***Lazy Binomial Heaps (Today)***
  - A powerful building block for designing advanced data structures.
- ***Fibonacci Heaps (Thursday)***
  - A heavyweight and theoretically excellent priority queue.

Review: Priority Queues

Priority Queues

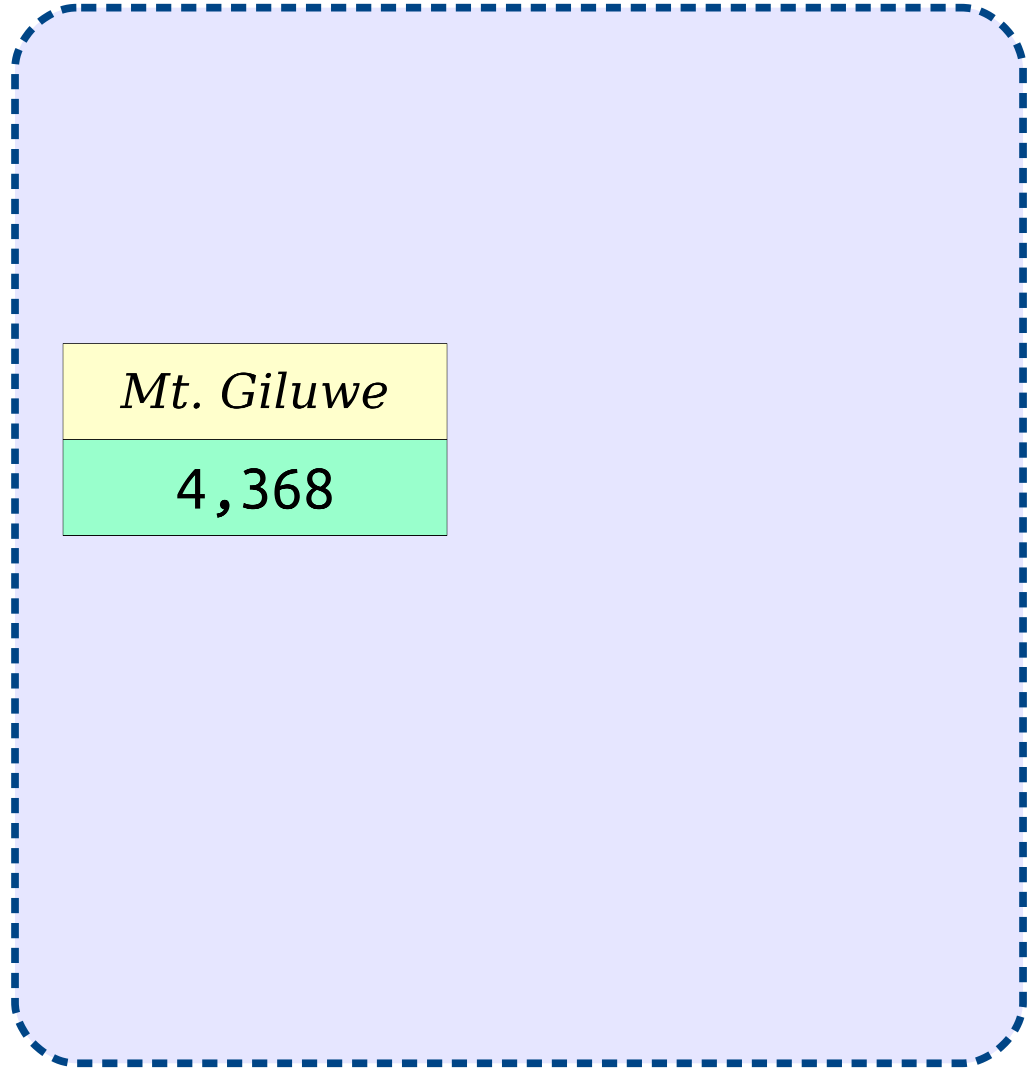
- A ***priority queue*** is a data structure that supports these operations:
  - pq.***enqueue***(v, k), which enqueues element v with key k;
  - pq.***find-min***$()$, which returns the element with the least key; and
  - pq.***extract-min***$()$, which removes and returns the element with the least 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.

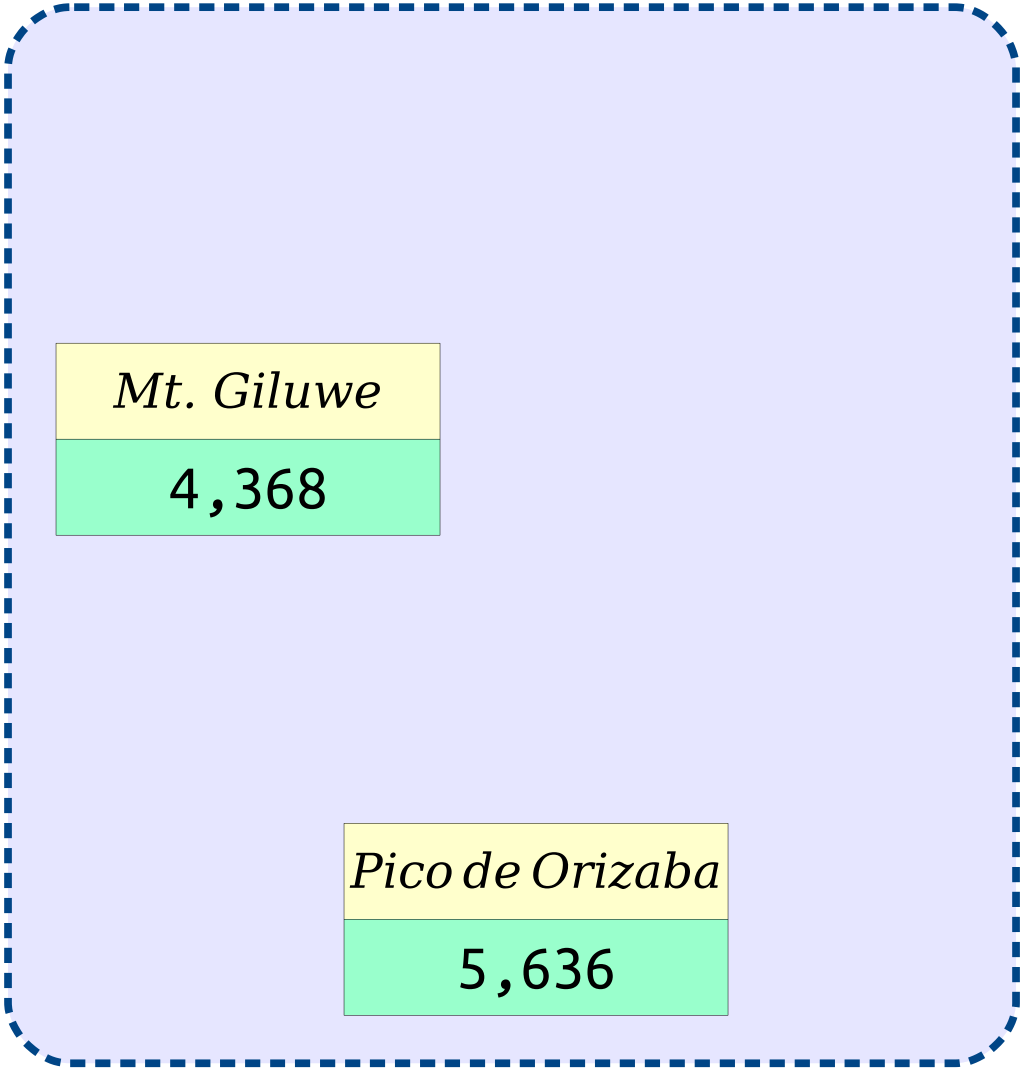
Priority Queues

- 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.



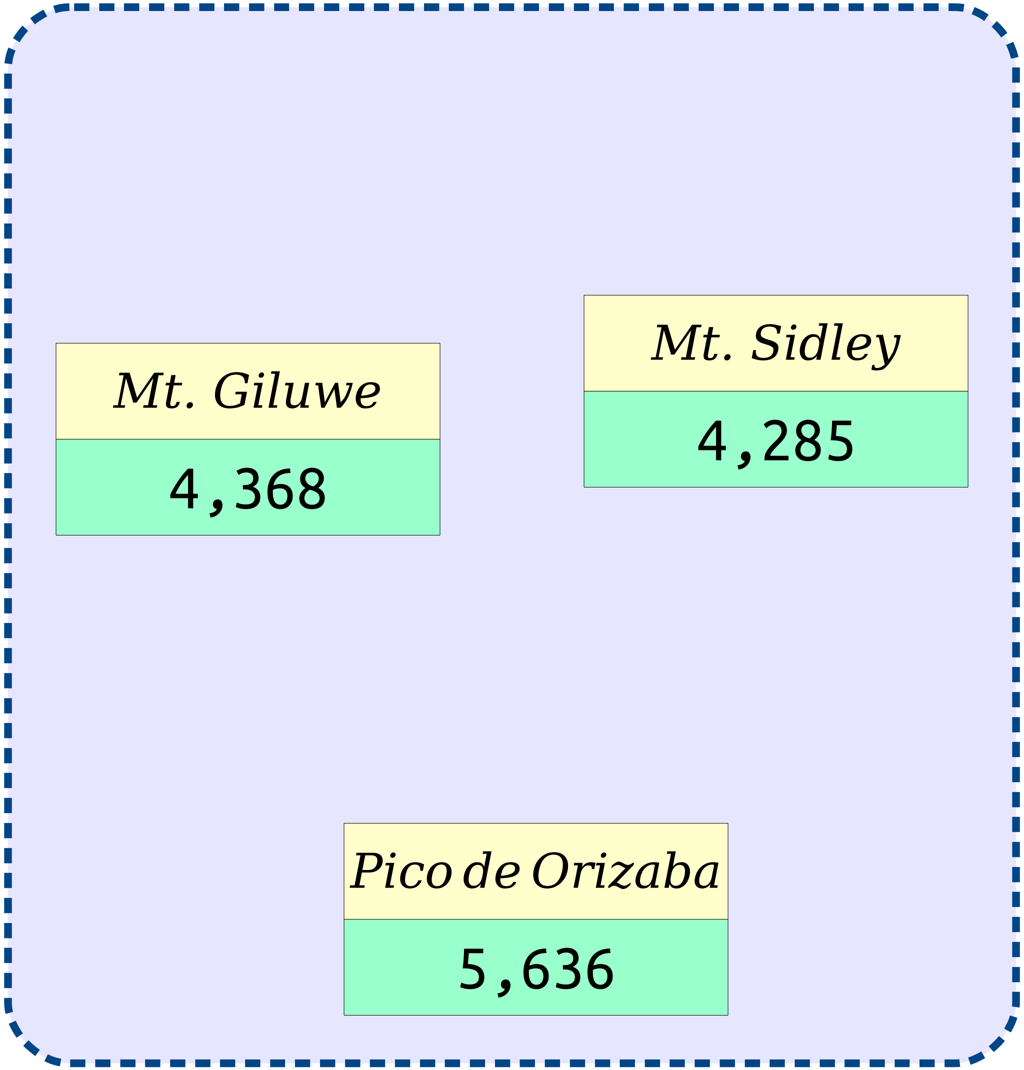
Priority Queues

- 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.



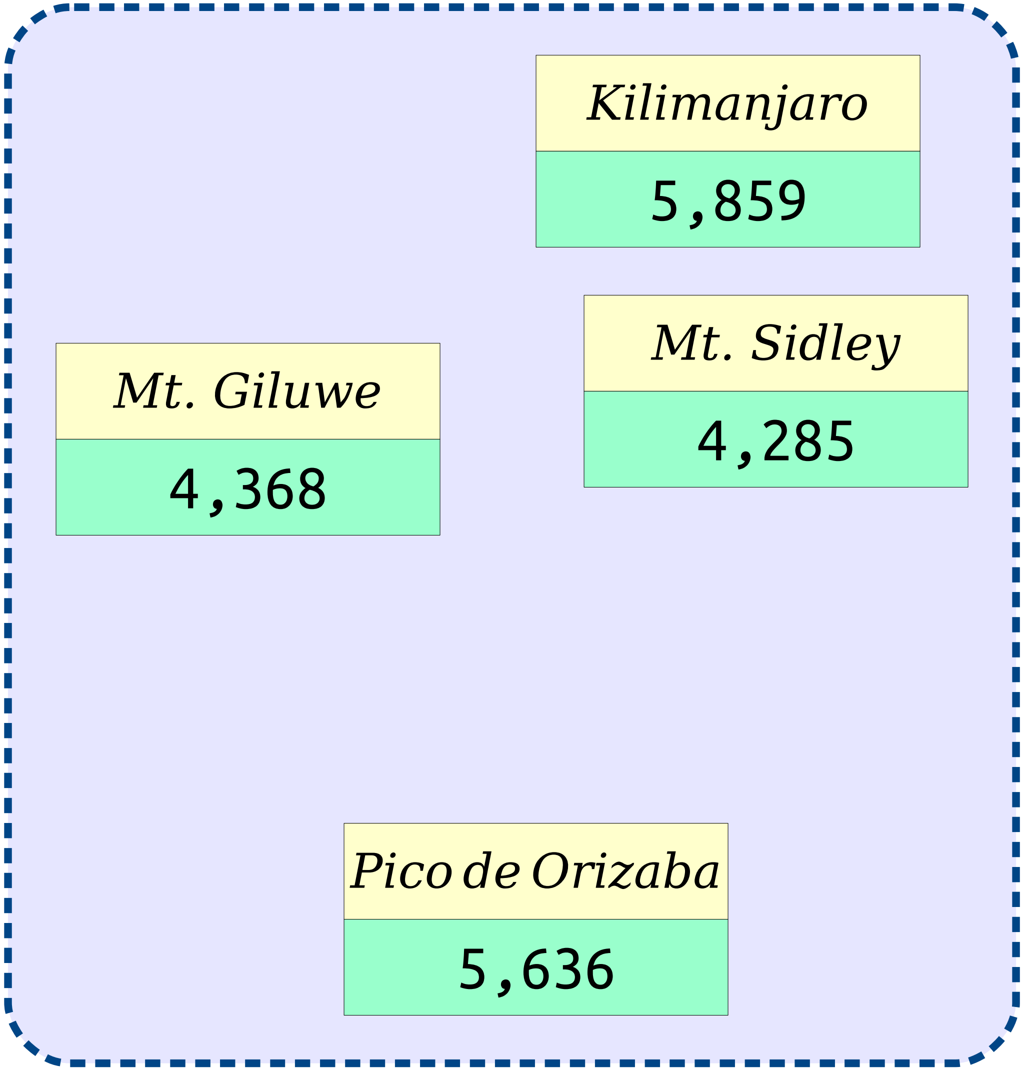
Priority Queues

- 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.



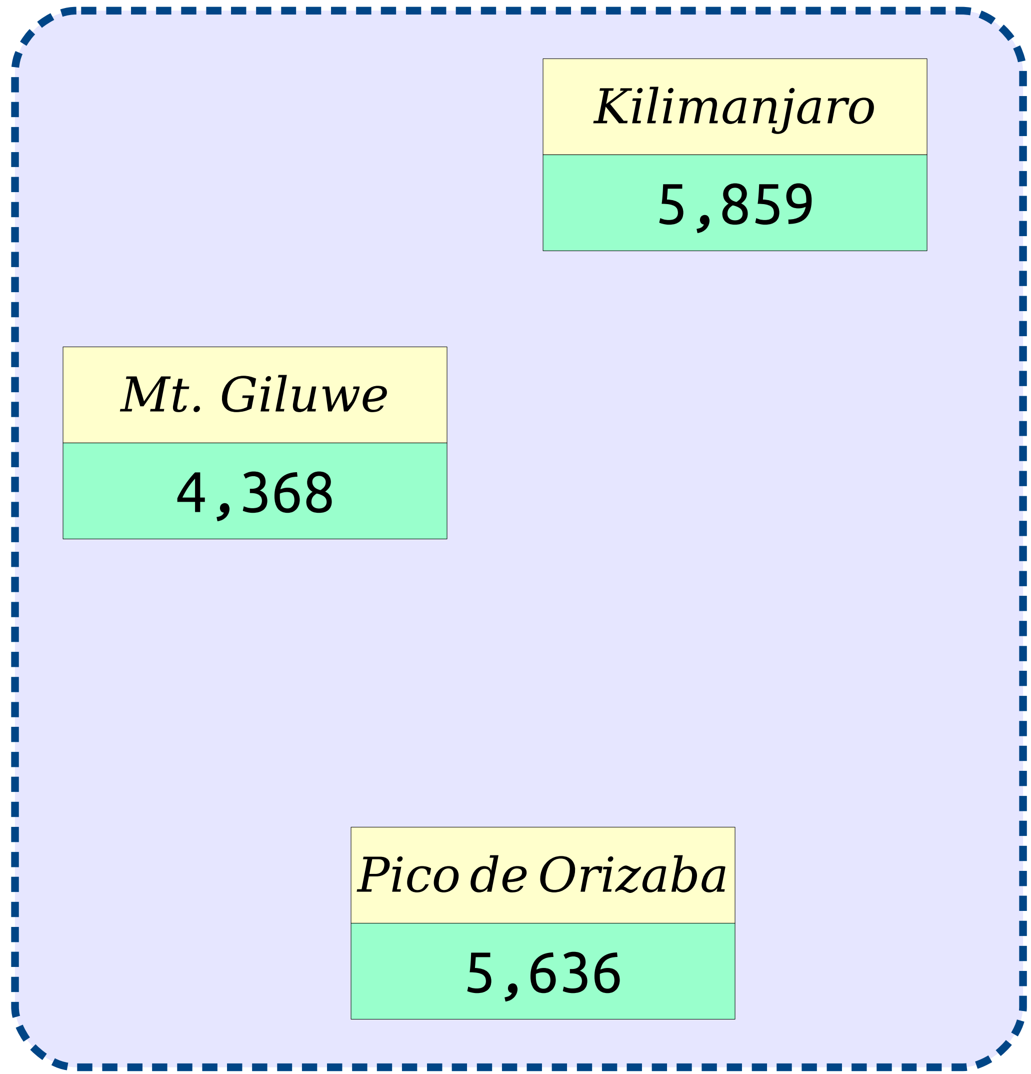
Priority Queues

- 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.



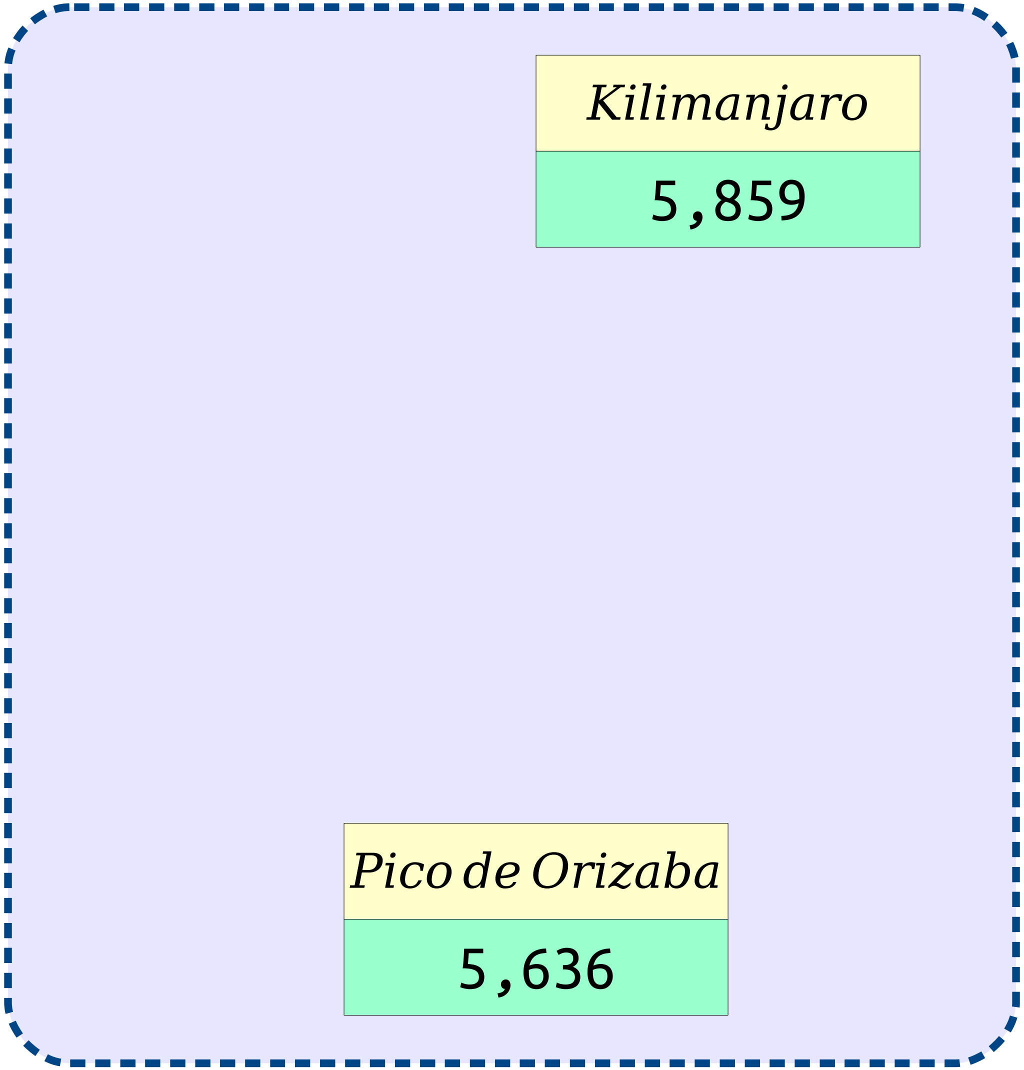
Priority Queues

- 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.



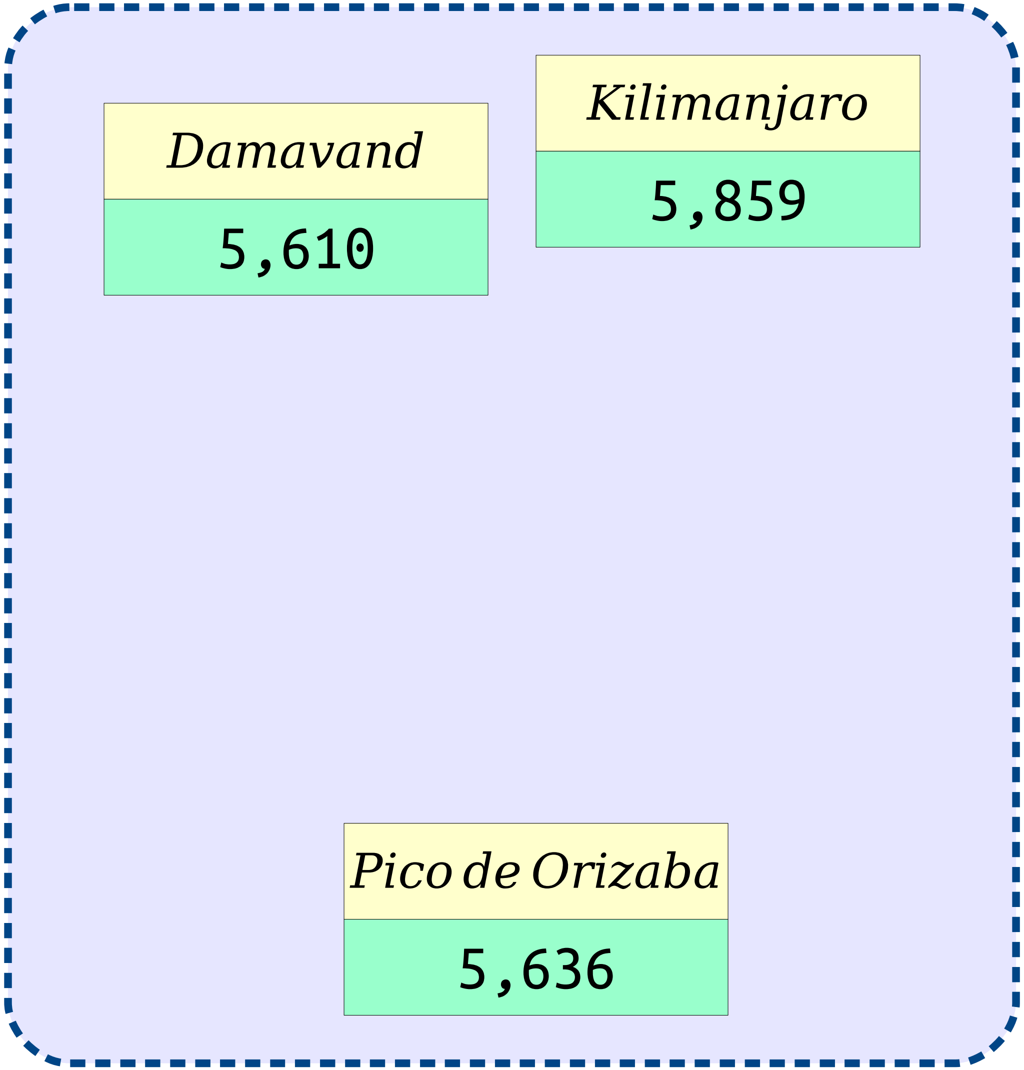
Priority Queues

- 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.



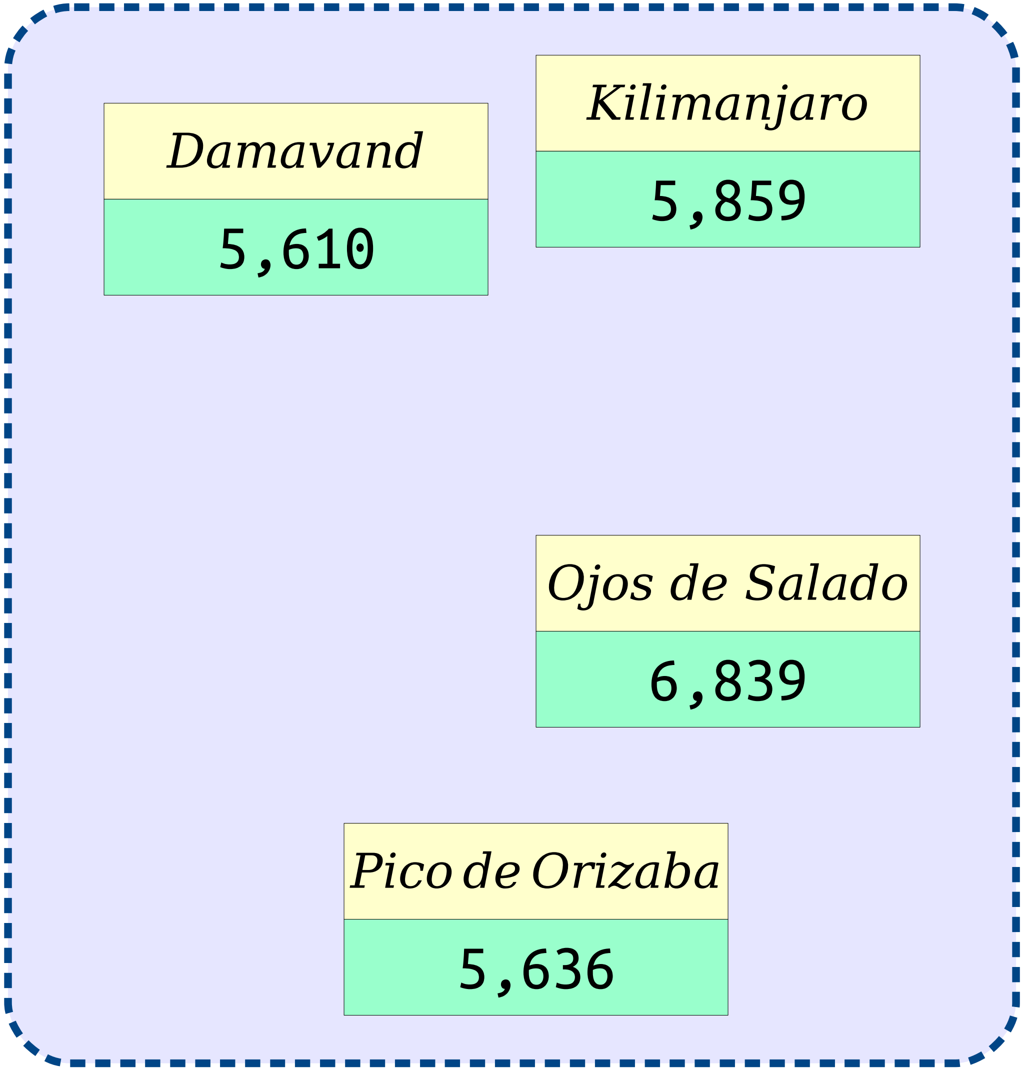
Priority Queues

- 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.



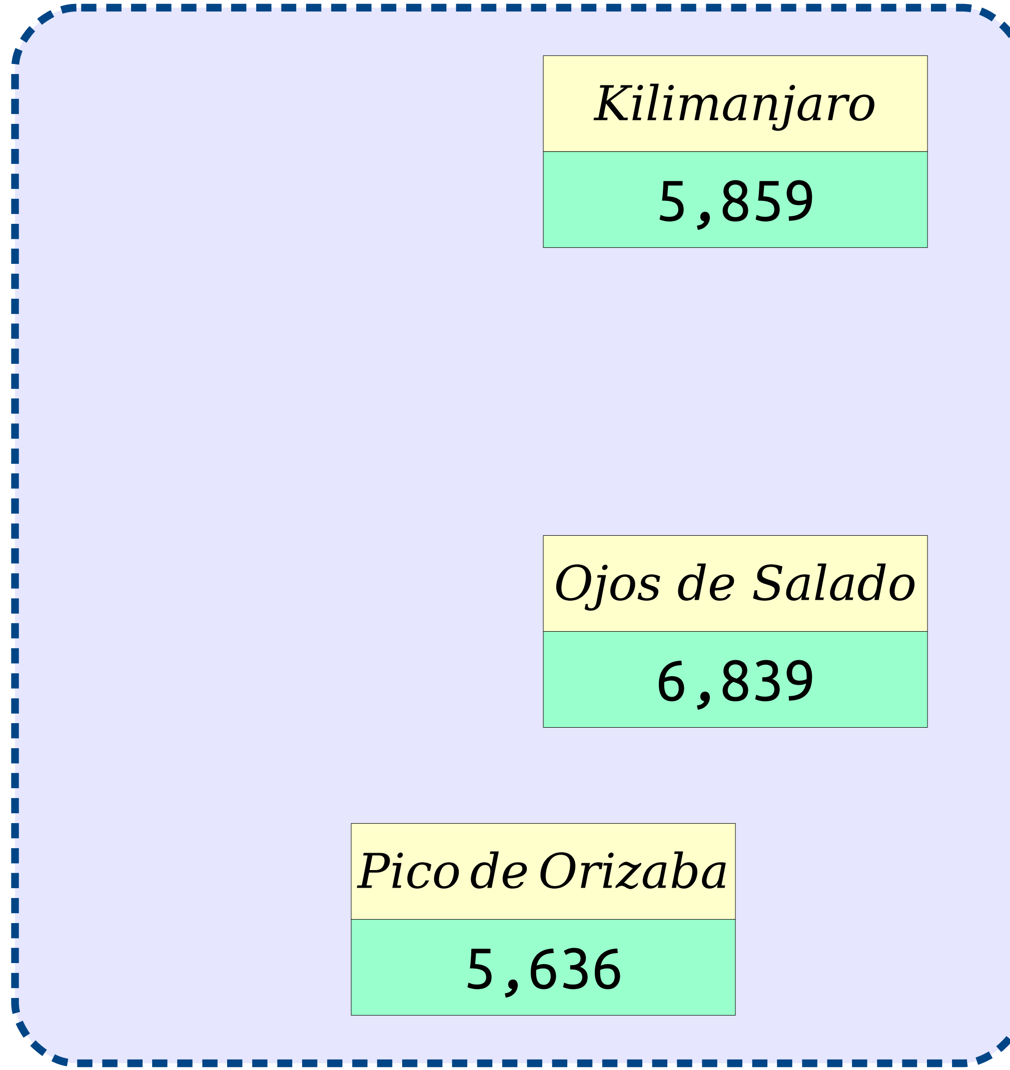
Priority Queues

- 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.



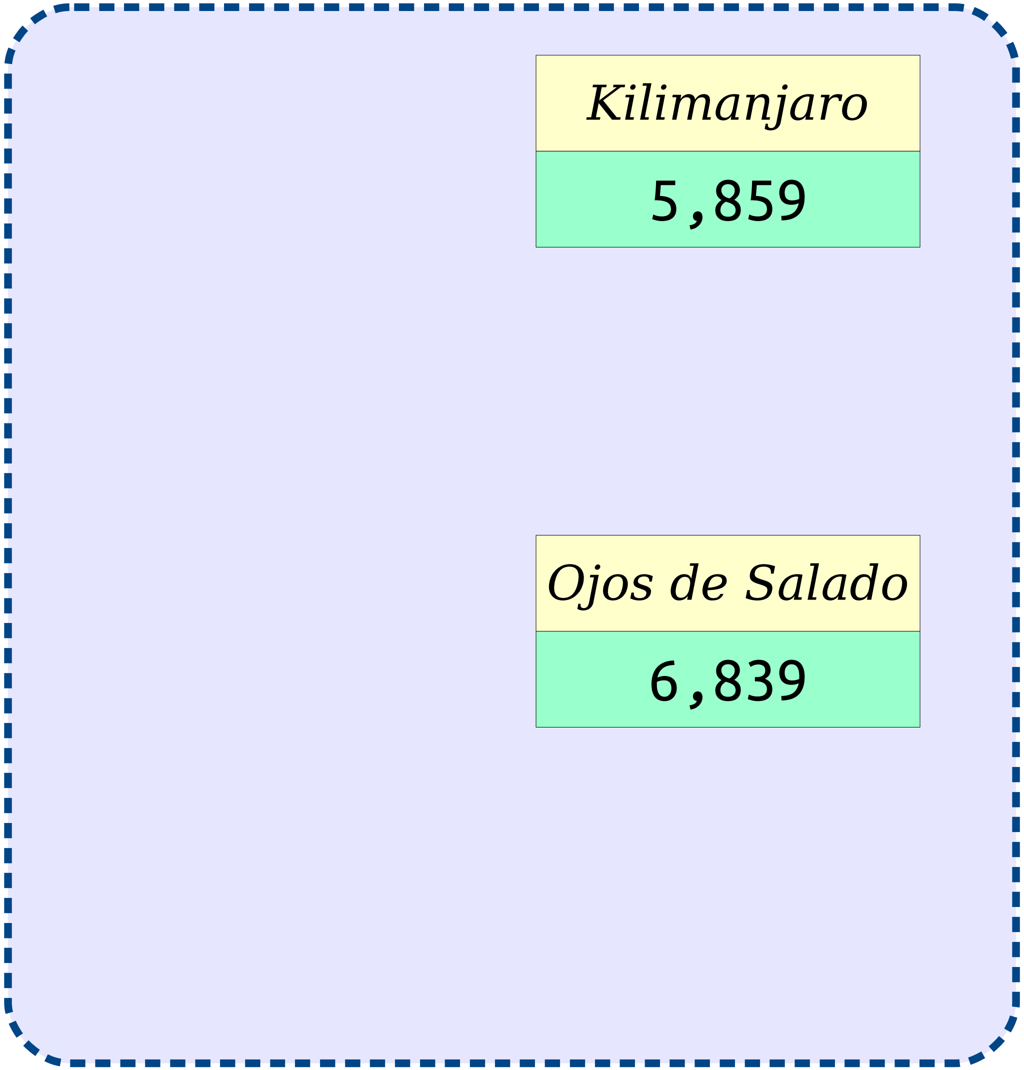
Priority Queues

- 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.



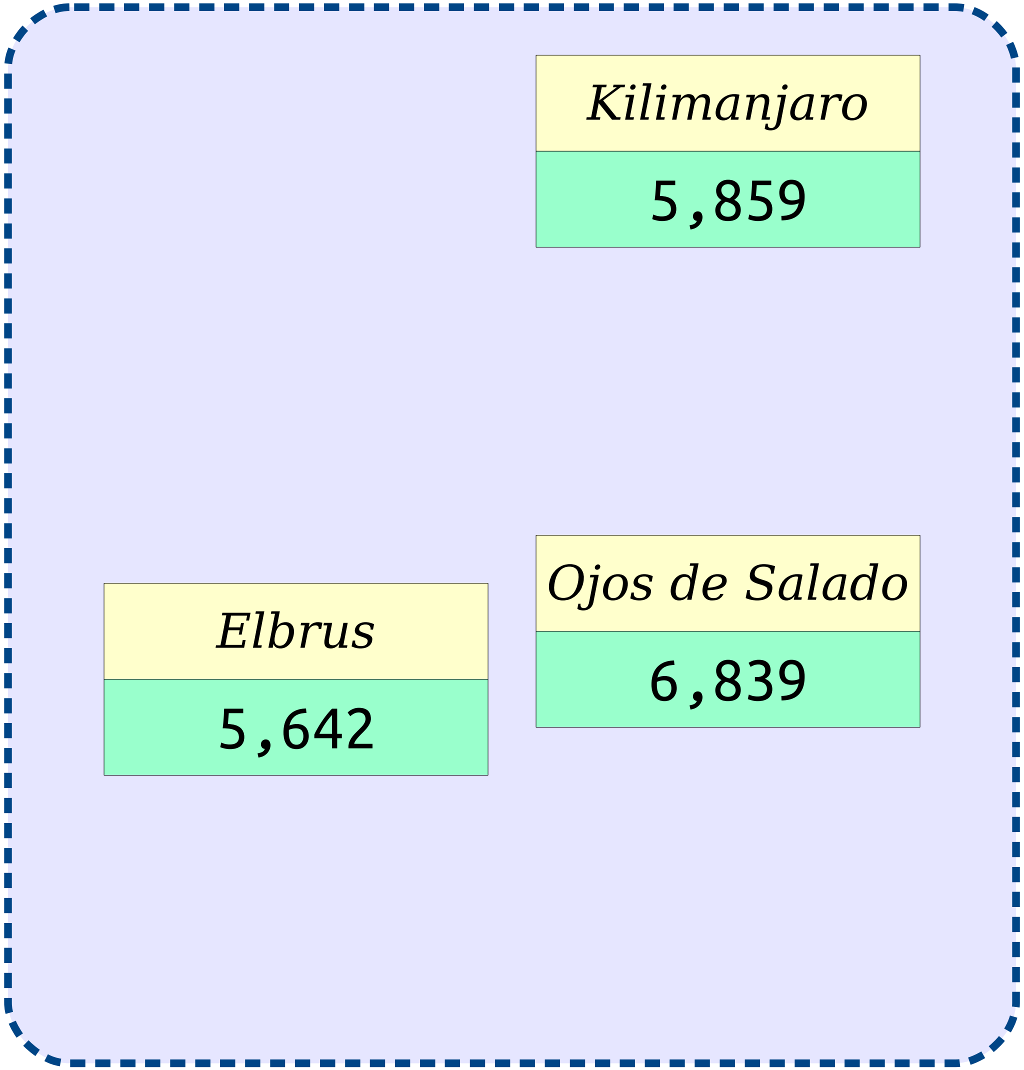
Priority Queues

- 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.



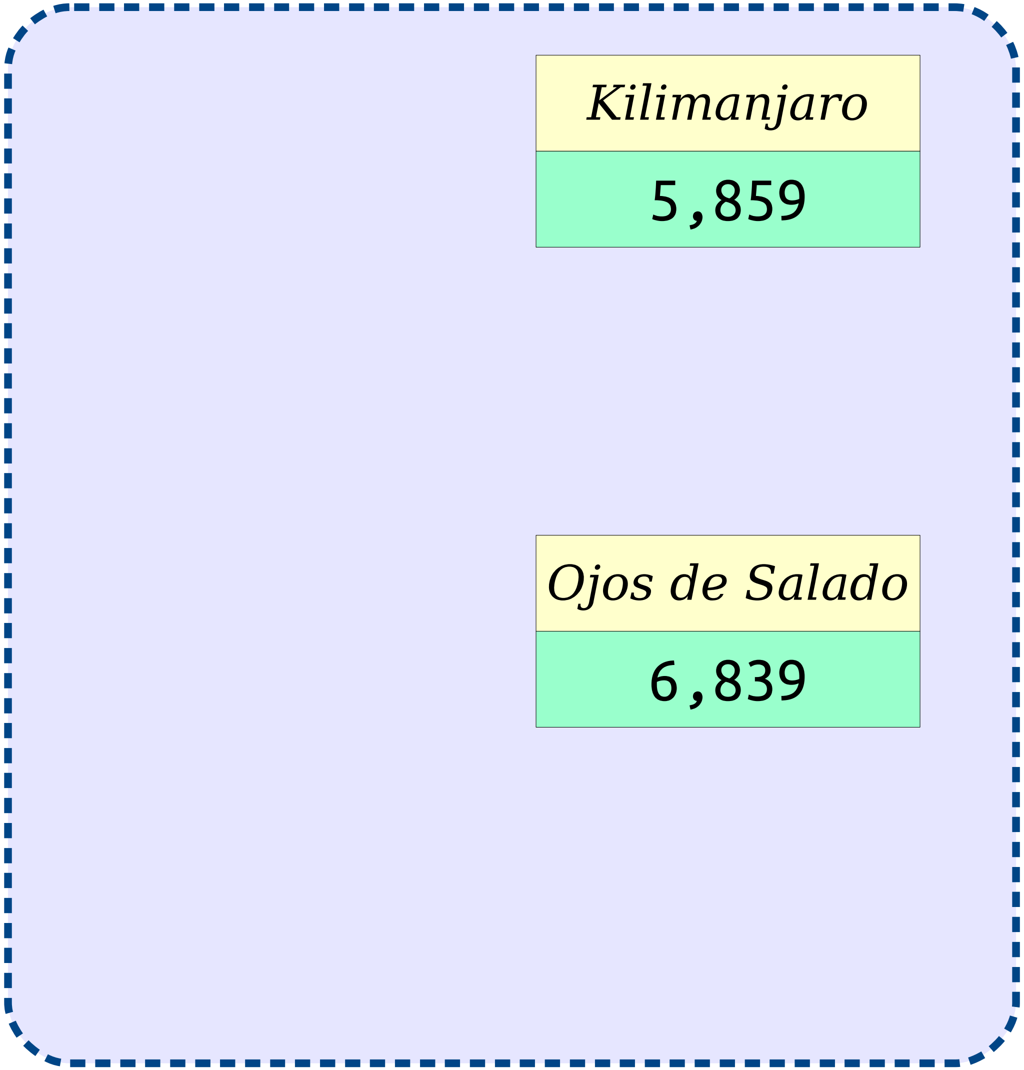
Priority Queues

- 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.



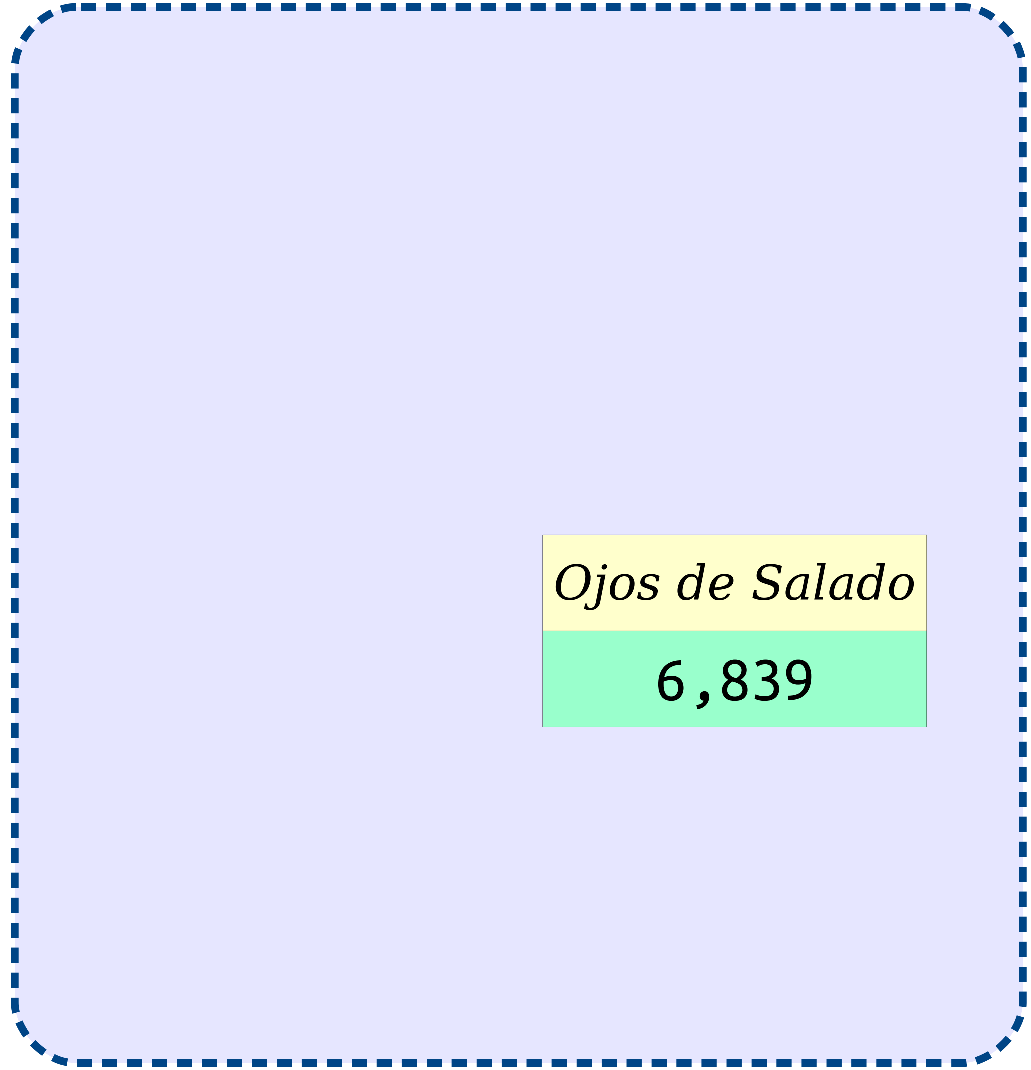
Priority Queues

- 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.

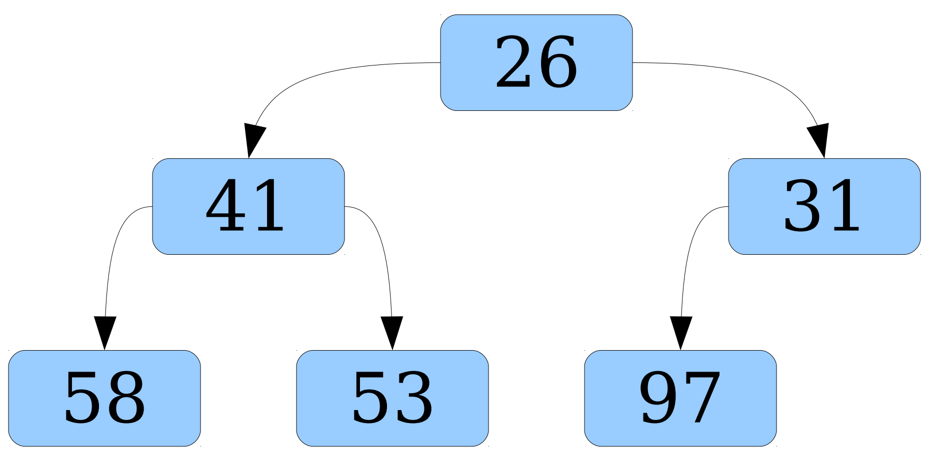


Priority Queues

- 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.

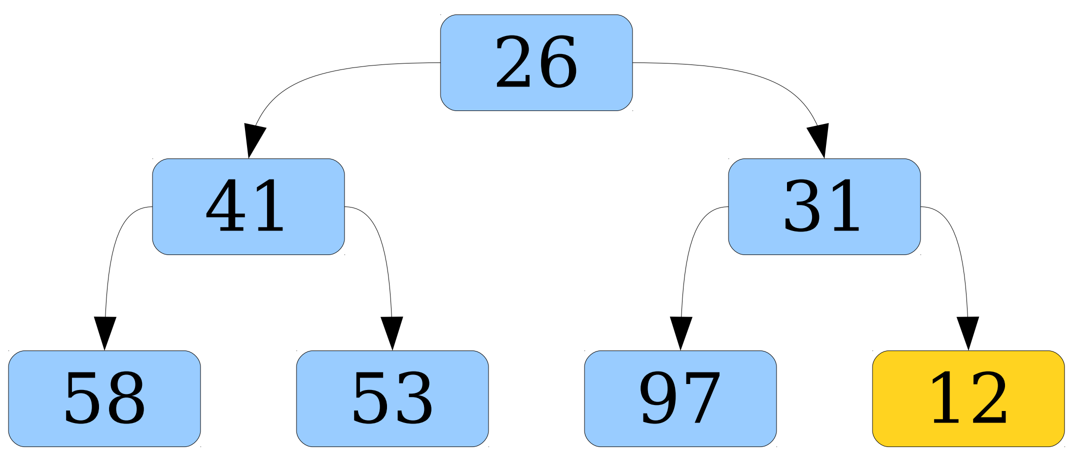
Binary Heaps

- 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!



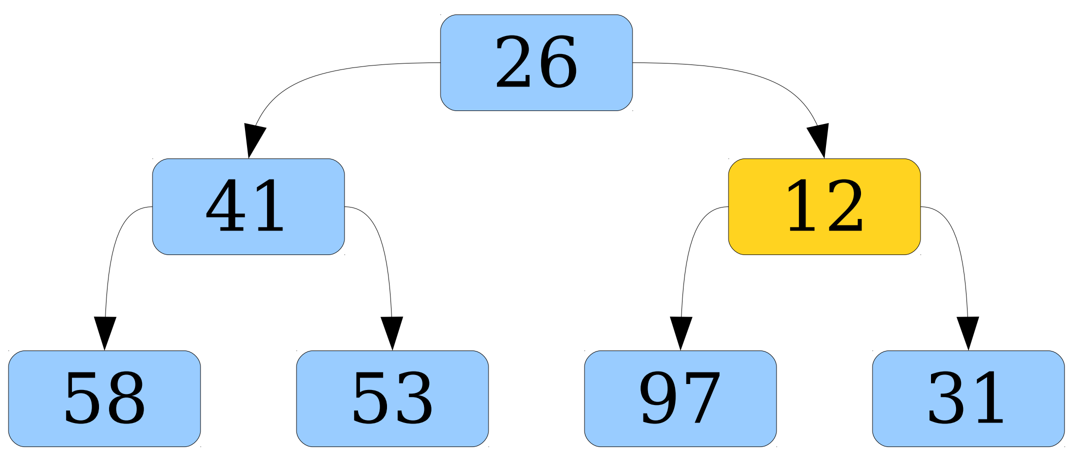
Binary Heaps

- 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!



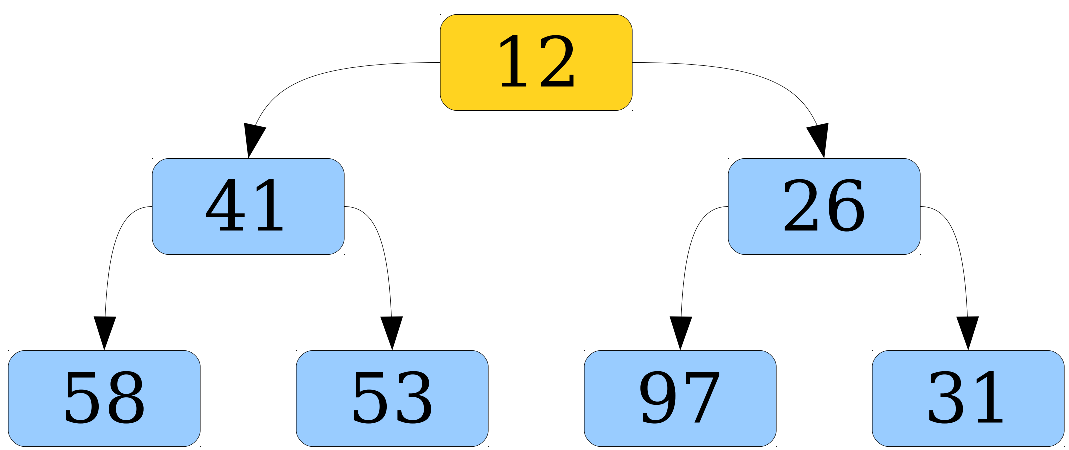
Binary Heaps

- 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!



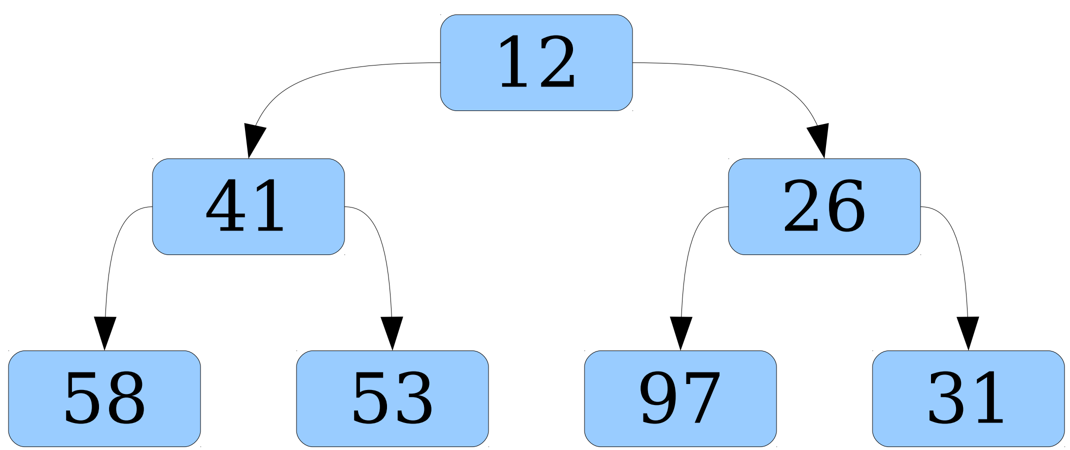
Binary Heaps

- 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!



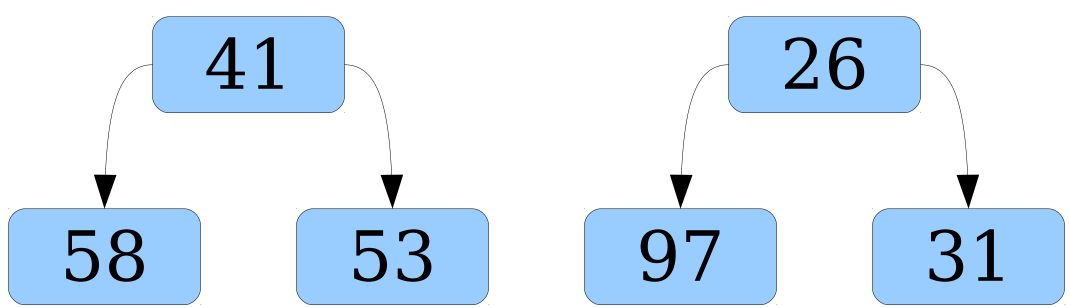
Binary Heaps

- 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!



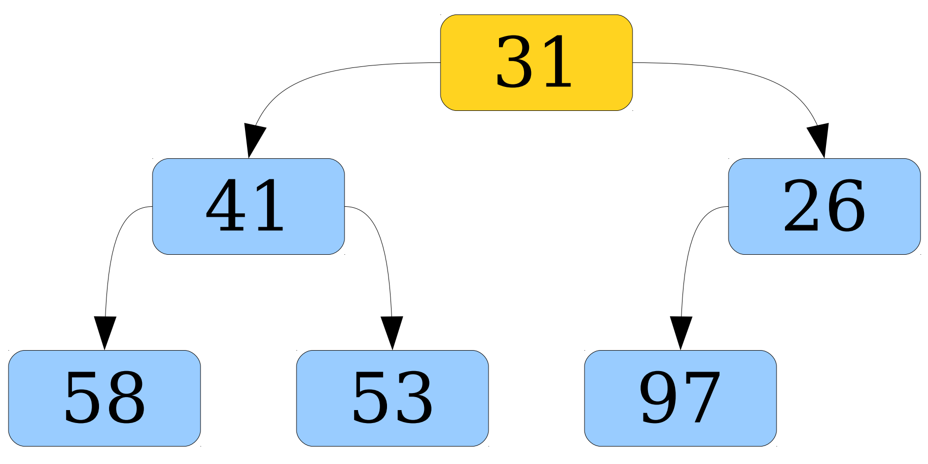
Binary Heaps

- 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!



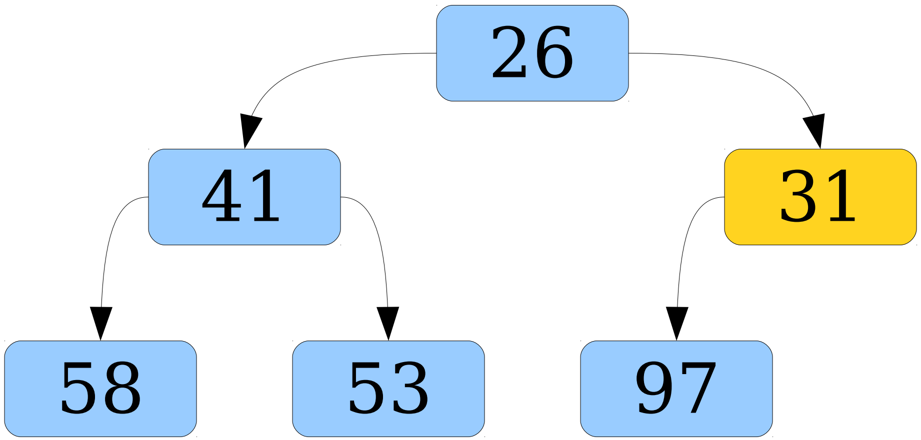
Binary Heaps

- 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!



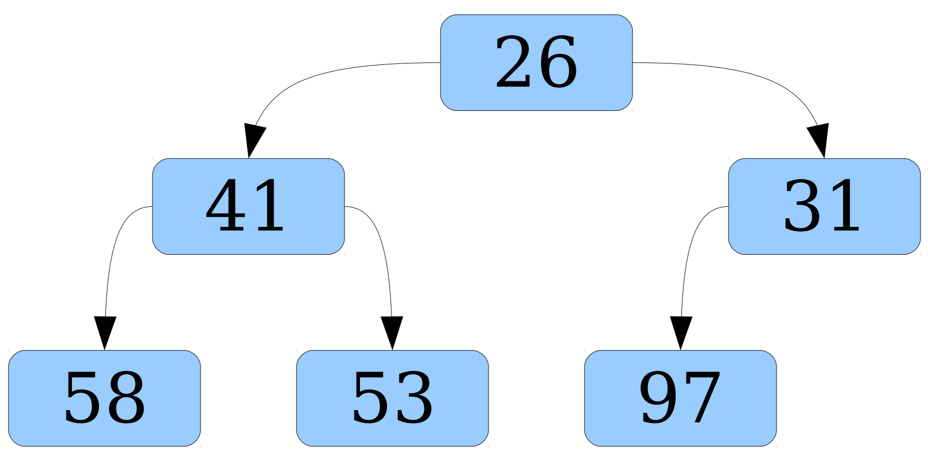
Binary Heaps

- 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!



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!

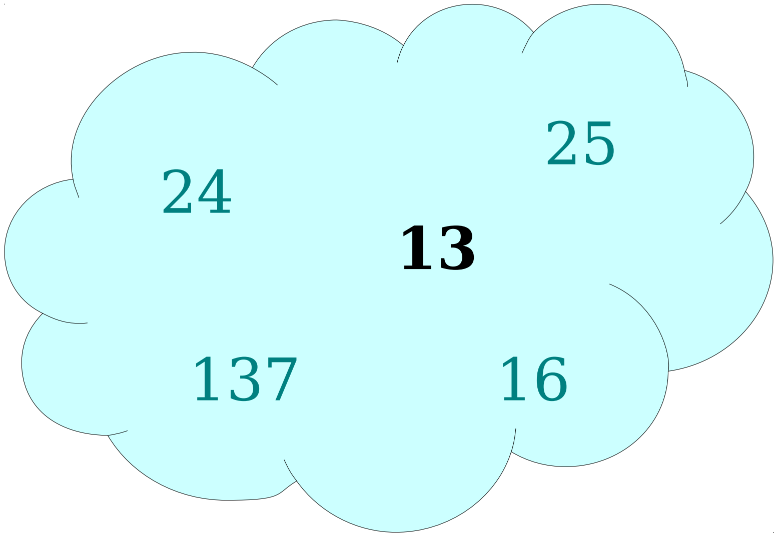


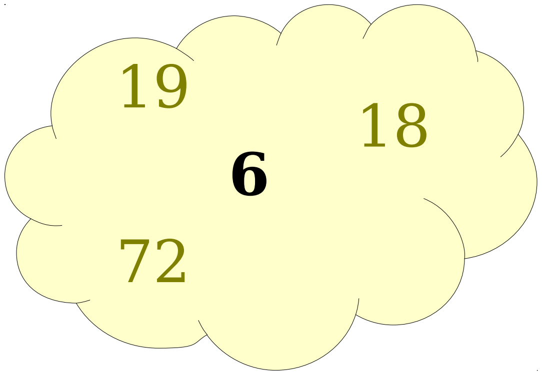
Priority Queues in Practice

- 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. *(Cheriton-Tarjan MST algorithm.)*
  - *pq*.***decrease-key***(v, k'): Given a pointer to element v already in the queue, lower its key to have new value k'. *(Dijkstra's algorithm, Prim's algorithm, Stoer-Wagner algorithm for global min cut.)*
  - *pq*.***add-to-all***(Δk): Add Δk to the keys of each element in the priority queue, typically used with ***meld***. *(Chu-Edmonds-Liu algorithm for directed MST.)*
- In lecture, we'll cover binomial heaps to efficiently support ***meld*** and Fibonacci heaps to efficiently support ***meld*** and ***decrease-key***.
- You'll design a priority queue supporting efficient ***meld*** and ***add-to-all*** on the next problem set.

Meldable Priority Queues

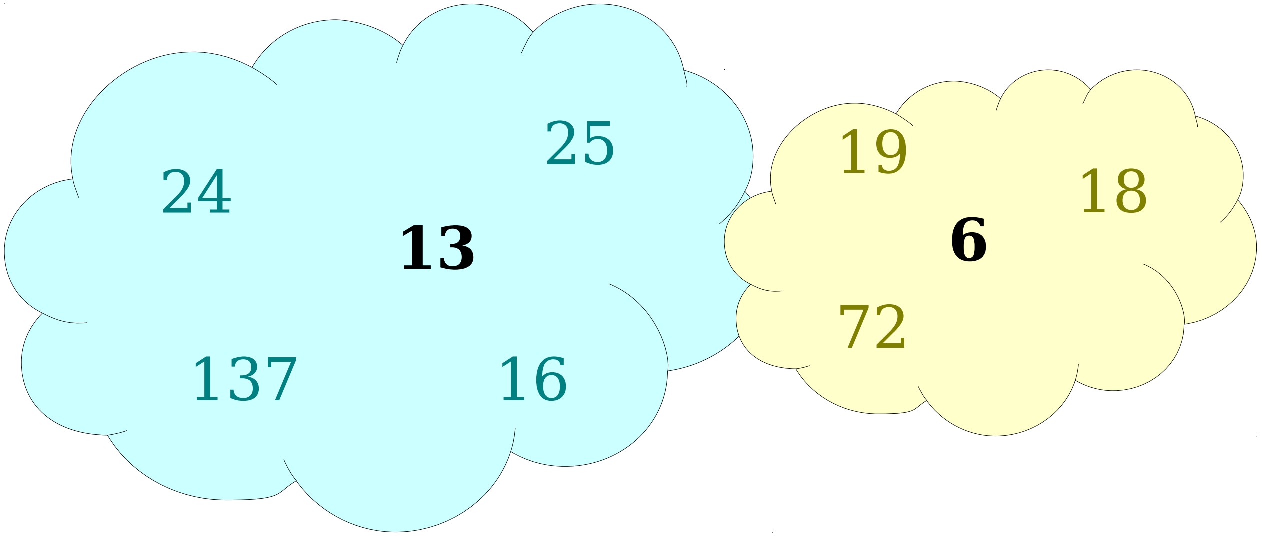
- A priority queue supporting the ***meld*** operation is called a ***meldable priority queue***.
- ***meld***(pq_1, pq_2) destructively modifies pq_1 and pq_2 and produces a new priority queue containing all elements of pq_1 and pq_2.





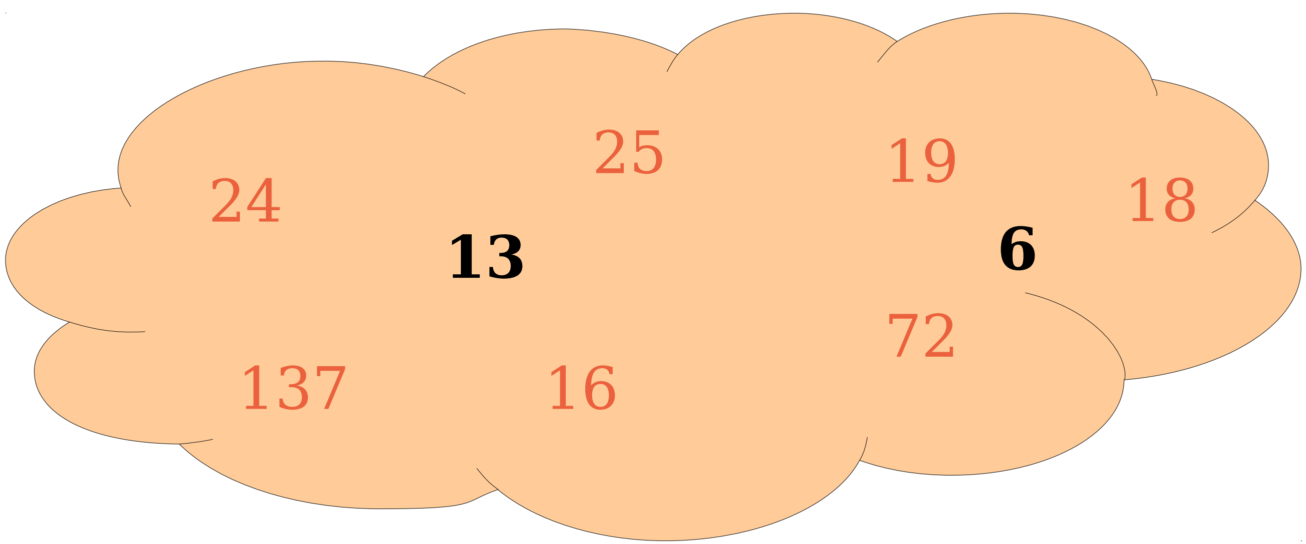
Meldable Priority Queues

- 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.



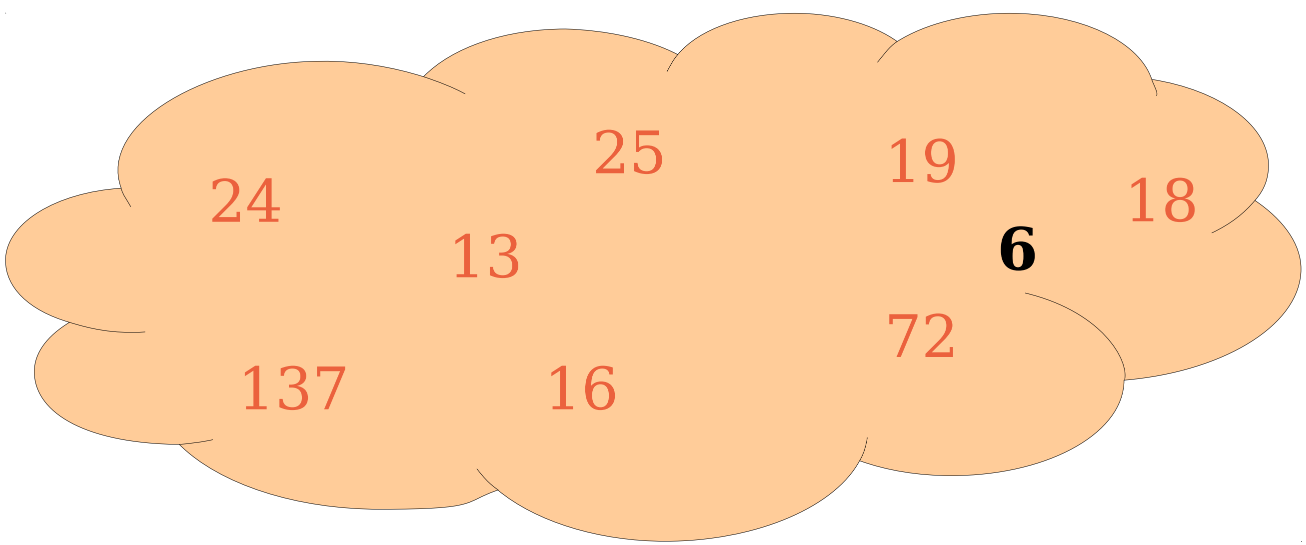
Meldable Priority Queues

- 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.



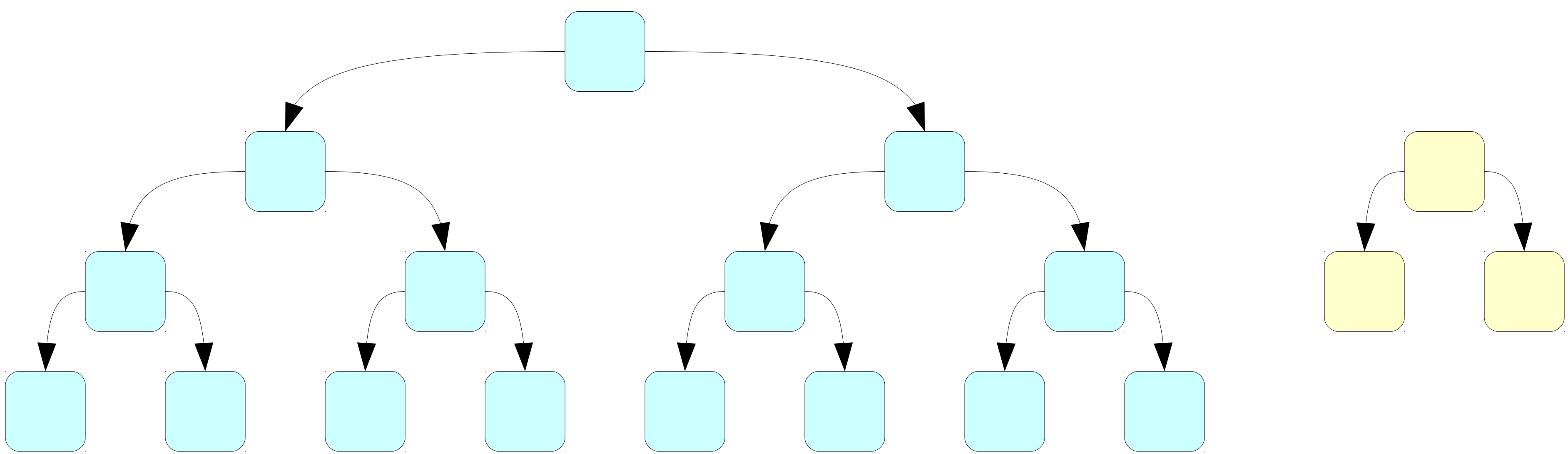
Meldable Priority Queues

- 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.



Binomial Heaps

- The ***binomial heap*** is an priority queue data structure that supports efficient melding.
- We'll study binomial heaps for several reasons:
  - They’re based on a *beautiful* intuition that’s totally different than that for binary heaps.
  - They’re used as a building block in other data structures (Fibonacci heaps, soft heaps, etc.)
  - They’re a great testbed for our topics from amortized analysis.

Supporting Efficient Melding

The Intuition: ***Binary Arithmetic***

Adding Binary Numbers

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

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 $\Theta(\max\{\log m, \log n\})$.

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

Adding Binary Numbers

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

	1	0	1	1	0
+		1	1	1	1

Adding Binary Numbers

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

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

Adding Binary Numbers

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

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

Adding Binary Numbers

- Given the binary representations of two numbers *n* and *m*, we can add those numbers in time $\Theta(\max\{\log m, \log n\})$.

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

Adding Binary Numbers

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

			1		
	1	0	1	1	0
+		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 $\Theta(\max\{\log m, \log n\})$.

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

Adding Binary Numbers

- Given the binary representations of two numbers *n* and *m*, we can add those numbers in time $\Theta(\max\{\log m, \log n\})$.

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

Adding Binary Numbers

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

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

Adding Binary Numbers

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

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

Adding Binary Numbers

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

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

Adding Binary Numbers

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

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

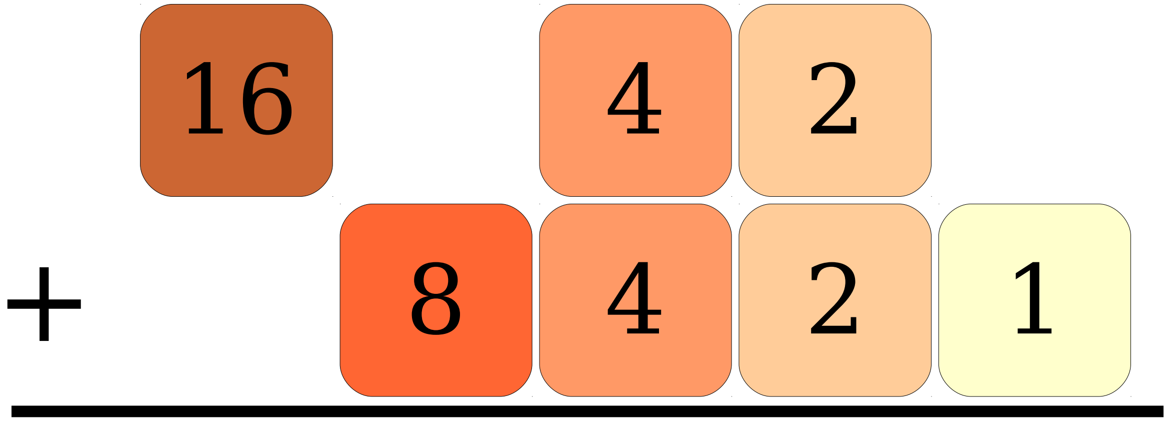
A Different Intuition

- 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

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

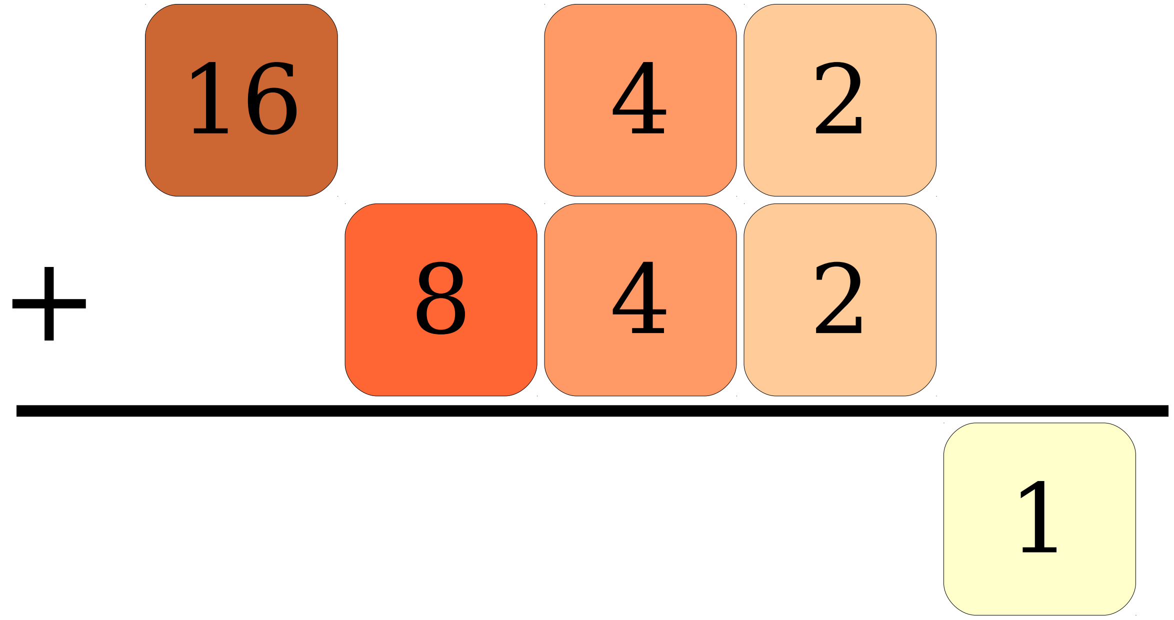
A Different Intuition

- 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



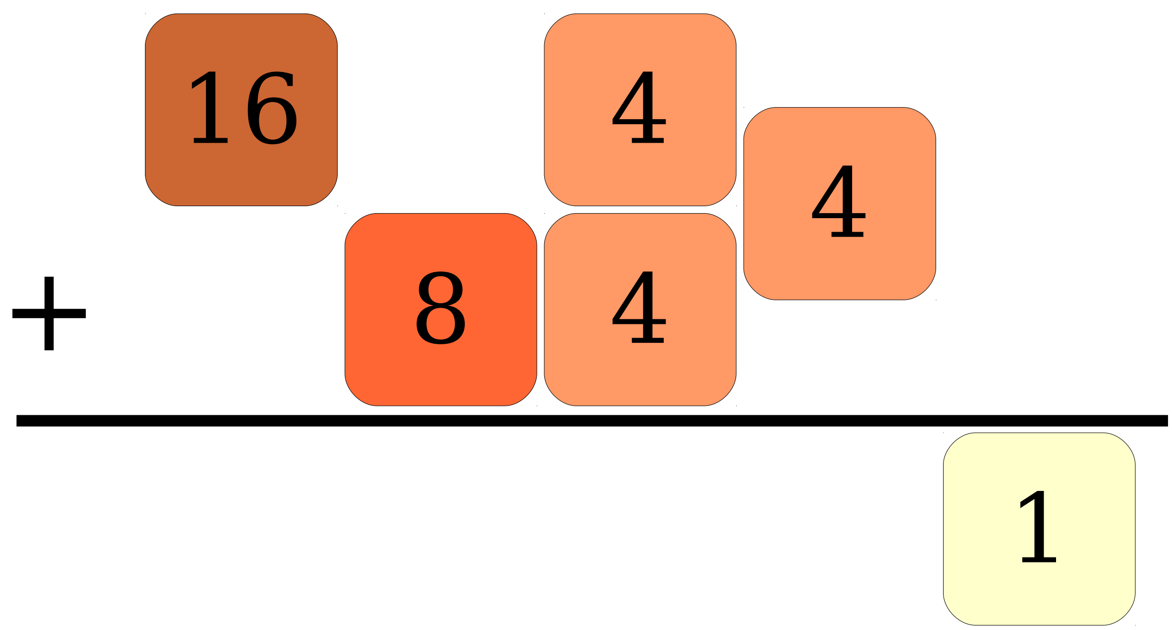
A Different Intuition

- 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



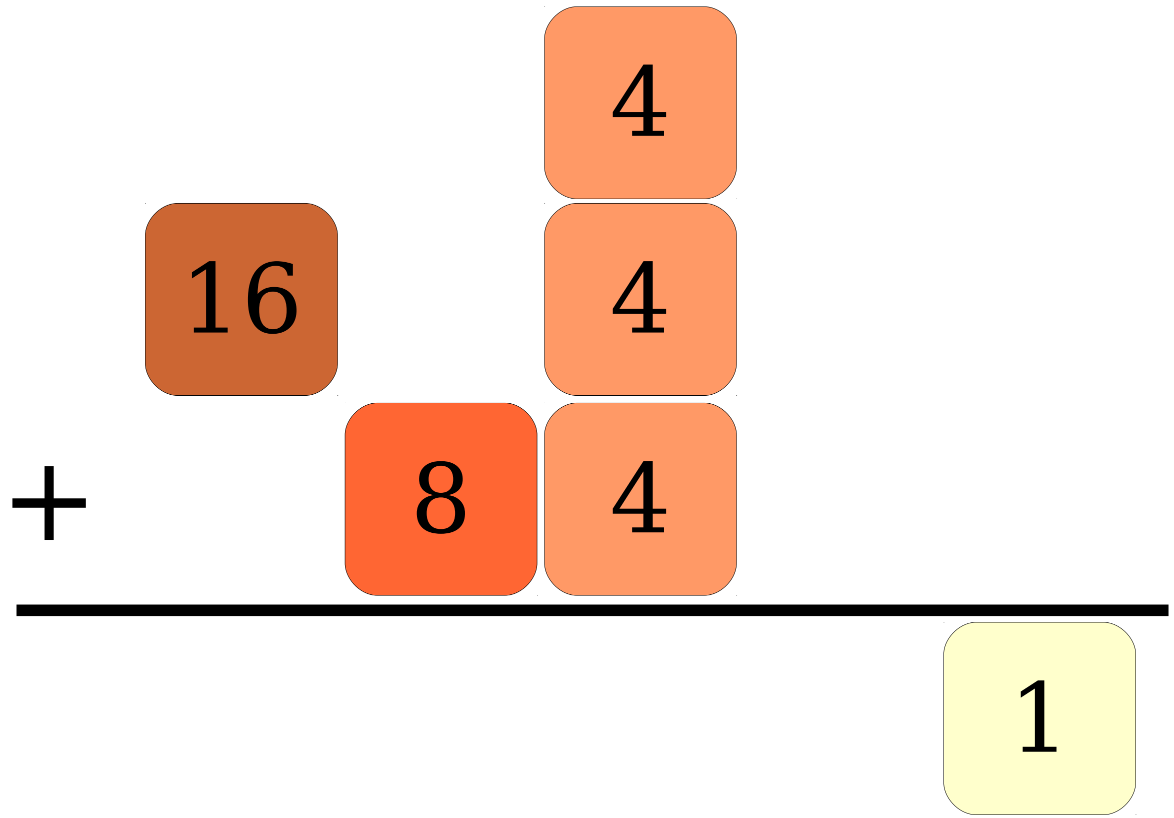
A Different Intuition

- 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



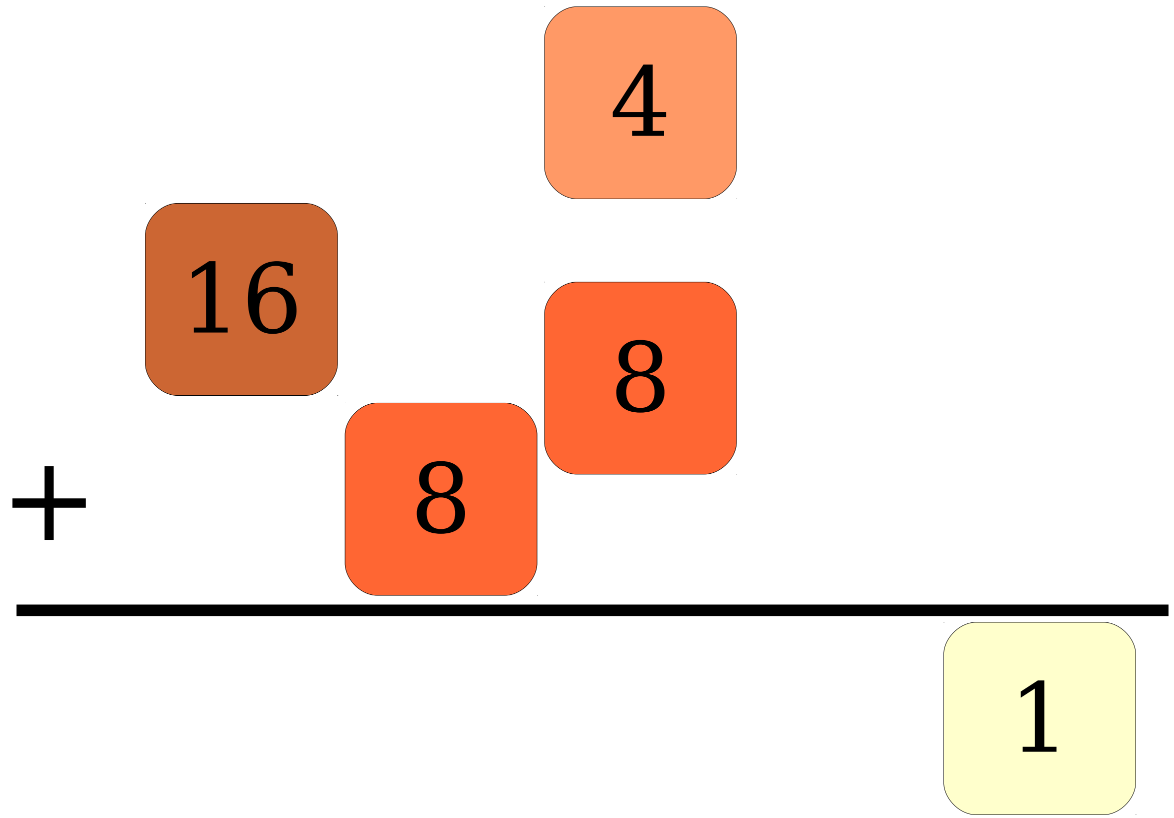
A Different Intuition

- 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



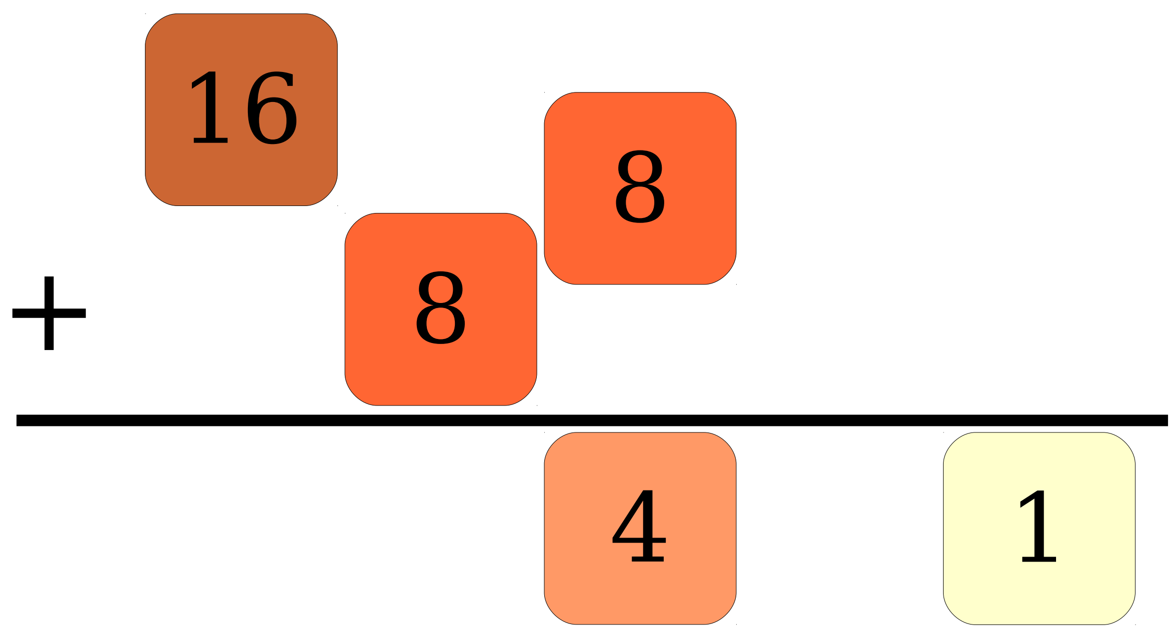
A Different Intuition

- 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



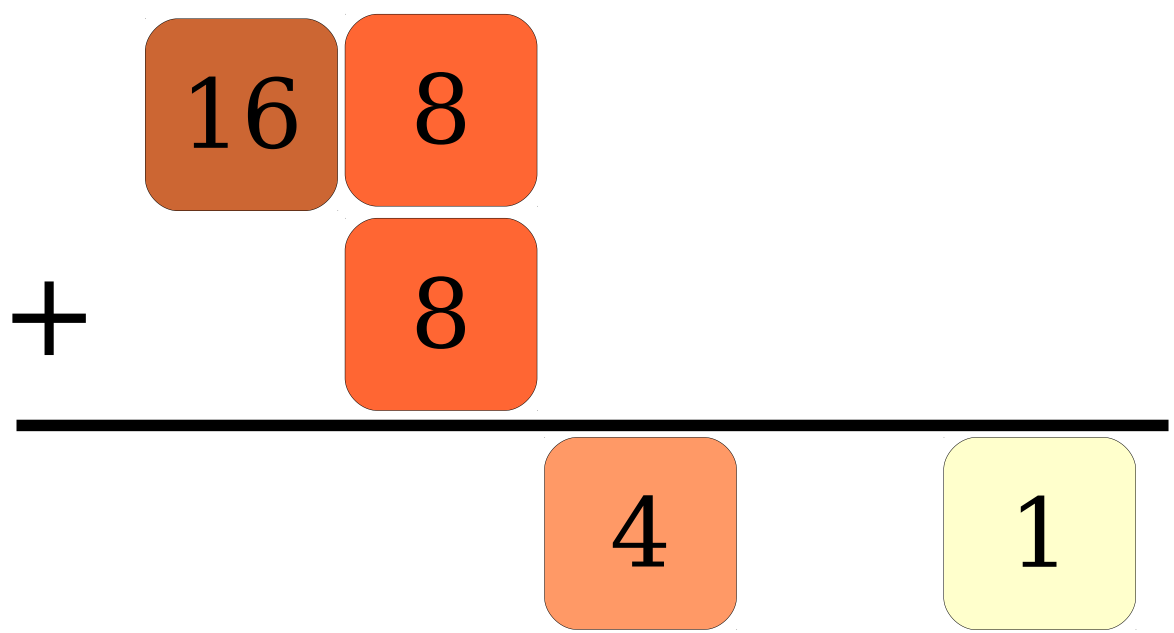
A Different Intuition

- 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



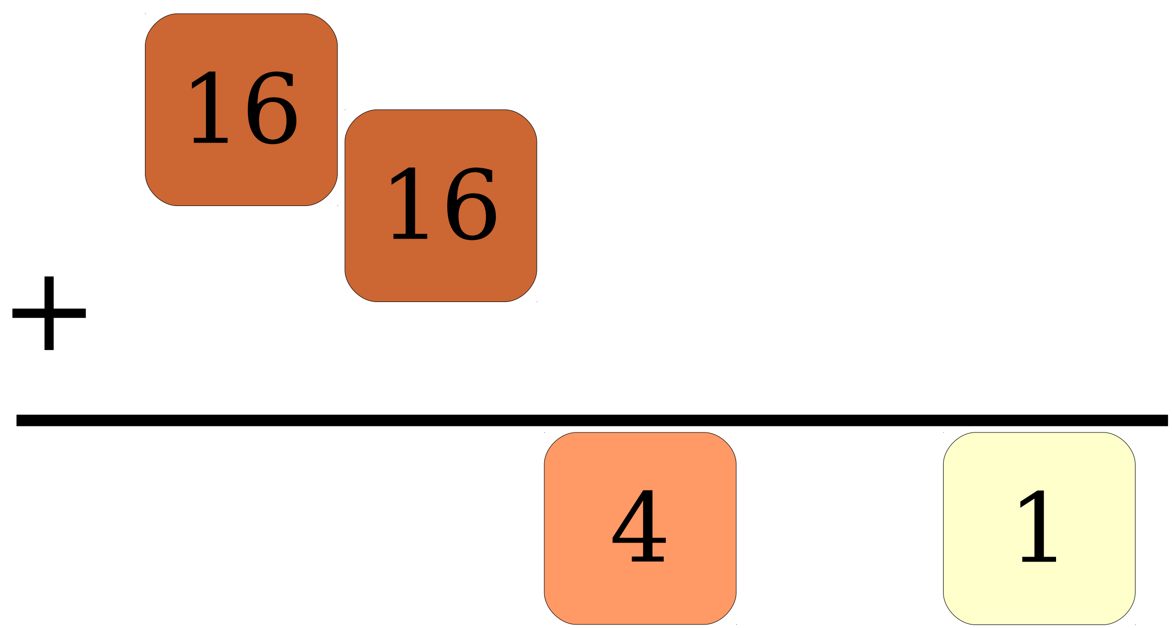
A Different Intuition

- 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



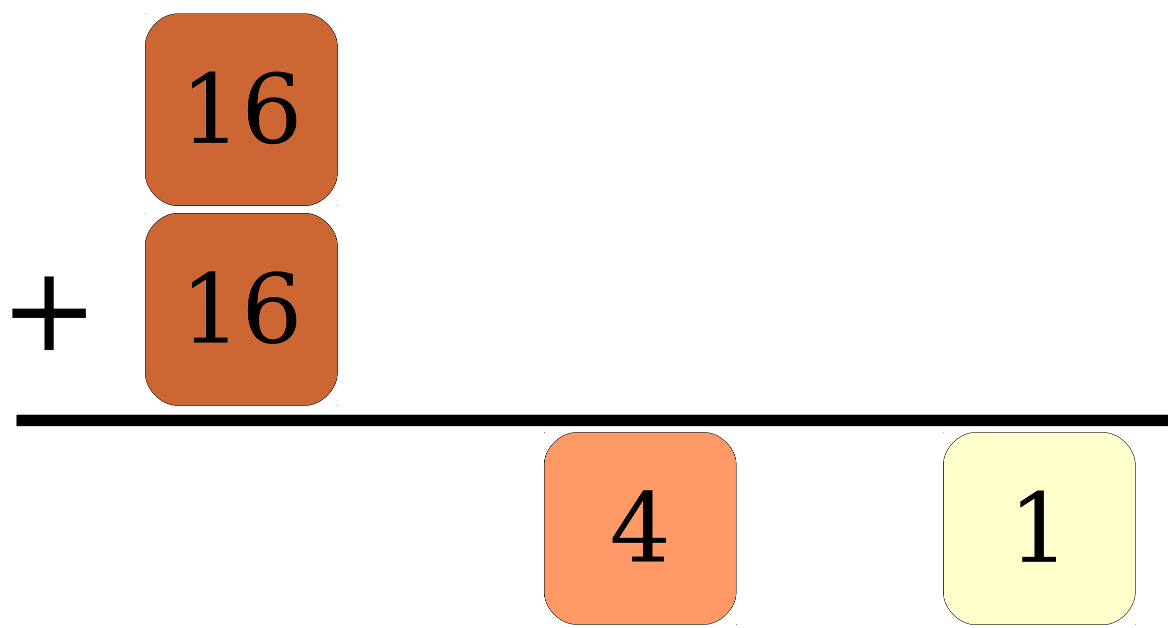
A Different Intuition

- 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



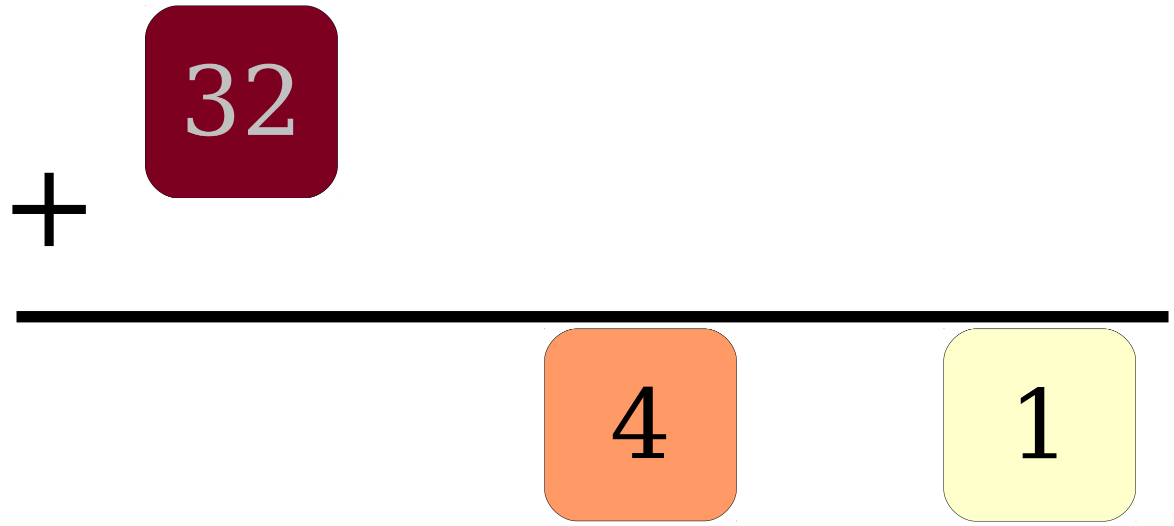
A Different Intuition

- 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



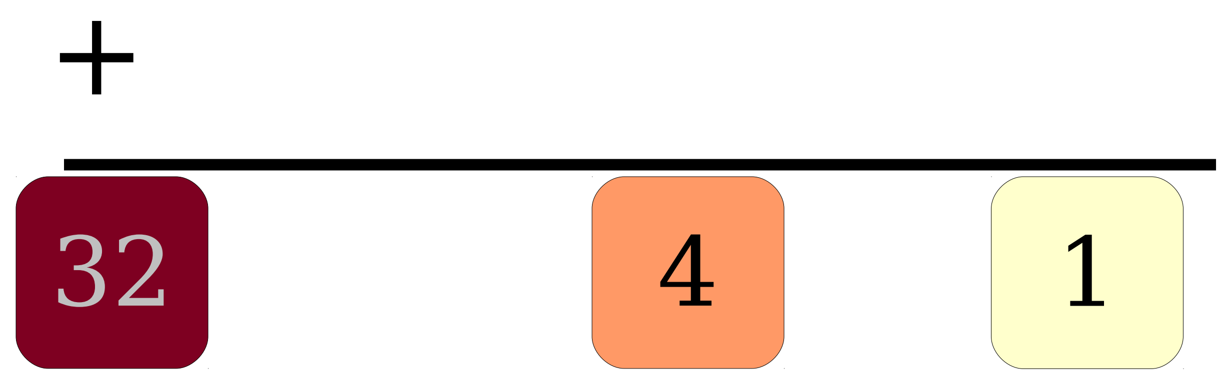
A Different Intuition

- 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



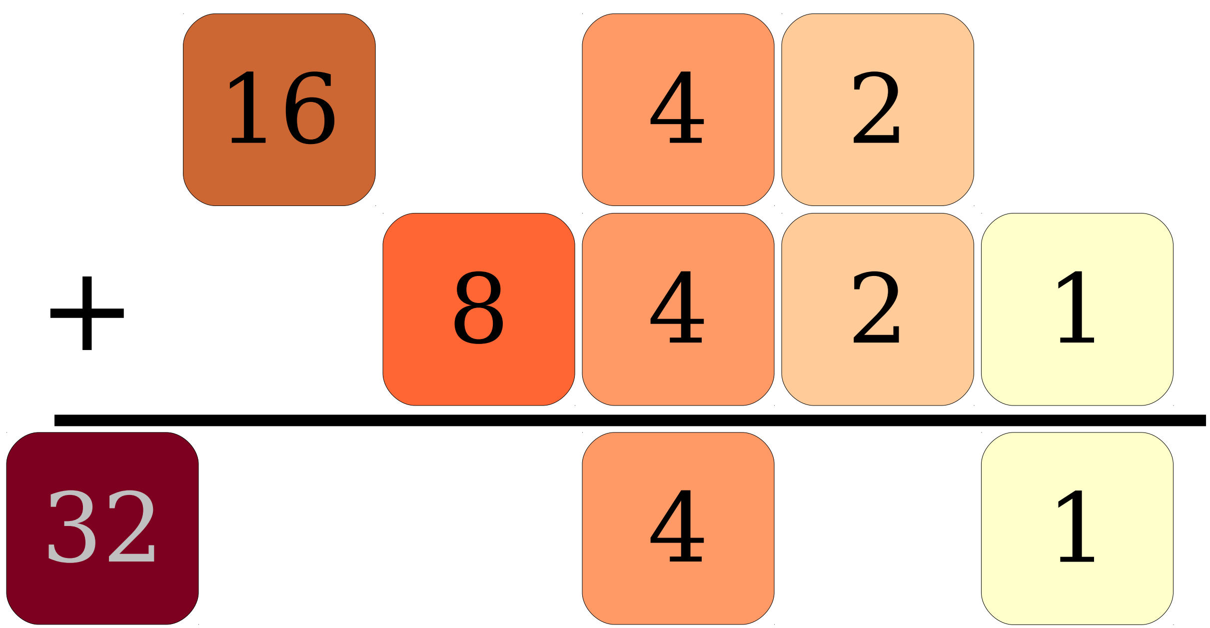
A Different Intuition

- 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



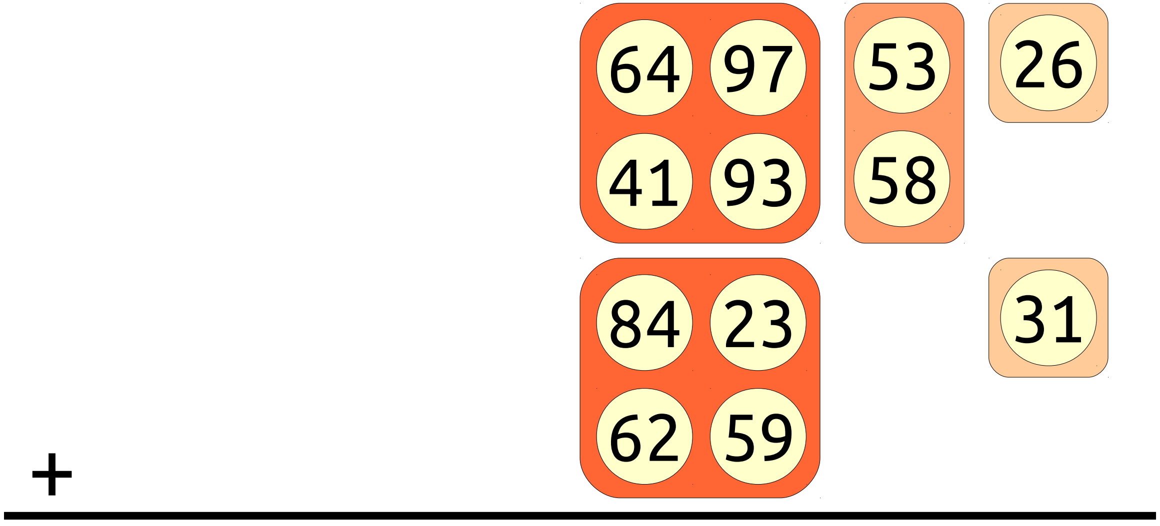
A Different Intuition

- 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



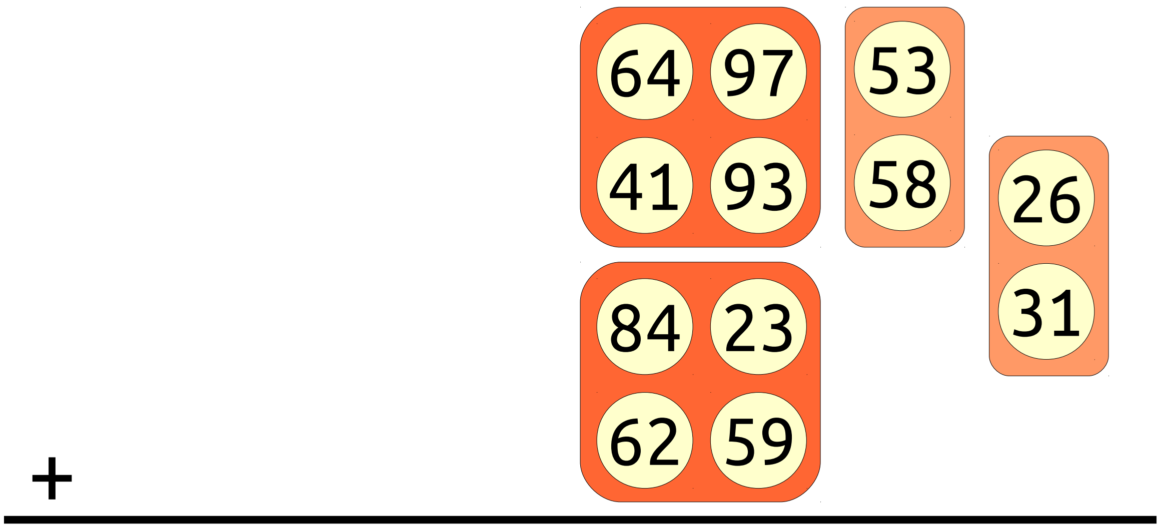
Building a Priority Queue

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



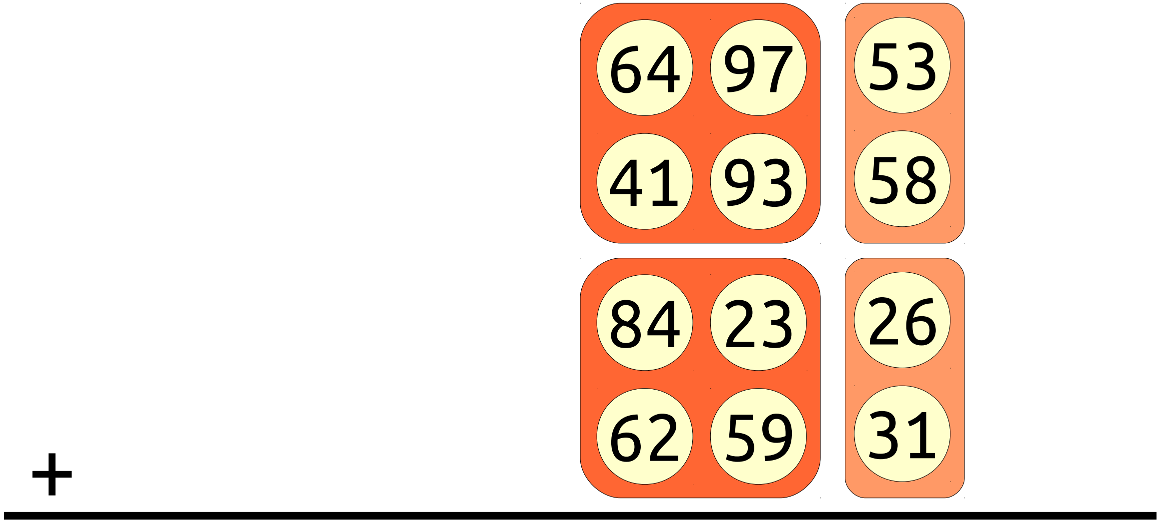
Building a Priority Queue

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



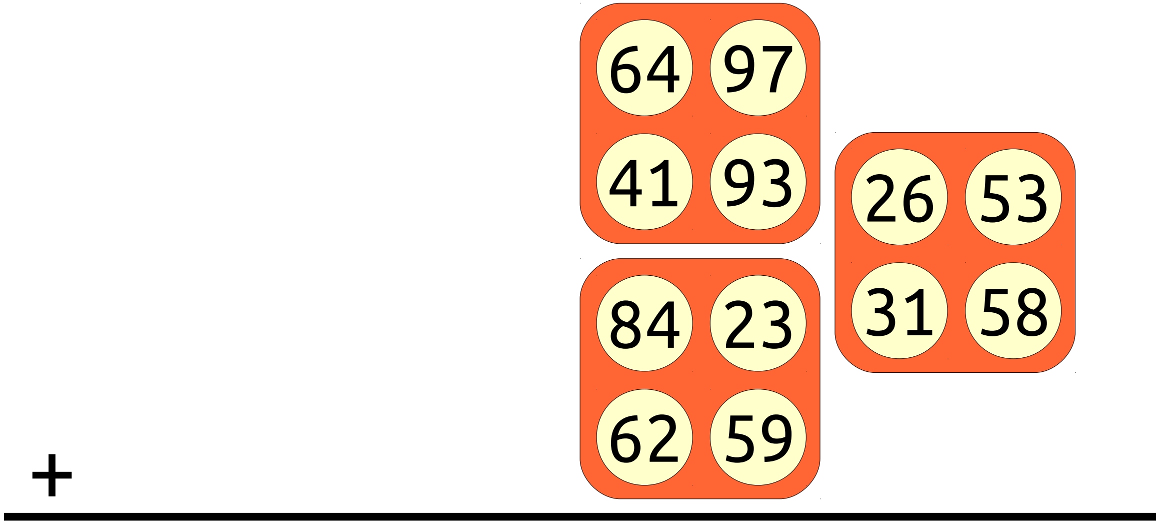
Building a Priority Queue

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



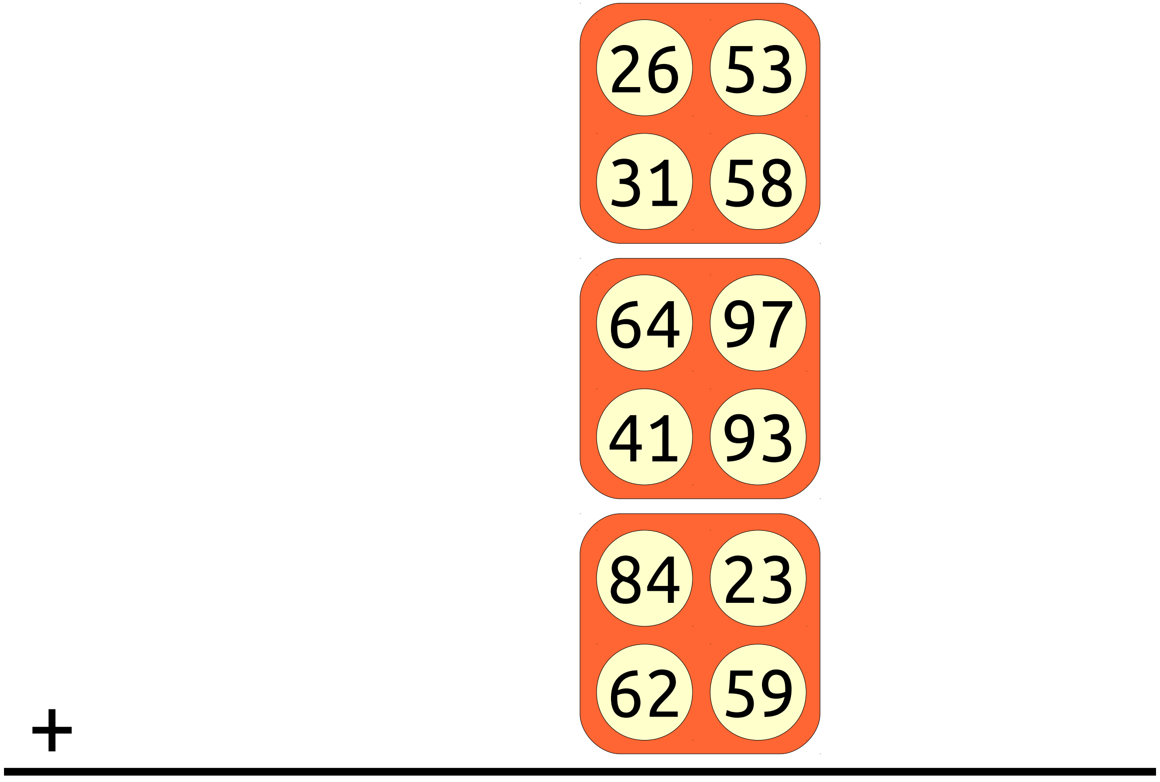
Building a Priority Queue

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



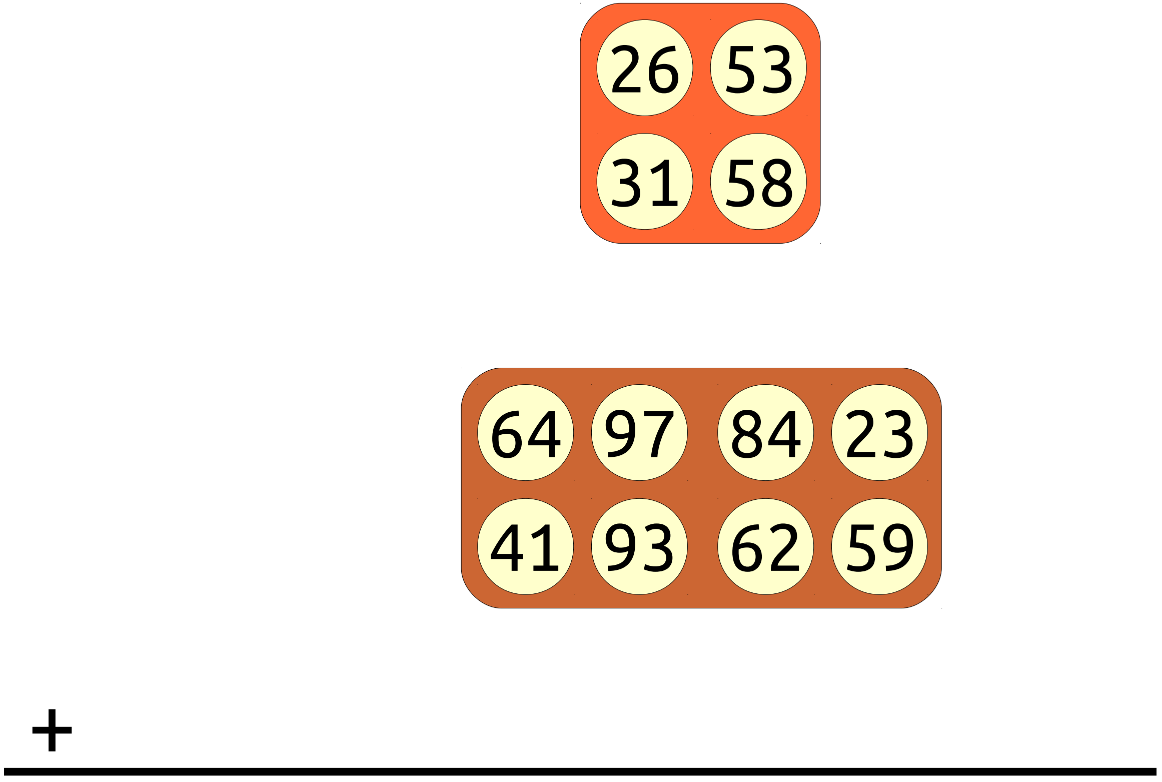
Building a Priority Queue

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



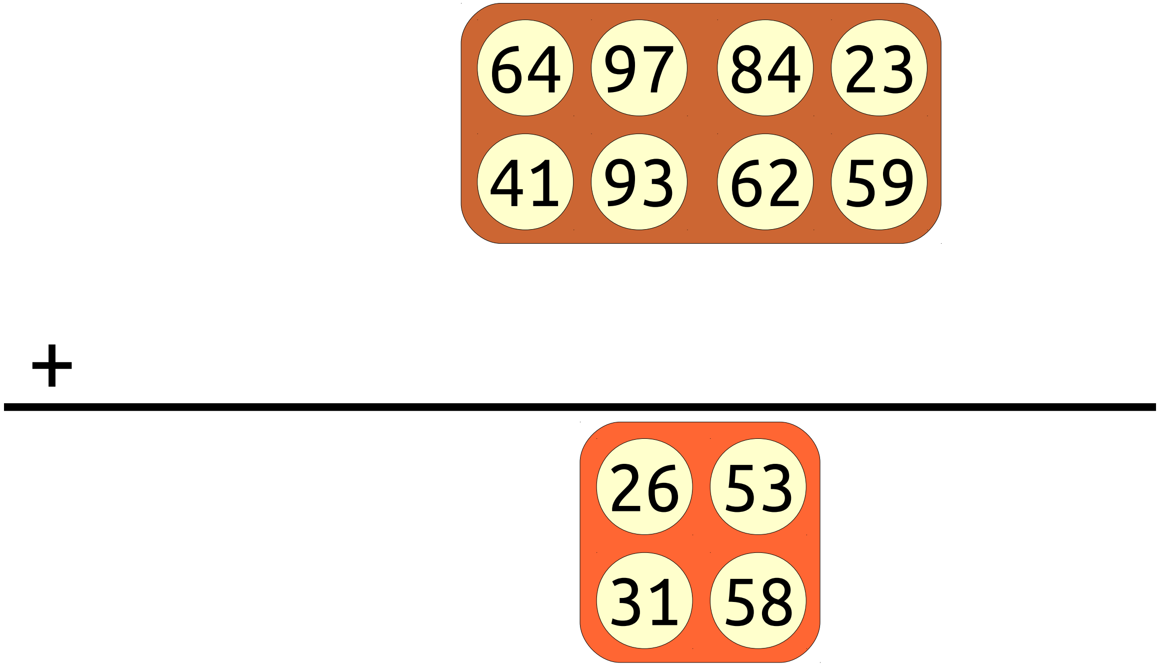
Building a Priority Queue

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



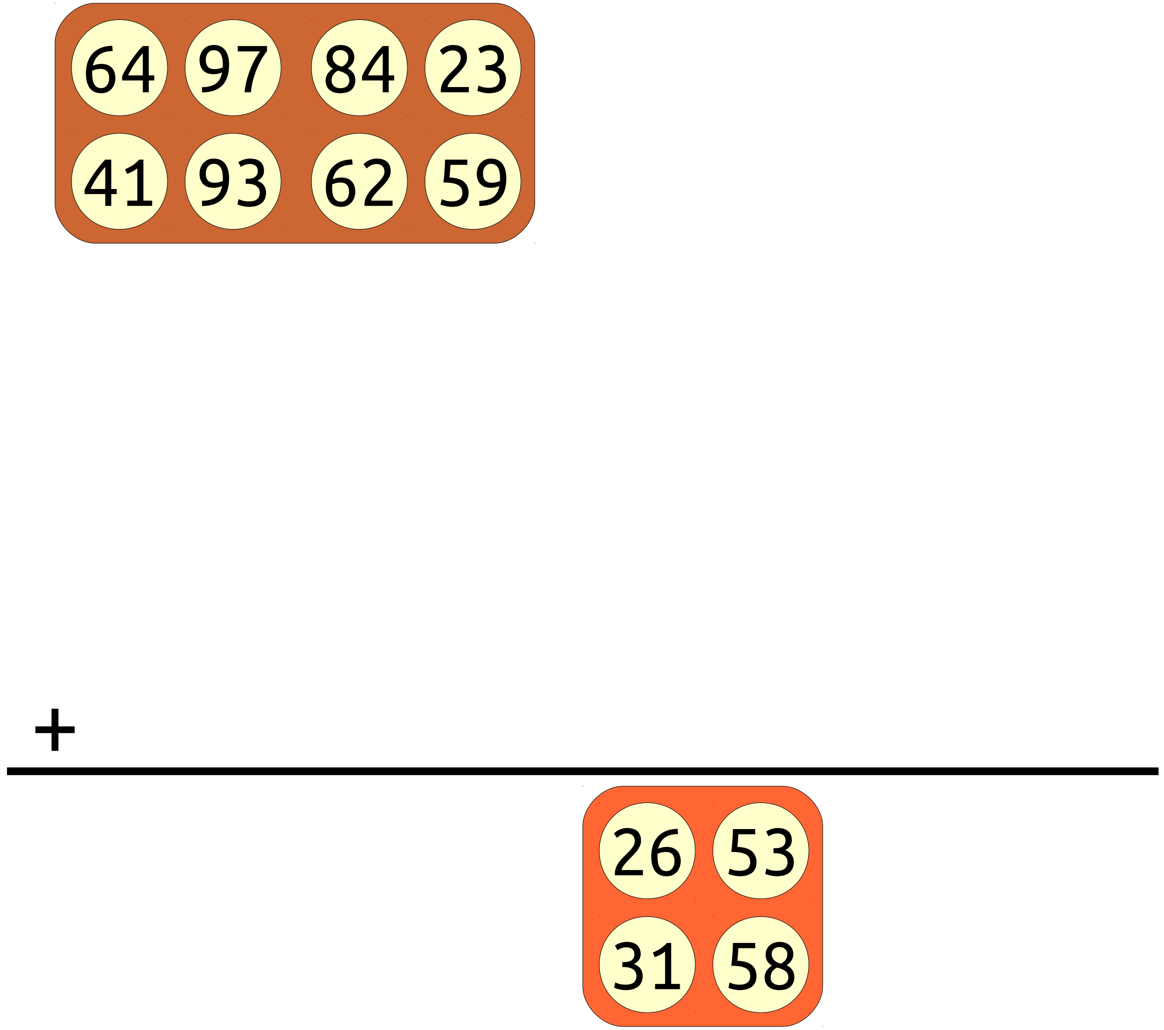
Building a Priority Queue

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



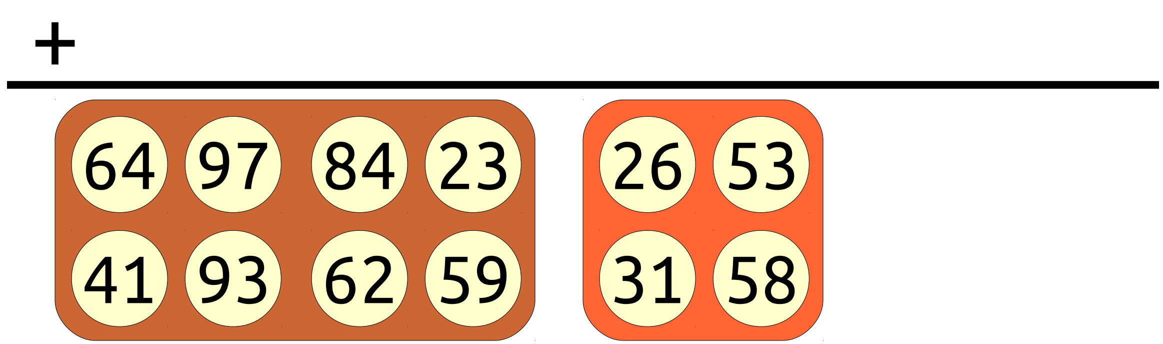
Building a Priority Queue

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



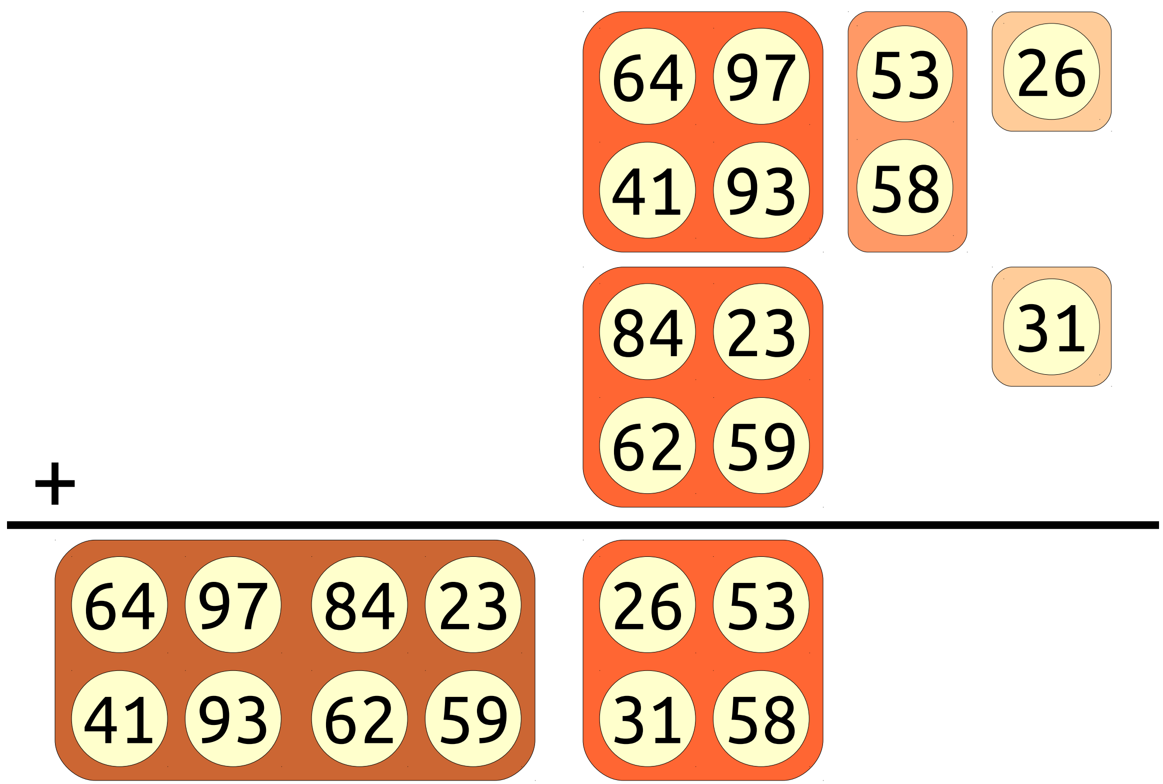
Building a Priority Queue

- ***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.

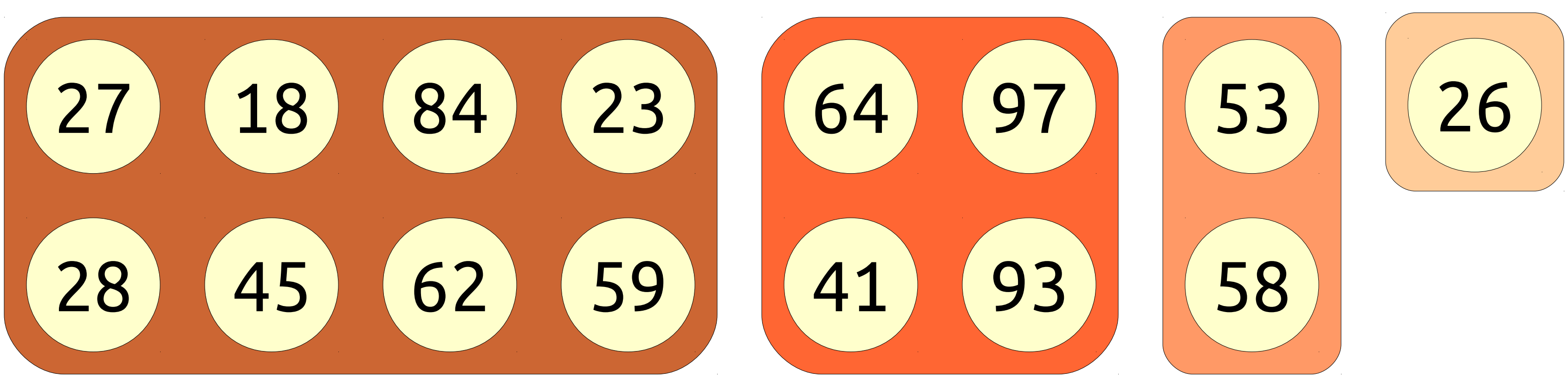


Building a Priority Queue

- What properties must our packets have?

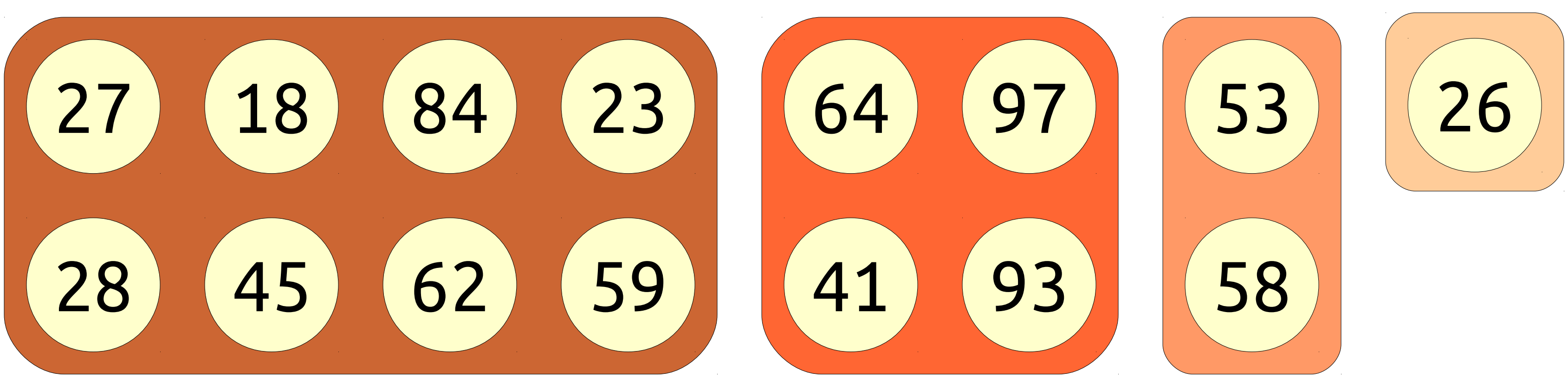
Building a Priority Queue

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



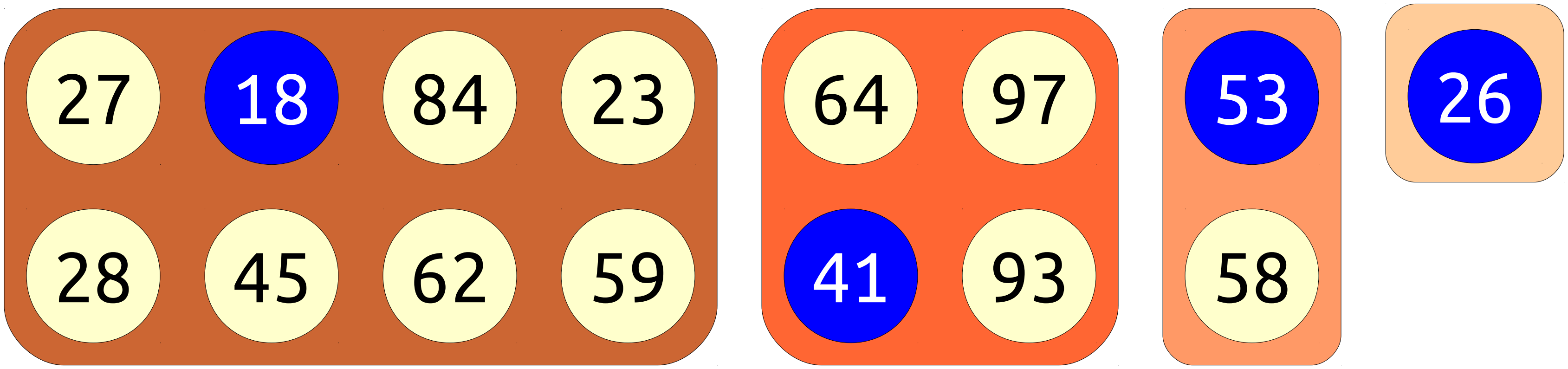
Building a Priority Queue

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



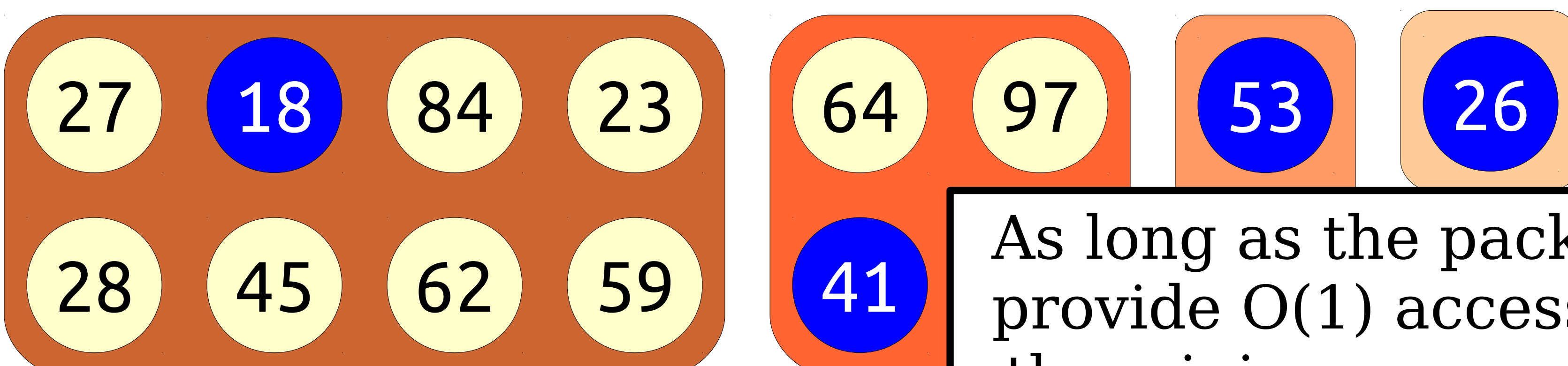
Building a Priority Queue

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



Building a Priority Queue

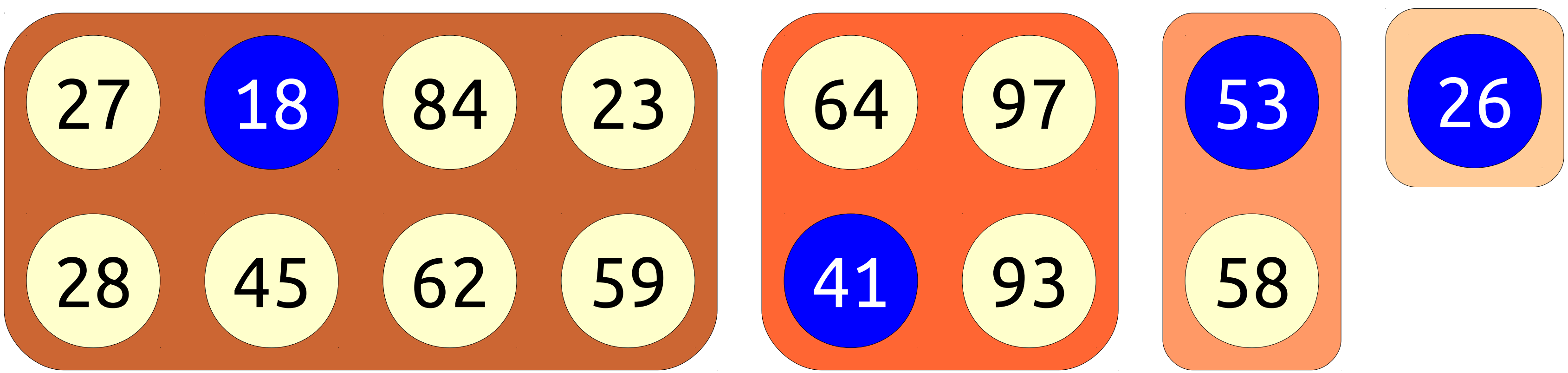
- What properties must our packets have?
  - Sizes must be powers of two.
  - Can efficiently fuse packets of the same size.



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.

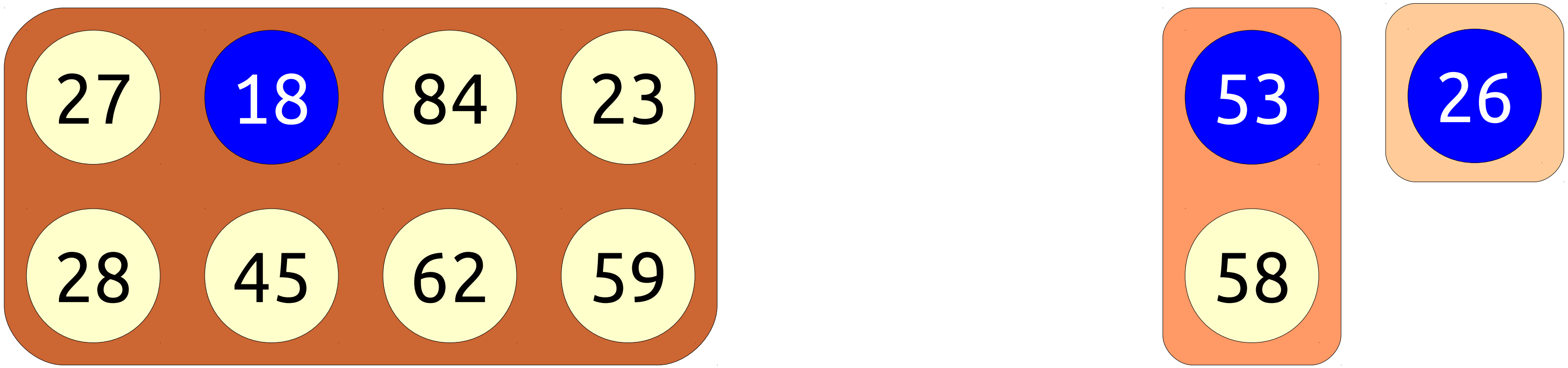


Inserting into the Queue

- 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.

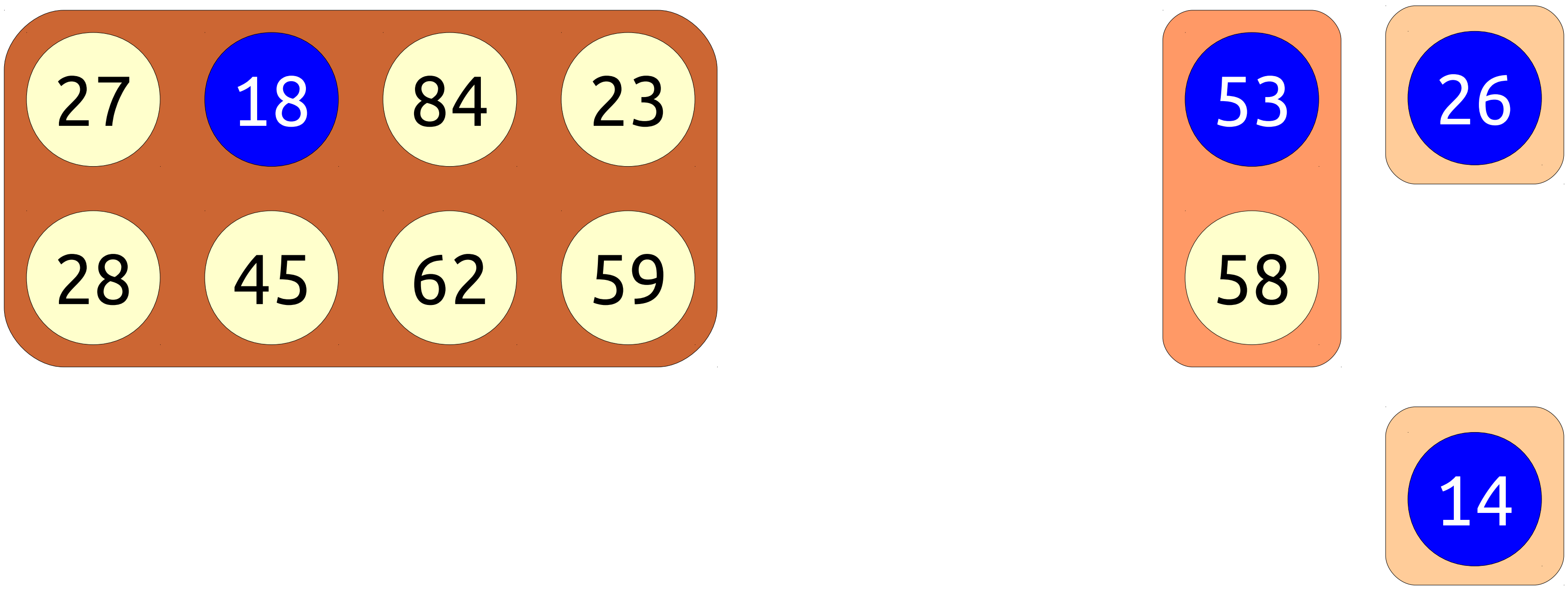
Inserting into the Queue

- 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.



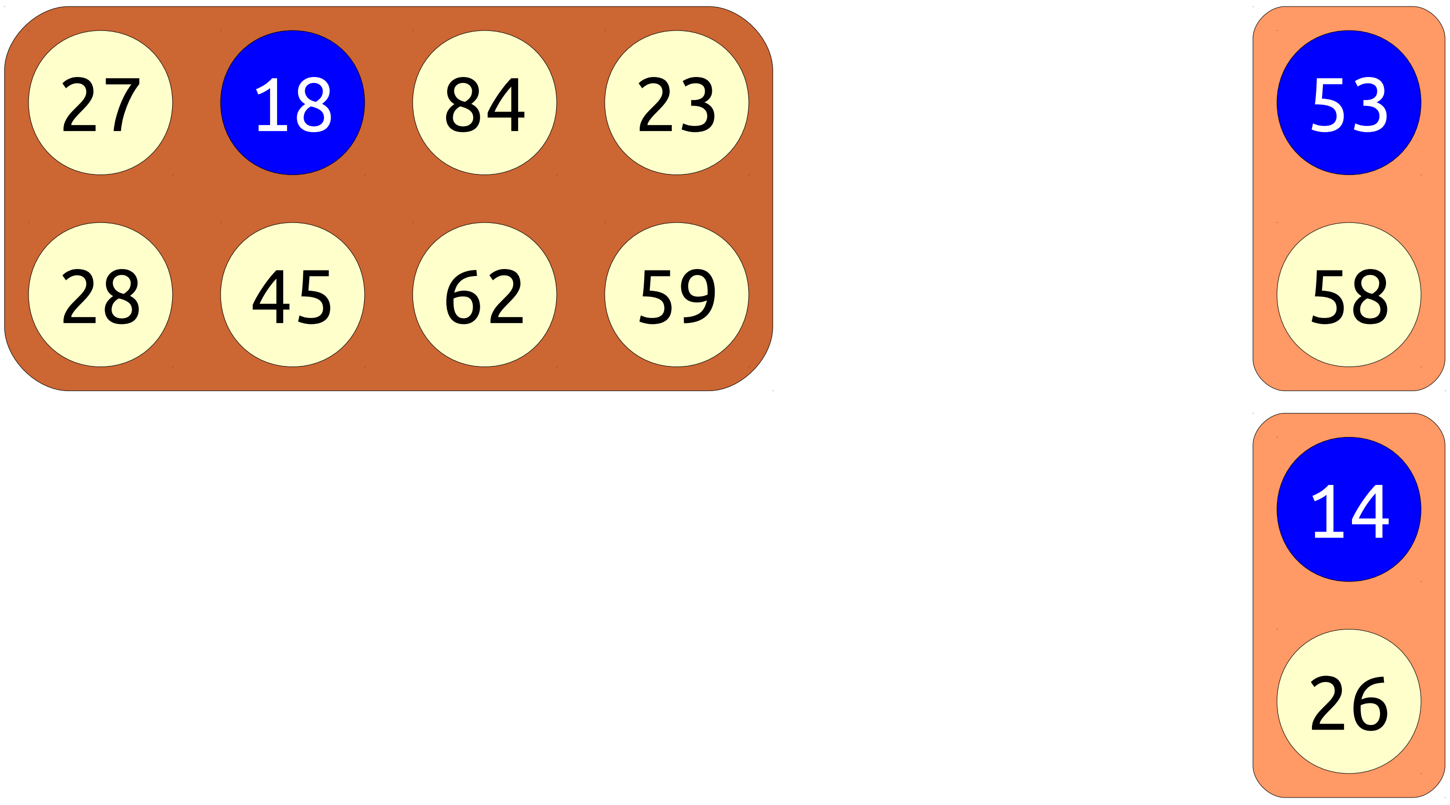
Inserting into the Queue

- 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.



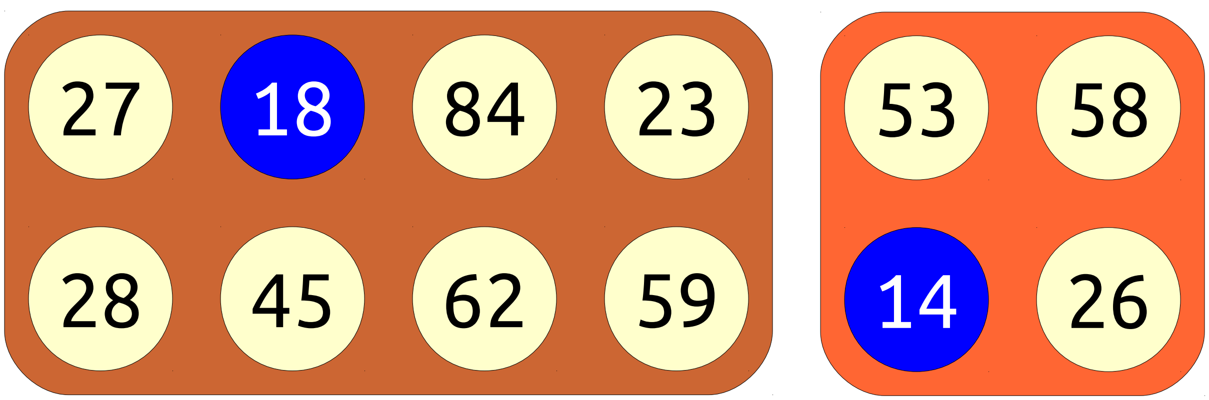
Inserting into the Queue

- 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.



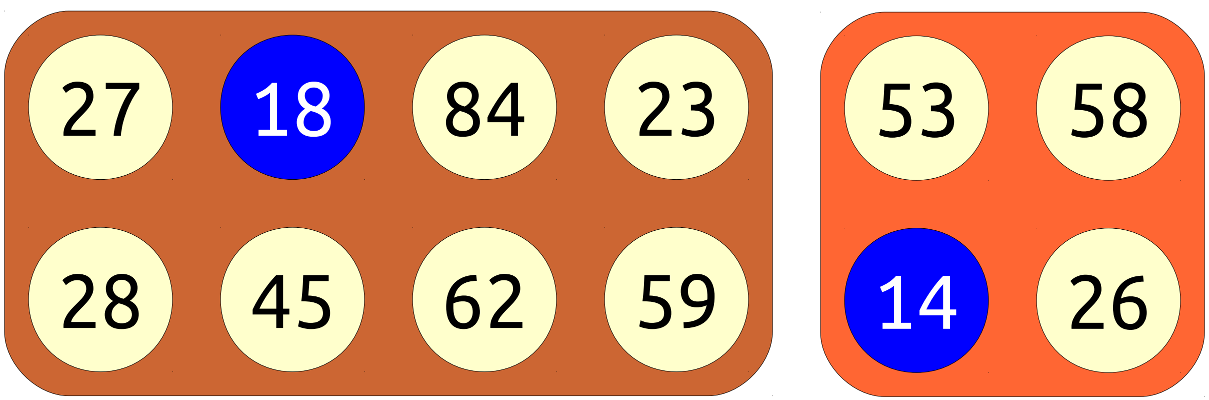
Inserting into the Queue

- 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.



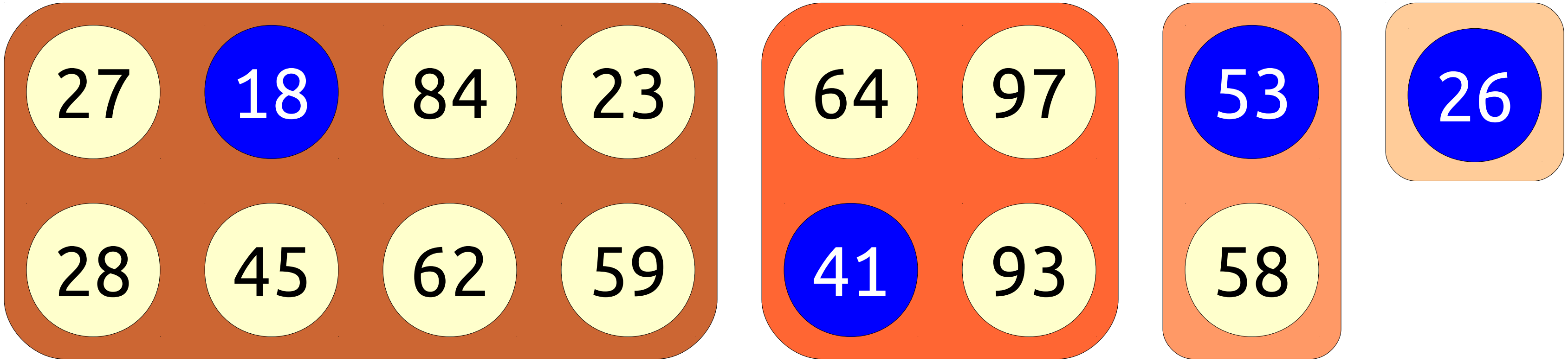
Time required:
O(log n) fuses.

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.

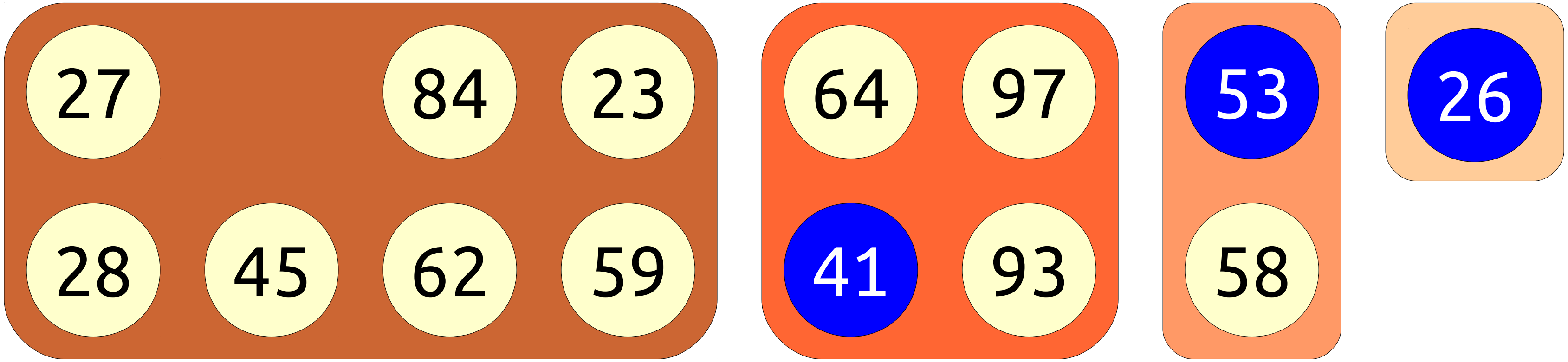
Deleting the Minimum

- 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.



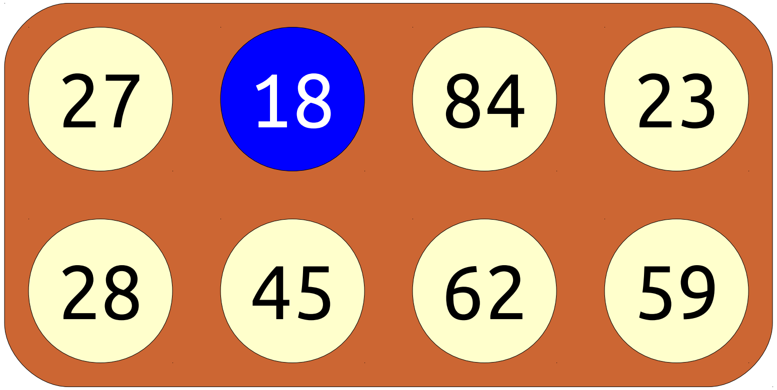
Deleting the Minimum

- 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.



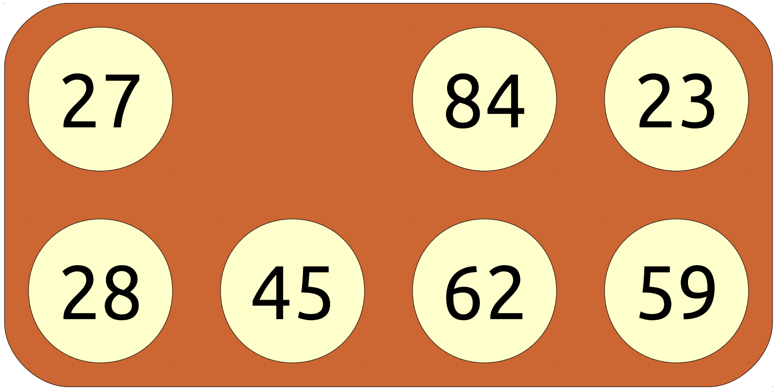
Deleting the Minimum

- 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.
- ***Fun fact***: $2^k - 1 = 2^0 + 2^1 + 2^2 + \ldots + 2^{k-1}$.



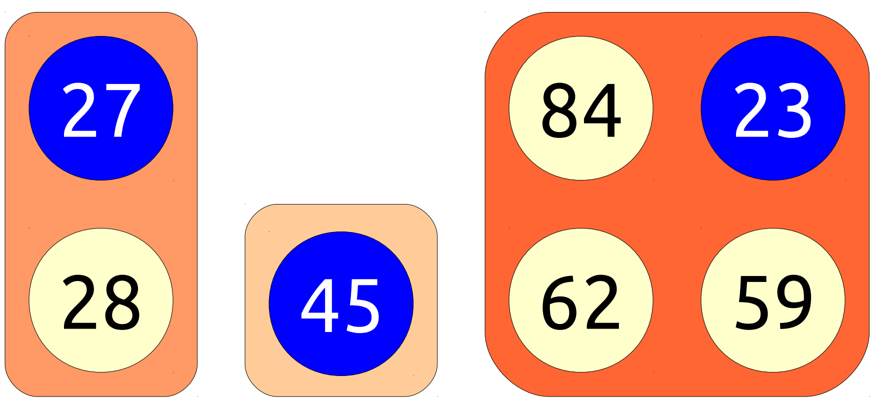
Deleting the Minimum

- 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***: $2^k – 1 = 2^0 + 2^1 + 2^2 + … + 2^{k-1}$.



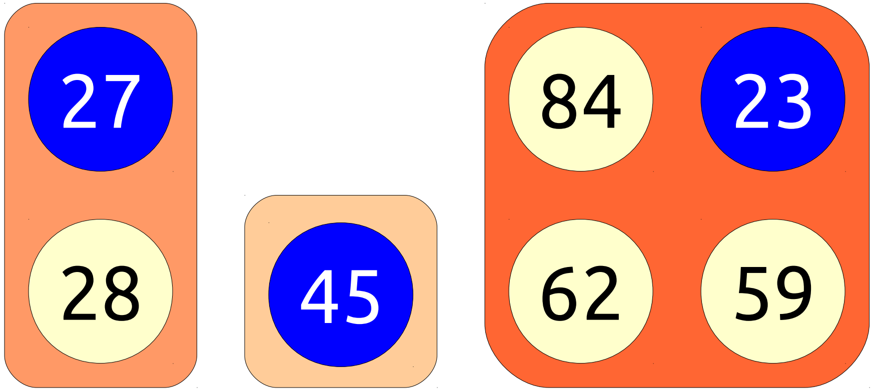
Deleting the Minimum

- 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***: $2^k - 1 = 2^0 + 2^1 + 2^2 + \ldots + 2^{k-1}$.



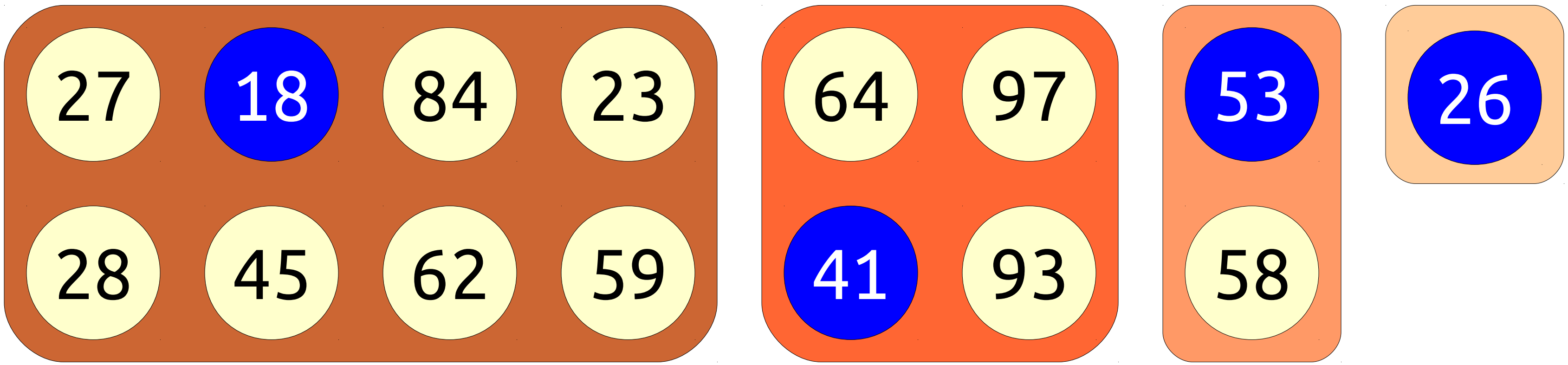
Deleting the Minimum

- 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***: $2^k - 1 = 2^0 + 2^1 + 2^2 + \ldots + 2^{k-1}$.
- ***Idea***: "Fracture" the packet into $k - 1$ smaller packets, then add them back in.



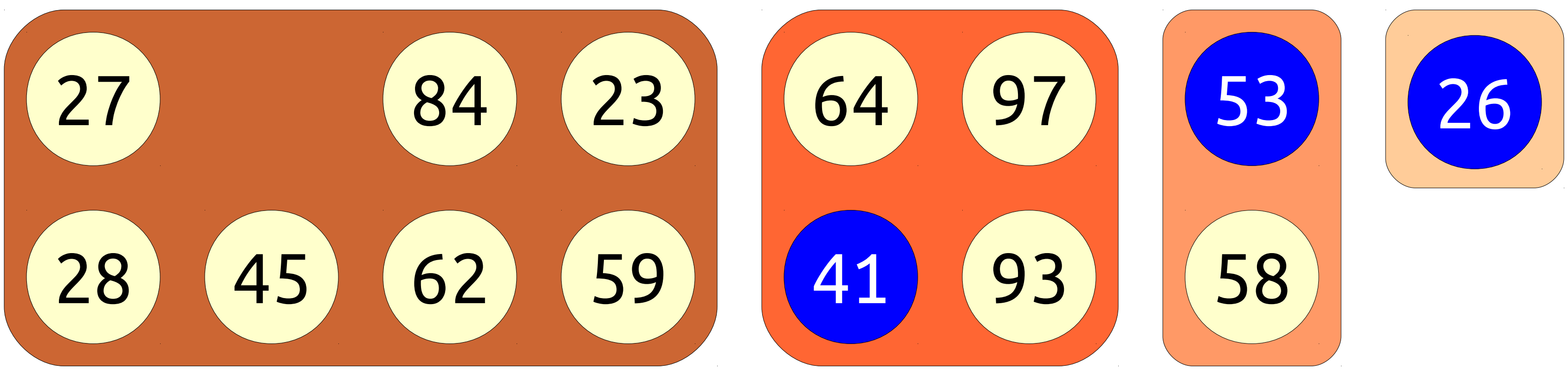
Fracturing Packets

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



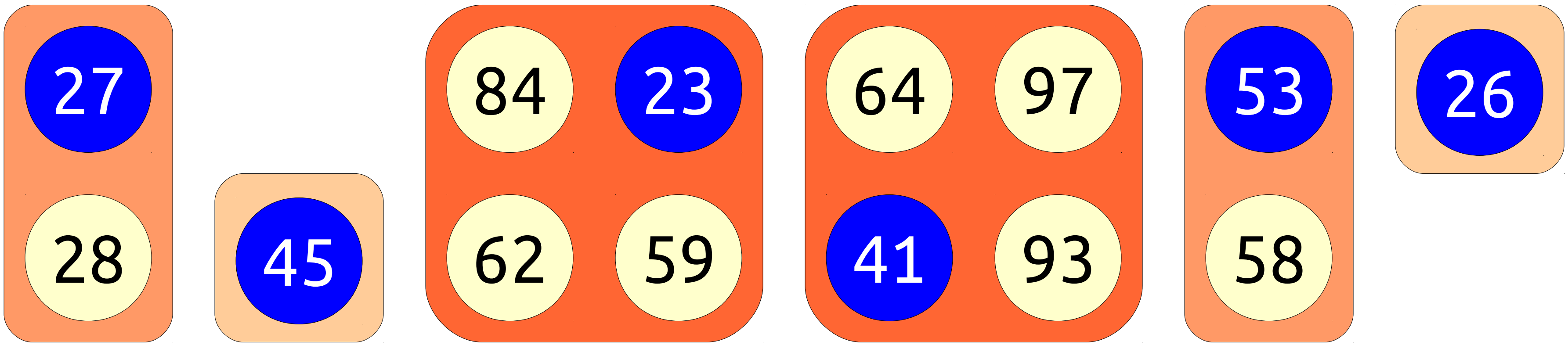
Fracturing Packets

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



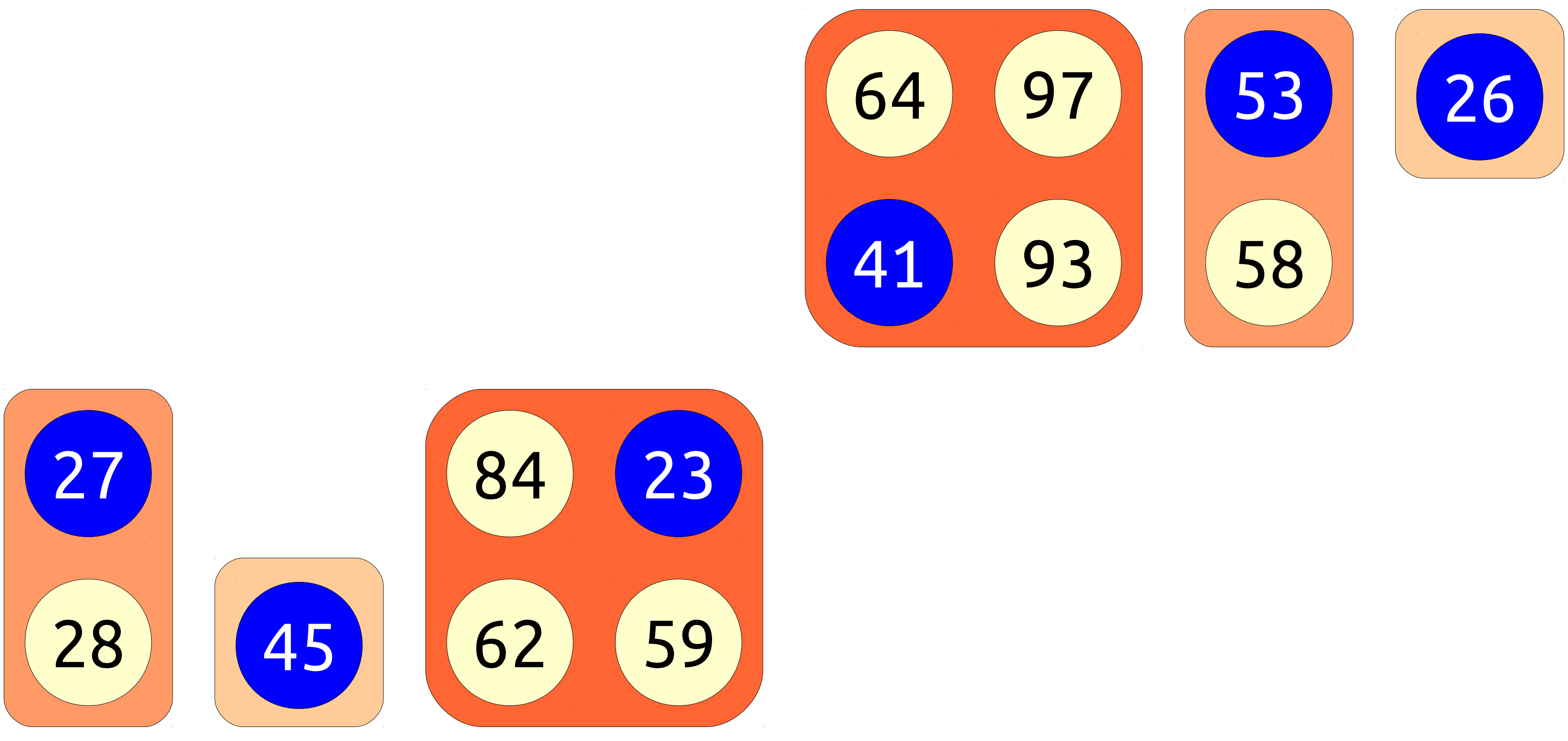
Fracturing Packets

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



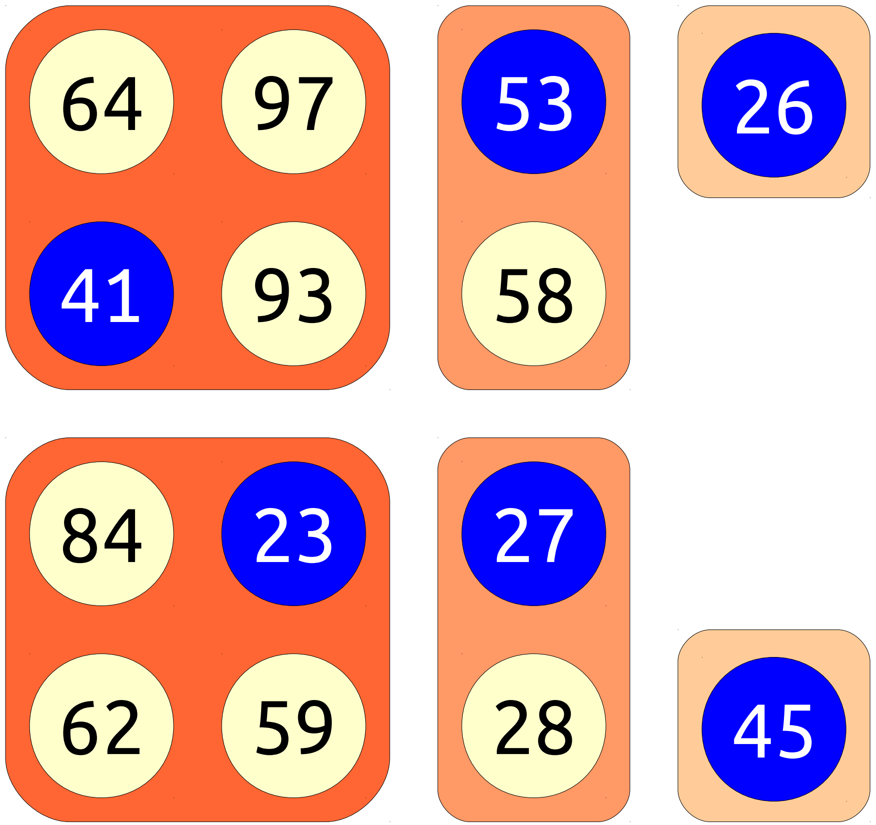
Fracturing Packets

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



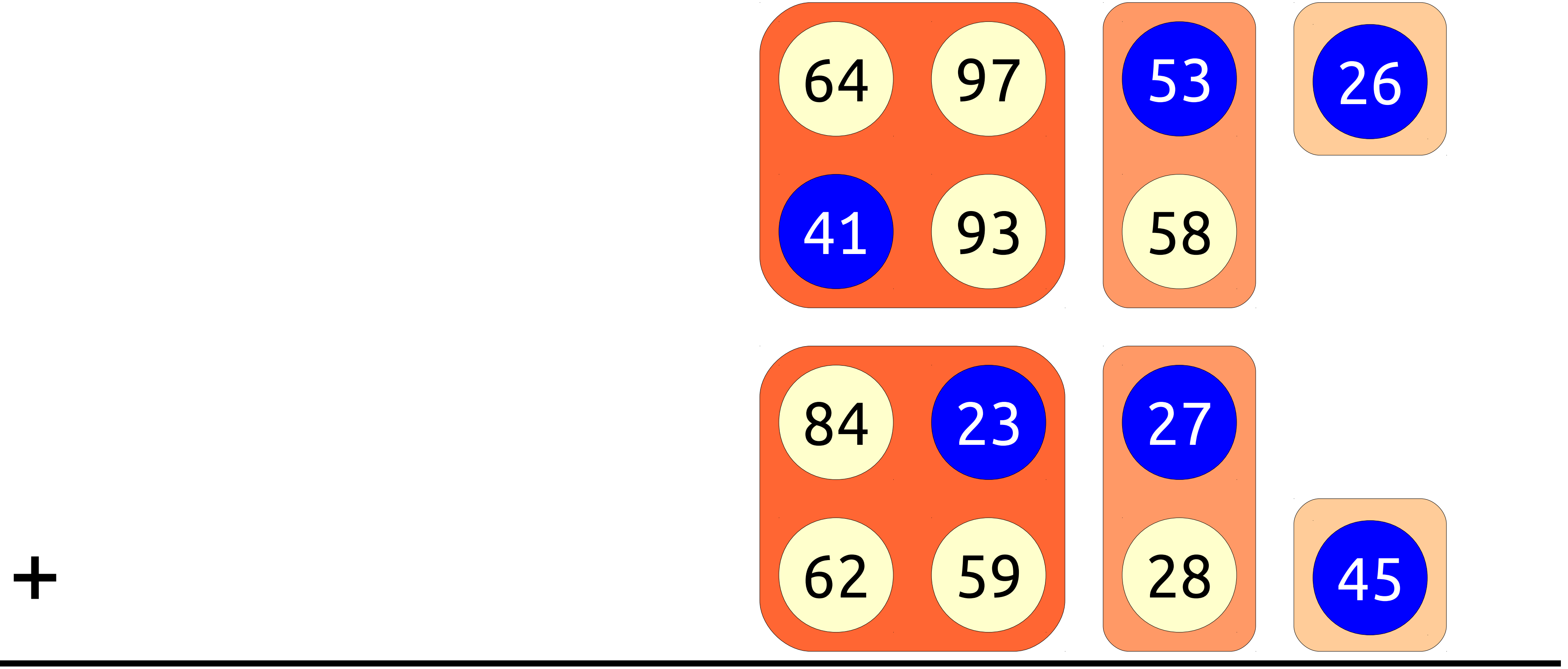
Fracturing Packets

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



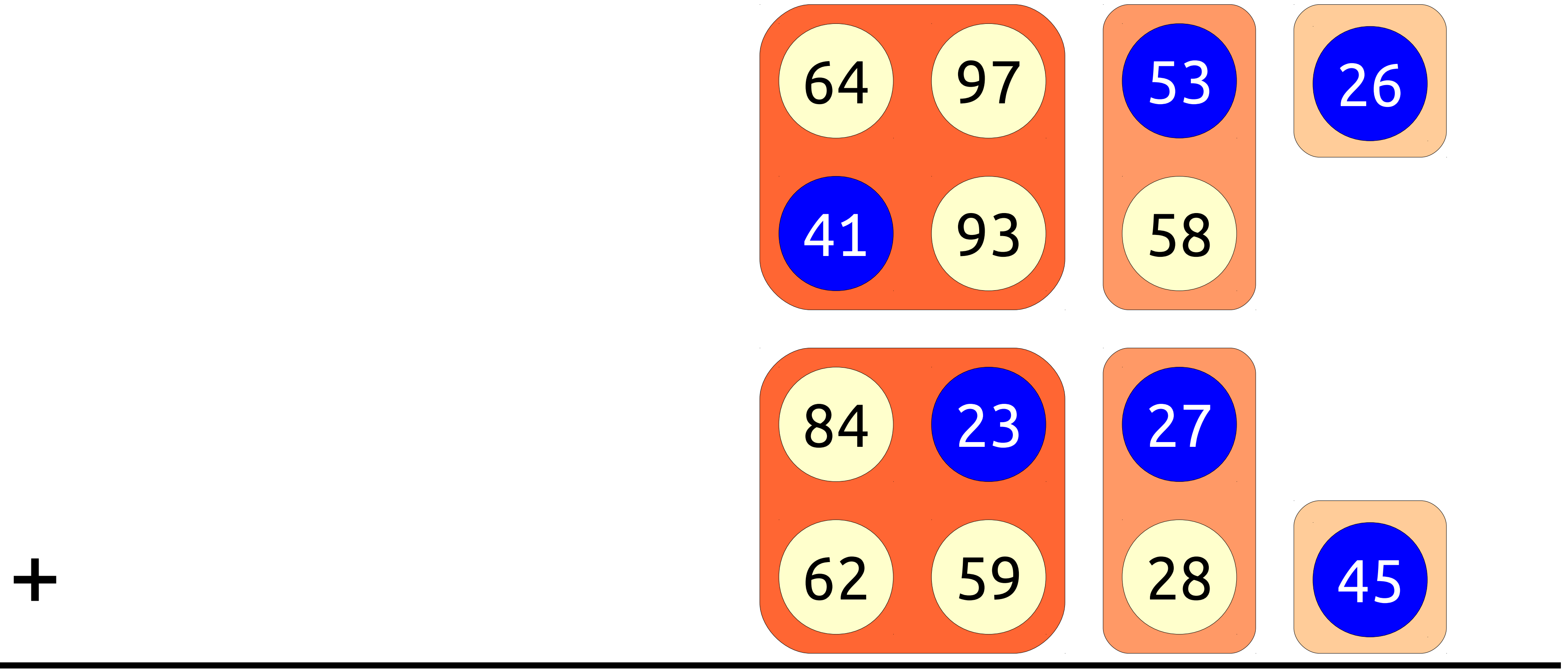
Fracturing Packets

- 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 fragment cost.

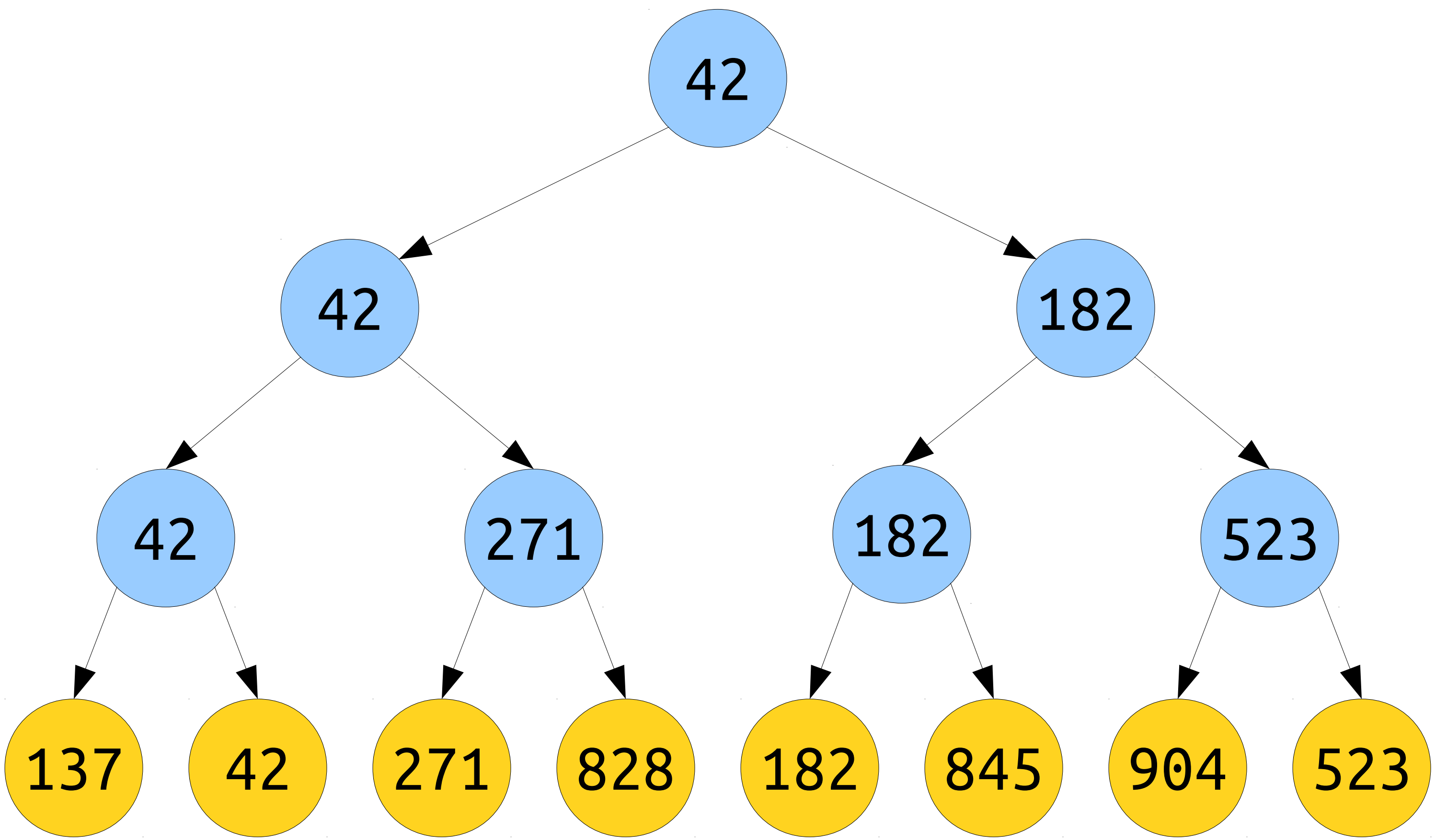


Building a Priority Queue

- 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 $2^0, 2^1, 2^2, 2^3, \ldots, 2^{k-1}$ nodes.
- ***Question:*** How can we represent our packets to support the above operations efficiently?

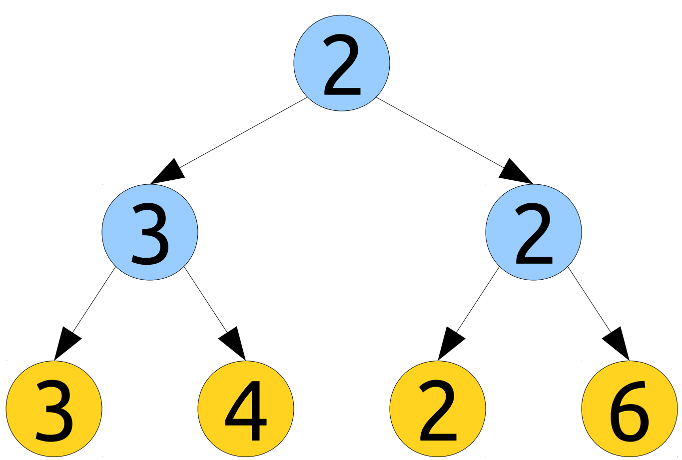
Tournament Trees

- A ***tournament tree*** is a complete binary tree representing the result of a tournament.



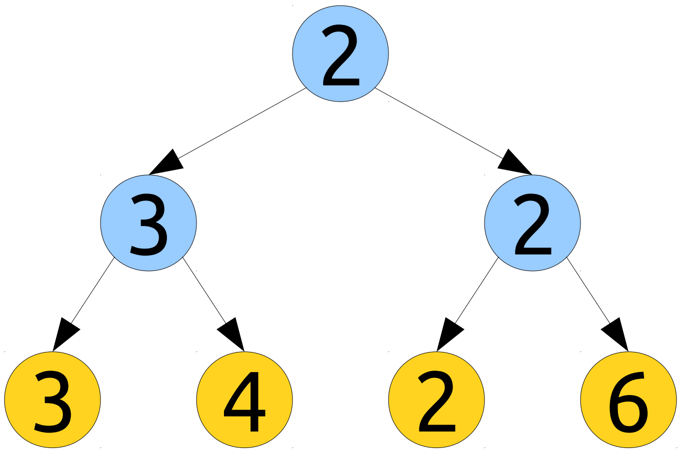
Tournament Trees

- 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, …, 2^{k-1} nodes.



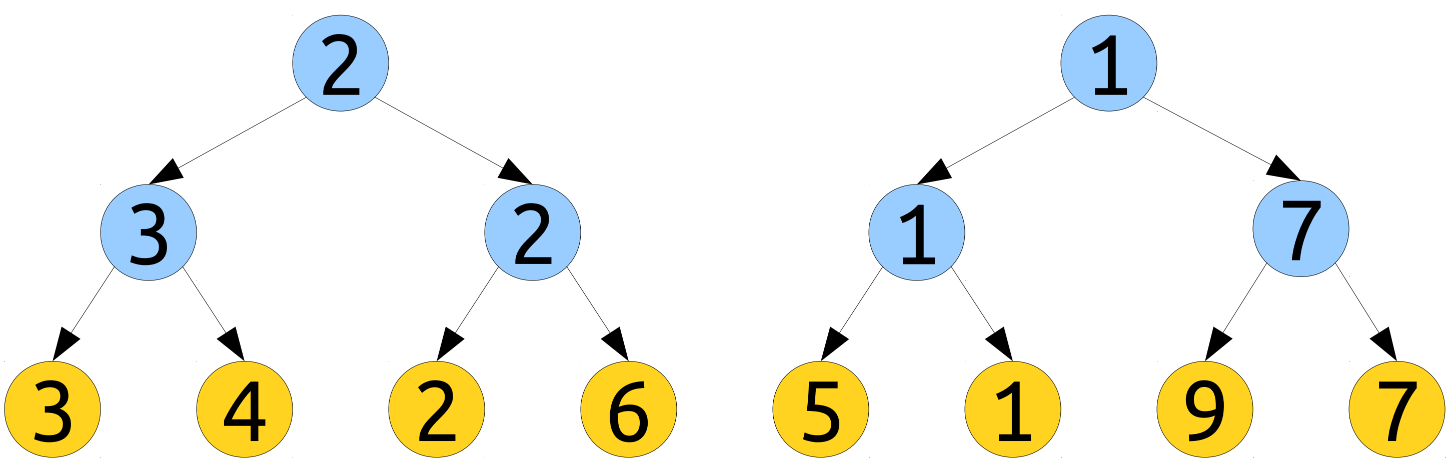
Tournament Trees

- 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, …, 2^{k-1} nodes.



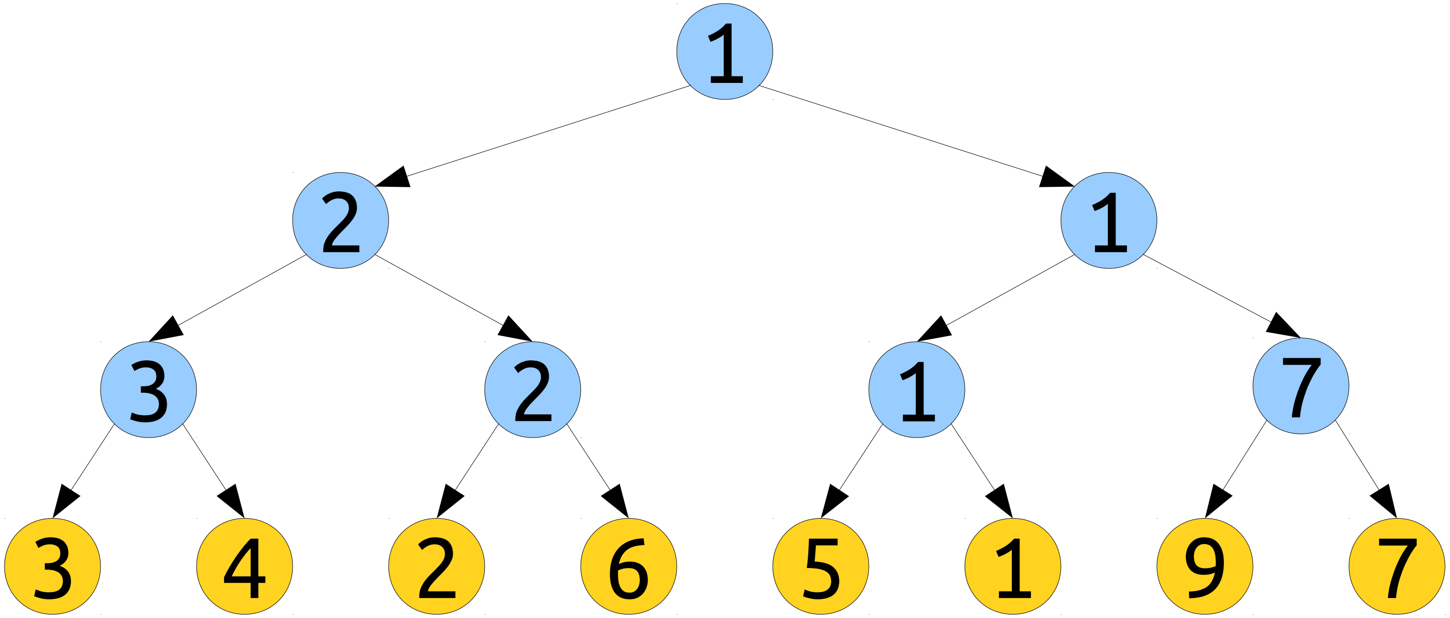
Tournament Trees

- 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, …, 2^{k-1} nodes.



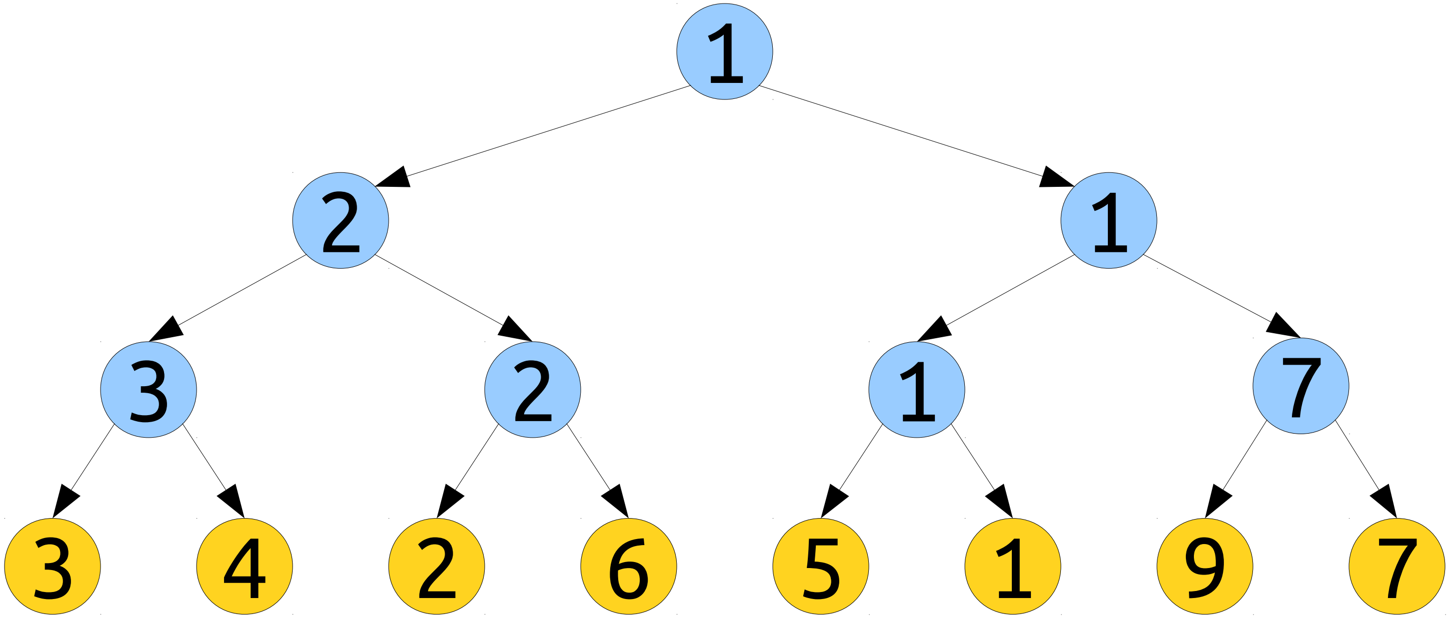
Tournament Trees

- 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, …, 2^{k-1} nodes.



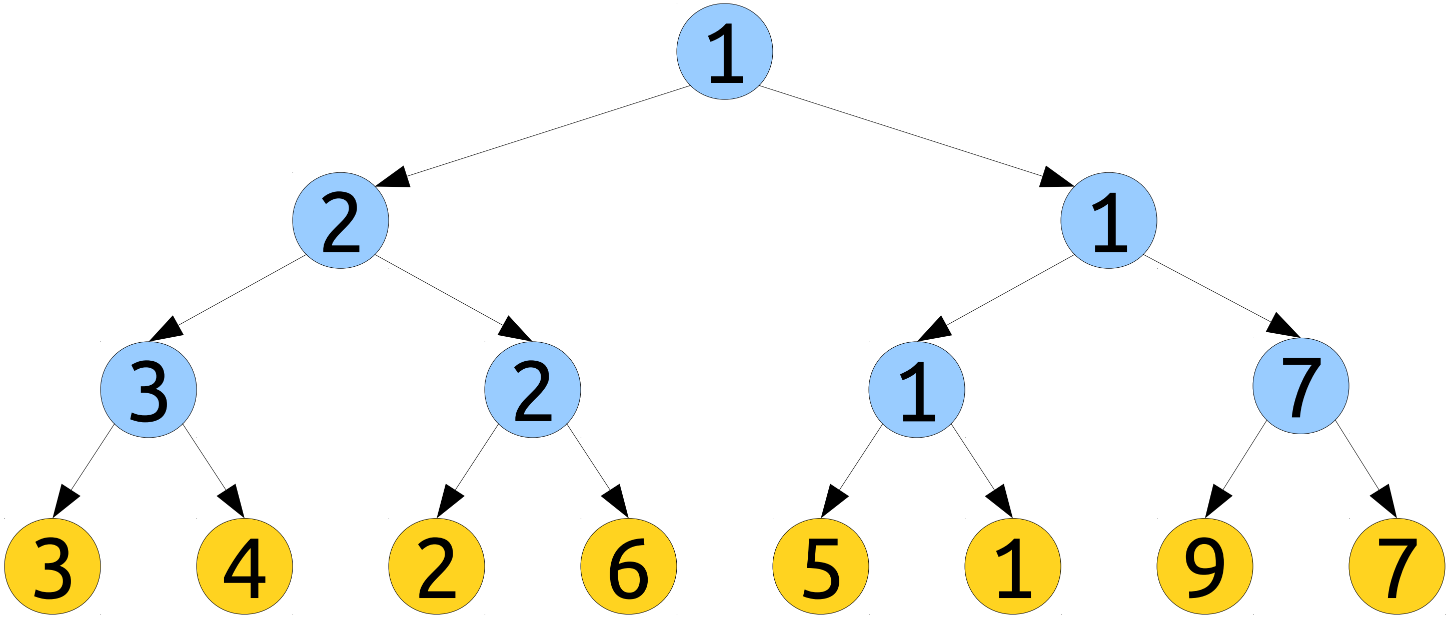
Tournament Trees

- 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, …, 2^{k-1} nodes.



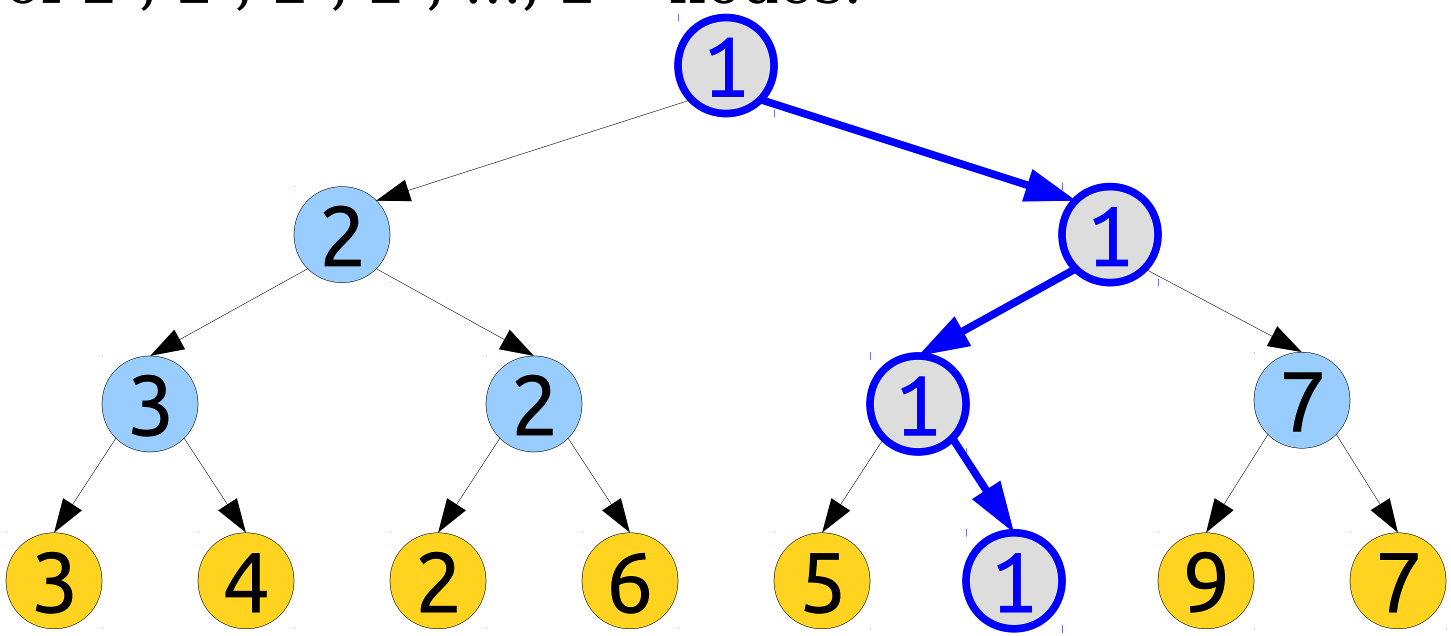
Tournament Trees

- 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 $2^0, 2^1, 2^2, 2^3, \ldots, 2^{k-1}$ nodes.



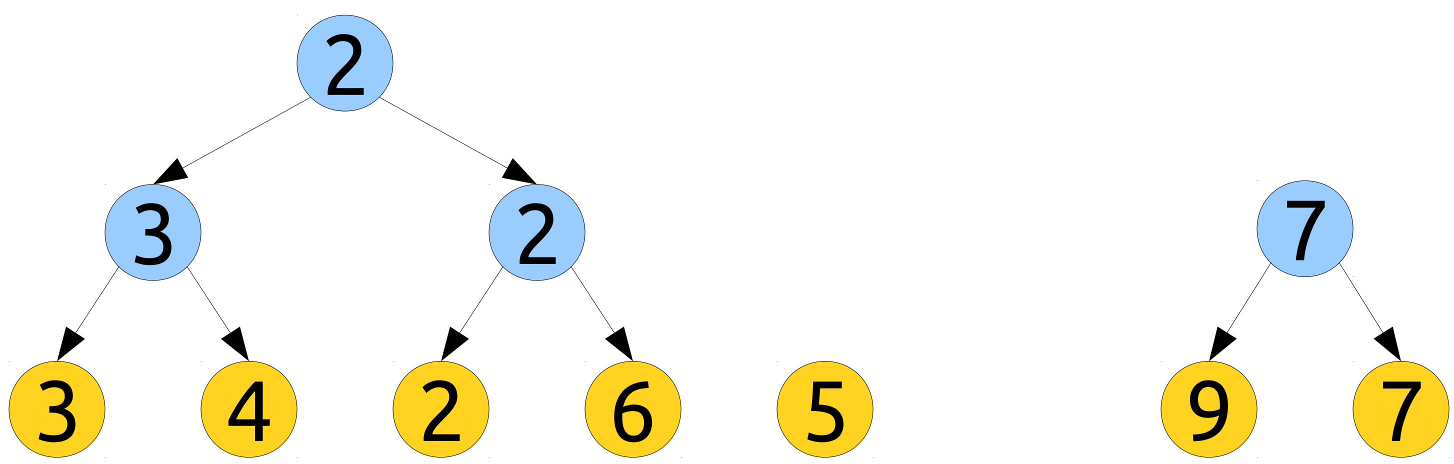
Tournament Trees

- 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, …, 2^{k-1} nodes.



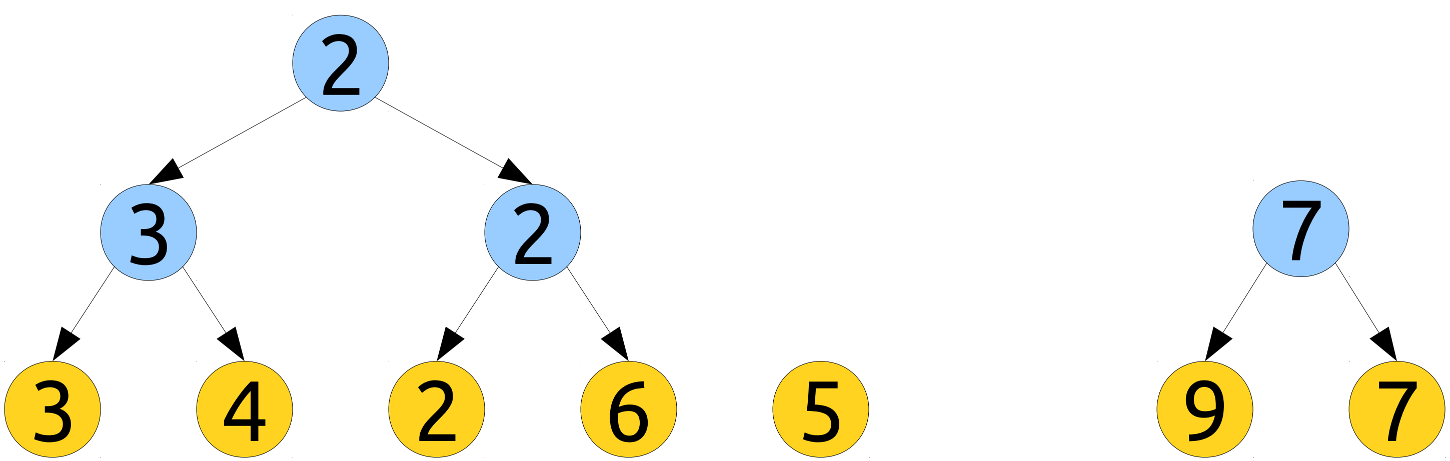
Tournament Trees

- 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, …, 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, …, 2^{k-1} nodes. √



Tournament Trees

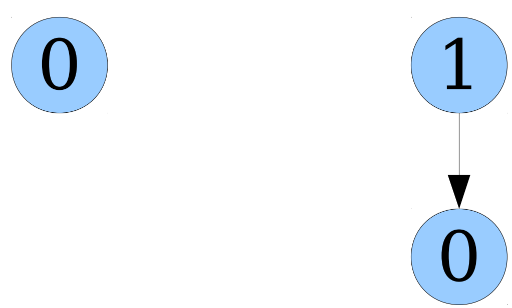
- Although tournament trees have the properties we need, they're typically not used for this application.
- However:
  - the idea of *storing keys purely in leaves* shows up in a bunch of other data structures (*y*-fast tries, various flavors of heaps), and
  - tournament trees are used in other fast priority queues, including the modern ***quake heap*** (cool project topic!)
- So what are people using instead?

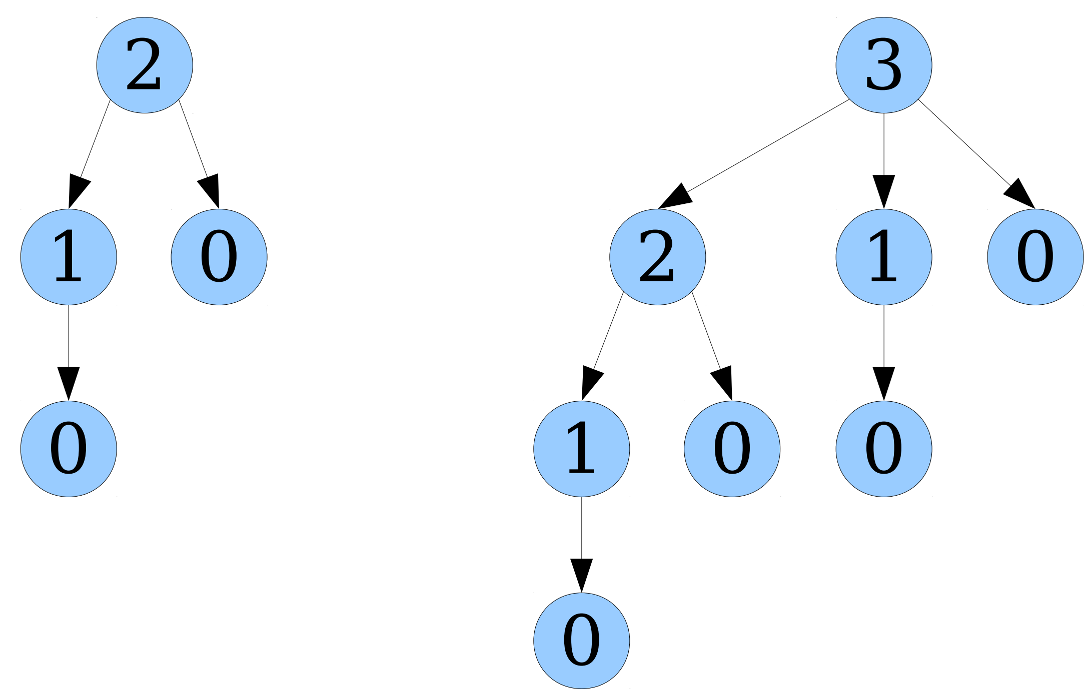
Binomial Trees

- A ***binomial tree of order k*** is a type of tree recursively defined as follows:

  A binomial tree of order k is a single node whose children are binomial trees of order 0, 1, 2, …, k – 1.

- Here are the first few binomial trees:





Why are these called binomial heaps? Look across the layers of these trees and see if you notice anything!

Binomial Trees

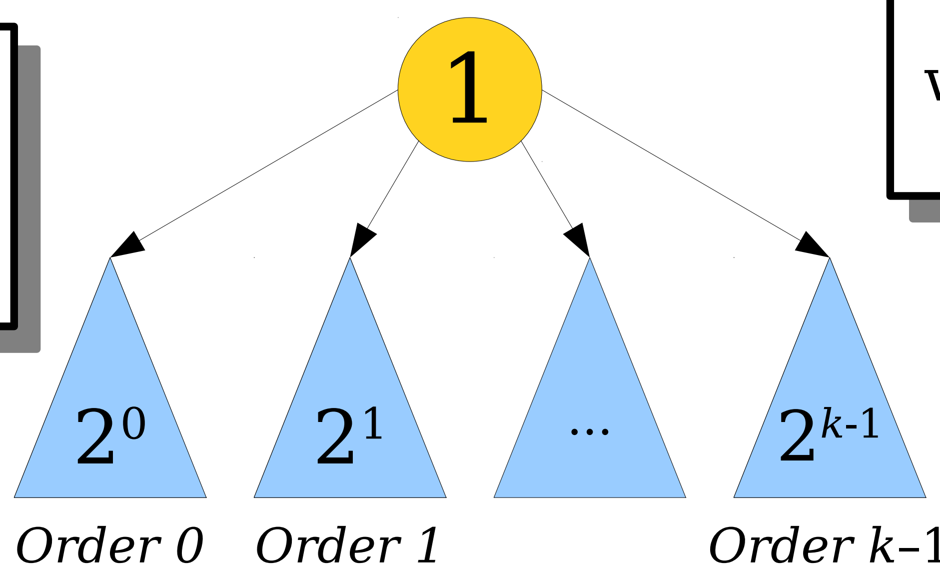
- ***Theorem:*** A binomial tree of order k has exactly 2^k nodes.
- ***Proof:*** Induction on k.

Assume that binomial trees of orders 0, 1, …, $k - 1$ have $2^0, 2^1, \ldots, 2^{k-1}$ nodes. The number of nodes in an order-k binomial tree is

$$2^0 + 2^1 + \ldots + 2^{k-1} + 1 = 2^k - 1 + 1 = 2^k$$

So the claim holds for k as well. ■

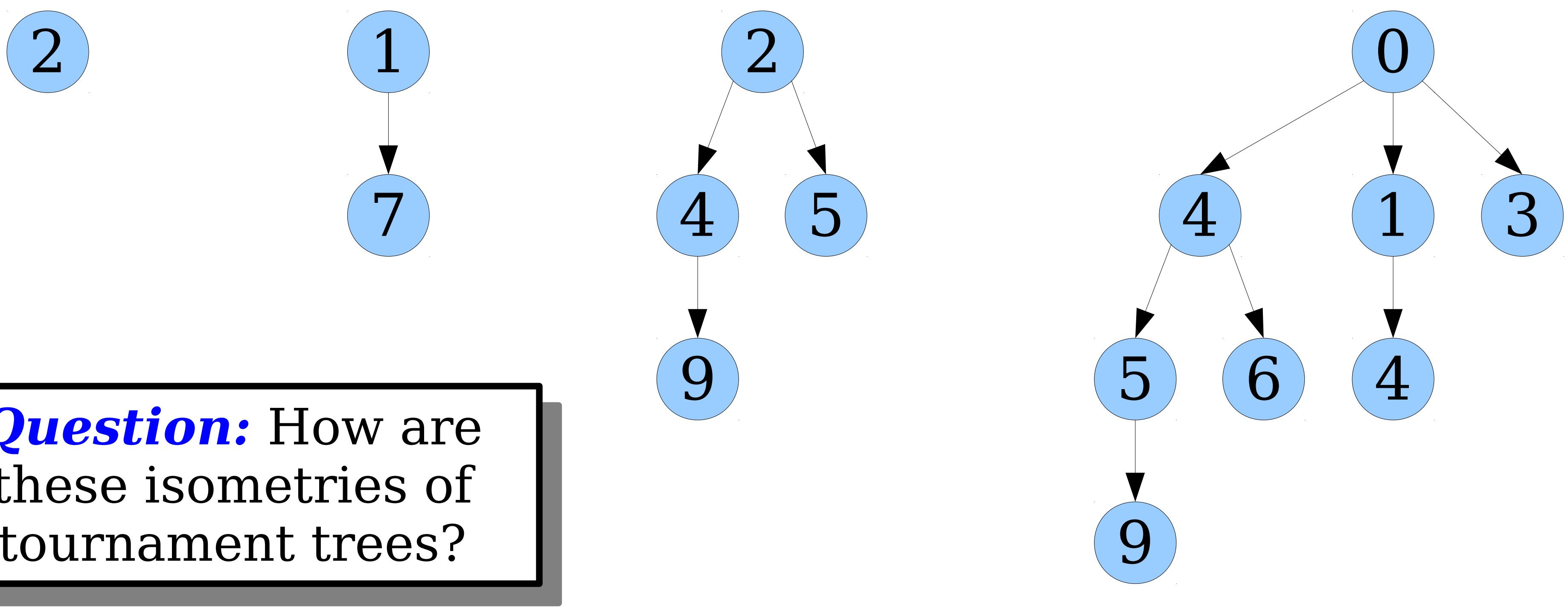
Deep Question: Why doesn't this inductive proof have a base case?



There's another way to show this. Stay tuned!

Binomial Trees

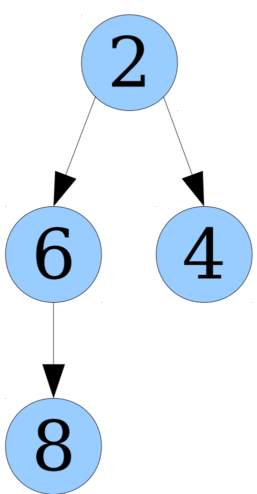
- A ***heap-ordered binomial tree*** is a binomial tree whose nodes obey the heap property: all nodes are less than or equal to their descendants.
- We will use heap-ordered binomial trees to implement our “packets.”



Question: How are these isometries of tournament trees?

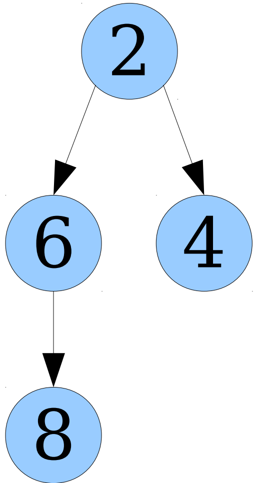
Binomial Trees

- 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, …, 2^{k-1} nodes.



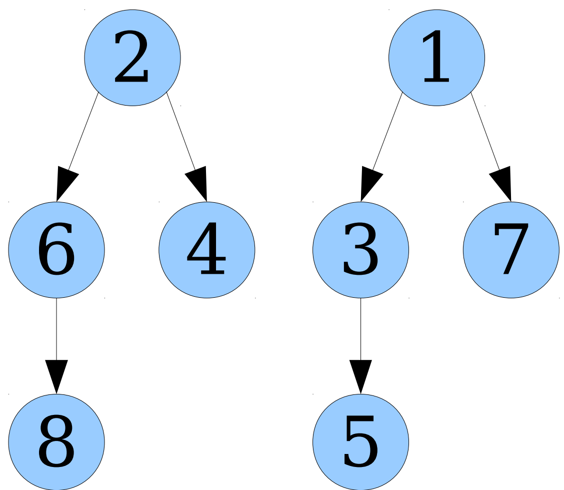
Binomial Trees

- 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 $2^0, 2^1, 2^2, 2^3, \ldots, 2^{k-1}$ nodes.



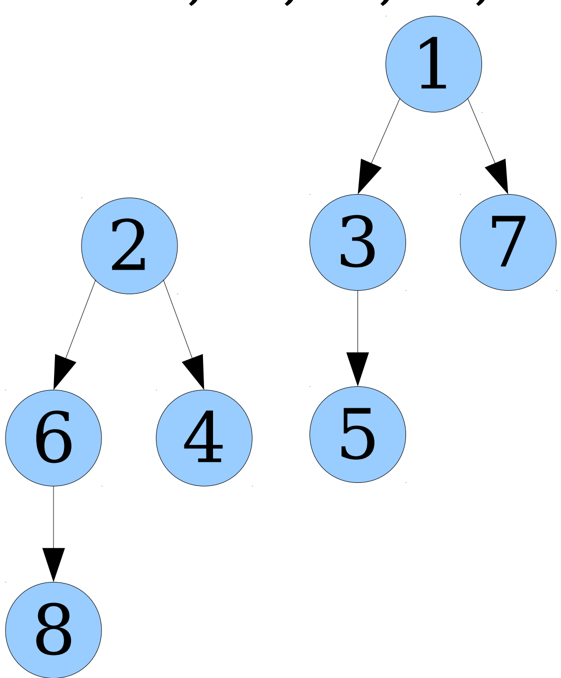
Binomial Trees

- 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, …, 2^{k-1} nodes.



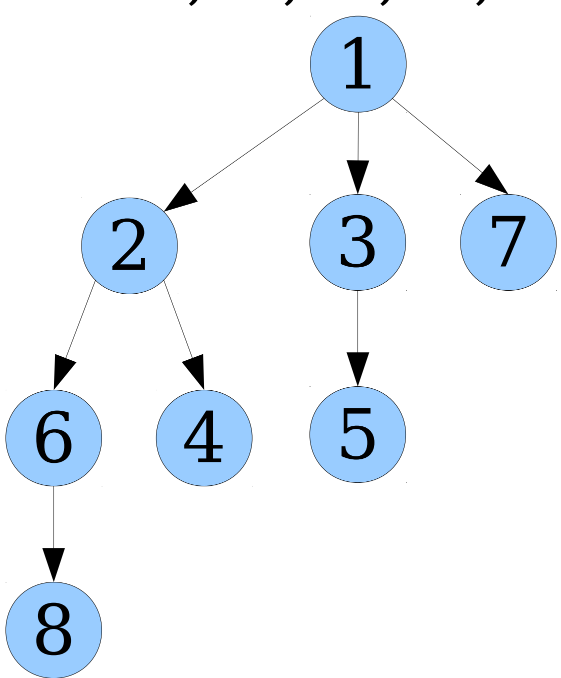
Binomial Trees

- 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, …, 2^{k-1} nodes.



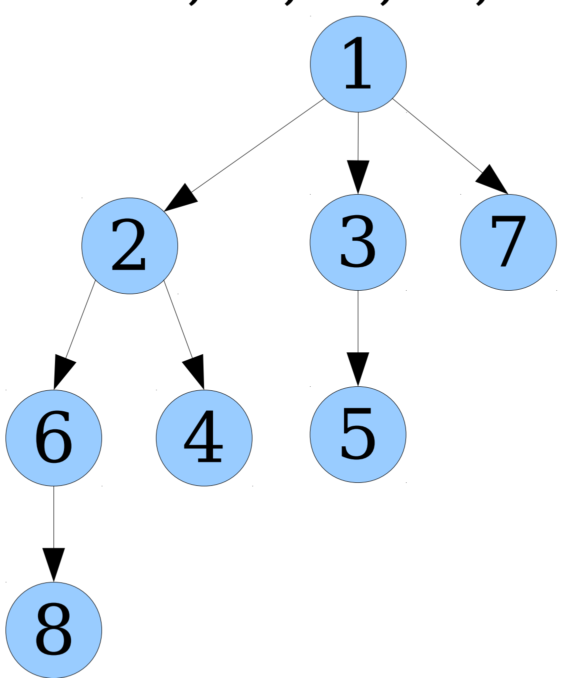
Binomial Trees

- 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, …, 2^{k-1} nodes.



Binomial Trees

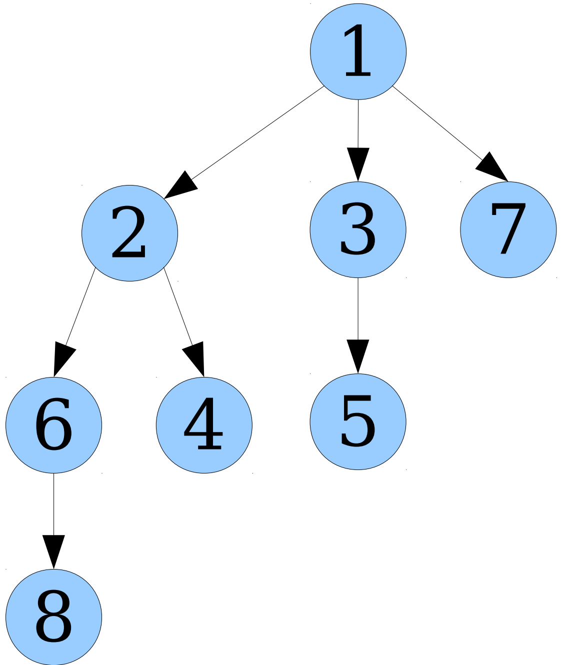
- What properties must our packets have?
  - Size must be a power of two. √
  - Can efficiently fuse packets of the same size.
  - Can efficiently find the minimum element of each packet.
  - Can efficiently "fracture" a packet of 2^k nodes into packets of 2^0, 2^1, 2^2, 2^3, …, 2^{k-1} nodes.



Make the binomial tree with the larger root the first child of the tree with the smaller root.

Binomial 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, …, 2^{k-1} nodes.

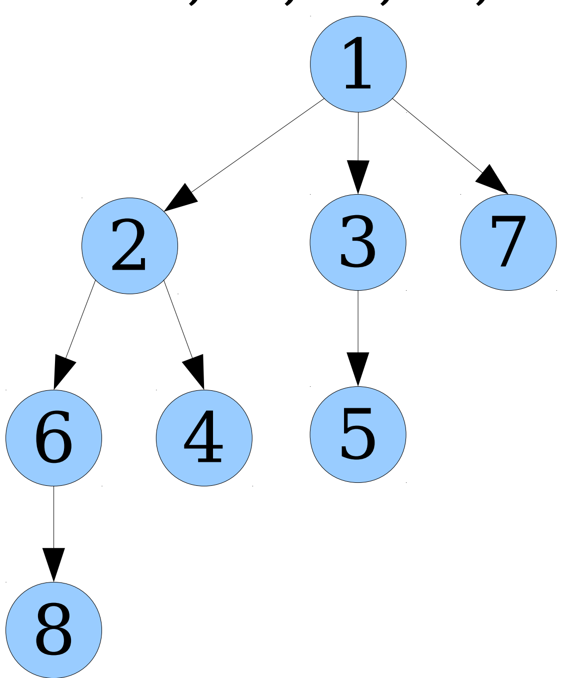


Make the binomial tree with the larger root the first child of the tree with the smaller root.

Hey! That also means the number of nodes in the tree doubles as the order increases!

Binomial 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 $2^0, 2^1, 2^2, 2^3, \ldots, 2^{k-1}$ nodes.

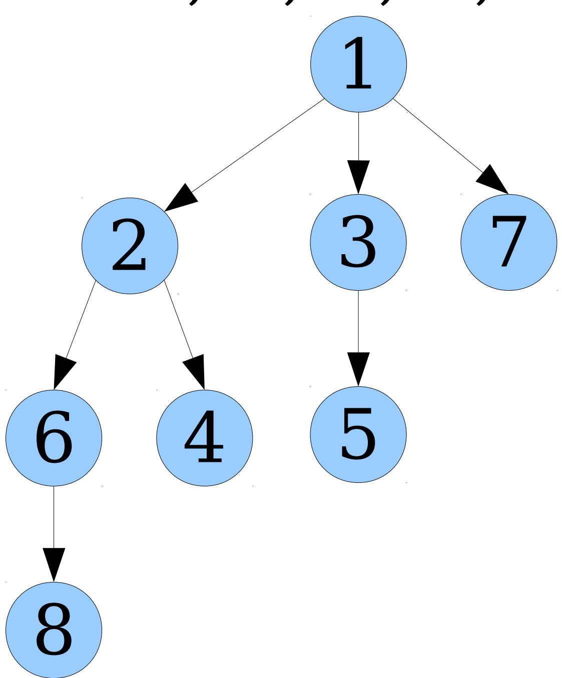


Make the binomial tree with the larger root the first child of the tree with the smaller root.

Hey! That also means the number of nodes in the tree doubles as the order increases!

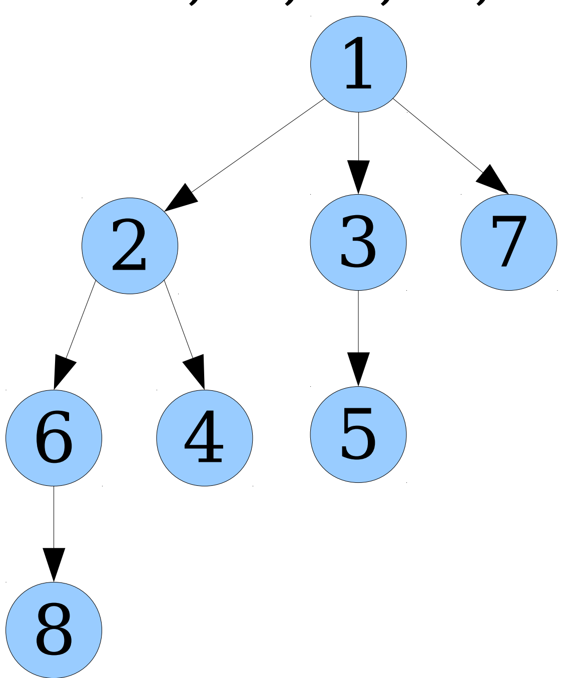
Binomial Trees

- 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, …, 2^{k-1} nodes.



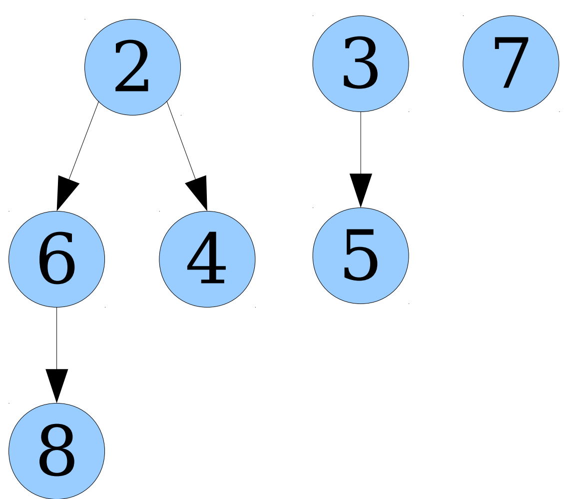
Binomial Trees

- 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, …, 2^{k-1} nodes.



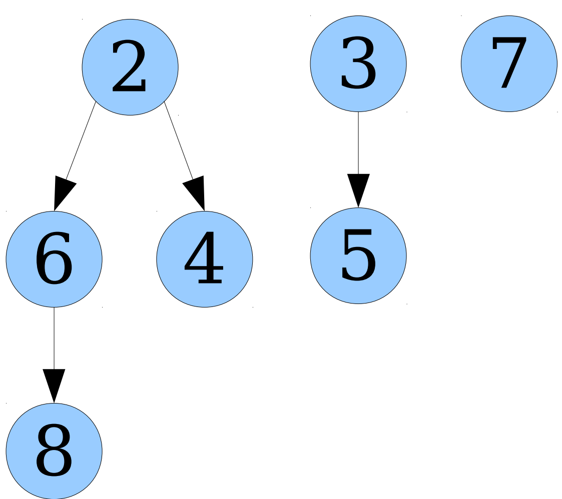
Binomial Trees

- 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, …, 2^{k-1} nodes.



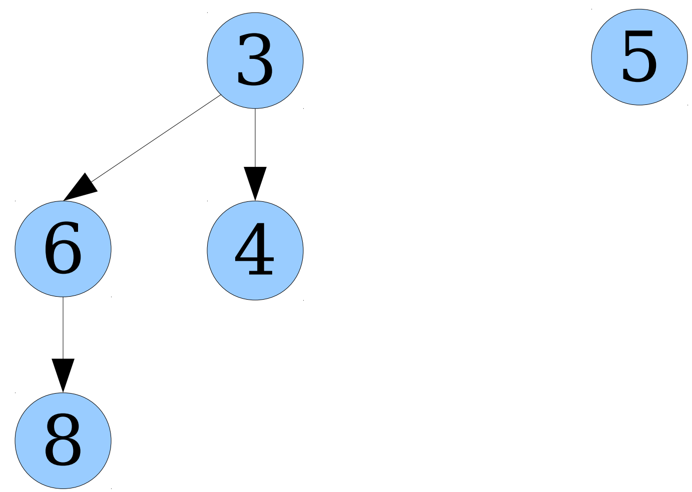
Binomial Trees

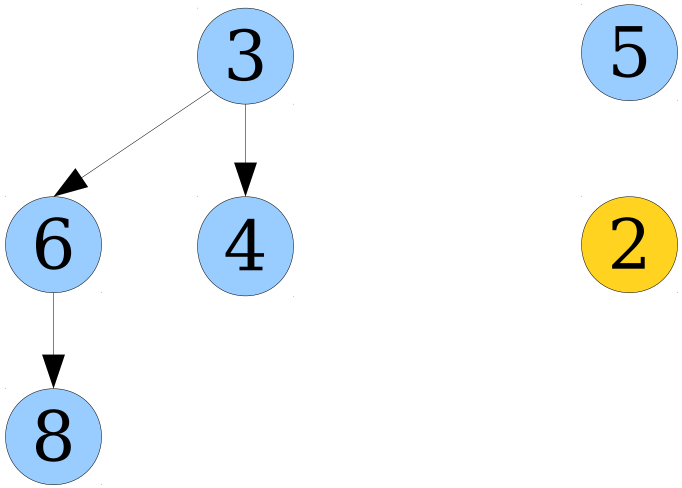
- What properties must our packets have?
  - Size must be a power of two. √
  - Can efficiently fuse packets of the same size. √
  - Can efficiently find the minimum element of each packet. √
  - Can efficiently "fracture" a packet of 2^k nodes into packets of 2^0, 2^1, 2^2, 2^3, …, 2^{k-1} nodes. √

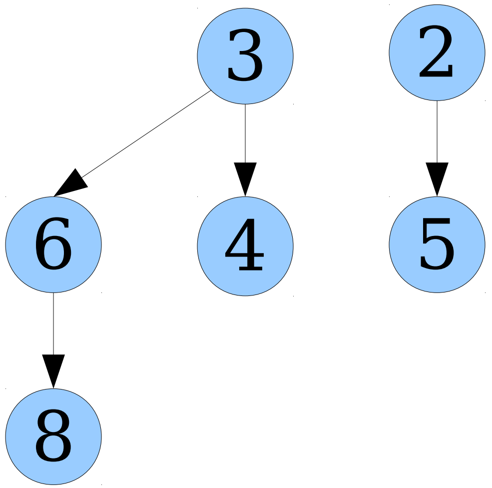


The Binomial Heap

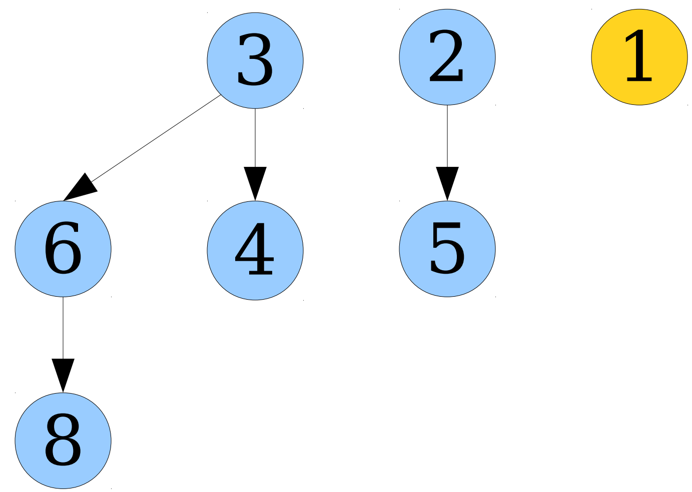
- A ***binomial heap*** is a collection of heap-ordered binomial trees stored in ascending order of size.
- Operations defined as follows:
  - ***meld***(pq_1, pq_2): Use addition to combine all the trees.
    - Fuses O(log *n*) trees. Total time: O(log *n*).
  - *pq*.***enqueue***(*v*, *k*): Meld *pq* 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*).


3
5
6
4
8

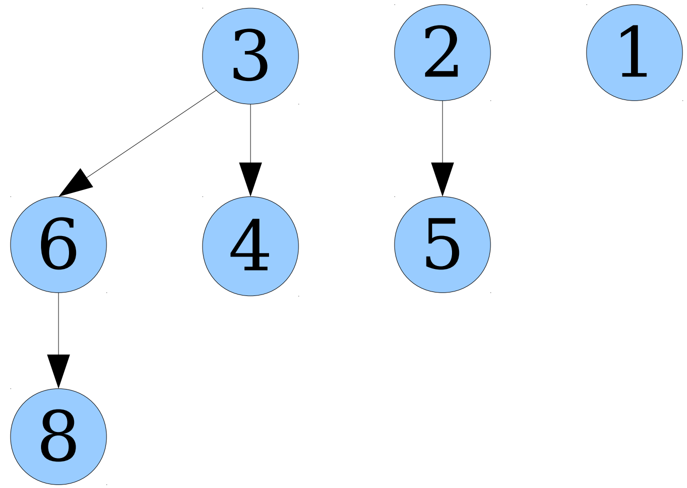

3
5
6
4
2
8

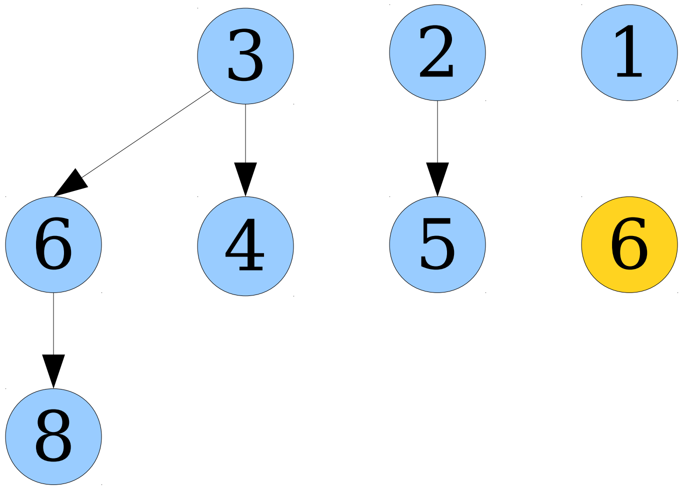


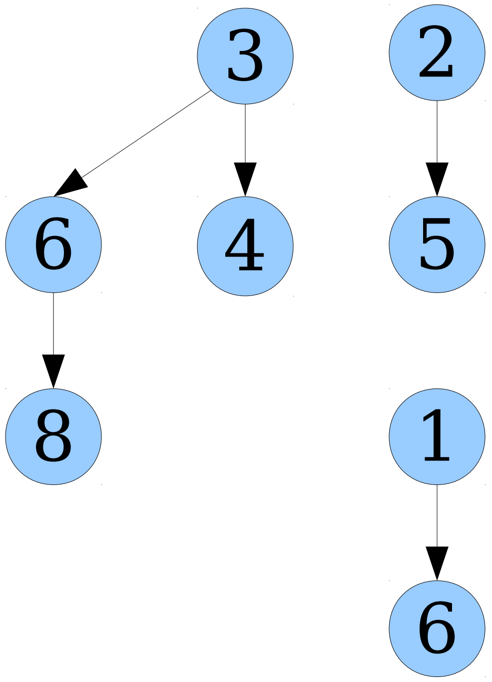
3
2
6
4
5
8

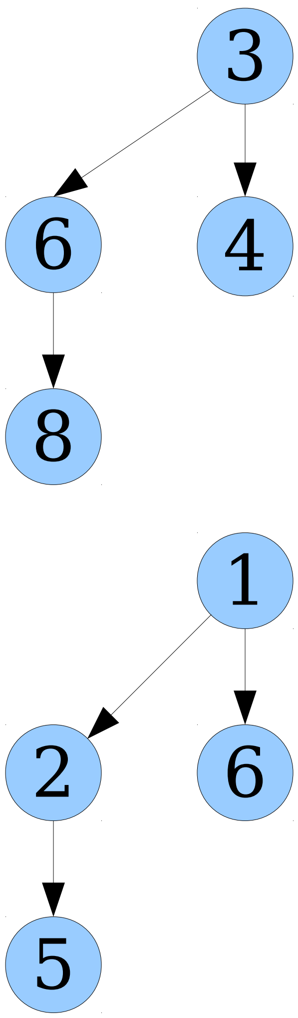


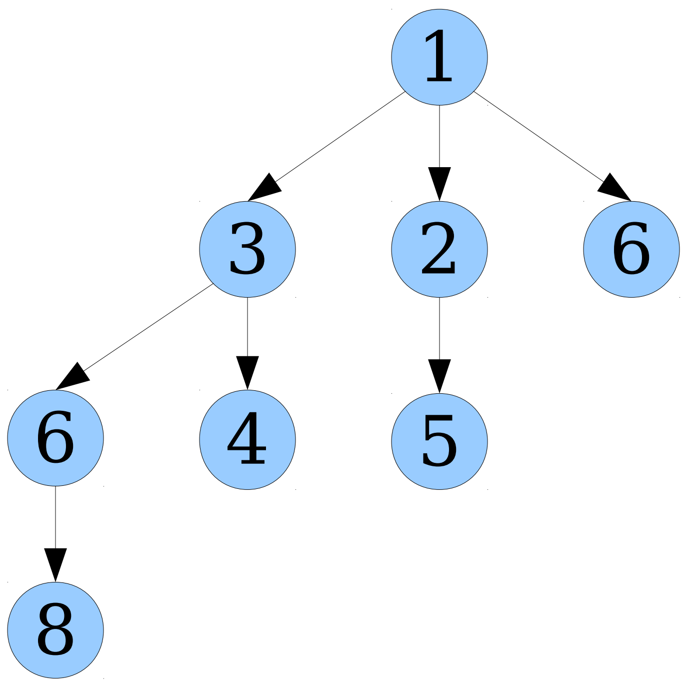
3
2
1
6
4
5
8

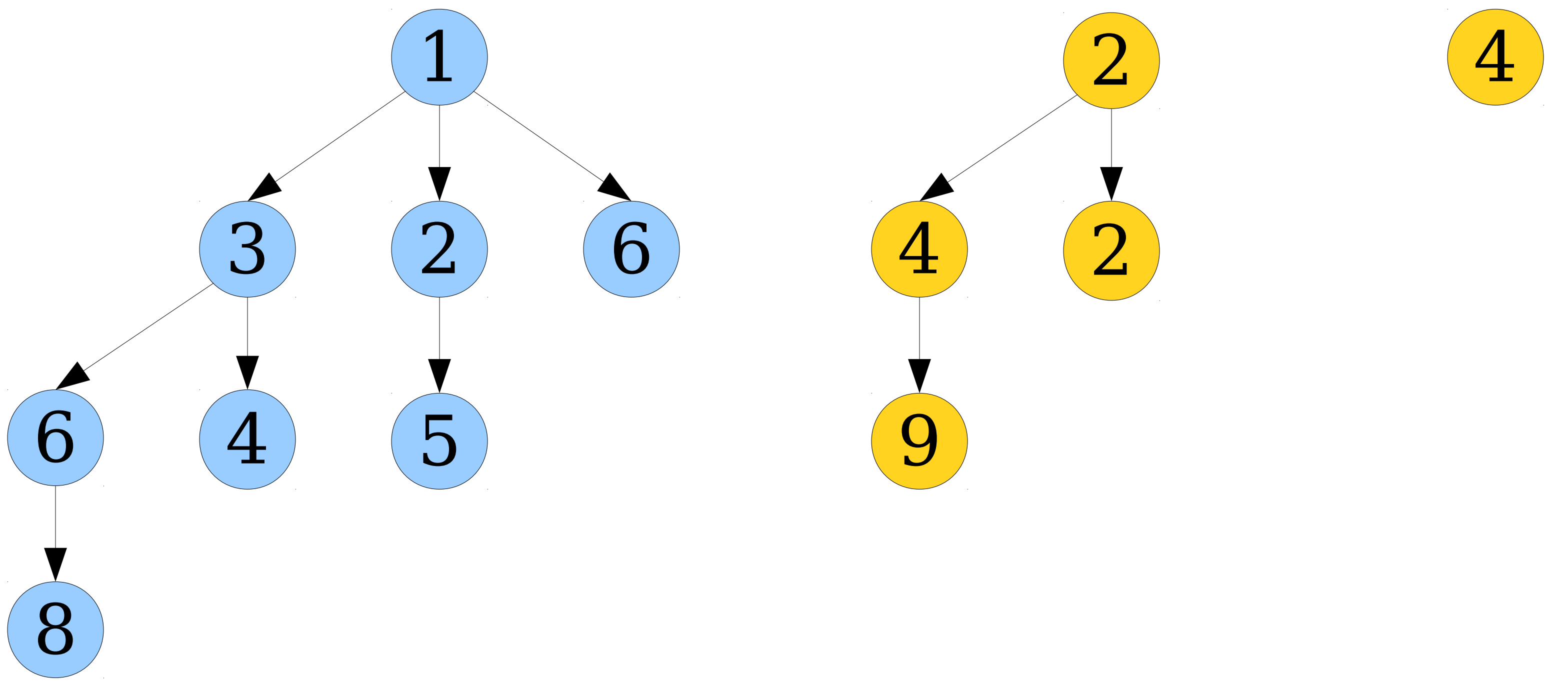

3
2
1
6
4
5
8

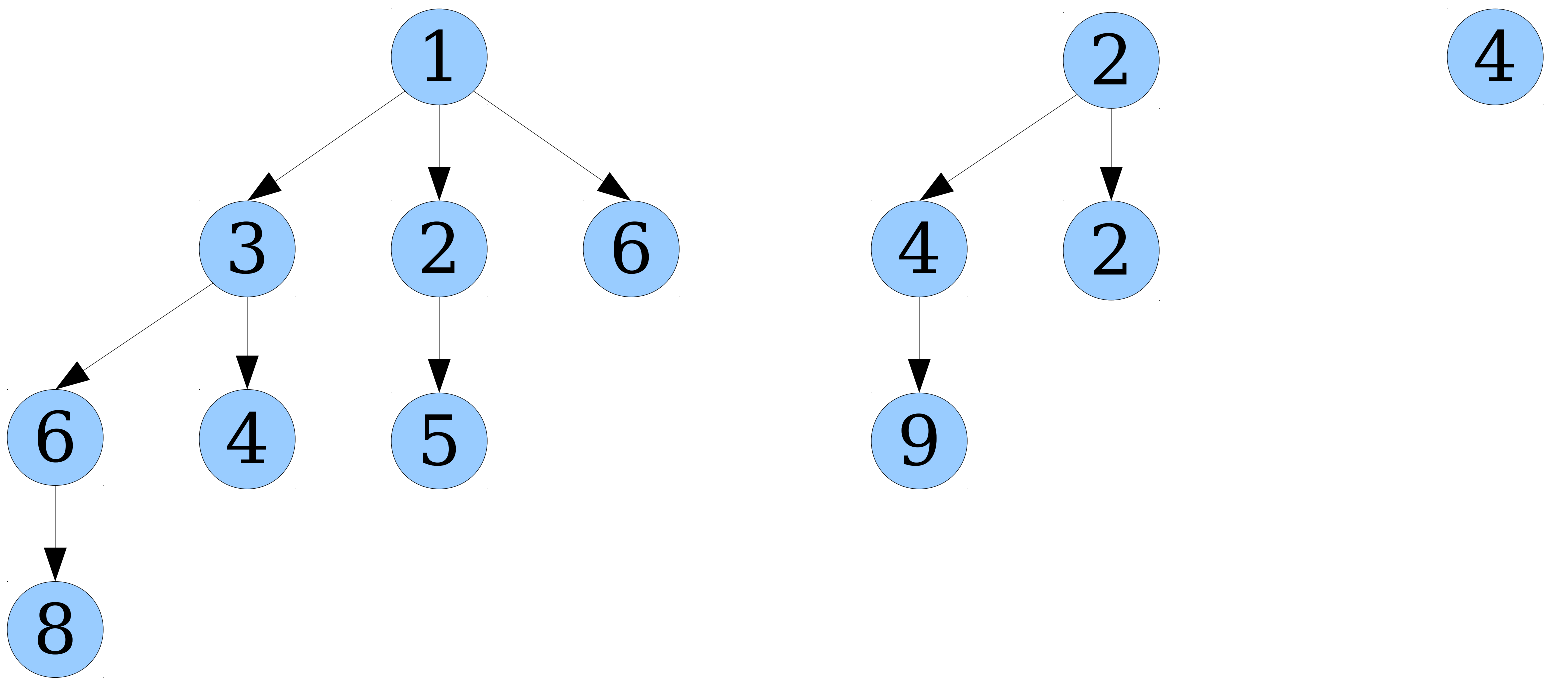

3
2
1
6
4
5
6
8

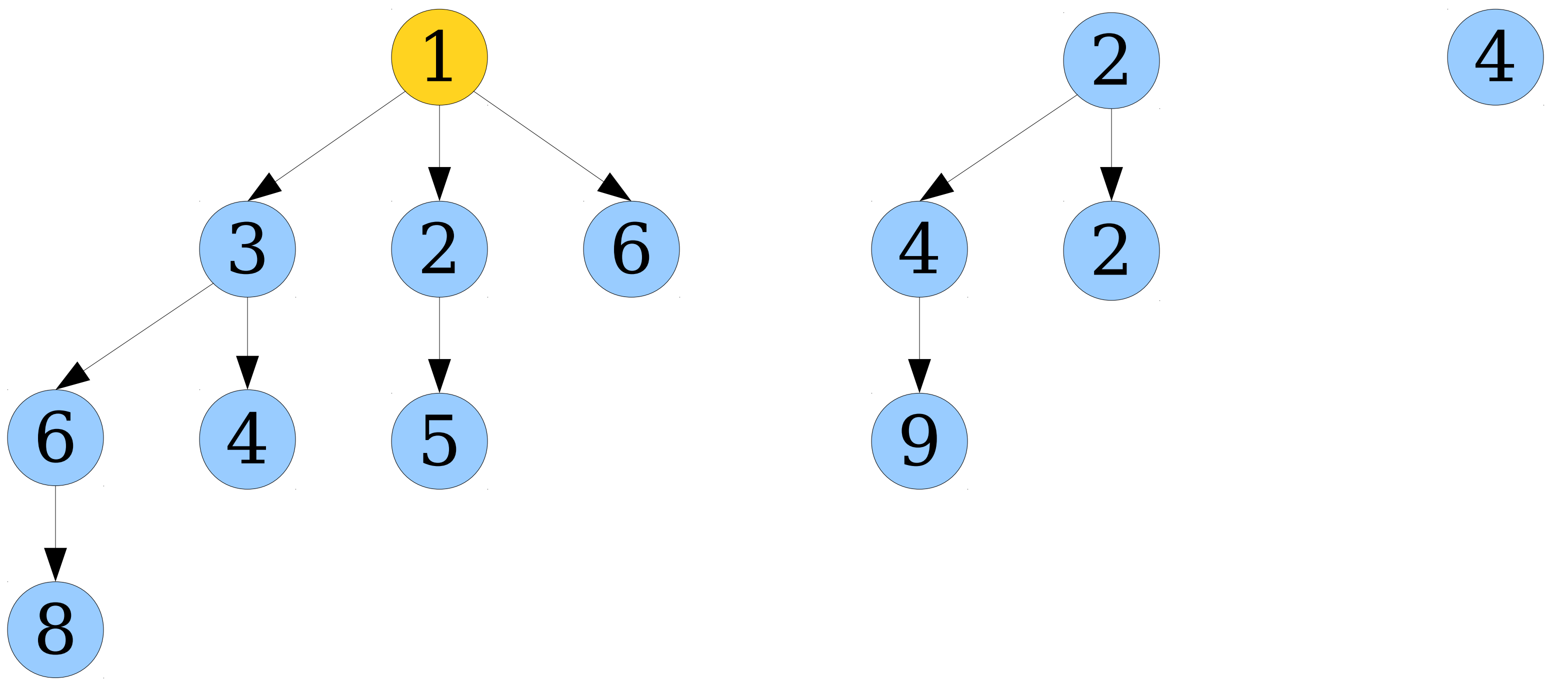

3
2
6
4
5
8
1
6

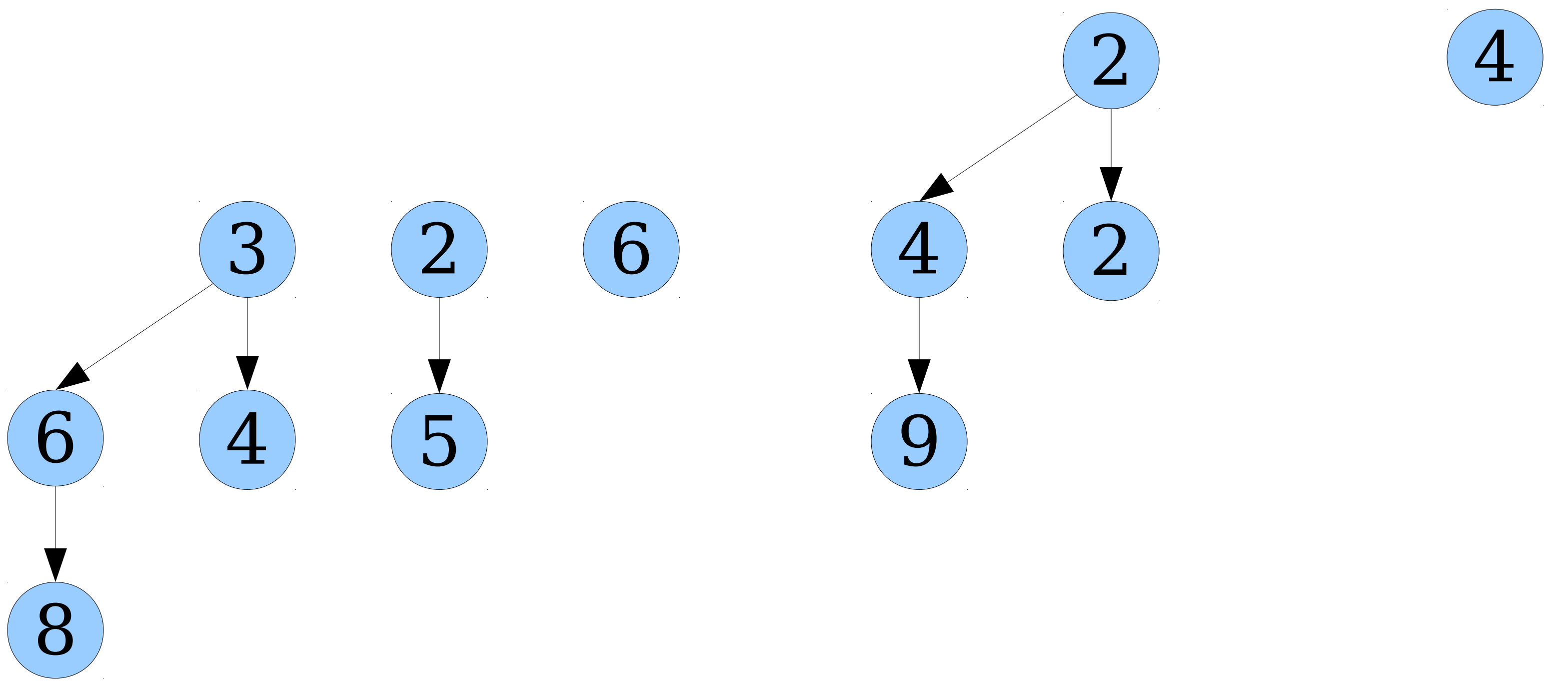

3
6
4
8
1
2
6
5


1
3
2
6
6
4
5
8

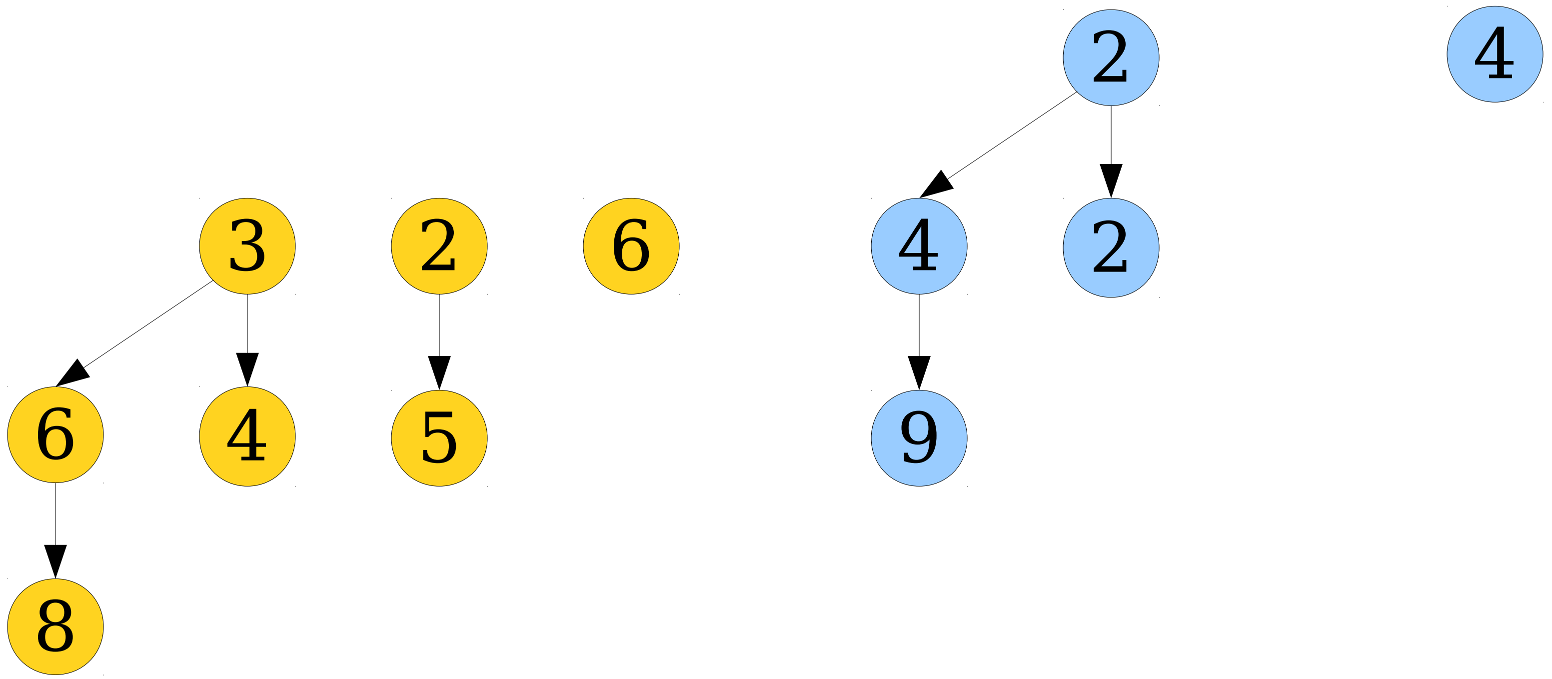

1
3
2
6
6
4
5
8
2
4
2
9
4

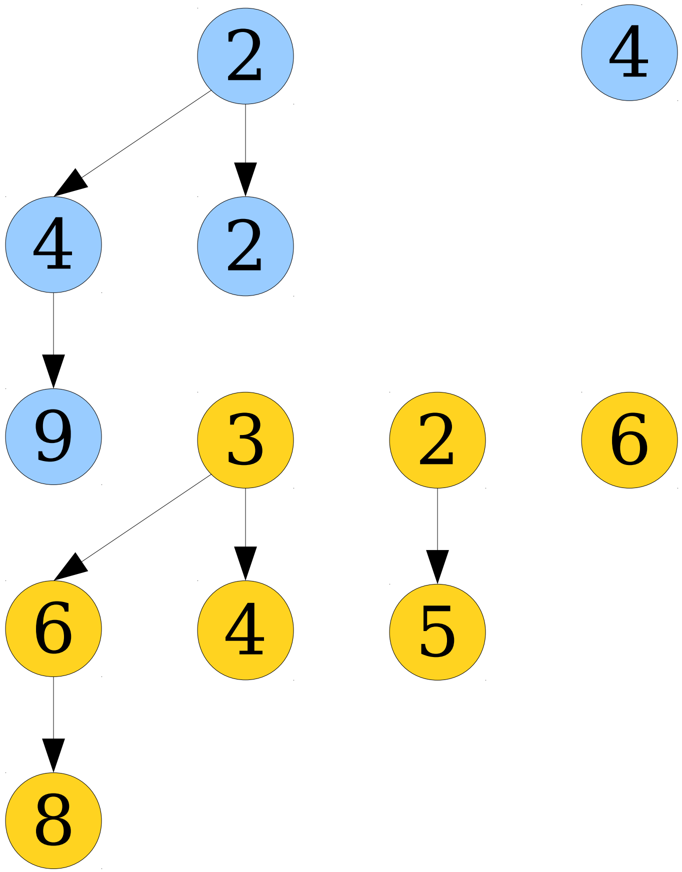

1
3
2
6
6
4
5
8
2
4
2
9
4

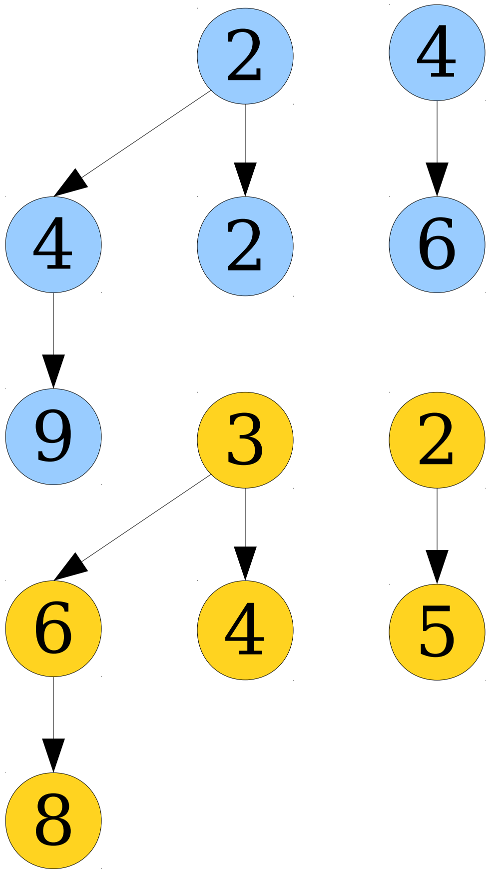

1
3
2
6
6
4
5
8
2
4
2
9
4

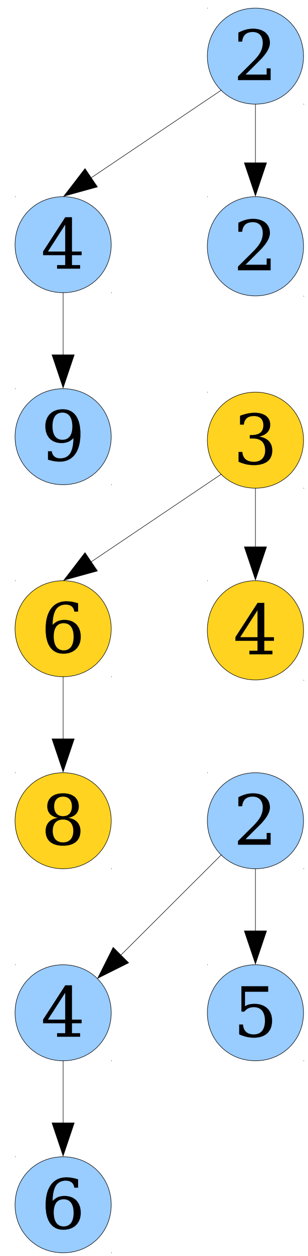


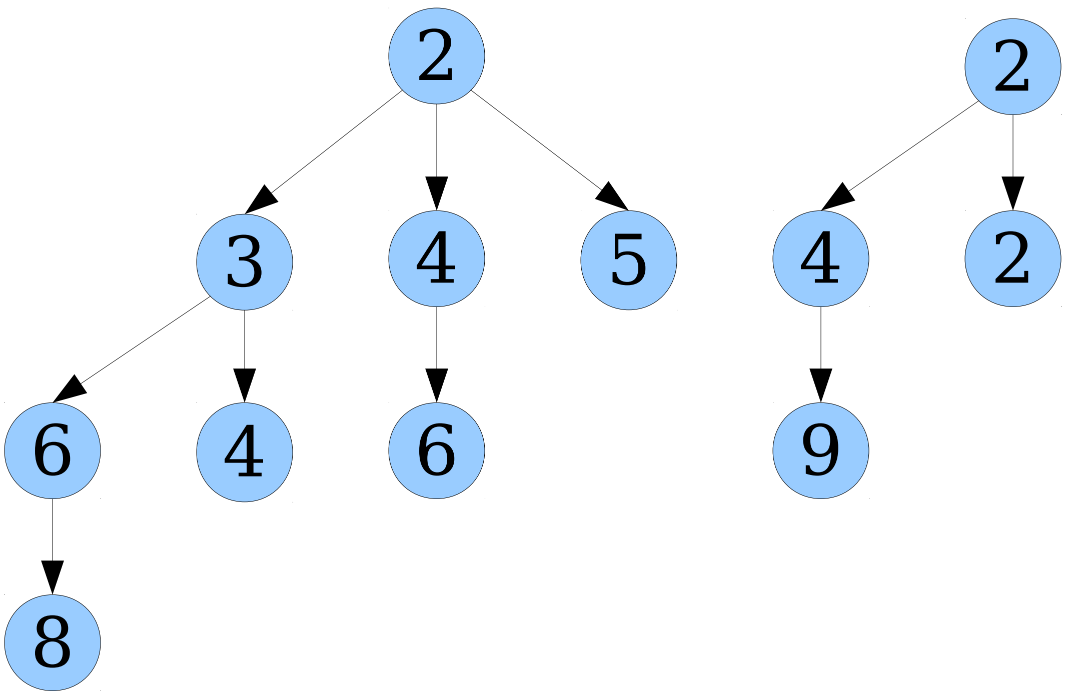
2
4
3
2
6
4
2
6
4
5
9
8


2
4
3
2
6
4
2
6
4
5
9
8


2
4
4
2
9
3
2
6
6
4
5
8

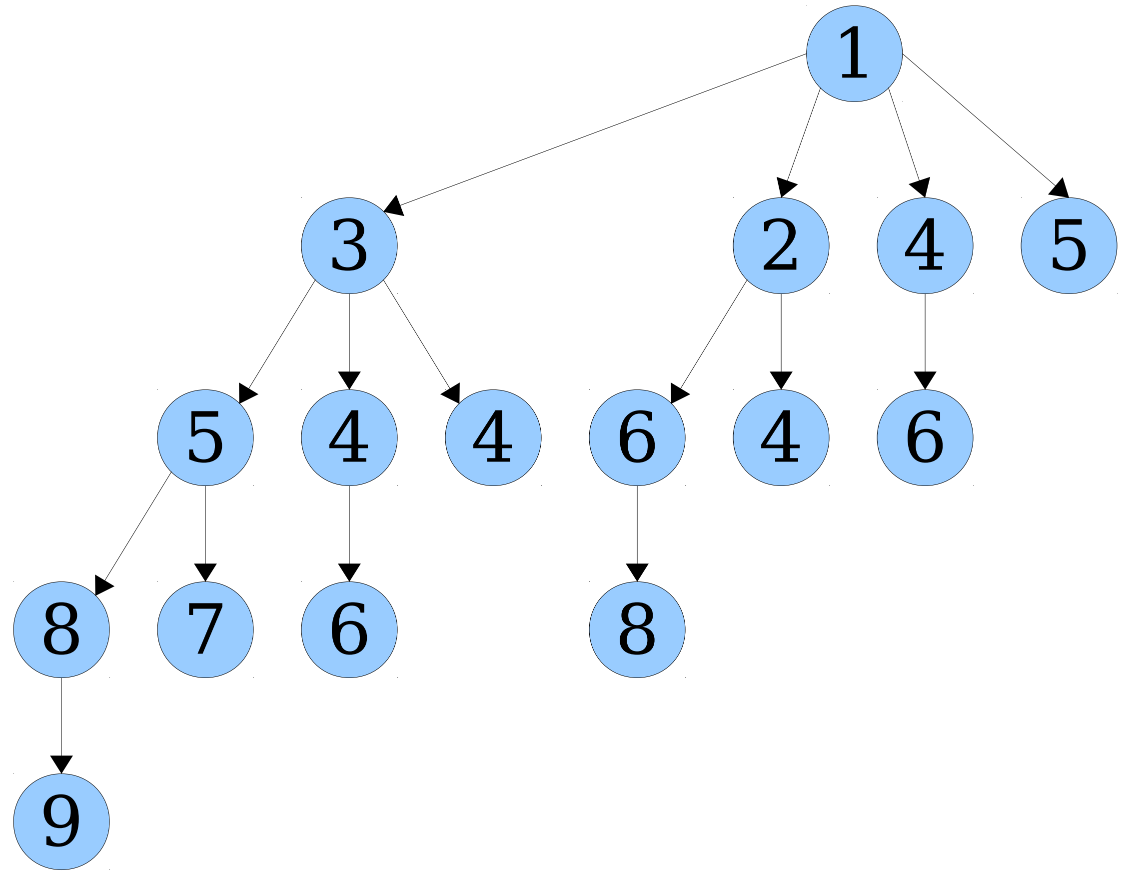

2
4
4
2
6
9
3
2
6
4
5
8


2
4
2
9
3
6
4
8
2
4
5
6


2
3
4
5
6
4
6
8
2
4
2
9

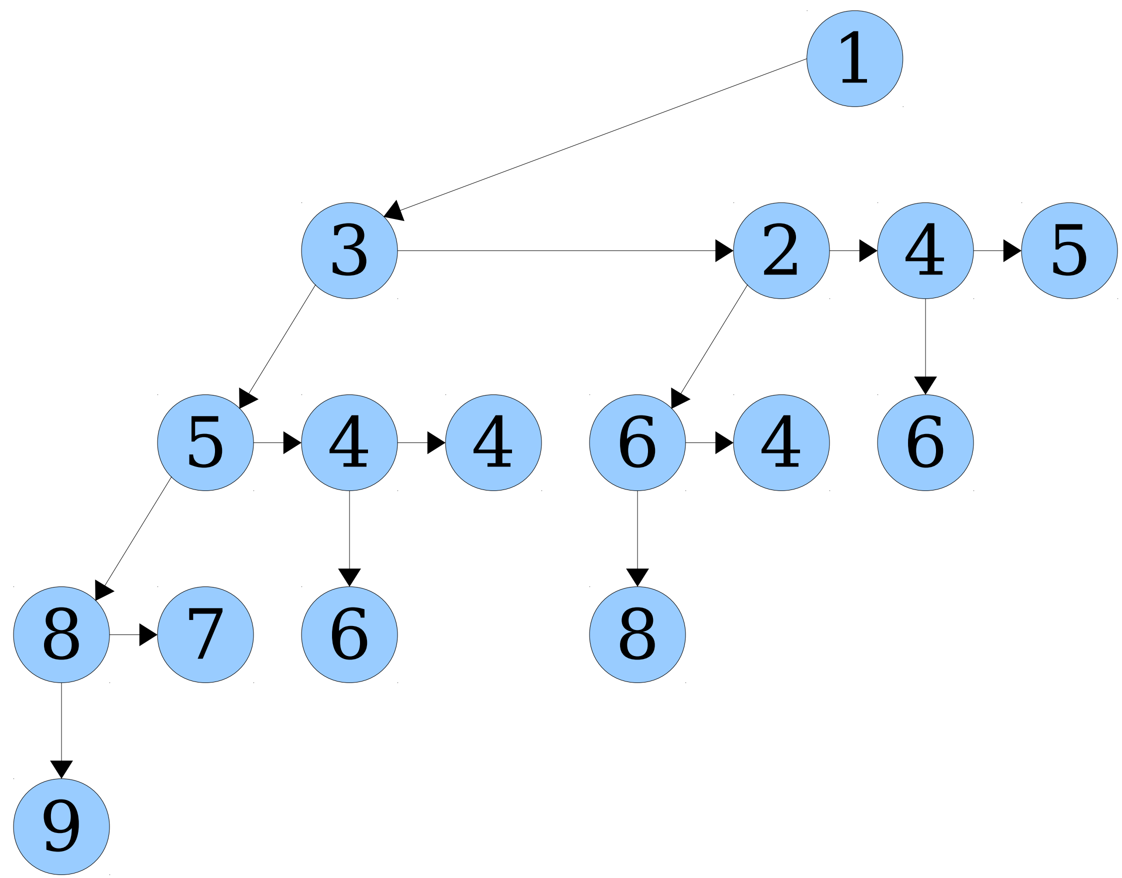
An Issue of Representation

- Binomial trees are *logically* multiway trees, but are typically *implemented* as binary trees.
- We use the ***left-child/right-sibling*** representation.
- Each node's left pointer points to its first child.
- Each node's right pointer points to its next sibling.



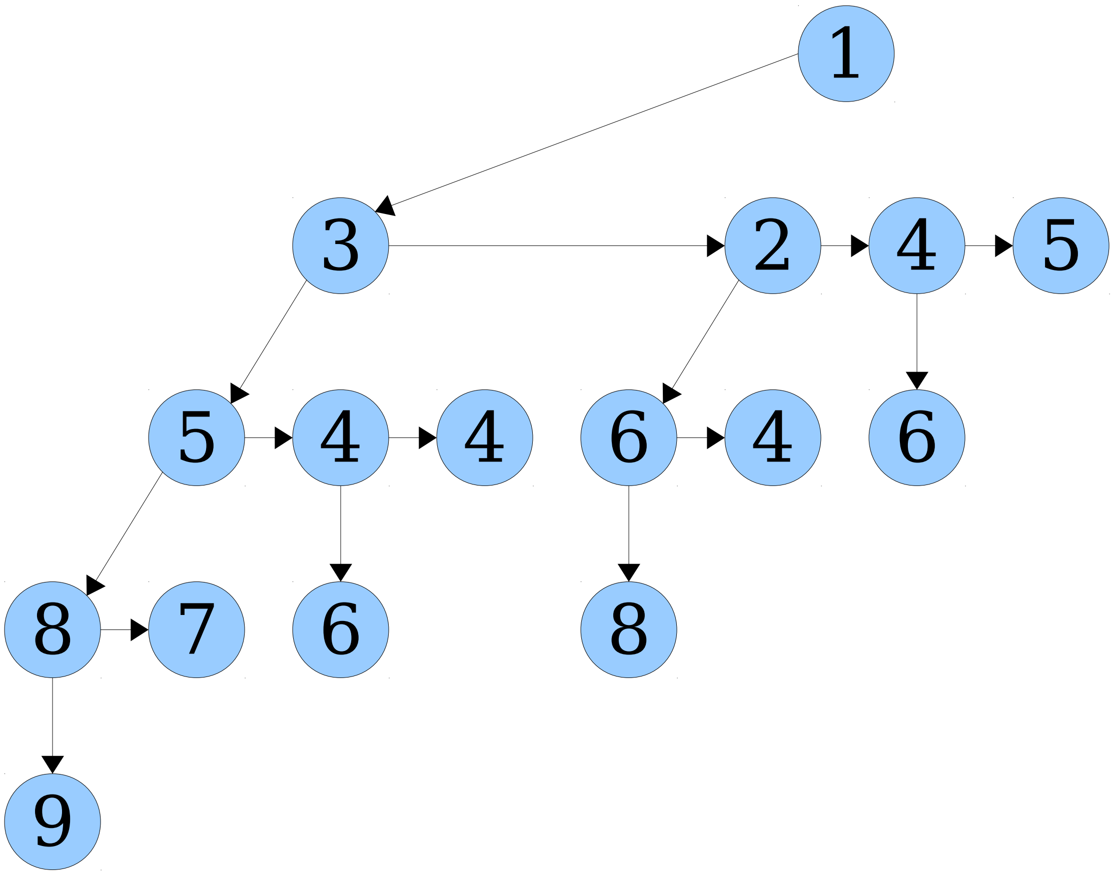
An Issue of Representation

- Binomial trees are *logically* multiway trees, but are typically *implemented* as binary trees.
- We use the ***left-child/right-sibling*** representation.
- Each node's left pointer points to its first child.
- Each node's right pointer points to its next sibling.
- ***Question:*** Why would we do this?



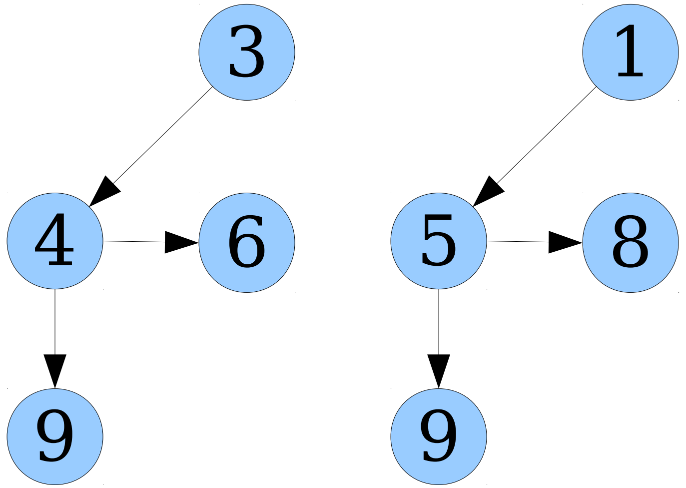
An Issue of Representation

- ***Claim 1:*** If you dive deep into the costs of representing a multiway tree using LC/RS versus a regular array, the LC/RS representation uses less memory.
- You should *definitely* think about this on your own time!



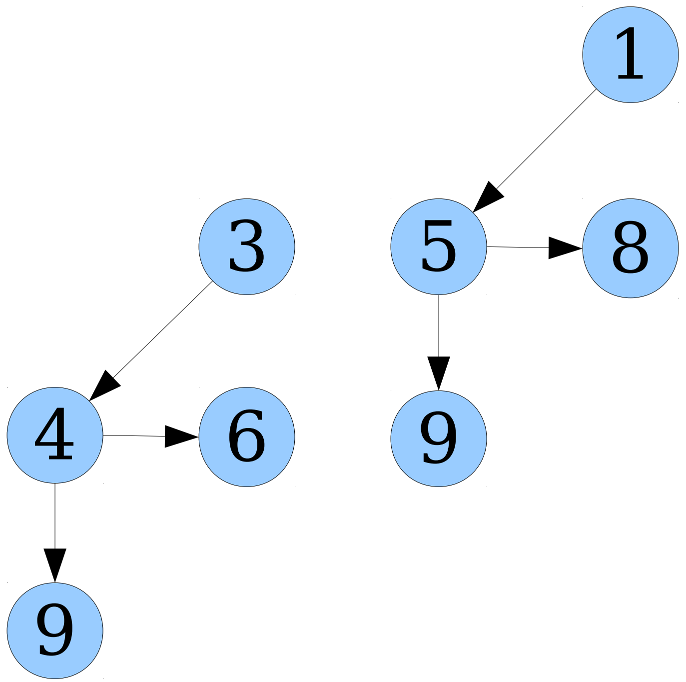
An Issue of Representation

- ***Claim 2:*** There's no runtime cost to using LC/RS for binomial trees.
- We only need to support
  - fusing two trees together, and
  - fracturing a tree into smaller trees.
- These operations are fast and simple in LC/RS.



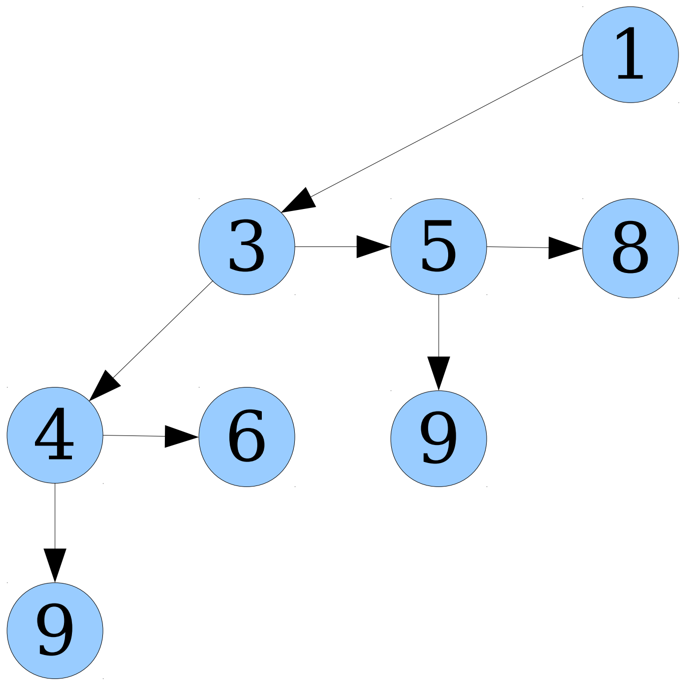
An Issue of Representation

- ***Claim 2:*** There's no runtime cost to using LC/RS for binomial trees.
- We only need to support
  - fusing two trees together, and
  - fracturing a tree into smaller trees.
- These operations are fast and simple in LC/RS.



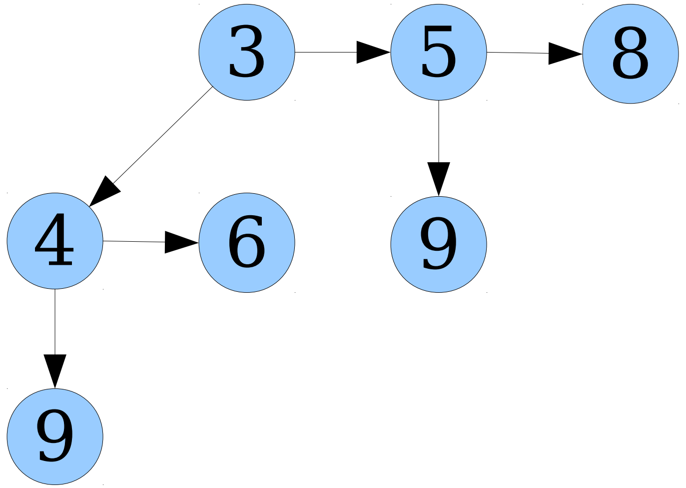
An Issue of Representation

- ***Claim 2:*** There's no runtime cost to using LC/RS for binomial trees.
- We only need to support
  - fusing two trees together, and
  - fracturing a tree into smaller trees.
- These operations are fast and simple in LC/RS.



An Issue of Representation

- ***Claim 2:*** There's no runtime cost to using LC/RS for binomial trees.
- We only need to support
  - fusing two trees together, and
  - fracturing a tree into smaller trees.
- These operations are fast and simple in LC/RS.



Time-Out for Announcements!

Theory AMA

- The CS department is hosting an “ask me anything” event with Moses Charikar and Mary Wootters of our theory group.
  - Event runs on Wednesday, May 8 from 6:00PM – 7:00PM.
- This sounds like a great opportunity for anyone who, like you, is taking more than the theoretical minimum number of theory courses. 😃
- Space is limited; ***RSVP here***!

Another CS Department Announcement

Project Proposals

- As a reminder, final project proposals are due on Thursday at 2:30PM.
  - ***No late periods may be used here***, since we want to assign topics as soon as possible.
- You’ll need to find a group of three people unless you have explicit permission from us.
- Looking for teammates? Use Piazza’s “Search for Teammates” feature, or hang around after class today!

Back to CS166!

A Deeper Look at Binomial Heaps

Analyzing Insertions

- Each ***enqueue*** into a binomial heap takes time O(log *n*), since we have to meld the new node into the rest of the trees.
- However, it turns out that the *amortized* cost of an insertion is lower in the case where we do a series of *n* insertions.

Adding One

- Suppose we want to execute n++ on the binary representation of n.
- Do the following:
  - Find the longest span of 1's at the right side of n.
  - Flip those 1's to 0's.
  - Set the preceding bit to 1.

1 0 1 1 0

Adding One

- Suppose we want to execute n++ on the binary representation of n.
- Do the following:
  - Find the longest span of 1's at the right side of n.
  - Flip those 1's to 0's.
  - Set the preceding bit to 1.

1 0 1 1 1

Adding One

- Suppose we want to execute n++ on the binary representation of n.
- Do the following:
  - Find the longest span of 1's at the right side of n.
  - Flip those 1's to 0's.
  - Set the preceding bit to 1.

1 0 1 1 1

Adding One

- Suppose we want to execute n++ on the binary representation of n.
- Do the following:
  - Find the longest span of 1's at the right side of n.
  - Flip those 1's to 0's.
  - Set the preceding bit to 1.

1 1 0 0 0

Adding One

- Suppose we want to execute n++ on the binary representation of n.
- Do the following:
  - Find the longest span of 1's at the right side of n.
  - Flip those 1's to 0's.
  - Set the preceding bit to 1.

1 1 0 0 0

Adding One

- Suppose we want to execute n++ on the binary representation of n.
- Do the following:
  - Find the longest span of 1's at the right side of n.
  - Flip those 1's to 0's.
  - Set the preceding bit to 1.

1 1 0 0 1

Adding One

- Suppose we want to execute n++ on the binary representation of n.
- Do the following:
  - Find the longest span of 1's at the right side of n.
  - Flip those 1's to 0's.
  - Set the preceding bit to 1.

1	1	0	0	1

Adding One

- Suppose we want to execute n++ on the binary representation of n.
- Do the following:
  - Find the longest span of 1's at the right side of n.
  - Flip those 1's to 0's.
  - Set the preceding bit to 1.

$$1 \quad 1 \quad 0 \quad 1 \quad 0$$

Adding One

- Suppose we want to execute n++ on the binary representation of n.
- Do the following:
    - Find the longest span of 1's at the right side of n.
    - Flip those 1's to 0's.
    - Set the preceding bit to 1.
- The runtime is $\Theta(b)$, where b is the number of bits flipped. However:
    - we usually don't have to flip many bits, and
    - when we do, it's going to be a while before we have to do it again.

An Amortized Analysis

- ***Claim:*** Starting at zero, the amortized cost of adding one to the total is O(1).
- ***Idea:*** Use as a potential function the number of 1's in the number.

$\boxed{\Phi = 0}$ 0 0 0 0 0

An Amortized Analysis

- ***Claim:*** Starting at zero, the amortized cost of adding one to the total is O(1).
- ***Idea:*** Use as a potential function the number of 1's in the number.

$\boxed{\Phi = 1}$ 0 0 0 0 1

An Amortized Analysis

- ***Claim:*** Starting at zero, the amortized cost of adding one to the total is O(1).
- ***Idea:*** Use as a potential function the number of 1's in the number.

$\Phi = 1$ 0 0 0 0 1

Actual cost: 1
$\Delta\Phi$: +1

Amortized cost: **2**

An Amortized Analysis

- ***Claim:*** Starting at zero, the amortized cost of adding one to the total is O(1).
- ***Idea:*** Use as a potential function the number of 1's in the number.

$\Phi = 1$ 0   0   0   0   1

An Amortized Analysis

- ***Claim:*** Starting at zero, the amortized cost of adding one to the total is O(1).
- ***Idea:*** Use as a potential function the number of 1's in the number.

$\Phi = 0$

0	0	0	0	0

An Amortized Analysis

- ***Claim:*** Starting at zero, the amortized cost of adding one to the total is O(1).
- ***Idea:*** Use as a potential function the number of 1's in the number.

$\Phi = 0$

0 0 0 0 0

An Amortized Analysis

- ***Claim:*** Starting at zero, the amortized cost of adding one to the total is O(1).
- ***Idea:*** Use as a potential function the number of 1's in the number.

$\Phi = 1$ 0 0 0 1 0

An Amortized Analysis

- ***Claim:*** Starting at zero, the amortized cost of adding one to the total is O(1).
- ***Idea:*** Use as a potential function the number of 1's in the number.

$\boxed{\Phi = 1}$ 0 0 0 1 0

An Amortized Analysis

- ***Claim:*** Starting at zero, the amortized cost of adding one to the total is O(1).
- ***Idea:*** Use as a potential function the number of 1's in the number.

$\Phi = 1$ 0   0   0   1   0

Actual cost: 2
$\Delta\Phi$: 0

Amortized cost: **2**

An Amortized Analysis

- ***Claim:*** Starting at zero, the amortized cost of adding one to the total is O(1).
- ***Idea:*** Use as a potential function the number of 1's in the number.

$\Phi = 2$ 0 0 0 1 1

An Amortized Analysis

- ***Claim:*** Starting at zero, the amortized cost of adding one to the total is O(1).
- ***Idea:*** Use as a potential function the number of 1's in the number.

$\Phi = 2$ 0 0 0 1 1

Actual cost: 1
$\Delta\Phi$: 1

Amortized cost: **2**

An Amortized Analysis

- ***Claim:*** Starting at zero, the amortized cost of adding one to the total is O(1).
- ***Idea:*** Use as a potential function the number of 1's in the number.

$\Phi = 2$

0	0	0	1	1

An Amortized Analysis

- ***Claim:*** Starting at zero, the amortized cost of adding one to the total is O(1).
- ***Idea:*** Use as a potential function the number of 1's in the number.

$\Phi = 1$	0	0	0	1	0

An Amortized Analysis

- ***Claim:*** Starting at zero, the amortized cost of adding one to the total is O(1).
- ***Idea:*** Use as a potential function the number of 1's in the number.

$\Phi = 1$ 0 0 0 1 0

An Amortized Analysis

- ***Claim:*** Starting at zero, the amortized cost of adding one to the total is O(1).
- ***Idea:*** Use as a potential function the number of 1's in the number.

$\Phi = 0$ 0 0 0 0 0

An Amortized Analysis

- ***Claim:*** Starting at zero, the amortized cost of adding one to the total is O(1).
- ***Idea:*** Use as a potential function the number of 1's in the number.

$\Phi = 0$

0	0	0	0	0

An Amortized Analysis

- ***Claim:*** Starting at zero, the amortized cost of adding one to the total is O(1).
- ***Idea:*** Use as a potential function the number of 1's in the number.

$\boxed{\Phi = 1}$ 0 0 1 0 0

An Amortized Analysis

- ***Claim:*** Starting at zero, the amortized cost of adding one to the total is O(1).
- ***Idea:*** Use as a potential function the number of 1's in the number.

$\boxed{\Phi = 1}$ 0 0 1 0 0

An Amortized Analysis

- ***Claim:*** Starting at zero, the amortized cost of adding one to the total is O(1).
- ***Idea:*** Use as a potential function the number of 1's in the number.

$\Phi = 1$ 0 0 1 0 0

Actual cost: 3
$\Delta\Phi$: -1

Amortized cost: **2**

Properties of Binomial Heaps

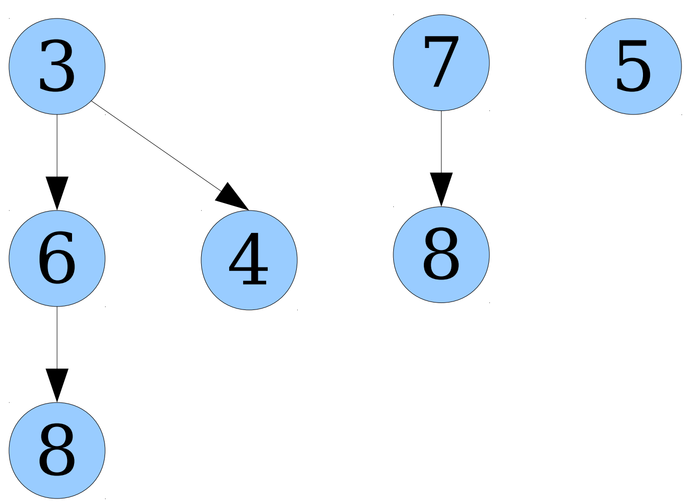
- Starting with an empty binomial heap, the amortized cost of each insertion into the heap is O(1), assuming there are no deletions.
- ***Rationale:*** Binomial heap operations are isomorphic to integer arithmetic.
- Since the amortized cost of incrementing a binary counter starting at zero is O(1), the amortized cost of enqueuing into an initially empty binomial heap is O(1).

Binomial vs Binary Heaps

- Interesting comparison:
  - The cost of inserting n elements into a binary heap, one after the other, is $\Theta(n \log n)$ in the worst-case.
  - If n is known in advance, a binary heap can be constructed out of n elements in time $\Theta(n)$.
  - The cost of inserting n elements into a binomial heap, one after the other, is $\Theta(n)$, even if n is not known in advance!

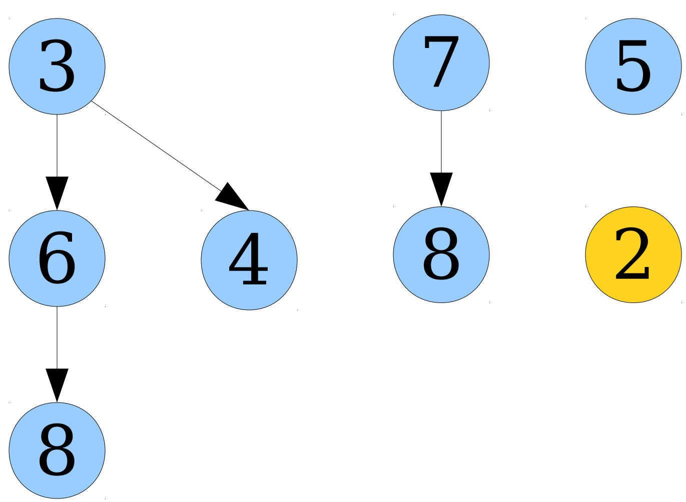
A Catch

- This amortized time bound does not hold if ***enqueue*** and ***extract-min*** are intermixed.
- ***Intuition:*** Can force expensive insertions to happen repeatedly.



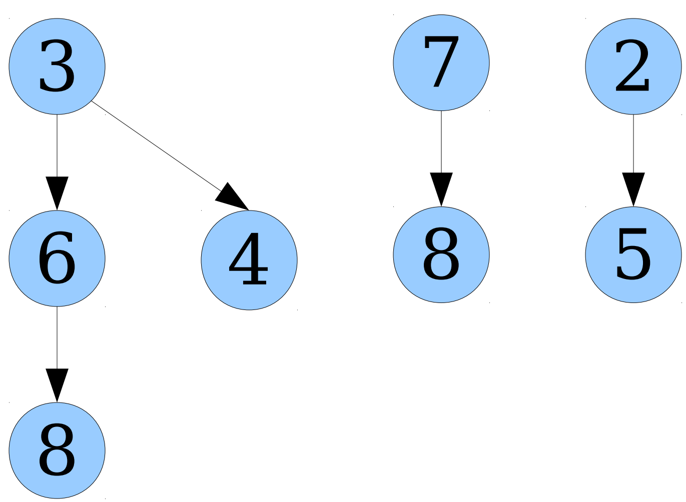
A Catch

- This amortized time bound does not hold if ***enqueue*** and ***extract-min*** are intermixed.
- ***Intuition:*** Can force expensive insertions to happen repeatedly.



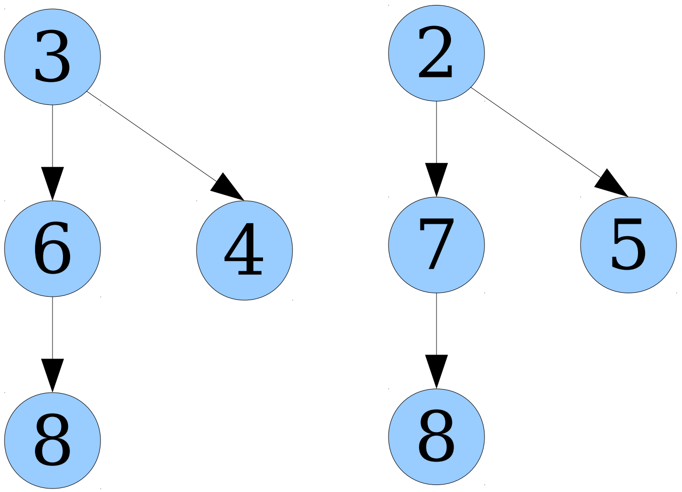
A Catch

- This amortized time bound does not hold if ***enqueue*** and ***extract-min*** are intermixed.
- ***Intuition:*** Can force expensive insertions to happen repeatedly.



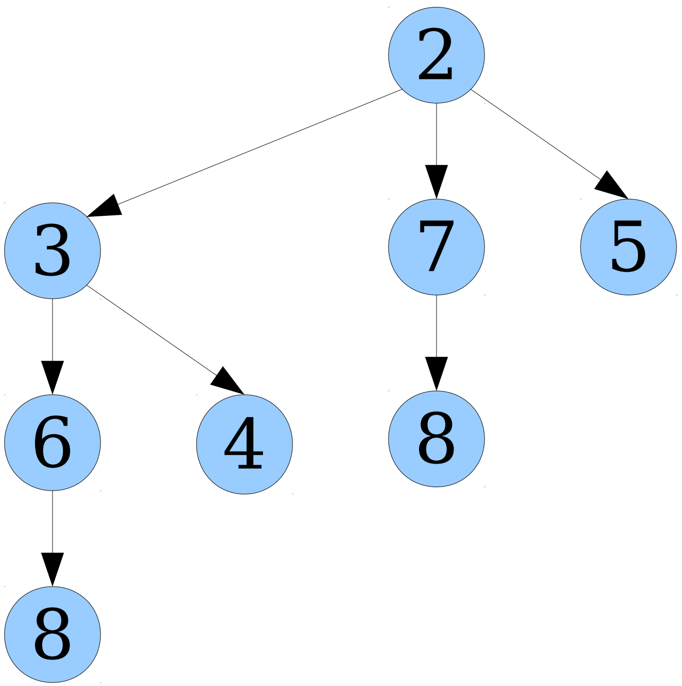
A Catch

- This amortized time bound does not hold if ***enqueue*** and ***extract-min*** are intermixed.
- ***Intuition:*** Can force expensive insertions to happen repeatedly.



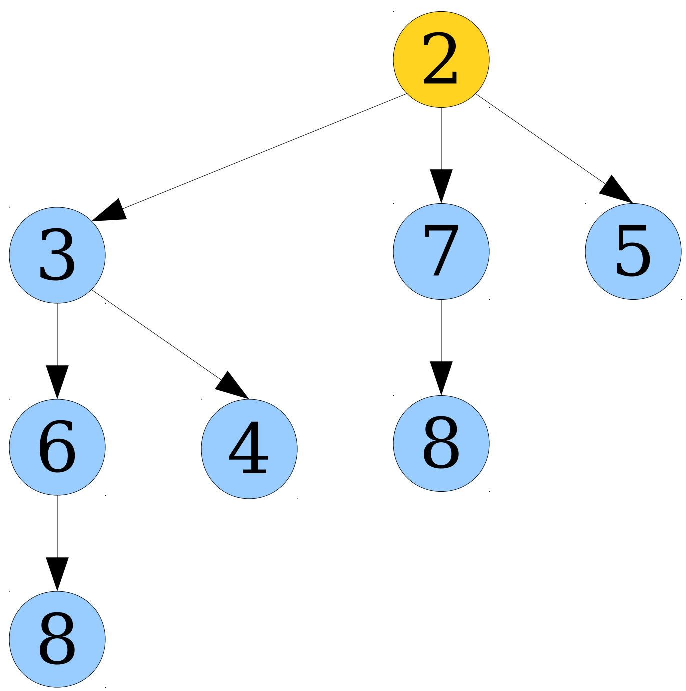
A Catch

- This amortized time bound does not hold if ***enqueue*** and ***extract-min*** are intermixed.
- ***Intuition:*** Can force expensive insertions to happen repeatedly.



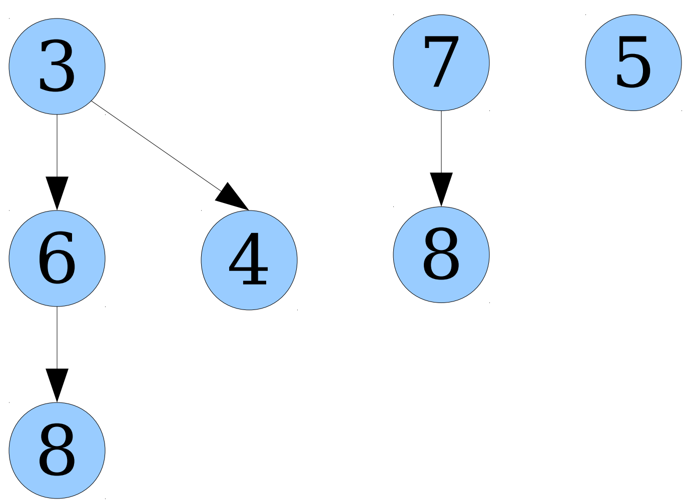
A Catch

- This amortized time bound does not hold if ***enqueue*** and ***extract-min*** are intermixed.
- ***Intuition:*** Can force expensive insertions to happen repeatedly.



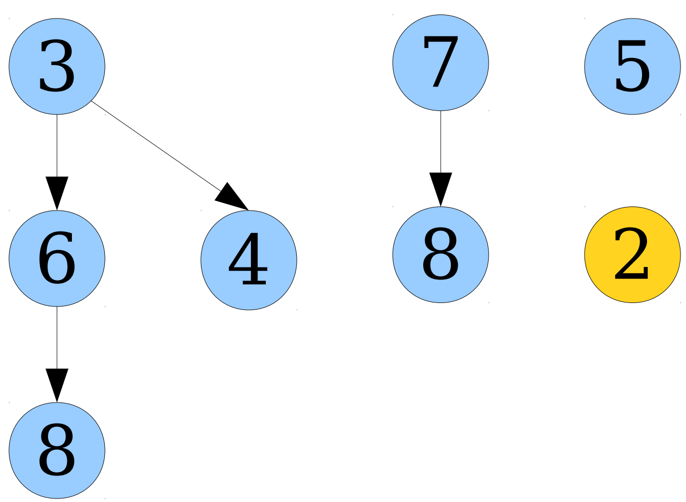
A Catch

- This amortized time bound does not hold if ***enqueue*** and ***extract-min*** are intermixed.
- ***Intuition:*** Can force expensive insertions to happen repeatedly.



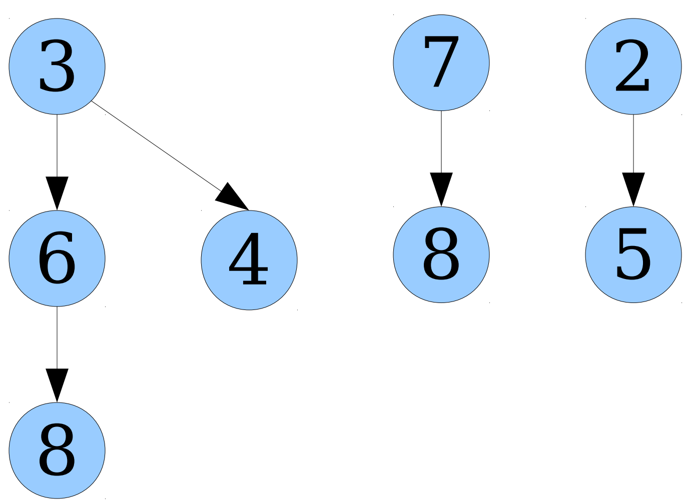
A Catch

- This amortized time bound does not hold if ***enqueue*** and ***extract-min*** are intermixed.
- ***Intuition:*** Can force expensive insertions to happen repeatedly.



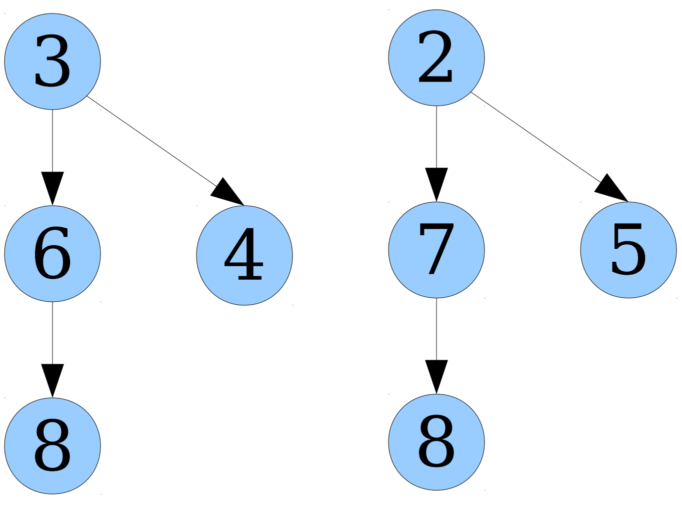
A Catch

- This amortized time bound does not hold if ***enqueue*** and ***extract-min*** are intermixed.
- ***Intuition:*** Can force expensive insertions to happen repeatedly.



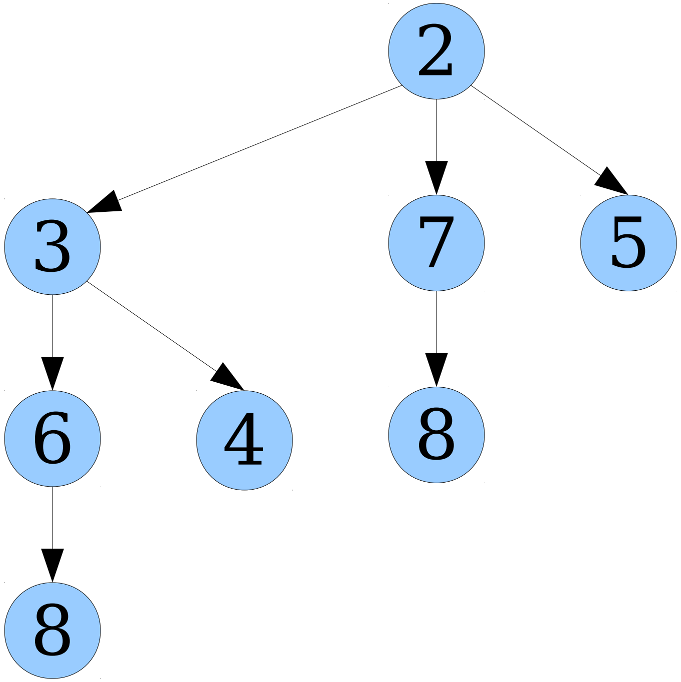
A Catch

- This amortized time bound does not hold if ***enqueue*** and ***extract-min*** are intermixed.
- ***Intuition:*** Can force expensive insertions to happen repeatedly.



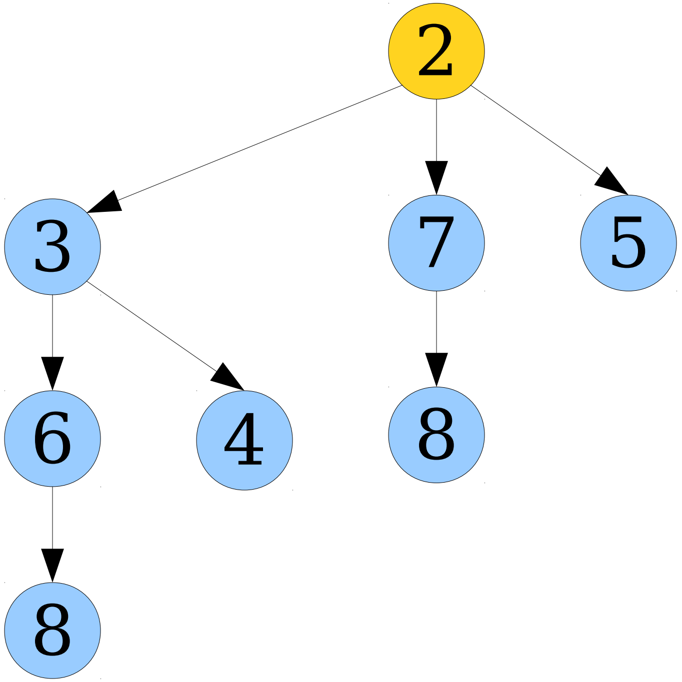
A Catch

- This amortized time bound does not hold if ***enqueue*** and ***extract-min*** are intermixed.
- ***Intuition:*** Can force expensive insertions to happen repeatedly.



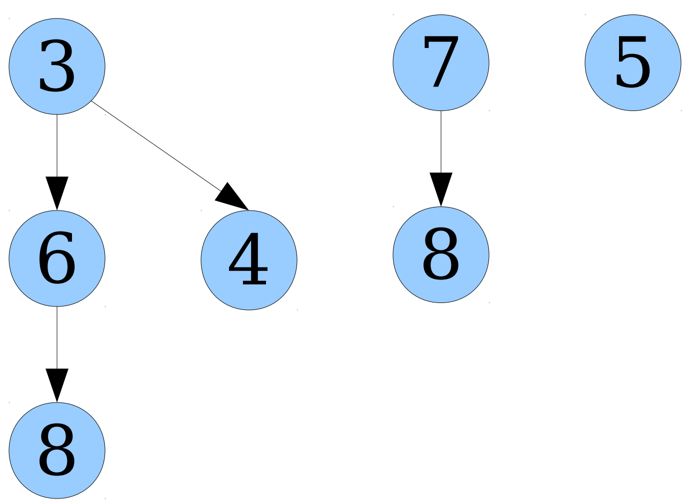
A Catch

- This amortized time bound does not hold if ***enqueue*** and ***extract-min*** are intermixed.
- ***Intuition:*** Can force expensive insertions to happen repeatedly.



A Catch

- This amortized time bound does not hold if ***enqueue*** and ***extract-min*** are intermixed.
- ***Intuition:*** Can force expensive insertions to happen repeatedly.



A Catch

- This amortized time bound does not hold if ***enqueue*** and ***extract-min*** are intermixed.
- ***Intuition:*** Can force expensive insertions to happen repeatedly.
- ***Important lesson:*** The amortized cost of an operation has to be analyzed in the context of the whole data structure, not just that operation itself.

Question: Can we make insertions take amortized time O(1), regardless of whether we intermix them with deletions?

Where's the Cost?

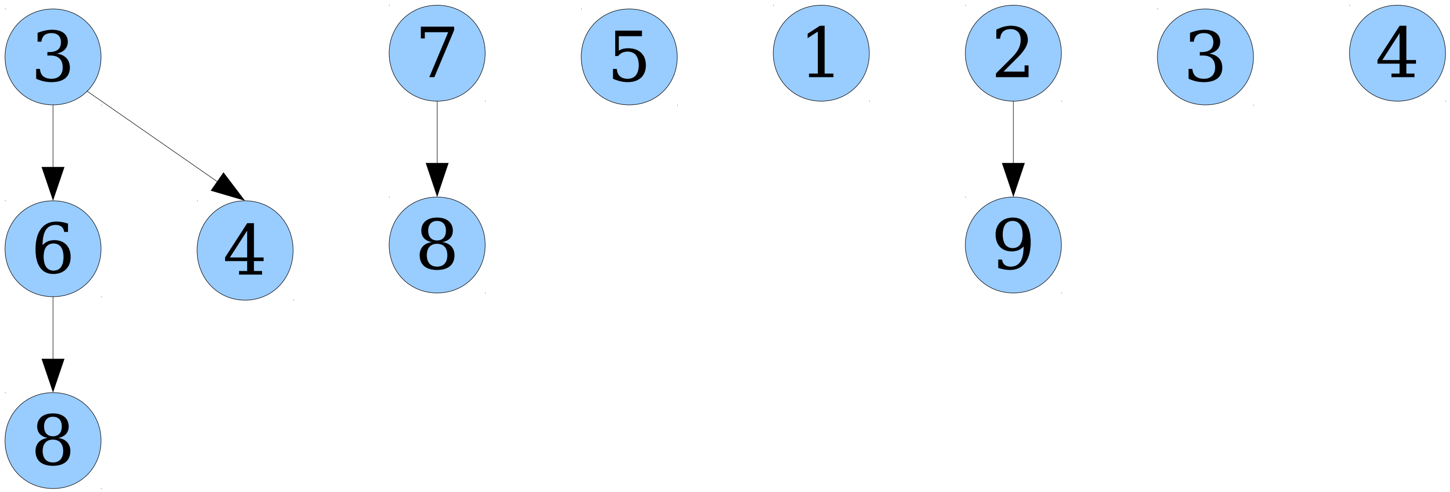
- Why does ***enqueue*** take time O(log *n*)?
- ***Answer***: May have to combine together O(log *n*) different binomial trees together into a single tree.
- ***New Question***: What happens if we don't combine trees together?
- That is, what if we just add a new singleton tree to the list?

Lazy Melding

- More generally, consider the following lazy melding approach:

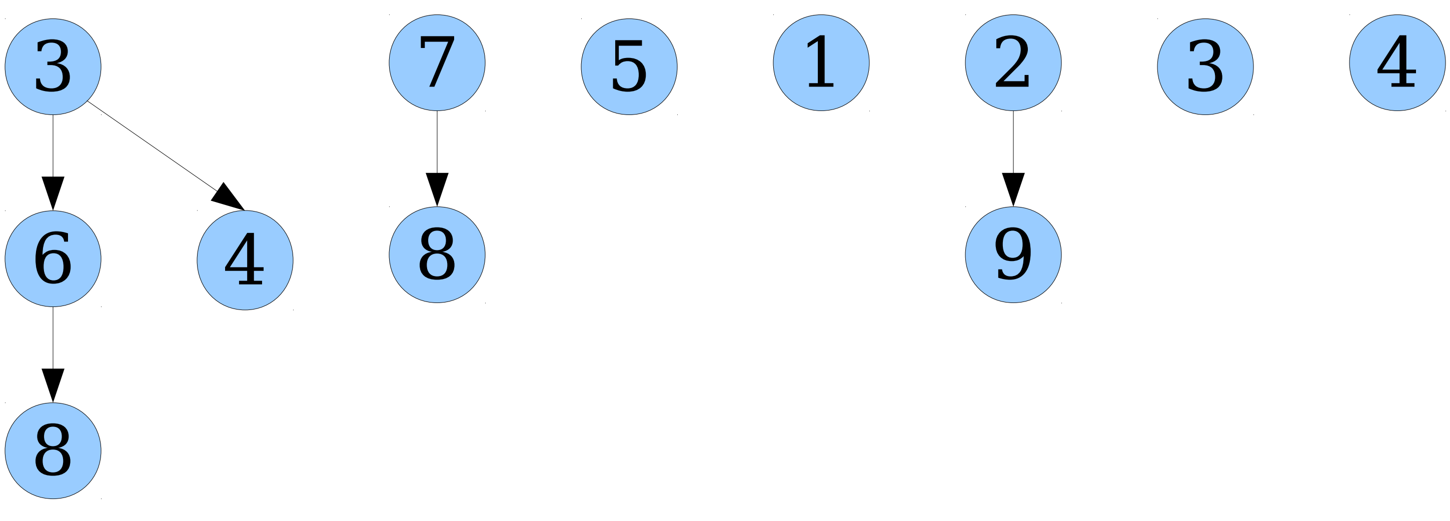
To meld together two binomial heaps, just combine the two sets of trees together.

- If we assume the trees are stored in doubly-linked lists, this can be done in time O(1).



The Catch: Part One

- When we use eager melding, the number of trees is O(log n).
- Therefore, ***find-min*** runs in time O(log n).
- ***Problem:*** ***find-min*** no longer runs in time O(log n) because there can be Θ(n) trees.

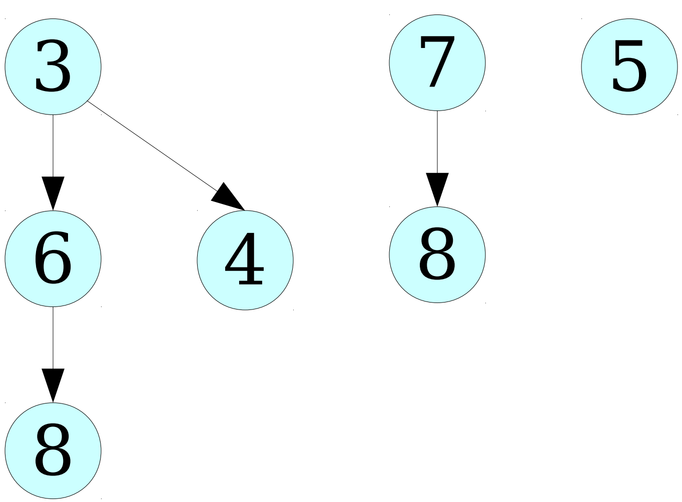


A Solution

- Have the binomial heap store a pointer to the minimum element.
- Can be updated in time O(1) after doing a meld by comparing the minima of the two heaps.

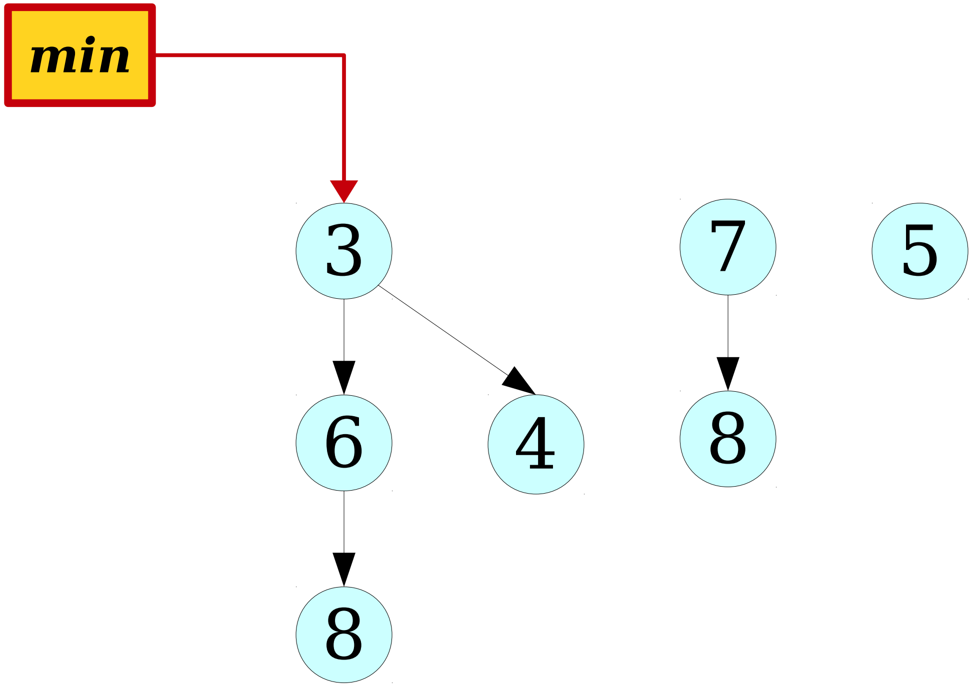
A Solution

- Have the binomial heap store a pointer to the minimum element.
- Can be updated in time O(1) after doing a meld by comparing the minima of the two heaps.



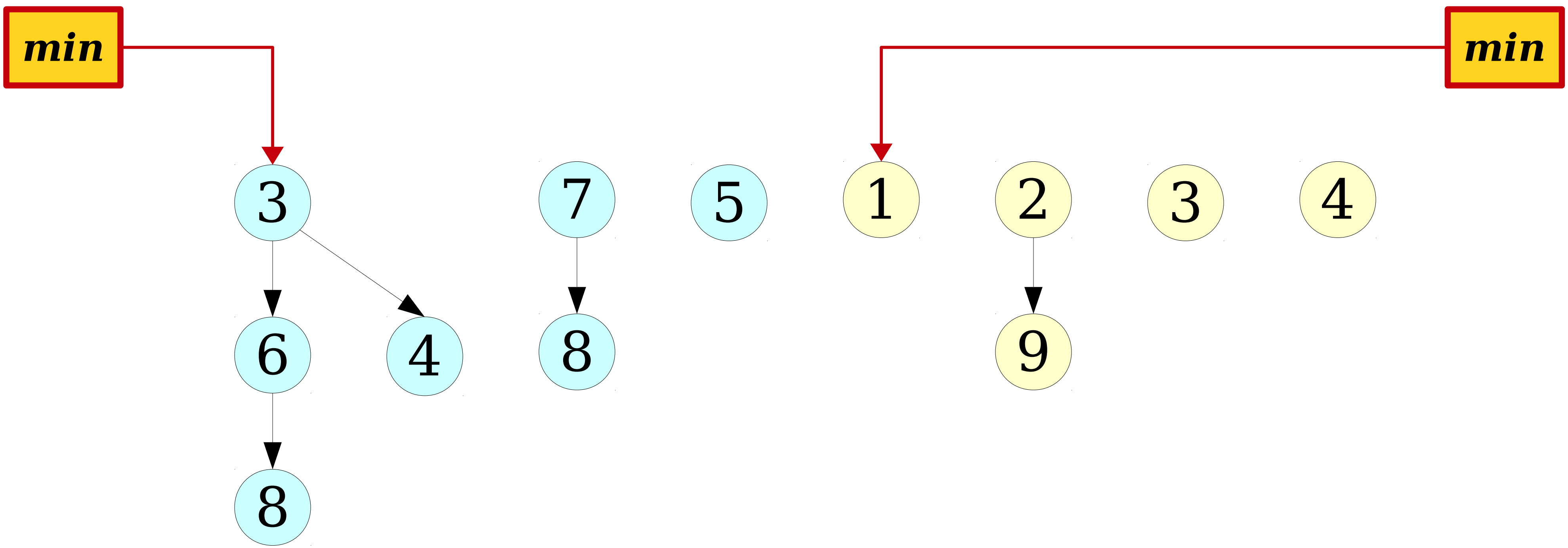
A Solution

- Have the binomial heap store a pointer to the minimum element.
- Can be updated in time O(1) after doing a meld by comparing the minima of the two heaps.



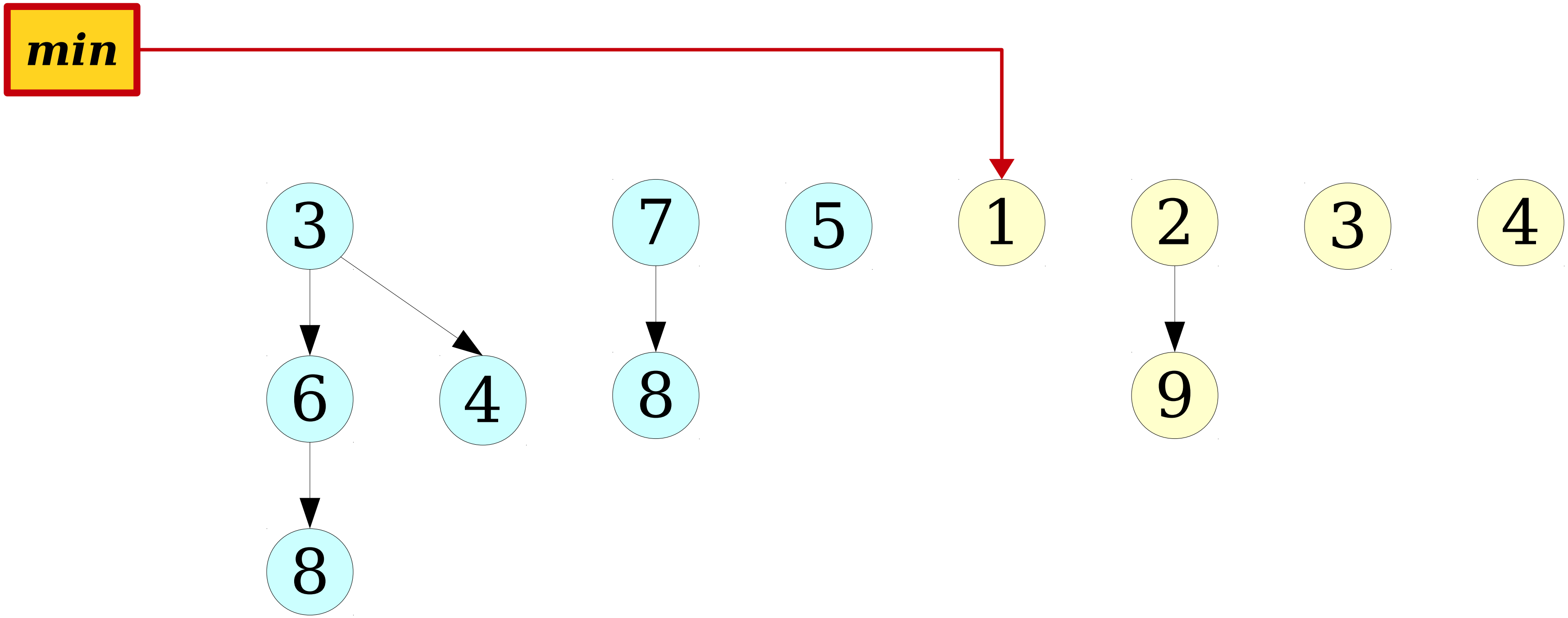
A Solution

- Have the binomial heap store a pointer to the minimum element.
- Can be updated in time O(1) after doing a meld by comparing the minima of the two heaps.



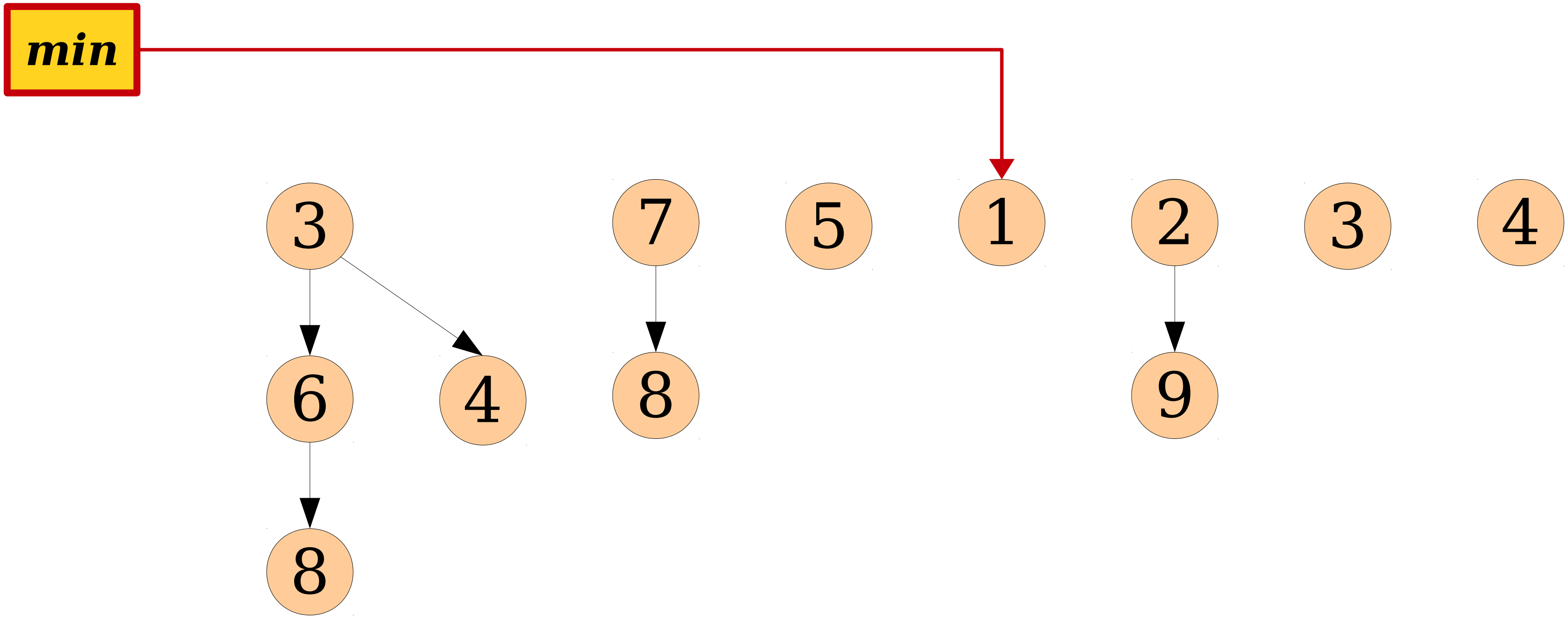
A Solution

- Have the binomial heap store a pointer to the minimum element.
- Can be updated in time O(1) after doing a meld by comparing the minima of the two heaps.



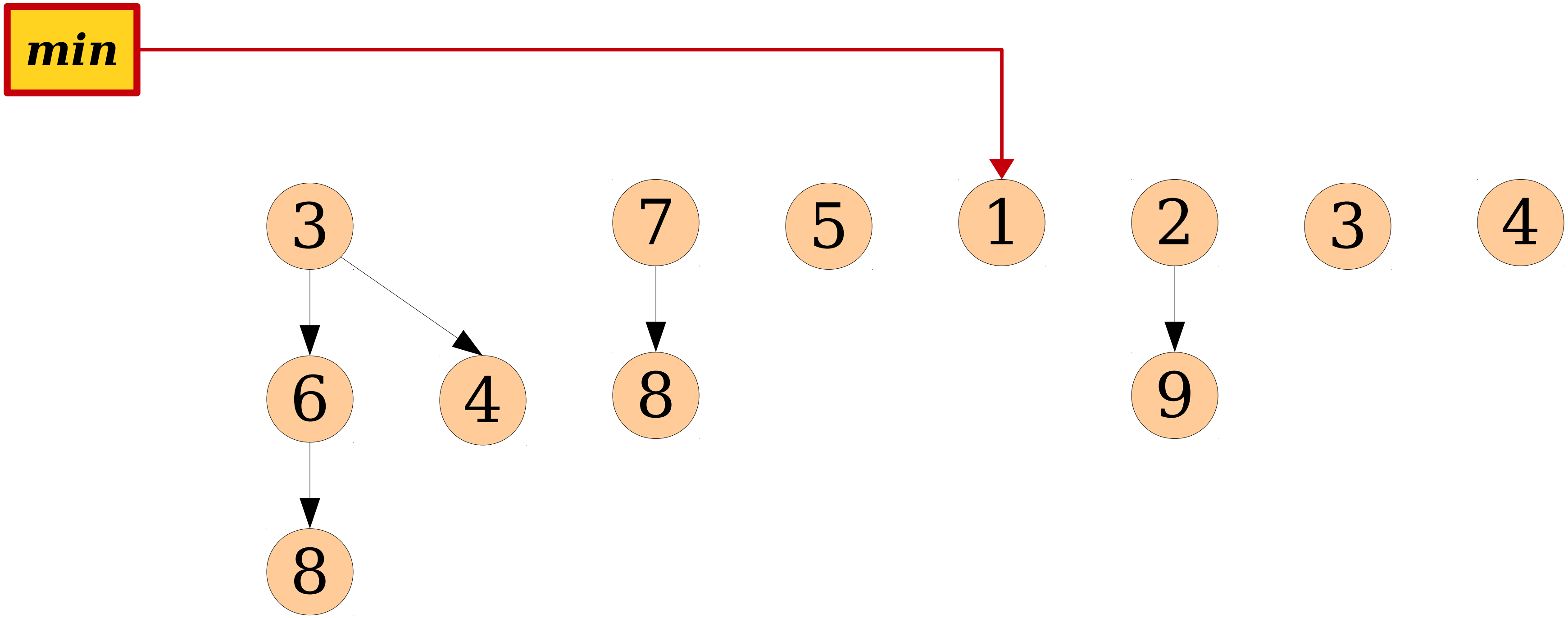
A Solution

- Have the binomial heap store a pointer to the minimum element.
- Can be updated in time O(1) after doing a meld by comparing the minima of the two heaps.



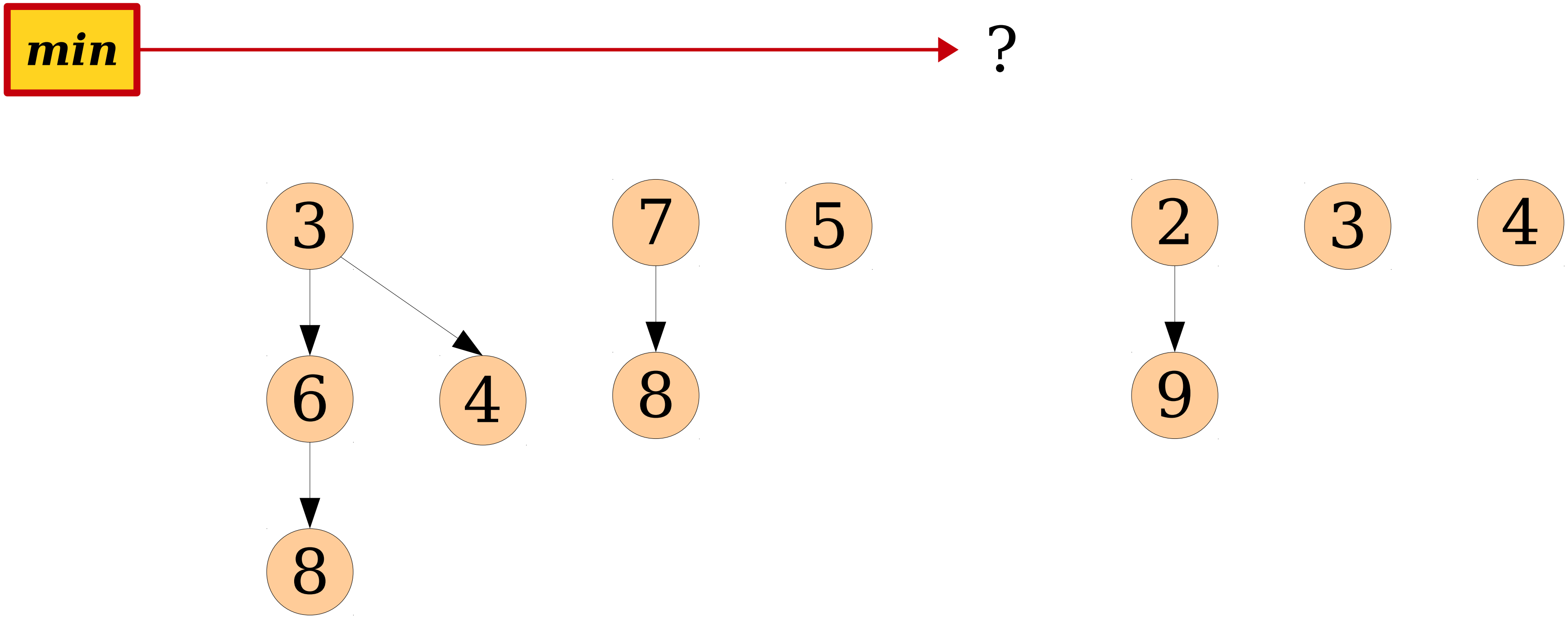
The Catch: Part Two

- Even with a pointer to the minimum, deletions might now run in time $\Theta(n)$.
- ***Rationale:*** Need to update the pointer to the minimum.



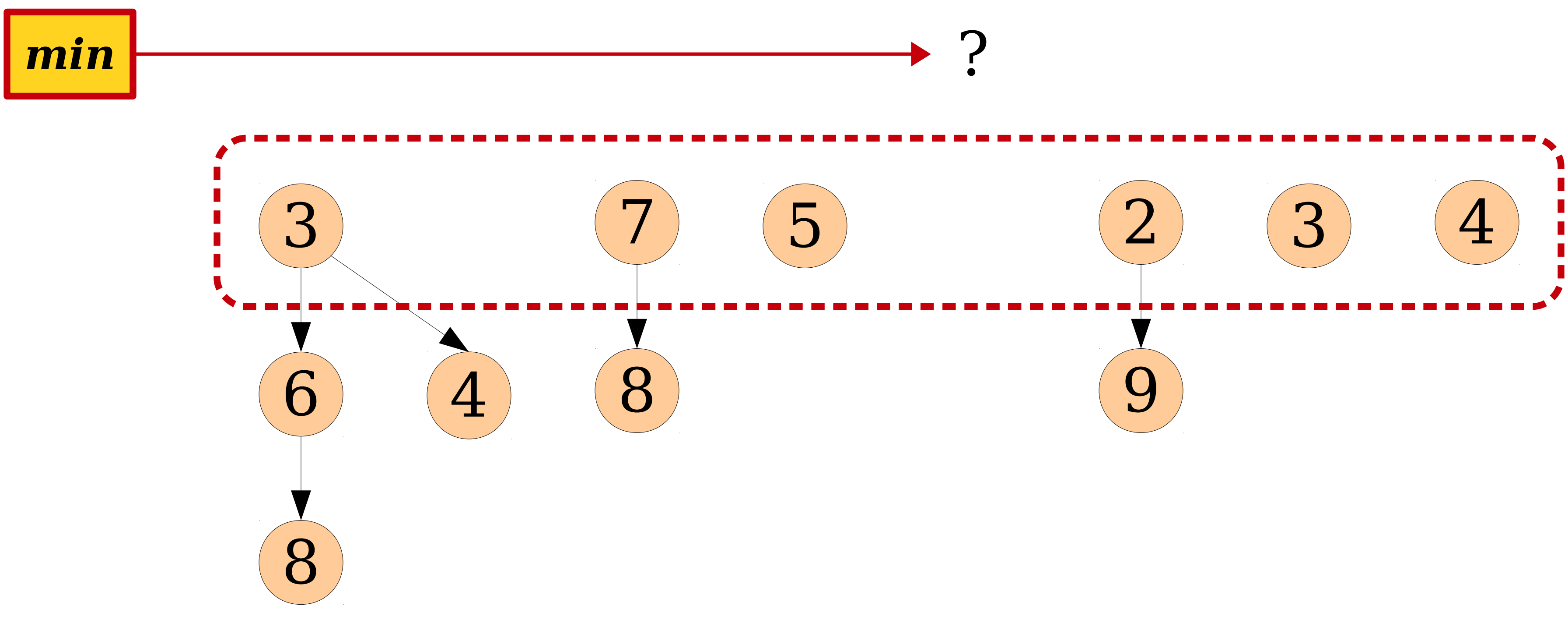
The Catch: Part Two

- Even with a pointer to the minimum, deletions might now run in time $\Theta(n)$.
- ***Rationale:*** Need to update the pointer to the minimum.



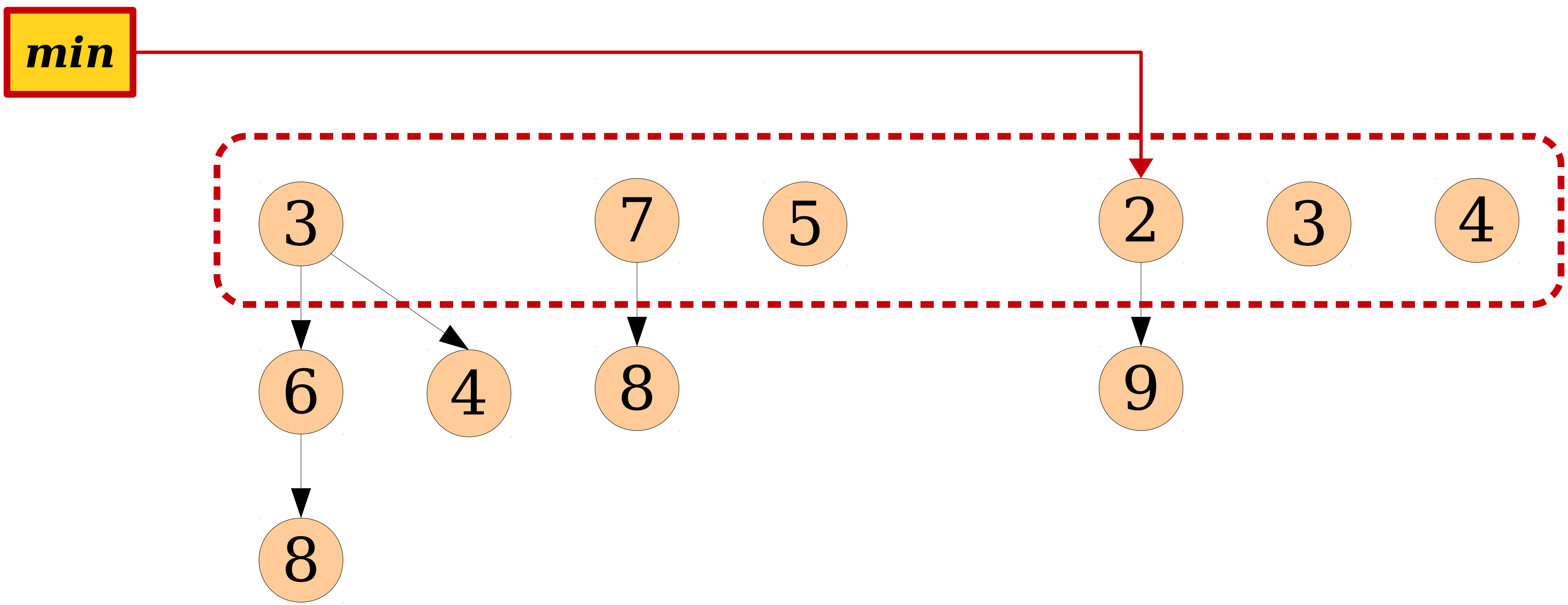
The Catch: Part Two

- Even with a pointer to the minimum, deletions might now run in time $\Theta(n)$.
- ***Rationale:*** Need to update the pointer to the minimum.



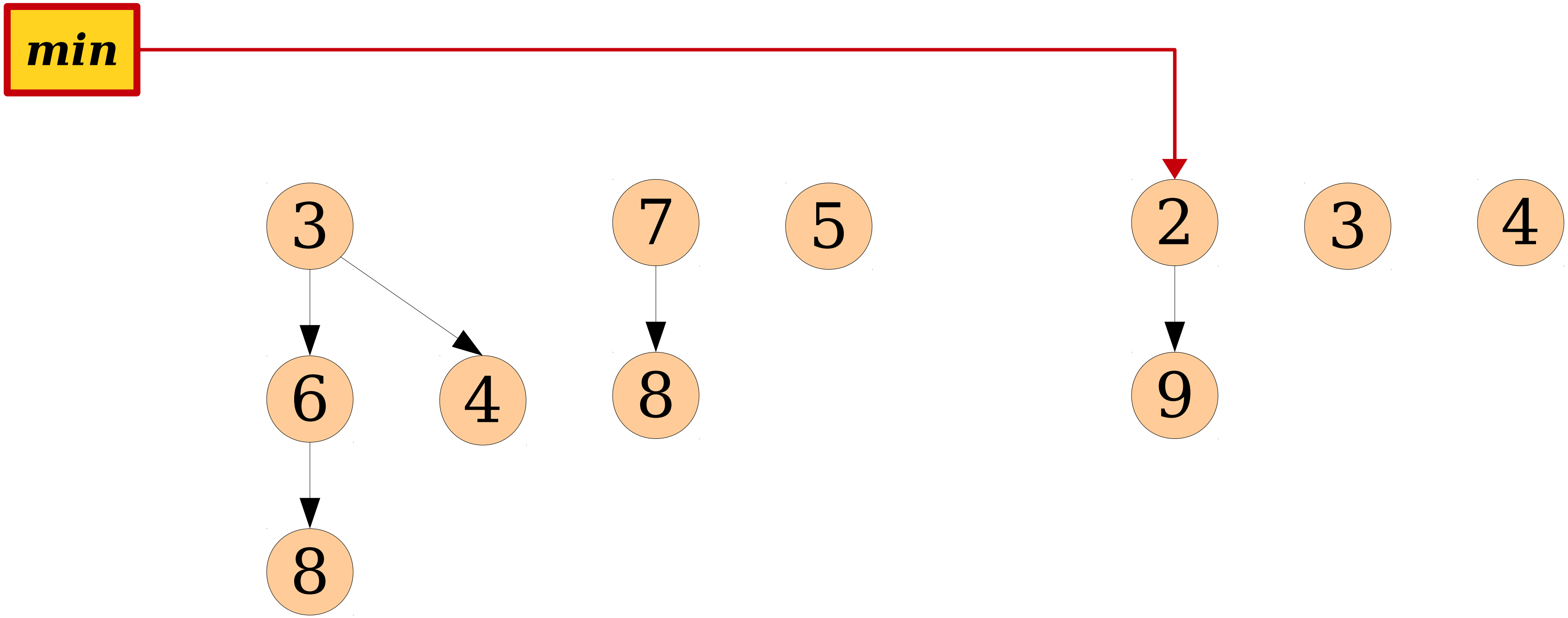
The Catch: Part Two

- Even with a pointer to the minimum, deletions might now run in time $\Theta(n)$.
- ***Rationale:*** Need to update the pointer to the minimum.



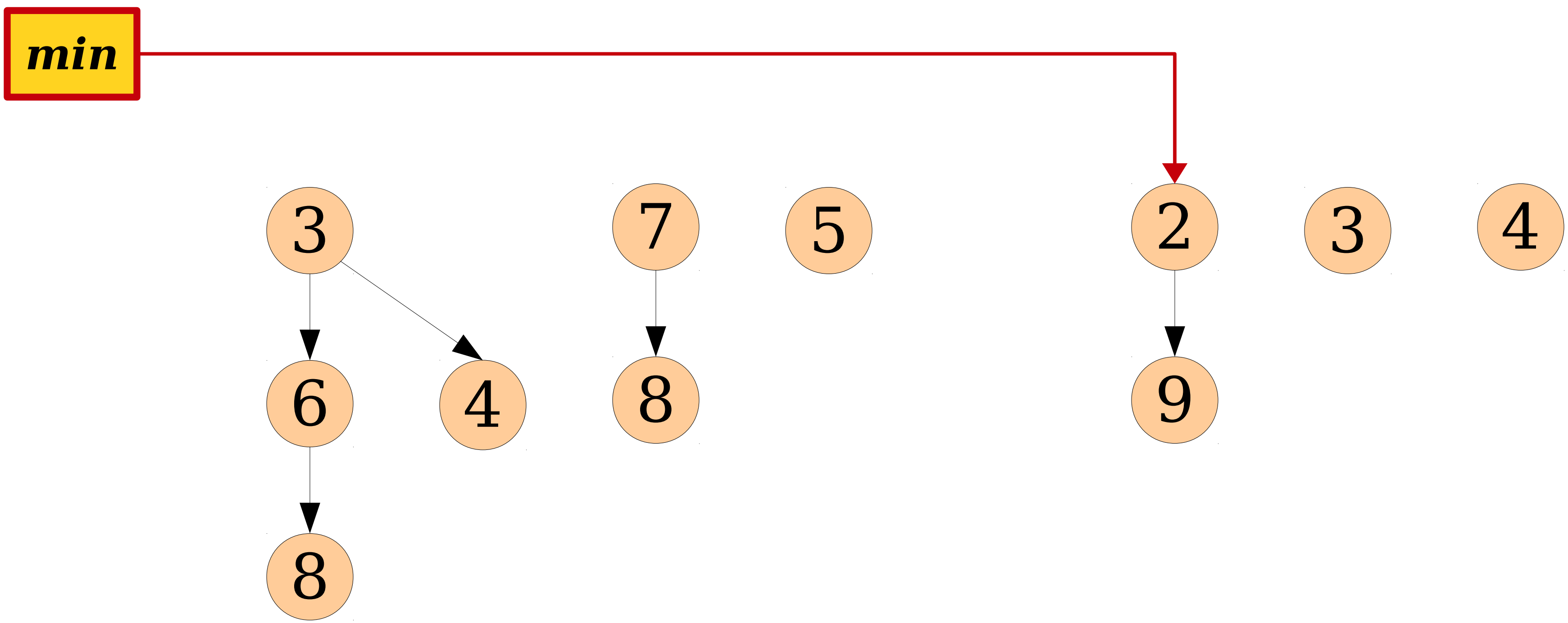
The Catch: Part Two

- Even with a pointer to the minimum, deletions might now run in time $\Theta(n)$.
- ***Rationale:*** Need to update the pointer to the minimum.



Resolving the Issue

- ***Intuition:*** Amortization works well when
  - imbalances accumulate slowly, and
  - imbalances get cleaned up quickly.
- We've got the first. How do we get the second?

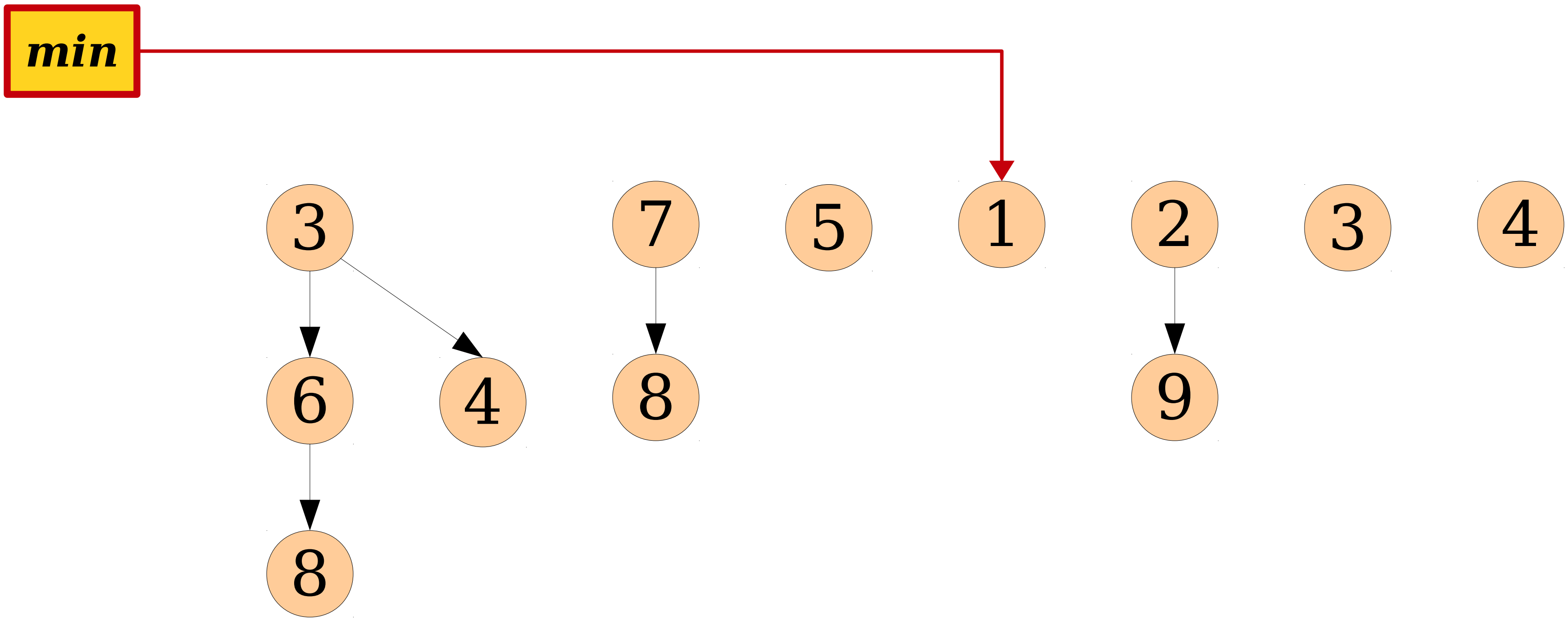


Resolving the Issue

- ***Idea:*** When doing an ***extract-min***, coalesce all of the trees so that there's at most one tree of each order.
- Intuitively:
  - The number of trees in a heap grows slowly (only during an ***enqueue*** or ***meld***).
  - The number of trees in a heap drops rapidly after coalescing (down to O(log n)).
  - Can backcharge the work done during an ***extract-min*** to ***enqueue*** or ***meld***.

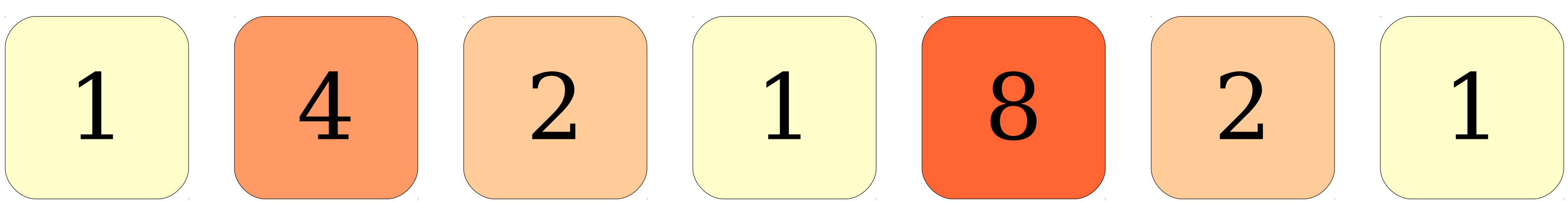
Coalescing Trees

- Our eager melding algorithm assumes that
  - there is either zero or one tree of each order, and that
  - the trees are stored in ascending order.
- ***Challenge:*** When coalescing trees in this case, neither of these properties necessarily hold.



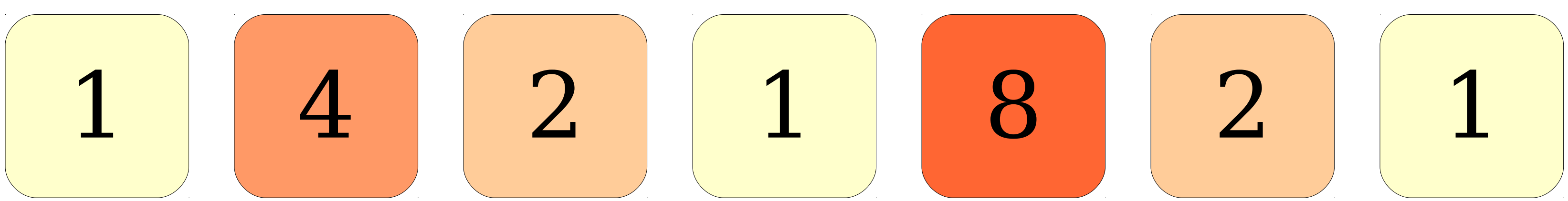
Combining Packets

- You're given a collection of packets of various orders in essentially random order.
- ***Goal:*** Combine the packets together such that there's at most one packet of each order.
- ***Question:*** How can we do this efficiently?



Combining Packets

- ***Observation:*** This would be a lot easier if the packets were in sorted order.

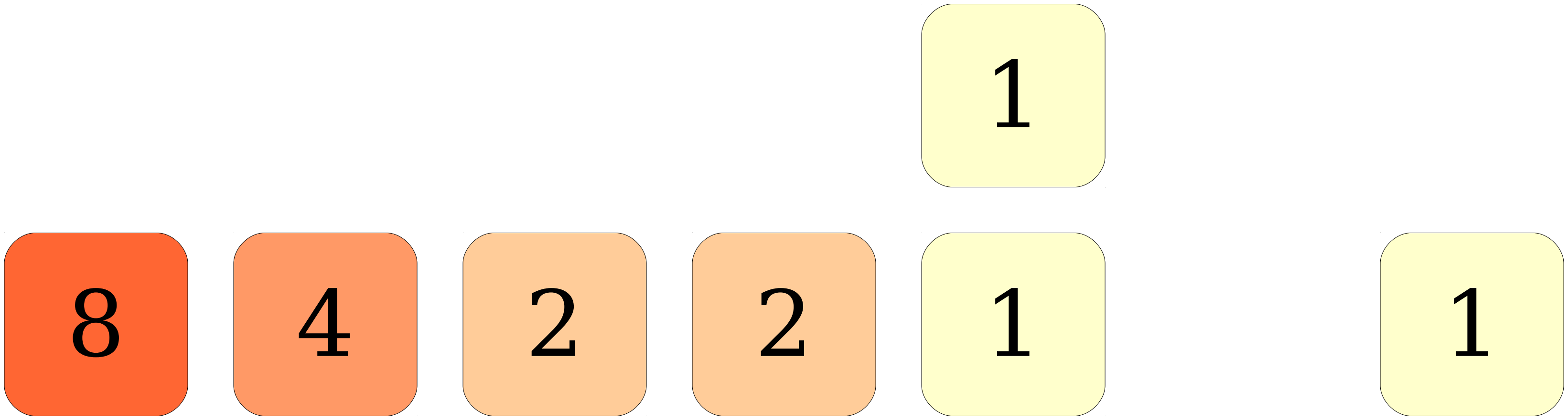


Combining Packets

- ***Observation:*** This would be a lot easier if the packets were in sorted order.

Combining Packets

- ***Observation:*** This would be a lot easier if the packets were in sorted order.

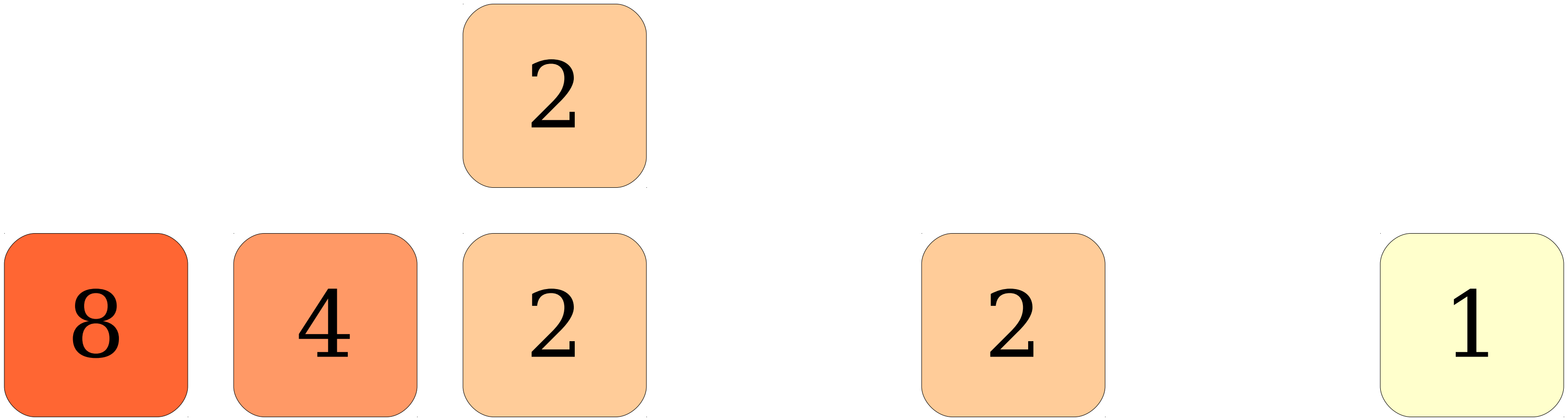


Combining Packets

- ***Observation:*** This would be a lot easier if the packets were in sorted order.

Combining Packets

- ***Observation:*** This would be a lot easier if the packets were in sorted order.

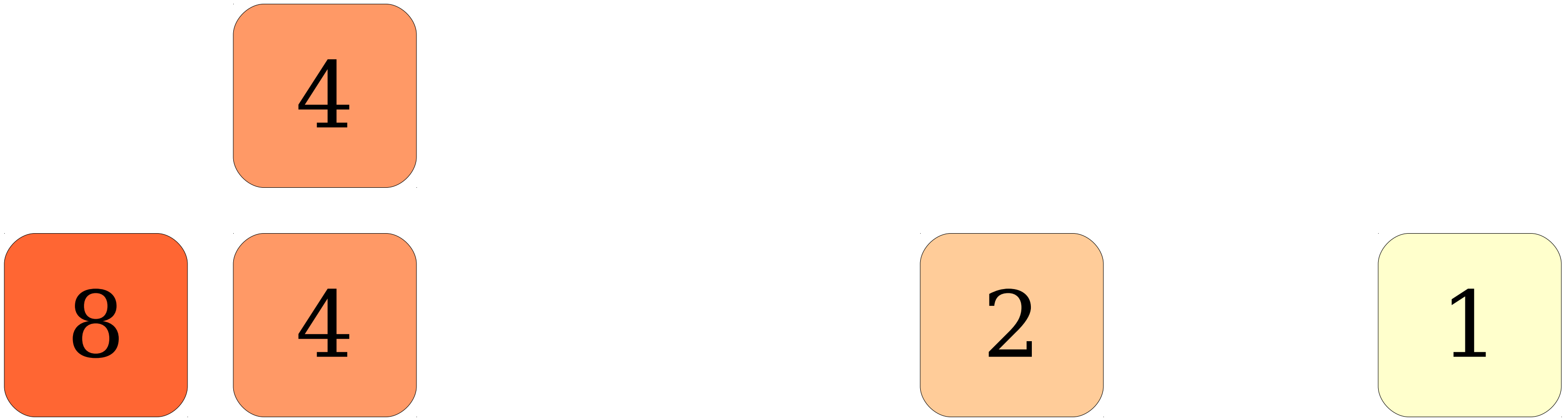


Combining Packets

- ***Observation:*** This would be a lot easier if the packets were in sorted order.

Combining Packets

- ***Observation:*** This would be a lot easier if the packets were in sorted order.

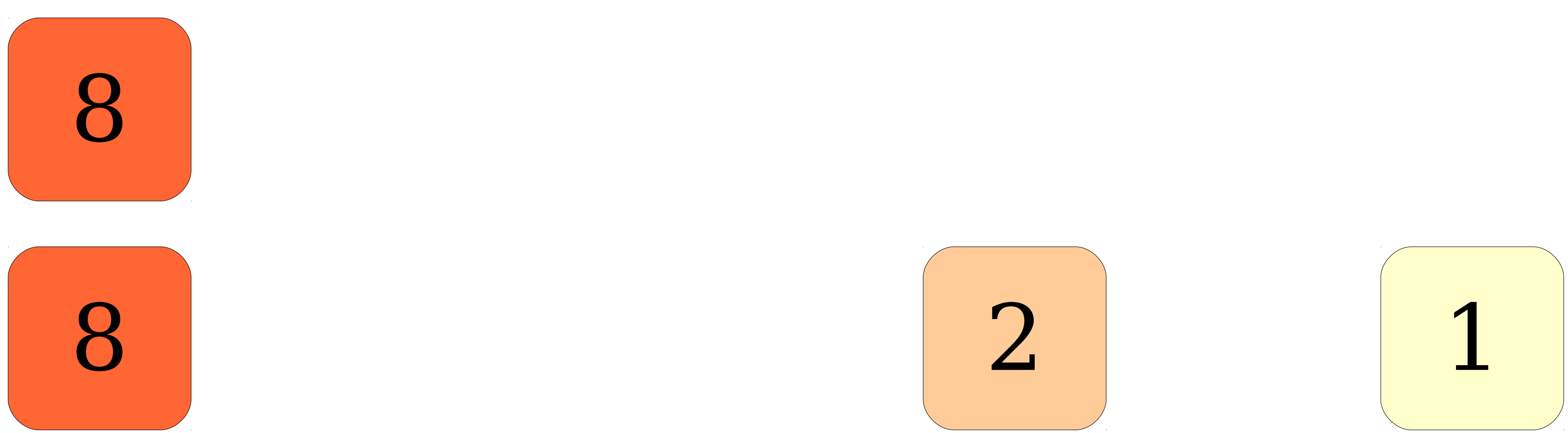


Combining Packets

- ***Observation:*** This would be a lot easier if the packets were in sorted order.

Combining Packets

- ***Observation:*** This would be a lot easier if the packets were in sorted order.



Combining Packets

- ***Observation:*** This would be a lot easier if the packets were in sorted order.

Combining Packets

- ***Observation:*** This would be a lot easier if the packets were in sorted order.

Combining Packets

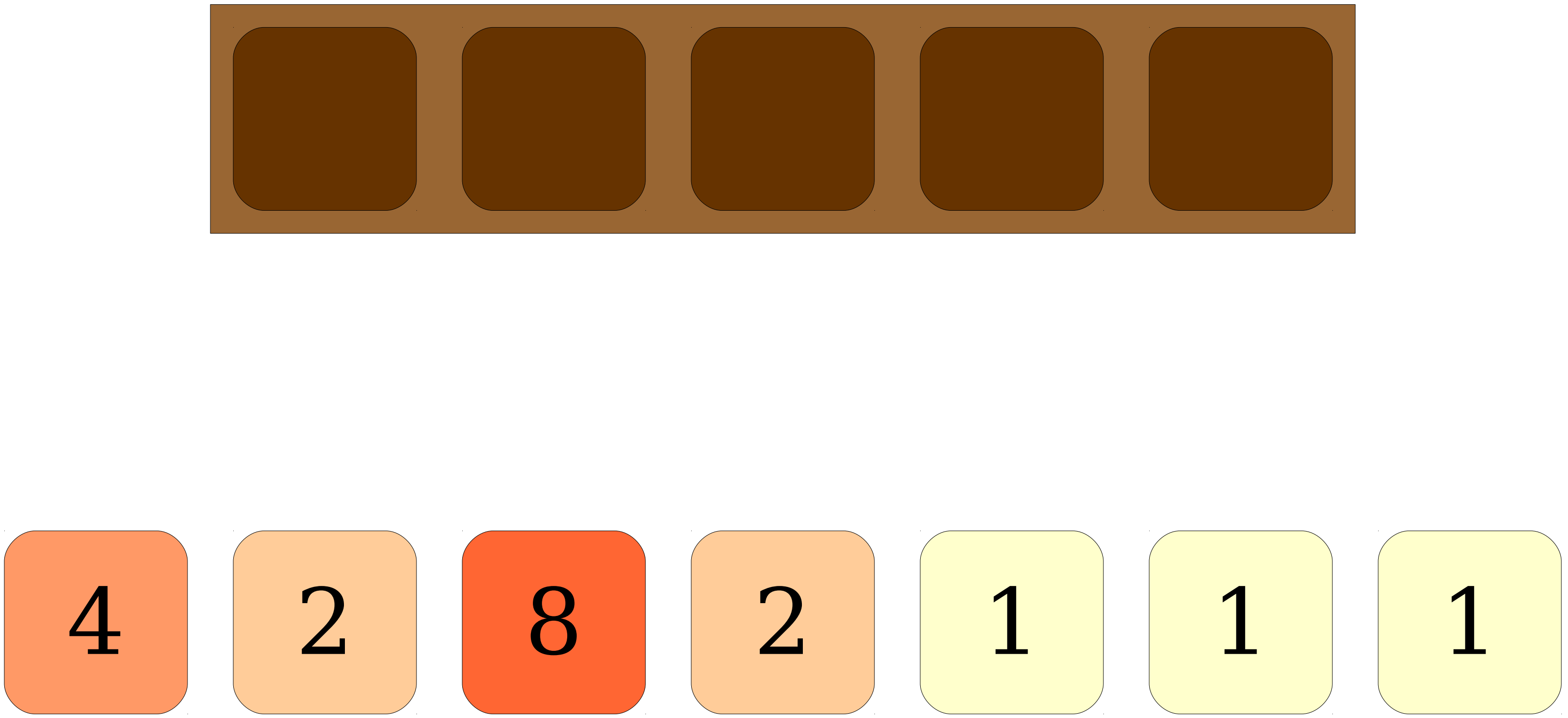
- ***Observation:*** This would be a lot easier if the packets were in sorted order.
- We know each packet has an integer size. What's a good sorting algorithm for integers?
- ***Answer:*** Counting sort!

Counting Sort Coalescing

- ***Idea:*** Use a modified version of counting sort to combine packets together.

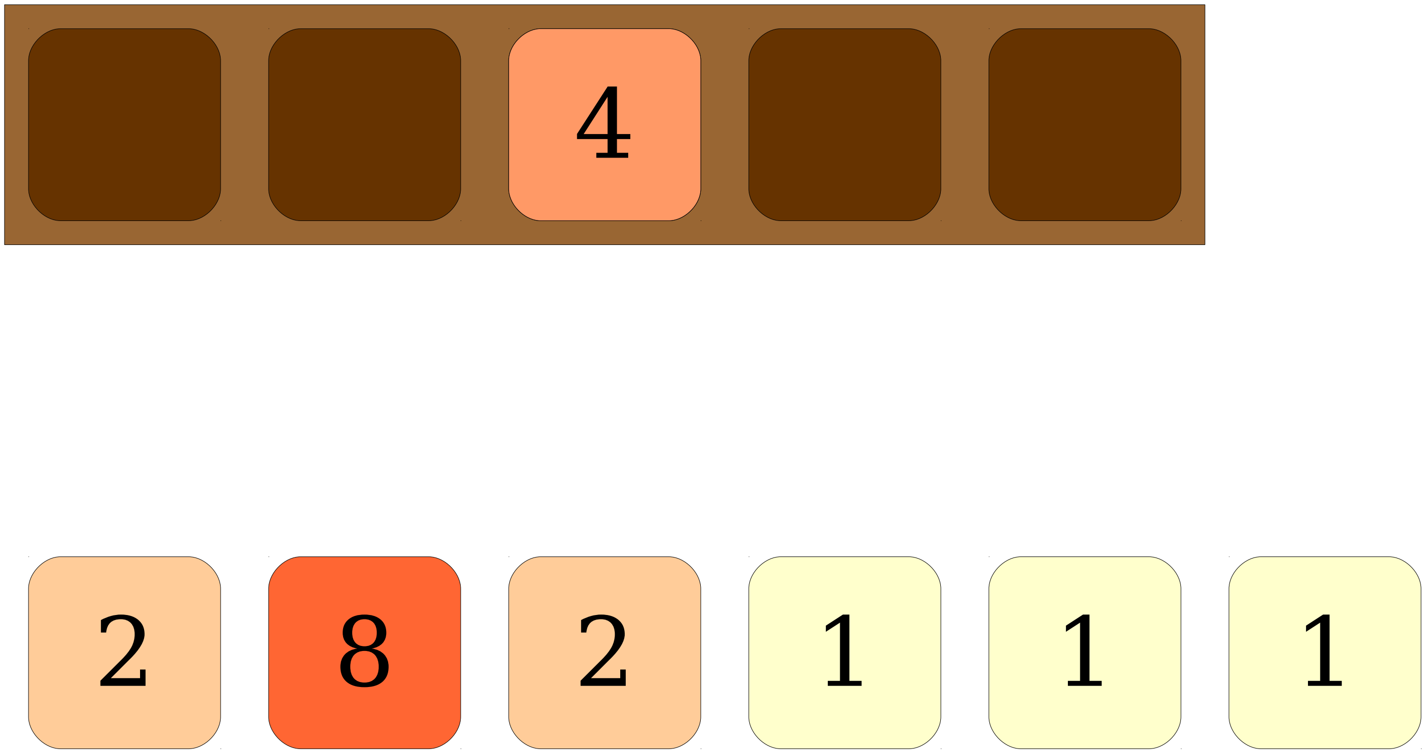
Counting Sort Coalescing

- ***Idea:*** Use a modified version of counting sort to combine packets together.



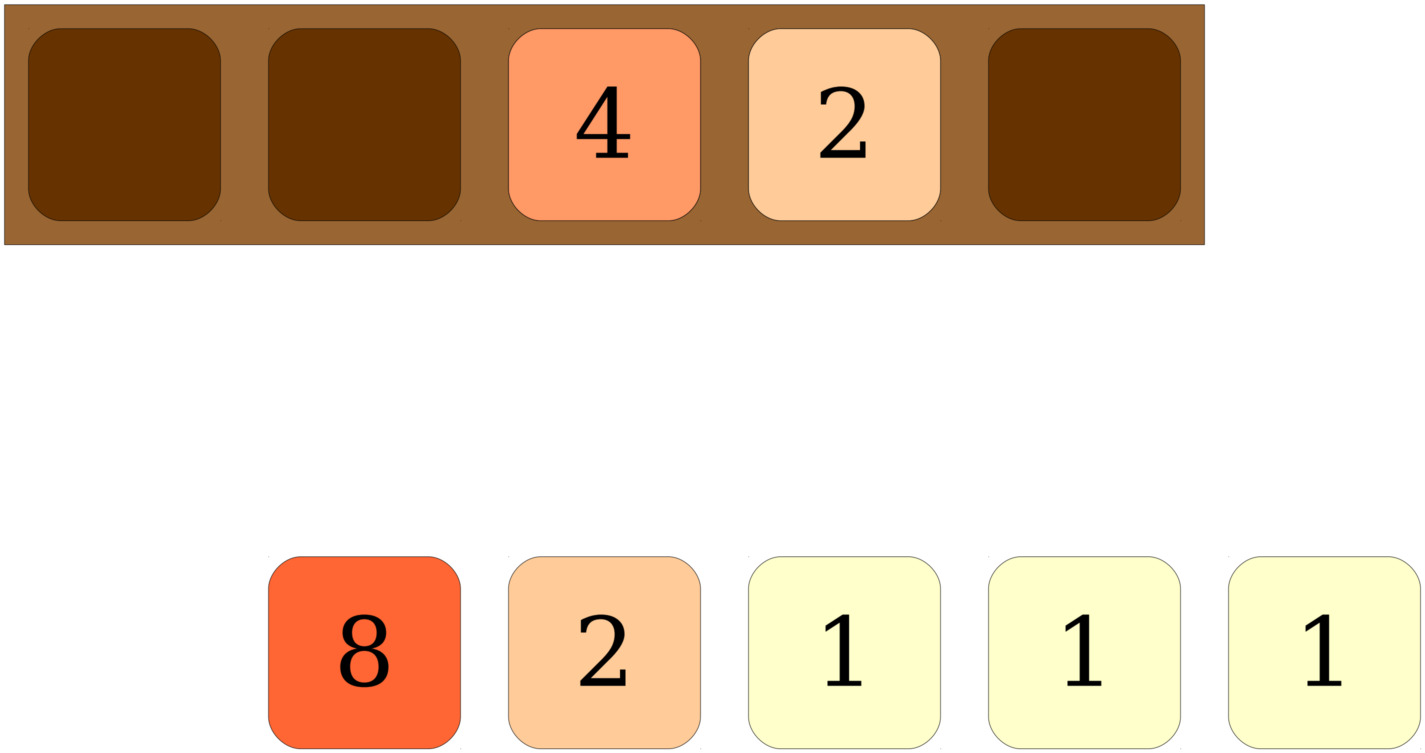
Counting Sort Coalescing

- ***Idea:*** Use a modified version of counting sort to combine packets together.



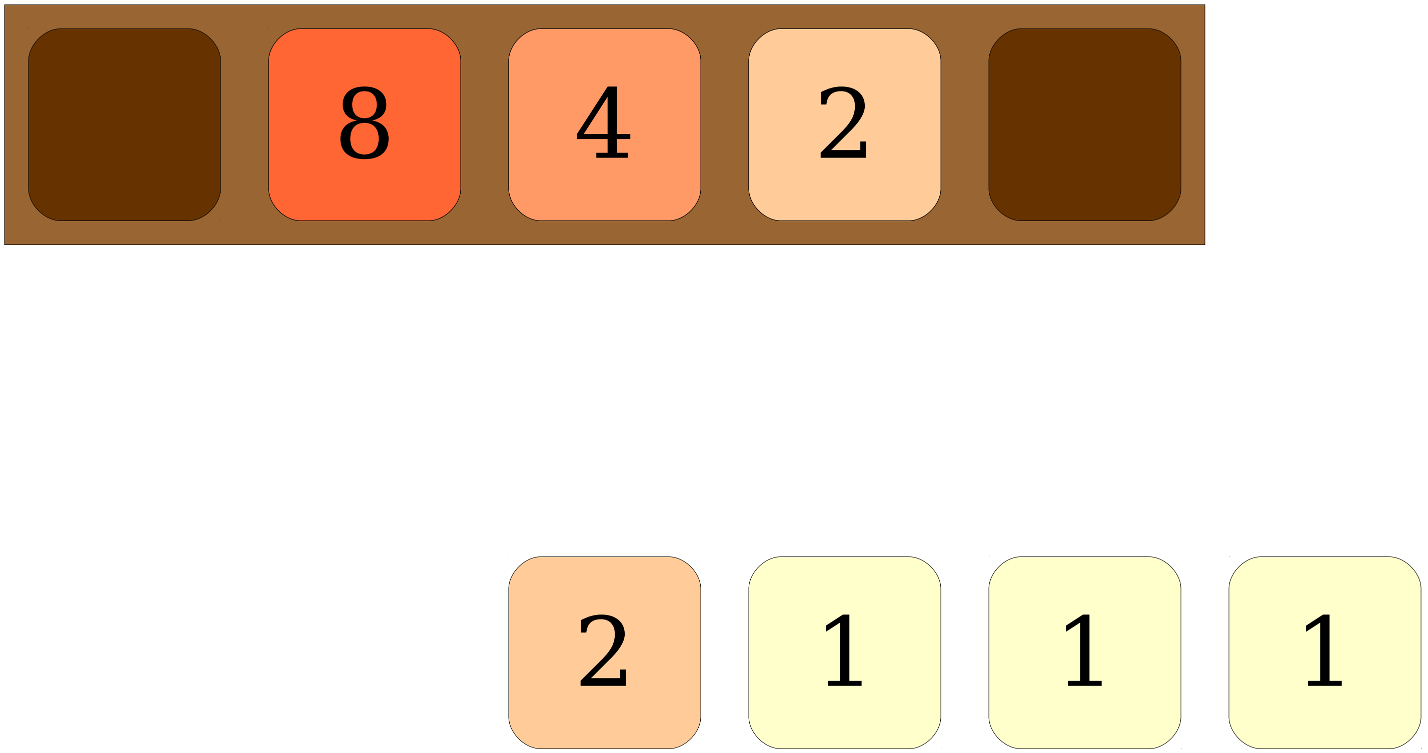
Counting Sort Coalescing

- ***Idea:*** Use a modified version of counting sort to combine packets together.



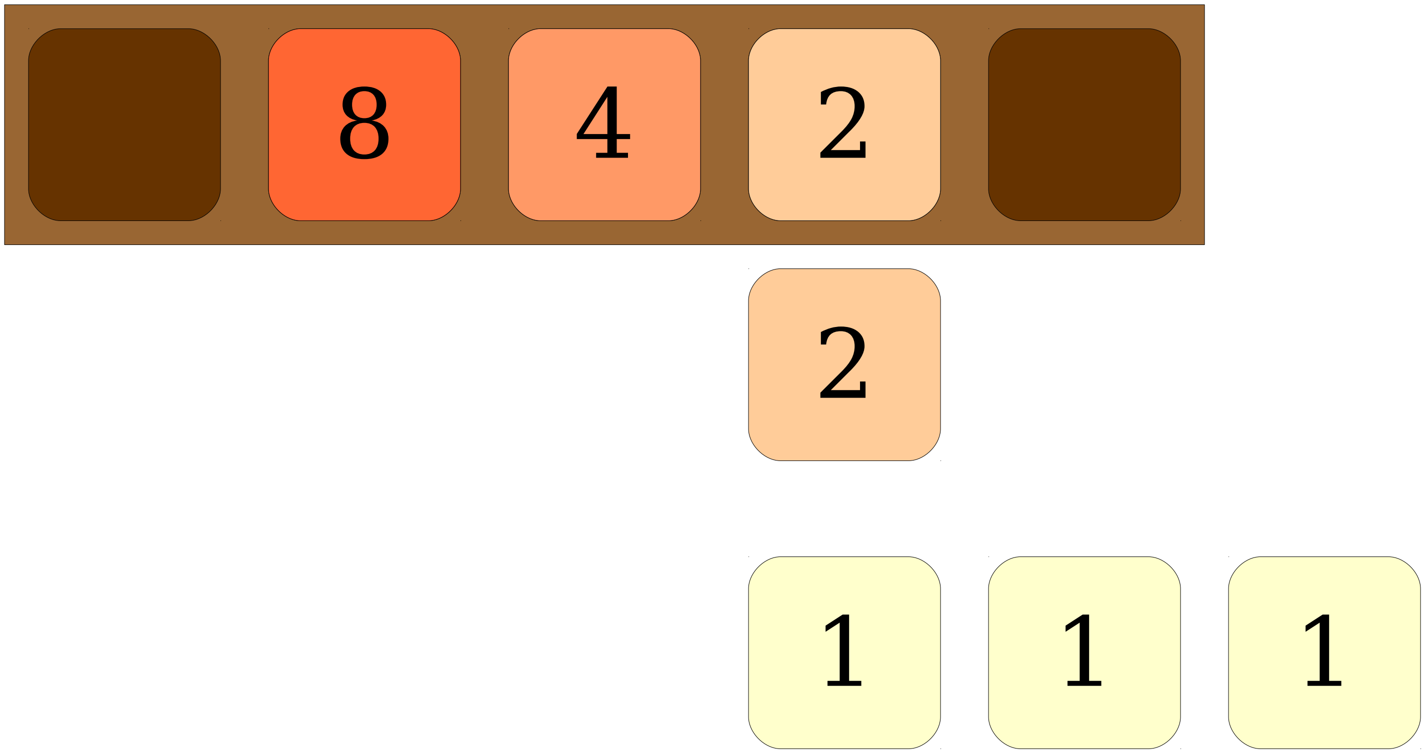
Counting Sort Coalescing

- ***Idea:*** Use a modified version of counting sort to combine packets together.



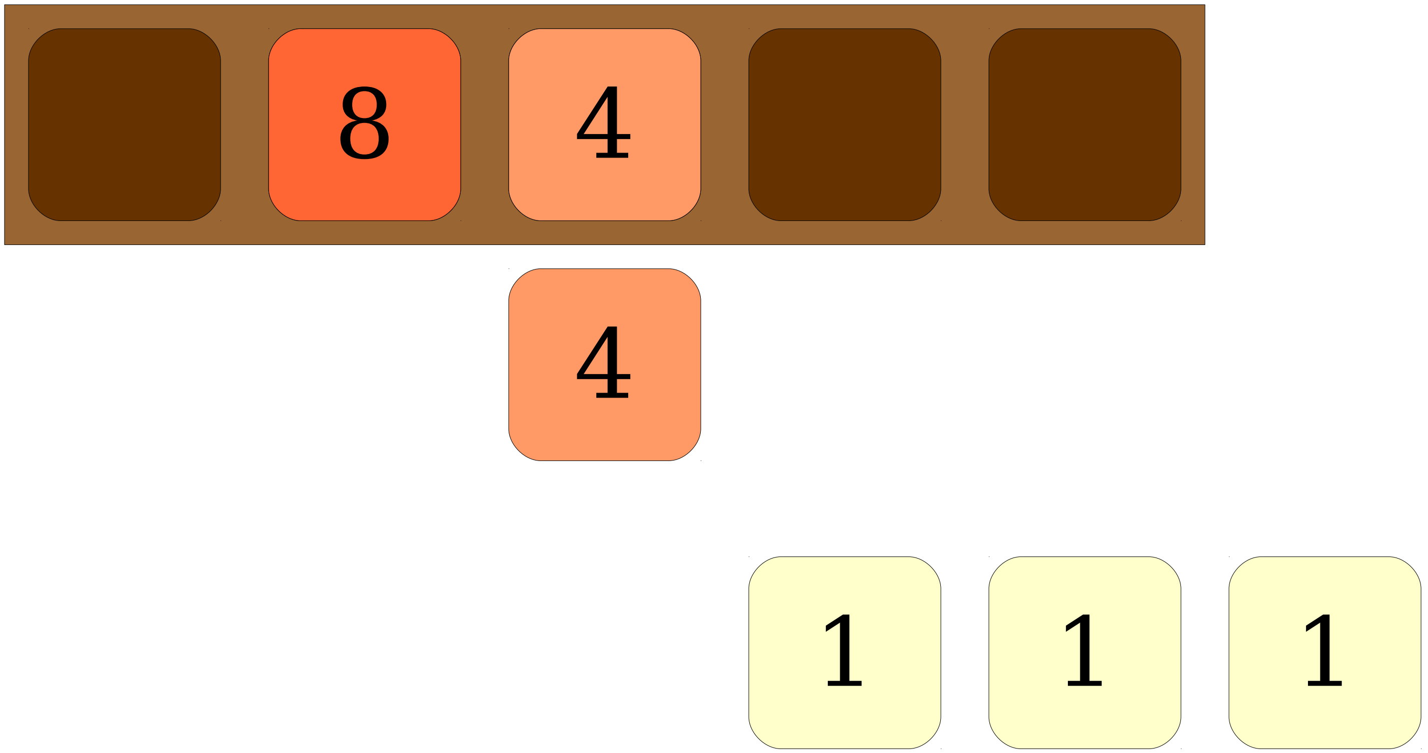
Counting Sort Coalescing

- ***Idea:*** Use a modified version of counting sort to combine packets together.



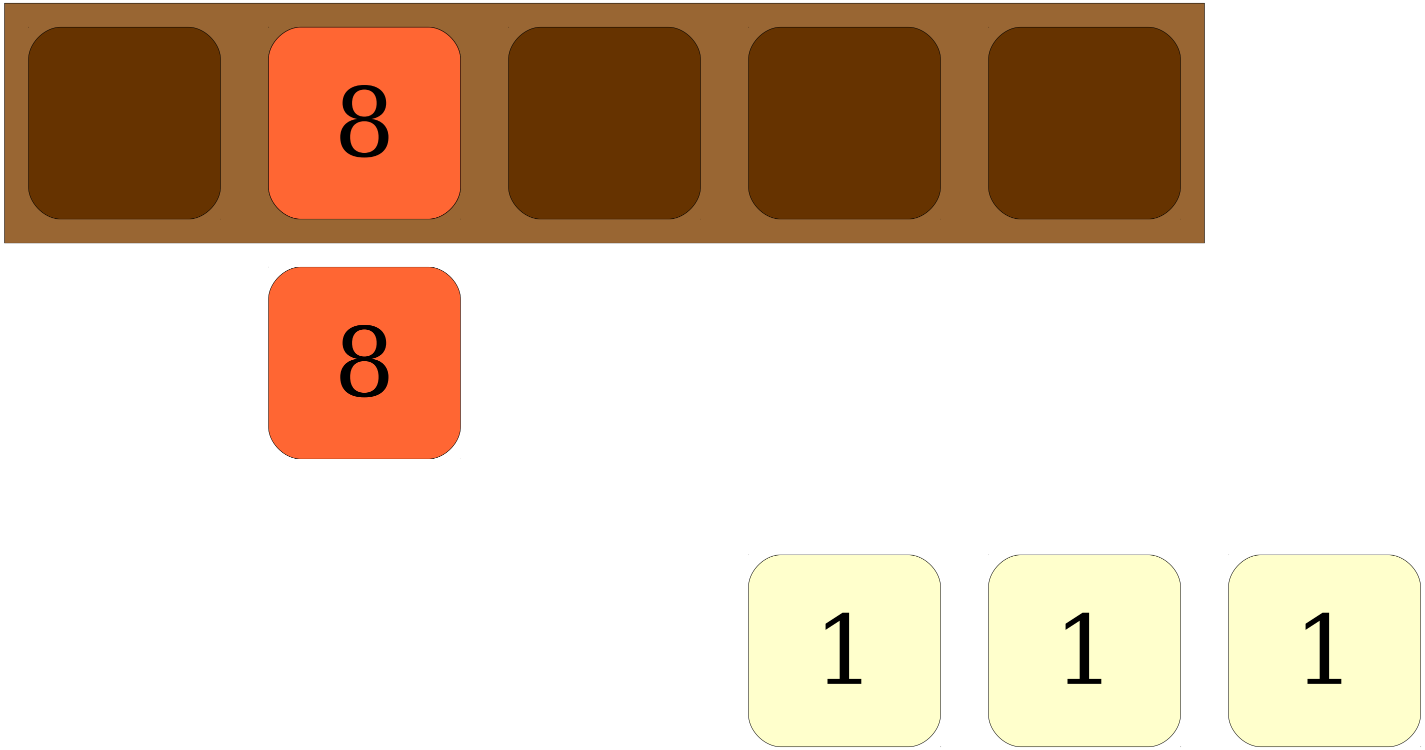
Counting Sort Coalescing

- ***Idea:*** Use a modified version of counting sort to combine packets together.



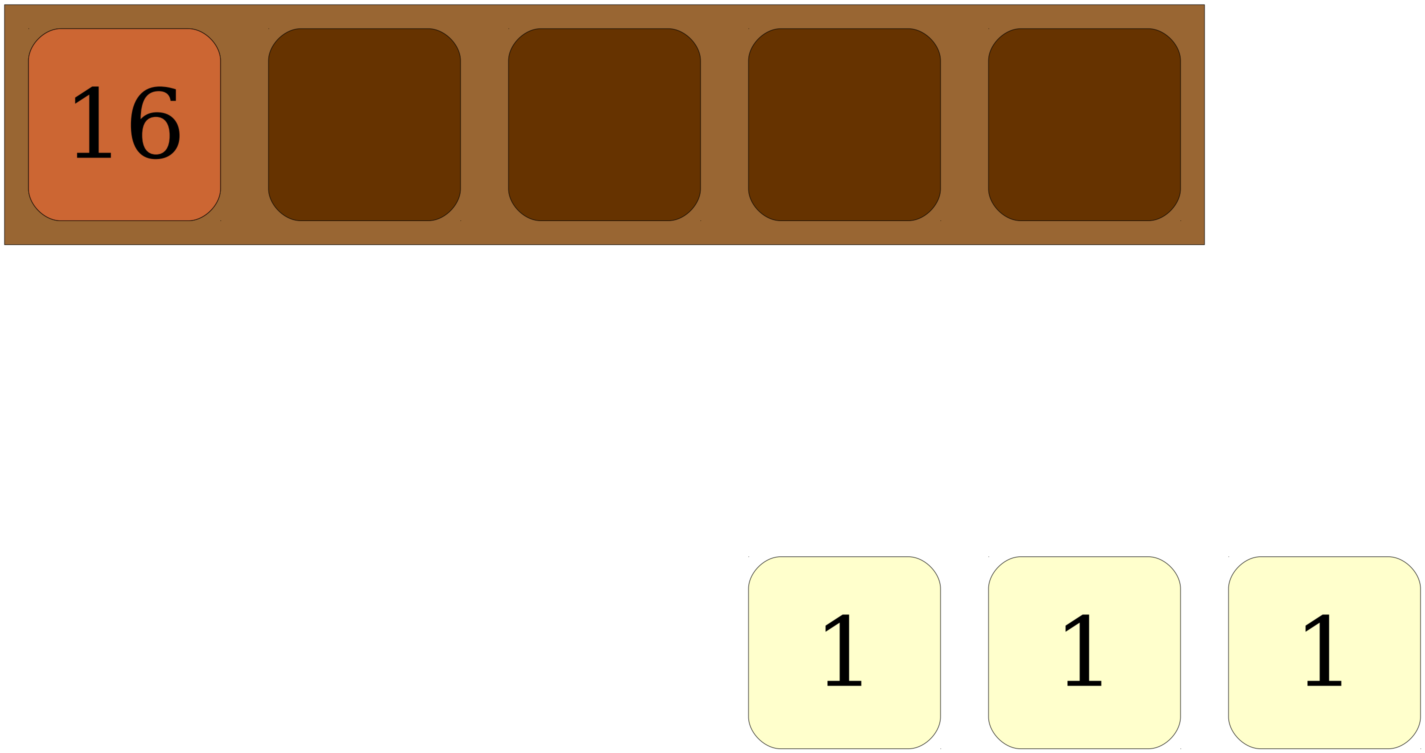
Counting Sort Coalescing

- ***Idea:*** Use a modified version of counting sort to combine packets together.



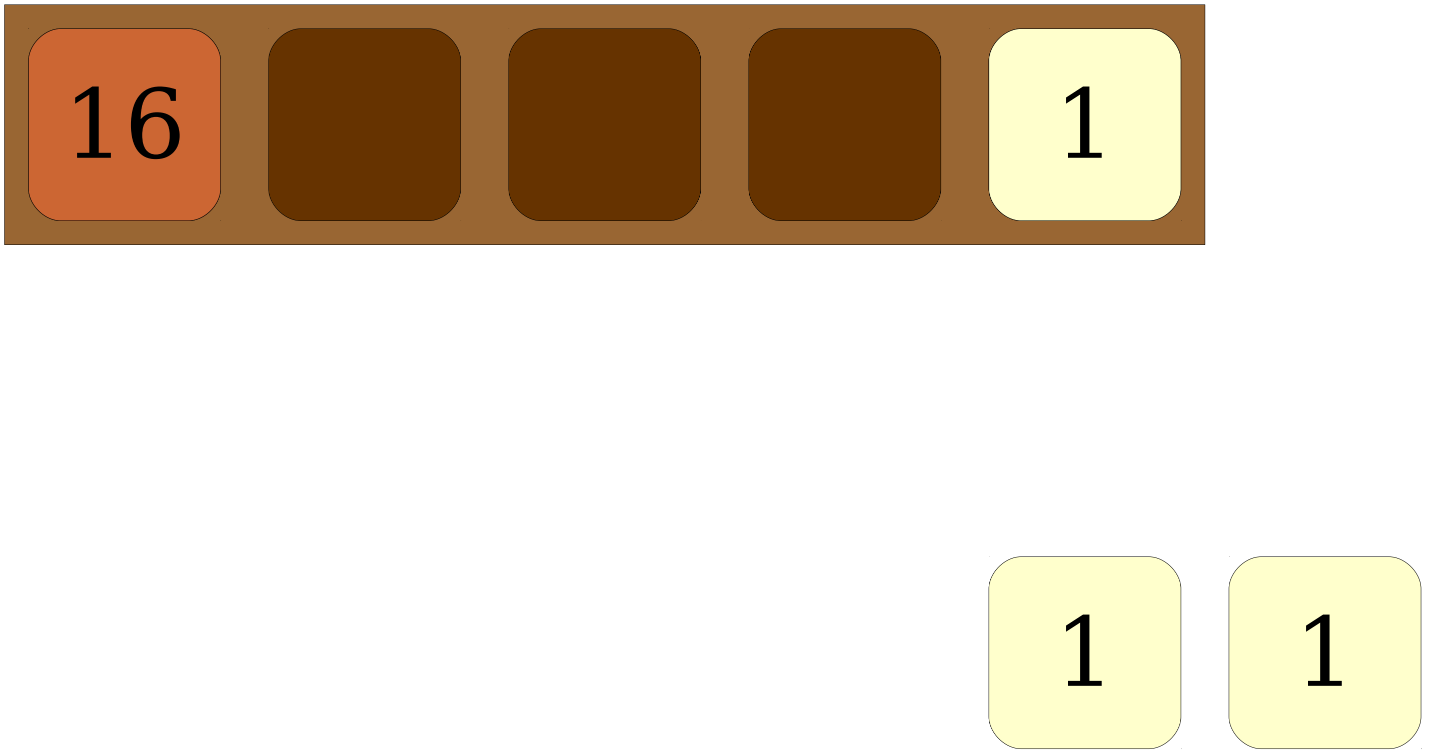
Counting Sort Coalescing

- ***Idea:*** Use a modified version of counting sort to combine packets together.



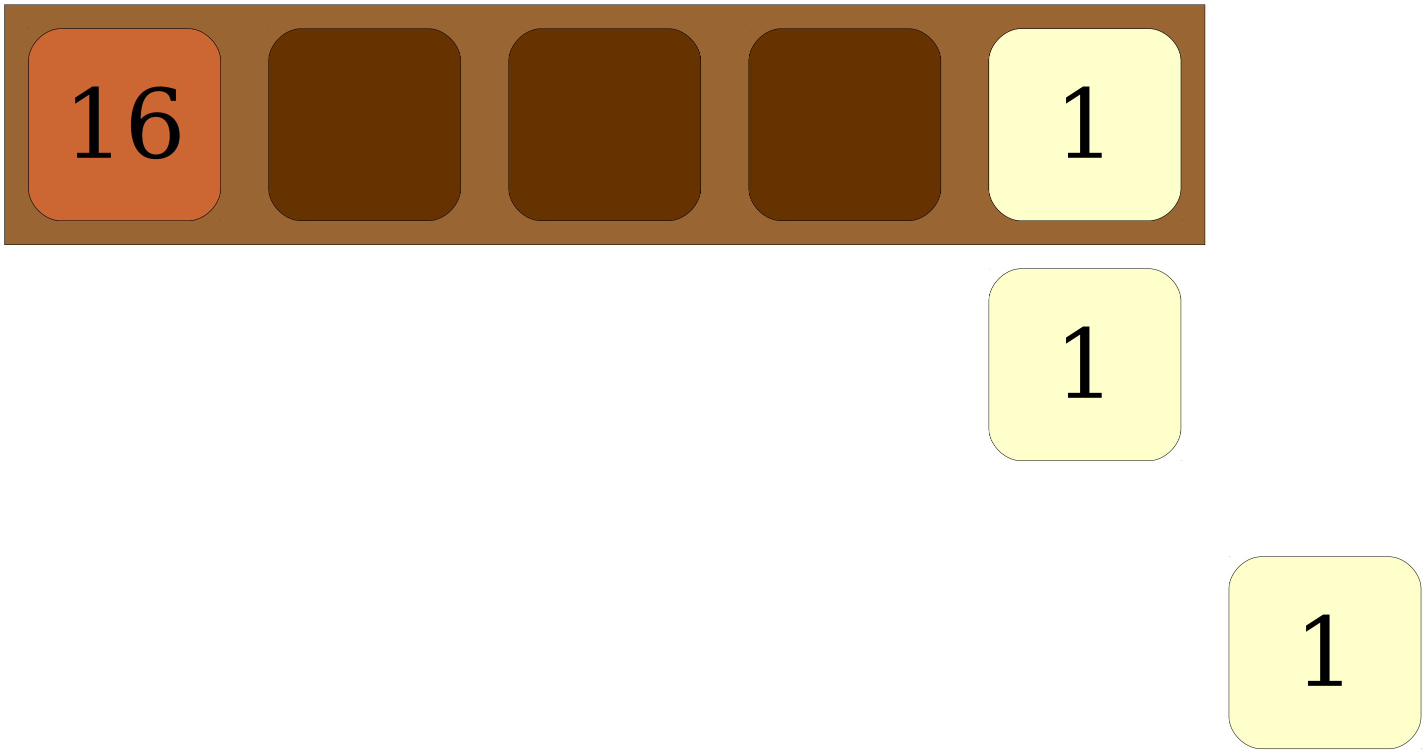
Counting Sort Coalescing

- ***Idea:*** Use a modified version of counting sort to combine packets together.



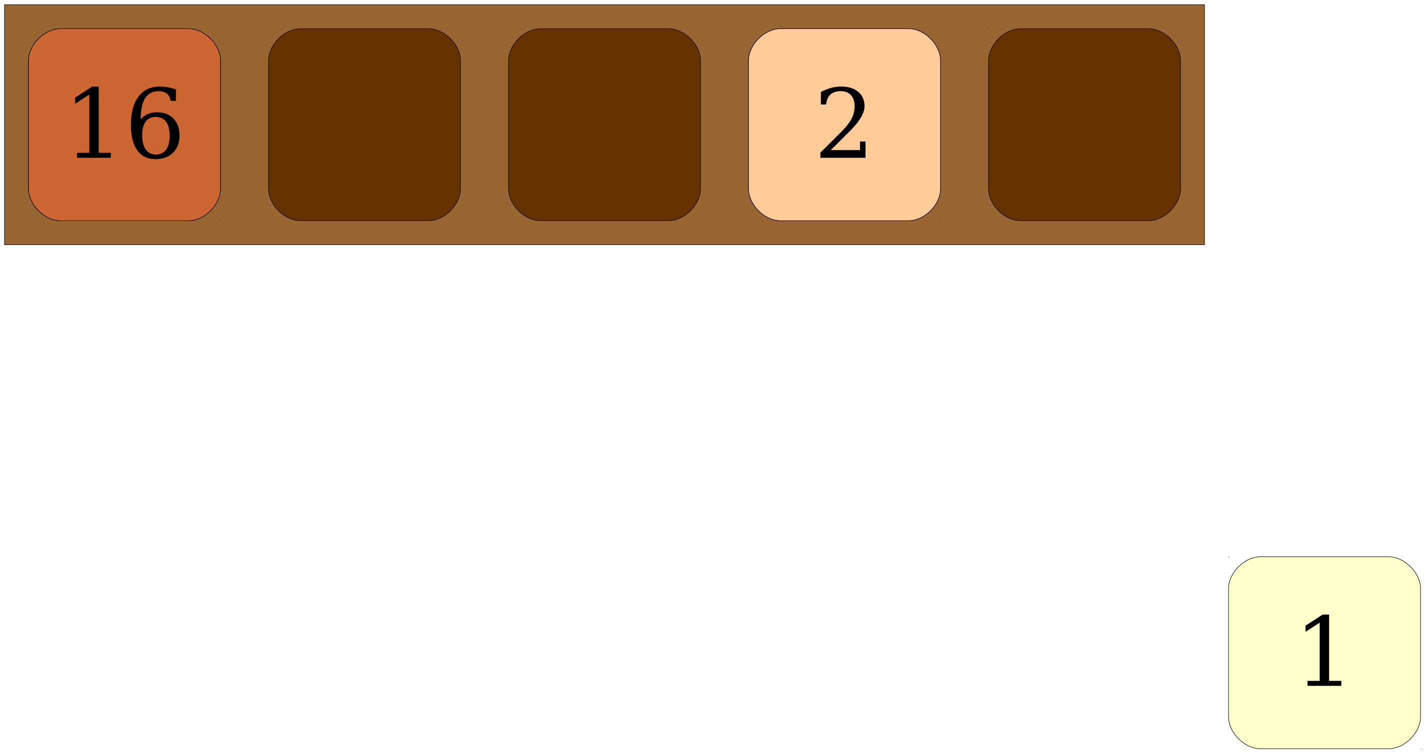
Counting Sort Coalescing

- ***Idea:*** Use a modified version of counting sort to combine packets together.



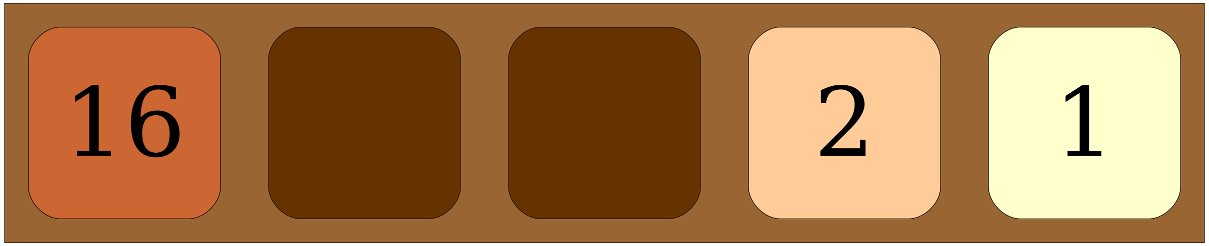
Counting Sort Coalescing

- ***Idea:*** Use a modified version of counting sort to combine packets together.



Counting Sort Coalescing

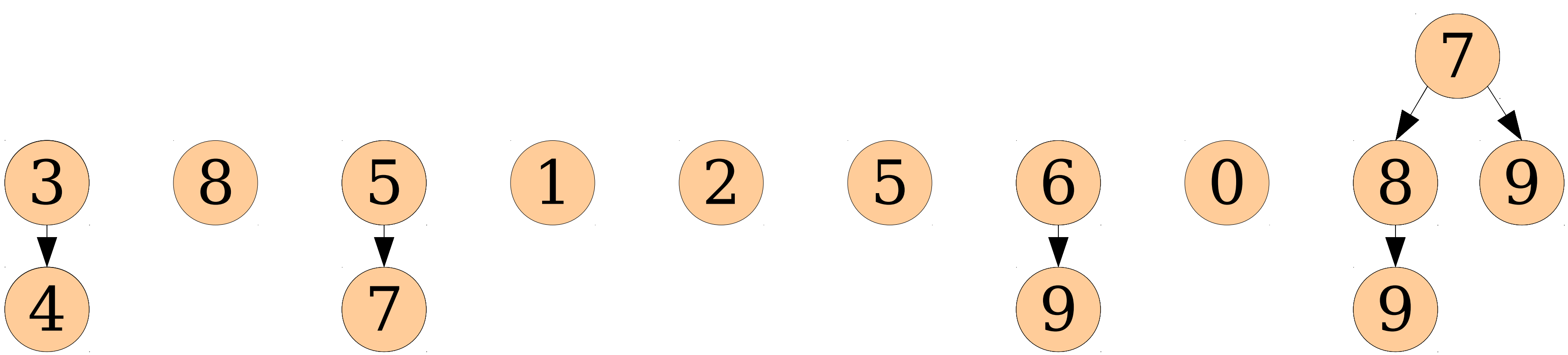
- ***Idea:*** Use a modified version of counting sort to combine packets together.



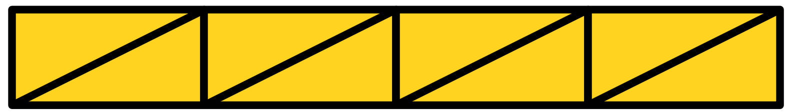
Counting Sort Coalescing

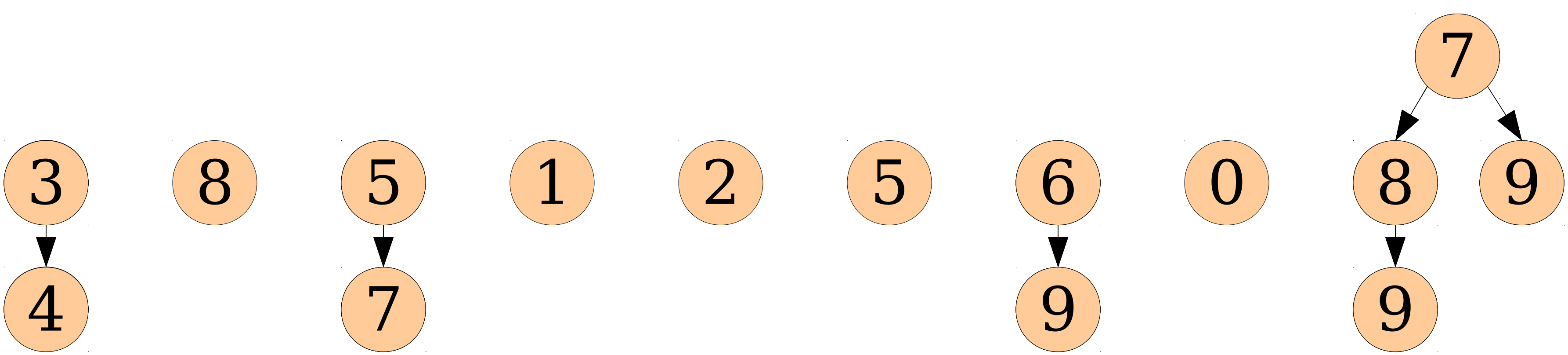
- ***Idea:*** Use a modified version of counting sort to combine packets together.

Counting Sort Coalescing

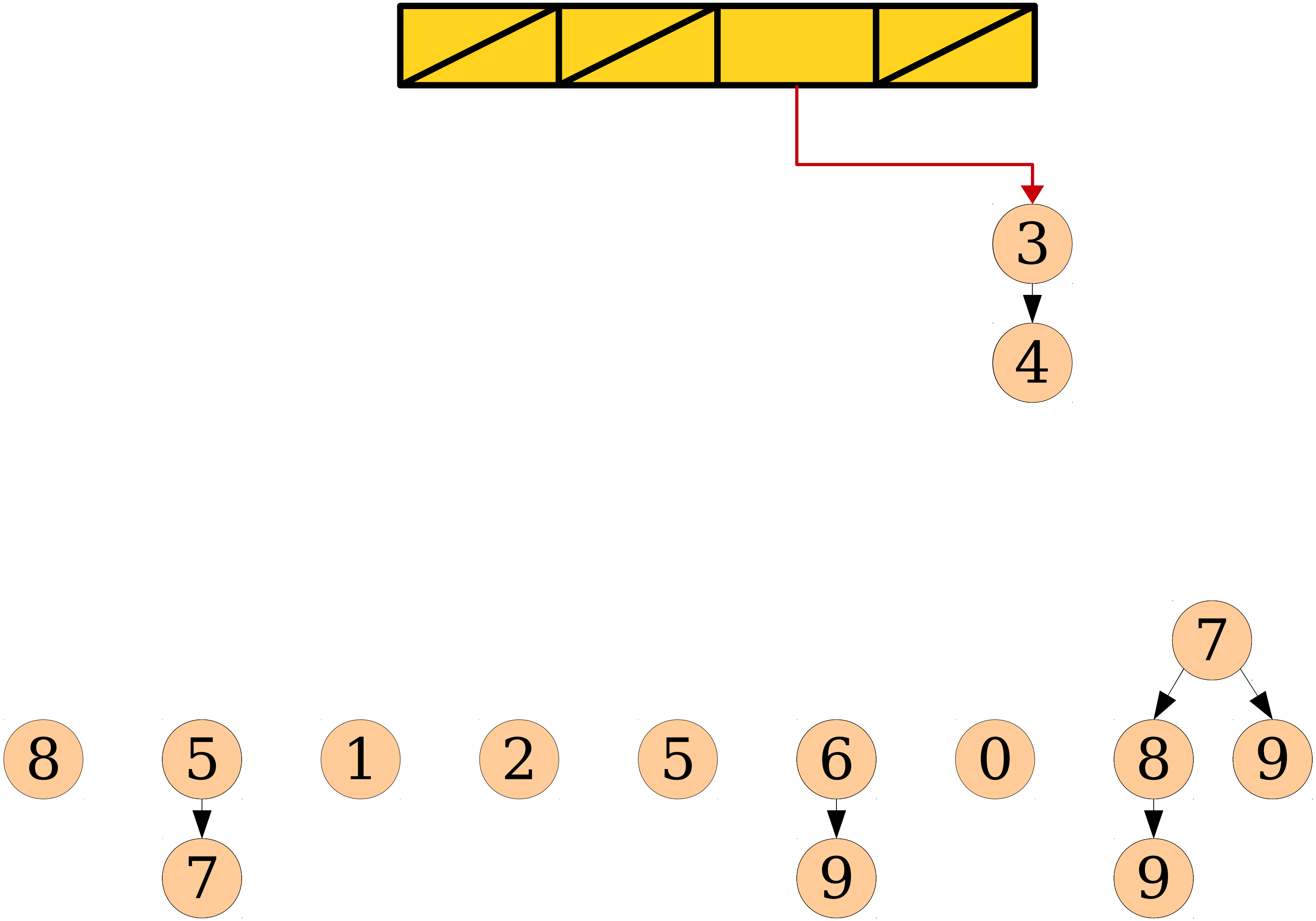


Counting Sort Coalescing

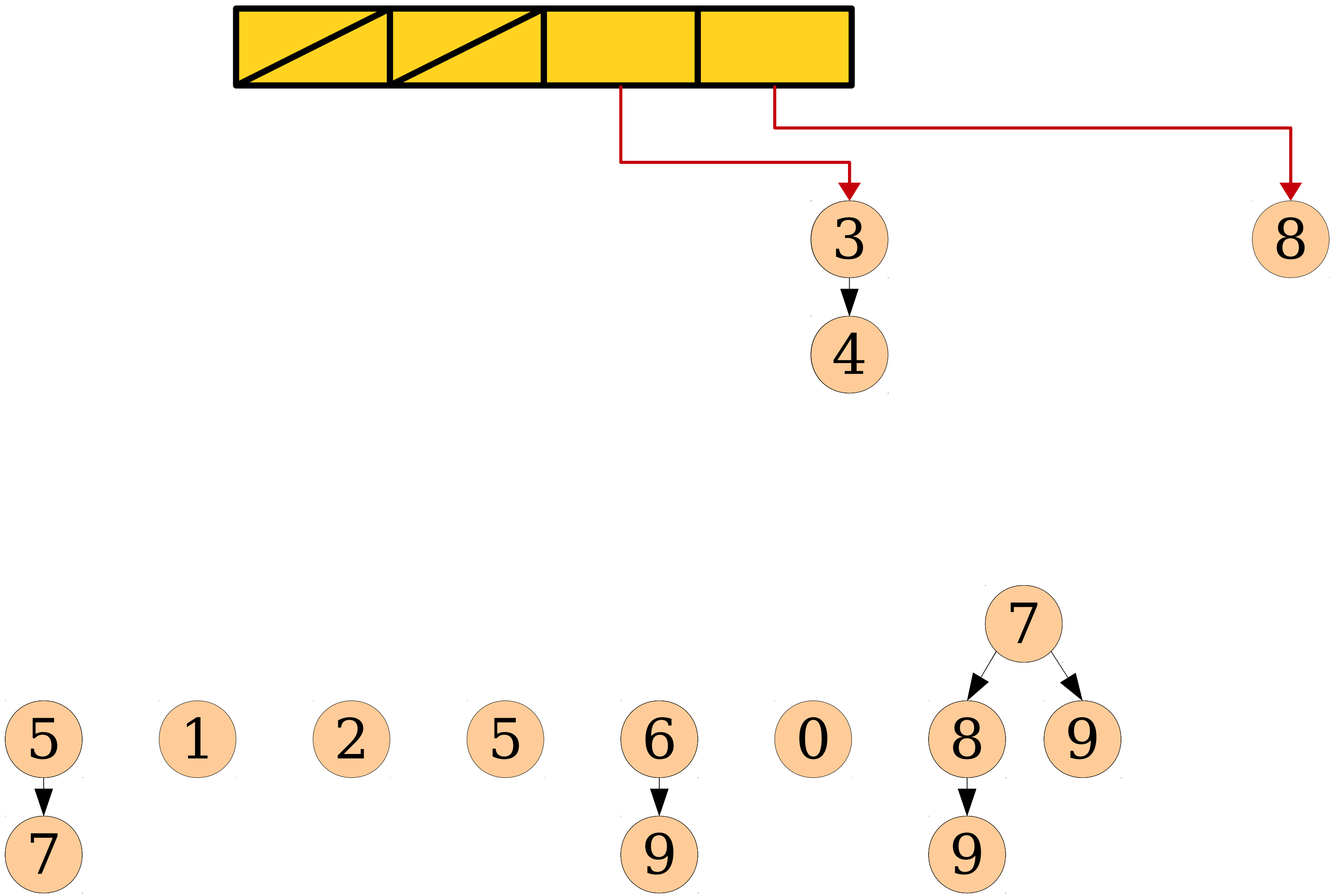




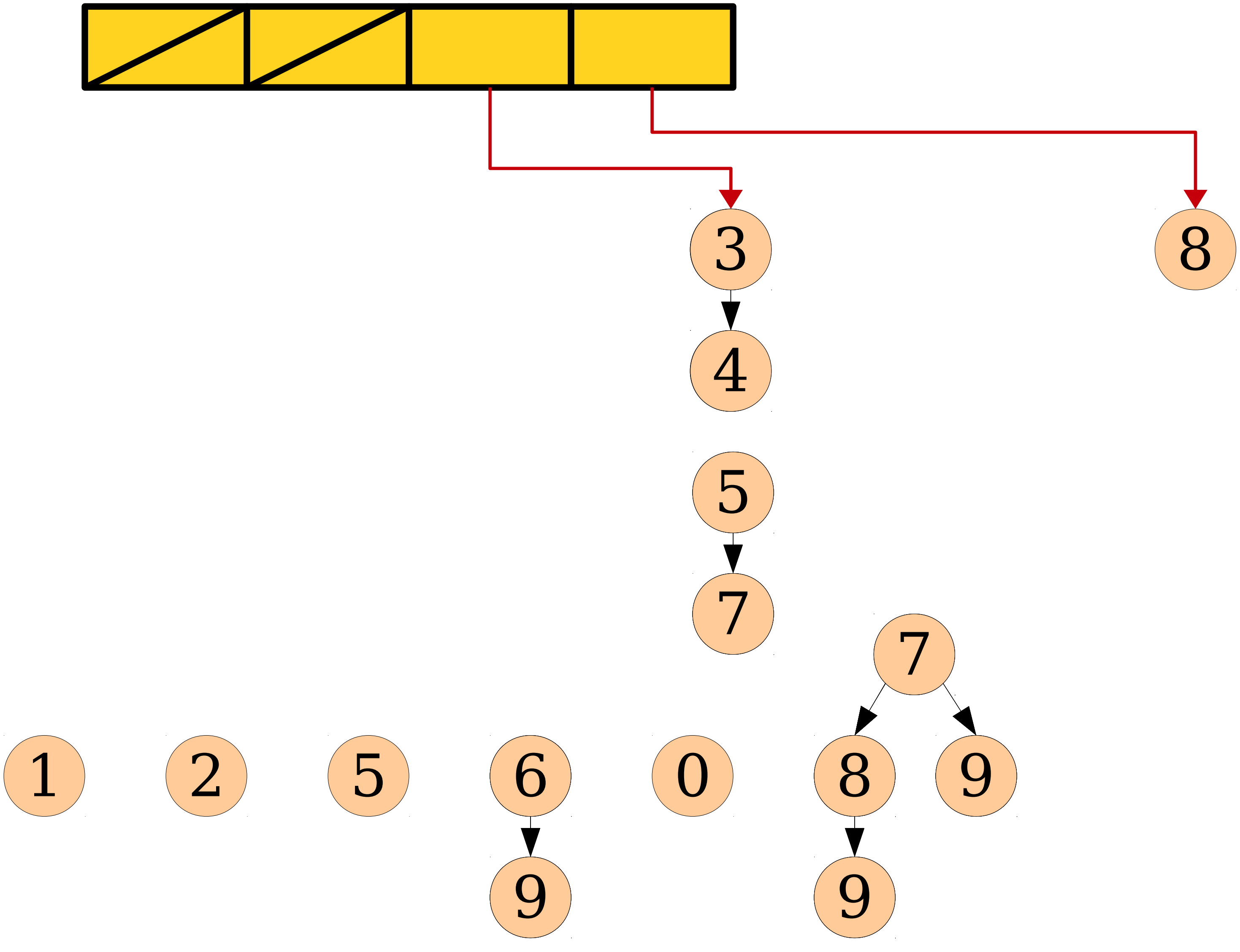
Counting Sort Coalescing



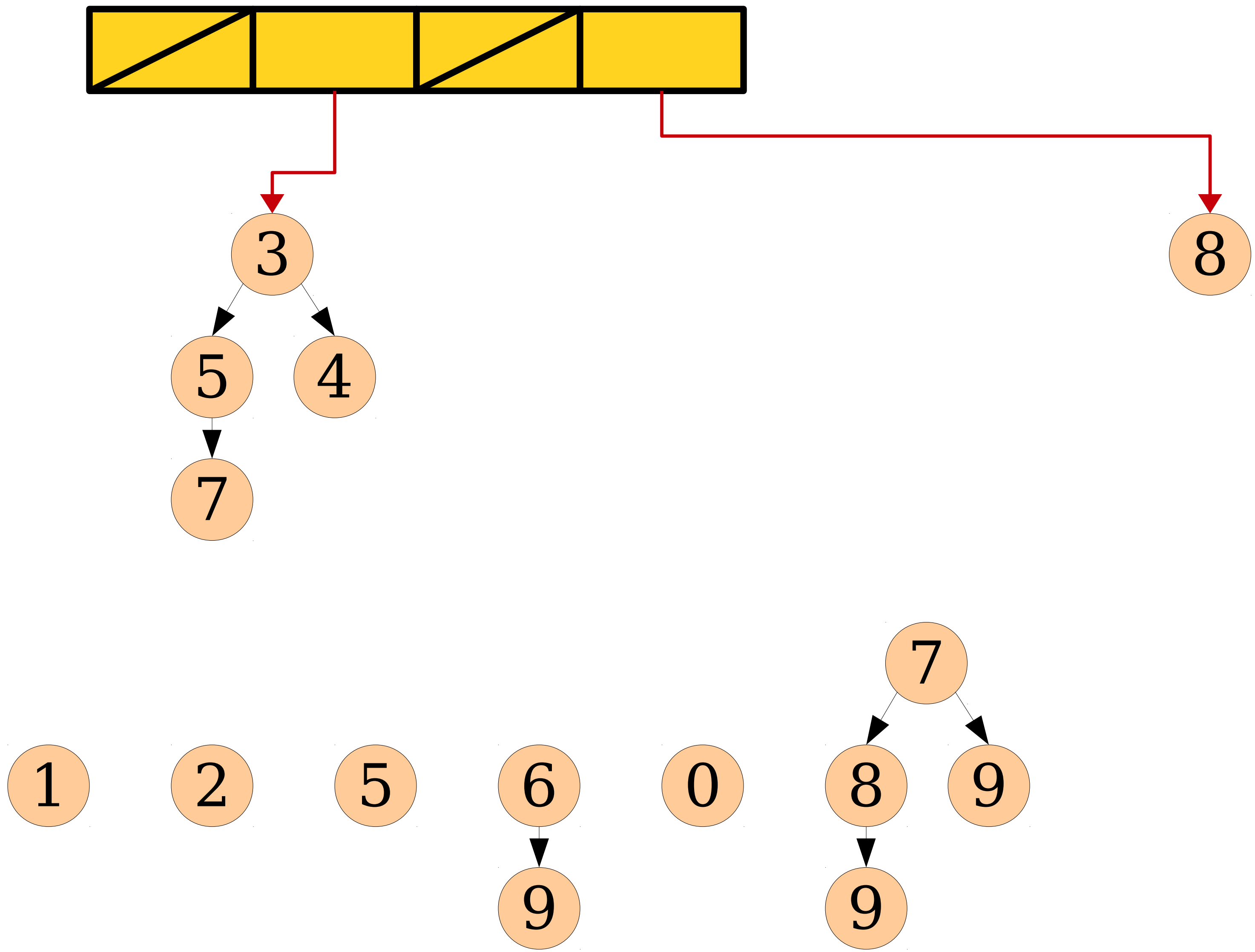
Counting Sort Coalescing



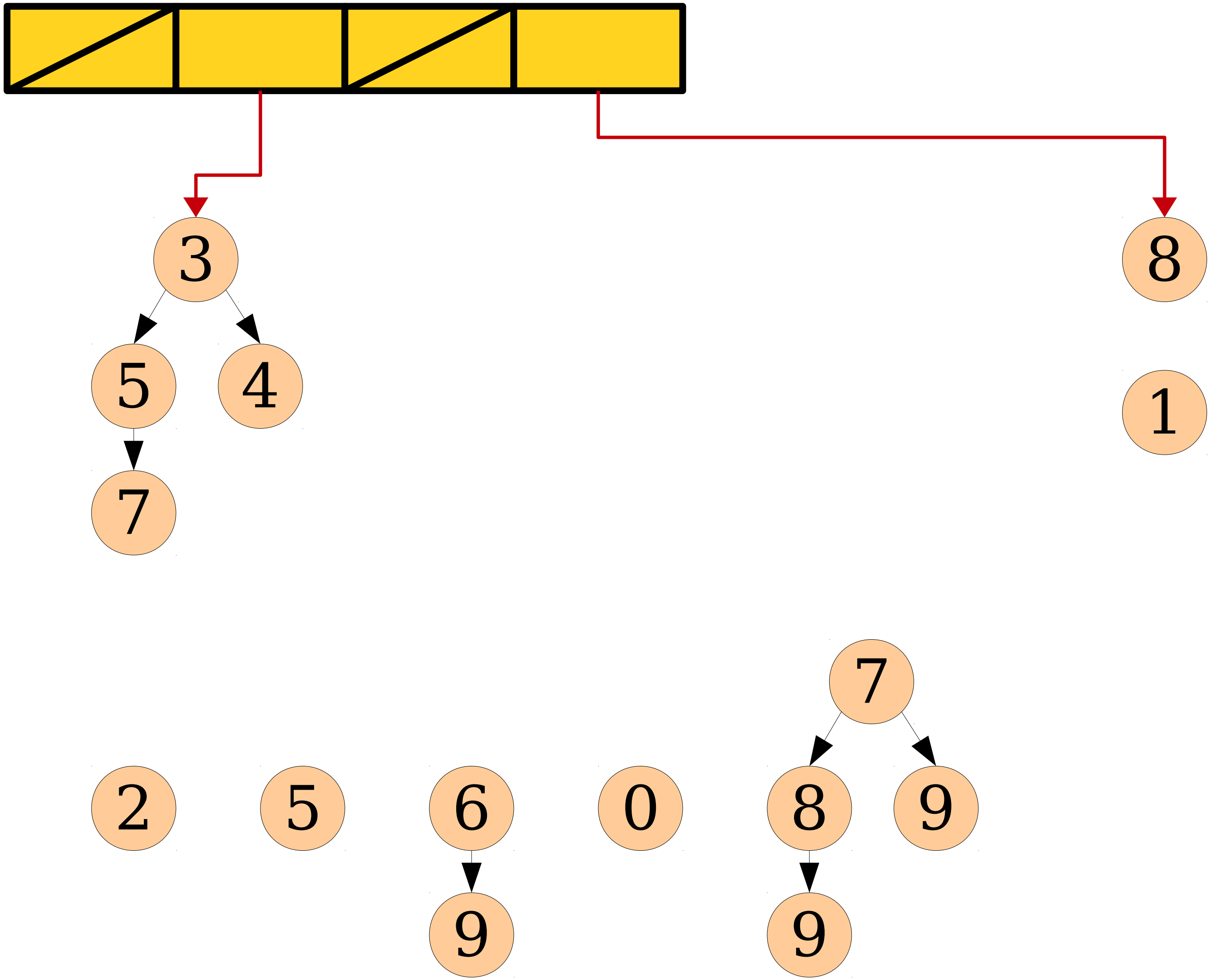
Counting Sort Coalescing



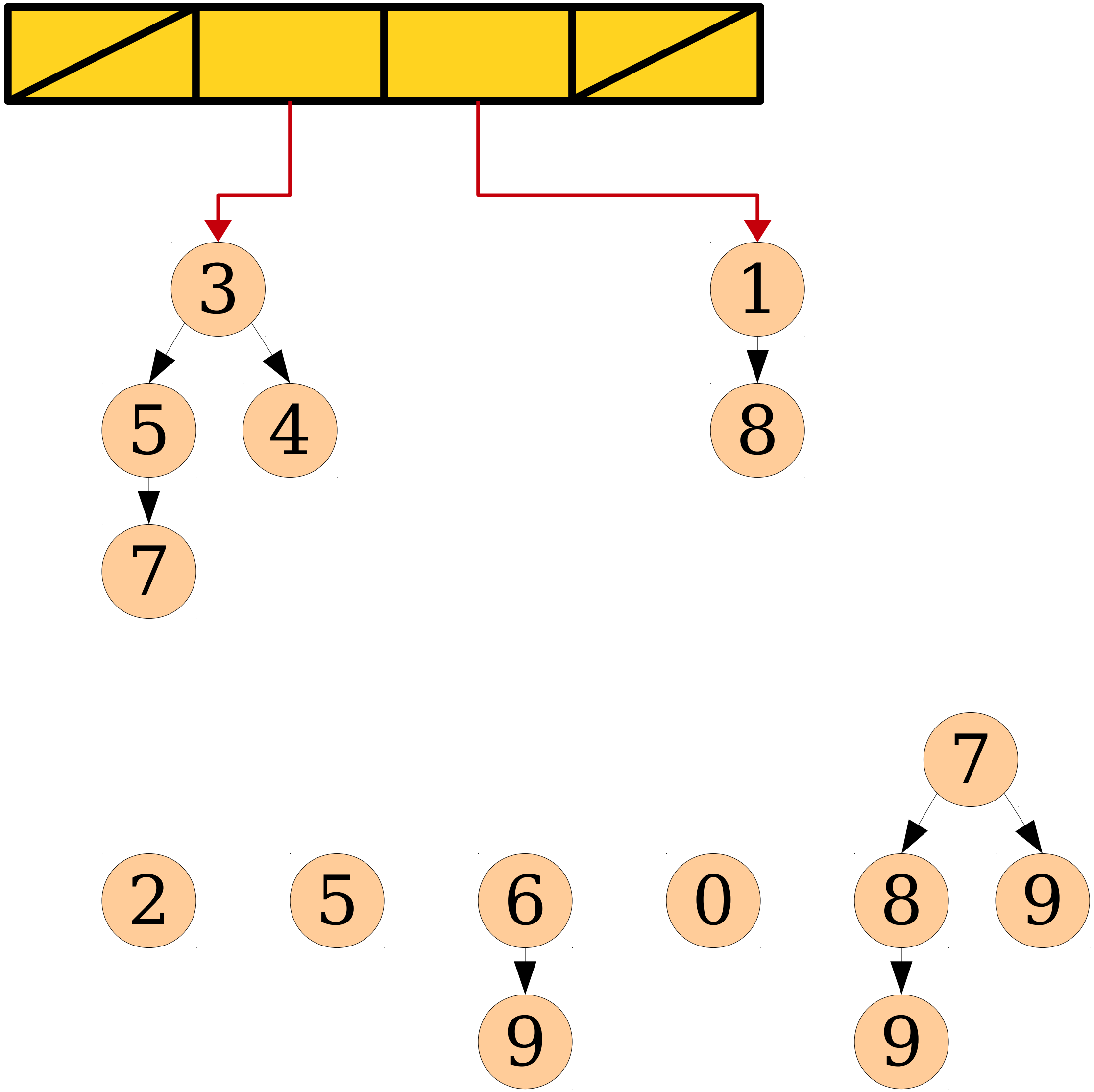
Counting Sort Coalescing



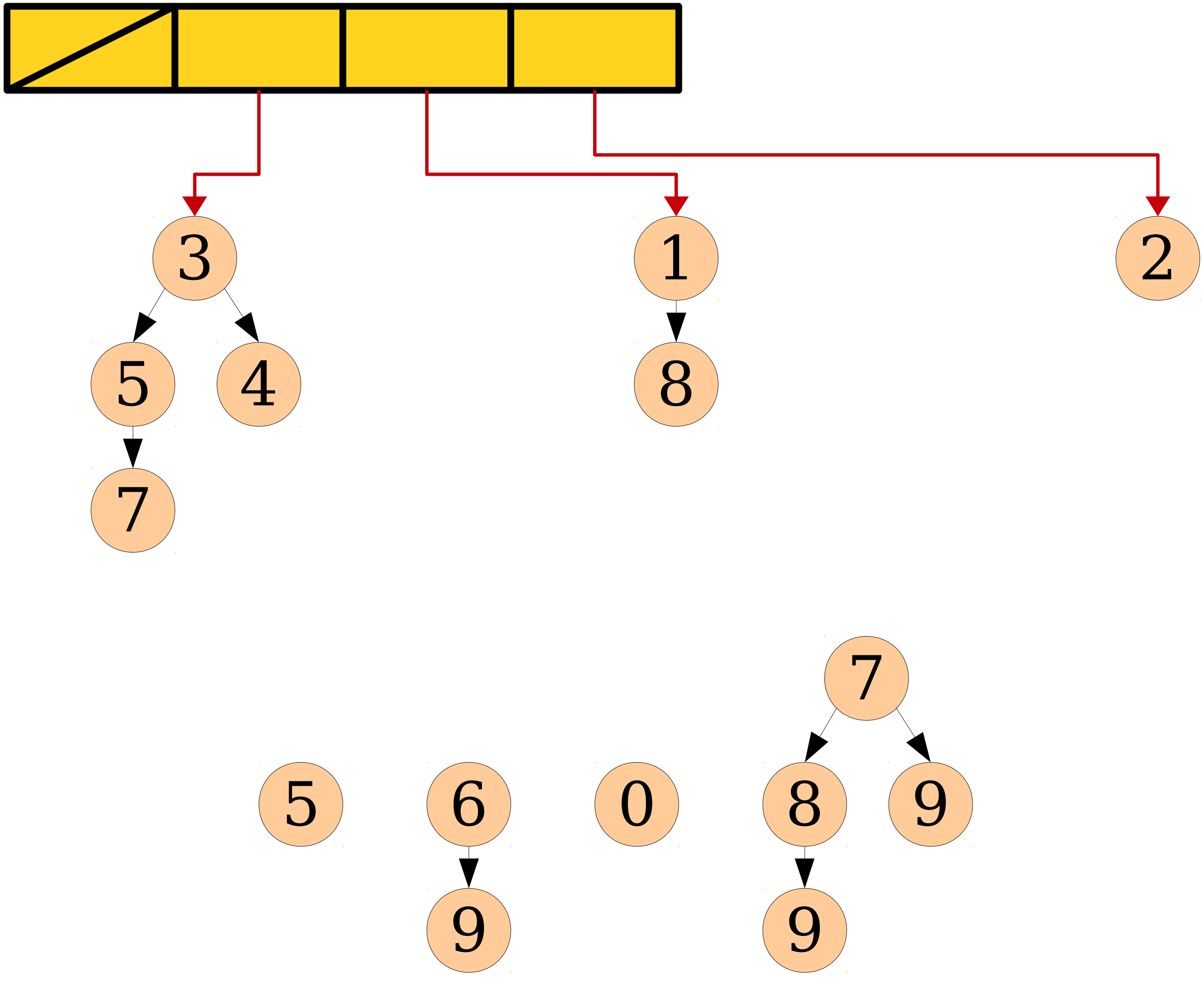
Counting Sort Coalescing



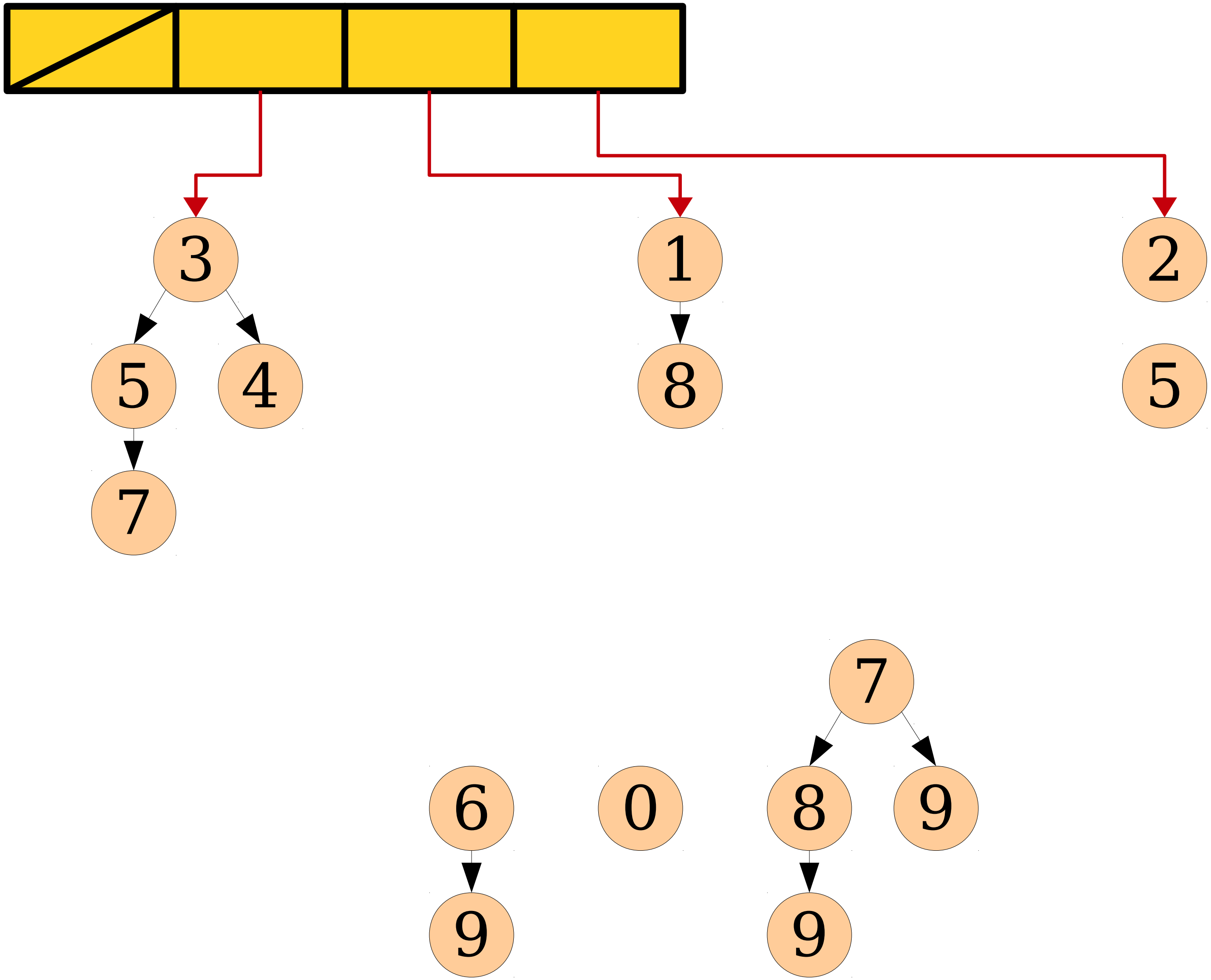
Counting Sort Coalescing



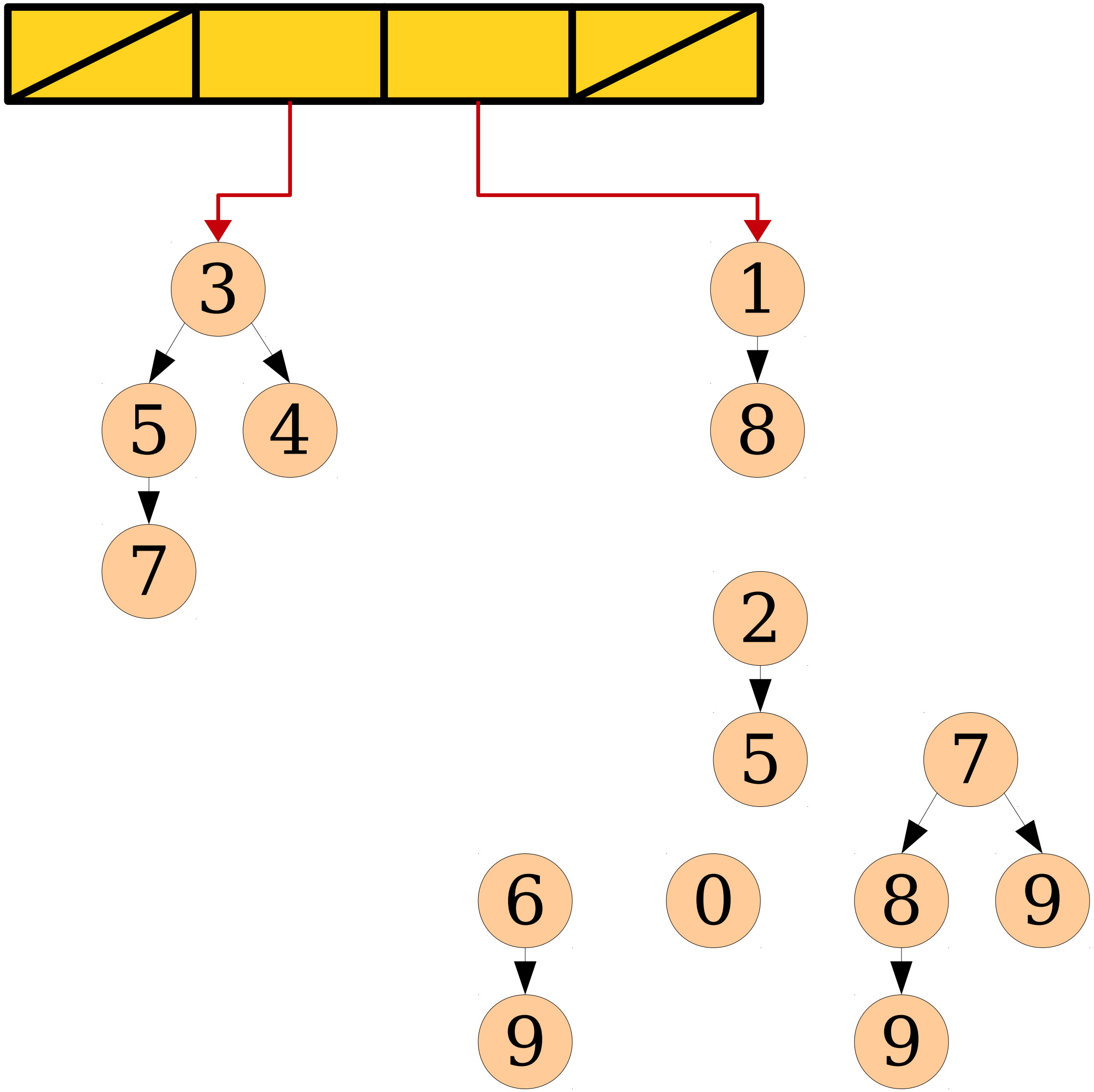
Counting Sort Coalescing



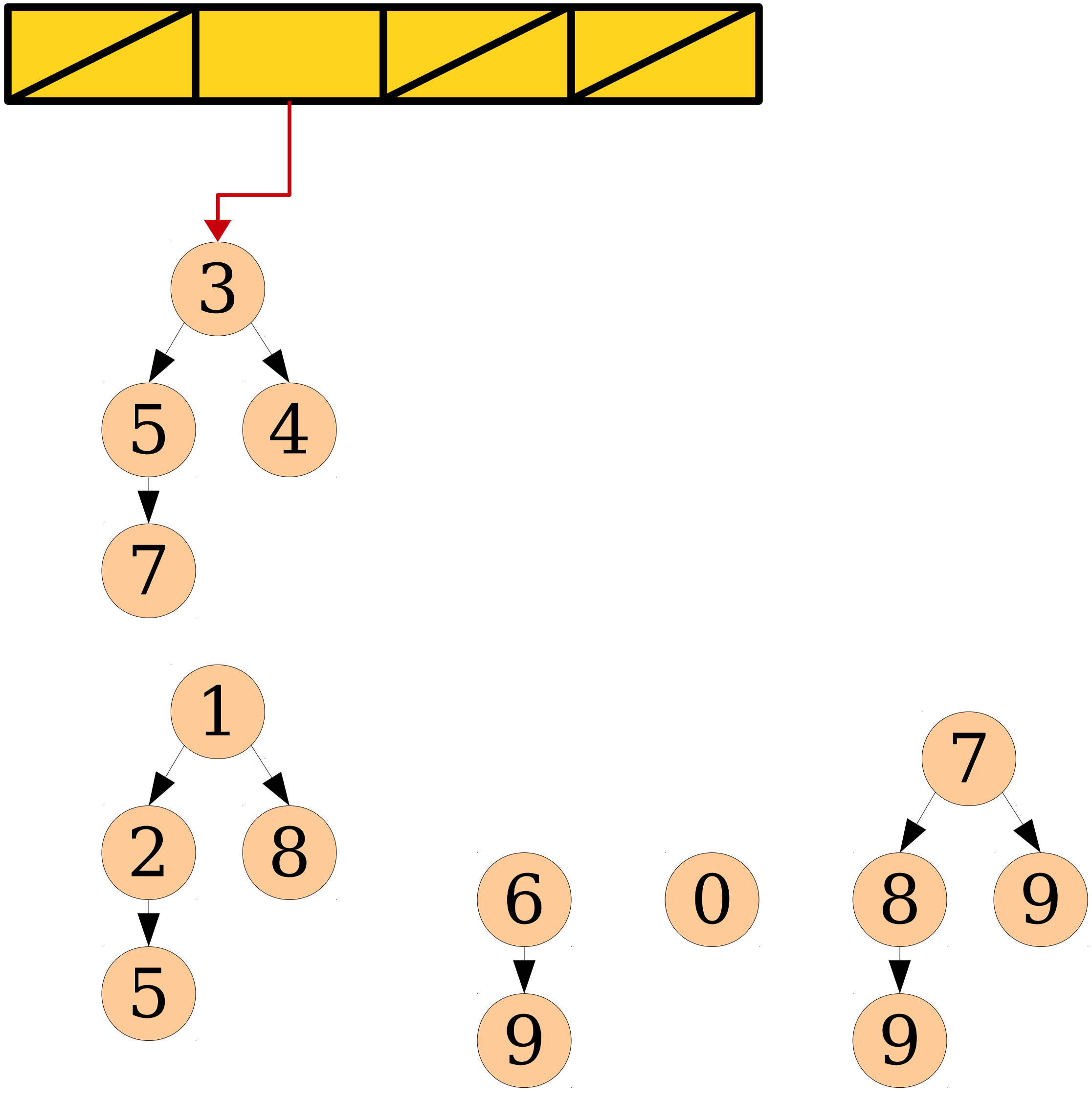
Counting Sort Coalescing



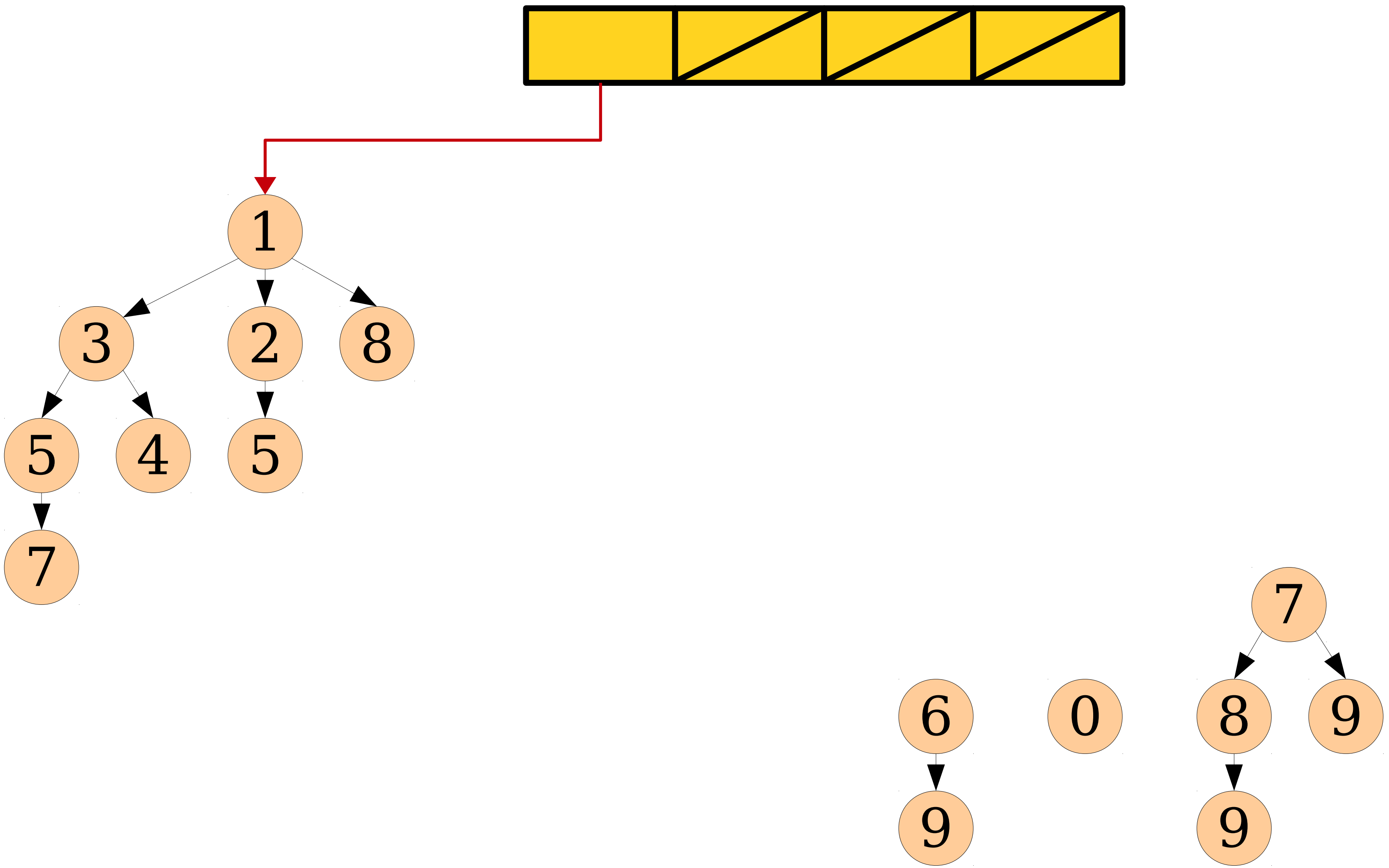
Counting Sort Coalescing



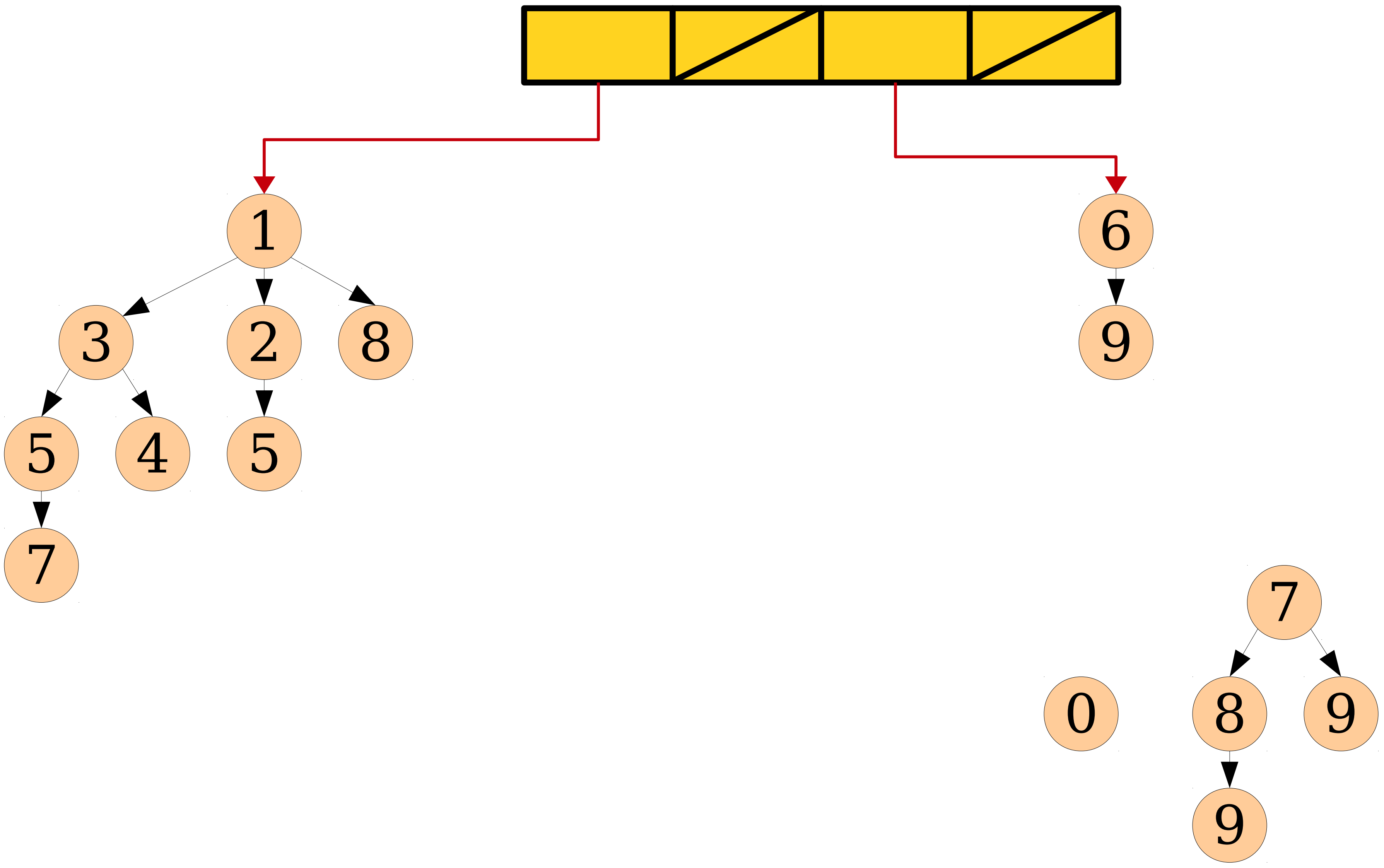
Counting Sort Coalescing



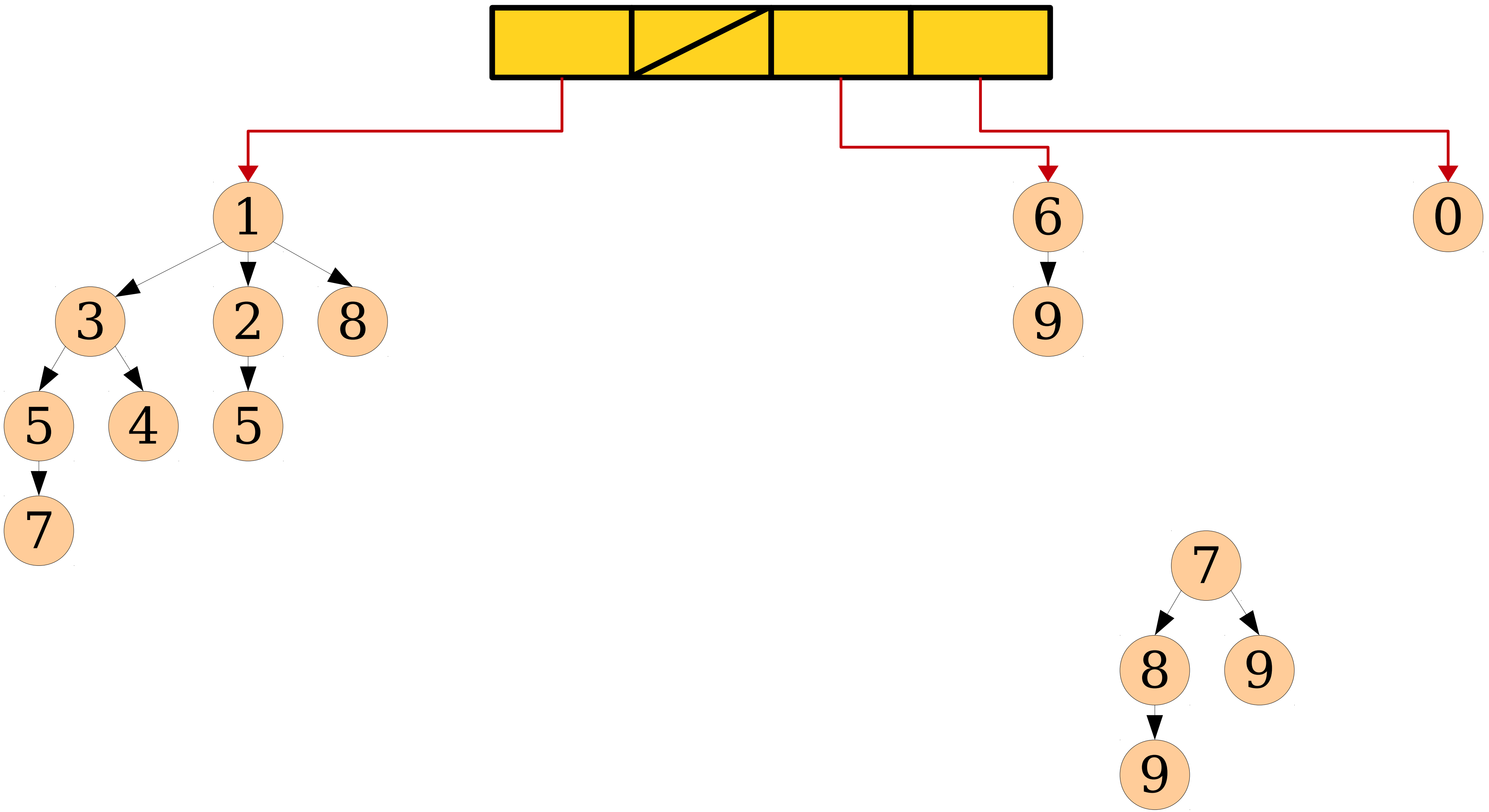
Counting Sort Coalescing



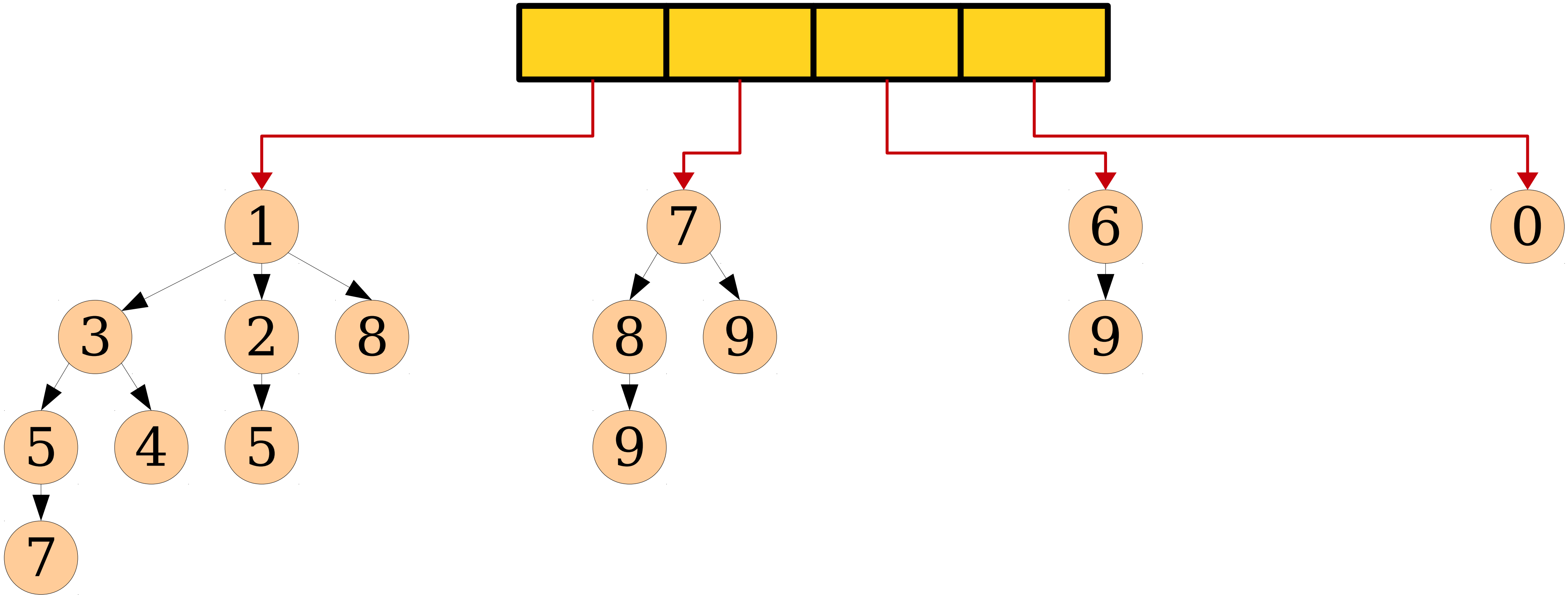
Counting Sort Coalescing



Counting Sort Coalescing



Counting Sort Coalescing



Question: How fast is this?

Analyzing Coalesce

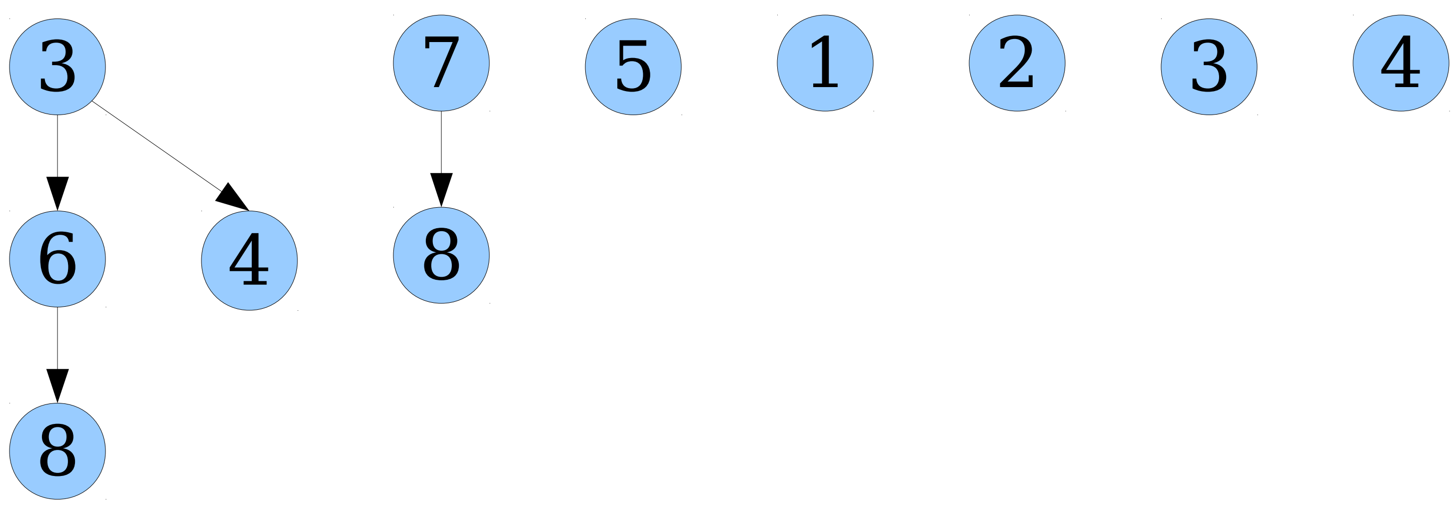
- ***Claim:*** Coalescing a group of k trees takes time $O(k)$.
  - We visit each tree once to place it into its bucket. Time: $O(k)$.
  - Each time we fuse two trees, the number of trees decreases by one.
  - We end with at most $O(\log n)$ trees at the end. *(Why?)*
  - Total number of fuses is $O(k)$.
- Total work done: $O(k)$.

The Story So Far

- A binomial 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 (down to O(log n) after an ***extract-min***).

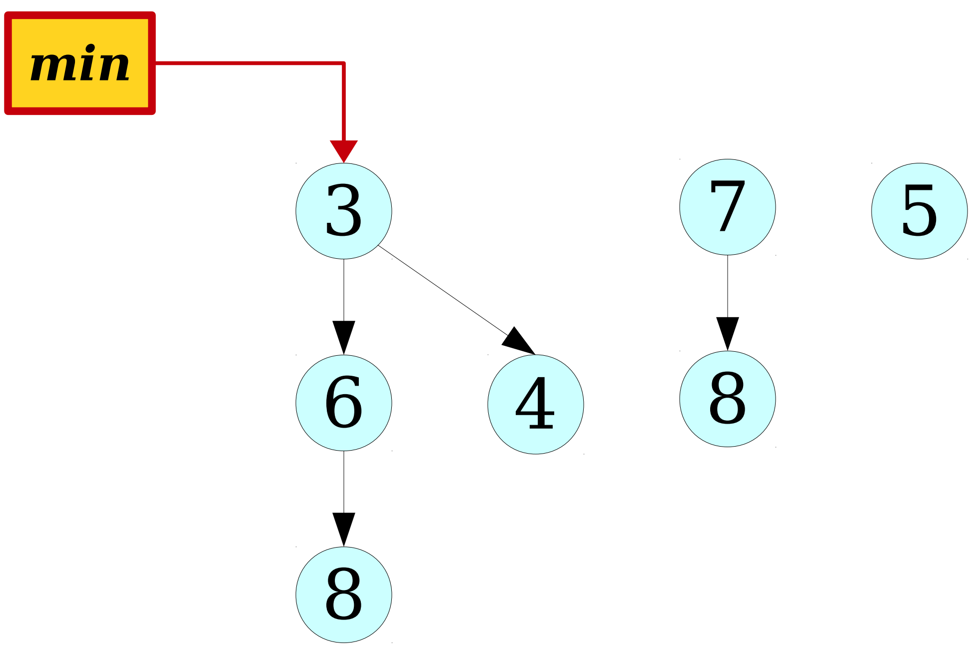
An Amortized Analysis

- Each new tree creates a “mess” we need to clean up.
- ***Idea:*** Use a potential function Φ that’s equal to the number of trees.
- ***Intuition:*** Each operation that increases the number of trees needs to pay for cleanup.



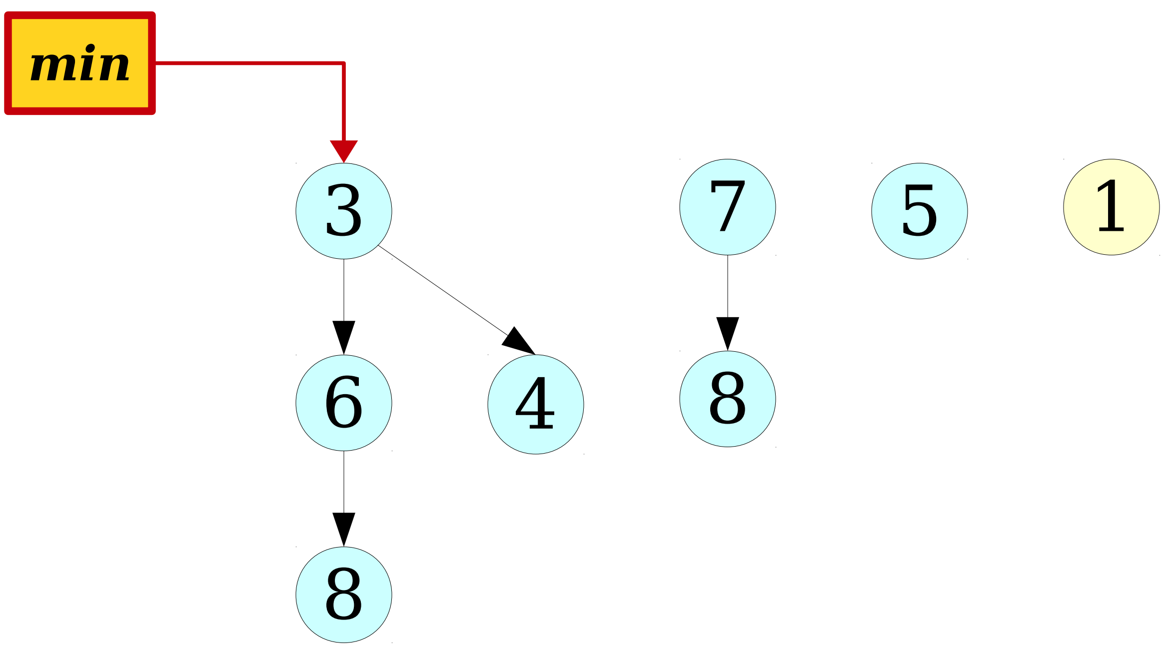
Analyzing an Insertion

- To ***enqueue*** a key, we add a new binomial tree to the forest and possibly update the *min* pointer.



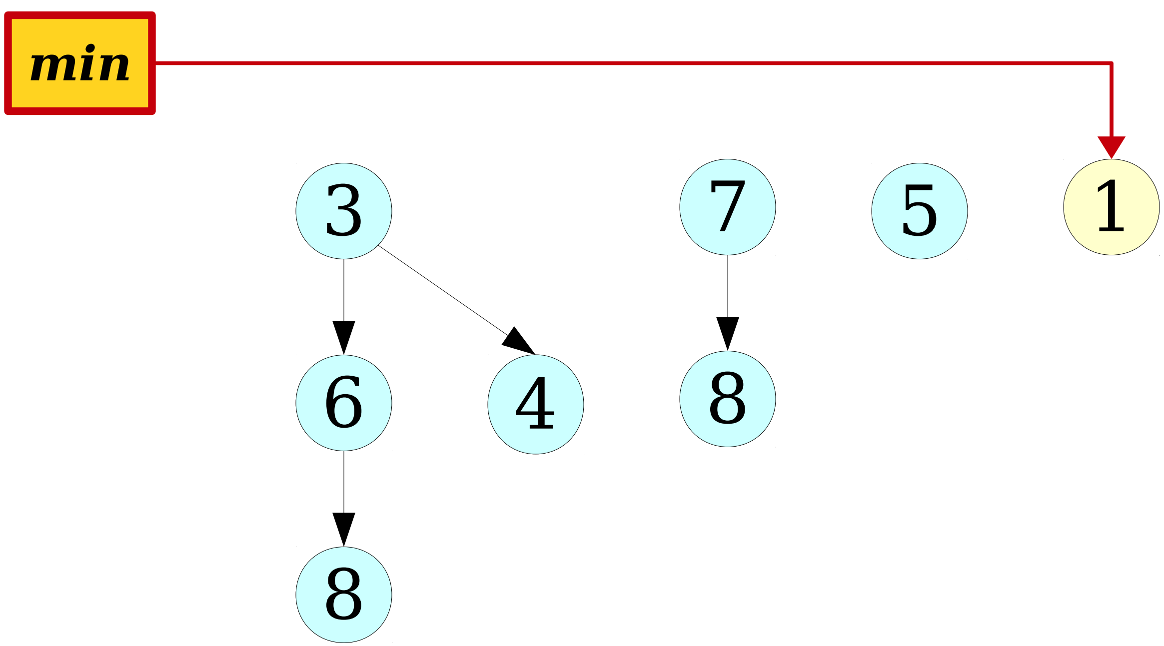
Analyzing an Insertion

- To ***enqueue*** a key, we add a new binomial tree to the forest and possibly update the *min* pointer.



Analyzing an Insertion

- To ***enqueue*** a key, we add a new binomial tree to the forest and possibly update the *min* pointer.

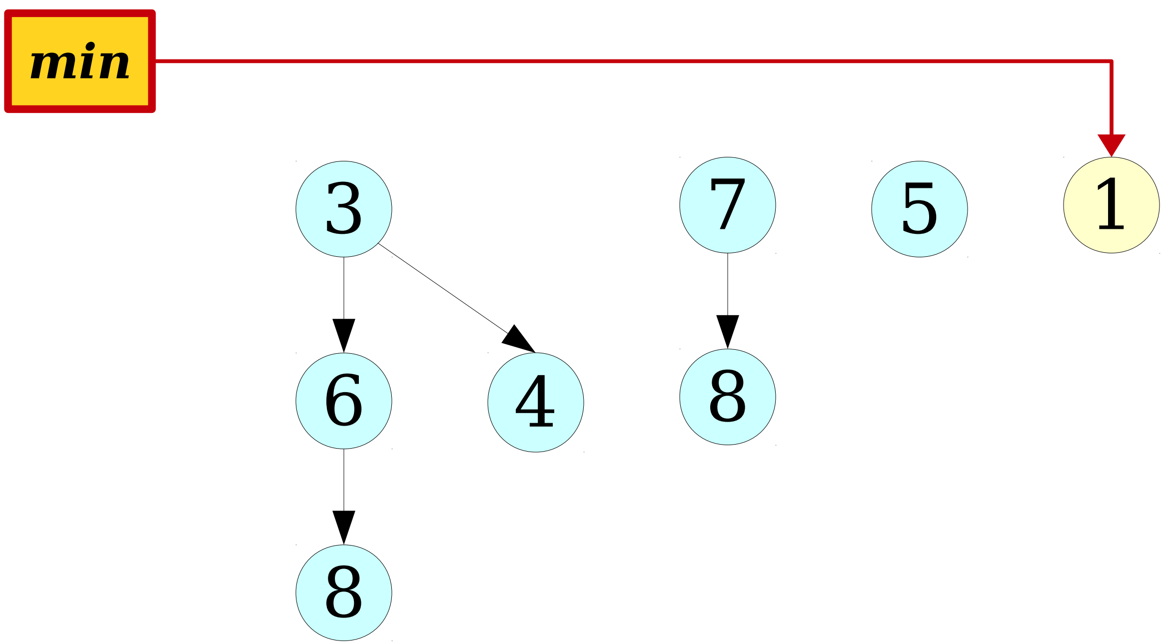


Analyzing an Insertion

- To ***enqueue*** a key, we add a new binomial tree to the forest and possibly update the *min* pointer.

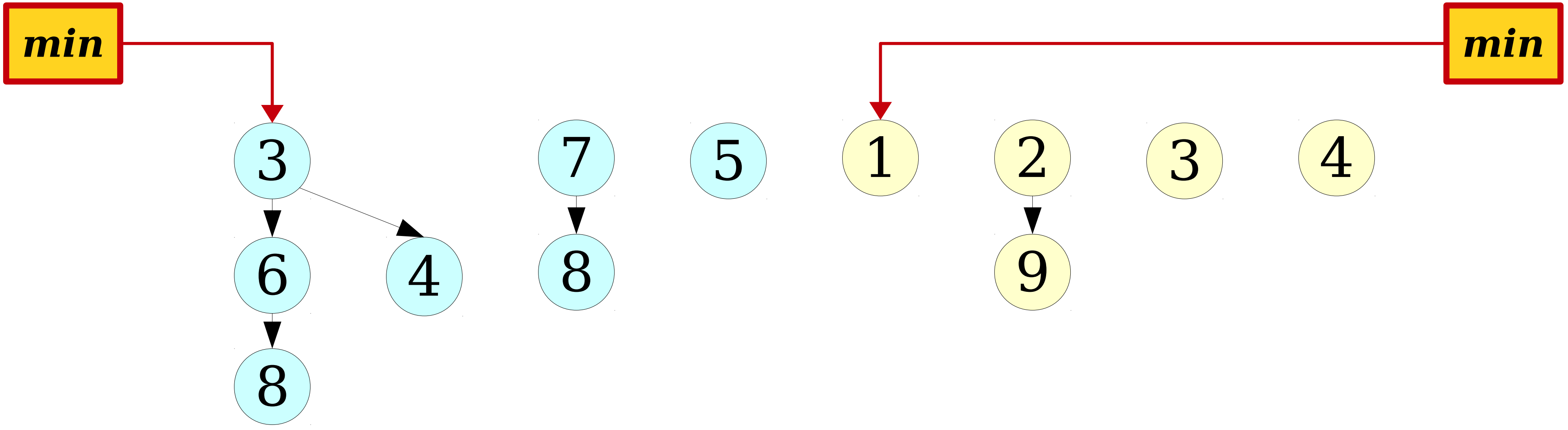
  Actual time: O(1). ΔΦ: +1

  Amortized cost: **O(1)**.



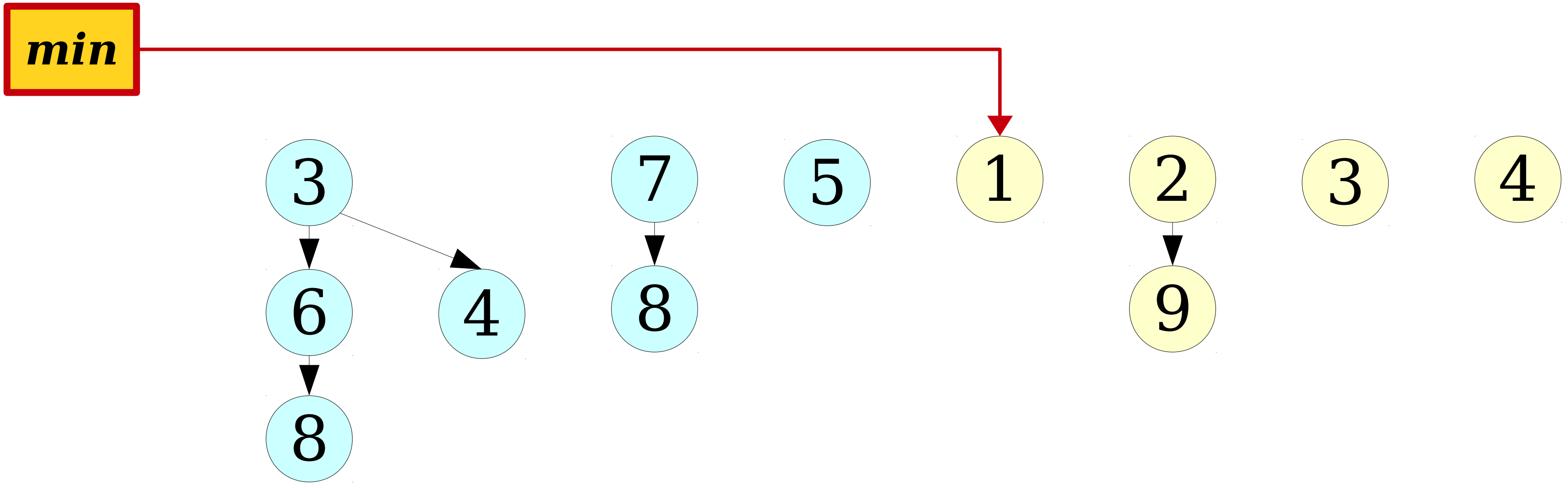
Analyzing a Meld

- Suppose that we ***meld*** two lazy binomial heaps B_1 and B_2. Actual cost: O(1).



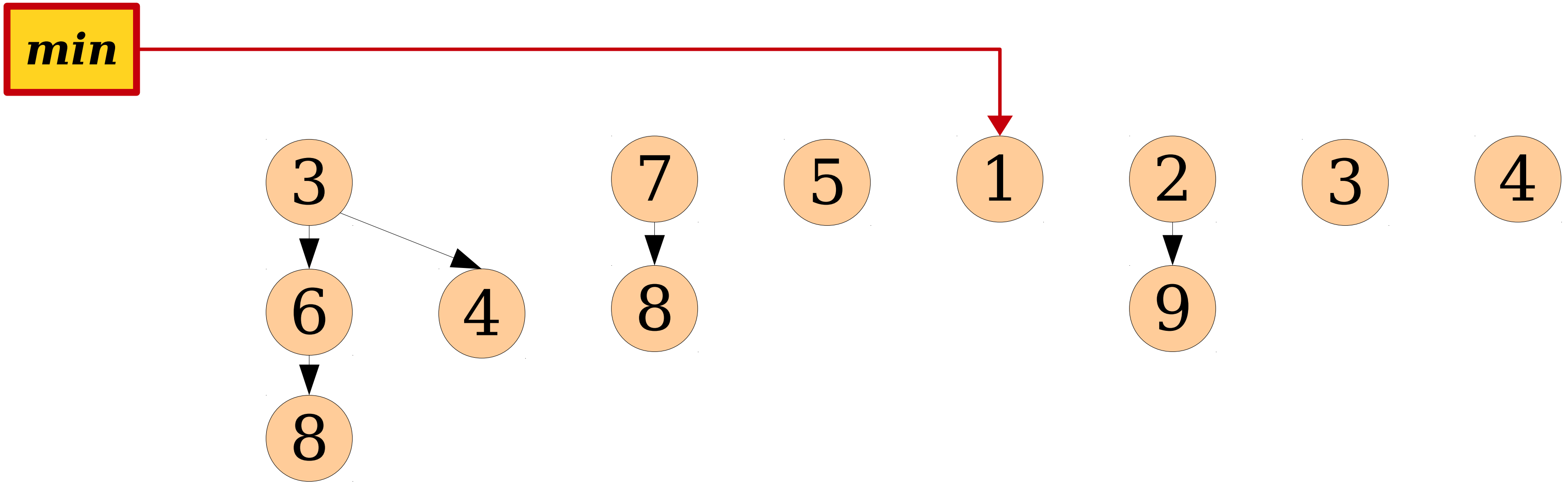
Analyzing a Meld

- Suppose that we ***meld*** two lazy binomial heaps B_1 and B_2. Actual cost: O(1).



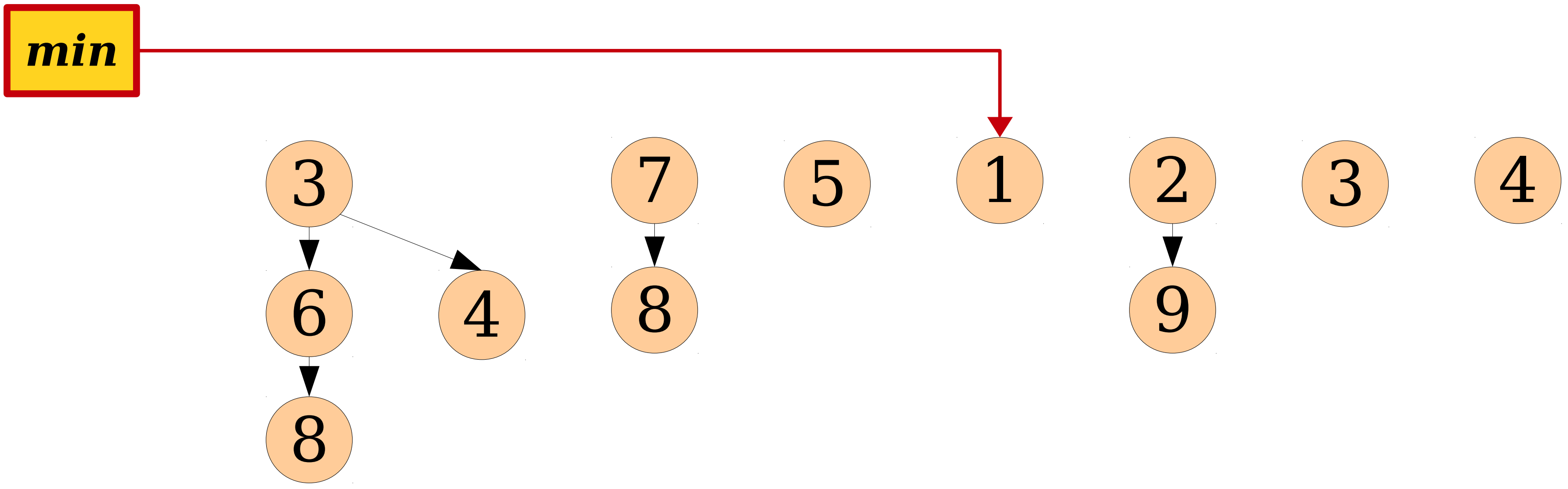
Analyzing a Meld

- Suppose that we ***meld*** two lazy binomial heaps B_1 and B_2. Actual cost: O(1).



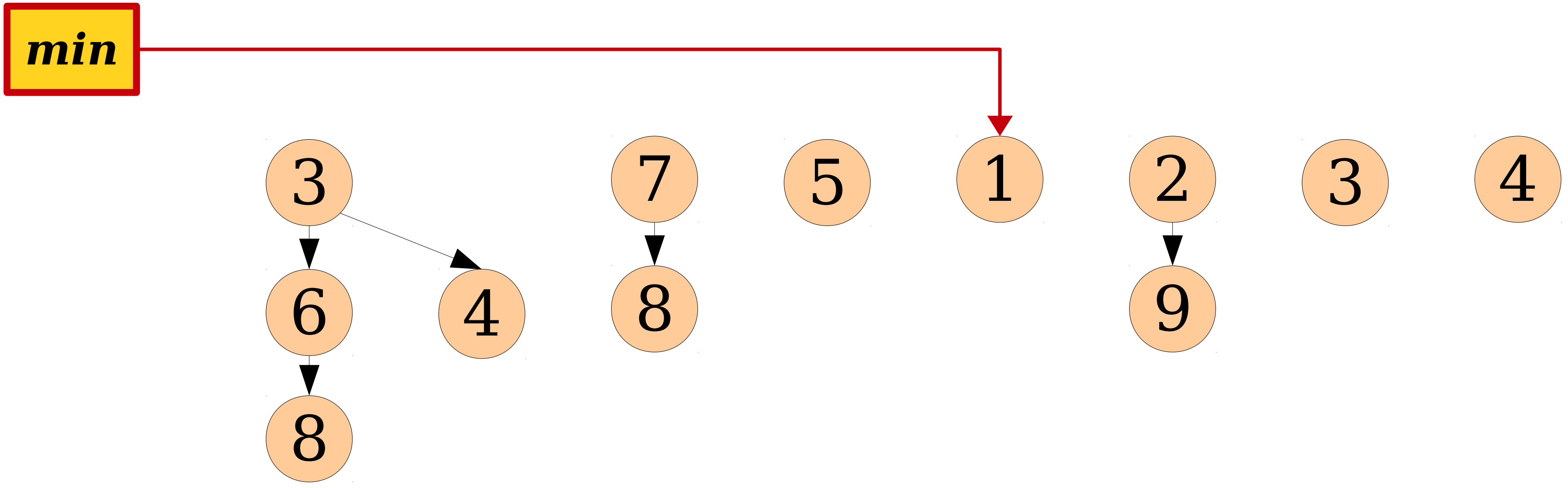
Analyzing a Meld

- Suppose that we ***meld*** two lazy binomial heaps B_1 and B_2. Actual cost: O(1).
- We have the same number of trees before and after we do this.
- Amortized cost: **O(1)**. *(Prove this!)*

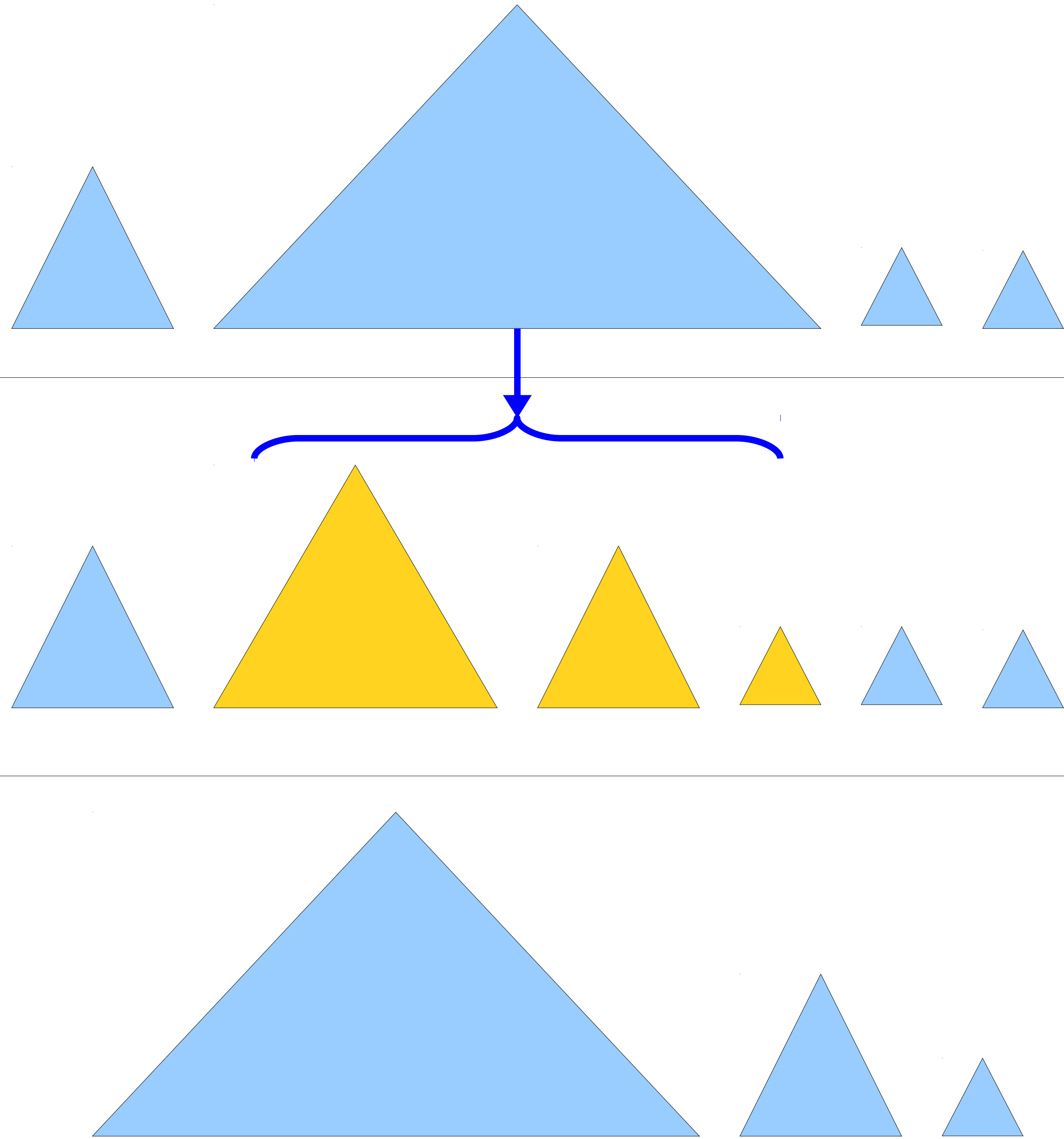


Analyzing a Find-Min

- Each ***find-min*** does O(1) work and does not add or remove trees.
- Amortized cost: **O(1)**.



Analyzing ***extract-min***



Find tree with minimum key.

Work: O(1)
$\Phi = T$

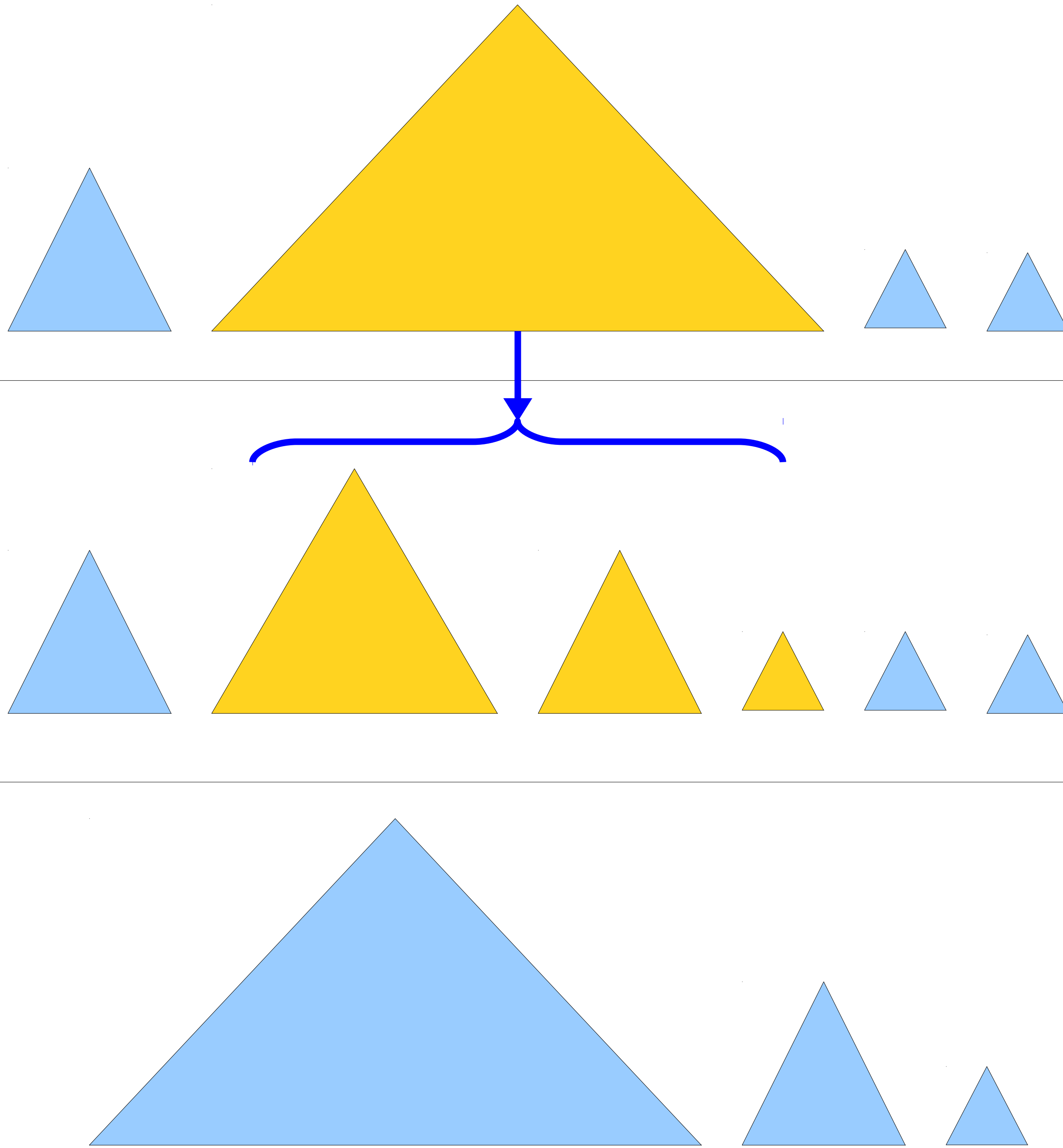
Remove min. Add children to list of trees.

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

Compact until at most one tree of each order.

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

Work: $\mathbf{O(\mathit{T} + \log \mathit{n})}$ **$\Delta\Phi$: $\mathbf{O(-\mathit{T} + \log \mathit{n})}$**



Find tree with minimum key.

Work: O(1)
$\Phi = T$

Remove min. Add children to list of trees.

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

Compact until at most one tree of each order.

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 binomial heap with T trees in it.
- Initially, we expose the children of the minimum element. This increases the number of trees to $T + \mathrm{O}(\log n)$.
- The runtime for coalescing these trees is $\mathrm{O}(T + \log n)$.
- When we're done merging, there will be $\mathrm{O}(\log n)$ trees remaining, so $\Delta\Phi = -T + \mathrm{O}(\log n)$.
- Amortized cost is

$$\mathrm{O}(T + \log n) + \mathrm{O}(1) \cdot (-T + \mathrm{O}(\log n))$$

$$= \mathrm{O}(T) - \mathrm{O}(1) \cdot T + \mathrm{O}(1) \cdot \mathrm{O}(\log n)$$

$$= \mathrm{O}(\log n).$$

The Overall Analysis

- The *amortized* costs of the operations on a lazy binomial heap are as follows:
  - ***enqueue***: O(1)
  - ***meld***: O(1)
  - ***find-min***: O(1)
  - ***extract-min***: O(log n)
- Any series of e ***enqueues*** mixed with d ***extract-min***s will take time O(e + d log e).

Why This Matters

- Lazy binomial heaps are a powerful building block used in many other data structures.
- We'll see one of them, the *Fibonacci heap*, when we come back on Thursday.
- You'll see another (supporting ***add-to-all***) on the problem set.

Major Ideas from Today

- Isometries are a *great* way to design data structures.
  - Here, binomial heaps come from binary arithmetic.
- The amortized cost of an operation has to be evaluated in a broader context.
  - Insertions in a regular binary heap are only amortized O(1) if you don't do any deletions.
- 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 ***dequeue*** fixes all the prior problems.

Next Time

- ***The Need for decrease-key***
  - A powerful and versatile operation on priority queues.
- ***Fibonacci Heaps***
  - A variation on lazy binomial heaps with efficient decrease-key.
- ***Implementing Fibonacci Heaps***
  - … is harder than it looks!