Fibonacci Heaps

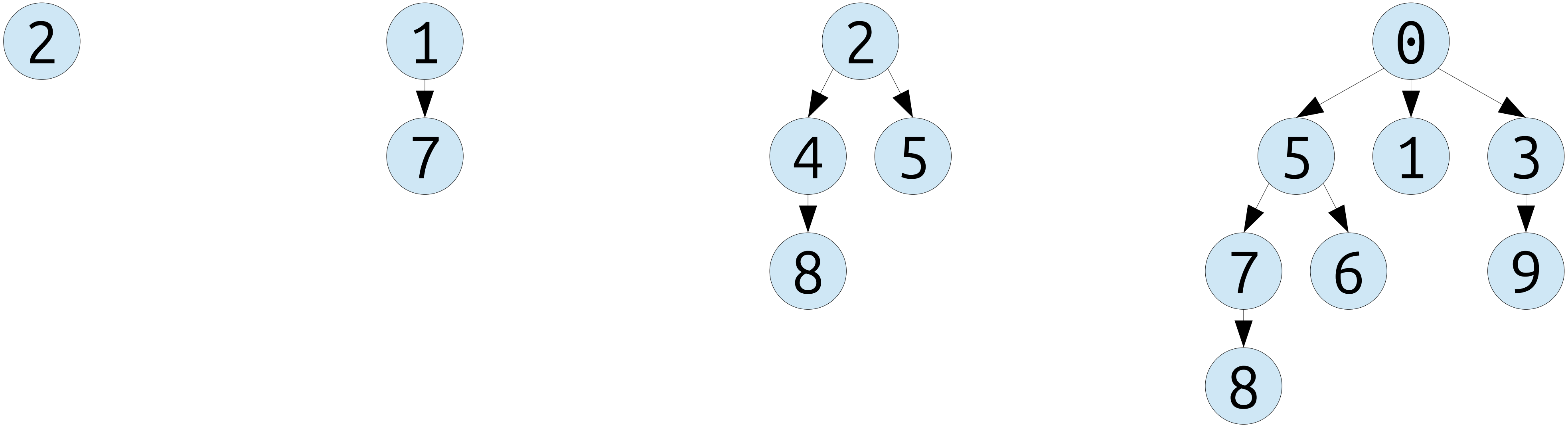
Outline for Today

- ***Recap from Last Time***
  - Quick refresher on binomial heaps and lazy binomial heaps.
- ***The Need for decrease-key***
  - An important operation in many graph algorithms.
- ***Fibonacci Heaps***
  - A data structure efficiently supporting ***decrease-key***.
- ***Representational Issues***
  - Some of the challenges in Fibonacci heaps.

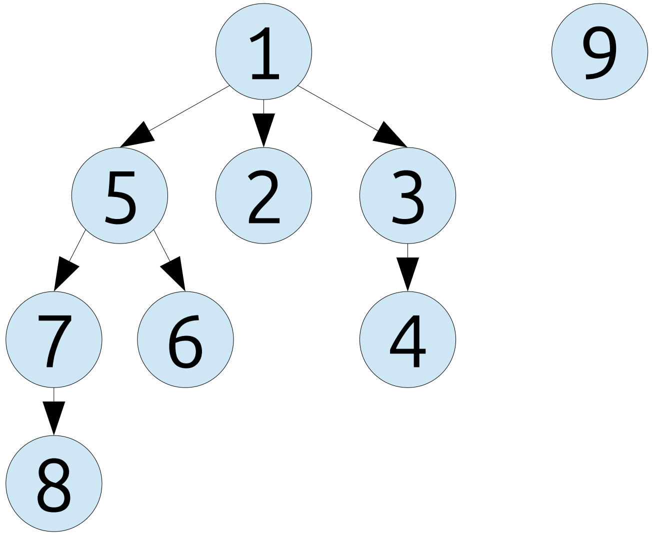
Recap from Last Time

(Lazy) Binomial Heaps

- Last time, we covered the ***binomial heap*** and a variant called the ***lazy binomial heap***.
- These are priority queue structures designed to support efficient ***meld***ing.
- Elements are stored in a collection of ***binomial trees***.



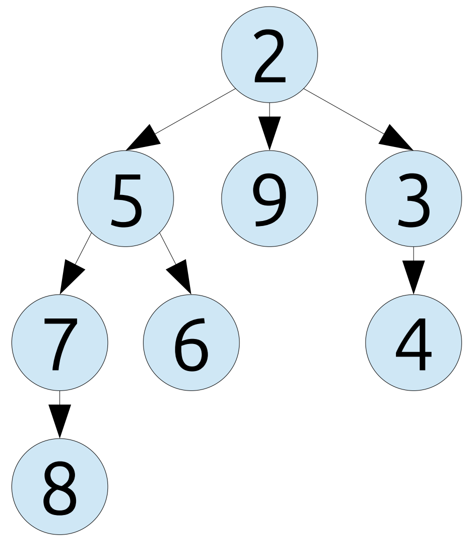
Eager Binomial Heap



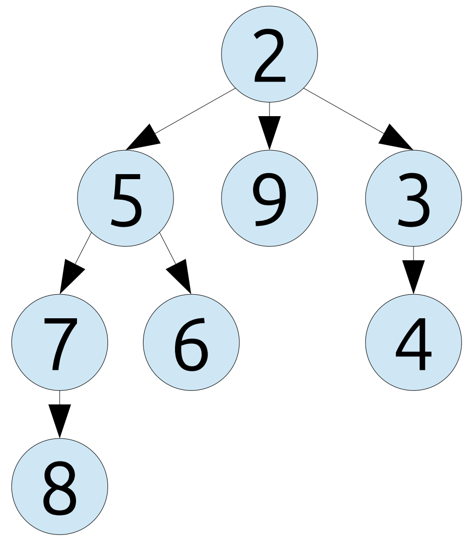
Lazy Binomial Heap

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

Eager Binomial Heap

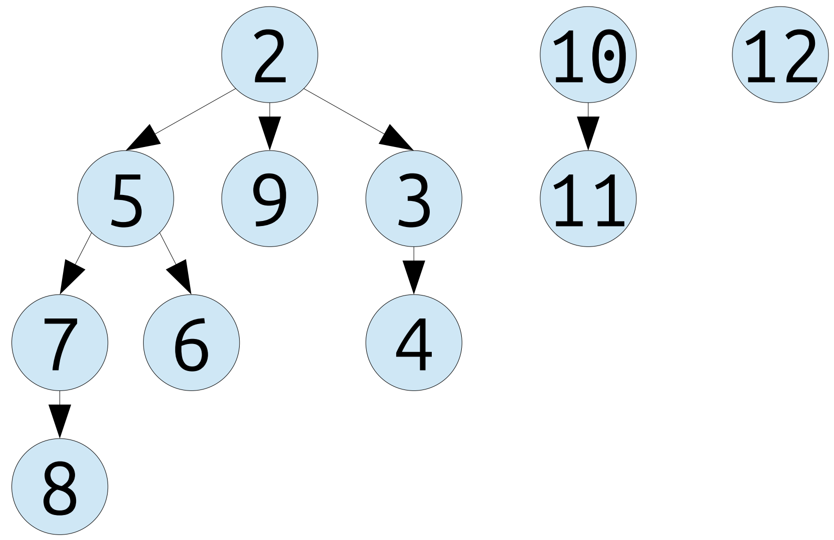


Lazy Binomial Heap

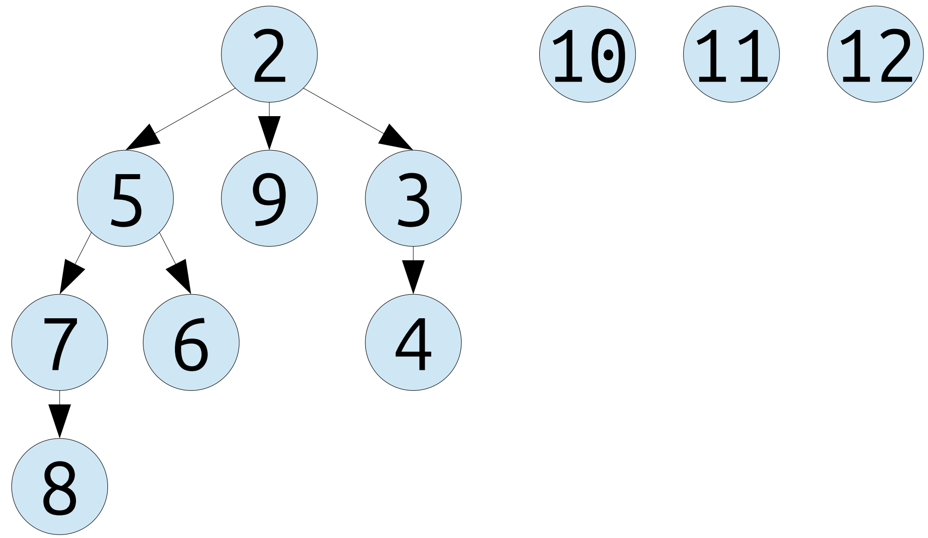


Draw what happens after performing an ***extract-min*** in each binomial heap.

Eager Binomial Heap

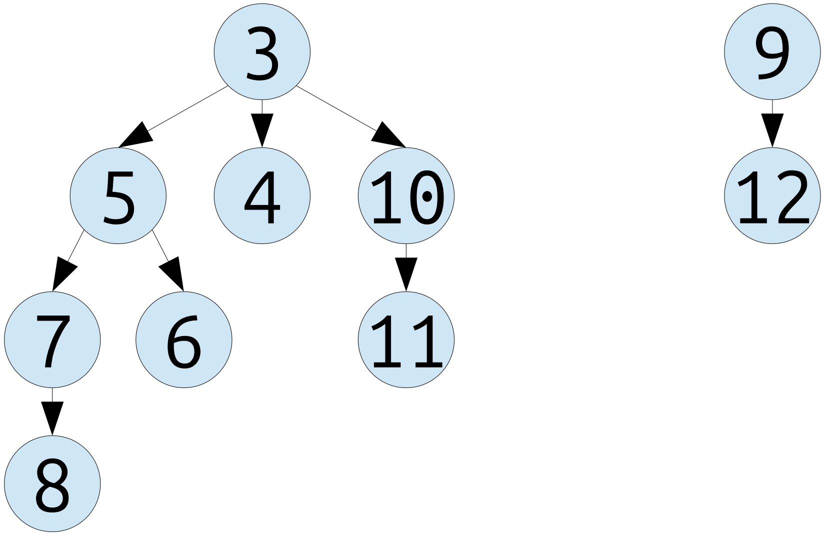


Lazy Binomial Heap

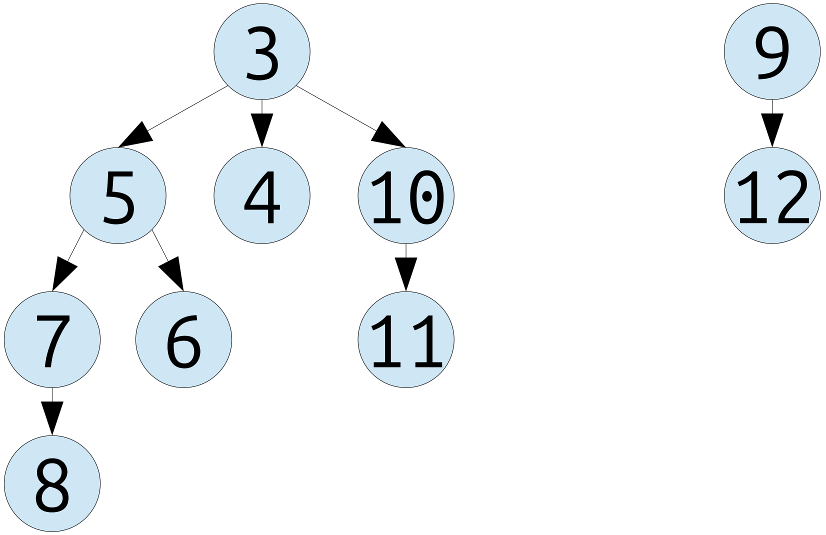


Let's ***enqueue*** 10, 11, and 12 into both heaps.

Eager Binomial Heap



Lazy Binomial Heap



Draw what happens after we do a ***extract-min*** from both heaps.

Operation Costs

- Each individual ***extract-min*** in a lazy binomial heap may take a while, but amortizes out to O(log n).
- ***Intuition:*** Each ***extract-min*** does cleanup for the earlier ***enqueue*** operations, leaving the heap with few trees.

Eager Binomial Heap:

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

Lazy Binomial Heap:

- ***enqueue***: O(1)
- ***meld***: O(1)
- ***find-min***: O(1)
- ***extract-min***: O(log n)*

**amortized*

New Stuff!

The Need for ***decrease-key***

The ***decrease-key*** Operation

- Some priority queues support the operation ***decrease-key***(v, k), which works as follows:

  Given a pointer to an element v in the heap, lower its key (priority) to k. It is assumed that k is less than the current priority of v.

- This operation is crucial in efficient implementations of Dijkstra's algorithm and Prim's MST algorithm.

Dijkstra and ***decrease-key***

- Dijkstra's algorithm can be implemented with a priority queue using
  - O(n) total ***enqueue***s,
  - O(n) total ***extract-min***s, and
  - O(m) total ***decrease-key***s.
- Dijkstra's algorithm runtime is

$$O(n \, T_{enq} + n \, T_{ext} + m \, T_{dec})$$

Prim and ***decrease-key***

- Prim's algorithm can be implemented with a priority queue using
  - O(n) total ***enqueue***s,
  - O(n) total ***extract-min***s, and
  - O(m) total ***decrease-key***s.
- Prim's algorithm runtime is

$$\mathbf{O}(\boldsymbol{n}\ \mathbf{T}_{\mathbf{enq}} + \boldsymbol{n}\ \mathbf{T}_{\mathbf{ext}} + \boldsymbol{m}\ \mathbf{T}_{\mathbf{dec}})$$

Standard Approaches

- In a binary heap, ***enqueue***, ***extract-min***, and ***decrease-key*** can be made to work in time O(log n) time each.
- Cost of Dijkstra's / Prim's algorithm:

$$O(n\ T_{enq} + n\ T_{ext} + m\ T_{dec})$$

$$= O(n \log n + n \log n + m \log n)$$

$$= \mathbf{O(m \log n)}$$

Standard Approaches

- In a lazy binomial heap, ***enqueue*** takes amortized time O(1), and ***extract-min*** and ***decrease-key*** take amortized time $O(\log n)$.
- Cost of Dijkstra's / Prim's algorithm:

$$O(n\,T_{enq} + n\,T_{ext} + m\,T_{dec})$$

$$= O(n + n \log n + m \log n)$$

$$= \mathbf{O(m \log n)}$$

Where We're Going

- The ***Fibonacci heap*** has these amortized runtimes:
  - ***enqueue***: O(1)
  - ***extract-min***: O(log n).
  - ***decrease-key***: O(1).
- Cost of Prim's or Dijkstra's algorithm:

$$O(n\,T_{enq} + n\,T_{ext} + m\,T_{dec})$$

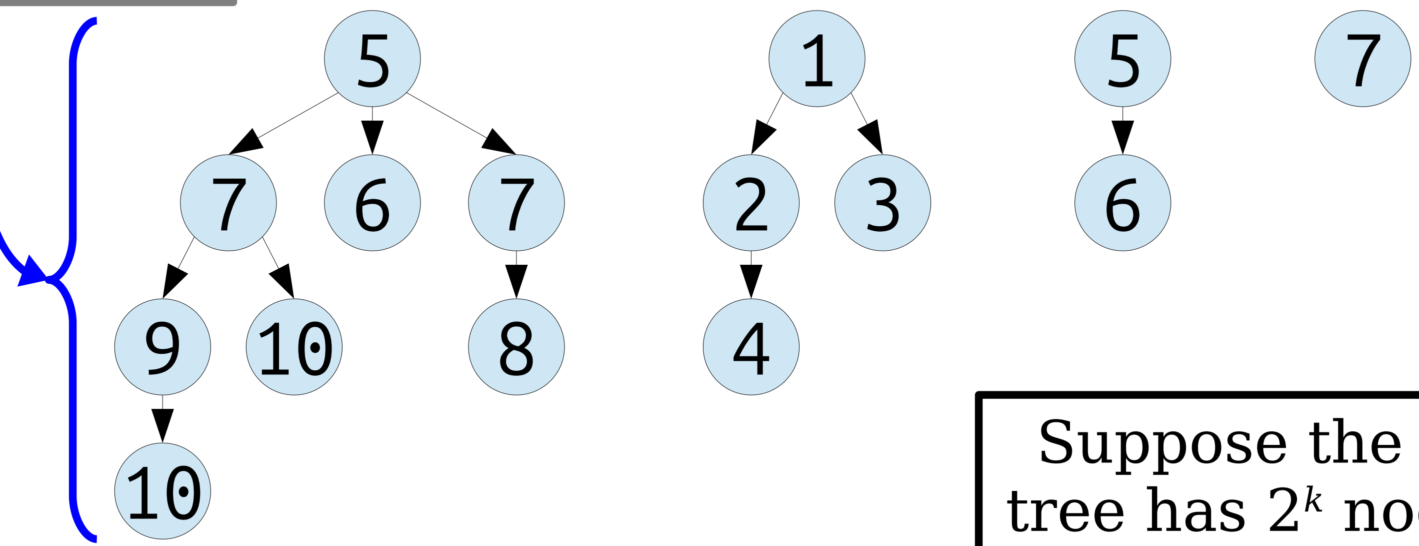
$$= O(n + n \log n + m)$$

$$= \mathbf{O(m + n \log n)}$$

- This is theoretically optimal for Dijkstra's algorithm with comparison-based priority queues if we want distances listed in sorted order. (Though it is possible to go faster if you don't need those last requirements; see ***this recent result***.)

The Challenge of ***decrease-key***

If our lazy binomial heap has *n* nodes, how tall can the tallest tree be?



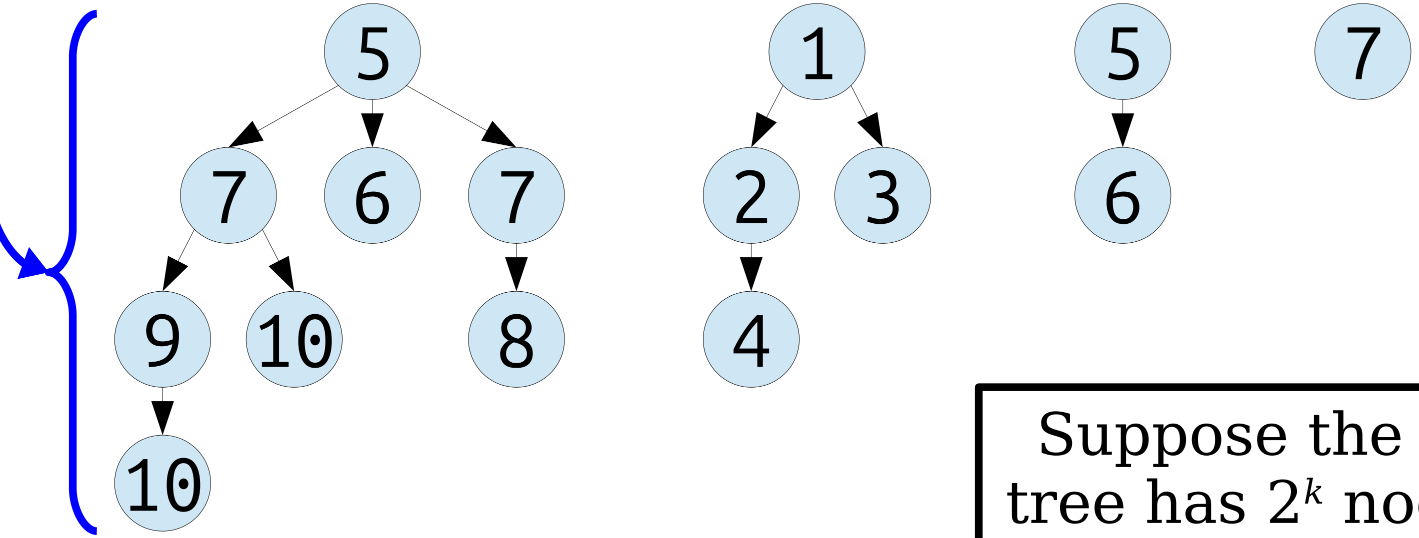
Suppose the biggest tree has 2^k nodes in it.

Then $2^k \leq n$.

So $k = O(\log n)$.

How might we implement ***decrease-key*** in a lazy binomial heap?

If our lazy binomial heap has n nodes, how tall can the tallest tree be?

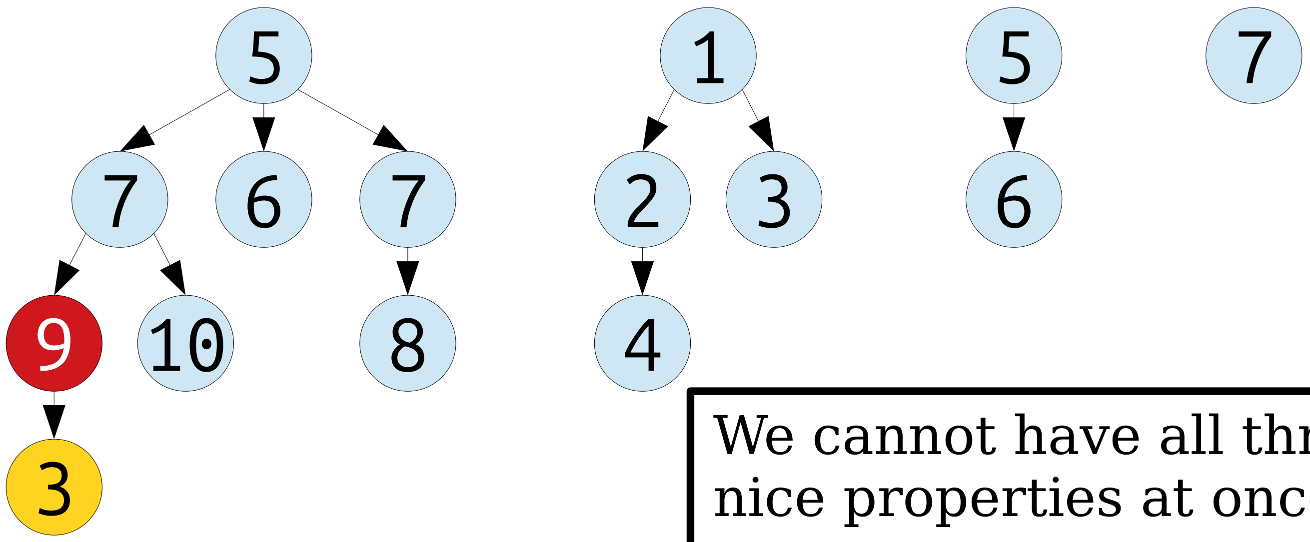


Suppose the biggest tree has 2^k nodes in it.

Then $2^k \leq n$.

So $k = O(\log n)$.

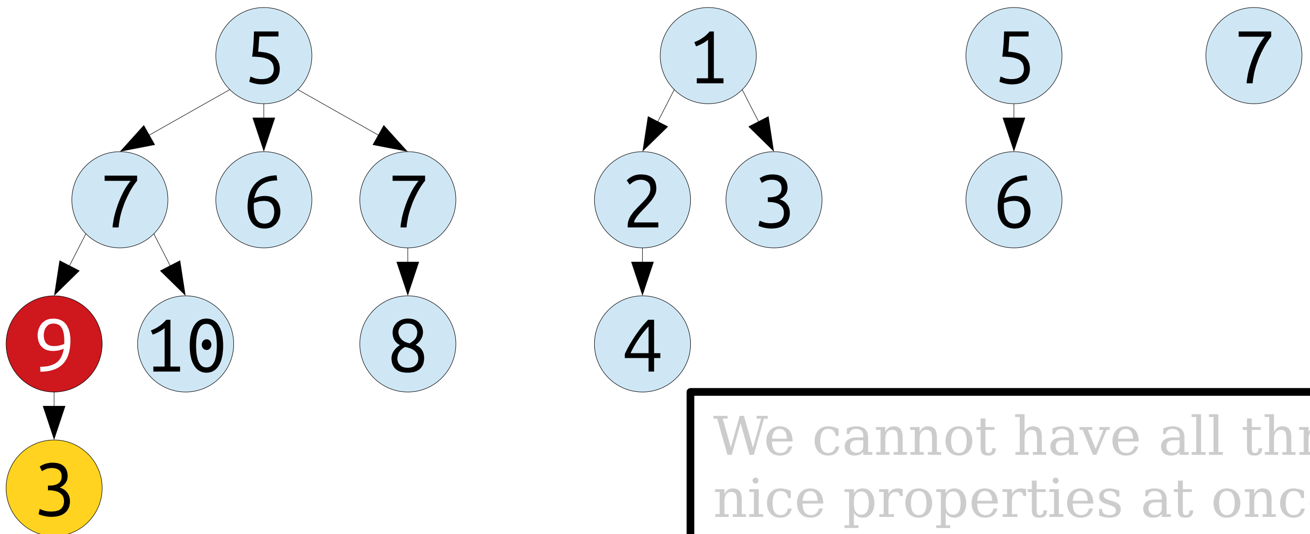
Challenge: Support ***decrease-key*** in (amortized) time O(1).



We cannot have all three of these nice properties at once:

1. ***decrease-key*** takes time O(1).
2. Our trees are heap-ordered.
3. Our trees are binomial trees.

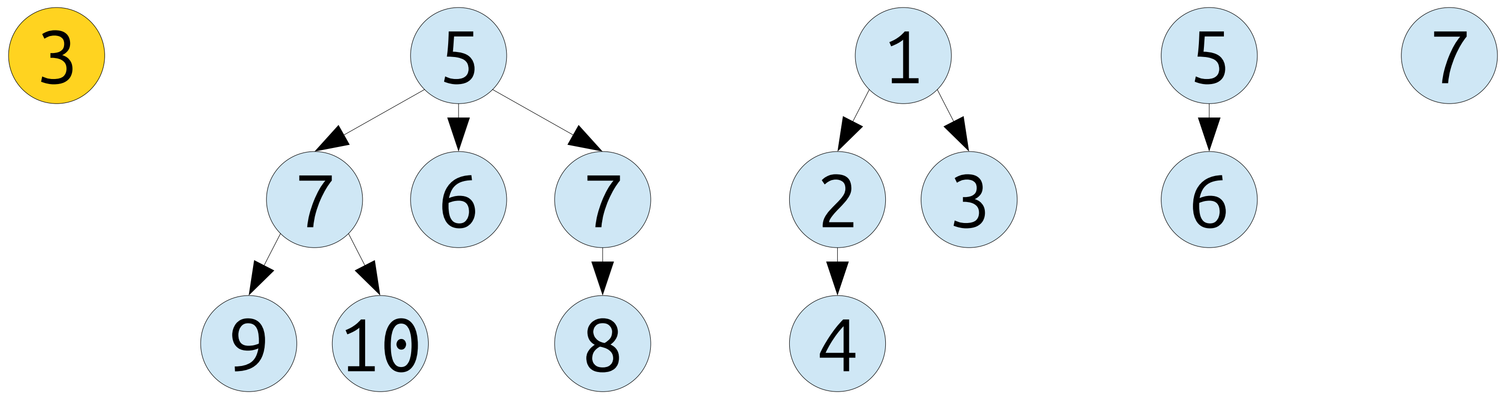
Challenge: Support ***decrease-key*** in (amortized) time O(1).



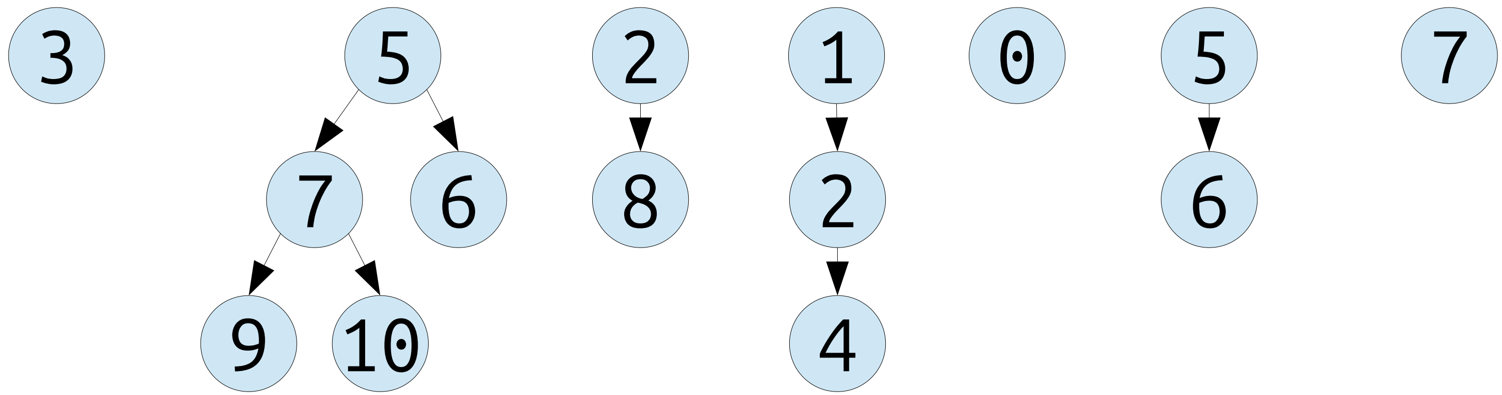
We cannot have all three of these nice properties at once:

1. ***decrease-key*** takes time O(1).
2. Our trees are heap-ordered.
3. Our trees are binomial trees.

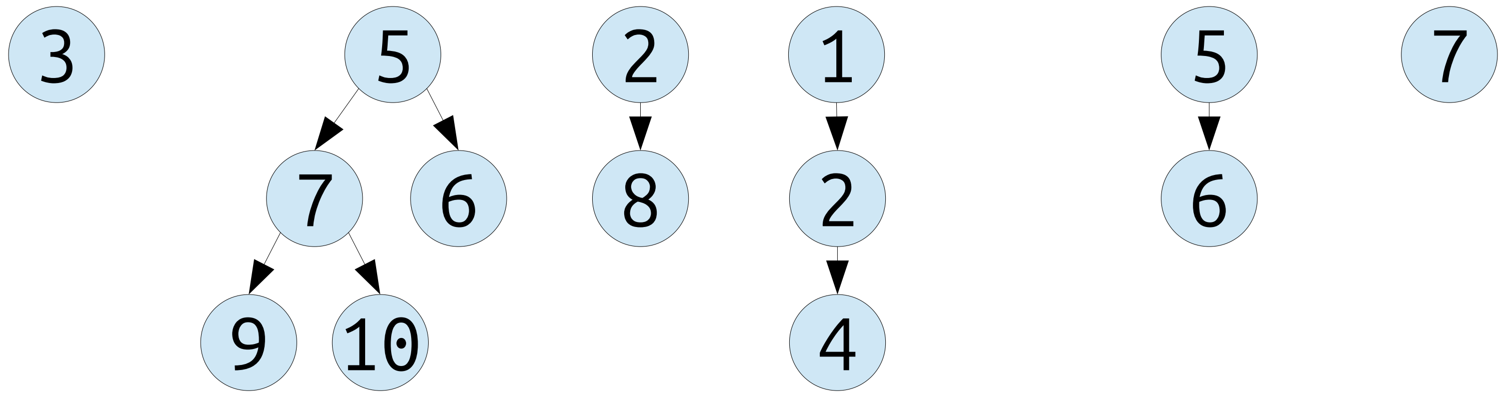
Challenge: Support ***decrease-key*** in (amortized) time O(1).

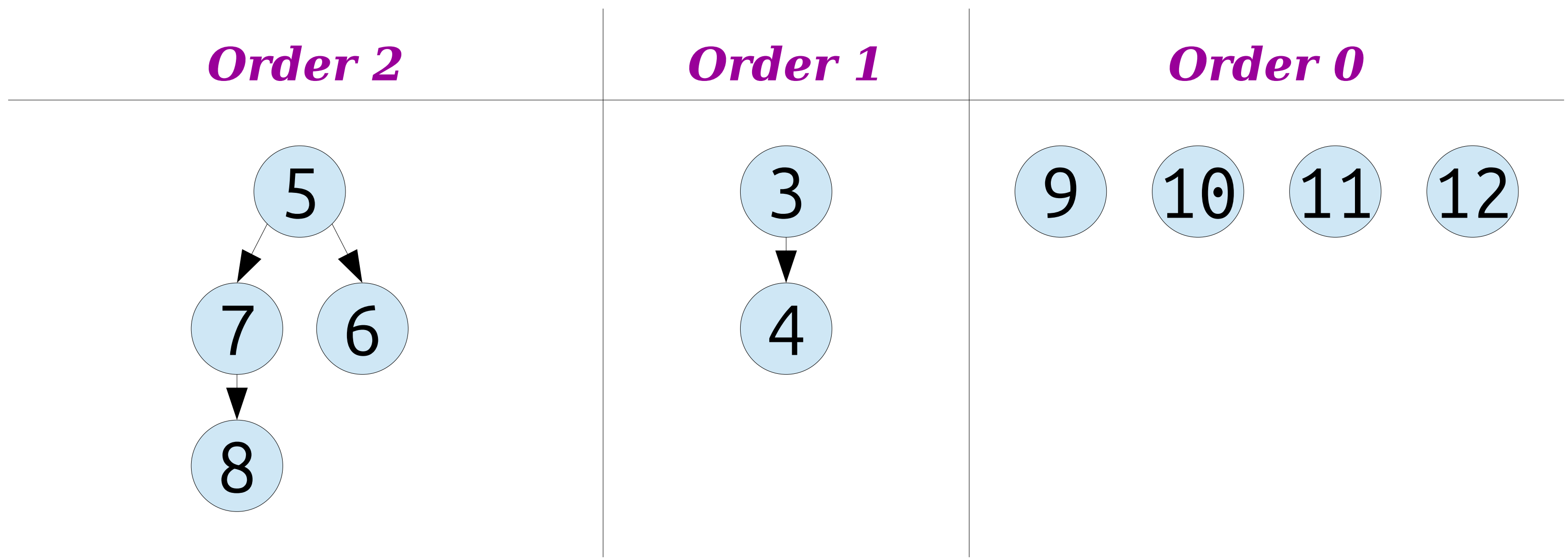


Challenge: Support ***decrease-key*** in (amortized) time O(1).



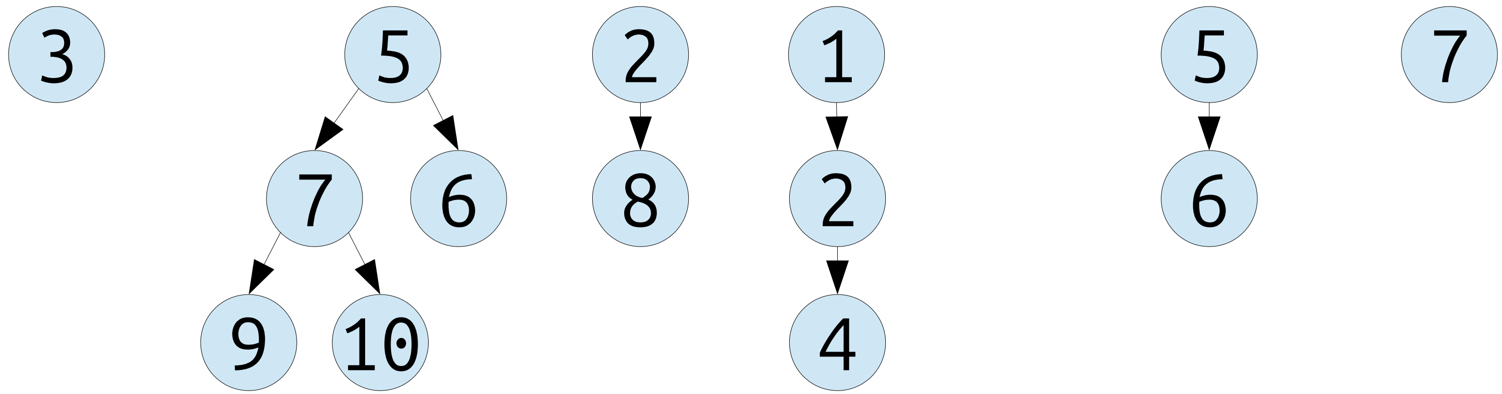
Problem: What do we do in an ***extract-min***?

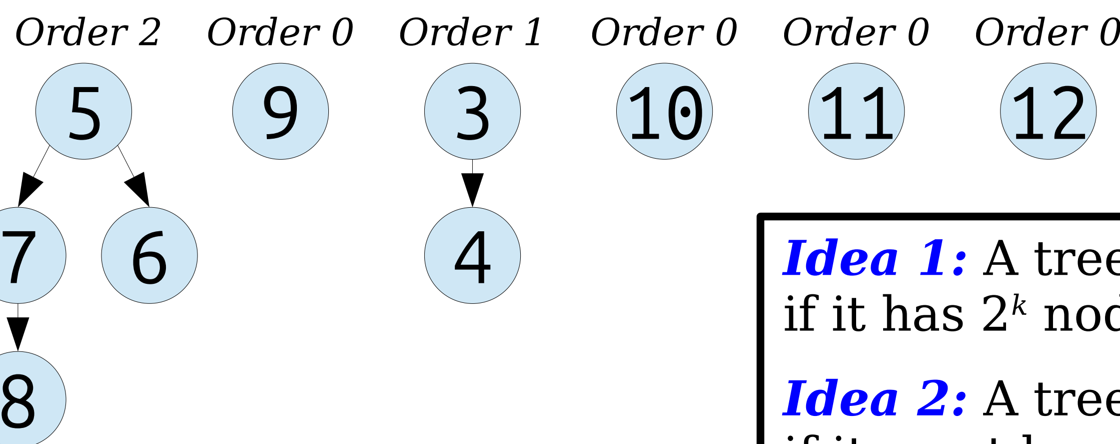




What We Used to Do

Problem: What do we do in an ***extract-min***?



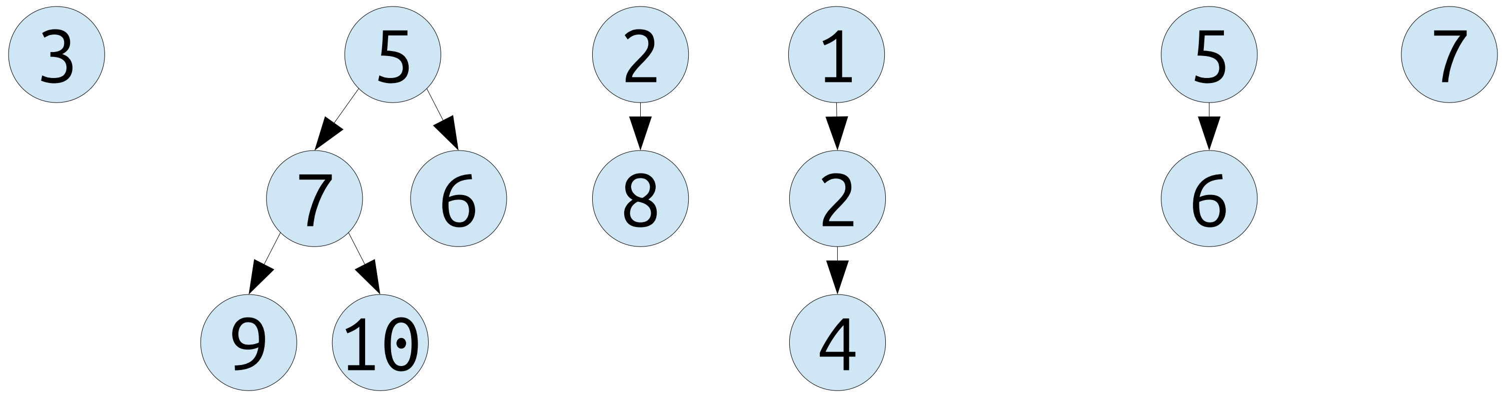


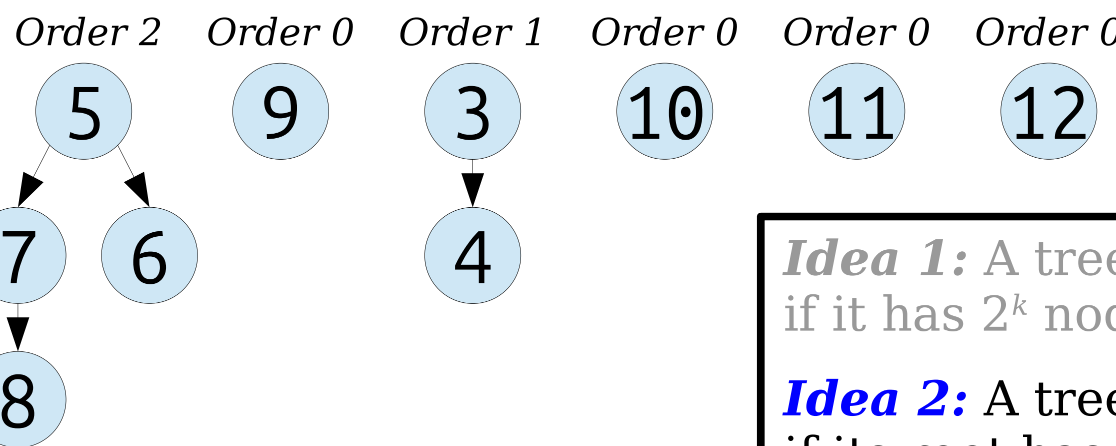
Idea 1: A tree has order k if it has 2^k nodes.

Idea 2: A tree has order k if its root has k children.

What We Used to Do

Problem: What do we do in an ***extract-min***?



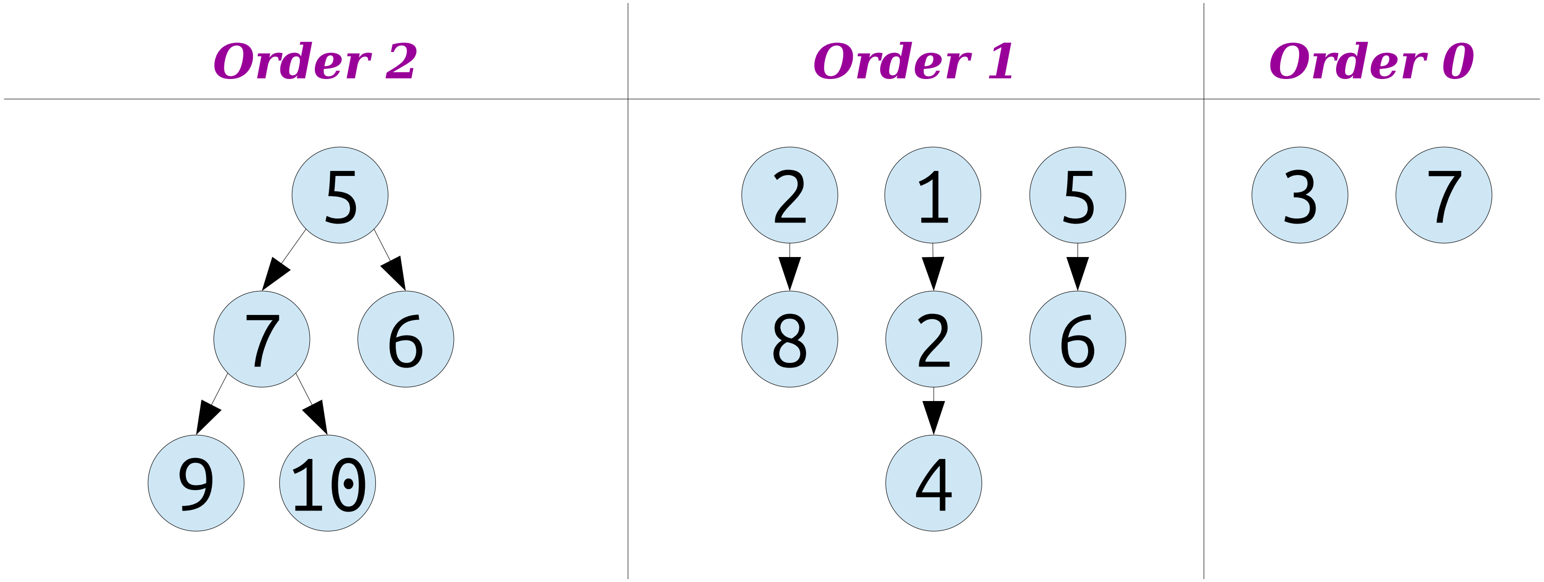


~~***Idea 1:*** A tree has order k if it has 2^k nodes.~~

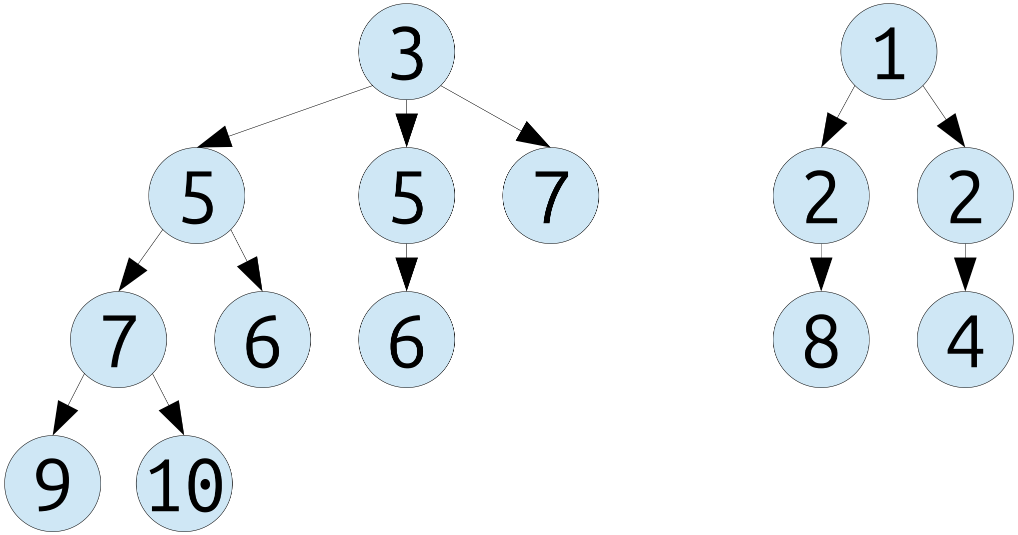
Idea 2: A tree has order k if its root has k children.

What We Used to Do

Problem: What do we do in an ***extract-min***?

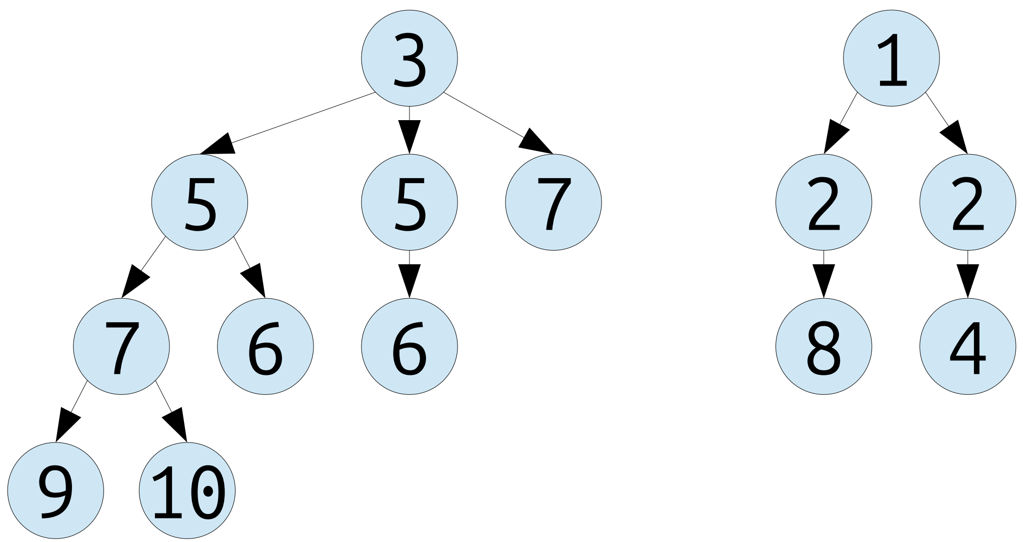


Problem: What do we do in an ***extract-min***?



Problem: What do we do in an ***extract-min***?

Question: How efficient is this?

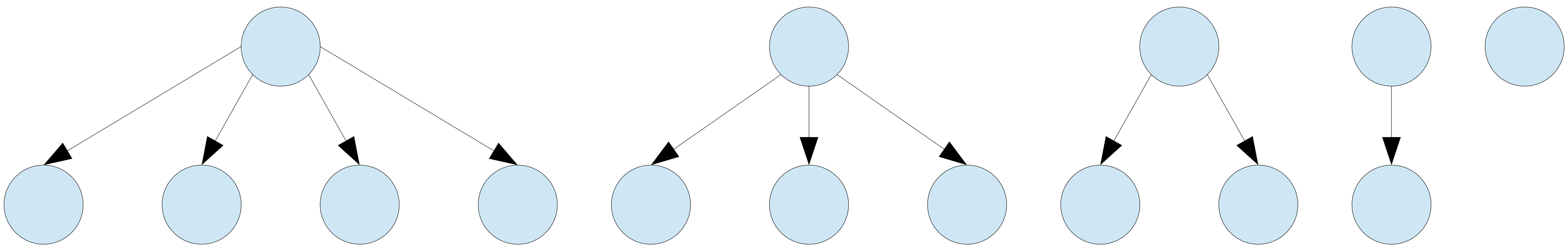


(1) To do a ***decrease-key***, cut the node from its parent.
(2) Do ***extract-min*** as usual, using child count as order.

Intuition: ***extract-min*** is only fast if it compacts nodes into a few trees.

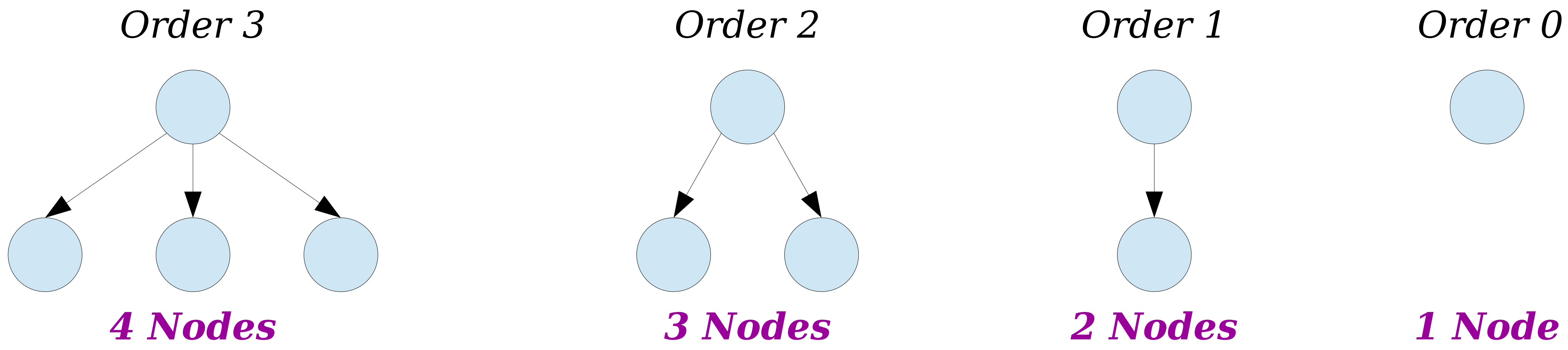
There are $\Theta(n^{1/2})$ trees here.

What happens if we repeatedly ***enqueue*** and ***extract-min*** a small value?

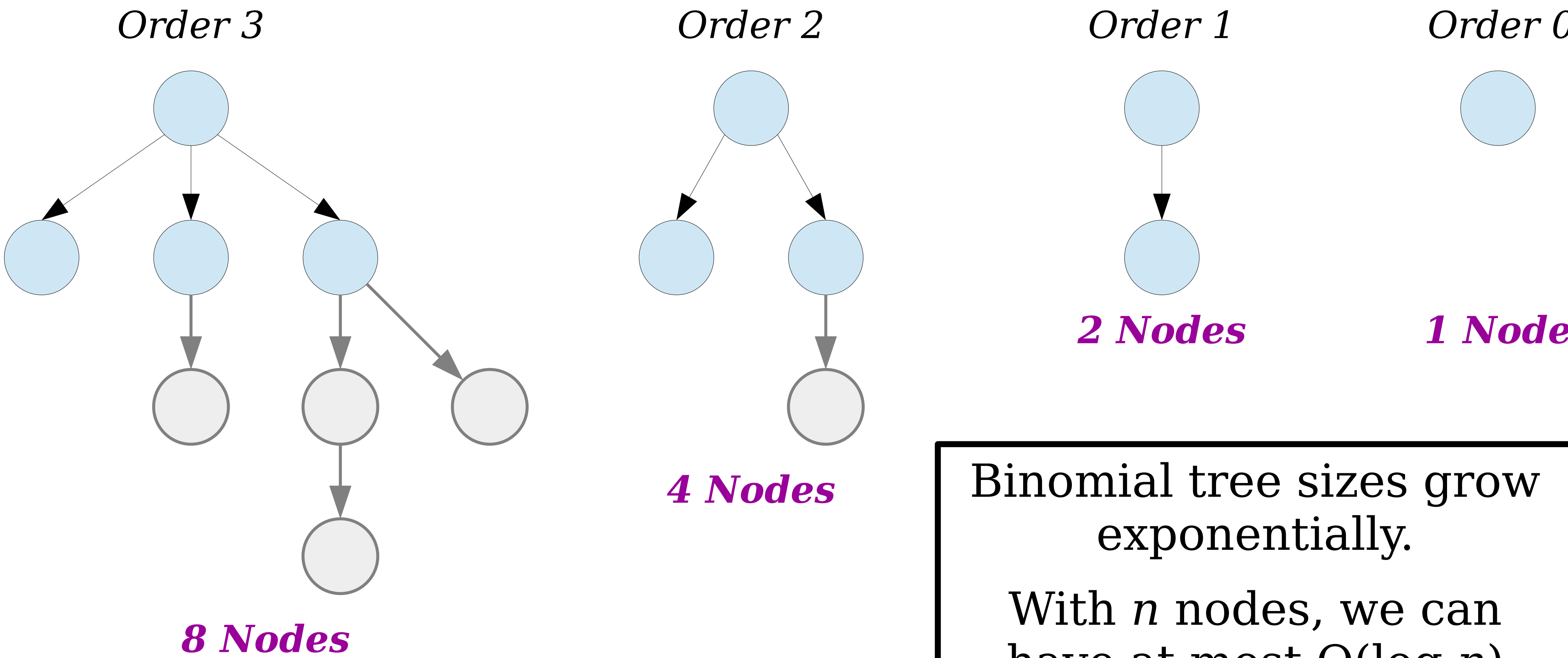


(Do a bunch of work to compact the trees, which doesn't accomplish anything.)

Claim: Because tree shapes aren't constrained, we can force ***extract-min*** to take amortized time $\Omega(n^{1/2})$.



Question: Why didn't this happen before?



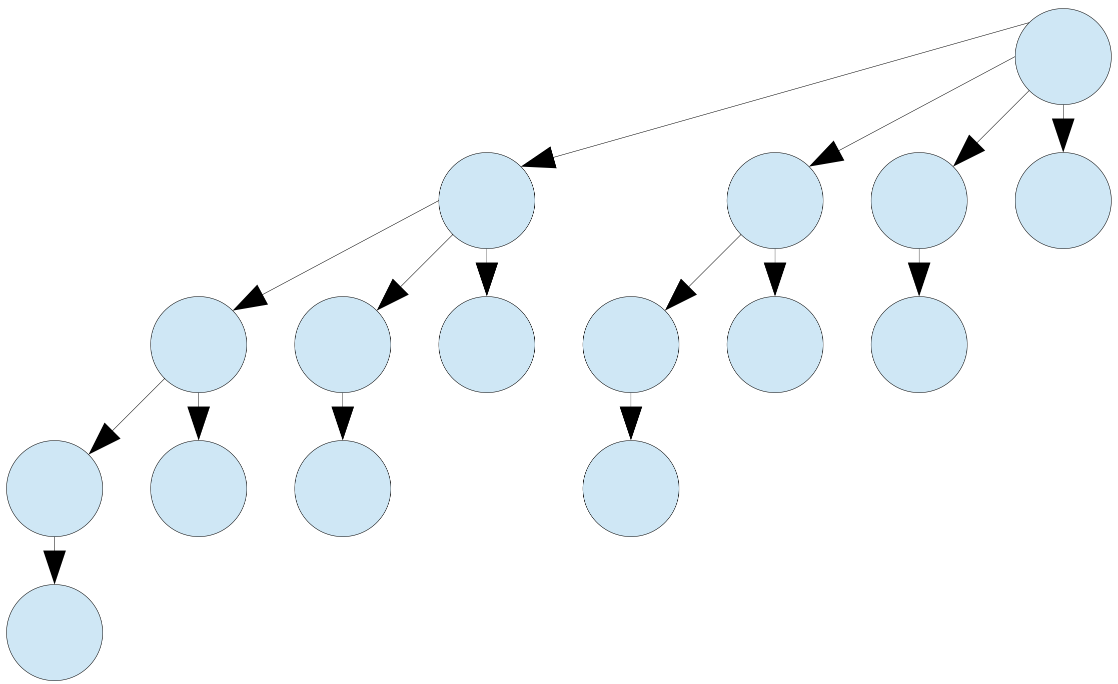
Binomial tree sizes grow exponentially.

With n nodes, we can have at most O(log n) trees of distinct orders.

Question: Why didn't this happen before?

Intuition: Allow trees to get somewhat imbalanced, slowly propagating information to the root.

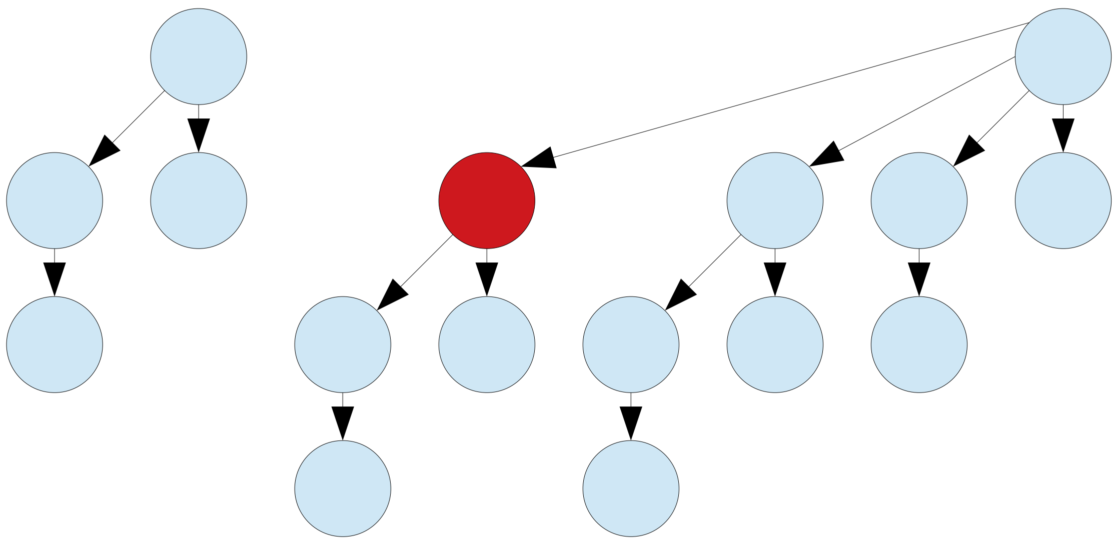
Rule: Nodes can lose at most one child. If a node loses two children, cut it from its parent.



Goal: Make tree sizes grow exponentially with order, but still allow for subtrees to be cut out quickly.

Intuition: Allow trees to get somewhat imbalanced, slowly propagating information to the root.

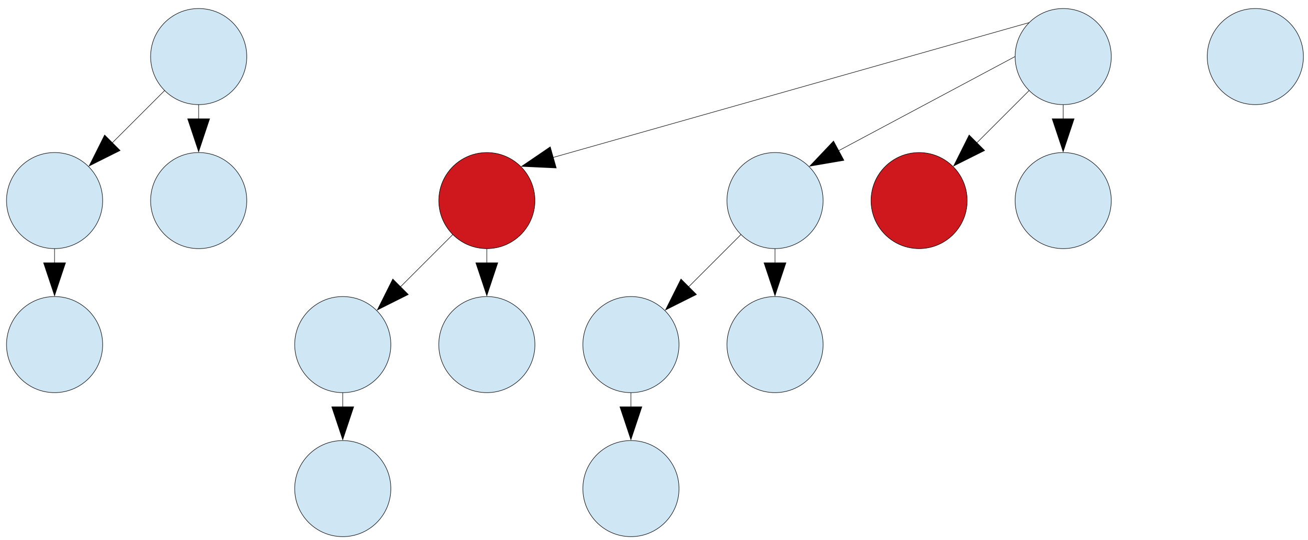
Rule: Nodes can lose at most one child. If a node loses two children, cut it from its parent.



Goal: Make tree sizes grow exponentially with order, but still allow for subtrees to be cut out quickly.

Intuition: Allow trees to get somewhat imbalanced, slowly propagating information to the root.

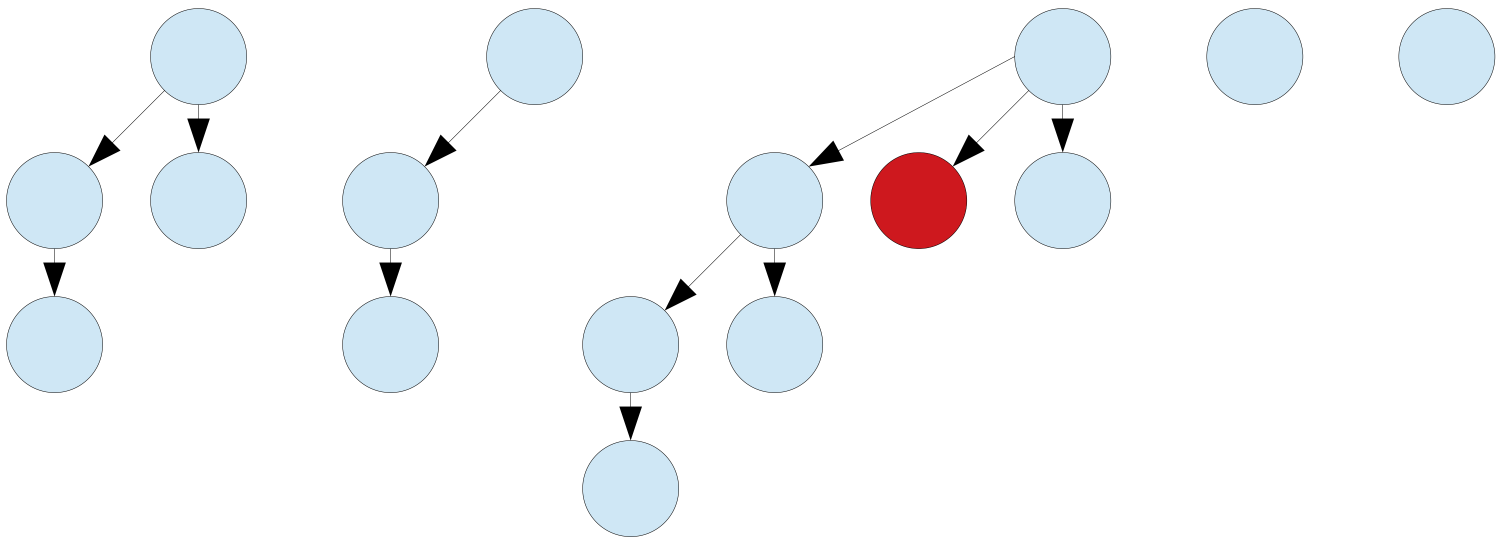
Rule: Nodes can lose at most one child. If a node loses two children, cut it from its parent.



Goal: Make tree sizes grow exponentially with order, but still allow for subtrees to be cut out quickly.

Intuition: Allow trees to get somewhat imbalanced, slowly propagating information to the root.

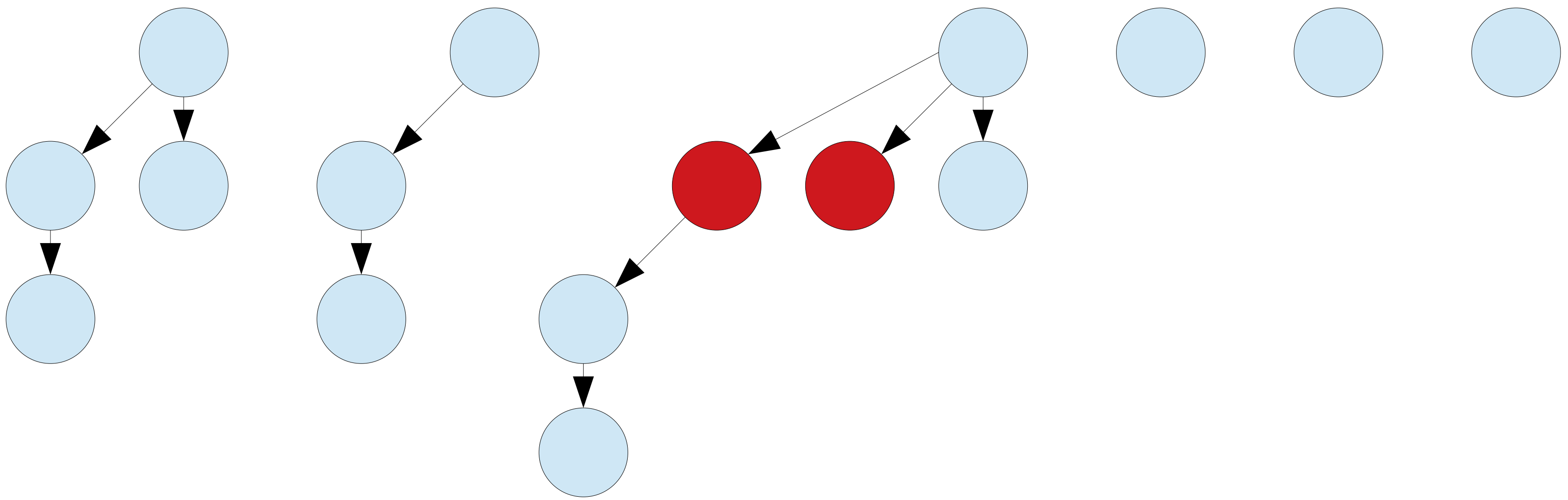
Rule: Nodes can lose at most one child. If a node loses two children, cut it from its parent.



Goal: Make tree sizes grow exponentially with order, but still allow for subtrees to be cut out quickly.

Intuition: Allow trees to get somewhat imbalanced, slowly propagating information to the root.

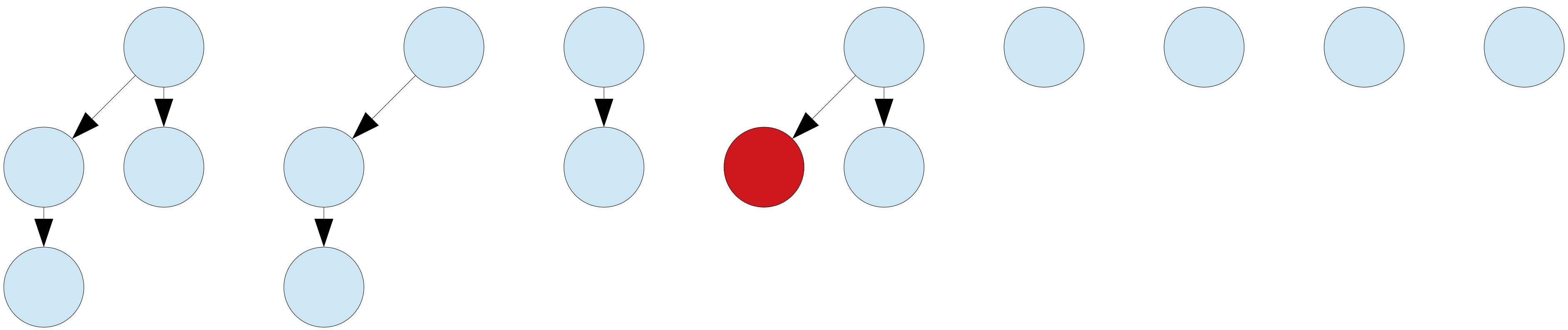
Rule: Nodes can lose at most one child. If a node loses two children, cut it from its parent.



Goal: Make tree sizes grow exponentially with order, but still allow for subtrees to be cut out quickly.

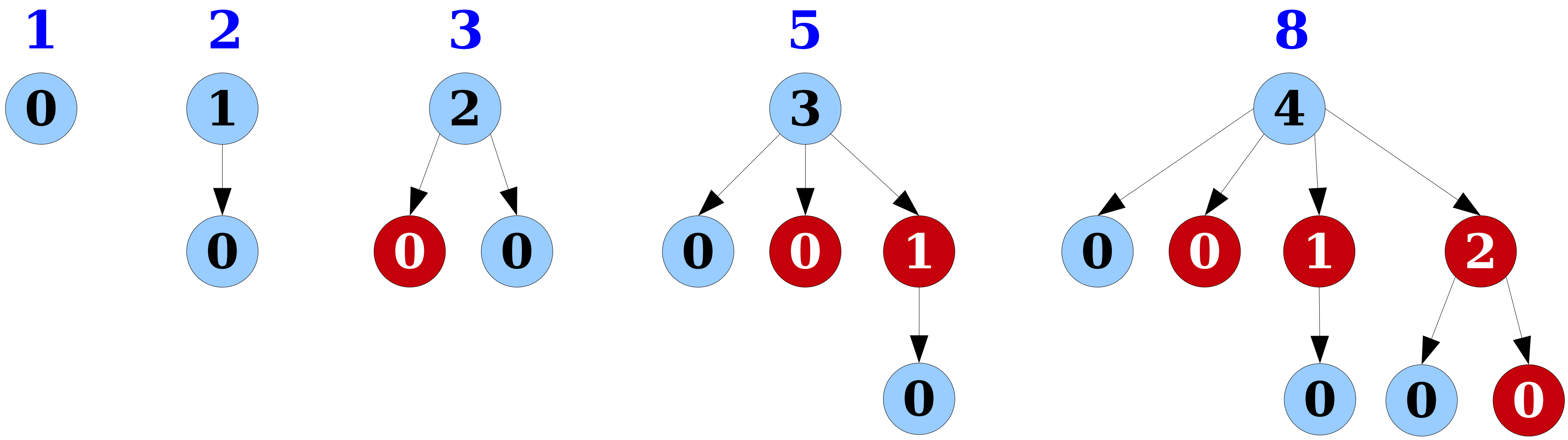
Intuition: Allow trees to get somewhat imbalanced, slowly propagating information to the root.

Rule: Nodes can lose at most one child. If a node loses two children, cut it from its parent.

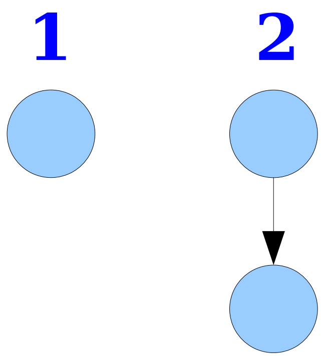


Goal: Make tree sizes grow exponentially with order, but still allow for subtrees to be cut out quickly.

Maximally-Damaged Trees



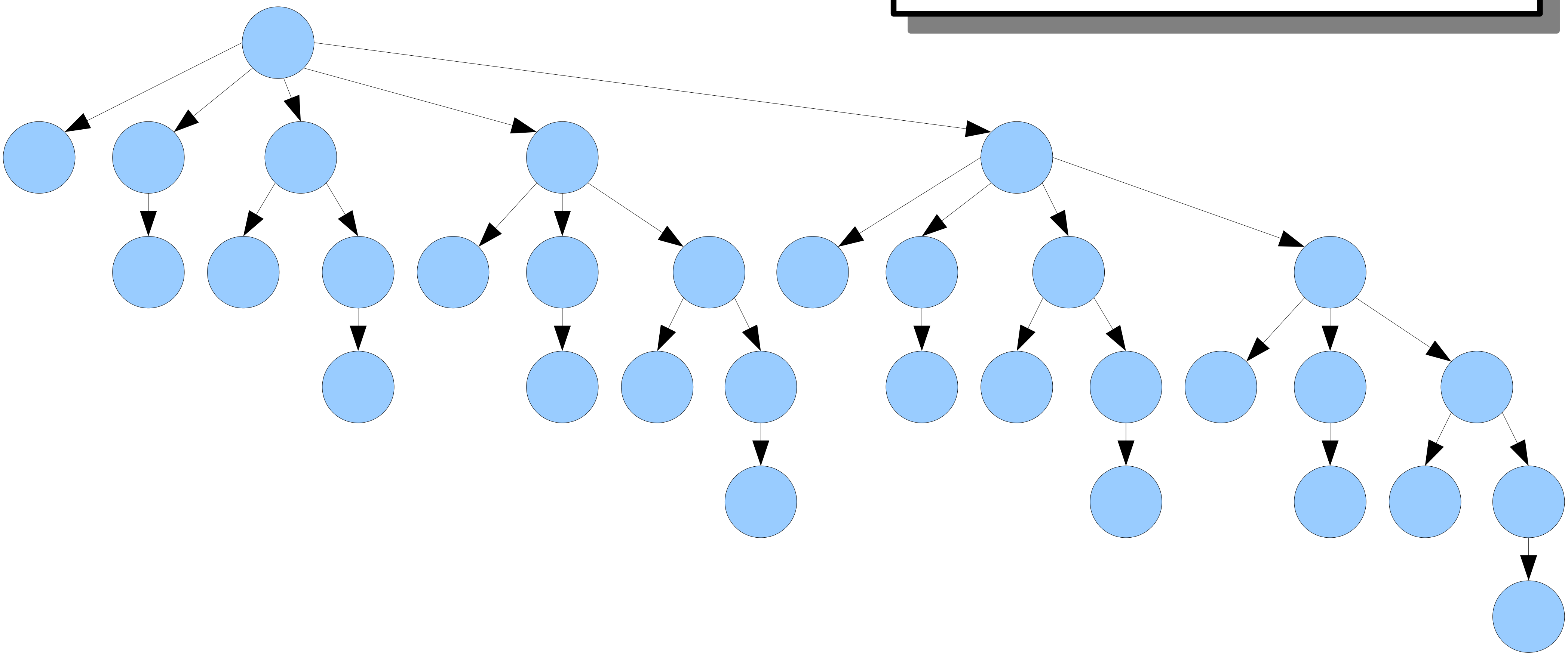
Claim: The minimum number of nodes in a tree of order *k* is $\boldsymbol{F_{k+2}}$



Theorem: The minimum number of nodes in a tree of order *k* is $\boldsymbol{F_{k+2}}$.

Thanks to former CS166ers Kevin Tan and Max Arseneault for this proof approach!

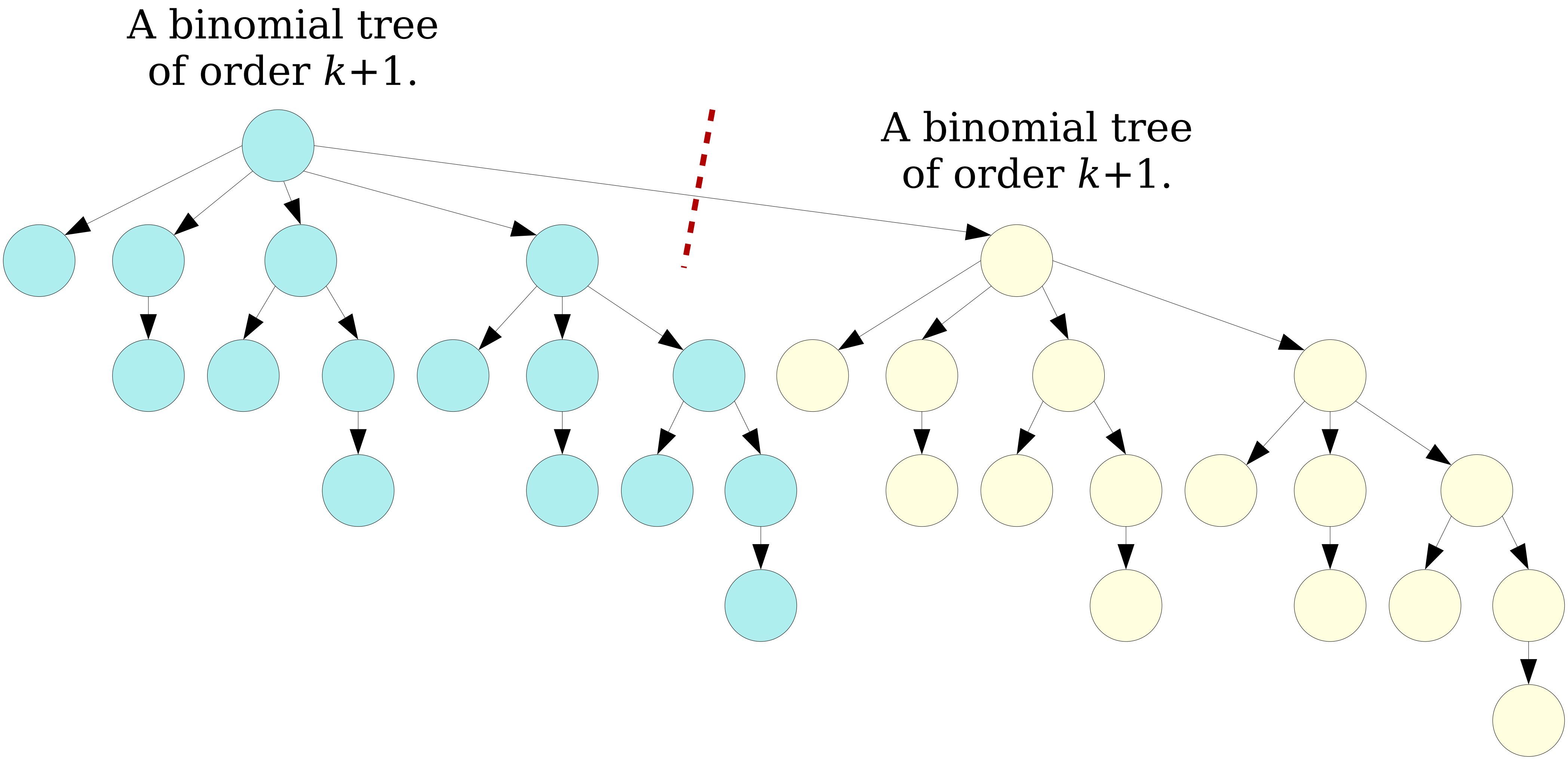
A binomial tree of order $k+2$.



What's the maximum amount of damage we can do to this tree without cutting any of the direct children of the root?

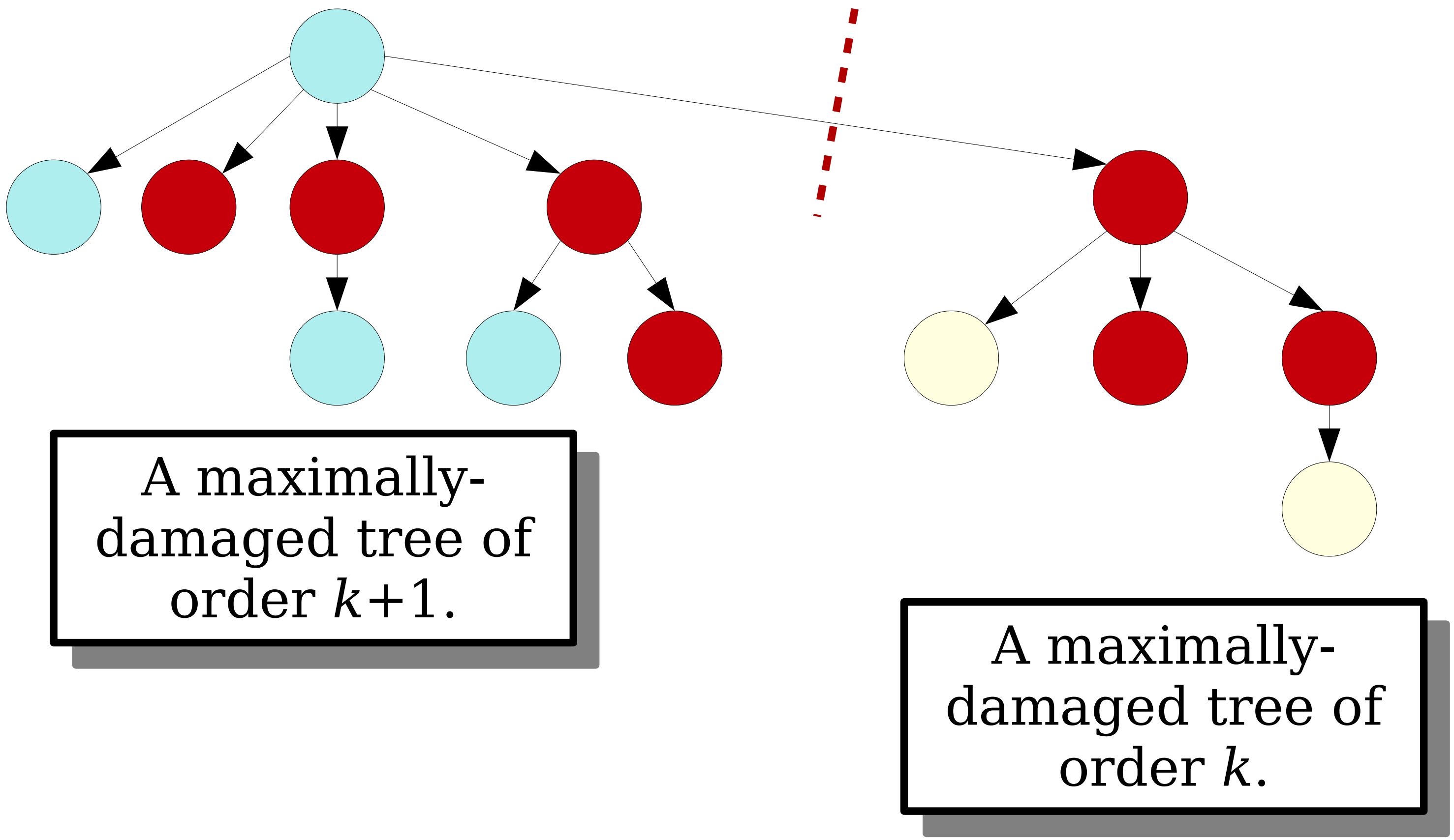
Theorem: The minimum number of nodes in a tree of order k is F_{k+2}.

Thanks to former CS166ers Kevin Tan and Max Arseneault for this proof approach!



Theorem: The minimum number of nodes in a tree of order k is F_{k+2}.

Thanks to former CS166ers Kevin Tan and Max Arseneault for this proof approach!



Theorem: The minimum number of nodes in a tree of order k is F_{k+2}.

Thanks to former CS166ers Kevin Tan and Max Arseneault for this proof approach!

Fact: $F_k = \Theta(\varphi^k)$, where

$$\varphi = \frac{1+\sqrt{5}}{2}$$

is the golden ratio.

Corollary: The number of nodes in a tree of order k grows exponentially with k (approximately 1.61^k versus our previous 2^k).

Theorem: The minimum number of nodes in a tree of order k is $\boldsymbol{F_{k+2}}$.

Thanks to former CS166ers Kevin Tan and Max Arseneault for this proof approach!

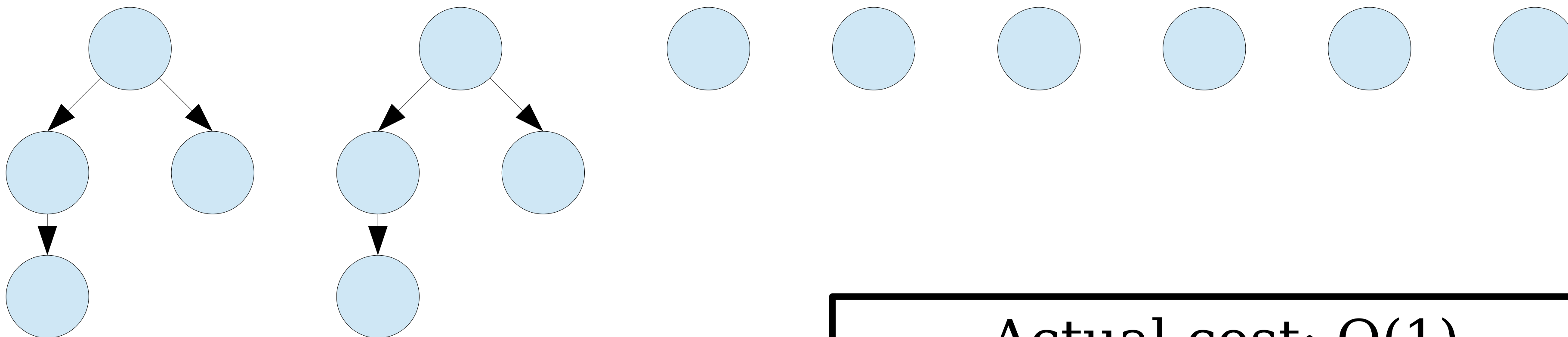
A ***Fibonacci heap*** is a lazy binomial heap with ***decrease-key*** implemented using the “lose at most one child” marking scheme.

How fast are the operations on Fibonacci heaps?

$$\Phi = t$$

where

t is the number of trees.



Actual cost: O(1)
ΔΦ: +1

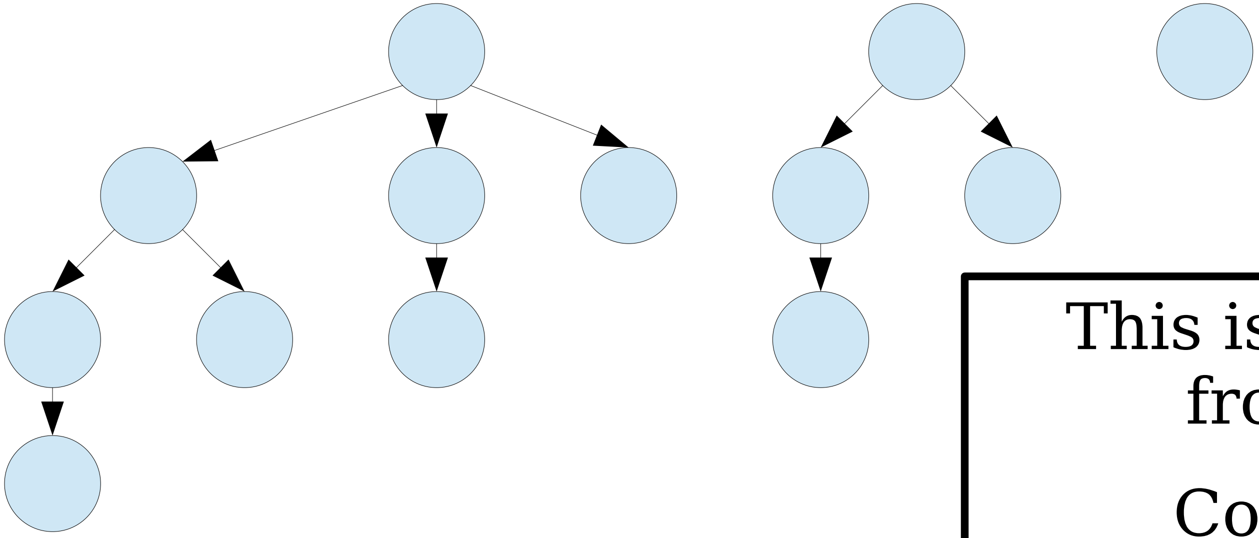
Amortized cost: **O(1)**.

Each ***enqueue*** slowly introduces trees.
Each ***extract-min*** rapidly cleans them up.

$$\Phi = t$$

where

t is the number of trees.



This is the same analysis from last lecture!

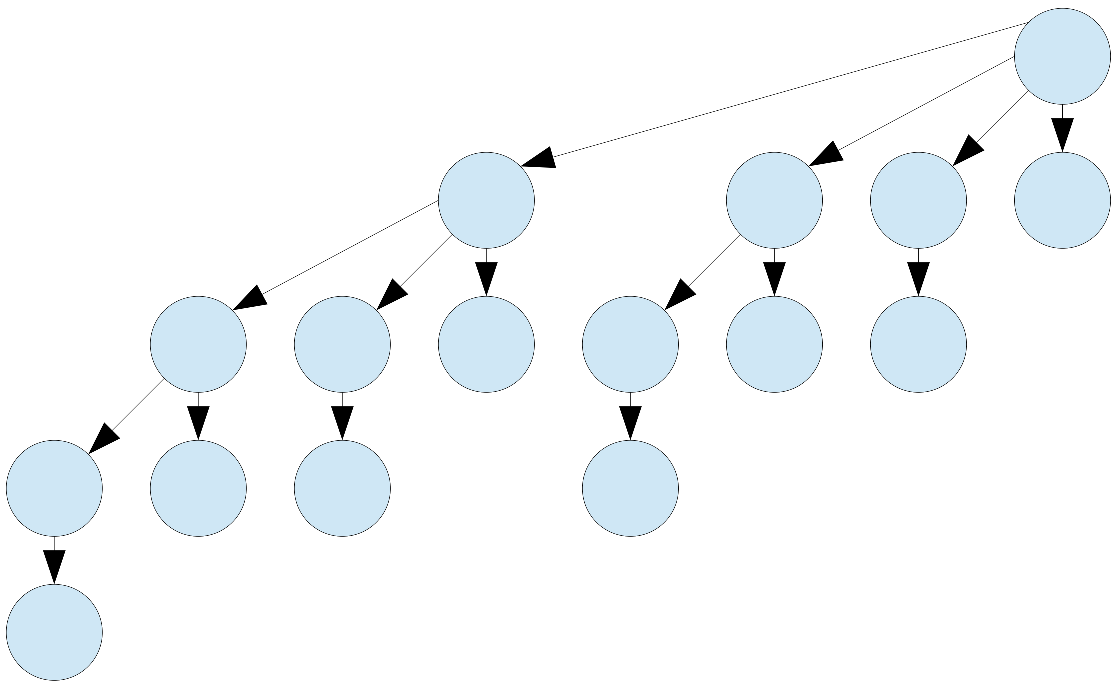
Cost: O(t + log n).
ΔΦ: O(-t + log n).
Amortized cost: **O(log *n*)**.

Each ***enqueue*** slowly introduces trees.
Each ***extract-min*** rapidly cleans them up.

$$\Phi = t + m$$

where

t is the number of trees and
m is the number of marked nodes.

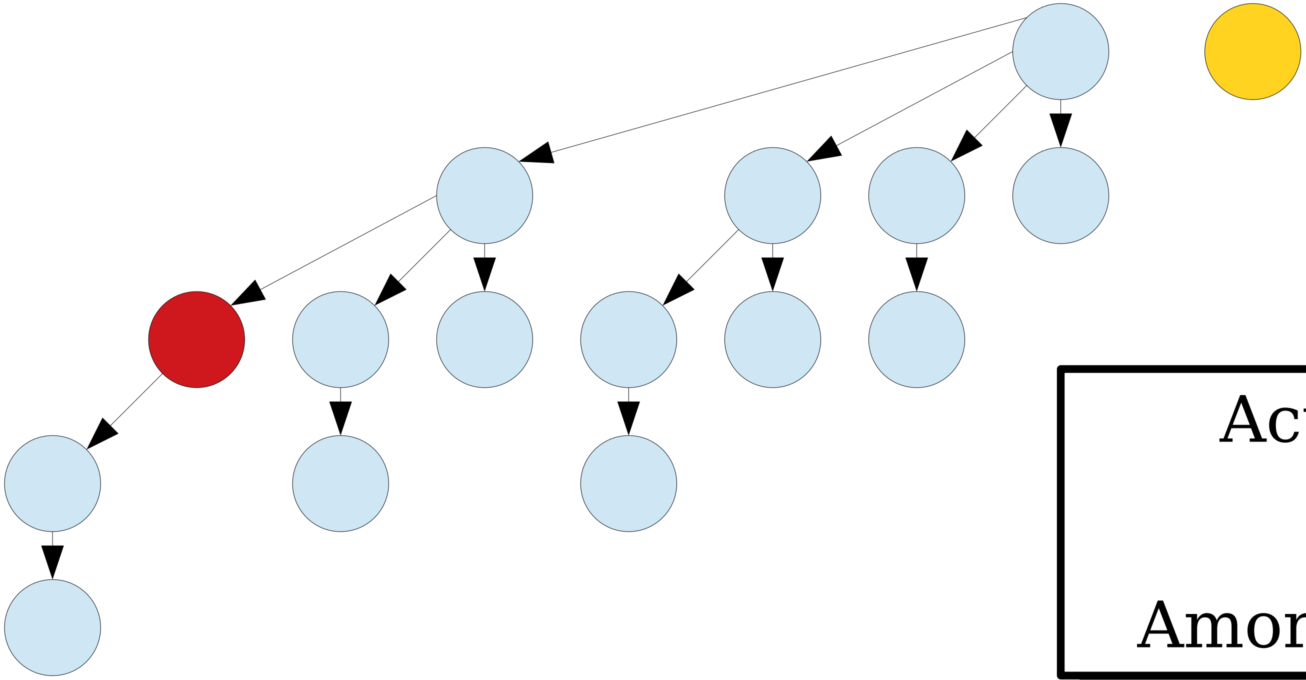


Idea: Factor the number of marked nodes into our potential to offset the cost of cascading cuts.

$$\Phi = t + m$$

where

t is the number of trees and
m is the number of marked nodes.



Actual cost: O(1)
$\Delta\Phi$: +2.

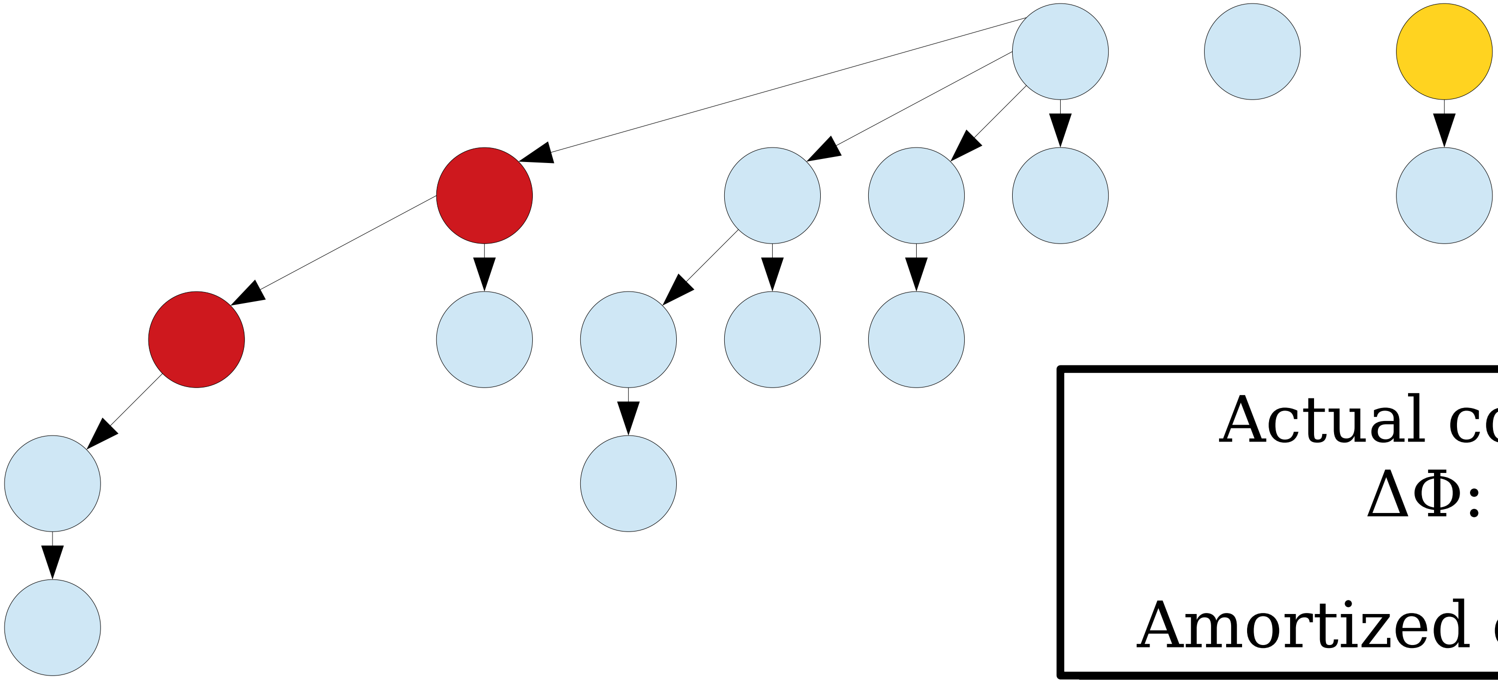
Amortized cost: **O(1)**.

Idea: Factor the number of marked nodes into our potential to offset the cost of cascading cuts.

$$\Phi = t + m$$

where

t is the number of trees and
m is the number of marked nodes.



Actual cost: O(1)
$\Delta\Phi$: +2.

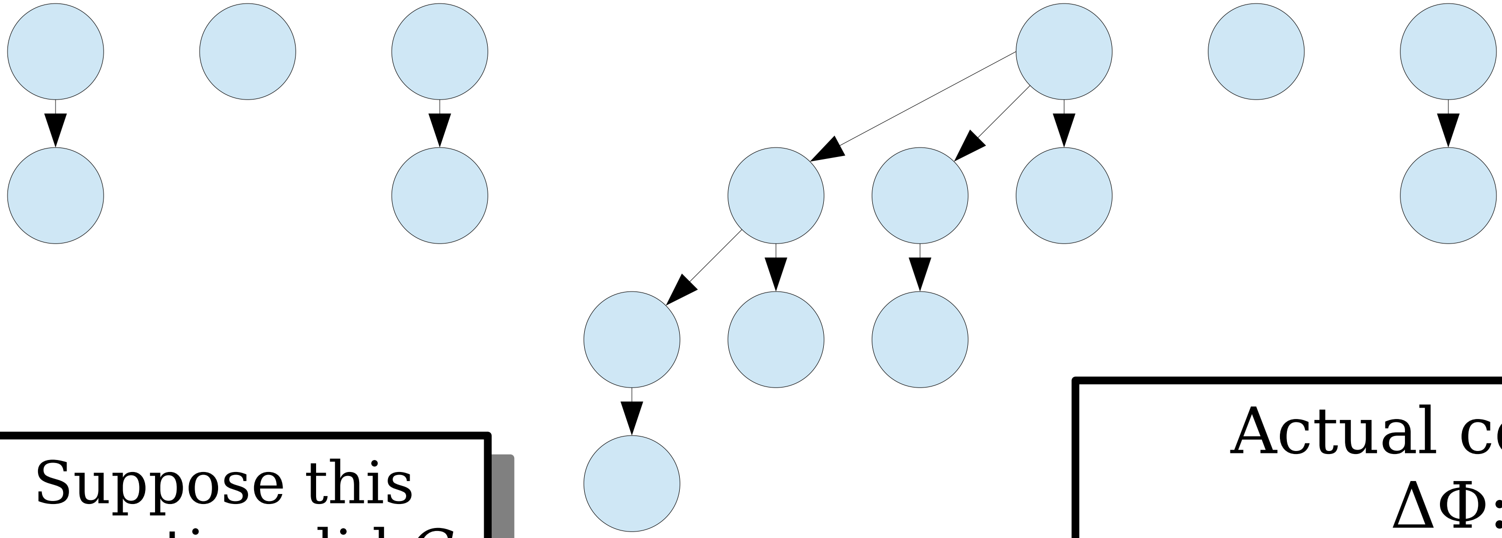
Amortized cost: **O(1)**.

Idea: Factor the number of marked nodes into our potential to offset the cost of cascading cuts.

$$\Phi = t + m$$

where

t is the number of trees and
m is the number of marked nodes.



Suppose this operation did C total cuts.

Actual cost: O(C)
ΔΦ: +1

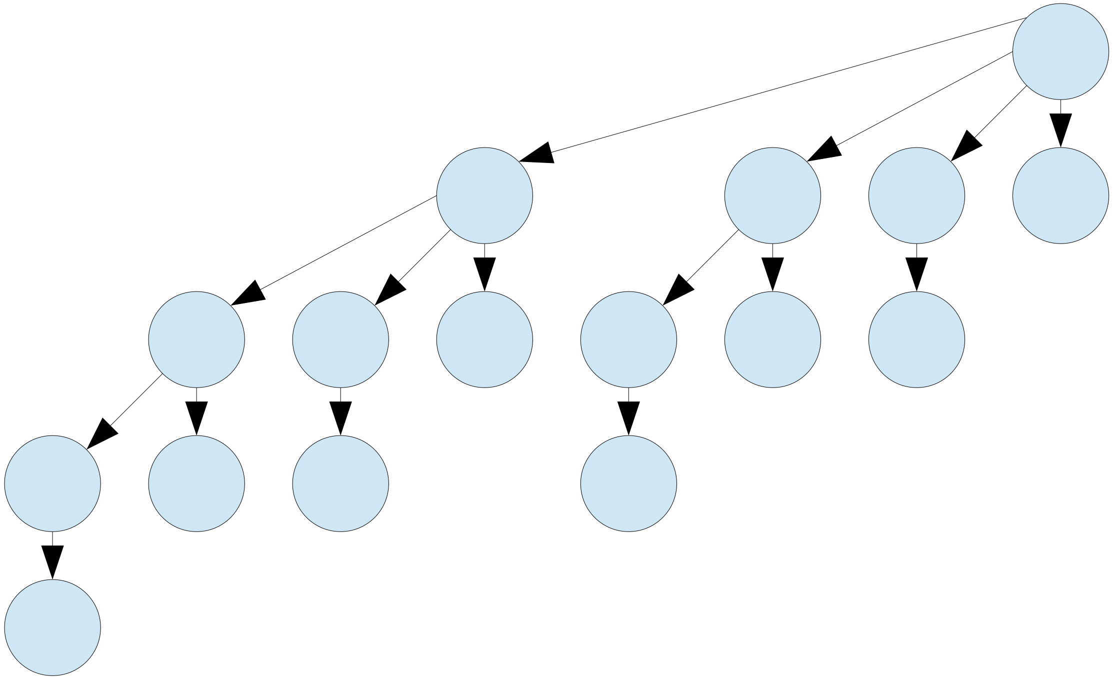
Amortized cost: **O(C)**.

Idea: Factor the number of marked nodes into our potential to offset the cost of cascading cuts.

$$\Phi = t + 2m$$

where

t is the number of trees and
m is the number of marked nodes.

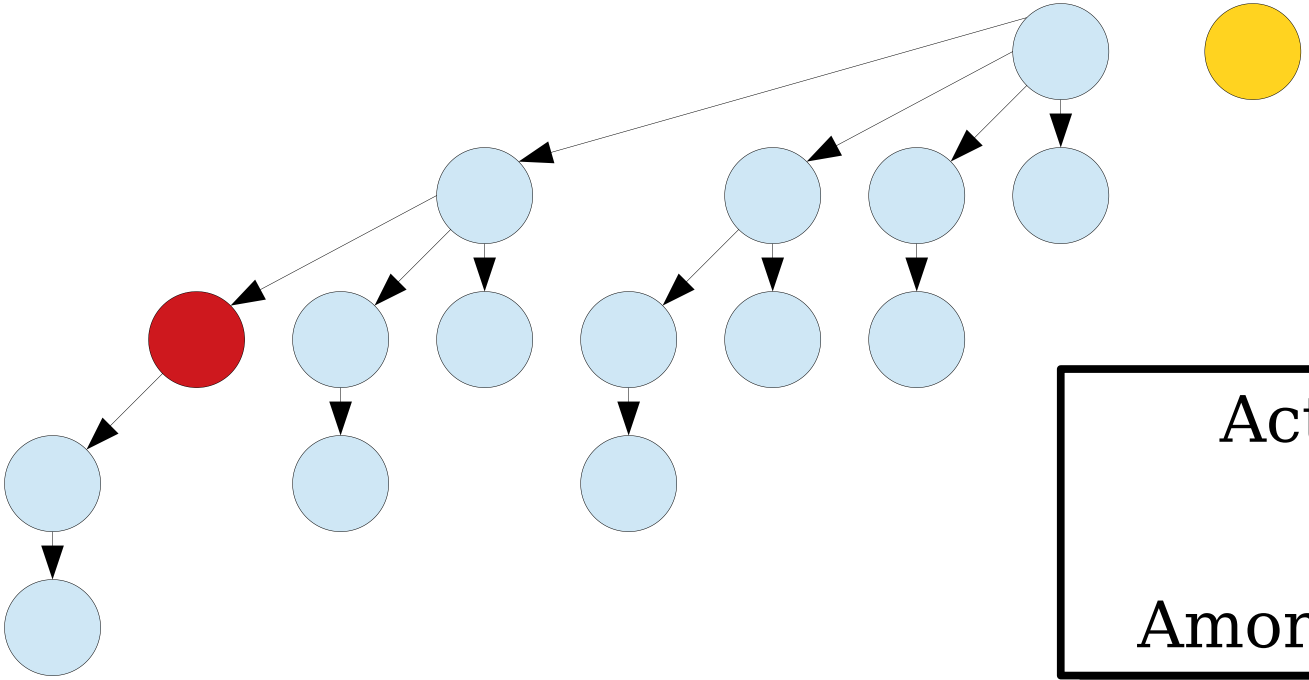


Idea 2: Each ***decrease-key*** hurts twice: once in a cascading cut, and once in an ***extract-min***.

$$\Phi = t + 2m$$

where

t is the number of trees and
m is the number of marked nodes.



Actual cost: O(1)
ΔΦ: +3.

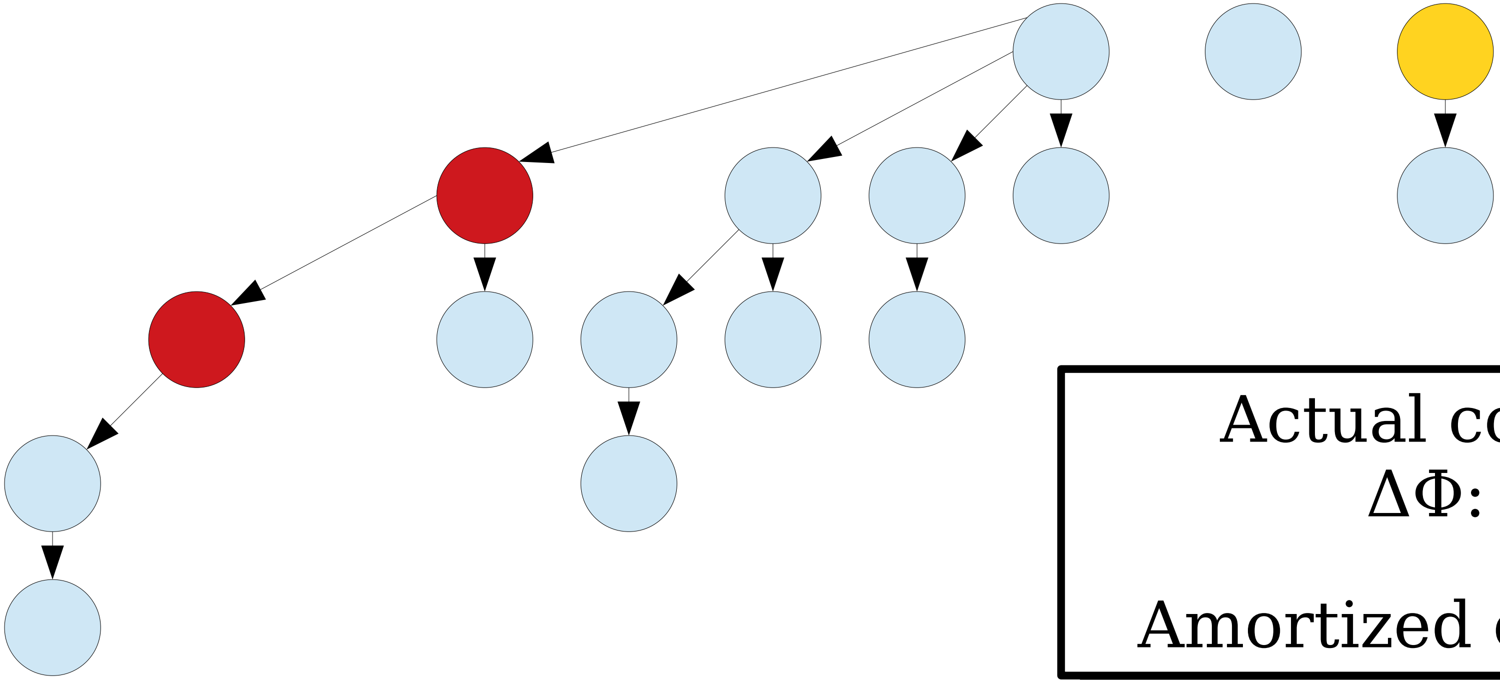
Amortized cost: **O(1)**.

Idea 2: Each ***decrease-key*** hurts twice: once in a cascading cut, and once in an ***extract-min***.

$$\Phi = t + 2m$$

where

t is the number of trees and
m is the number of marked nodes.



Actual cost: O(1)
ΔΦ: +3.

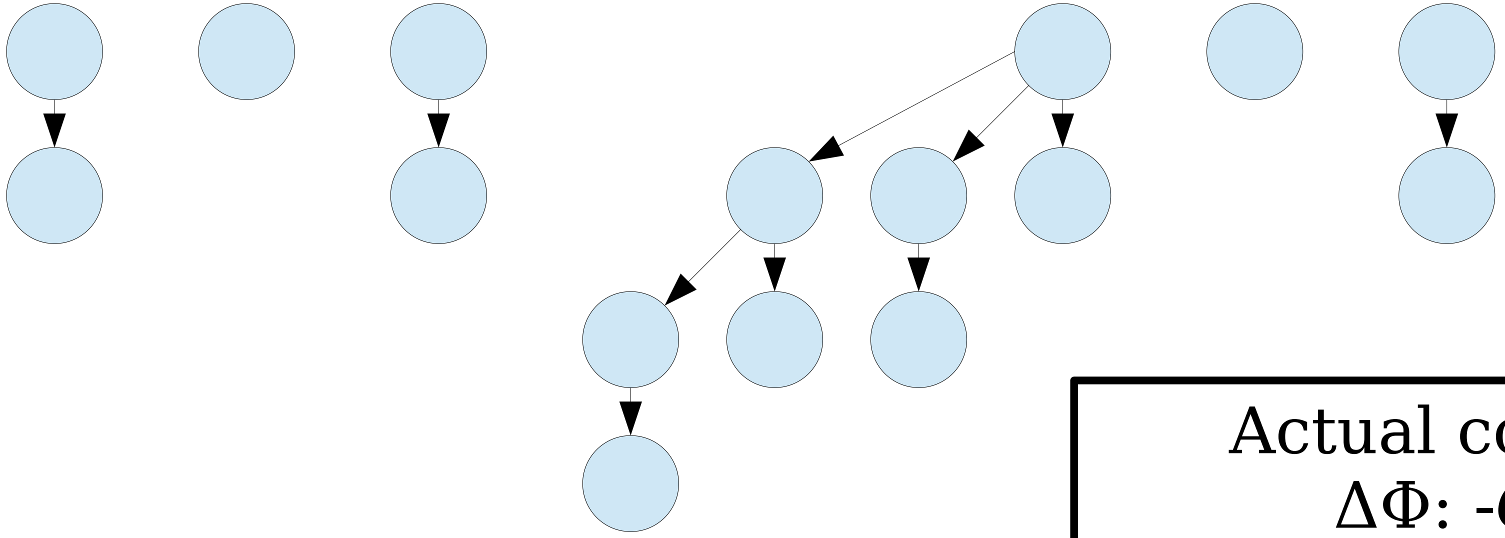
Amortized cost: **O(1)**.

Idea 2: Each ***decrease-key*** hurts twice: once in a cascading cut, and once in an ***extract-min***.

$$\Phi = t + 2m$$

where

t is the number of trees and
m is the number of marked nodes.



Actual cost: O(C)
ΔΦ: -C + 1

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

Idea 2: Each ***decrease-key*** hurts twice: once in a cascading cut, and once in an ***extract-min***.

The Overall Analysis

- Here's the final scorecard for the Fibonacci heap.
- These are excellent theoretical runtimes. There's minimal room for improvement!
- Later work made all these operations *worst-case efficient* at a significant increase in both runtime and intellectual complexity.

enqueue: O(1)

find-min: O(1)

meld: O(1)

extract-min: O(log *n*)*

decrease-key: O(1)*

**amortized*

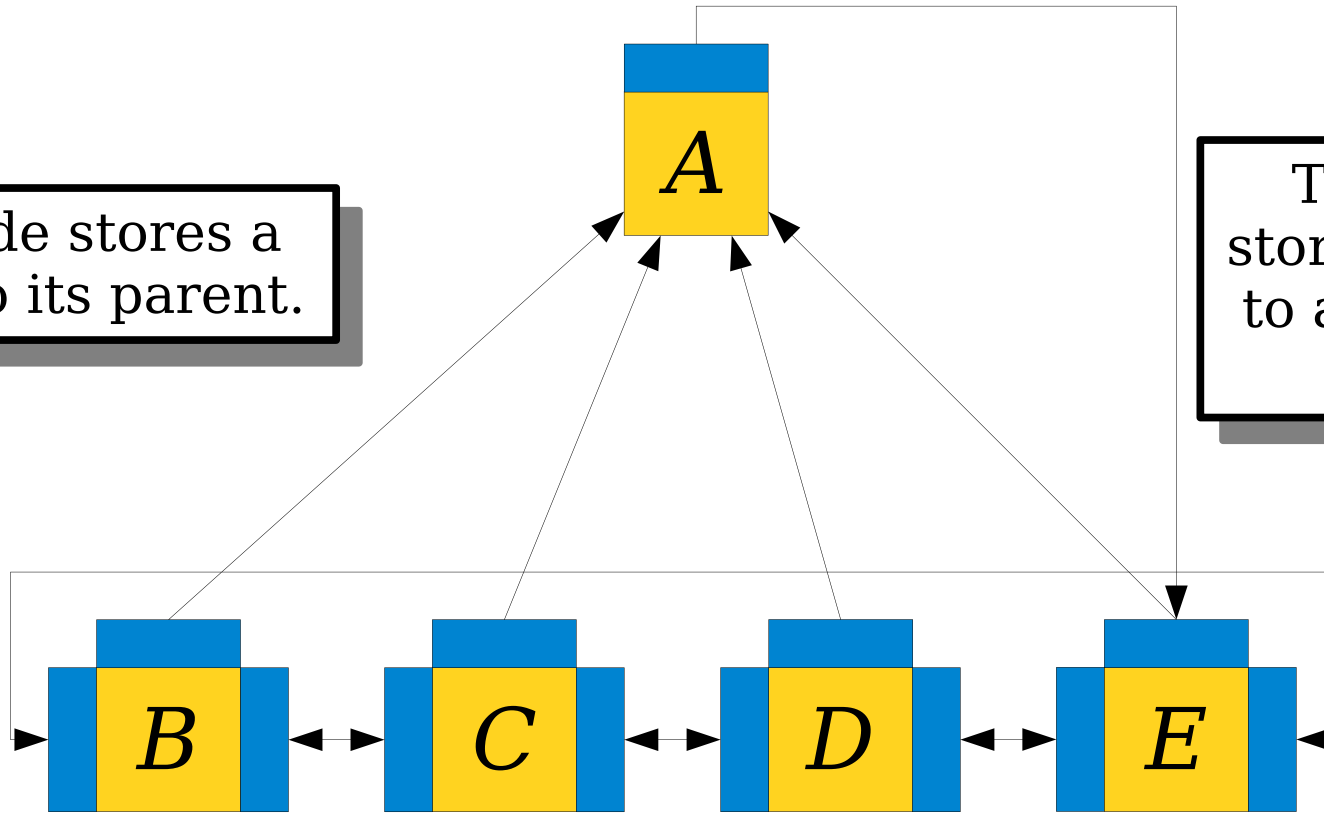
Representation Issues

Representing Trees

- The trees in a Fibonacci heap must be able to do the following:
  - During a merge: Add one tree as a child of the root of another tree.
  - During a cut: Cut a node from its parent in time O(1).
- ***Claim:*** This is trickier than it looks.

The Solution

Each node stores a pointer to its parent.



The parent stores a pointer to an arbitrary child.

The children of each node are in a circularly, doubly-linked list.

Awful Linked Lists

- Trees are stored as follows:
  - Each node stores a pointer to *some* child.
  - Each node stores a pointer to its parent.
  - Each node is in a circularly-linked list of its siblings.
- The following possible are now possible in time O(1):
  - Cut a node from its parent.
  - Add another child node to a node.

Fibonacci Heap Nodes

- Each node in a Fibonacci heap stores
  - A pointer to its parent.
  - A pointer to the next sibling.
  - A pointer to the previous sibling.
  - A pointer to an arbitrary child.
  - A bit for whether it's marked.
  - Its order.
  - Its key.
  - Its element.

In Practice

- In practice, the constant factors on Fibonacci heaps make it slower than other heaps, except on huge graphs or workflows with tons of ***decrease-key***s.
- Why?
  - Huge memory requirements per node.
  - High constant factors on all operations.
  - Poor locality of reference and caching.

In Theory

- That said, Fibonacci heaps are worth knowing about for several reasons:
  - Clever use of a two-tiered potential function shows up in lots of data structures.
  - Implementation of ***decrease-key*** forms the basis for many other advanced priority queues.
  - Gives the theoretically optimal comparison-based implementation of Prim's and Dijkstra's algorithms.

More to Explore

- Since the development of Fibonacci heaps, there have been a number of other priority queues with similar runtimes.
  - In 1986, a powerhouse team (Fredman, Sedgewick, Sleator, and Tarjan) invented the ***pairing heap***. It's much simpler than a Fibonacci heap, is fast in practice, but its runtime bounds are unknown!
  - In 2012, Brodal et al. invented the ***strict Fibonacci heap***. It has the same time bounds as a Fibonacci heap, but in a *worst-case* rather than *amortized* sense.
  - In 2013, Chan invented the ***quake heap***. It matches the asymptotic bounds of a Fibonacci heap but uses a totally different strategy.
- Also interesting to explore: if the weights on the edges in a graph are chosen from a continuous distribution, the expected number of ***decrease-key***s in Dijkstra's algorithm is $O(n \log (m / n))$. That might counsel another heap structure!
- Also interesting to explore: binary heaps generalize to b-ary heaps, where each node has b children. Picking $b = \log (2 + m/n)$ makes Dijkstra and Prim run in time $O(m \log n / \log m/n)$, which is $O(m)$ if $m = \Theta(n^{1+\varepsilon})$ for any $\varepsilon > 0$.
- Recent result: Dijkstra's algorithm with Fibonacci heaps can be combined with other data structures to be ***instance optimal*** for single-source shortest paths; *any* algorithm for solving SSSP while reporting distances in increasing order is at most a constant factor faster than Dijkstra's plus the modified heap.

Next Time

- ***Better-than-Balanced BSTs***
  - When $\Theta(\log n)$ worst case isn't enough.
- ***Shannon Entropy***
  - How predictable is a distribution?
- ***Working Sets and Dynamic Fingers***
  - Improving performance on skewed workflows.