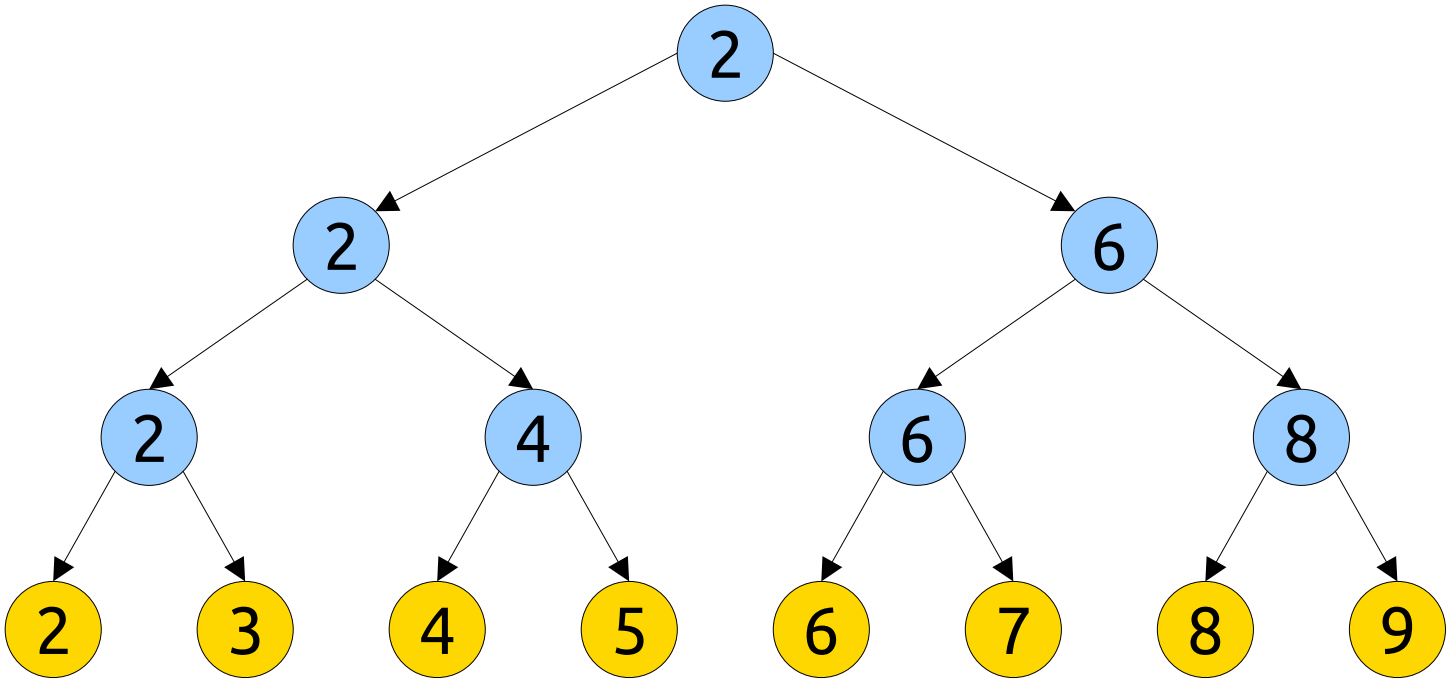
Abdication Heaps

Outline for Today

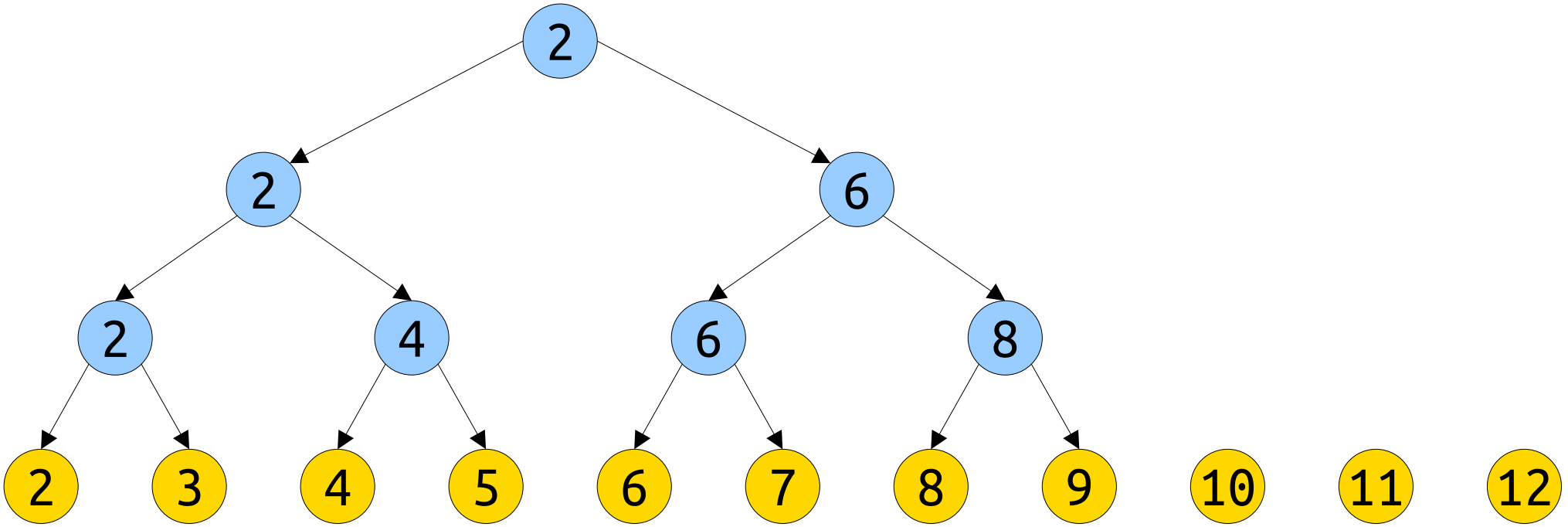
- ***Recap from Last Time***
  - Quick refresher on tournament heaps and lazy tournament heaps.
- ***The Need for decrease-key***
  - An important operation in many graph algorithms.
- ***Abdication Heaps***
  - A data structure efficiently supporting ***decrease-key***.

Recap from Last Time

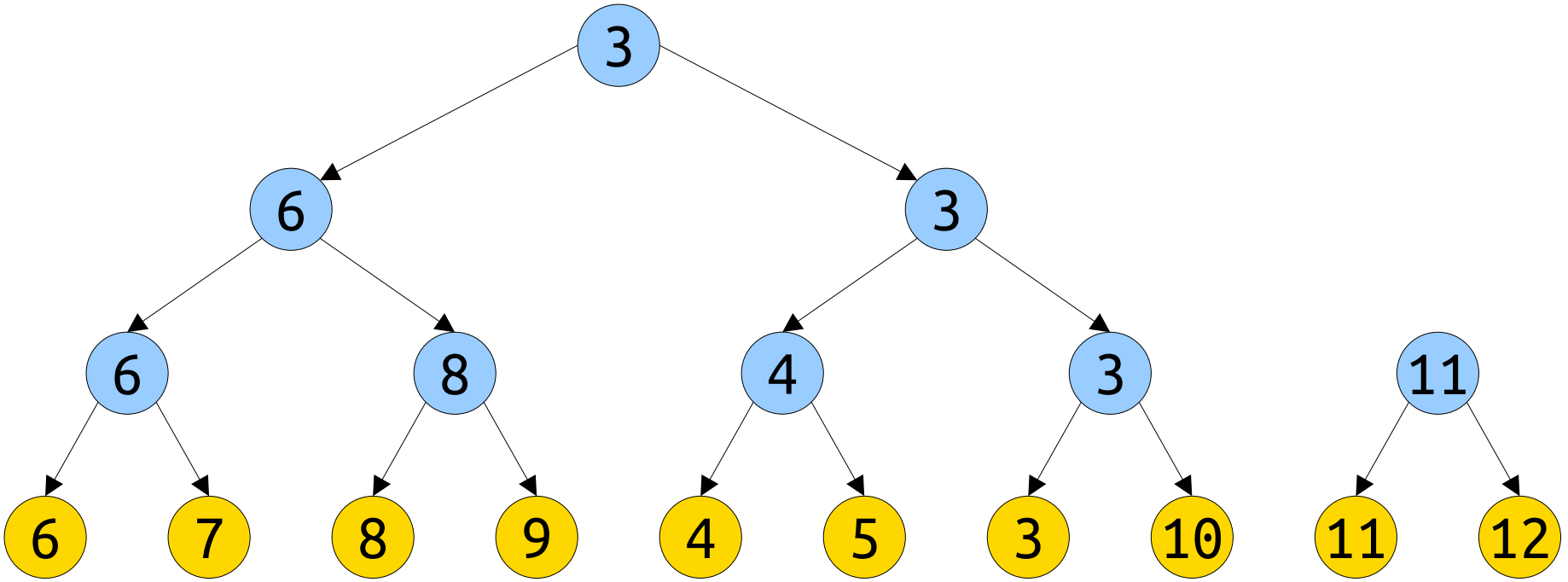
Draw what happens if we ***enqueue*** the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into a lazy tournament heap.



Draw what happens after performing an ***extract-min*** in the lazy tournament heap.



Let’s ***enqueue*** 10, 11, and 12 into this heap.



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

Operation Costs

- Eager Tournament Heap:
  - ***enqueue***: O(log n)
  - ***meld***: O(log n)
  - ***find-min***: O(log n)
  - ***extract-min***: O(log n)
- Lazy Tournament Heap:
  - ***enqueue***: O(1)
  - ***meld***: O(1)
  - ***find-min***: O(1)
  - ***extract-min***: O(log n)*
- **amortized*

Intuition: Each ***extract-min*** has to do a bunch of cleanup for the earlier ***enqueue*** operations, but then leaves us with few trees.

New Stuff!

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

The ***decrease-key*** Operation

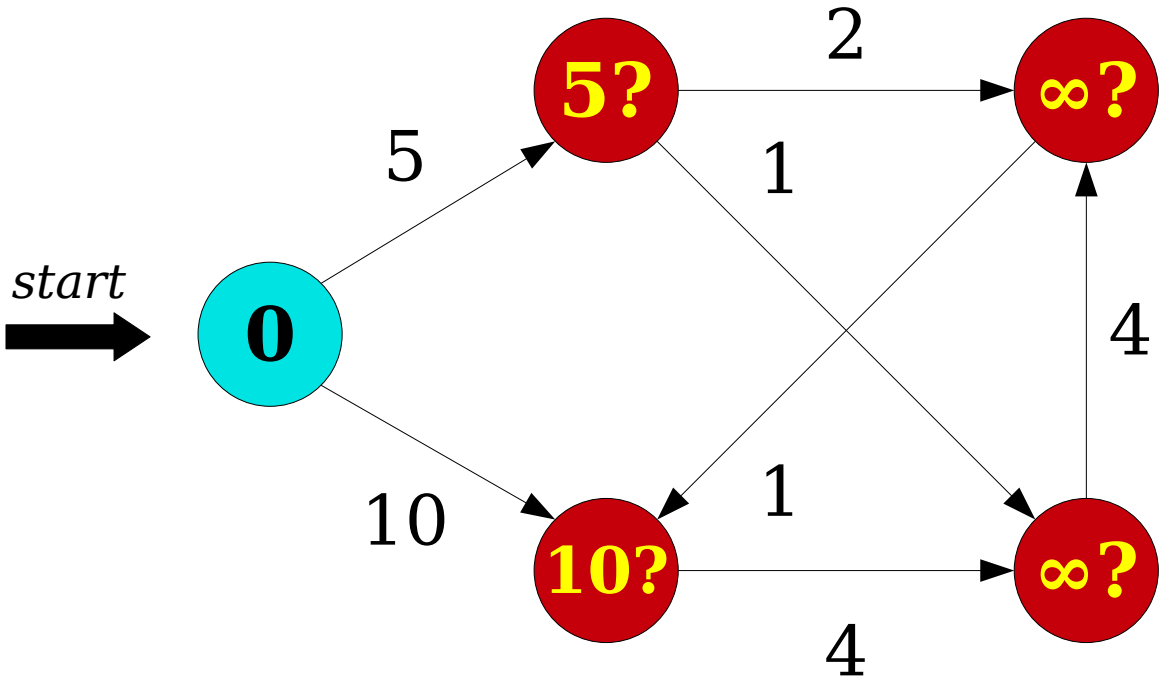
- Some priority queues support the operation ***decrease-key***(v, k), which works as follows:

  Given a pointer to an element v, 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.



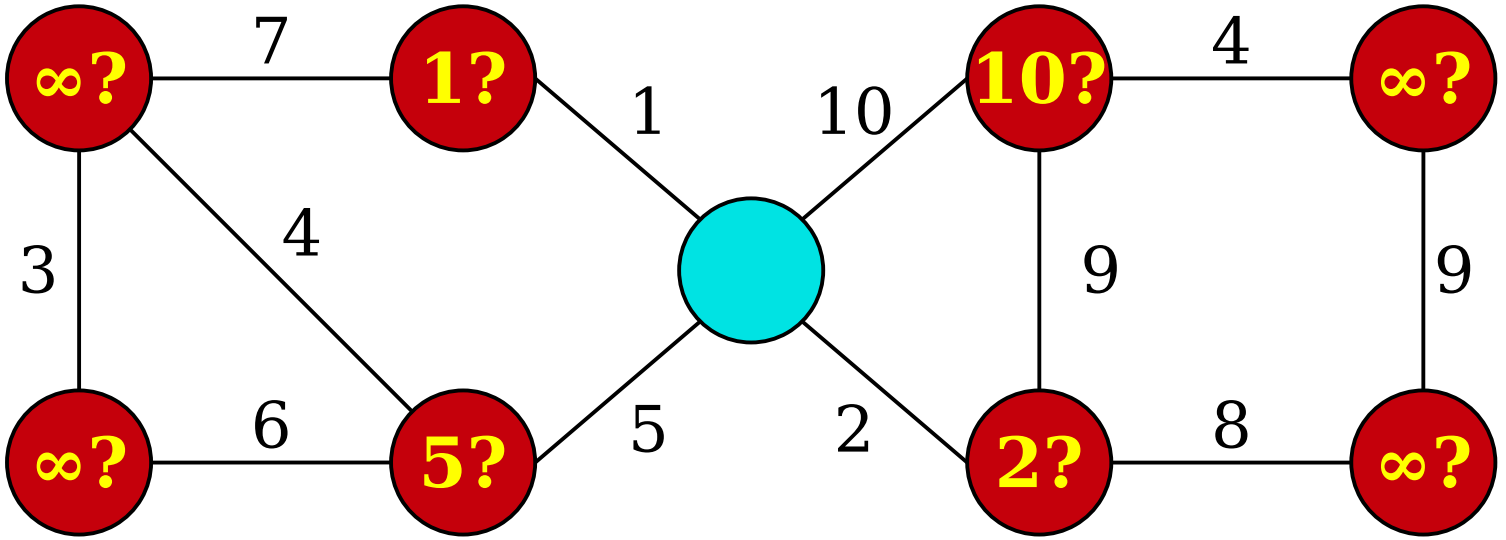
Dijkstra and ***decrease-key***

- 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

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

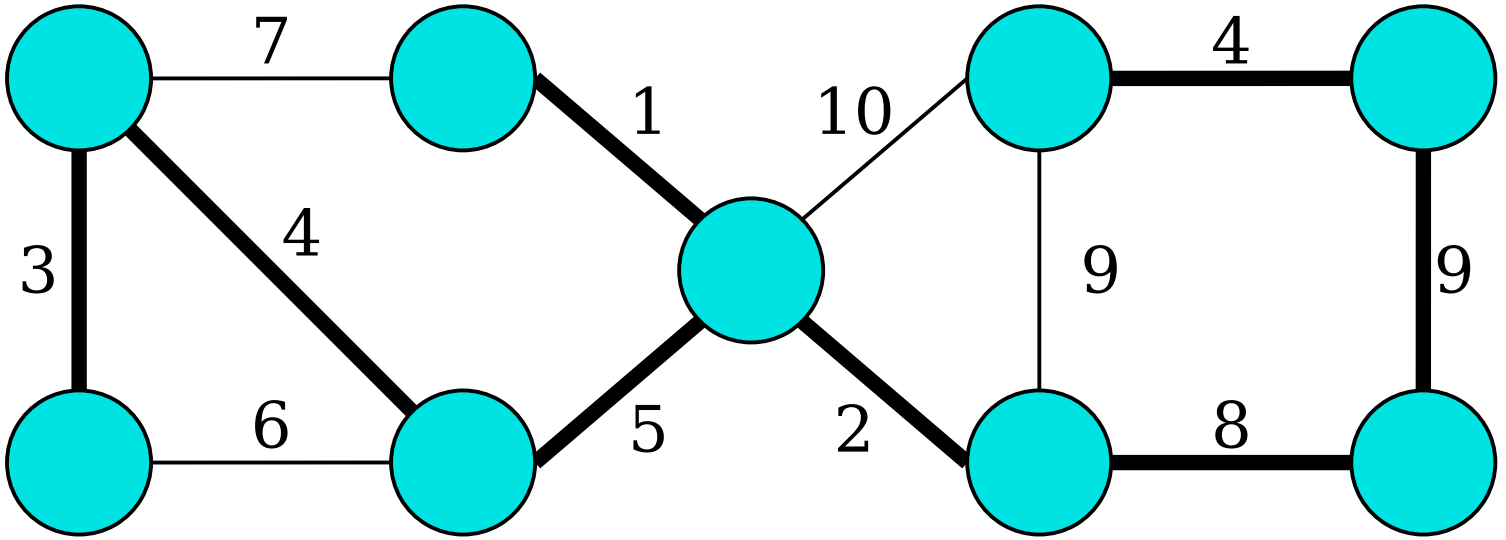
Prim and *decrease-key*

- 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 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 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

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

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:

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

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

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

Standard Approaches

- In a lazy tournament 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:

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

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

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

Where We're Going

- The ***abdication 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:

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

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

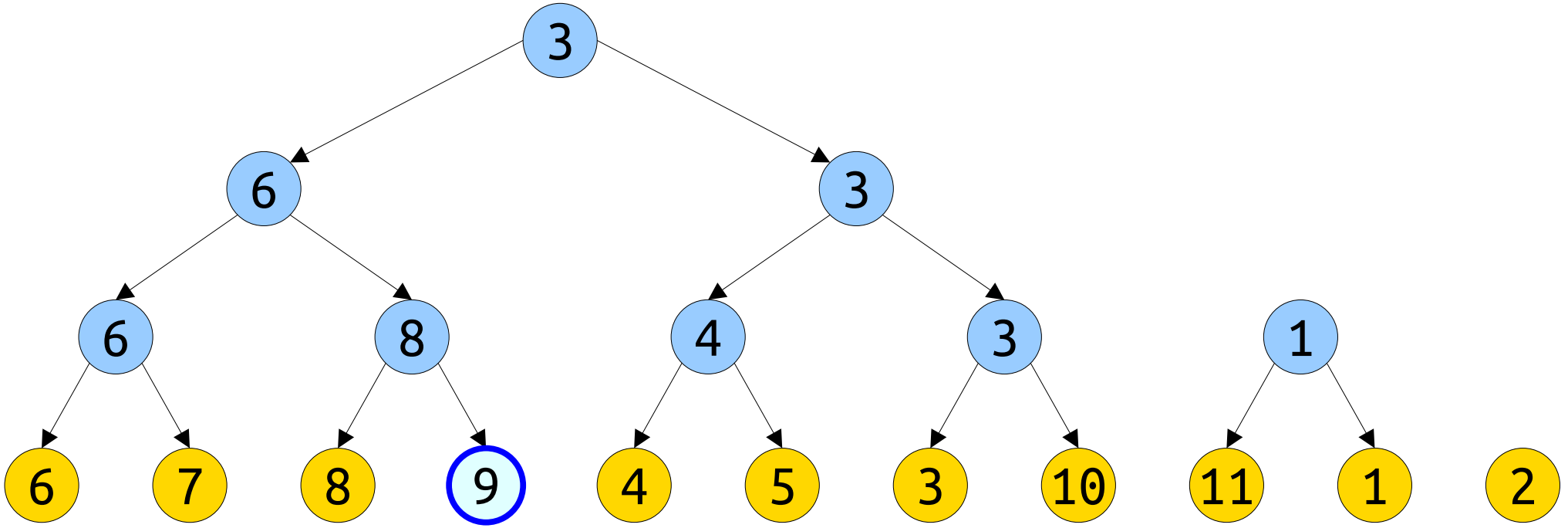
$$= \mathbf{O(m + n \log n)}$$

- This is theoretically optimal for a comparison-based priority queue in Dijkstra's or Prim's algorithms.

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

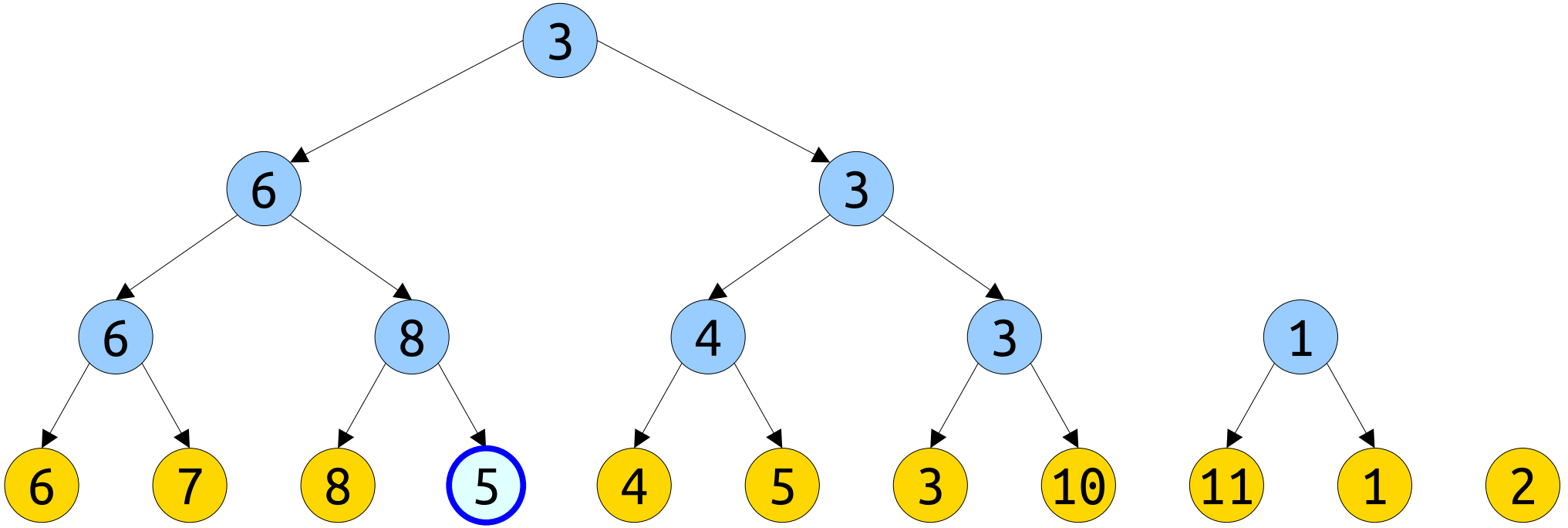
Tournament Heaps Revisited

- How might we implement ***decrease-key*** in a lazy tournament heap?
- ***Idea:*** When ***enqueue***-ing an element, hand out a pointer to the leaf corresponding to that element.
- Then, do do a ***decrease-key***, walk upwards from the leaf to the root of the tree, changing values as needed.



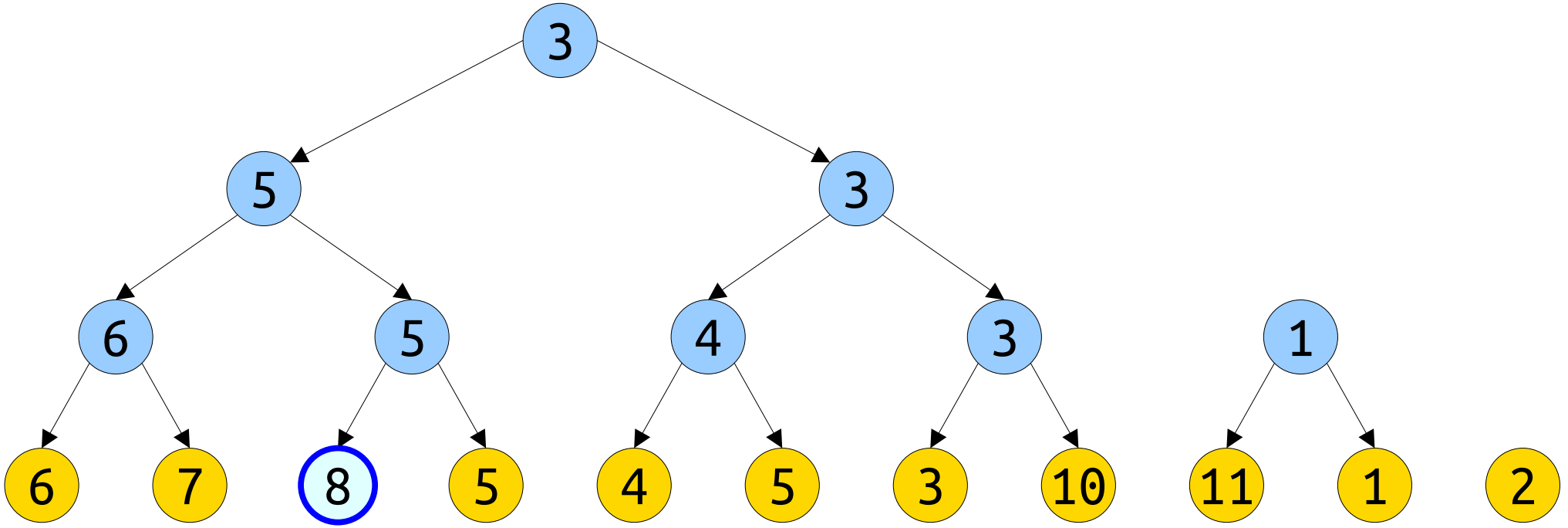
Tournament Heaps Revisited

- How might we implement ***decrease-key*** in a lazy tournament heap?
- ***Idea:*** When ***enqueue***-ing an element, hand out a pointer to the leaf corresponding to that element.
- Then, do do a ***decrease-key***, walk upwards from the leaf to the root of the tree, changing values as needed.



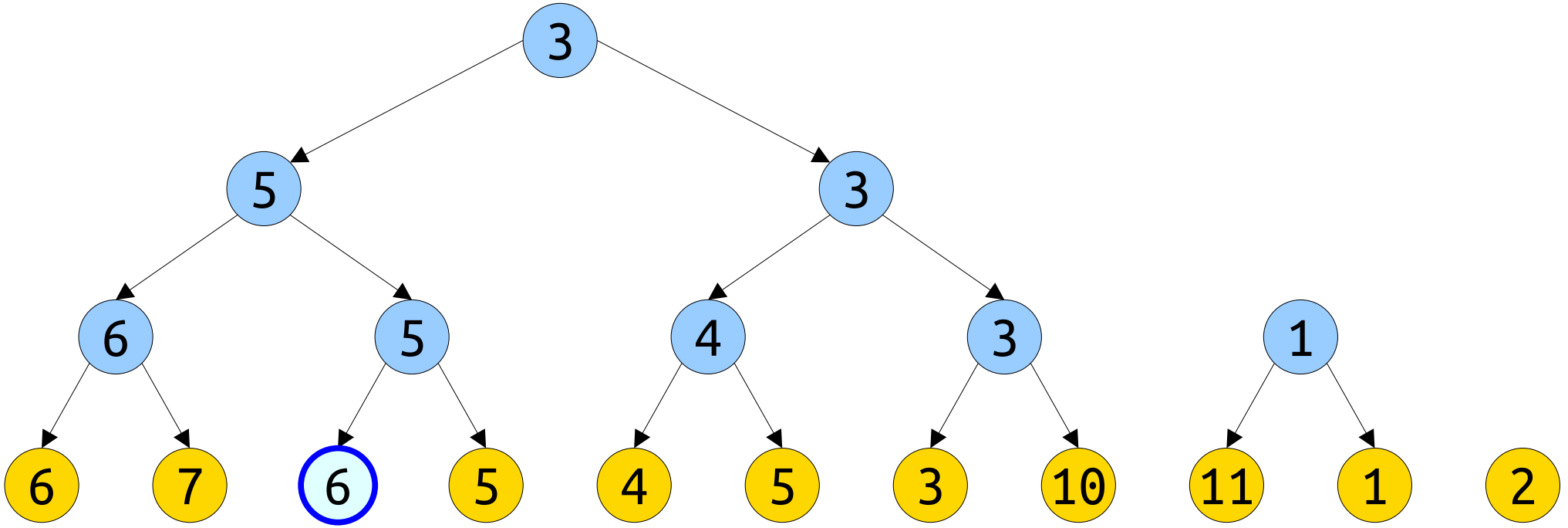
Tournament Heaps Revisited

- How might we implement ***decrease-key*** in a lazy tournament heap?
- ***Idea:*** When ***enqueue***-ing an element, hand out a pointer to the leaf corresponding to that element.
- Then, do do a ***decrease-key***, walk upwards from the leaf to the root of the tree, changing values as needed.



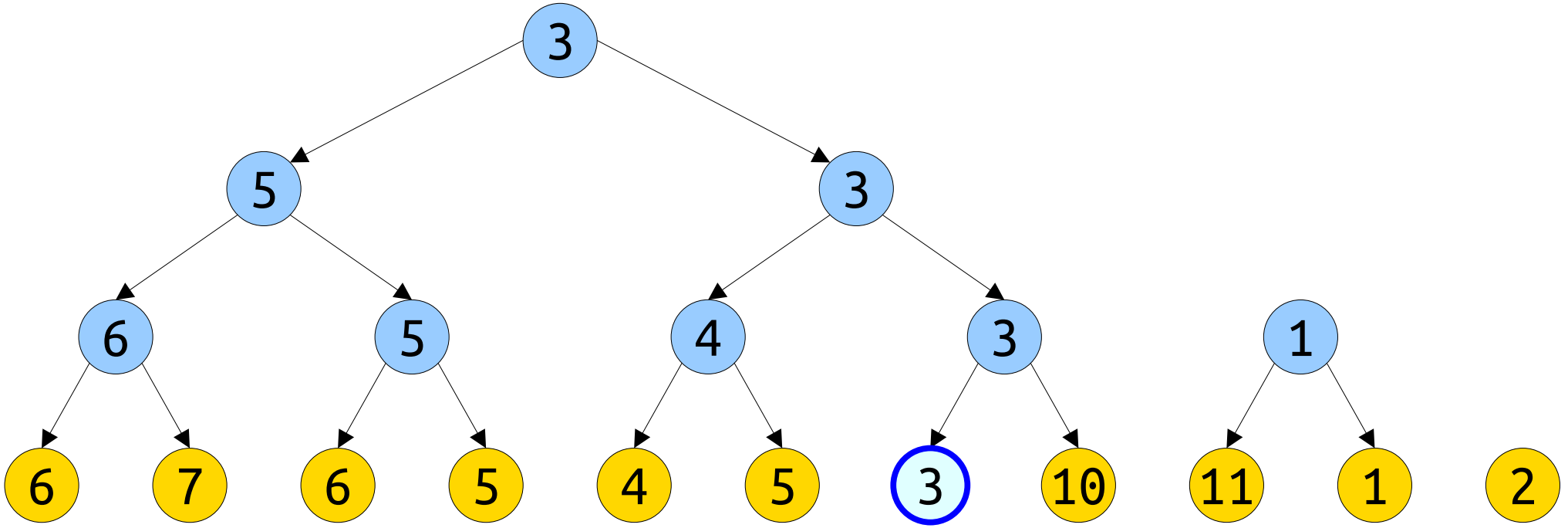
Tournament Heaps Revisited

- How might we implement ***decrease-key*** in a lazy tournament heap?
- ***Idea:*** When ***enqueue***-ing an element, hand out a pointer to the leaf corresponding to that element.
- Then, do do a ***decrease-key***, walk upwards from the leaf to the root of the tree, changing values as needed.



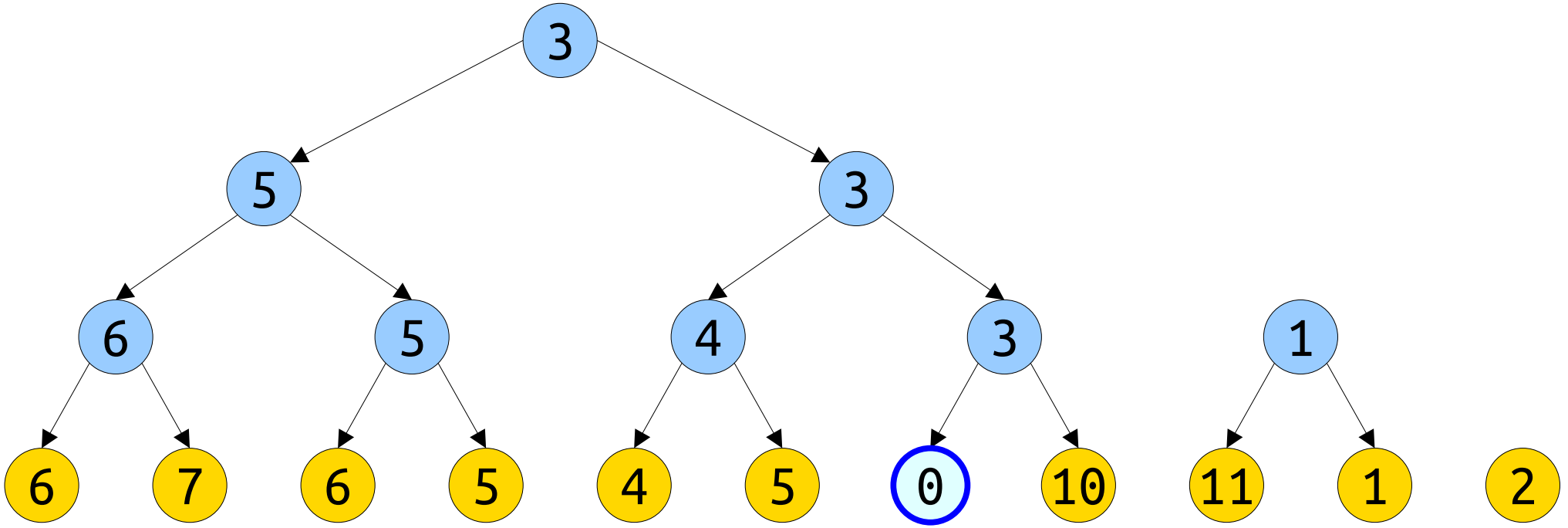
Tournament Heaps Revisited

- How might we implement ***decrease-key*** in a lazy tournament heap?
- ***Idea:*** When ***enqueue***-ing an element, hand out a pointer to the leaf corresponding to that element.
- Then, do do a ***decrease-key***, walk upwards from the leaf to the root of the tree, changing values as needed.



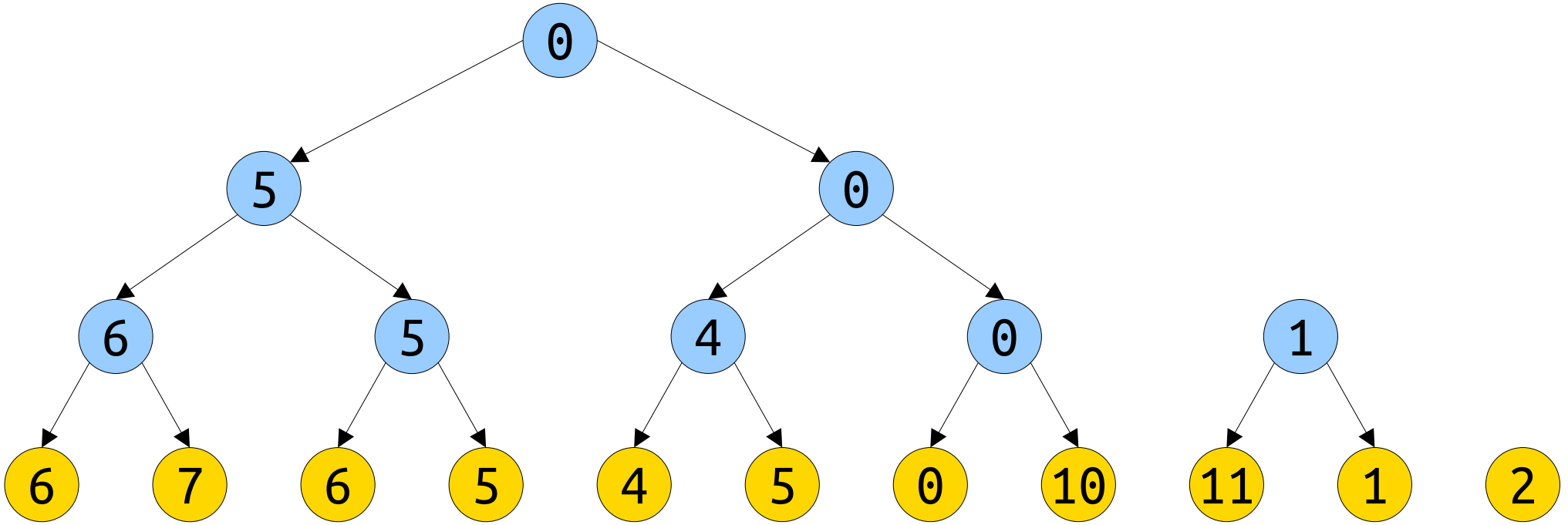
Tournament Heaps Revisited

- How might we implement ***decrease-key*** in a lazy tournament heap?
- ***Idea:*** When ***enqueue***-ing an element, hand out a pointer to the leaf corresponding to that element.
- Then, do do a ***decrease-key***, walk upwards from the leaf to the root of the tree, changing values as needed.



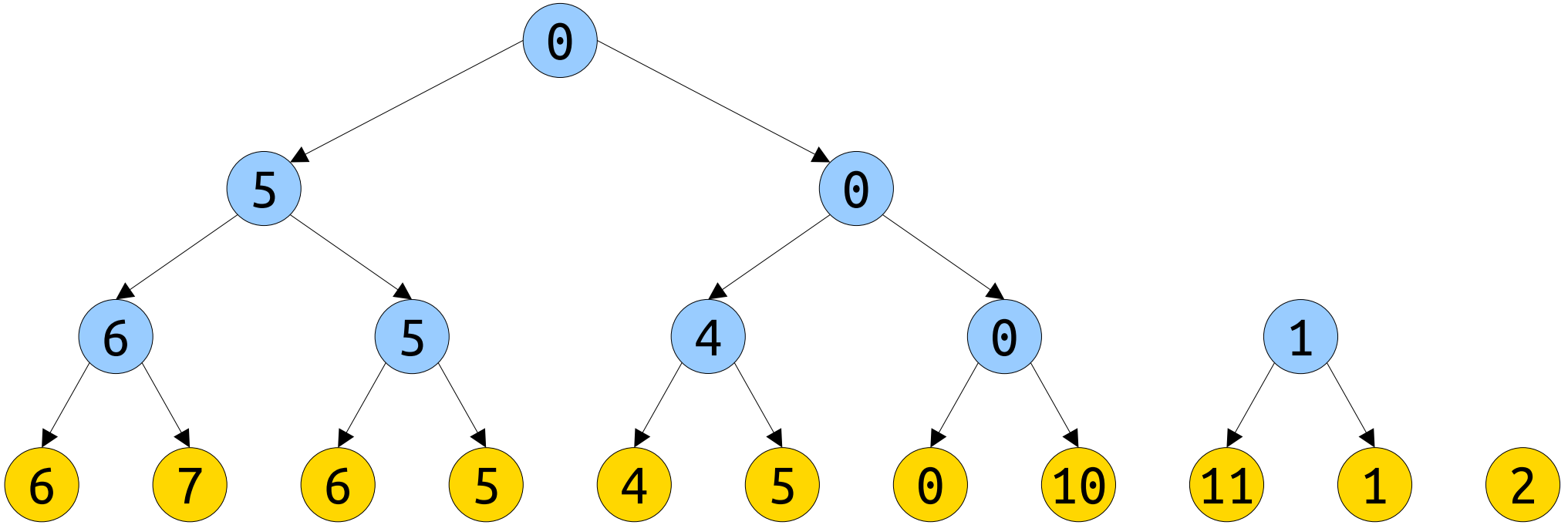
Tournament Heaps Revisited

- How might we implement ***decrease-key*** in a lazy tournament heap?
- ***Idea:*** When ***enqueue***-ing an element, hand out a pointer to the leaf corresponding to that element.
- Then, do do a ***decrease-key***, walk upwards from the leaf to the root of the tree, changing values as needed.



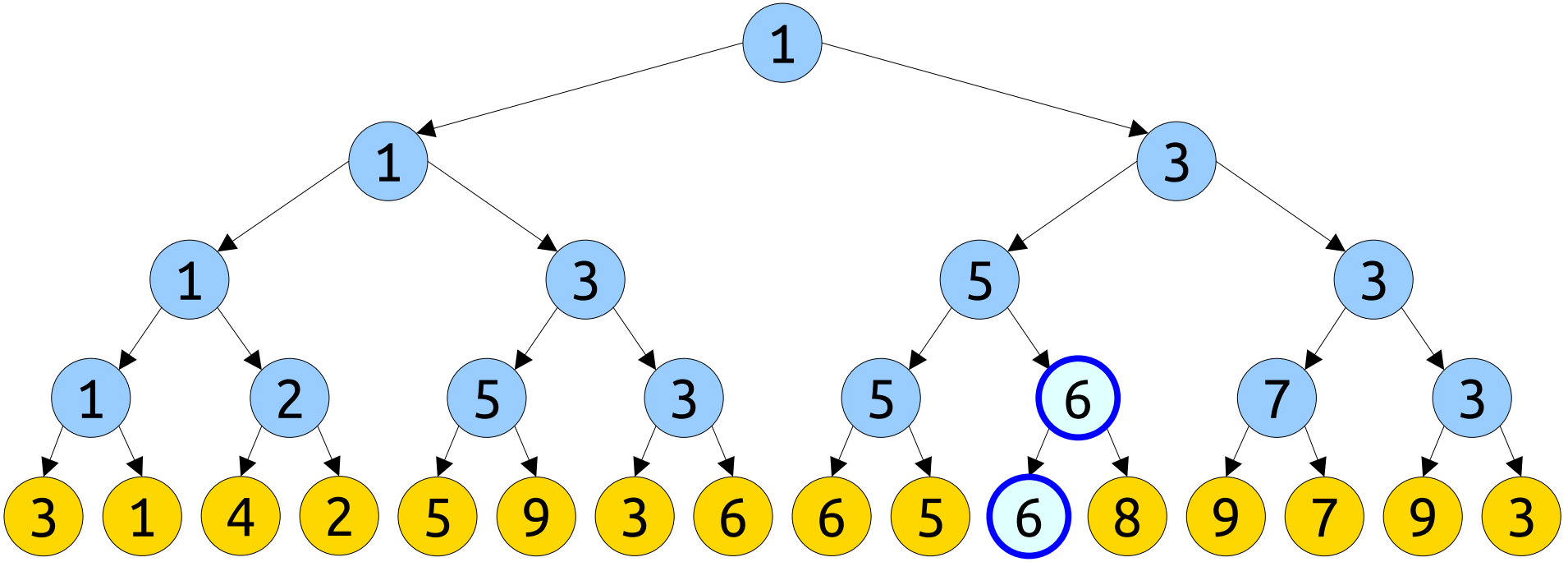
Tournament Heaps Revisited

- ***Claim:*** This process takes time $\Theta(\log n)$ in the worst case.
  - The tallest tree has height $\Theta(\log n)$.
- We want to speed this up to run in time O(1). Is that even possible?



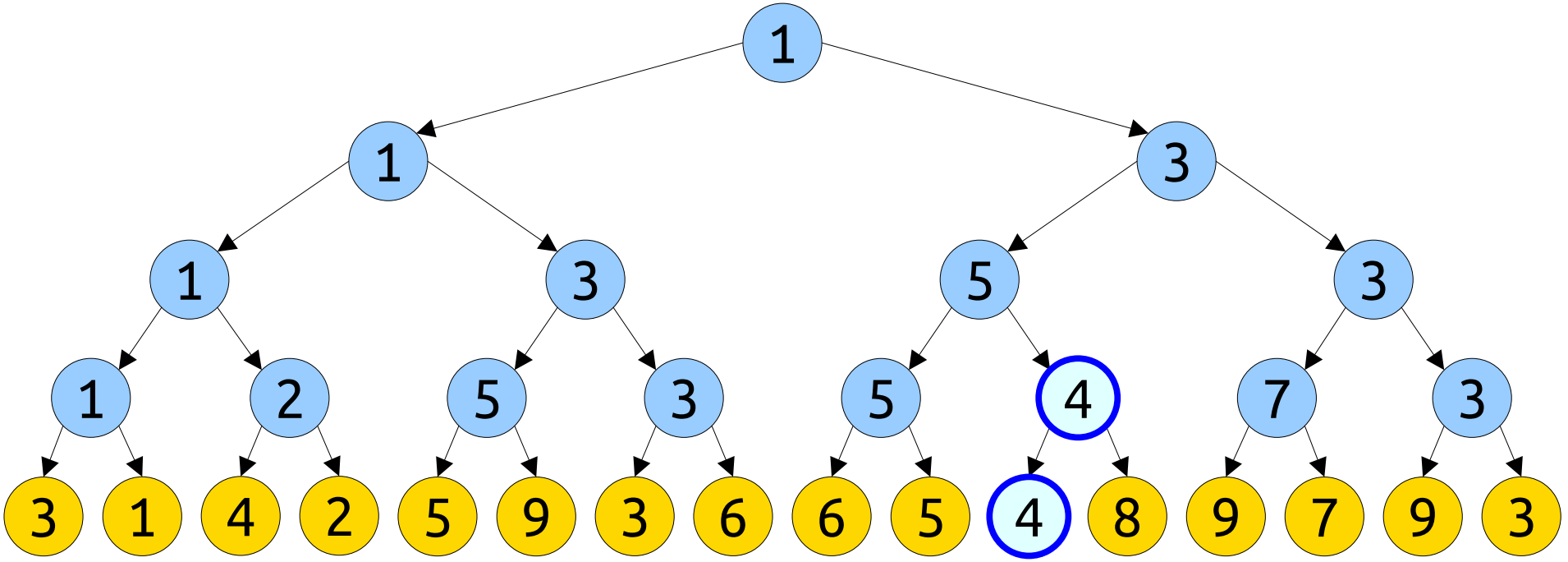
Speeding Things Up

- There are essentially two steps involved in updating the leaf-root path.
  - First, update nodes corresponding to the ***decrease-key***-ed element.



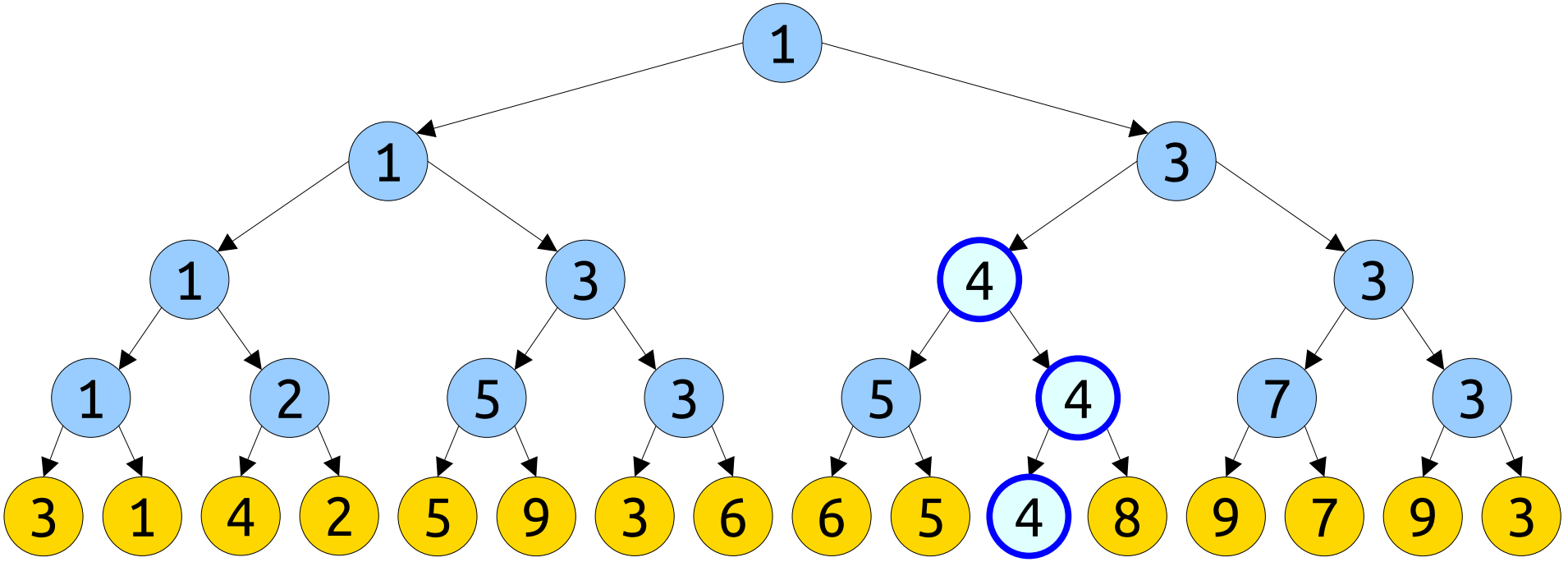
Speeding Things Up

- There are essentially two steps involved in updating the leaf-root path.
  - First, update nodes corresponding to the ***decrease-key***-ed element.



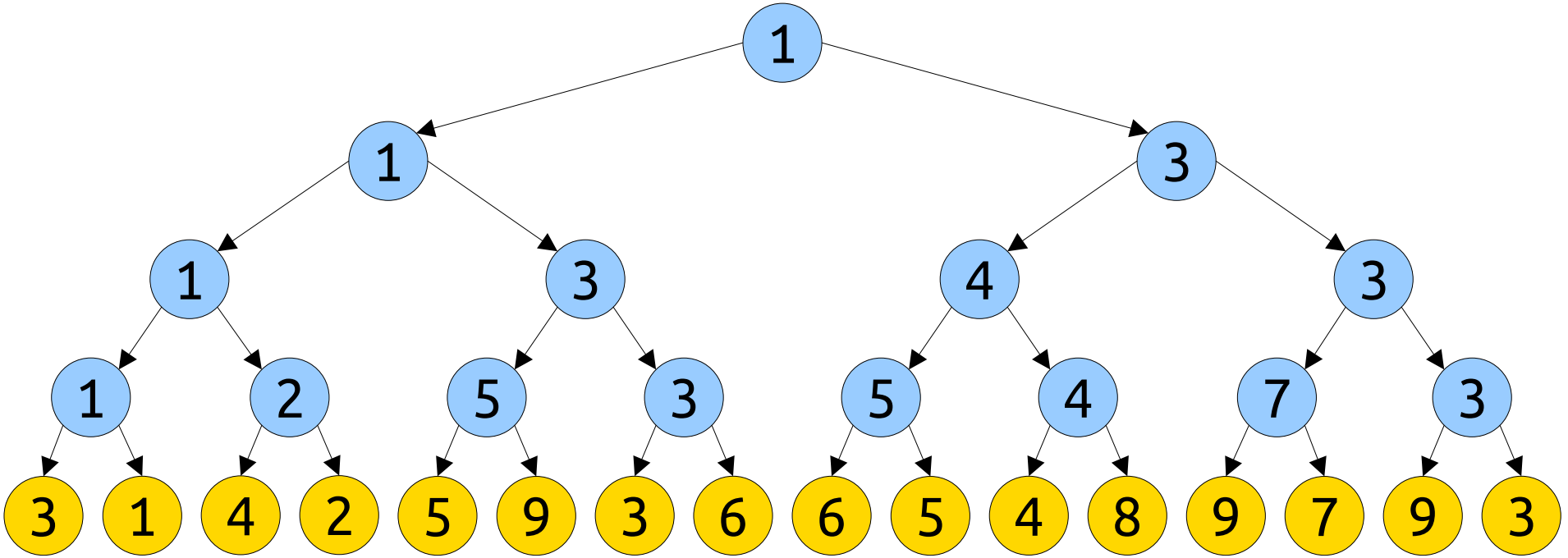
Speeding Things Up

- There are essentially two steps involved in updating the leaf-root path.
  - First, update nodes corresponding to the ***decrease-key***-ed element.
  - Next, update nodes corresponding to other elements.



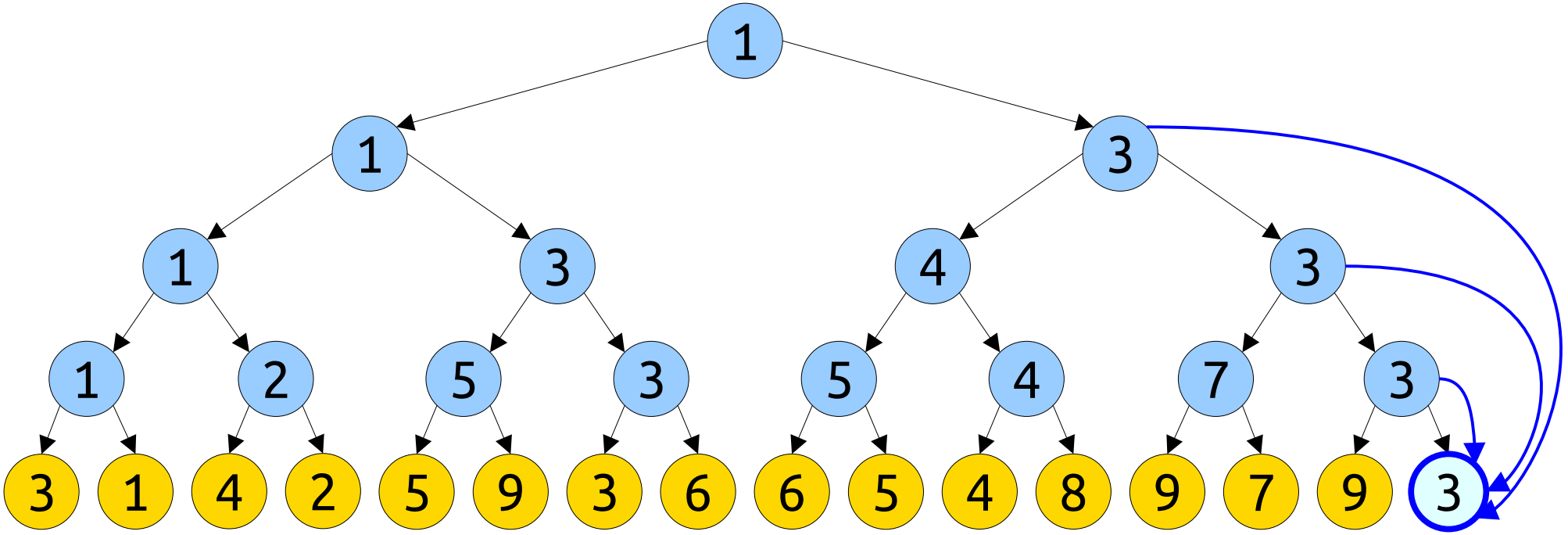
Speeding Things Up

- There are essentially two steps involved in updating the leaf-root path.
  - First, update nodes corresponding to the ***decrease-key***-ed element.
  - Next, update nodes corresponding to other elements.
- Let's address each of these in turn.



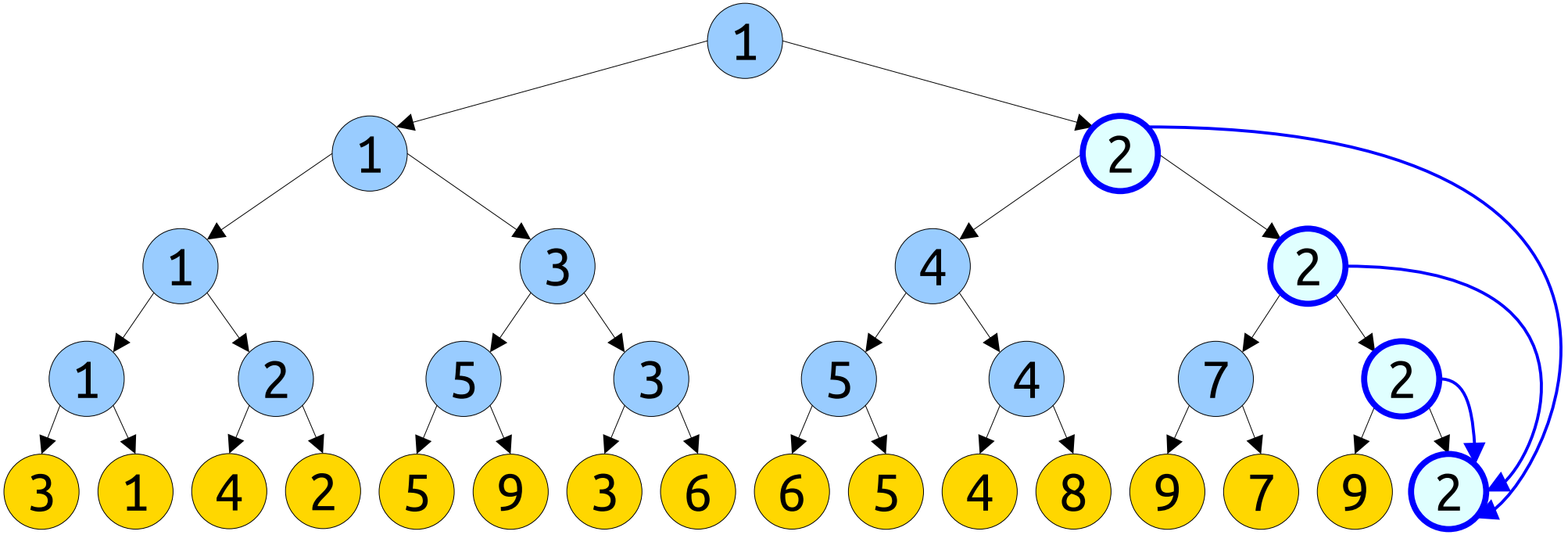
Speeding Things Up

- ***Optimization 1:*** Have each internal node store a pointer to the leaf it corresponds to.
- This lets us (implicitly) update the key of each node on the access path corresponding to the ***decrease-key***-ed item in time O(1) by just updating one spot.



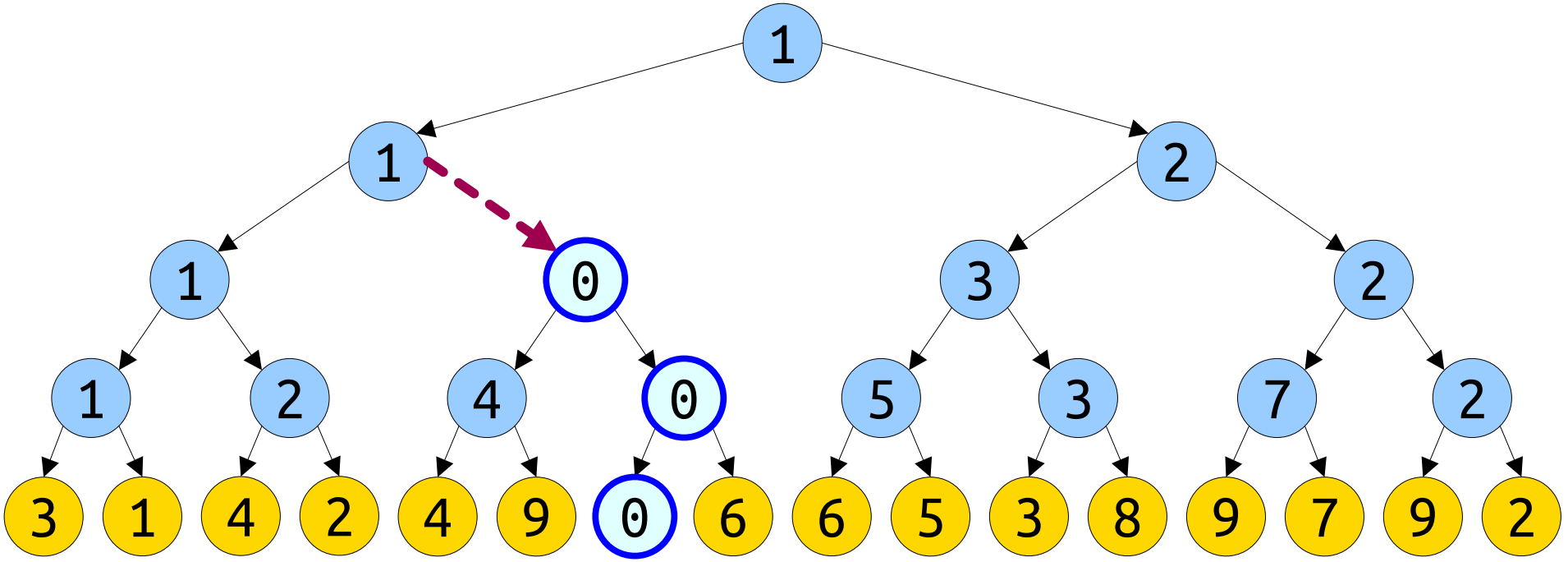
Speeding Things Up

- ***Optimization 1:*** Have each internal node store a pointer to the leaf it corresponds to.
- This lets us (implicitly) update the key of each node on the access path corresponding to the ***decrease-key***-ed item in time O(1) by just updating one spot.



Speeding Things Up

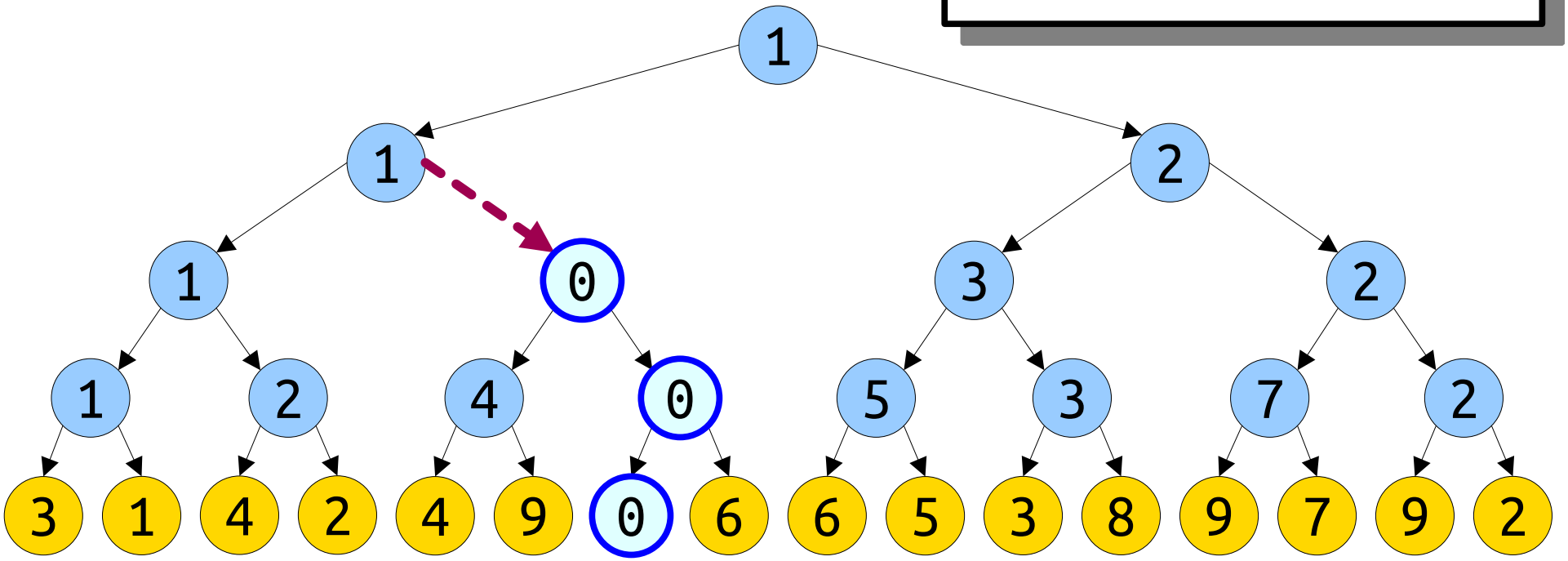
- After updating these copied nodes, one of two conditions holds.
  - ***Case 1:*** We're still left with a tournament tree. In that case, we're done.
  - ***Case 2:*** We've broken the min-heap property and no longer have a tournament tree. We need to do something to fix this.
- ***Fundamental Challenge:*** Updating the tournament tree above this point can take $\Theta(\log n)$ time in the worst-case.



Speeding Things Up

- We are operating under several tight constraints:
  - ***decrease-key*** must run in time O(1).
  - Our trees must be heap-ordered.
  - Our trees must be perfect bina

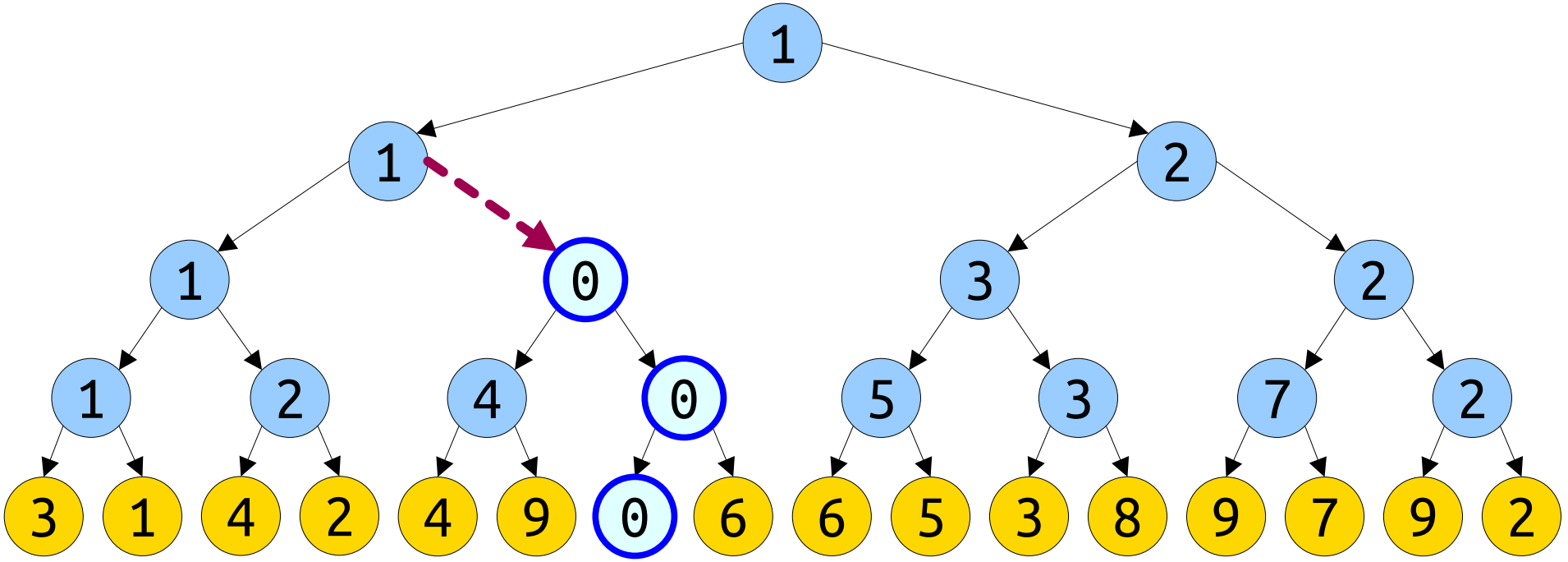
This is non-negotiable; it's the whole point of what we're trying to do.



Speeding Things Up

- We are operating under several
  - ***decrease-key*** must run in time O
  - Our trees must be heap-ordered.
  - Our trees must be perfect binary trees.

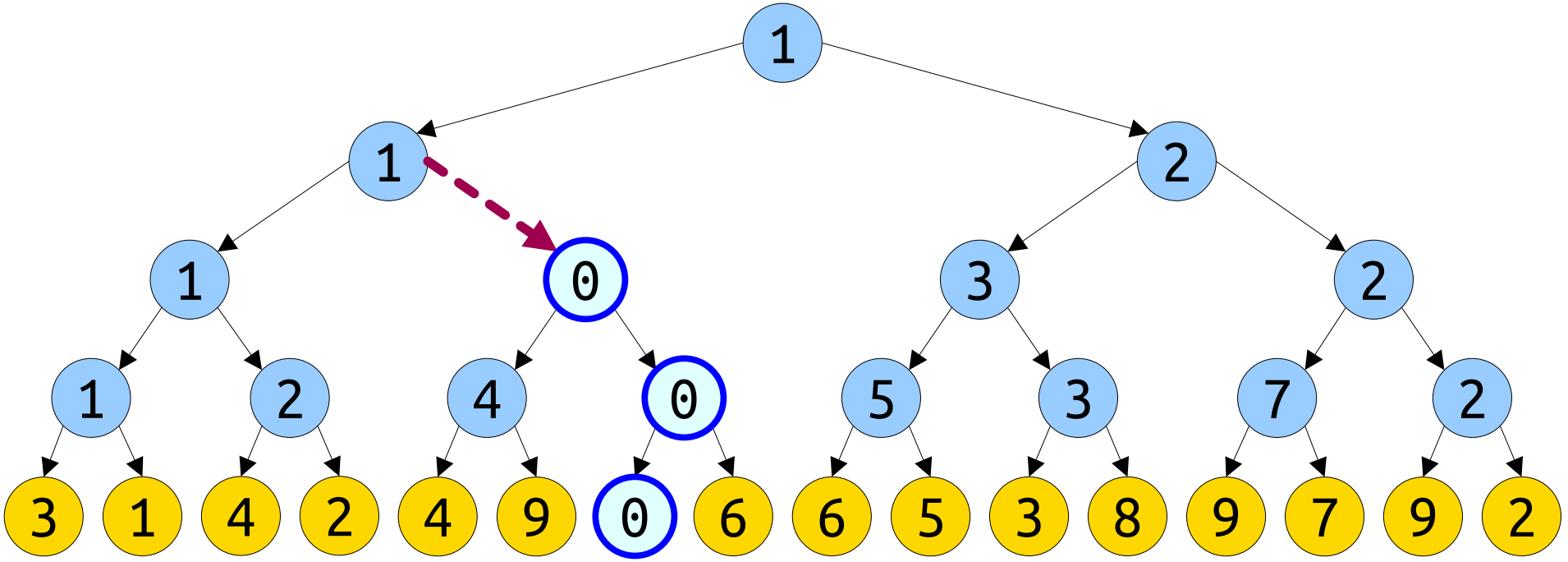
If we break the heap property, it's going to be hard to find the minimum element.



Speeding Things Up

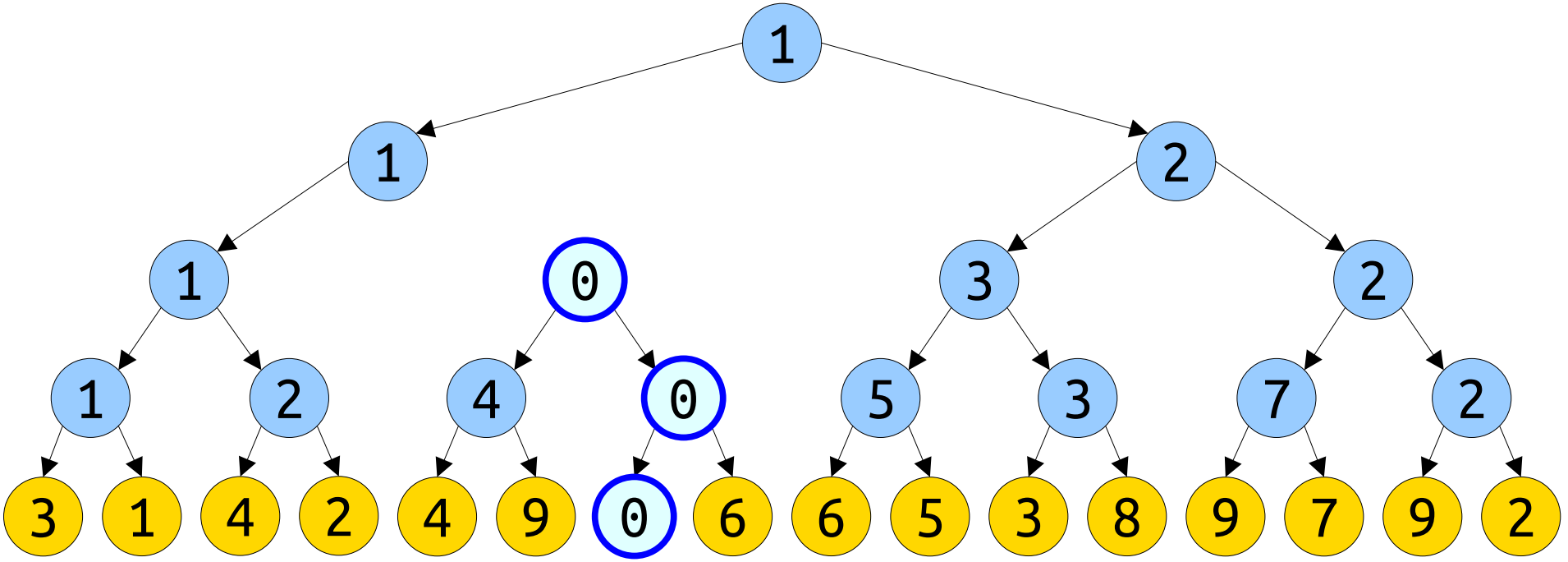
- We are operating under several tight constraints:
  - ***decrease-key*** must run in time O(1)
  - Our trees must be heap-ordered.
  - Our trees must be perfect binary trees.

Could we relax this?



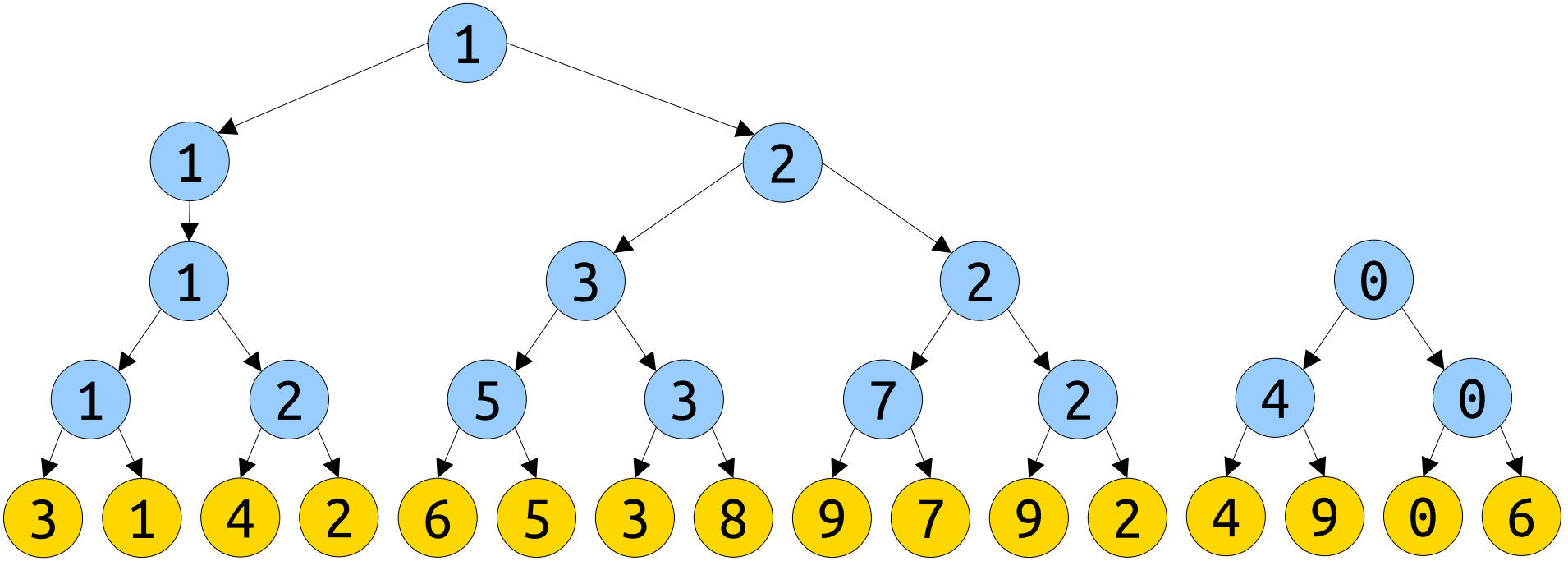
Speeding Things Up

- ***Key idea:*** When the heap property is violated, fix the issue by cutting the highest copy of the element from its parent.
- The resulting trees are heap-ordered, but aren't necessarily perfect binary trees.



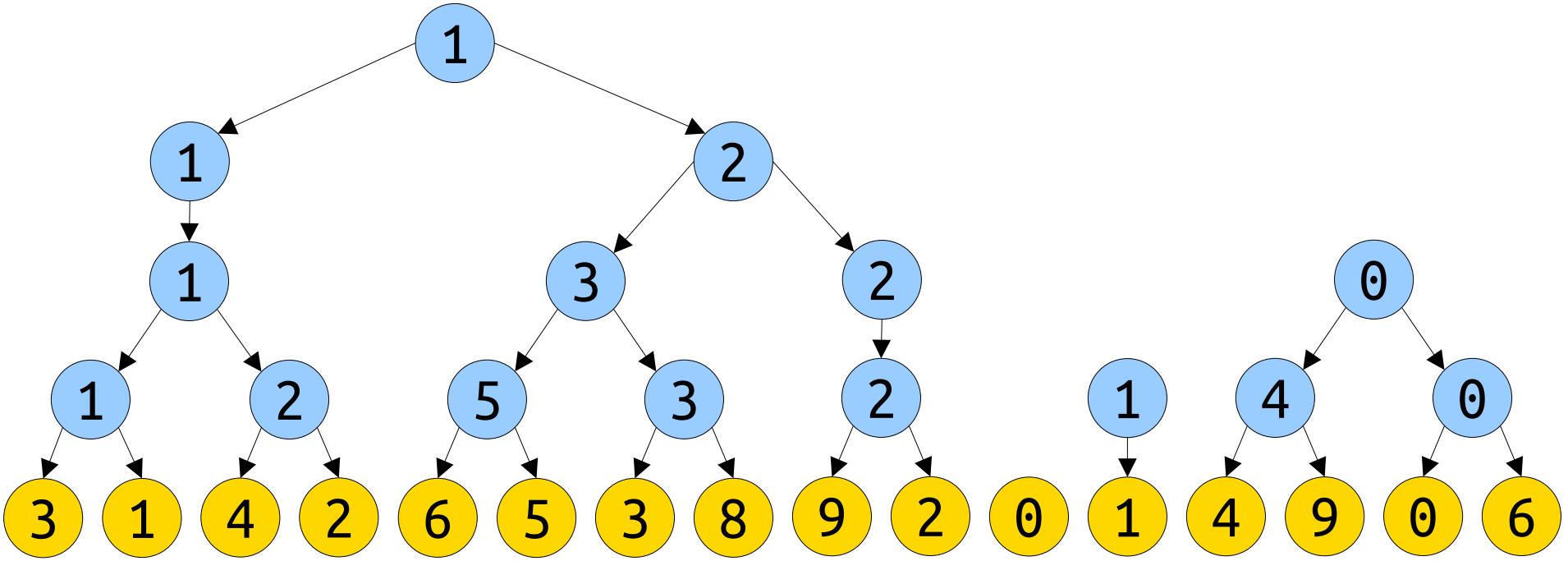
Speeding Things Up

- ***Key idea:*** When the heap property is violated, fix the issue by cutting the highest copy of the element from its parent.
- The resulting trees are heap-ordered, but aren't necessarily perfect binary trees.



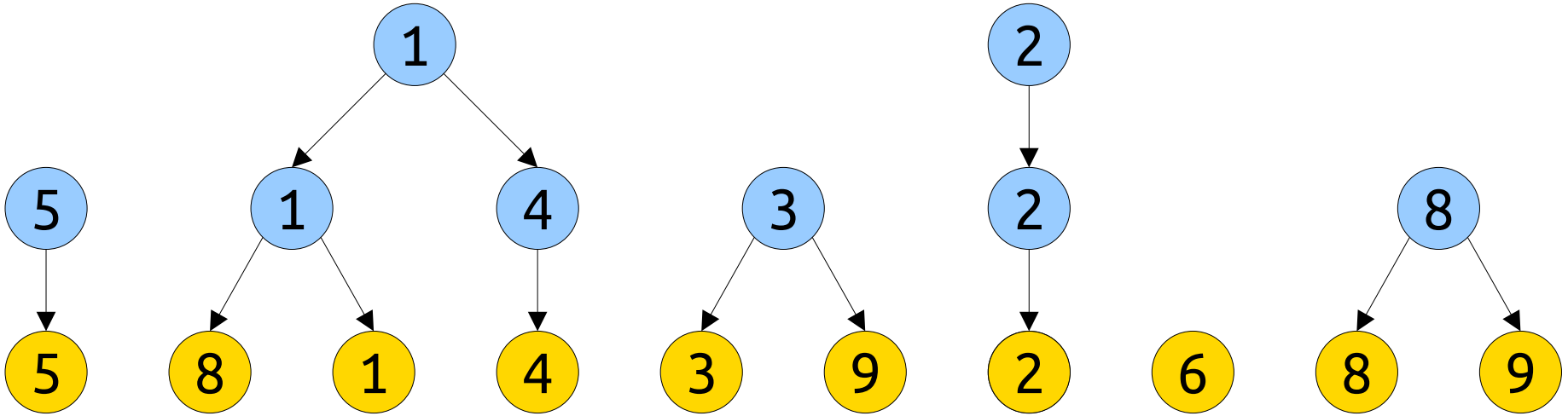
Assessing the Damage

- We've just made a fundamental change to lazy tournament heaps to support ***decrease-key***.
- Will our previous operations (***meld***, ***enqueue***, and ***extract-min***) still work?
- Good news: the answer is ***yes***!



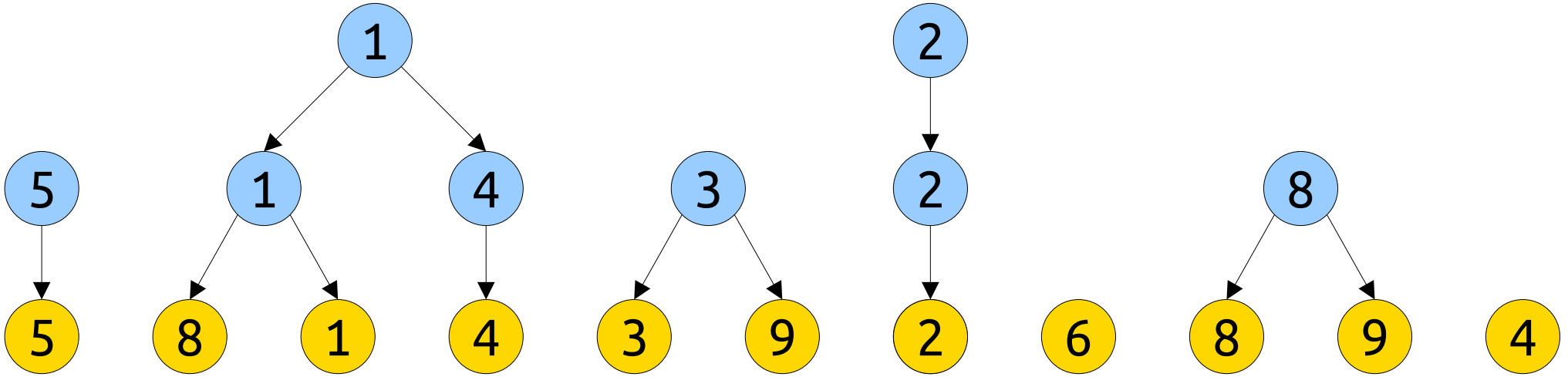
Assessing the Damage

- Our ***meld*** operation works by combining two groups of trees together.
- This approach is agnostic to the shapes of the trees, so it still works just fine.



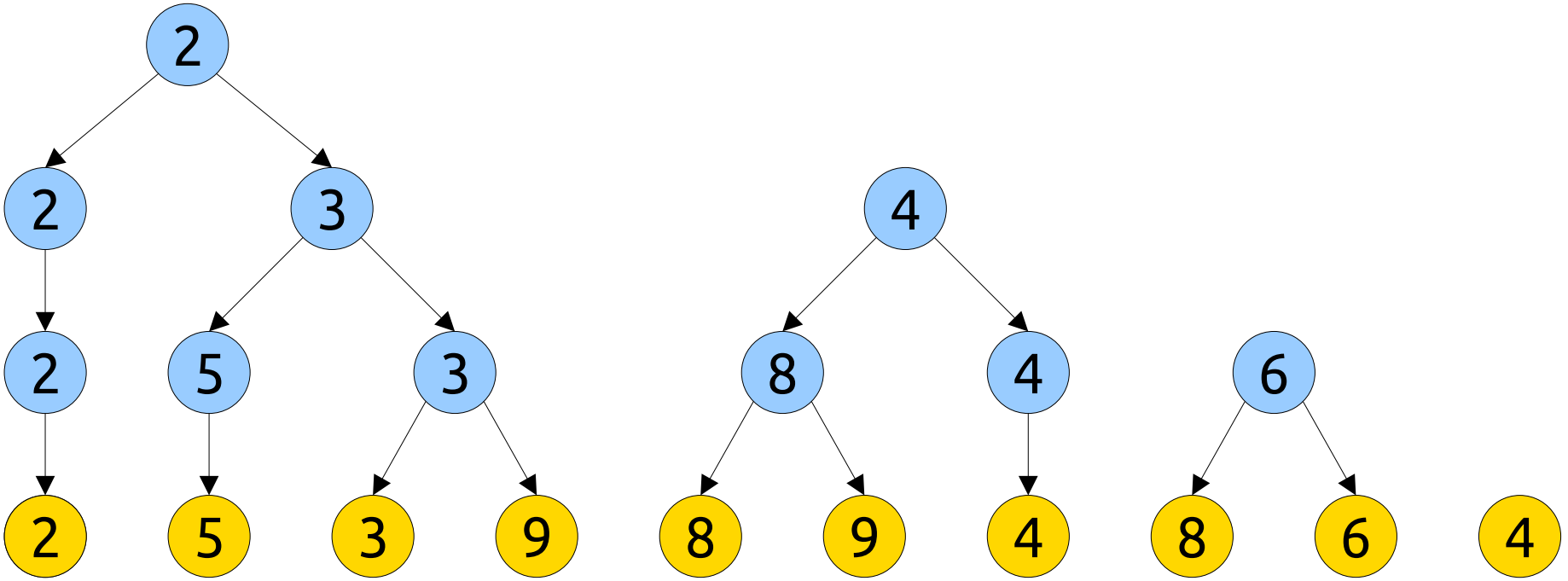
Assessing the Damage

- Our ***enqueue*** operation creates a new singleton node, then ***meld***s it in.
- That's still fine as well.



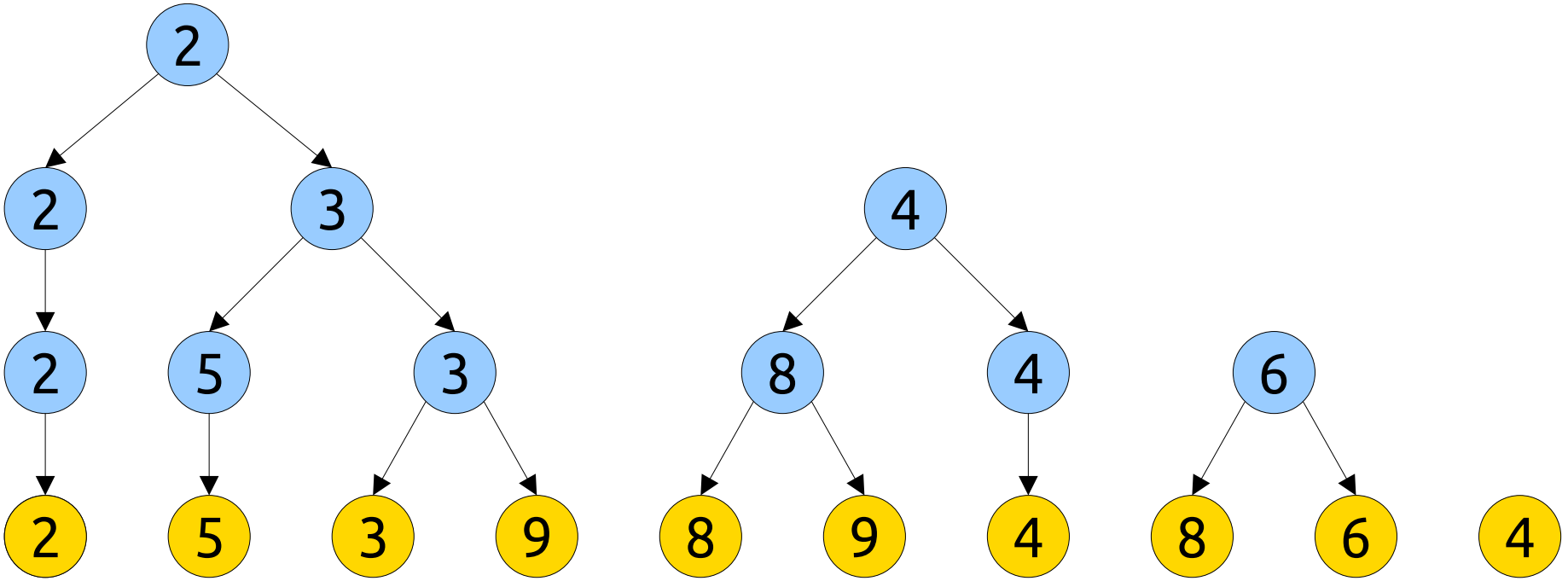
Assessing the Damage

- Each individual step in ***extract-min*** still works.
  - Find the tree with the smallest root.
  - Delete the root-leaf path corresponding to the minimum.
  - ***meld*** the newly-formed trees back into the main list of trees.
  - ***coalesce*** trees of the same height together.



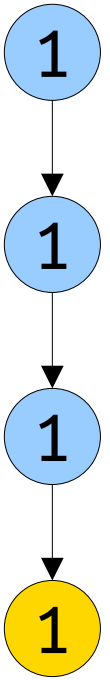
Assessing the Damage

- So… we're done?
- Unfortunately, no.
- Each operation still works, but ***extract-min*** no longer runs in (amortized) time O(log *n*).
- The reason is a bit subtle. Let's explore why.



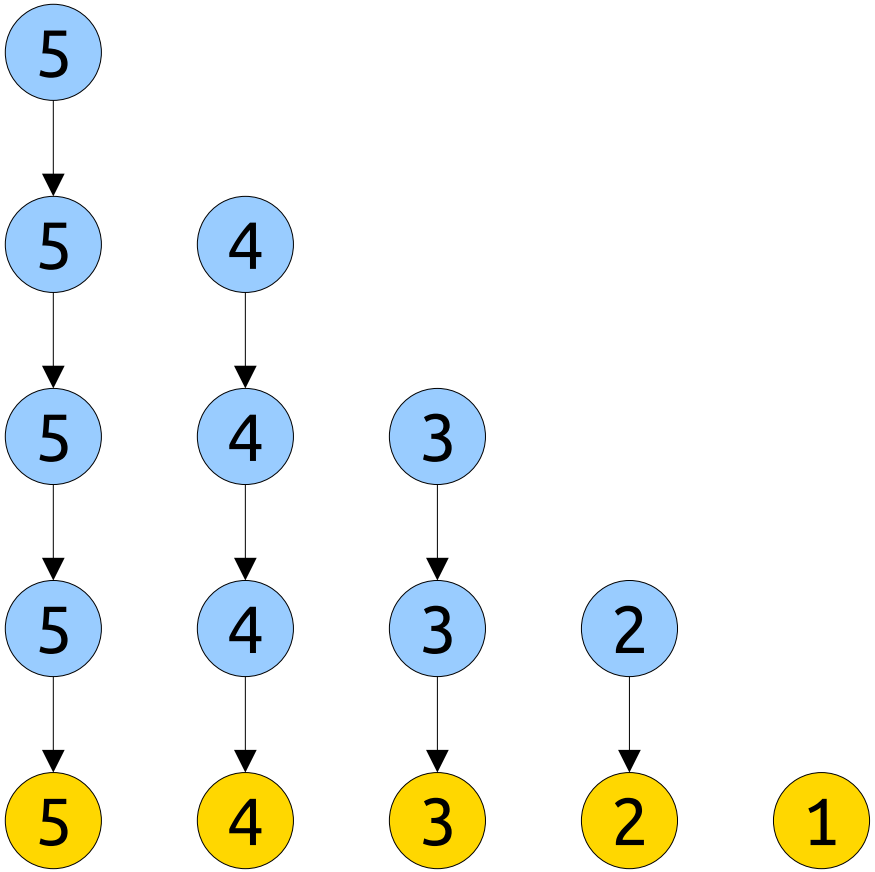
Exploring Tree Shapes

- Our ***decrease-key*** operation allows us to produce trees with unusual shapes.
- ***Claim:*** For any $k \geq 1$, we can produce a tree that's a chain of k copies of the same value.
- ***Corollary:*** There's no upper limit to the heights of the trees in our modified heap.



Exploring Tree Shapes

- The ***coalesce*** step of an ***extract-min*** links together trees until there's only one tree of each height.
- However, if the tree heights can be arbitrarily large, then running a ***coalesce*** might not actually compact any trees together.
- As a result, we can build heaps where each ***extract-min*** takes time $\Theta(n)$ without reducing the number of trees, meaning we can't backcharge the work.
- So ***extract-min*** still works correctly, but no longer runs in (amortized) time $O(\log n)$.
- How do we fix this?



Before and After

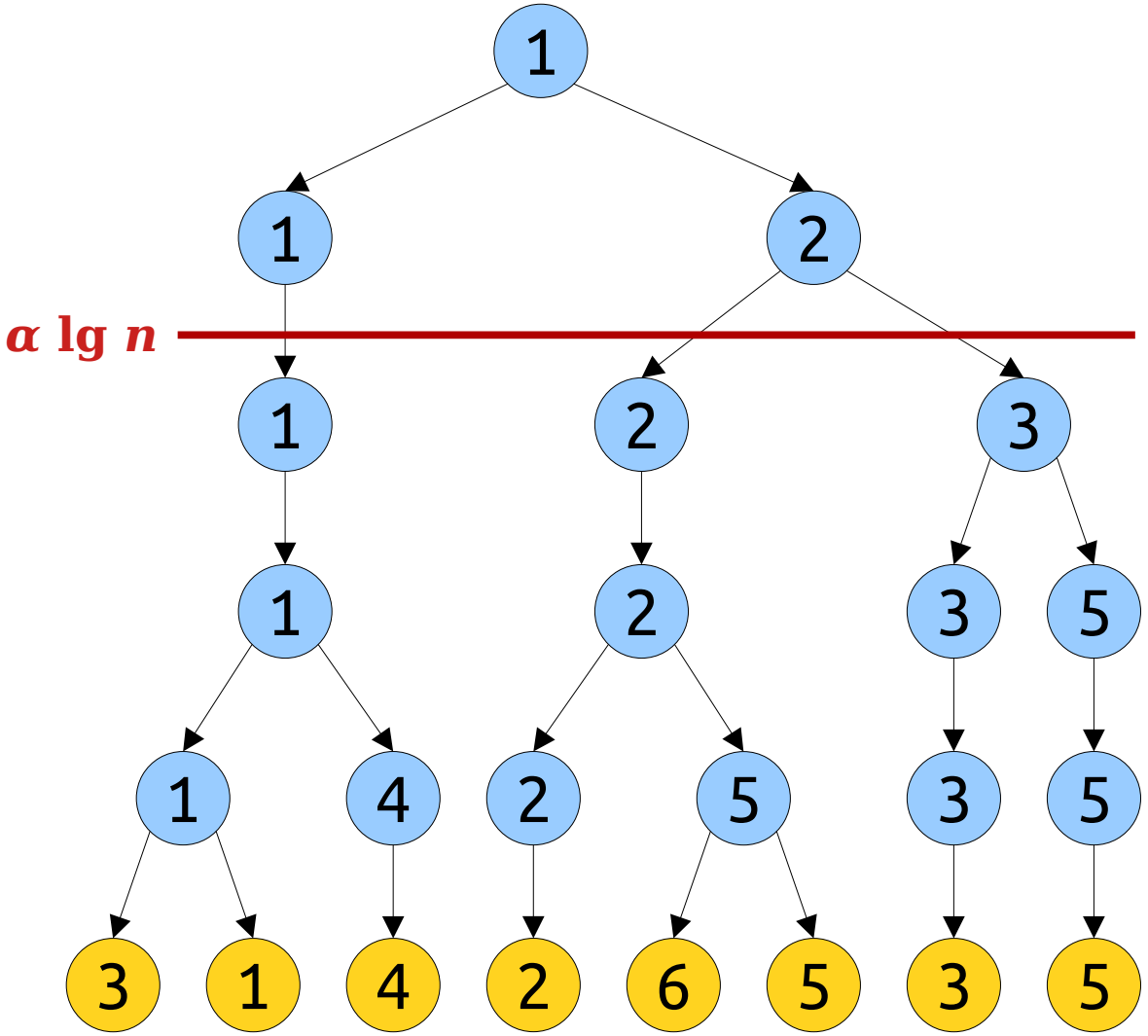
- Before we allowed for ***decrease-key***, each of our trees was a perfect binary tree.
- This meant that the maximum height of any tree in our heap is lg n, which is why ***extract-min*** ran quickly.
- If we're too strict with our tree shapes, we can't get ***decrease-key*** to run quickly enough.
- ***Goal:*** Find a way to keep tree heights low without placing too many structural constraints on the tree shape.
- ***Question:*** Where have we seen something like this?

The Scapegoat Strategy

- ***Idea:*** Pick some constant $\alpha > 1$ and enforce a height limit of $\alpha \lg n$ across all trees in the heap.
  - Here, n denotes the total number of elements in the heap, not the number of nodes in any one tree.
- Intuitively, keeping the heights of the trees low will make all operations run quickly.
- As long as the tree heights obey this rule, we don't need to do anything fancy to get our heap to run quickly.
- Once the rule is violated, we need to do something to restore the $\alpha \lg n$ max height.

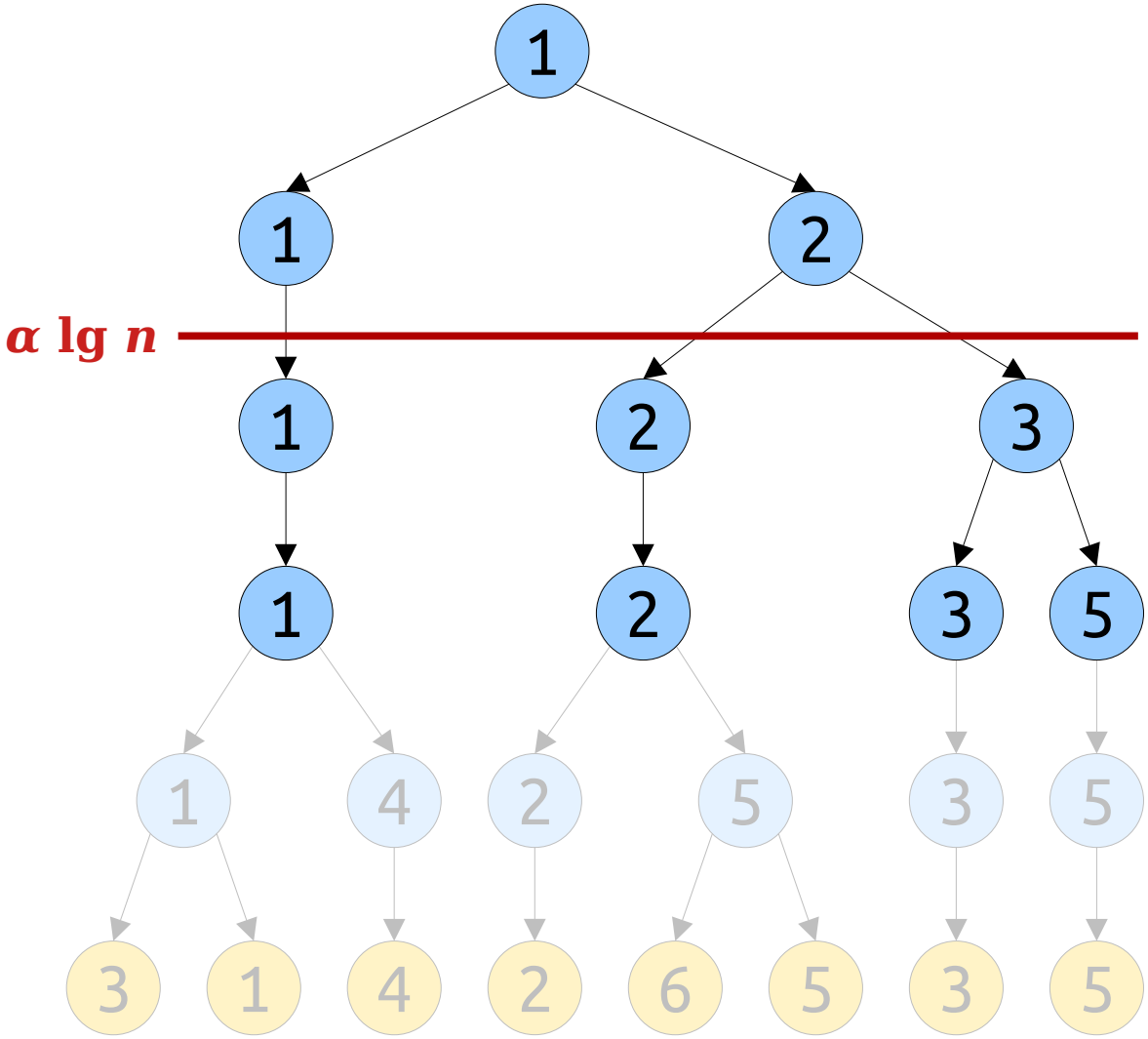
The Scapegoat Strategy

- Suppose that at the end of an ***extract-min*** operation we end up with a tree whose height exceeds $\alpha \lg n$.
- We need to reshape that tree to reduce its height below $\alpha \lg n$.



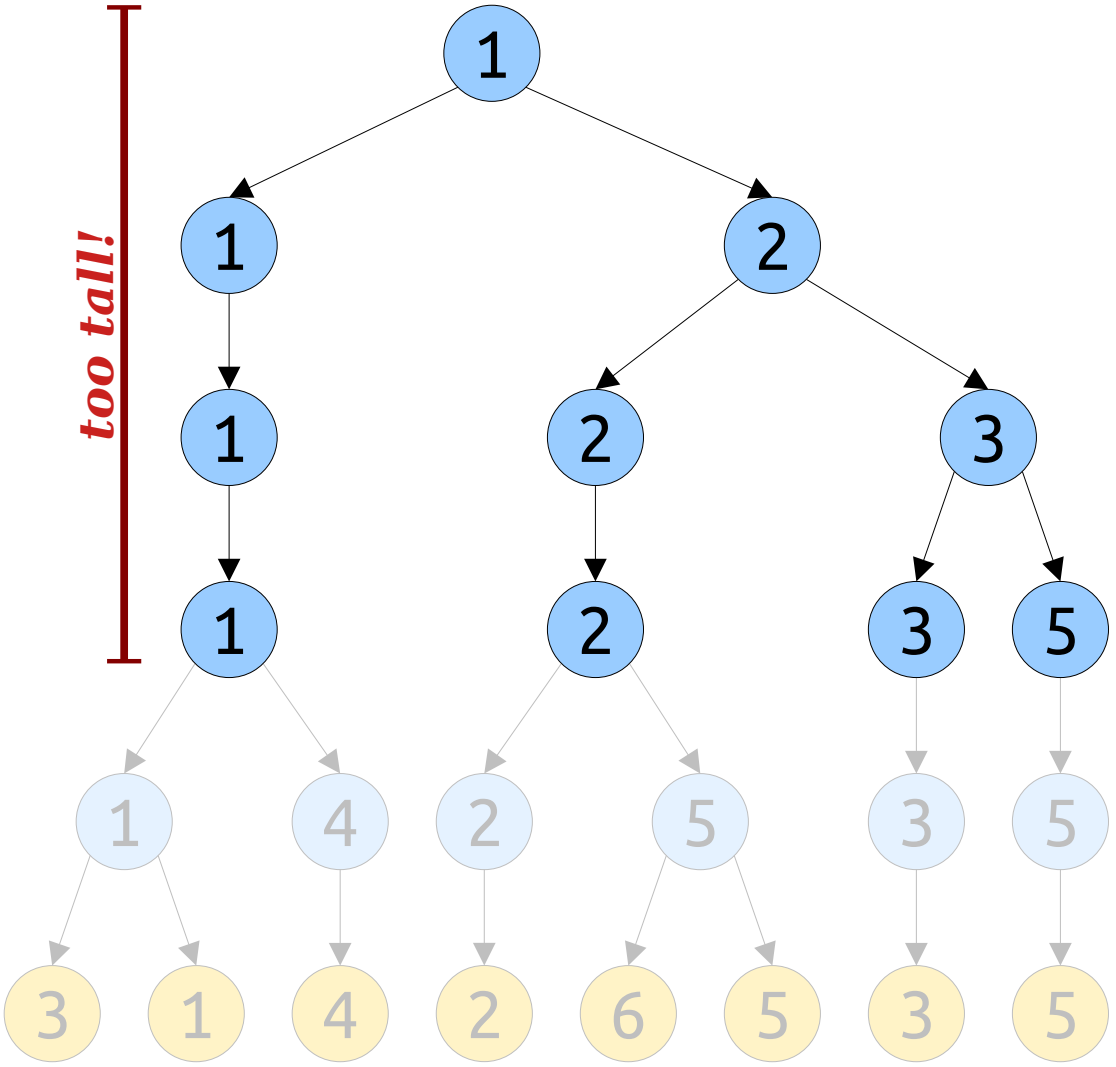
The Scapegoat Strategy

- Call a tree with L leaves ***α-balanced*** if its height is at most $\alpha \lg L$.
- If we discover a tree whose height exceeds $\alpha \lg n$, then the whole tree is not α-balanced.
- However, the top layer of the tree by itself must be α-balanced.
- Therefore, there is some highest layer in the tree whose corresponding subtree is not α-balanced.



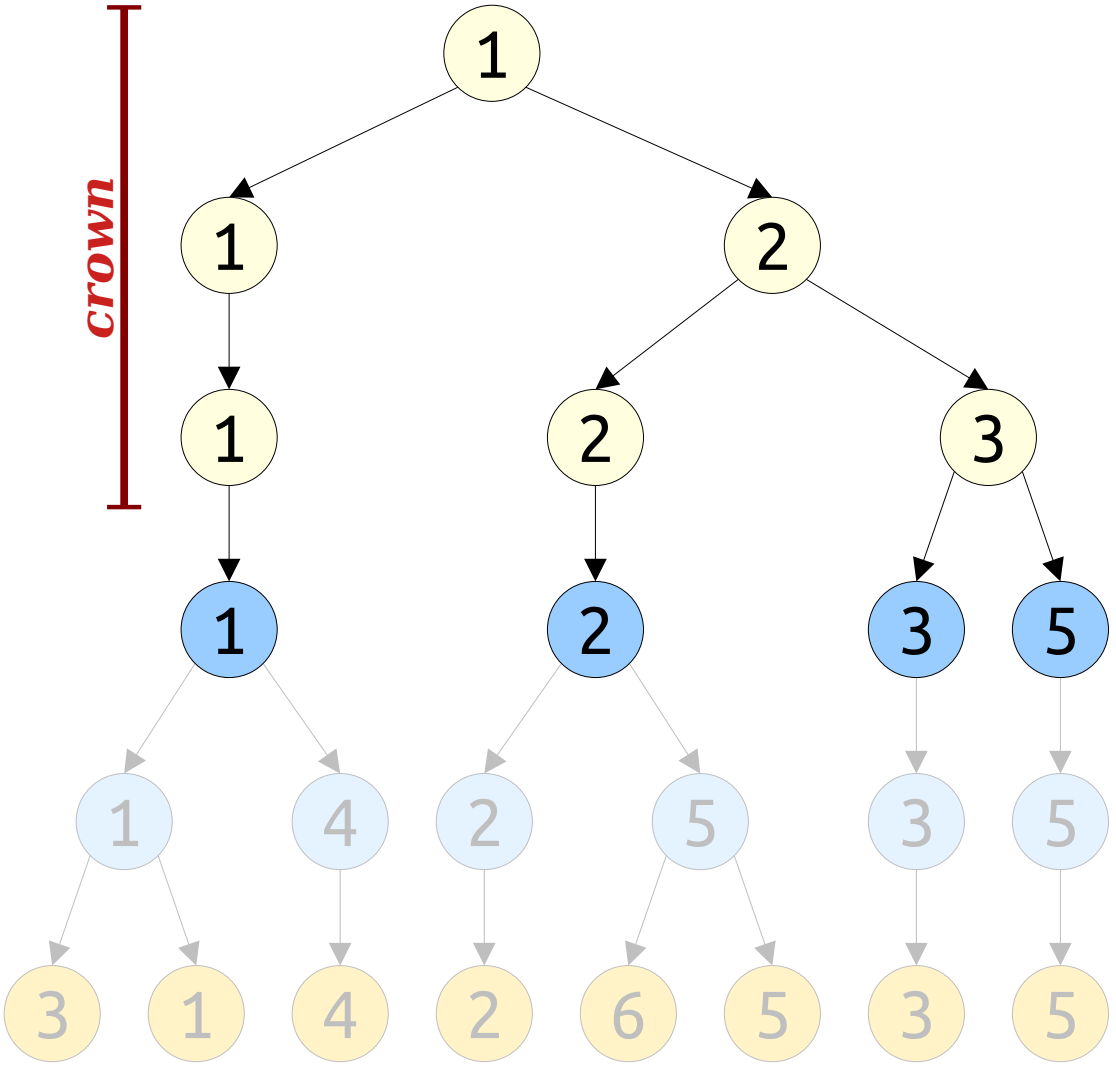
The Scapegoat Strategy

- Call a tree with L leaves ***α-balanced*** if its height is at most $\alpha \lg L$.
- If we discover a tree whose height exceeds $\alpha \lg n$, then the whole tree is not α-balanced.
- However, the top layer of the tree by itself must be α-balanced.
- Therefore, there is some highest layer in the tree whose corresponding subtree is not α-balanced.
- All nodes above this layer form our ***crown***.



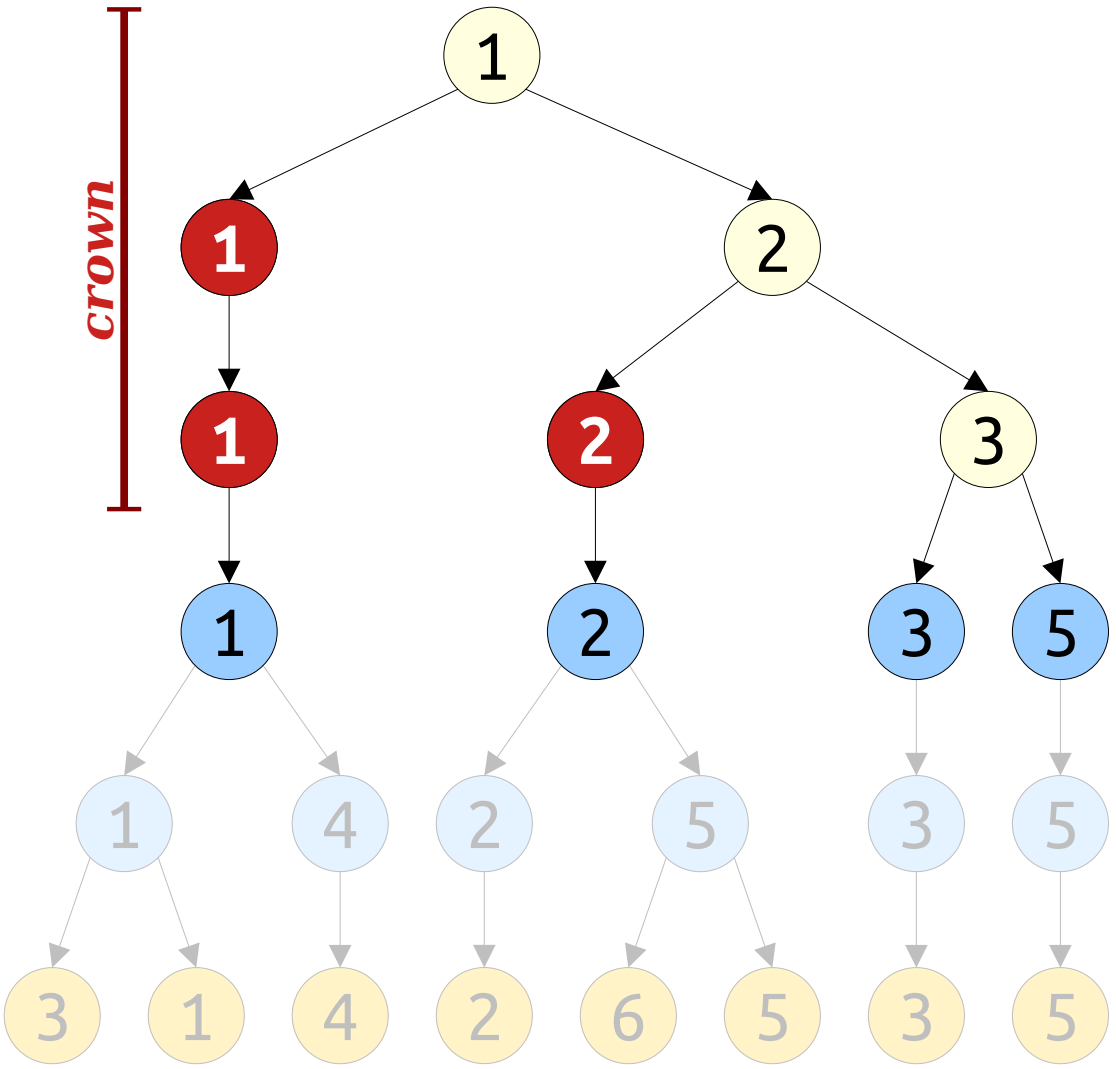
The Scapegoat Strategy

- Call a tree with L leaves ***α-balanced*** if its height is at most $\alpha \lg L$.
- If we discover a tree whose height exceeds $\alpha \lg n$, then the whole tree is not α-balanced.
- However, the top layer of the tree by itself must be α-balanced.
- Therefore, there is some highest layer in the tree whose corresponding subtree is not α-balanced.
- All nodes above this layer form our ***crown***.



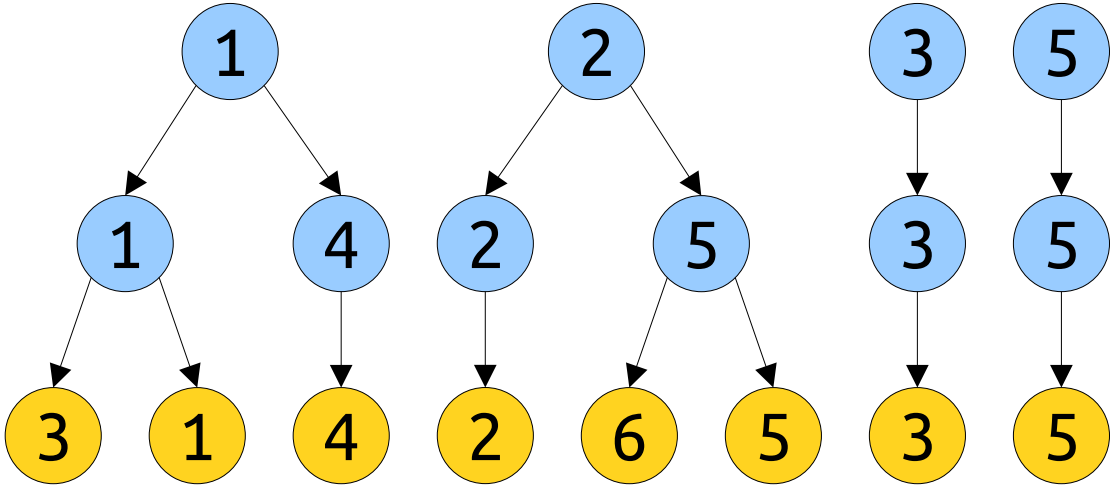
The Scapegoat Strategy

- Intuitively, this crown must consist of many ***bad nodes***, where a bad node is a node with one child.
- To remove those bad nodes, simply delete all nodes in the crown.



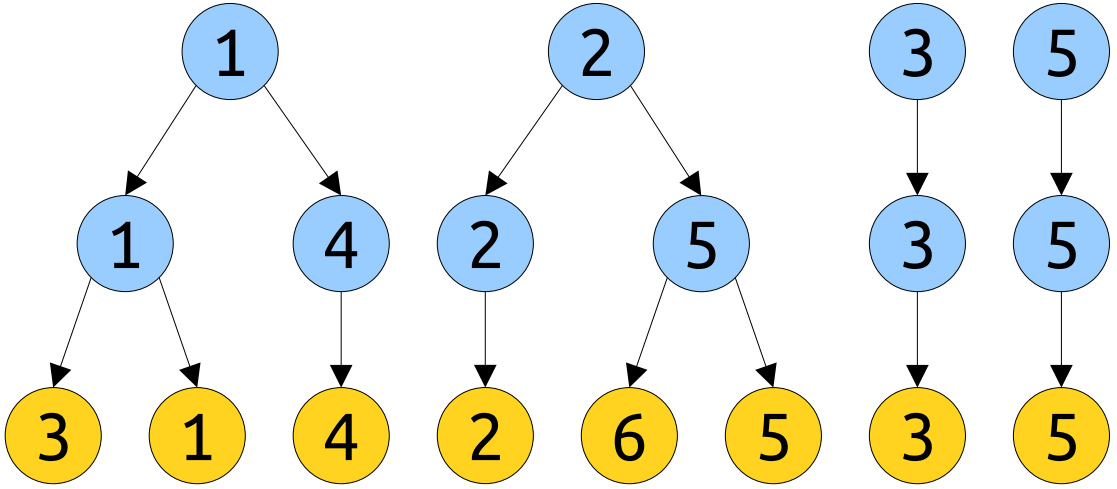
The Scapegoat Strategy

- Intuitively, this crown must consist of many ***bad nodes***, where a bad node is a node with one child.
- To remove those bad nodes, simply delete all nodes in the crown.
- This is guaranteed to drop the heights of the remaining trees down below $\alpha \lg n$. (Why?)



The Scapegoat Strategy

- Removing the crown restores balance and removes many bad nodes from the tree.
- It also increases the number of trees in the heap, making more work for future ***coalesce*** steps.
- However, we'll show that we can blame all this extra work on past ***decrease-key*** operations, and it'll all amortize away.
  - Intuitively, bad nodes accumulate slowly, then get cleaned up rapidly.



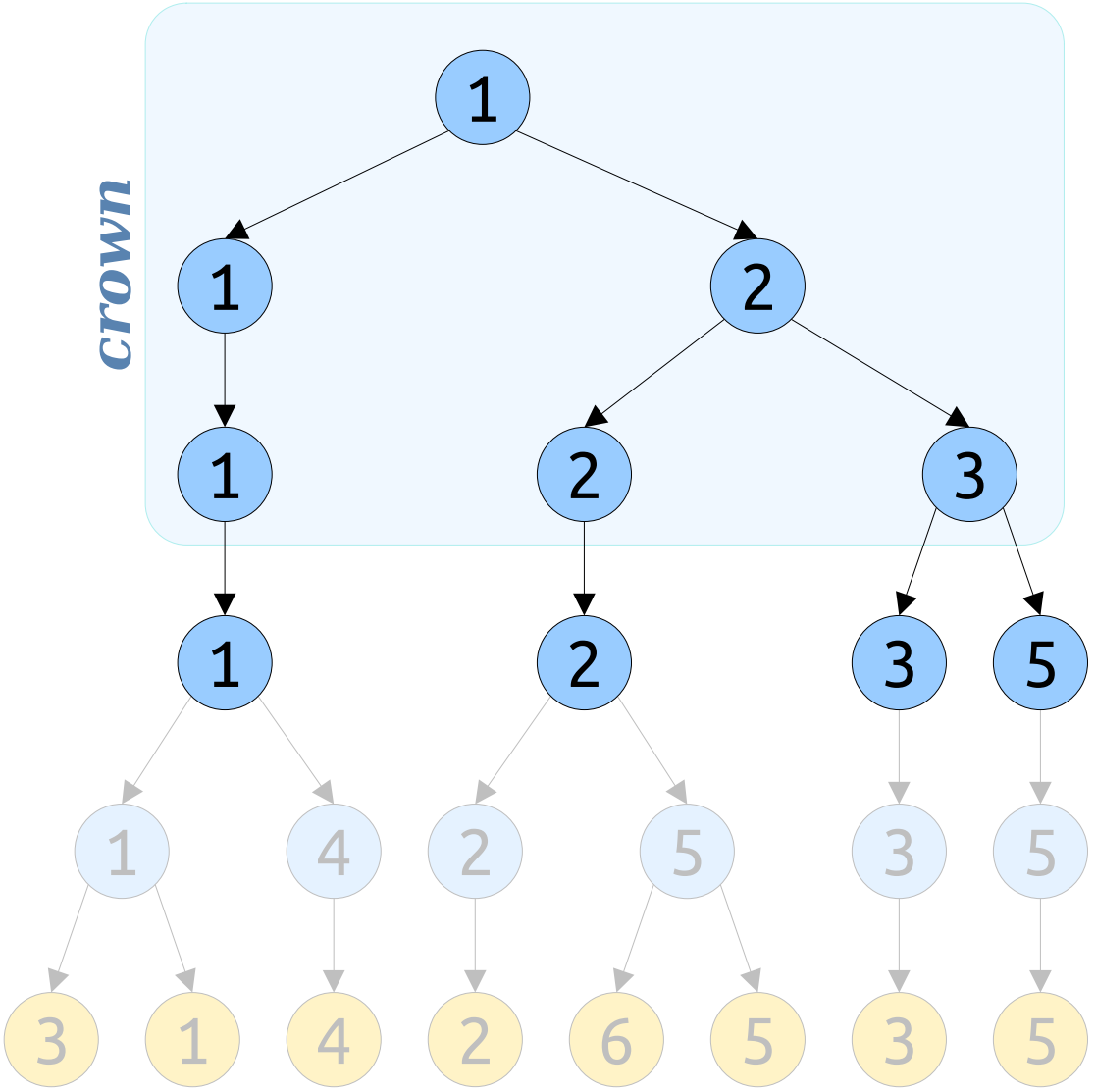
Abdication Heaps

- A ***abdication heap*** is a lazy tournament heap modified as follows:
  - We pick a constant $\alpha > 1$ and enforce that all trees have height at most $\alpha \lg n$.
  - To perform a ***decrease-key***, find the highest node containing a copy of the element, then cut it from its parent if it breaks the heap property.
  - After performing an ***extract-min***, for each tree whose height exceeds $\alpha \lg n$, find and remove its crown.
- The name comes from the last step.

How Fast is This?

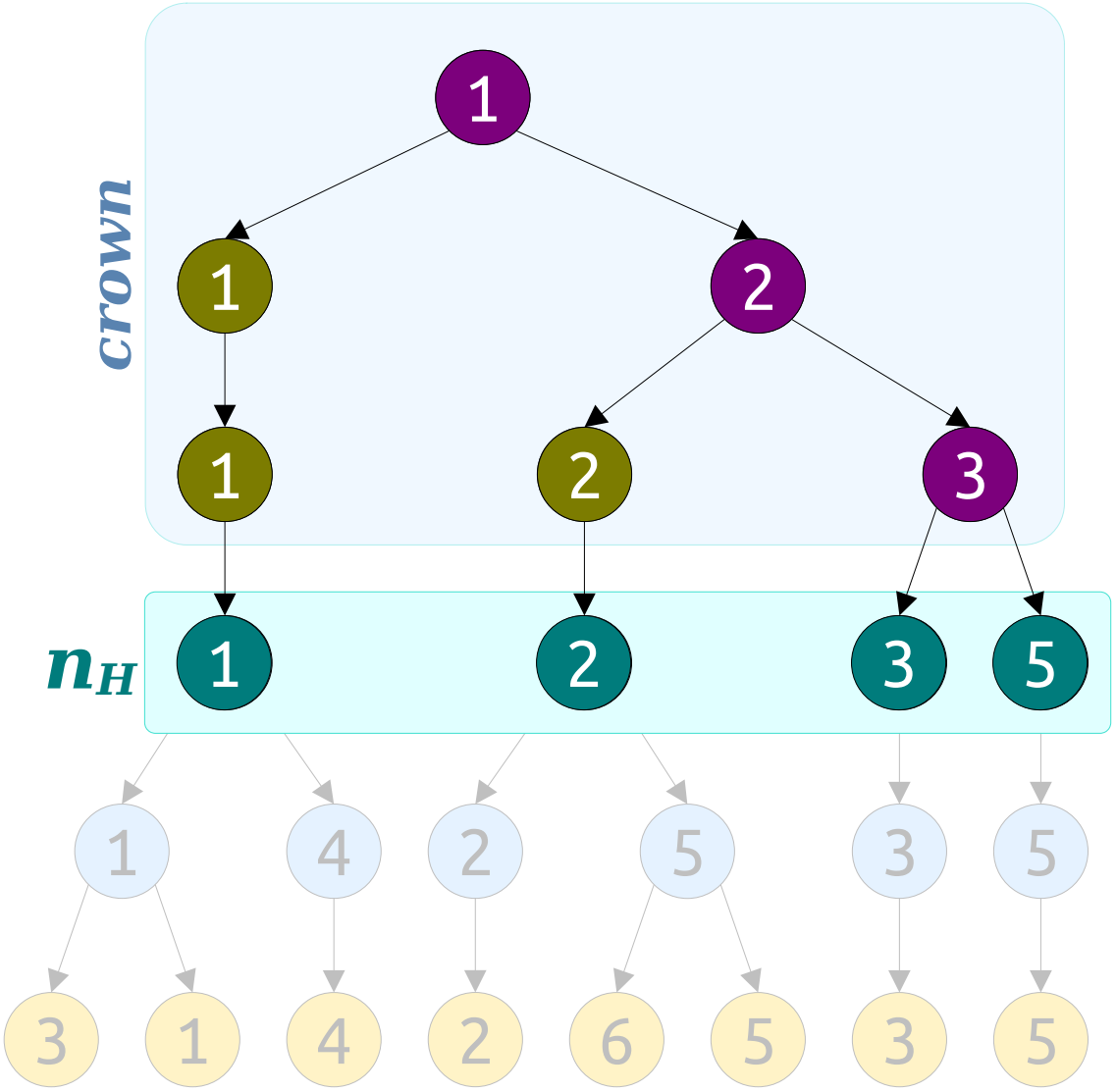
The Anatomy of the Crown

- Removing the crown from a tree requires work proportional to
  - the number of nodes in the crown, plus
  - the number of nodes one layer below the crown.
- ***Intuition:*** Some large fraction of these nodes must be bad.
  - Otherwise, the trees wouldn't exceed the $\alpha \lg n$ threshold.
- ***Goal:*** Bound the number of nodes in the crown, plus the layer below the crown, in terms of the number of bad nodes in the crown.



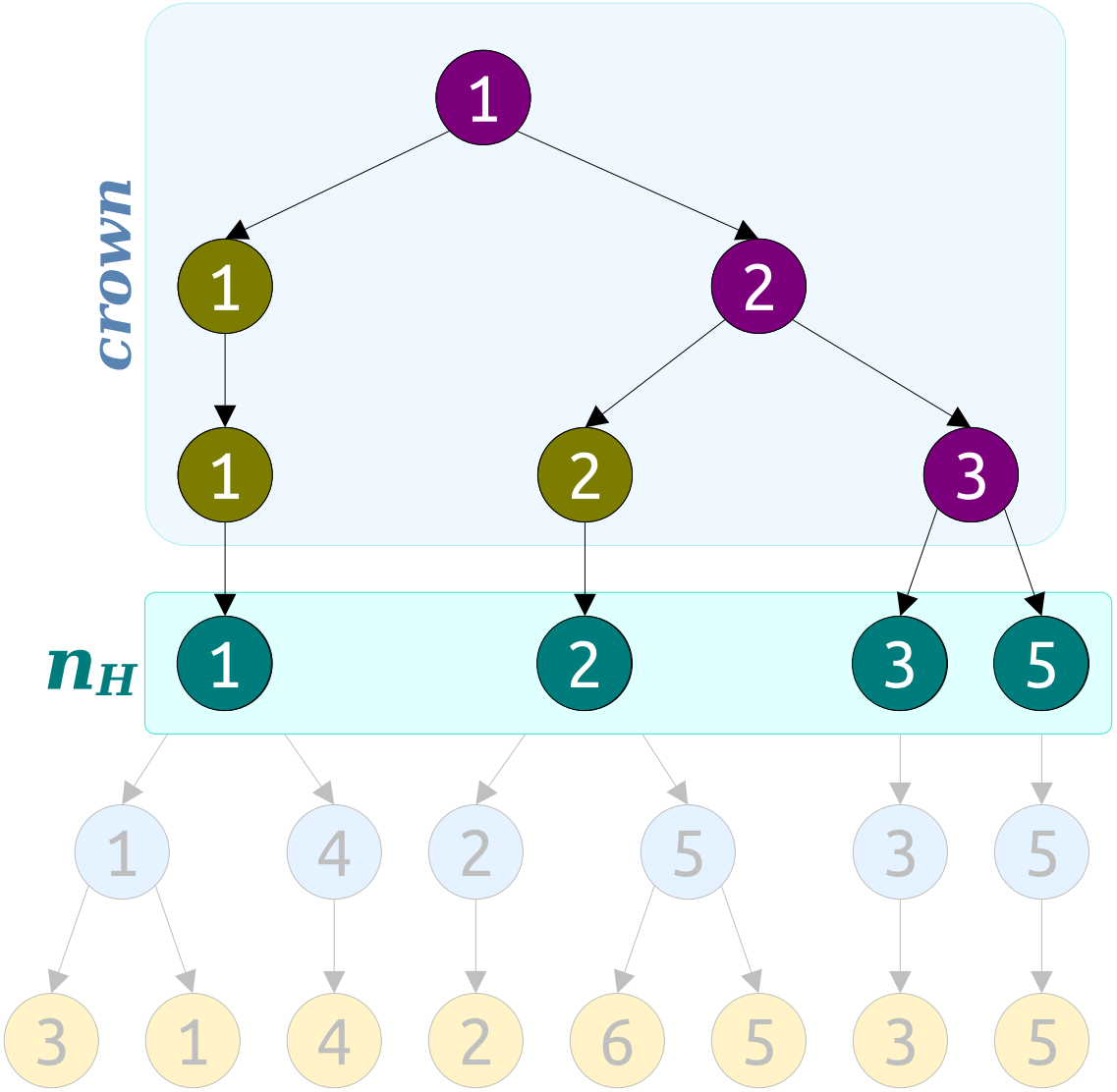
The Anatomy of the Crown

- Let's introduce some notation to make things easier.
- Let ***g*** be the number of good nodes in the crown.
- Let ***b*** be the number of bad nodes in the crown.
- Let n_H be the number of nodes in the layer below the crown.
- ***Goal:*** Bound ***g*** + ***b*** + n_H, the number of nodes scanned when removing a crown.



Bounding g

- ***Claim:*** $n_H - 1 = g$.
- ***Proof Sketch:***
  Initially, there is one node at the top layer of the tree. Each good node increases the number of nodes in the next level by one. Once we reach the layer below the crown, we have n_H nodes. So we must see exactly $n_H - 1$ good nodes.



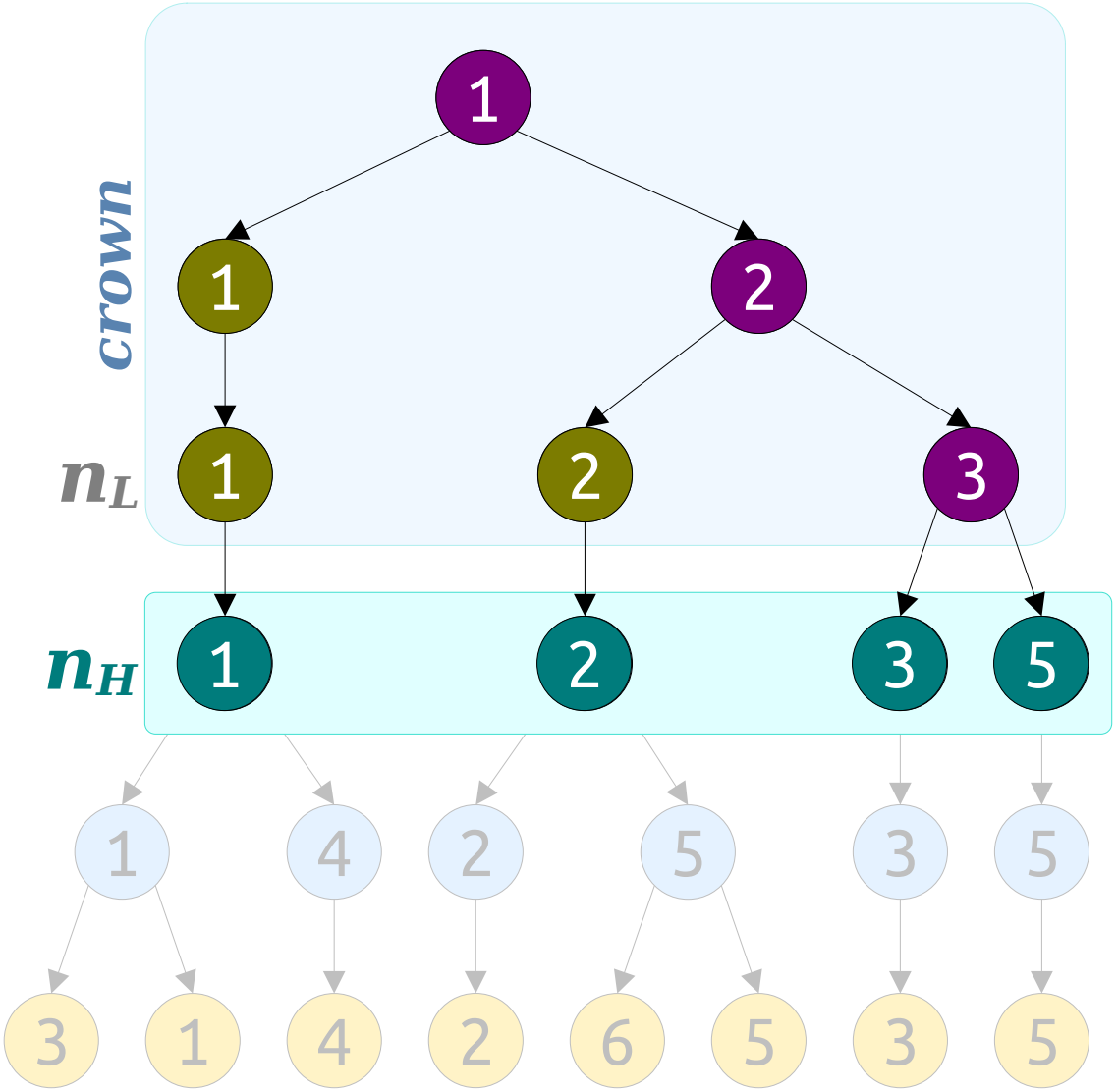
Bounding n_H

- Let n_L be the number of leaves in the crown.
- Each bad leaf contributes one node in the level below, and there are at most b bad leaves.
- Each good leaf contributes two nodes in the level below.
- This means that

$$n_H > 2n_L - b.$$

- Equivalently,

$$b > 2n_L - n_H.$$



Bounding n_H

- Suppose our crown has height h.
- Because the crown, plus the layer below, isn't α-balanced, we know that

$$h + 1 > \alpha \lg n_H.$$

- The crown itself must be α-balanced (why?), so

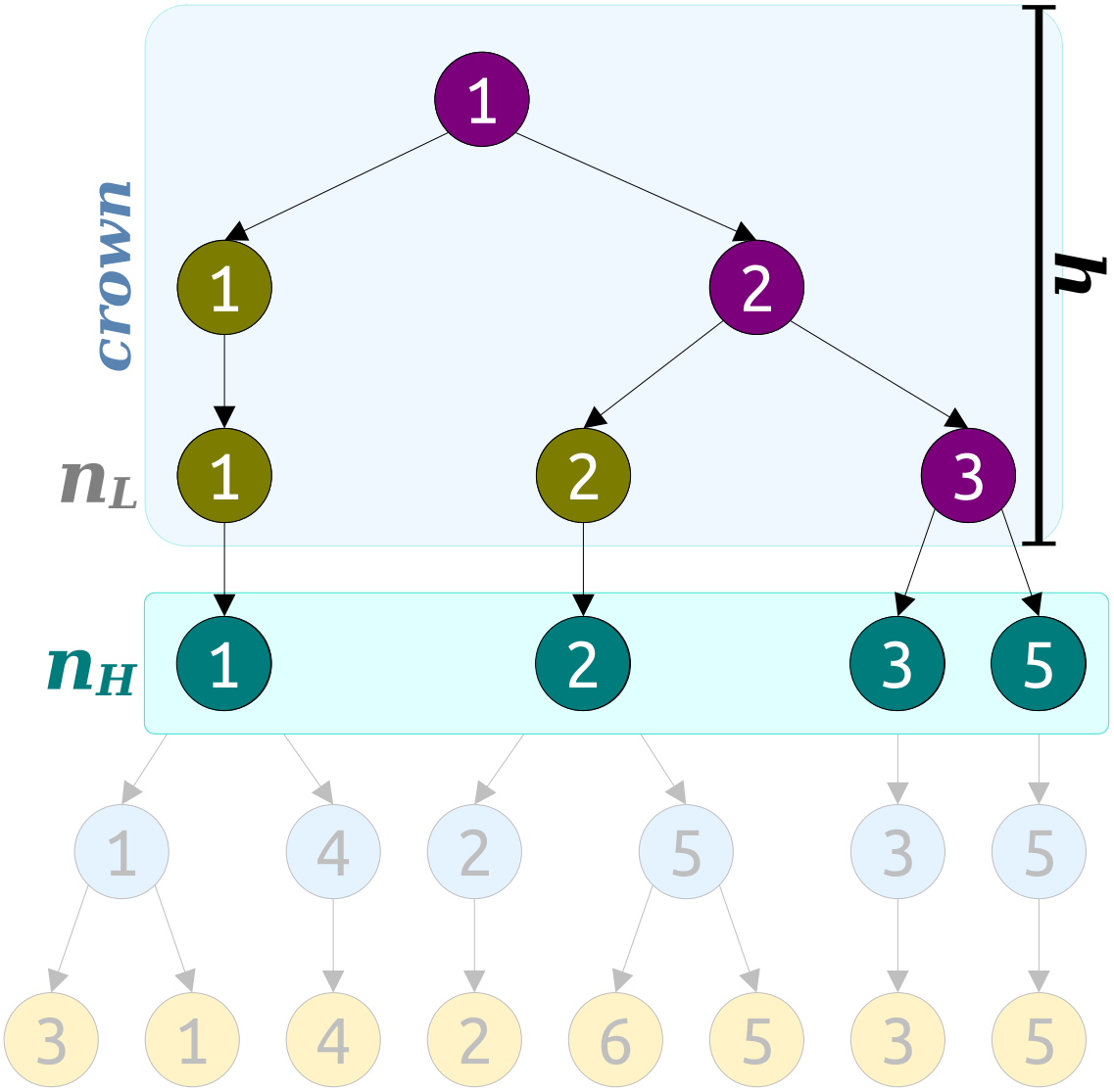
$$h \leq \alpha \lg n_L.$$

- This means that

$$\alpha \lg n_L + 1 > \alpha \lg n_H.$$

- Rearranging gives us that

$$n_L > n_H \cdot 2^{-1/\alpha}.$$



Bounding n_H

- We've just proved these facts:

$$b > 2n_L - n_H.$$

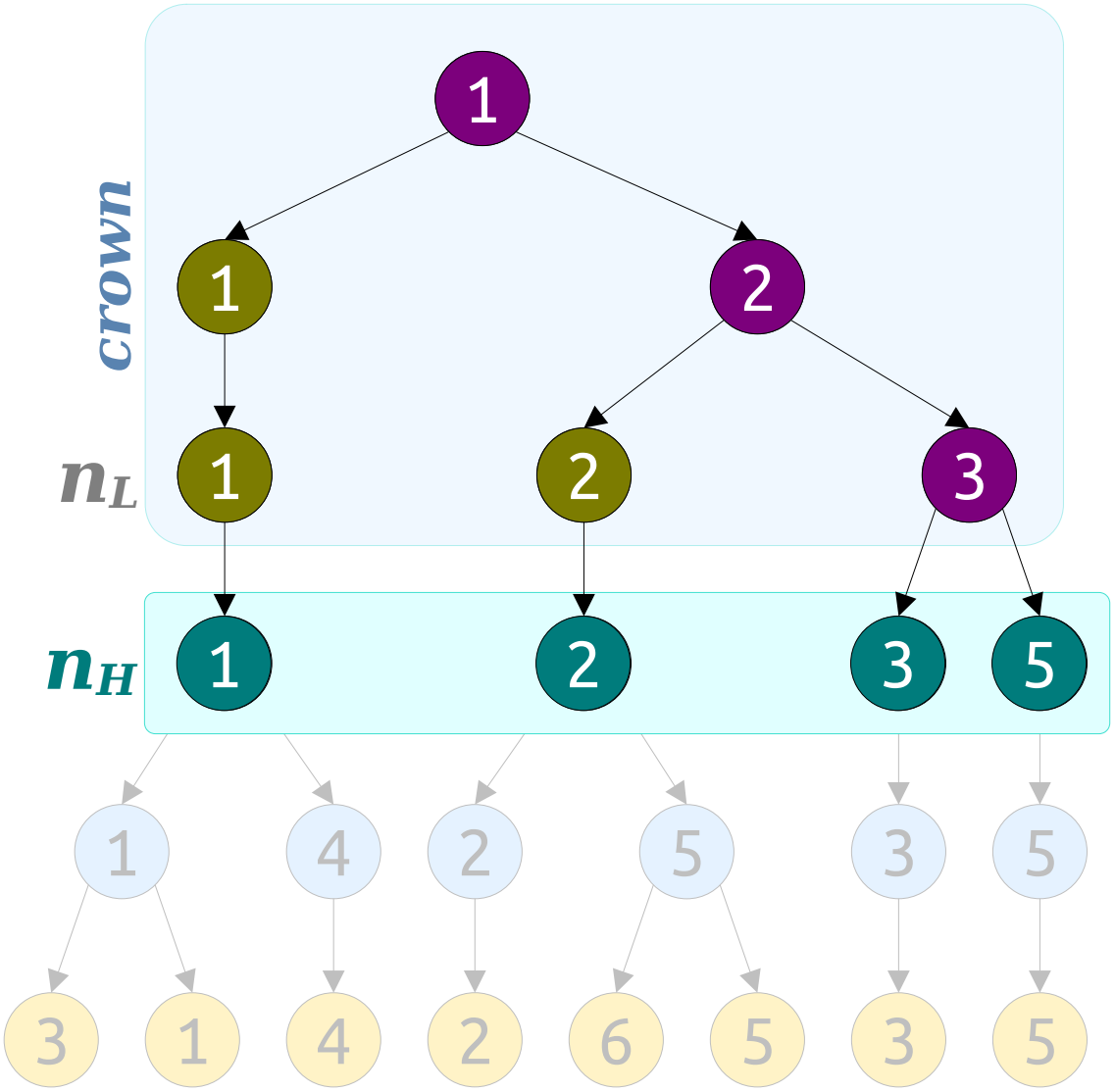
$$n_L > n_H \cdot 2^{-1/\alpha}.$$

- Combining and rearranging gives us

$$(2^{1-1/\alpha} - 1)n_H < b.$$

- Equivalently:

$$n_H < b \cdot (2^{1-1/\alpha} - 1)^{-1}.$$



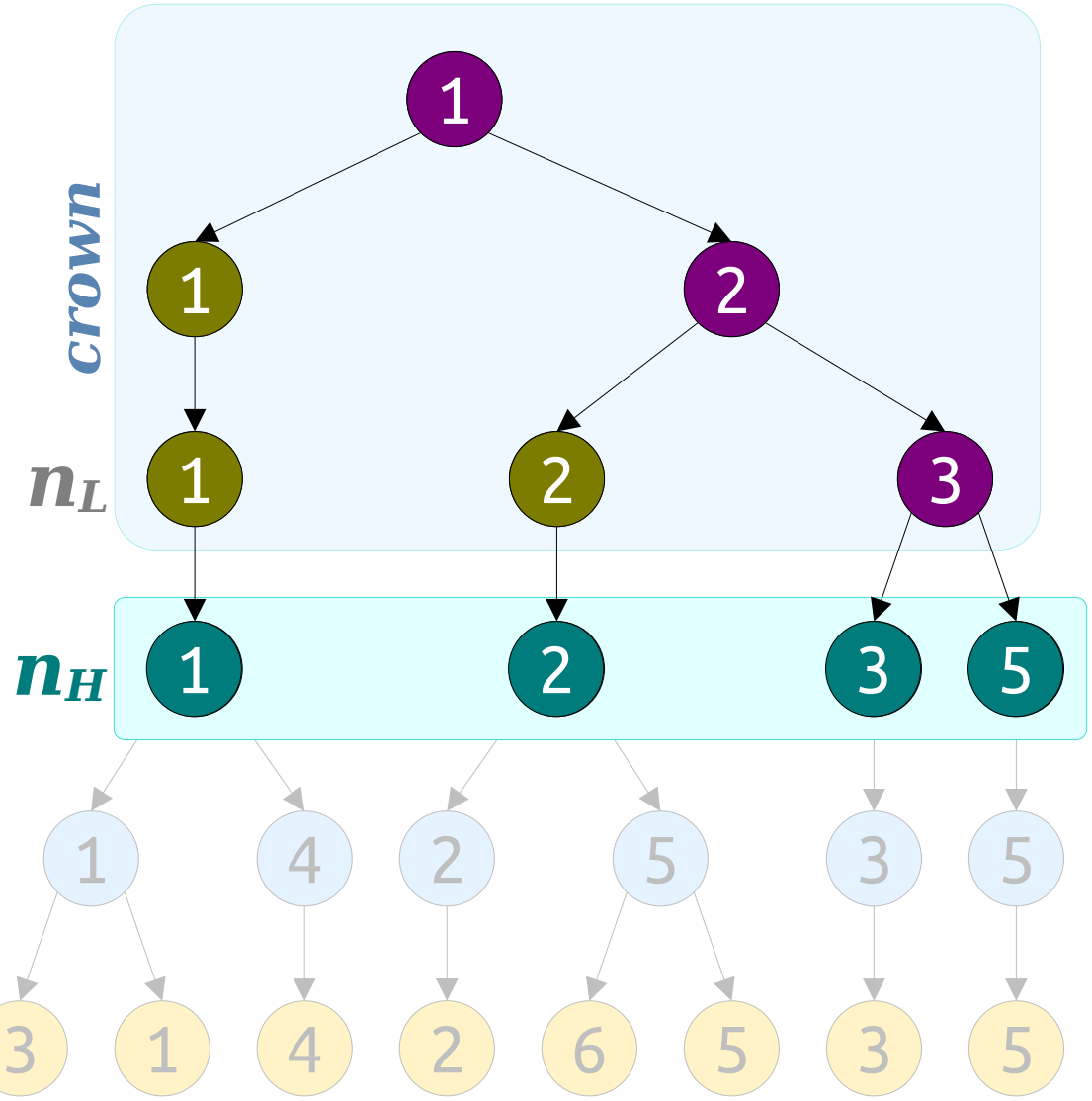
The Final Bound

- Our goal is to bound

  $$g + b + n_H,$$

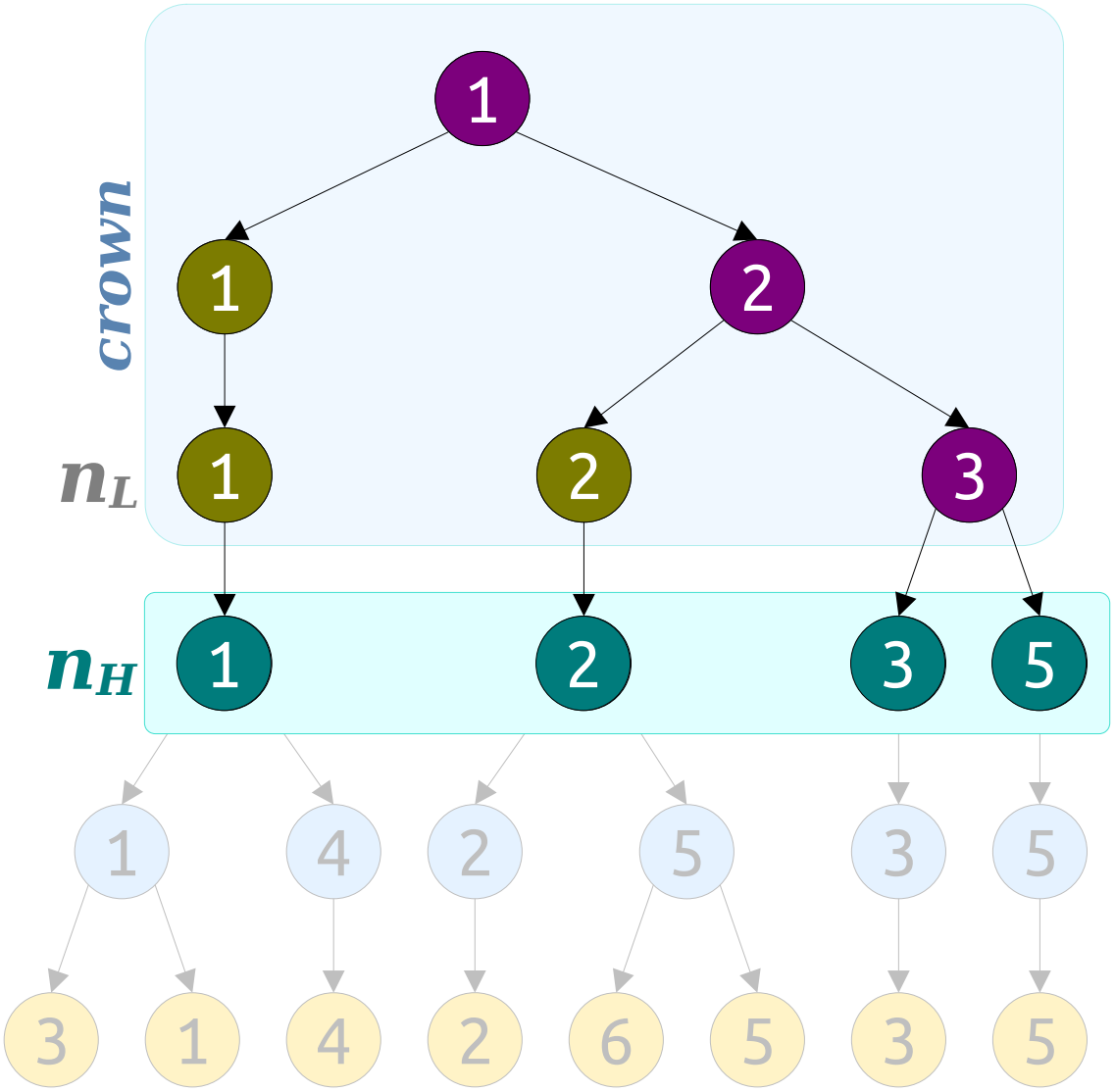
  the total number of nodes scanned when removing the crown.
- Combining together all our previous results, we can bound that expression at

  $$b \cdot (1 + 2(2^{1 - 1/\alpha} - 1)^{-1}).$$
- Letting $z(\alpha)$ denote that latter term, this means we visit at most $b \cdot z(\alpha)$ nodes when deleting the crown.



The Final Bound

- Removing a crown requires at most $b \cdot z(\alpha) = O(b)$ work.
- Intuitively:
  - If we have a large crown, it's because it has a large number of bad nodes in it.
  - We only get one bad node per ***decrease-key***. If we pretend each ***decrease-key*** does O(1) units of extra work, we can pay for the later cost of removing a crown.
- This latter argument suggests we can make everything here amortize away.



Analyzing Abdication Heaps

Analyzing Abdication Heaps

- The worst-case runtimes of each operation on abdication heaps is shown here:
  - ***meld***: O(1)
  - ***enqueue***: O(1)
  - ***decrease-key***: O(1)
  - ***extract-min***: O(n).
- We'll do an amortized analysis to show that ***extract-min*** runs in amortized time O(log n).

Analyzing Abdication Heaps

- We need to define a potential function that is low when our heap has a “nice” shape and is high when our heap has a “bad” shape.
- Intuitively, a “good” heap is one where
  - there are few total trees, and
  - there are few bad nodes.
- ***Idea:*** Define

$$\Phi = T + 2z(\alpha) \cdot B,$$

  where T is the number of trees in the heap and B is the number of bad nodes in the heap.
- The choice of the constant $2z(\alpha)$ here is not arbitrary; we’ll see where that comes into play in a moment.

Analyzing Abdication Heaps

- With this choice of Φ in mind, we can compute the amortized cost of all operations other than ***extract-min*** pretty easily.
  - ***meld*** does O(1) work and doesn't change the number of trees or bad nodes.
  - ***enqueue*** does O(1) work and increases the number of trees by one.
  - ***decrease-key*** does O(1) work and increases the number of trees by one and the number of bad nodes by one.
- Each of these does O(1) work and has $\Delta\Phi = O(1)$, so their amortized costs are all O(1).

Analyzing Abdication Heaps

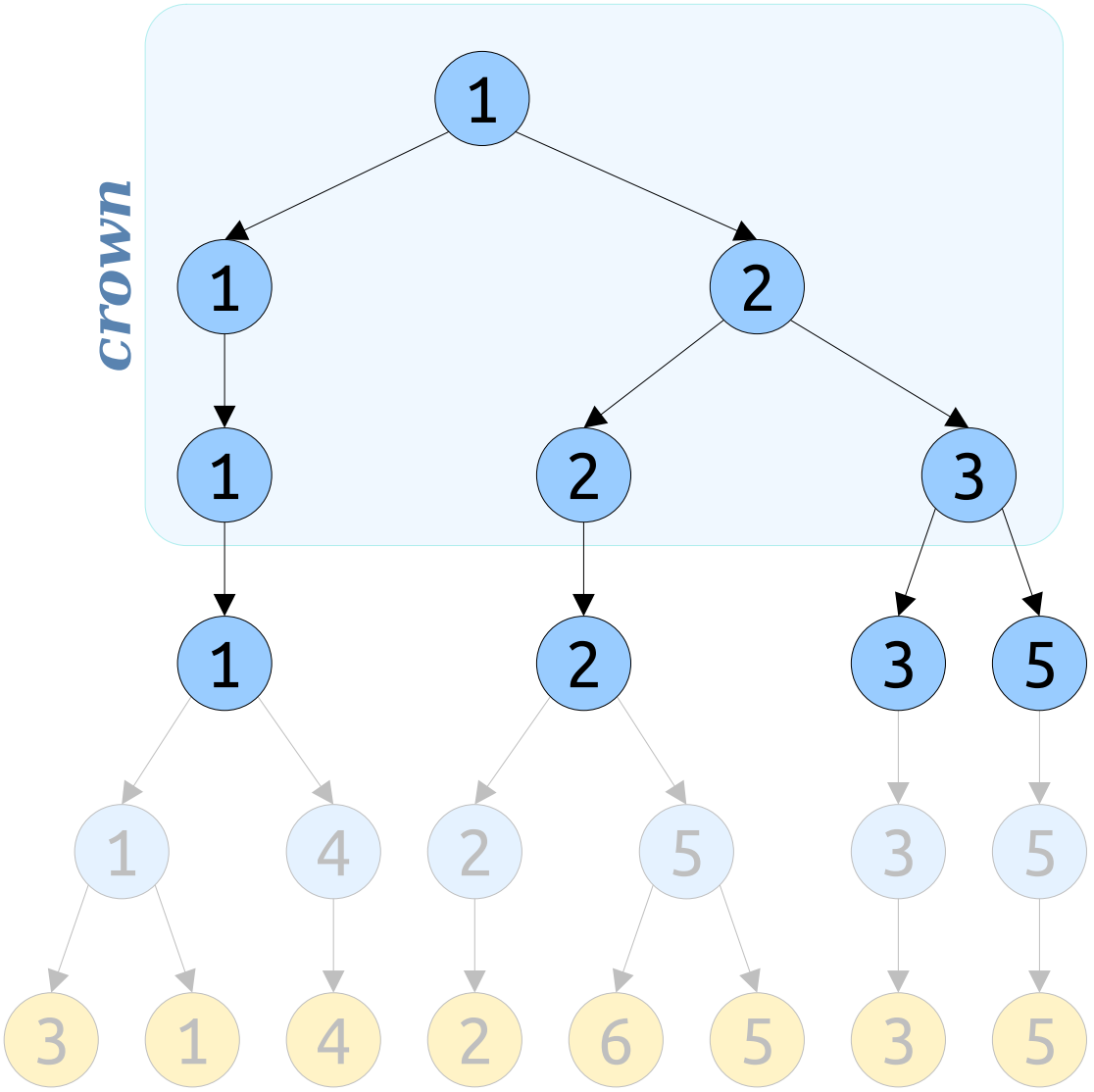
- The trickier part will be analyzing ***extract-min*** procedure. Here's a rundown of the steps involved:
  - ***Step 1:*** Perform the standard deletion algorithm for a lazy tournament heap.
  - ***Step 2:*** Remove the crowns of all overtall trees.
- Let's analyze the cost of each step in turn.

Analyzing Step 1

- ***Step 1:*** The standard deletion logic for a lazy tournament heap.
- The amortized cost is O(log n), using more or less the same amortized analysis as we did with regular lazy tournament heaps.
- Real Cost: O(t + log n).
  - O(t) work to find the tree containing the minimum.
  - O(log n) work to remove the root-leaf path containing the minimum, since all trees have height at most α lg n.
  - O(log n) work to link newly-exposed trees into the main list of trees.
  - O(t + log n) work to perform the ***coalesce***.
- $\Delta\Phi$: -t + O(log n).
  - Begin with t trees.
  - When done, we have O(log n) trees. *(Prove this!)*

Analyzing Step 2

- ***Step 2:*** Remove the crowns of each overtall tree.
- Pick a tree with b bad nodes in its crown. There are at most $b \cdot z(\alpha)$ nodes in the crown and the layer below it.
- Real cost to decrown: $O(b)$.
- $\Delta\Phi$: $-b \cdot z(\alpha)$
  - $2z(\alpha) \cdot \Delta B = -2b \cdot z(\alpha)$.
  - $\Delta T = +b \cdot z(\alpha)$.
- Amortized cost per tree: O(1).
- Total amortized cost: **O(log *n*)**, since there are at most O(log n) overtall trees.



The Overall Analysis

- Here's the final scorecard for the abdication heap.
- These are excellent theoretical runtimes. There's minimal room for improvement!
- It is possible to make all these operations *worst-case efficient* at a significant increase in both runtime and intellectual complexity.

enqueue: O(1)

meld: O(1)

extract-min: O(log n)*

decrease-key: O(1)

**amortized*

In Practice

- In practice, the constant factors on abdication 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, abdication 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

- If you're interested in these sorts of priority queues, consider reading up on the following:
  - In 1984, Fredman and Tarjan invented the ***Fibonacci heap***, the first priority queue that supported ***decrease-key*** in (amortized) constant time.
  - In 1986, a powerhouse team (Fredman, Sedgewick, Sleator, and Tarjan) invented the ***pairing heap***. It's much simpler than an abdication 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 an abdication heap, but in a *worst-case* rather than *amortized* sense.
  - In 2013, Chan invented the ***quake heap***. It uses a very similar strategy to our abdication heaps, but looks globally across all trees when deciding whether to remove nodes. (Abdication heaps are my own reworking of quake heaps to make the edits local per tree rather than global across the heap.)
- 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 $\varepsilon > 0$.

Next Time

- ***String Data Structures***
  - Data structures for storing text.
- ***Suffix Trees***
  - A beautiful and subtle data structure for text processing.