

Scapegoat Trees

Outline for Today

- ***Recap from Last Time***
 - What is amortization, again?
- ***Lazy Balanced Trees***
 - Messes are okay, up to a point.
- ***Lazy Tree Insertions***
 - Deferring updates until they're needed.
- ***Lazy Tree Deletions***
 - And some associated subtleties.

Recap from Last Time

Amortized Analysis

- We will assign amortized costs to each operation such that

$$\sum \text{amortized-cost} \geq \sum \text{real-cost}$$

- To do so, define a **potential function** Φ such that

- Φ measures how “messy” the data structure is,
- $\Phi_{start} = 0$, and
- $\Phi \geq 0$.

- Then, define amortized costs of operations as

$$\text{amortized-cost} = \text{real-cost} + k \cdot \Delta\Phi$$

for a choice of k under our control.

- Intuitively:

- If an operation makes a mess that needs to be cleaned up later, its amortized cost will be higher than its original cost.
- If an operation cleans up a mess, its amortized cost will be lower than its real cost.

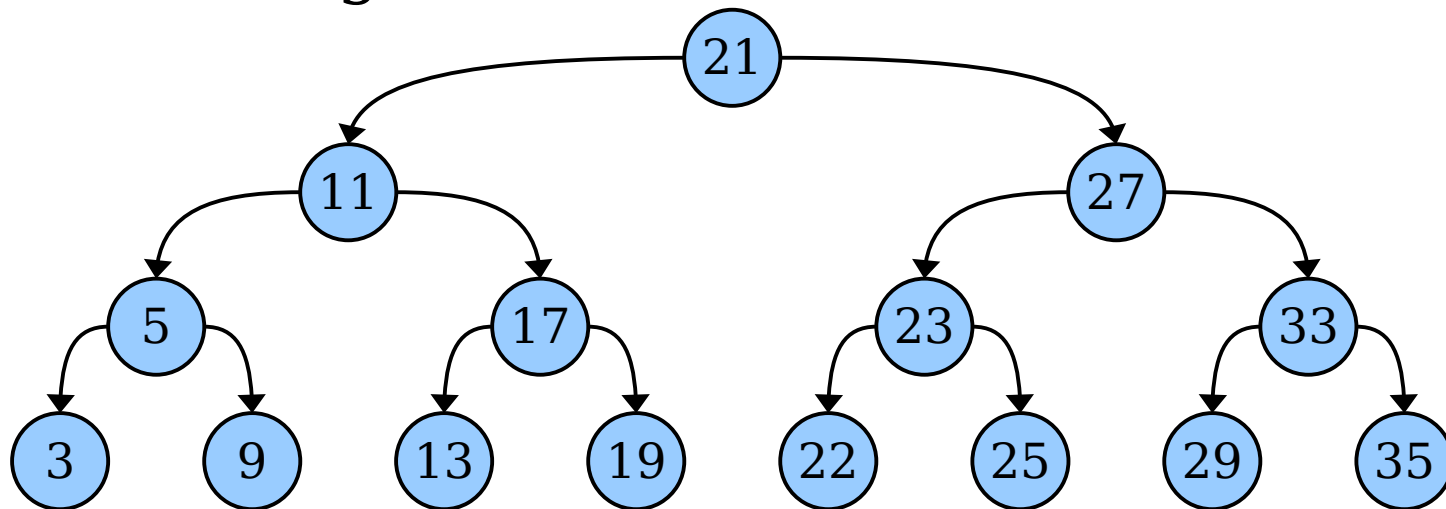
New Stuff!

Balanced Trees

- The red/black trees we explored earlier are worst-case efficient and guaranteed to have a height of $O(\log n)$.
- However, explaining how they work and deriving the basic insertion rules took two lectures – and we still didn't finish covering all cases.
- ***Goal for today:*** Find a simpler way to keep a tree balanced, under the assumption we're okay with amortized-efficient rather than worst-case efficient lookups.

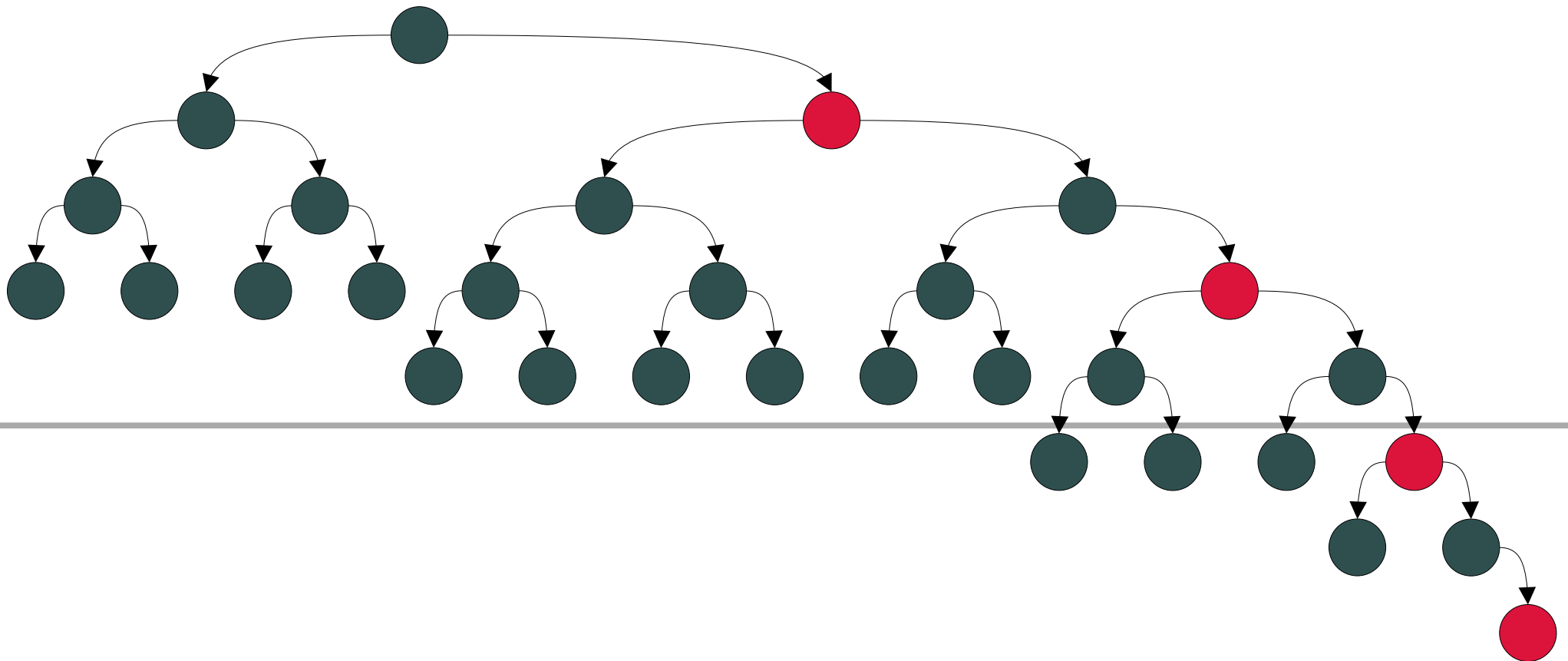
On $O(\log n)$ Height

- A perfectly-balanced binary search tree with $n > 0$ nodes has height at most $\lg n$.
 - ($\lg n$ denotes $\log_2 n$.)
- However, this tree shape is difficult to maintain: a single insertion or deletion might require a lot of node reshuffling.



On $O(\log n)$ Height

- To speed up logic after insertions or deletions, most balanced BSTs only guarantee height of multiple of $\lg n$.
- For example, red/black trees have height at most (roughly) $2 \lg n$ in the worst case.

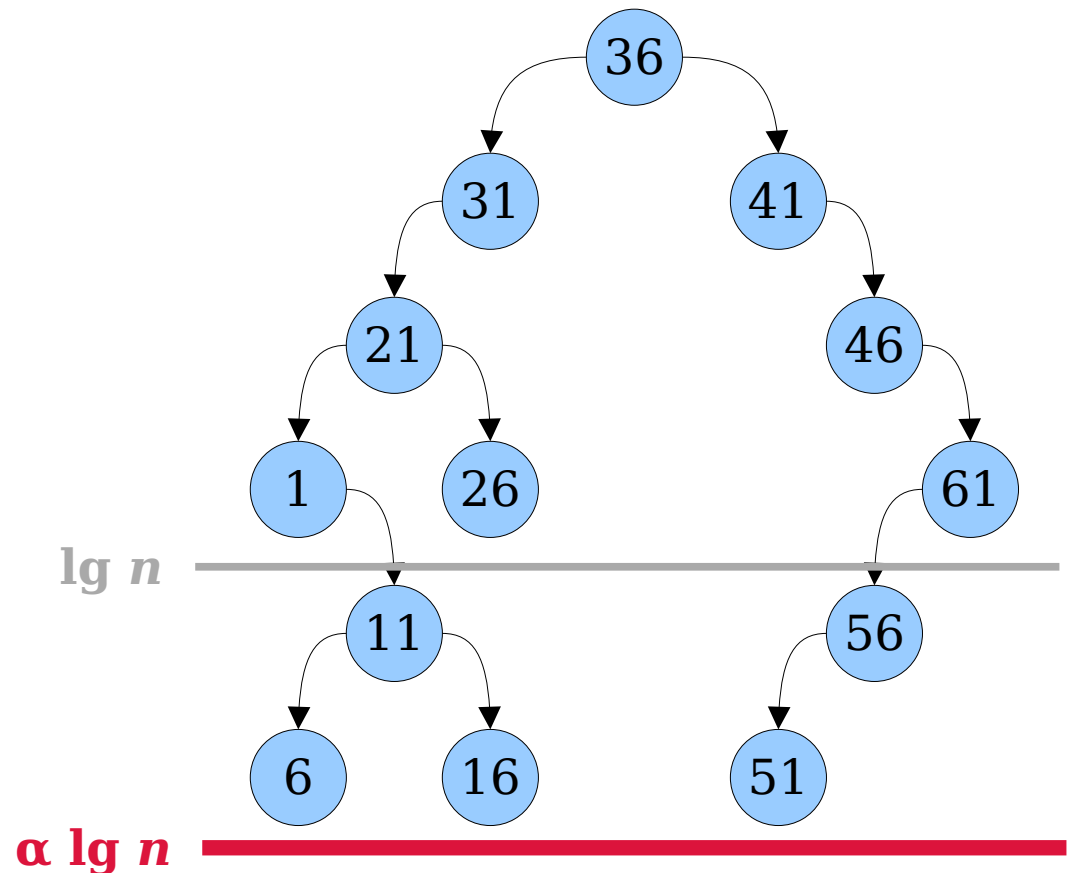


On $O(\log n)$ Height

- We're already comfortable with trees whose heights are $\alpha \lg n$ for some $\alpha > 1$.
- **Question:** Can we design a balanced tree purely based on this restriction, without any other structural constraints?

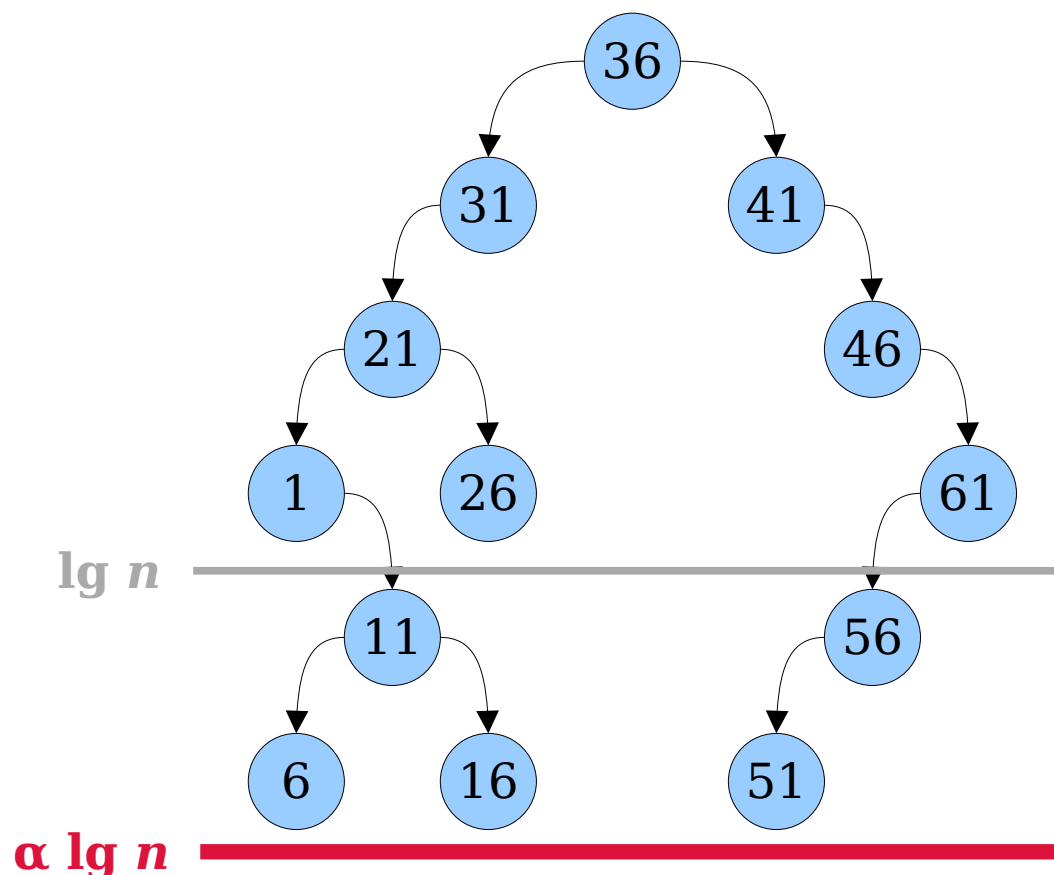
Adding Slack Space

- Pick a fixed constant $\alpha > 1$.
- Set the maximum height on our tree to $\alpha \lg n$.
- As long as we don't exceed this maximum height, all operations on our BST will run in time $O(\log n)$, and we don't really care about the shape of the tree.



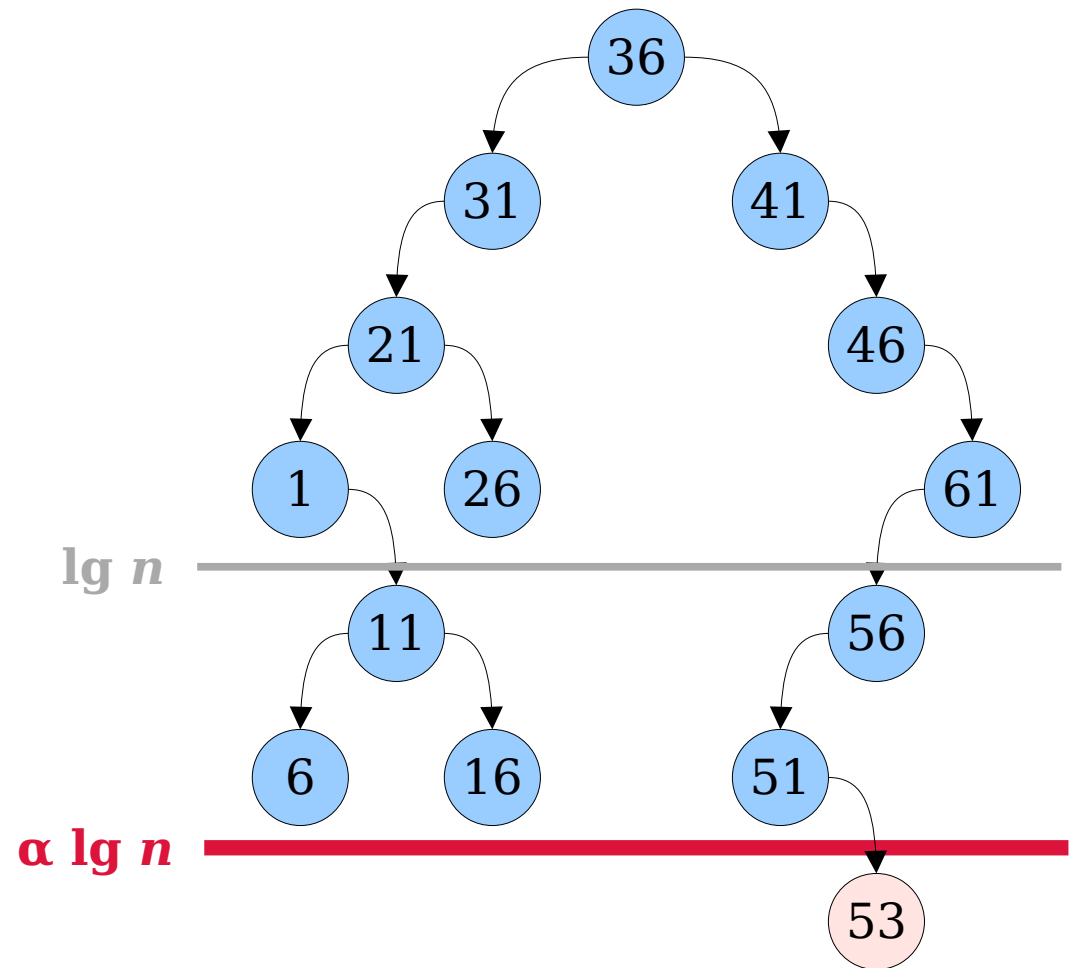
Adding Slack Space

- For each node v in our BST, let **size(v)** denote the number of nodes in the subtree rooted at v and **height(v)** denote the height of the subtree rooted at v .
- We'll say that a node v is **α -balanced** if $\text{height}(v) \leq \alpha \lg \text{size}(v)$.
- Intuitively, a α -balanced node is the root of a subtree whose height is within a factor of α of optimal.



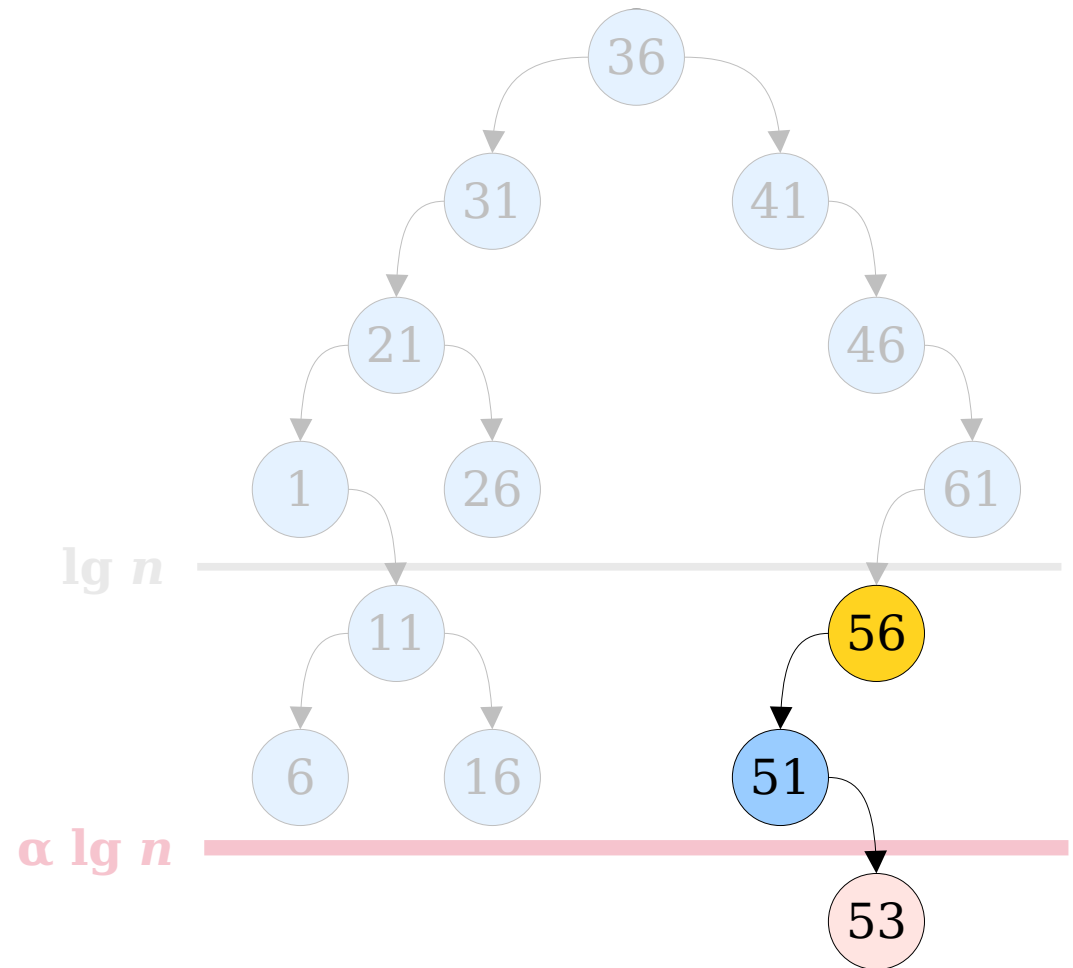
Adding Slack Space

- Suppose, however, that after doing an insertion, our tree exceeds $\alpha \lg n$.
- At this point, we need to do some sort of “cleanup” on the tree to pull it back to a reasonable height.
- Ideally, we’ll want to minimize the amount of cleanup we need to do so that this step will run quickly.



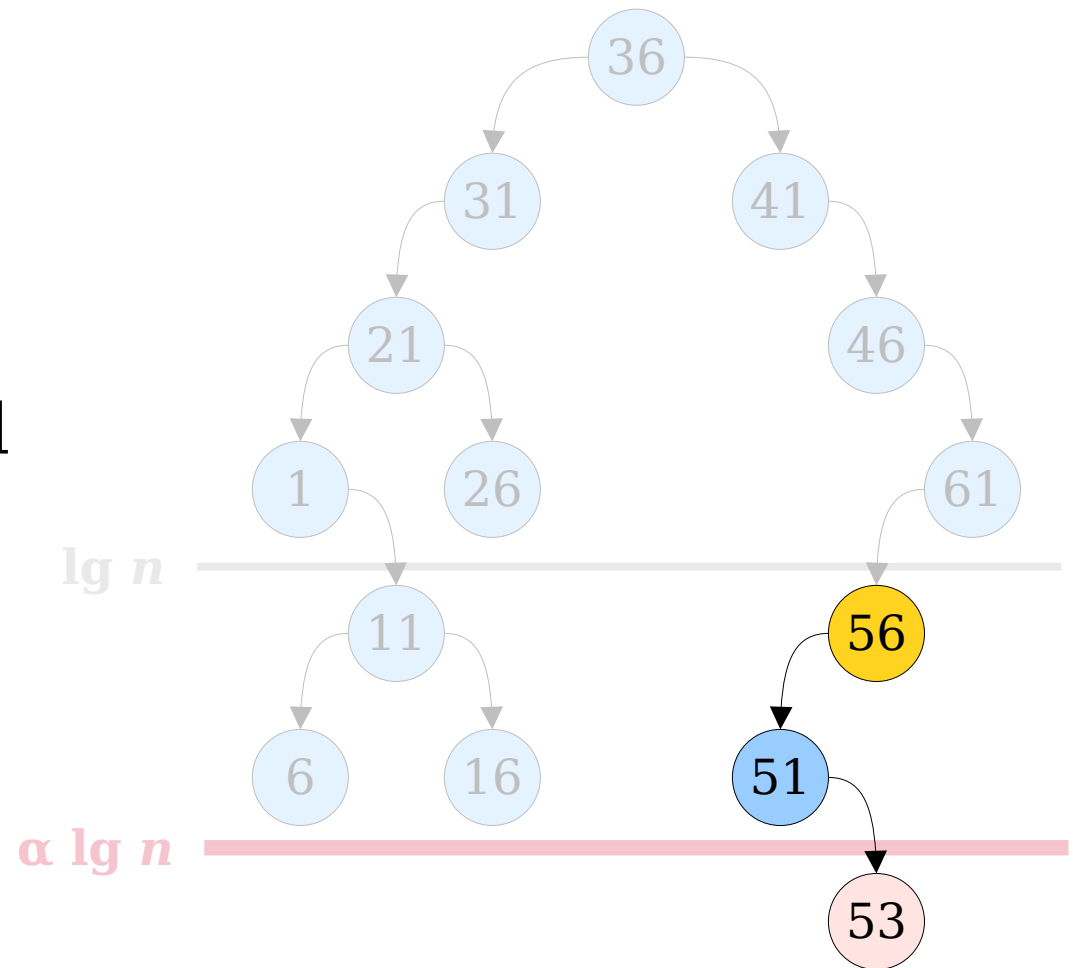
Scapegoat Nodes

- Look at the access path from the root node to the newly-inserted node.
- We know the root node is not α -balanced, since the overall tree is too tall.
- We also know that the newly-inserted node is α -balanced, since it has no children.
- Therefore, there has to be some deepest node on the access path that isn't α -balanced.
- We can “blame” the imbalance in the overall tree on this subtree. The node chosen this way is called the **scapegoat**.



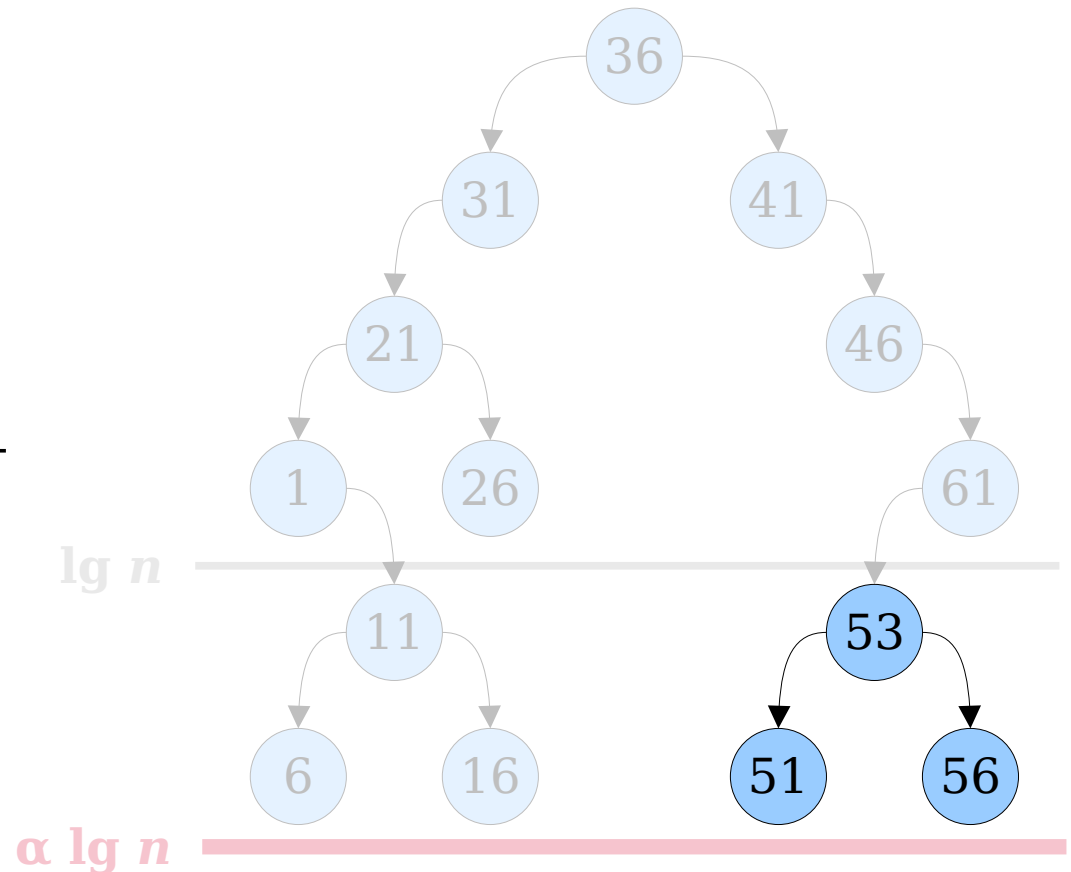
Scapegoat Nodes

- We know that the subtree rooted at the scapegoat isn't α -balanced.
- **Idea:** Rebuild this tree as a perfectly-balanced BST.
- This will reduce the height of the subtree, which in turn restores the requirement that the height be at most $\alpha \lg n$.



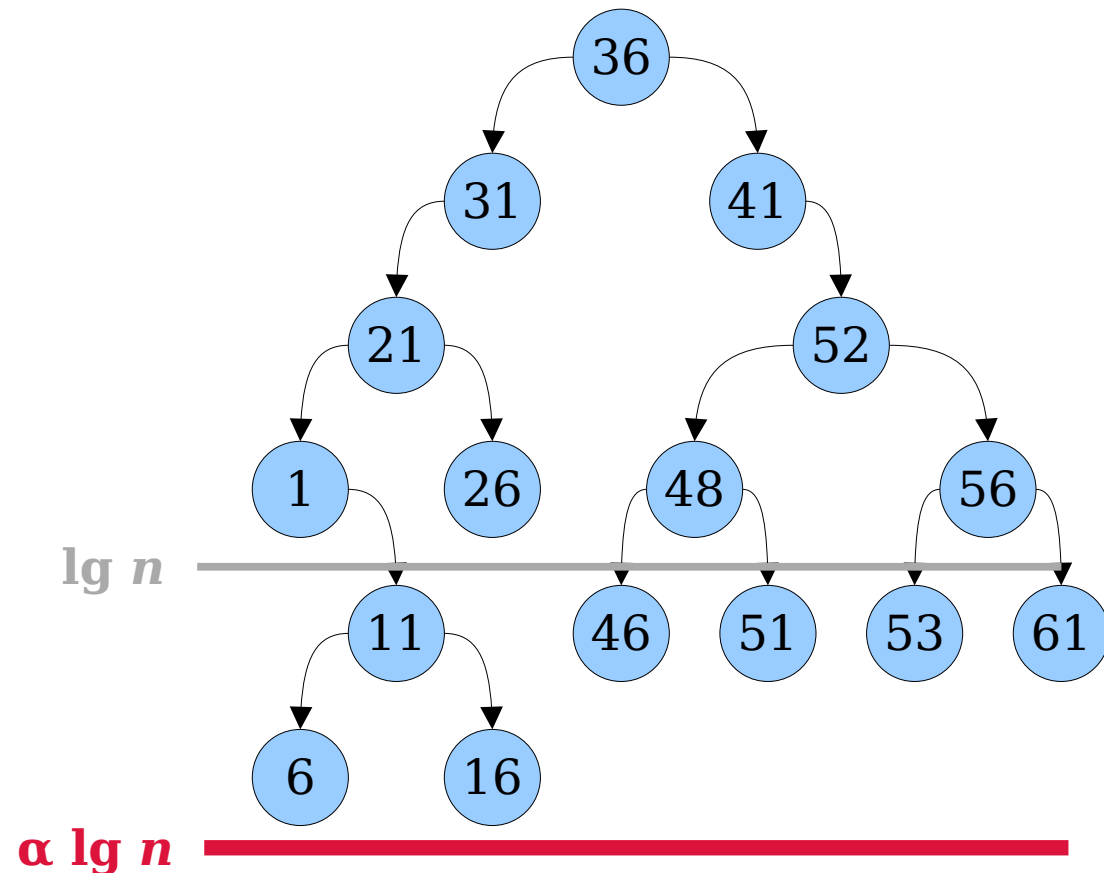
Scapegoat Nodes

- We know that the subtree rooted at the scapegoat isn't α -balanced.
- **Idea:** Rebuild this tree as a perfectly-balanced BST.
- This will reduce the height of the subtree, which in turn restores the requirement that the height be at most $\alpha \lg n$.



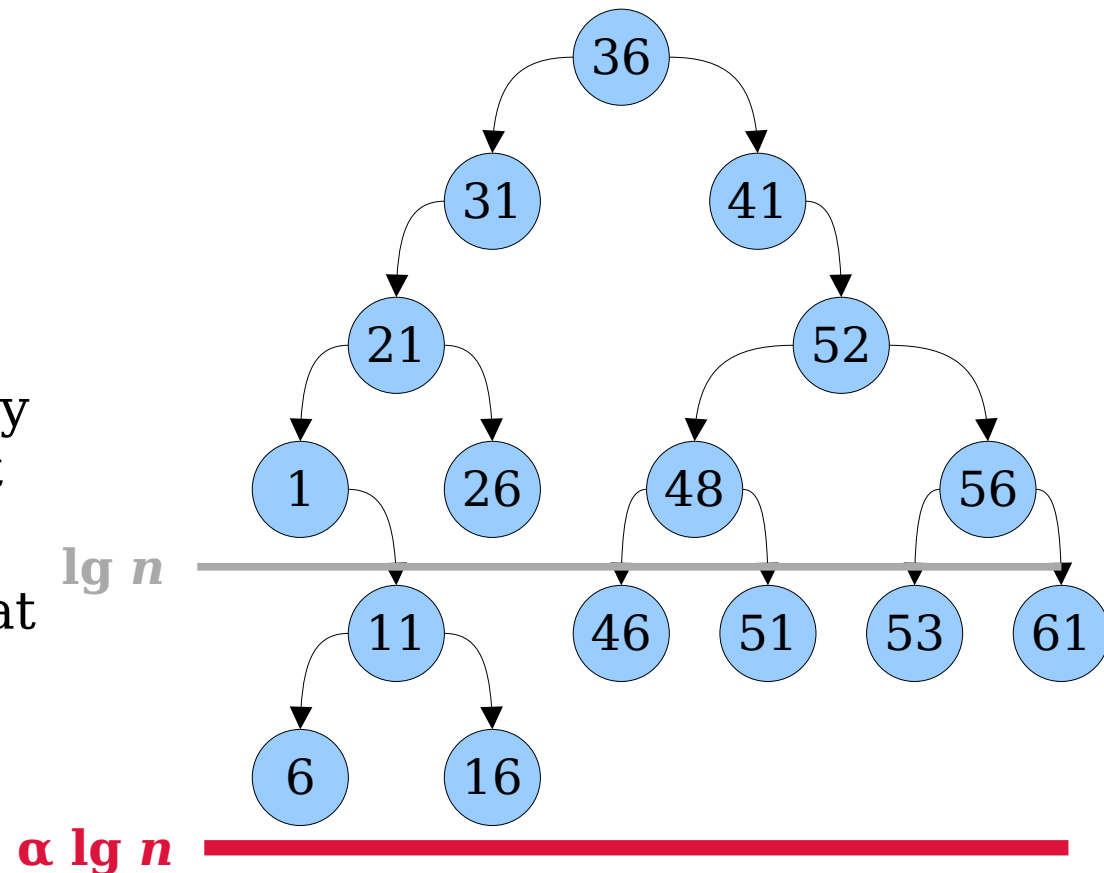
Scapegoat Trees

- A **scapegoat tree** is a balanced binary search tree that works as follows:
 - Pick some constant $\alpha > 1$.
 - As long as the tree height is below $\alpha \lg n$, don't do any rebalancing after insertions.
 - Once the tree exceeds that height, find the scapegoat (the deepest α -imbalanced node on the insertion path).
 - Then, optimally rebuild the subtree rooted at that node.
- All that's left now is to work through the details.



Scapegoat Trees

- Questions we need to address:
 - How do we know that optimally rebuilding the scapegoat's subtree will fix the tree height?
 - How quickly can we optimally rebuild the subtree rooted at the scapegoat node?
 - How do we find the scapegoat node?
 - In an amortized sense, how fast is this strategy?
- Let's address each of these in turn.



The Impact of Rebuilding

Scapegoat Rebuilding

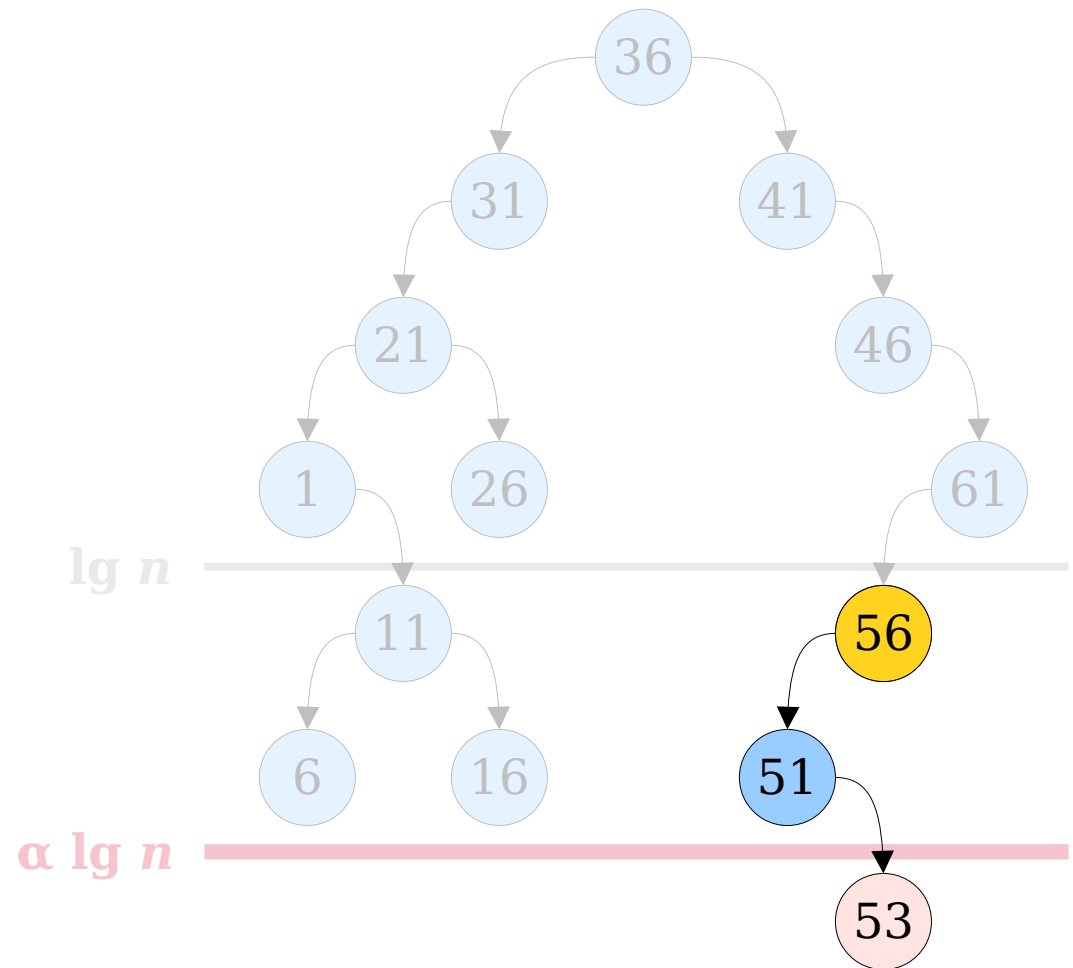
- Our strategy relies on the following claim:

Optimally rebuilding the subtree rooted at the scapegoat node ensures that, as a whole, the tree has height at most $\alpha \lg n$.

- This turns out to not be too difficult to prove. Let's break it down into pieces.

Scapegoat Rebuilding

- Suppose we insert a node that causes the $\alpha \lg n$ size limit to be violated.
- Just before we inserted that node, all other nodes in the tree were at height $\alpha \lg n$ or below.
- That means each other node is at depth $\lfloor \alpha \lg n \rfloor$, and our new node is at depth $\lfloor \alpha \lg n \rfloor + 1$.
- Now, look at the scapegoat node and its subtree.
- Because our offending node is only one level too deep, we just need to show that optimally rebuilding the scapegoat subtree reduces its depth by at least one.



Scapegoat Rebuilding

- Let v be our scapegoat node. Since it's not α -balanced, we know that

$$\text{height}_{\text{before}}(v) > \alpha \lg \text{size}(v).$$

- Let r be the root of the subtree we get after rebuilding at v . Because we rebuilt v 's tree perfectly, we know that

$$\lg \text{size}(v) \geq \text{height}_{\text{after}}(r).$$

- Putting this together gives us that

$$\text{height}_{\text{before}}(v) > \alpha \lg \text{size}(v) > \lg \text{size}(v) \geq \text{height}_{\text{after}}(r).$$

- This means that

$$\text{height}_{\text{before}}(v) > \text{height}_{\text{after}}(r).$$

- Therefore, the height of v 's subtree after rebuilding has decreased by at least one, so overall balance is restored.

The Cost of Rebuilding

The Cost of Rebuilding

- Once we've identified the scapegoat node, we need to rebuild the subtree rooted at that node as a perfectly-balanced BST.
- How quickly can we do this?

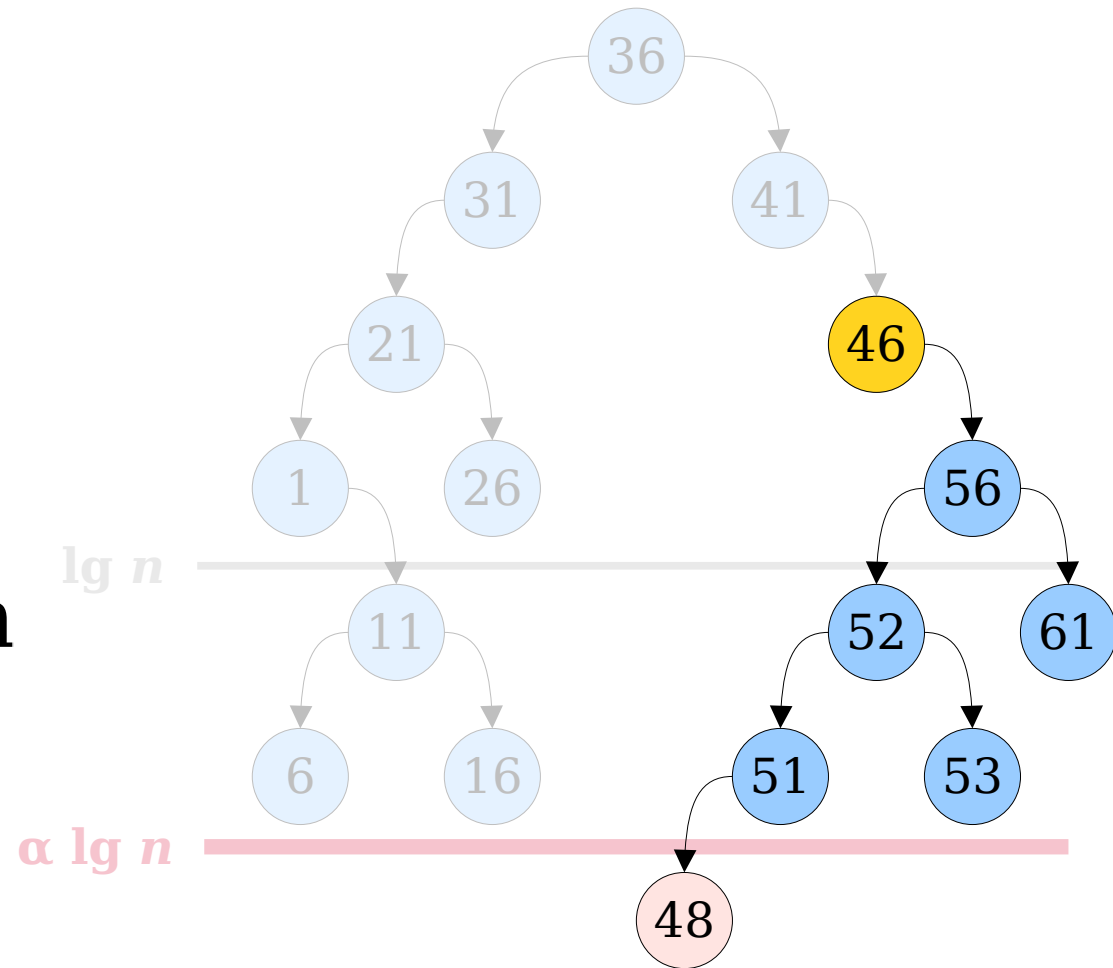
The Cost of Rebuilding

- Run an inorder traversal over the subtree and form an array of its nodes in sorted order.
- Use the following recursive algorithm to build an optimal tree:
 - If there are no nodes left, return an empty tree.
 - Otherwise, put the median element at the root of the tree, and recursively build its left and right subtrees optimally.
- The cost of this strategy is $O(\text{size}(v))$, where v is the node at the root of the subtree.
 - Quick way to see this: the inorder traversal takes time $O(\text{size}(v))$ because there are $\text{size}(v)$ nodes visited, and the recursive algorithm has the recurrence $T(m) = 2T(m / 2) + O(1)$.
- This is the simplest algorithm to optimally rebuild the tree, but others exist that are faster in practice or more space-efficient. Look up the **Galperin-Rivest** or **Day-Stout-Warren** algorithms for other ways to do this in time $O(\text{size}(v))$ in less space.

Finding the Scapegoat Node

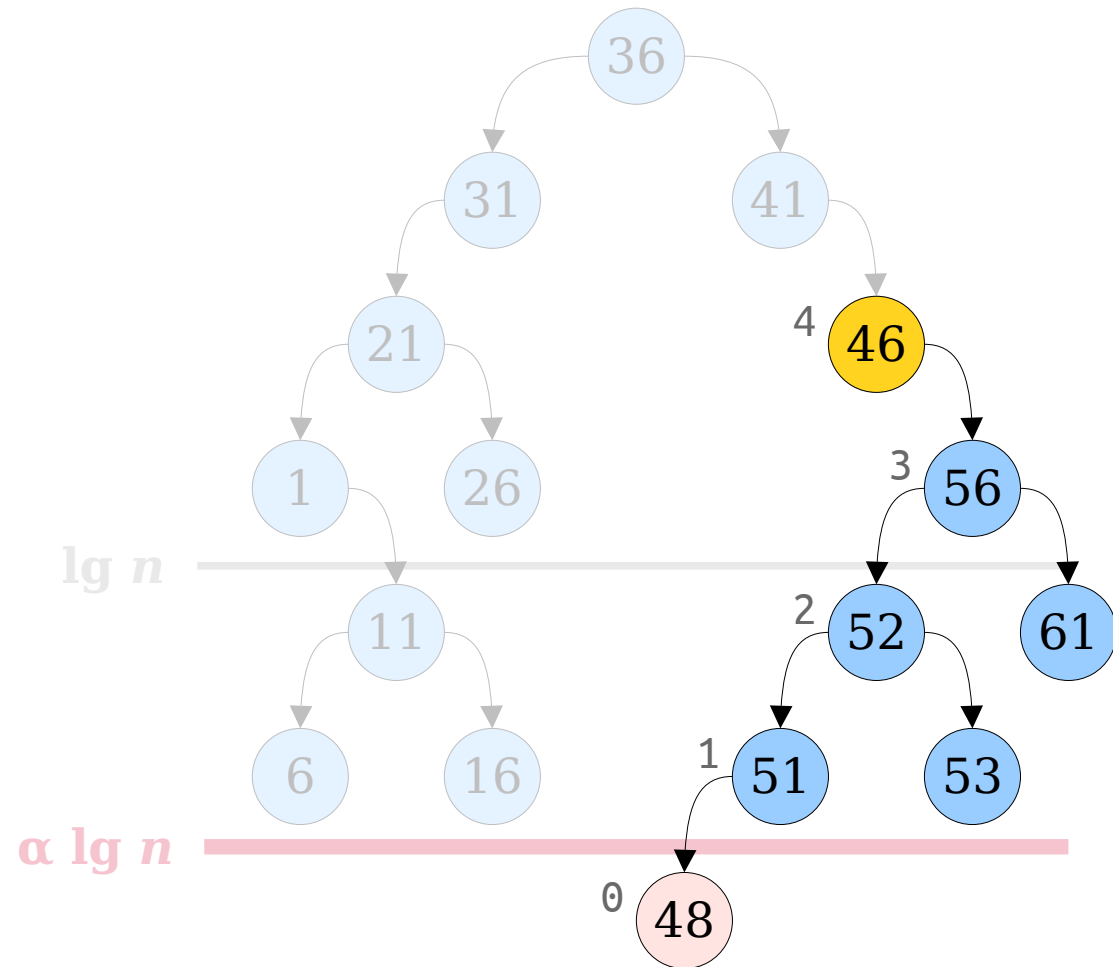
Finding the Scapegoat

- **Recall:** The scapegoat node is the deepest node on the access path that isn't α -balanced.
- How efficiently can we identify this node?



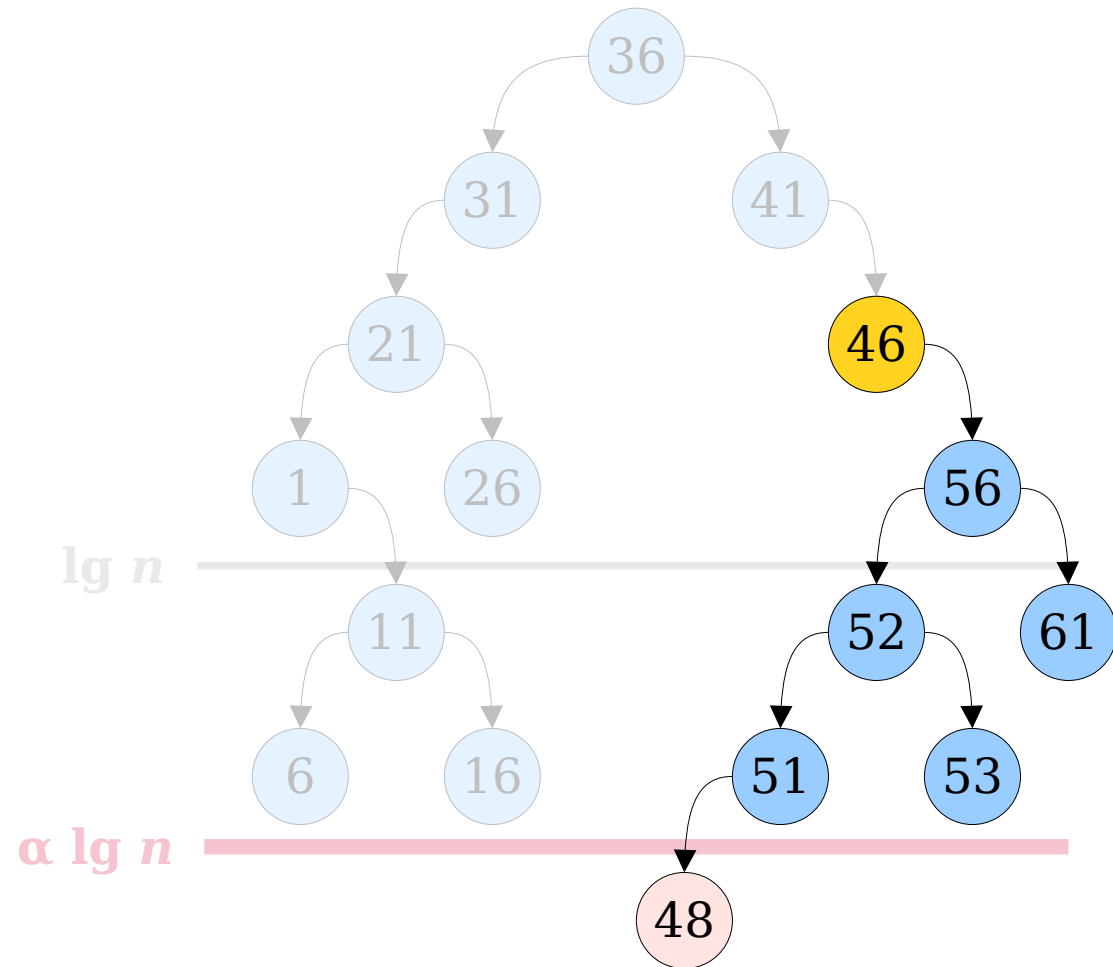
Finding the Scapegoat

- We need to check if $\text{height}(v) > \alpha \lg \text{size}(v)$.
- **Observation:** For each node v on the access path, $\text{height}(v)$ is the number of steps between v and the newly-added node.
 - This can be computed by counting upward from the new node.
- That just leaves computing $\text{size}(v)$.



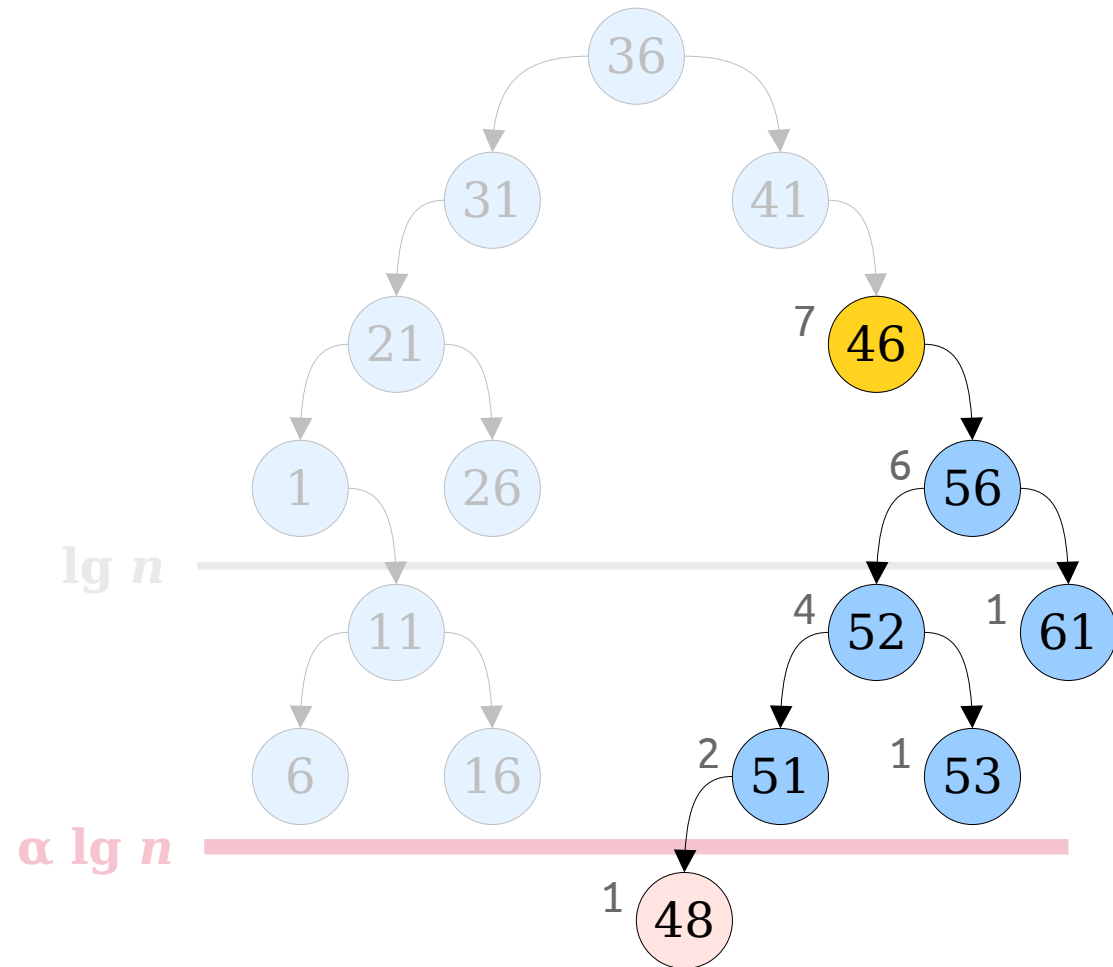
Finding the Scapegoat

- There are two ways we can compute $\text{size}(v)$ for the nodes on the access path.
- **Approach 1:** Augment each node with the number of nodes in its subtree.
 - (This can be done without changing the cost of an insertion or deletion.)
- We can then read $\text{size}(v)$ by looking at the cached value.
- This has the disadvantage of requiring an extra integer in each node of the tree.



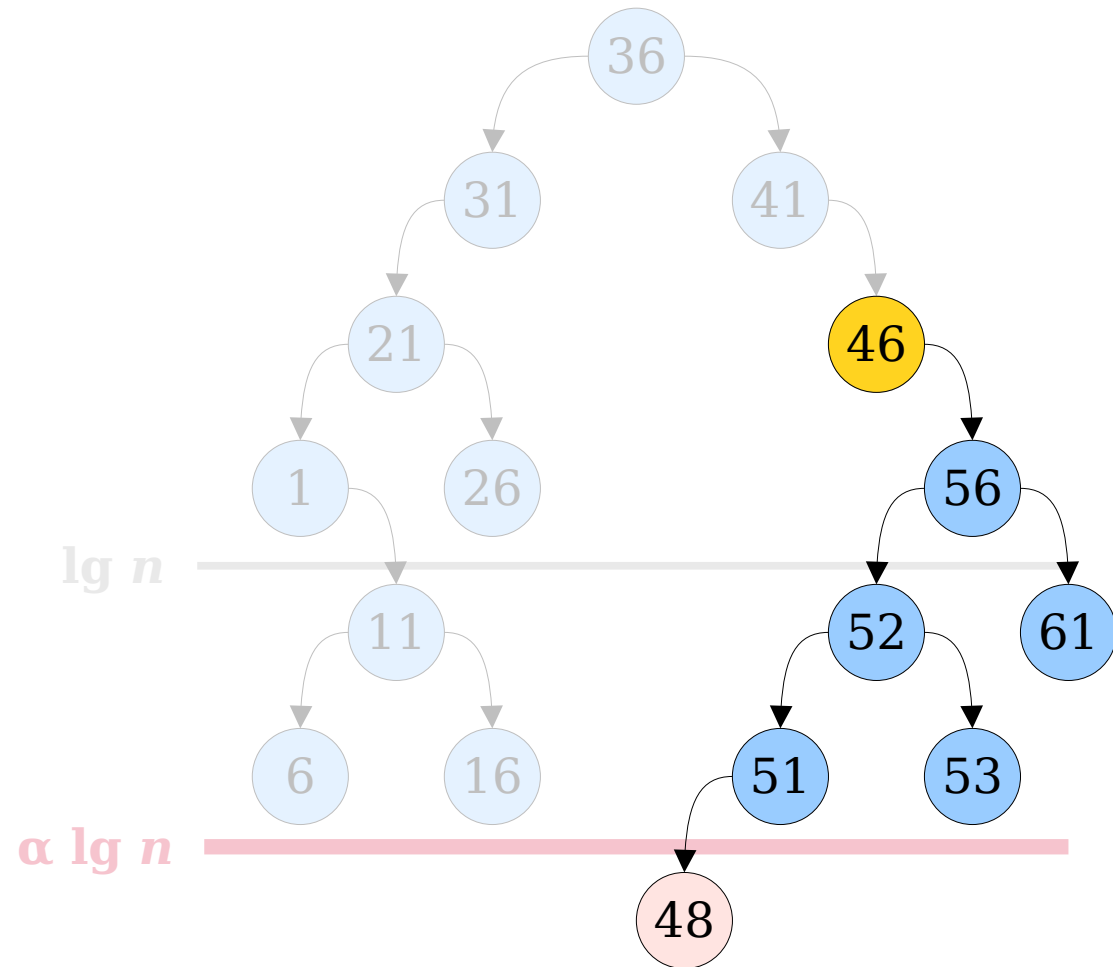
Finding the Scapegoat

- **Approach 2:** Compute these values bottom-up.
- Start with a total of 1 for the newly-added node.
- Each time we move upward a step, run a DFS in the opposite subtree to count the number of nodes there.
- Once we hit the scapegoat node v , we'll have done $O(\text{size}(v))$ total work counting nodes.



Finding the Scapegoat

- **Approach 1** does less work, but requires more storage in each node.
- **Approach 2** does more work, but means each node just stores data and two child pointers.
- Which of these ends up being more important depends on a mix of engineering constraints and personal preference.



Analyzing Efficiency

Analyzing Efficiency

- Based on what we've seen so far, the cost of an insertion is
 - $O(\log n)$ if the insertion keeps us below the $\alpha \lg n$ height threshold, and
 - $O(\log n + \text{size}(v))$ if we have to rebuild v as a scapegoat.
- The $\text{size}(v)$ term can be as large as n , which may happen if the whole tree has to be rebuilt.
- However, it turns out that we can amortize this $\text{size}(v)$ term away.

Analyzing Efficiency

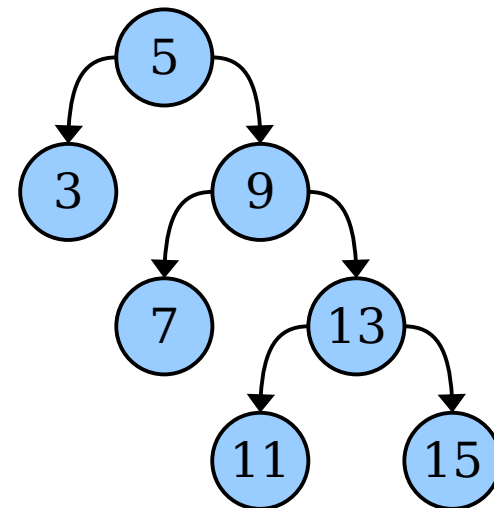
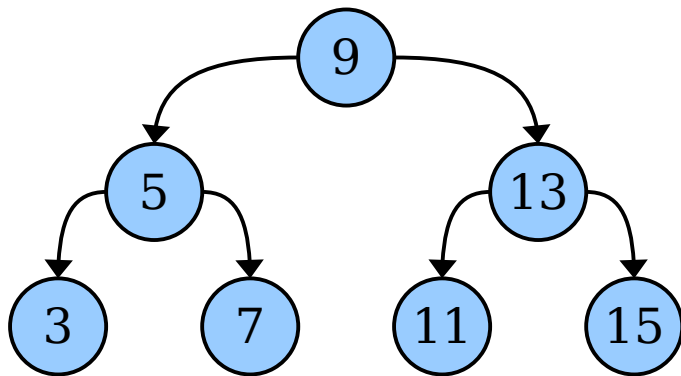
- **Recall:** To perform an amortized analysis, we do the following:
 - Find a potential function Φ that, intuitively, is small when the data structure is “clean” and large when the data structure is “messy.”
 - Compute the value of $\Delta\Phi = \Phi_{\text{after}} - \Phi_{\text{before}}$ for each operation.
 - Assign amortized costs as
$$\text{amortized-cost} = \text{real-cost} + k \cdot \Delta\Phi$$
for some constant k we get to pick.
- Our first step is to find a choice of Φ that’s large when our tree is imbalanced and small when it’s balanced.

Quantifying Imbalance

- Right before we rebuild a scapegoat subtree, that tree is α -imbalanced.
- Right after we rebuild a scapegoat subtree, that tree is perfectly balanced.
- **Goal:** Find a choice of Φ for our tree so that
 - perfectly-balanced trees have low Φ , and
 - α -imbalanced trees have high Φ .
- At this point, we need to do some exploring to see what we find.

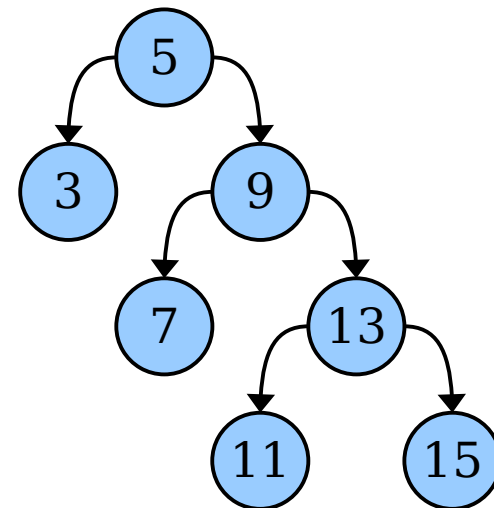
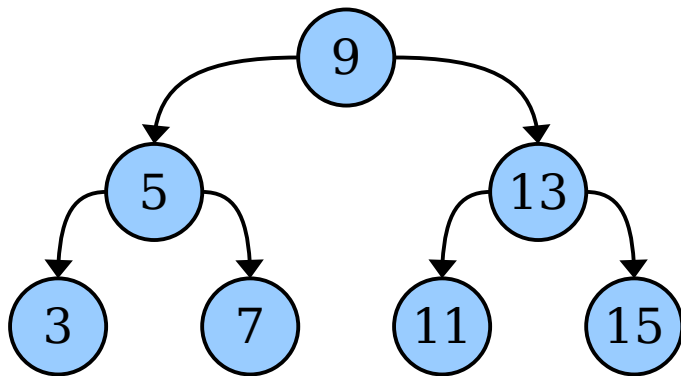
Quantifying Imbalance

- When we talk about “perfectly balanced” trees, what exactly is this “balance” in reference to?
- **Intuition 1:** A perfectly balanced tree is one where each node has roughly the same number of children in its left subtree as in its right subtree.
- **Intuition 2:** An “imbalanced” tree will have nodes whose left and right subtrees have differing numbers of nodes.



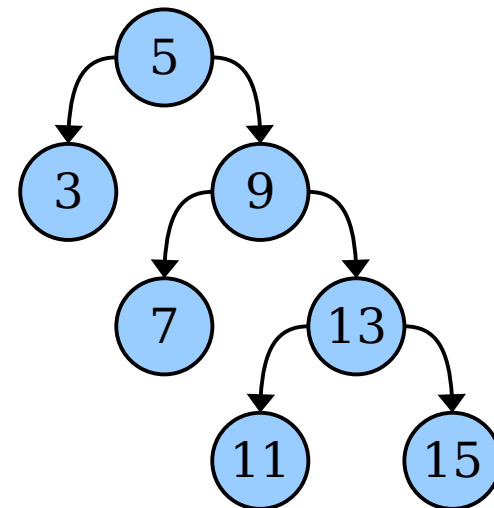
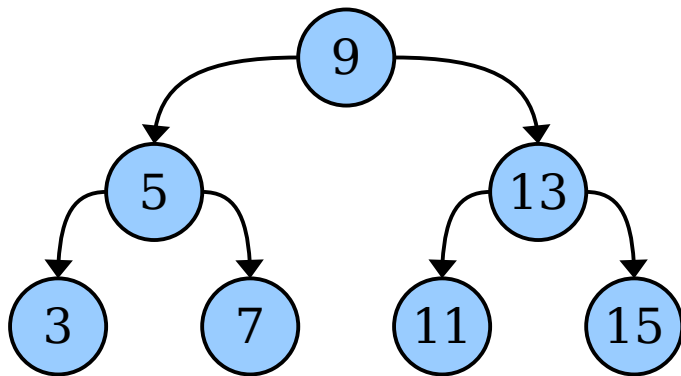
Quantifying Imbalance

- For each node v , define the ***imbalance*** of the node as
$$(v) = |\text{size}(v.\text{left}) - \text{size}(v.\text{right})|.$$
- This gives us a quantitative measure of our more nebulous concept of “imbalance.”



Defining our Potential

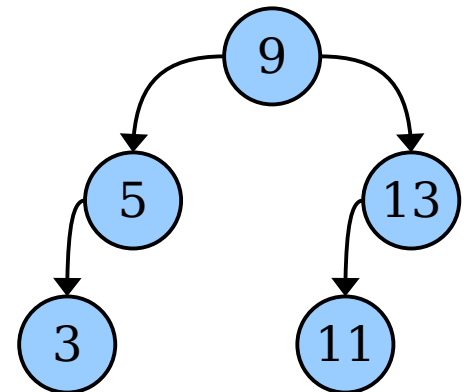
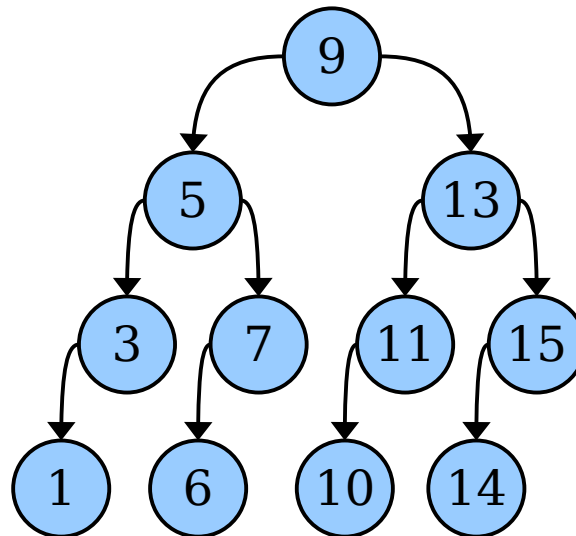
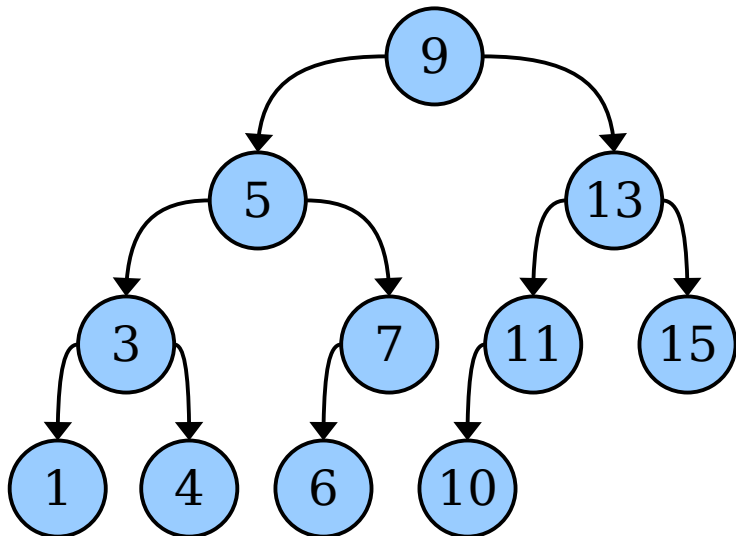
- We're looking for a potential function Φ where
 - a perfectly-balanced tree has low Φ , and
 - an imbalanced tree has progressively higher Φ .
- A balanced tree has $\Phi(v)$ low for all its nodes.
- An imbalanced tree has $\Phi(v)$ high for many nodes.
- **Initial Idea:** Define $\Phi = \sum_v \Phi(v)$.



Defining our Potential

- We've set $\Phi = \sum_v \quad (v)$.
- What is Φ for the three trees shown below?

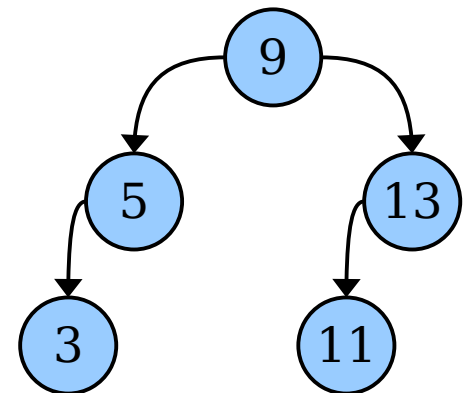
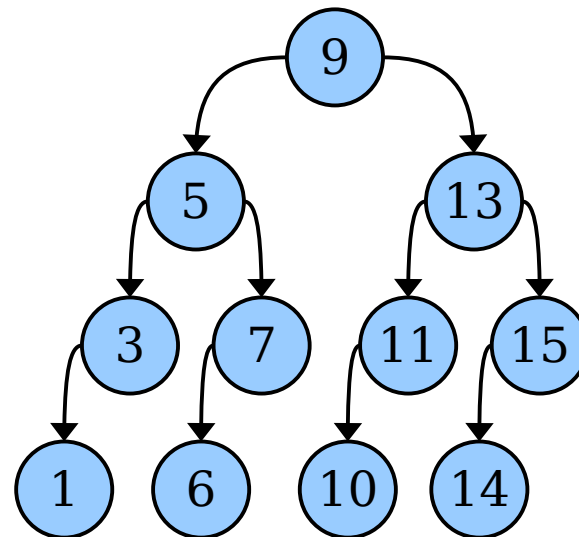
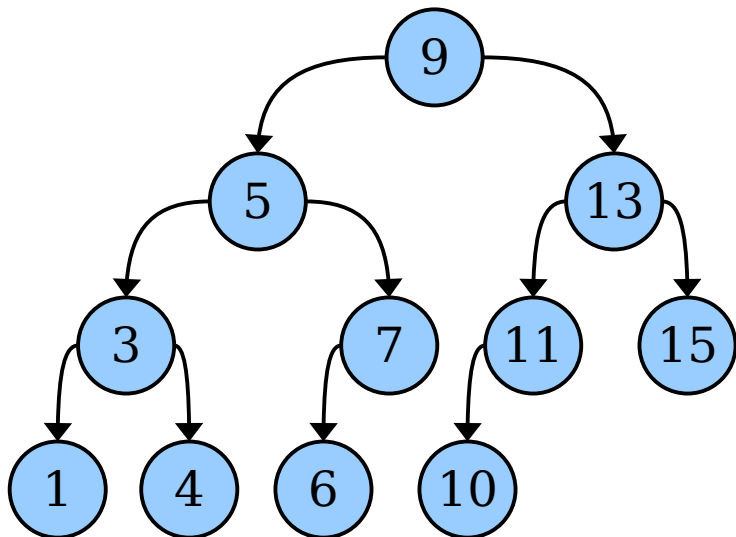
Formulate a hypothesis!



Defining our Potential

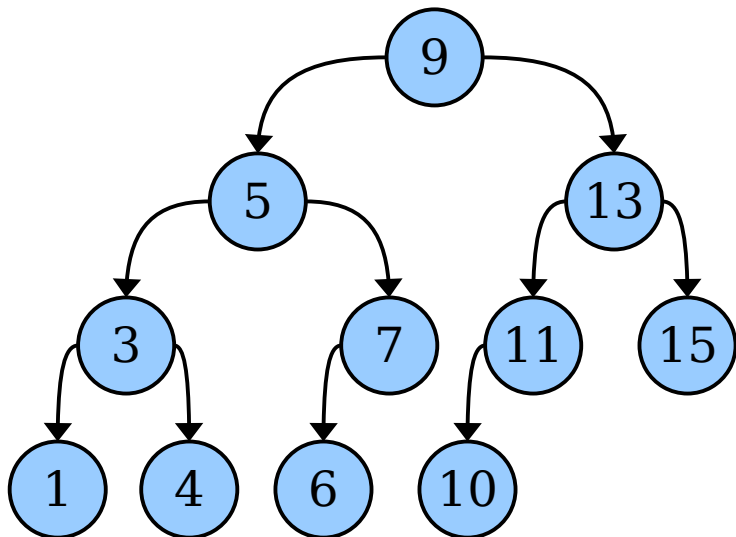
- We've set $\Phi = \sum_v \quad (v)$.
- What is Φ for the three trees shown below?

Discuss with your neighbors!

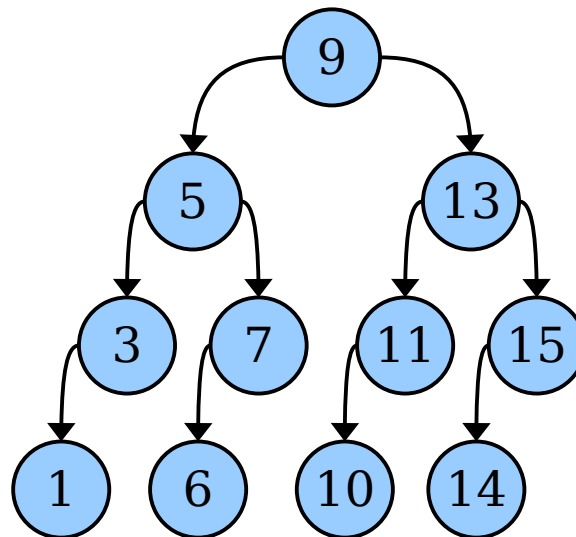


Defining our Potential

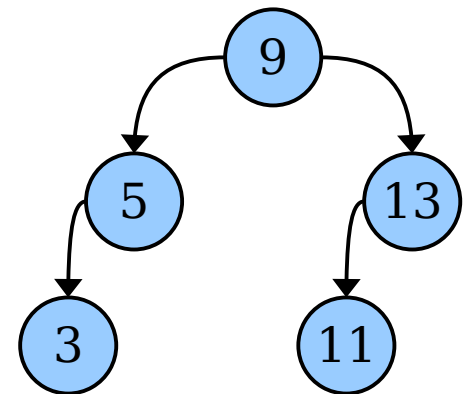
- We've set $\Phi = \sum_v \quad (v)$.
- What is Φ for the three trees shown below?



$$\Phi = 6$$



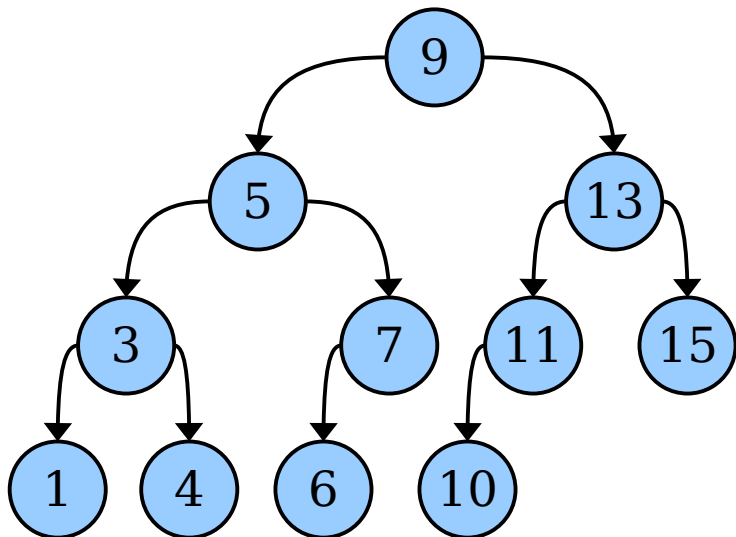
$$\Phi = 4$$



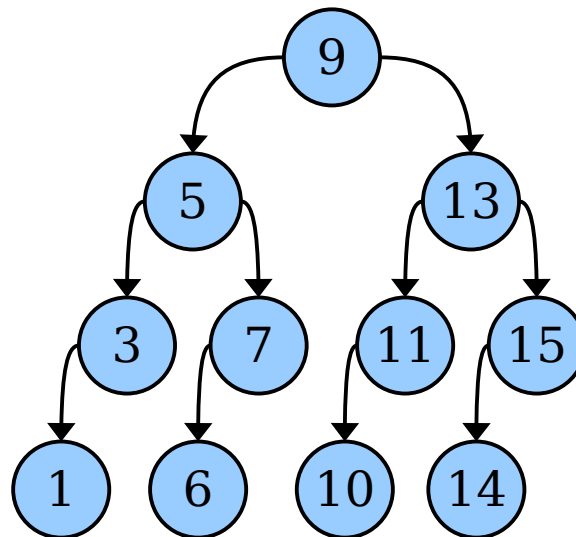
$$\Phi = 2$$

Defining our Potential

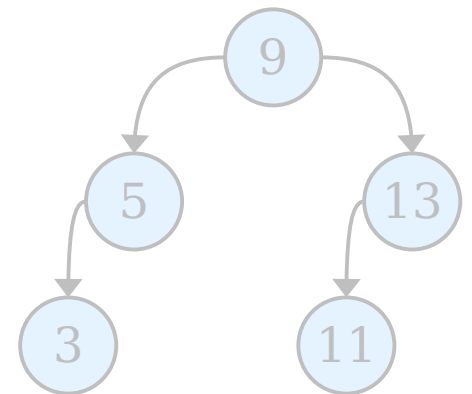
- **Observation 1:** Two trees that fill their rows as efficiently as possible may have different potentials.
- This means that when we rebalance trees, we need to make sure to equalize the number of nodes in the left and right subtrees of each node.



$$\Phi = 6$$



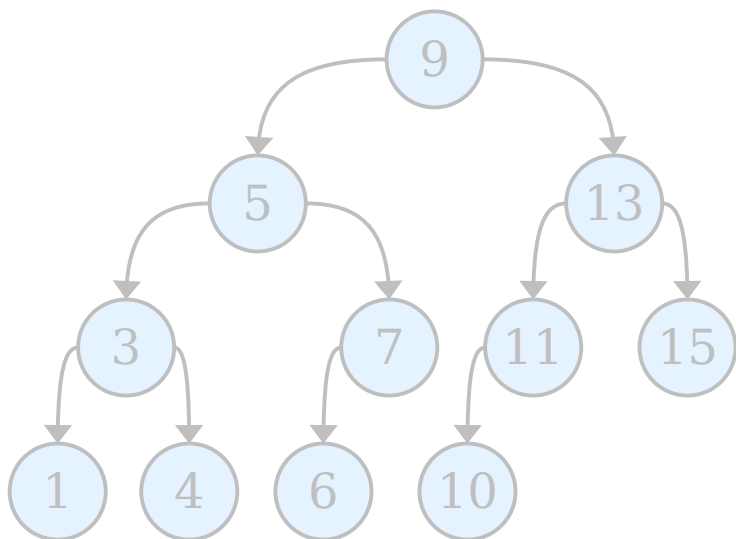
$$\Phi = 4$$



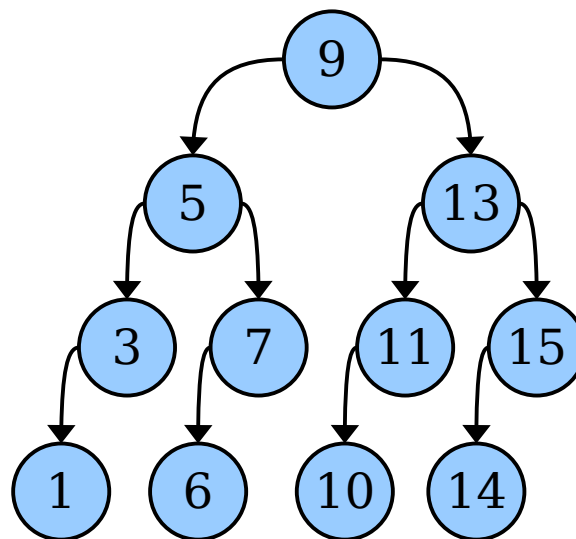
$$\Phi = 2$$

Defining our Potential

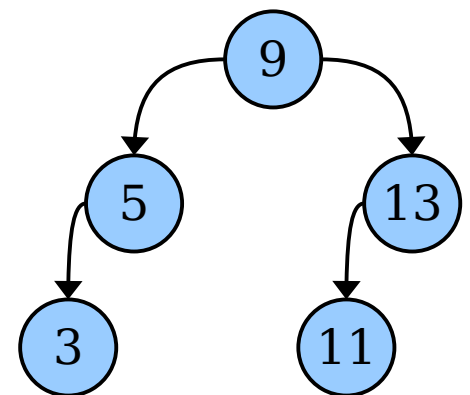
- **Observation 2:** The potential of a perfectly-balanced tree can grow as a function of its number of nodes.
- Ideally, both of these trees should have potential 0, indicating “perfectly balanced.” The potential shouldn’t depend on the number of nodes in the tree.



$$\Phi = 6$$



$$\Phi = 4$$



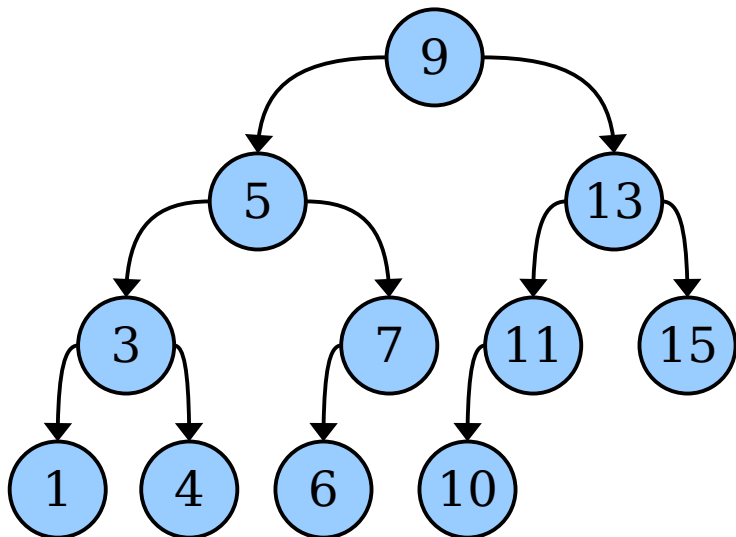
$$\Phi = 2$$

Defining our Potential

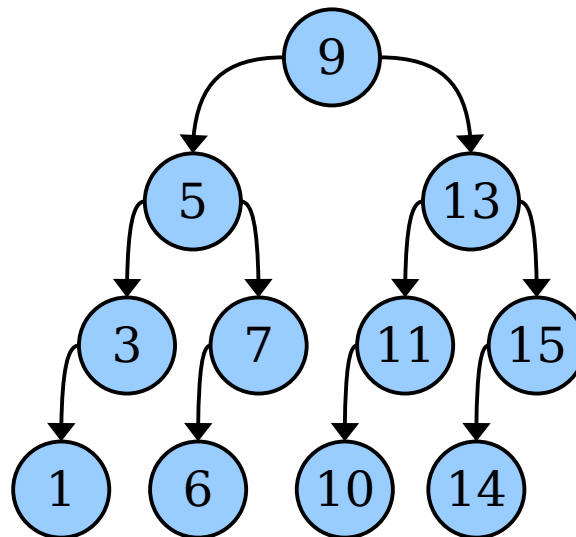
- To account for otherwise balanced trees with extra nodes in their bottom layers, let's define $\phi(v)$ as
 - $\phi(v) = 0$ if $h(v) \leq 1$.
 - $\phi(v) = h(v)$ otherwise.
- **Revised Idea:** Set $\Phi = \sum_v \phi(v)$.

Defining our Potential

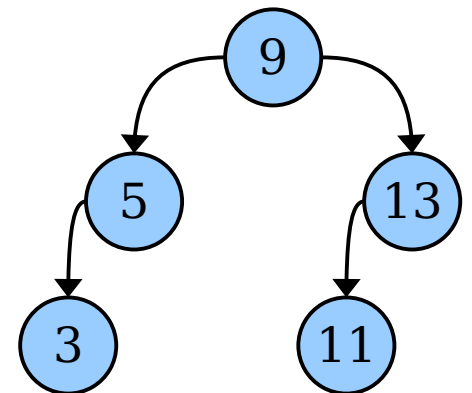
- We're now using $\Phi = \sum_v \phi'(v)$.
- What is Φ for the three trees shown below?
- **Intuition:** If a subtree rooted at v is perfectly balanced, then $\phi'(v) = 0$.



$$\Phi = 2$$



$$\Phi = 0$$



$$\Phi = 0$$

Analyzing Scapegoat Trees

- Now that we have a definition of Φ , we can look at the amortized cost of an insertion.
- We need to consider two cases:
 - **Case 1:** The insertion doesn't trigger a rebuild.
 - **Case 2:** The insertion triggers a rebuild.
- Intuitively, we're hoping that Case 1 has a small positive $\Delta\Phi$ (messes accumulate slowly) and that Case 2 has a large negative $\Delta\Phi$ (messes get cleaned up quickly).
- Let's run the numbers!

Analyzing Scapegoat Trees

- **Case 1:** Our insertion does not trigger a rebuild.
- Recall that

$$\text{amortized-cost} = \text{real-cost} + k \cdot \Delta\Phi$$

for a constant k that we get to pick.

What are *real-cost* and $\Delta\Phi$,
as a function of n ?

Formulate a hypothesis!

Analyzing Scapegoat Trees

- **Case 1:** Our insertion does not trigger a rebuild.
- Recall that

$$\text{amortized-cost} = \text{real-cost} + k \cdot \Delta\Phi$$

for a constant k that we get to pick.

What are *real-cost* and $\Delta\Phi$,
as a function of n ?

Discuss with your neighbors!

Analyzing Scapegoat Trees

- **Case 1:** Our insertion does not trigger a rebuild.
- Recall that

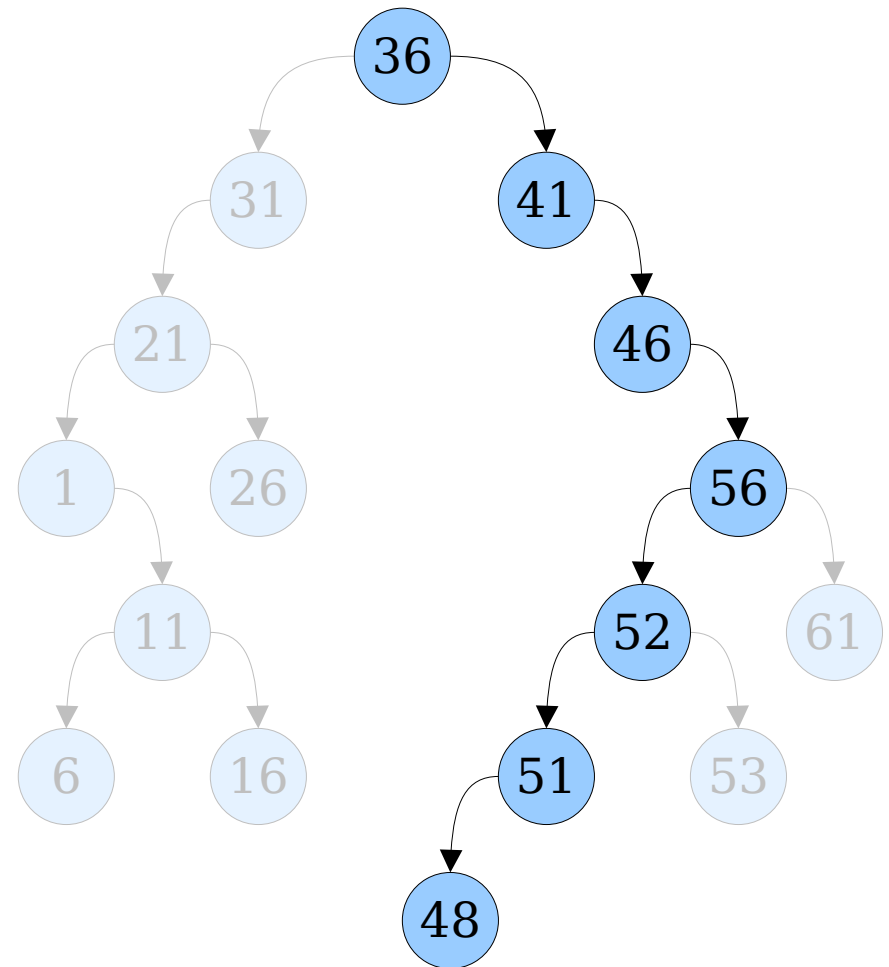
$$\text{amortized-cost} = \text{real-cost} + k \cdot \Delta\Phi$$

for a constant k that we get to pick.

- We're inserting into a tree of height at most $\alpha \lg n$, so our *real-cost* is $O(\log n)$.
- When we insert the node, it changes $\Phi(v)$ by ± 1 for each node v on its access path.
- There are $O(\log n)$ nodes on this access path, and $\Phi(v)$ increases by at most one for each of those nodes. This means $\Delta\Phi$ increases by at most two for each of those nodes.
- Therefore, $\Delta\Phi = O(\log n)$.
- Amortized cost: $O(\log n) + k \cdot O(\log n) = \mathbf{O(\log n)}$.

Analyzing Scapegoat Trees

- **Case 1:** Our insertion does not trigger a rebuild.
- In this case, $\Delta\Phi = O(\log n)$.
- Focus on any one of the new node's ancestors.
- If we rebuild the subtree rooted at that node in the future, we have to do some work to move the new node.
- **Intuition:** The $O(\log n)$ added potential corresponds to paying $O(1)$ work in advance to each of $O(\log n)$ future rebuilds.



Analyzing Scapegoat Trees

- **Case 2:** Our insertion triggers a rebuild.
- Recall that

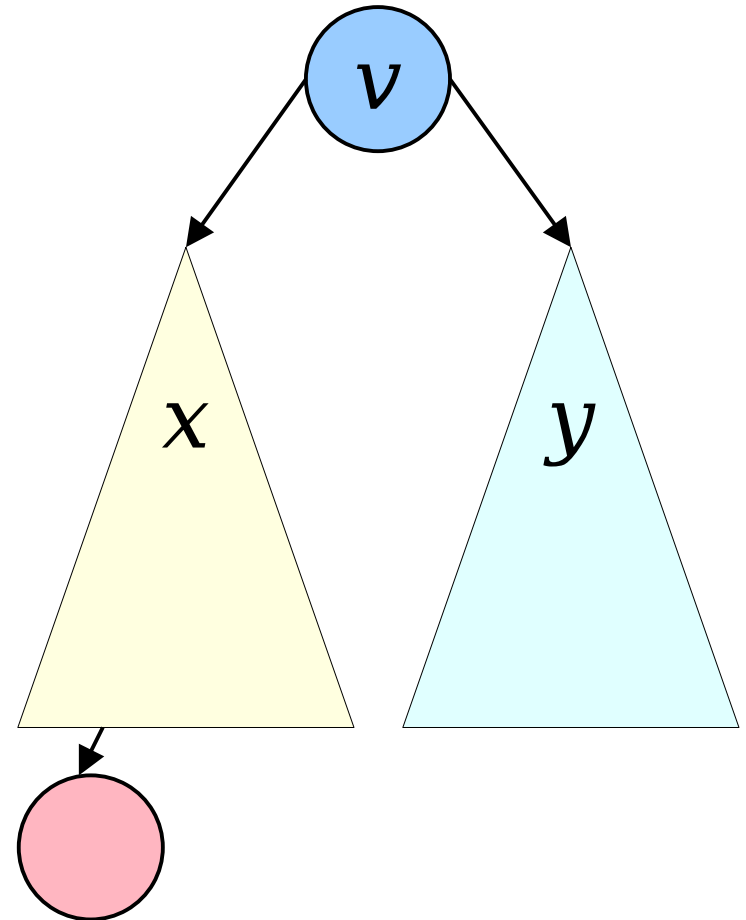
$$\textit{amortized-cost} = \textit{real-cost} + k \cdot \Delta\Phi$$

for a constant k that we pick.

- Here, *real-cost* is $O(\log n + \text{size}(v))$, where v is the scapegoat node.
 - The $O(\log n)$ comes from the cost of the actual insertion.
 - The $O(\text{size}(v))$ is for the cost of rebuilding.
- For this to amortize away, we need $\Delta\Phi$ to be $-\Omega(\text{size}(v))$.
- Our previous intuition tells us this should be the case.
- Let's run the numbers to check.

Analyzing Scapegoat Trees

- Let v be the scapegoat node. We're interested in (v) .
- One of v 's children is a tree containing our newly-inserted node. Call that subtree x .
- Call v 's other child y .
- **Goal:** Determine $(v) = |\text{size}(x) - \text{size}(y)|$.

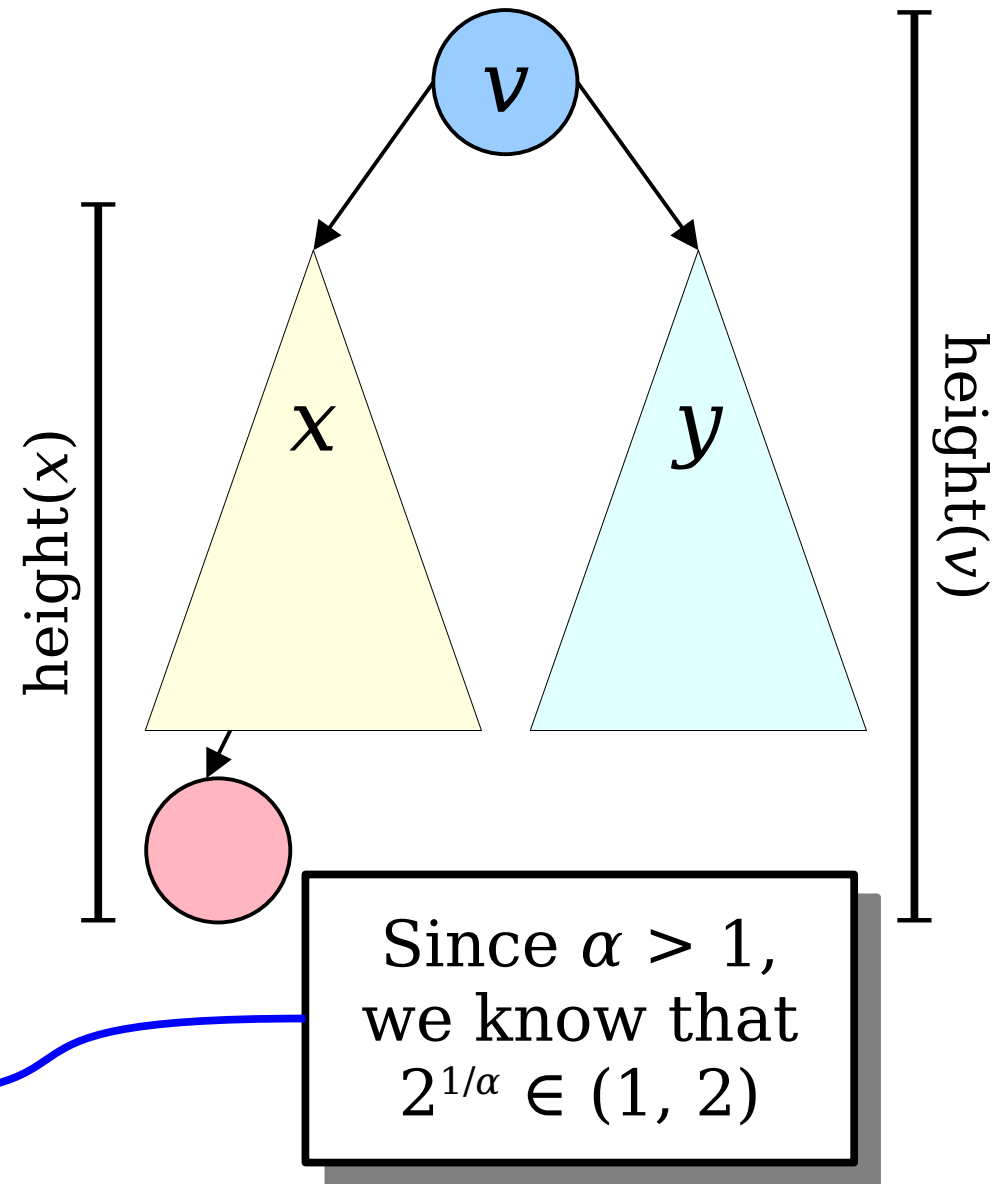


Analyzing Scapegoat Trees

- Since v is α -imbalanced, we know
$$\text{height}(v) > \alpha \lg \text{size}(v).$$
- v is the **deepest** α -imbalanced node on the access path. This means x is α -balanced, so
$$\text{height}(x) \leq \alpha \lg \text{size}(x).$$
- Since the newly-inserted node is the deepest node in v 's subtree, we know that

$$\text{height}(v) = \text{height}(x) + 1.$$

- Putting all this together gives
$$\alpha \lg \text{size}(v) < \alpha \lg \text{size}(x) + 1.$$
- That in turn means that
$$\text{size}(v) < \text{size}(x) \cdot 2^{1/\alpha}.$$



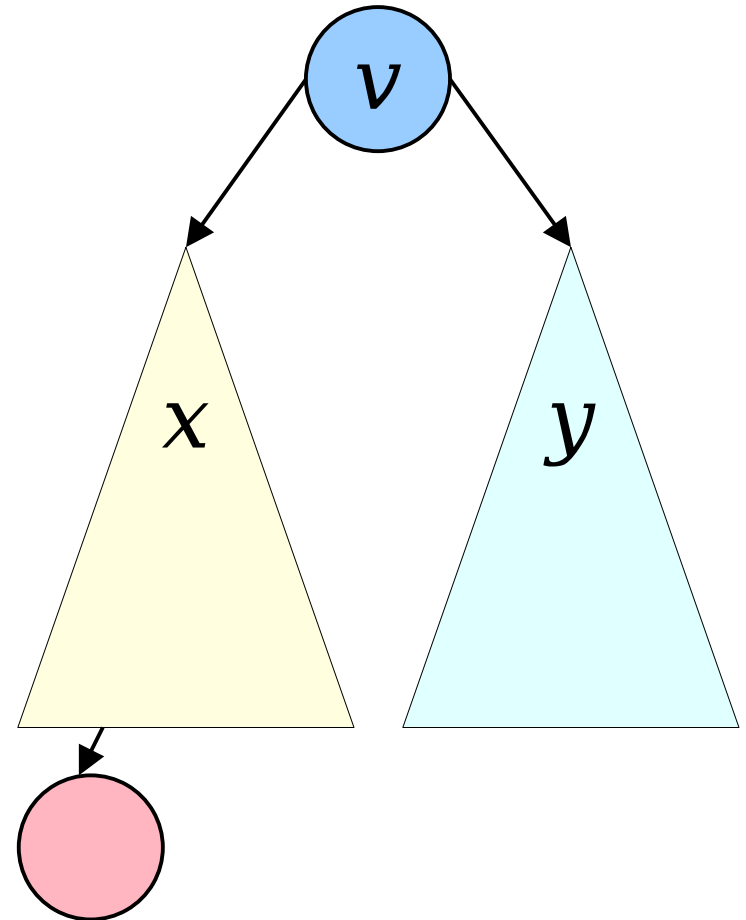
Analyzing Scapegoat Trees

- We just proved that
$$\text{size}(v) < \text{size}(x) \cdot 2^{1/\alpha}.$$
- We also know that
$$\text{size}(v) = 1 + \text{size}(x) + \text{size}(y).$$
- That means
$$\text{size}(x) + \text{size}(y) < \text{size}(x) \cdot 2^{1/\alpha}.$$
- Therefore,
$$\text{size}(y) < \text{size}(x) \cdot (2^{1/\alpha} - 1).$$

Since $2^{1/\alpha} \in (1, 2)$, we know $2^{1/\alpha} - 1 \in (0, 1)$.

So y must have fewer nodes than x .

(Surprising, but true!
Explore and see why!)



Analyzing Scapegoat Trees

- We just proved that

$$\text{size}(v) < \text{size}(x) \cdot 2^{1/\alpha}.$$

- We also know that

$$\text{size}(v) = 1 + \text{size}(x) + \text{size}(y).$$

- That means

$$\text{size}(x) + \text{size}(y) < \text{size}(x) \cdot 2^{1/\alpha}.$$

- Therefore,

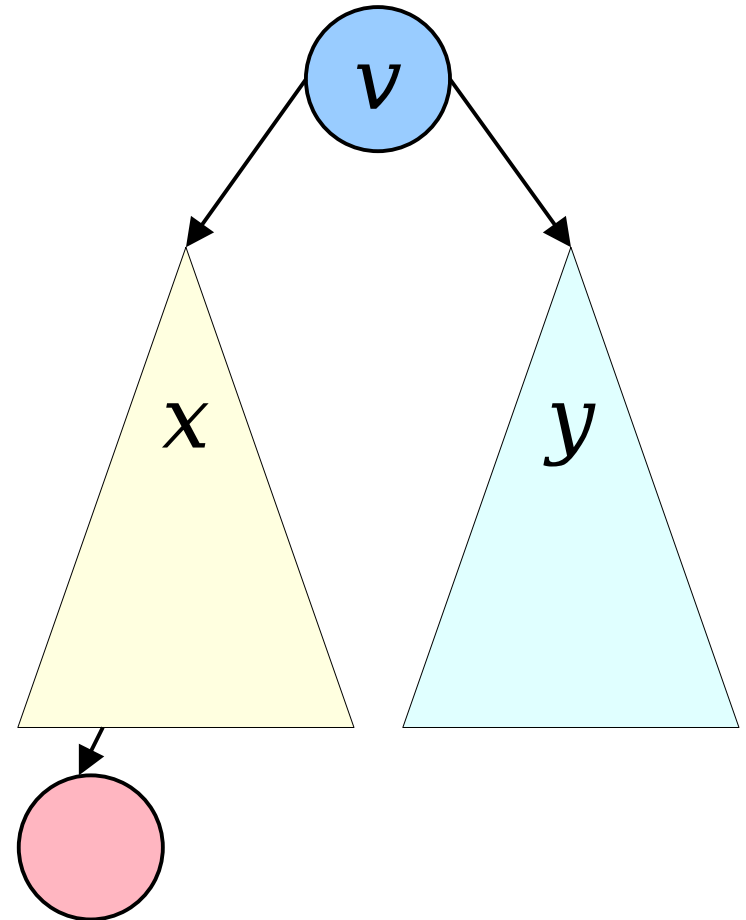
$$\text{size}(y) < \text{size}(x) \cdot (2^{1/\alpha} - 1).$$

- This means that

$$\begin{aligned} (v) &= |\text{size}(x) - \text{size}(y)| \\ &> \text{size}(x) - \text{size}(x) \cdot (2^{1/\alpha} - 1) \\ &= \text{size}(x) \cdot (2 - 2^{1/\alpha}). \end{aligned}$$

- Combined with the initial inequality, this gives us that

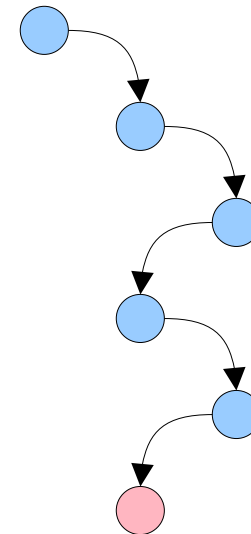
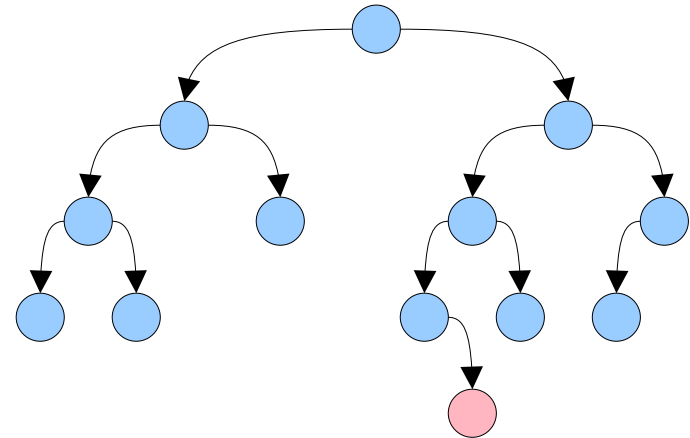
$$(v) > \text{size}(v) \cdot (2^{1-1/\alpha} - 1).$$



$$\begin{aligned} 2^{1-1/\alpha} &\in (1, 2), \\ \text{So } 2^{1-1/\alpha} - 1 &\in (0, 1). \end{aligned}$$

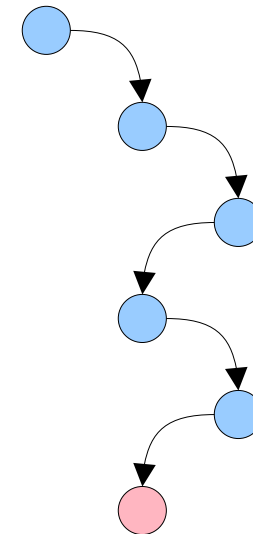
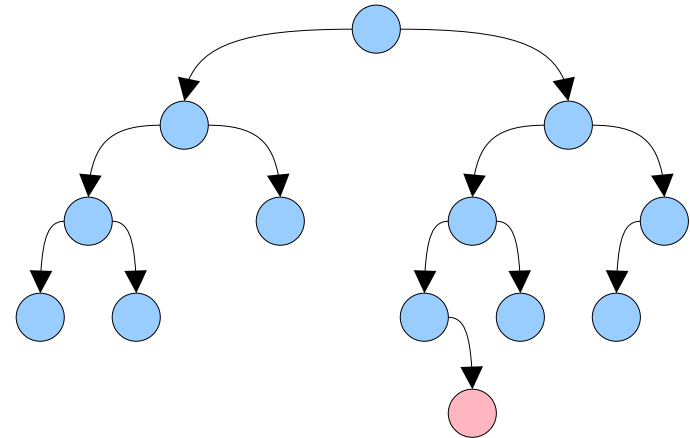
Analyzing Scapegoat Trees

- We've just concluded that
$$\beta(v) > \text{size}(v) \cdot (2^{1 - 1/\alpha} - 1)$$
- Let's take a minute to check our math.
- If α is close to 1, we're requiring the trees to be very tightly balanced. Therefore, when an imbalance occurs, we'd expect $\beta(v)$ to be small relative to $\text{size}(v)$.
- If α is large, we're allowing for huge imbalances in the trees. Therefore, when a node is too deep, we expect the tree it's a part of to be highly imbalanced, so we'd expect $\beta(v)$ to be large relative to $\text{size}(v)$.



Analyzing Scapegoat Trees

- We've just concluded that
$$b(v) > \text{size}(v) \cdot (2^{1 - 1/\alpha} - 1)$$
- Notice that for any fixed value of α that we have
$$b(v) = \Omega(\text{size}(v)).$$
- In other words, the scapegoat node always has an imbalance that is (at least) linear in the size of its subtree.
- We can then backcharge the linear work required to optimally rebuild it to the operations that caused the imbalance in the first place.



Analyzing Scapegoat Trees

- We can now work out the amortized cost of an insertion that triggers a rebuild.
 - Actual cost of inserting a new node: $O(\log n)$.
 - Actual cost of rebuilding at the scapegoat node: $O(\text{size}(v))$.
 - Change in potential: $\Delta\Phi < -\Omega(\text{size}(v))$.
- Amortized cost:
$$O(\log n) + O(\text{size}(v)) - k \cdot \Omega(\text{size}(v)).$$
- By tuning k based on the hidden constant factors in the O and Ω terms, we can get them to cancel, leaving an amortized cost of **$O(\log n)$** .

Where We Stand

- Here's the current scorecard for scapegoat trees.
- Intuitively:
 - If you pick α to be smaller, you get a more balanced tree (faster lookups), but the overhead to optimally rebuild subtrees gets bigger (slower insertions).
 - If you pick α to be larger, you get a less balanced tree (slower lookups), but the overhead to optimally rebuild trees is smaller (faster insertions).
- Tuning α appropriately now becomes a matter of engineering.
- **Question:** What about deletions?

Scapegoat Tree

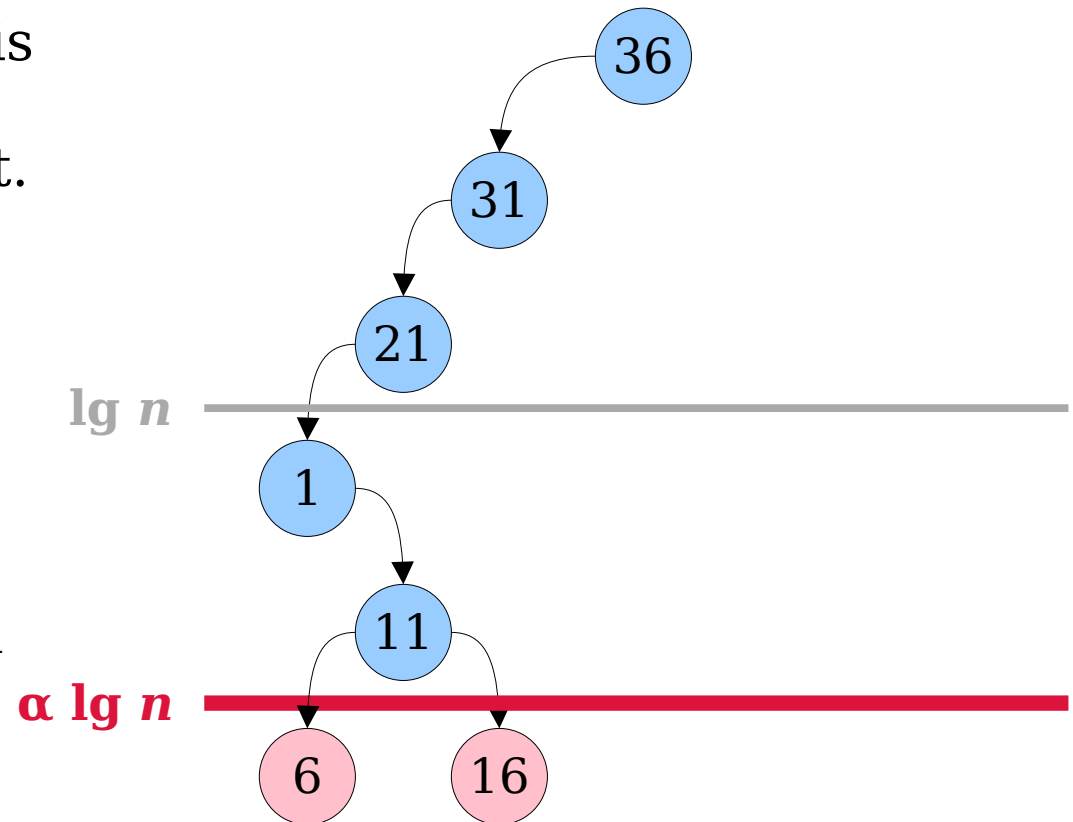
Lookup: $O(\log n)$

Insert: $O(\log n)^*$

* *amortized*

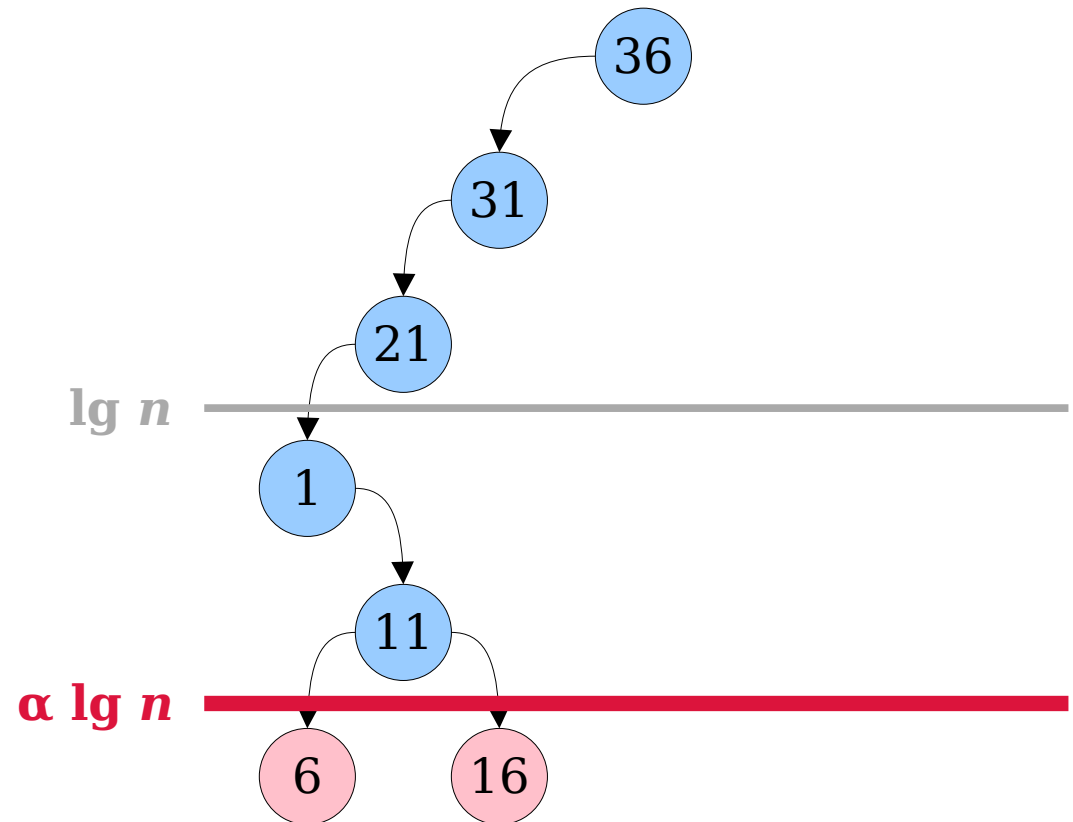
Why Deletions are Different

- In the insert-only case, we can easily detect when the height is violated, and we know which node exceeded the height limit.
- Neither of these are true with deletions.
 - Deleting one node may make an unrelated node height above the threshold.
 - Deleting one node may make multiple unrelated nodes exceed the threshold.
- **Intuition:** Deletions will require some sort of *global* rebuilding of the tree, rather than the *local* rebuilding we saw earlier.



Why Deletions are Different

- As we delete nodes from our BST, the value of $\alpha \lg n$ will decrease, but it does so slowly.
- Leaf nodes will be the first to exceed the $\alpha \lg n$ threshold.
- However, a very large number of nodes need to be deleted before non-leaves cross the threshold.
- Let's quantify this.



Why Deletions are Different

- Suppose our tree currently has n nodes in it. We'll perform some number of deletions and arrive at a tree with n_{new} nodes.
- At what value of n_{new} is it possible for non-leaf nodes to have a depth greater than $\alpha \lg n_{new}$?
- We need to solve

$$\alpha \lg n_{new} < \alpha \lg n - 1.$$

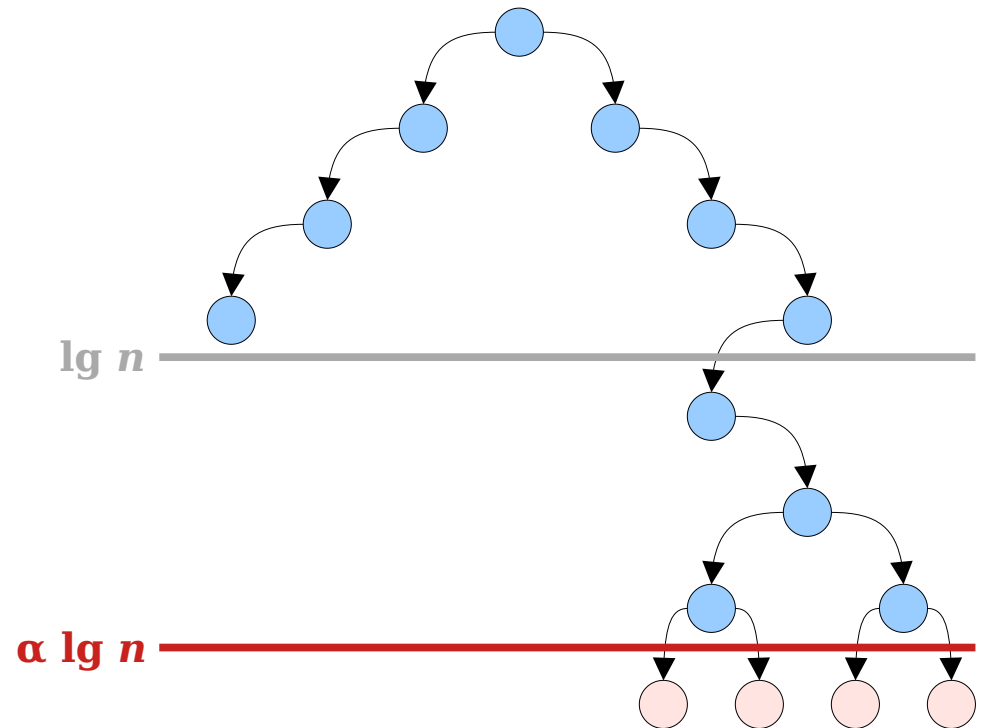
- Rearranging gives us that

$$n_{new} < n \cdot 2^{-1/\alpha}.$$

- Note that $2^{-1/\alpha} \in (1/2, 1)$ for any $\alpha > 1$.
- We need to delete at least a constant fraction (specifically, a $1 - 2^{-1/\alpha}$ fraction) of the nodes before nodes one layer above the bottom could exceed the $\alpha \lg n$ limit.

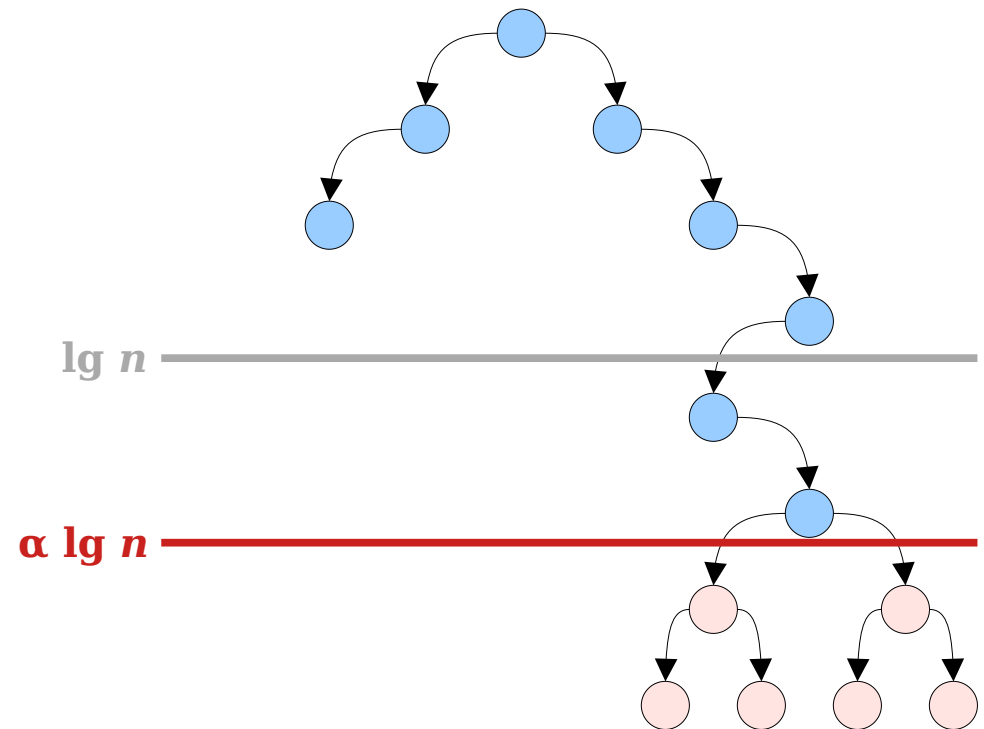
Why Deletions are Different

- **Idea:** Don't worry about rebalancing until we lose a $(1 - 2^{-1/\alpha})$ fraction of the nodes.
- Assuming we lose fewer than this many nodes, all nodes in the tree will be at depth at most $\alpha \lg n + 1$.
 - Focus on any node. Assume there were n_0 nodes at the point when the node was inserted. The node depth is then at most $\alpha \lg n_0$.
 - As long as we haven't lost at least a $(1 - 2^{-1/\alpha})$ fraction of the nodes, the current value of n is such that $\alpha \lg n \geq \alpha \lg n_0 - 1$.
- This still gives us lookups that run in time $O(\log n)$, and insertions still work properly.



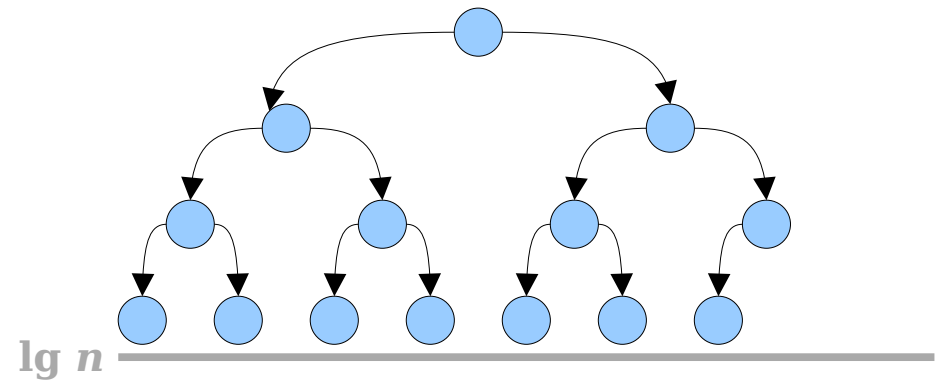
Why Deletions are Different

- Once we've lost a $(1 - 2^{-1/\alpha})$ fraction of the nodes, we need to worry about rebalancing the tree.
- We won't know much about the tree shape.
 - It could have a large number of deep nodes.
 - It could be perfectly balanced.
- **Idea:** Don't try to analyze the tree. Just rebuild the entire tree from scratch.



Why Deletions are Different

- Once we've lost a $(1 - 2^{-1/\alpha})$ fraction of the nodes, we need to worry about rebalancing the tree.
- We won't know much about the tree shape.
 - It could have a large number of deep nodes.
 - It could be perfectly balanced.
- **Idea:** Don't try to analyze the tree. Just rebuild the entire tree from scratch.



$\alpha \lg n$

Scapegoat Tree Deletions

- Here's how this approach will work.
 - Keep track of the maximum number of nodes the tree has had since it was last globally rebuilt. (Call this n_{max}).
 - If the number of nodes drops to a $n_{max} \cdot 2^{-1/\alpha}$, globally rebuild the tree as a perfectly balanced tree, then reset n_{max} to the current tree size.
- Although rebuilding the tree is an expensive operation, intuitively we expect to be able to “backcharge” the work to the lazy delete operations that triggered it.

Scapegoat Tree Deletions

- Our goal now is to work out the amortized cost of doing global rebuilds on deletions.
- **Recall:** Our current potential function is

$$\Phi = \sum_v \phi(v),$$

which we chose to make the cost of local rebuilds on insertions amortize away.

- We need to adjust this potential function to account for the fact that deleted nodes slowly lead us to do a global rebuild of the whole tree.
- **Idea:** Change our potential to

$$\Phi = D + \sum_v \phi(v),$$

where D is the number of deletions that have been performed since we last did a global rebuild.

Scapegoat Tree Deletions

- What is the amortized cost of a deletion when we don't trigger a global rebuild?
- Actual cost: $O(\log n)$, since the tree height is at most $\alpha \lg n + 1$.
- Change in potential (recall that $\Phi = D + \sum_v \phi(v)$):
 - D increases by one, since we've performed a deletion.
 - $\phi(v)$ changes by at most two for each node on the access path of the removed node, and there are $O(\log n)$ such nodes.
 - Net change: $O(\log n)$.
- Amortized cost:

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

Scapegoat Tree Deletions

- What is the amortized cost of a deletion when we *do* trigger a global rebuild?
- We picked

$$\Phi = D + \sum_v \phi'(v).$$

- After the rebuild, we have $\sum_v \phi'(v) = 0$. Therefore, there is an unknown but nonpositive change in potential for this term.
- How much does D change?
 - At the point where we start the rebuild, we have $n = n_{\max} \cdot 2^{-1/\alpha}$ nodes left in the tree.
 - This means that $D \geq n_{\max} \cdot (1 - 2^{-1/\alpha})$.
 - Rewriting in terms of n , this means $D \geq n \cdot (2^{1/\alpha} - 1) = \Omega(n)$.
 - Since after this step we drop D to zero, we have $\Delta D \leq -\Omega(n)$.
- Overall, we have $\Delta\Phi \leq -\Omega(n)$.

Scapegoat Tree Deletions

- Actual cost of the deletion:
 - $O(\log n)$ for the actual deletion logic.
 - $O(n)$ to rebuild the tree.
- Amortized cost:
$$O(\log n) + O(n) - k \cdot \Omega(n).$$
- As before, we can tune k based on the hidden constant factors in the O and Ω terms to make them cancel out and leave behind an amortized cost of **$O(\log n)$** .

The Final Scorecard

- Here's the final scorecard for our scapegoat tree.
- It matches the time bounds we'd expect of a red/black tree, in an amortized sense, with a *dramatically* simpler implementation.
- This gives a sense of just how useful a technique amortization can be!

Scapegoat Tree

Lookup: $O(\log n)$

Insert: $O(\log n)^*$

Delete: $O(\log n)^*$

* *amortized*

Further Exploration

- I haven't seen much work done into building an optimized scapegoat tree implementation. How fast can you make this idea work? Is it competitive with a red/black tree?
- We've treated α as a constant. What if you allow it to vary based on the workflow (say, decreasing it as more lookups happen and increasing it as more deletions/insertions happen)? A past CS166 project team looked into this in 2014, and I'm curious to see it on modern hardware.
- Are there other, less aggressive strategies besides rebuilding the scapegoat subtree that can be used to restore balance?
- Are there other ways of picking a scapegoat node that work better in practice? For example, could you pick a scapegoat higher up in the tree that would do a better job rebalancing things?
- What is the practical time/space tradeoff between the two approaches for calculating $\text{size}(v)$ when finding a scapegoat?
- The version of scapegoat trees described here is a hybrid between two approaches: the original developed by Galperin and Rivest and a simplification by Jeff Erickson. The Galperin/Rivest version has tighter structural constraints, while Erickson's version uses a different deletion strategy. Can you remix these ideas in other ways?
- Because there are no rotations, it should be way easier to augment a scapegoat tree than it is to augment a red/black tree. Can you find a weaker set of requirements for augmenting a BST if you assume the tree you're augmenting is a scapegoat tree?

Next Time

- ***Tournament Heaps***
 - A simple and fast priority queue.
- ***Lazy Tournament Heaps***
 - An asymptotically faster priority queue.

Scapegoat Trees

Outline for Today

- ***Recap from Last Time***
 - What is amortization, again?
- ***Lazy Balanced Trees***
 - Messes are okay, up to a point.
- ***Lazy Tree Insertions***
 - Deferring updates until they're needed.
- ***Lazy Tree Deletions***
 - And some associated subtleties.

Recap from Last Time

Amortized Analysis

- We will assign amortized costs to each operation such that

$$\sum \text{amortized-cost} \geq \sum \text{real-cost}$$

- To do so, define a **potential function** Φ such that

- Φ measures how “messy” the data structure is,
- $\Phi_{start} = 0$, and
- $\Phi \geq 0$.

- Then, define amortized costs of operations as

$$\text{amortized-cost} = \text{real-cost} + k \cdot \Delta\Phi$$

for a choice of k under our control.

- Intuitively:

- If an operation makes a mess that needs to be cleaned up later, its amortized cost will be higher than its original cost.
- If an operation cleans up a mess, its amortized cost will be lower than its real cost.

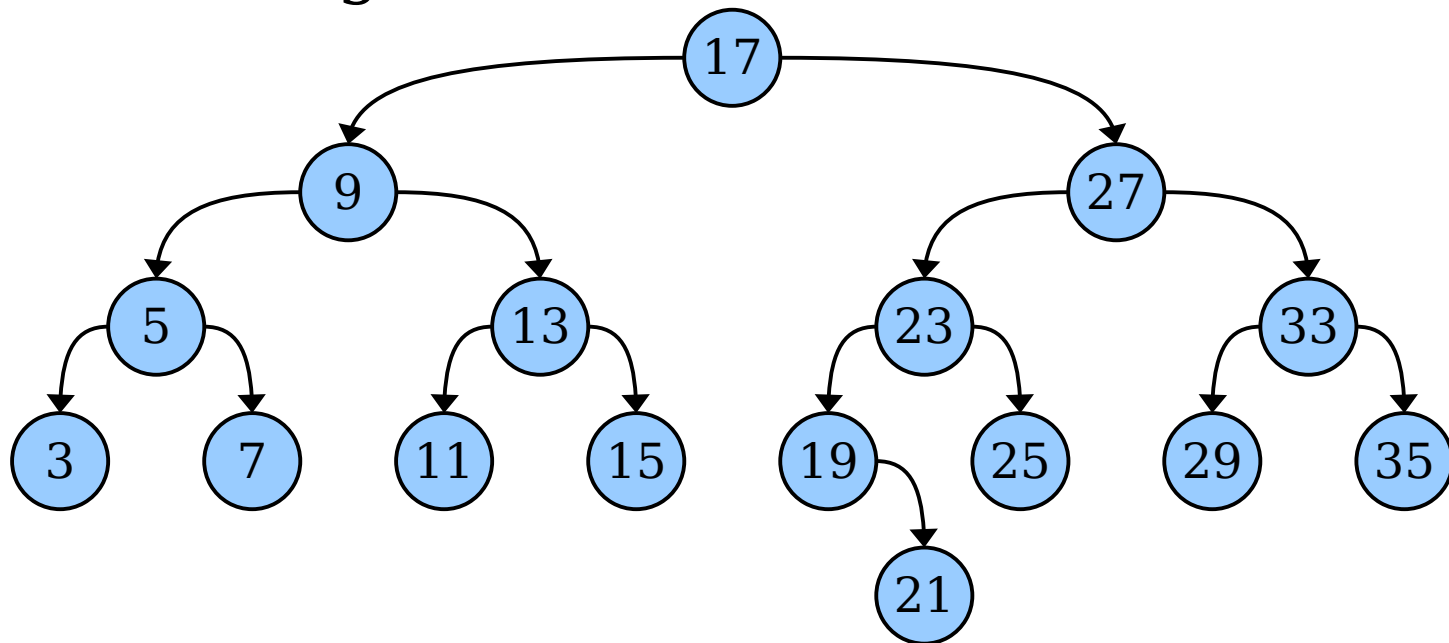
New Stuff!

Balanced Trees

- The red/black trees we explored earlier are worst-case efficient and guaranteed to have a height of $O(\log n)$.
- However, explaining how they work and deriving the basic insertion rules took two lectures – and we still didn't finish covering all cases.
- ***Goal for today:*** Find a simpler way to keep a tree balanced, under the assumption we're okay with amortized-efficient rather than worst-case efficient lookups.

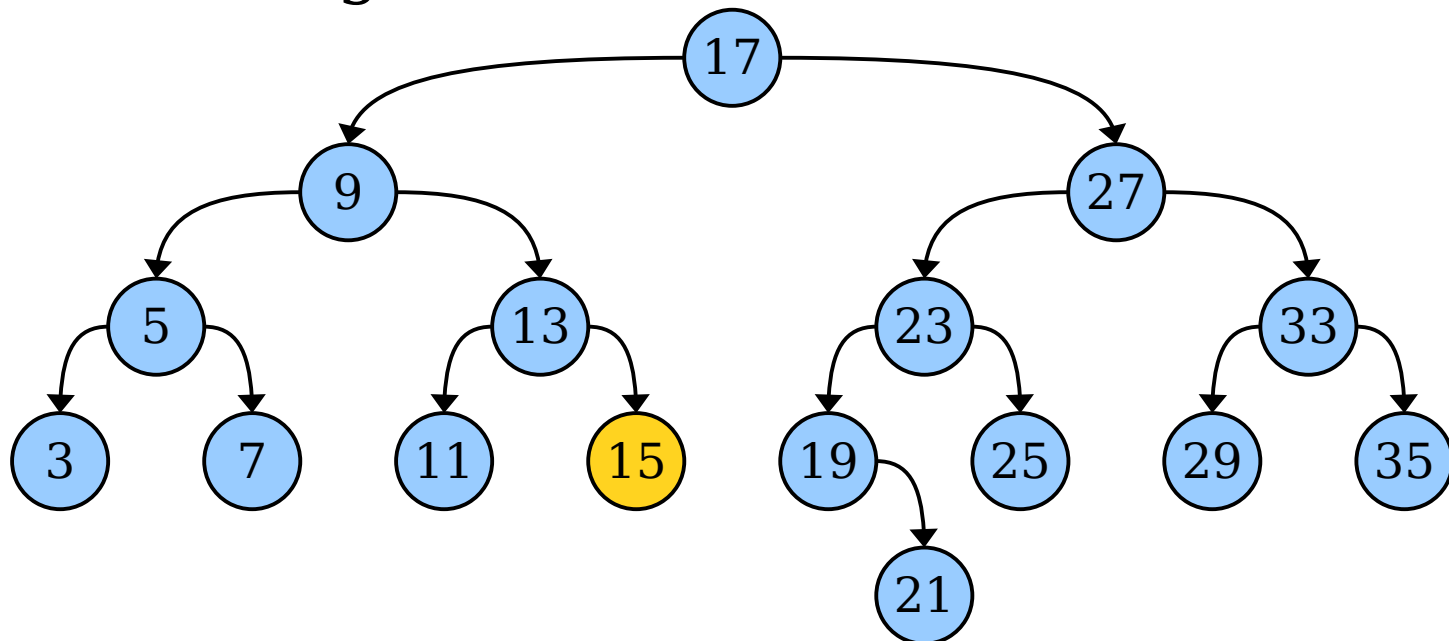
On $O(\log n)$ Height

- A perfectly-balanced binary search tree with $n > 0$ nodes has height at most $\lg n$.
 - ($\lg n$ denotes $\log_2 n$.)
- However, this tree shape is difficult to maintain: a single insertion or deletion might require a lot of node reshuffling.



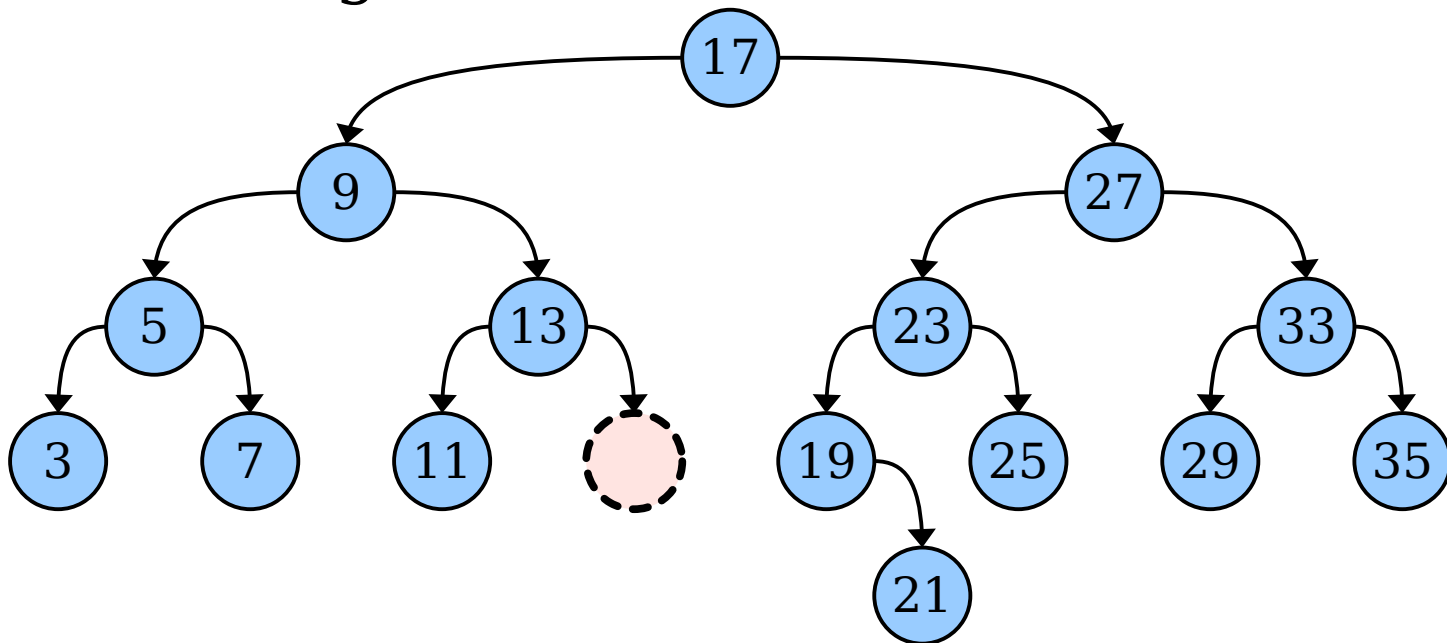
On $O(\log n)$ Height

- A perfectly-balanced binary search tree with $n > 0$ nodes has height at most $\lg n$.
 - ($\lg n$ denotes $\log_2 n$.)
- However, this tree shape is difficult to maintain: a single insertion or deletion might require a lot of node reshuffling.



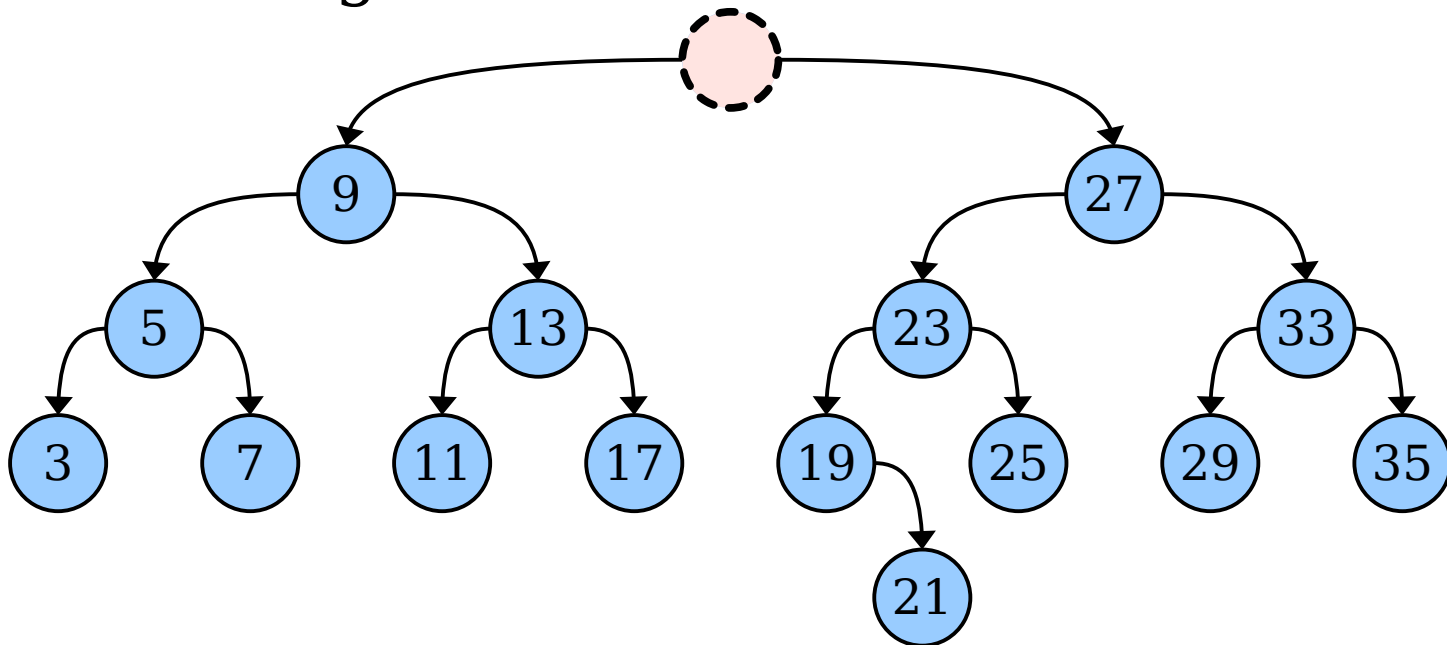
On $O(\log n)$ Height

- A perfectly-balanced binary search tree with $n > 0$ nodes has height at most $\lg n$.
 - ($\lg n$ denotes $\log_2 n$.)
- However, this tree shape is difficult to maintain: a single insertion or deletion might require a lot of node reshuffling.



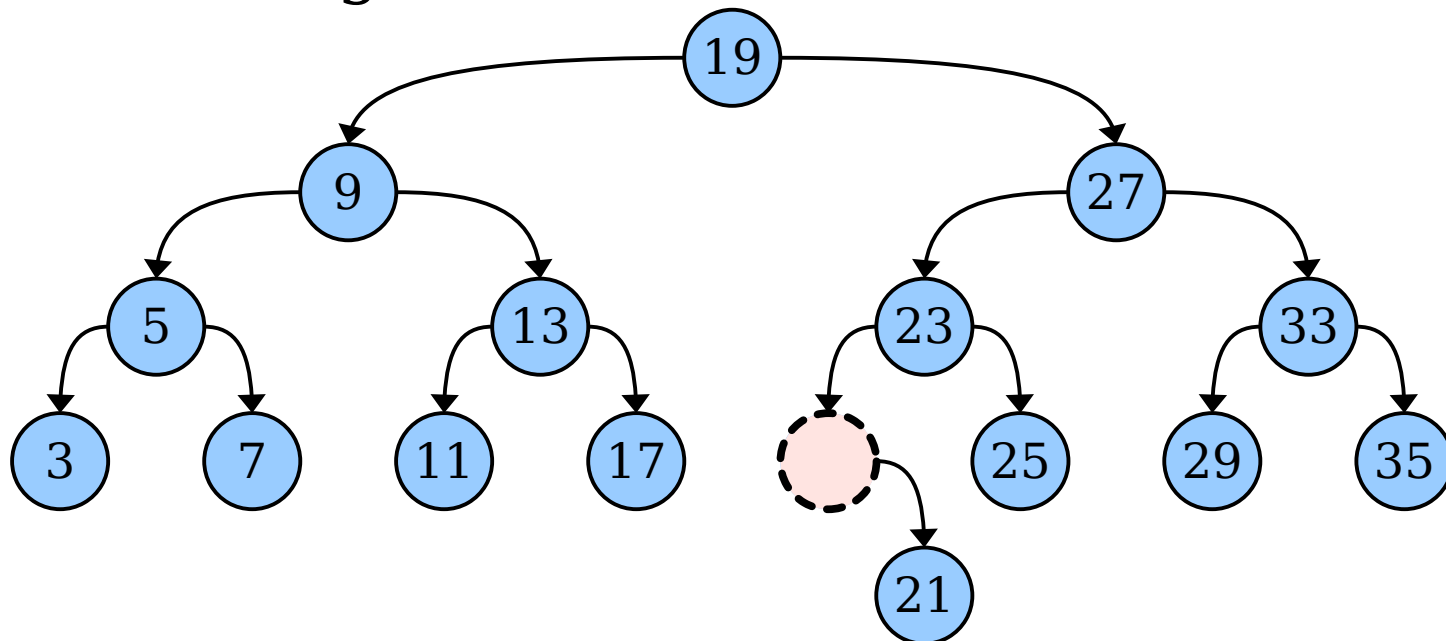
On $O(\log n)$ Height

- A perfectly-balanced binary search tree with $n > 0$ nodes has height at most $\lg n$.
 - ($\lg n$ denotes $\log_2 n$.)
- However, this tree shape is difficult to maintain: a single insertion or deletion might require a lot of node reshuffling.



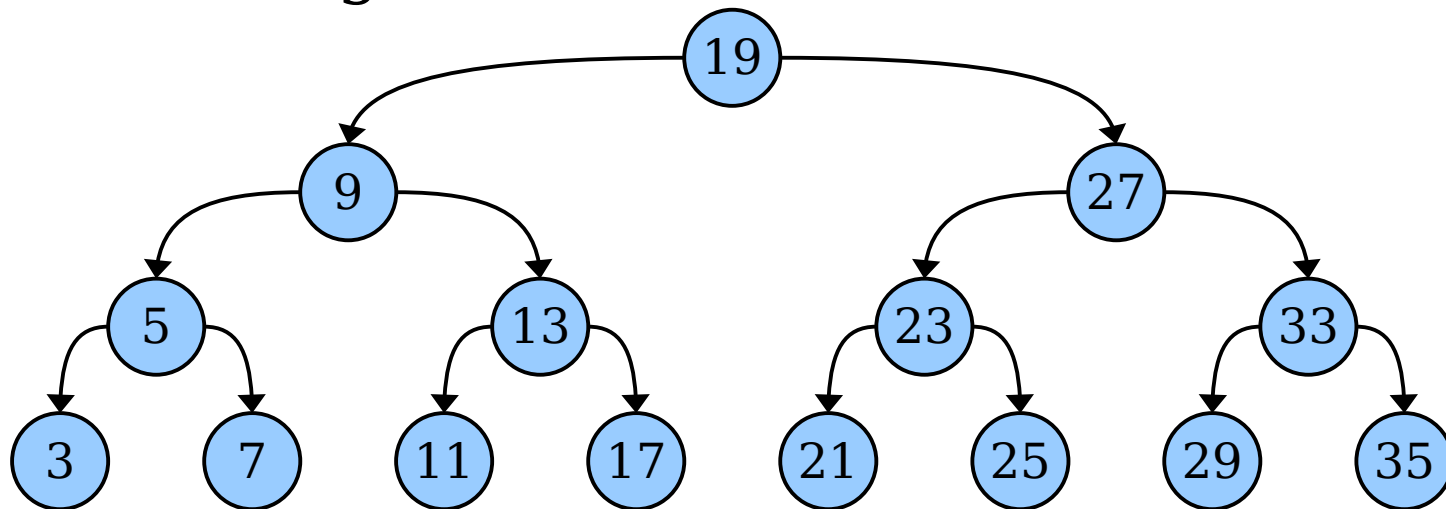
On $O(\log n)$ Height

- A perfectly-balanced binary search tree with $n > 0$ nodes has height at most $\lg n$.
 - ($\lg n$ denotes $\log_2 n$.)
- However, this tree shape is difficult to maintain: a single insertion or deletion might require a lot of node reshuffling.



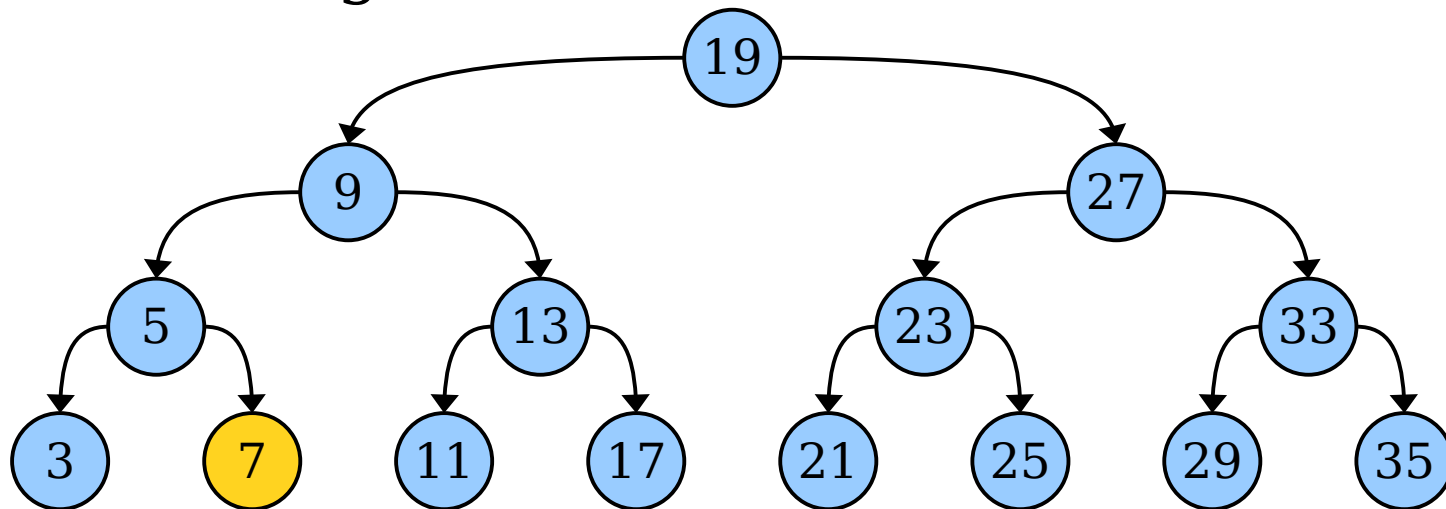
On $O(\log n)$ Height

- A perfectly-balanced binary search tree with $n > 0$ nodes has height at most $\lg n$.
 - ($\lg n$ denotes $\log_2 n$.)
- However, this tree shape is difficult to maintain: a single insertion or deletion might require a lot of node reshuffling.



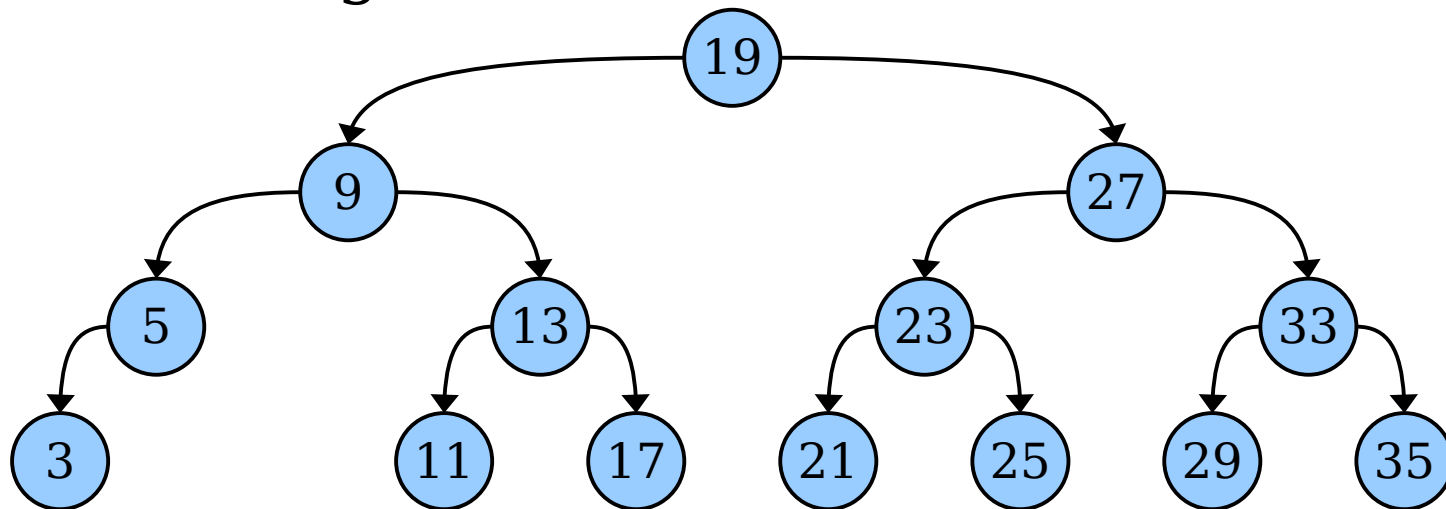
On $O(\log n)$ Height

- A perfectly-balanced binary search tree with $n > 0$ nodes has height at most $\lg n$.
 - ($\lg n$ denotes $\log_2 n$.)
- However, this tree shape is difficult to maintain: a single insertion or deletion might require a lot of node reshuffling.



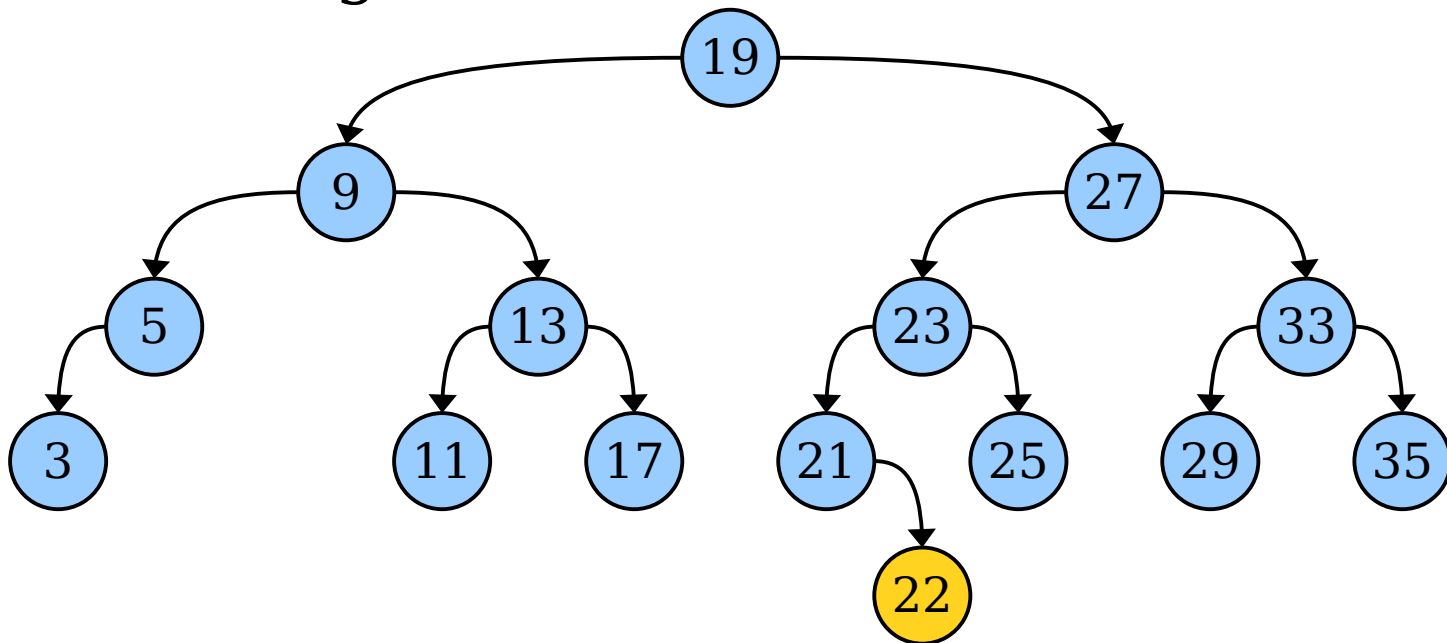
On $O(\log n)$ Height

- A perfectly-balanced binary search tree with $n > 0$ nodes has height at most $\lg n$.
 - ($\lg n$ denotes $\log_2 n$.)
- However, this tree shape is difficult to maintain: a single insertion or deletion might require a lot of node reshuffling.



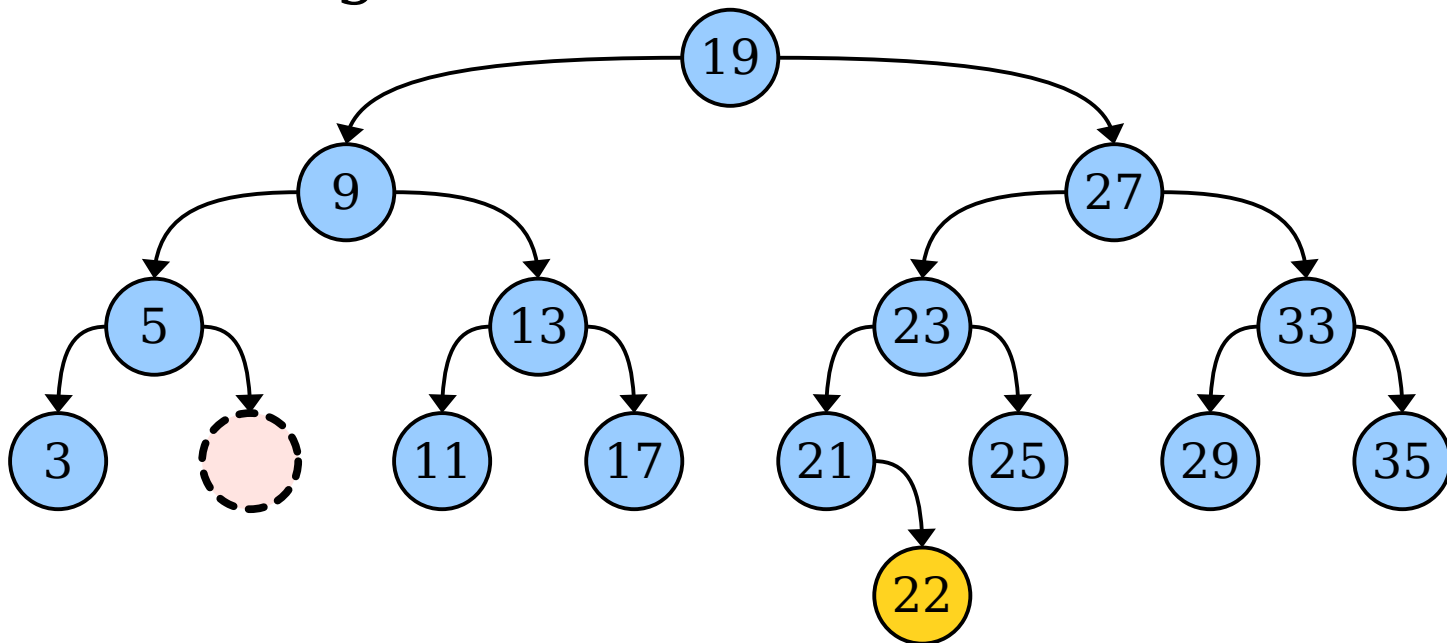
On $O(\log n)$ Height

- A perfectly-balanced binary search tree with $n > 0$ nodes has height at most $\lg n$.
 - ($\lg n$ denotes $\log_2 n$.)
- However, this tree shape is difficult to maintain: a single insertion or deletion might require a lot of node reshuffling.



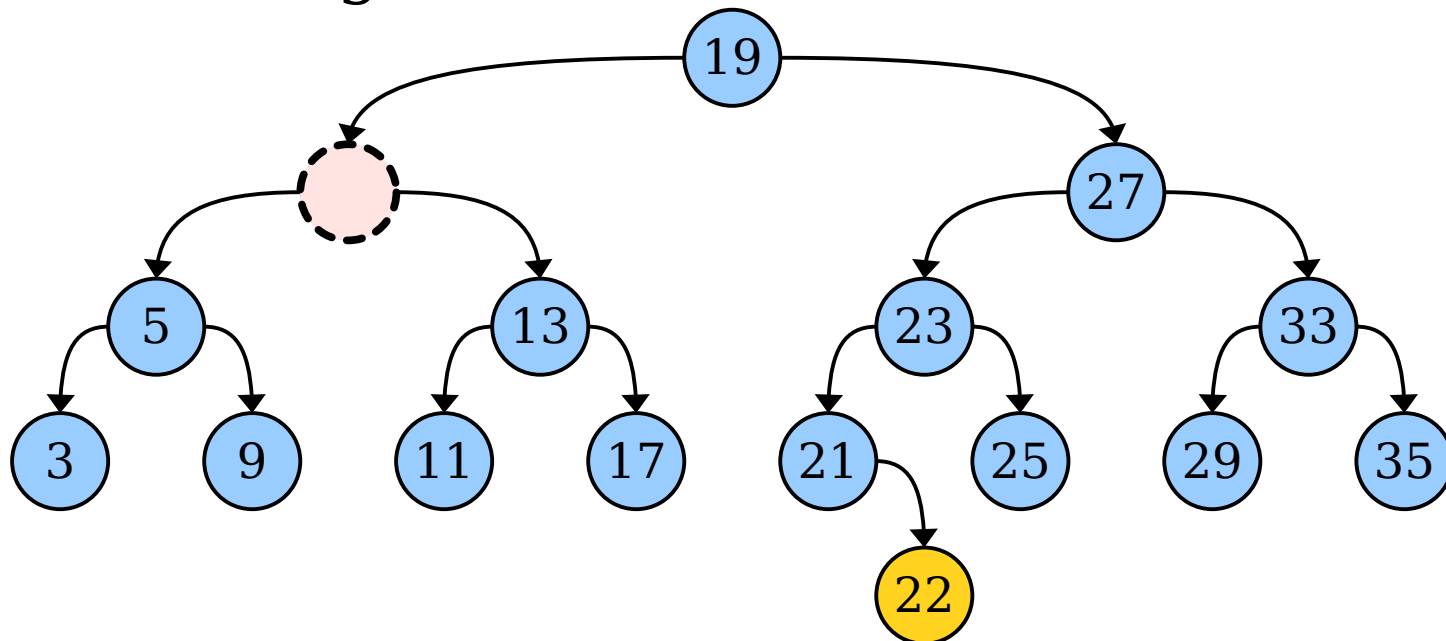
On $O(\log n)$ Height

- A perfectly-balanced binary search tree with $n > 0$ nodes has height at most $\lg n$.
 - ($\lg n$ denotes $\log_2 n$.)
- However, this tree shape is difficult to maintain: a single insertion or deletion might require a lot of node reshuffling.



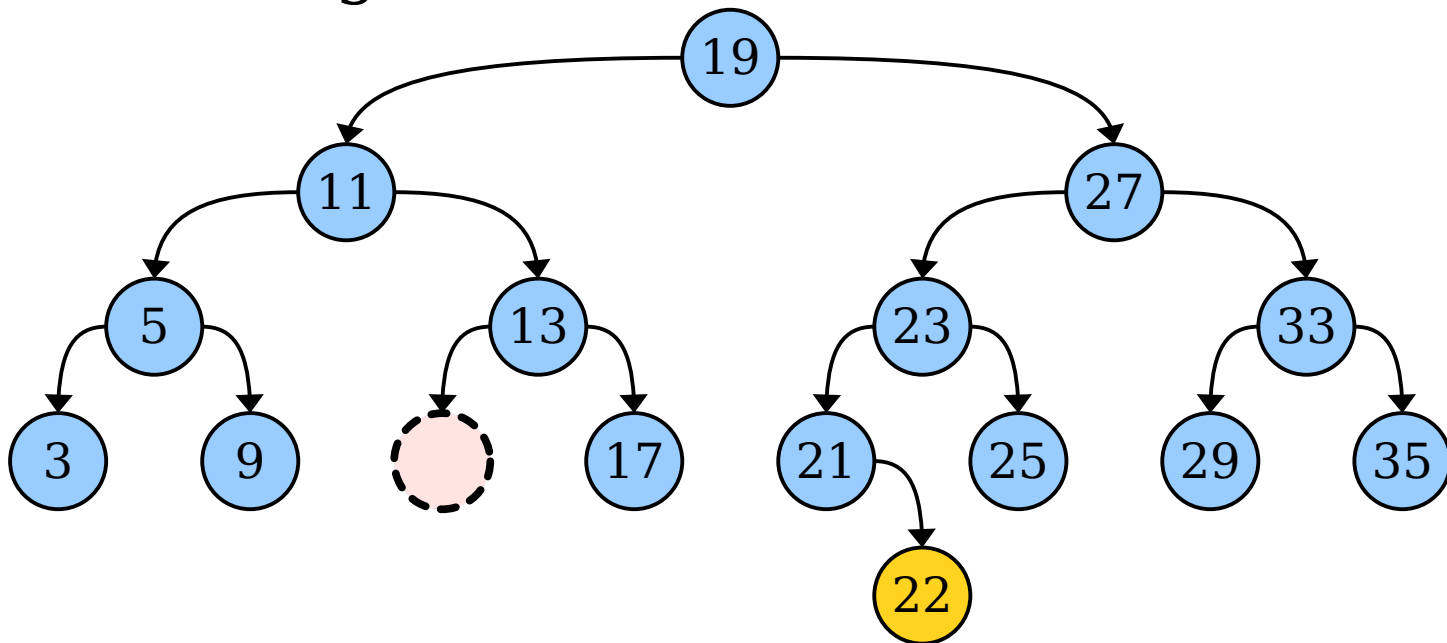
On $O(\log n)$ Height

- A perfectly-balanced binary search tree with $n > 0$ nodes has height at most $\lg n$.
 - ($\lg n$ denotes $\log_2 n$.)
- However, this tree shape is difficult to maintain: a single insertion or deletion might require a lot of node reshuffling.



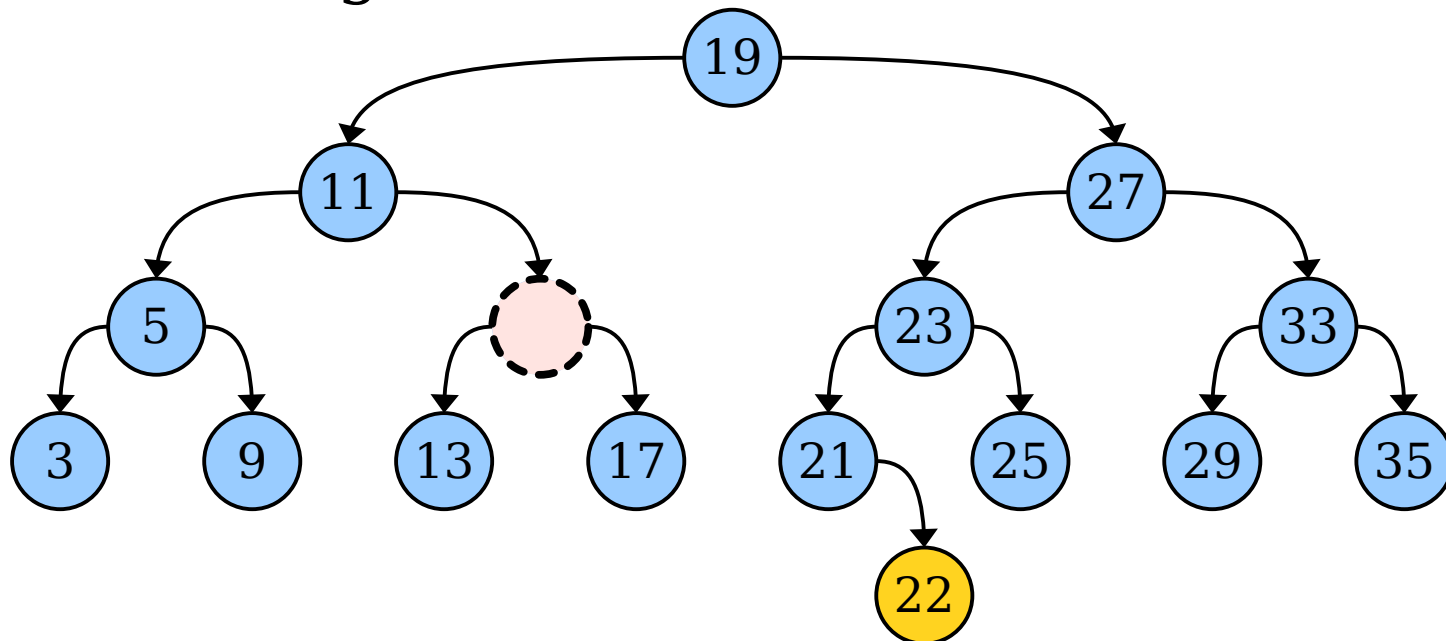
On $O(\log n)$ Height

- A perfectly-balanced binary search tree with $n > 0$ nodes has height at most $\lg n$.
 - ($\lg n$ denotes $\log_2 n$.)
- However, this tree shape is difficult to maintain: a single insertion or deletion might require a lot of node reshuffling.



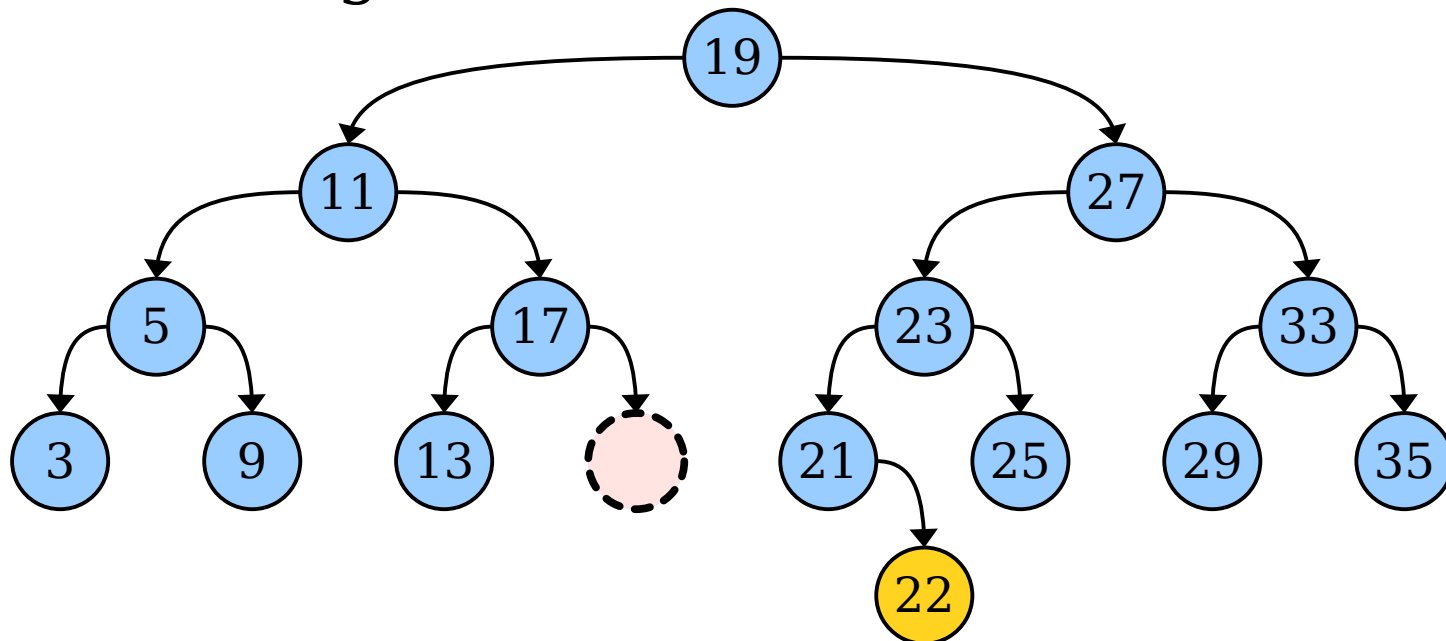
On $O(\log n)$ Height

- A perfectly-balanced binary search tree with $n > 0$ nodes has height at most $\lg n$.
 - ($\lg n$ denotes $\log_2 n$.)
- However, this tree shape is difficult to maintain: a single insertion or deletion might require a lot of node reshuffling.



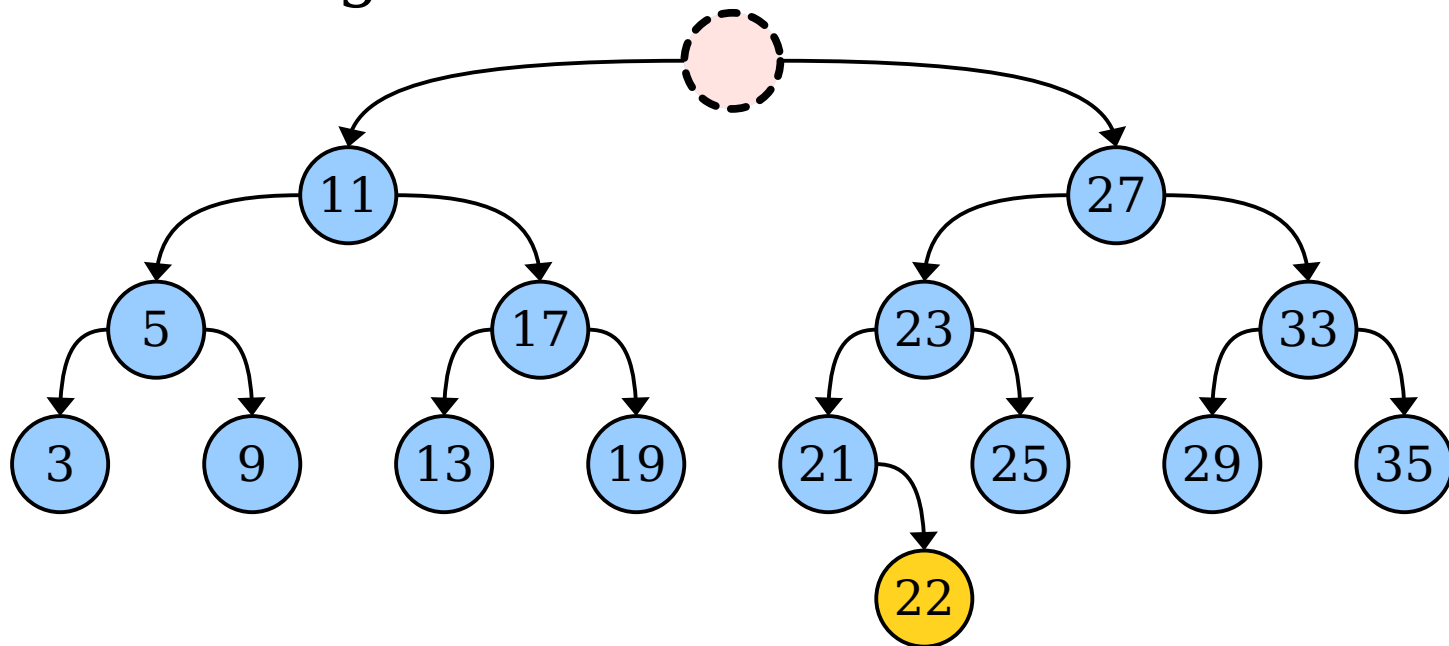
On $O(\log n)$ Height

- A perfectly-balanced binary search tree with $n > 0$ nodes has height at most $\lg n$.
 - ($\lg n$ denotes $\log_2 n$.)
- However, this tree shape is difficult to maintain: a single insertion or deletion might require a lot of node reshuffling.



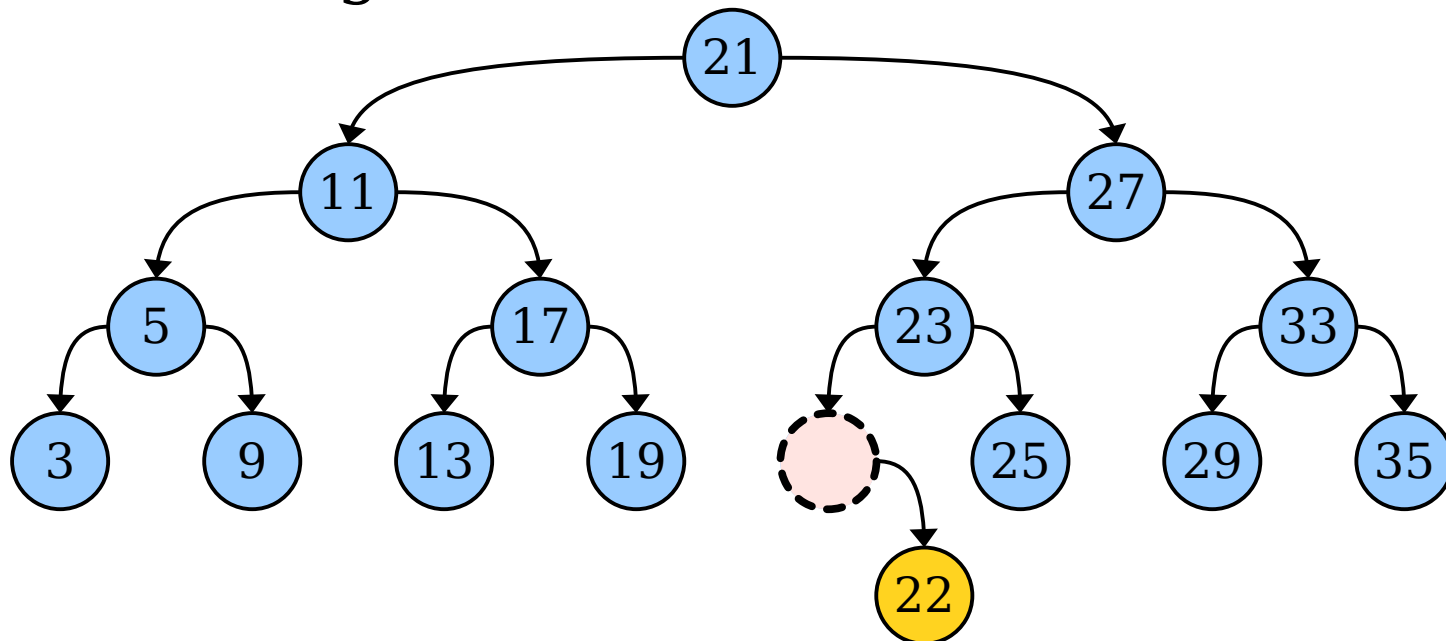
On $O(\log n)$ Height

- A perfectly-balanced binary search tree with $n > 0$ nodes has height at most $\lg n$.
 - ($\lg n$ denotes $\log_2 n$.)
- However, this tree shape is difficult to maintain: a single insertion or deletion might require a lot of node reshuffling.



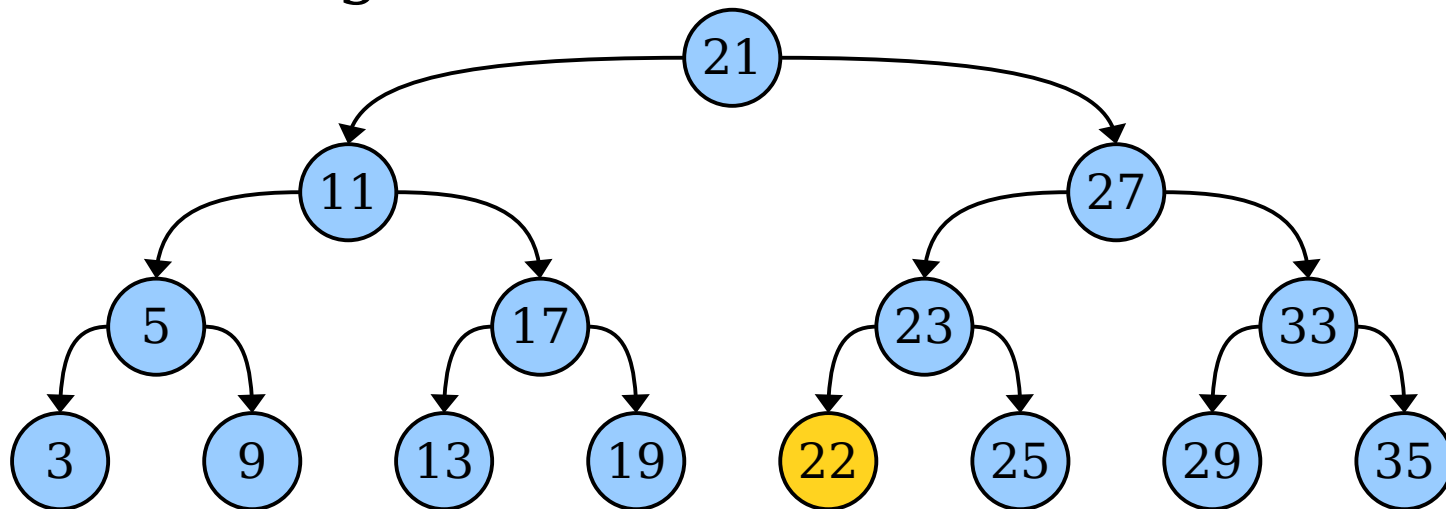
On $O(\log n)$ Height

- A perfectly-balanced binary search tree with $n > 0$ nodes has height at most $\lg n$.
 - ($\lg n$ denotes $\log_2 n$.)
- However, this tree shape is difficult to maintain: a single insertion or deletion might require a lot of node reshuffling.



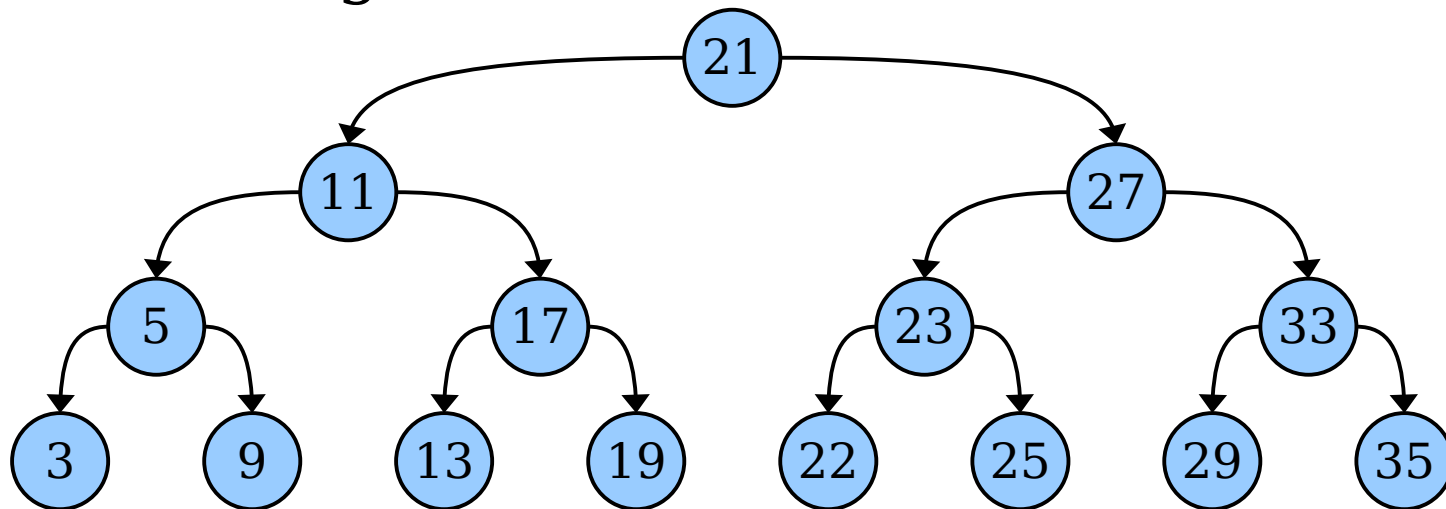
On $O(\log n)$ Height

- A perfectly-balanced binary search tree with $n > 0$ nodes has height at most $\lg n$.
 - ($\lg n$ denotes $\log_2 n$.)
- However, this tree shape is difficult to maintain: a single insertion or deletion might require a lot of node reshuffling.



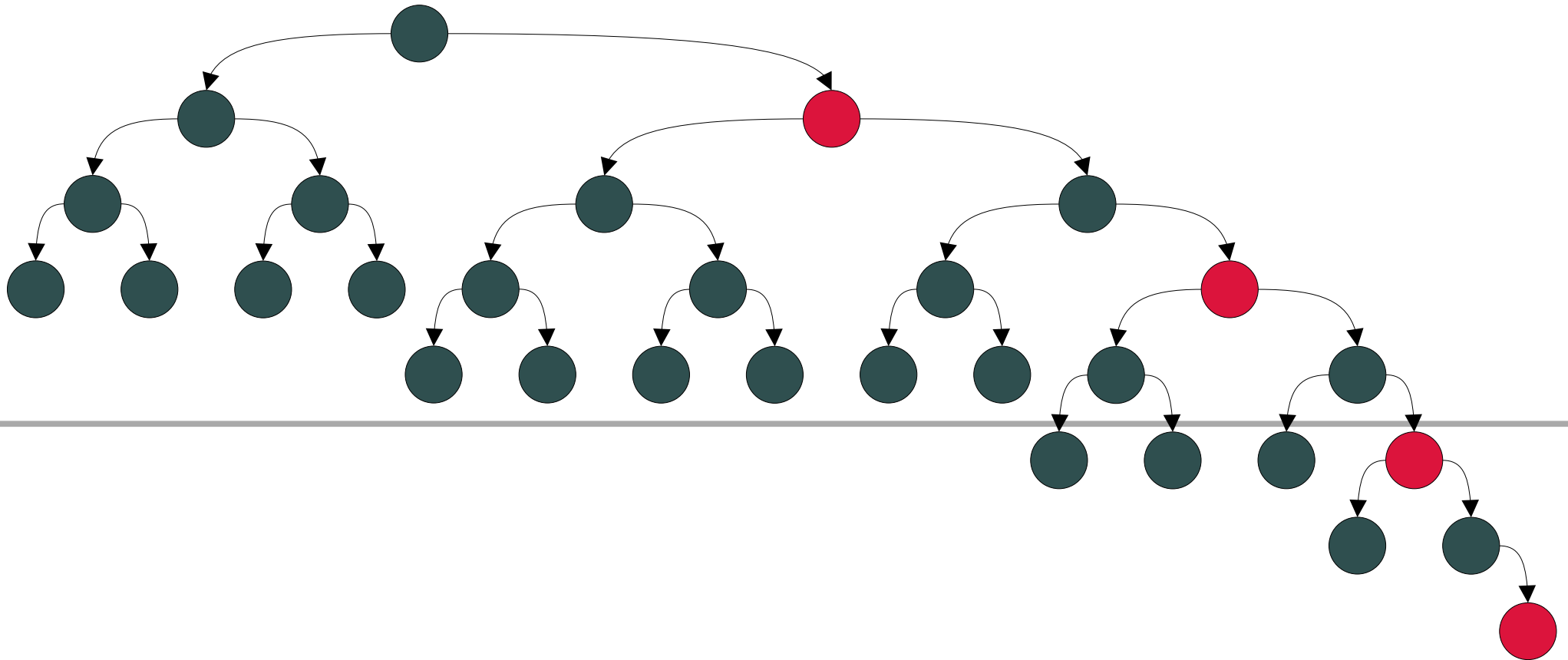
On $O(\log n)$ Height

- A perfectly-balanced binary search tree with $n > 0$ nodes has height at most $\lg n$.
 - ($\lg n$ denotes $\log_2 n$.)
- However, this tree shape is difficult to maintain: a single insertion or deletion might require a lot of node reshuffling.



On $O(\log n)$ Height

- To speed up logic after insertions or deletions, most balanced BSTs only guarantee height of multiple of $\lg n$.
- For example, red/black trees have height at most (roughly) $2 \lg n$ in the worst case.

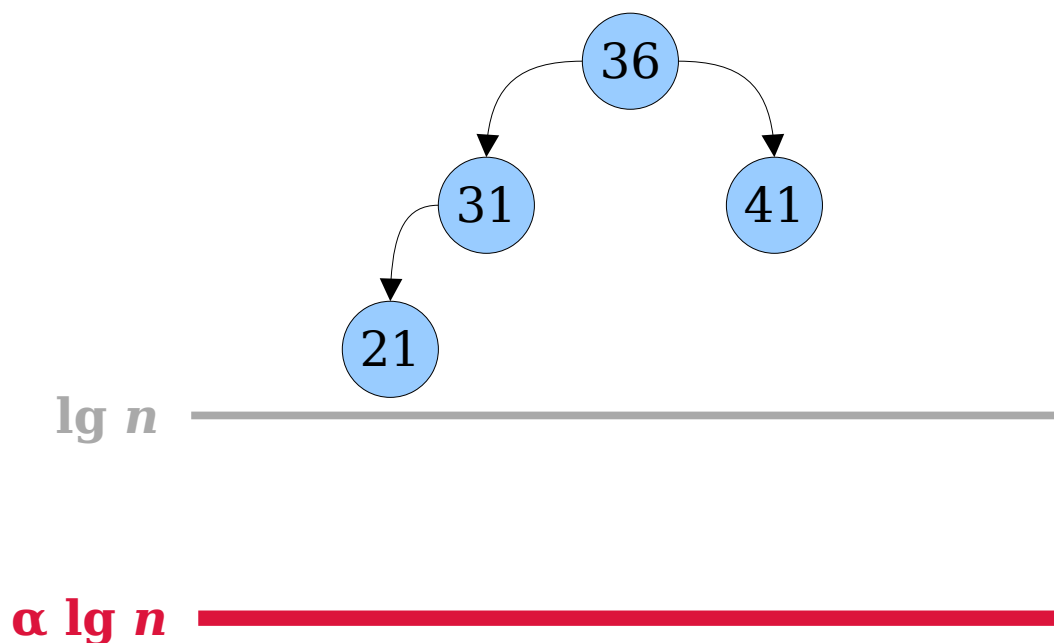


On $O(\log n)$ Height

- We're already comfortable with trees whose heights are $\alpha \lg n$ for some $\alpha > 1$.
- **Question:** Can we design a balanced tree purely based on this restriction, without any other structural constraints?

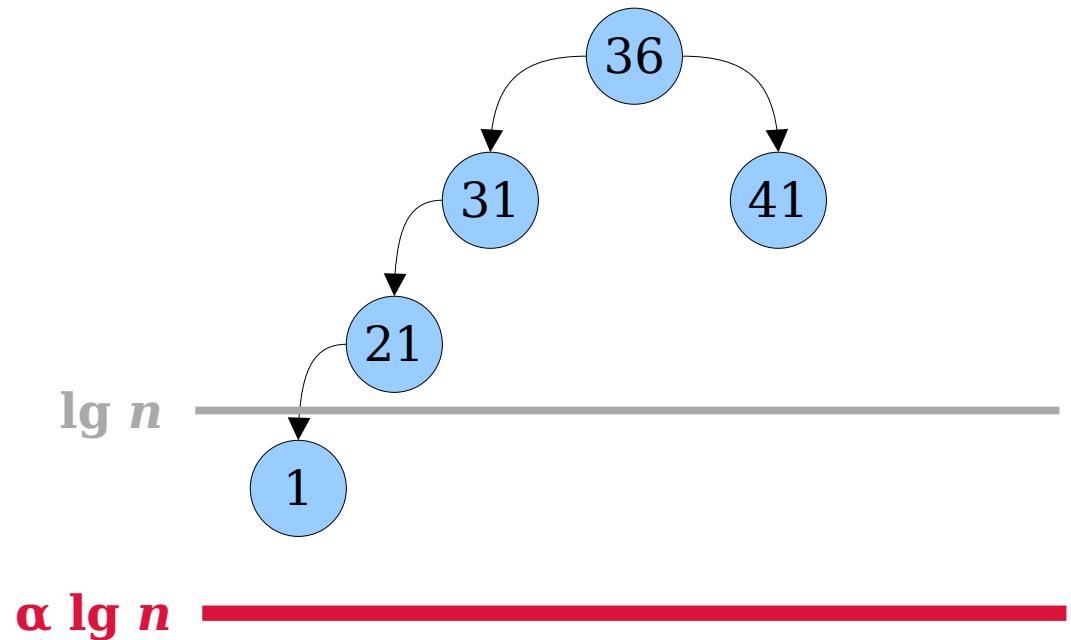
Adding Slack Space

- Pick a fixed constant $\alpha > 1$.
- Set the maximum height on our tree to $\alpha \lg n$.
- As long as we don't exceed this maximum height, all operations on our BST will run in time $O(\log n)$, and we don't really care about the shape of the tree.



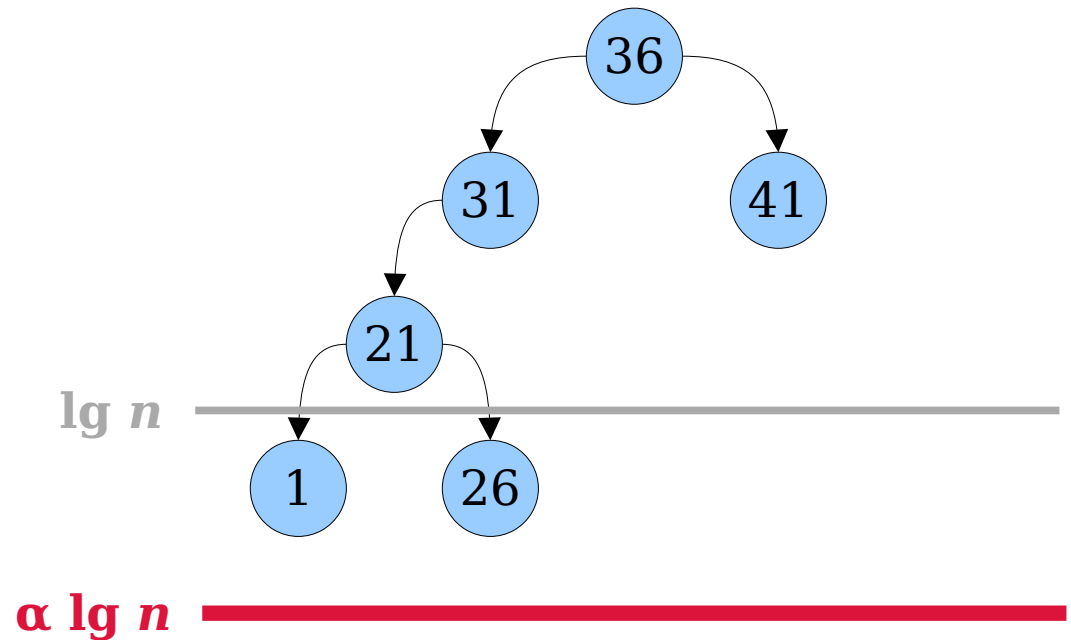
Adding Slack Space

- Pick a fixed constant $\alpha > 1$.
- Set the maximum height on our tree to $\alpha \lg n$.
- As long as we don't exceed this maximum height, all operations on our BST will run in time $O(\log n)$, and we don't really care about the shape of the tree.



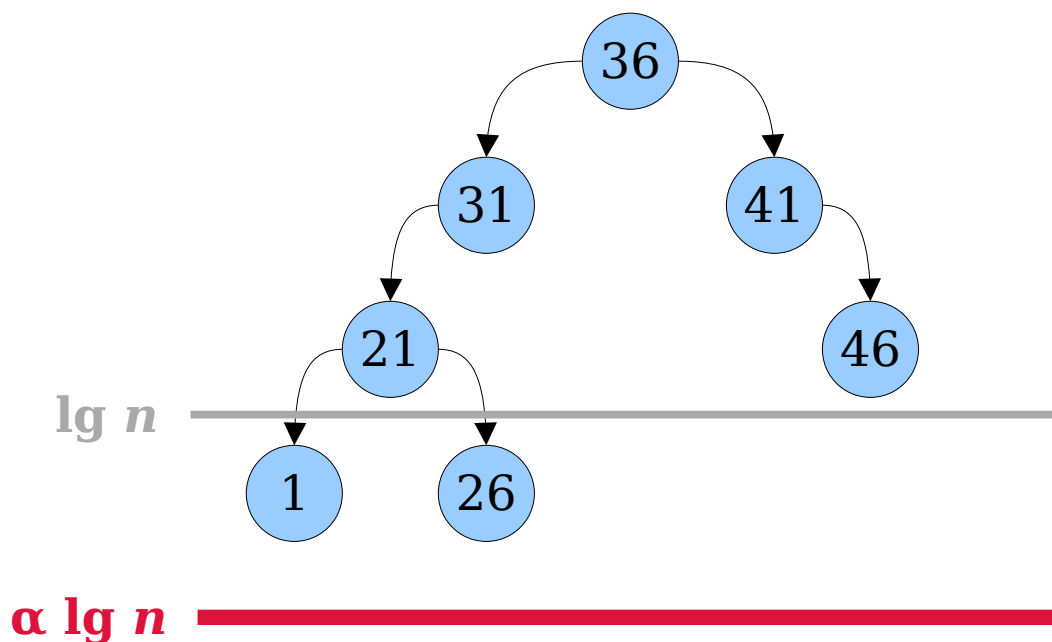
Adding Slack Space

- Pick a fixed constant $\alpha > 1$.
- Set the maximum height on our tree to $\alpha \lg n$.
- As long as we don't exceed this maximum height, all operations on our BST will run in time $O(\log n)$, and we don't really care about the shape of the tree.



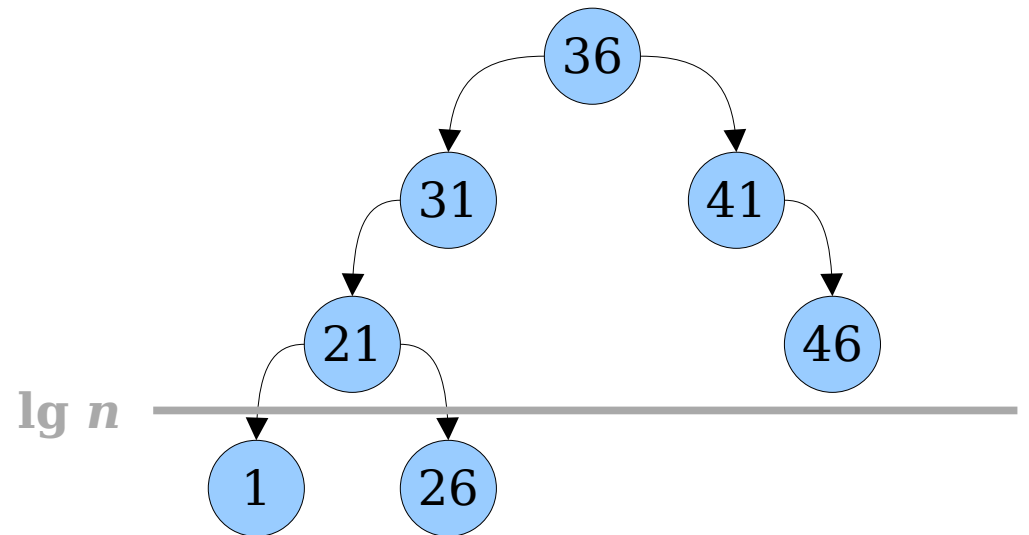
Adding Slack Space

- Pick a fixed constant $\alpha > 1$.
- Set the maximum height on our tree to $\alpha \lg n$.
- As long as we don't exceed this maximum height, all operations on our BST will run in time $O(\log n)$, and we don't really care about the shape of the tree.



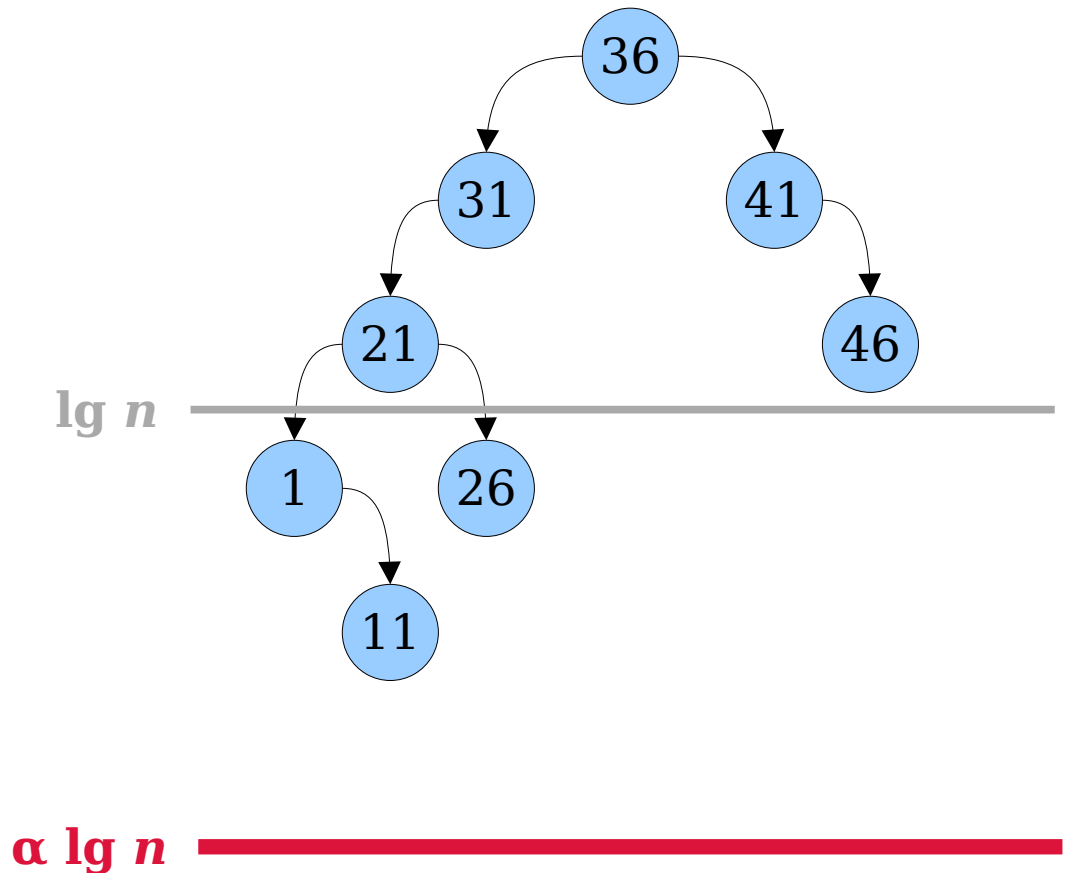
Adding Slack Space

- Pick a fixed constant $\alpha > 1$.
- Set the maximum height on our tree to $\alpha \lg n$.
- As long as we don't exceed this maximum height, all operations on our BST will run in time $O(\log n)$, and we don't really care about the shape of the tree.



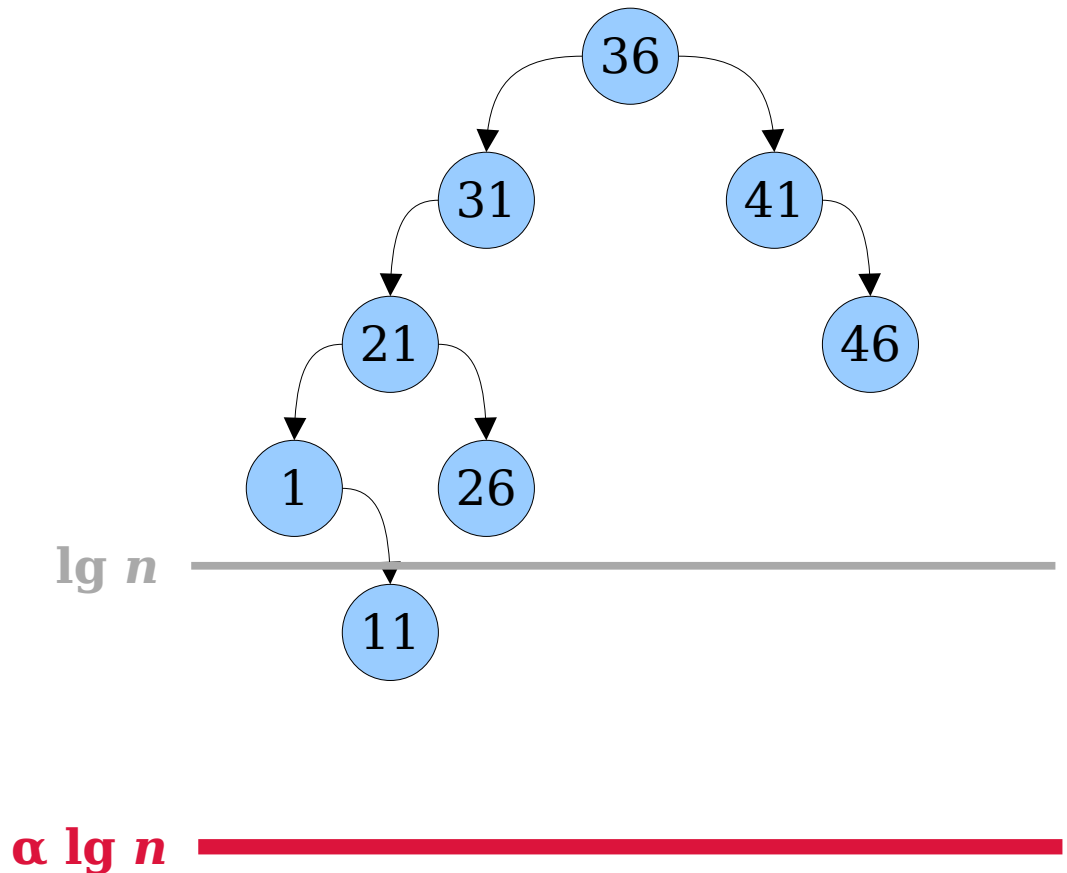
Adding Slack Space

- Pick a fixed constant $\alpha > 1$.
- Set the maximum height on our tree to $\alpha \lg n$.
- As long as we don't exceed this maximum height, all operations on our BST will run in time $O(\log n)$, and we don't really care about the shape of the tree.



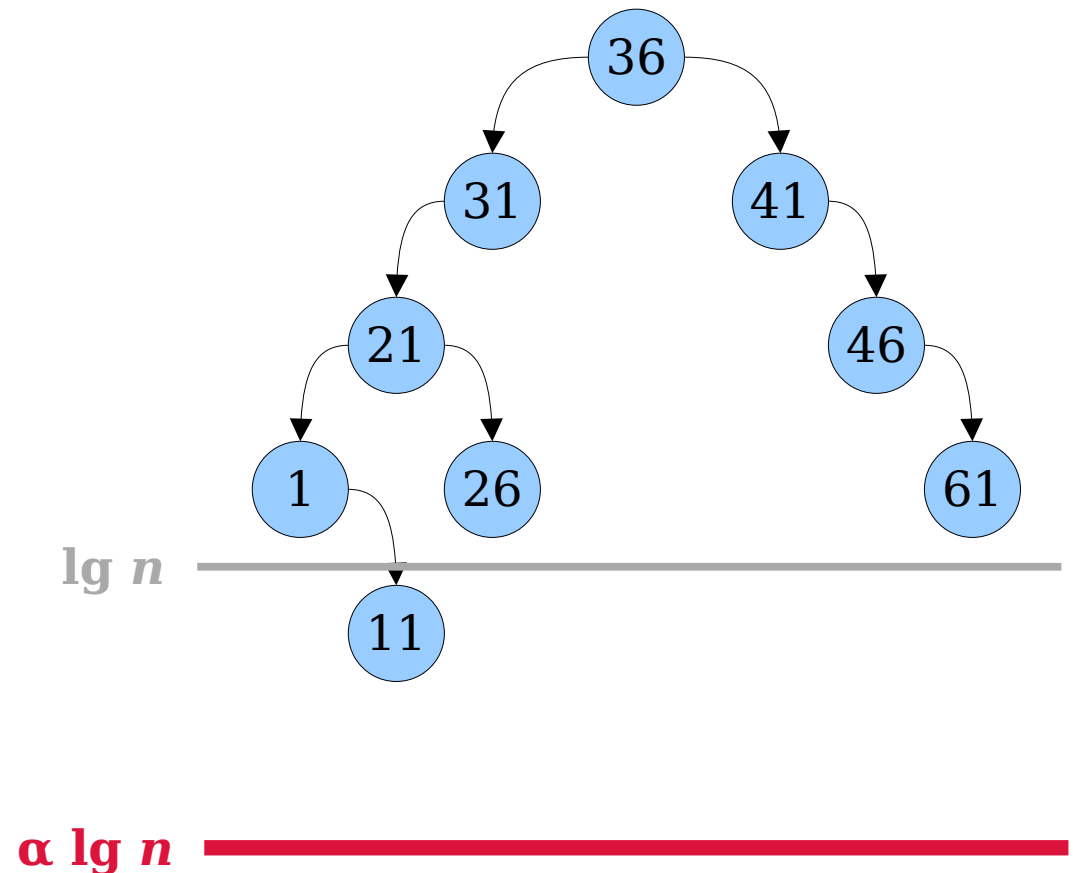
Adding Slack Space

- Pick a fixed constant $\alpha > 1$.
- Set the maximum height on our tree to $\alpha \lg n$.
- As long as we don't exceed this maximum height, all operations on our BST will run in time $O(\log n)$, and we don't really care about the shape of the tree.



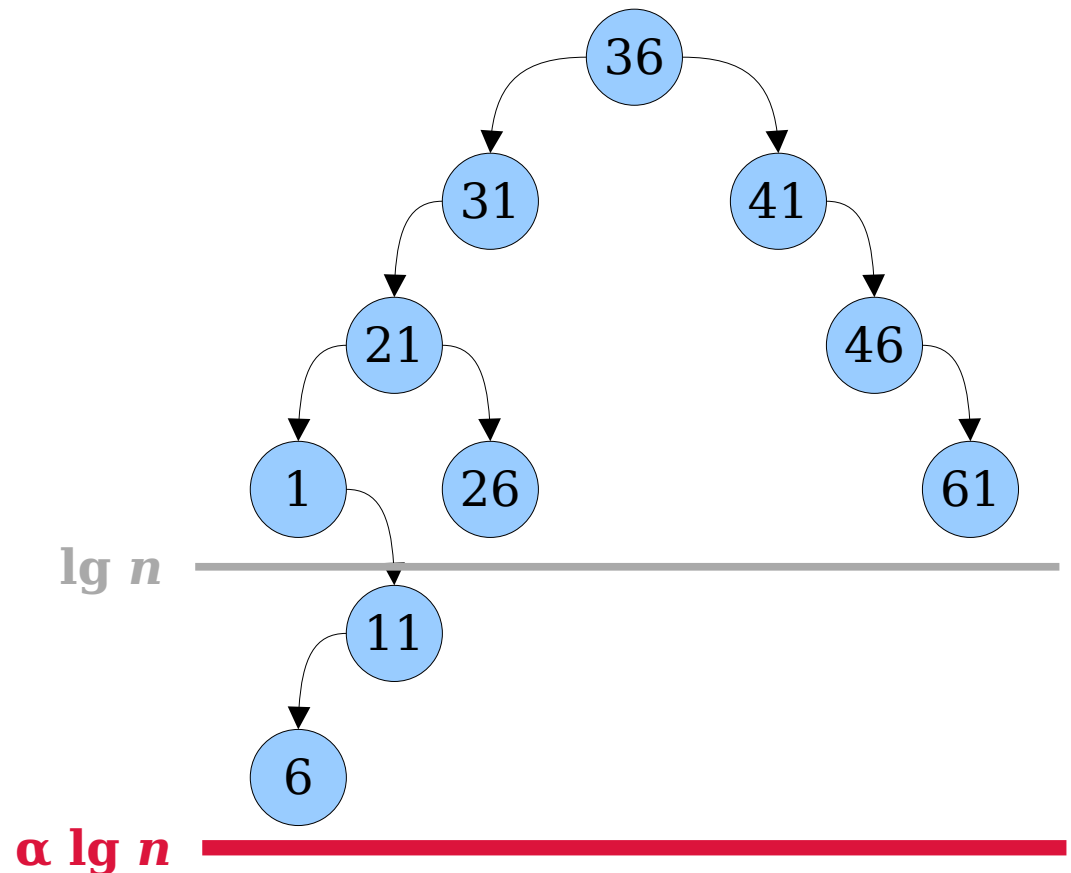
Adding Slack Space

- Pick a fixed constant $\alpha > 1$.
- Set the maximum height on our tree to $\alpha \lg n$.
- As long as we don't exceed this maximum height, all operations on our BST will run in time $O(\log n)$, and we don't really care about the shape of the tree.



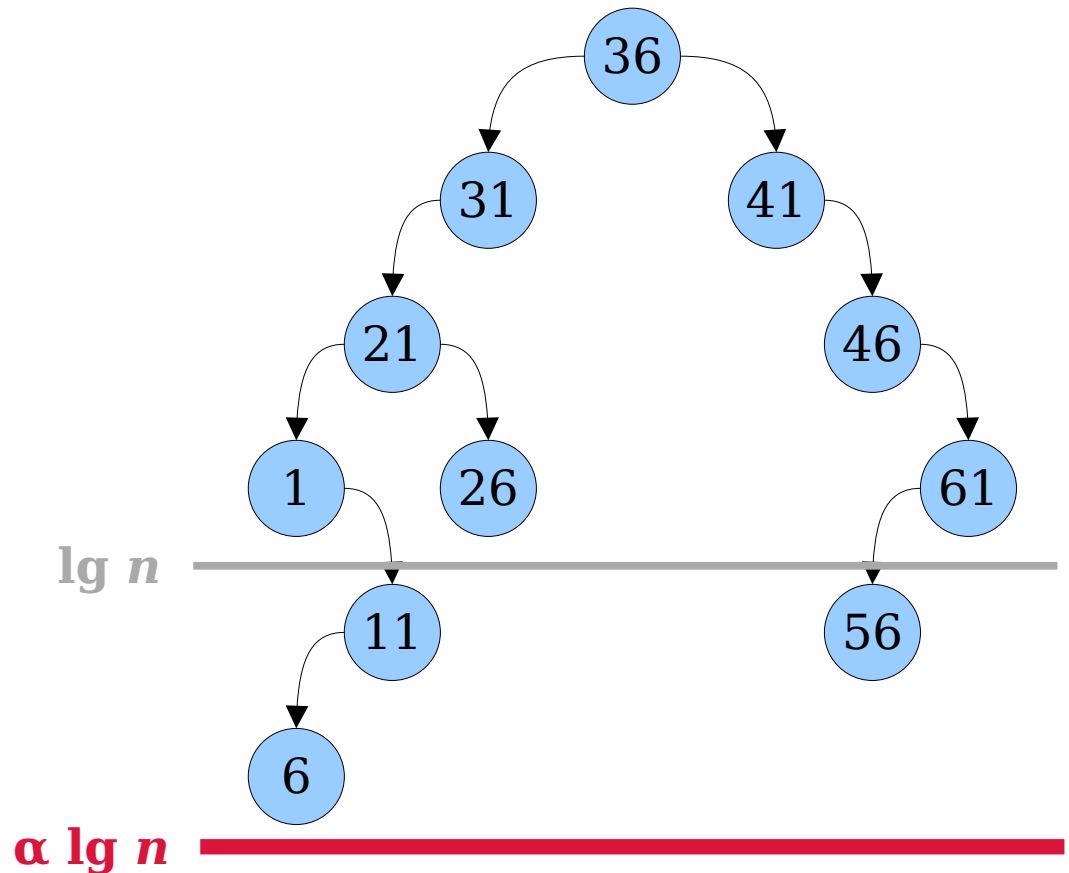
Adding Slack Space

- Pick a fixed constant $\alpha > 1$.
- Set the maximum height on our tree to $\alpha \lg n$.
- As long as we don't exceed this maximum height, all operations on our BST will run in time $O(\log n)$, and we don't really care about the shape of the tree.



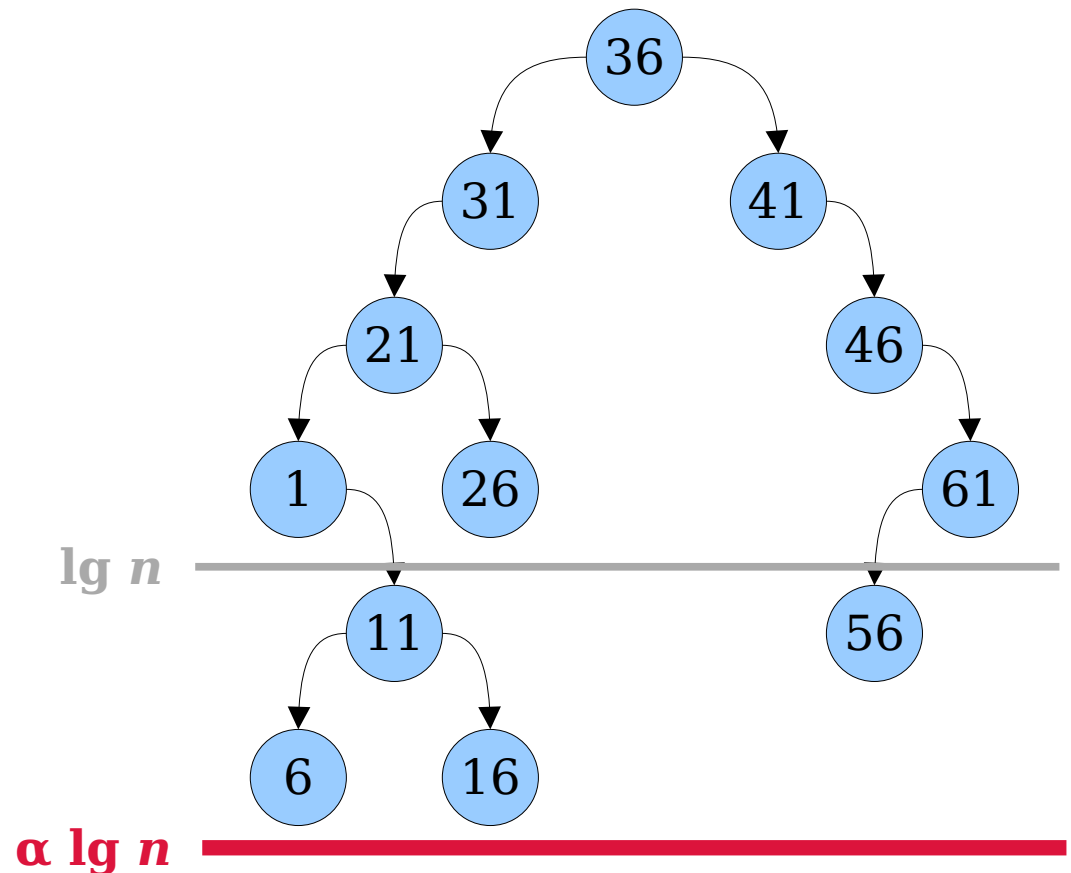
Adding Slack Space

- Pick a fixed constant $\alpha > 1$.
- Set the maximum height on our tree to $\alpha \lg n$.
- As long as we don't exceed this maximum height, all operations on our BST will run in time $O(\log n)$, and we don't really care about the shape of the tree.



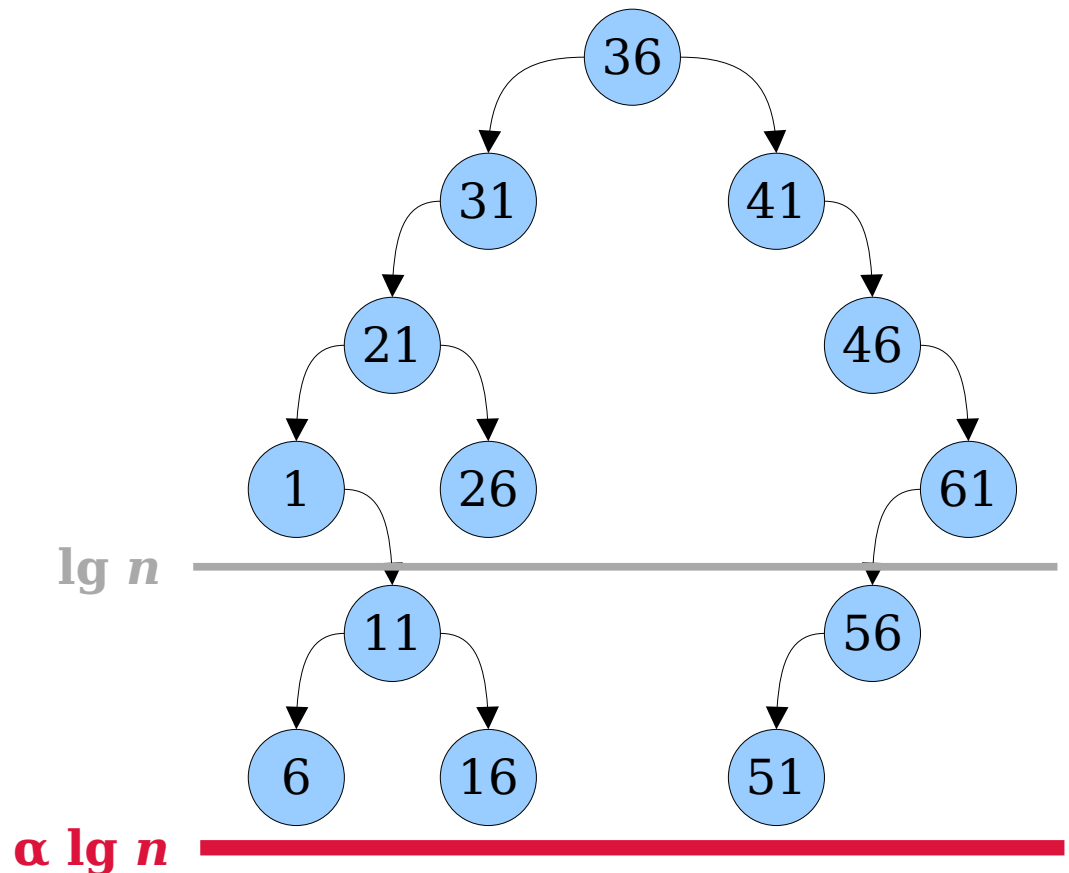
Adding Slack Space

- Pick a fixed constant $\alpha > 1$.
- Set the maximum height on our tree to $\alpha \lg n$.
- As long as we don't exceed this maximum height, all operations on our BST will run in time $O(\log n)$, and we don't really care about the shape of the tree.



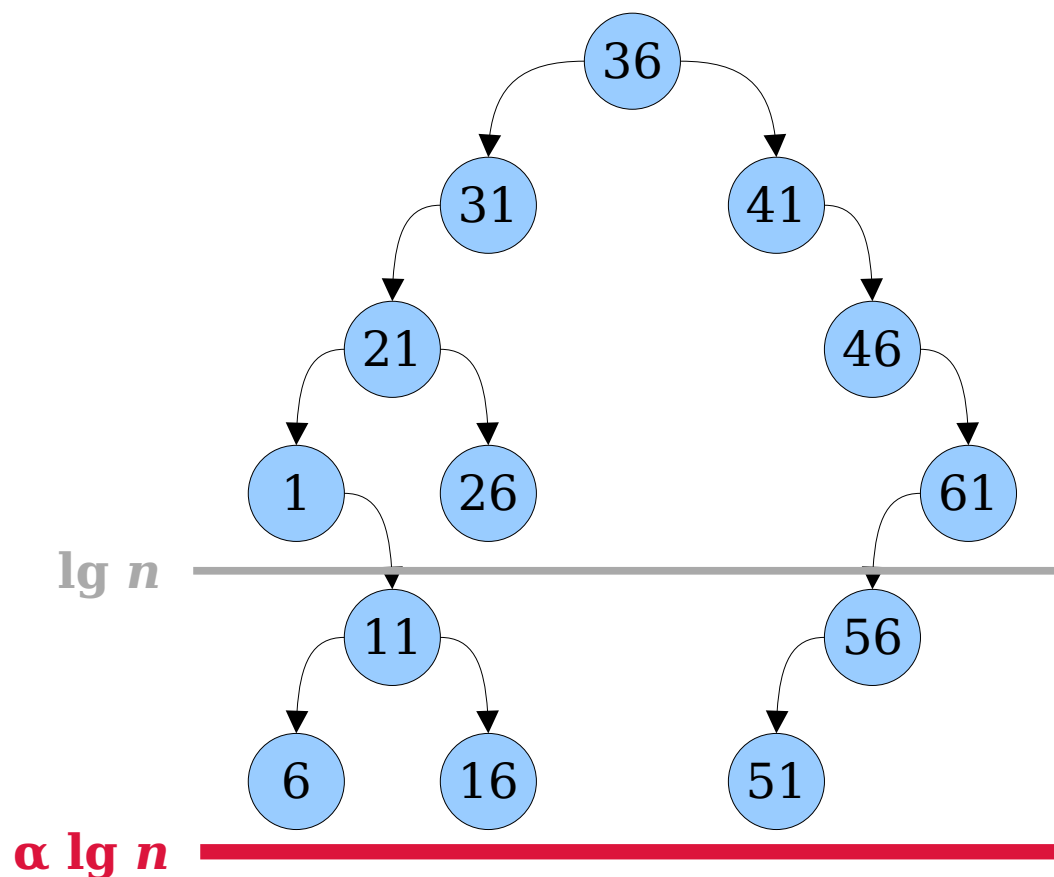
Adding Slack Space

- Pick a fixed constant $\alpha > 1$.
- Set the maximum height on our tree to $\alpha \lg n$.
- As long as we don't exceed this maximum height, all operations on our BST will run in time $O(\log n)$, and we don't really care about the shape of the tree.



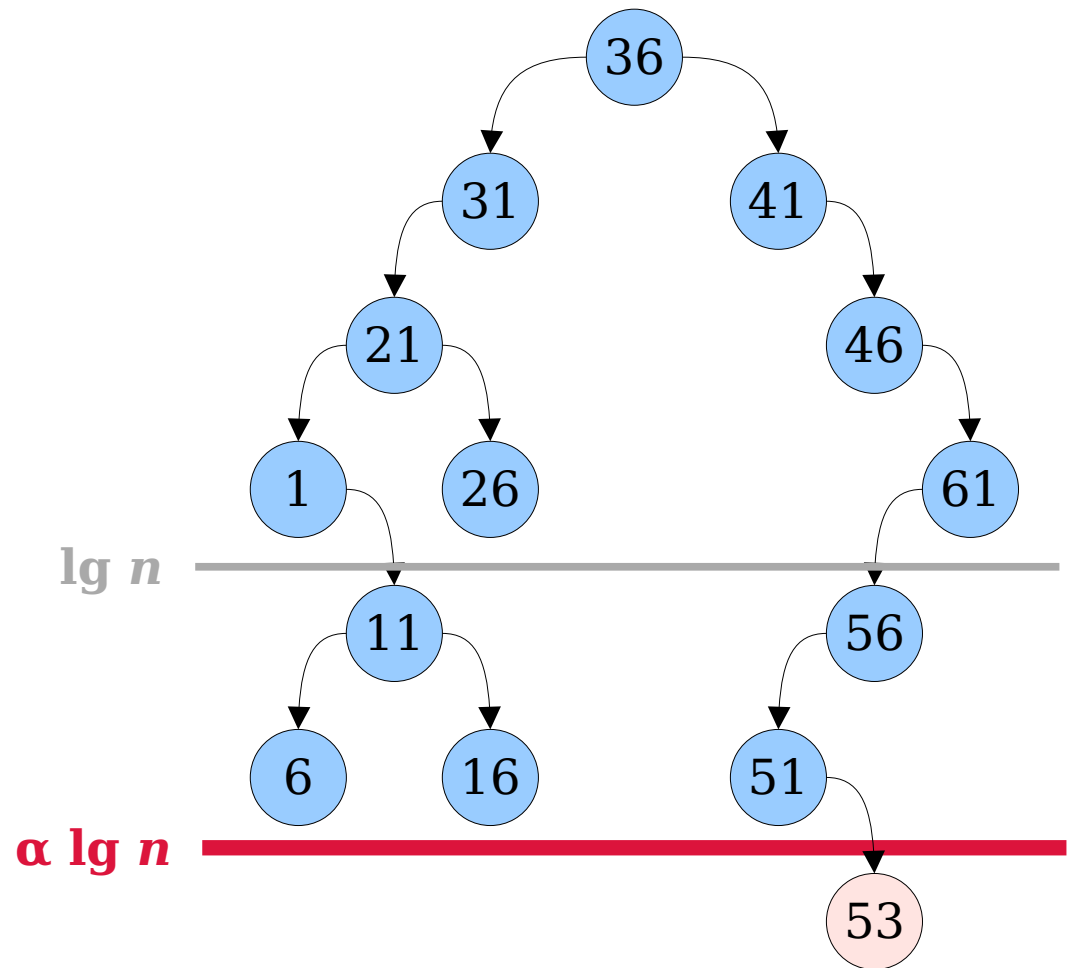
Adding Slack Space

- For each node v in our BST, let **size(v)** denote the number of nodes in the subtree rooted at v and **height(v)** denote the height of the subtree rooted at v .
- We'll say that a node v is **α -balanced** if $\text{height}(v) \leq \alpha \lg \text{size}(v)$.
- Intuitively, a α -balanced node is the root of a subtree whose height is within a factor of α of optimal.



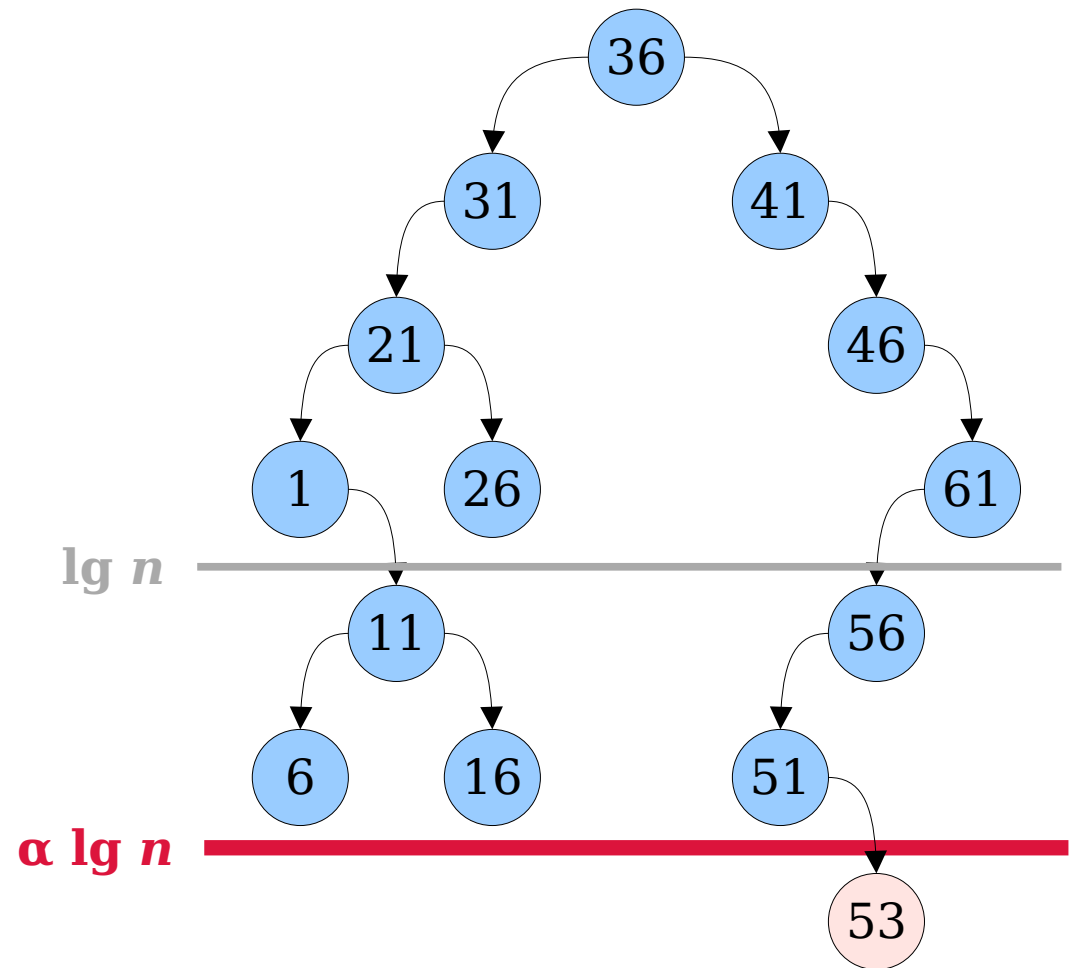
Adding Slack Space

- Suppose, however, that after doing an insertion, our tree exceeds $\alpha \lg n$.
- At this point, we need to do some sort of “cleanup” on the tree to pull it back to a reasonable height.
- Ideally, we’ll want to minimize the amount of cleanup we need to do so that this step will run quickly.



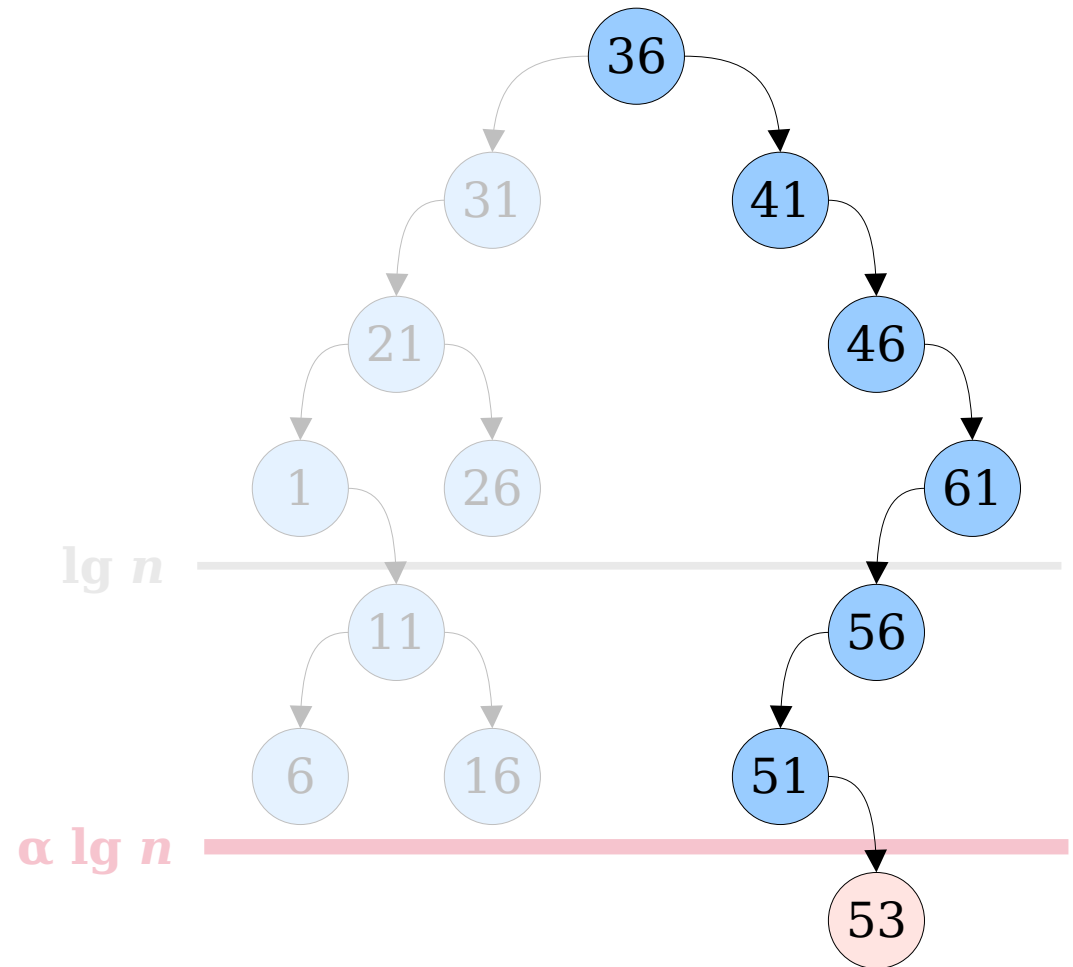
Scapegoat Nodes

- Look at the access path from the root node to the newly-inserted node.



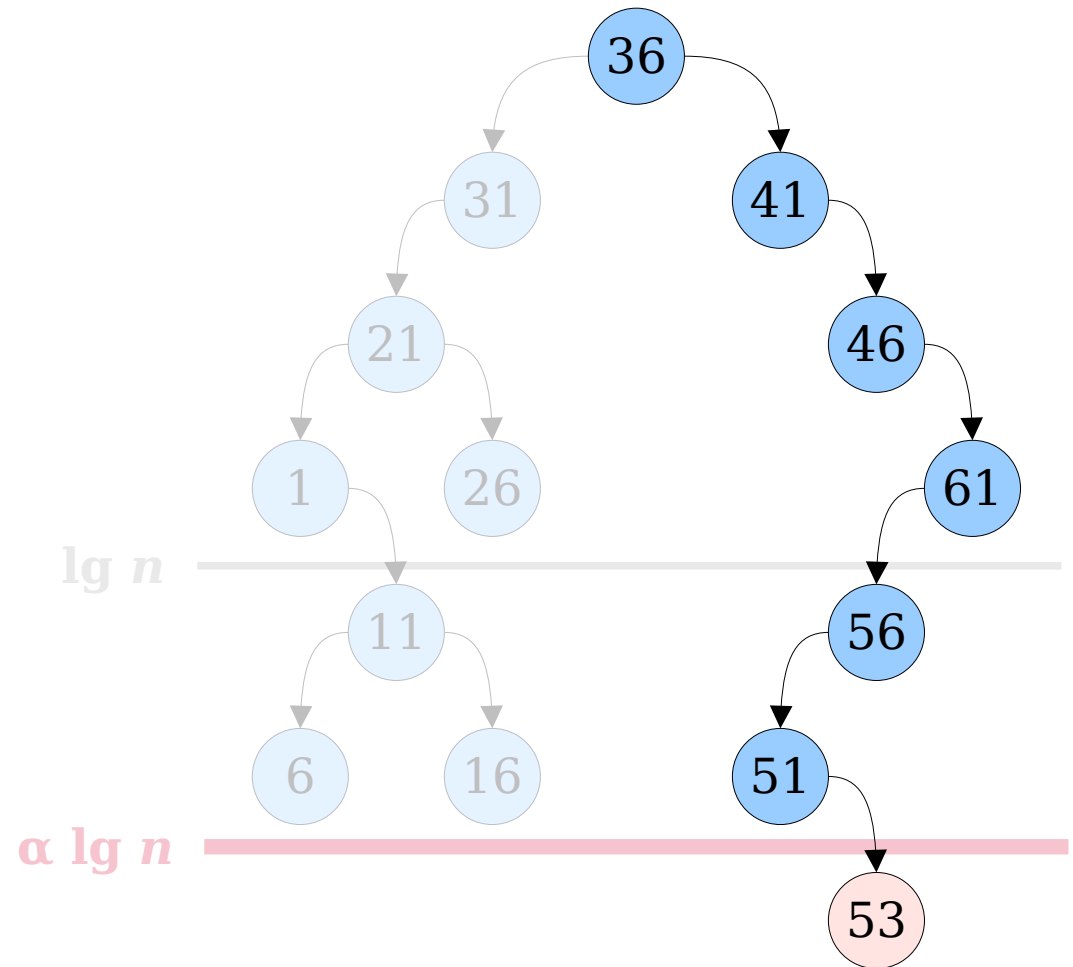
Scapegoat Nodes

- Look at the access path from the root node to the newly-inserted node.



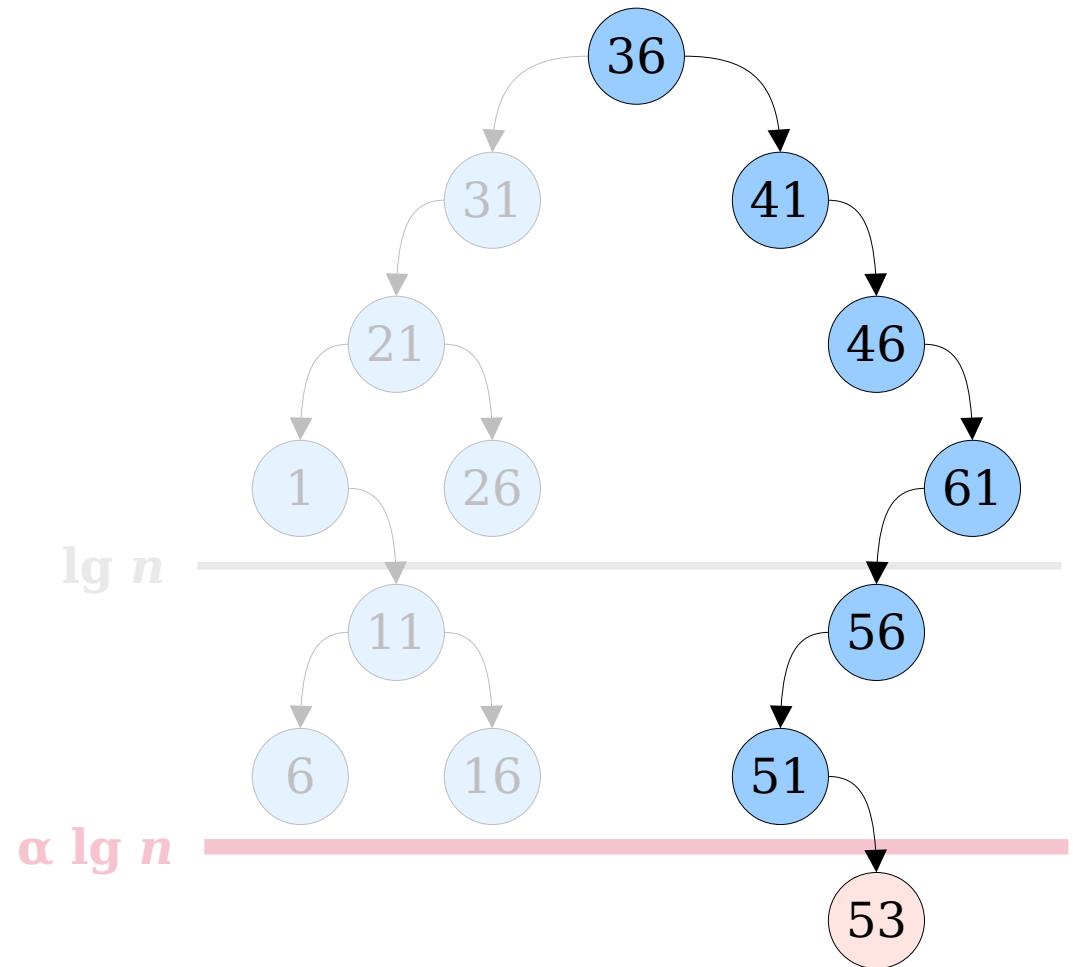
Scapegoat Nodes

- Look at the access path from the root node to the newly-inserted node.
- We know the root node is not α -balanced, since the overall tree is too tall.



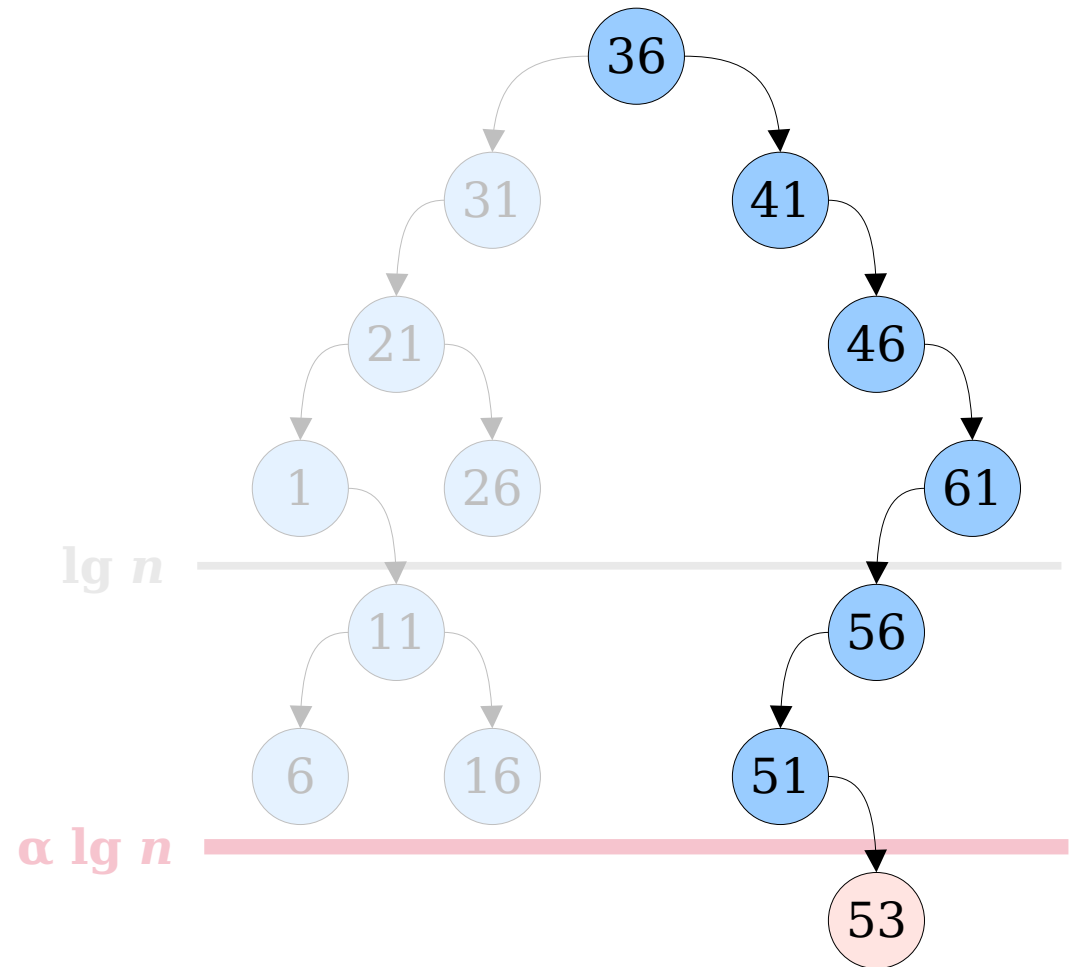
Scapegoat Nodes

- Look at the access path from the root node to the newly-inserted node.
- We know the root node is not α -balanced, since the overall tree is too tall.
- We also know that the newly-inserted node is α -balanced, since it has no children.



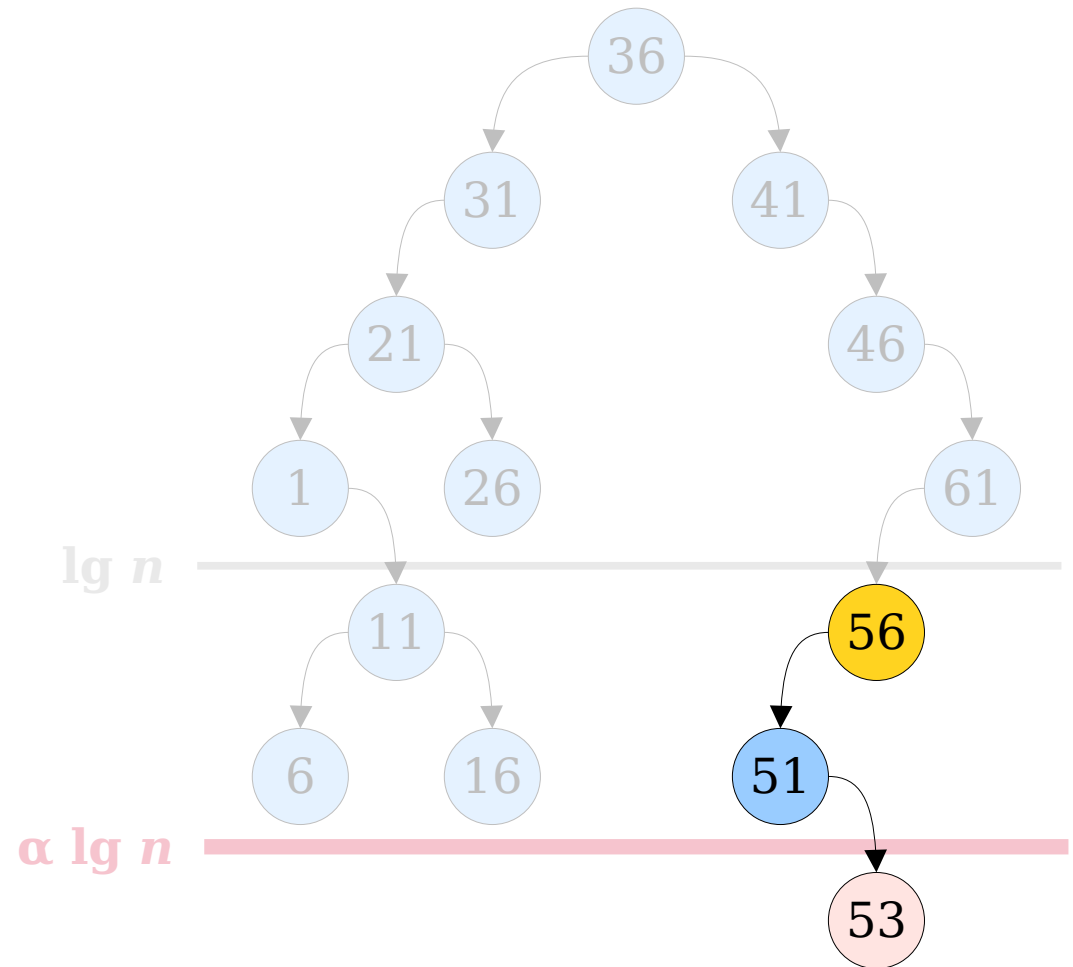
Scapegoat Nodes

- Look at the access path from the root node to the newly-inserted node.
- We know the root node is not α -balanced, since the overall tree is too tall.
- We also know that the newly-inserted node is α -balanced, since it has no children.
- Therefore, there has to be some deepest node on the access path that isn't α -balanced.



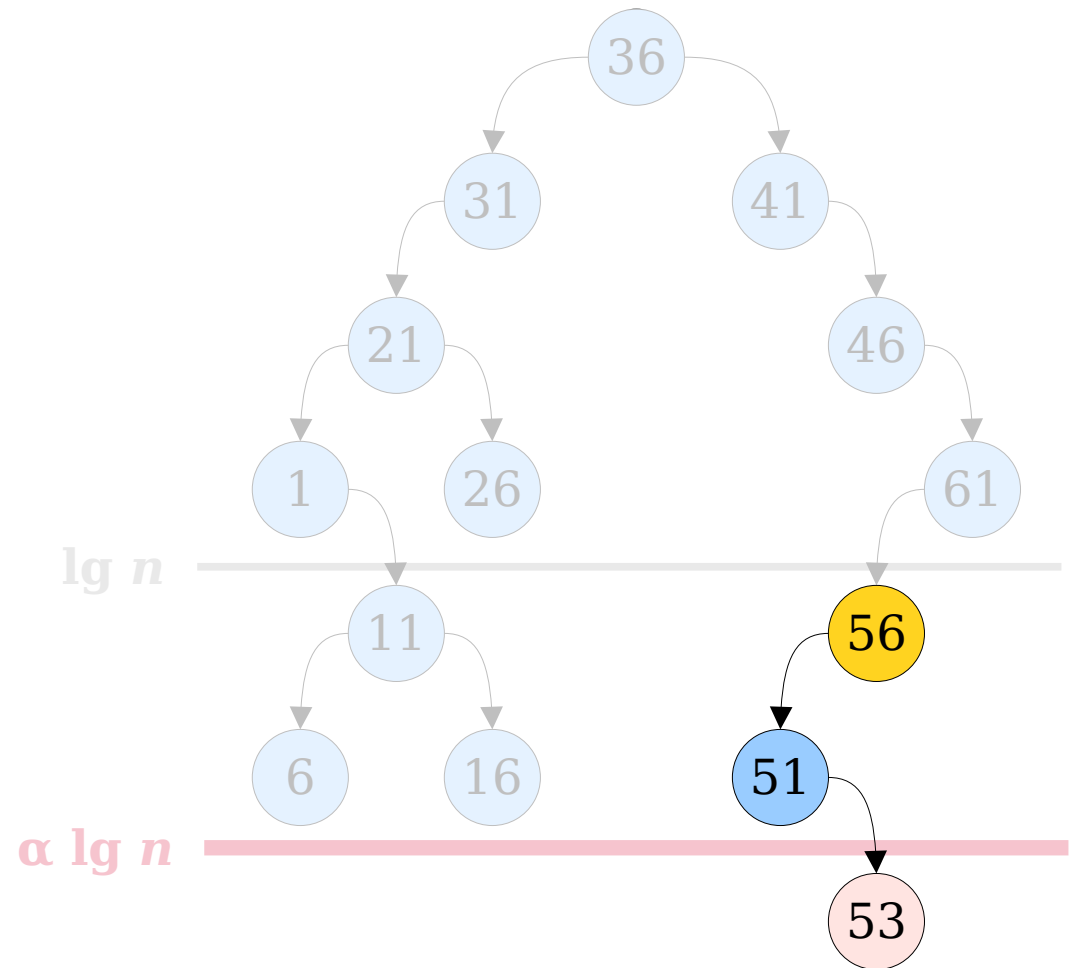
Scapegoat Nodes

- Look at the access path from the root node to the newly-inserted node.
- We know the root node is not α -balanced, since the overall tree is too tall.
- We also know that the newly-inserted node is α -balanced, since it has no children.
- Therefore, there has to be some deepest node on the access path that isn't α -balanced.



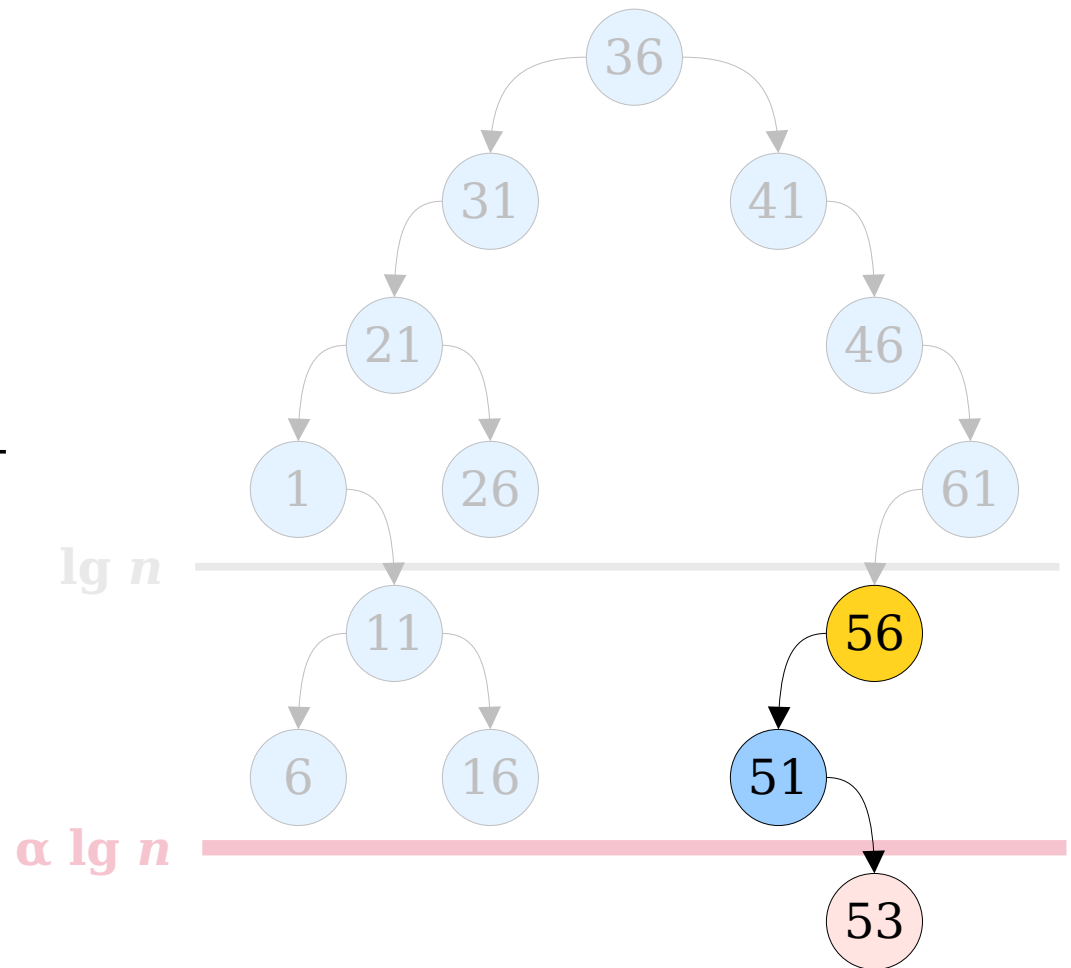
Scapegoat Nodes

- Look at the access path from the root node to the newly-inserted node.
- We know the root node is not α -balanced, since the overall tree is too tall.
- We also know that the newly-inserted node is α -balanced, since it has no children.
- Therefore, there has to be some deepest node on the access path that isn't α -balanced.
- We can “blame” the imbalance in the overall tree on this subtree. The node chosen this way is called the **scapegoat**.



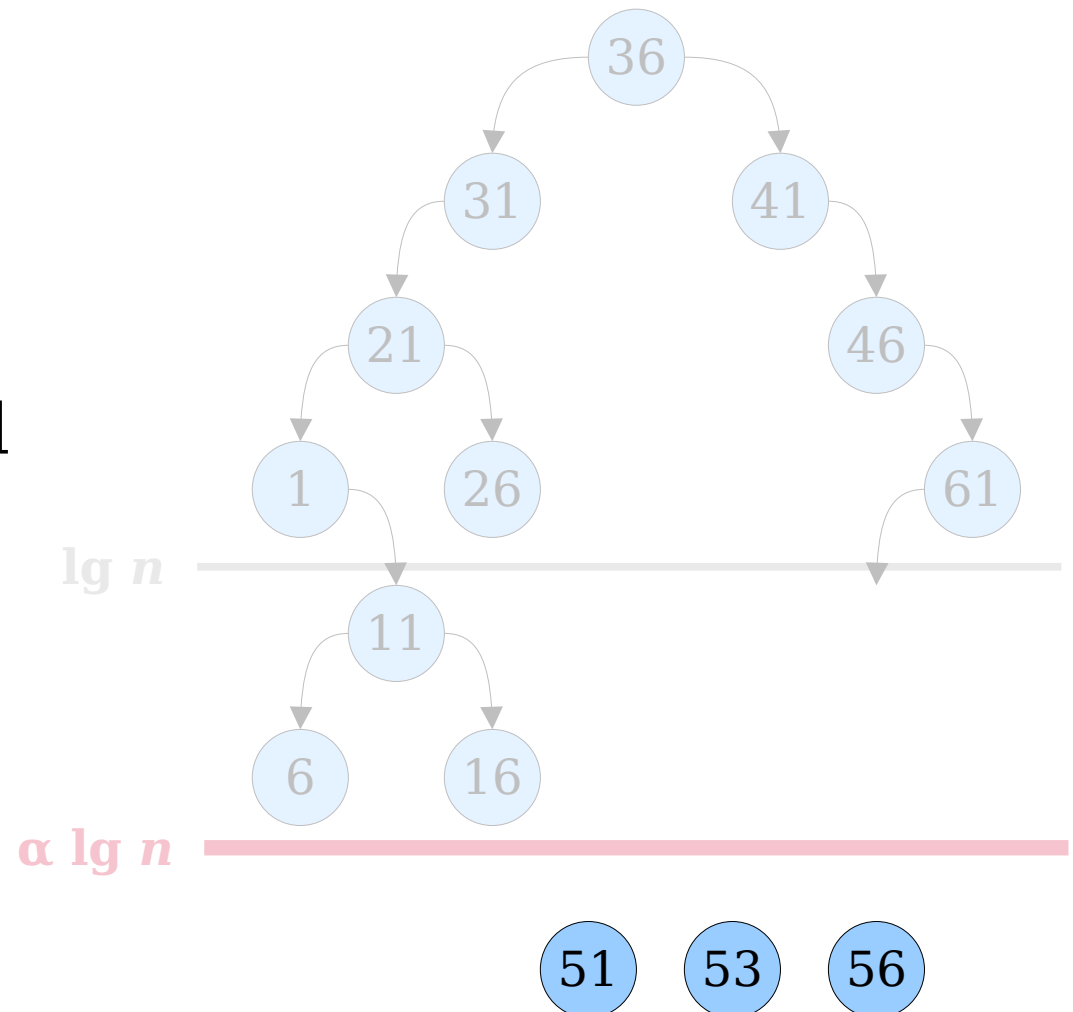
Scapegoat Nodes

- We know that the subtree rooted at the scapegoat isn't α -balanced.
- **Idea:** Rebuild this tree as a perfectly-balanced BST.
- This will reduce the height of the subtree, which in turn restores the requirement that the height be at most $\alpha \lg n$.



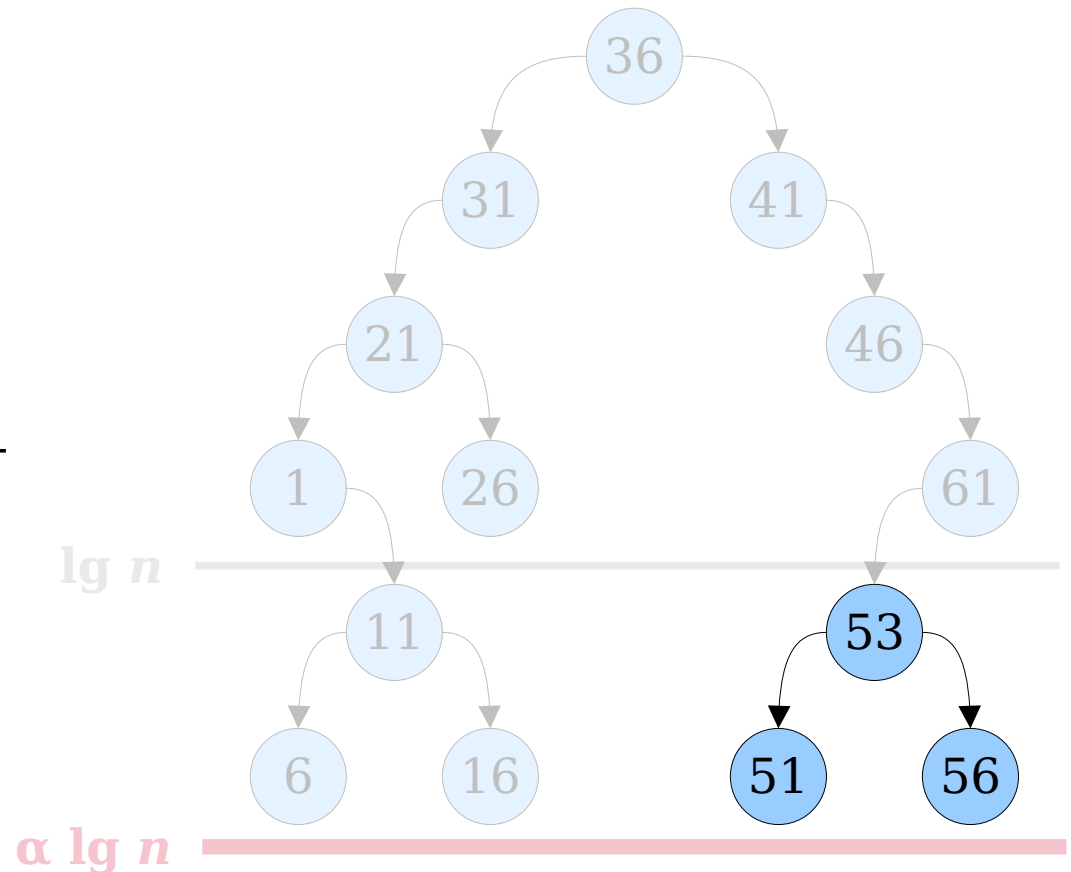
Scapegoat Nodes

- We know that the subtree rooted at the scapegoat isn't α -balanced.
- **Idea:** Rebuild this tree as a perfectly-balanced BST.
- This will reduce the height of the subtree, which in turn restores the requirement that the height be at most $\alpha \lg n$.



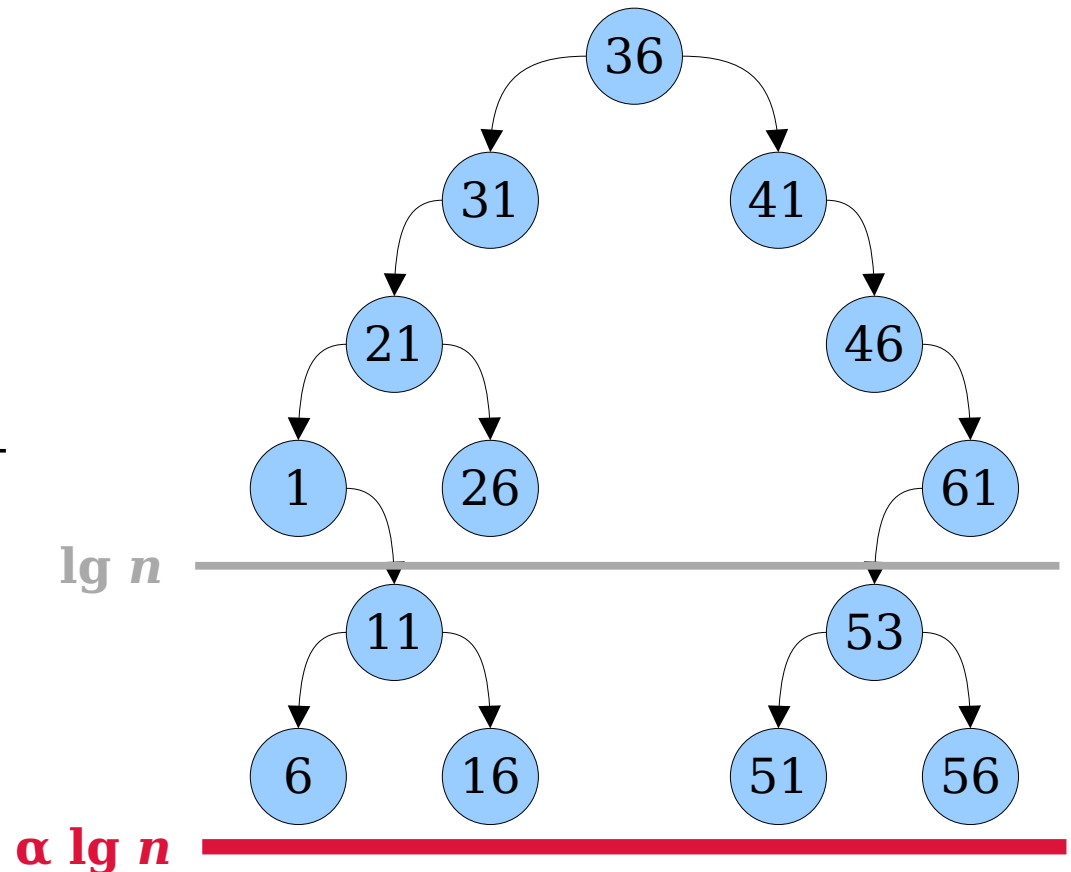
Scapegoat Nodes

- We know that the subtree rooted at the scapegoat isn't α -balanced.
- **Idea:** Rebuild this tree as a perfectly-balanced BST.
- This will reduce the height of the subtree, which in turn restores the requirement that the height be at most $\alpha \lg n$.



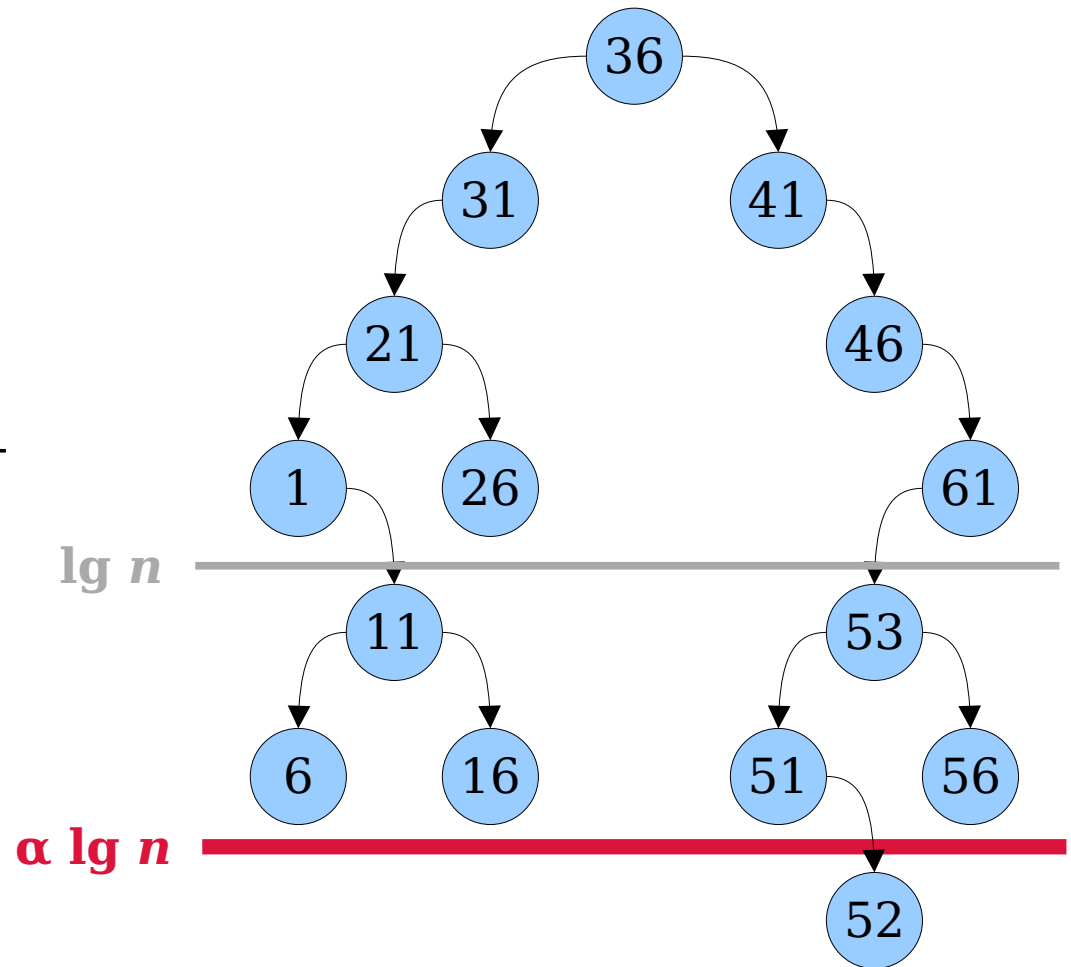
Scapegoat Nodes

- We know that the subtree rooted at the scapegoat isn't α -balanced.
- **Idea:** Rebuild this tree as a perfectly-balanced BST.
- This will reduce the height of the subtree, which in turn restores the requirement that the height be at most $\alpha \lg n$.



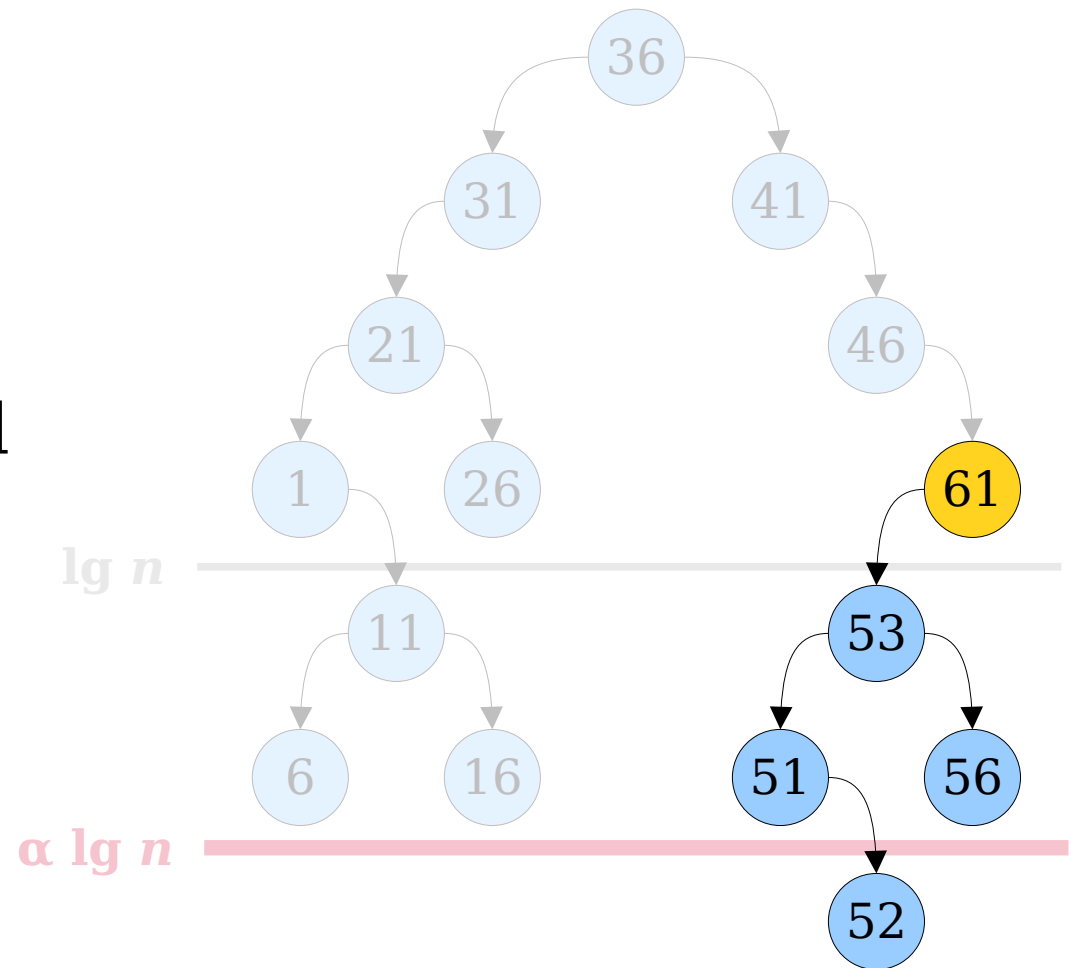
Scapegoat Nodes

- We know that the subtree rooted at the scapegoat isn't α -balanced.
- **Idea:** Rebuild this tree as a perfectly-balanced BST.
- This will reduce the height of the subtree, which in turn restores the requirement that the height be at most $\alpha \lg n$.



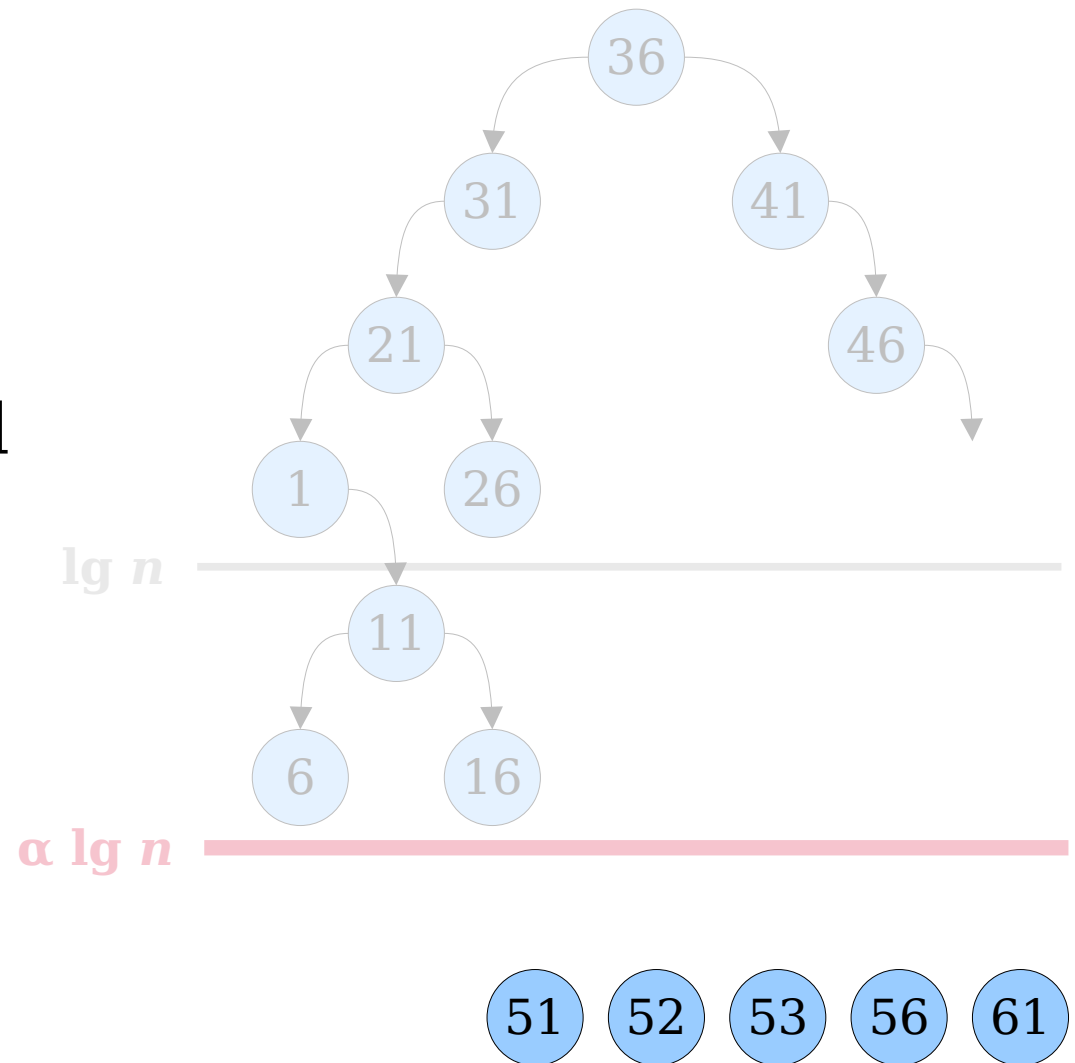
Scapegoat Nodes

- We know that the subtree rooted at the scapegoat isn't α -balanced.
- **Idea:** Rebuild this tree as a perfectly-balanced BST.
- This will reduce the height of the subtree, which in turn restores the requirement that the height be at most $\alpha \lg n$.



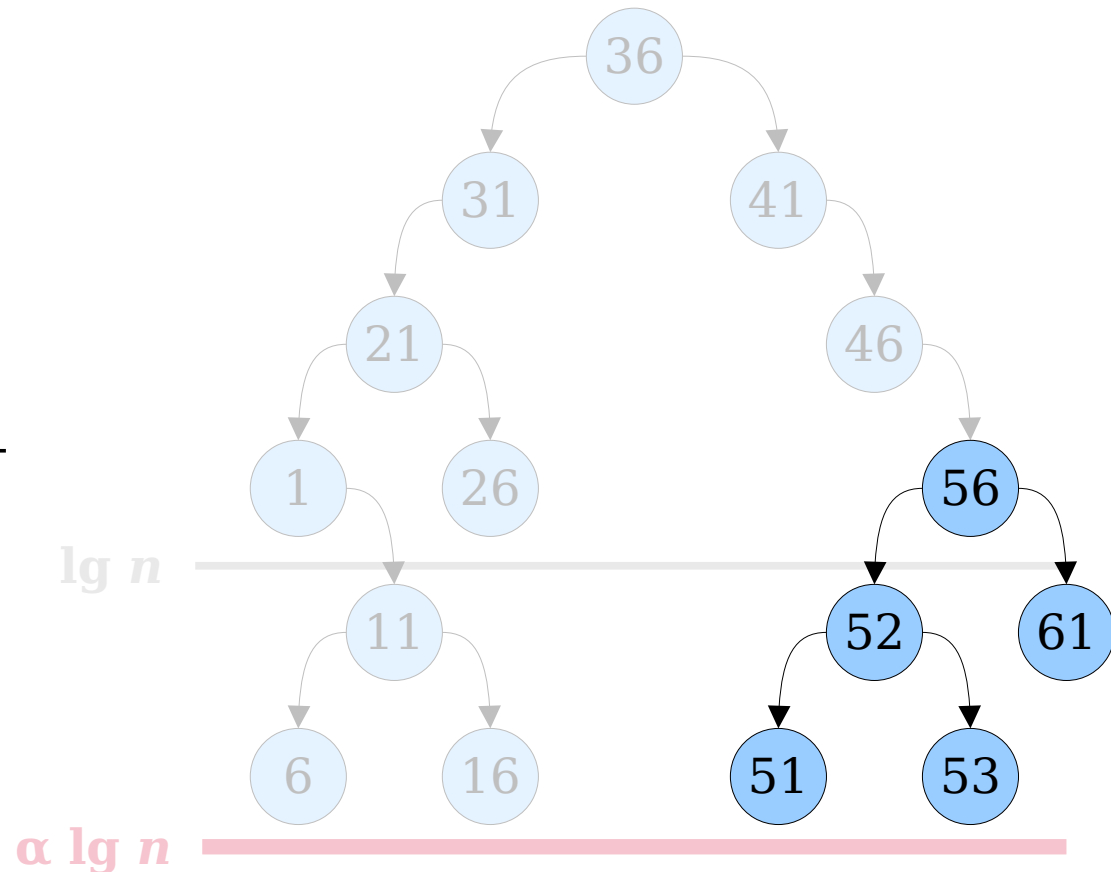
Scapegoat Nodes

- We know that the subtree rooted at the scapegoat isn't α -balanced.
- **Idea:** Rebuild this tree as a perfectly-balanced BST.
- This will reduce the height of the subtree, which in turn restores the requirement that the height be at most $\alpha \lg n$.



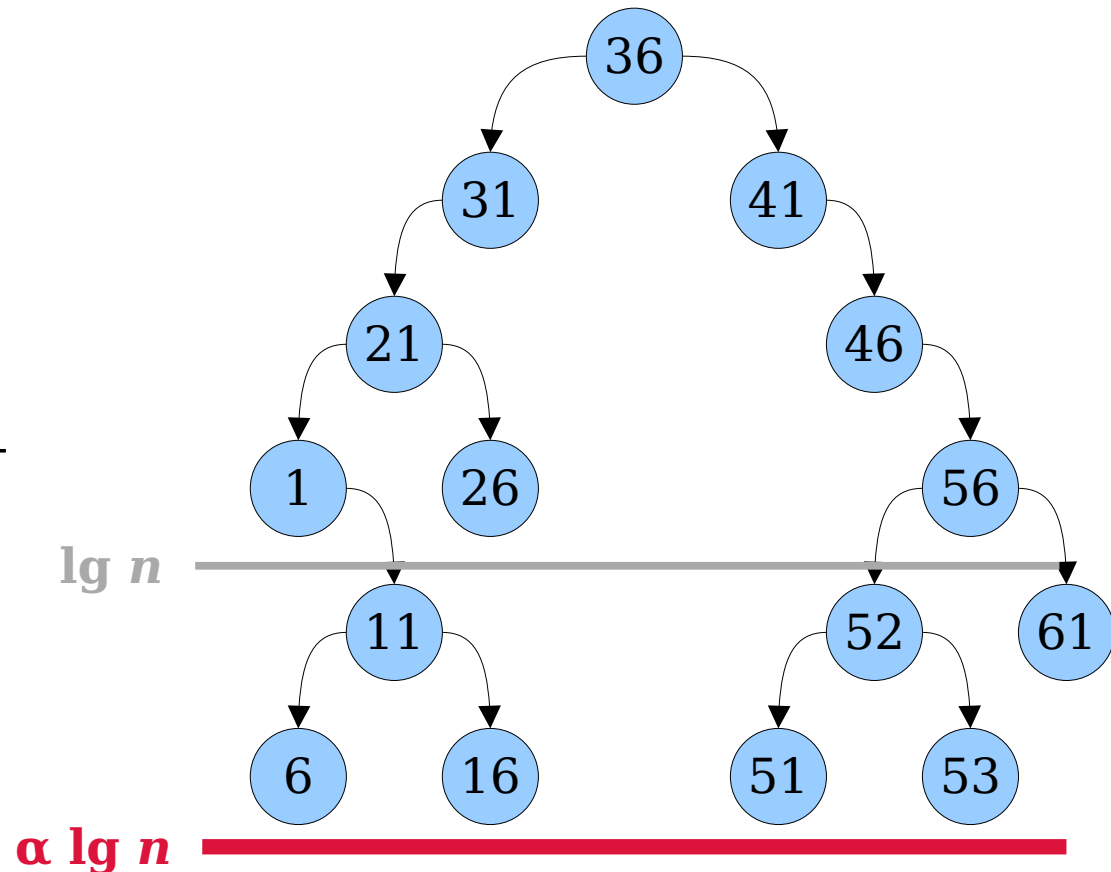
Scapegoat Nodes

- We know that the subtree rooted at the scapegoat isn't α -balanced.
- **Idea:** Rebuild this tree as a perfectly-balanced BST.
- This will reduce the height of the subtree, which in turn restores the requirement that the height be at most $\alpha \lg n$.



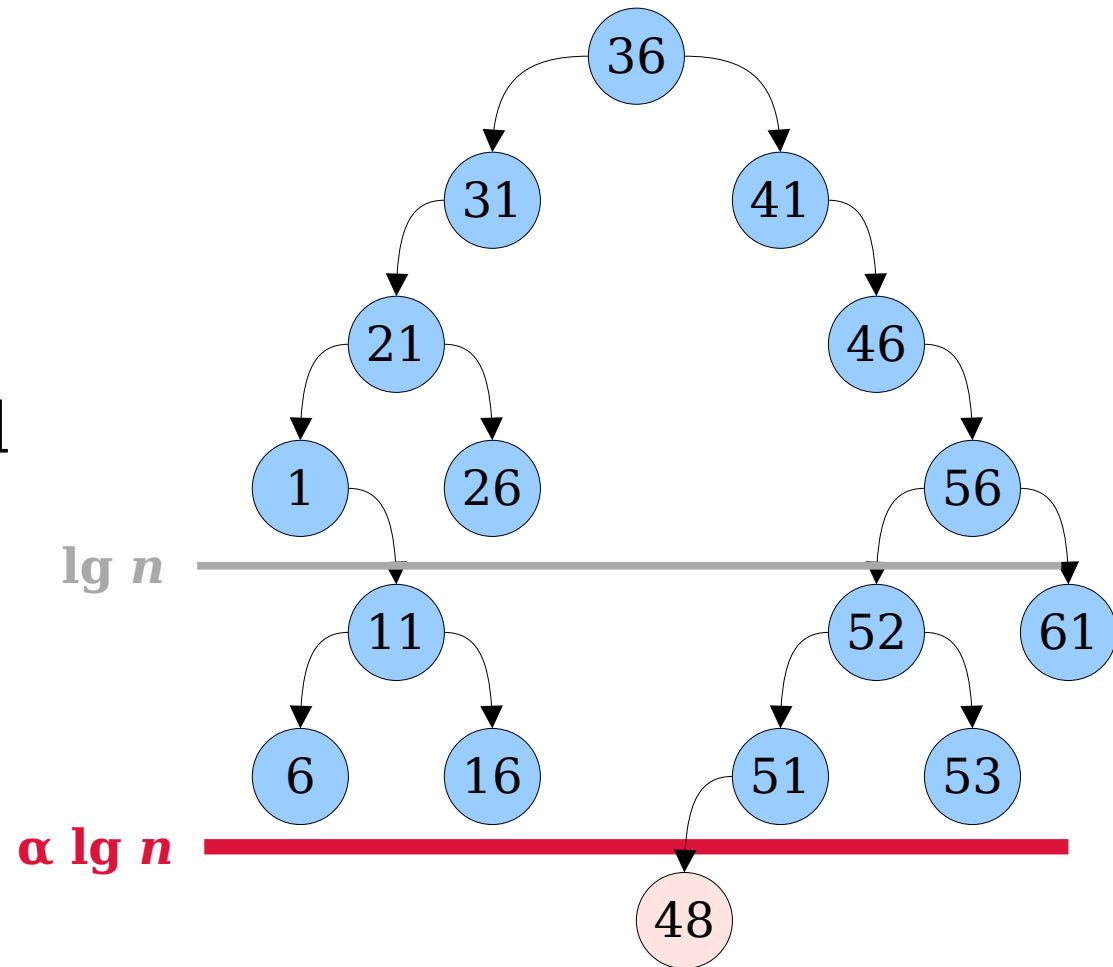
Scapegoat Nodes

- We know that the subtree rooted at the scapegoat isn't α -balanced.
- **Idea:** Rebuild this tree as a perfectly-balanced BST.
- This will reduce the height of the subtree, which in turn restores the requirement that the height be at most $\alpha \lg n$.



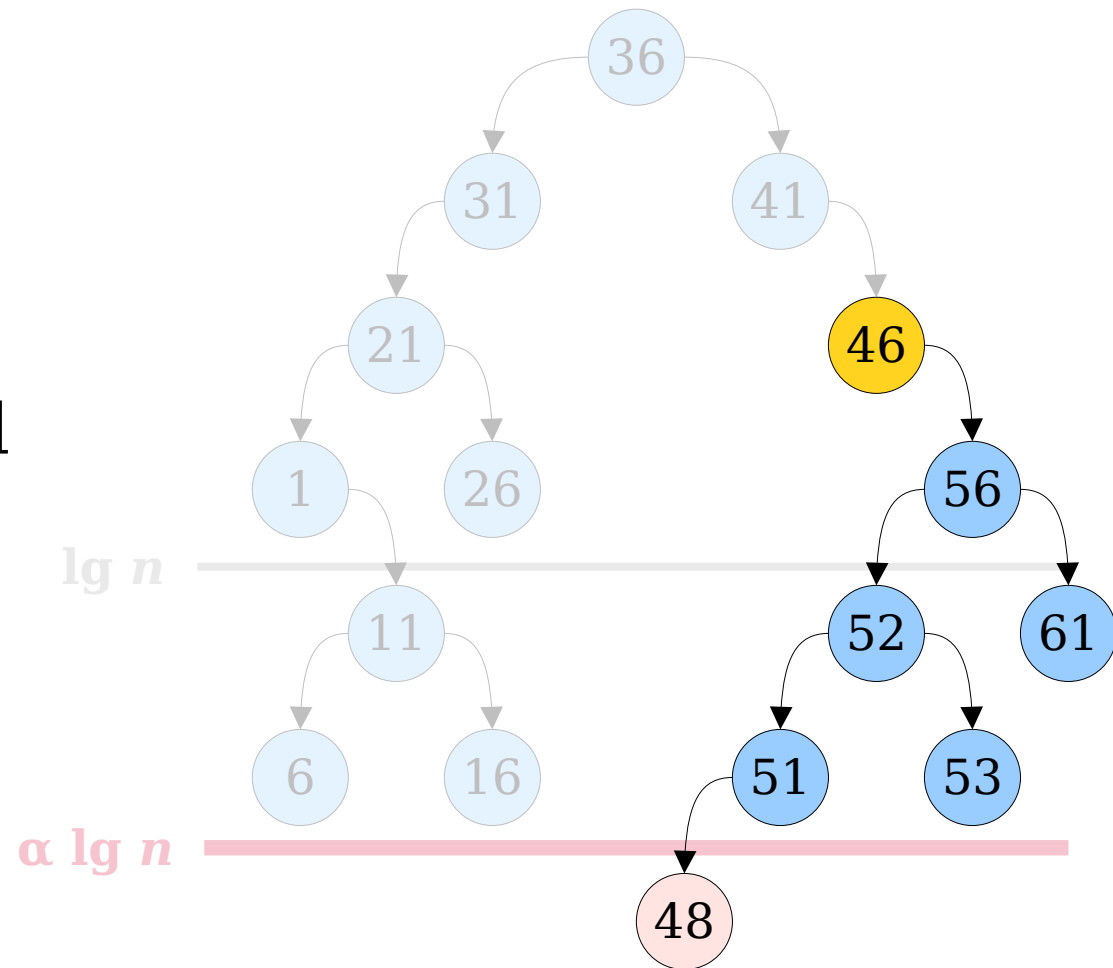
Scapegoat Nodes

- We know that the subtree rooted at the scapegoat isn't α -balanced.
- **Idea:** Rebuild this tree as a perfectly-balanced BST.
- This will reduce the height of the subtree, which in turn restores the requirement that the height be at most $\alpha \lg n$.



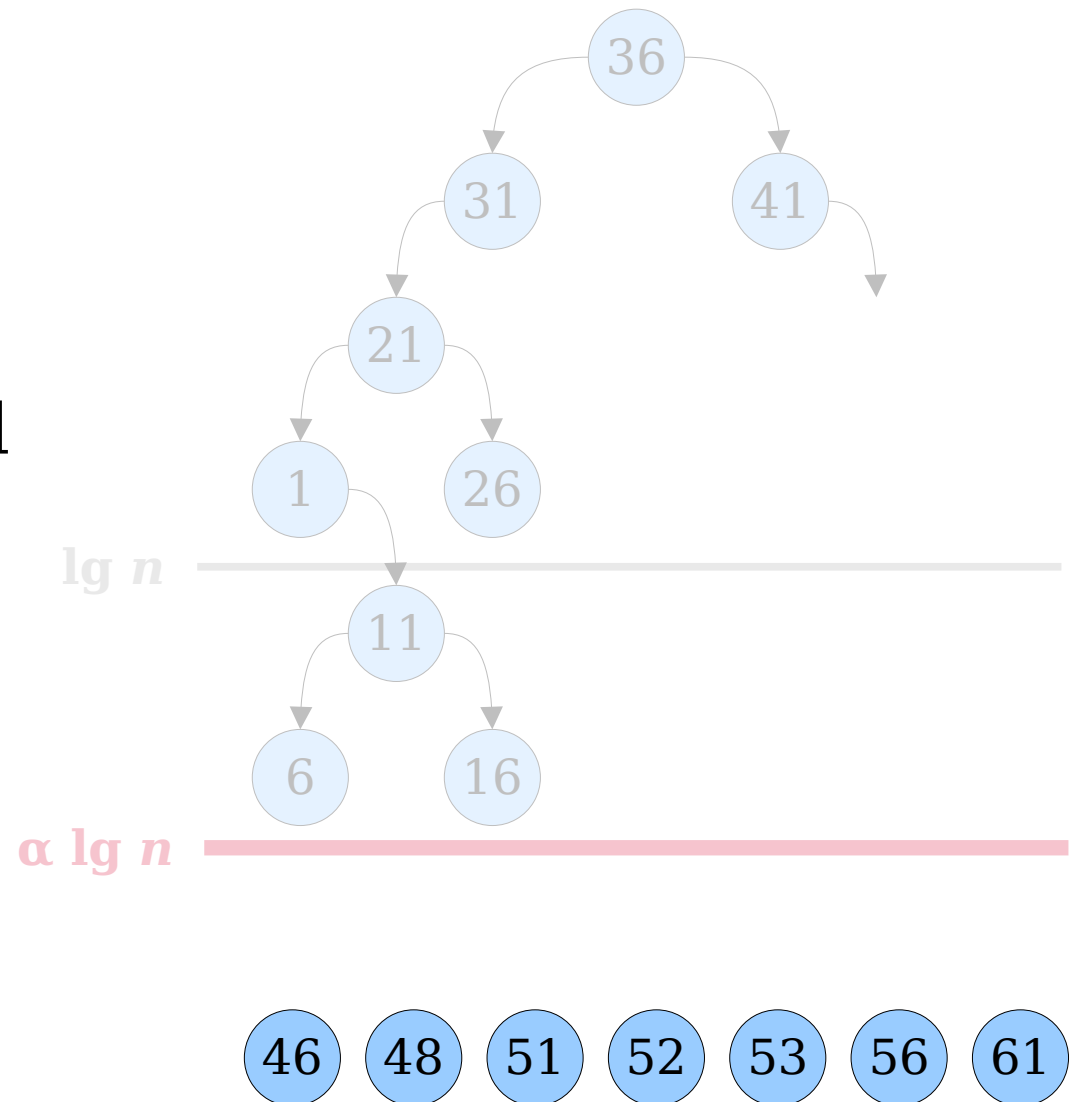
Scapegoat Nodes

- We know that the subtree rooted at the scapegoat isn't α -balanced.
- **Idea:** Rebuild this tree as a perfectly-balanced BST.
- This will reduce the height of the subtree, which in turn restores the requirement that the height be at most $\alpha \lg n$.



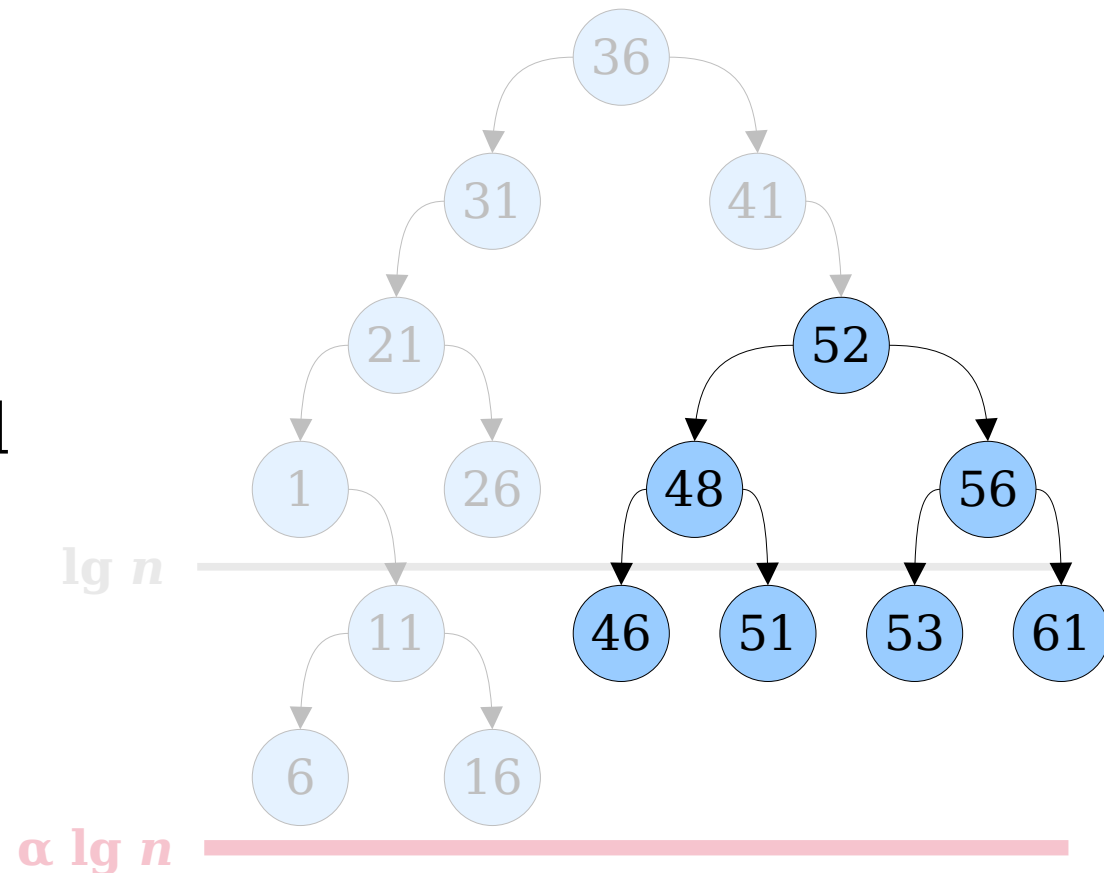
Scapegoat Nodes

- We know that the subtree rooted at the scapegoat isn't α -balanced.
- **Idea:** Rebuild this tree as a perfectly-balanced BST.
- This will reduce the height of the subtree, which in turn restores the requirement that the height be at most $\alpha \lg n$.



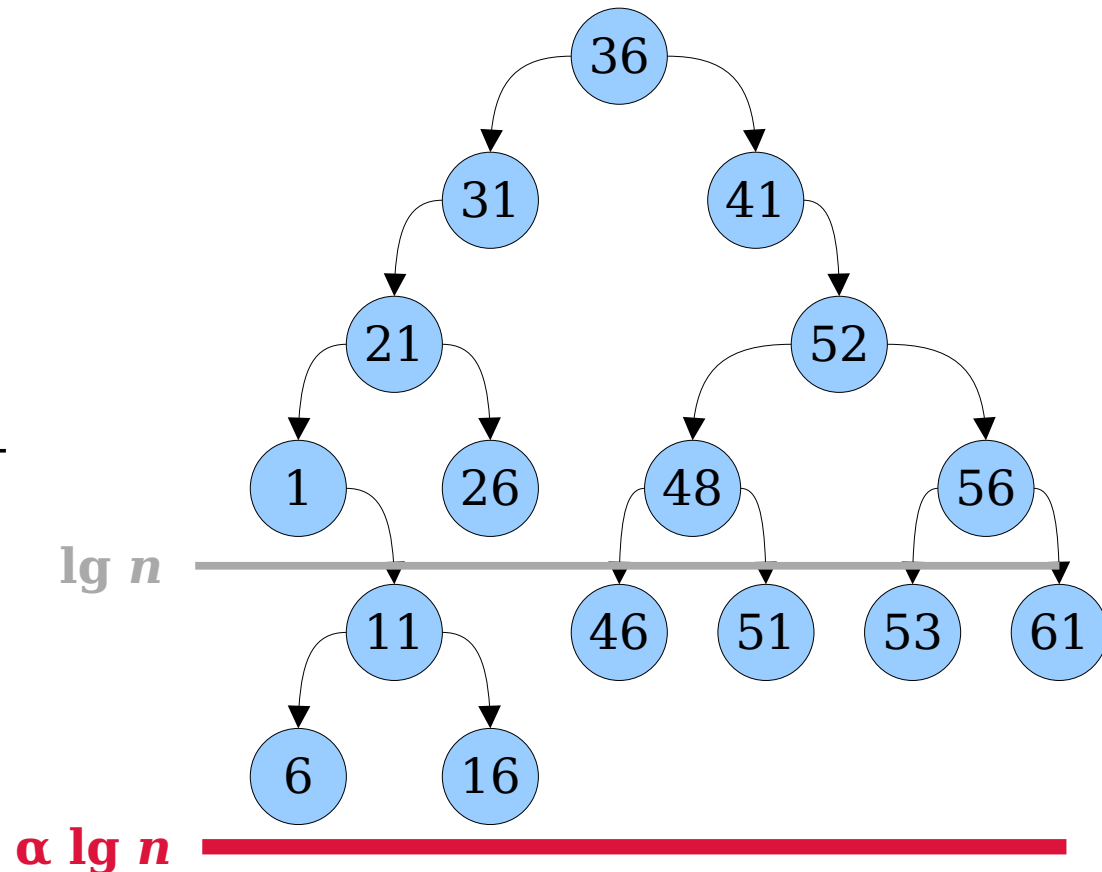
Scapegoat Nodes

- We know that the subtree rooted at the scapegoat isn't α -balanced.
- **Idea:** Rebuild this tree as a perfectly-balanced BST.
- This will reduce the height of the subtree, which in turn restores the requirement that the height be at most $\alpha \lg n$.



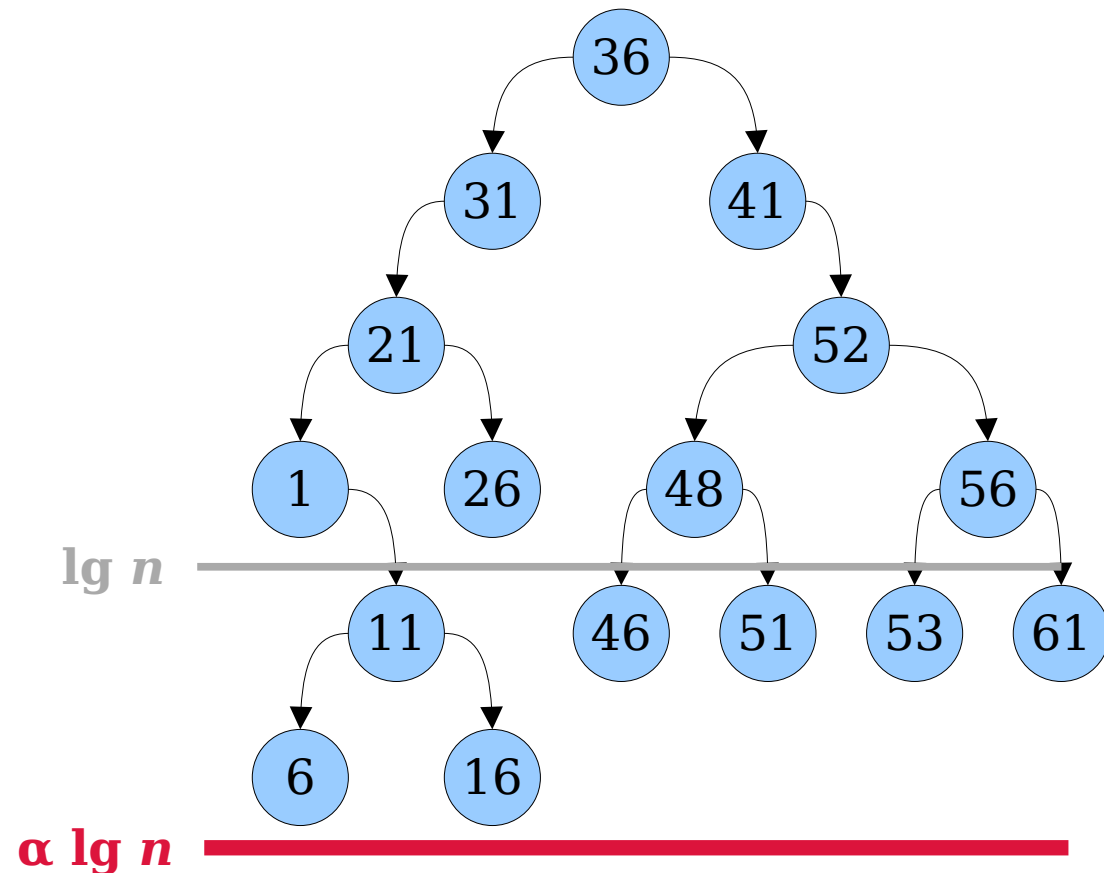
Scapegoat Nodes

- We know that the subtree rooted at the scapegoat isn't α -balanced.
- **Idea:** Rebuild this tree as a perfectly-balanced BST.
- This will reduce the height of the subtree, which in turn restores the requirement that the height be at most $\alpha \lg n$.



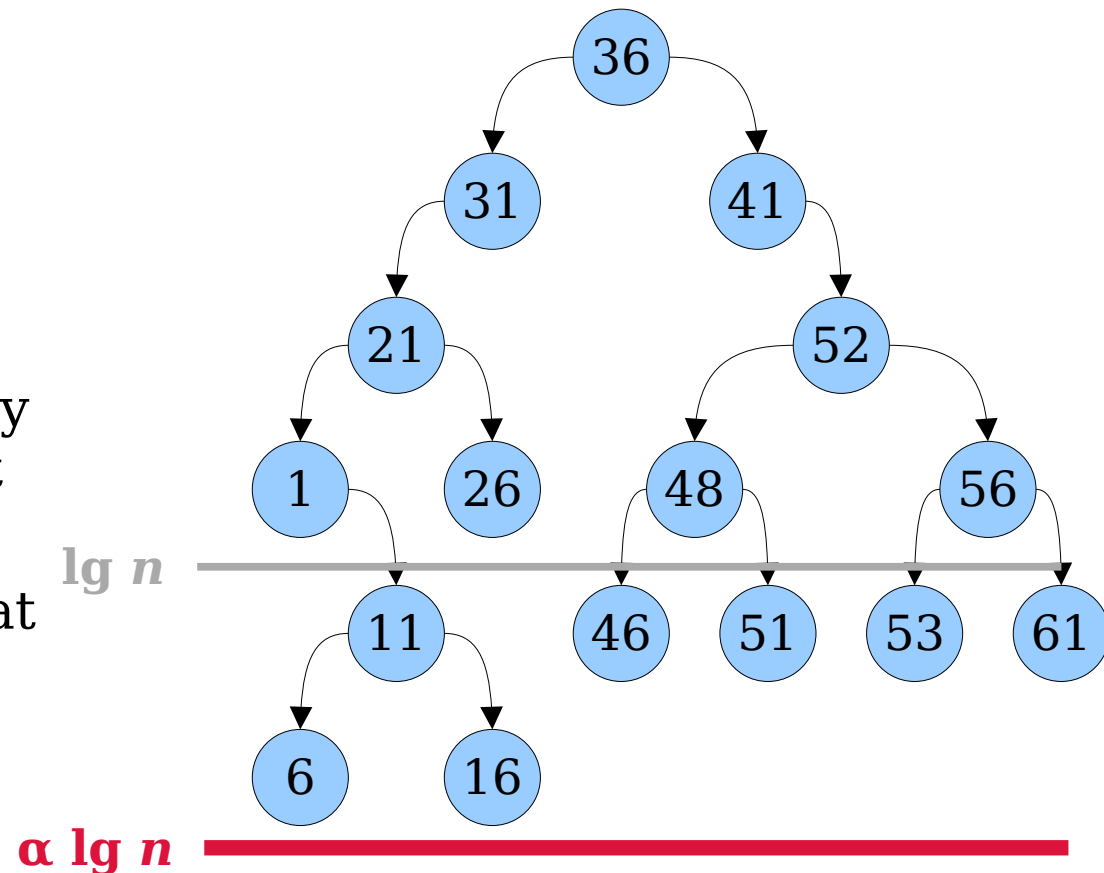
Scapegoat Trees

- A **scapegoat tree** is a balanced binary search tree that works as follows:
 - Pick some constant $\alpha > 1$.
 - As long as the tree height is below $\alpha \lg n$, don't do any rebalancing after insertions.
 - Once the tree exceeds that height, find the scapegoat (the deepest α -imbalanced node on the insertion path).
 - Then, optimally rebuild the subtree rooted at that node.
- All that's left now is to work through the details.



Scapegoat Trees

- Questions we need to address:
 - How do we know that optimally rebuilding the scapegoat's subtree will fix the tree height?
 - How quickly can we optimally rebuild the subtree rooted at the scapegoat node?
 - How do we find the scapegoat node?
 - In an amortized sense, how fast is this strategy?
- Let's address each of these in turn.



The Impact of Rebuilding

Scapegoat Rebuilding

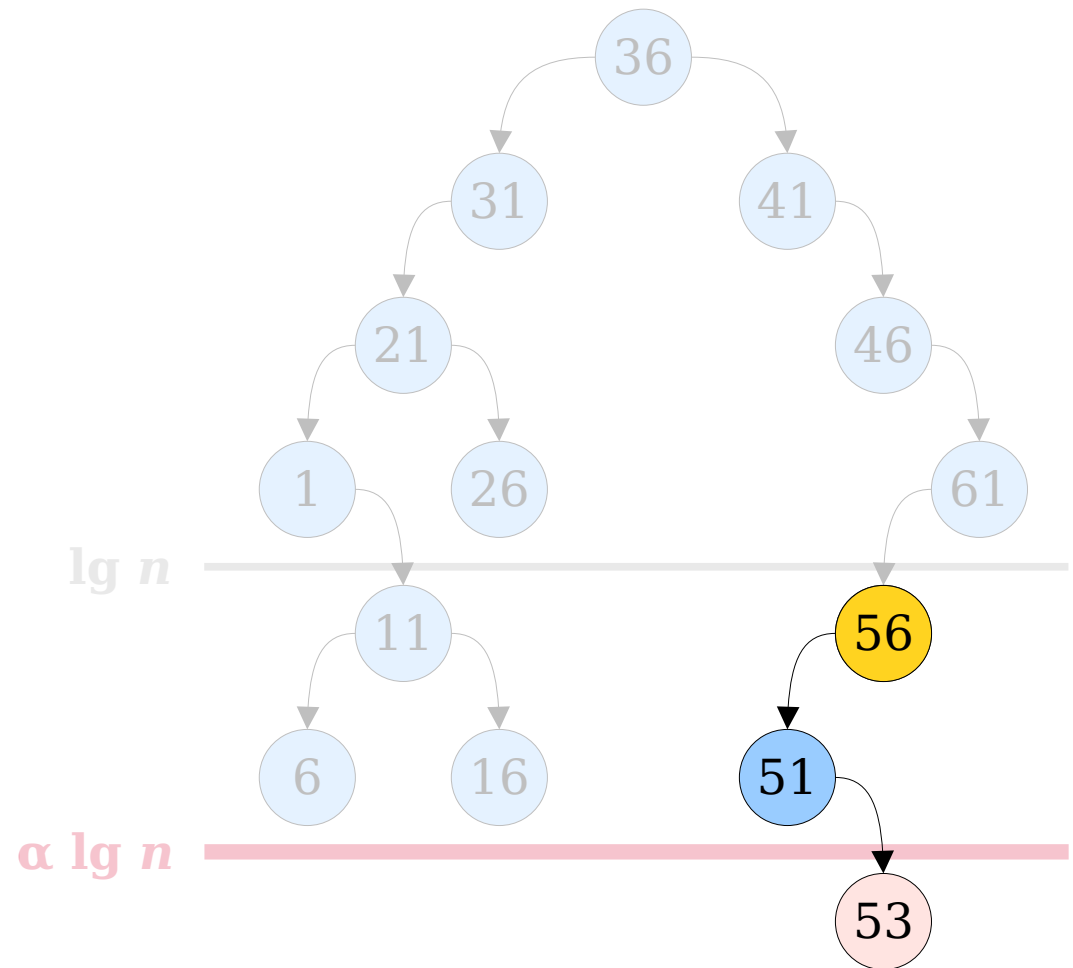
- Our strategy relies on the following claim:

Optimally rebuilding the subtree rooted at the scapegoat node ensures that, as a whole, the tree has height at most $\alpha \lg n$.

- This turns out to not be too difficult to prove. Let's break it down into pieces.

Scapegoat Rebuilding

- Suppose we insert a node that causes the $\alpha \lg n$ size limit to be violated.
- Just before we inserted that node, all other nodes in the tree were at height $\alpha \lg n$ or below.
- That means each other node is at depth $\lfloor \alpha \lg n \rfloor$, and our new node is at depth $\lfloor \alpha \lg n \rfloor + 1$.
- Now, look at the scapegoat node and its subtree.
- Because our offending node is only one level too deep, we just need to show that optimally rebuilding the scapegoat subtree reduces its depth by at least one.



Scapegoat Rebuilding

- Let v be our scapegoat node. Since it's not α -balanced, we know that

$$\text{height}_{\text{before}}(v) > \alpha \lg \text{size}(v).$$

- Let r be the root of the subtree we get after rebuilding at v . Because we rebuilt v 's tree perfectly, we know that

$$\lg \text{size}(v) \geq \text{height}_{\text{after}}(r).$$

- Putting this together gives us that

$$\text{height}_{\text{before}}(v) > \alpha \lg \text{size}(v) > \lg \text{size}(v) \geq \text{height}_{\text{after}}(r).$$

- This means that

$$\text{height}_{\text{before}}(v) > \text{height}_{\text{after}}(r).$$

- Therefore, the height of v 's subtree after rebuilding has decreased by at least one, so overall balance is restored.

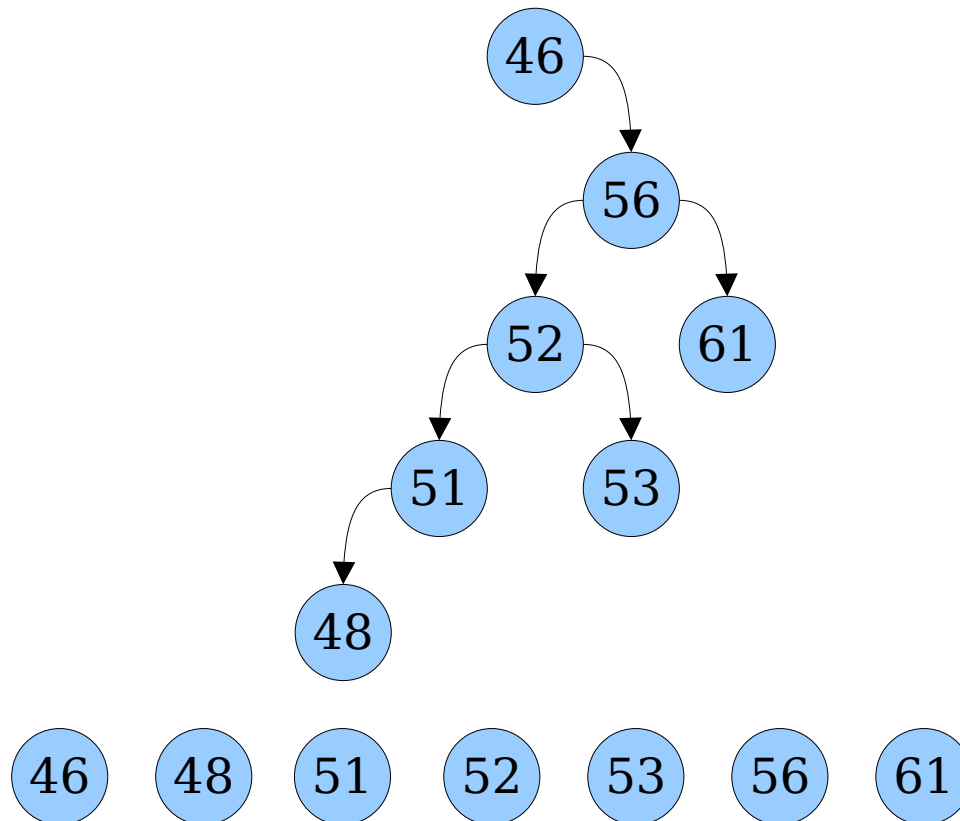
The Cost of Rebuilding

The Cost of Rebuilding

- Once we've identified the scapegoat node, we need to rebuild the subtree rooted at that node as a perfectly-balanced BST.
- How quickly can we do this?

The Cost of Rebuilding

- Run an inorder traversal over the subtree and form an array of its nodes in sorted order.



The Cost of Rebuilding

- Run an inorder traversal over the subtree and form an array of its nodes in sorted order.



The Cost of Rebuilding

- Run an inorder traversal over the subtree and form an array of its nodes in sorted order.
- Use the following recursive algorithm to build an optimal tree:
 - If there are no nodes left, return an empty tree.
 - Otherwise, put the median element at the root of the tree, and recursively build its left and right subtrees optimally.



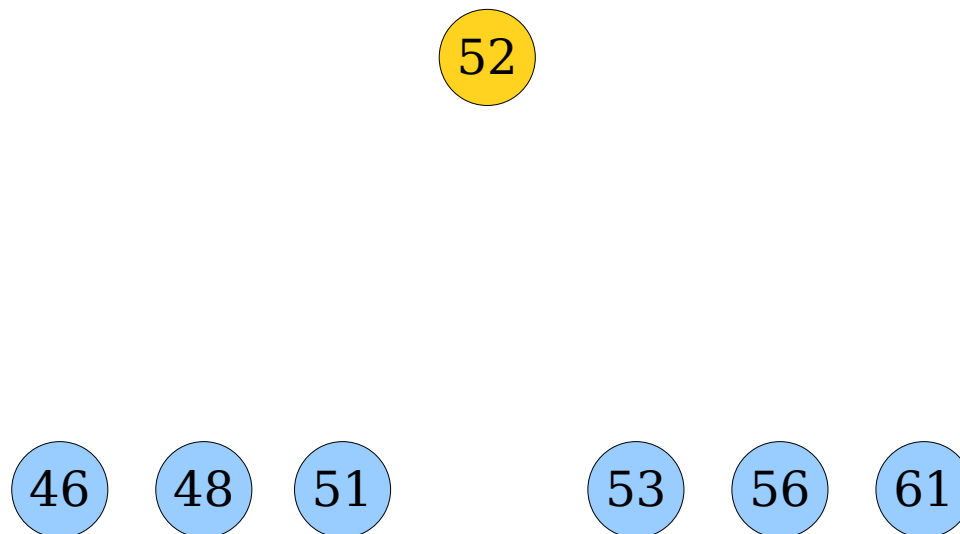
The Cost of Rebuilding

- Run an inorder traversal over the subtree and form an array of its nodes in sorted order.
- Use the following recursive algorithm to build an optimal tree:
 - If there are no nodes left, return an empty tree.
 - Otherwise, put the median element at the root of the tree, and recursively build its left and right subtrees optimally.



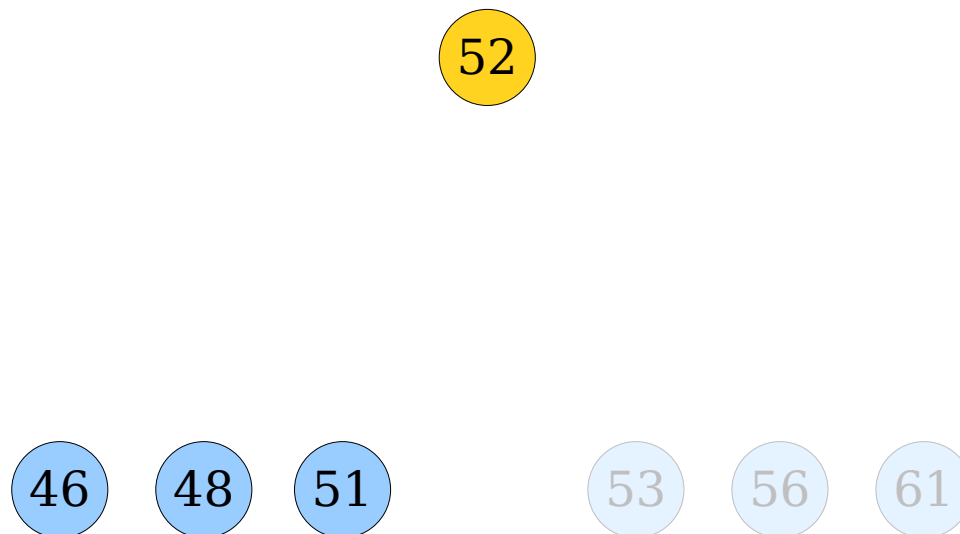
The Cost of Rebuilding

- Run an inorder traversal over the subtree and form an array of its nodes in sorted order.
- Use the following recursive algorithm to build an optimal tree:
 - If there are no nodes left, return an empty tree.
 - Otherwise, put the median element at the root of the tree, and recursively build its left and right subtrees optimally.



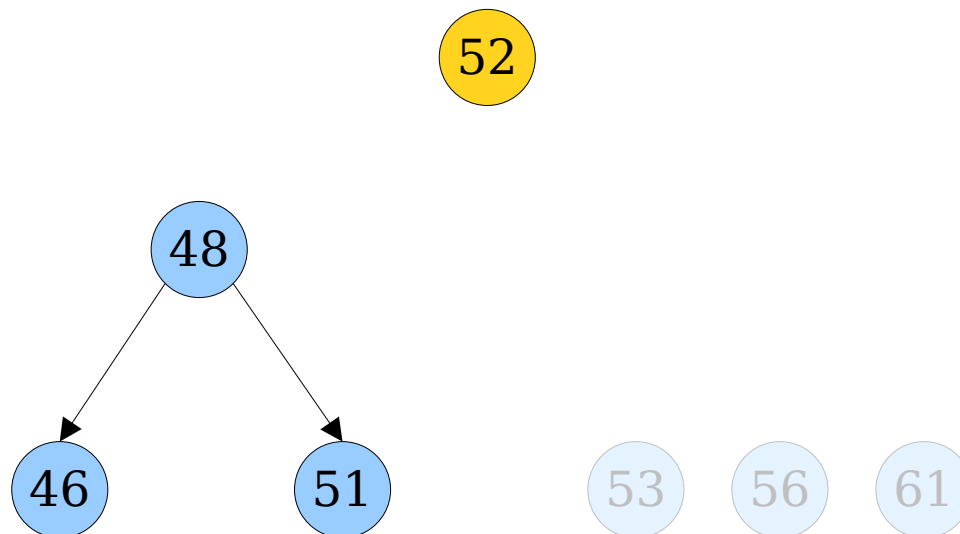
The Cost of Rebuilding

- Run an inorder traversal over the subtree and form an array of its nodes in sorted order.
- Use the following recursive algorithm to build an optimal tree:
 - If there are no nodes left, return an empty tree.
 - Otherwise, put the median element at the root of the tree, and recursively build its left and right subtrees optimally.



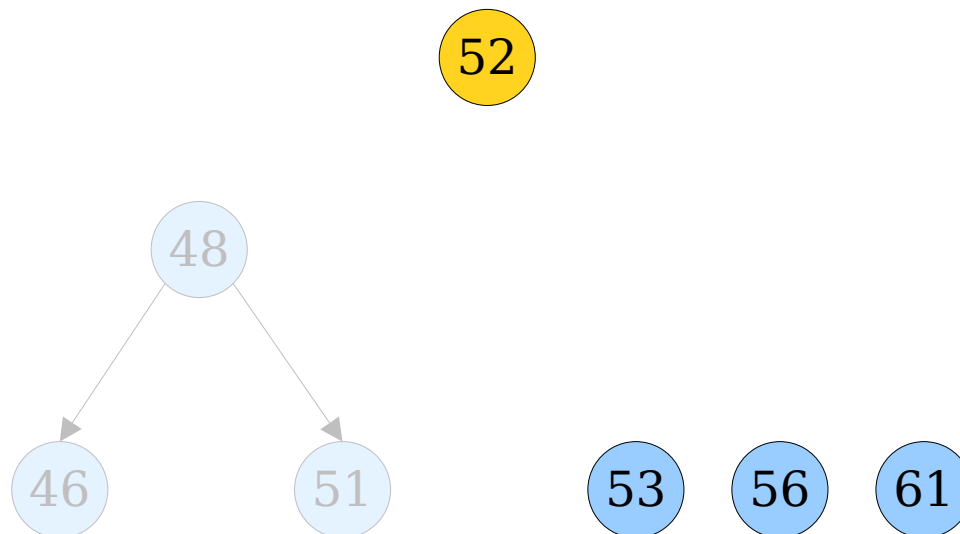
The Cost of Rebuilding

- Run an inorder traversal over the subtree and form an array of its nodes in sorted order.
- Use the following recursive algorithm to build an optimal tree:
 - If there are no nodes left, return an empty tree.
 - Otherwise, put the median element at the root of the tree, and recursively build its left and right subtrees optimally.



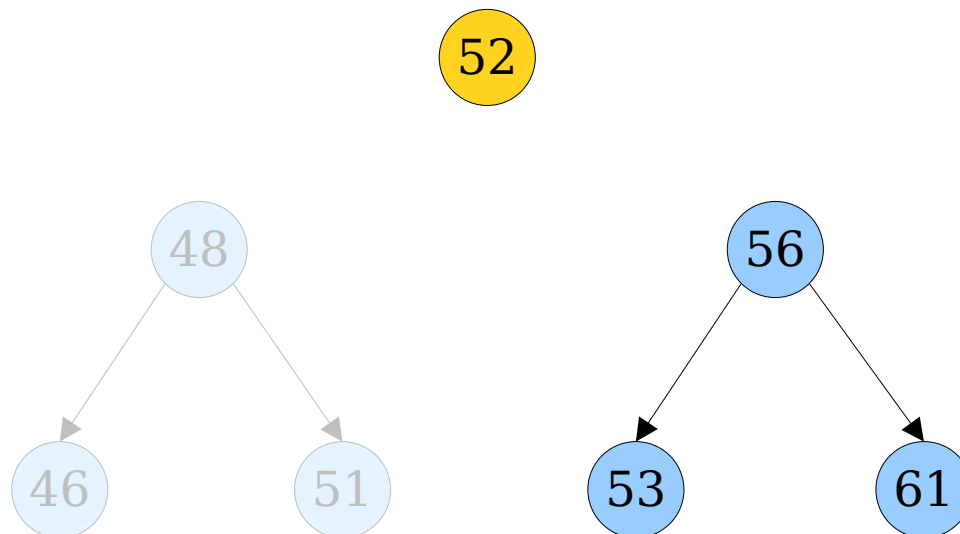
The Cost of Rebuilding

- Run an inorder traversal over the subtree and form an array of its nodes in sorted order.
- Use the following recursive algorithm to build an optimal tree:
 - If there are no nodes left, return an empty tree.
 - Otherwise, put the median element at the root of the tree, and recursively build its left and right subtrees optimally.



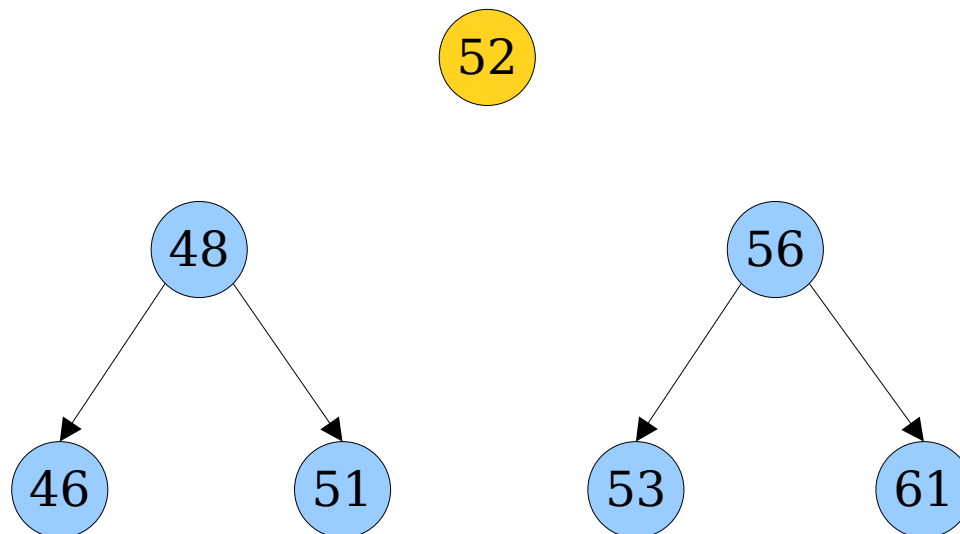
The Cost of Rebuilding

- Run an inorder traversal over the subtree and form an array of its nodes in sorted order.
- Use the following recursive algorithm to build an optimal tree:
 - If there are no nodes left, return an empty tree.
 - Otherwise, put the median element at the root of the tree, and recursively build its left and right subtrees optimally.



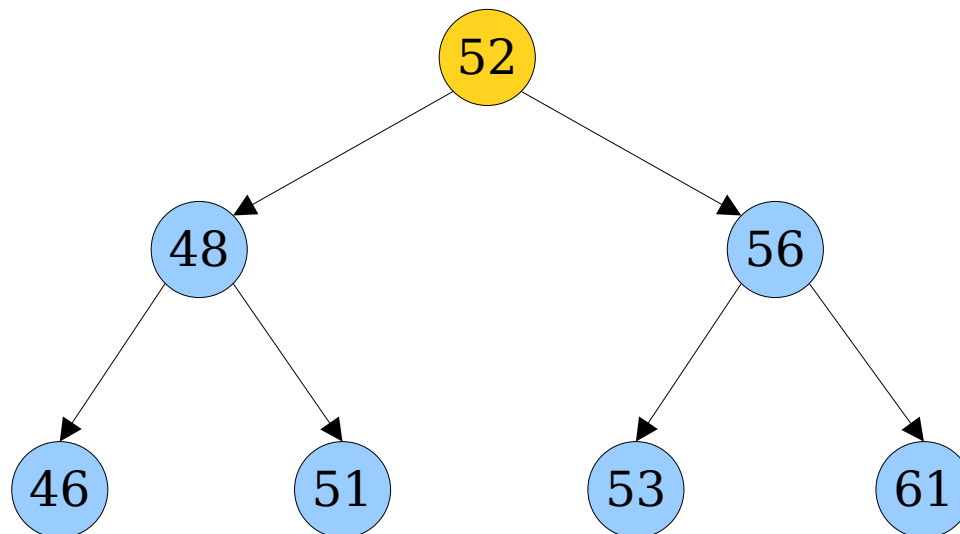
The Cost of Rebuilding

- Run an inorder traversal over the subtree and form an array of its nodes in sorted order.
- Use the following recursive algorithm to build an optimal tree:
 - If there are no nodes left, return an empty tree.
 - Otherwise, put the median element at the root of the tree, and recursively build its left and right subtrees optimally.



The Cost of Rebuilding

- Run an inorder traversal over the subtree and form an array of its nodes in sorted order.
- Use the following recursive algorithm to build an optimal tree:
 - If there are no nodes left, return an empty tree.
 - Otherwise, put the median element at the root of the tree, and recursively build its left and right subtrees optimally.



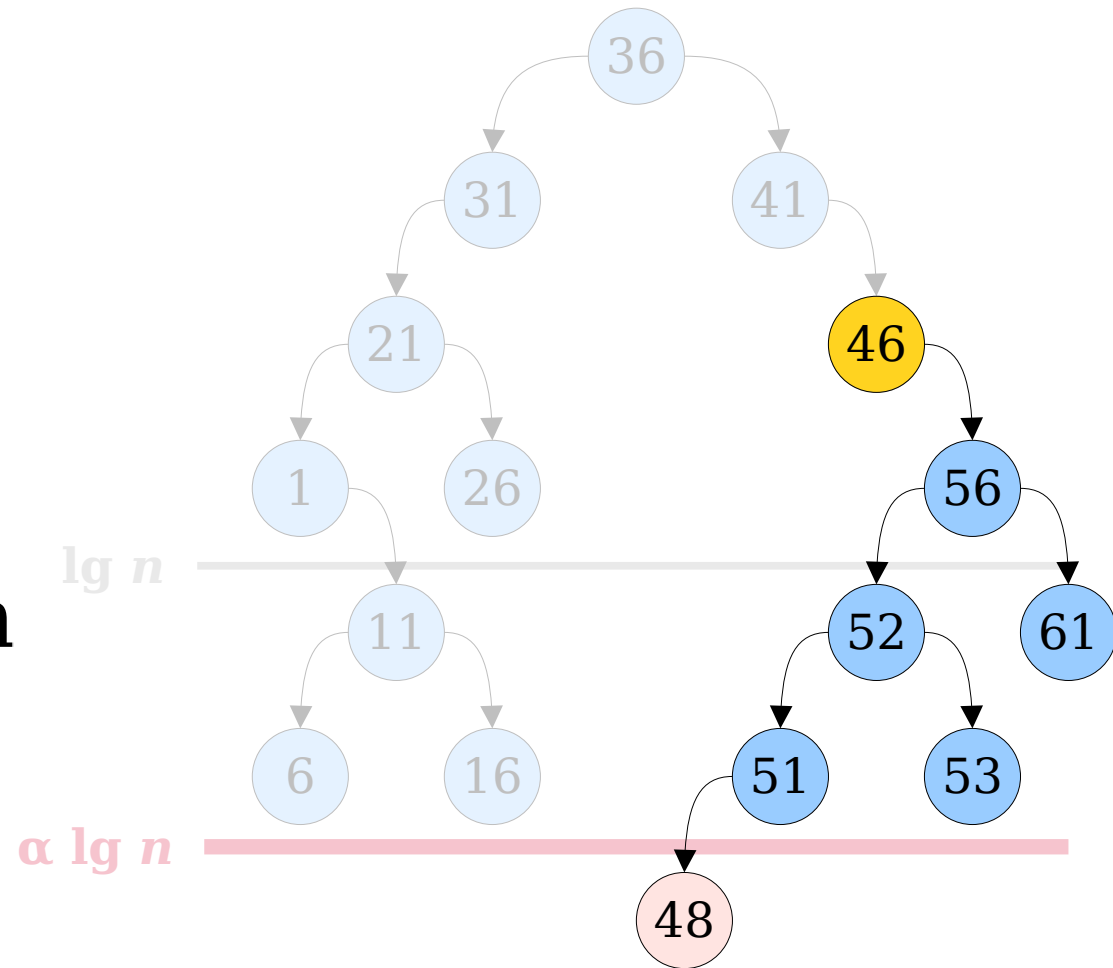
The Cost of Rebuilding

- Run an inorder traversal over the subtree and form an array of its nodes in sorted order.
- Use the following recursive algorithm to build an optimal tree:
 - If there are no nodes left, return an empty tree.
 - Otherwise, put the median element at the root of the tree, and recursively build its left and right subtrees optimally.
- The cost of this strategy is $O(\text{size}(v))$, where v is the node at the root of the subtree.
 - Quick way to see this: the inorder traversal takes time $O(\text{size}(v))$ because there are $\text{size}(v)$ nodes visited, and the recursive algorithm has the recurrence $T(m) = 2T(m / 2) + O(1)$.
- This is the simplest algorithm to optimally rebuild the tree, but others exist that are faster in practice or more space-efficient. Look up the **Galperin-Rivest** or **Day-Stout-Warren** algorithms for other ways to do this in time $O(\text{size}(v))$ in less space.

Finding the Scapegoat Node

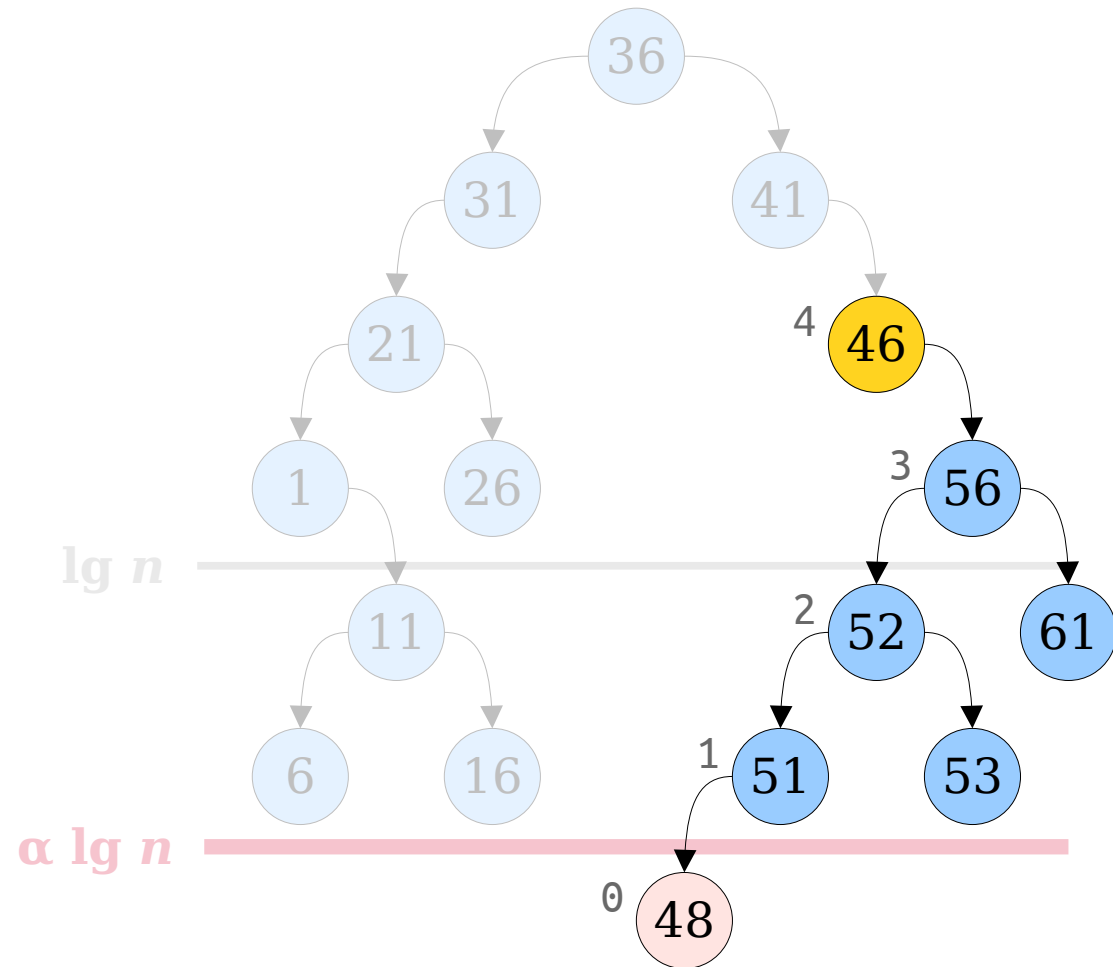
Finding the Scapegoat

- **Recall:** The scapegoat node is the deepest node on the access path that isn't α -balanced.
- How efficiently can we identify this node?



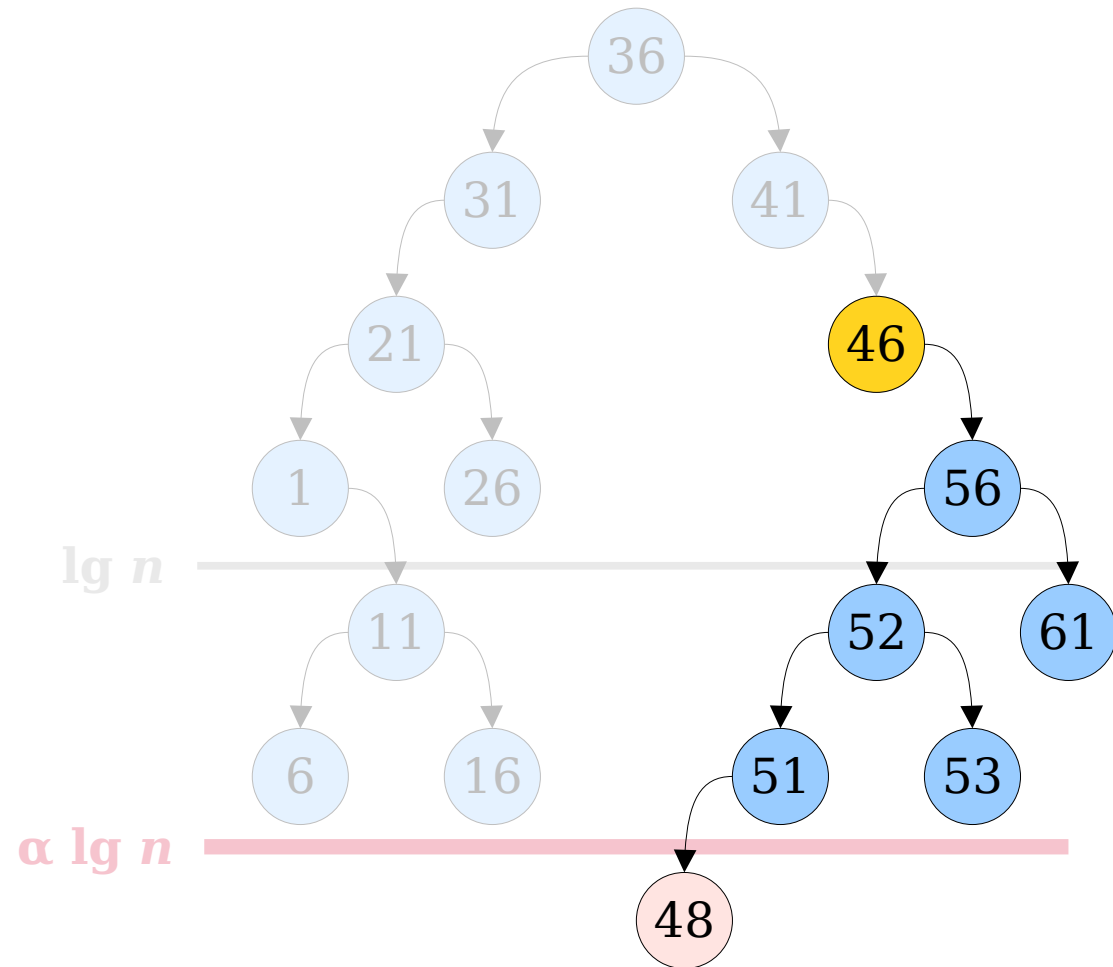
Finding the Scapegoat

- We need to check if $\text{height}(v) > \alpha \lg \text{size}(v)$.
- **Observation:** For each node v on the access path, $\text{height}(v)$ is the number of steps between v and the newly-added node.
 - This can be computed by counting upward from the new node.
- That just leaves computing $\text{size}(v)$.



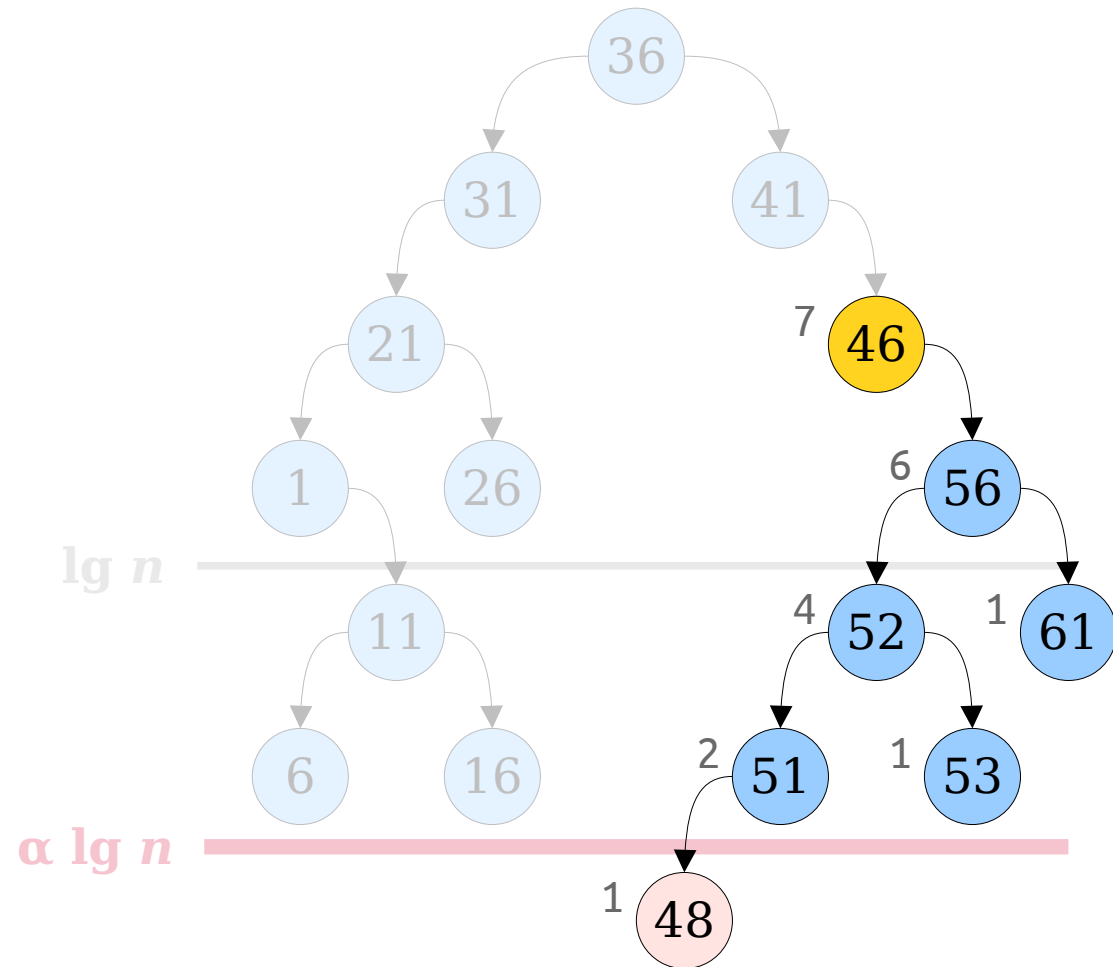
Finding the Scapegoat

- There are two ways we can compute $\text{size}(v)$ for the nodes on the access path.
- **Approach 1:** Augment each node with the number of nodes in its subtree.
 - (This can be done without changing the cost of an insertion or deletion.)
- We can then read $\text{size}(v)$ by looking at the cached value.
- This has the disadvantage of requiring an extra integer in each node of the tree.



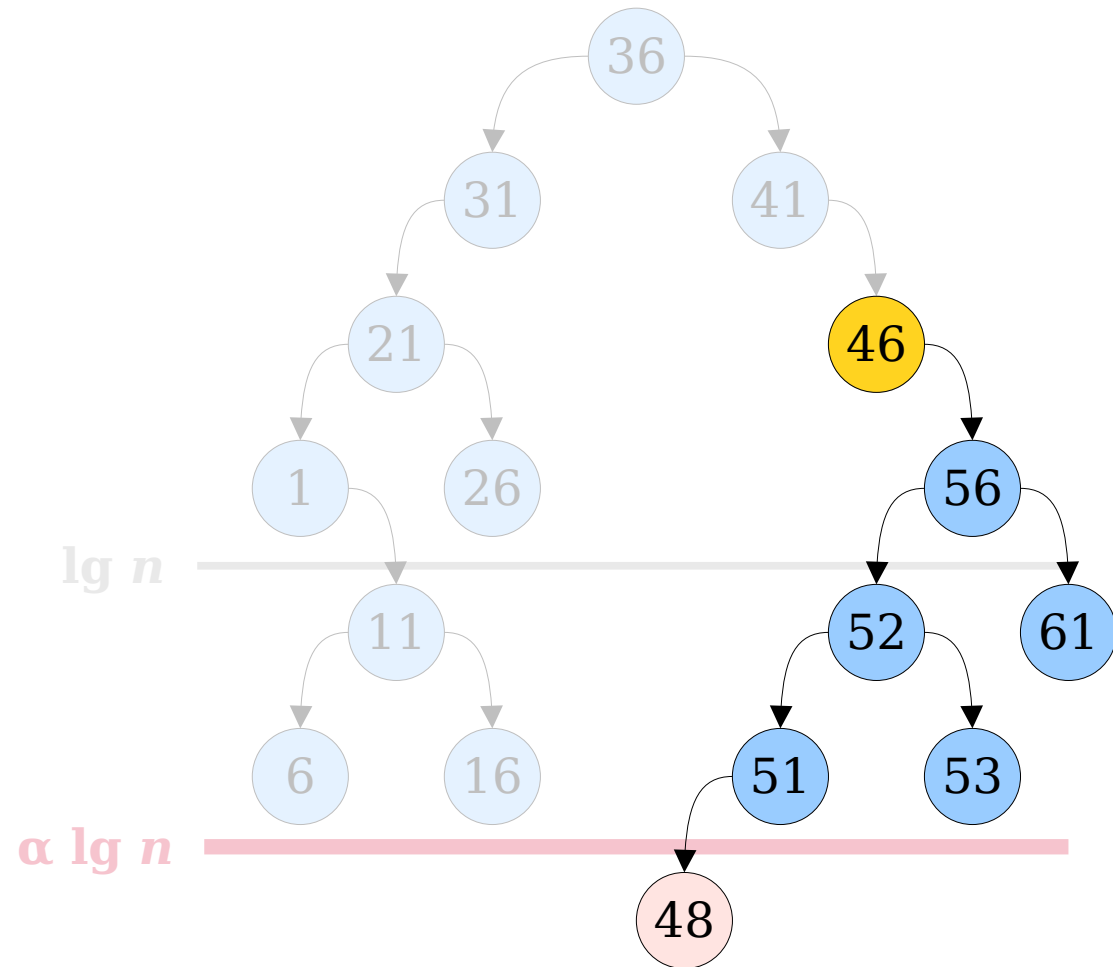
Finding the Scapegoat

- **Approach 2:** Compute these values bottom-up.
- Start with a total of 1 for the newly-added node.
- Each time we move upward a step, run a DFS in the opposite subtree to count the number of nodes there.
- Once we hit the scapegoat node v , we'll have done $O(\text{size}(v))$ total work counting nodes.



Finding the Scapegoat

- **Approach 1** does less work, but requires more storage in each node.
- **Approach 2** does more work, but means each node just stores data and two child pointers.
- Which of these ends up being more important depends on a mix of engineering constraints and personal preference.



Analyzing Efficiency

Analyzing Efficiency

- Based on what we've seen so far, the cost of an insertion is
 - $O(\log n)$ if the insertion keeps us below the $\alpha \lg n$ height threshold, and
 - $O(\log n + \text{size}(v))$ if we have to rebuild v as a scapegoat.
- The $\text{size}(v)$ term can be as large as n , which may happen if the whole tree has to be rebuilt.
- However, it turns out that we can amortize this $\text{size}(v)$ term away.

Analyzing Efficiency

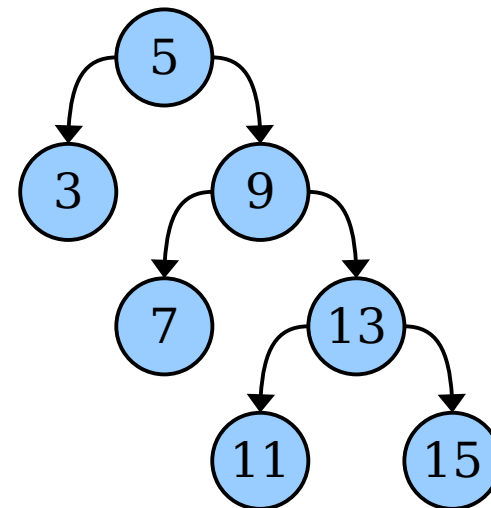
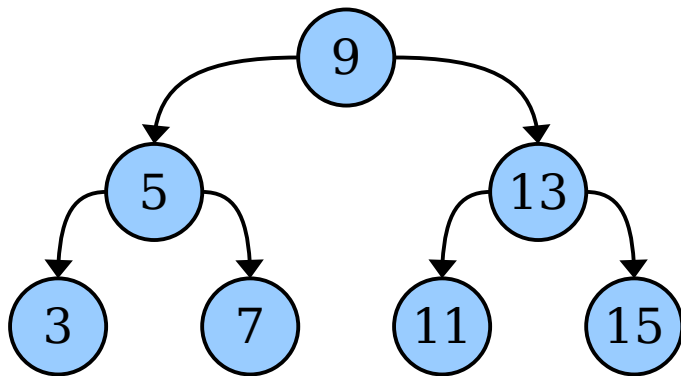
- **Recall:** To perform an amortized analysis, we do the following:
 - Find a potential function Φ that, intuitively, is small when the data structure is “clean” and large when the data structure is “messy.”
 - Compute the value of $\Delta\Phi = \Phi_{\text{after}} - \Phi_{\text{before}}$ for each operation.
 - Assign amortized costs as
$$\text{amortized-cost} = \text{real-cost} + k \cdot \Delta\Phi$$
for some constant k we get to pick.
- Our first step is to find a choice of Φ that’s large when our tree is imbalanced and small when it’s balanced.

Quantifying Imbalance

- Right before we rebuild a scapegoat subtree, that tree is α -imbalanced.
- Right after we rebuild a scapegoat subtree, that tree is perfectly balanced.
- **Goal:** Find a choice of Φ for our tree so that
 - perfectly-balanced trees have low Φ , and
 - α -imbalanced trees have high Φ .
- At this point, we need to do some exploring to see what we find.

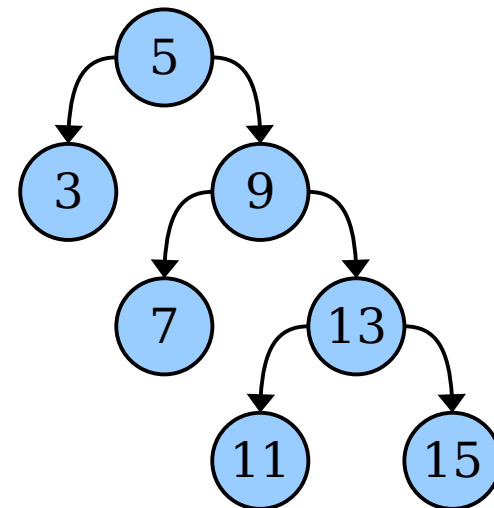
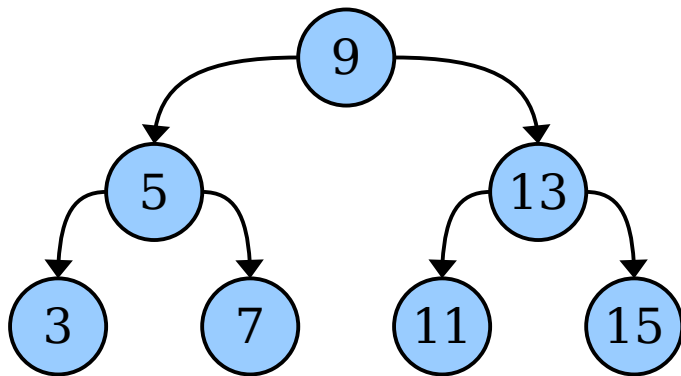
Quantifying Imbalance

- When we talk about “perfectly balanced” trees, what exactly is this “balance” in reference to?
- **Intuition 1:** A perfectly balanced tree is one where each node has roughly the same number of children in its left subtree as in its right subtree.
- **Intuition 2:** An “imbalanced” tree will have nodes whose left and right subtrees have differing numbers of nodes.



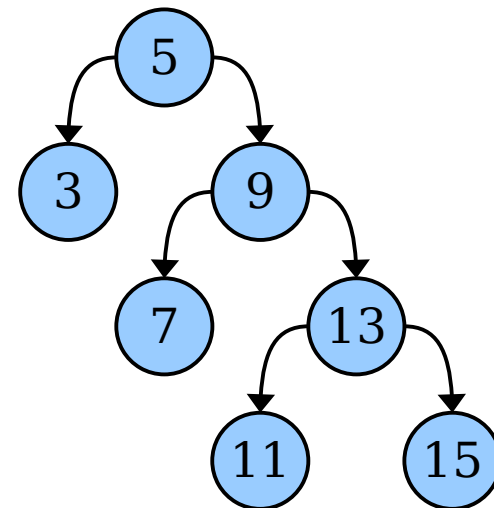
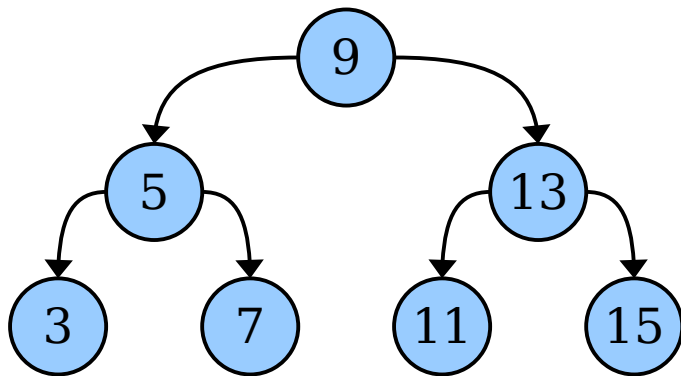
Quantifying Imbalance

- For each node v , define the ***imbalance*** of the node as
$$(v) = |\text{size}(v.\text{left}) - \text{size}(v.\text{right})|.$$
- This gives us a quantitative measure of our more nebulous concept of “imbalance.”



Defining our Potential

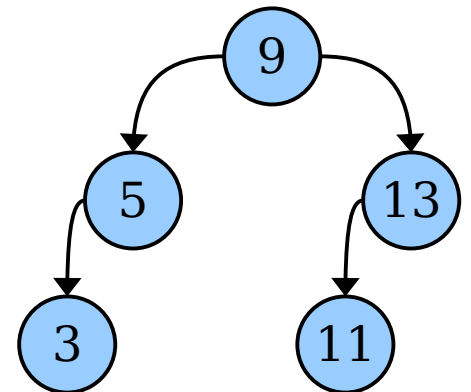
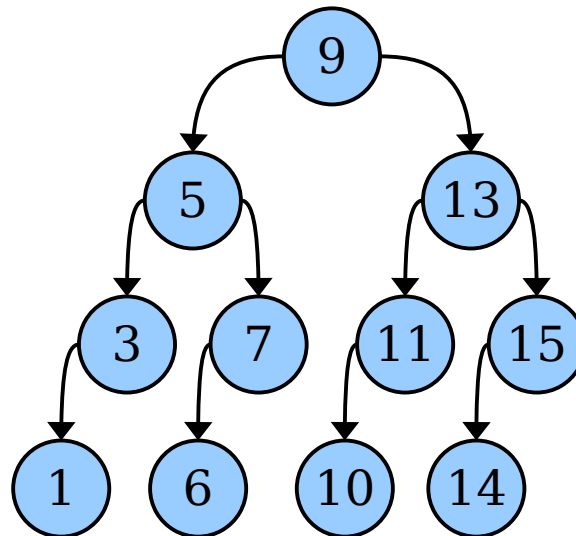
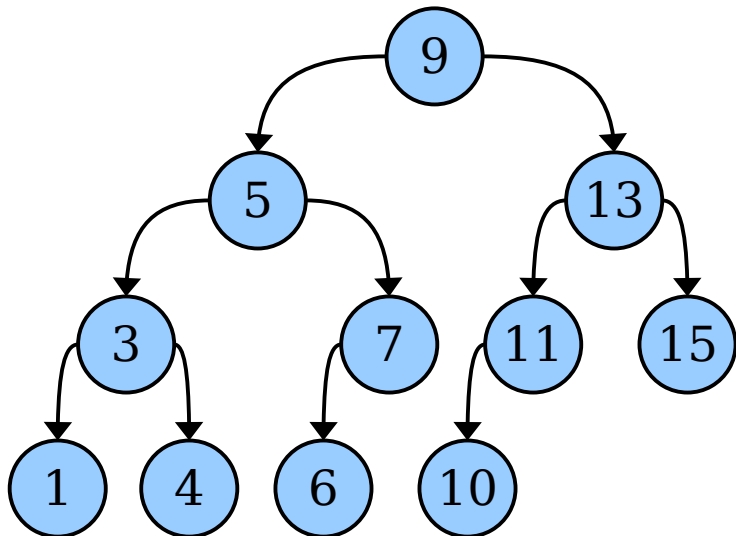
- We're looking for a potential function Φ where
 - a perfectly-balanced tree has low Φ , and
 - an imbalanced tree has progressively higher Φ .
- A balanced tree has $\Phi(v)$ low for all its nodes.
- An imbalanced tree has $\Phi(v)$ high for many nodes.
- **Initial Idea:** Define $\Phi = \sum_v \Phi(v)$.



Defining our Potential

- We've set $\Phi = \sum_v \quad (v)$.
- What is Φ for the three trees shown below?

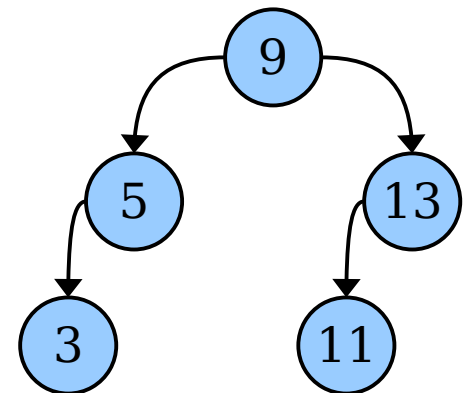
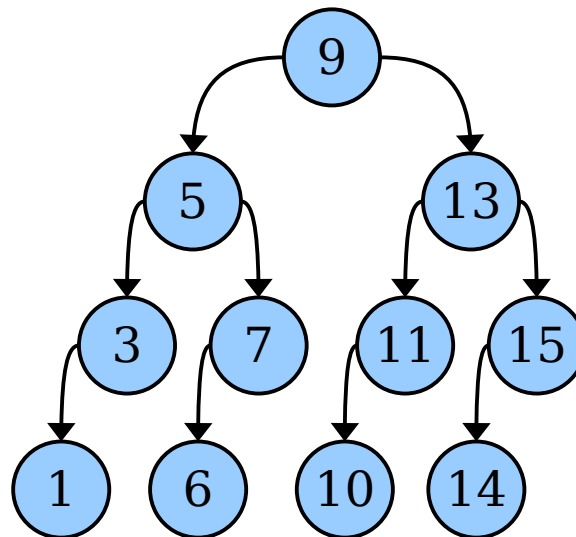
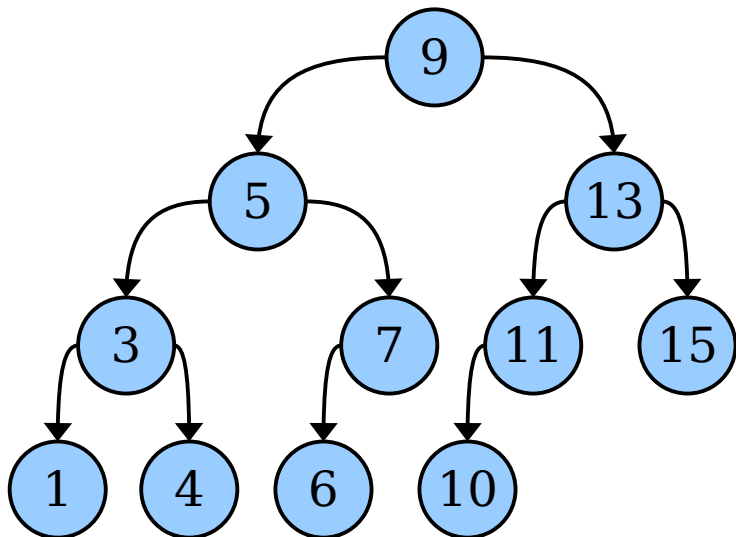
Formulate a hypothesis!



Defining our Potential

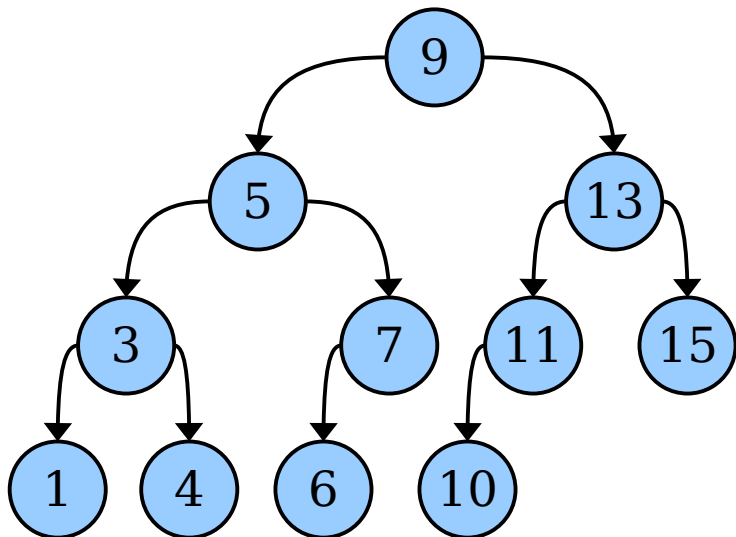
- We've set $\Phi = \sum_v \quad (v)$.
- What is Φ for the three trees shown below?

Discuss with your neighbors!

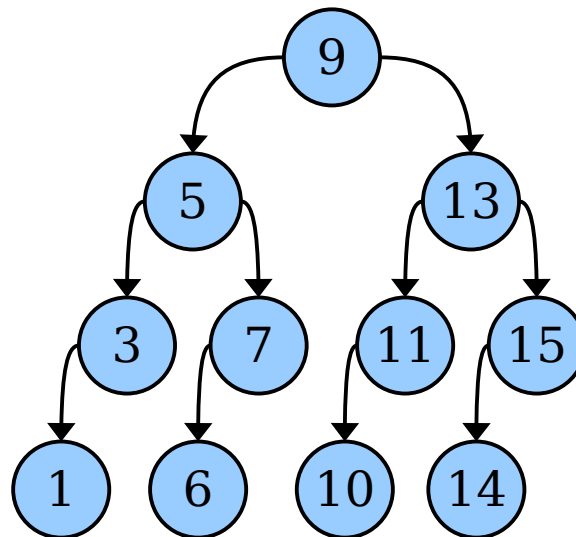


Defining our Potential

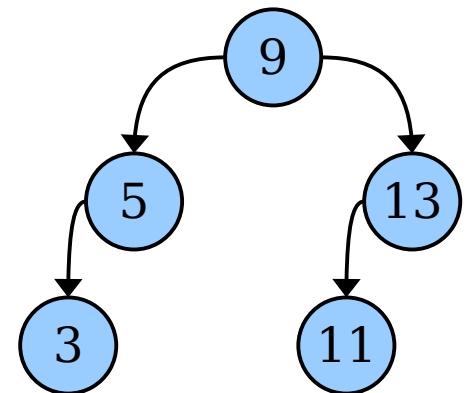
- We've set $\Phi = \sum_v \quad (v)$.
- What is Φ for the three trees shown below?



$$\Phi = 6$$



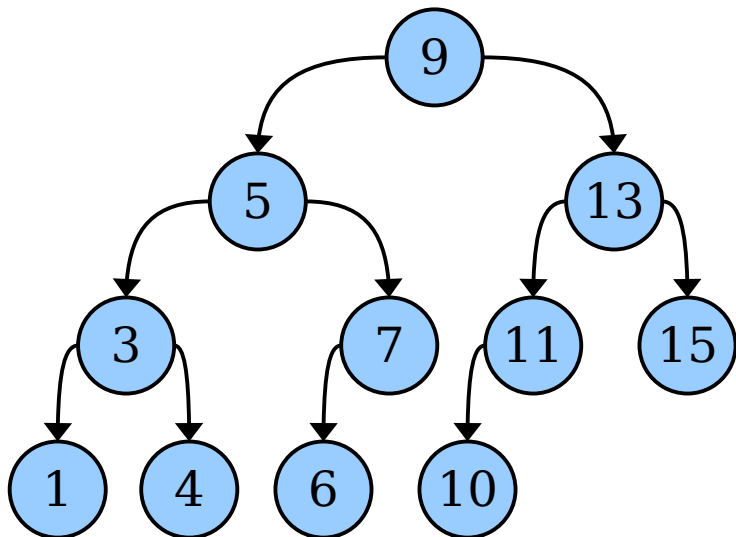
$$\Phi = 4$$



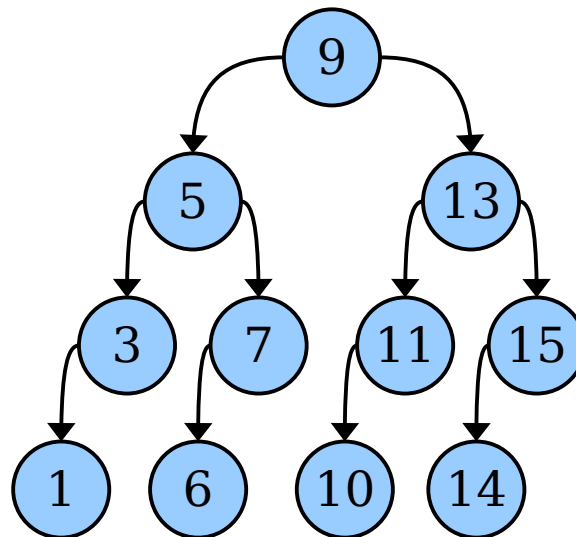
$$\Phi = 2$$

Defining our Potential

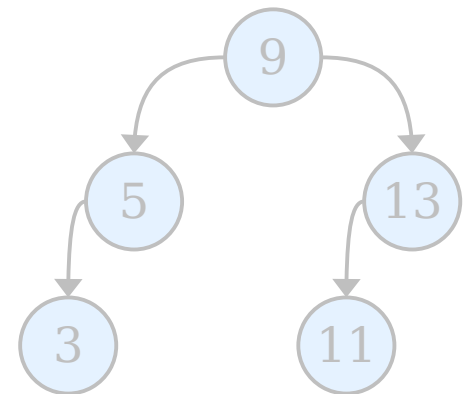
- **Observation 1:** Two trees that fill their rows as efficiently as possible may have different potentials.
- This means that when we rebalance trees, we need to make sure to equalize the number of nodes in the left and right subtrees of each node.



$$\Phi = 6$$



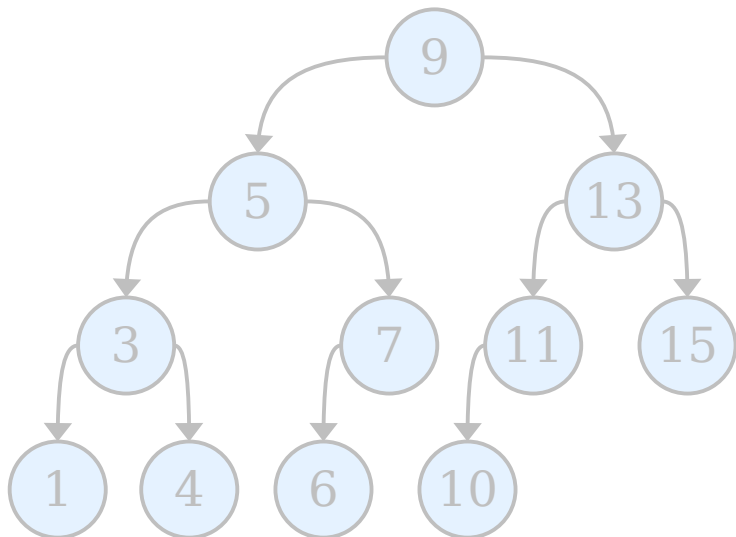
$$\Phi = 4$$



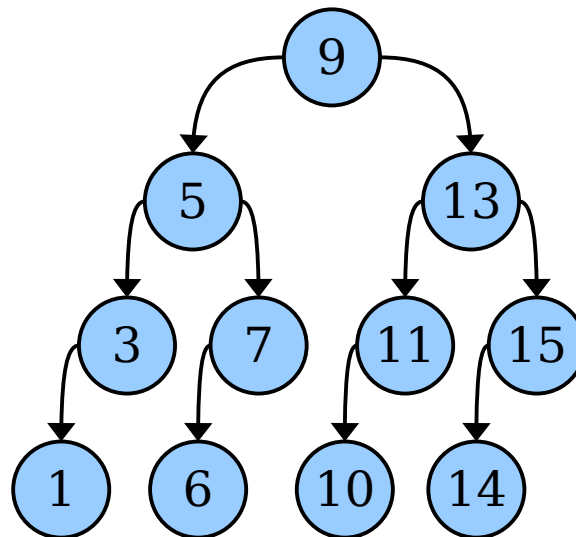
$$\Phi = 2$$

Defining our Potential

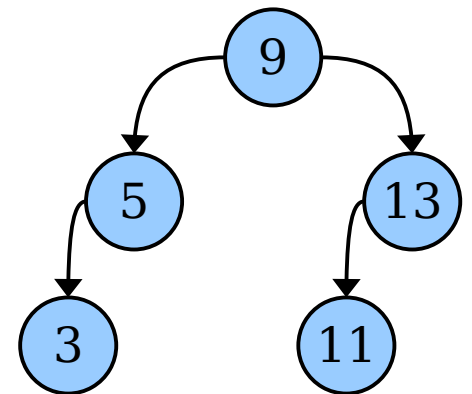
- **Observation 2:** The potential of a perfectly-balanced tree can grow as a function of its number of nodes.
- Ideally, both of these trees should have potential 0, indicating “perfectly balanced.” The potential shouldn’t depend on the number of nodes in the tree.



$$\Phi = 6$$



$$\Phi = 4$$



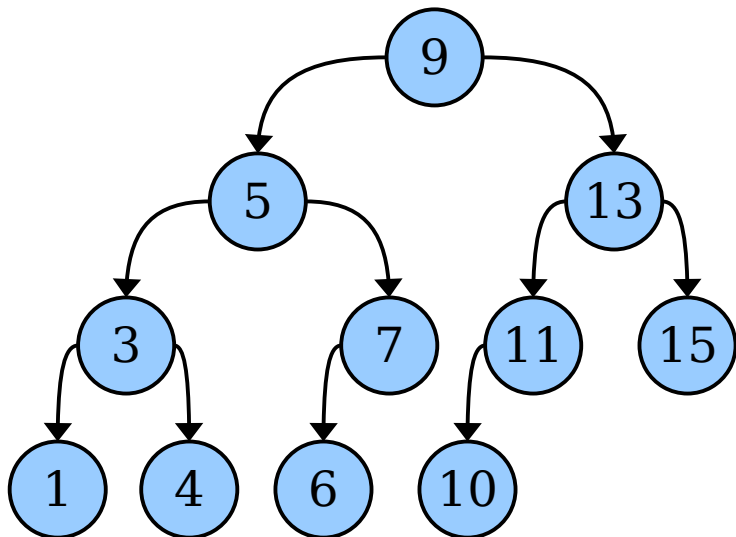
$$\Phi = 2$$

Defining our Potential

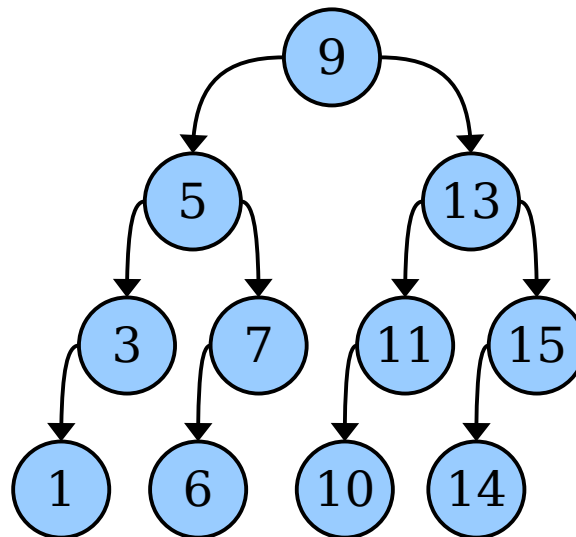
- To account for otherwise balanced trees with extra nodes in their bottom layers, let's define $\phi(v)$ as
 - $\phi(v) = 0$ if $h(v) \leq 1$.
 - $\phi(v) = h(v)$ otherwise.
- ***Revised Idea:*** Set $\Phi = \sum_v \phi(v)$.

Defining our Potential

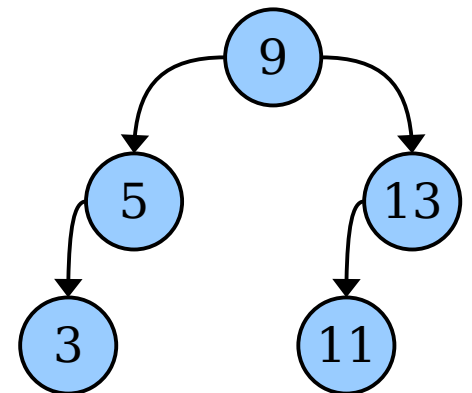
- We're now using $\Phi = \sum_v \phi'(v)$.
- What is Φ for the three trees shown below?
- **Intuition:** If a subtree rooted at v is perfectly balanced, then $\phi'(v) = 0$.



$$\Phi = 2$$



$$\Phi = 0$$



$$\Phi = 0$$

Analyzing Scapegoat Trees

- Now that we have a definition of Φ , we can look at the amortized cost of an insertion.
- We need to consider two cases:
 - **Case 1:** The insertion doesn't trigger a rebuild.
 - **Case 2:** The insertion triggers a rebuild.
- Intuitively, we're hoping that Case 1 has a small positive $\Delta\Phi$ (messes accumulate slowly) and that Case 2 has a large negative $\Delta\Phi$ (messes get cleaned up quickly).
- Let's run the numbers!

Analyzing Scapegoat Trees

- **Case 1:** Our insertion does not trigger a rebuild.
- Recall that

$$\text{amortized-cost} = \text{real-cost} + k \cdot \Delta\Phi$$

for a constant k that we get to pick.

What are *real-cost* and $\Delta\Phi$,
as a function of n ?

Formulate a hypothesis!

Analyzing Scapegoat Trees

- **Case 1:** Our insertion does not trigger a rebuild.
- Recall that

$$\text{amortized-cost} = \text{real-cost} + k \cdot \Delta\Phi$$

for a constant k that we get to pick.

What are *real-cost* and $\Delta\Phi$,
as a function of n ?

Discuss with your neighbors!

Analyzing Scapegoat Trees

- **Case 1:** Our insertion does not trigger a rebuild.
- Recall that

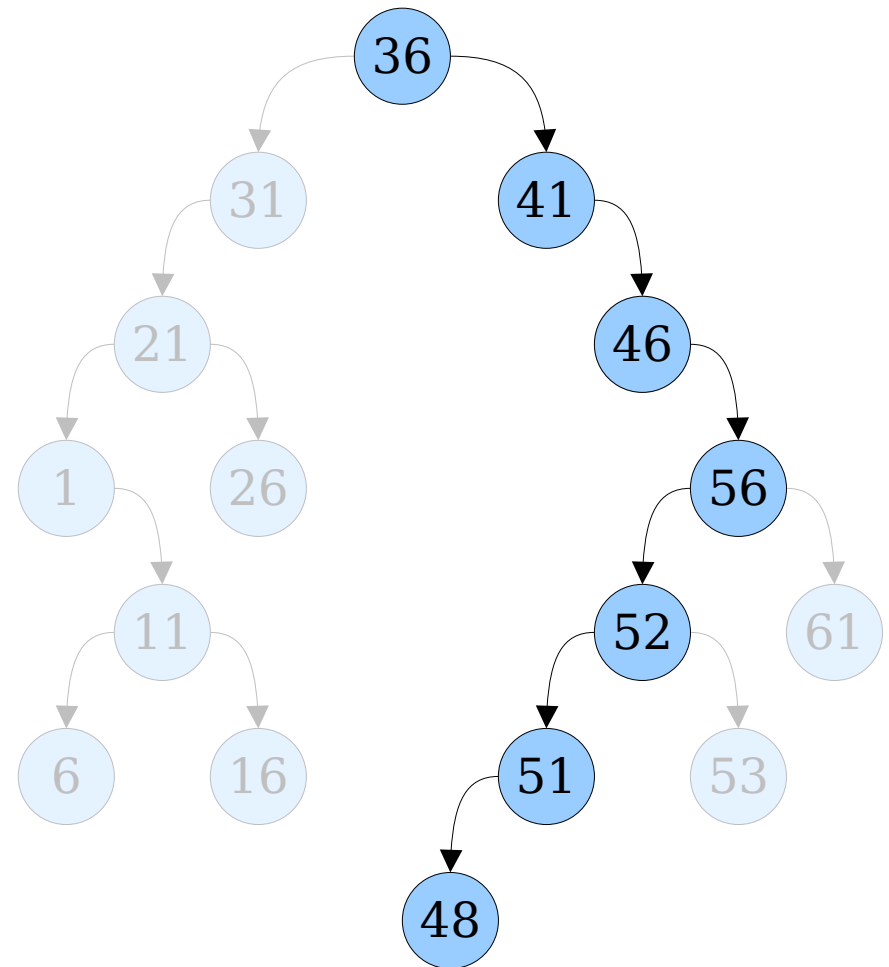
$$\text{amortized-cost} = \text{real-cost} + k \cdot \Delta\Phi$$

for a constant k that we get to pick.

- We're inserting into a tree of height at most $\alpha \lg n$, so our *real-cost* is $O(\log n)$.
- When we insert the node, it changes $\Phi(v)$ by ± 1 for each node v on its access path.
- There are $O(\log n)$ nodes on this access path, and $\Phi(v)$ increases by at most one for each of those nodes. This means $\Delta\Phi$ increases by at most two for each of those nodes.
- Therefore, $\Delta\Phi = O(\log n)$.
- Amortized cost: $O(\log n) + k \cdot O(\log n) = \mathbf{O(\log n)}$.

Analyzing Scapegoat Trees

- **Case 1:** Our insertion does not trigger a rebuild.
- In this case, $\Delta\Phi = O(\log n)$.
- Focus on any one of the new node's ancestors.
- If we rebuild the subtree rooted at that node in the future, we have to do some work to move the new node.
- **Intuition:** The $O(\log n)$ added potential corresponds to paying $O(1)$ work in advance to each of $O(\log n)$ future rebuilds.



Analyzing Scapegoat Trees

- **Case 2:** Our insertion triggers a rebuild.
- Recall that

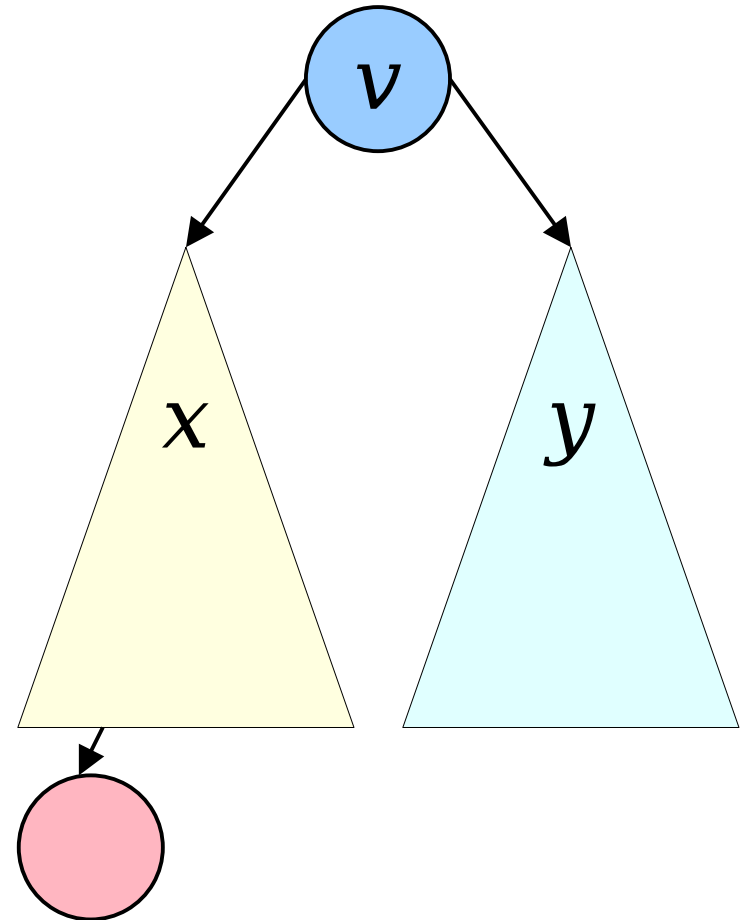
$$\textit{amortized-cost} = \textit{real-cost} + k \cdot \Delta\Phi$$

for a constant k that we pick.

- Here, *real-cost* is $O(\log n + \text{size}(v))$, where v is the scapegoat node.
 - The $O(\log n)$ comes from the cost of the actual insertion.
 - The $O(\text{size}(v))$ is for the cost of rebuilding.
- For this to amortize away, we need $\Delta\Phi$ to be $-\Omega(\text{size}(v))$.
- Our previous intuition tells us this should be the case.
- Let's run the numbers to check.

Analyzing Scapegoat Trees

- Let v be the scapegoat node. We're interested in (v) .
- One of v 's children is a tree containing our newly-inserted node. Call that subtree x .
- Call v 's other child y .
- **Goal:** Determine $(v) = |\text{size}(x) - \text{size}(y)|$.

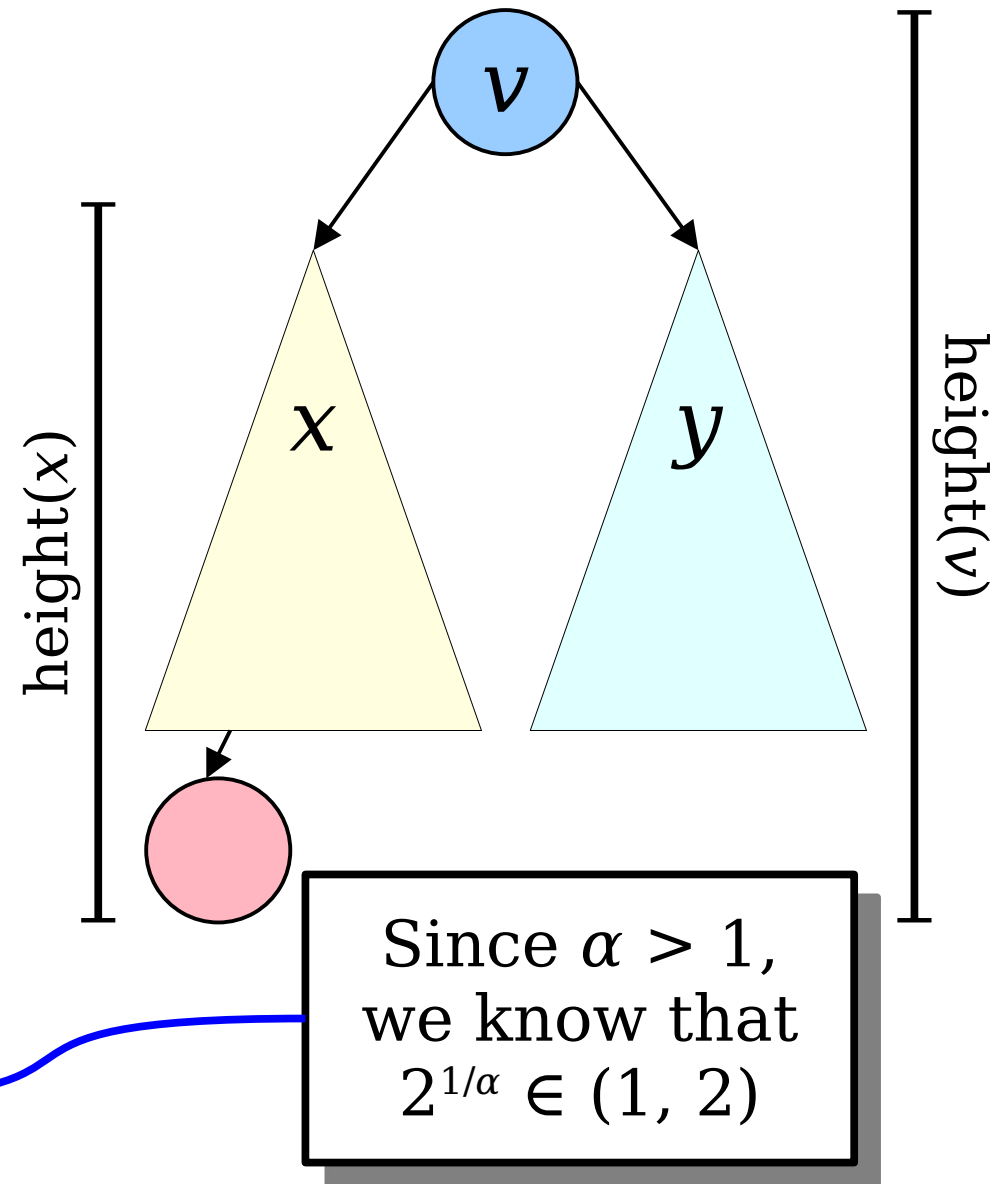


Analyzing Scapegoat Trees

- Since v is α -imbalanced, we know
$$\text{height}(v) > \alpha \lg \text{size}(v).$$
- v is the **deepest** α -imbalanced node on the access path. This means x is α -balanced, so
$$\text{height}(x) \leq \alpha \lg \text{size}(x).$$
- Since the newly-inserted node is the deepest node in v 's subtree, we know that

$$\text{height}(v) = \text{height}(x) + 1.$$

- Putting all this together gives
$$\alpha \lg \text{size}(v) < \alpha \lg \text{size}(x) + 1.$$
- That in turn means that
$$\text{size}(v) < \text{size}(x) \cdot 2^{1/\alpha}.$$



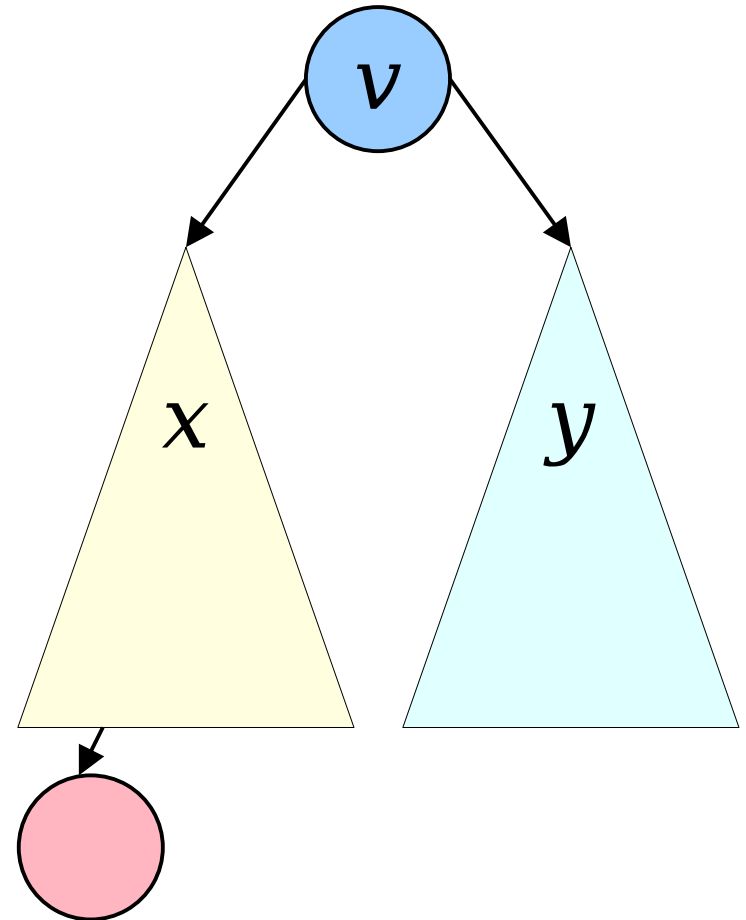
Analyzing Scapegoat Trees

- We just proved that
$$\text{size}(v) < \text{size}(x) \cdot 2^{1/\alpha}.$$
- We also know that
$$\text{size}(v) = 1 + \text{size}(x) + \text{size}(y).$$
- That means
$$\text{size}(x) + \text{size}(y) < \text{size}(x) \cdot 2^{1/\alpha}.$$
- Therefore,
$$\text{size}(y) < \text{size}(x) \cdot (2^{1/\alpha} - 1).$$

Since $2^{1/\alpha} \in (1, 2)$, we know $2^{1/\alpha} - 1 \in (0, 1)$.

So y must have fewer nodes than x .

(Surprising, but true!
Explore and see why!)



Analyzing Scapegoat Trees

- We just proved that

$$\text{size}(v) < \text{size}(x) \cdot 2^{1/\alpha}.$$

- We also know that

$$\text{size}(v) = 1 + \text{size}(x) + \text{size}(y).$$

- That means

$$\text{size}(x) + \text{size}(y) < \text{size}(x) \cdot 2^{1/\alpha}.$$

- Therefore,

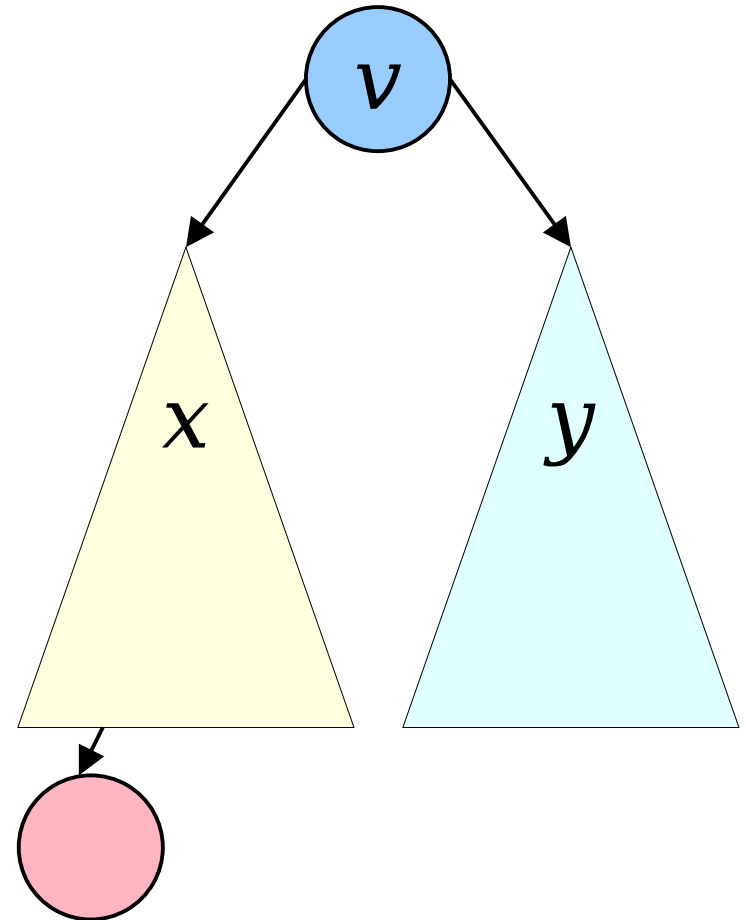
$$\text{size}(y) < \text{size}(x) \cdot (2^{1/\alpha} - 1).$$

- This means that

$$\begin{aligned} (v) &= |\text{size}(x) - \text{size}(y)| \\ &> \text{size}(x) - \text{size}(x) \cdot (2^{1/\alpha} - 1) \\ &= \text{size}(x) \cdot (2 - 2^{1/\alpha}). \end{aligned}$$

- Combined with the initial inequality, this gives us that

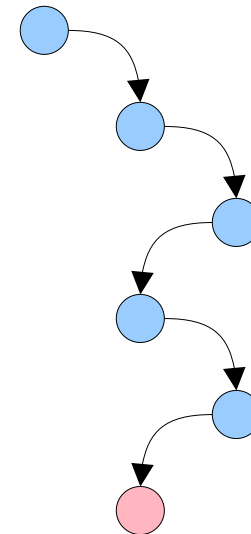
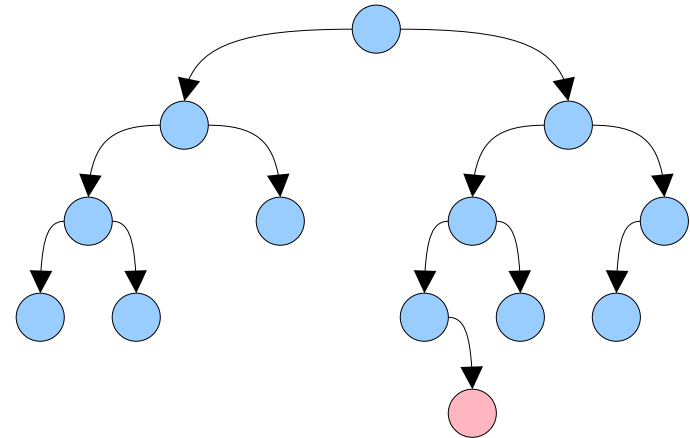
$$(v) > \text{size}(v) \cdot (2^{1 - 1/\alpha} - 1).$$



$$\begin{aligned} 2^{1 - 1/\alpha} &\in (1, 2), \\ \text{So } 2^{1 - 1/\alpha} - 1 &\in (0, 1). \end{aligned}$$

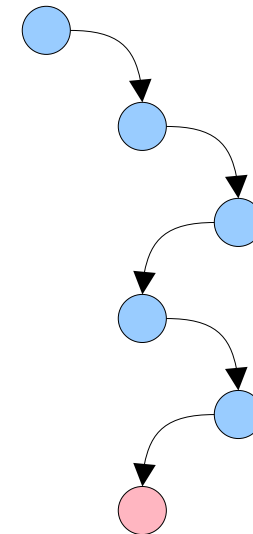
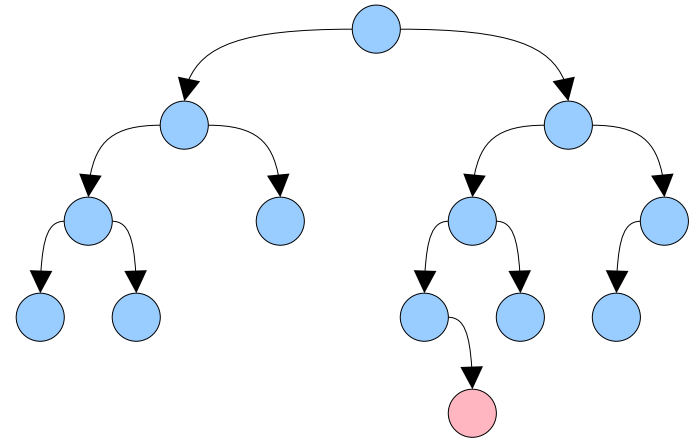
Analyzing Scapegoat Trees

- We've just concluded that
$$\beta(v) > \text{size}(v) \cdot (2^{1 - 1/\alpha} - 1)$$
- Let's take a minute to check our math.
- If α is close to 1, we're requiring the trees to be very tightly balanced. Therefore, when an imbalance occurs, we'd expect $\beta(v)$ to be small relative to $\text{size}(v)$.
- If α is large, we're allowing for huge imbalances in the trees. Therefore, when a node is too deep, we expect the tree it's a part of to be highly imbalanced, so we'd expect $\beta(v)$ to be large relative to $\text{size}(v)$.



Analyzing Scapegoat Trees

- We've just concluded that
$$b(v) > \text{size}(v) \cdot (2^{1 - 1/\alpha} - 1)$$
- Notice that for any fixed value of α that we have
$$b(v) = \Omega(\text{size}(v)).$$
- In other words, the scapegoat node always has an imbalance that is (at least) linear in the size of its subtree.
- We can then backcharge the linear work required to optimally rebuild it to the operations that caused the imbalance in the first place.



Analyzing Scapegoat Trees

- We can now work out the amortized cost of an insertion that triggers a rebuild.
 - Actual cost of inserting a new node: $O(\log n)$.
 - Actual cost of rebuilding at the scapegoat node: $O(\text{size}(v))$.
 - Change in potential: $\Delta\Phi < -\Omega(\text{size}(v))$.
- Amortized cost:
$$O(\log n) + O(\text{size}(v)) - k \cdot \Omega(\text{size}(v)).$$
- By tuning k based on the hidden constant factors in the O and Ω terms, we can get them to cancel, leaving an amortized cost of **$O(\log n)$** .

Where We Stand

- Here's the current scorecard for scapegoat trees.
- Intuitively:
 - If you pick α to be smaller, you get a more balanced tree (faster lookups), but the overhead to optimally rebuild subtrees gets bigger (slower insertions).
 - If you pick α to be larger, you get a less balanced tree (slower lookups), but the overhead to optimally rebuild trees is smaller (faster insertions).
- Tuning α appropriately now becomes a matter of engineering.
- **Question:** What about deletions?

Scapegoat Tree

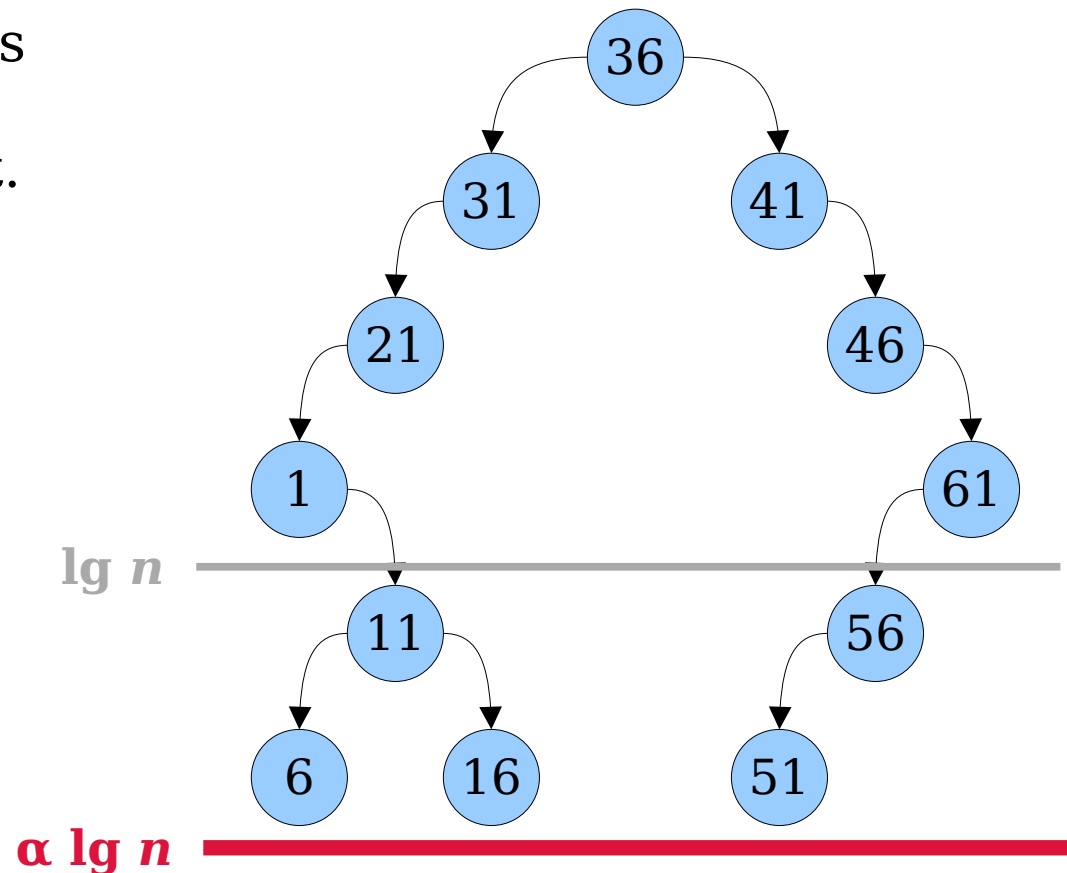
Lookup: $O(\log n)$

Insert: $O(\log n)^*$

* *amortized*

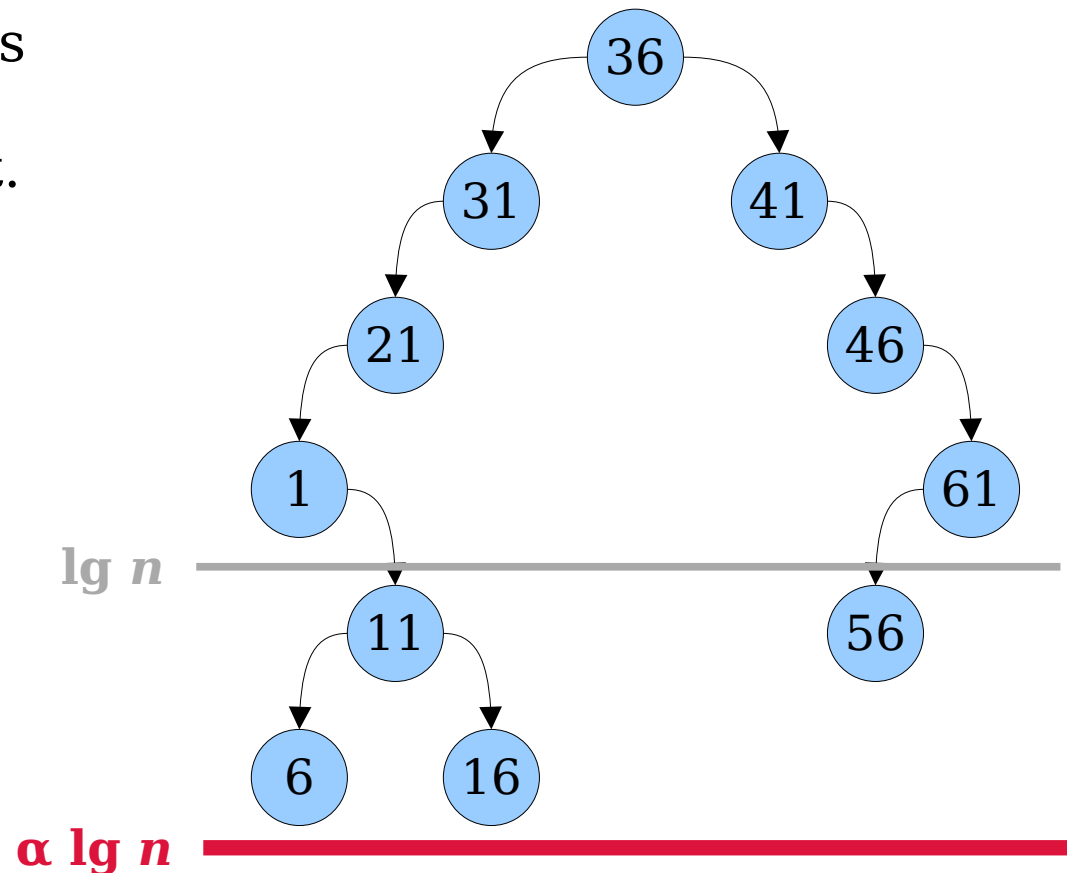
Why Deletions are Different

- In the insert-only case, we can easily detect when the height is violated, and we know which node exceeded the height limit.
- Neither of these are true with deletions.
 - Deleting one node may make an unrelated node height above the threshold.
 - Deleting one node may make multiple unrelated nodes exceed the threshold.



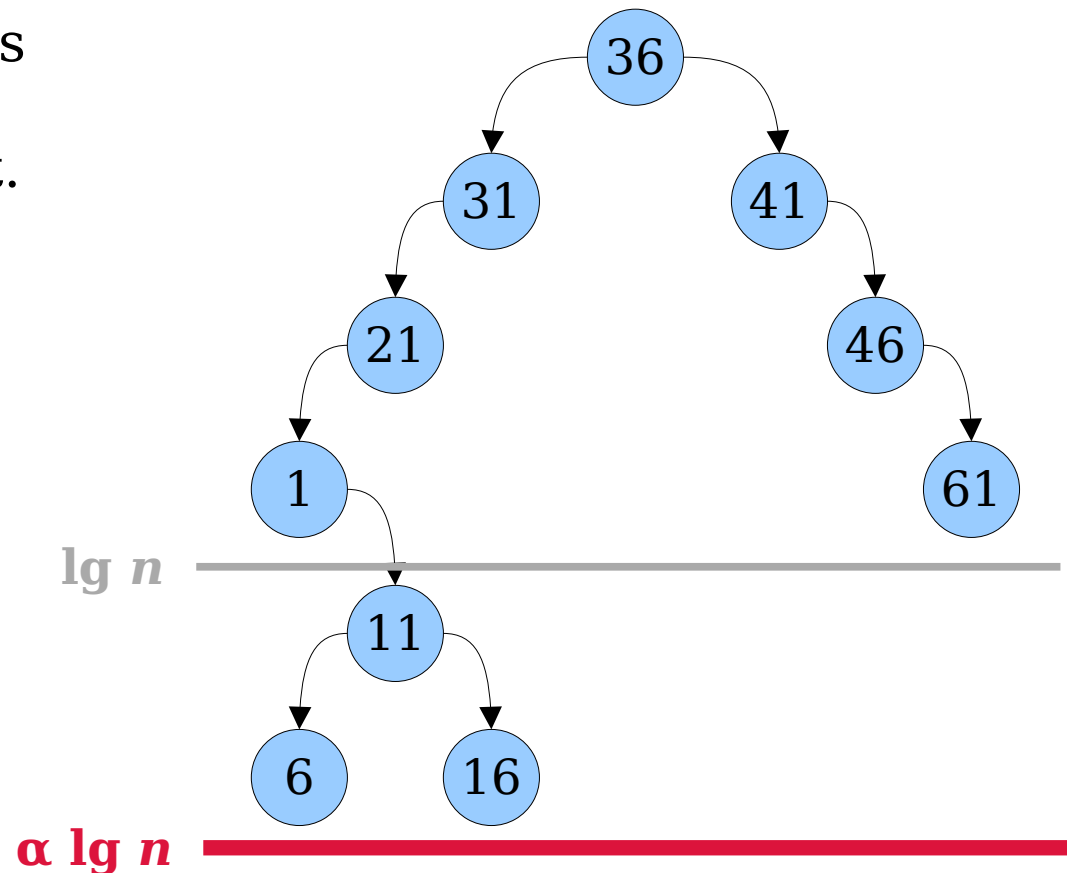
Why Deletions are Different

- In the insert-only case, we can easily detect when the height is violated, and we know which node exceeded the height limit.
- Neither of these are true with deletions.
 - Deleting one node may make an unrelated node height above the threshold.
 - Deleting one node may make multiple unrelated nodes exceed the threshold.



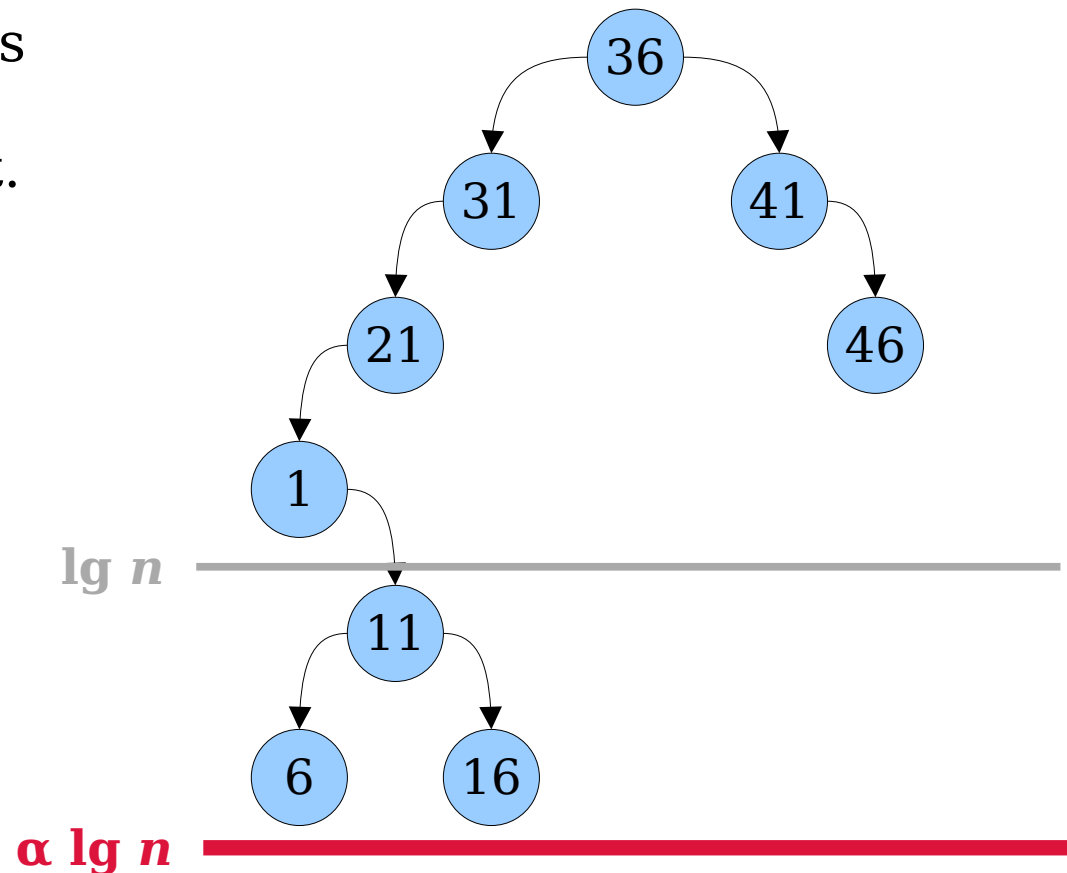
Why Deletions are Different

- In the insert-only case, we can easily detect when the height is violated, and we know which node exceeded the height limit.
- Neither of these are true with deletions.
 - Deleting one node may make an unrelated node height above the threshold.
 - Deleting one node may make multiple unrelated nodes exceed the threshold.



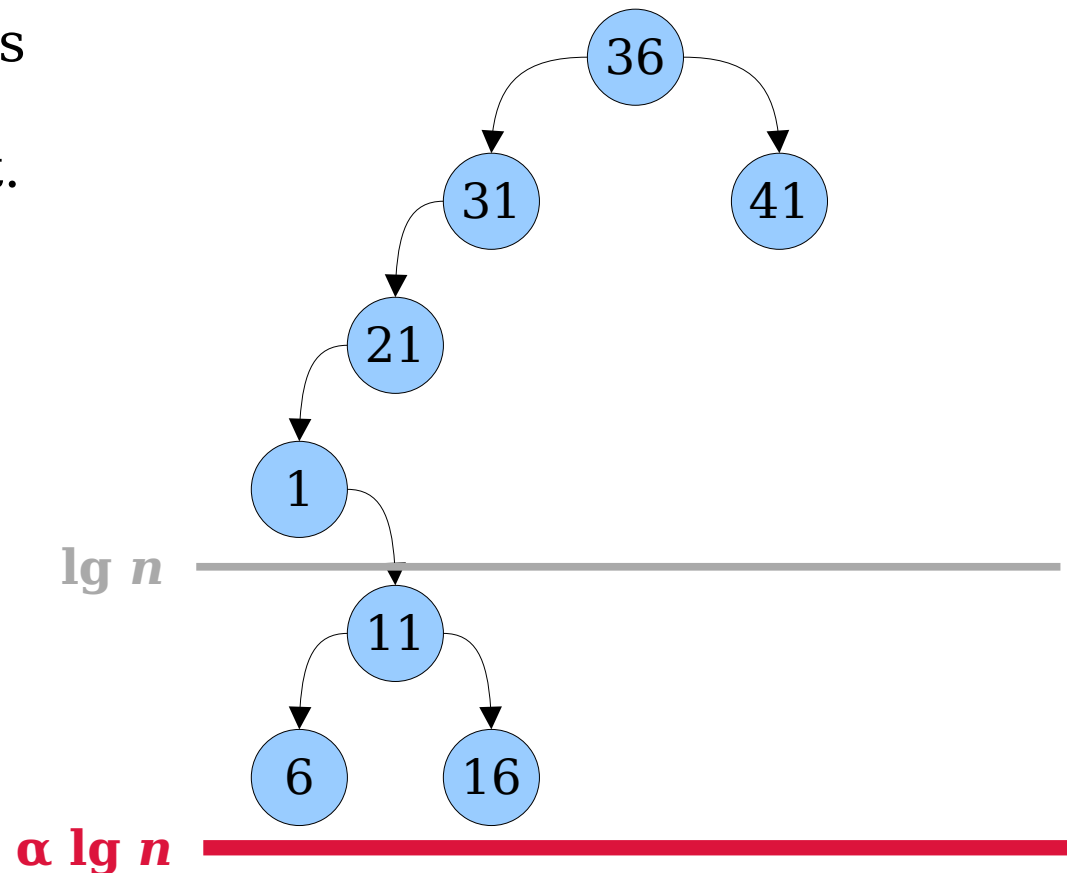
Why Deletions are Different

- In the insert-only case, we can easily detect when the height is violated, and we know which node exceeded the height limit.
- Neither of these are true with deletions.
 - Deleting one node may make an unrelated node height above the threshold.
 - Deleting one node may make multiple unrelated nodes exceed the threshold.



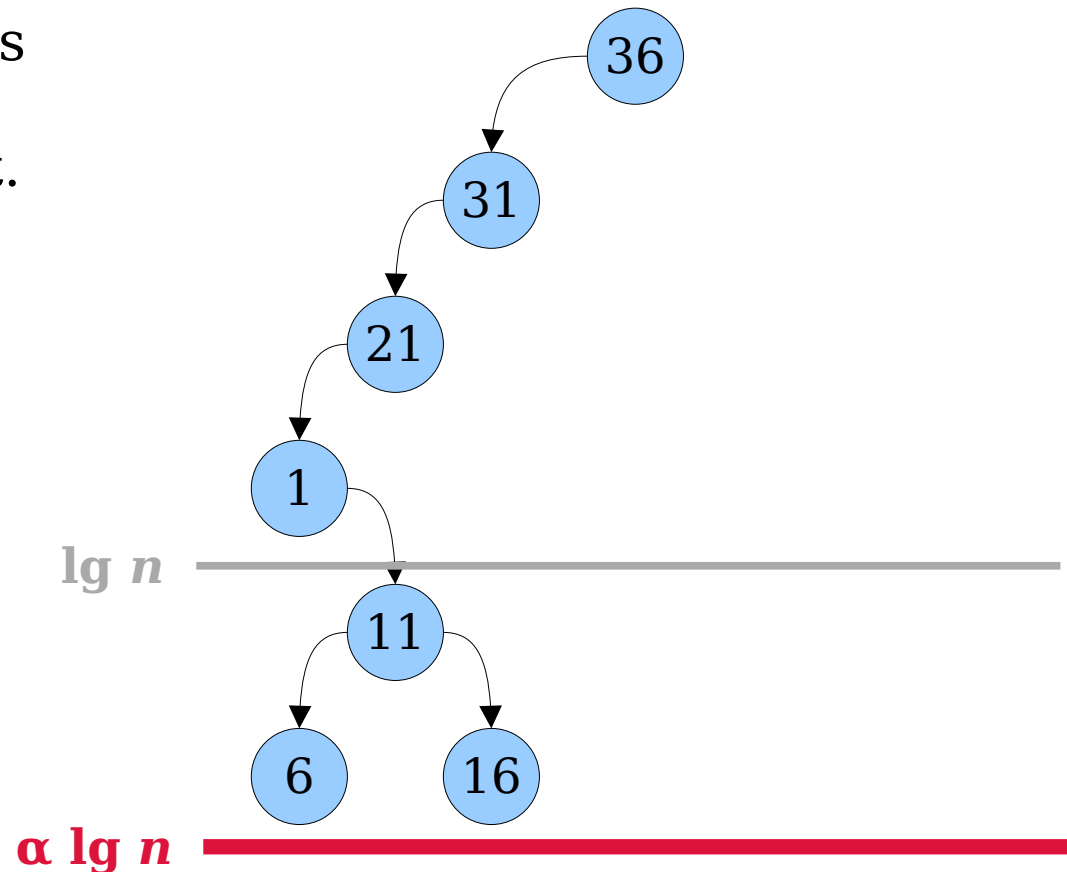
Why Deletions are Different

- In the insert-only case, we can easily detect when the height is violated, and we know which node exceeded the height limit.
- Neither of these are true with deletions.
 - Deleting one node may make an unrelated node height above the threshold.
 - Deleting one node may make multiple unrelated nodes exceed the threshold.



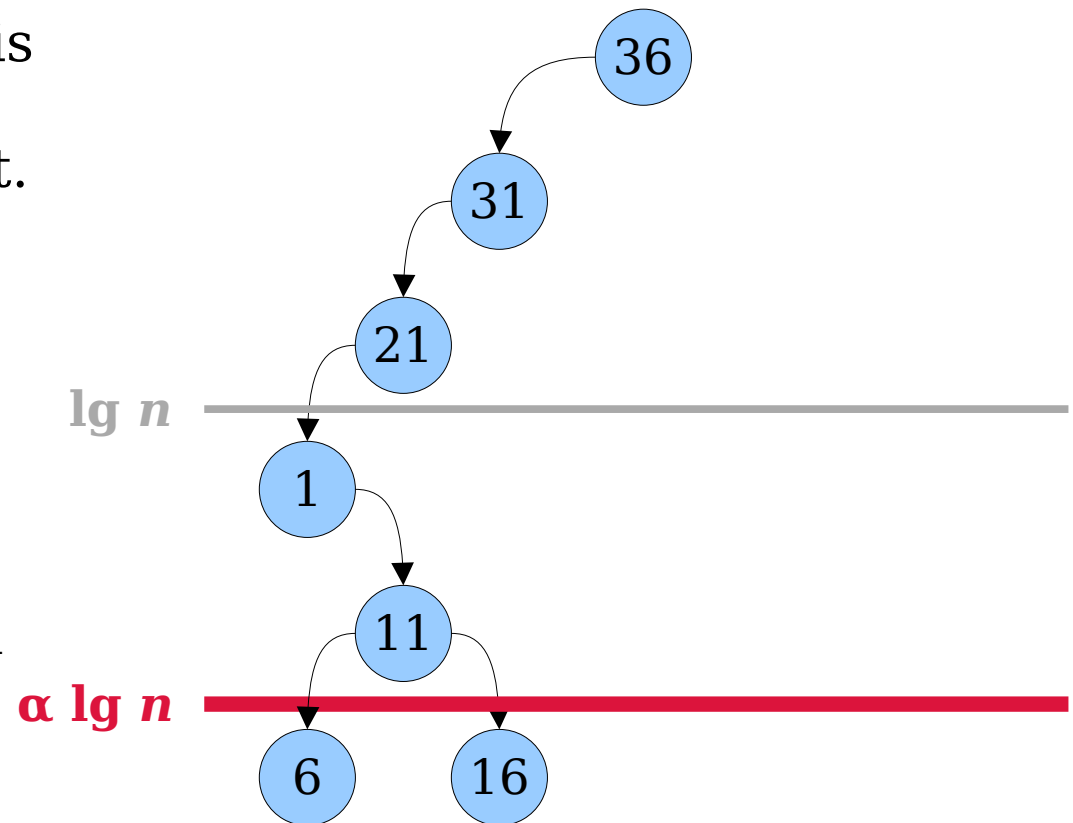
Why Deletions are Different

- In the insert-only case, we can easily detect when the height is violated, and we know which node exceeded the height limit.
- Neither of these are true with deletions.
 - Deleting one node may make an unrelated node height above the threshold.
 - Deleting one node may make multiple unrelated nodes exceed the threshold.



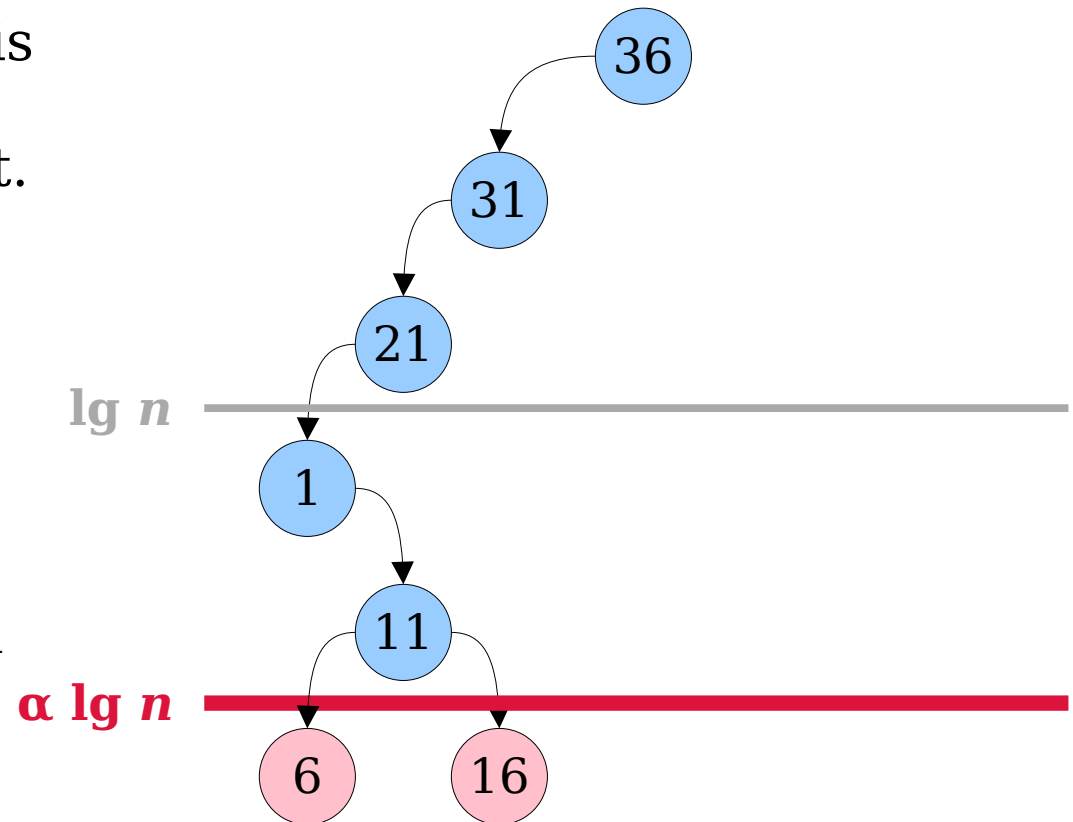
Why Deletions are Different

- In the insert-only case, we can easily detect when the height is violated, and we know which node exceeded the height limit.
- Neither of these are true with deletions.
 - Deleting one node may make an unrelated node height above the threshold.
 - Deleting one node may make multiple unrelated nodes exceed the threshold.



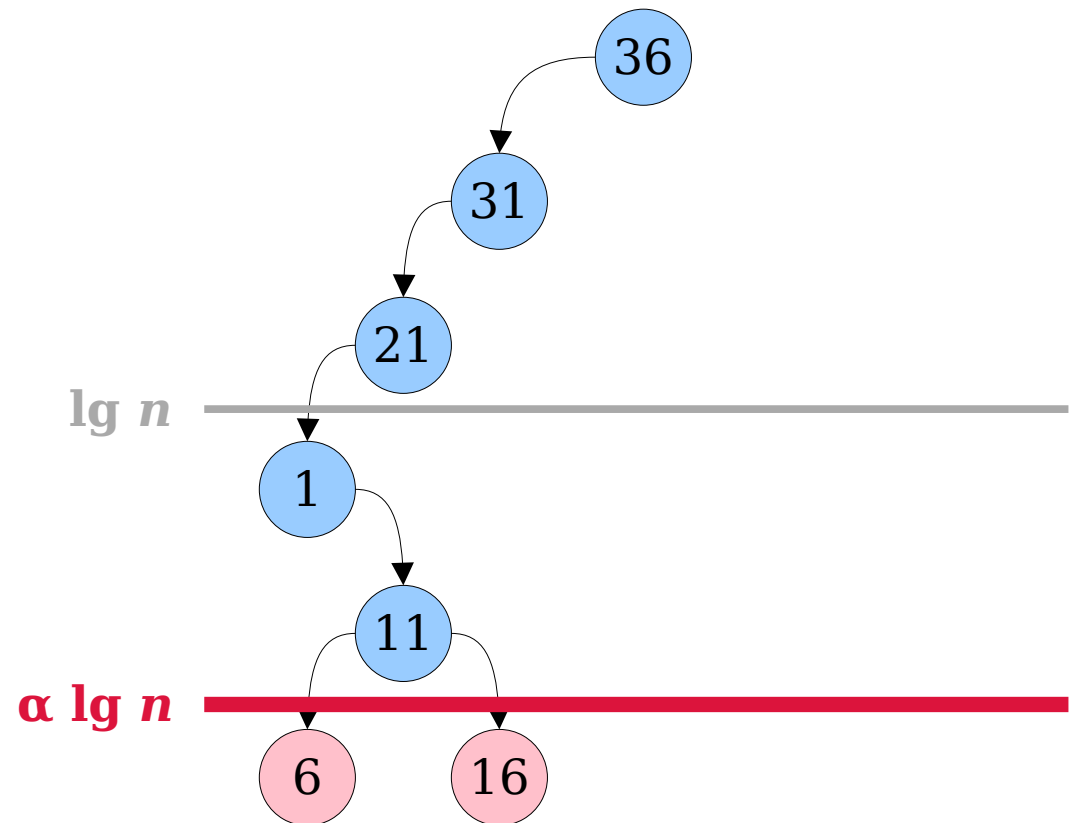
Why Deletions are Different

- In the insert-only case, we can easily detect when the height is violated, and we know which node exceeded the height limit.
- Neither of these are true with deletions.
 - Deleting one node may make an unrelated node height above the threshold.
 - Deleting one node may make multiple unrelated nodes exceed the threshold.
- **Intuition:** Deletions will require some sort of *global* rebuilding of the tree, rather than the *local* rebuilding we saw earlier.



Why Deletions are Different

- As we delete nodes from our BST, the value of $\alpha \lg n$ will decrease, but it does so slowly.
- Leaf nodes will be the first to exceed the $\alpha \lg n$ threshold.
- However, a very large number of nodes need to be deleted before non-leaves cross the threshold.
- Let's quantify this.

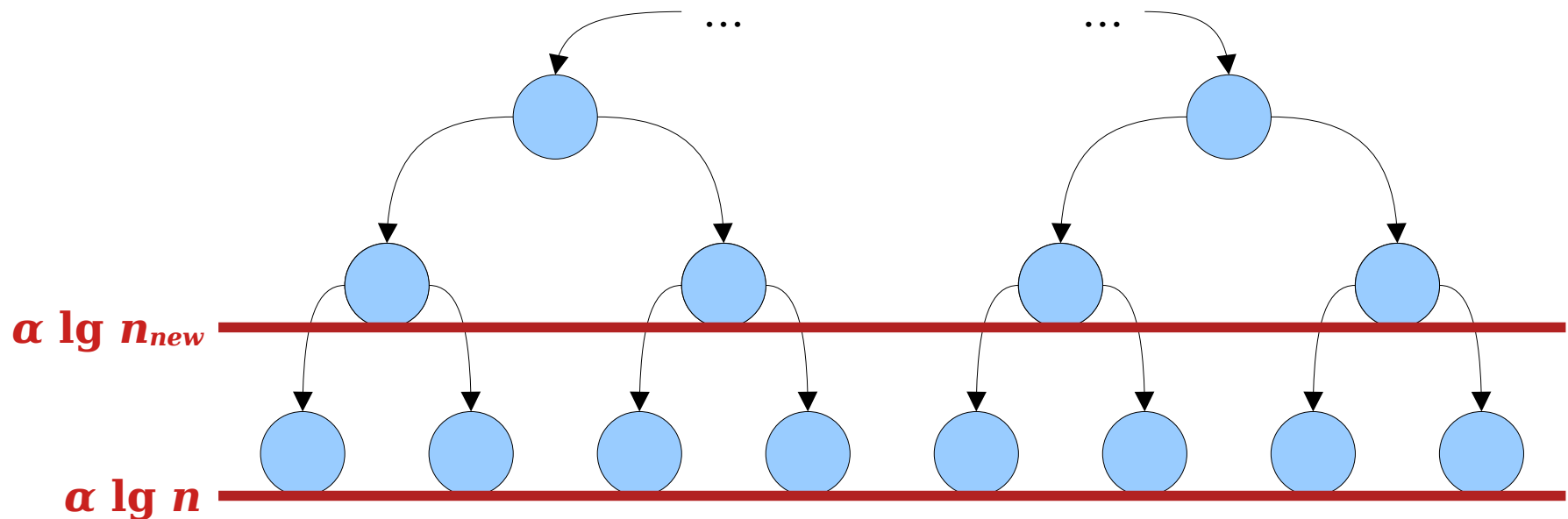


Why Deletions are Different

- Suppose our tree currently has n nodes in it. We'll perform some number of deletions and arrive at a tree with n_{new} nodes.
- At what value of n_{new} is it possible for non-leaf nodes to have a depth greater than $\alpha \lg n_{new}$?

Why Deletions are Different

- Suppose our tree currently has n nodes in it. We'll perform some number of deletions and arrive at a tree with n_{new} nodes.
- At what value of n_{new} is it possible for non-leaf nodes to have a depth greater than $\alpha \lg n_{new}$?



Why Deletions are Different

- Suppose our tree currently has n nodes in it. We'll perform some number of deletions and arrive at a tree with n_{new} nodes.
- At what value of n_{new} is it possible for non-leaf nodes to have a depth greater than $\alpha \lg n_{new}$?
- We need to solve

$$\alpha \lg n_{new} < \alpha \lg n - 1.$$

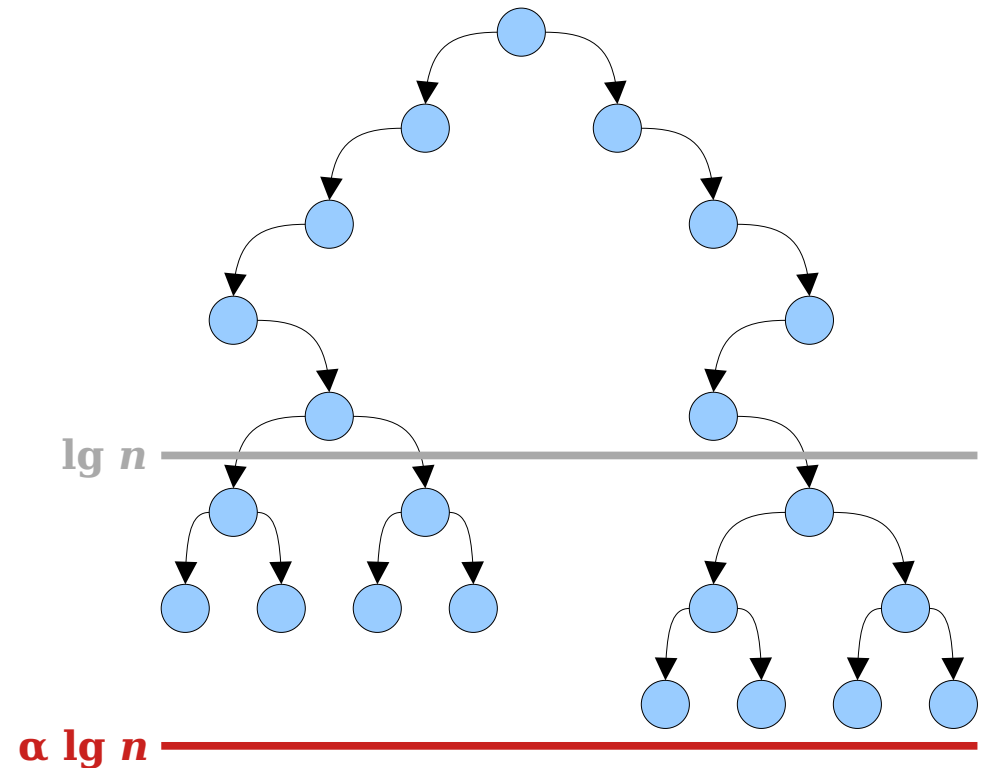
- Rearranging gives us that

$$n_{new} < n \cdot 2^{-1/\alpha}.$$

- Note that $2^{-1/\alpha} \in (1/2, 1)$ for any $\alpha > 1$.
- We need to delete at least a constant fraction (specifically, a $1 - 2^{-1/\alpha}$ fraction) of the nodes before nodes one layer above the bottom could exceed the $\alpha \lg n$ limit.

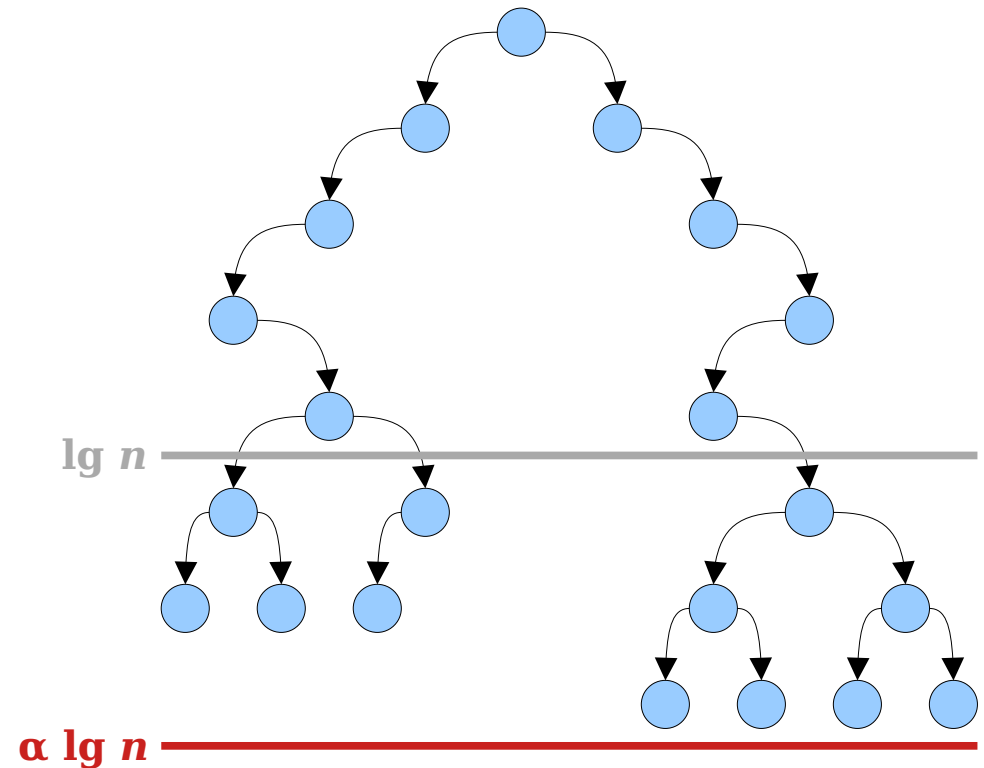
Why Deletions are Different

- **Idea:** Don't worry about rebalancing until we lose a $(1 - 2^{-1/\alpha})$ fraction of the nodes.



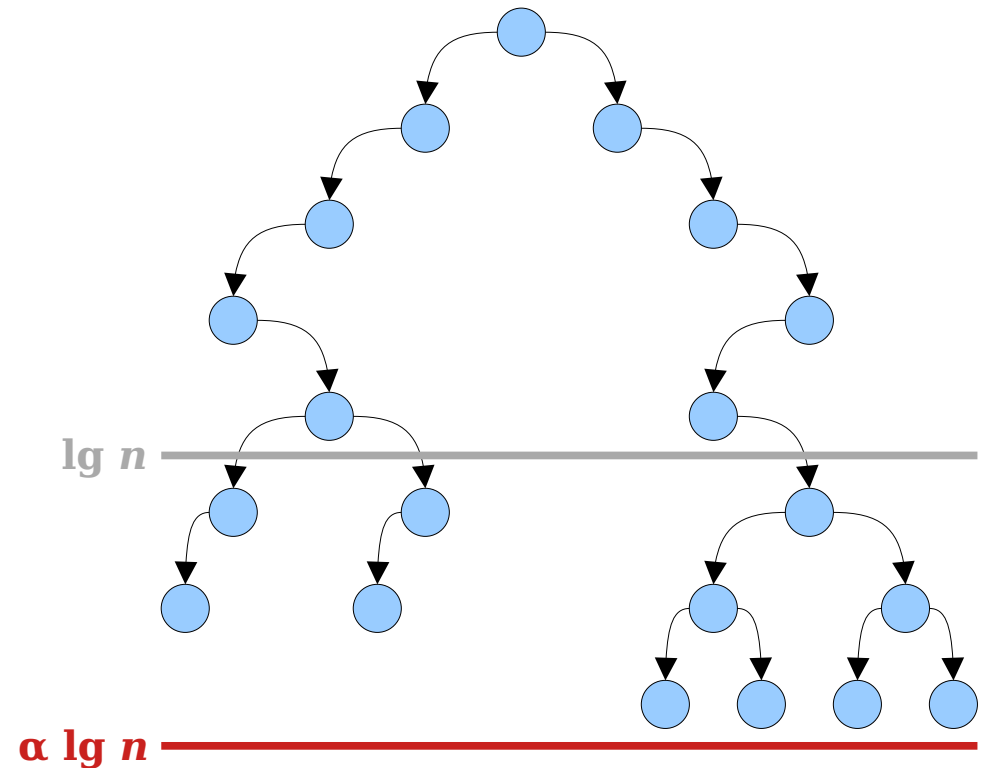
Why Deletions are Different

- **Idea:** Don't worry about rebalancing until we lose a $(1 - 2^{-1/\alpha})$ fraction of the nodes.



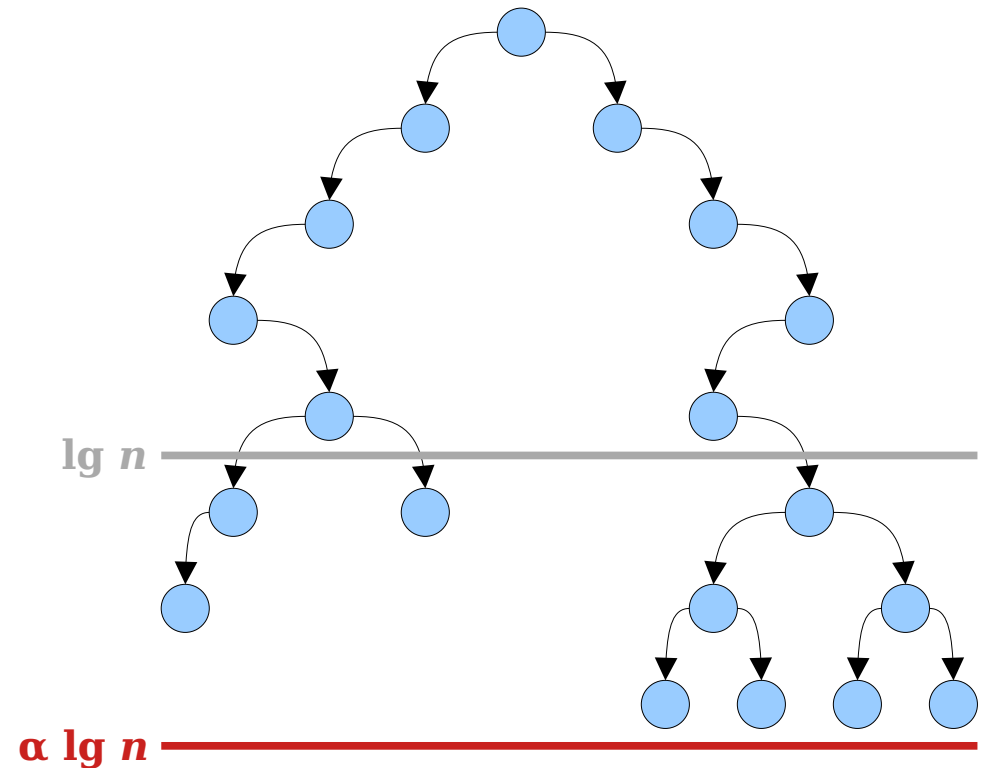
Why Deletions are Different

- **Idea:** Don't worry about rebalancing until we lose a $(1 - 2^{-1/\alpha})$ fraction of the nodes.



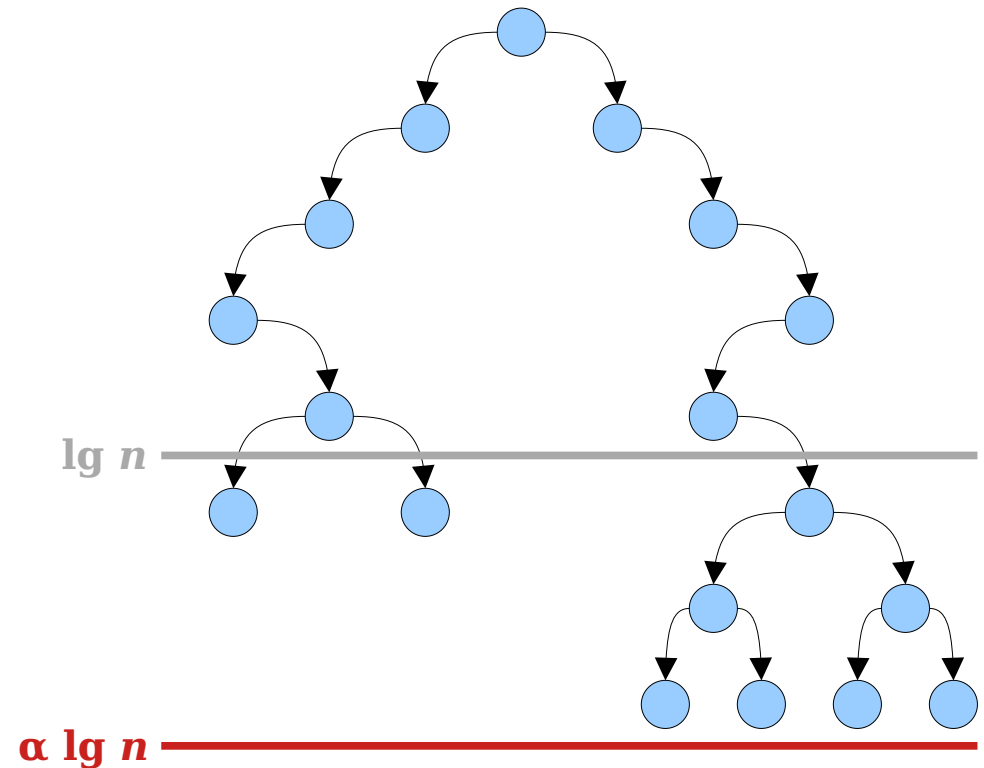
Why Deletions are Different

- **Idea:** Don't worry about rebalancing until we lose a $(1 - 2^{-1/\alpha})$ fraction of the nodes.



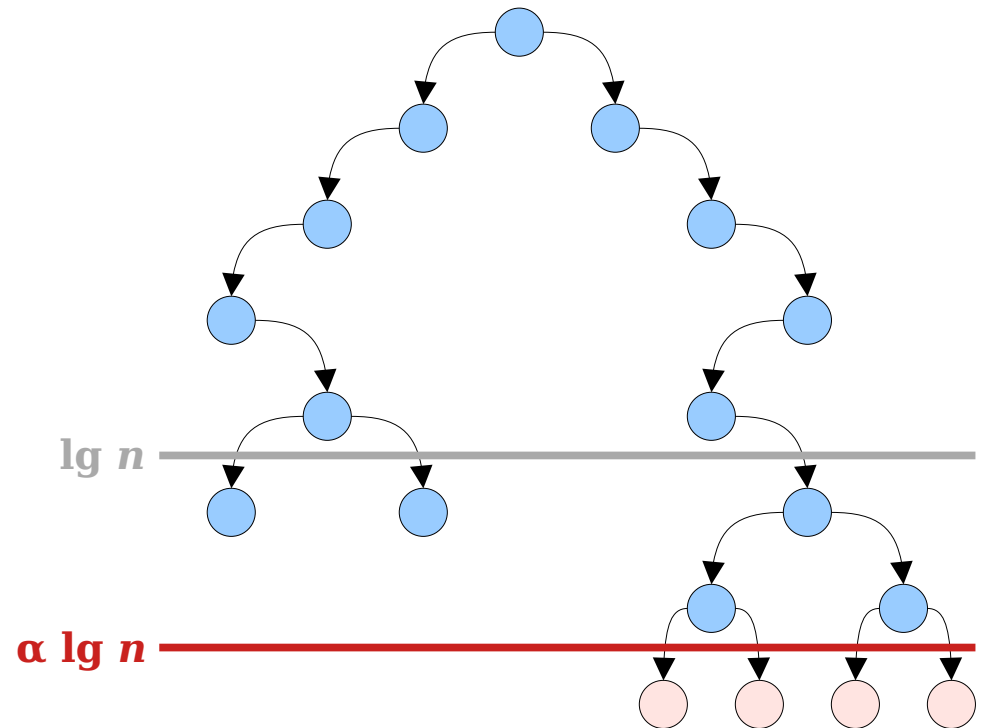
Why Deletions are Different

- **Idea:** Don't worry about rebalancing until we lose a $(1 - 2^{-1/\alpha})$ fraction of the nodes.



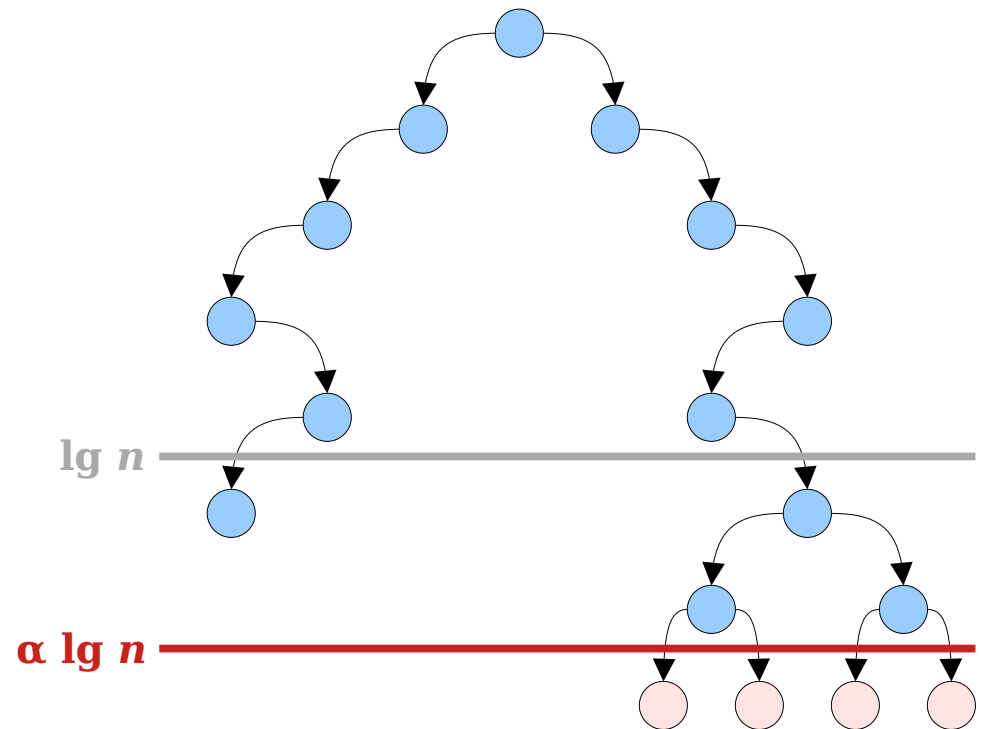
Why Deletions are Different

- **Idea:** Don't worry about rebalancing until we lose a $(1 - 2^{-1/\alpha})$ fraction of the nodes.
- Assuming we lose fewer than this many nodes, all nodes in the tree will be at depth at most $\alpha \lg n + 1$.
 - Focus on any node. Assume there were n_0 nodes at the point when the node was inserted. The node depth is then at most $\alpha \lg n_0$.
 - As long as we haven't lost at least a $(1 - 2^{-1/\alpha})$ fraction of the nodes, the current value of n is such that $\alpha \lg n \geq \alpha \lg n_0 - 1$.
- This still gives us lookups that run in time $O(\log n)$, and insertions still work properly.



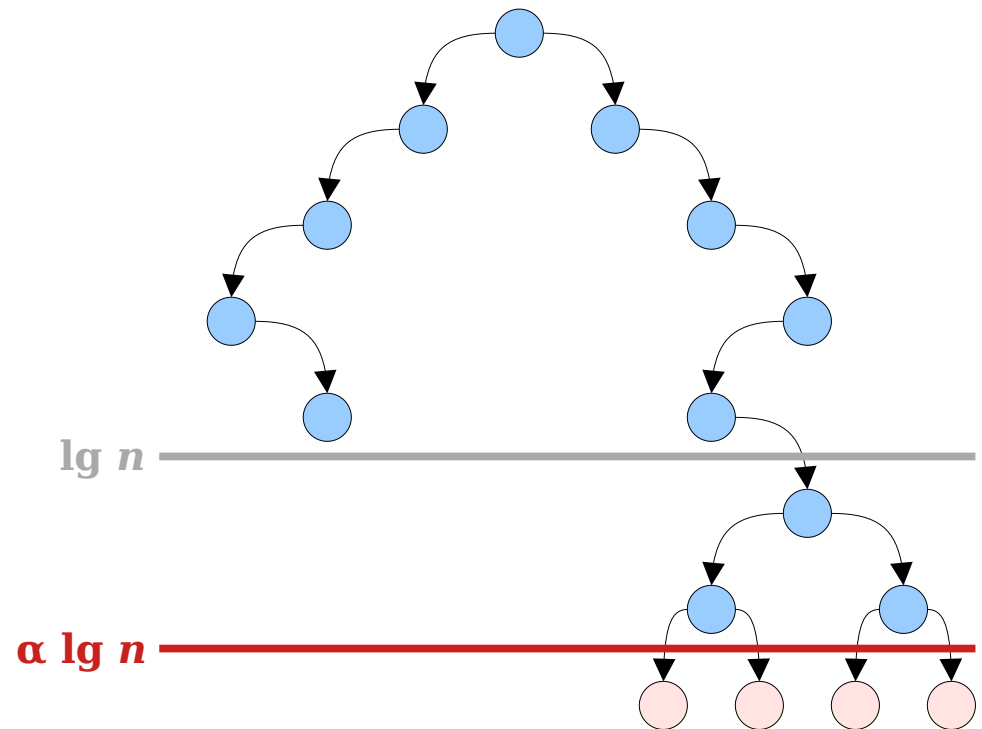
Why Deletions are Different

- **Idea:** Don't worry about rebalancing until we lose a $(1 - 2^{-1/\alpha})$ fraction of the nodes.
- Assuming we lose fewer than this many nodes, all nodes in the tree will be at depth at most $\alpha \lg n + 1$.
 - Focus on any node. Assume there were n_0 nodes at the point when the node was inserted. The node depth is then at most $\alpha \lg n_0$.
 - As long as we haven't lost at least a $(1 - 2^{-1/\alpha})$ fraction of the nodes, the current value of n is such that $\alpha \lg n \geq \alpha \lg n_0 - 1$.
- This still gives us lookups that run in time $O(\log n)$, and insertions still work properly.



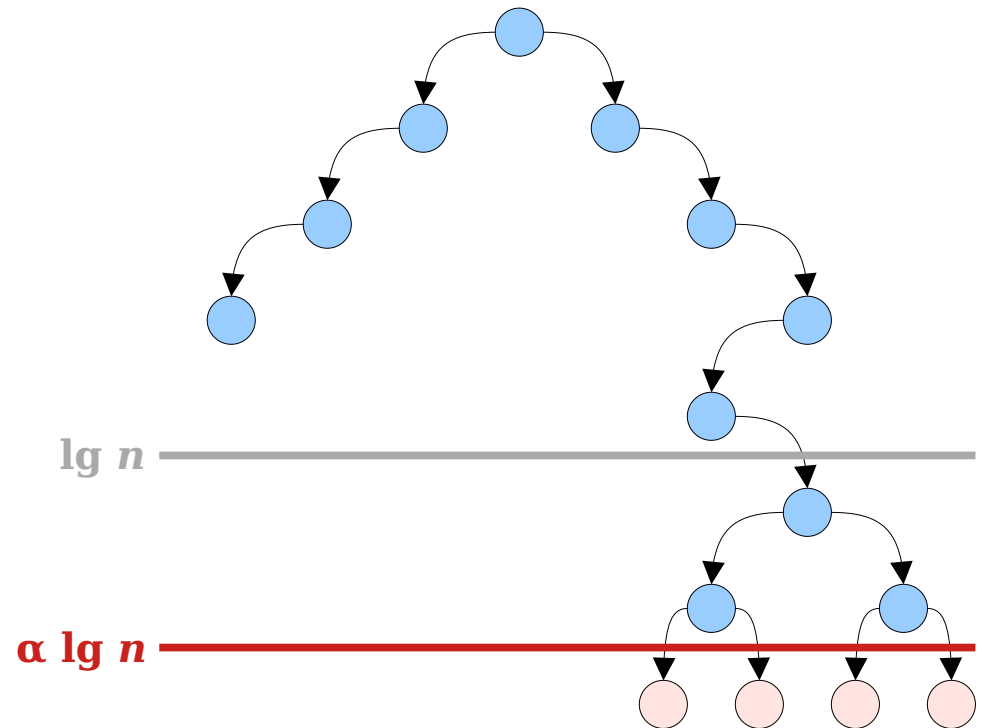
Why Deletions are Different

- **Idea:** Don't worry about rebalancing until we lose a $(1 - 2^{-1/\alpha})$ fraction of the nodes.
- Assuming we lose fewer than this many nodes, all nodes in the tree will be at depth at most $\alpha \lg n + 1$.
 - Focus on any node. Assume there were n_0 nodes at the point when the node was inserted. The node depth is then at most $\alpha \lg n_0$.
 - As long as we haven't lost at least a $(1 - 2^{-1/\alpha})$ fraction of the nodes, the current value of n is such that $\alpha \lg n \geq \alpha \lg n_0 - 1$.
- This still gives us lookups that run in time $O(\log n)$, and insertions still work properly.



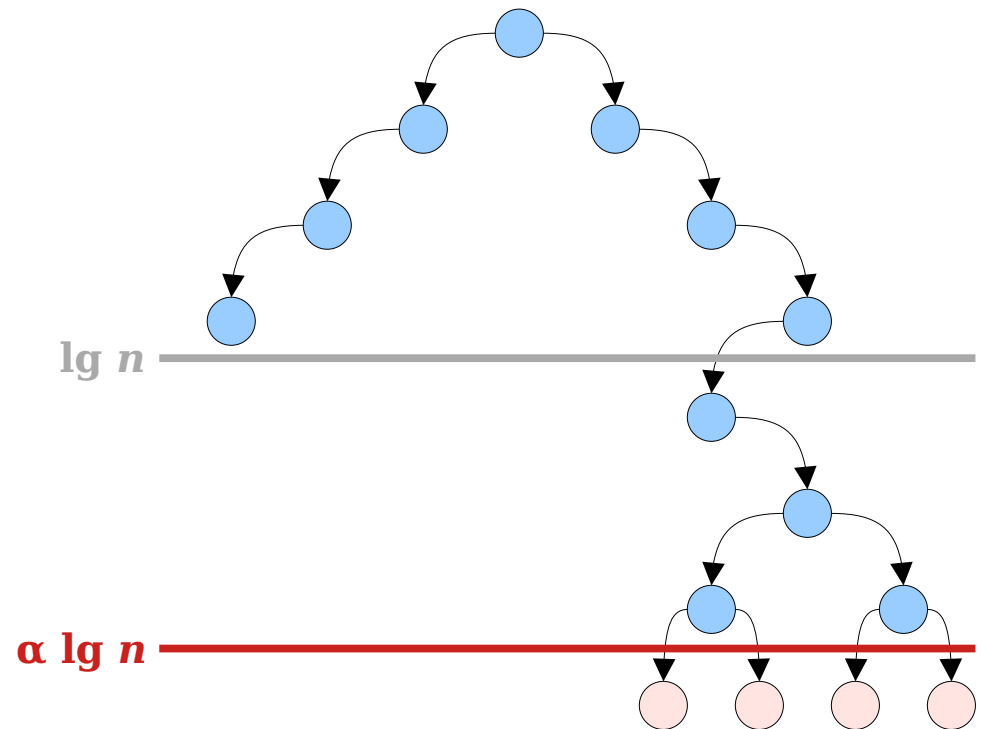
Why Deletions are Different

- **Idea:** Don't worry about rebalancing until we lose a $(1 - 2^{-1/\alpha})$ fraction of the nodes.
- Assuming we lose fewer than this many nodes, all nodes in the tree will be at depth at most $\alpha \lg n + 1$.
 - Focus on any node. Assume there were n_0 nodes at the point when the node was inserted. The node depth is then at most $\alpha \lg n_0$.
 - As long as we haven't lost at least a $(1 - 2^{-1/\alpha})$ fraction of the nodes, the current value of n is such that $\alpha \lg n \geq \alpha \lg n_0 - 1$.
- This still gives us lookups that run in time $O(\log n)$, and insertions still work properly.



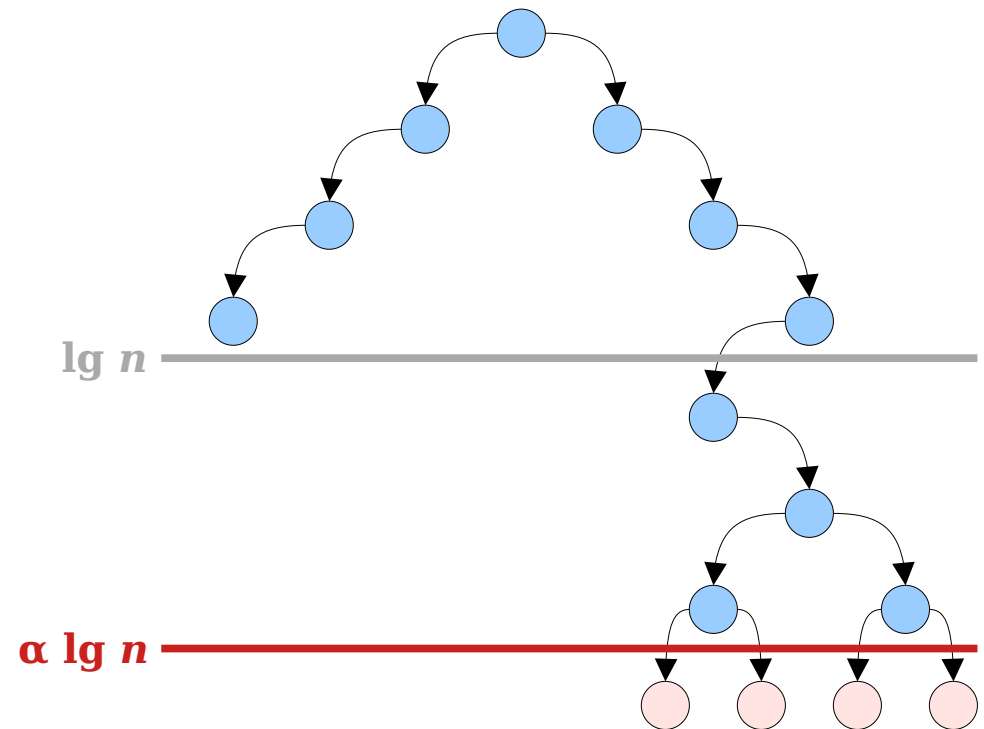
Why Deletions are Different

- **Idea:** Don't worry about rebalancing until we lose a $(1 - 2^{-1/\alpha})$ fraction of the nodes.
- Assuming we lose fewer than this many nodes, all nodes in the tree will be at depth at most $\alpha \lg n + 1$.
 - Focus on any node. Assume there were n_0 nodes at the point when the node was inserted. The node depth is then at most $\alpha \lg n_0$.
 - As long as we haven't lost at least a $(1 - 2^{-1/\alpha})$ fraction of the nodes, the current value of n is such that $\alpha \lg n \geq \alpha \lg n_0 - 1$.
- This still gives us lookups that run in time $O(\log n)$, and insertions still work properly.



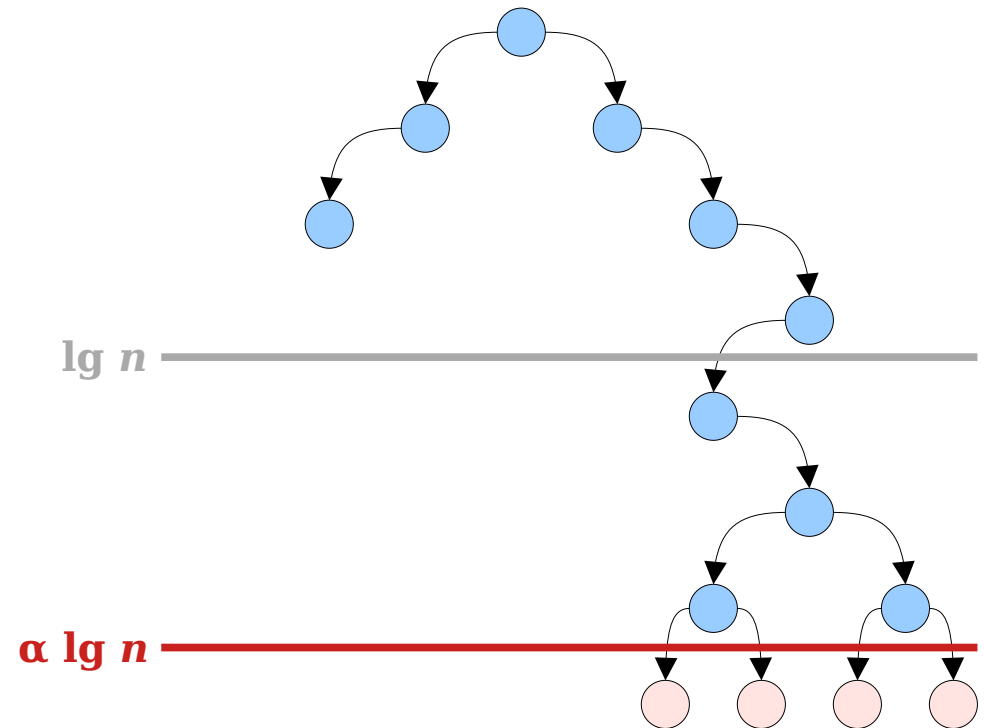
Why Deletions are Different

- Once we've lost a $(1 - 2^{-1/\alpha})$ fraction of the nodes, we need to worry about rebalancing the tree.



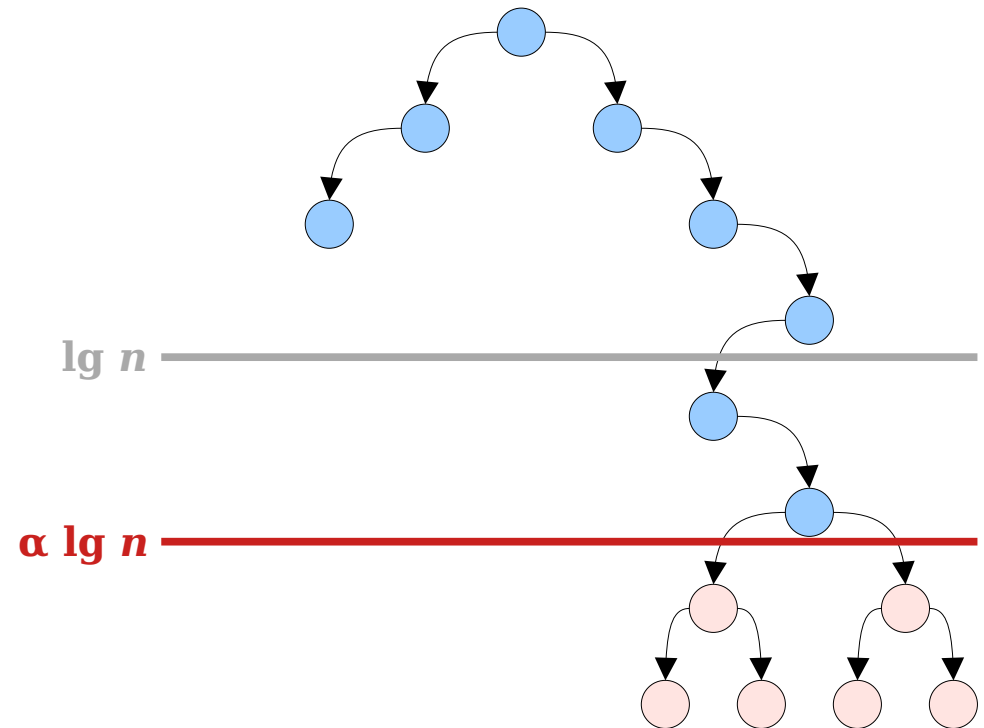
Why Deletions are Different

- Once we've lost a $(1 - 2^{-1/\alpha})$ fraction of the nodes, we need to worry about rebalancing the tree.



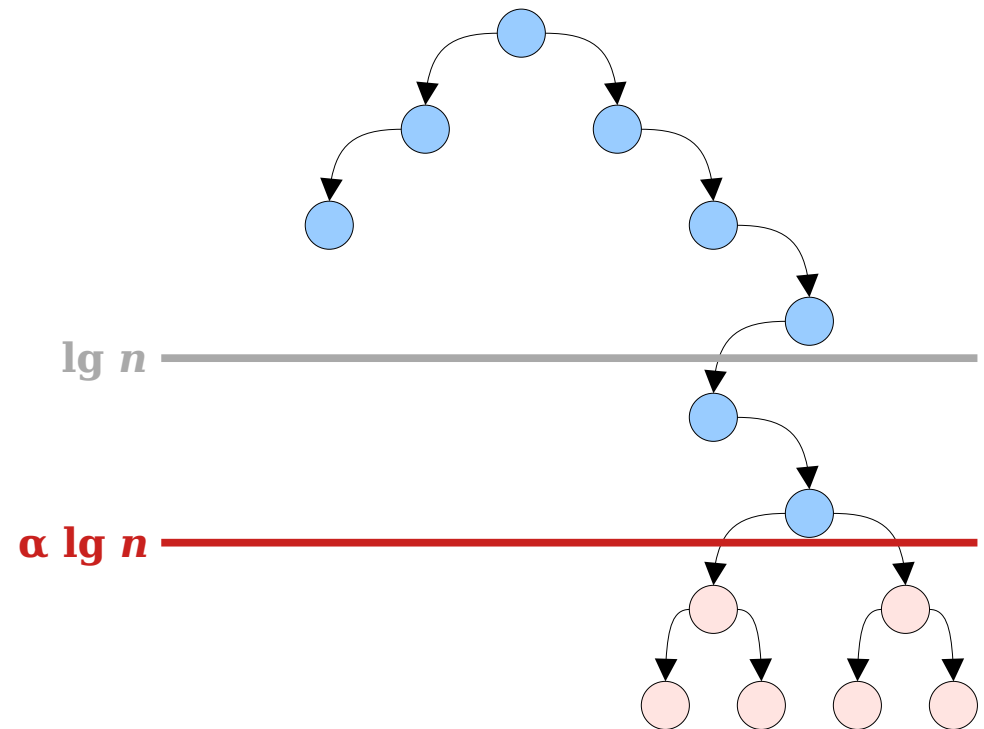
Why Deletions are Different

- Once we've lost a $(1 - 2^{-1/\alpha})$ fraction of the nodes, we need to worry about rebalancing the tree.



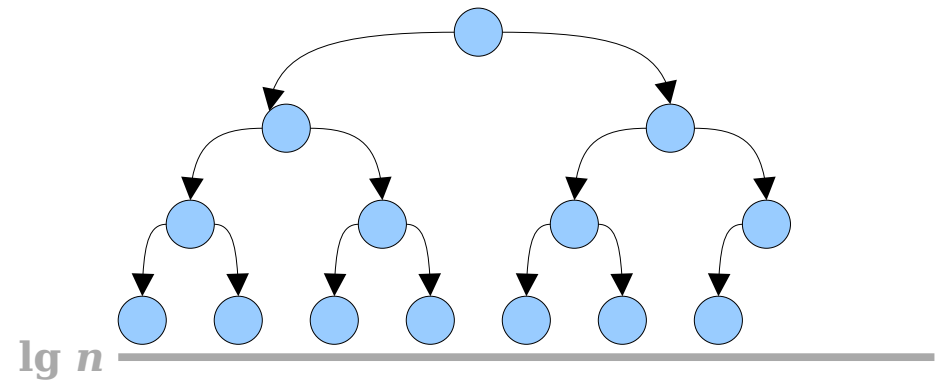
Why Deletions are Different

- Once we've lost a $(1 - 2^{-1/\alpha})$ fraction of the nodes, we need to worry about rebalancing the tree.
- We won't know much about the tree shape.
 - It could have a large number of deep nodes.
 - It could be perfectly balanced.
- **Idea:** Don't try to analyze the tree. Just rebuild the entire tree from scratch.



Why Deletions are Different

- Once we've lost a $(1 - 2^{-1/\alpha})$ fraction of the nodes, we need to worry about rebalancing the tree.
- We won't know much about the tree shape.
 - It could have a large number of deep nodes.
 - It could be perfectly balanced.
- **Idea:** Don't try to analyze the tree. Just rebuild the entire tree from scratch.



$\alpha \lg n$

Scapegoat Tree Deletions

- Here's how this approach will work.
 - Keep track of the maximum number of nodes the tree has had since it was last globally rebuilt. (Call this n_{max}).
 - If the number of nodes drops to a $n_{max} \cdot 2^{-1/\alpha}$, globally rebuild the tree as a perfectly balanced tree, then reset n_{max} to the current tree size.
- Although rebuilding the tree is an expensive operation, intuitively we expect to be able to “backcharge” the work to the lazy delete operations that triggered it.

Scapegoat Tree Deletions

- Our goal now is to work out the amortized cost of doing global rebuilds on deletions.
- **Recall:** Our current potential function is

$$\Phi = \sum_v \phi(v),$$

which we chose to make the cost of local rebuilds on insertions amortize away.

- We need to adjust this potential function to account for the fact that deleted nodes slowly lead us to do a global rebuild of the whole tree.
- **Idea:** Change our potential to

$$\Phi = D + \sum_v \phi(v),$$

where D is the number of deletions that have been performed since we last did a global rebuild.

Scapegoat Tree Deletions

- What is the amortized cost of a deletion when we don't trigger a global rebuild?
- Actual cost: $O(\log n)$, since the tree height is at most $\alpha \lg n + 1$.
- Change in potential (recall that $\Phi = D + \sum_v \phi(v)$):
 - D increases by one, since we've performed a deletion.
 - $\phi(v)$ changes by at most two for each node on the access path of the removed node, and there are $O(\log n)$ such nodes.
 - Net change: $O(\log n)$.
- Amortized cost:

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

Scapegoat Tree Deletions

- What is the amortized cost of a deletion when we *do* trigger a global rebuild?
- We picked

$$\Phi = D + \sum_v \phi'(v).$$

- After the rebuild, we have $\sum_v \phi'(v) = 0$. Therefore, there is an unknown but nonpositive change in potential for this term.
- How much does D change?
 - At the point where we start the rebuild, we have $n = n_{\max} \cdot 2^{-1/\alpha}$ nodes left in the tree.
 - This means that $D \geq n_{\max} \cdot (1 - 2^{-1/\alpha})$.
 - Rewriting in terms of n , this means $D \geq n \cdot (2^{1/\alpha} - 1) = \Omega(n)$.
 - Since after this step we drop D to zero, we have $\Delta D \leq -\Omega(n)$.
- Overall, we have $\Delta\Phi \leq -\Omega(n)$.

Scapegoat Tree Deletions

- Actual cost of the deletion:
 - $O(\log n)$ for the actual deletion logic.
 - $O(n)$ to rebuild the tree.
- Amortized cost:
$$O(\log n) + O(n) - k \cdot \Omega(n).$$
- As before, we can tune k based on the hidden constant factors in the O and Ω terms to make them cancel out and leave behind an amortized cost of **$O(\log n)$** .

The Final Scorecard

- Here's the final scorecard for our scapegoat tree.
- It matches the time bounds we'd expect of a red/black tree, in an amortized sense, with a *dramatically* simpler implementation.
- This gives a sense of just how useful a technique amortization can be!

Scapegoat Tree

Lookup: $O(\log n)$

Insert: $O(\log n)^*$

Delete: $O(\log n)^*$

* *amortized*

Further Exploration

- I haven't seen much work done into building an optimized scapegoat tree implementation. How fast can you make this idea work? Is it competitive with a red/black tree?
- We've treated α as a constant. What if you allow it to vary based on the workflow (say, decreasing it as more lookups happen and increasing it as more deletions/insertions happen)? A past CS166 project team looked into this in 2014, and I'm curious to see it on modern hardware.
- Are there other, less aggressive strategies besides rebuilding the scapegoat subtree that can be used to restore balance?
- Are there other ways of picking a scapegoat node that work better in practice? For example, could you pick a scapegoat higher up in the tree that would do a better job rebalancing things?
- What is the practical time/space tradeoff between the two approaches for calculating $\text{size}(v)$ when finding a scapegoat?
- The version of scapegoat trees described here is a hybrid between two approaches: the original developed by Galperin and Rivest and a simplification by Jeff Erickson. The Galperin/Rivest version has tighter structural constraints, while Erickson's version uses a different deletion strategy. Can you remix these ideas in other ways?
- Because there are no rotations, it should be way easier to augment a scapegoat tree than it is to augment a red/black tree. Can you find a weaker set of requirements for augmenting a BST if you assume the tree you're augmenting is a scapegoat tree?

Next Time

- ***Tournament Heaps***
 - A simple and fast priority queue.
- ***Lazy Tournament Heaps***
 - An asymptotically faster priority queue.