Amortized Analysis

Outline for Today

- **Euler Tour Trees**
  - A quick bug fix from last time.
- **Cartesian Trees Revisited**
  - Why could we construct them in time O(n)?
- **Amortized Analysis**
  - Analyzing data structures over the long term.
- **2-3-4 Trees**
  - A better analysis of 2-3-4 tree insertions and deletions.

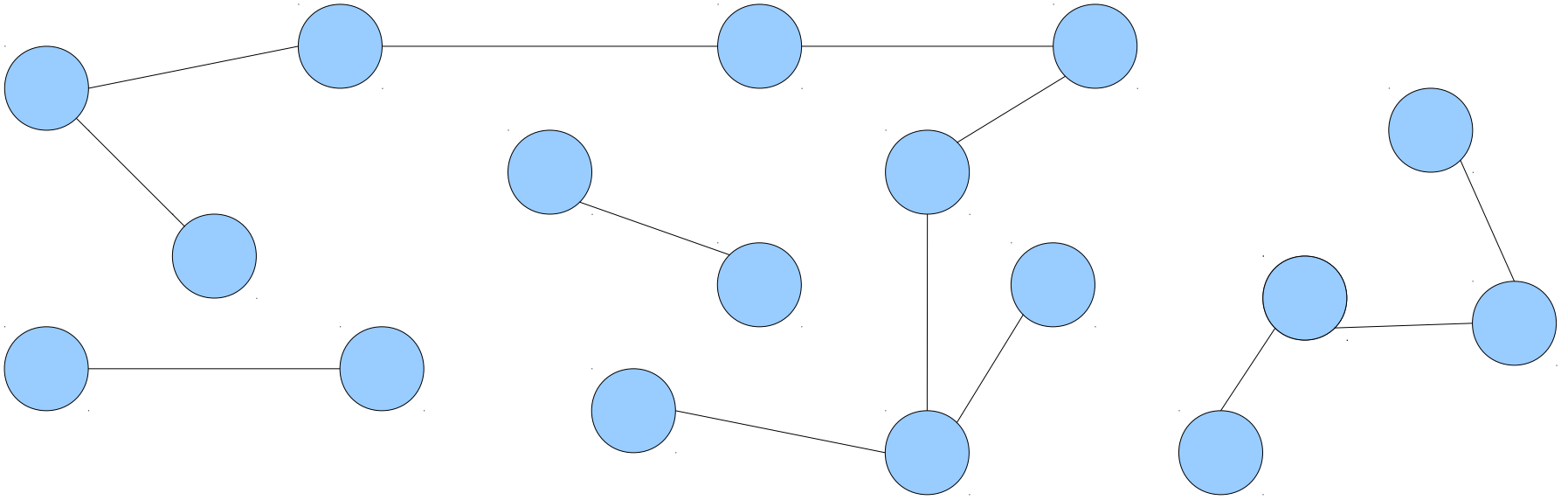
Review from Last Time

Dynamic Connectivity in Forests

- Consider the following special-case of the dynamic connectivity problem:

  Maintain an undirected **forest** *G* so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- Each deleted edge splits a tree in two; each added edge joins two trees and never closes a cycle.

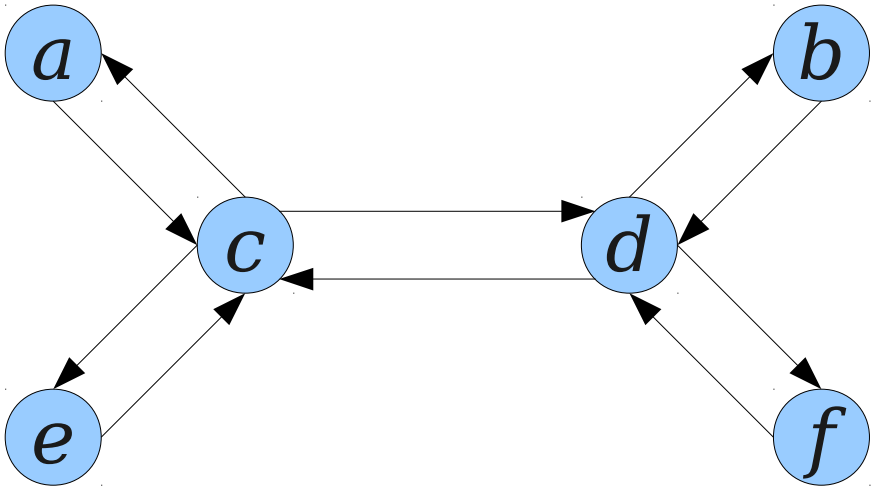


Dynamic Connectivity in Forests

- **Goal**: Support these three operations:
  - *link*(u, v): Add in edge $\{u, v\}$. The assumption is that u and v are in separate trees.
  - *cut*(u, v): Cut the edge $\{u, v\}$. The assumption is that the edge exists in the tree.
  - *is-connected*(u, v): Return whether u and v are connected.
- The data structure we'll develop can perform these operations time **O(log *n*)** each.

Euler Tours on Trees

- In general, trees do not have Euler tours.



a c d b d f d c e c a

- **Technique:** replace each edge $\{u, v\}$ with two edges (u, v) and (v, u).
- Resulting graph has an Euler tour.

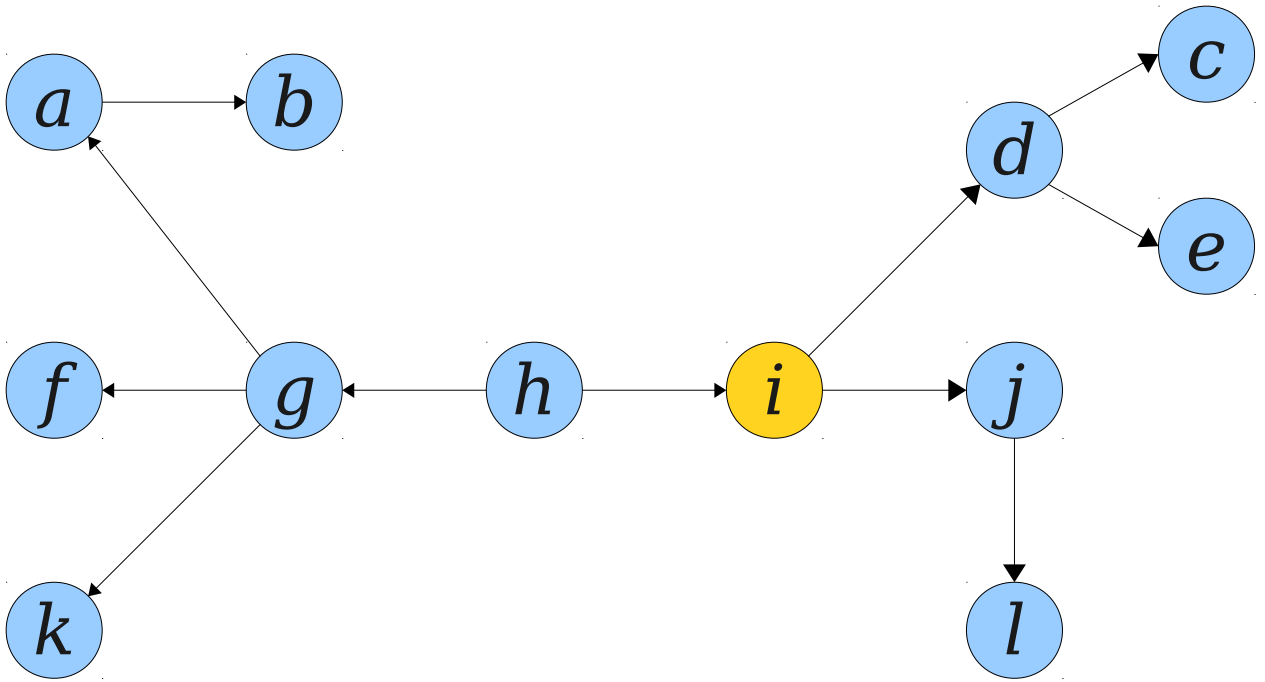
A Correction from Last Time

The Bug

- The previous representation of Euler tour trees required us to store pointers to the first and last instance of each node in the tours.
- This can't be updated in time O(1) after rotating a tour, so the operations have the wrong time bounds.
- We need to update our approach so that this is no longer necessary.

Rerooting a Tour

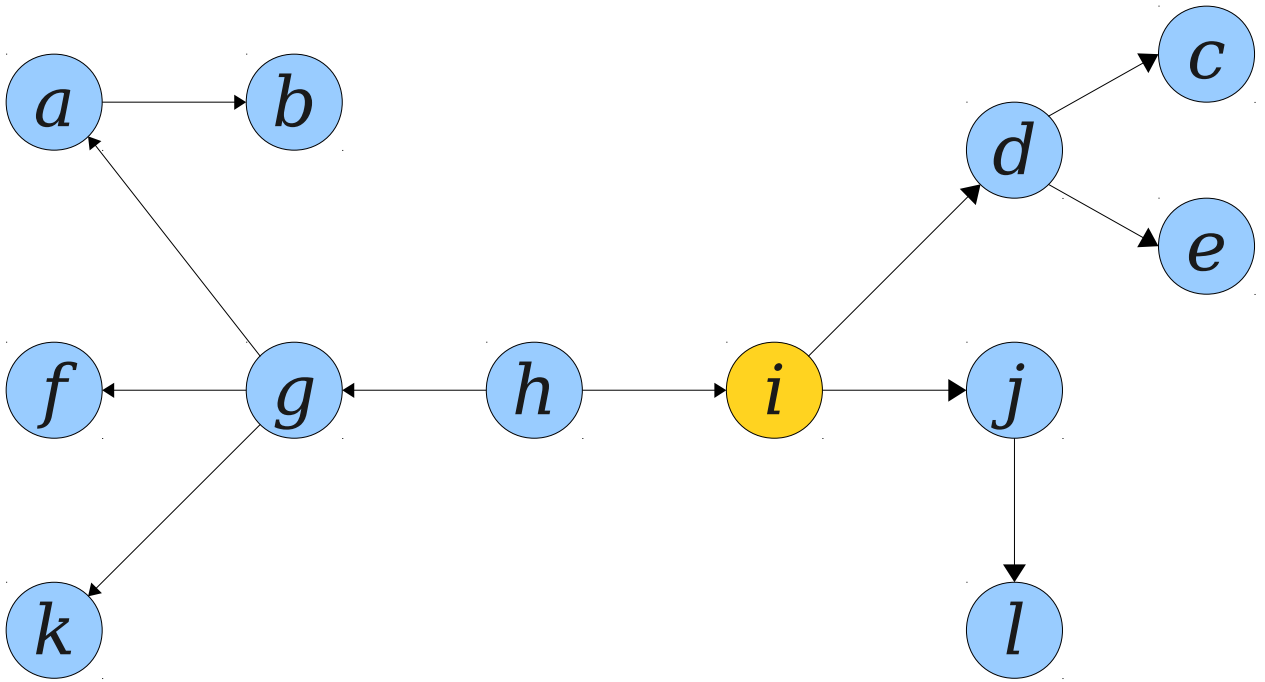
- The subtrees defined by ranges in Euler tours on trees depend on the root.
- In some cases, we will need to change the root of the tree.



h i d c d e d i j l j i h g f g k g a b a g h

Rerooting a Tour

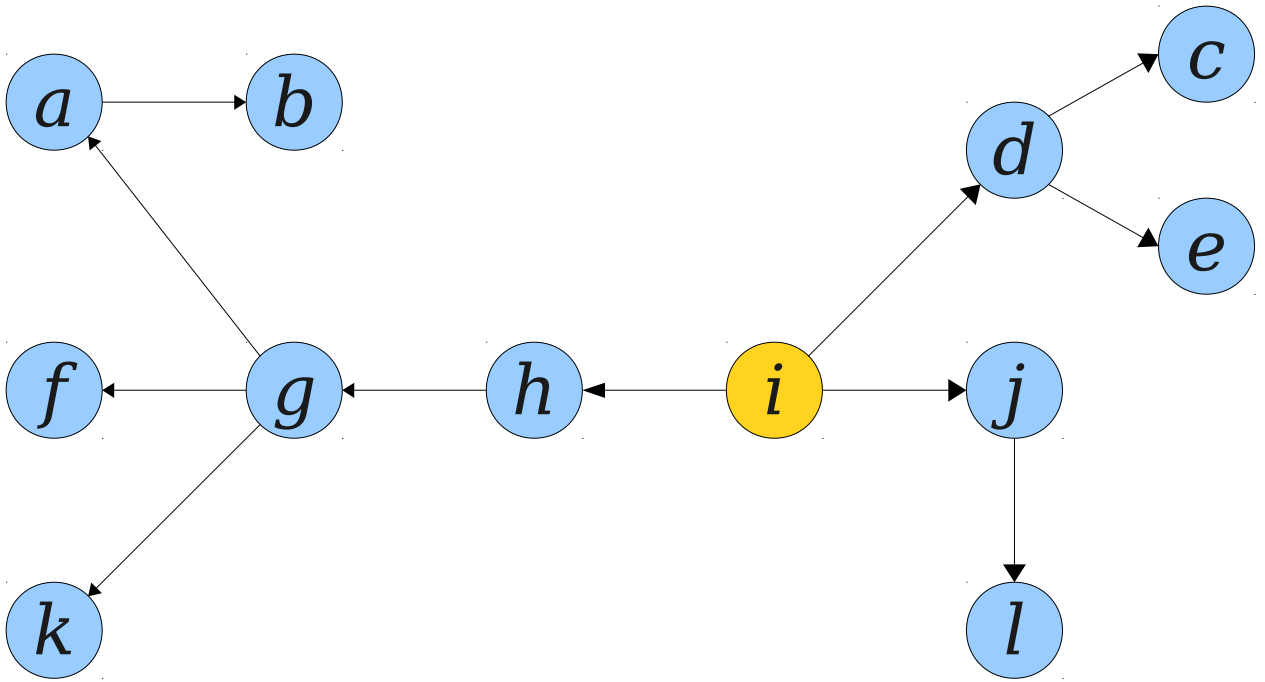
- The subtrees defined by ranges in Euler tours on trees depend on the root.
- In some cases, we will need to change the root of the tree.



h i d c d e d i j l j i h g f g k g a b a g h

Rerooting a Tour

- The subtrees defined by ranges in Euler tours on trees depend on the root.
- In some cases, we will need to change the root of the tree.



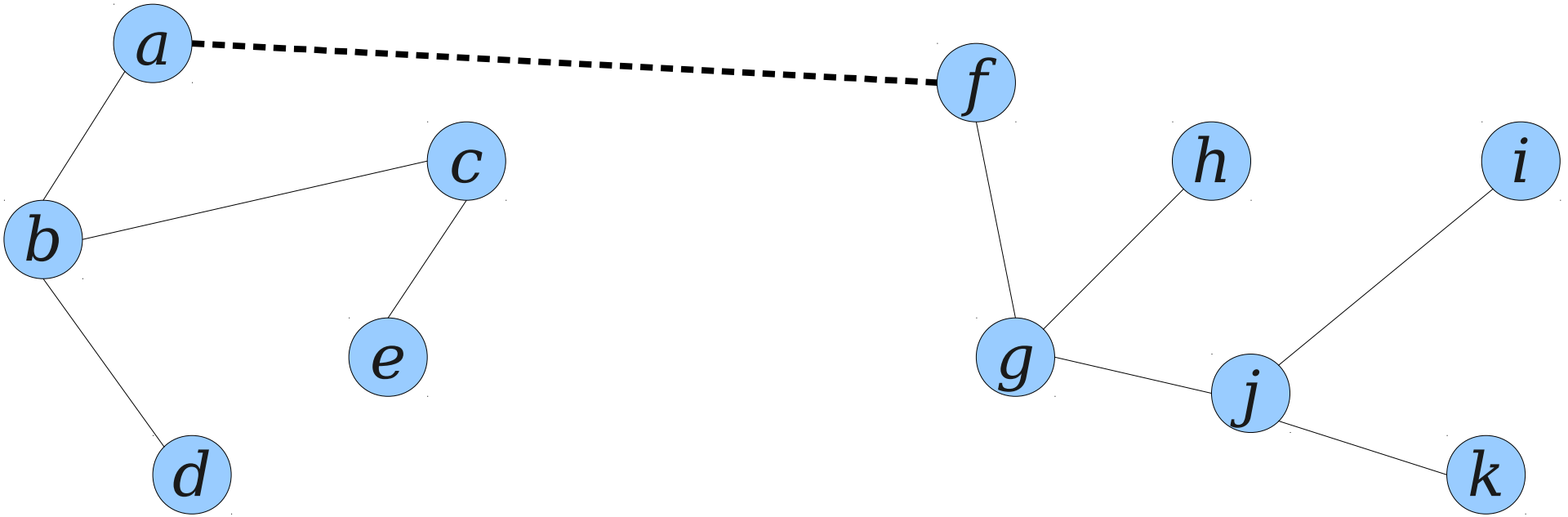
i j l j i h g f g k g a b a g h i d c d e d i

Rerooting a Tour

- **Algorithm:**
  - Find ***any*** copy of the node r that will be the new root.
  - Split the tour E right before r into E_1 and E_2.
  - Delete the first node from E_1.
  - Concatenate E_1, E_2, $\{r\}$.
- **Difference from before:** We only need a single pointer to r, not the full range.

Euler Tours and Dynamic Trees

- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link(u, v)*** links the trees together by adding edge $\{u, v\}$.
- Watch what happens to the Euler tours:

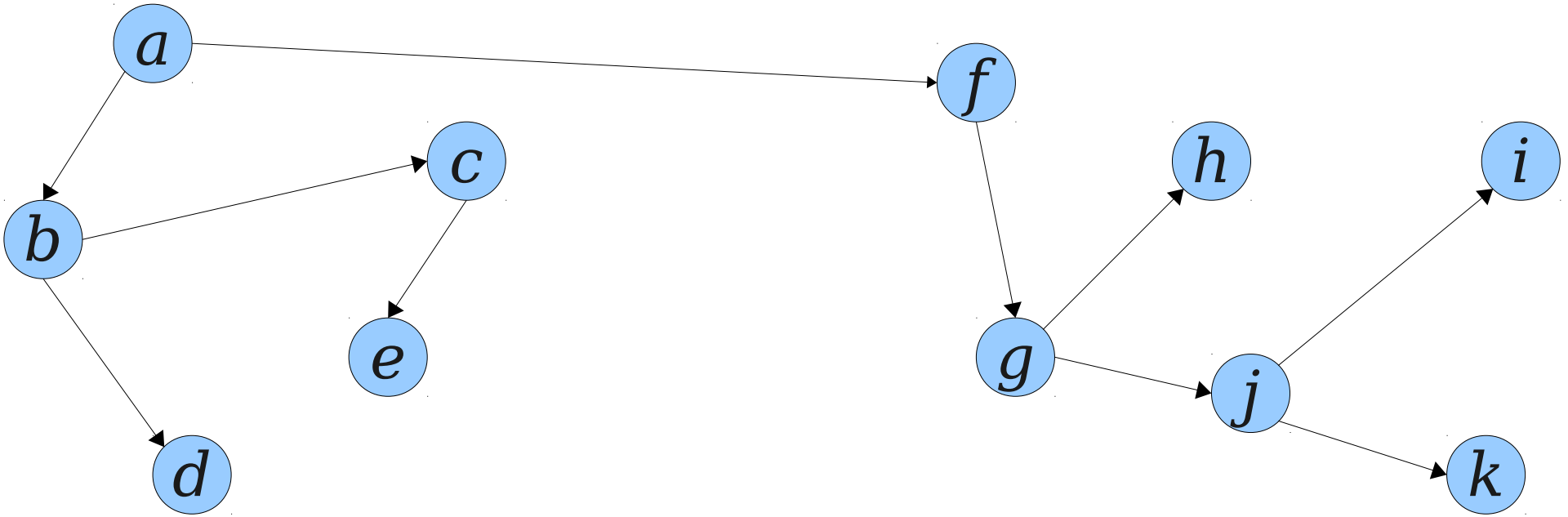


a b d b c e c b a

f g j k j i j g h g f

Euler Tours and Dynamic Trees

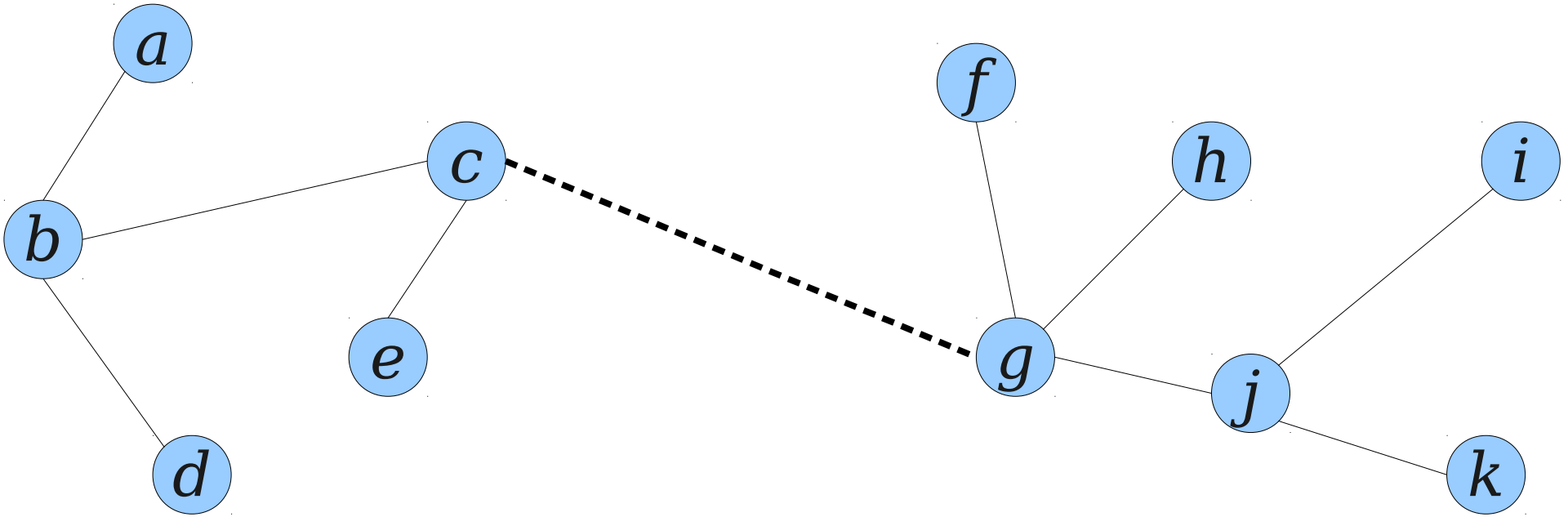
- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link(u, v)*** links the trees together by adding edge $\{u, v\}$.
- Watch what happens to the Euler tours:



a b d b c e c b a f g j k j i j g h g f a

Euler Tours and Dynamic Trees

- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link(u, v)*** links the trees together by adding edge $\{u, v\}$.
- Watch what happens to the Euler tours:

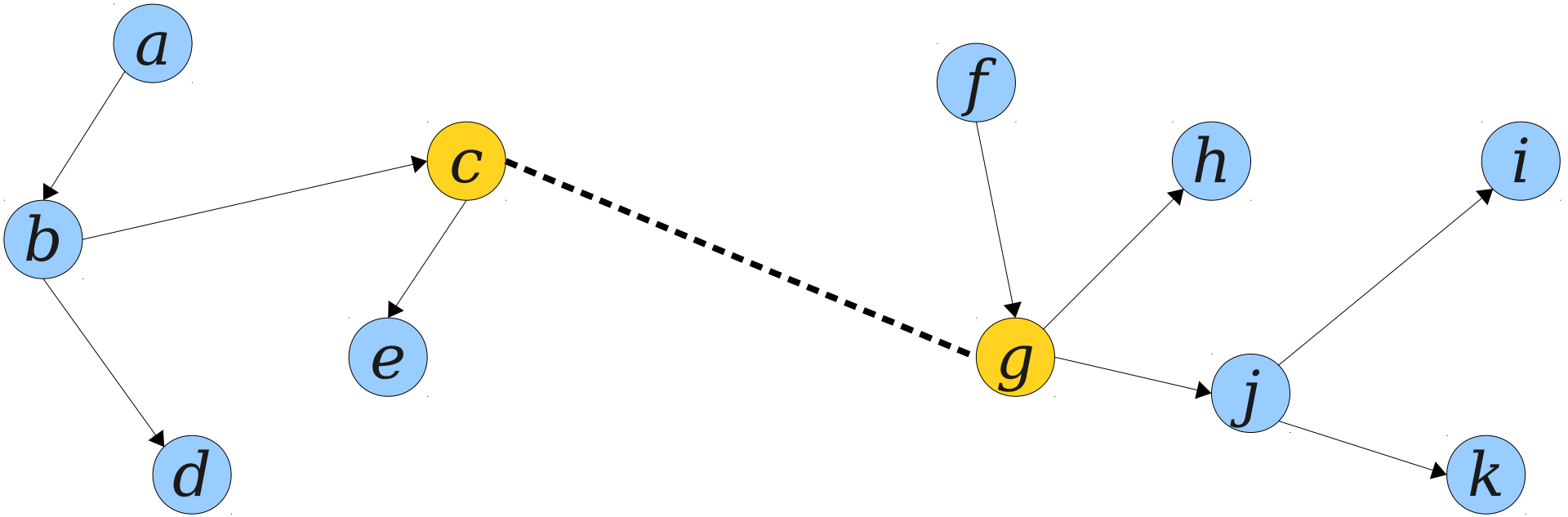


a b d b c e c b a

f g j k j i j g h g f

Euler Tours and Dynamic Trees

- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link(u, v)*** links the trees together by adding edge $\{u, v\}$.
- Watch what happens to the Euler tours:

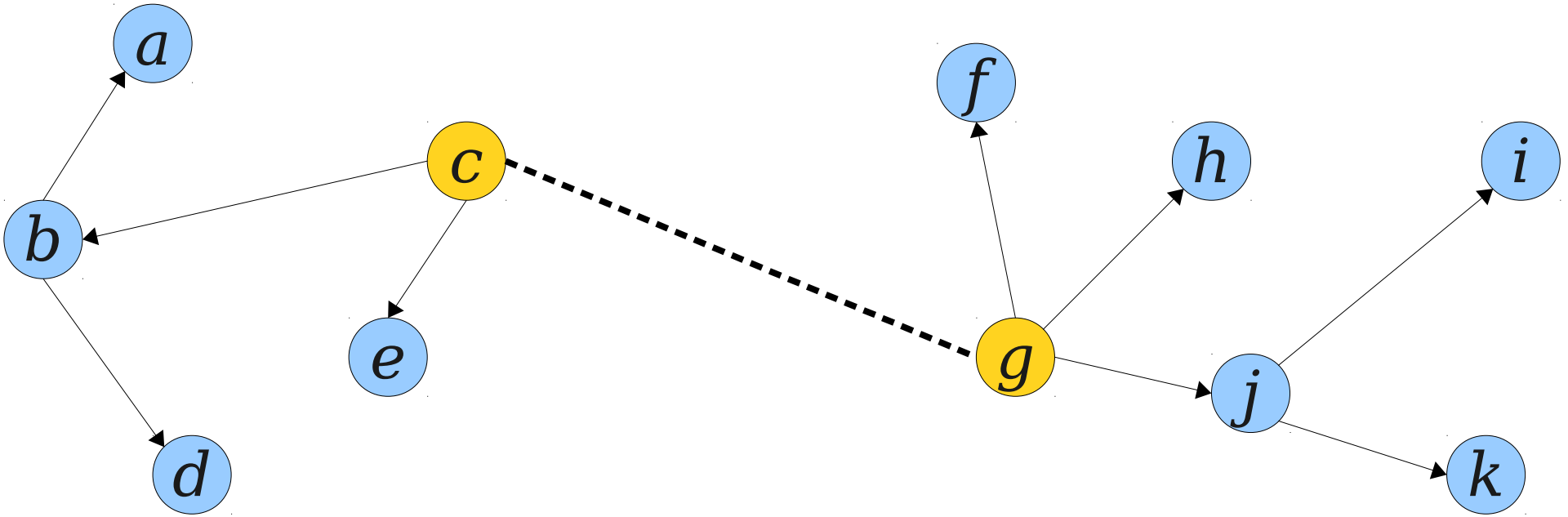


a b d b c e c b a

f g j k j i j g h g f

Euler Tours and Dynamic Trees

- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link(u, v)*** links the trees together by adding edge $\{u, v\}$.
- Watch what happens to the Euler tours:

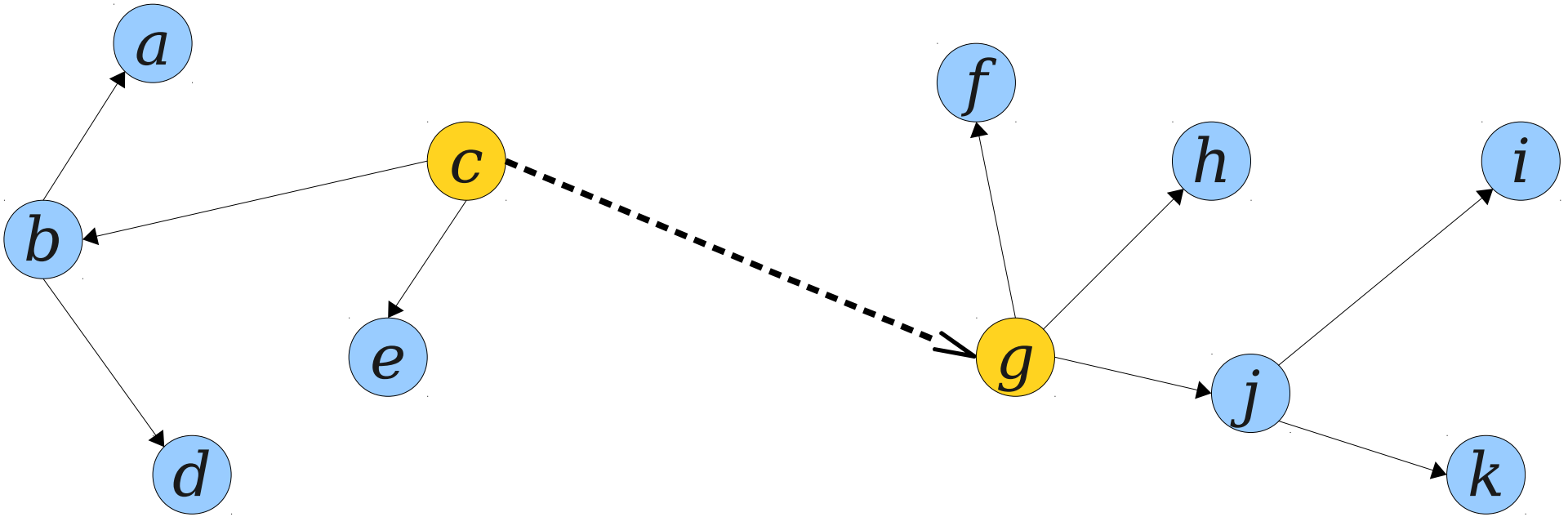


c e c b a b d b c

g j k j i j g h g f g

Euler Tours and Dynamic Trees

- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link(u, v)*** links the trees together by adding edge $\{u, v\}$.
- Watch what happens to the Euler tours:

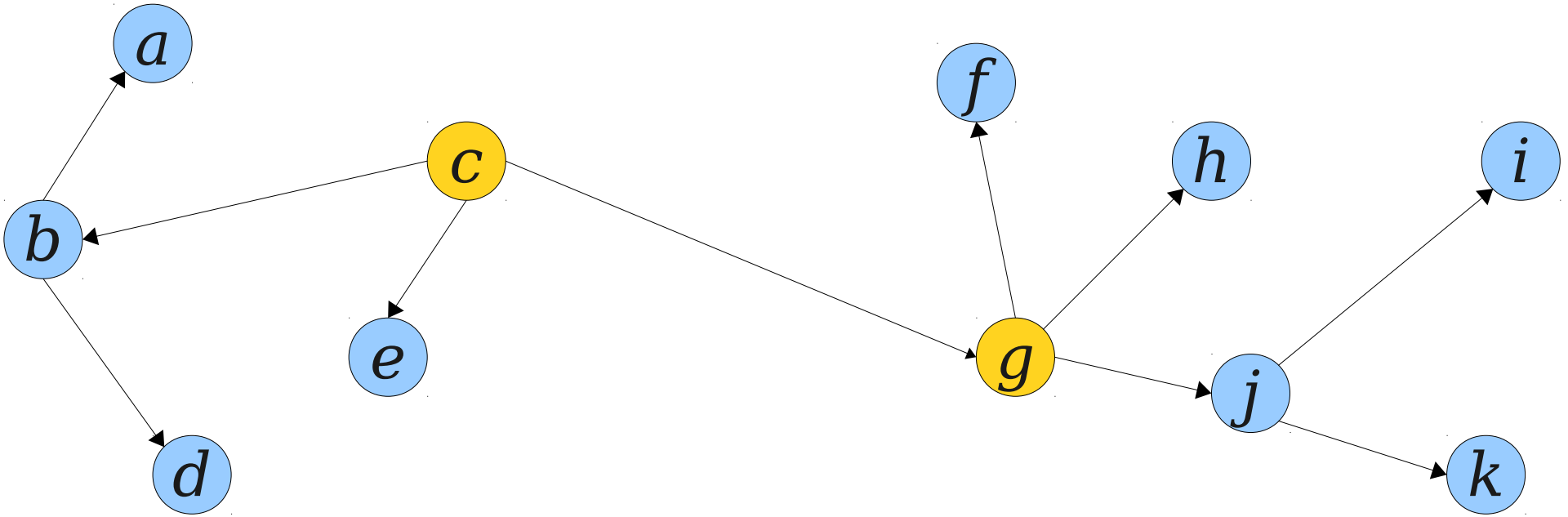


c e c b a b d b c

g j k j i j g h g f g

Euler Tours and Dynamic Trees

- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link(u, v)*** links the trees together by adding edge $\{u, v\}$.
- Watch what happens to the Euler tours:



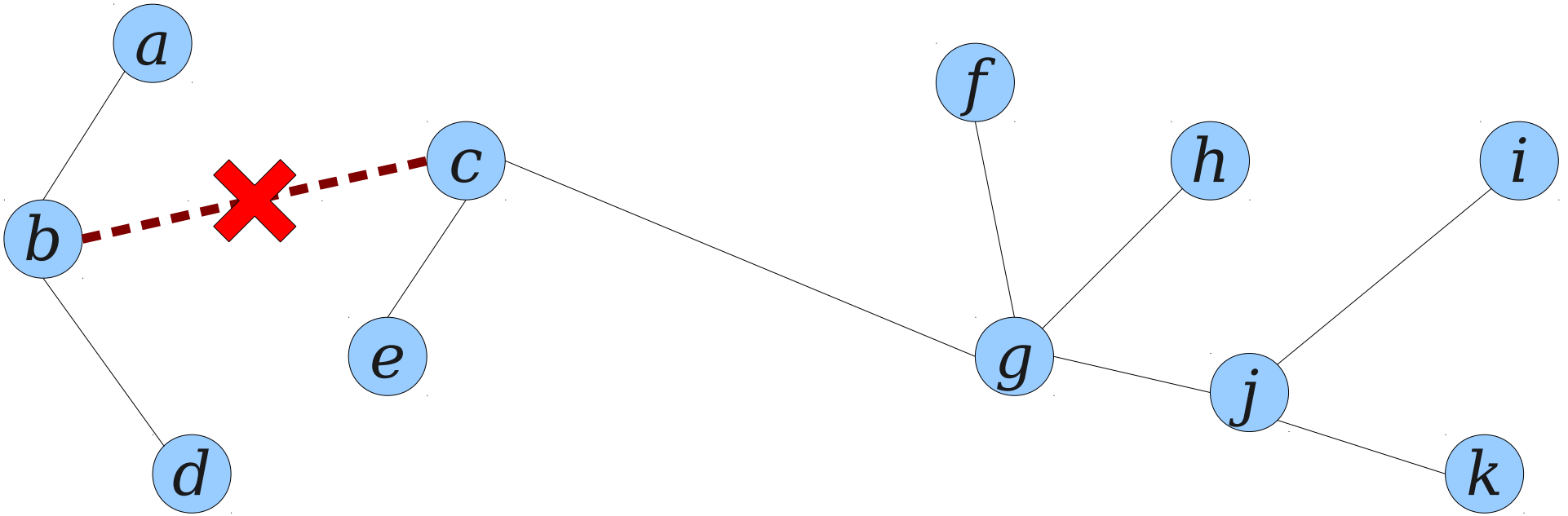
c e c b a b d b c g j k j i j g h g f g c

Euler Tours and Dynamic Trees

- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link*(*u*, *v*)** links the trees together by adding edge $\{u, v\}$.
- To link T_1 and T_2 by adding $\{u, v\}$:
  - Let E_1 and E_2 be Euler tours of T_1 and T_2, respectively.
  - Rotate E_1 to root the tour at u.
  - Rotate E_2 to root the tour at v.
  - Concatenate E_1, E_2, $\{u\}$.
- **This is the same as before.**

Euler Tours and Dynamic Trees

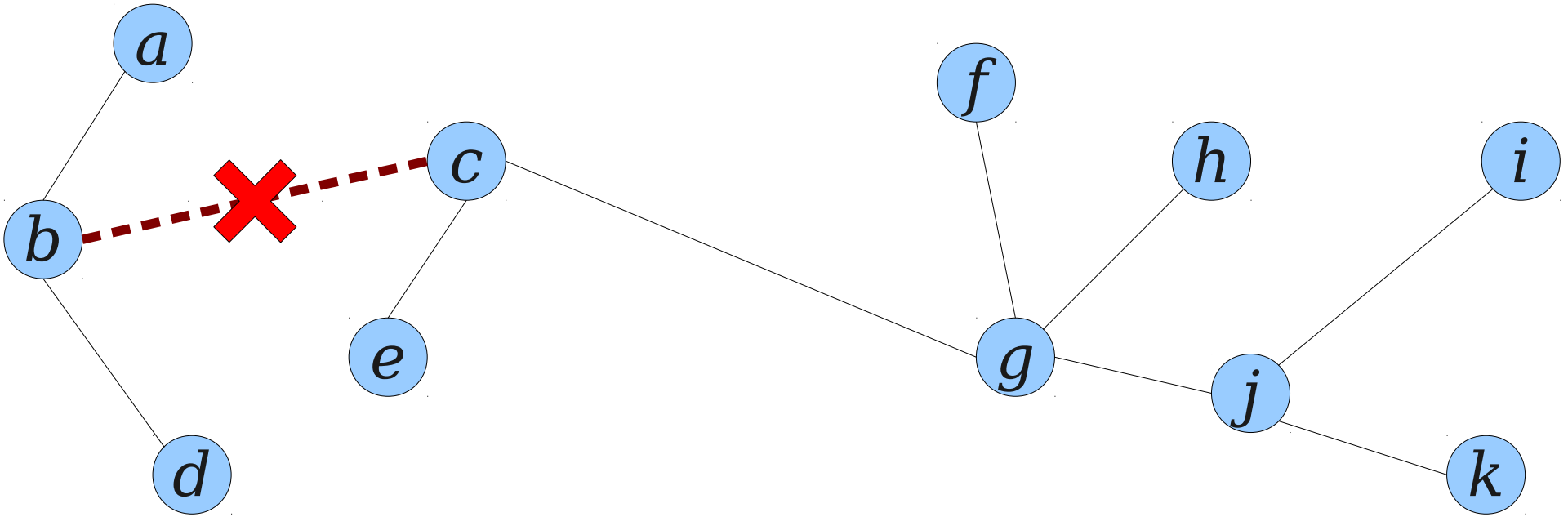
- Given a tree T, executing **cut(u, v)** cuts the edge $\{u, v\}$ from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



c e c b a b d b c g j k j i j g h g f g c

Euler Tours and Dynamic Trees

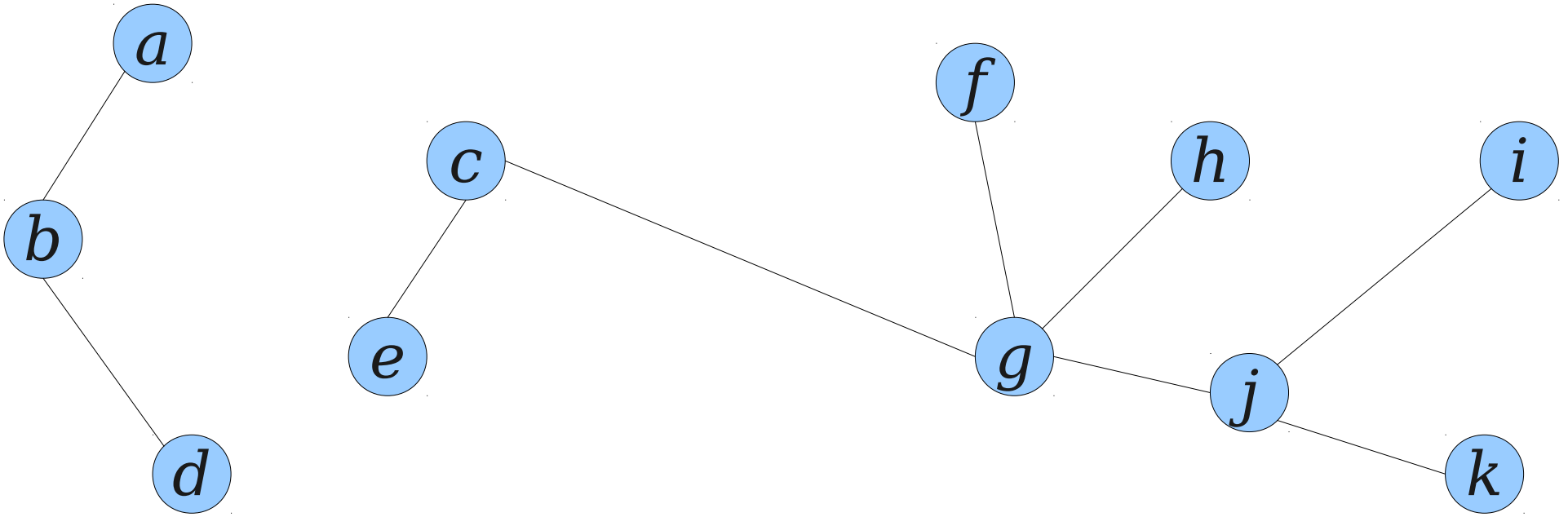
- Given a tree T, executing **cut(u, v)** cuts the edge $\{u, v\}$ from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



c e c b a b d b c g j k j i j g h g f g c

Euler Tours and Dynamic Trees

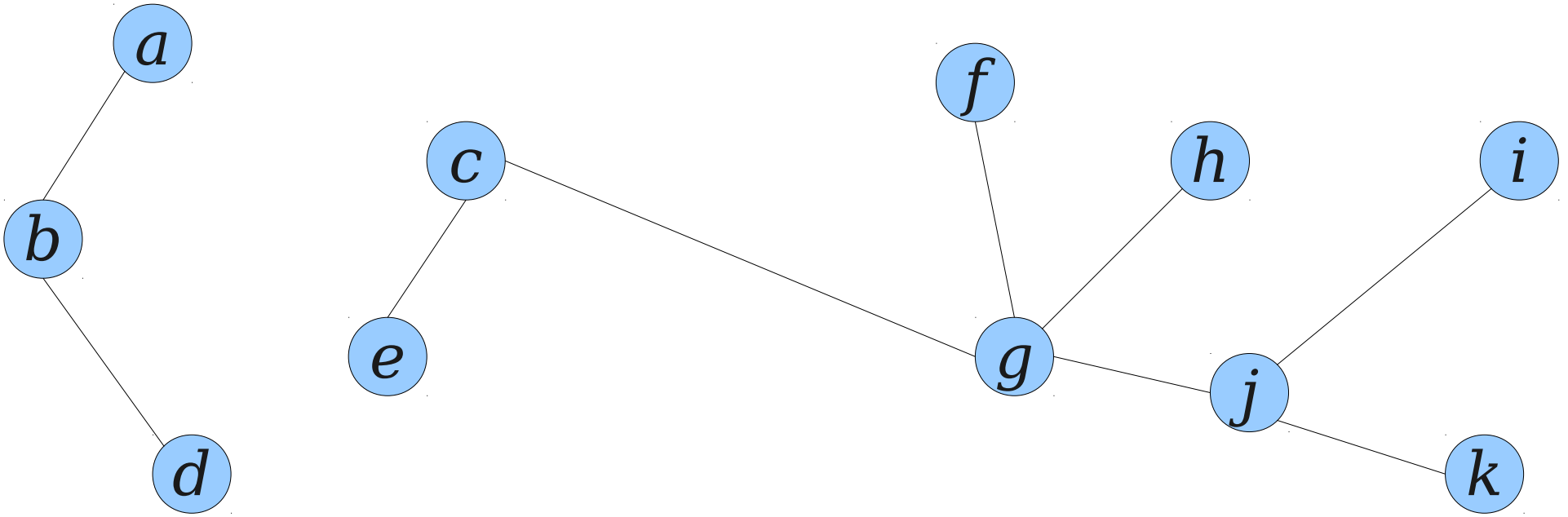
- Given a tree T, executing **cut(u, v)** cuts the edge $\{u, v\}$ from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



c e c **b a b d b** **c g j k j i j g h g f g c**

Euler Tours and Dynamic Trees

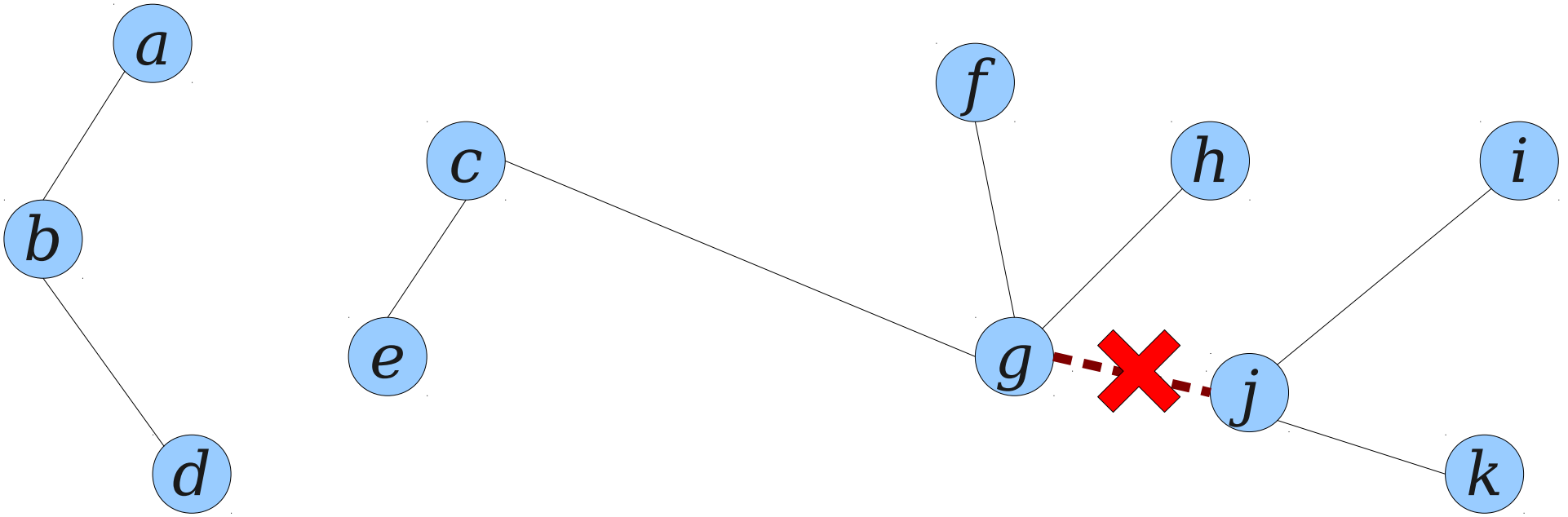
- Given a tree T, executing **cut(u, v)** cuts the edge $\{u, v\}$ from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



b a b d b **c e c g j k j i j g h g f g c**

Euler Tours and Dynamic Trees

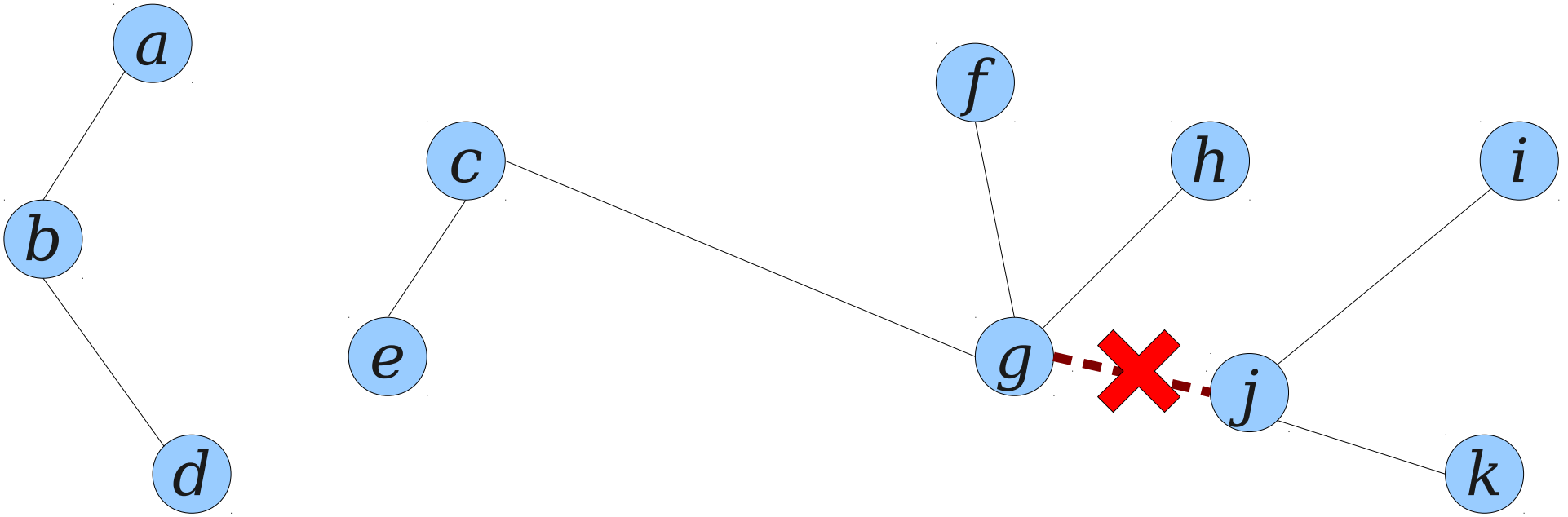
- Given a tree T, executing **cut(u, v)** cuts the edge $\{u, v\}$ from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



b a b d b ***c e c g j k j i j g h g f g c***

Euler Tours and Dynamic Trees

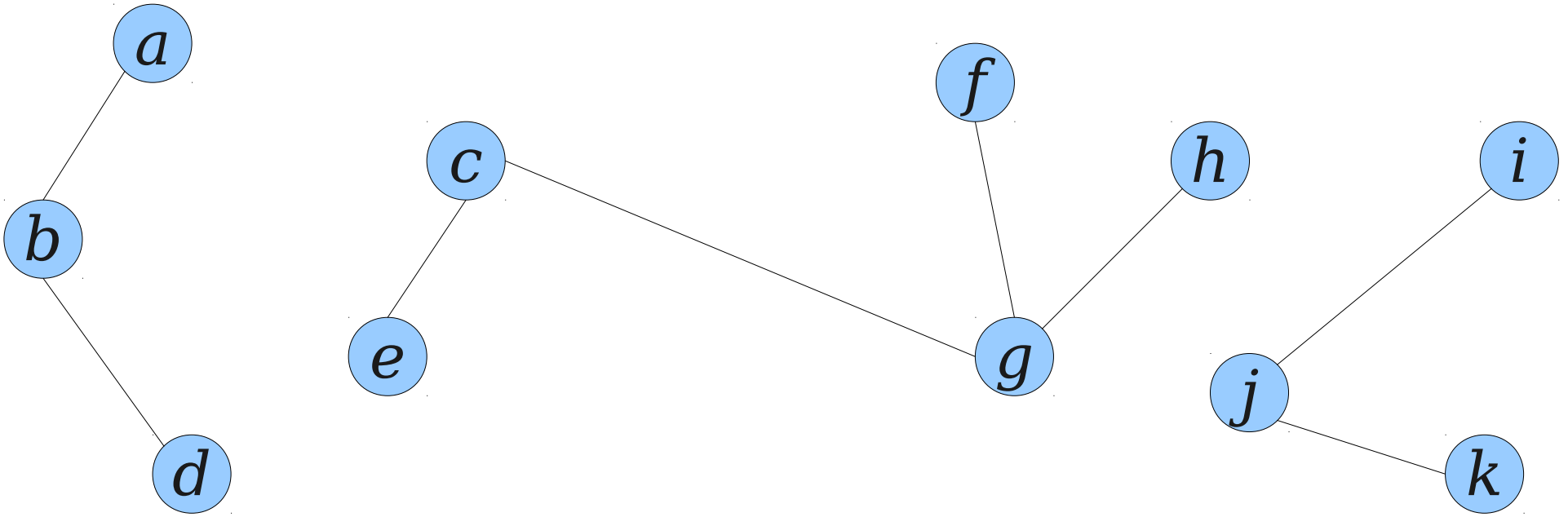
- Given a tree T, executing **cut(u, v)** cuts the edge $\{u, v\}$ from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



b a b d b ***c e c g j k j i j g h g f g c***

Euler Tours and Dynamic Trees

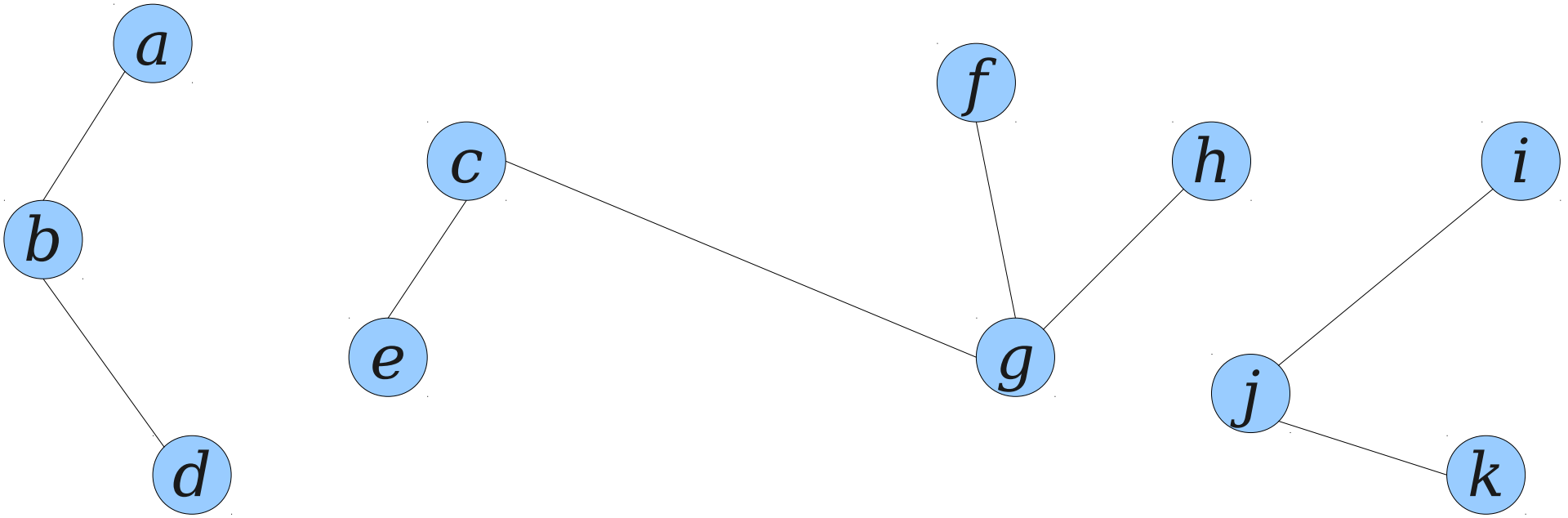
- Given a tree T, executing **cut(u, v)** cuts the edge $\{u, v\}$ from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



b a b d b ***c e c g*** ***j k j i j*** ***g h g f g c***

Euler Tours and Dynamic Trees

- Given a tree T, executing **cut(u, v)** cuts the edge $\{u, v\}$ from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



b a b d b ***c e c g h g f g c*** ***j k j i j***

Euler Tours and Dynamic Trees

- Given a tree T, executing **cut(u, v)** cuts the edge $\{u, v\}$ from the tree (assuming it exists).
- To cut T into T_1 and T_2 by cutting $\{u, v\}$:
  - Let E be an Euler tour for T.
  - Split E at (u, v) and (v, u) to get J, K, L, in that order.
  - Delete the last entry of J.
  - Then $E_1 = K$.
  - Then $E_2 = J, L$
- **No longer necessary to store the full range of u and v.**
- **Now need to store pointers from the edges to the spots where the nodes appear in the trees.**

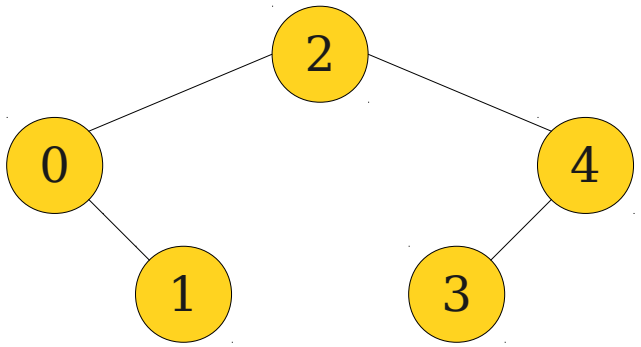
Euler Tour Trees

- The data structure:
  - Represent each tree as an Euler tour.
  - Store those sequences as balanced binary trees.
  - Each node in the original forest stores a pointer to some arbitrary occurrence of that node.
  - Each edge in the original forest stores pointers to the nodes appearing when that edge is visited.
    - Can store these edges in a balanced BST.
  - Each node in the balanced trees stores a pointer to its parent.
- *link*, *cut*, and *is-connected* queries take time only **O(log *n*)** each.

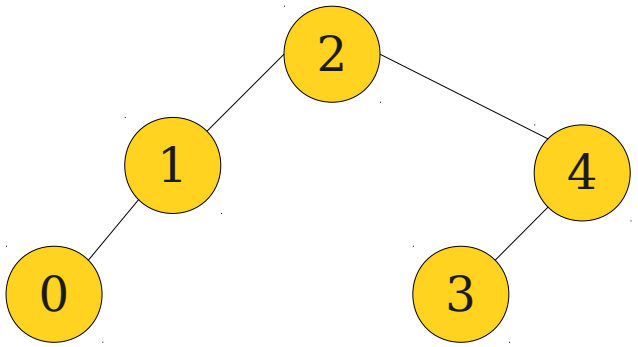
Cartesian Trees Revisited

Cartesian Trees

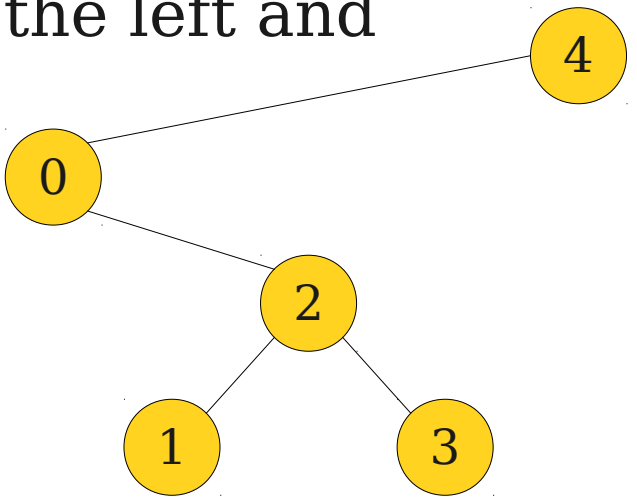
- A **Cartesian tree** is a binary tree derived from an array and defined as follows:
  - The empty array has an empty Cartesian tree.
  - For a nonempty array, the root stores the index of the minimum value. Its left and right children are Cartesian trees for the subarrays to the left and right of the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



261	268	161	167	166



6	5	3	9	7



14	55	22	43	11

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, …, 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

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

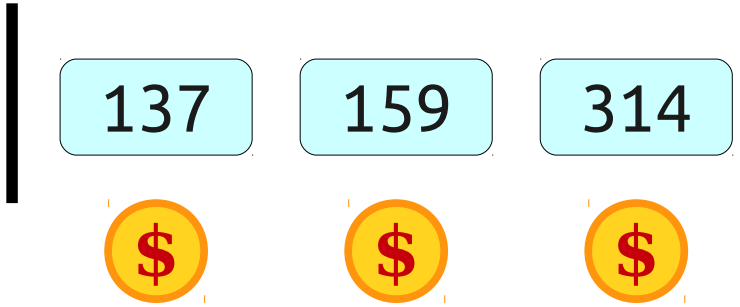
The Banker's Method

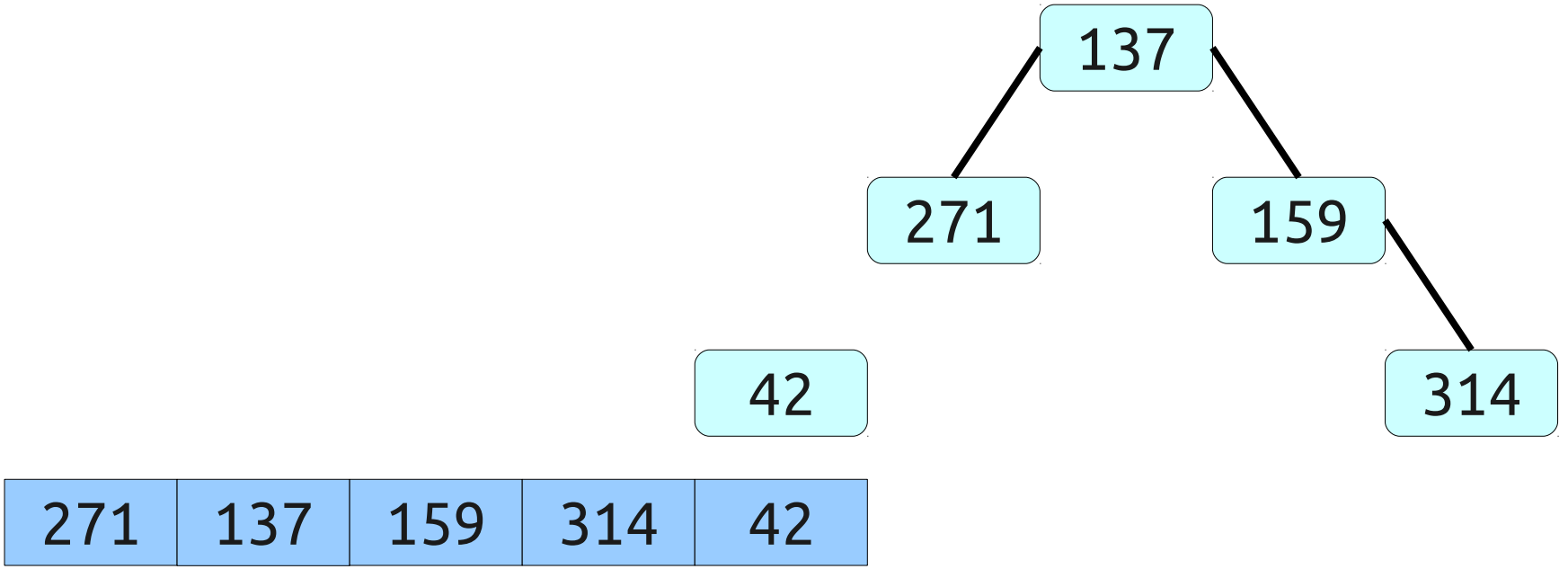
- If we never spend credits we don't have:

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

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

Constructing Cartesian Trees





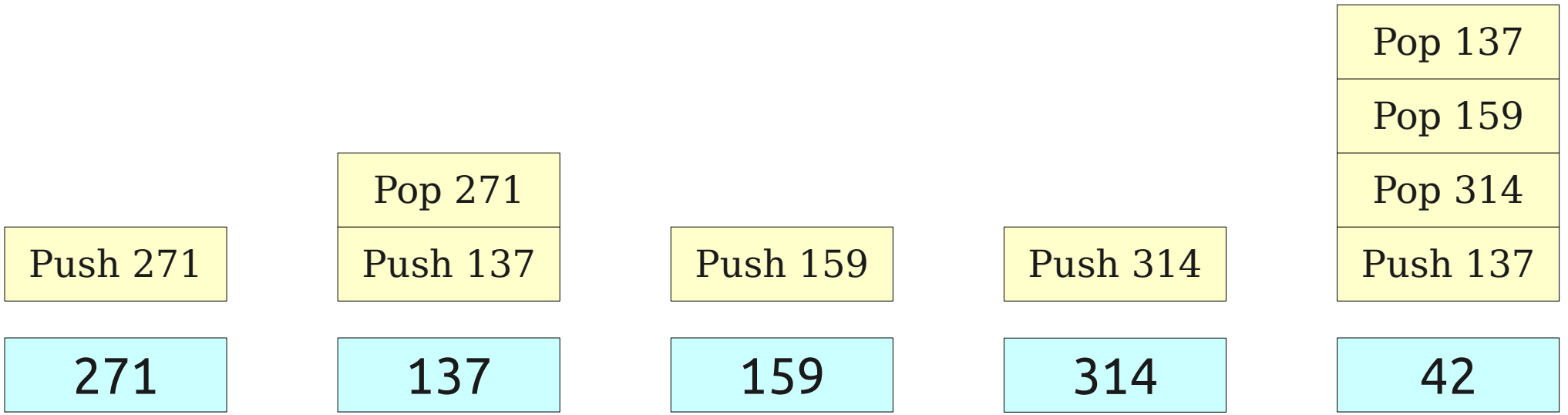
The Banker's Method

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

$$t(op) + O(1) \cdot (added_i - removed_i)$$
$$= 1 + k + O(1) \cdot (1 - k)$$
$$= 1 + k + 1 - k$$
$$= 2$$
$$= \mathbf{O(1)}$$

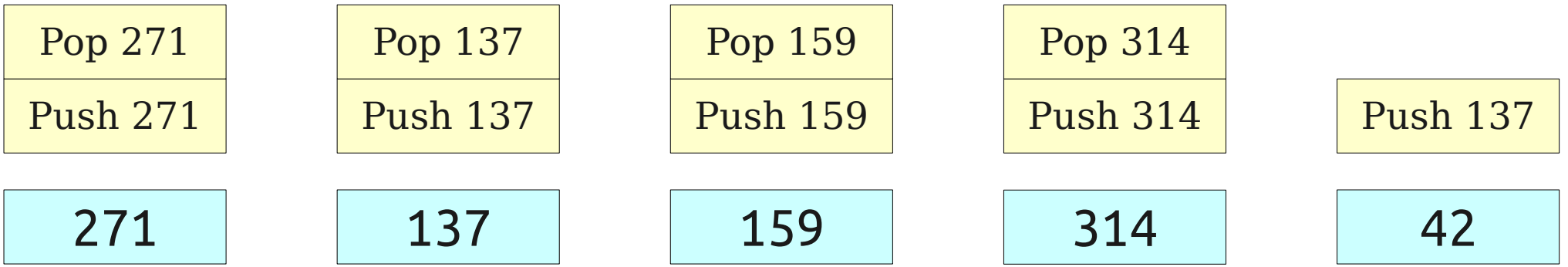
- Each insertion has amortized cost O(1).
- Any n insertions will take time $O(n)$.

Intuiting the Banker's Method

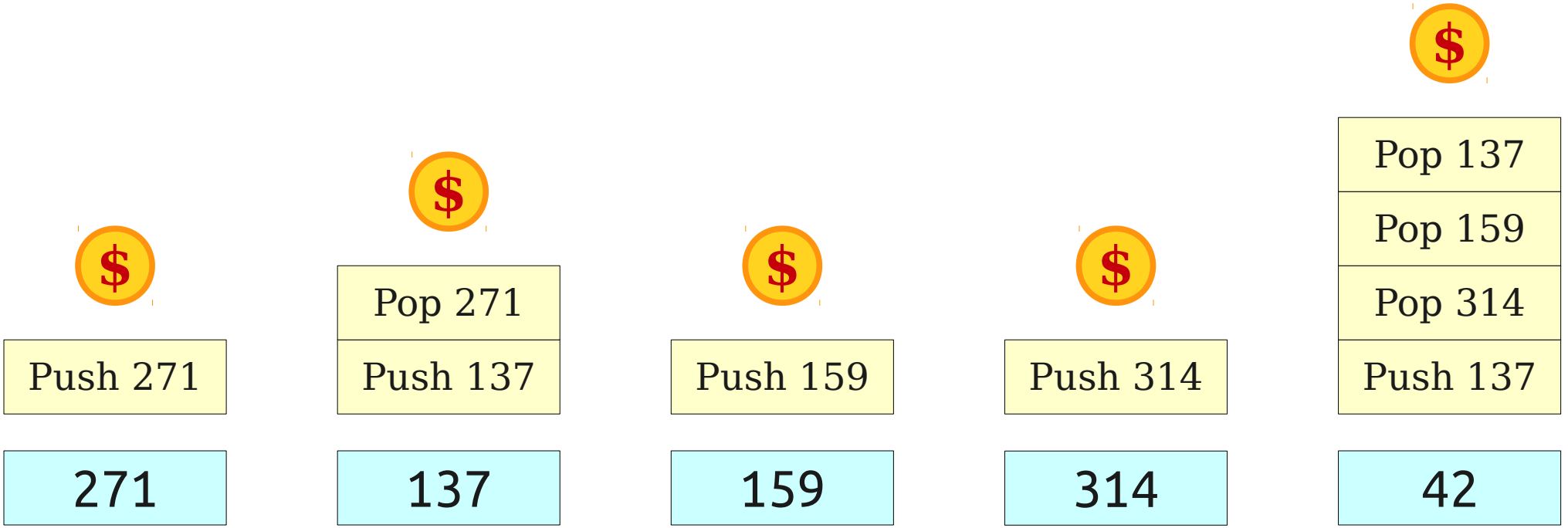


Intuiting the Banker's Method

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

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.

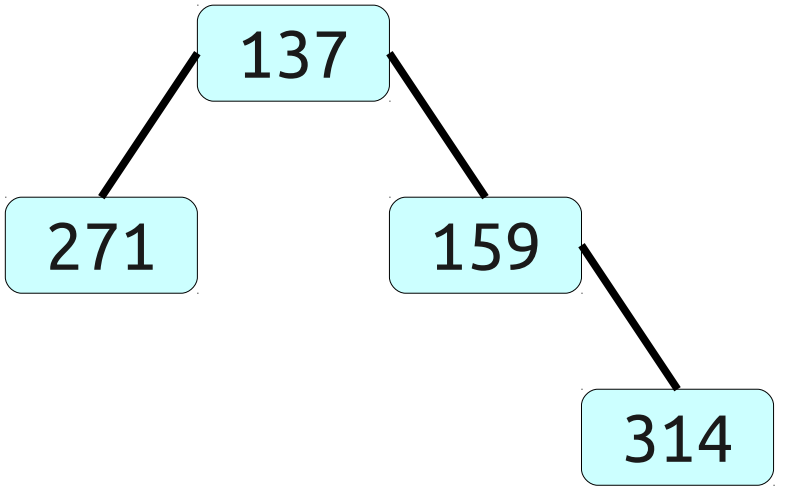
The Potential Method

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

- 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

$\Phi = 3$ | 137 | 159 | 314 |



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).

Time-Out for Announcements!

Problem Set Three

- Problem Set Three goes out today. It's due next Wednesday at 2:15PM.
- Explore amortized analysis and a different data structure for dynamic connectivity in trees!
- Keith will be holding office hours in Gates 178 tomorrow from 2PM – 4PM.

Your Questions

“Is it possible to please get an explanation of where we lost points in the homeworks and a sample solution set?”

Of course! We'll release hardcopy solutions for the problem sets in lecture. You're welcome to stop by office hours to ask questions about the problem sets.

“How can we access the grades specifically for our code submissions?”

You should have received an email with your grade and feedback from your grader. If you didn't, let me know ASAP!

“Could we use a Piazza forum for the class? It would make answering a lot of questions, both practical and conceptual, much easier. Thanks!”

I'll think about this. In the meantime, feel free to email the staff list with questions!

cs166-spr1314-staff@lists.stanford.edu

A Note on Questions

Back to CS166!

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.

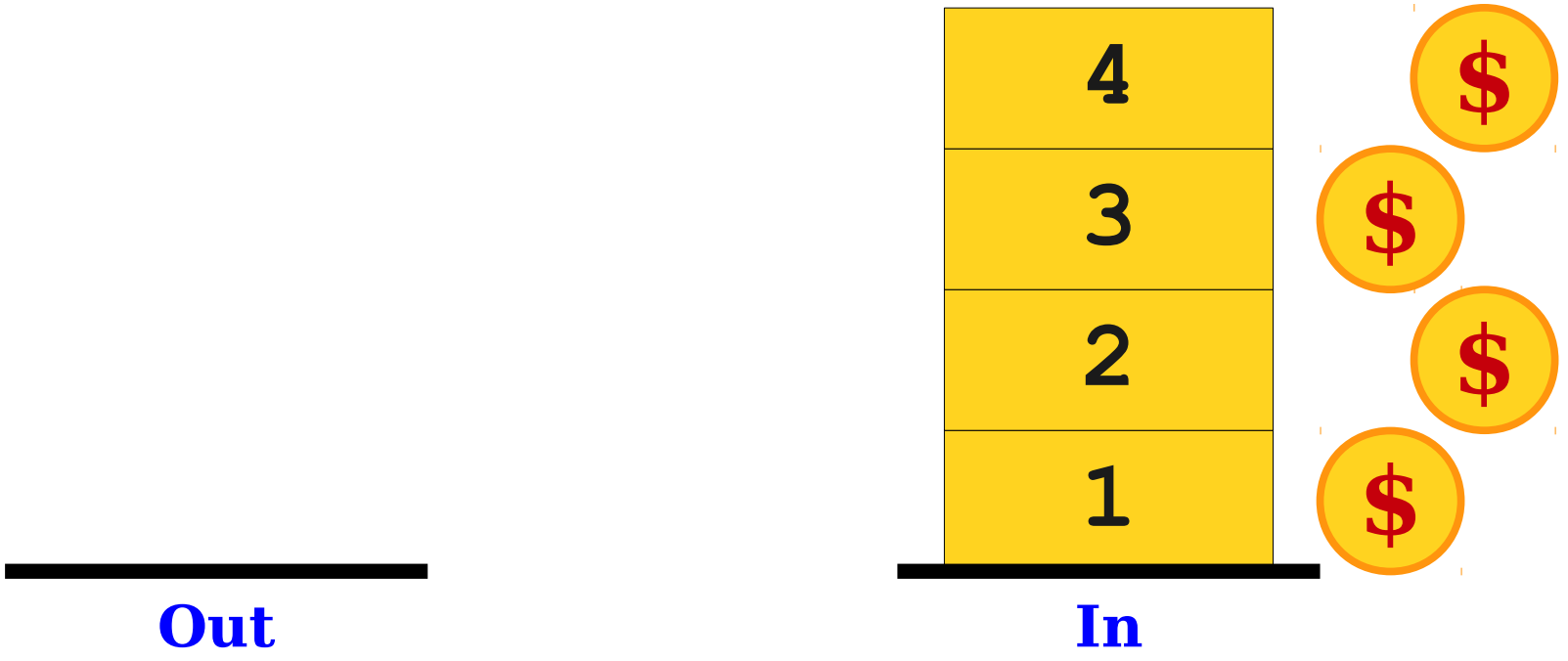
An Aggregate Analysis

- **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)**.

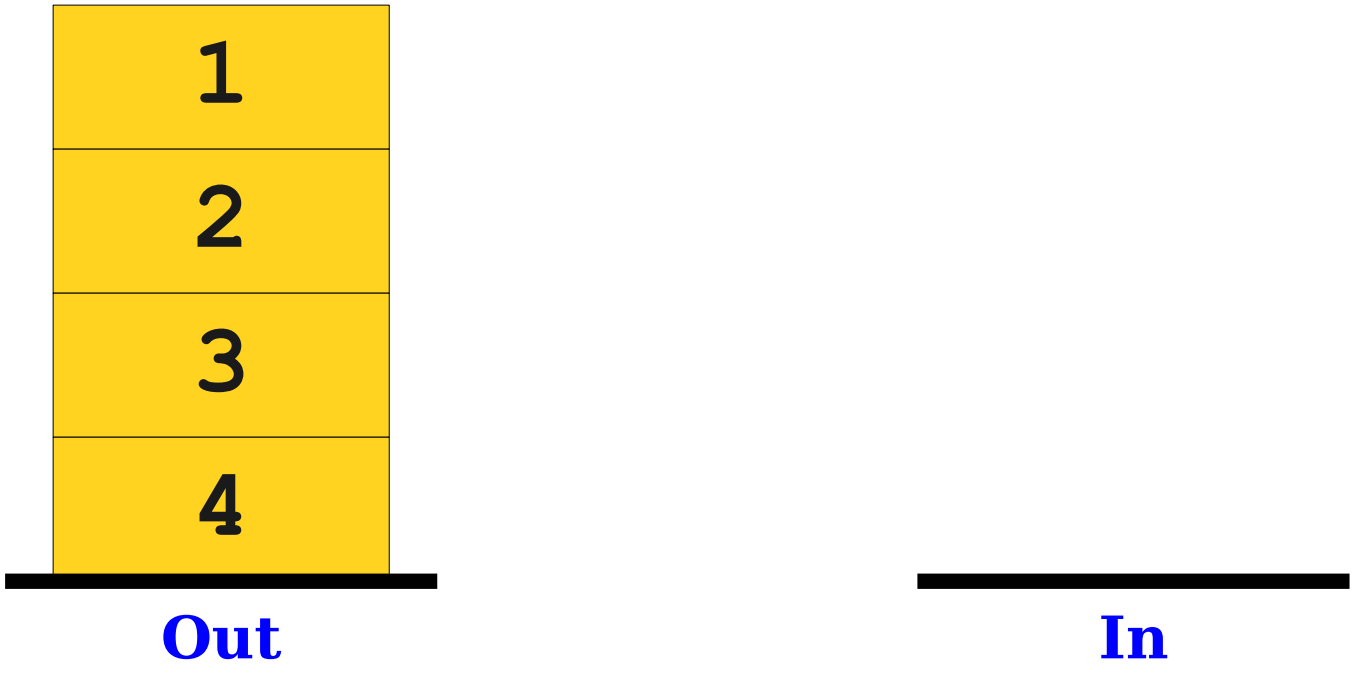
The Banker's Method

- 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)**.

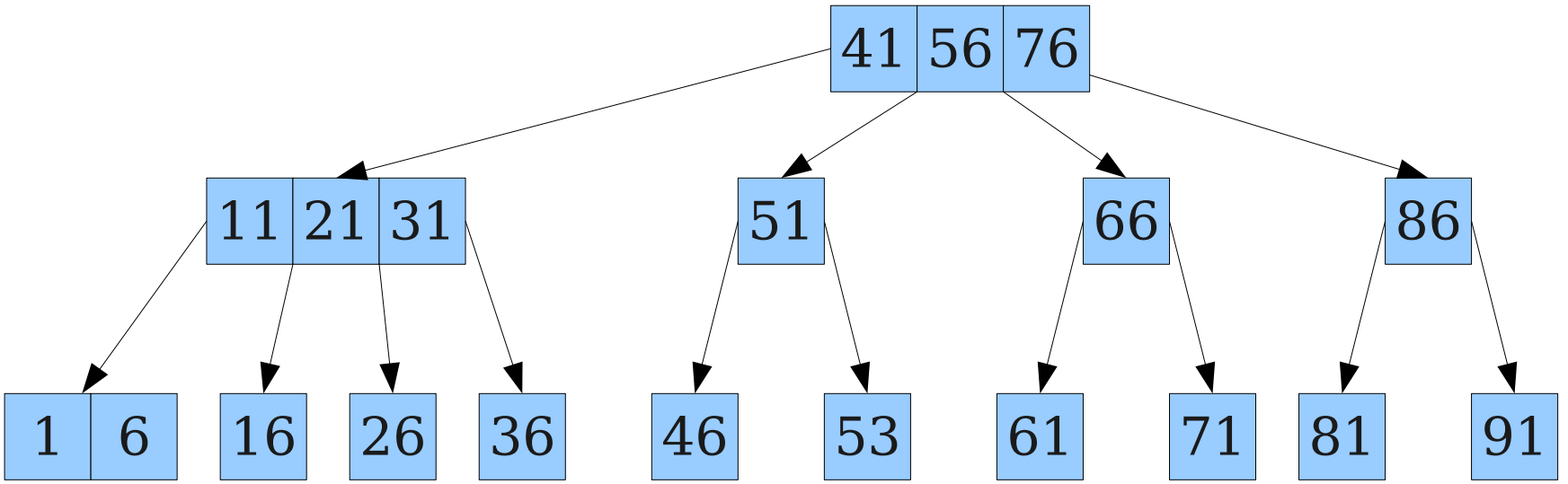
Another Examples: **2-3-4 Trees**

2-3-4 Trees

- Inserting or deleting values from a 2-3-4 trees takes time O(log n).
- Why is that?
  - Some amount of work finding the insertion or deletion point, which is Θ(log n).
  - Some amount of work “fixing up” the tree by doing insertions or deletions.
- How much is that second amount?

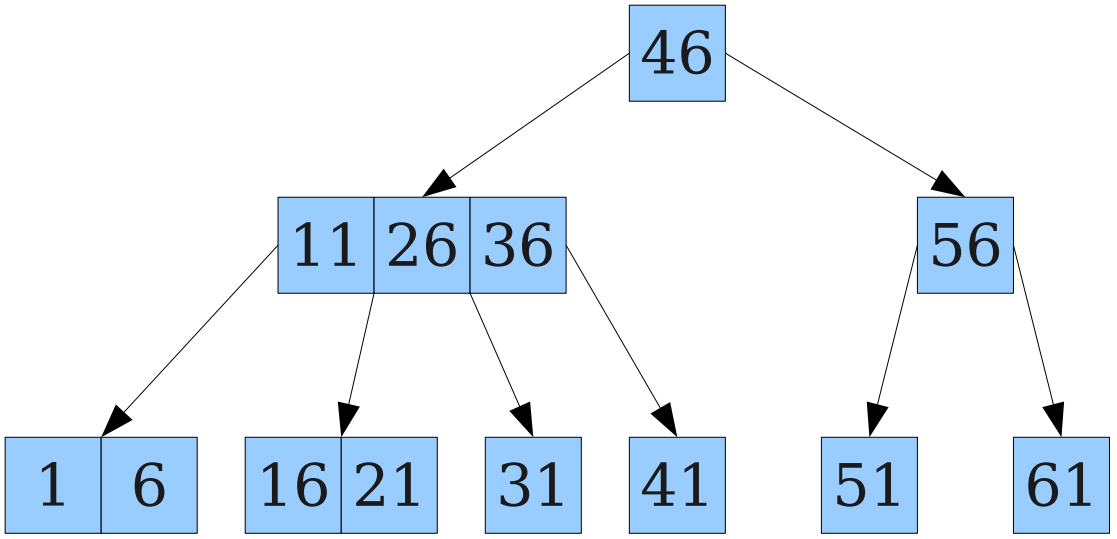
2-3-4 Tree Insertions

- 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 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.

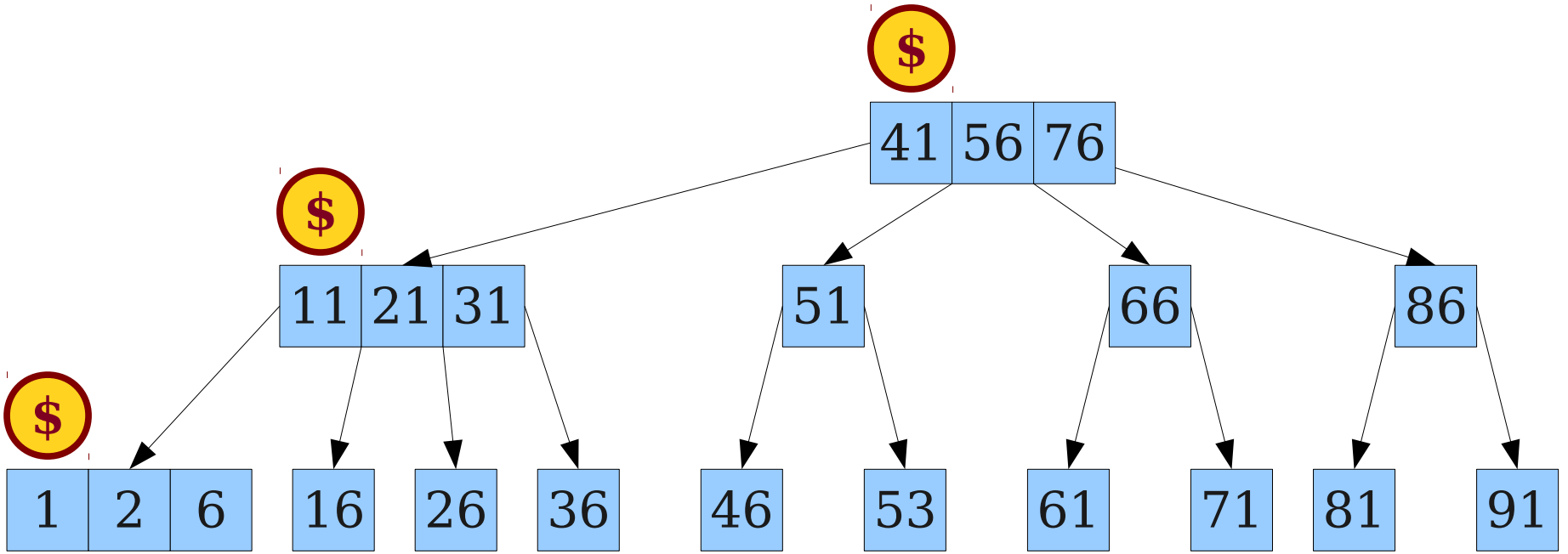


2-3-4 Tree Fixup

- **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).

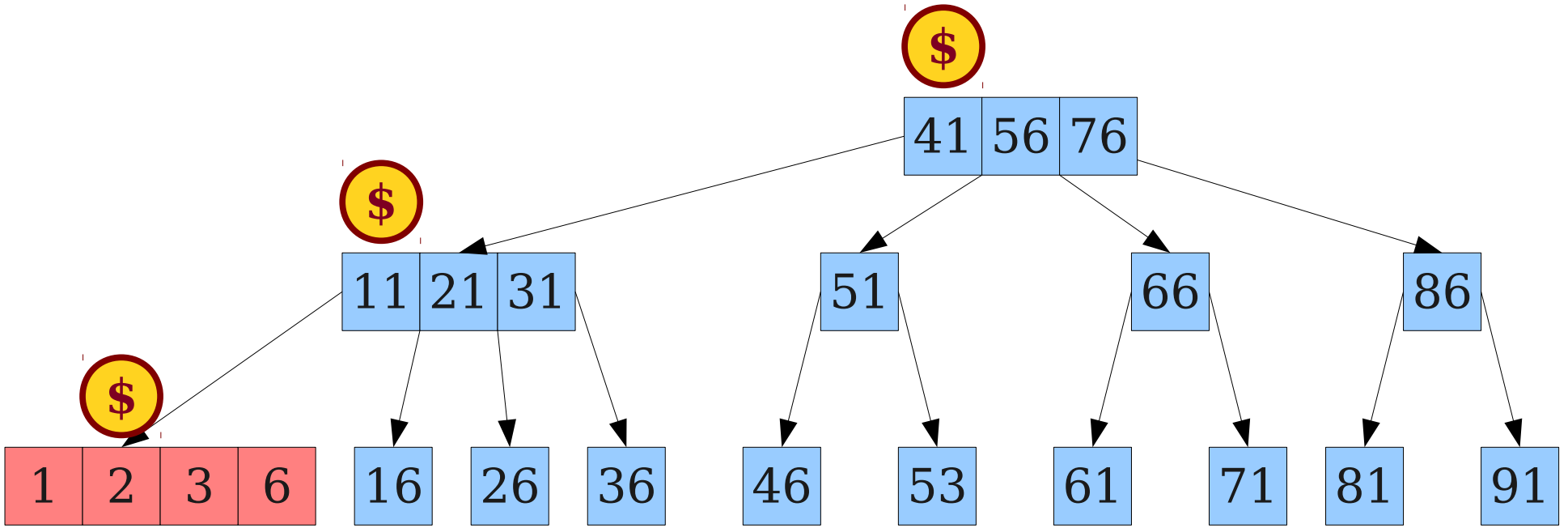
2-3-4 Tree Insertions

- 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.



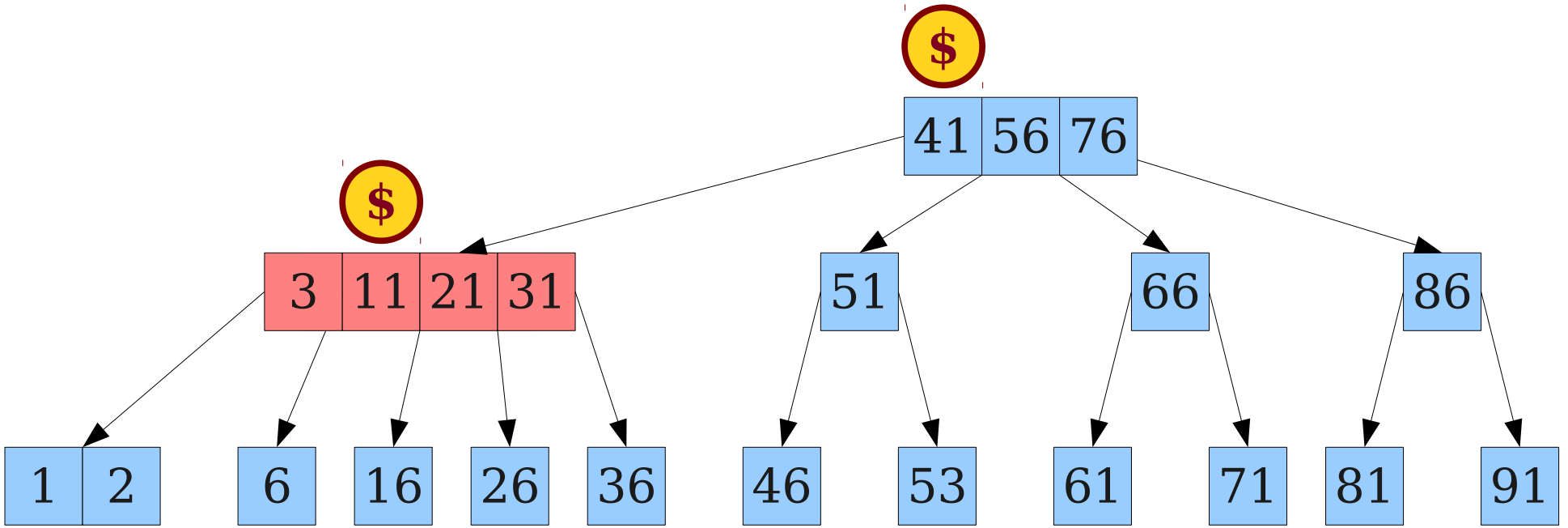
2-3-4 Tree Insertions

- 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.



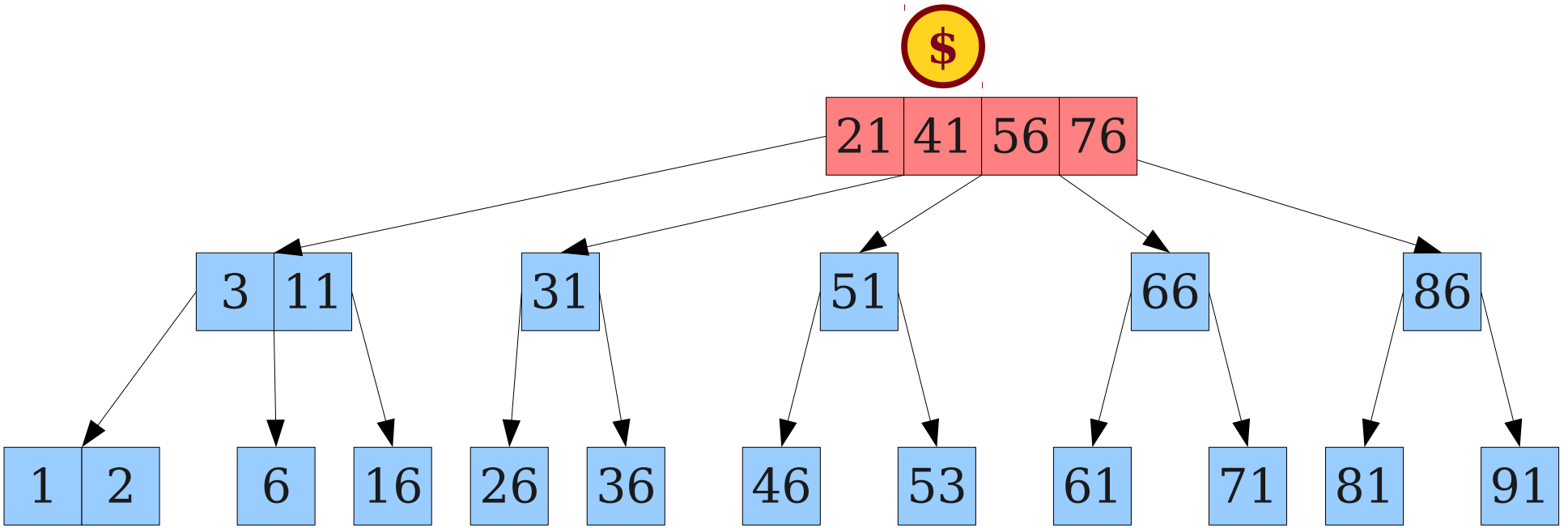
2-3-4 Tree Insertions

- 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.



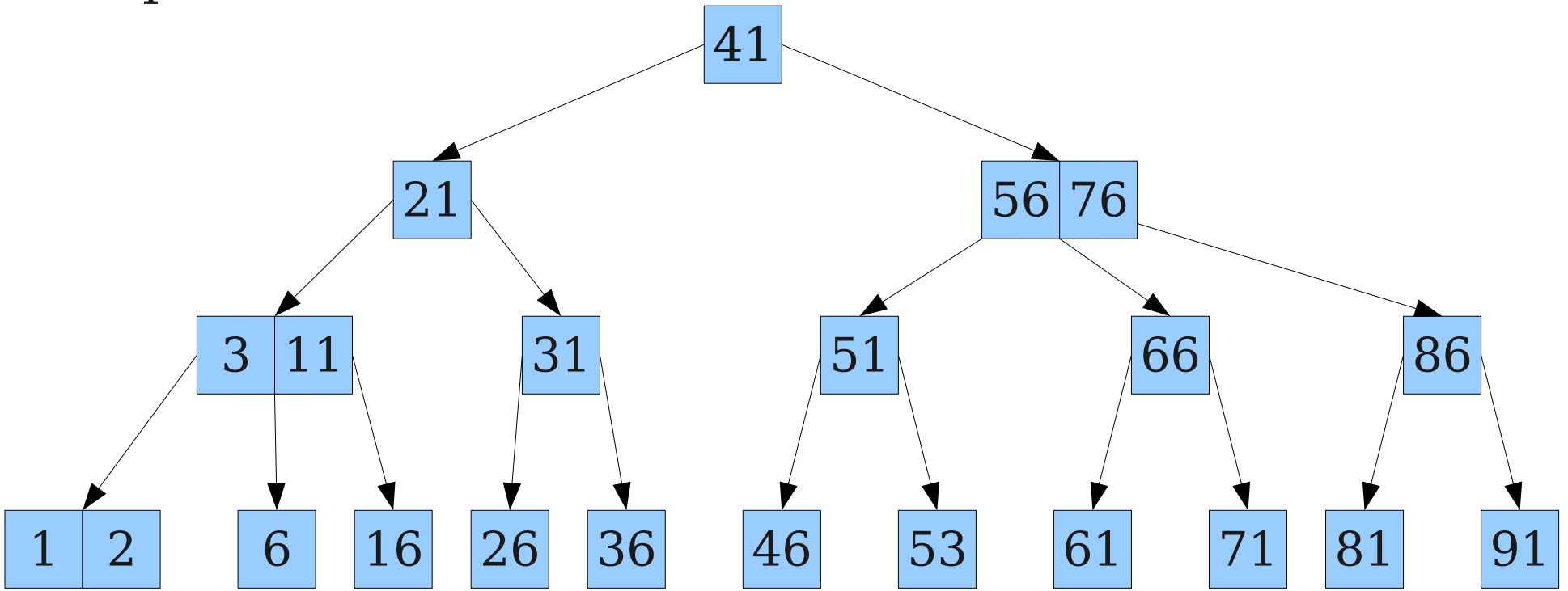
2-3-4 Tree Insertions

- 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.

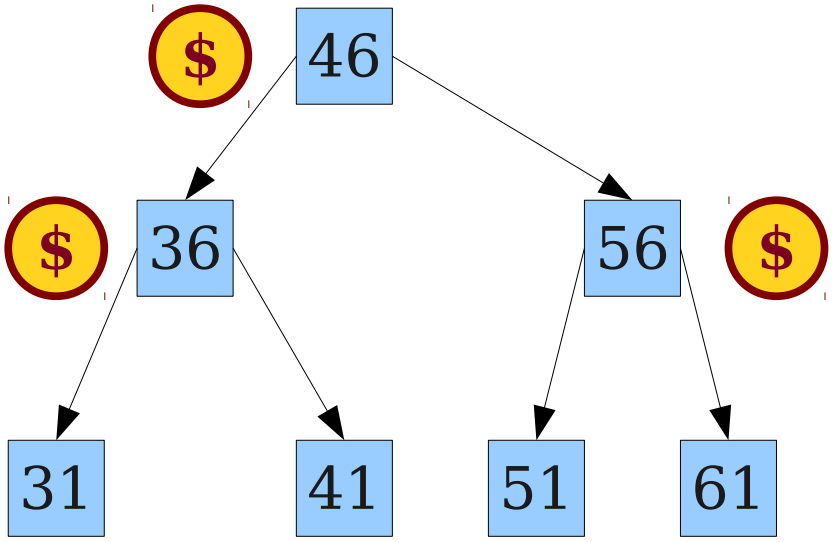


2-3-4 Tree Insertions

- 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.

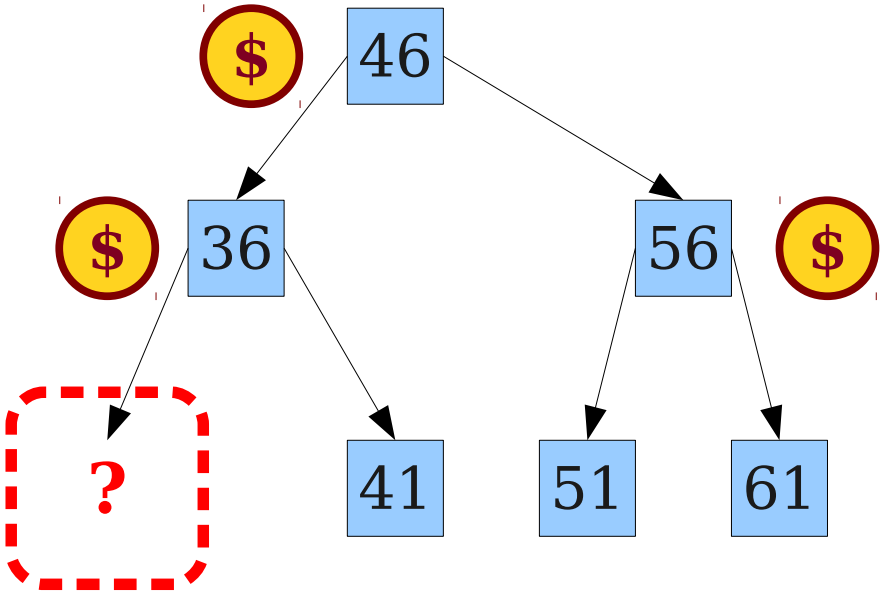
2-3-4 Tree Deletions

- 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.”



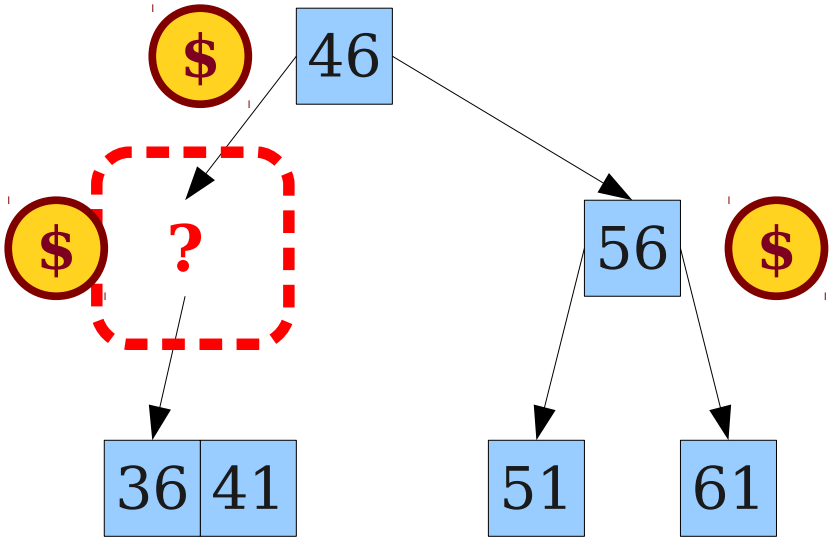
2-3-4 Tree Deletions

- 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.”



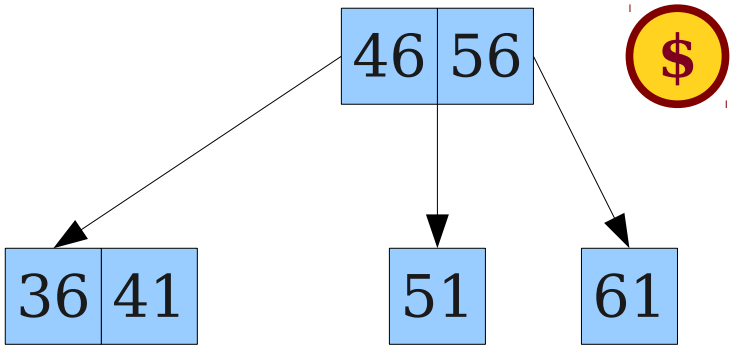
2-3-4 Tree Deletions

- 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.”



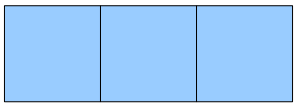
2-3-4 Tree Deletions

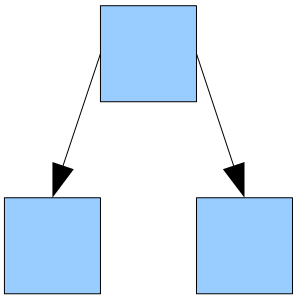
- Using the banker's method, we get that pure deletions have O(1) amortized fixup work.
- Could also do this using the potential method.
  - Define Φ to be the number of 2-nodes with two 2-node children (call these “tiny triangles.”)
  - Each “light” deletion might introduce two tiny triangles: one at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

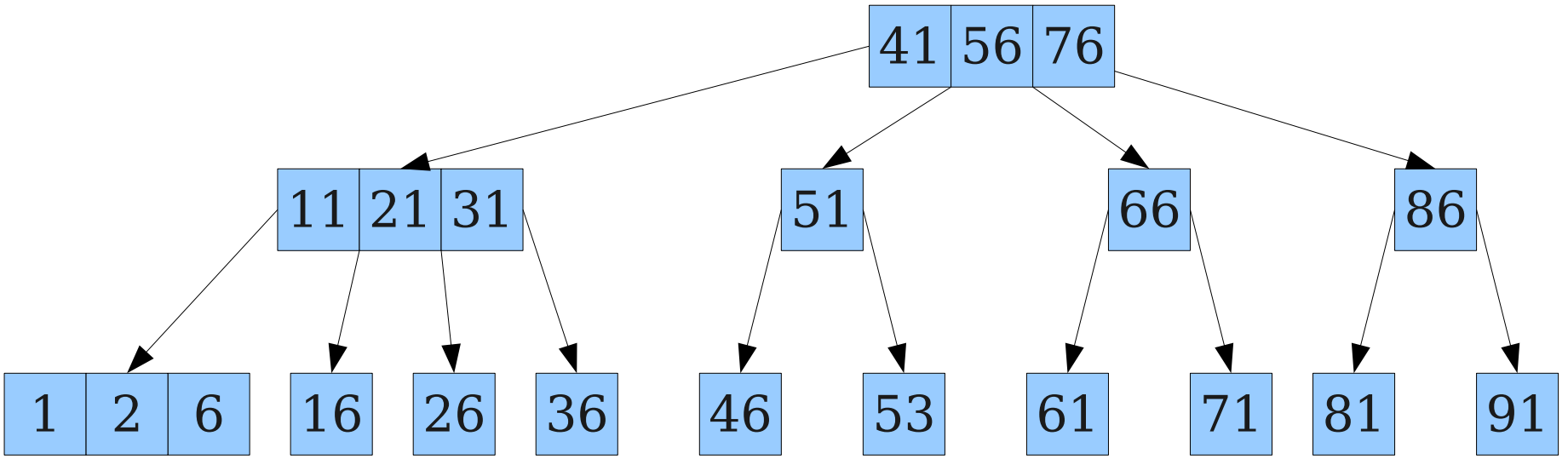
$$\Phi = \#(\quad) + \#(\quad)$$





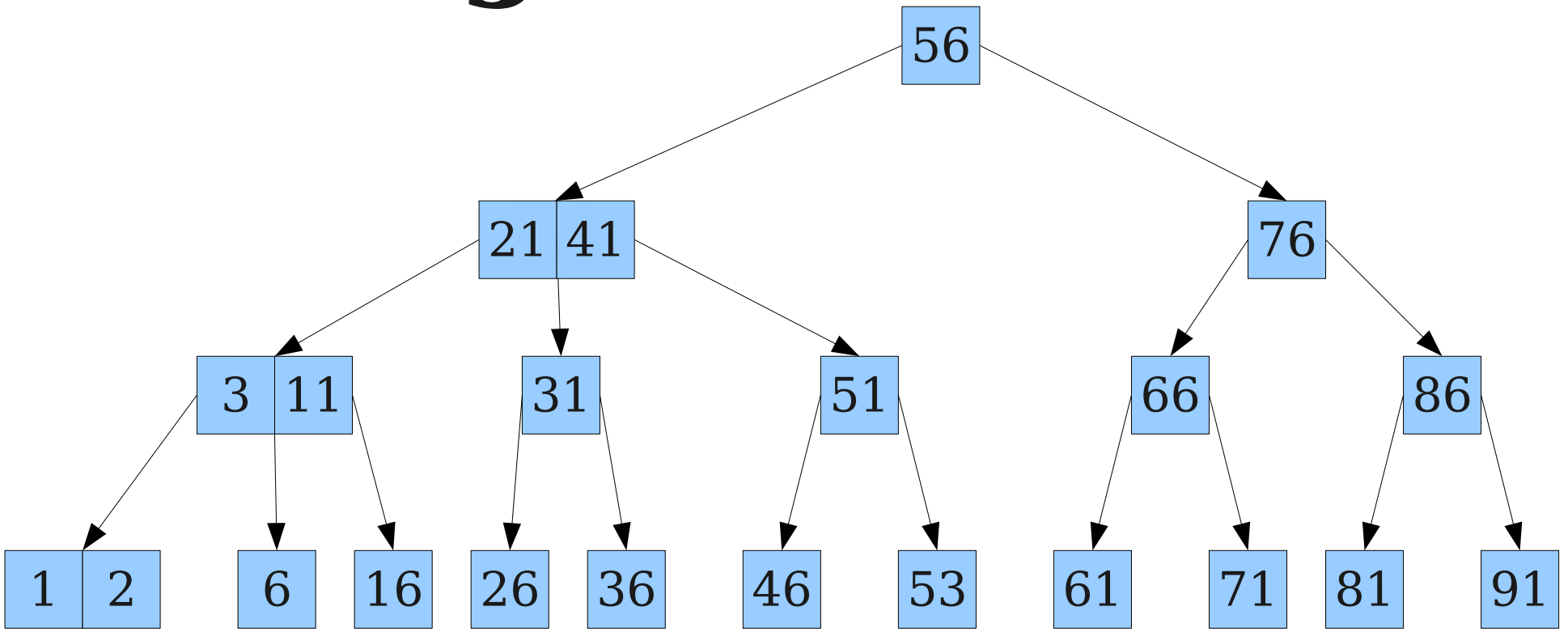
A Problem

$$\Phi = \#(\square\square\square) + \#(\text{tree}) = \mathbf{6}$$



A Problem

$$\Phi = \#(\square\square\square) + \#(\text{tree}) = 5$$



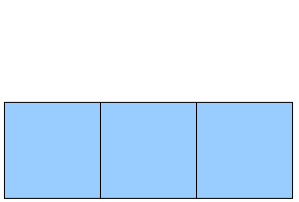
A Problem

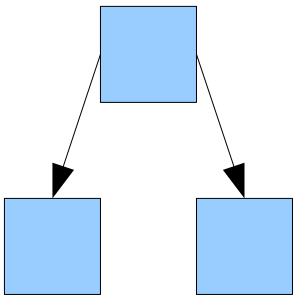
- When doing a “heavy” insertion that splits multiple 4-nodes, the resulting nodes might produce new “tiny triangles.”
- Net result: The potential only drops by one even in a long chain of splits.
- Amortized cost of the operation works out to $\Theta(\log n)$, not $O(1)$ as we hoped.

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:** Have each 4-node pay for the work to split itself and to propagate up a deletion one layer.

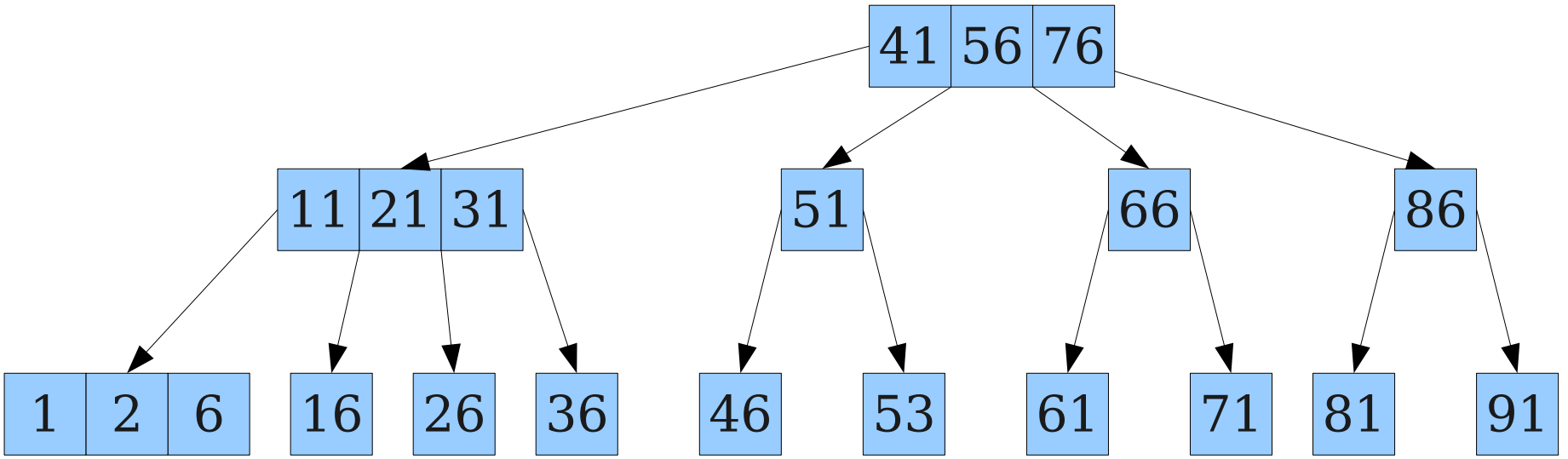
$$\Phi = 2\#(\quad) + \#(\quad)$$





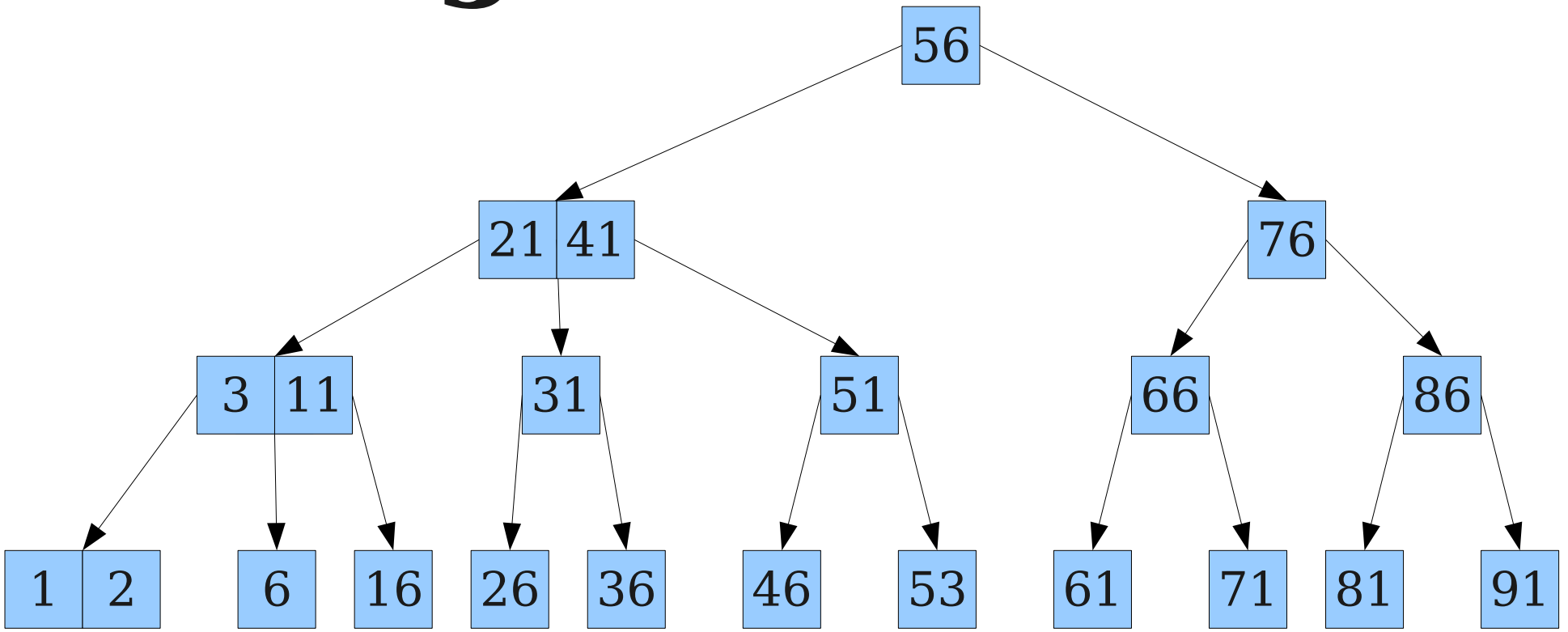
A Problem

$$\Phi = 2\#(\square\square\square) + \#(\square(\square\ \square)) = \mathbf{9}$$



A Problem

$$\Phi = 2\#(\square\square\square) + \#(\square\ \square\ \square) = 5$$



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.

Next Time

- **Binomial Heaps**
  - A simple and versatile heap data structure.
- **Fibonacci Heaps, Part One**
  - A specialized data structure for graph algorithms.