Amortized Analysis

Outline for Today

- ***Amortized Analysis***
  - Analyzing data structures over the long term.
- ***Cartesian Trees Revisited***
  - Why could we construct them in time O(n)?
- ***The Two-Stack Queue***
  - A simple and elegant queue implementation.
- ***2-3-4 Trees***
  - A better analysis of 2-3-4 tree insertions and deletions.

Two Worlds

- Data structures have different requirements in different contexts.
  - In real-time applications, each operation on a given data structure needs to be fast and snappy.
  - In long data processing pipelines, we care more about the total time used than we do the cost of any one operation.
- In many cases, we can get better performance in the long-run than we can on a per-operation basis.
  - Good intuition: “economy of scale.”

Key Idea: Design data structures that trade *per-operation efficiency* for *overall efficiency*.

Claims We'd Like to Make

- "The total runtime of this algorithm is $O(n \log n)$, even though there are $\Theta(n)$ steps and each step, in the worst case, takes time $\Theta(n)$."
- "If you perform m operations on this data structure, although each operation could take time $\Theta(n)$, the average cost of an operation is $O(1)$."
- "Operations on this data structure can take up to $\Theta(n^2)$ time to complete, but if you pretend that each operation takes time $O(\log n)$, you'll never overestimate the total amount of work done."

What We Need

- First, we need a ***mathematical framework*** for analyzing algorithms and data structures when the costs of individual operations vary.
- Next, we need a set of ***design techniques*** for building data structures that nicely fit into this framework.
- Today is mostly about the first of these ideas. We'll explore design techniques all next week.

Amortized Analysis

The Goal

- Suppose we have a data structure and perform a series of ***m*** operations op_1, op_2, …, op_m.
  - These operations might be the same operation, or they might be different.
- Let ***t(op$_k$)*** denote the time required to perform operation op_k.
- ***Goal:*** Bound this expression, which represents the total runtime across all operations:

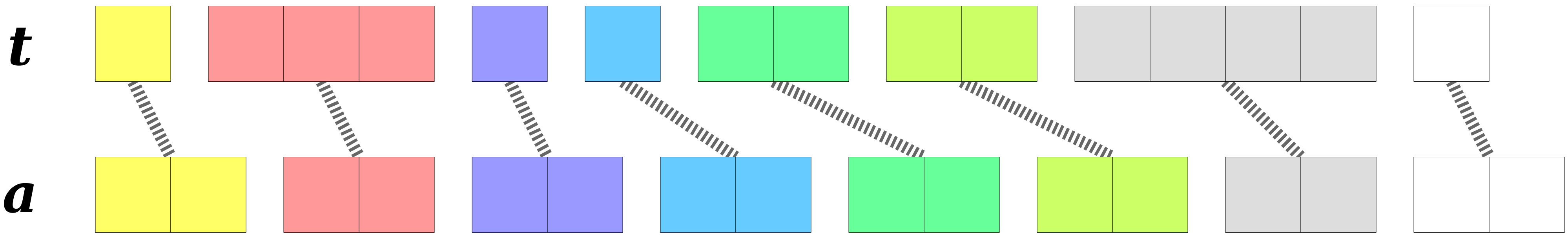
$$T = \sum_{i=1}^{m} t(op_i)$$

Amortized Analysis

- An ***amortized analysis*** is a different way of bounding the runtime of a sequence of operations.
- ***Idea:*** Assign to each operation op_i a new cost $a(op_i)$, called the ***amortized cost***, such that the following is true for any sequence of m operations:

$$\sum_{i=1}^{m} t(op_i) \leq \sum_{i=1}^{m} a(op_i)$$

Question: How do you choose amortized costs?



Where We're Going

- There are three standard techniques for assigning amortized costs to operations:
  - The ***aggregate method*** directly assigns each operation its average cost.
  - The ***banker's method*** places credits on the data structure, which are redeemable for units of work.
  - The ***potential method*** assigns a potential function to the data structure, which can be charged to pay for future work or released to pay for recent work.

Where We're Going

There are three standard techniques for assigning amortized costs to operations:

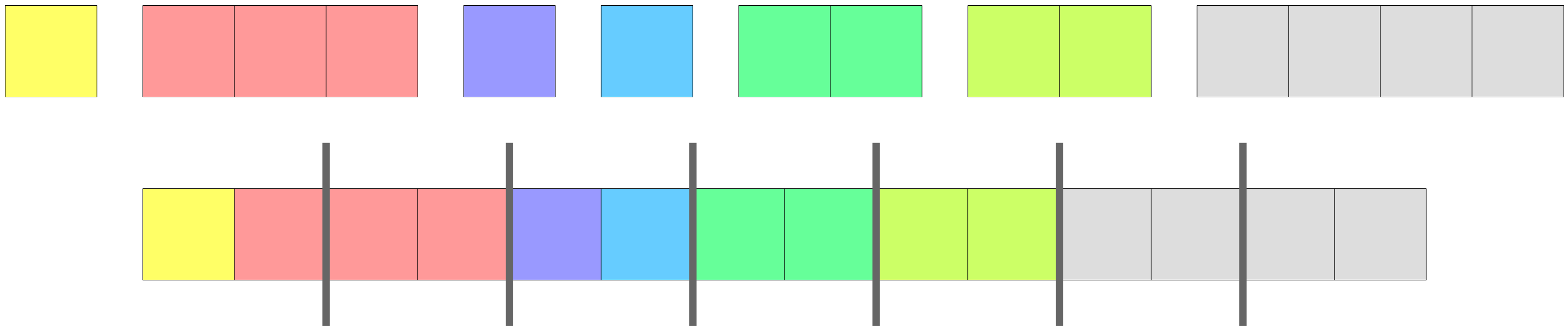
- The ***aggregate method*** directly assigns each operation its average cost.
- The ***banker's method*** places credits on the data structure, which are redeemable for units of work.
- The ***potential method*** assigns a potential function to the data structure, which can be charged to pay for future work or released to pay for recent work.

The Aggregate Method

- In the ***aggregate method***, we assign each operation a cost of

$$a(op_i) = T^*(m) / m$$

  where $T^*(m)$ is the maximum amount of work done by any series of m operations.
- We essentially pretend that each operation's runtime is the the average cost of all operations performed.

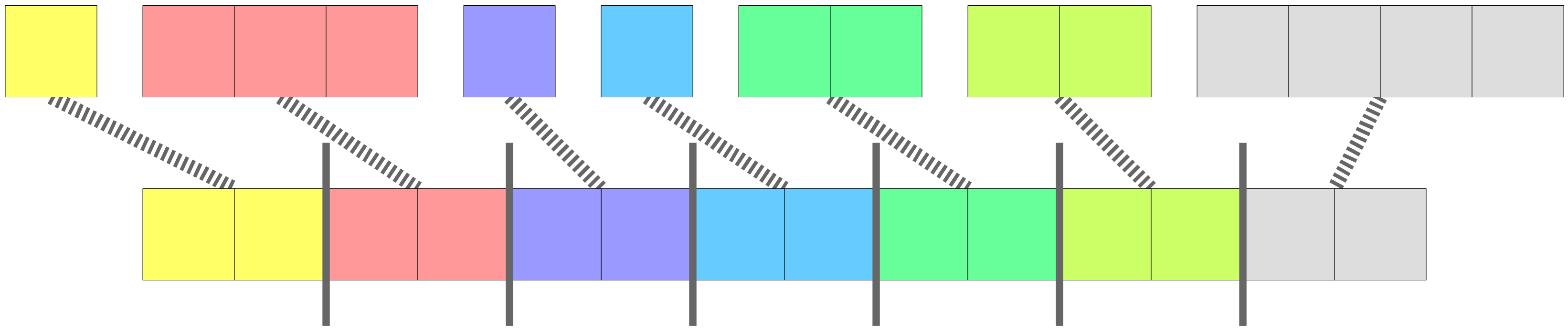


The Aggregate Method

- In the ***aggregate method***, we assign each operation a cost of

$$\boldsymbol{a(op_i) = T^*(m) / m}$$

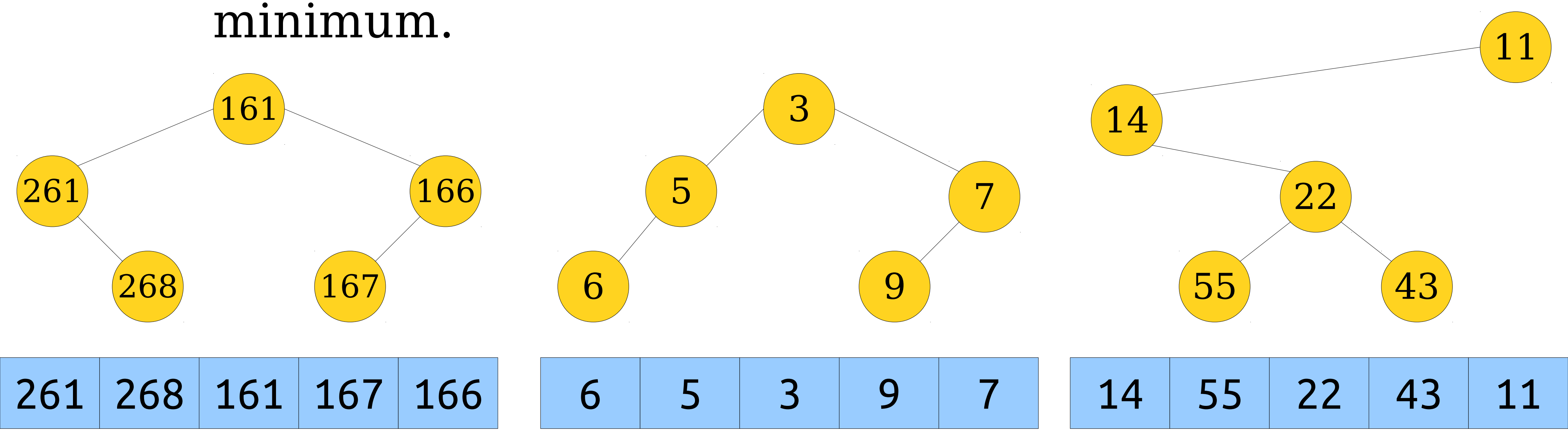
  where $T^*(m)$ is the maximum amount of work done by any series of m operations.
- We essentially pretend that each operation's runtime is the the average cost of all operations performed.



Cartesian Trees Revisited

Cartesian Trees

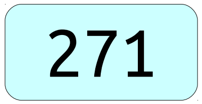
- A ***Cartesian tree*** is a binary tree derived from an array and defined as follows:
  - The empty array has an empty Cartesian tree.
  - For a nonempty array, the root stores the minimum value. Its left and right children are Cartesian trees for the subarrays to the left and right of the minimum.



Constructing Cartesian Trees

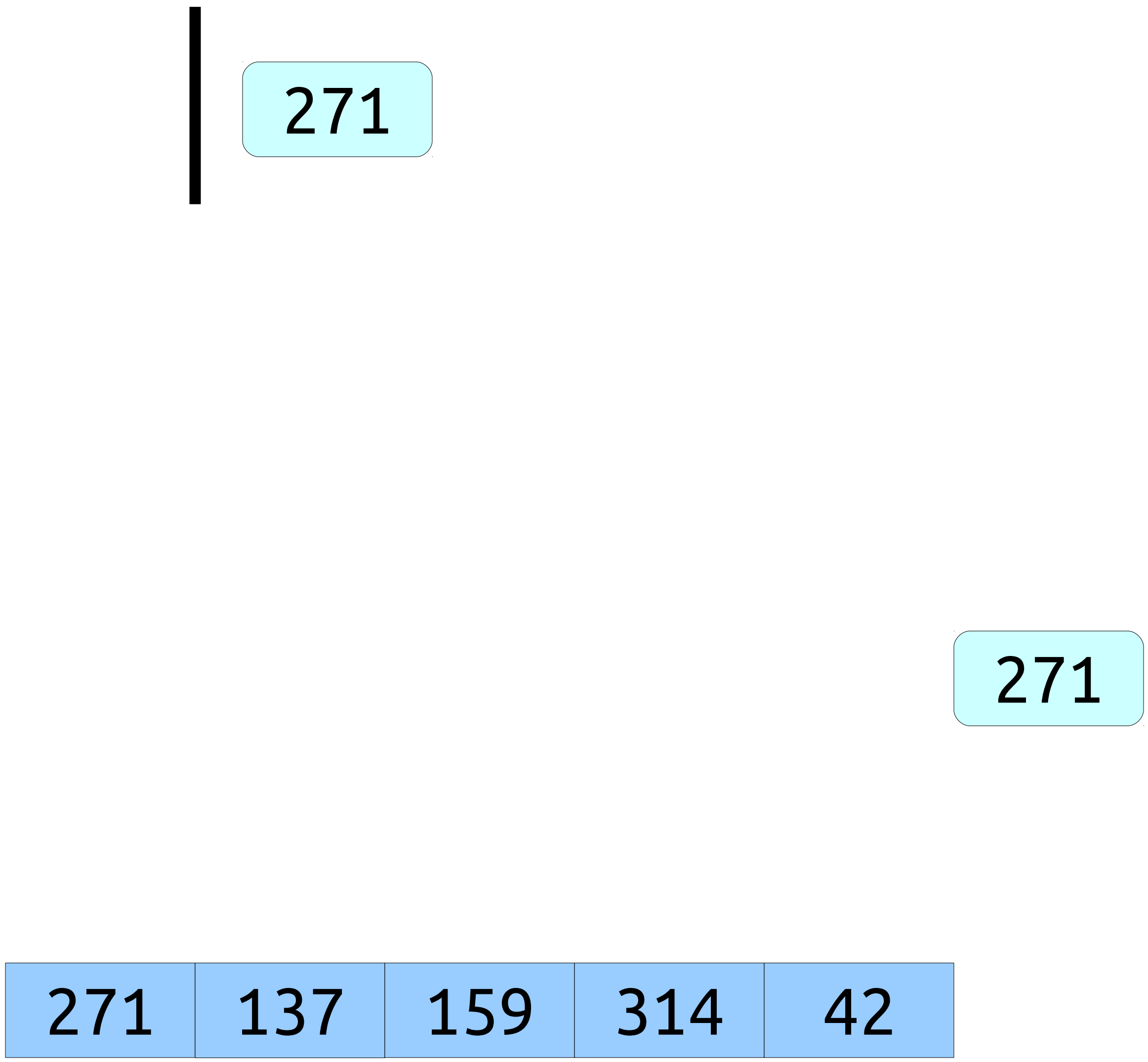
271	137	159	314	42

Constructing Cartesian Trees

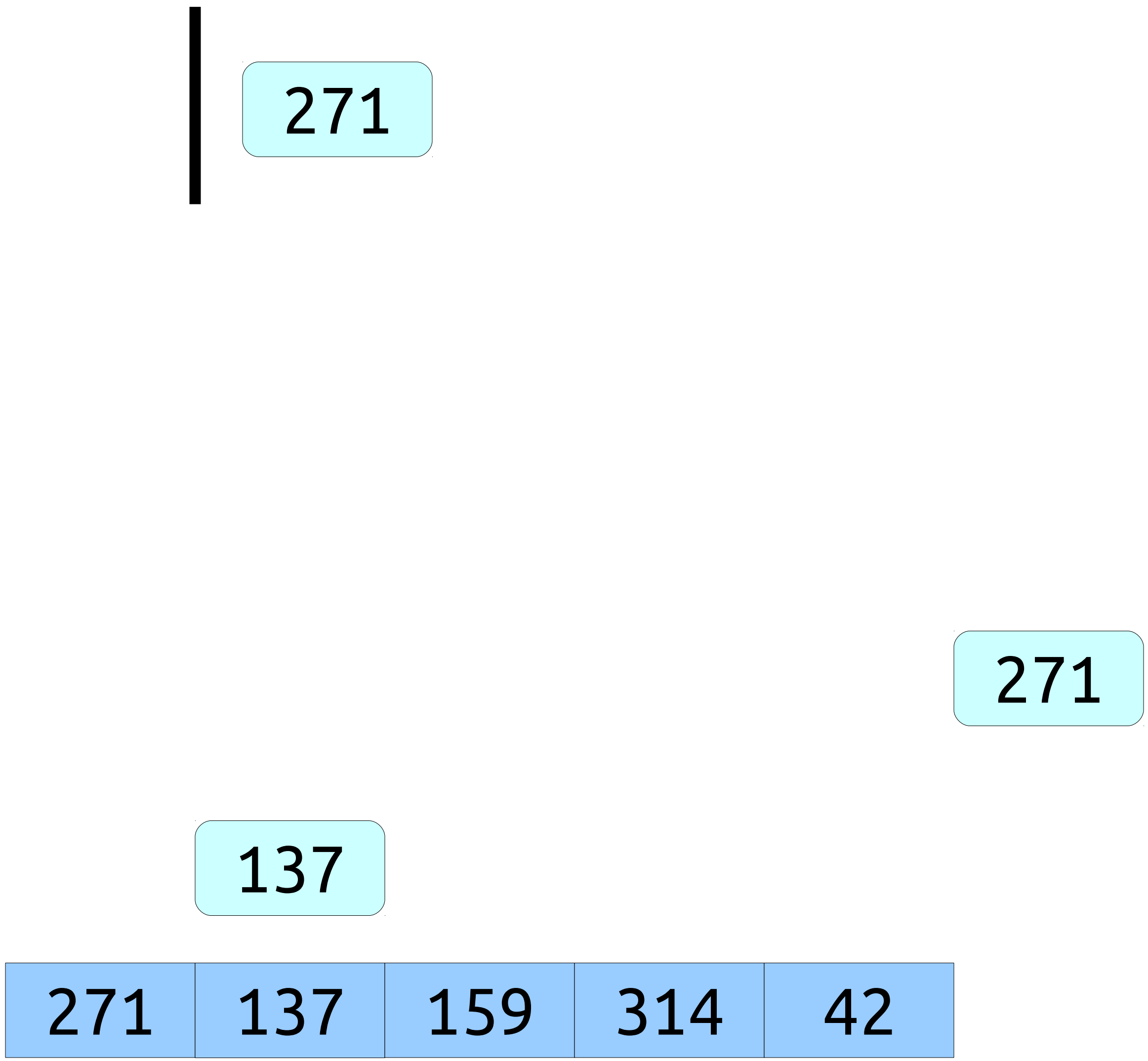


271	137	159	314	42

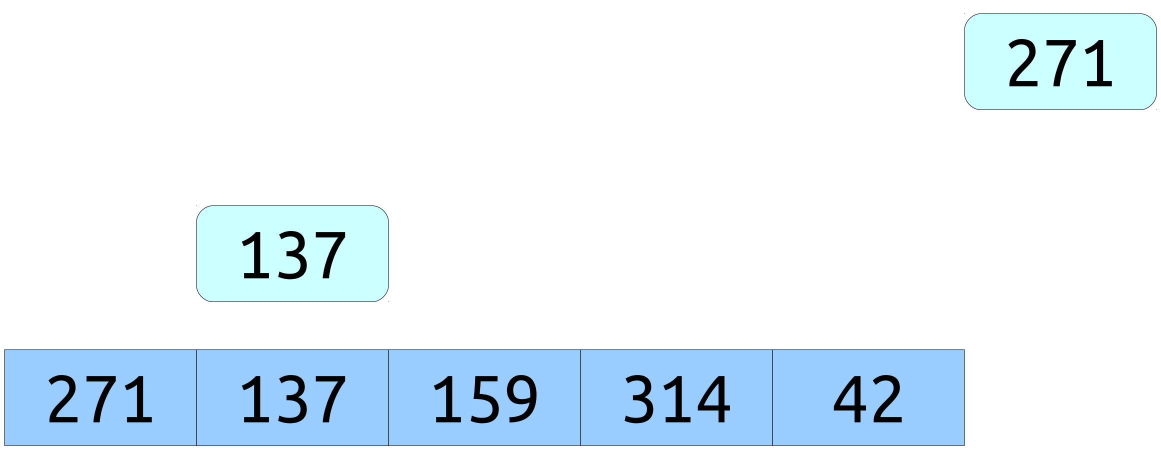
Constructing Cartesian Trees



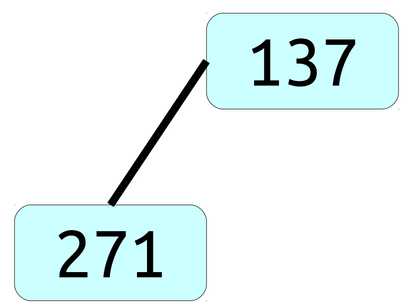
Constructing Cartesian Trees



Constructing Cartesian Trees

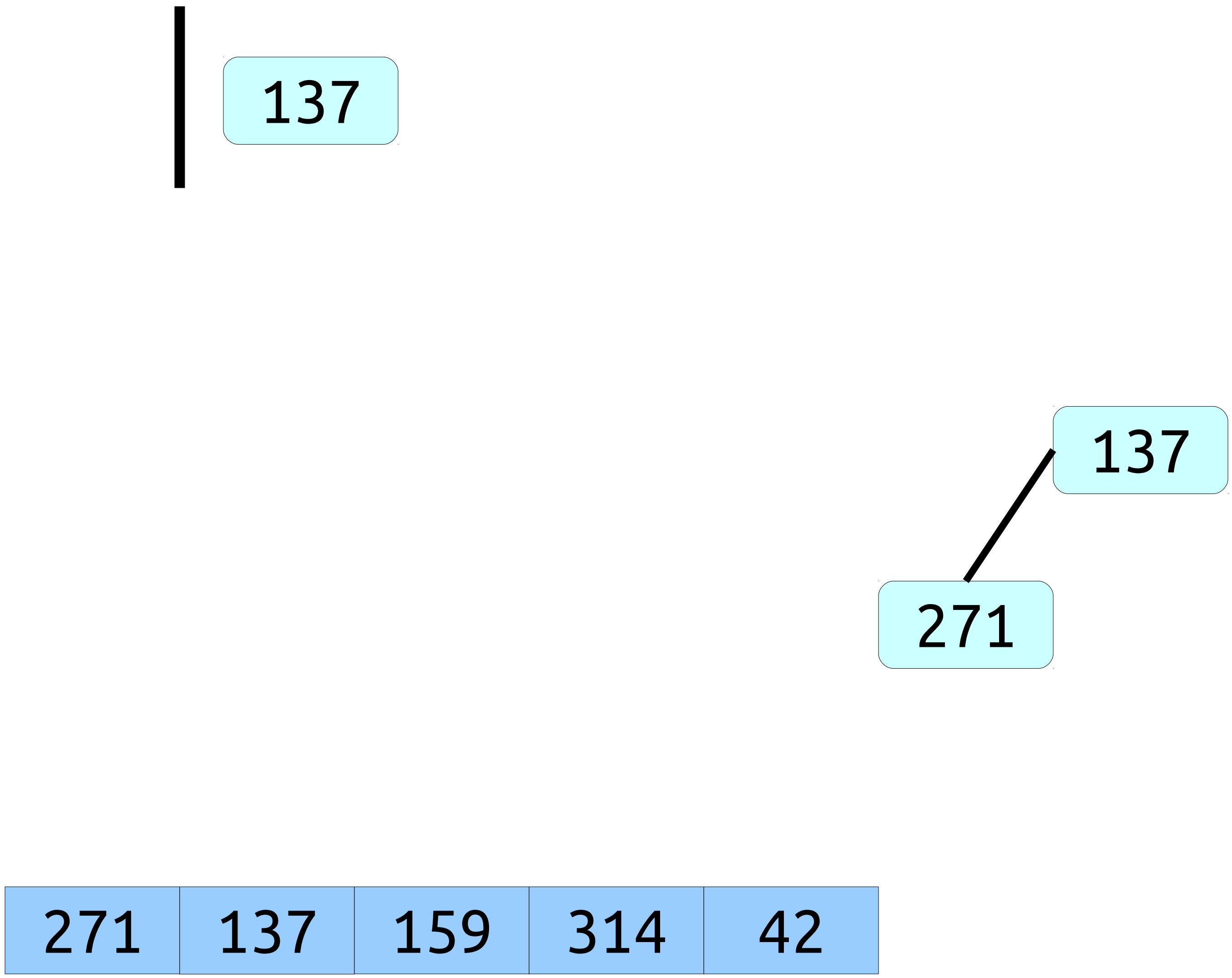


Constructing Cartesian Trees

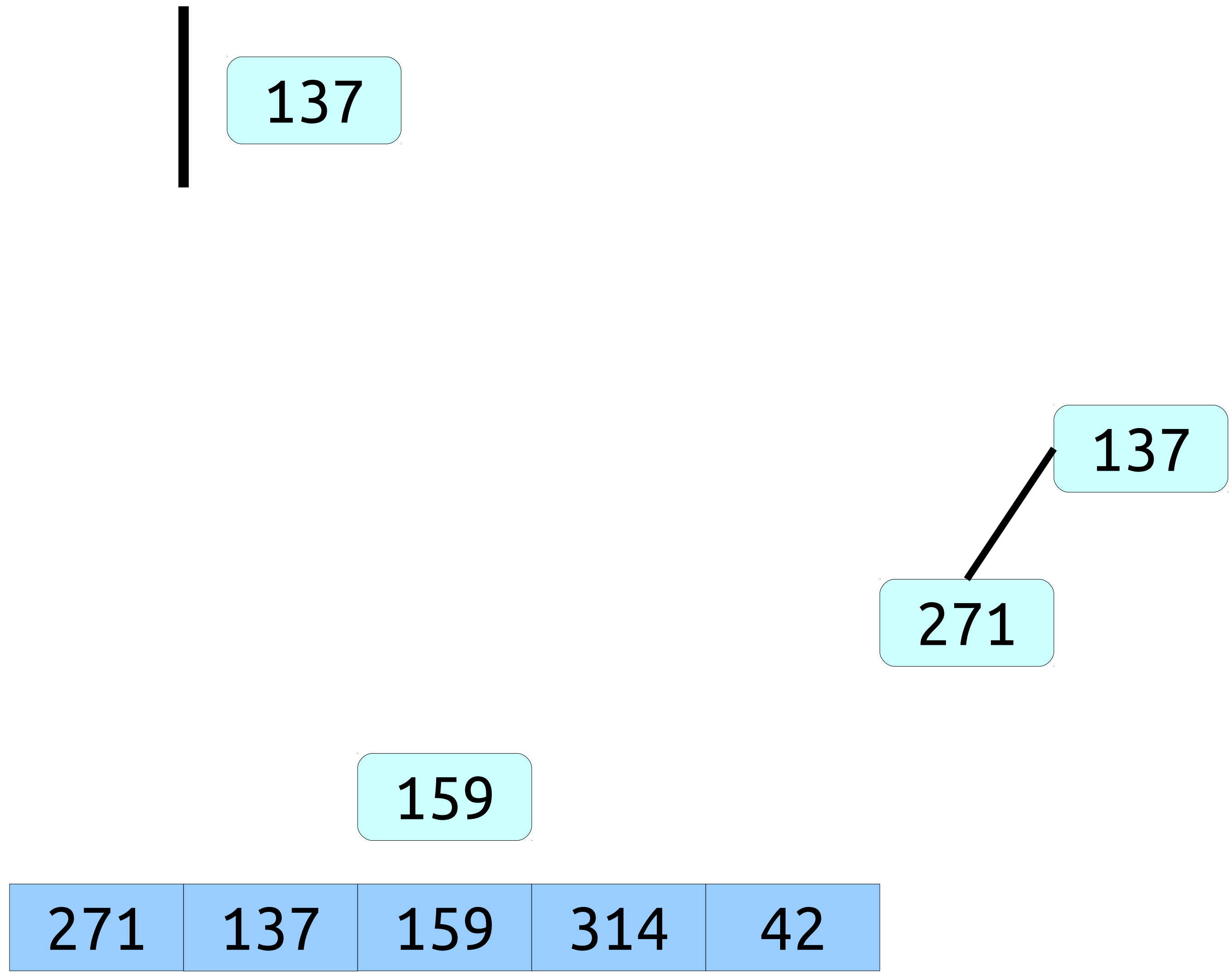


271	137	159	314	42

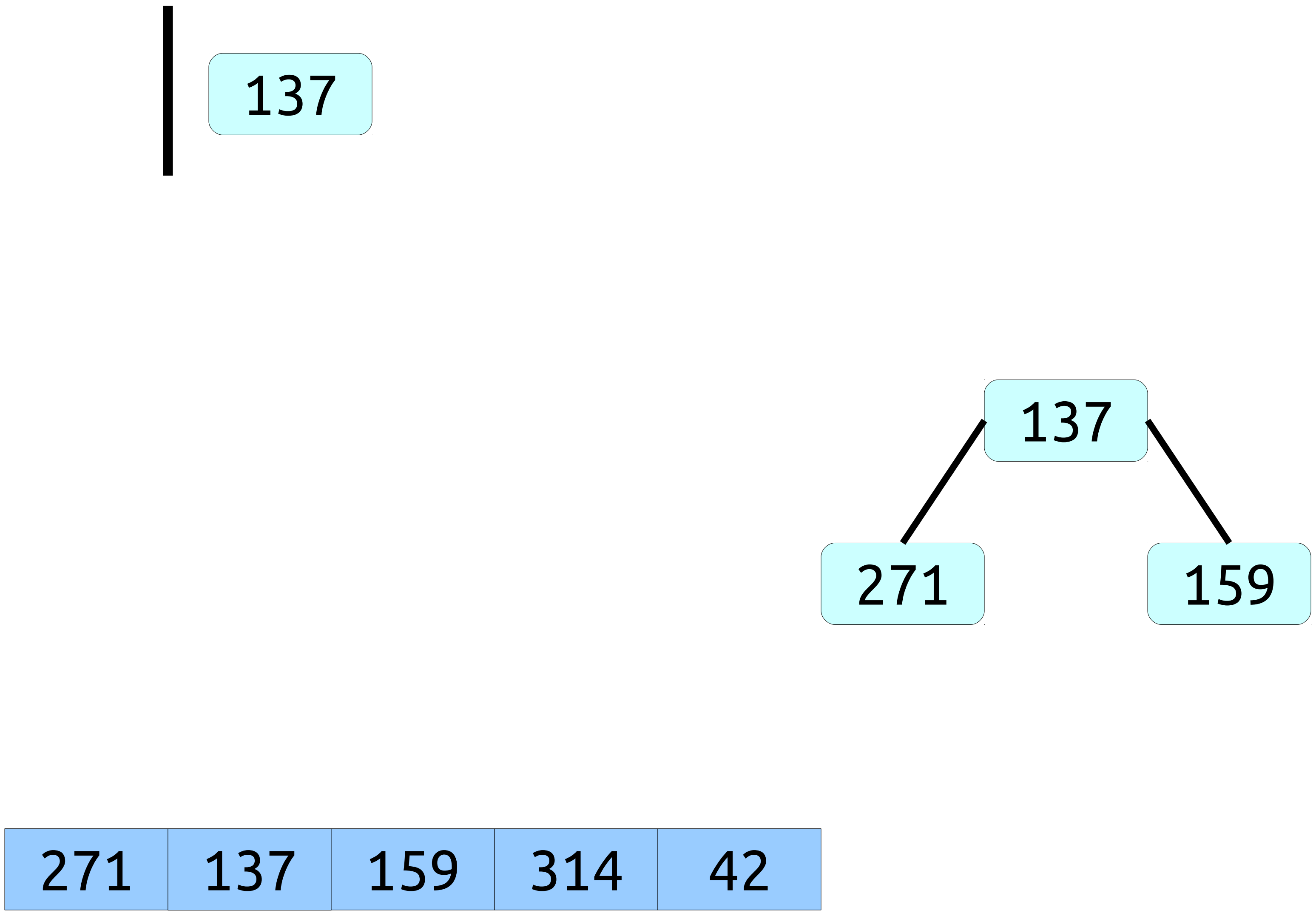
Constructing Cartesian Trees



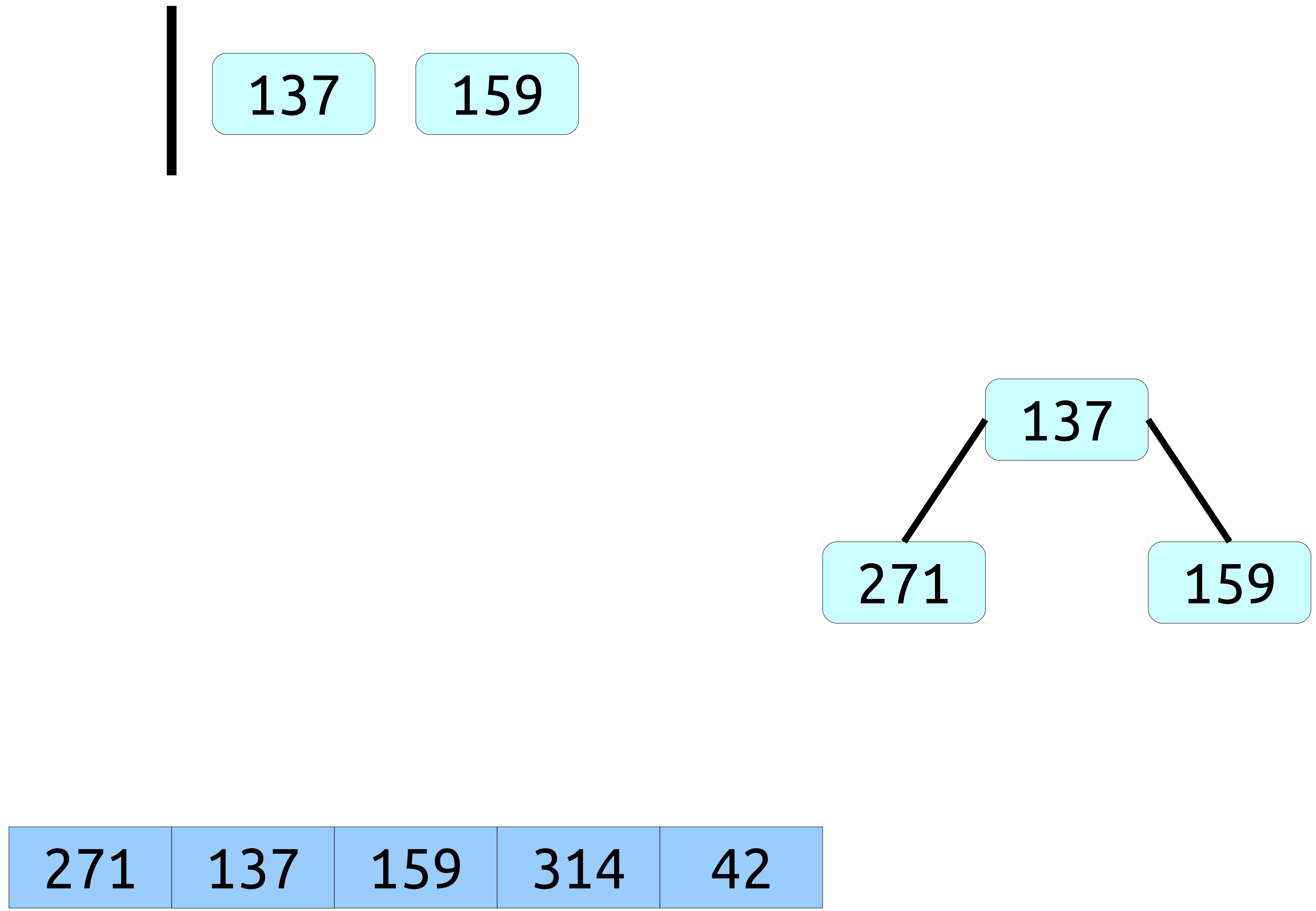
Constructing Cartesian Trees



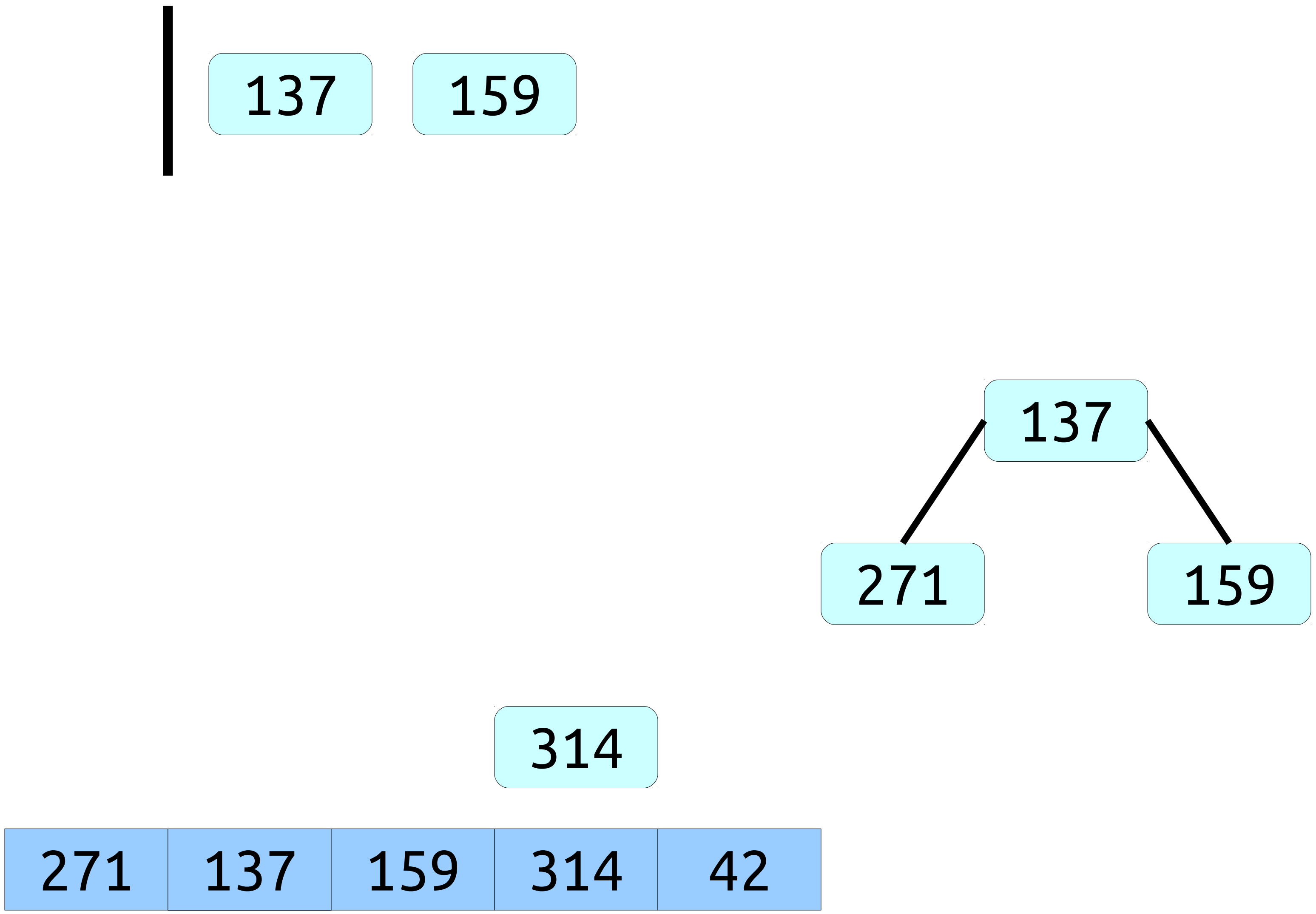
Constructing Cartesian Trees



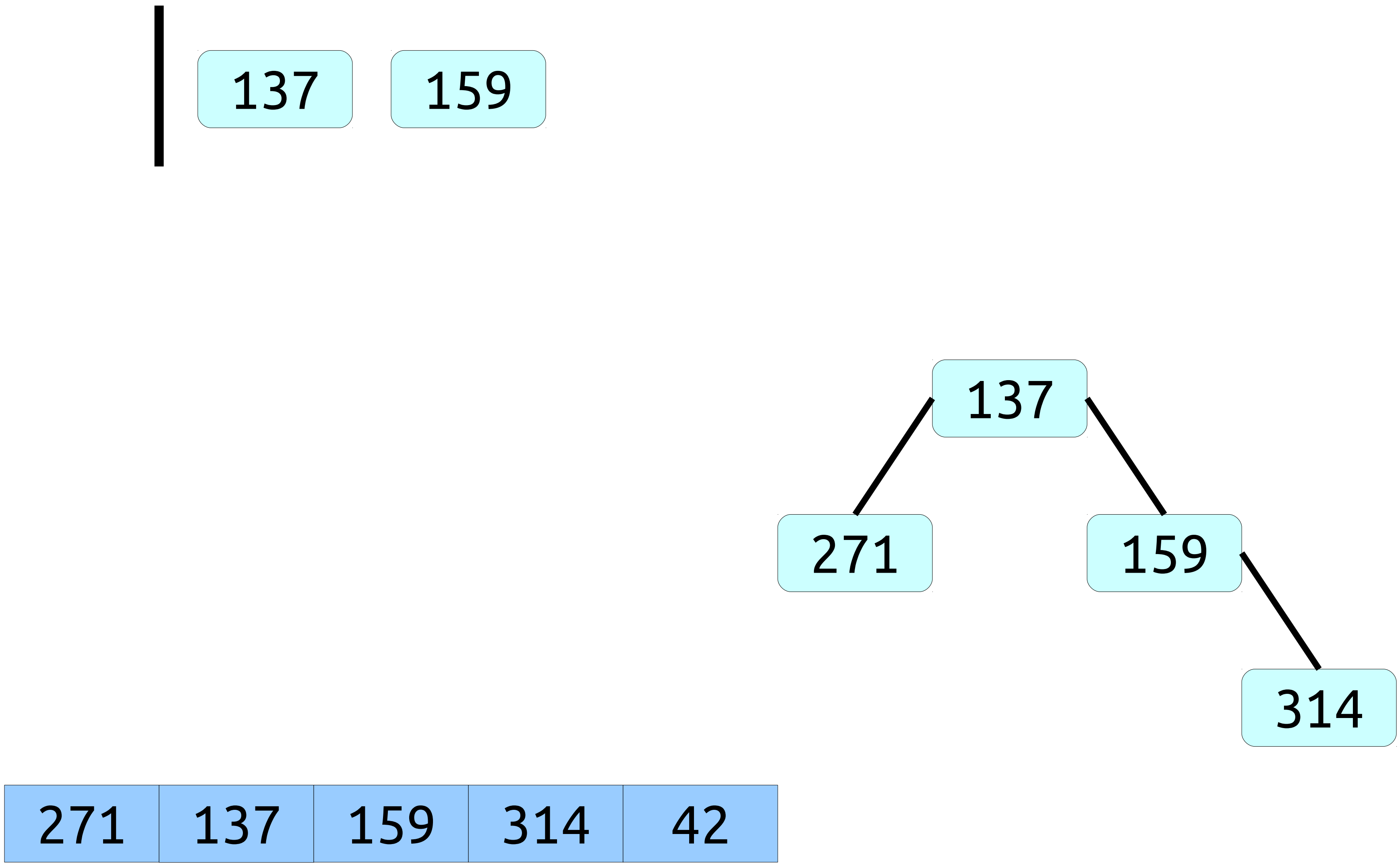
Constructing Cartesian Trees



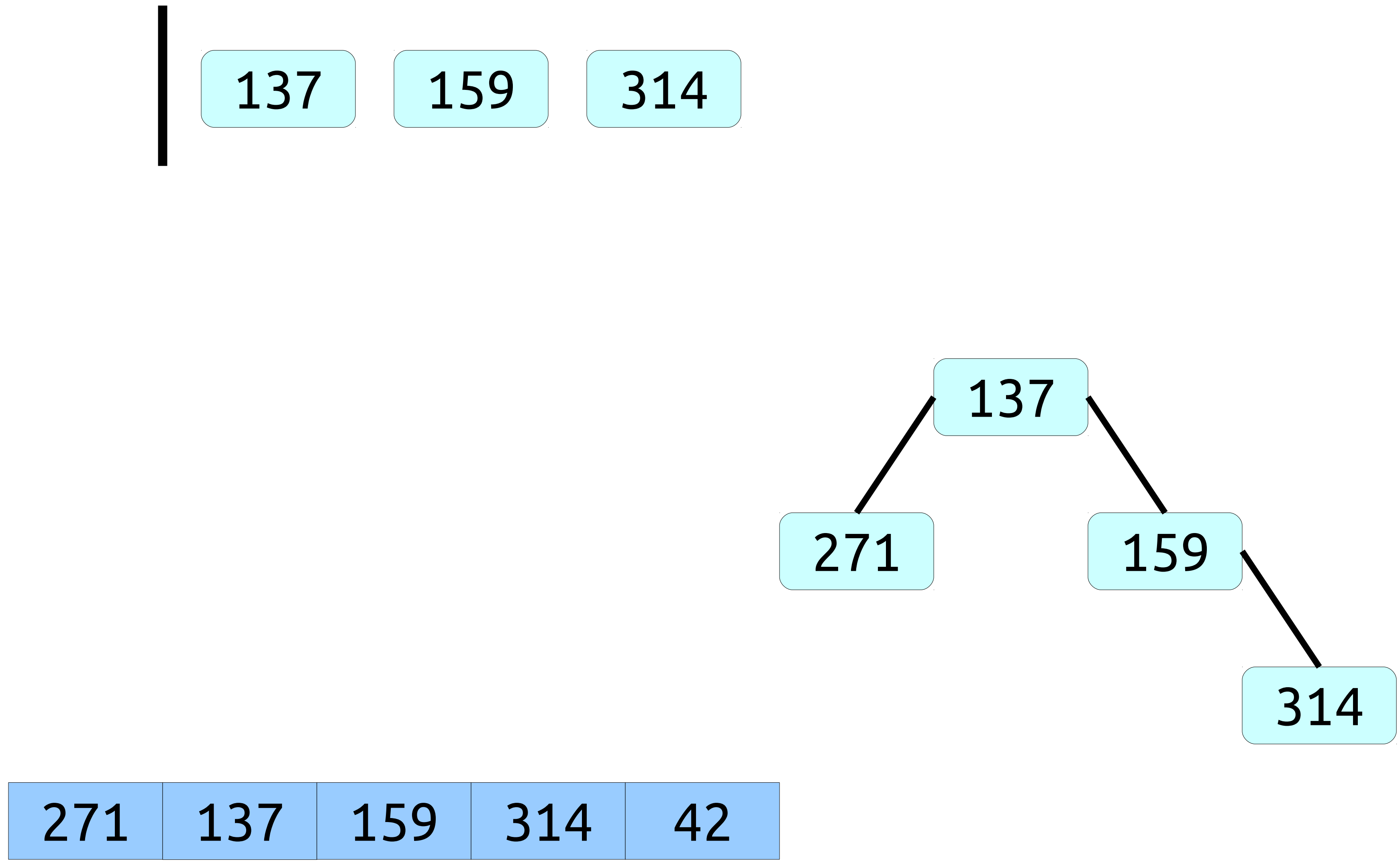
Constructing Cartesian Trees



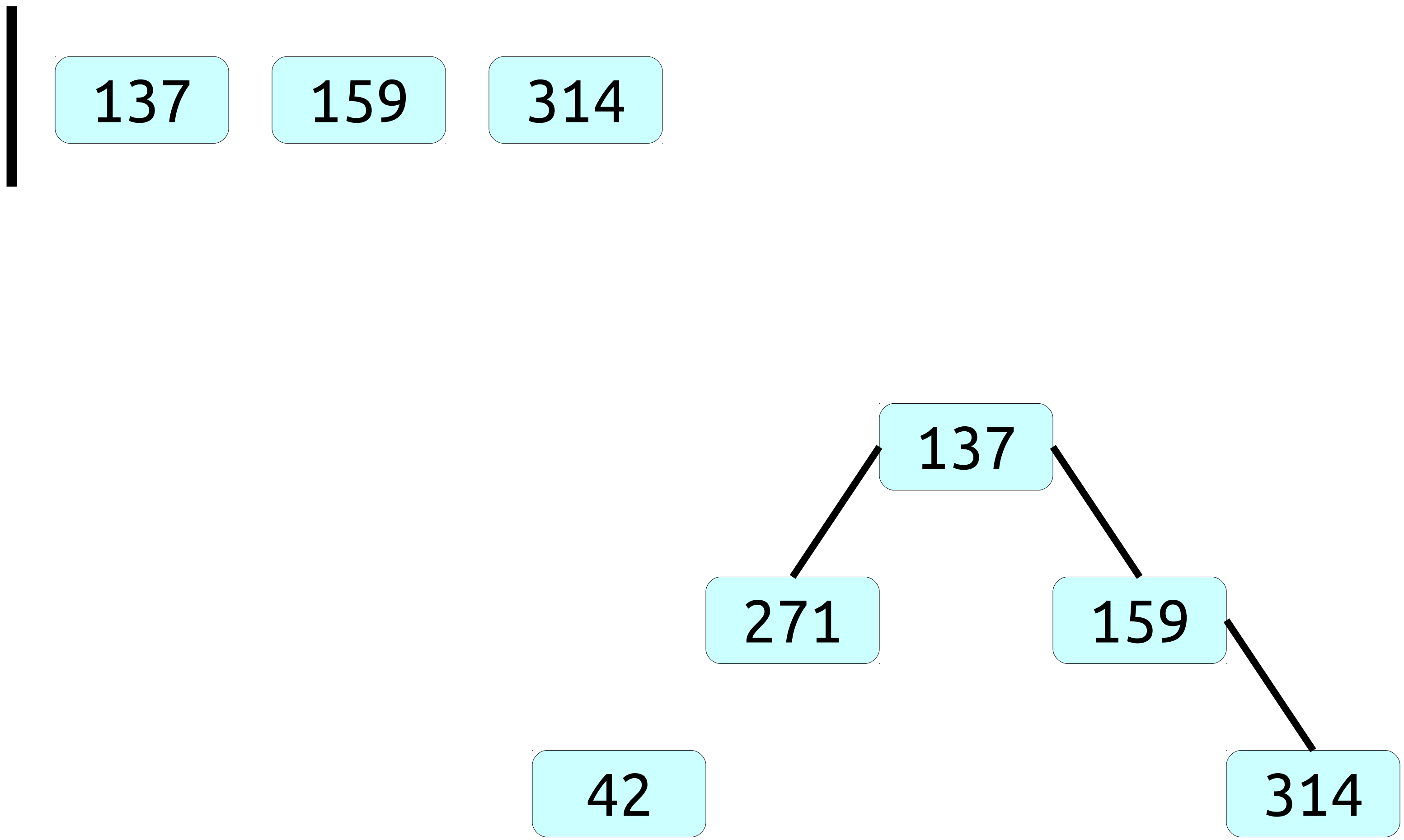
Constructing Cartesian Trees



Constructing Cartesian Trees

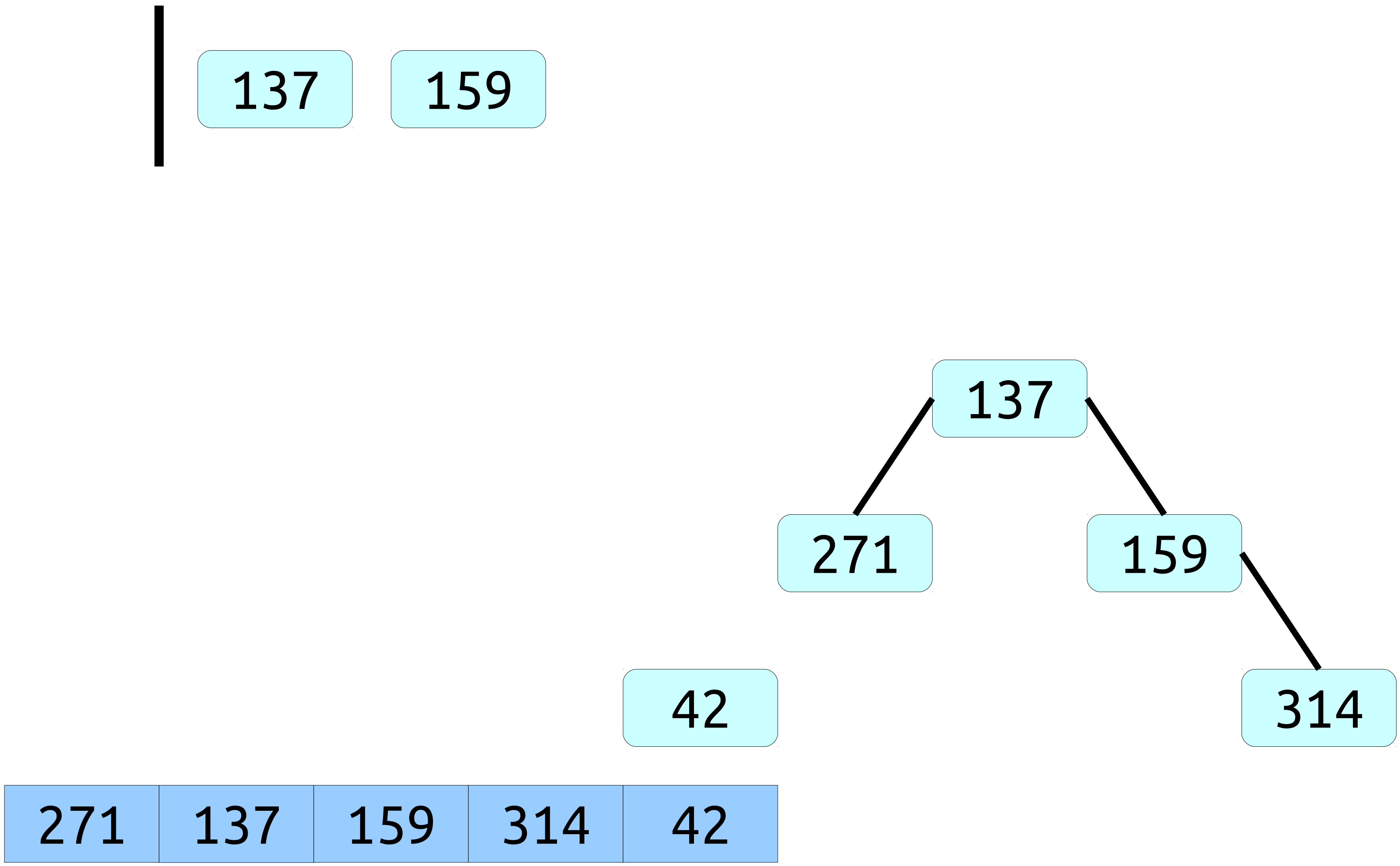


Constructing Cartesian Trees

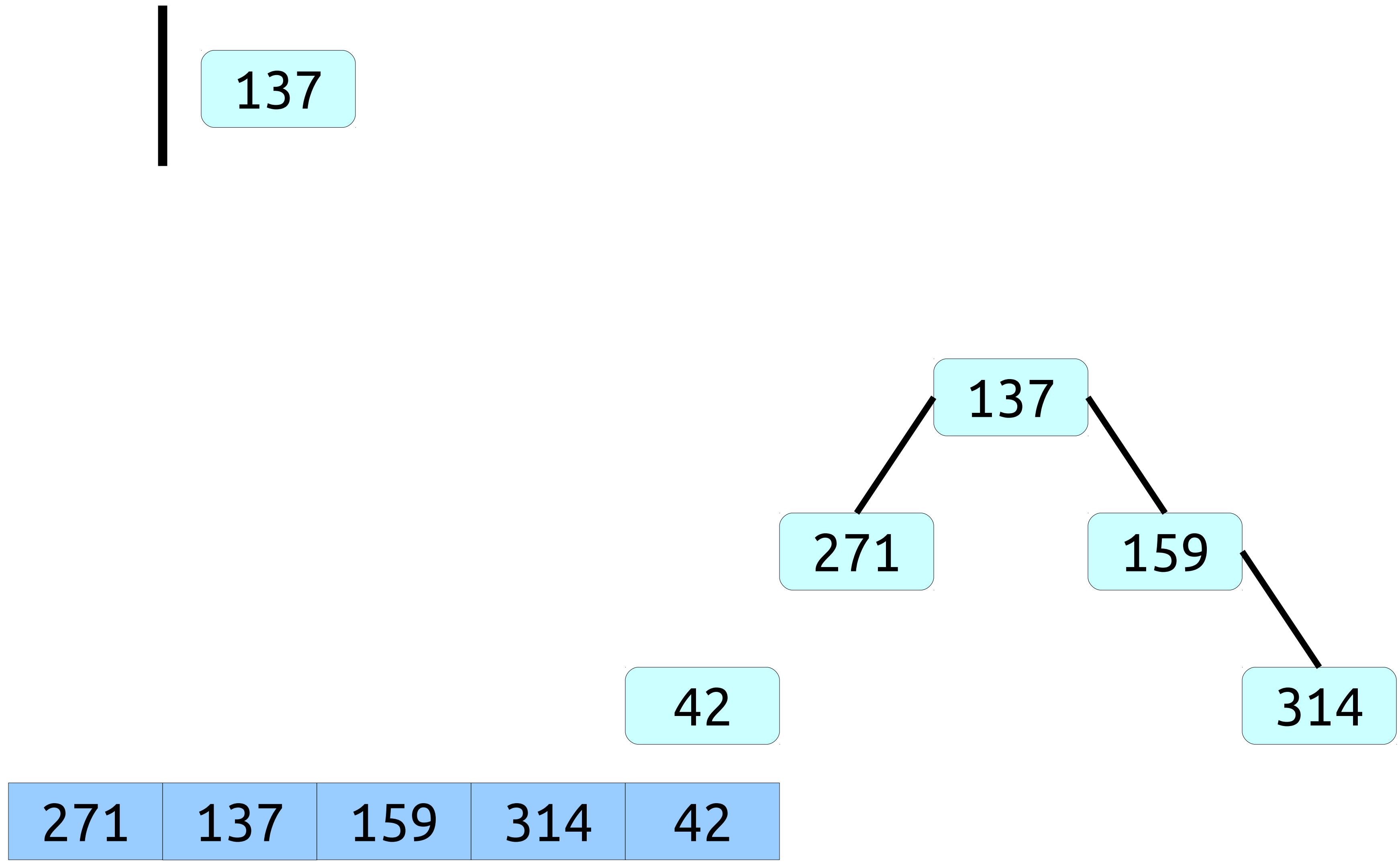


271	137	159	314	42

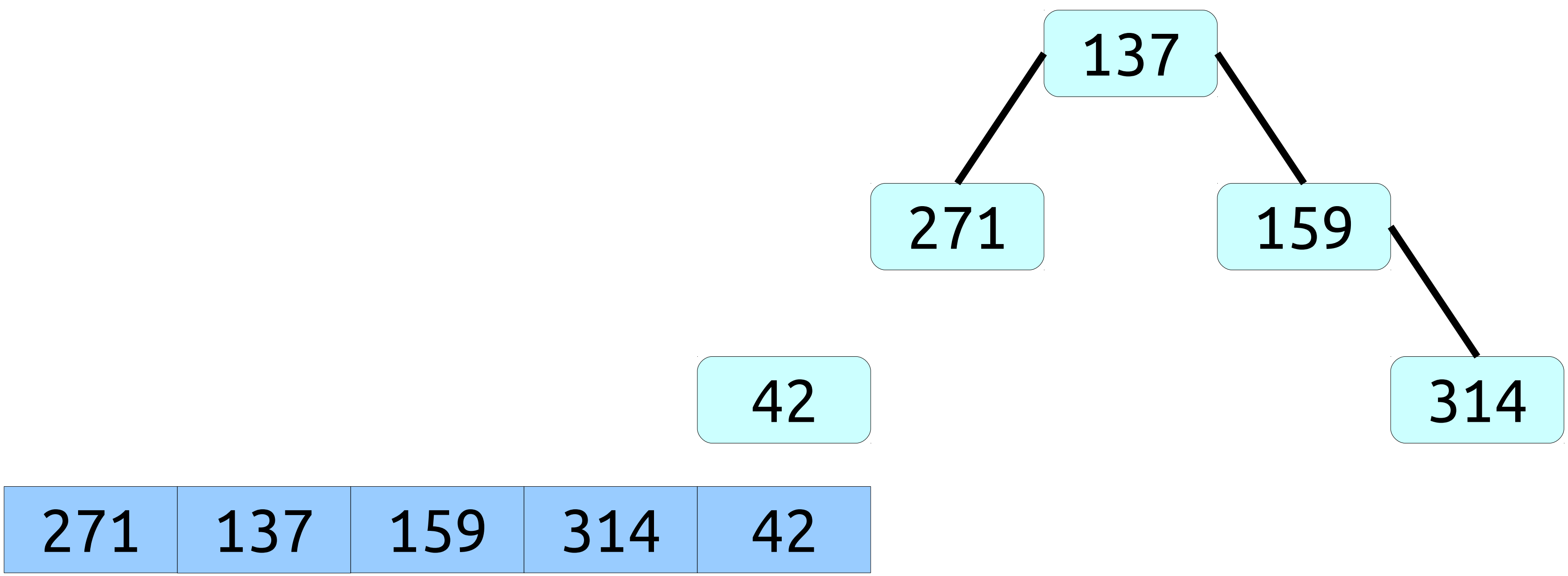
Constructing Cartesian Trees



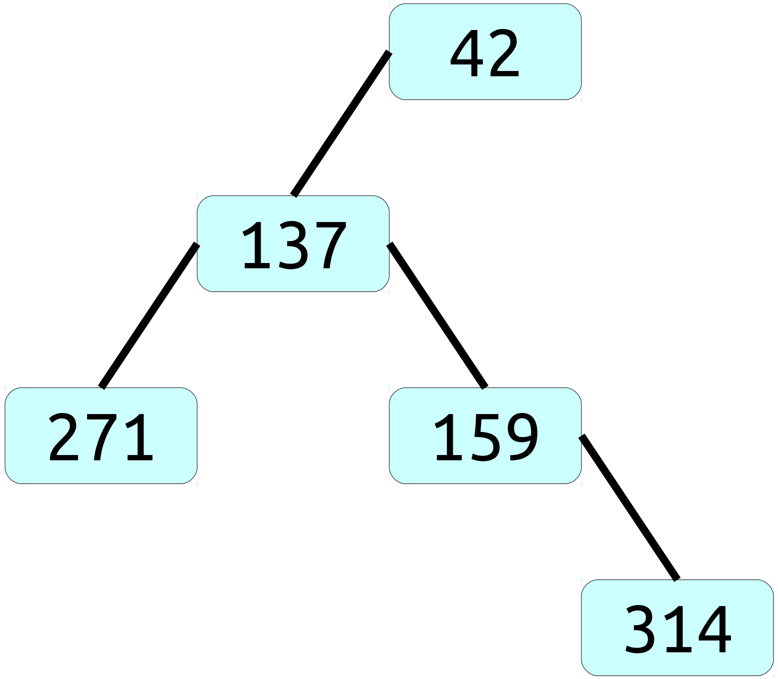
Constructing Cartesian Trees



Constructing Cartesian Trees

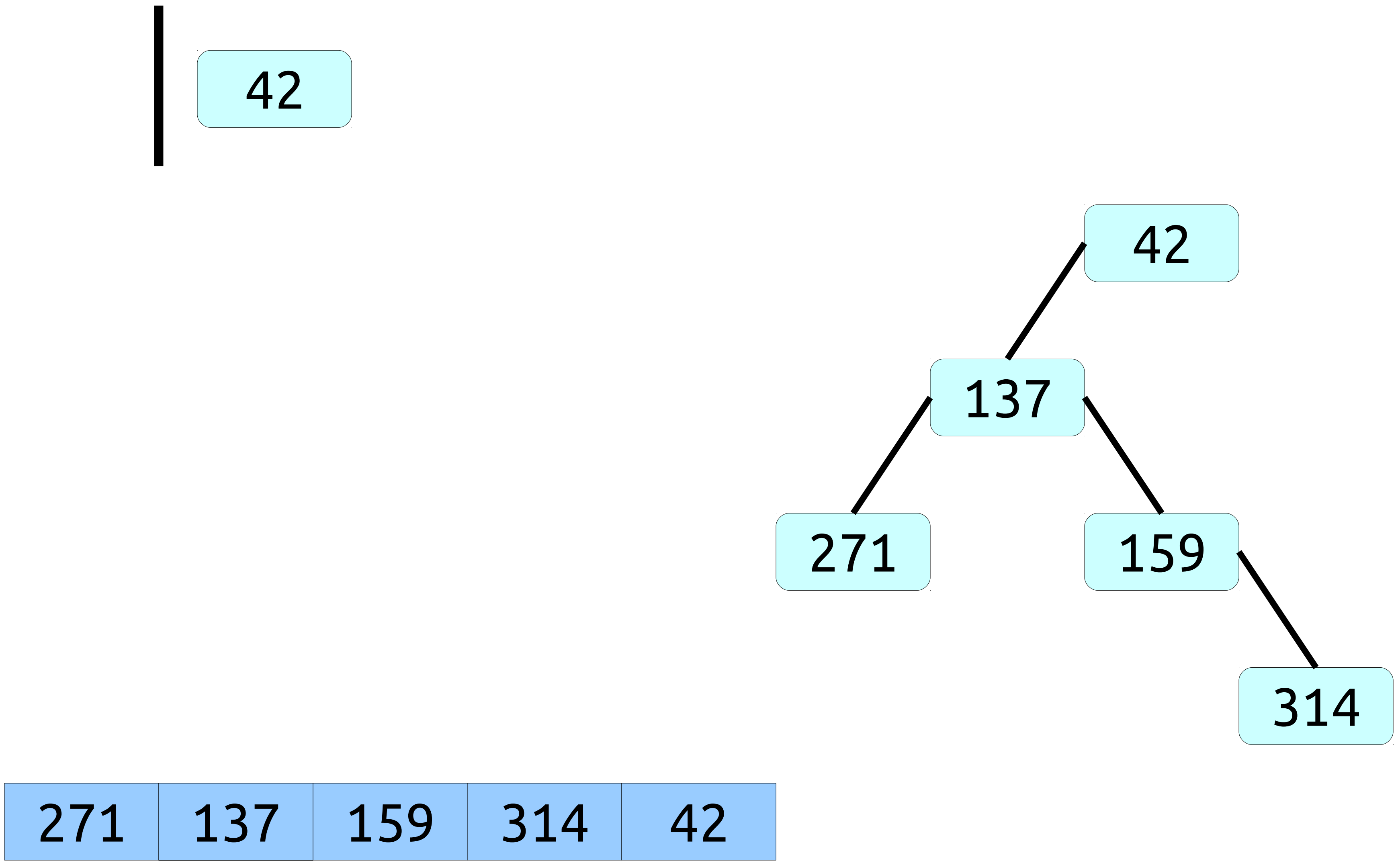


Constructing Cartesian Trees



271	137	159	314	42

Constructing Cartesian Trees



The Runtime Analysis

- In a sequence of operations that adds n elements to a Cartesian tree, adding an individual node to a Cartesian tree might take time $\Theta(n)$.
- However, the net time spent adding new nodes across the whole tree is $O(n)$.
- Why is this?
  - Every node pushed at most once.
  - Every node popped at most once.
  - Work done is proportional to the number of pushes and pops.
  - Total runtime is $O(n)$.
- The ***amortized*** cost of adding a node is $O(n) / n =$ **O(1)**.

Where We're Going

- There are three standard techniques for assigning amortized costs to operations:
  - The ***aggregate method*** directly assigns each operation its average cost.
  - The ***banker's method*** places credits on the data structure, which are redeemable for units of work.
  - The ***potential method*** assigns a potential function to the data structure, which can be charged to pay for future work or released to pay for recent work.

Where We're Going

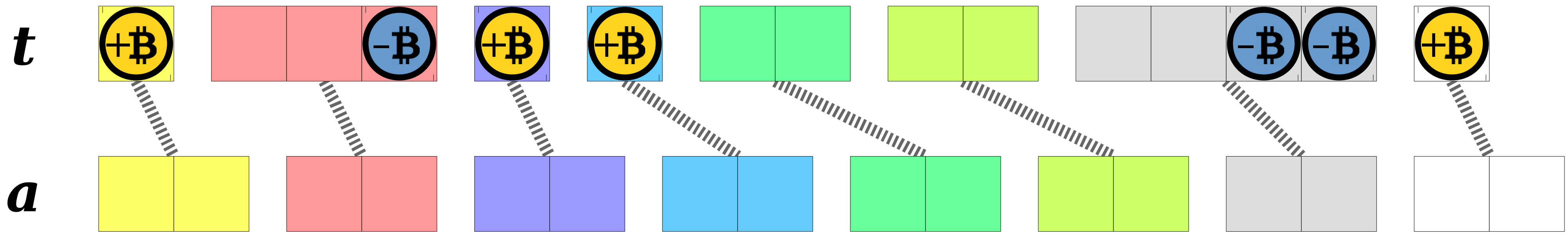
- There are three standard techniques for assigning amortized costs to operations:
  - The ***aggregate method*** directly assigns each operation its average cost.
  - The ***banker's method*** places credits on the data structure, which are redeemable for units of work.
  - The ***potential method*** assigns a potential function to the data structure, which can be charged to pay for future work or released to pay for recent work.

The Banker's Method

- In the ***banker's method***, operations can place ***credits*** on the data structure or spend credits that have already been placed.
- Placing a credit on the data structure takes time O(1).
- Spending a credit previously placed on the data structure takes time -O(1). *(Yes, that's negative time!)*
- The amortized cost of an operation is then

$$a(op_i) = t(op_i) + O(1) \cdot (added_i - removed_i)$$

- There aren't any real credits anywhere. They're just an accounting trick.



The Banker's Method

In the ***banker's method***, operations can place ***credits*** on the data structure or spend credits that have already been placed.

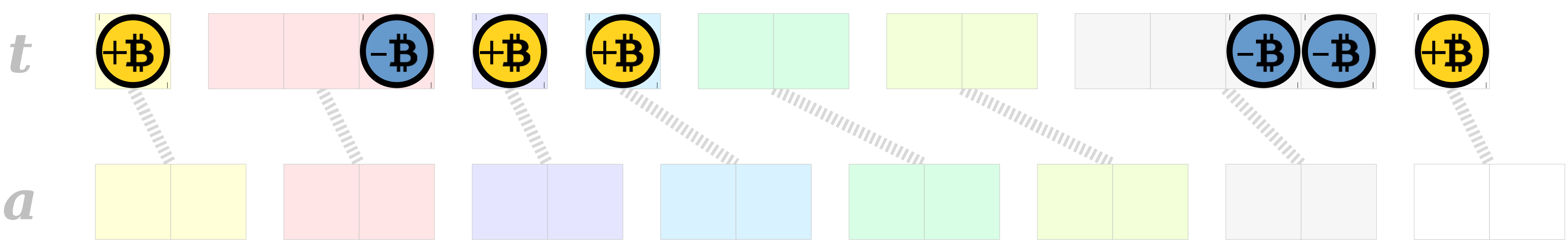
Placing a credit on the data structure takes time O(1).

Spending a credit previously placed on the data structure takes time -O(1). *(Yes, that's negative time!)*

The amortized cost of an operation is then

$$a(op_i) = t(op_i) + O(1) \cdot (added_i - removed_i)$$

- There aren't any real credits anywhere. They're just an accounting trick.



The Banker's Method

- If we never spend credits we don't have:

$$\sum_{i=1}^{k} a(op_i) = \sum_{i=1}^{k} \left(t(op_i) + \mathrm{O}(1) \cdot (added_i - removed_i)\right)$$

The Banker's Method

- If we never spend credits we don't have:

$$\begin{aligned}\sum_{i=1}^{k} a(op_i) &= \sum_{i=1}^{k} \left(t(op_i) + O(1) \cdot (added_i - removed_i)\right) \\ &= \sum_{i=1}^{k} t(op_i) + O(1) \sum_{i=1}^{k} (added_i - removed_i)\end{aligned}$$

The Banker's Method

- If we never spend credits we don't have:

$$
\begin{aligned}
\sum_{i=1}^{k} a(op_i) &= \sum_{i=1}^{k} \left(t(op_i) + O(1) \cdot (added_i - removed_i)\right) \\
&= \sum_{i=1}^{k} t(op_i) + O(1) \sum_{i=1}^{k} (added_i - removed_i) \\
&= \sum_{i=1}^{k} t(op_i) + O(1) \cdot netCredits
\end{aligned}
$$

The Banker's Method

- If we never spend credits we don't have:

$$\begin{aligned}
\sum_{i=1}^{k} a(op_i) &= \sum_{i=1}^{k} \left(t(op_i) + \mathrm{O}(1) \cdot (added_i - removed_i)\right) \\
&= \sum_{i=1}^{k} t(op_i) + \mathrm{O}(1) \sum_{i=1}^{k} (added_i - removed_i) \\
&= \sum_{i=1}^{k} t(op_i) + \mathrm{O}(1) \cdot netCredits \\
&\geq \sum_{i=1}^{k} t(op_i)
\end{aligned}$$

The Banker's Method

- If we never spend credits we don't have:

$$\begin{aligned}
\sum_{i=1}^{k} a(op_i) &= \sum_{i=1}^{k} (t(op_i) + O(1) \cdot (added_i - removed_i)) \\
&= \sum_{i=1}^{k} t(op_i) + O(1) \sum_{i=1}^{k} (added_i - removed_i) \\
&= \sum_{i=1}^{k} t(op_i) + O(1) \cdot netCredits \\
&\geq \sum_{i=1}^{k} t(op_i)
\end{aligned}$$

- The sum of the amortized costs upper-bounds the sum of the true costs.

Constructing Cartesian Trees

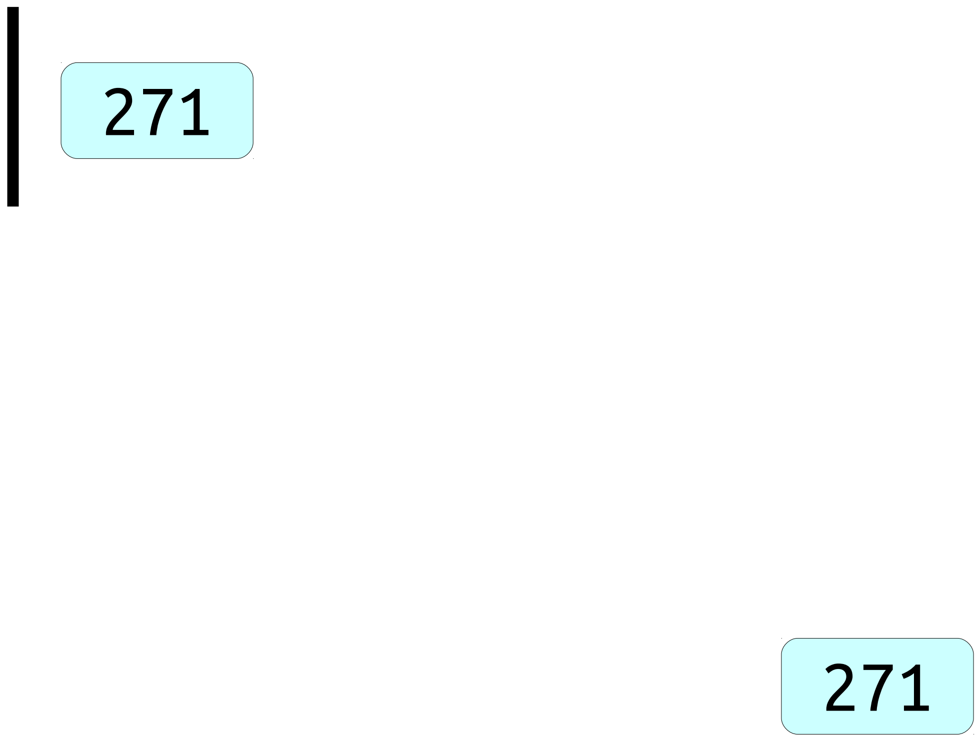
271	137	159	314	42

Constructing Cartesian Trees

271

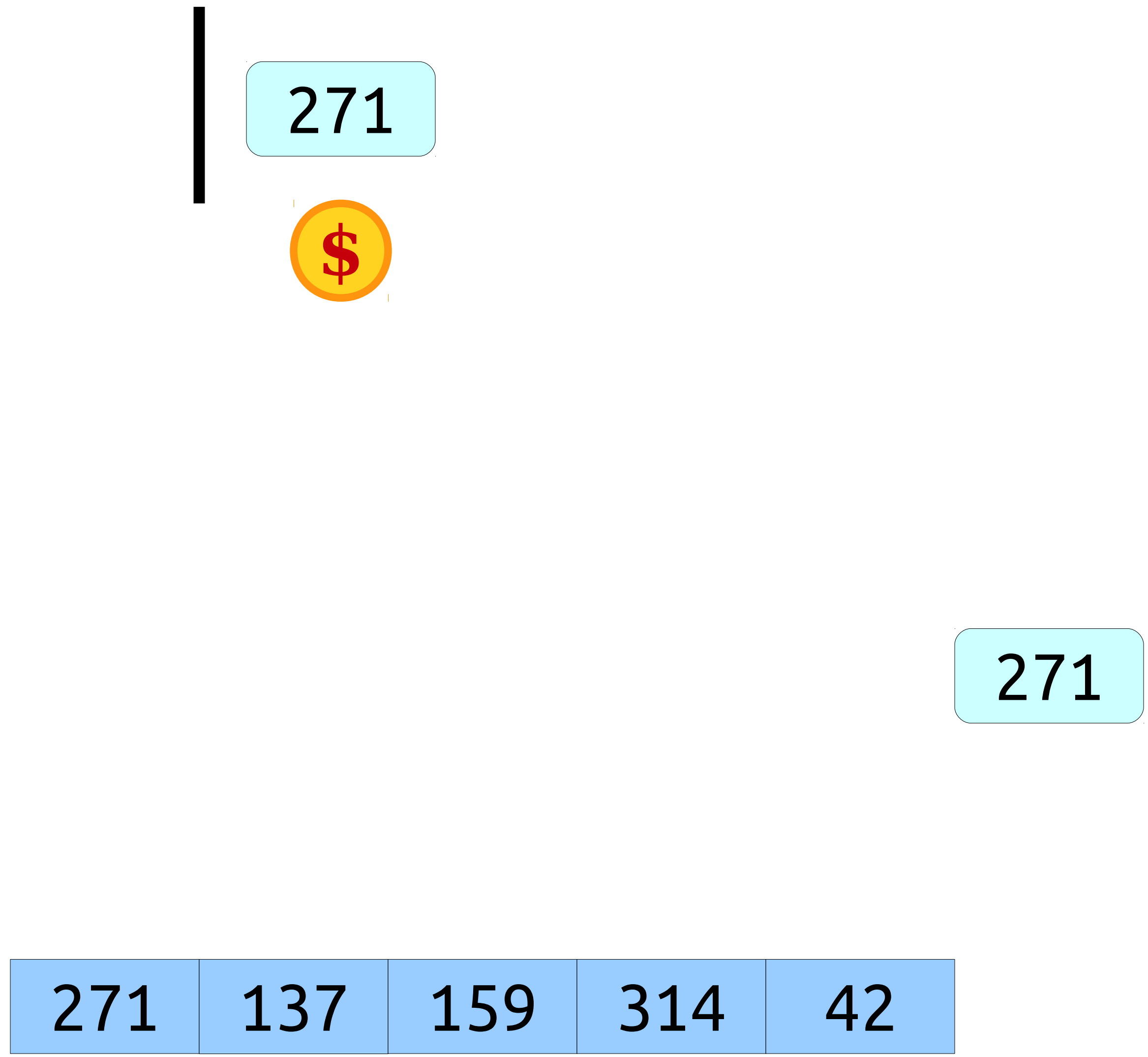
271	137	159	314	42

Constructing Cartesian Trees



271	137	159	314	42

Constructing Cartesian Trees



Constructing Cartesian Trees

271

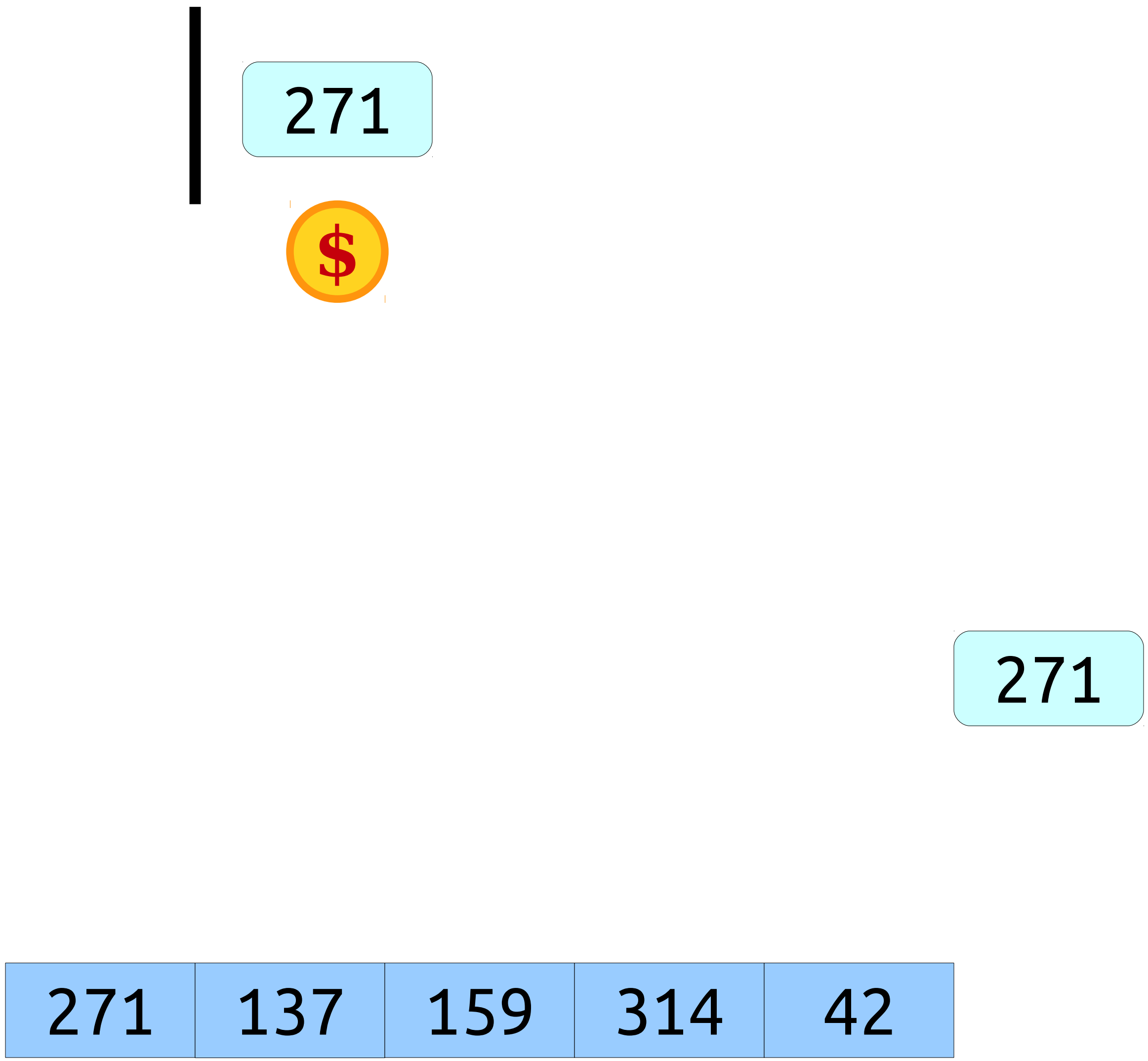
Work done: 1 push
Credits Added: $1

Amortized Cost: **2**

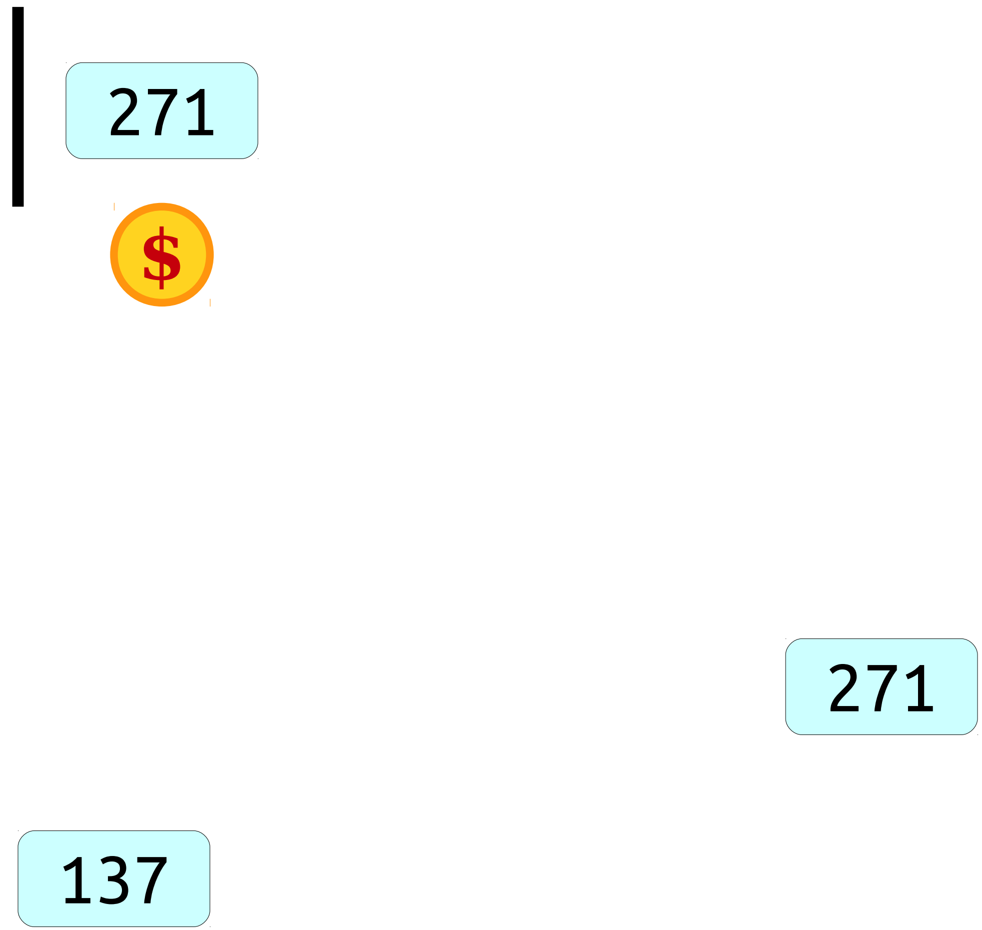
271

271	137	159	314	42

Constructing Cartesian Trees

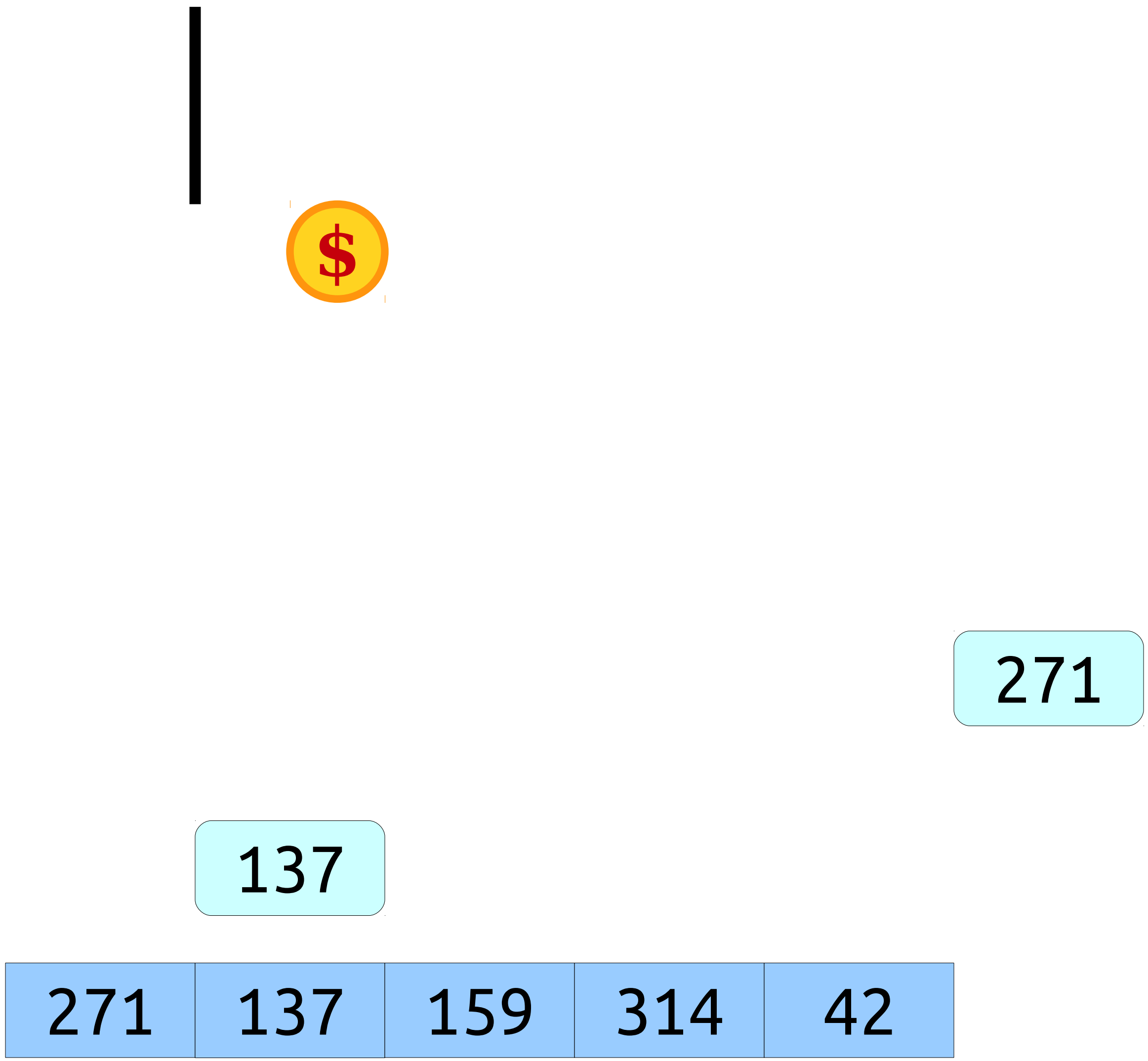


Constructing Cartesian Trees

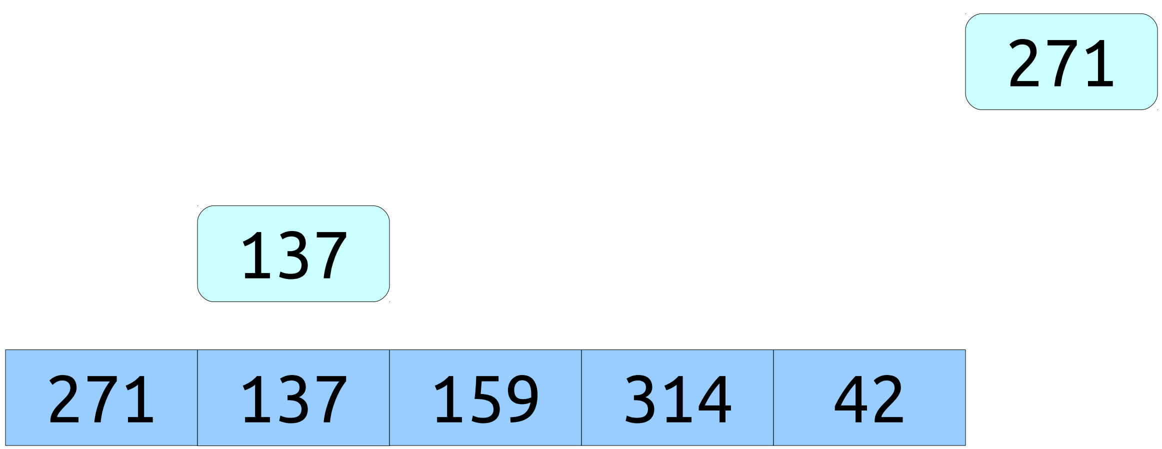


271	137	159	314	42

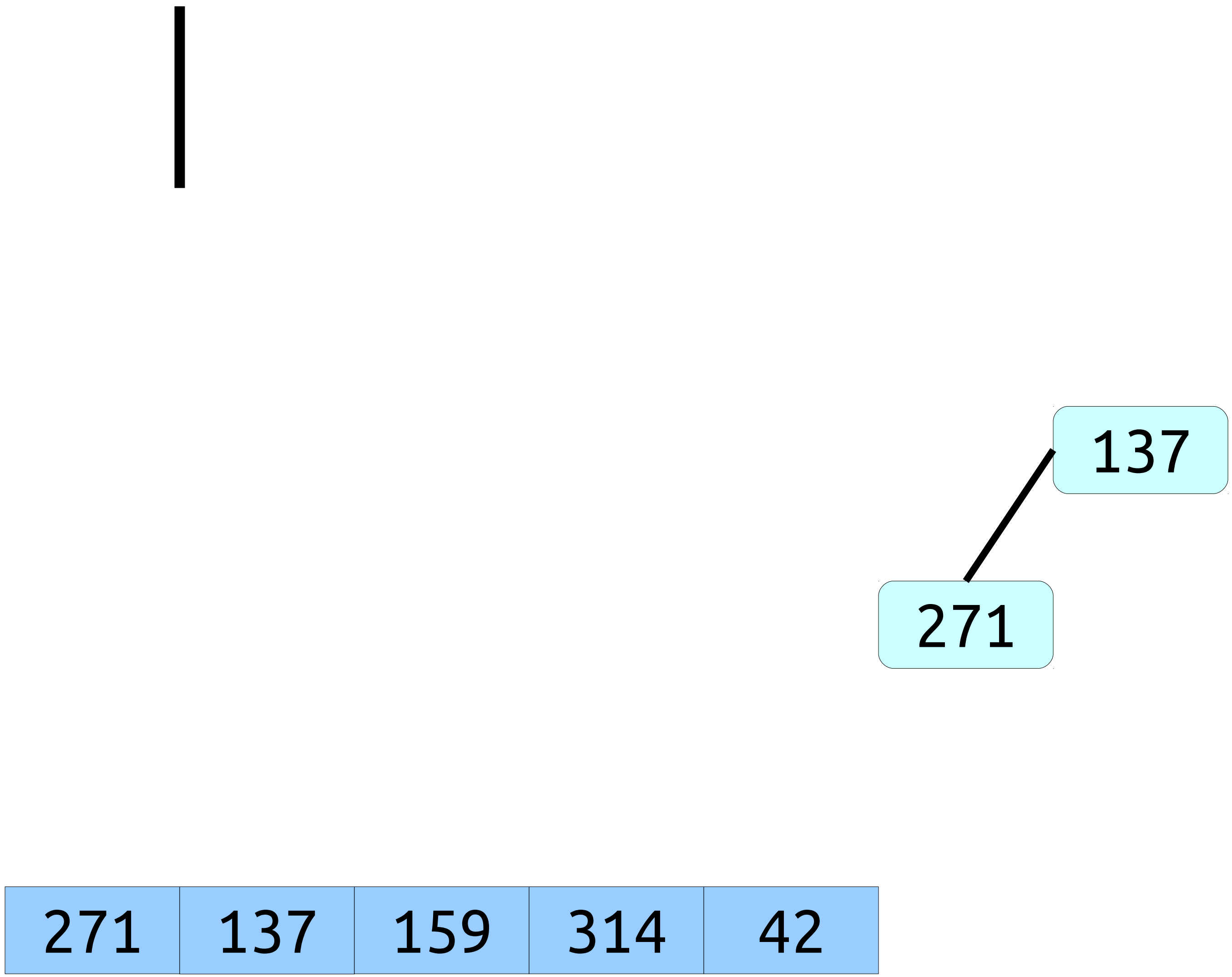
Constructing Cartesian Trees



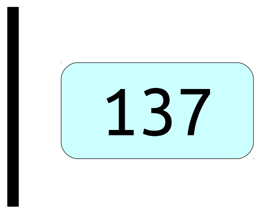
Constructing Cartesian Trees

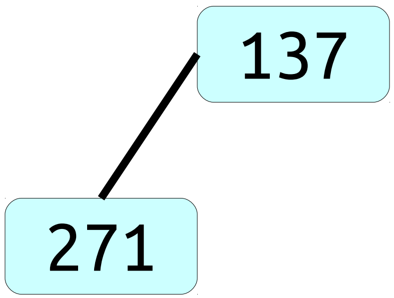


Constructing Cartesian Trees



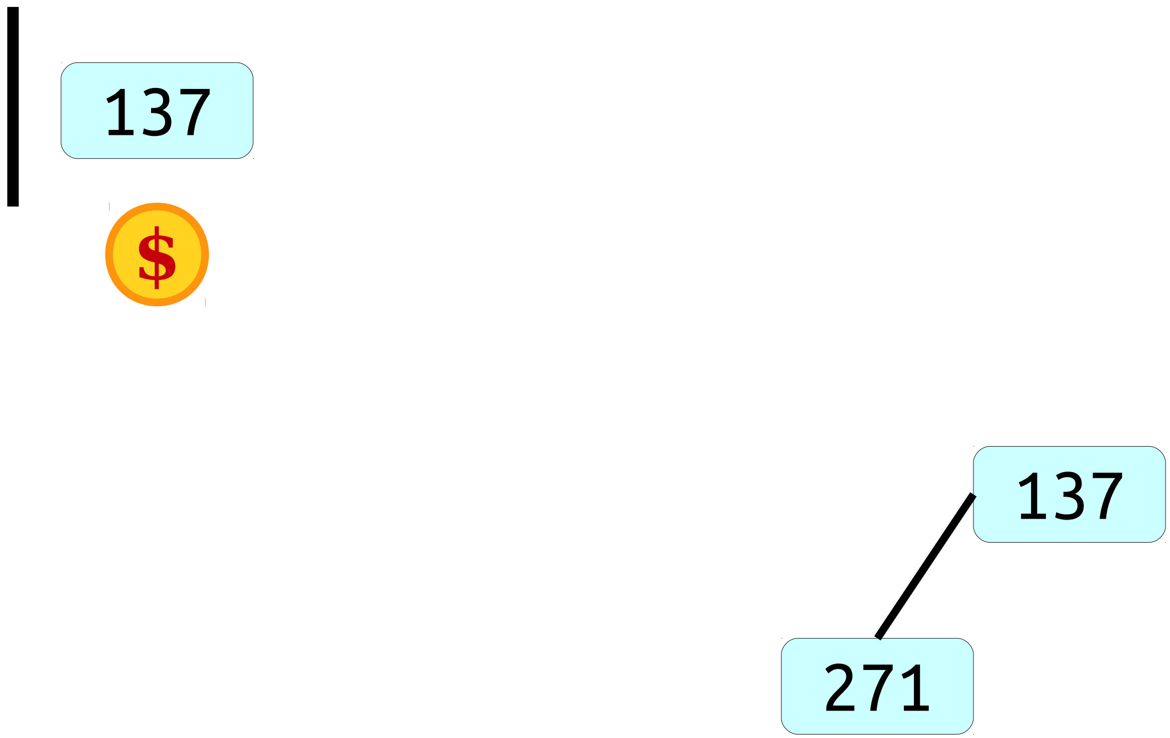
Constructing Cartesian Trees





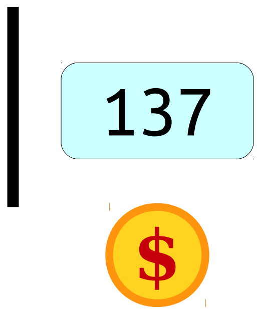
271	137	159	314	42

Constructing Cartesian Trees



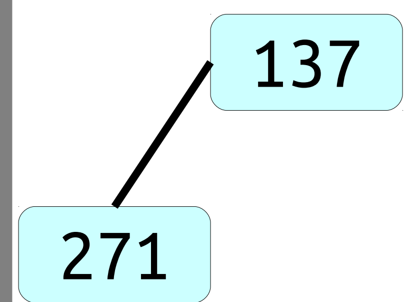
271	137	159	314	42

Constructing Cartesian Trees



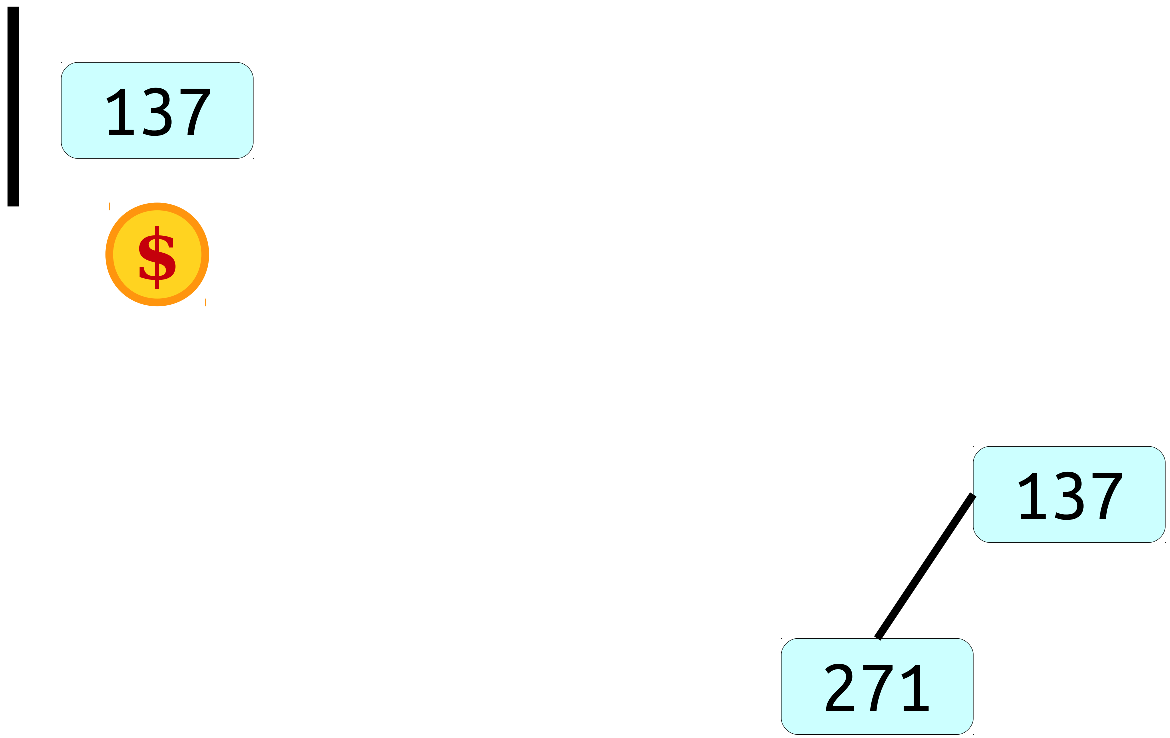
Work done: 1 push, 1 pop
Credits Removed: $1
Credits Added: $1

Amortized Cost: **2**



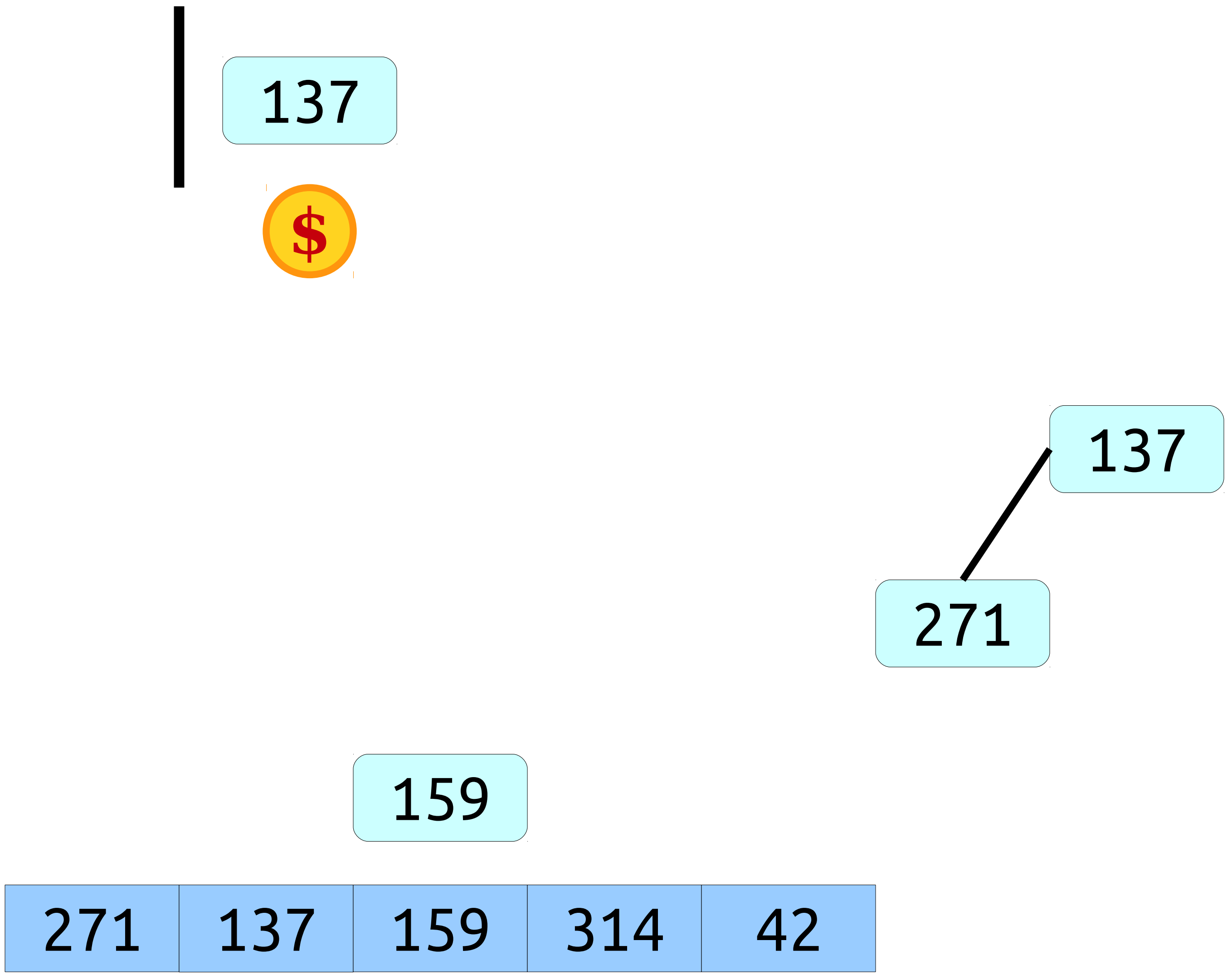
271	137	159	314	42

Constructing Cartesian Trees

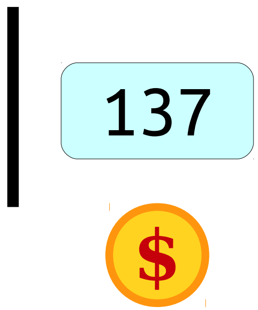


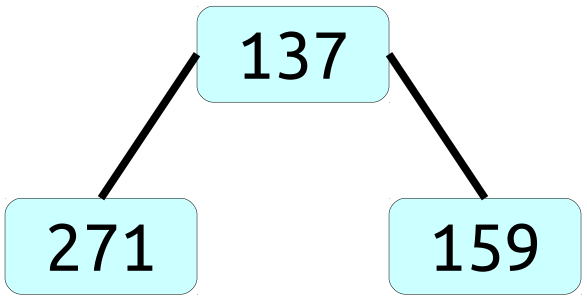
271	137	159	314	42

Constructing Cartesian Trees



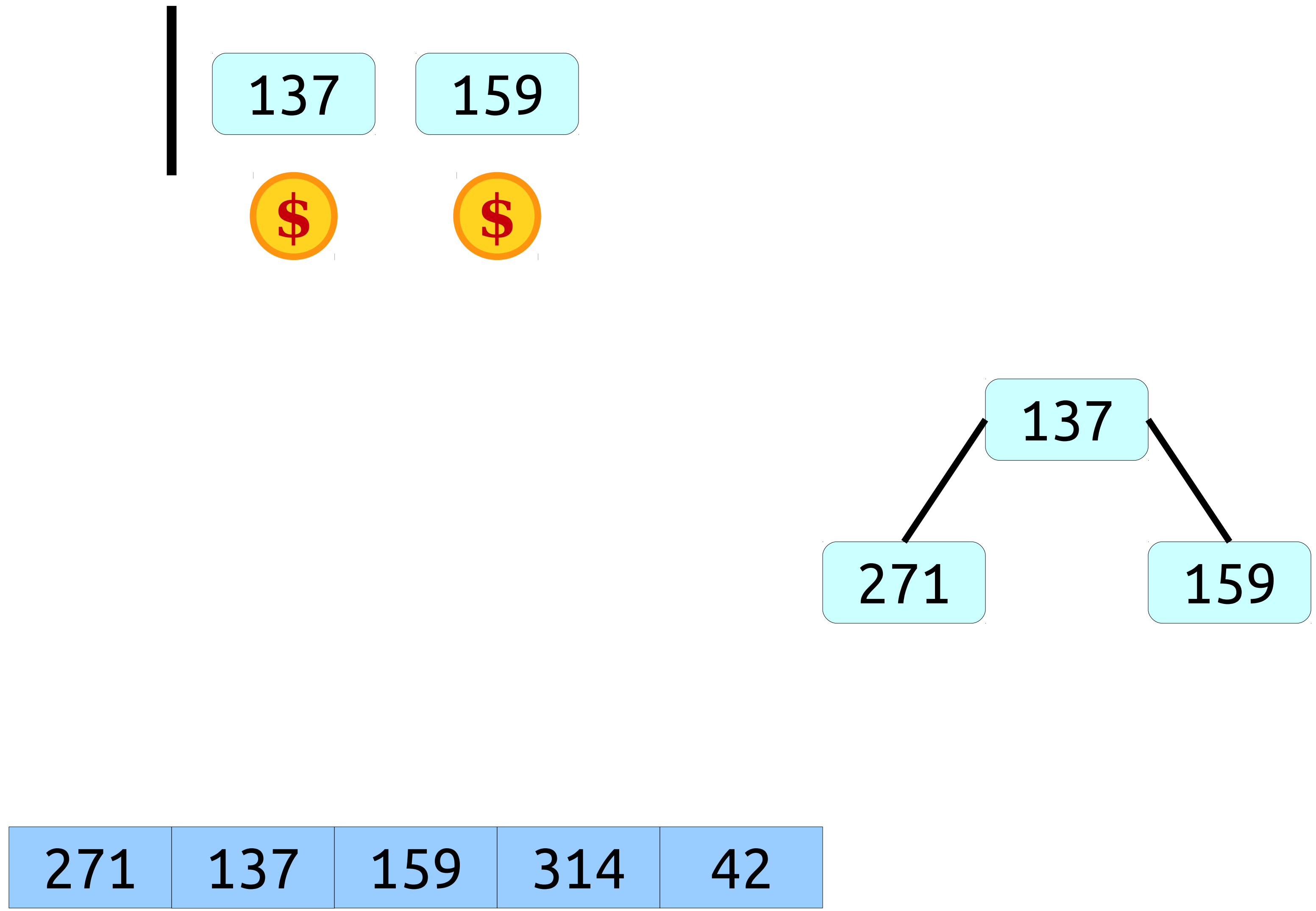
Constructing Cartesian Trees



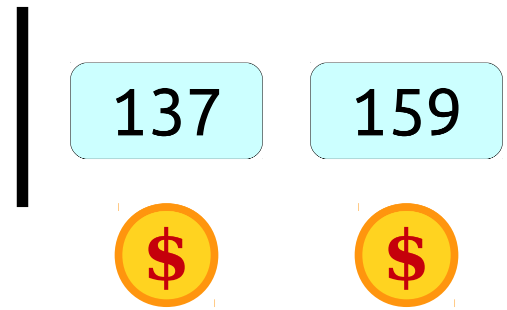


271	137	159	314	42

Constructing Cartesian Trees

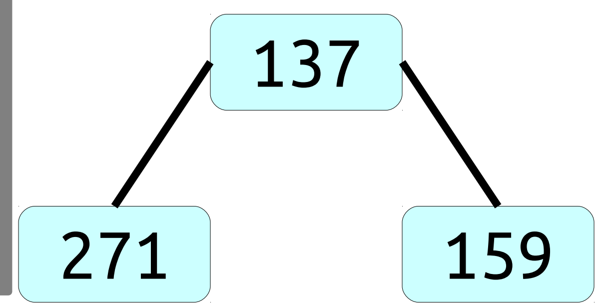


Constructing Cartesian Trees



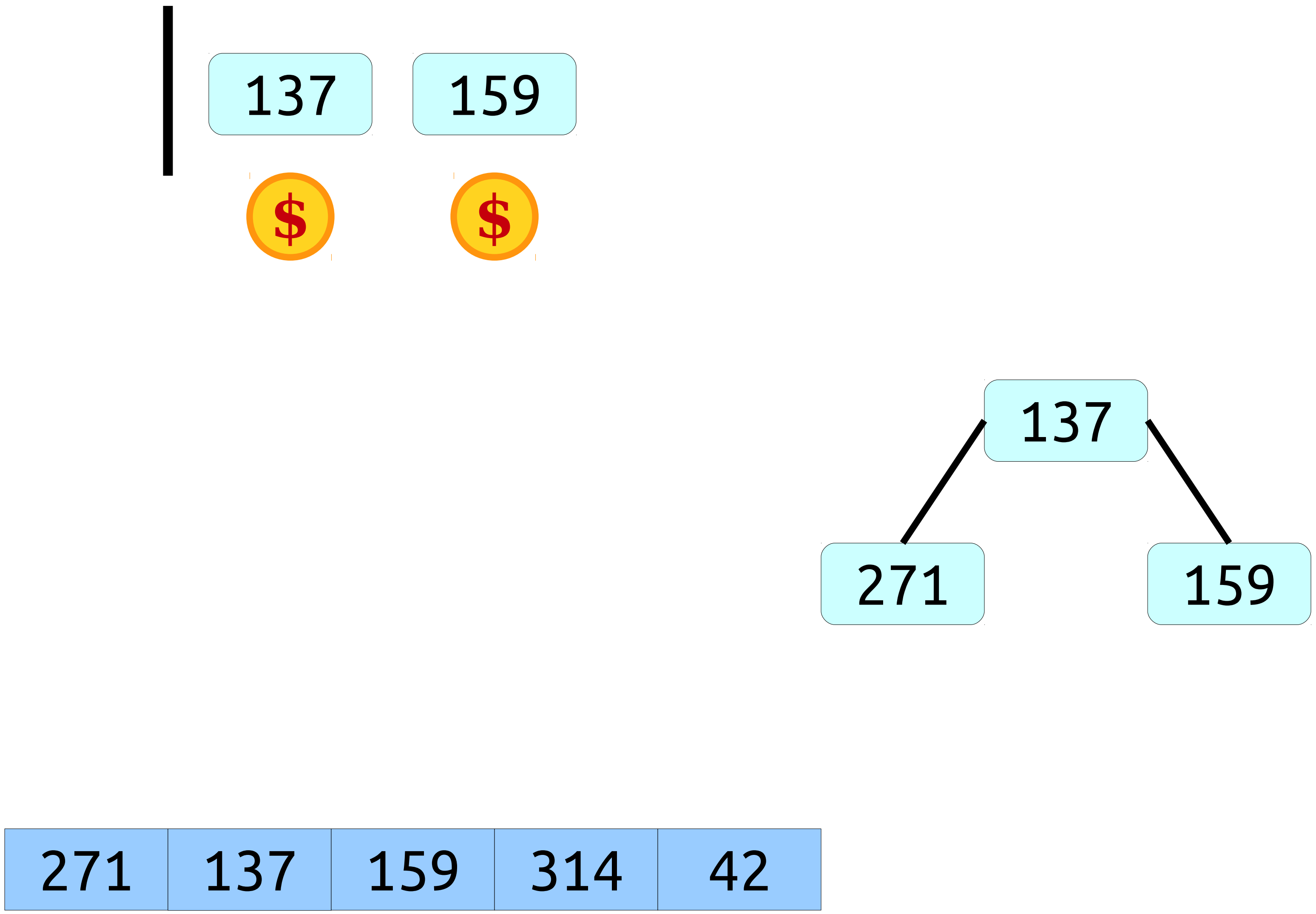
Work done: 1 push
Credits Added: $1

Amortized Cost: **2**

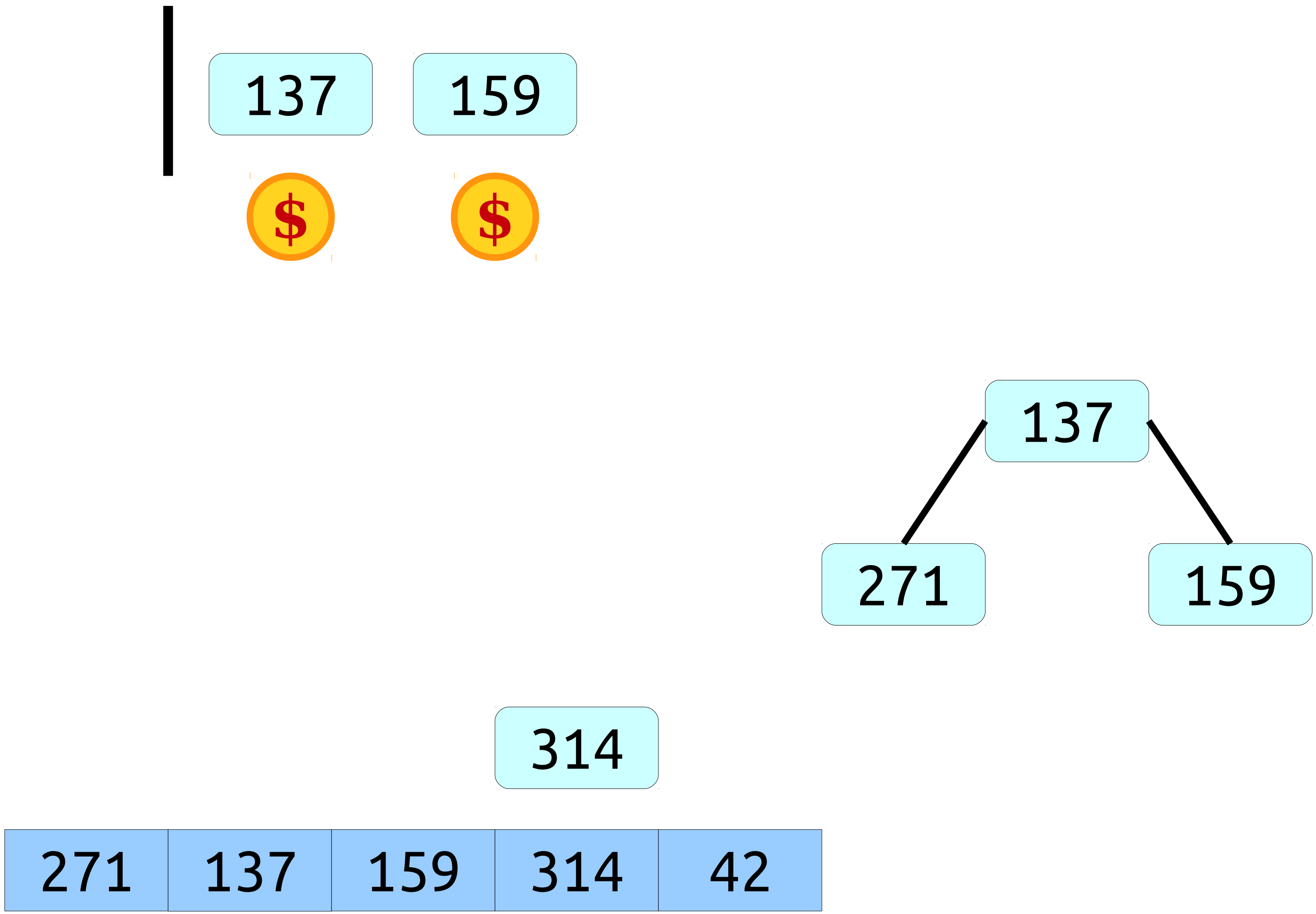


271	137	159	314	42

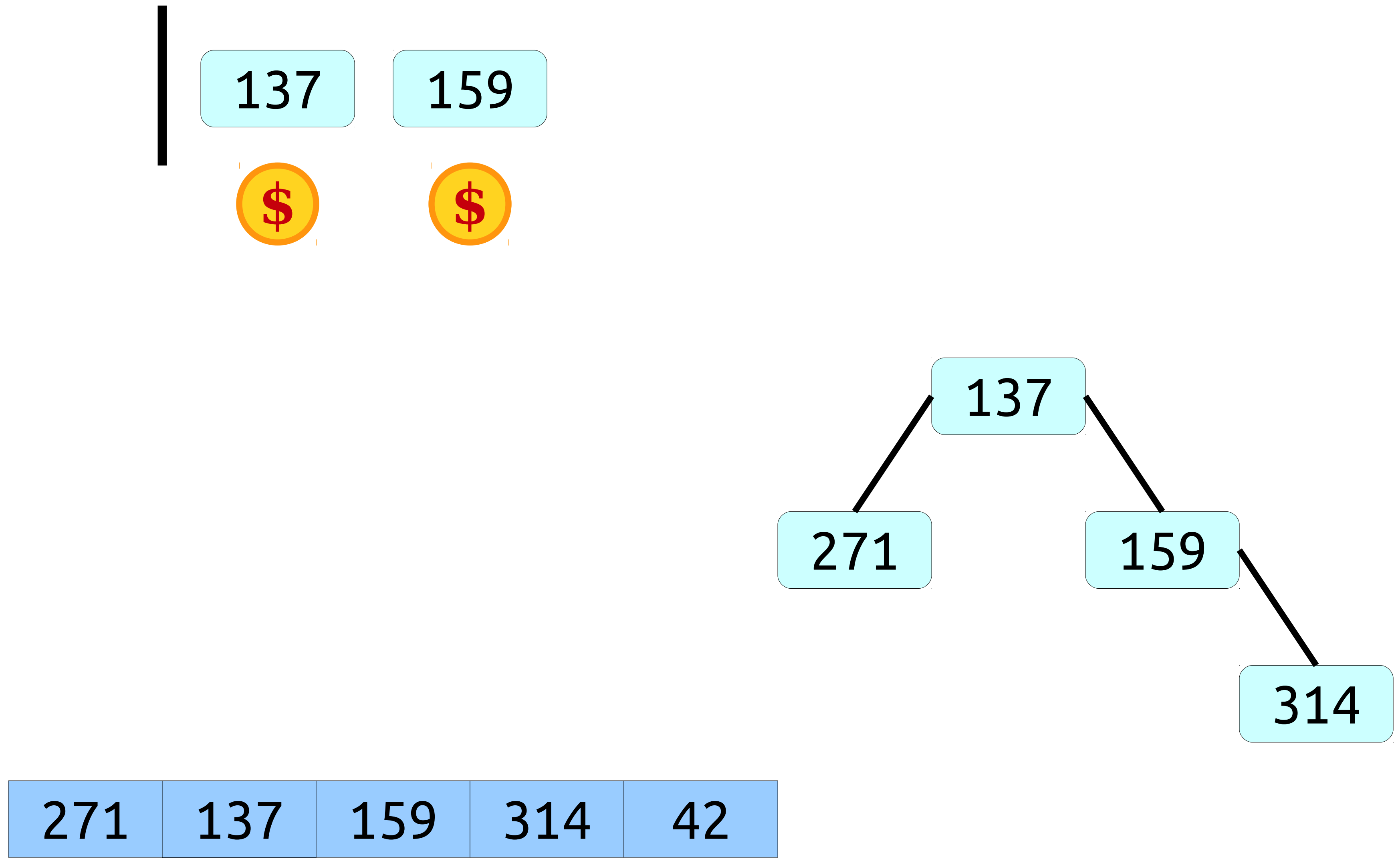
Constructing Cartesian Trees



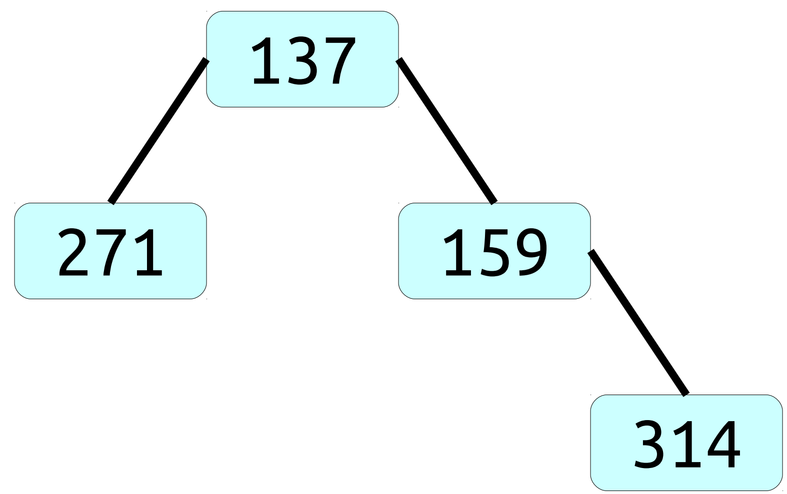
Constructing Cartesian Trees



Constructing Cartesian Trees



Constructing Cartesian Trees

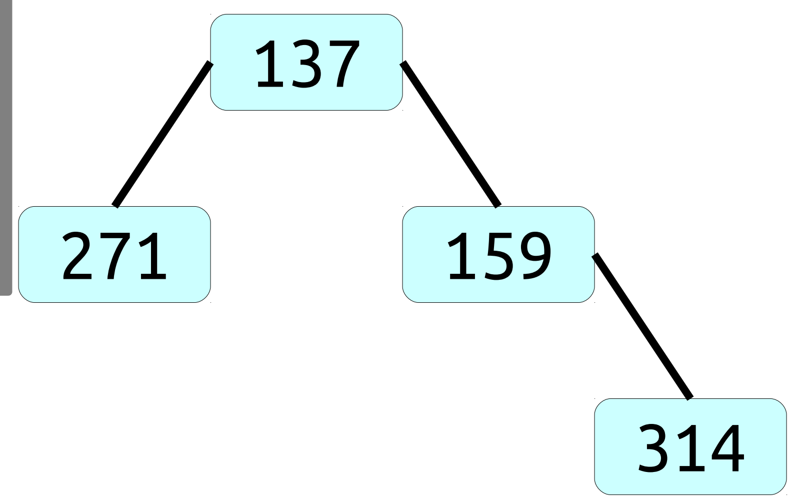


271	137	159	314	42

Constructing Cartesian Trees

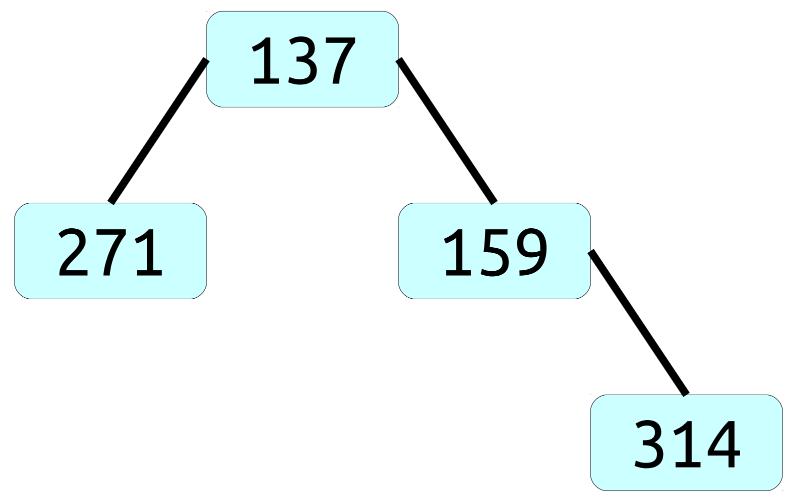
Work done: 1 push
Credits Added: $1

Amortized Cost: **2**



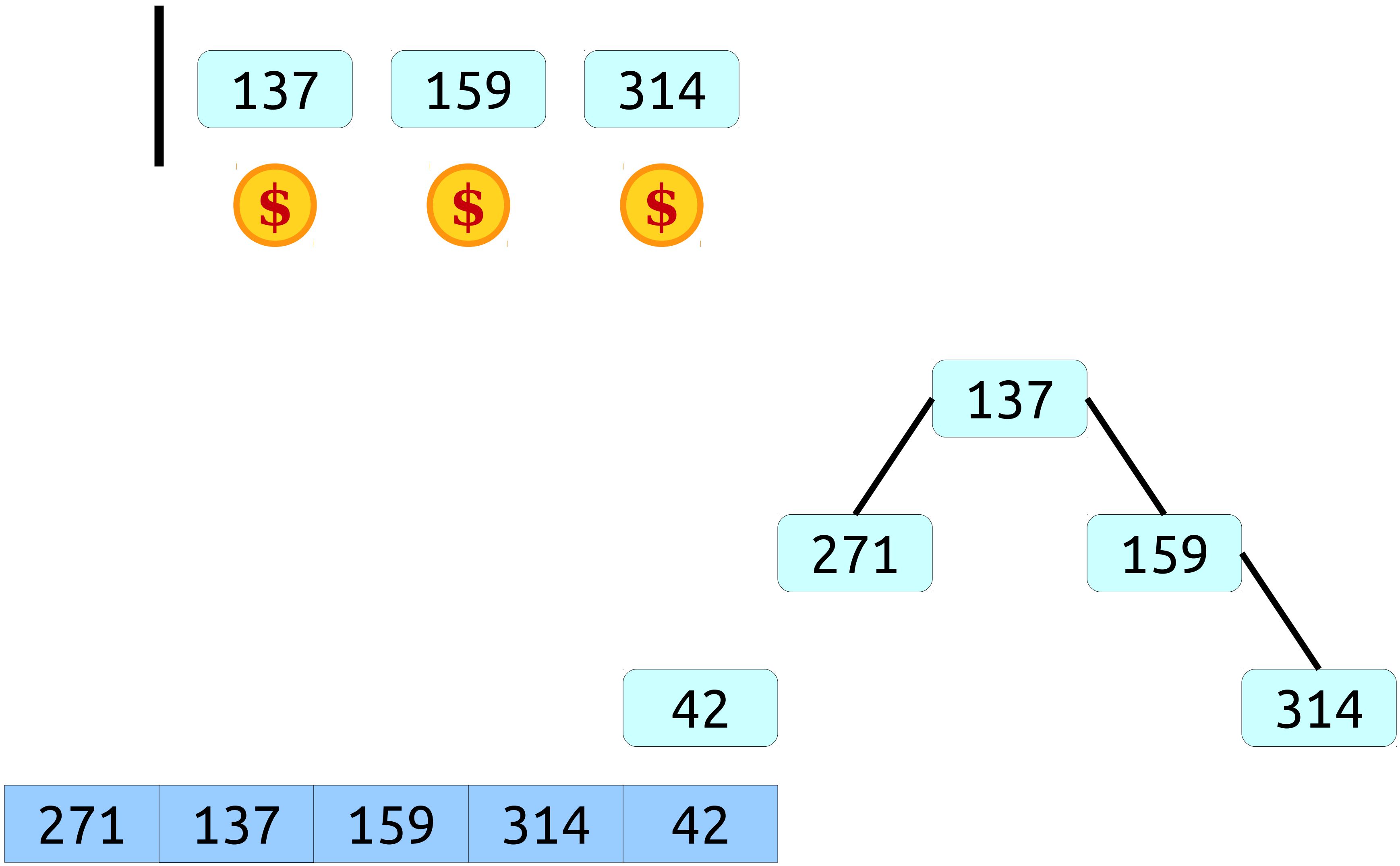
271	137	159	314	42

Constructing Cartesian Trees

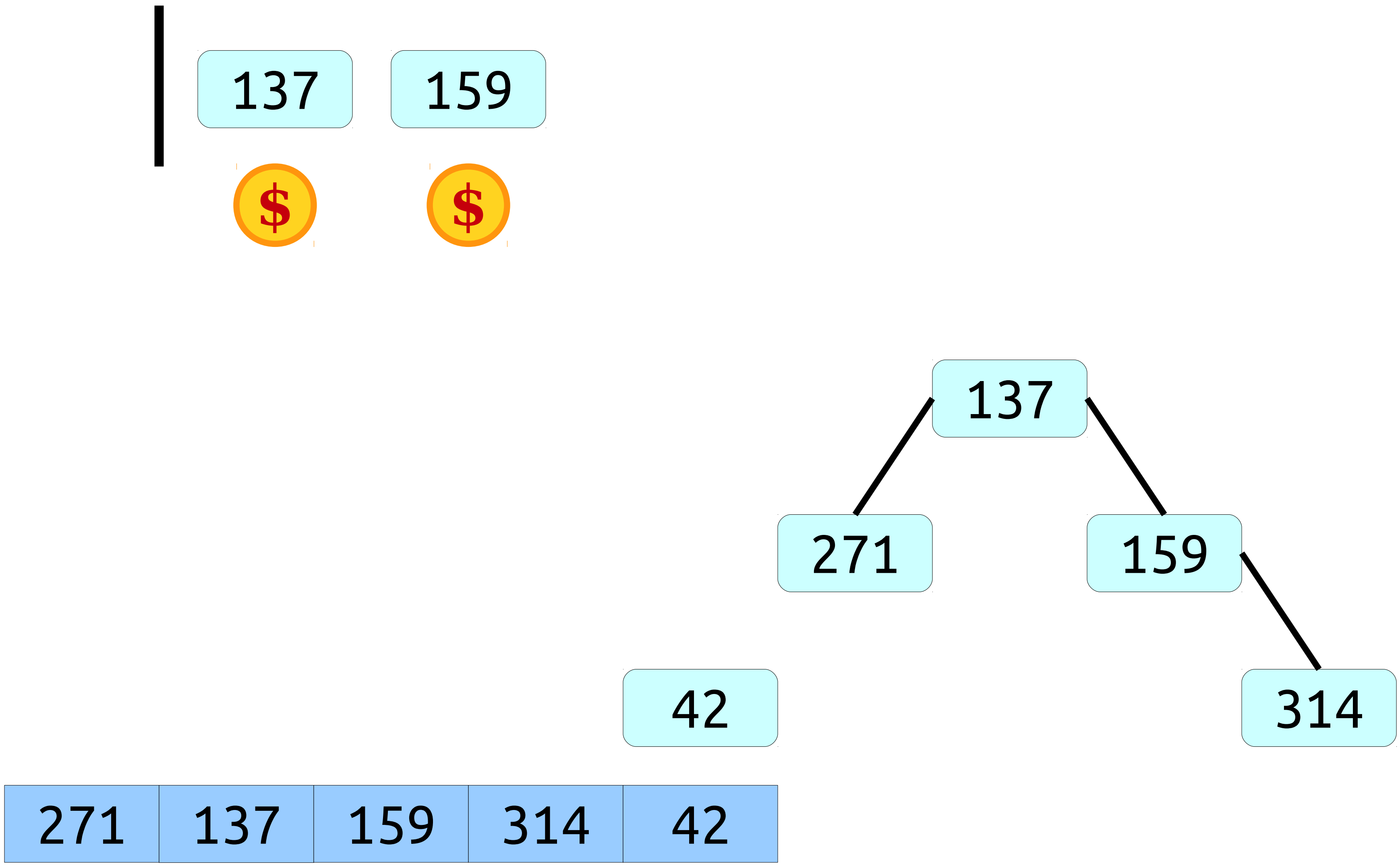


271	137	159	314	42

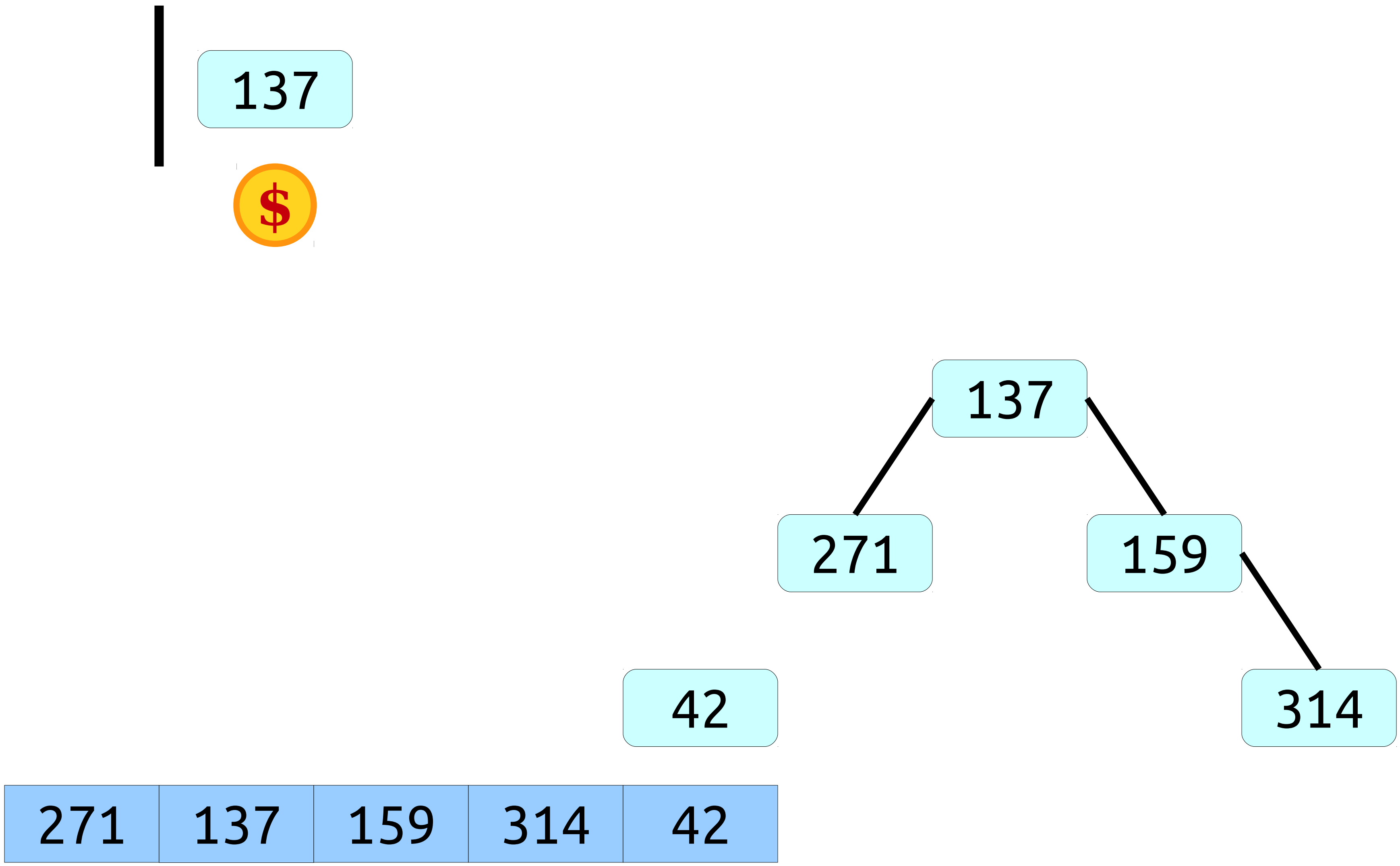
Constructing Cartesian Trees



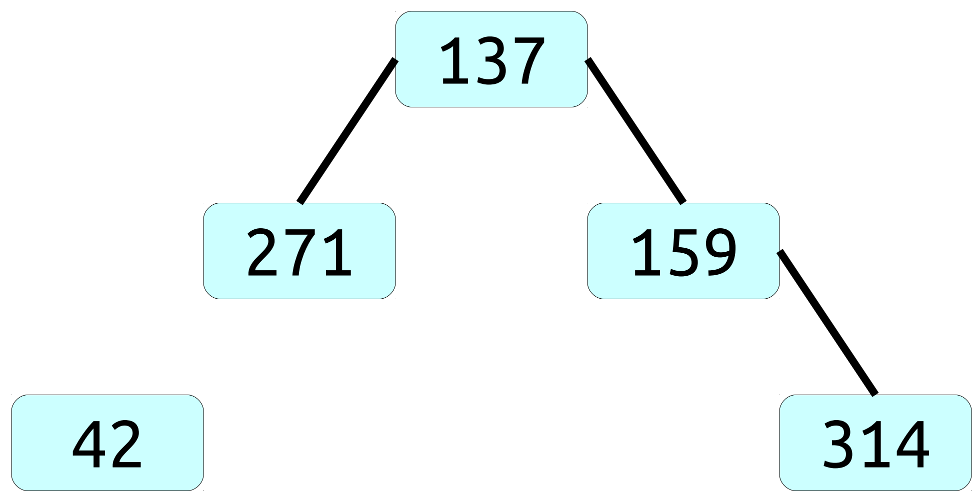
Constructing Cartesian Trees



Constructing Cartesian Trees

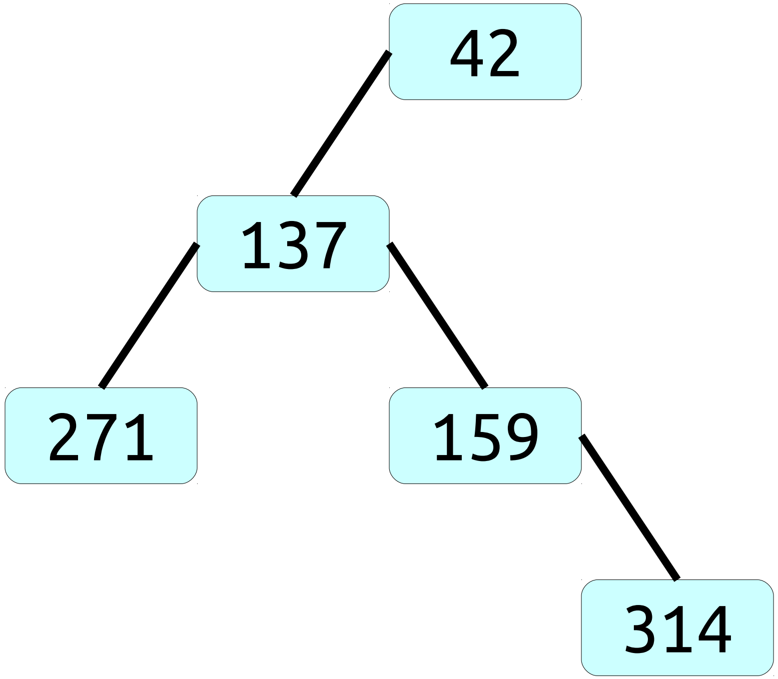


Constructing Cartesian Trees



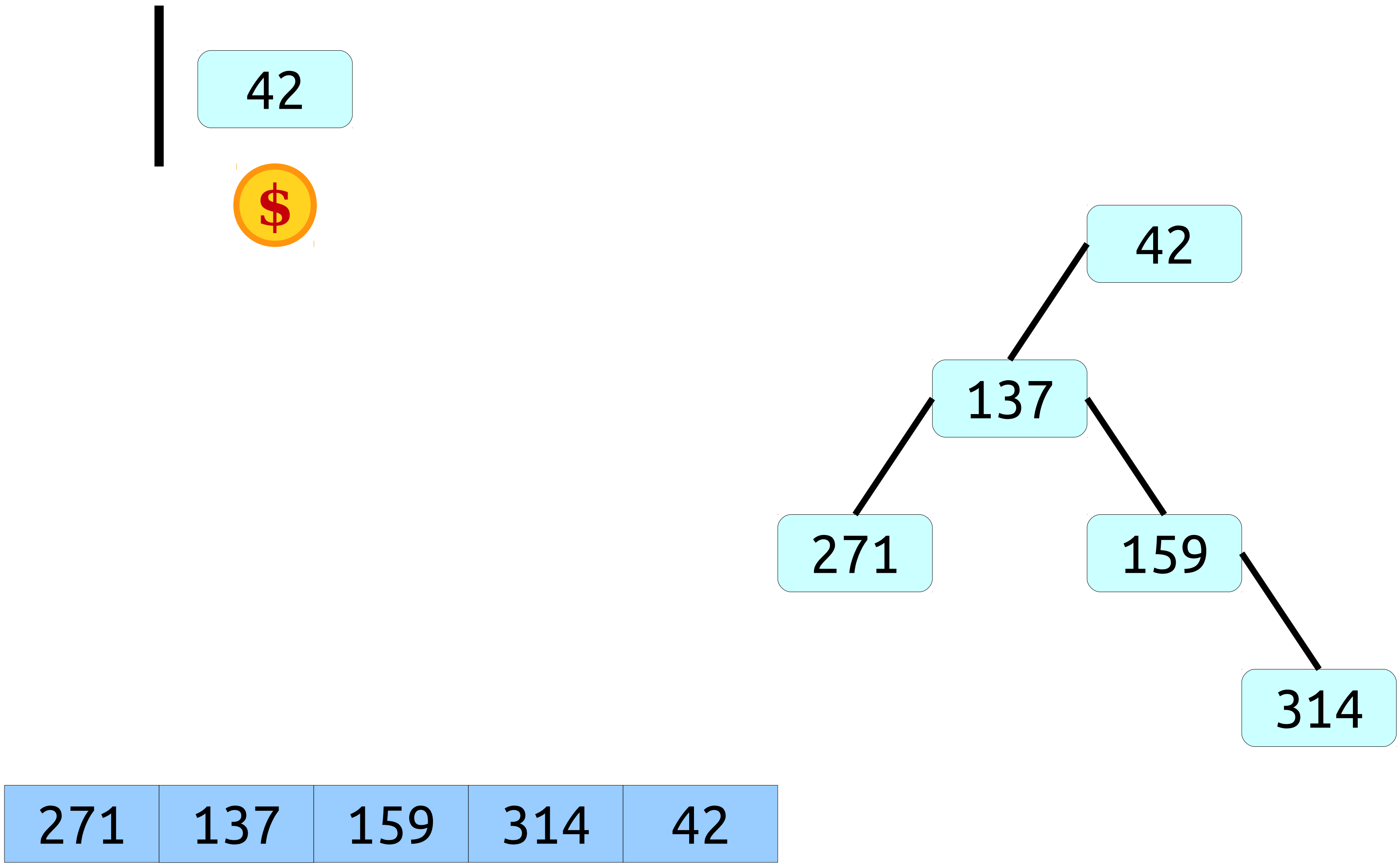
271	137	159	314	42

Constructing Cartesian Trees

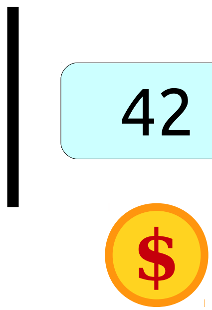


271	137	159	314	42

Constructing Cartesian Trees

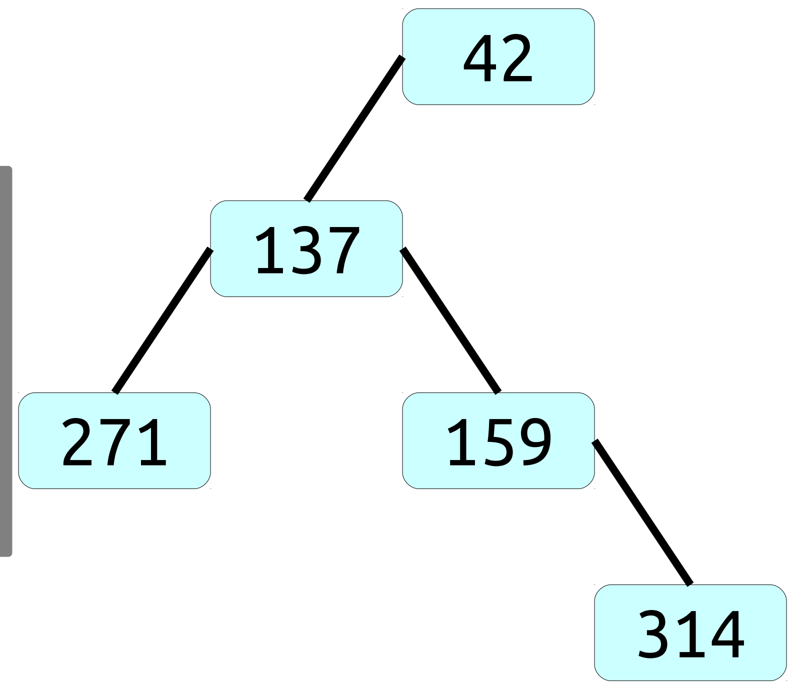


Constructing Cartesian Trees



Work done: 1 push, 3 pops
Credits Removed: $3
Credits Added: $1

Amortized Cost: **2**



271	137	159	314	42

The Banker's Method

- Using the banker's method, the cost of an insertion is

$$t(op) + O(1) \cdot (added_i - removed_i)$$

$$= 1 + k + O(1) \cdot (1 - k)$$

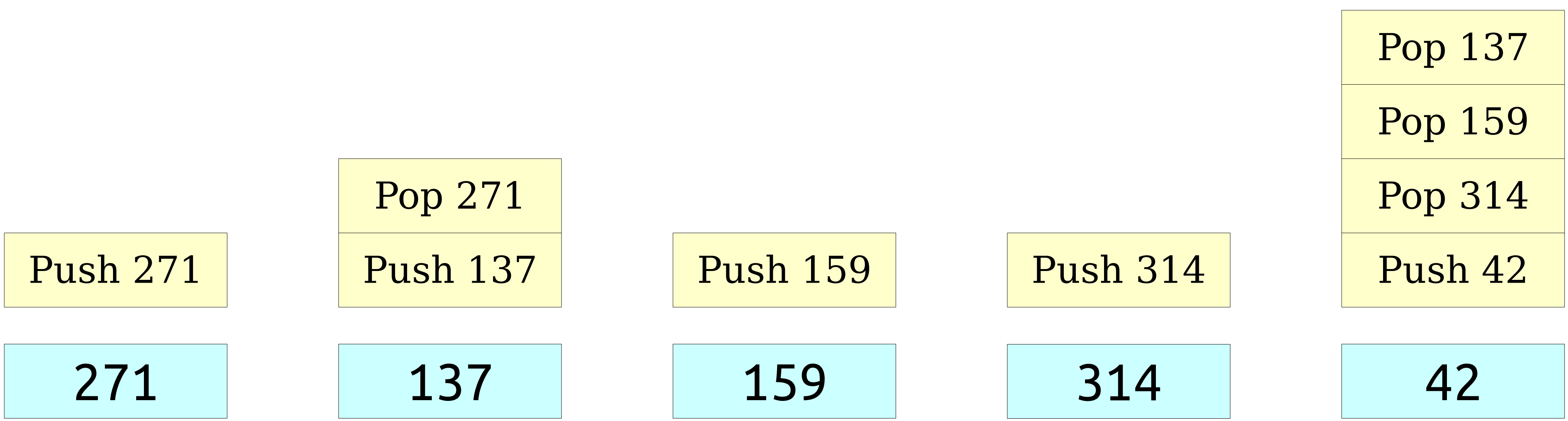
$$= 1 + k + 1 - k$$

$$= 2$$

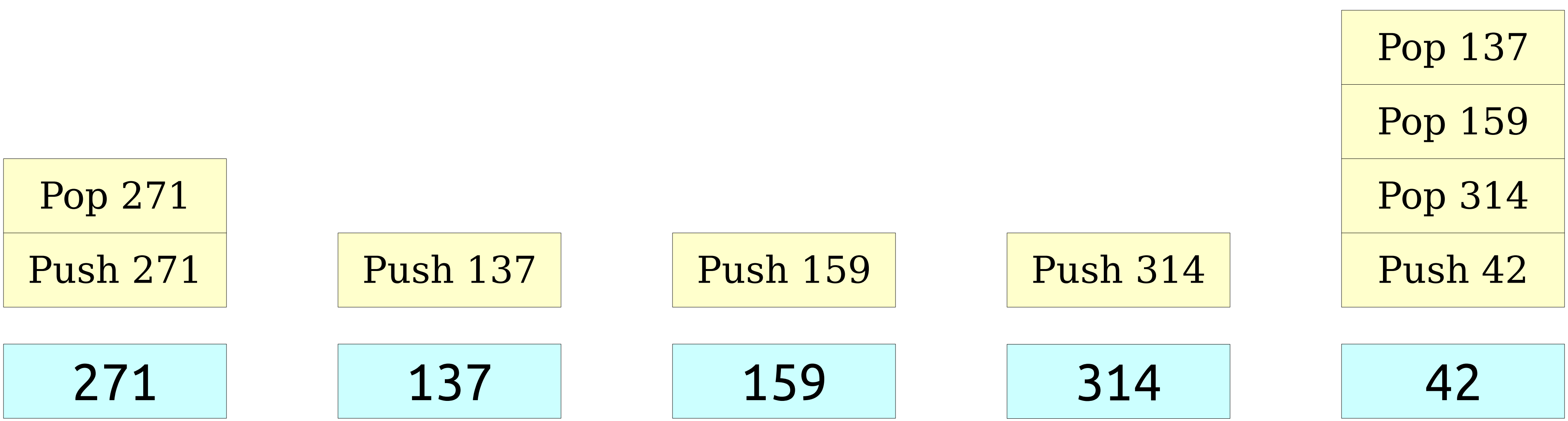
$$= \mathbf{O(1)}$$

- Each insertion has amortized cost O(1).
- Any n insertions will take time $O(n)$.

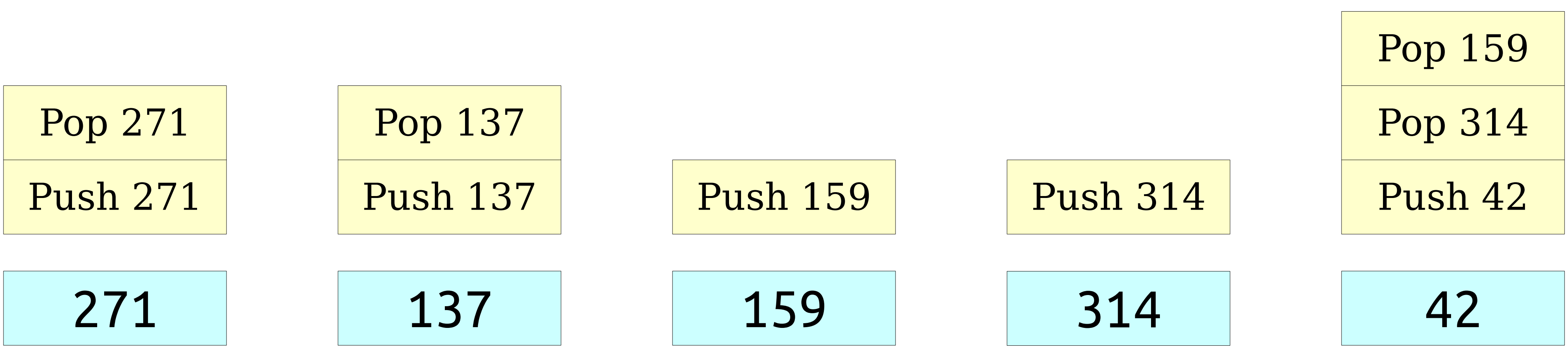
Intuiting the Banker's Method



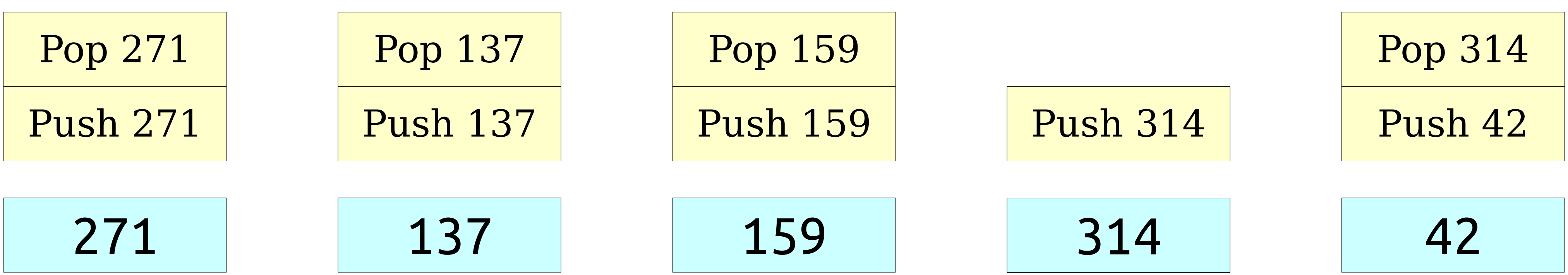
Intuiting the Banker's Method



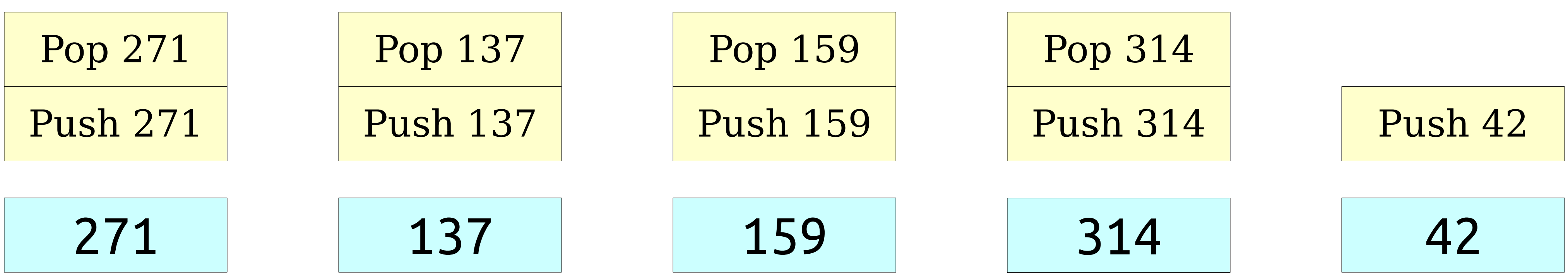
Intuiting the Banker's Method



Intuiting the Banker's Method

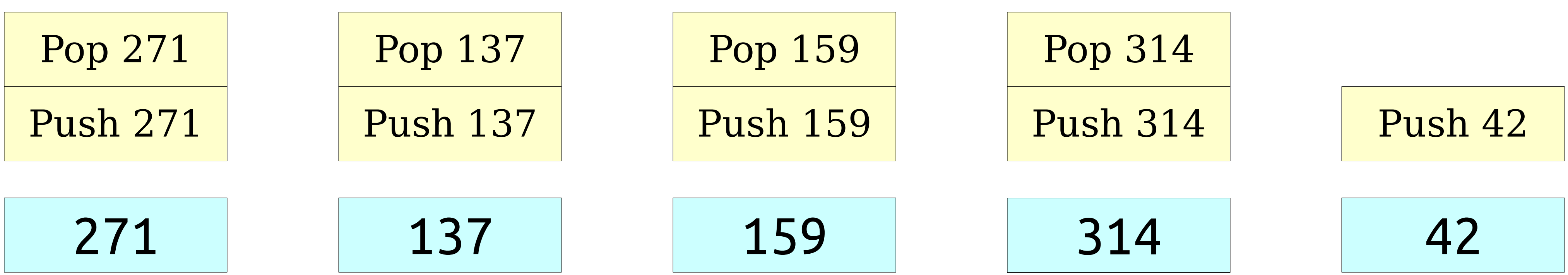


Intuiting the Banker's Method

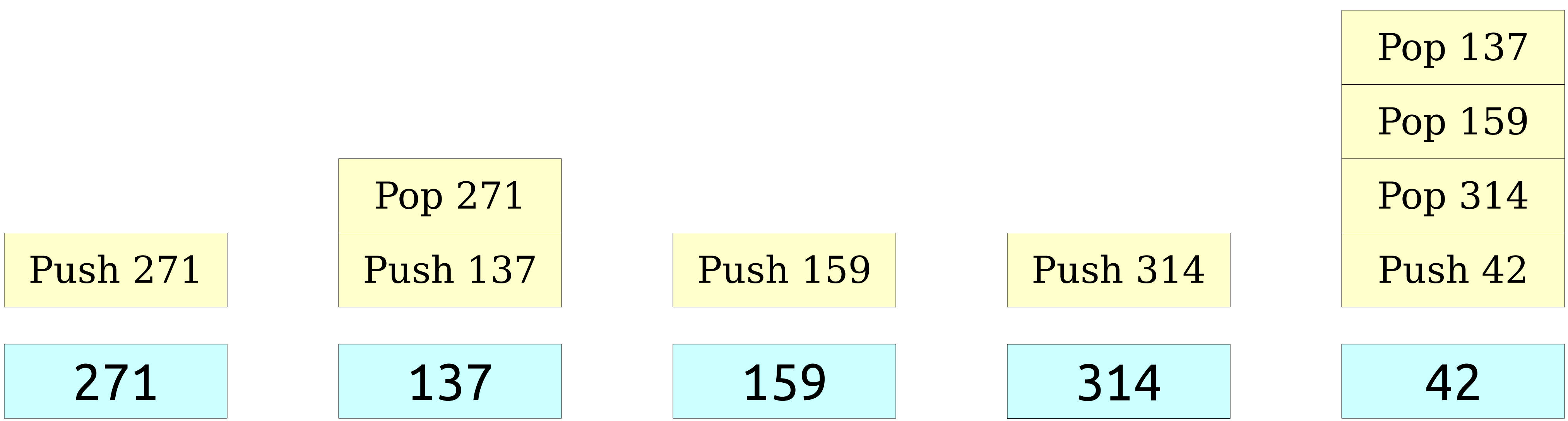


Intuiting the Banker's Method

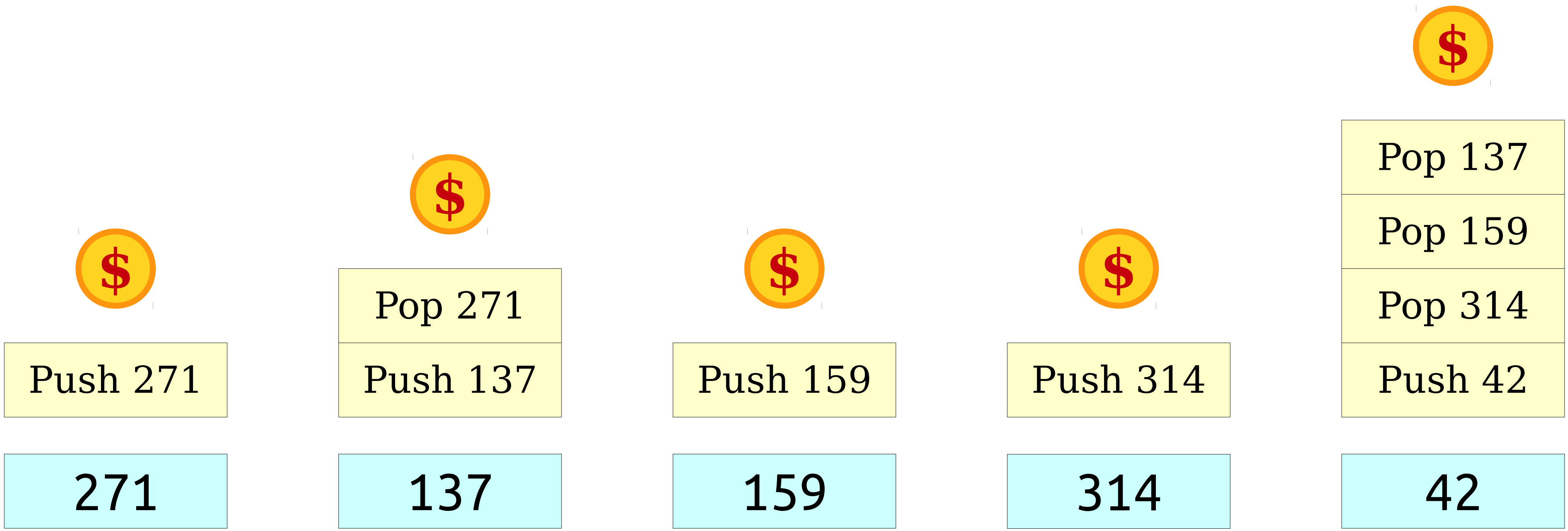
Each operation here is being "charged" for two units of work, even if didn't actually do two units of work.



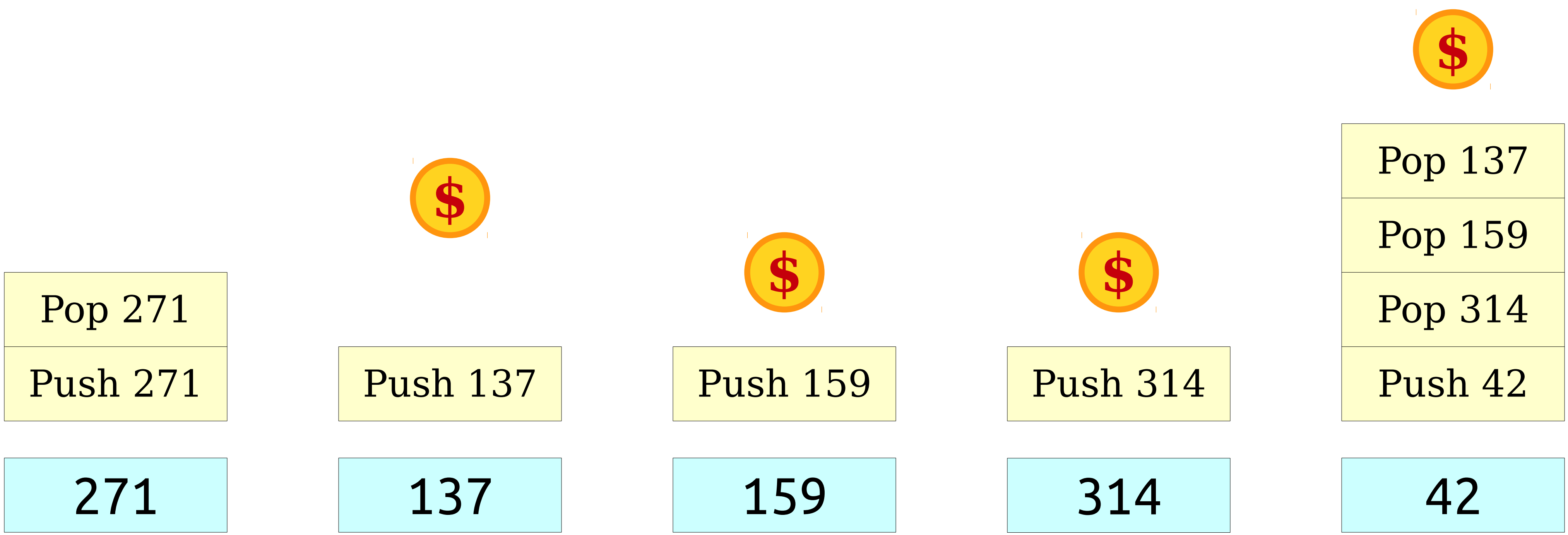
Intuiting the Banker's Method



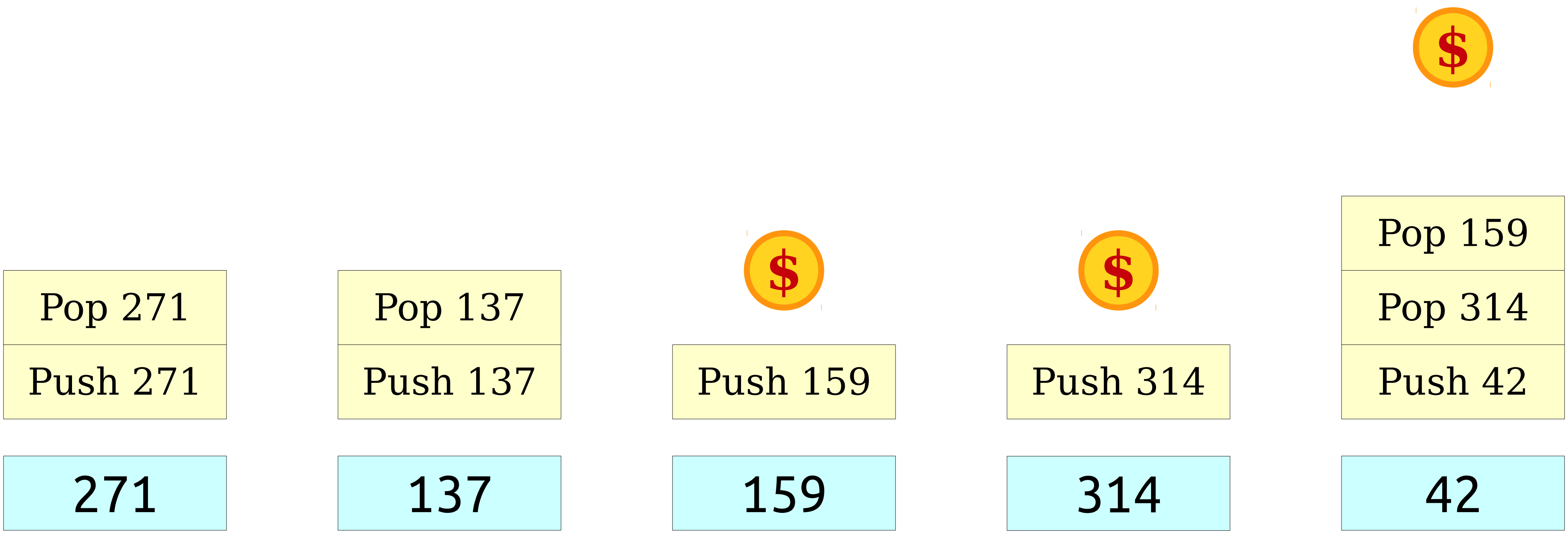
Intuiting the Banker's Method



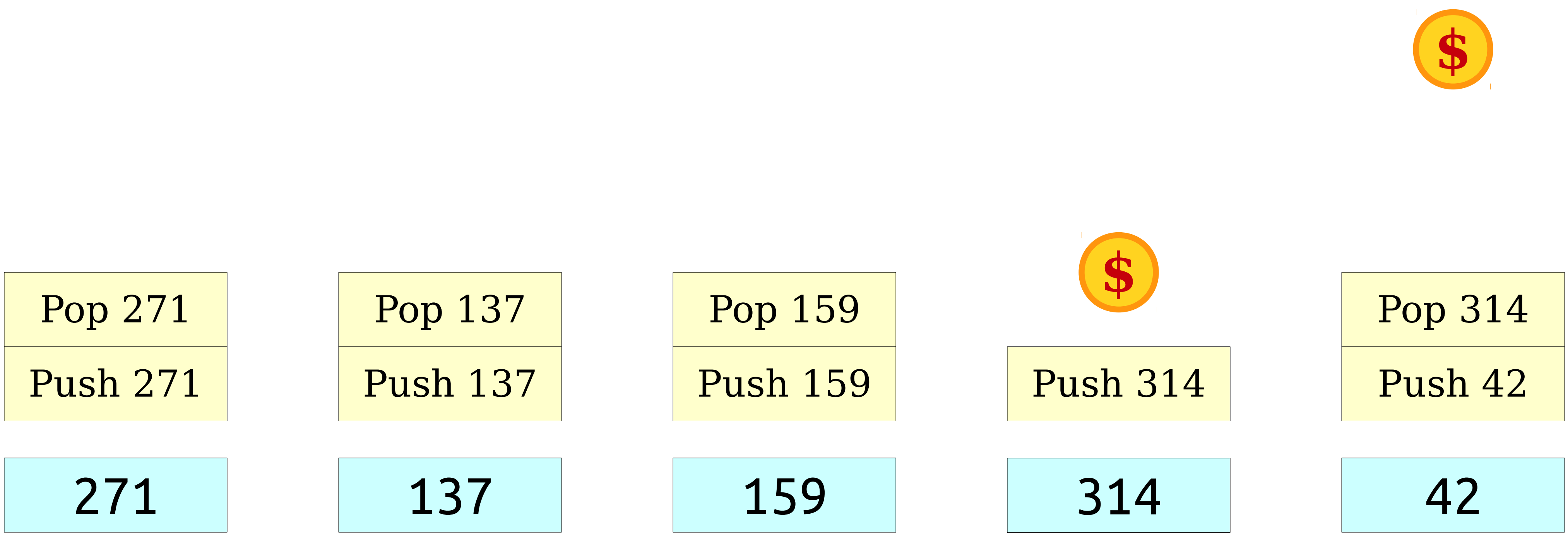
Intuiting the Banker's Method



Intuiting the Banker's Method



Intuiting the Banker's Method



Intuiting the Banker's Method

Pop 271	Pop 137	Pop 159	Pop 314	
Push 271	Push 137	Push 159	Push 314	Push 42
271	137	159	314	42

Intuiting the Banker's Method

Each credit placed can be used to “move” a unit of work from one operation to another.

Pop 271	Pop 137	Pop 159	Pop 314	
Push 271	Push 137	Push 159	Push 314	Push 42
271	137	159	314	42

An Observation

- We defined the amortized cost of an operation to be

$$a(op_i) = t(op_i) + O(1) \cdot (added_i - removed_i)$$

- Equivalently, this is

$$a(op_i) = t(op_i) + O(1) \cdot \Delta credits_i$$

- Some observations:
  - It doesn't matter where these credits are placed or removed from.
  - The total number of credits added and removed doesn't matter; all that matters is the *difference* between these two.

Where We're Going

- There are three standard techniques for assigning amortized costs to operations:
  - The ***aggregate method*** directly assigns each operation its average cost.
  - The ***banker's method*** places credits on the data structure, which are redeemable for units of work.
  - The ***potential method*** assigns a potential function to the data structure, which can be charged to pay for future work or released to pay for recent work.

Where We're Going

There are three standard techniques for assigning amortized costs to operations:

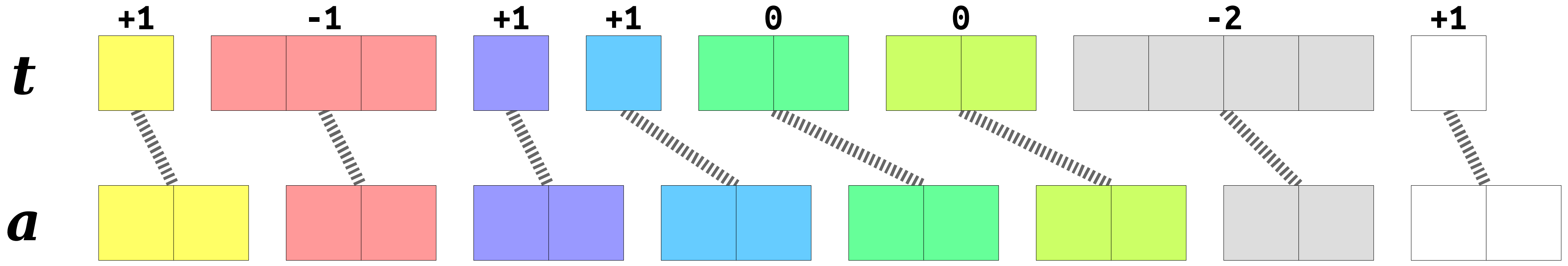
- The ***aggregate method*** directly assigns each operation its average cost.
- The ***banker's method*** places credits on the data structure, which are redeemable for units of work.
- The ***potential method*** assigns a potential function to the data structure, which can be charged to pay for future work or released to pay for recent work.

The Potential Method

- In the ***potential method***, we define a ***potential function*** Φ that maps a data structure to a non-negative real value.
- Each operation may change this potential.
- If we denote by Φ_i the potential of the data structure just before operation i, then we can define $a(op_i)$ as

$$\boldsymbol{a(op_i) = t(op_i) + O(1) \cdot (\Phi_{i+1} - \Phi_i)}$$

- Intuitively, operations that increase the potential have amortized cost greater than their true cost, and operations that decrease the potential have amortized cost less than their true cost.



The Potential Method

$$\sum_{i=1}^{k} a(op_i) \quad = \quad \sum_{i=1}^{k} \left(t(op_i) + O(1) \cdot (\Phi_{i+1} - \Phi_i)\right)$$

The Potential Method

$$\sum_{i=1}^{k} a(op_i) = \sum_{i=1}^{k} \left(t(op_i) + \mathrm{O}(1) \cdot (\Phi_{i+1} - \Phi_i) \right)$$

$$= \sum_{i=1}^{k} t(op_i) + \mathrm{O}(1) \cdot \sum_{i=1}^{k} (\Phi_{i+1} - \Phi_i)$$

The Potential Method

$$
\begin{aligned}
\sum_{i=1}^{k} a(op_i) &= \sum_{i=1}^{k} \left(t(op_i) + \mathrm{O}(1)\cdot(\Phi_{i+1} - \Phi_i)\right) \\
&= \sum_{i=1}^{k} t(op_i) \;+\; \mathrm{O}(1)\cdot\sum_{i=1}^{k} (\Phi_{i+1} - \Phi_i) \\
&= \sum_{i=1}^{k} t(op_i) \;+\; \mathrm{O}(1)\cdot(\Phi_{k+1} - \Phi_1)
\end{aligned}
$$

The Potential Method

$$
\begin{aligned}
\sum_{i=1}^{k} a(op_i) &= \sum_{i=1}^{k} \left(t(op_i) + O(1) \cdot (\Phi_{i+1} - \Phi_i)\right) \\
&= \sum_{i=1}^{k} t(op_i) + O(1) \cdot \sum_{i=1}^{k} (\Phi_{i+1} - \Phi_i) \\
&= \sum_{i=1}^{k} t(op_i) + O(1) \cdot (\Phi_{k+1} - \Phi_1)
\end{aligned}
$$

- Assuming that $\Phi_{k+1} - \Phi_1 \geq 0$, this means that the sum of the amortized costs upper-bounds the sum of the real costs.
- Typically, $\Phi_1 = 0$, so $\Phi_{k+1} - \Phi_1 \geq 0$ holds.

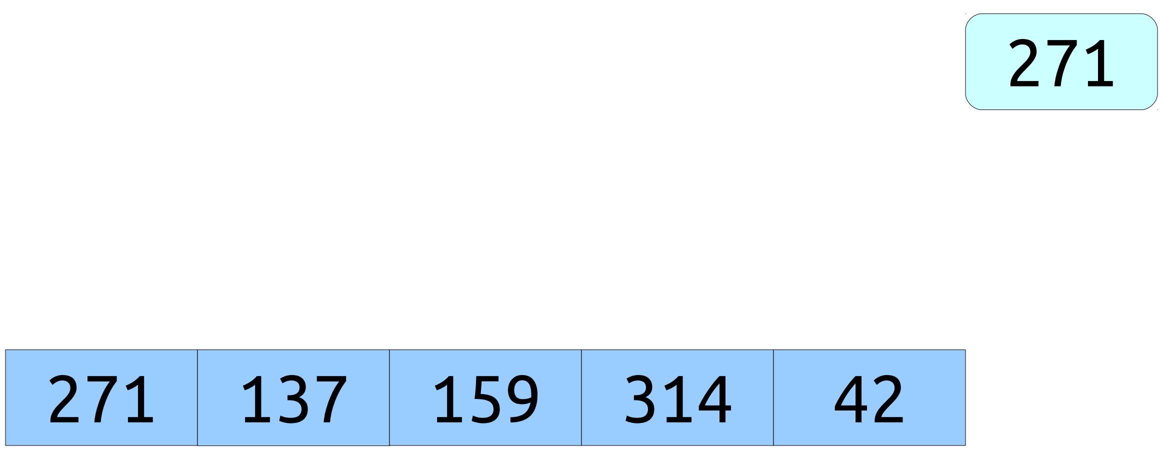
Constructing Cartesian Trees

$\Phi = \mathbf{0}$

271	137	159	314	42

Constructing Cartesian Trees

$\Phi = \mathbf{0}$



Constructing Cartesian Trees

Φ = **1** | 271

271

271	137	159	314	42

Constructing Cartesian Trees

$\Phi = \mathbf{1}$ | 271

Work done: 1 push
$\Delta\Phi$: +1

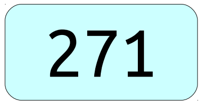
Amortized Cost: **2**

271

271	137	159	314	42

Constructing Cartesian Trees

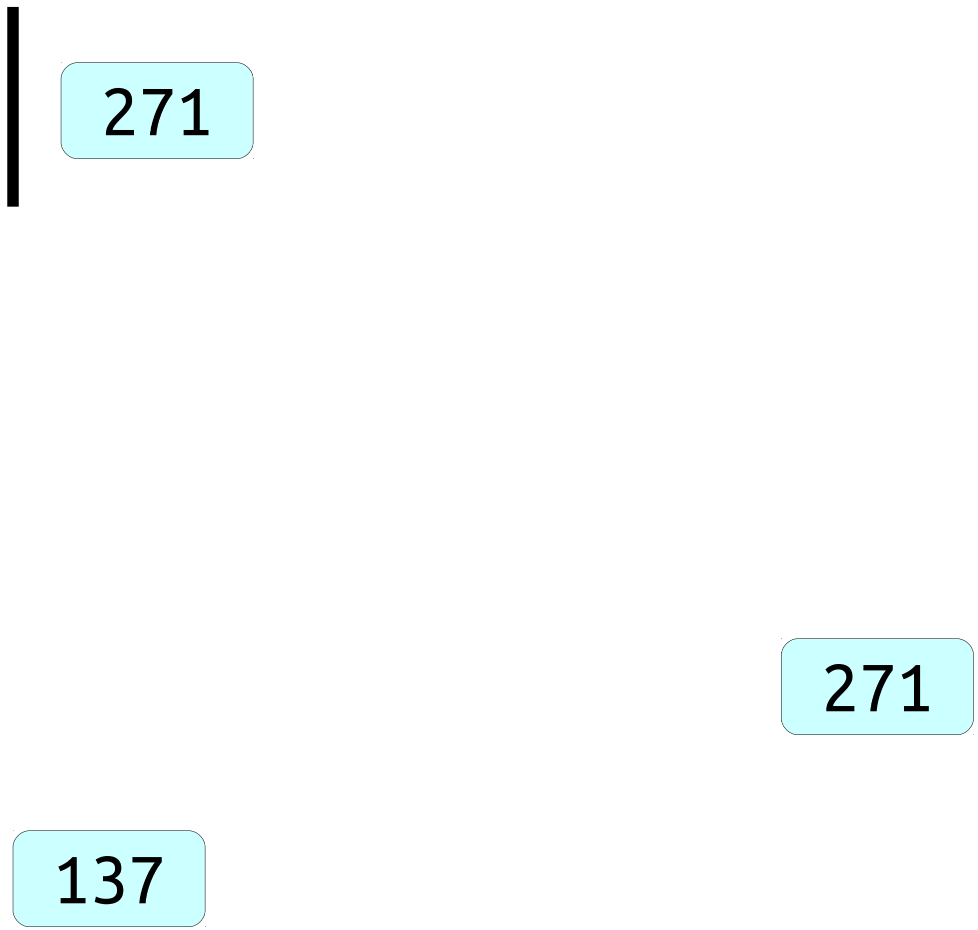
$\Phi = \mathbf{1}$ | 271



271	137	159	314	42

Constructing Cartesian Trees

$\Phi = \mathbf{1}$



271	137	159	314	42

Constructing Cartesian Trees

$\Phi = \mathbf{0}$

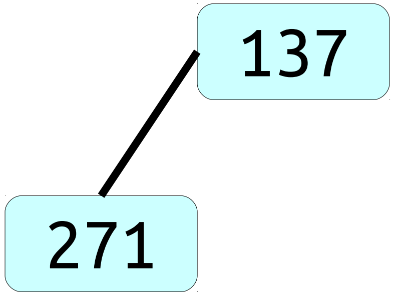
271

137

271	137	159	314	42

Constructing Cartesian Trees

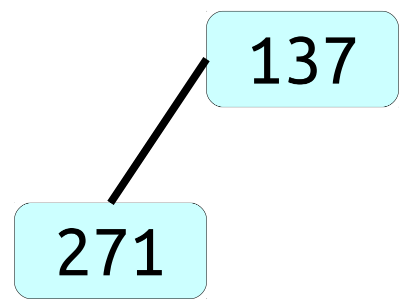
$\Phi = \mathbf{0}$



271	137	159	314	42

Constructing Cartesian Trees

$\Phi = \mathbf{1}$ | 137



271	137	159	314	42

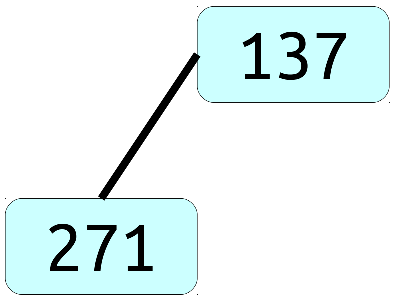
Constructing Cartesian Trees

$\Phi = \mathbf{1}$

137

Work done: 1 push, 1 pop
$\Delta\Phi$: 0

Amortized Cost: **2**



271	137	159	314	42

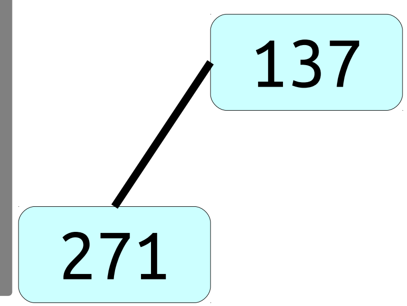
Constructing Cartesian Trees

$\Phi = \mathbf{1}$

137

Work done: 1 push, 1 pop
$\Delta\Phi$: 0

Amortized Cost: **2**



271	137	159	314	42

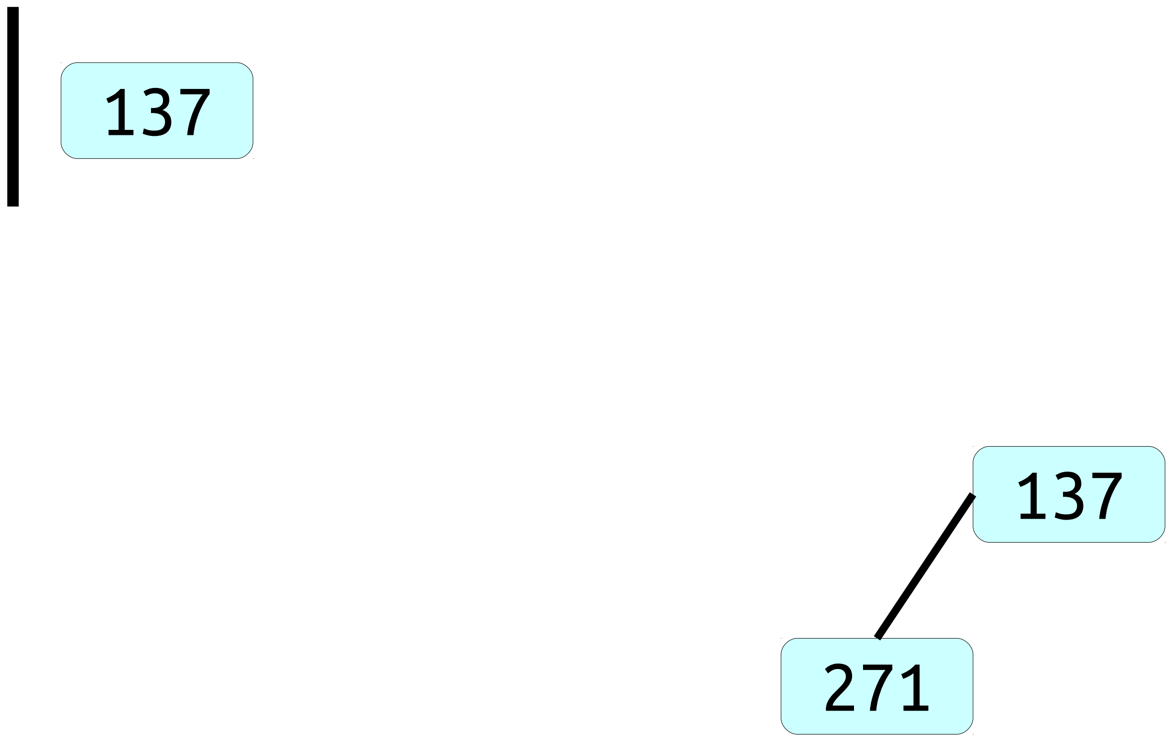
Notice that Φ went

$1 \rightarrow 0 \rightarrow 1$

All that matters is the ***net*** change.

Constructing Cartesian Trees

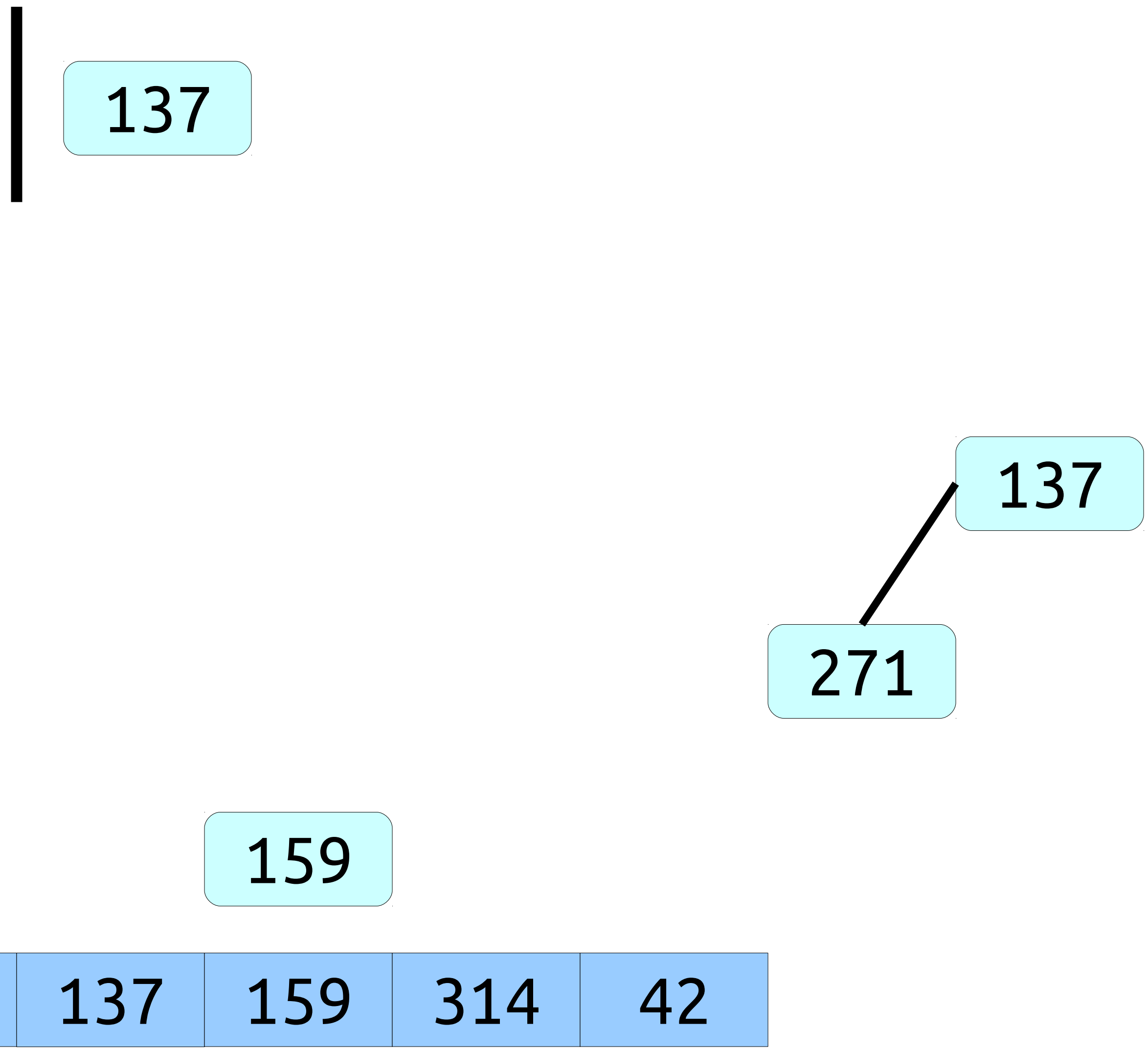
$\Phi = \mathbf{1}$



271	137	159	314	42

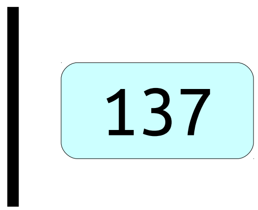
Constructing Cartesian Trees

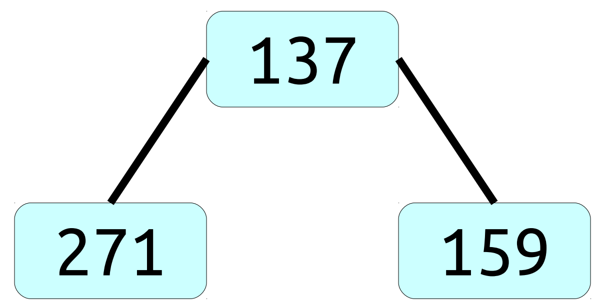
$\Phi = \mathbf{1}$



Constructing Cartesian Trees

$\Phi = \mathbf{1}$

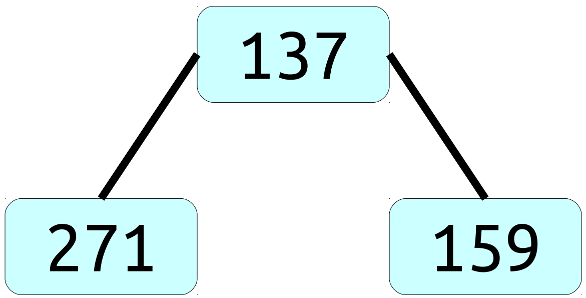




271	137	159	314	42

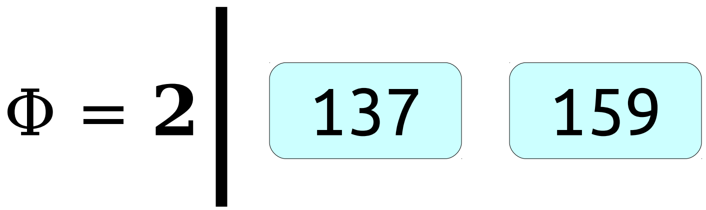
Constructing Cartesian Trees

$\Phi = \mathbf{2}$ | 137 159



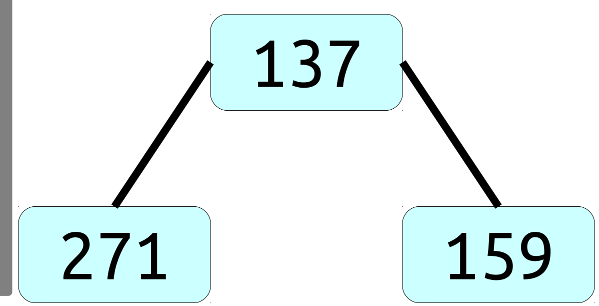
271	137	159	314	42

Constructing Cartesian Trees



Work done: 1 push
ΔΦ: +1

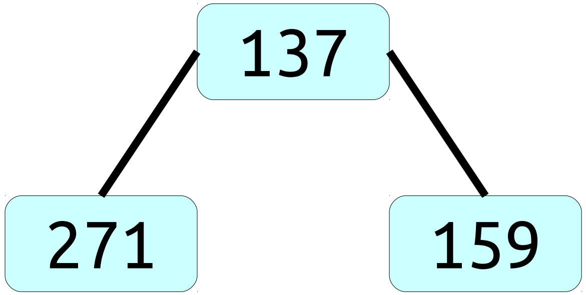
Amortized Cost: **2**



271	137	159	314	42

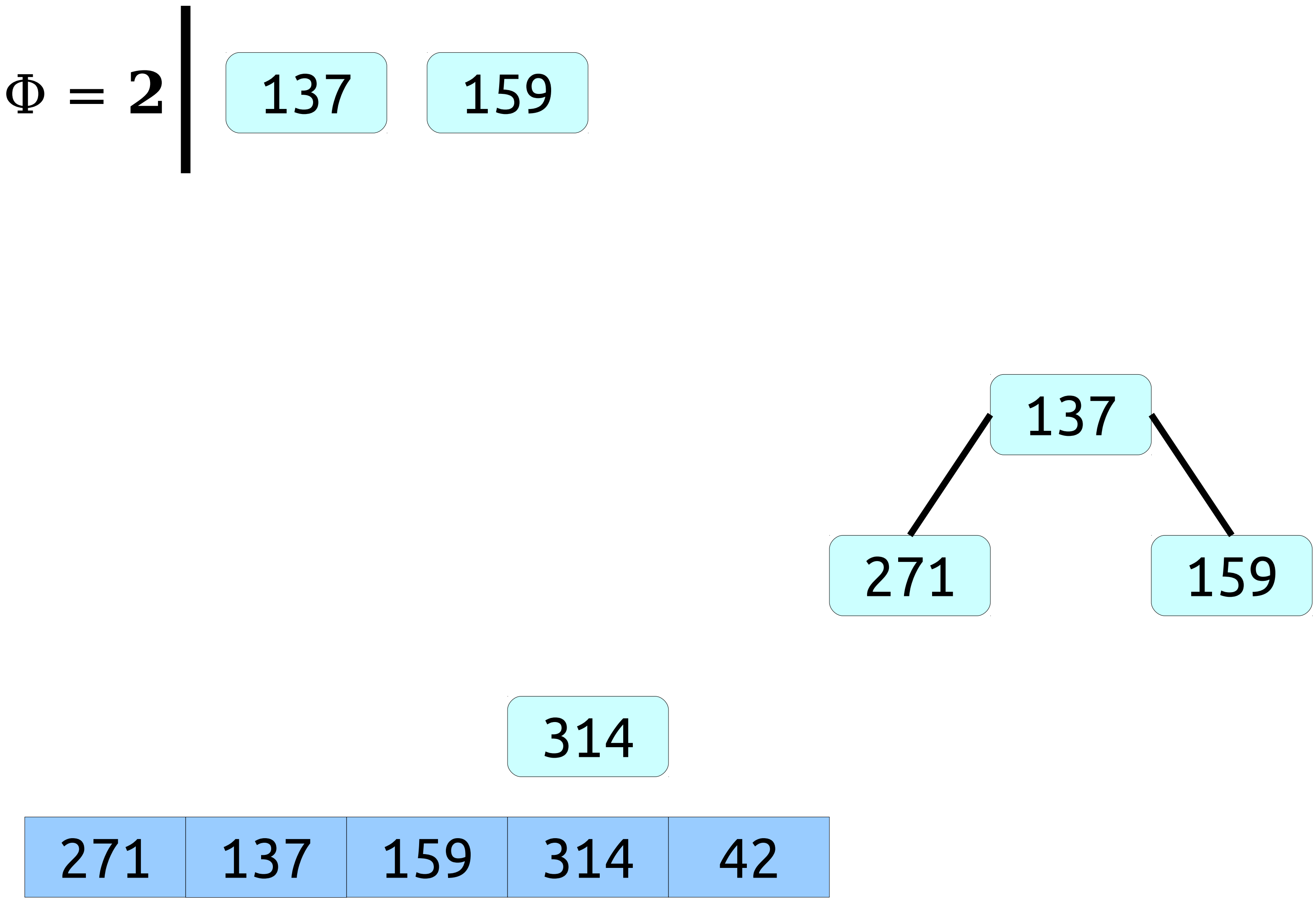
Constructing Cartesian Trees

$\Phi = \mathbf{2}$ | 137 159



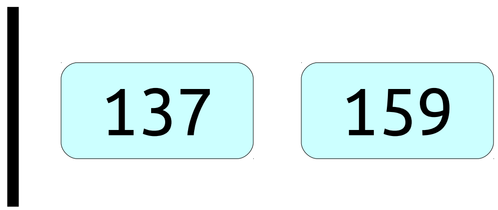
271	137	159	314	42

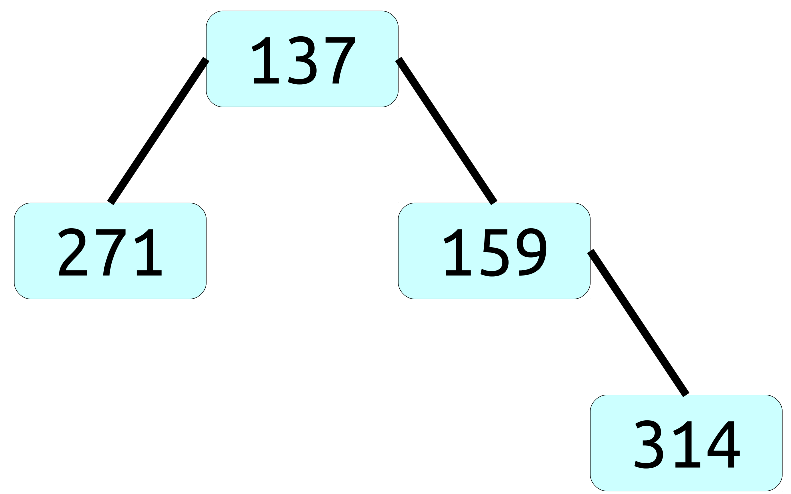
Constructing Cartesian Trees



Constructing Cartesian Trees

$\Phi = \mathbf{2}$

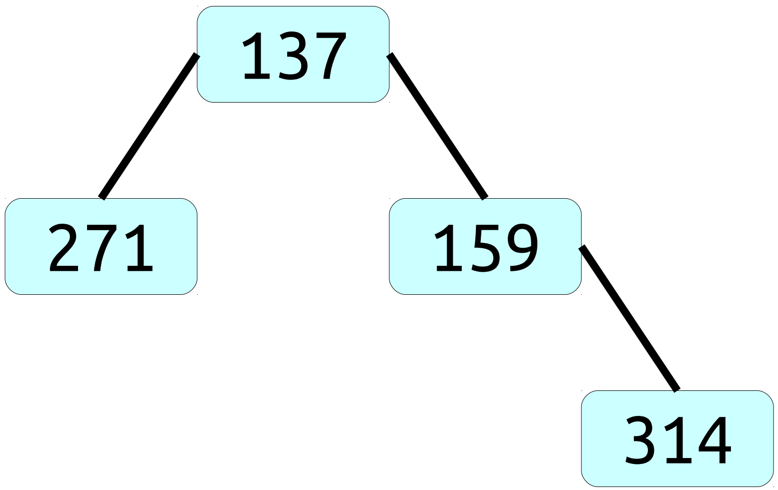




271	137	159	314	42

Constructing Cartesian Trees

$\Phi = \mathbf{3}$ | 137 159 314



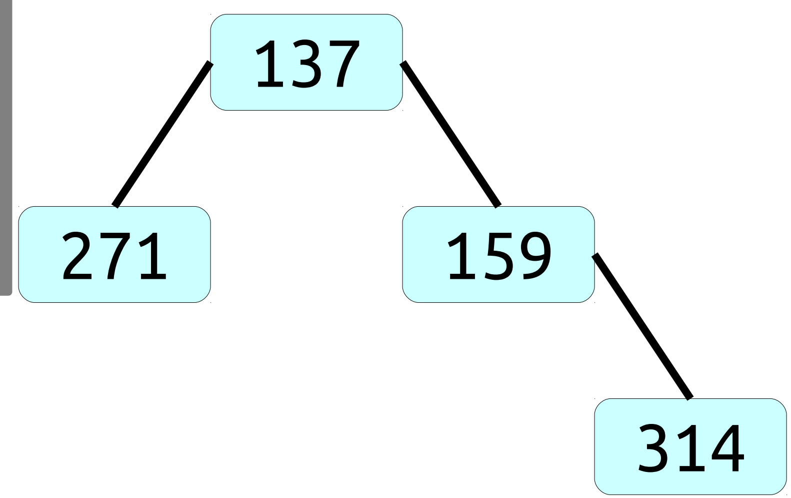
271	137	159	314	42

Constructing Cartesian Trees

Φ = **3** | 137 | 159 | 314

Work done: 1 push
Credits Added: $\Delta\Phi$: +1

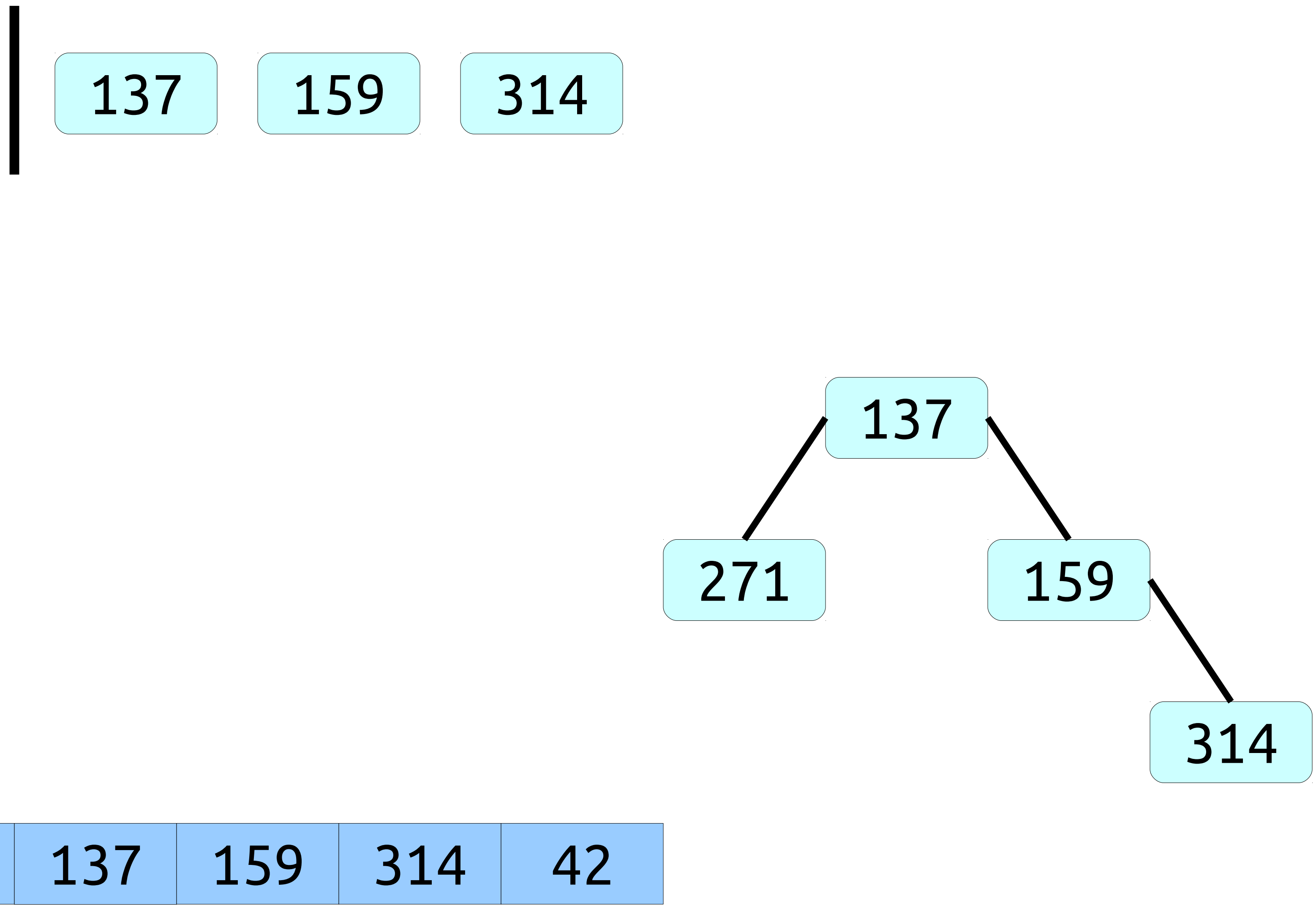
Amortized Cost: **2**



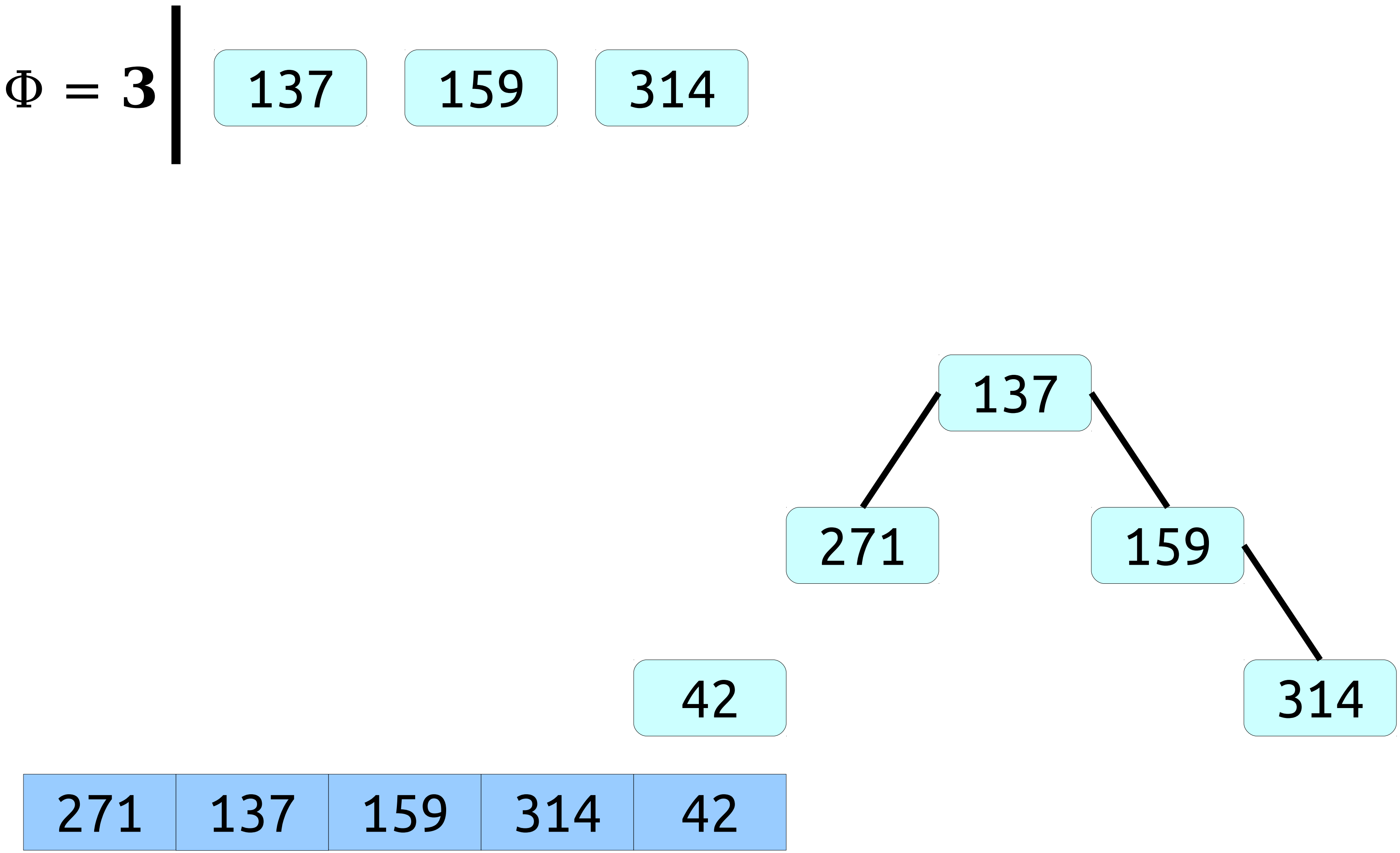
271	137	159	314	42

Constructing Cartesian Trees

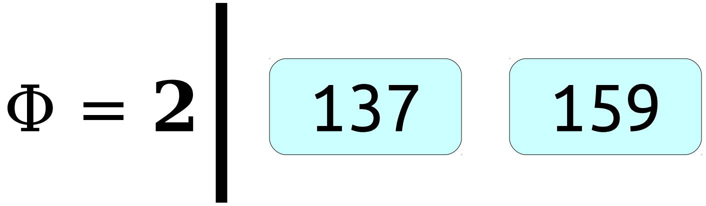
$\Phi = \mathbf{3}$

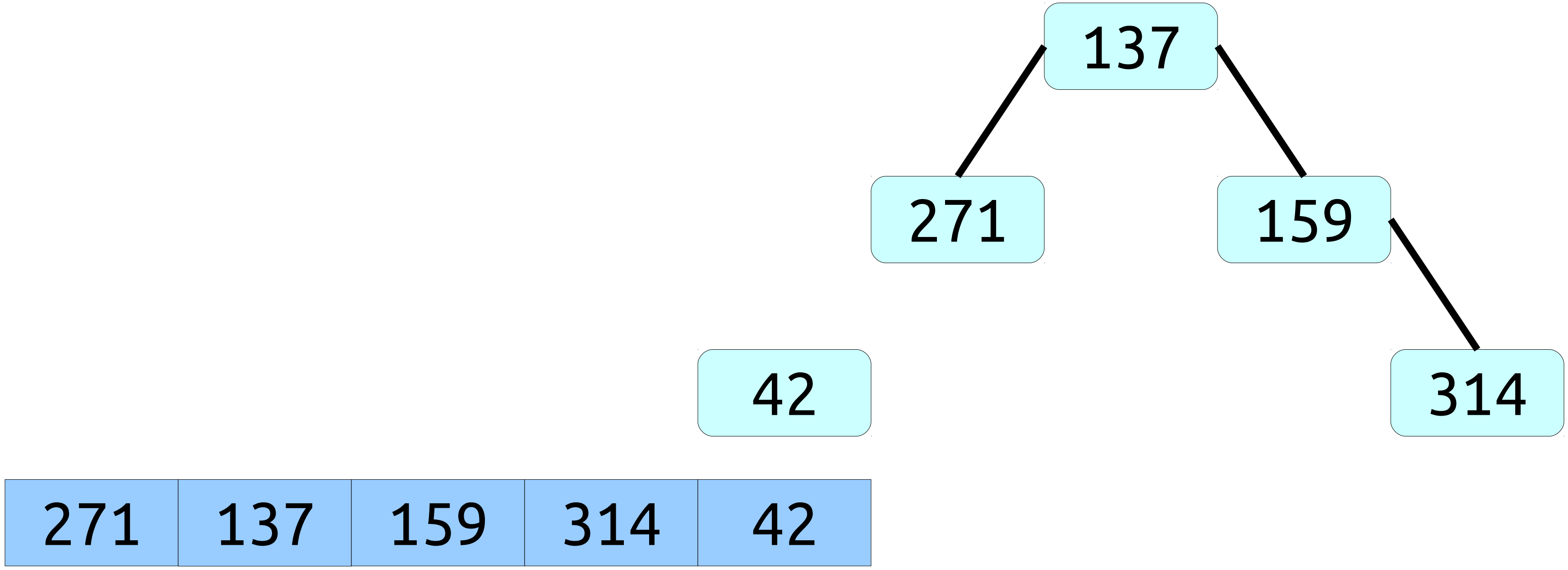


Constructing Cartesian Trees



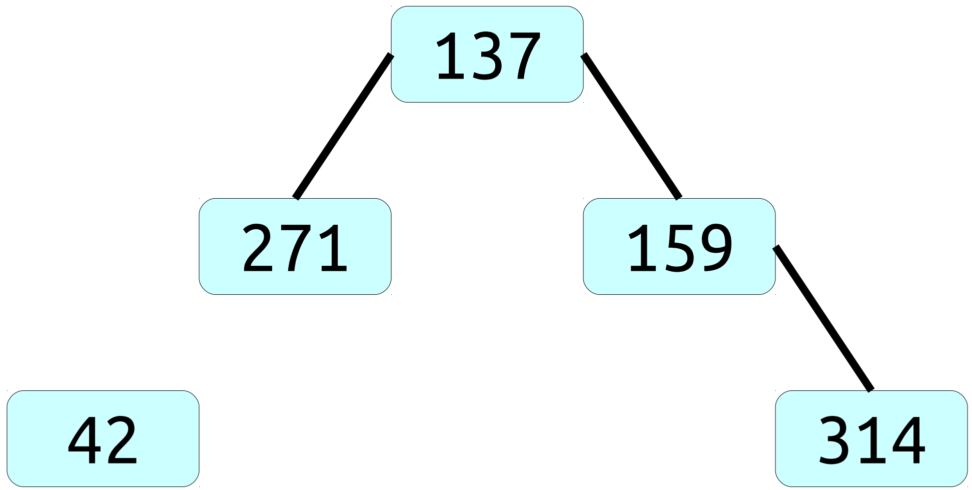
Constructing Cartesian Trees





Constructing Cartesian Trees

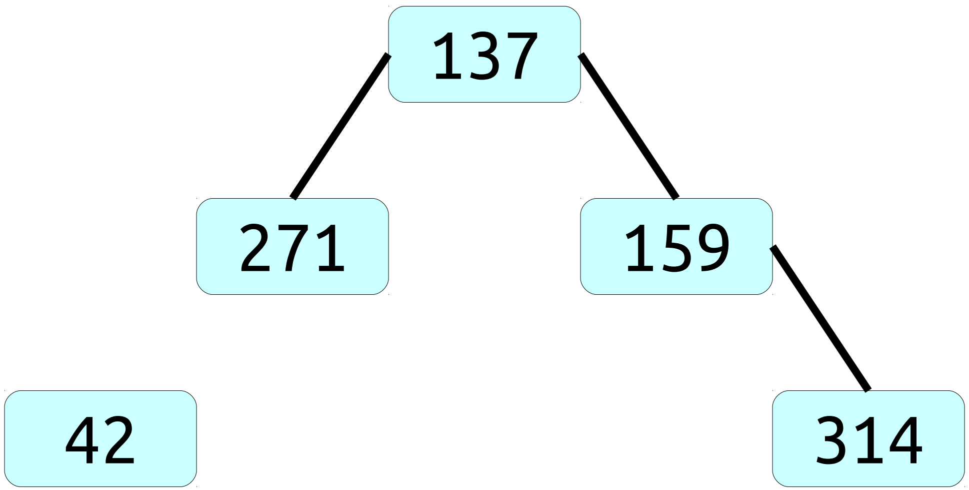
$\Phi = \mathbf{1}$ | 137



271	137	159	314	42

Constructing Cartesian Trees

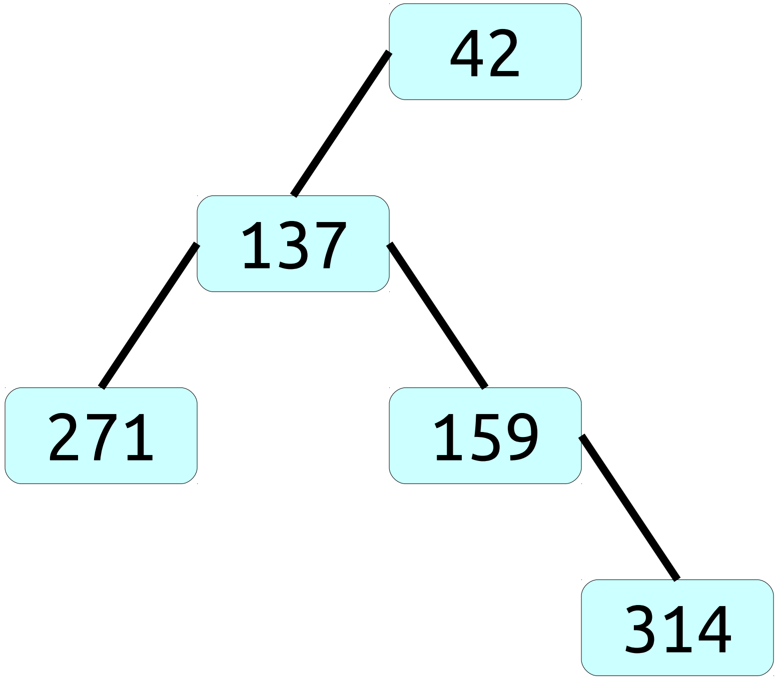
$\Phi = \mathbf{0}$



271	137	159	314	42

Constructing Cartesian Trees

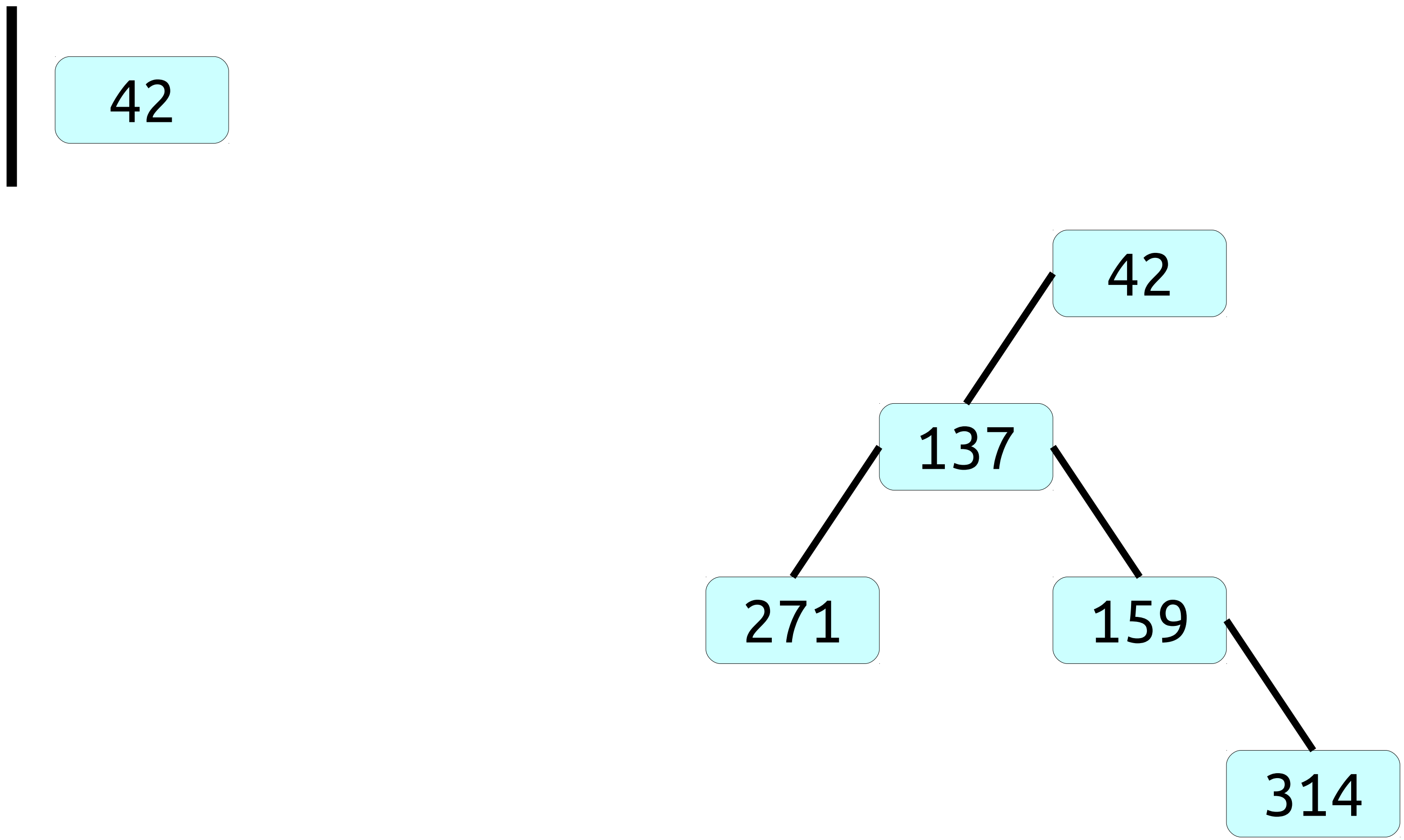
$\Phi = \mathbf{0}$



271	137	159	314	42

Constructing Cartesian Trees

$\Phi = \mathbf{1}$



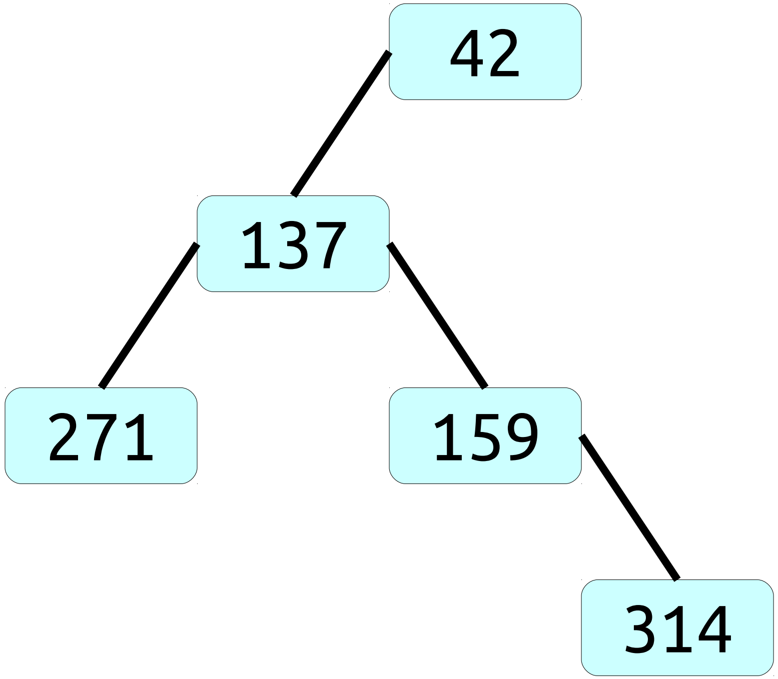
271	137	159	314	42

Constructing Cartesian Trees

$\Phi = \mathbf{1}$ | 42

Work done: 1 push, 3 pops
ΔΦ: -2

Amortized Cost: **2**



271	137	159	314	42

The Potential Method

- Using the potential method, the cost of an insertion into a Cartesian tree can be computed as

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

- So the amortized cost of an insertion is O(1).
- Therefore, n total insertions takes time O(n).

Amortization in Practice:
The Two-Stack Queue

The Two-Stack Queue

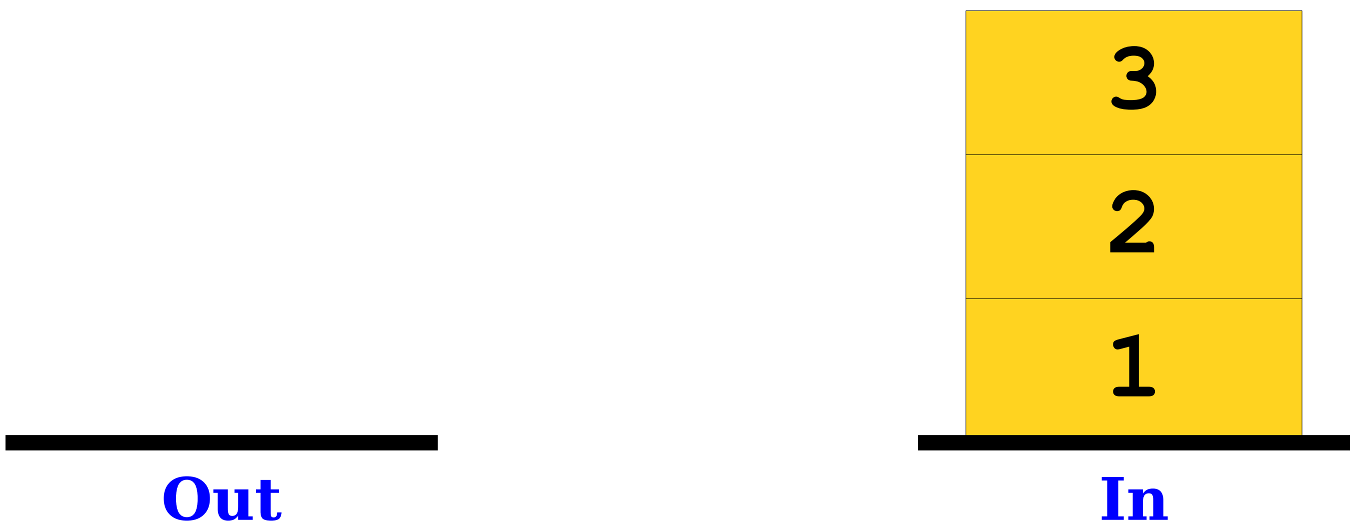
Out

In

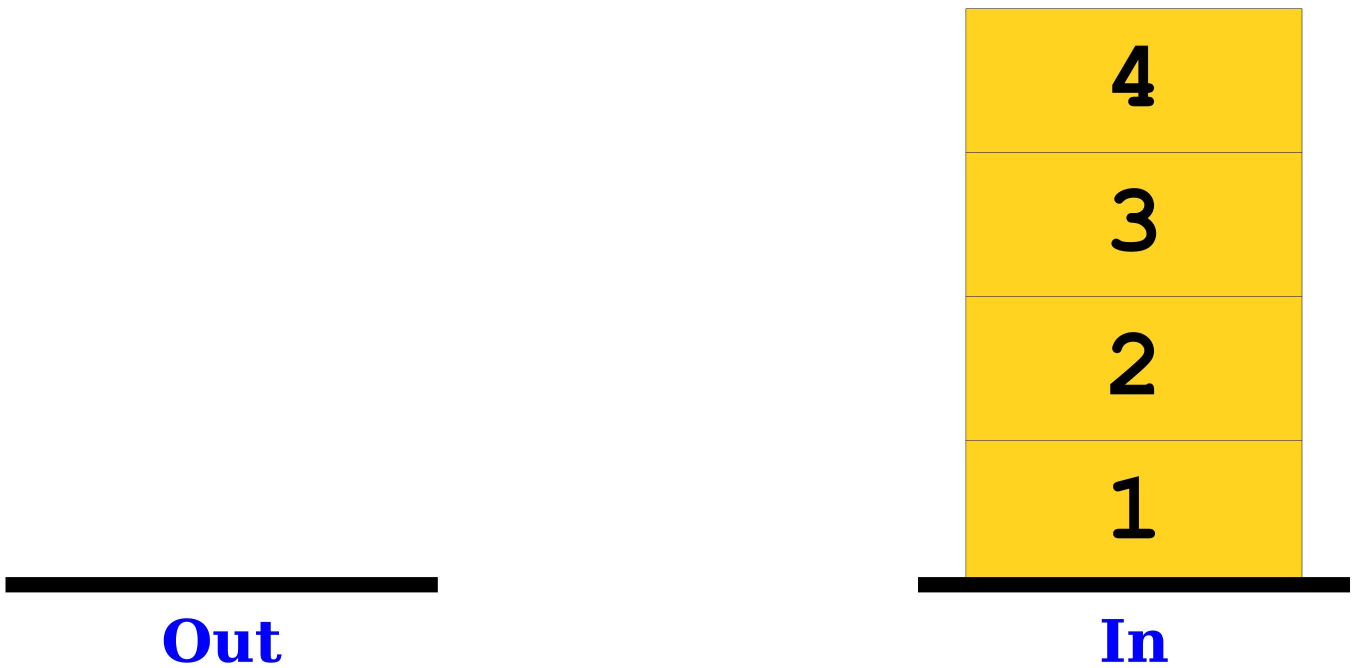
The Two-Stack Queue

The Two-Stack Queue

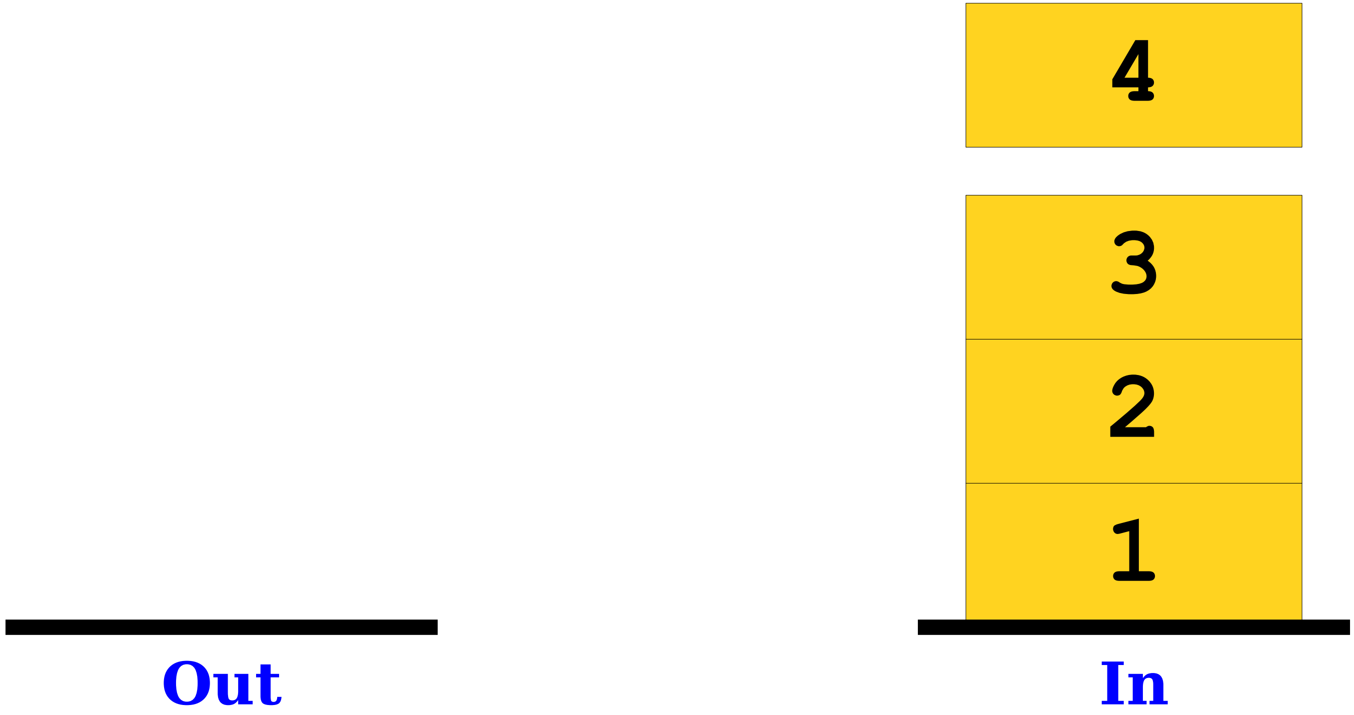
The Two-Stack Queue



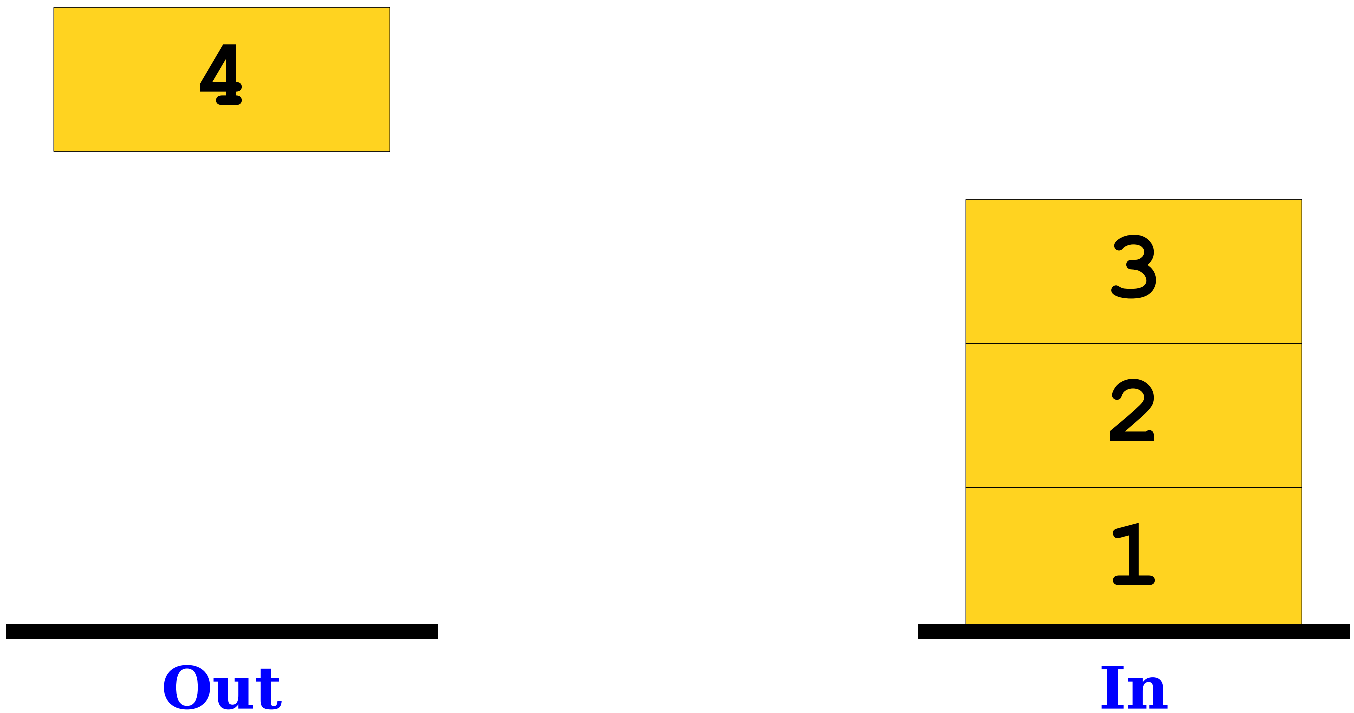
The Two-Stack Queue



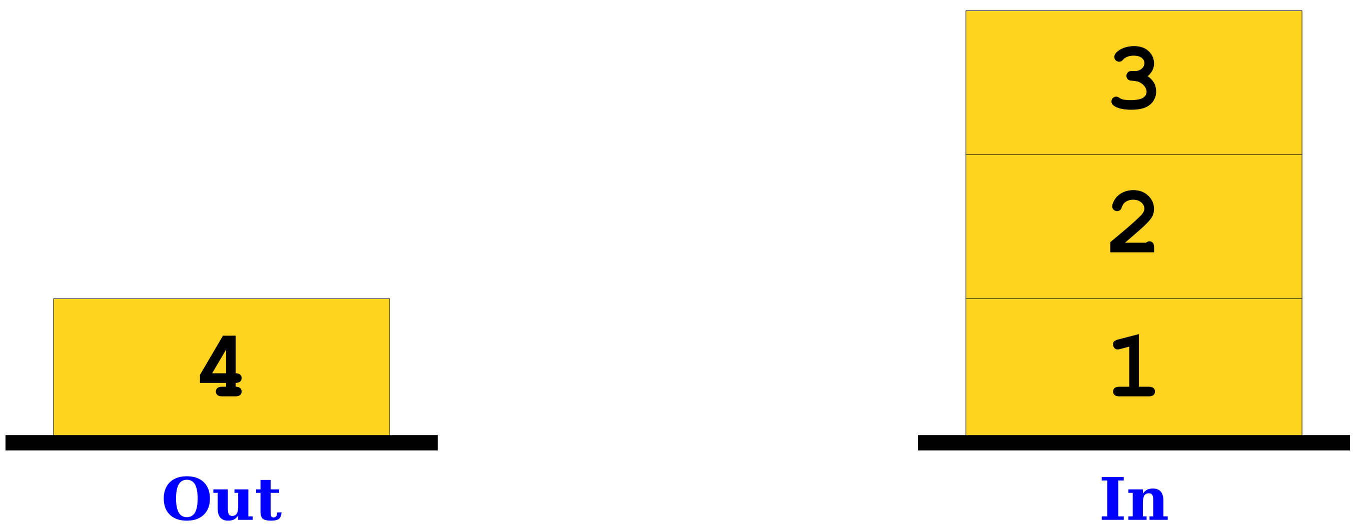
The Two-Stack Queue



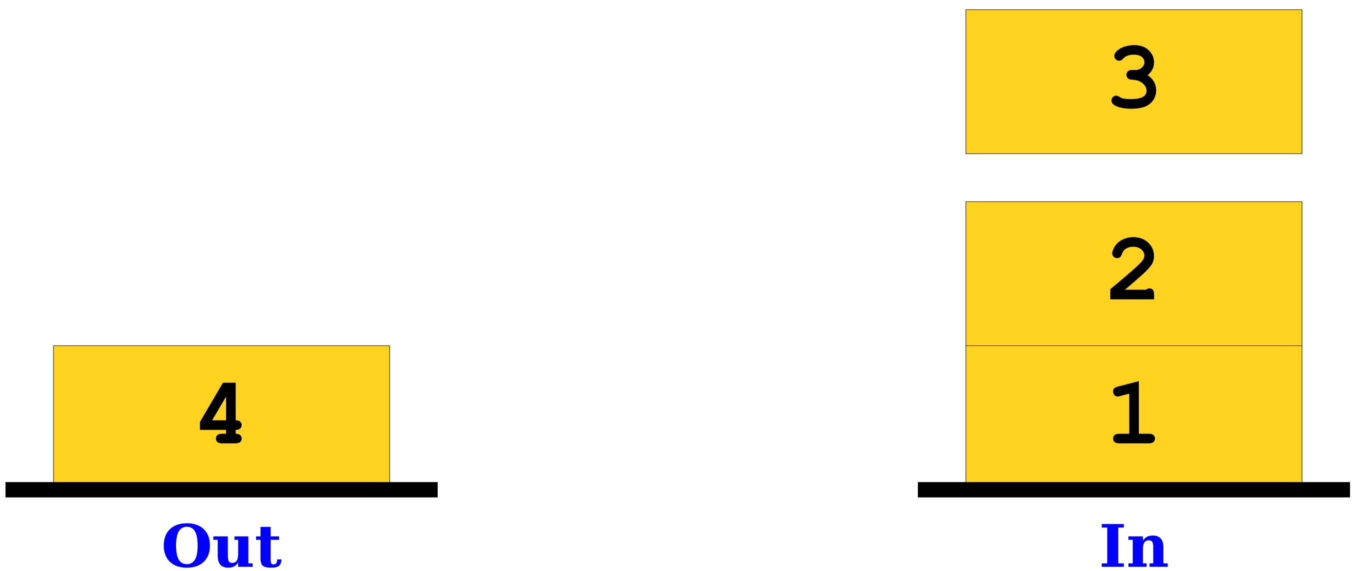
The Two-Stack Queue



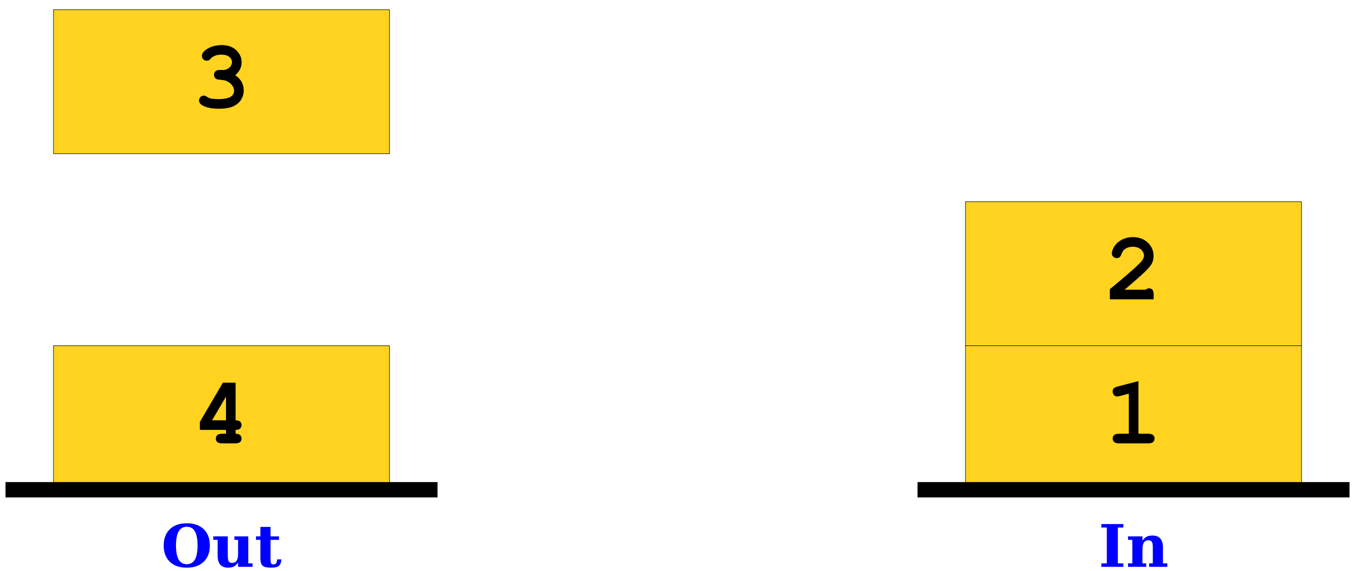
The Two-Stack Queue



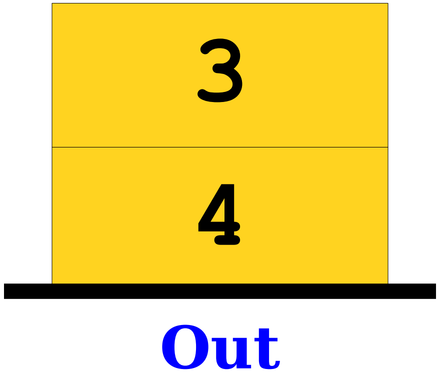
The Two-Stack Queue



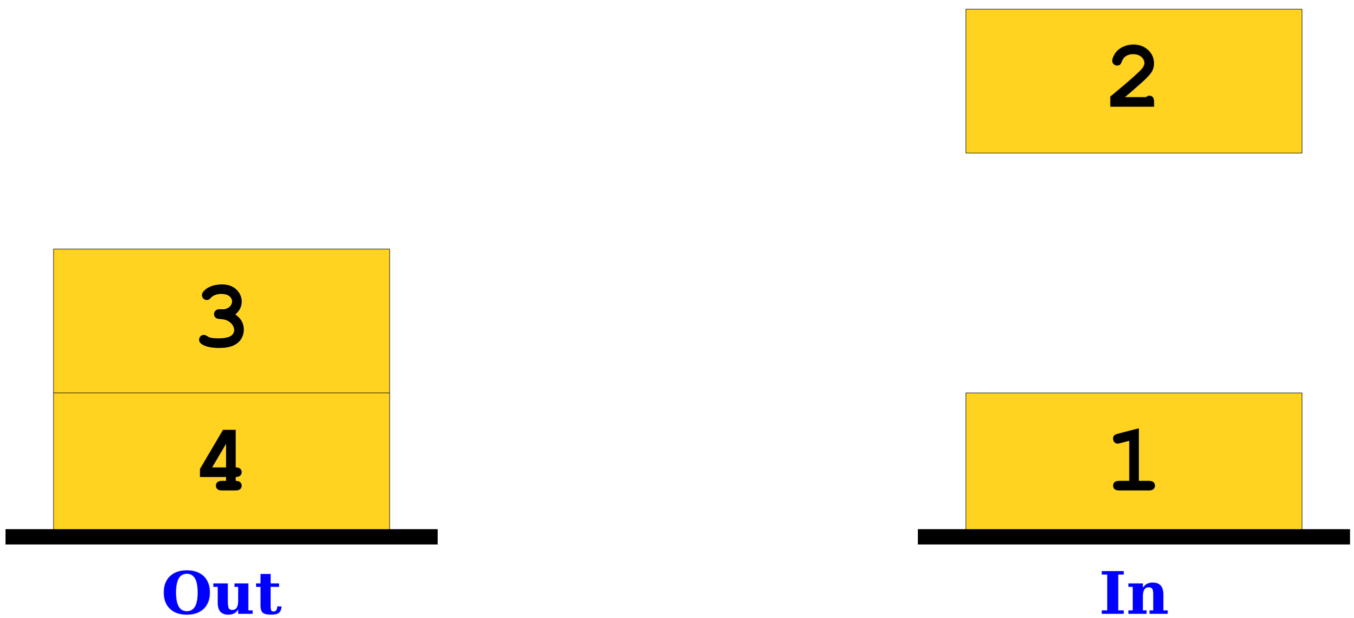
The Two-Stack Queue



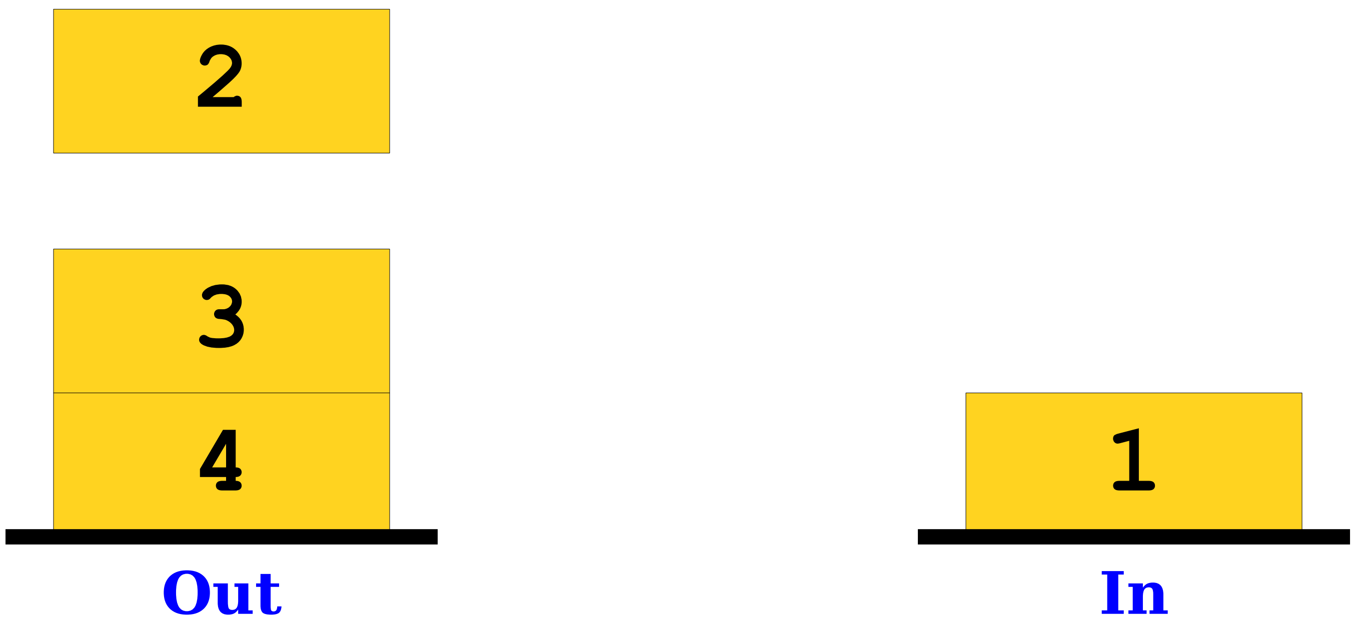
The Two-Stack Queue



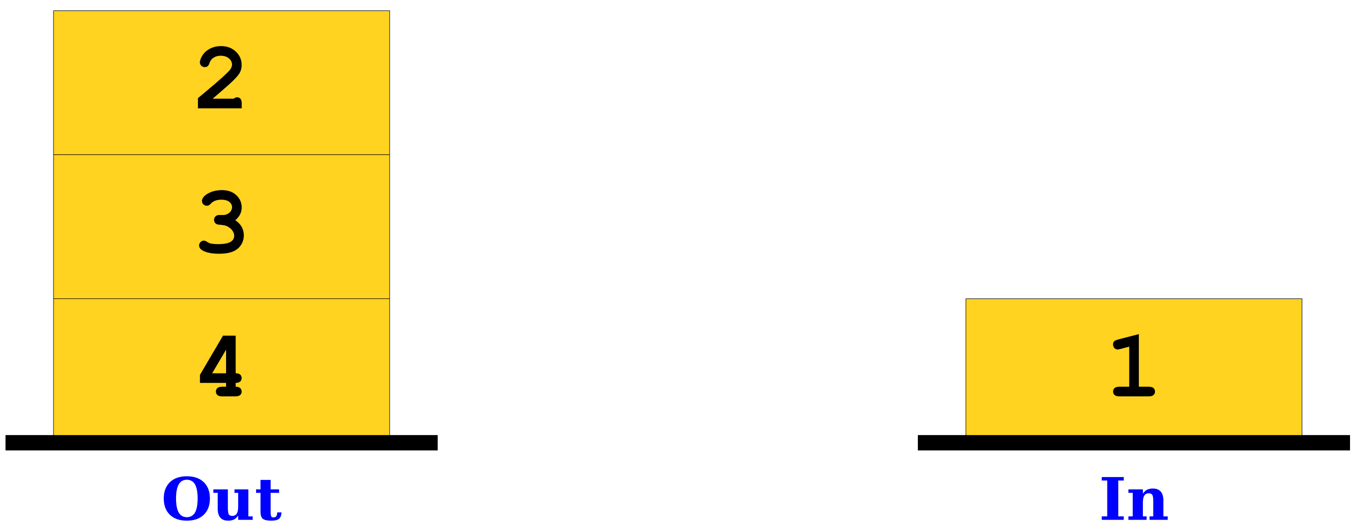
The Two-Stack Queue



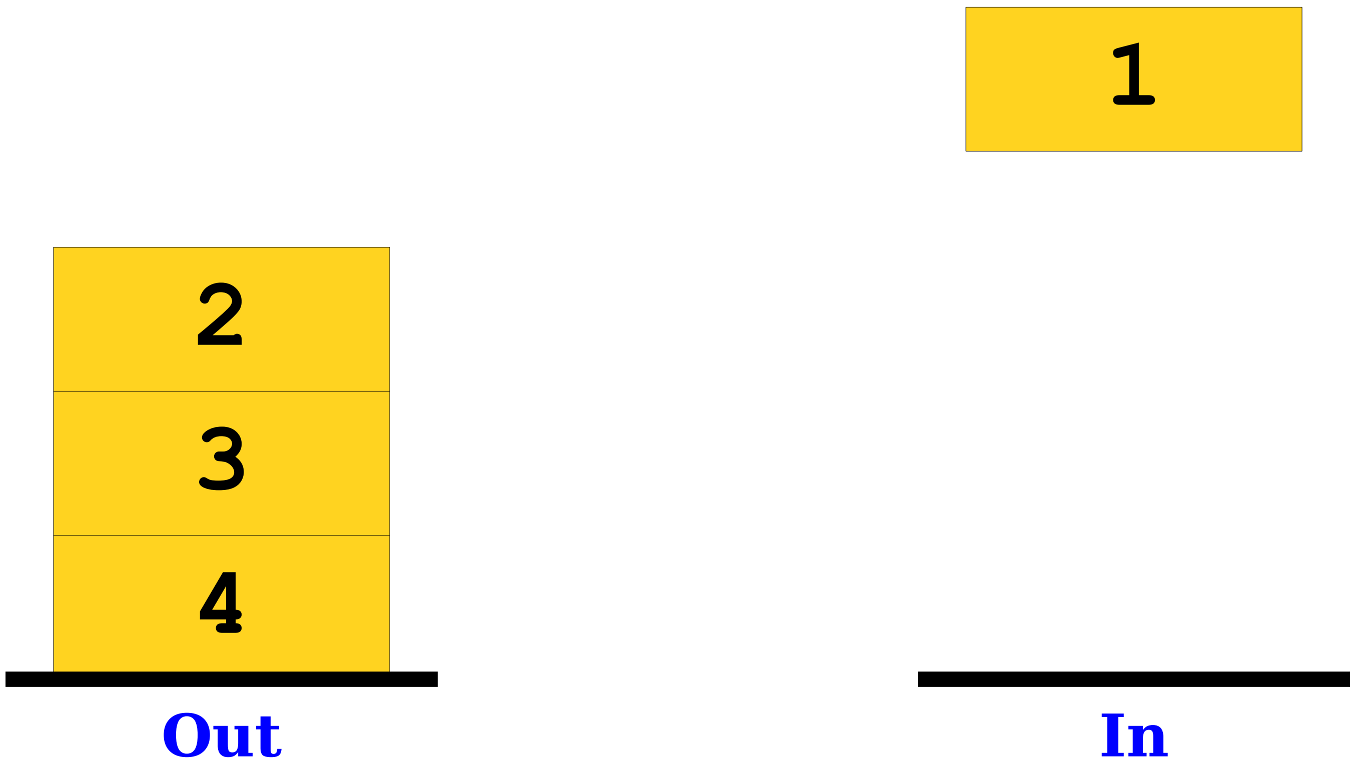
The Two-Stack Queue



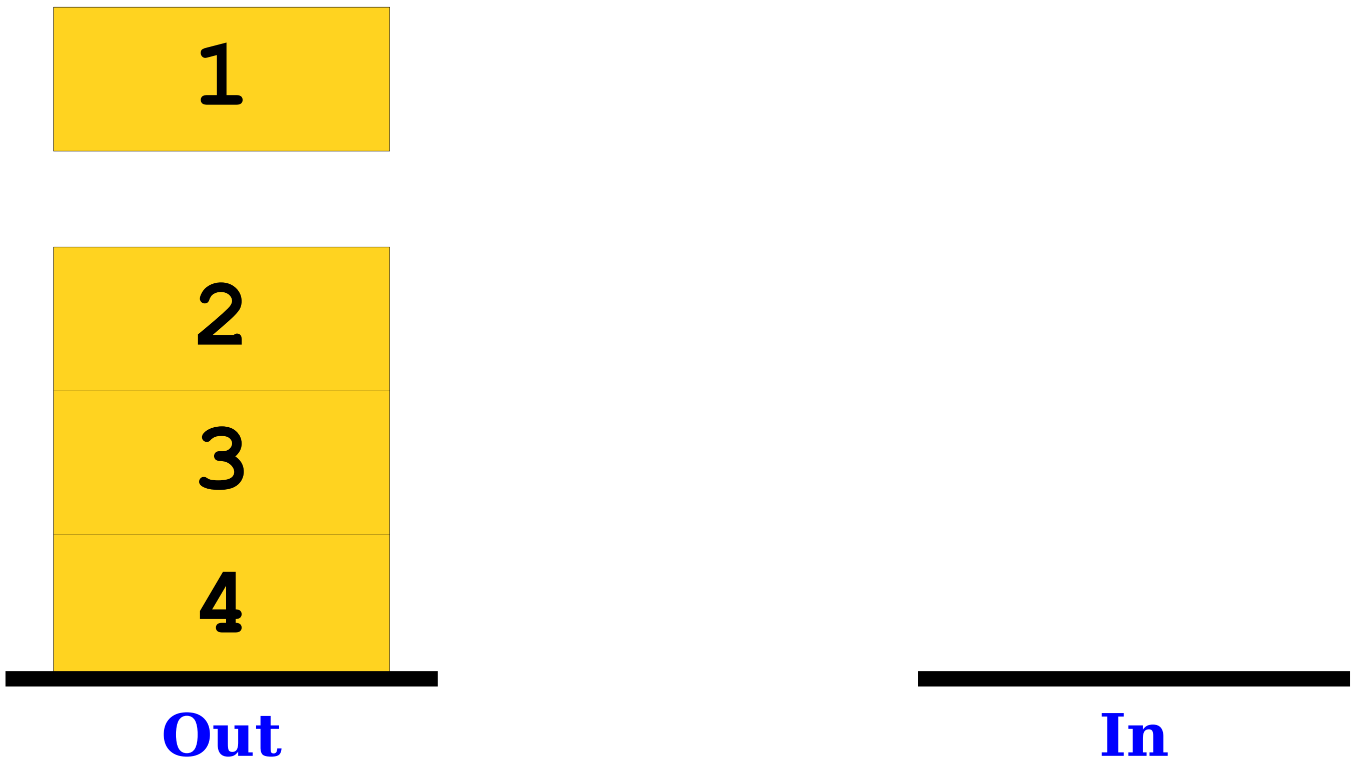
The Two-Stack Queue



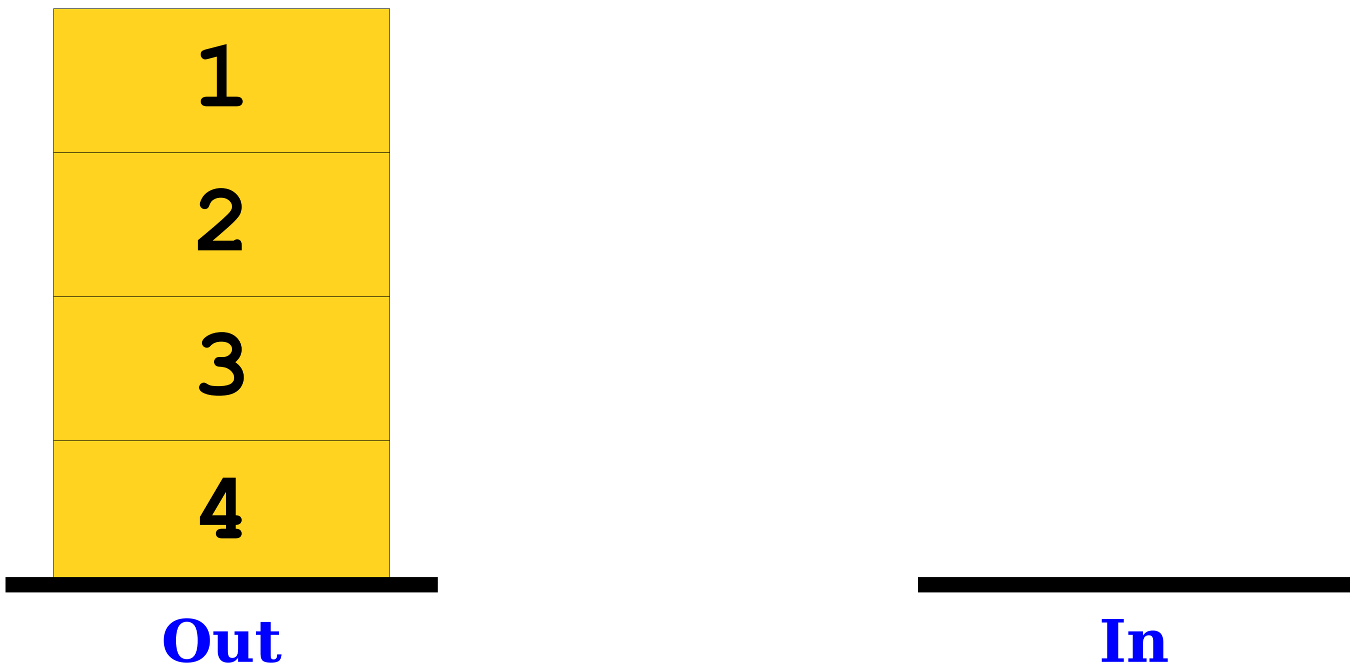
The Two-Stack Queue



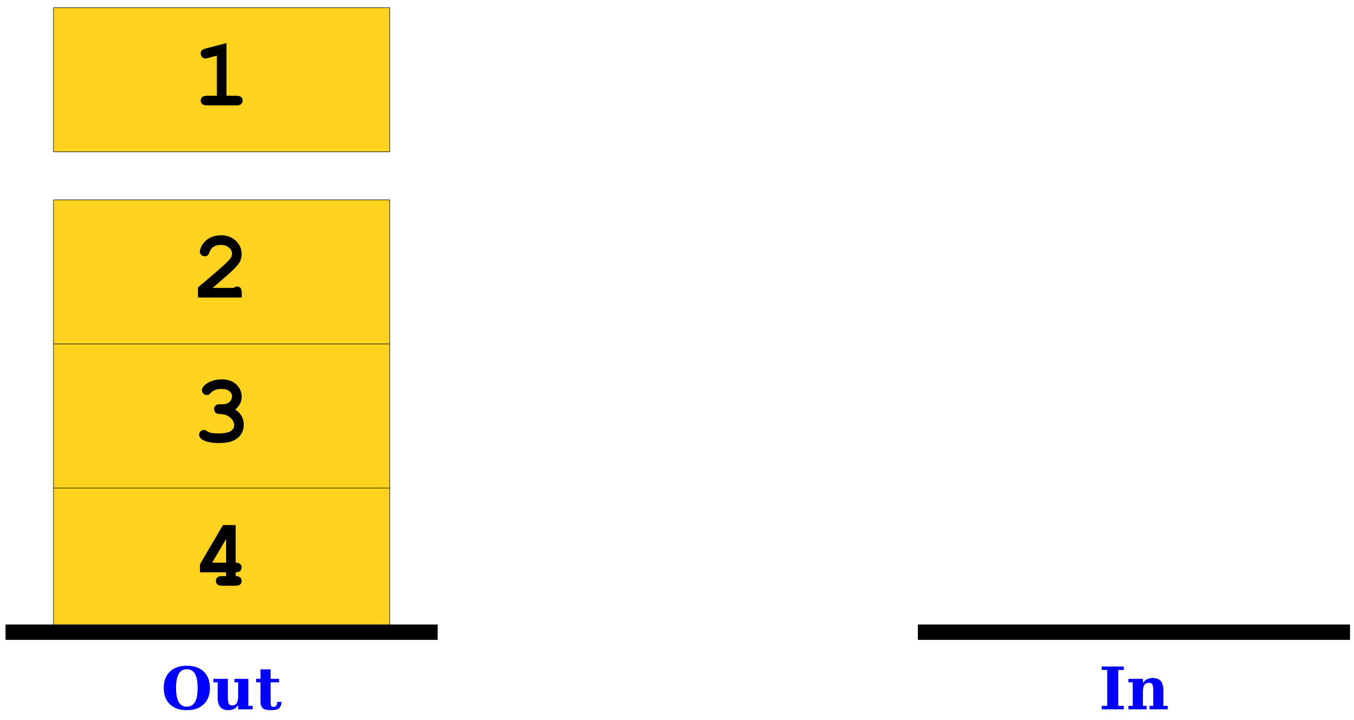
The Two-Stack Queue



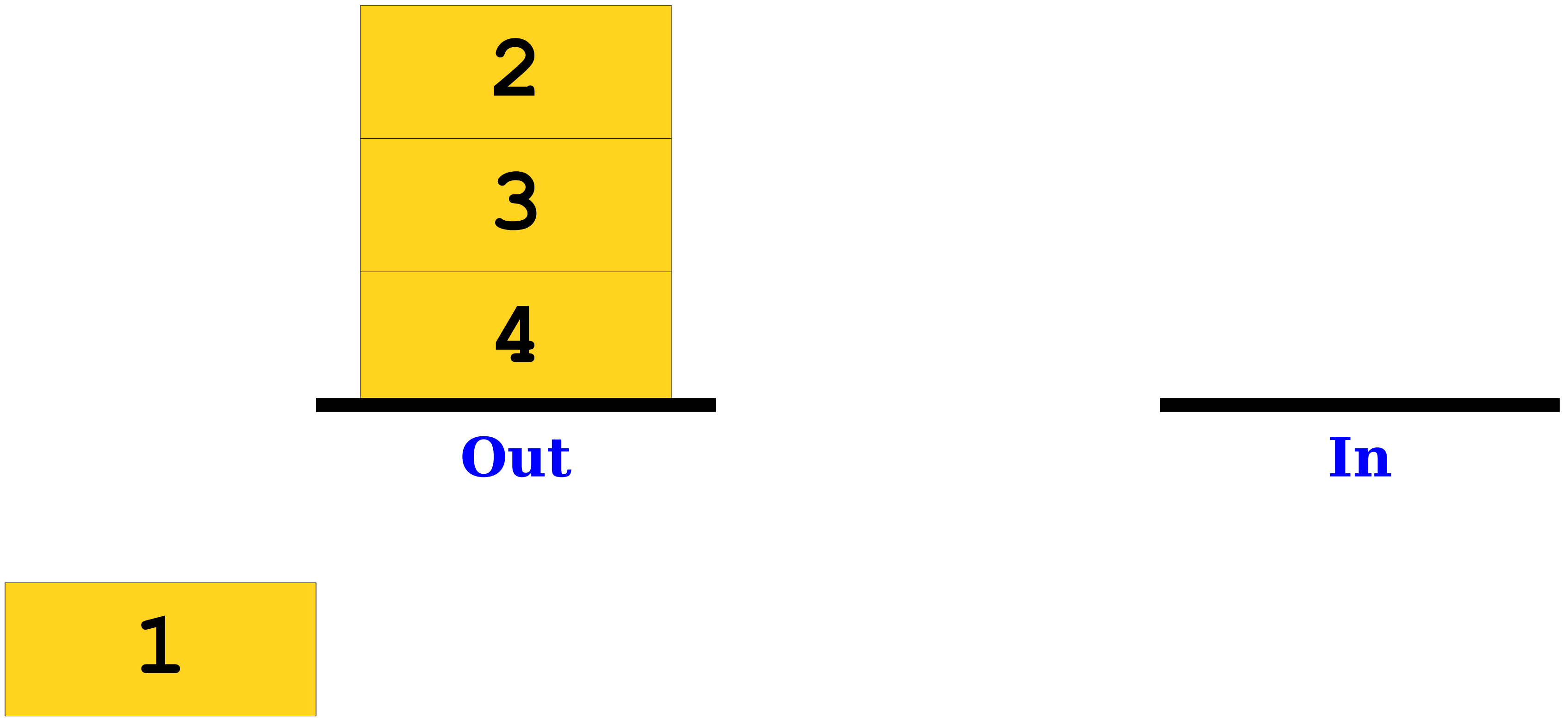
The Two-Stack Queue



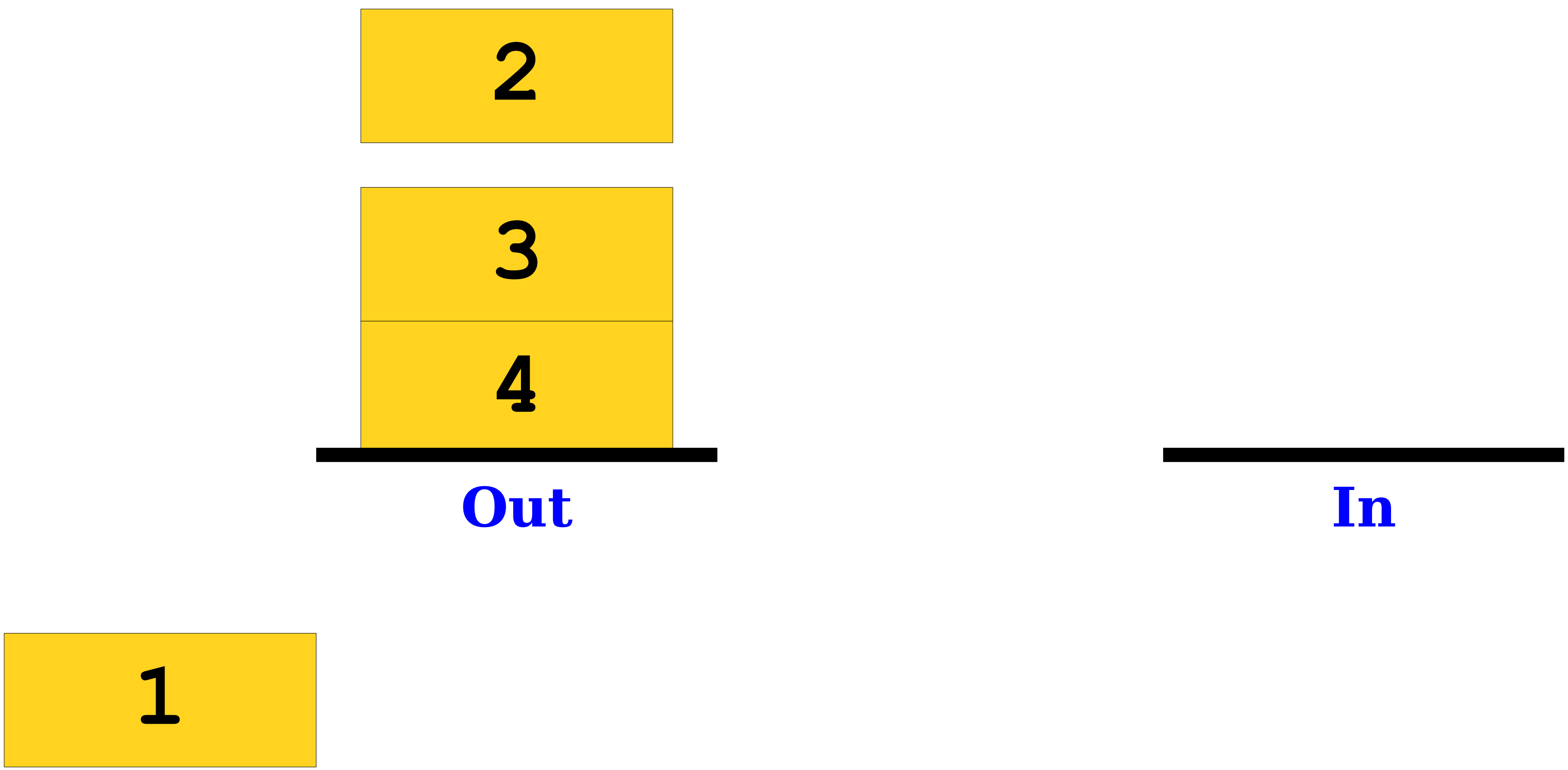
The Two-Stack Queue



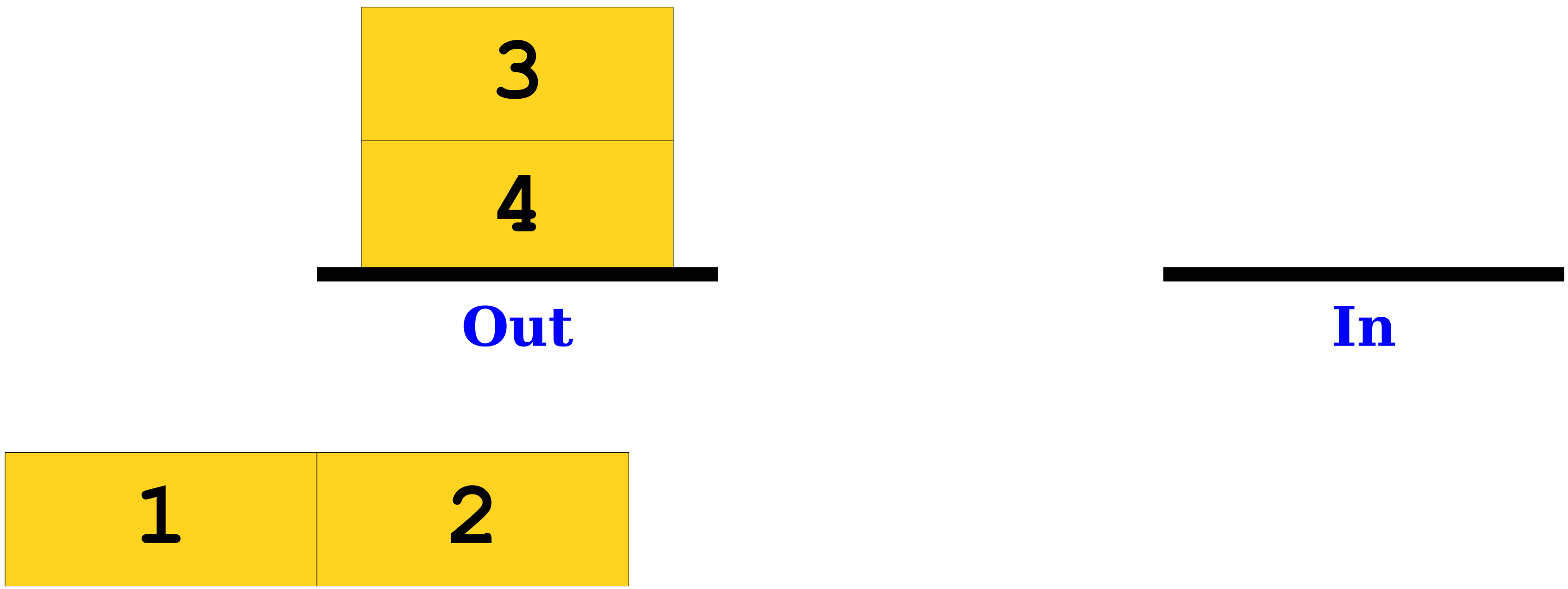
The Two-Stack Queue



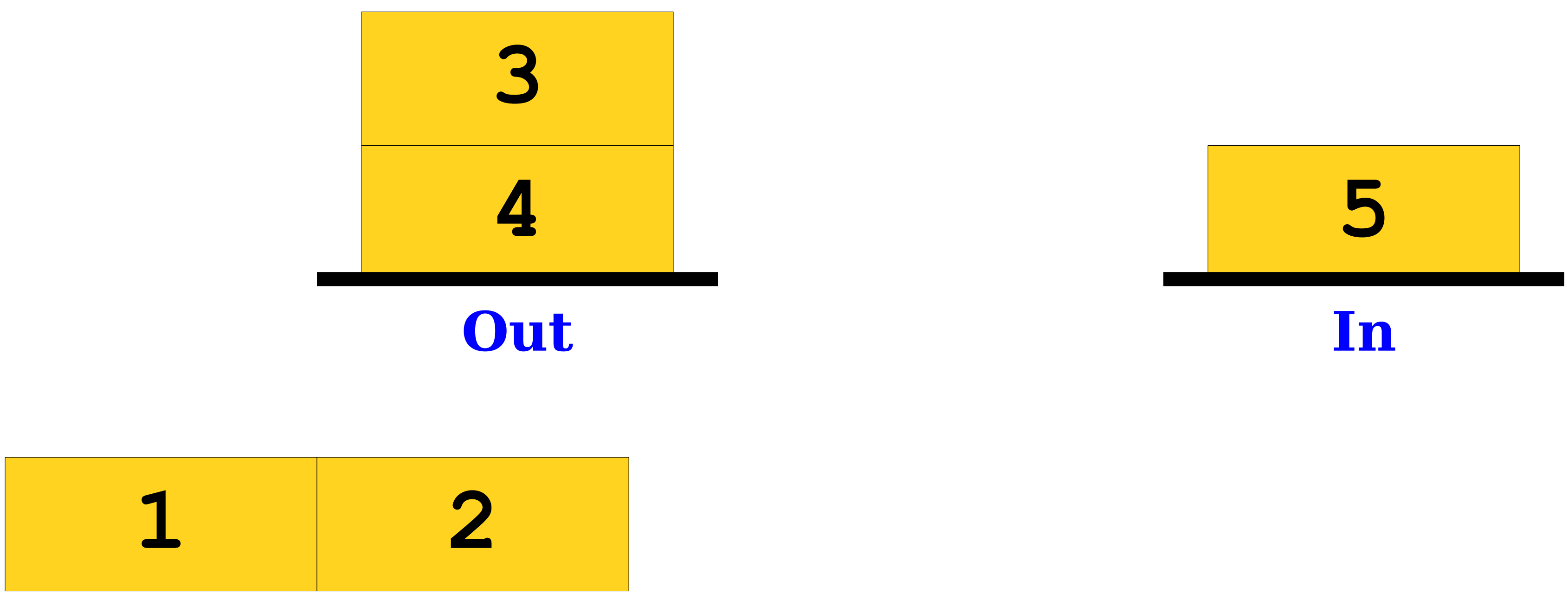
The Two-Stack Queue



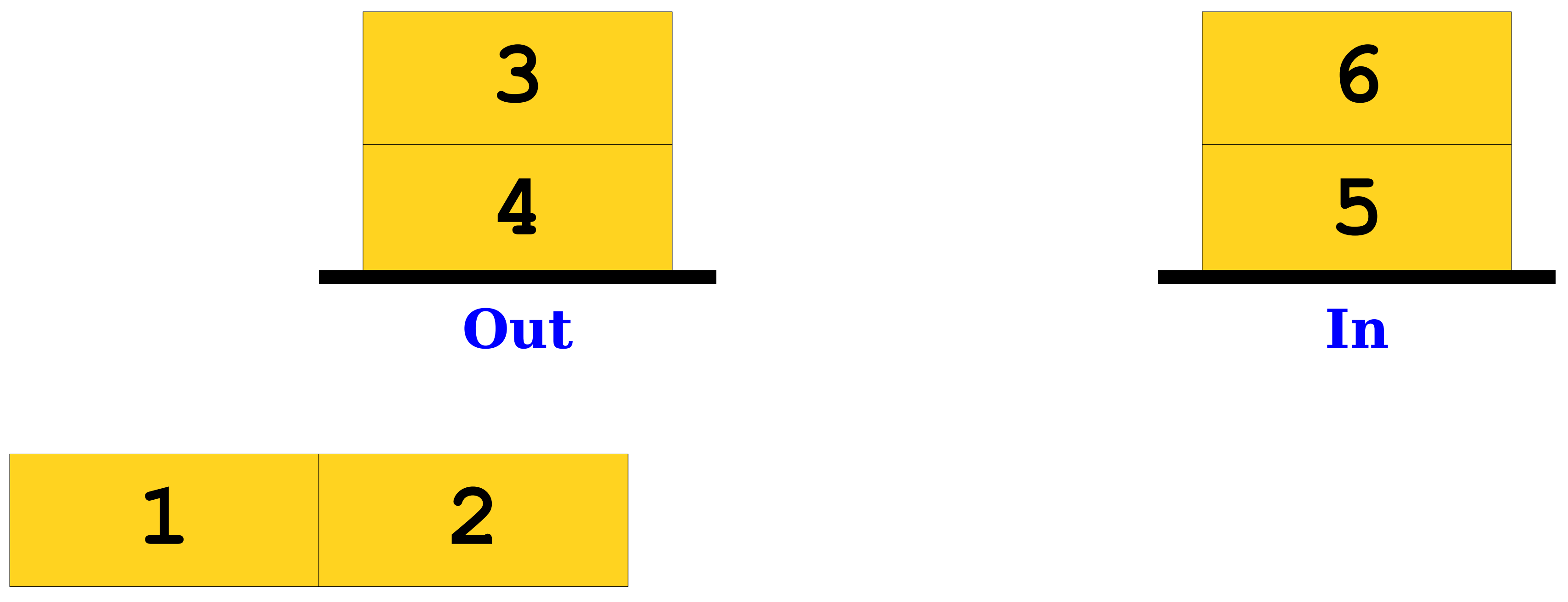
The Two-Stack Queue



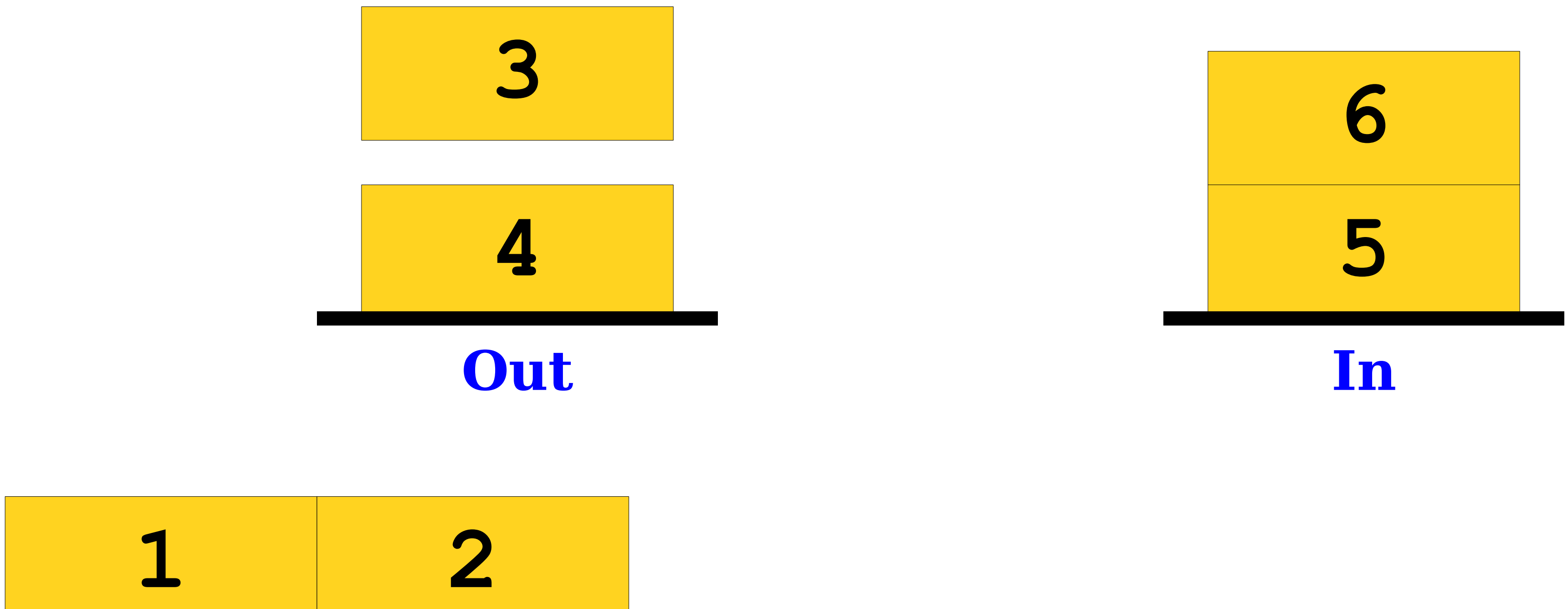
The Two-Stack Queue



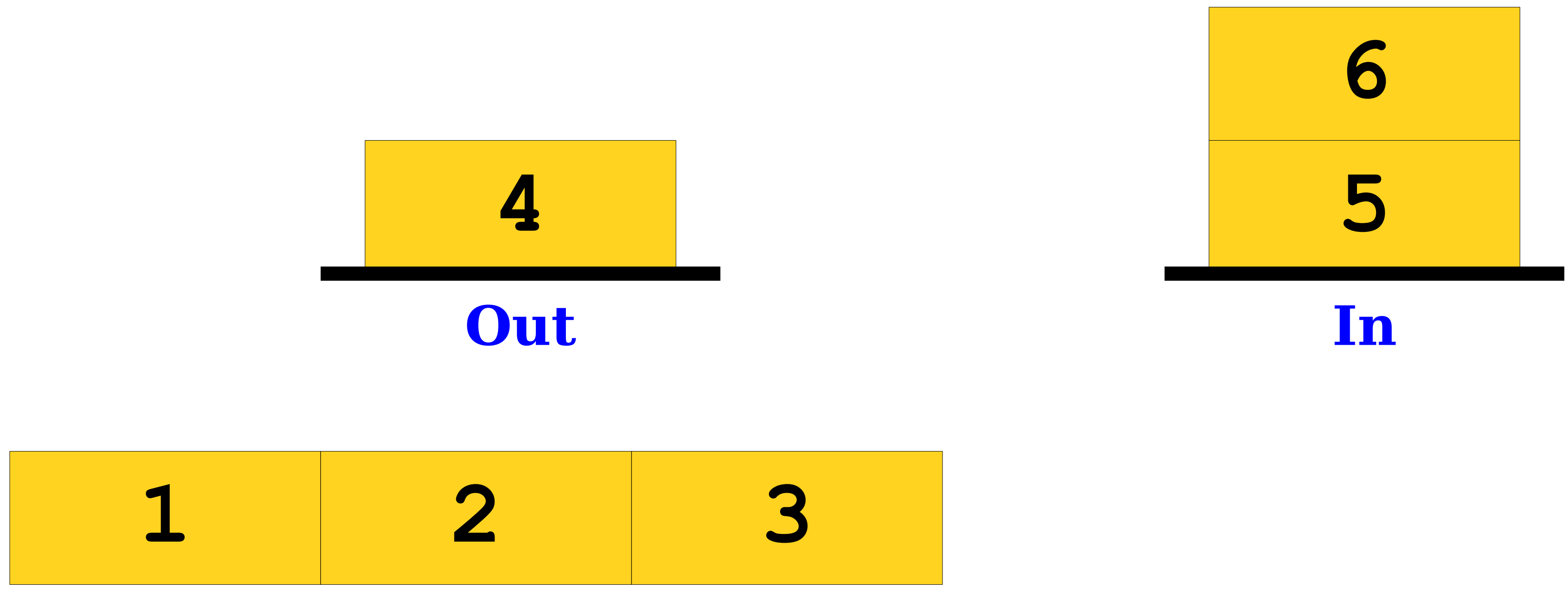
The Two-Stack Queue



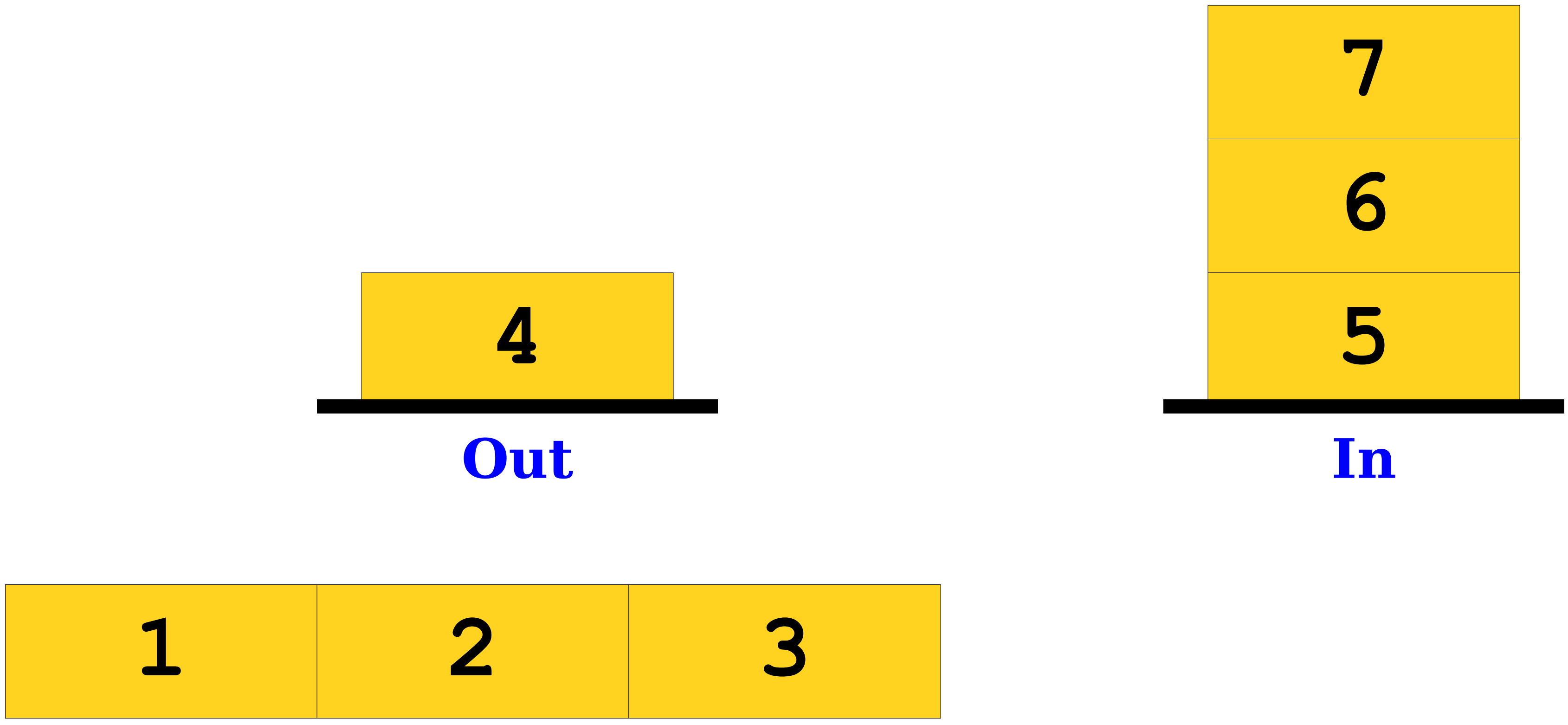
The Two-Stack Queue



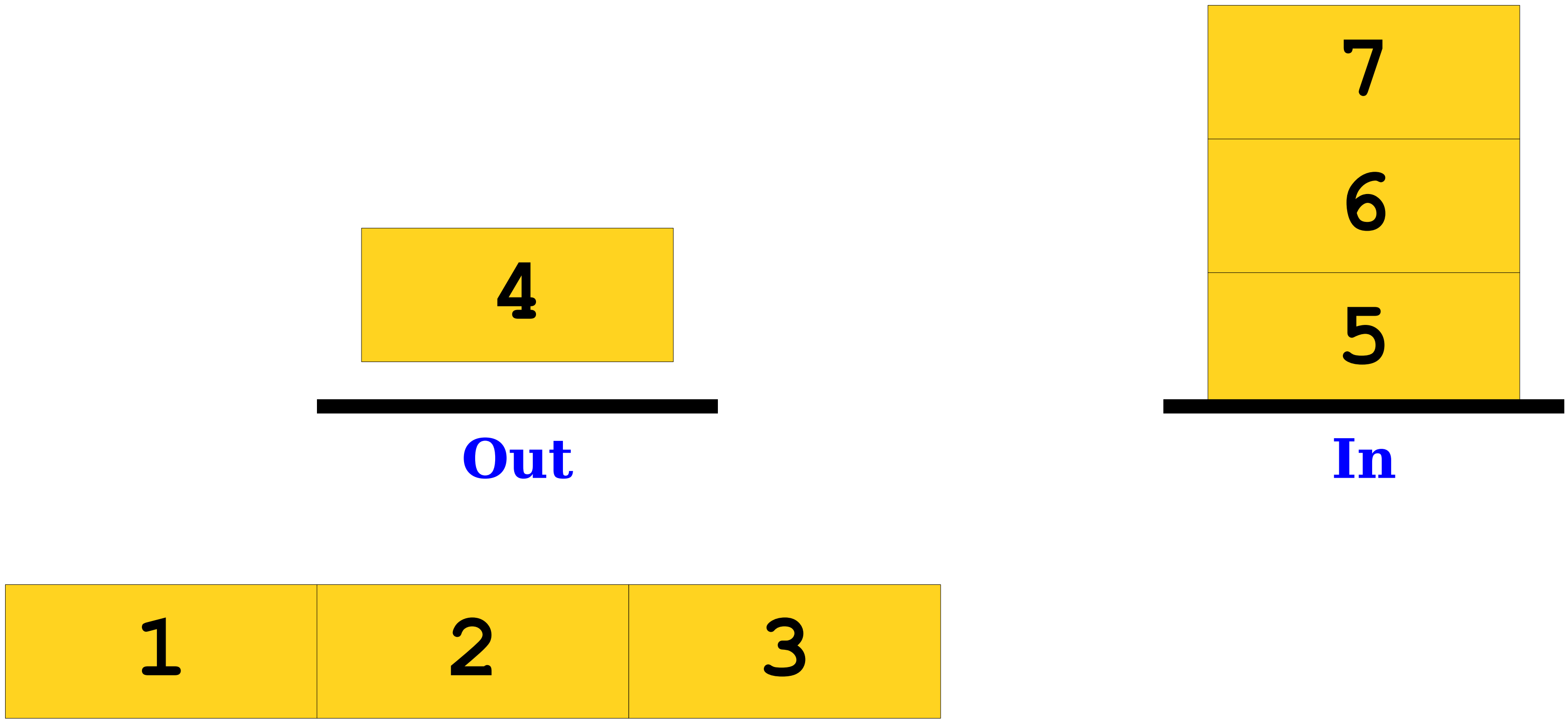
The Two-Stack Queue



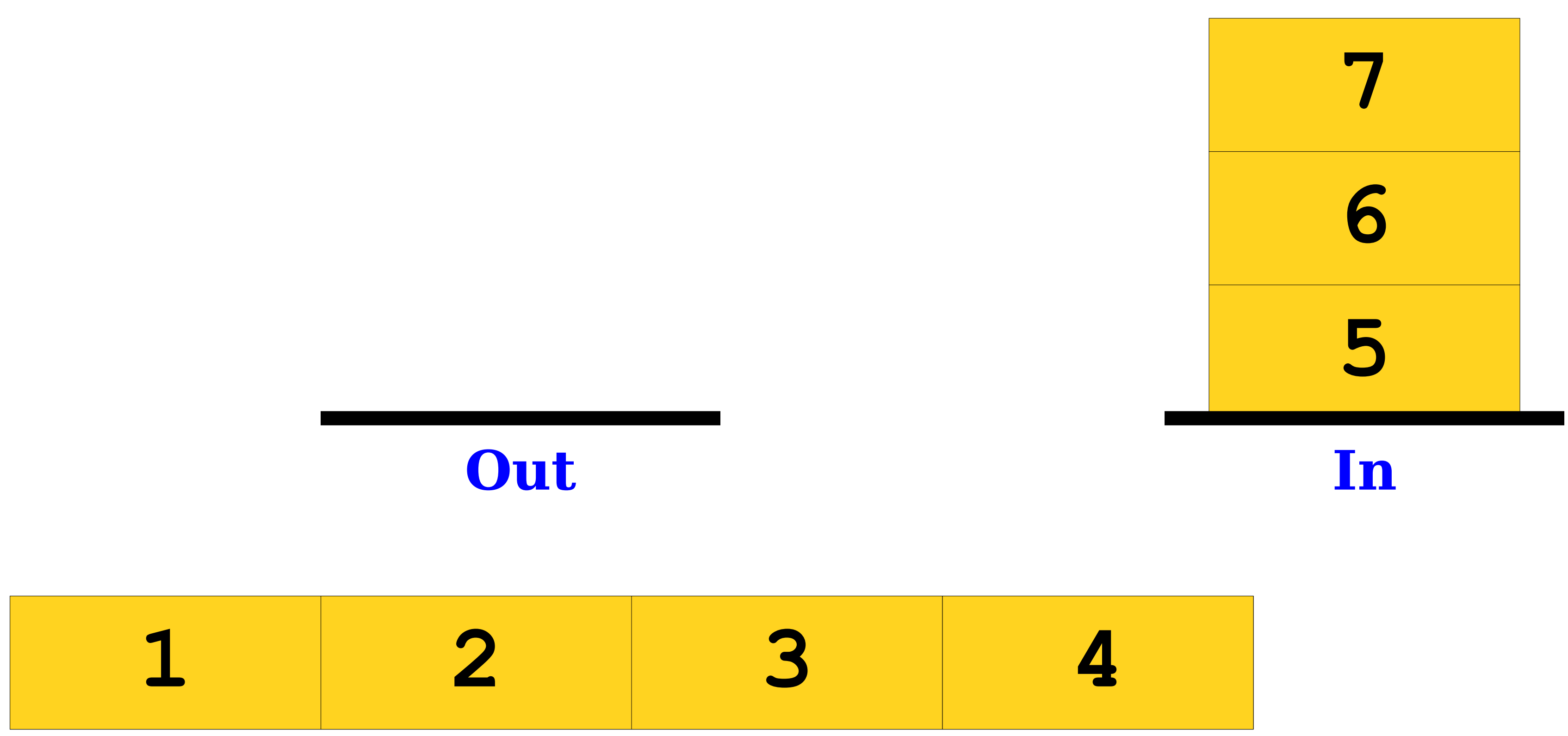
The Two-Stack Queue



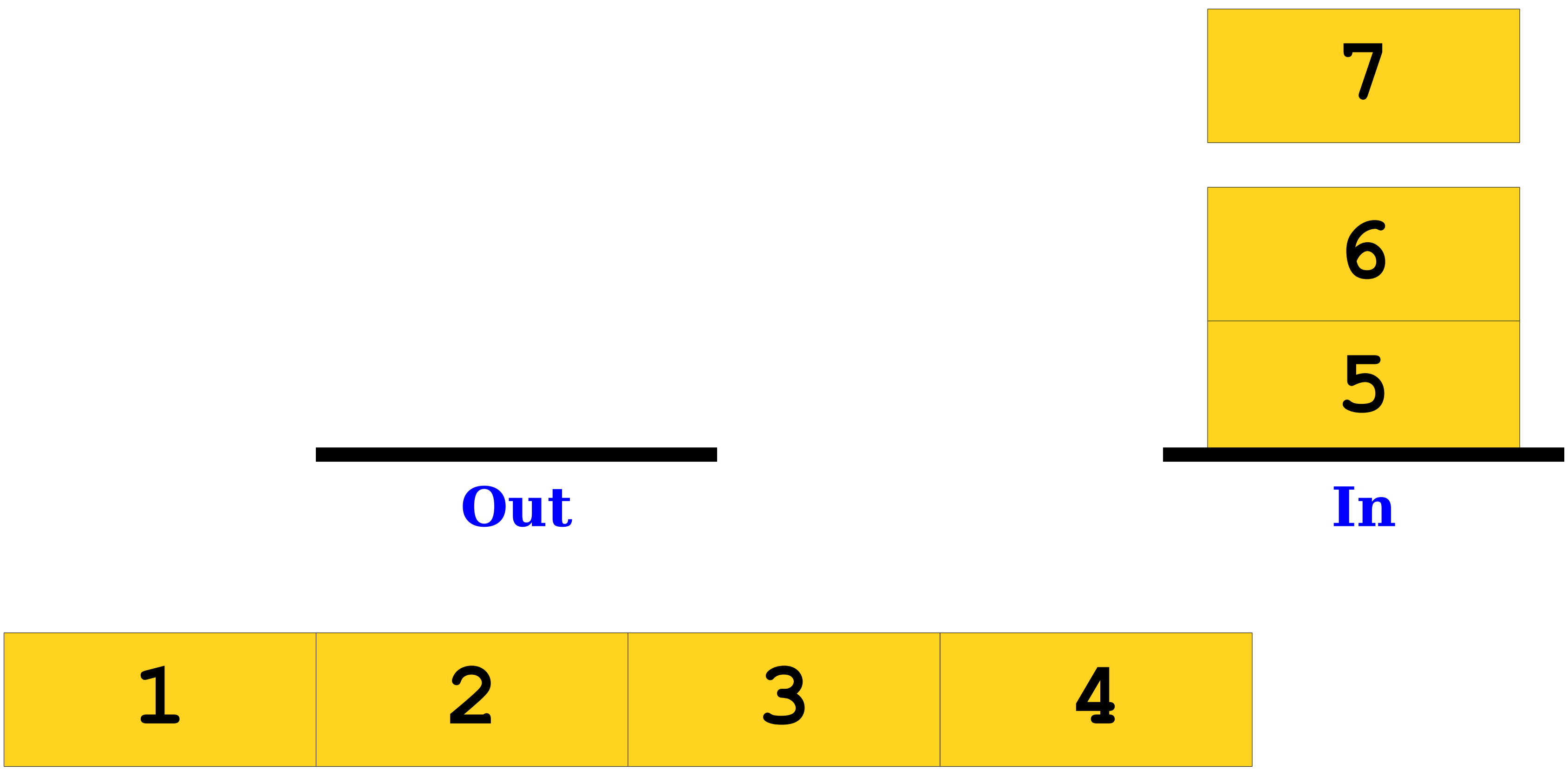
The Two-Stack Queue



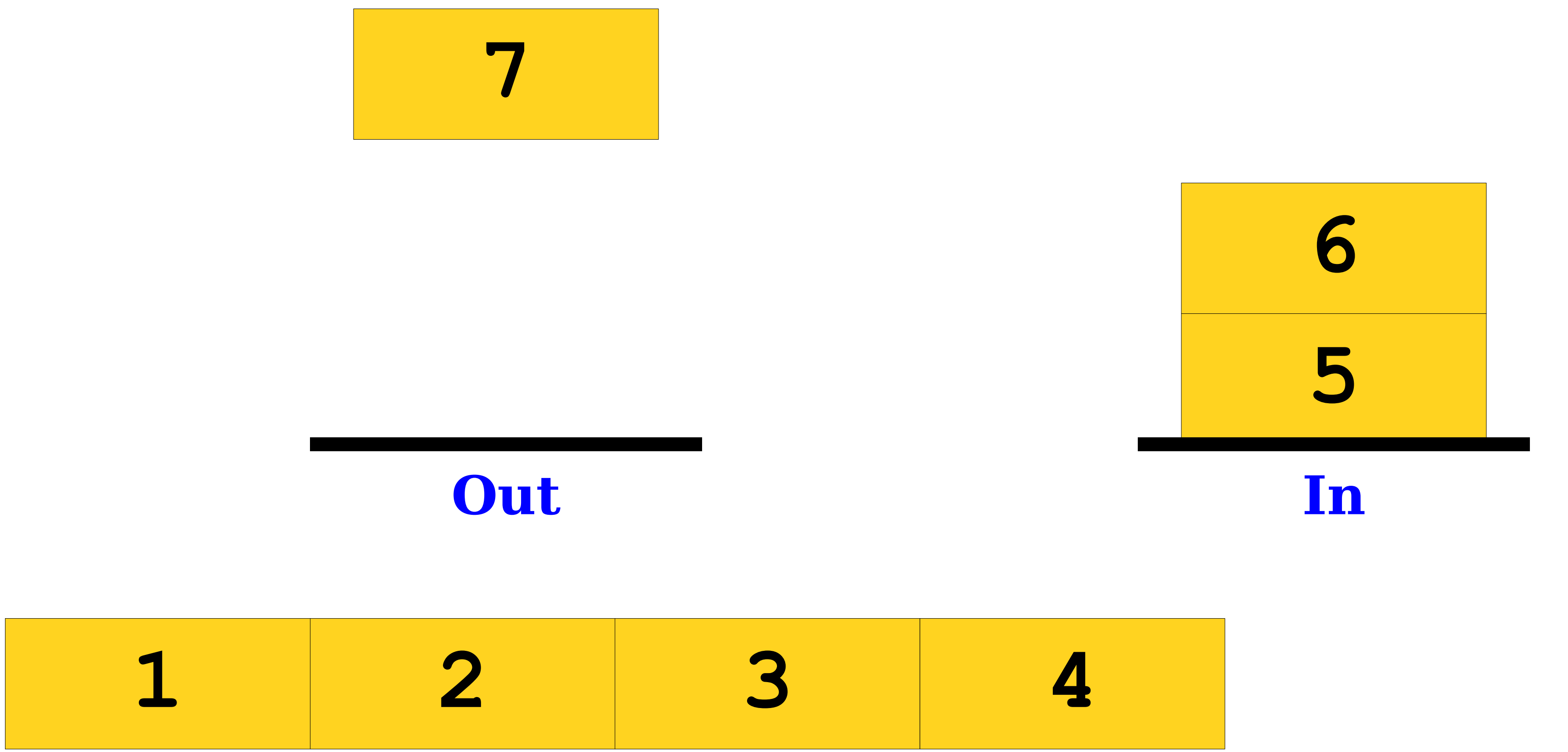
The Two-Stack Queue



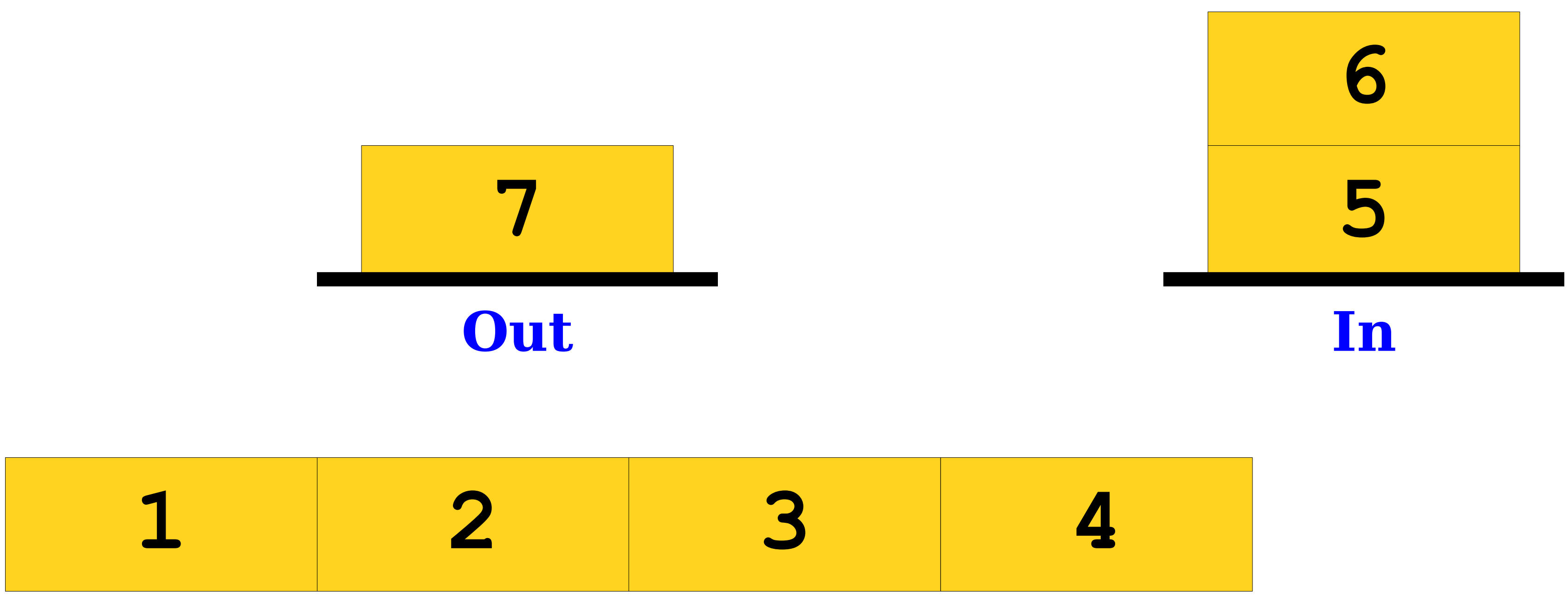
The Two-Stack Queue



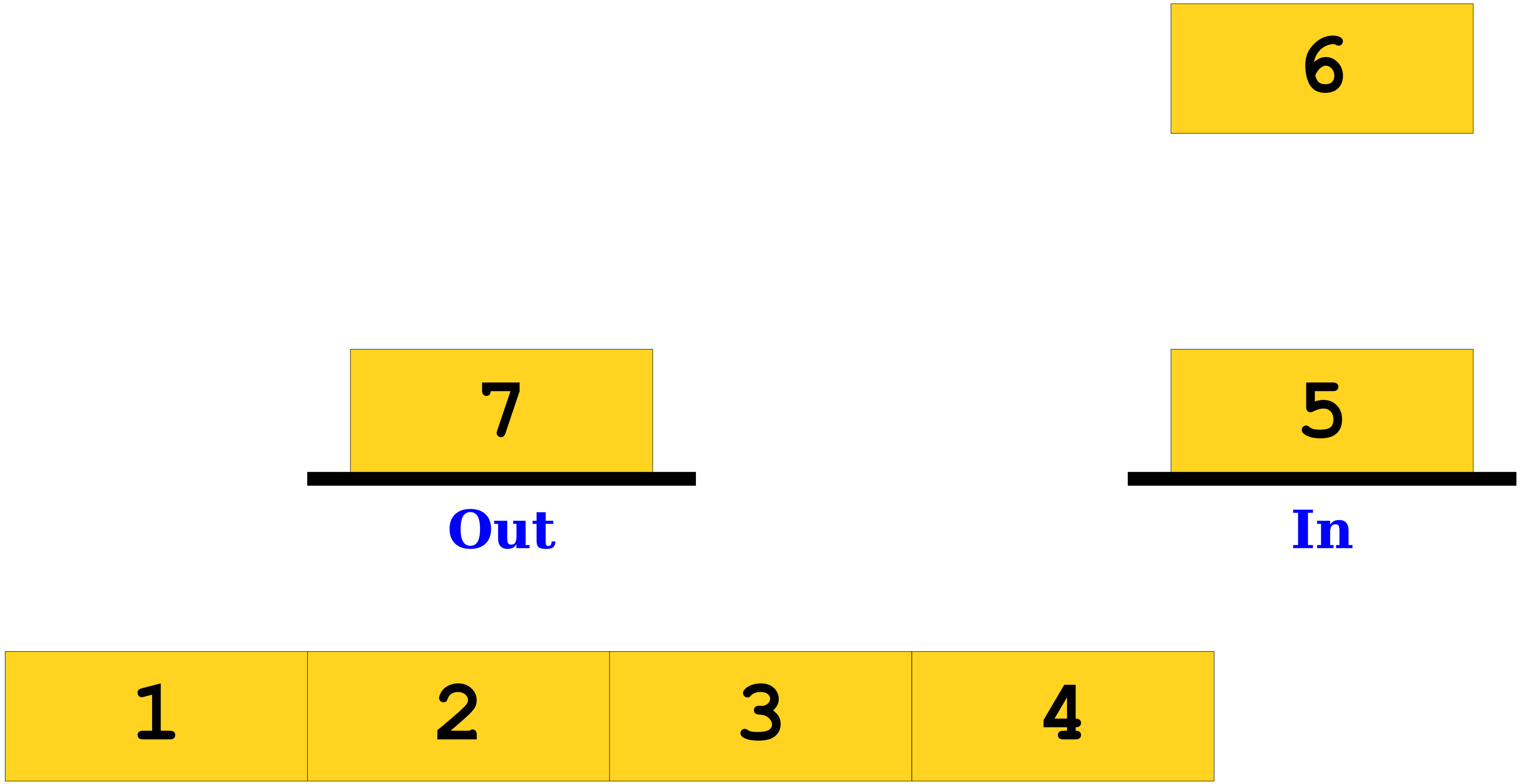
The Two-Stack Queue



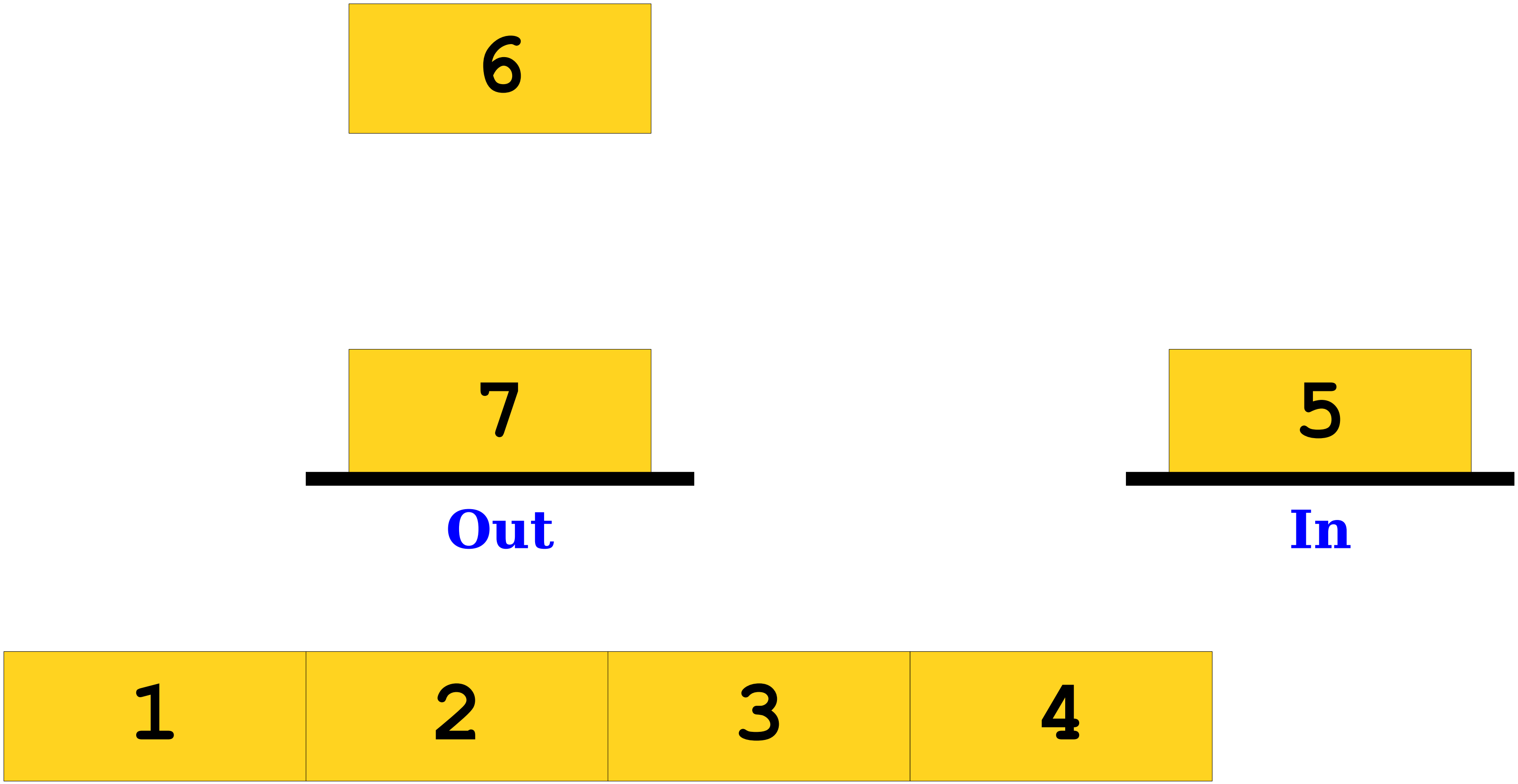
The Two-Stack Queue



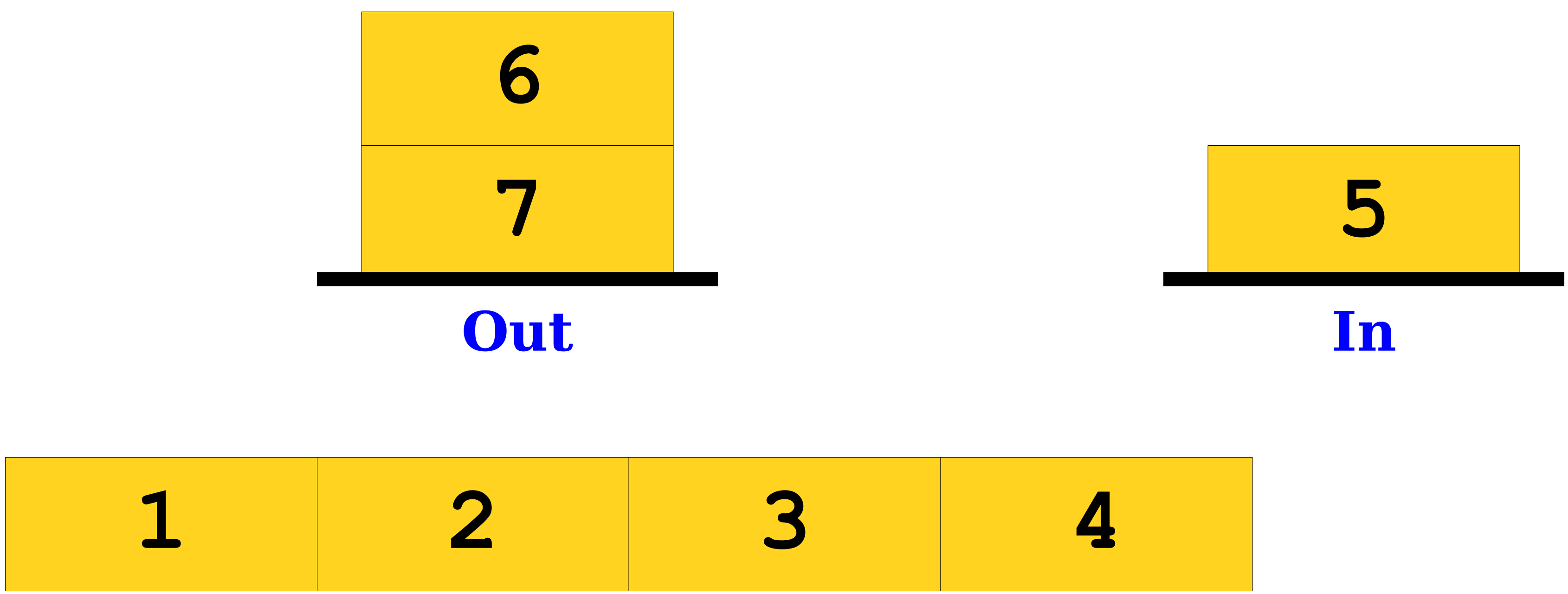
The Two-Stack Queue



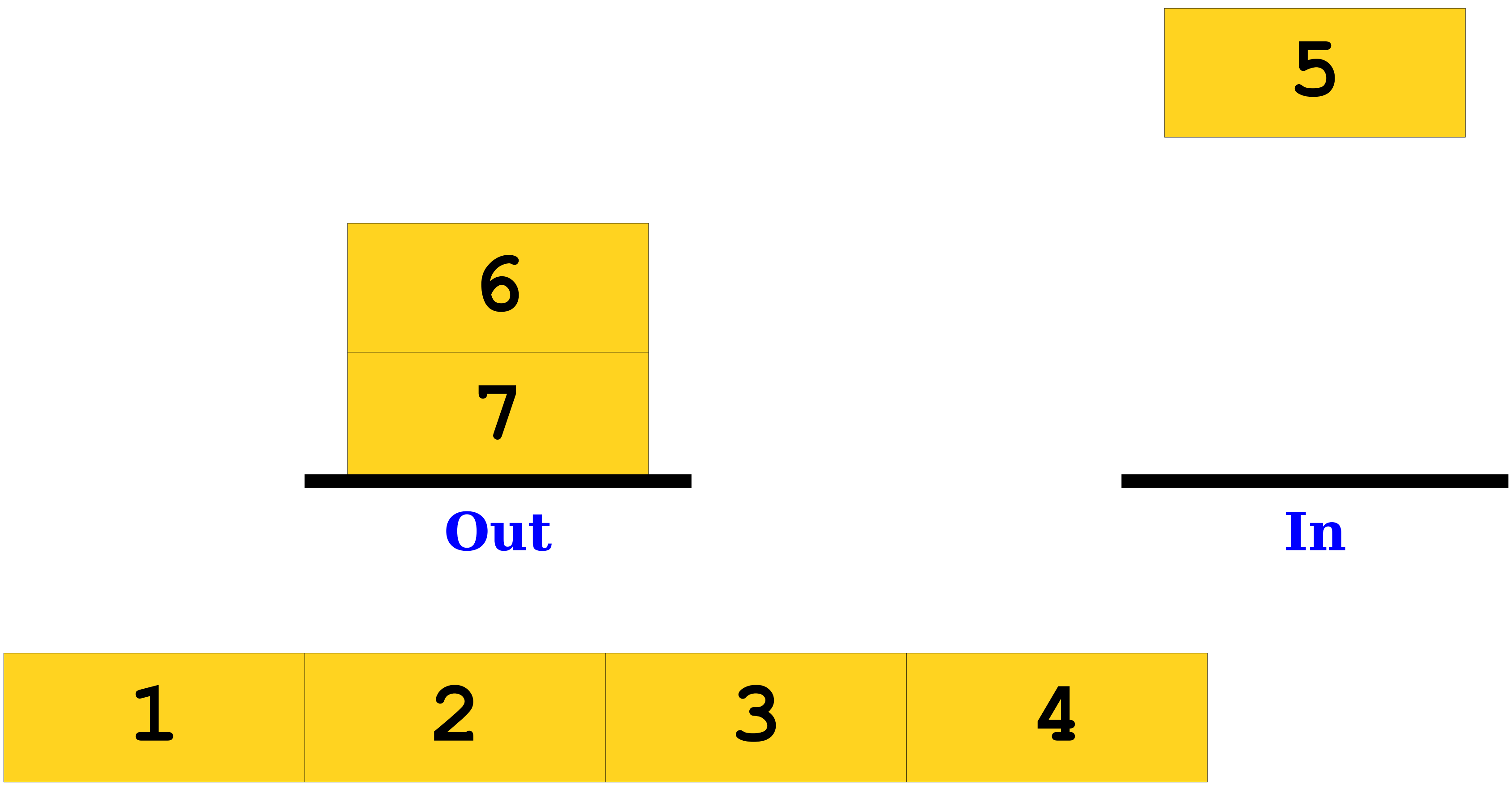
The Two-Stack Queue



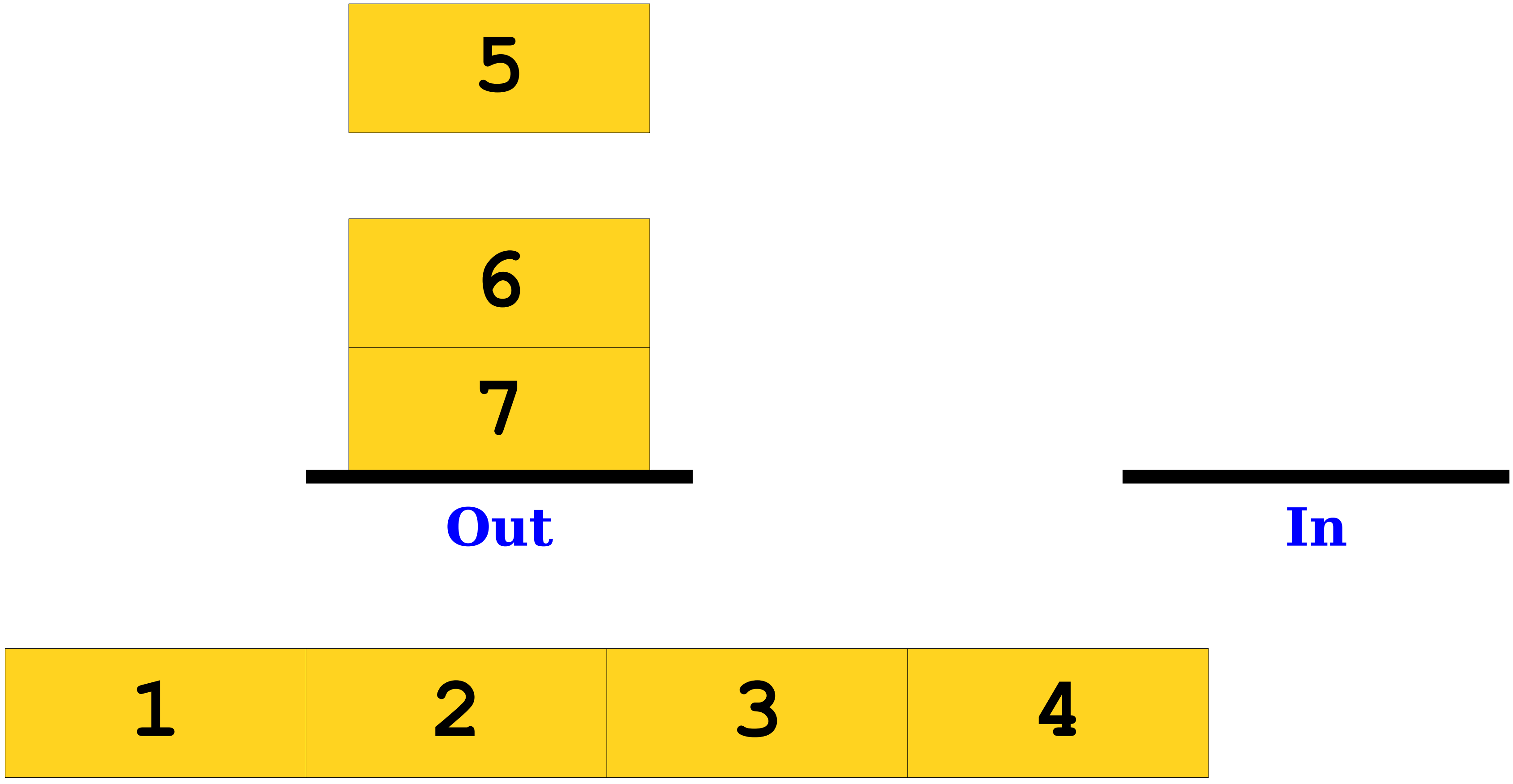
The Two-Stack Queue



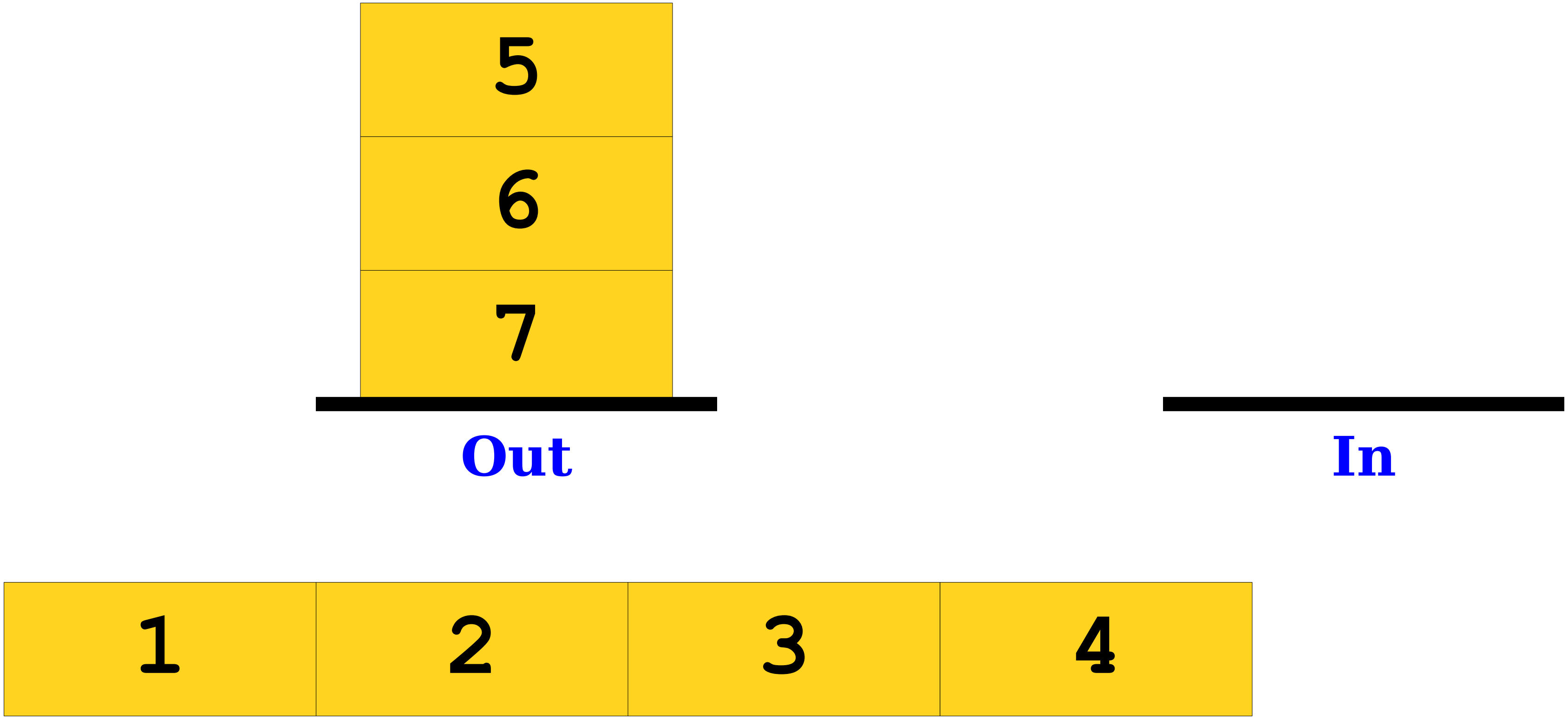
The Two-Stack Queue



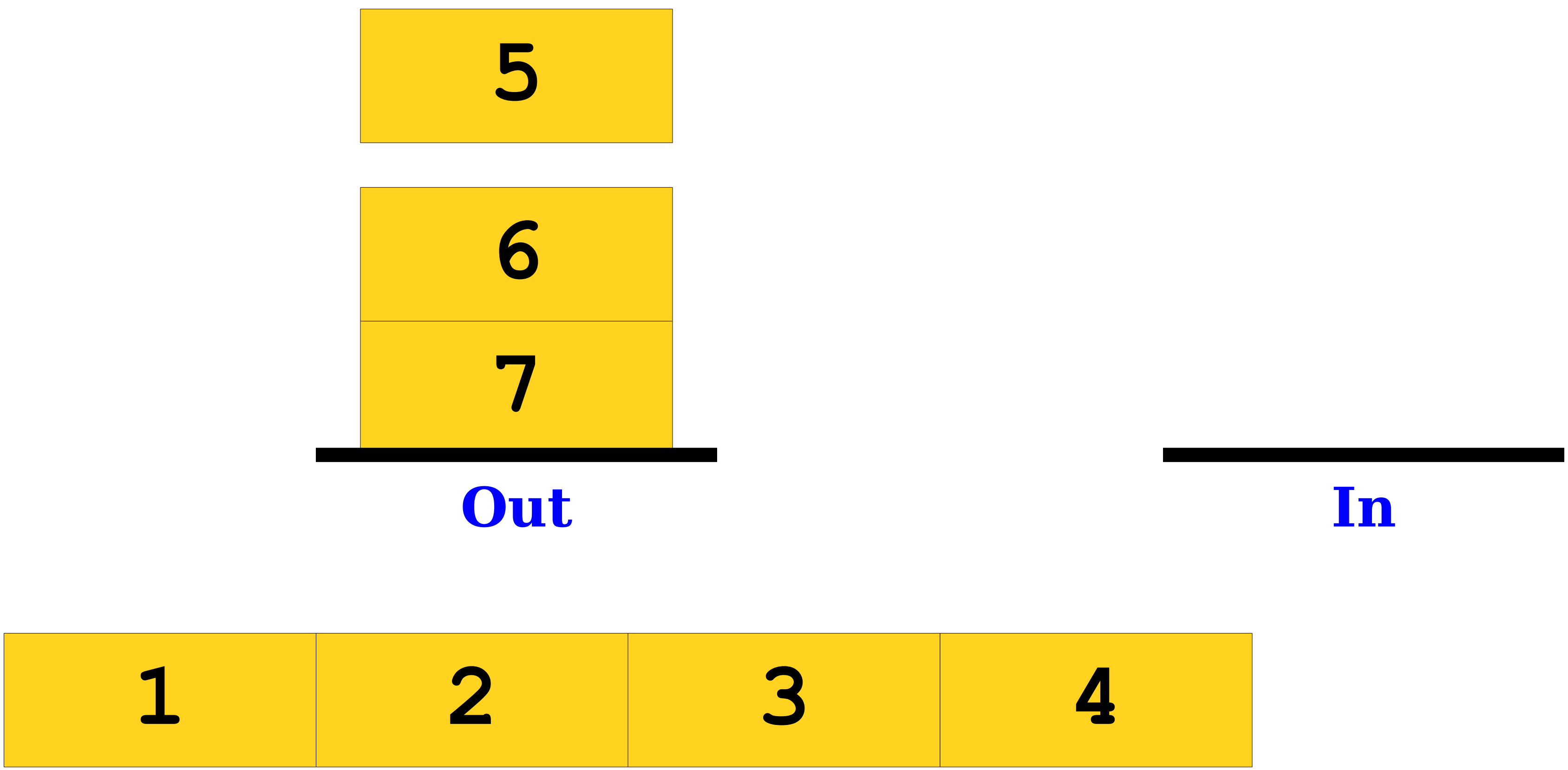
The Two-Stack Queue



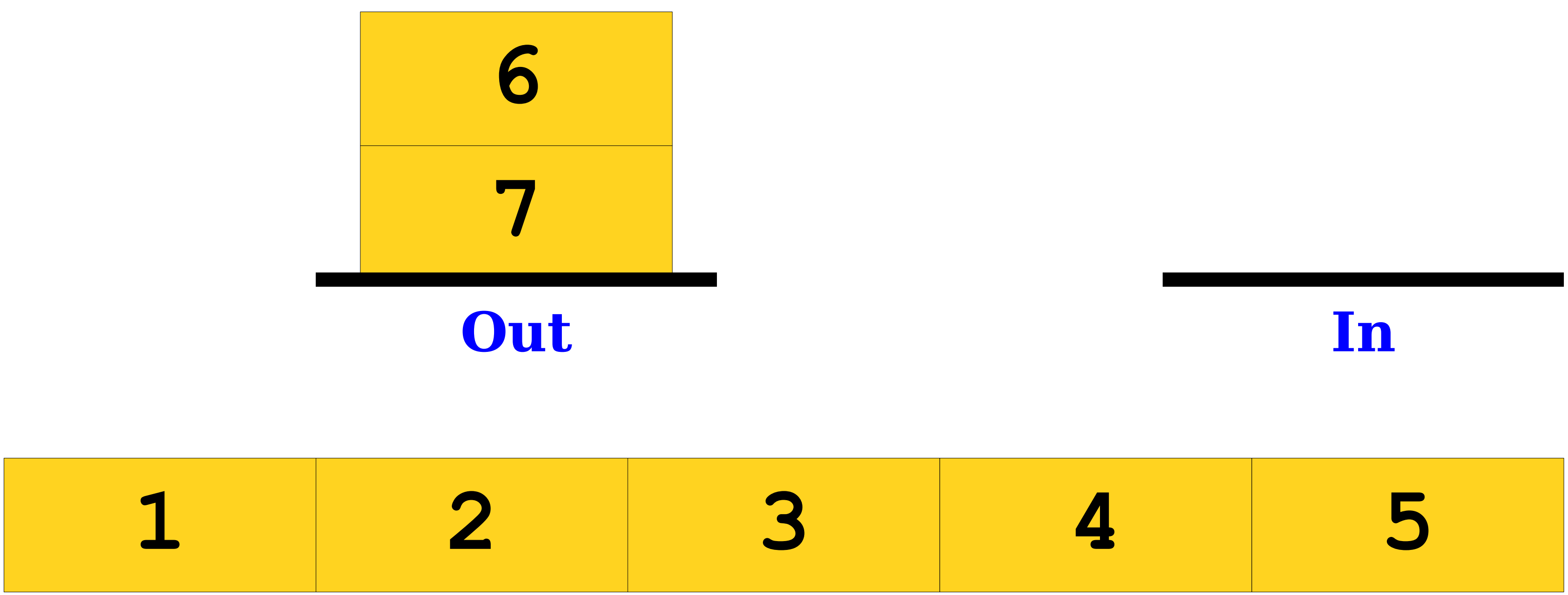
The Two-Stack Queue



The Two-Stack Queue



The Two-Stack Queue



The Two-Stack Queue

- Maintain two stacks, an ***In*** stack and an ***Out*** stack.
- To enqueue an element, push it onto the ***In*** stack.
- To dequeue an element:
  - If the ***Out*** stack is empty, pop everything off the ***In*** stack and push it onto the ***Out*** stack.
  - Pop the ***Out*** stack and return its value.

An Aggregate Analysis

- ***Claim:*** The amortized cost of popping an element is O(1).
- ***Proof:***
  - Every value is pushed onto a stack at most twice: once for *in*, once for *out*.
  - Every value is popped off of a stack at most twice: once for *in*, once for *out*.
  - Each push/pop takes time O(1).
  - Net runtime: O(n).
  - Amortized cost: O(n) / n = **O(1)**.

The Banker's Method

- Let's analyze this data structure using the banker's method.
- Some observations:
- All enqueues take worst-case time O(1).
- Each dequeue can be split into a “light” or “heavy” dequeue.
  - In a “light” dequeue, the ***out*** stack is nonempty. Worst-case time is O(1).
  - In a “heavy” dequeue, the ***out*** stack is empty. Worst-case time is O(n).

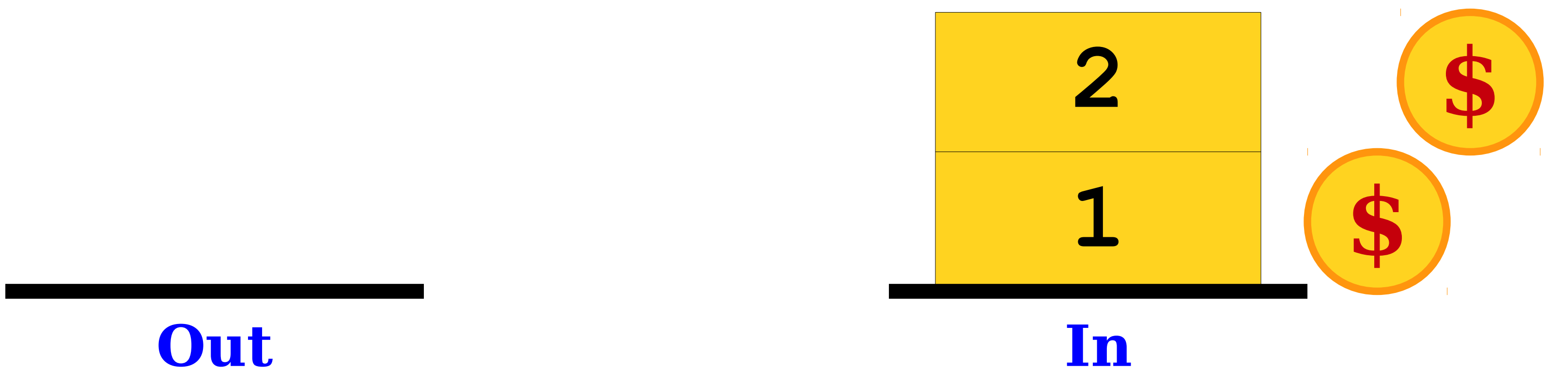
The Two-Stack Queue

Out

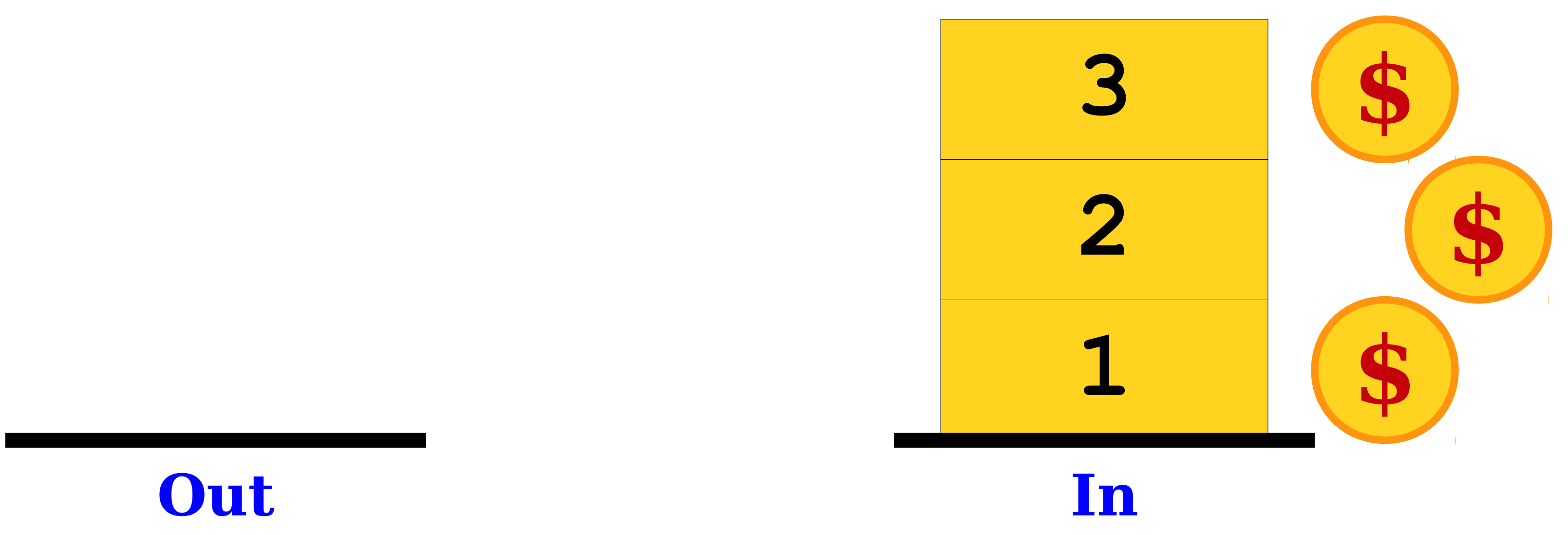
In

The Two-Stack Queue

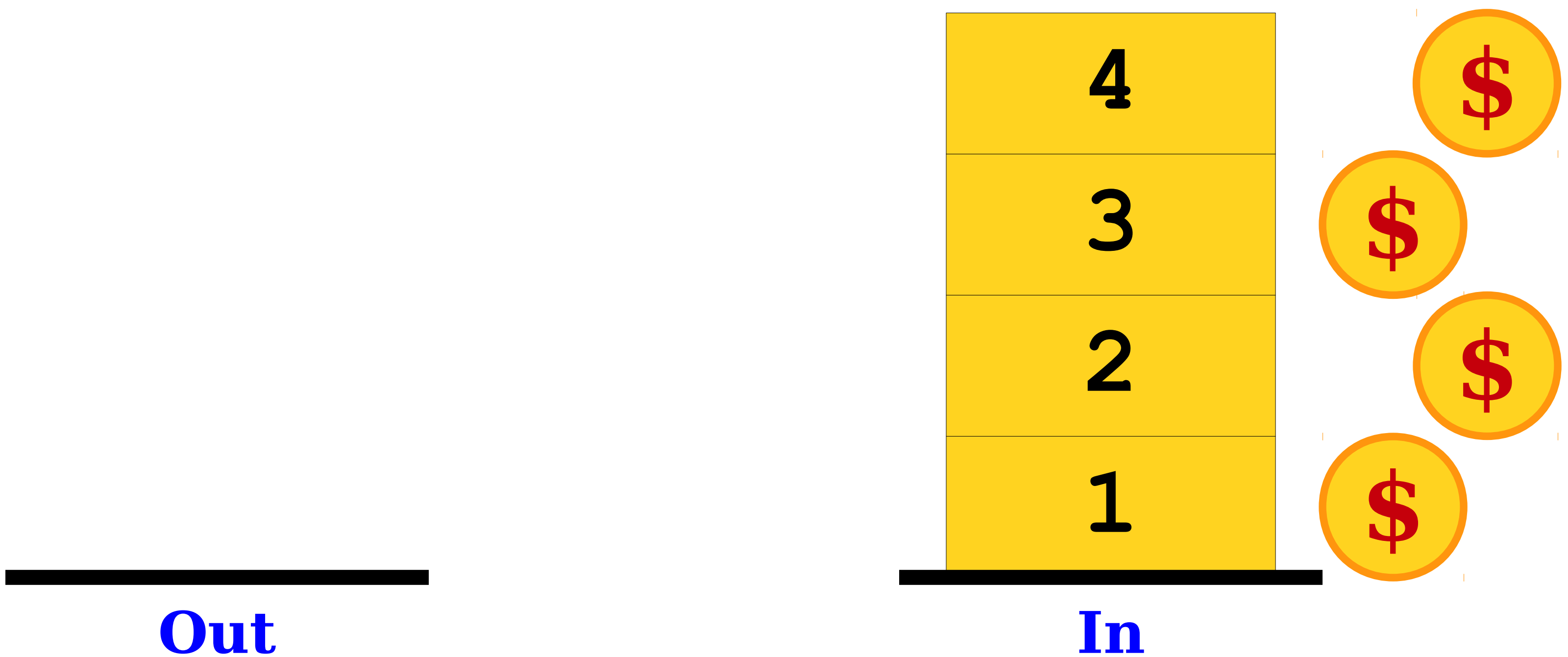
The Two-Stack Queue



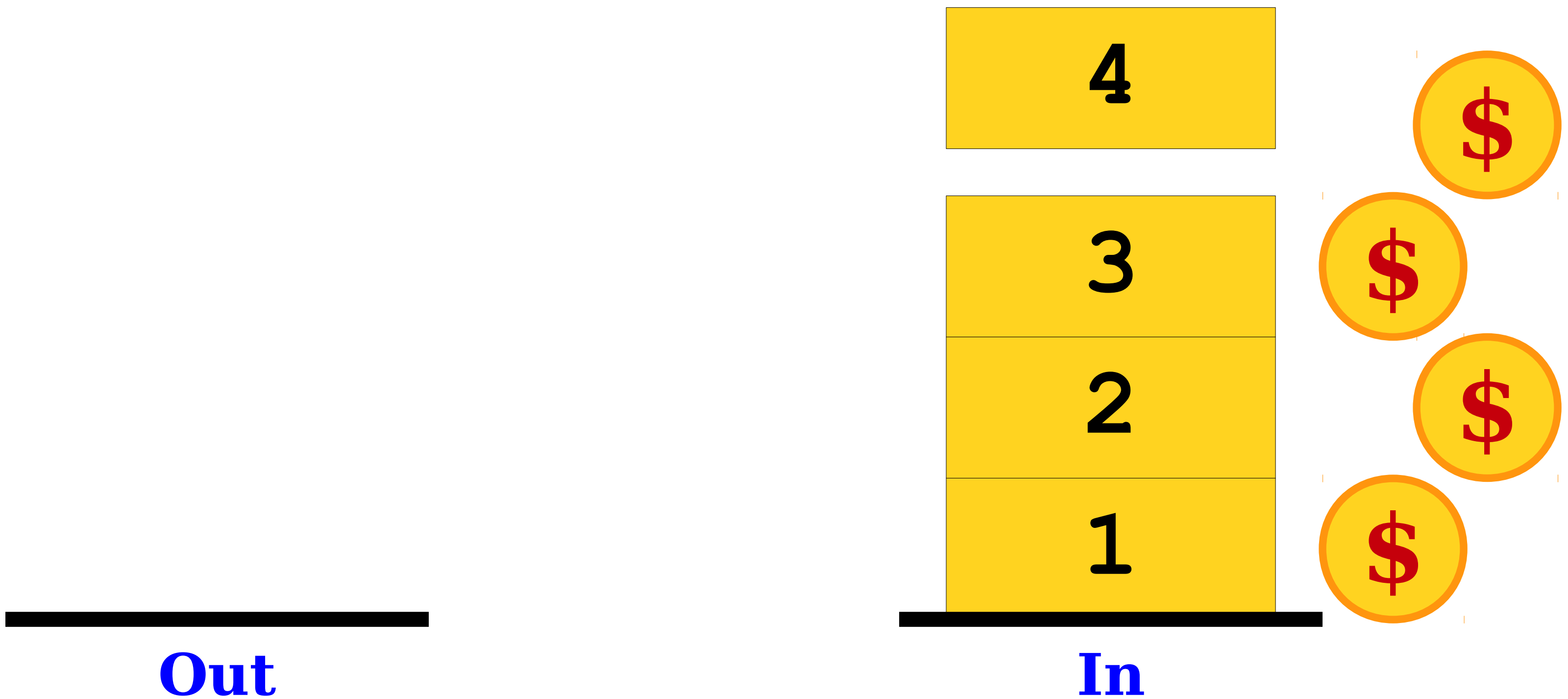
The Two-Stack Queue



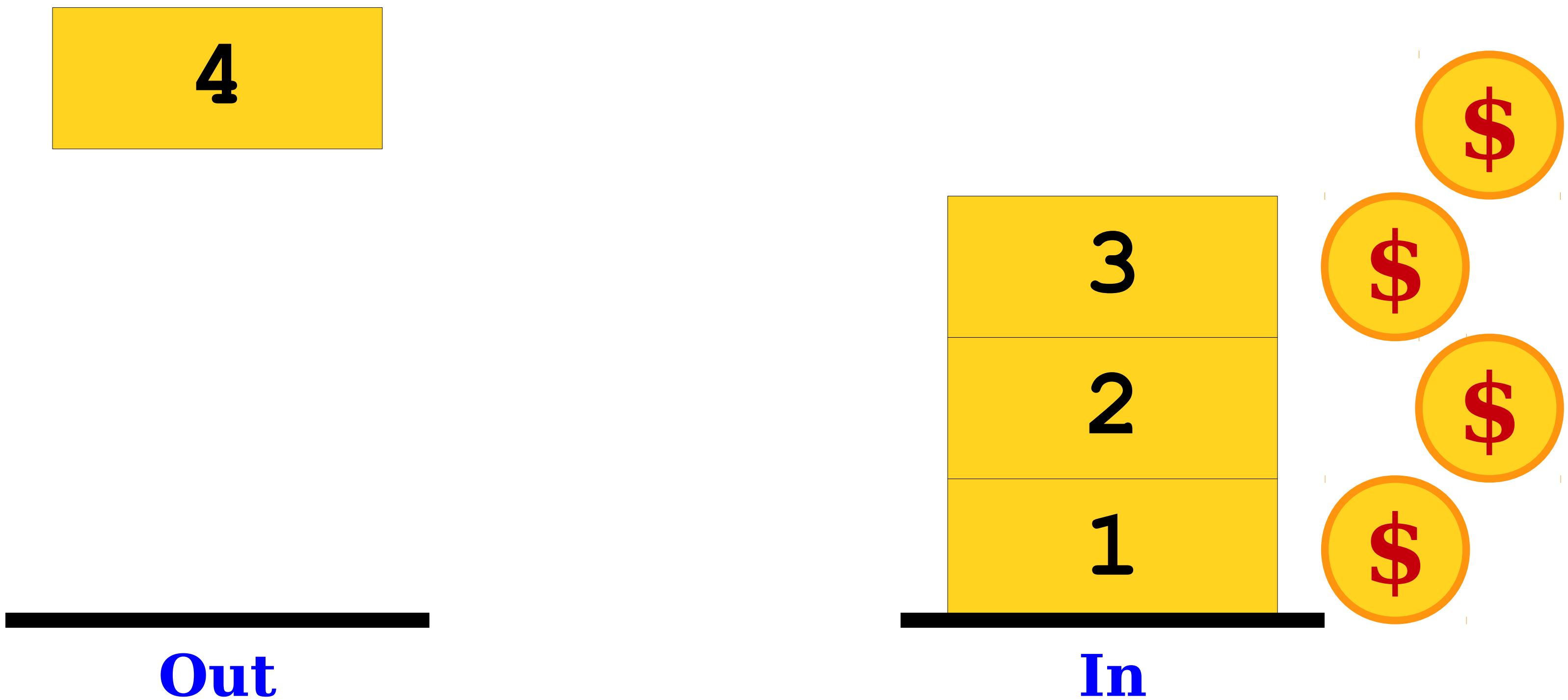
The Two-Stack Queue



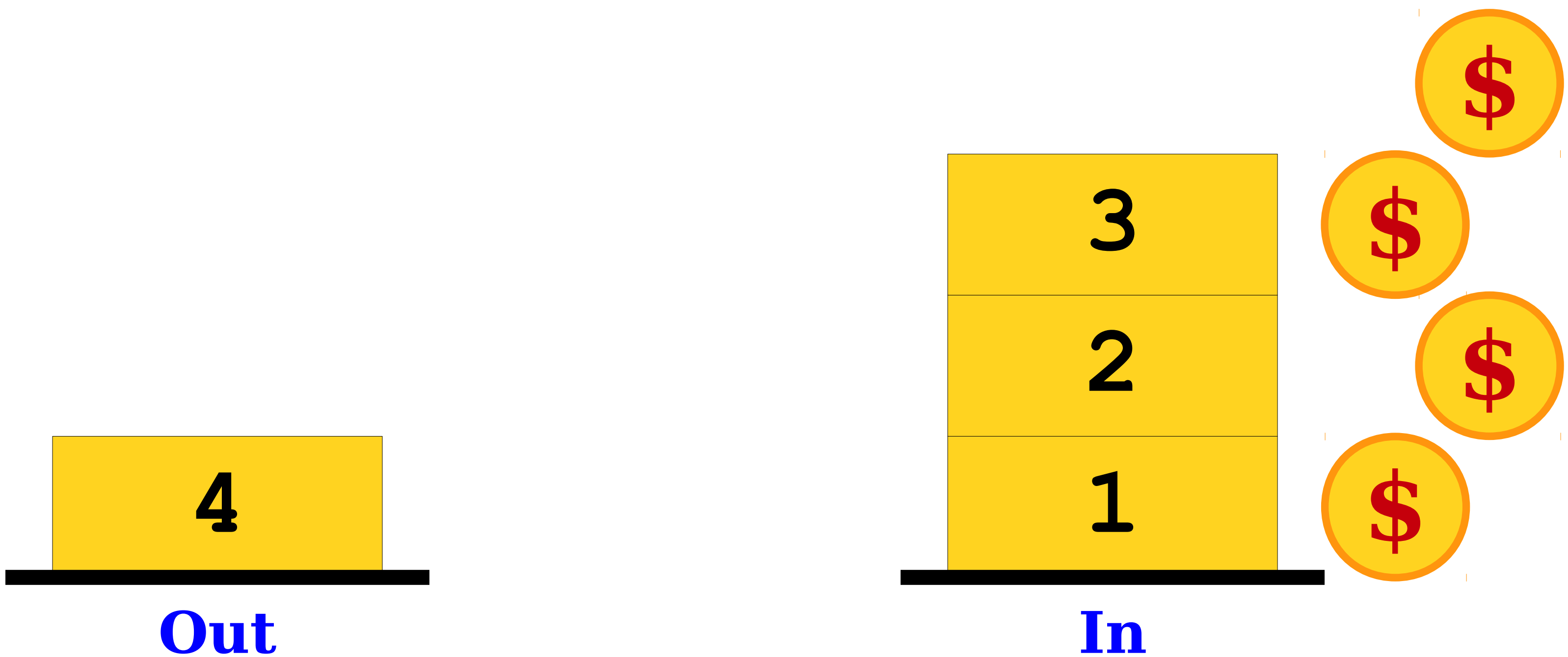
The Two-Stack Queue



The Two-Stack Queue

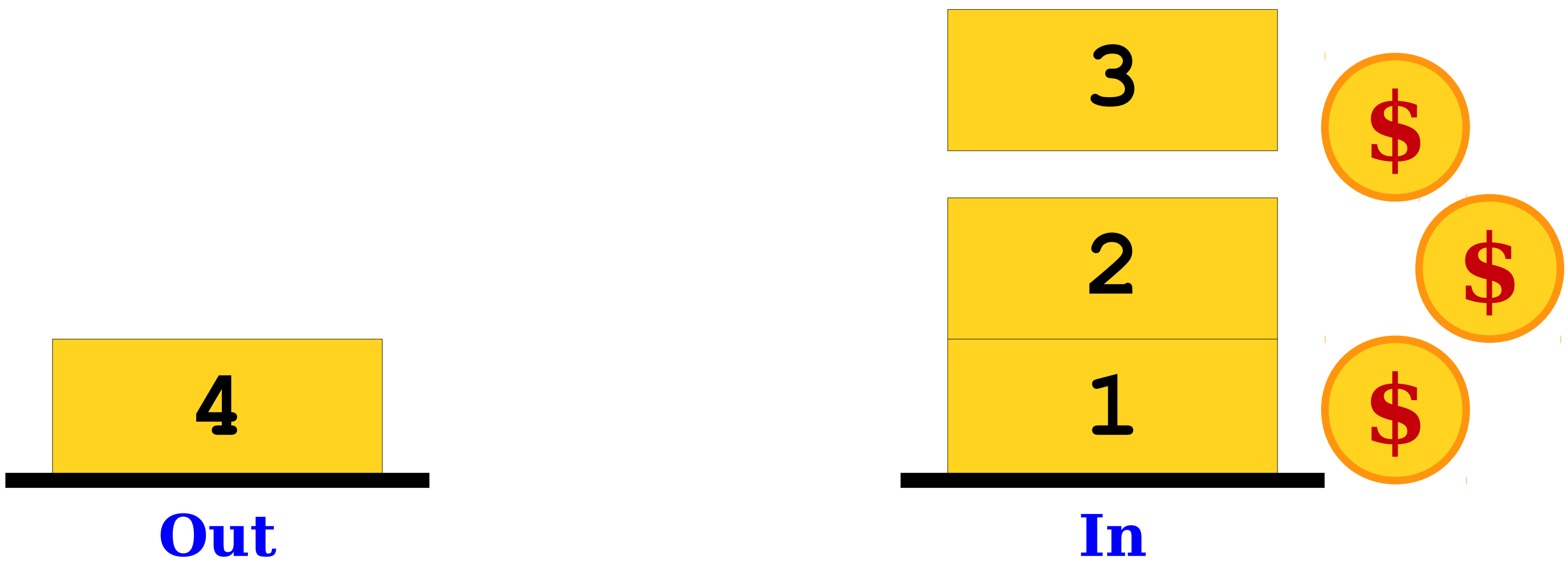


The Two-Stack Queue

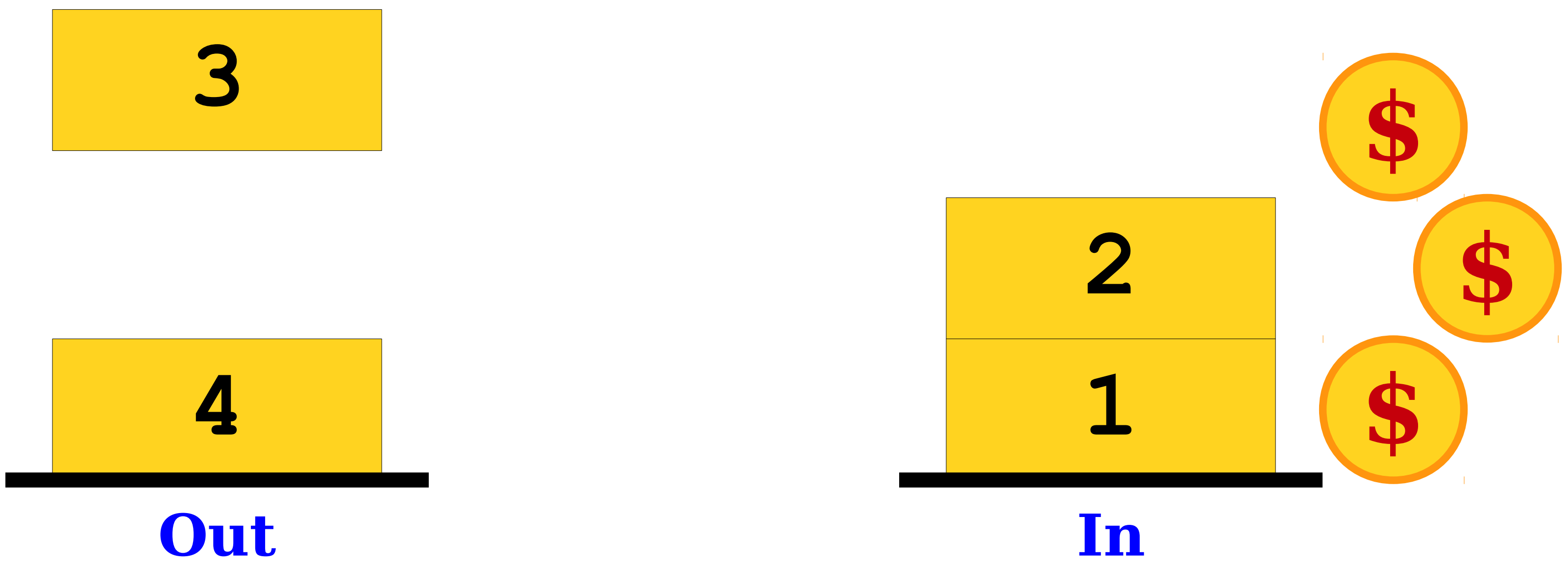


The Two-Stack Queue

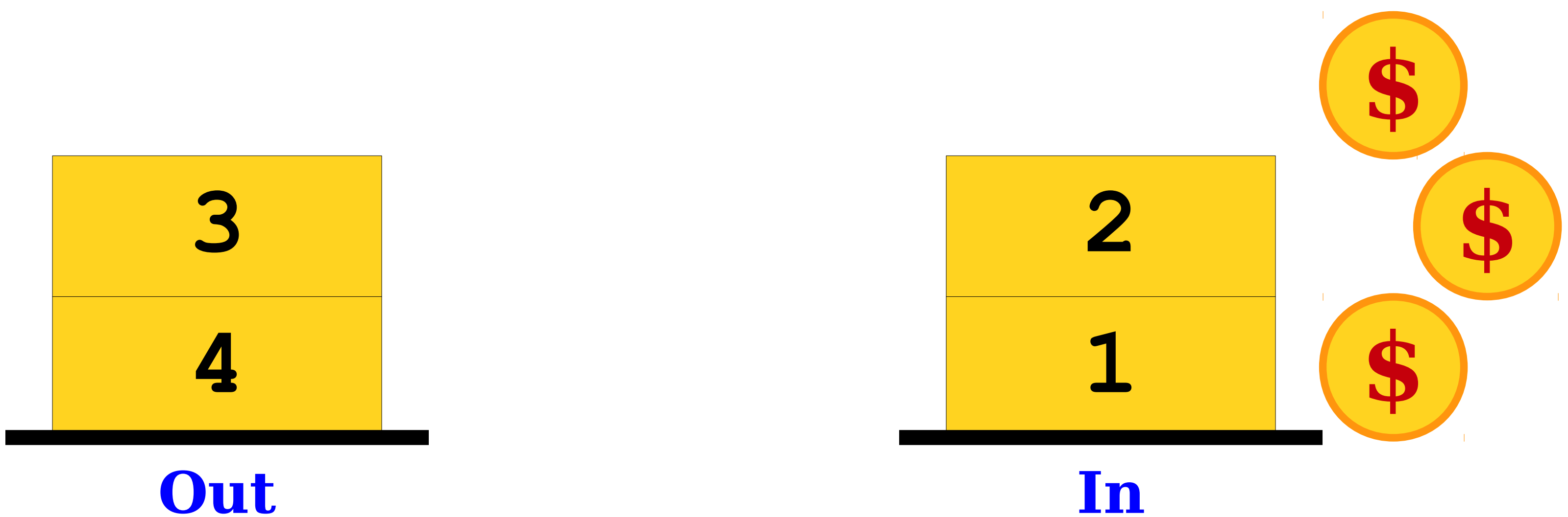
The Two-Stack Queue



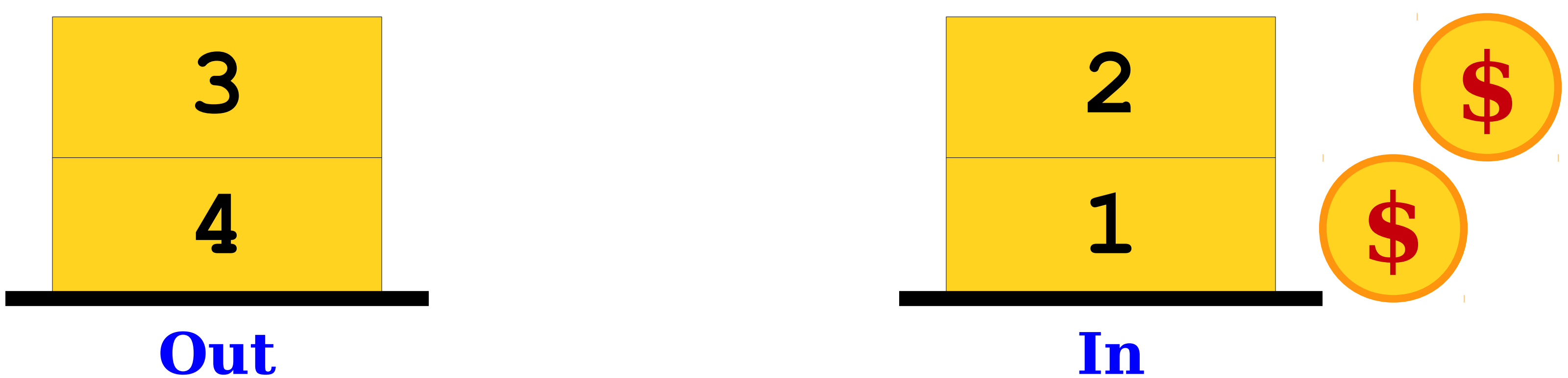
The Two-Stack Queue



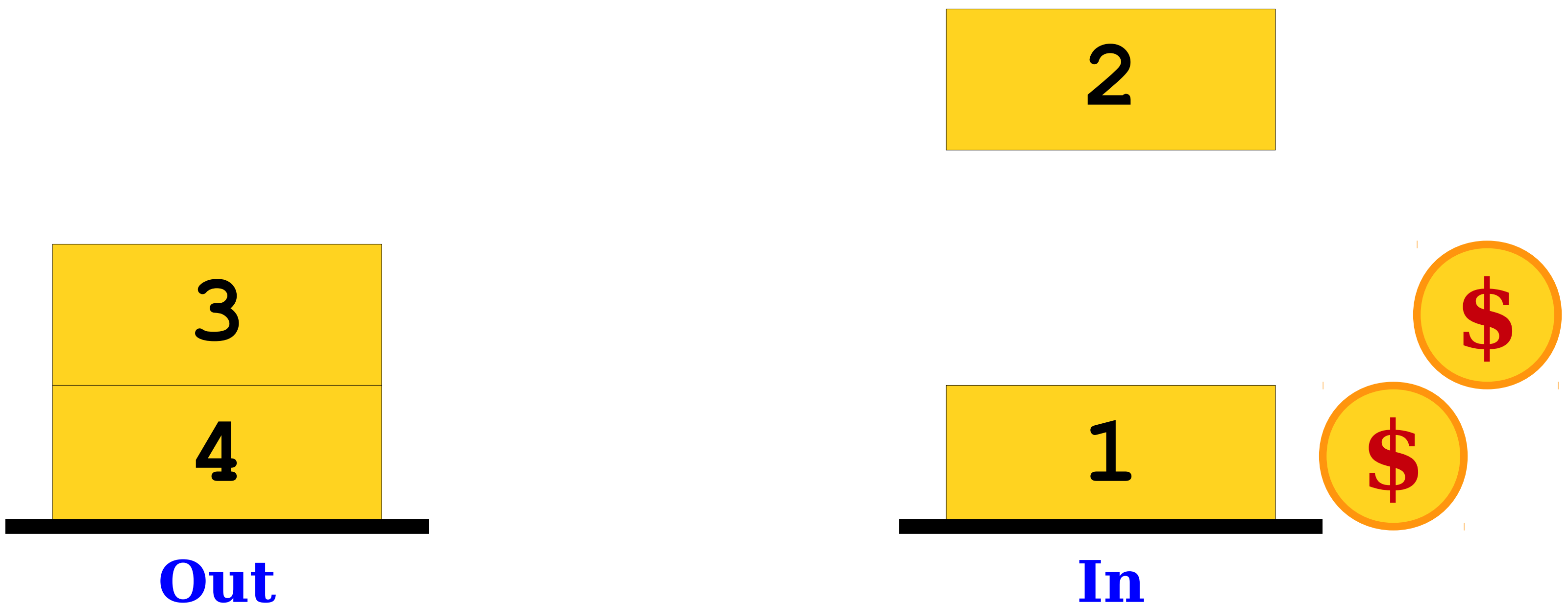
The Two-Stack Queue



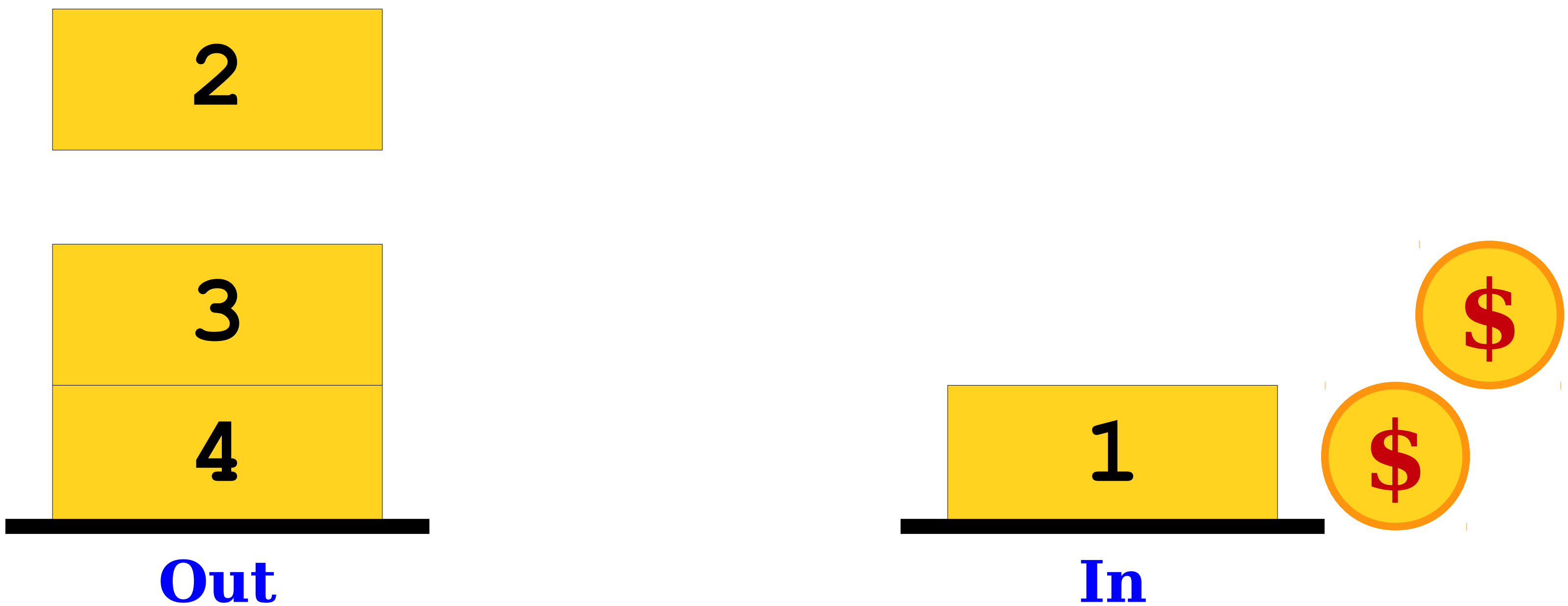
The Two-Stack Queue



The Two-Stack Queue

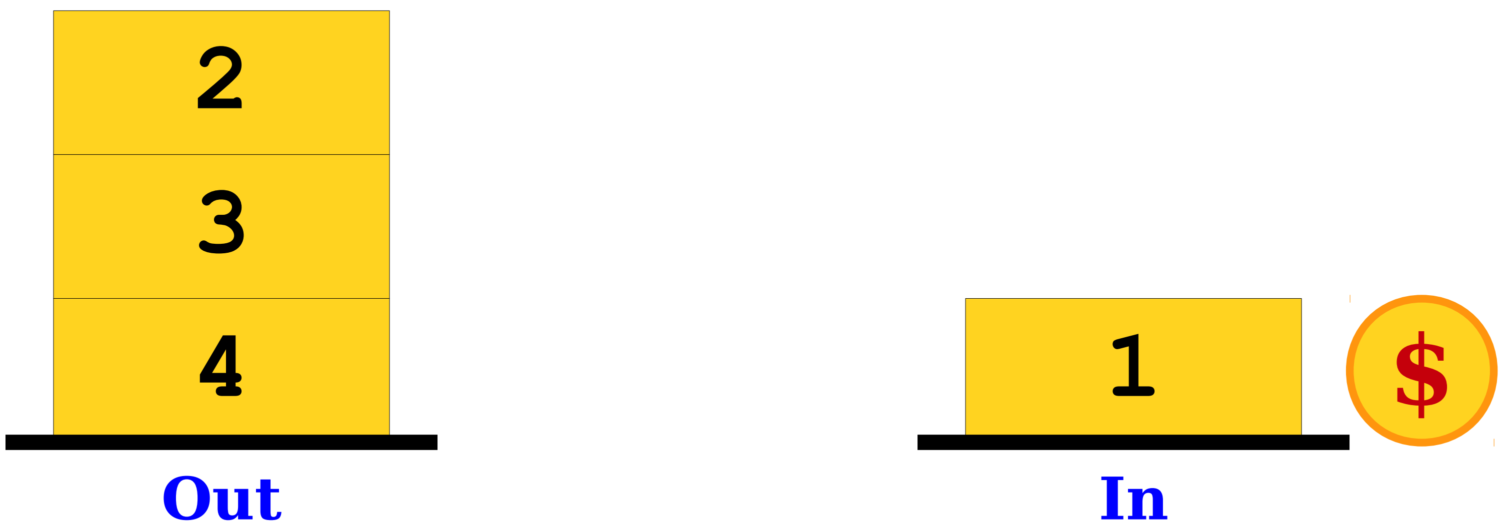


The Two-Stack Queue

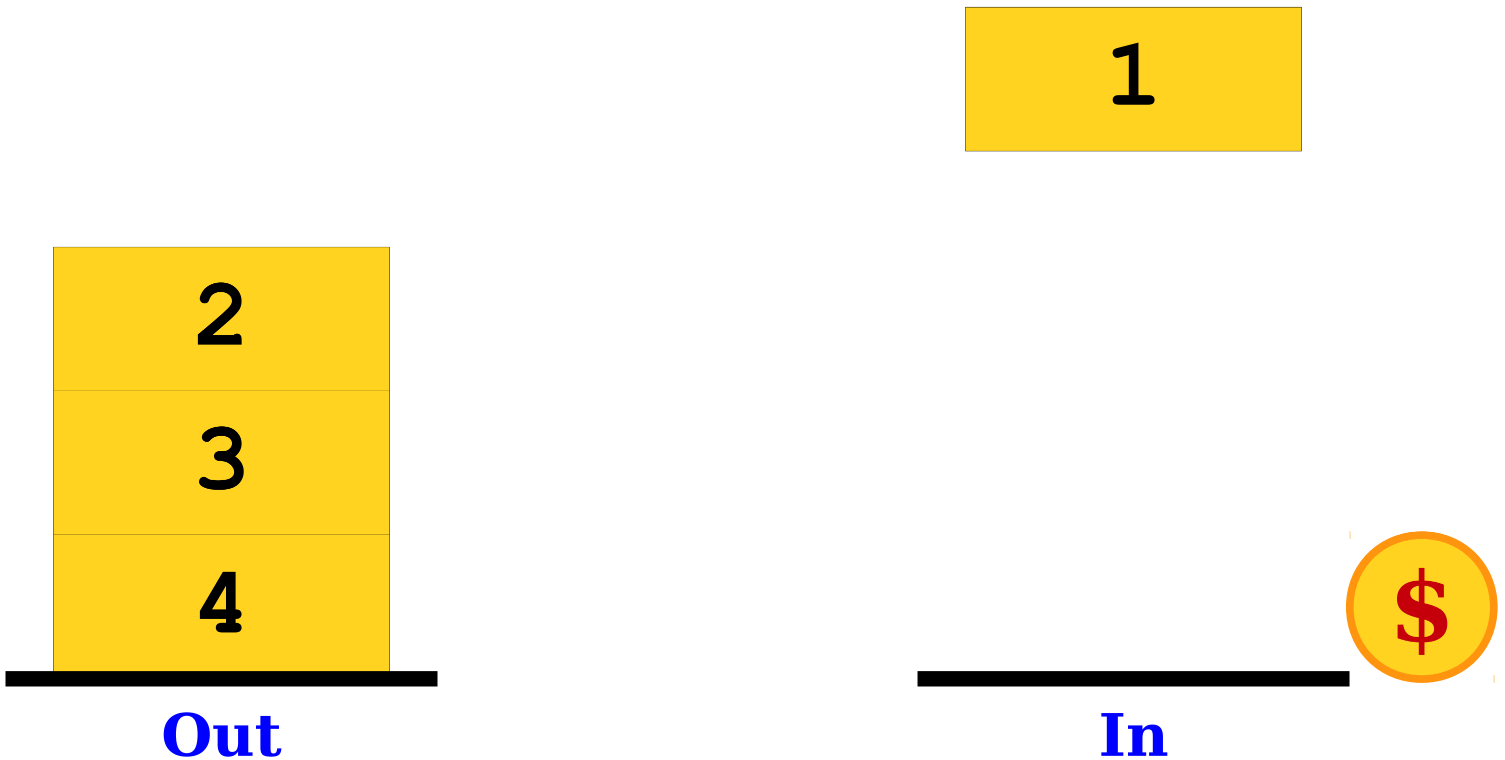


The Two-Stack Queue

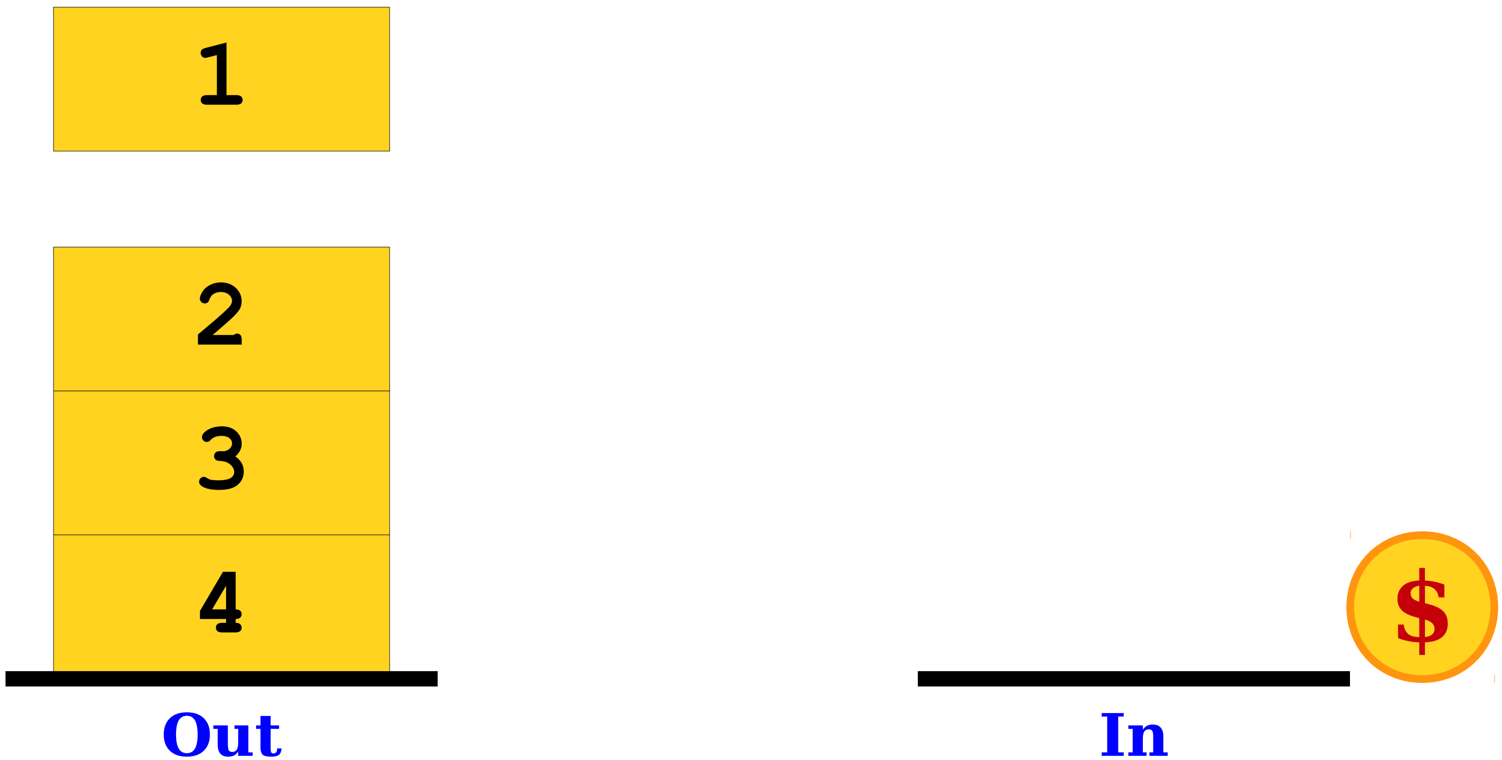
The Two-Stack Queue



The Two-Stack Queue

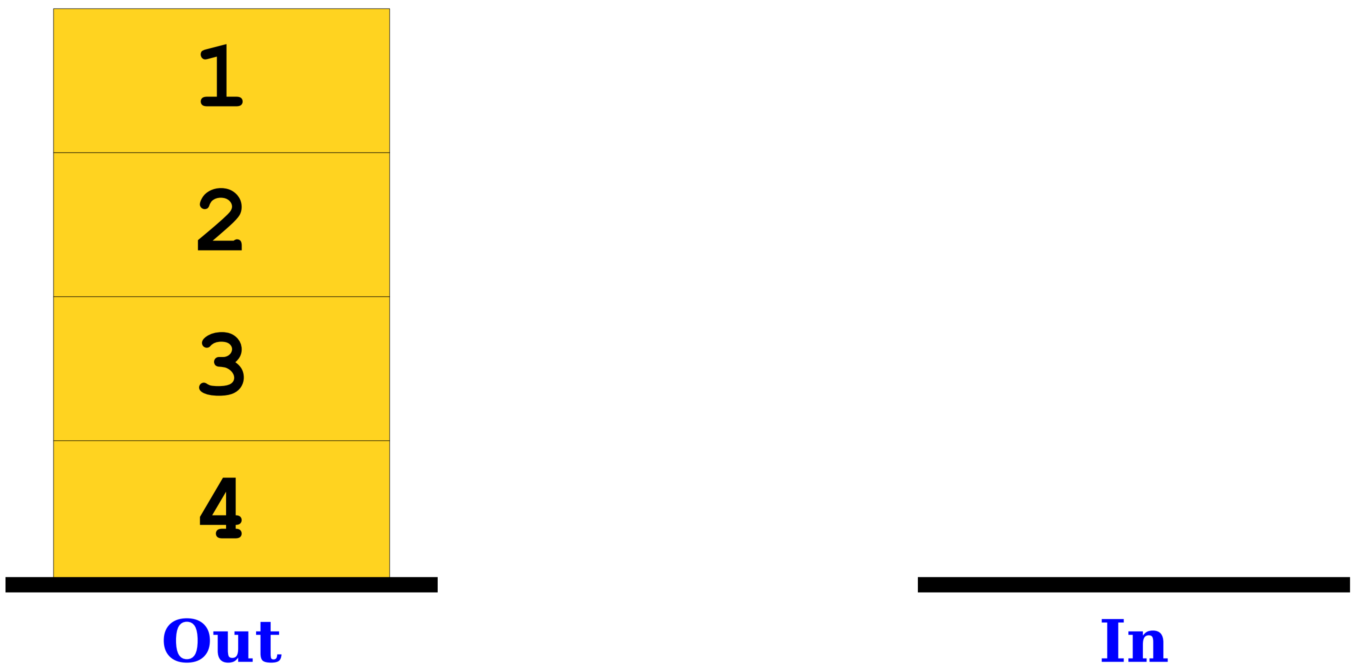


The Two-Stack Queue

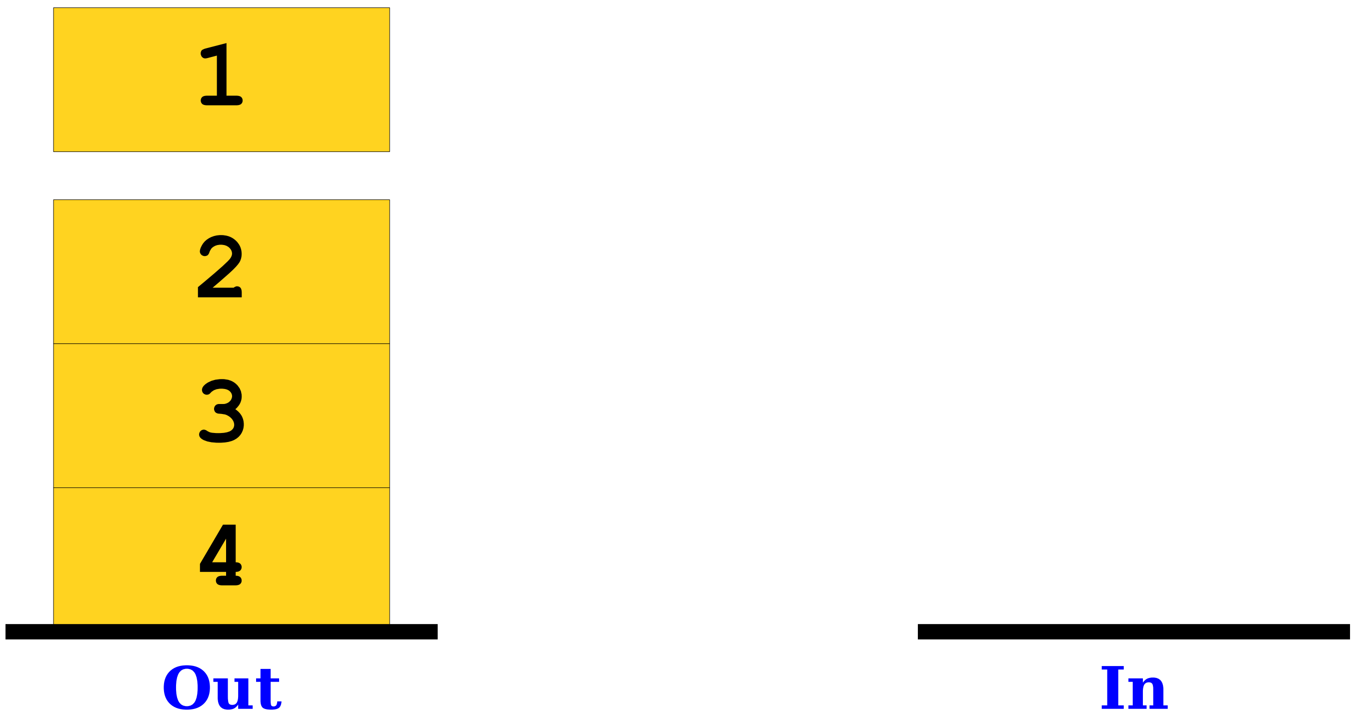


The Two-Stack Queue

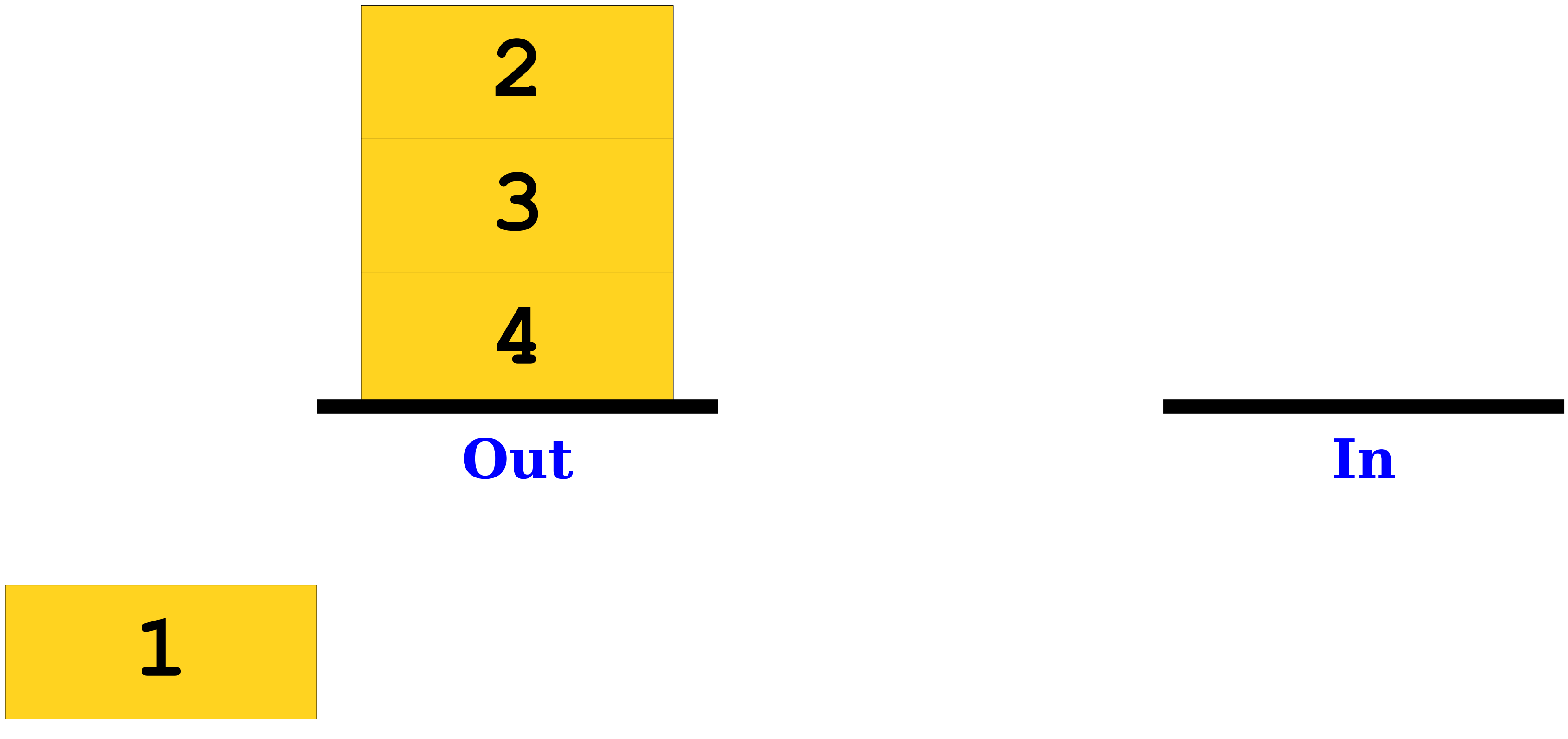
The Two-Stack Queue



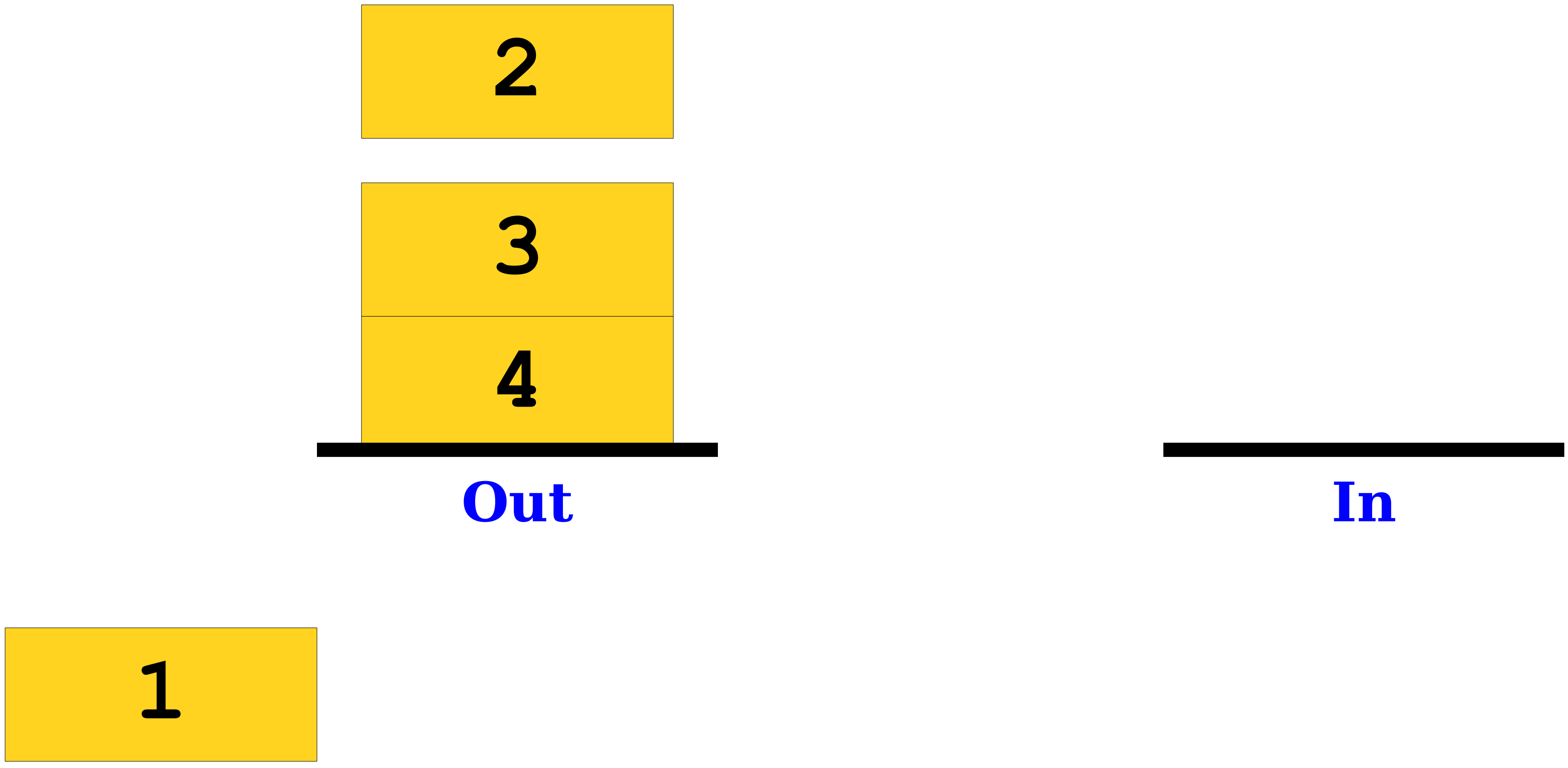
The Two-Stack Queue



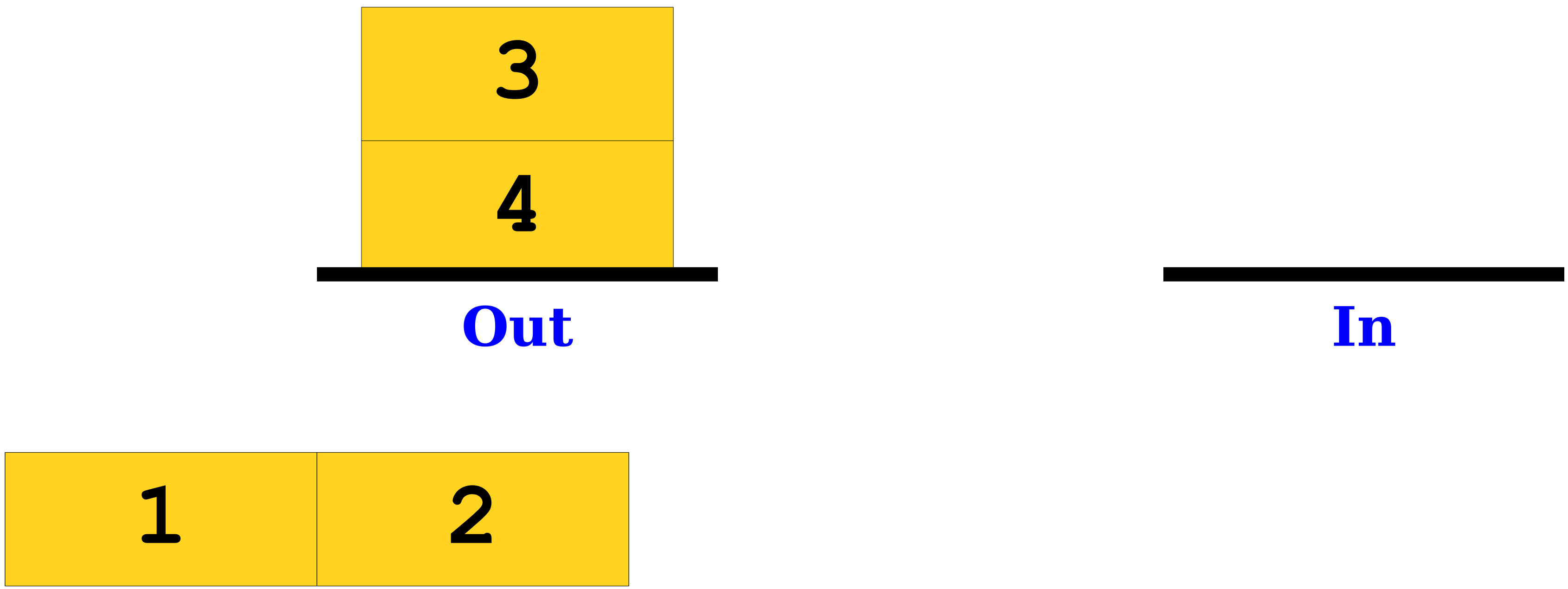
The Two-Stack Queue



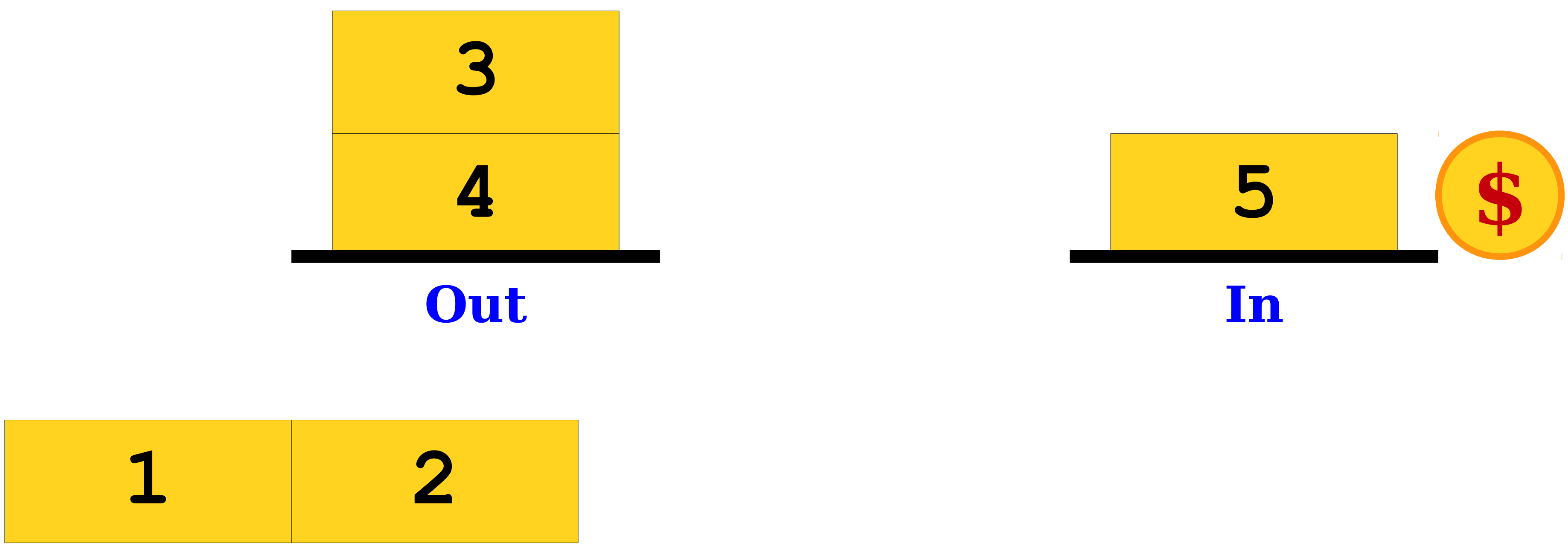
The Two-Stack Queue



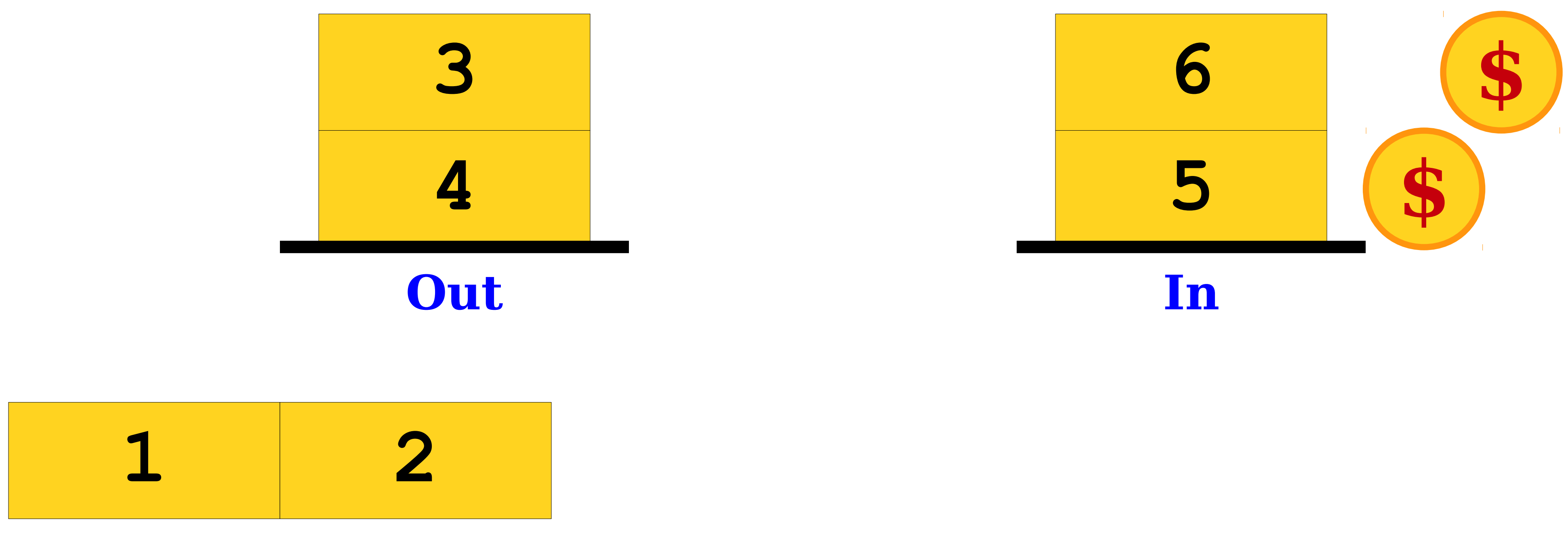
The Two-Stack Queue



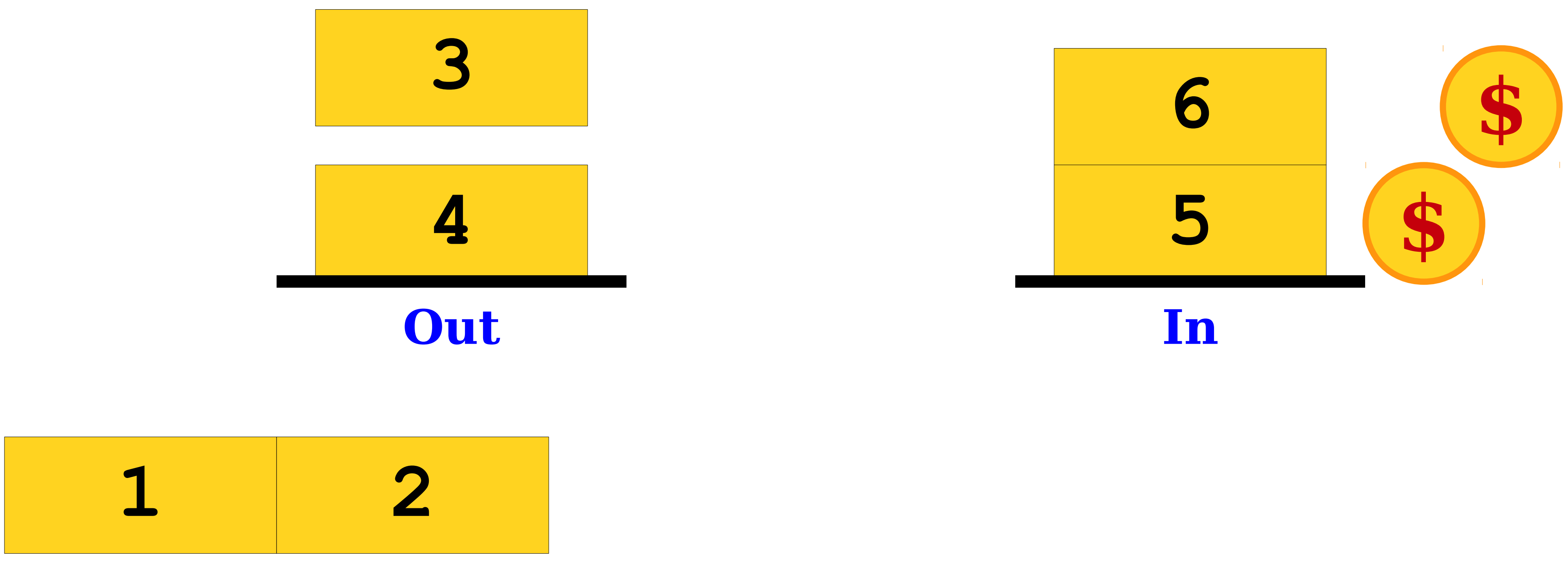
The Two-Stack Queue



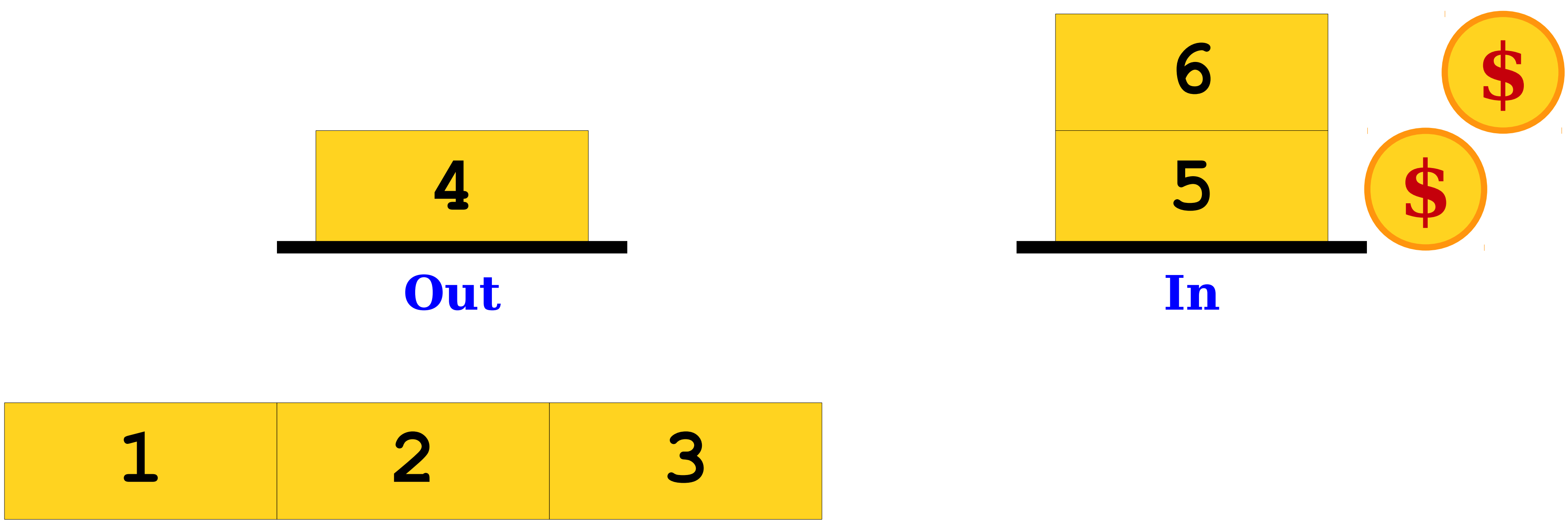
The Two-Stack Queue



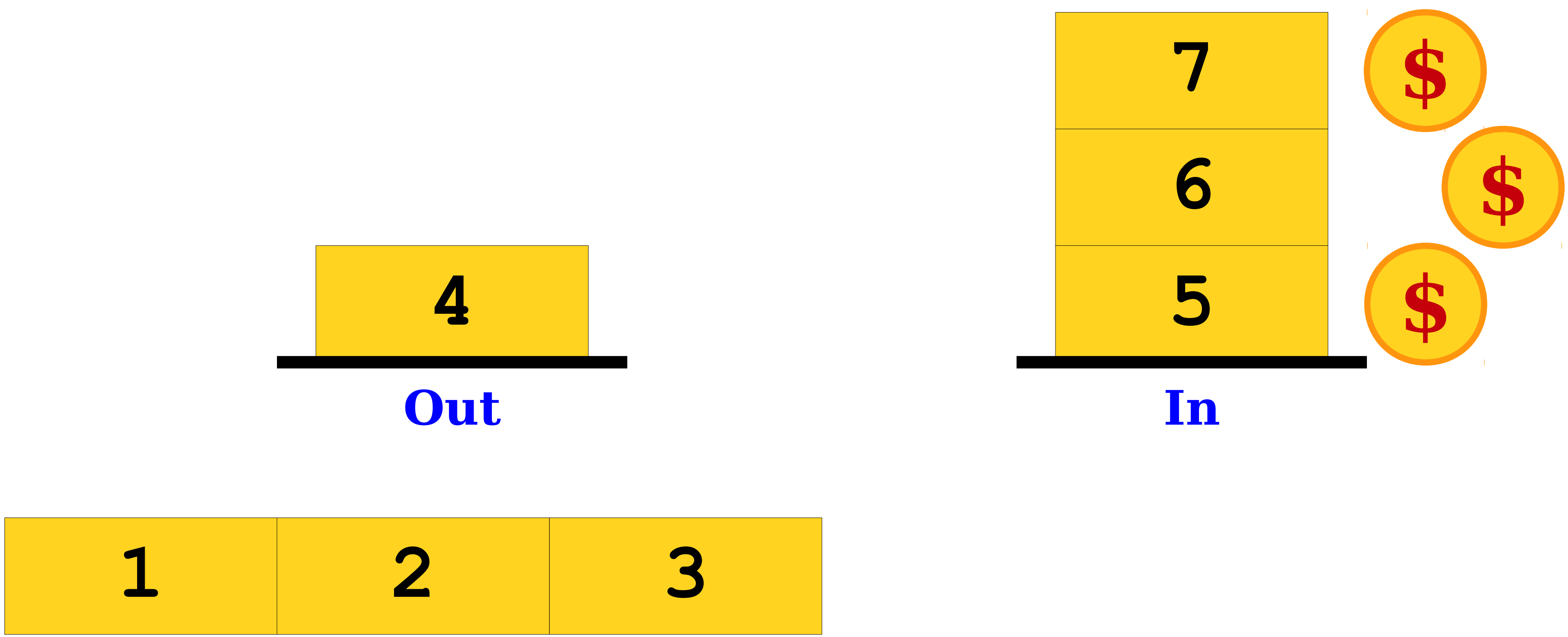
The Two-Stack Queue



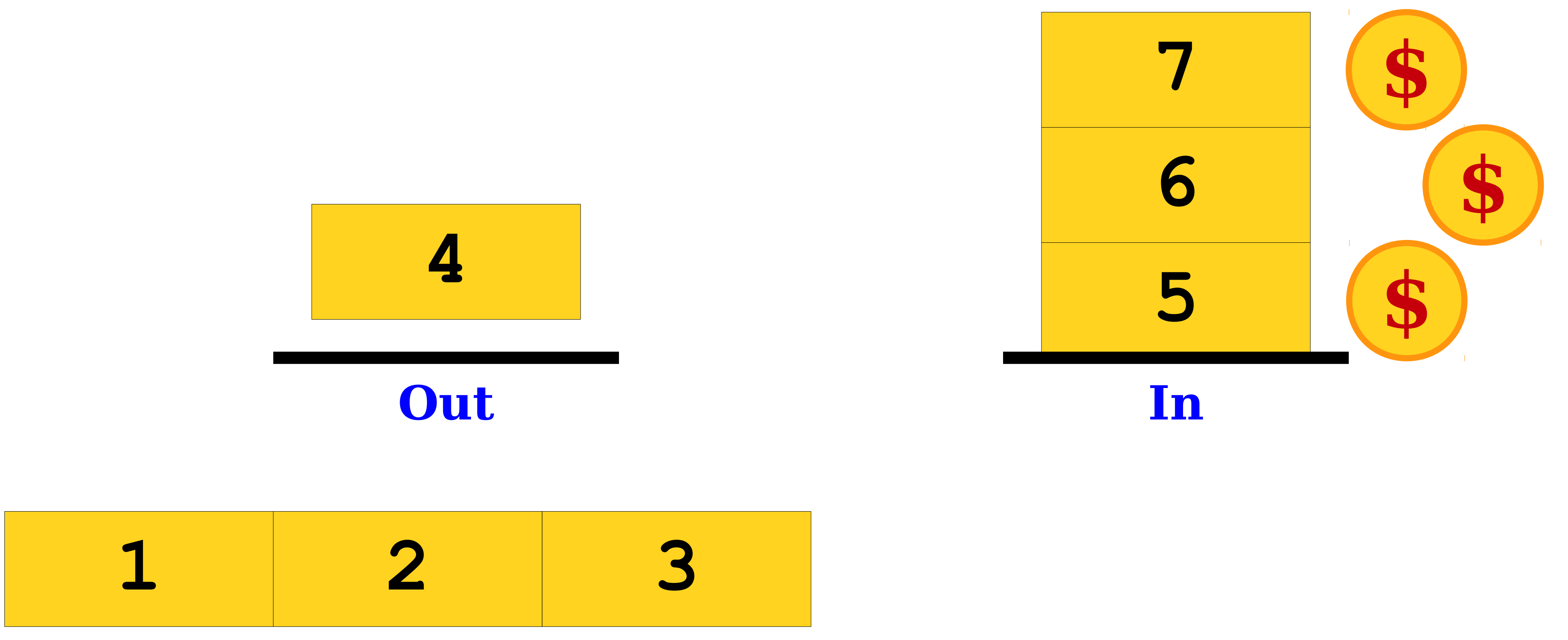
The Two-Stack Queue



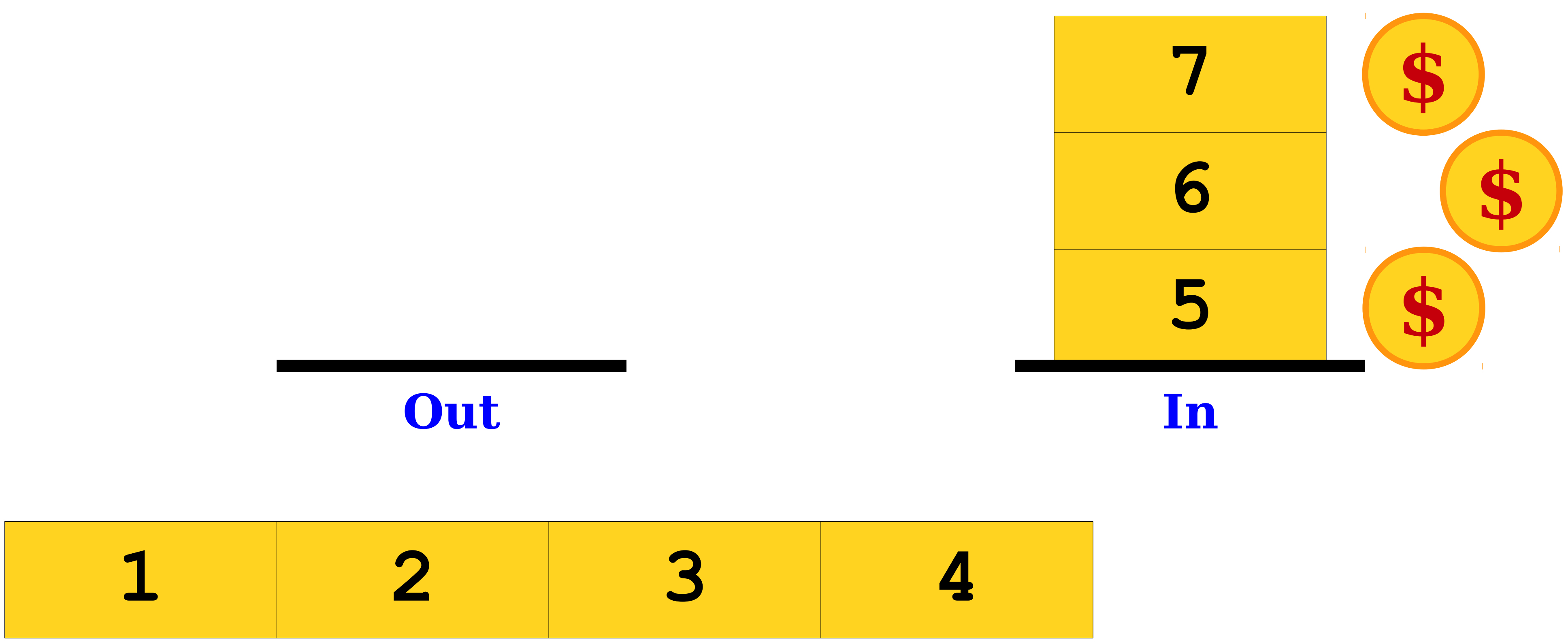
The Two-Stack Queue



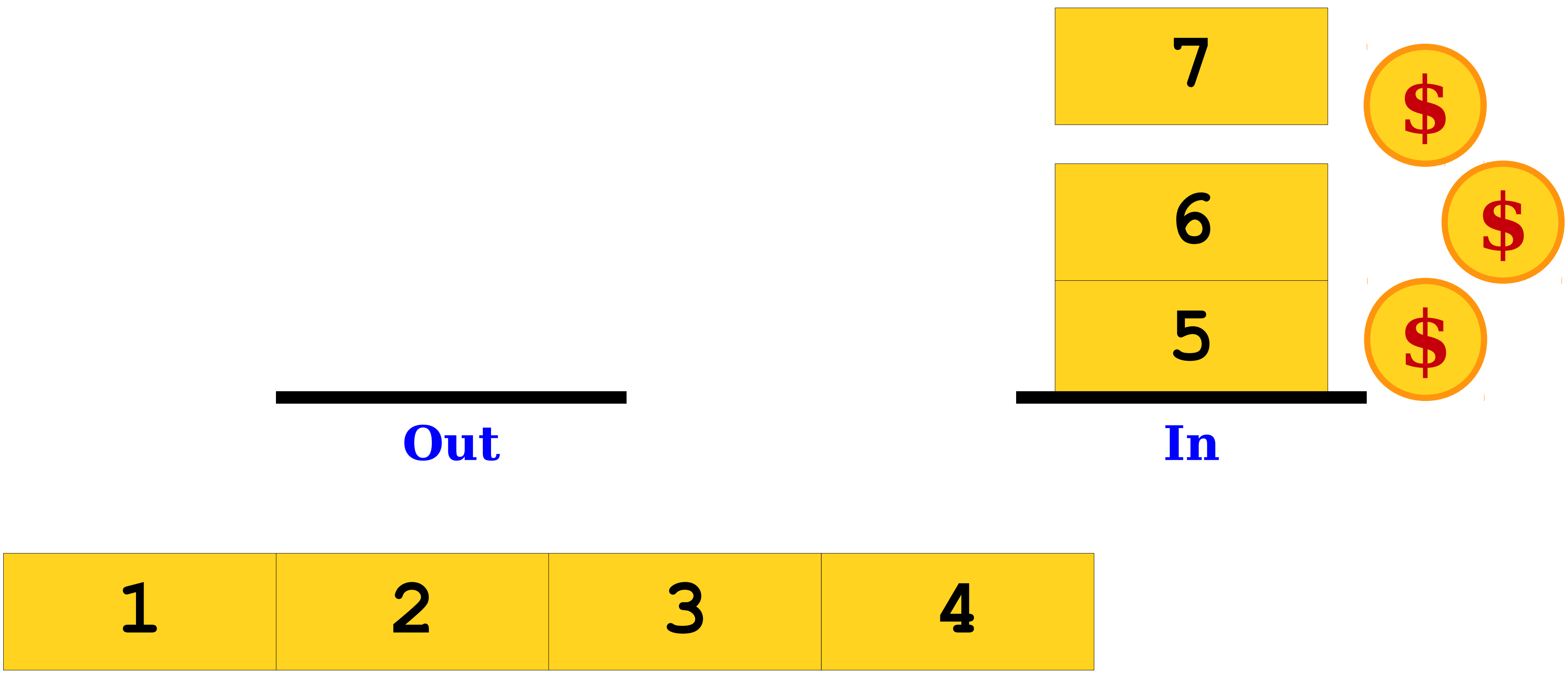
The Two-Stack Queue



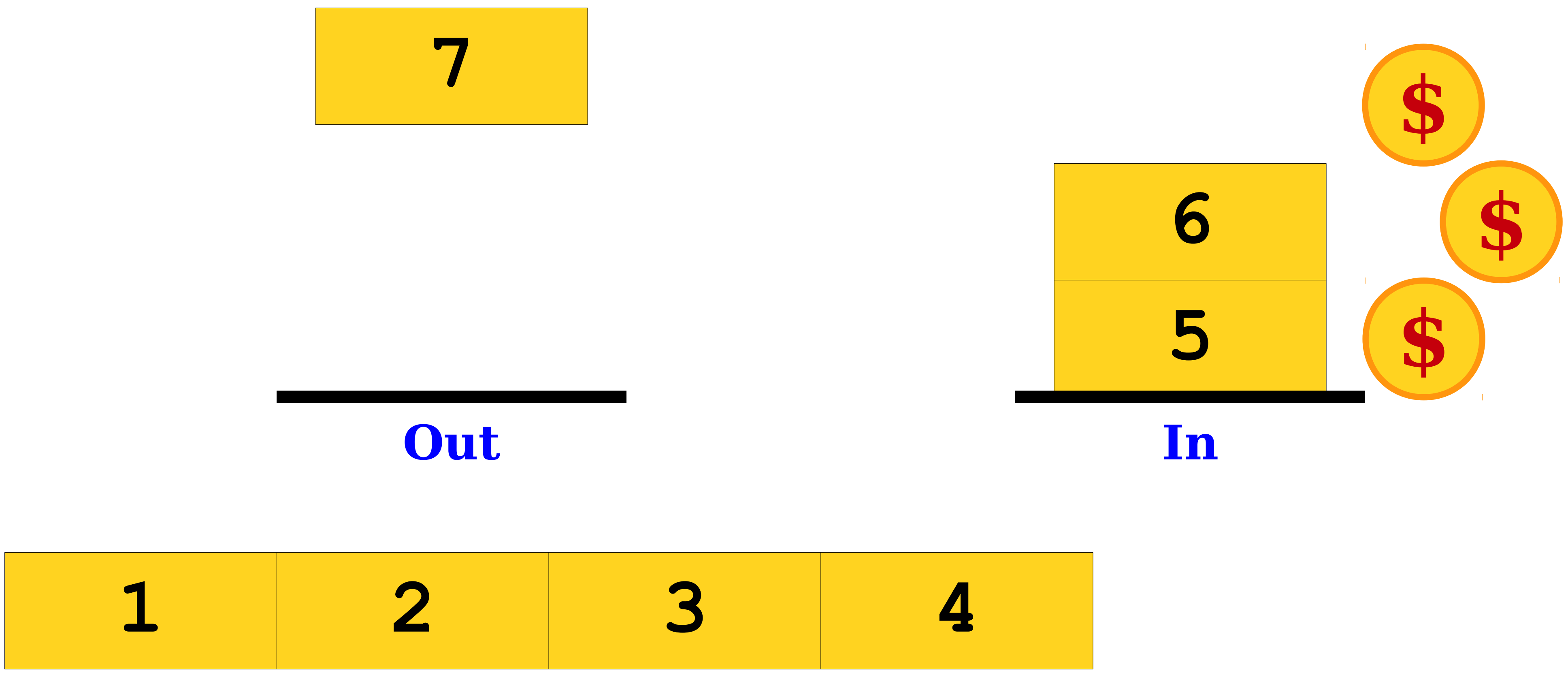
The Two-Stack Queue



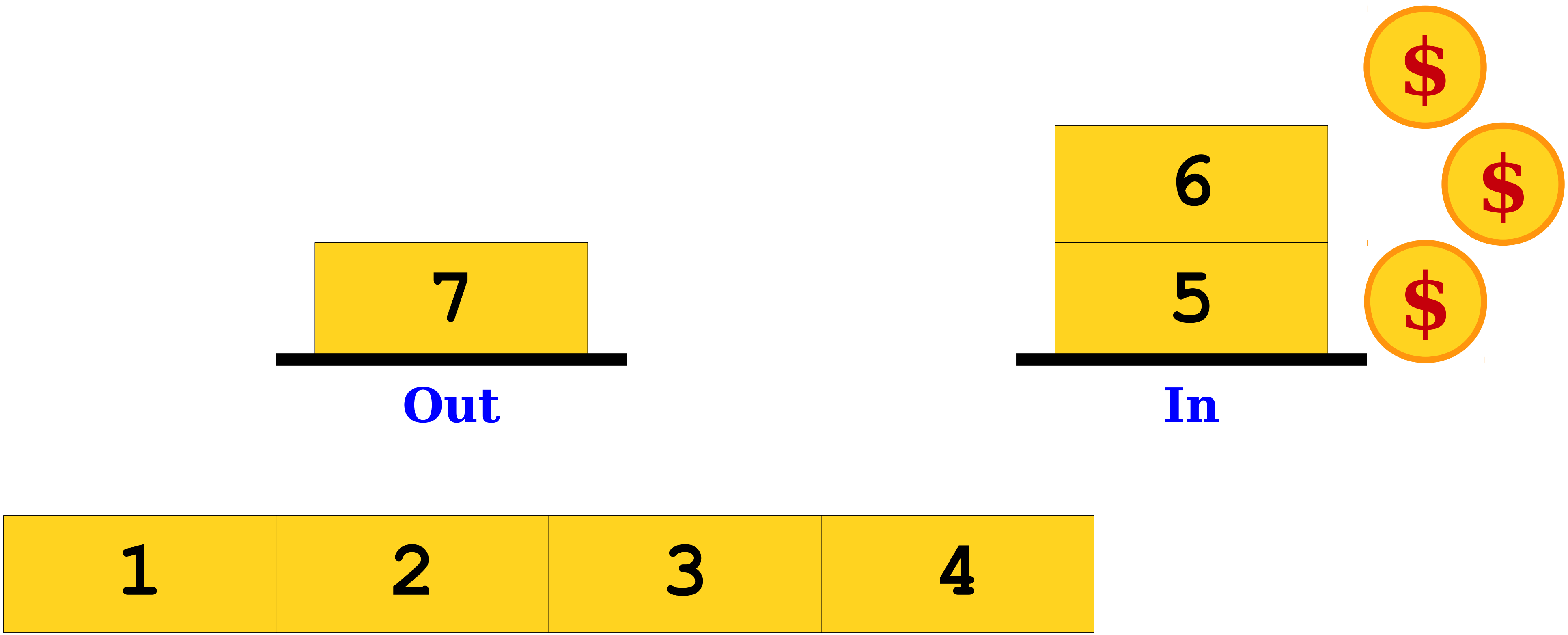
The Two-Stack Queue



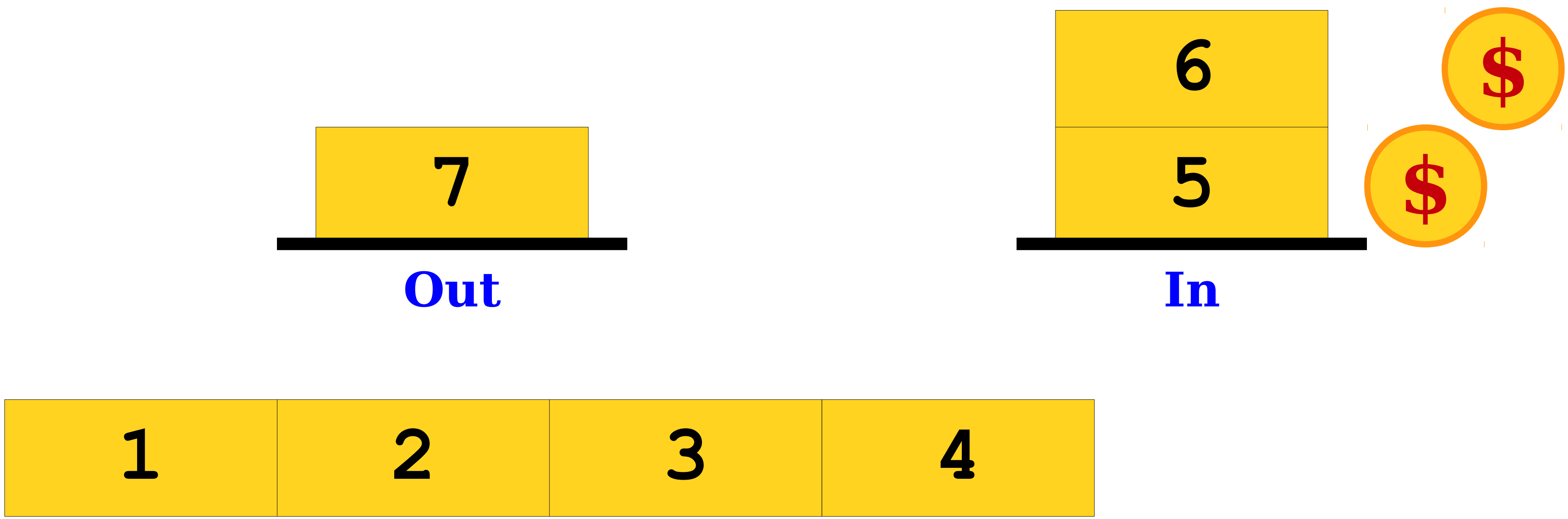
The Two-Stack Queue



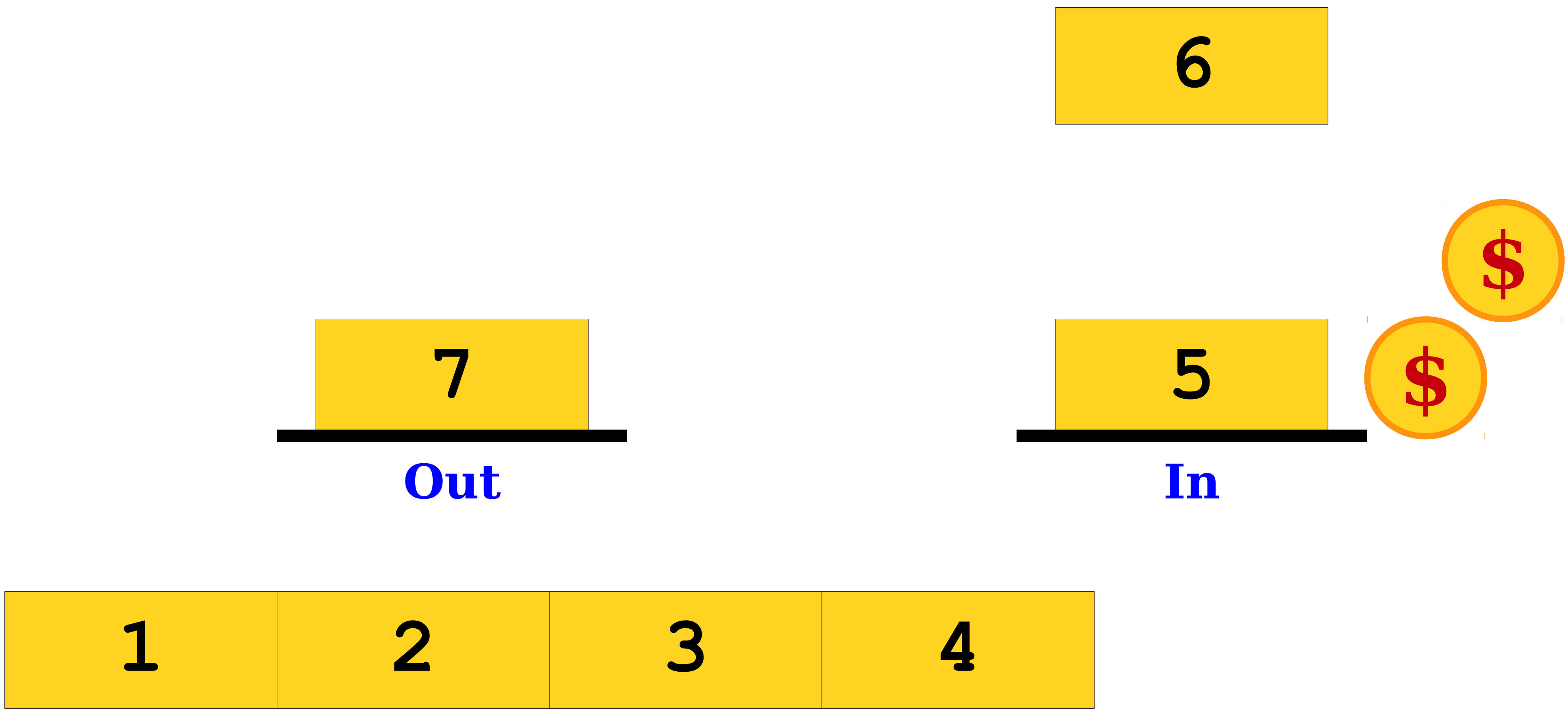
The Two-Stack Queue



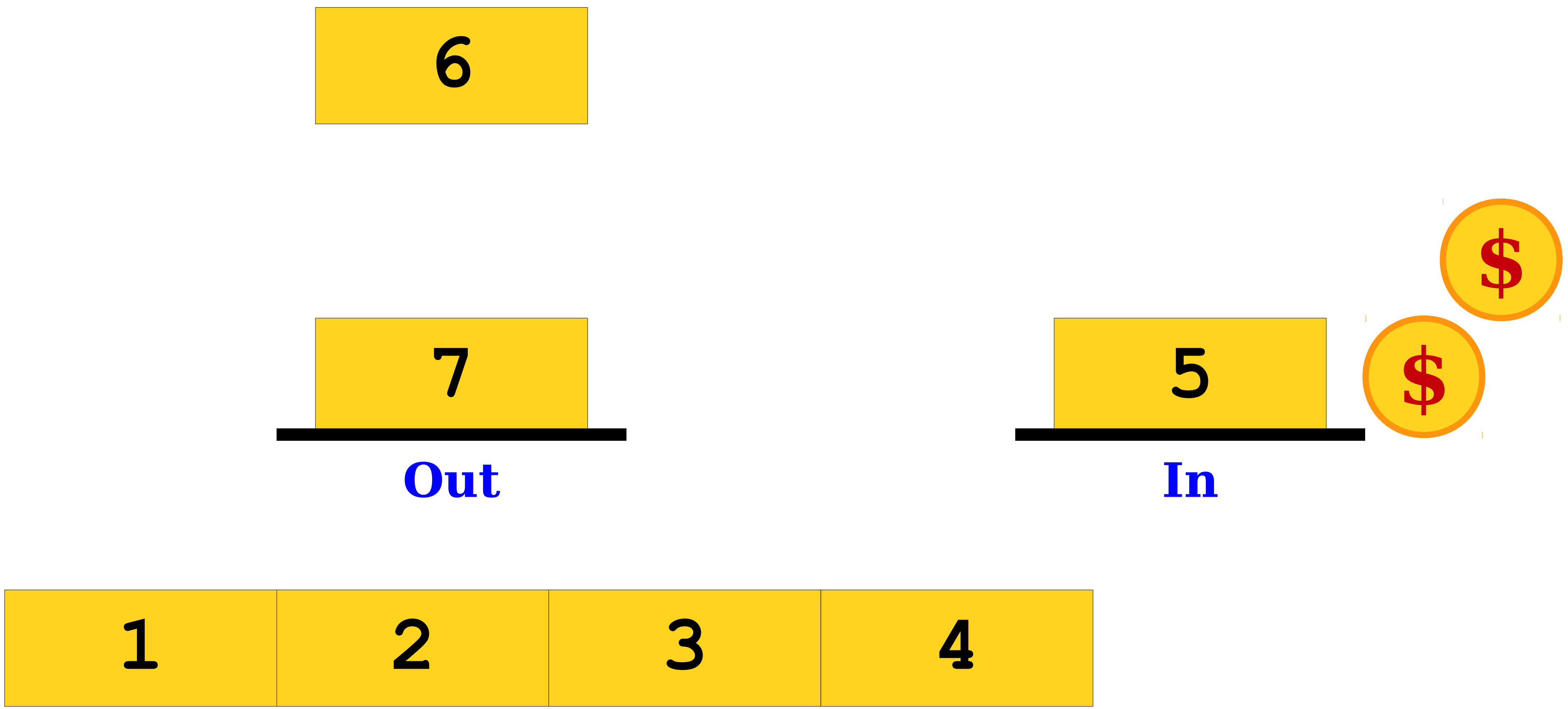
The Two-Stack Queue



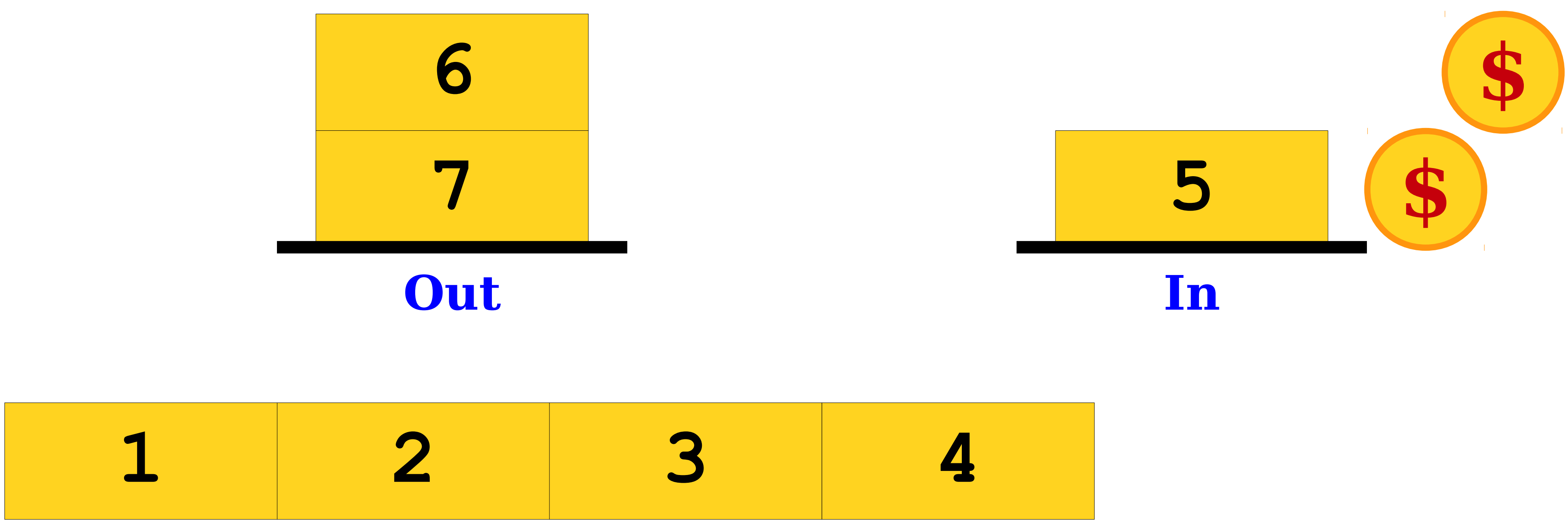
The Two-Stack Queue



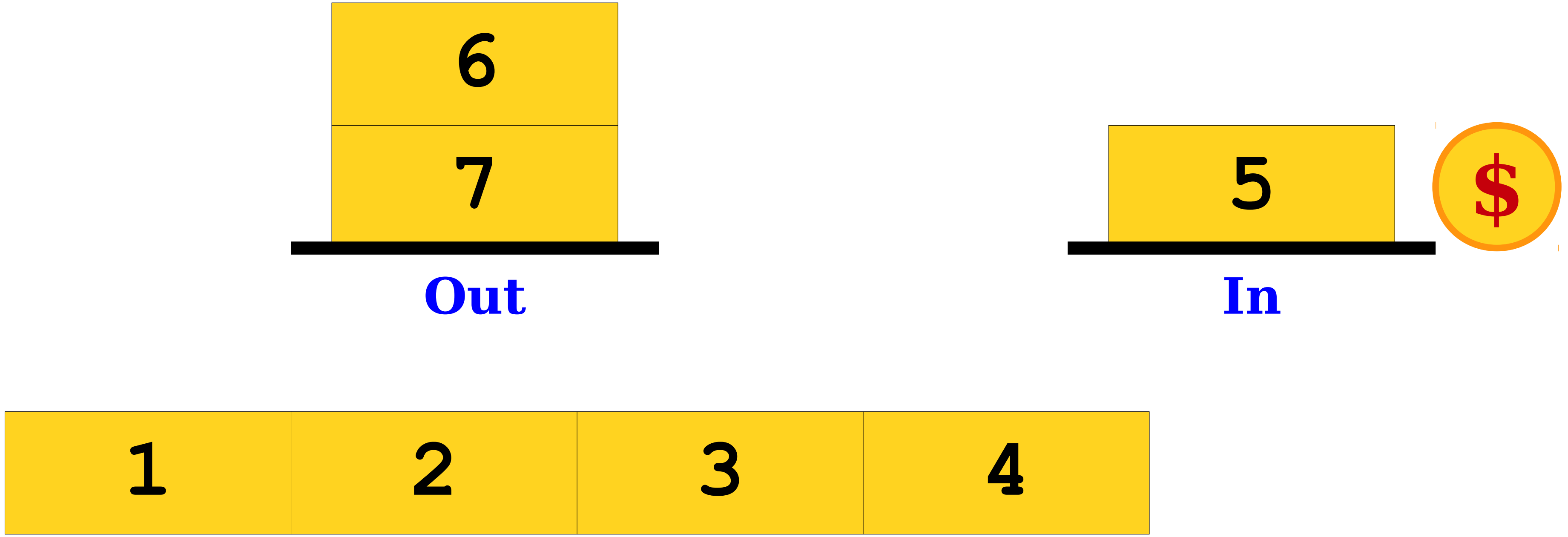
The Two-Stack Queue



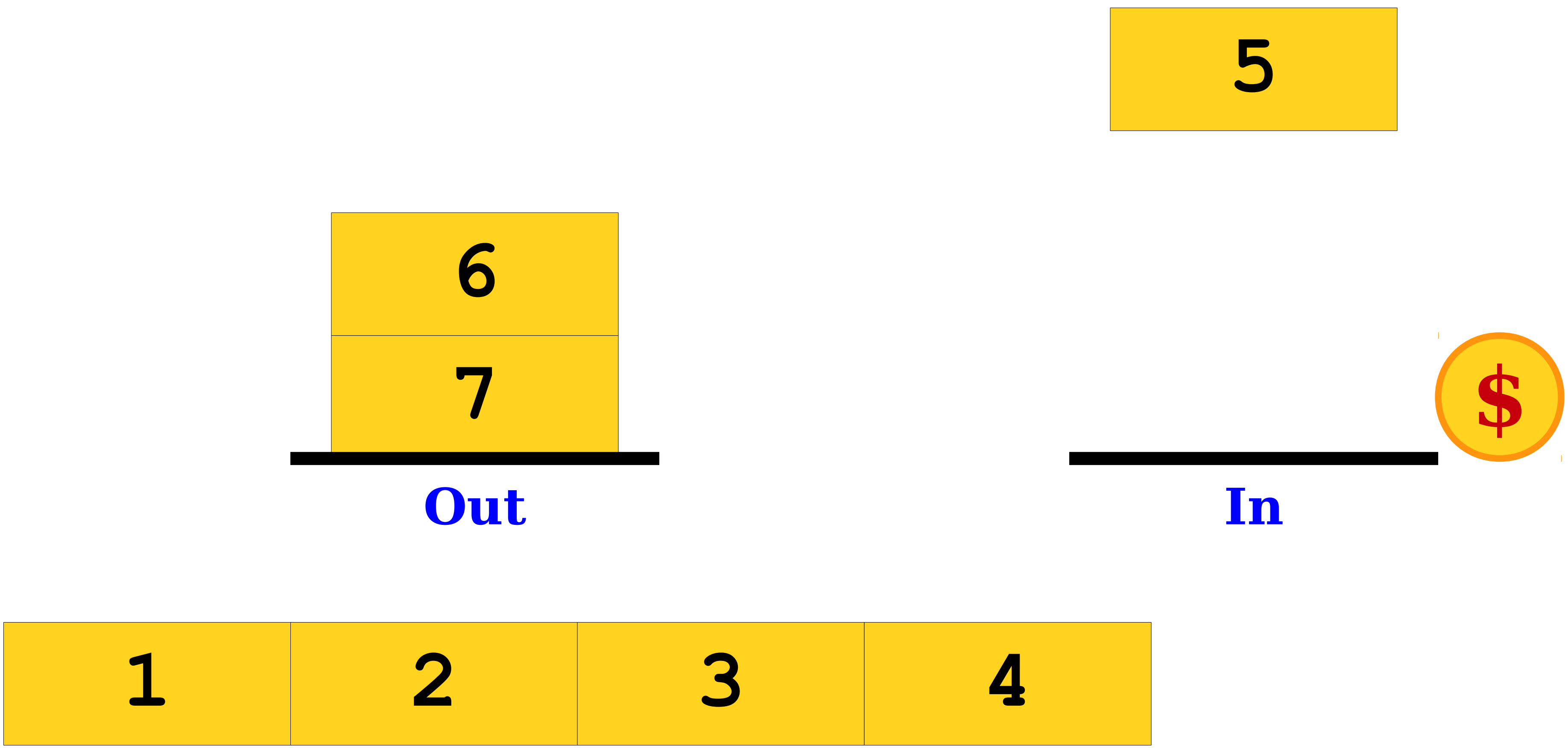
The Two-Stack Queue



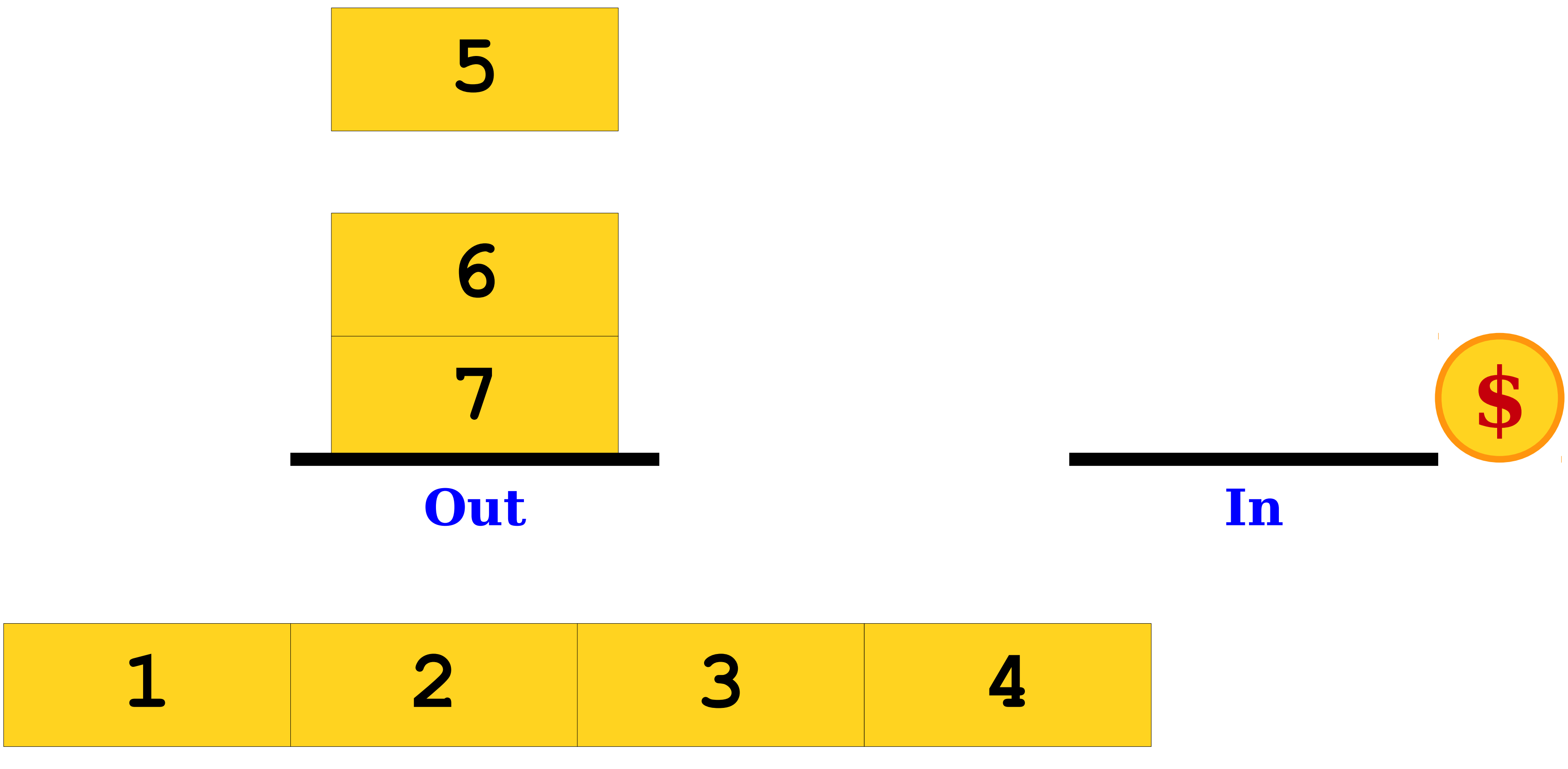
The Two-Stack Queue



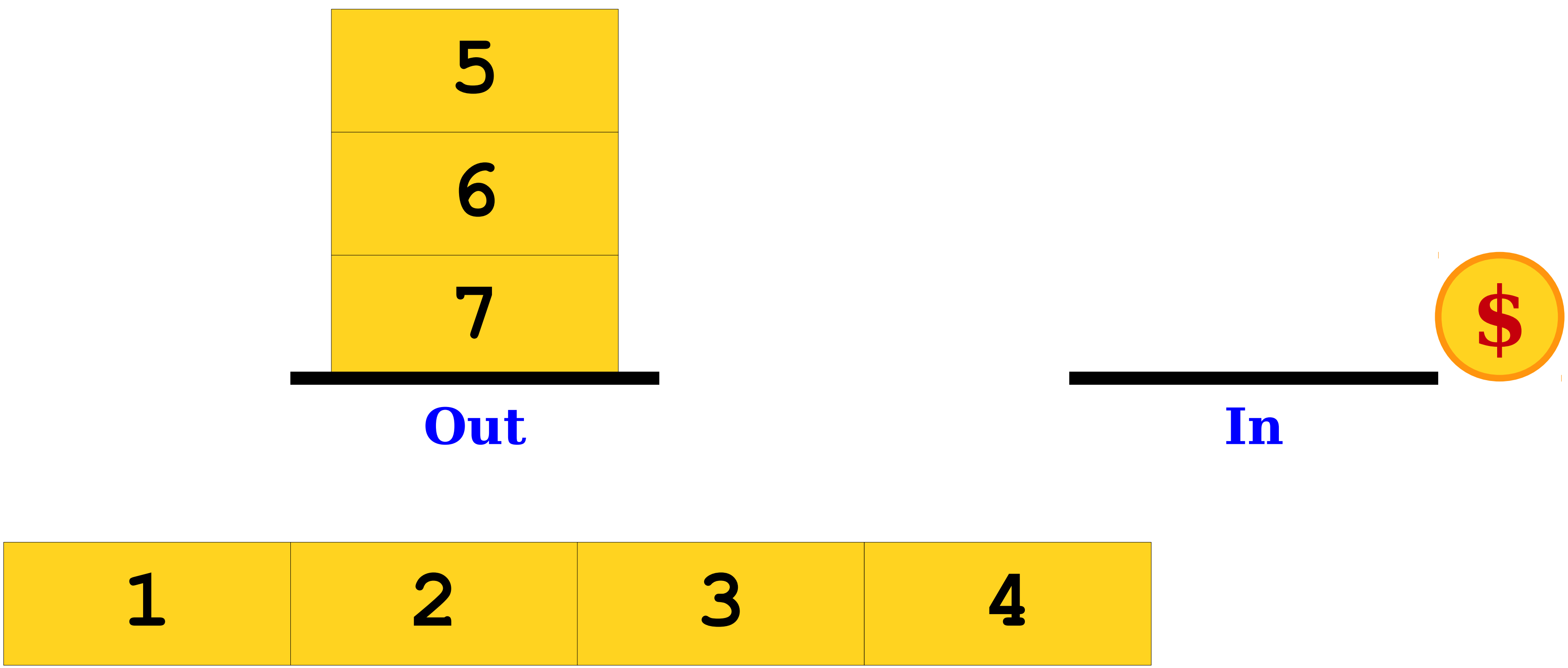
The Two-Stack Queue



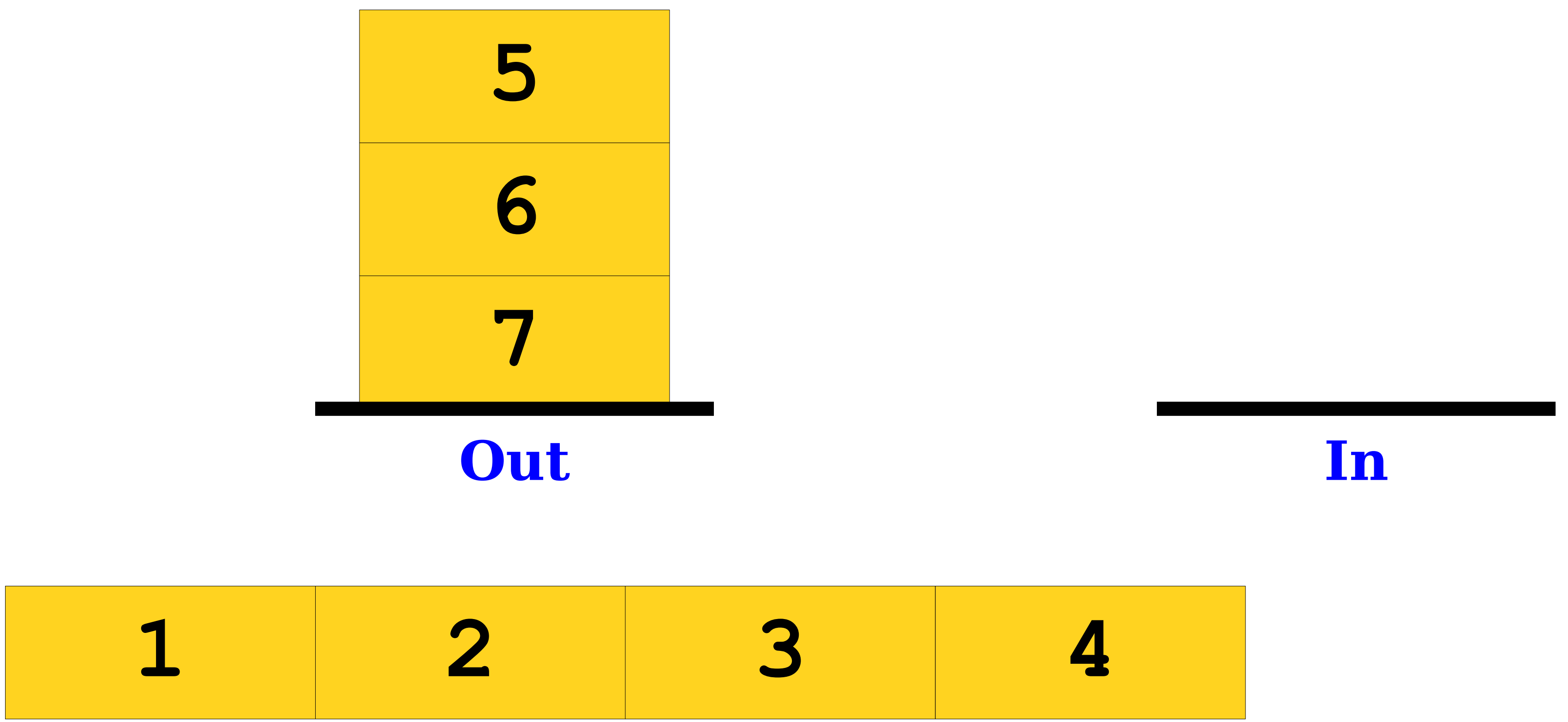
The Two-Stack Queue



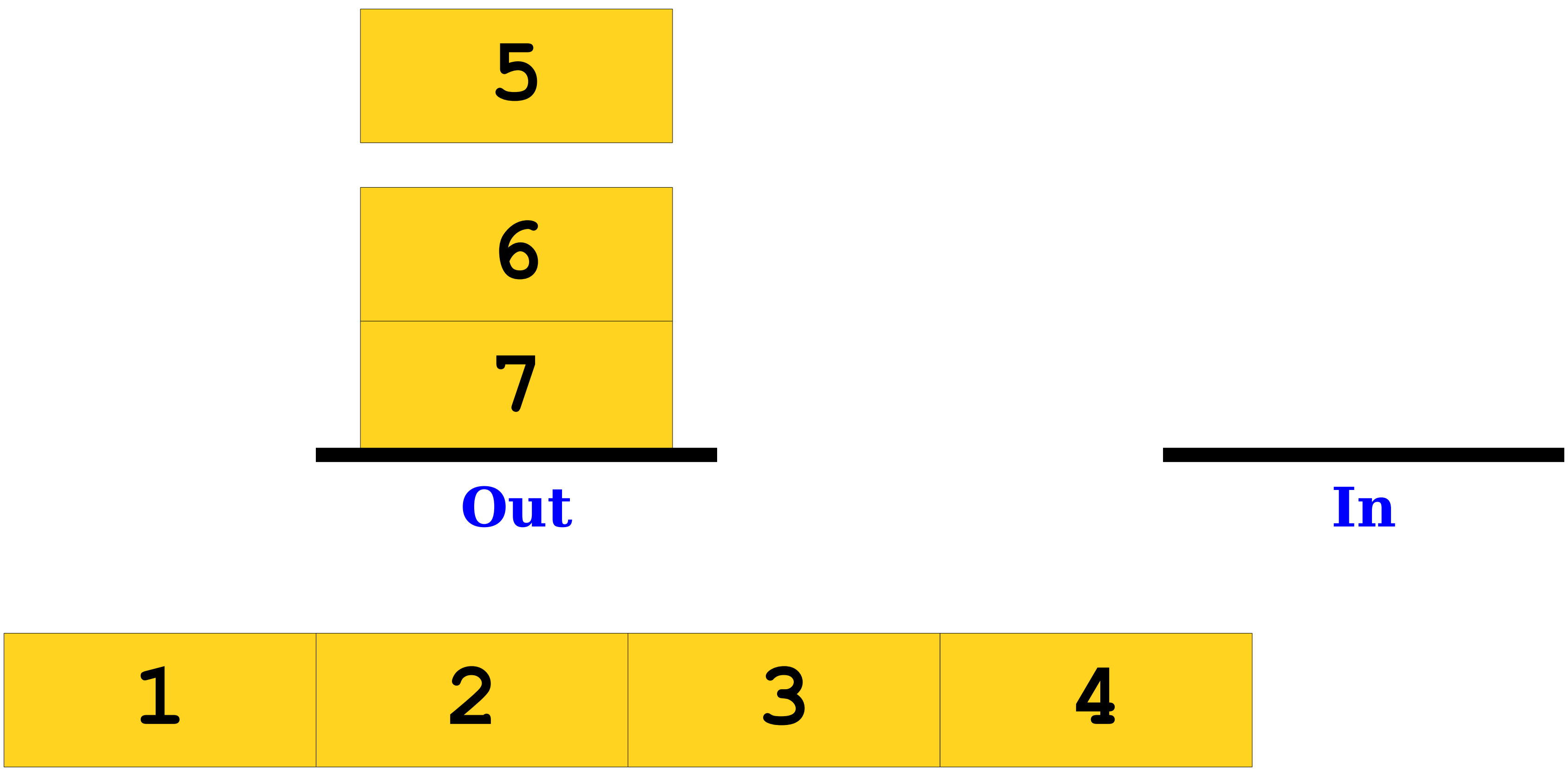
The Two-Stack Queue



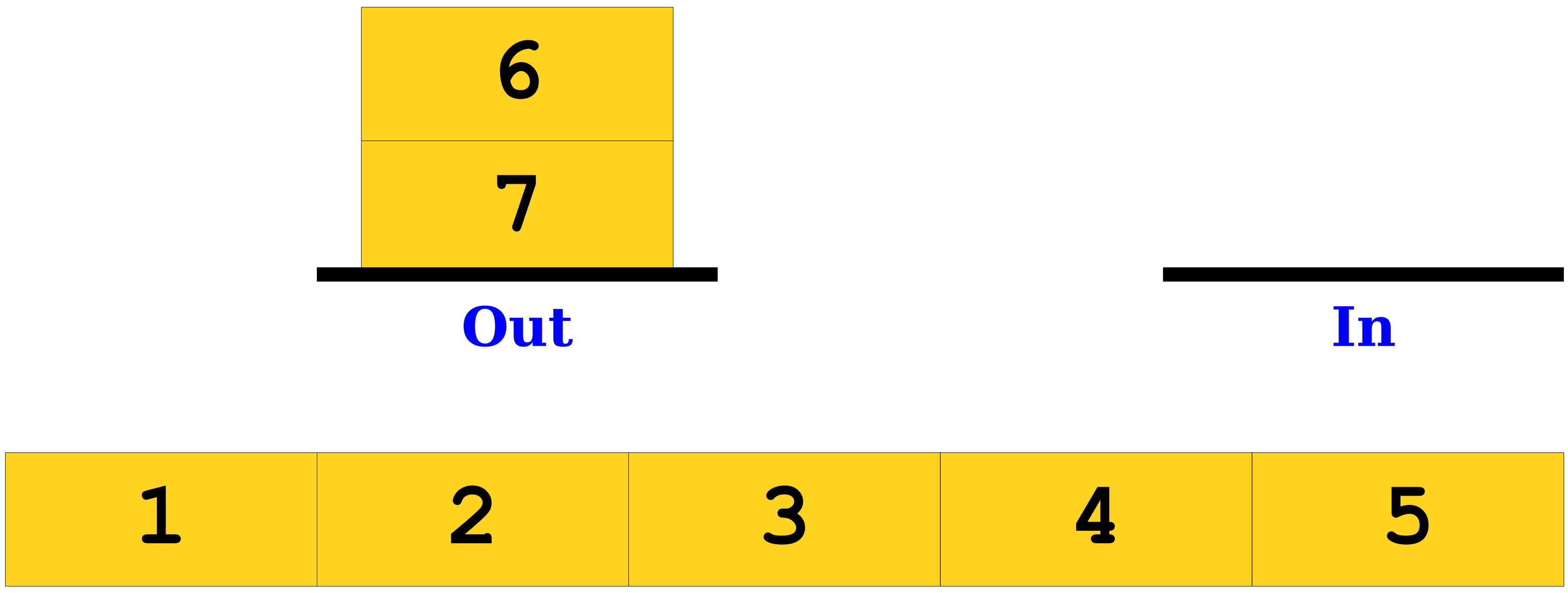
The Two-Stack Queue



The Two-Stack Queue



The Two-Stack Queue



The Banker's Method

- Enqueue:
  - O(1) work, plus one credit added.
  - Amortized cost: **O(1)**.
- "Light" dequeue:
  - O(1) work, plus no change in credits.
  - Amortized cost: **O(1)**.
- "Heavy" dequeue:
  - $\Theta(k)$ work, where k is the number of entries that started in the "in" stack.
  - k credits spent.
  - By choosing the amount of work in a credit appropriately, amortized cost is **O(1)**.

The Potential Method

- Define $\Phi(D)$ to be the height of the ***in*** stack.
- Enqueue:
  - Does O(1) work and increases Φ by one.
  - Amortized cost: **O(1)**.
- "Light" dequeue:
  - Does O(1) work and leaves Φ unchanged.
  - Amortized cost: **O(1)**.
- "Heavy" dequeue:
  - Does $\Theta(k)$ work, where k is the number of entries moved from the "in" stack.
  - $\Delta\Phi = -k$.
  - By choosing the amount of work stored in each unit of potential correctly, amortized cost becomes **O(1)**.

Time-Out for Announcements!

Problem Sets

- Problem Set Two solutions are now up on the course website.
  - The TAs are hard at work grading everything. We'll try to get everything back as soon as possible!
- Problem Set Three is due next Thursday, May 3rd, at 2:30PM.
  - Have questions? Stop by office hours or ask them on Piazza!

Grace Hopper Conference

- Applications are now open for CS department funding to attend next year's Grace Hopper Conference
  - (September 26 – 28, Houston, TX)
- Phenomenal opportunity for anyone interested.
- Apply online using ***this link***.

Back to CS166!

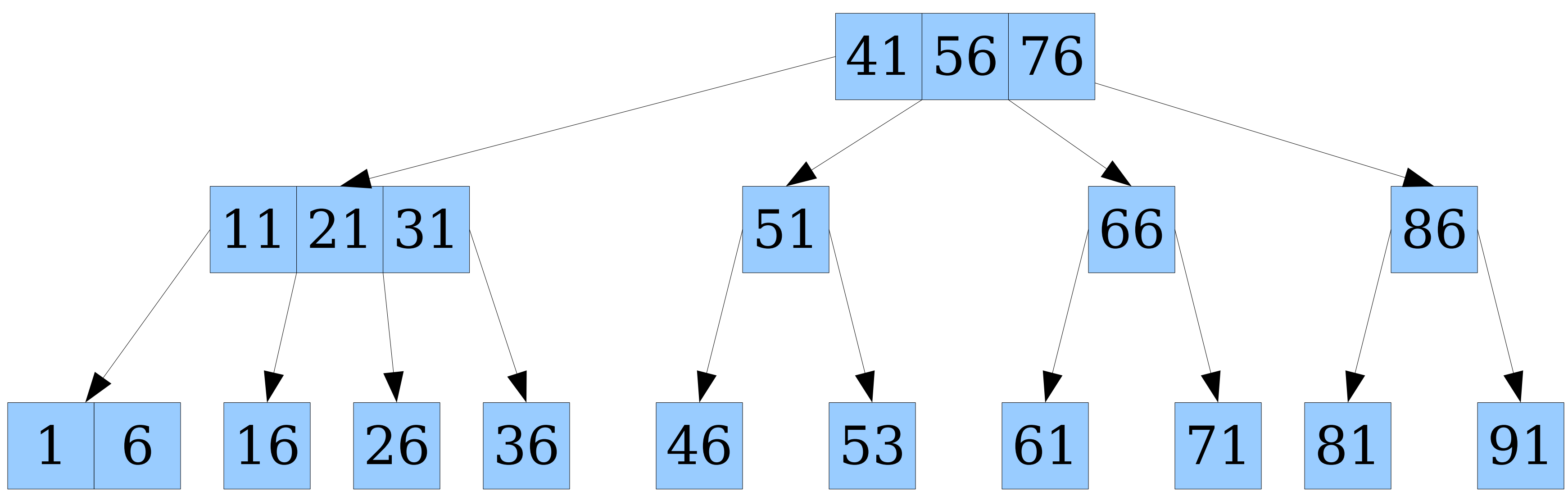
Another Example: ***2-3-4 Trees***

2-3-4 Trees

- Inserting or deleting values from a 2-3-4 trees takes time O(log n).
- Why is that?
  - We do some amount of work finding the insertion or deletion point, which is $\Theta(\log n)$.
  - We also do some amount of work "fixing up" the tree by doing insertions or deletions.
- What is the cost of that second amount of work?

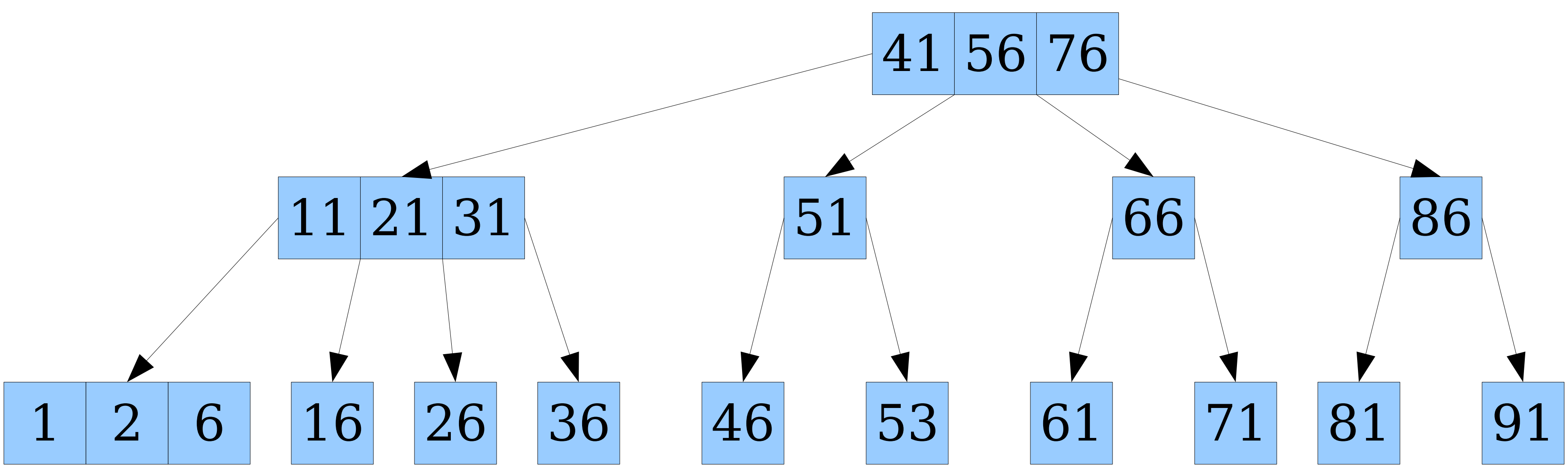
2-3-4 Tree Insertions

- Most insertions into 2-3-4 trees require no fixup – we just insert an extra key into a leaf.
- Some insertions require some fixup to split nodes and propagate upward.



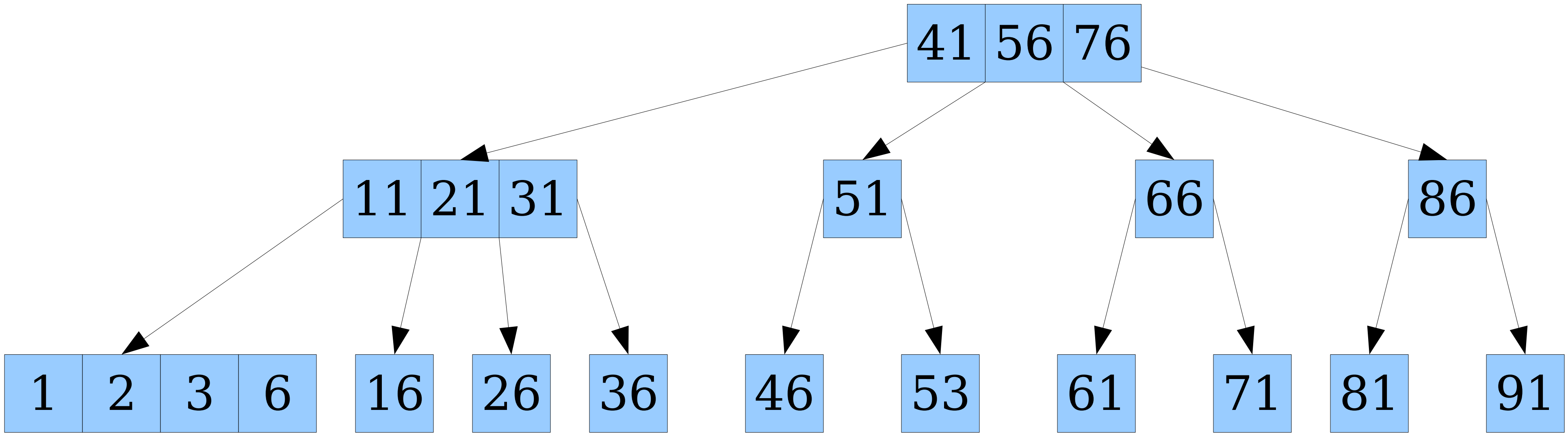
2-3-4 Tree Insertions

- Most insertions into 2-3-4 trees require no fixup – we just insert an extra key into a leaf.
- Some insertions require some fixup to split nodes and propagate upward.



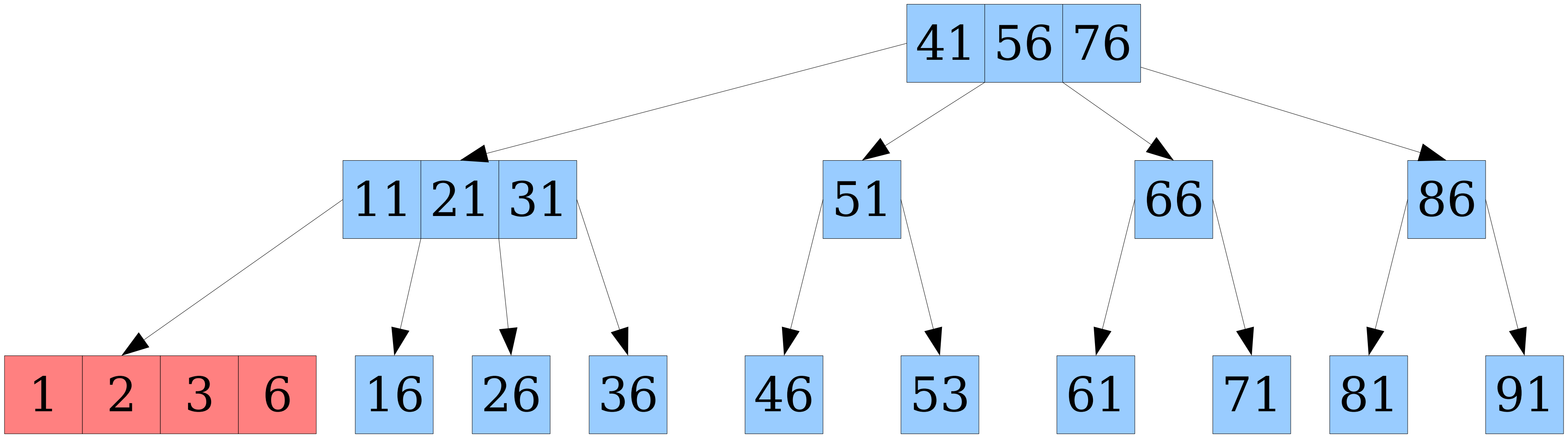
2-3-4 Tree Insertions

- Most insertions into 2-3-4 trees require no fixup – we just insert an extra key into a leaf.
- Some insertions require some fixup to split nodes and propagate upward.



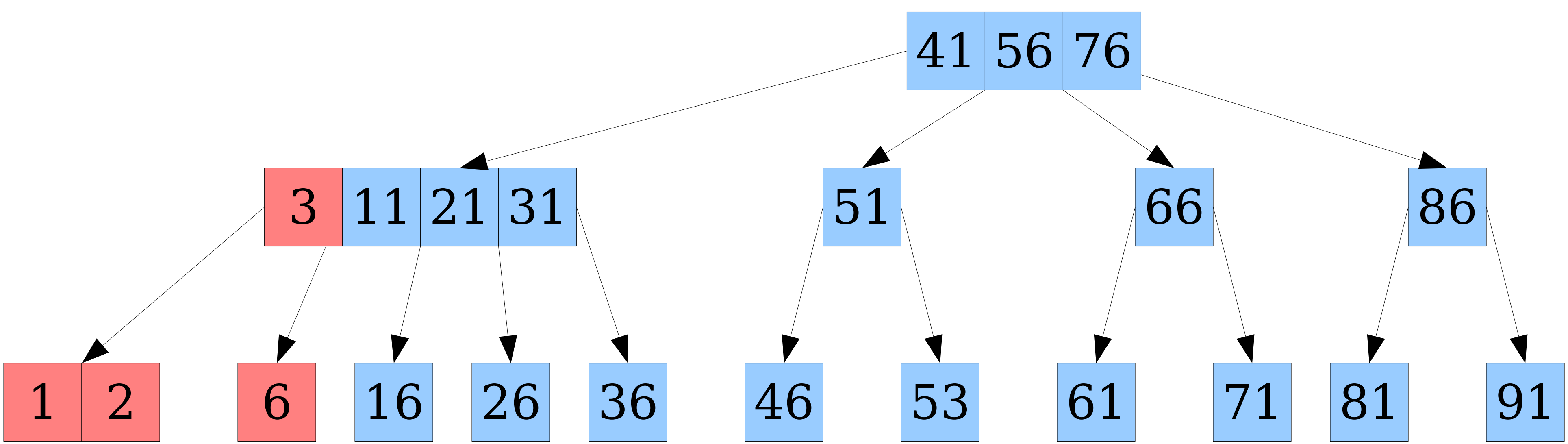
2-3-4 Tree Insertions

- Most insertions into 2-3-4 trees require no fixup – we just insert an extra key into a leaf.
- Some insertions require some fixup to split nodes and propagate upward.



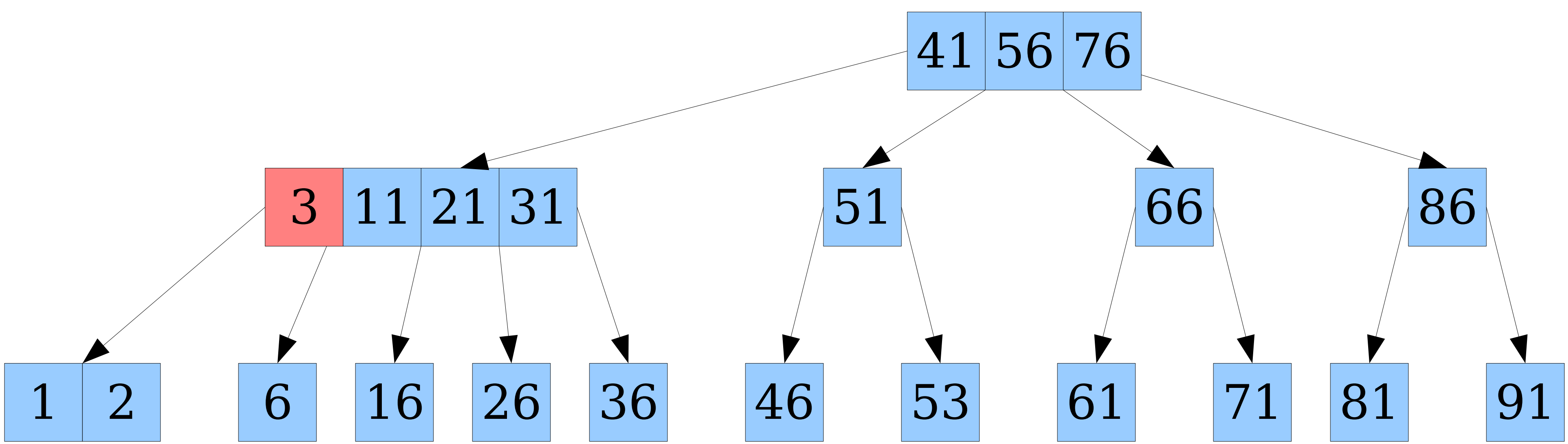
2-3-4 Tree Insertions

- Most insertions into 2-3-4 trees require no fixup – we just insert an extra key into a leaf.
- Some insertions require some fixup to split nodes and propagate upward.



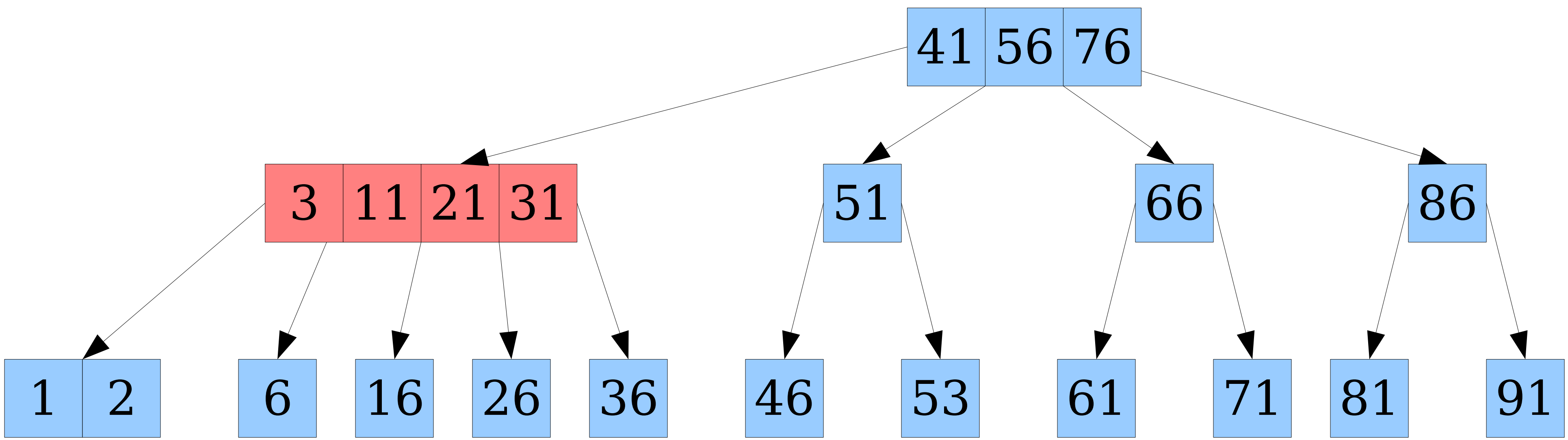
2-3-4 Tree Insertions

- Most insertions into 2-3-4 trees require no fixup – we just insert an extra key into a leaf.
- Some insertions require some fixup to split nodes and propagate upward.



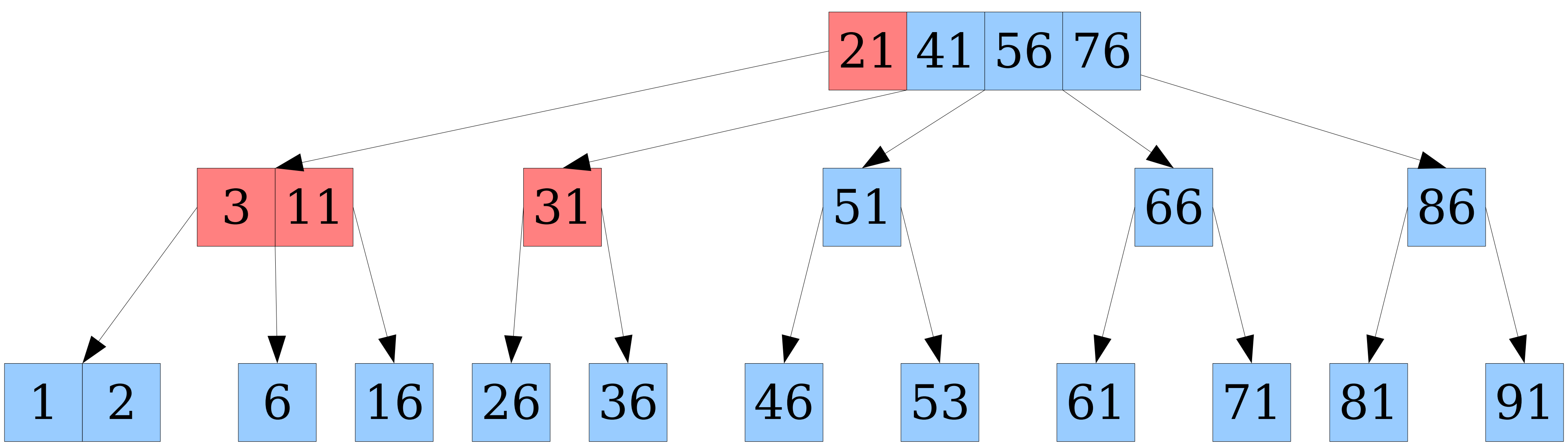
2-3-4 Tree Insertions

- Most insertions into 2-3-4 trees require no fixup – we just insert an extra key into a leaf.
- Some insertions require some fixup to split nodes and propagate upward.



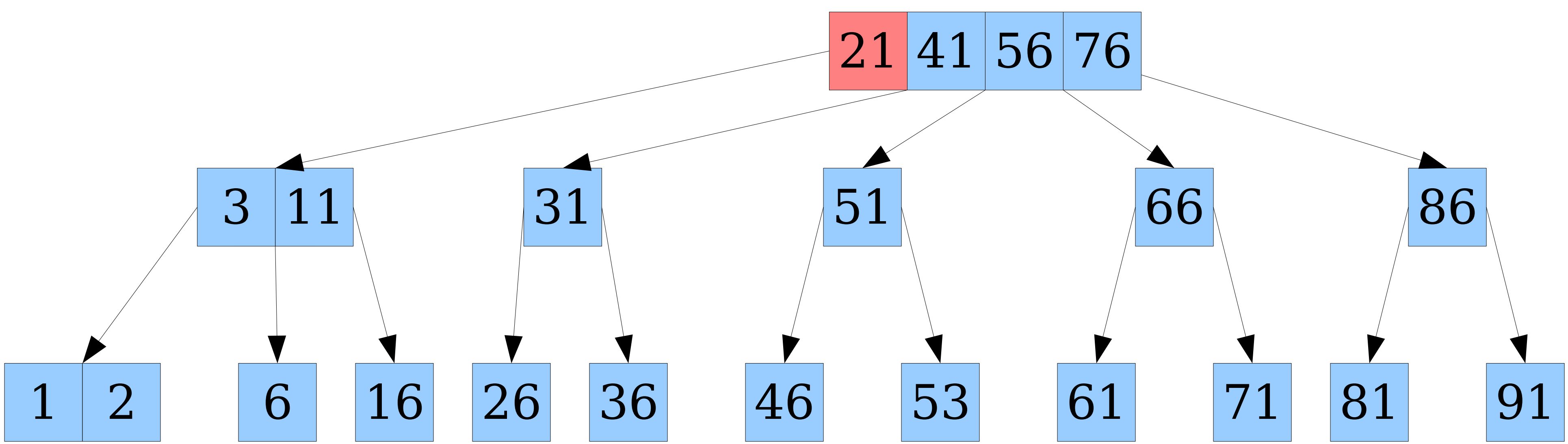
2-3-4 Tree Insertions

- Most insertions into 2-3-4 trees require no fixup – we just insert an extra key into a leaf.
- Some insertions require some fixup to split nodes and propagate upward.



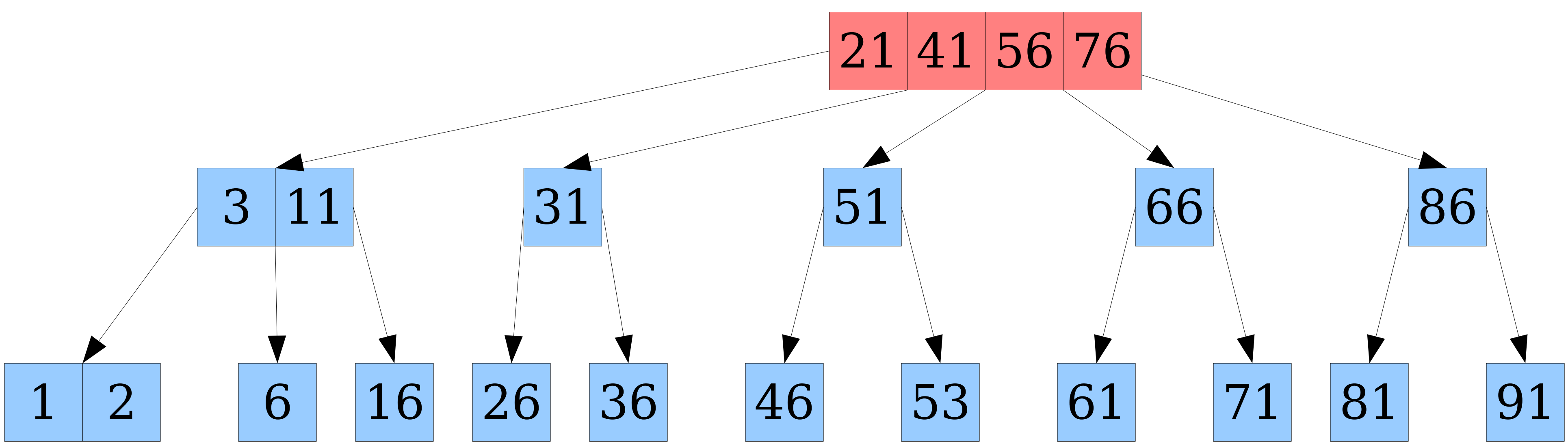
2-3-4 Tree Insertions

- Most insertions into 2-3-4 trees require no fixup – we just insert an extra key into a leaf.
- Some insertions require some fixup to split nodes and propagate upward.



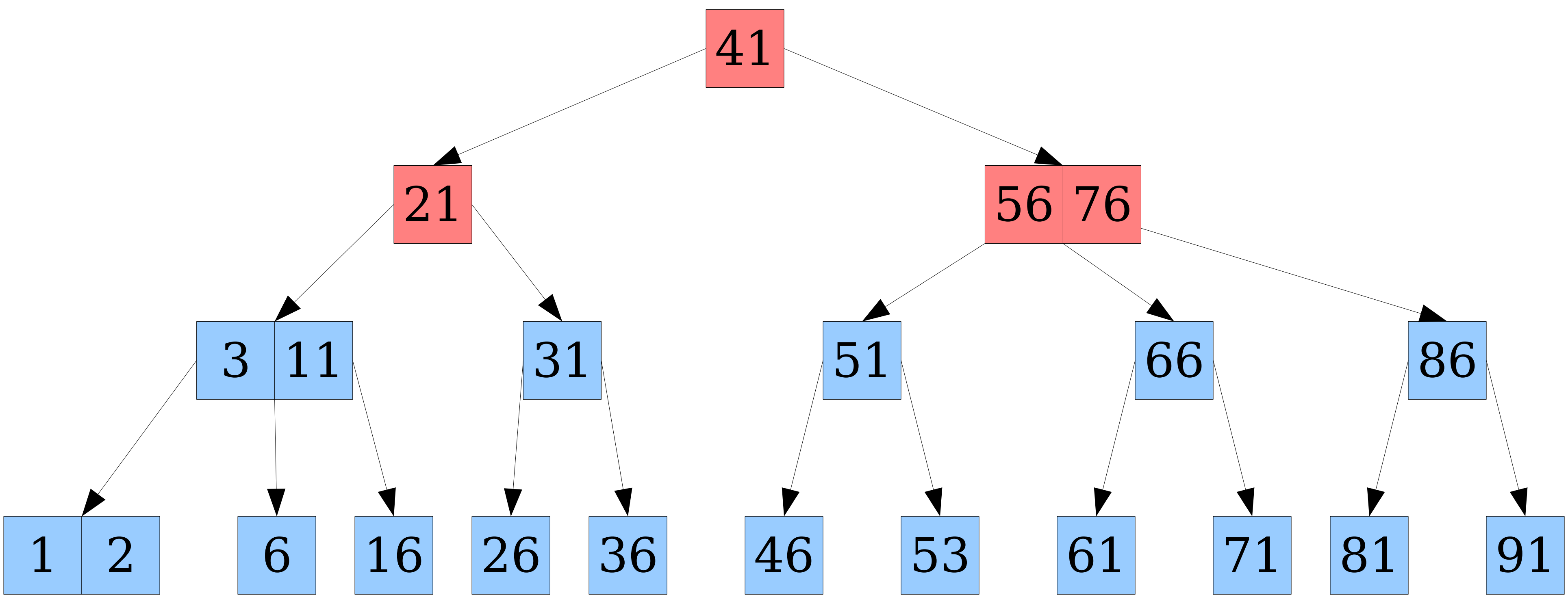
2-3-4 Tree Insertions

- Most insertions into 2-3-4 trees require no fixup – we just insert an extra key into a leaf.
- Some insertions require some fixup to split nodes and propagate upward.



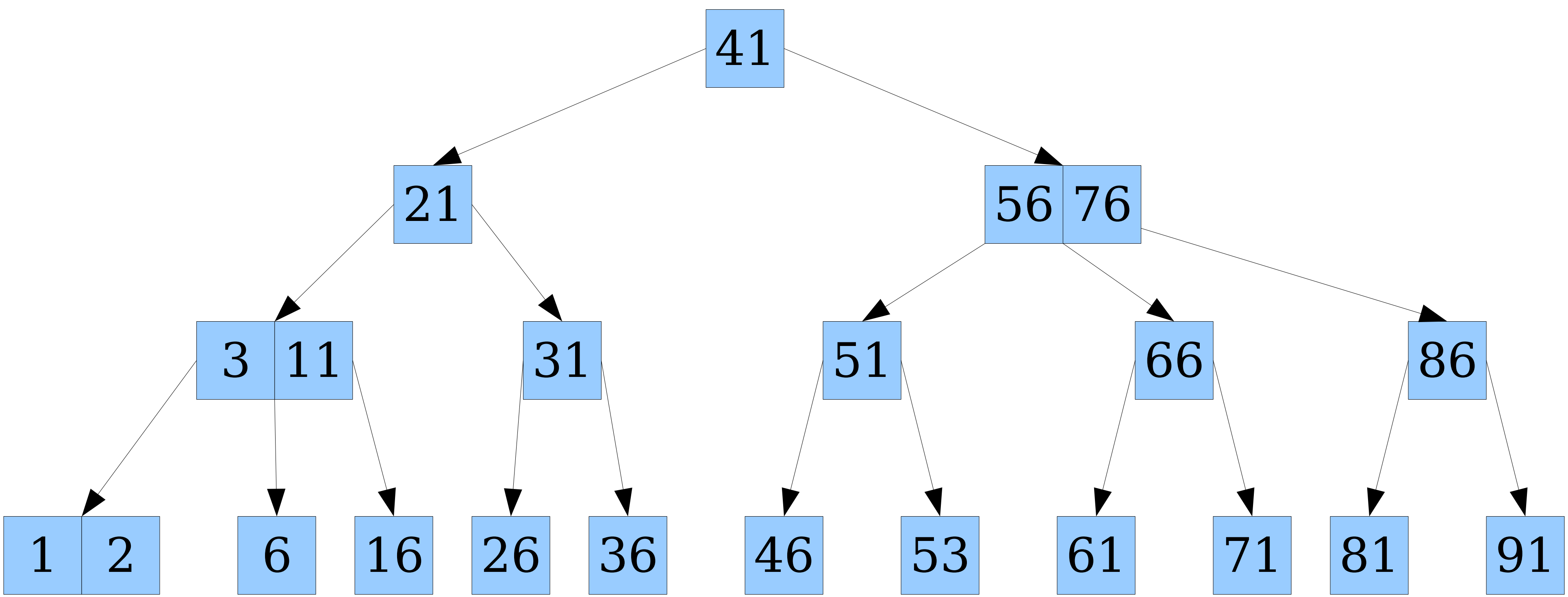
2-3-4 Tree Insertions

- Most insertions into 2-3-4 trees require no fixup – we just insert an extra key into a leaf.
- Some insertions require some fixup to split nodes and propagate upward.



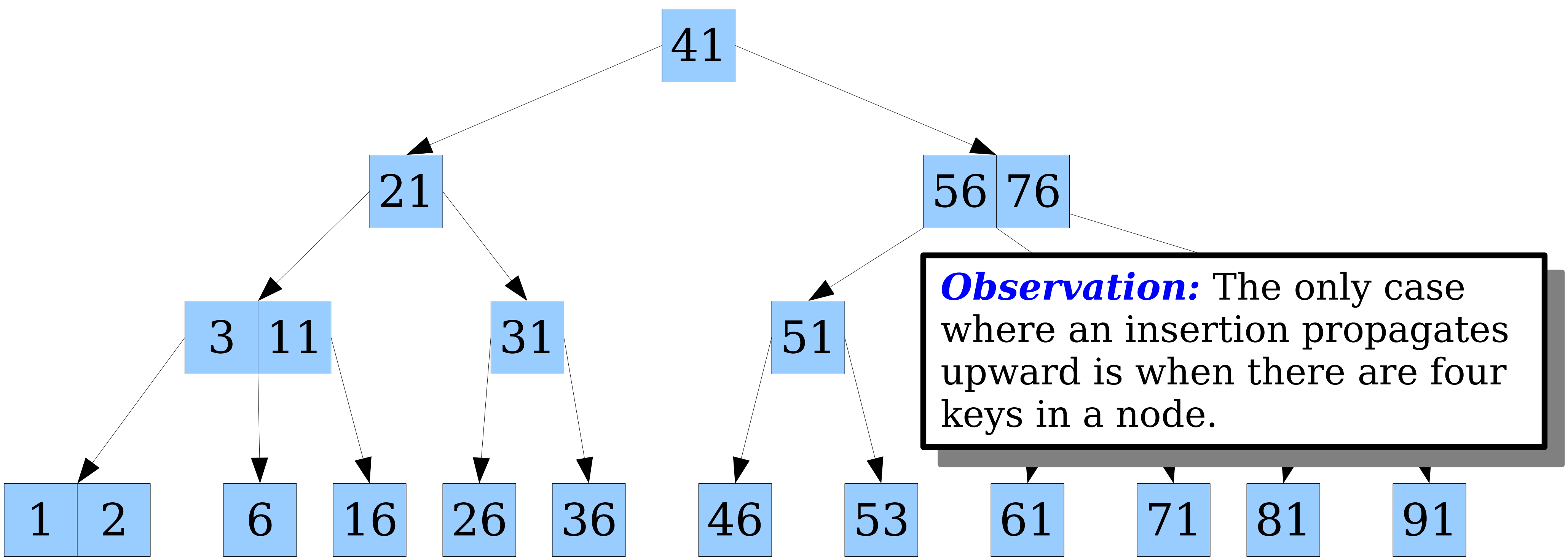
2-3-4 Tree Insertions

- Most insertions into 2-3-4 trees require no fixup – we just insert an extra key into a leaf.
- Some insertions require some fixup to split nodes and propagate upward.



2-3-4 Tree Insertions

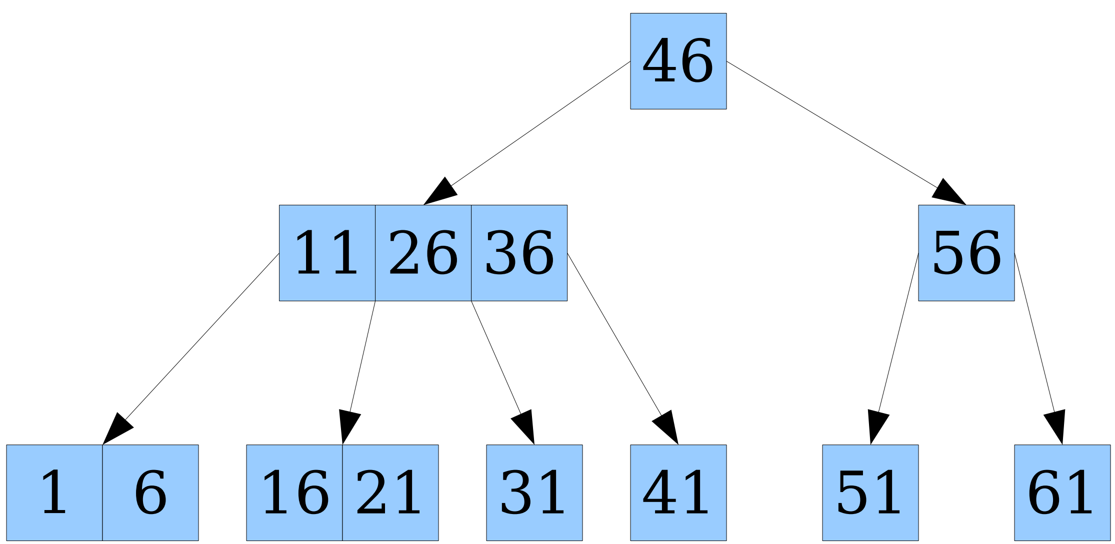
- Most insertions into 2-3-4 trees require no fixup – we just insert an extra key into a leaf.
- Some insertions require some fixup to split nodes and propagate upward.



Observation: The only case where an insertion propagates upward is when there are four keys in a node.

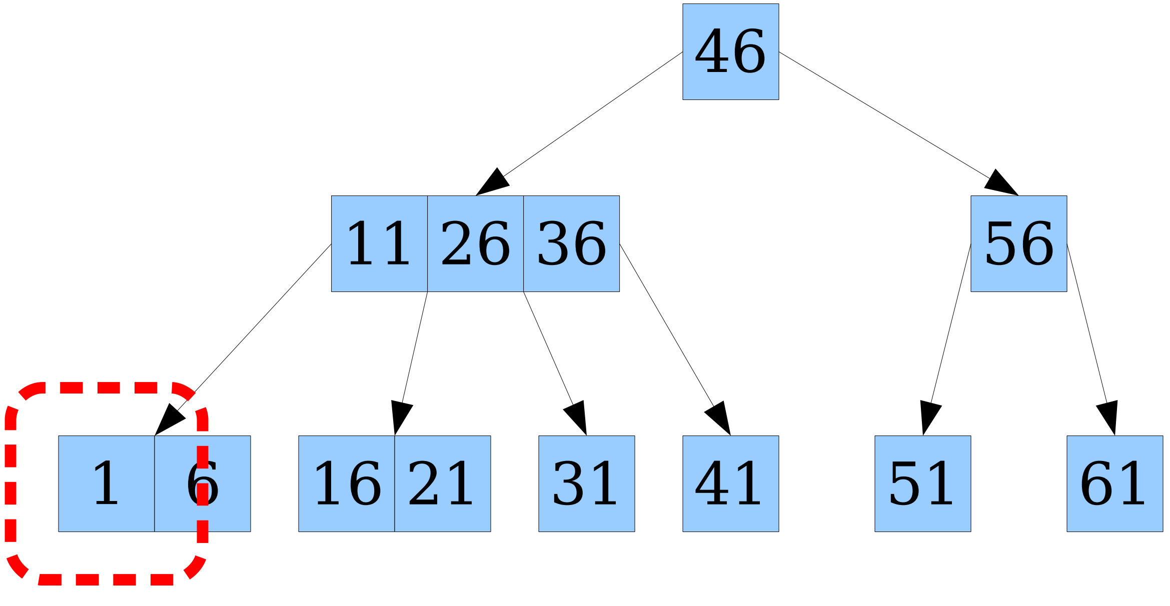
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



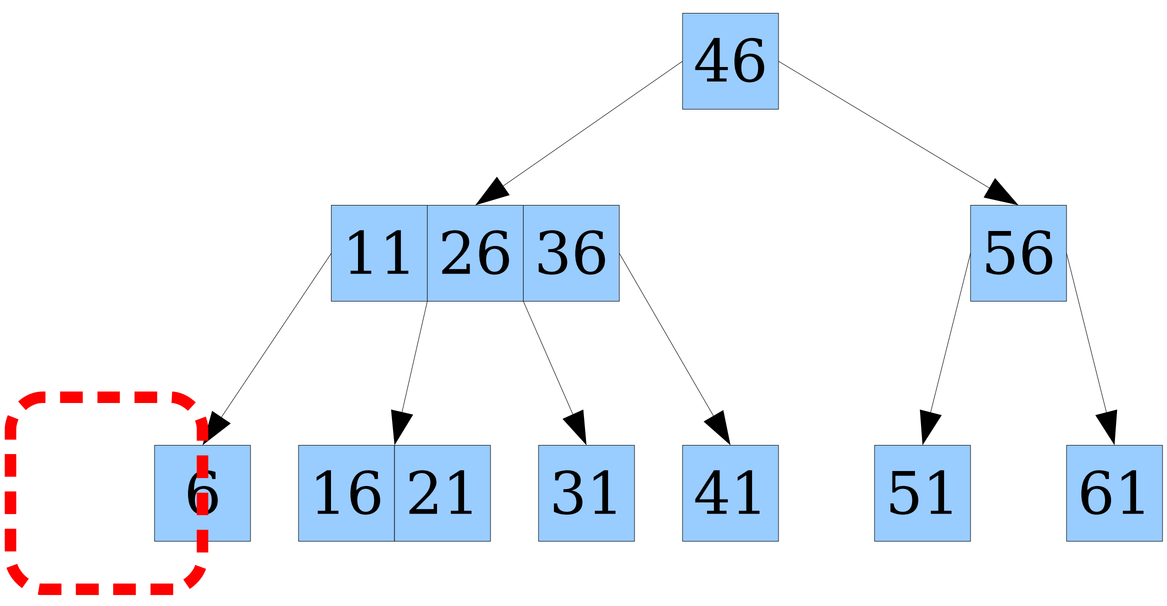
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



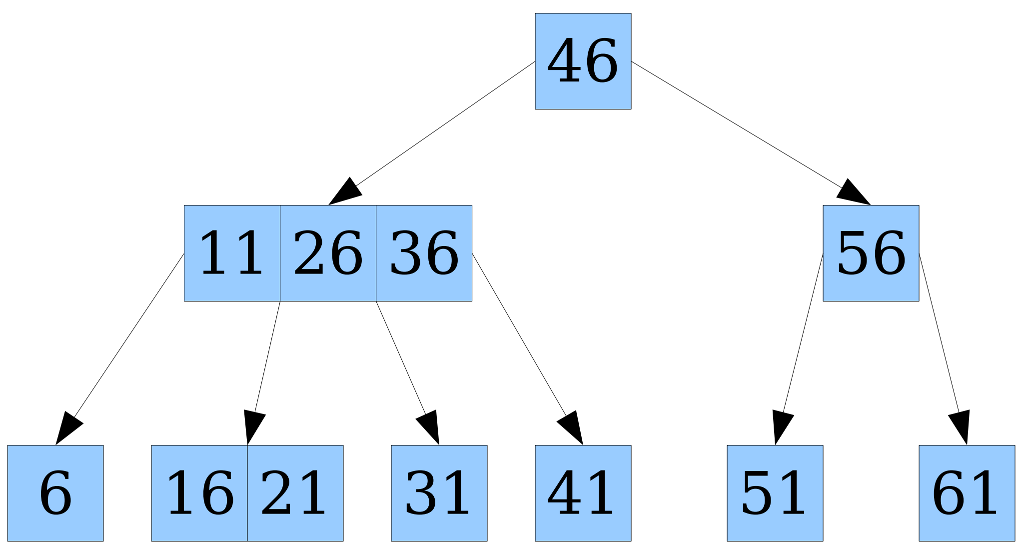
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



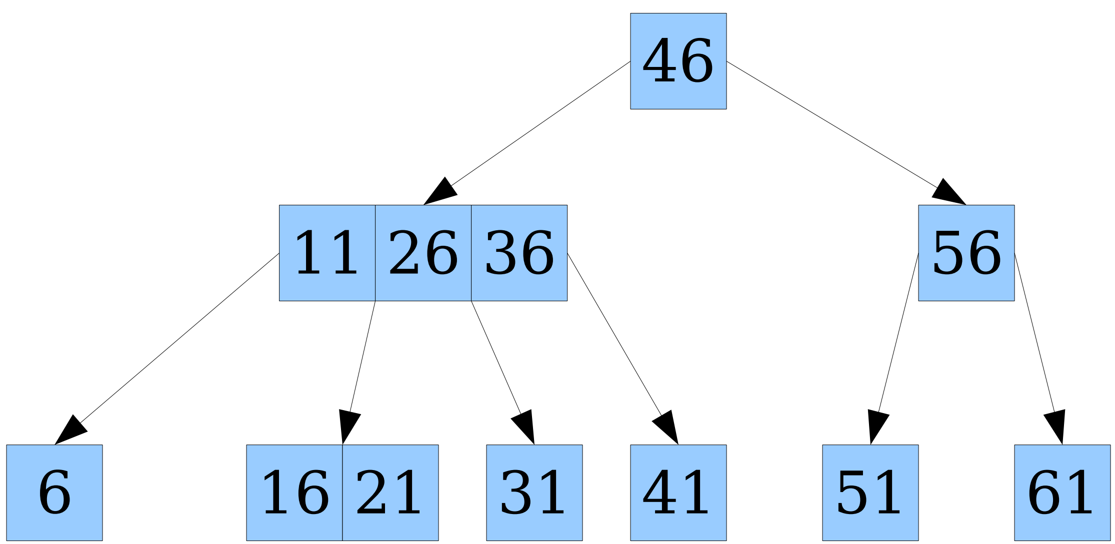
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



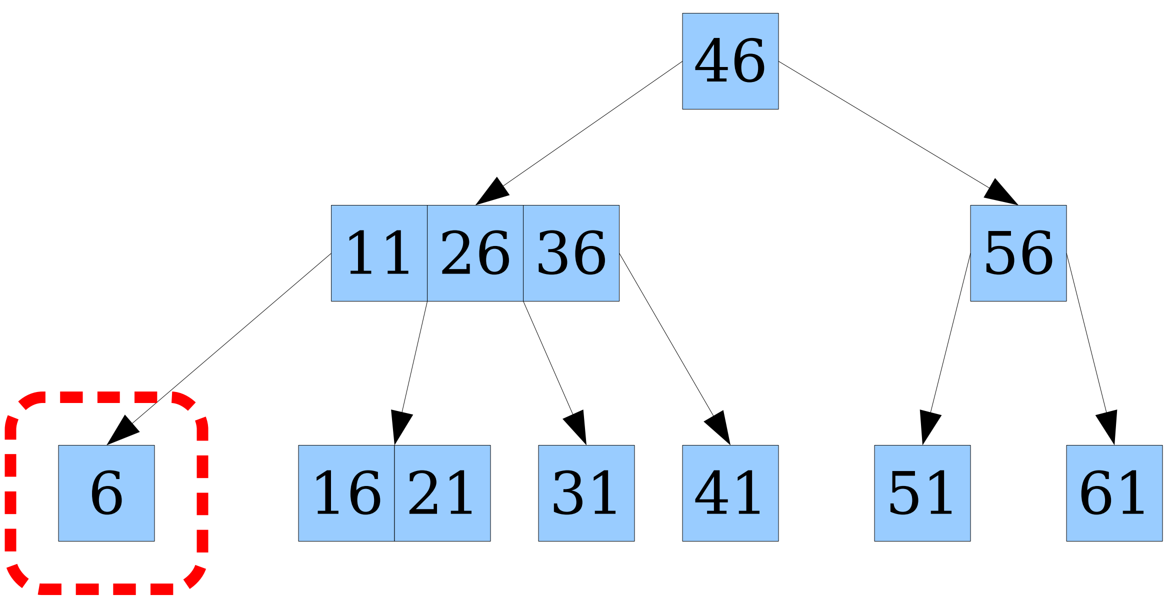
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



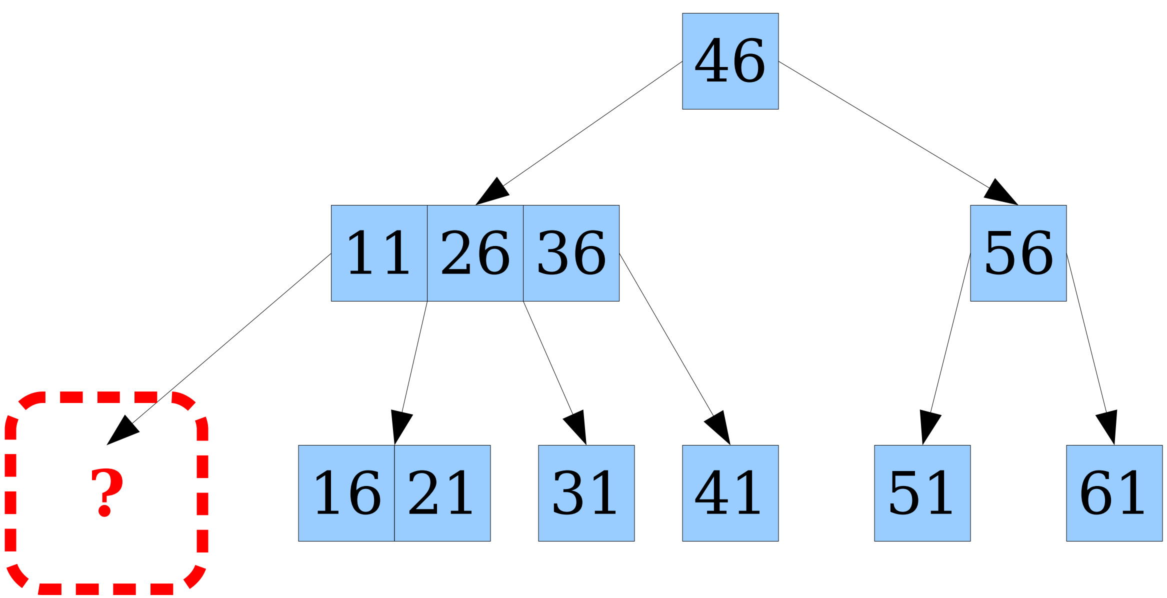
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



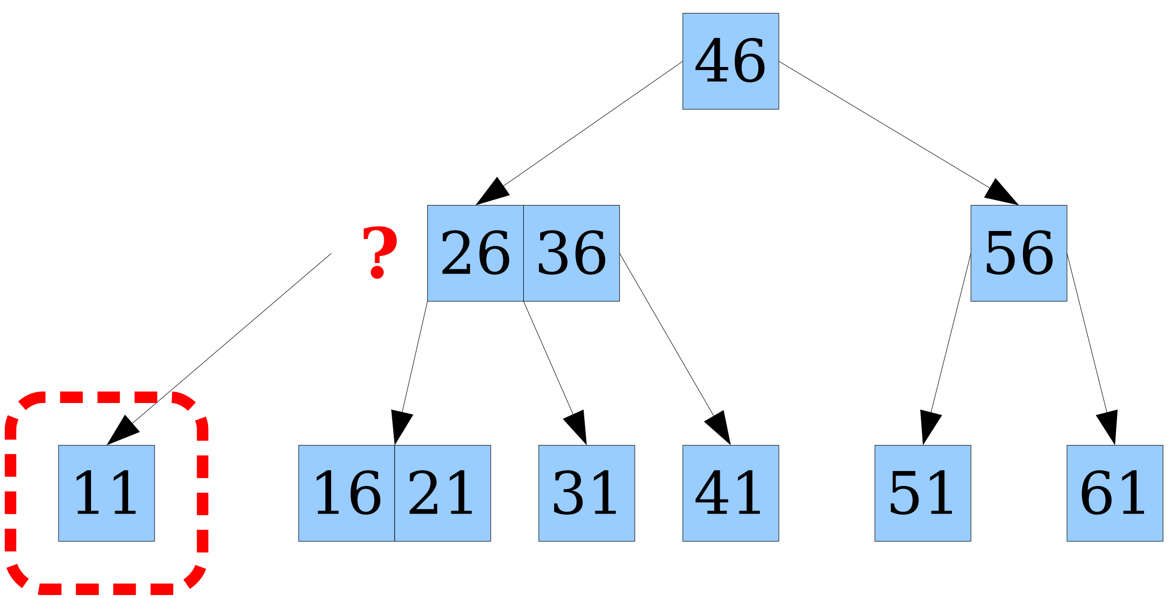
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



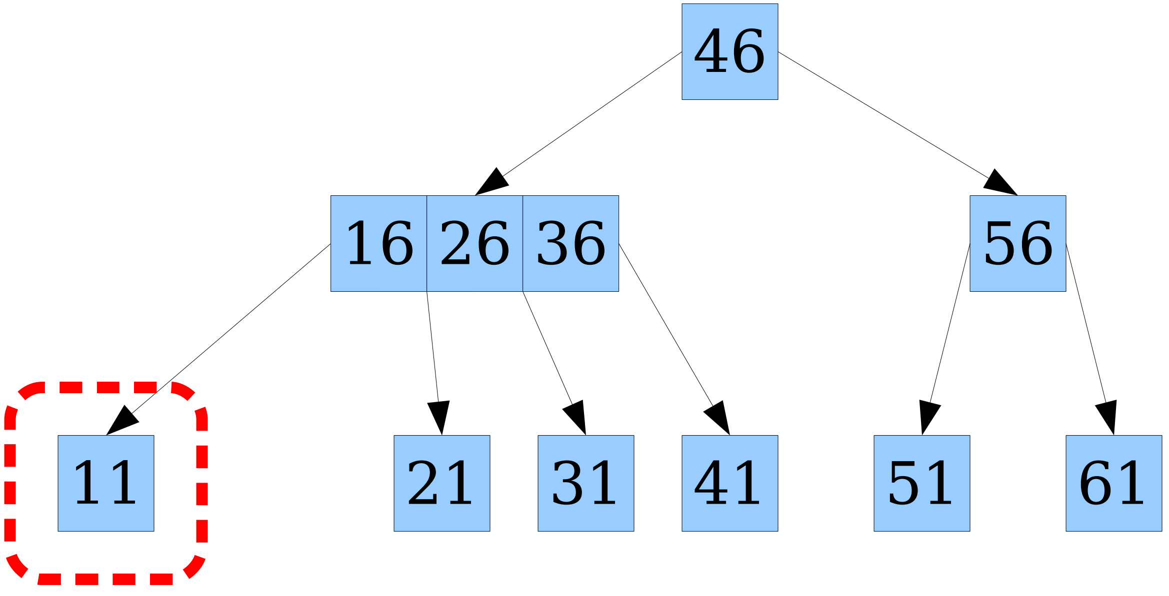
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



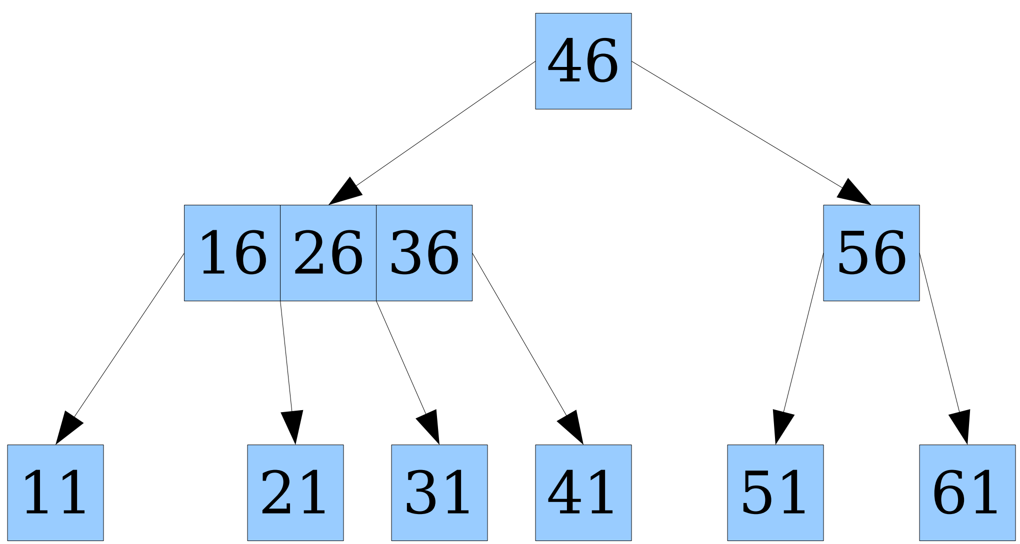
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



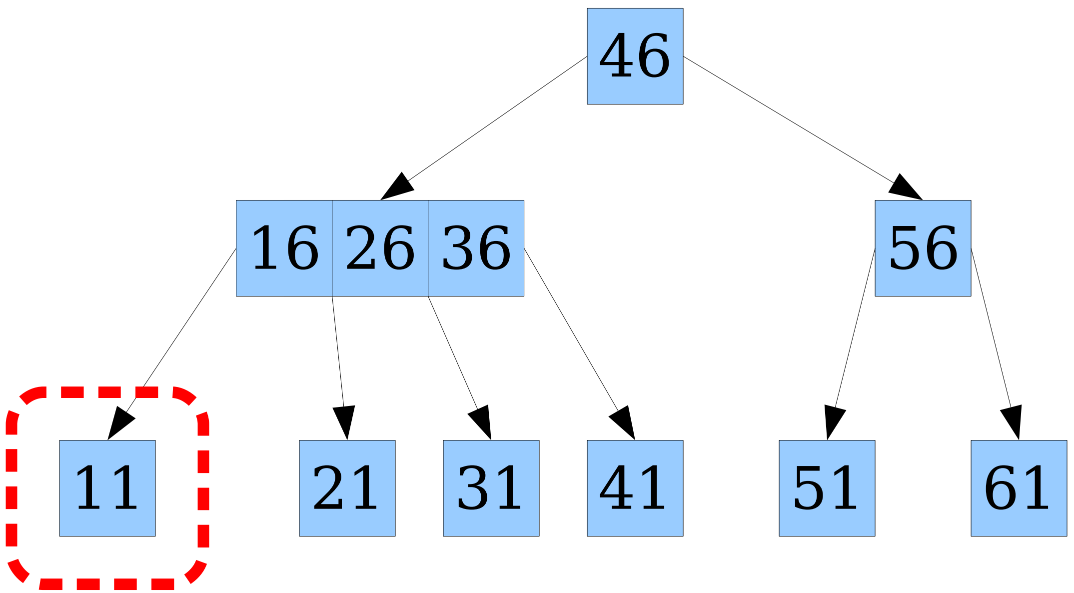
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



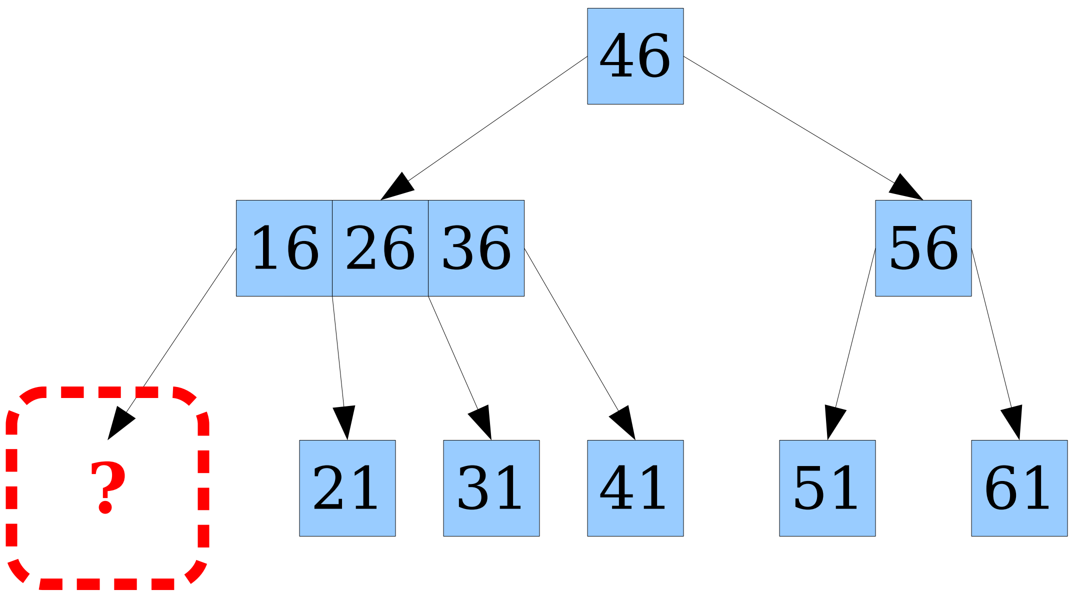
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



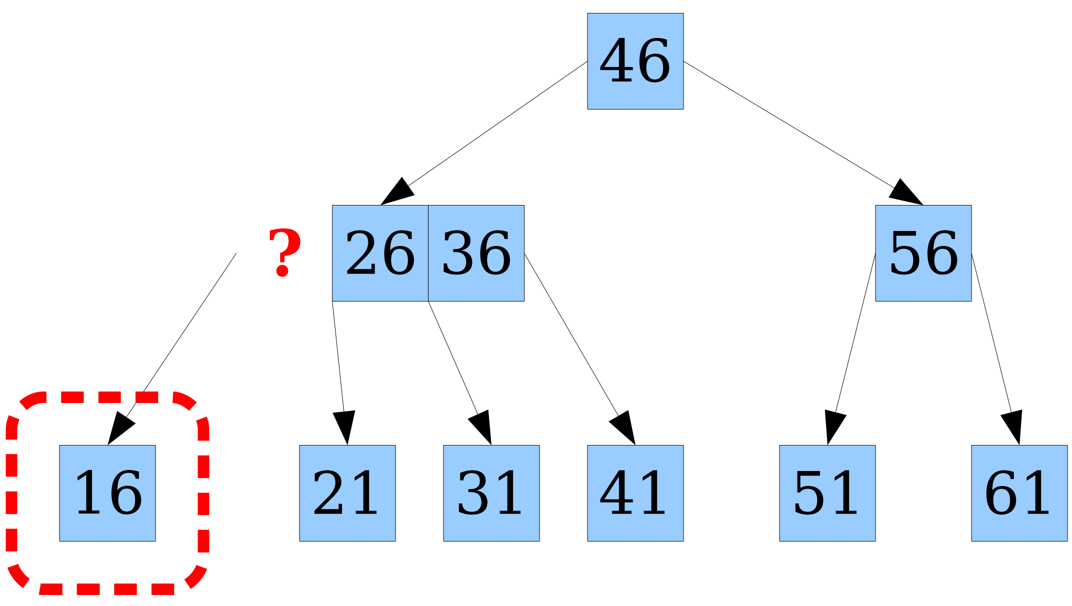
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



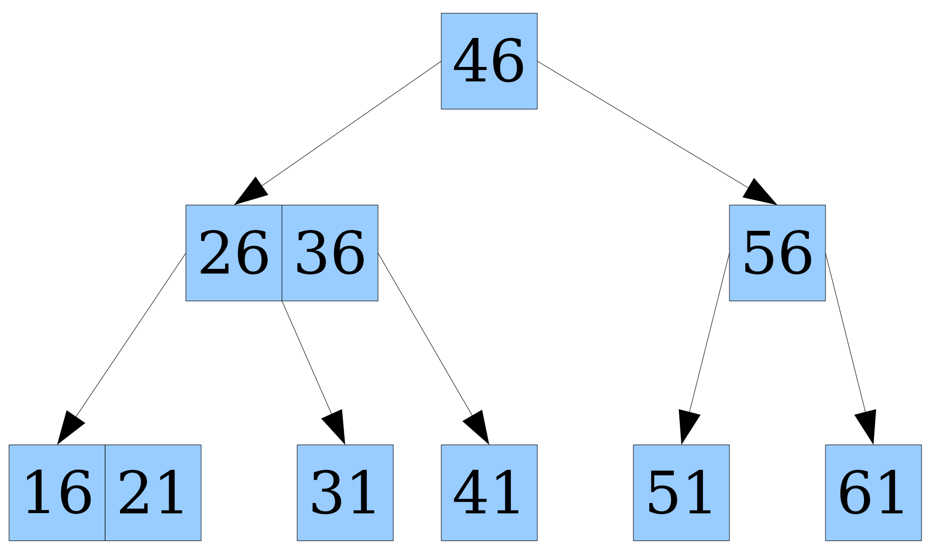
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



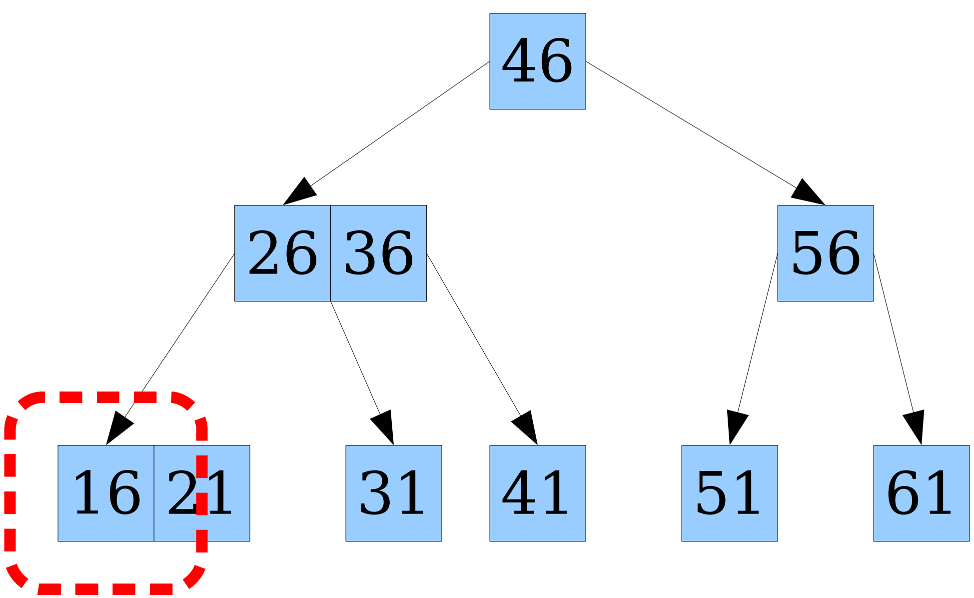
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



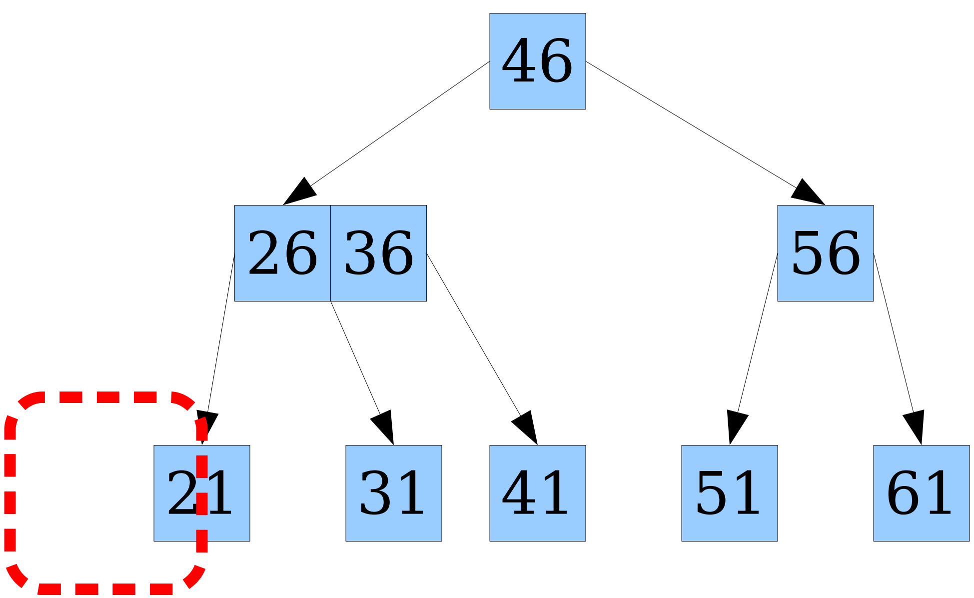
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



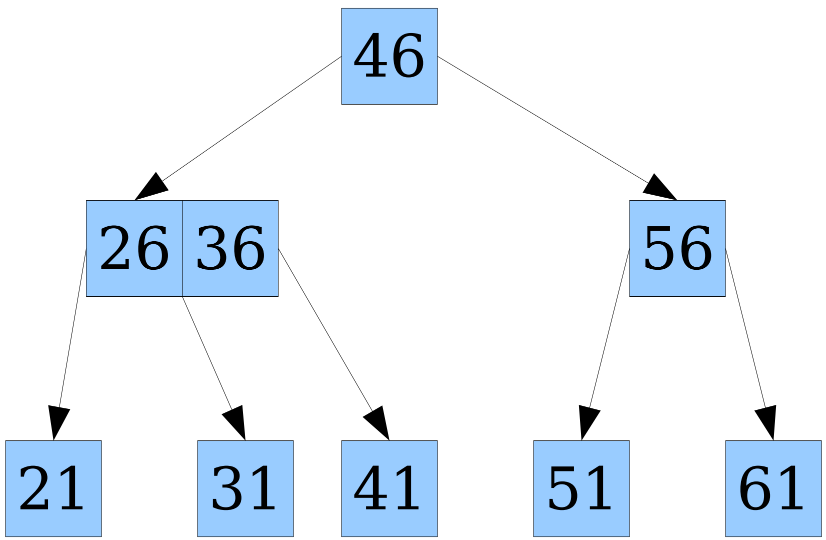
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



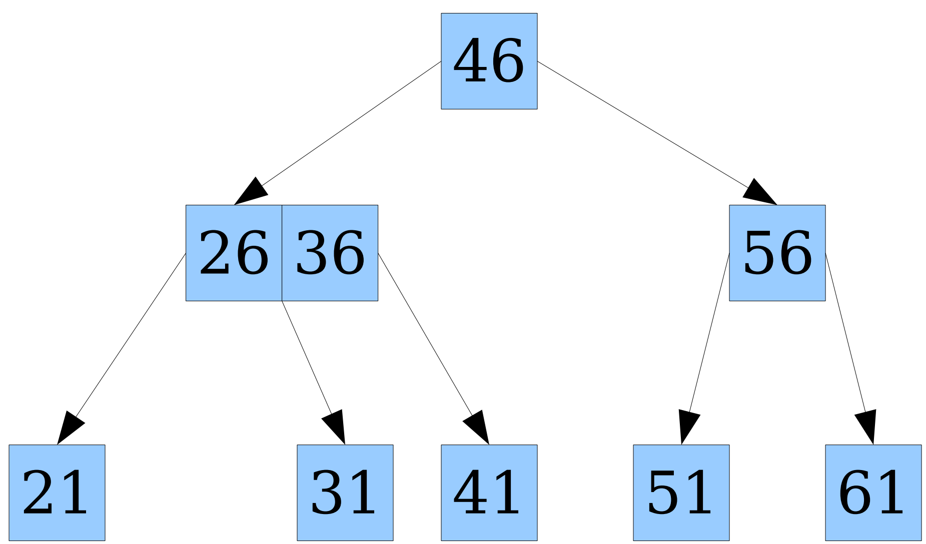
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



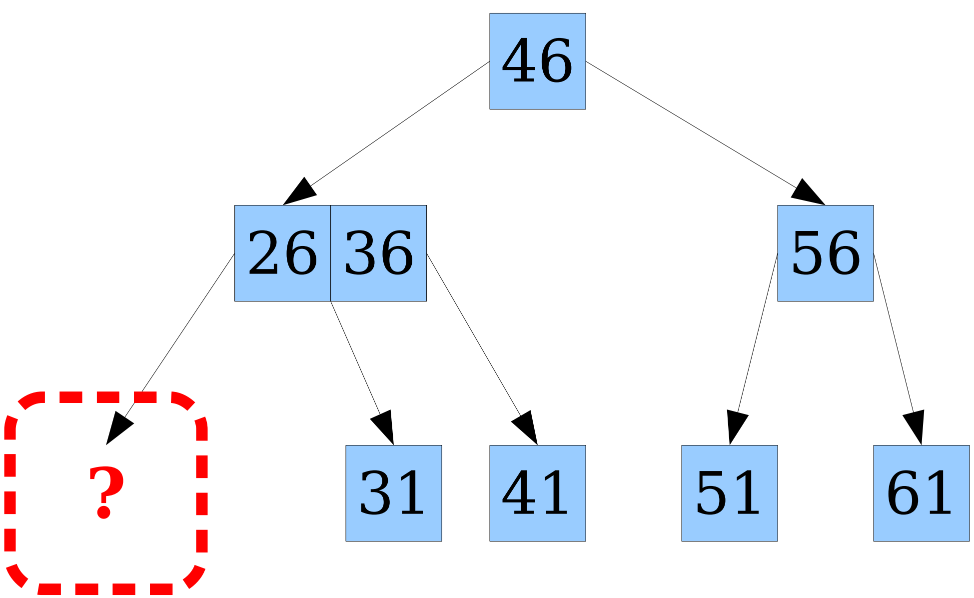
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



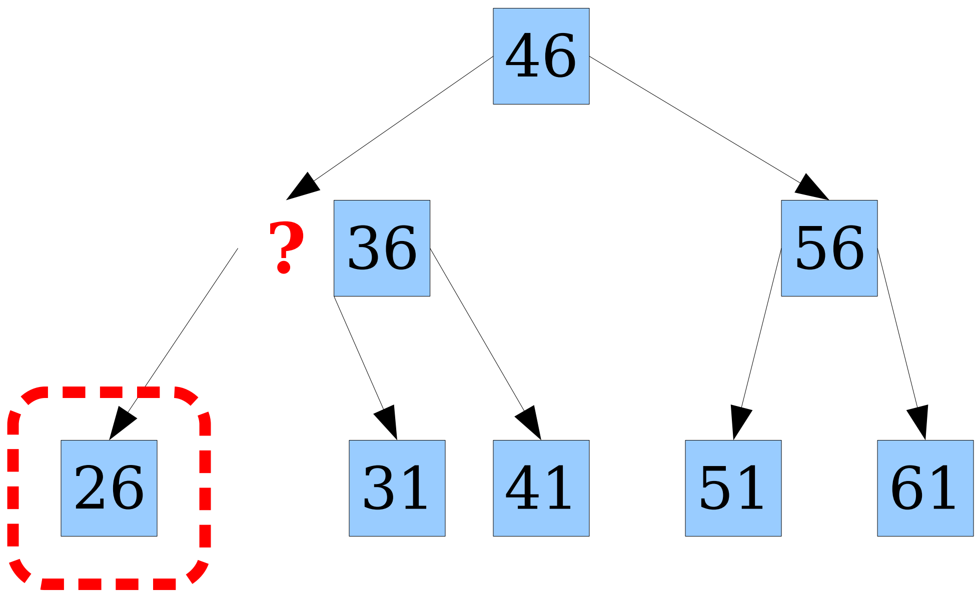
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



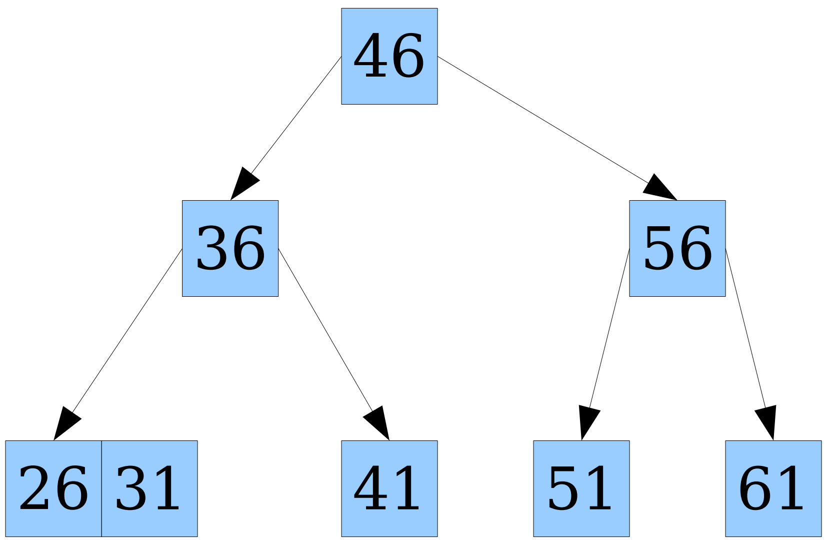
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



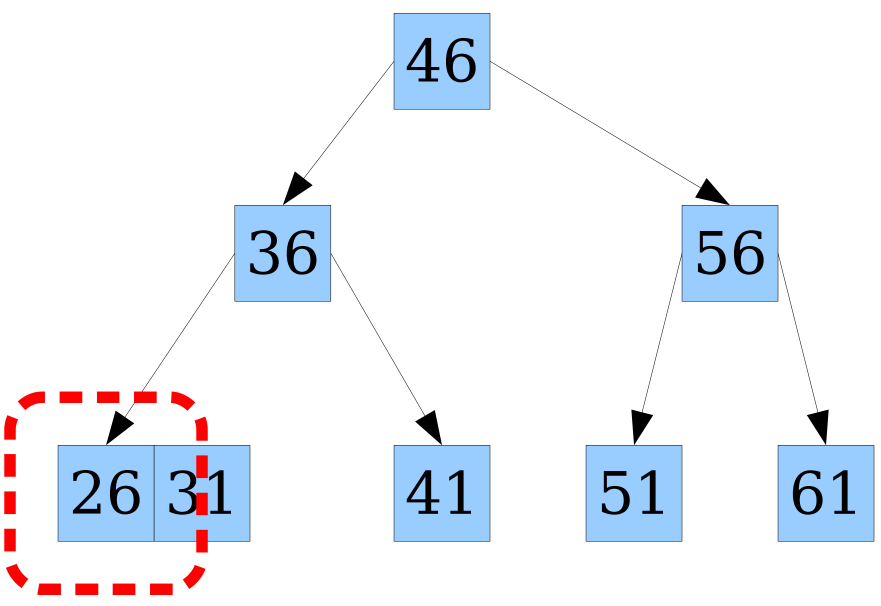
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



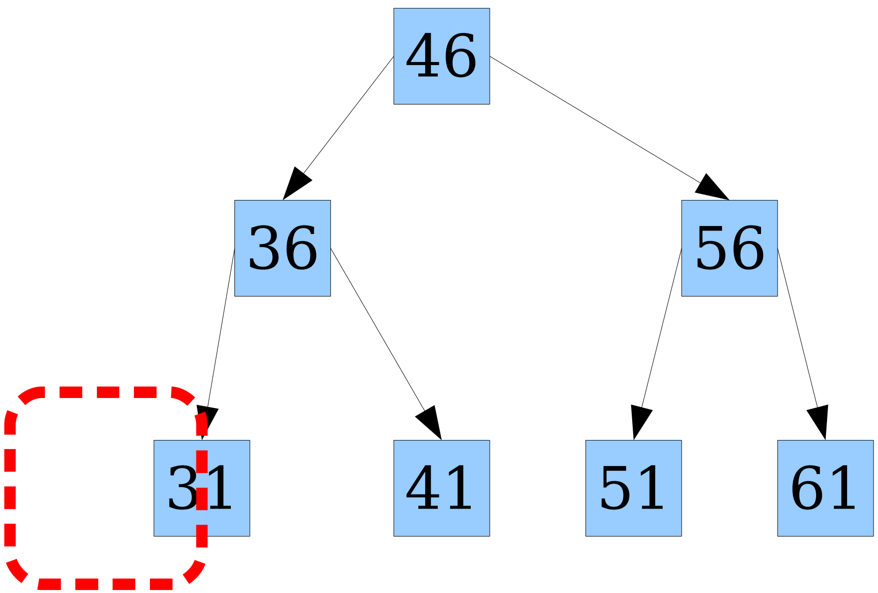
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



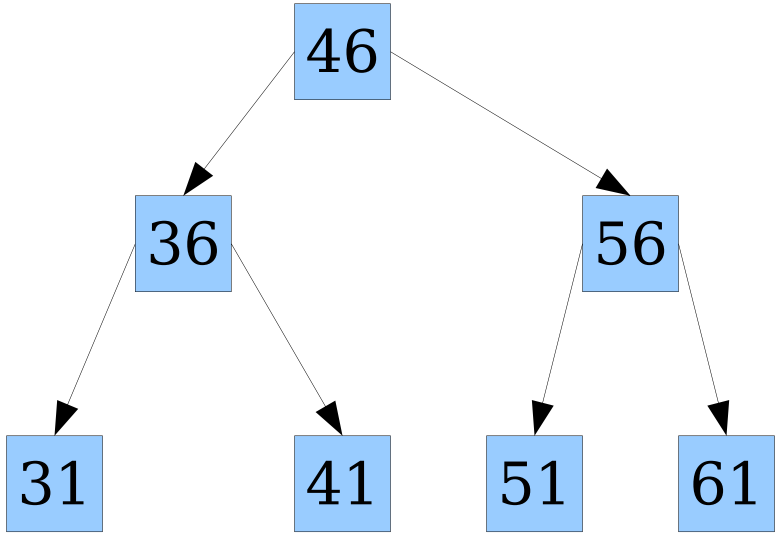
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



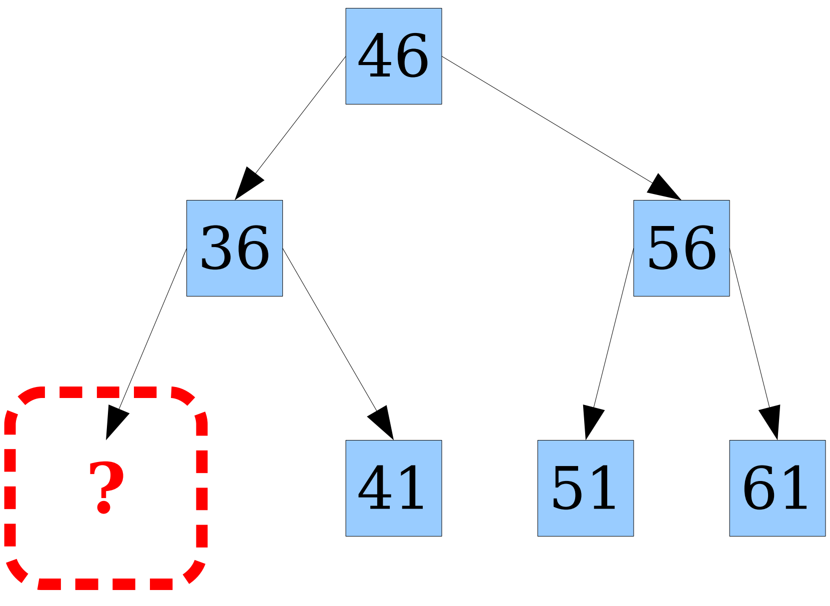
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



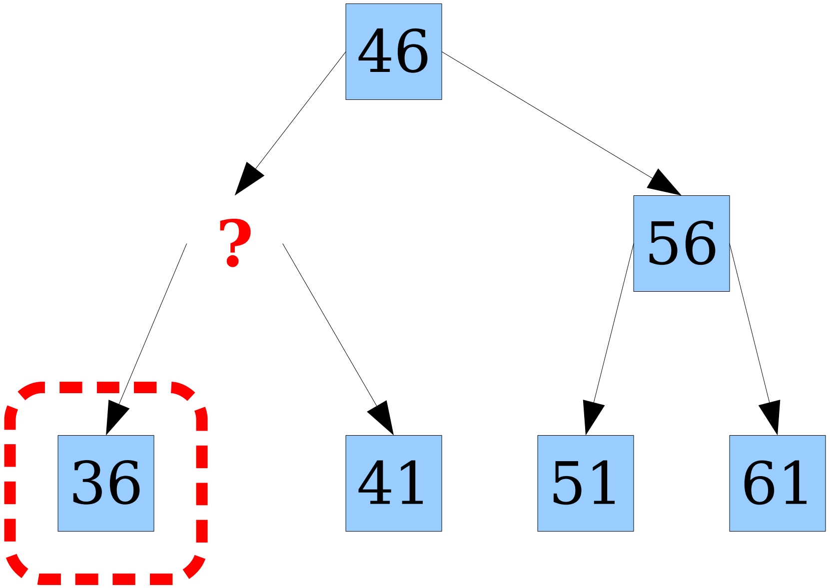
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



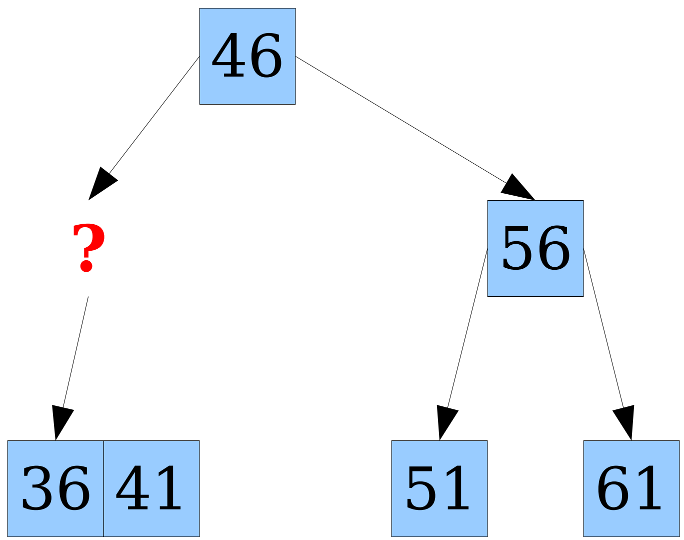
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



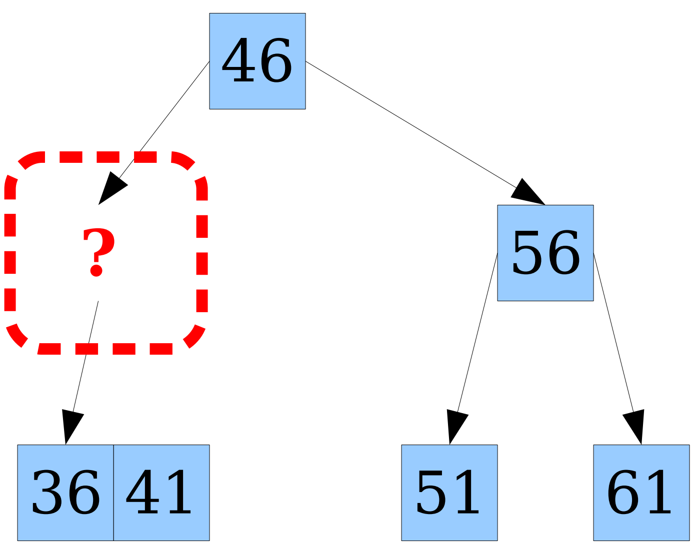
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



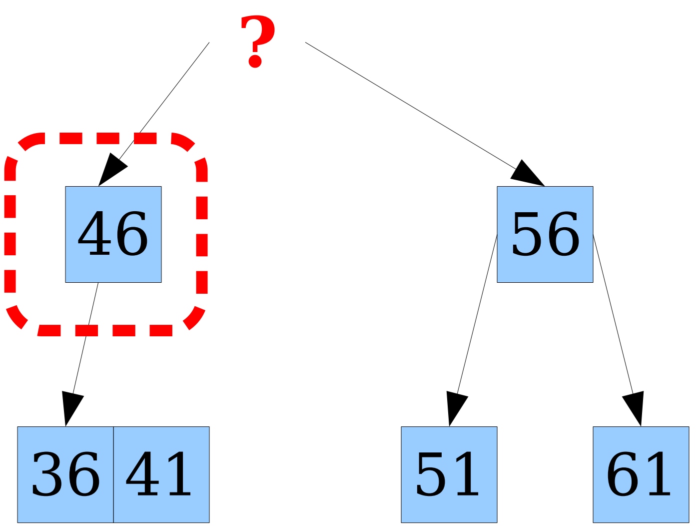
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



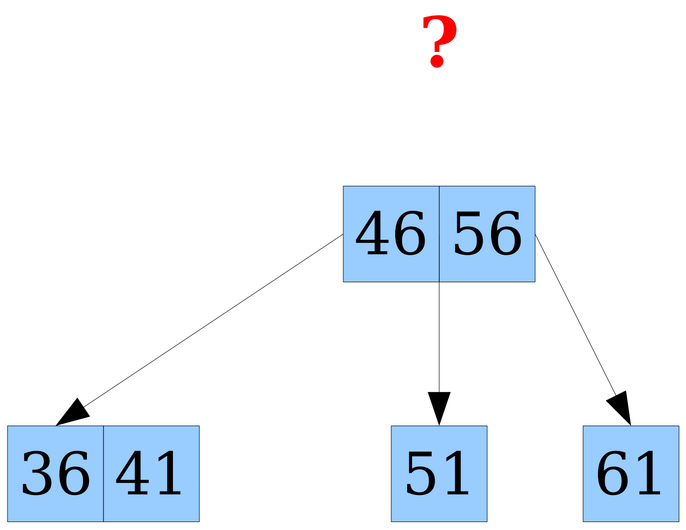
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



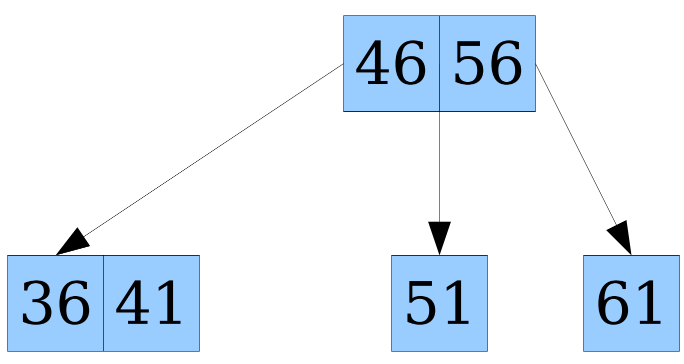
2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



2-3-4 Tree Deletions

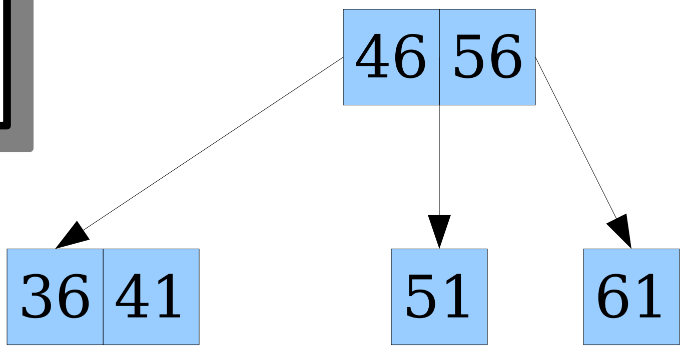
- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.



2-3-4 Tree Deletions

- Most deletions from a 2-3-4 tree require no fixup; we just delete a key from a leaf.
- Some deletions require fixup work to propagate the deletion upward in the tree.

Observation: The only case where a deletion propagates upward is when there are two sibling nodes that each have one key.

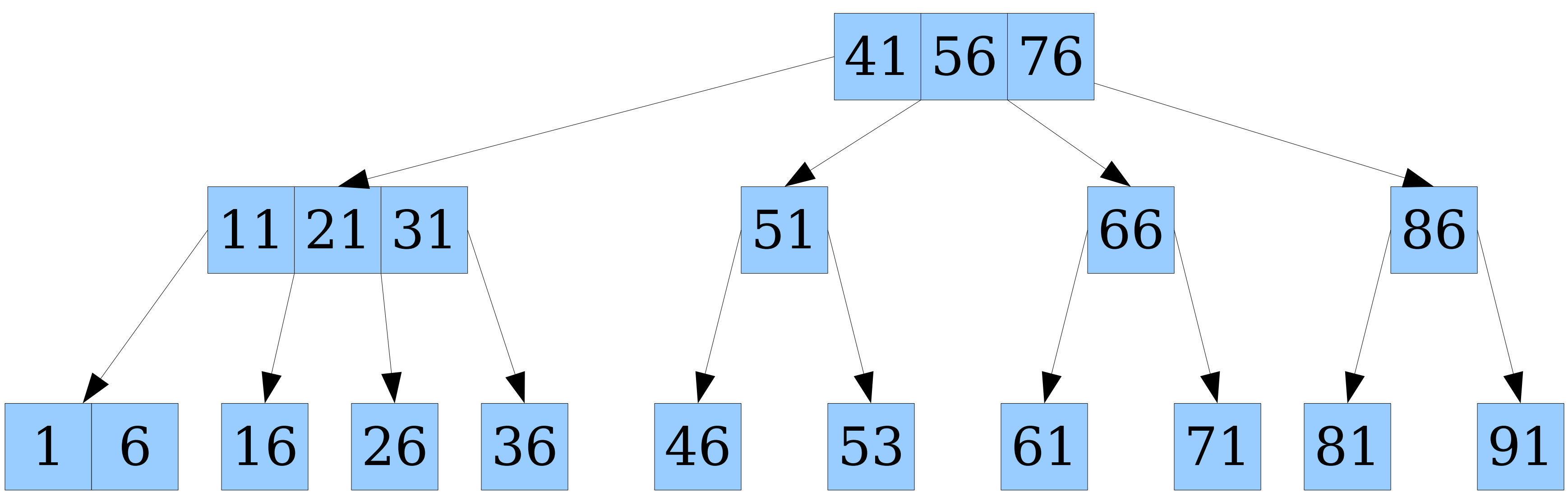


2-3-4 Tree Fixup

- ***Claim:*** The fixup work on 2-3-4 trees is amortized O(1).
- We'll prove this in three steps:
  - First, we'll prove that in any sequence of *m* insertions, the amortized fixup work is O(1).
  - Next, we'll prove that in any sequence of *m* deletions, the amortized fixup work is O(1).
  - Finally, we'll show that in any sequence of insertions and deletions, the amortized fixup work is O(1).

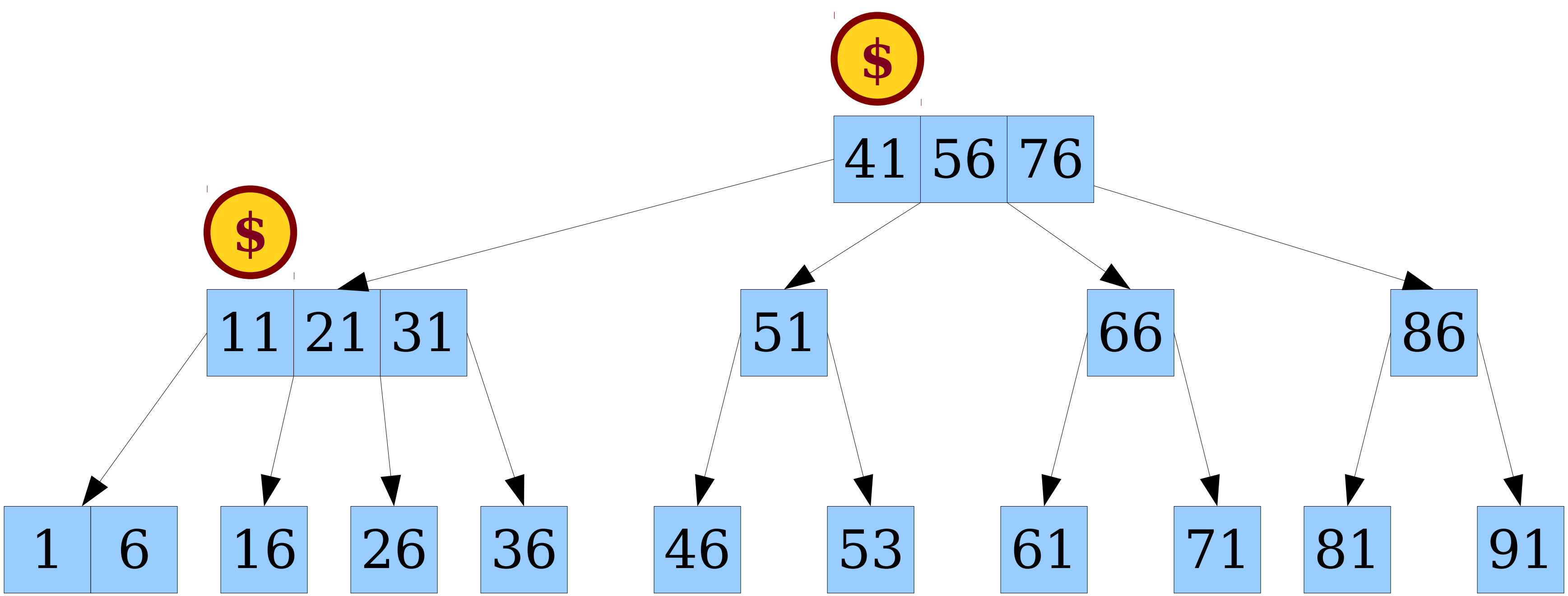
2-3-4 Tree Insertions

- Suppose we only insert and never delete.
- The fixup work for an insertion is proportional to the number of 4-nodes that get split.
- ***Idea:*** Place a credit on each 4-node to pay for future splits.



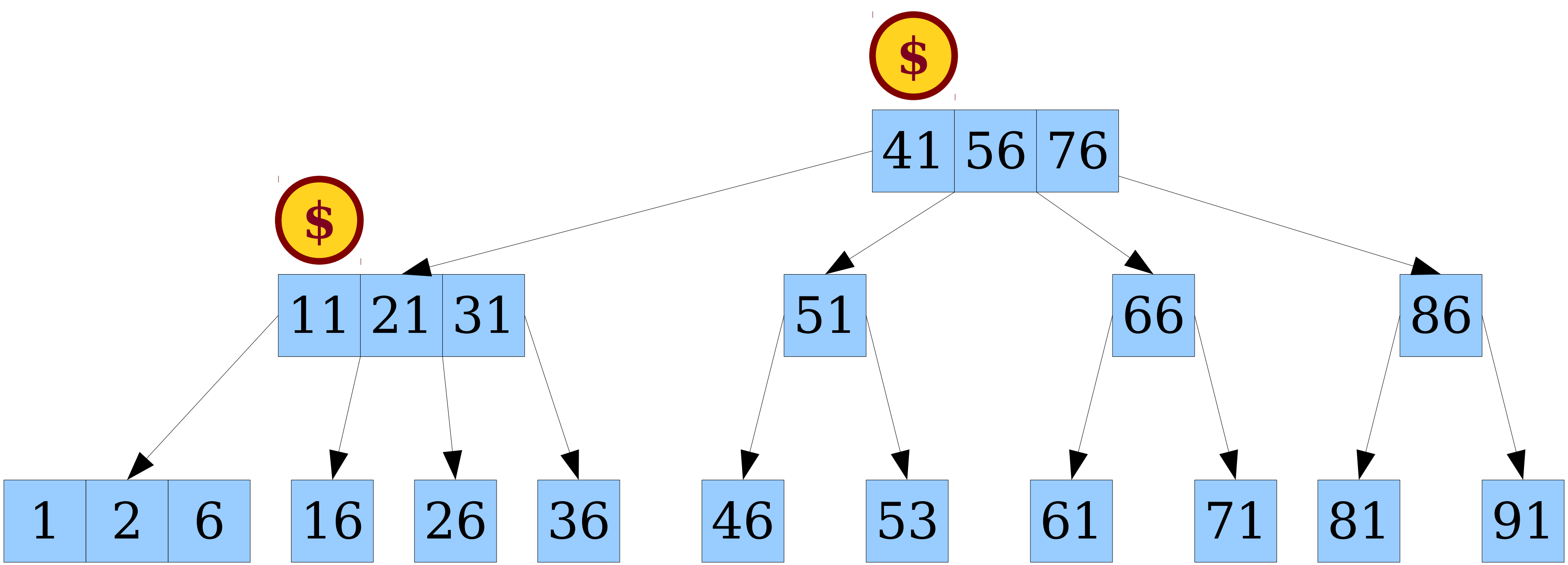
2-3-4 Tree Insertions

- Suppose we only insert and never delete.
- The fixup work for an insertion is proportional to the number of 4-nodes that get split.
- ***Idea:*** Place a credit on each 4-node to pay for future splits.



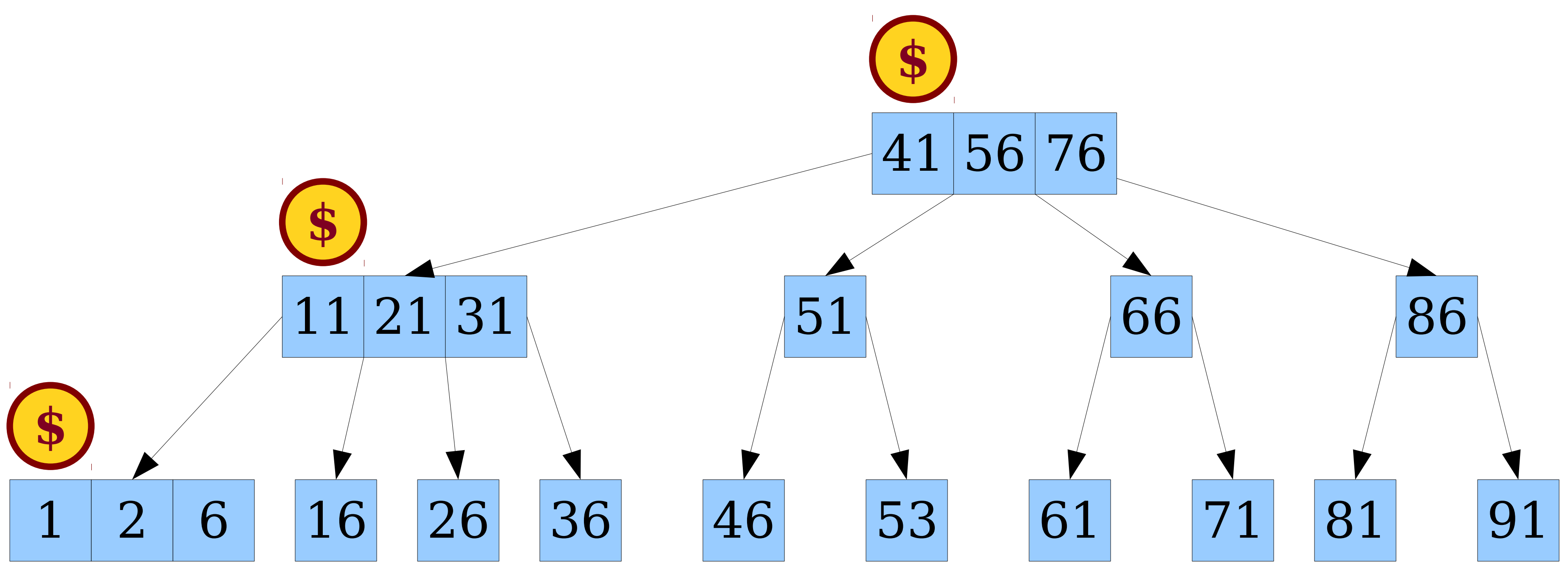
2-3-4 Tree Insertions

- Suppose we only insert and never delete.
- The fixup work for an insertion is proportional to the number of 4-nodes that get split.
- ***Idea:*** Place a credit on each 4-node to pay for future splits.



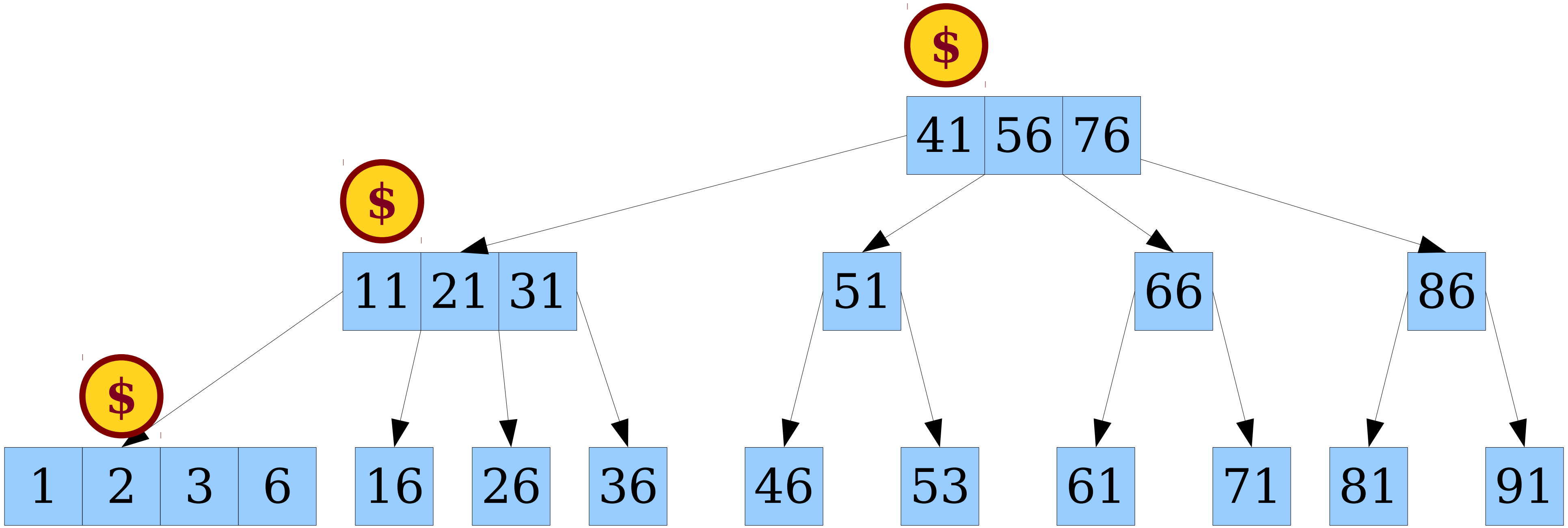
2-3-4 Tree Insertions

- Suppose we only insert and never delete.
- The fixup work for an insertion is proportional to the number of 4-nodes that get split.
- ***Idea:*** Place a credit on each 4-node to pay for future splits.



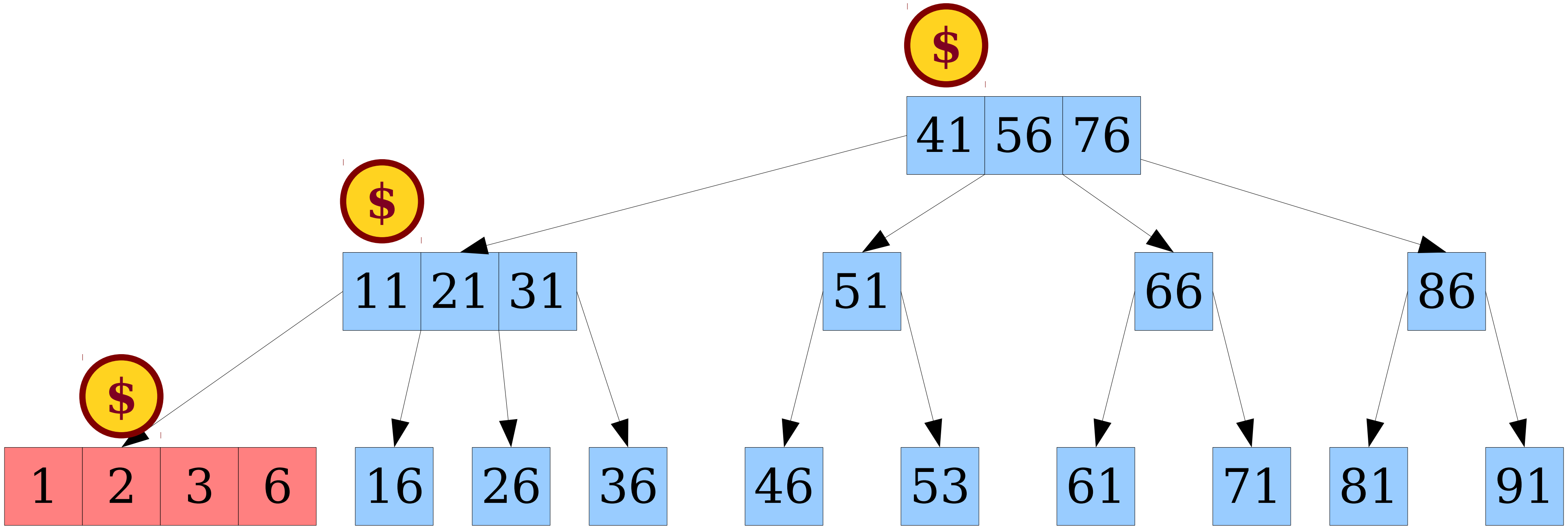
2-3-4 Tree Insertions

- Suppose we only insert and never delete.
- The fixup work for an insertion is proportional to the number of 4-nodes that get split.
- ***Idea:*** Place a credit on each 4-node to pay for future splits.



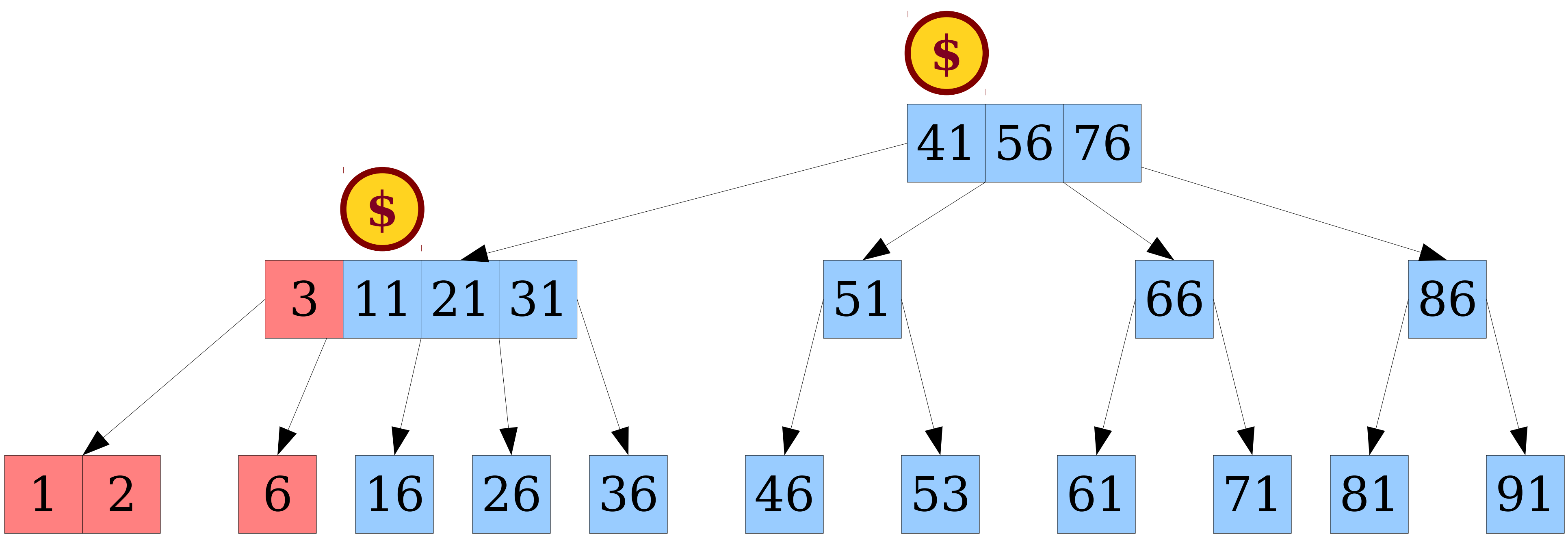
2-3-4 Tree Insertions

- Suppose we only insert and never delete.
- The fixup work for an insertion is proportional to the number of 4-nodes that get split.
- ***Idea:*** Place a credit on each 4-node to pay for future splits.



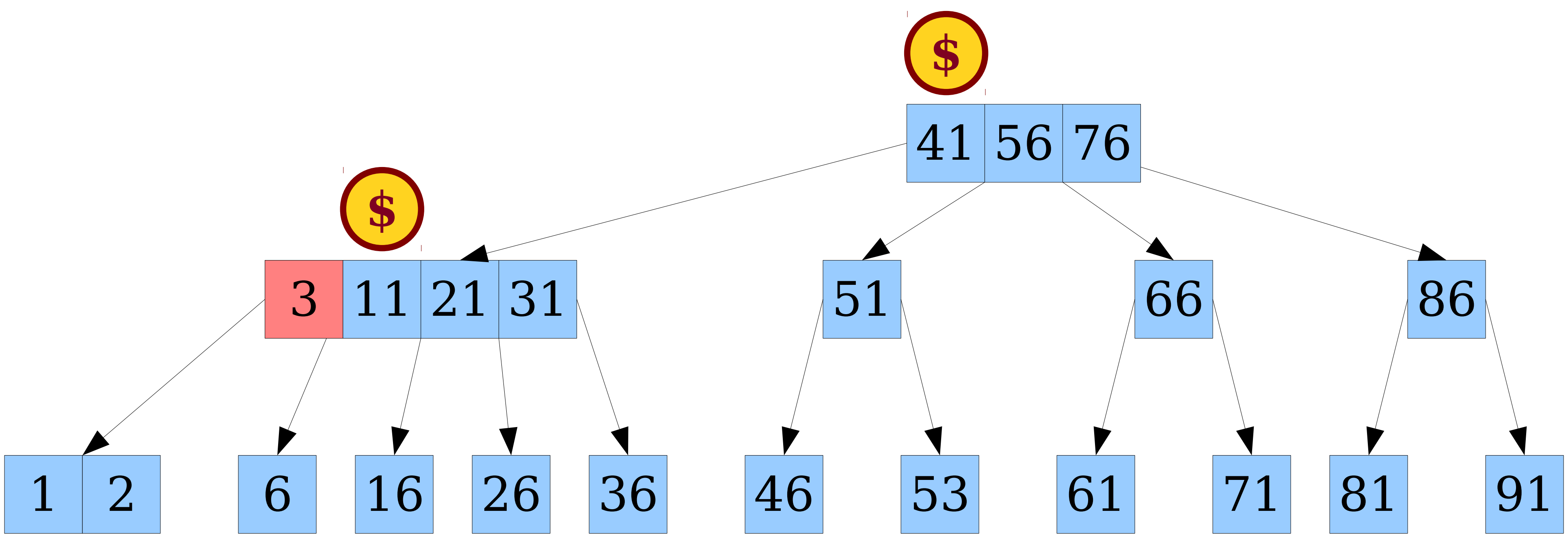
2-3-4 Tree Insertions

- Suppose we only insert and never delete.
- The fixup work for an insertion is proportional to the number of 4-nodes that get split.
- ***Idea:*** Place a credit on each 4-node to pay for future splits.



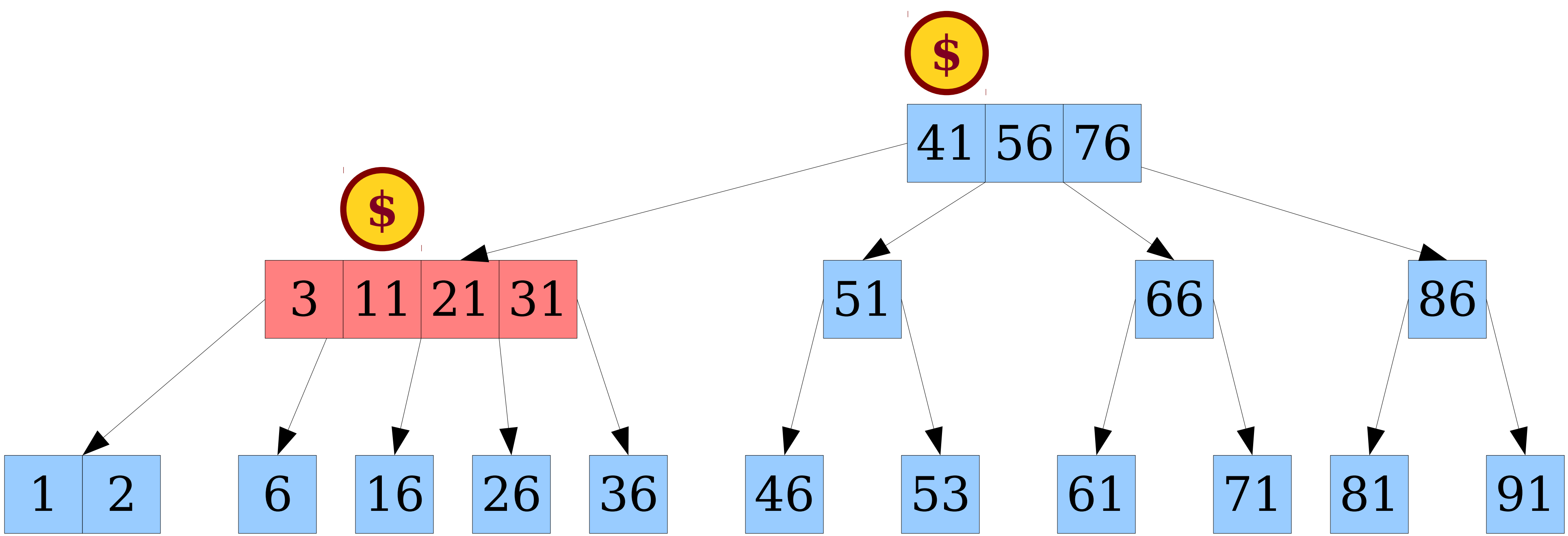
2-3-4 Tree Insertions

- Suppose we only insert and never delete.
- The fixup work for an insertion is proportional to the number of 4-nodes that get split.
- ***Idea:*** Place a credit on each 4-node to pay for future splits.



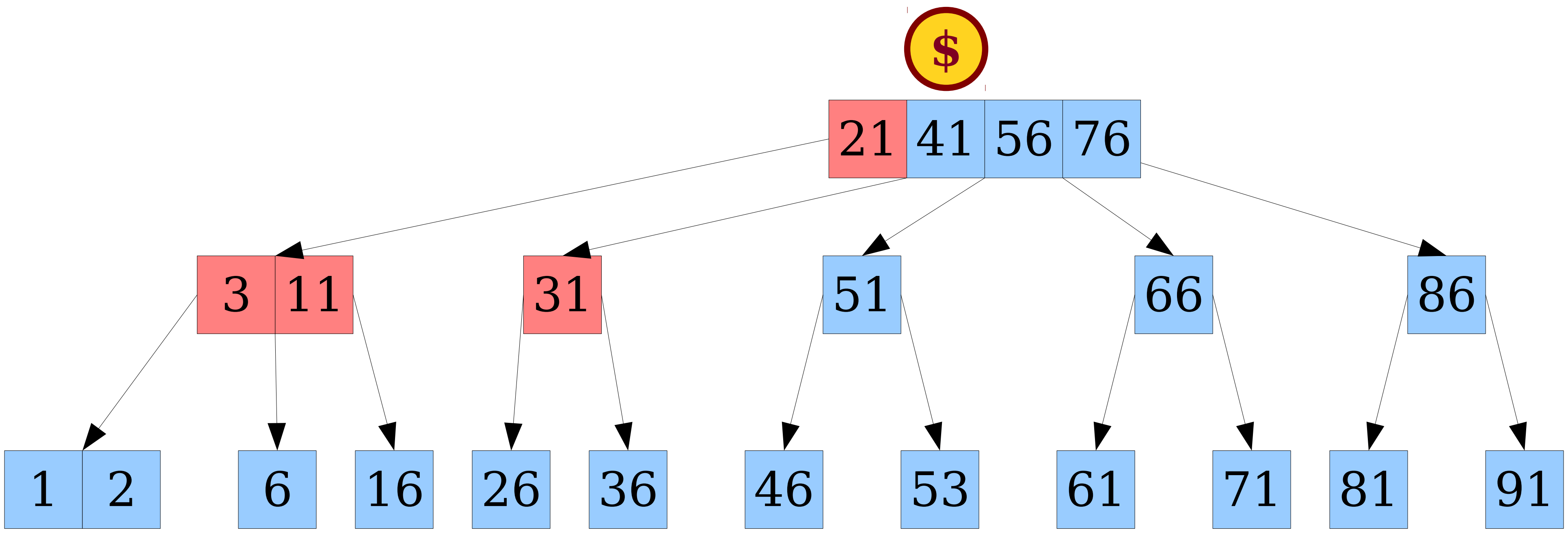
2-3-4 Tree Insertions

- Suppose we only insert and never delete.
- The fixup work for an insertion is proportional to the number of 4-nodes that get split.
- ***Idea:*** Place a credit on each 4-node to pay for future splits.



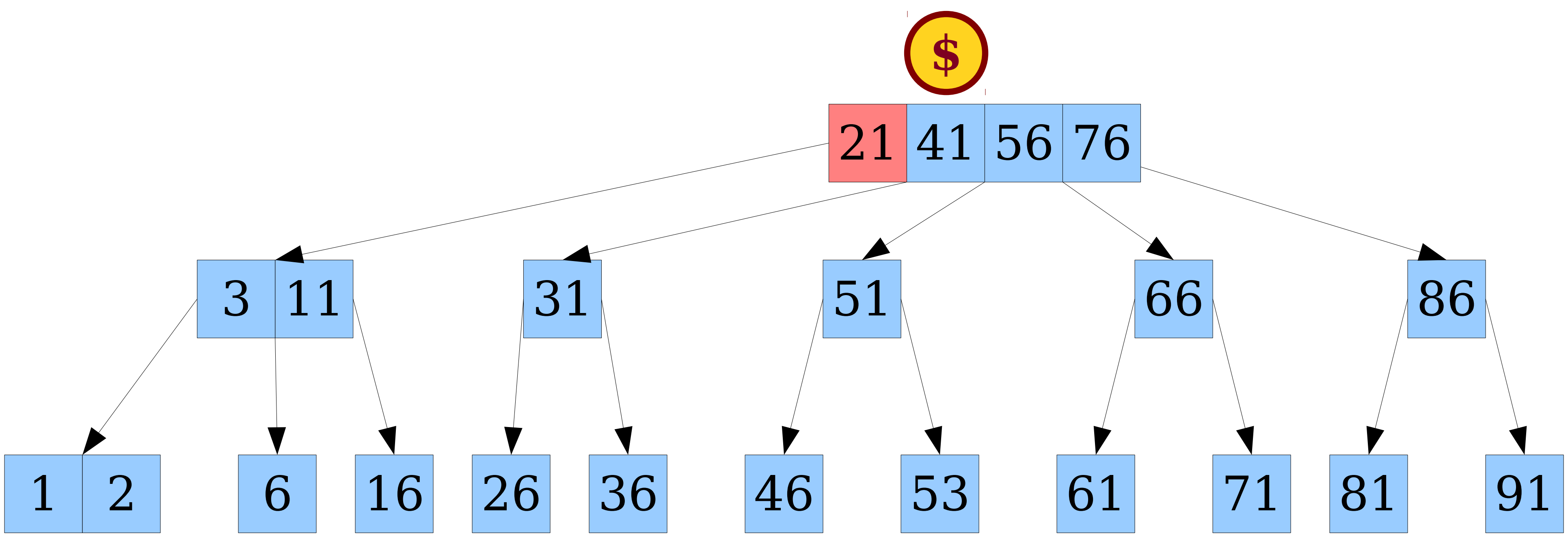
2-3-4 Tree Insertions

- Suppose we only insert and never delete.
- The fixup work for an insertion is proportional to the number of 4-nodes that get split.
- ***Idea:*** Place a credit on each 4-node to pay for future splits.



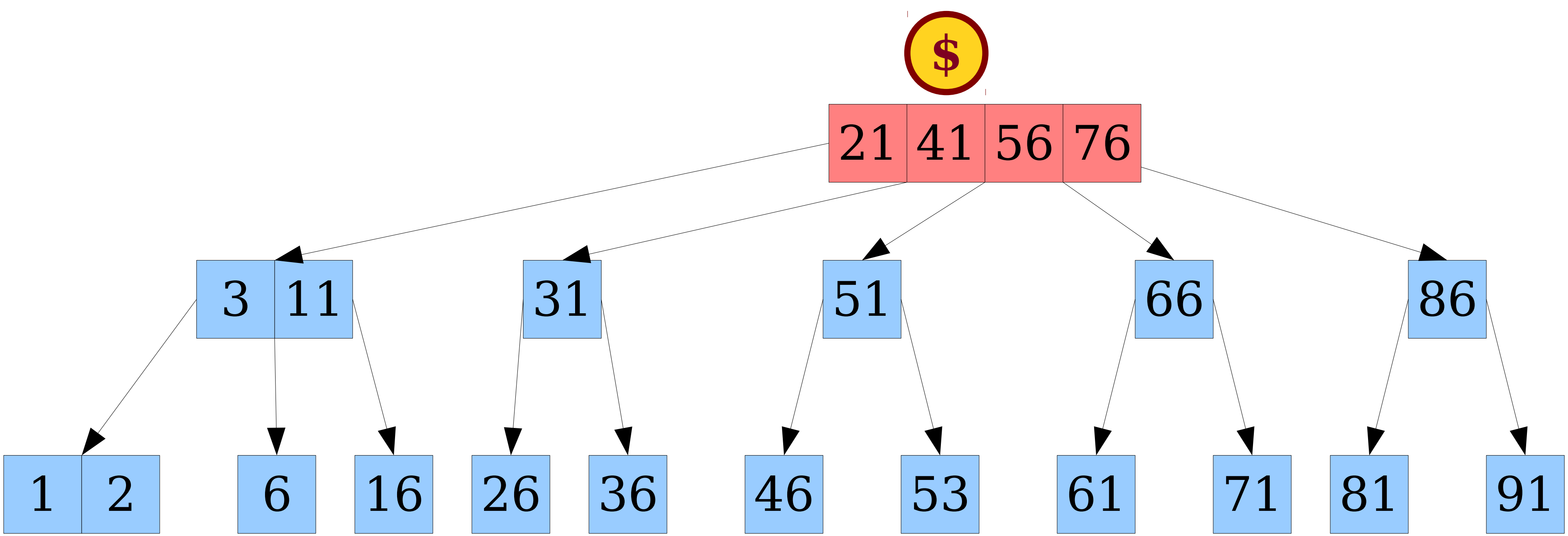
2-3-4 Tree Insertions

- Suppose we only insert and never delete.
- The fixup work for an insertion is proportional to the number of 4-nodes that get split.
- ***Idea:*** Place a credit on each 4-node to pay for future splits.



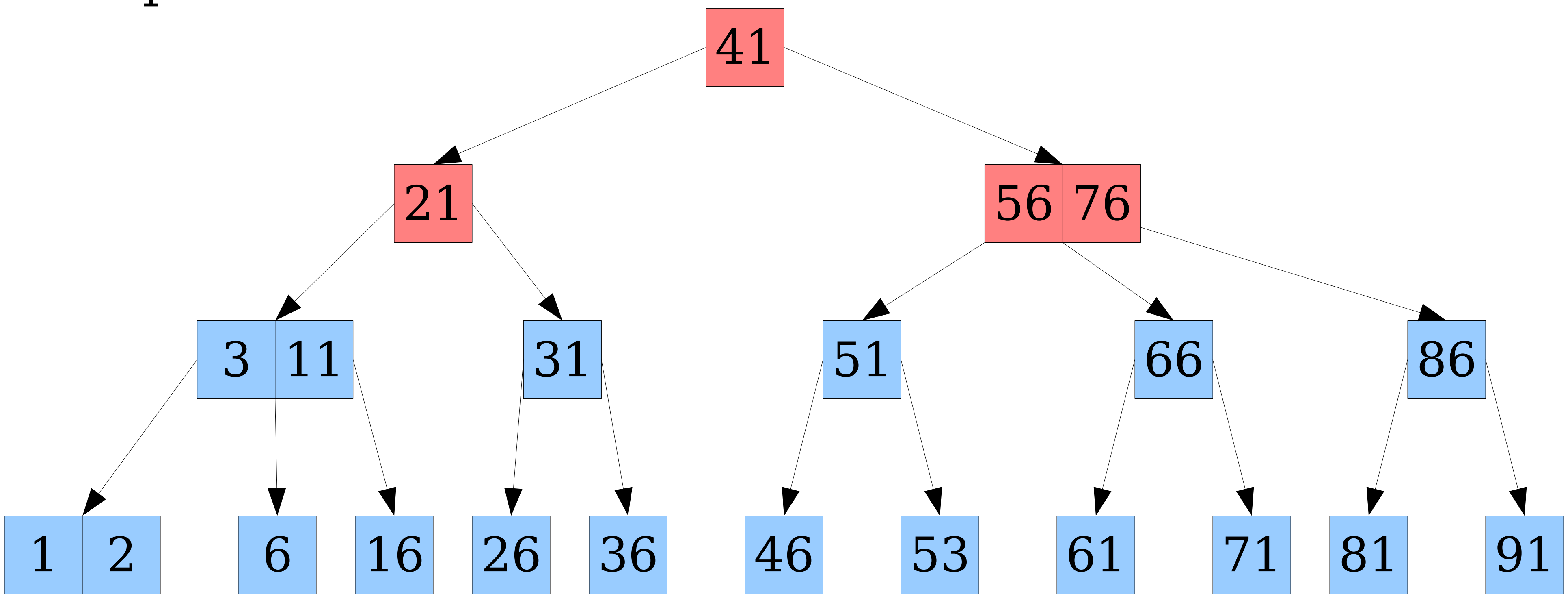
2-3-4 Tree Insertions

- Suppose we only insert and never delete.
- The fixup work for an insertion is proportional to the number of 4-nodes that get split.
- ***Idea:*** Place a credit on each 4-node to pay for future splits.



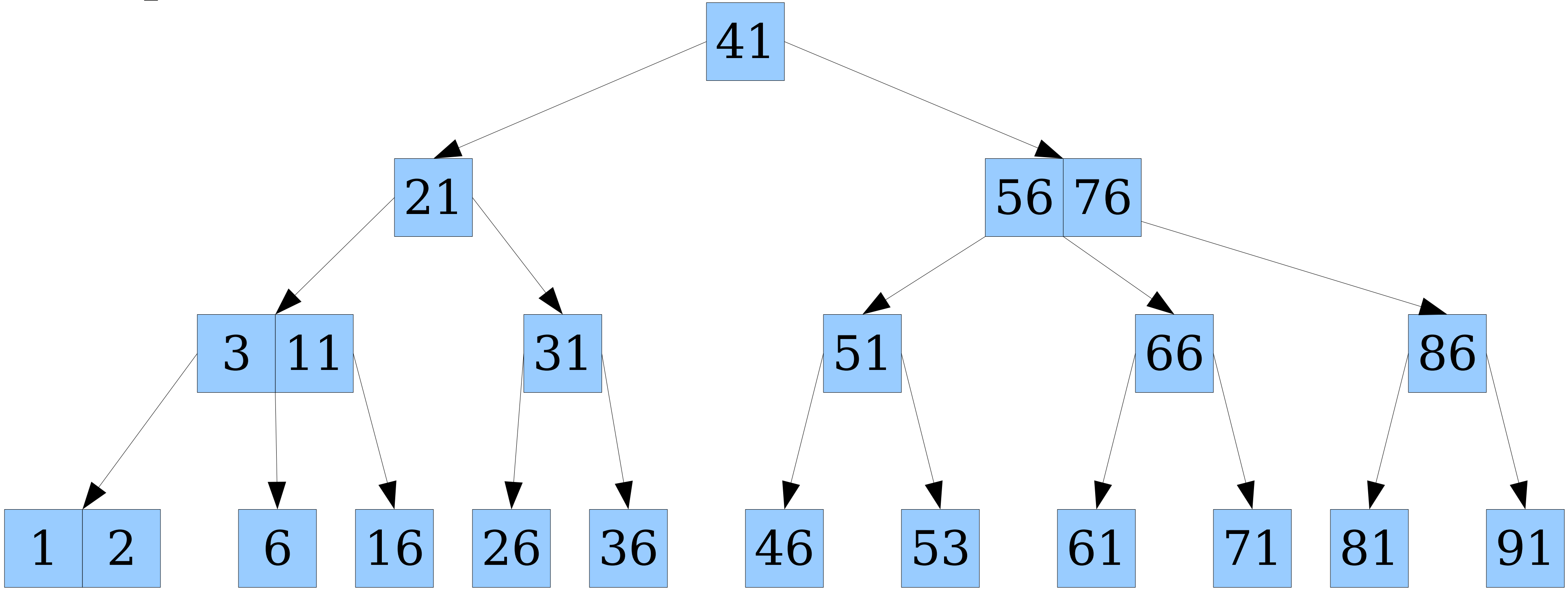
2-3-4 Tree Insertions

- Suppose we only insert and never delete.
- The fixup work for an insertion is proportional to the number of 4-nodes that get split.
- ***Idea:*** Place a credit on each 4-node to pay for future splits.



2-3-4 Tree Insertions

- Suppose we only insert and never delete.
- The fixup work for an insertion is proportional to the number of 4-nodes that get split.
- ***Idea:*** Place a credit on each 4-node to pay for future splits.

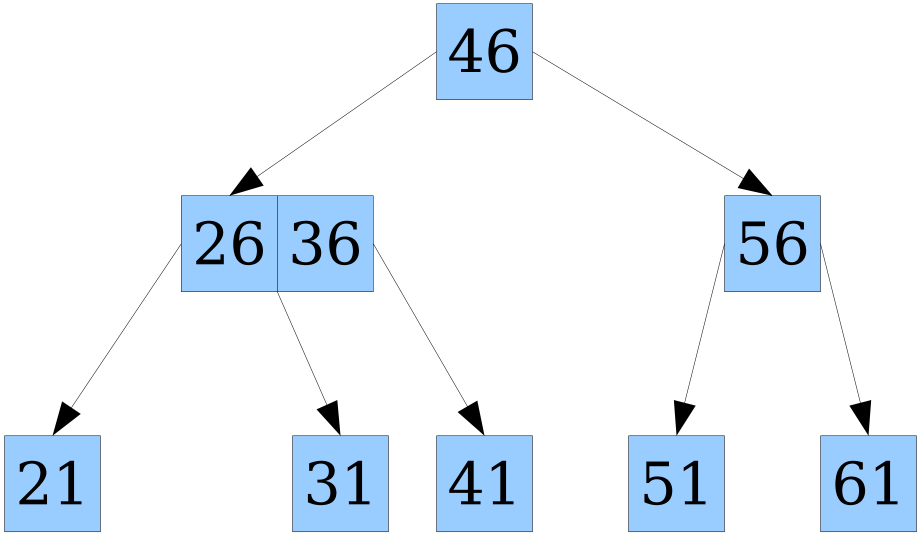


2-3-4 Tree Insertions

- Using the banker's method, we get that pure insertions have O(1) amortized fixup work.
- Could also do this using the potential method.
  - Define Φ to be the number of 4-nodes.
  - Each "light" insertion might introduce a new 4-node, requiring amortized O(1) work.
  - Each "heavy" insertion splits *k* 4-nodes and decreases the potential by *k* for O(1) amortized work.

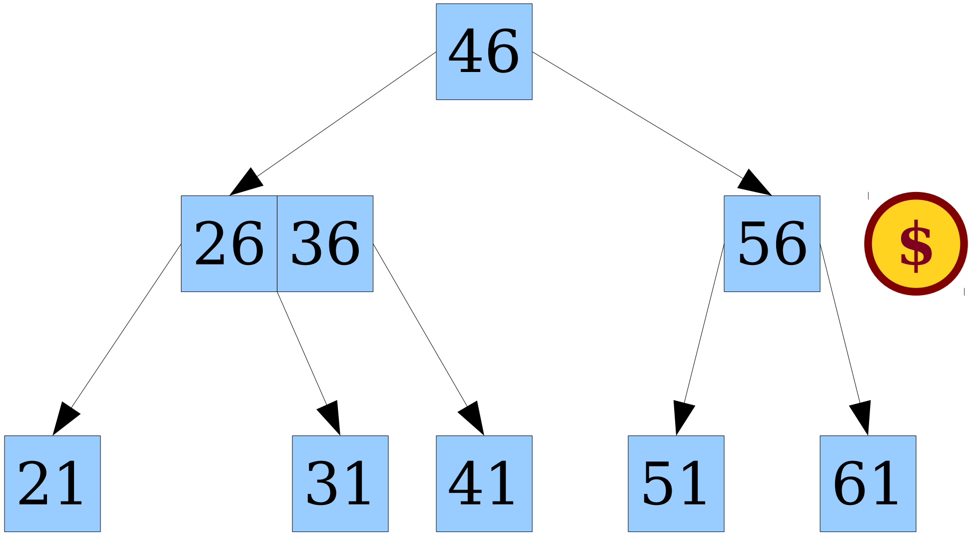
2-3-4 Tree Deletions

- Suppose we only delete and never insert.
- The fixup work per layer is O(1) and only propagates if we combine three 2-nodes together into a 4-node.
- ***Idea:*** Place a credit on each 2-node whose children are 2-nodes (call them "tiny triangles.")



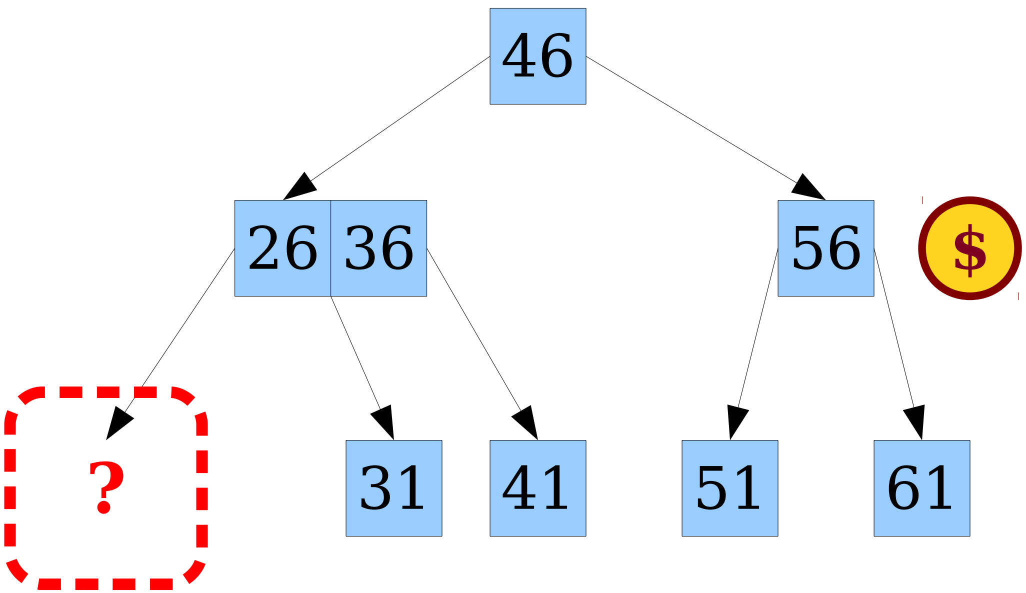
2-3-4 Tree Deletions

- Suppose we only delete and never insert.
- The fixup work per layer is O(1) and only propagates if we combine three 2-nodes together into a 4-node.
- ***Idea:*** Place a credit on each 2-node whose children are 2-nodes (call them "tiny triangles.")



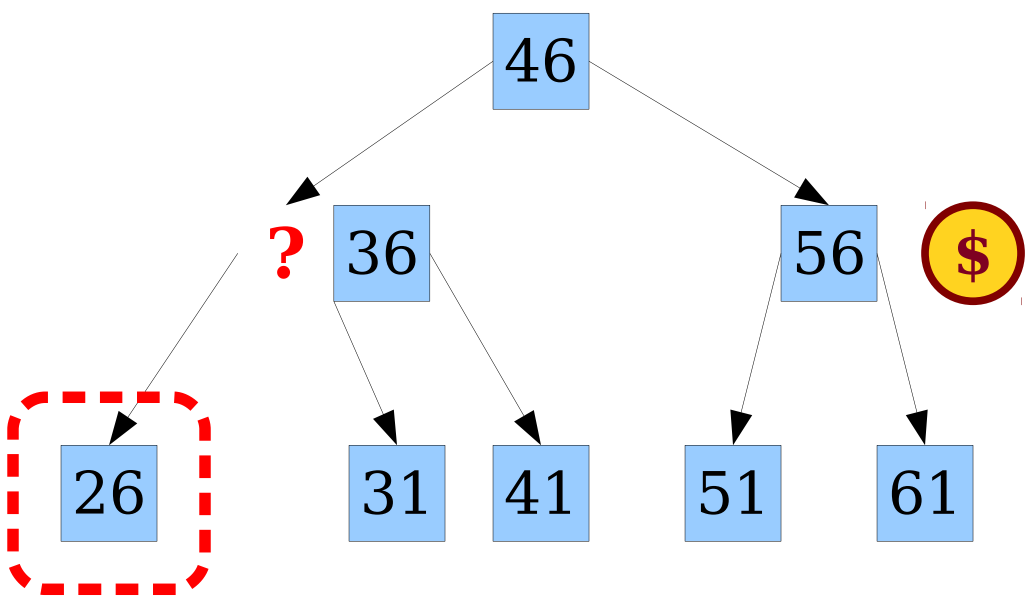
2-3-4 Tree Deletions

- Suppose we only delete and never insert.
- The fixup work per layer is O(1) and only propagates if we combine three 2-nodes together into a 4-node.
- ***Idea:*** Place a credit on each 2-node whose children are 2-nodes (call them "tiny triangles.")



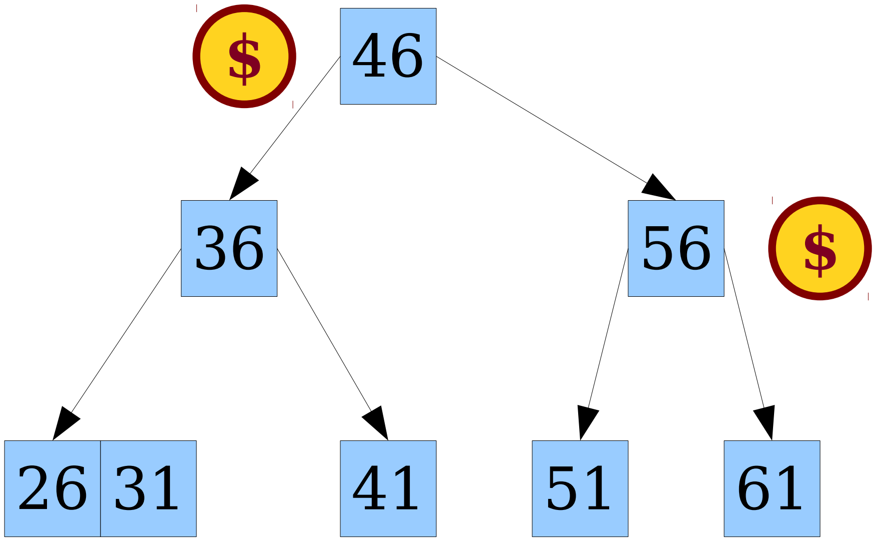
2-3-4 Tree Deletions

- Suppose we only delete and never insert.
- The fixup work per layer is O(1) and only propagates if we combine three 2-nodes together into a 4-node.
- ***Idea:*** Place a credit on each 2-node whose children are 2-nodes (call them "tiny triangles.")



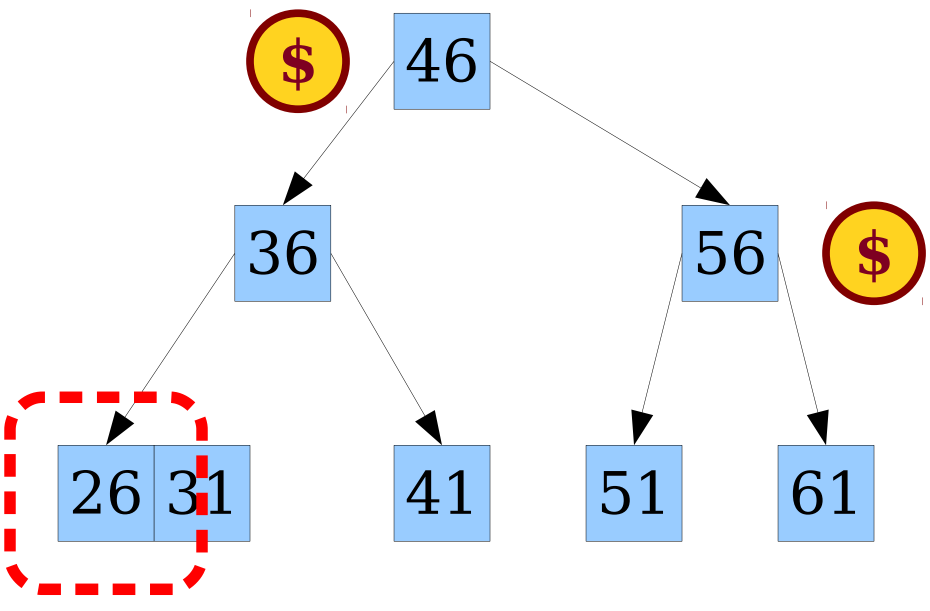
2-3-4 Tree Deletions

- Suppose we only delete and never insert.
- The fixup work per layer is O(1) and only propagates if we combine three 2-nodes together into a 4-node.
- ***Idea:*** Place a credit on each 2-node whose children are 2-nodes (call them "tiny triangles.")



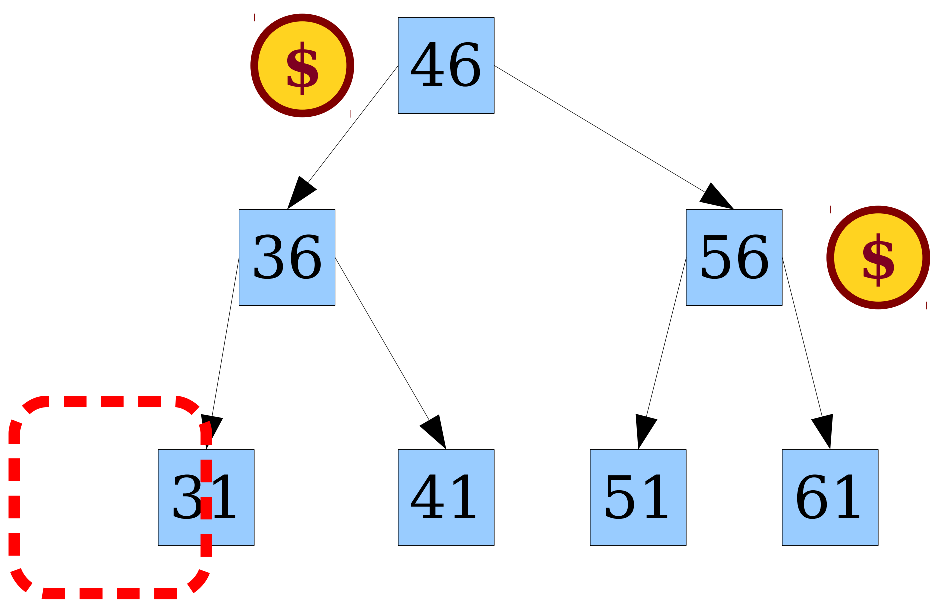
2-3-4 Tree Deletions

- Suppose we only delete and never insert.
- The fixup work per layer is O(1) and only propagates if we combine three 2-nodes together into a 4-node.
- ***Idea:*** Place a credit on each 2-node whose children are 2-nodes (call them "tiny triangles.")



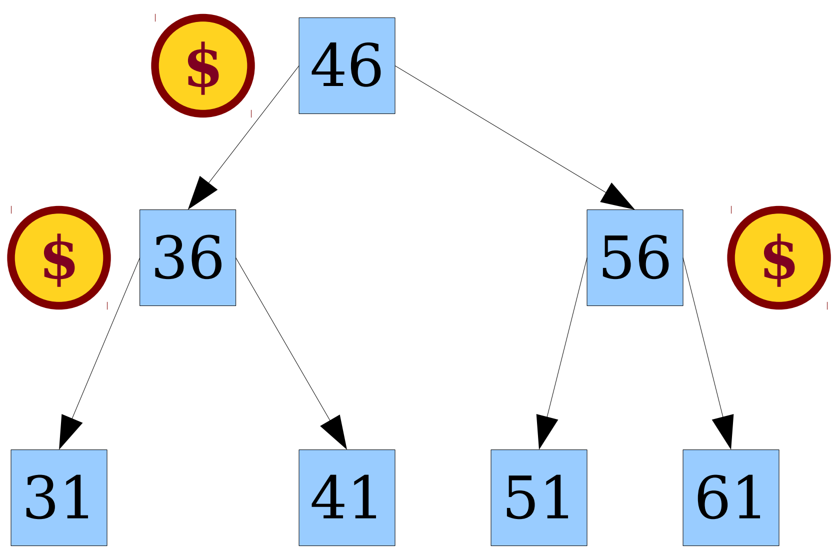
2-3-4 Tree Deletions

- Suppose we only delete and never insert.
- The fixup work per layer is O(1) and only propagates if we combine three 2-nodes together into a 4-node.
- ***Idea:*** Place a credit on each 2-node whose children are 2-nodes (call them "tiny triangles.")



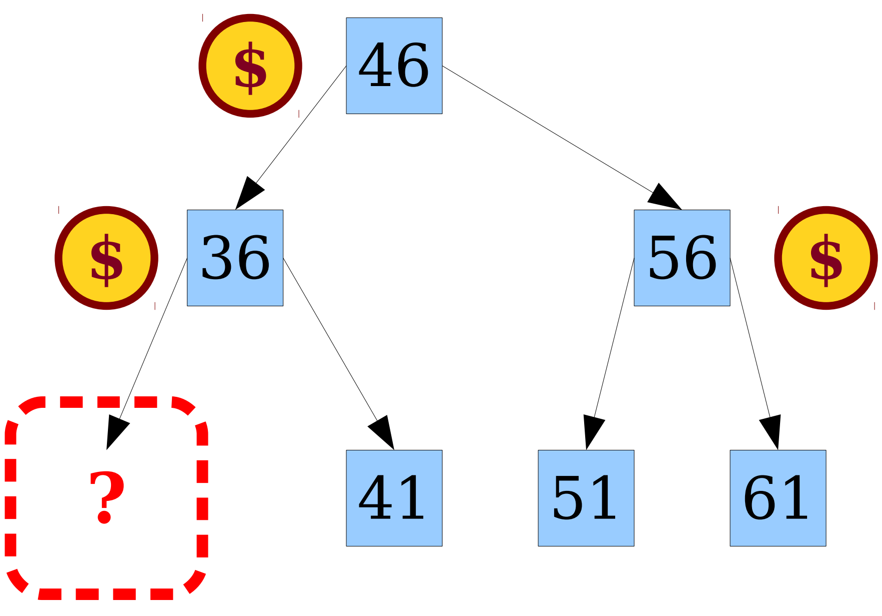
2-3-4 Tree Deletions

- Suppose we only delete and never insert.
- The fixup work per layer is O(1) and only propagates if we combine three 2-nodes together into a 4-node.
- ***Idea:*** Place a credit on each 2-node whose children are 2-nodes (call them "tiny triangles.")



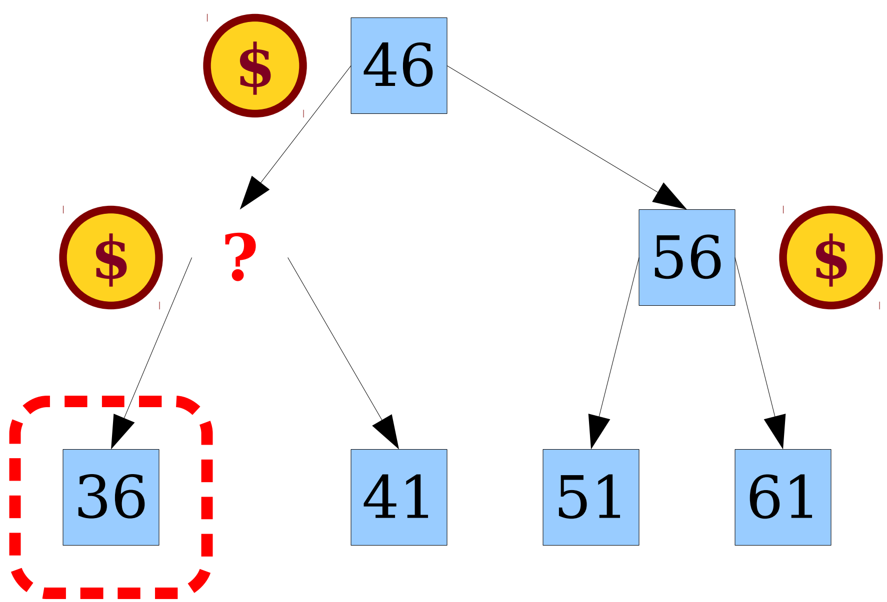
2-3-4 Tree Deletions

- Suppose we only delete and never insert.
- The fixup work per layer is O(1) and only propagates if we combine three 2-nodes together into a 4-node.
- ***Idea:*** Place a credit on each 2-node whose children are 2-nodes (call them "tiny triangles.")



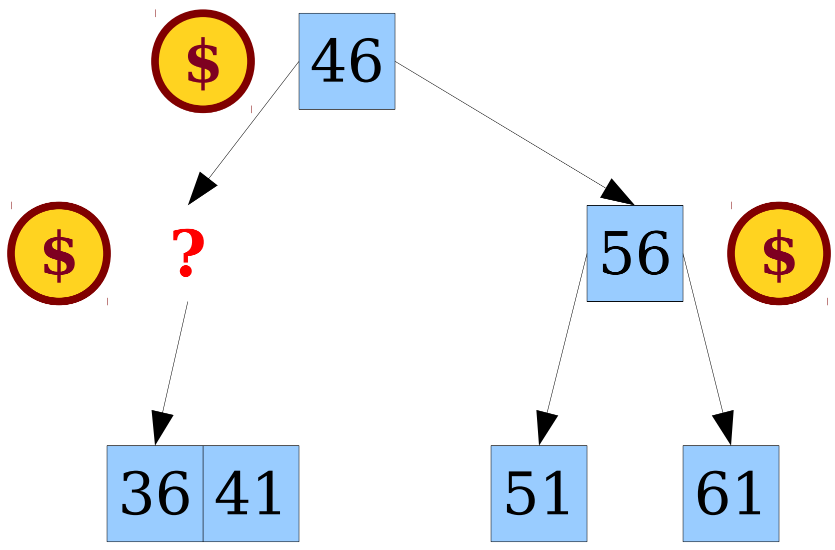
2-3-4 Tree Deletions

- Suppose we only delete and never insert.
- The fixup work per layer is O(1) and only propagates if we combine three 2-nodes together into a 4-node.
- ***Idea:*** Place a credit on each 2-node whose children are 2-nodes (call them "tiny triangles.")



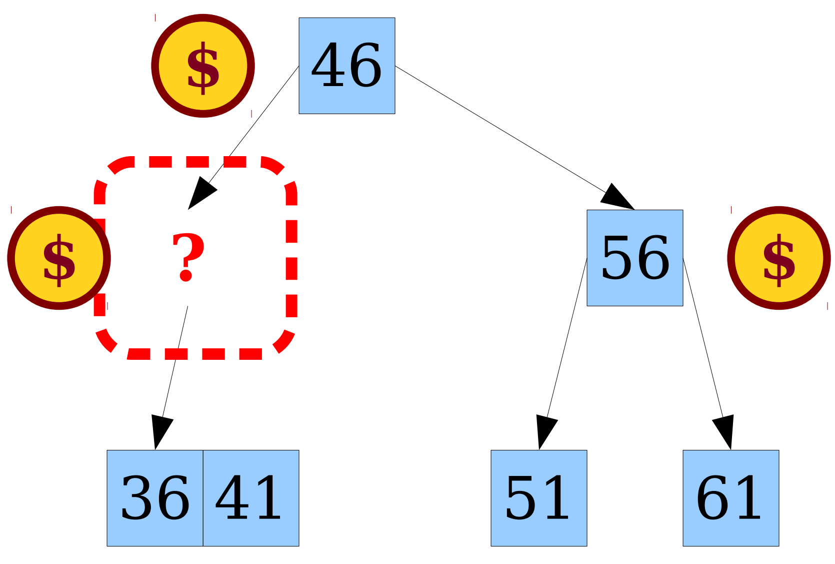
2-3-4 Tree Deletions

- Suppose we only delete and never insert.
- The fixup work per layer is O(1) and only propagates if we combine three 2-nodes together into a 4-node.
- ***Idea:*** Place a credit on each 2-node whose children are 2-nodes (call them “tiny triangles.”)



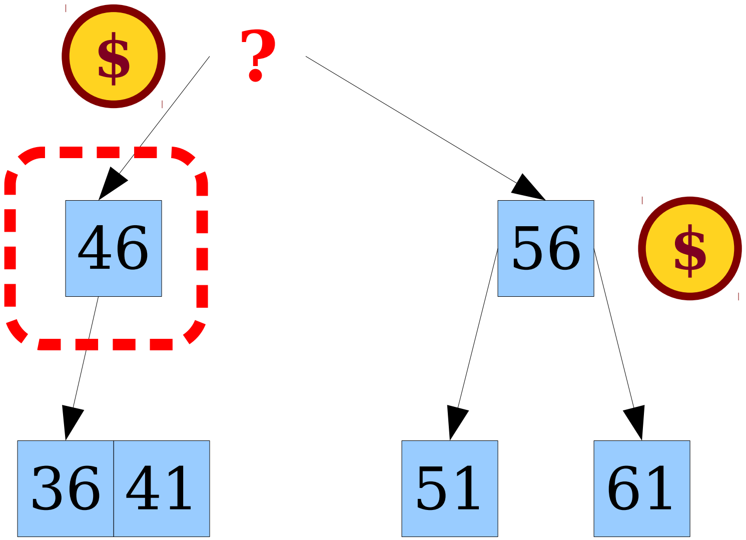
2-3-4 Tree Deletions

- Suppose we only delete and never insert.
- The fixup work per layer is O(1) and only propagates if we combine three 2-nodes together into a 4-node.
- ***Idea:*** Place a credit on each 2-node whose children are 2-nodes (call them "tiny triangles.")



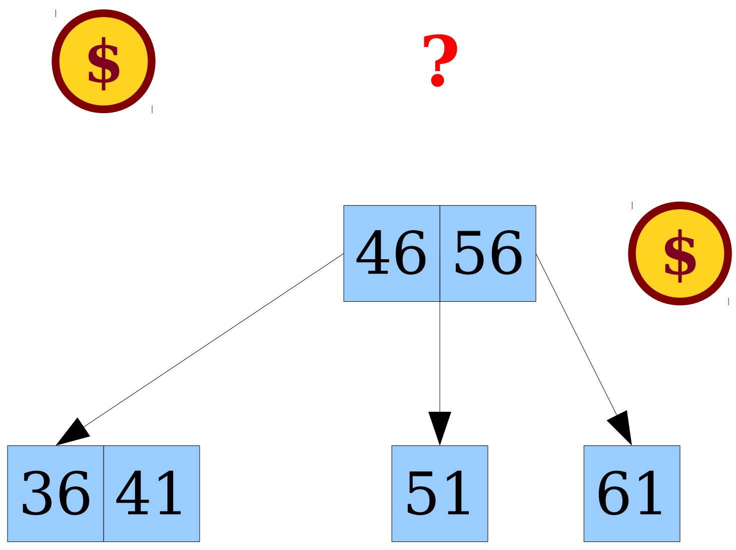
2-3-4 Tree Deletions

- Suppose we only delete and never insert.
- The fixup work per layer is O(1) and only propagates if we combine three 2-nodes together into a 4-node.
- ***Idea:*** Place a credit on each 2-node whose children are 2-nodes (call them “tiny triangles.”)



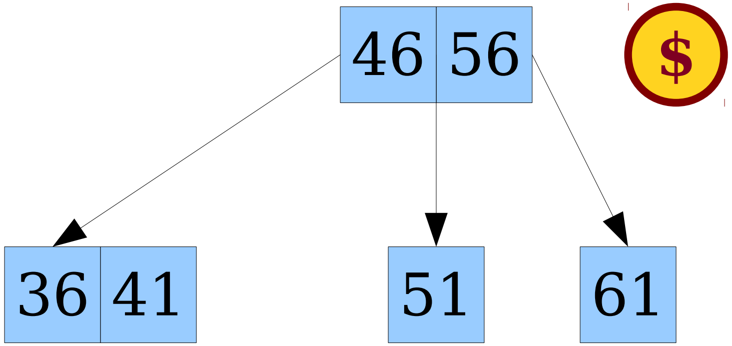
2-3-4 Tree Deletions

- Suppose we only delete and never insert.
- The fixup work per layer is O(1) and only propagates if we combine three 2-nodes together into a 4-node.
- ***Idea:*** Place a credit on each 2-node whose children are 2-nodes (call them "tiny triangles.")



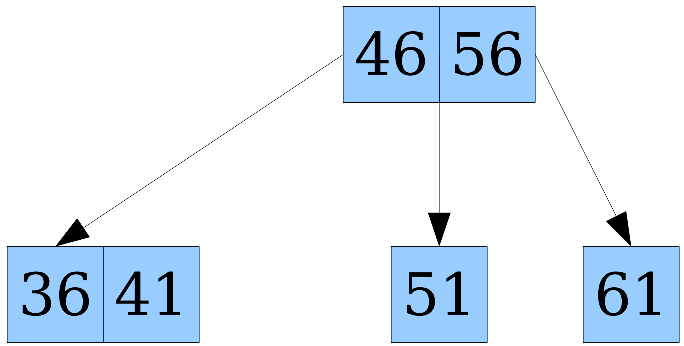
2-3-4 Tree Deletions

- Suppose we only delete and never insert.
- The fixup work per layer is O(1) and only propagates if we combine three 2-nodes together into a 4-node.
- ***Idea:*** Place a credit on each 2-node whose children are 2-nodes (call them “tiny triangles.”)



2-3-4 Tree Deletions

- Suppose we only delete and never insert.
- The fixup work per layer is O(1) and only propagates if we combine three 2-nodes together into a 4-node.
- ***Idea:*** Place a credit on each 2-node whose children are 2-nodes (call them "tiny triangles.")

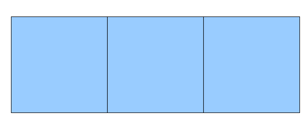
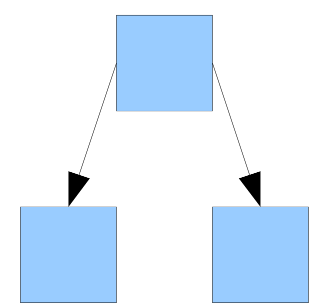


2-3-4 Tree Deletions

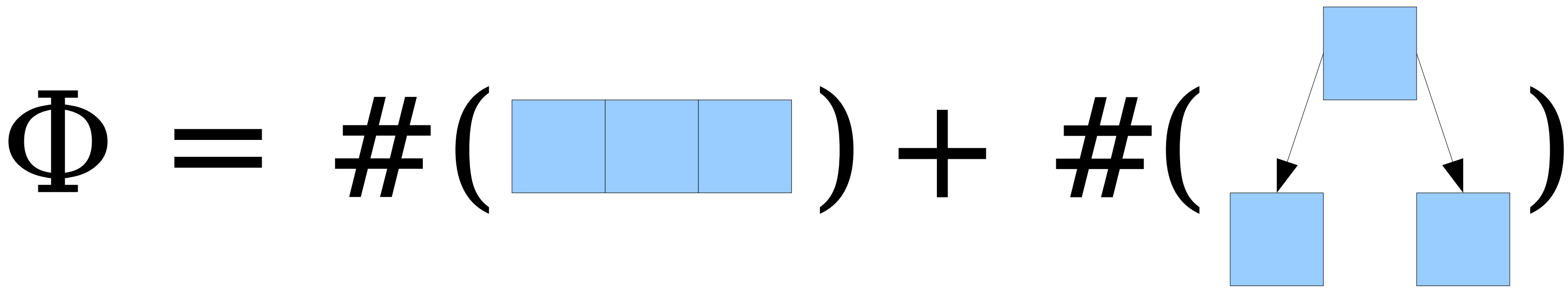
- Using the banker's method, we get that pure deletions have O(1) amortized fixup work.
- Could also do this using the potential method.
  - Define Φ to be the number of 2-nodes with two 2-node children (call these “tiny triangles.”)
  - Each “light” deletion might introduce two tiny triangles: one at the node where the deletion ended and one right above it. Amortized time is O(1).
  - Each “heavy” deletion combines k tiny triangles and decreases the potential by at least k. Amortized time is O(1).

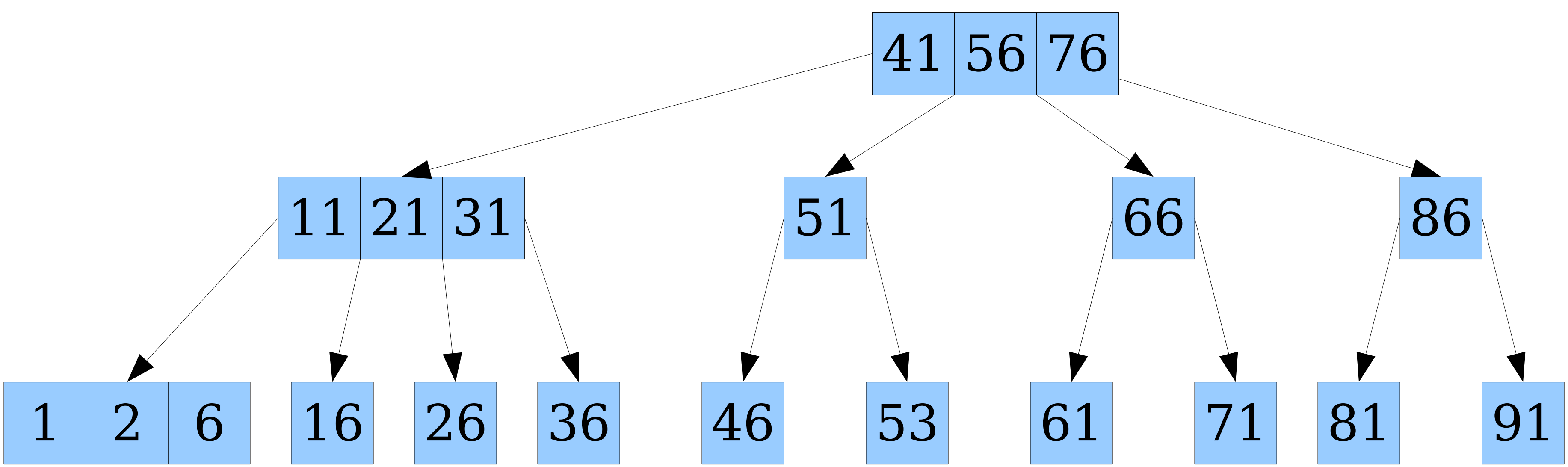
Combining the Two

- We've shown that pure insertions and pure deletions require O(1) amortized fixup time.
- What about interleaved insertions and deletions?
- ***Initial idea:*** Use a potential function that's the sum of the two previous potential functions.
- Φ is the number of 4-nodes plus the number of tiny triangles.

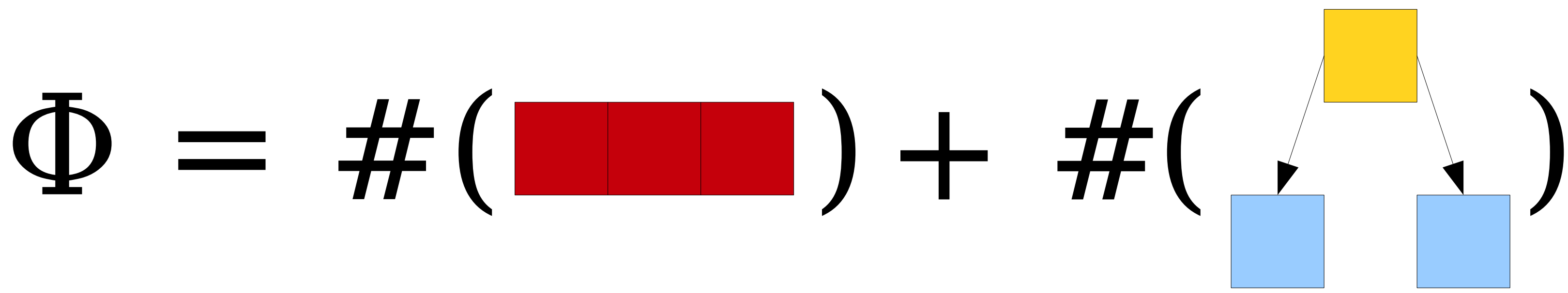
$\Phi = \#($  $) + \#($  $)$

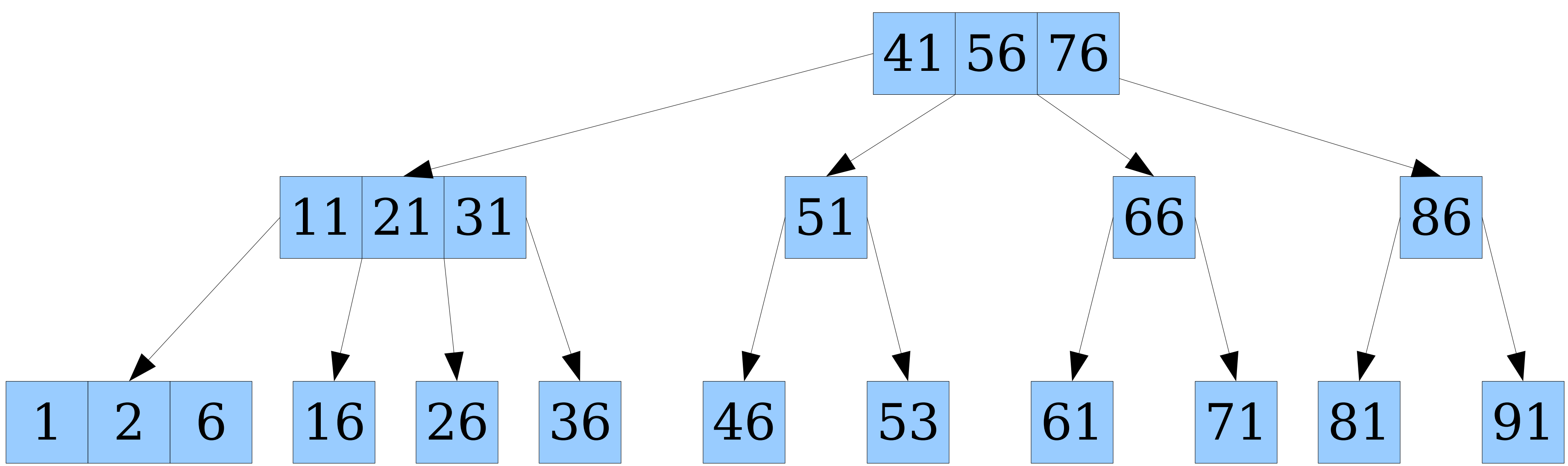
A Potential Issue



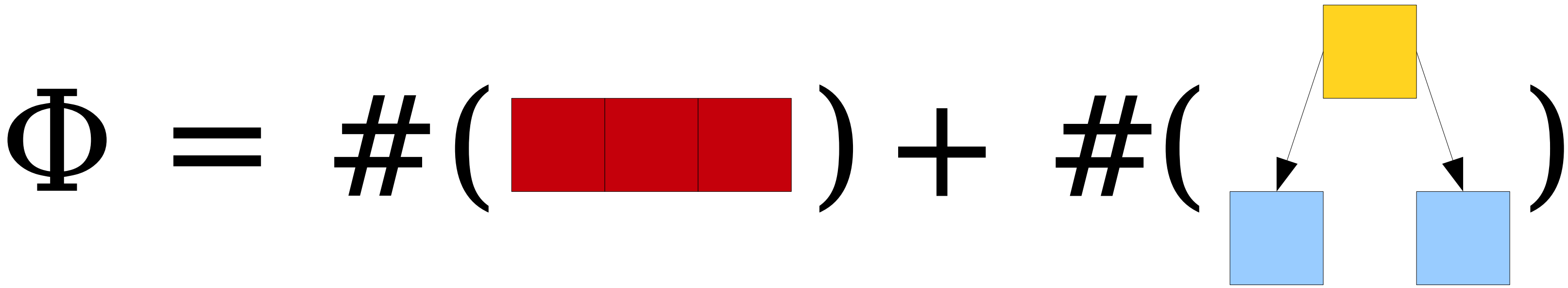


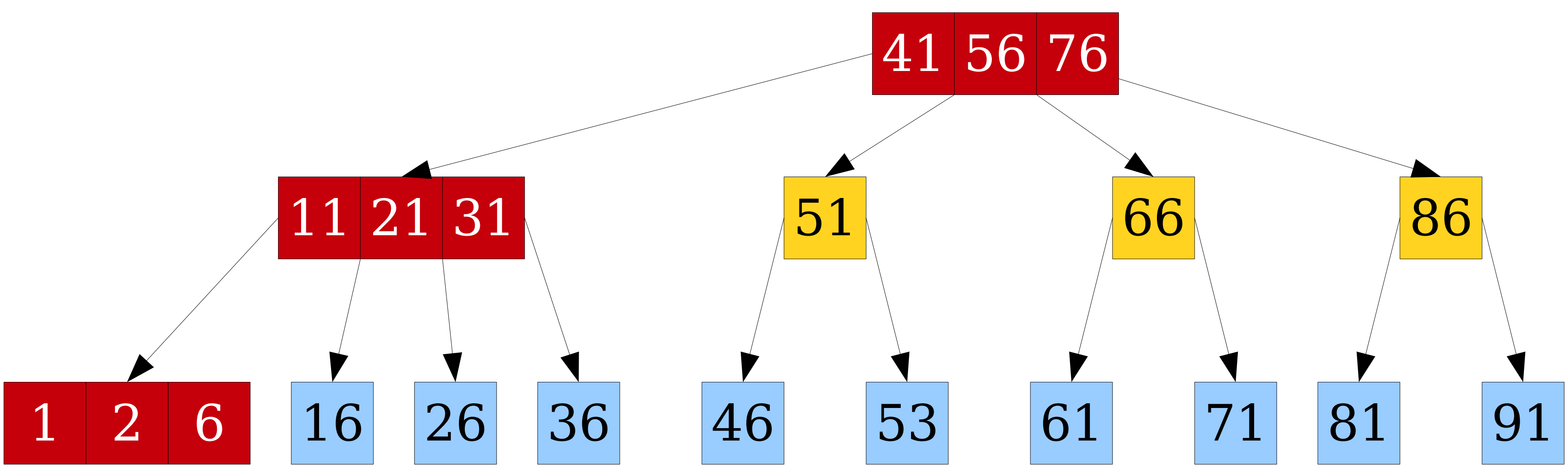
A Potential Issue



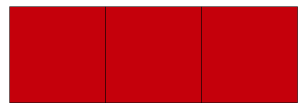
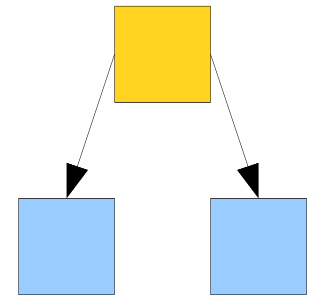


A Potential Issue

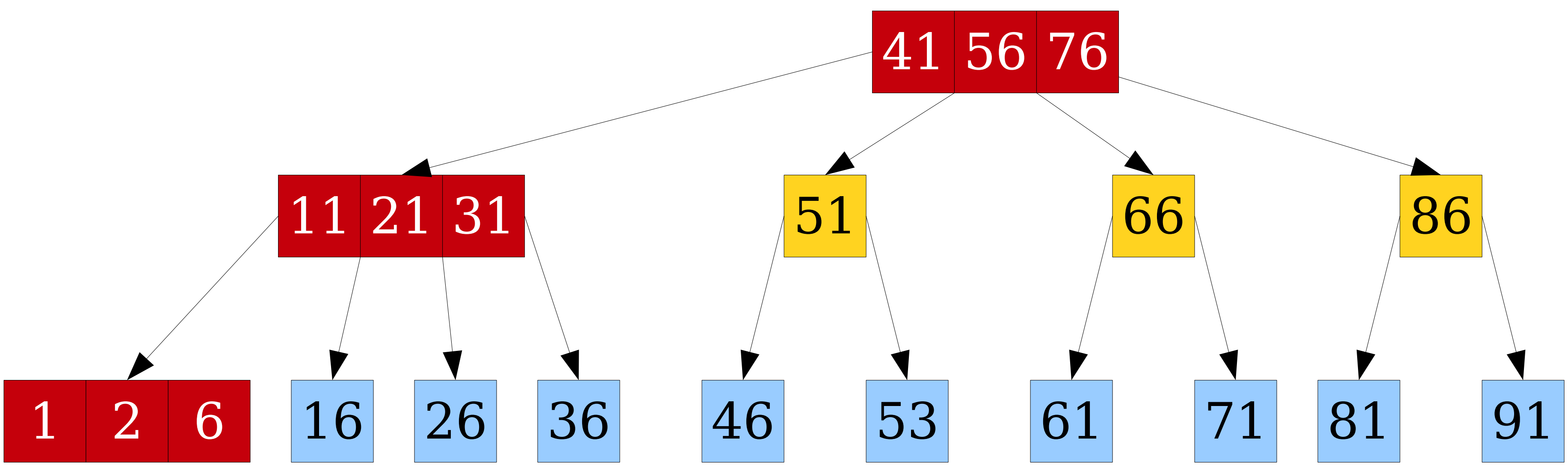




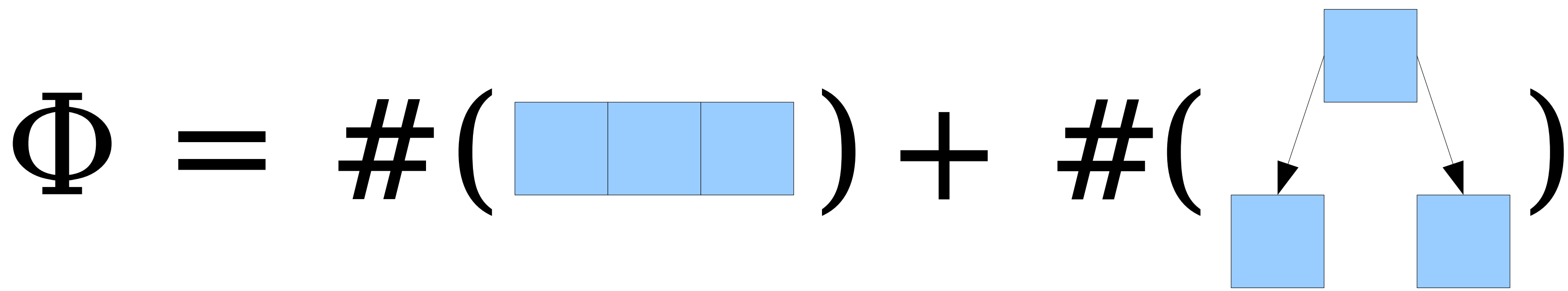
A Potential Issue

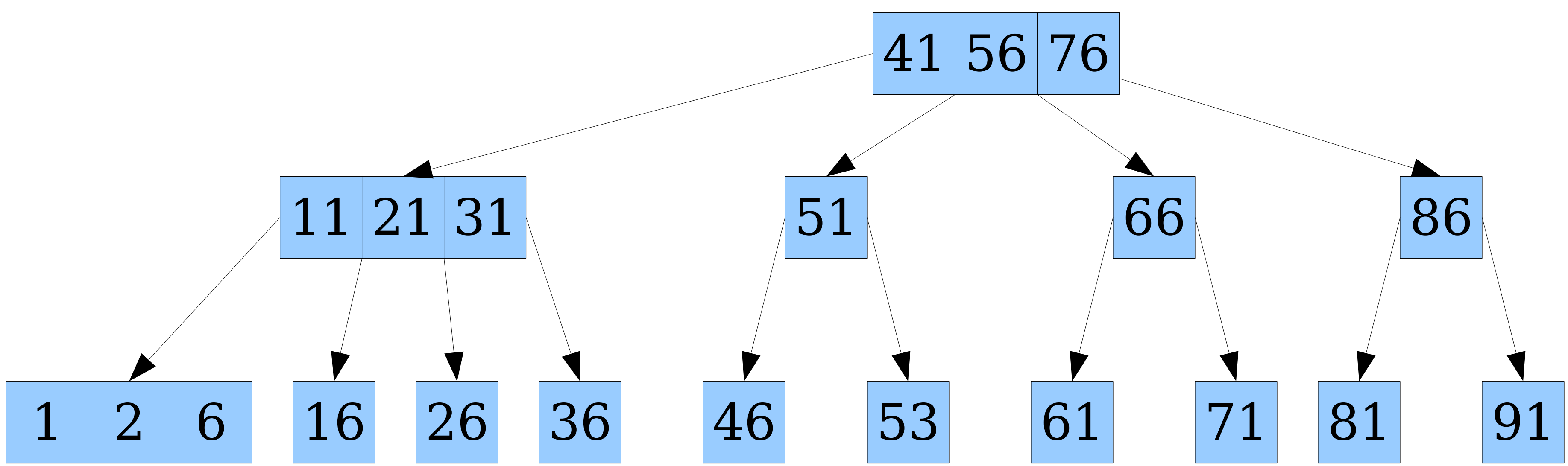
$\Phi = \#($  $) + \#($  $)$

$= \mathbf{6}$

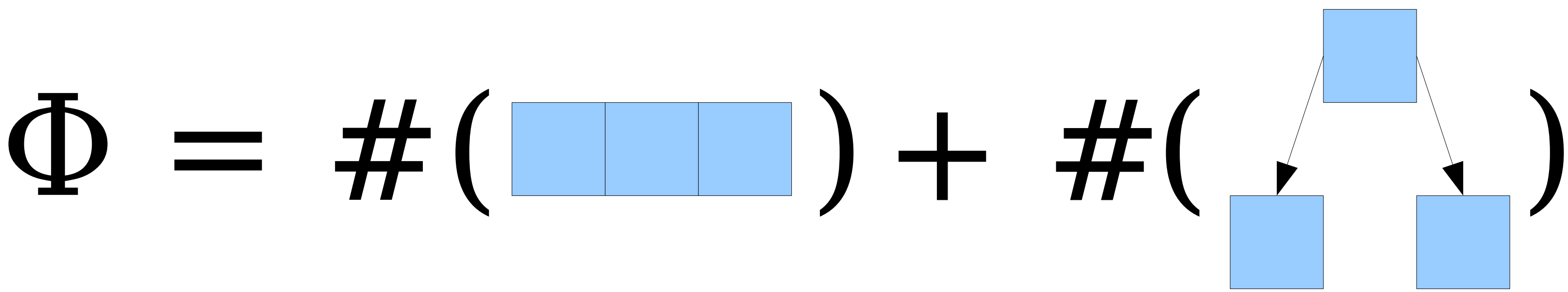


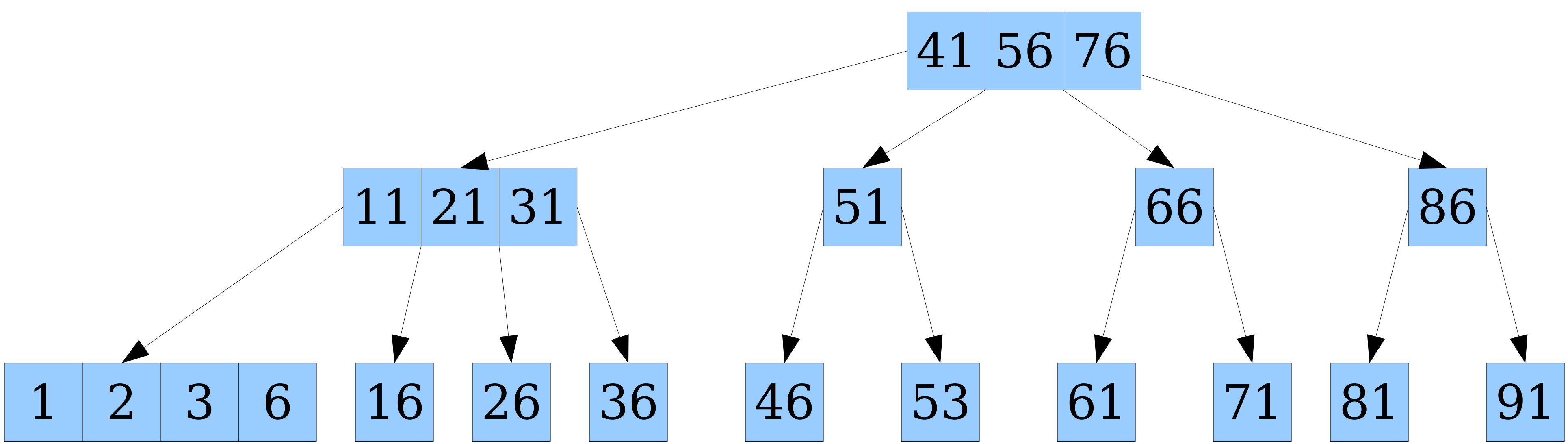
A Potential Issue



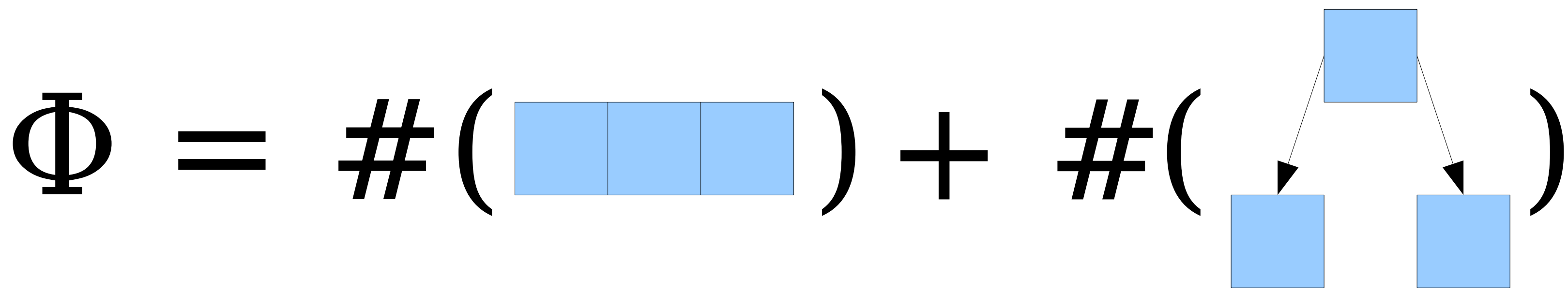


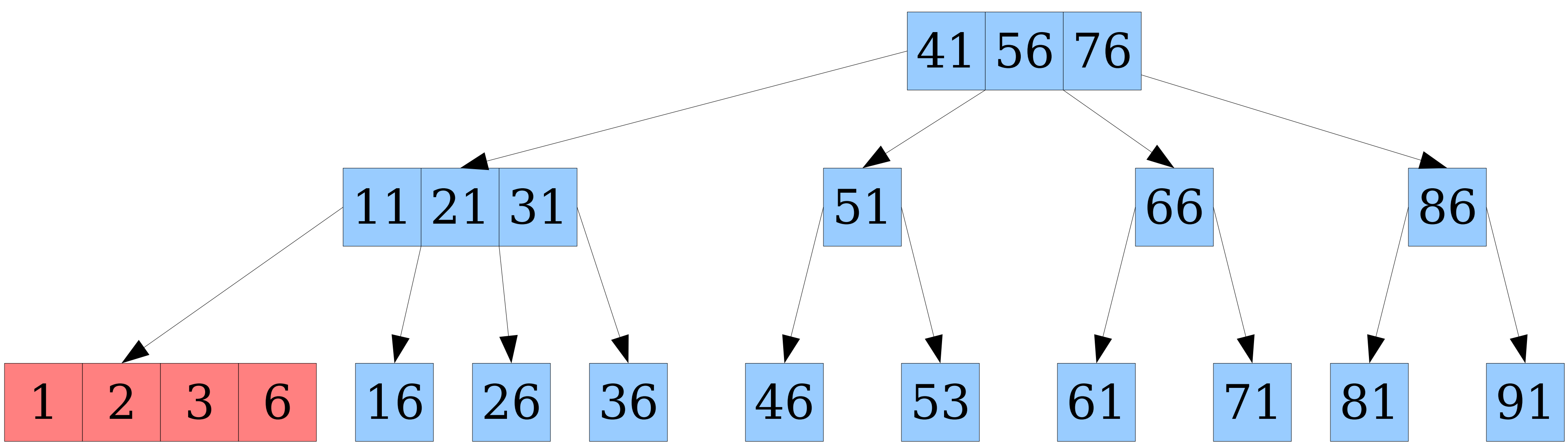
A Potential Issue



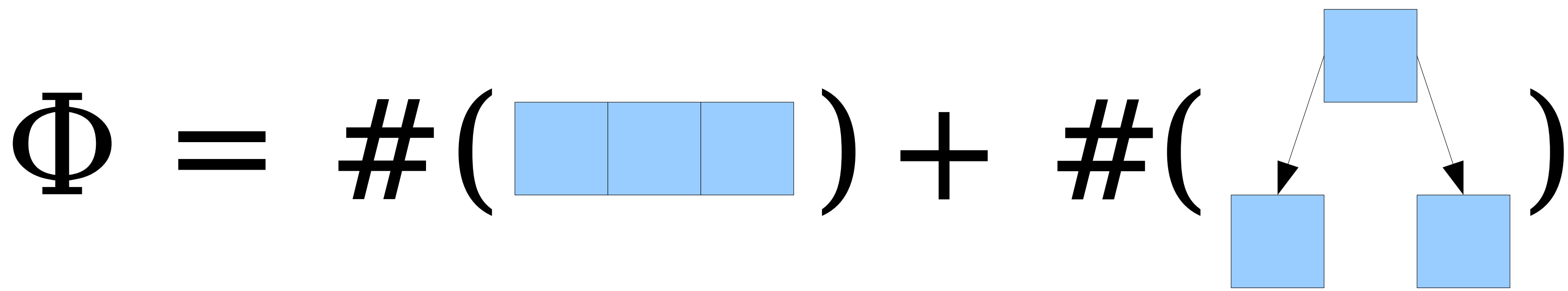


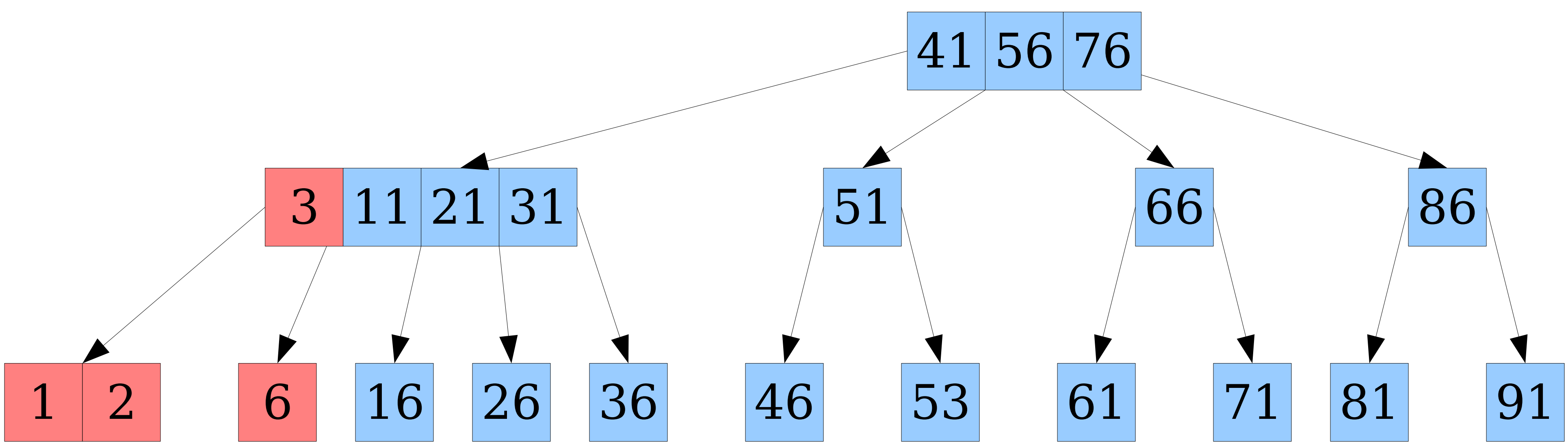
A Potential Issue



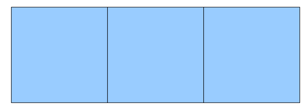
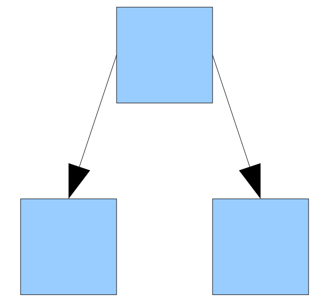


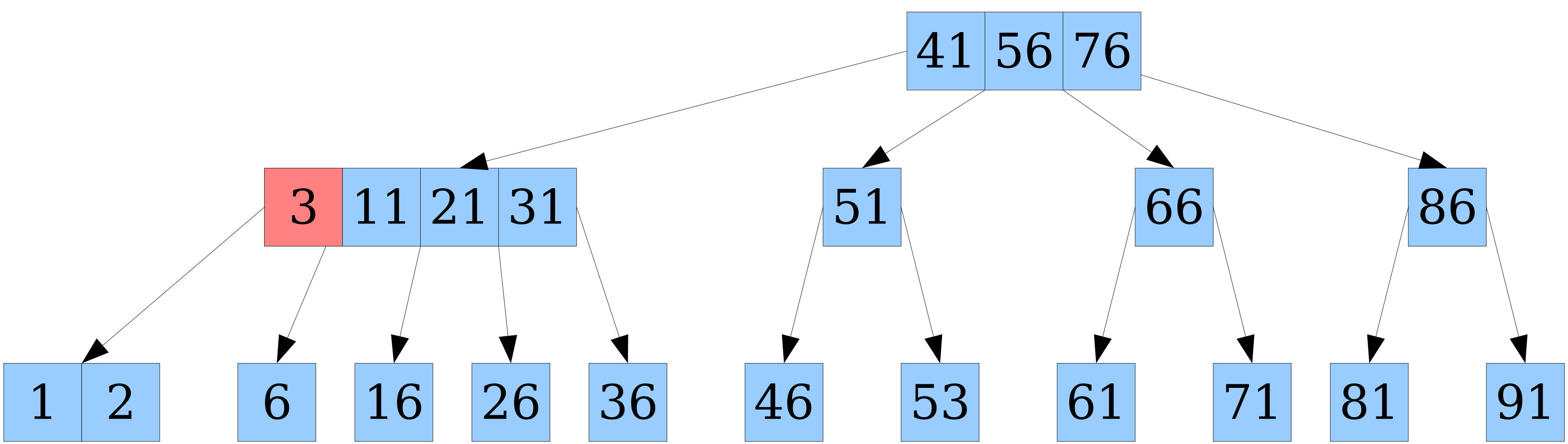
A Potential Issue



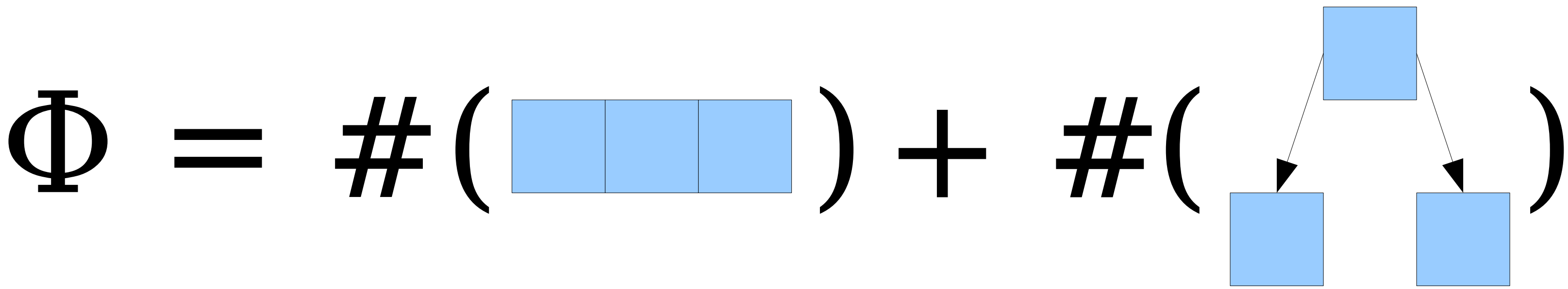


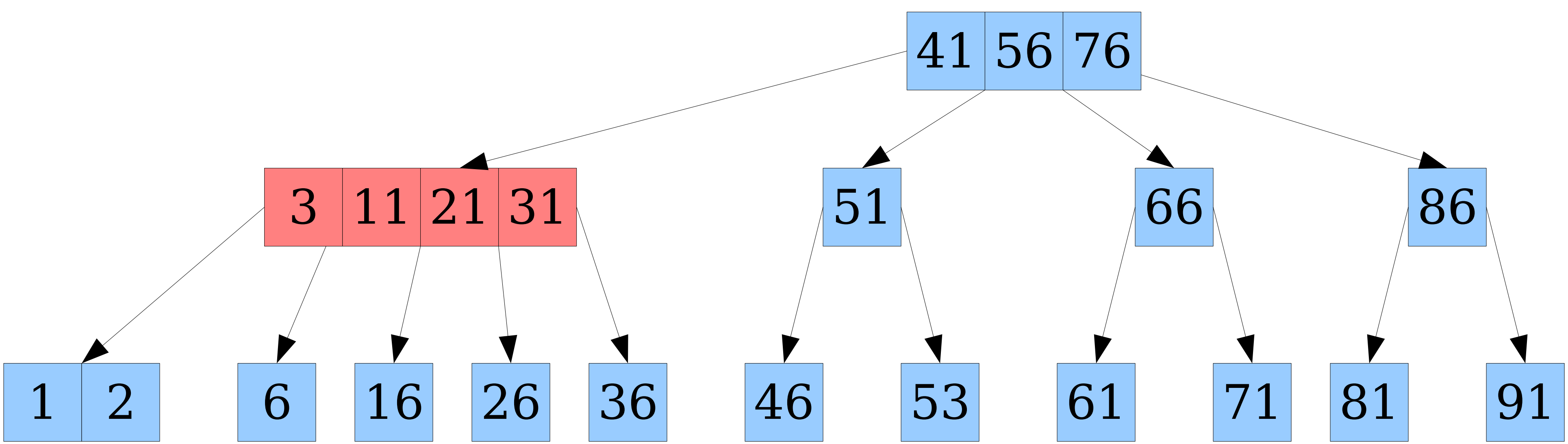
A Potential Issue

$\Phi = \#($  $) + \#($  $)$

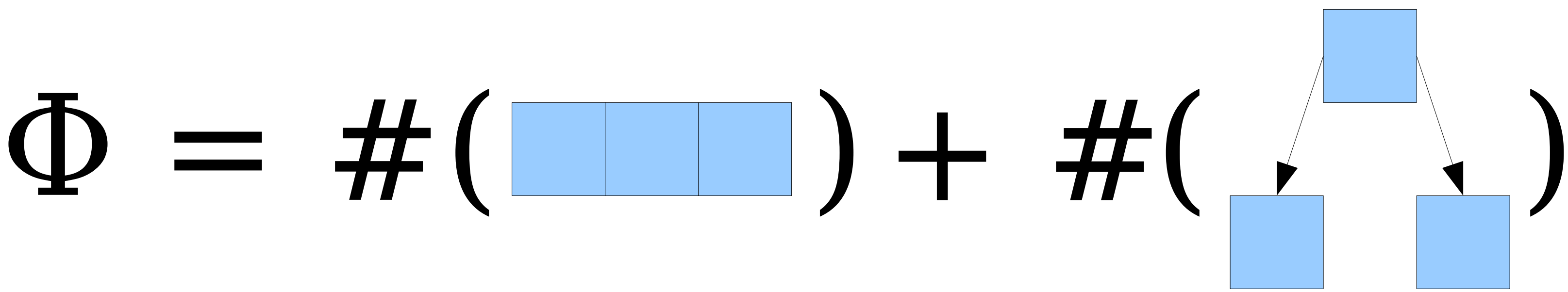


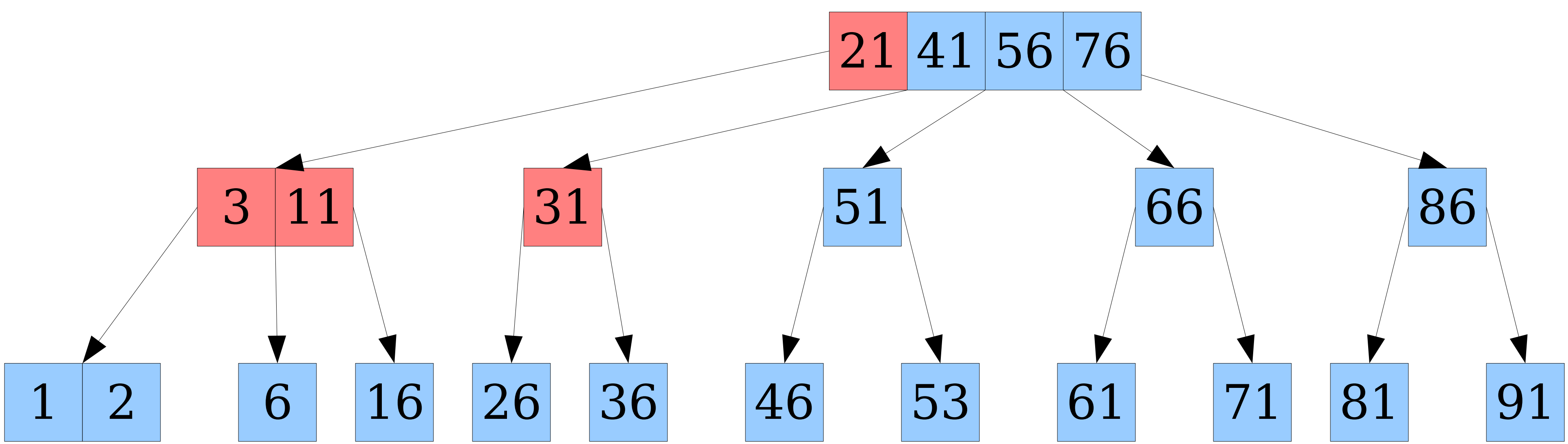
A Potential Issue



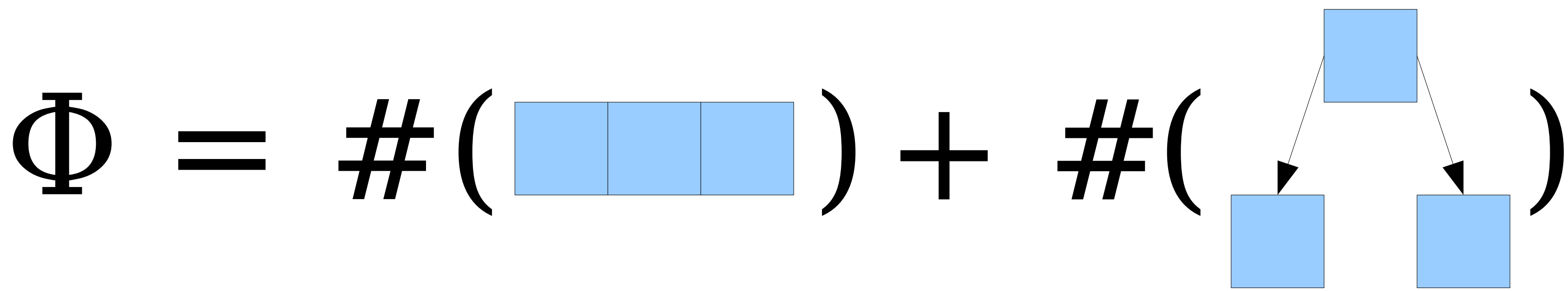


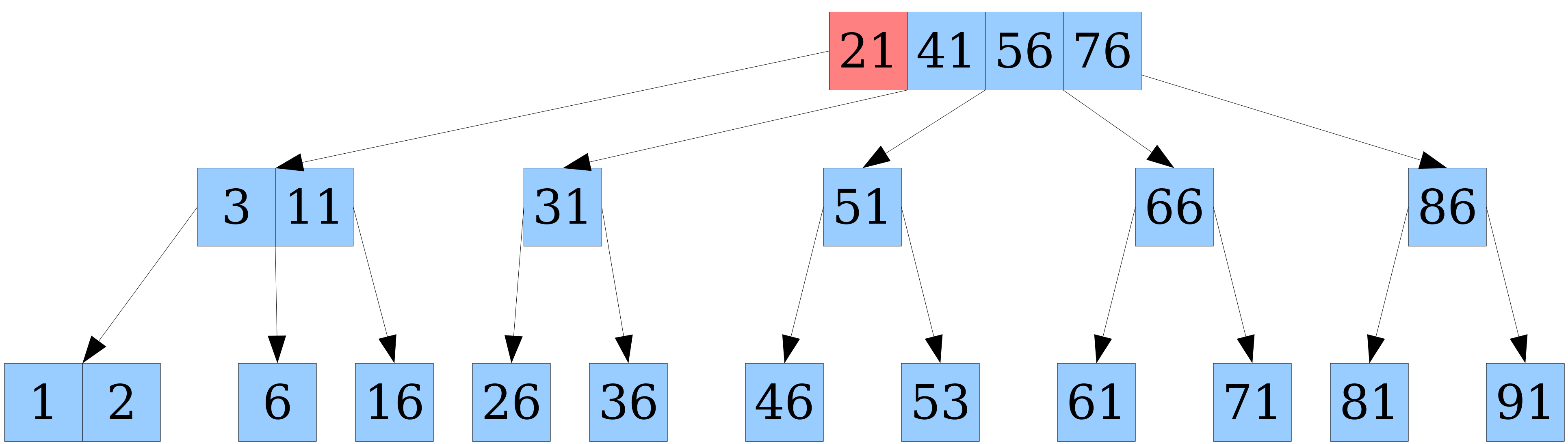
A Potential Issue



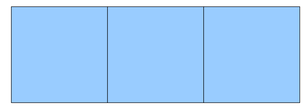
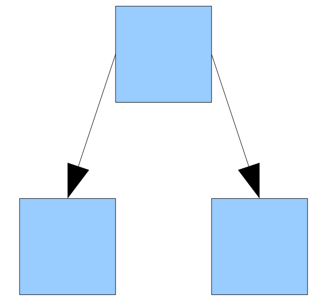


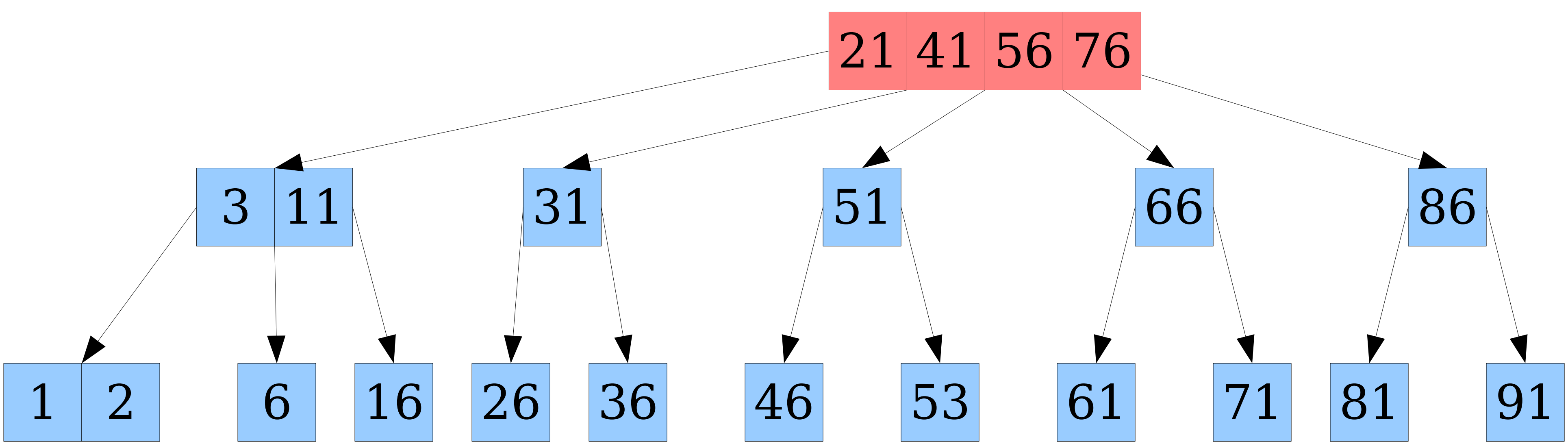
A Potential Issue



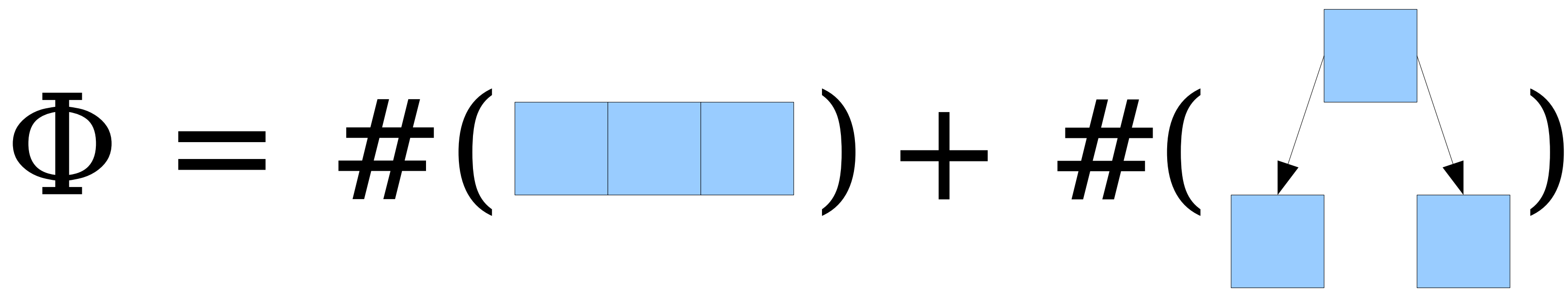


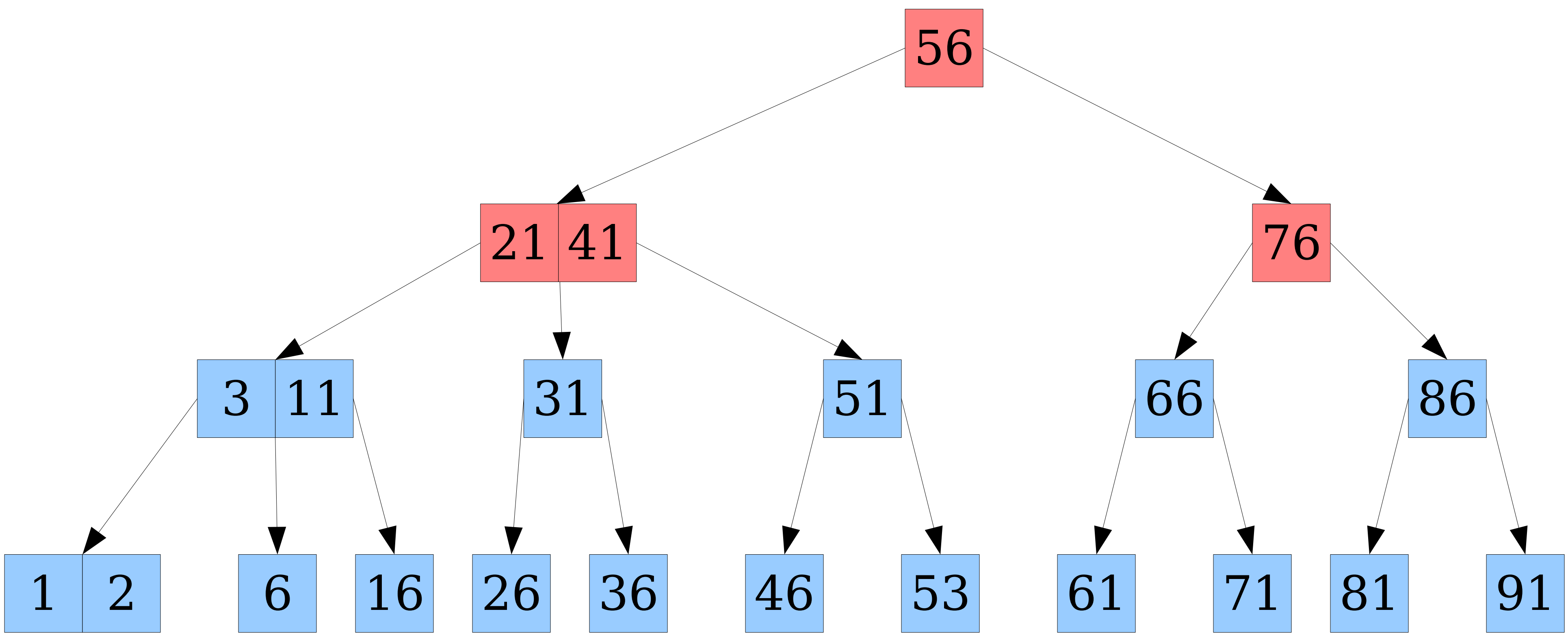
A Potential Issue

$\Phi = \#($  $) + \#($  $)$

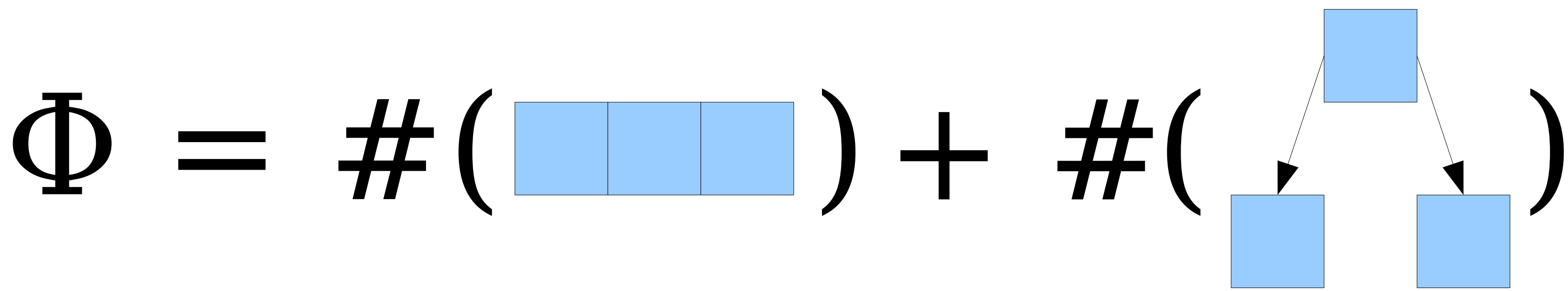


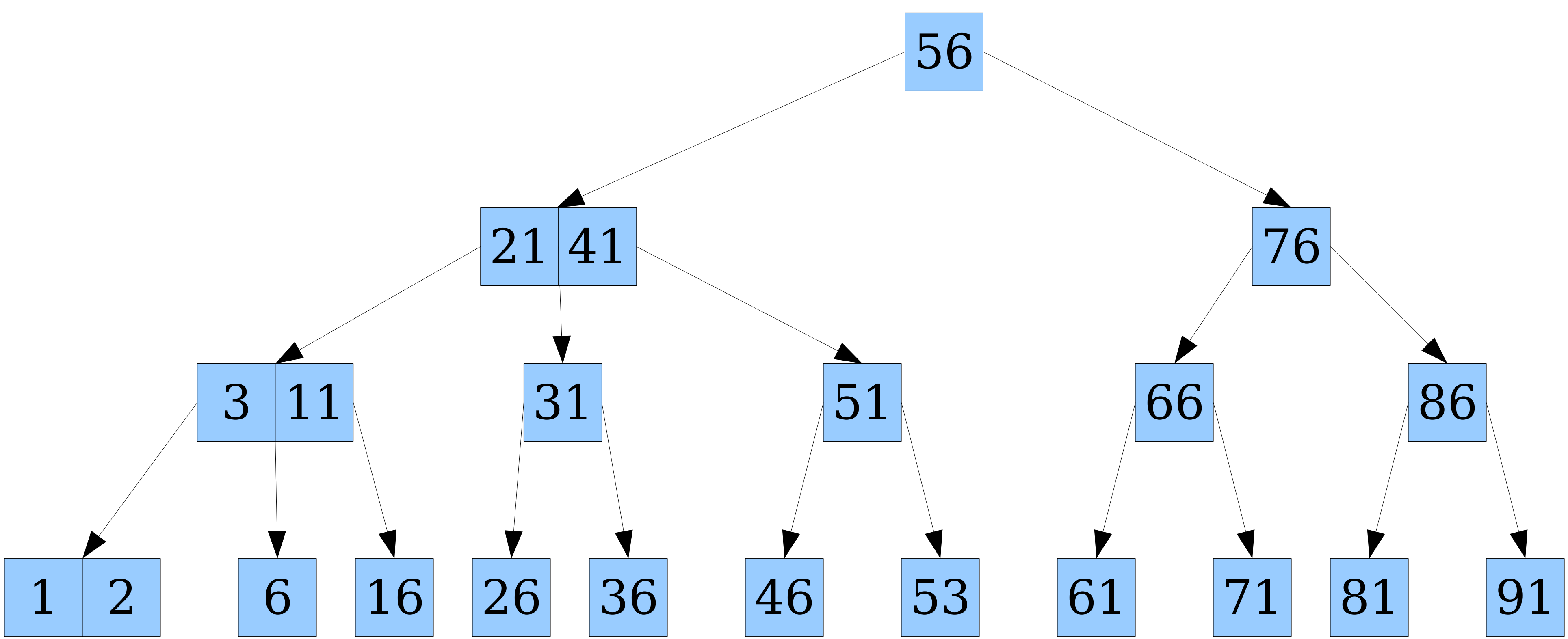
A Potential Issue



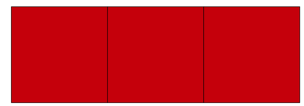
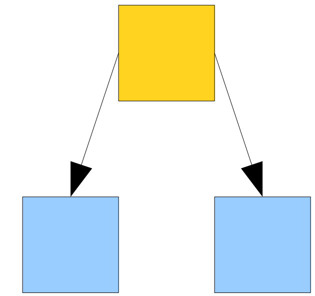


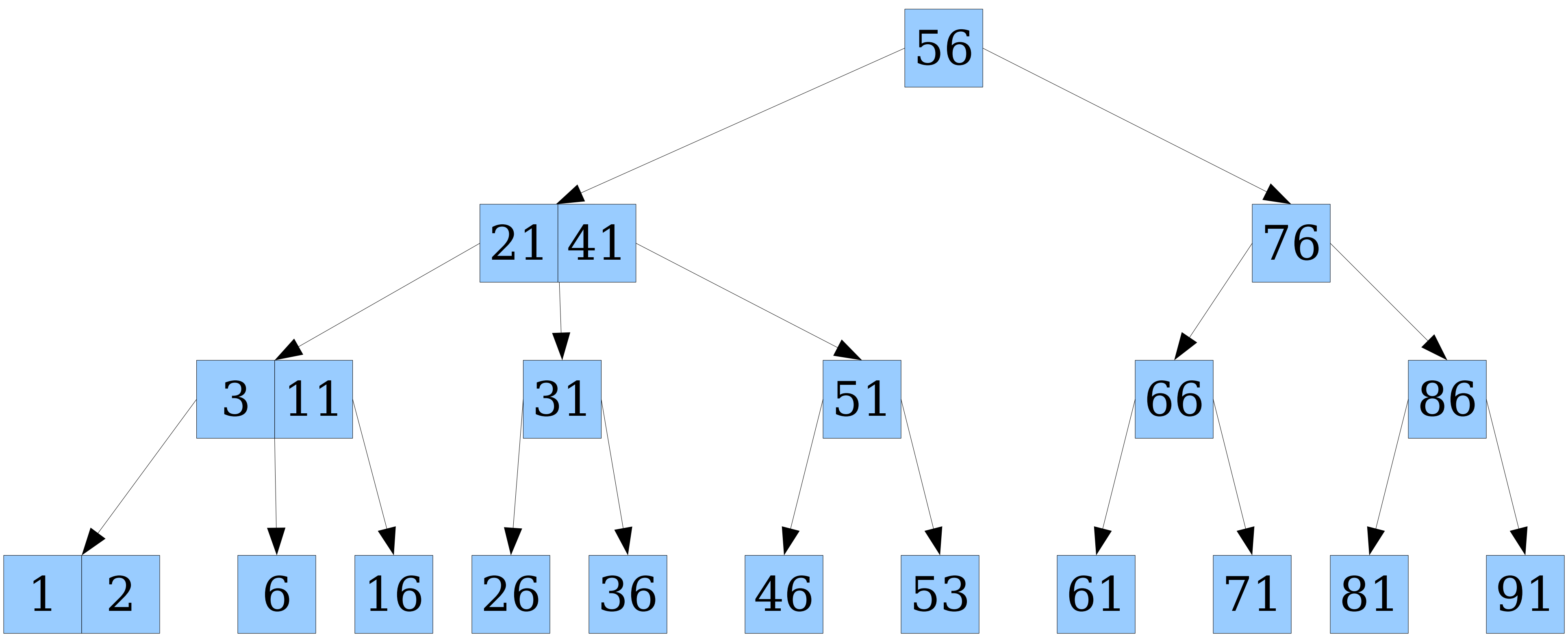
A Potential Issue



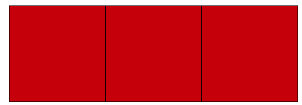
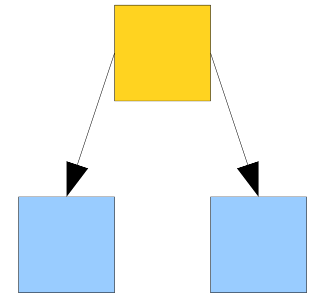


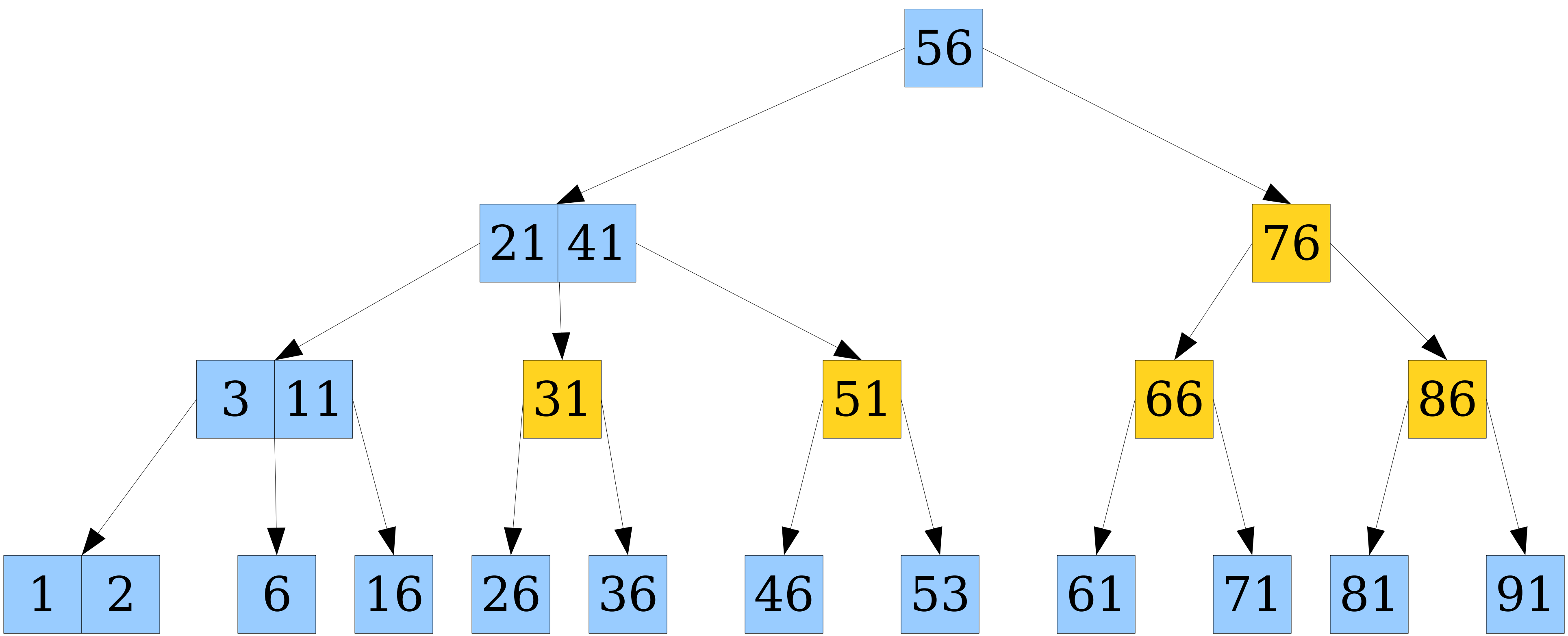
A Potential Issue

$\Phi = \#($  $) + \#($  $)$

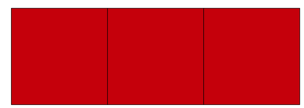
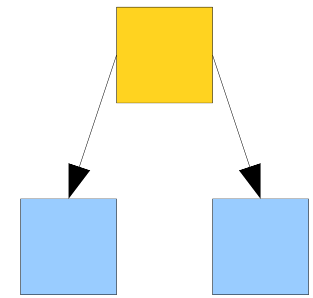


A Potential Issue

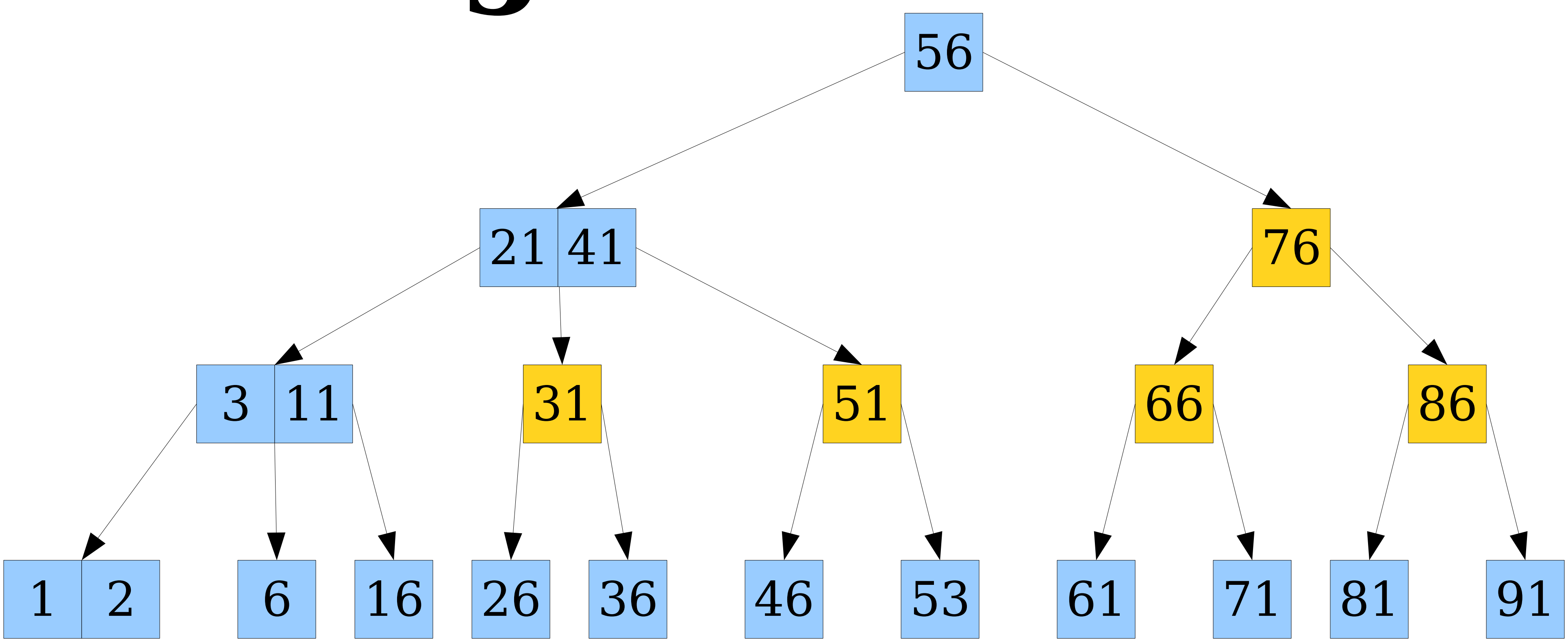
$\Phi = \#($  $) + \#($  $)$



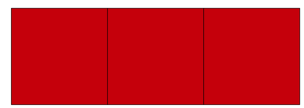
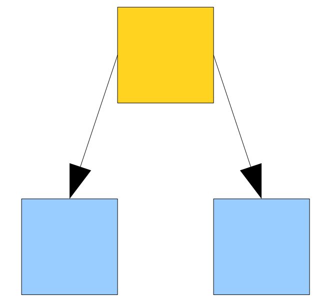
A Potential Issue

Φ = #() + #()

= **5**

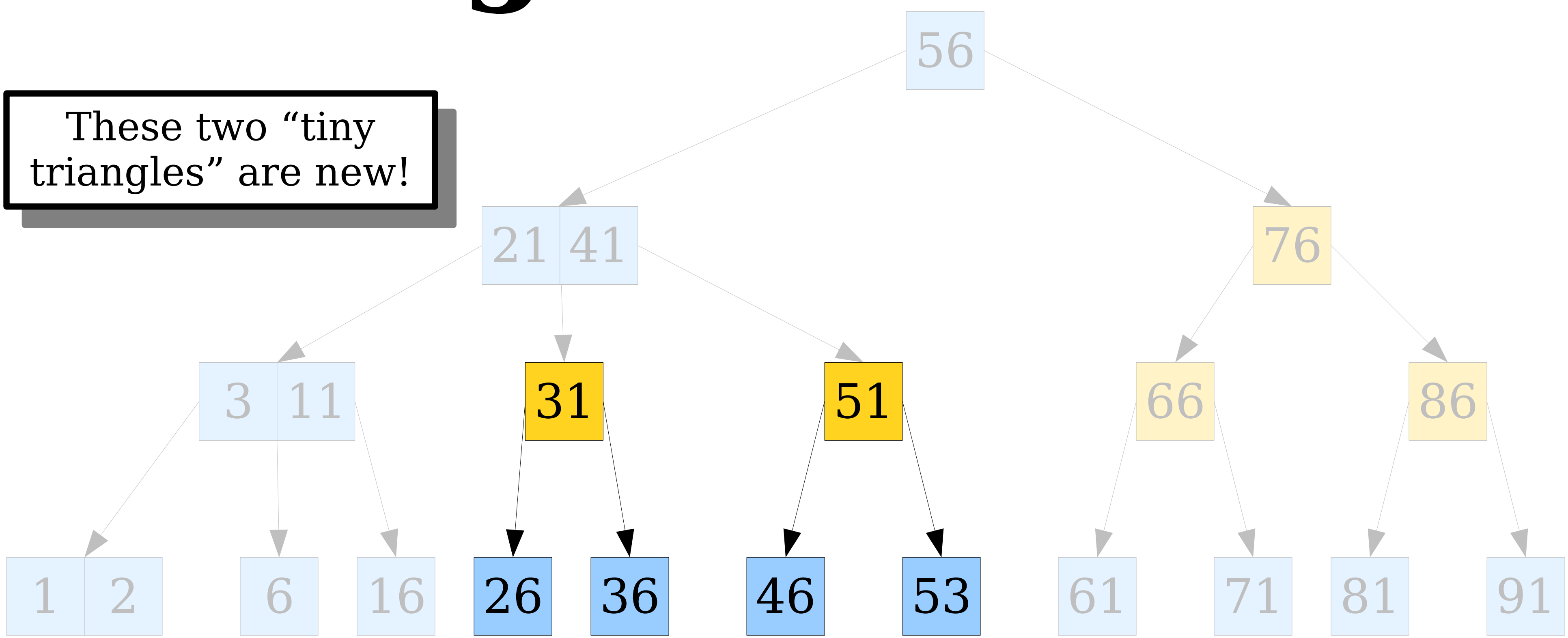


A Potential Issue

Φ = #() + #()

= **5**

These two "tiny triangles" are new!

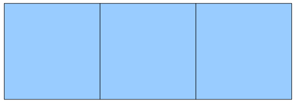
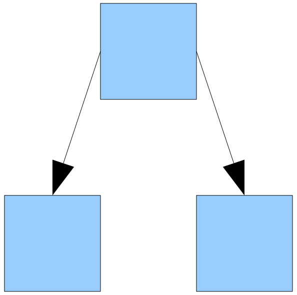


A Problem

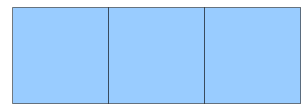
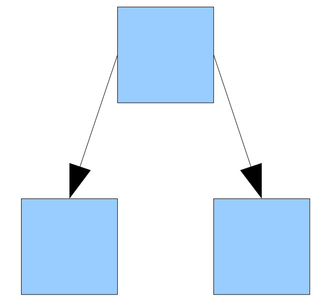
- When doing a “heavy” insertion that splits multiple 4-nodes, the resulting nodes might produce new “tiny triangles.”
- ***Symptom:*** Our potential doesn't drop nearly as much as it should, so we can't pay for future operations. Amortized cost of the operation works out to $\Theta(\log n)$, not $O(1)$ as we hoped.
- ***Root Cause:*** Splitting a 4-node into a 2-node and a 3-node might introduce new “tiny triangles,” which in turn might cause future deletes to become more expensive.

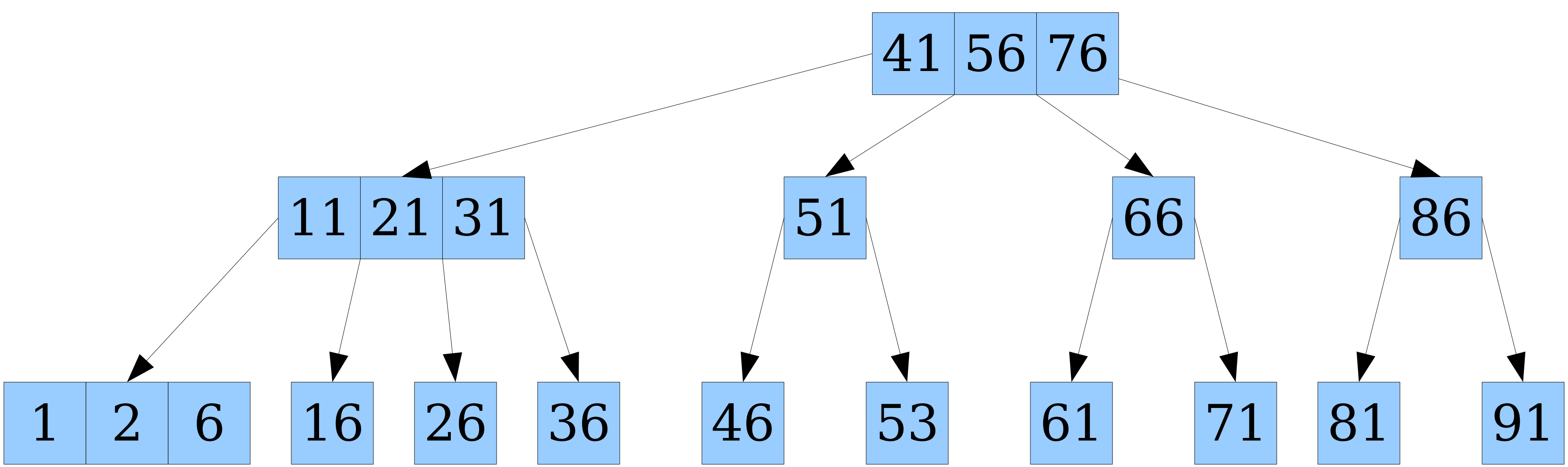
The Solution

- 4-nodes are troublesome for two separate reasons:
  - They cause chained splits in an insertion.
  - After an insertion, they might split and produce a tiny triangle.
- ***Idea:*** Charge each 4-node for two different costs: the cost of an expensive insertion, plus the (possible) future cost of doing an expensive deletion.

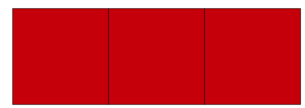
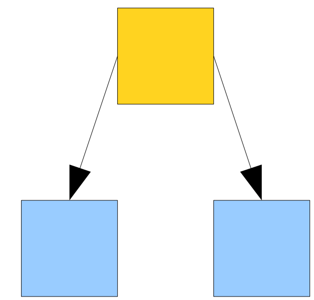
$\Phi = 2\#($  $) + \#($  $)$

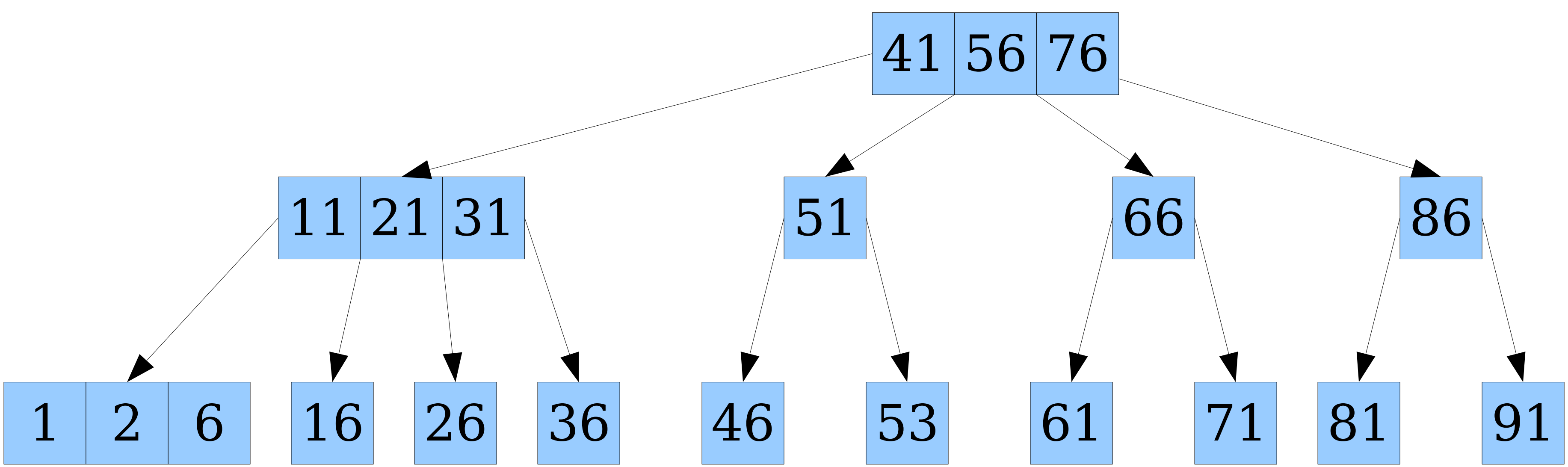
Unlocking our Potential

$\Phi = 2\#($  $) + \#($  $)$

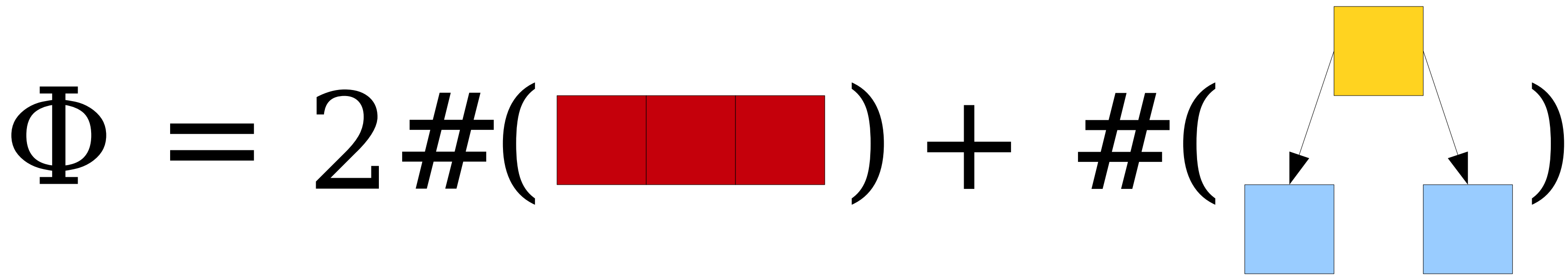


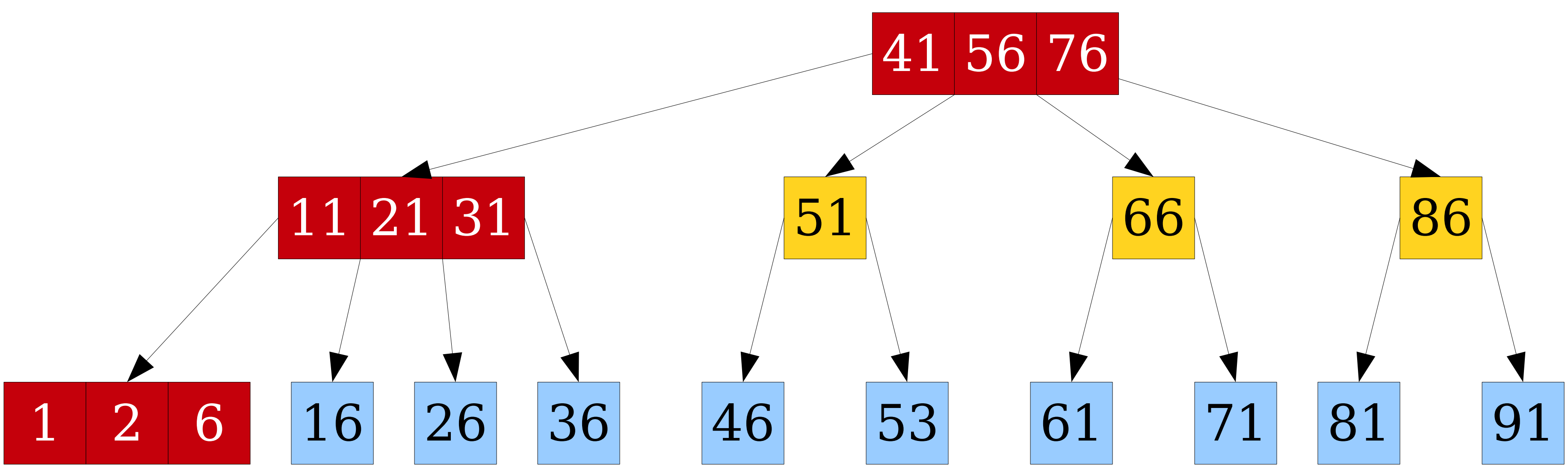
Unlocking our Potential

$\Phi = 2\#($  $) + \#($  $)$

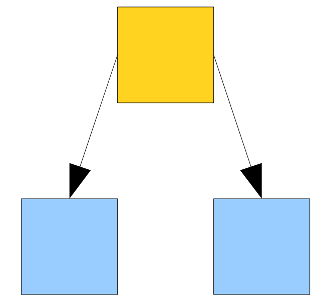


Unlocking our Potential

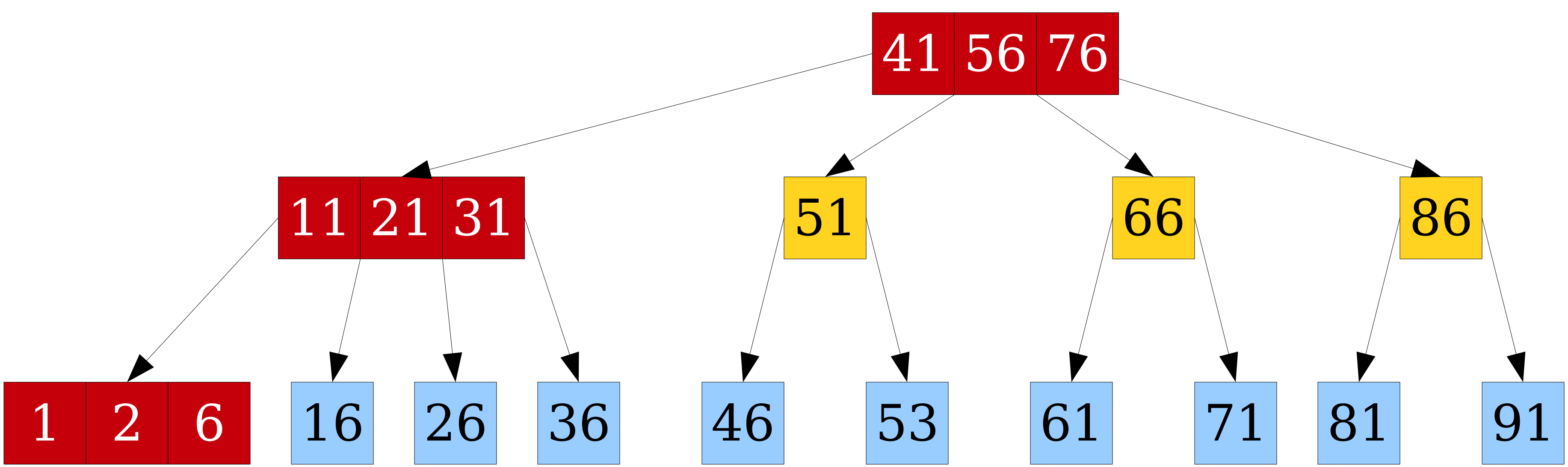




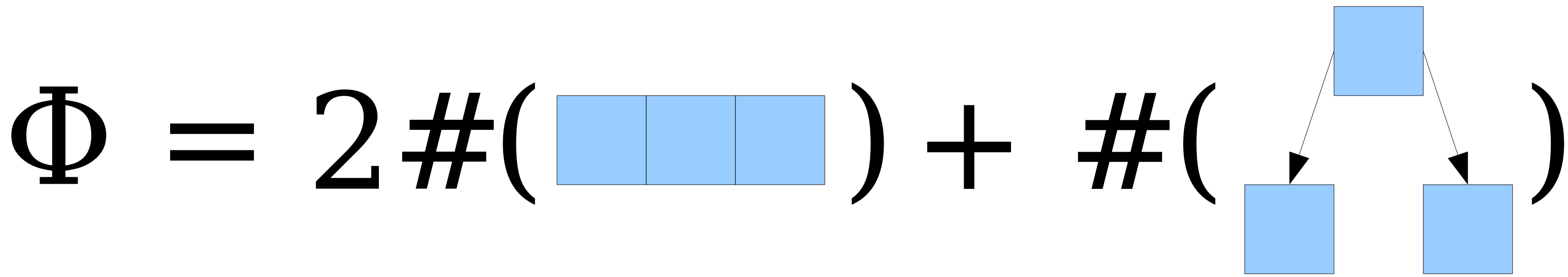
Unlocking our Potential

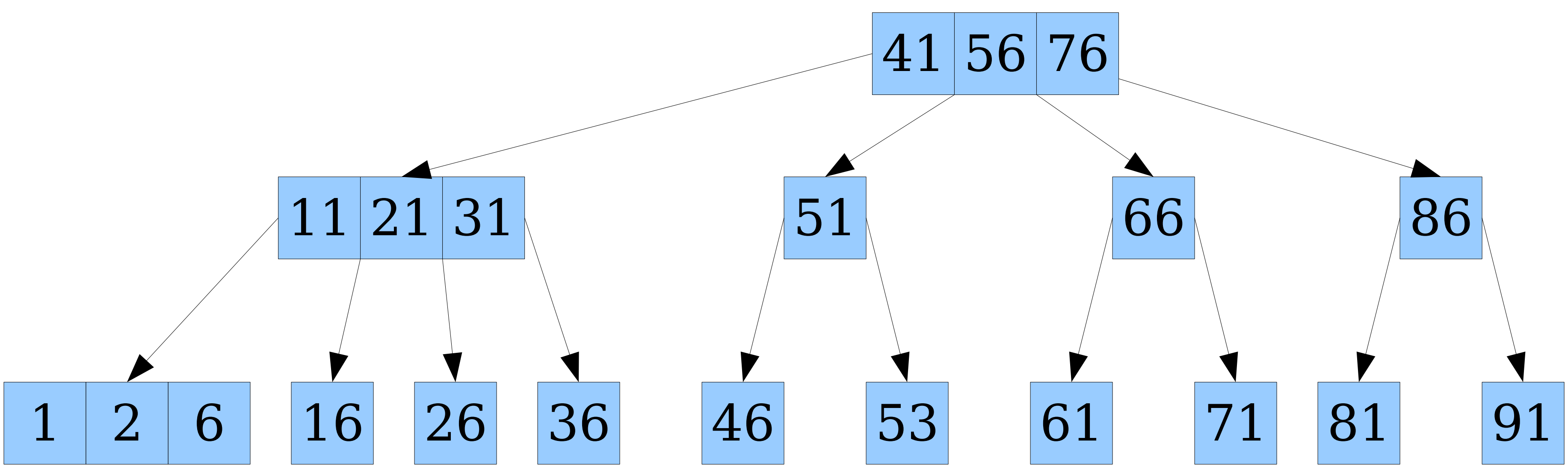
$\Phi = 2\#($  $) + \#($  $)$

$= \mathbf{9}$

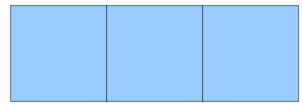
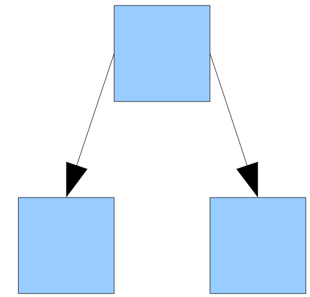


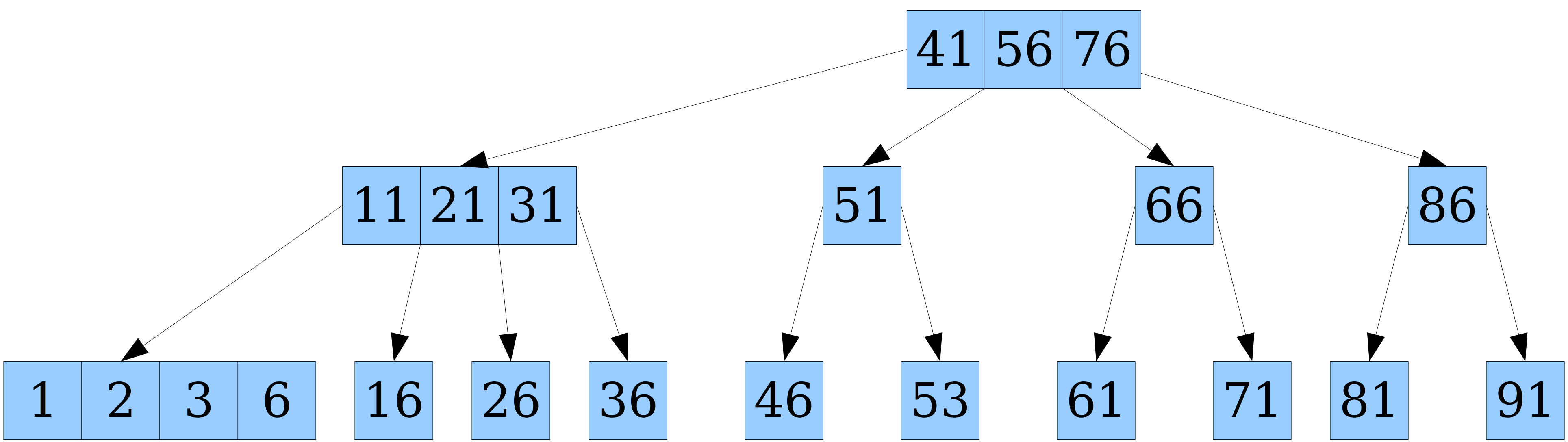
Unlocking our Potential



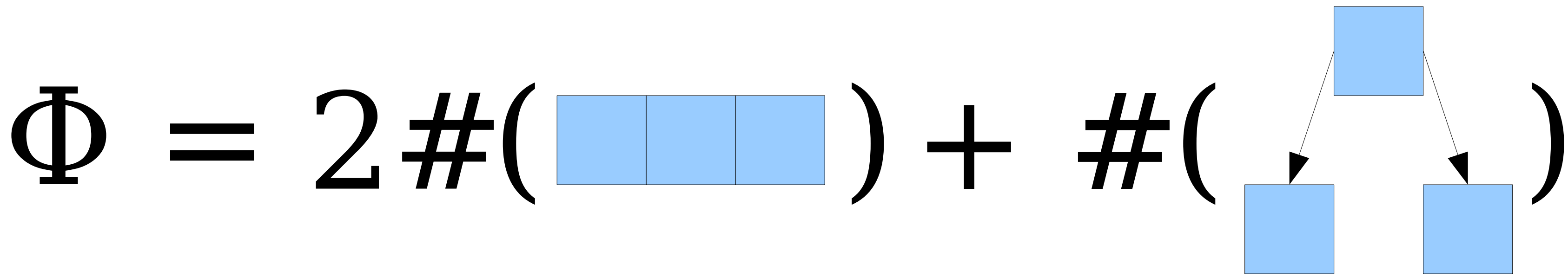


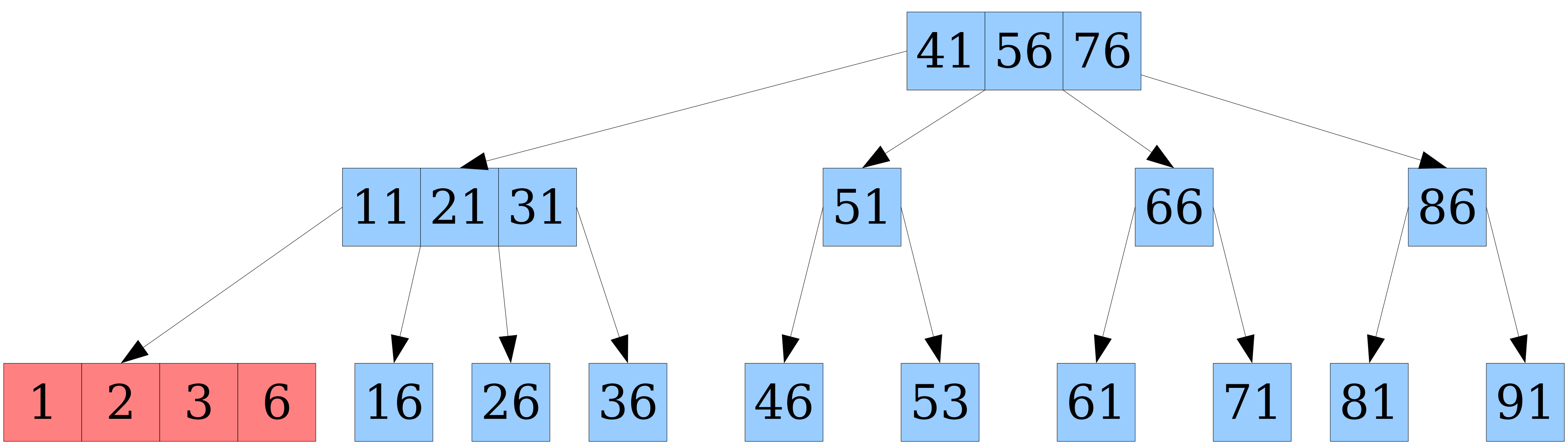
Unlocking our Potential

$\Phi = 2\#($  $) + \#($  $)$

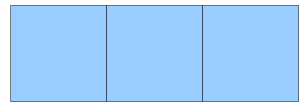
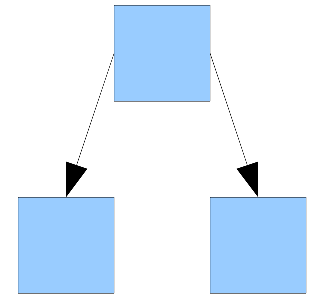


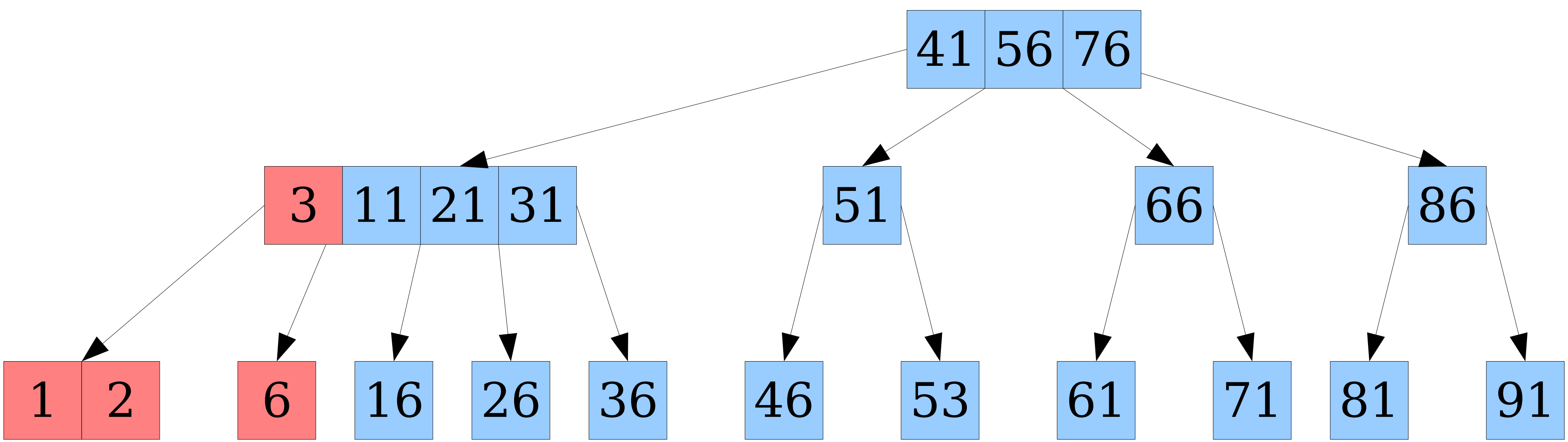
Unlocking our Potential



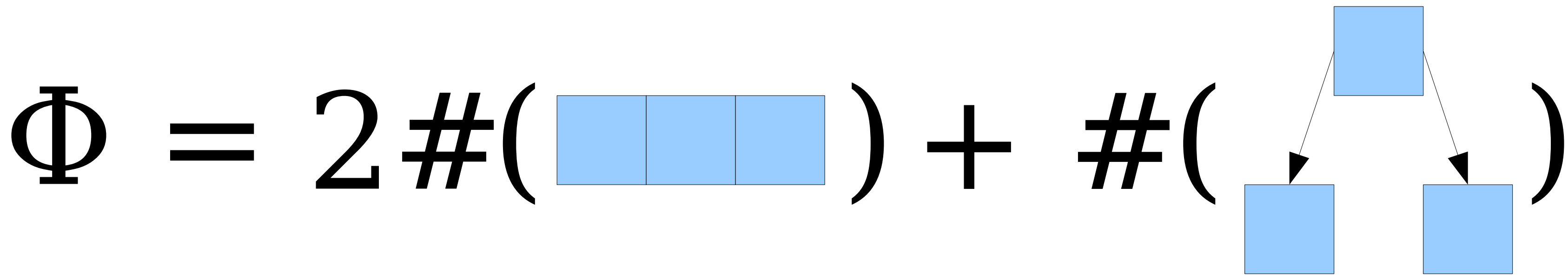


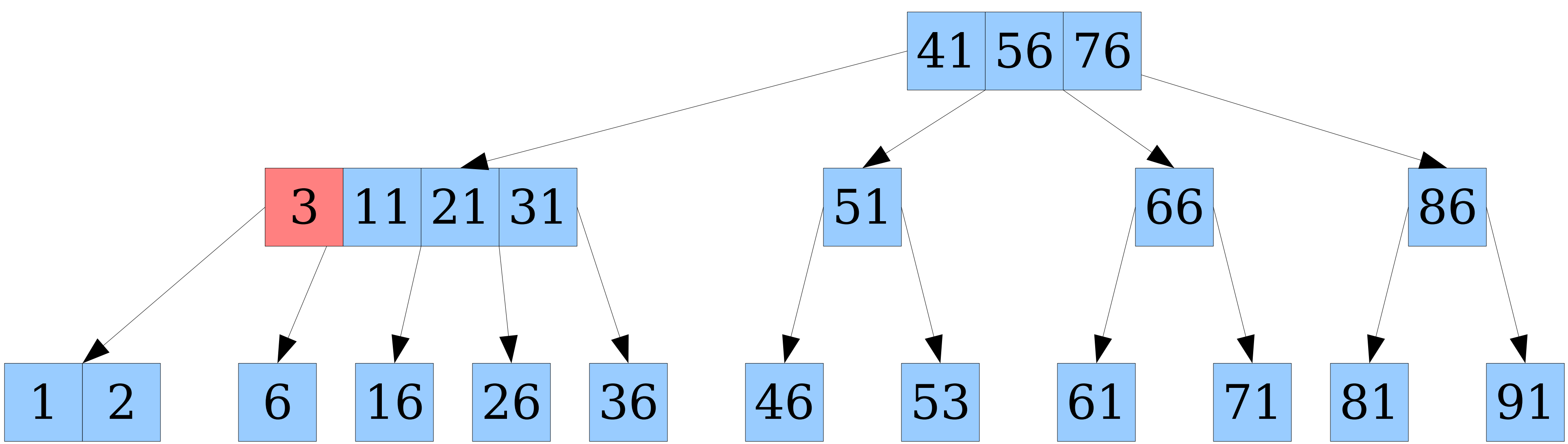
Unlocking our Potential

$\Phi = 2\#($  $) + \#($  $)$

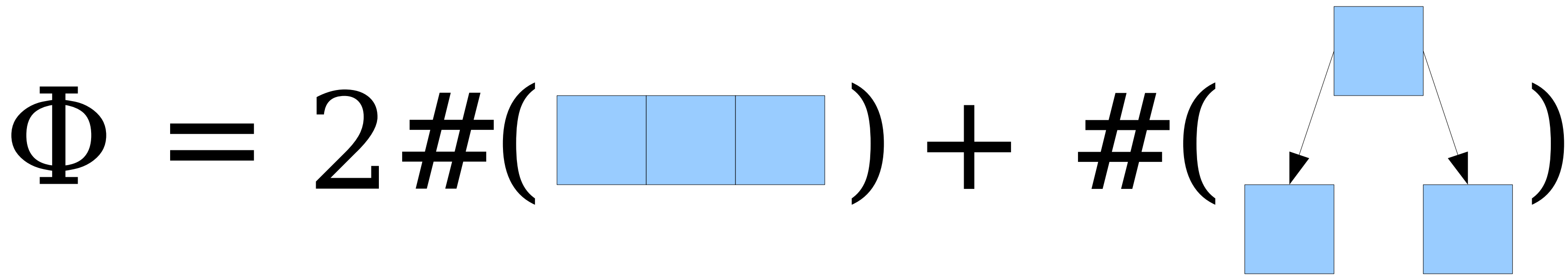


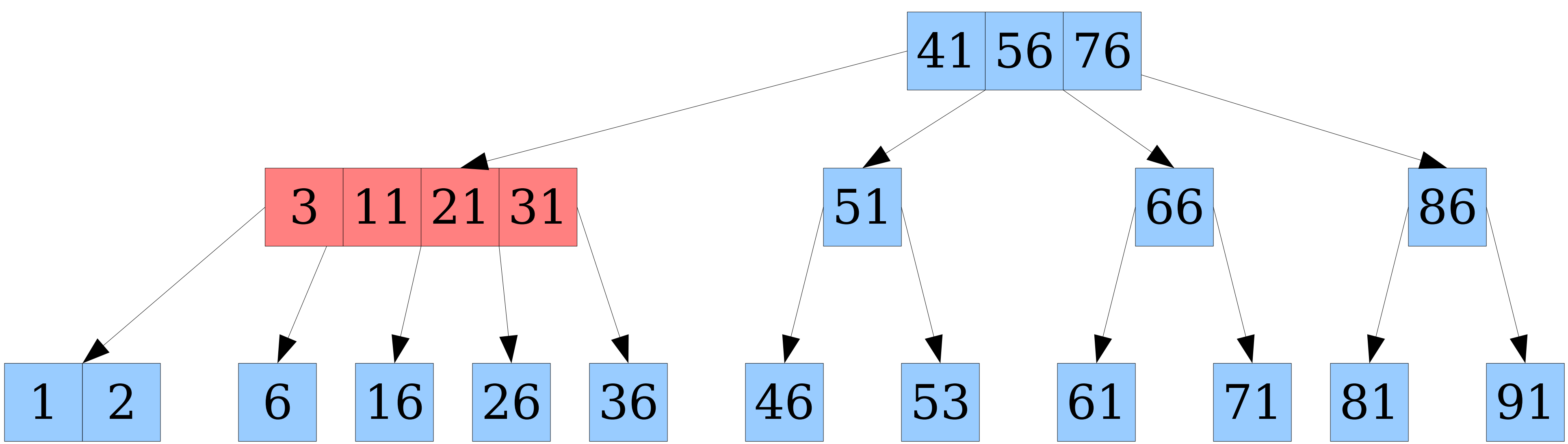
Unlocking our Potential



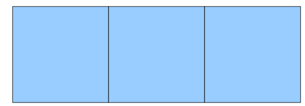
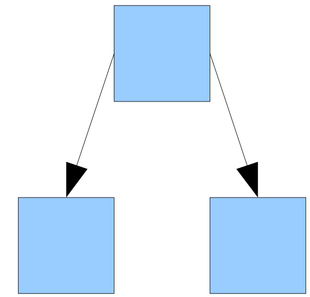


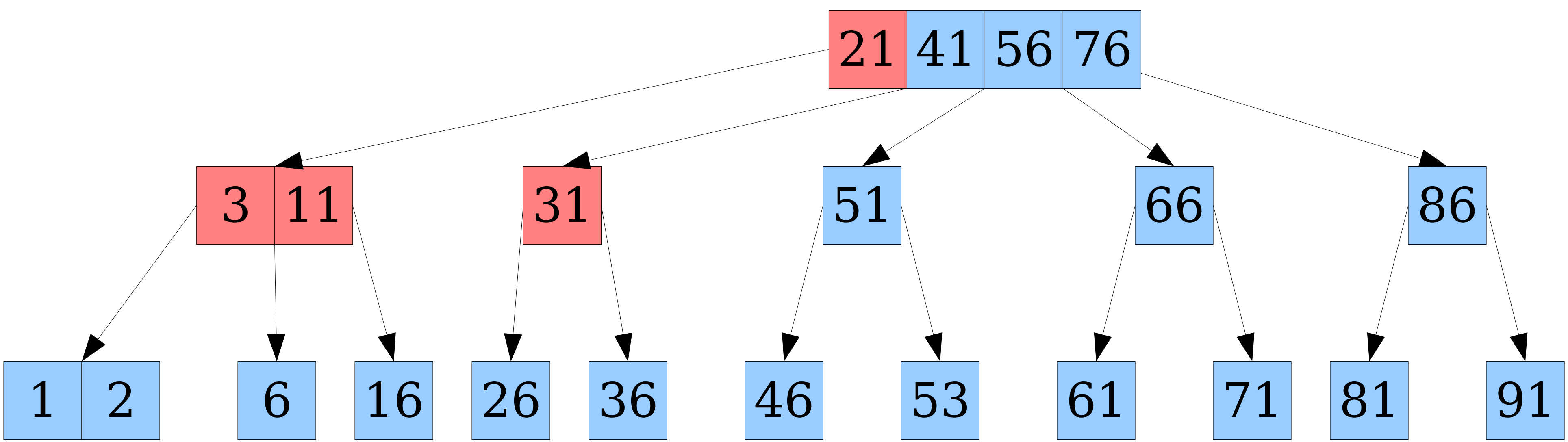
Unlocking our Potential



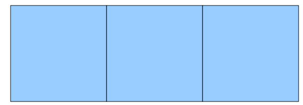
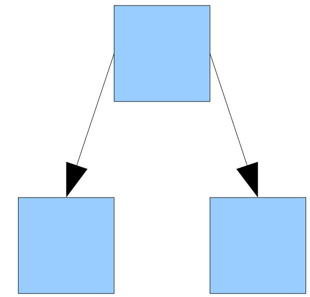


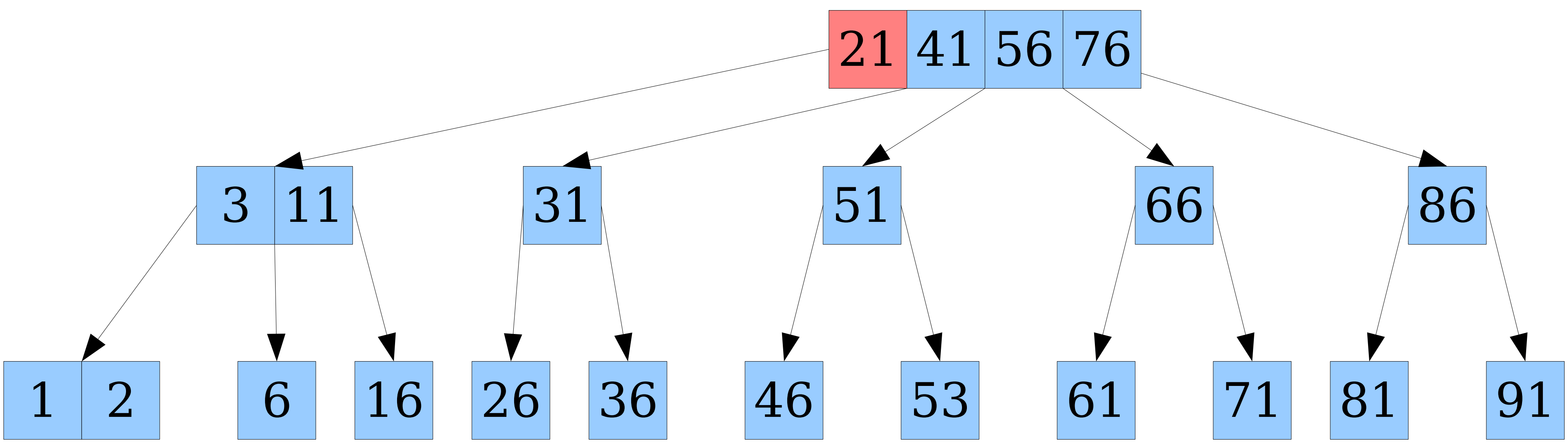
Unlocking our Potential

$\Phi = 2\#($  $) + \#($  $)$

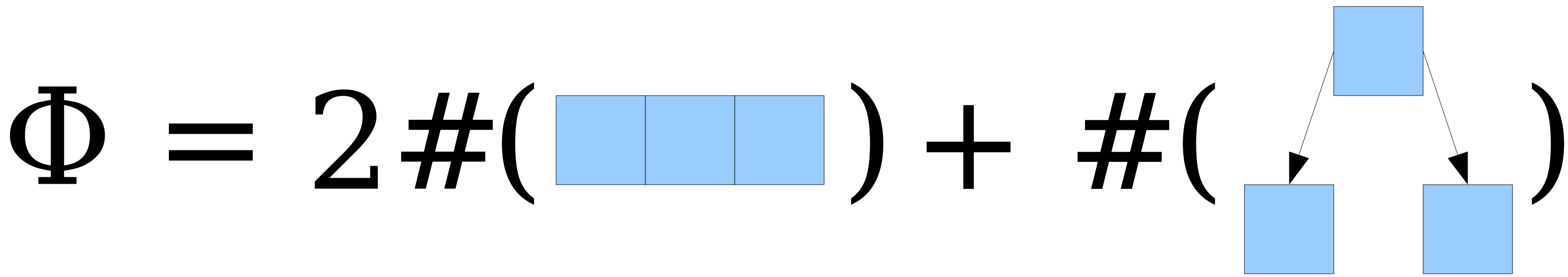


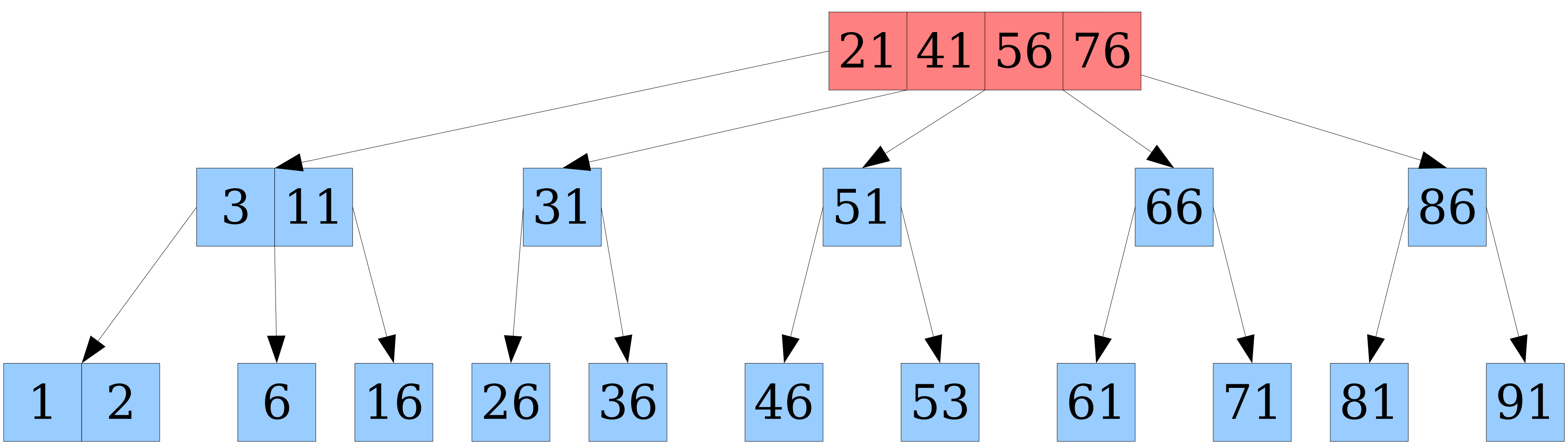
Unlocking our Potential

$\Phi = 2\#($  $) + \#($  $)$

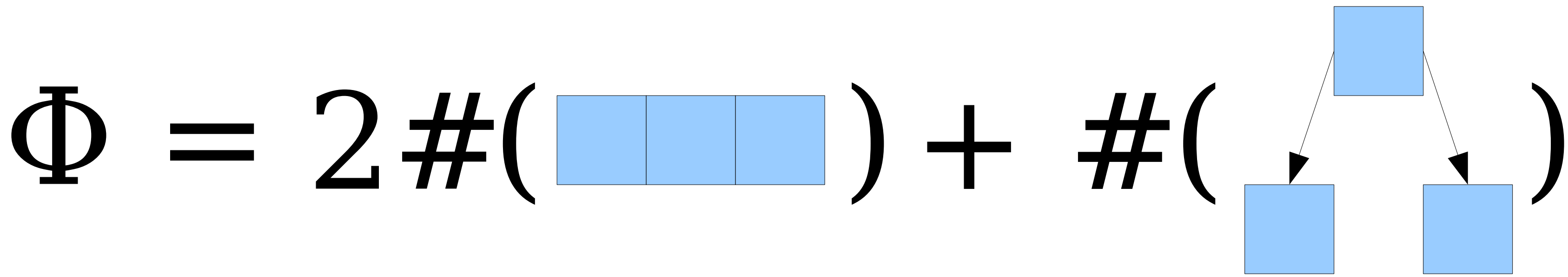


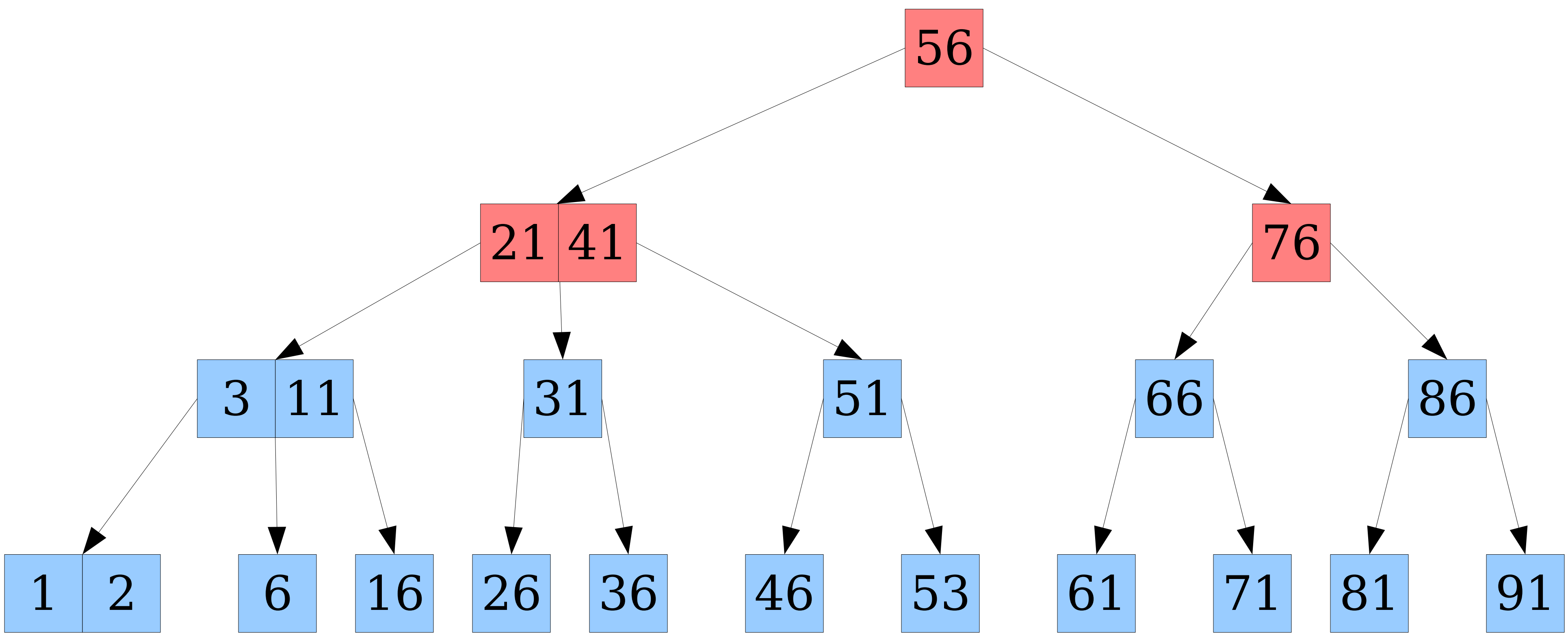
Unlocking our Potential



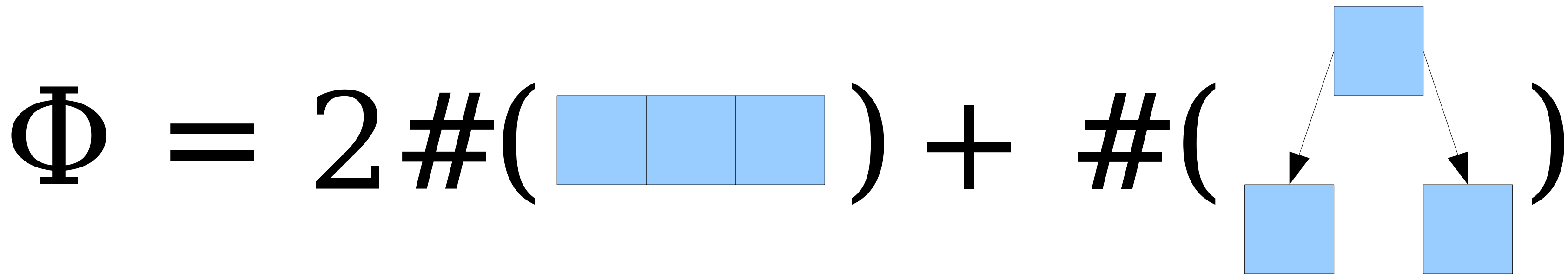


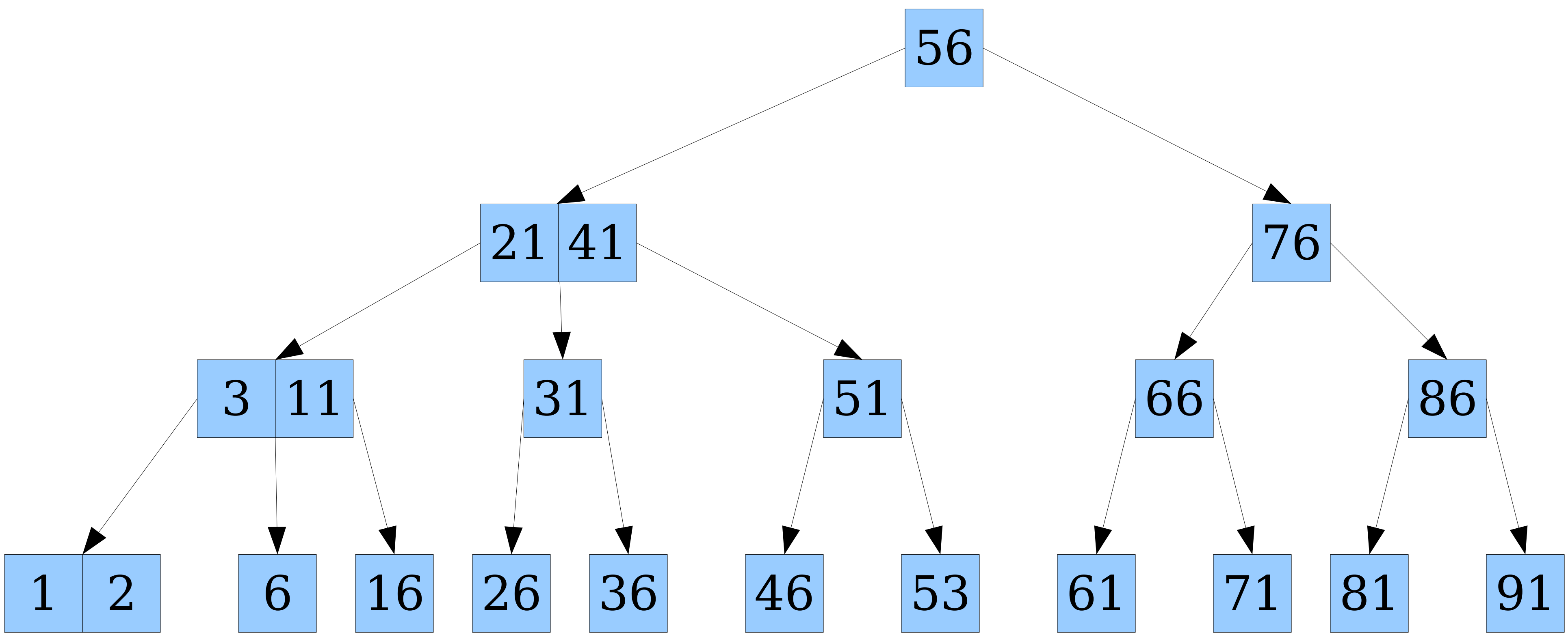
Unlocking our Potential



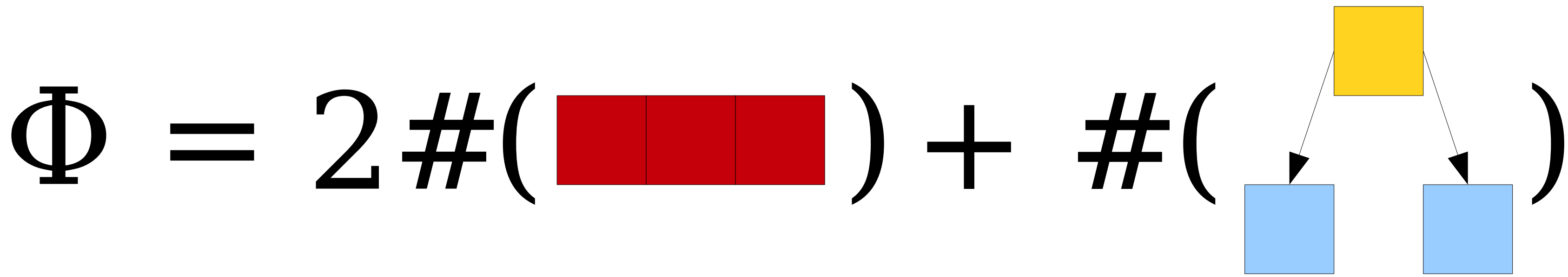


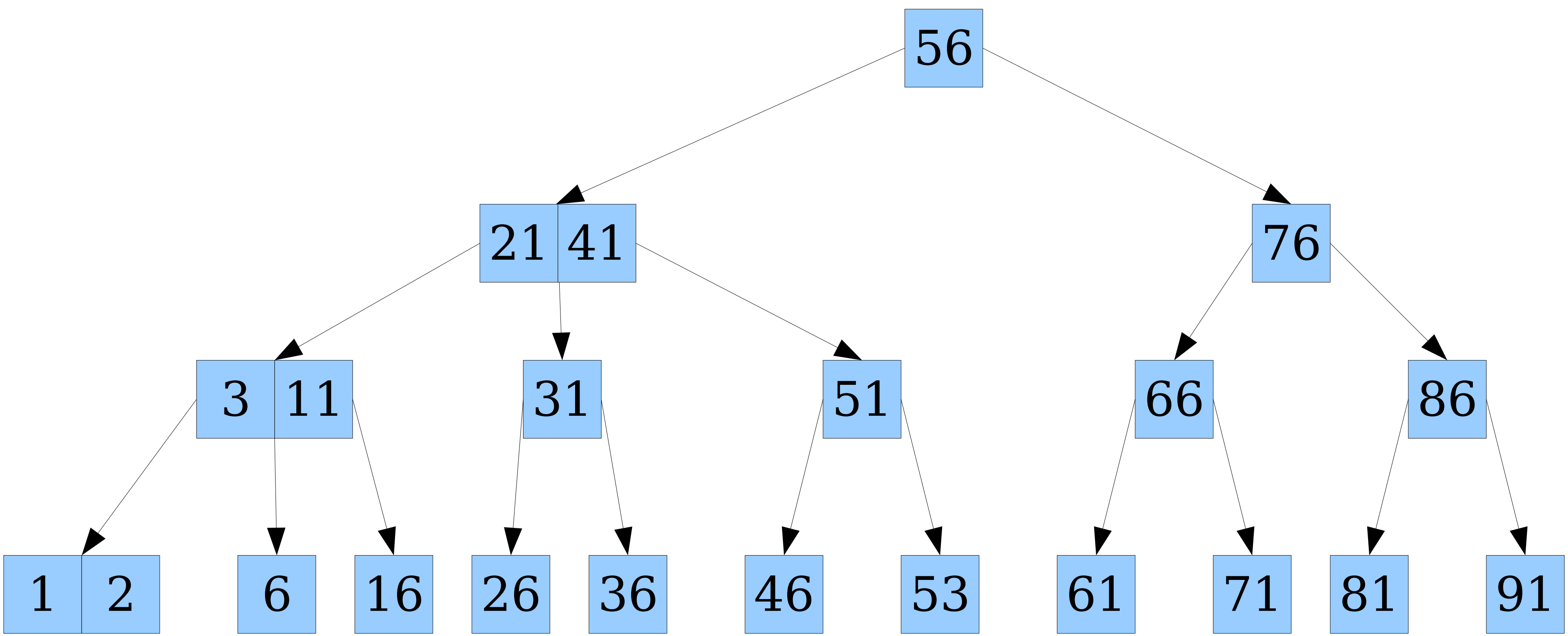
Unlocking our Potential



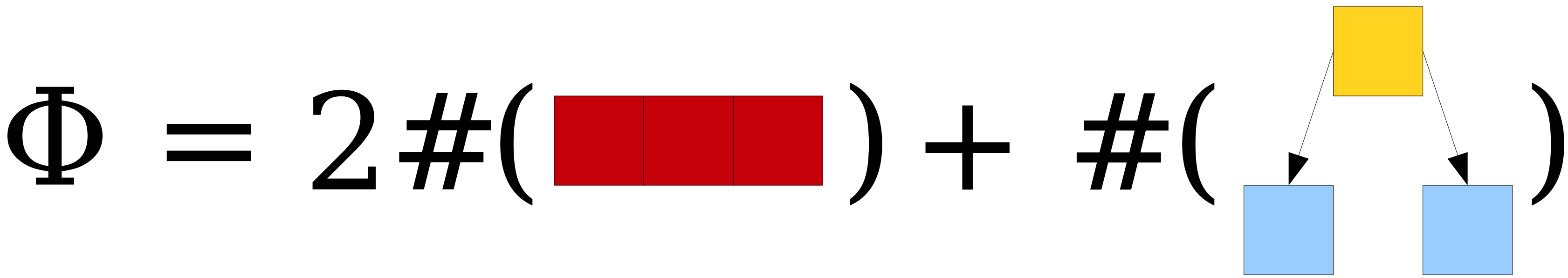


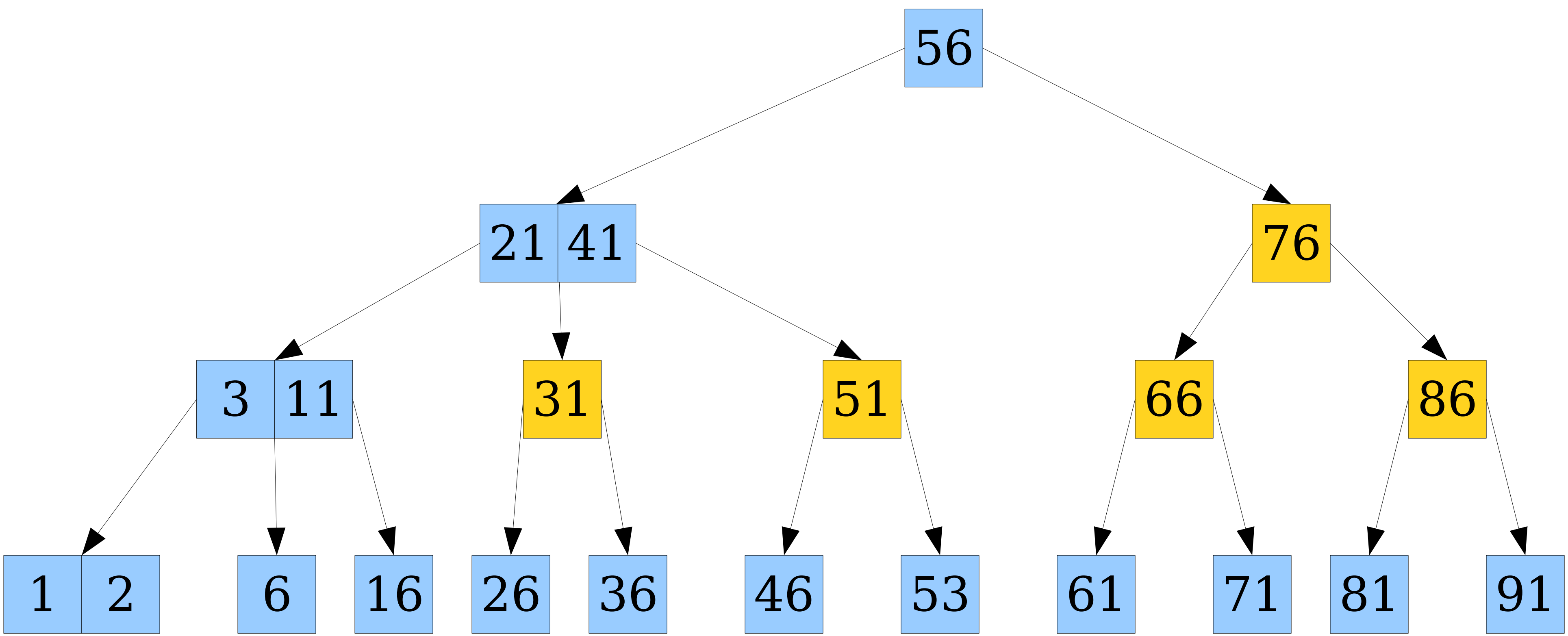
Unlocking our Potential



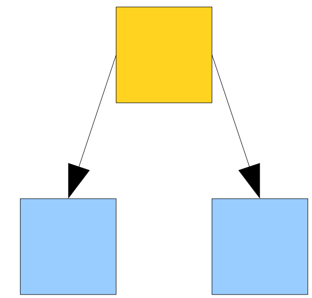


Unlocking our Potential

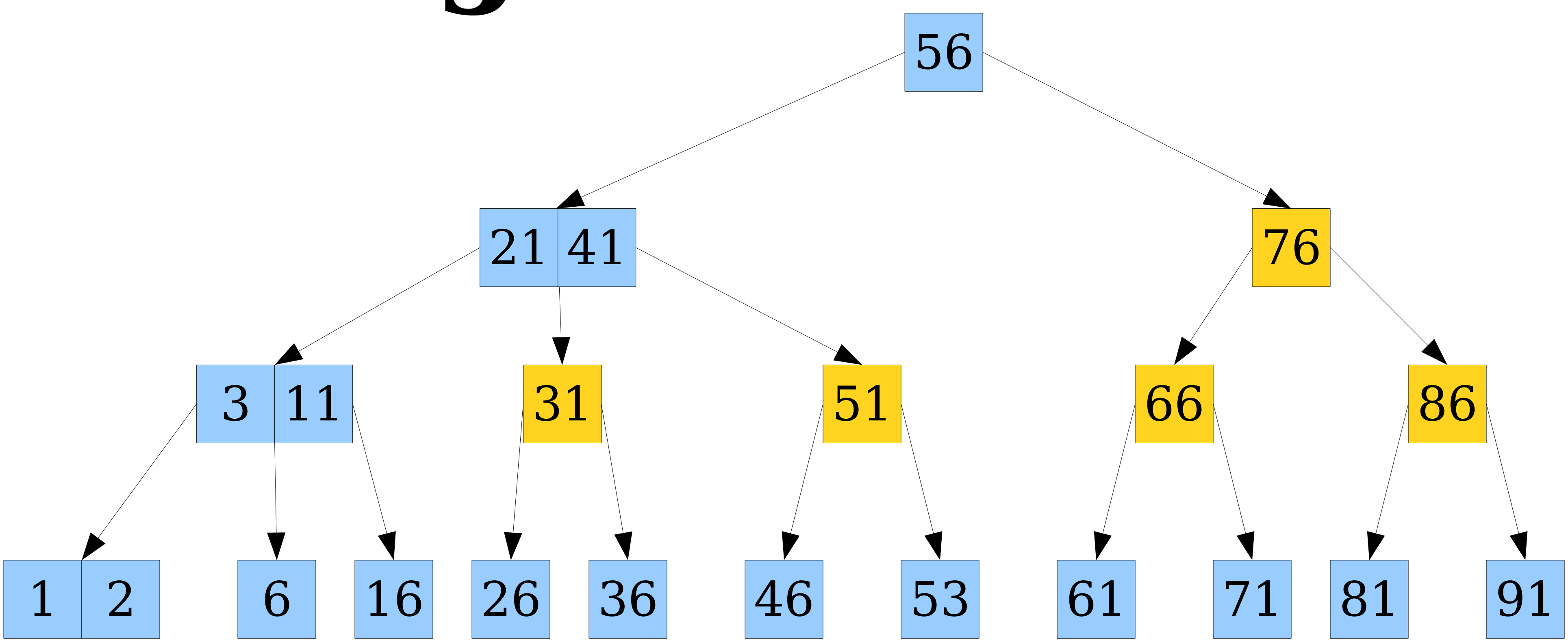




Unlocking our Potential

$\Phi = 2\#($  $) + \#($  $)$

$= \mathbf{5}$



The Solution

- This new potential function ensures that if an insertion chains up k levels, the potential drop is at least k (and possibly up to $2k$).
- Therefore, the amortized fixup work for an insertion is O(1).
- Using the same argument as before, deletions require amortized O(1) fixups.

Why This Matters

- Via the isometry, red/black trees have O(1) amortized fixup per insertion or deletion.
- In practice, this makes red/black trees much faster than other balanced trees on insertions and deletions, even though other balanced trees can be better balanced.

More to Explore

- A ***finger tree*** is a variation on a B-tree in which certain nodes are pointed at by “fingers.” Insertions and deletions are then done only around the fingers.
- Because the only cost of doing an insertion or deletion is the fixup cost, these trees have amortized O(1) insertions and deletions.
- They're often used in purely functional settings to implement queues and deques with excellent runtimes.
- Liked the previous analysis? Consider looking into this for your final project!

Next Time

- ***Binomial Heaps***
  - A simple and versatile heap data structure based on binary arithmetic.
- ***Lazy Binomial Heaps***
  - Rejiggering binomial heaps for fun and profit.