Amortized Analysis

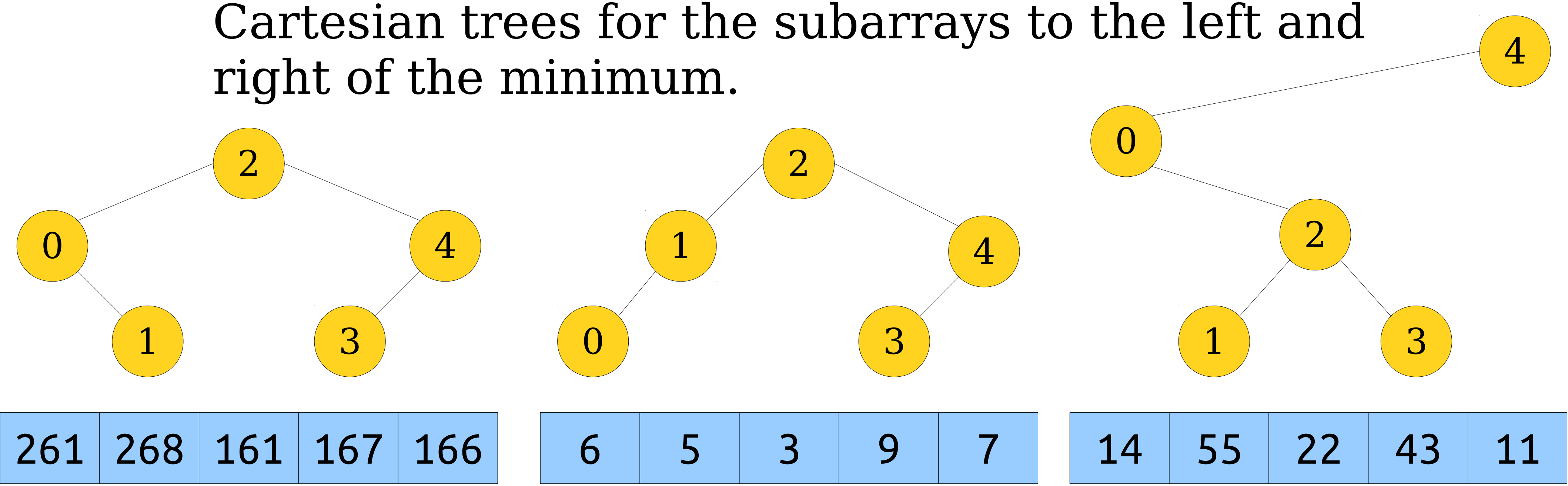
Outline for Today

- ***Cartesian Trees Revisited***
  - Why could we construct them in time O(n)?
- ***Amortized Analysis***
  - Analyzing data structures over the long term.
- ***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.

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 index of the minimum value. Its left and right children are Cartesian trees for the subarrays to the left and right of the minimum.



The Runtime Analysis

- Adding an individual node to a Cartesian tree might take time O(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 Tradeoff

- Typically, we've analyzed data structures by bounding the worst-case runtime of each operation.
- Sometimes, all we care about is the total runtime of a sequence of m operations, not the cost of each individual operation.
- ***Trade worst-case runtime per operation for worst-case runtime overall.***
- This is a fundamental technique in data structure design.

The Goal

- Suppose we have a data structure and perform a series of operations $op_1, op_2, \ldots, op_m$.
  - 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 the expression

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

- There are many ways to do this. We'll see three recurring techniques.

Amortized Analysis

- An ***amortized analysis*** is a different way of bounding the runtime of a sequence of operations.
- Each operation op_i really takes time $t(op_i)$.
- ***Idea:*** Assign to each operation op_i a new cost $a(op_i)$, called the ***amortized cost***, such that

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

- If the values of $a(op_i)$ are chosen wisely, the second sum can be much easier to evaluate than the first.

The Aggregate Method

- In the ***aggregate method***, we directly evaluate

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

  and then set $a(op_i) = T / m$.
- Assigns each operation the average of all the operation costs.
- The aggregate method says that the cost of a Cartesian tree insertion is amortized O(1).

Amortized Analysis

- We will see two types of amortized analysis today:
  - The ***banker's method*** (also called the ***accounting method***) works by placing "credits" on the data structure redeemable for units of work.
  - The ***potential method*** (also called the ***physicist's method***) works by assigning a potential function to the data structure and factoring in changes to that potential to the overall runtime.
- All three techniques are useful at different times, so we'll see how to use all three today.

The Banker's Method

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 somewhere takes time O(1).
- Credits may be removed from the data structure to pay for O(1) units of work.
- ***Note:*** the credits don't actually show up in the data structure. It's just an accounting trick.
- The amortized cost of an operation is

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

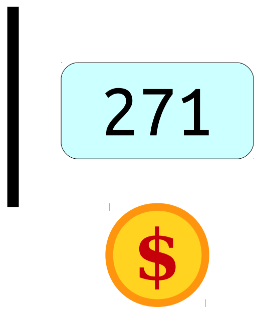
The Banker's Method

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

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

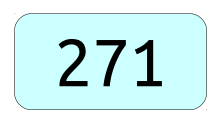
- The sum of the amortized costs upper-bounds the sum of the true costs.

Constructing Cartesian Trees



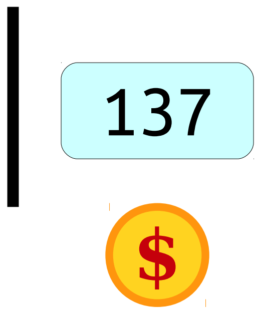
Work done: 1 push
Credits Added: $1

Amortized Cost: **2**



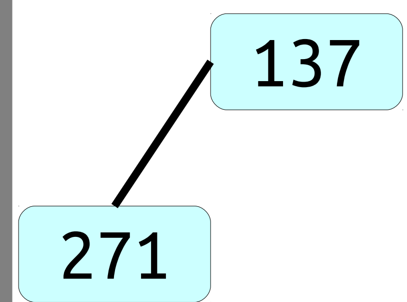
271	137	159	314	42

Constructing Cartesian Trees



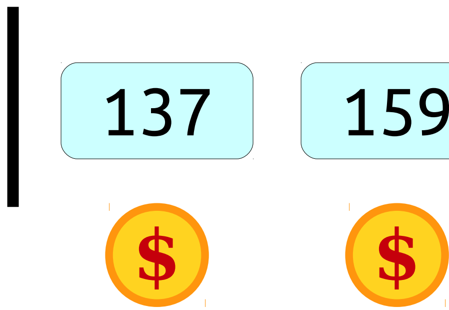
Work done: 1 push, 1 pop
Credits Removed: $1
Credits Added: $1

Amortized Cost: **2**



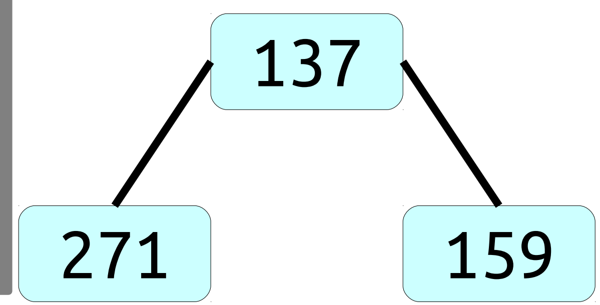
271	137	159	314	42

Constructing Cartesian Trees



Work done: 1 push
Credits Added: $1

Amortized Cost: **2**

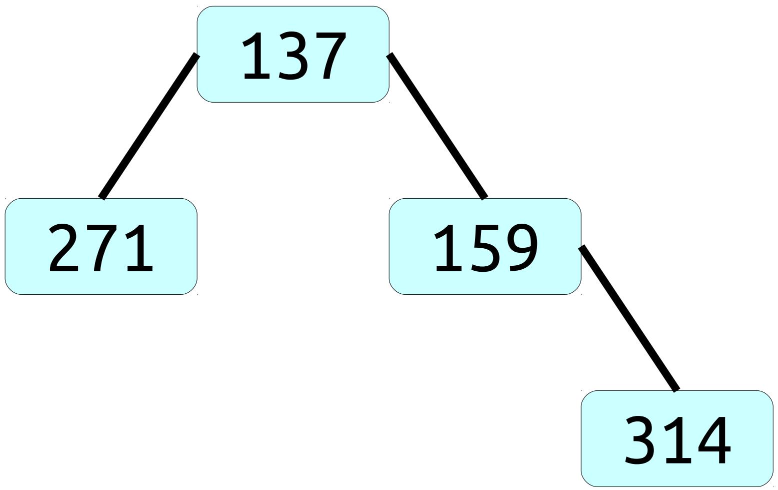


271	137	159	314	42

Constructing Cartesian Trees

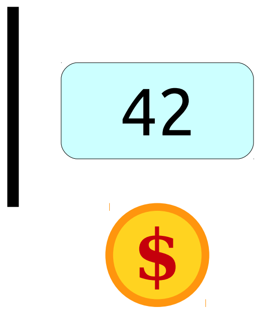
Work done: 1 push
Credits Added: $1

Amortized Cost: **2**



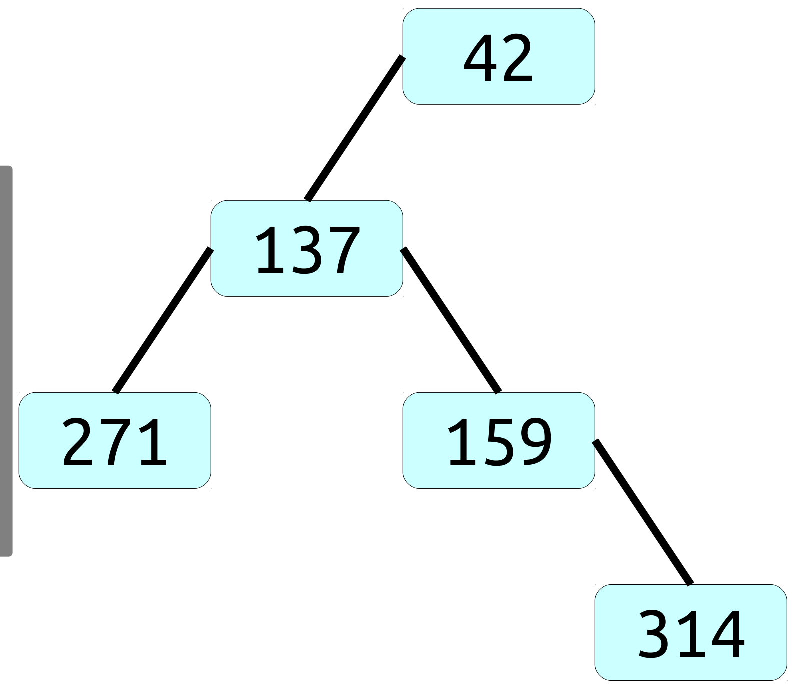
271	137	159	314	42

Constructing Cartesian Trees



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

Amortized Cost: **2**



271	137	159	314	42

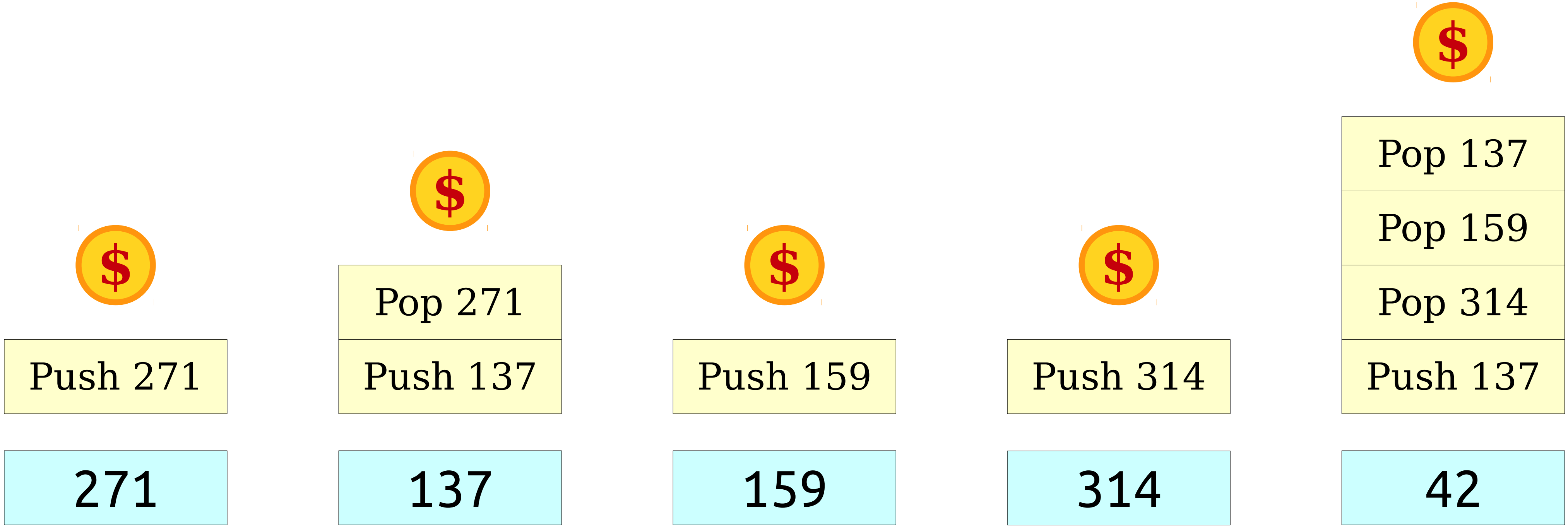
The Banker's Method

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

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

- 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

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 137
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.

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 on the data structure might 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

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

- Intuitively:
  - Operations that increase the potential have amortized cost greater than their true cost.
  - Operations that decrease the potential have amortized cost less than their true cost.

The Potential Method

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

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

Constructing Cartesian Trees

$\Phi = \mathbf{1}$

271

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

Amortized Cost: **2**

271

271	137	159	314	42

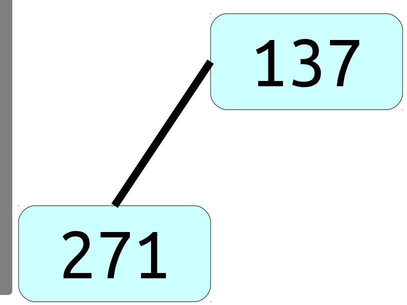
Constructing Cartesian Trees

Φ = **1**

137

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

Amortized Cost: **2**



271	137	159	314	42

Notice that Φ went

$1 \to 0 \to 1$

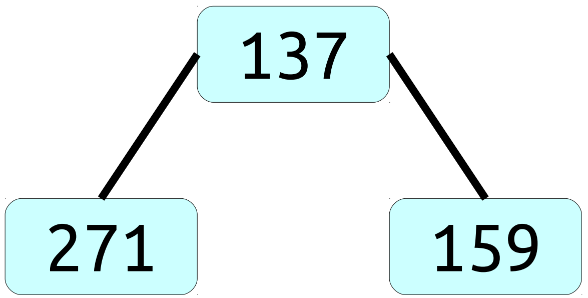
All that matters is the ***net*** change.

Constructing Cartesian Trees

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

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

Amortized Cost: **2**



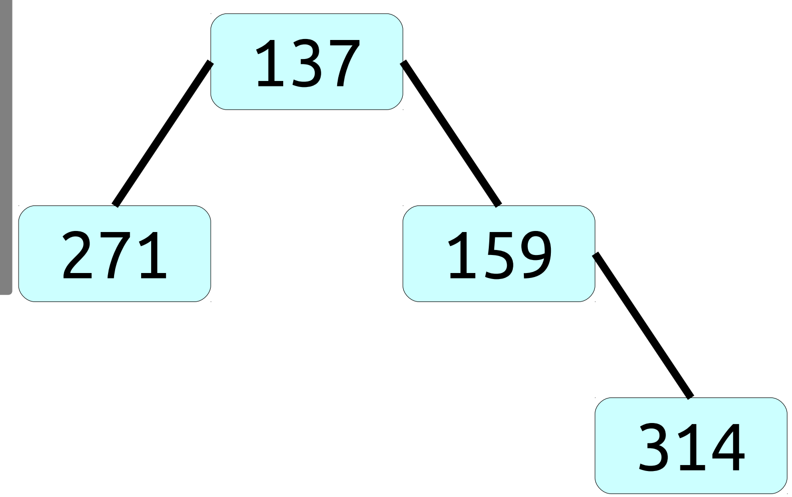
271	137	159	314	42

Constructing Cartesian Trees

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

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

Amortized Cost: **2**



271	137	159	314	42

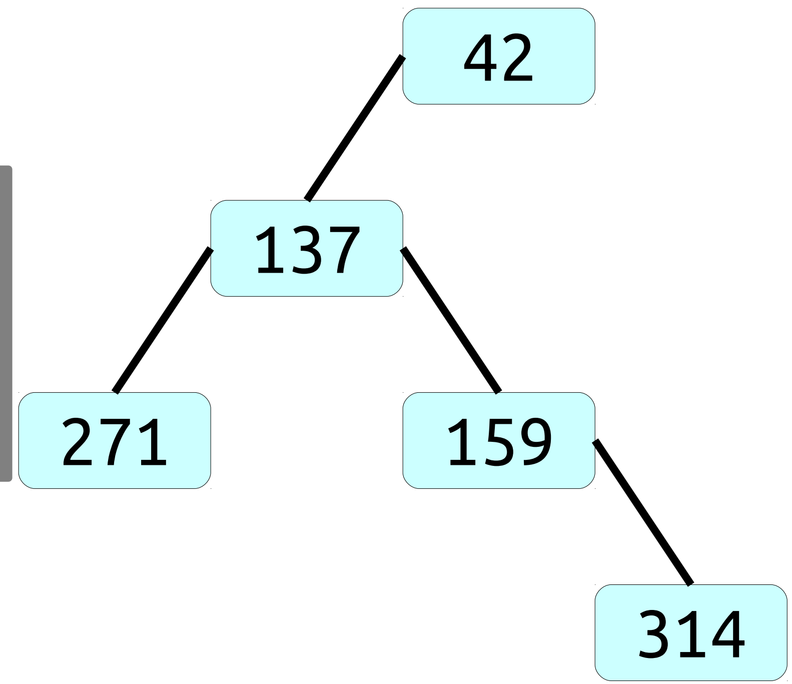
Constructing Cartesian Trees

$\Phi = \mathbf{1}$

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

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

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

Another Example: ***Two-Stack Queues***

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.

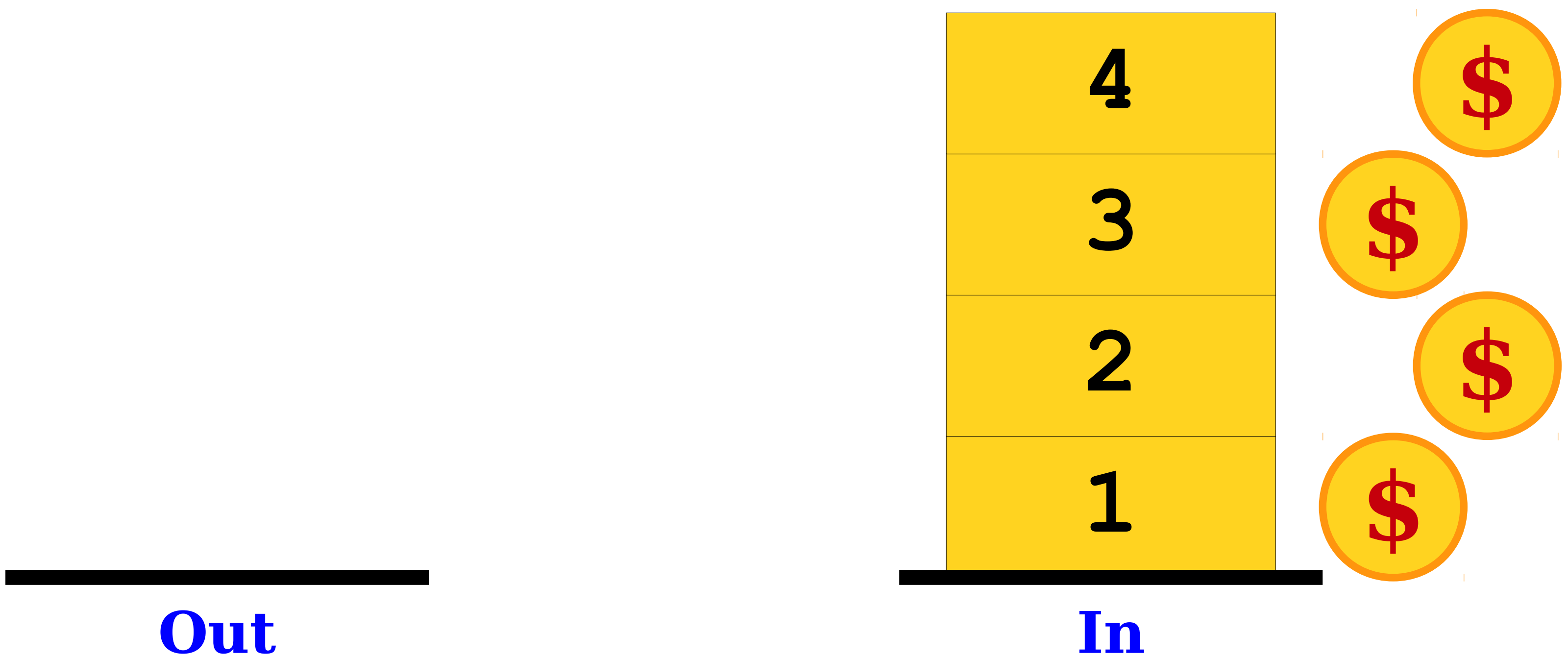
An Aggregate Analysis

- ***Claim:*** Cost of a sequence of *n* intermixed enqueues and dequeues is O(*n*).
- ***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*)**.

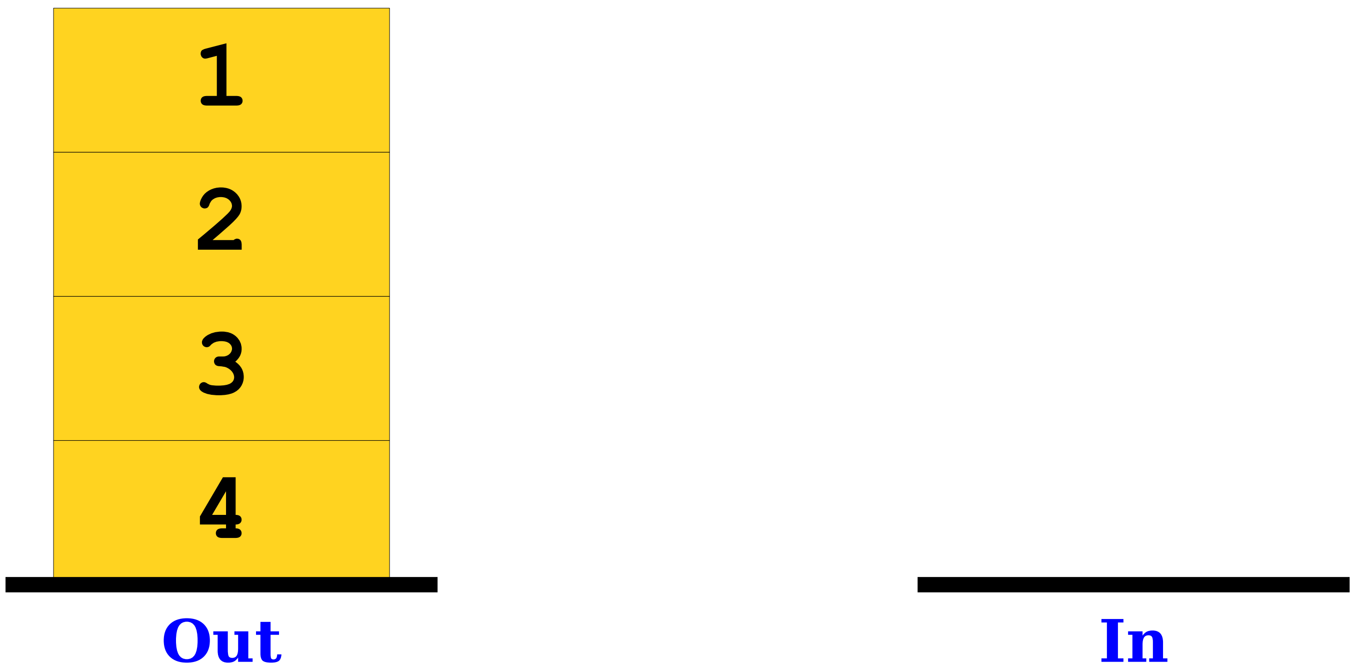
The Banker's Method

- 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



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 Set Two

- Problem Set Two solutions are now available. If you didn't pick them up in class, you can grab them from the Gates building.
- We're working on grading PS2 right now. We're aiming to have them returned by Tuesday of next week.

Problem Set Mixer

- Looking for a partner for the problem sets? Stick around after class today for our problem set mixer event.
- Free snacks!

Back to CS166!

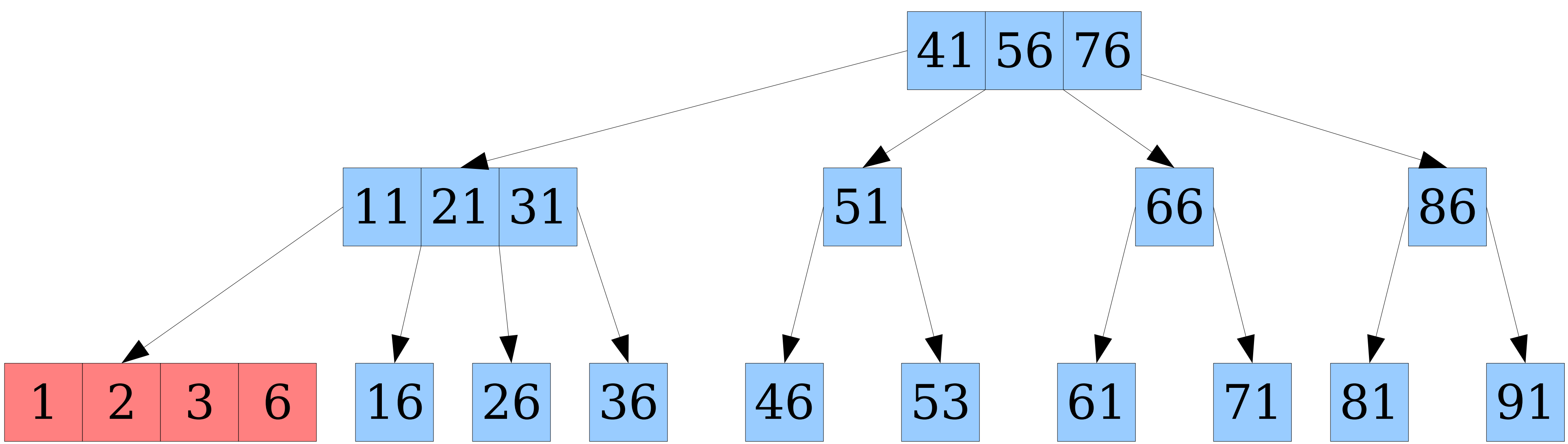
Another Example: ***2-3-4 Trees***

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 Θ(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?

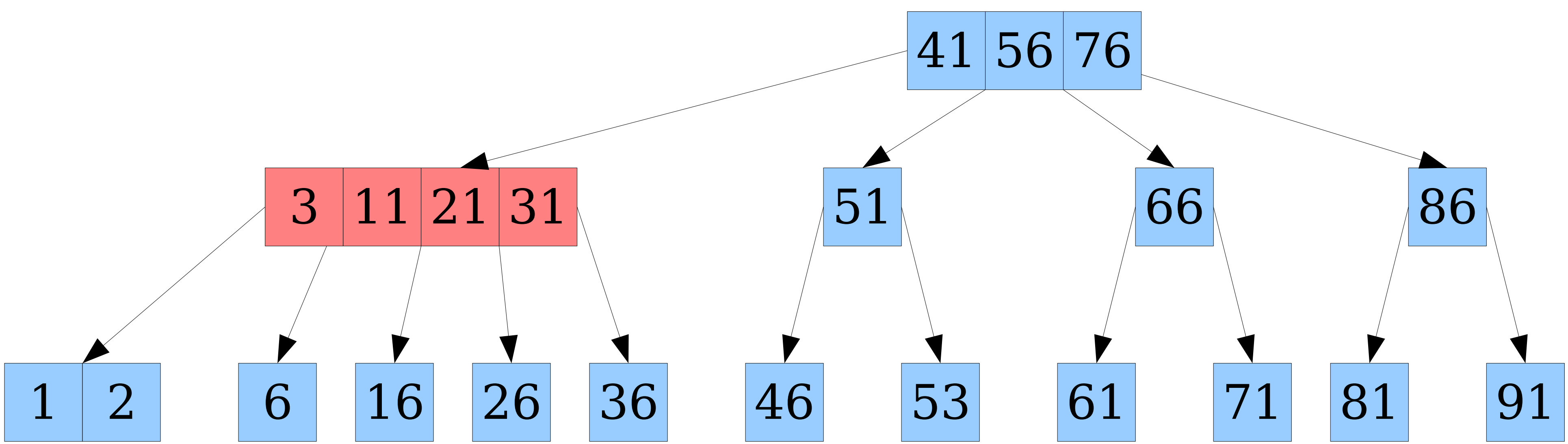
2-3-4 Tree Insertions

- 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.



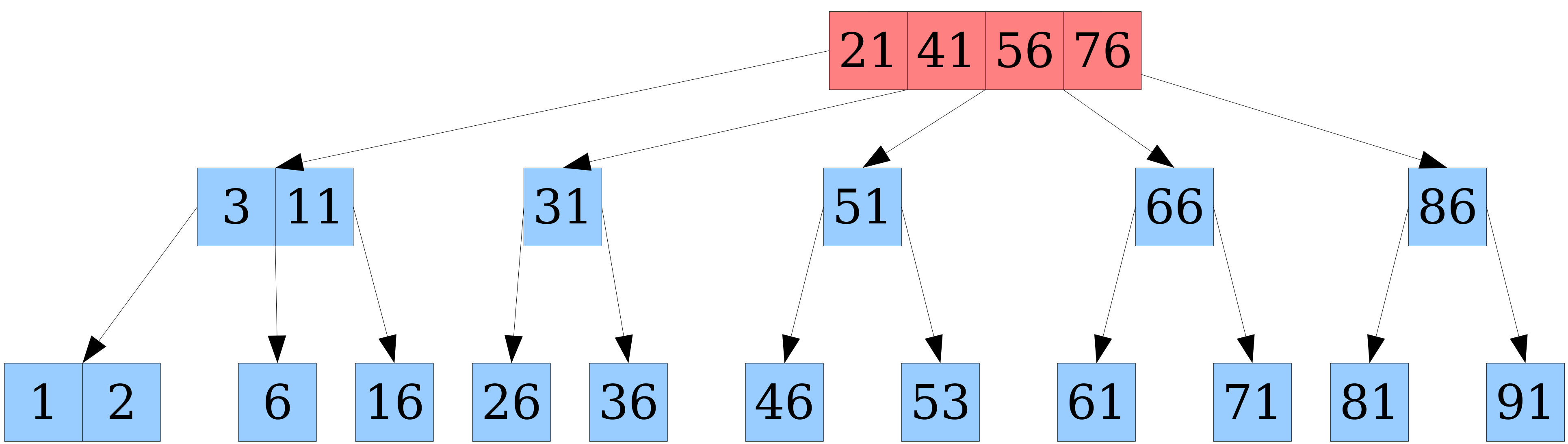
2-3-4 Tree Insertions

- 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.



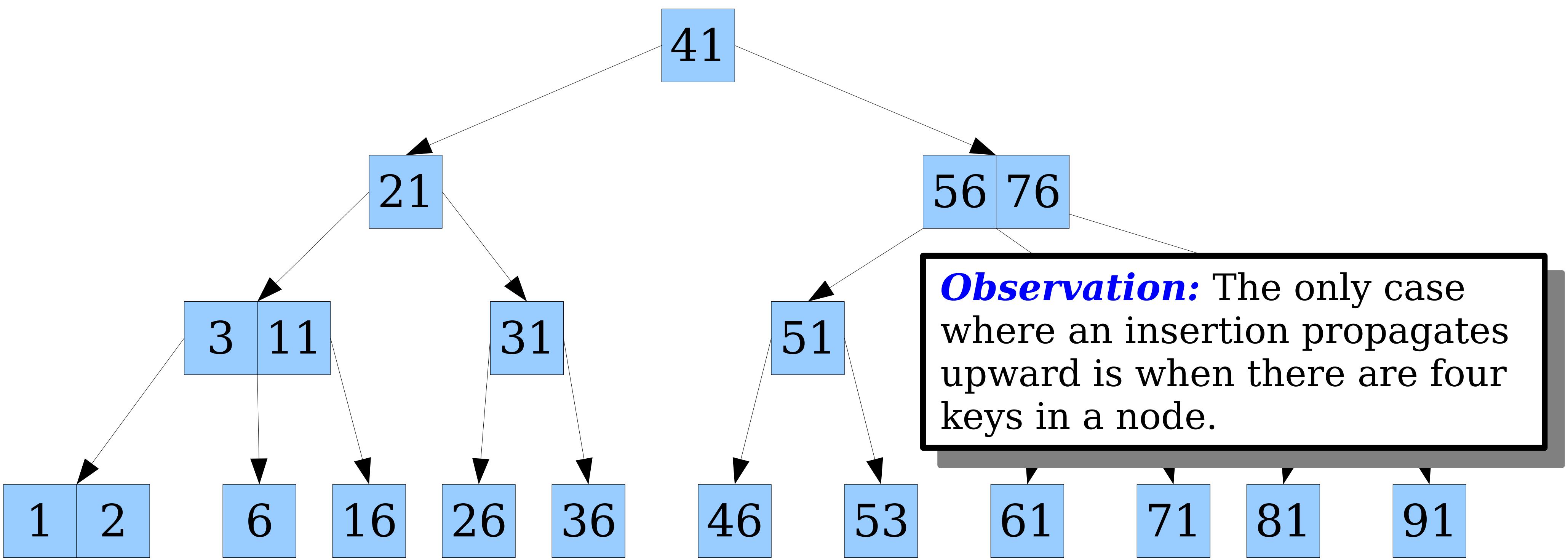
2-3-4 Tree Insertions

- 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.

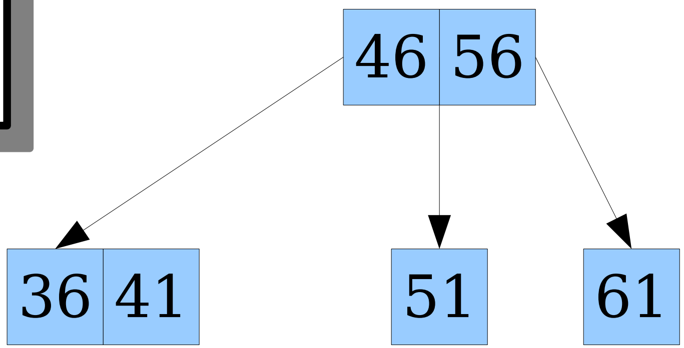


Observation: The only case where an insertion propagates upward is when there are four keys in a node.

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.

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

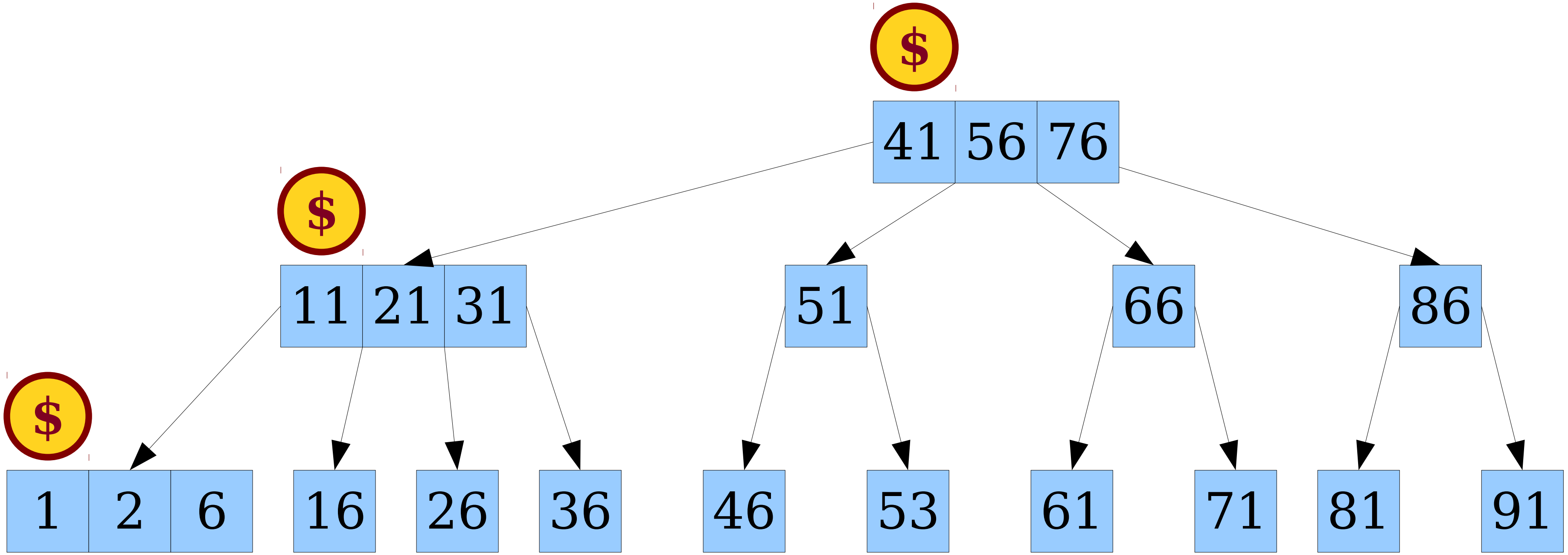


2-3-4 Tree Fixup

- ***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).

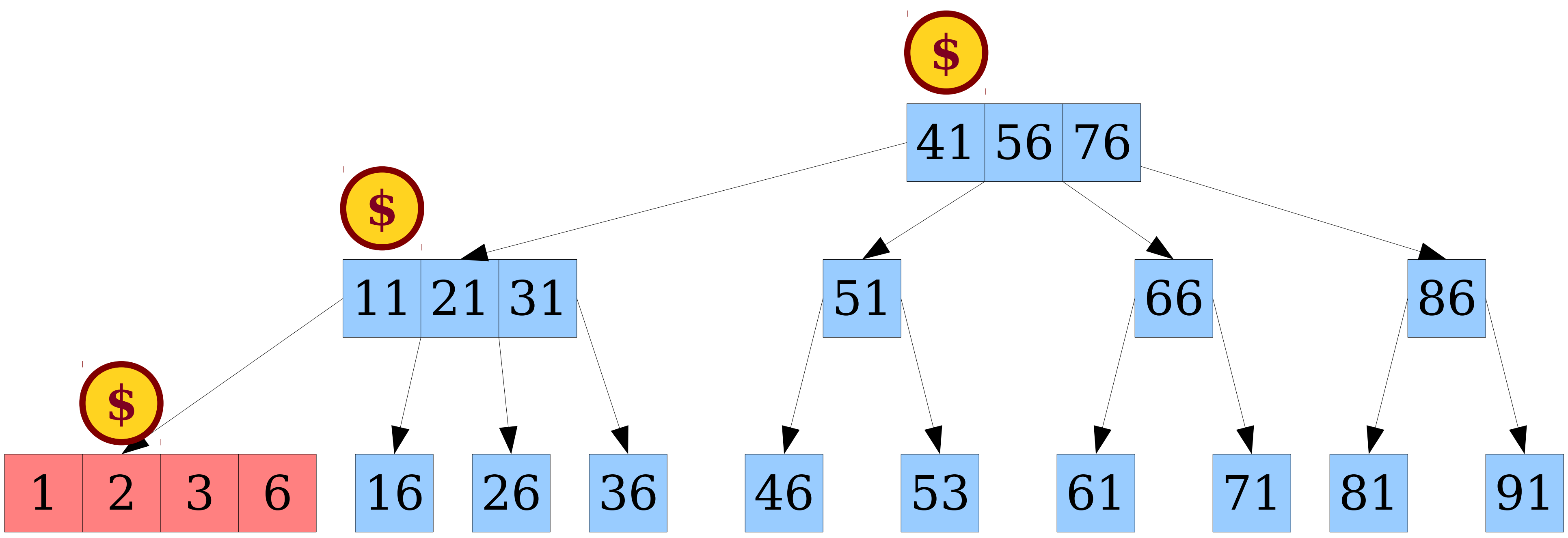
2-3-4 Tree Insertions

- 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.



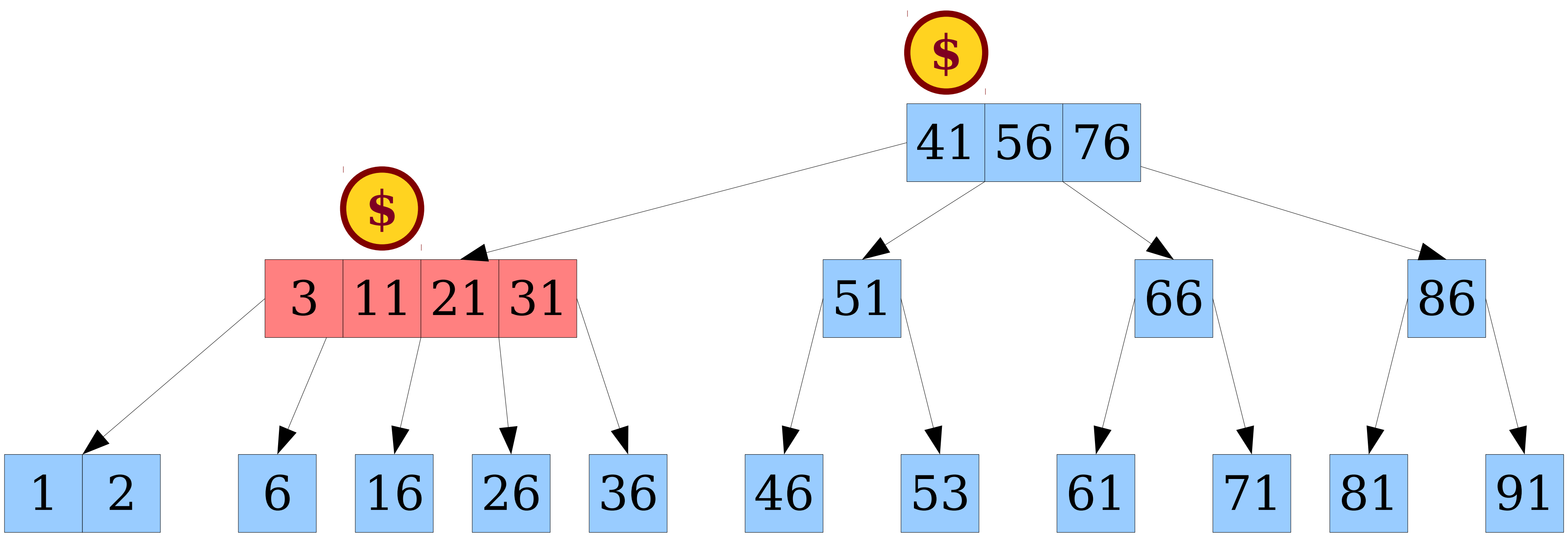
2-3-4 Tree Insertions

- 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.



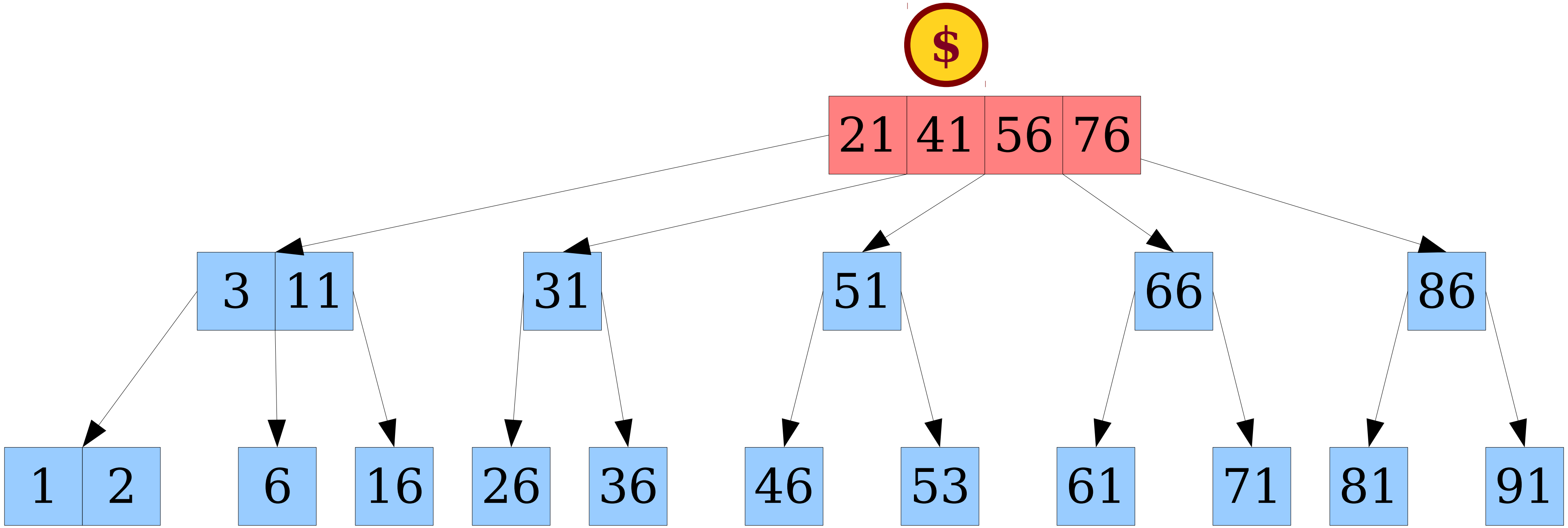
2-3-4 Tree Insertions

- 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.



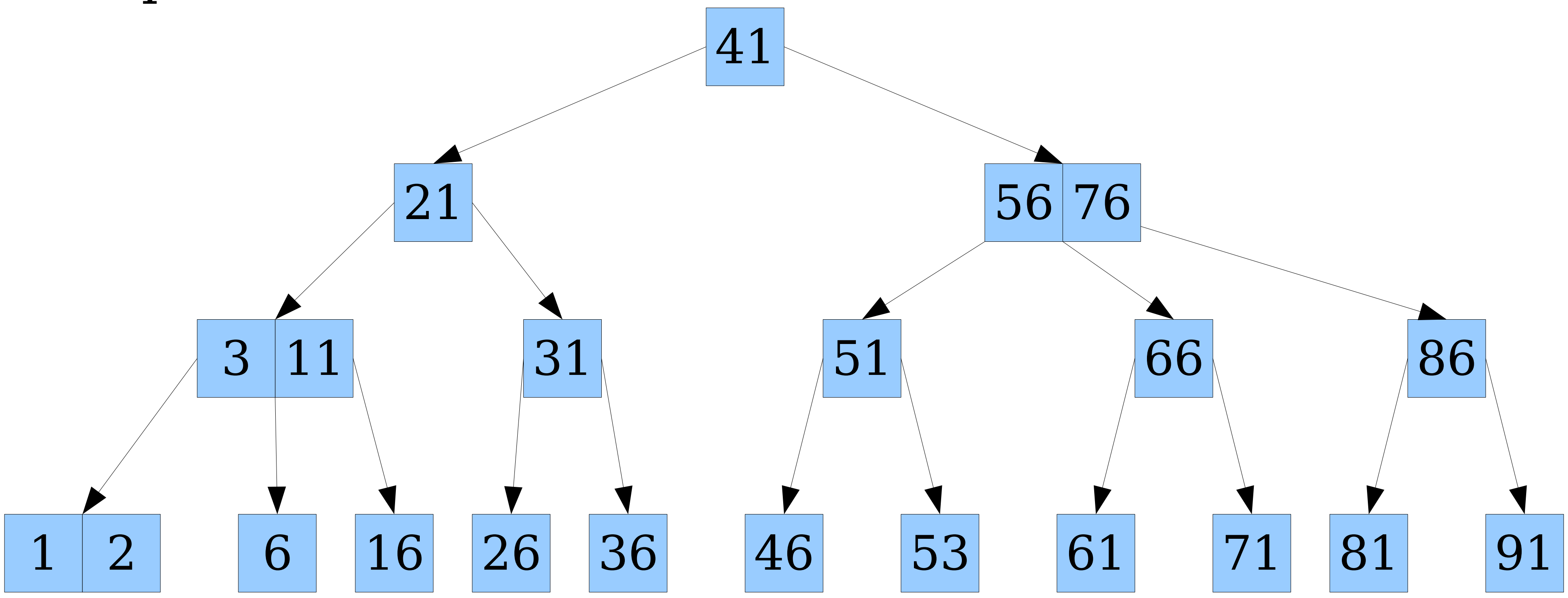
2-3-4 Tree Insertions

- 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.

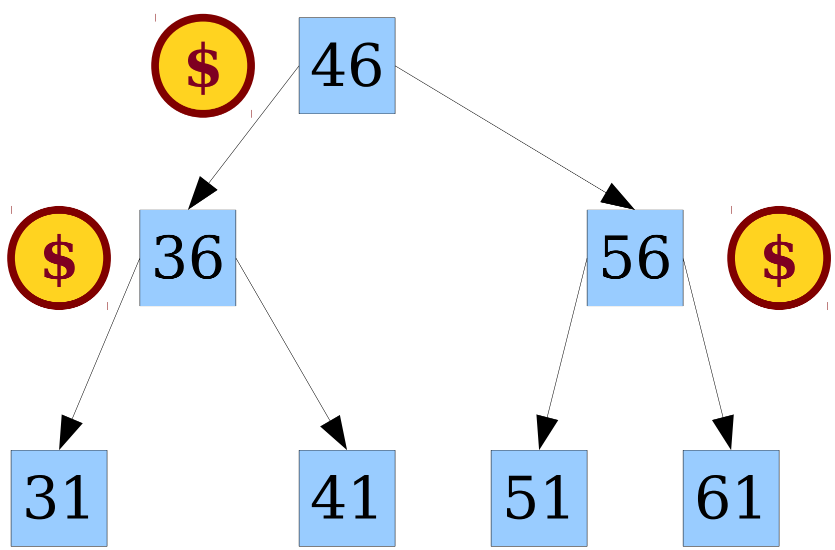


2-3-4 Tree Insertions

- 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.

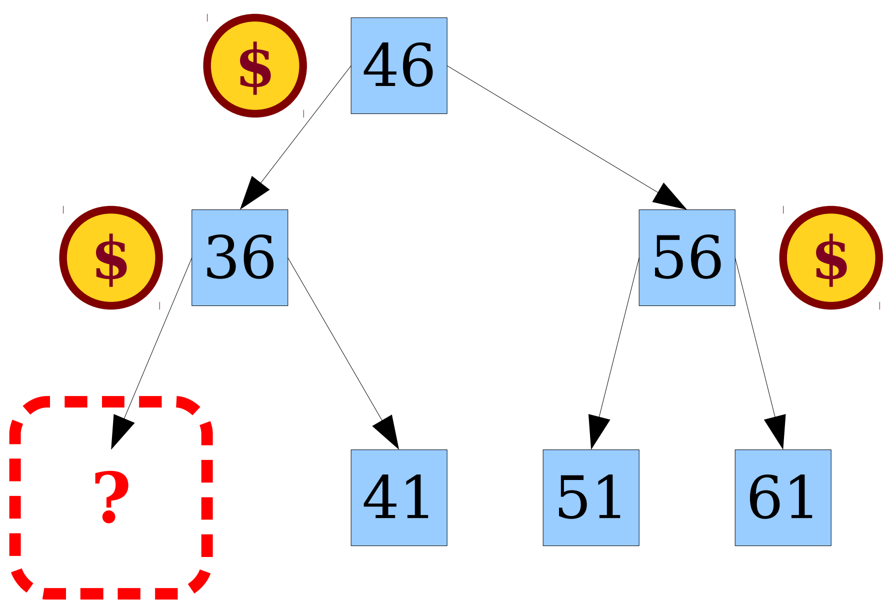
2-3-4 Tree Deletions

- 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.")



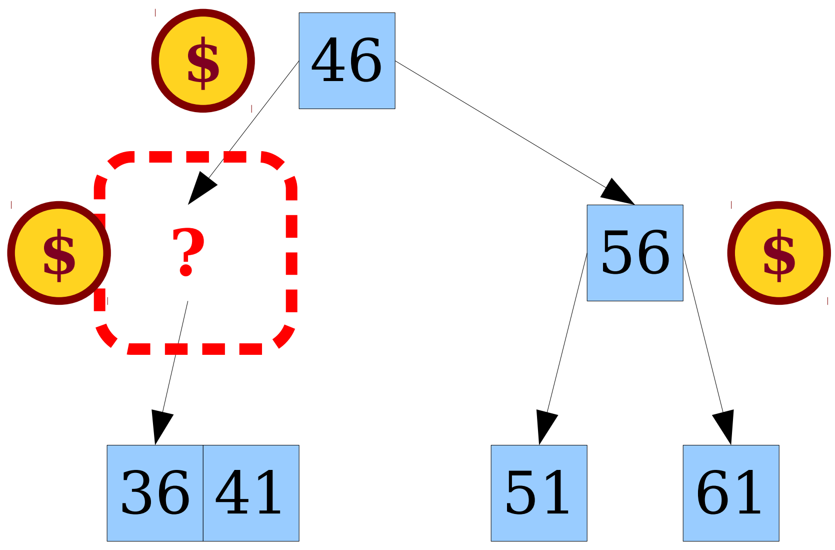
2-3-4 Tree Deletions

- 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.”)



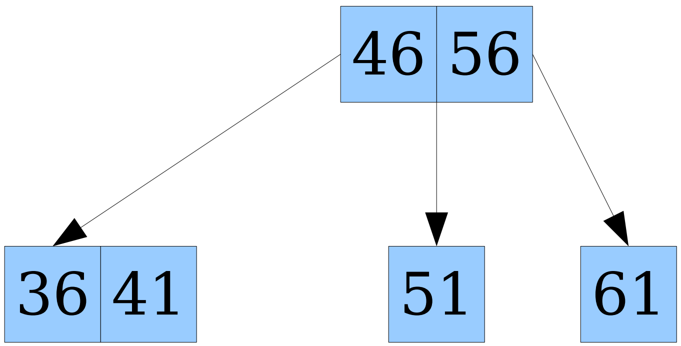
2-3-4 Tree Deletions

- 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.")

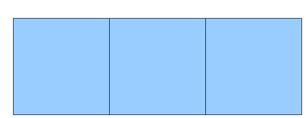
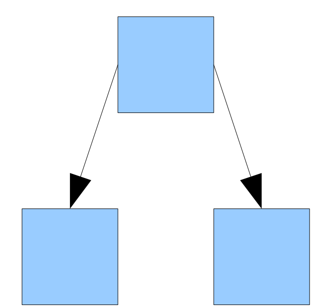


2-3-4 Tree Deletions

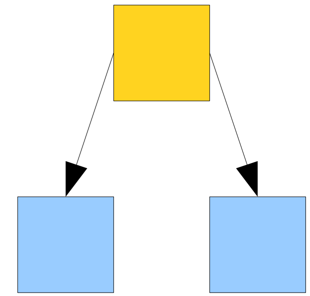
- Using the banker's method, we get that pure deletions have O(1) amortized fixup work.
- Could also do this using the potential method.
  - Define Φ to be the number of 2-nodes with two 2-node children (call these "tiny triangles.")
  - Each "light" deletion might introduce two tiny triangles: one at 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

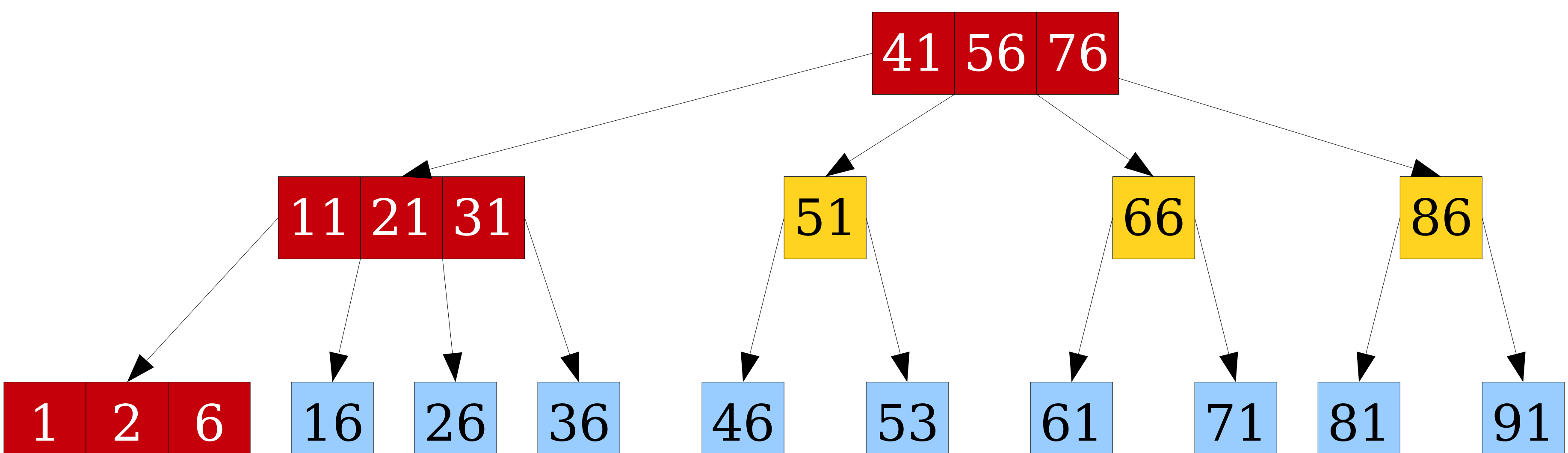
- We've shown that pure insertions and pure deletions require O(1) amortized fixup time.
- What about interleaved insertions and deletions?
- ***Initial idea:*** Use a potential function that's the sum of the two previous potential functions.
- Φ is the number of 4-nodes plus the number of tiny triangles.

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

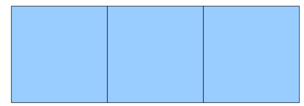
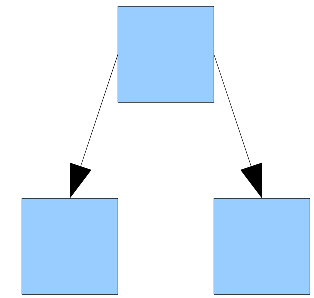
A Problem

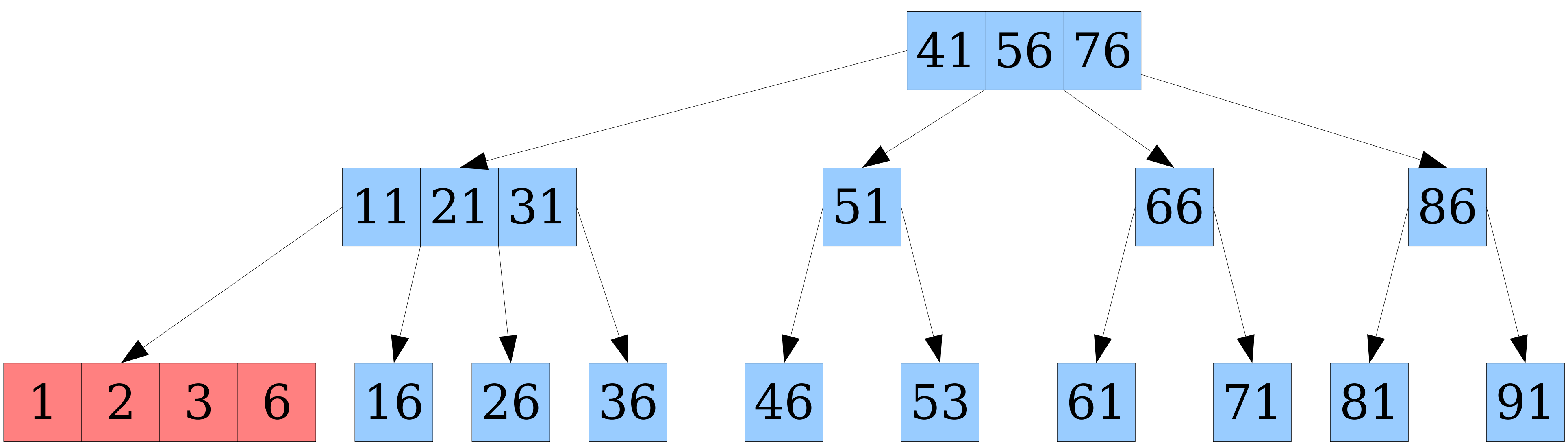
$\Phi =$ #() + #()

$= \mathbf{6}$

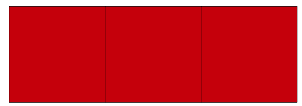
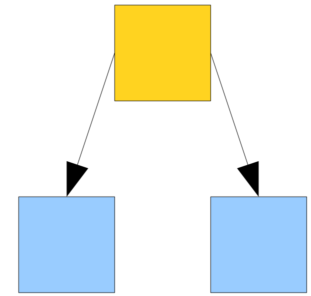


A Problem

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

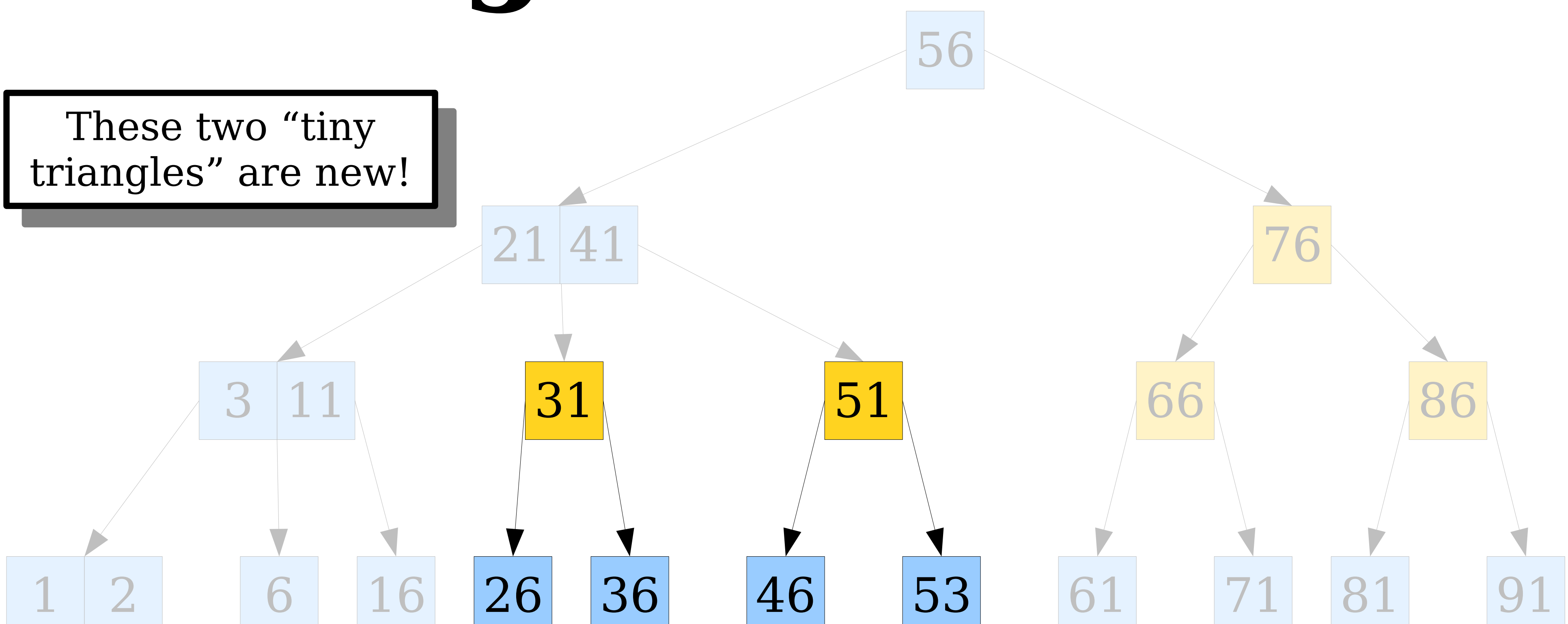


A Problem

Φ = #() + #()

= **5**

These two “tiny triangles” are new!

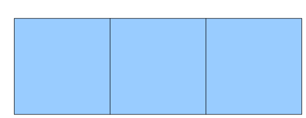
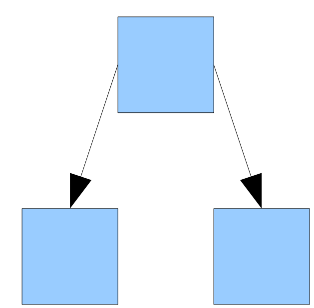


A Problem

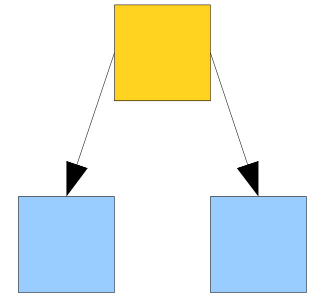
- When doing a “heavy” insertion that splits multiple 4-nodes, the resulting nodes might produce new “tiny triangles.”
- ***Symptom:*** Our potential doesn't drop nearly as much as it should, so we can't pay for future operations. Amortized cost of the operation works out to $\Theta(\log n)$, not $O(1)$ 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

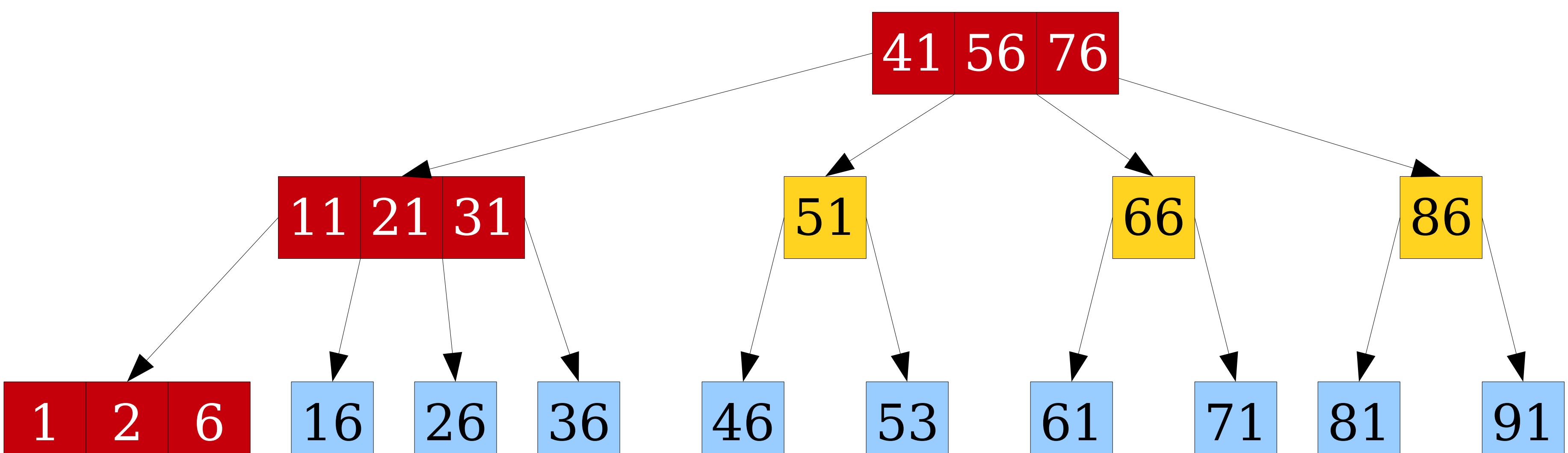
- 4-nodes are troublesome for two separate reasons:
  - They cause chained splits in an insertion.
  - After an insertion, they might split and produce a tiny triangle.
- ***Idea:*** Charge each 4-node for two different costs: the cost of an expensive insertion, plus the (possible) future cost of doing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

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

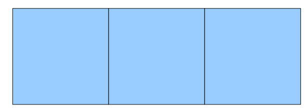
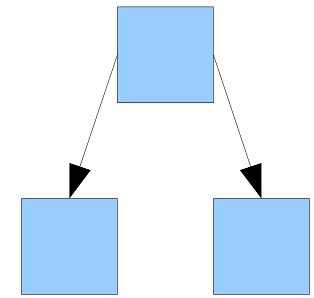
Unlocking our Potential

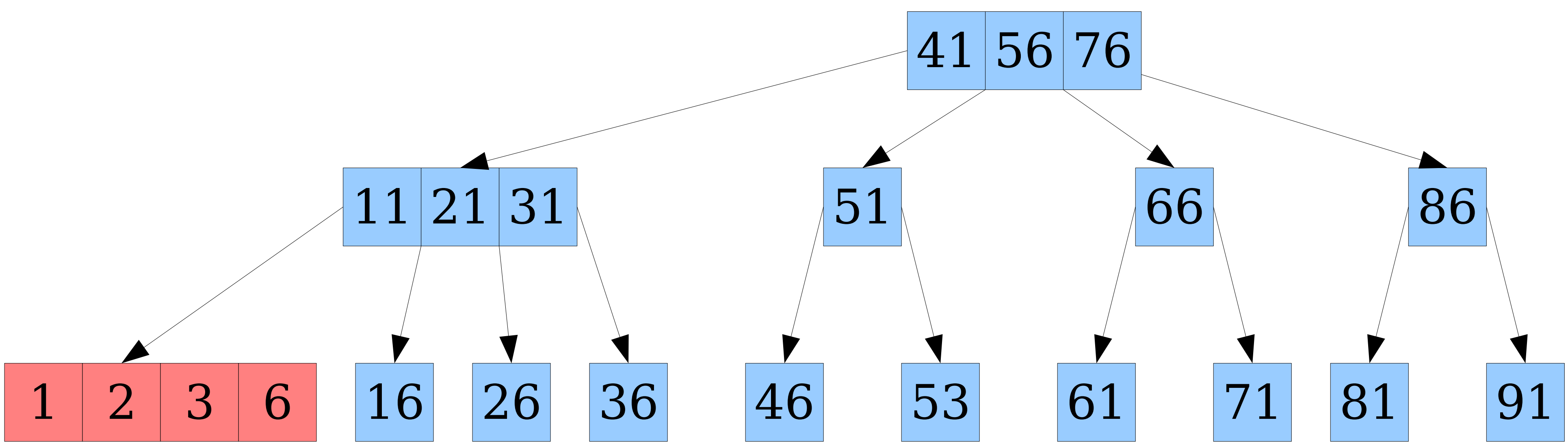
$\Phi = 2\#($  $) + \#($  $)$

$= \mathbf{9}$

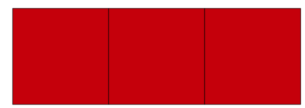
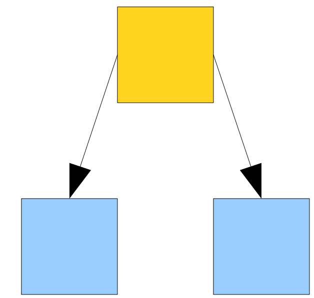


Unlocking our Potential

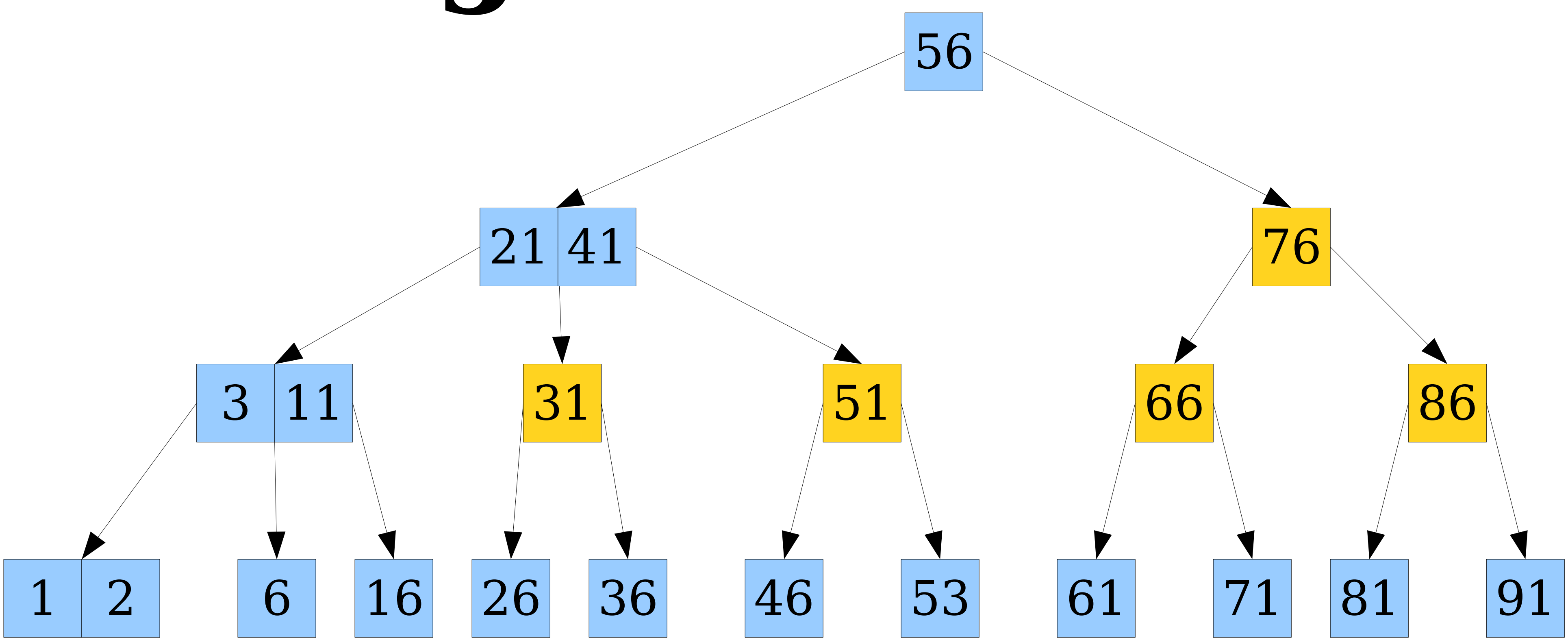
$\Phi = 2\#($  $) + \#($  $)$



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.