Binomial Heaps

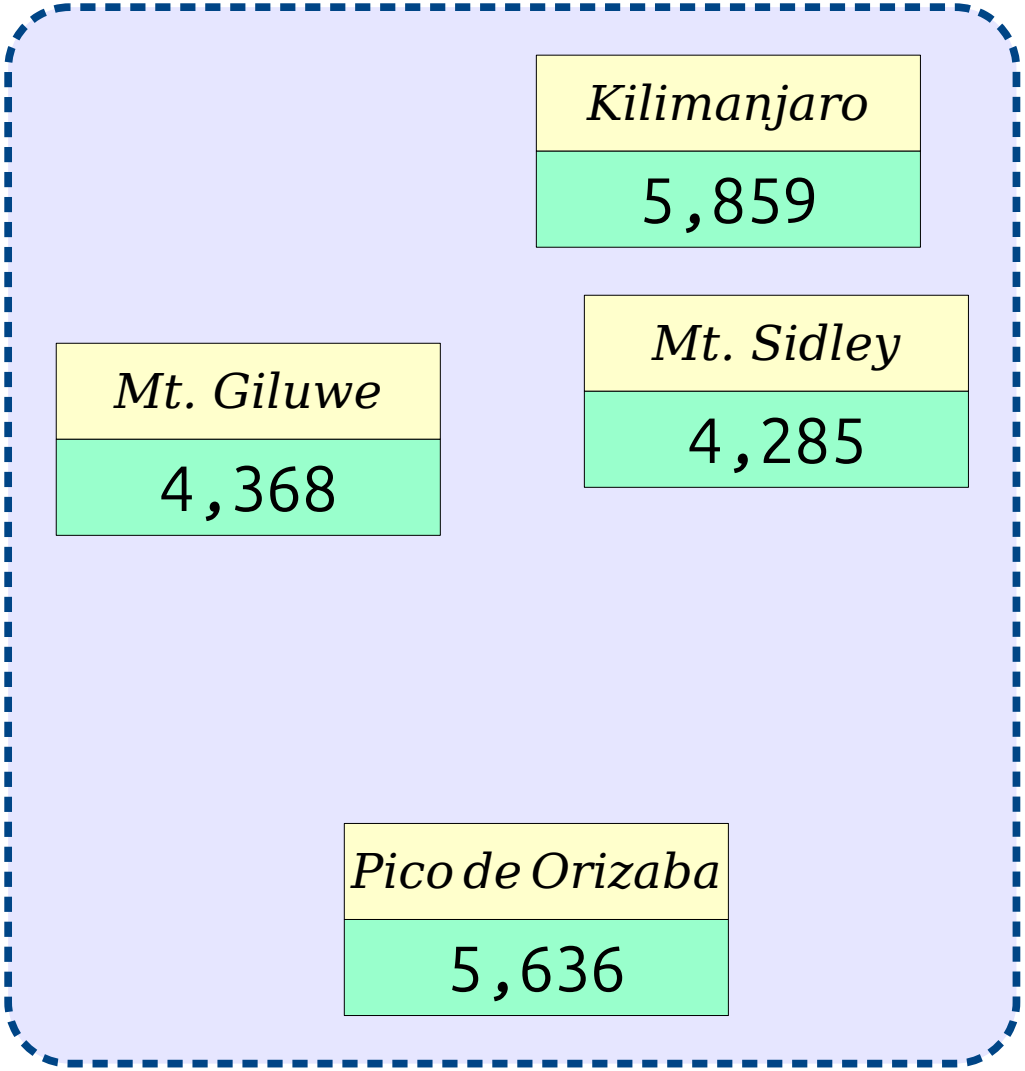
Where We're Going

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



Binary Heaps

- Priority queues are often implemented as ***binary heaps***.
  - ***enqueue*** and ***extract-min*** run in time O(log *n*); ***find-min*** runs in time O(1).
- These heaps are surprisingly fast in practice. It's tough to beat their performance!
  - *d*-ary heaps outperform binary heaps for well-tuned values of *d*, and otherwise only ***sequence heaps*** are known to specifically outperform this family.
  - (Is this information incorrect as of 2026? Let me know and I'll update it.)
- In that case, why do we need other heaps?

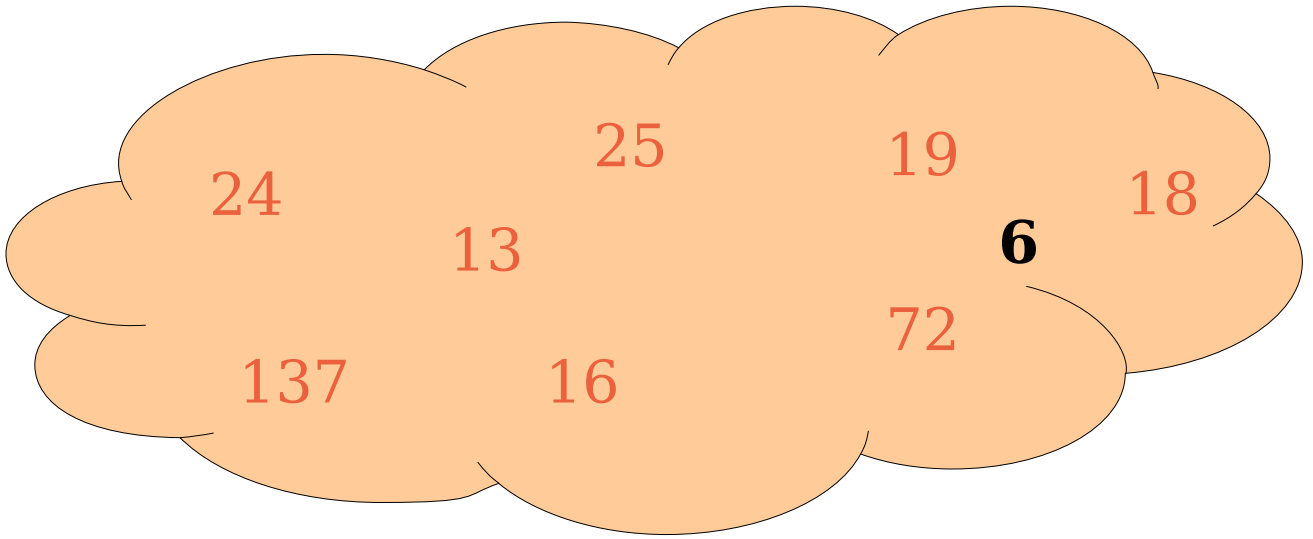
Priority Queues in Practice

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

Meldable Priority Queues

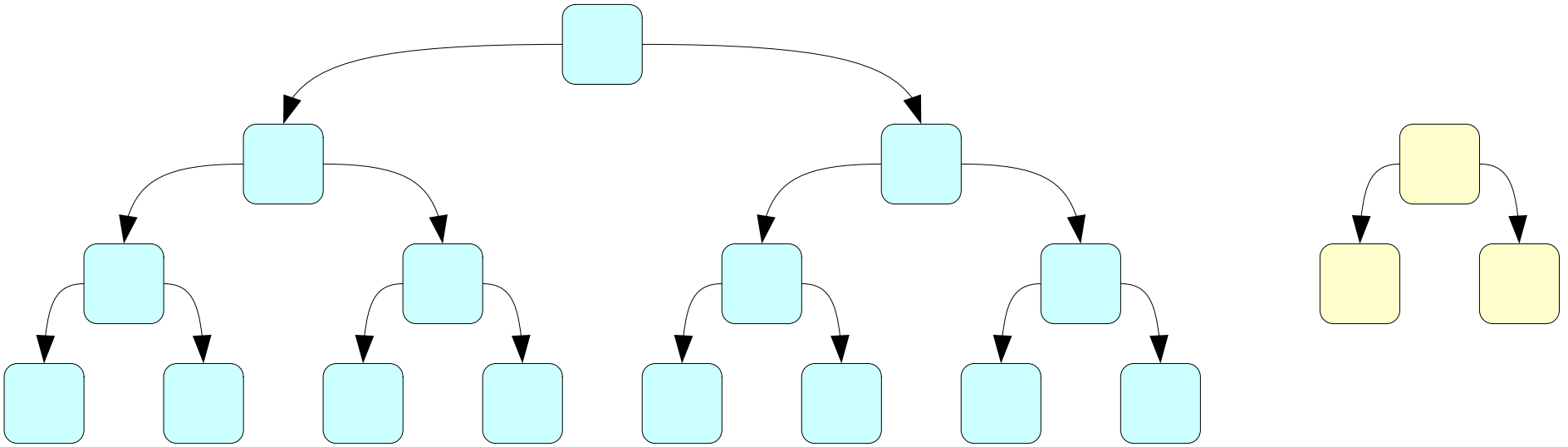
Meldable Priority Queues

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



Efficiently Meldable Queues

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



What things *can* be combined together efficiently?

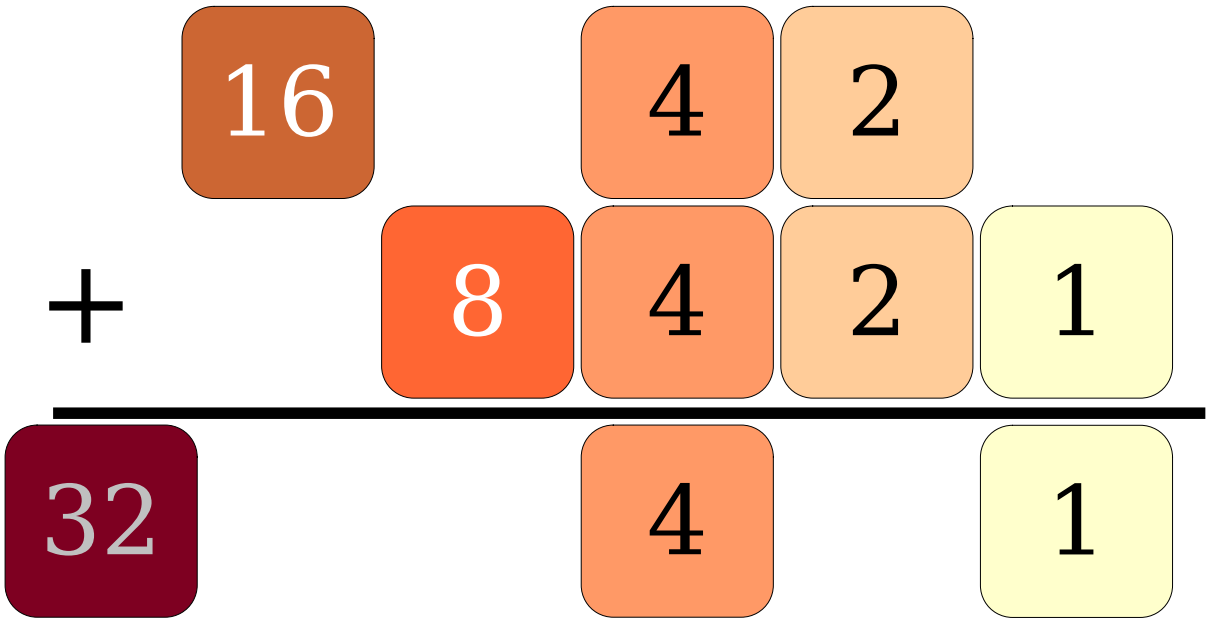
Adding Binary Numbers

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

Intuition:
Writing out n in any "reasonable" base requires $\Theta(\log n)$ digits.

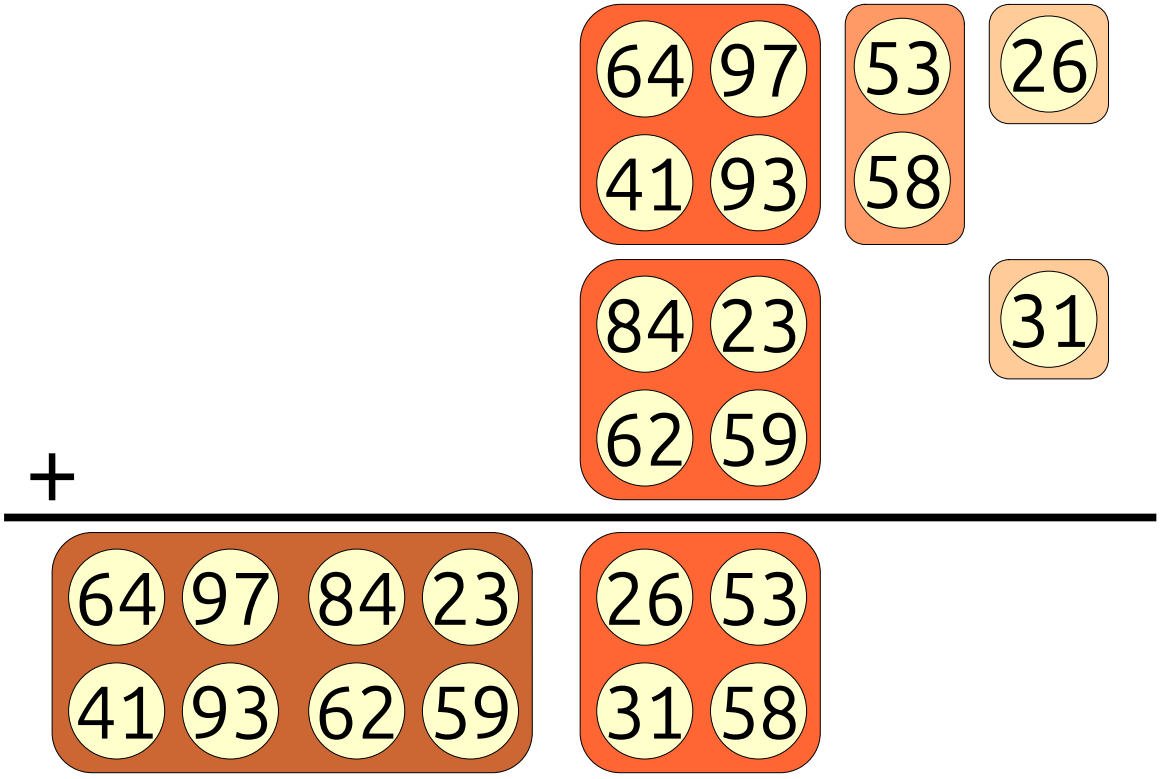
A Different Intuition

- 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



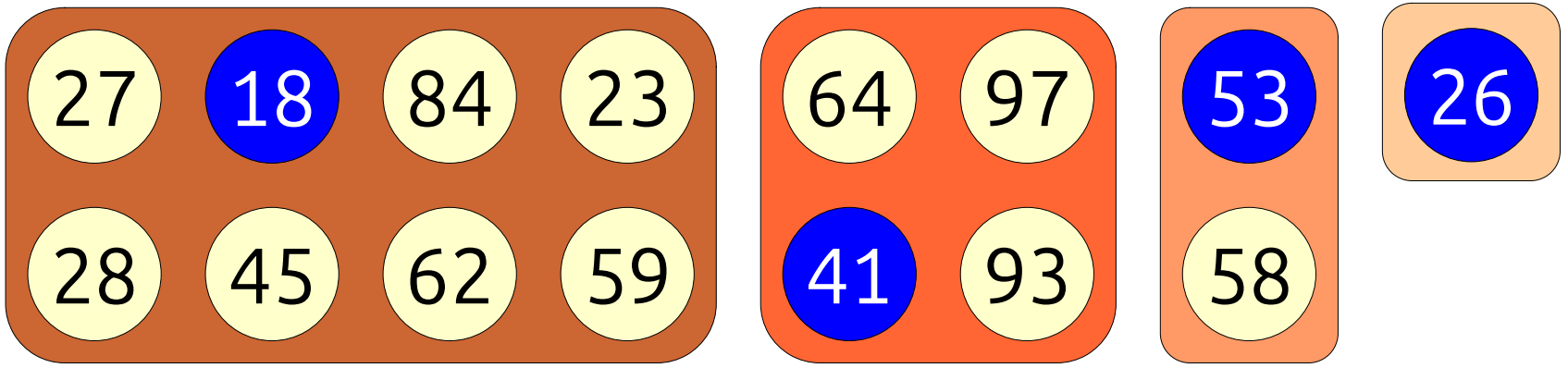
Building a Priority Queue

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



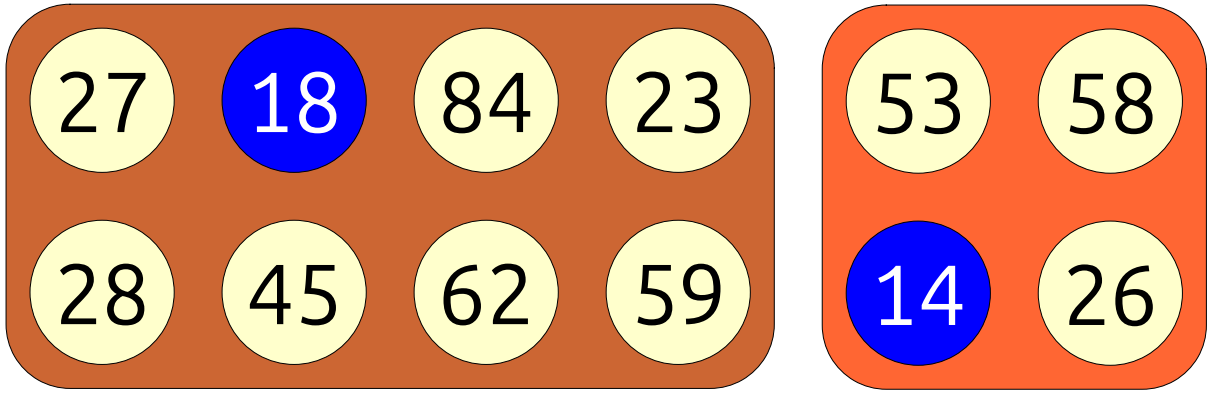
Building a Priority Queue

- 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

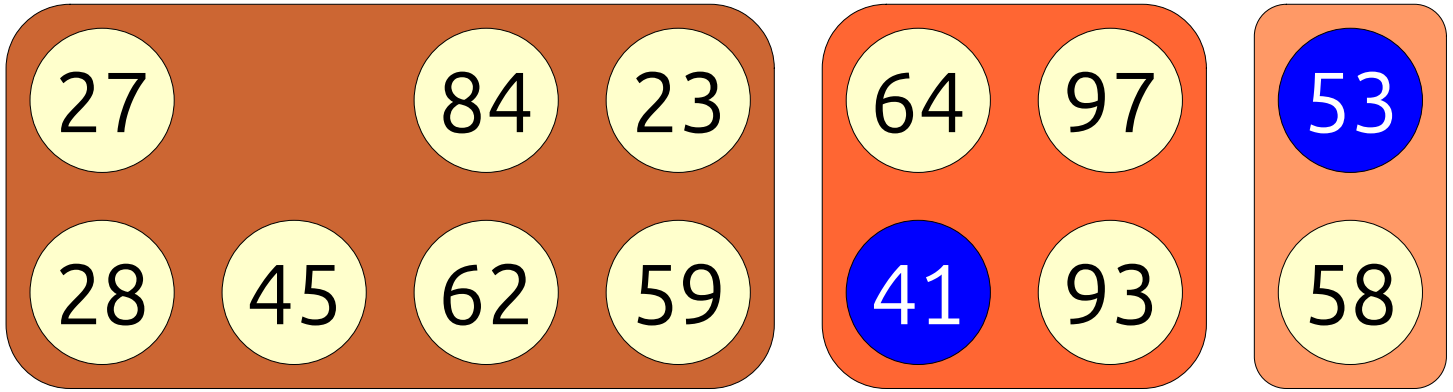
- If we can efficiently meld two priority queues, we can efficiently enqueue elements to the queue.
- ***Idea:*** Meld together the queue and a new queue with a single packet.



Time required:
O(log n) fuses.

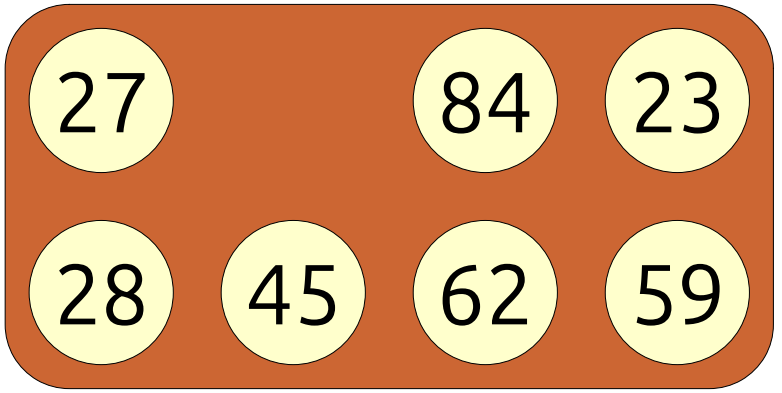
Deleting the Minimum

- 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.



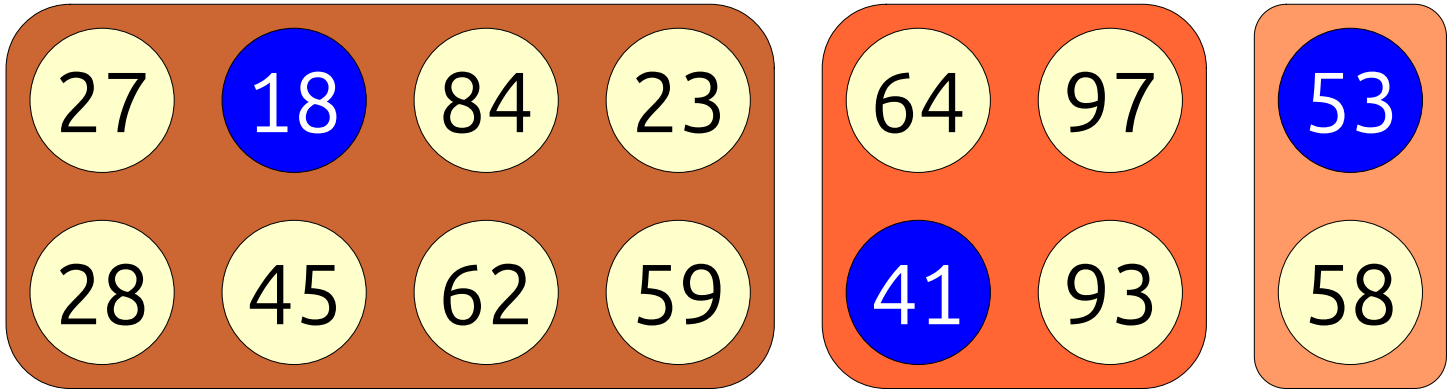
Deleting the Minimum

- 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 smaller packets, then add them back in.



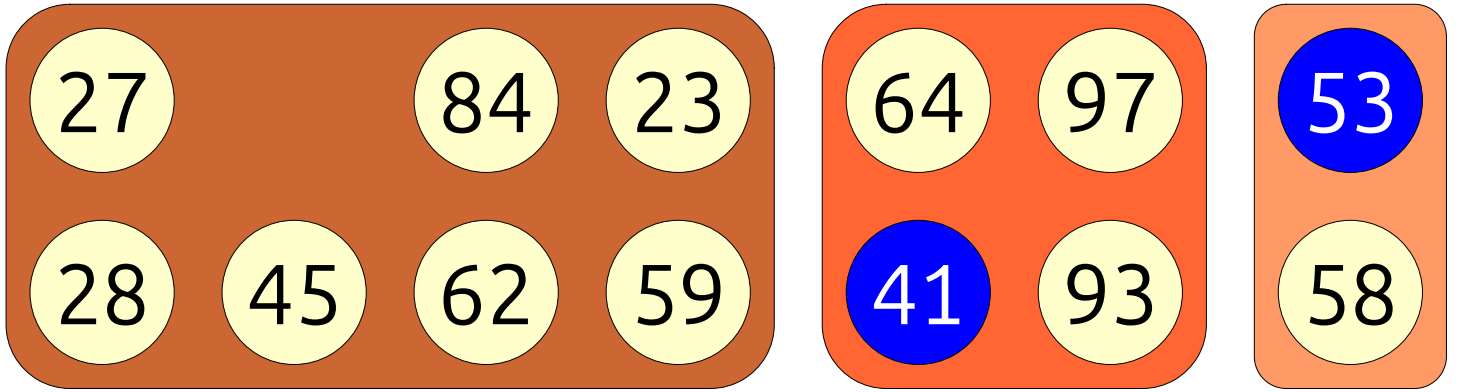
Fracturing Packets

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



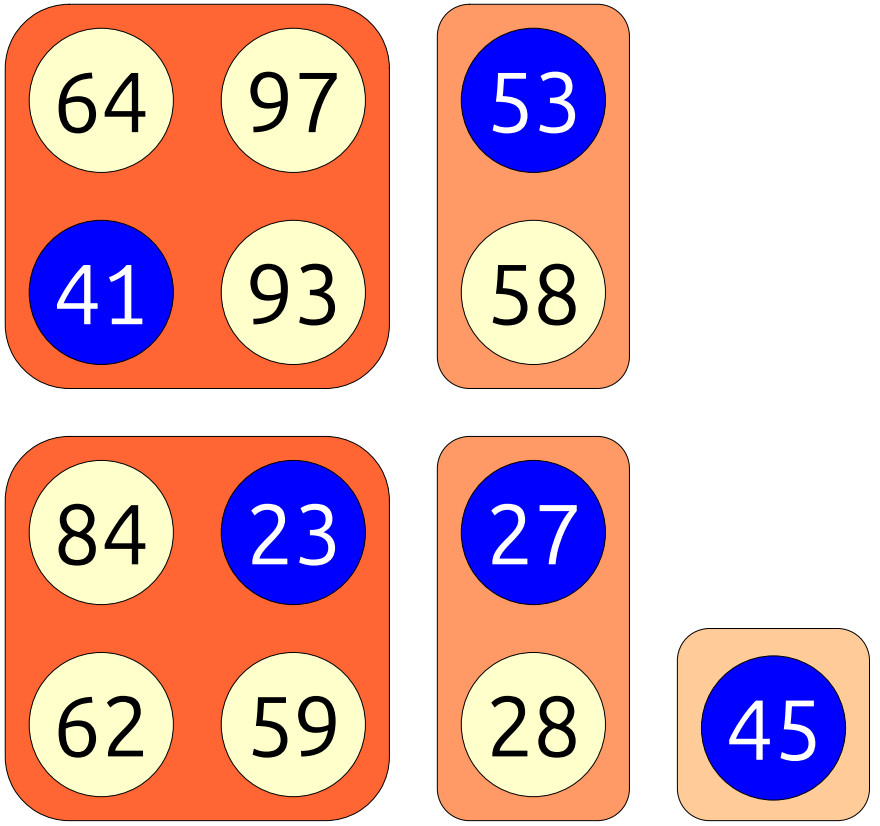
Fracturing Packets

- 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.
- Runtime is O(log *n*) fuses in ***meld***, plus fracture cost.



+

Building a Priority Queue

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

Propose a solution!

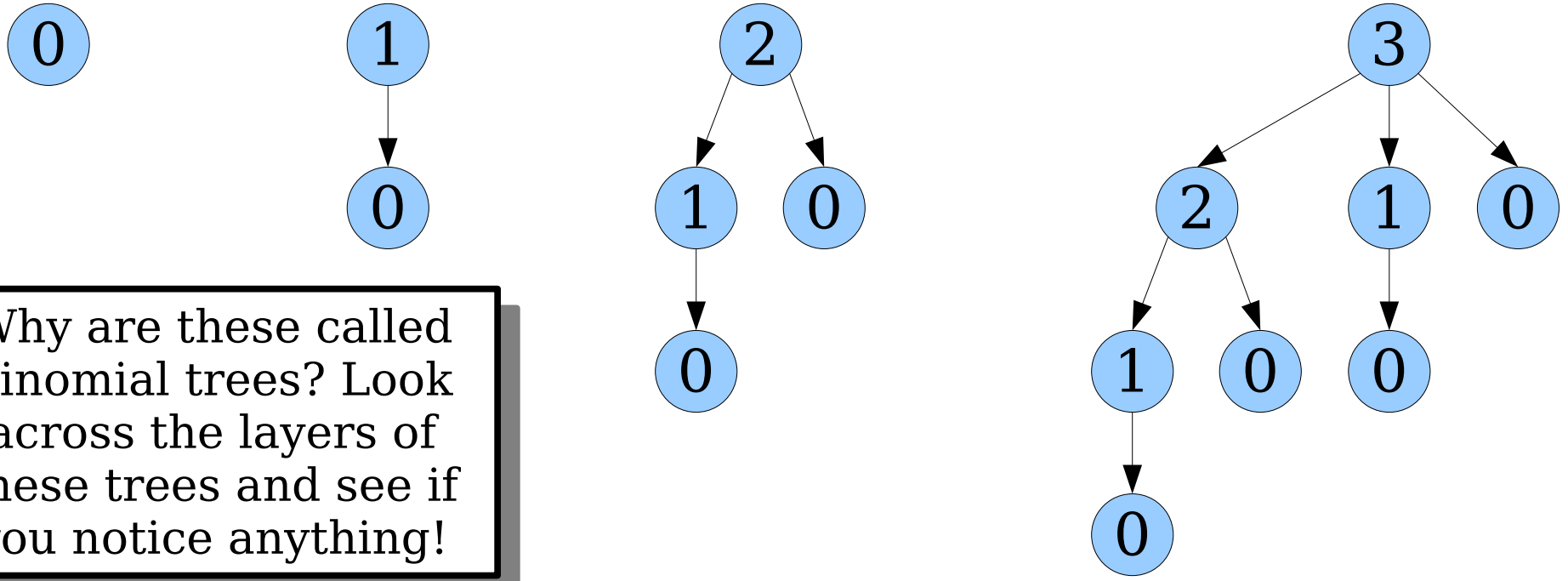
https://cs166.stanford.edu/pollev

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, …, k – 1.

- Here are the first few binomial trees:



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

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 + log m) trees. Cost: O(log n + log m). Here, assume one binomial heap has n nodes, the other m.
  - pq.***enqueue***(v, k): Meld pq and a singleton heap of (v, k).
    - Total time: O(log n).
  - pq.***find-min***(): Find the minimum of all tree roots.
    - Total time: O(log n).
  - pq.***extract-min***(): Find the min, delete the tree root, then meld together the queue and the exposed children.
    - Total time: O(log n).

Where We Stand

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

Binomial Heap

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

Time-Out for Announcements!

Problem Set Four

- Problem Set Three was due today at 1:00PM.
  - Need more time? Use a late day to extend the deadline by 24 hours, or two late days to extend it by 48 hours.
- Problem Set Four (***Balanced Trees***) goes out today. It's due on Thursday, May 14th at 1:00PM.
  - Play around with balanced binary search trees and data structure isometries.
  - Use augmented trees to solve problems much faster than seems feasible at first glance.
  - See how to bound the heights of randomly-built tree structures.
- As usual, ping us if you have any questions!

Back to CS166!

Where We Stand

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

Binomial Heap

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

Where We Stand

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

Binomial Heap

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

Where We Stand

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

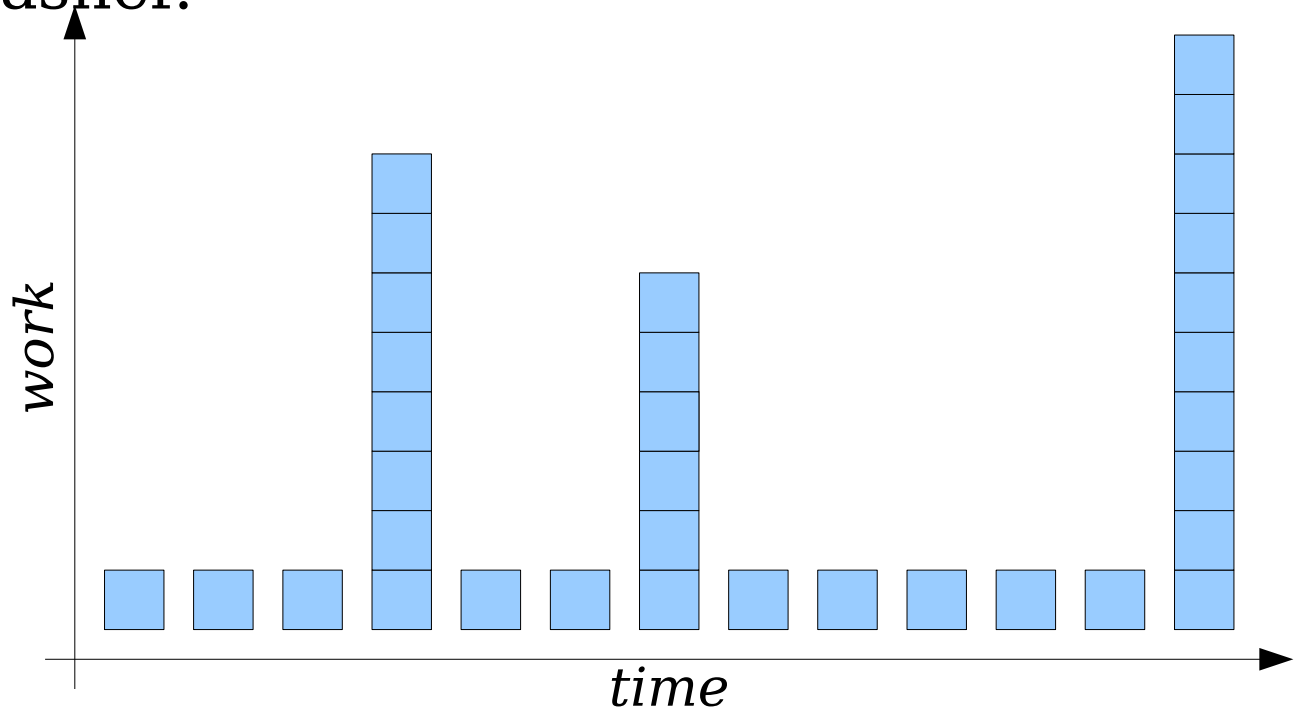
Binomial Heap

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

Thinking With Amortization

Refresher: Amortization

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

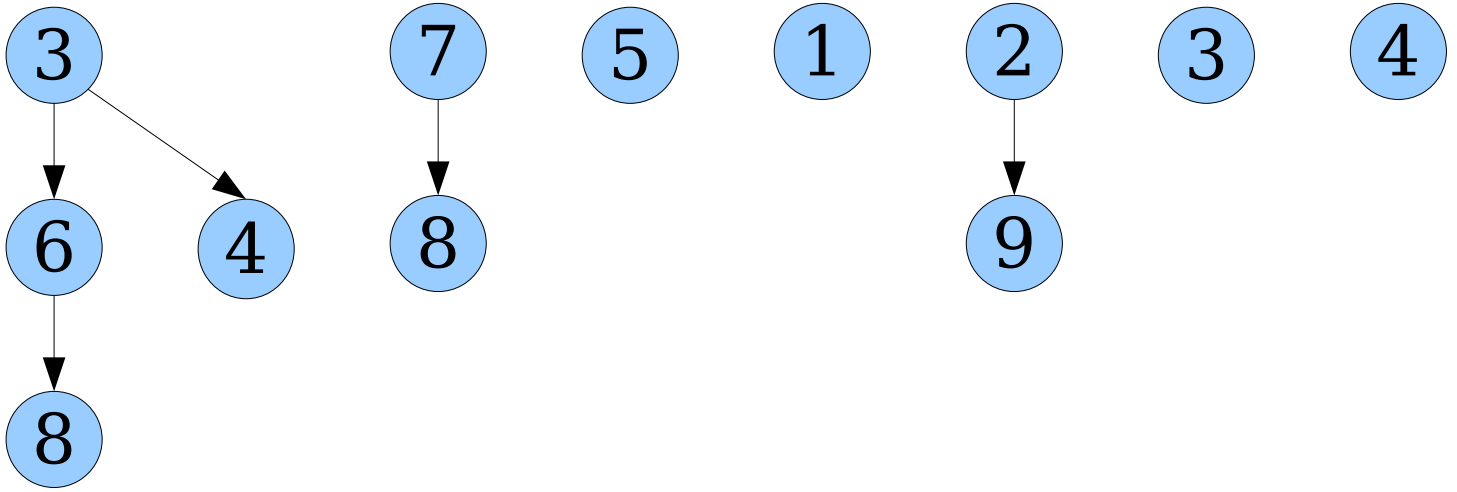


Lazy Melding

- Consider the following lazy ***meld***ing approach:

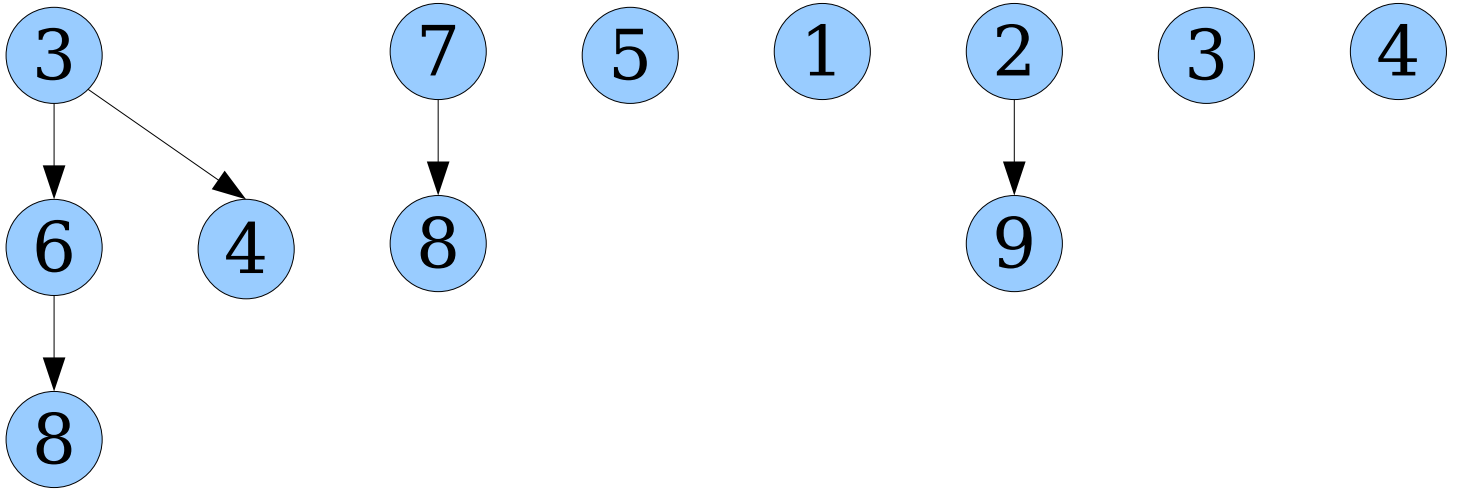
  To meld together two binomial heaps, just combine the two sets of trees together.

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



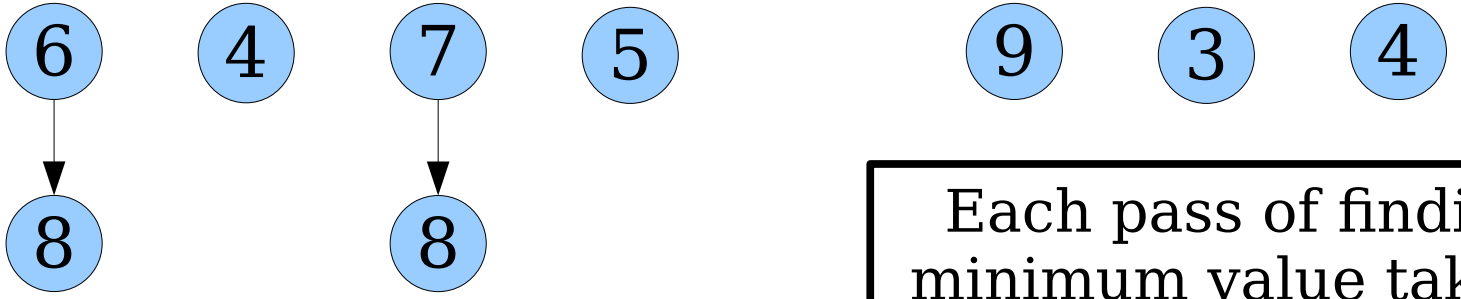
Lazy Melding

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



Lazy Melding

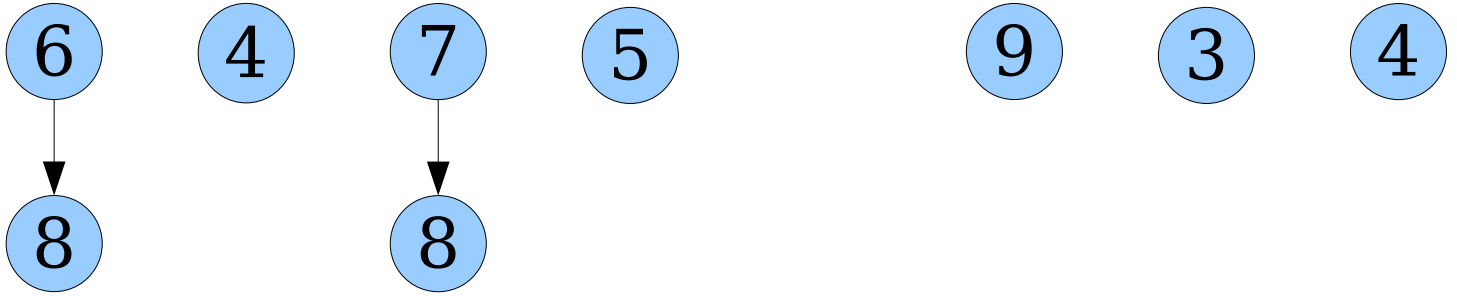
- Imagine that we implement ***extract-min*** the same way as before:
  - Find the packet with the minimum.
  - “Fracture” that packet to expose smaller packets.
  - Meld those packets back in with the master list.
- What happens if we do this with lazy melding?



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

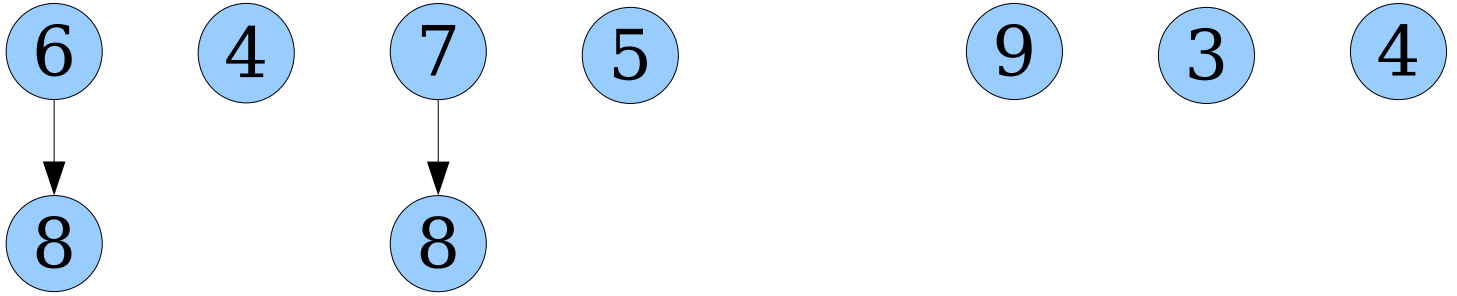
Washing the Dishes

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



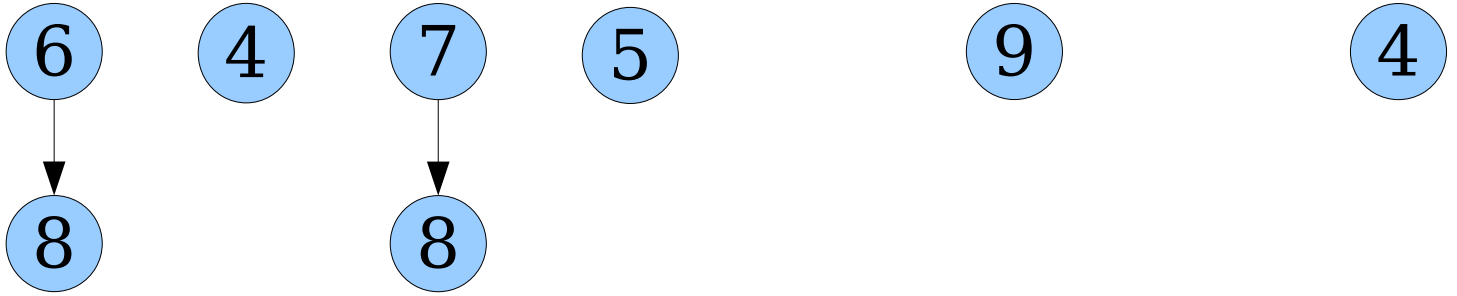
Washing the Dishes

- With our eager ***meld*** (and ***enqueue***) strategy, our priority queue never had more than one tree of each order.
- This kept the number of trees low, which is why each operation was so fast.
- ***Idea:*** After doing an ***extract-min***, do a ***coalesce*** to ensure there's at most one tree of each order.



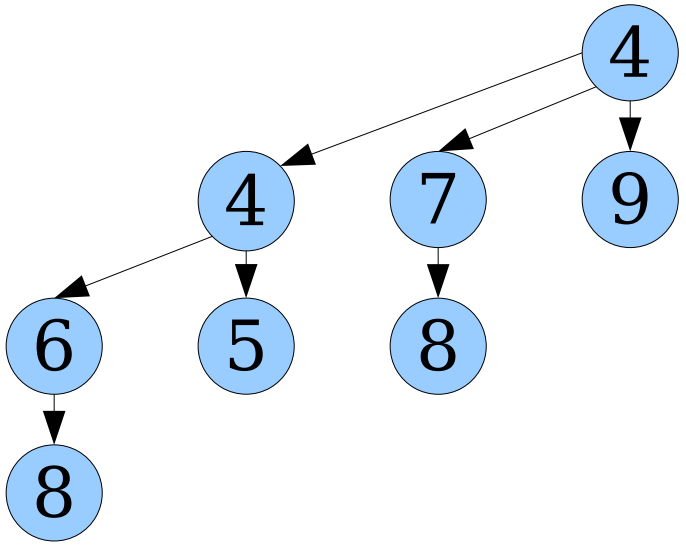
Washing the Dishes

- With our eager ***meld*** (and ***enqueue***) strategy, our priority queue never had more than one tree of each order.
- This kept the number of trees low, which is why each operation was so fast.
- ***Idea:*** After doing an ***extract-min***, do a ***coalesce*** to ensure there's at most one tree of each order.



Washing the Dishes

- With our eager ***meld*** (and ***enqueue***) strategy, our priority queue never had more than one tree of each order.
- This kept the number of trees low, which is why each operation was so fast.
- ***Idea:*** After doing an ***extract-min***, do a ***coalesce*** to ensure there's at most one tree of each order.



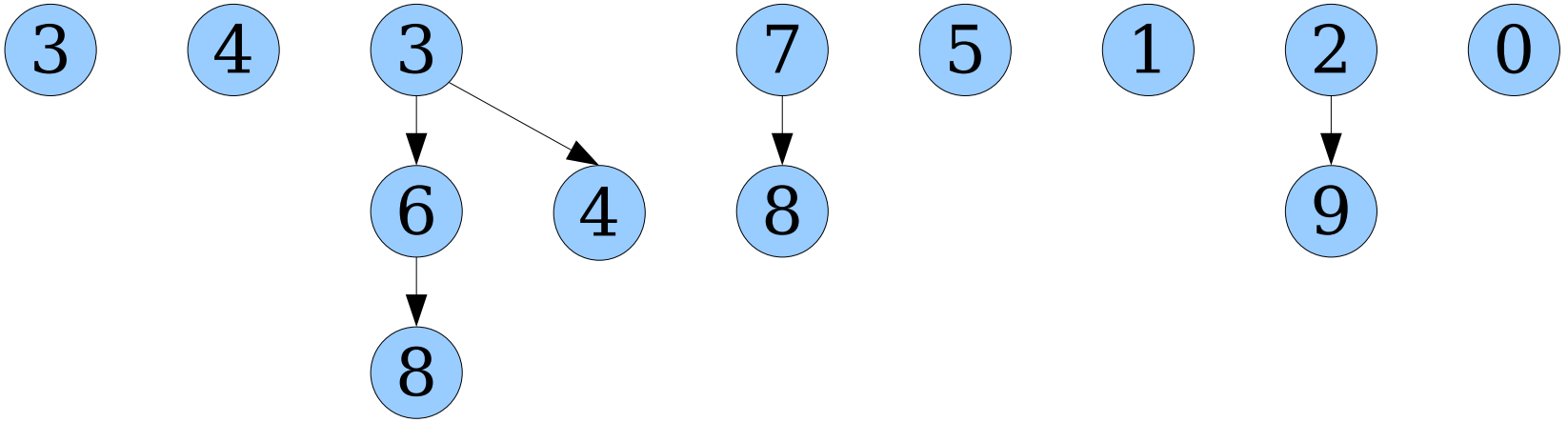
This is what we would have had with the eager implementation.

Where We're Going

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

Coalescing Trees

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

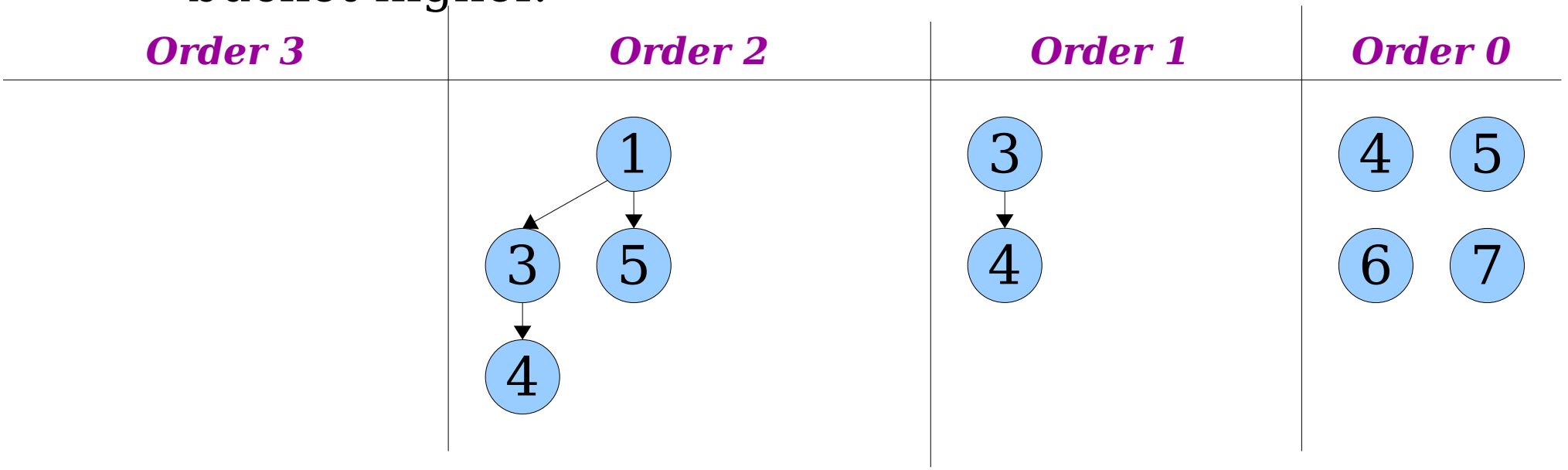


Coalescing Trees

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

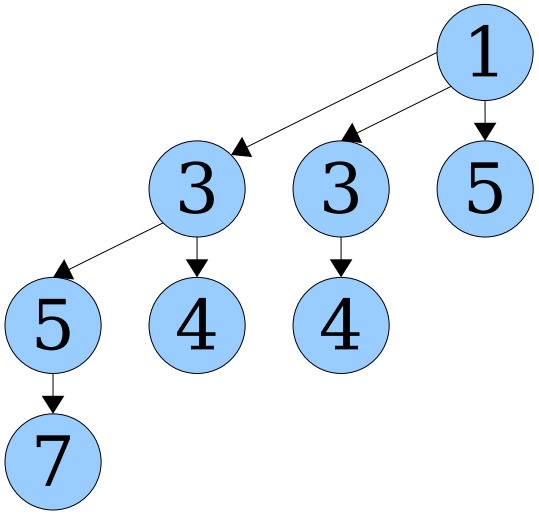
Coalescing Trees

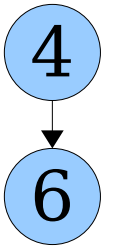
- Here is a fast implementation of ***coalesce***:
  - Distribute the trees into an array of buckets big enough to hold trees of orders 0, 1, 2, …, $\lceil \log_2 (n + 1) \rceil$.
  - Start at bucket 0. While there's two or more trees in the bucket, fuse them and place the result one bucket higher.



Coalescing Trees

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



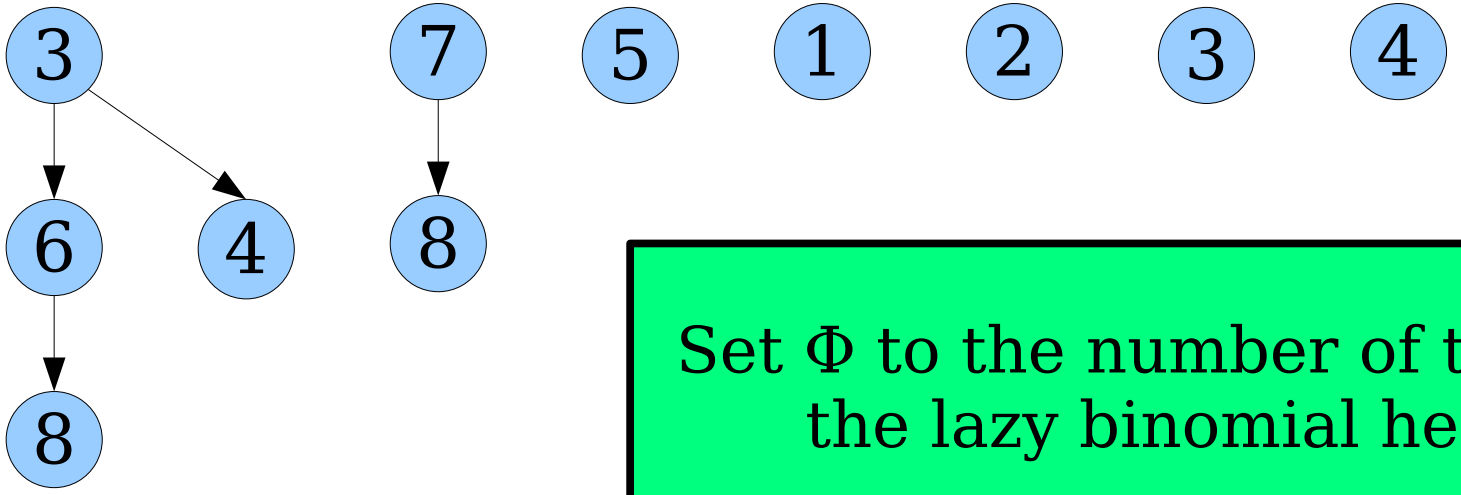


Analyzing Coalesce

- How much time does it take to coalesce a group of t trees?
  - Time to create the array of buckets: O(log n).
  - Time to distribute trees into buckets: O(t).
  - Time to fuse trees: O(t + log n)
    - Number of fuses is O(t), since each fuse decreases the number of trees by one. Cost per fuse is O(1).
    - Need to iterate across O(log n) buckets.
- Total work done: **O(t + log n)**.
- In the worst case, this is O(n).
- Can we amortize away that t term?

An Amortized Analysis

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



Set Φ to the number of trees in the lazy binomial heap.

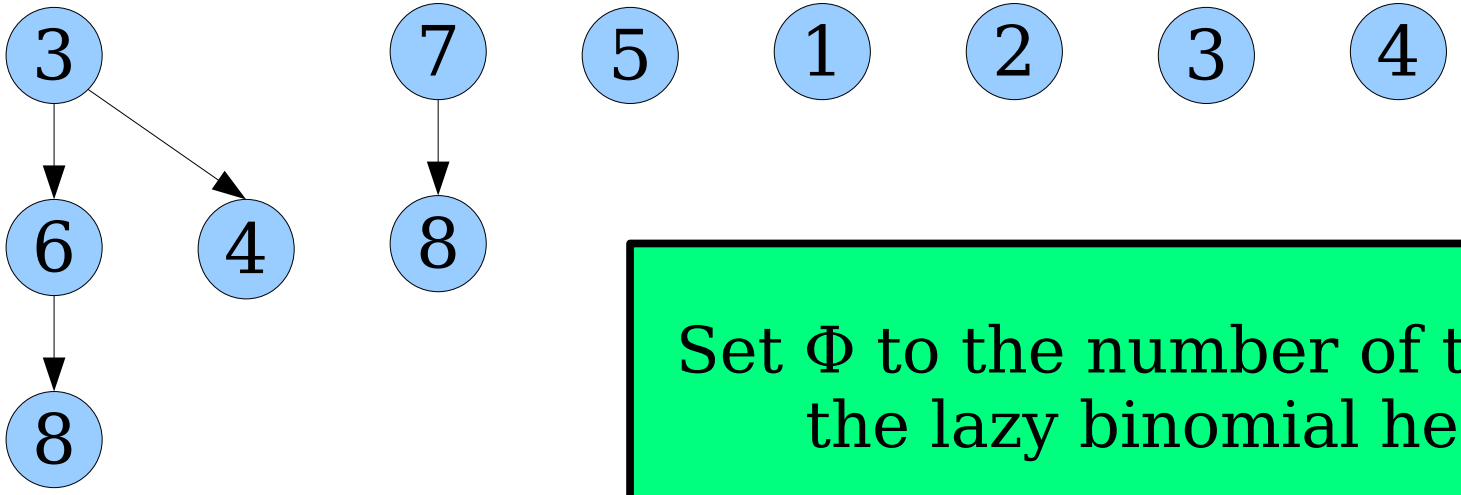
An Amortized Analysis

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

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

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

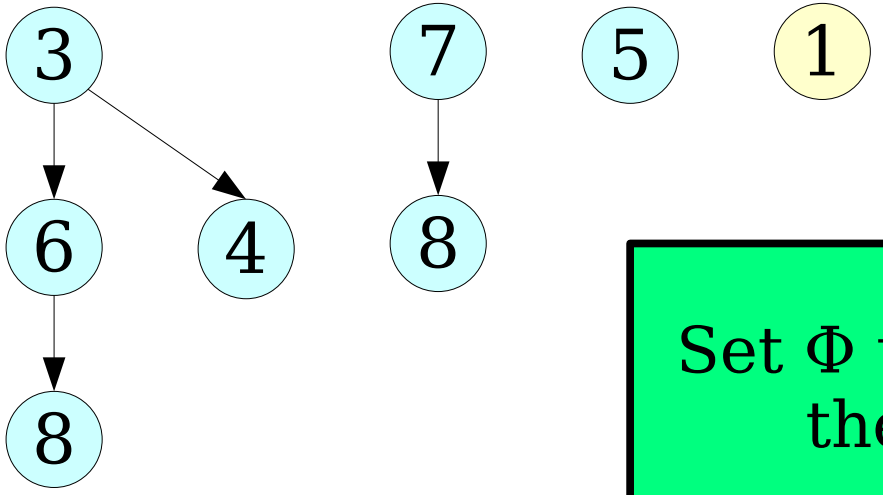
  - Increasing Φ (adding more trees) artificially boosts costs.
  - Decreasing Φ (removing trees) artificially lowers costs.
- Let's work out the amortized costs of each operation on a lazy binomial heap.



Set Φ to the number of trees in the lazy binomial heap.

Analyzing an Insertion

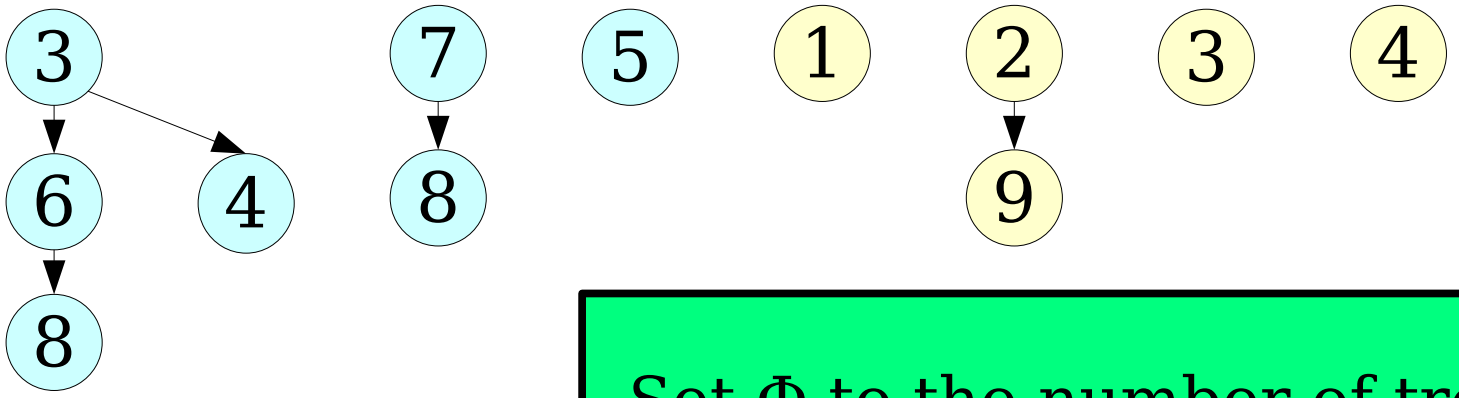
- To ***enqueue*** a key, we add a new binomial tree to the forest.
- Real cost: O(1). $\Delta\Phi$: +1
- Amortized cost: **O(1)**.



Set Φ to the number of trees in the lazy binomial heap.

Analyzing a Meld

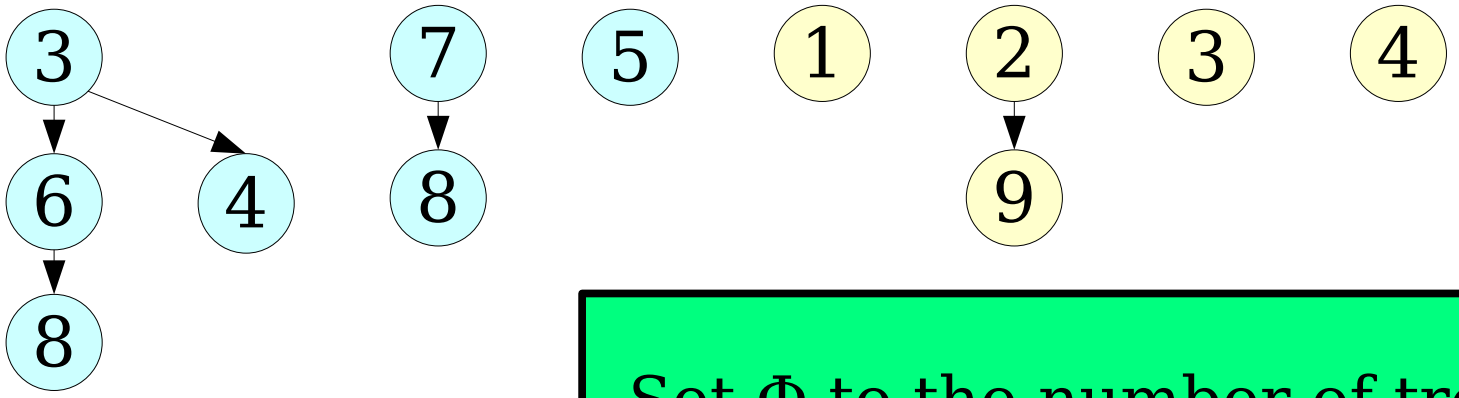
- What is the amortized cost of ***meld***?
- The real cost is O(1).
- What's ΔΦ?
- That's trickier – there are two separate collections of trees here.



Set Φ to the number of trees in the lazy binomial heap.

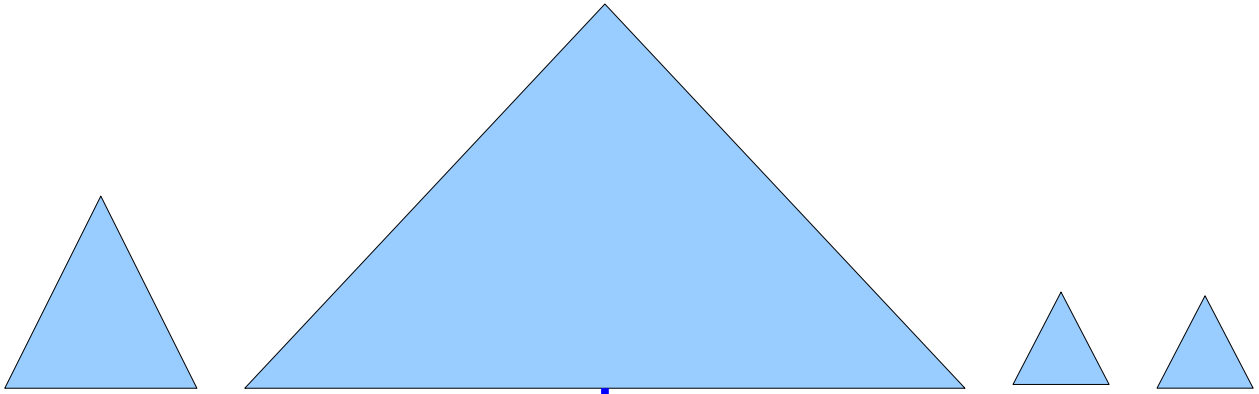
Analyzing a Meld

- What is the amortized cost of ***meld***?
- ***Common trick:*** When working with mergeable data structures, define Φ globally across all instances of the data structure.
- Now $\Delta\Phi = 0$ and the amortized cost is **O(1)**.



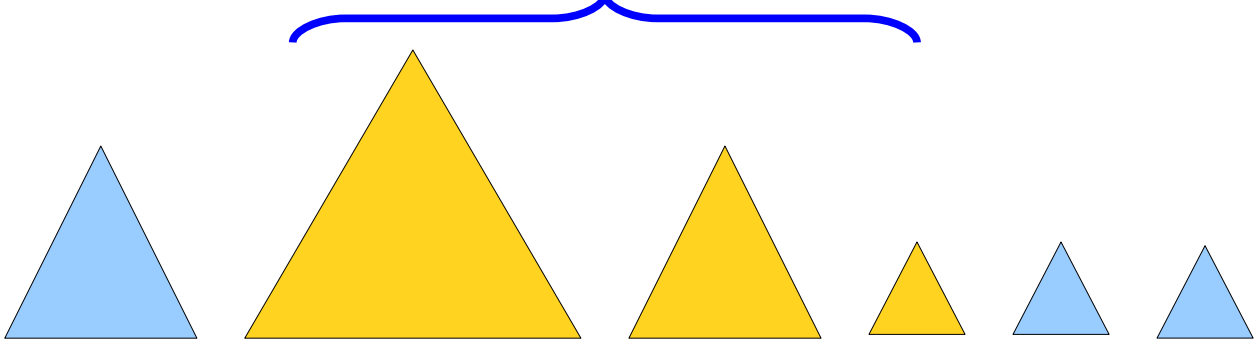
Set Φ to the number of trees in ***all*** lazy binomial heaps.

Analyzing ***extract-min***



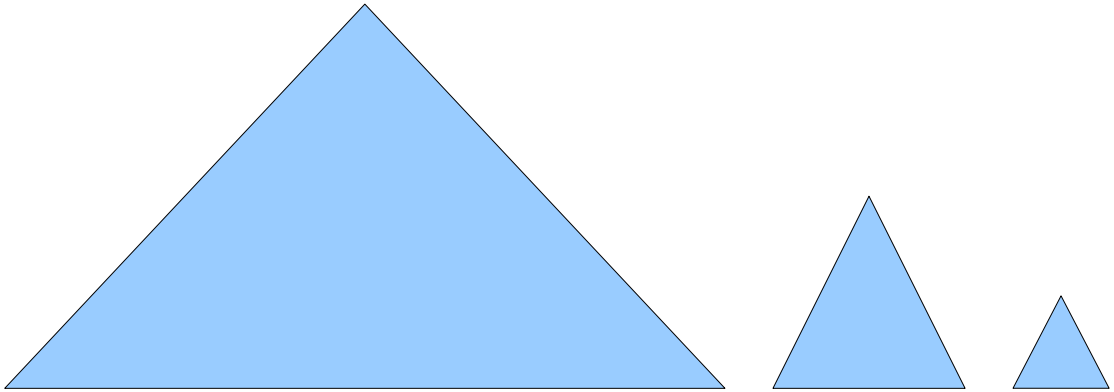
Find tree with minimum key.

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



Remove min. Add children to list of trees.

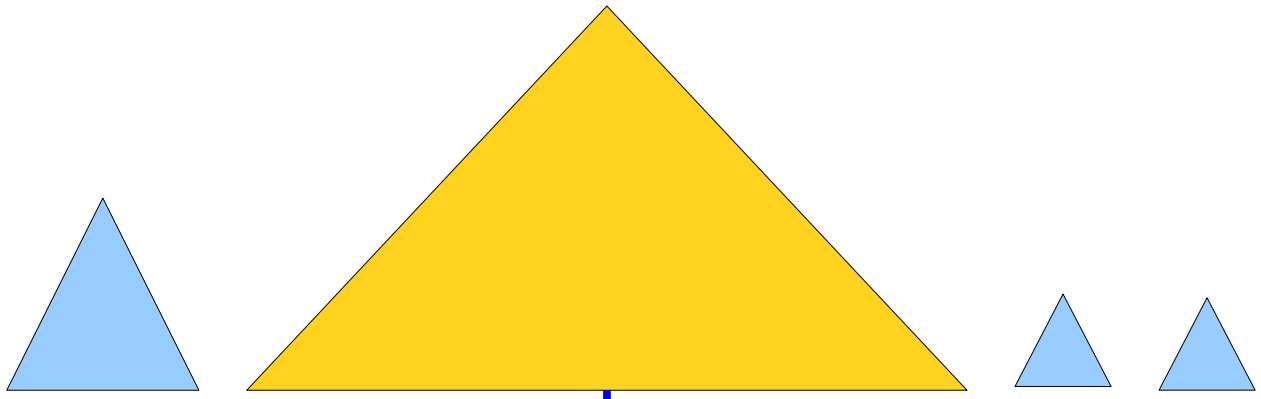
Work: O(log n)



Run the coalesce algorithm.

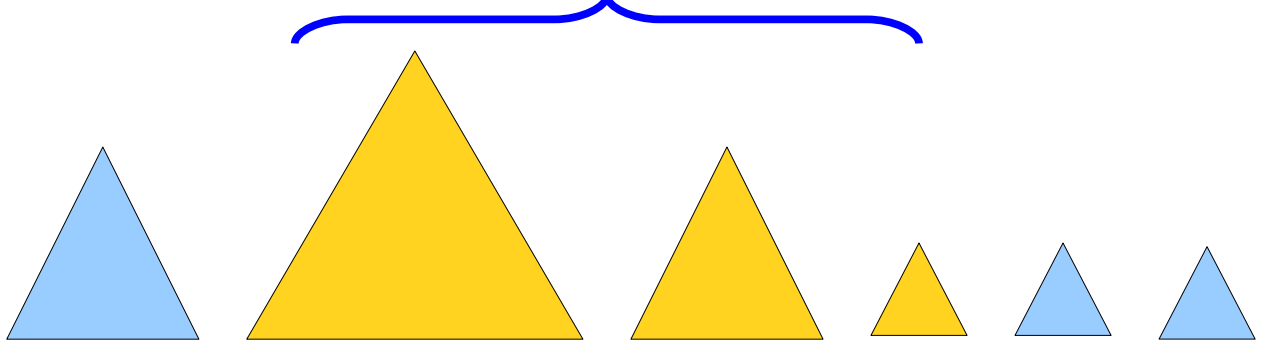
Work: O(t + log n)
Φ = O(log n)

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



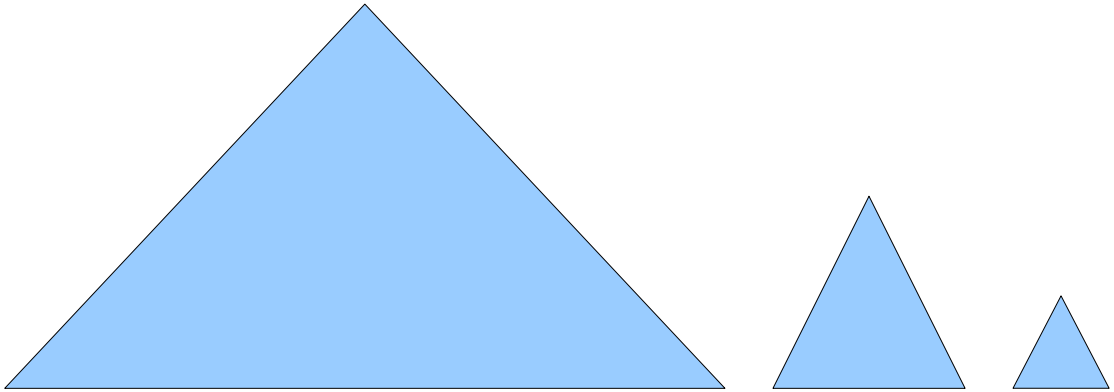
Find tree with minimum key.

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



Remove min. Add children to list of trees.

Work: O(log n)



Run the coalesce algorithm.

Work: O(t + log n)
Φ = O(log n)

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

The Final Scorecard

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

Lazy Binomial Heap

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

* *amortized*

Major Ideas from Today

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

Next Time

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