Fibonacci Heaps

Outline for Today

- ***Review 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.

Review: (Lazy) Binomial Heaps

Building a Priority Queue

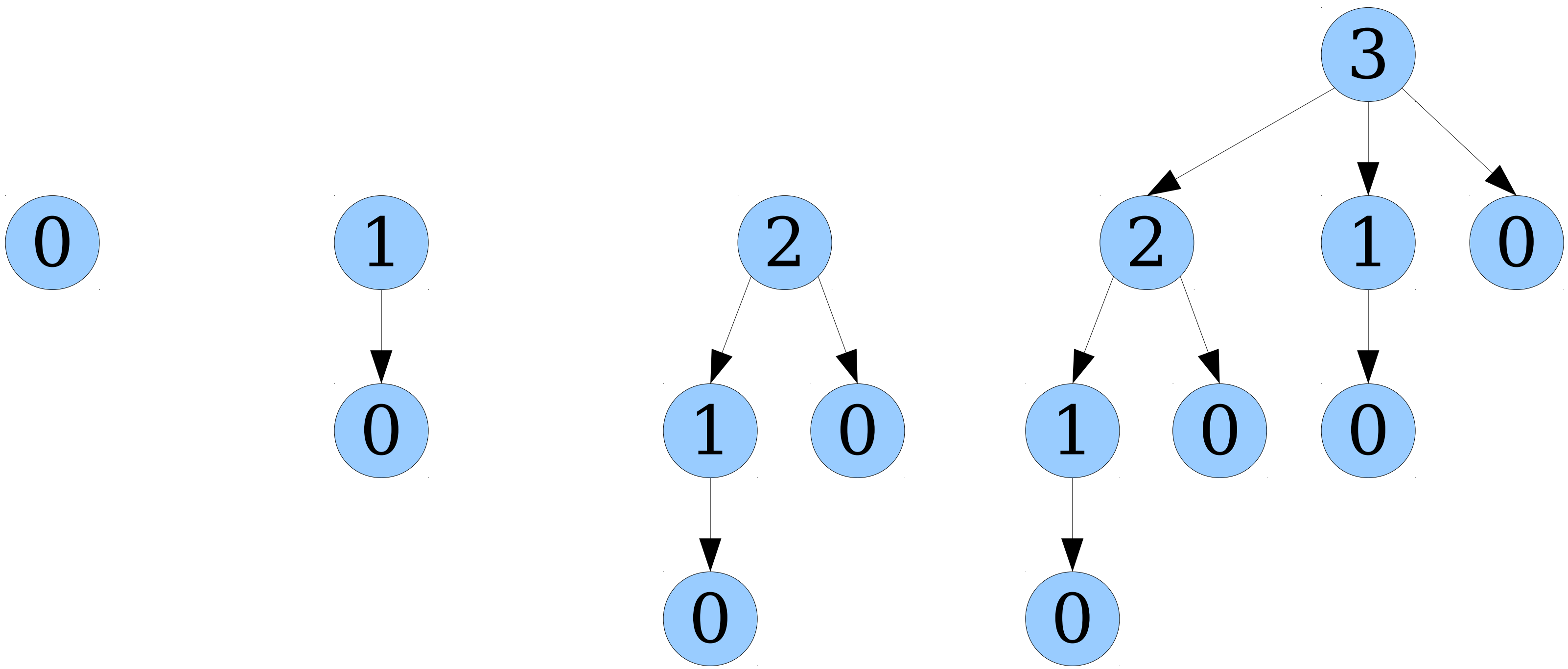
- Group nodes into “packets” with the following properties:
  - Size must be a power of two.
  - Can efficiently fuse packets of the same size.
  - Can efficiently find the minimum element of each packet.
  - Can efficiently “fracture” a packet of 2^k nodes into packets of 1, 2, 4, 8, …, 2^{k-1} nodes.

Binomial Trees

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

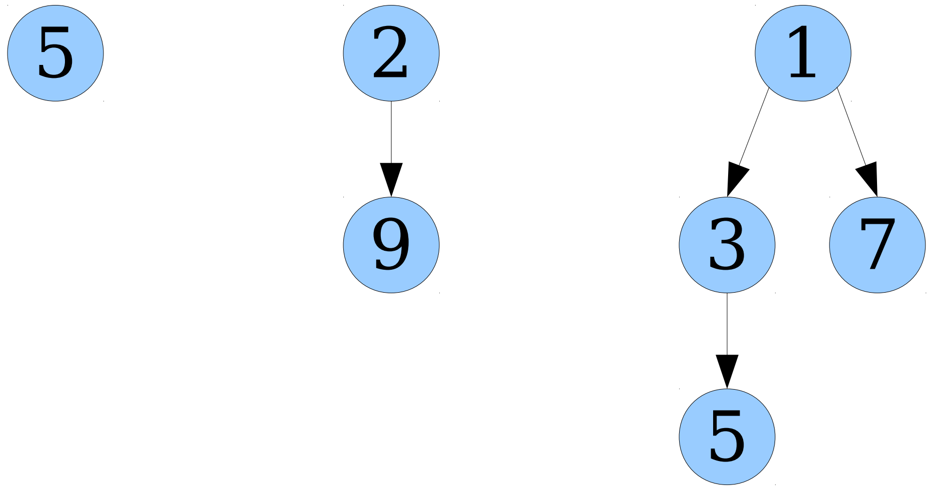
  A binomial tree of order k is a single node whose children are binomial trees of order 0, 1, 2, …, k – 1.

- Here are the first few binomial trees:



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.”



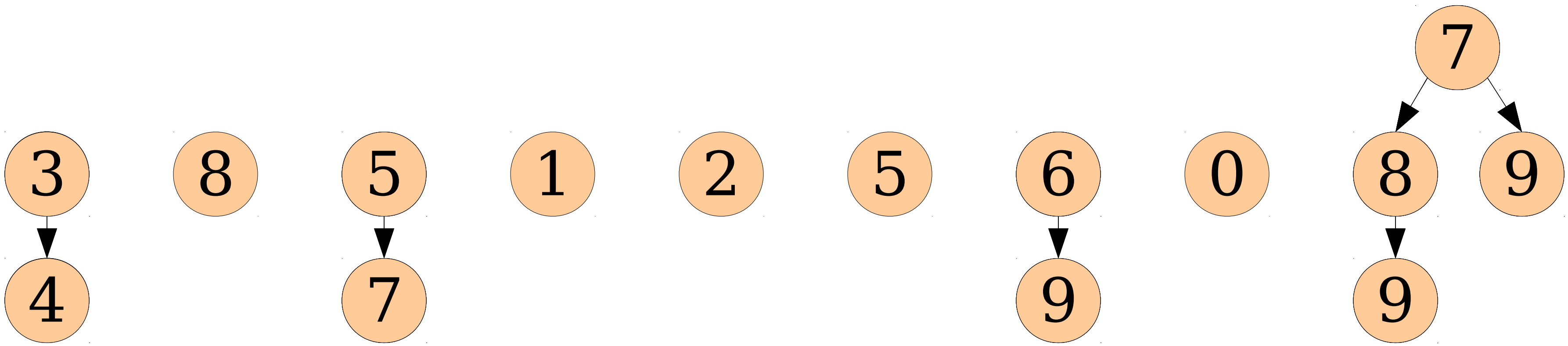
The Binomial Heap

- 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*).

Lazy Binomial Heaps

- A ***lazy binomial heap*** is a variation on a standard binomial heap in which ***meld***s are done lazily by concatenating tree lists together.
- Tree roots are stored in a doubly-linked list.
- An extra pointer is required that points to the minimum element.
- ***extract-min*** eagerly coalesces binomial trees together and runs in amortized time O(log n).

Coalescing Trees

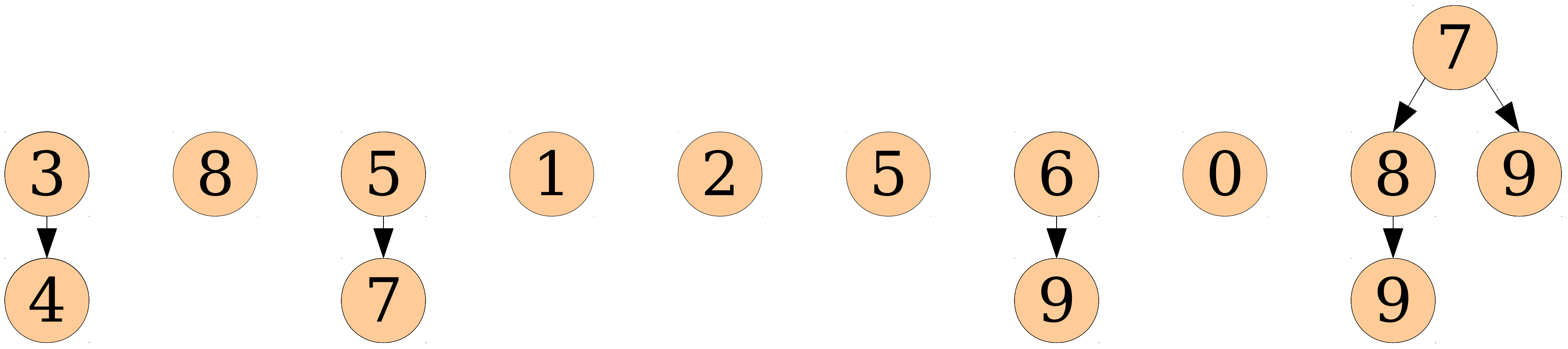


Coalescing Trees

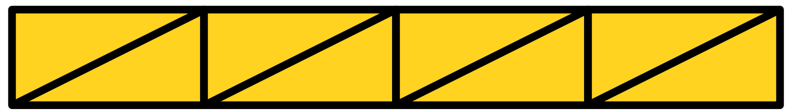
Total number of nodes: **15**

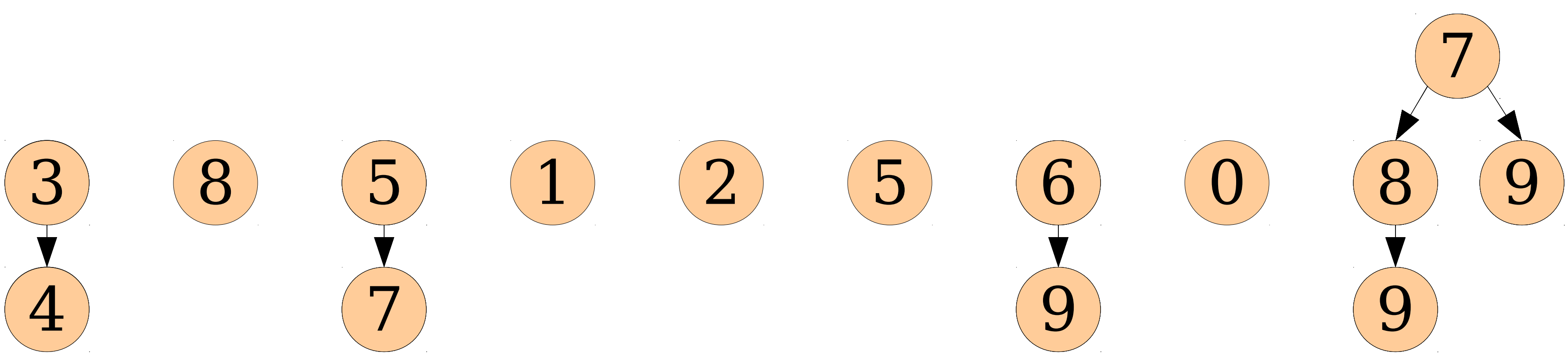
(Can compute in time $\Theta(T)$, where T is the number of trees, if each tree is tagged with its order)

Bits needed: **4**

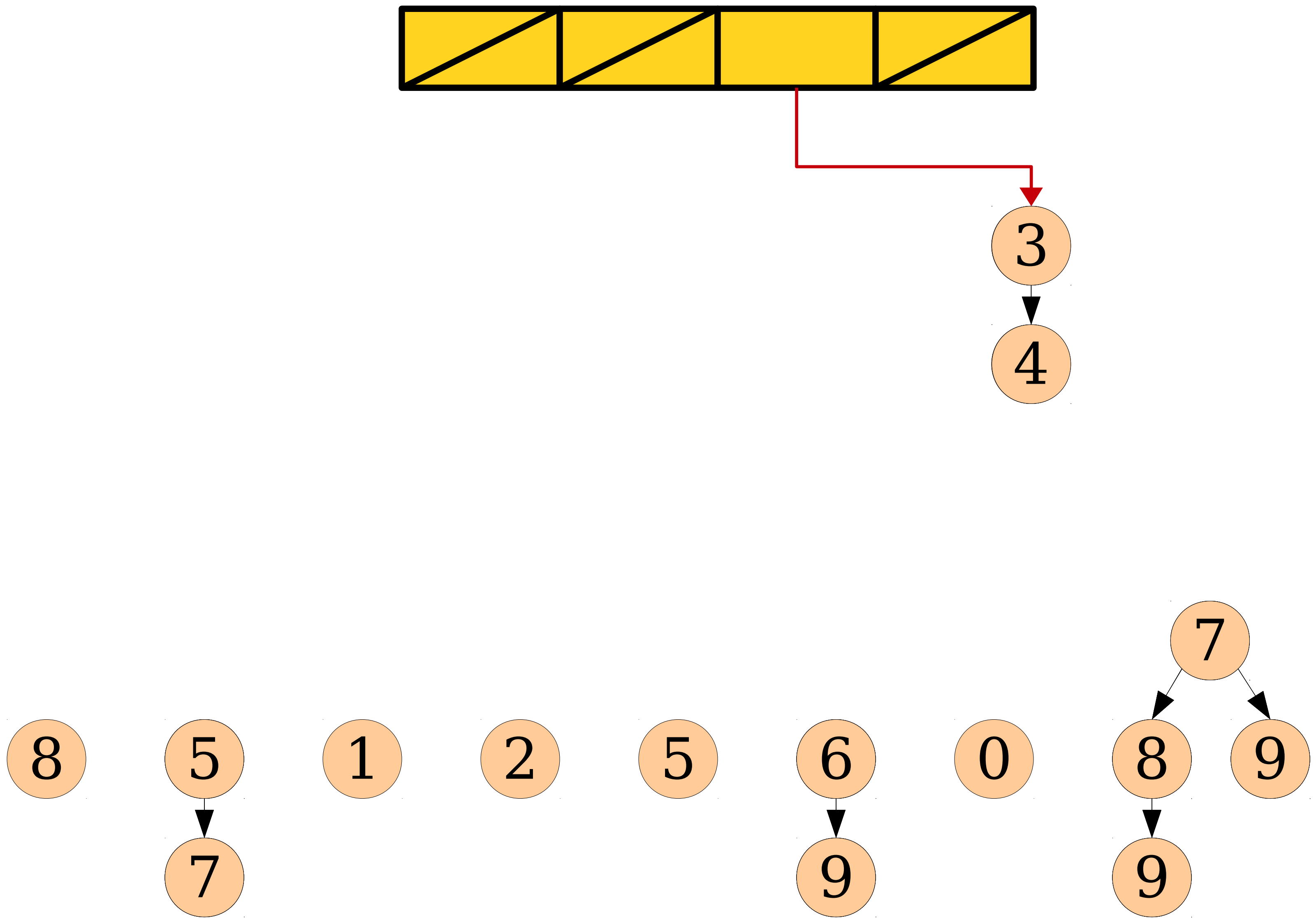


Coalescing Trees

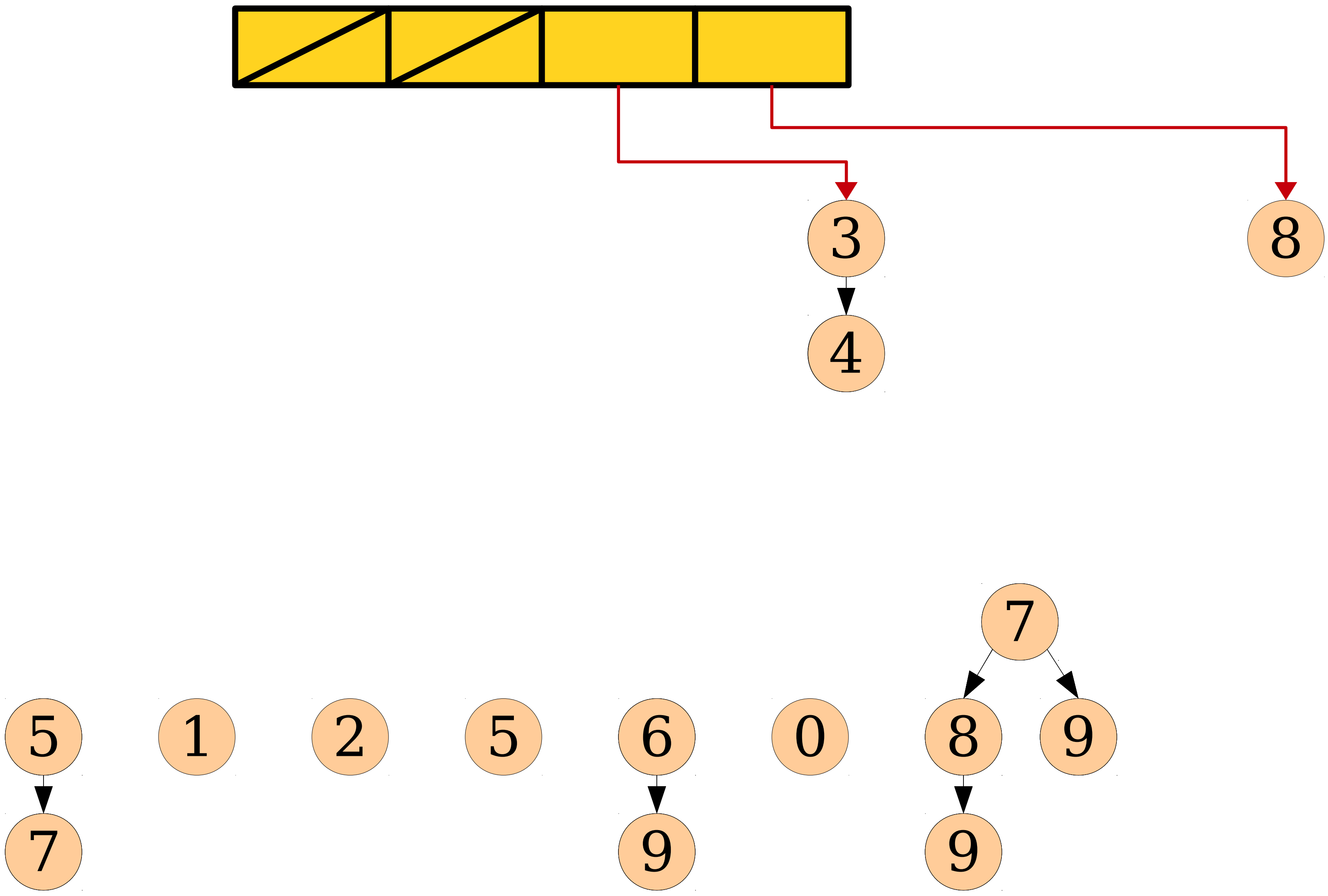




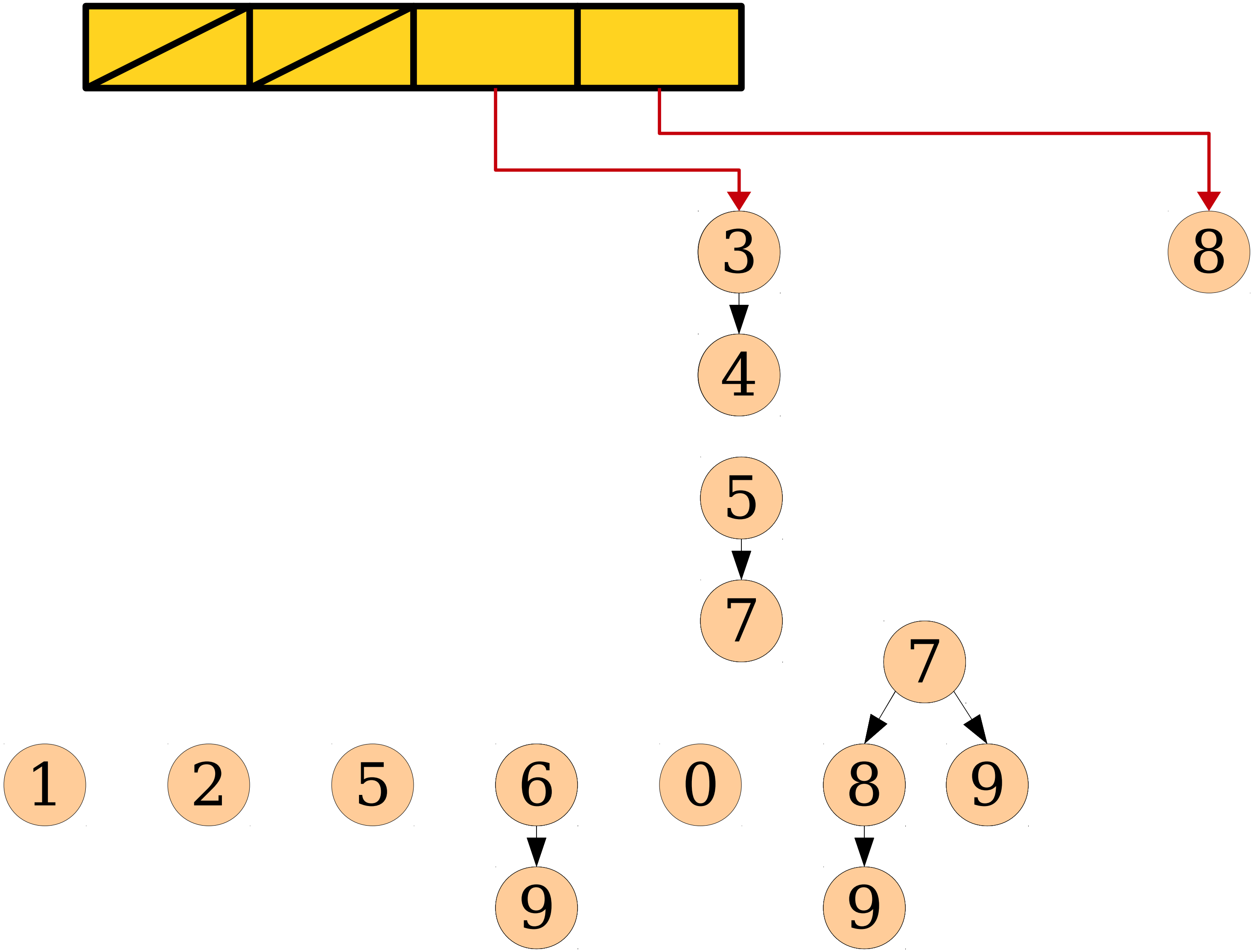
Coalescing Trees



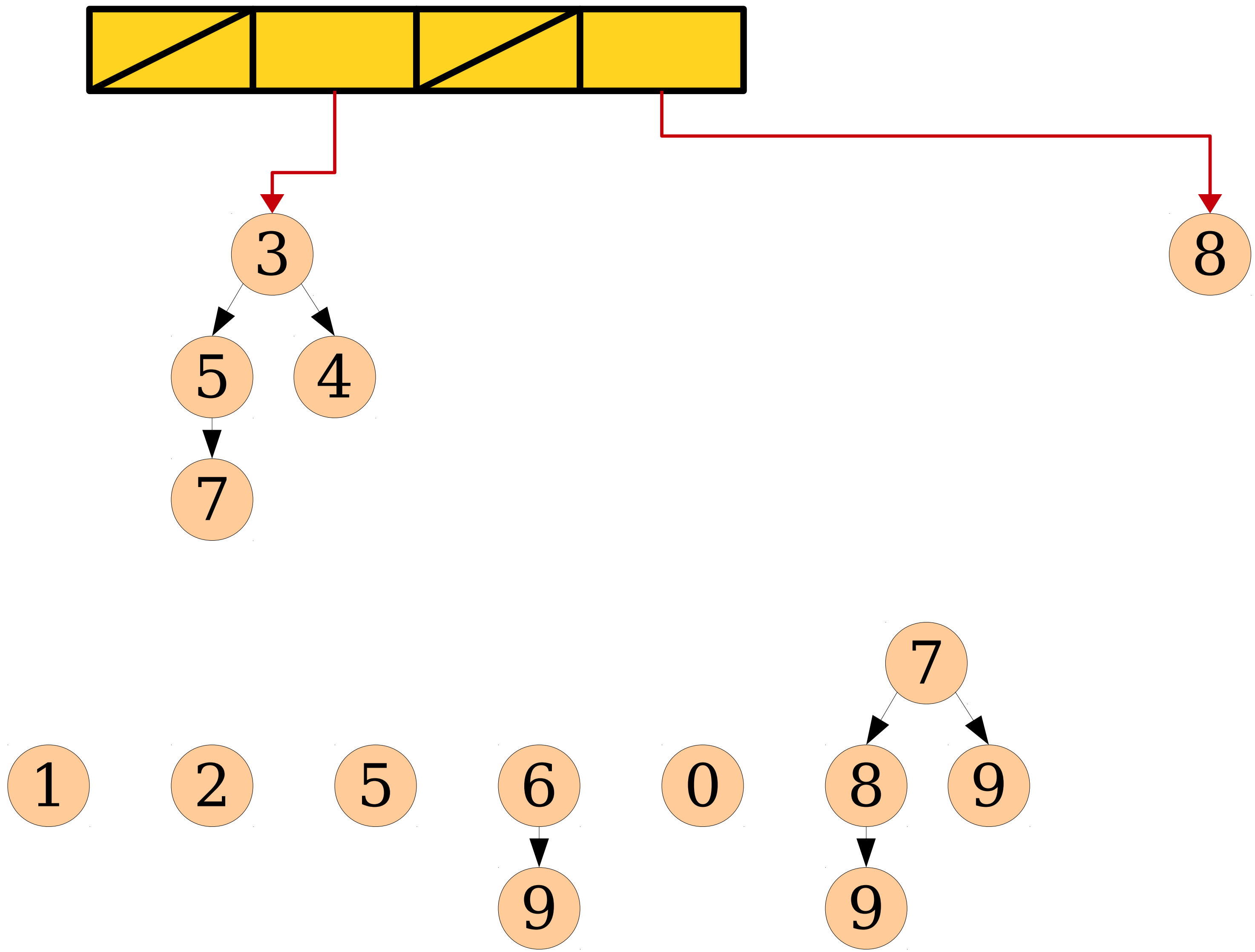
Coalescing Trees



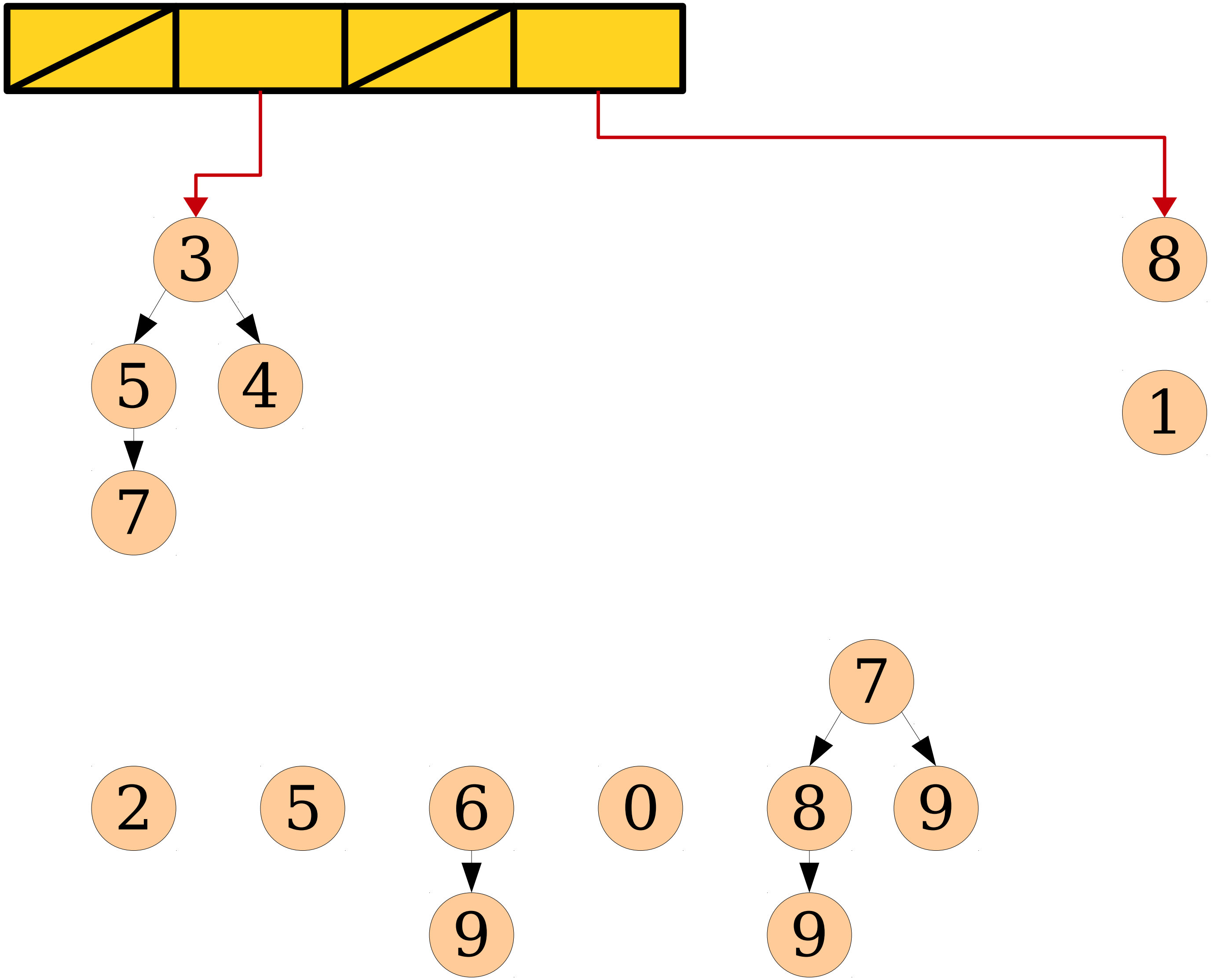
Coalescing Trees



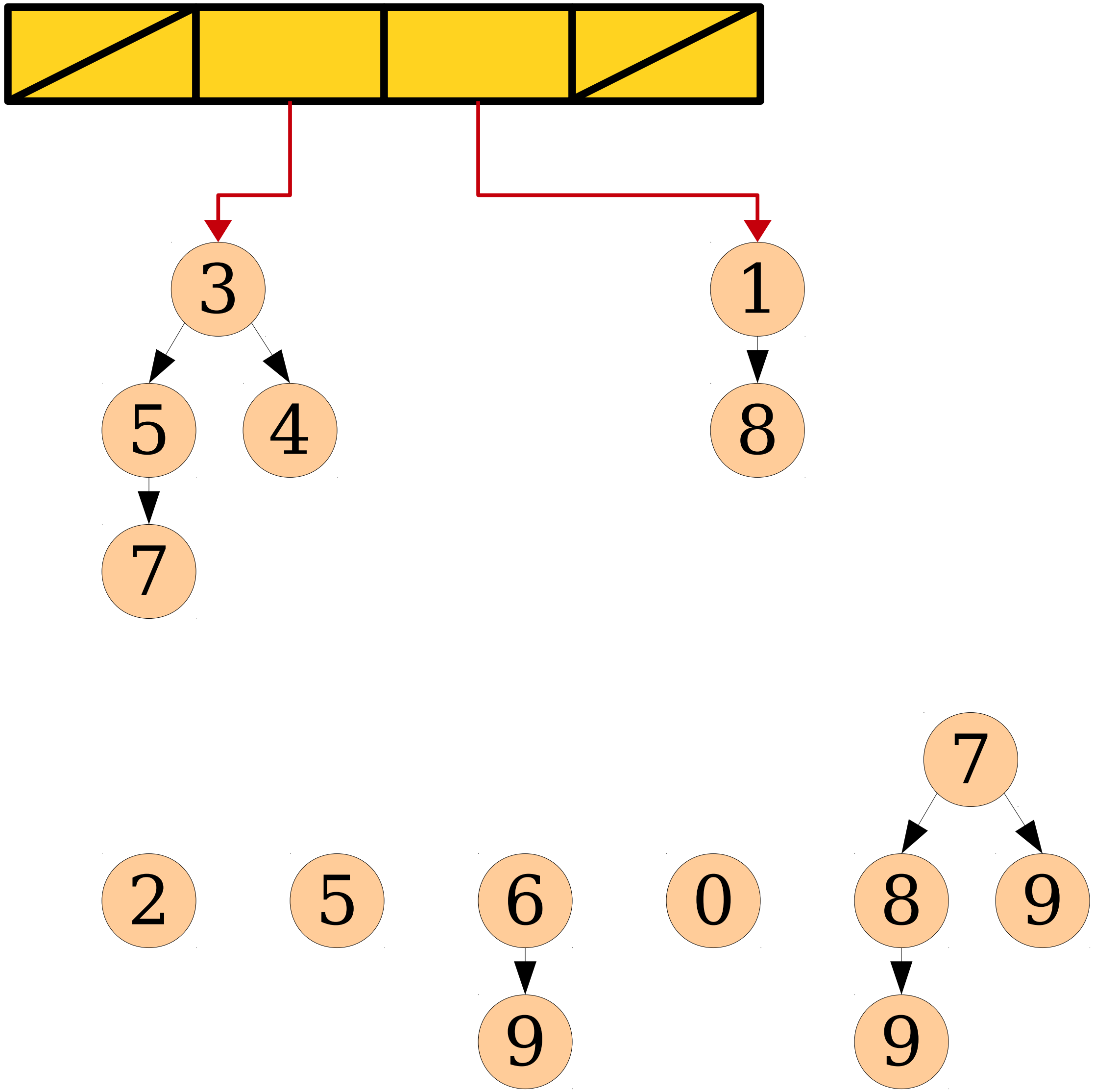
Coalescing Trees



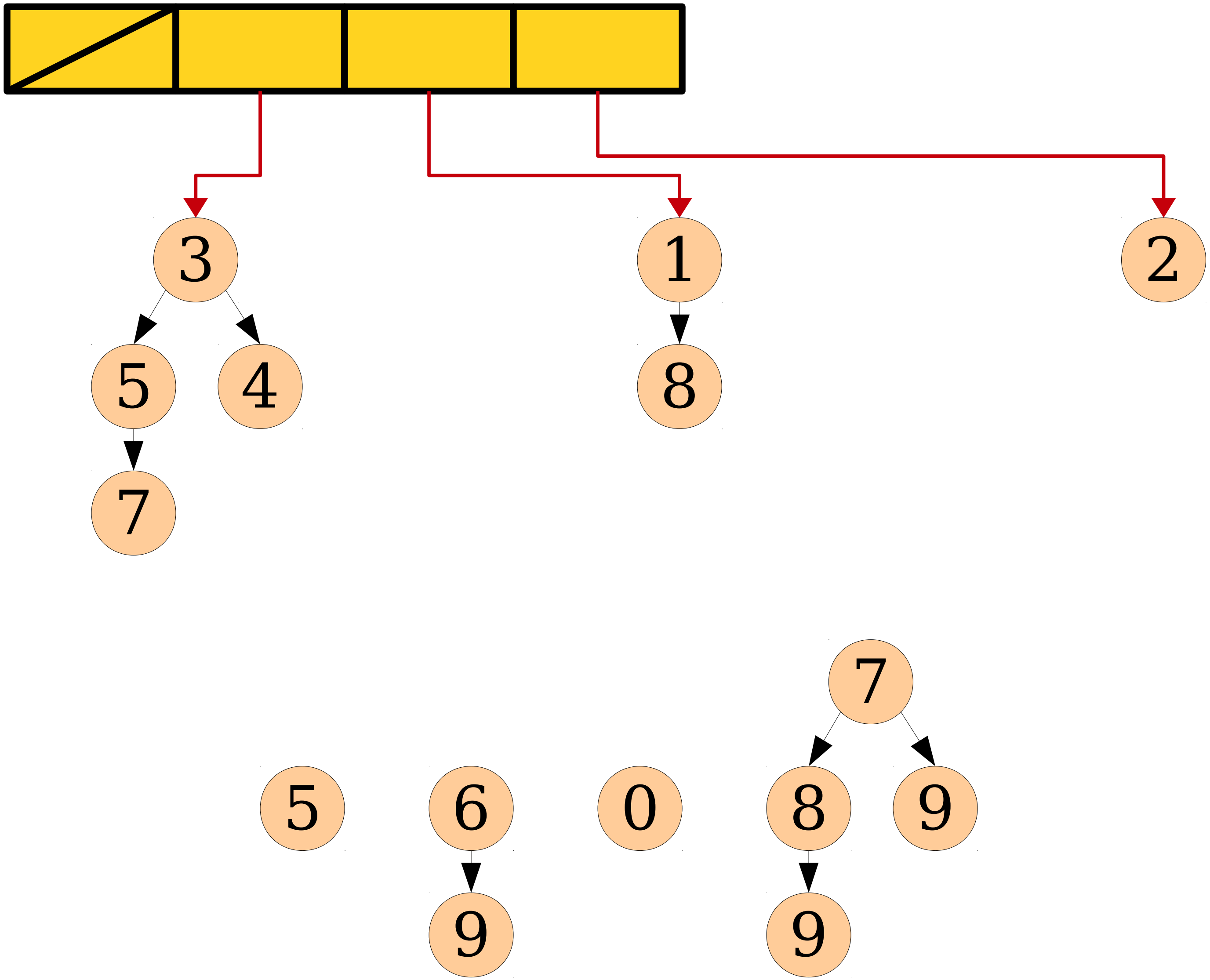
Coalescing Trees



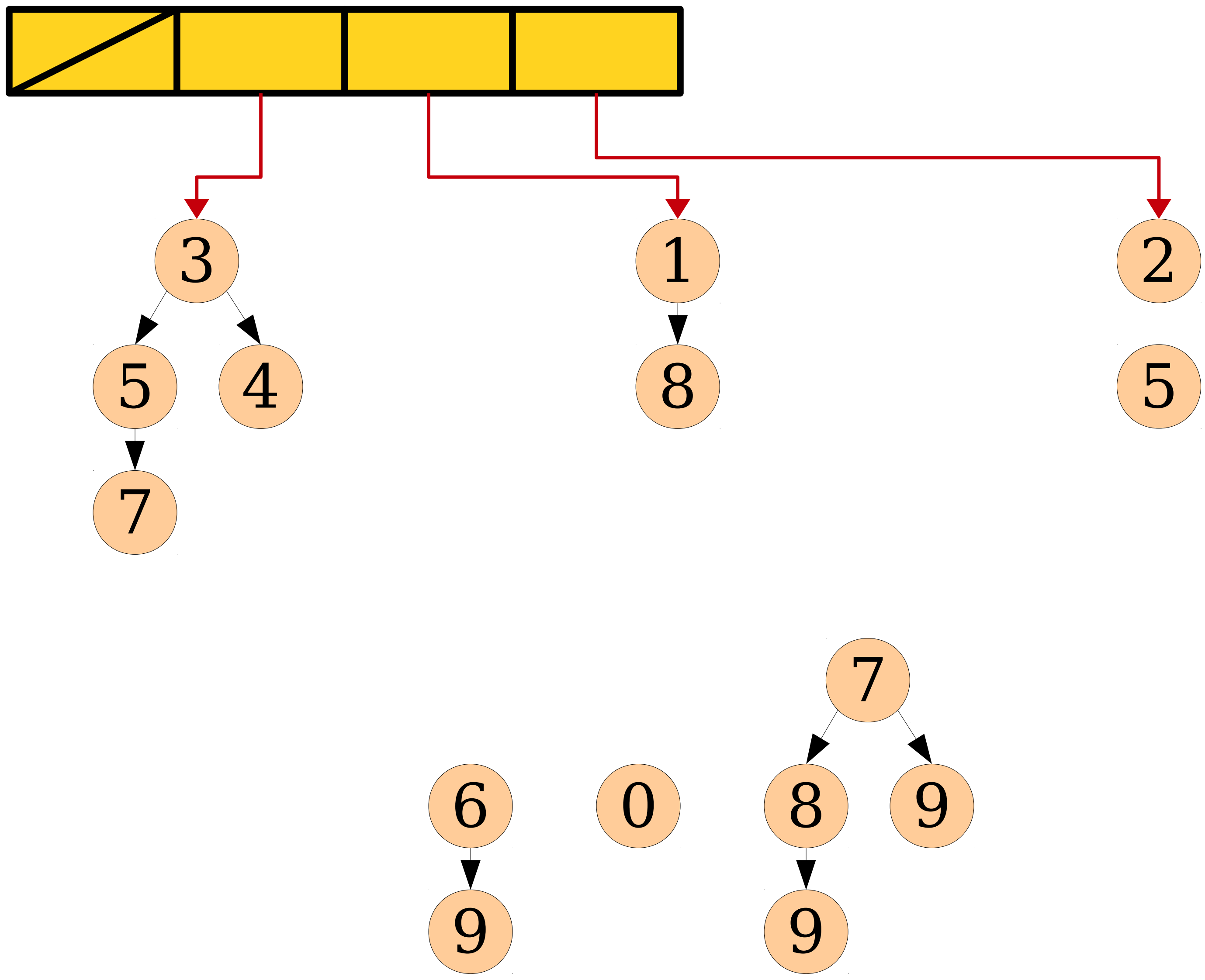
Coalescing Trees



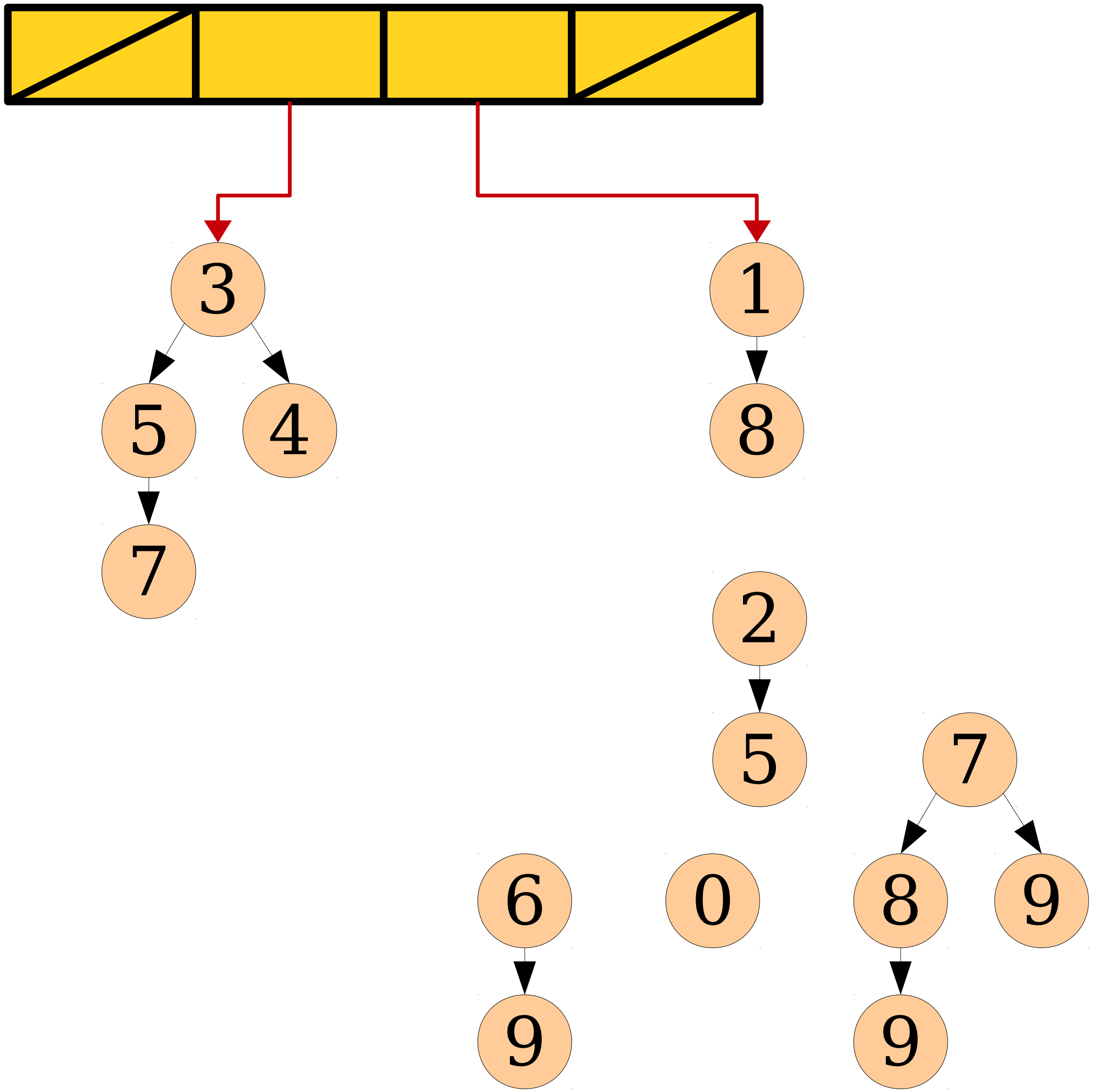
Coalescing Trees



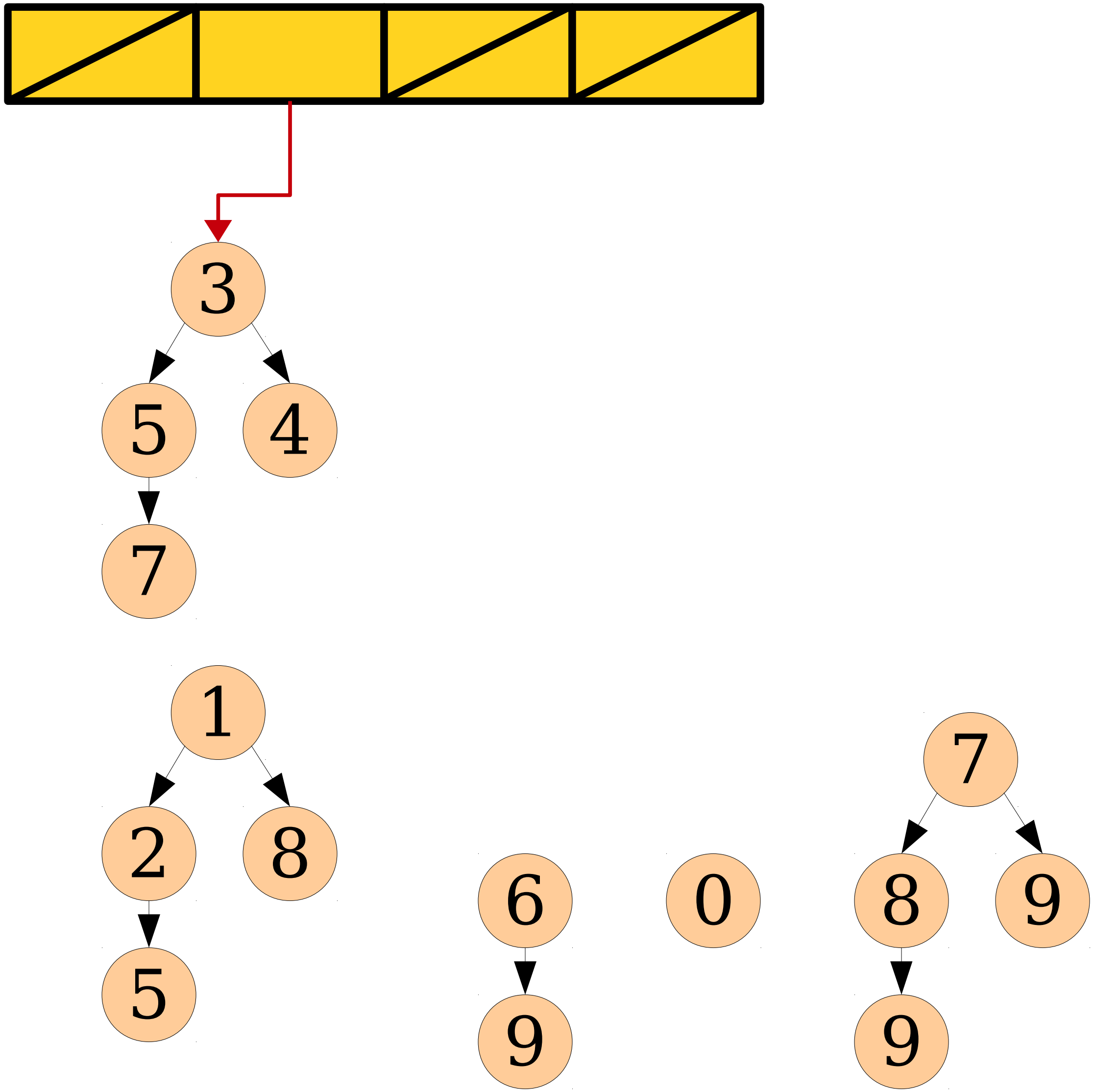
Coalescing Trees



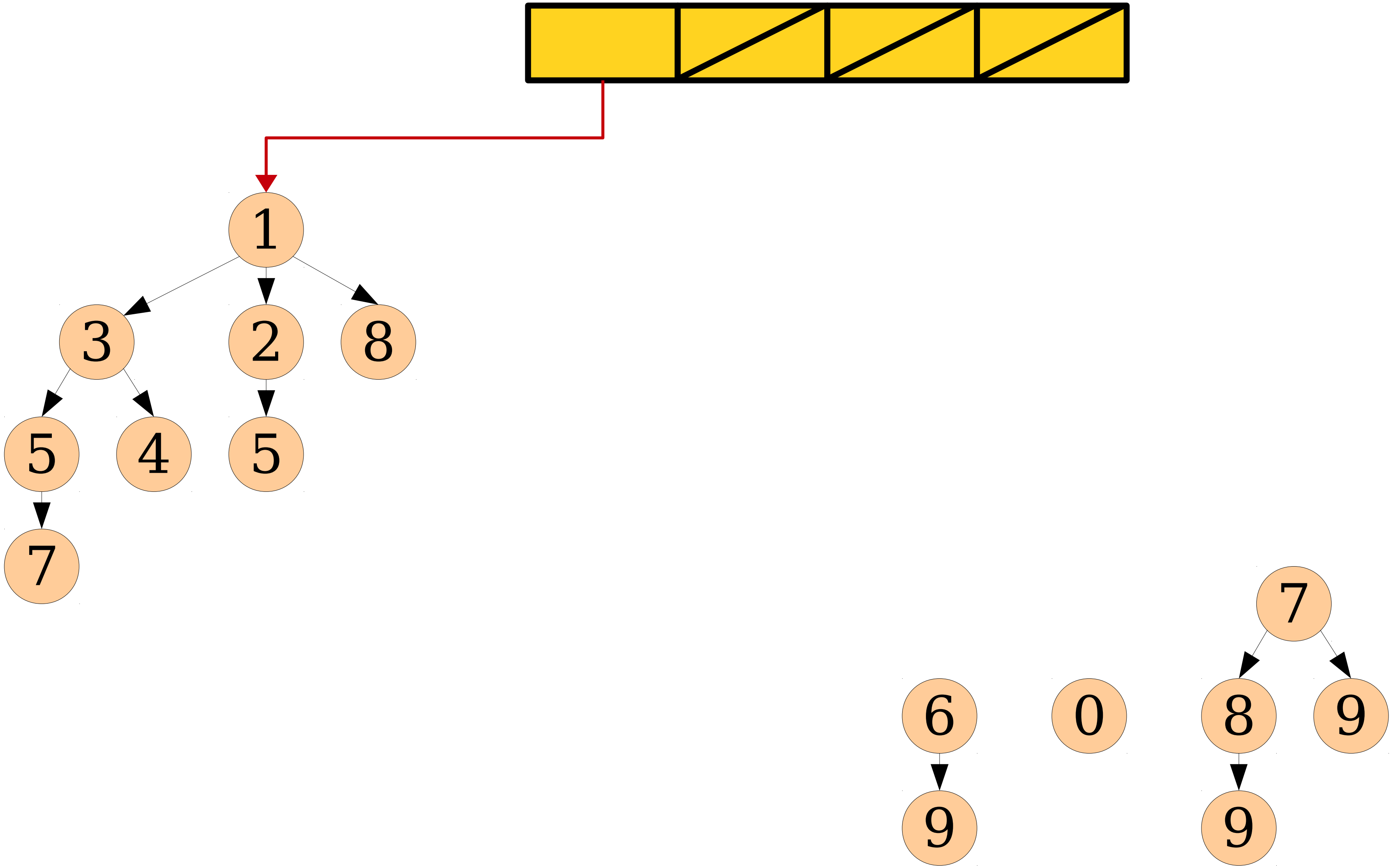
Coalescing Trees



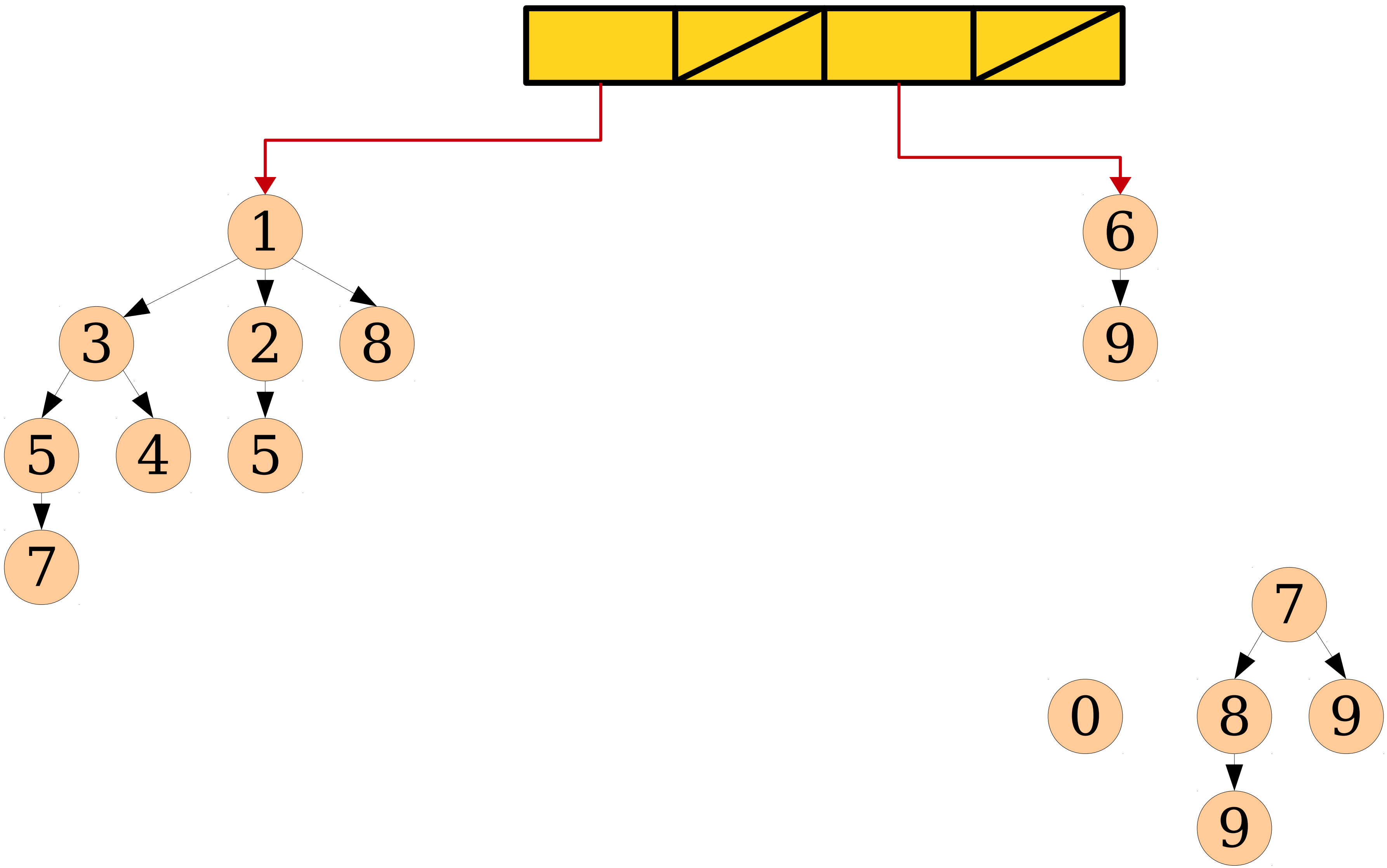
Coalescing Trees



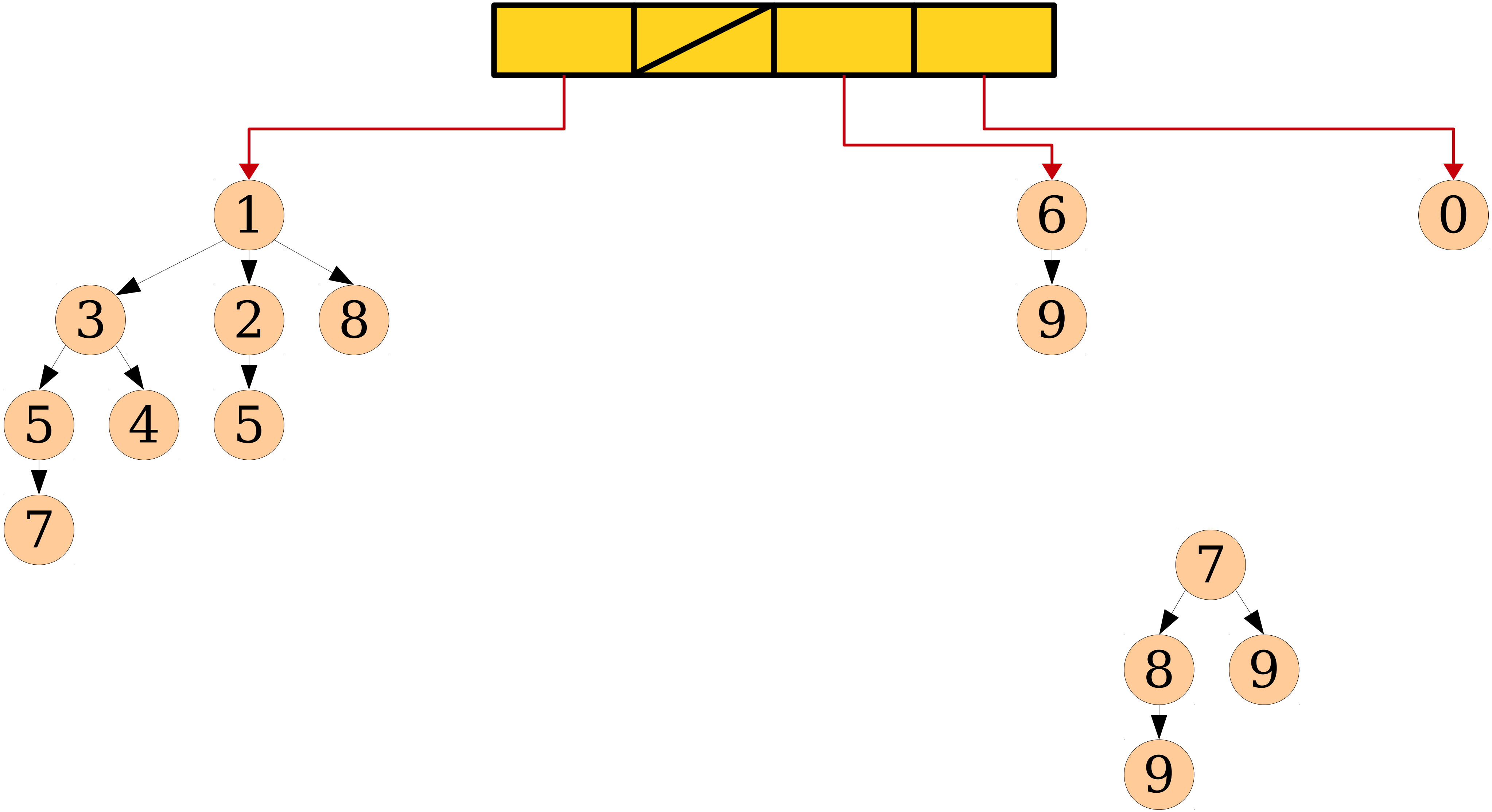
Coalescing Trees



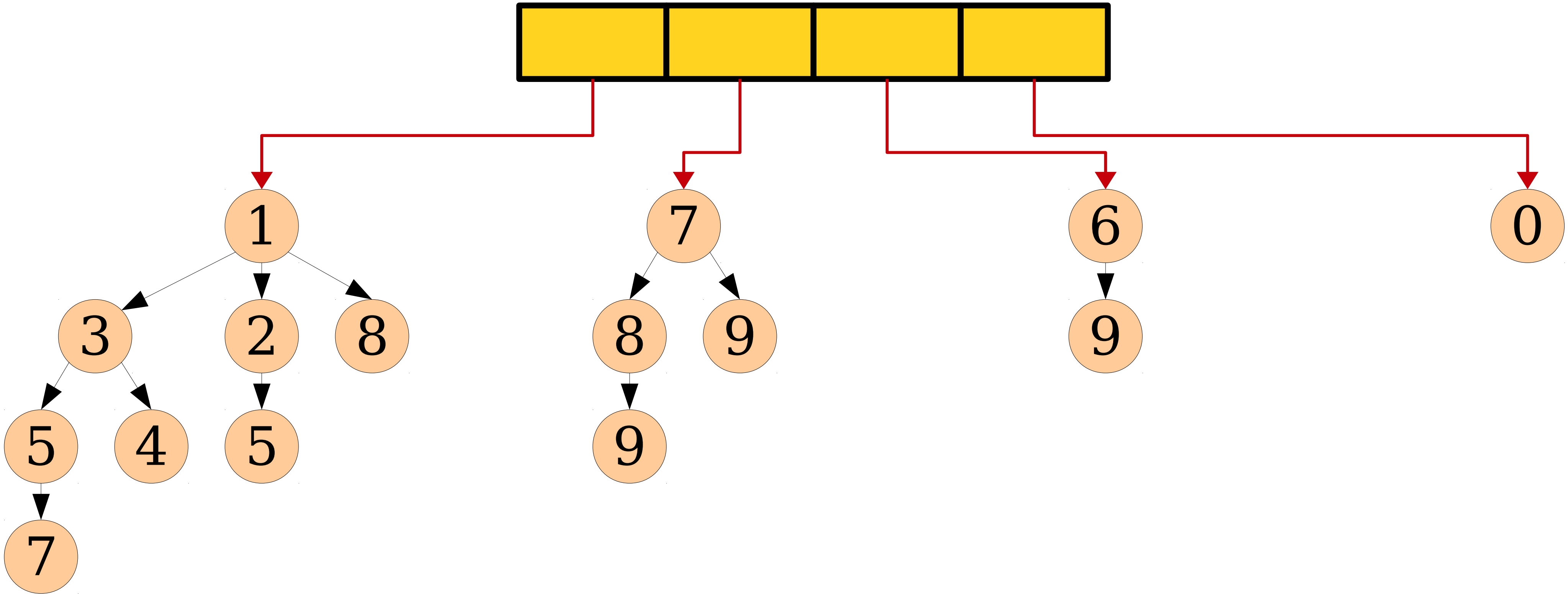
Coalescing Trees



Coalescing Trees



Coalescing Trees



The Overall Analysis

- Set $\Phi(D)$ to be the number of trees in D.
- 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)
- Details are in the previous lecture.
- Let's quickly review ***extract-min***'s analysis.

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 + O(log n).
- The runtime for coalescing these trees is O(T + log n).
- When we're done merging, there will be O(log n) trees remaining, so ΔΦ = -T + O(log n).
- Amortized cost is

$$\Theta(T + \log n) + O(1) \cdot (-T + O(\log n))$$

$$= \Theta(T) - O(1) \cdot T + O(1) \cdot O(\log n)$$

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

A Detail in the Analysis

- The amortized cost of an extract-min is

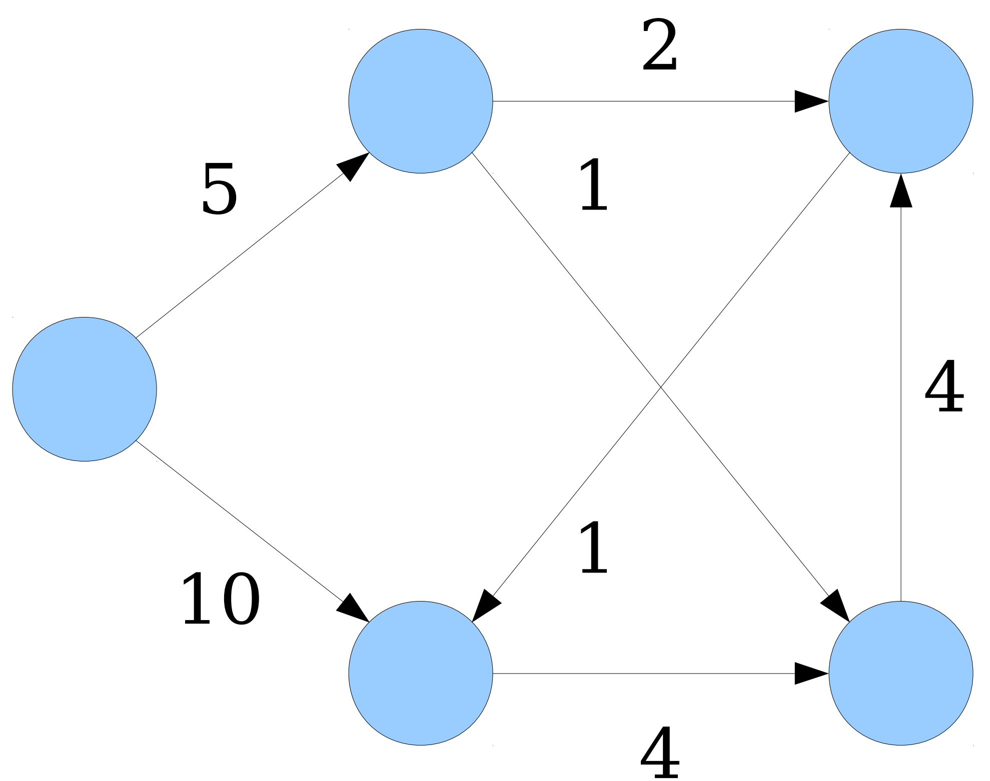
$$O(\log n + T) + O(1) \cdot (-T + O(\log n))$$

- Where do these $O(\log n)$ terms come from?
  - First $O(\log n)$: Removing the minimum element might expose $O(\log n)$ children, since the maximum order of a tree is $O(\log n)$.
  - Second $O(\log n)$: Maximum number of trees after a coalesce is $O(\log n)$.
- ***Key idea:*** This $O(\log n)$ term arises because the number of nodes in an order-k binomial tree grows exponentially with k.

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

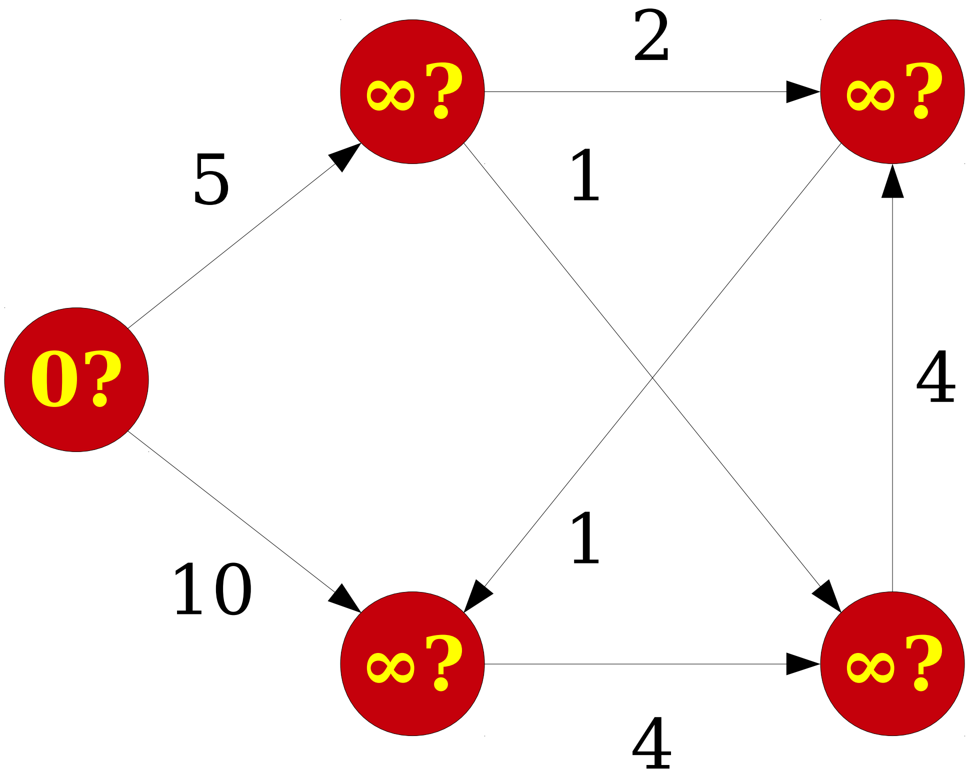
Review: Dijkstra's Algorithm

- Dijkstra's algorithm solves the single-source shortest paths (SSSP) problem in graphs with nonnegative edge weights.



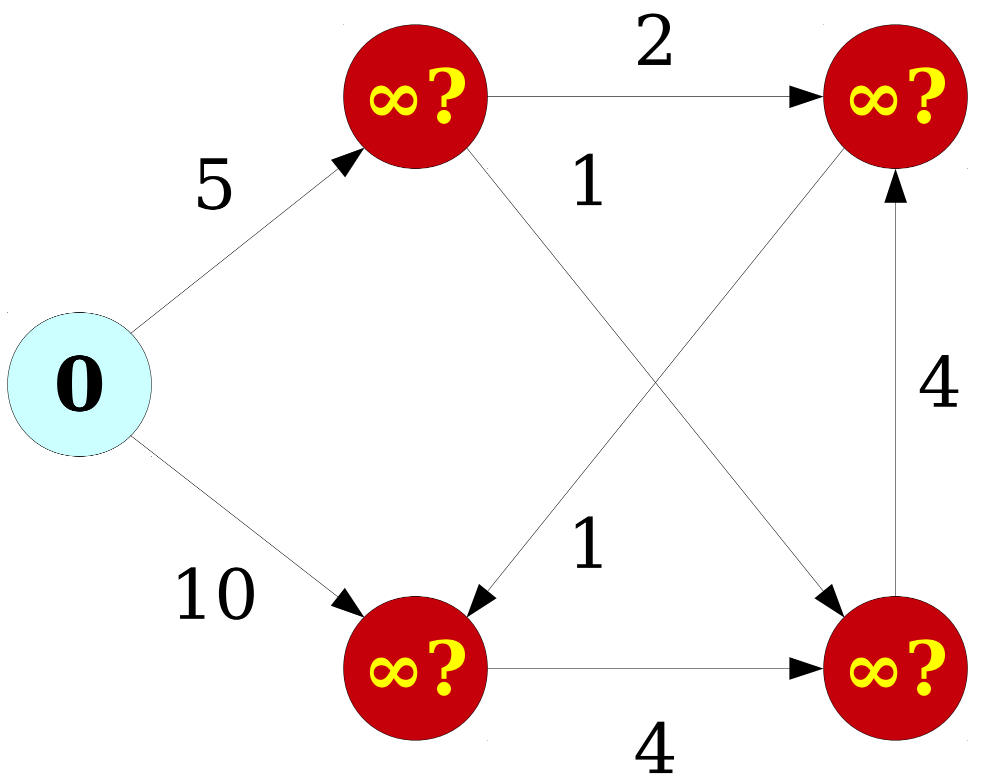
Review: Dijkstra's Algorithm

- Dijkstra's algorithm solves the single-source shortest paths (SSSP) problem in graphs with nonnegative edge weights.



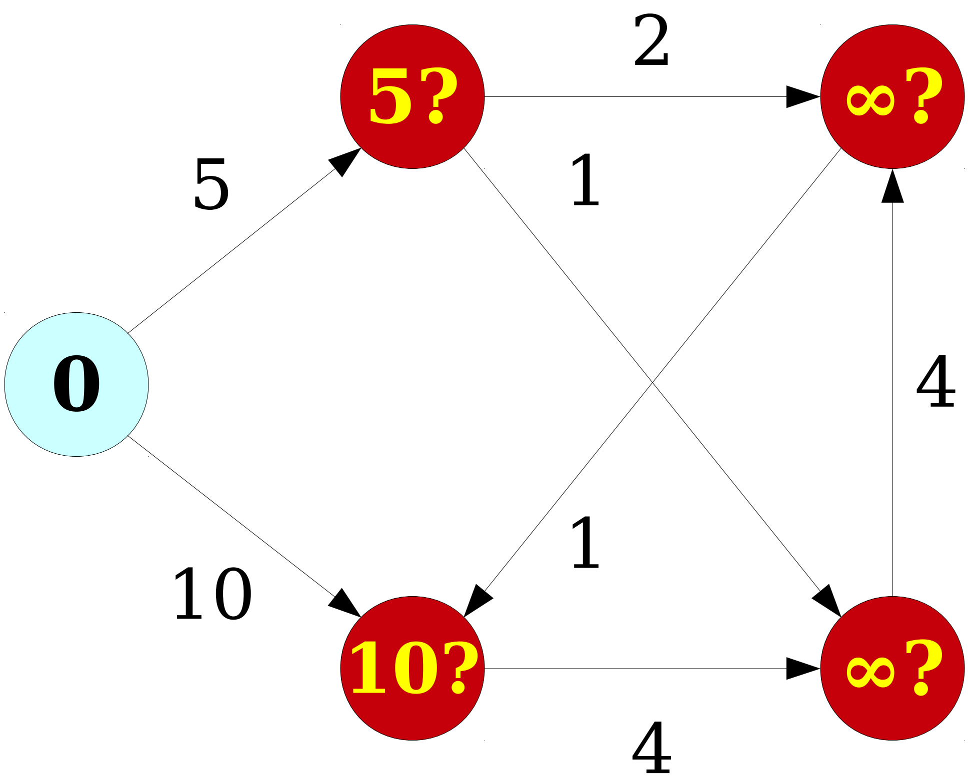
Review: Dijkstra's Algorithm

- Dijkstra's algorithm solves the single-source shortest paths (SSSP) problem in graphs with nonnegative edge weights.



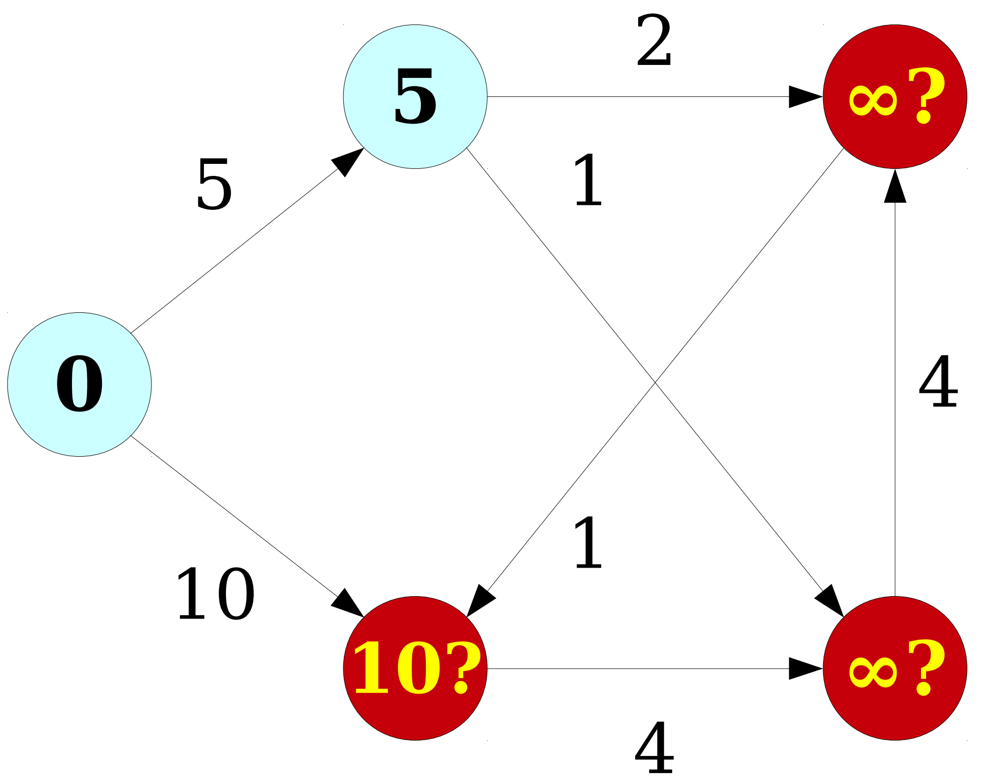
Review: Dijkstra's Algorithm

- Dijkstra's algorithm solves the single-source shortest paths (SSSP) problem in graphs with nonnegative edge weights.



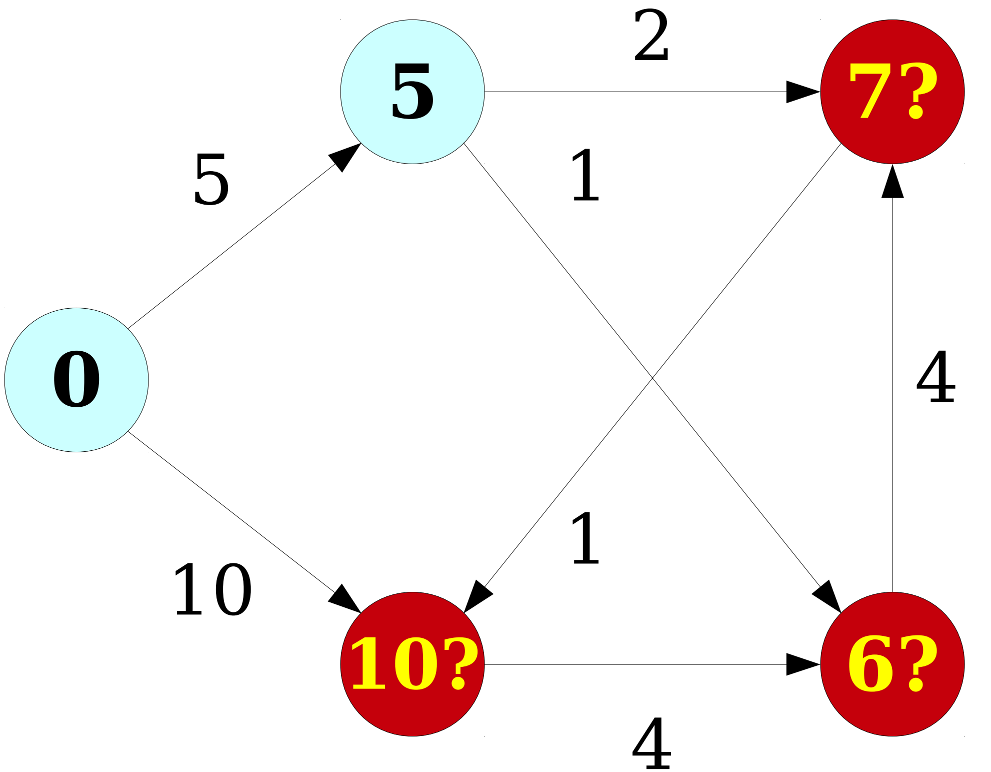
Review: Dijkstra's Algorithm

- Dijkstra's algorithm solves the single-source shortest paths (SSSP) problem in graphs with nonnegative edge weights.



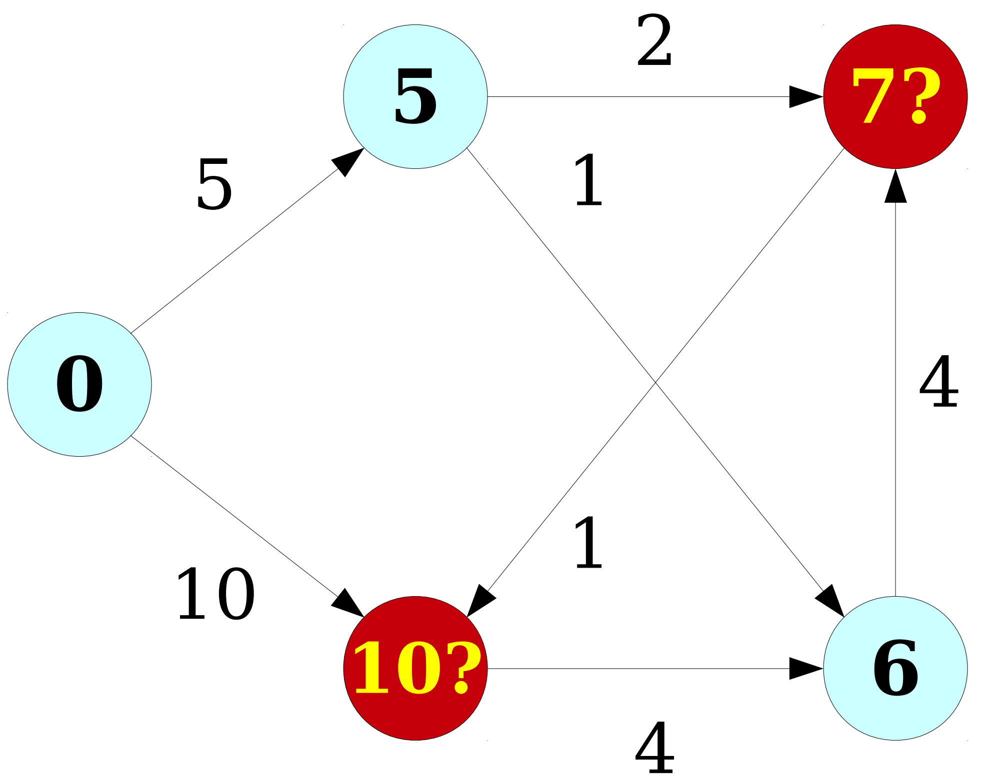
Review: Dijkstra's Algorithm

- Dijkstra's algorithm solves the single-source shortest paths (SSSP) problem in graphs with nonnegative edge weights.



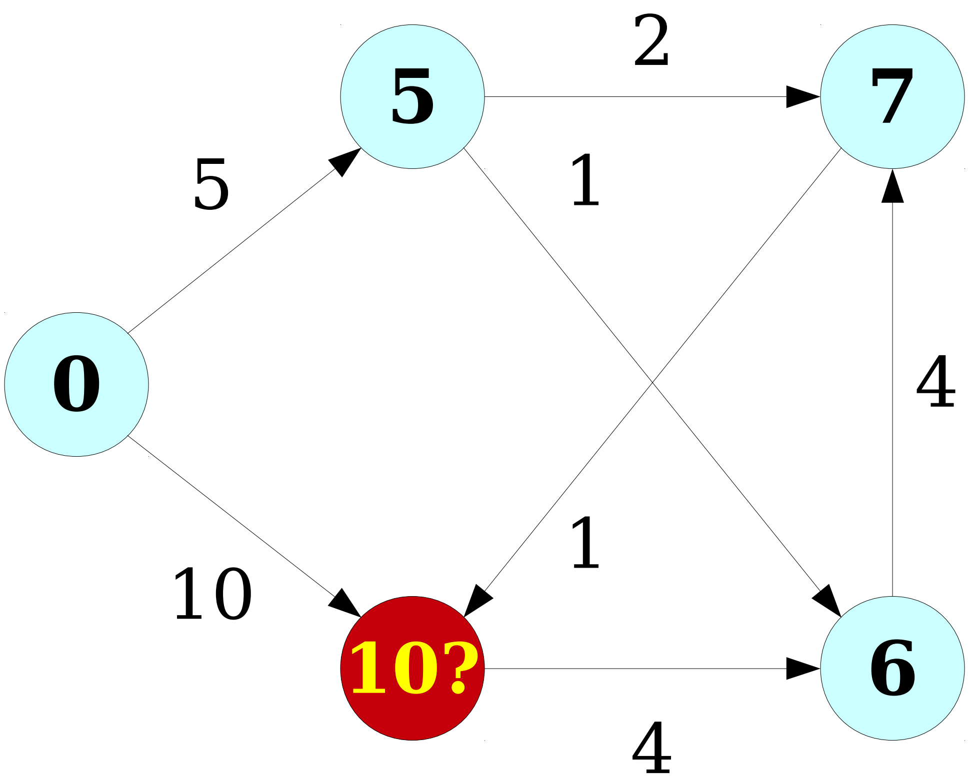
Review: Dijkstra's Algorithm

- Dijkstra's algorithm solves the single-source shortest paths (SSSP) problem in graphs with nonnegative edge weights.



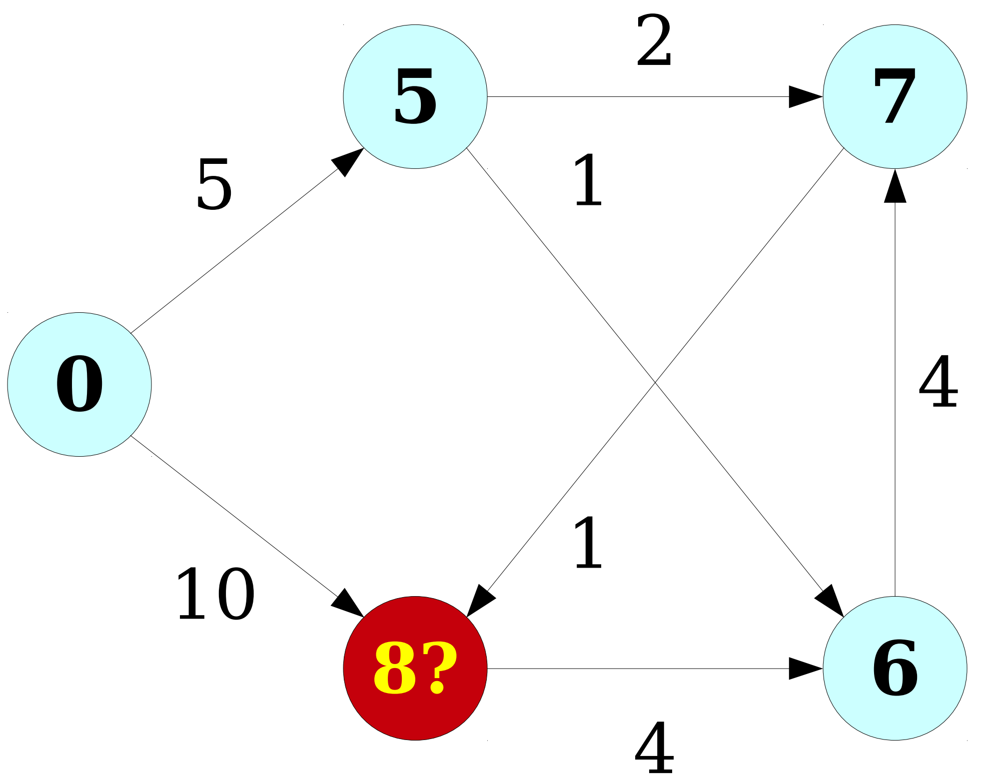
Review: Dijkstra's Algorithm

- Dijkstra's algorithm solves the single-source shortest paths (SSSP) problem in graphs with nonnegative edge weights.



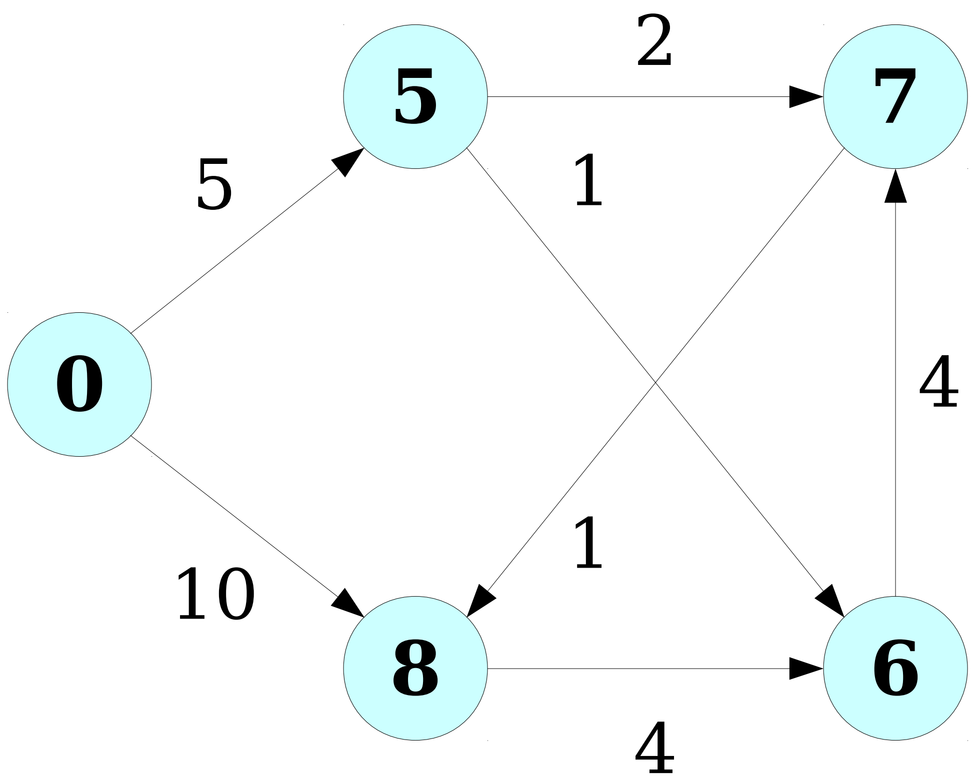
Review: Dijkstra's Algorithm

- Dijkstra's algorithm solves the single-source shortest paths (SSSP) problem in graphs with nonnegative edge weights.



Review: Dijkstra's Algorithm

- Dijkstra's algorithm solves the single-source shortest paths (SSSP) problem in graphs with nonnegative edge weights.

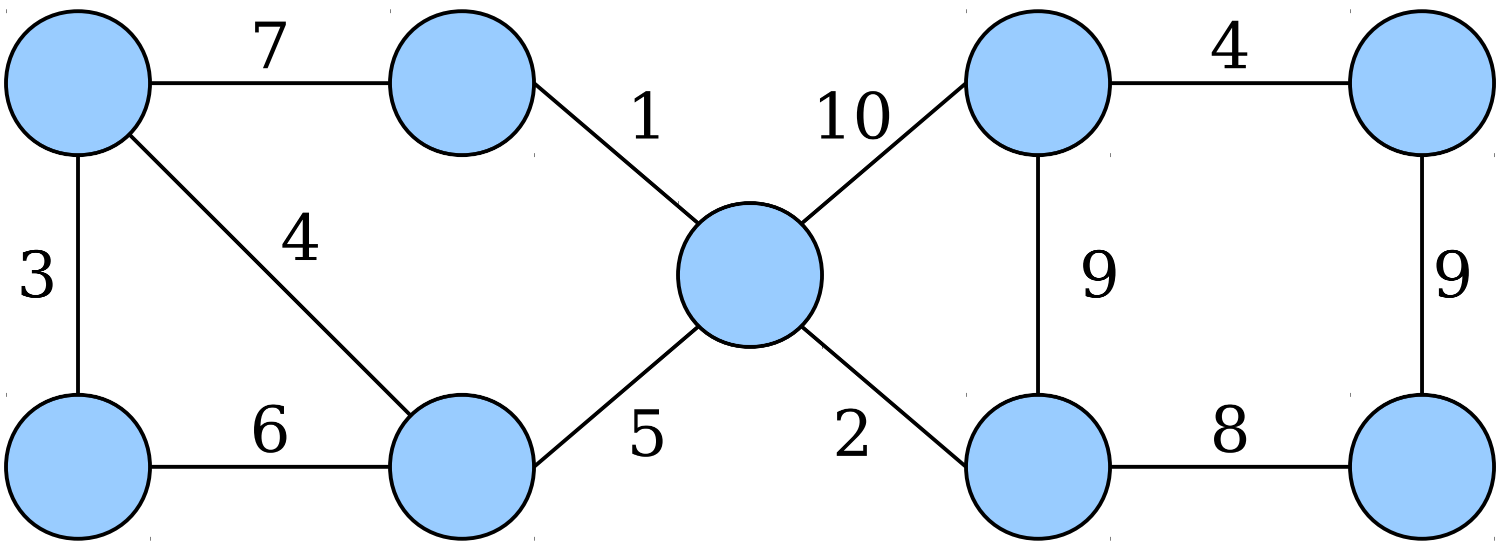


Dijkstra and Priority Queues

- At each step of Dijkstra's algorithm, we need to do the following:
  - Find the node at v minimum distance from s.
  - Update the candidate distances of all the nodes connected to v. (Distances only decrease in this step.)
- This first step sounds like an ***extract-min*** on a priority queue.
- How would we implement the second step?

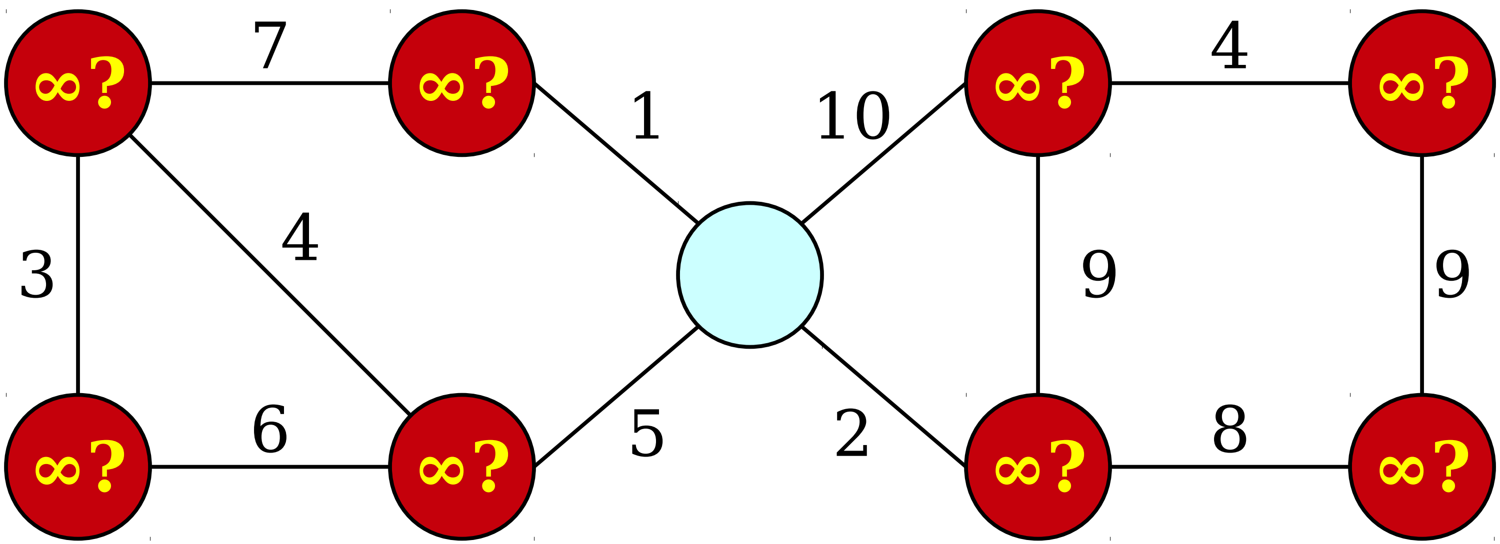
Review: Prim's Algorithm

- Prim's algorithm solves the minimum spanning tree (MST) problem in undirected graphs.



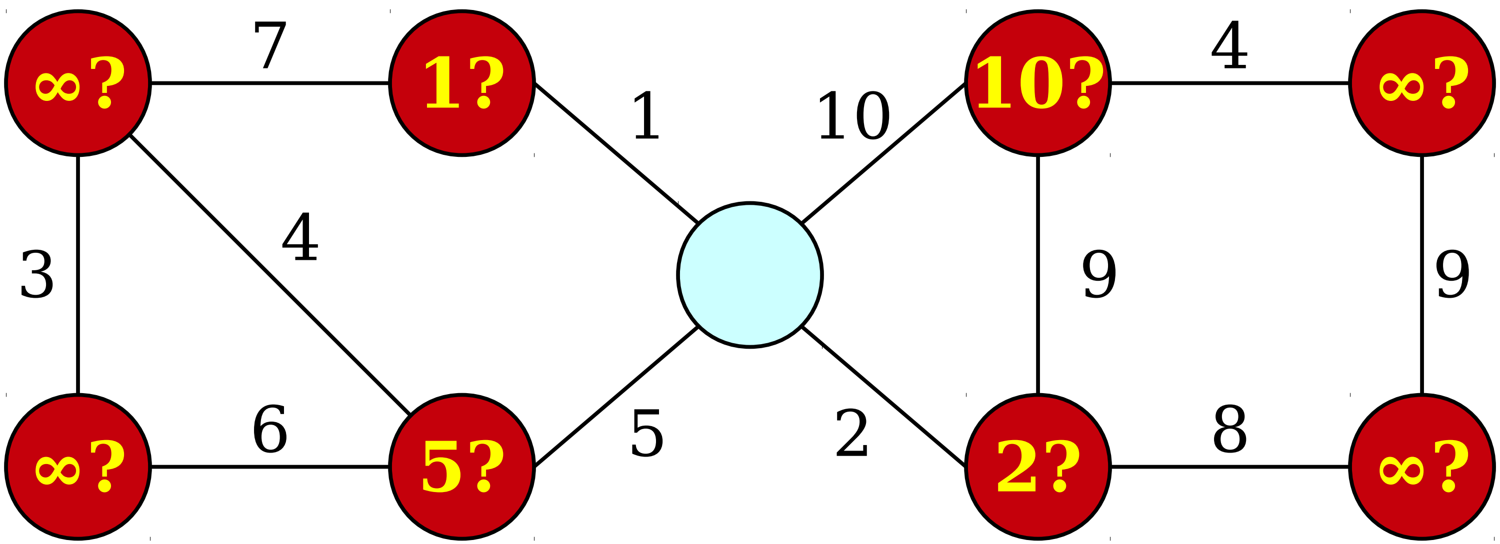
Review: Prim's Algorithm

- Prim's algorithm solves the minimum spanning tree (MST) problem in undirected graphs.



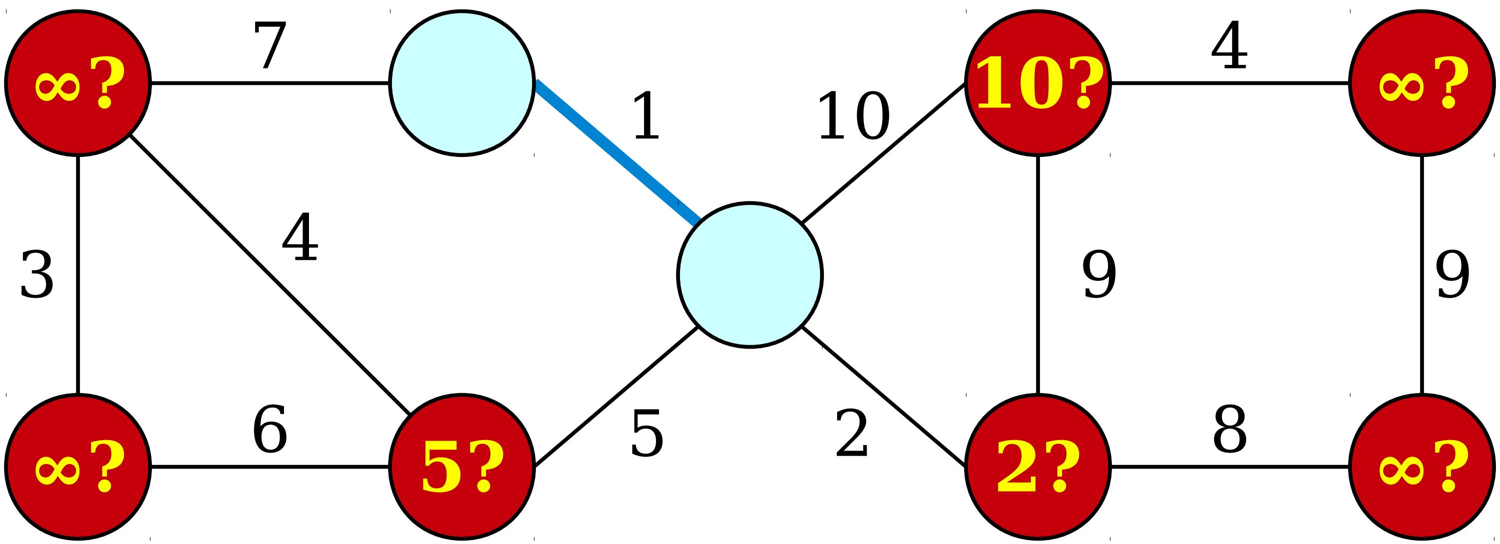
Review: Prim's Algorithm

- Prim's algorithm solves the minimum spanning tree (MST) problem in undirected graphs.



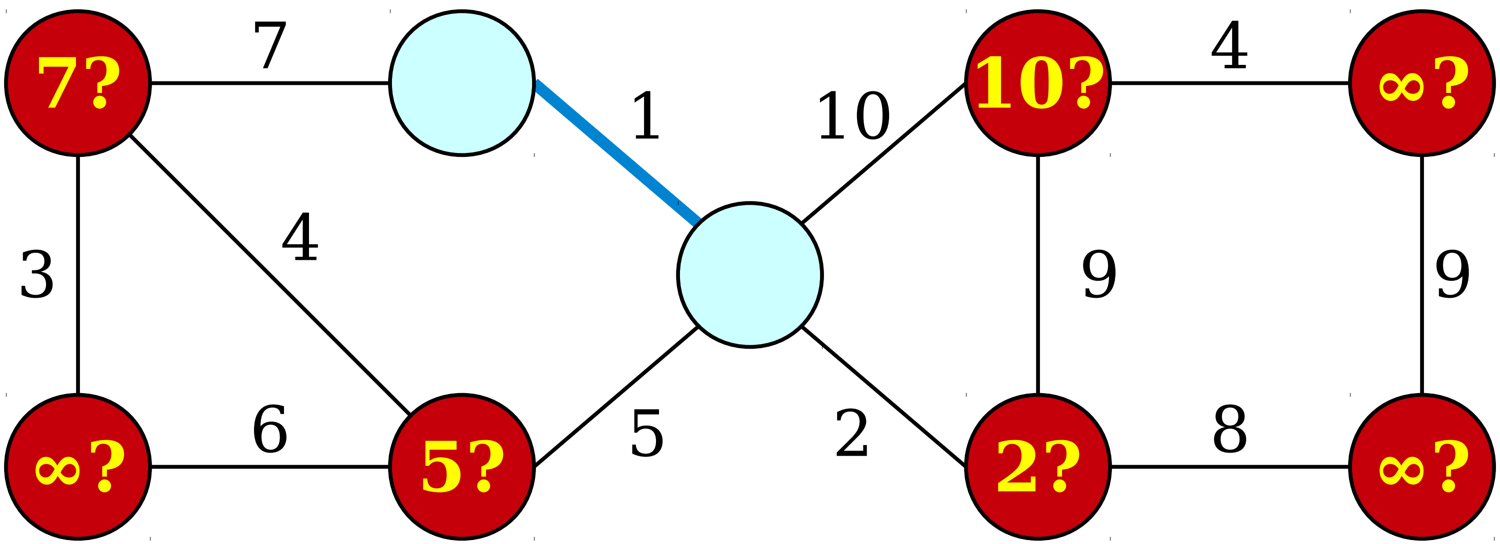
Review: Prim's Algorithm

- Prim's algorithm solves the minimum spanning tree (MST) problem in undirected graphs.



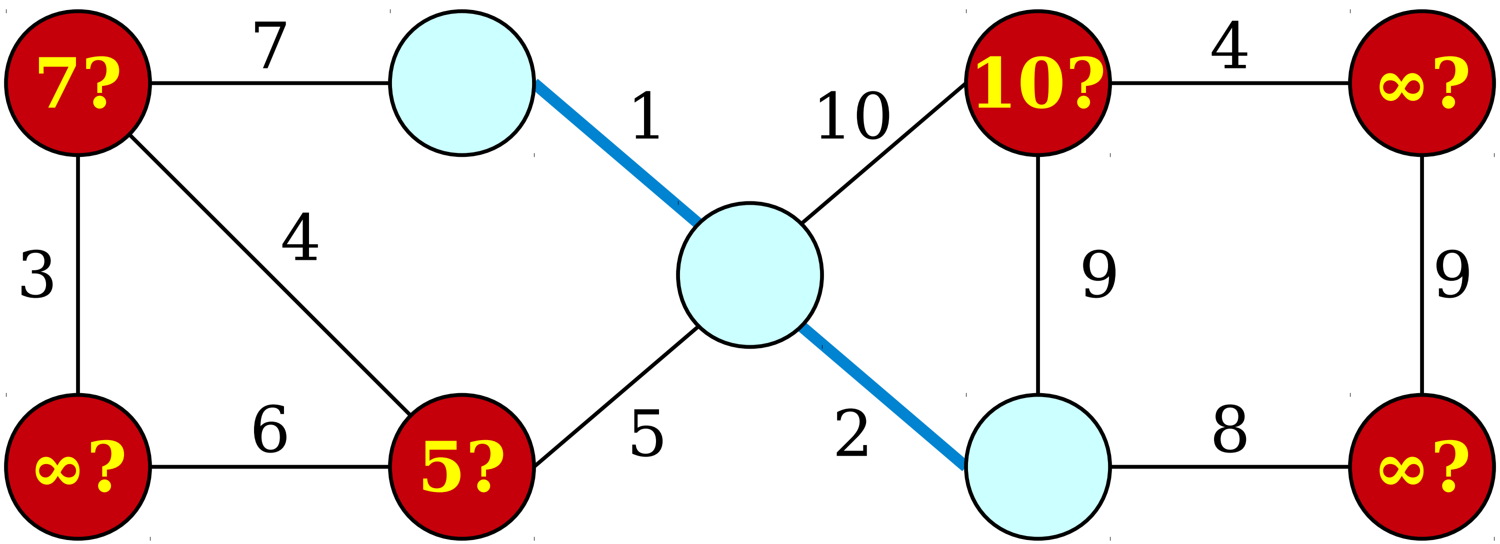
Review: Prim's Algorithm

- Prim's algorithm solves the minimum spanning tree (MST) problem in undirected graphs.



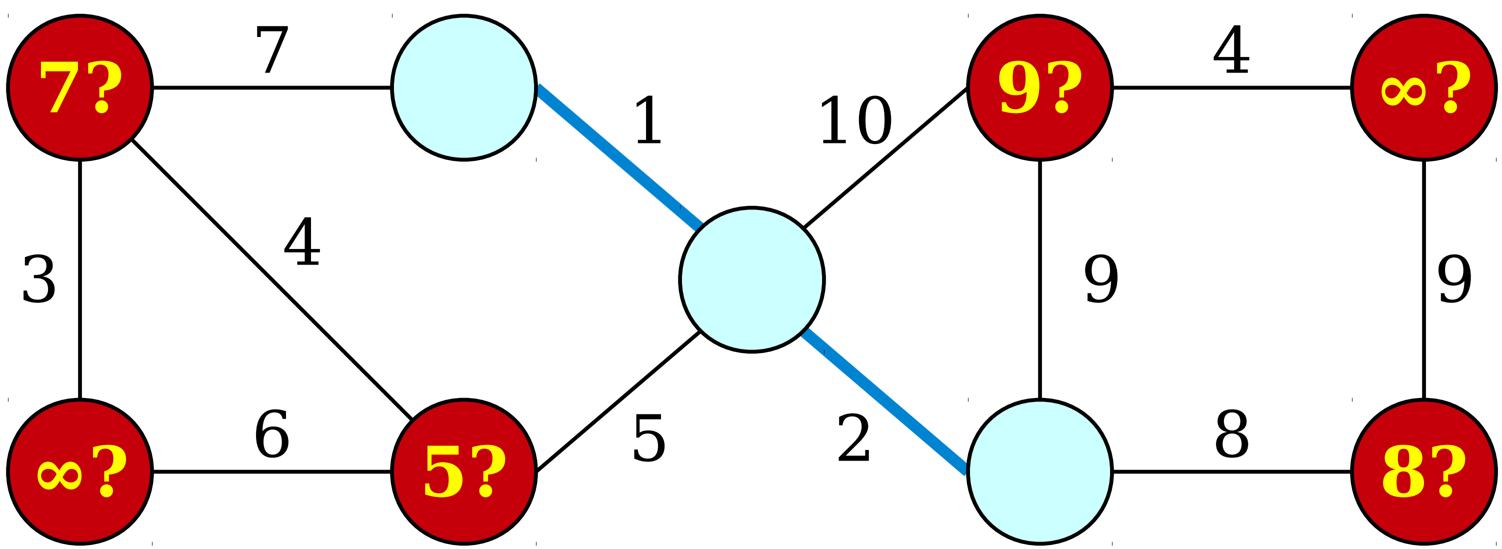
Review: Prim's Algorithm

- Prim's algorithm solves the minimum spanning tree (MST) problem in undirected graphs.



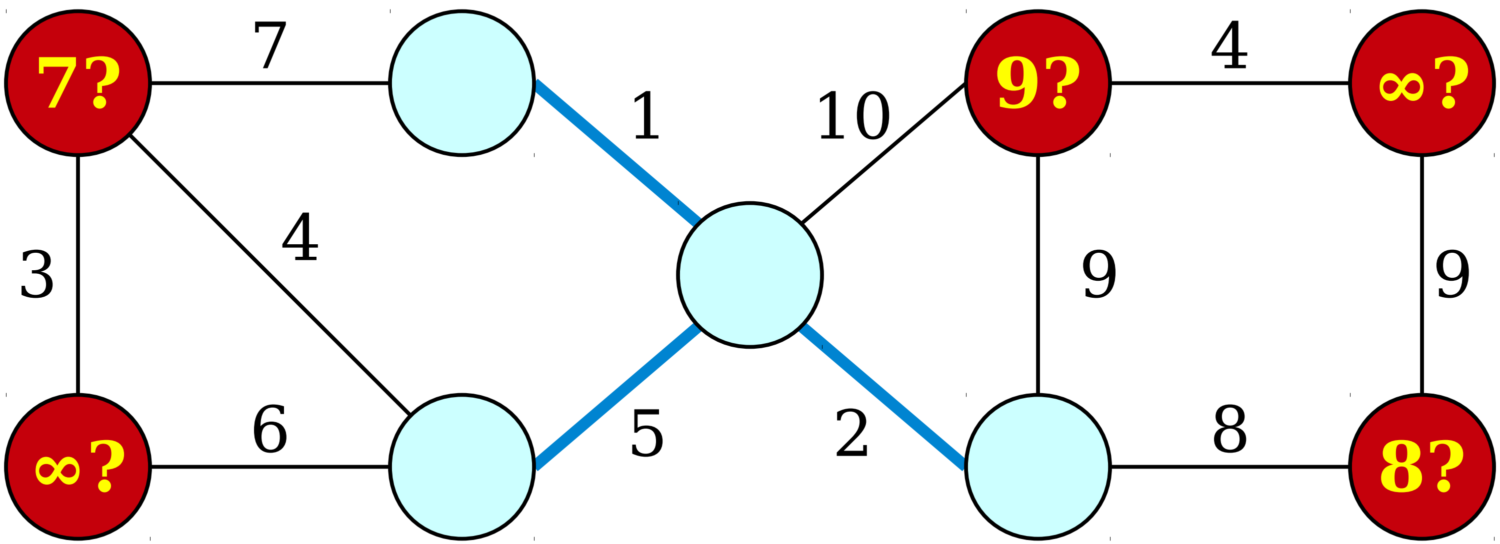
Review: Prim's Algorithm

- Prim's algorithm solves the minimum spanning tree (MST) problem in undirected graphs.



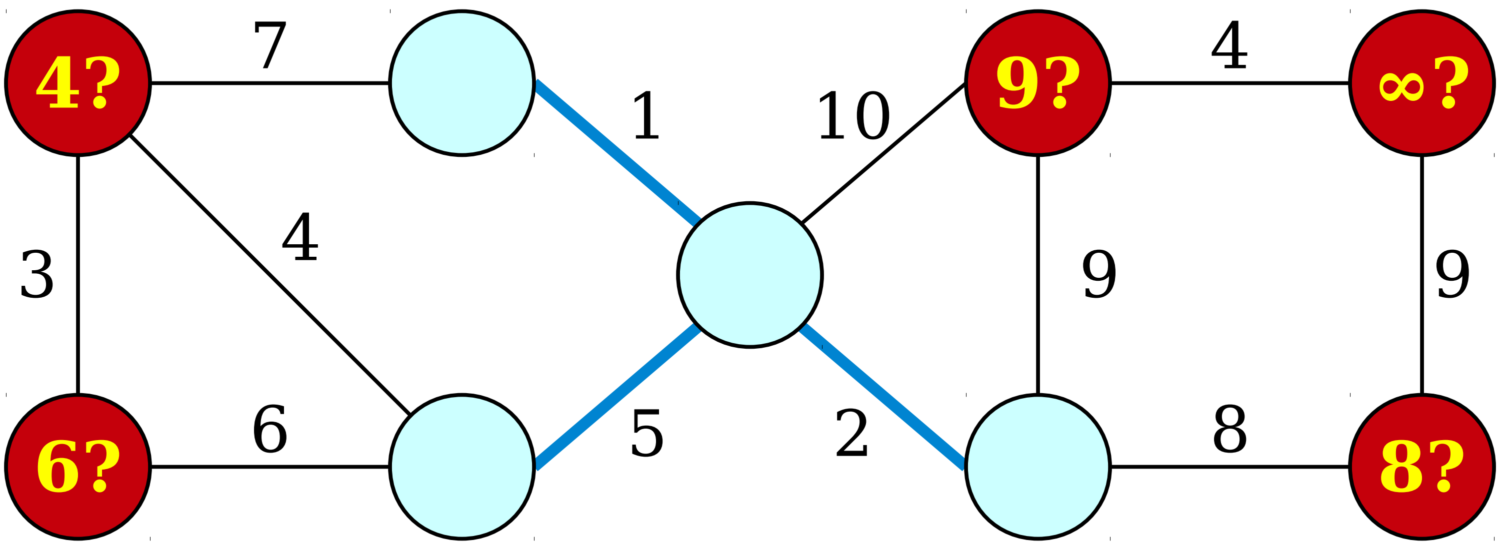
Review: Prim's Algorithm

- Prim's algorithm solves the minimum spanning tree (MST) problem in undirected graphs.



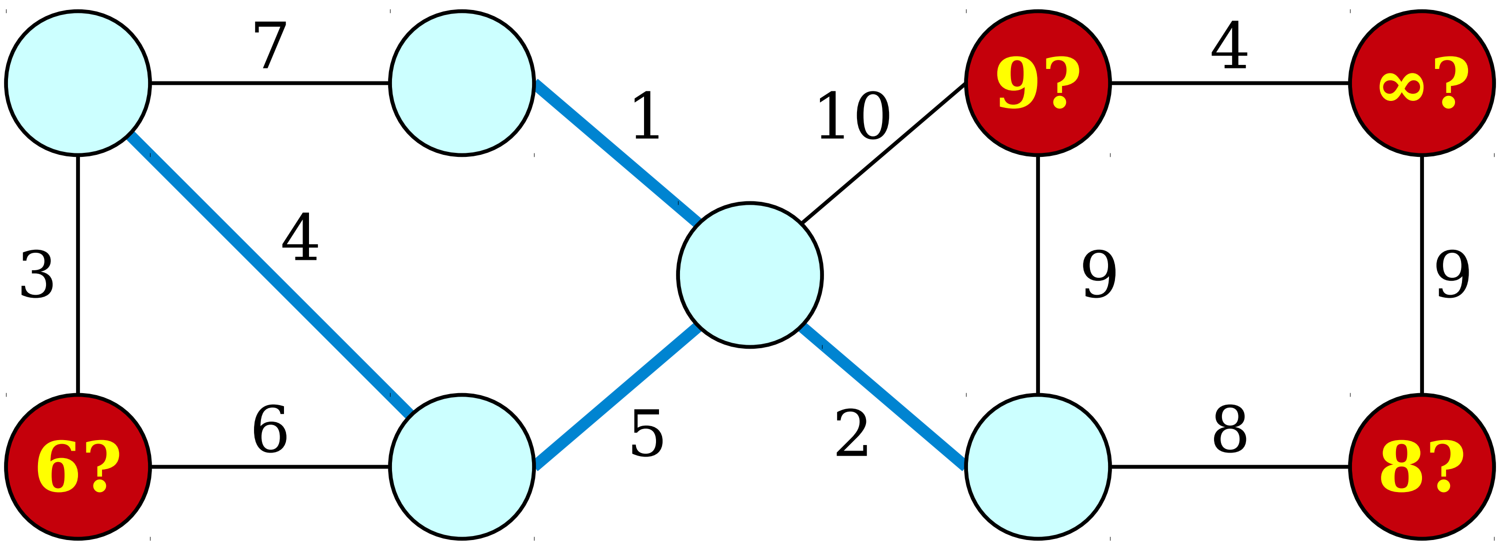
Review: Prim's Algorithm

- Prim's algorithm solves the minimum spanning tree (MST) problem in undirected graphs.



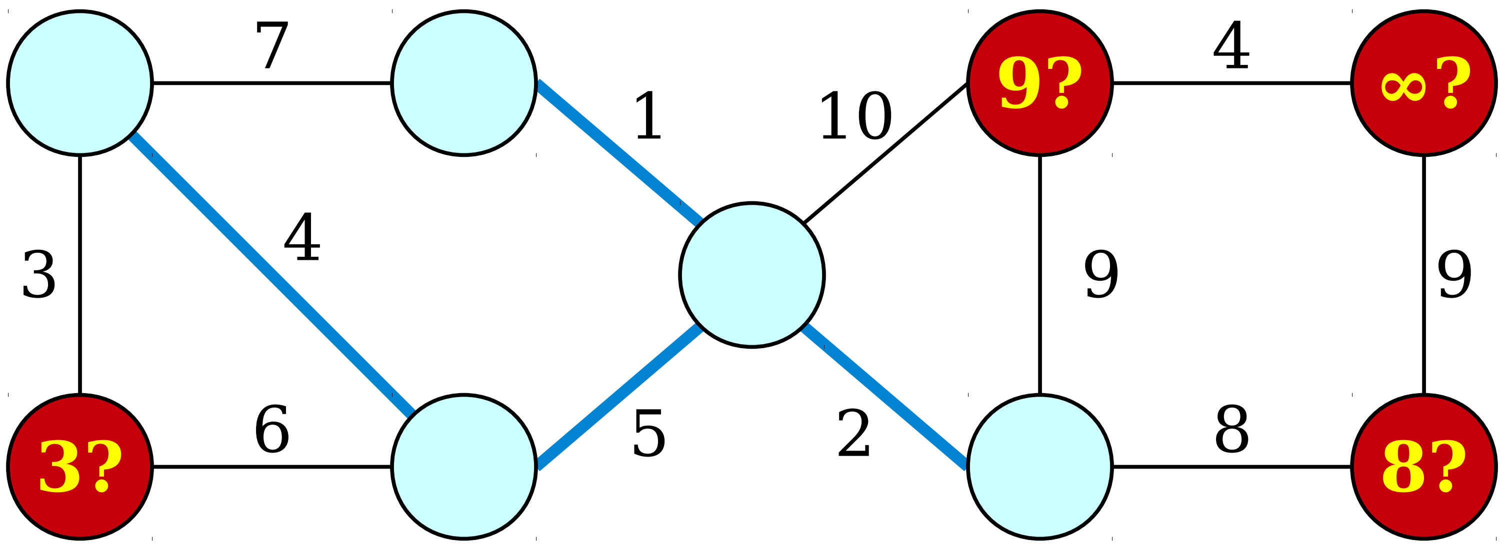
Review: Prim's Algorithm

- Prim's algorithm solves the minimum spanning tree (MST) problem in undirected graphs.



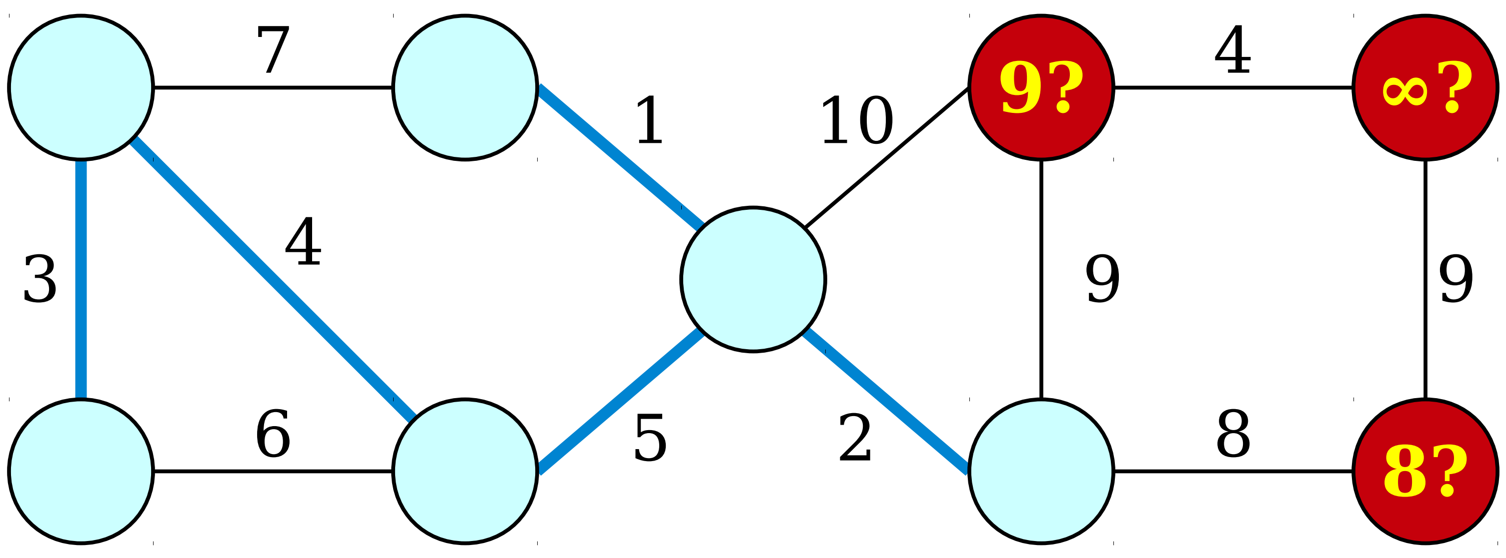
Review: Prim's Algorithm

- Prim's algorithm solves the minimum spanning tree (MST) problem in undirected graphs.



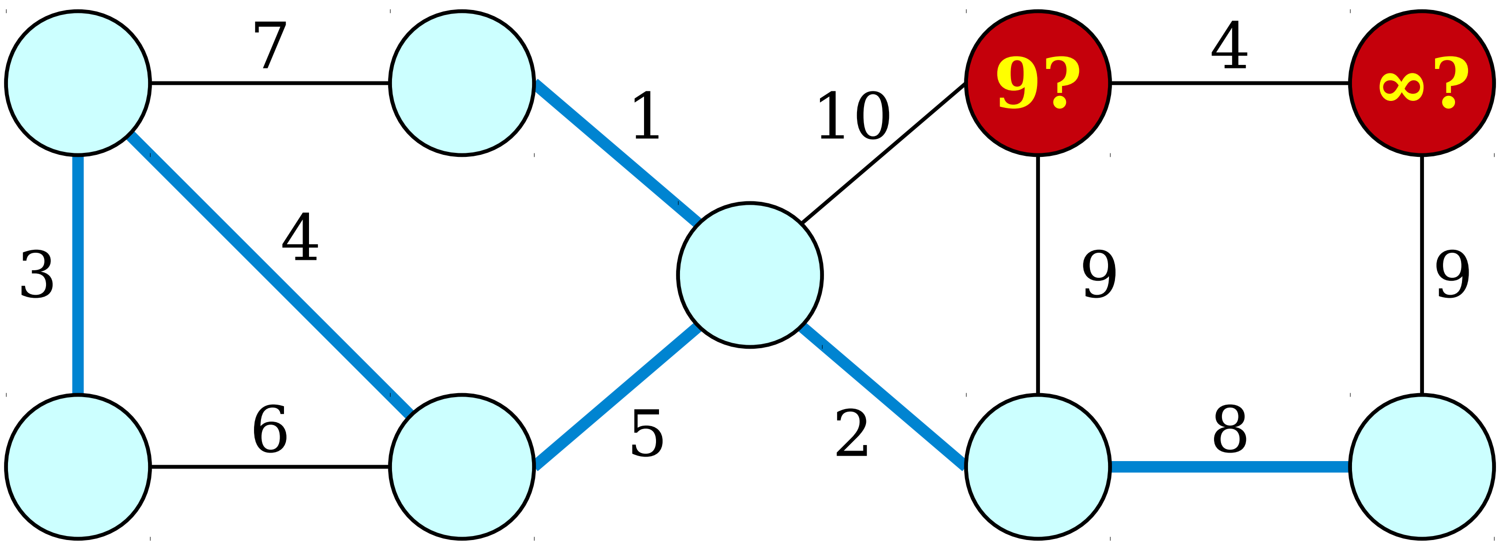
Review: Prim's Algorithm

- Prim's algorithm solves the minimum spanning tree (MST) problem in undirected graphs.



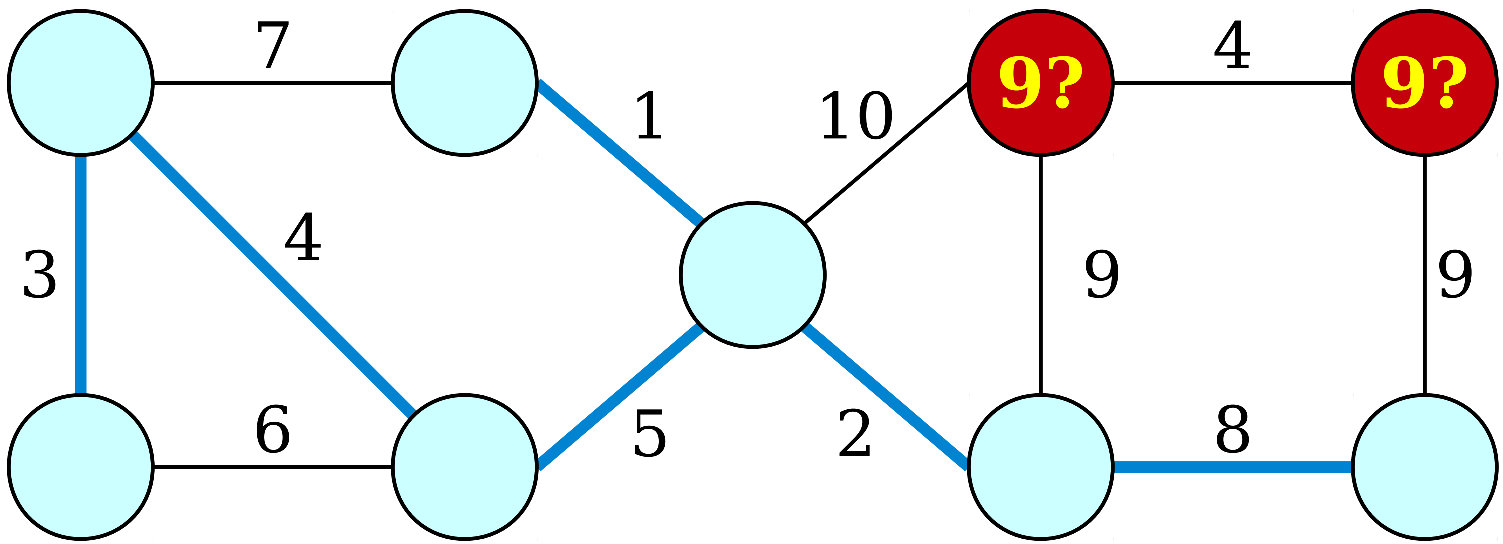
Review: Prim's Algorithm

- Prim's algorithm solves the minimum spanning tree (MST) problem in undirected graphs.



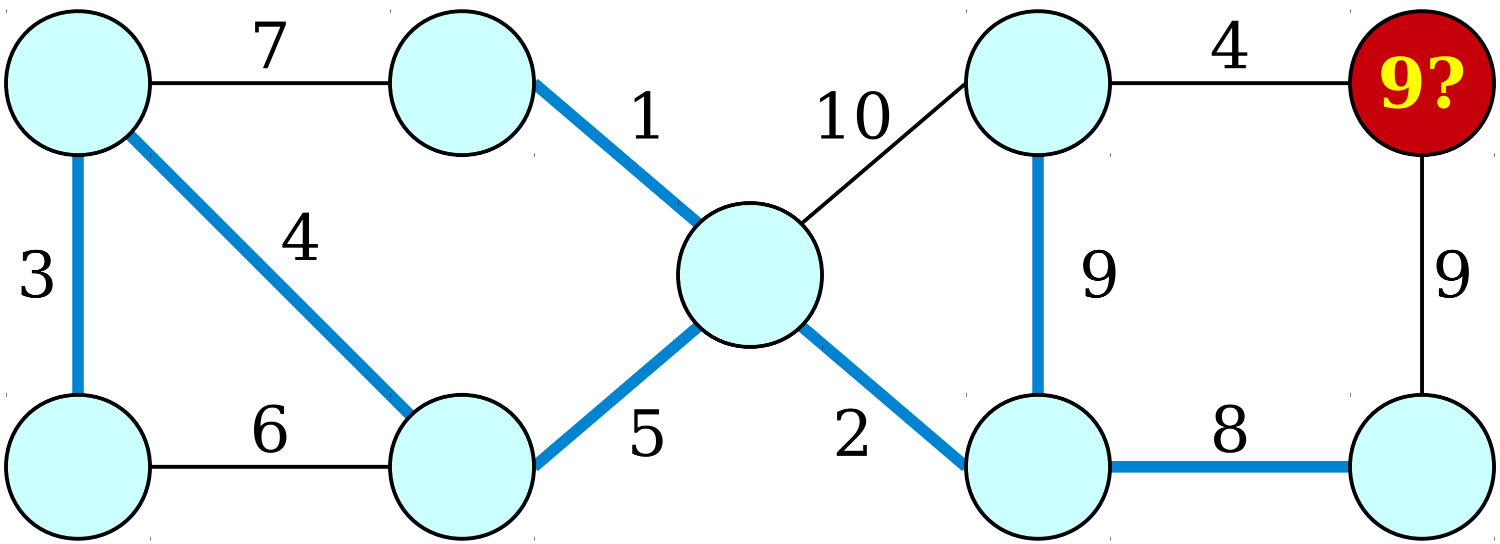
Review: Prim's Algorithm

- Prim's algorithm solves the minimum spanning tree (MST) problem in undirected graphs.



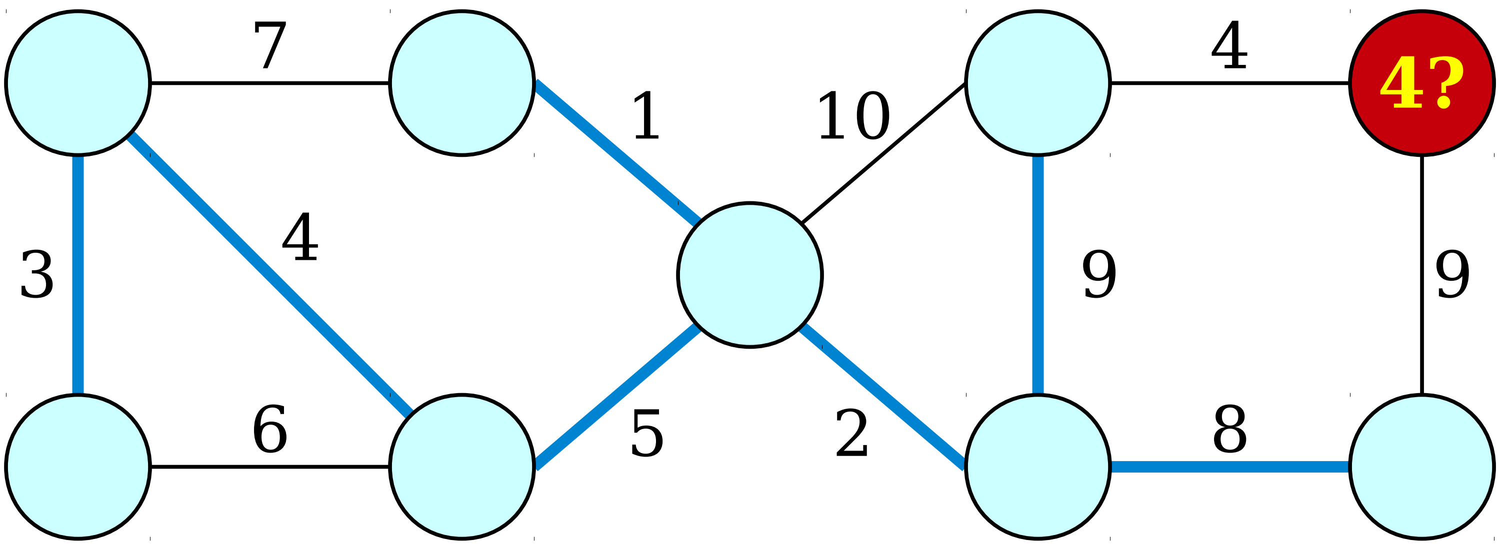
Review: Prim's Algorithm

- Prim's algorithm solves the minimum spanning tree (MST) problem in undirected graphs.



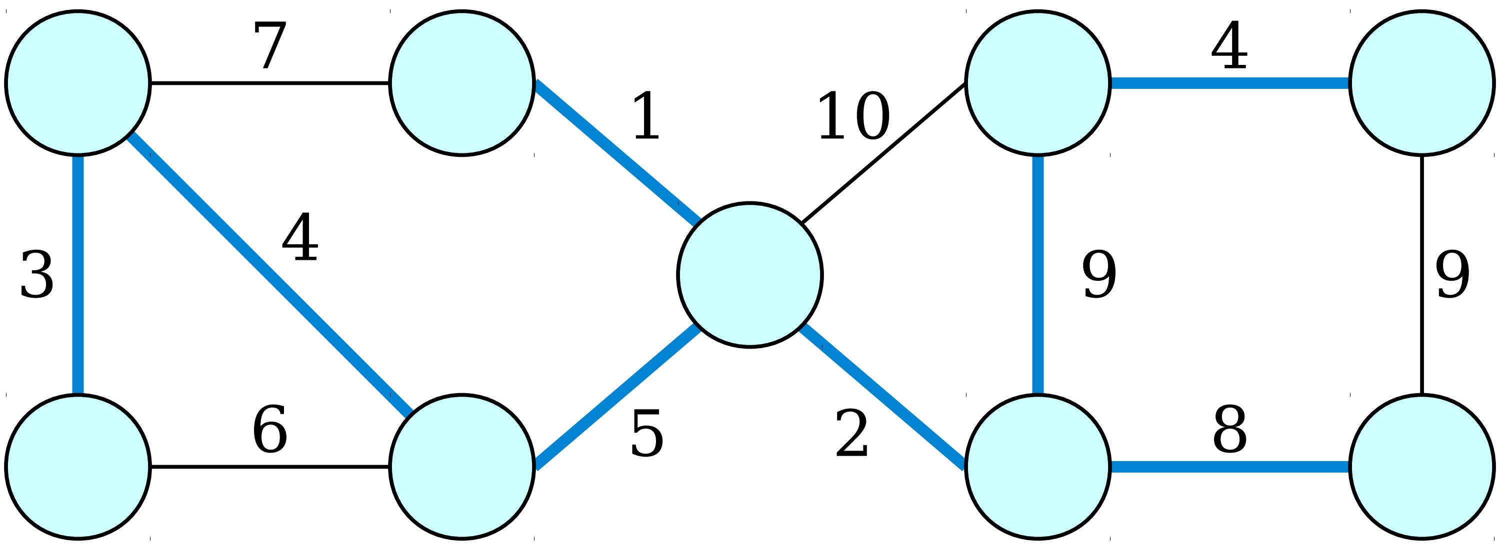
Review: Prim's Algorithm

- Prim's algorithm solves the minimum spanning tree (MST) problem in undirected graphs.



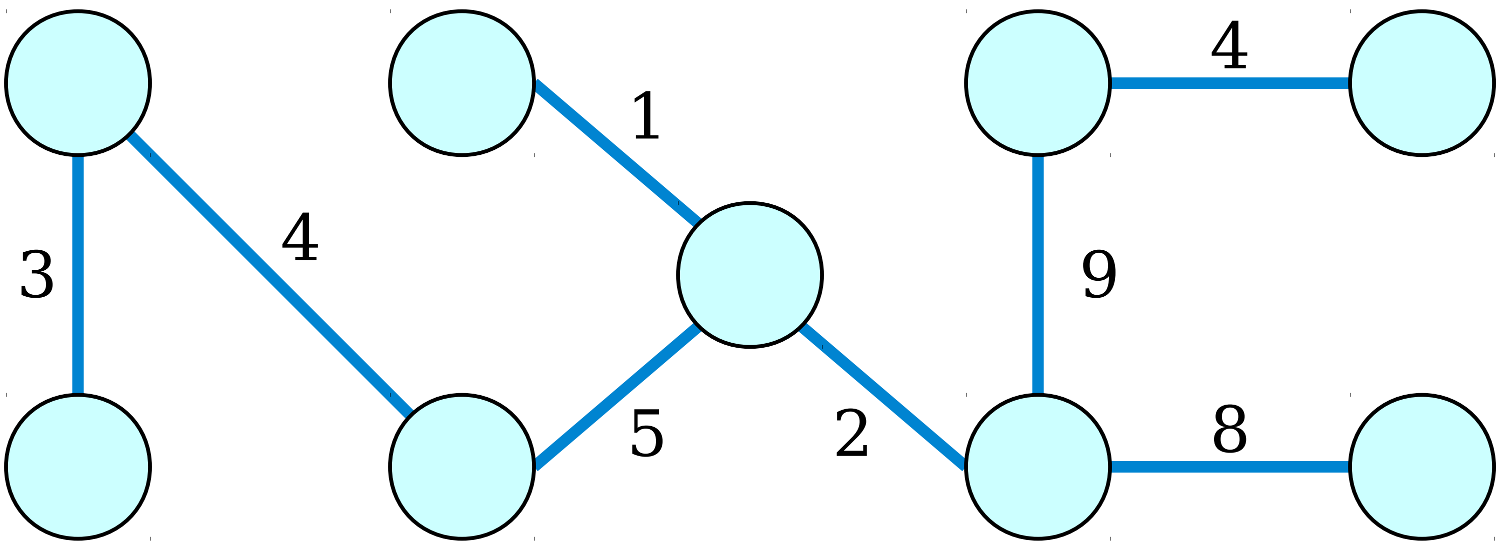
Review: Prim's Algorithm

- Prim's algorithm solves the minimum spanning tree (MST) problem in undirected graphs.



Review: Prim's Algorithm

- Prim's algorithm solves the minimum spanning tree (MST) problem in undirected graphs.



Prim and Priority Queues

- At each step of Prim's algorithm, we need to do the following:
  - Find the node *v* outside of the spanning tree with the lowest-cost connection to the tree.
  - Update the candidate distances from *v* to nodes outside the set *S*.
- This first step sounds like an ***extract-min*** on a priority queue.
- How would we implement the second step?

The ***decrease-key*** Operation

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

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

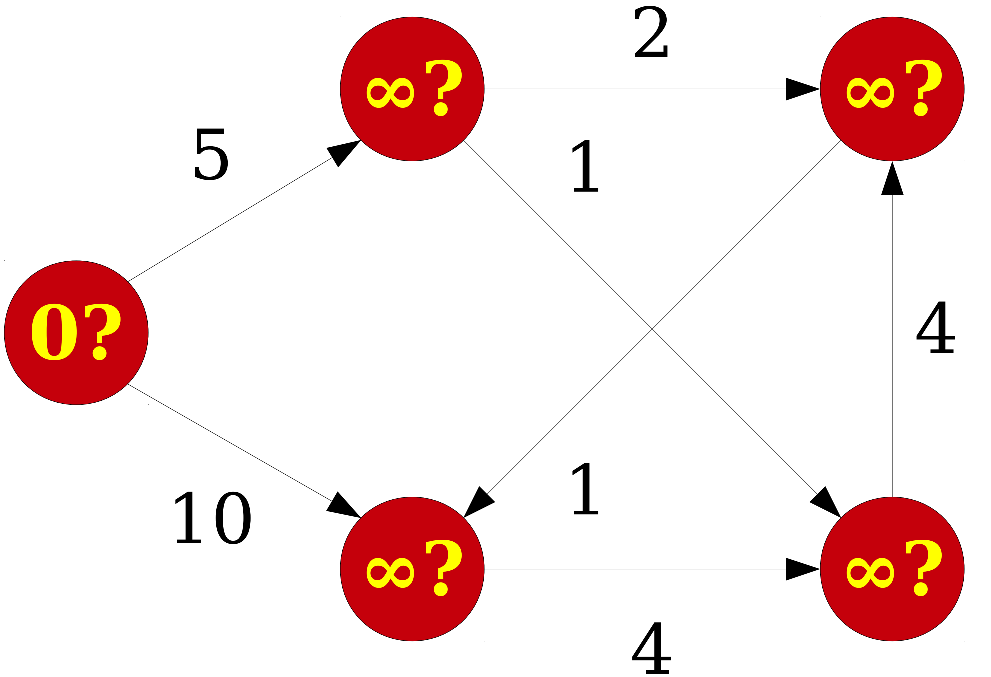
- This operation is crucial in efficient implementations of Dijkstra's algorithm and Prim's MST algorithm.

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.

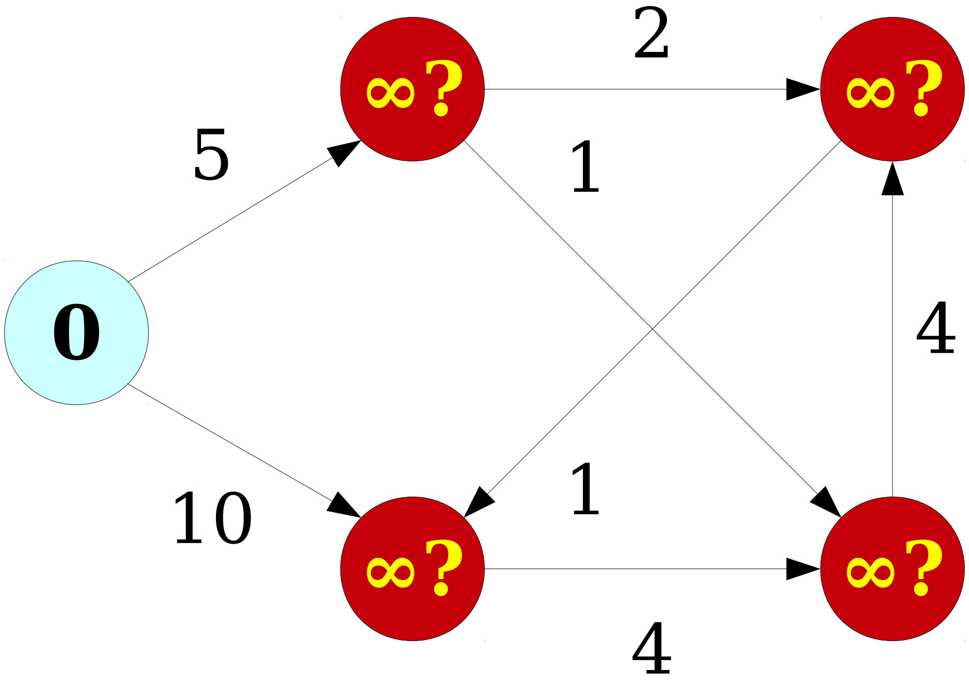
Dijkstra and ***decrease-key***

- 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.



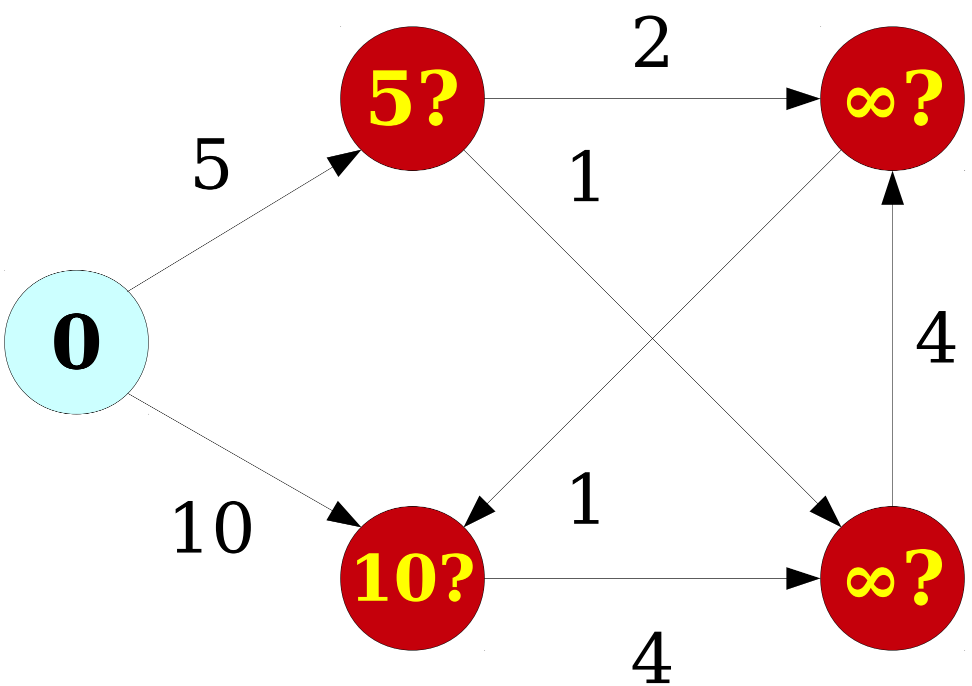
Dijkstra and ***decrease-key***

- 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 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

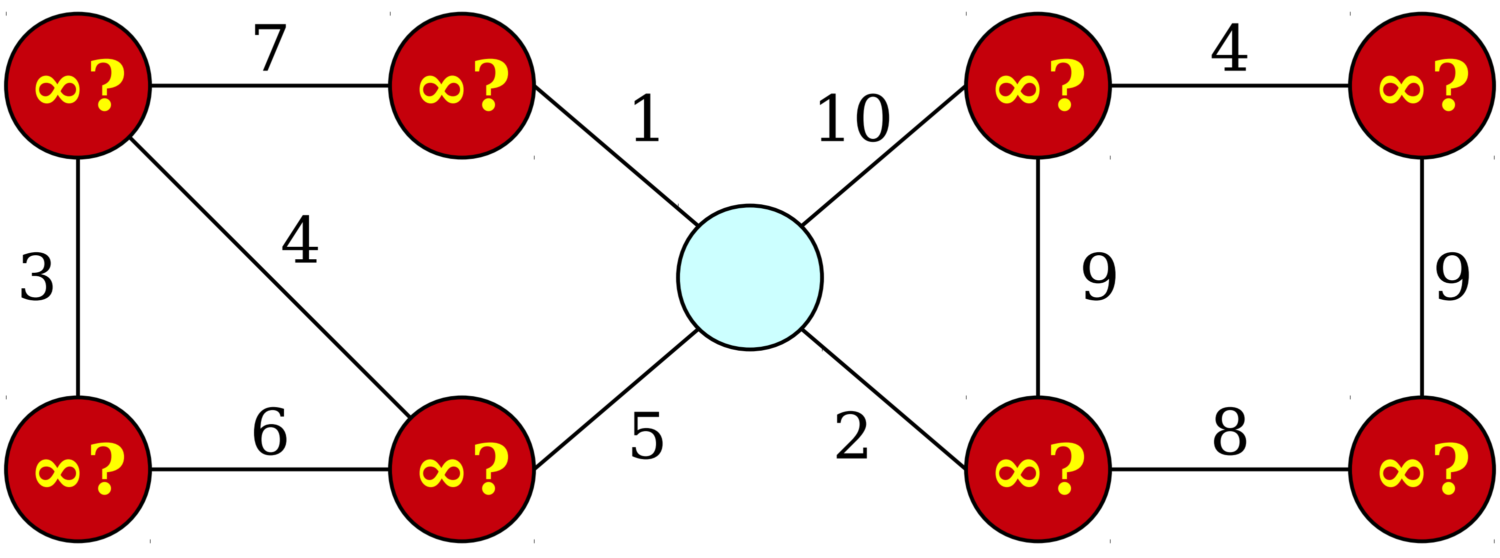
$$O(n\, T_{enq} + n\, T_{ext} + m\, T_{dec})$$

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.

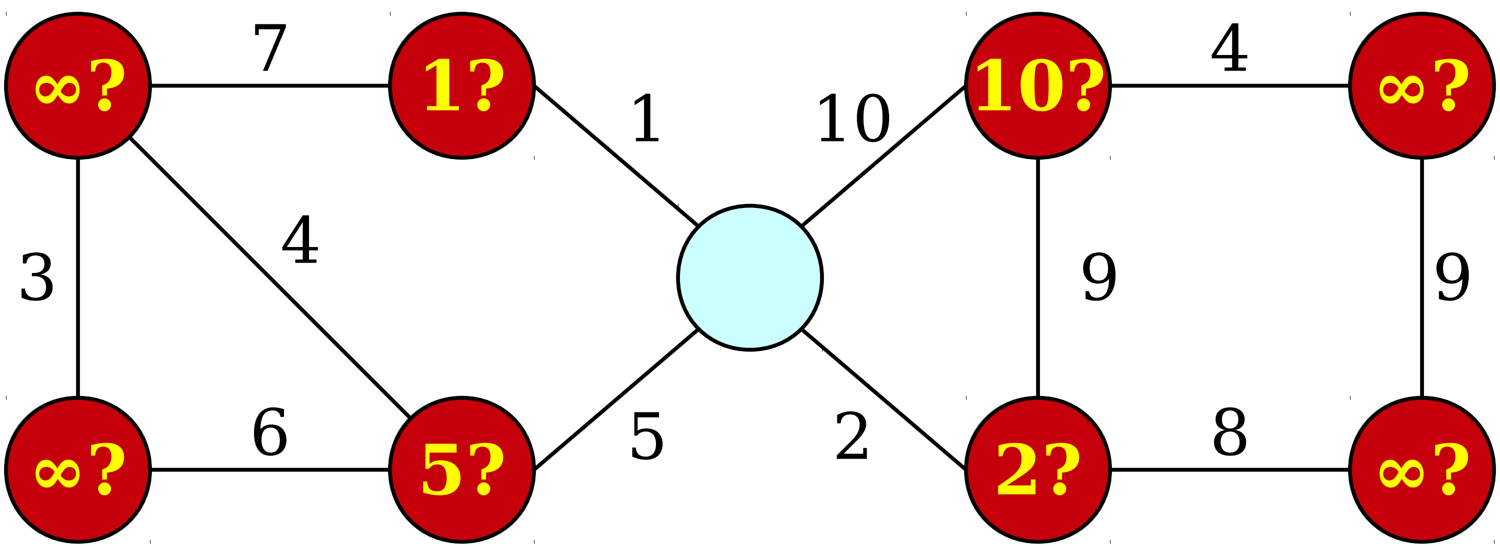
Prim and ***decrease-key***

- 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.



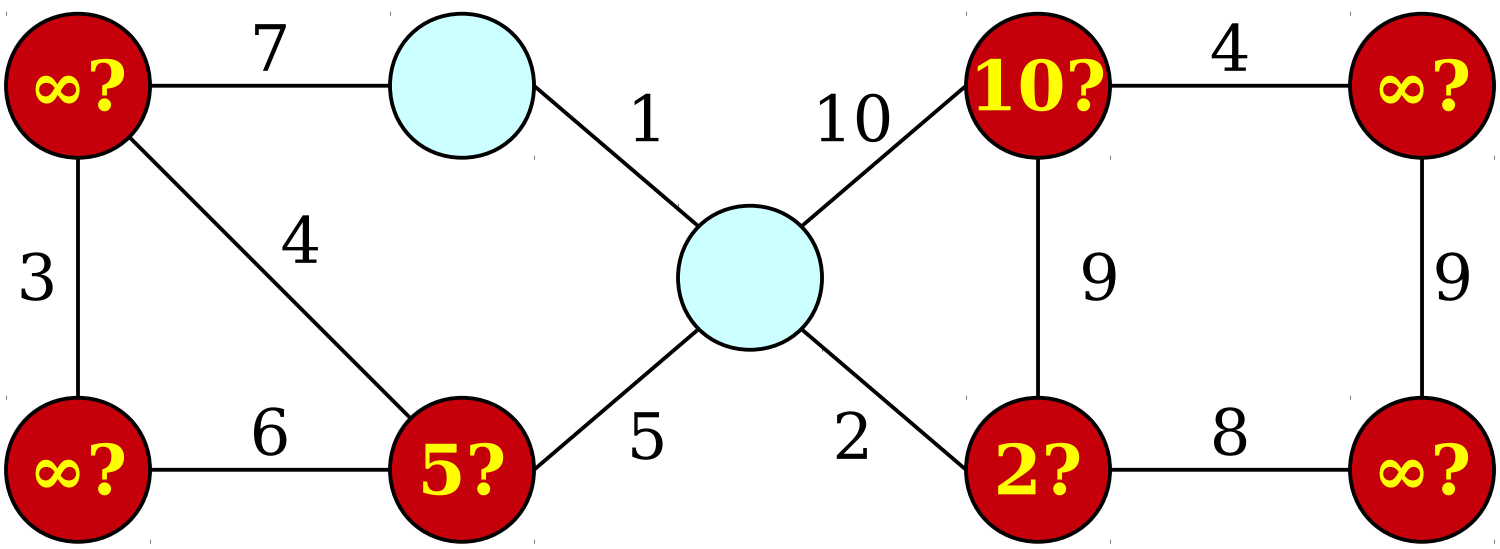
Prim and ***decrease-key***

- 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.



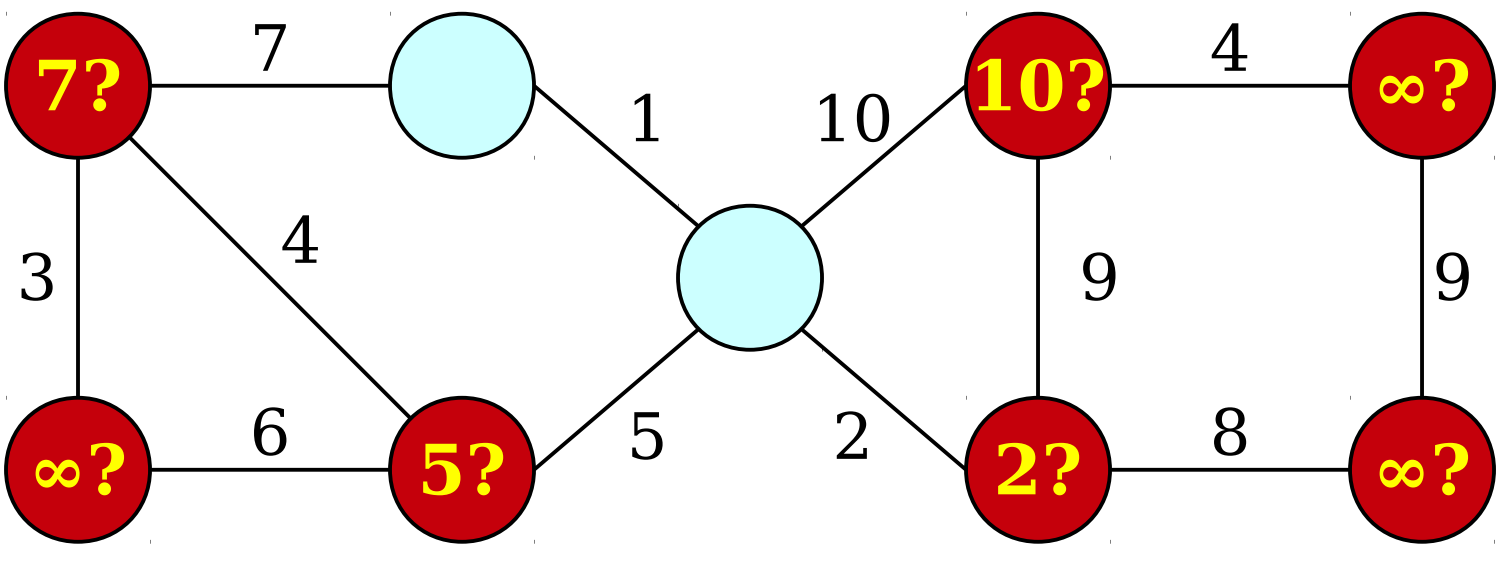
Prim and ***decrease-key***

- 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 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

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

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:

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

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

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

Standard Approaches

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

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

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

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

Where We're Going

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

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

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

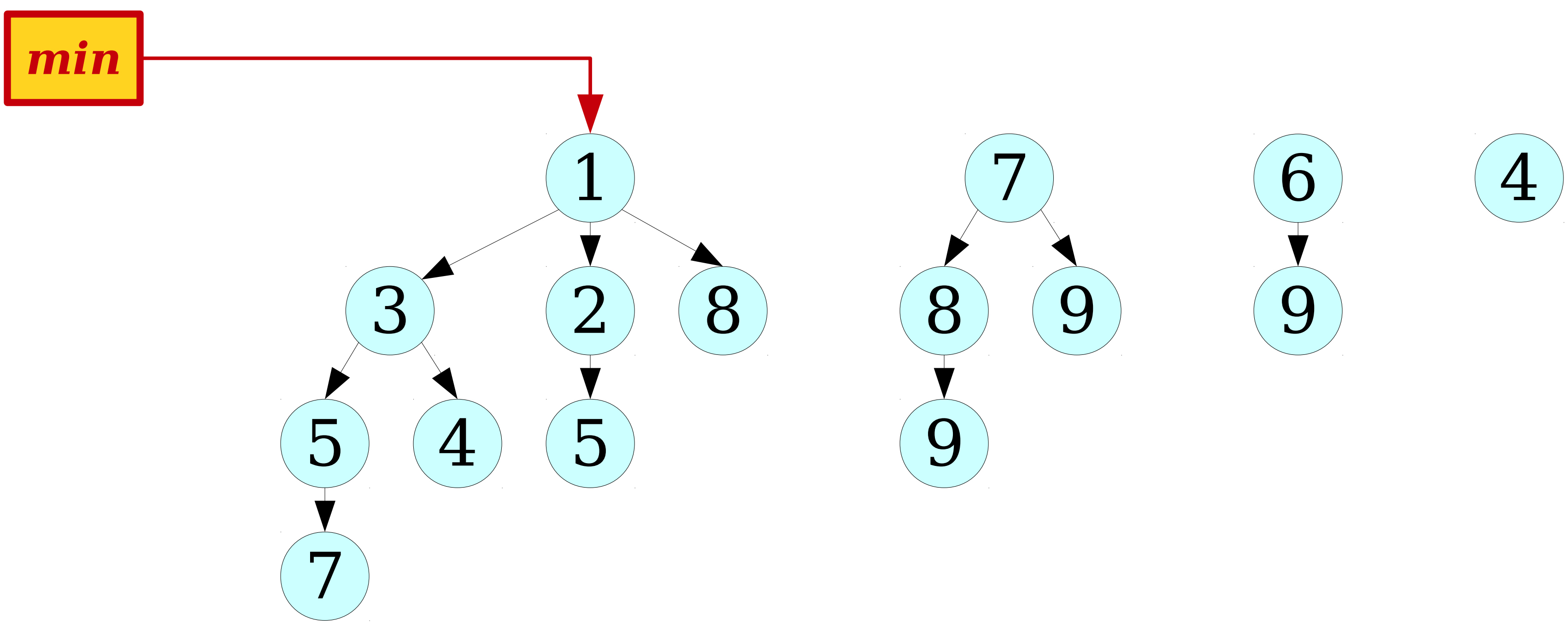
$$= \mathbf{O(m + n \log n)}$$

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

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

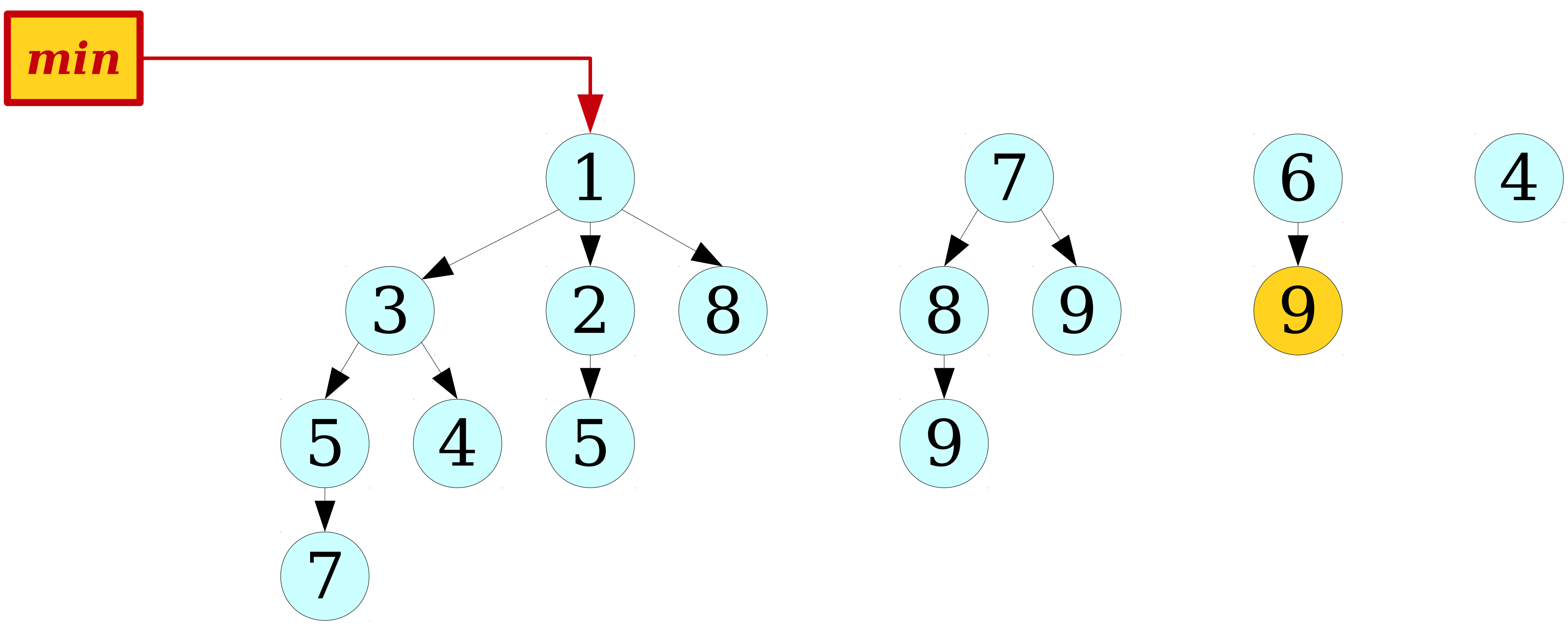
A Simple Implementation

- It is possible to implement ***decrease-key*** in time O(log *n*) using lazy binomial heaps.
- ***Idea:*** "Bubble" the element up toward the root of the binomial tree containing it and (potentially) update the *min* pointer.



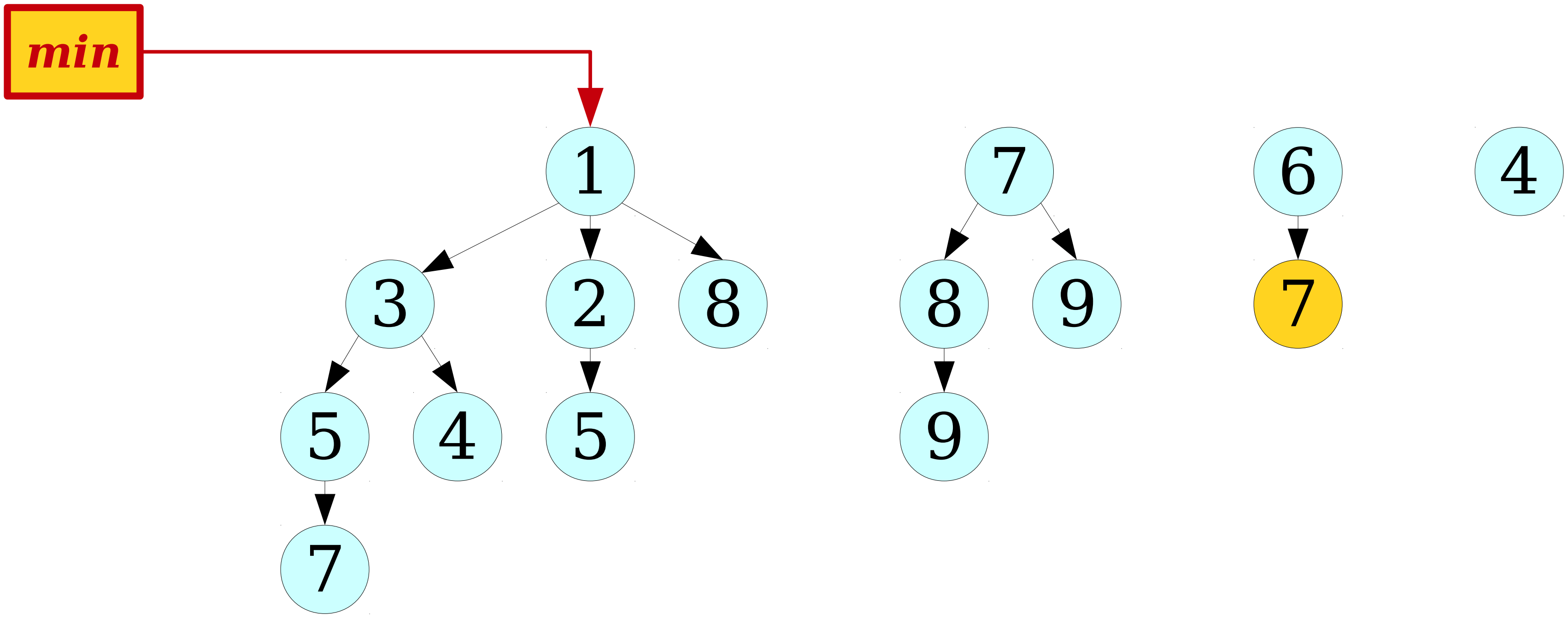
A Simple Implementation

- It is possible to implement ***decrease-key*** in time O(log *n*) using lazy binomial heaps.
- ***Idea:*** "Bubble" the element up toward the root of the binomial tree containing it and (potentially) update the *min* pointer.



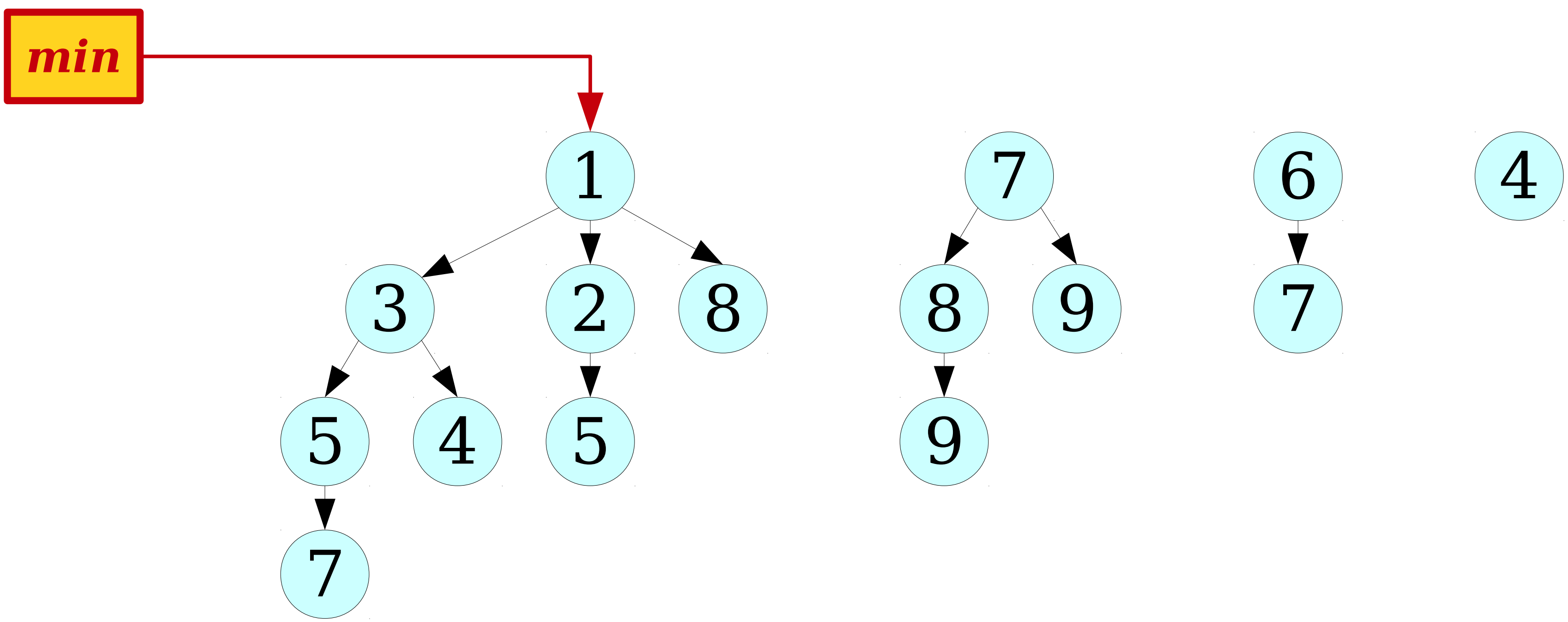
A Simple Implementation

- It is possible to implement ***decrease-key*** in time O(log *n*) using lazy binomial heaps.
- ***Idea:*** "Bubble" the element up toward the root of the binomial tree containing it and (potentially) update the *min* pointer.



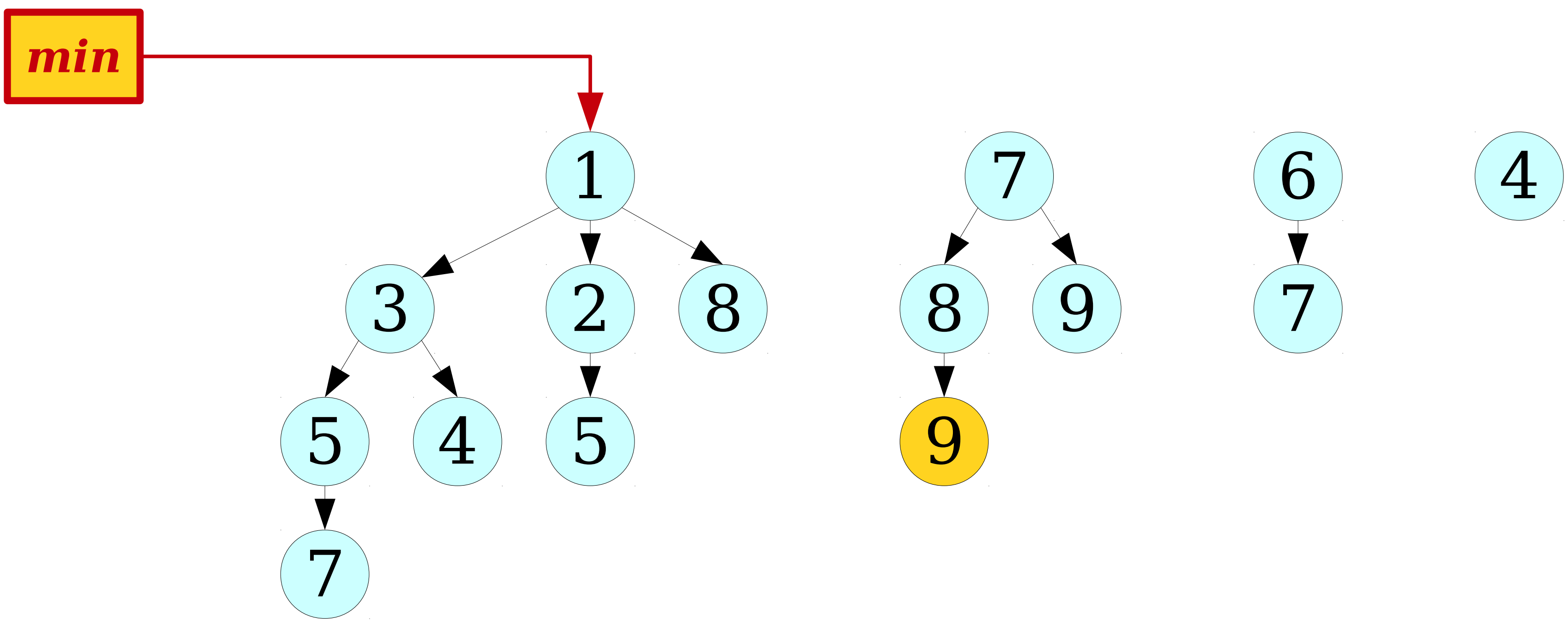
A Simple Implementation

- It is possible to implement ***decrease-key*** in time O(log *n*) using lazy binomial heaps.
- ***Idea:*** "Bubble" the element up toward the root of the binomial tree containing it and (potentially) update the *min* pointer.



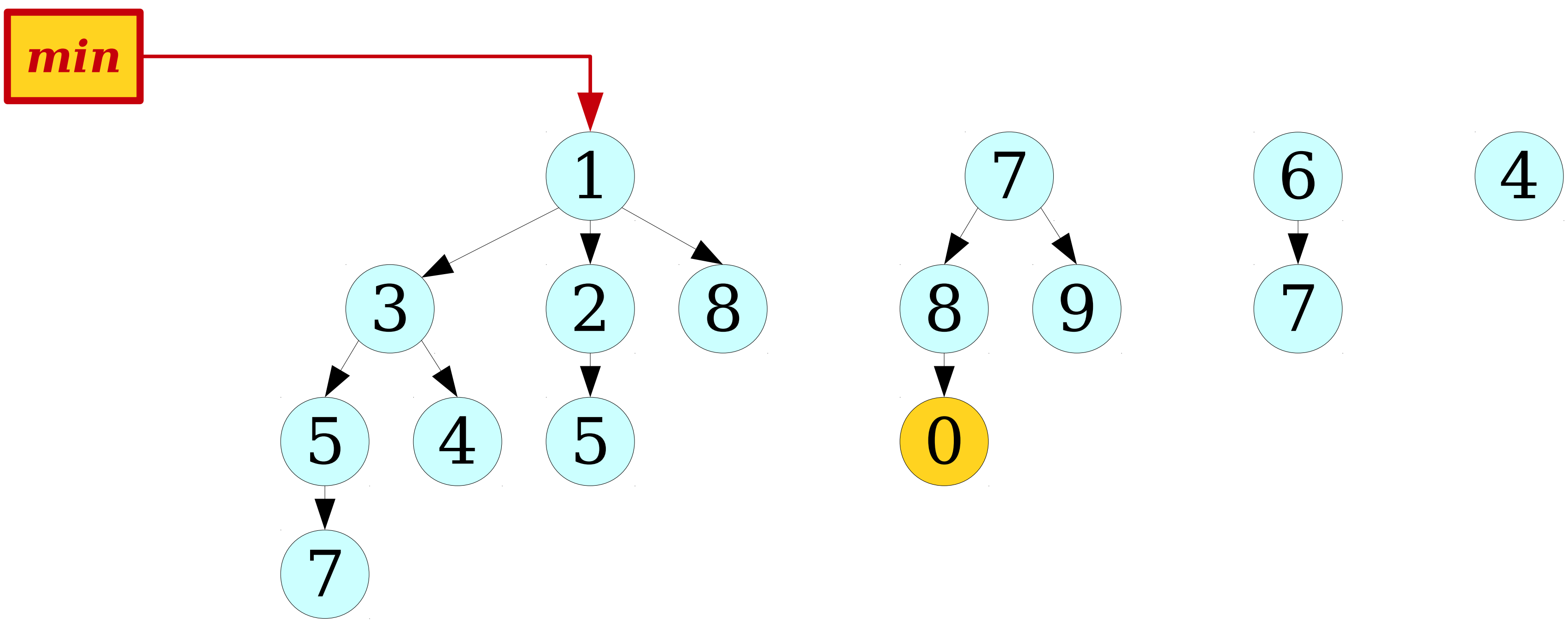
A Simple Implementation

- It is possible to implement ***decrease-key*** in time O(log *n*) using lazy binomial heaps.
- ***Idea:*** "Bubble" the element up toward the root of the binomial tree containing it and (potentially) update the *min* pointer.



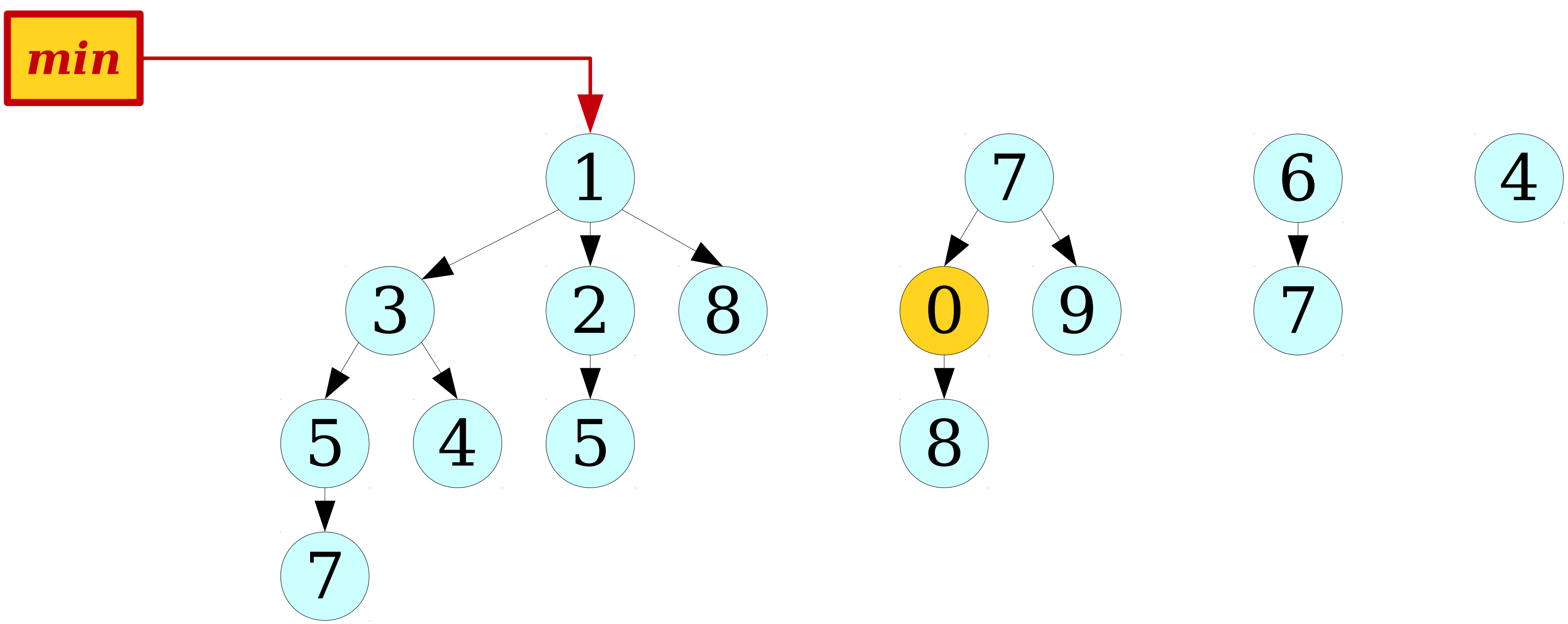
A Simple Implementation

- It is possible to implement ***decrease-key*** in time O(log *n*) using lazy binomial heaps.
- ***Idea:*** "Bubble" the element up toward the root of the binomial tree containing it and (potentially) update the *min* pointer.



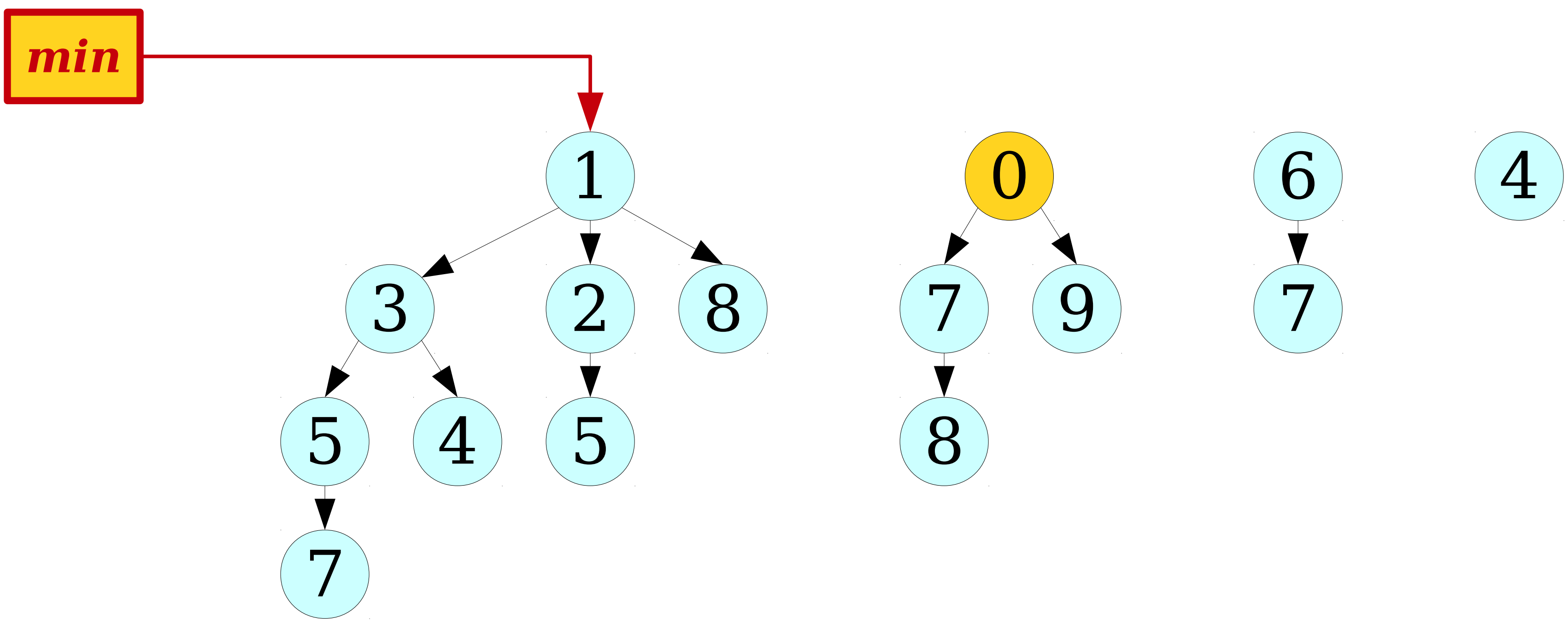
A Simple Implementation

- It is possible to implement ***decrease-key*** in time O(log n) using lazy binomial heaps.
- ***Idea:*** "Bubble" the element up toward the root of the binomial tree containing it and (potentially) update the *min* pointer.



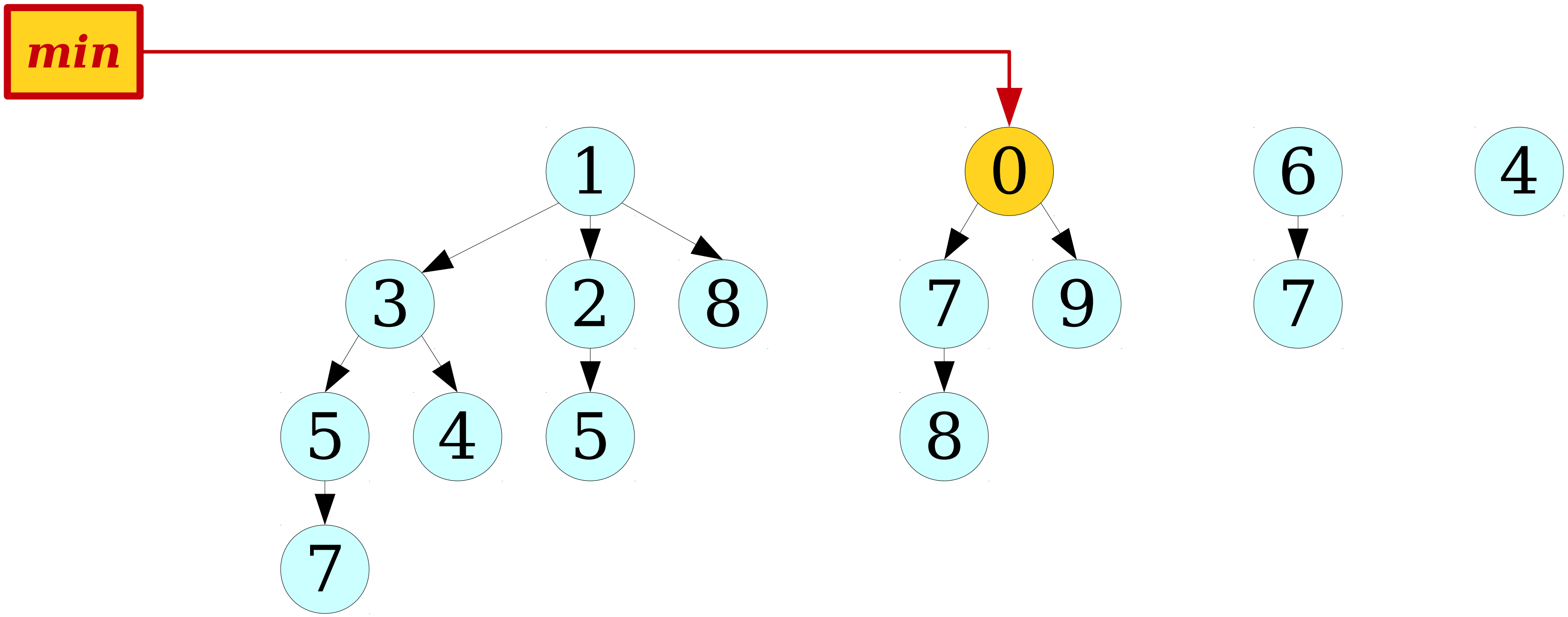
A Simple Implementation

- It is possible to implement ***decrease-key*** in time O(log *n*) using lazy binomial heaps.
- ***Idea:*** "Bubble" the element up toward the root of the binomial tree containing it and (potentially) update the *min* pointer.



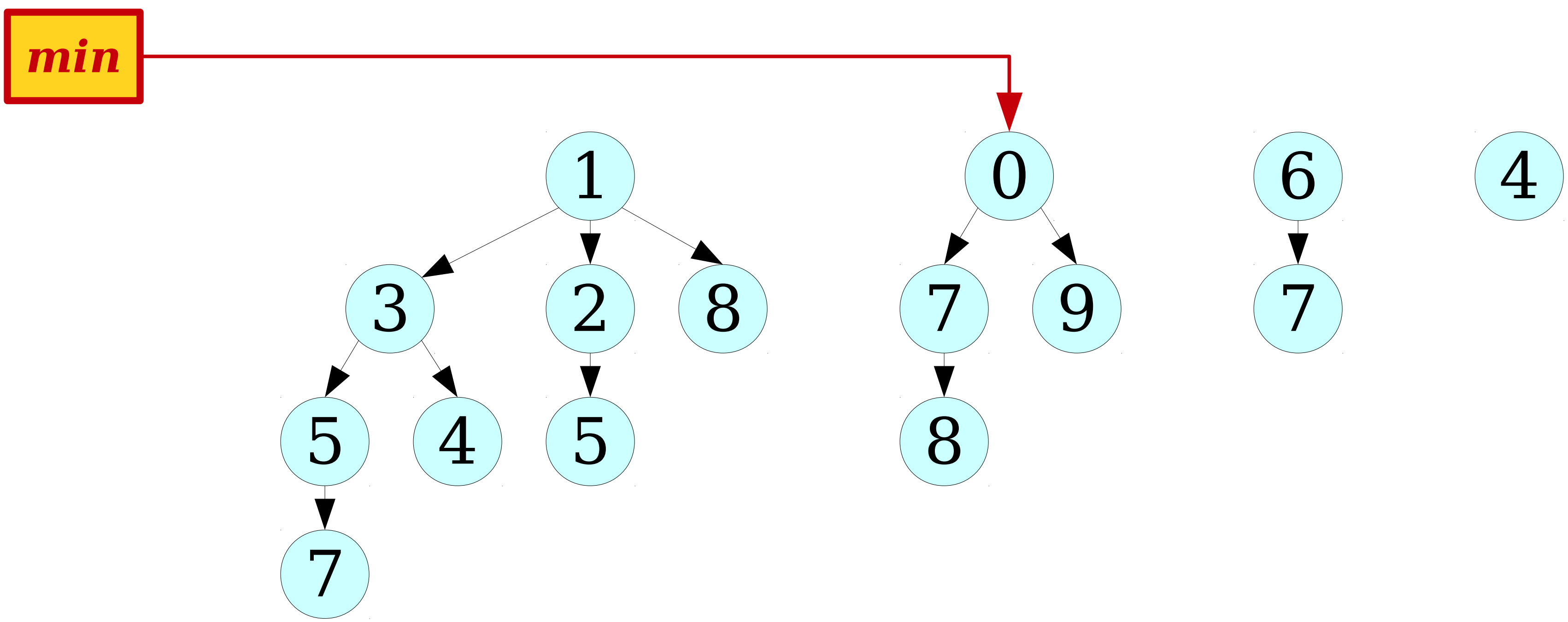
A Simple Implementation

- It is possible to implement ***decrease-key*** in time O(log *n*) using lazy binomial heaps.
- ***Idea:*** "Bubble" the element up toward the root of the binomial tree containing it and (potentially) update the *min* pointer.



A Simple Implementation

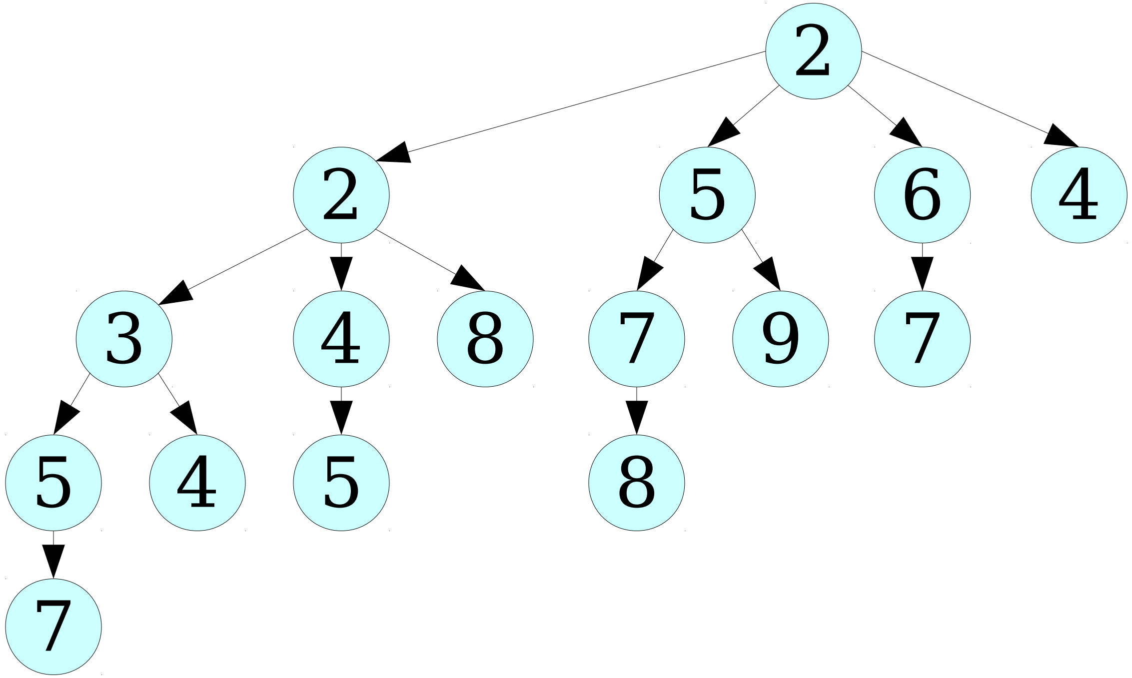
- It is possible to implement ***decrease-key*** in time O(log *n*) using lazy binomial heaps.
- ***Idea:*** "Bubble" the element up toward the root of the binomial tree containing it and (potentially) update the *min* pointer.



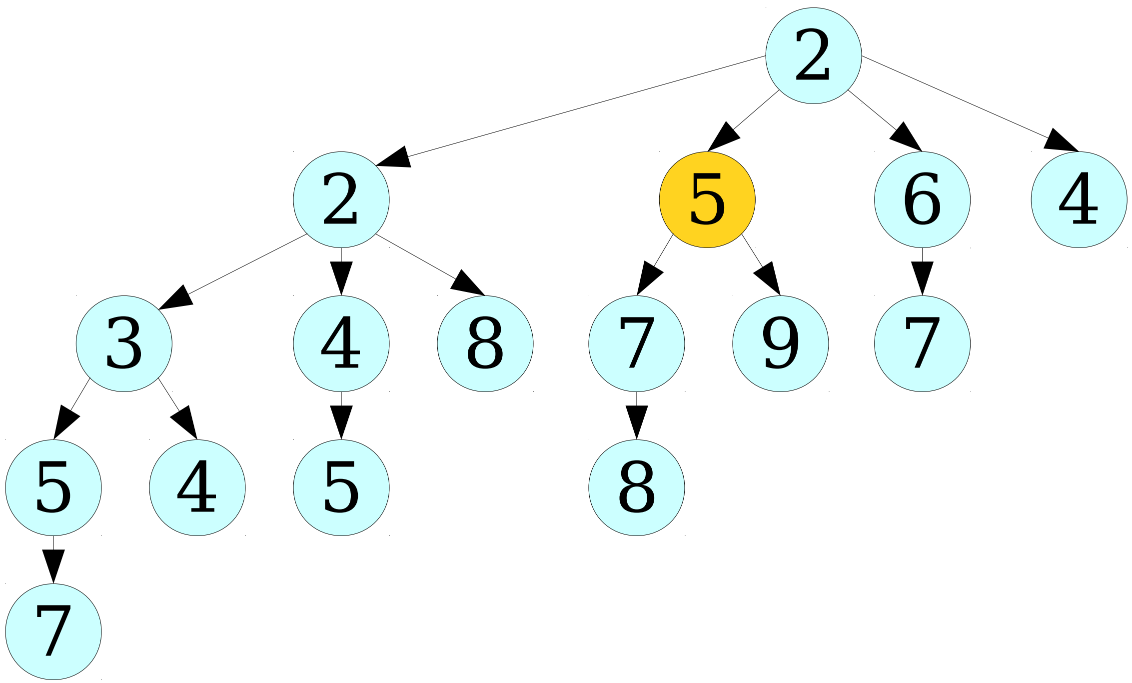
The Challenge

- ***Goal:*** Implement ***decrease-key*** in amortized time O(1).
- Why is this hard?
  - Lowering a node's priority might break the heap property.
  - Correcting the imbalance O(log n) layers deep in a tree might take time O(log n).
- We will need to change our approach.

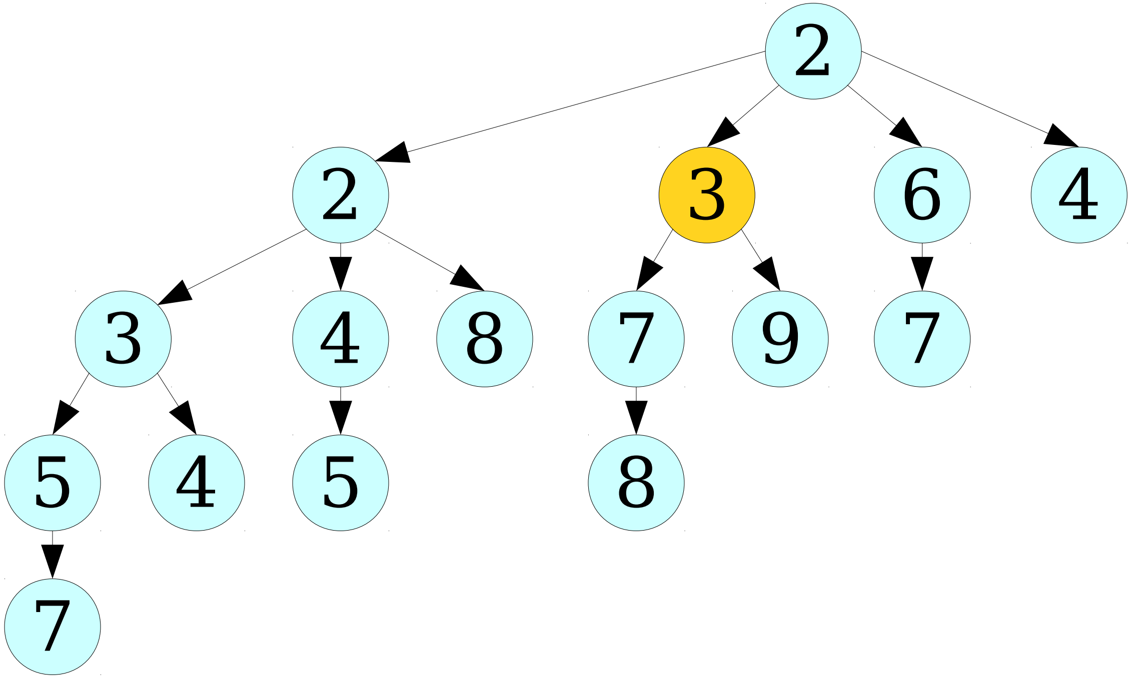
A Crazy Idea



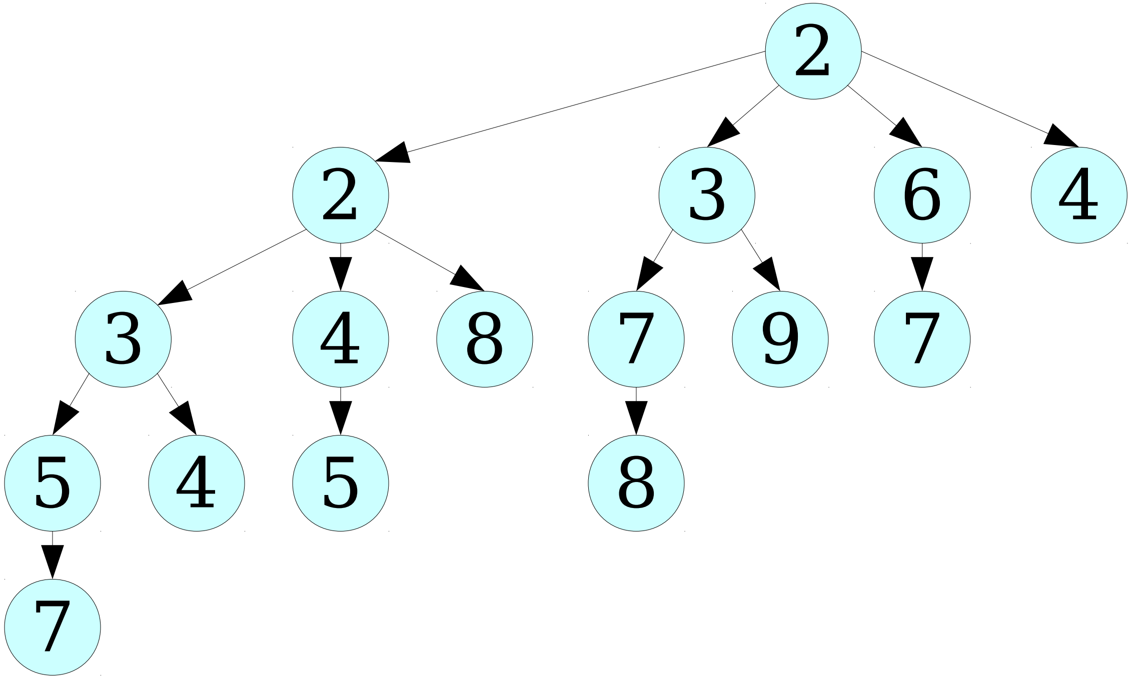
A Crazy Idea



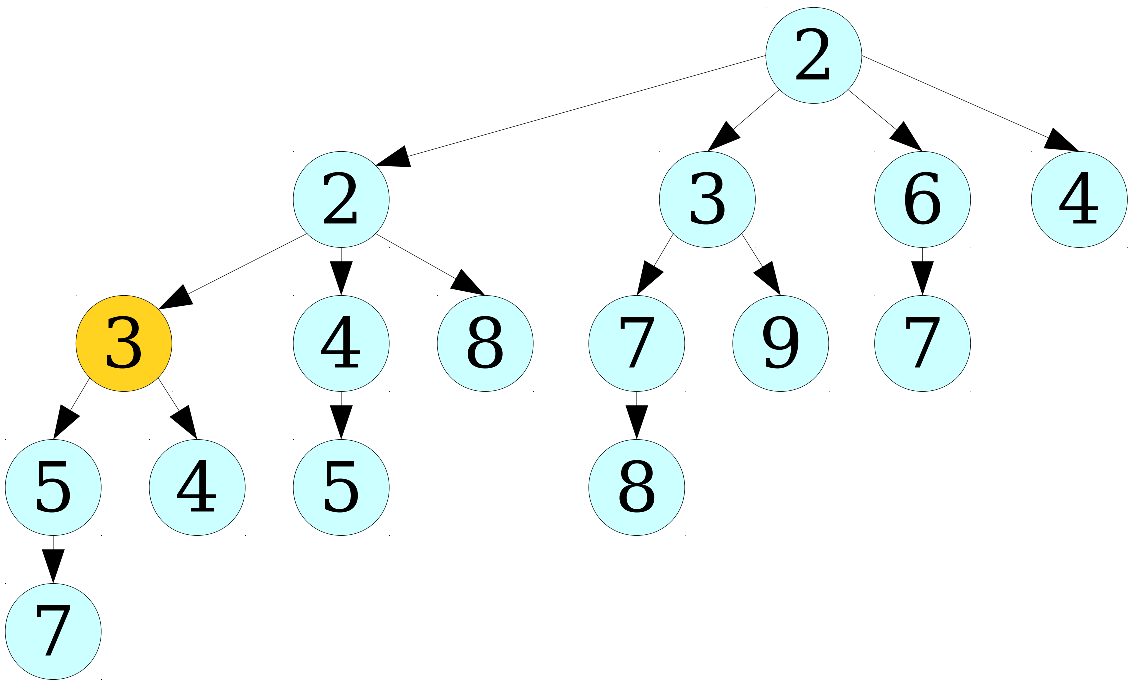
A Crazy Idea



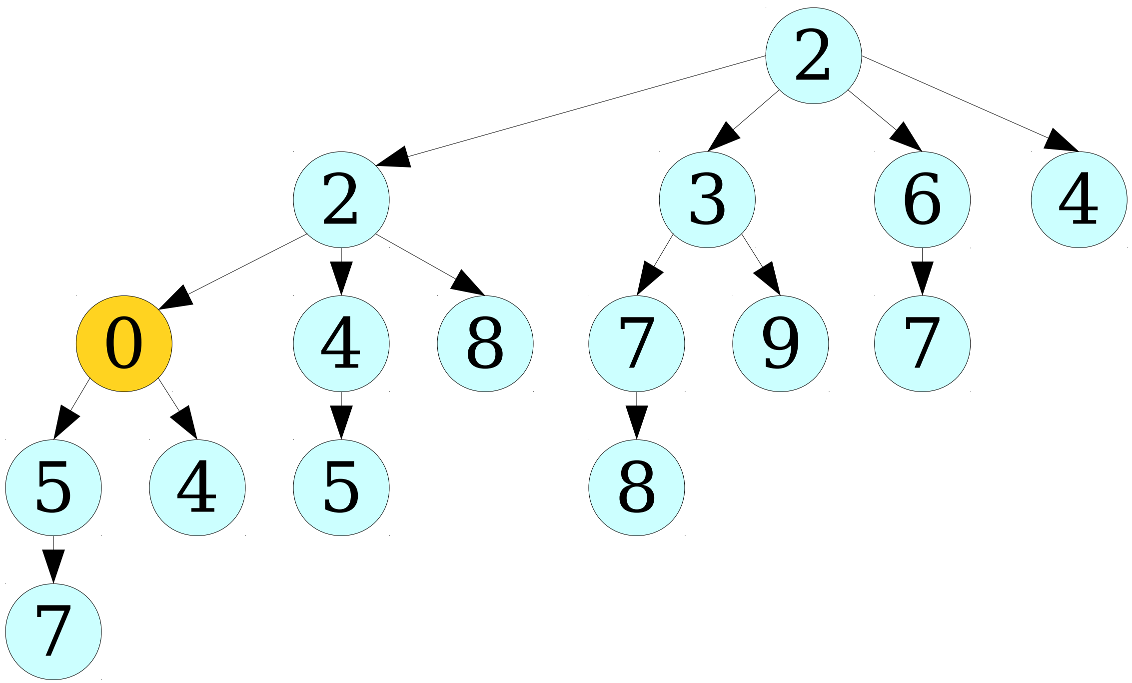
A Crazy Idea



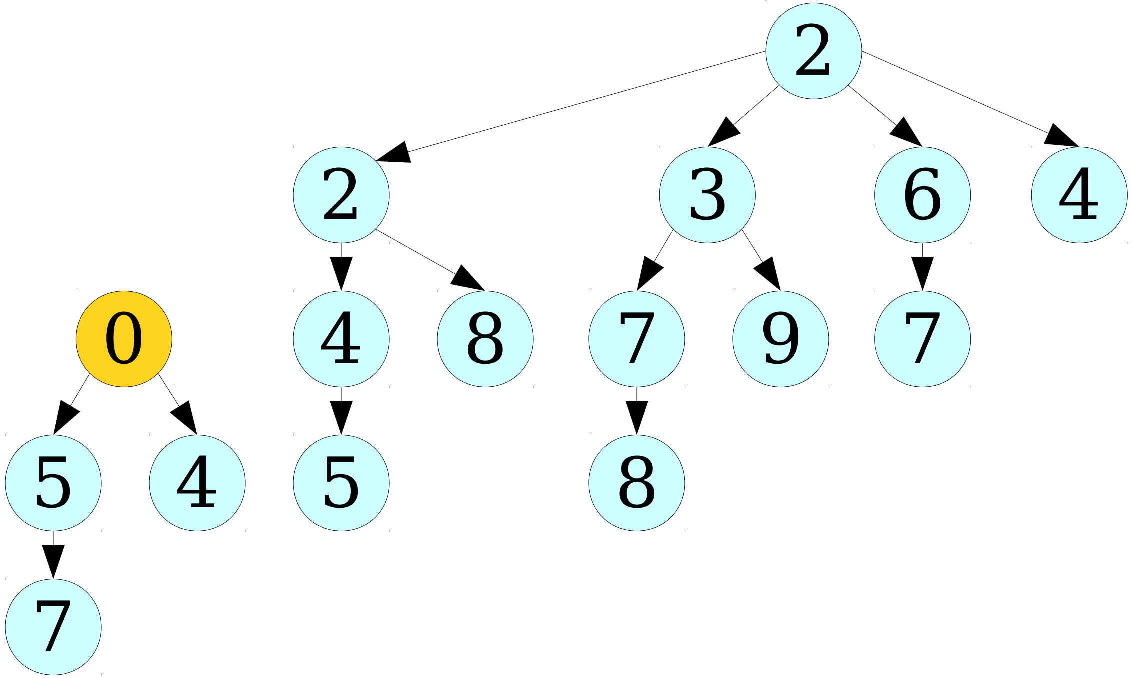
A Crazy Idea



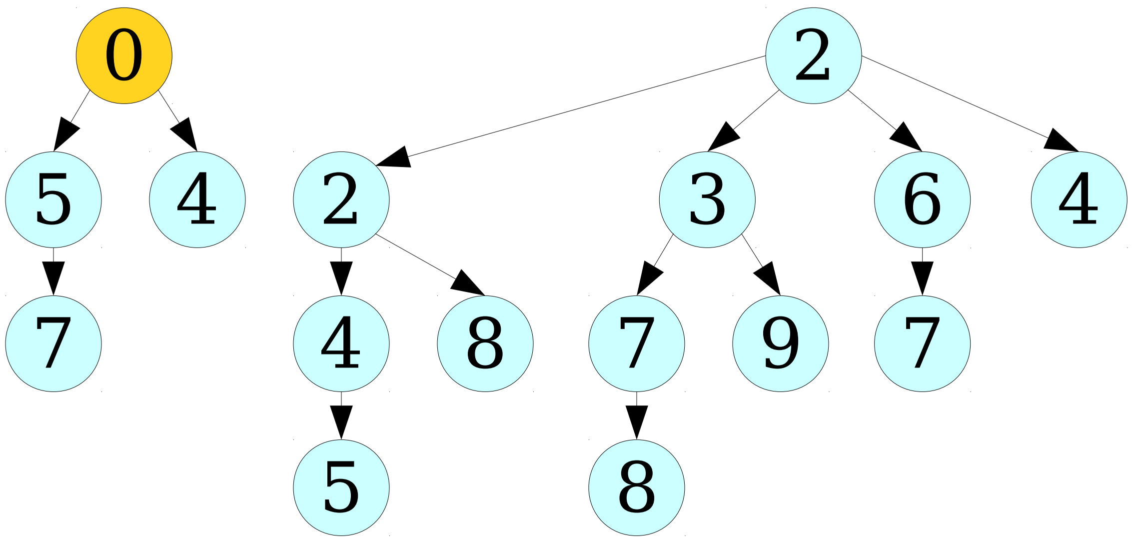
A Crazy Idea



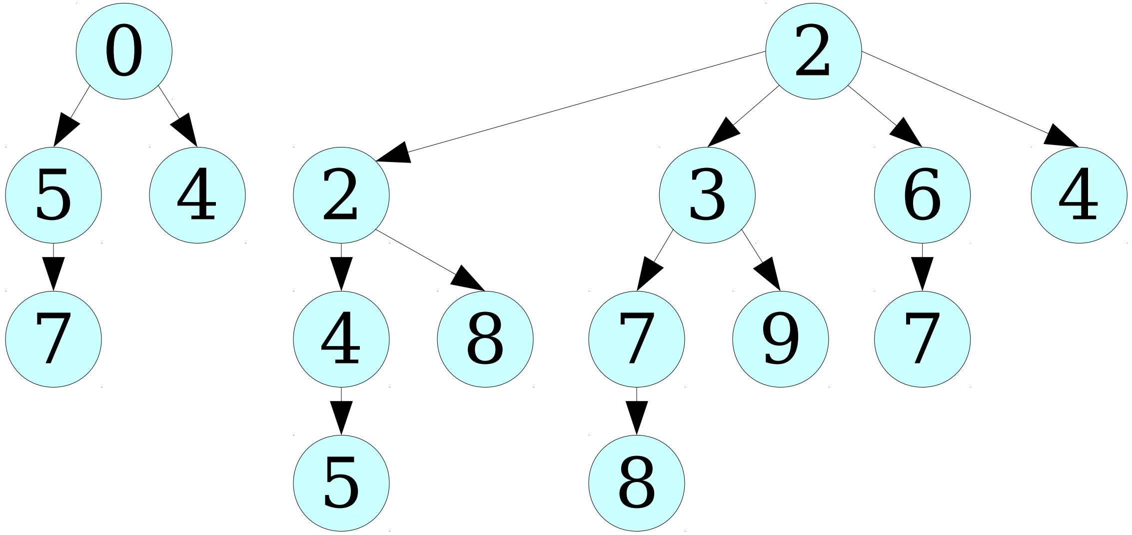
A Crazy Idea



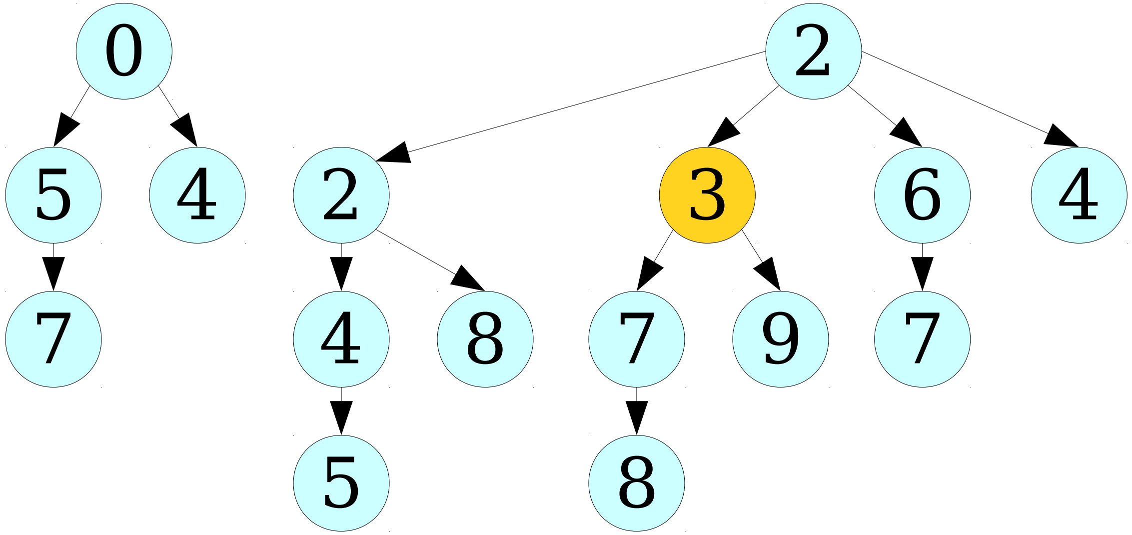
A Crazy Idea



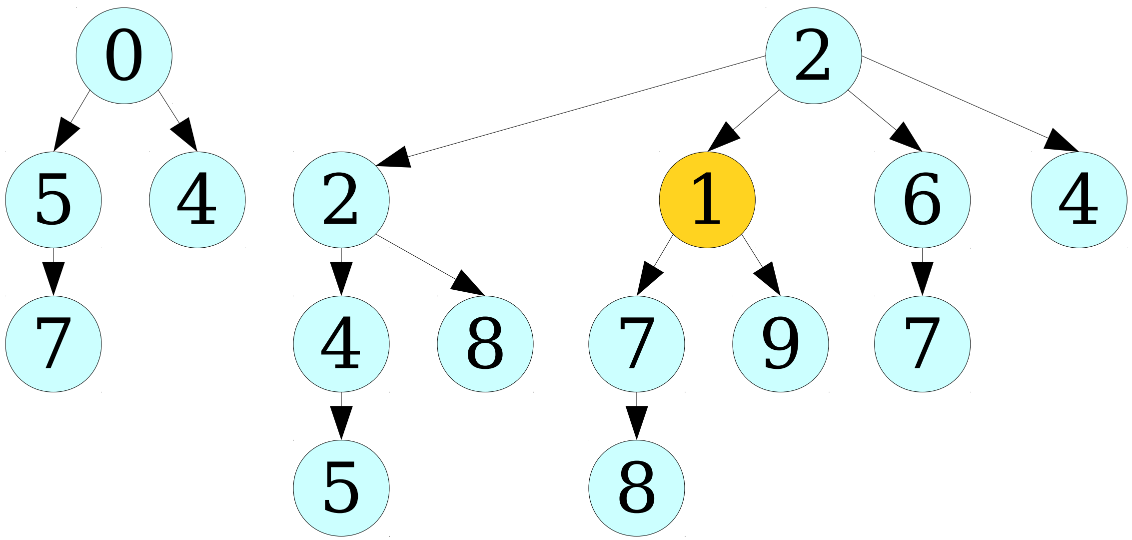
A Crazy Idea



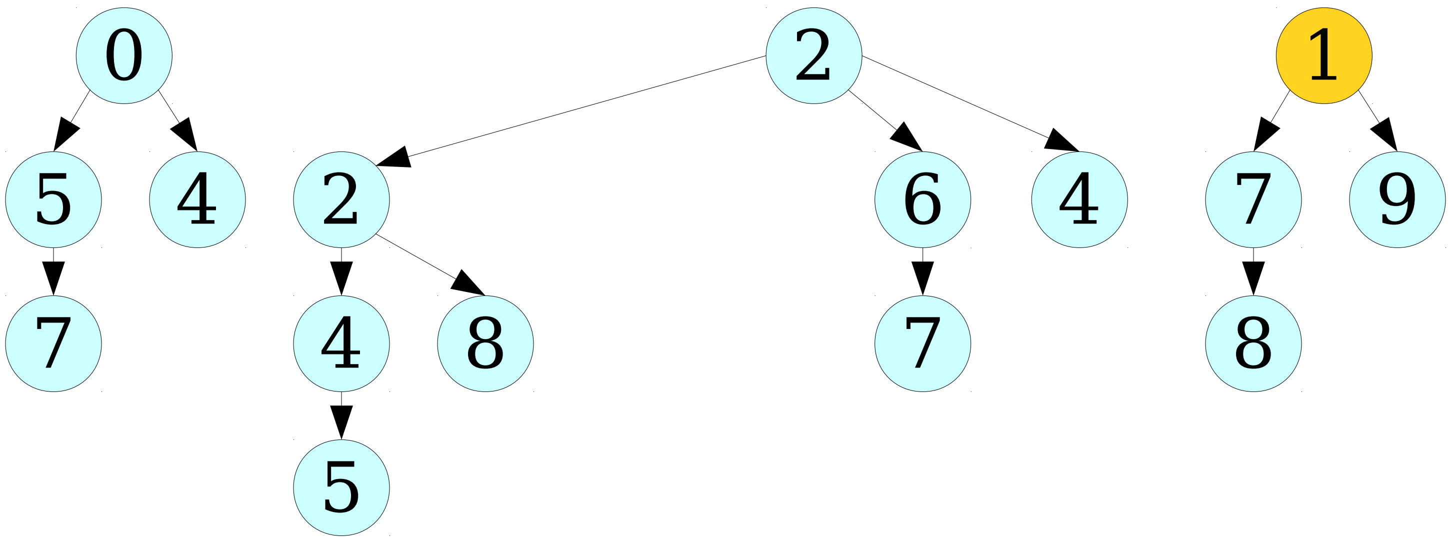
A Crazy Idea



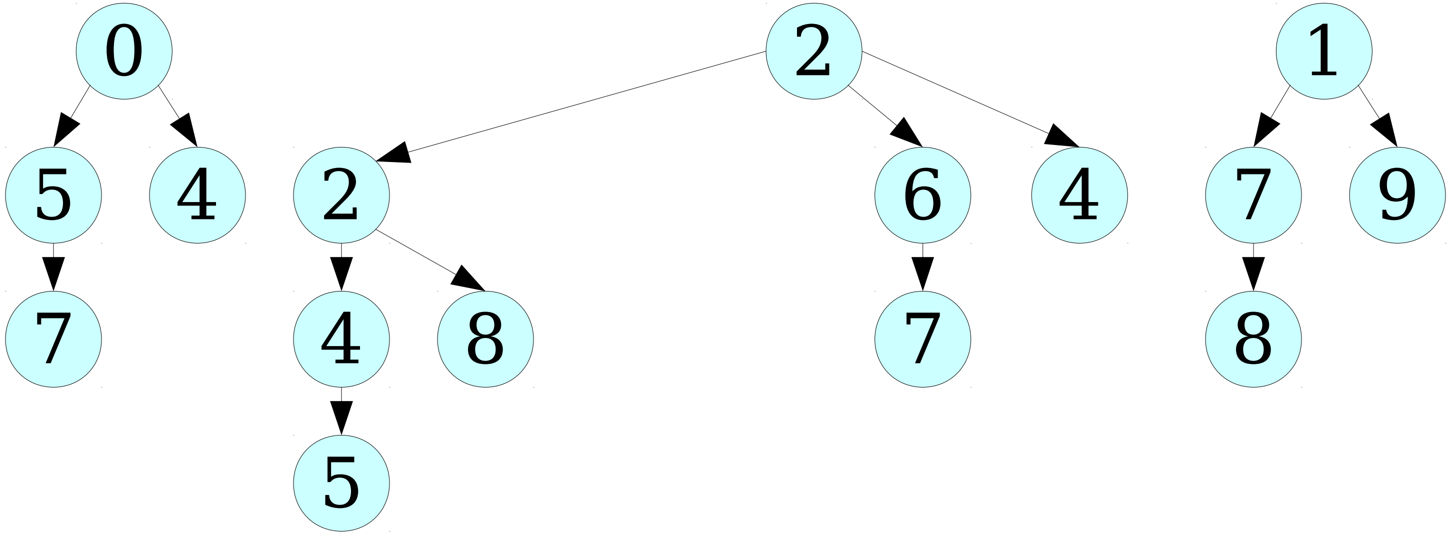
A Crazy Idea



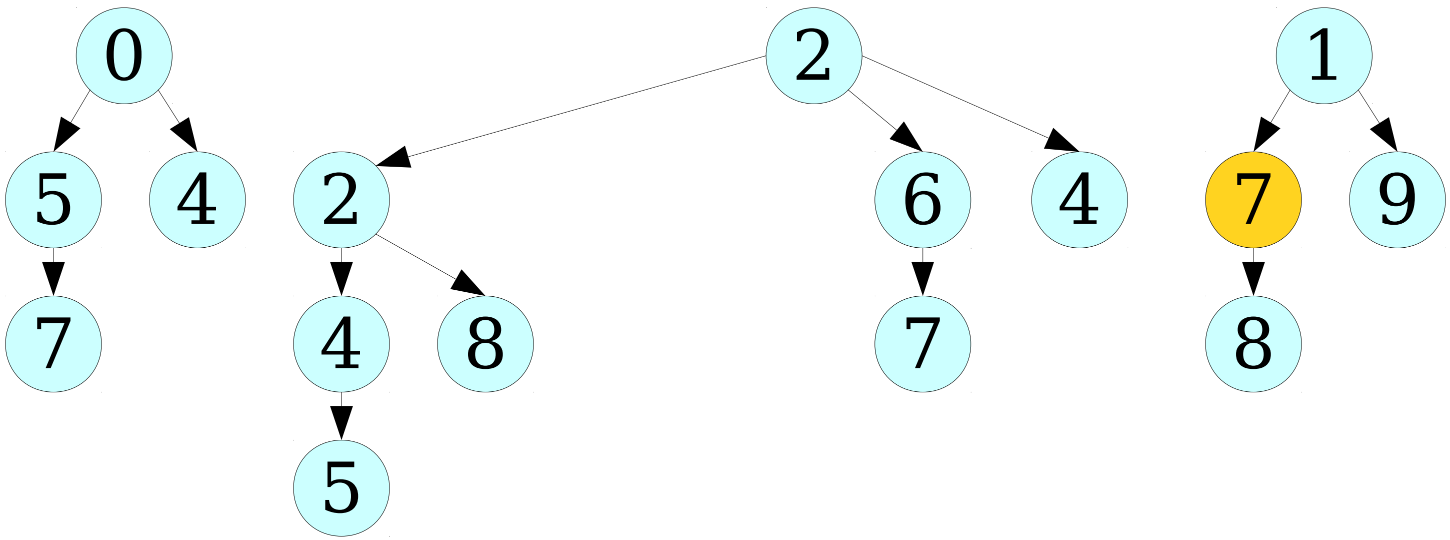
A Crazy Idea



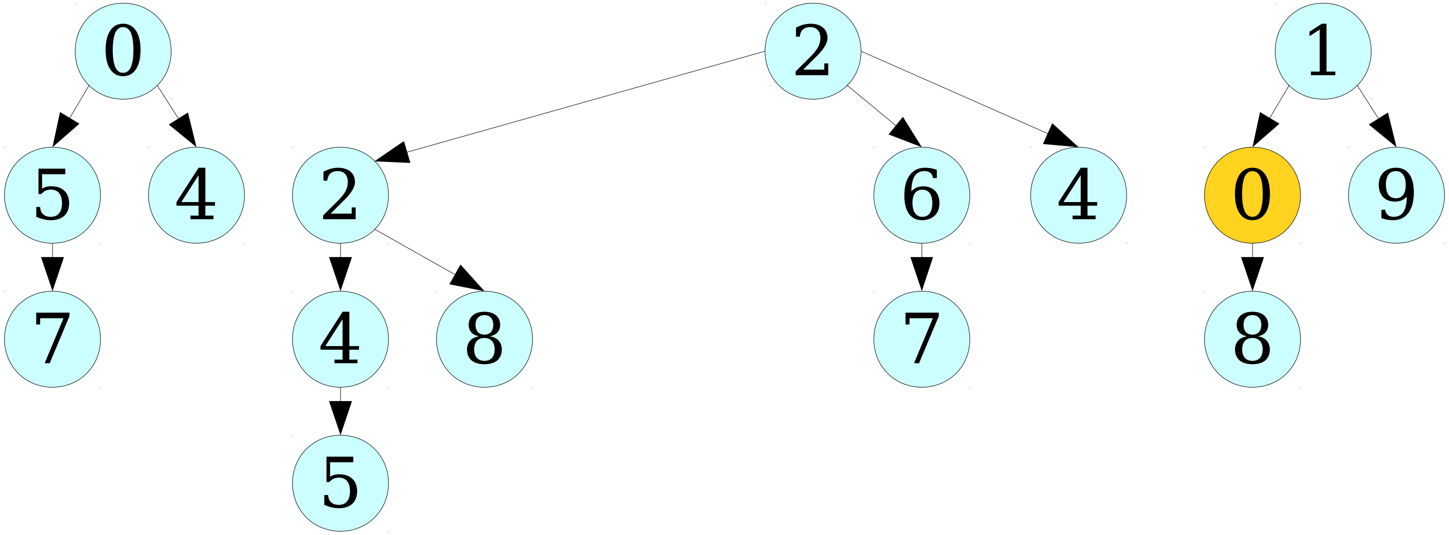
A Crazy Idea



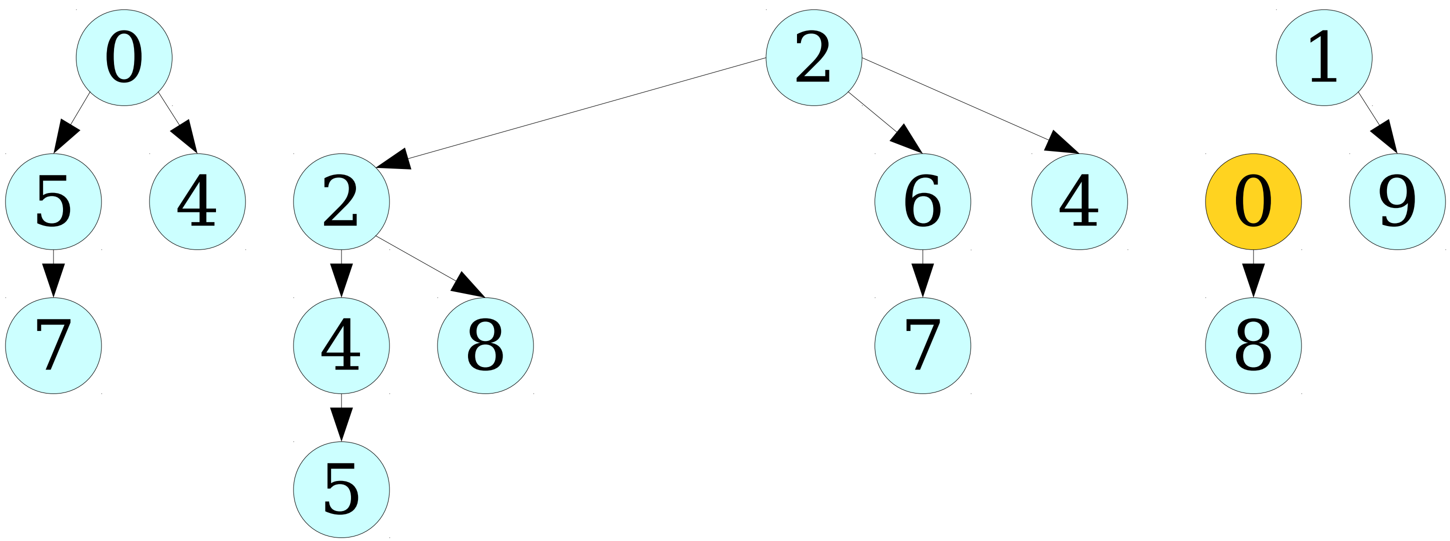
A Crazy Idea



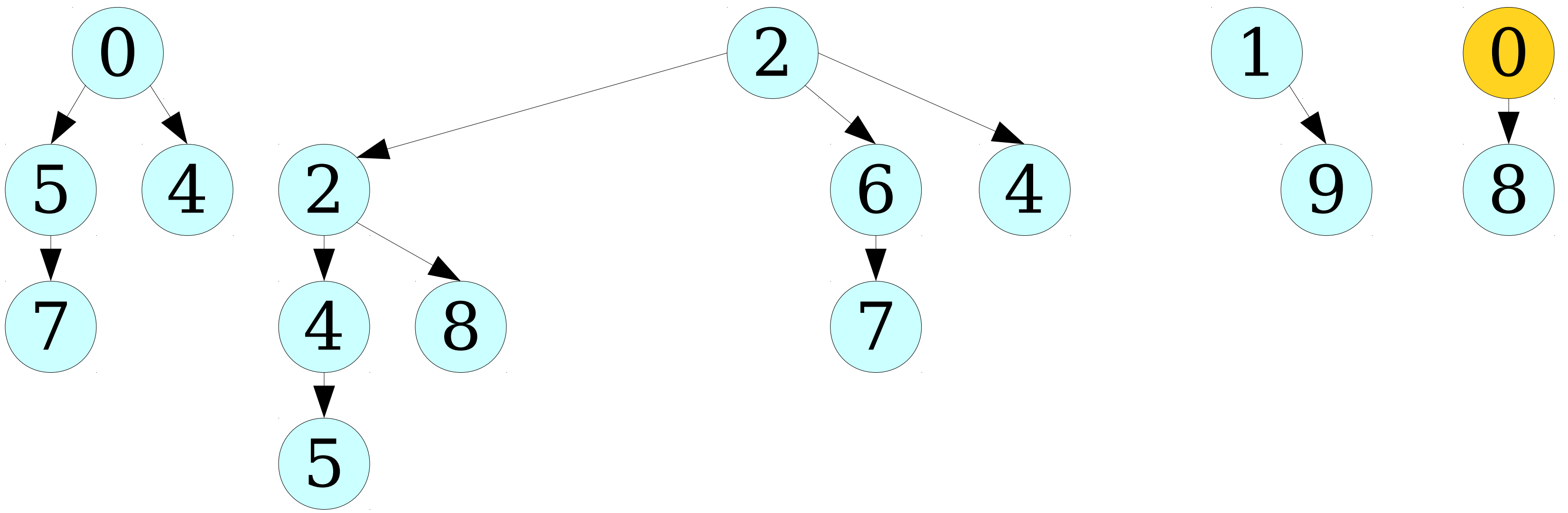
A Crazy Idea



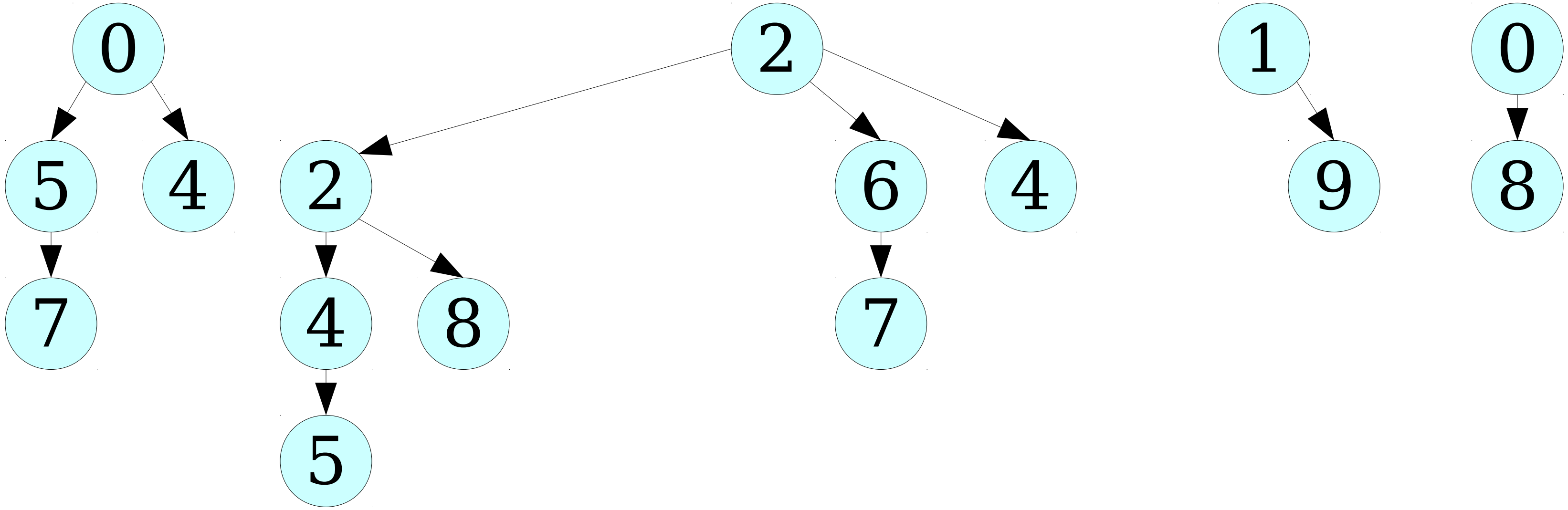
A Crazy Idea



A Crazy Idea



A Crazy Idea



A Crazy Idea

- To implement ***decrease-key*** efficiently:
  - Lower the key of the specified node.
  - If its key is greater than or equal to its parent's key, we're done.
  - Otherwise, cut that node from its parent and hoist it up to the root list, optionally updating the min pointer.
- Time required: O(1).
  - This requires some changes to the tree representation; more details later.

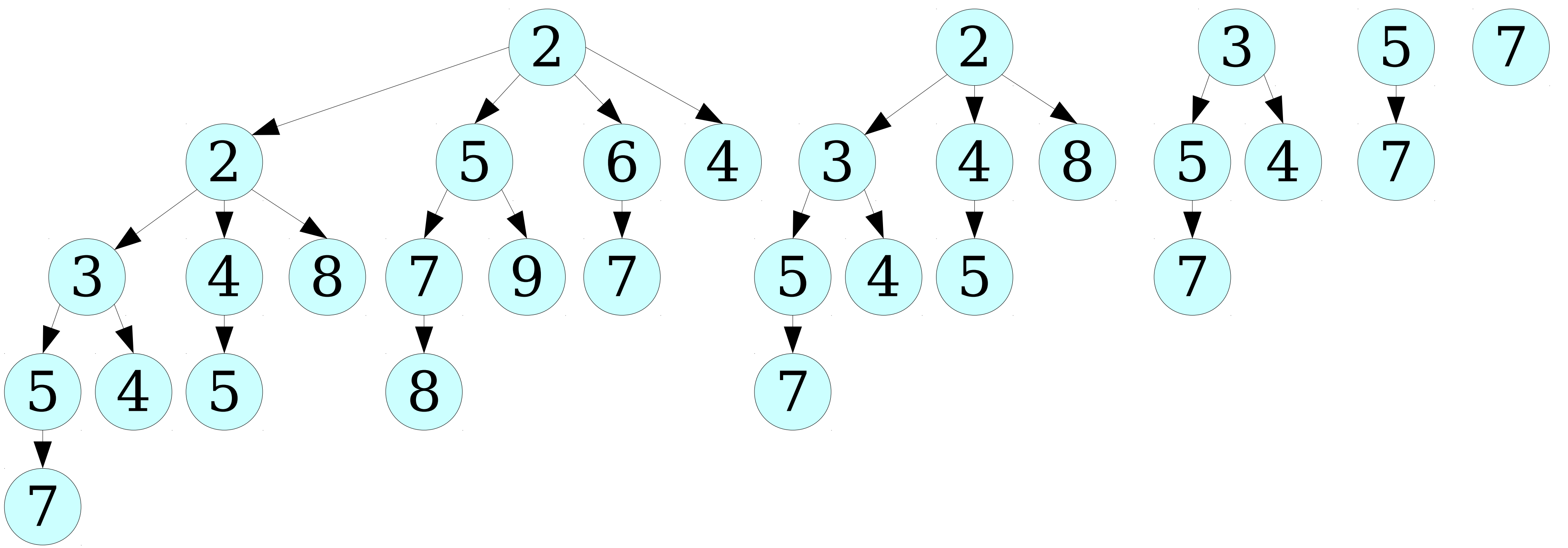
Analyzing our Approach

(or: The Madness in the Method)

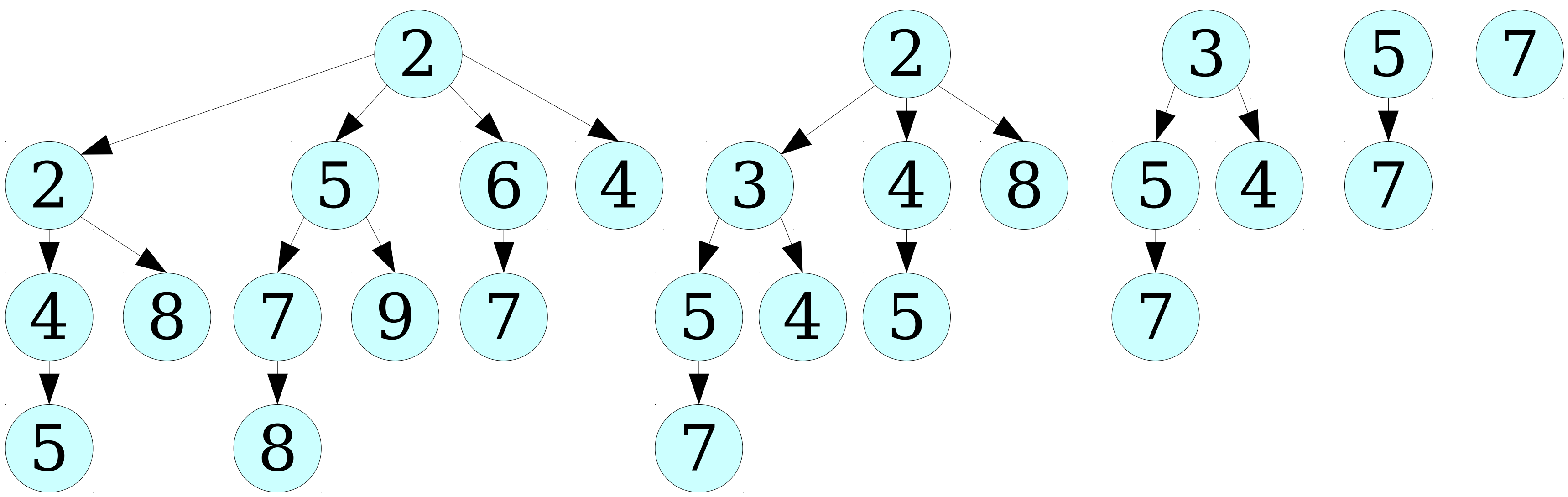
Tree Sizes and Orders

- ***Recall:*** The ***order*** of a binomial tree is the number of children of the root.
- In a true binomial tree, a binomial tree of order k has exactly 2^k nodes.
- ***Concern:*** If trees can be cut from their parents, a tree of order k might have many fewer than 2^k nodes.

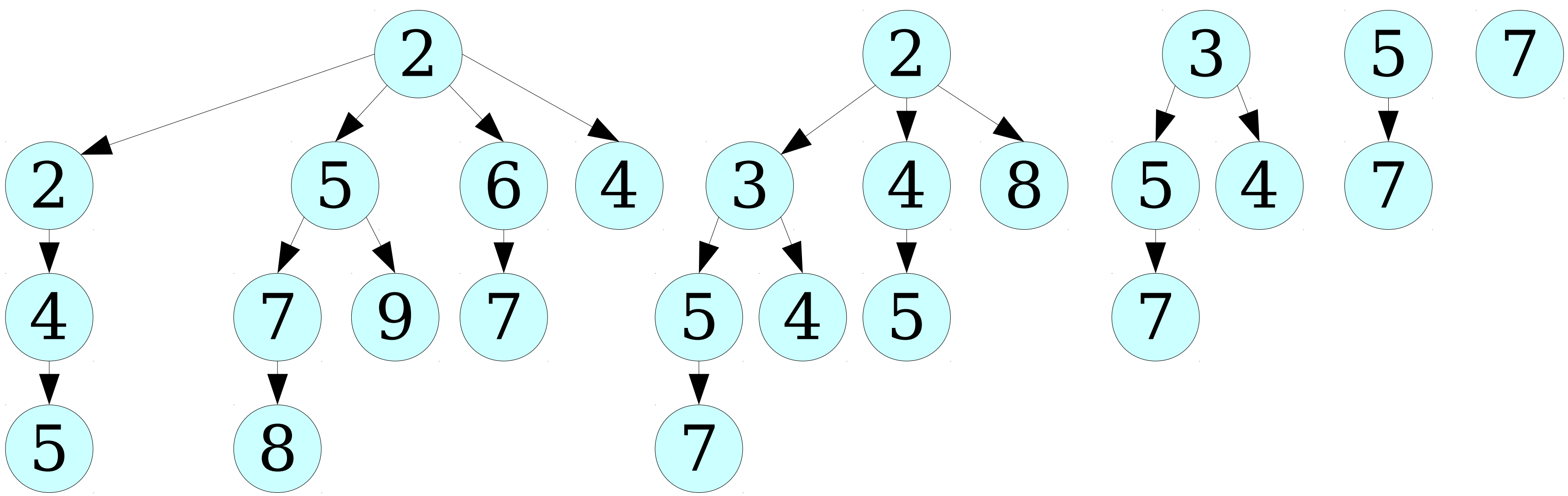
The Problem



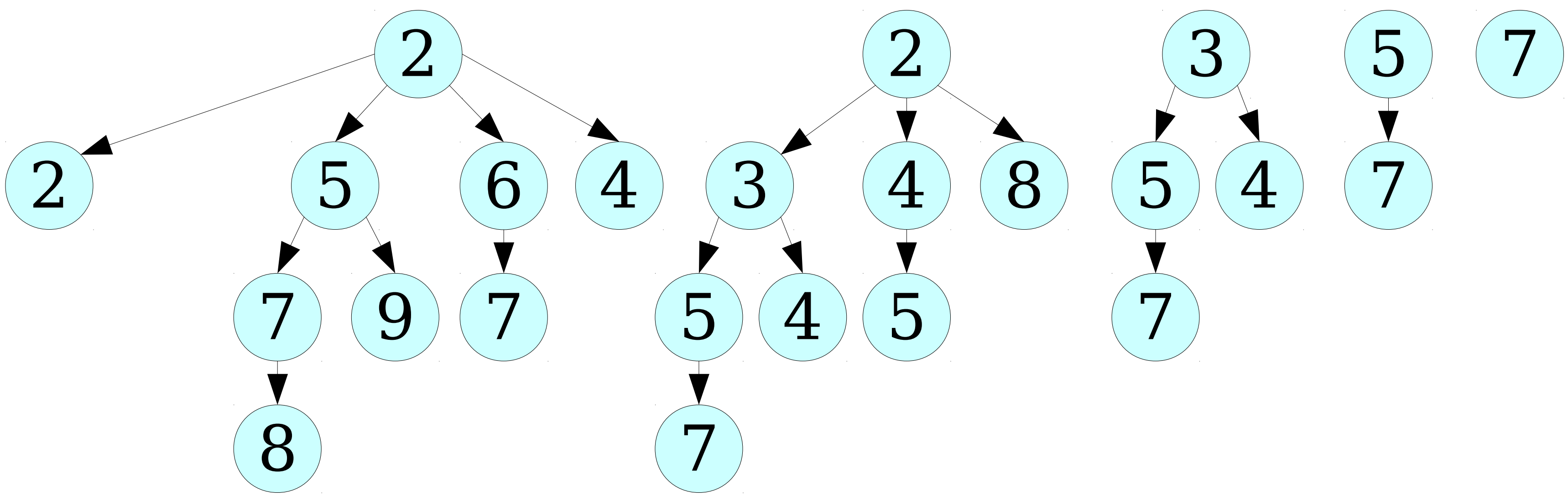
The Problem



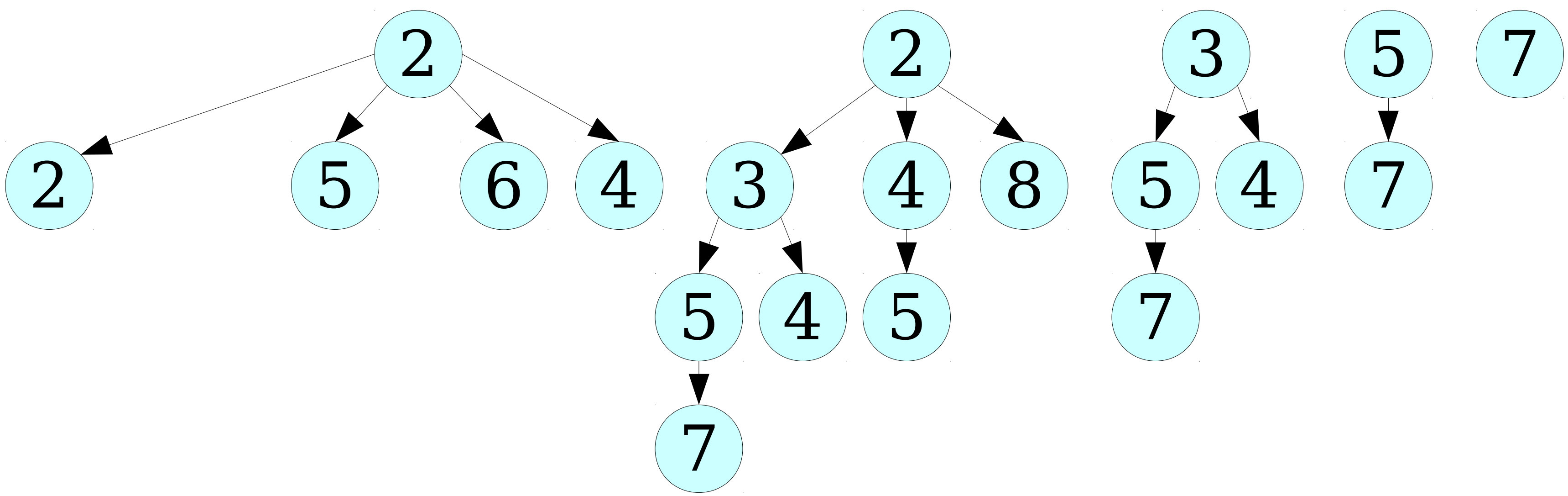
The Problem



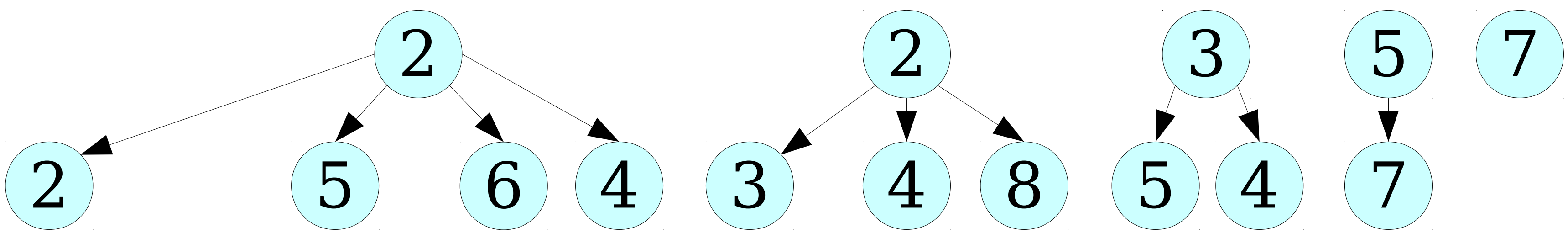
The Problem



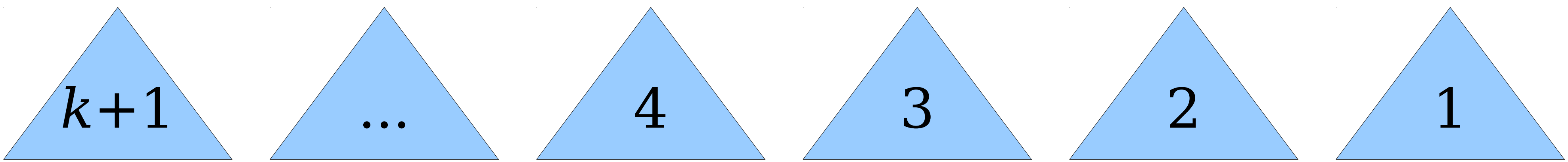
The Problem



The Problem



The Problem



Number of nodes: $\Theta(k^2)$

Number of trees: $\boldsymbol{\Theta(n^{1/2})}$

The Problem

- ***Recall:*** The amortized cost of an ***extract-min*** is only O(log n) if each tree of order k has an exponential number of nodes in it.
- With our “damaged” binomial trees, this is no longer the case, and the amortized cost of an ***extract-min*** grows to O($n^{1/2}$).
- We've lost our runtime bounds!

Time-Out for Announcements!

Problem Sets

- Problem Set Three was due at the start of class today.
  - Want to use late days? Feel free to submit it by Saturday at 2:30PM.
- Problem Set Two has been graded. Feedback is now available up on GradeScope.
- The next problem set goes out on Tuesday. We recommend using the interstitial time to think about your project proposal.
  - Proposals are due next Thursday at 2:30PM.
  - Looking for a team? Use the “Search for Teammates” features up on Piazza!

Back to CS166!

The Problem

- This problem arises because we have lost one of the guarantees of binomial trees:

A binomial tree of order k has 2^k nodes.

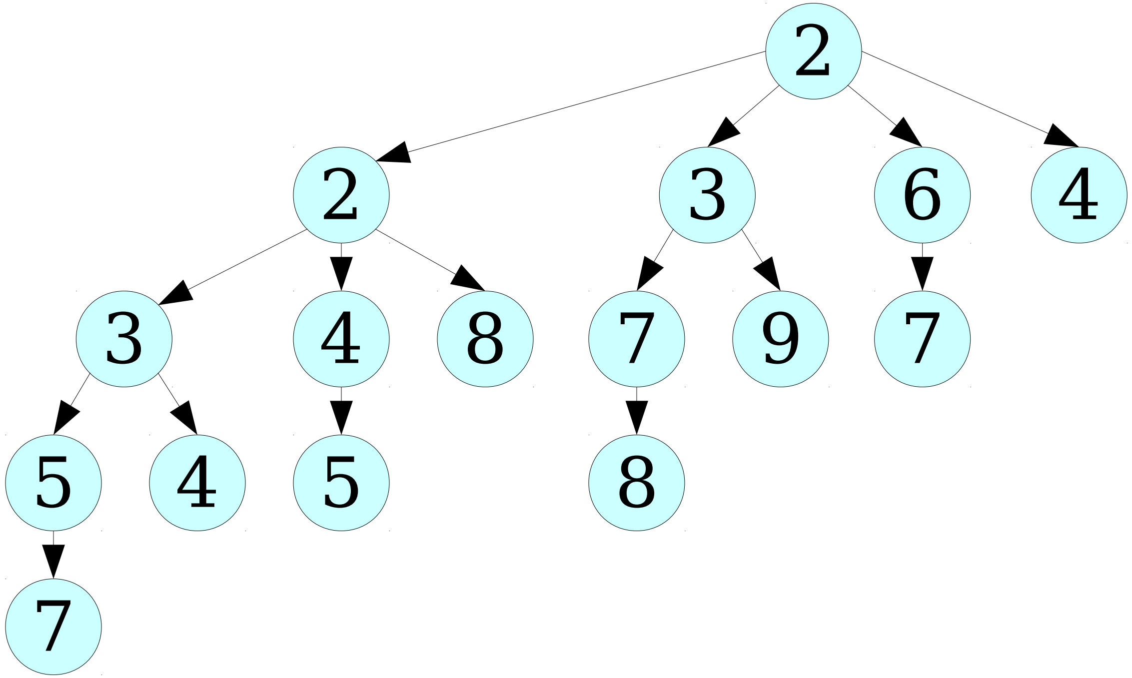
- When we cut low-hanging trees, the root node won't learn that these trees are missing.
- However, communicating this information up from the leaves to the root might take time O(log n)!

The Tradeoff

- If we don't impose any structural constraints on our trees, then trees of large order may have too few nodes.
  - Leads to having lots of short, small trees, wrecking our runtime bounds for ***extract-min***.
- If we impose too many structural constraints on our trees, then we have to spend too much time fixing up trees.
  - Leads to ***decrease-key*** taking too long.
- How can we strike a balance?

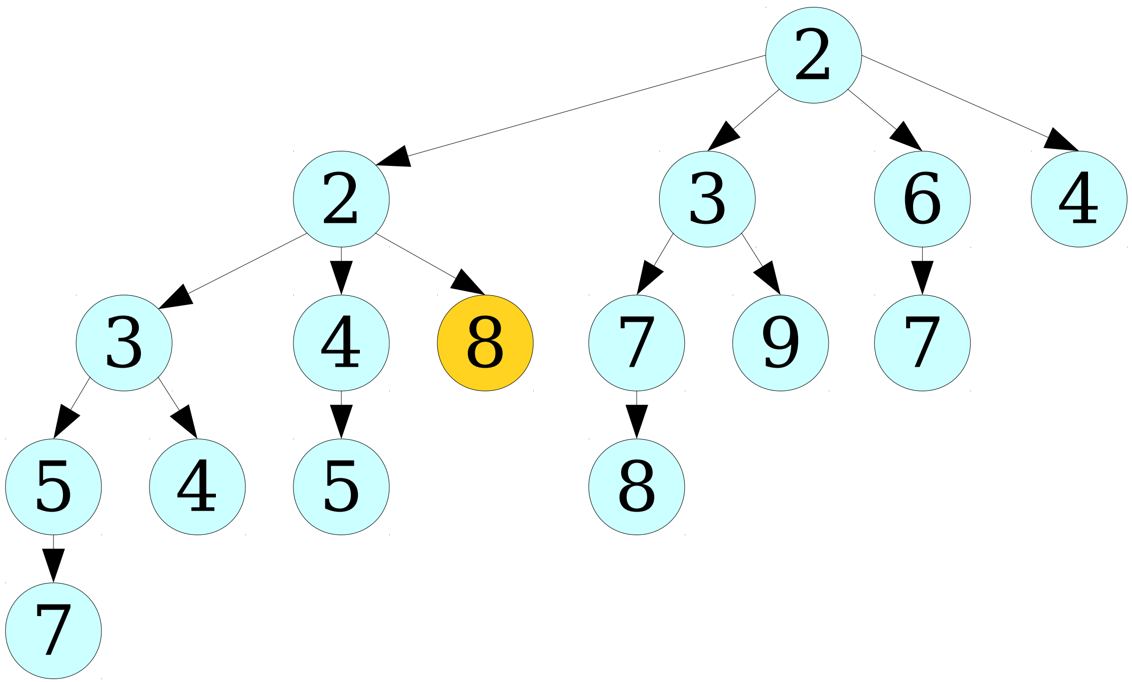
The Compromise

- Every non-root node is allowed to lose at most one child.
- If a non-root node loses two children, we cut it from its parent. (This might trigger more cuts.)
- We will ***mark*** nodes in the heap that have lost children to keep track of this fact.



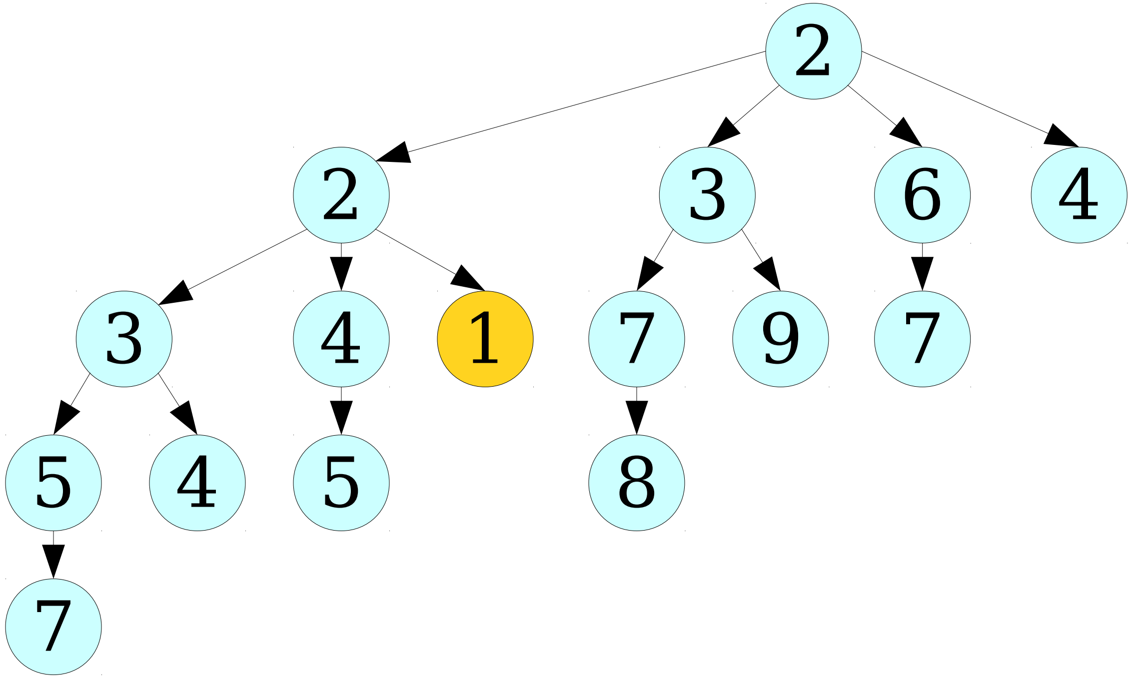
The Compromise

- Every non-root node is allowed to lose at most one child.
- If a non-root node loses two children, we cut it from its parent. (This might trigger more cuts.)
- We will ***mark*** nodes in the heap that have lost children to keep track of this fact.



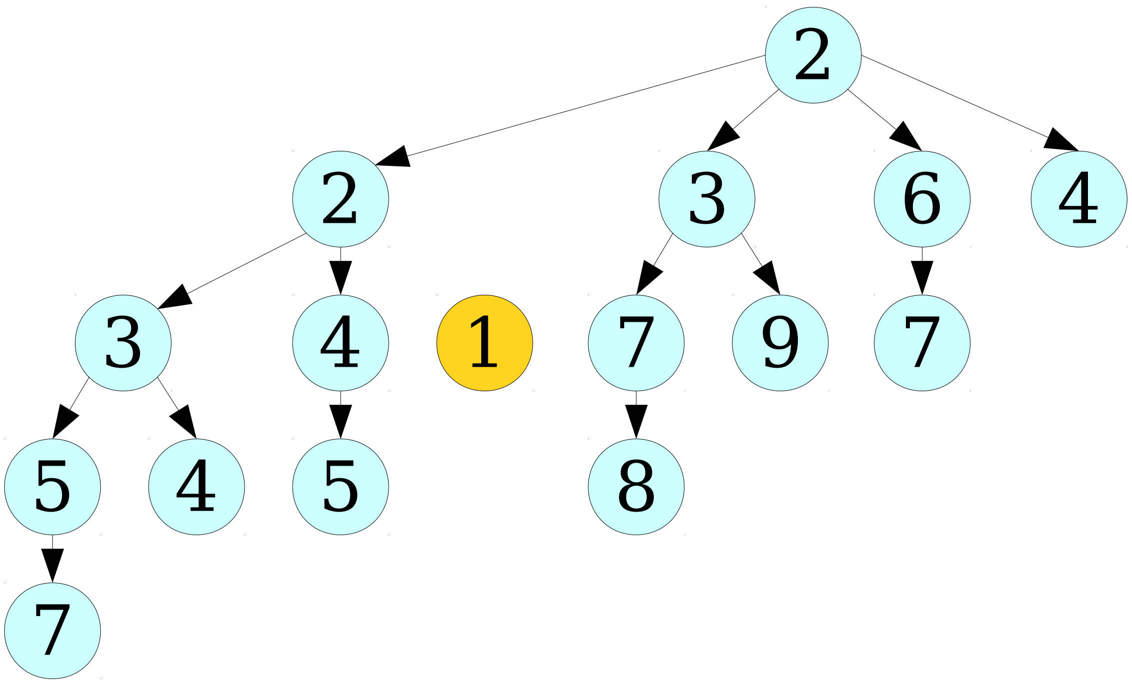
The Compromise

- Every non-root node is allowed to lose at most one child.
- If a non-root node loses two children, we cut it from its parent. (This might trigger more cuts.)
- We will ***mark*** nodes in the heap that have lost children to keep track of this fact.



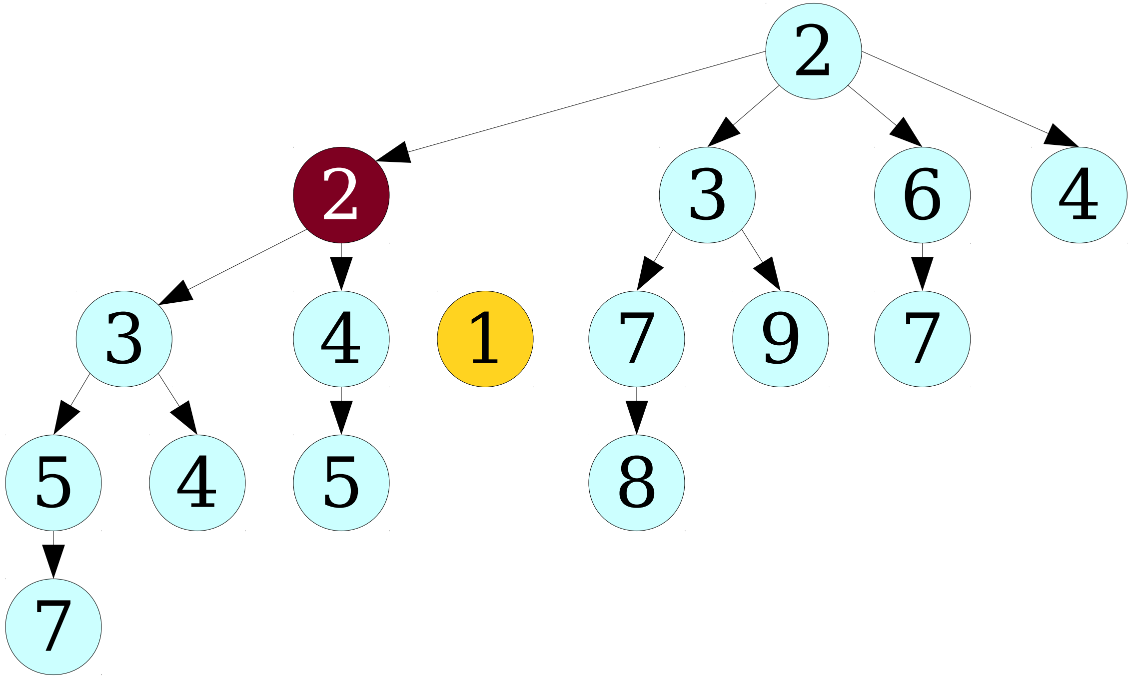
The Compromise

- Every non-root node is allowed to lose at most one child.
- If a non-root node loses two children, we cut it from its parent. (This might trigger more cuts.)
- We will ***mark*** nodes in the heap that have lost children to keep track of this fact.



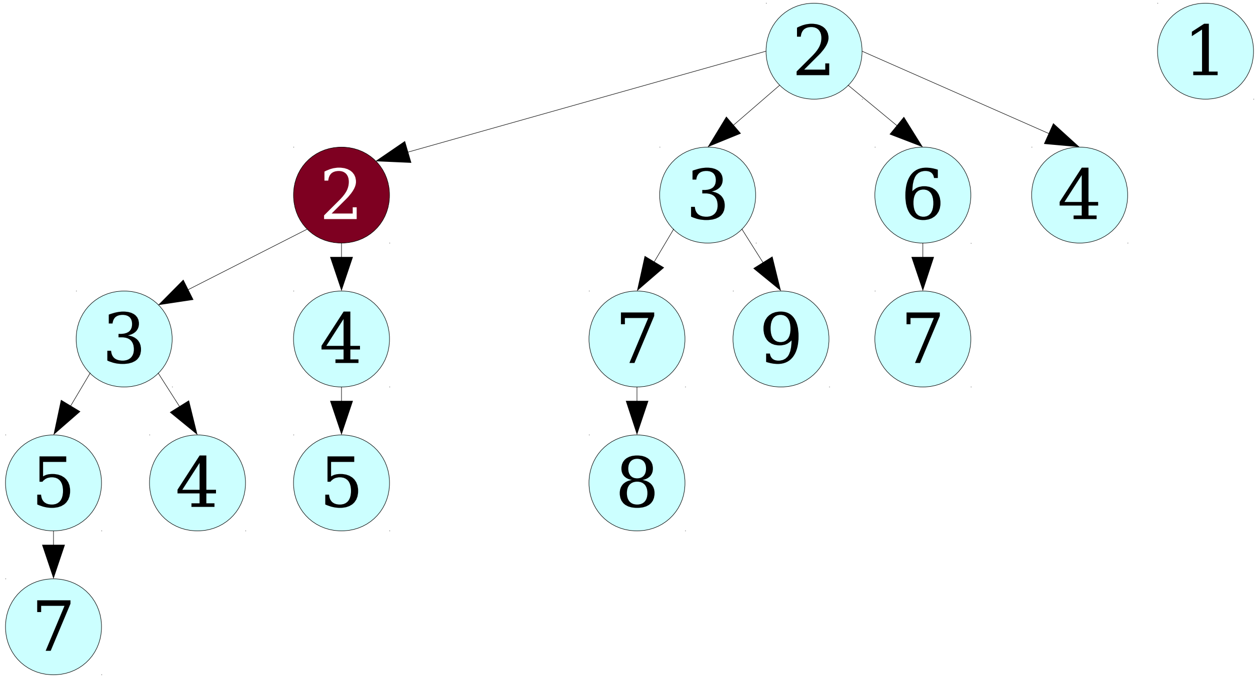
The Compromise

- Every non-root node is allowed to lose at most one child.
- If a non-root node loses two children, we cut it from its parent. (This might trigger more cuts.)
- We will ***mark*** nodes in the heap that have lost children to keep track of this fact.



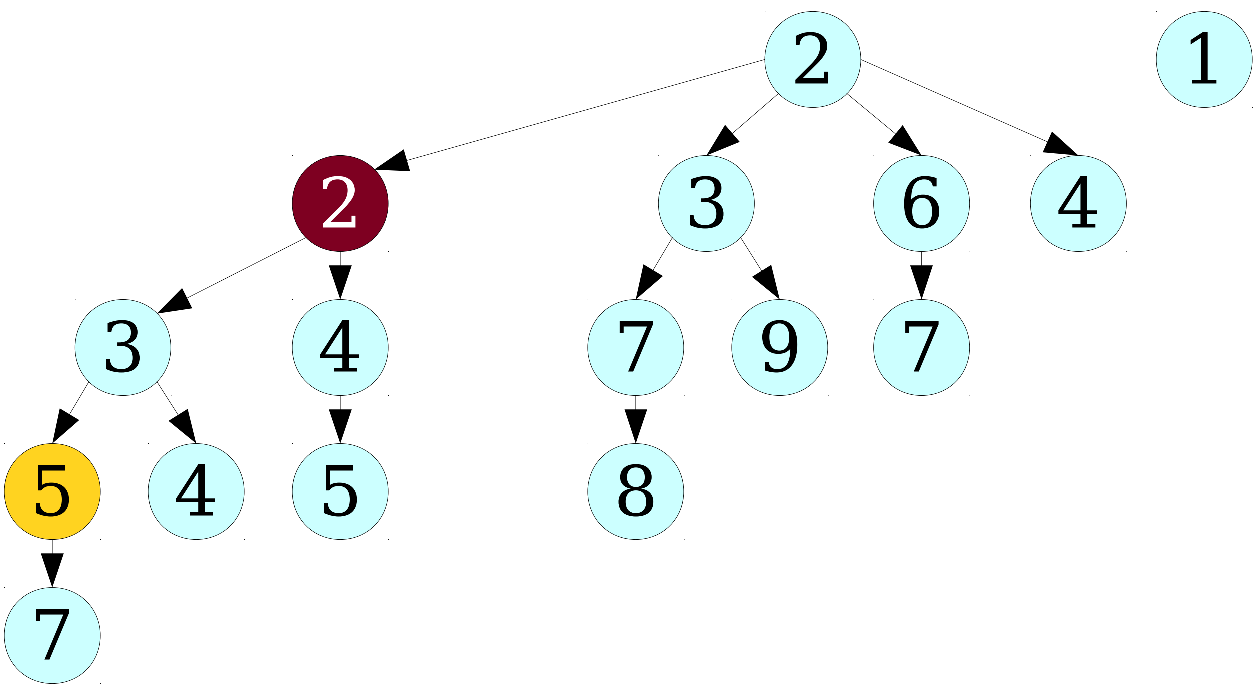
The Compromise

- Every non-root node is allowed to lose at most one child.
- If a non-root node loses two children, we cut it from its parent. (This might trigger more cuts.)
- We will ***mark*** nodes in the heap that have lost children to keep track of this fact.



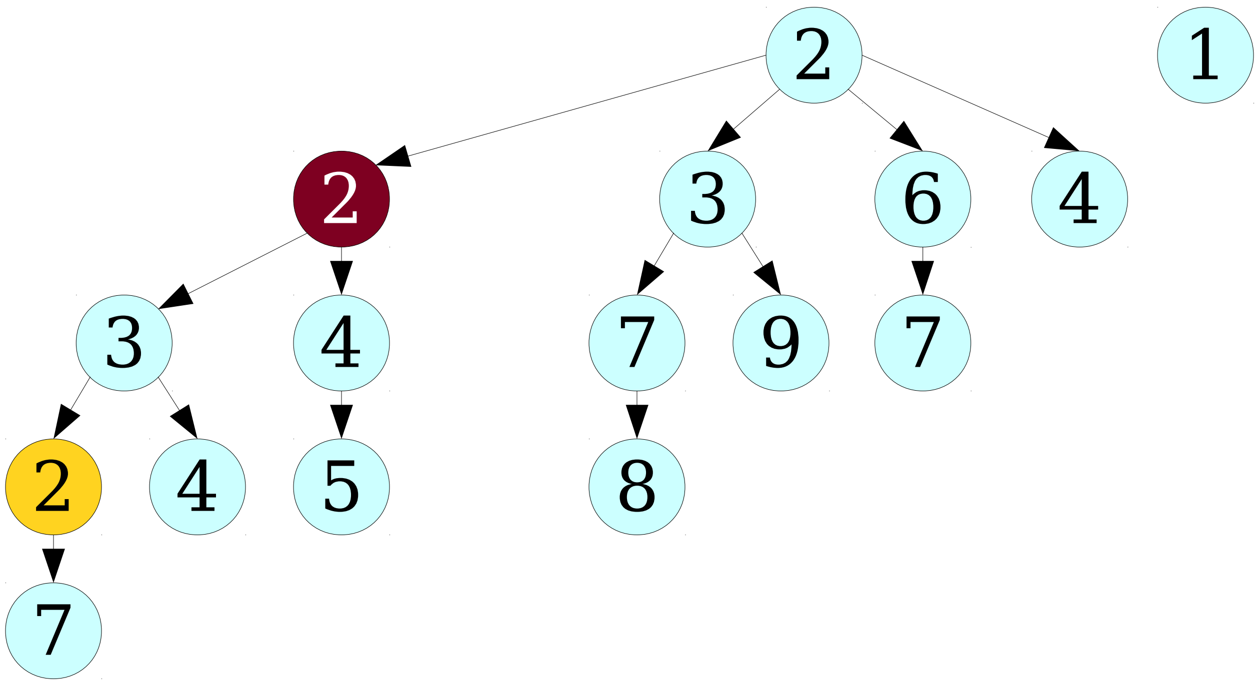
The Compromise

- Every non-root node is allowed to lose at most one child.
- If a non-root node loses two children, we cut it from its parent. (This might trigger more cuts.)
- We will ***mark*** nodes in the heap that have lost children to keep track of this fact.



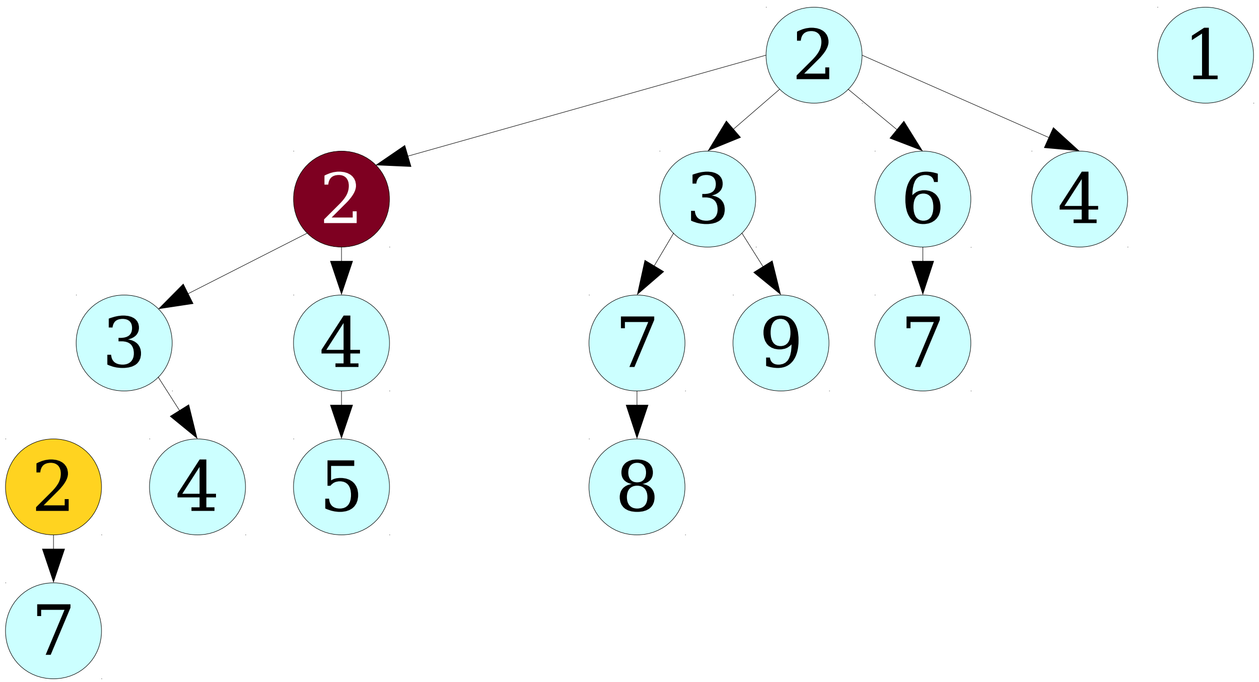
The Compromise

- Every non-root node is allowed to lose at most one child.
- If a non-root node loses two children, we cut it from its parent. (This might trigger more cuts.)
- We will ***mark*** nodes in the heap that have lost children to keep track of this fact.



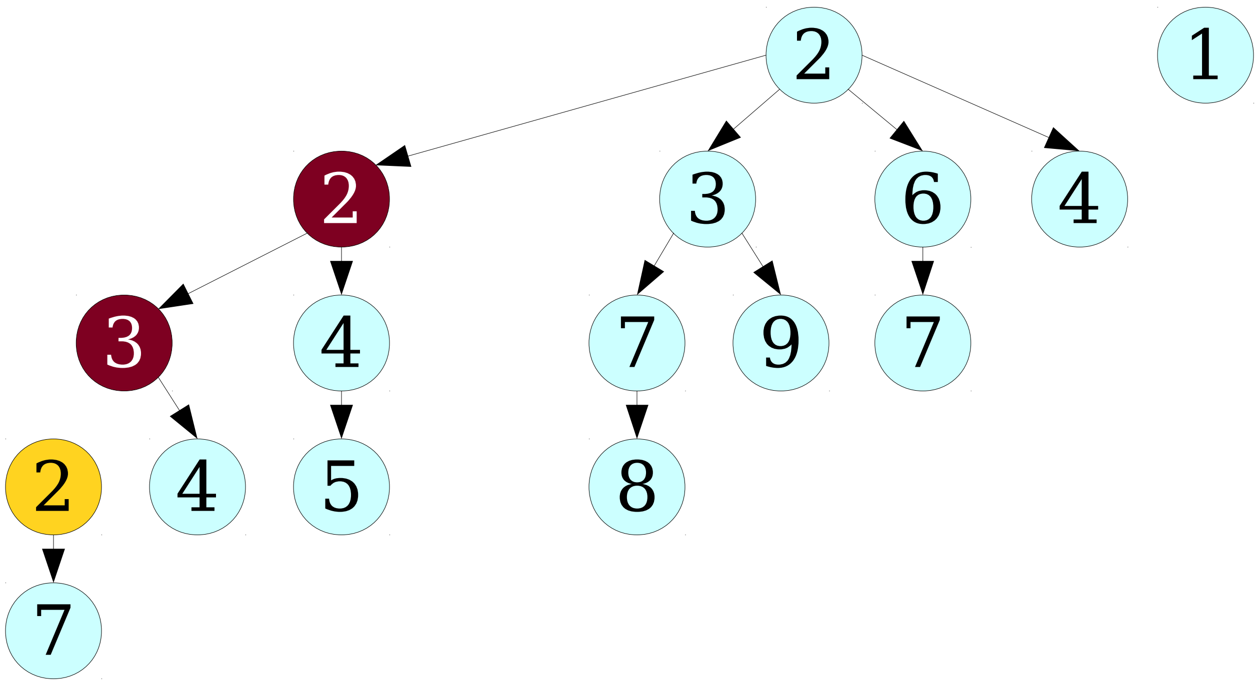
The Compromise

- Every non-root node is allowed to lose at most one child.
- If a non-root node loses two children, we cut it from its parent. (This might trigger more cuts.)
- We will ***mark*** nodes in the heap that have lost children to keep track of this fact.



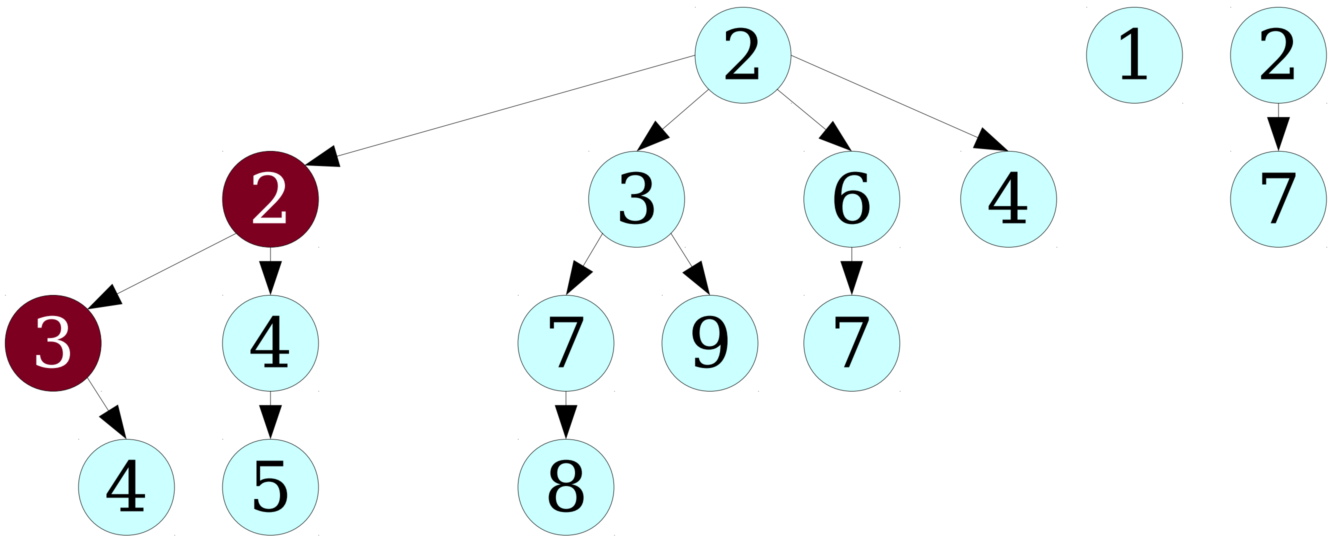
The Compromise

- Every non-root node is allowed to lose at most one child.
- If a non-root node loses two children, we cut it from its parent. (This might trigger more cuts.)
- We will ***mark*** nodes in the heap that have lost children to keep track of this fact.



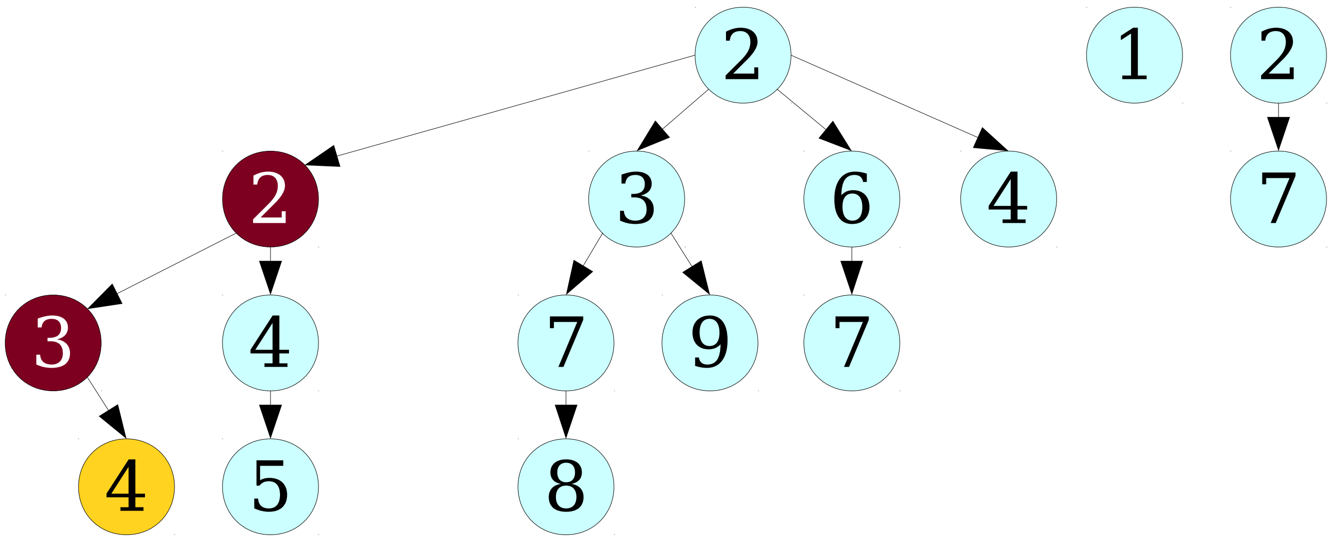
The Compromise

- Every non-root node is allowed to lose at most one child.
- If a non-root node loses two children, we cut it from its parent. (This might trigger more cuts.)
- We will ***mark*** nodes in the heap that have lost children to keep track of this fact.



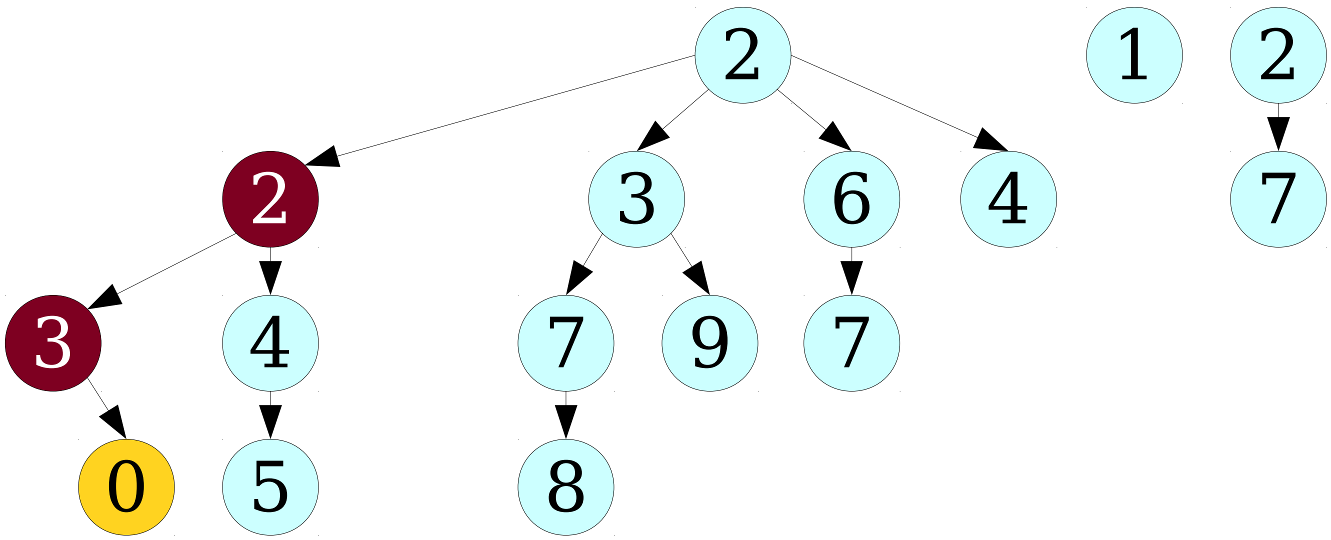
The Compromise

- Every non-root node is allowed to lose at most one child.
- If a non-root node loses two children, we cut it from its parent. (This might trigger more cuts.)
- We will ***mark*** nodes in the heap that have lost children to keep track of this fact.



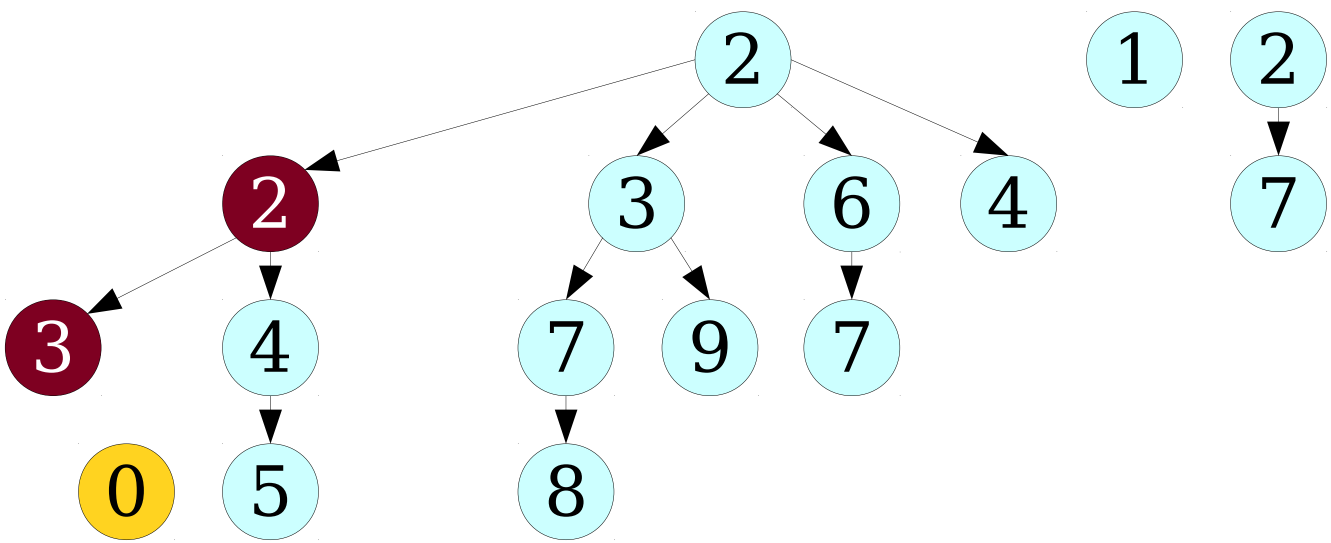
The Compromise

- Every non-root node is allowed to lose at most one child.
- If a non-root node loses two children, we cut it from its parent. (This might trigger more cuts.)
- We will ***mark*** nodes in the heap that have lost children to keep track of this fact.



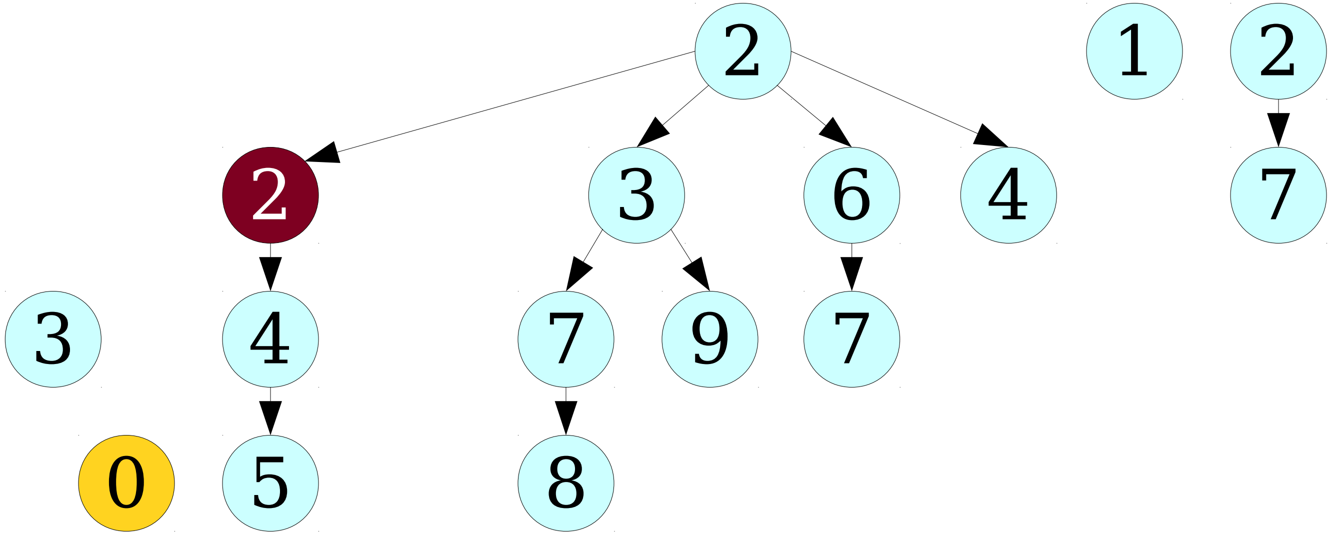
The Compromise

- Every non-root node is allowed to lose at most one child.
- If a non-root node loses two children, we cut it from its parent. (This might trigger more cuts.)
- We will ***mark*** nodes in the heap that have lost children to keep track of this fact.



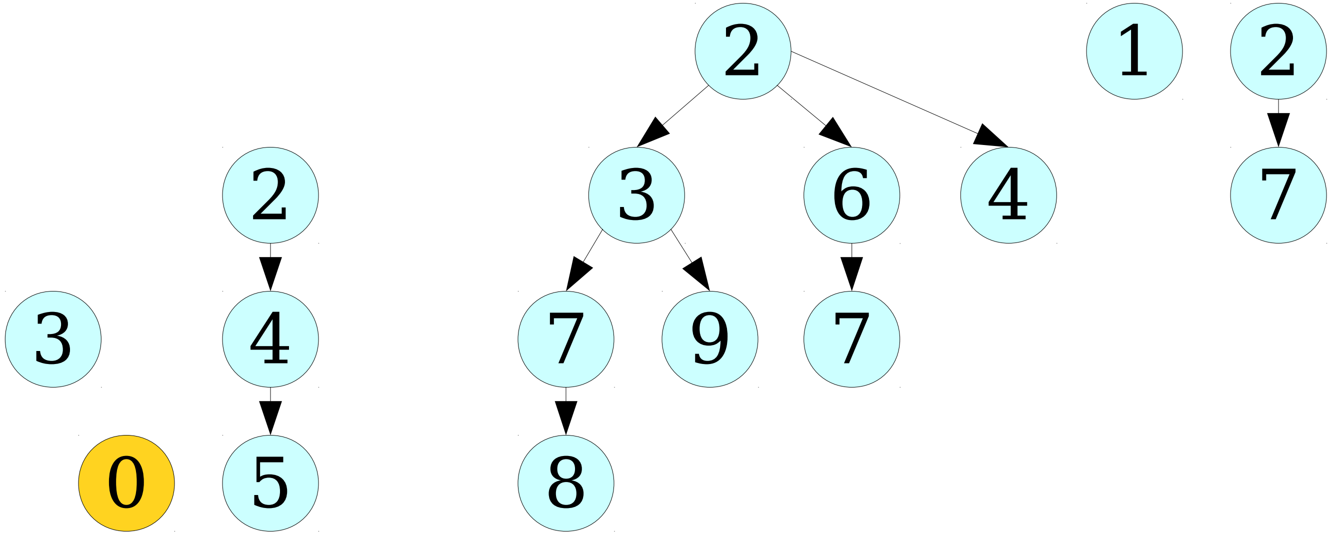
The Compromise

- Every non-root node is allowed to lose at most one child.
- If a non-root node loses two children, we cut it from its parent. (This might trigger more cuts.)
- We will ***mark*** nodes in the heap that have lost children to keep track of this fact.



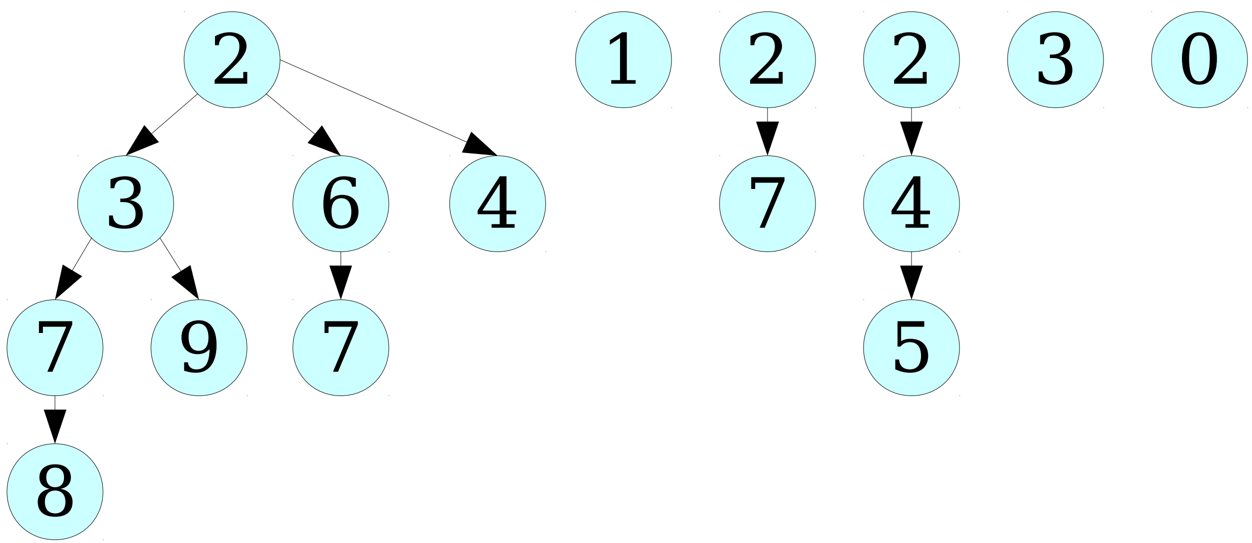
The Compromise

- Every non-root node is allowed to lose at most one child.
- If a non-root node loses two children, we cut it from its parent. (This might trigger more cuts.)
- We will ***mark*** nodes in the heap that have lost children to keep track of this fact.



The Compromise

- Every non-root node is allowed to lose at most one child.
- If a non-root node loses two children, we cut it from its parent. (This might trigger more cuts.)
- We will ***mark*** nodes in the heap that have lost children to keep track of this fact.

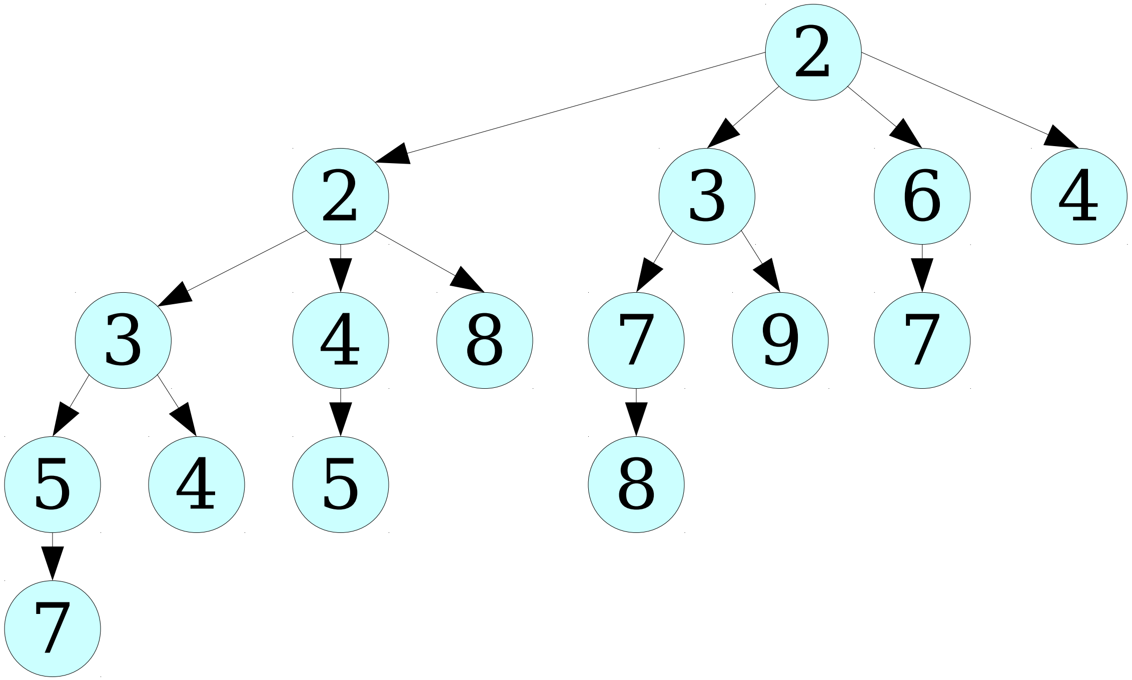


The Compromise

- To cut node *v* from its parent *p*:
  - Unmark *v*.
  - Cut *v* from *p*.
  - If *p* is not already marked and is not the root of a tree, mark it.
  - If *p* was already marked, recursively cut *p* from its parent.

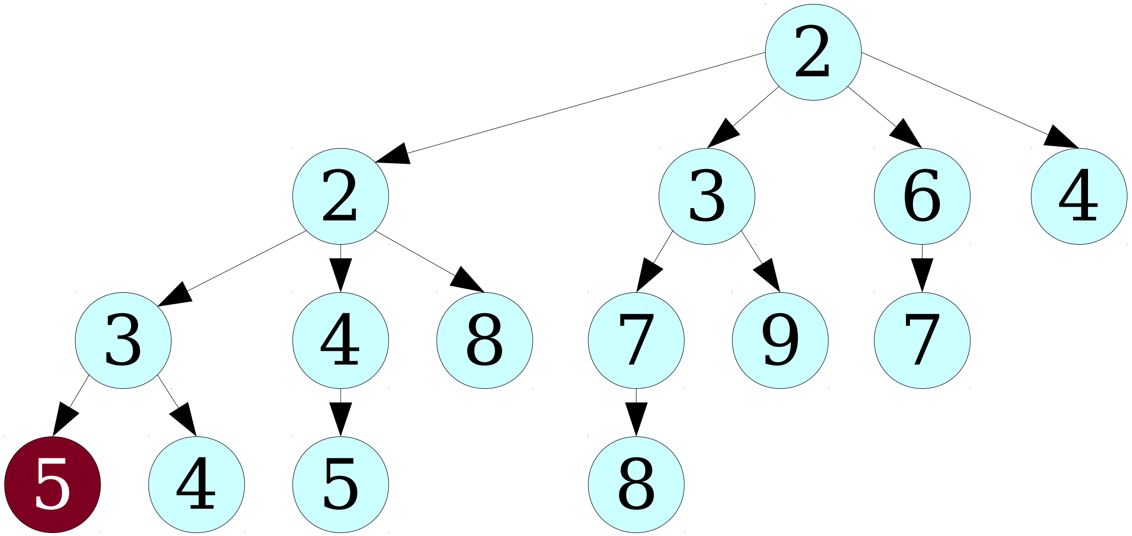
The Compromise

- If we do a few ***decrease-key***s, then the tree won't lose “too many” nodes.
- If we do many ***decrease-key***s, the information slowly propagates to the root.



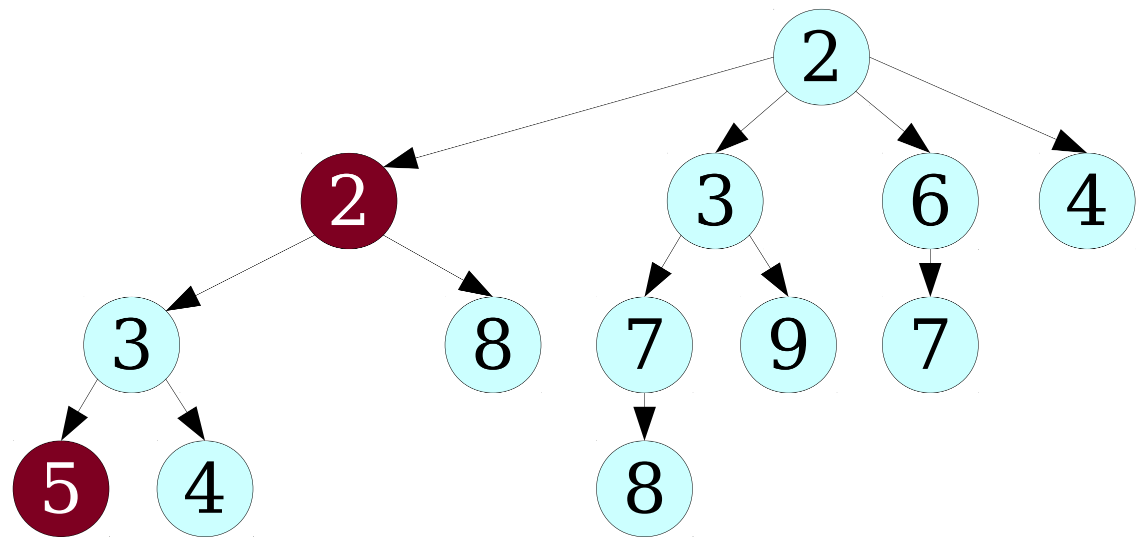
The Compromise

- If we do a few ***decrease-key***s, then the tree won't lose “too many” nodes.
- If we do many ***decrease-key***s, the information slowly propagates to the root.



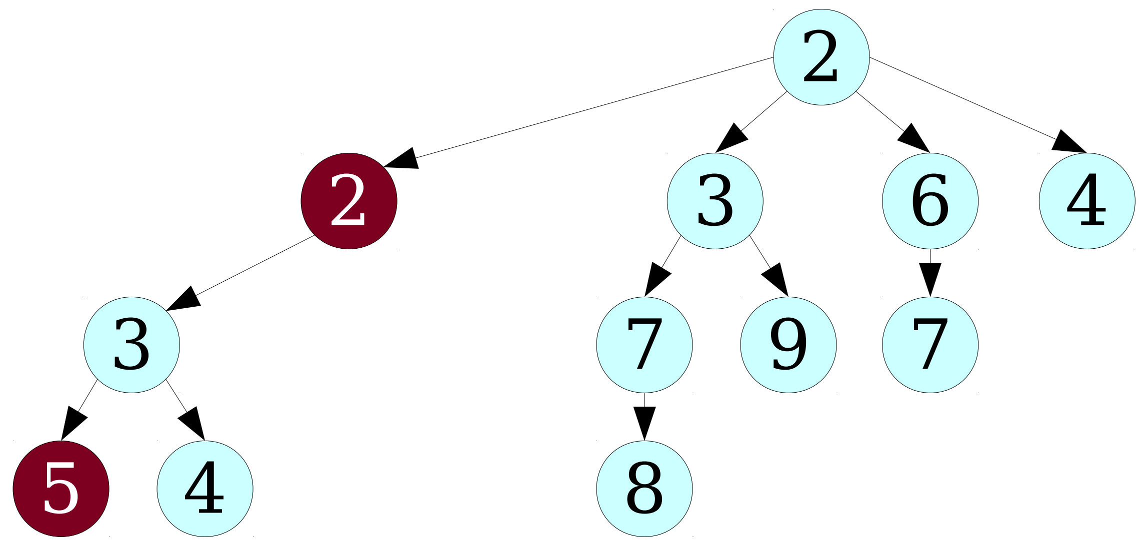
The Compromise

- If we do a few ***decrease-key***s, then the tree won't lose "too many" nodes.
- If we do many ***decrease-key***s, the information slowly propagates to the root.



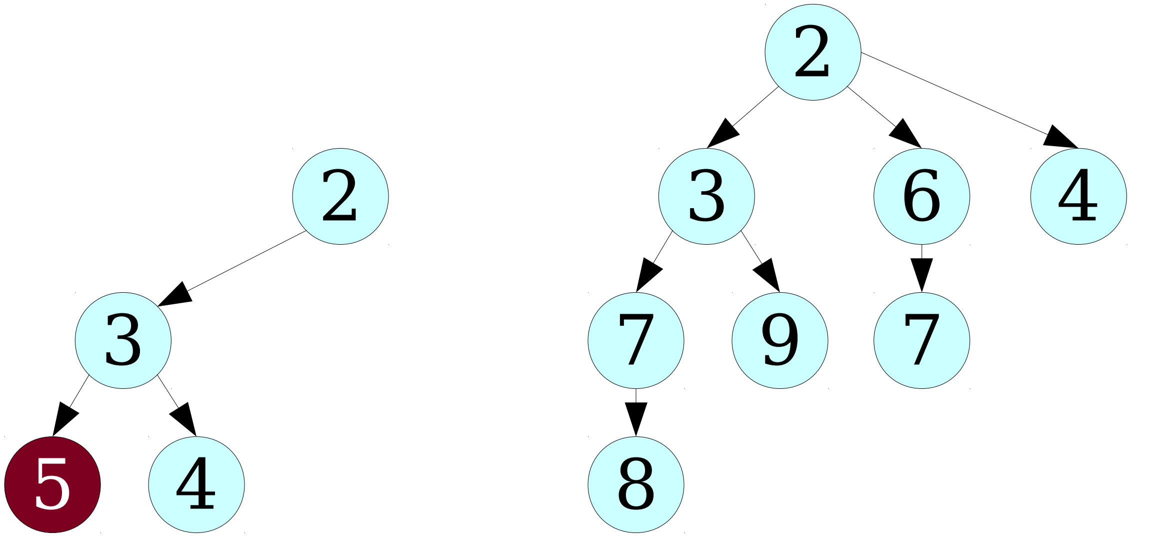
The Compromise

- If we do a few ***decrease-key***s, then the tree won't lose "too many" nodes.
- If we do many ***decrease-key***s, the information slowly propagates to the root.



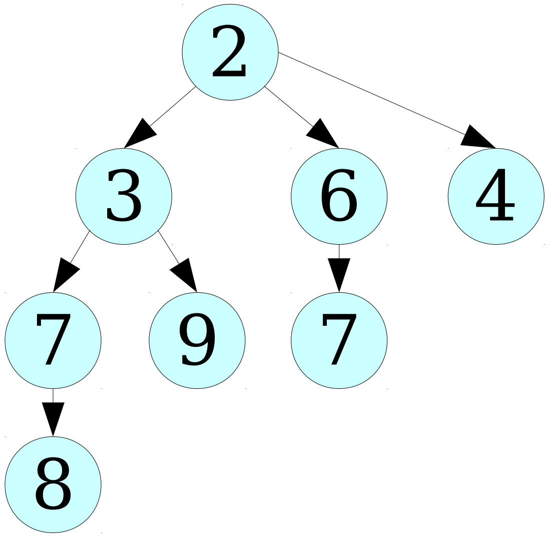
The Compromise

- If we do a few ***decrease-key***s, then the tree won't lose “too many” nodes.
- If we do many ***decrease-key***s, the information slowly propagates to the root.



The Compromise

- If we do a few ***decrease-key***s, then the tree won't lose "too many" nodes.
- If we do many ***decrease-key***s, the information slowly propagates to the root.



Dr. Strange Runtime Analysis

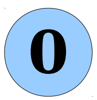
Or: How I Learned to Stop Worrying and Love the Cut

Two Extremes

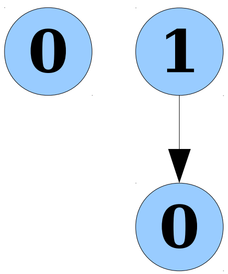
- If we never do any ***decrease-key***s, then the trees in our data structure are all binomial trees.
- Each tree of order k has 2^k nodes in it, so the tree sizes grow exponentially and the runtime of an ***extract-min*** is O(log n).
- On the other hand, suppose that all trees in the binomial heap have lost the maximum possible number of nodes.
- In that case, how many nodes will each tree have?

Maximally-Damaged Trees

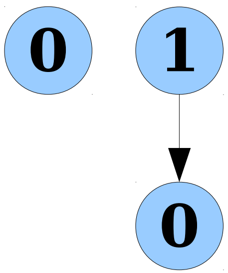
Maximally-Damaged Trees



Maximally-Damaged Trees

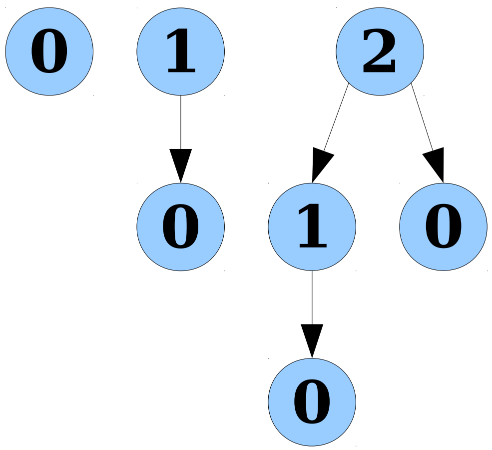


Maximally-Damaged Trees

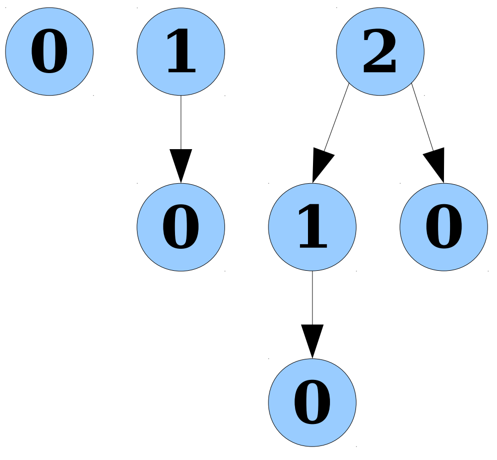


We can't cut any nodes from this tree without making the root node have order 0.

Maximally-Damaged Trees

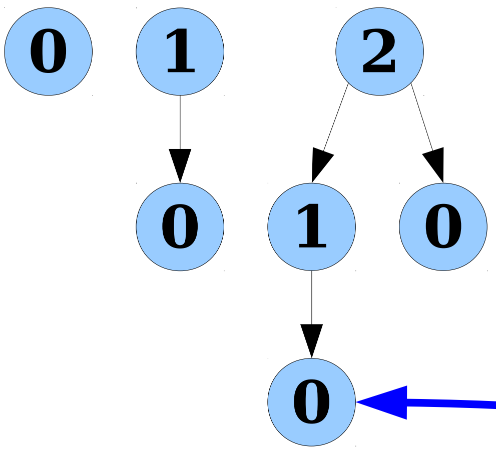


Maximally-Damaged Trees



We can't cut any of the root's children without decreasing its order.

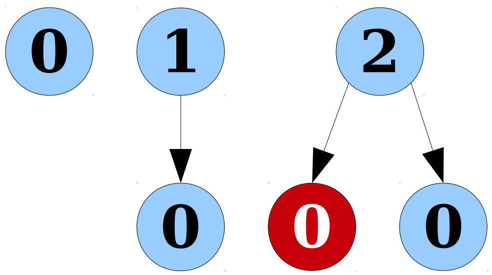
Maximally-Damaged Trees



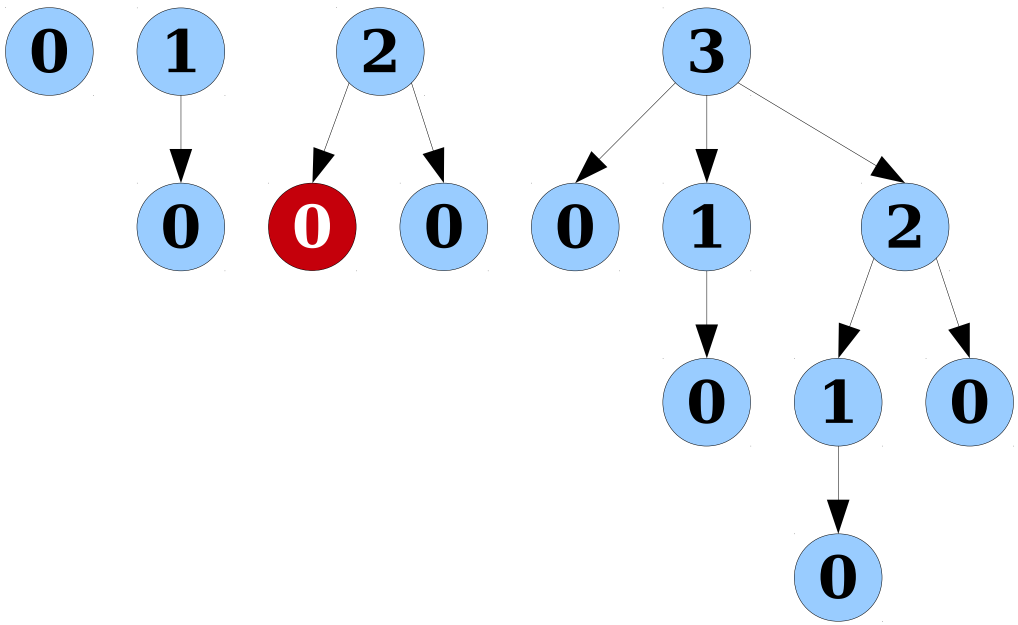
We can't cut any of the root's children without decreasing its order.

However, we can cut this node, leaving the root node with two children.

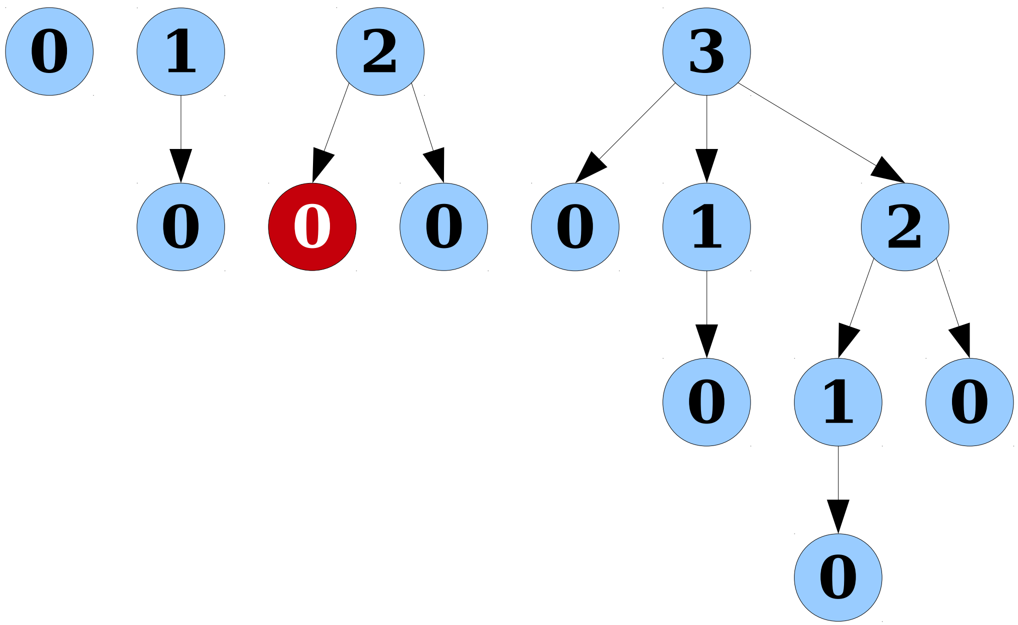
Maximally-Damaged Trees



Maximally-Damaged Trees

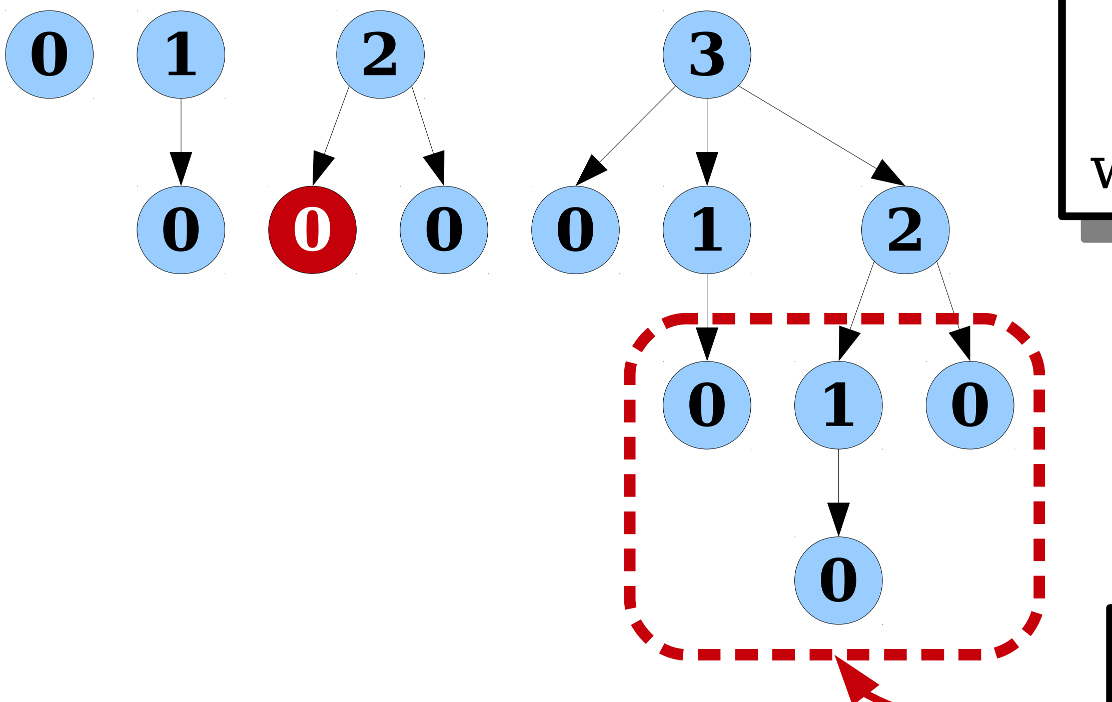


Maximally-Damaged Trees



As before, we can't cut any of the root's children without decreasing its order.

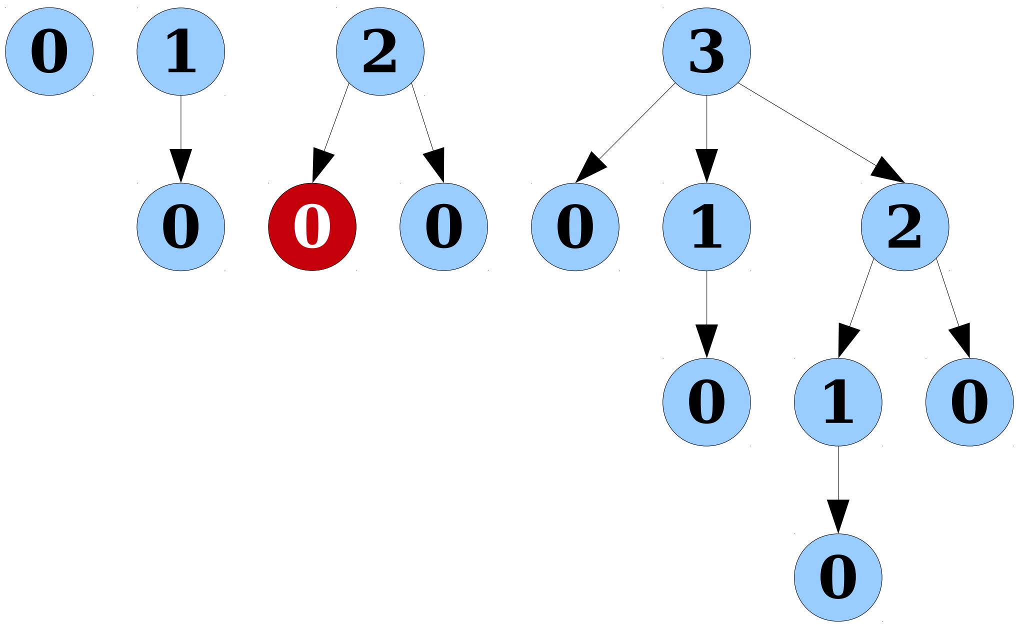
Maximally-Damaged Trees



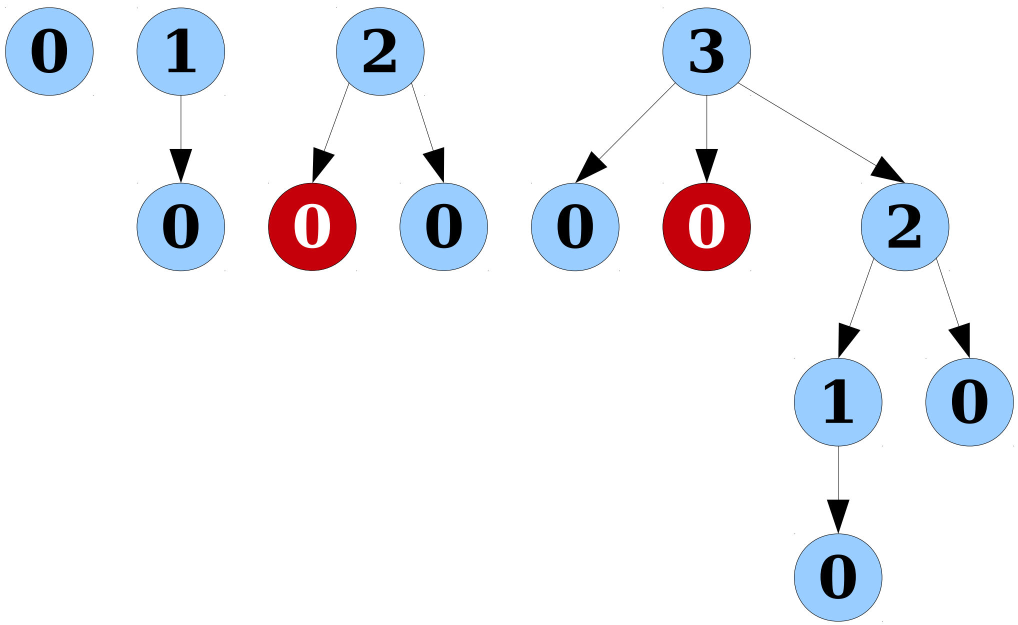
As before, we can't cut any of the root's children without decreasing its order.

However, any nodes below the second layer are fair game to be eliminated.

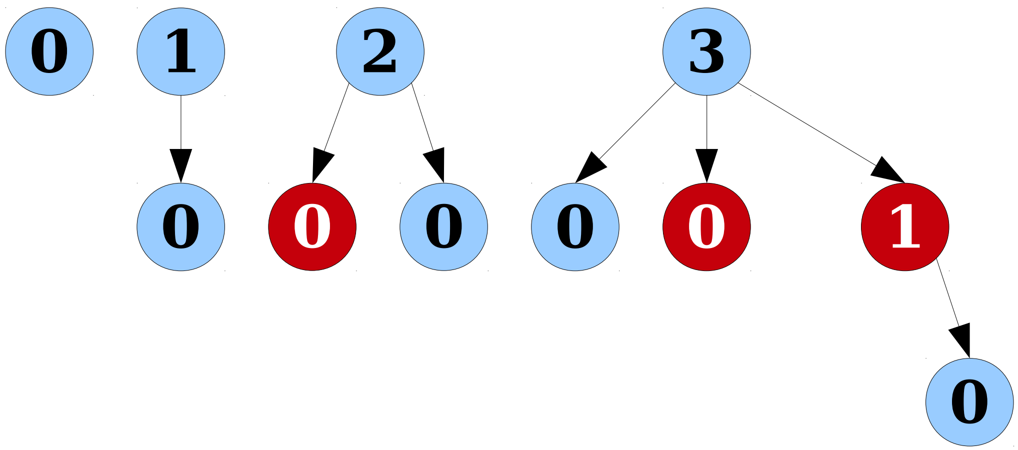
Maximally-Damaged Trees



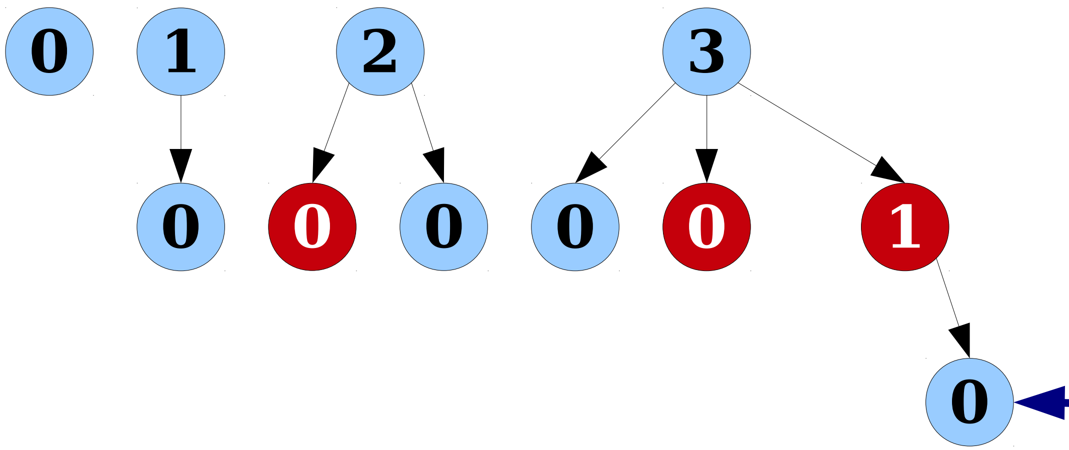
Maximally-Damaged Trees



Maximally-Damaged Trees

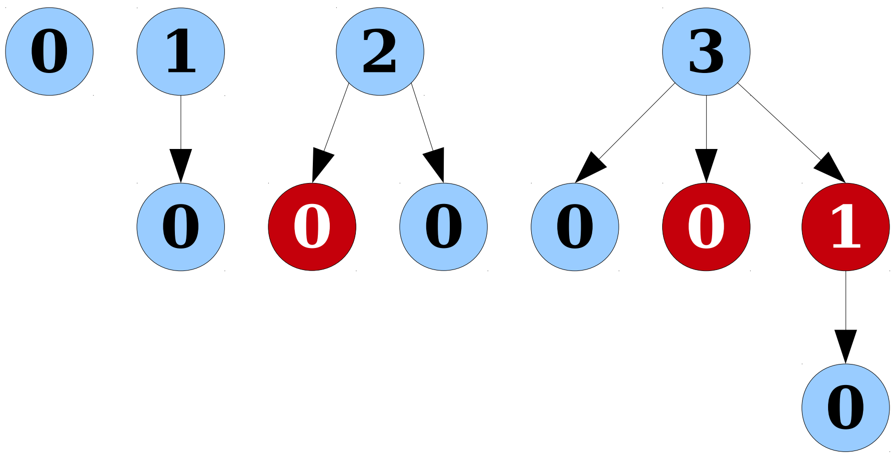


Maximally-Damaged Trees

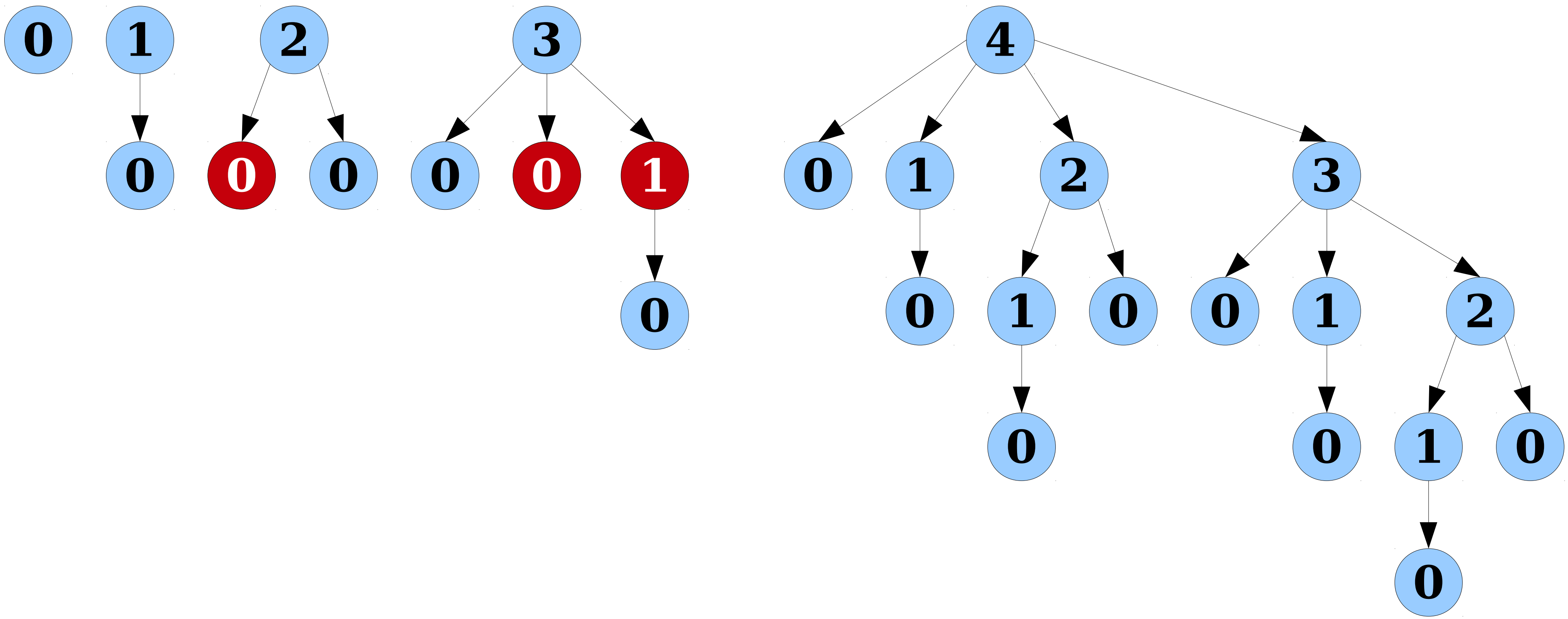


We can't cut this node without triggering a cascading cut, so we're done.

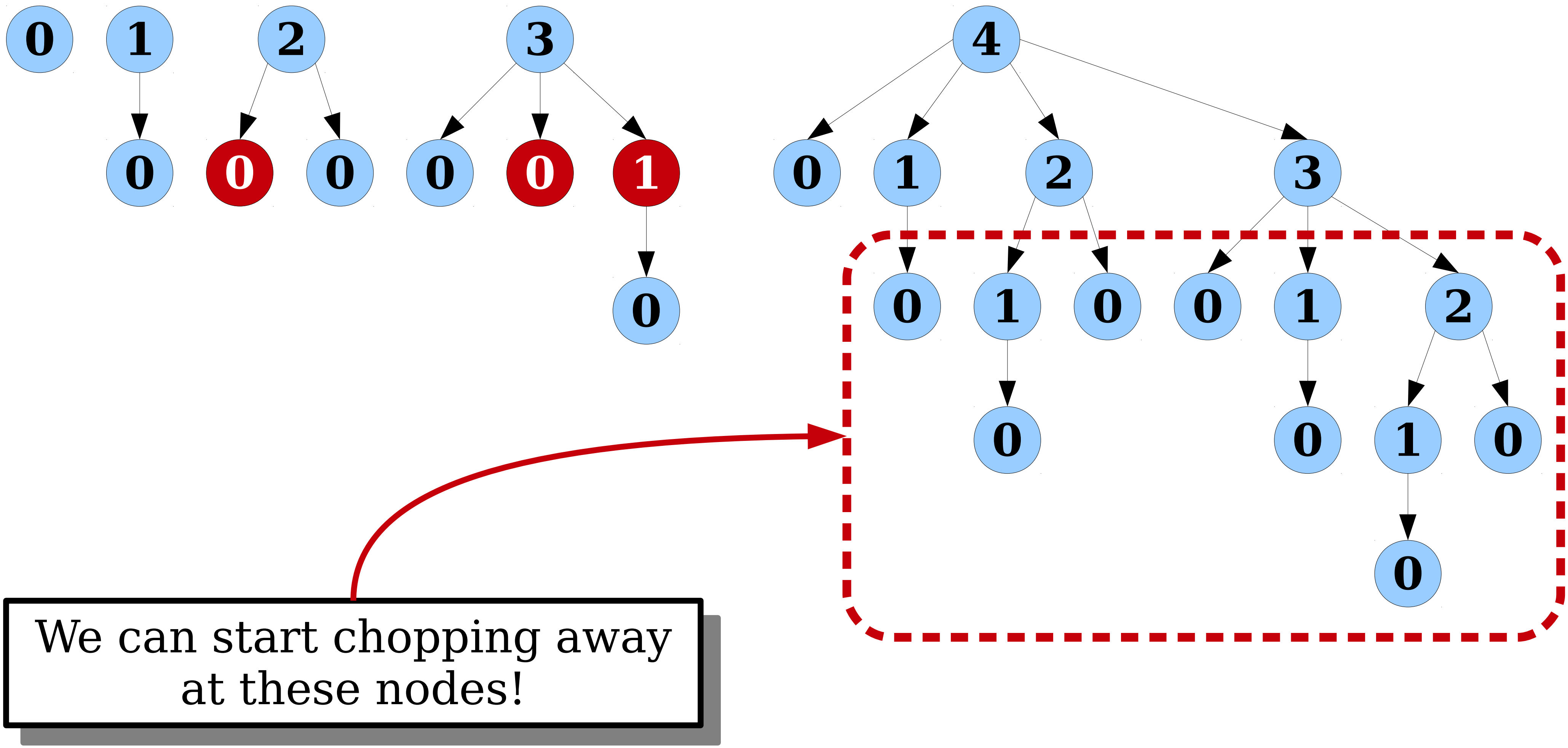
Maximally-Damaged Trees



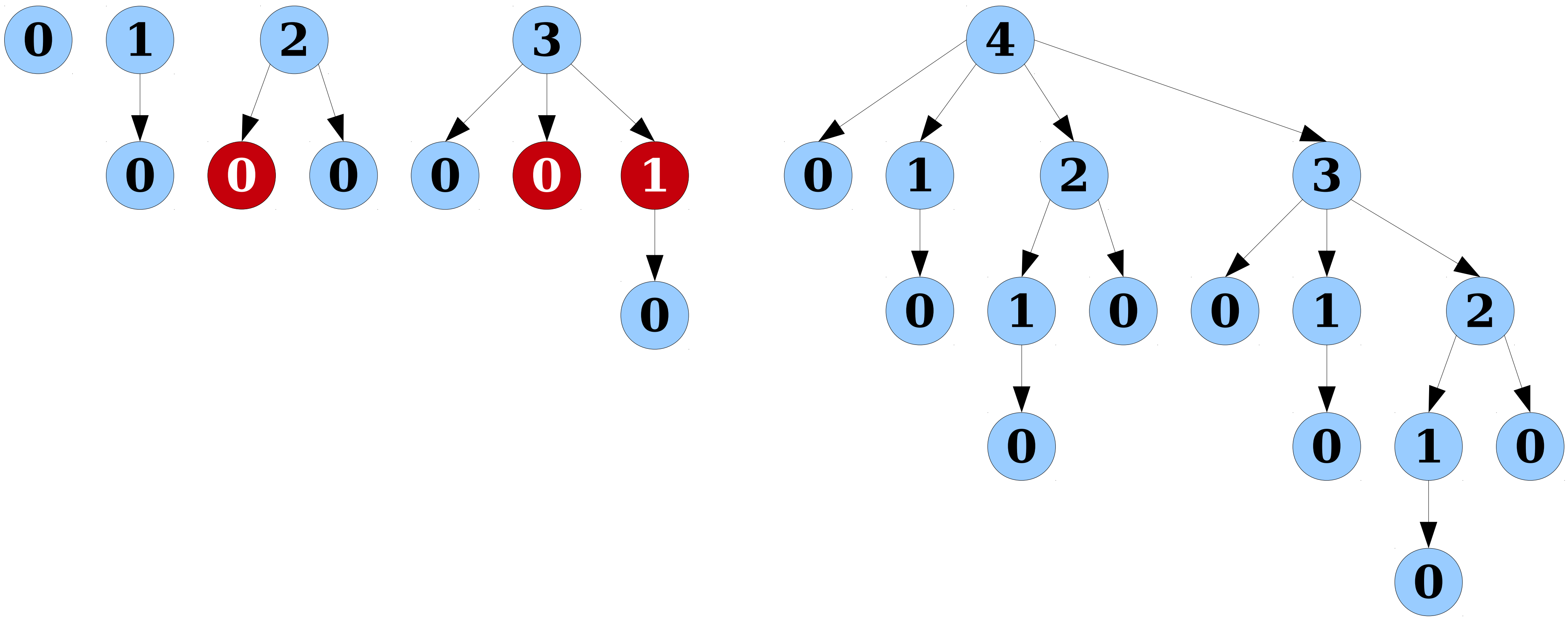
Maximally-Damaged Trees



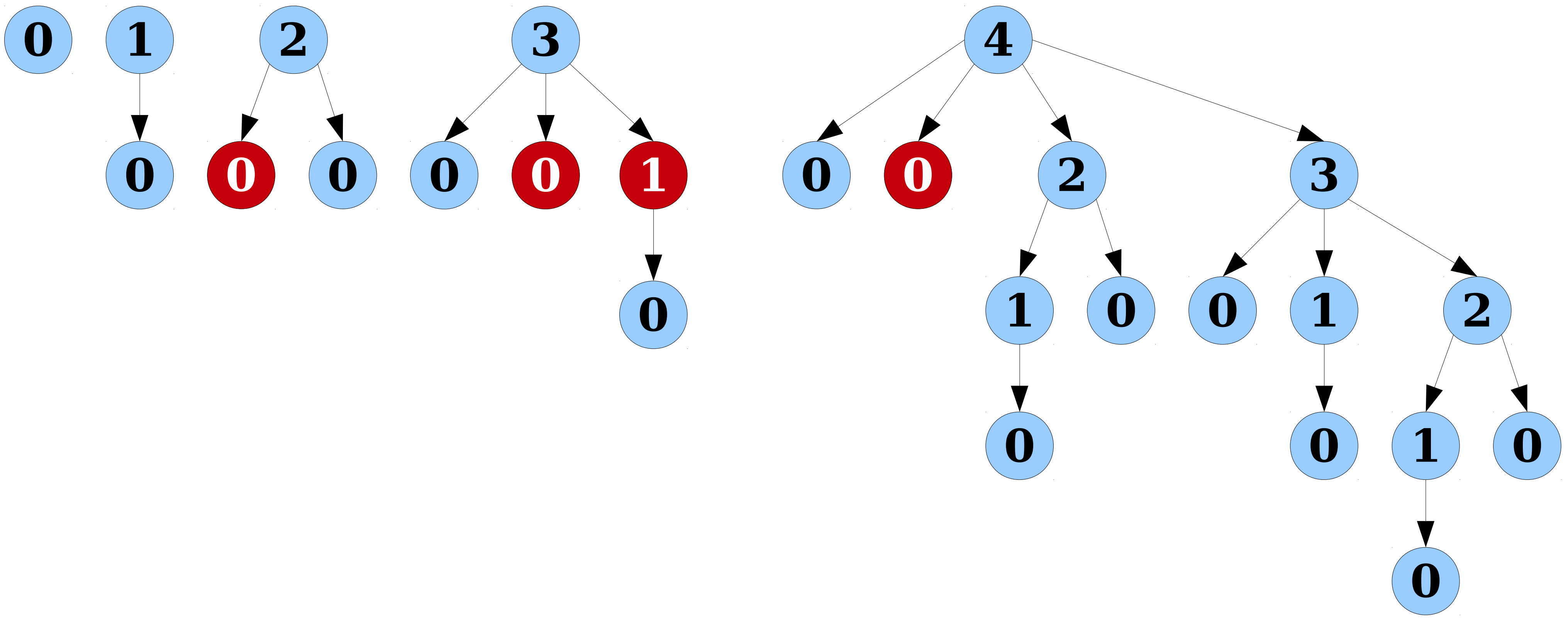
Maximally-Damaged Trees



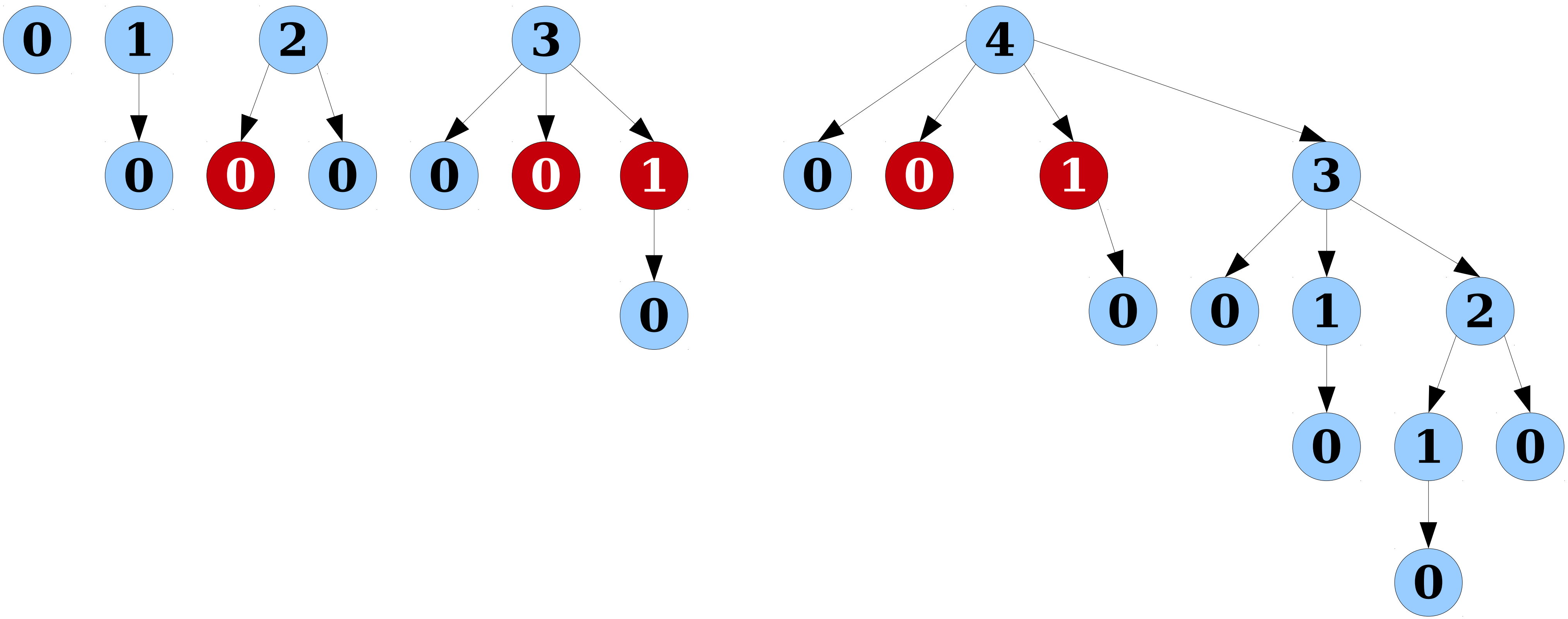
Maximally-Damaged Trees



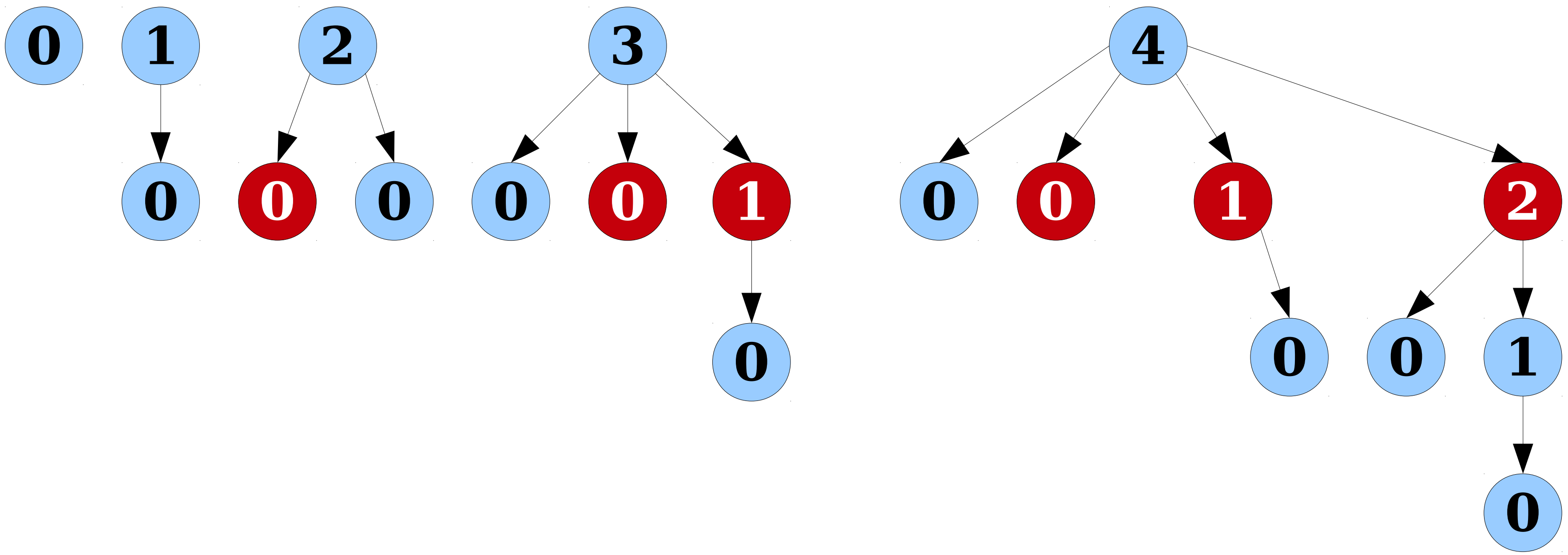
Maximally-Damaged Trees



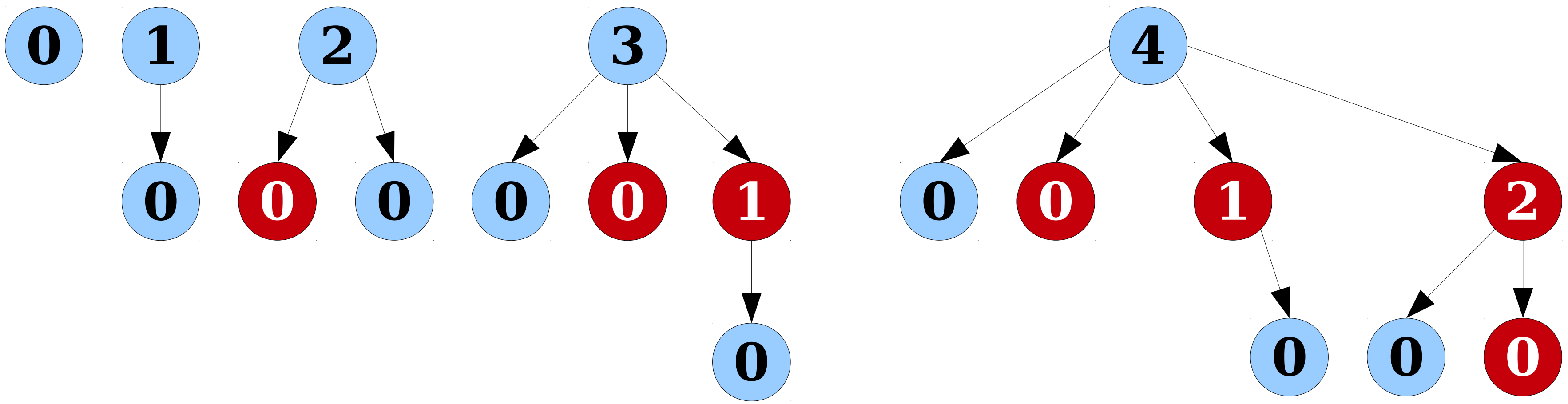
Maximally-Damaged Trees



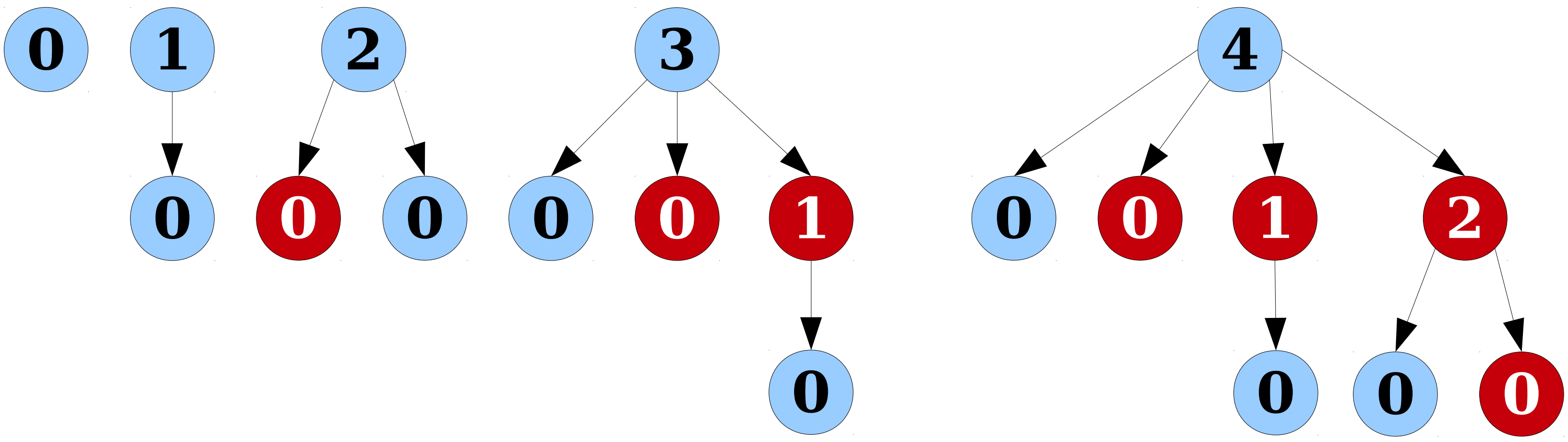
Maximally-Damaged Trees



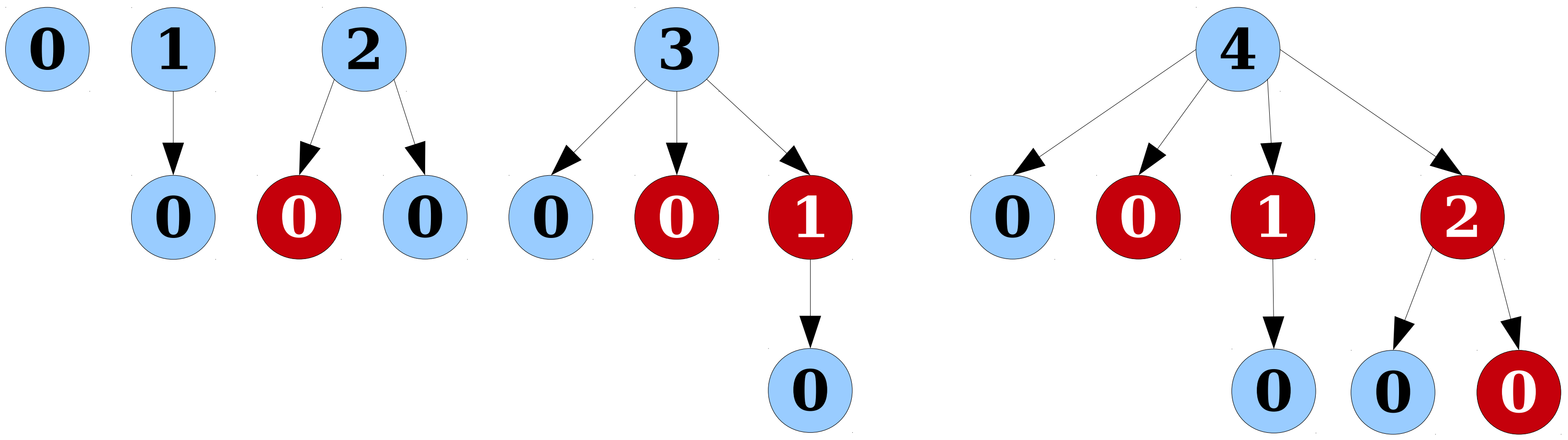
Maximally-Damaged Trees

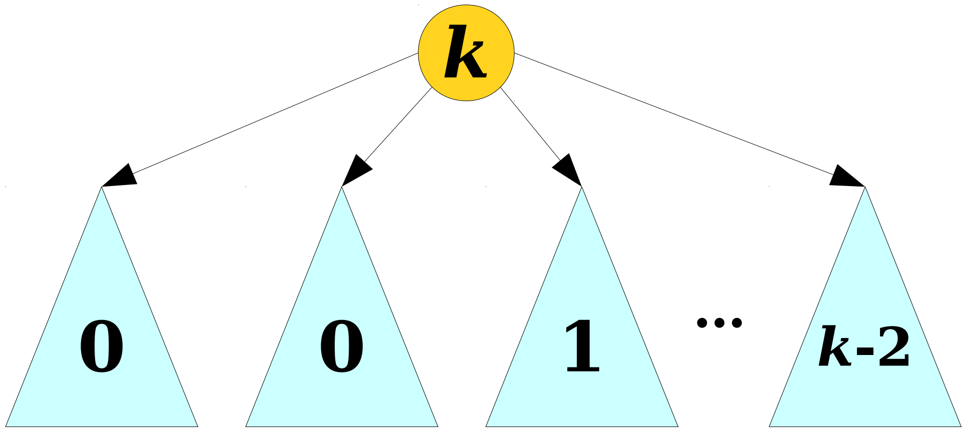


Maximally-Damaged Trees

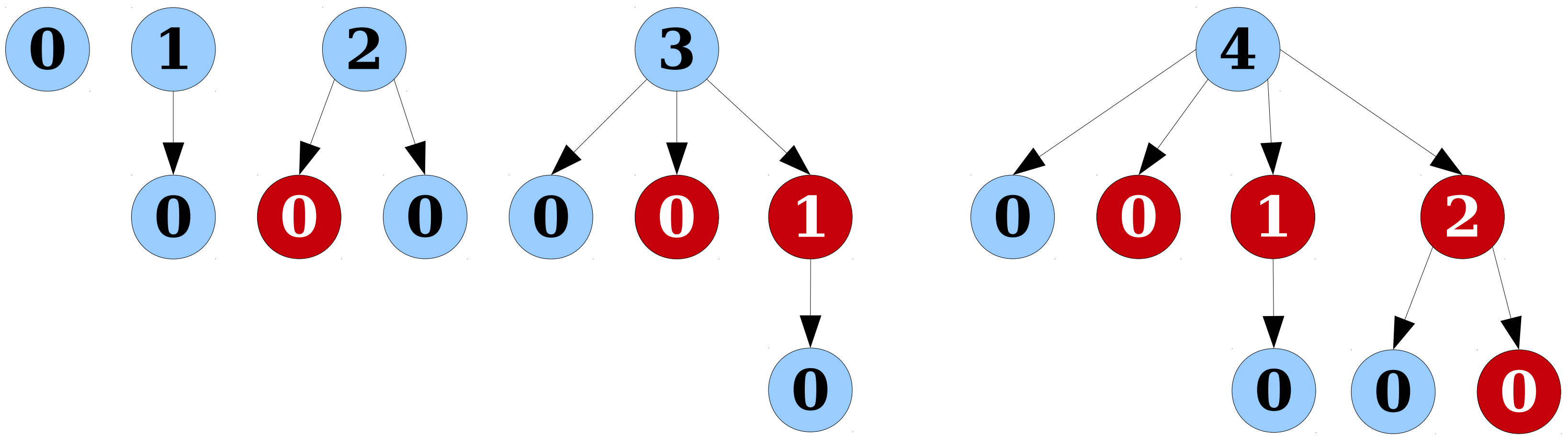


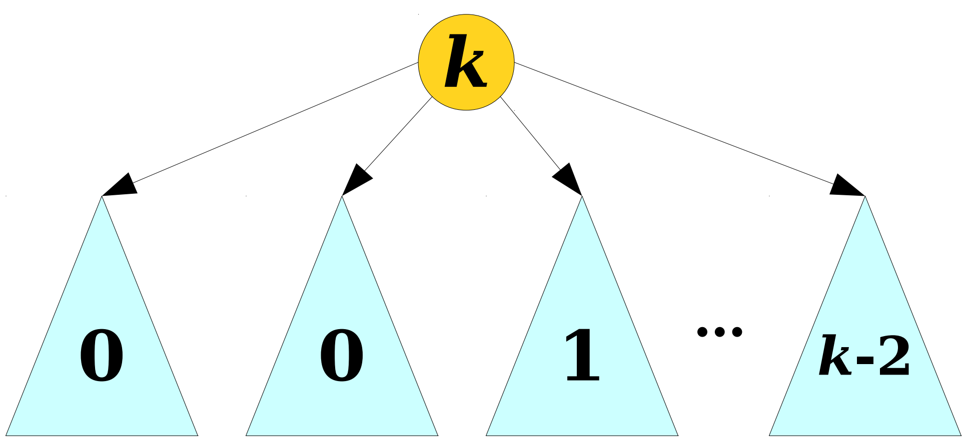
Maximally-Damaged Trees





Maximally-Damaged Trees

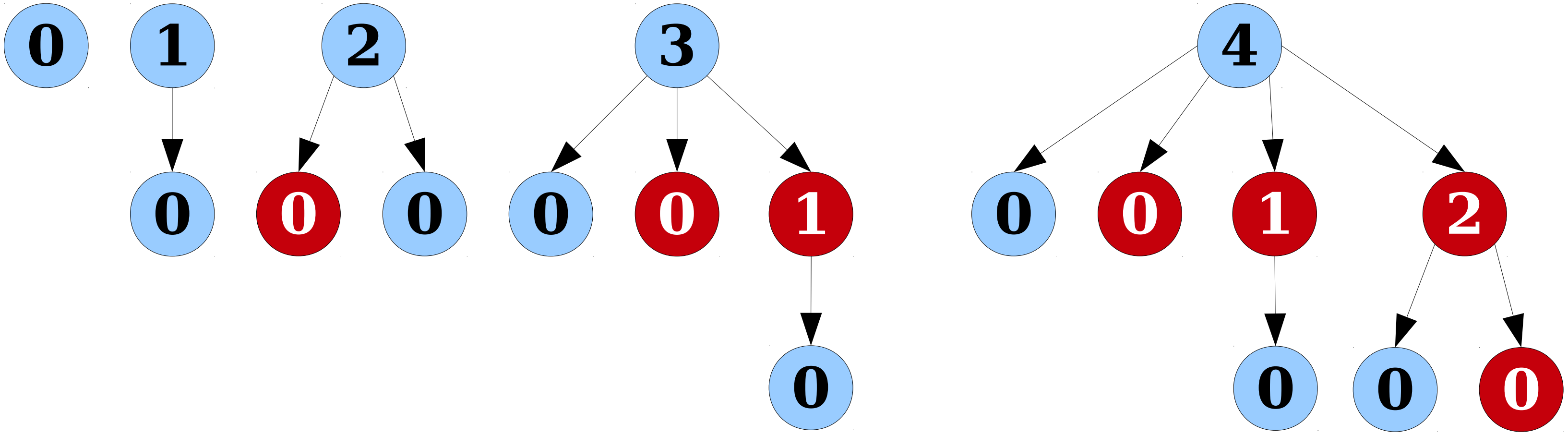




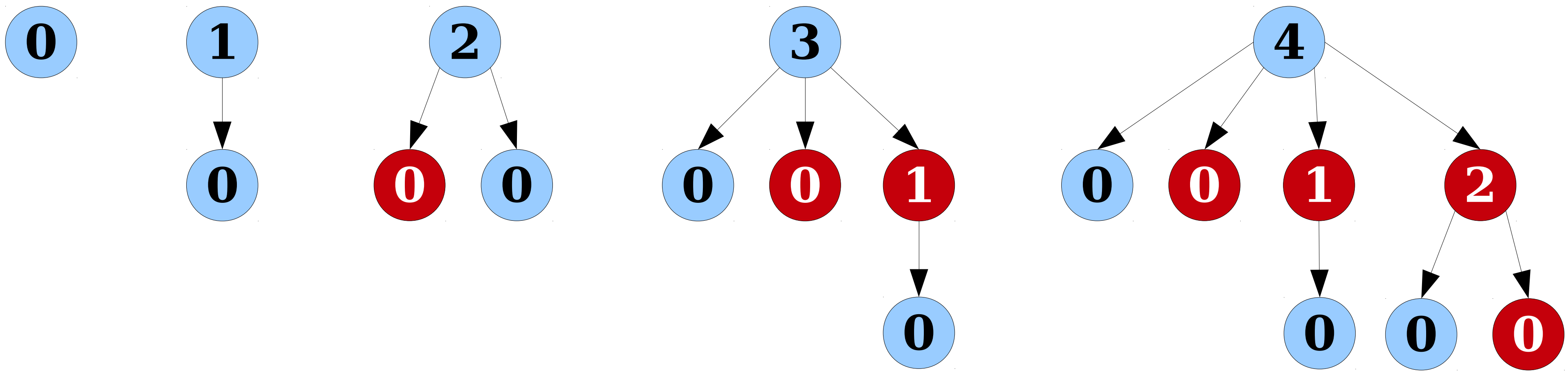
A ***maximally-damaged tree of order k*** is a node whose children are maximally-damaged trees of orders

0, 0, 1, 2, 3, …, *k* – 2.

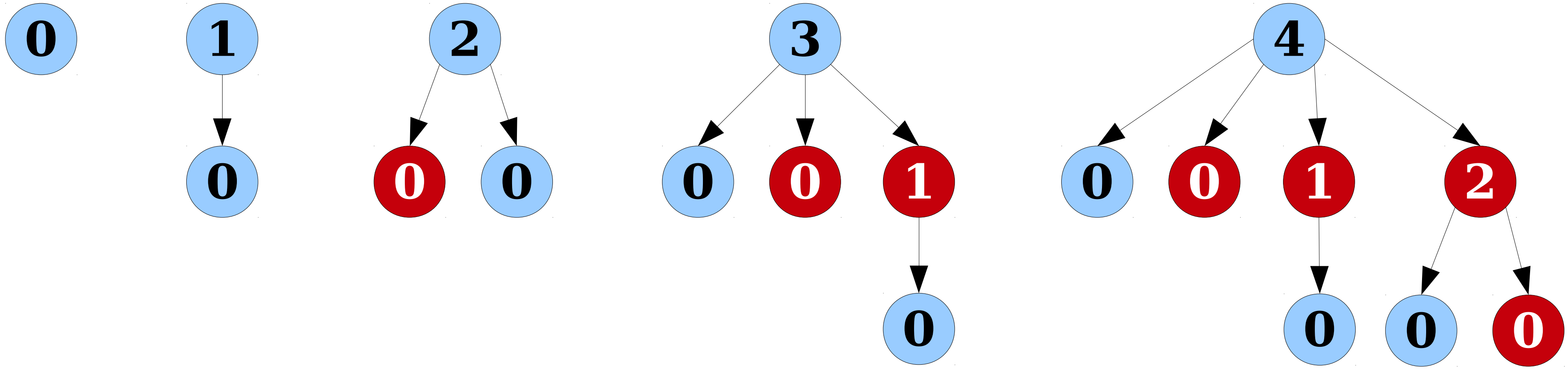
Maximally-Damaged Trees



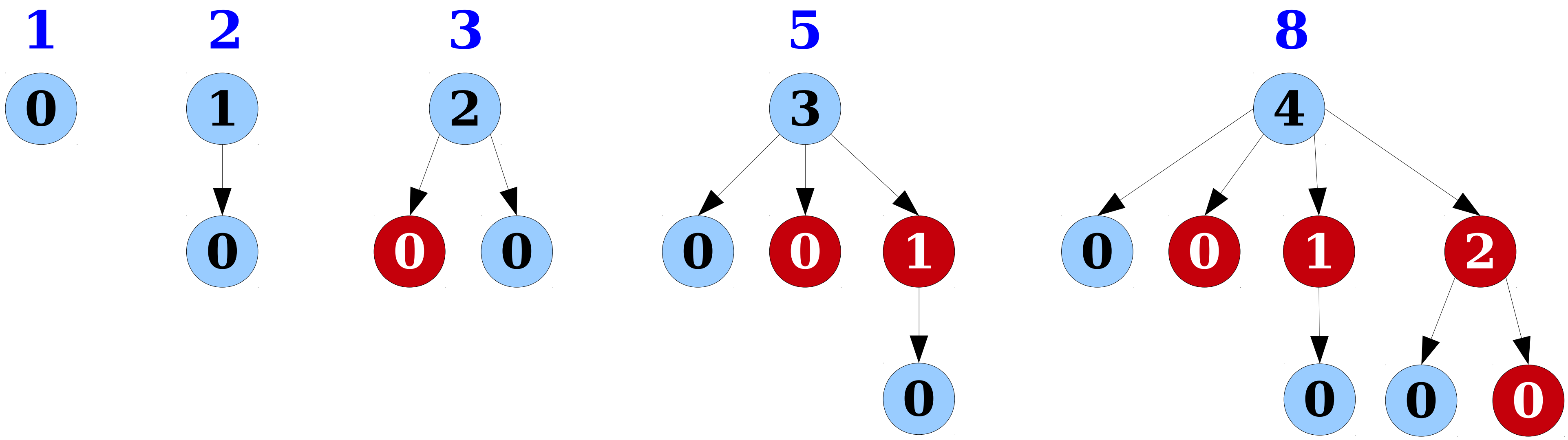
Maximally-Damaged Trees



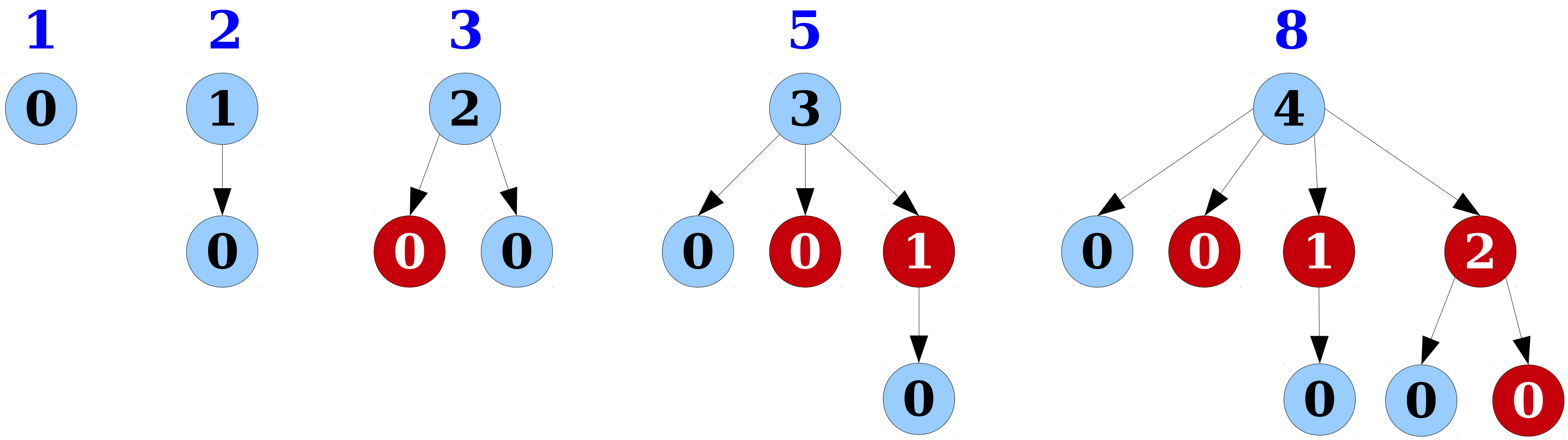
Maximally-Damaged Trees



Maximally-Damaged Trees



Maximally-Damaged Trees



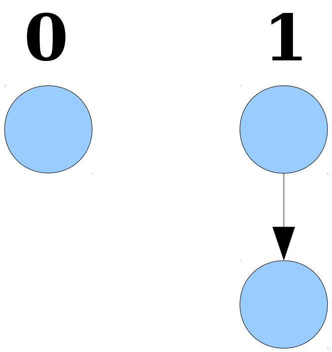
Claim: The minimum number of nodes in a tree of order *k* is F_{k+2}

Maximally-Damaged Trees

- ***Theorem:*** The number of nodes in a maximally-damaged tree of order k is F_{k+2}.
- ***Proof:*** Induction.

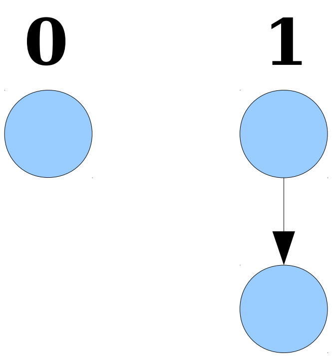
Maximally-Damaged Trees

- ***Theorem:*** The number of nodes in a maximally-damaged tree of order k is F_{k+2}.
- ***Proof:*** Induction.



Maximally-Damaged Trees

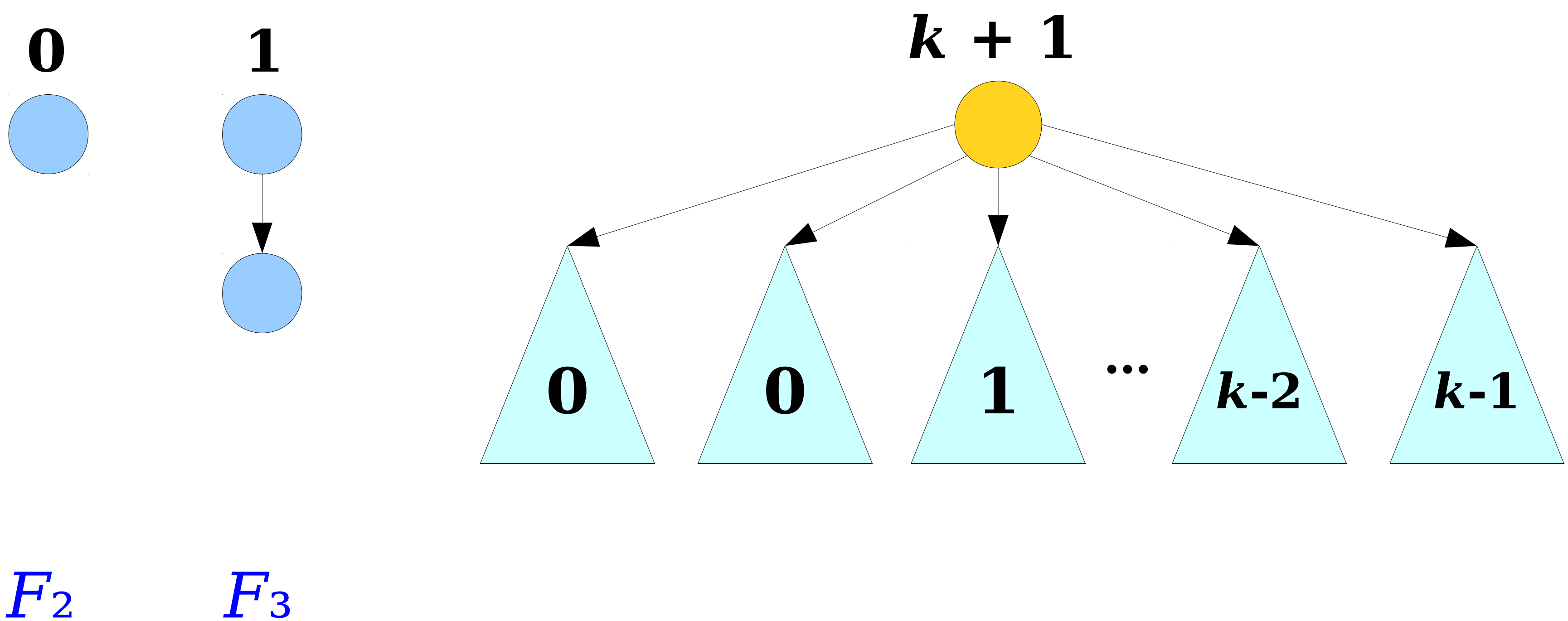
- ***Theorem:*** The number of nodes in a maximally-damaged tree of order k is F_{k+2}.
- ***Proof:*** Induction.



F_2 F_3

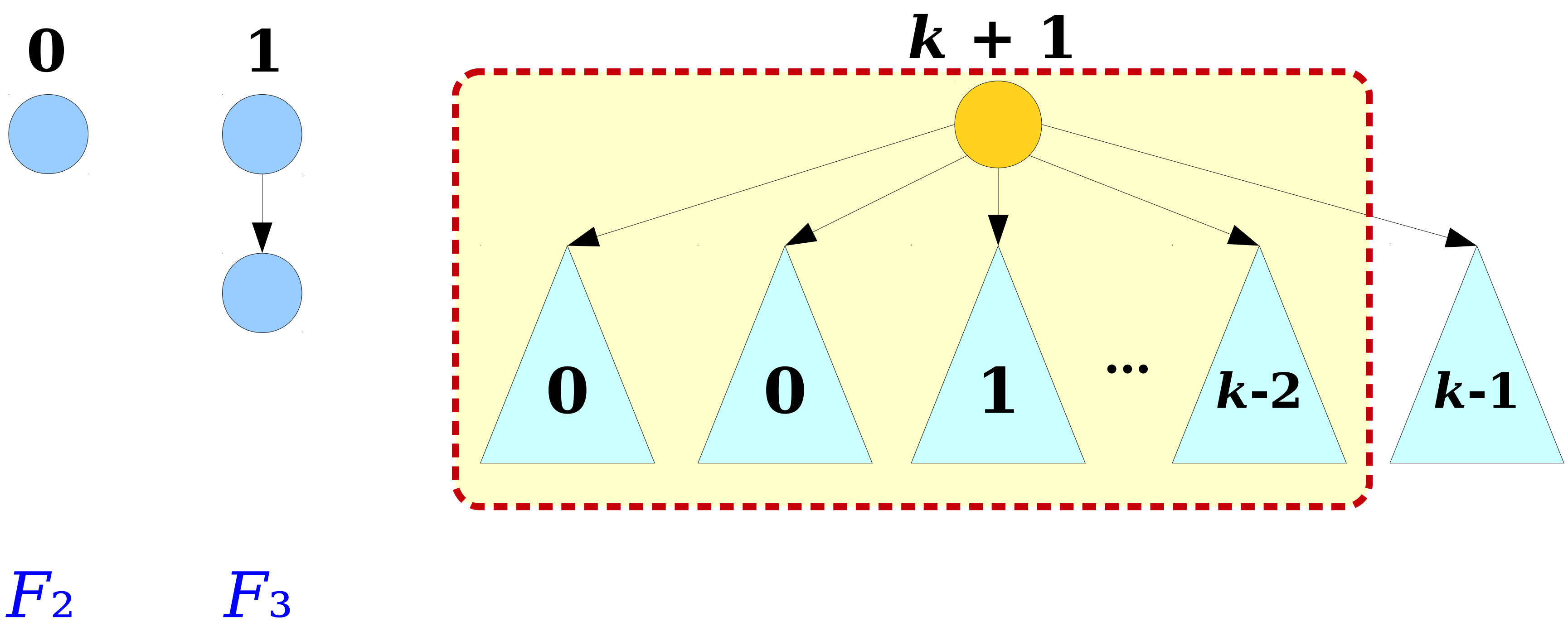
Maximally-Damaged Trees

- ***Theorem:*** The number of nodes in a maximally-damaged tree of order k is F_{k+2}.
- ***Proof:*** Induction.



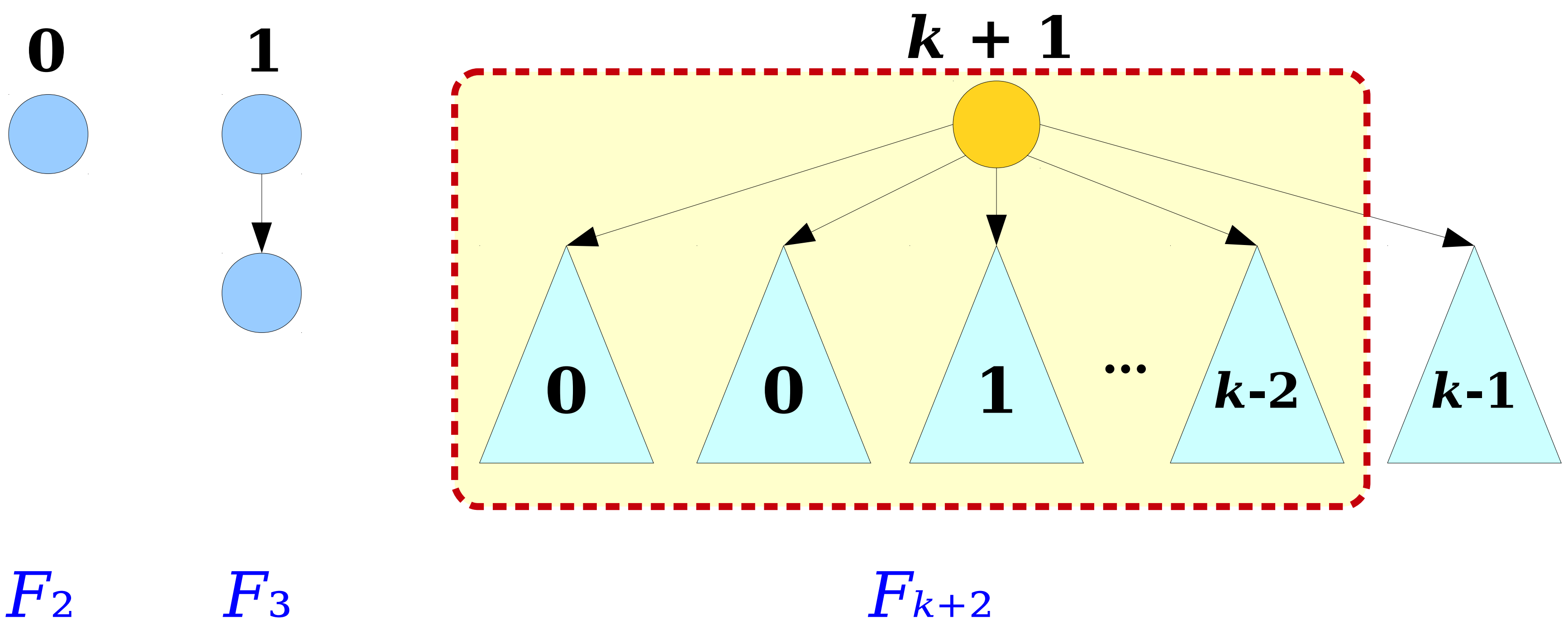
Maximally-Damaged Trees

- ***Theorem:*** The number of nodes in a maximally-damaged tree of order k is F_{k+2}.
- ***Proof:*** Induction.



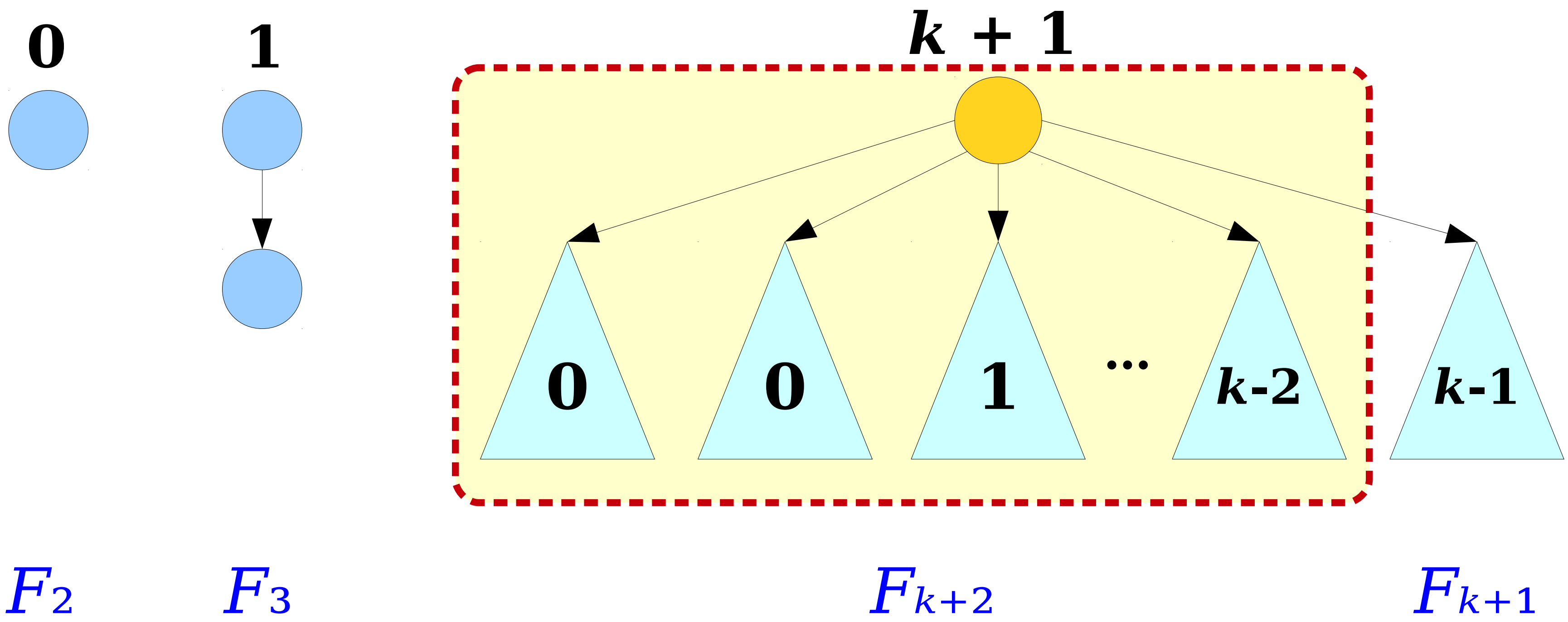
Maximally-Damaged Trees

- ***Theorem:*** The number of nodes in a maximally-damaged tree of order k is F_{k+2}.
- ***Proof:*** Induction.



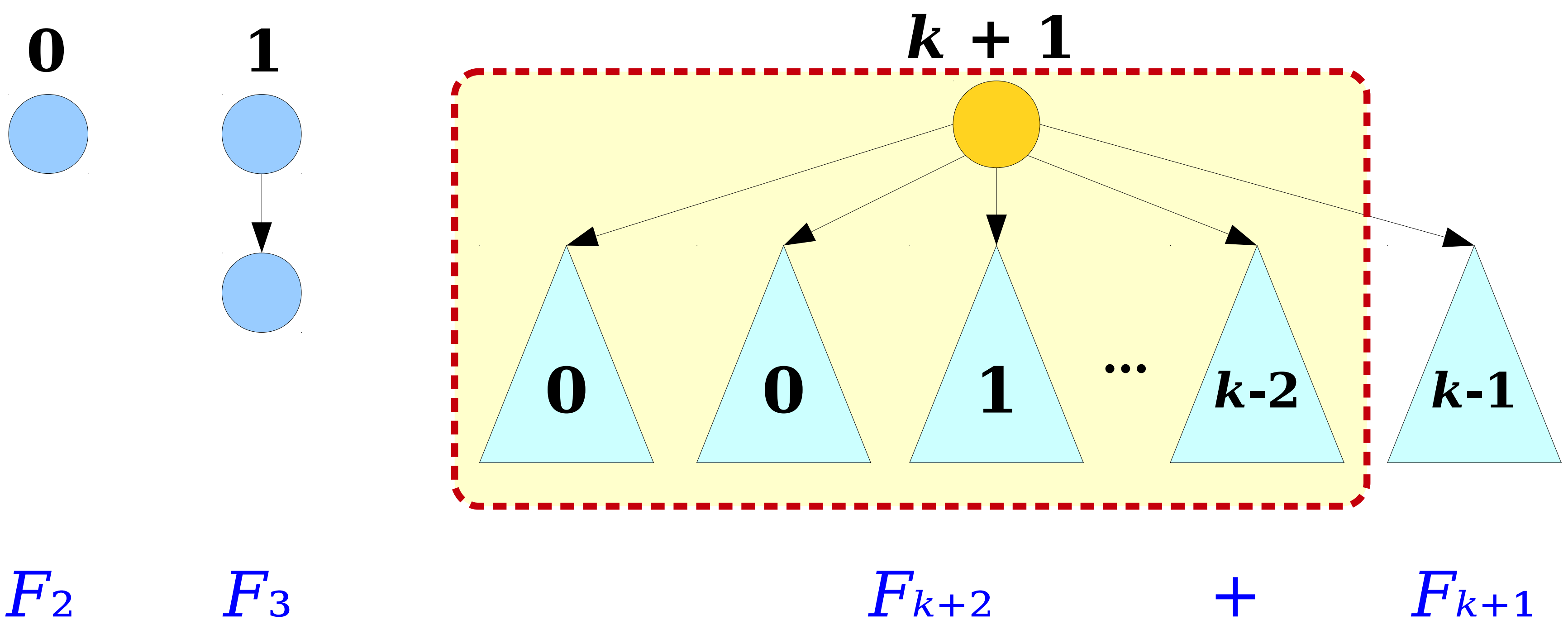
Maximally-Damaged Trees

- ***Theorem:*** The number of nodes in a maximally-damaged tree of order k is F_{k+2}.
- ***Proof:*** Induction.



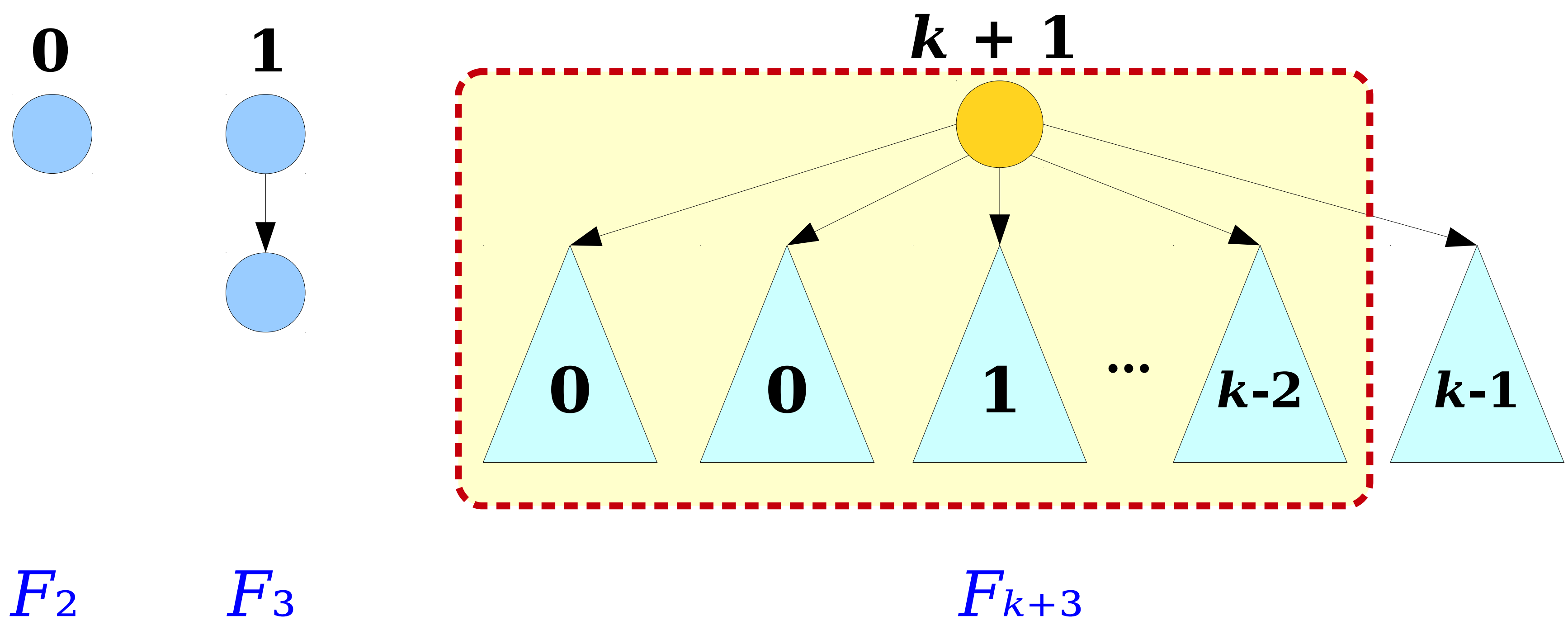
Maximally-Damaged Trees

- ***Theorem:*** The number of nodes in a maximally-damaged tree of order k is F_{k+2}.
- ***Proof:*** Induction.



Maximally-Damaged Trees

- ***Theorem:*** The number of nodes in a maximally-damaged tree of order k is F_{k+2}.
- ***Proof:*** Induction.



φ-bonacci Numbers

- ***Fact:*** For $n \geq 2$, we have $F_n \geq \varphi^{n-2}$, where φ is the golden ratio:

$$\varphi \approx 1.61803398875...$$

- ***Claim:*** In our modified data structure, the amortized cost of an ***extract-min*** is O(log *n*).
- ***Proof:*** In a tree of order *k*, there are at least $F_{k+2} \geq \varphi^k$ nodes. Therefore, a tree of order *k* has exponentially many nodes in it, so the previous analysis still holds. ■

Fibonacci Heaps

- A ***Fibonacci heap*** is a lazy binomial heap where ***decrease-key*** is implemented using the earlier cutting-and-marking scheme.
- Operation runtimes:
  - ***enqueue***: O(1)
  - ***meld***: O(1)
  - ***find-min***: O(1)
  - ***extract-min***: O(log n) amortized
  - ***decrease-key***: Up next!

Analyzing ***decrease-key***

- When performing a ***decrease-key***, the runtime depends on the number of total cuts made.
  - These cuts only “cascade” if we cut from a node whose parent is already marked.
- The runtime of ***decrease-key*** is specifically $\Theta(C)$, where C is the number of cuts made.
- What is the *amortized* cost of a ***decrease-key***?

Refresher: Our Choice of Φ

- In our amortized analysis of lazy binomial heaps, we set Φ to be the number of trees in the heap.
- With this choice of Φ, we obtained these amortized time bounds:
  - ***enqueue***: O(1)
  - ***meld***: O(1)
  - ***find-min***: O(1)
  - ***extract-min***: O(log *n*)

Rethinking our Potential

- Intuitively, a cascading cut only occurs if we have a long chain of marked nodes.
- Those nodes were only marked because of previous ***decrease-key*** operations.
- ***Idea:*** Backcharge the work required to do the cascading cut to each preceding ***decrease-key*** that contributed to it.
- Specifically, change Φ as follows:

$$\mathbf{\Phi = \#trees + \#marked}$$

- ***Note:*** Since only ***decrease-key*** interacts with marked nodes, our amortized analysis of all previous operations is still the same.

The (New) Amortized Cost

- Using our new Φ, a ***decrease-key*** makes C cuts, it
  - Marks one new node (+1),
  - Unmarks C nodes ($-C$), and
  - Adds C trees to the root list ($+C$).
- Amortized cost is

$$
\begin{aligned}
& \Theta(C) + O(1) \cdot \Delta\Phi \\
&= \Theta(C) + O(1) \cdot (1 - C + C) \\
&= \Theta(C) + O(1) \cdot 1 \\
&= \Theta(C) + O(1) \\
&= \mathbf{\Theta(C)}
\end{aligned}
$$

- Hmmm... that didn't work.

The Trick

- Each ***decrease-key*** makes extra work for *two* future operations, since
  - future ***decrease-key***s have to do cascading cuts.
  - future ***extract-min***s now have more trees to coalesce, and
- We can make this explicit in our potential function:

$$\mathbf{\Phi = \#trees + 2 \cdot \#marked}$$

The (Final) Amortized Cost

- Using our new Φ, a ***decrease-key*** makes C cuts, it
  - Marks one new node (+2),
  - Unmarks C nodes ($-2C$), and
  - Adds C trees to the root list ($+C$).
- Amortized cost is

$$
\begin{aligned}
& \Theta(C) + O(1) \cdot \Delta\Phi \\
= {} & \Theta(C) + O(1) \cdot (2 - 2C + C) \\
= {} & \Theta(C) + O(1) \cdot (2 - C) \\
= {} & \Theta(C) - O(C) + O(1) \\
= {} & \mathbf{\Theta(1)}
\end{aligned}
$$

- We now have amortized O(1) ***decrease-key***!

The Story So Far

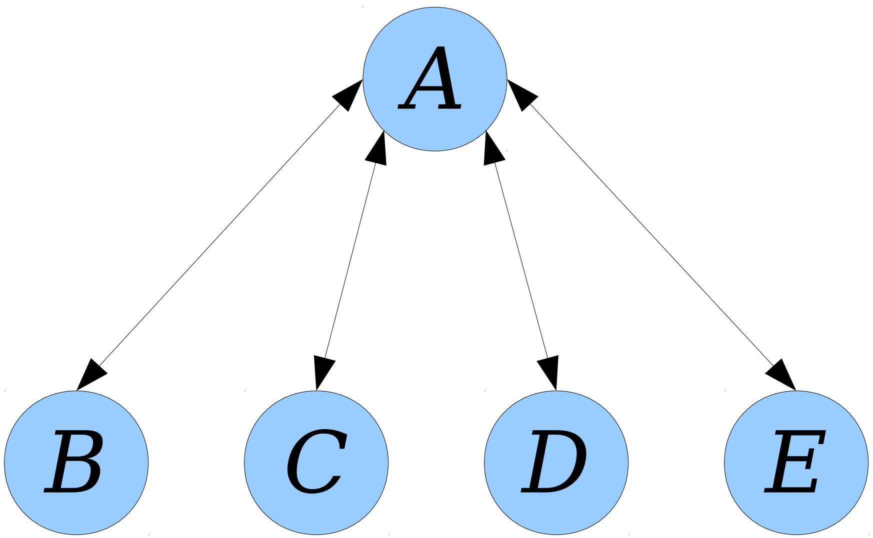
- The Fibonacci heap has the following amortized time bounds:
  - ***enqueue***: O(1)
  - ***find-min***: O(1)
  - ***meld***: O(1)
  - ***decrease-key***: O(1) amortized
  - ***extract-min***: O(log n) amortized
- This is about as good as it gets!

The Catch: Representation Issues

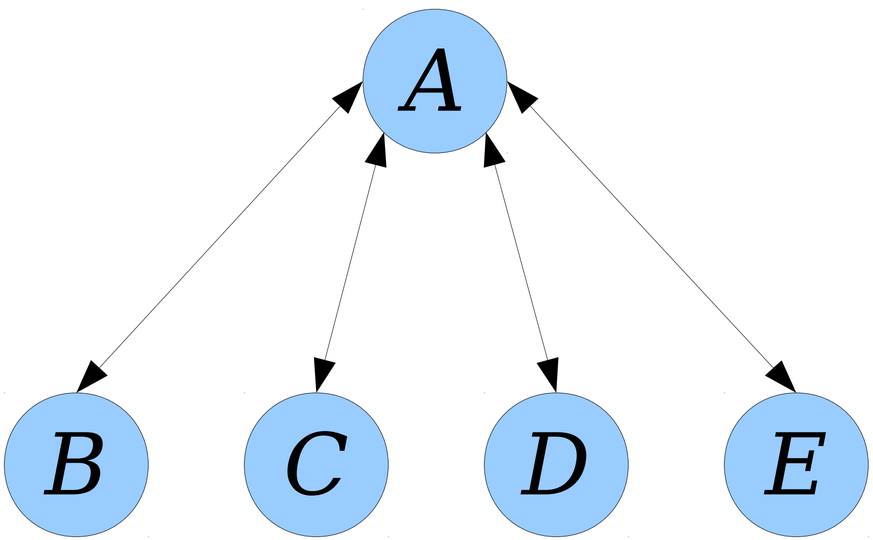
Representing Trees

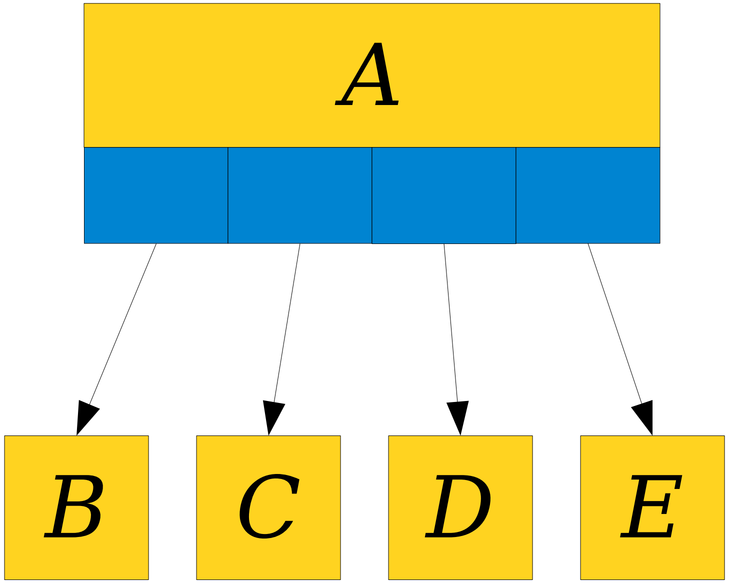
- The trees in a Fibonacci heap must be able to do the following:
  - During a merge: Add one tree as a child of the root of another tree.
  - During a cut: Cut a node from its parent in time O(1).
- ***Claim:*** This is trickier than it looks.

Representing Trees

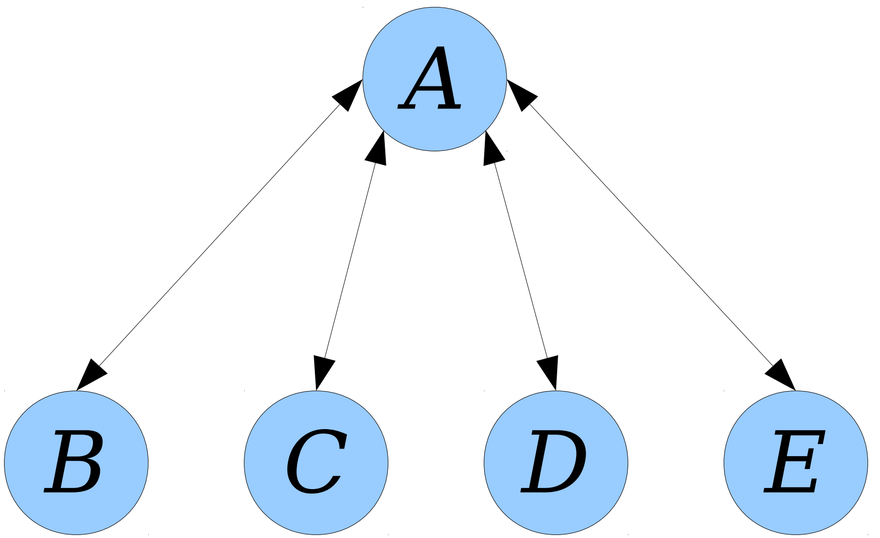


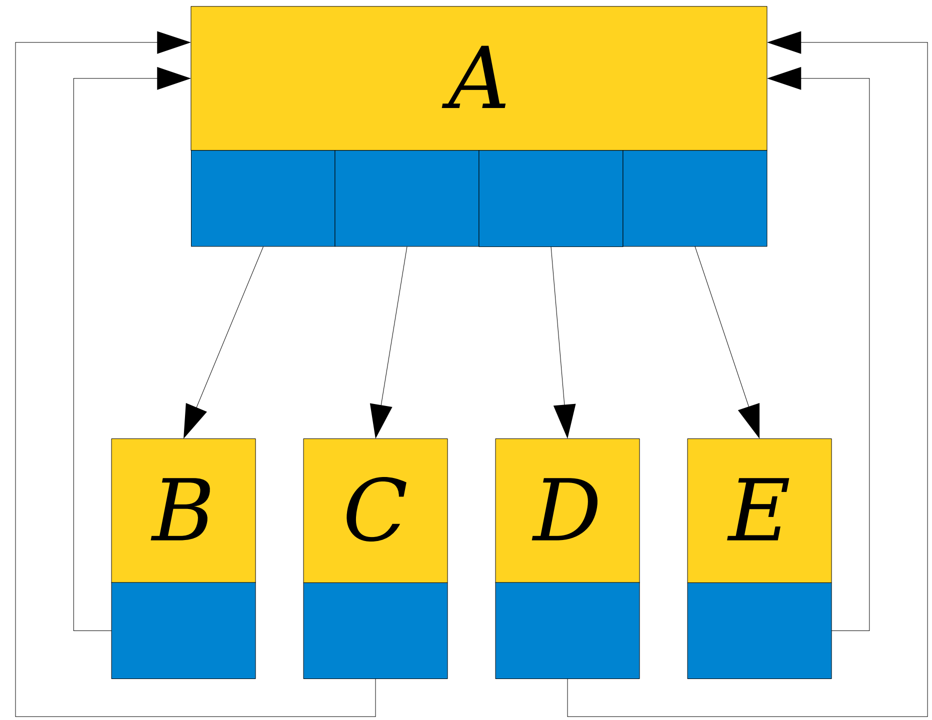
Representing Trees



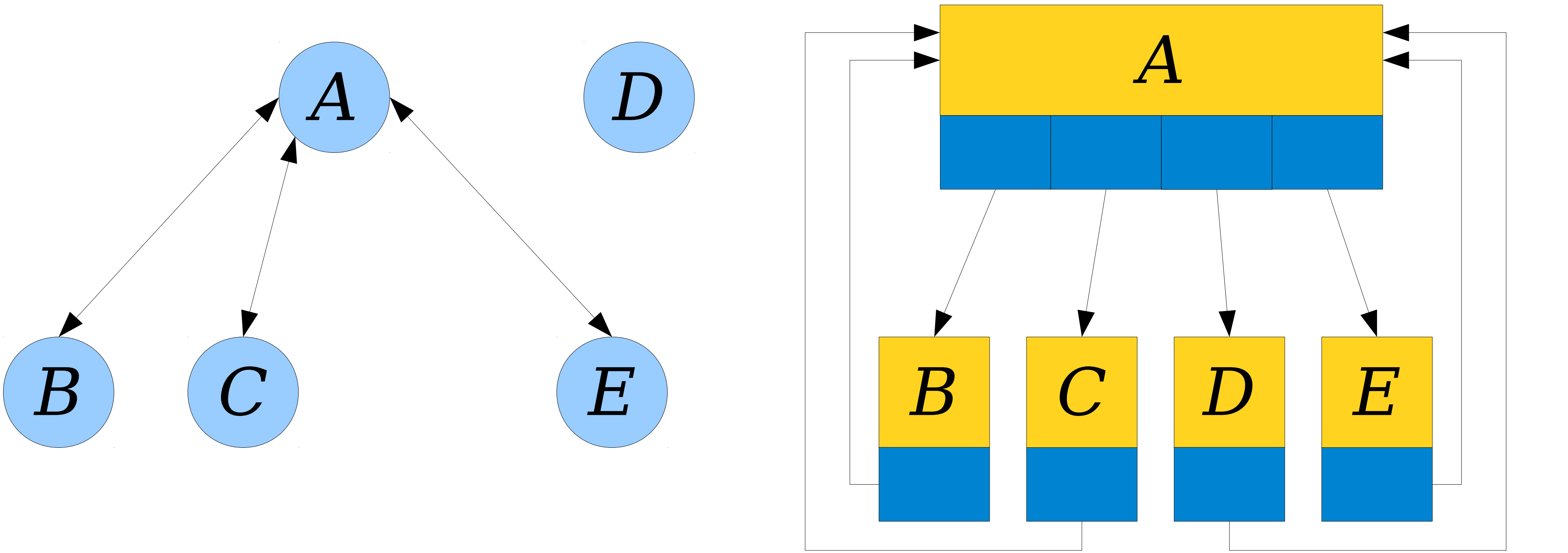


Representing Trees

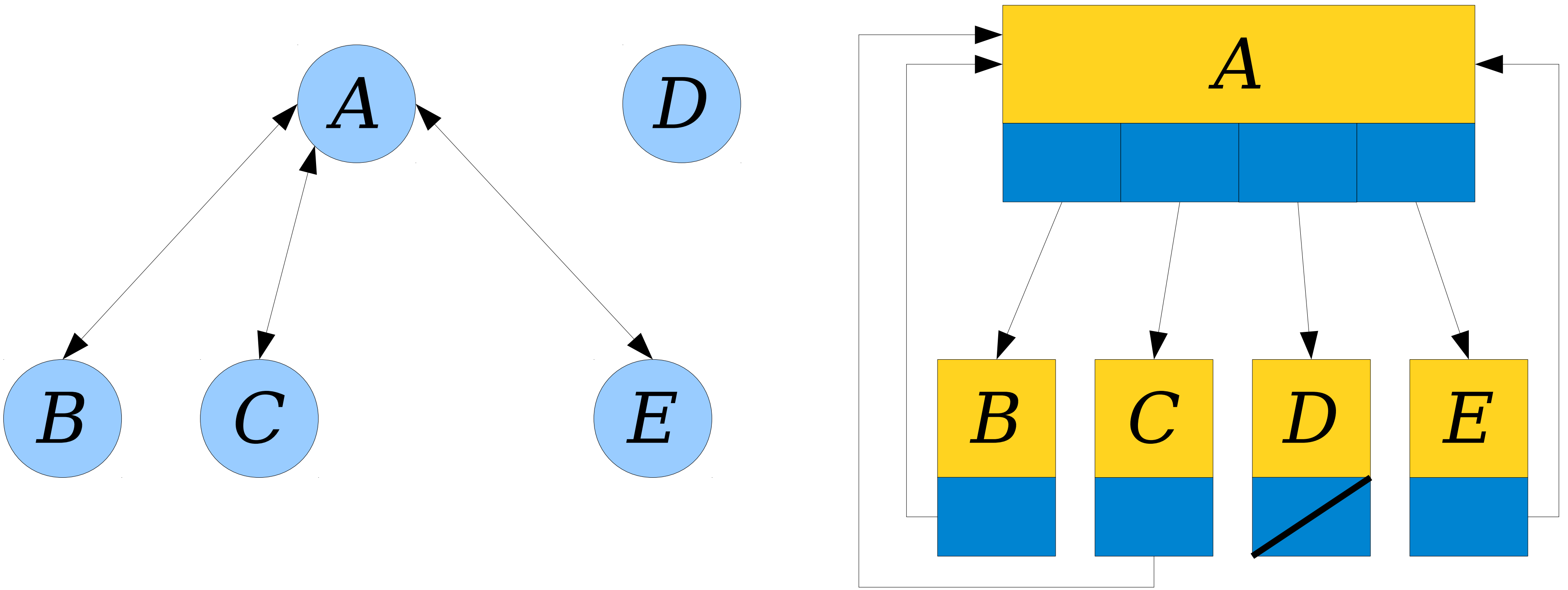




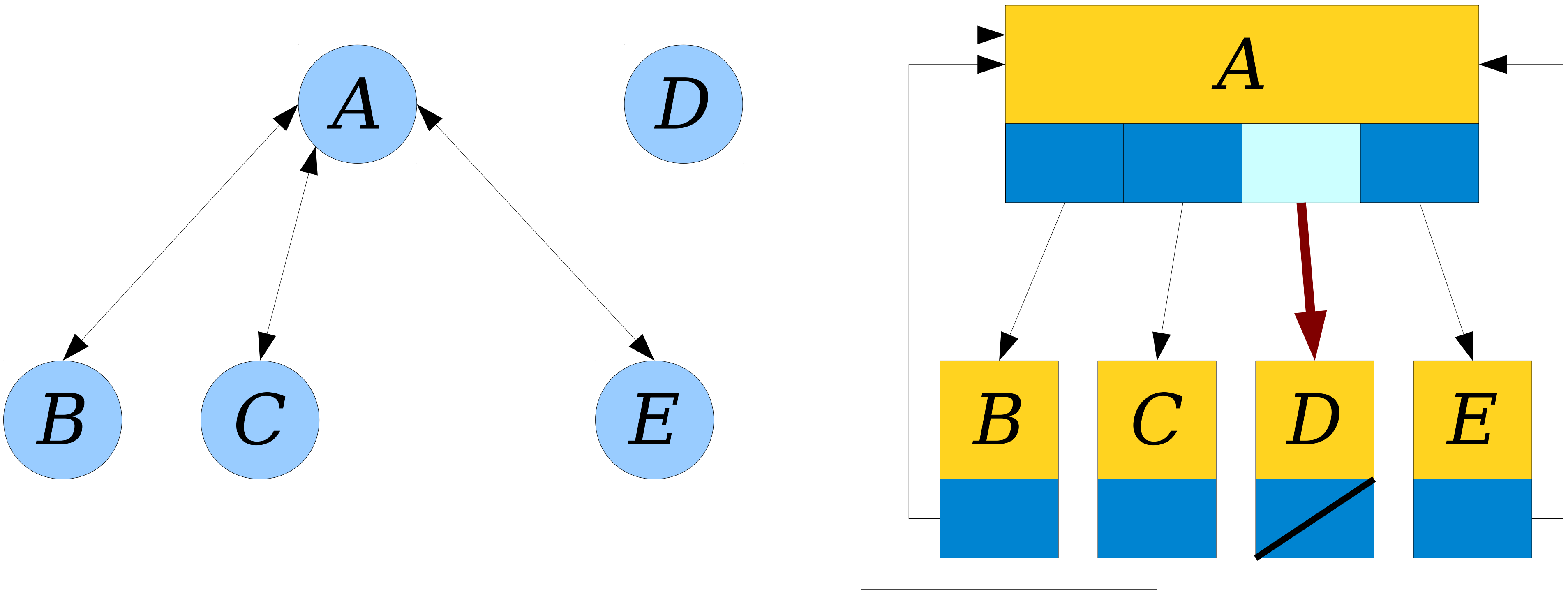
Representing Trees



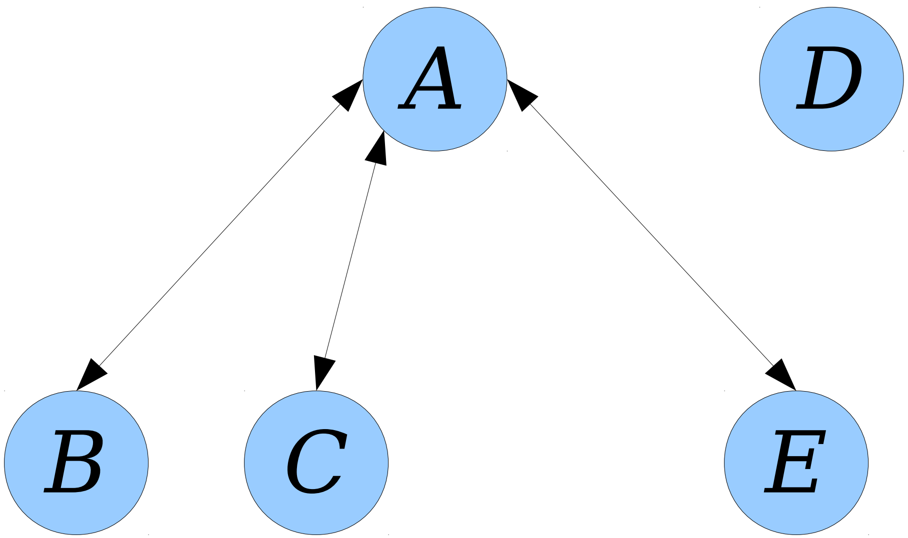
Representing Trees

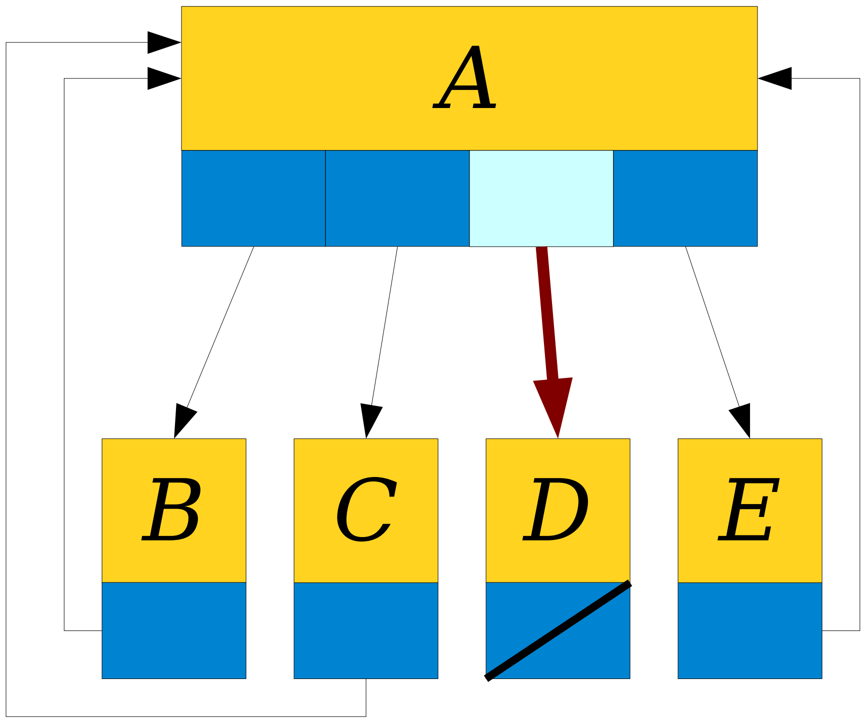


Representing Trees



Representing Trees





Finding this pointer might take time Θ(log n)!

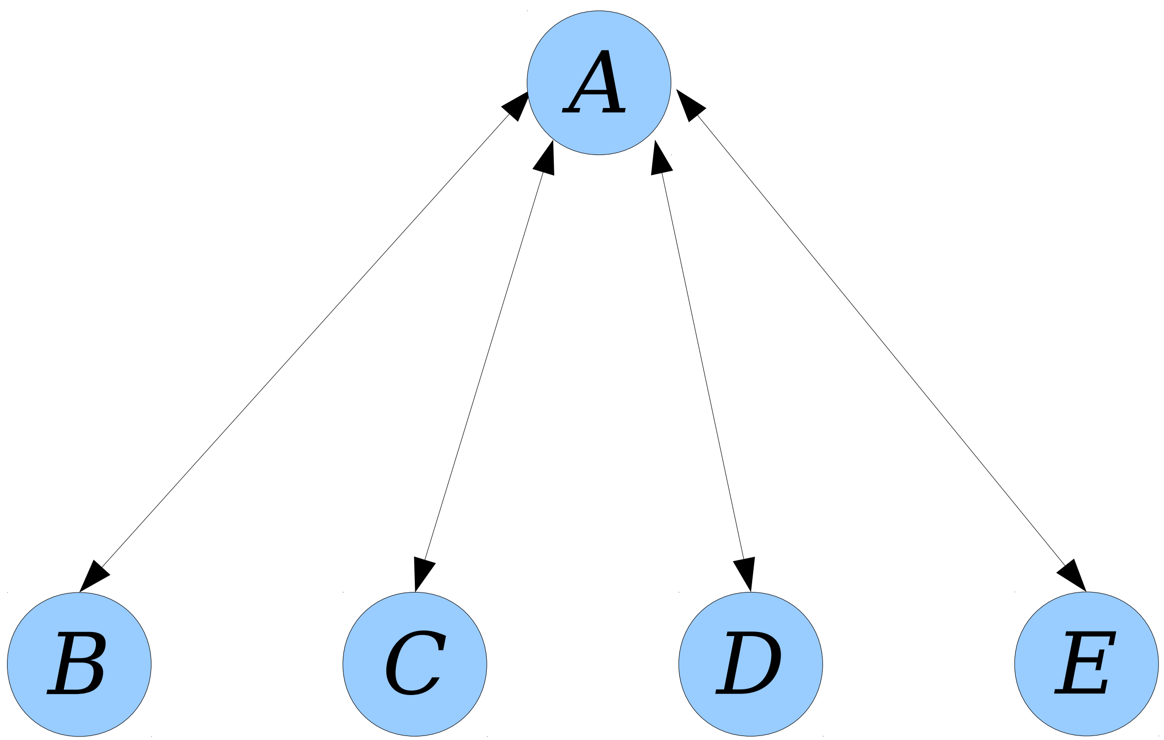
The Solution

The Solution

This is going to be weird.

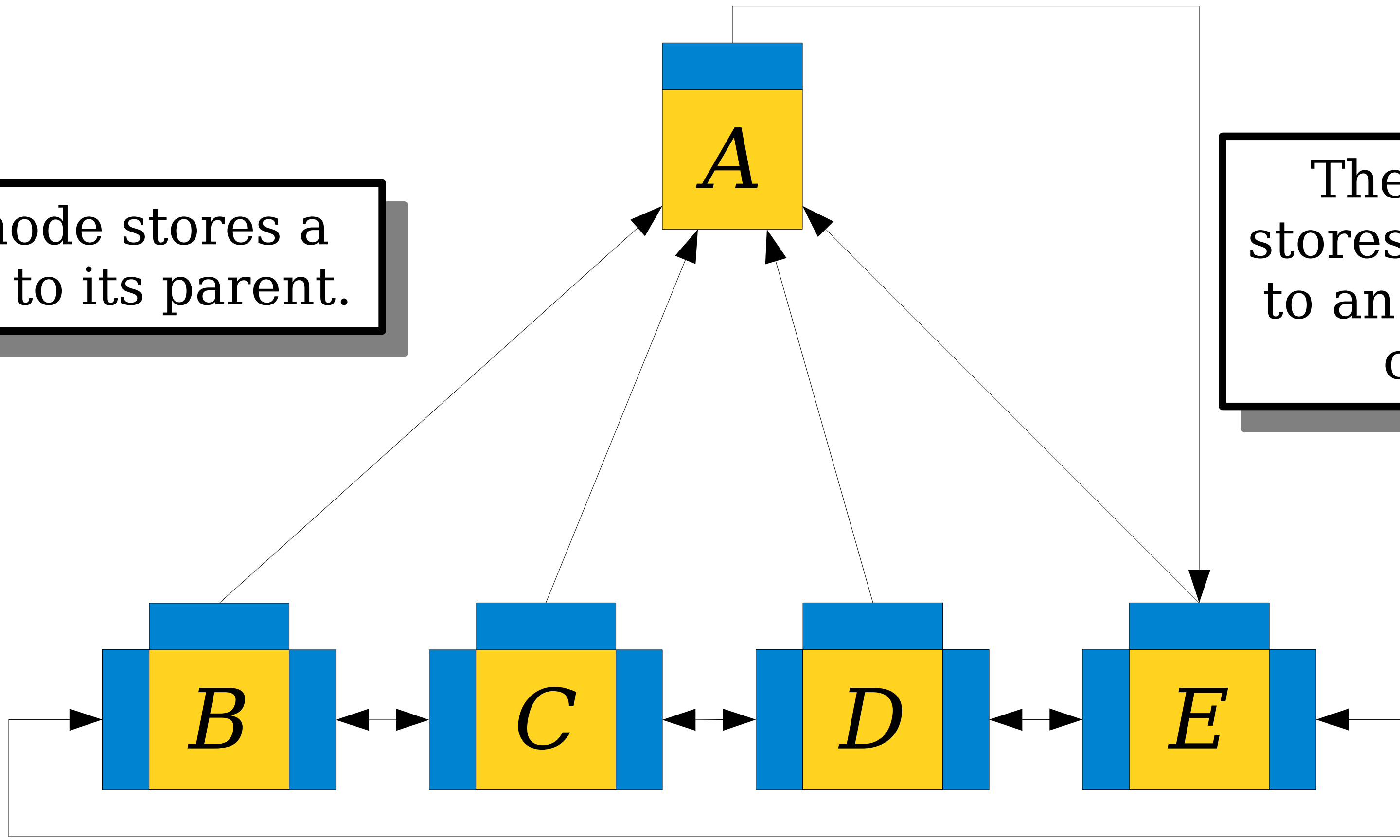
Sorry.

The Solution



The Solution

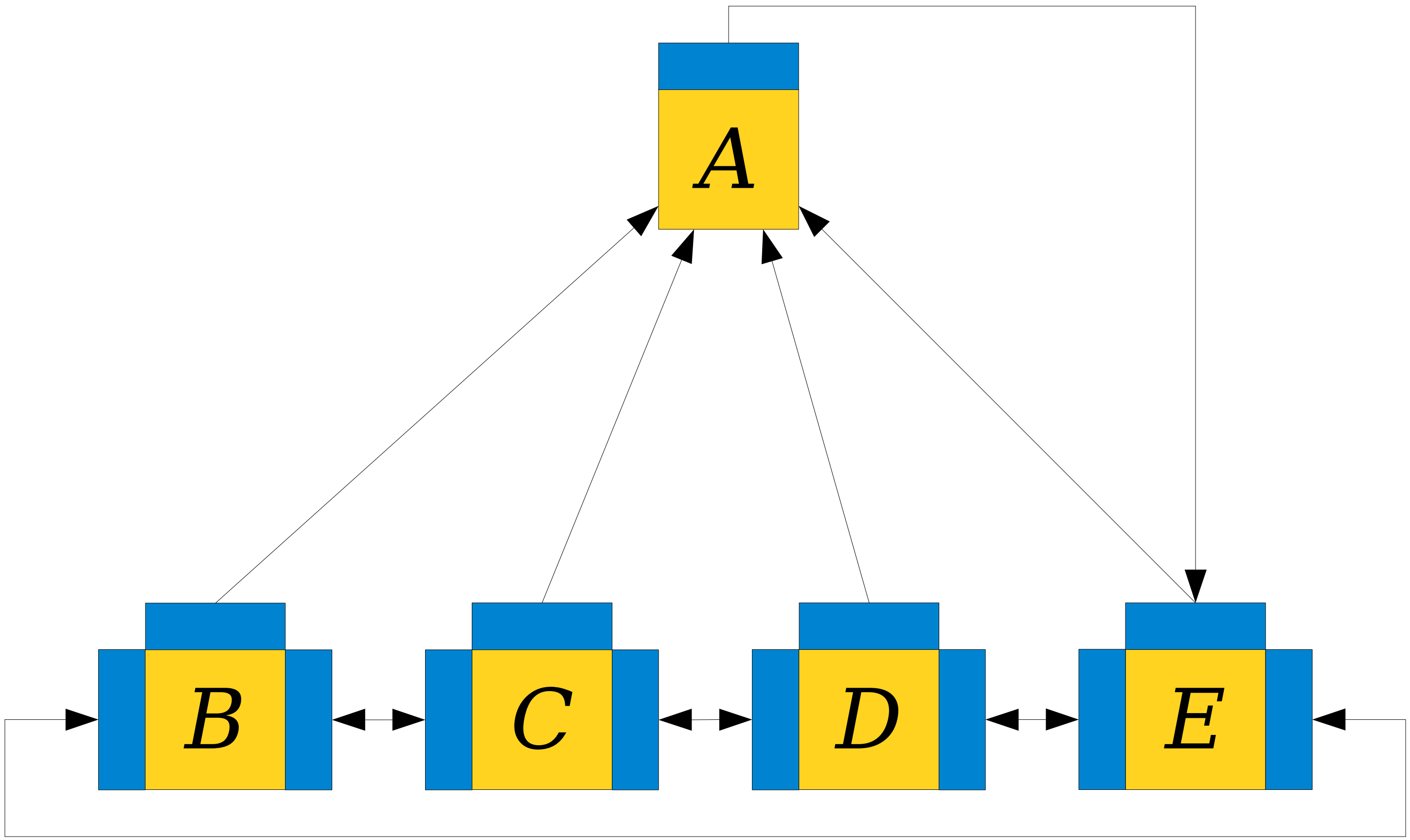
Each node stores a pointer to its parent.



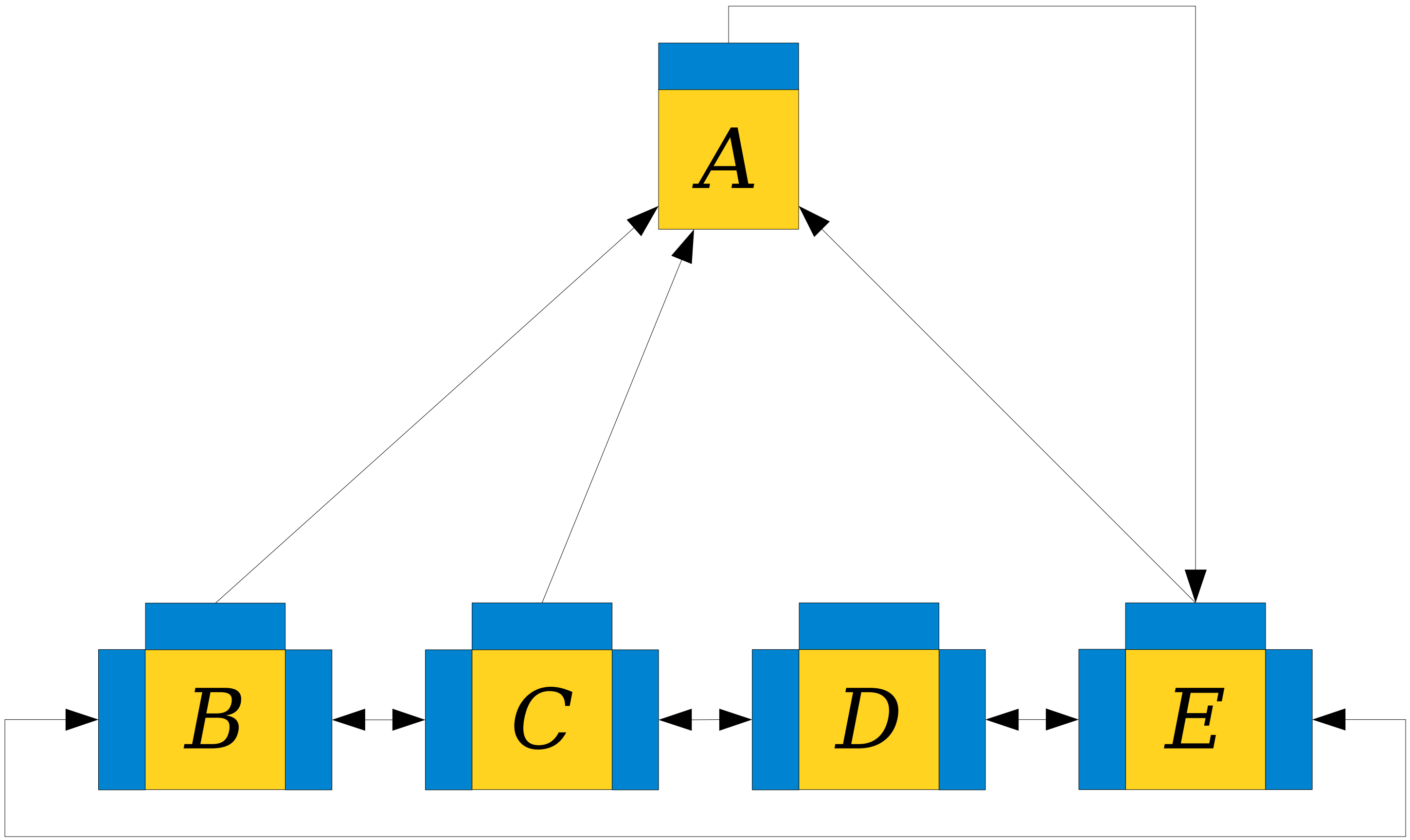
The parent stores a pointer to an arbitrary child.

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

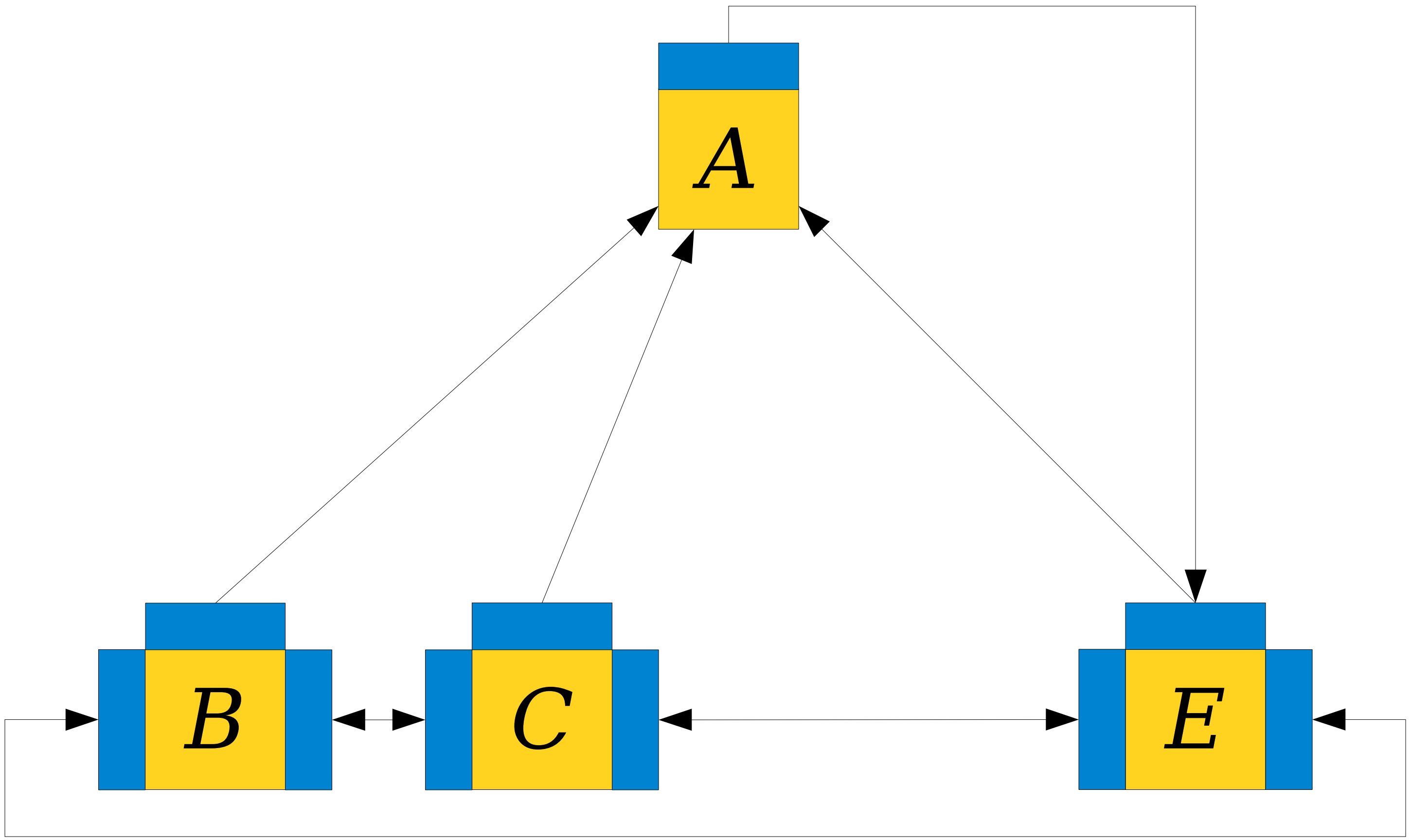
The Solution



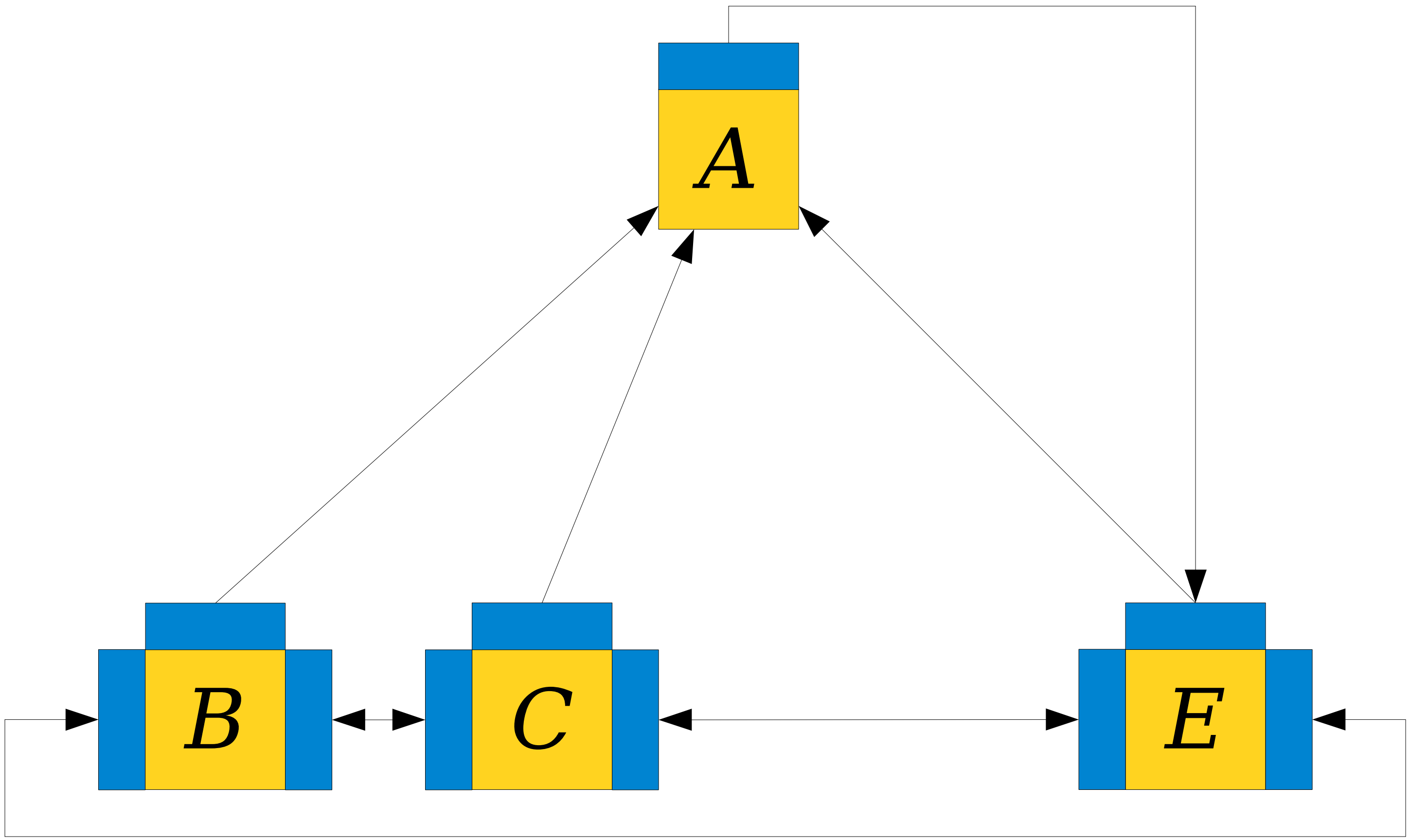
The Solution



The Solution

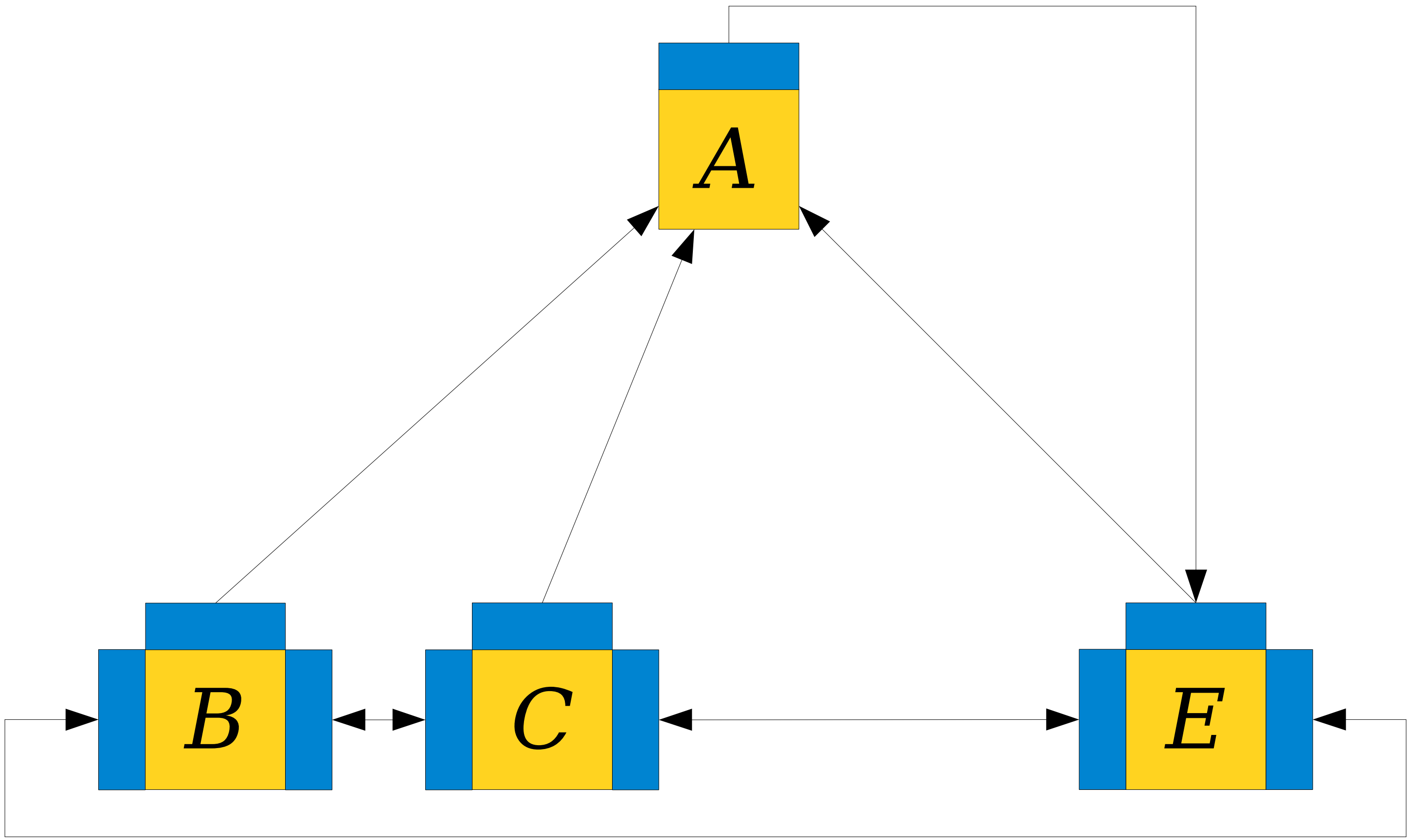


The Solution

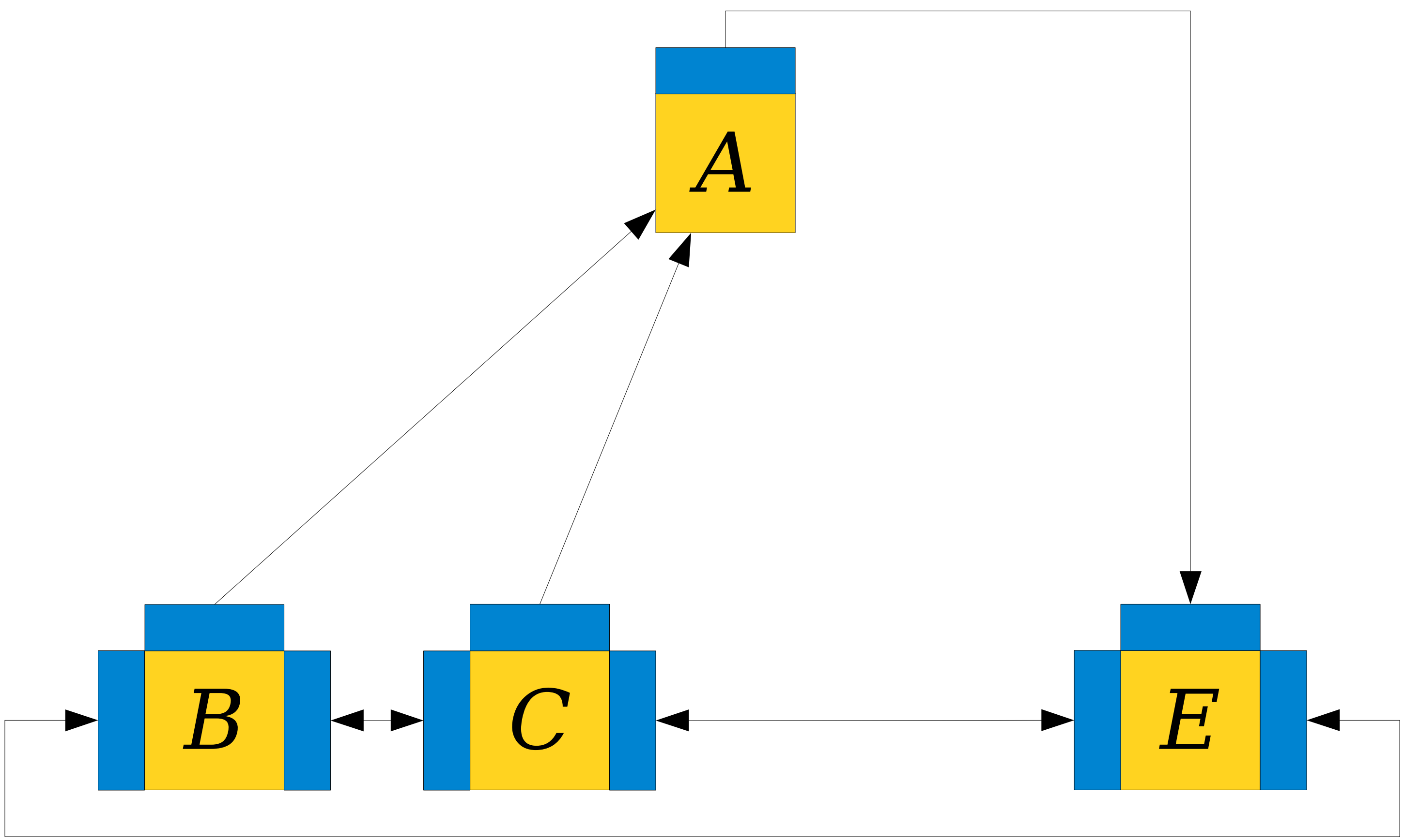


To cut a node from its parent, if it isn't the representative child, just splice it out of its linked list.

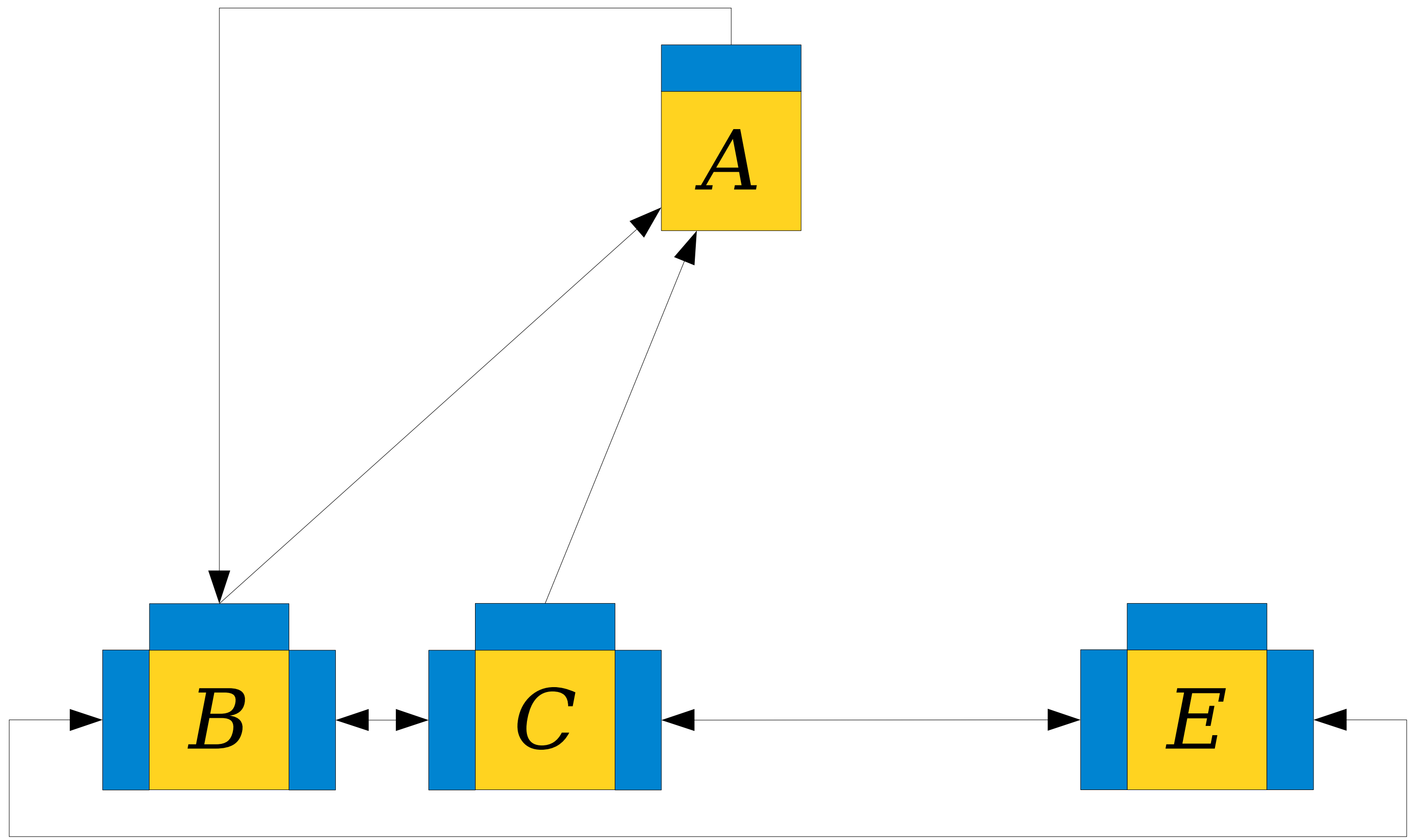
The Solution



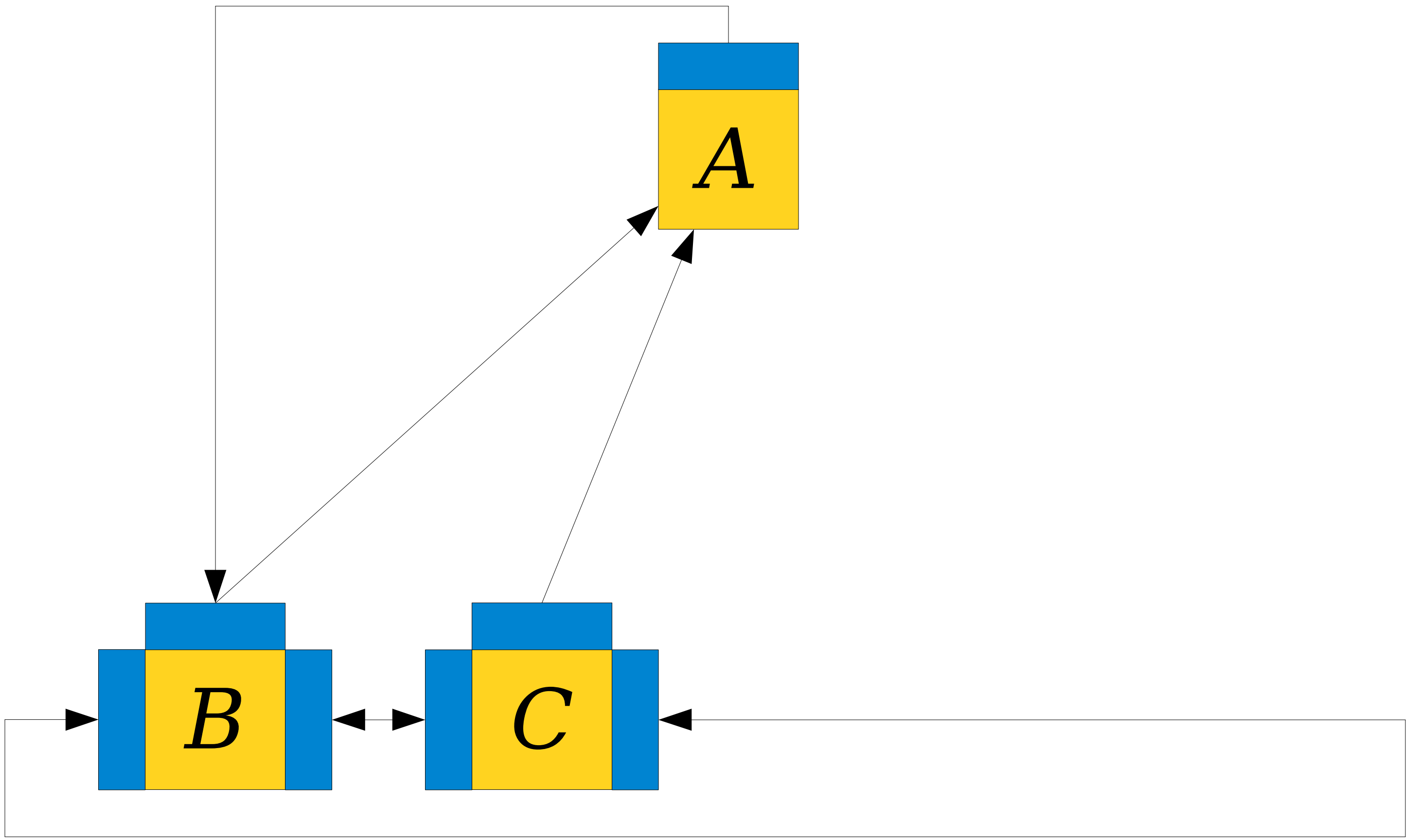
The Solution



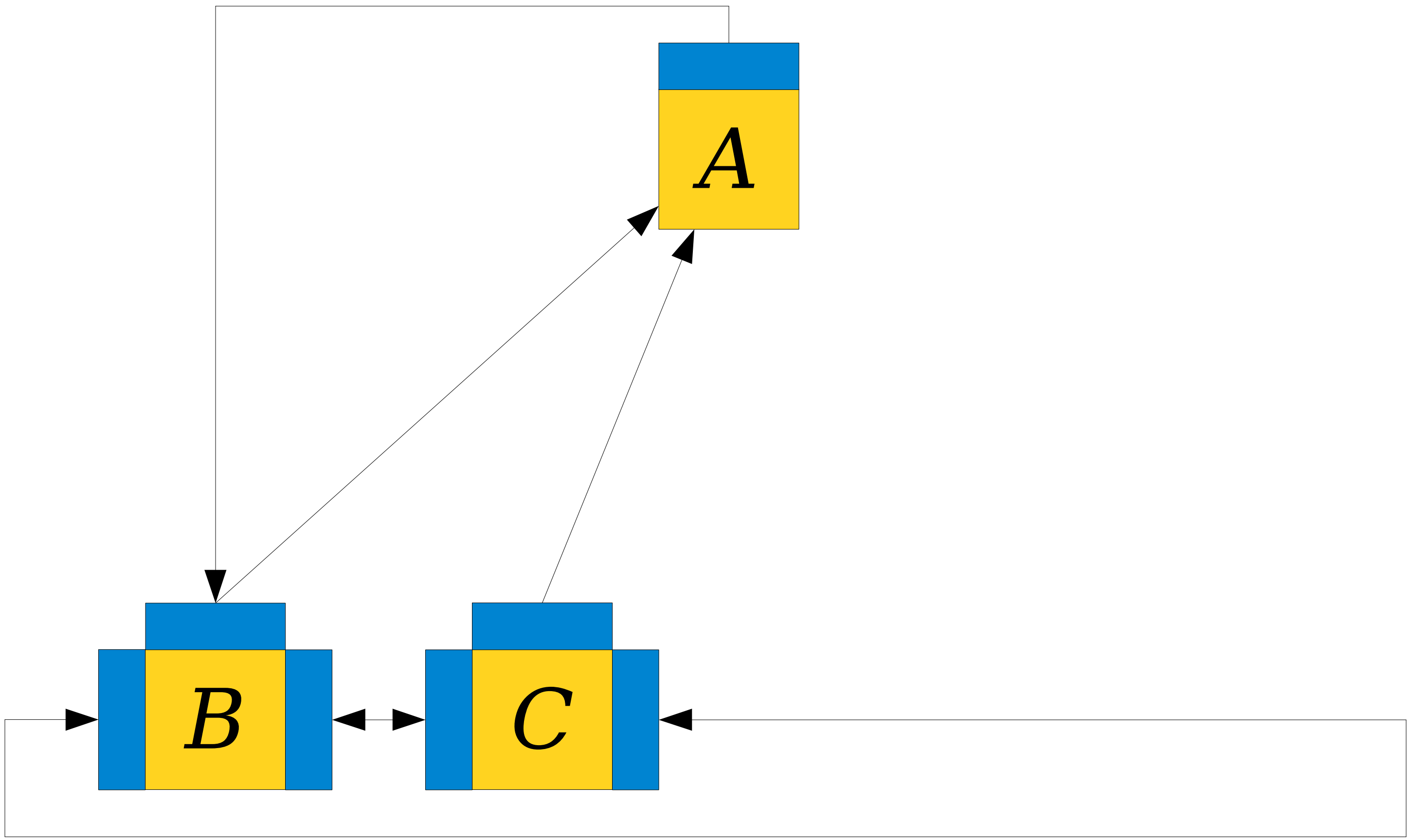
The Solution



The Solution



The Solution



If it is the representative, change the parent's representative child to be one of the node's siblings.

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.
- Awful, but the following possible are now possible in time O(1):
  - Cut a node from its parent.
  - Add another child node to a node.
- This is the main reason Fibonacci heaps are so complex.

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, Fibonacci heaps are slower than other heaps with worse asymptotic performance.
- 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 2011, Haeupler, Sen, and Tajran developed the ***rank-pairing heap***, which matches the amortized time bounds of Fibonacci heaps but with significantly fewer structural guarantees.
- In 2012, Brodal et al. invented the ***strict Fibonacci heap*** was developed. It has the same time bounds as a Fibonacci heap, but in a *worst-case* rather than *amortized* sense.
- All of these would make for great final project topics!

Summary

- ***decrease-key*** is a useful operation in many graph algorithms.
- Implement ***decrease-key*** by cutting a node from its parent and hoisting it up to the root list.
- To make sure trees of high order have lots of nodes, add a marking scheme and cut nodes that lose two or more children.
- Represent the data structure using Awful Linked Lists.
- Can prove that the number of nodes in each tree grows exponentially with φ by looking at maximally-damaged trees.

Next Time

- ***Splay Trees***
  - Amortized-efficient balanced trees.
- ***Static Optimality***
  - Is there a single best BST for a set of data?
- ***Dynamic Optimality***
  - Is there a single best BST for a set of data if that BST can change over time?