

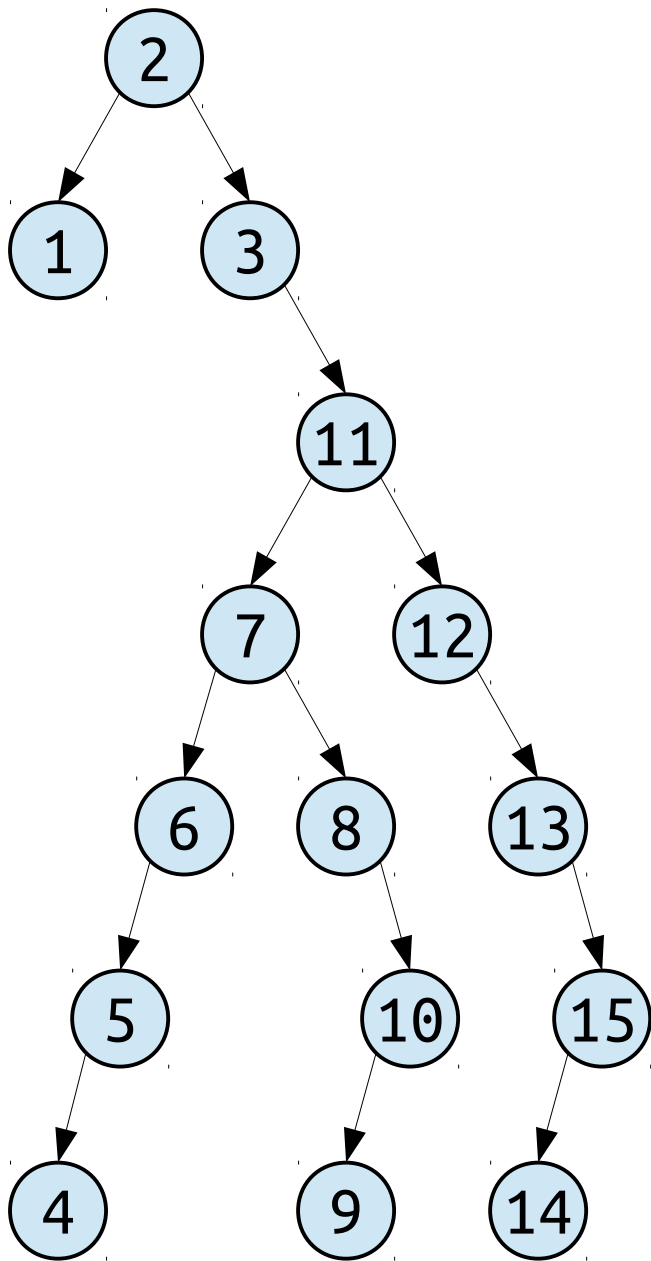
Splay Trees

Outline for Today

- ***Beyond Worst-Case Efficiency***
 - When $O(\log n)$ isn't enough.
- ***Self-Adjusting Search Trees***
 - Trees that improve themselves.
- ***Splay Trees***
 - One tree to rule them all?
- ***Dynamic Optimality***
 - Is there a single best BST structure?

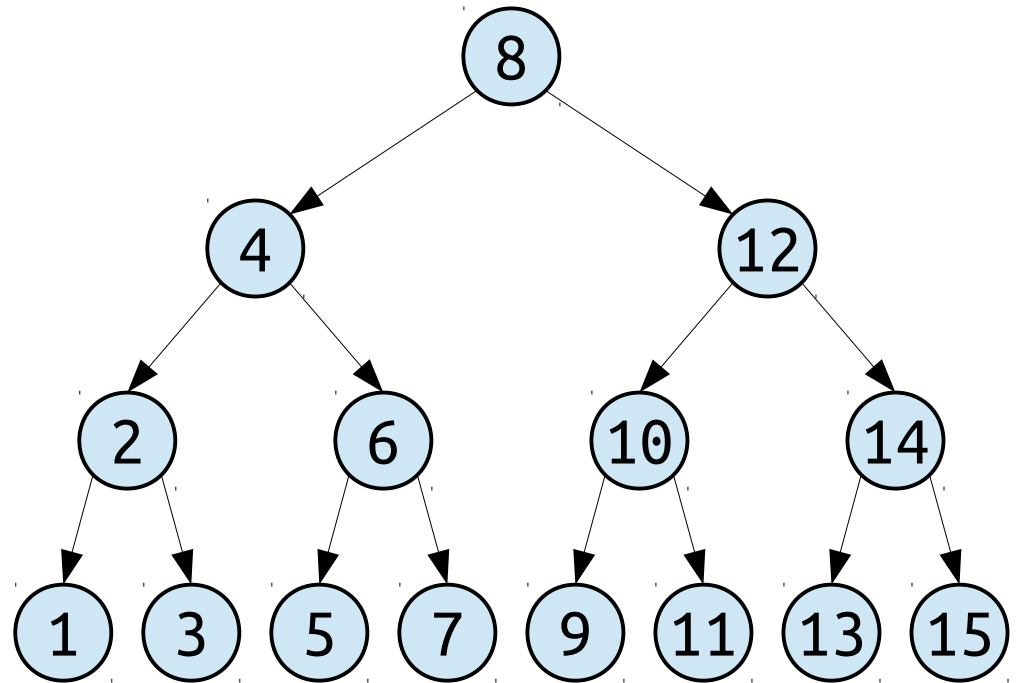
Beyond Worst-Case Efficiency

Key Idea: The guarantees we want from a data structure depend on our model of how that data structure will be used.

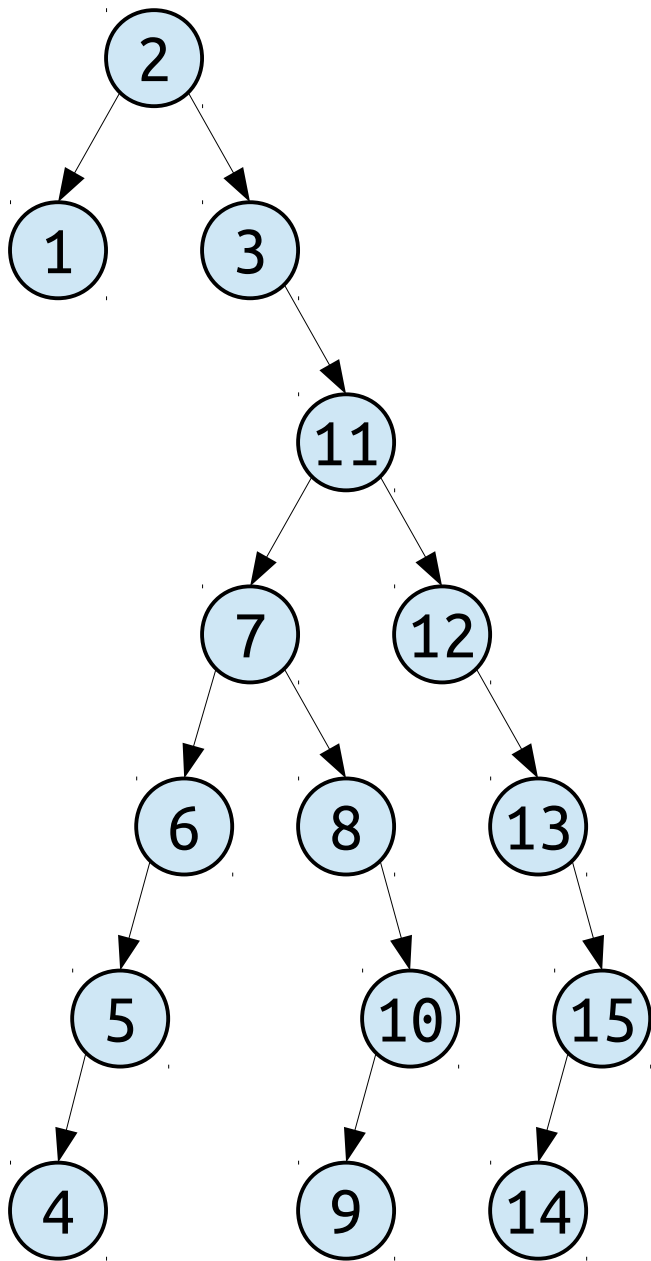


Claim: The worst-case lookup cost of a lookup in *any* BST with n nodes is at least $\Omega(\log n)$.

Proof Idea: Every tree with n nodes has height $\Omega(\log n)$. Pick the deepest node in the tree.

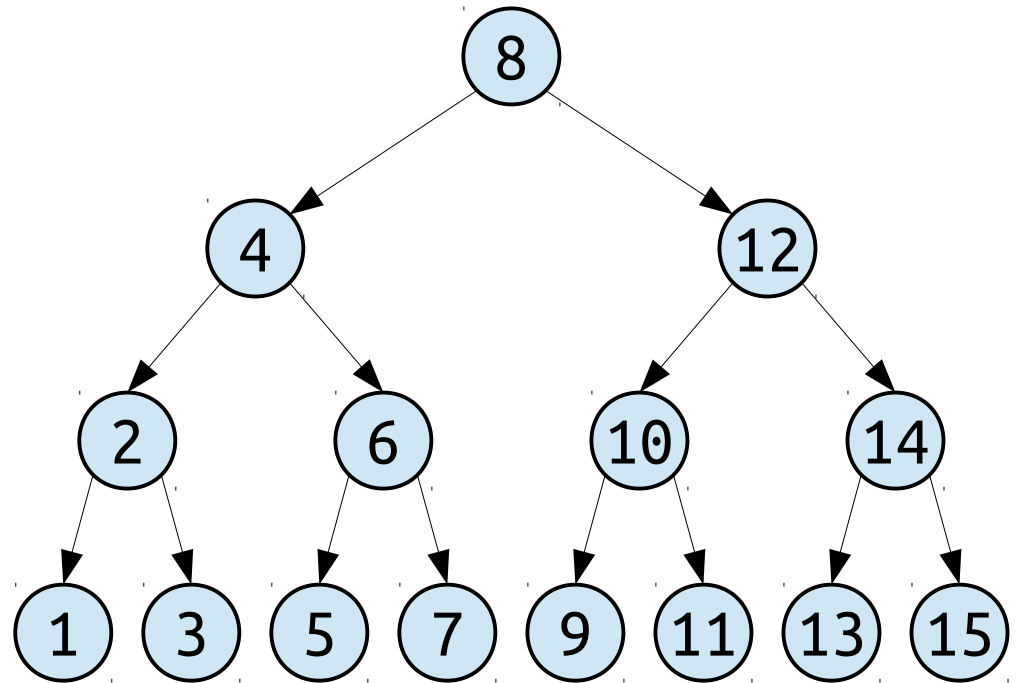


Model 1: Queries are chosen maliciously.



A binary search tree satisfies the **balance property** if the (amortized) cost of any lookup in that tree is $O(\log n)$.

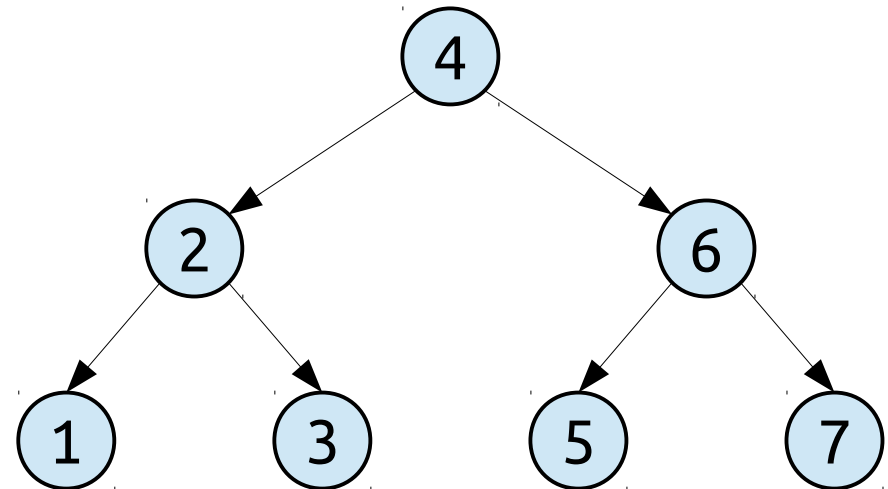
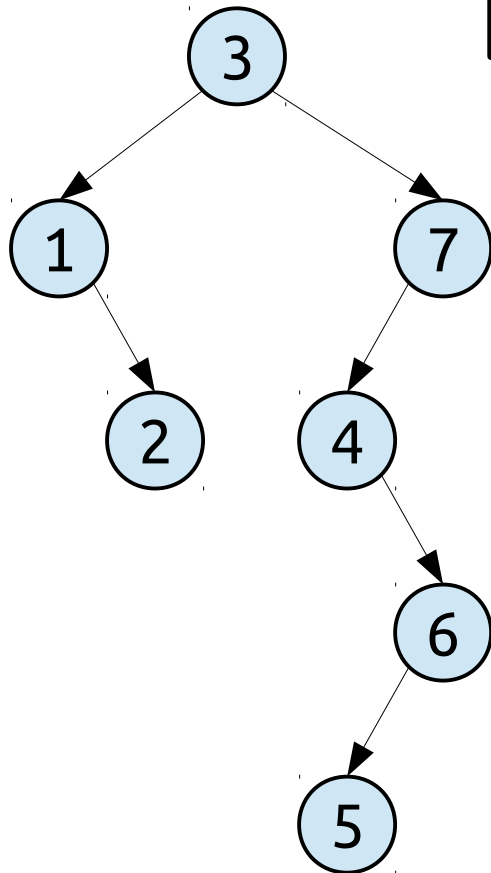
Any BST with this property is optimal from a *worst-case* perspective.



Model 1: Queries are chosen maliciously.

Access Probabilities

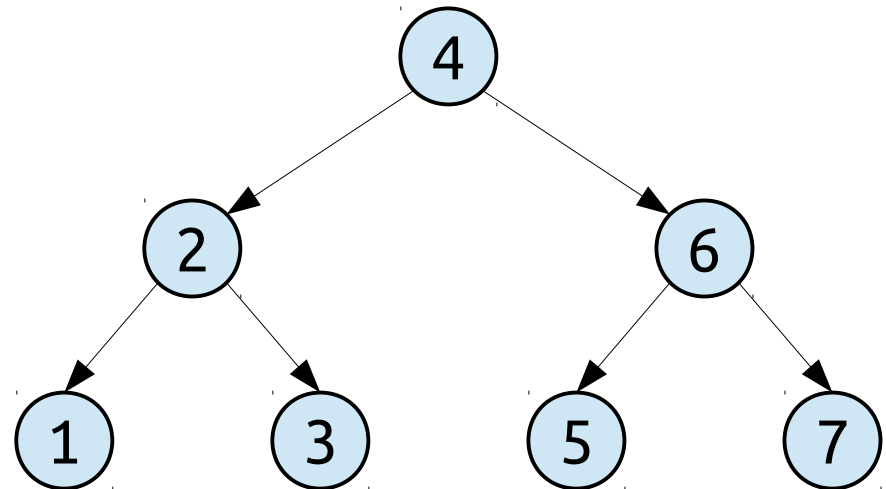
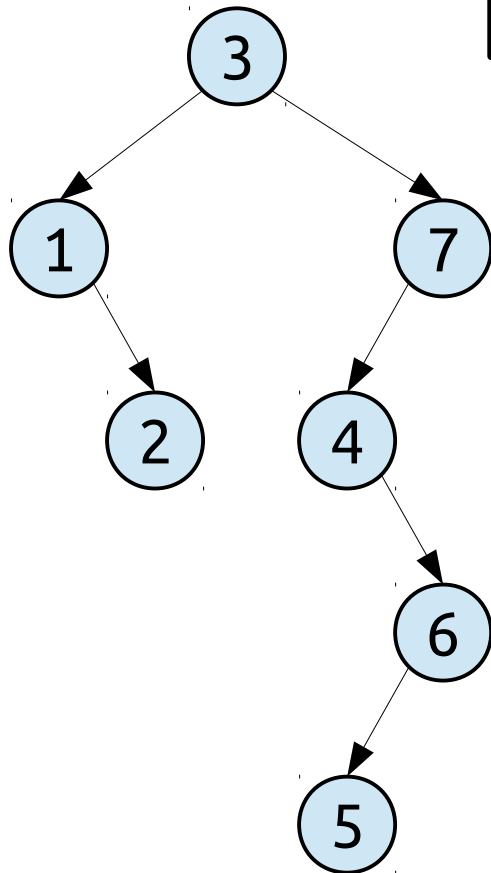
①	②	③	④	⑤	⑥	⑦
20%	10%	40%	8%	1%	1%	20%



Model 2: Queries are sampled from a fixed, known probability distribution.

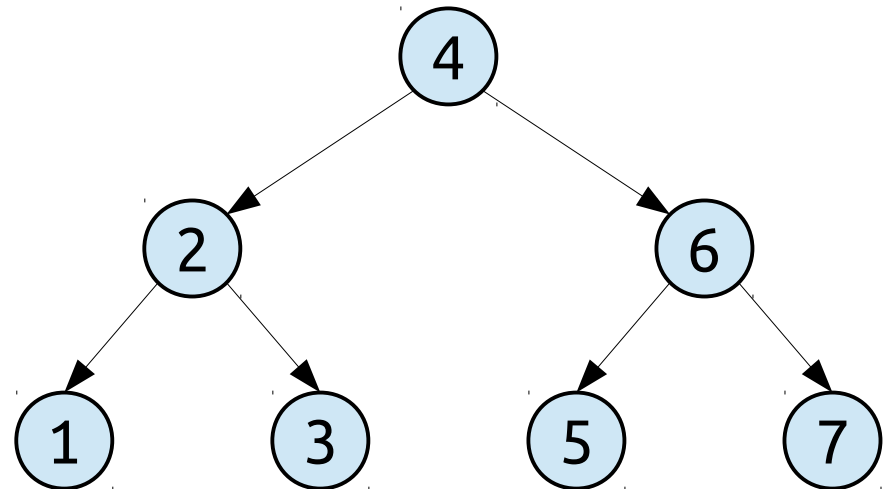
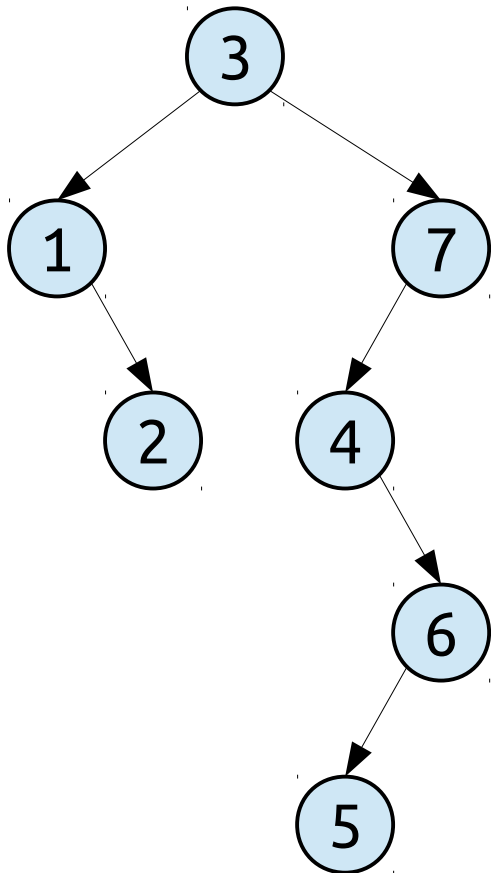
Access Probabilities

①	②	③	④	⑤	⑥	⑦
14%	15%	14%	14%	14%	15%	14%



Model 2: Queries are sampled from a fixed, known probability distribution.

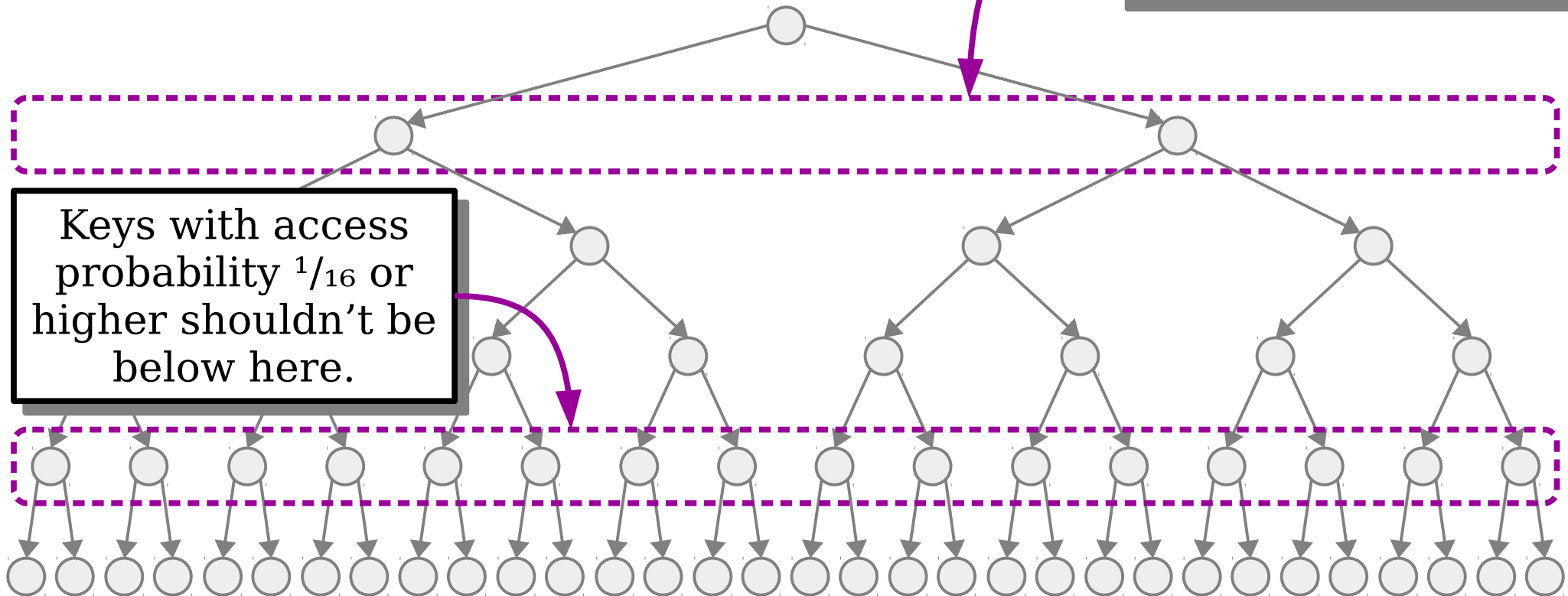
How do we know when we have a BST that's optimal with respect to *expected* lookup costs?



Model 2: Queries are sampled from a fixed, known probability distribution.

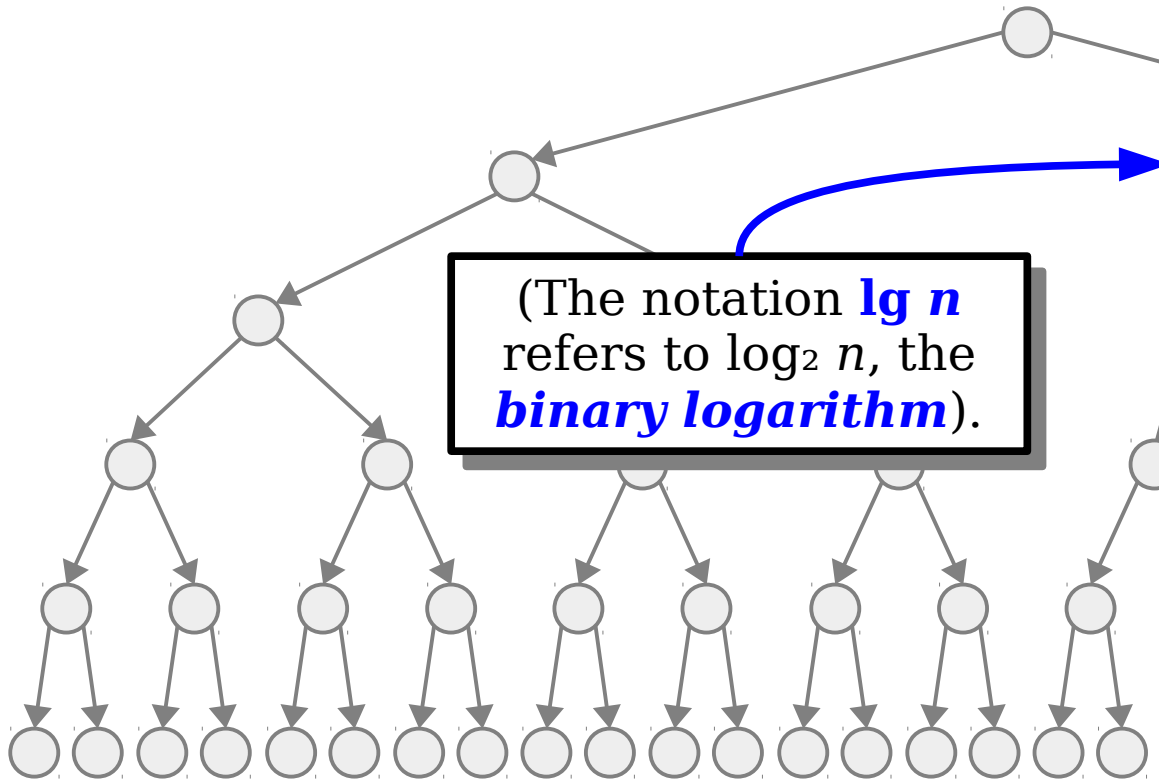
Intuition: Place high probability elements high in the tree.

Keys with access probability $\frac{1}{2}$ or higher shouldn't go below here.



Model 2: Queries are sampled from a fixed, known probability distribution.

Intuition: Place high probability elements high in the tree.



(The notation **lg** n refers to $\log_2 n$, the **binary logarithm**).

A node should go in layer k or above if its access probability is at least 2^{-k} .

Equivalently, a node with access probability p should go in layer $\lg(1/p)$ or higher.

Note that $\lg(1/p) = -\lg p$.

Expected cost of a lookup would then be

$$\sum_{i=1}^n -p_i \lg p_i.$$

Model 2: Queries are sampled from a fixed, known probability distribution.



Theory Time!

Model 2: Queries are sampled from a fixed, known probability distribution.

Consider a discrete probability distribution with elements x_1, \dots, x_n , where element x_i has access probability p_i .

The **Shannon entropy** of this probability distribution, denoted H_p (or just H , where p is implicit) is the quantity

$$H_p = \sum_{i=1}^n -p_i \lg p_i.$$

If all elements have equal access probability ($p_i = 1/n$):

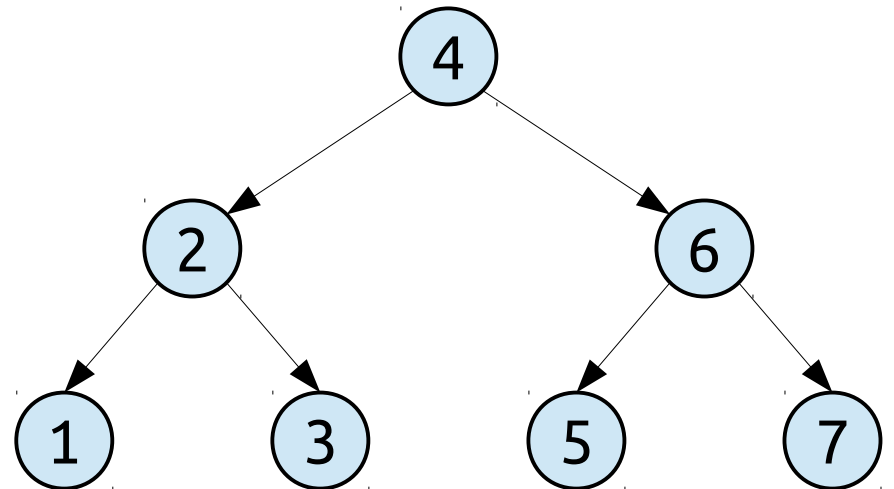
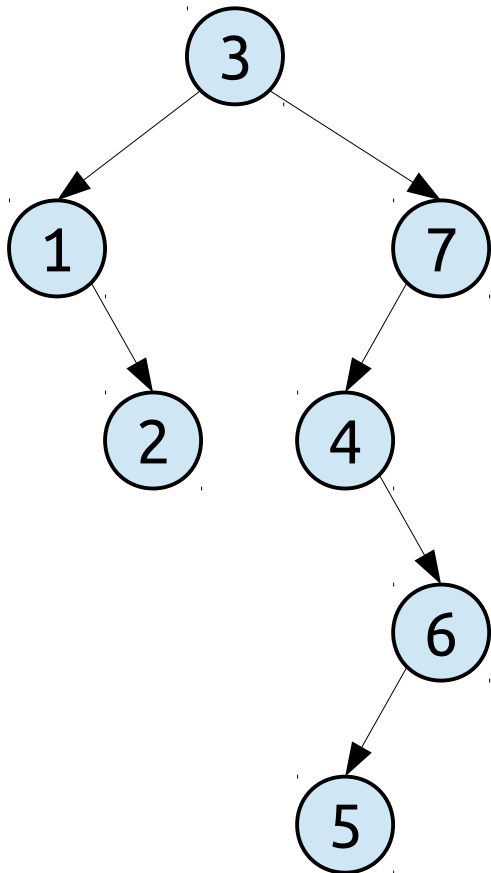
$$\begin{aligned} H_p &= \sum_{i=1}^n -p_i \lg p_i \\ &= \sum_{i=1}^n \frac{1}{n} \left(-\lg \frac{1}{n} \right) \\ &= \sum_{i=1}^n \frac{1}{n} \lg n \\ &= \lg n \end{aligned}$$

If only one element is ever accessed ($p_1 = 1, p_i = 0$), then

$$\begin{aligned} H_p &= \sum_{i=1}^n -p_i \lg p_i \\ &= -\lg 1 + \sum_{i=2}^n 0 \lg 0 \\ &= \mathbf{0} \end{aligned}$$

Model 2: Queries are sampled from a fixed, known probability distribution.

How do we know when we have a BST that's optimal with respect to *expected* lookup costs?

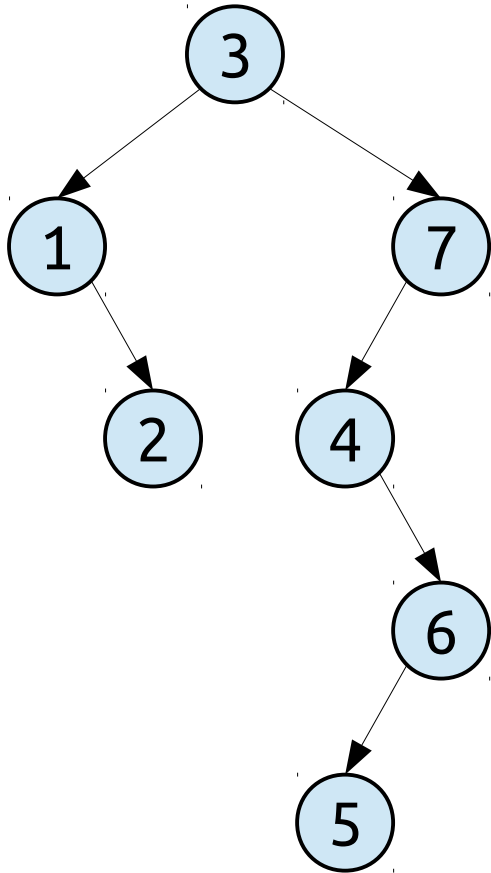


Model 2: Queries are sampled from a fixed, known probability distribution.

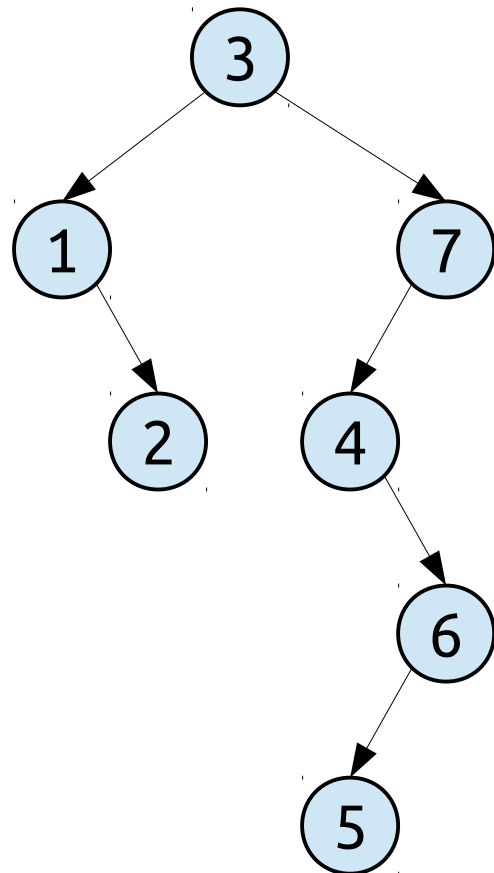
Theorem: If accesses are sampled over a discrete distribution, then the expected cost of a lookup in *any* BST is $\Omega(1 + H)$, where H is the Shannon entropy of the distribution.

A binary search tree has the **entropy property** if the (amortized) expected cost of any lookup on that BST is $O(1 + H)$.

(Any BST with this property is optimal from a *expected-case* perspective, assuming a fixed probability distribution.)



Model 2: Queries are sampled from a fixed, known probability distribution.



Theorem: If accesses are sampled over a discrete distribution, then the expected cost of a lookup in *any* BST is $\Omega(1 + H)$, where H is the Shannon entropy of the distribution.

A binary search tree has the **entropy property** if the (amortized) expected cost of any lookup on that BST is $O(1 + H)$.

Theorem (Mehlhorn, 1975): Weight-balanced trees have the entropy property...

Theorem (Fredman, 1975): ... and they can be built in time $O(n)$ if the keys are already sorted.

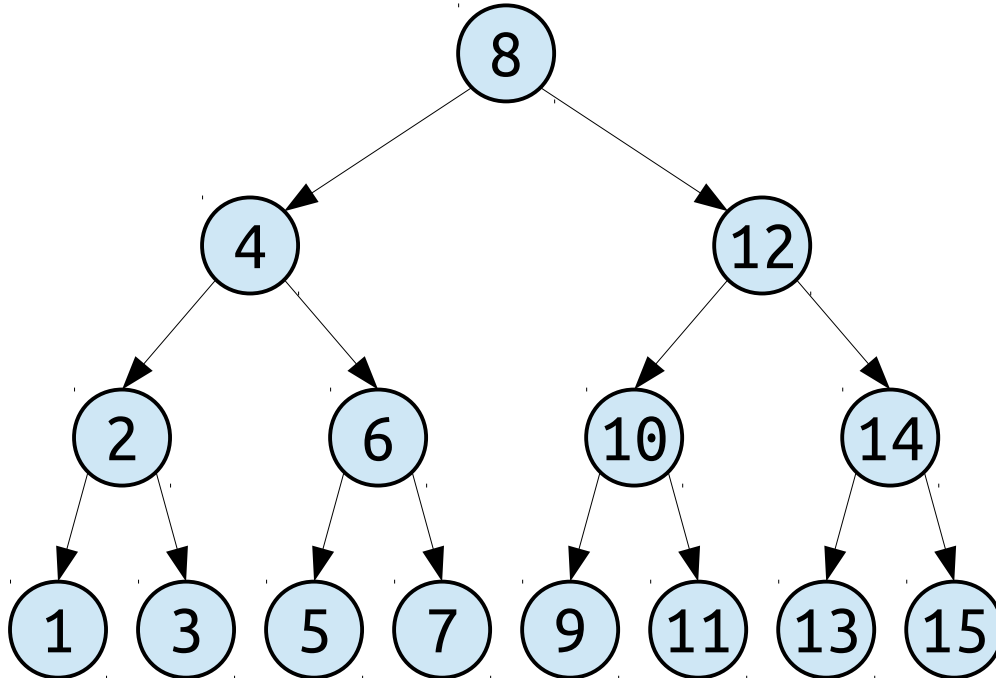
Model 2: Queries are sampled from a fixed, known probability distribution.

Suppose we want to look up all elements in the tree once and in order.

Elements are all accessed with the same frequency, so $H = \lg n$.

Both the balance and entropy properties say each query should take time $O(\log n)$.

Question: Can we do better?



Model 3: Queries have *spatial locality*. If a key is queried, keys with nearby values will likely be queried.

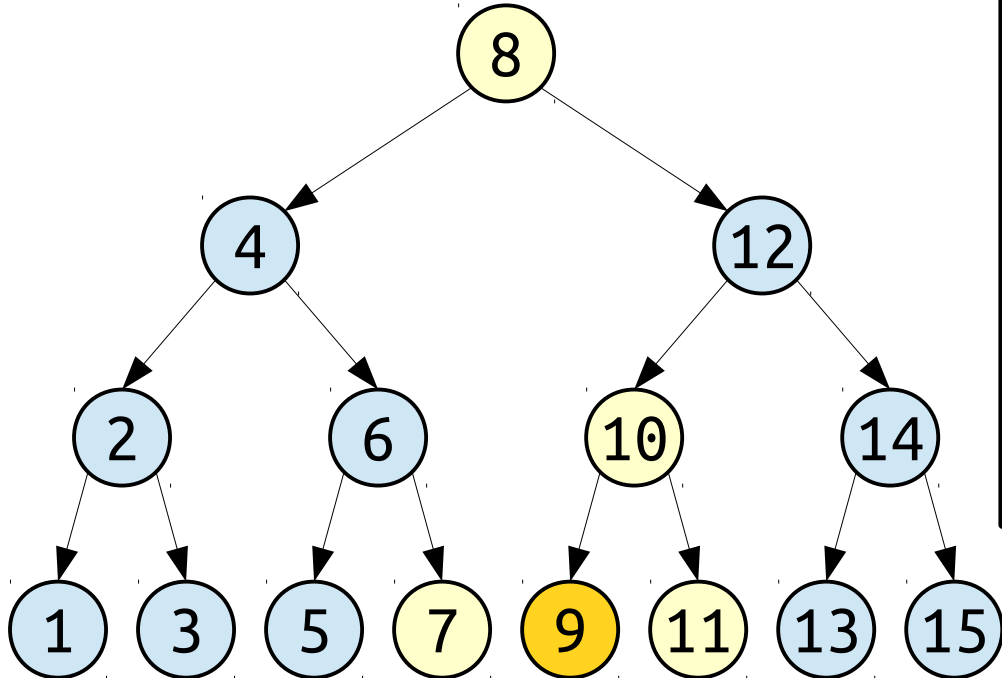
Imagine that our BST keys represent times.

If someone looks up an event at one time, they'll likely search for other events around that time.

The entropy bound of $O(1 + H)$ is probably better than $O(\log n)$ here.

However, accesses aren't uniformly random.

Question: Can we do better?

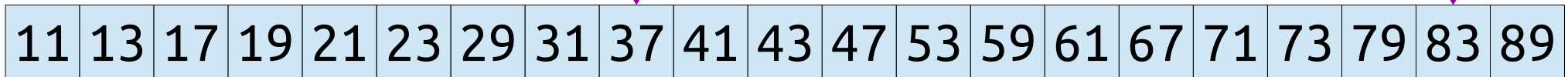


Model 3: Queries have *spatial locality*. If a key is queried, keys with nearby values will likely be queried.

Suppose our last search was for some key x_1 . Our next search is for key x_2 . We know where key x_1 is.

... it should be faster to find 37, which is near...

... than 83, which is far.



If the last key we searched for was 21...

Model 3: Queries have **spatial locality**. If a key is queried, keys with nearby values will likely be queried.

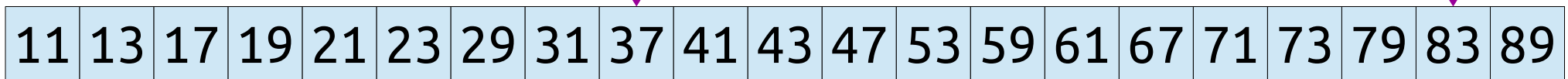
Suppose our last search was for some key x_1 . Our next search is for key x_2 . We know where key x_1 is.

$$\text{Let } \Delta = |\text{rank}(x_2) - \text{rank}(x_1)|.$$

(The number of positions away the two elements are in the sorted sequence.)

... it should be faster to find 37, which is near...

... than 83, which is far.



If the last key we searched for was 21...

Model 3: Queries have **spatial locality**. If a key is queried, keys with nearby values will likely be queried.

Suppose our last search was for some key x_1 . Our next search is for key x_2 . We know where key x_1 is.

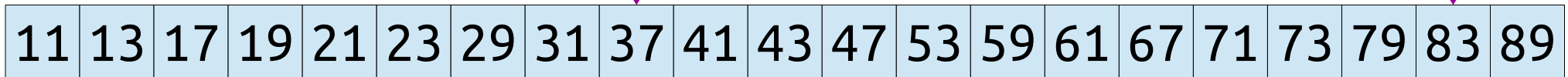
Let $\Delta = |\text{rank}(x_2) - \text{rank}(x_1)|$.

Can we do the search in time $O(\Delta)$?
How about time $O(\log \Delta)$?

(The number of positions away the two elements are in the sorted sequence.)

... it should be faster to find 37, which is near...

... than 83, which is far.



If the last key we searched for was 21...

Model 3: Queries have **spatial locality**. If a key is queried, keys with nearby values will likely be queried.

Suppose our last search was for some key x_1 . Our next search is for key x_2 . We know where key x_1 is.

Let $\Delta = |\text{rank}(x_2) - \text{rank}(x_1)|$.

Can we do the search in time $O(\Delta)$?

How about time $O(\log \Delta)$?

Idea: Just do a simple linear scan.

11	13	17	19	21	23	29	31	37	41	43	47	53	59	61	67	71	73	79	83	89
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----



Model 3: Queries have **spatial locality**. If a key is queried, keys with nearby values will likely be queried.

Suppose our last search was for some key x_1 . Our next search is for key x_2 . We know where key x_1 is.

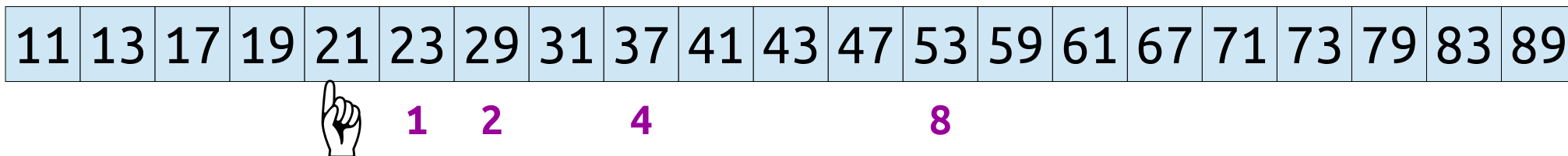
$$\text{Let } \Delta = |\text{rank}(x_2) - \text{rank}(x_1)|.$$

Can we do the search in time $O(\Delta)$?

How about time $O(\log \Delta)$?

Idea: Use an *exponential search* to overshoot, then binary search over the range.

Observation: This is asymptotically at least as good as a binary search.



Model 3: Queries have *spatial locality*. If a key is queried, keys with nearby values will likely be queried.

Suppose our last search was for some key x_1 . Our next search is for key x_2 . We know where key x_1 is.

Let $\Delta = |\text{rank}(x_2) - \text{rank}(x_1)|$.

Can we do the search in time $O(\Delta)$?

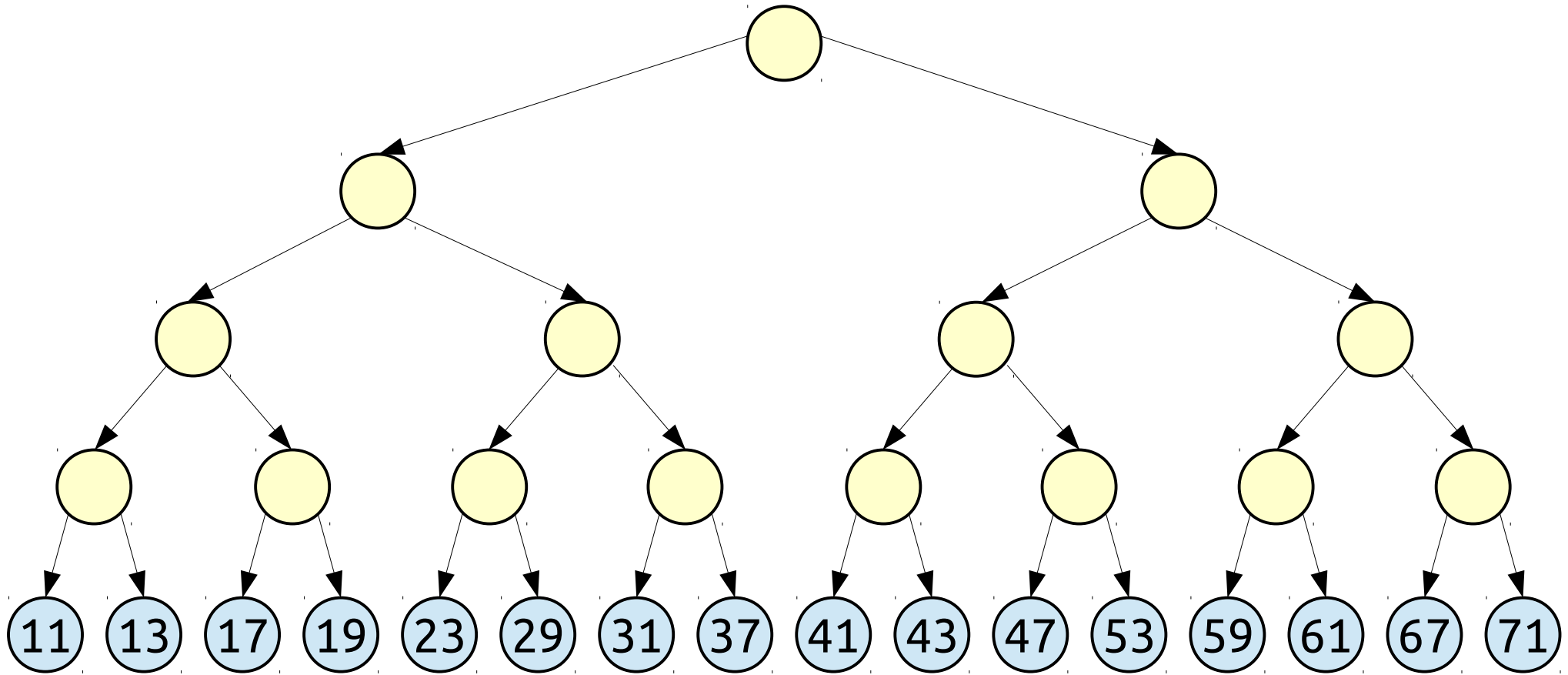
How about time $O(\log \Delta)$?

Question: Can we do this efficiently if the underlying set is changing?

11	13	17	19	21	23	29	31	37	41	43	47	53	59	61	67	71	73	79	83	89
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

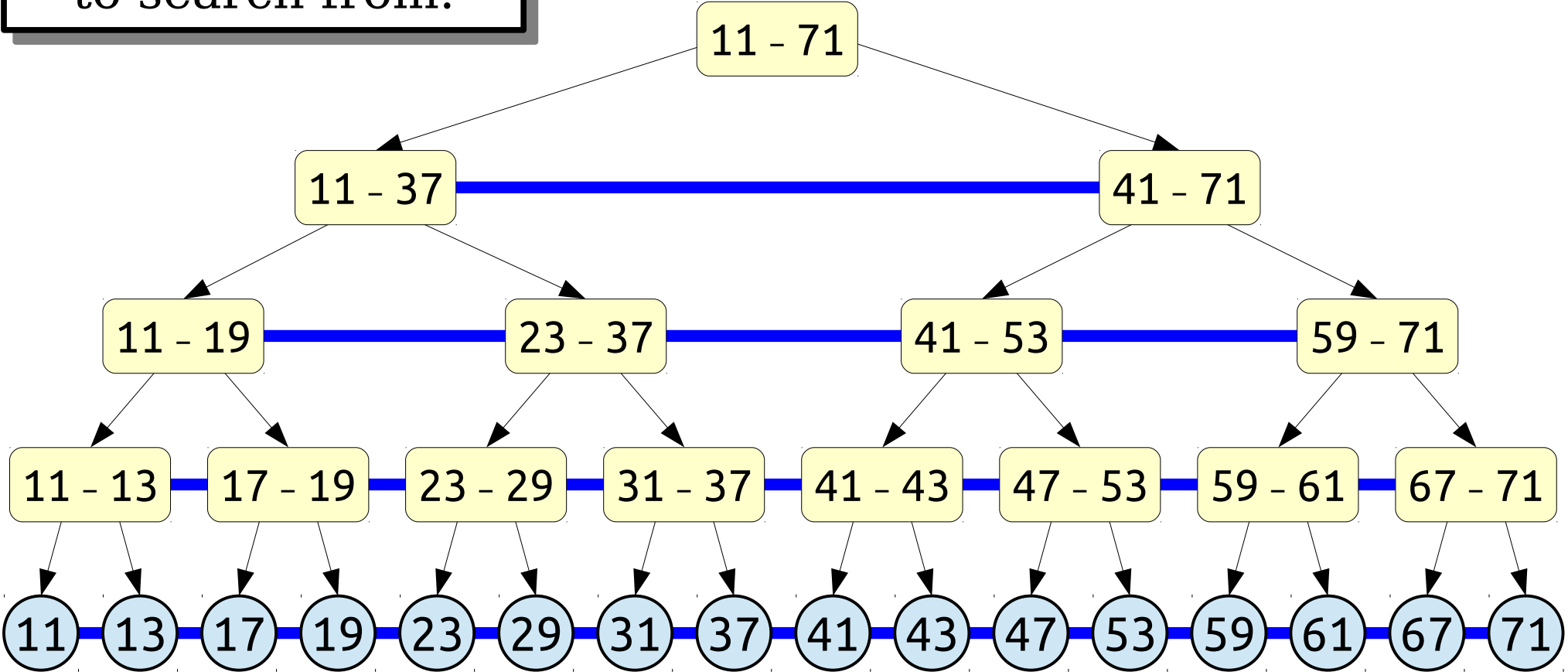


Model 3: Queries have **spatial locality**. If a key is queried, keys with nearby values will likely be queried.



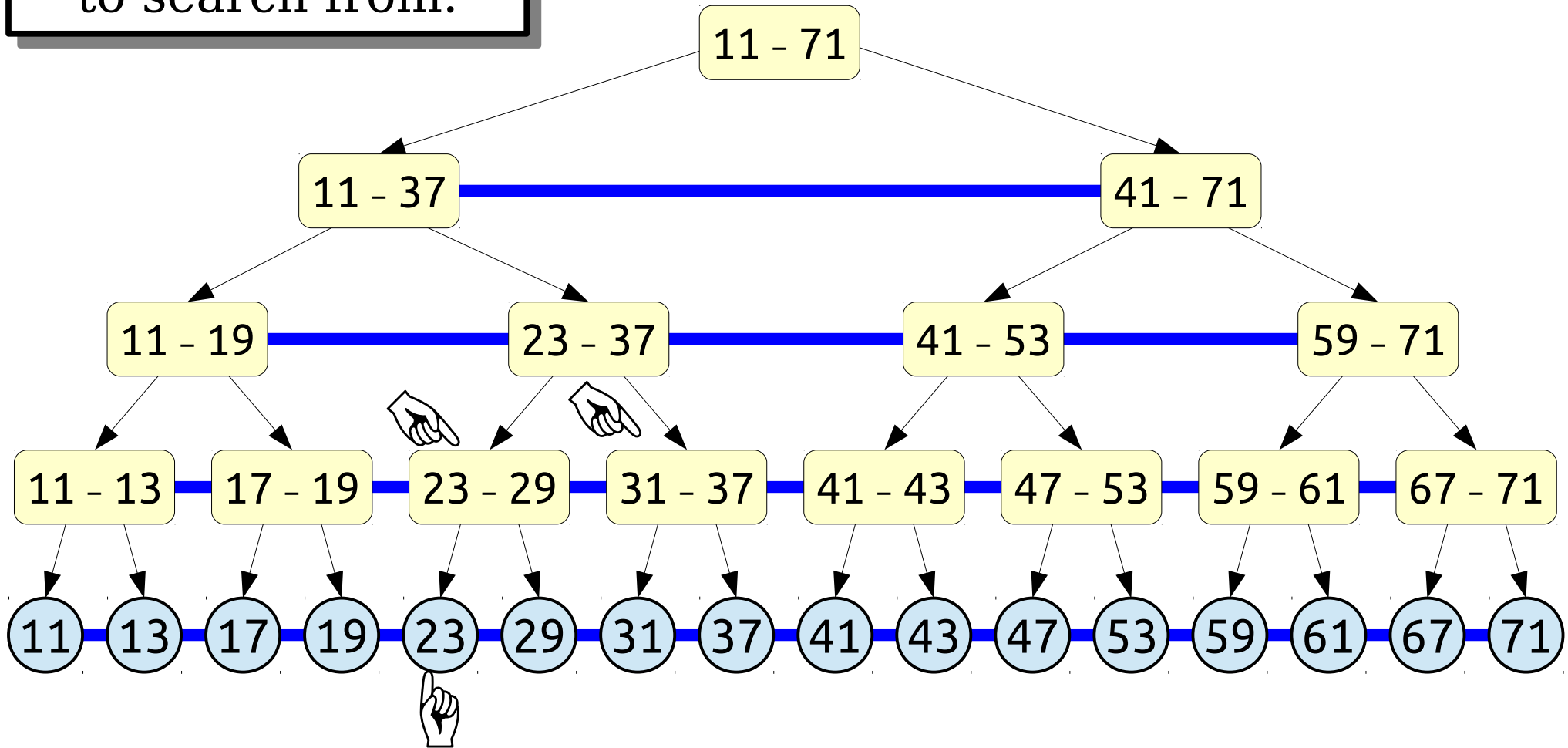
Model 3: Queries have *spatial locality*. If a key is queried, keys with nearby values will likely be queried.

Scan up, looking at sibling nodes to determine where to search from.



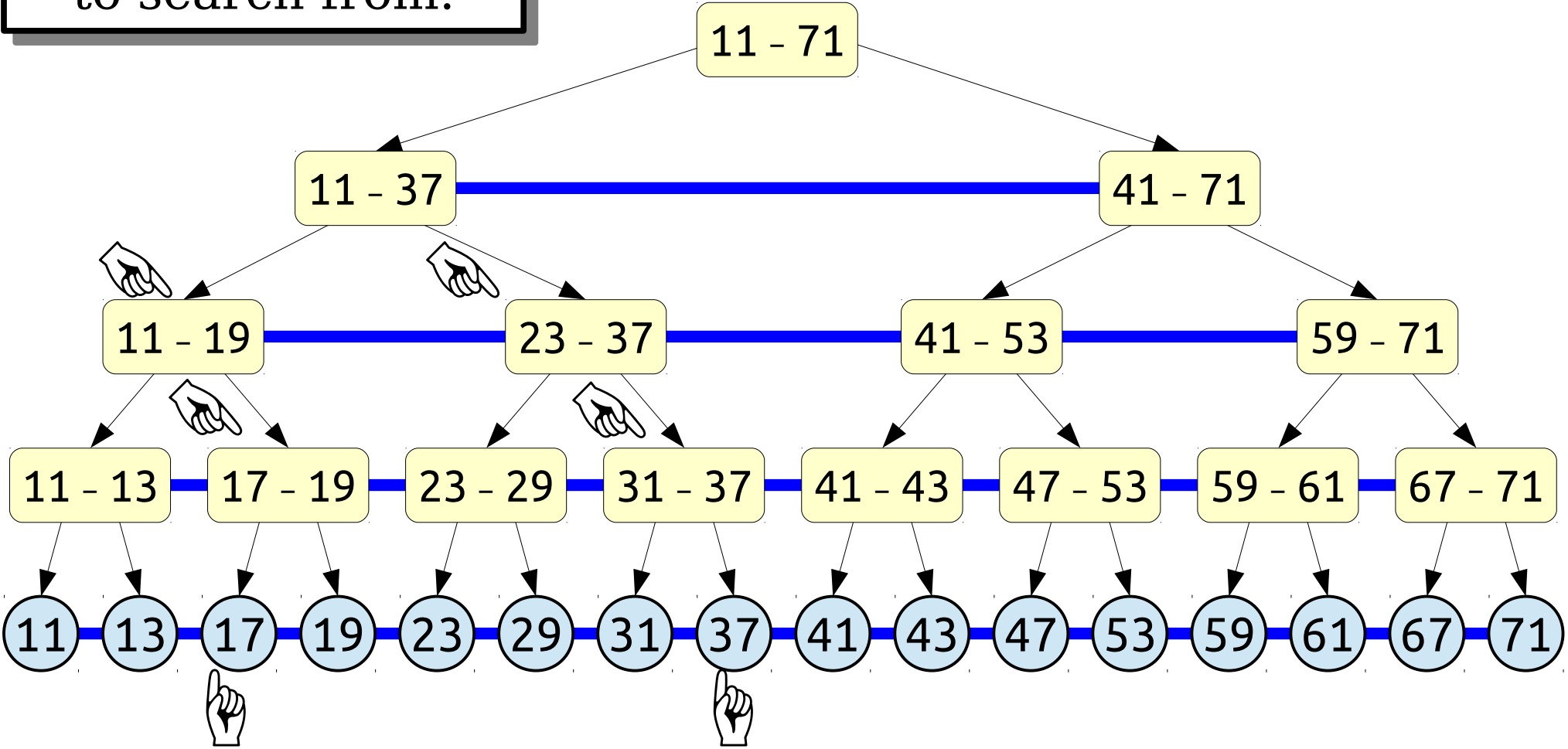
Model 3: Queries have *spatial locality*. If a key is queried, keys with nearby values will likely be queried.

Scan up, looking at sibling nodes to determine where to search from.



Model 3: Queries have *spatial locality*. If a key is queried, keys with nearby values will likely be queried.

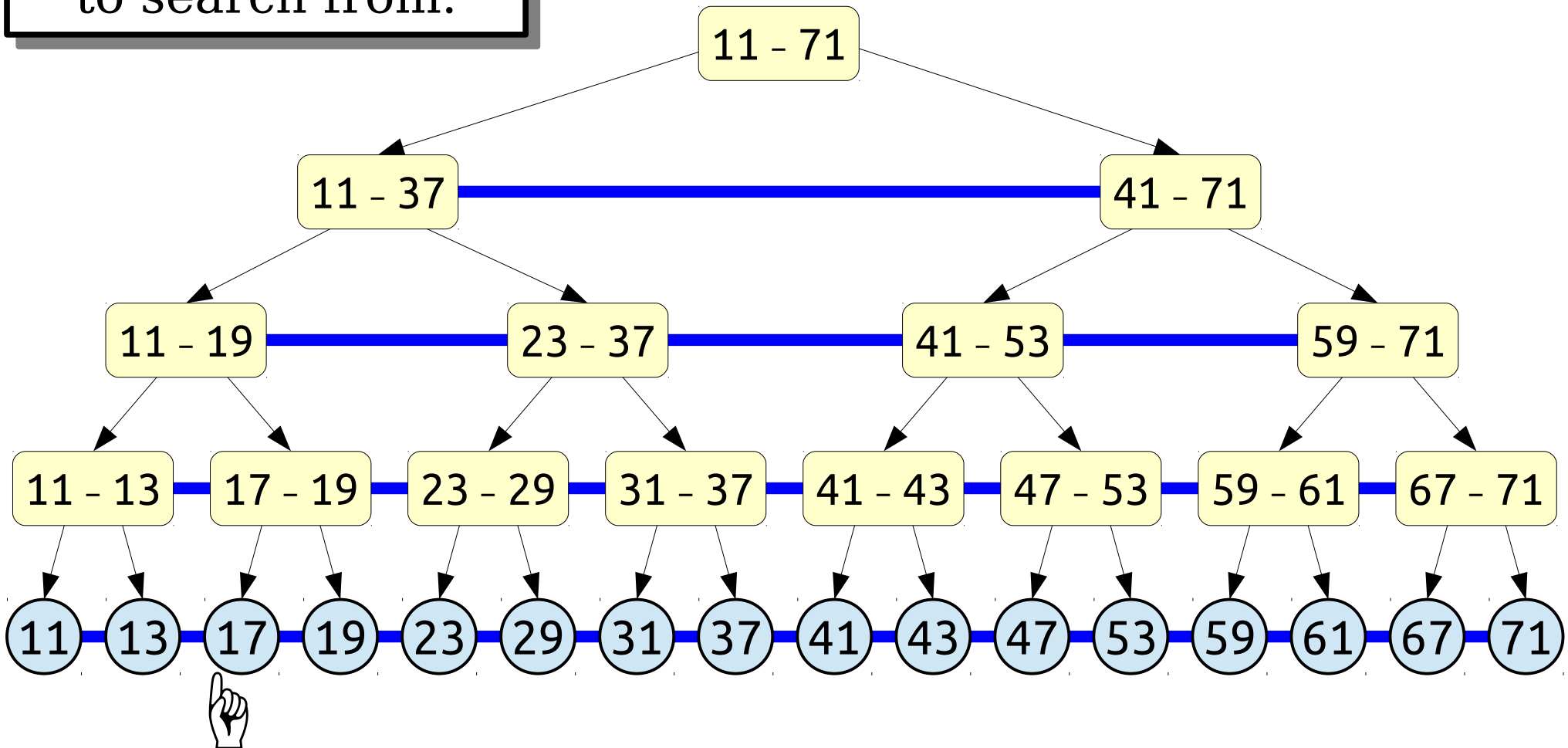
Scan up, looking at sibling nodes to determine where to search from.



Model 3: Queries have *spatial locality*. If a key is queried, keys with nearby values will likely be queried.

Scan up, looking at sibling nodes to determine where to search from.

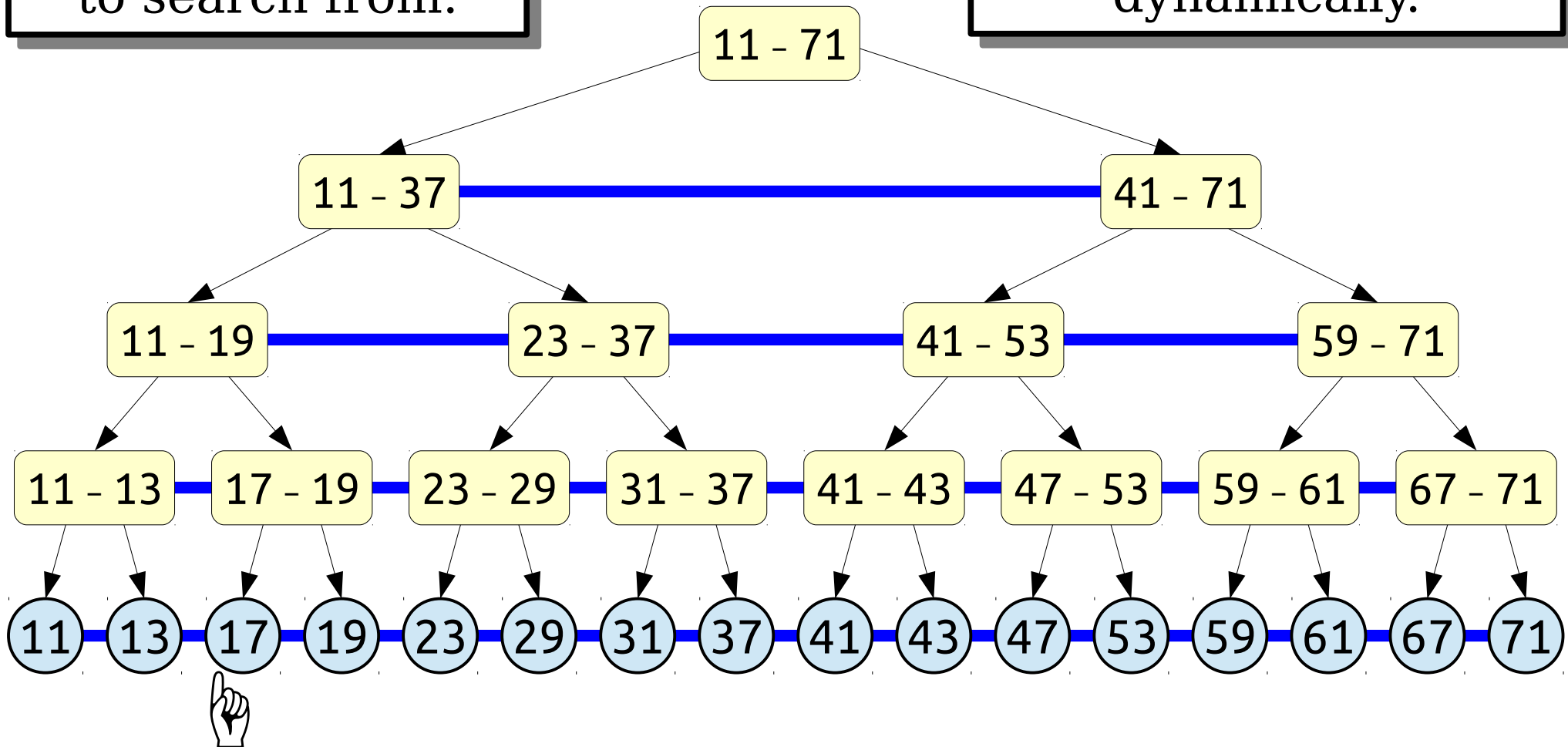
Claim: This simulates our earlier search. Runtime is $O(\log \Delta)$.



Model 3: Queries have *spatial locality*. If a key is queried, keys with nearby values will likely be queried.

Scan up, looking at sibling nodes to determine where to search from.

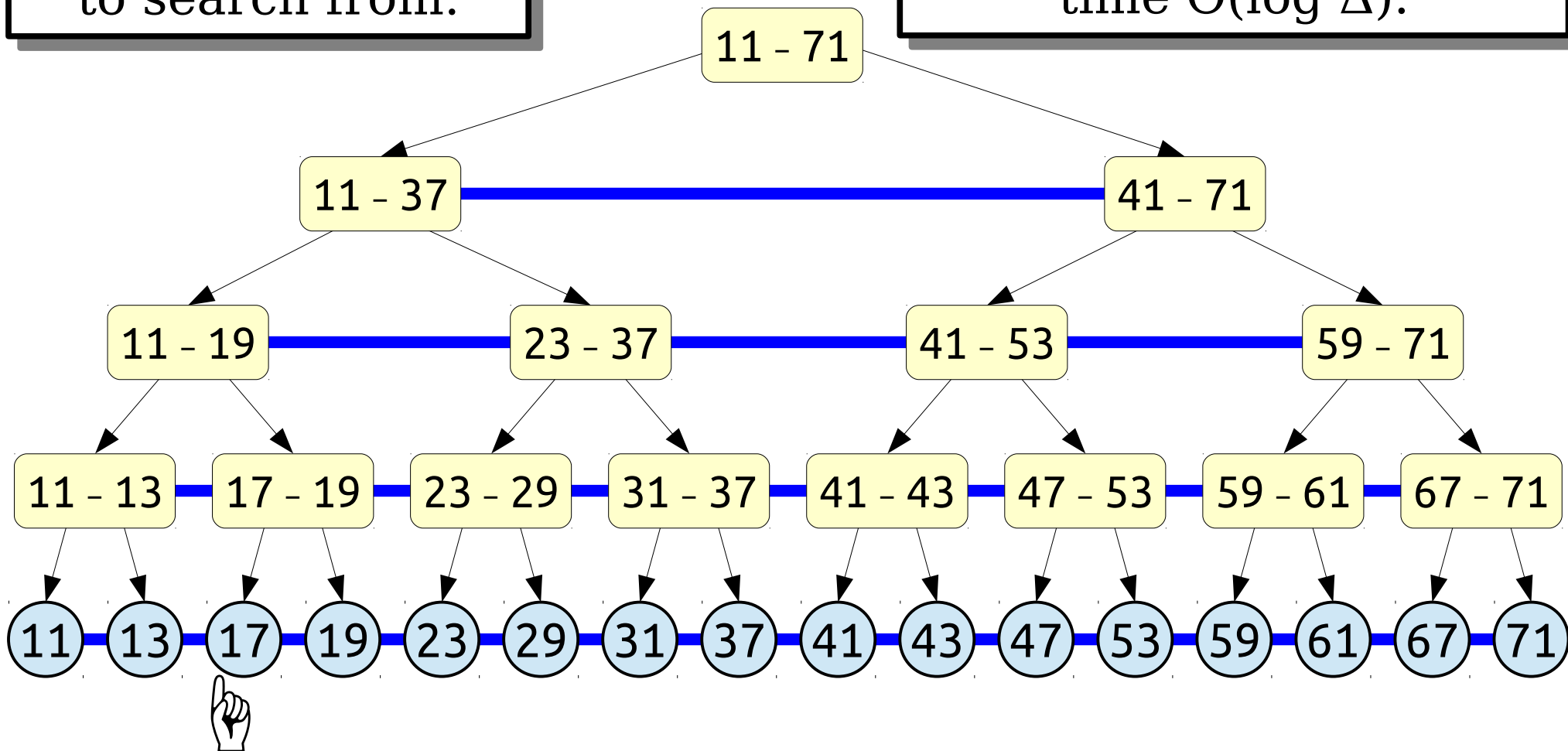
The *level-linked red/black tree* implements this dynamically.



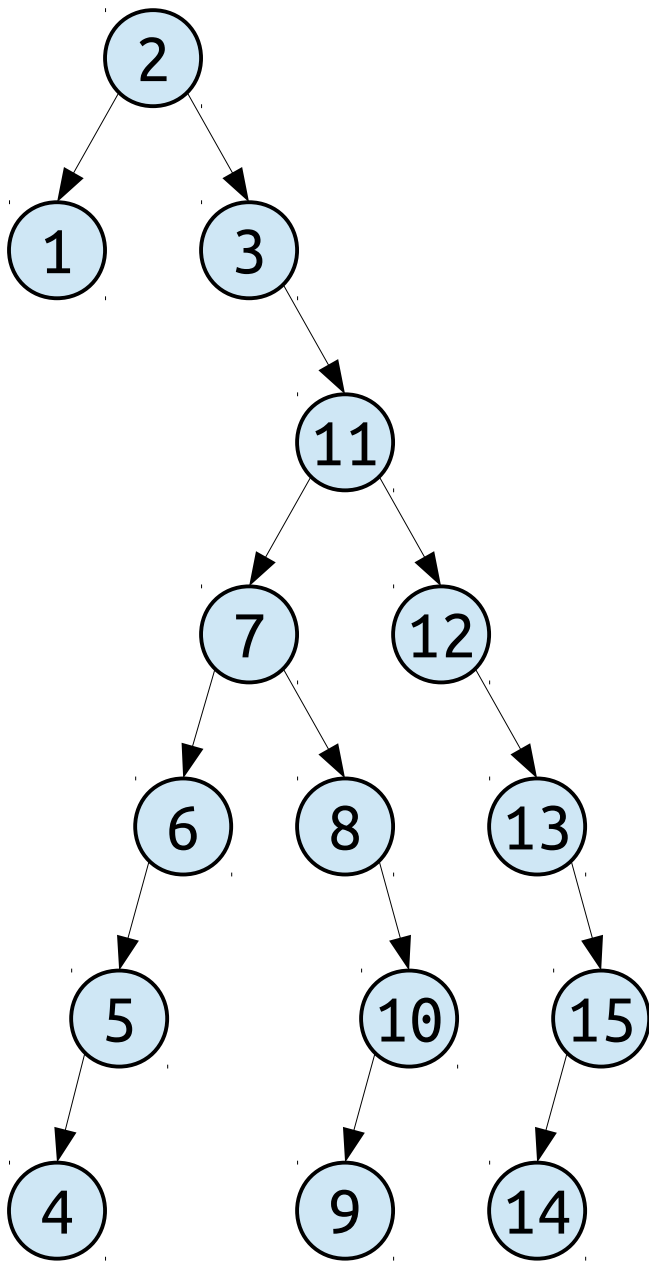
Model 3: Queries have *spatial locality*. If a key is queried, keys with nearby values will likely be queried.

Scan up, looking at sibling nodes to determine where to search from.

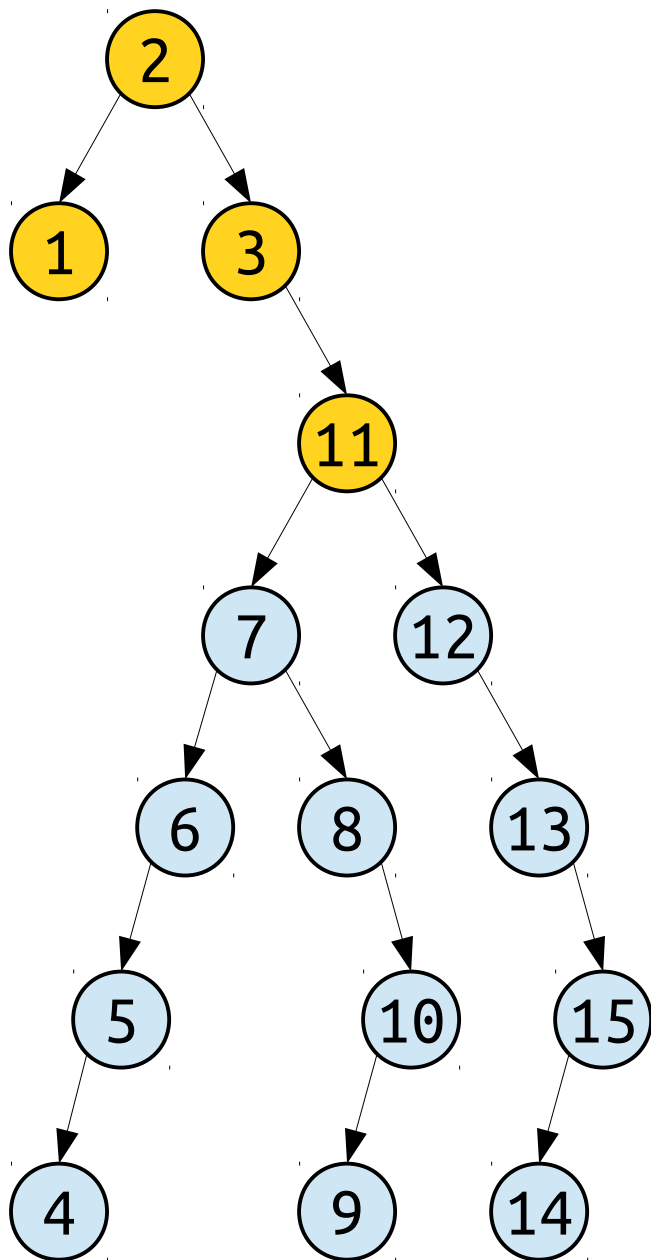
A BST has the *dynamic finger property* if lookups take (amortized) time $O(\log \Delta)$.



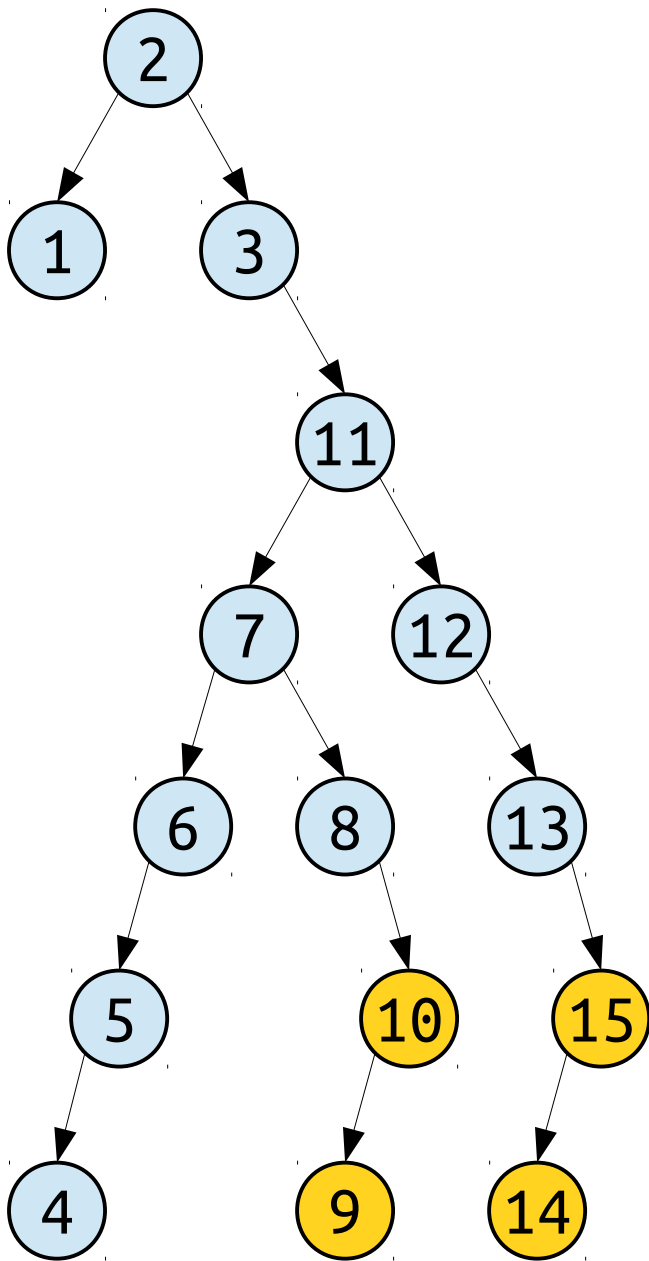
Model 3: Queries have *spatial locality*. If a key is queried, keys with nearby values will likely be queried.



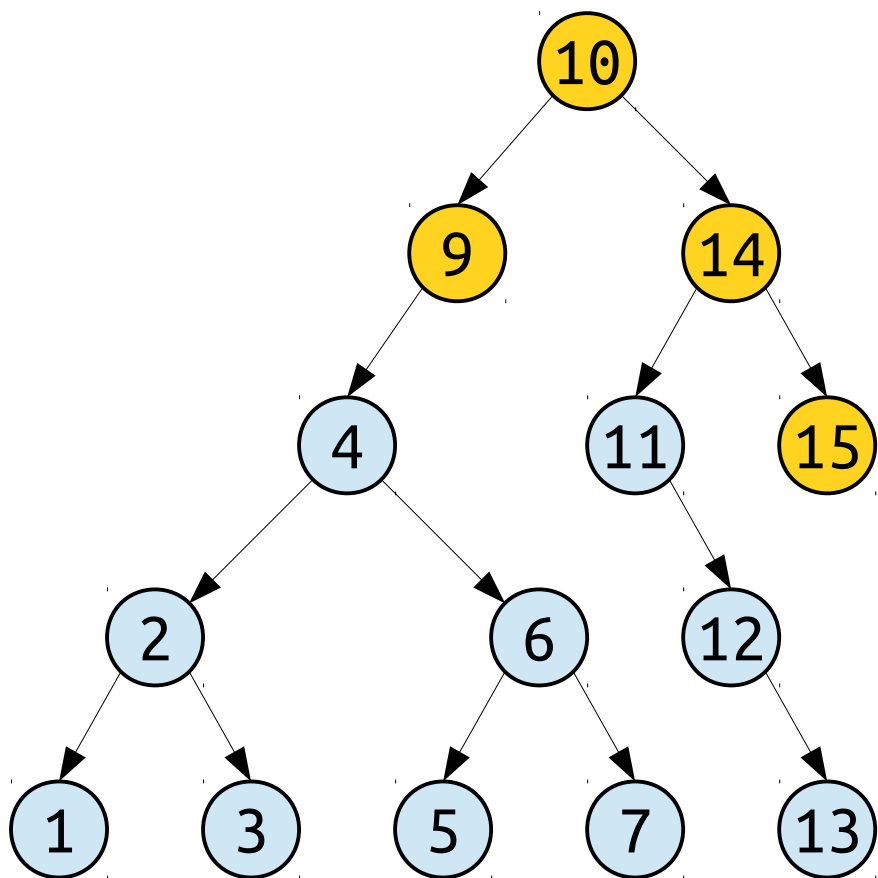
Model 4: Queries have *temporal locality*. If a key is queried, it's likely going to be queried again soon.



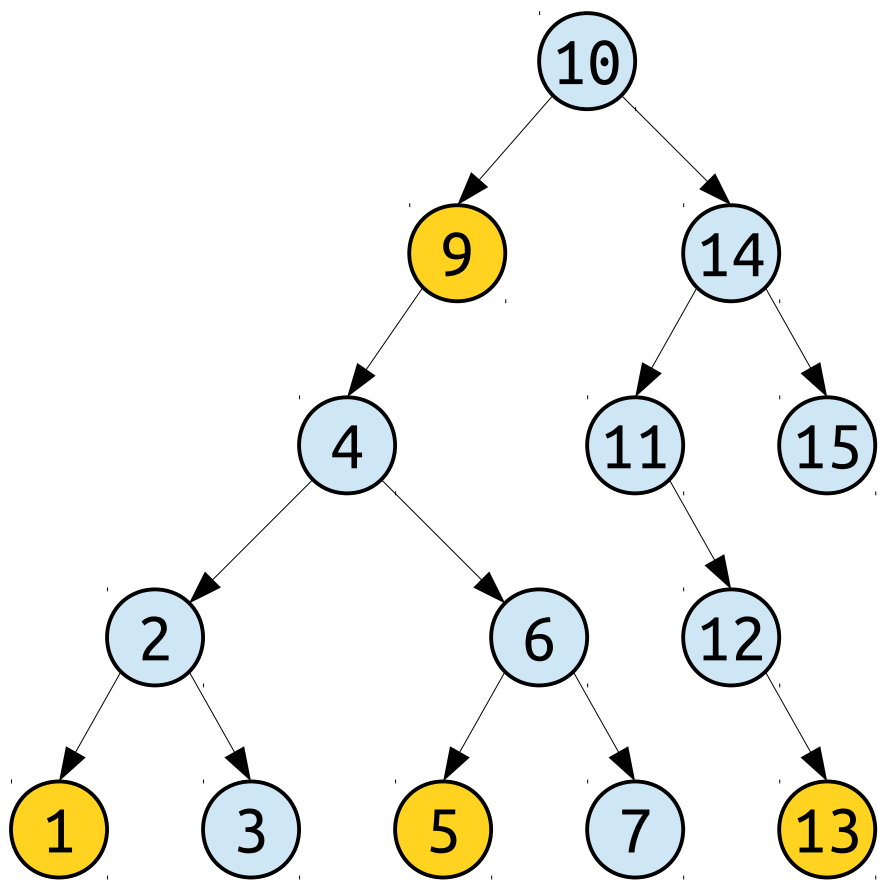
Model 4: Queries have *temporal locality*. If a key is queried, it's likely going to be queried again soon.



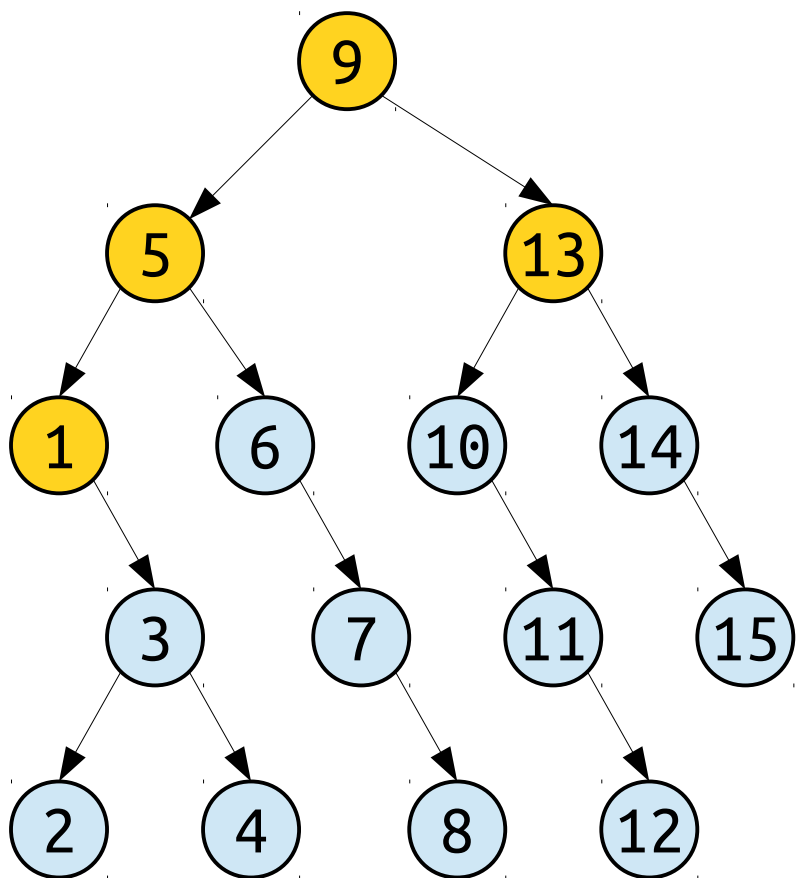
Model 4: Queries have **temporal locality**. If a key is queried, it's likely going to be queried again soon.



Model 4: Queries have *temporal locality*. If a key is queried, it's likely going to be queried again soon.

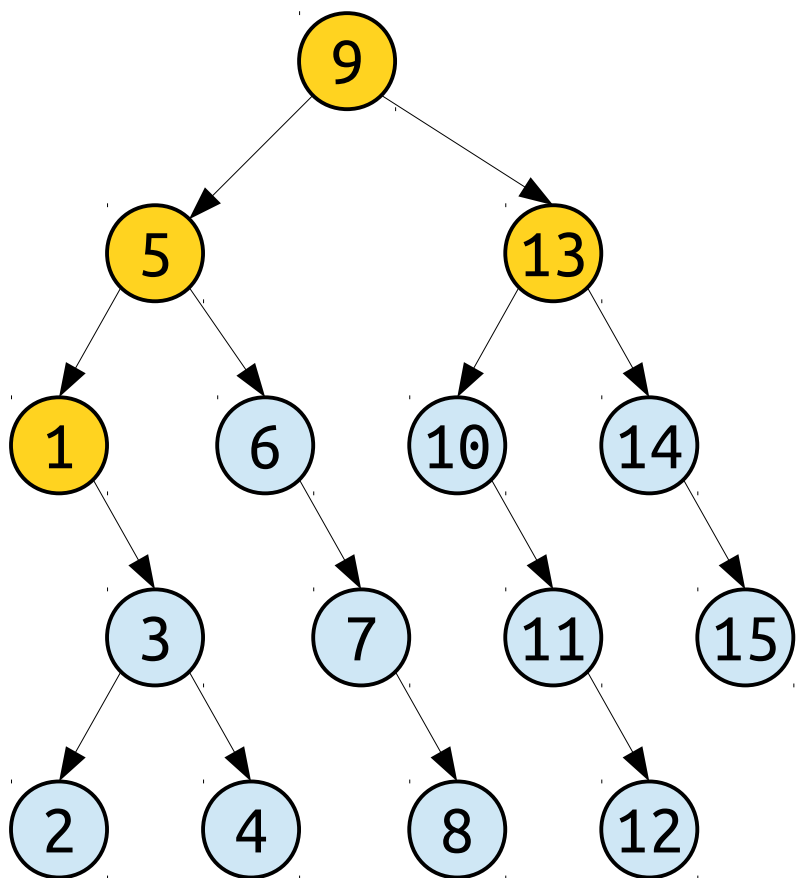


Model 4: Queries have *temporal locality*. If a key is queried, it's likely going to be queried again soon.



Goal: If only t elements are “hot” at a particular time, make accesses to those “hot” elements take time $O(\log t)$, not $O(\log n)$.

Model 4: Queries have **temporal locality**. If a key is queried, it’s likely going to be queried again soon.

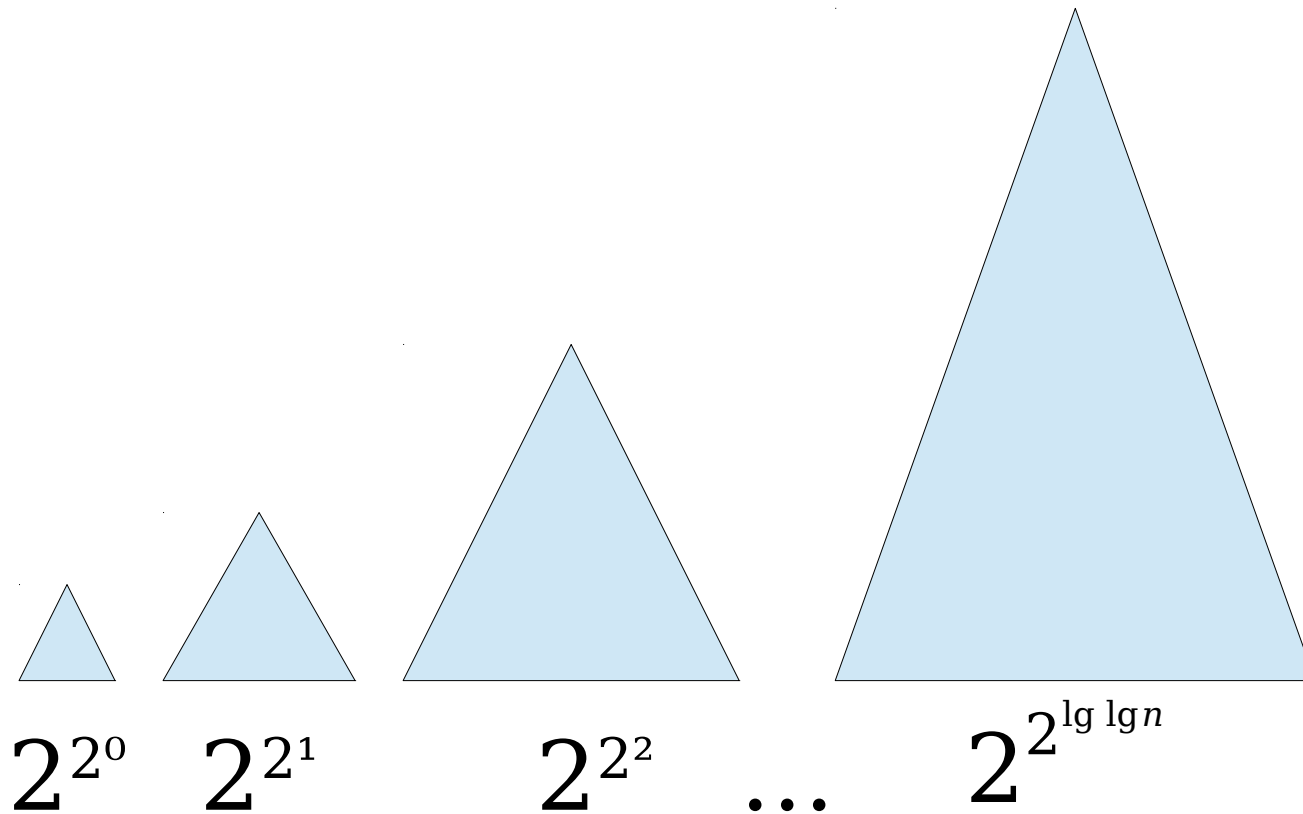


Intuition: Any tree structure with a fixed shape is going to have a hard time making these queries fast.

Idea: What if we move elements around?

Model 4: Queries have **temporal locality**. If a key is queried, it's likely going to be queried again soon.

Intuition: Use a sequence of trees. Keep “hot” elements in the earlier trees.

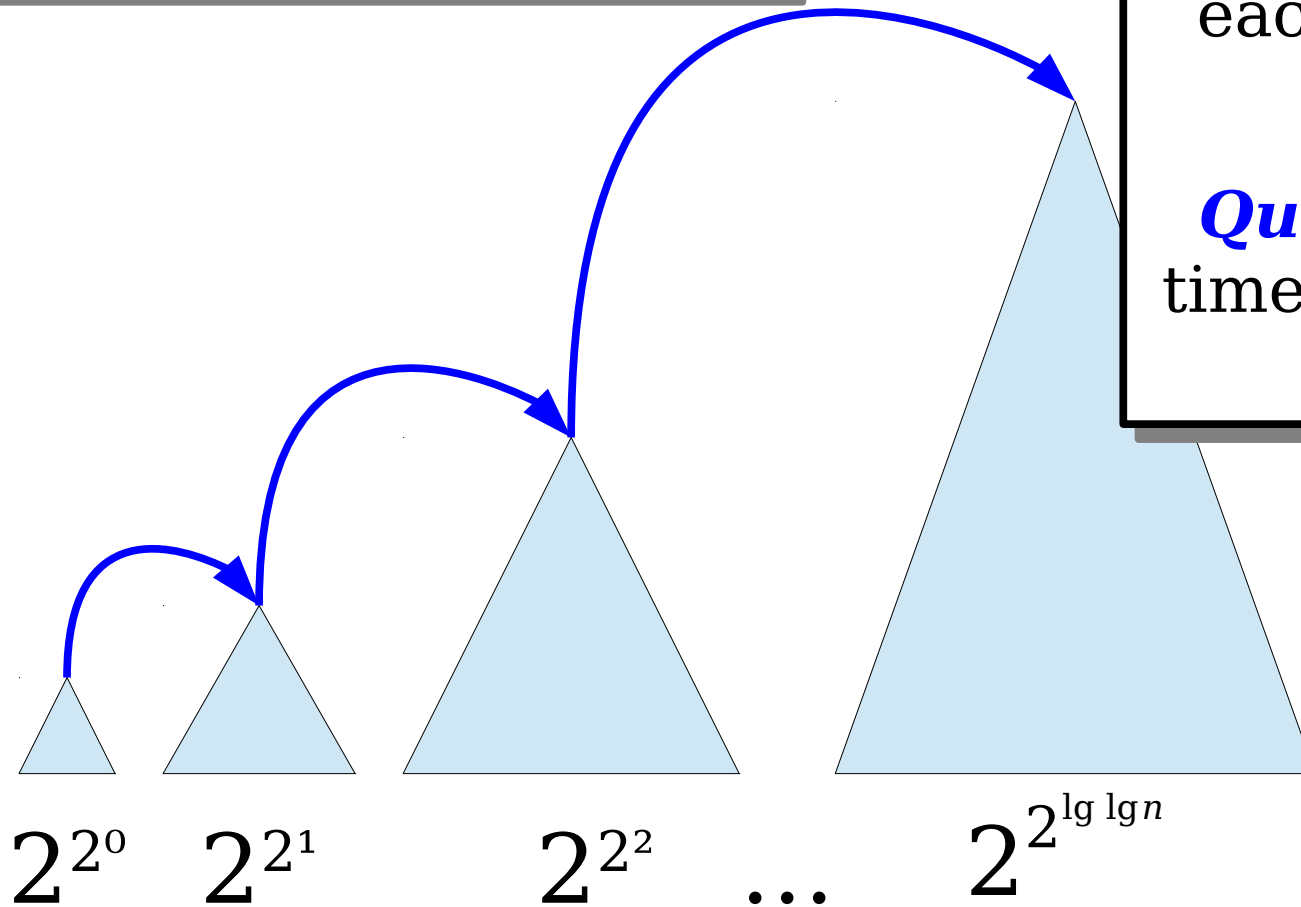


Model 4: Queries have **temporal locality**. If a key is queried, it's likely going to be queried again soon.

Intuition: Use a sequence of trees. Keep “hot” elements in the earlier trees.

To insert an element, put it in the first tree. Then, repeatedly kick the oldest element out of each tree and into the next.

Question: How much time does this take, as a function of n ?



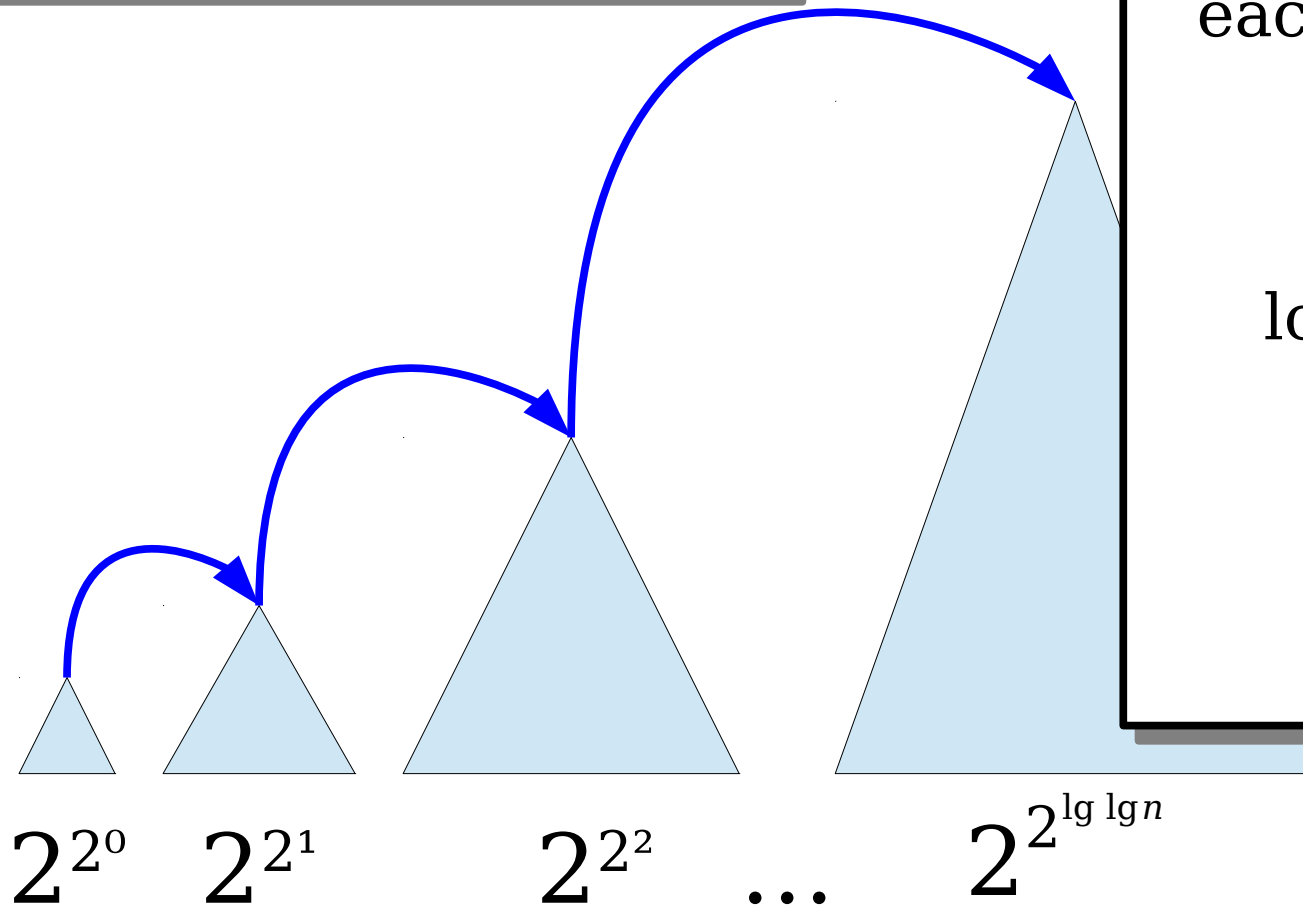
Model 4: Queries have **temporal locality**. If a key is queried, it's likely going to be queried again soon.

Intuition: Use a sequence of trees. Keep “hot” elements in the earlier trees.

To insert an element, put it in the first tree. Then, repeatedly kick the oldest element out of each tree and into the next.

Cost:

$$\log 2^{2^0} + \dots + \log 2^{2^{\lg \lg n}}$$



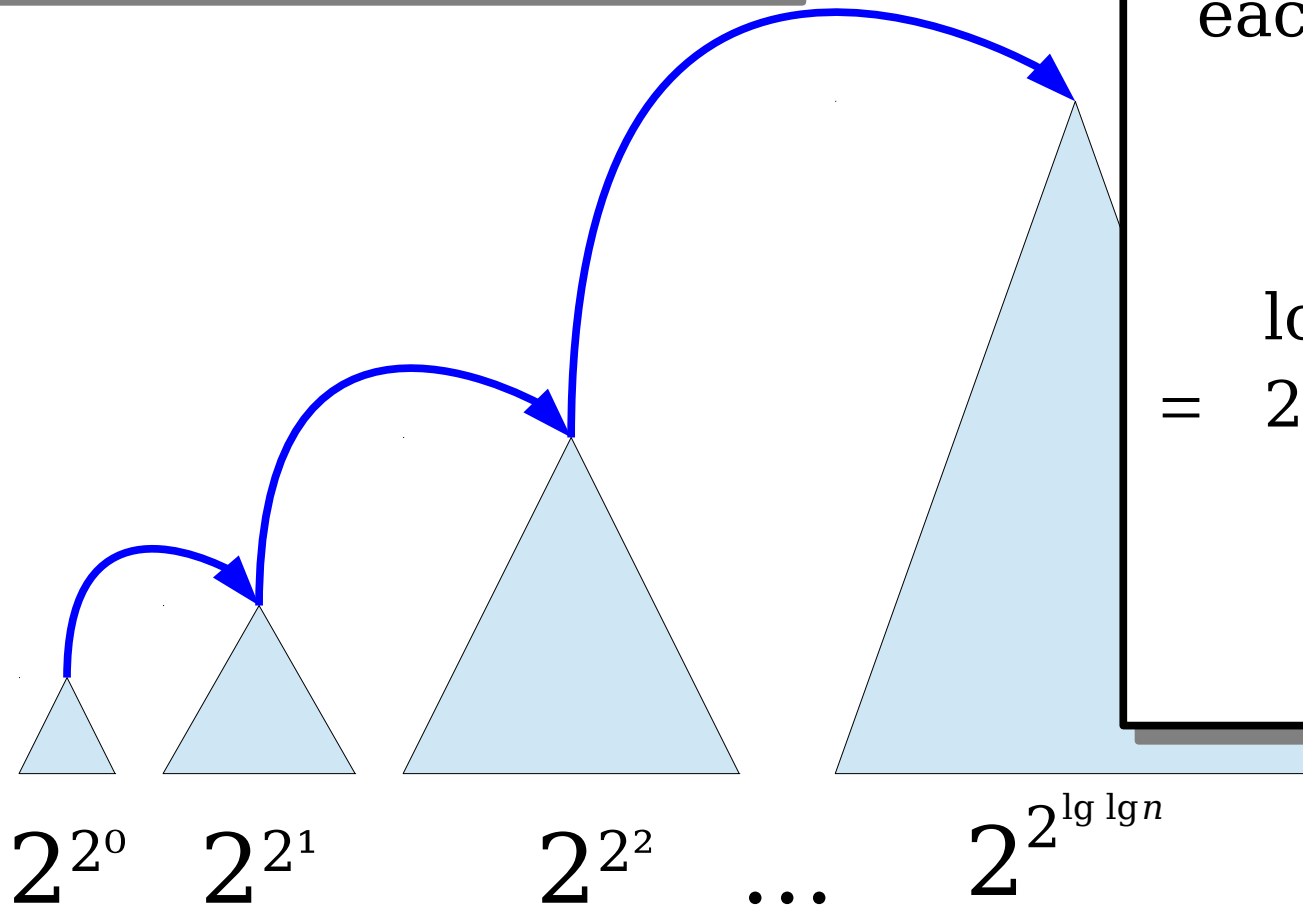
Model 4: Queries have **temporal locality**. If a key is queried, it's likely going to be queried again soon.

Intuition: Use a sequence of trees. Keep “hot” elements in the earlier trees.

To insert an element, put it in the first tree. Then, repeatedly kick the oldest element out of each tree and into the next.

Cost:

$$\begin{aligned} & \log 2^{2^0} + \dots + \log 2^{2^{\lg \lg n}} \\ = & 2^0 + 2^1 + \dots + 2^{\lg \lg n} \end{aligned}$$



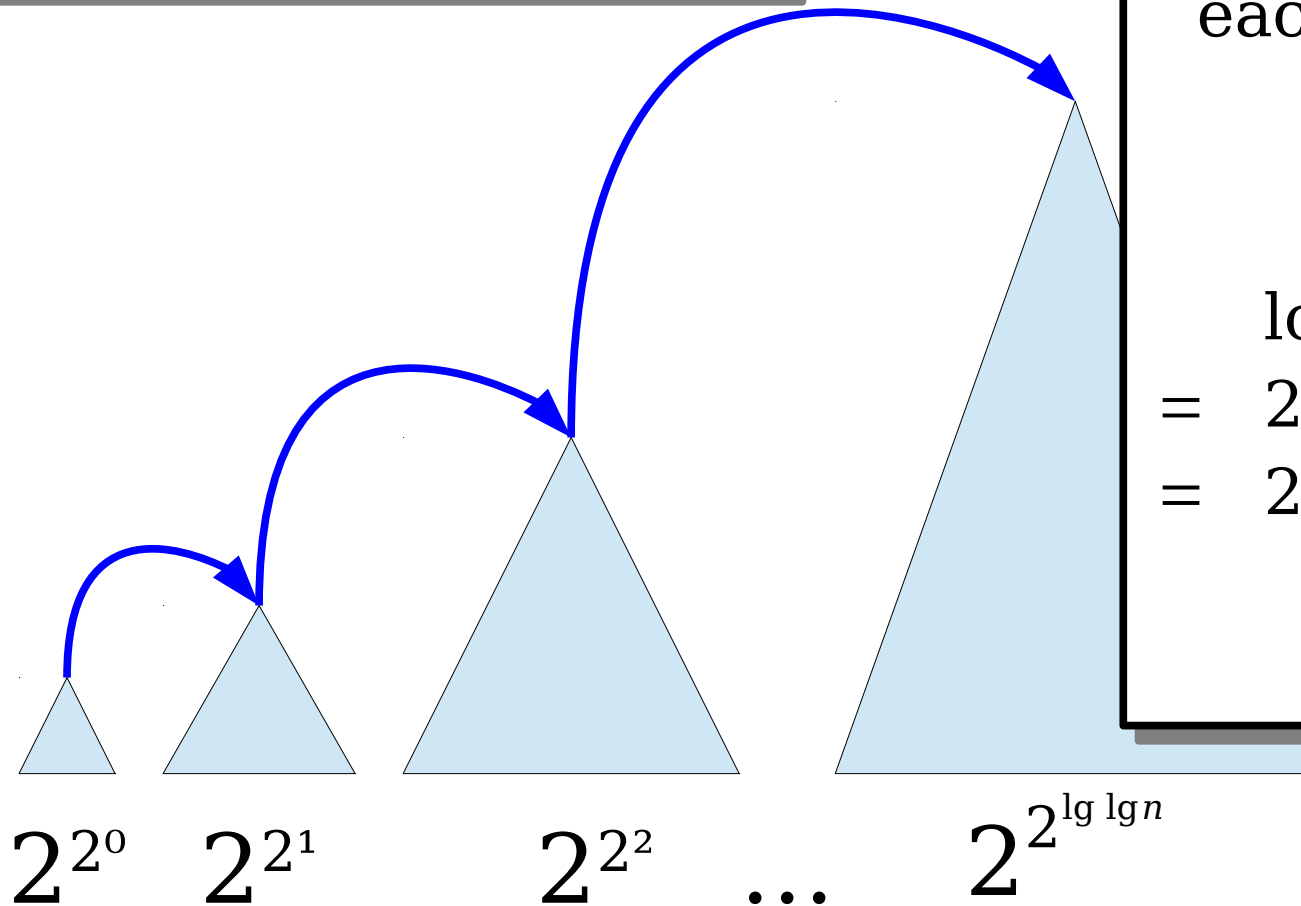
Model 4: Queries have **temporal locality**. If a key is queried, it's likely going to be queried again soon.

Intuition: Use a sequence of trees. Keep “hot” elements in the earlier trees.

To insert an element, put it in the first tree. Then, repeatedly kick the oldest element out of each tree and into the next.

Cost:

$$\begin{aligned} & \log 2^{2^0} + \dots + \log 2^{2^{\lg \lg n}} \\ &= 2^0 + 2^1 + \dots + 2^{\lg \lg n} \\ &= 2^{1+\lg \lg n} - 1 \end{aligned}$$



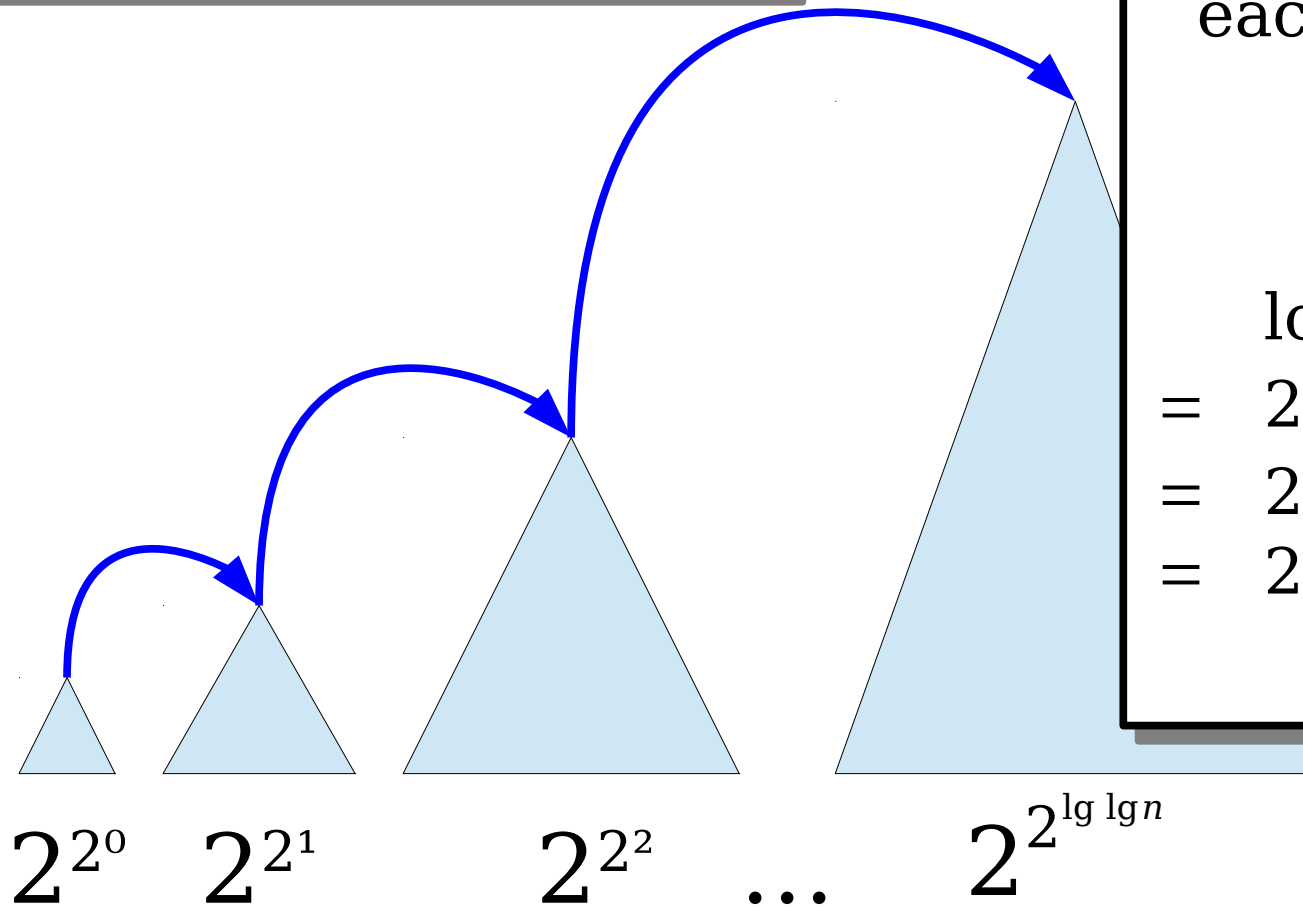
Model 4: Queries have **temporal locality**. If a key is queried, it's likely going to be queried again soon.

Intuition: Use a sequence of trees. Keep “hot” elements in the earlier trees.

To insert an element, put it in the first tree. Then, repeatedly kick the oldest element out of each tree and into the next.

Cost:

$$\begin{aligned} & \log 2^{2^0} + \dots + \log 2^{2^{\lg \lg n}} \\ &= 2^0 + 2^1 + \dots + 2^{\lg \lg n} \\ &= 2^{1+\lg \lg n} - 1 \\ &= 2 \lg n - 1 \end{aligned}$$



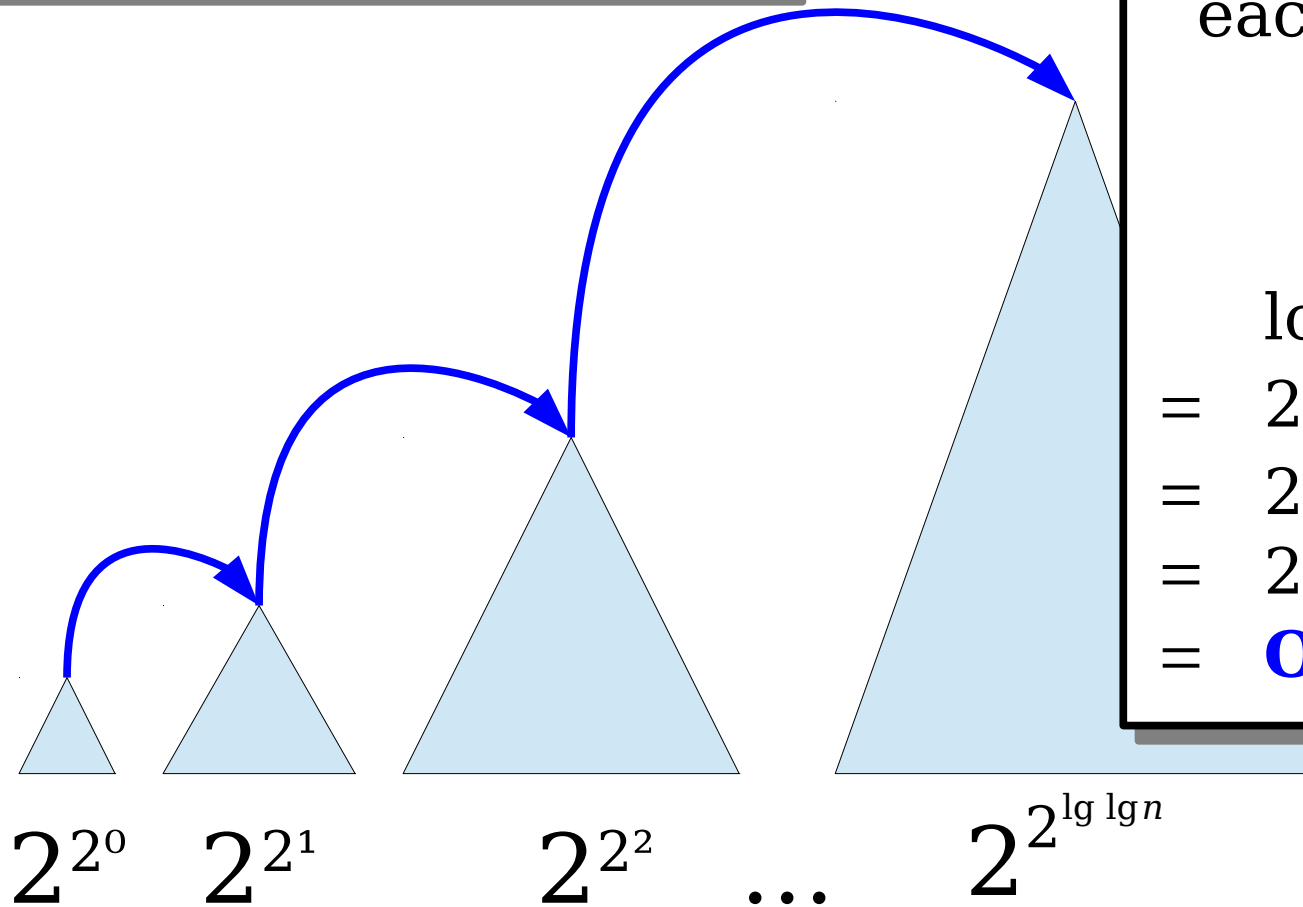
Model 4: Queries have **temporal locality**. If a key is queried, it's likely going to be queried again soon.

Intuition: Use a sequence of trees. Keep “hot” elements in the earlier trees.

To insert an element, put it in the first tree. Then, repeatedly kick the oldest element out of each tree and into the next.

Cost:

$$\begin{aligned} & \log 2^{2^0} + \dots + \log 2^{2^{\lg \lg n}} \\ &= 2^0 + 2^1 + \dots + 2^{\lg \lg n} \\ &= 2^{1+\lg \lg n} - 1 \\ &= 2 \lg n - 1 \\ &= \mathbf{O(\log n)} \end{aligned}$$



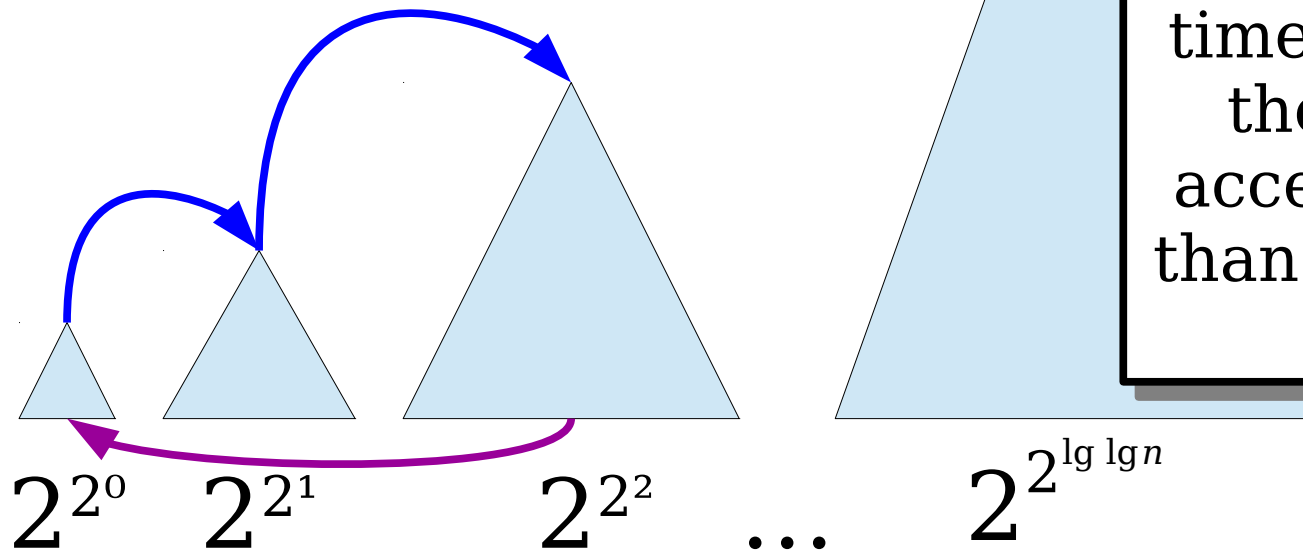
Model 4: Queries have **temporal locality**. If a key is queried, it's likely going to be queried again soon.

Intuition: Use a sequence of trees. Keep “hot” elements in the earlier trees.

To look up an element, search each tree in order, move it to the first tree, then kick older elements back.

Elements are roughly sorted by access time.

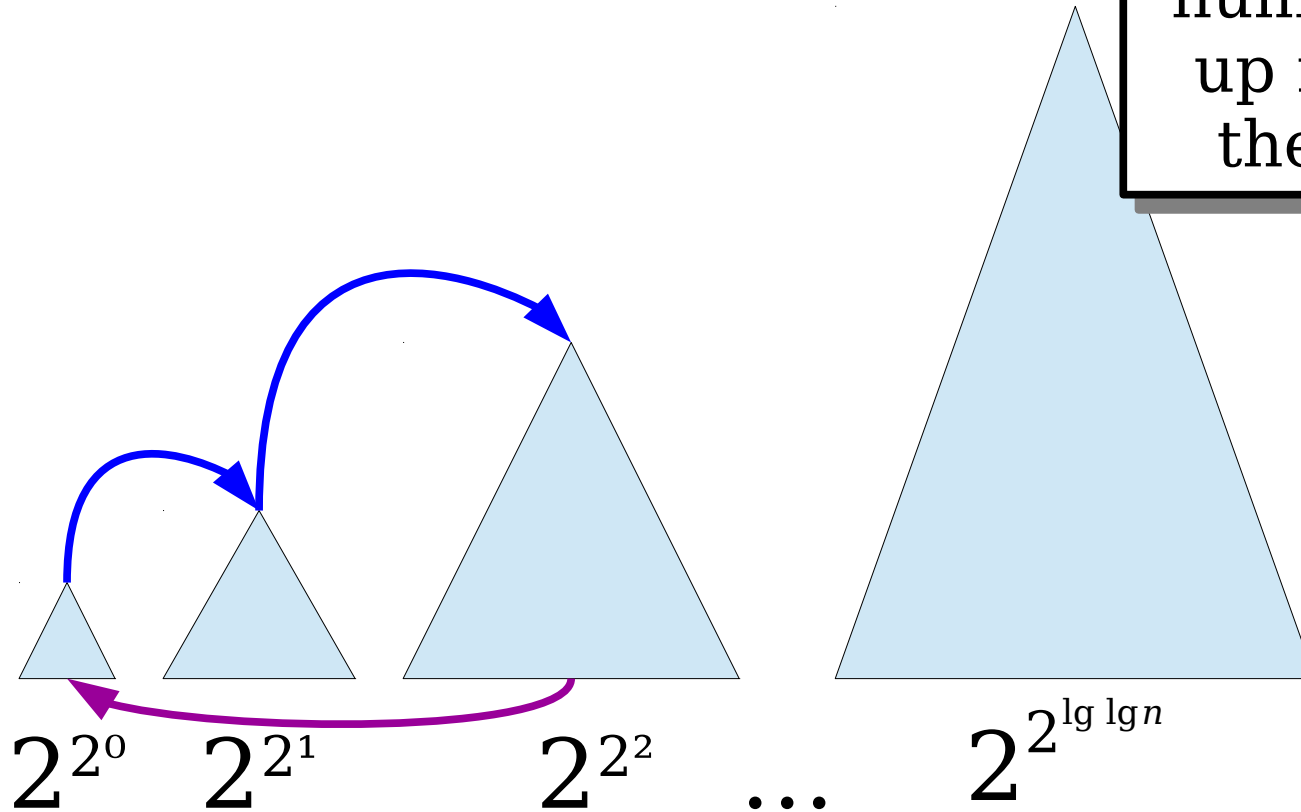
Theorem: Lookups take time $O(\log t)$, where t is the number of items accessed more recently than the key in question.
(Prove this!)



Model 4: Queries have **temporal locality**. If a key is queried, it's likely going to be queried again soon.

Intuition: Use a sequence of trees. Keep “hot” elements in the earlier trees.

A BST has the **working set property** if the (amortized) cost of looking up an element is $O(\log t)$, where t is the number of items looked up more recently than the queried element.



Model 4: Queries have **temporal locality**. If a key is queried, it's likely going to be queried again soon.

<i>Property</i>	<i>Description</i>	<i>Met by</i>
<i>Balance</i>	Lookups take time $O(\log n)$.	Traditional balanced BST
<i>Entropy</i>	Lookups take expected time $O(1 + H)$.	Weight-balanced trees
<i>Dynamic Finger</i>	Lookups take $O(\log \Delta)$. Δ measures distance.	Level-linked BST with finger
<i>Working Set</i>	Lookups take $O(\log t)$, t measures recency.	Iacono's structure

These models are in tension with one another.

Property		Met by
<i>Balance</i>	<p>All elements are equally important!</p> <p>Lookups take time</p>	Traditional balanced BST
<i>Entropy</i>	<p>No they aren't! Some get queried more!</p> <p>Lookups $\mathcal{O}(1)$</p>	Weight-balanced trees
<i>Dynamic Finger</i>	<p>And some are similar to the last query!</p> <p>Δ measures distance.</p>	Level-linked BST with finger
<i>Working Set</i>	<p>And some were queried more recently!</p> <p>Lookups take time</p>	Iacono's structure

These models are in tension with one another.

Property		Met by
<i>Balance</i>	Lookups take time $O(n)$.	Traditional balanced BST
<i>Entropy</i>	Lookups take time $O(\log n)$.	Self-balanced trees
<i>Dynamic Finger</i>	Lookups take Δ measure.	Dynamic Finger
<i>Working Set</i>	Lookups take $O(\log t)$, t measures recency.	Iacono's structure

Lookups are sampled from a fixed distribution.

What if I do a linear scan?

What if there are correlations?

These models are in tension with one another.

<i>Property</i>	<i>Description</i>	<i>Met by</i>
<i>Balance</i>	Distance in key space is what's important!	Traditional balanced BST
<i>Entropy</i>	Lookups take expected time $O(\log t)$	Weight-balanced trees
<i>Dynamic Finger</i>	Lookups Δ measure distance.	linked BST with finger
<i>Working Set</i>	Lookups take $O(\log t)$, t measures recency.	Iacono's structure

These models are in tension with one another.

<i>Property</i>	<i>Description</i>	<i>Met by</i>
<i>Balance</i>	Lookups take time $O(\log n)$.	Traditional balanced BST
<i>Entropy</i>	Lookups take expected time $O(1 + H)$.	Weight-balanced trees
<i>Dynamic Finger</i>	Lookups take $O(\log \Delta)$. Δ measures distance.	Level-linked BST with finger
<i>Working Set</i>	Lookups take $O(\log t)$, t measures recency.	Iacono's structure

Is there a single BST that guarantees all of these properties?

<i>Property</i>	<i>Description</i>	<i>Met by</i>
<i>Balance</i>	Lookups take time $O(\log n)$.	<i>Splay tree</i>
<i>Entropy</i>	Lookups take expected time $O(1 + H)$.	<i>Splay tree</i>
<i>Dynamic Finger</i>	Lookups take $O(\log \Delta)$. Δ measures distance.	<i>Splay tree</i>
<i>Working Set</i>	Lookups take $O(\log t)$, t measures recency.	<i>Splay tree</i>

Yes!

Time-Out for Announcements!

CS Research Course

- Michael Bernstein is offering a new course in CS research methodology next fall. ***Hooray!***
- The course, ***CS197***, has limited enrollment. There's an application form online:

<https://forms.gle/CTByQ4cayRqy4VNr5>

- There's an info session next Tuesday, May 14th at 5PM in Gates 219. Feel free to stop by if you're curious!
- ***This is an incredible opportunity. You should really seriously look into this!***

Problem Set Four

- Problem Set Three was due today at 2:30PM.
 - Want to use a late period? Submit it by this Thursday at 2:30PM.
- Problem Set Four goes out today. It's due on Tuesday, May 21st at 2:30PM.
 - Play around with amortized efficiency and the data structures we've covered so far!

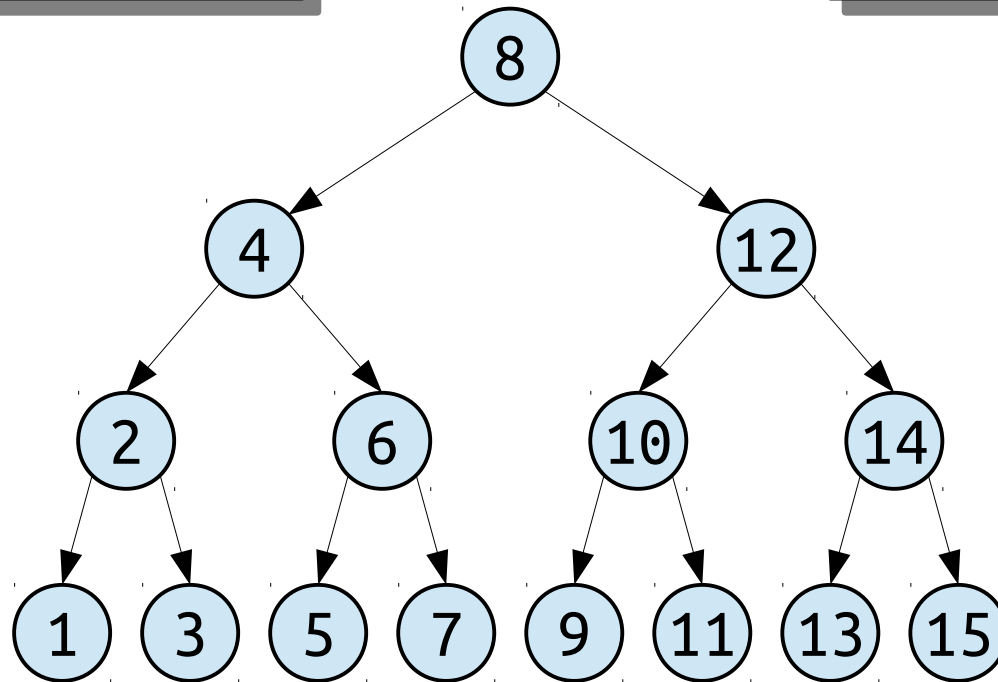
Project Checkpoints

- Project topics are assigned! Let us know if you didn't get an email.
- Your next deliverable is the project checkpoint, which is due on **Thursday, May 16th**. No late periods may be used.
- What you need to do between now and then:
 - **Become an expert**. You should aim to have a solid command of your topic by this point. Read everything you can on the subject. Tinker around with the topic and see what makes it tick.
 - **Answer our questions**. You received two questions about your topic over email. Come up with the best answers to those questions that you can.
 - **Decide on your "interesting" component**. Once you're an expert on the topic and have answered our questions, you're in a good spot to determine what to do next. Really give this some thought.
- Your deliverable is a (2,000ish-word) progress report detailing what you've learned about your data structure, your answers to our questions, and your proposed "interesting" component.
- Full details are in Handout 09 on the final project.

Back to CS166!

Idea 1: Get the working set property by choosing a clever BST shape.

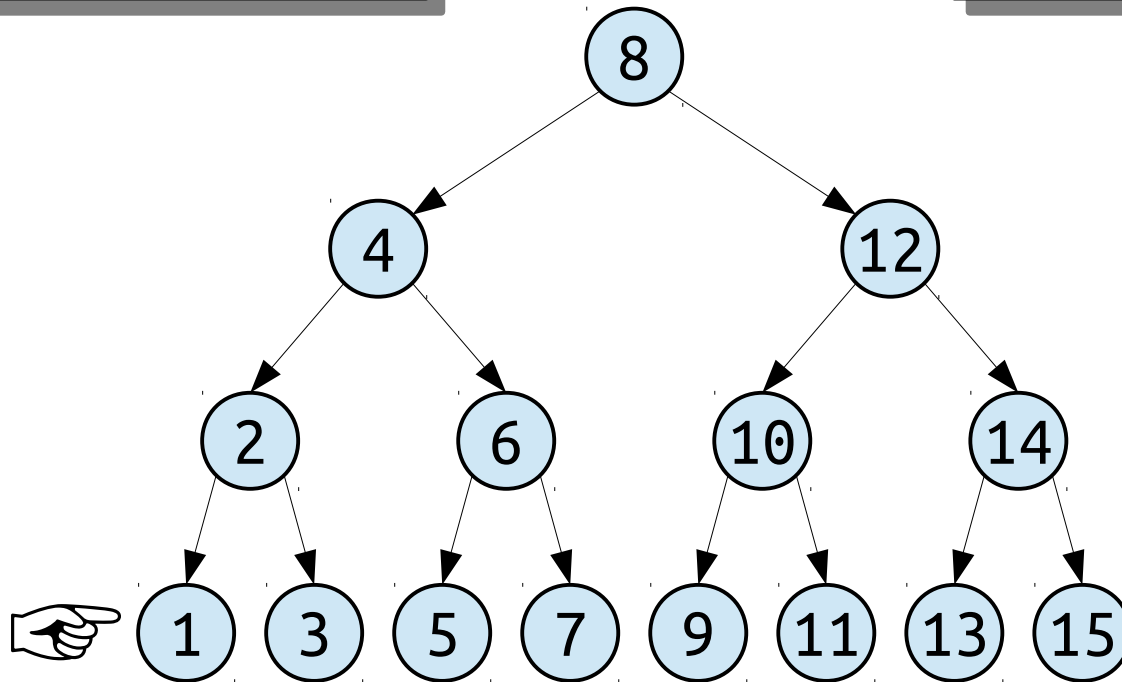
Problem: We can always pick a set of hot elements deep in the tree.



How do we build a BST with the working set property?

Idea 2: Get the working set property by adding a finger into our BST.

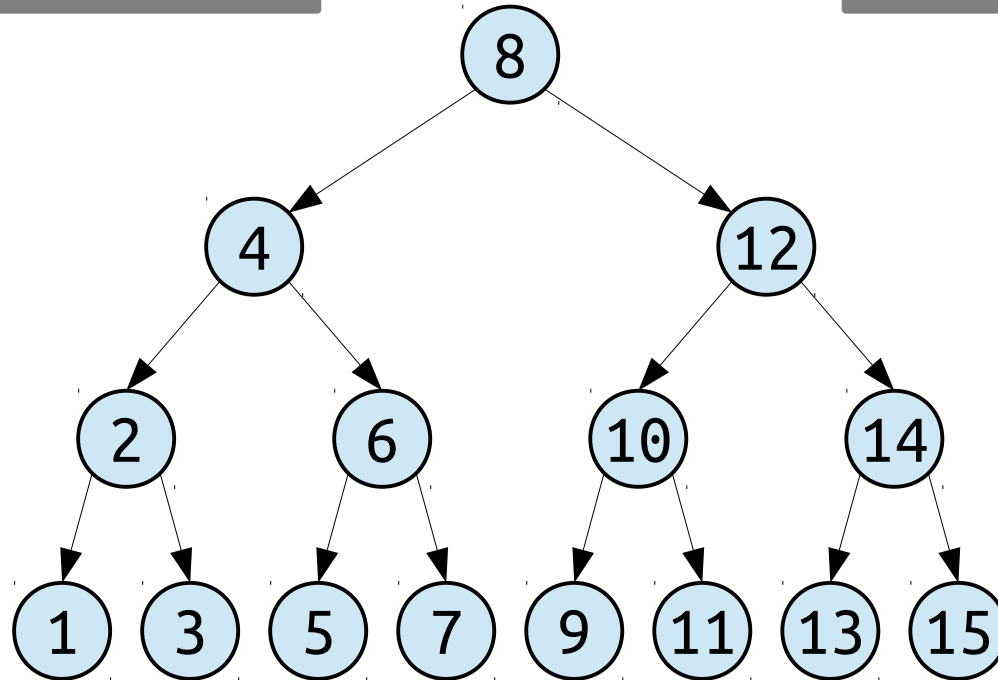
Problem: What if those keys aren't near each other in the BST?



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

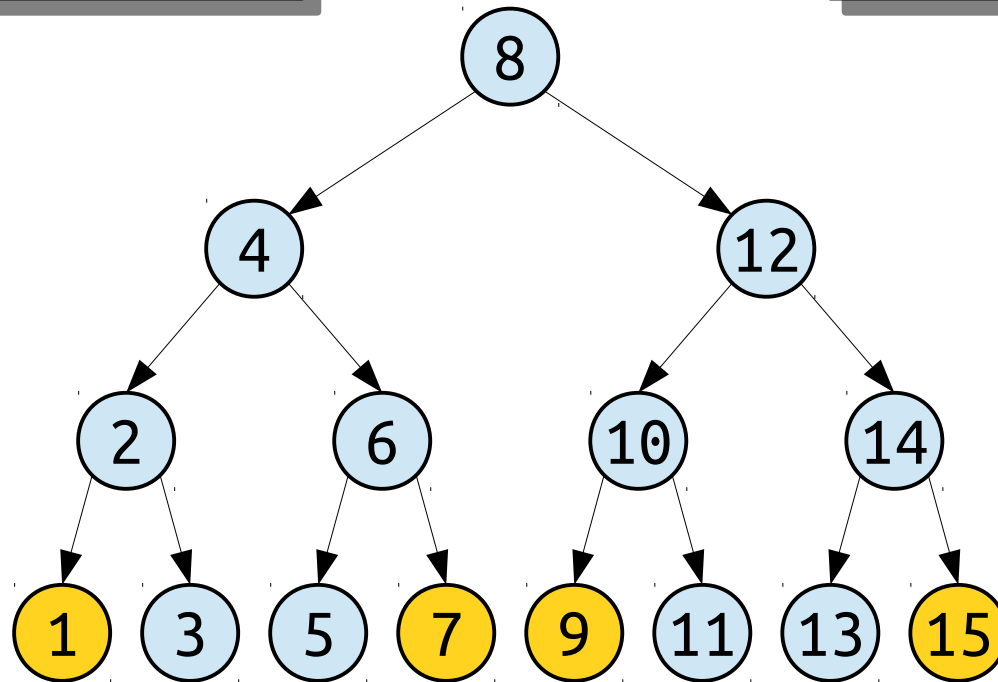
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

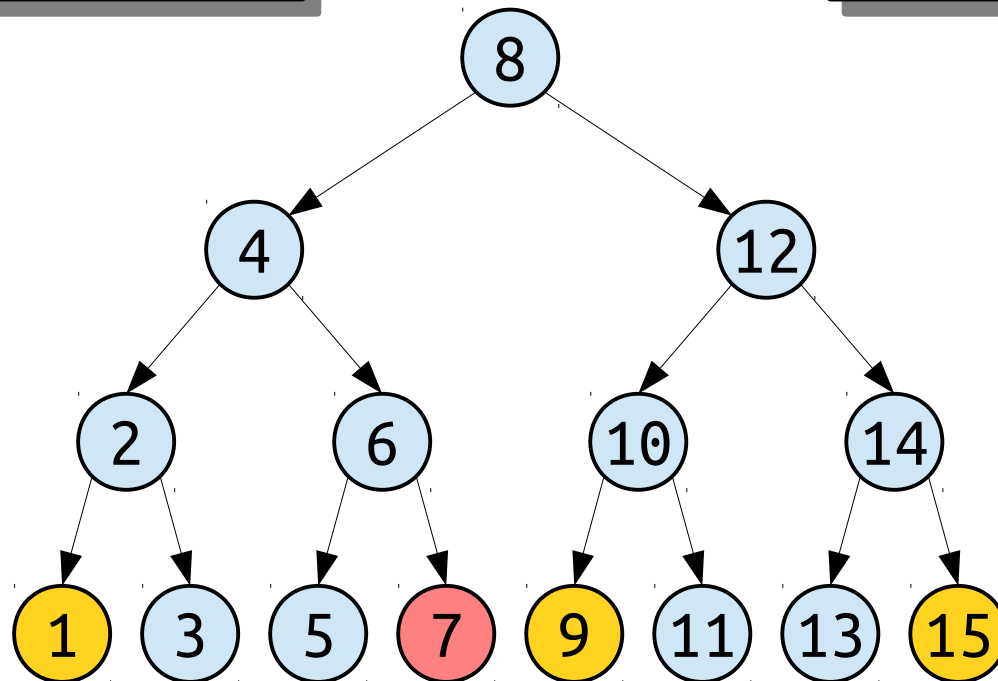
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

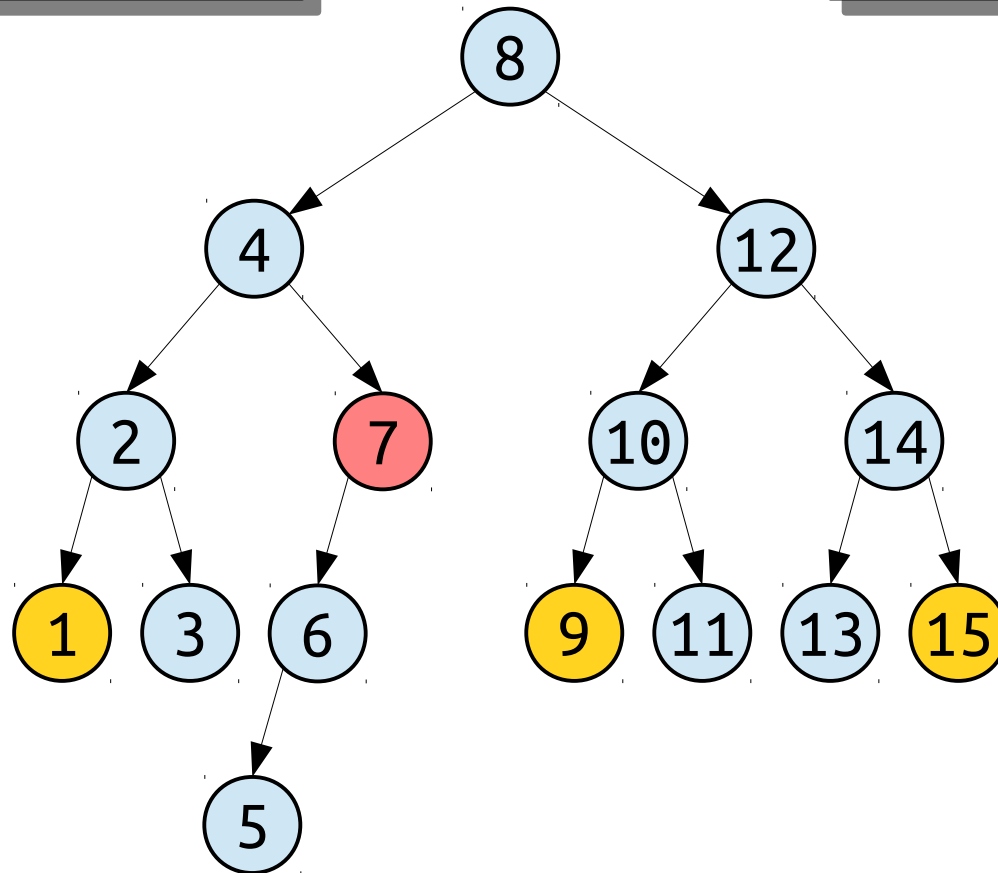
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

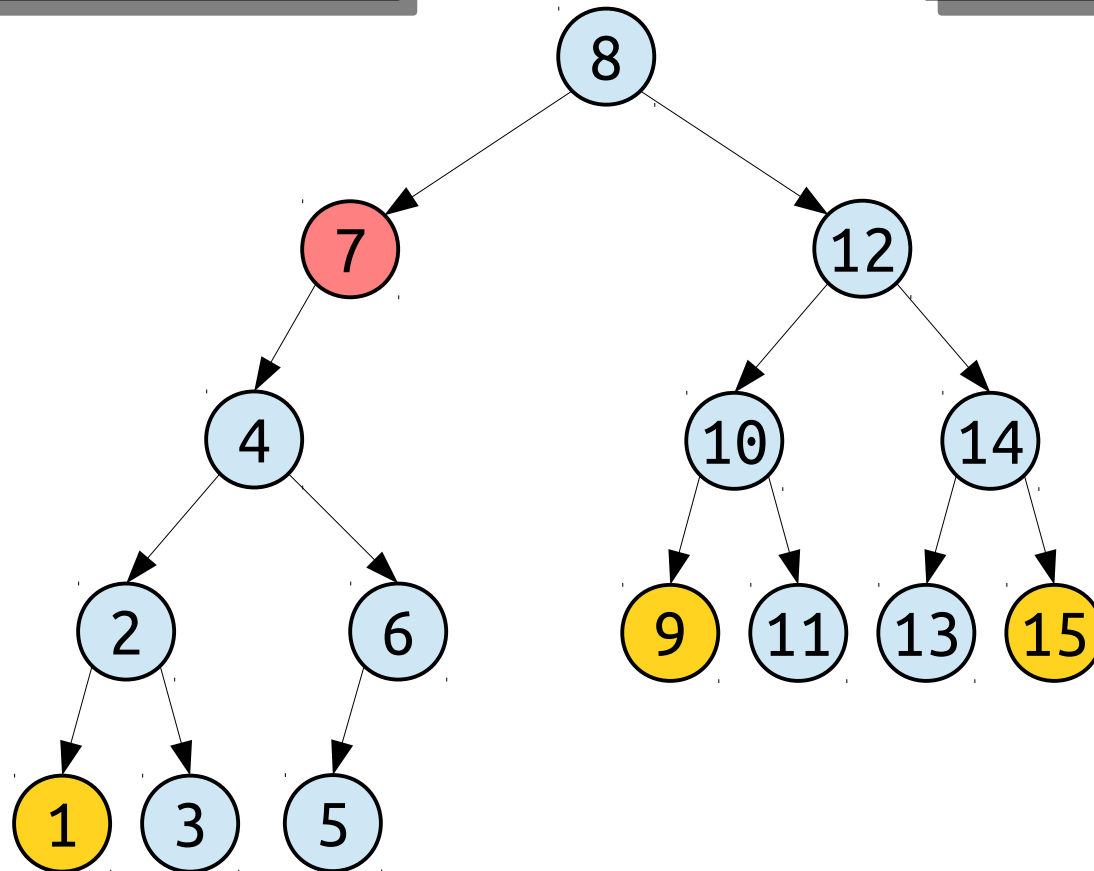
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

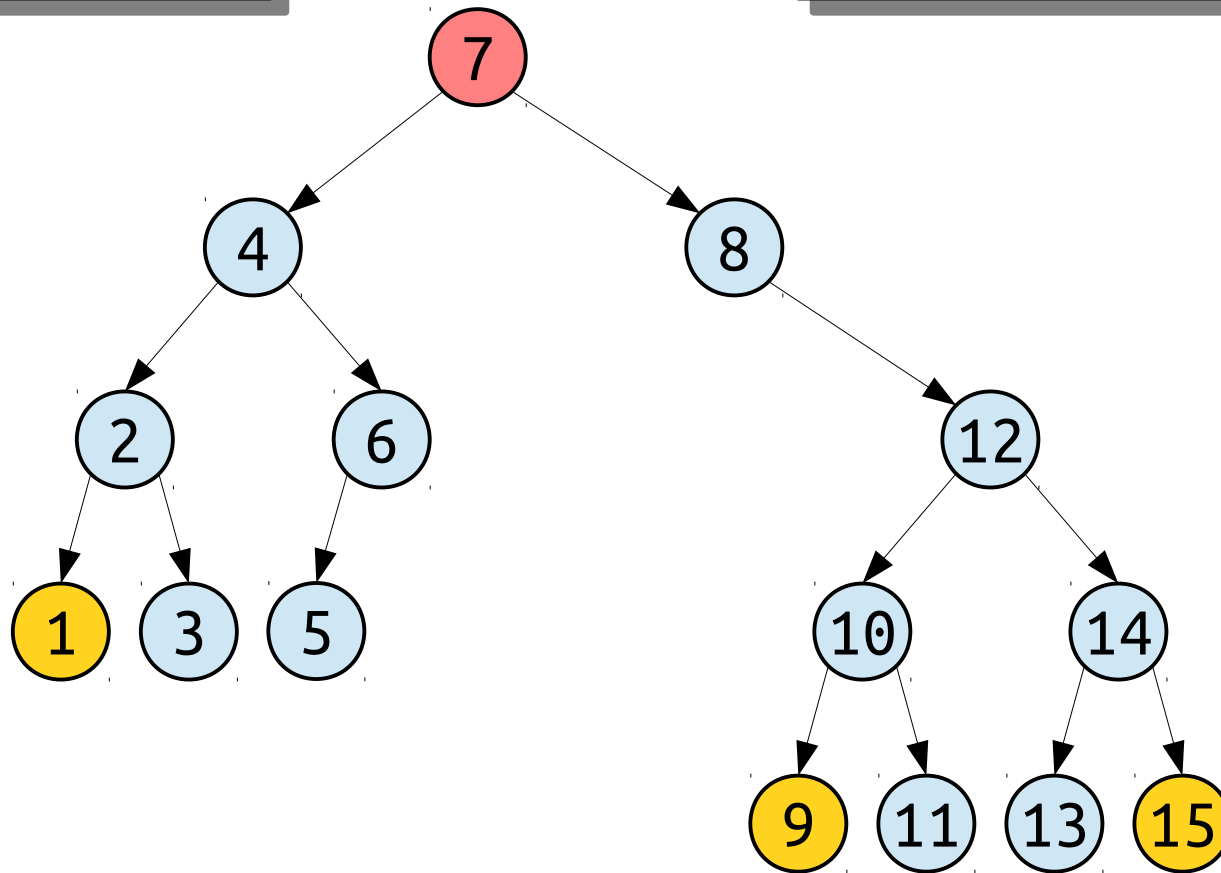
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

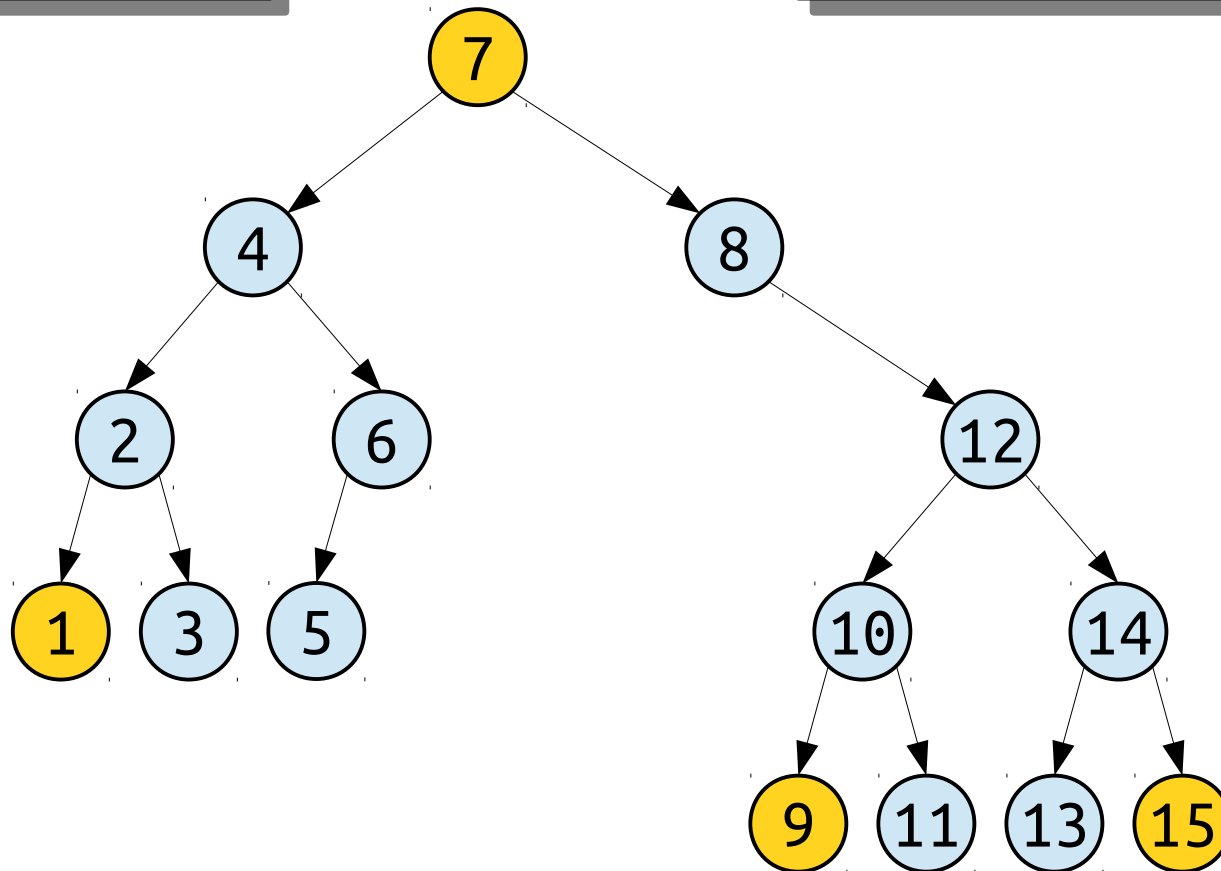
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

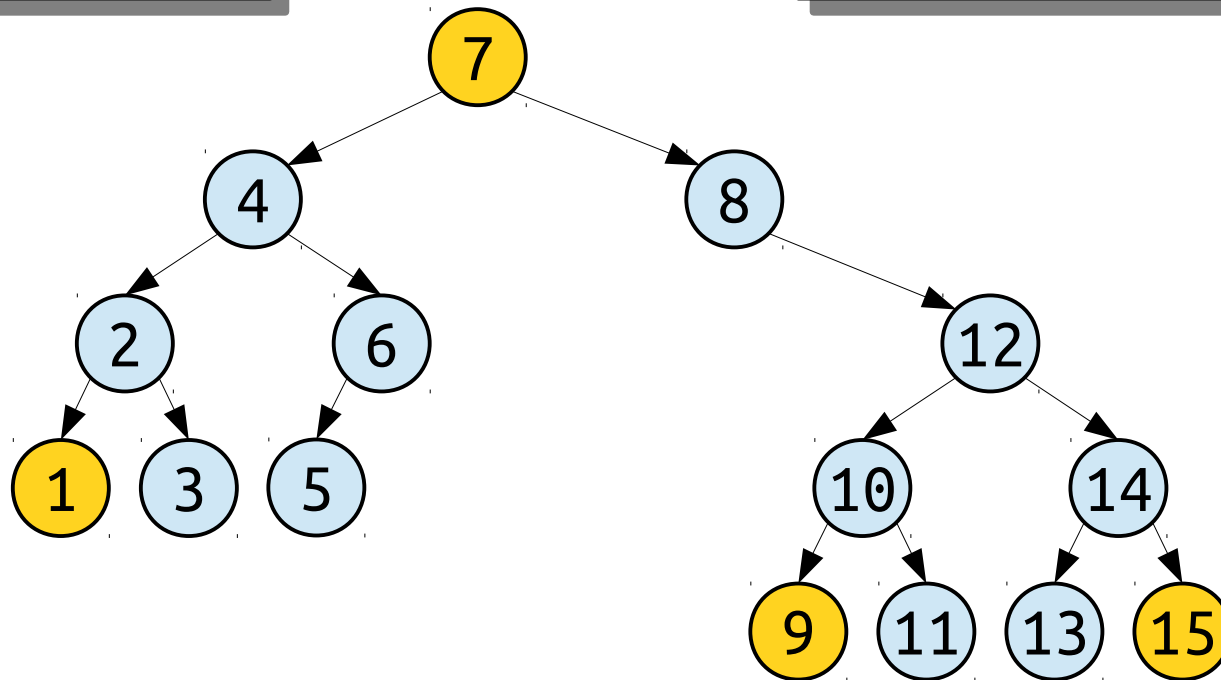
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

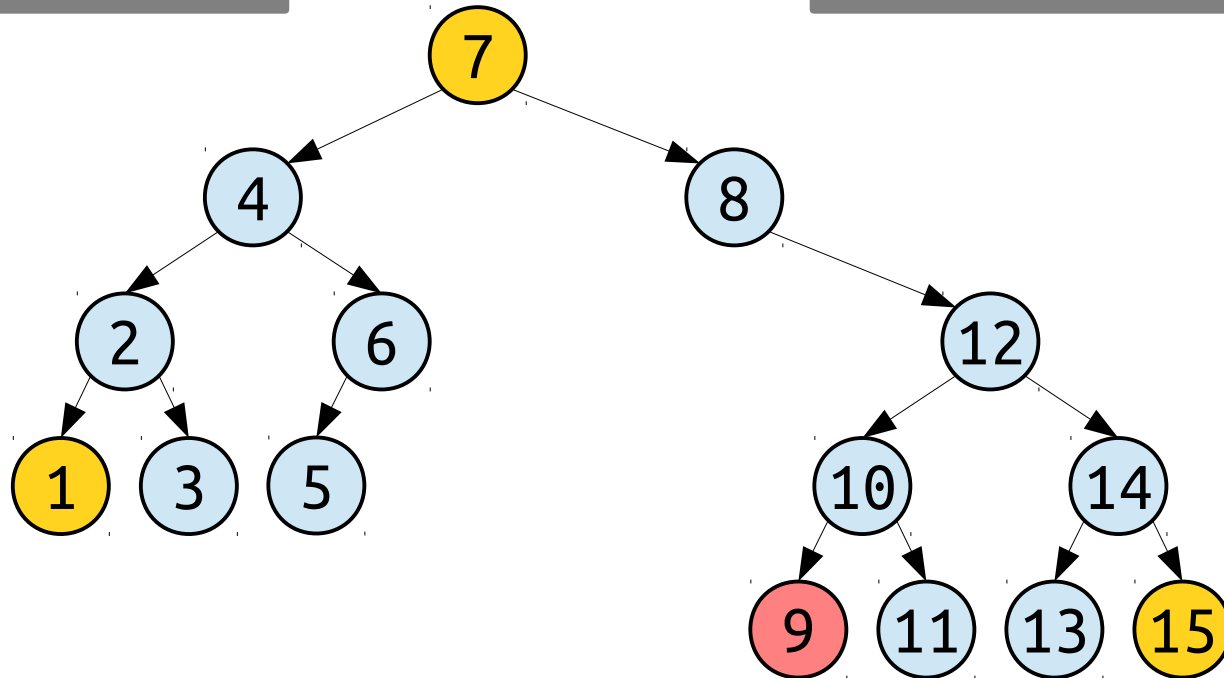
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

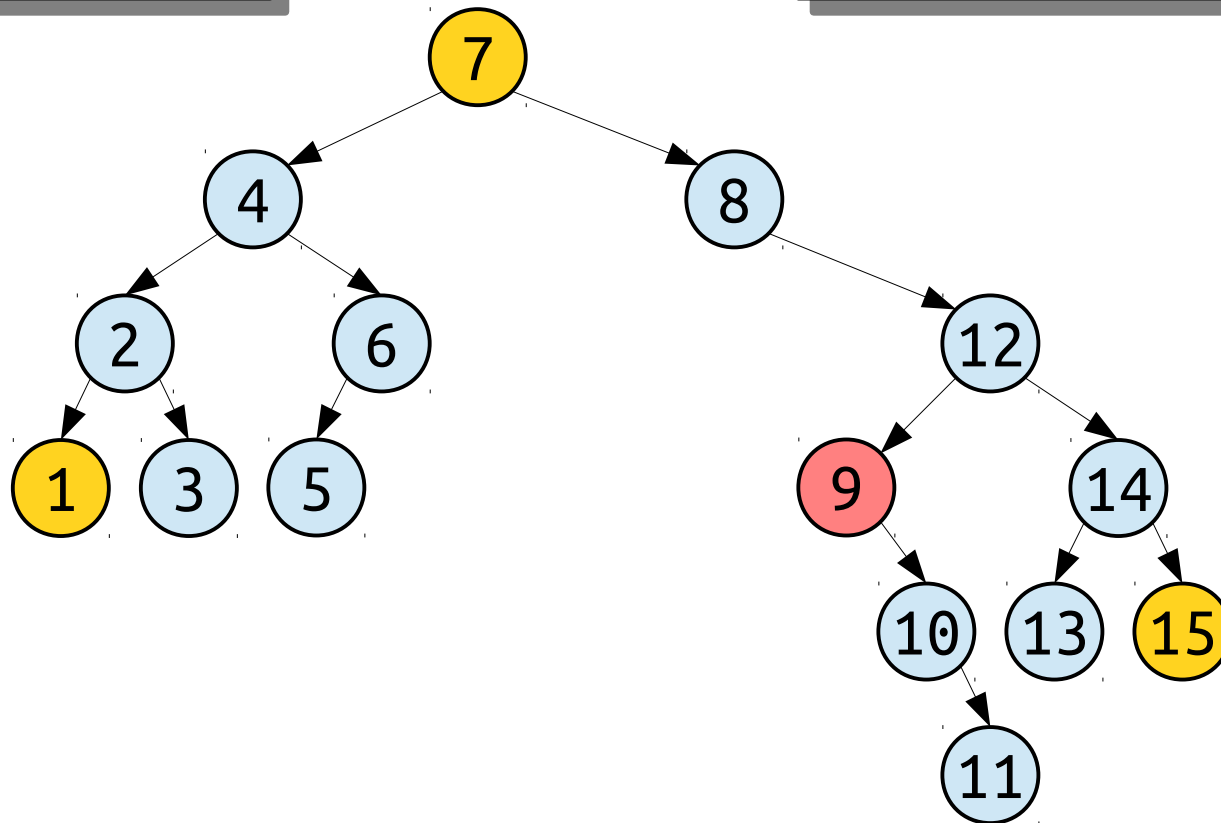
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

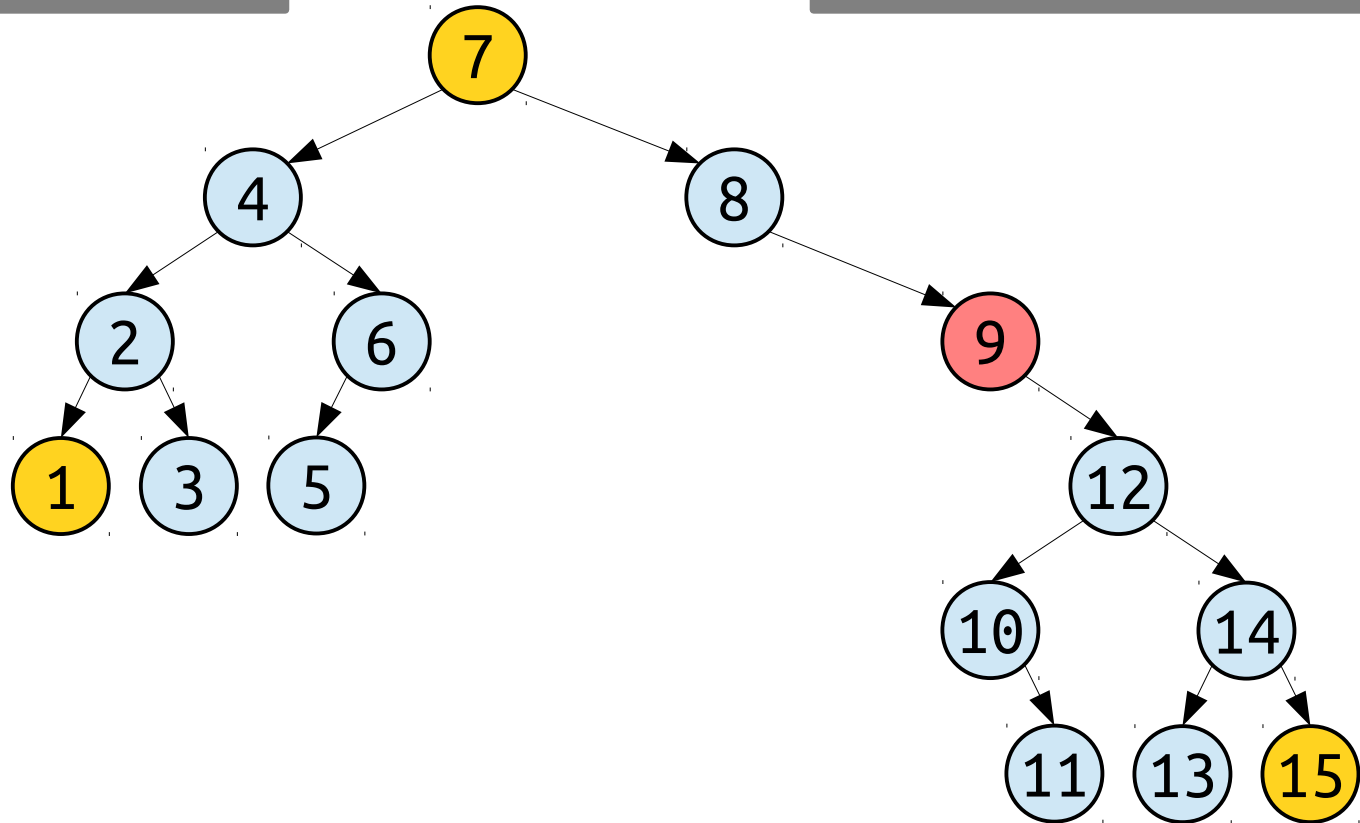
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

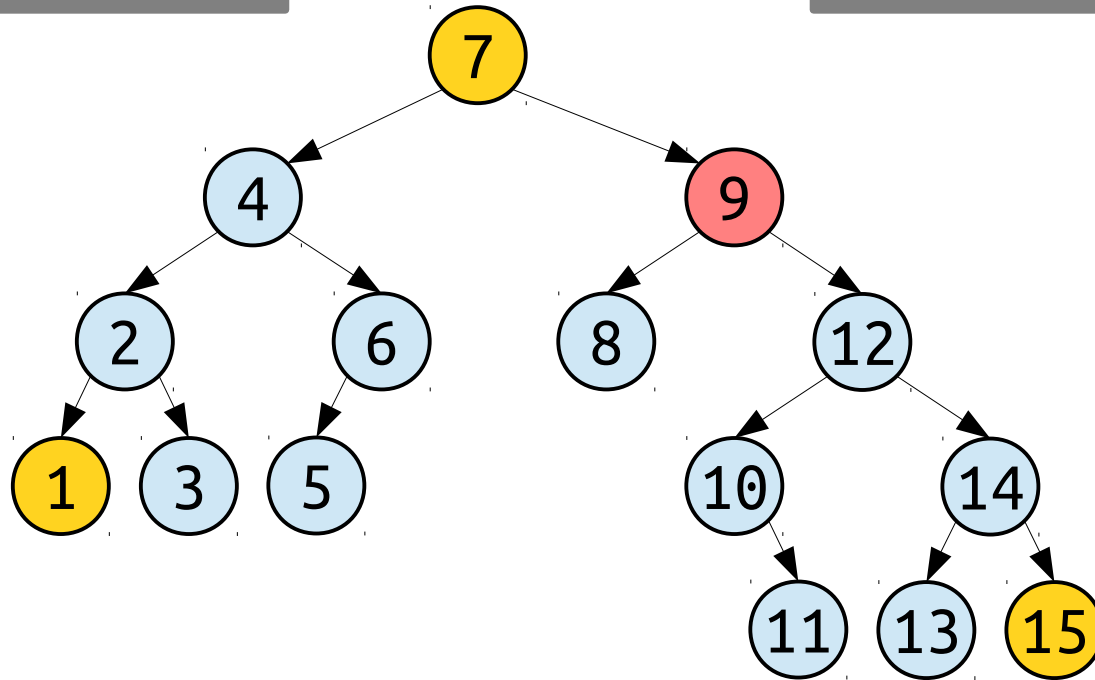
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

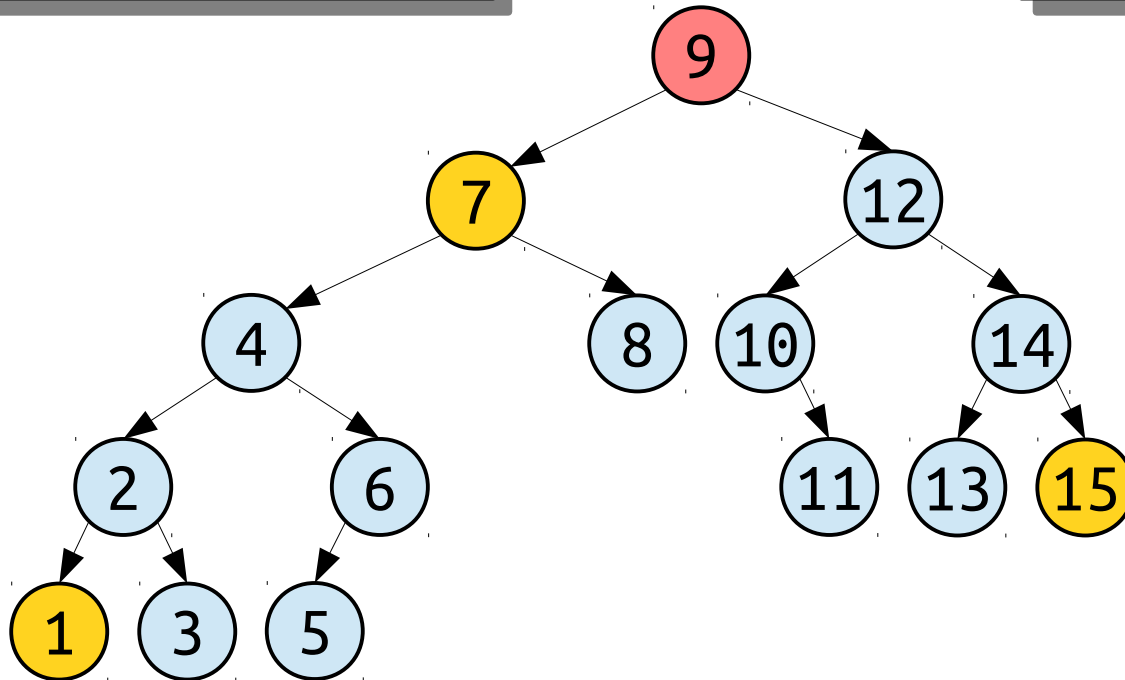
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

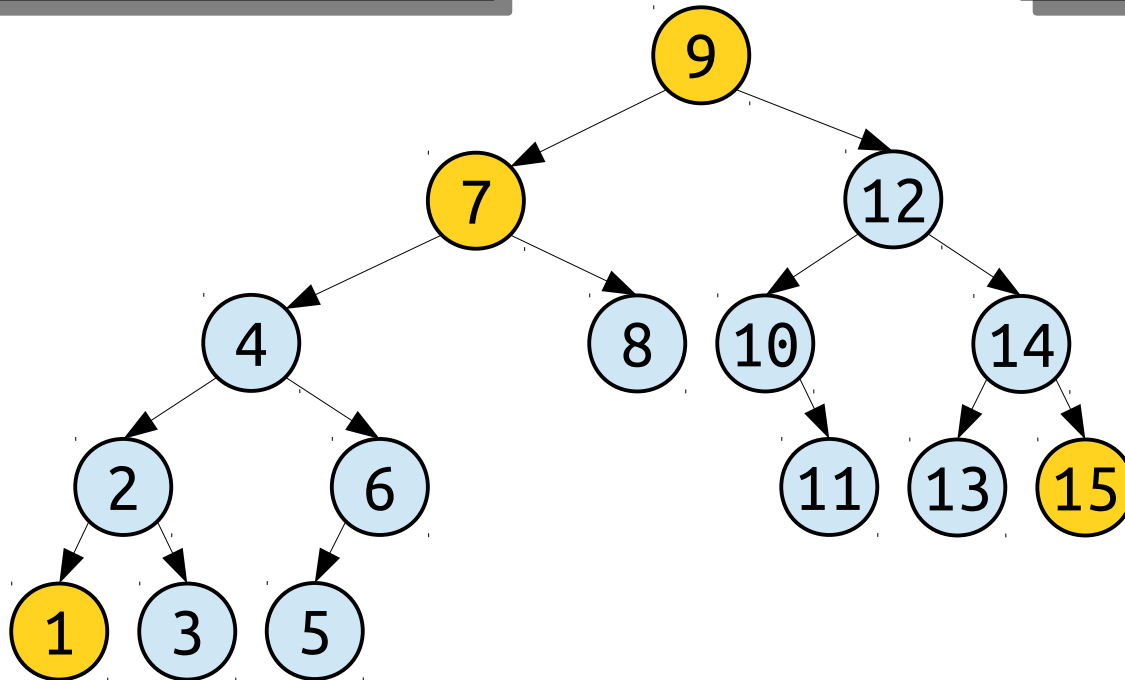
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

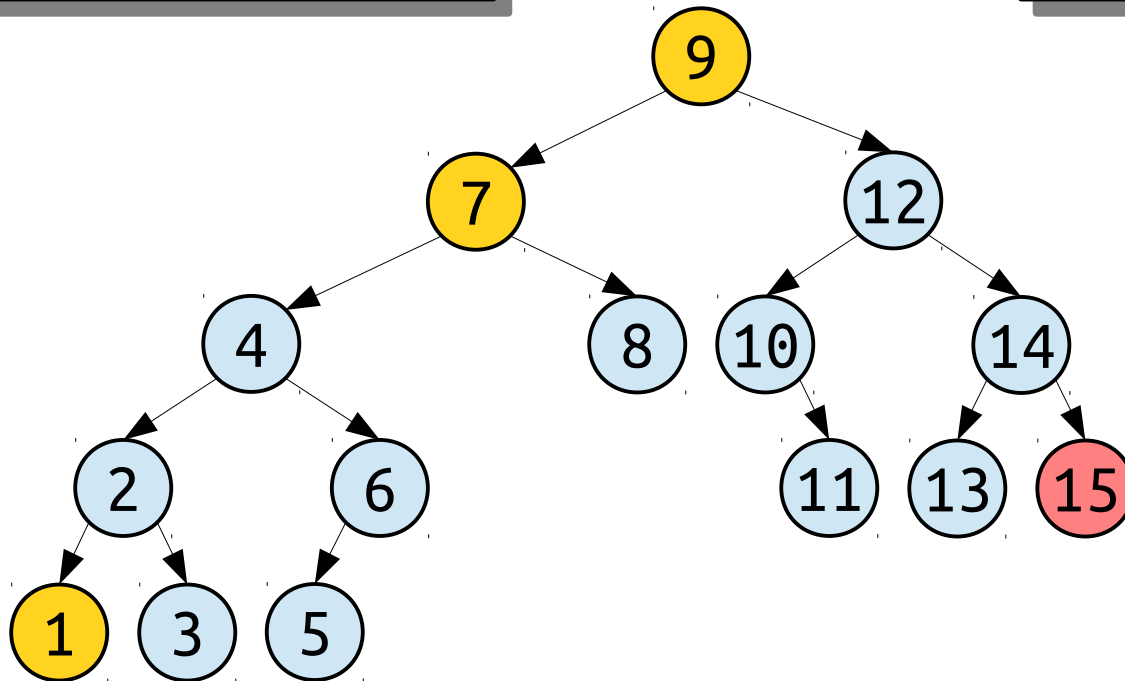
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

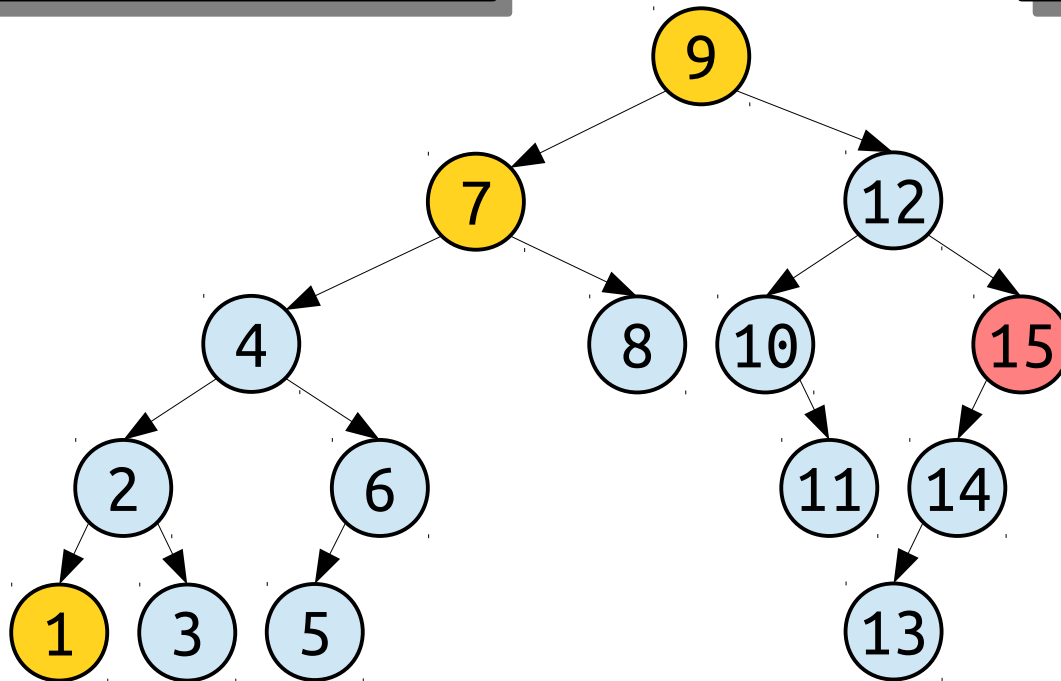
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

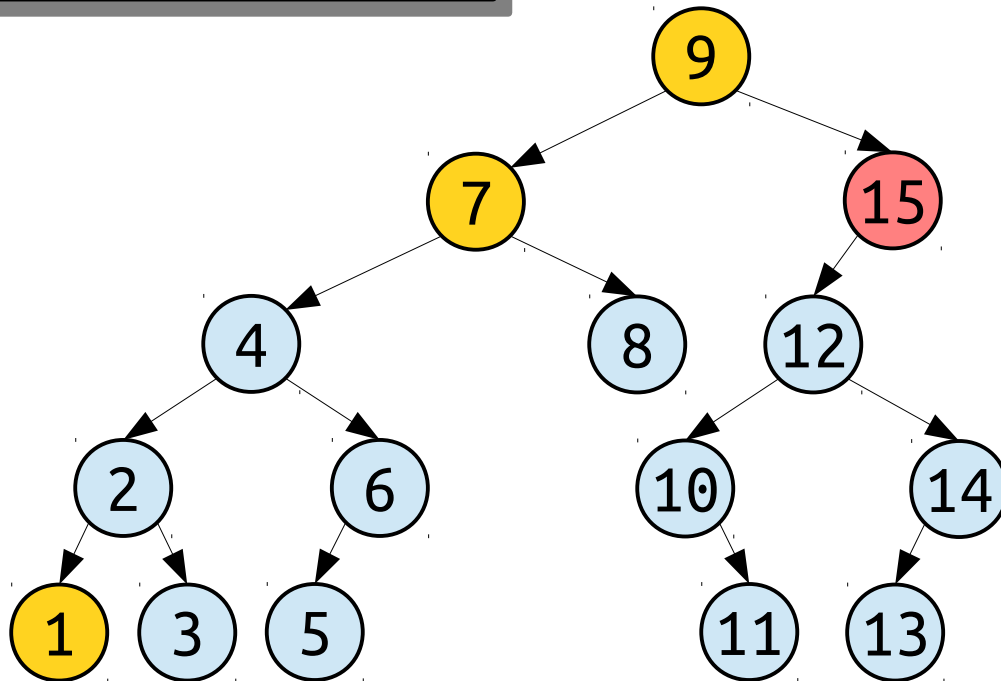
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

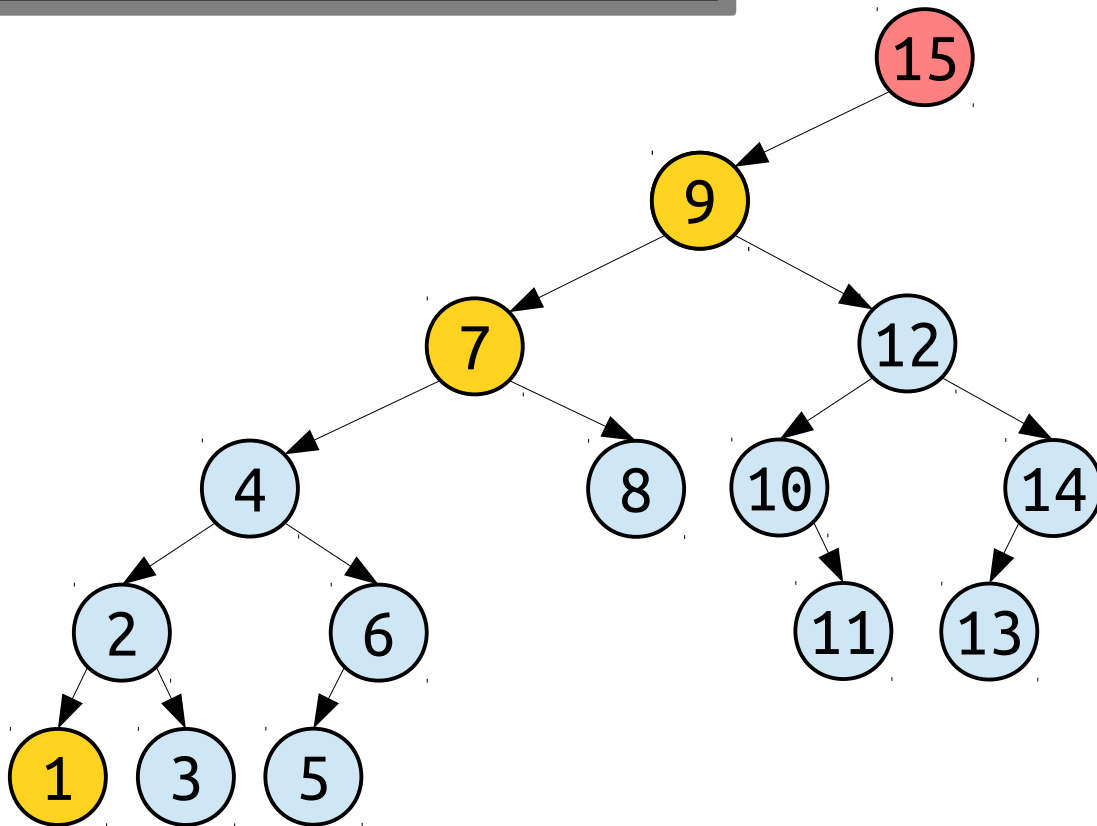
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

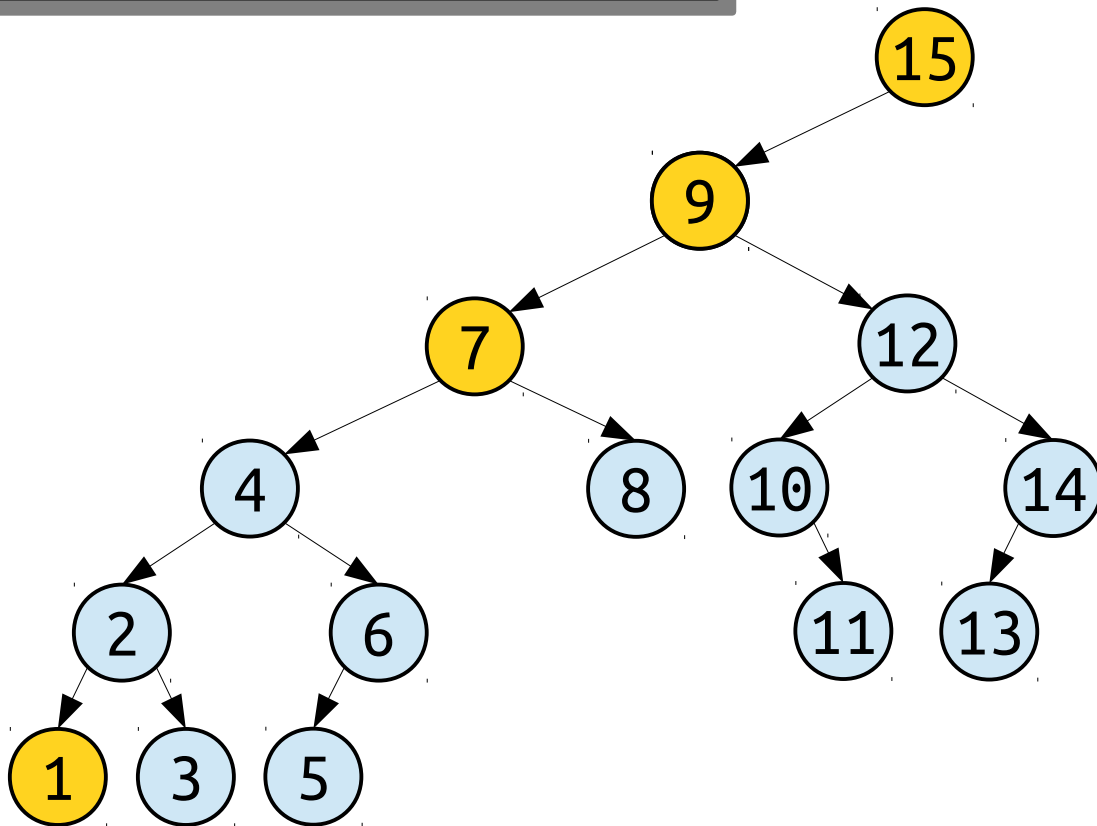
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

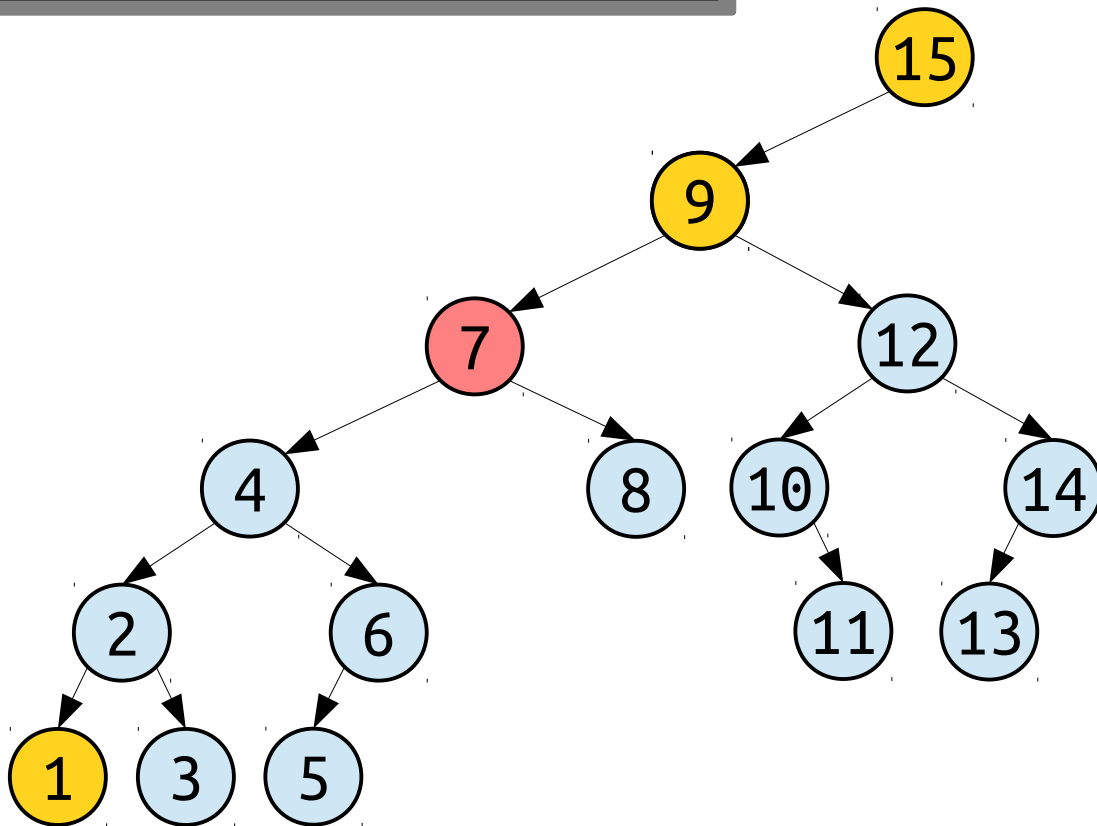
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

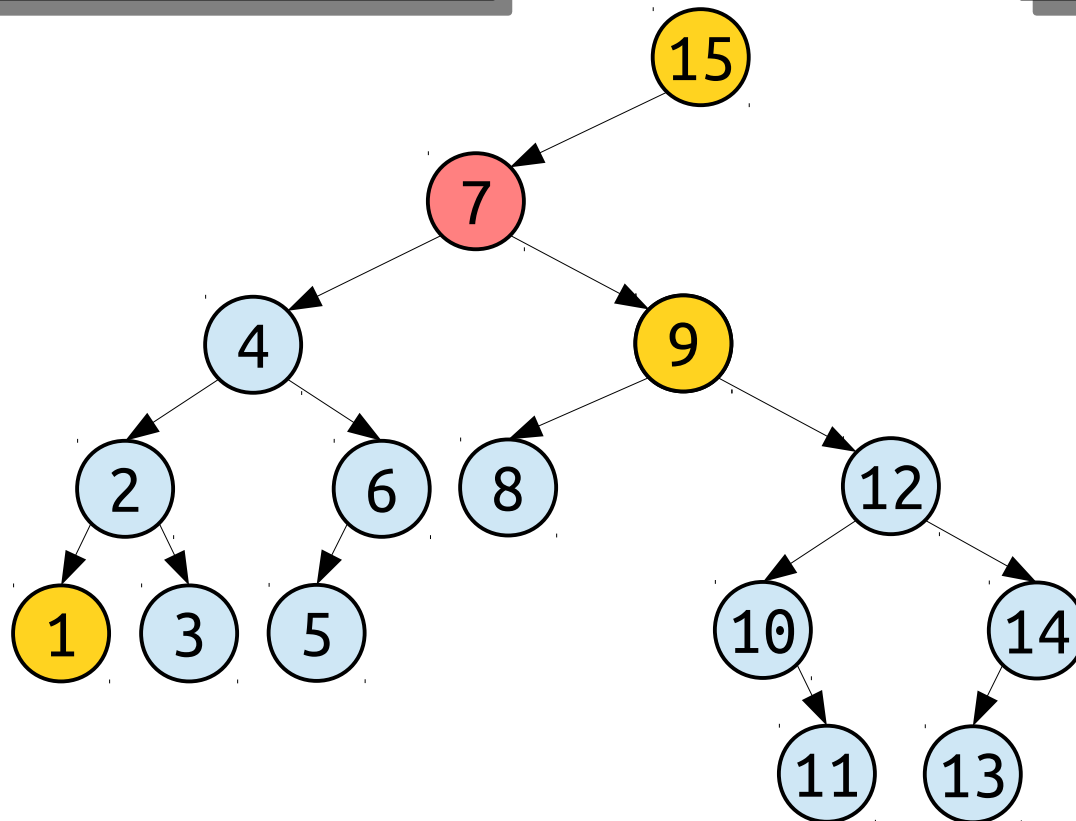
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

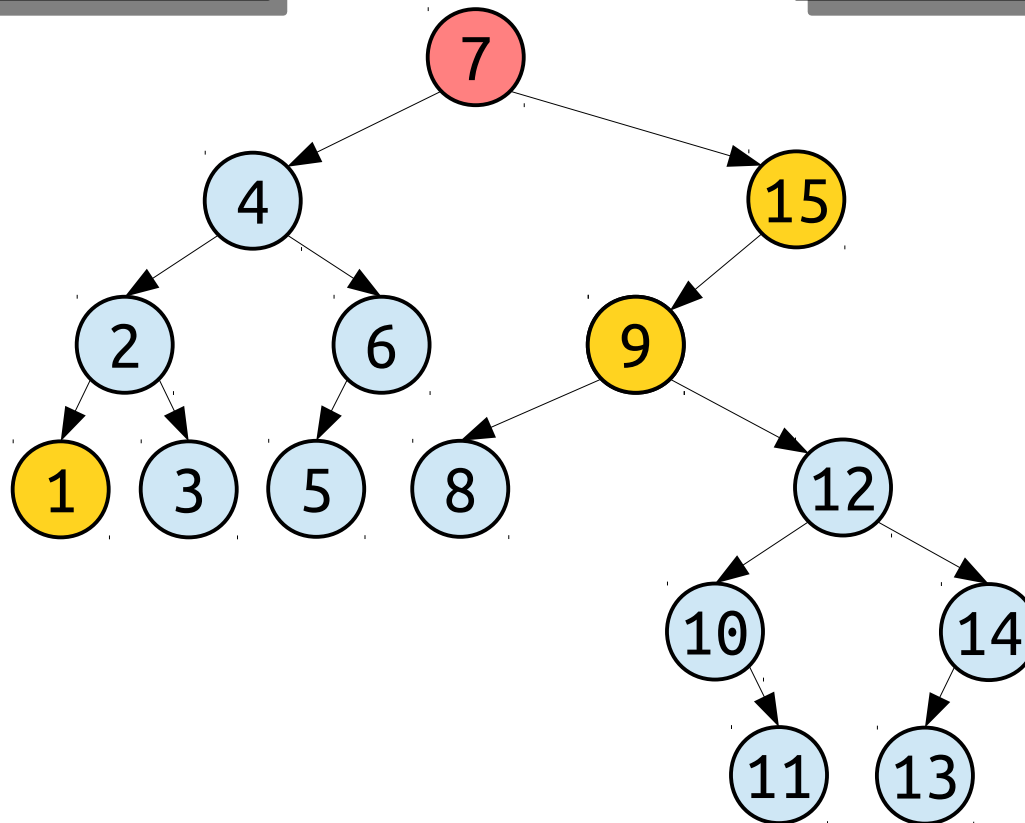
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

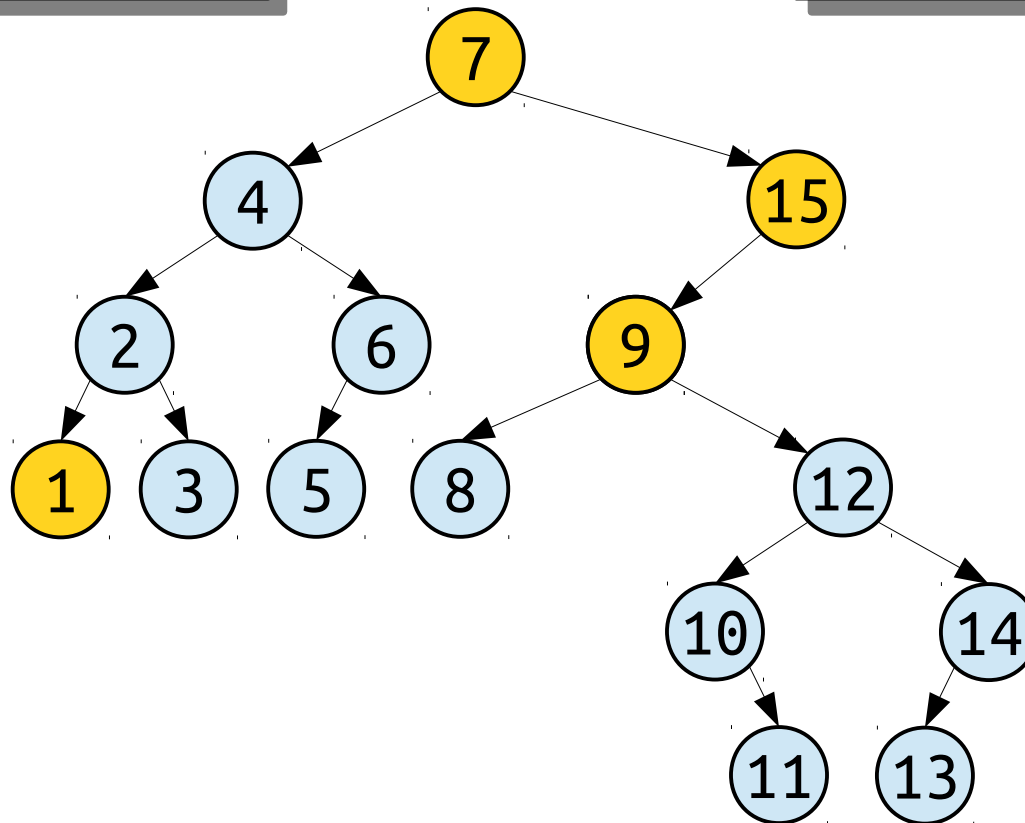
Strategy: After querying a node, rotate it up to the root of the tree.



How do we build a BST with the working set property?

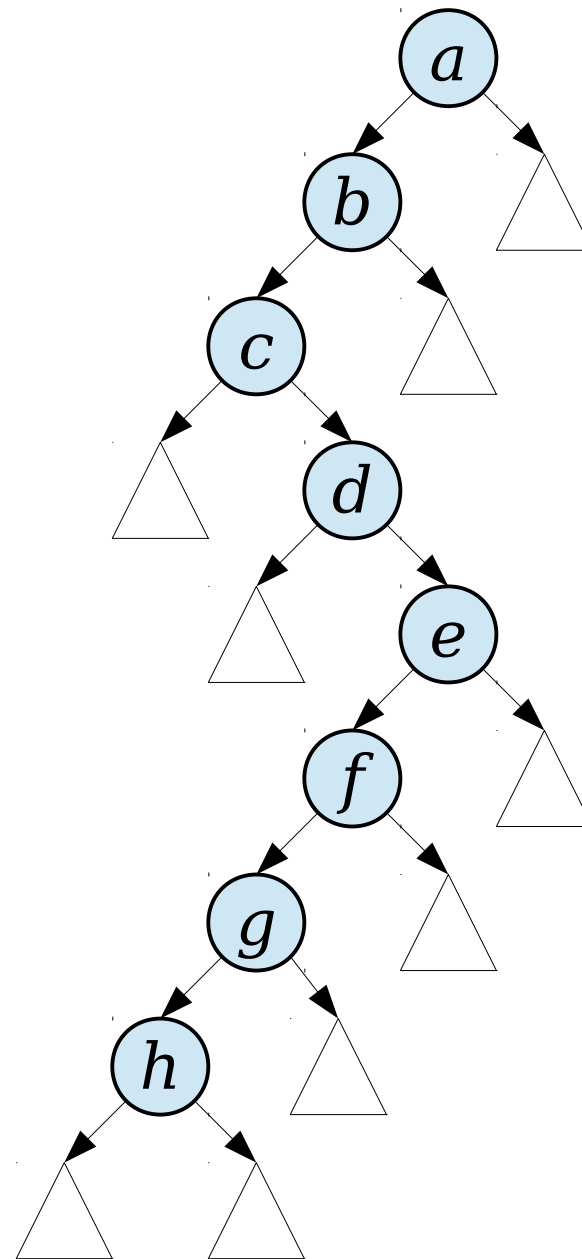
Idea 3: Get the working set property by moving nodes around the BST.

Strategy: After querying a node, rotate it up to the root of the tree.



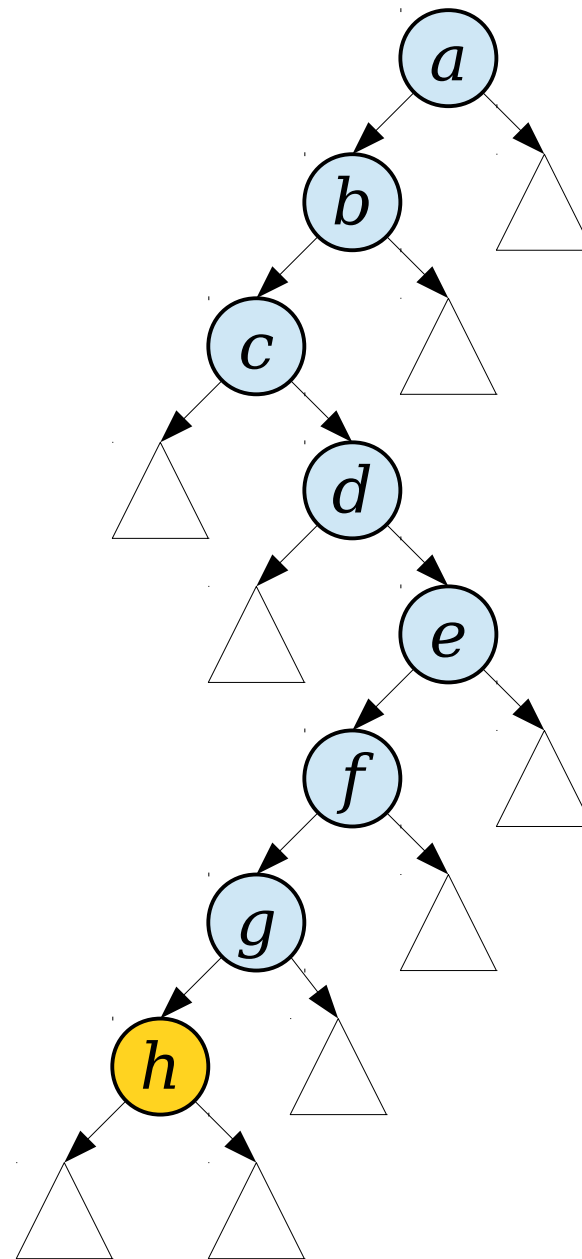
How do we build a BST with the working set property?

We have a *mechanical* description of how we reshape the tree. Can we get an *operational* description?



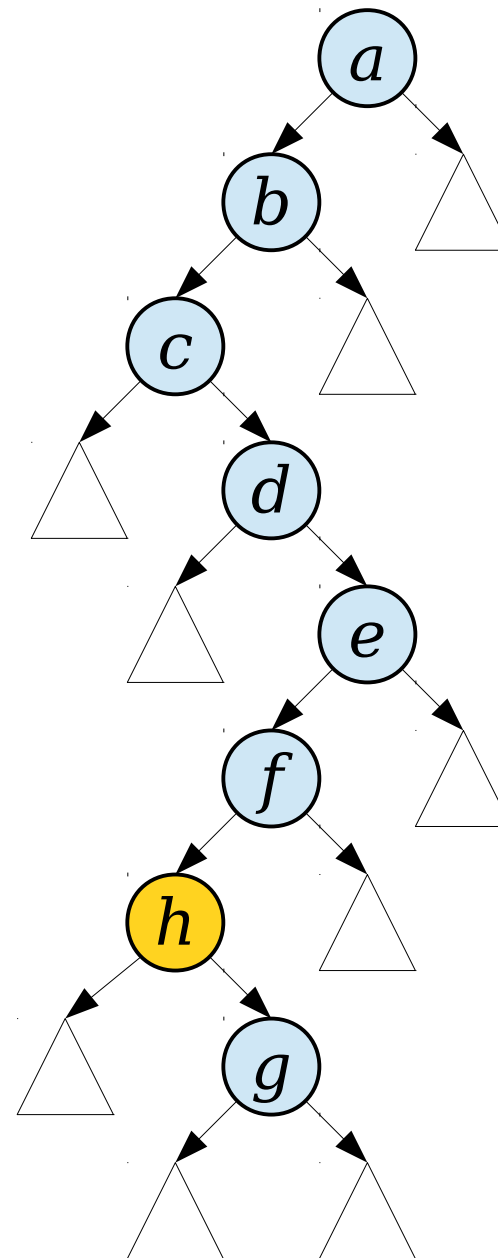
Question: Does rotating each accessed key to the root guarantee good overall performance?

We have a ***mechanical*** description of how we reshape the tree. Can we get an ***operational*** description?



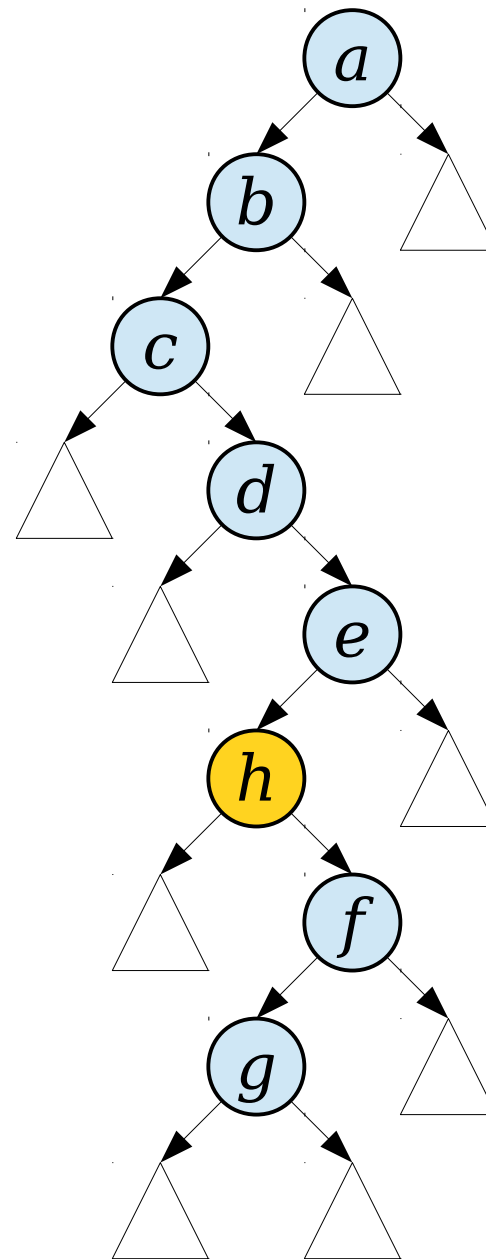
Question: Does rotating each accessed key to the root guarantee good overall performance?

We have a *mechanical* description of how we reshape the tree. Can we get an *operational* description?



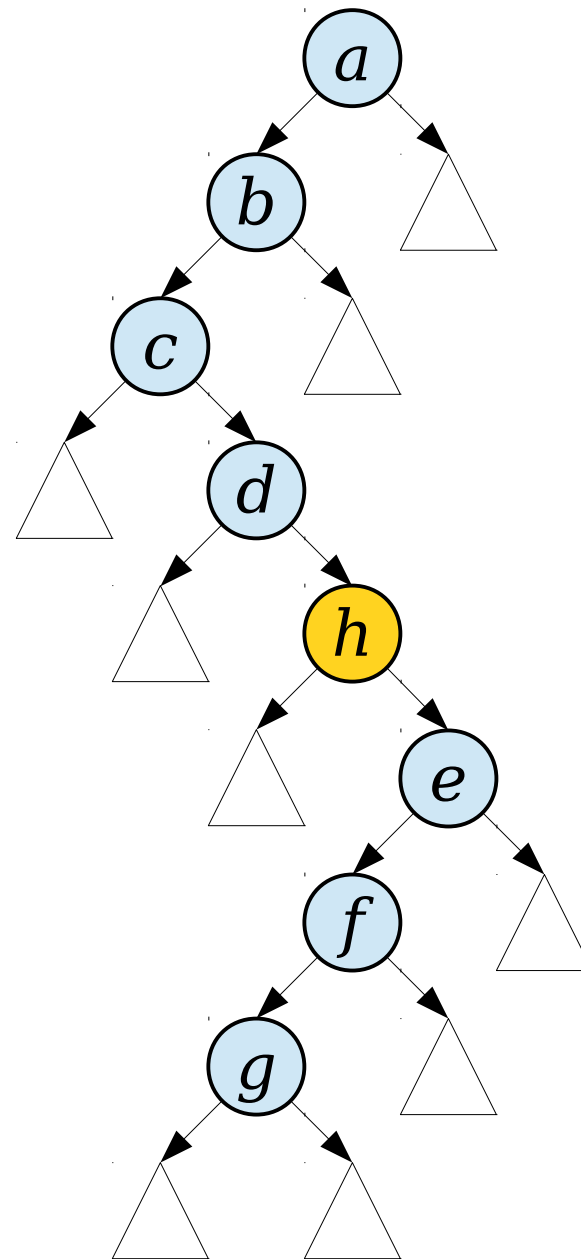
Question: Does rotating each accessed key to the root guarantee good overall performance?

We have a *mechanical* description of how we reshape the tree. Can we get an *operational* description?



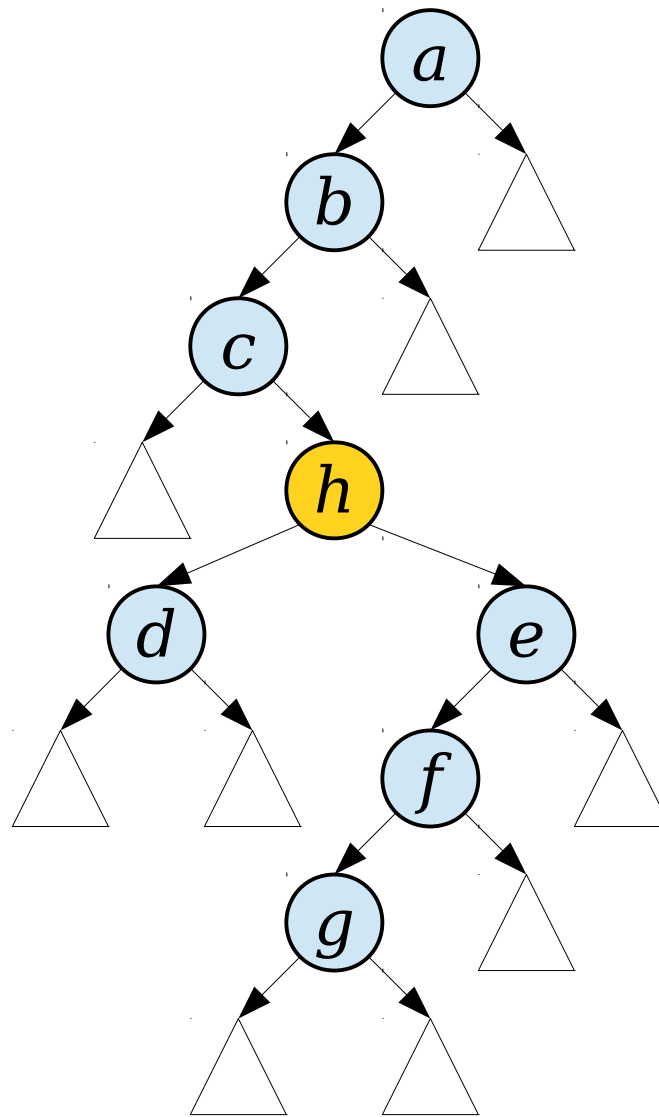
Question: Does rotating each accessed key to the root guarantee good overall performance?

We have a ***mechanical*** description of how we reshape the tree. Can we get an ***operational*** description?



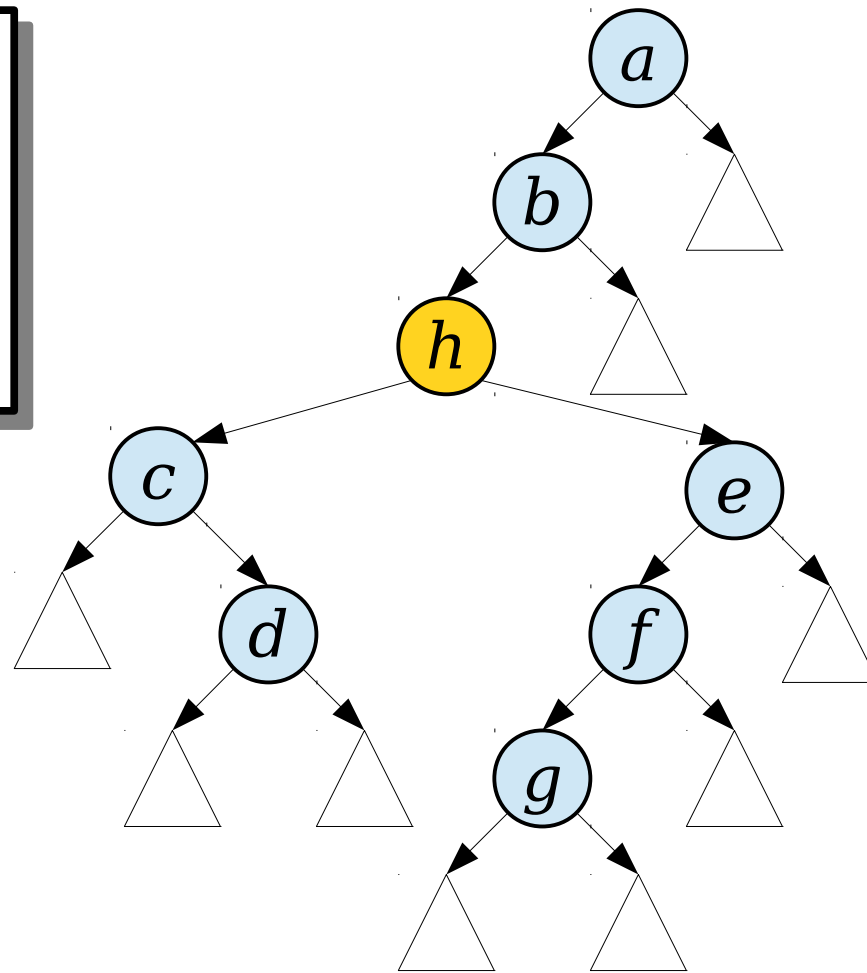
Question: Does rotating each accessed key to the root guarantee good overall performance?

We have a *mechanical* description of how we reshape the tree. Can we get an *operational* description?



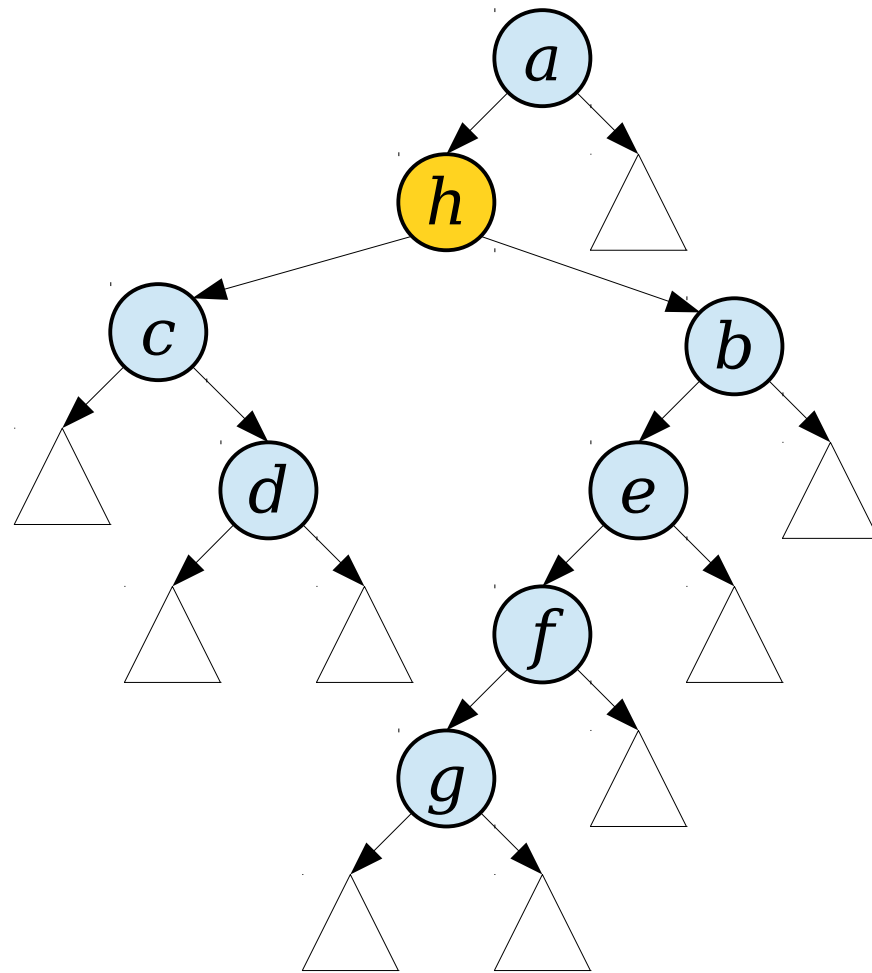
Question: Does rotating each accessed key to the root guarantee good overall performance?

We have a *mechanical* description of how we reshape the tree. Can we get an *operational* description?



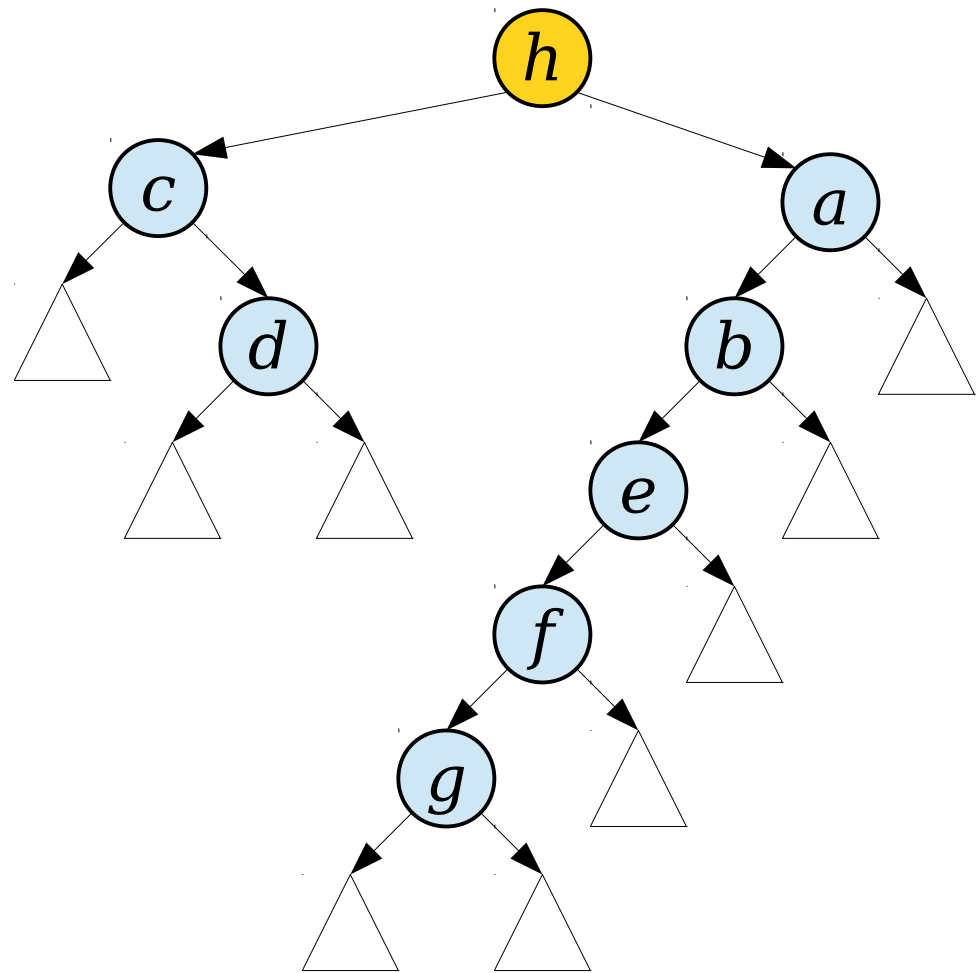
Question: Does rotating each accessed key to the root guarantee good overall performance?

We have a *mechanical* description of how we reshape the tree. Can we get an *operational* description?



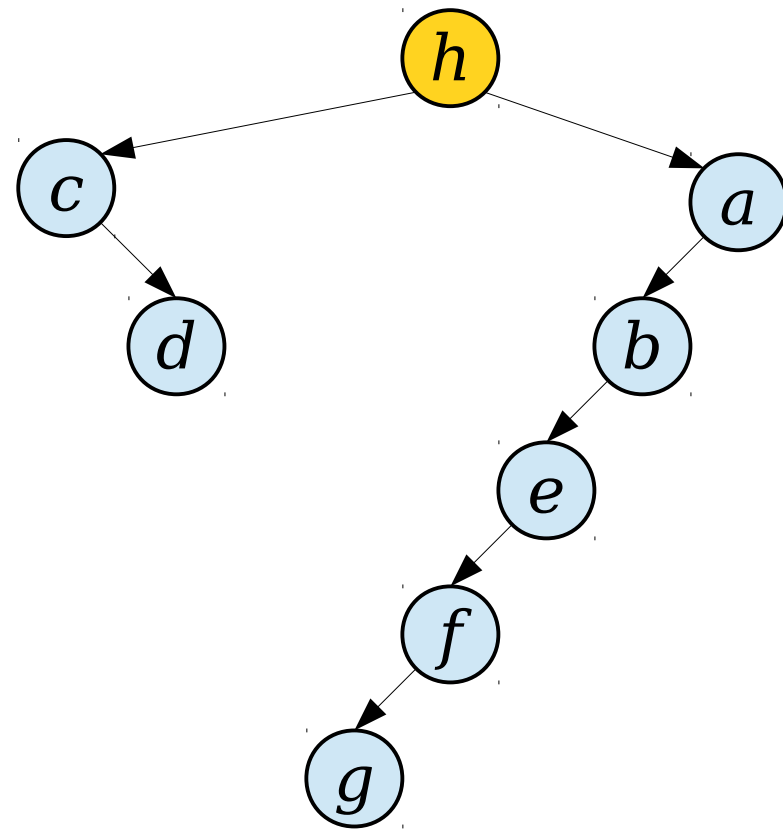
Question: Does rotating each accessed key to the root guarantee good overall performance?

We have a *mechanical* description of how we reshape the tree. Can we get an *operational* description?



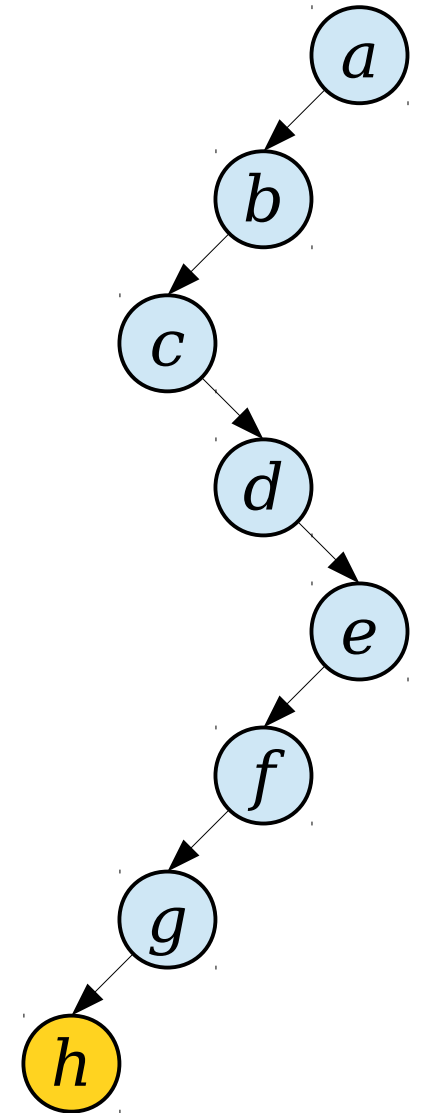
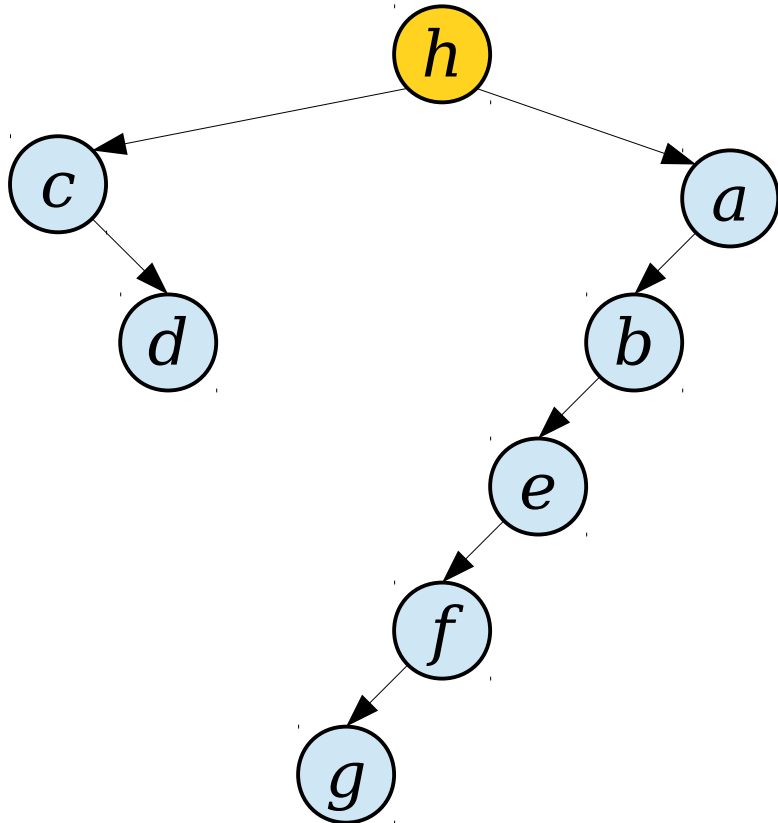
Question: Does rotating each accessed key to the root guarantee good overall performance?

We have a *mechanical* description of how we reshape the tree. Can we get an *operational* description?



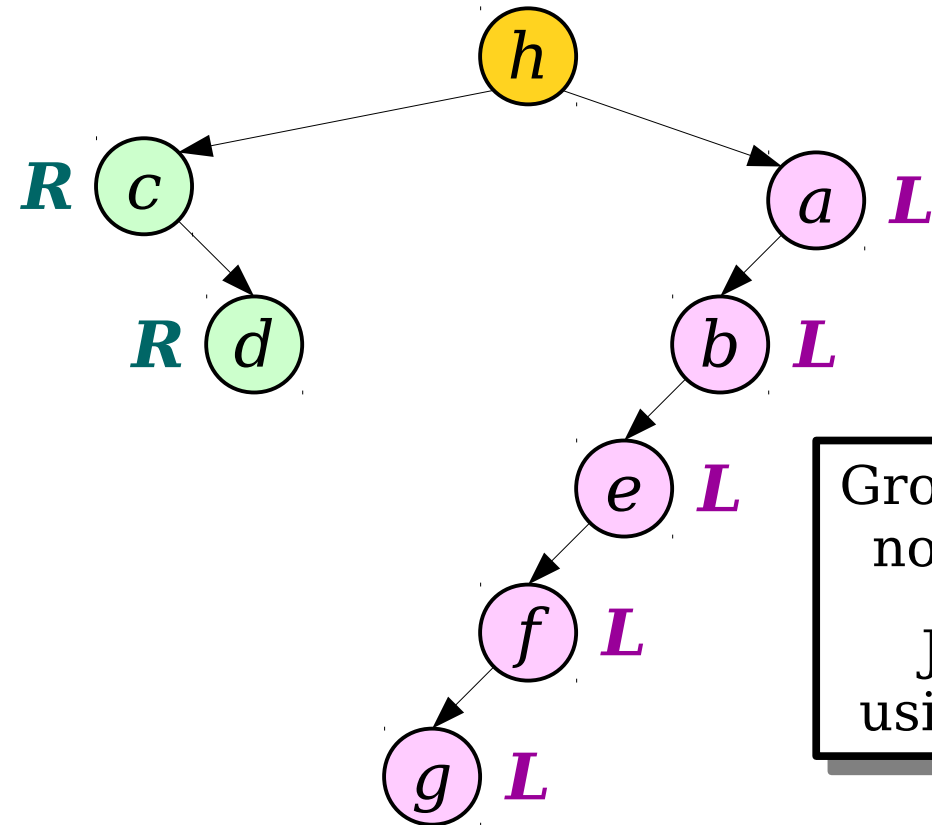
Question: Does rotating each accessed key to the root guarantee good overall performance?

We have a **mechanical** description of how we reshape the tree. Can we get an **operational** description?



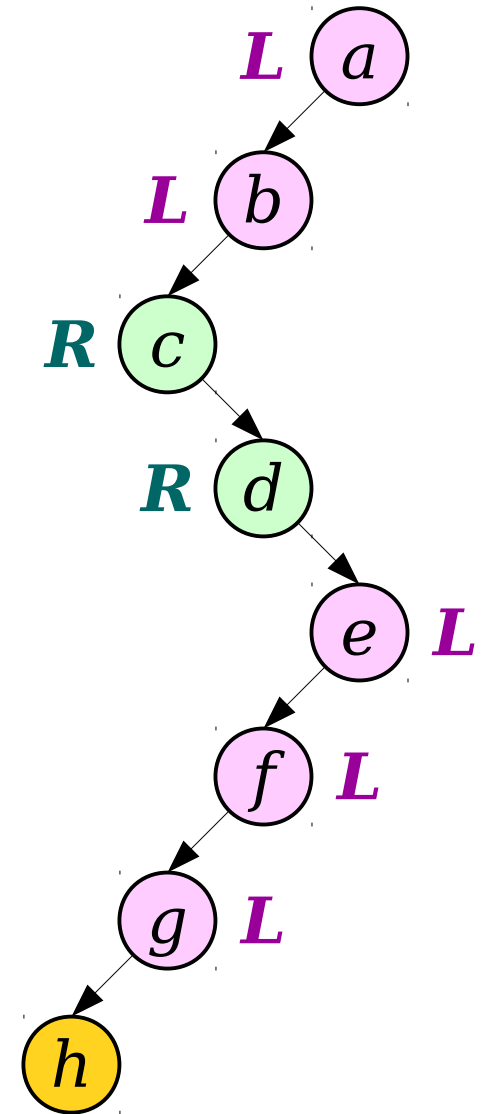
Question: Does rotating each accessed key to the root guarantee good overall performance?

We have a **mechanical** description of how we reshape the tree. Can we get an **operational** description?



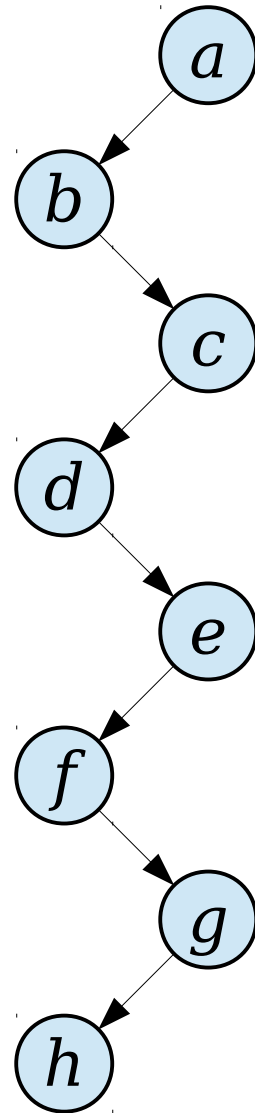
Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.



Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 1: This works really well on zig-zag-shaped trees.

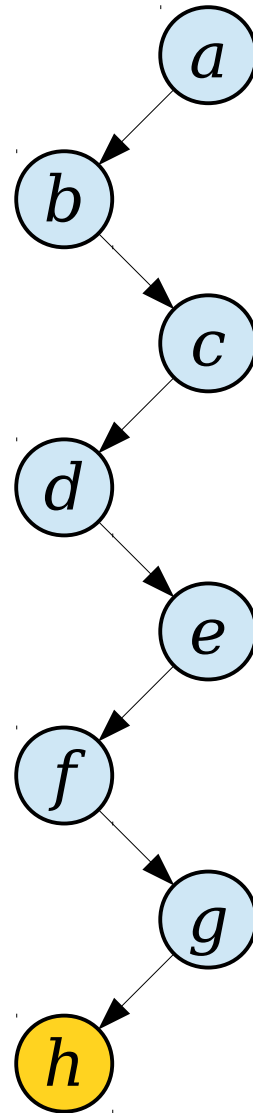


Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 1: This works really well on zig-zag-shaped trees.

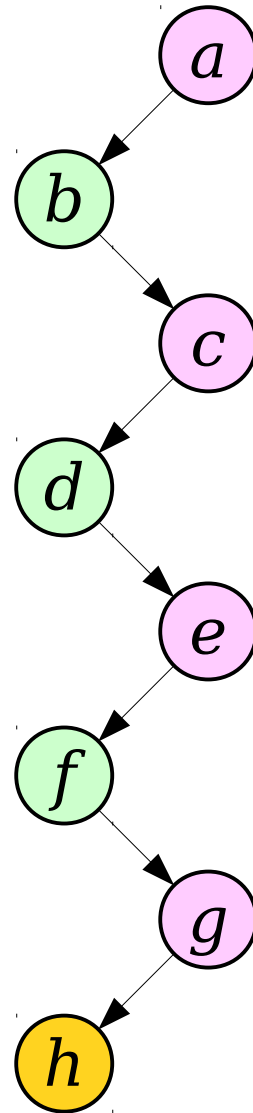


Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 1: This works really well on zig-zag-shaped trees.



Group **R**-type and **L**-type nodes into two chains.

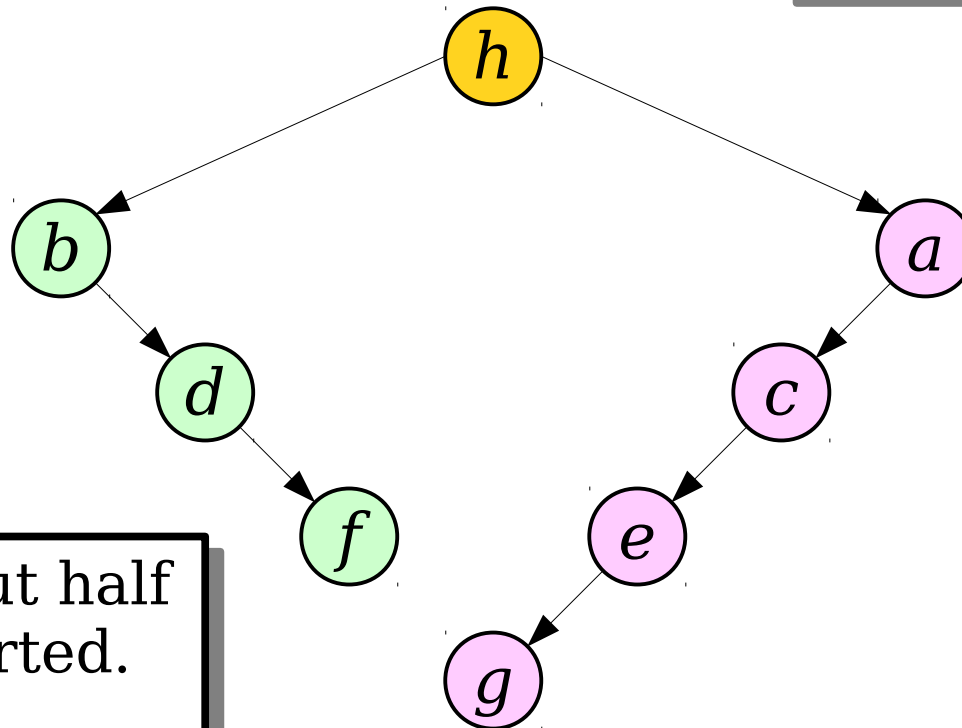
Join them together using the rotated node.

Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 1: This works really well on zig-zag-shaped trees.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.



This tree is about half as tall as it started.

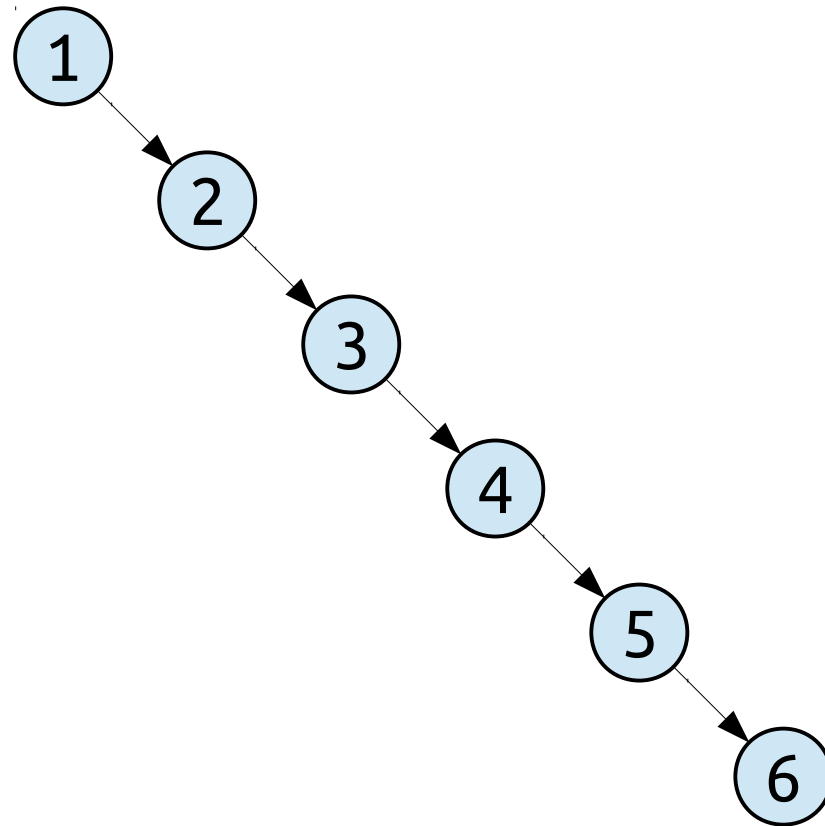
Most nodes on the access path are much closer to the root.

Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

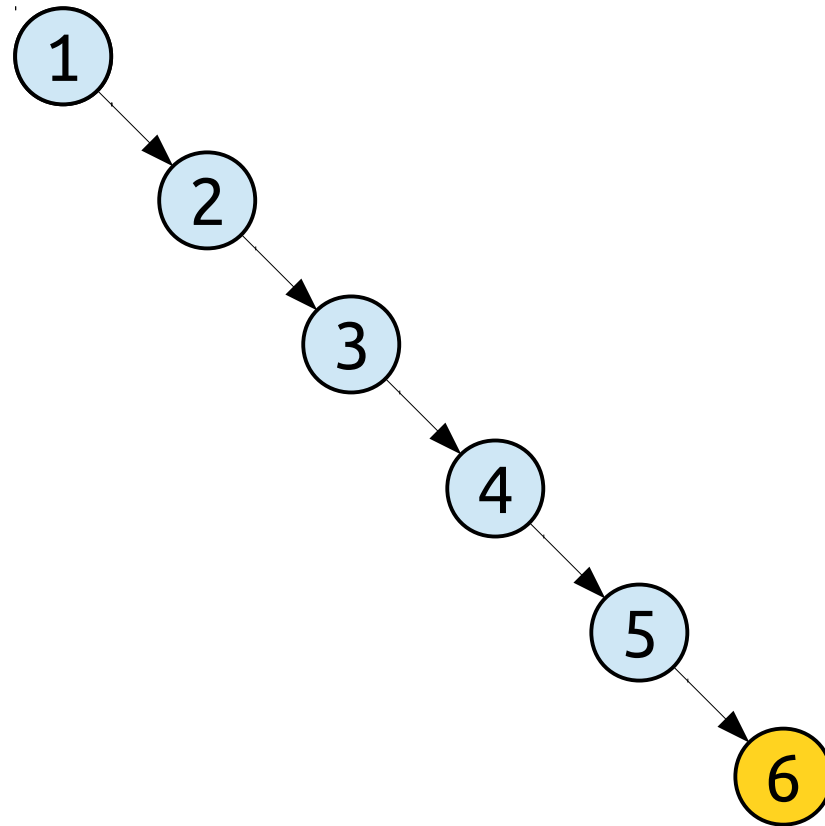


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

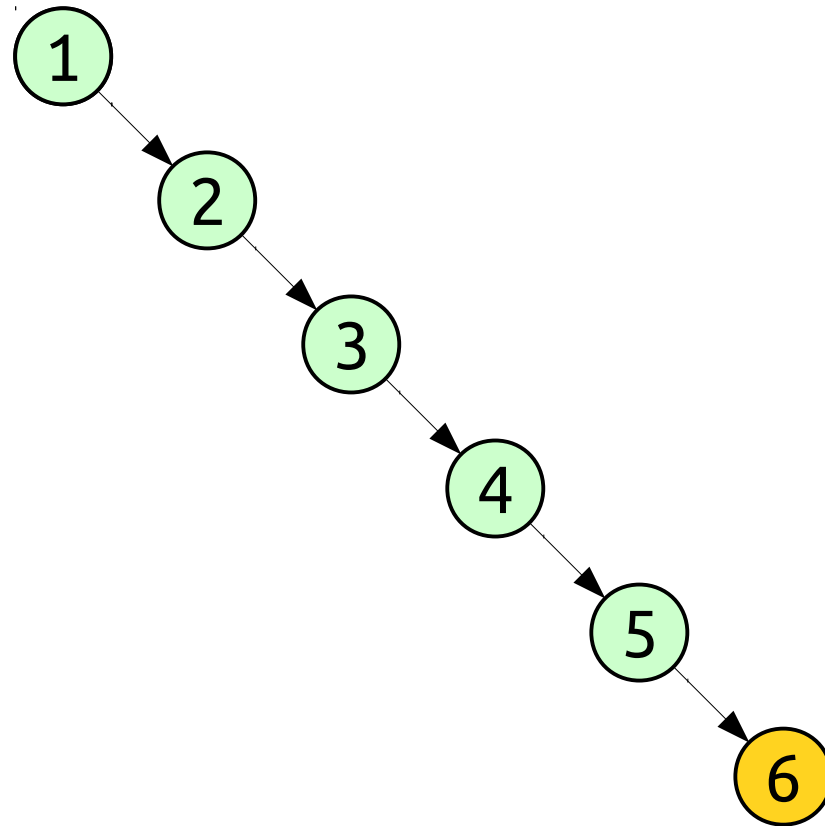


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

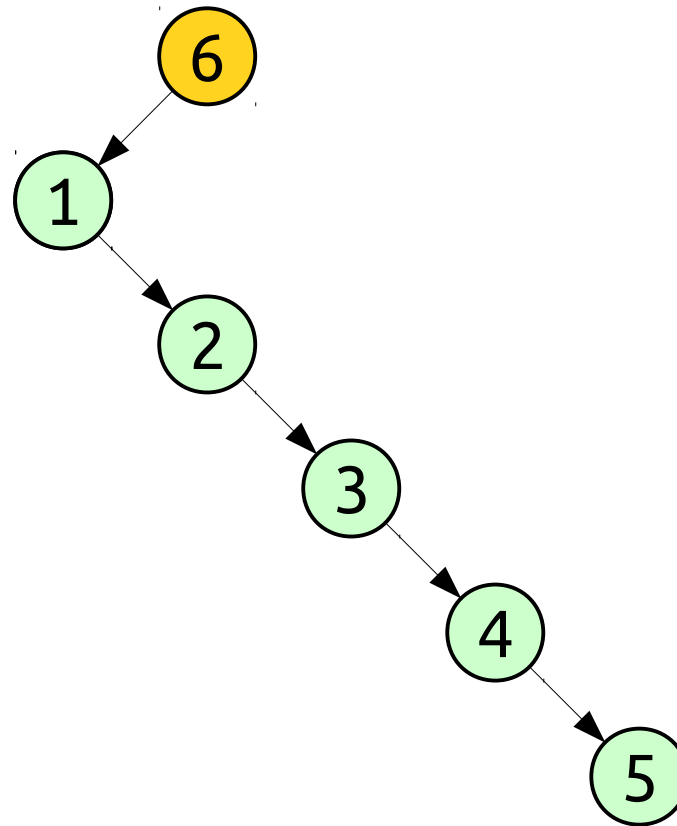


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

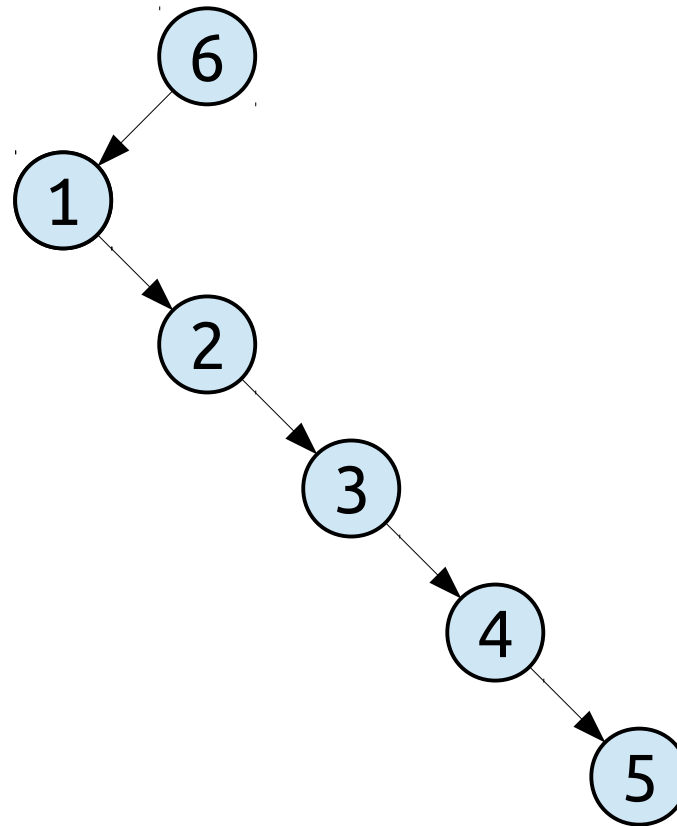


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

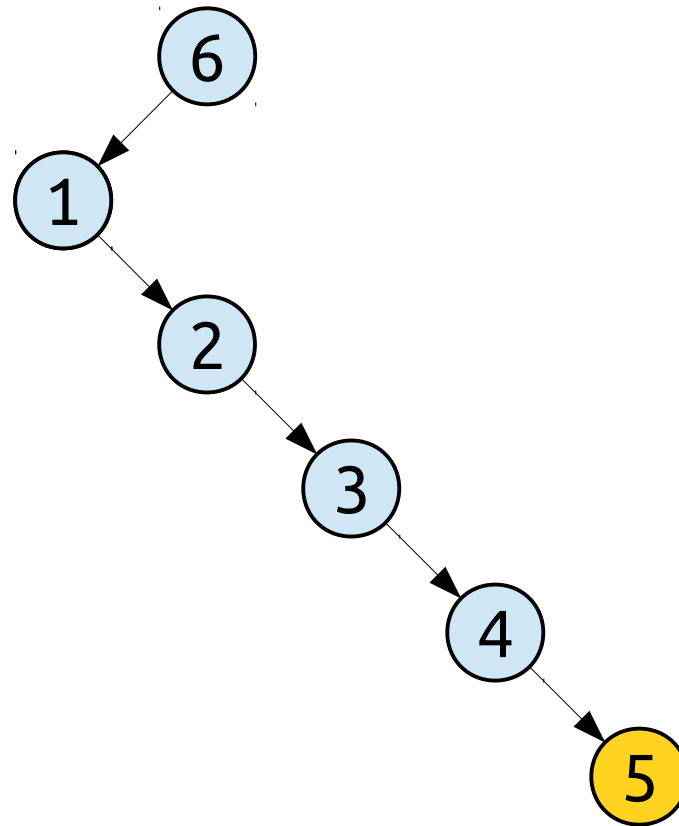


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

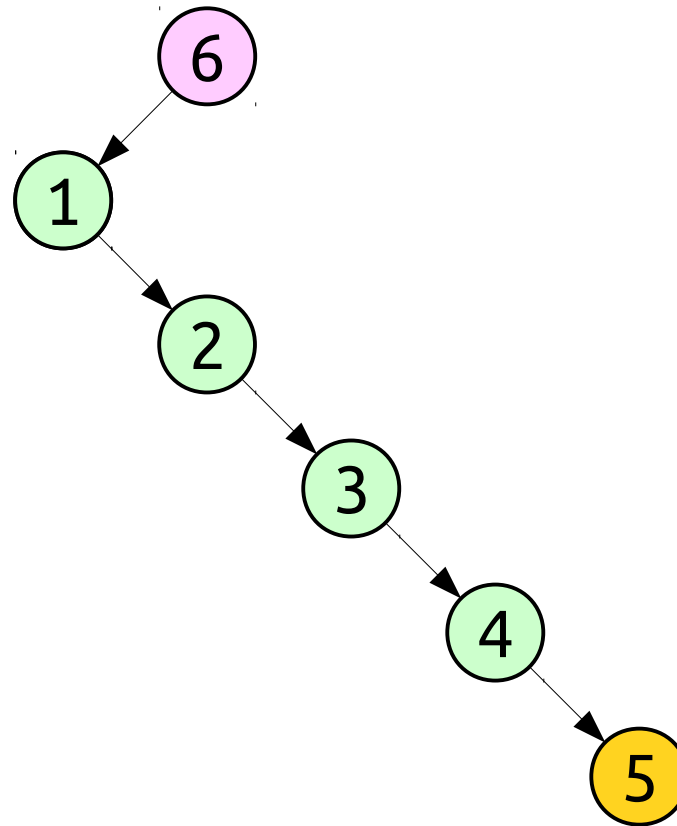


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

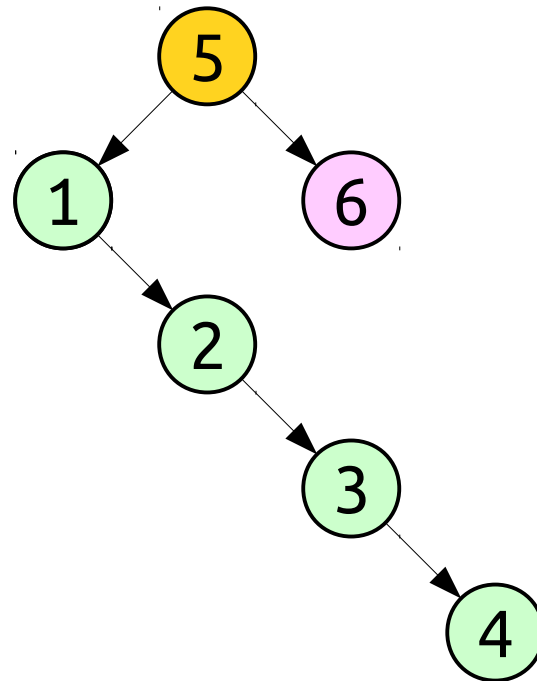


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

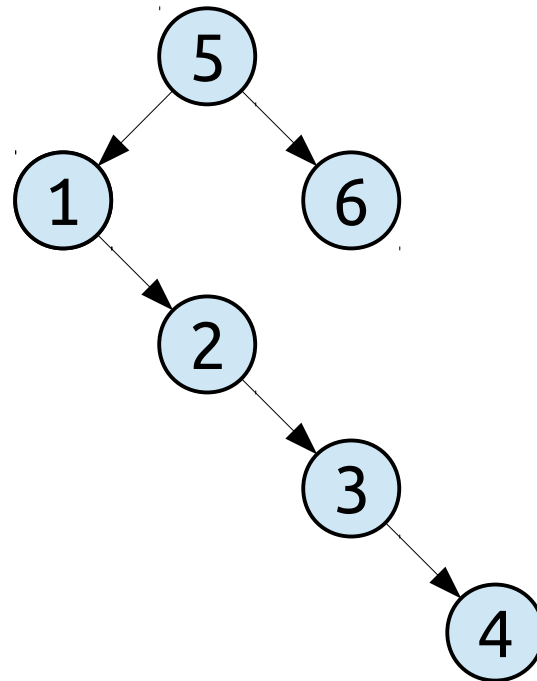


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

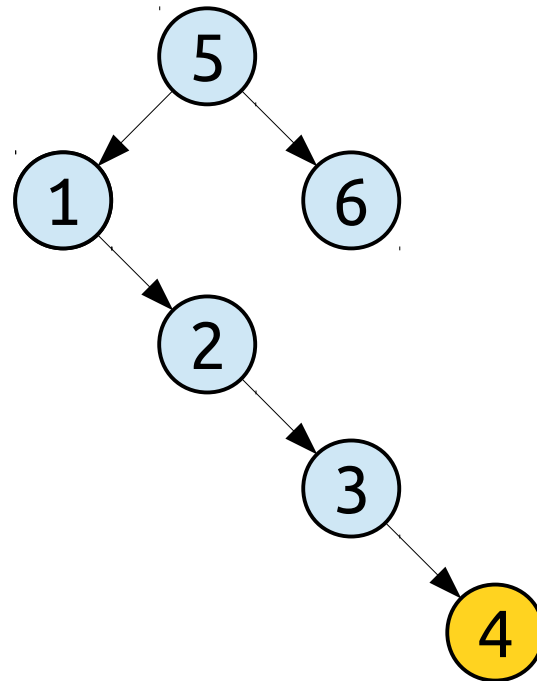


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

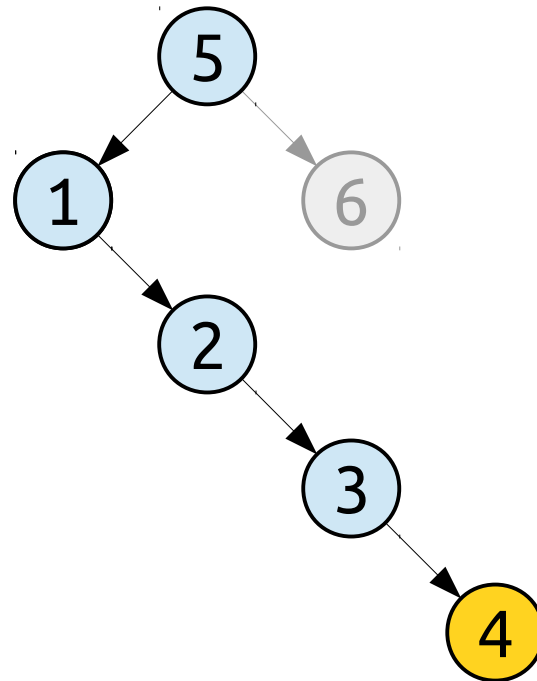


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

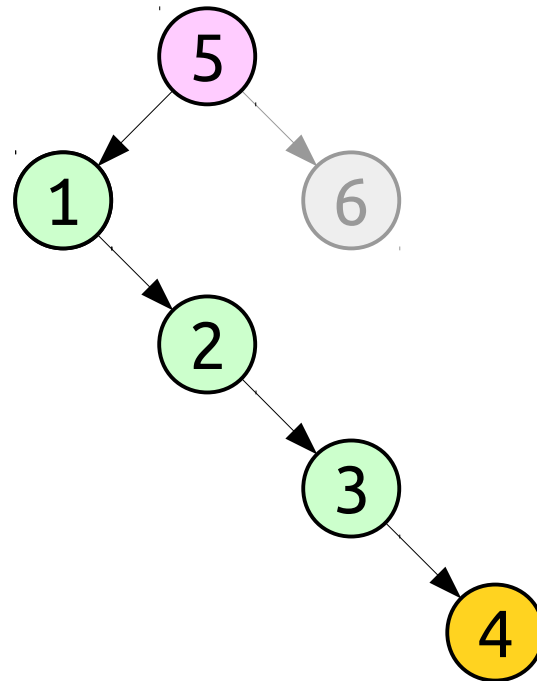


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

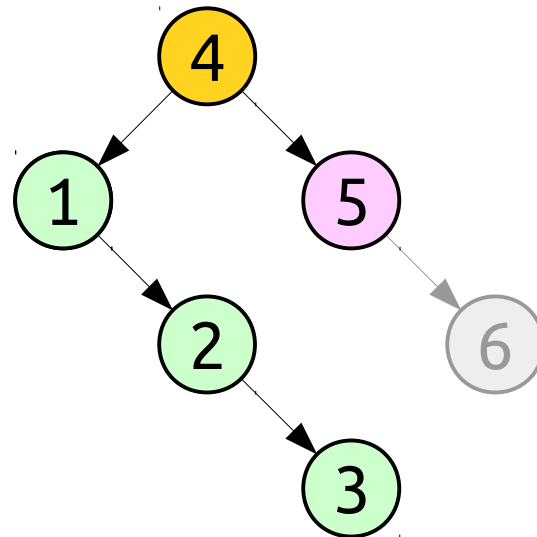


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

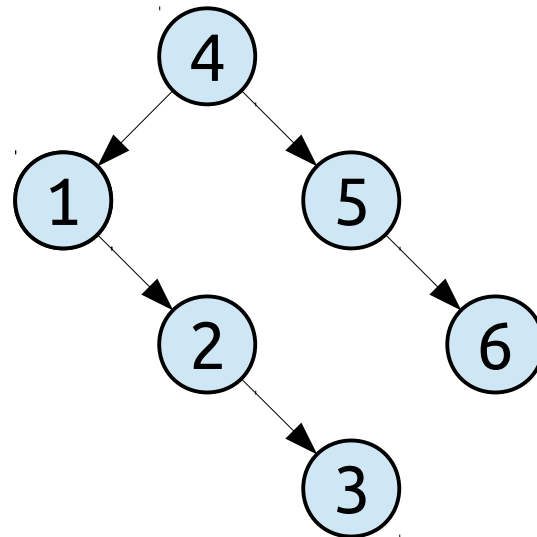


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

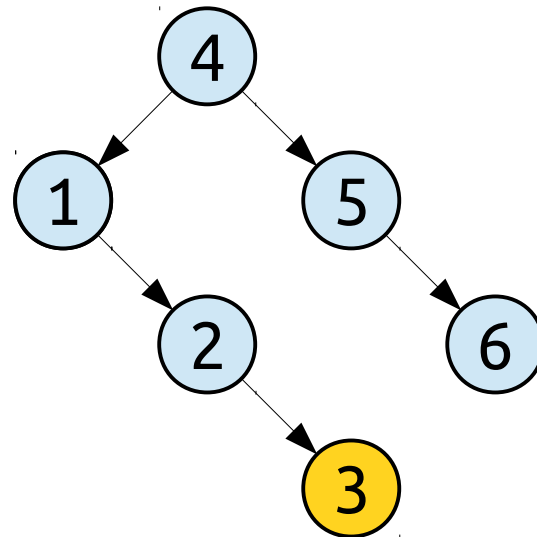


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

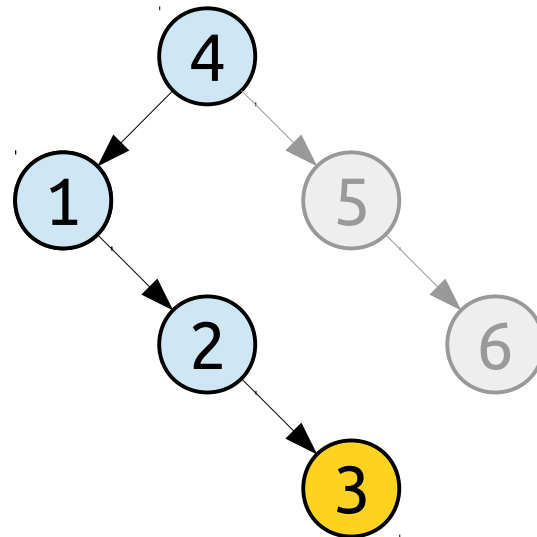


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

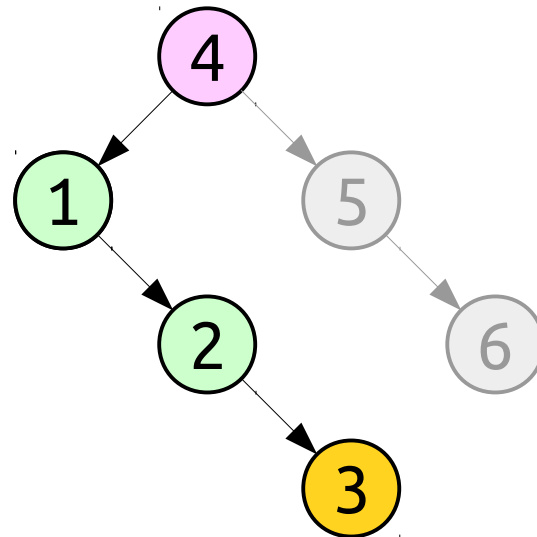


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

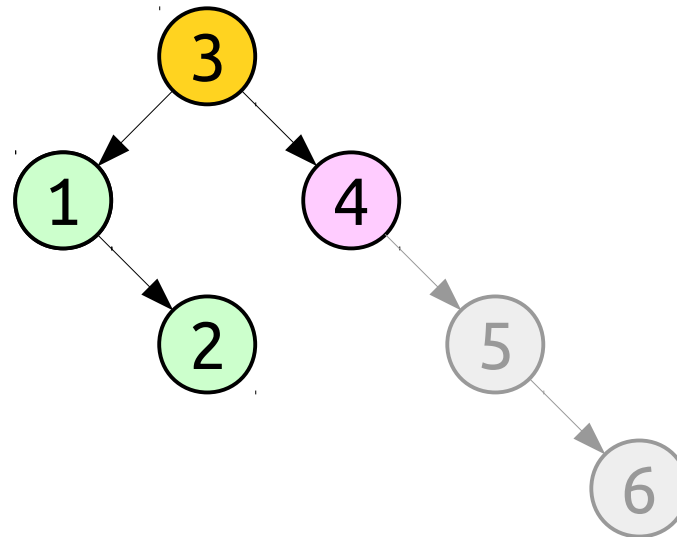


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

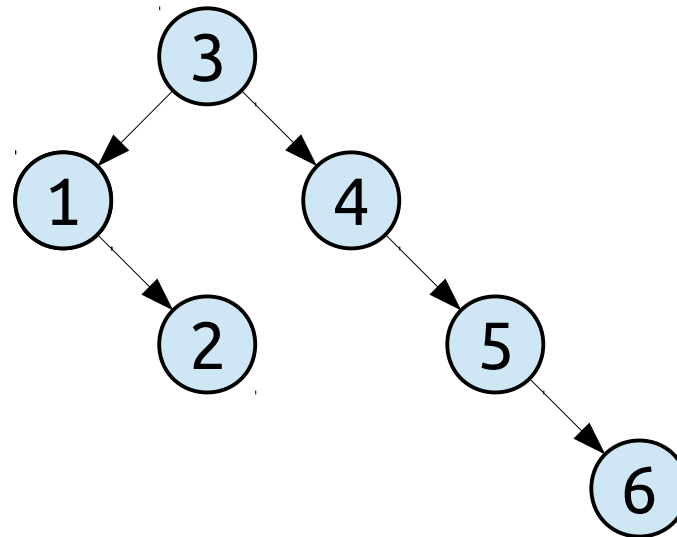


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

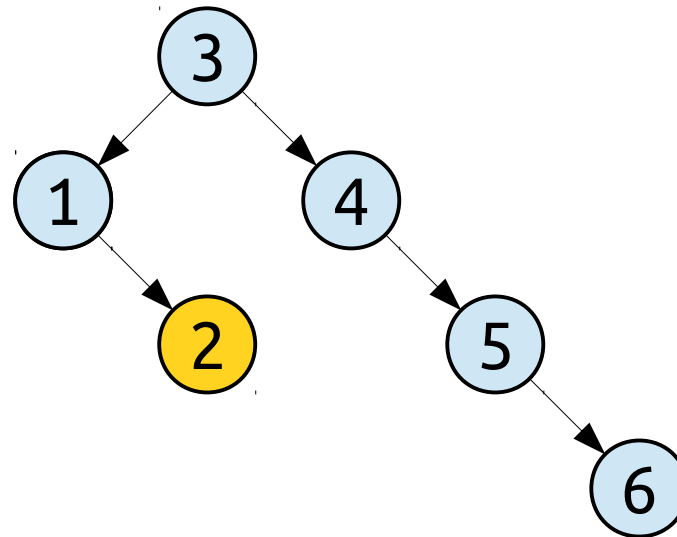


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

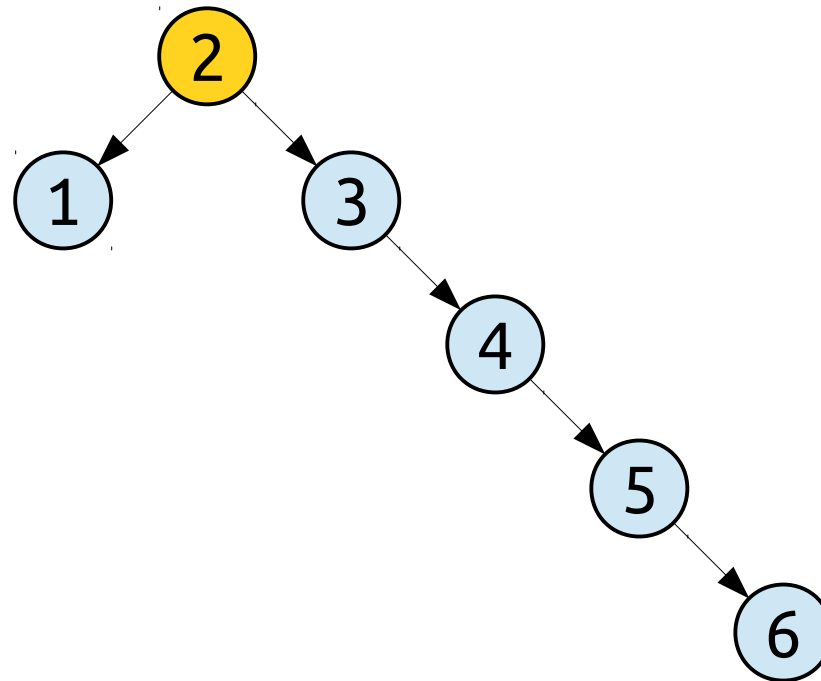


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

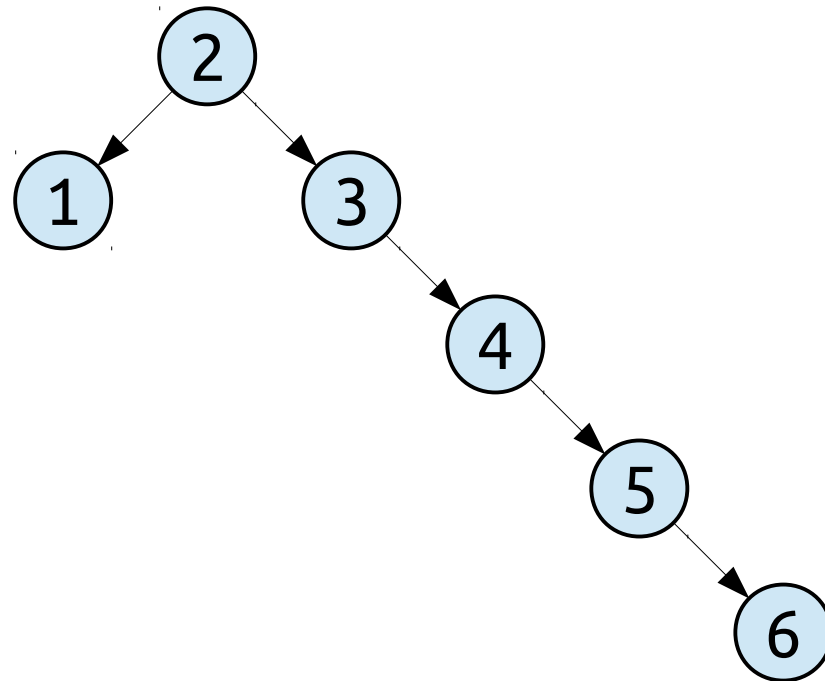


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

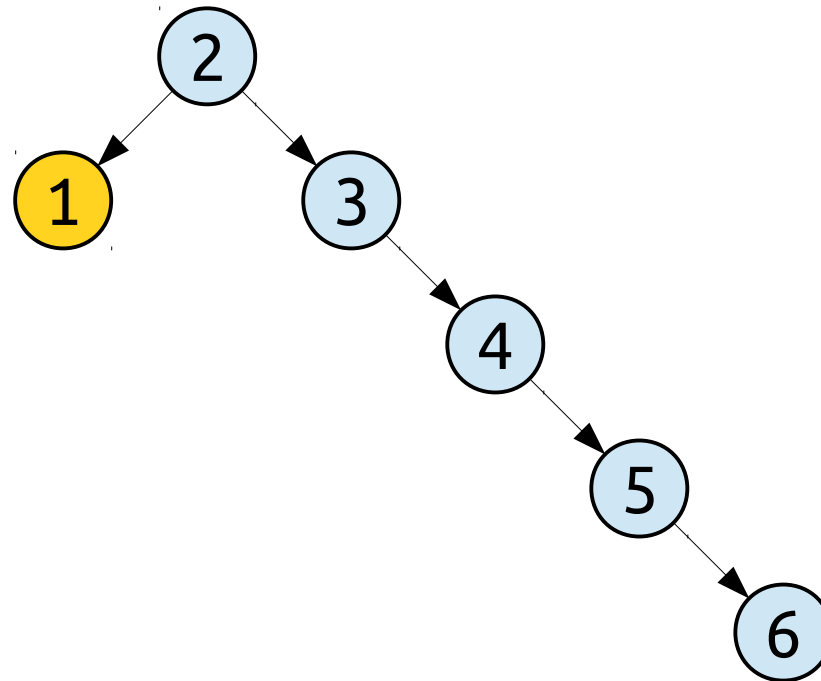


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

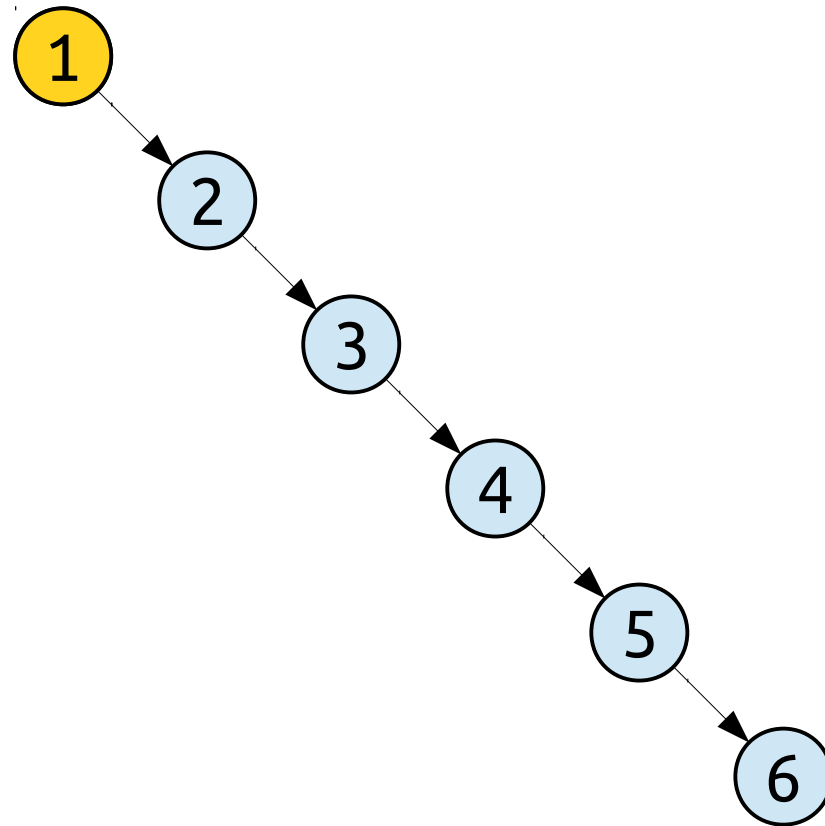


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

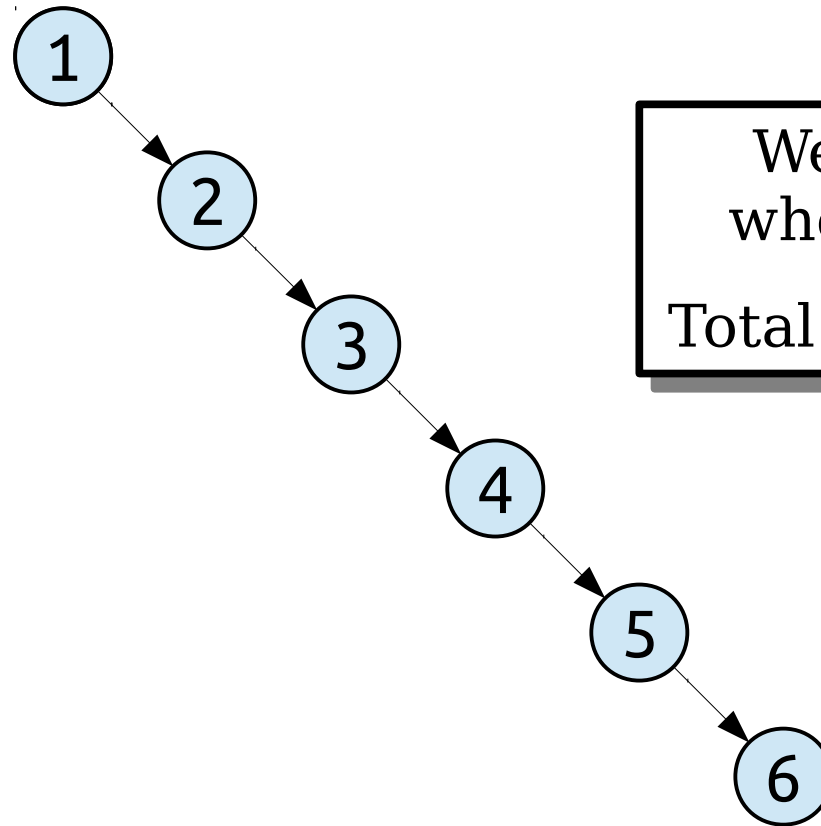


Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group **R**-type and **L**-type nodes into two chains.

Join them together using the rotated node.

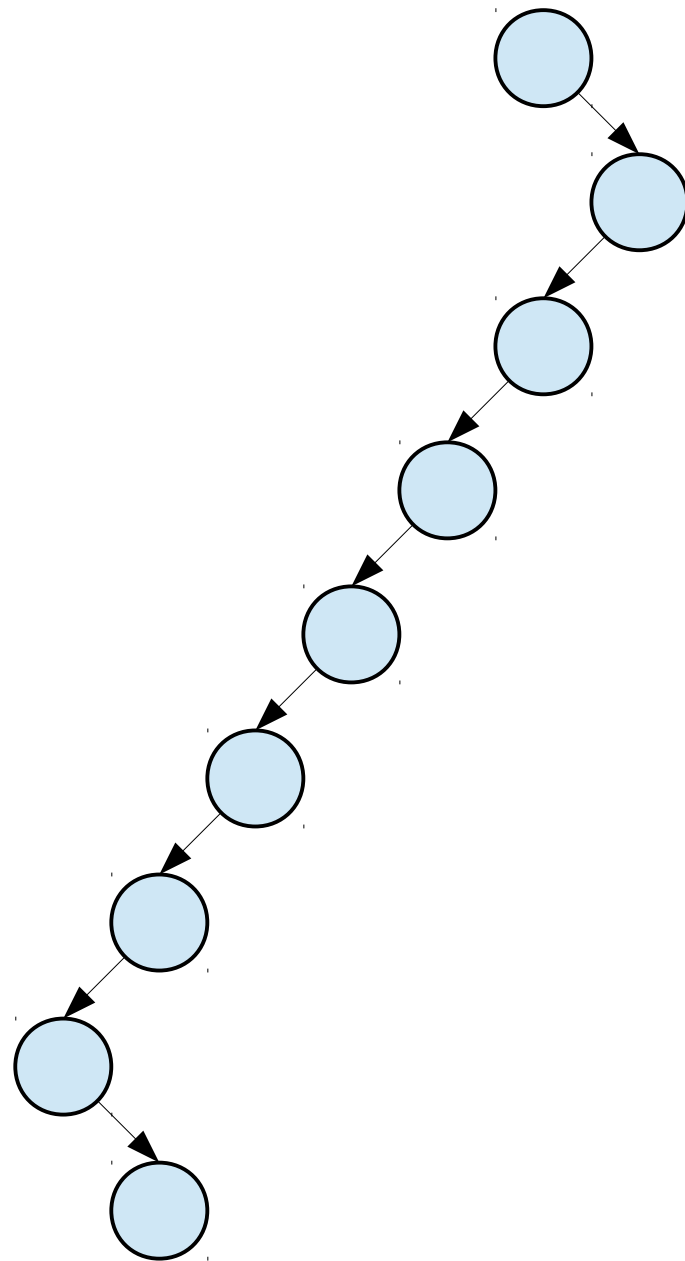


We're right back where we started!

Total rotations: $\Theta(n^2)$.

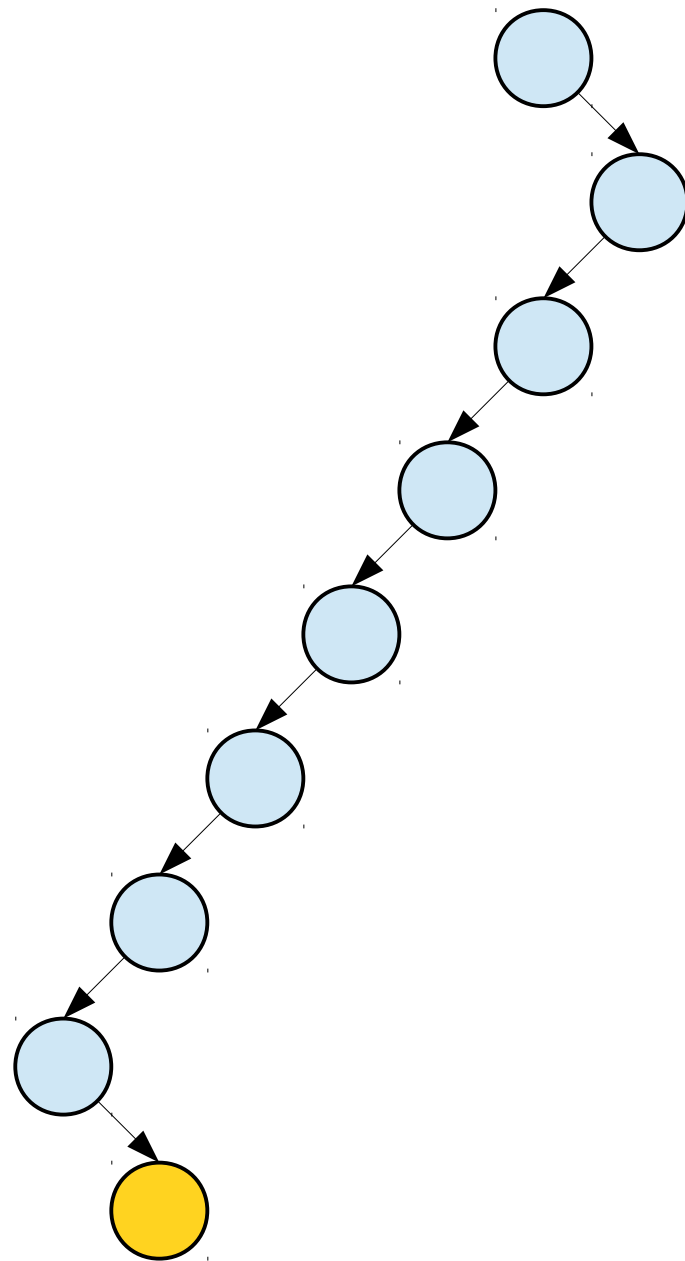
Question: Does rotating each accessed key to the root guarantee good overall performance?

Intuition: We already handle zig-zags well. Let's just fix the linear case.



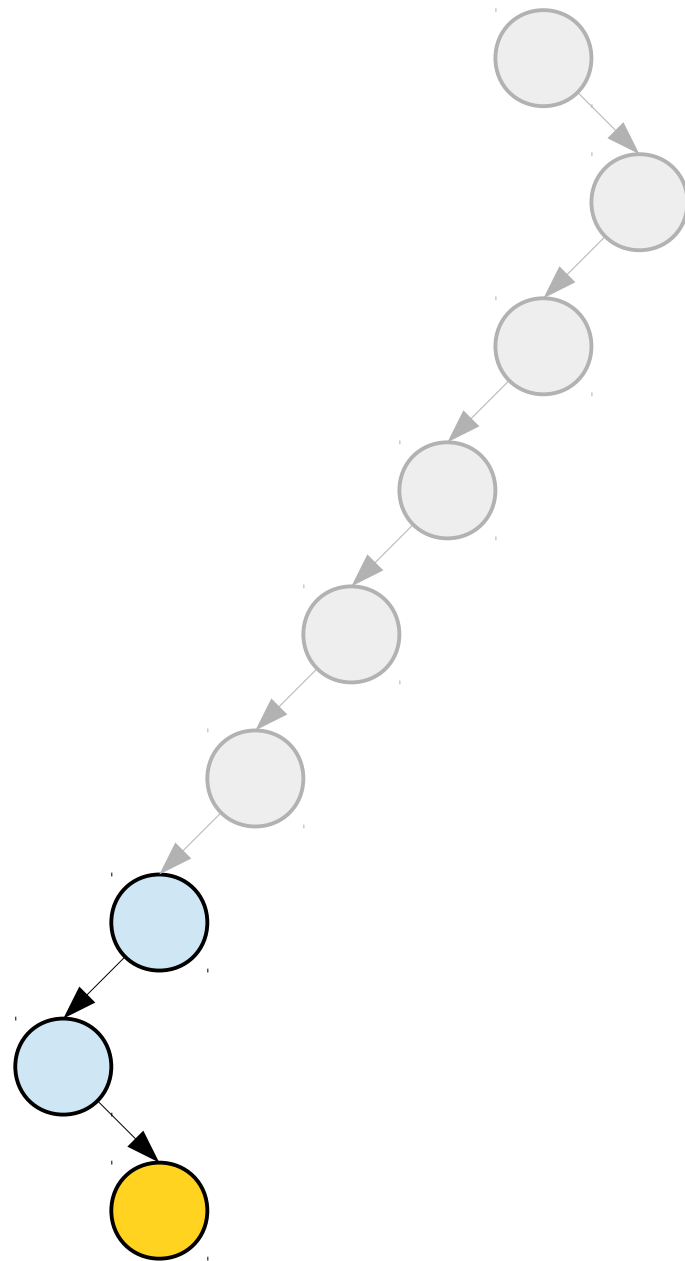
Question: How do we fix rotate-to-root to work well with long chains of nodes?

Intuition: We already handle zig-zags well. Let's just fix the linear case.



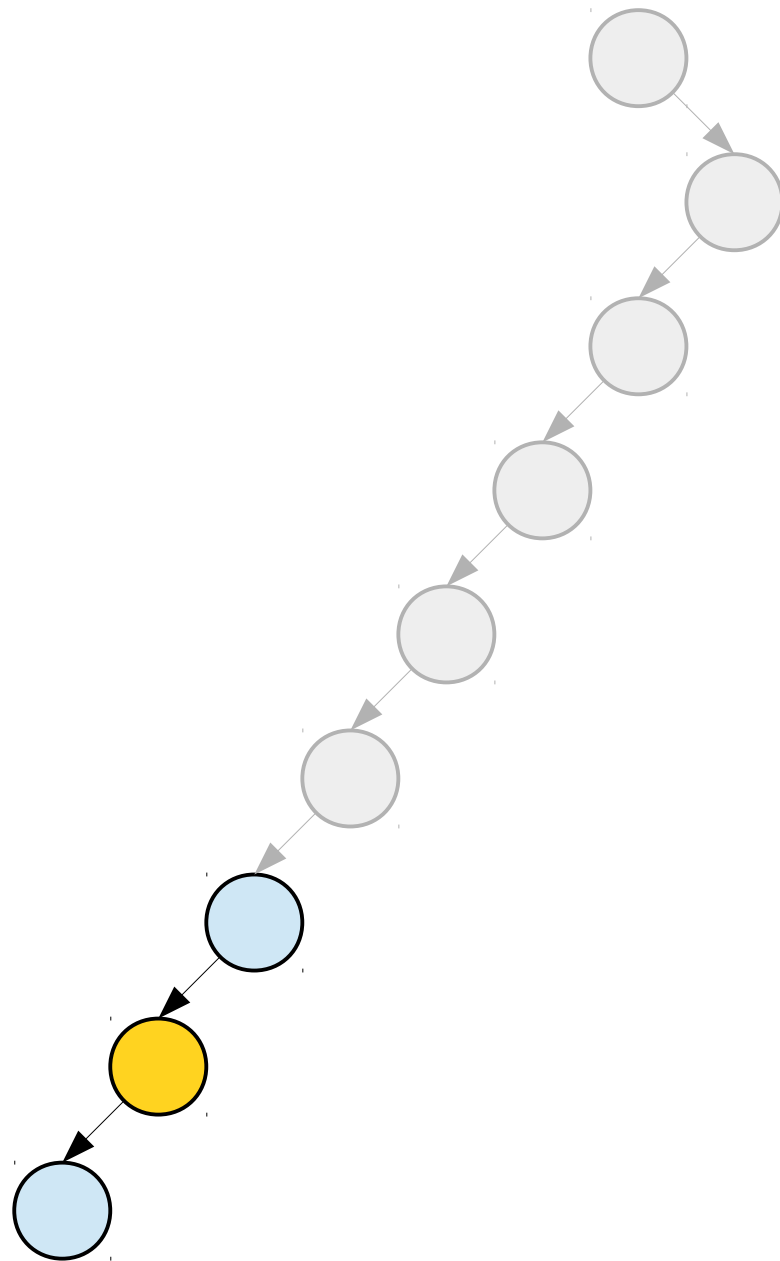
Question: How do we fix rotate-to-root to work well with long chains of nodes?

Intuition: We already handle zig-zags well. Let's just fix the linear case.



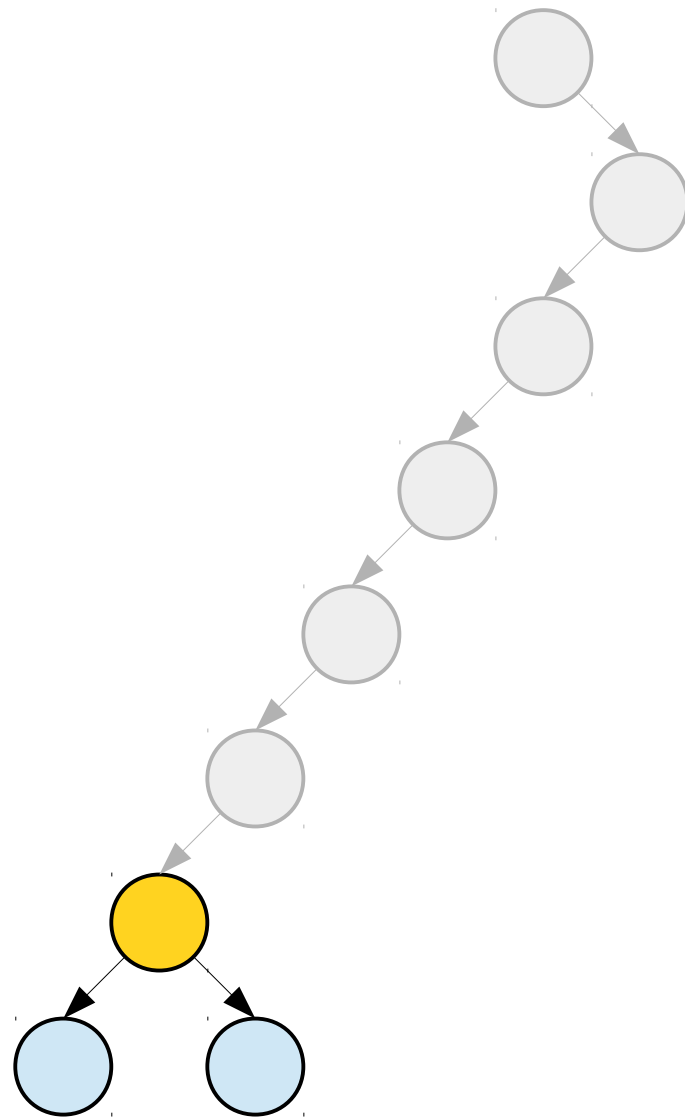
Question: How do we fix rotate-to-root to work well with long chains of nodes?

Intuition: We already handle zig-zags well. Let's just fix the linear case.



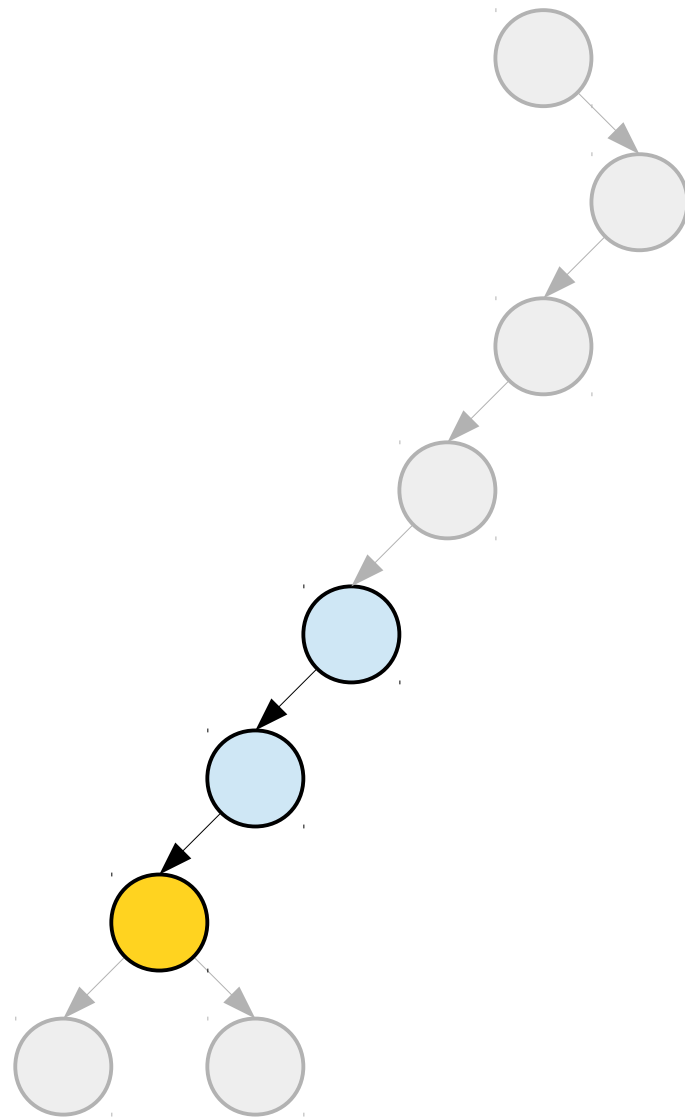
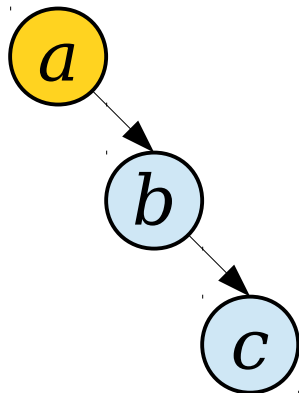
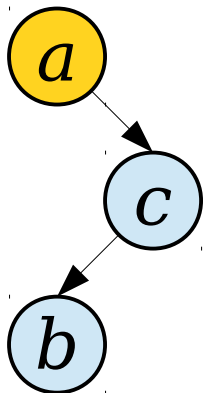
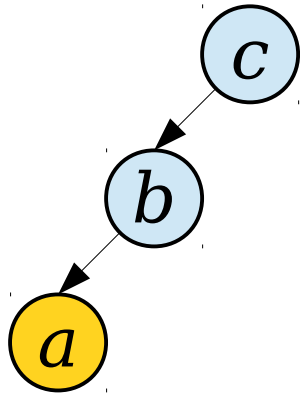
Question: How do we fix rotate-to-root to work well with long chains of nodes?

Intuition: We already handle zig-zags well. Let's just fix the linear case.



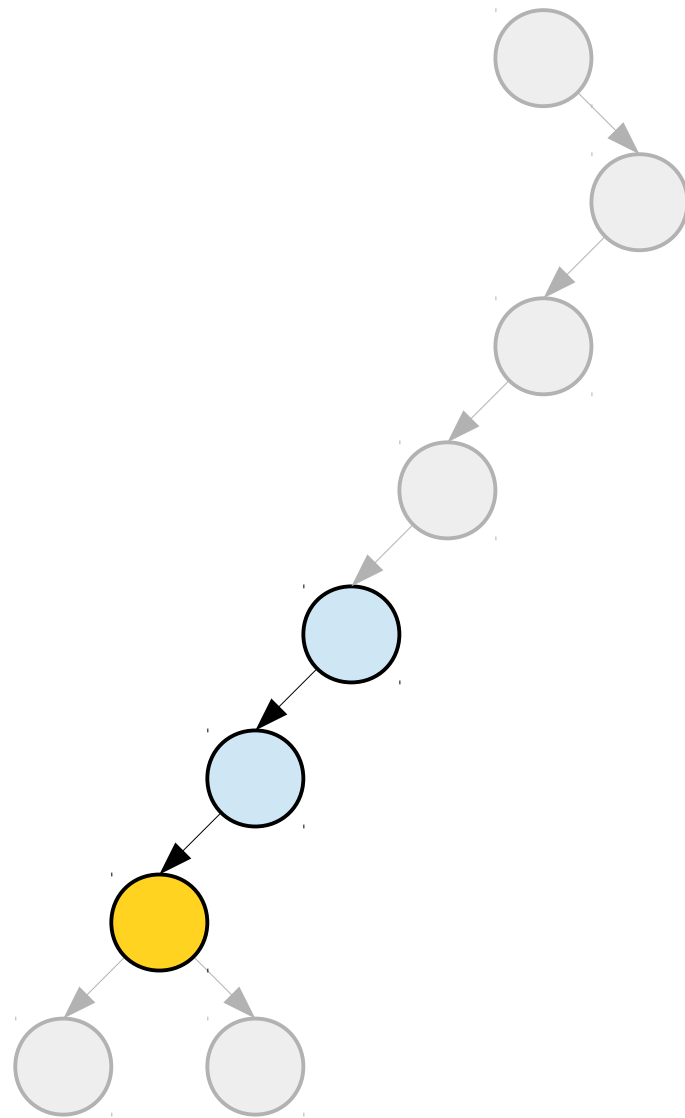
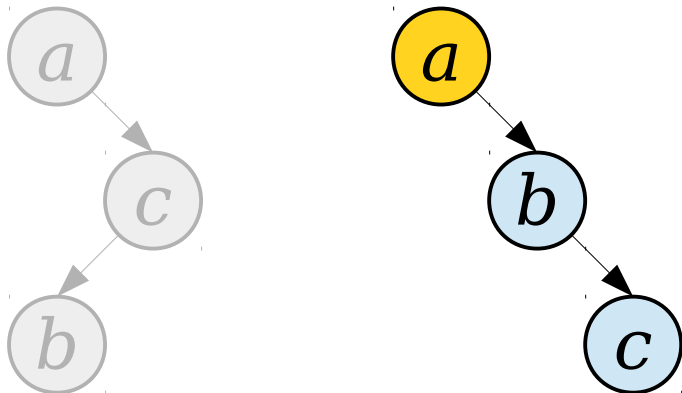
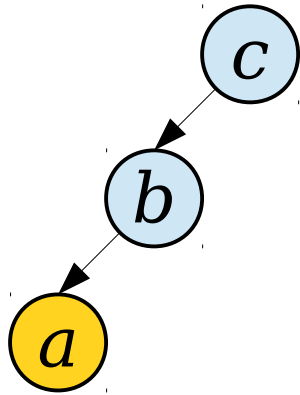
Question: How do we fix rotate-to-root to work well with long chains of nodes?

Intuition: We already handle zig-zags well. Let's just fix the linear case.



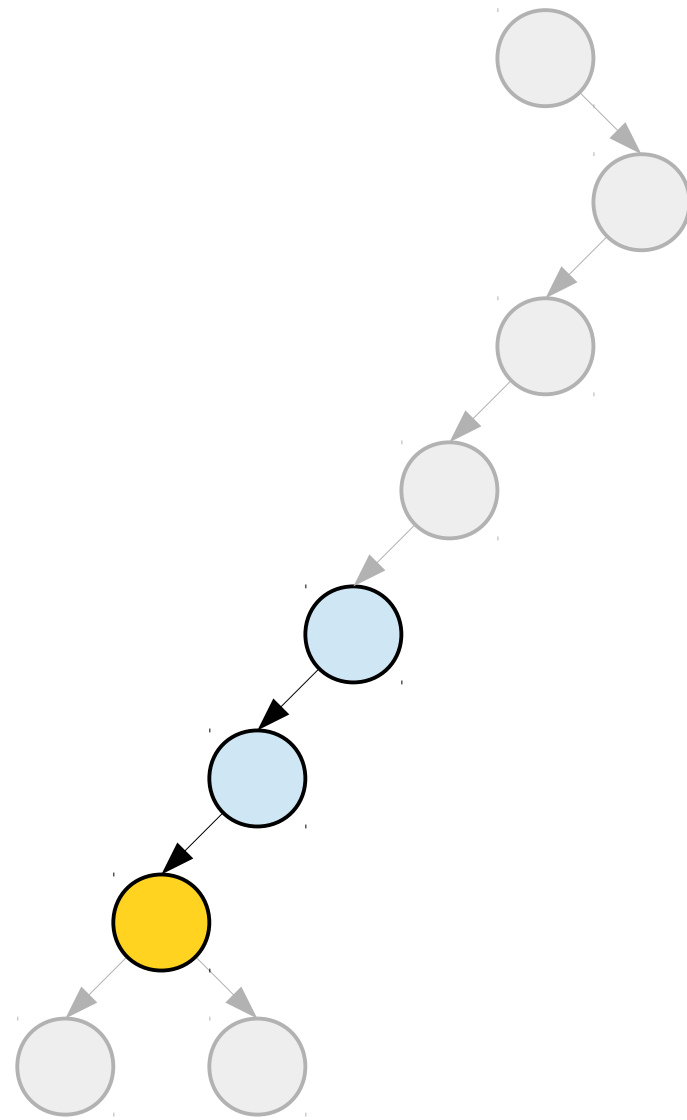
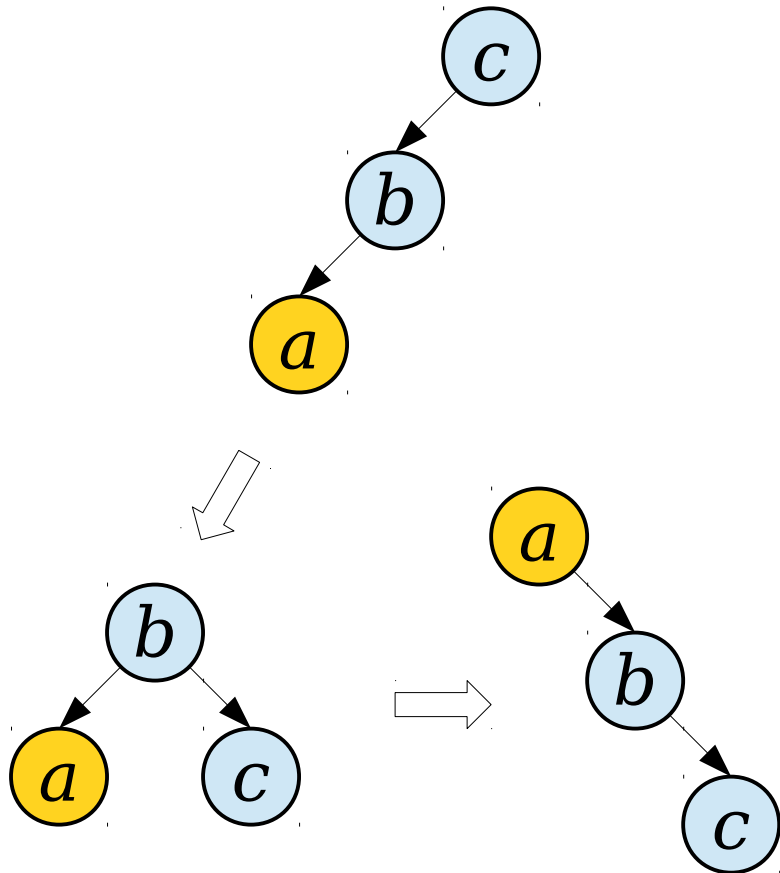
Question: How do we fix rotate-to-root to work well with long chains of nodes?

Intuition: We already handle zig-zags well. Let's just fix the linear case.



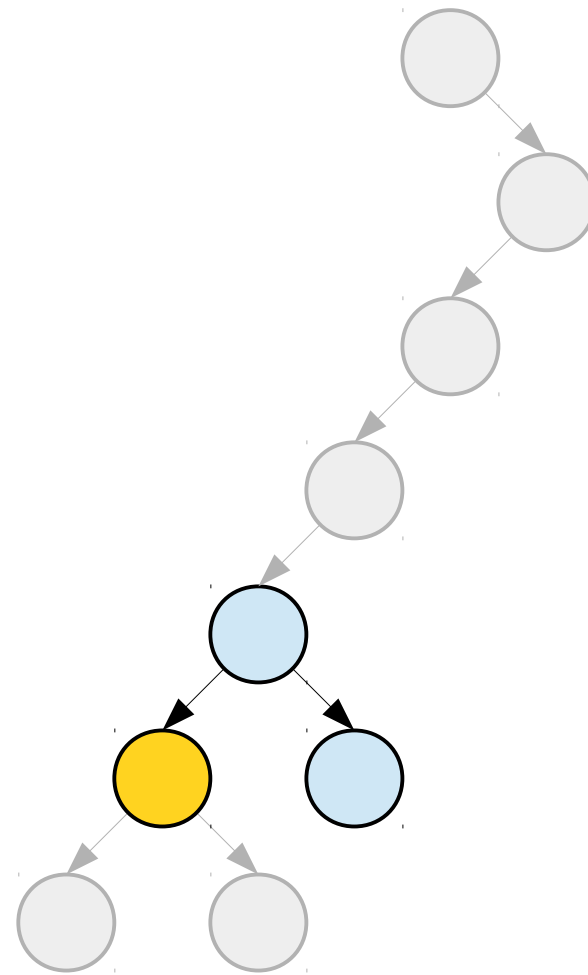
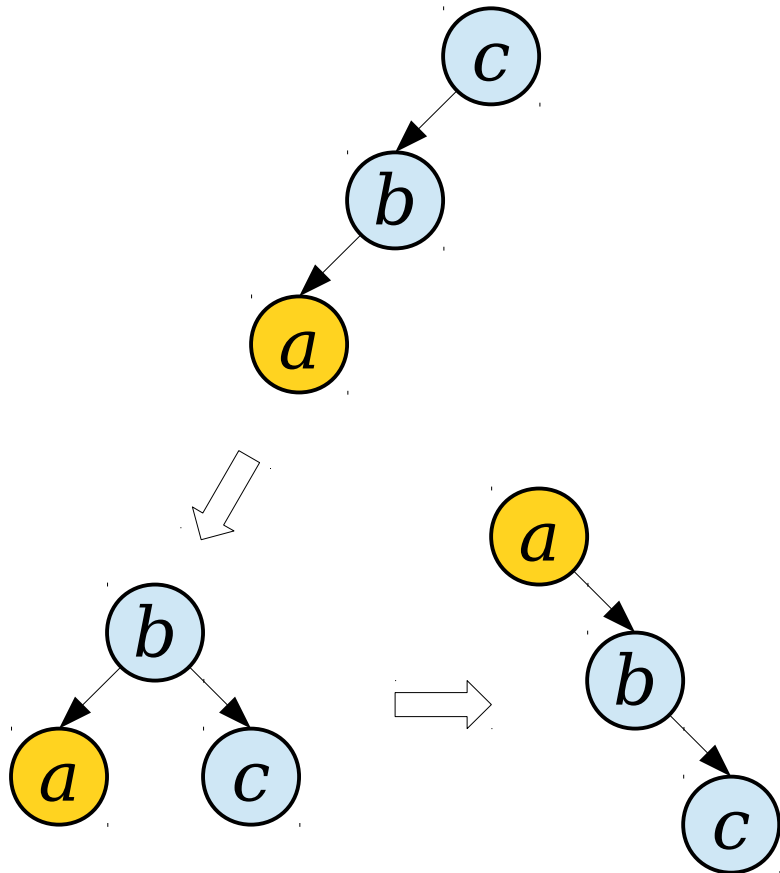
Question: How do we fix rotate-to-root to work well with long chains of nodes?

Intuition: We already handle zig-zags well. Let's just fix the linear case.



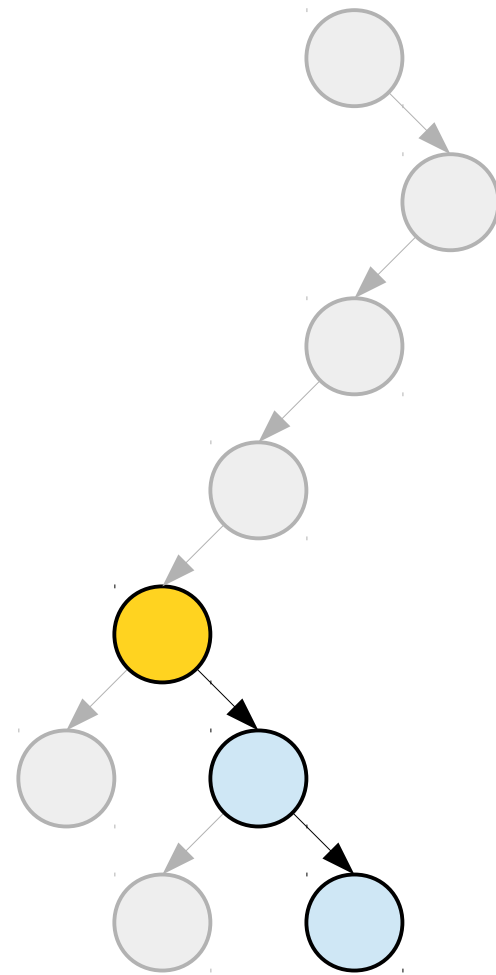
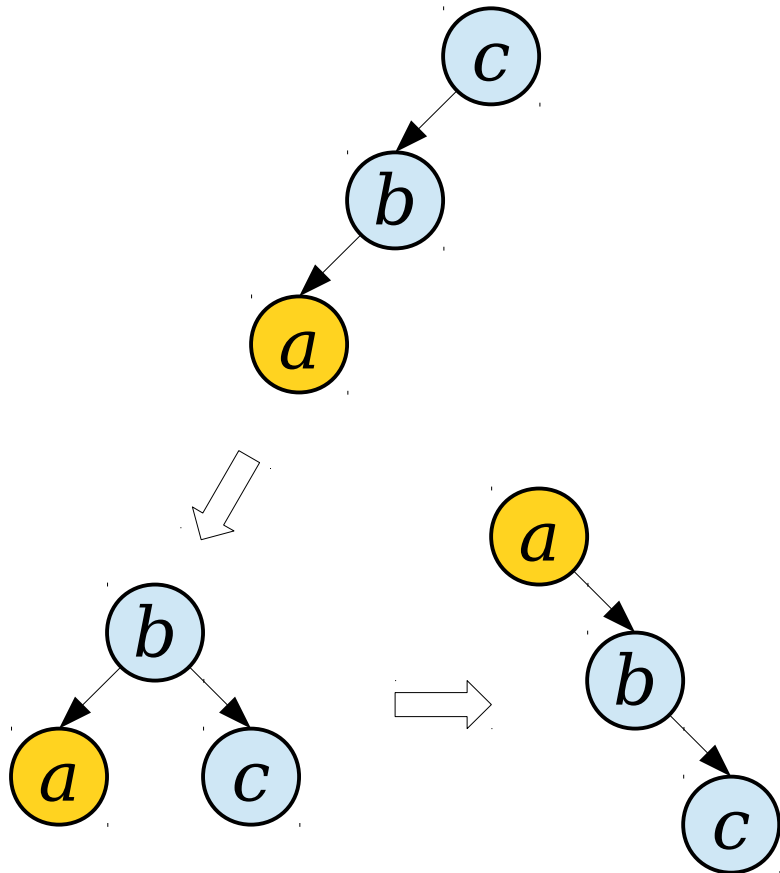
Question: How do we fix rotate-to-root to work well with long chains of nodes?

Intuition: We already handle zig-zags well. Let's just fix the linear case.



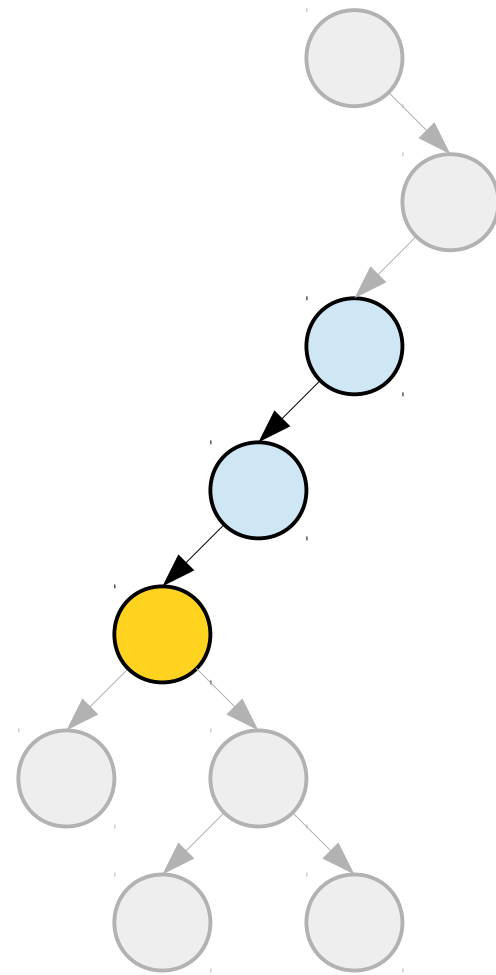
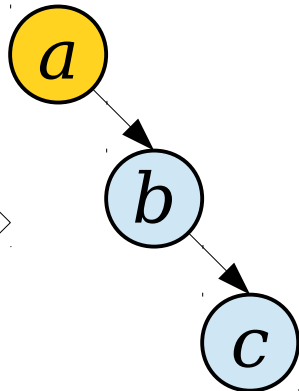
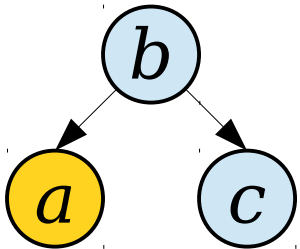
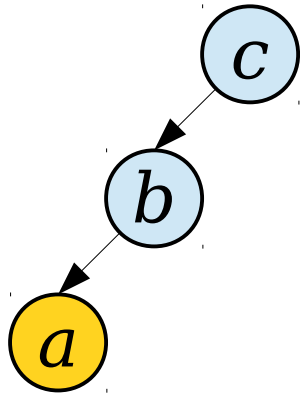
Question: How do we fix rotate-to-root to work well with long chains of nodes?

Intuition: We already handle zig-zags well. Let's just fix the linear case.



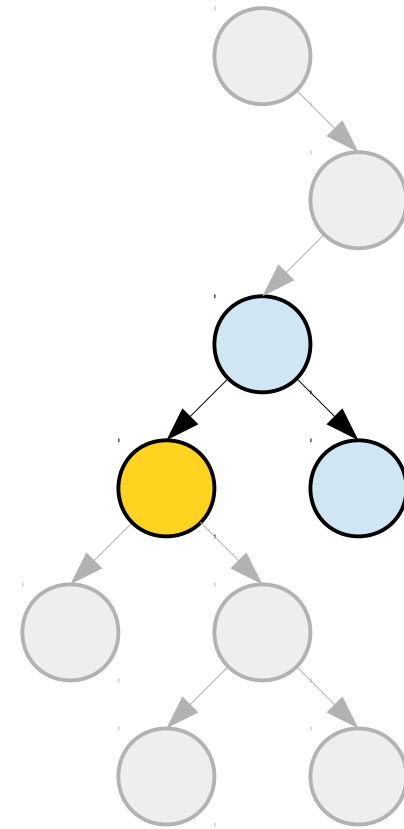
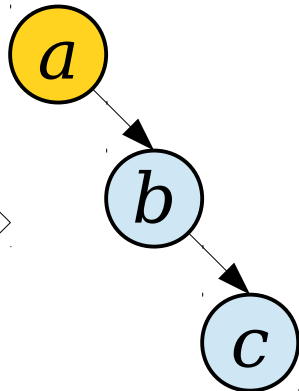
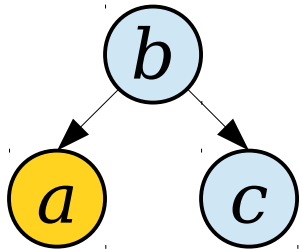
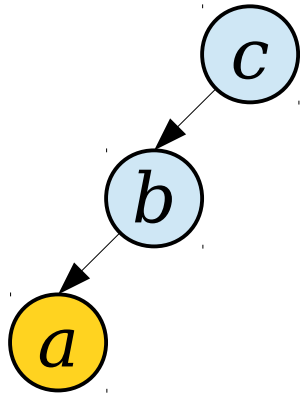
Question: How do we fix rotate-to-root to work well with long chains of nodes?

Intuition: We already handle zig-zags well. Let's just fix the linear case.



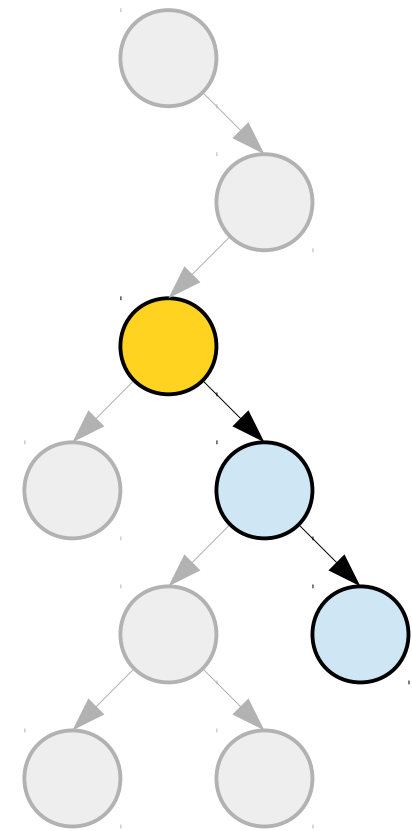
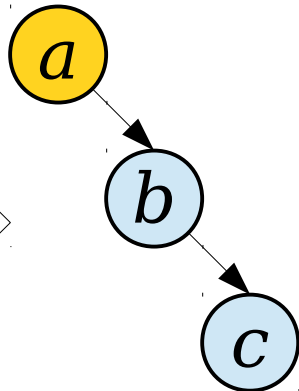
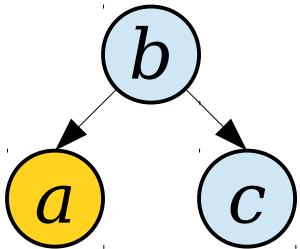
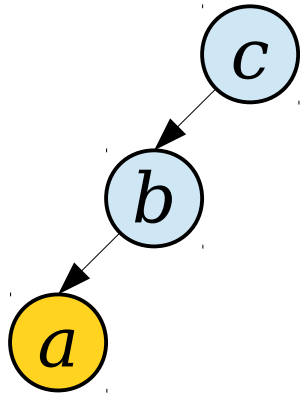
Question: How do we fix rotate-to-root to work well with long chains of nodes?

Intuition: We already handle zig-zags well. Let's just fix the linear case.



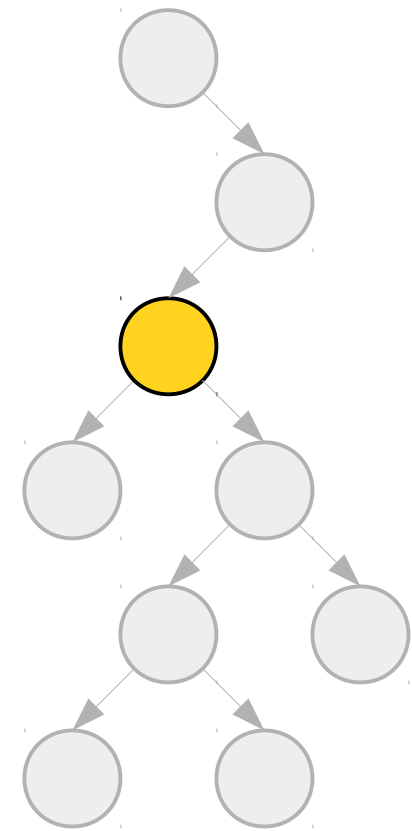
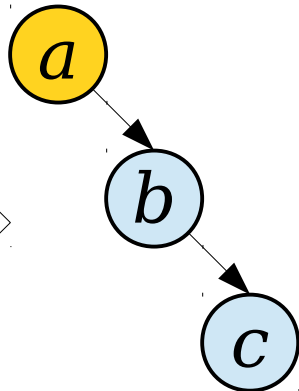
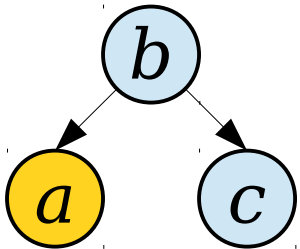
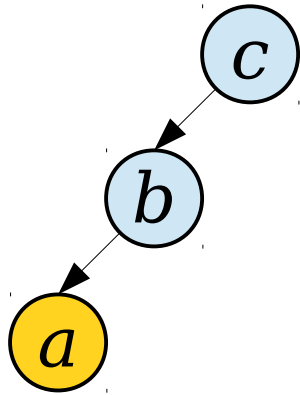
Question: How do we fix rotate-to-root to work well with long chains of nodes?

Intuition: We already handle zig-zags well. Let's just fix the linear case.



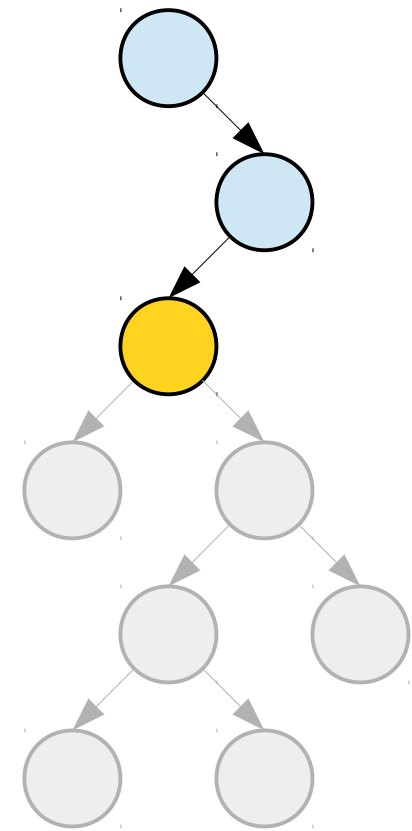
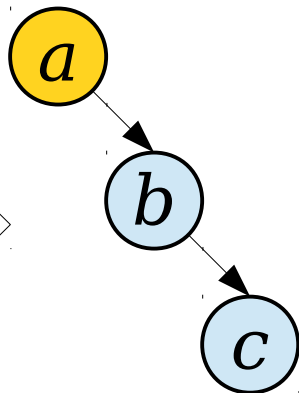
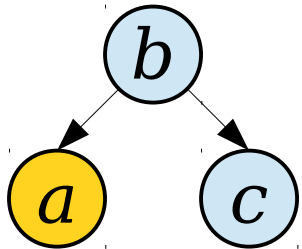
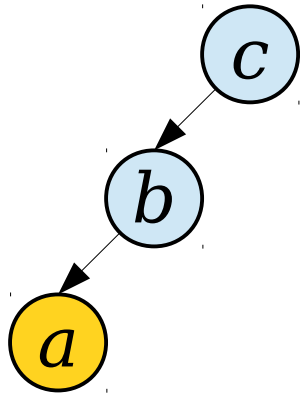
Question: How do we fix rotate-to-root to work well with long chains of nodes?

Intuition: We already handle zig-zags well. Let's just fix the linear case.



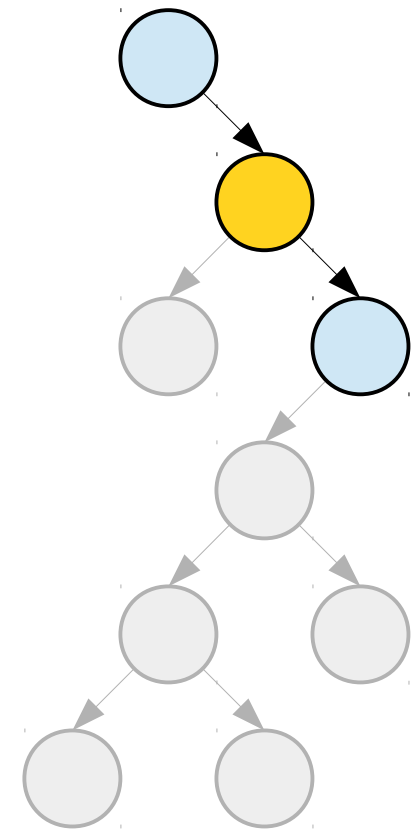
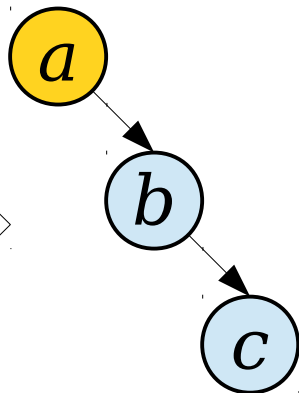
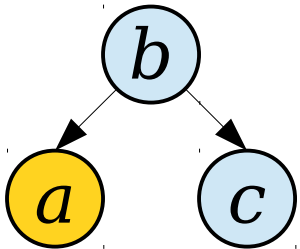
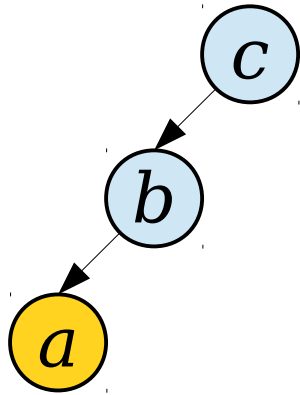
Question: How do we fix rotate-to-root to work well with long chains of nodes?

Intuition: We already handle zig-zags well. Let's just fix the linear case.



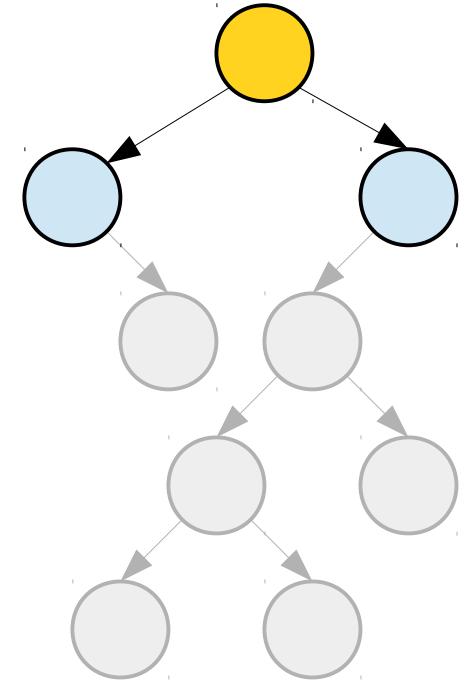
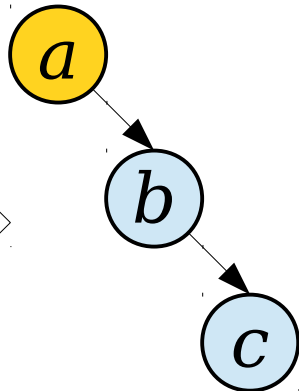
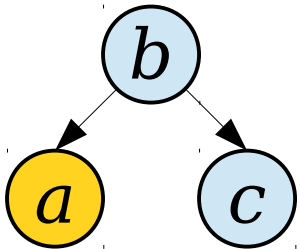
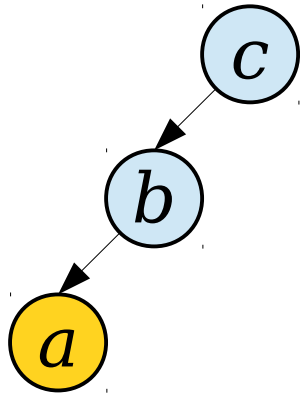
Question: How do we fix rotate-to-root to work well with long chains of nodes?

Intuition: We already handle zig-zags well. Let's just fix the linear case.



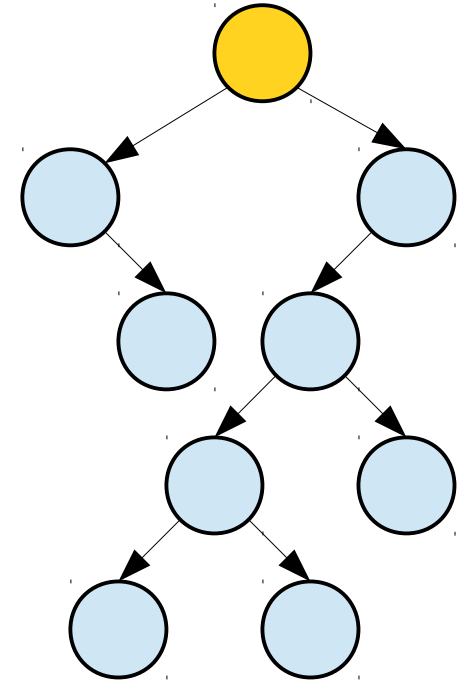
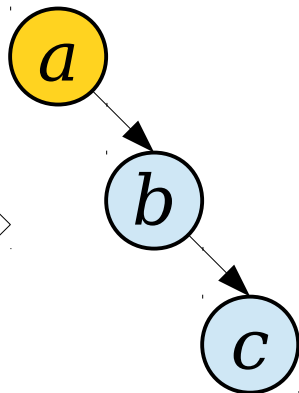
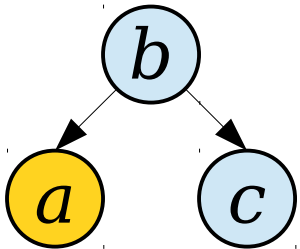
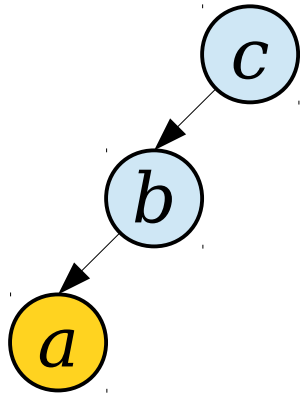
Question: How do we fix rotate-to-root to work well with long chains of nodes?

Intuition: We already handle zig-zags well. Let's just fix the linear case.



Question: How do we fix rotate-to-root to work well with long chains of nodes?

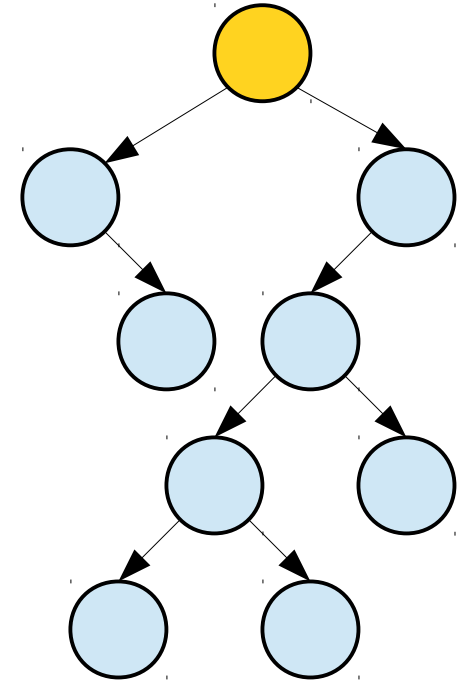
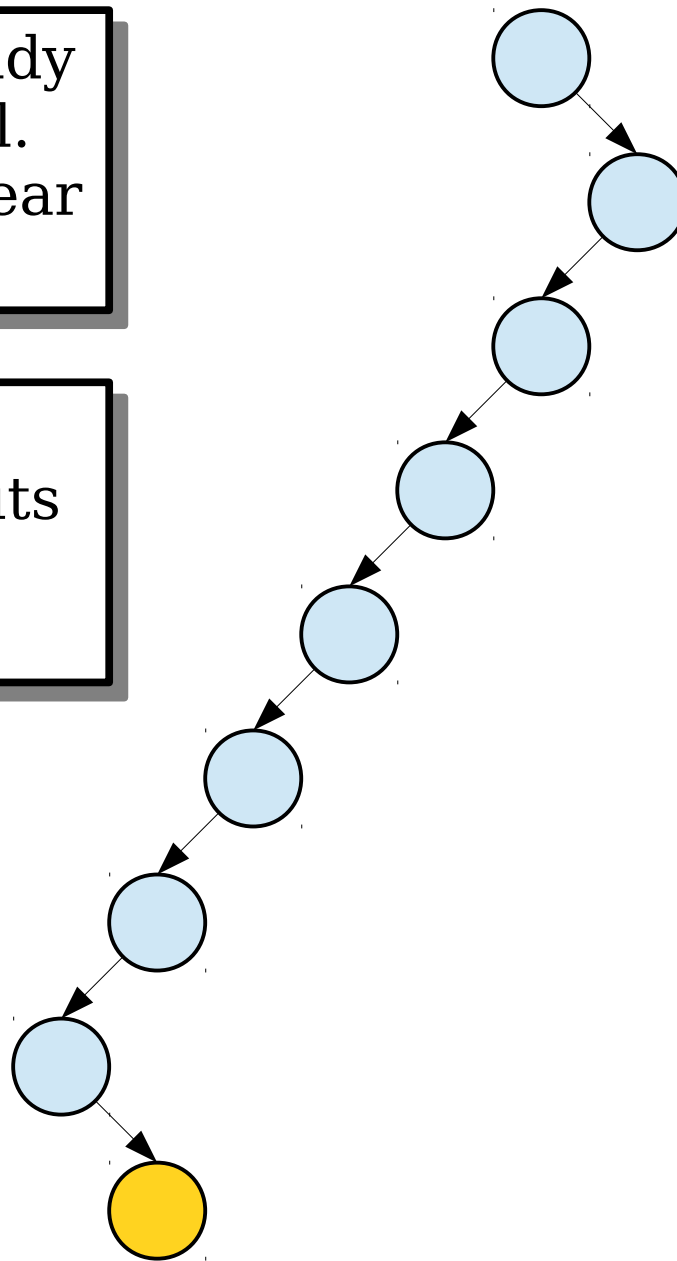
Intuition: We already handle zig-zags well. Let's just fix the linear case.



Question: How do we fix rotate-to-root to work well with long chains of nodes?

Intuition: We already handle zig-zags well. Let's just fix the linear case.

Observation: This new rule roughly cuts the height of the access path in half.

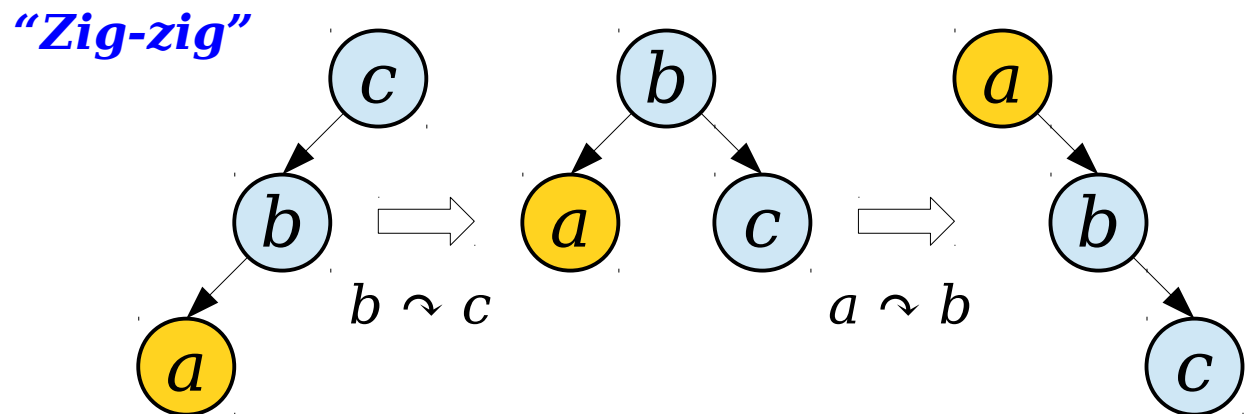
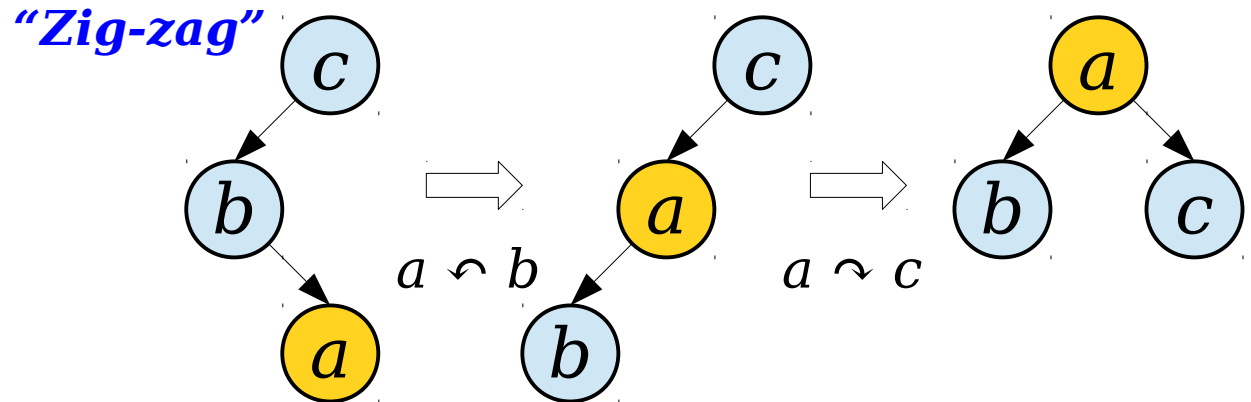
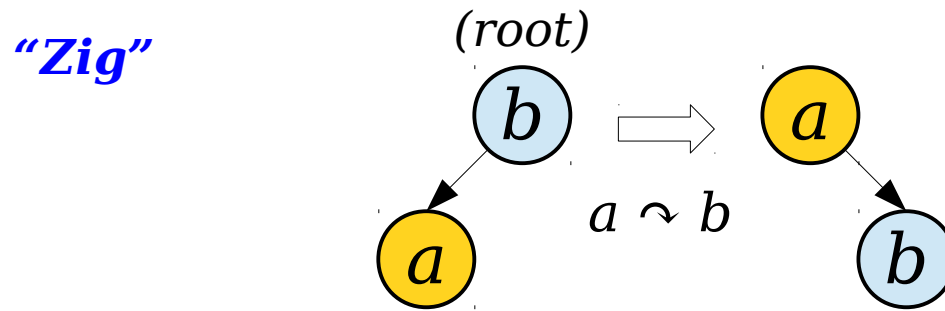


Question: How do we fix rotate-to-root to work well with long chains of nodes?

This procedure for moving a node to the root of the tree is called **splaying**.

Intuition: Use rotate-to-root, except when nodes chain in the same direction.

Mechanics: Look back two steps in the tree and apply the appropriate rotation rules.



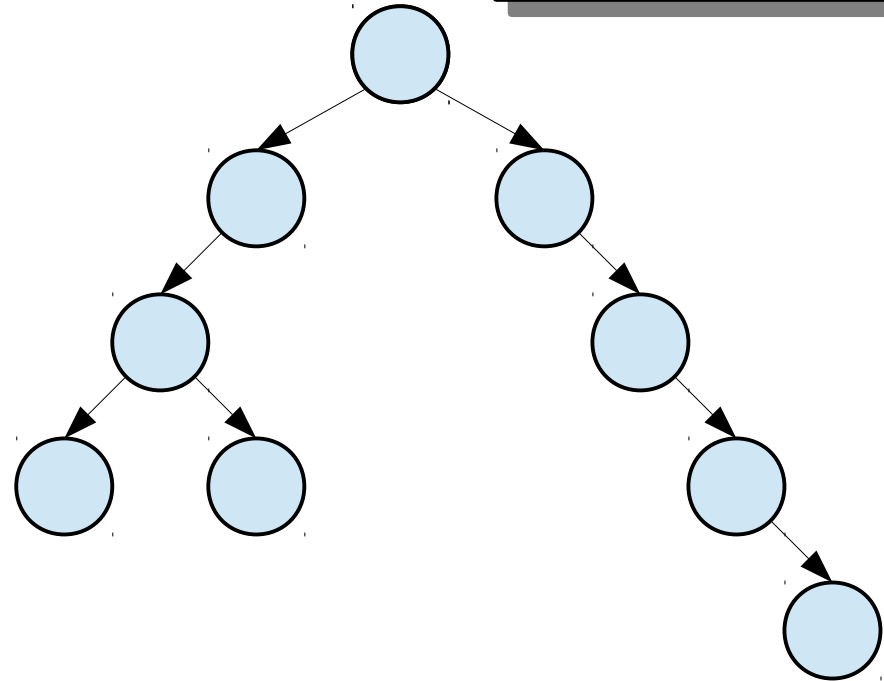
Question: How do we fix rotate-to-root to work well with long chains of nodes?

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Insert: Add as usual, then splay the new node.



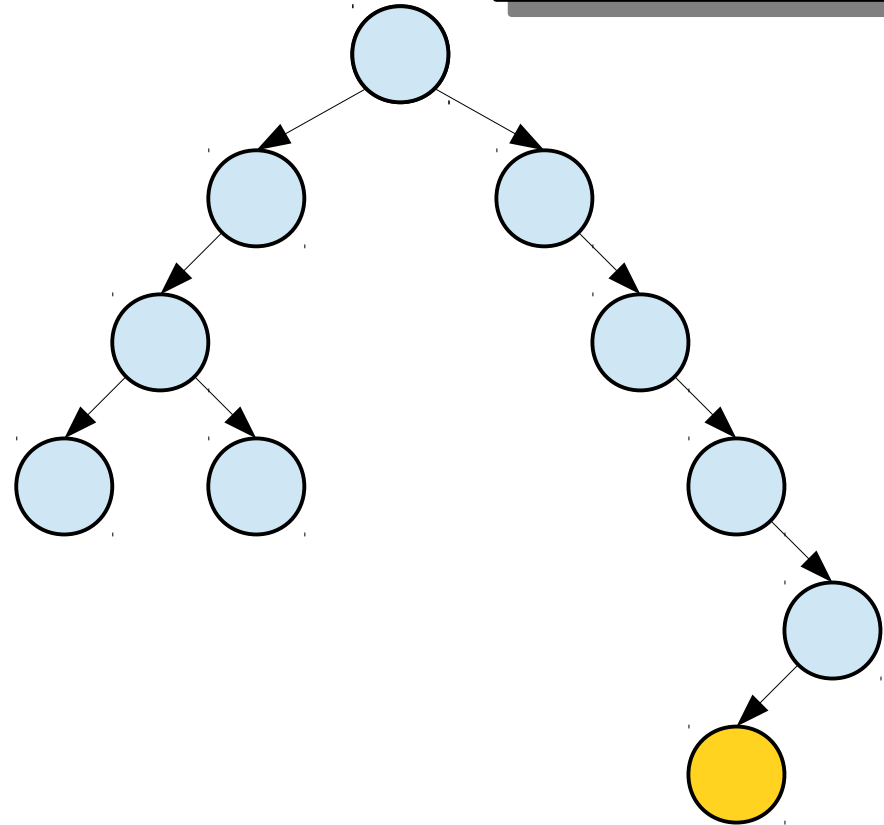
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Insert: Add as usual, then splay the new node.



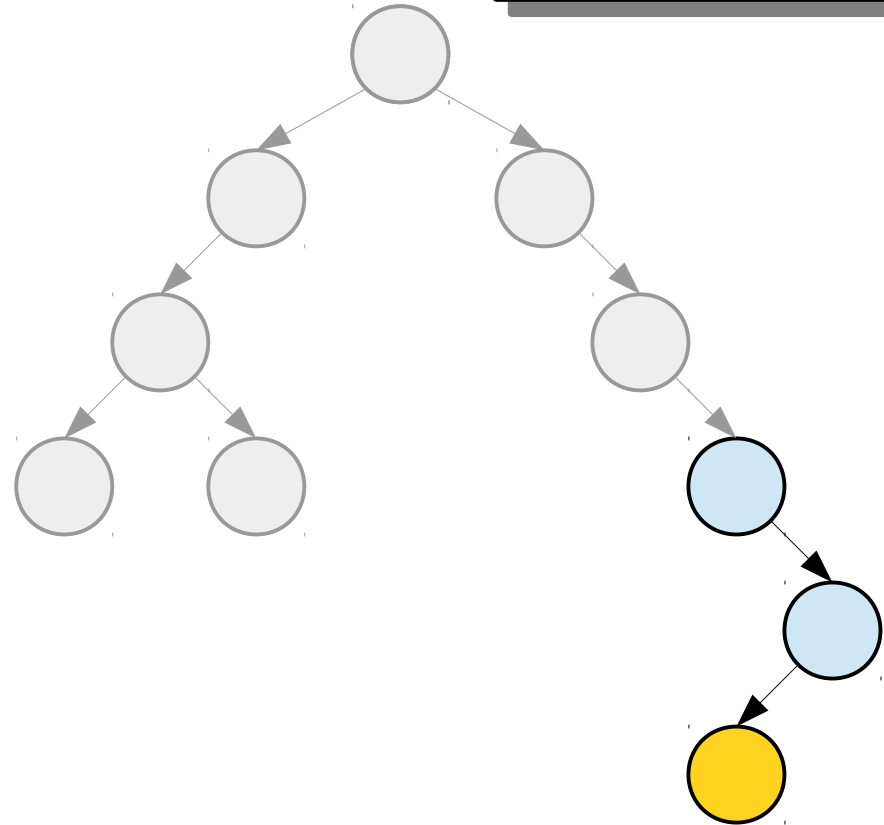
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Insert: Add as usual, then splay the new node.



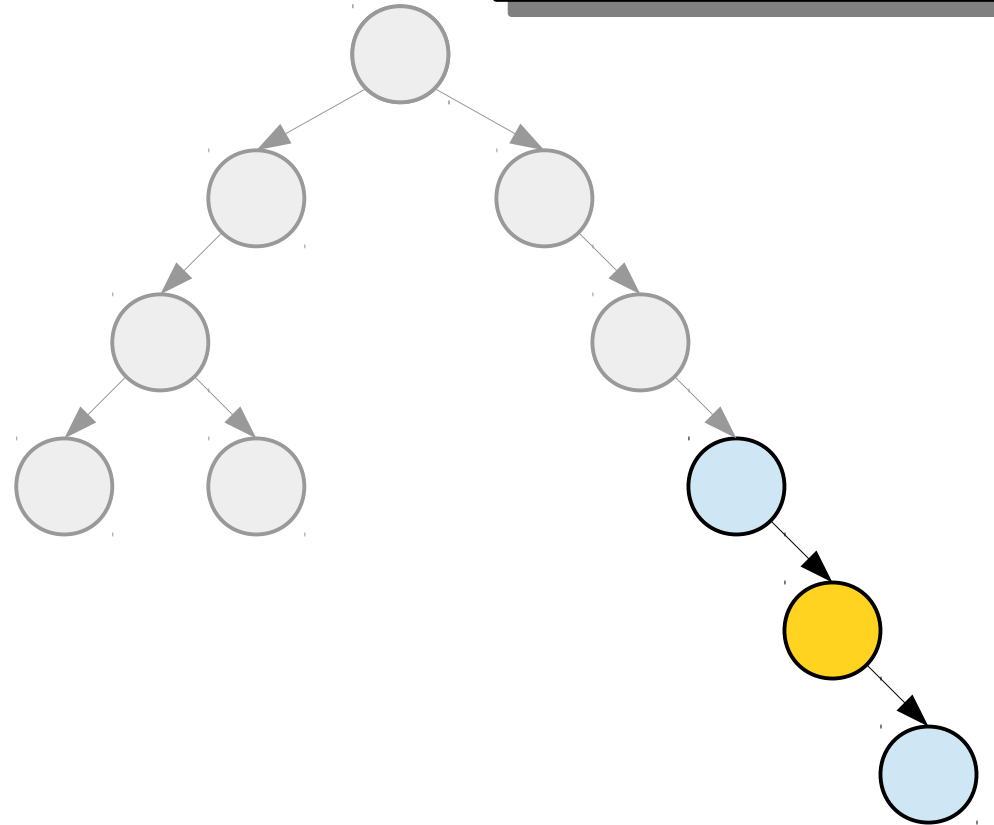
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Insert: Add as usual, then splay the new node.



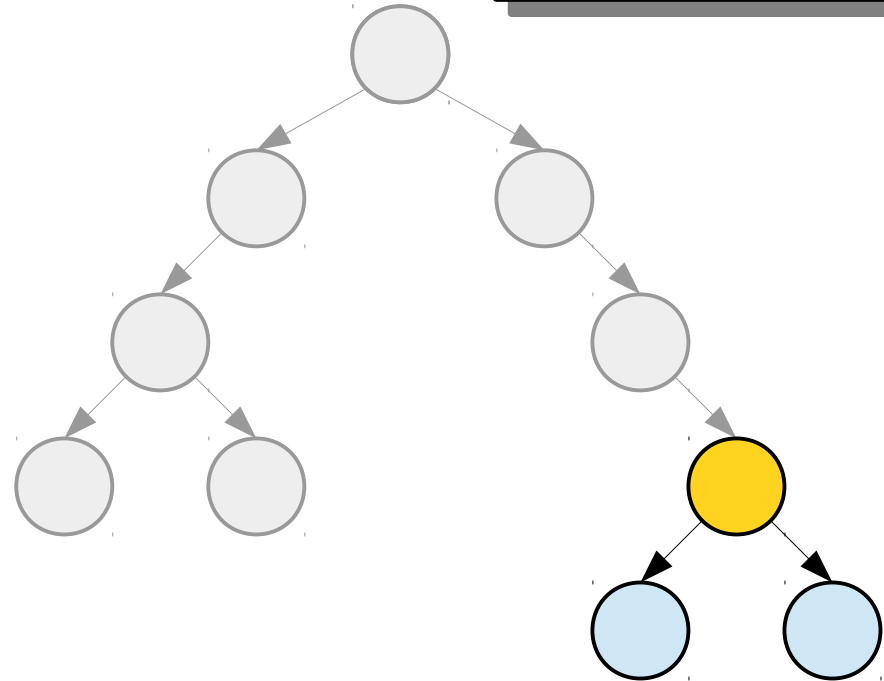
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Insert: Add as usual, then splay the new node.



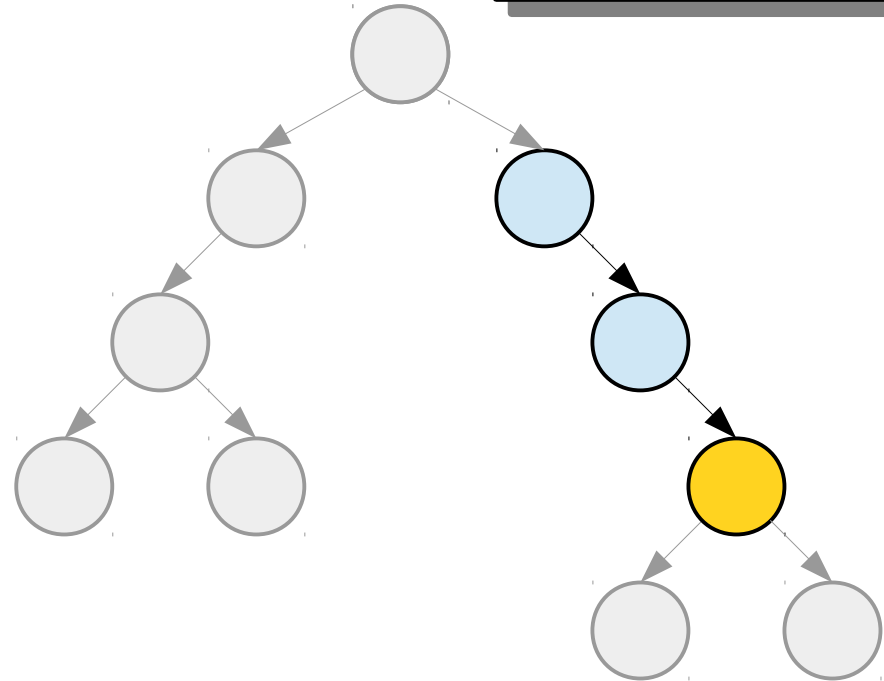
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Insert: Add as usual, then splay the new node.



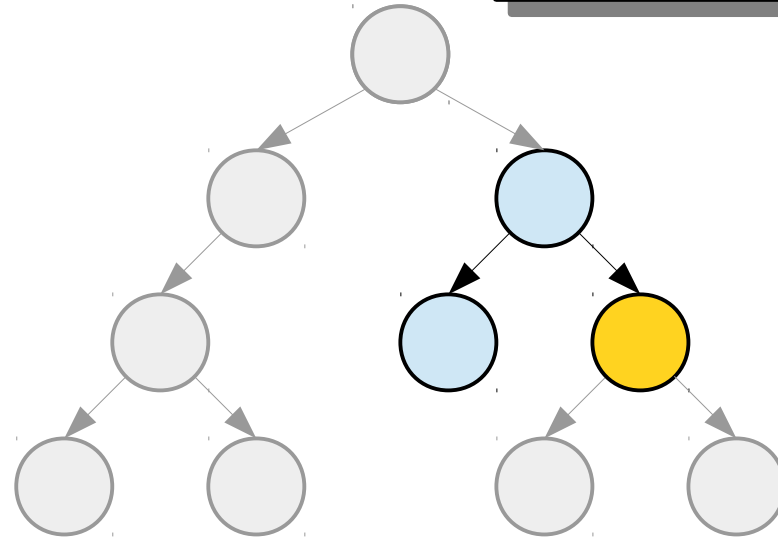
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Insert: Add as usual, then splay the new node.



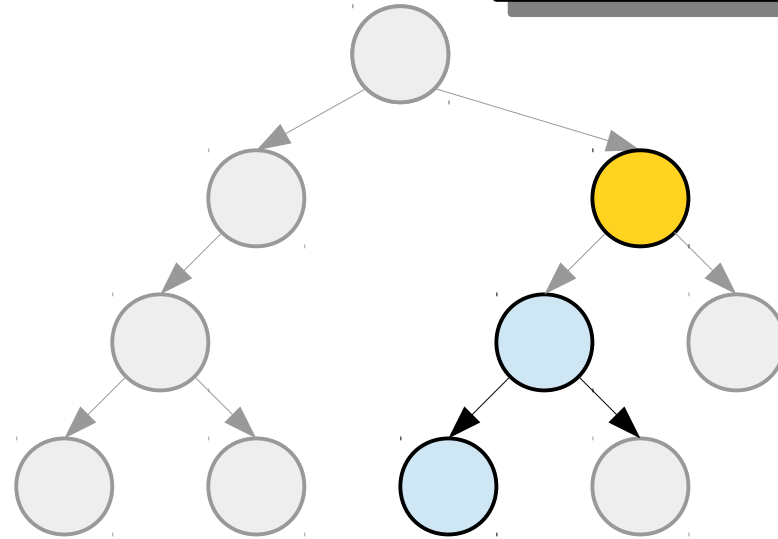
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Insert: Add as usual, then splay the new node.



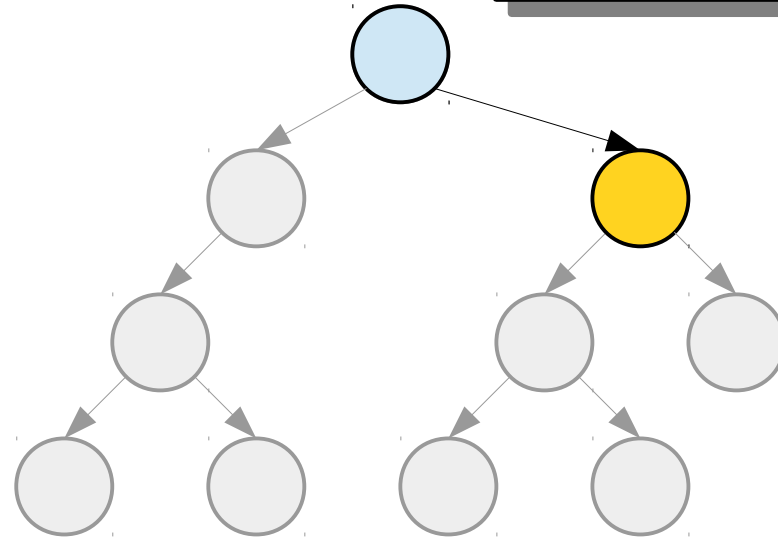
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Insert: Add as usual, then splay the new node.



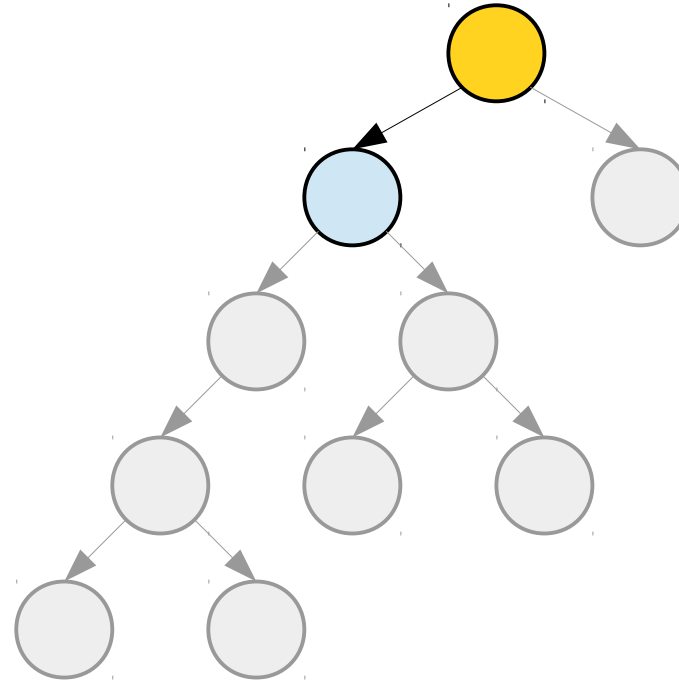
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Insert: Add as usual, then splay the new node.



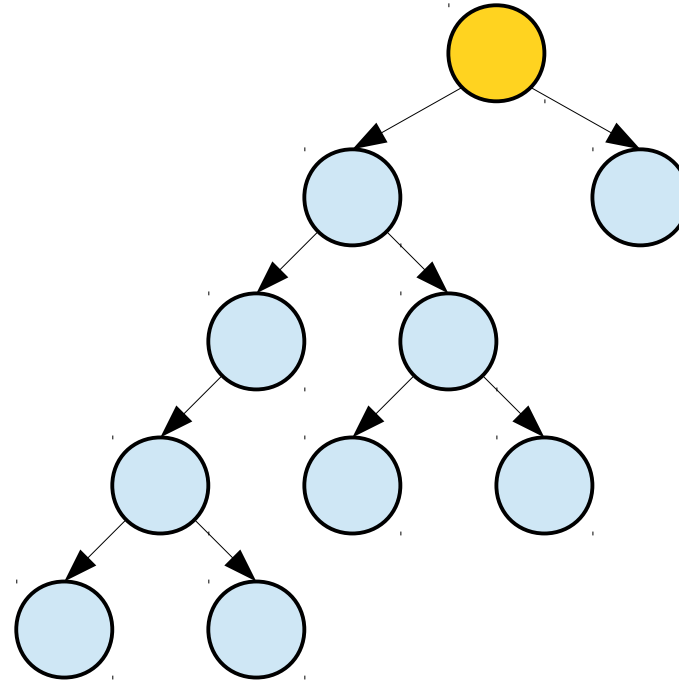
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Insert: Add as usual, then splay the new node.



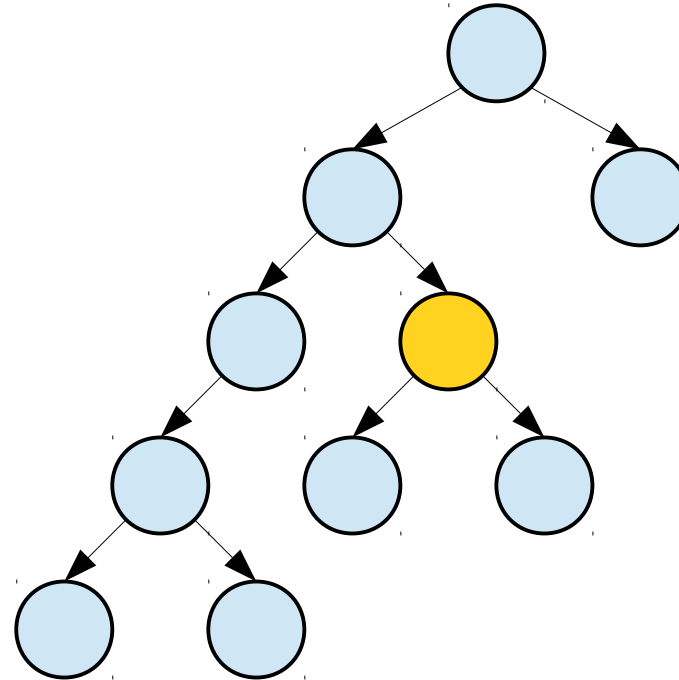
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Lookup: Search, then splay the last node seen.



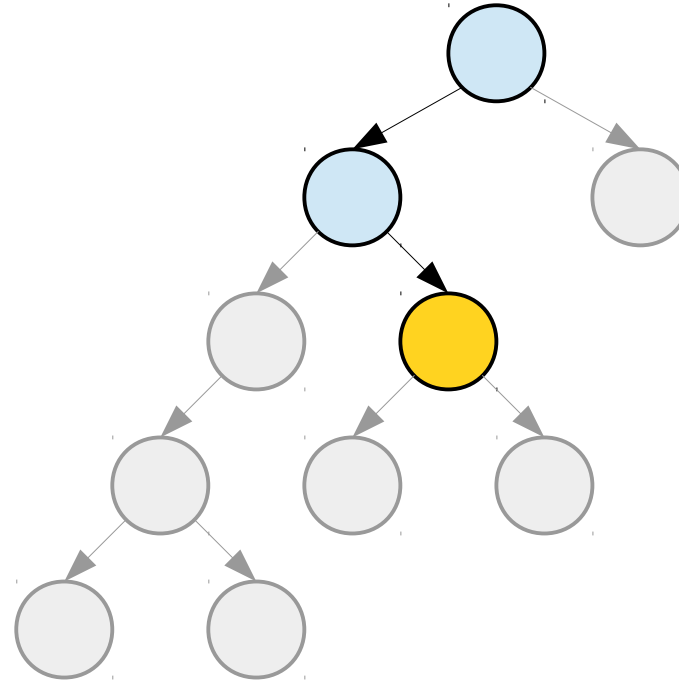
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Lookup: Search, then splay the last node seen.



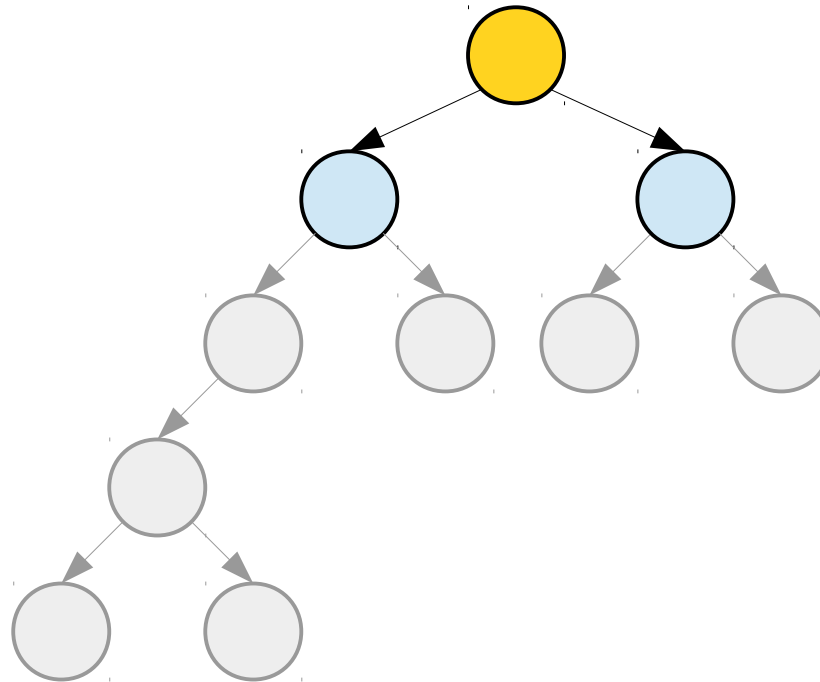
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Lookup: Search, then splay the last node seen.



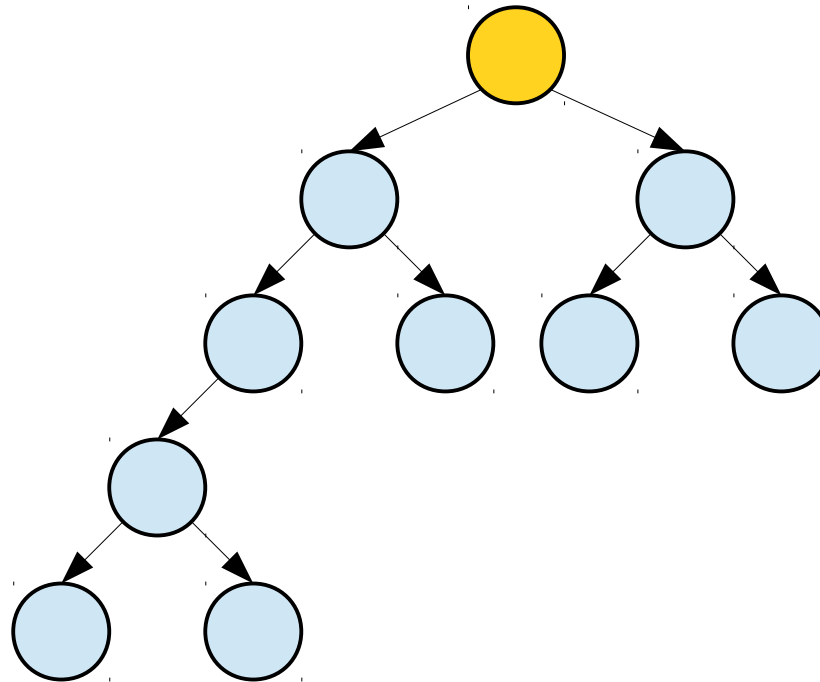
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Lookup: Search, then splay the last node seen.



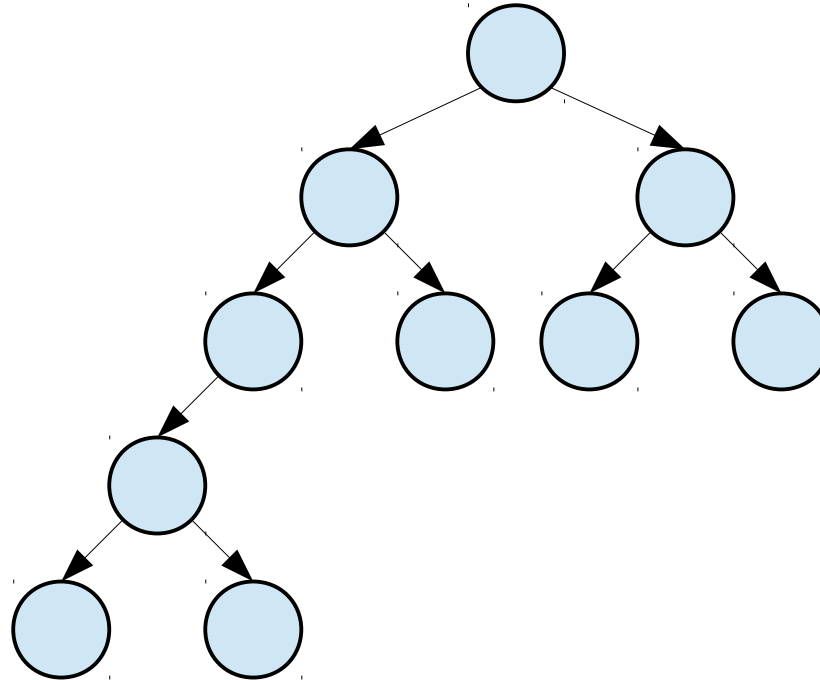
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Lookup: Search, then splay the last node seen.

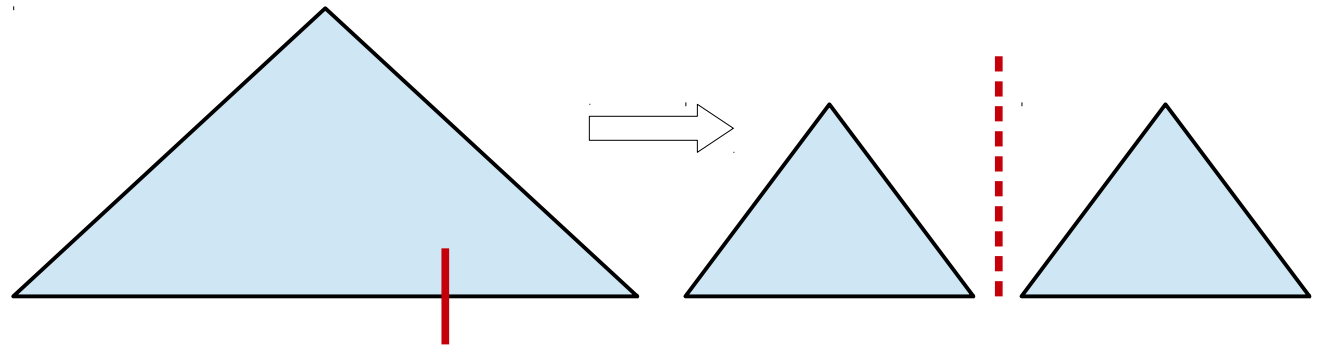


Splaying dramatically simplifies BST operations.

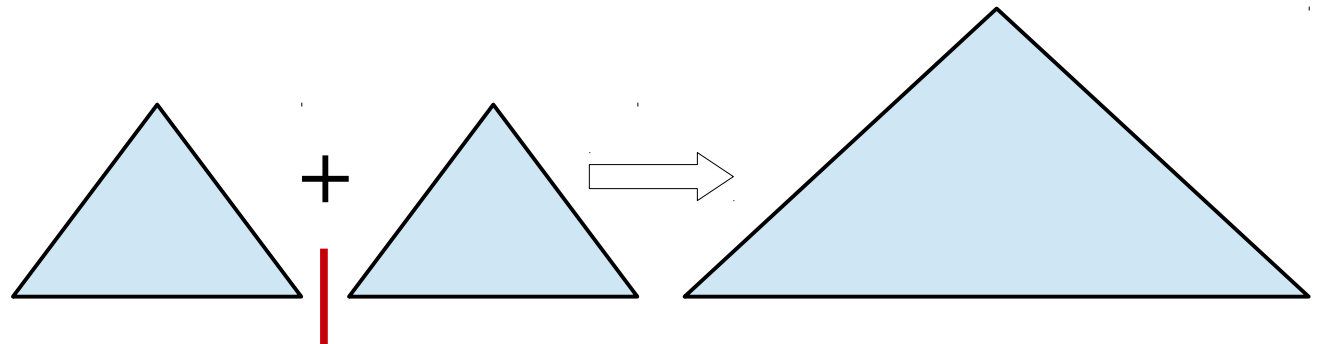
A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.



Split: How might you do this?



Join: How might you do this?

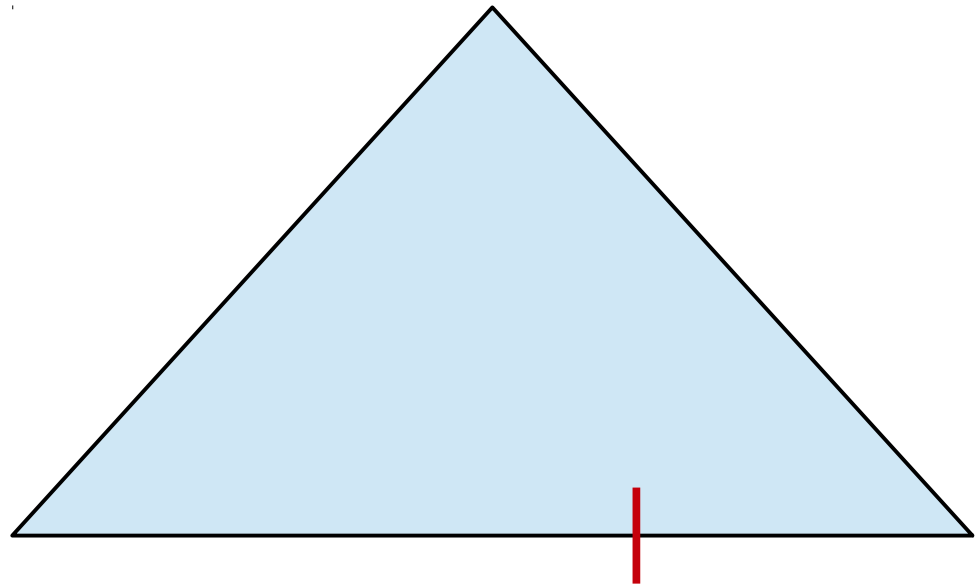
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Split: Search for the smallest value bigger than the split point. Splay it to the root and cut one link.



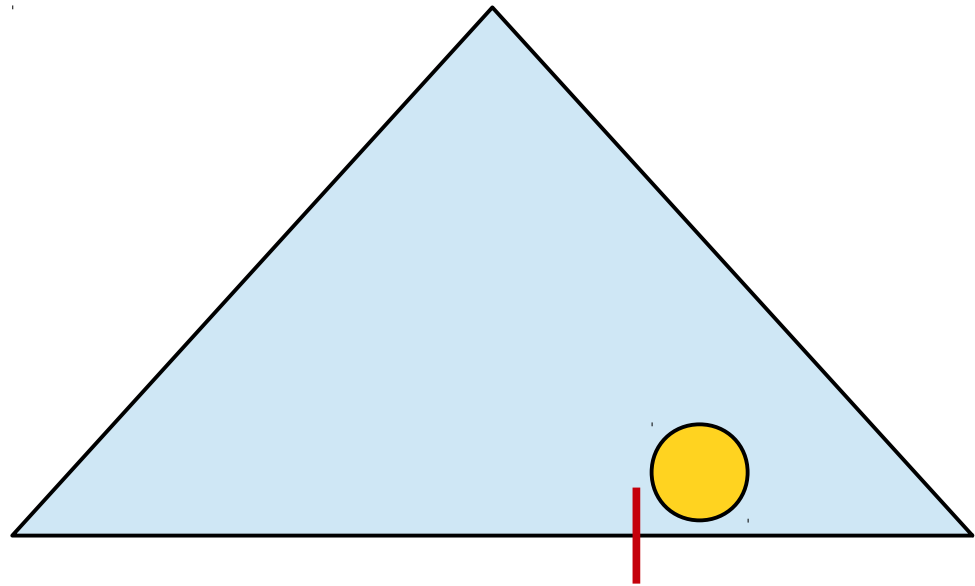
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Split: Search for the smallest value bigger than the split point. Splay it to the root and cut one link.



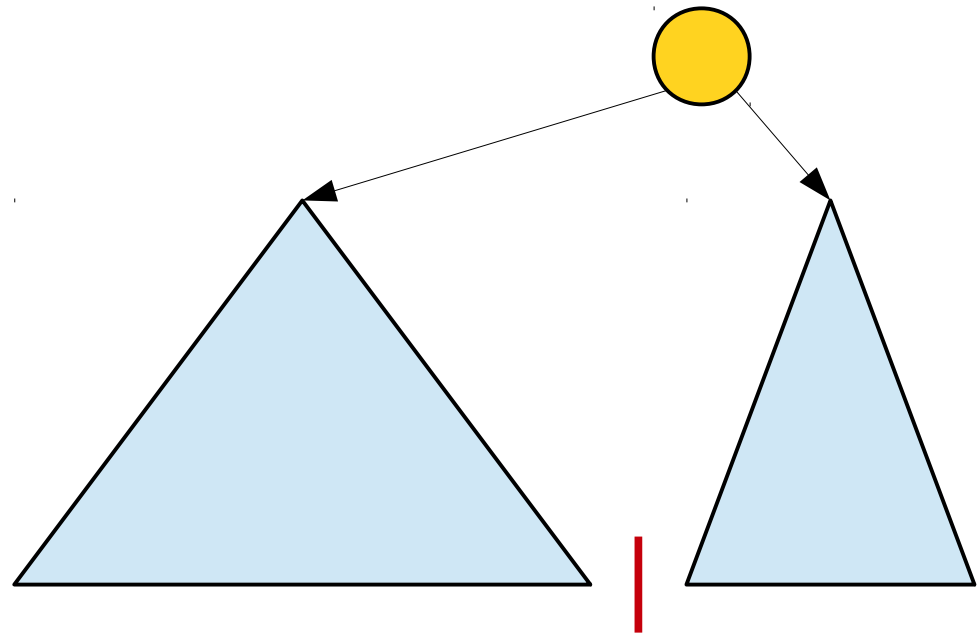
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Split: Search for the smallest value bigger than the split point. Splay it to the root and cut one link.



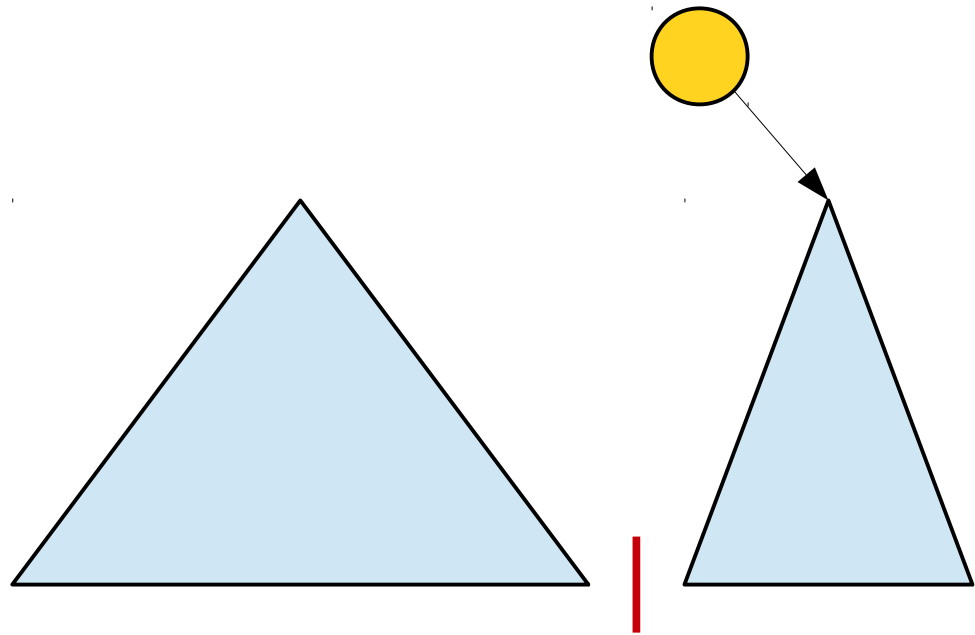
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Split: Search for the smallest value bigger than the split point. Splay it to the root and cut one link.



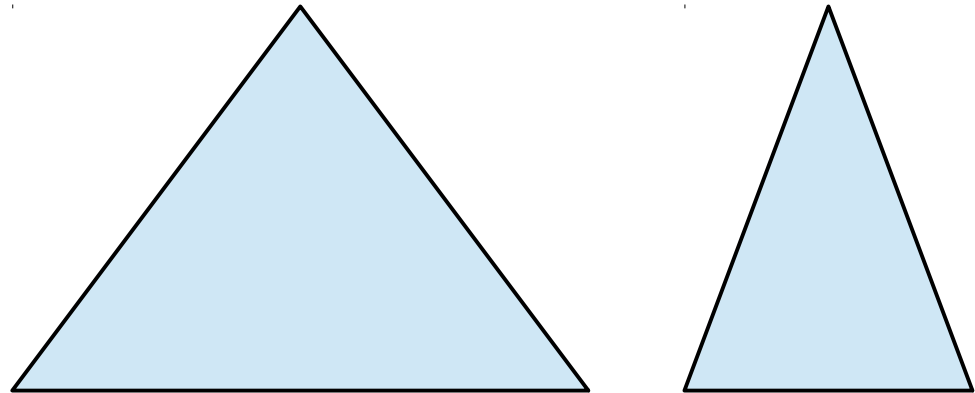
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Join: Splay the bigger value in the left tree to the root, then add the right tree as its right child.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.



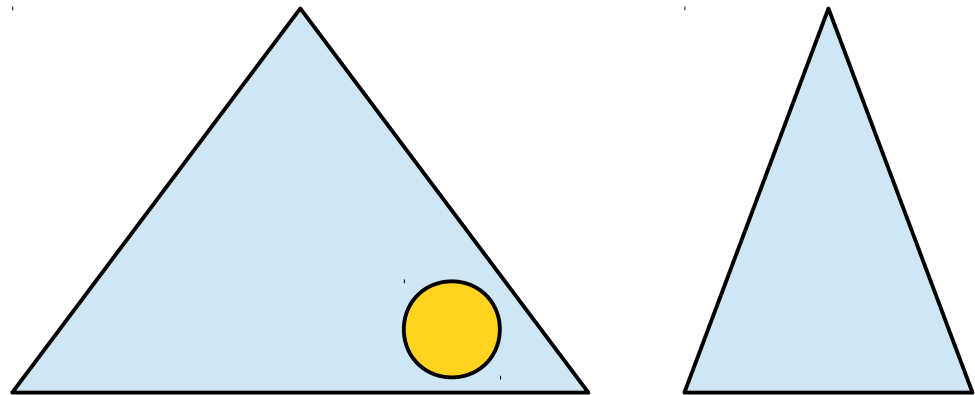
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Join: Splay the bigger value in the left tree to the root, then add the right tree as its right child.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.



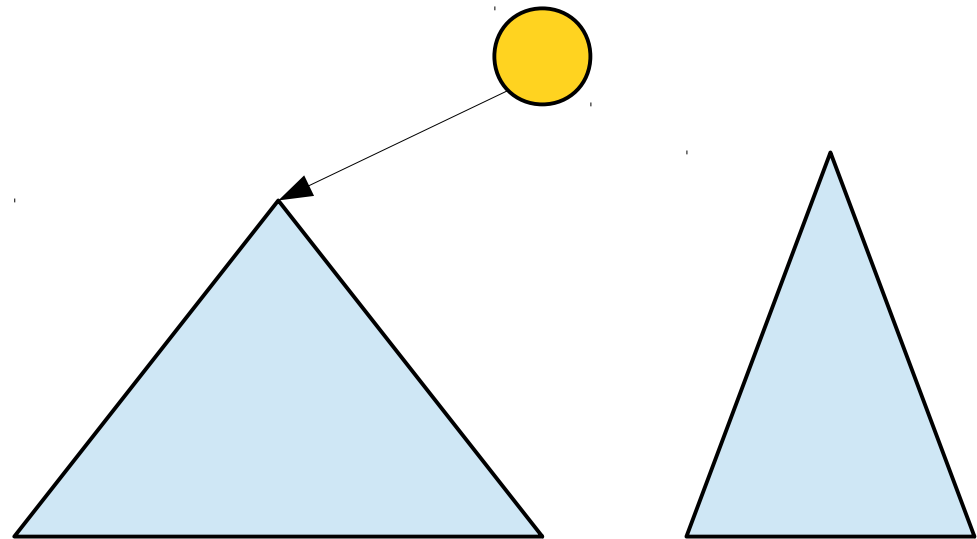
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Join: Splay the bigger value in the left tree to the root, then add the right tree as its right child.



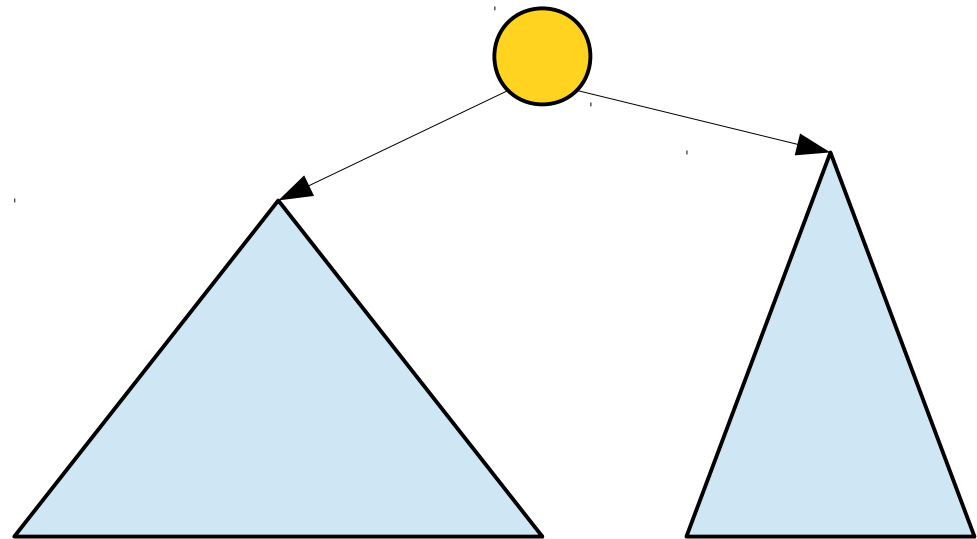
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Join: Splay the bigger value in the left tree to the root, then add the right tree as its right child.



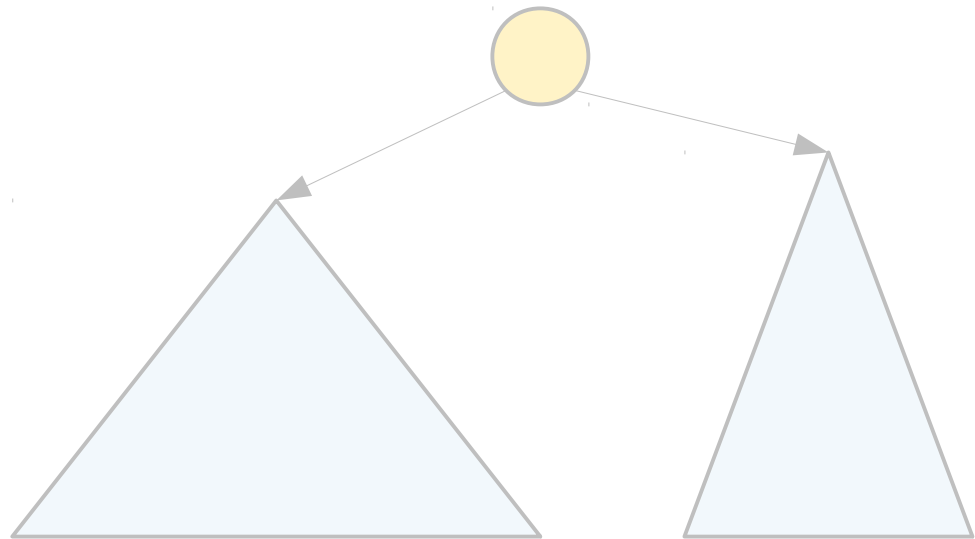
Splaying dramatically simplifies BST operations.

A **splay tree** is a regular BST where we splay the last node touched after each operation.

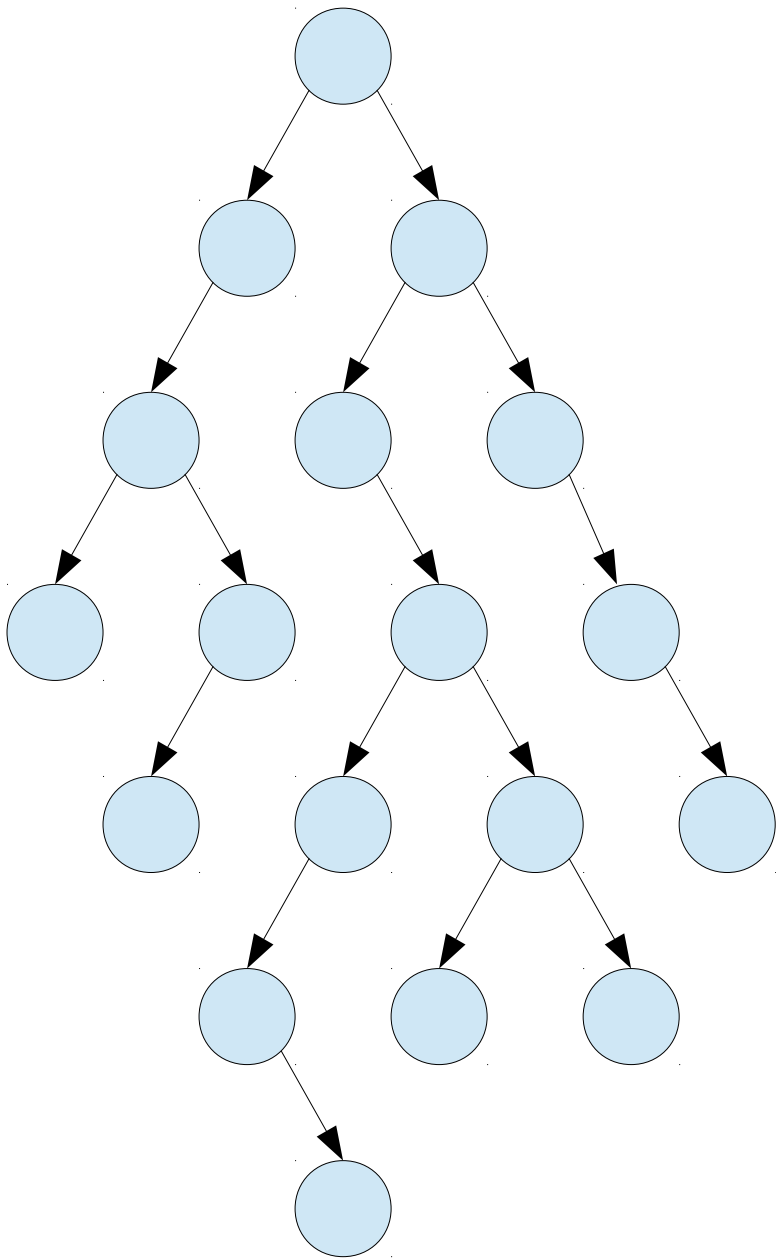
Theorem: The amortized cost of splaying a node is $O(\log n)$.

Claim: Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Join: Splay the bigger value in the left tree to the root, then add the right tree as its right child.

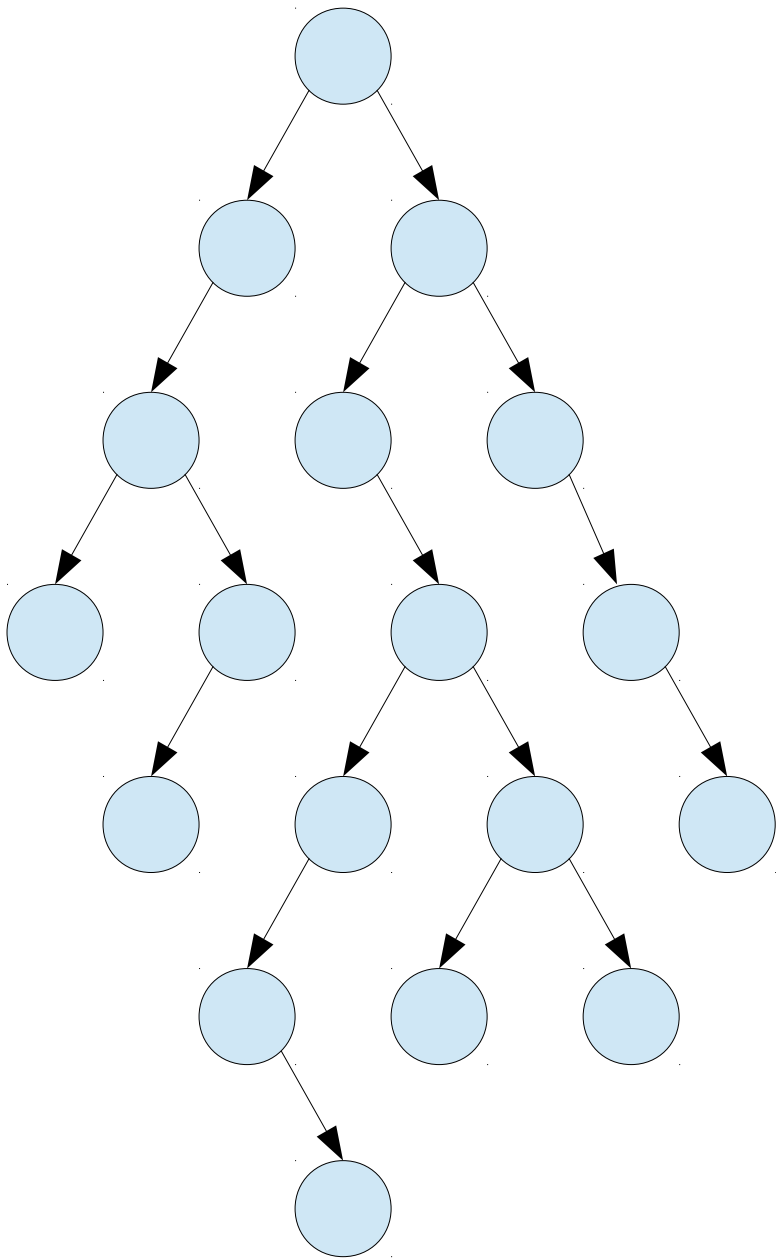


Splaying dramatically simplifies BST operations.



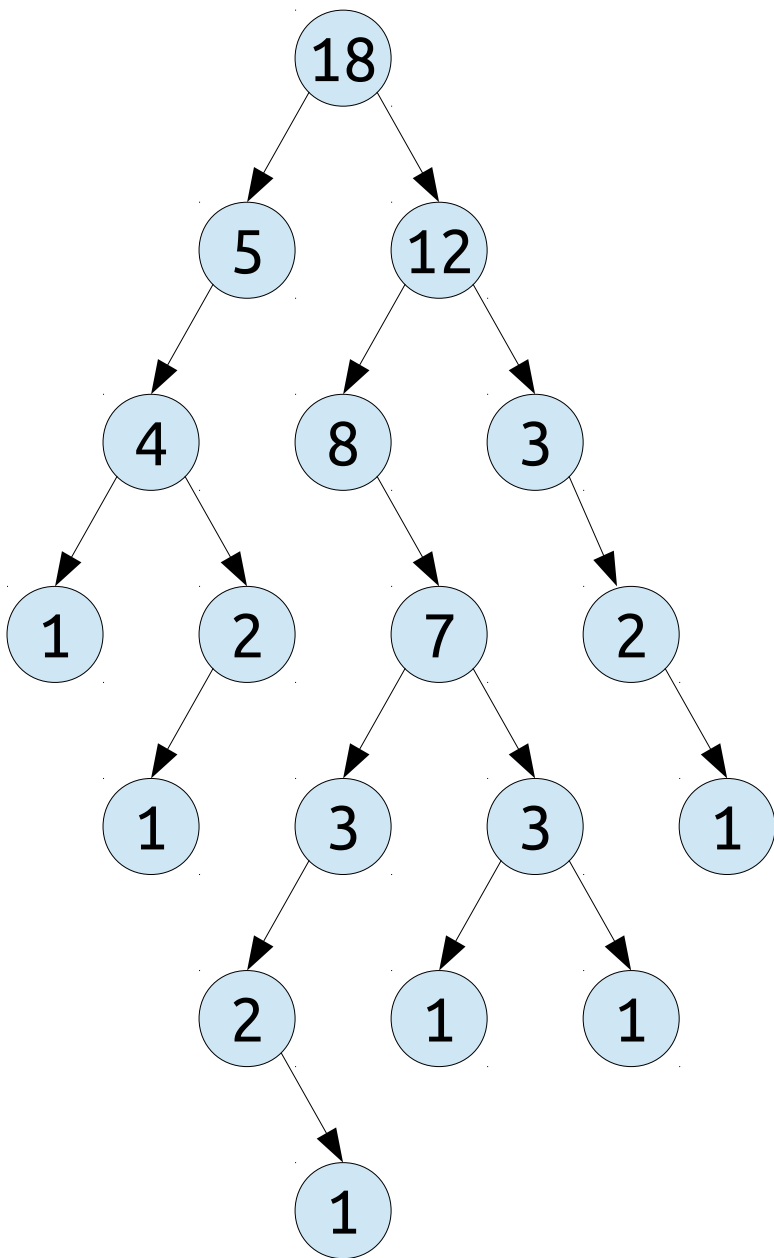
Question: Why is splaying fast?

Let $s(x)$ denote the number of nodes in the subtree rooted at x .

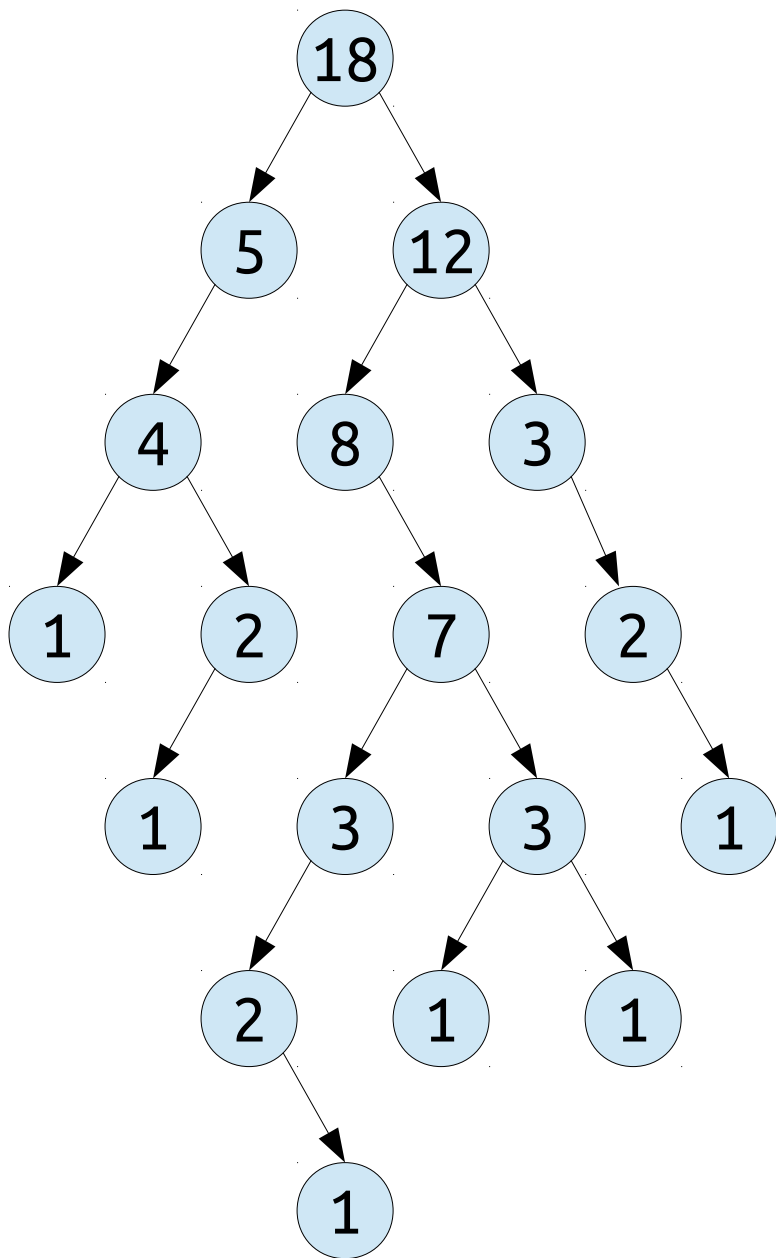


Question: Why is splaying fast?

Let $s(x)$ denote the number of nodes in the subtree rooted at x .





Question: Why is splaying fast?



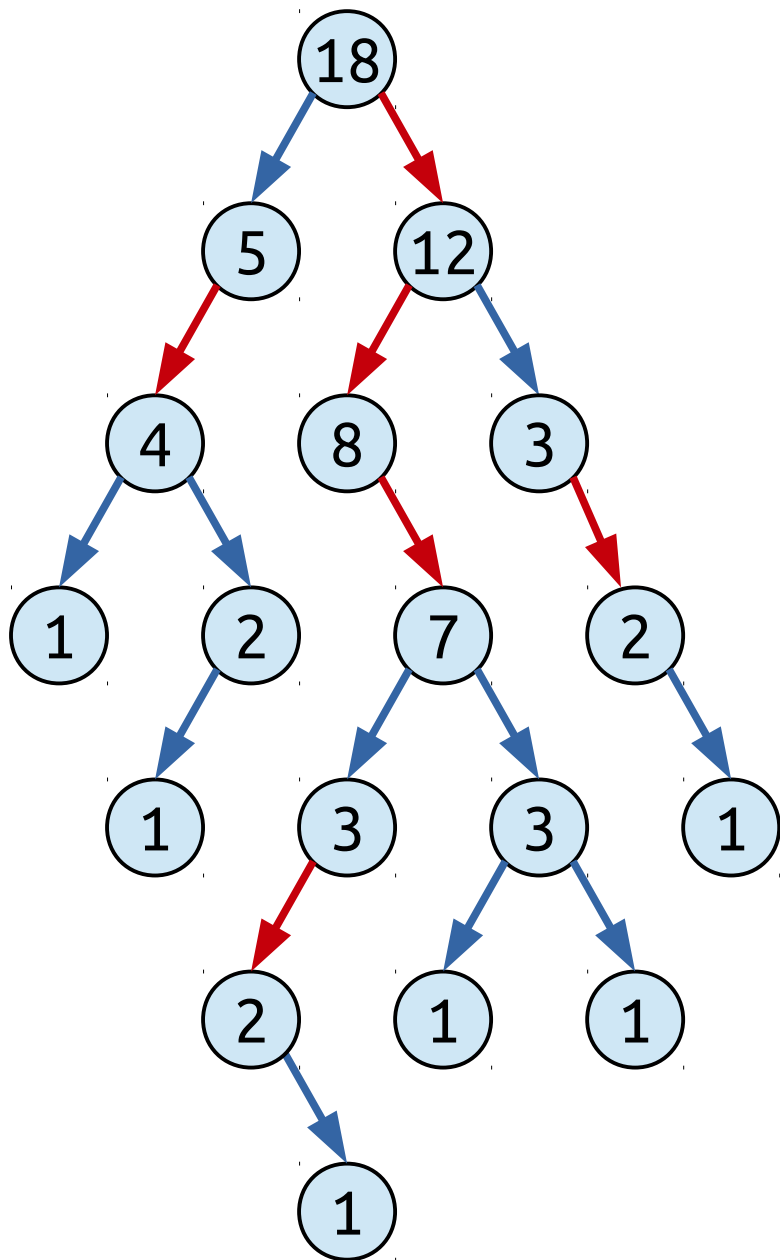
Let $s(x)$ denote the number of nodes in the subtree rooted at x .

Mark each edge as blue or red:

 $s(\text{child}) \leq \frac{1}{2} \cdot s(\text{parent})$
 $s(\text{child}) > \frac{1}{2} \cdot s(\text{parent})$



Blue edges make lots of progress.

Question: Why is splaying fast?



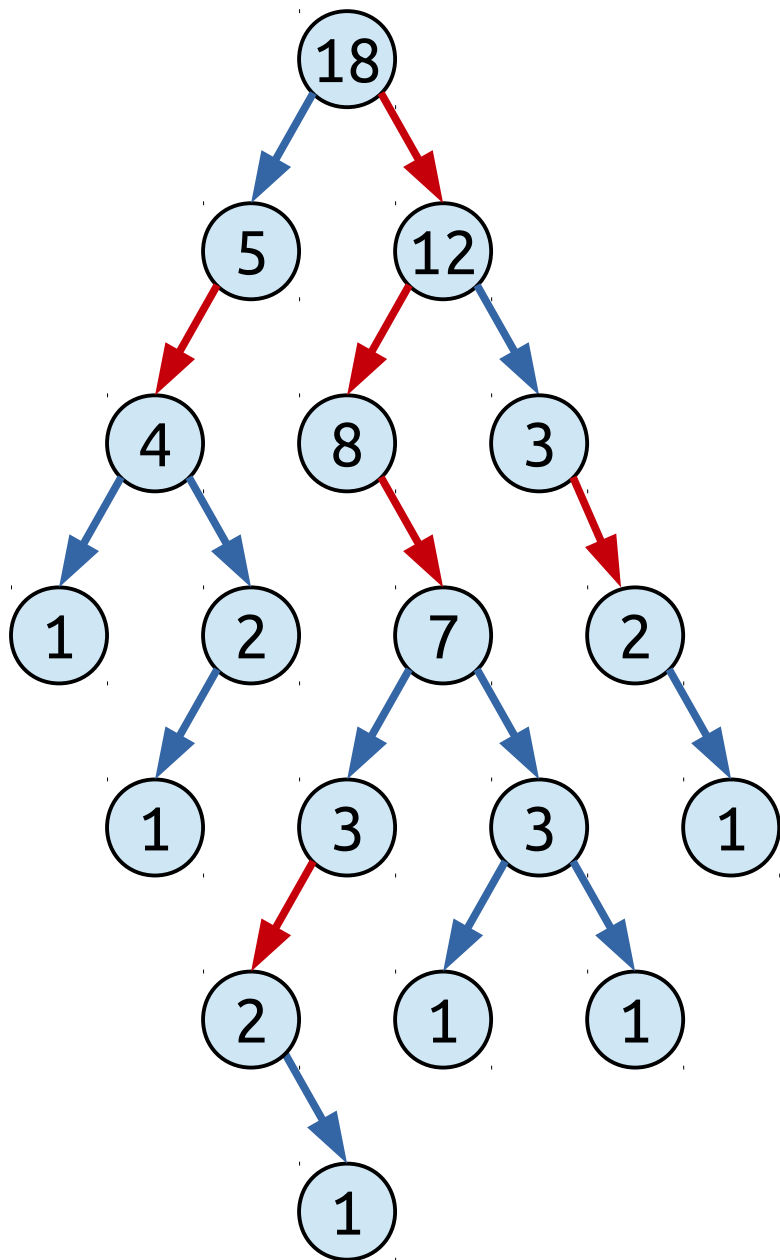
Let $s(x)$ denote the number of nodes in the subtree rooted at x .

Mark each edge as blue or red:

 $s(\text{child}) \leq \frac{1}{2} \cdot s(\text{parent})$
 $s(\text{child}) > \frac{1}{2} \cdot s(\text{parent})$



Blue edges make lots of progress.

Question: Why is splaying fast?



Let $s(x)$ denote the number of nodes in the subtree rooted at x .

Mark each edge as blue or red:

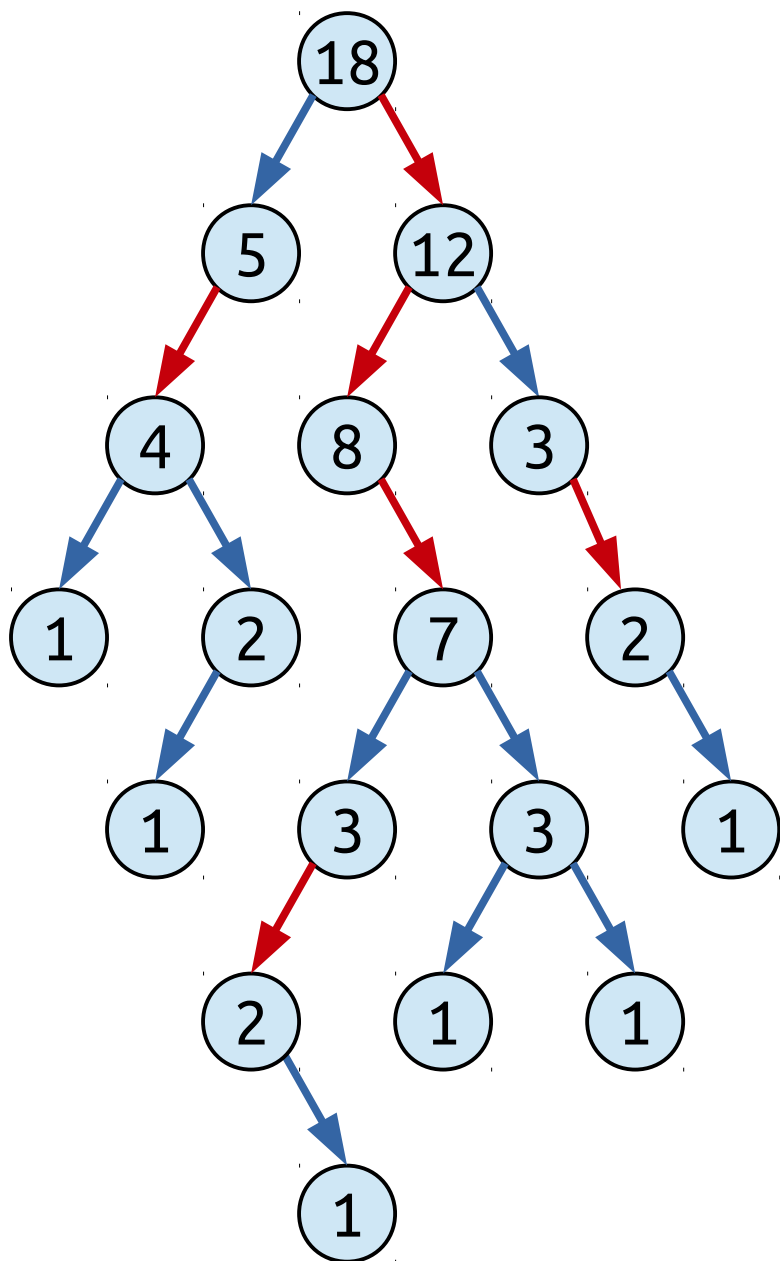
 $s(\text{child}) \leq \frac{1}{2} \cdot s(\text{parent})$
 $s(\text{child}) > \frac{1}{2} \cdot s(\text{parent})$

Blue edges make lots of progress.

Cost of visiting a node:



$O(\text{\#blue-used} + \text{\#red-used})$

Question: Why is splaying fast?



Let $s(x)$ denote the number of nodes in the subtree rooted at x .

Mark each edge as blue or red:

 $s(\text{child}) \leq \frac{1}{2} \cdot s(\text{parent})$
 $s(\text{child}) > \frac{1}{2} \cdot s(\text{parent})$

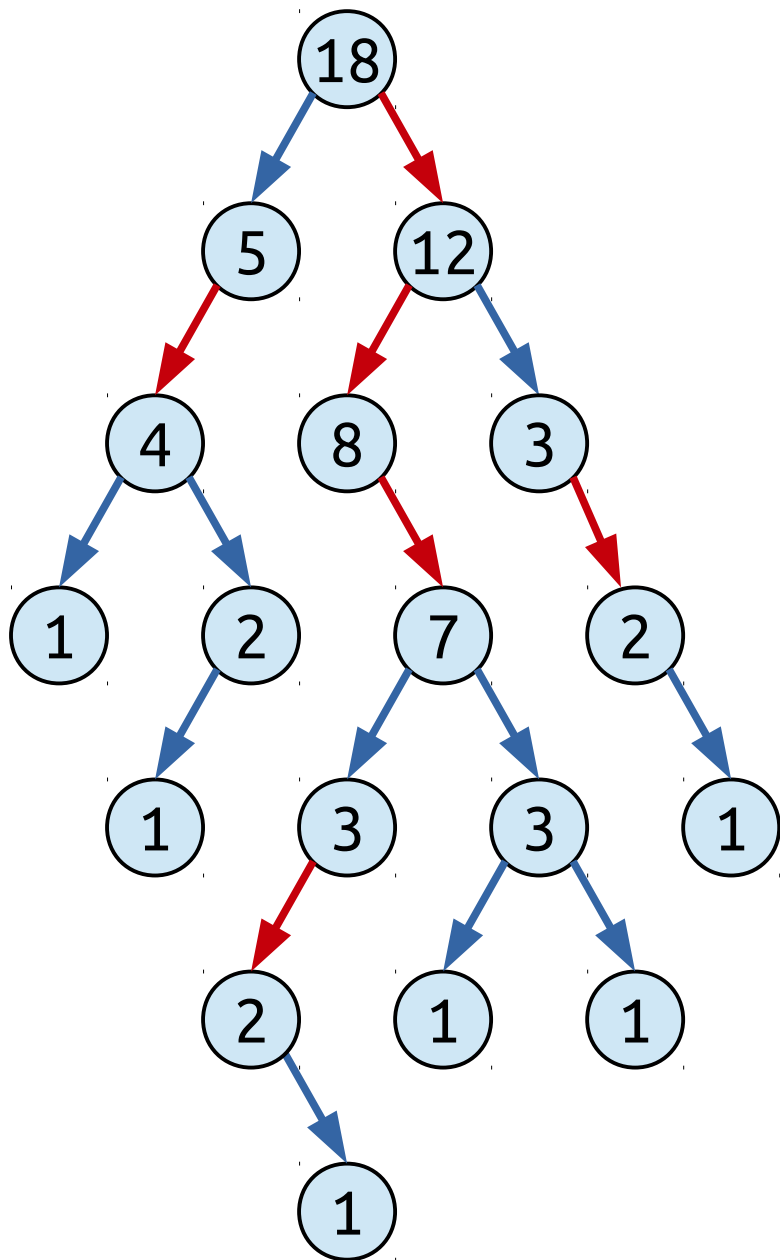
Blue edges make lots of progress.

Cost of visiting a node:

$O(\text{\#blue-used} + \text{\#red-used})$



Idea: Bound the cost of blue edges, then amortize away the cost of red edges. This is called a **heavy/light decomposition**.

Question: Why is splaying fast?



Let $s(x)$ denote the number of nodes in the subtree rooted at x .

Mark each edge as blue or red:

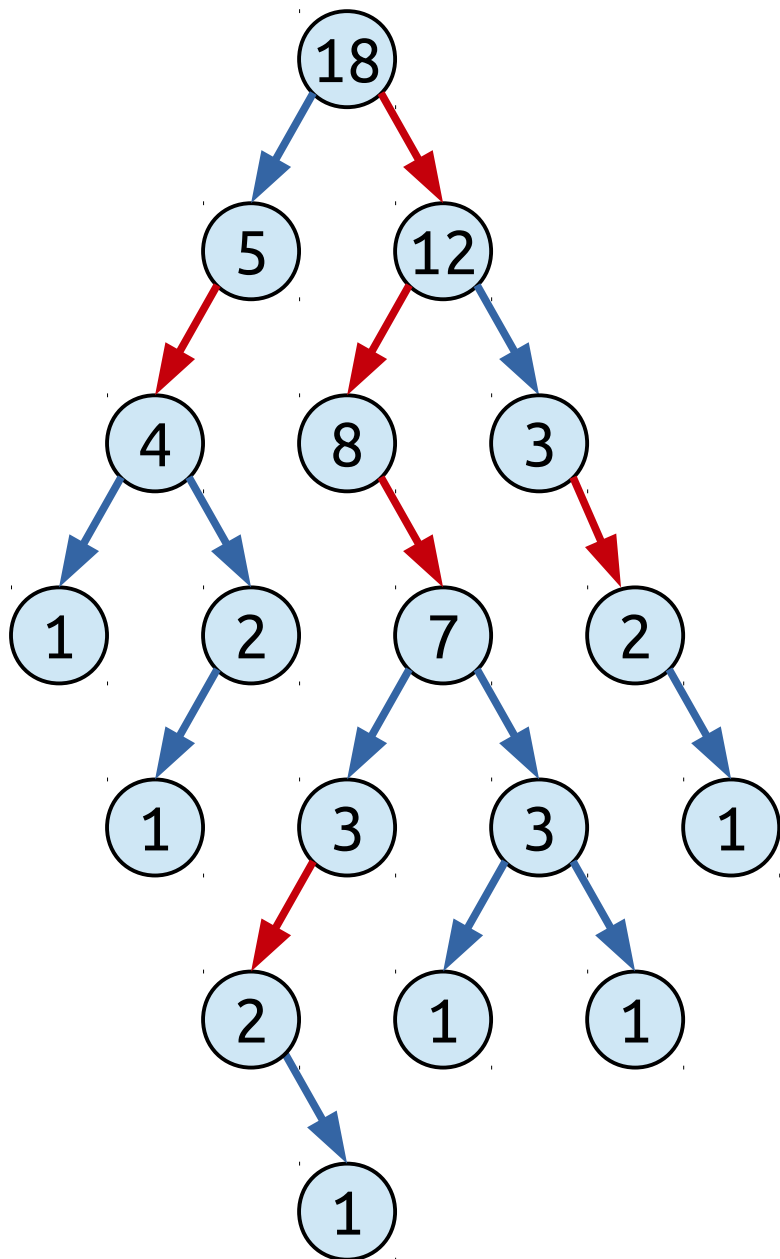
 $s(\text{child}) \leq \frac{1}{2} \cdot s(\text{parent})$
 $s(\text{child}) > \frac{1}{2} \cdot s(\text{parent})$

Blue edges make lots of progress.

Cost of visiting a node:



$O(\text{\#blue-used} + \text{\#red-used})$

Question: Why is splaying fast?



Let $s(x)$ denote the number of nodes in the subtree rooted at x .

Mark each edge as blue or red:

 $s(\text{child}) \leq \frac{1}{2} \cdot s(\text{parent})$
 $s(\text{child}) > \frac{1}{2} \cdot s(\text{parent})$

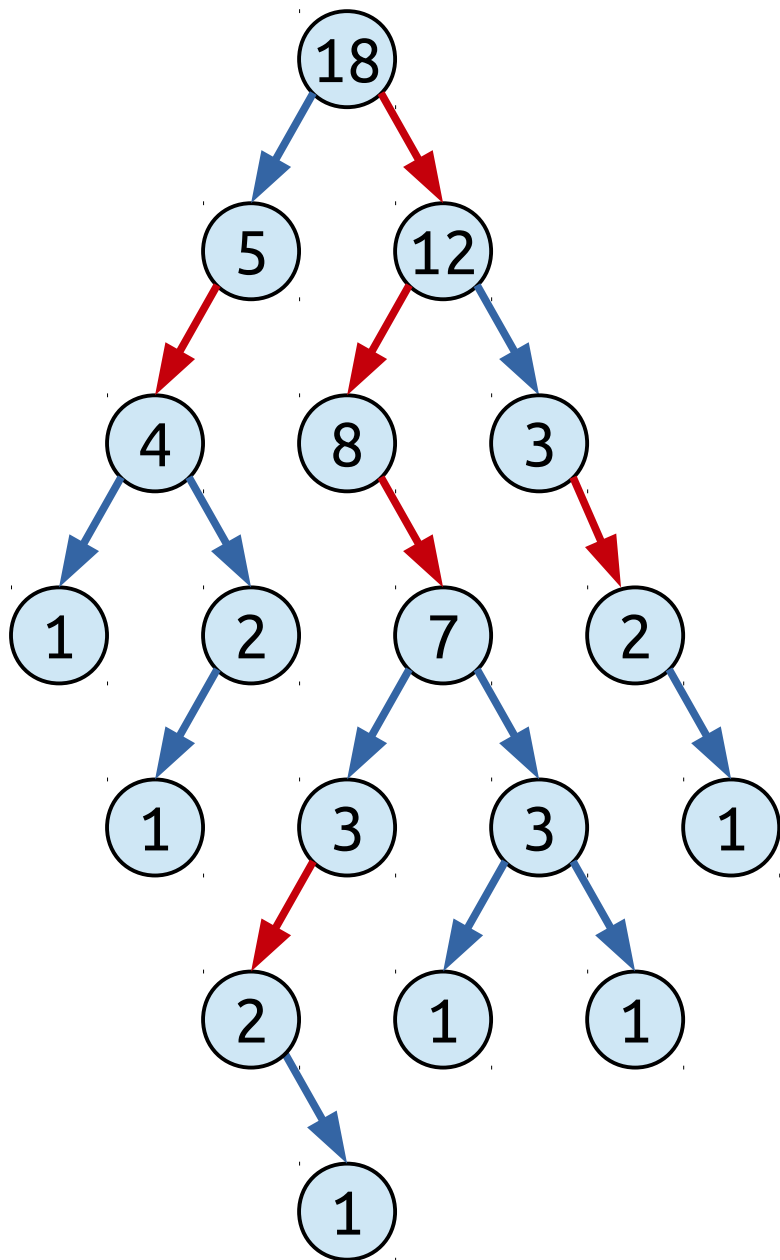
Blue edges make lots of progress.

Cost of visiting a node:

$O(\log n + \text{\#red-used})$



Intuition: Blue edges discard half the remaining nodes. You can only do that $O(\log n)$ times before running out of nodes.

Question: Why is splaying fast?



Let $s(x)$ denote the number of nodes in the subtree rooted at x .

Mark each edge as blue or red:

 $s(\text{child}) \leq \frac{1}{2} \cdot s(\text{parent})$
 $s(\text{child}) > \frac{1}{2} \cdot s(\text{parent})$

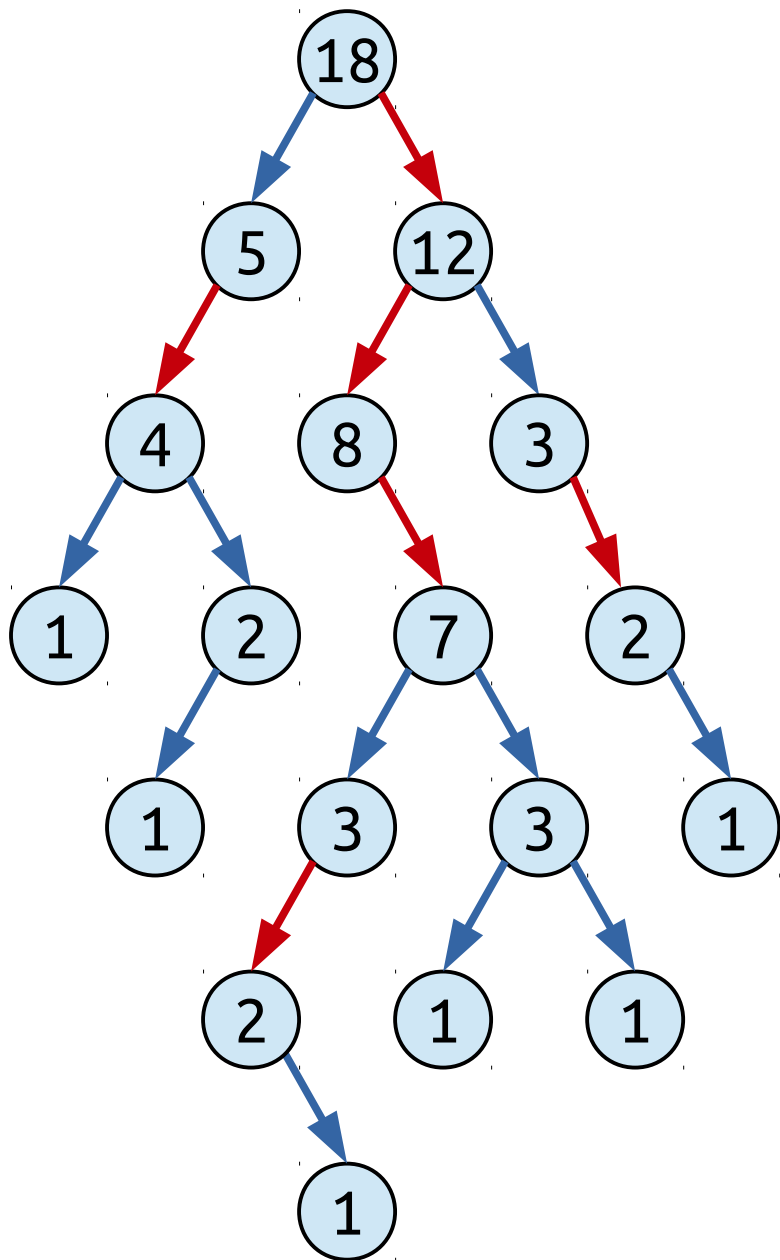
Blue edges make lots of progress.

Cost of visiting a node:

$O(\log n + \text{\#red-used})$

Goal: Find a potential function that penalizes red edges and rewards blue edges.

Question: Why is splaying fast?



Let $s(x)$ denote the number of nodes in the subtree rooted at x .

Mark each edge as blue or red:

\rightarrow blue $\lg s(\text{child}) \leq \lg s(\text{parent}) - 1$
 \rightarrow red $\lg s(\text{child}) > \lg s(\text{parent}) - 1$

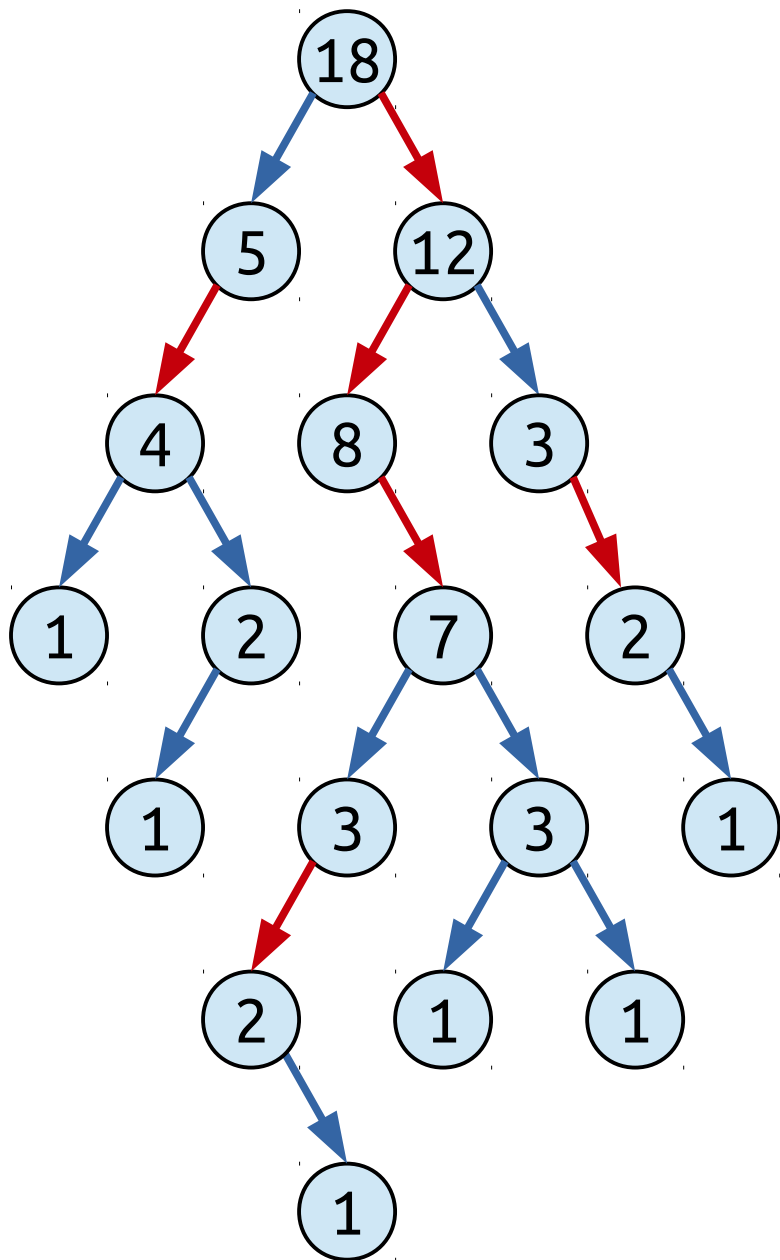
Blue edges make lots of progress.

Cost of visiting a node:

$O(\log n + \text{\#red-used})$

Goal: Find a potential function that penalizes red edges and rewards blue edges.

Question: Why is splaying fast?



Let $s(x)$ denote the number of nodes in the subtree rooted at x .

Mark each edge as blue or red:

→ $\lg s(\text{child}) \leq \lg s(\text{parent}) - 1$
→ $\lg s(\text{child}) > \lg s(\text{parent}) - 1$

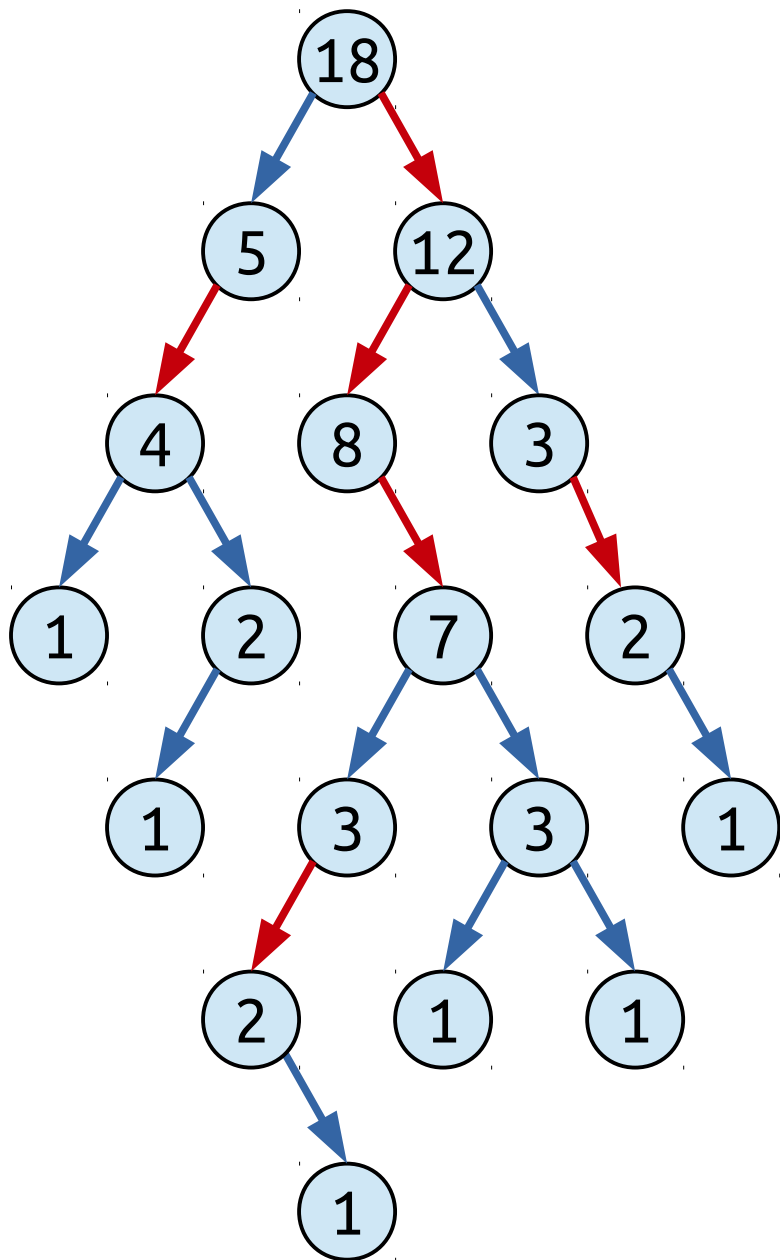
Blue edges make lots of progress.

Cost of visiting a node:

$O(\log n + \text{\#red-used})$

Observation: If there are a lot of red edges, then $\lg s(x)$ will frequently be large.

Question: Why is splaying fast?



Let $s(x)$ denote the number of nodes in the subtree rooted at x .

Mark each edge as blue or red:

- $\lg s(\text{child}) \leq \lg s(\text{parent}) - 1$
- $\lg s(\text{child}) > \lg s(\text{parent}) - 1$

Blue edges make lots of progress.

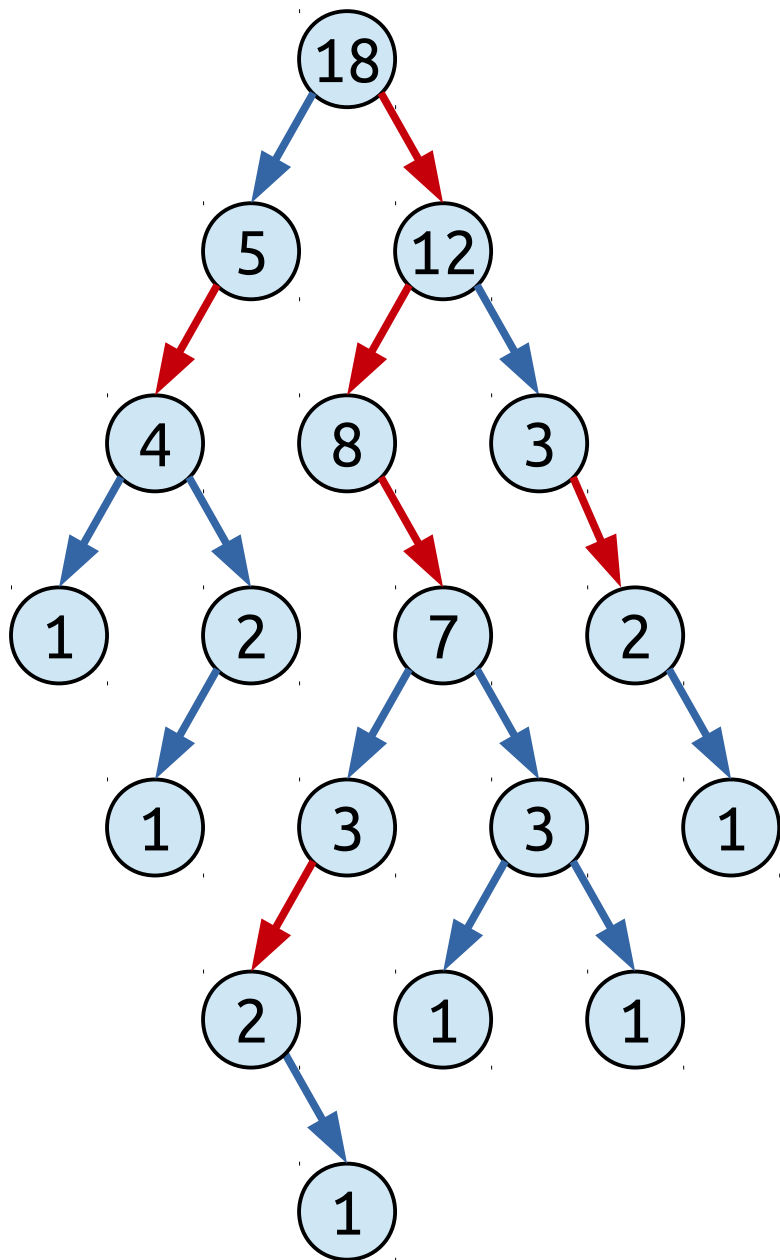
Cost of visiting a node:

$$O(\log n + \text{\#red-used})$$

Choose our potential to be

$$\Phi = \sum_{i=1}^n \lg s(x_i).$$

Question: Why is splaying fast?



Cost of visiting a node:

$$O(\log n + \text{\#red-used})$$

Choose our potential to be

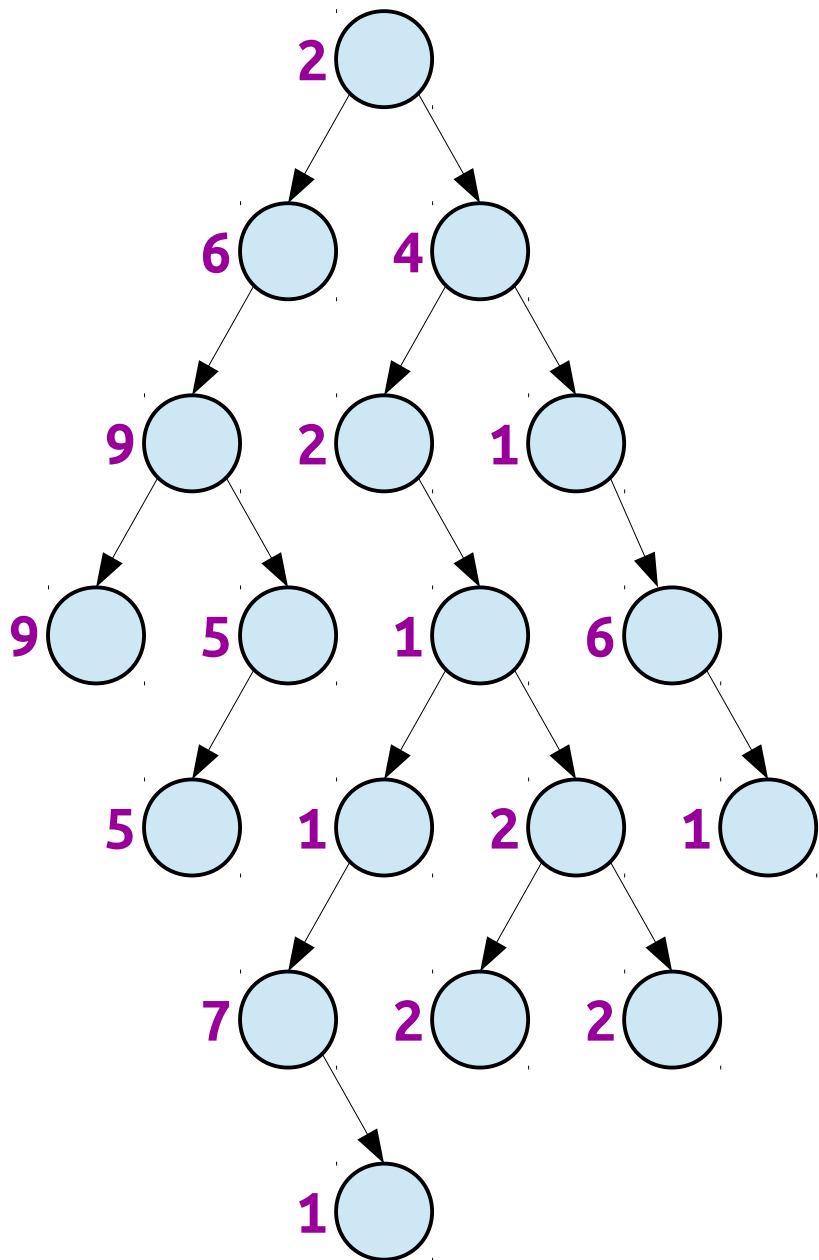
$$\Phi = \sum_{i=1}^n \lg s(x_i).$$

Proving Φ amortizes away the **\#red-used** term is tricky and the analysis doesn't generalize well. See the Sleator-Tarjan paper.

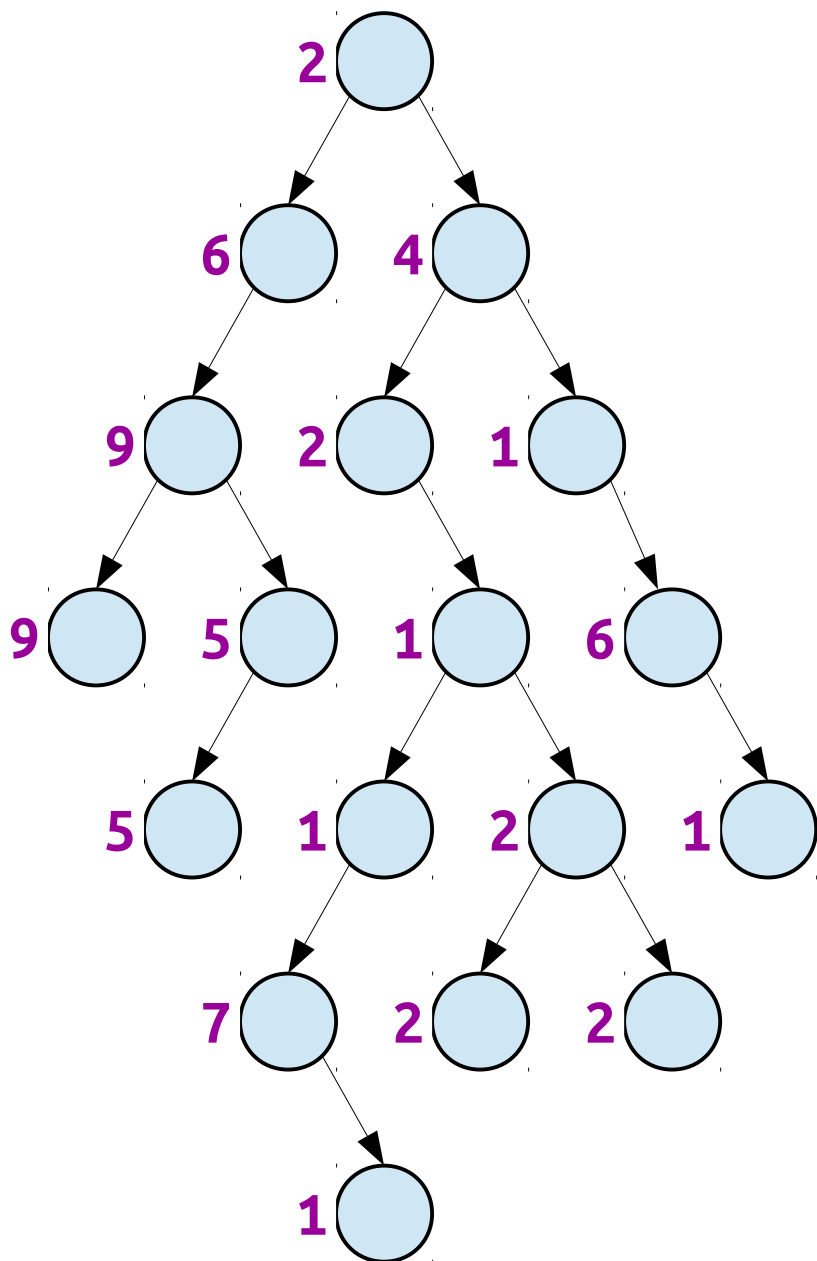
This choice of Φ is called a **sum-of-logs potential** and does show up in other places.

Question: Why is splaying fast?

Some nodes are more important than others. Assign each a weight w_i and let the total weight be W .



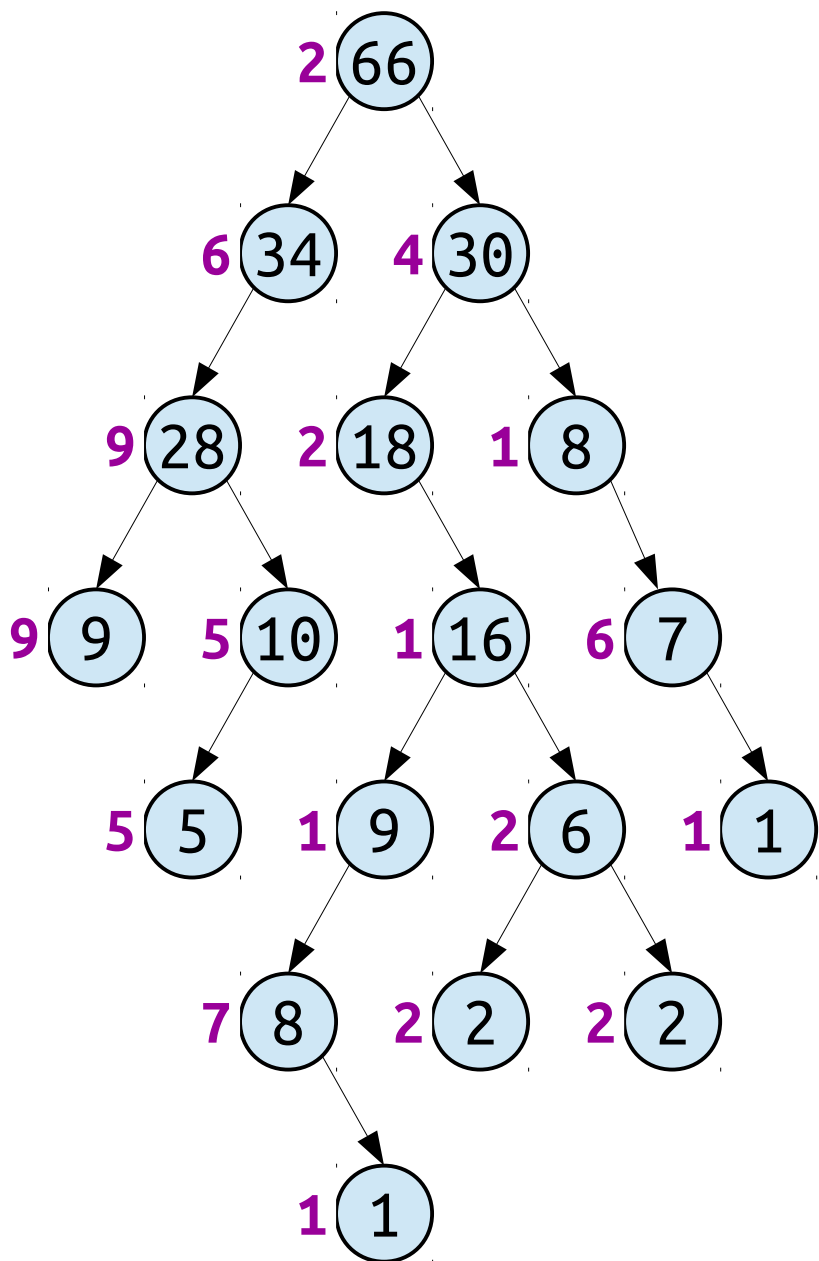
Question: Why is splaying fast?



Some nodes are more important than others. Assign each a weight w_i and let the total weight be W .

Let $s(x_i)$ be the sum of the weights in the tree rooted at x_i .

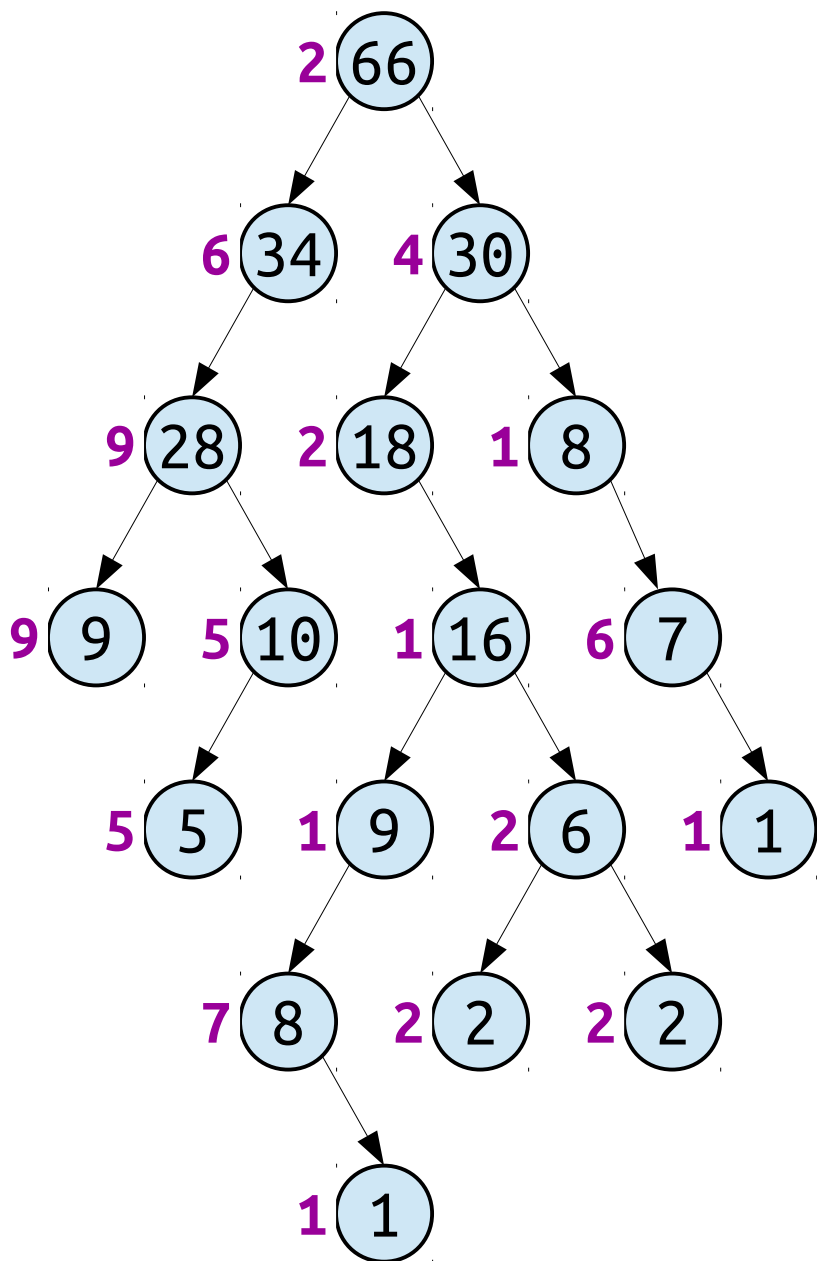
Question: Why is splaying fast?



Some nodes are more important than others. Assign each a weight w_i and let the total weight be W .

Let $s(x_i)$ be the sum of the weights in the tree rooted at x_i .

Question: Why is splaying fast?



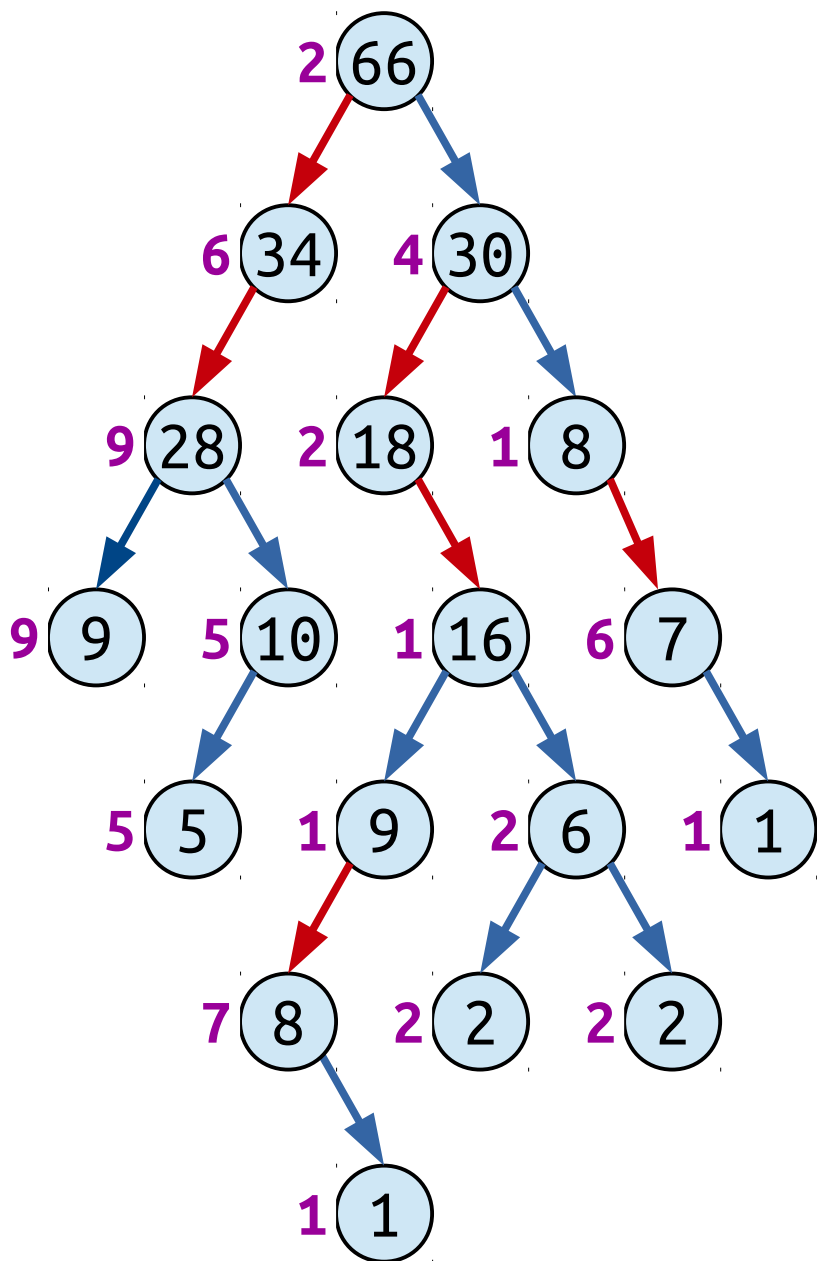
Some nodes are more important than others. Assign each a weight w_i and let the total weight be W .

Let $s(x_i)$ be the sum of the weights in the tree rooted at x_i .

Mark each edge as blue or red:

- $\lg s(\text{child}) \leq \lg s(\text{parent}) - 1$
- $\lg s(\text{child}) > \lg s(\text{parent}) - 1$

Question: Why is splaying fast?



Some nodes are more important than others. Assign each a weight w_i and let the total weight be W .

Let $s(x_i)$ be the sum of the weights in the tree rooted at x_i .

Mark each edge as blue or red:

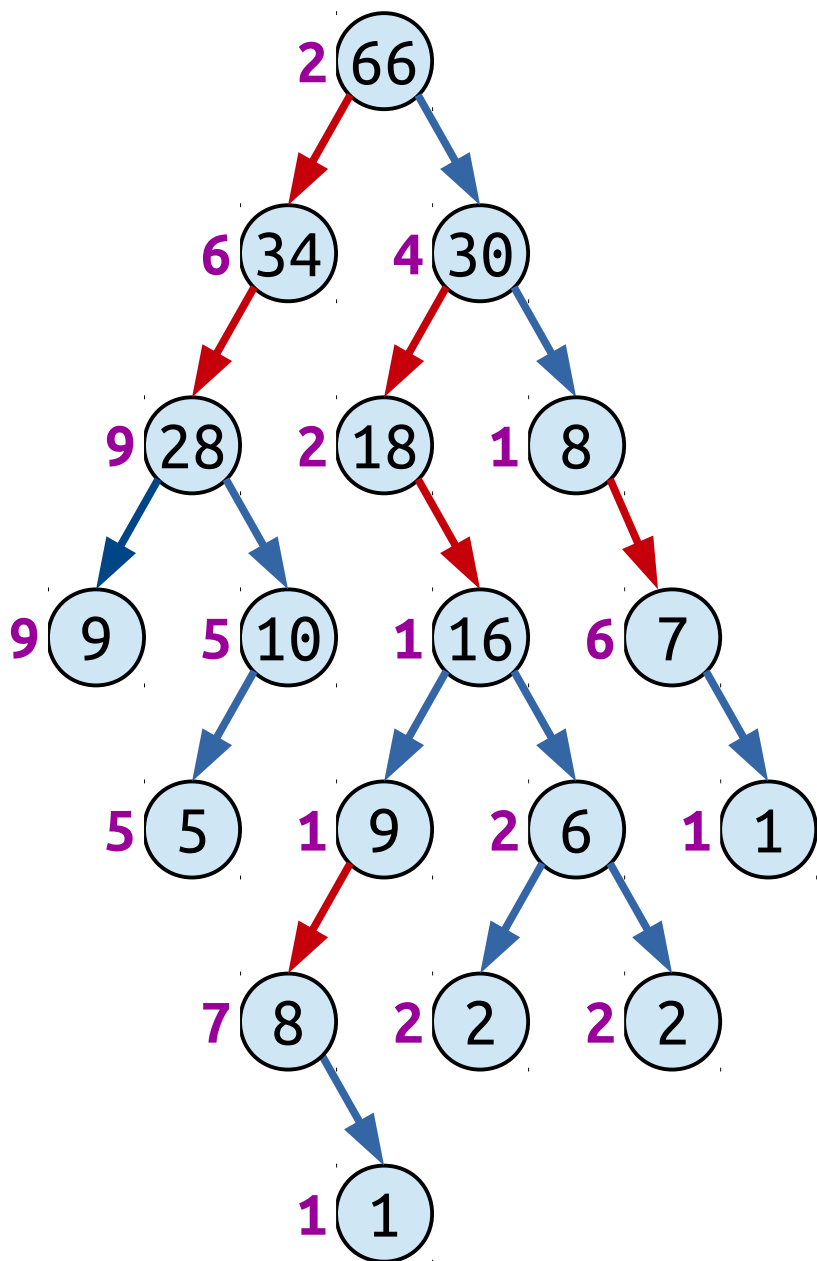
→ $\lg s(\text{child}) \leq \lg s(\text{parent}) - 1$
→ $\lg s(\text{child}) > \lg s(\text{parent}) - 1$

Cost of visiting a node:

$O(\text{\#blue-used} + \text{\#red-used})$

Question: How do we bound **\#blue-used** in this case?

Question: Why is splaying fast?



Some nodes are more important than others. Assign each a weight w_i and let the total weight be W .

Let $s(x_i)$ be the sum of the weights in the tree rooted at x_i .

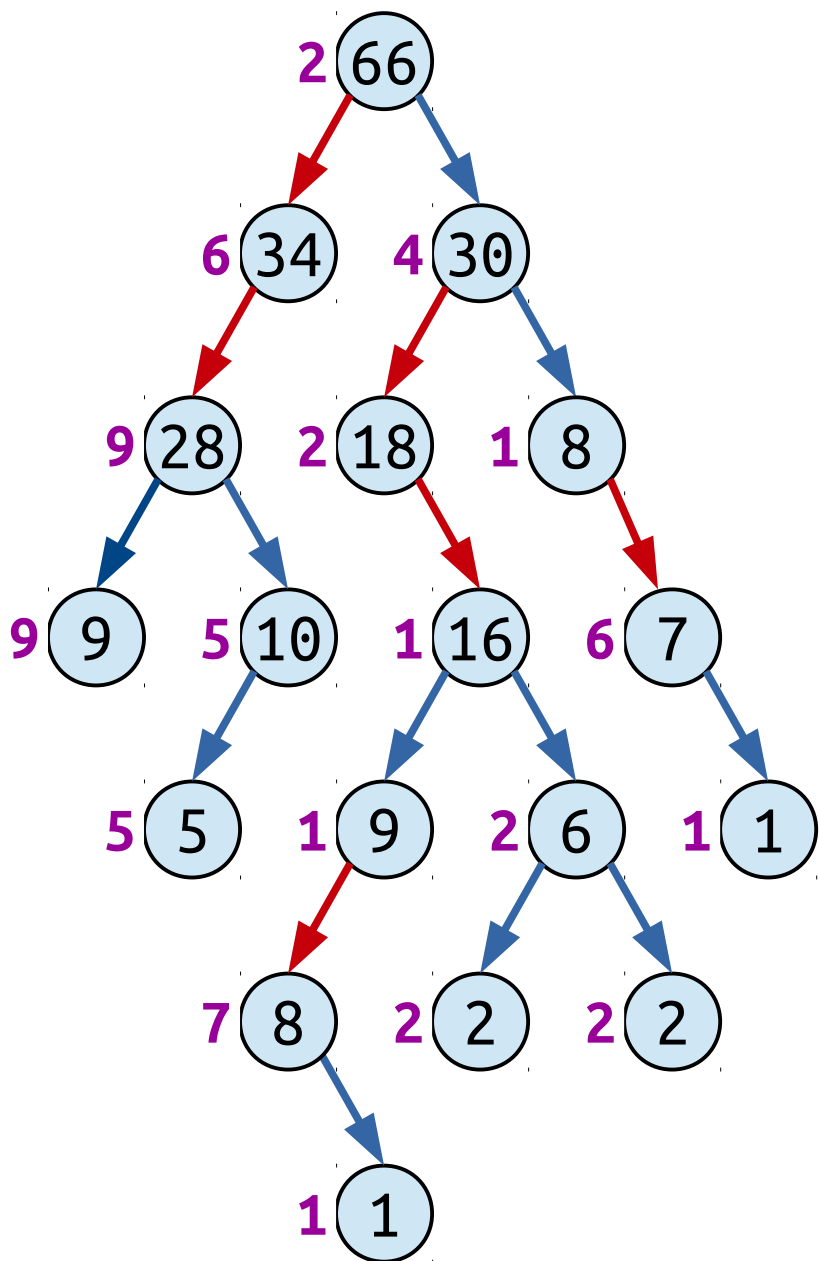
Mark each edge as blue or red:

→ $\lg s(\text{child}) \leq \lg s(\text{parent}) - 1$
→ $\lg s(\text{child}) > \lg s(\text{parent}) - 1$

Cost of visiting a node:

$O(\log(W / w_i) + \text{\#red-used})$

Question: Why is splaying fast?



Some nodes are more important than others. Assign each a weight w_i and let the total weight be W .

Let $s(x_i)$ be the sum of the weights in the tree rooted at x_i .

Mark each edge as blue or red:

→ $\lg s(\text{child}) \leq \lg s(\text{parent}) - 1$
→ $\lg s(\text{child}) > \lg s(\text{parent}) - 1$

Cost of visiting a node:

$O(\log(W / w_i) + \text{\#red-used})$

Set $\Phi = \sum_{i=1}^n \lg s(x_i)$.

Question: Why is splaying fast?

Theorem: Using the potential from before, the amortized cost of splaying a node with weight w_i is $O(\log (W / w_i))$.

Intuition: That's the cost of the blue edges.
The red edge costs get amortized away.

Corollary: Setting each node's weight to 1 shows splays satisfy the balance property.

Corollary: Setting each node's weight to its access probability shows splays satisfy the entropy property.

Corollary: By assigning each node a cleverly-chosen weight that changes over the lifetime of the splay tree, and being really, really careful with the math, the above theorem shows splay trees satisfy the working set property.

Question: Why is splaying fast?

Theorem: Using extremely detailed, technical analyses, it's possible to prove the following properties of splay trees:

Splay trees have the dynamic finger property.

Treating a splay tree as a deque gives $O(\alpha^*(n))$ amortized cost per operation, where $\alpha^*(n)$ is the number of times that the Ackermann inverse function α must be applied to n to drop it to a constant.

These proofs are nontrivial. We currently lack a deep way of thinking about how splay trees work.

Question: Why is splaying fast?

Consider *any* series of queries that can be performed on any BST. Out of all the possible ways you could implement a single binary search tree augmented with a finger, let OPT be the minimum number of BST operations to correctly answer those queries.

Conjecture (Dynamic Optimality): The cost of performing those on a splay tree is $O(OPT)$.

*This is an open problem!
It's not $P \stackrel{?}{=} NP$, but it's still a big one!*

This is an active area of research, despite the fact that splay trees were invented about 35 years ago!

Is there anything splay trees are known to be bad at?

- Worst-case efficiency (the *balance property*) isn't the only metric we can use to measure BST performance.
- Specialized data structures like weight-balanced trees, level-linked finger search trees, and Iacono's structure can be designed to meet these bounds.
- For a BST to have all these properties at once, it needs to be able to move nodes around.
- Rotate-to-root is a plausible but inefficient mechanism for reordering nodes.
- Splaying corrects for rotate-to-root by handling linear chains more intelligently.
- Splaying provides simple implementations of all common BST operations.
- By using a heavy/light decomposition, we can isolate the effects of poor tree shapes.
- Using a sum-of-logs potential allows us to amortize away heavy edges.
- Splay trees have the balance property, entropy property, dynamic finger property, and working set property.
- It's an open problem in data structure theory whether it's possible to improve upon splay trees in an amortized sense.

To Summarize...

Next Time

- ***Classes of Hash Functions***
 - Hashing in theory versus hashing in practice.
- ***Frequency Estimation***
 - Counting without counting.
- ***Concentration Inequalities***
 - How good are our approximations?
- ***Probability Amplification***
 - Can be be probably approximately correct?