Splay Trees

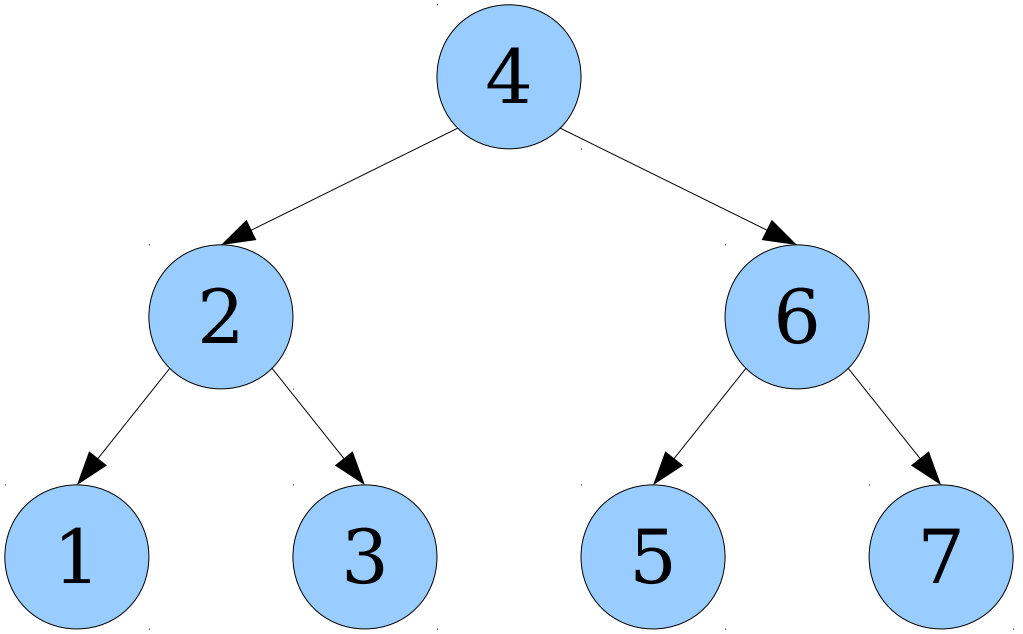
Outline for Today

- ***Static Optimality***
  - Balanced BSTs aren't necessarily optimal!
- ***Splay Trees***
  - A self-adjusting binary search tree.
- ***Properties of Splay Trees***
  - Why is splaying worthwhile?
- ***Dynamic Optimality (ITA)***
  - An open problem in data structures.

Static Optimality

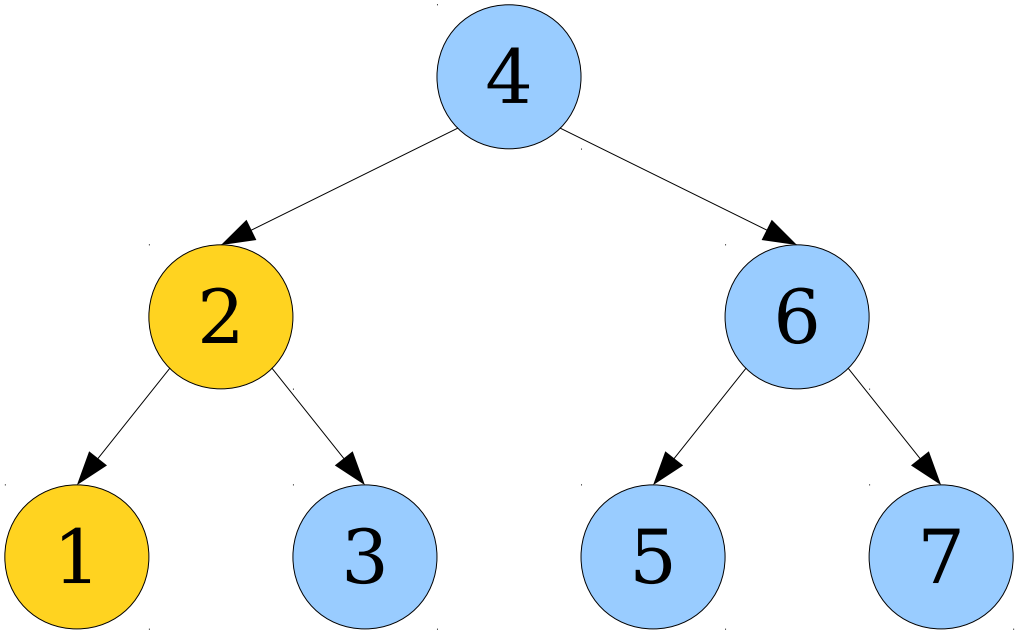
Balanced BSTs

- We've explored balanced BSTs this quarter because they guarantee worst-case O(log *n*) operations.
- ***Claim:*** If the elements in the tree aren't accessed uniformly, then a balanced BST might not actually be the ideal BST.



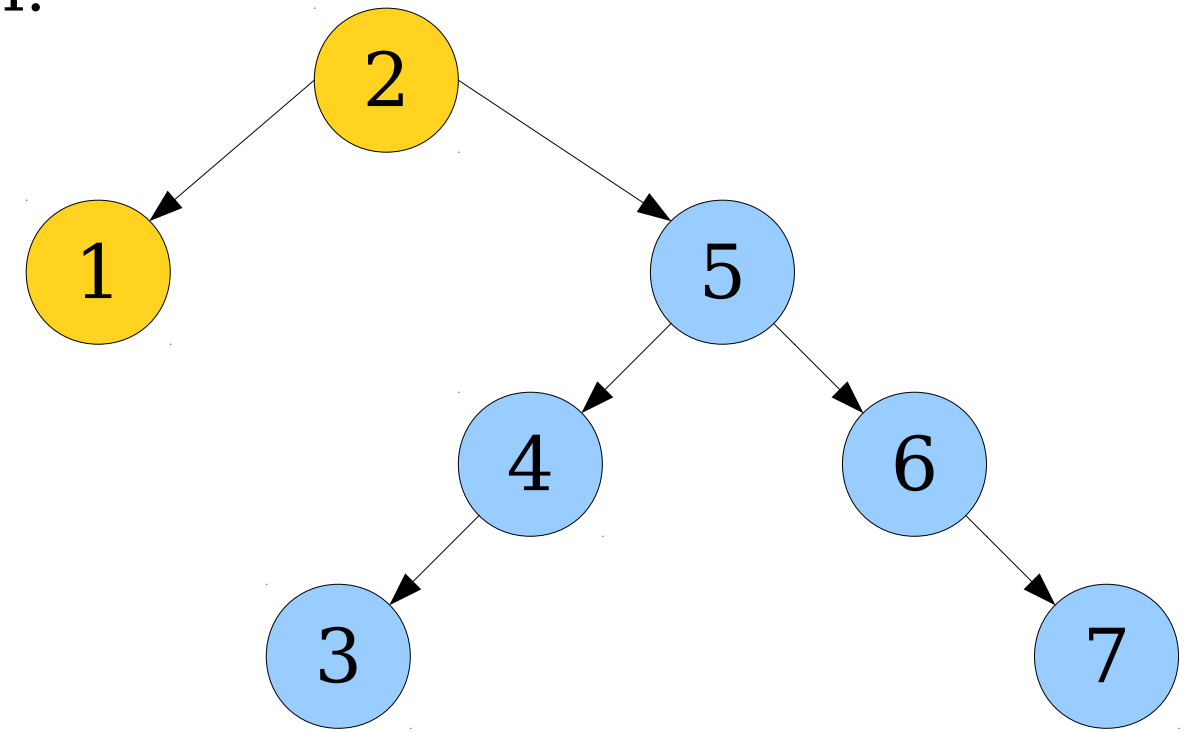
Balanced BSTs

- We've explored balanced BSTs this quarter because they guarantee worst-case O(log *n*) operations.
- ***Claim:*** If the elements in the tree aren't accessed uniformly, then a balanced BST might not actually be the ideal BST.



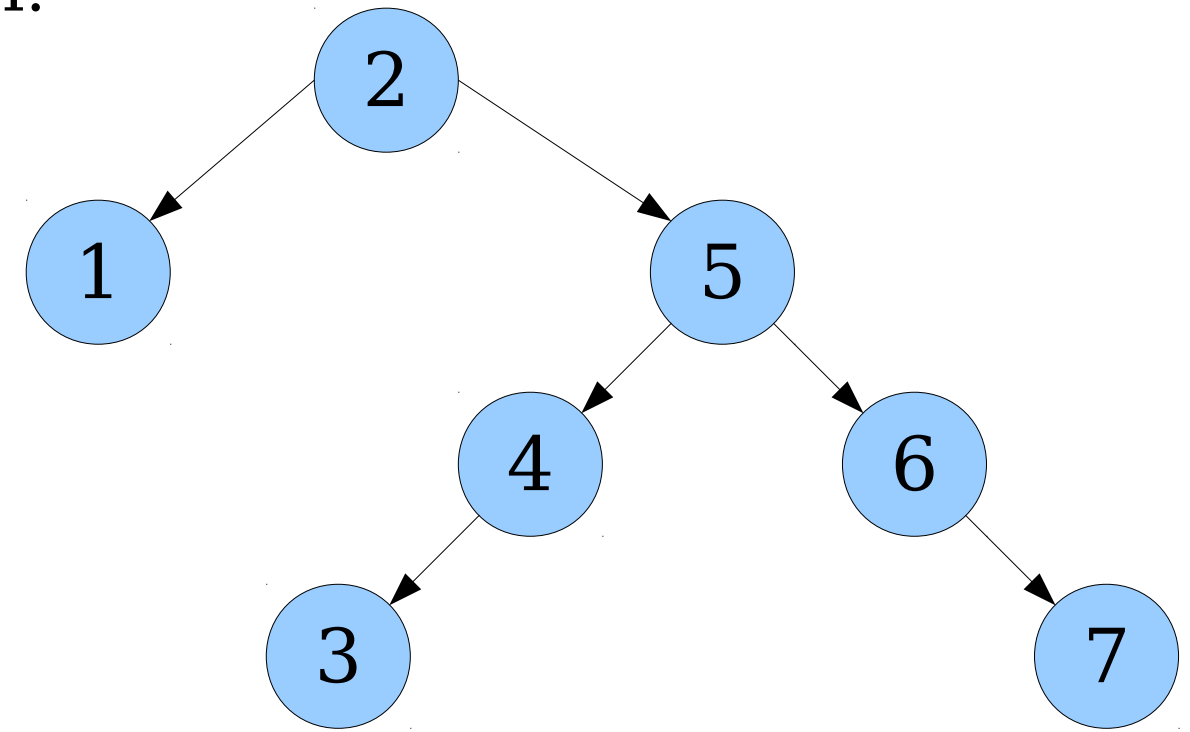
Balanced BSTs

- We've explored balanced BSTs this quarter because they guarantee worst-case O(log n) operations.
- ***Claim:*** If the elements in the tree aren't accessed uniformly, then a balanced BST might not actually be the ideal BST.



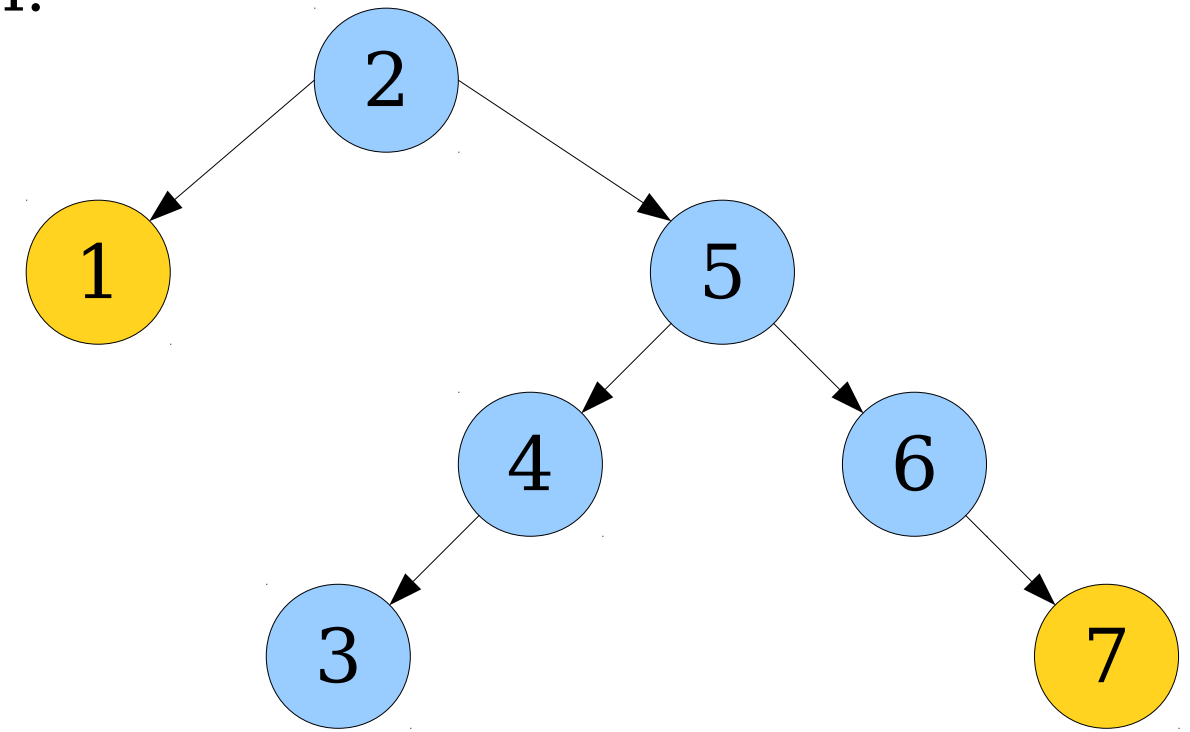
Balanced BSTs

- We've explored balanced BSTs this quarter because they guarantee worst-case O(log n) operations.
- ***Claim:*** If the elements in the tree aren't accessed uniformly, then a balanced BST might not actually be the ideal BST.



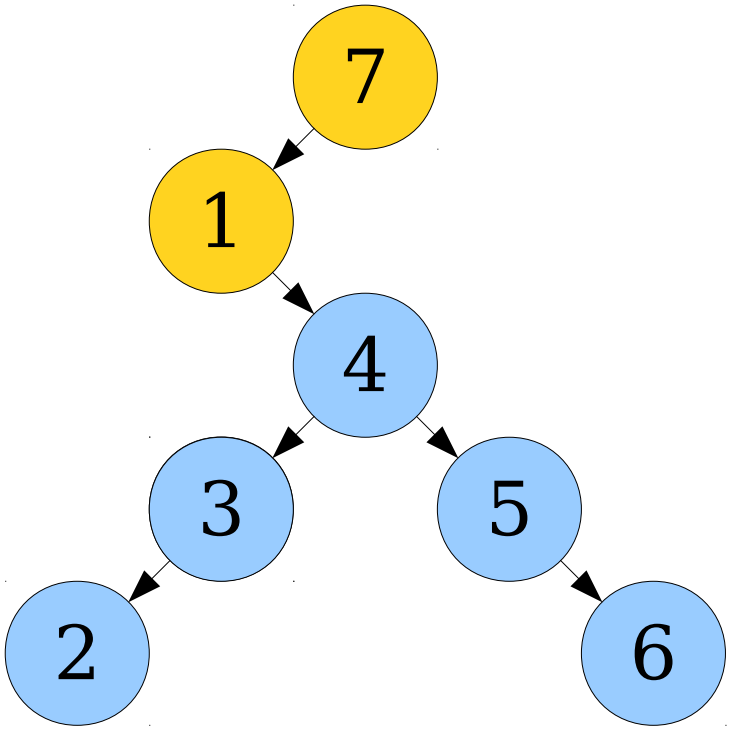
Balanced BSTs

- We've explored balanced BSTs this quarter because they guarantee worst-case O(log n) operations.
- ***Claim:*** If the elements in the tree aren't accessed uniformly, then a balanced BST might not actually be the ideal BST.



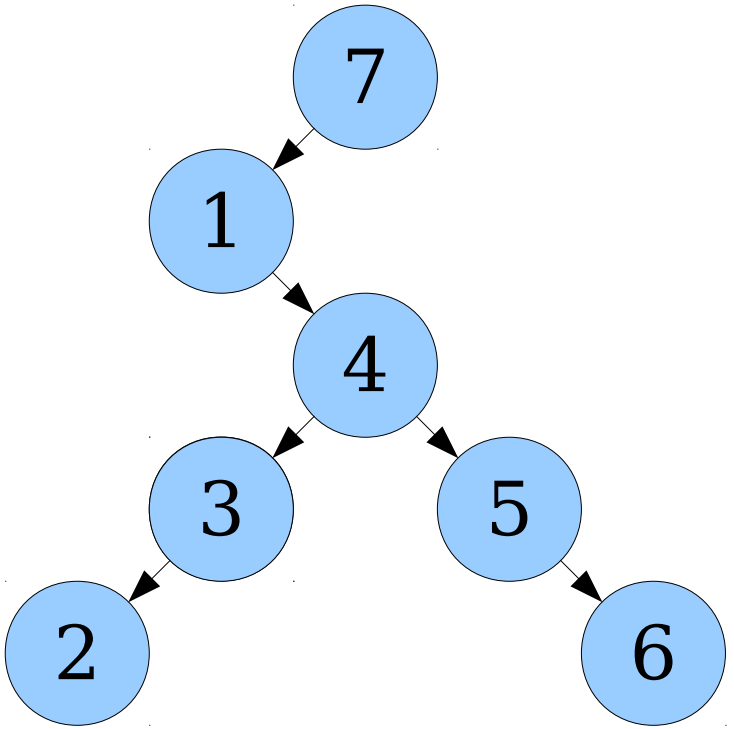
Balanced BSTs

- We've explored balanced BSTs this quarter because they guarantee worst-case O(log n) operations.
- ***Claim:*** If the elements in the tree aren't accessed uniformly, then a balanced BST might not actually be the ideal BST.



Balanced BSTs

- We've explored balanced BSTs this quarter because they guarantee worst-case O(log n) operations.
- ***Claim:*** If the elements in the tree aren't accessed uniformly, then a balanced BST might not actually be the ideal BST.



Static Optimality

- Let $S = \{ x_1, x_2, \ldots, x_n \}$ be a set with access probabilities $p_1, p_2, \ldots, p_n$.
- If T is a BST whose keys are the keys in S, then let X_T be a random variable equal to the number of nodes in T that are touched when performing a lookup, assuming the key to look up is sampled from the above probability distribution.
- ***Goal:*** Construct a binary search tree T^* such that $E[X_{T^*}]$ is minimal.
- T^* is called a ***statically optimal binary search tree***.

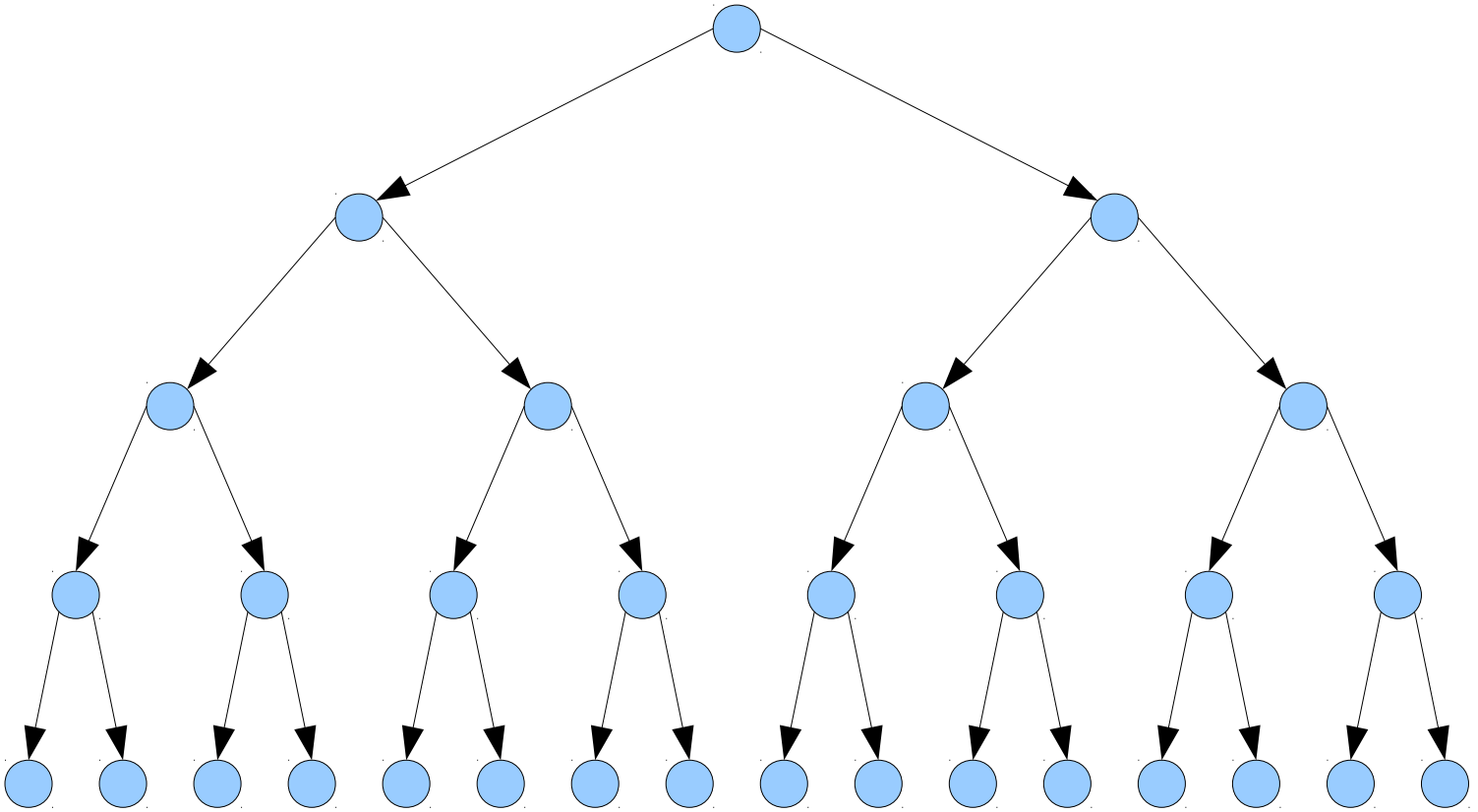
Static Optimality

- ***Theorem:*** There is an $O(n^2)$-time dynamic programming algorithm for constructing statically optimal binary search trees.
  - Knuth, 1971. (See CLRS)
- ***Theorem:*** Weight-balanced trees whose weights are the element access probabilities have an expected lookup cost with a factor of 1.5 of a statically-optimal tree.
  - Mehlhorn, 1975.
- You can build a weight-balanced tree for a set of keys in time $O(n \log n)$ using a clever divide-and-conquer algorithm. You'll see this in PS4.

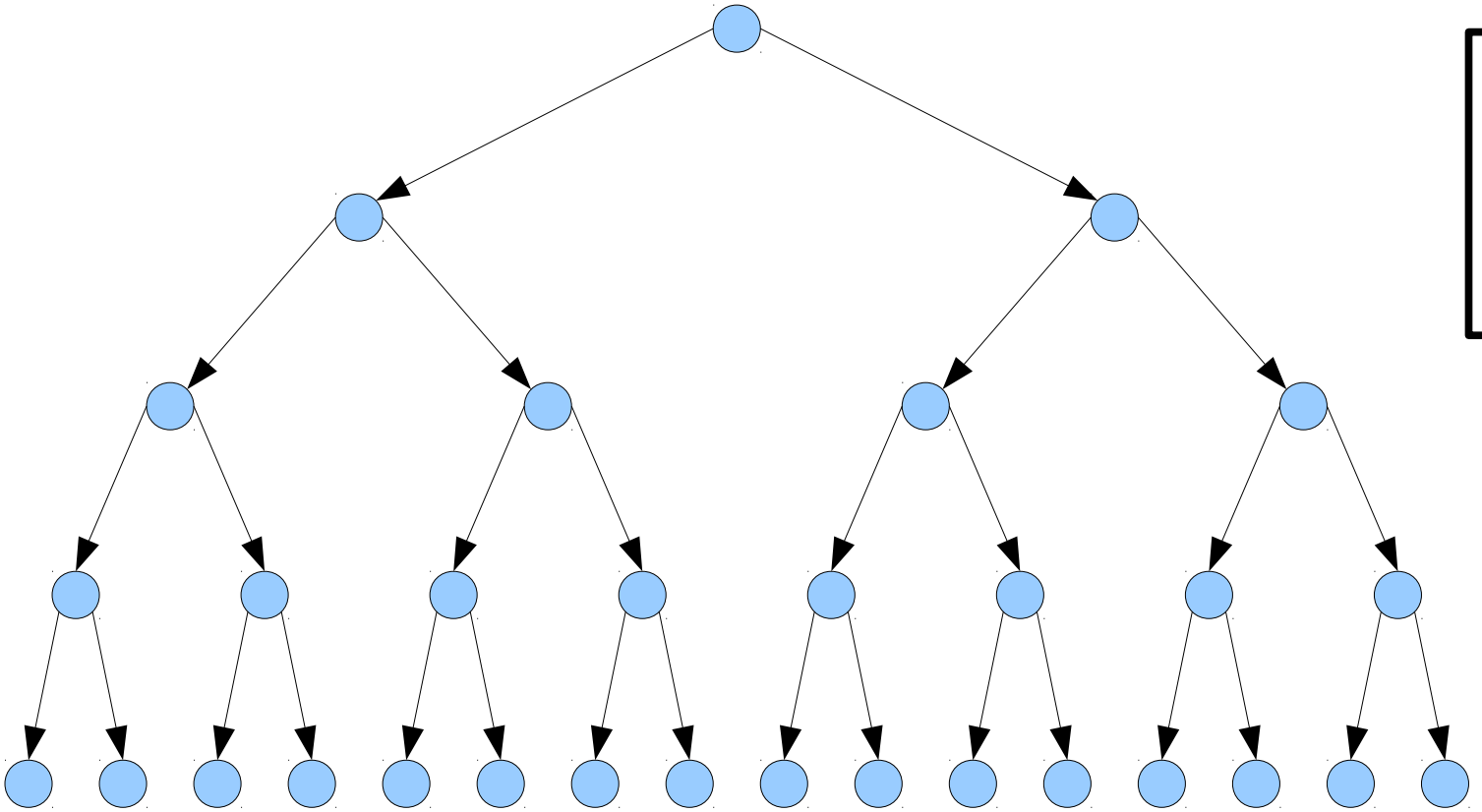
Lower Bounds

- We know that, for worst-case efficiency, any BST for a set of keys will have a worst-case lookup time of $\Omega(\log n)$.
- This means that if we find a BST whose worst-case lookup time is $O(\log n)$, it must be optimal.
- Can we derive a similar sort of lower bound for the *expected* cost of a lookup in a BST if the access probabilities are known?

Static Optimality

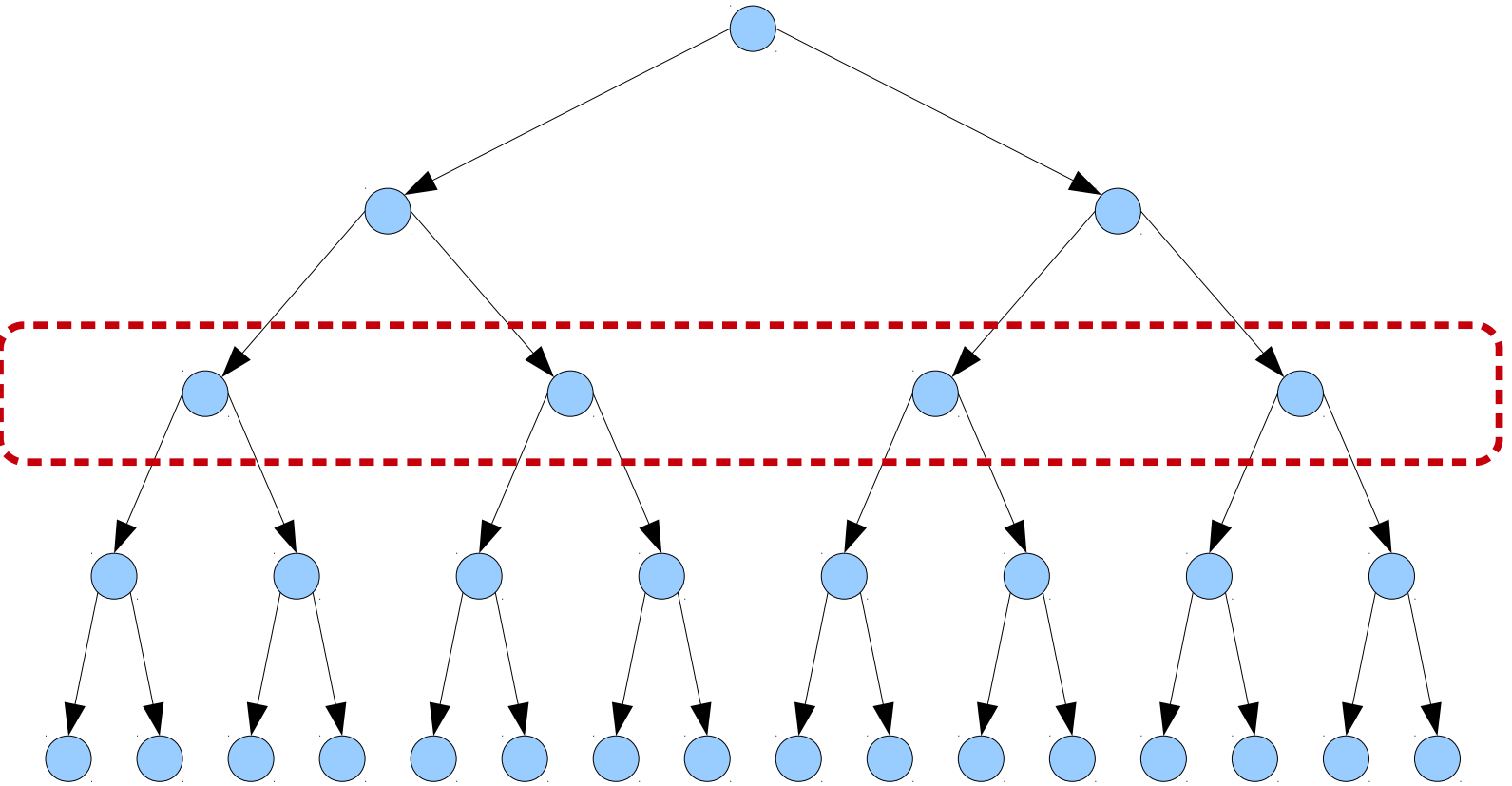


Static Optimality

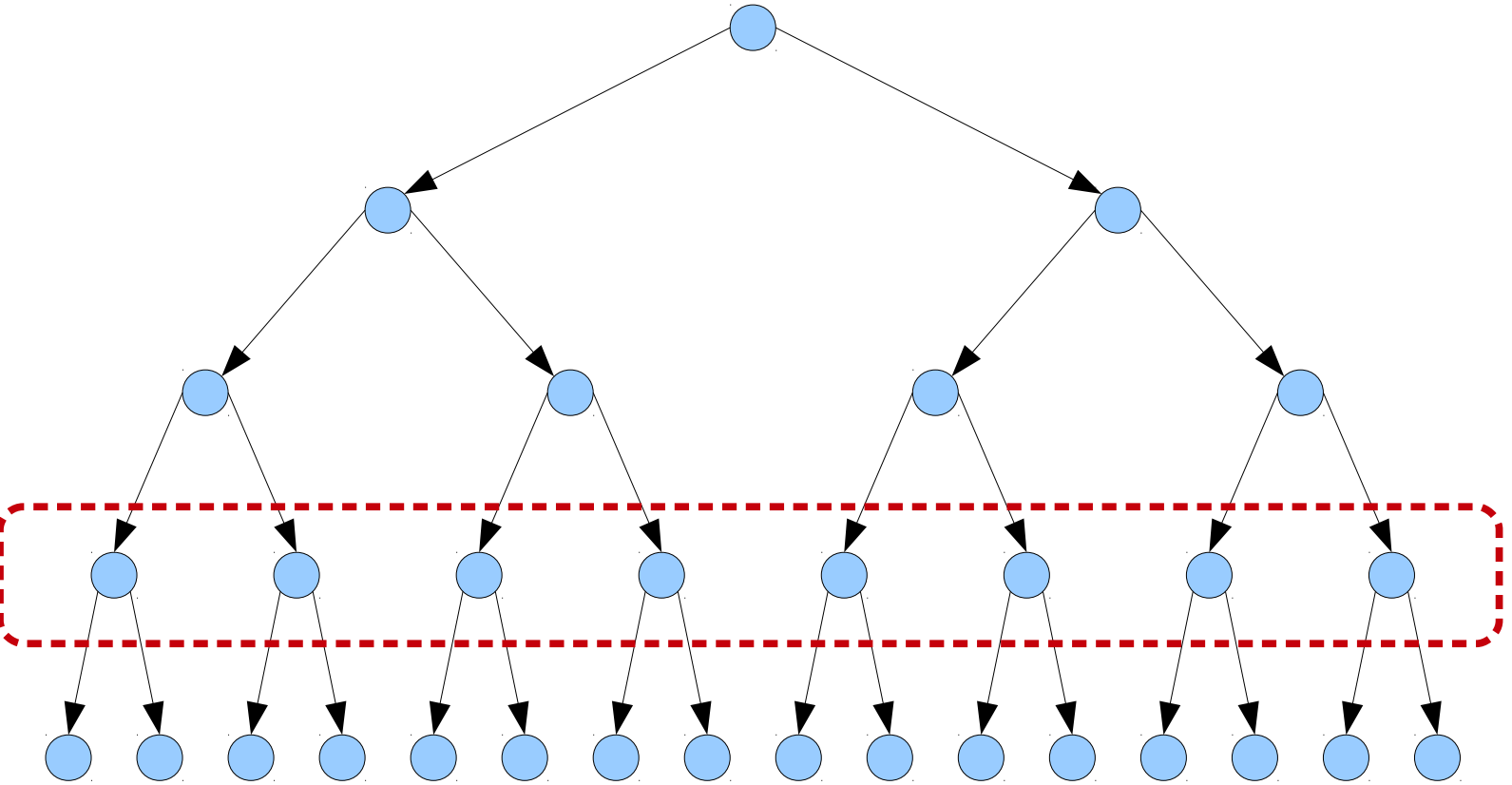


Intuition: Try to place nodes with higher access probabilities higher up in the tree.

Static Optimality

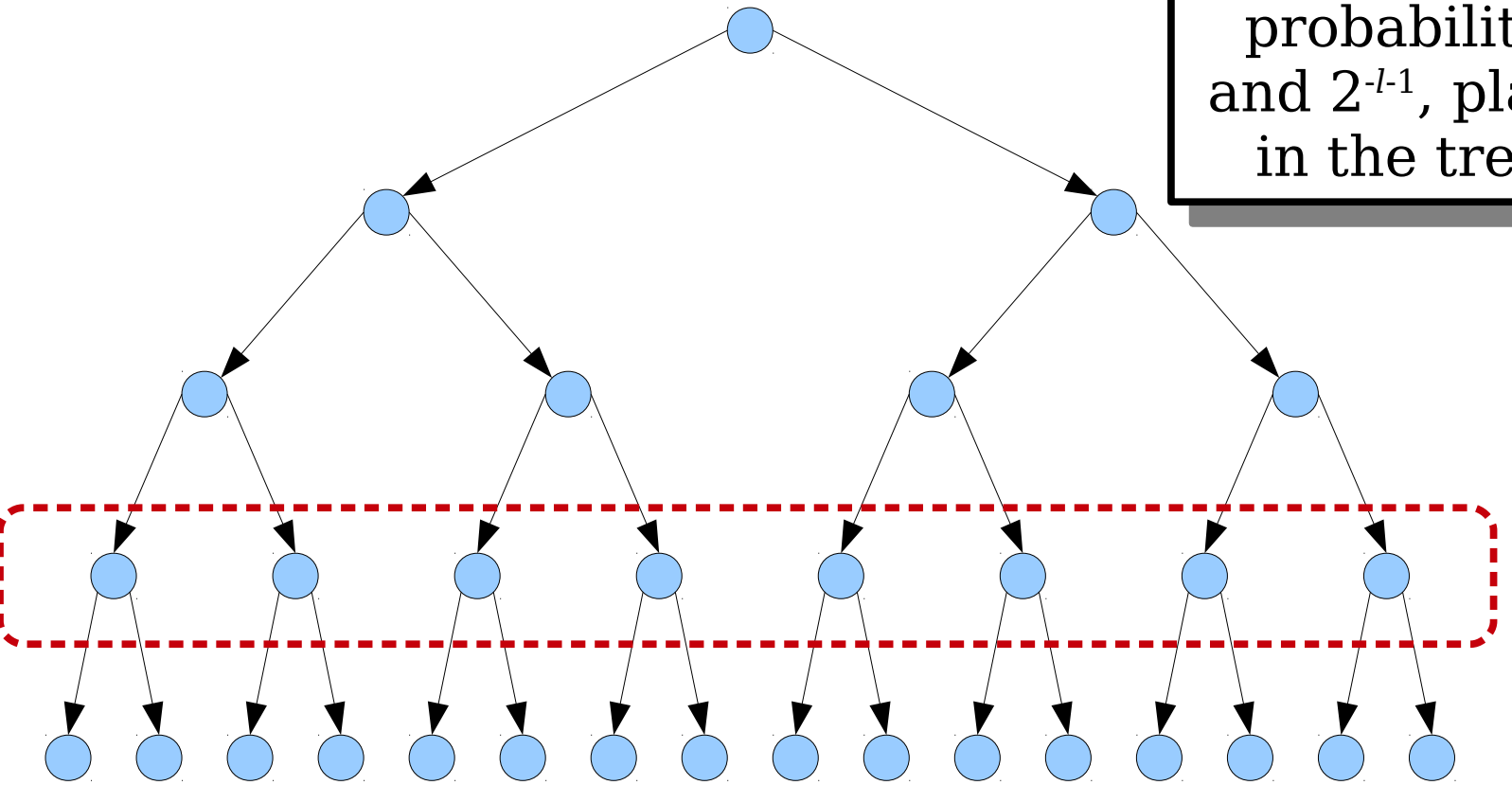


Static Optimality

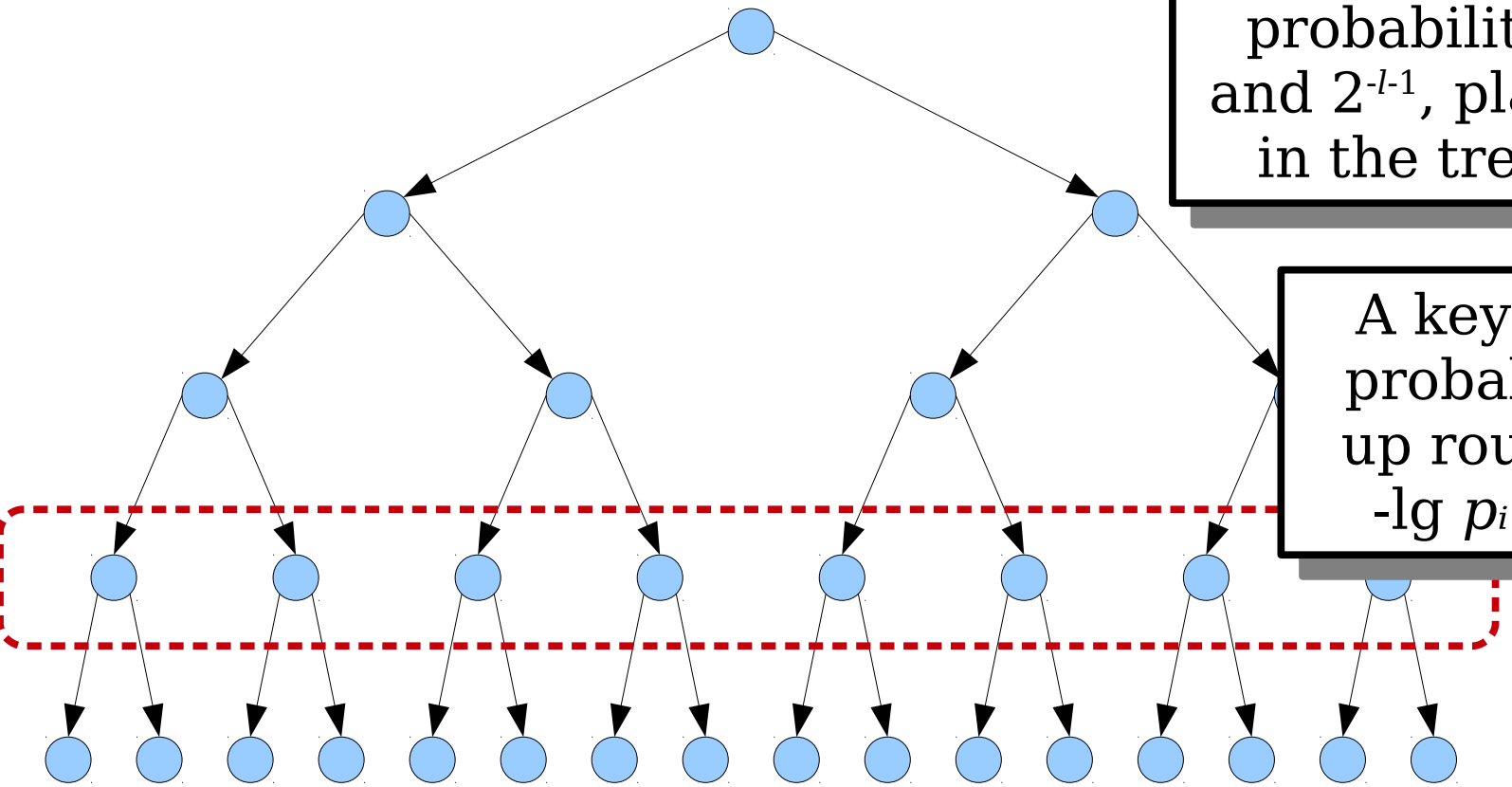


Static Optimality

Idea: If a key has access probability between 2^{-l} and 2^{-l-1}, place it at level l in the tree if possible.



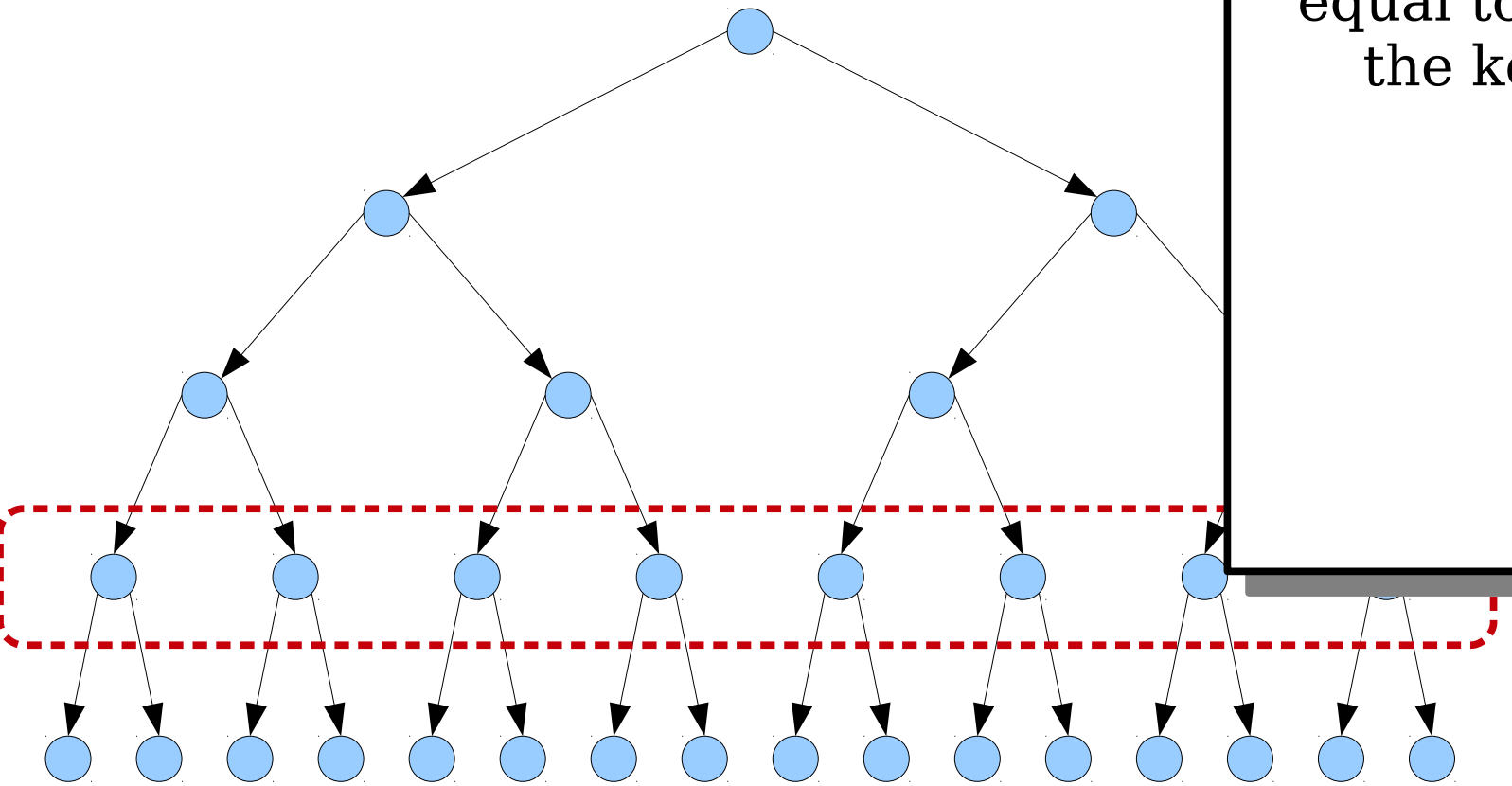
Static Optimality



Idea: If a key has access probability between 2^{-l} and 2^{-l-1}, place it at level l in the tree if possible.

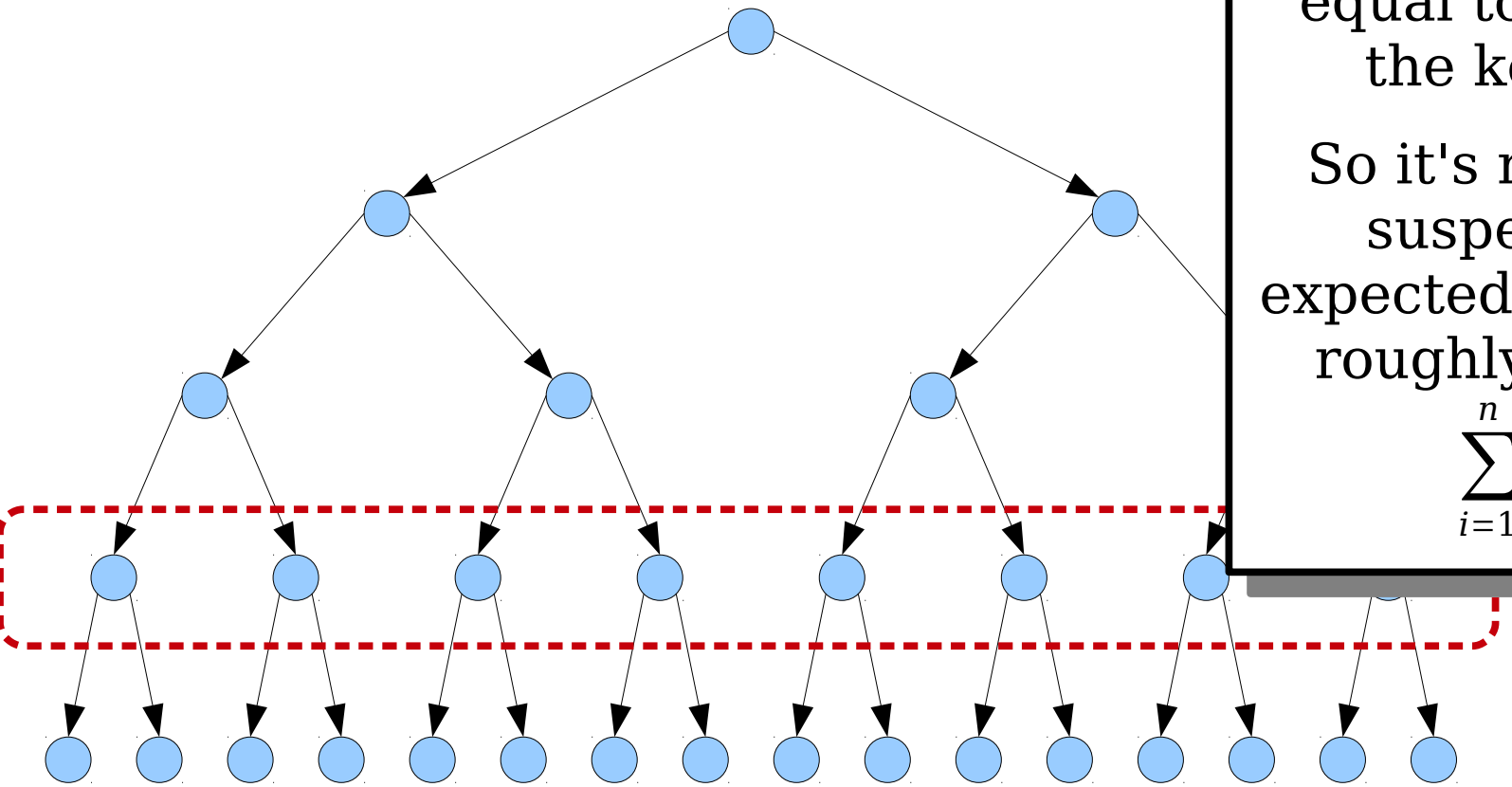
A key with access probability p_i ends up roughly at level $-\lg p_i$ in the tree.

Static Optimality



The cost of looking up some key x_i is roughly equal to the depth of the key x_i: -lg p_i.

Static Optimality



The cost of looking up some key x_i is roughly equal to the depth of the key x_i: -lg p_i.

So it's reasonable to suspect that the expected lookup cost to roughly work out to

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

Shannon Entropy

- Consider a discrete probability distribution with elements $x_1, \ldots, x_n$, 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

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

- Notice that $H = \lg n$ if all elements have equal access probability ($p_i = {}^1/_n$).
- Notice that $H = 0$ if a single element has the entire probability mass.

Static Optimality

- Consider a discrete probability distribution with elements $x_1, \ldots, x_n$, 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

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

- ***Theorem:*** The expected lookup cost in *any* binary search tree for keys $x_1, \ldots, x_n$ with access probabilities $p_1, \ldots, p_n$ is $\Omega(1 + H)$.
- ***Theorem:*** For any set of keys x_i with access probabilities p_i, there is a BST that whose expected lookup time is $\Theta(1 + H)$.

Weaknesses of Static Optimality

- Statically optimal BSTs are fantastic if the lookups are sampled randomly from a fixed distribution.
- However, what if you don't know anything about the particular access pattern you're going to have?
- ***Question 1:*** Is it possible to build a BST with $O(1 + H)$ expected lookup time if the probability distribution isn't known in advance?

Weaknesses of Static Optimality

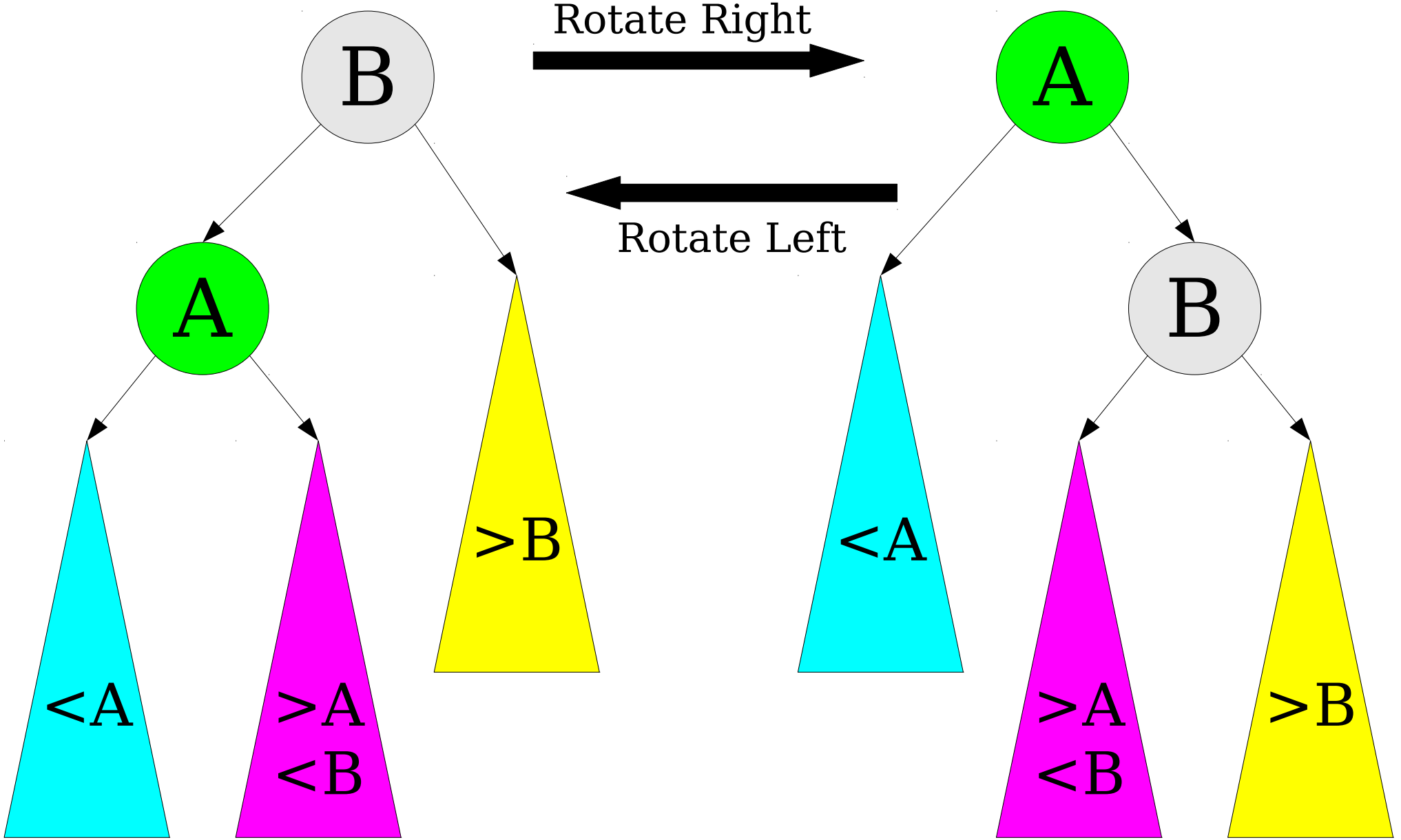
- Just knowing the access probabilities doesn't guarantee that you can build a good BST.
- ***Example:*** Suppose you want to build a BST where all elements are repeatedly looked up sequentially.
- All elements are visited the same number of times, so a statically-optimal BST would be a perfectly balanced BST. This means that the lookups take time $\Theta(n \log n)$.
- There's a simple $O(n)$-time algorithm for visiting all the nodes of a BST sequentially. If we knew in advance that we'd visit everything in this order, it makes more sense to use this algorithm than to just do a bunch of lookups.
- ***Question 2:*** Can we build a BST that adapts to access patterns beyond just the resulting access probabilities?

Challenge: Can we build a type of BST that meets the static optimality requirements, but is also sensitive to access patterns?

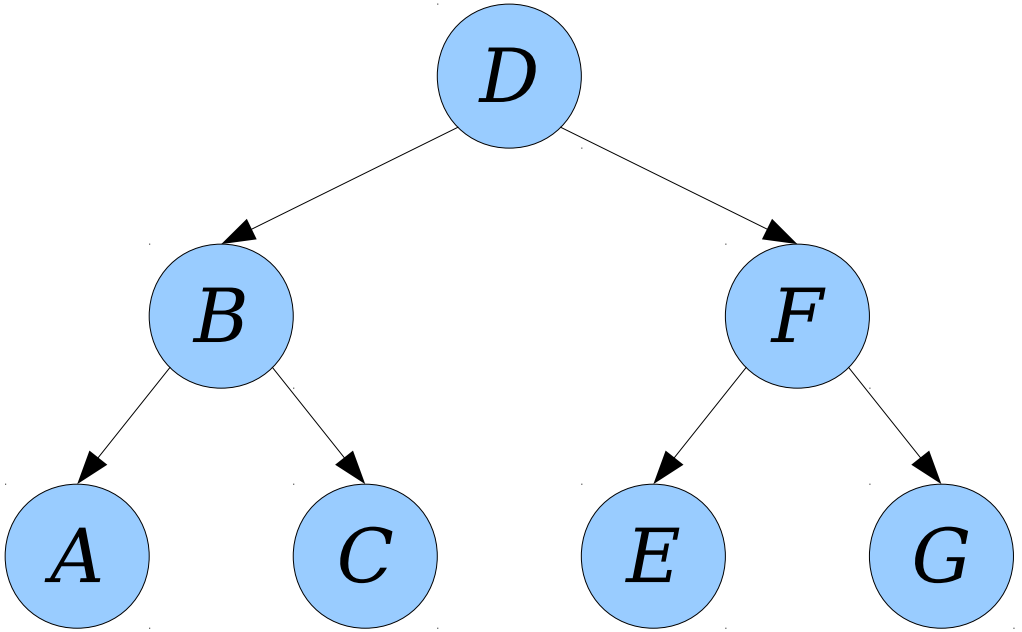
The Intuition

- If we don't know the access probabilities in advance, we can't build a fixed BST and then “hope” it works correctly.
- Instead, we'll have to restructure the BST as operations are performed.
- For now, let's focus on lookups; we'll handle insertions and deletions later on.

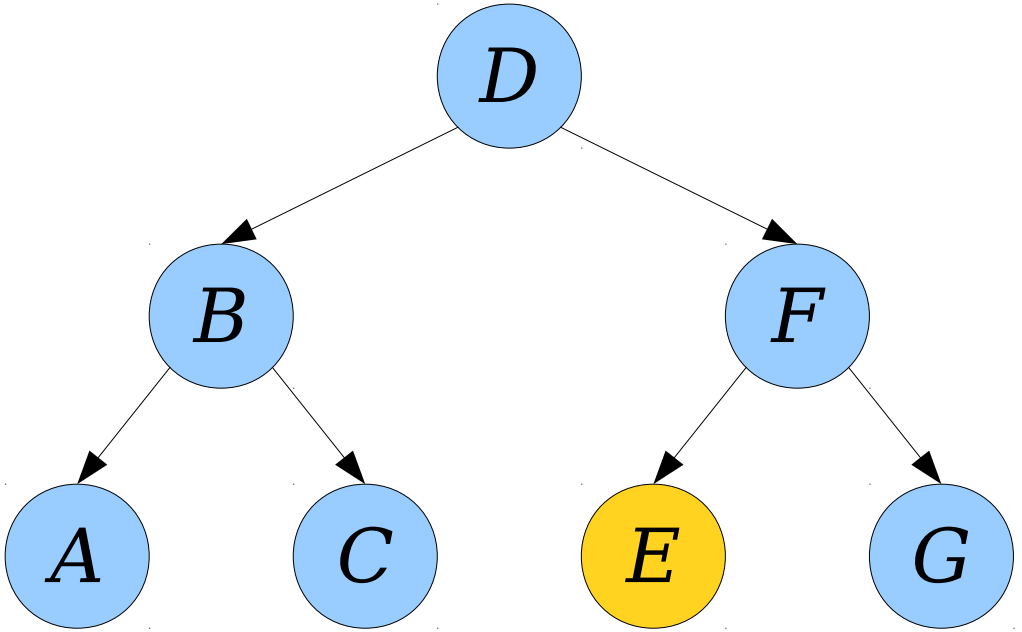
Refresher: Tree Rotations



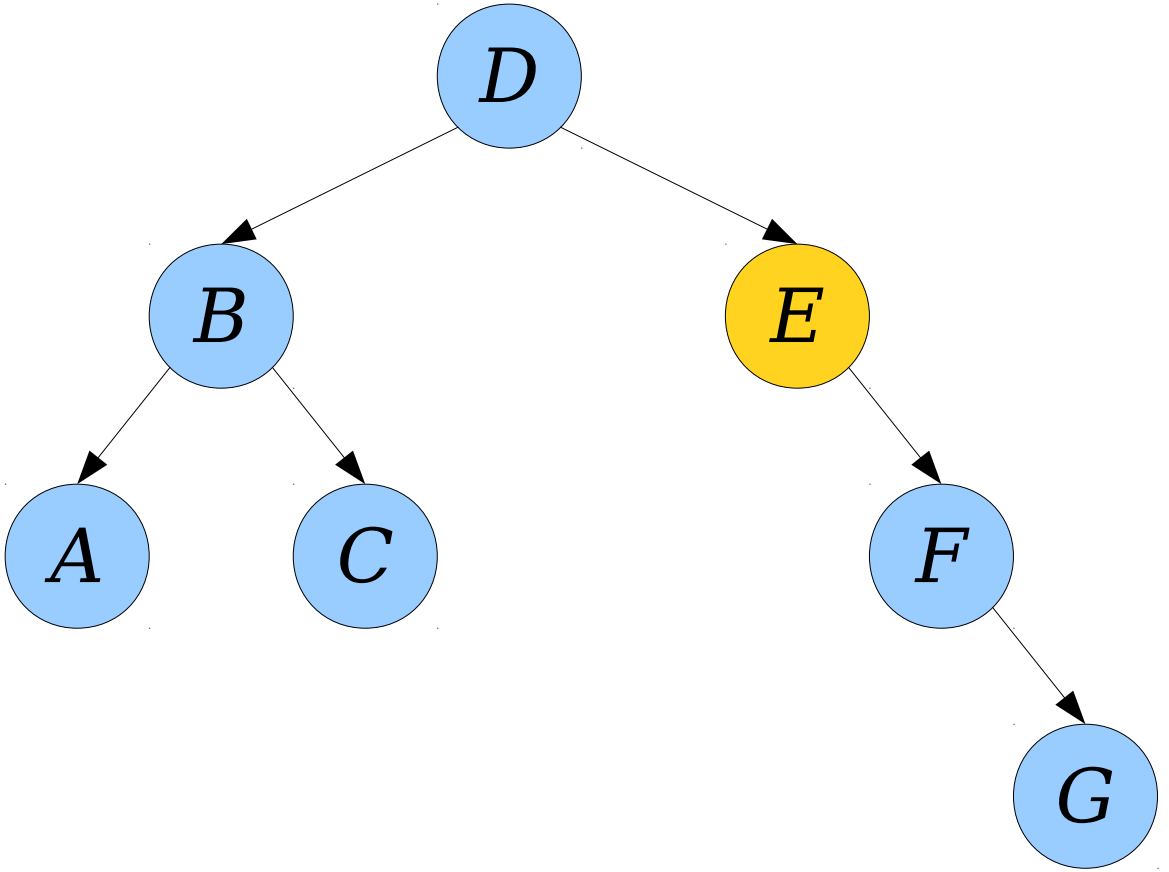
An Initial Approach



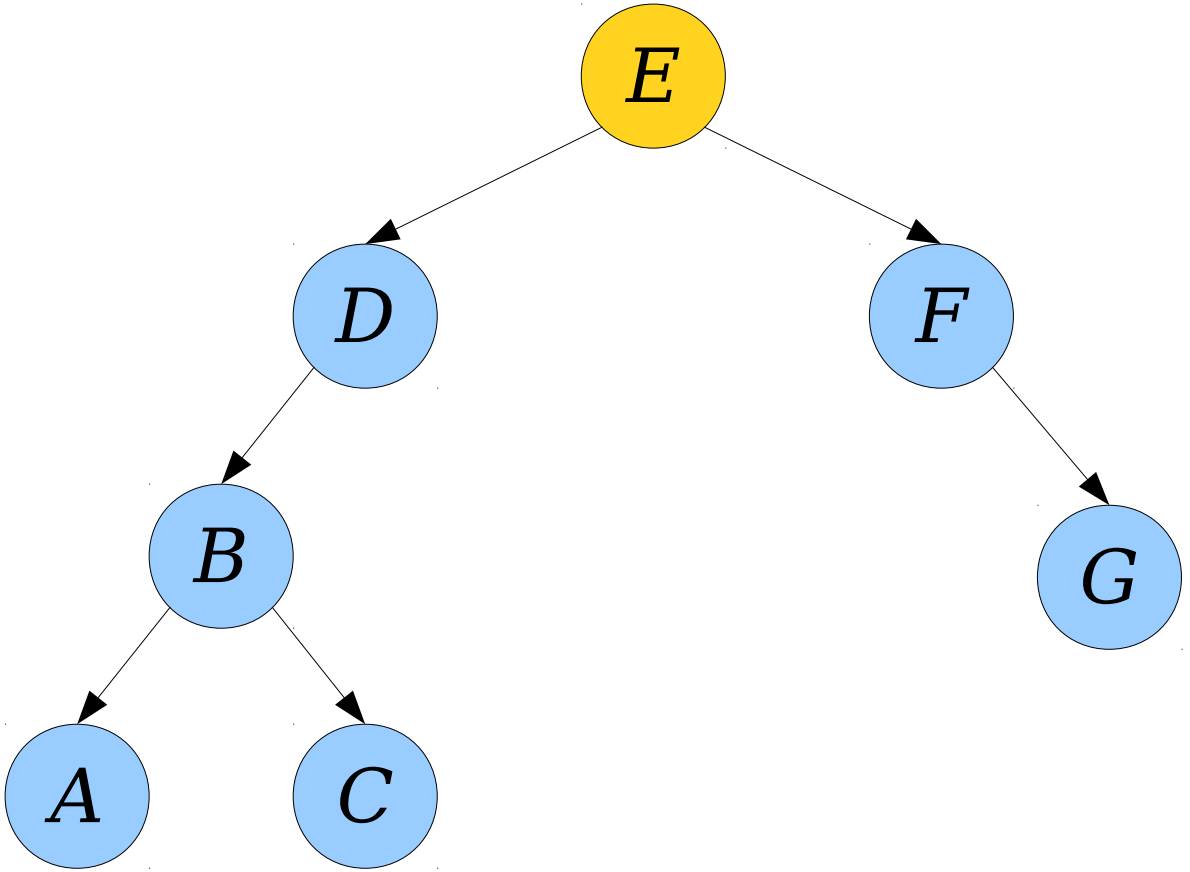
An Initial Approach



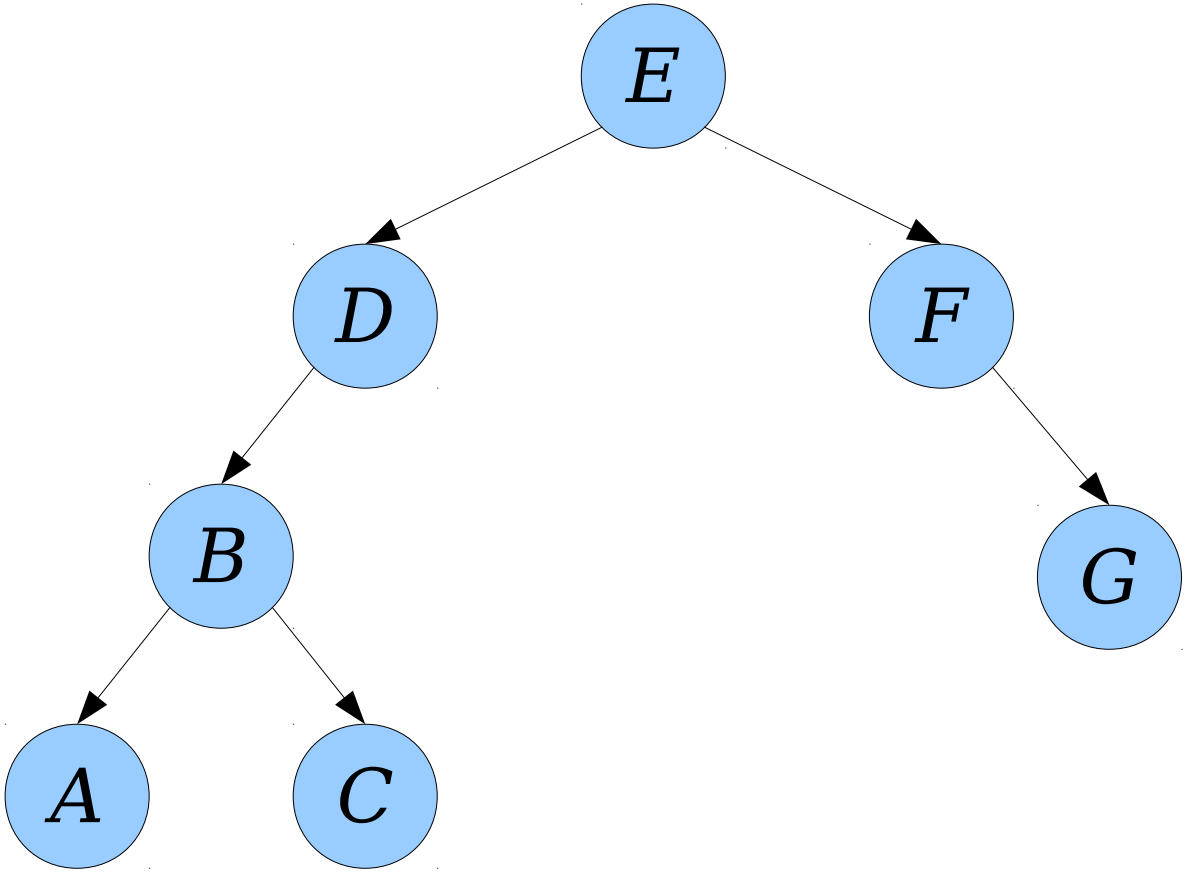
An Initial Approach



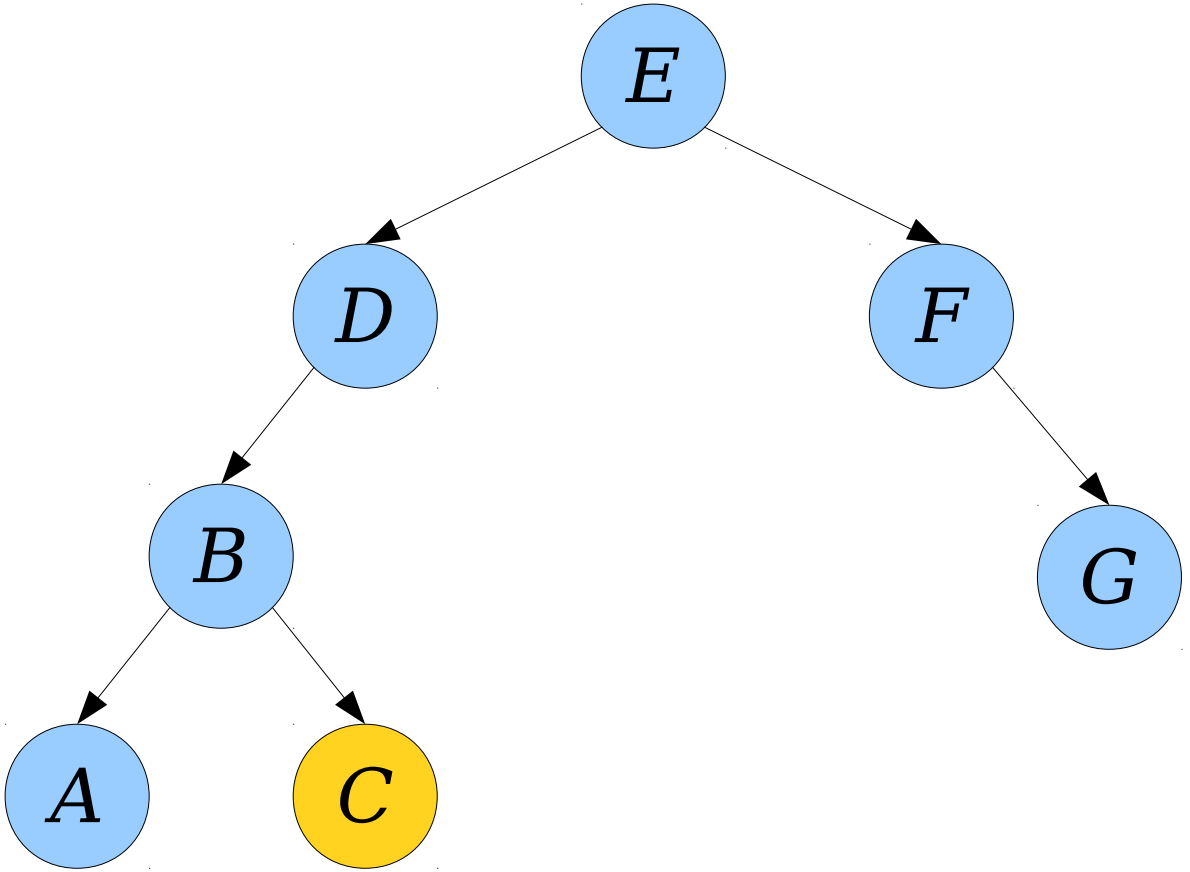
An Initial Approach



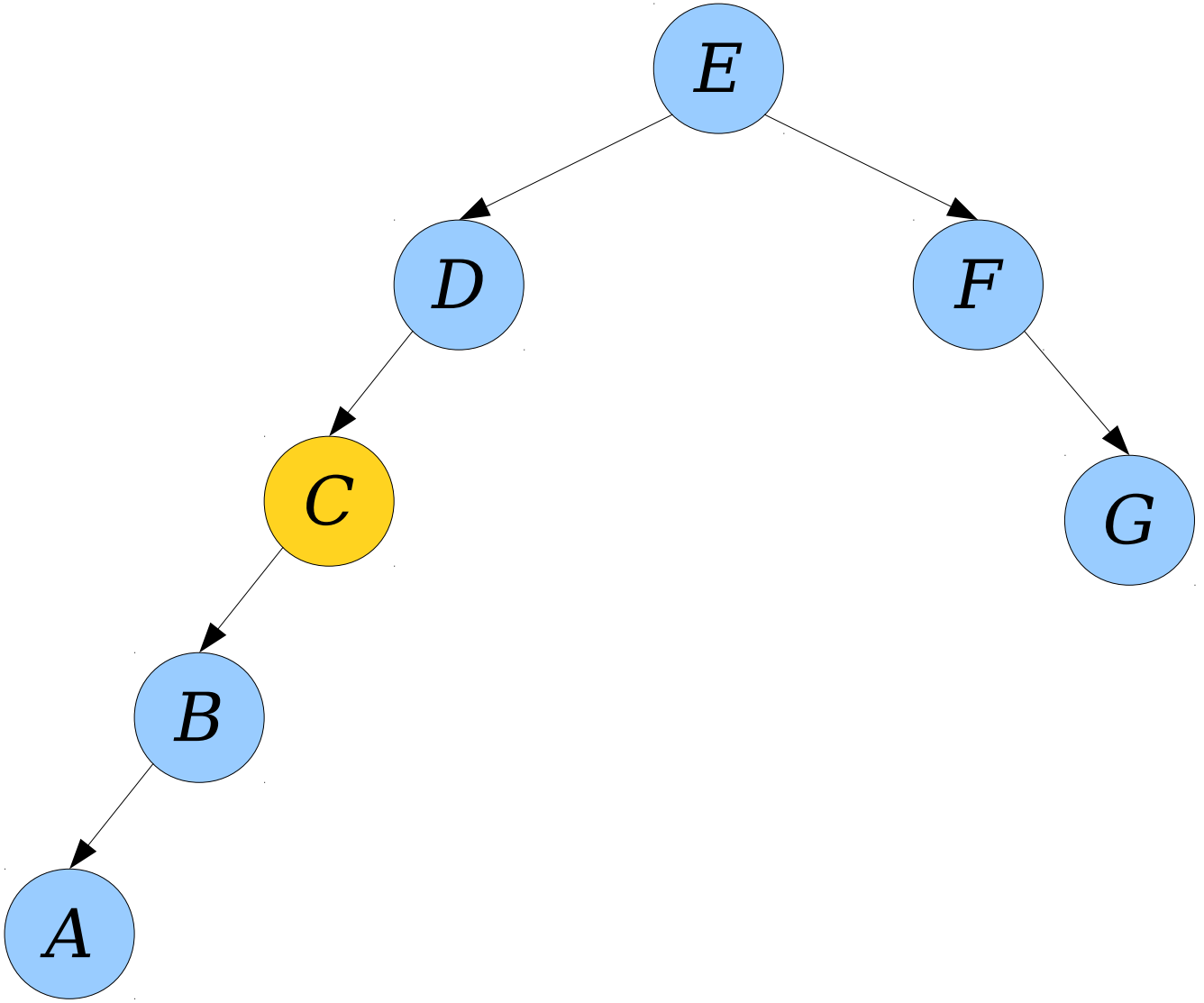
An Initial Approach



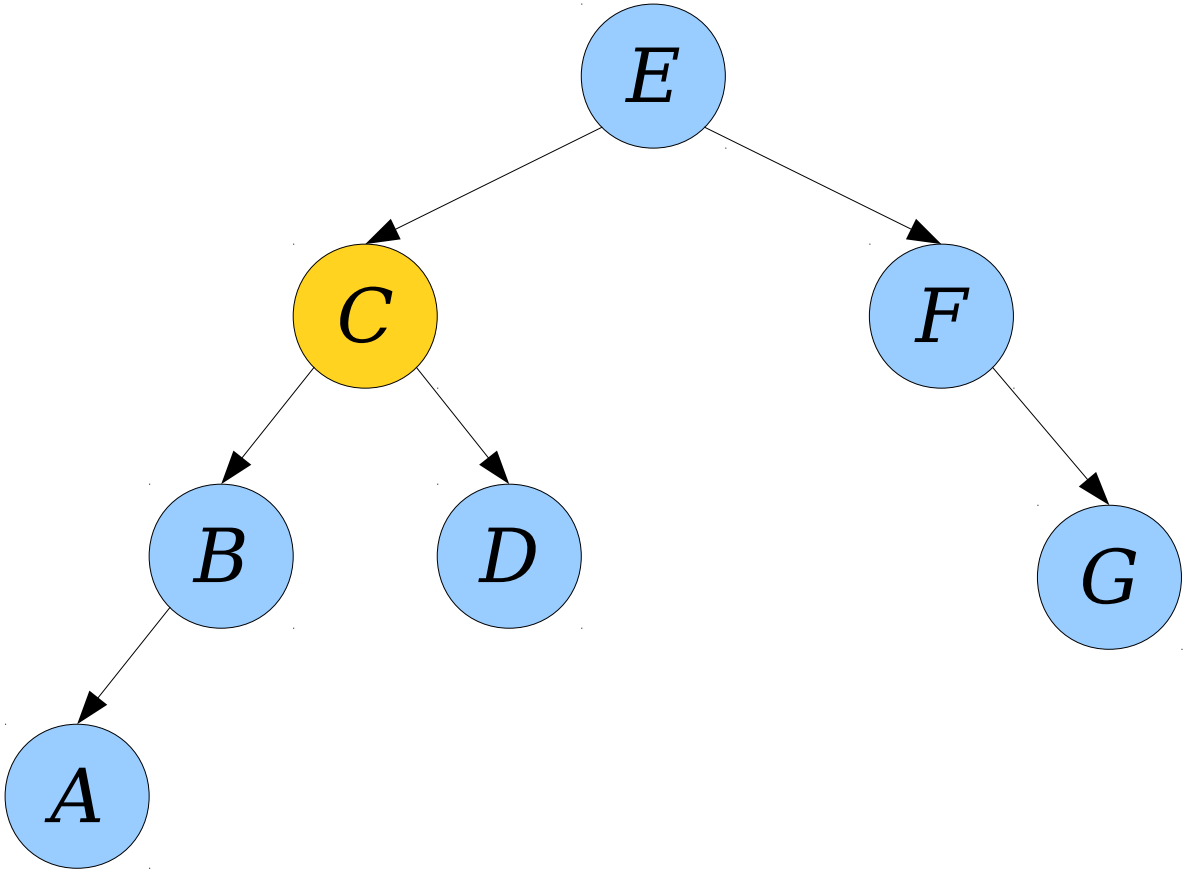
An Initial Approach



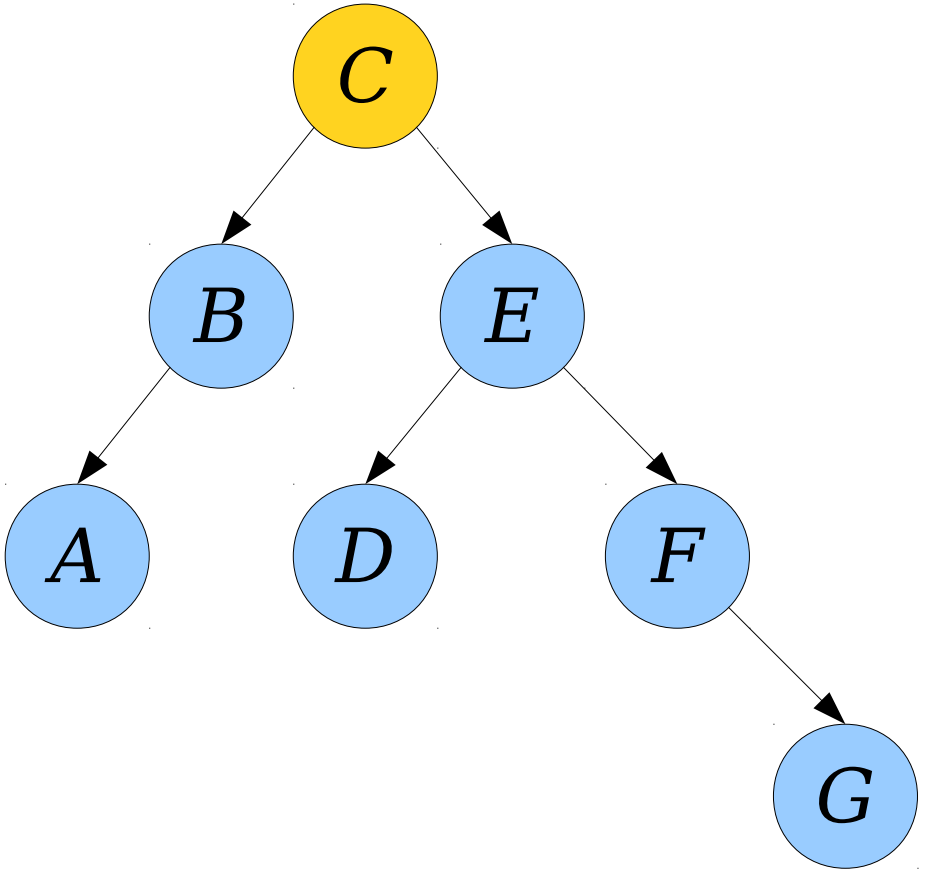
An Initial Approach



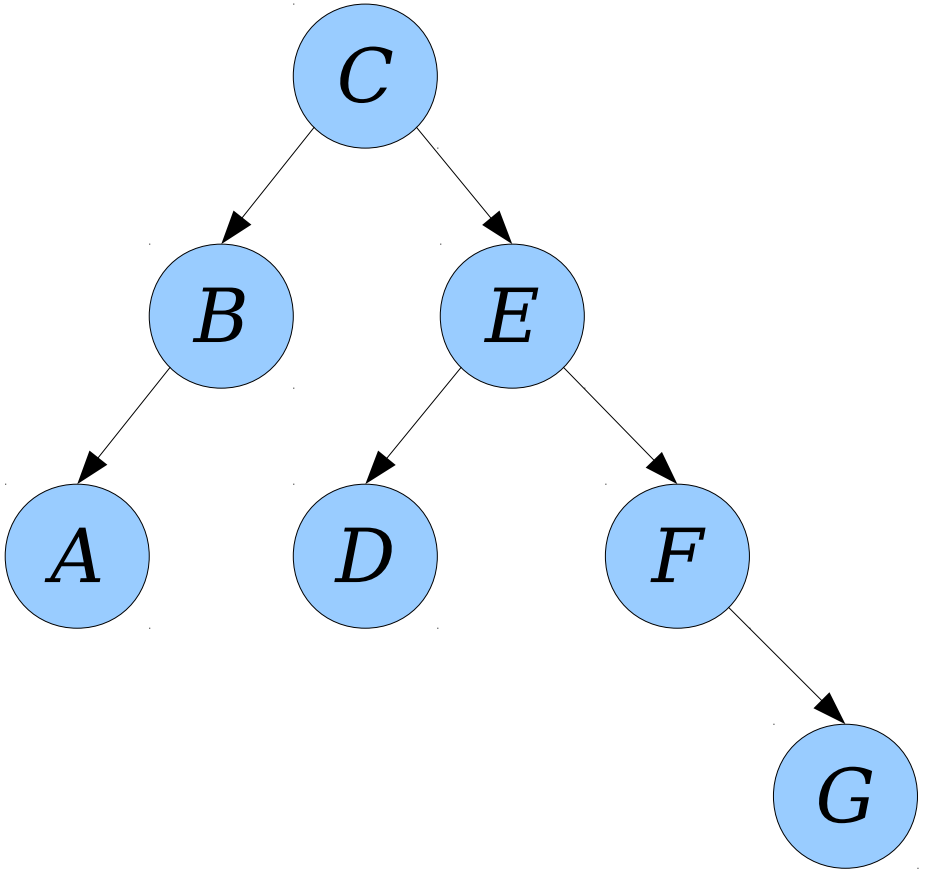
An Initial Approach



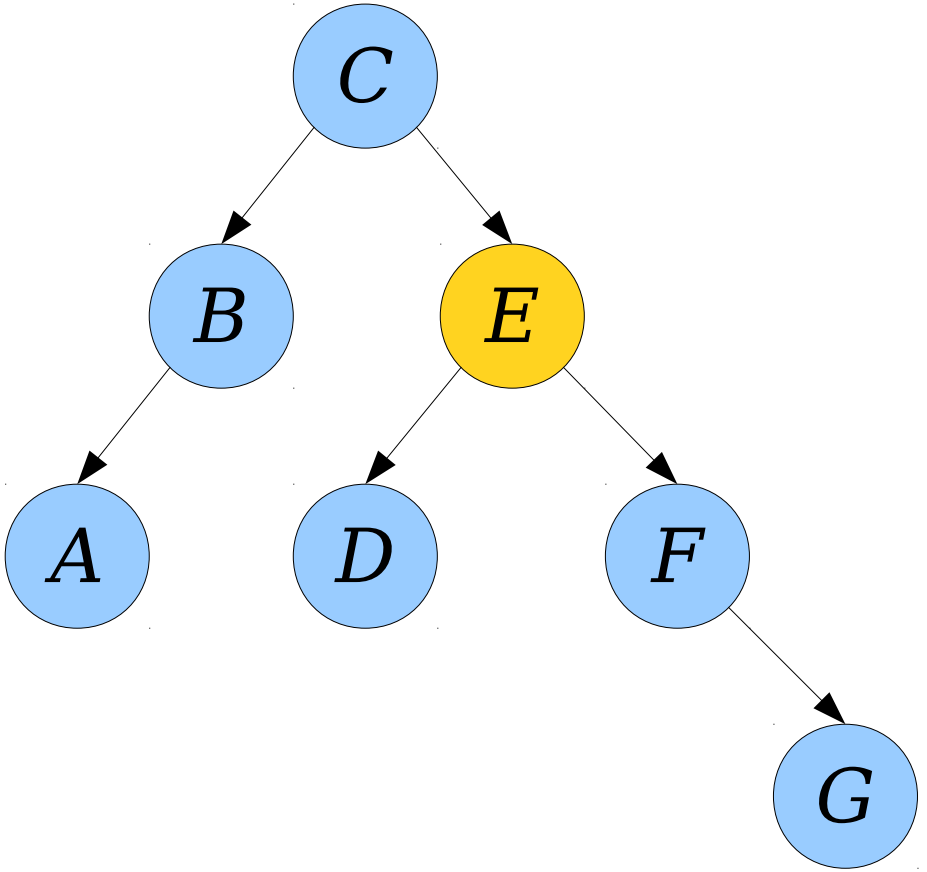
An Initial Approach



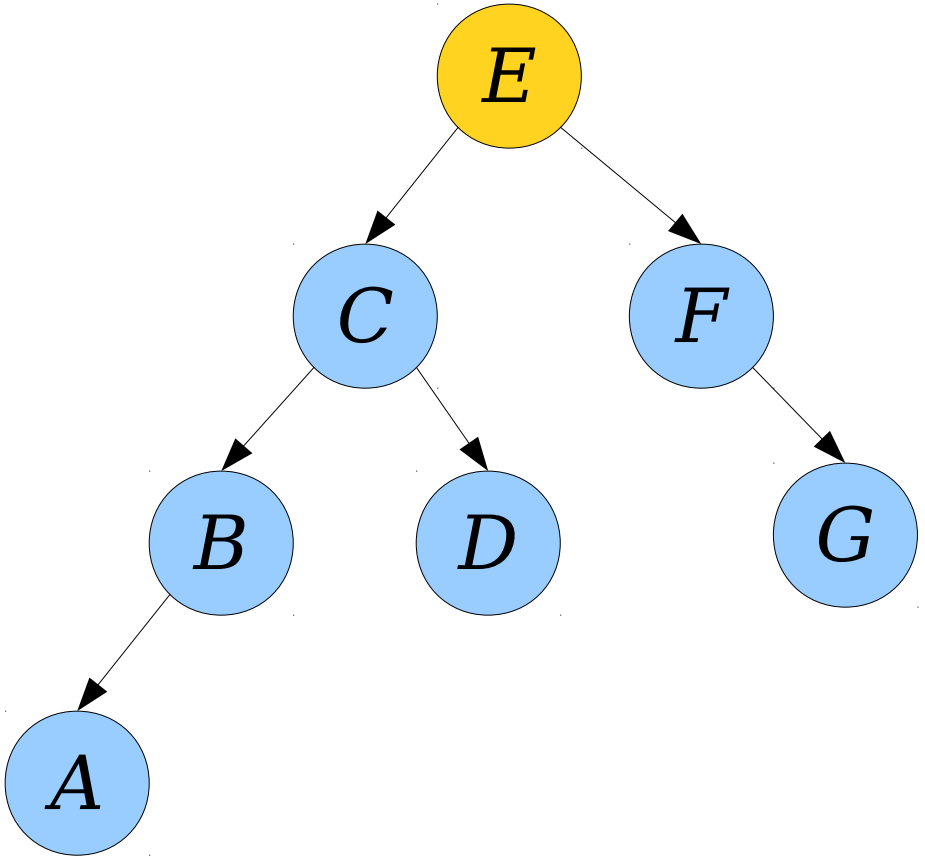
An Initial Approach



An Initial Approach



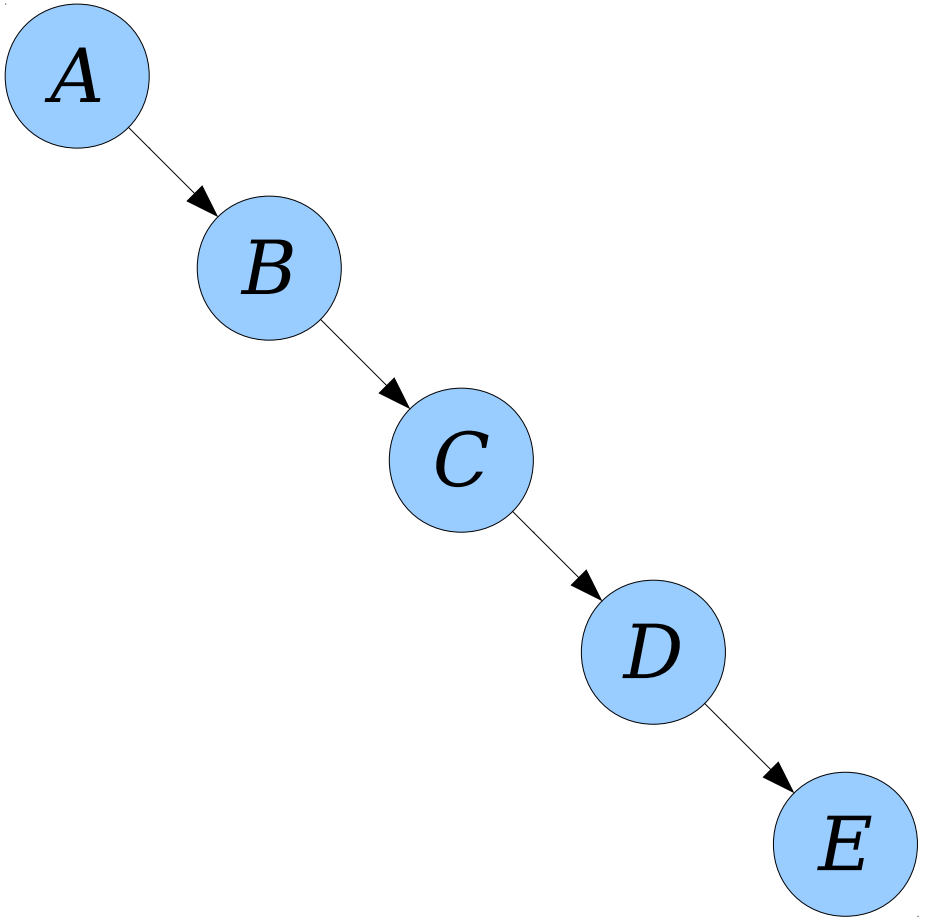
An Initial Approach



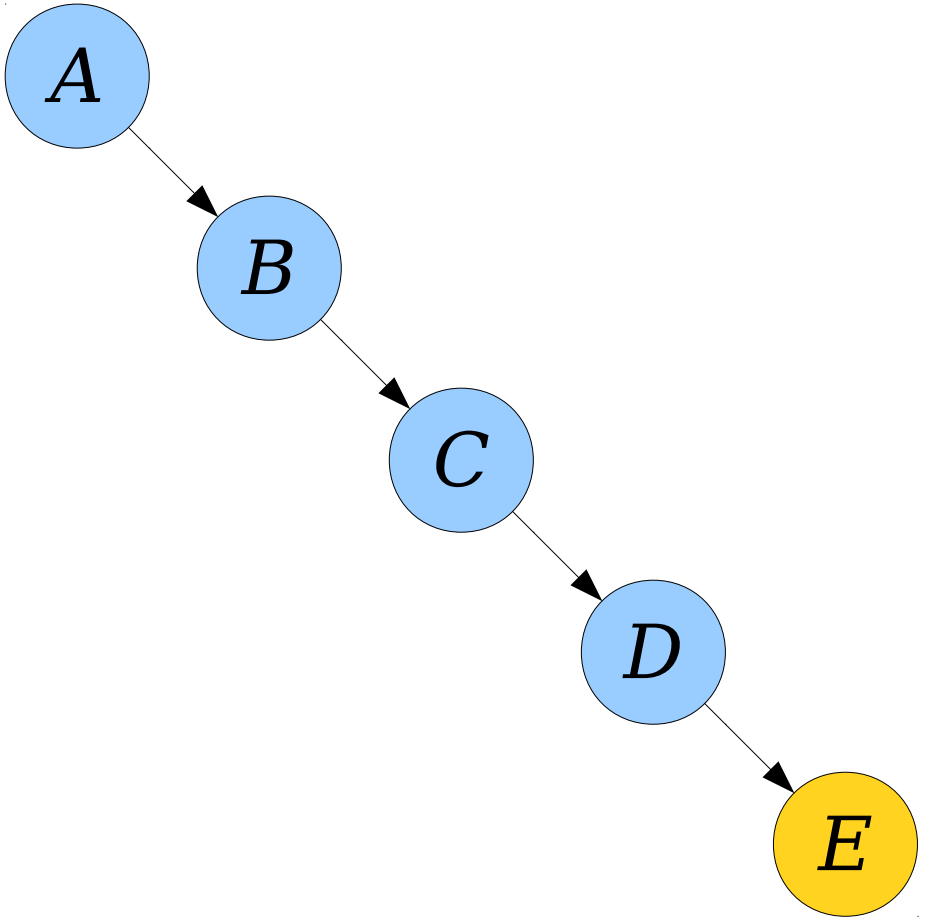
An Initial Idea

- Begin with an arbitrary BST.
- After looking up an element, repeatedly rotate that element with its parent until it becomes the root.
- ***Intuition:***
  - Recently-accessed elements will be up near the root of the tree, lowering access time.
  - Unused elements stay low in the tree.

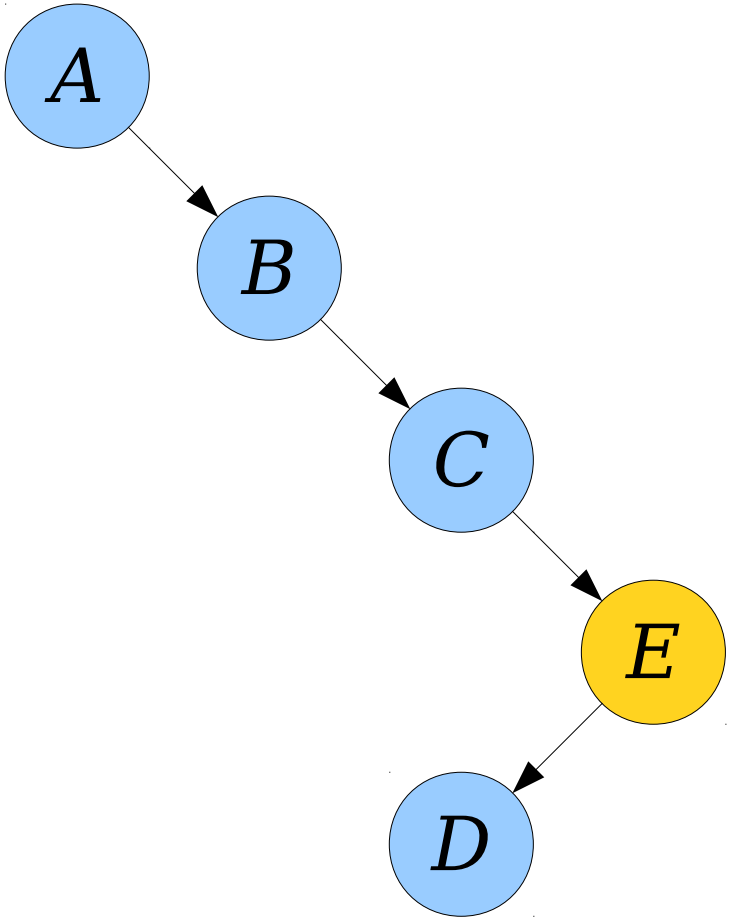
The Problem



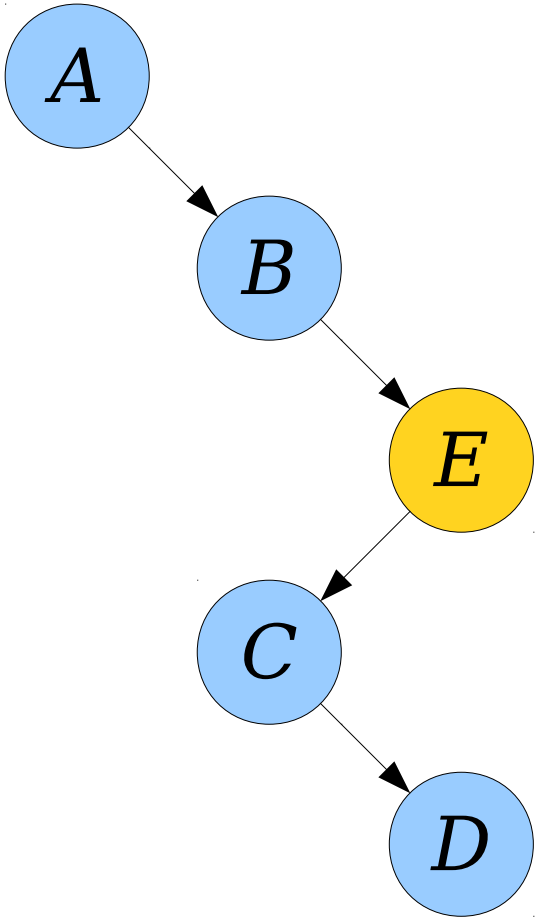
The Problem



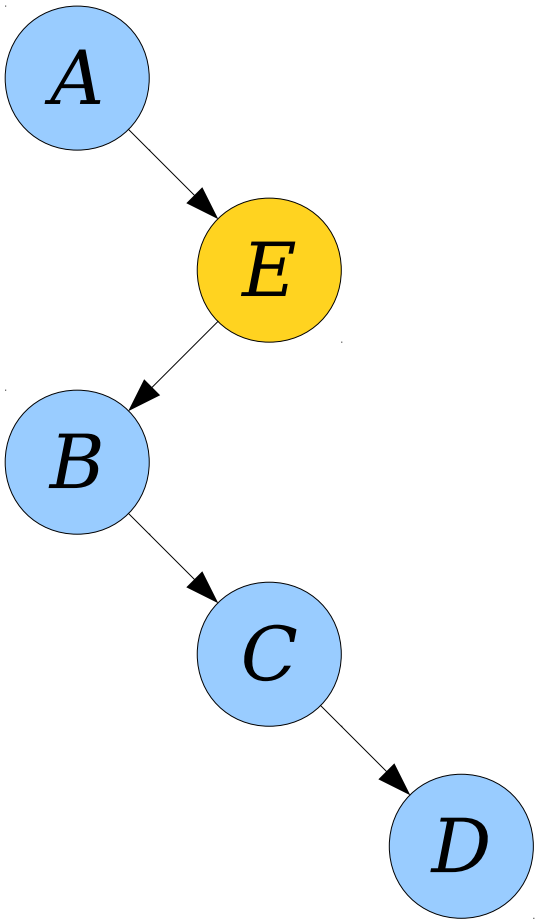
The Problem



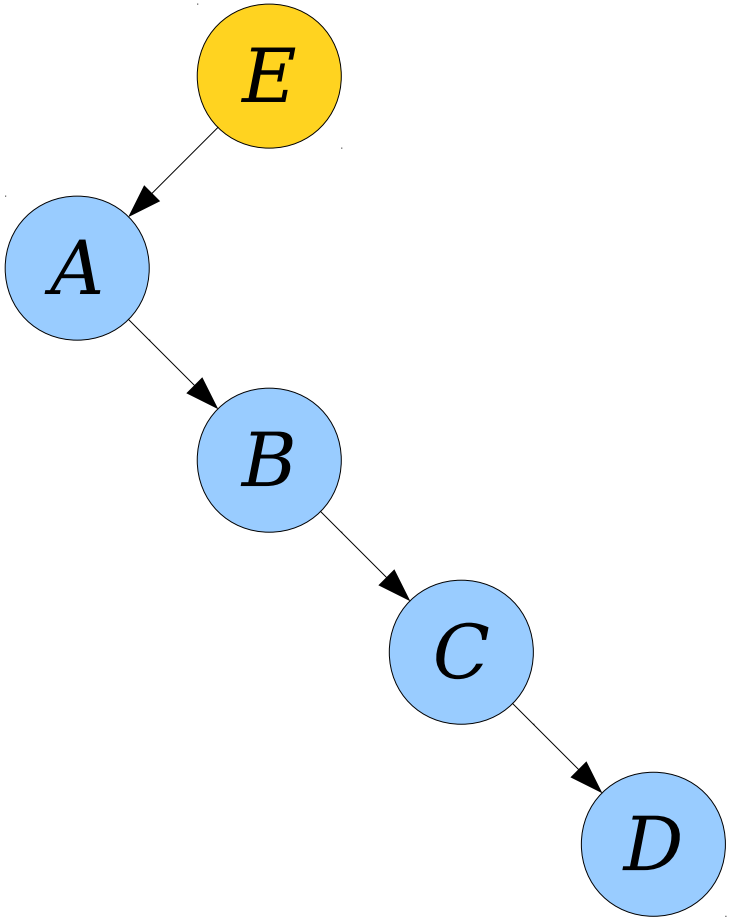
The Problem



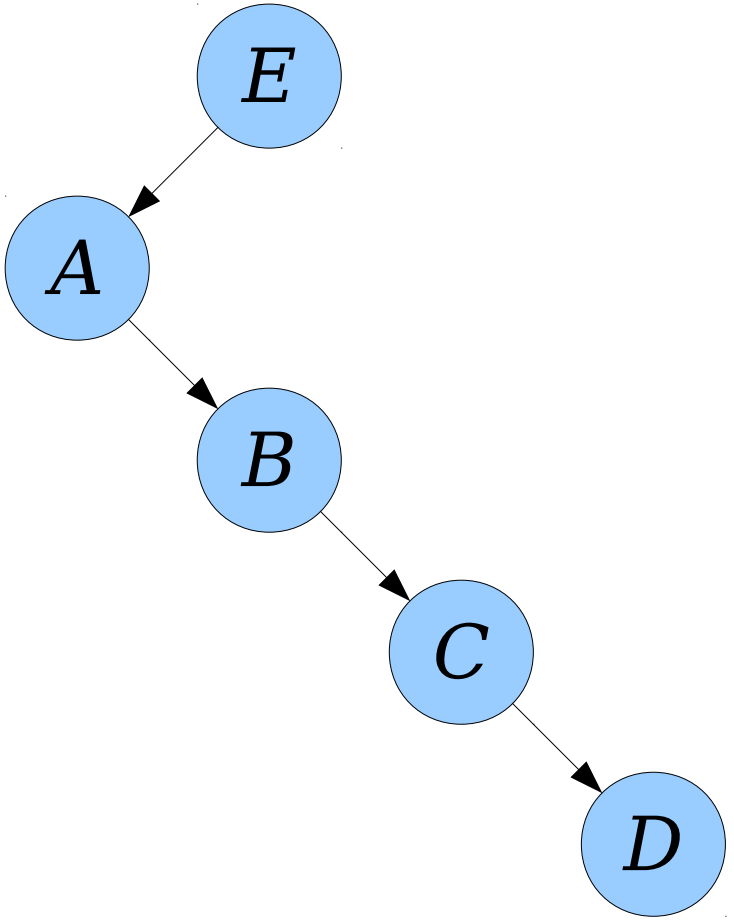
The Problem



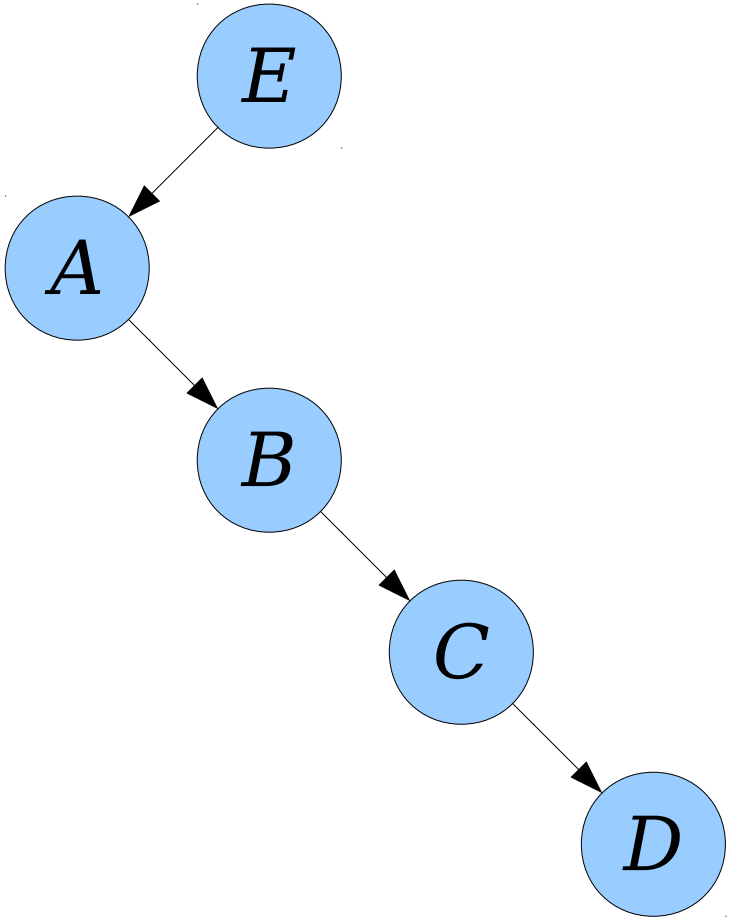
The Problem



The Problem

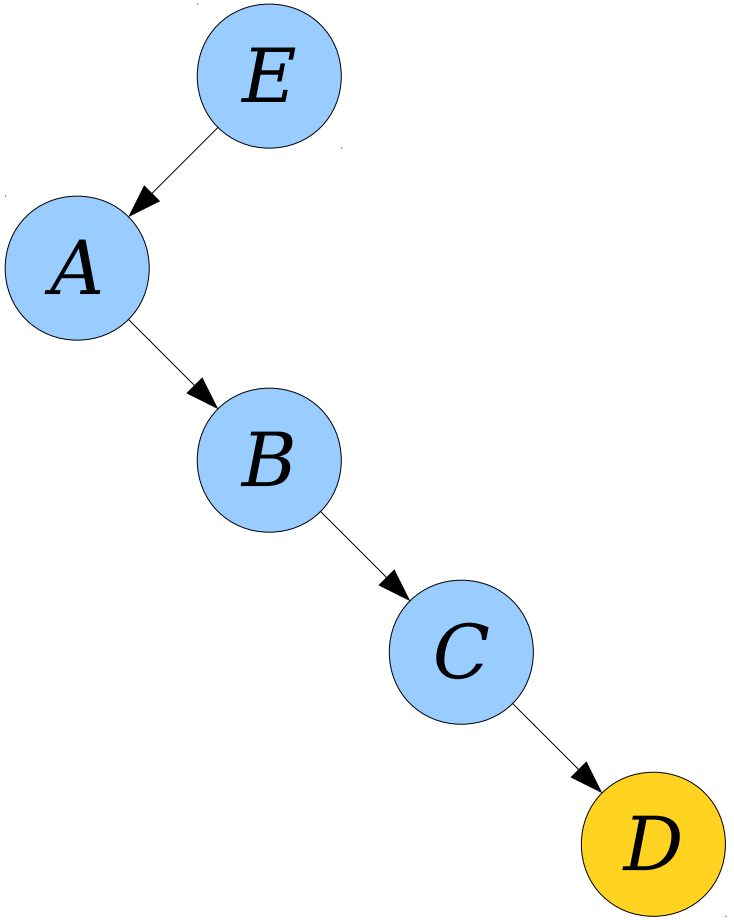


The Problem

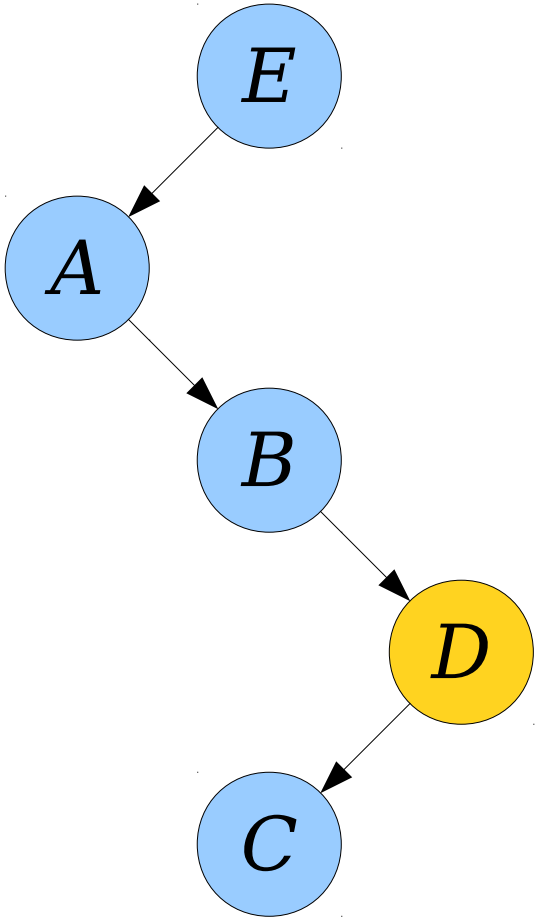


Rotations Needed: **5**

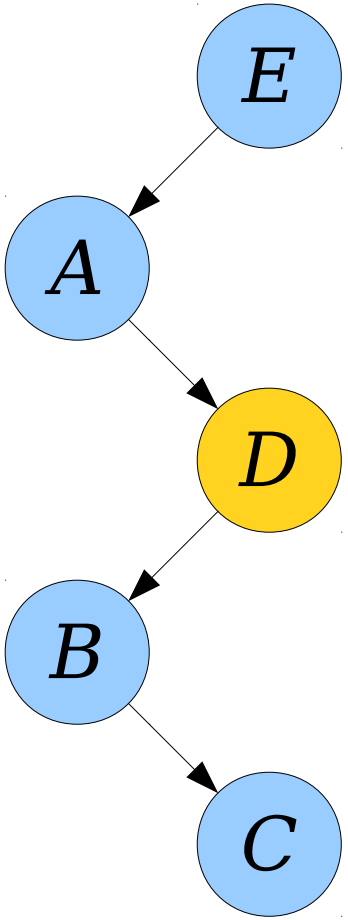
The Problem



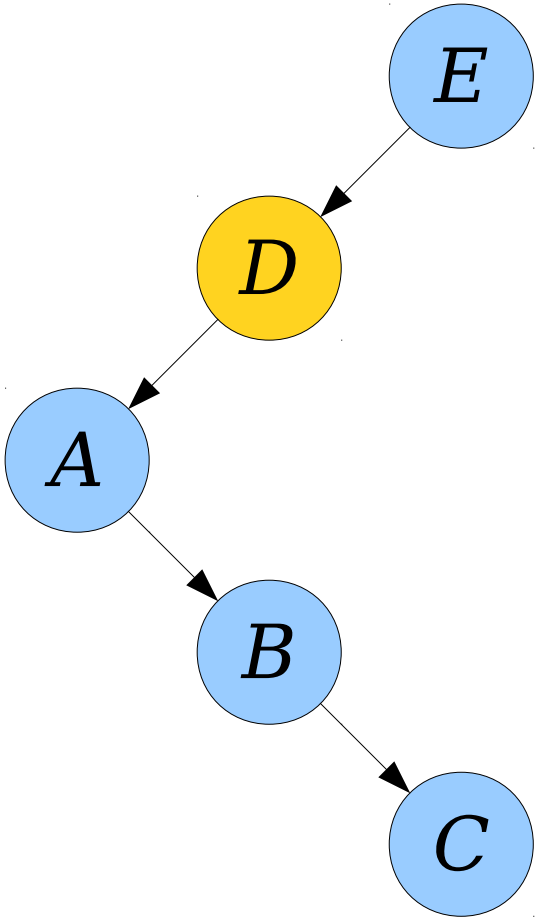
The Problem



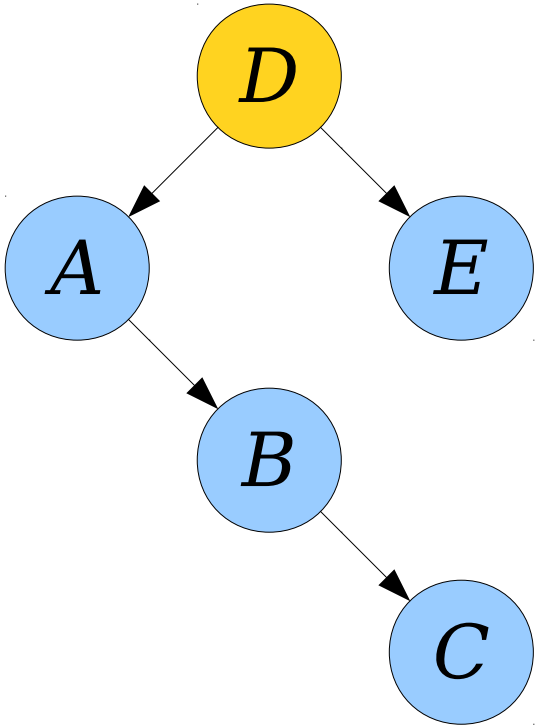
The Problem



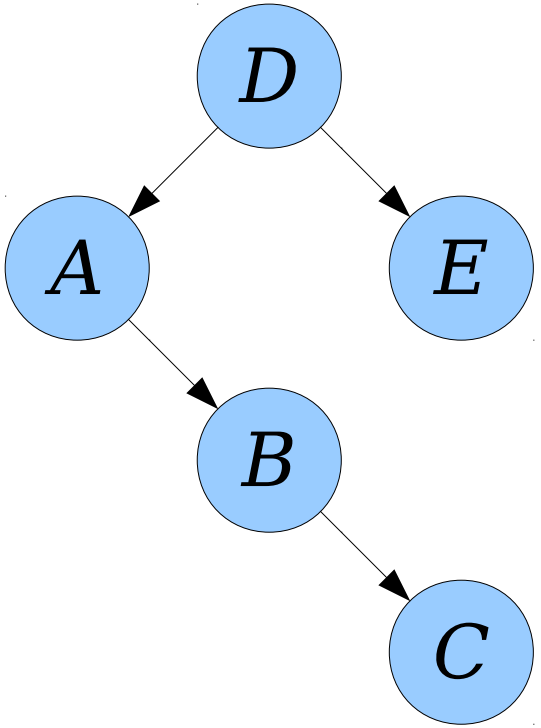
The Problem



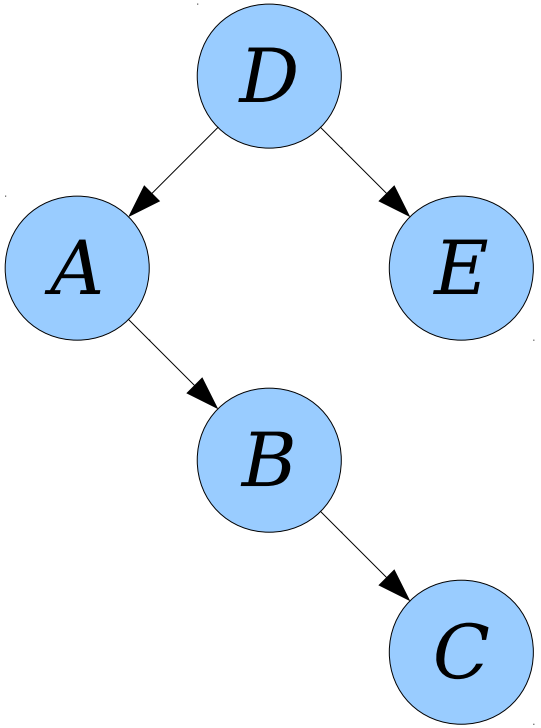
The Problem



The Problem

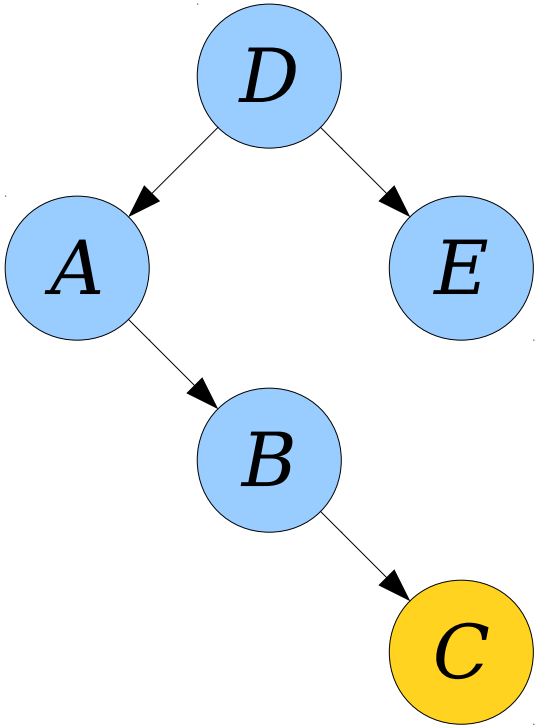


The Problem

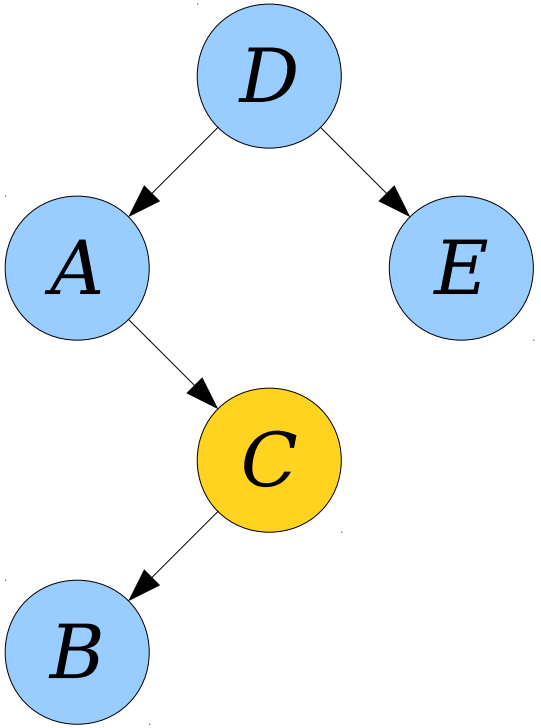


Rotations Needed: **4**

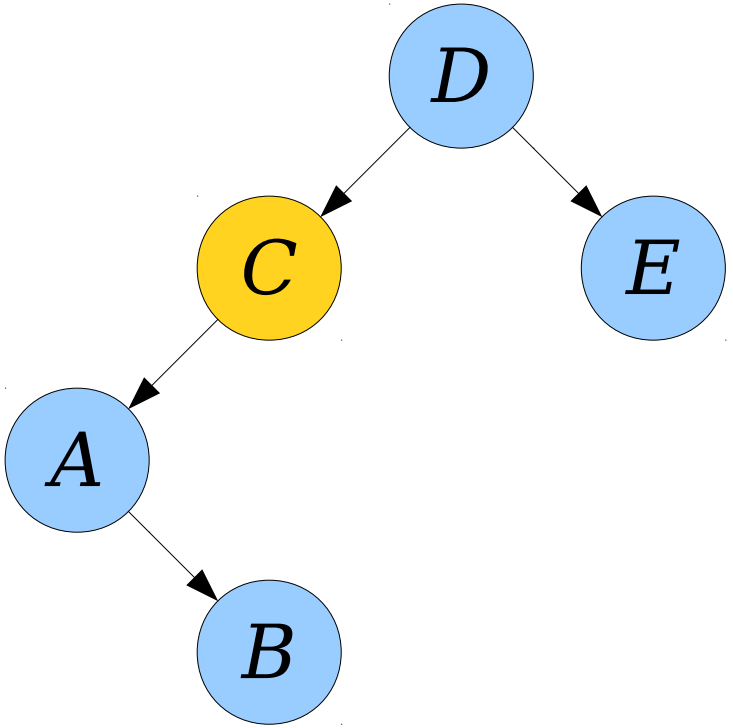
The Problem



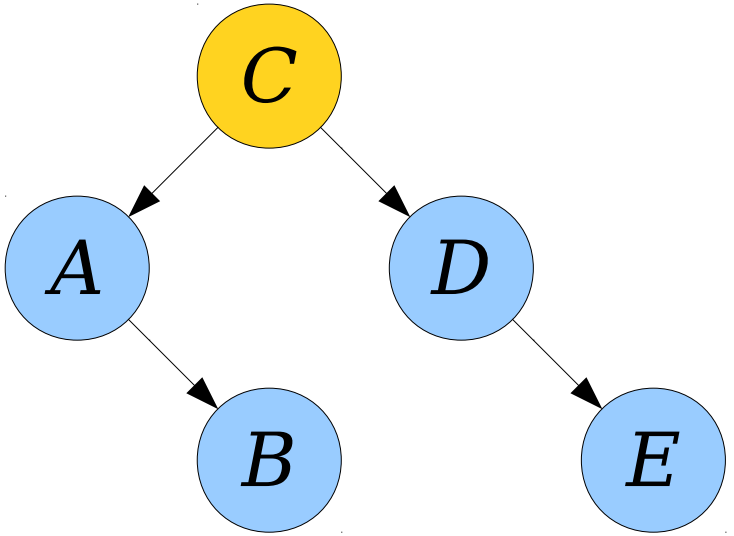
The Problem



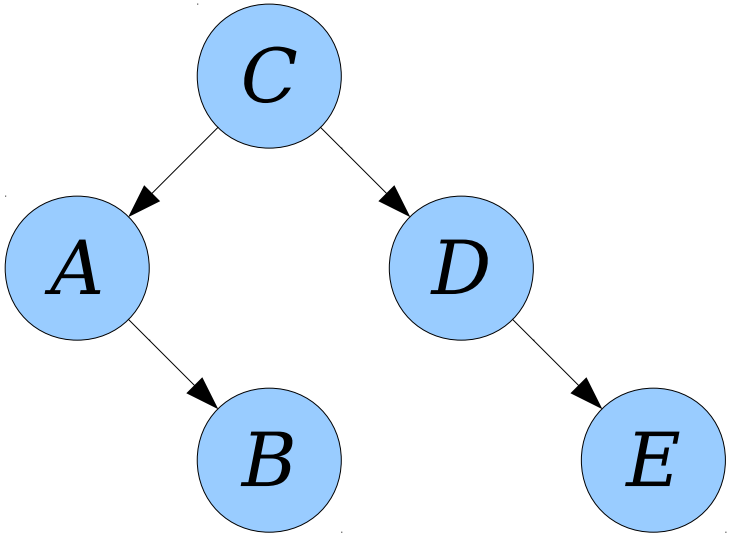
The Problem



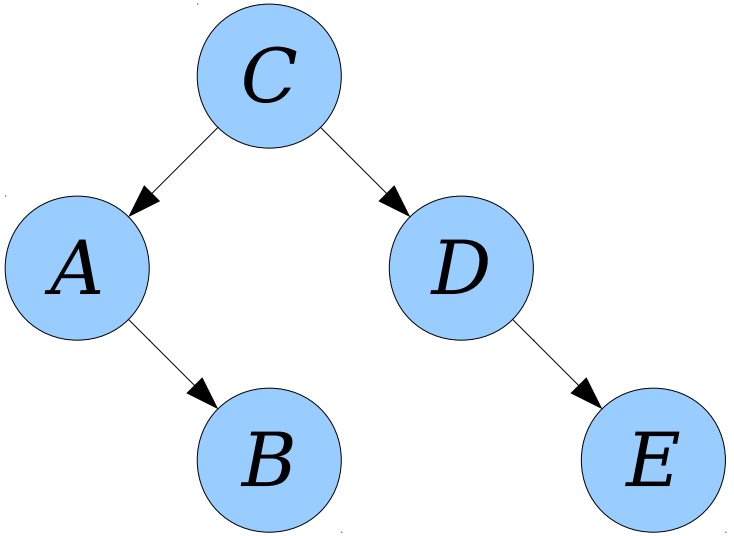
The Problem



The Problem

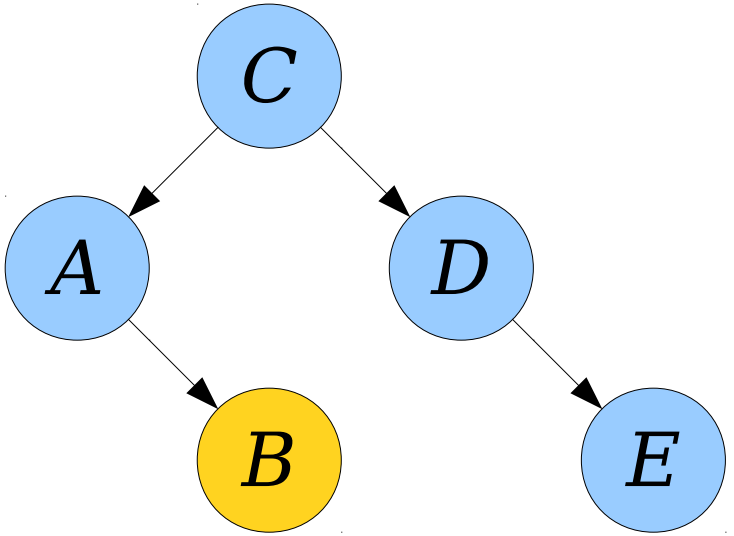


The Problem

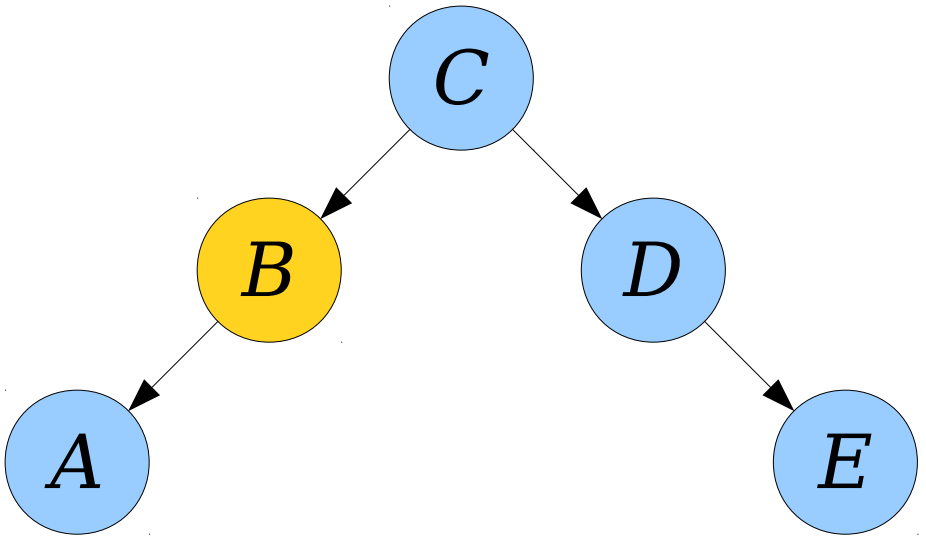


Rotations Needed: **3**

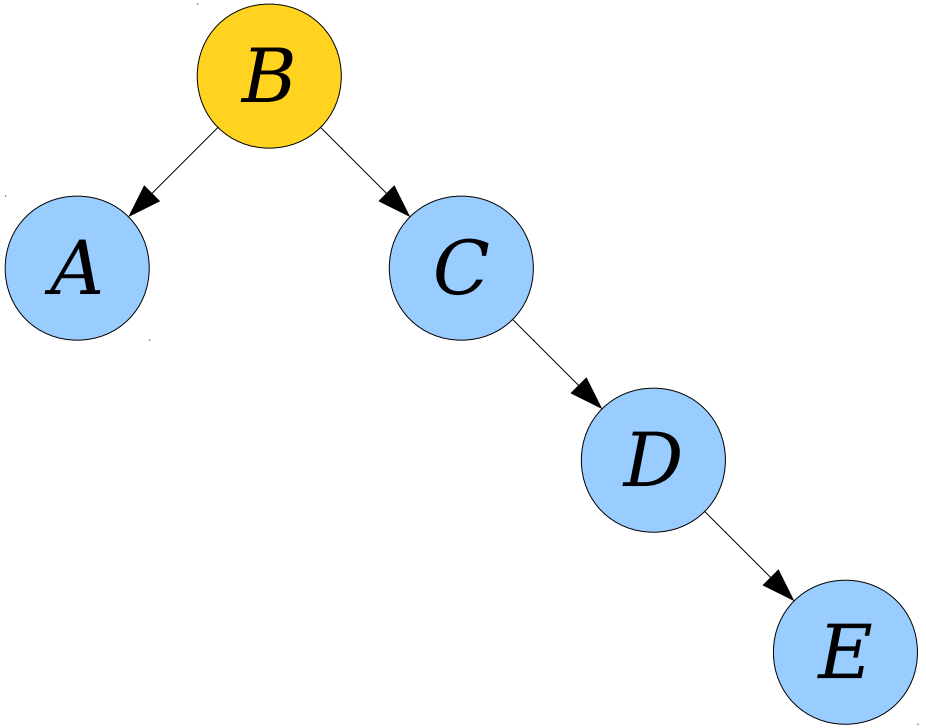
The Problem



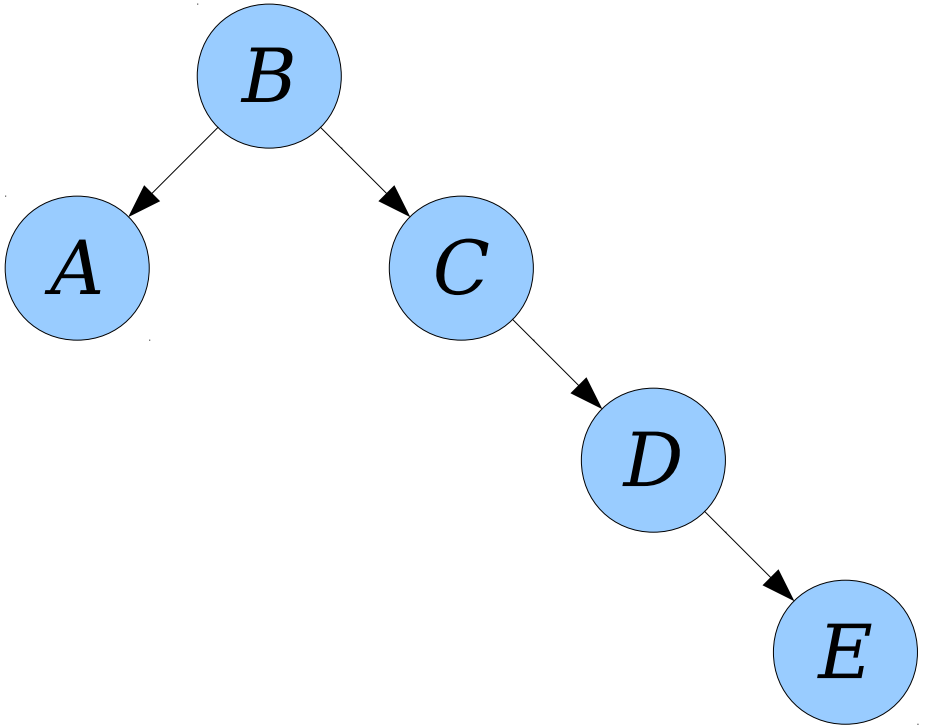
The Problem



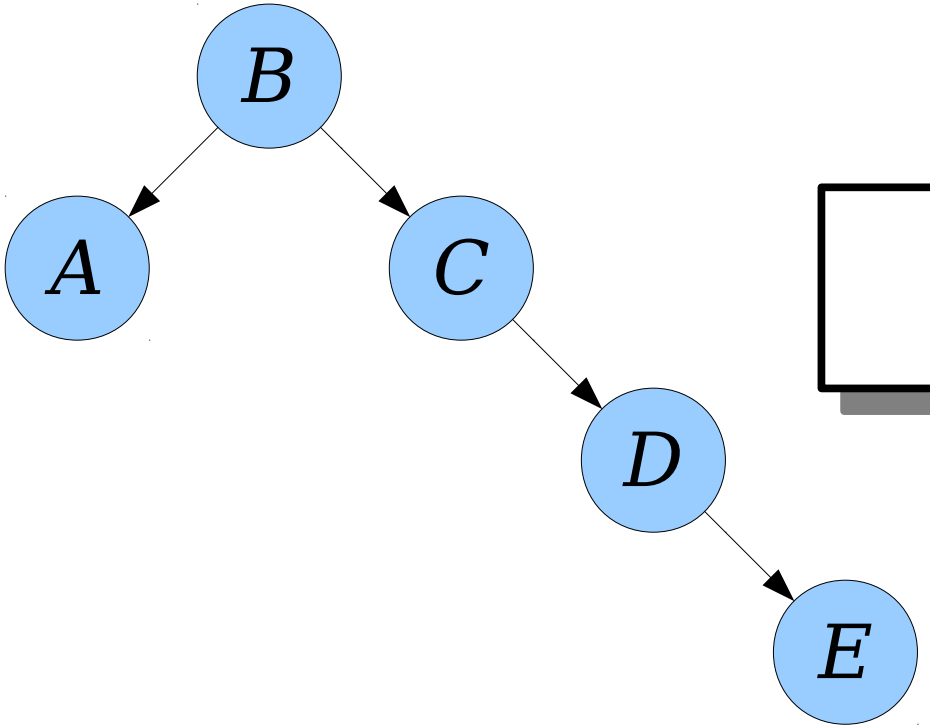
The Problem



The Problem

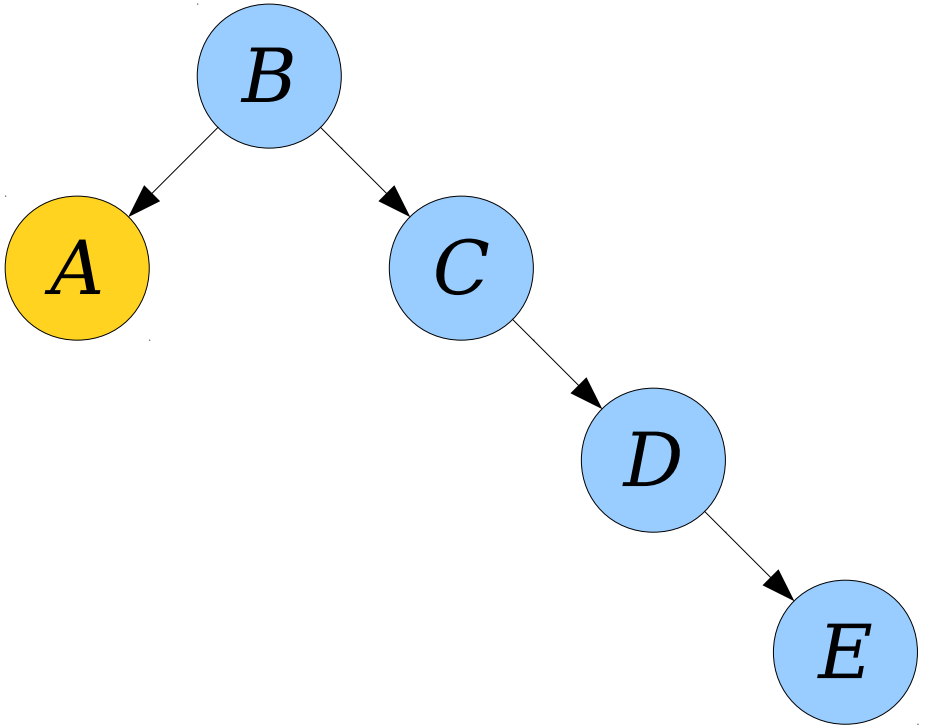


The Problem

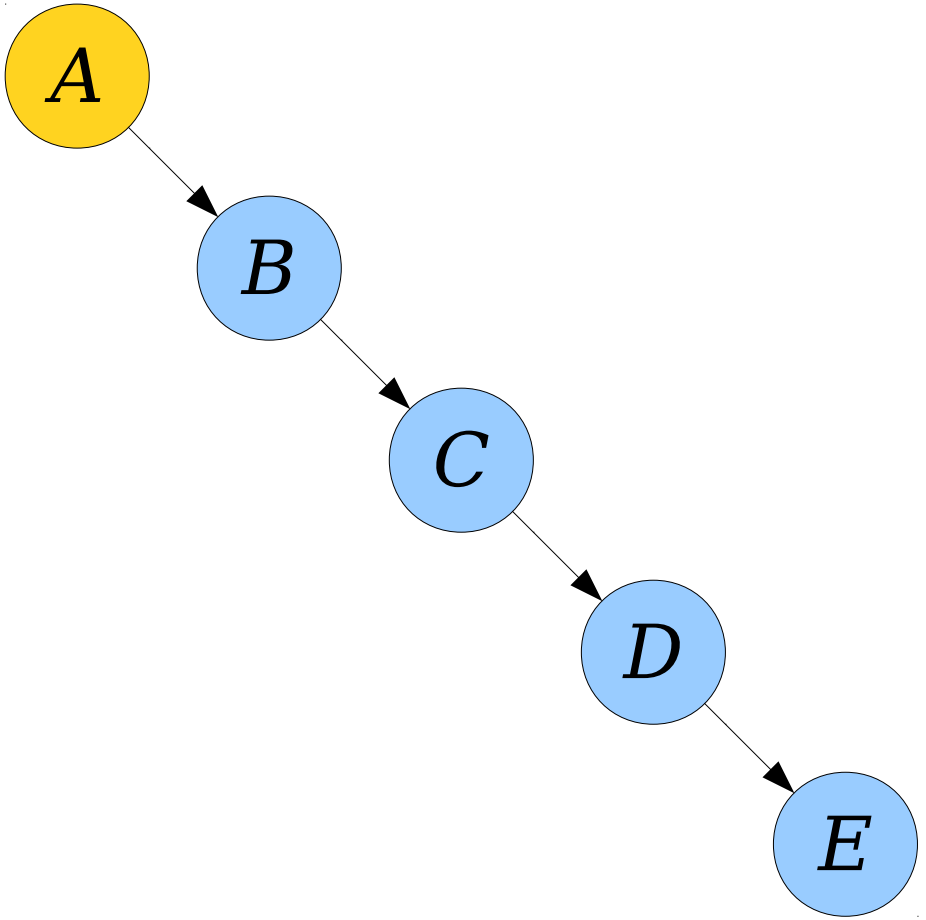


Rotations Needed: **2**

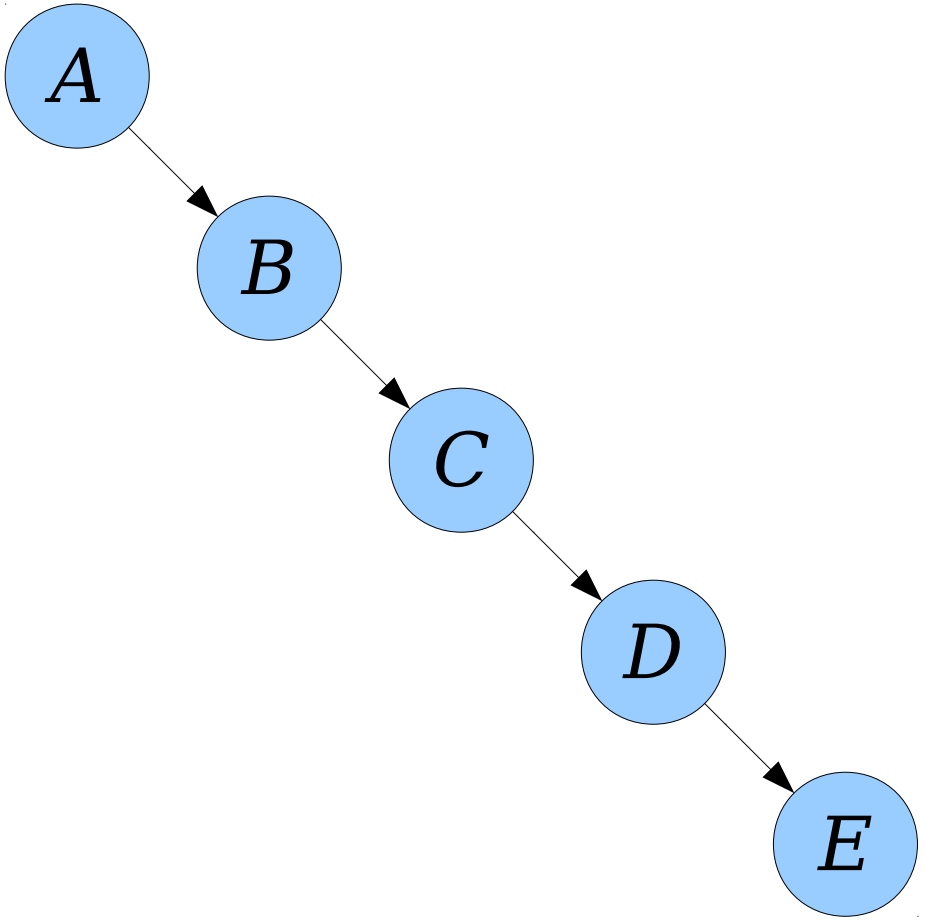
The Problem



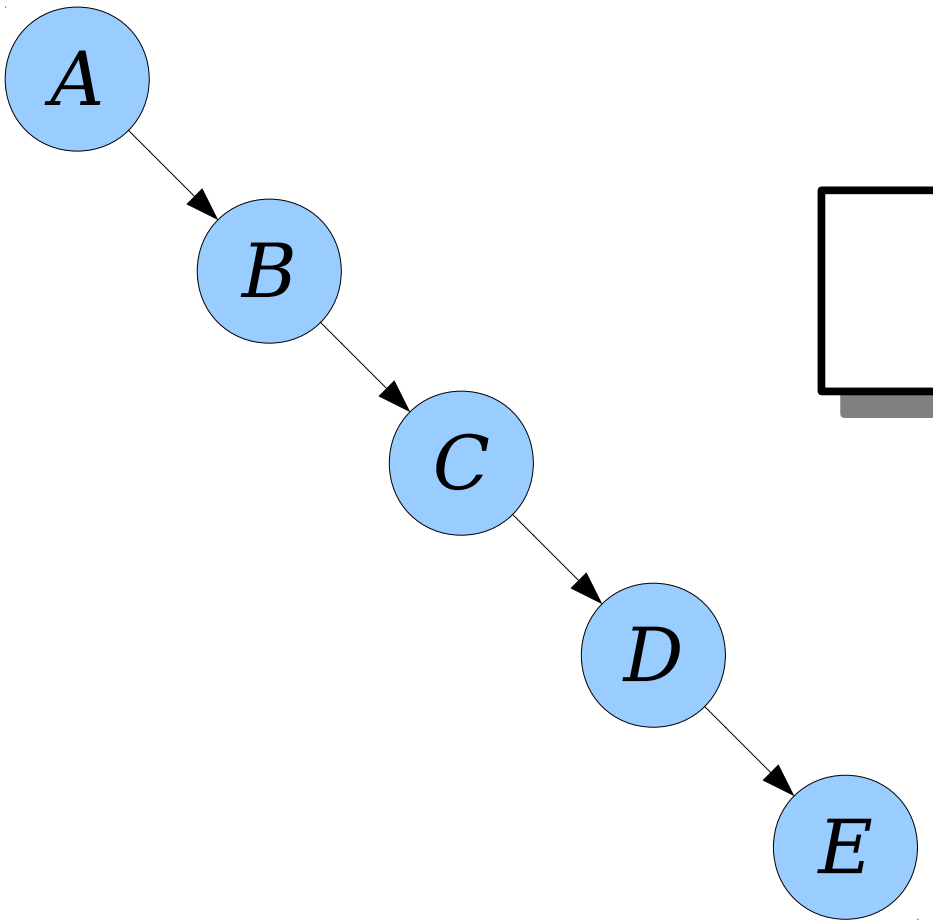
The Problem



The Problem

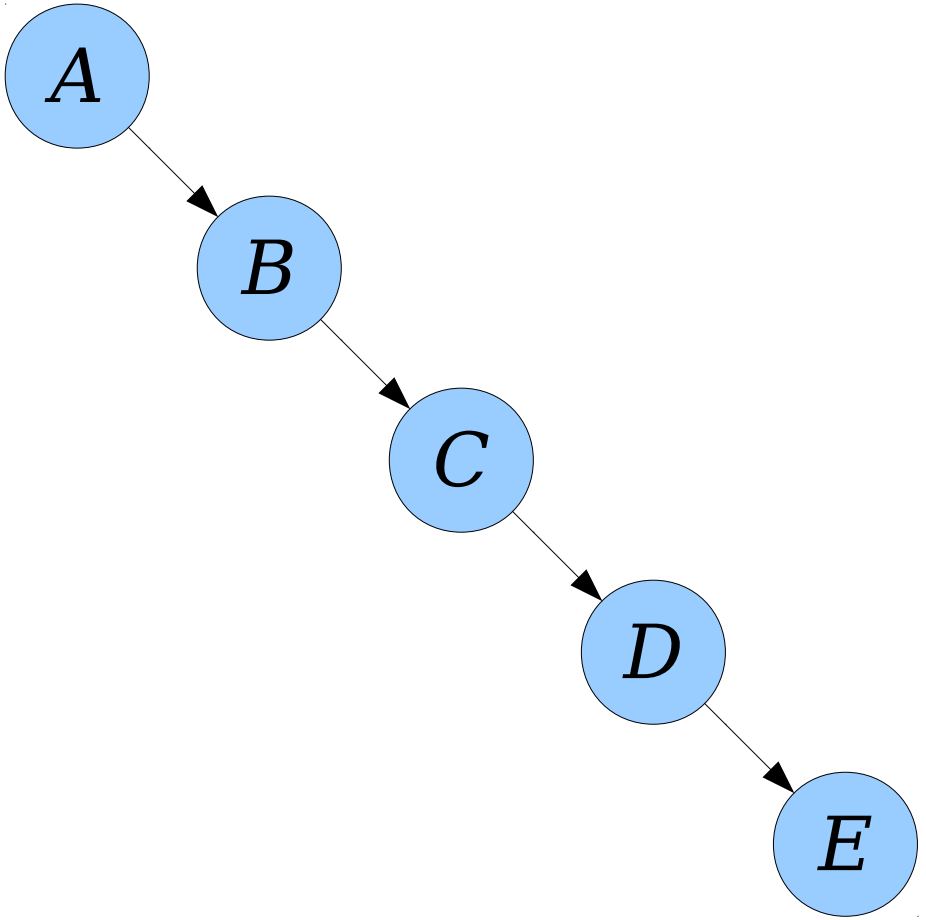


The Problem

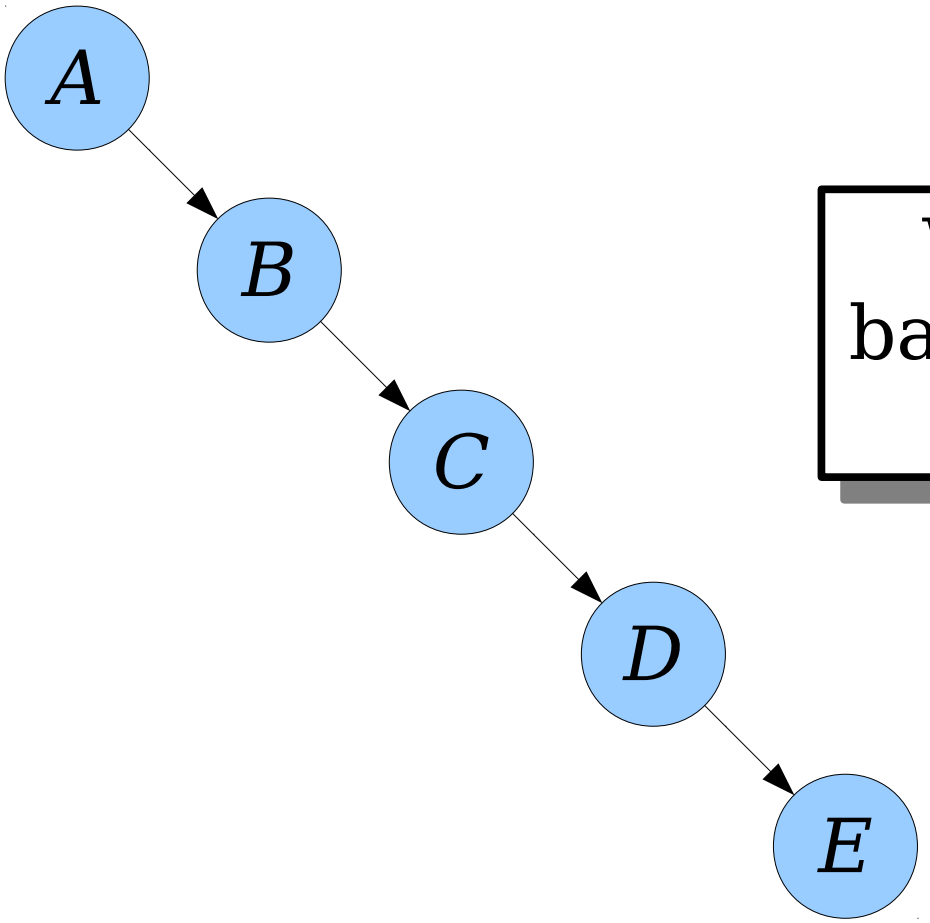


Rotations Needed: **1**

The Problem



The Problem



We're right back where we started!

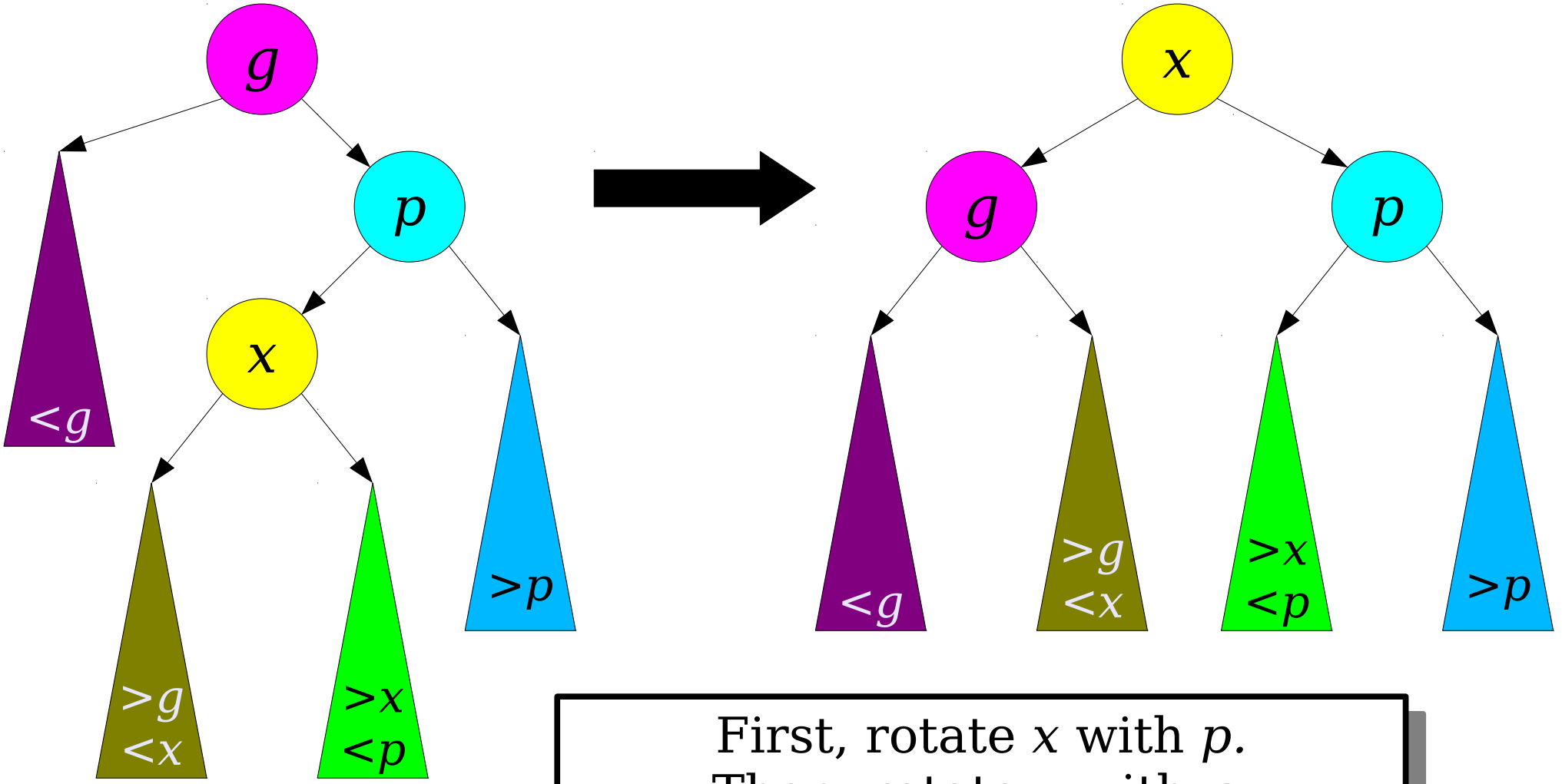
The Problem

- The “rotate to root” method might result in n accesses taking time $\Theta(n^2)$.
- ***Why?***
- Rotating an element x to the root significantly “helps” x, but “hurts” the rest of the tree.
- Most of the nodes on the access path to x have depth that increases or is unchanged.

A More Balanced Approach

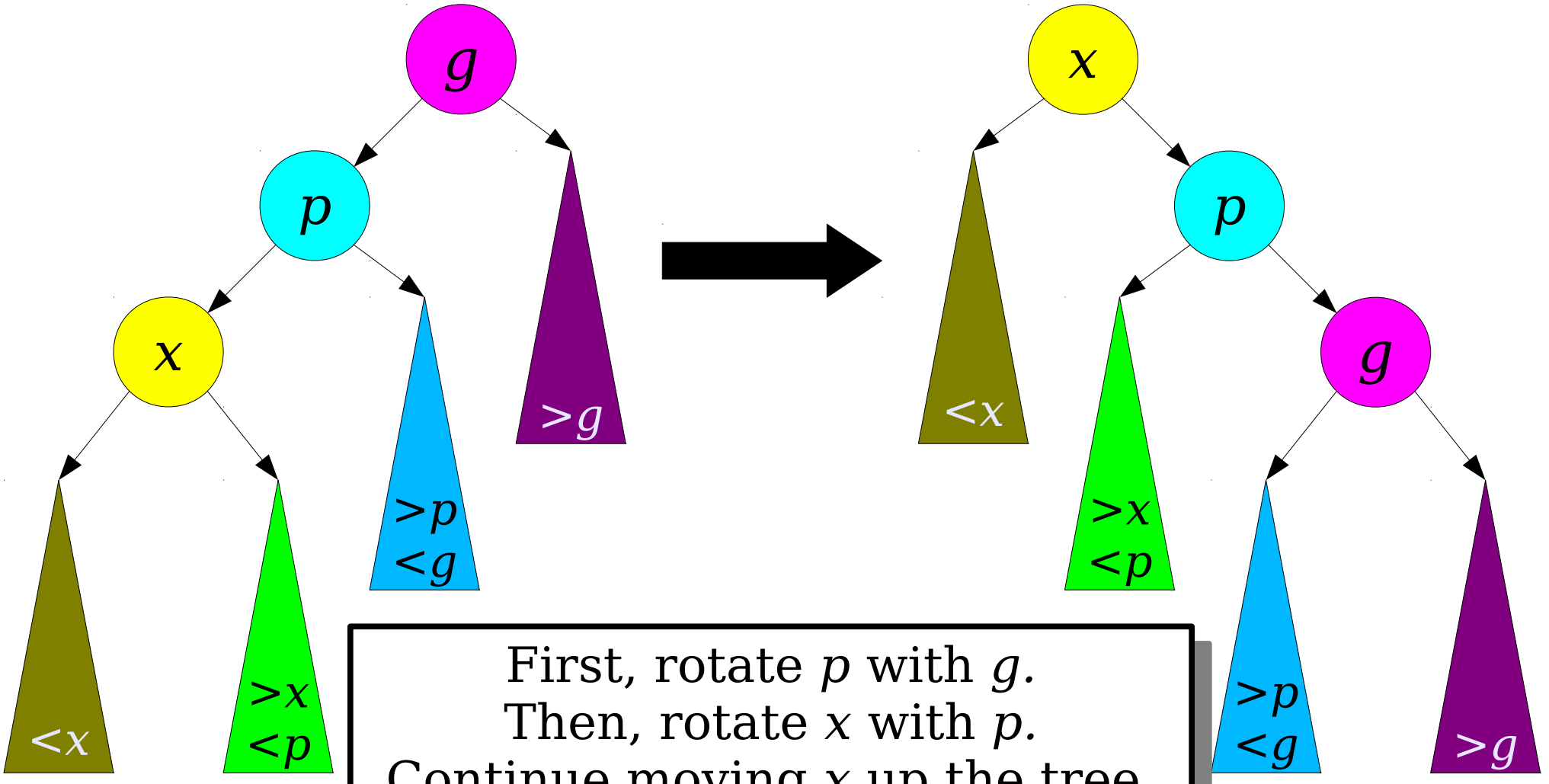
- In 1983, Daniel Sleator and Robert Tarjan invented an operation called ***splaying***.
- Splaying rotates an element to the root of the tree, but does so in a way that's more “fair” to other nodes in the tree.
- Each splay works by applying one of three templates to determine which rotations to apply.

Case 1: Zig-Zag



First, rotate x with p.
Then, rotate x with g.
Continue moving x up the tree.

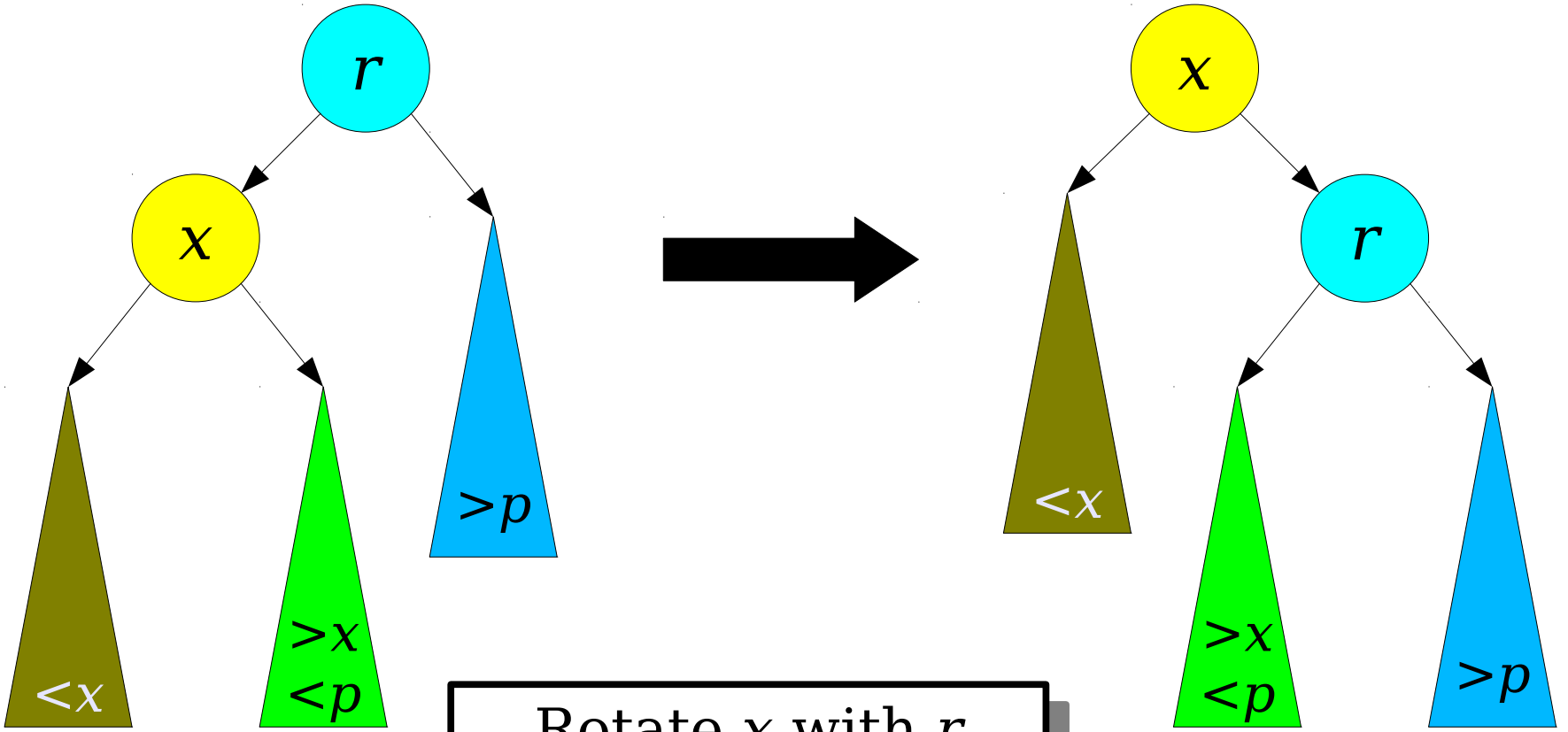
Case 2: Zig-Zig



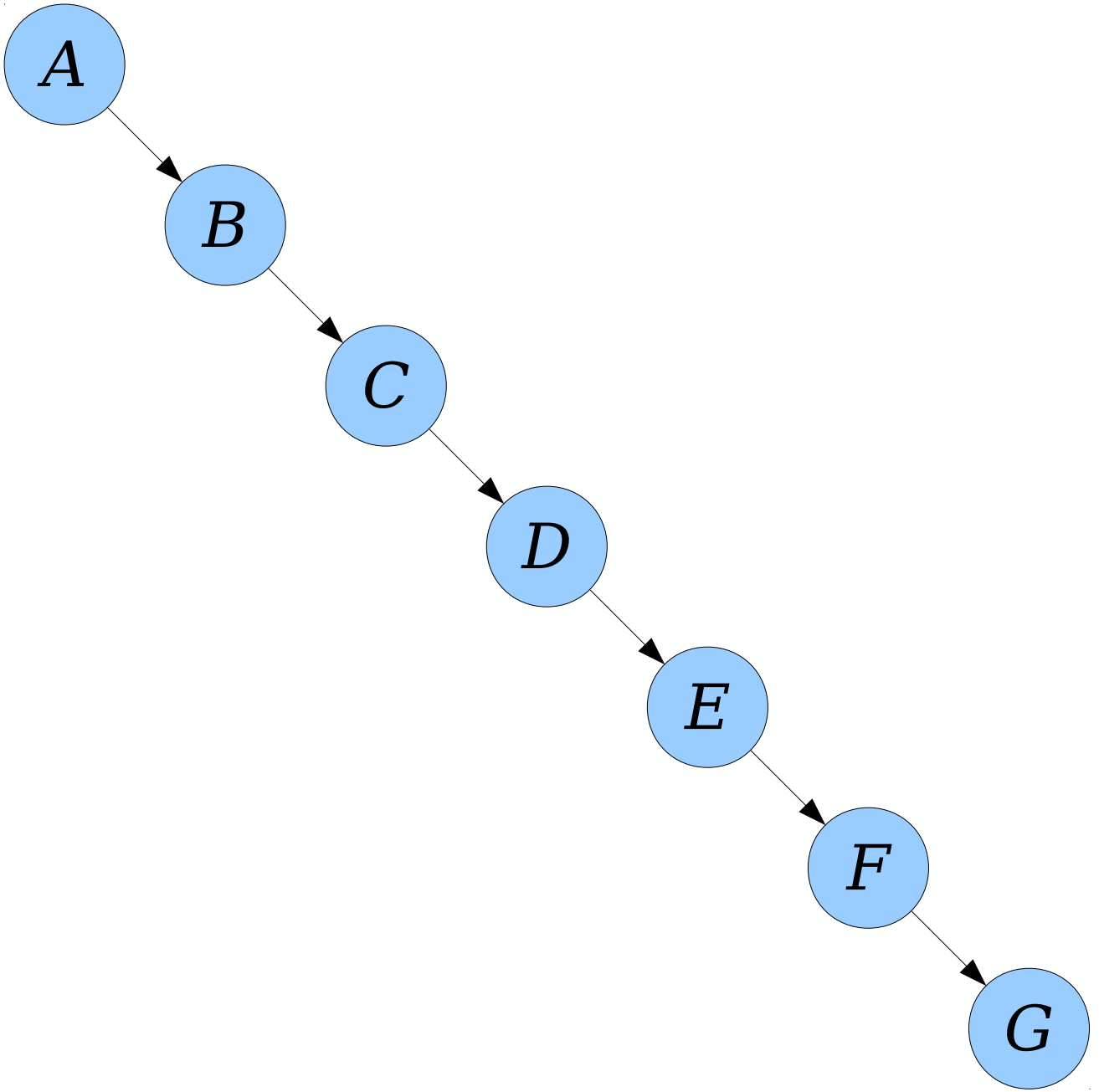
First, rotate p with g.
Then, rotate x with p.
Continue moving x up the tree.

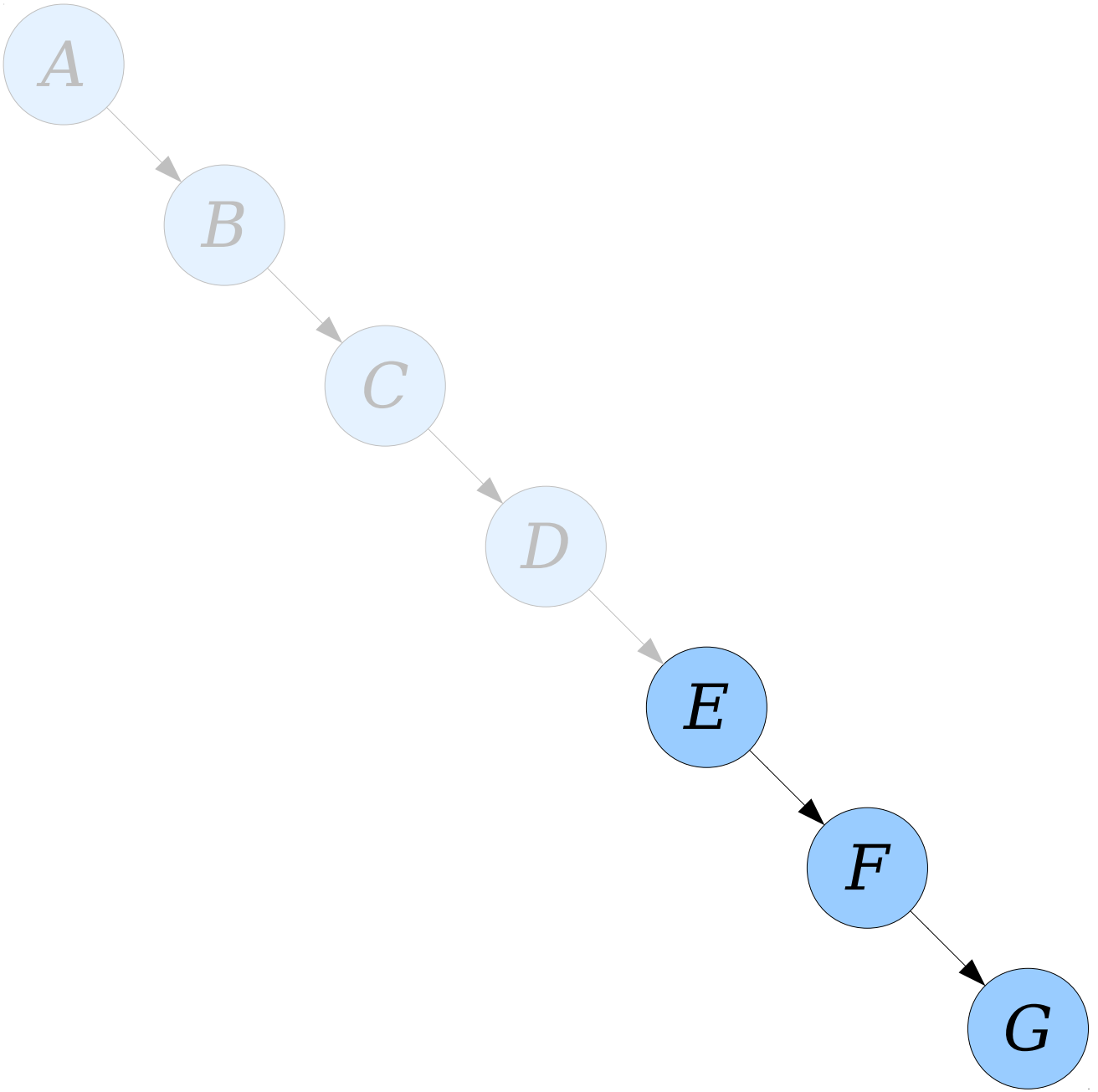
Case 3: Zig

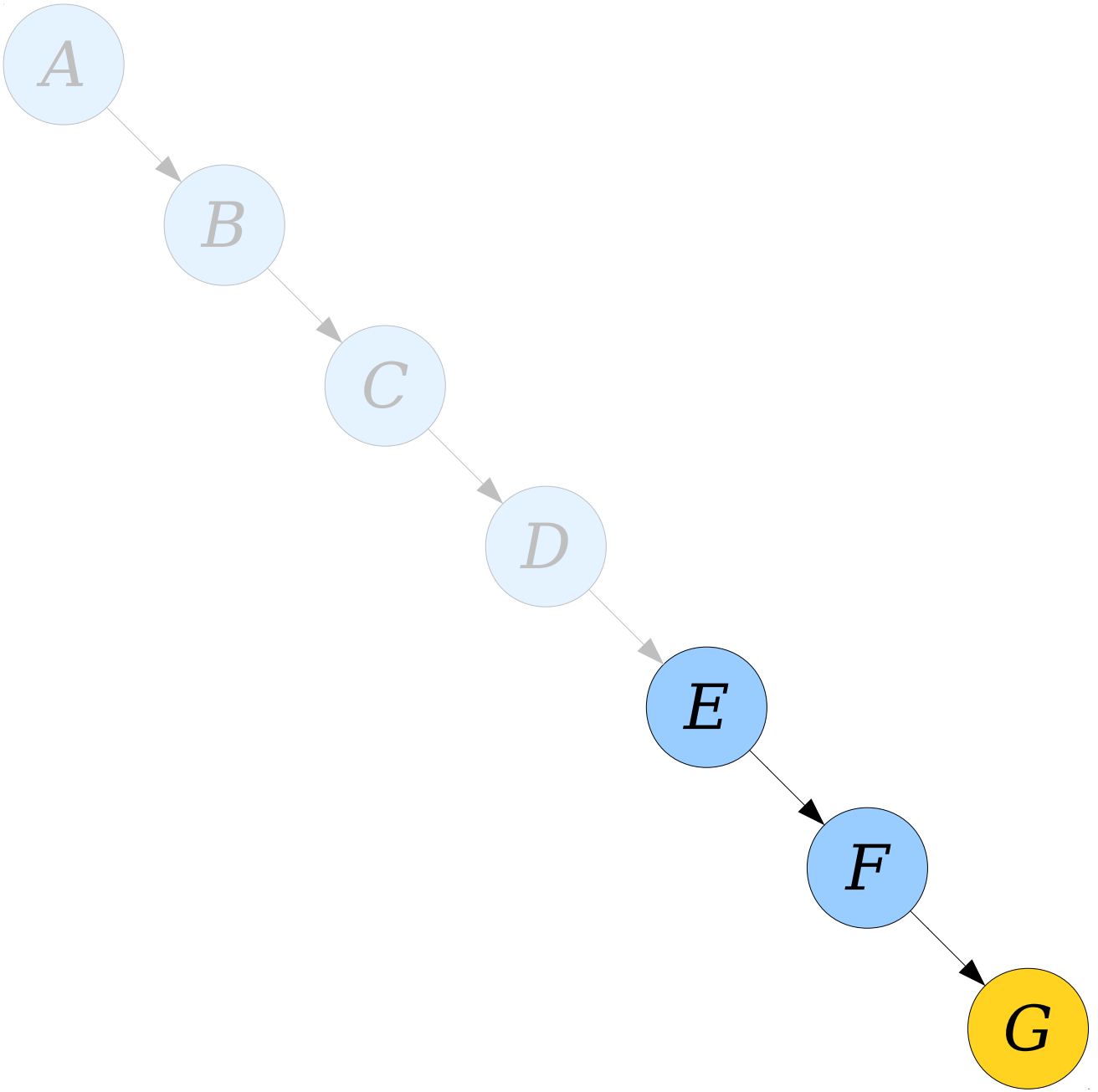
(Assume r is the tree root)

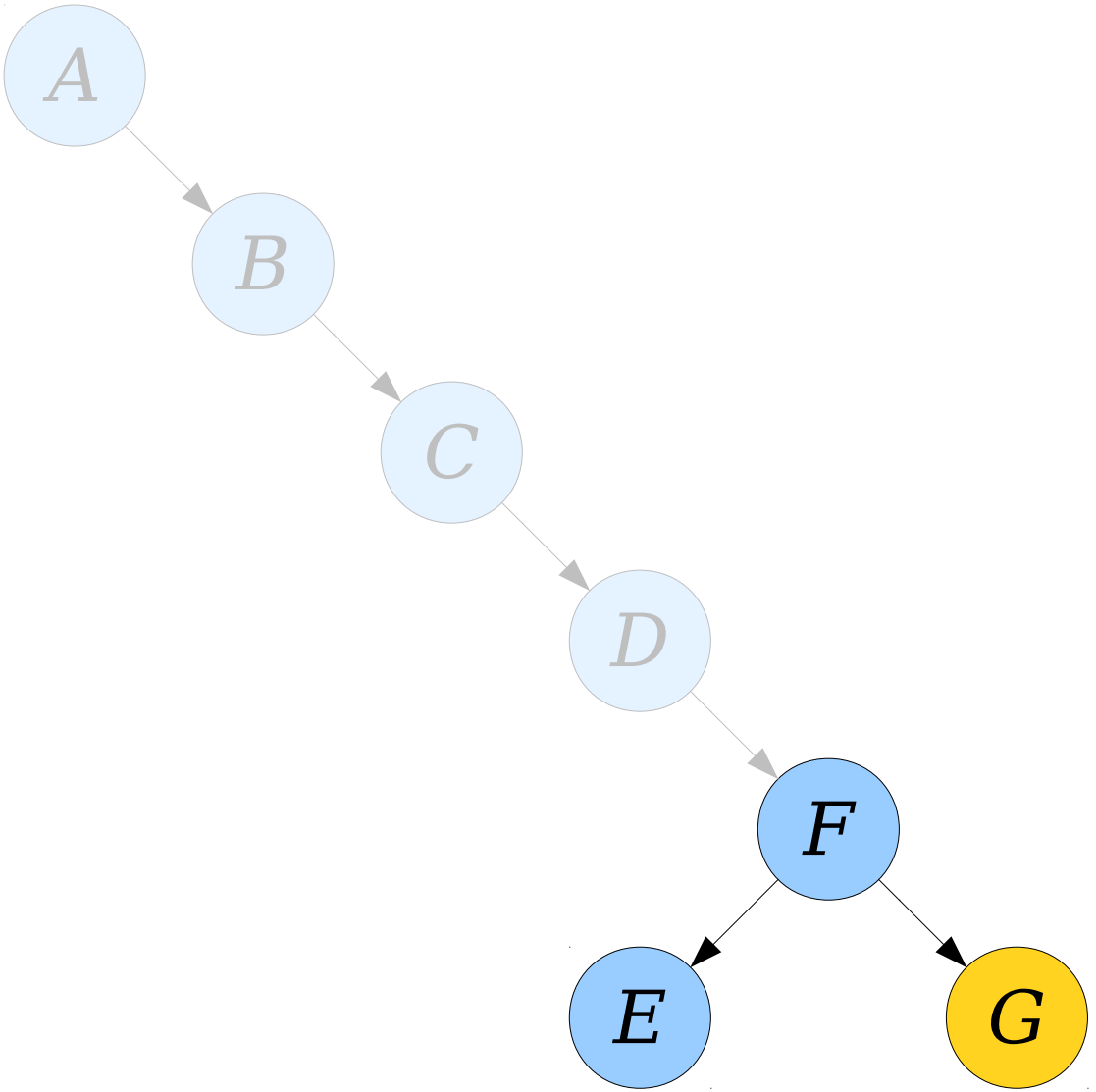


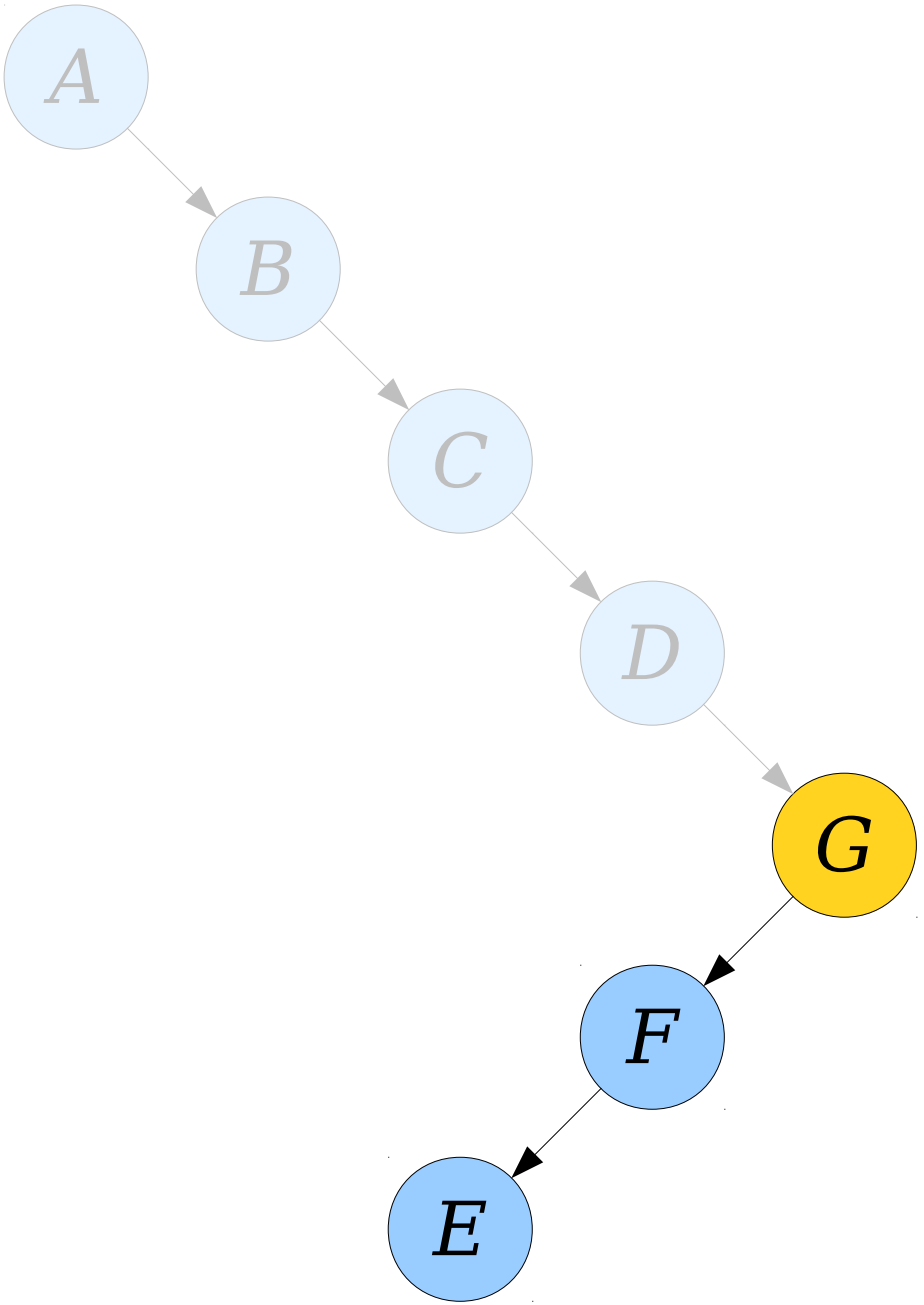
Rotate x with r
x is now the root.

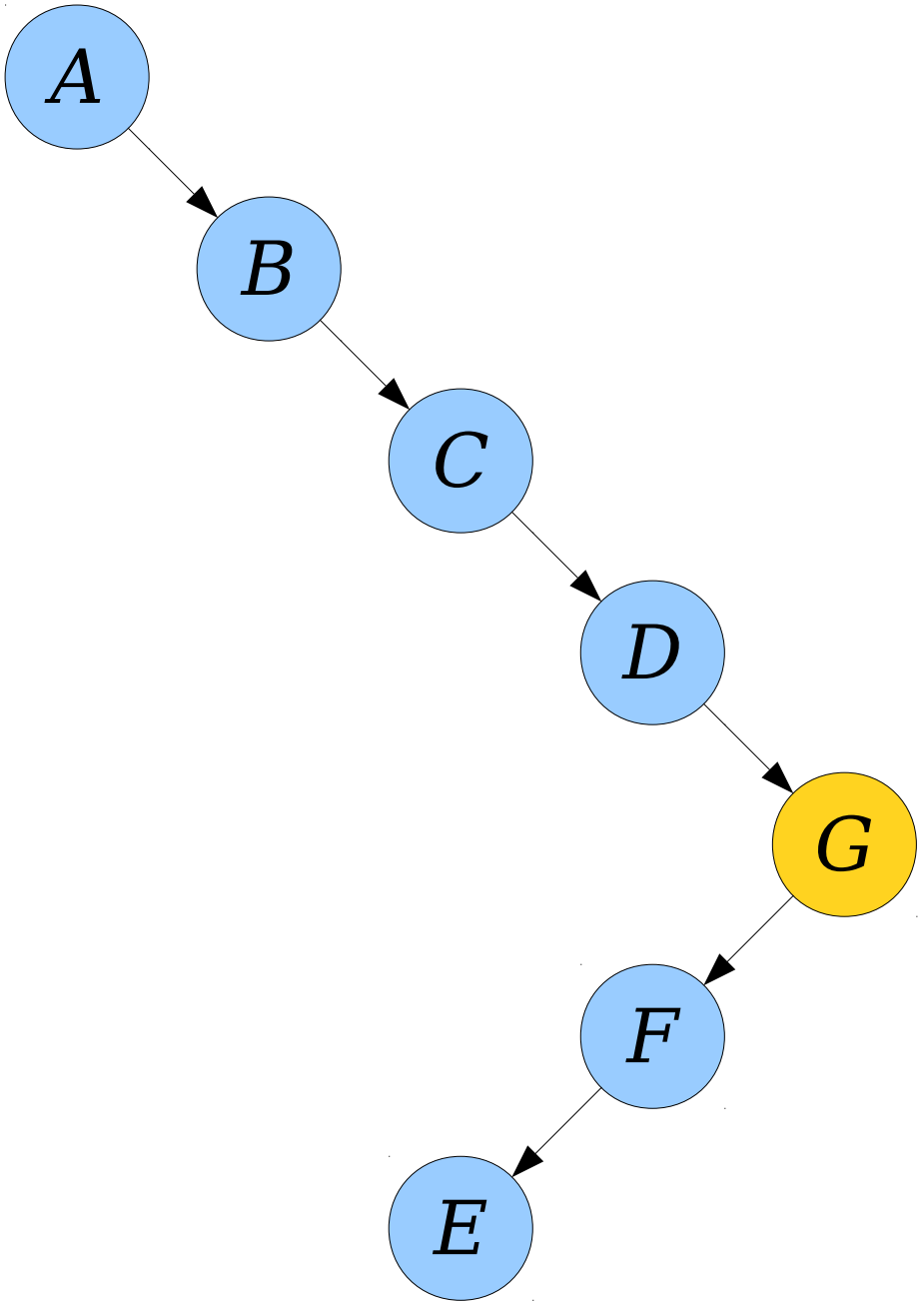

A
B
C
D
E
F
G

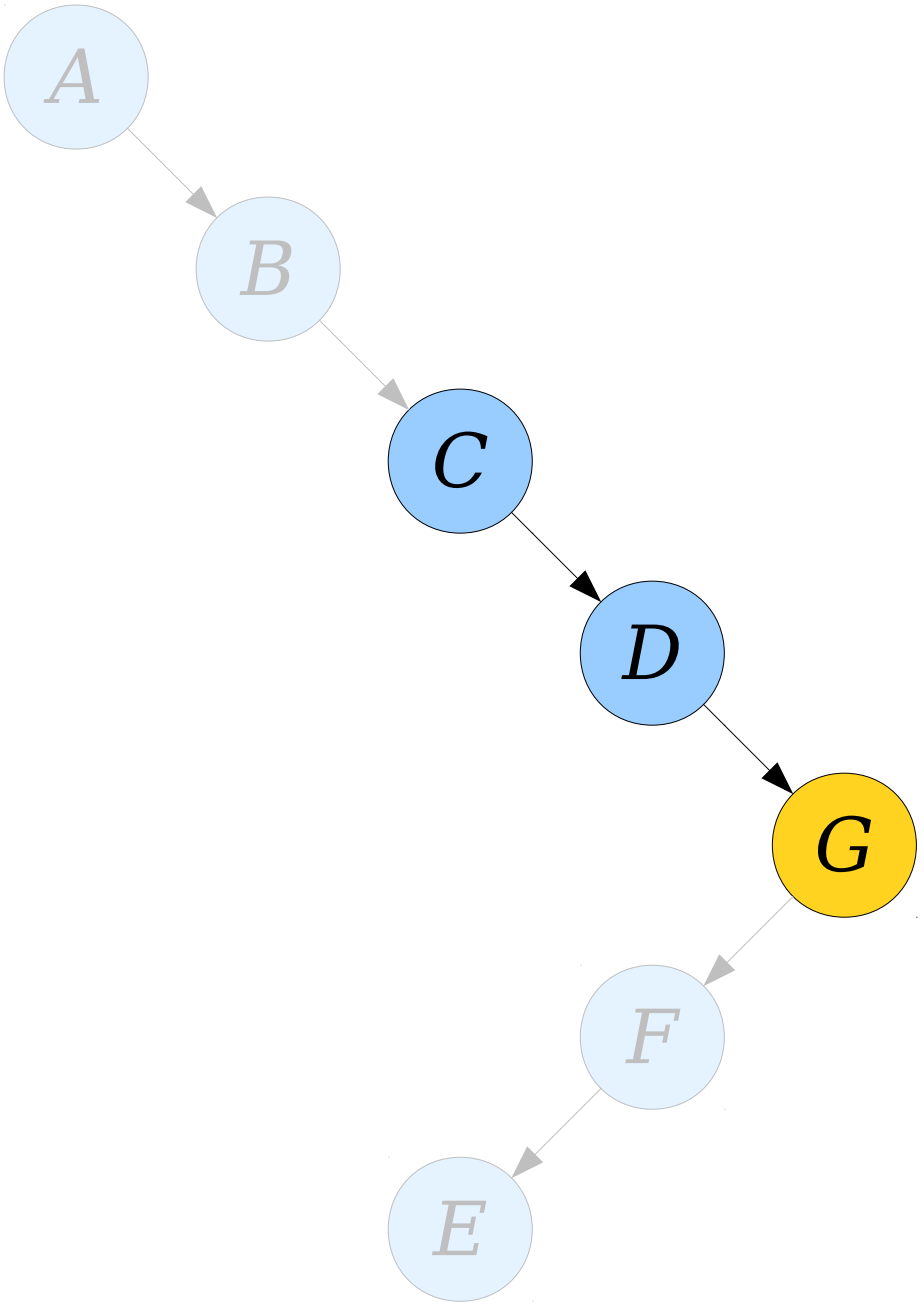

A
B
C
D
E
F
G

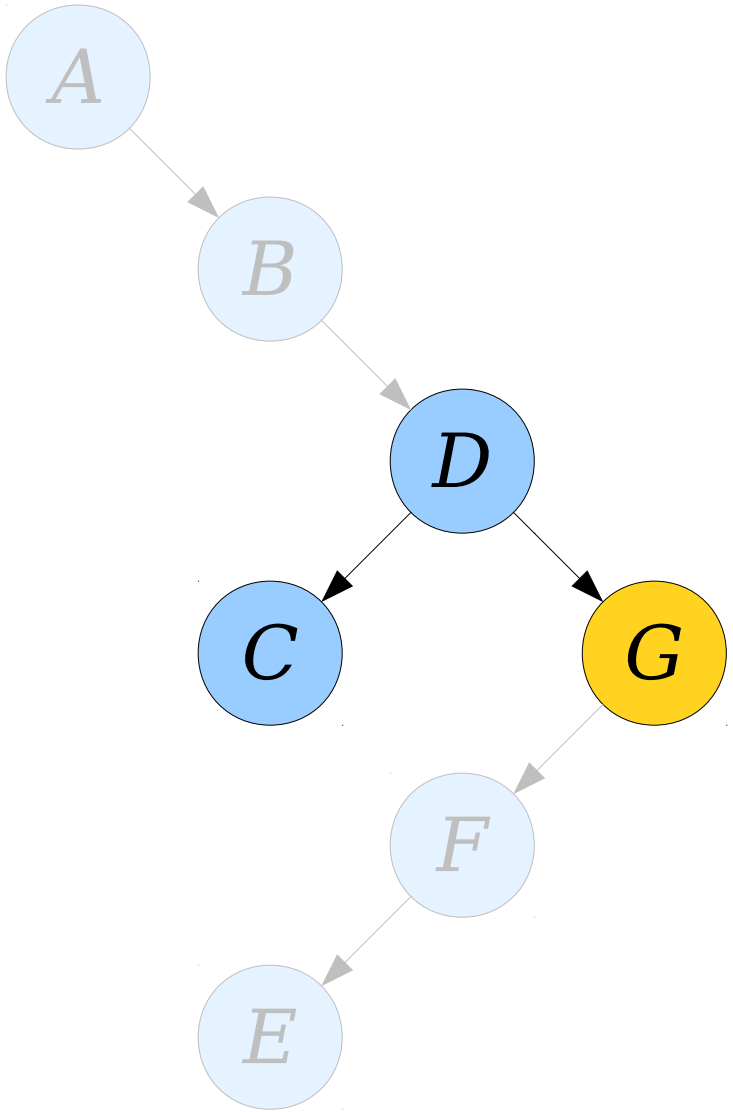

A
B
C
D
E
F
G

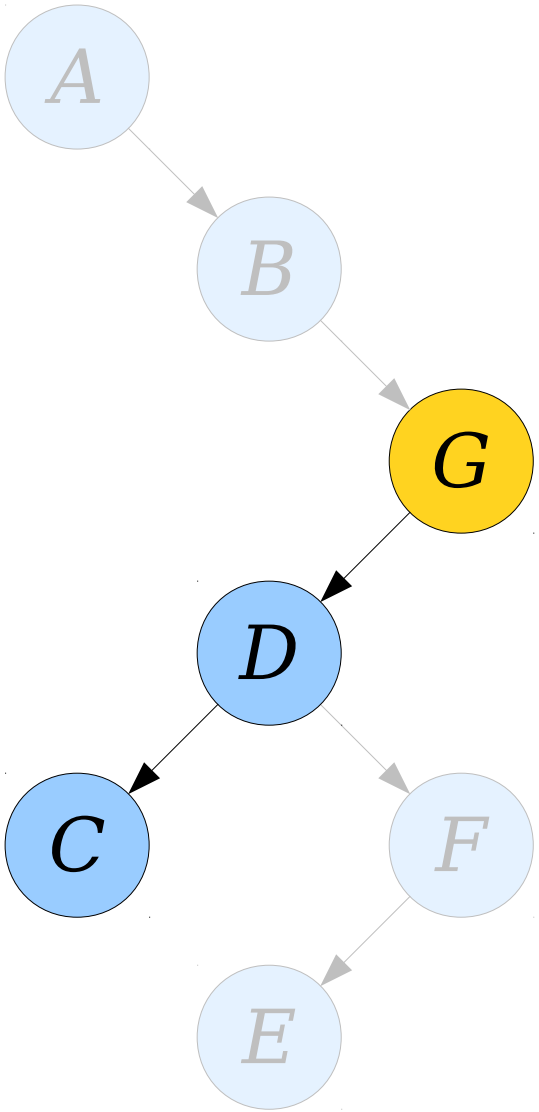

A
B
C
D
F
E
G

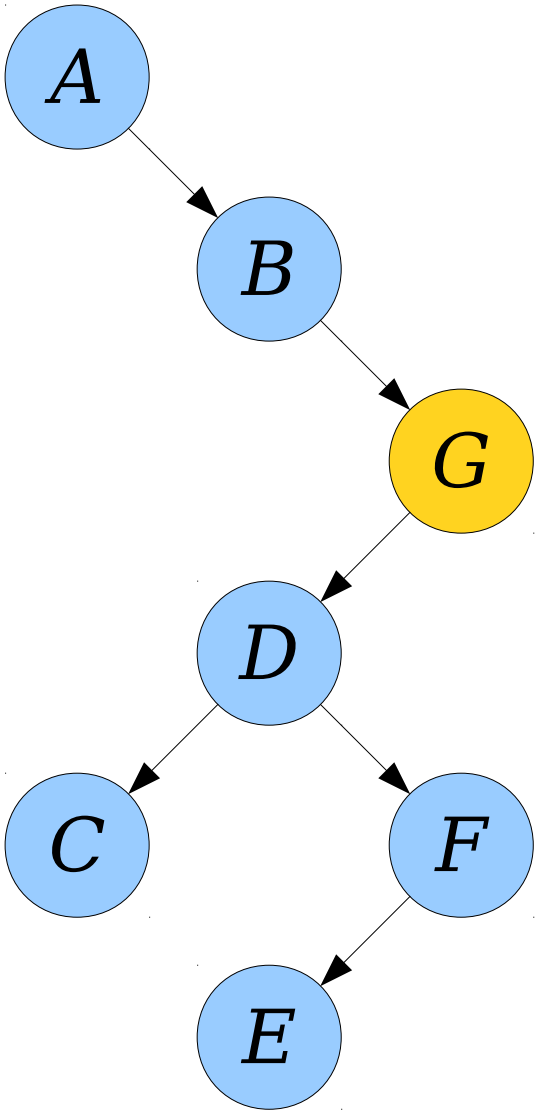

A
B
C
D
G
F
E

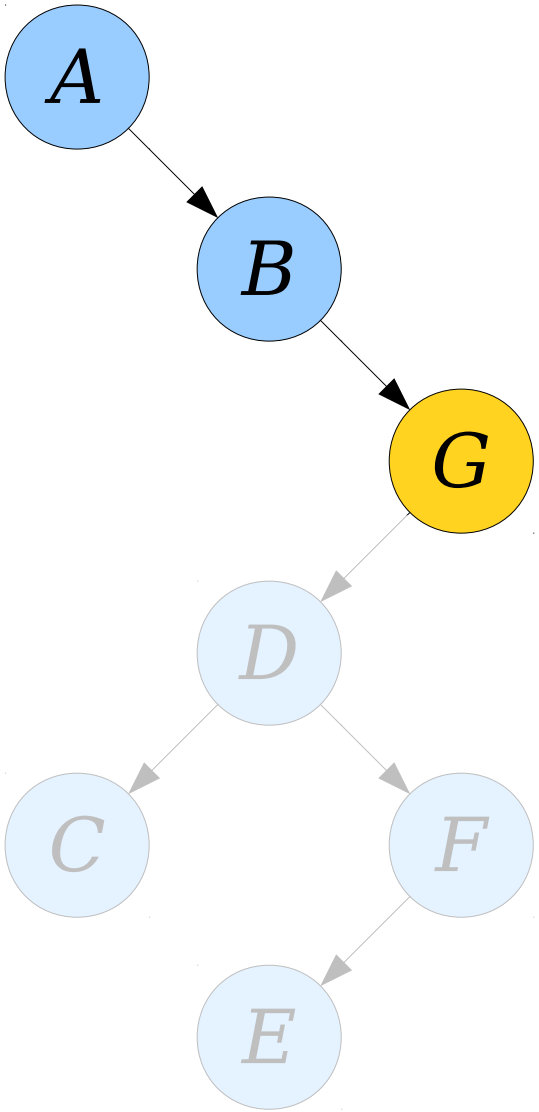

A
B
C
D
G
F
E

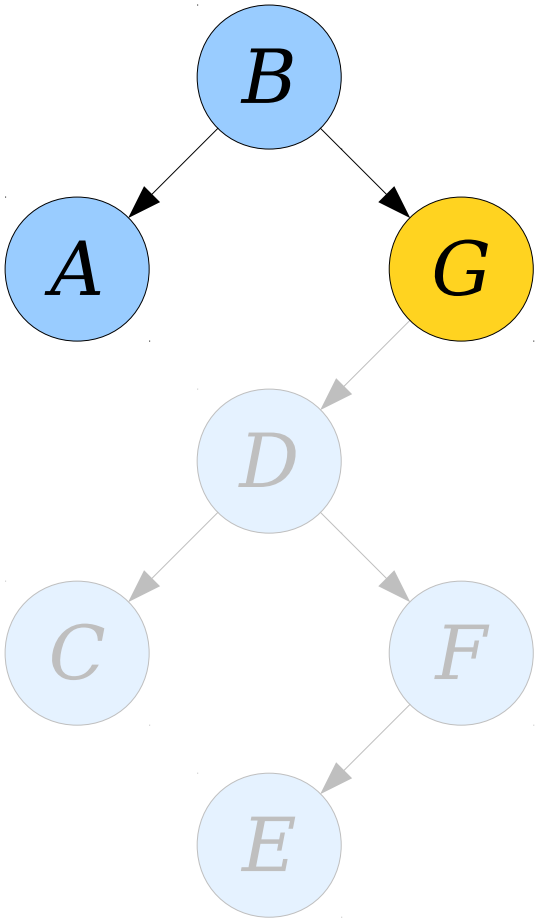

A
B
C
D
G
F
E

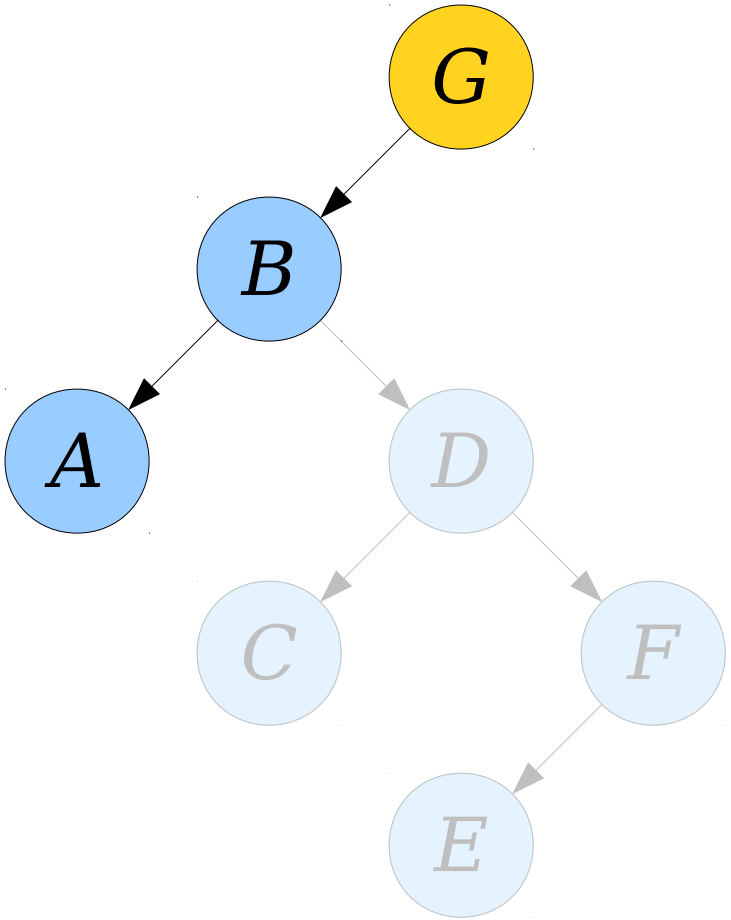

A
B
D
C
G
F
E

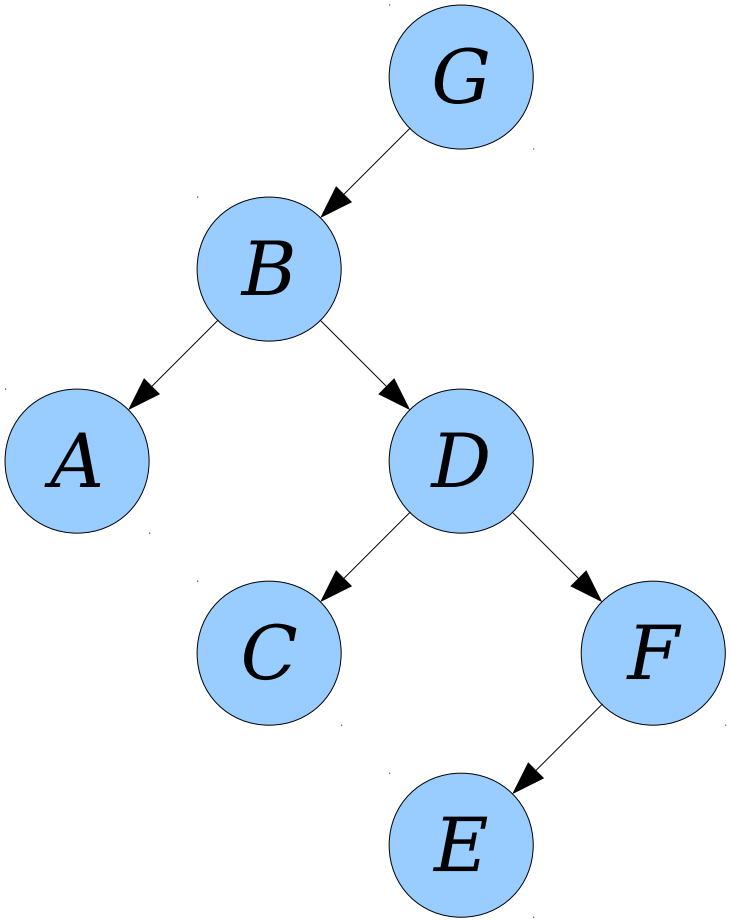

A
B
G
D
C
F
E

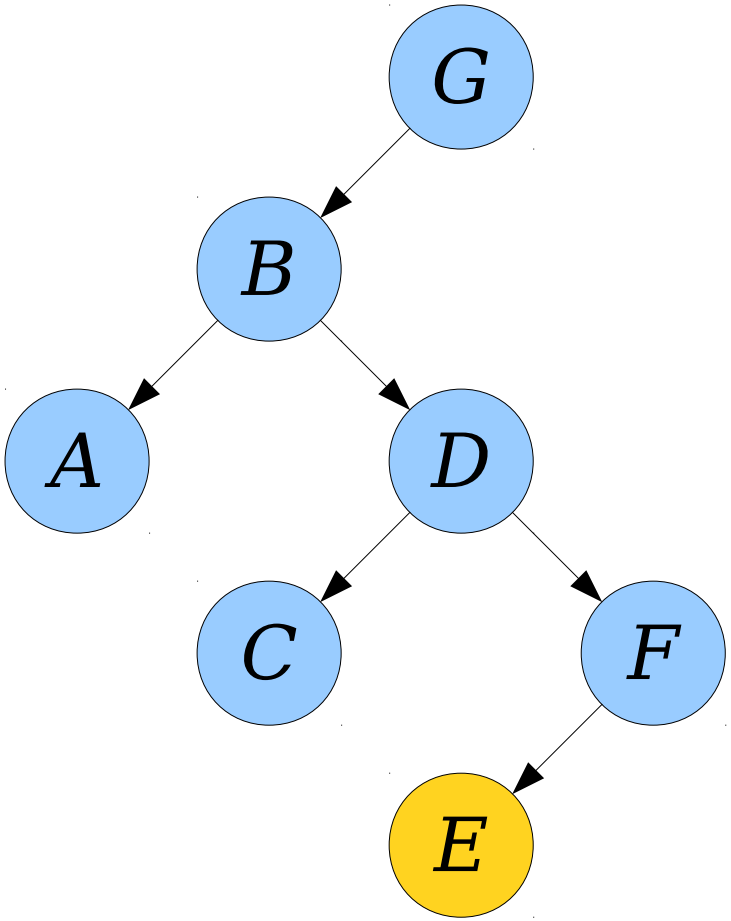

A
B
G
D
C
F
E

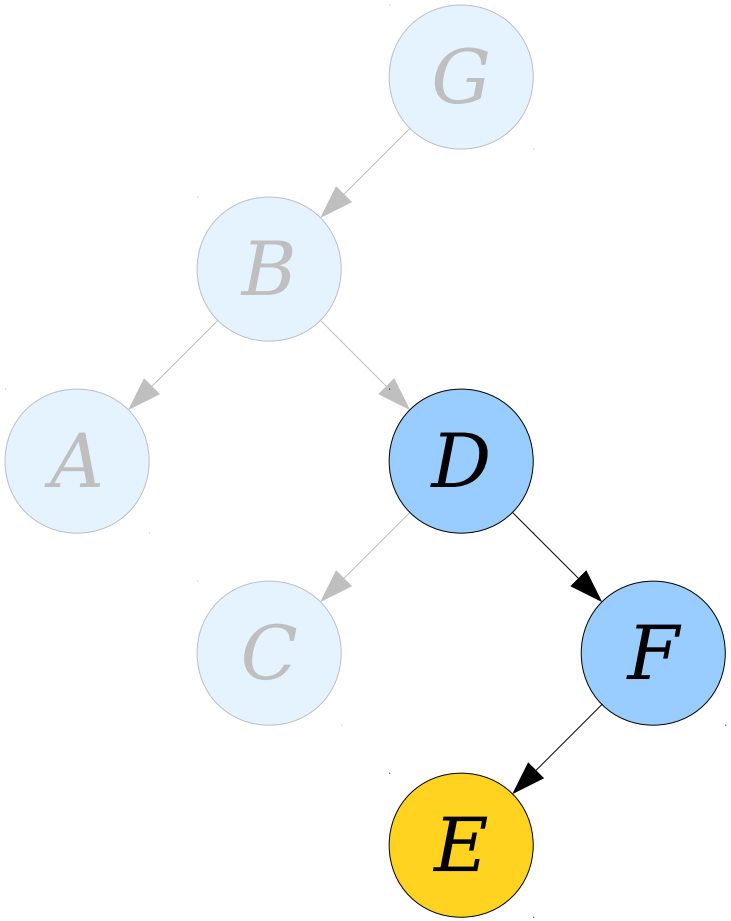

A
B
G
D
C
F
E

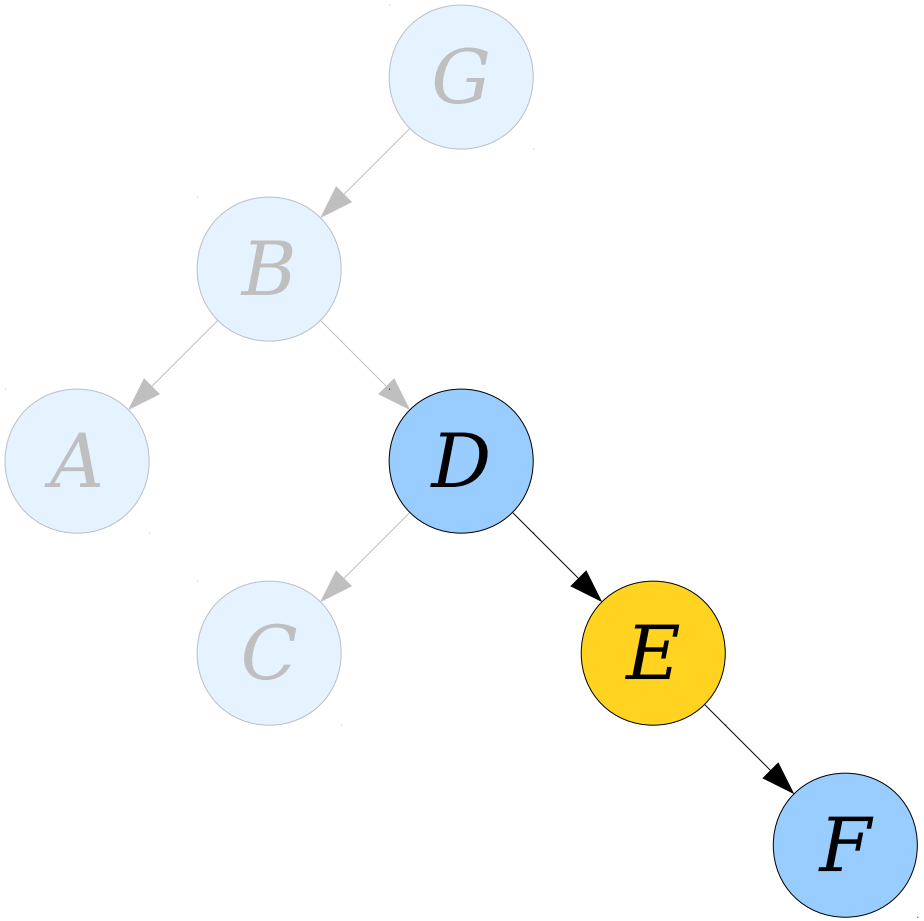

B
A
G
D
C
F
E

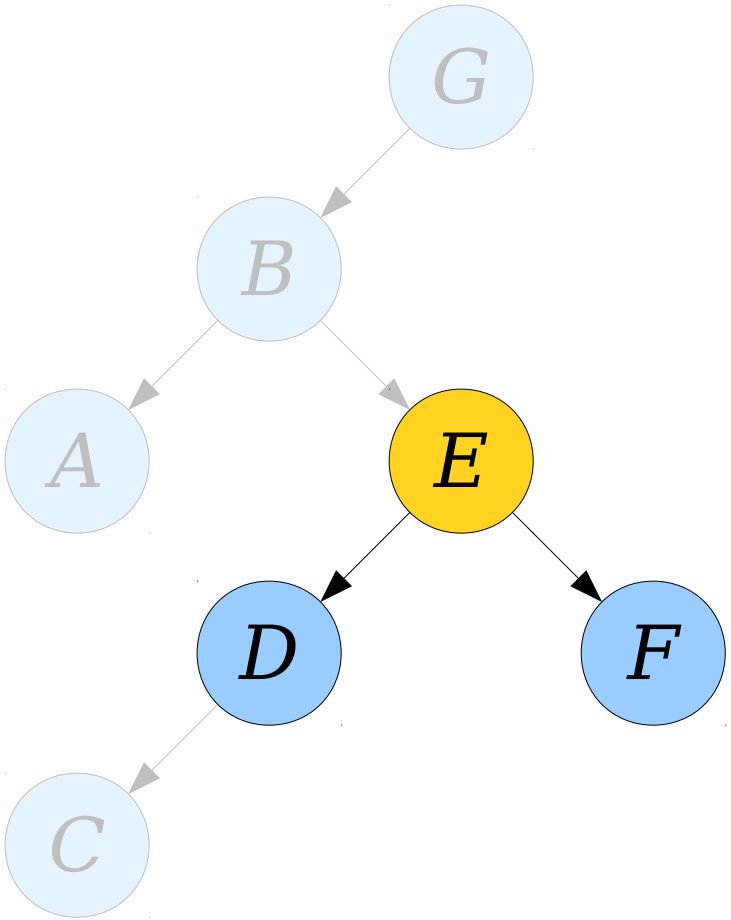

G
B
A
D
C
F
E

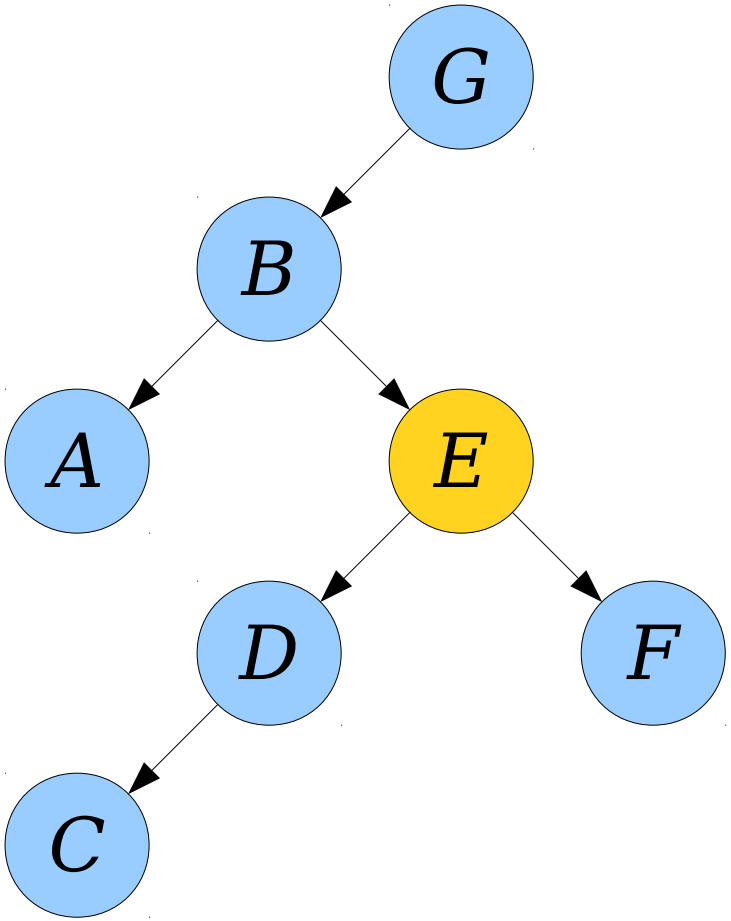

G
B
A
D
C
F
E

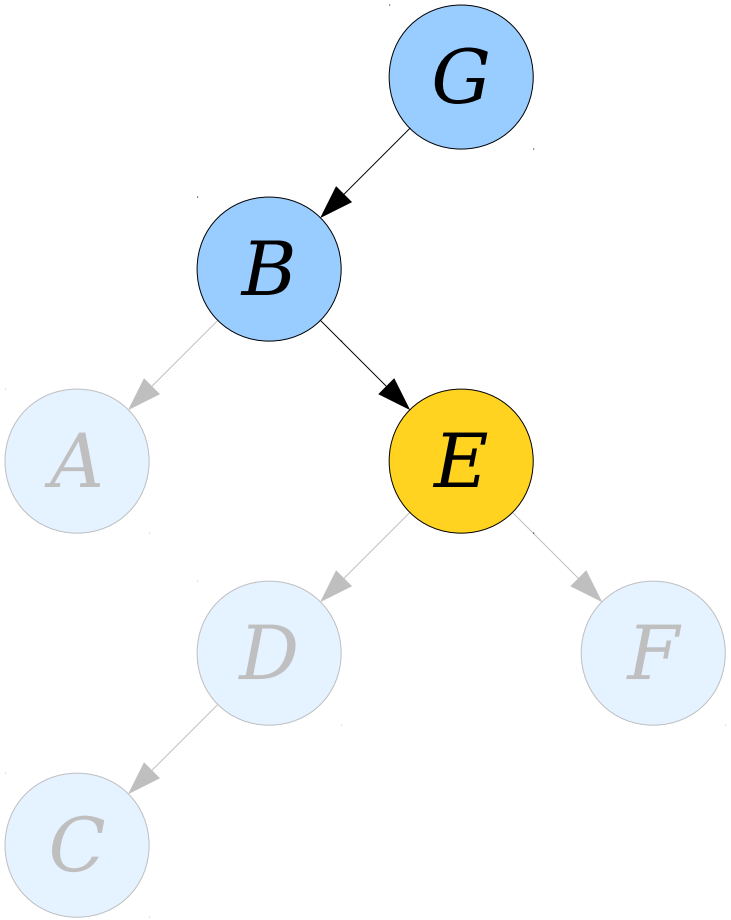

G
B
A
D
C
F
E

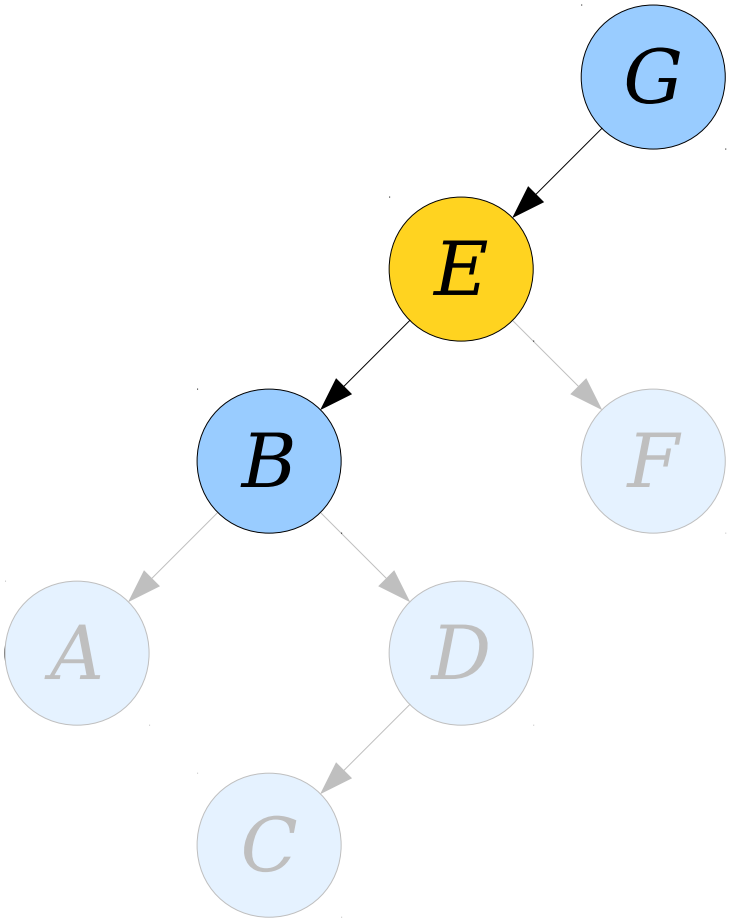

G
B
A
D
C
F
E

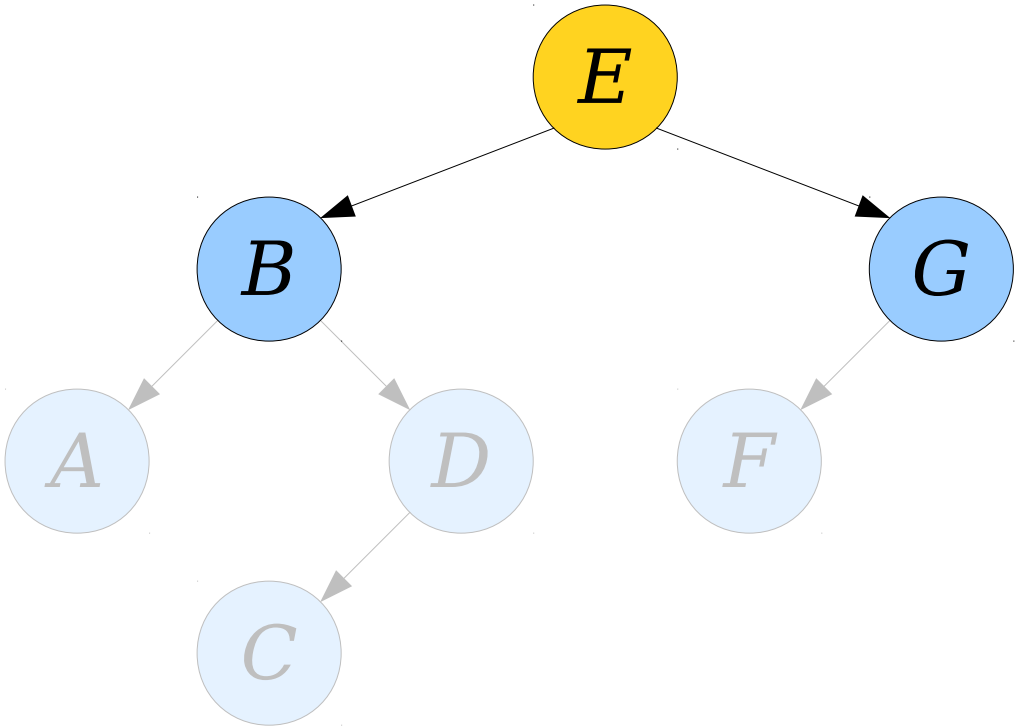

G
B
A
D
C
E
F

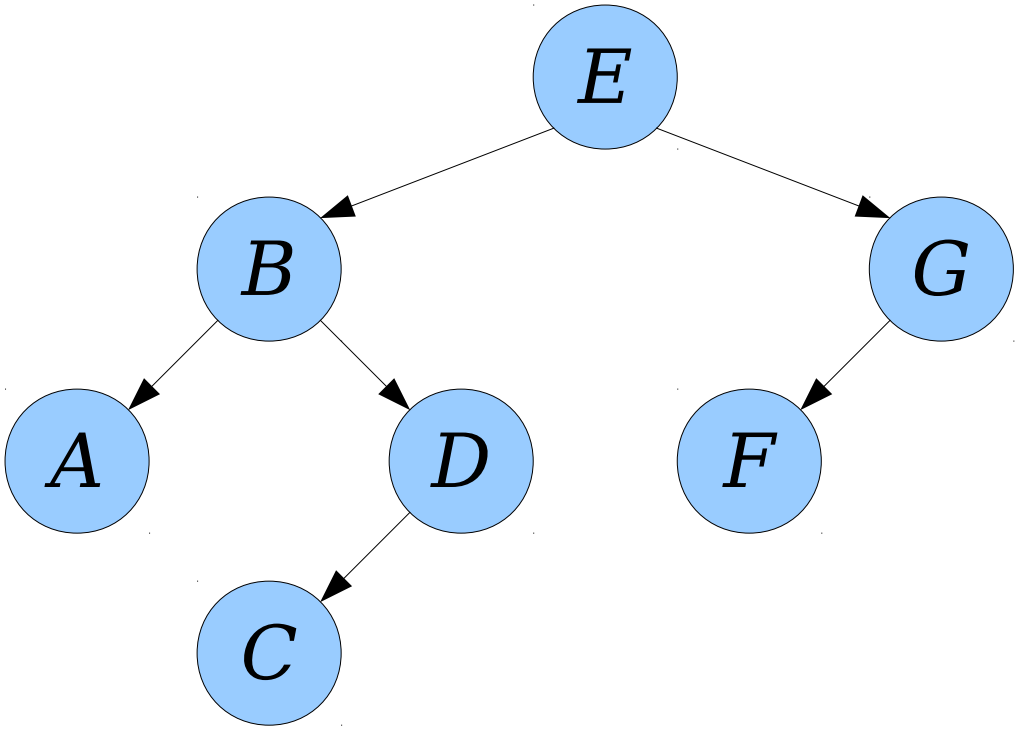

G
B
A
E
D
F
C


G
B
A
E
D
F
C


G
B
A
E
D
F
C


G
E
B
F
A
D
C


E
B
G
A
D
F
C

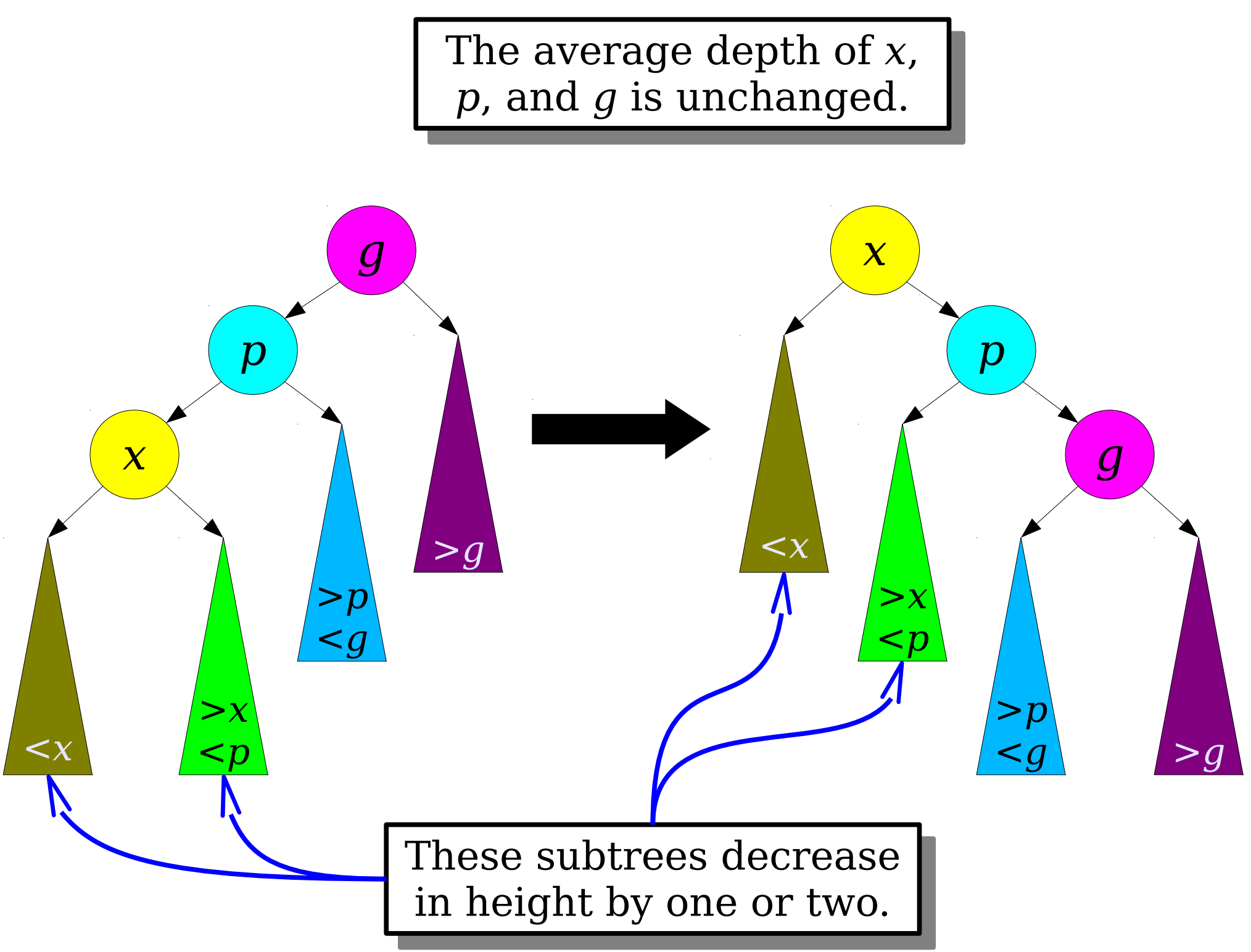

E
B
G
A
D
F
C

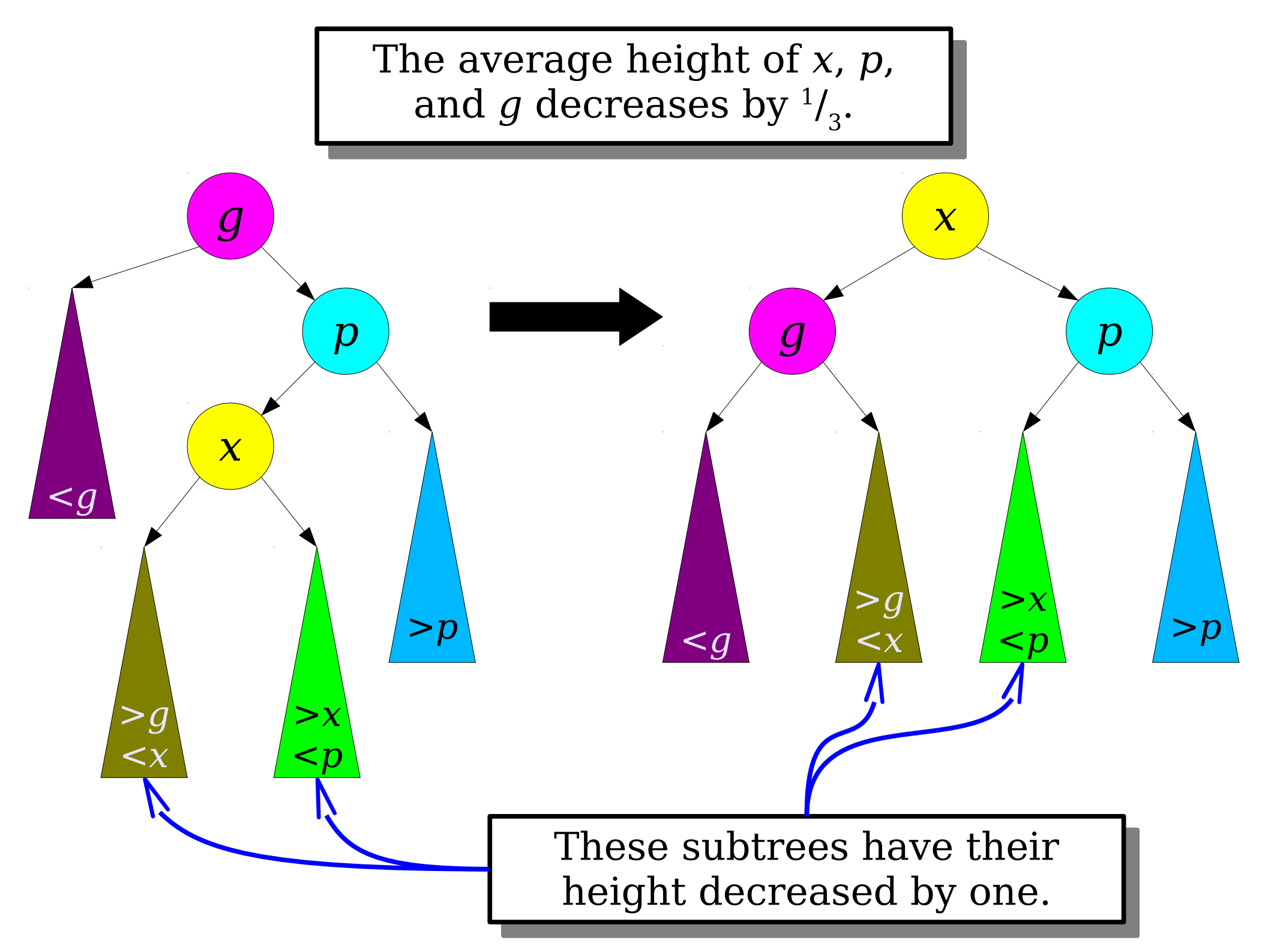
Splaying, Empirically

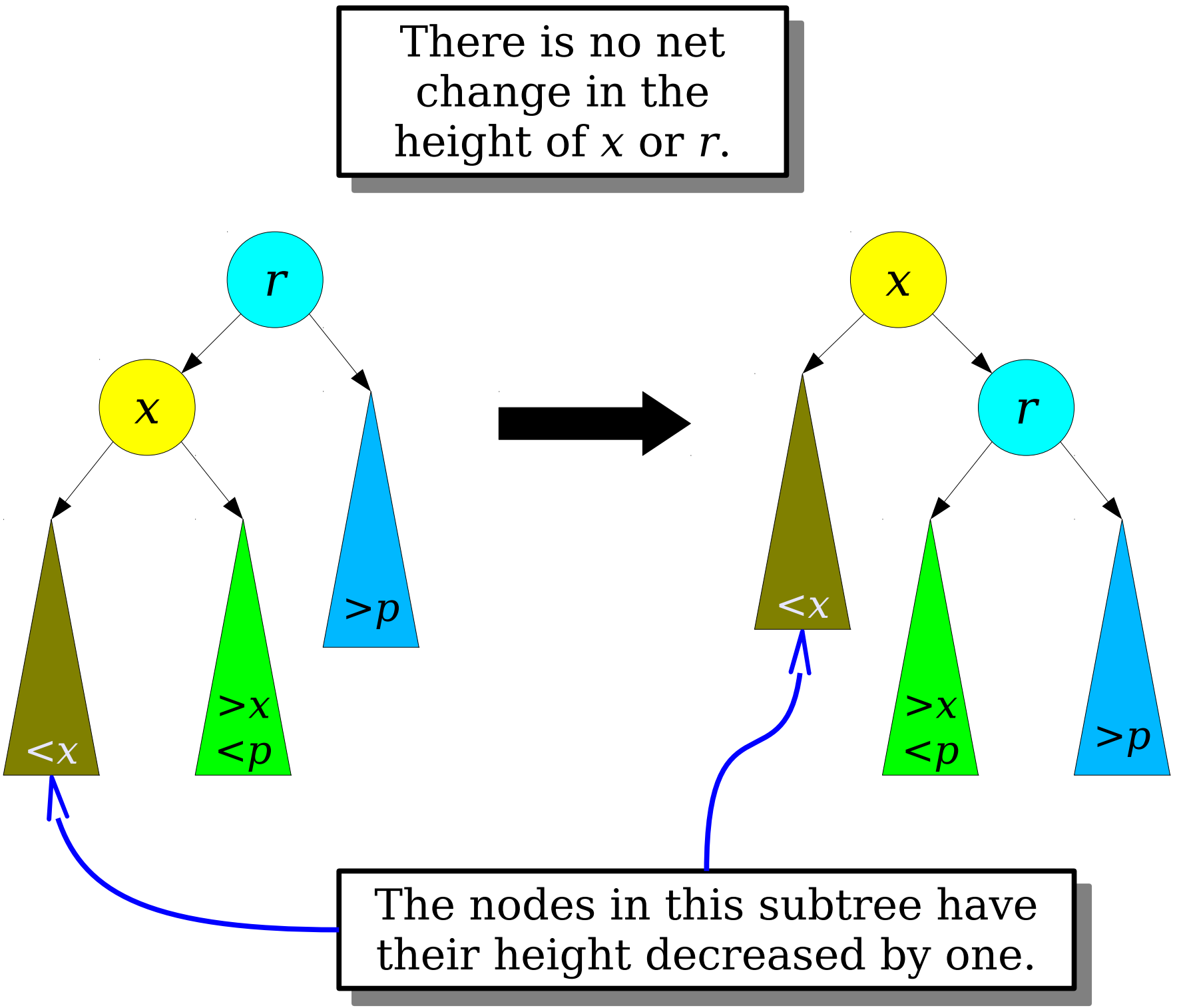
- After a few splays, we went from a totally degenerate tree to a reasonably-balanced tree.
- Splaying nodes that are deep in the tree tends to correct the tree shape.
- Why is this?
- Is this a coincidence?

Why Splaying Works

- ***Claim:*** After doing a splay at x, the average depth of any nodes on the access path to x is halved.
- Intuitively, splaying x benefits nodes near x, not just x itself.
- This “altruism” will ensure that splays are efficient.


The average depth of x, p, and g is unchanged.
g
p
x
<x
>x
<p
>p
<g
>g
x
p
g
<x
>x
<p
>p
<g
>g
These subtrees decrease in height by one or two.

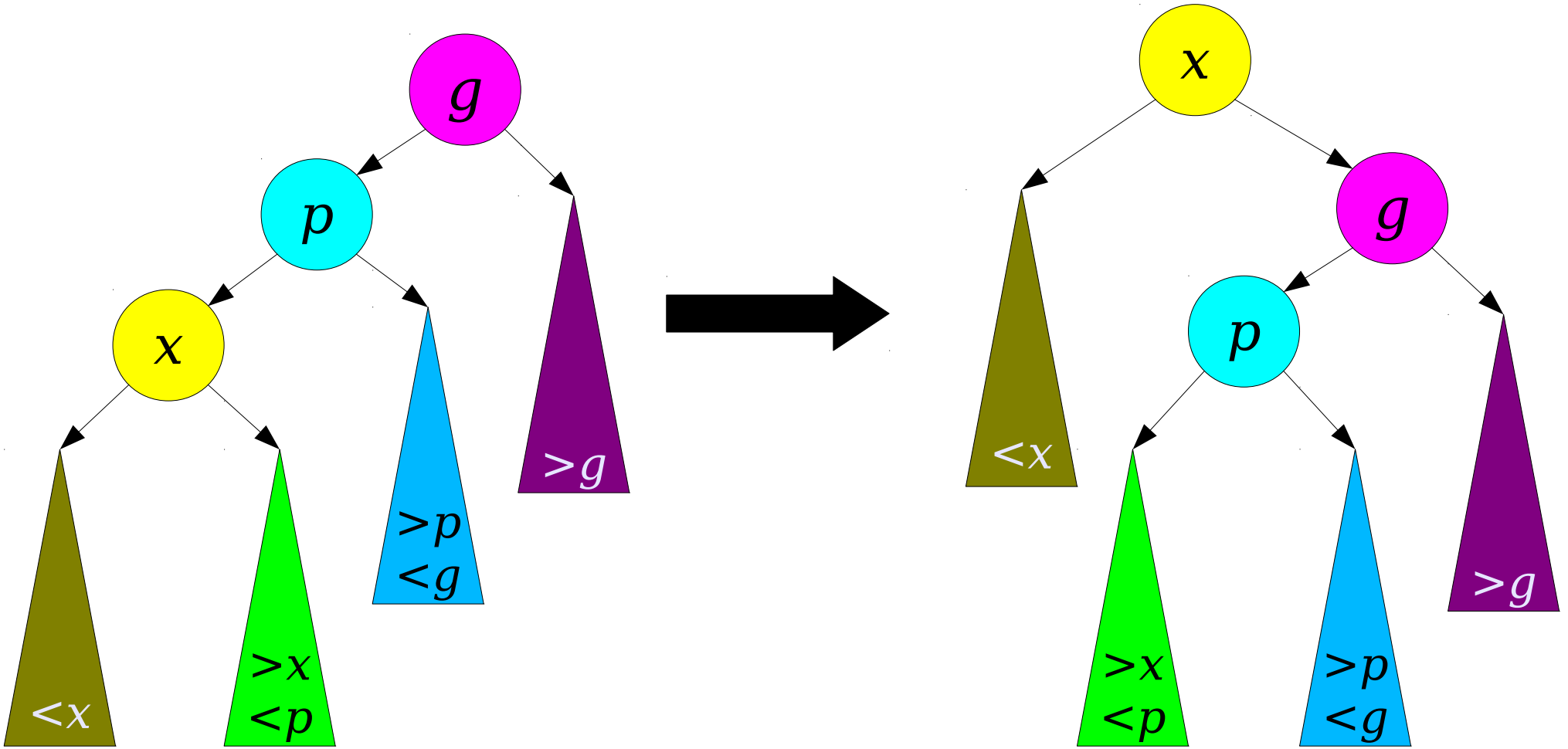

The average height of x, p, and g decreases by $^1/_3$.
g
p
x
$<g$
$>p$
$>g$ $<x$
$>x$ $<p$
x
g
p
$<g$
$>g$ $<x$
$>x$ $<p$
$>p$
These subtrees have their height decreased by one.


There is no net change in the height of x or r.
r
x
<x
>x <p
>p
x
r
<x
>x <p
>p
The nodes in this subtree have their height decreased by one.

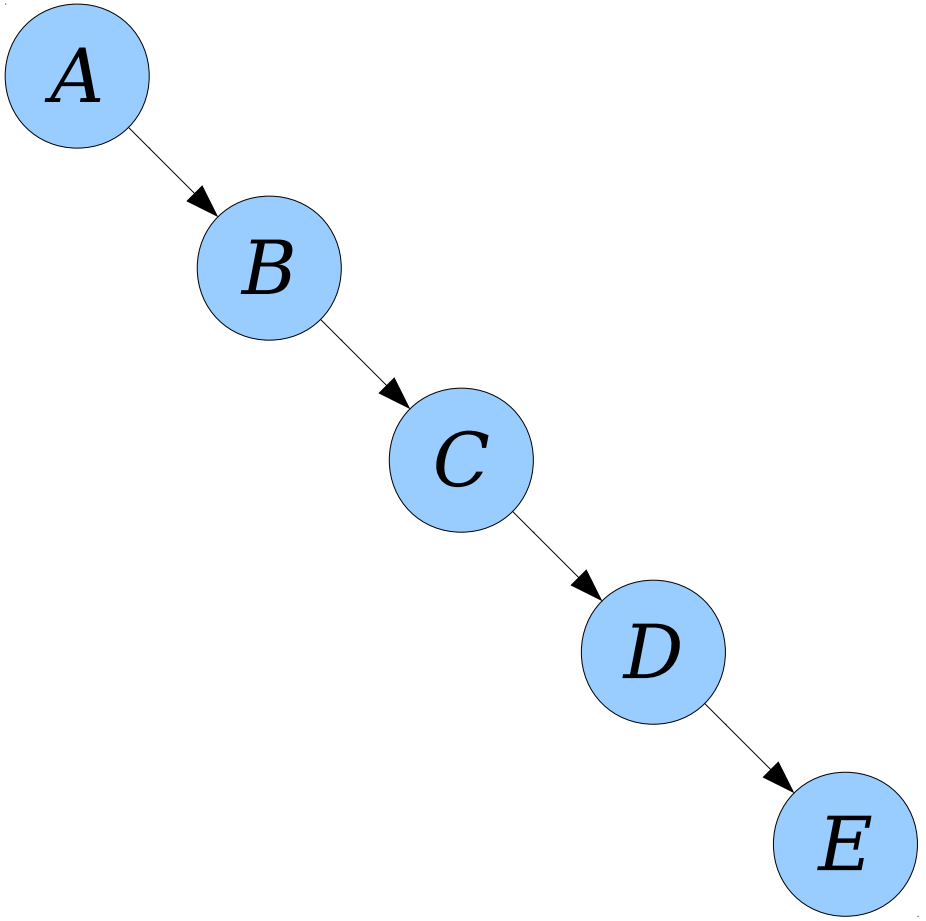
An Intuition for Splaying

- Each rotation done only slightly penalizes each other part of the tree (say, adding +1 or +2 depth).
- Each splay rapidly cuts down the height of each node on the access path.
- Slow growth in height, combined with rapid drop in height, is a hallmark of amortized efficiency.
- ***Claim:*** The original “rotate-to-root” idea from before doesn't do this, which partially explains why it's not a good strategy.

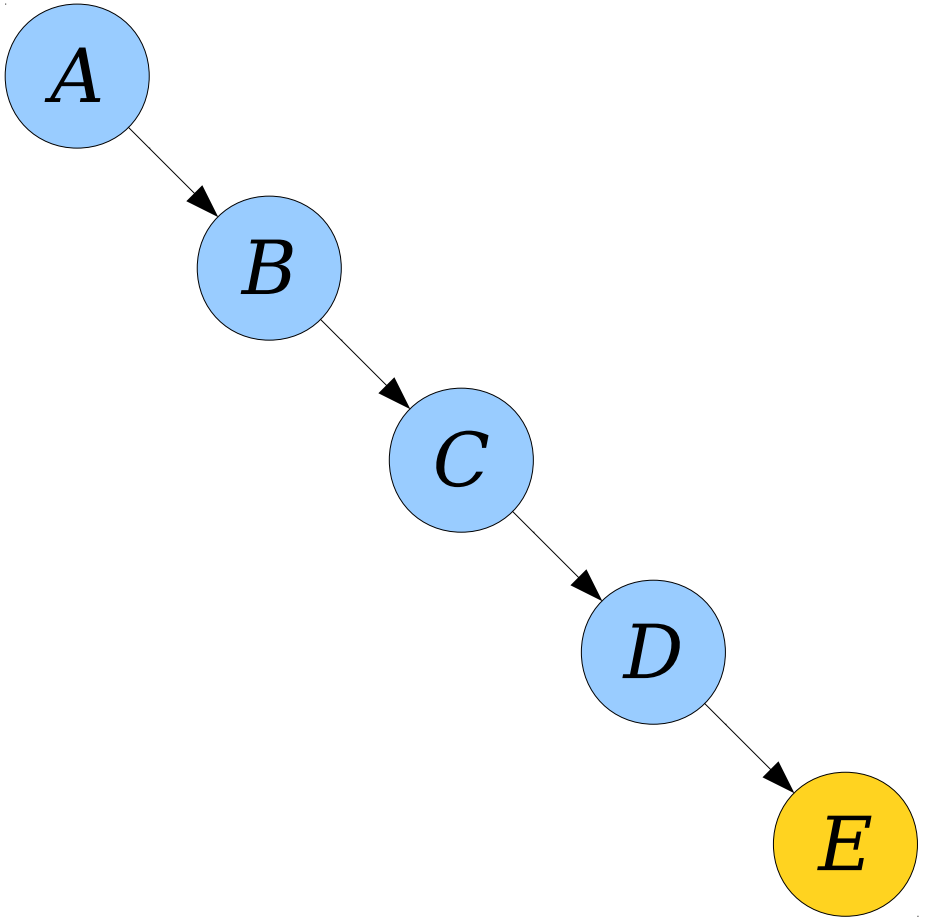
Rotate-to-Root



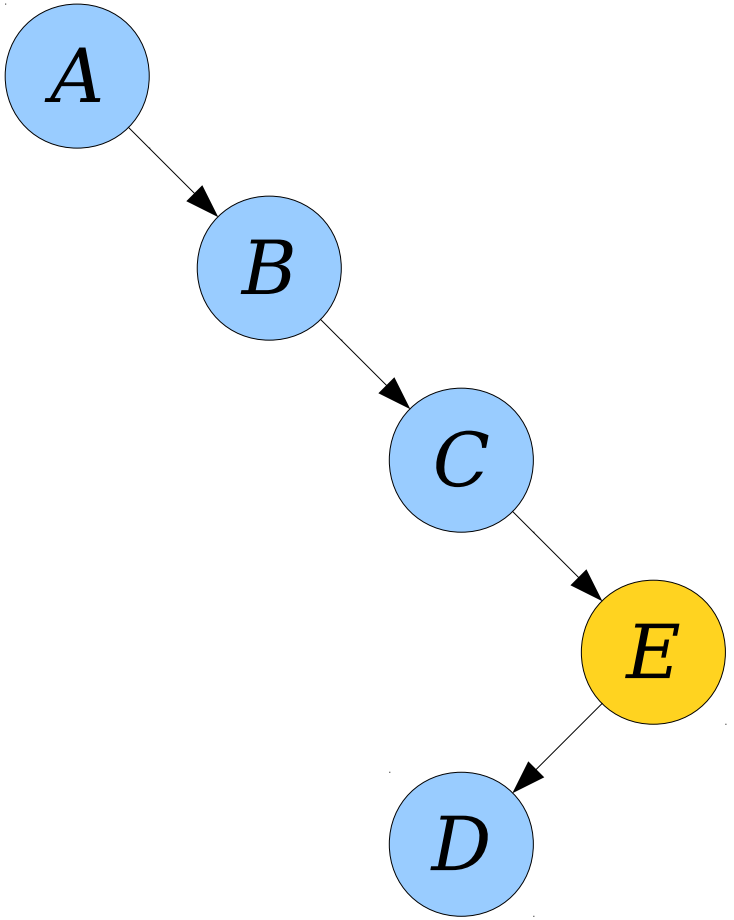
Why Rotate-to-Root Fails



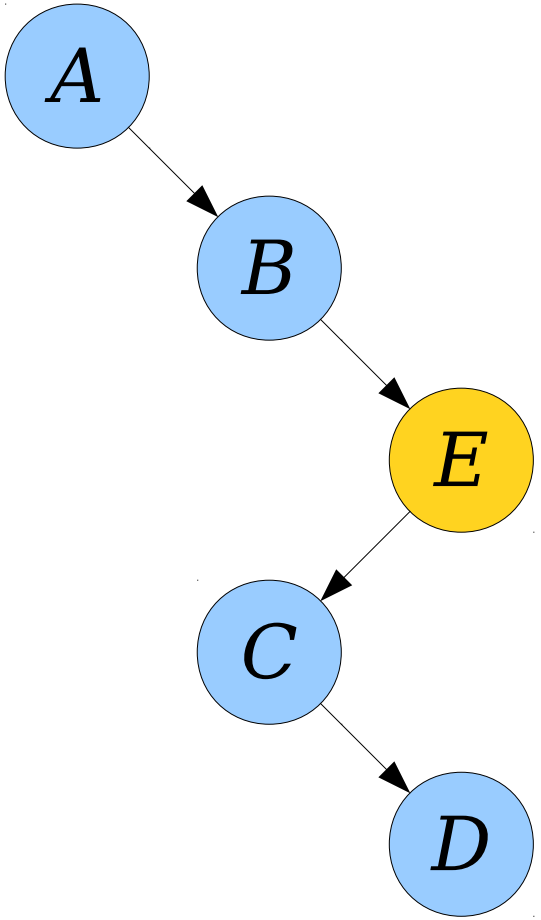
Why Rotate-to-Root Fails



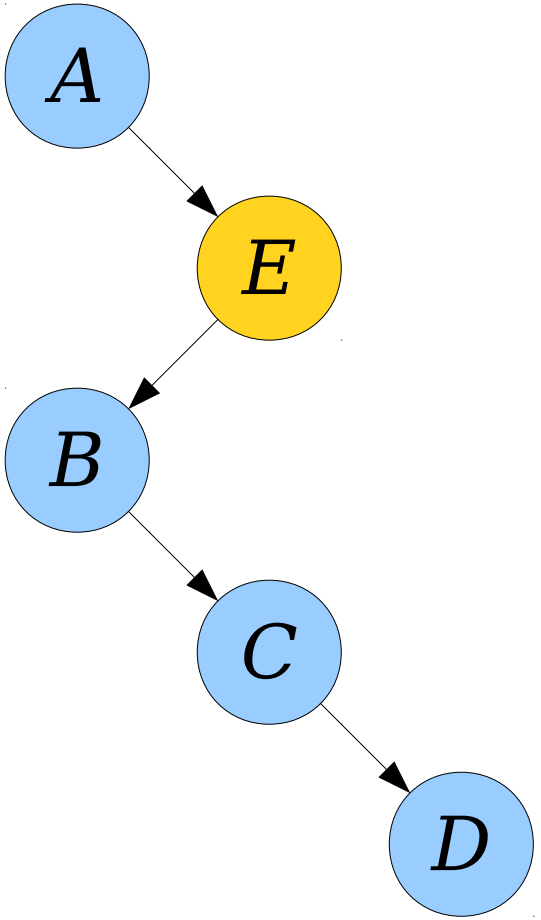
Why Rotate-to-Root Fails



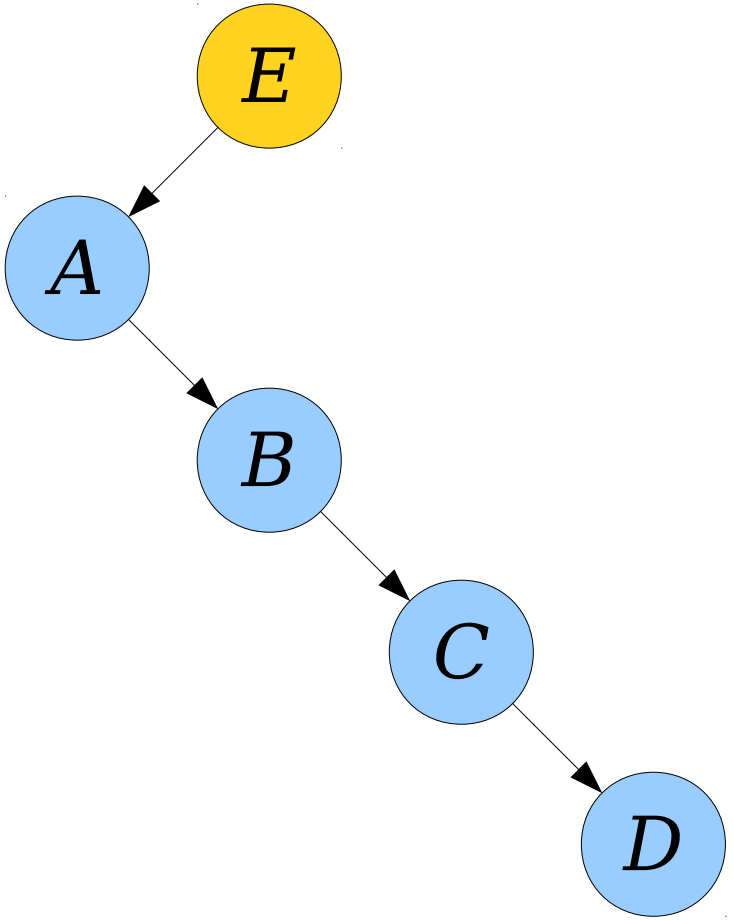
Why Rotate-to-Root Fails



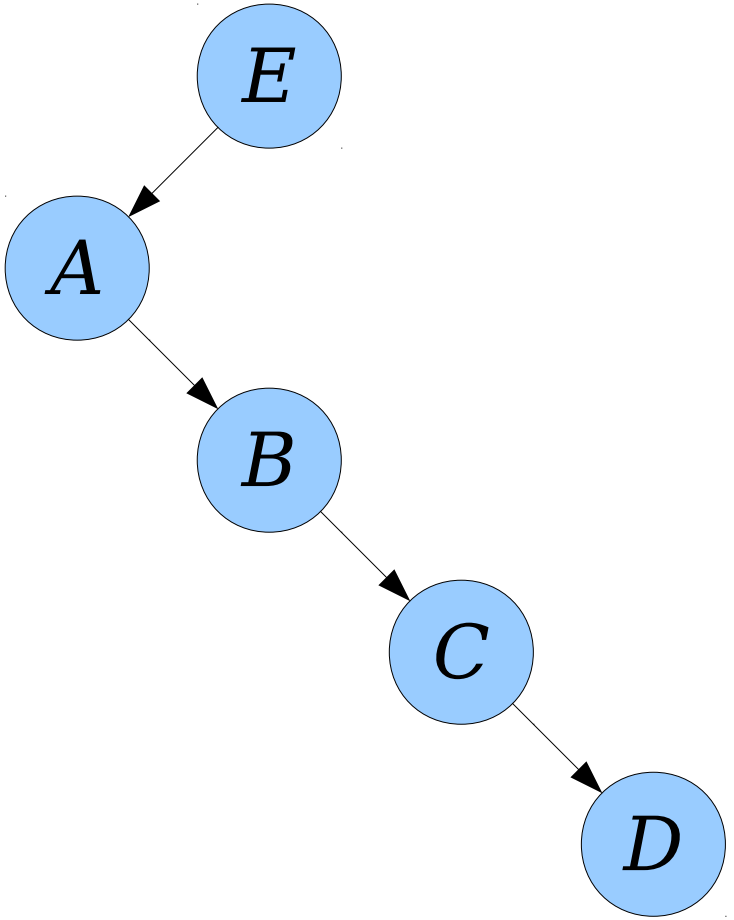
Why Rotate-to-Root Fails



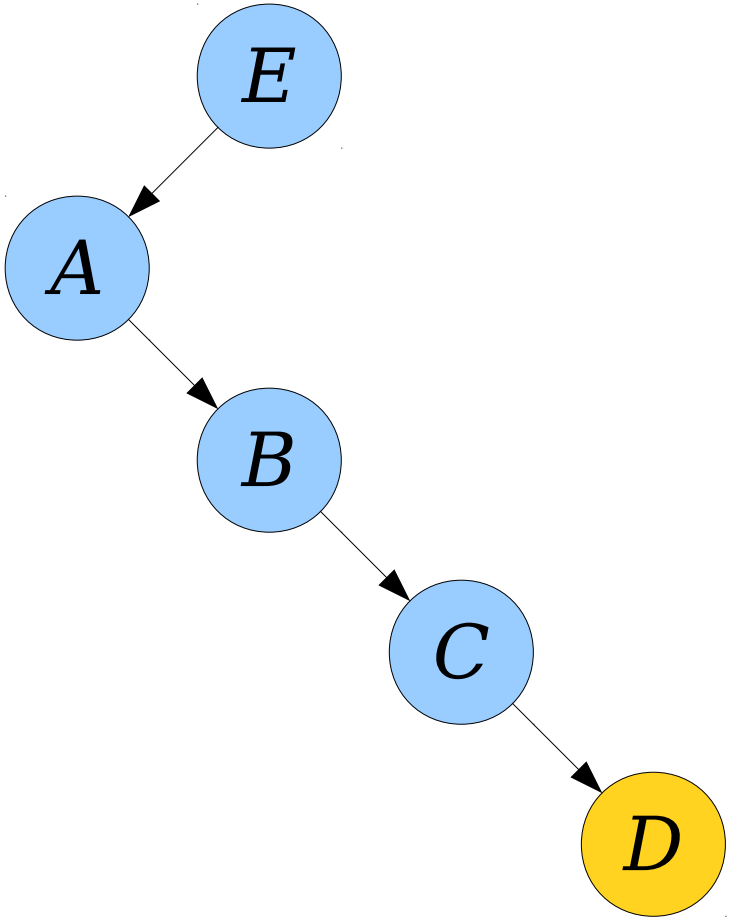
Why Rotate-to-Root Fails



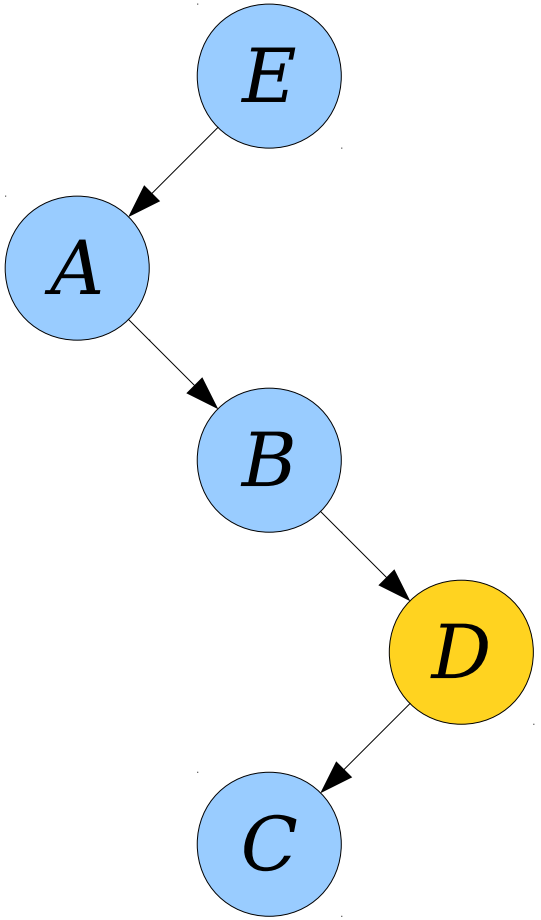
Why Rotate-to-Root Fails



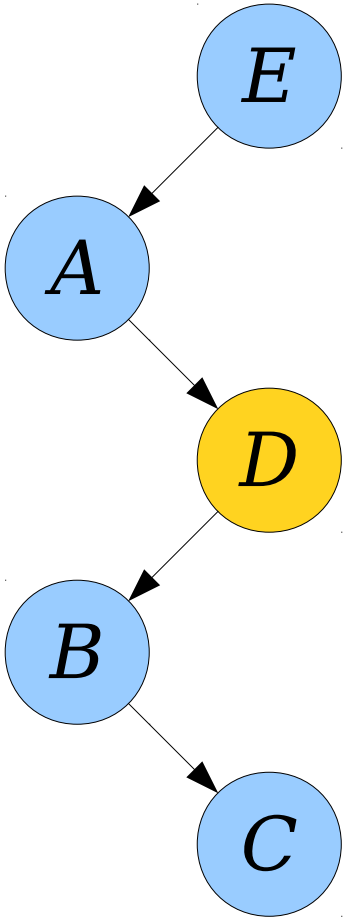
Why Rotate-to-Root Fails



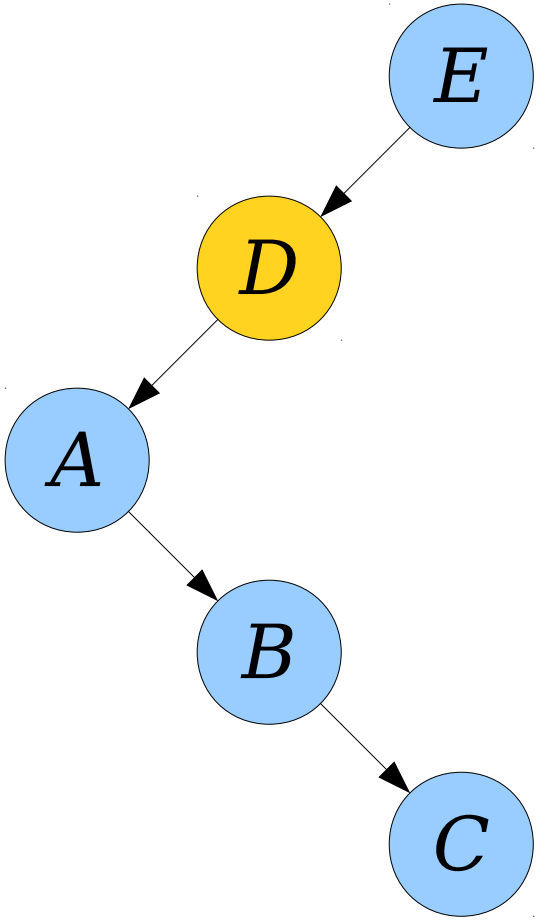
Why Rotate-to-Root Fails



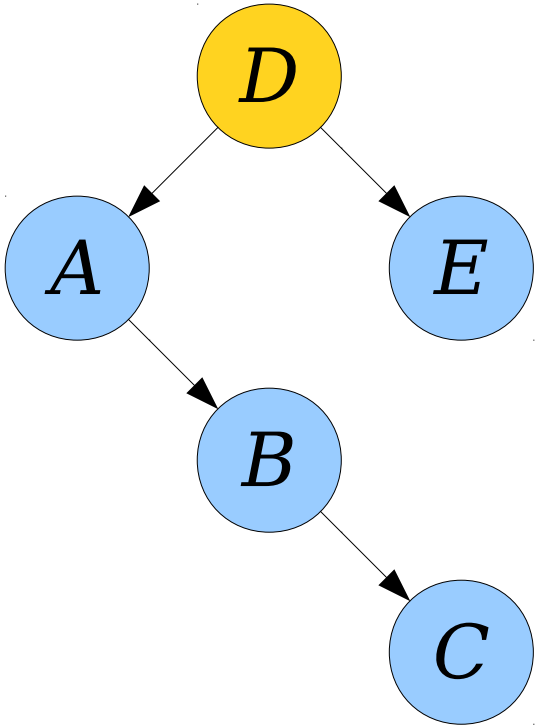
Why Rotate-to-Root Fails



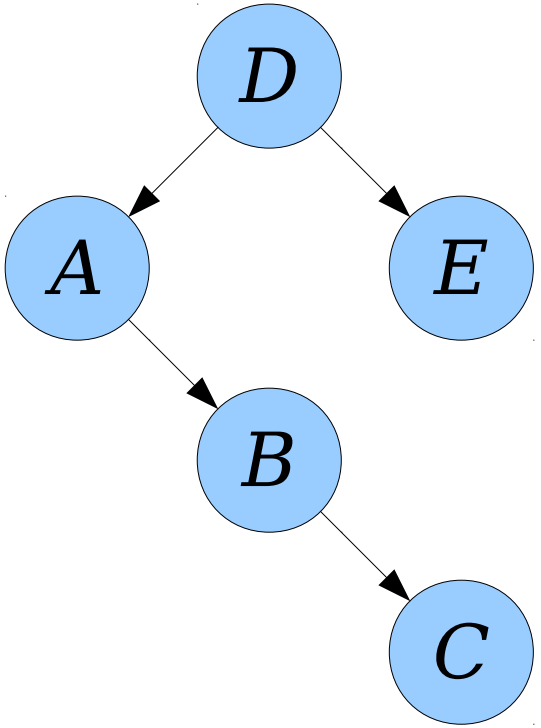
Why Rotate-to-Root Fails



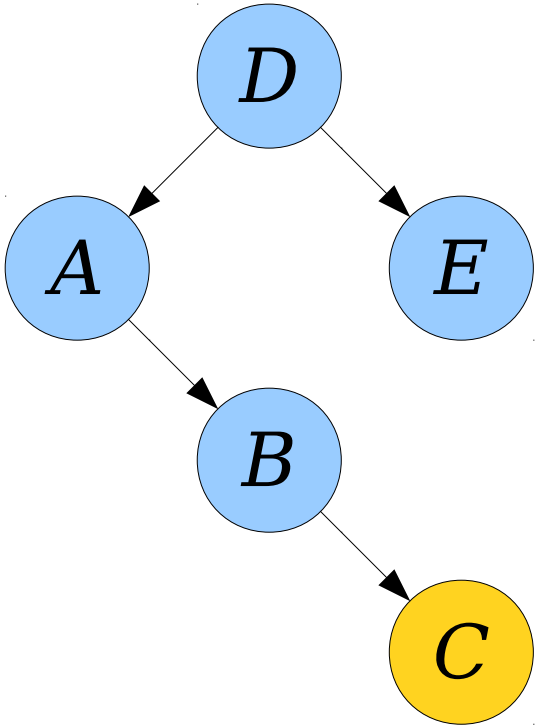
Why Rotate-to-Root Fails



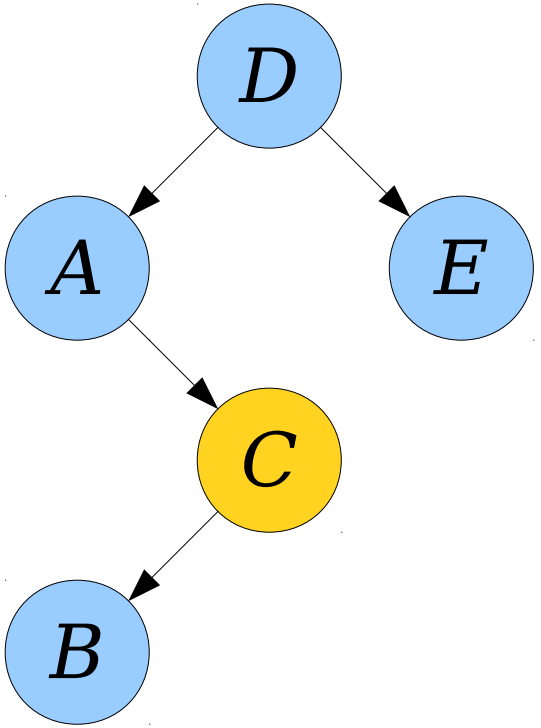
Why Rotate-to-Root Fails



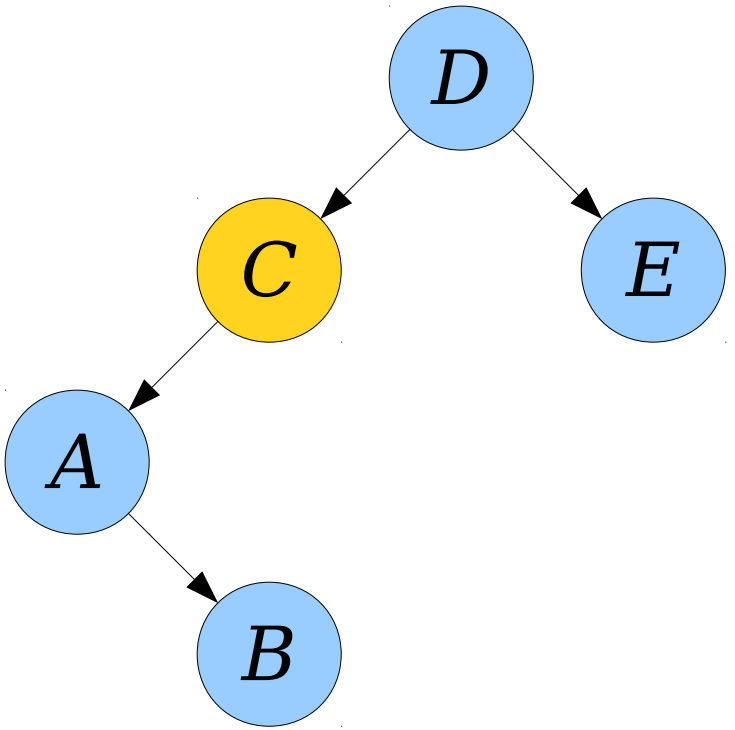
Why Rotate-to-Root Fails



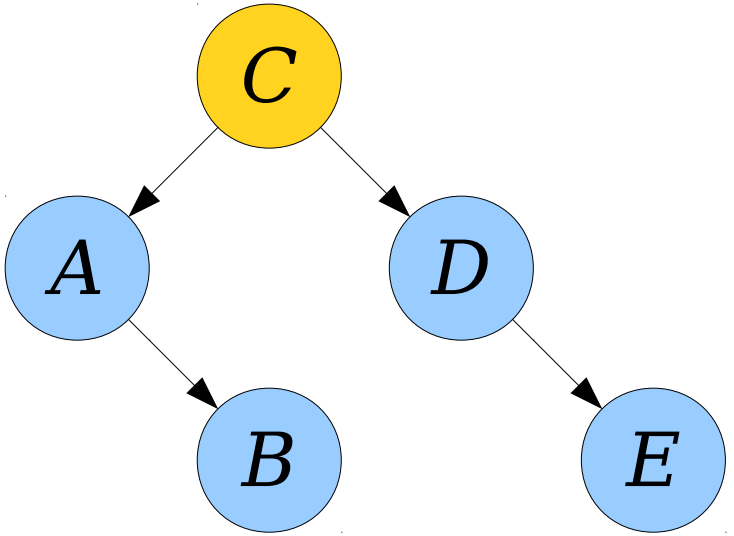
Why Rotate-to-Root Fails



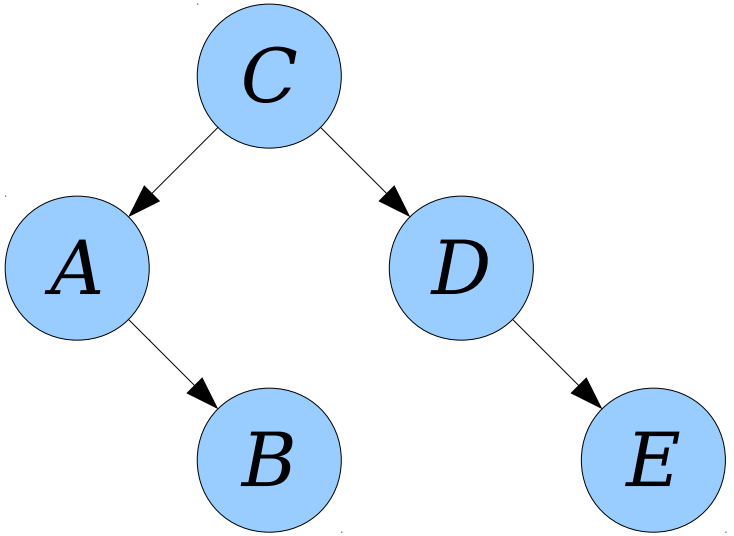
Why Rotate-to-Root Fails



Why Rotate-to-Root Fails



Why Rotate-to-Root Fails



Some Claims

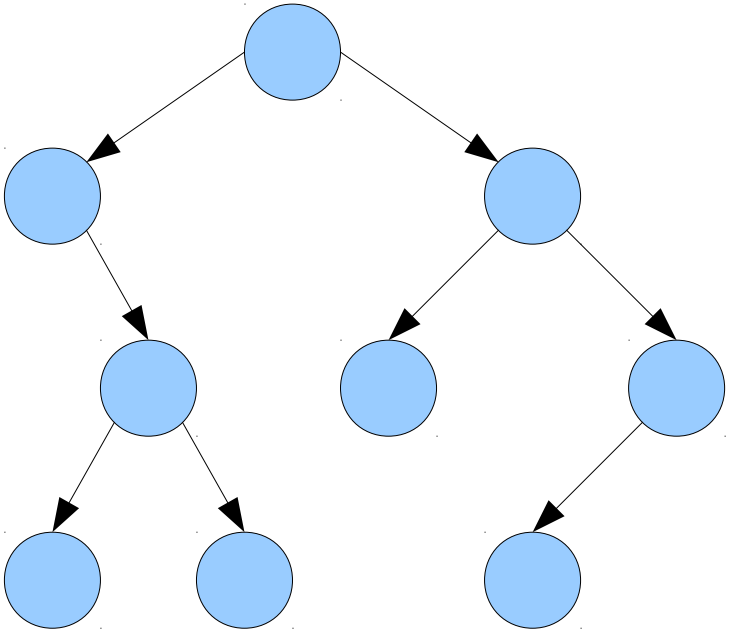
- ***Claim 1:*** The amortized cost of splaying a node up to the root is O(log n).
- ***Claim 2:*** The amortized cost of splaying a node up to the root can be o(log n) if the access pattern is non-uniform.
- We'll prove these results later today.

Making Things Easy

- Splay trees provide make it extremely easy to perform the following operations:
  - lookup
  - insert
  - delete
  - predecessor / successor
  - join
  - split
- Let's see why.

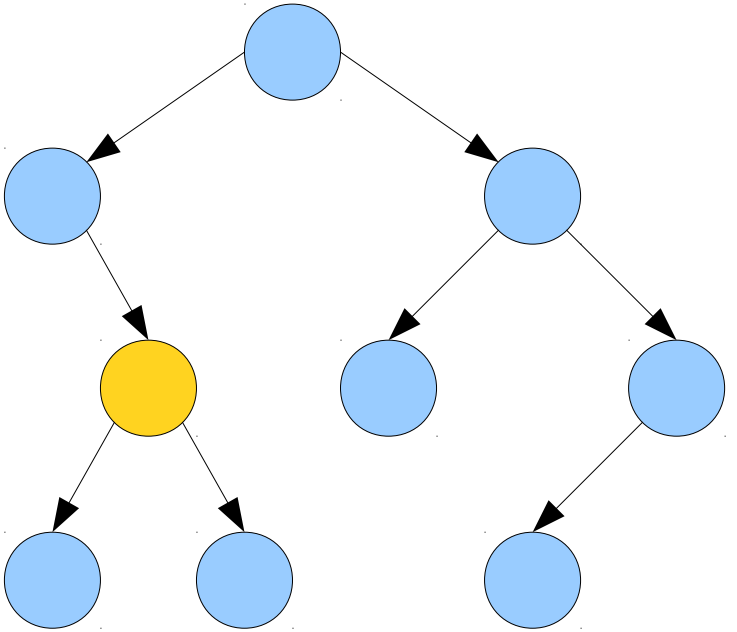
Lookups

- To do a lookup in a splay tree:
  - Search for that item as usual.
  - If it's found, splay it up to the root.
  - Otherwise, splay the last-visited node to the root.



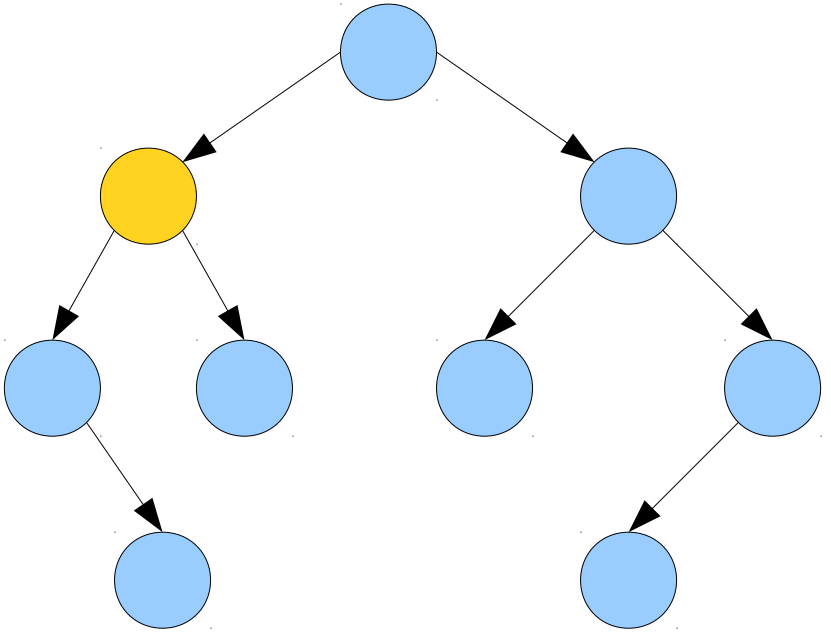
Lookups

- To do a lookup in a splay tree:
  - Search for that item as usual.
  - If it's found, splay it up to the root.
  - Otherwise, splay the last-visited node to the root.



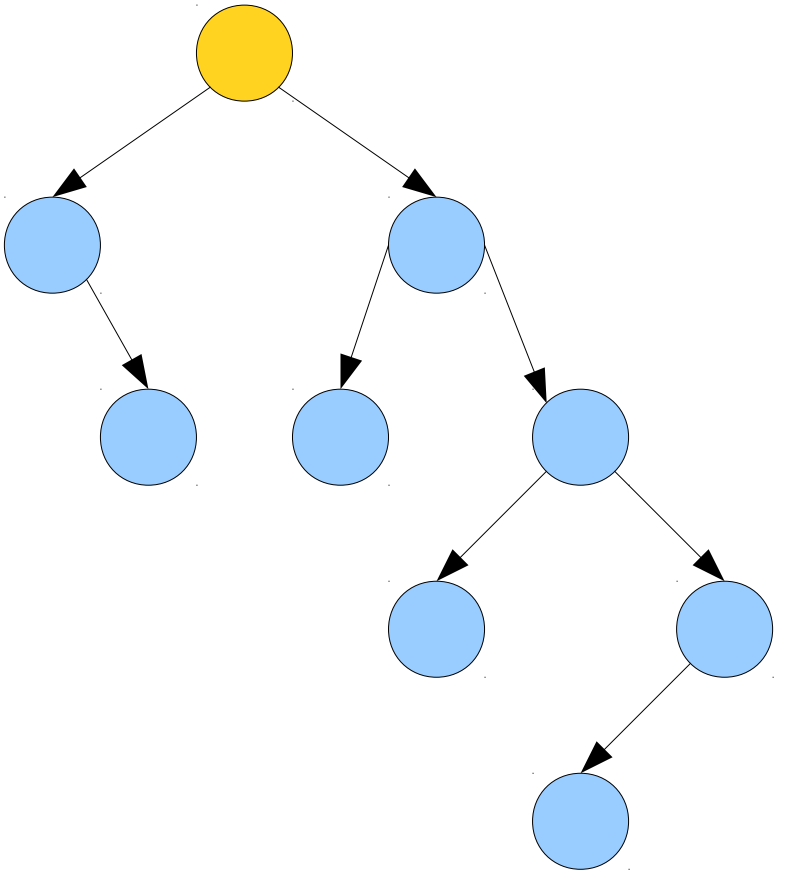
Lookups

- To do a lookup in a splay tree:
  - Search for that item as usual.
  - If it's found, splay it up to the root.
  - Otherwise, splay the last-visited node to the root.



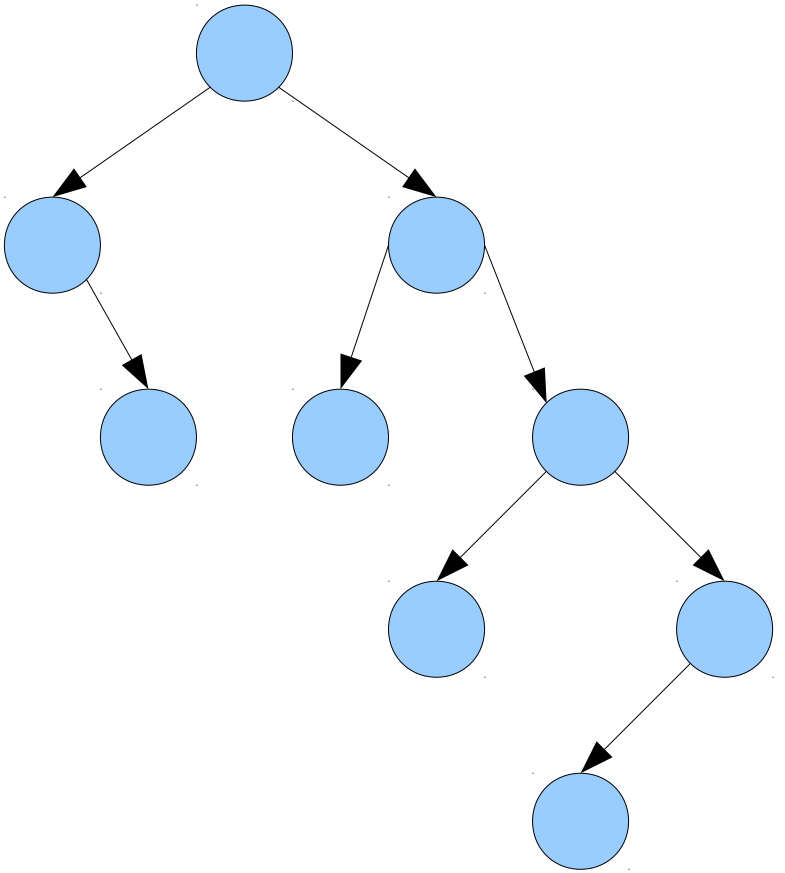
Lookups

- To do a lookup in a splay tree:
  - Search for that item as usual.
  - If it's found, splay it up to the root.
  - Otherwise, splay the last-visited node to the root.



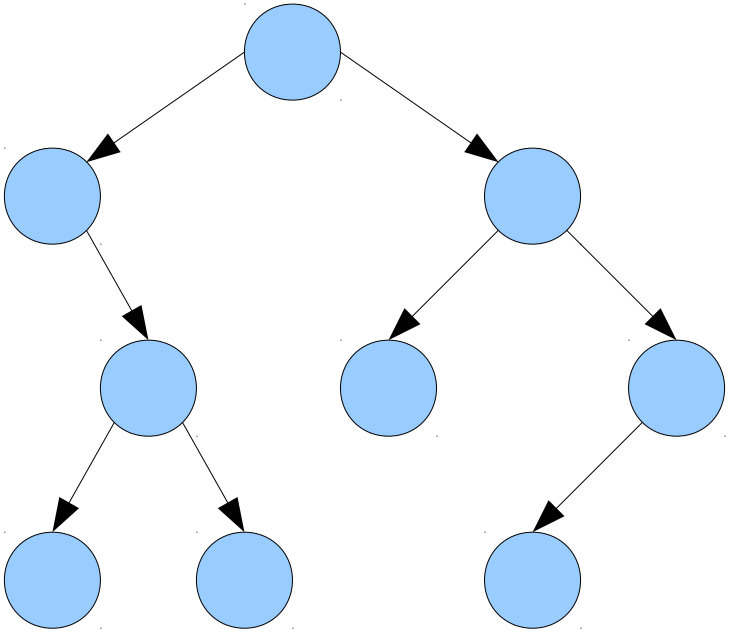
Lookups

- To do a lookup in a splay tree:
  - Search for that item as usual.
  - If it's found, splay it up to the root.
  - Otherwise, splay the last-visited node to the root.



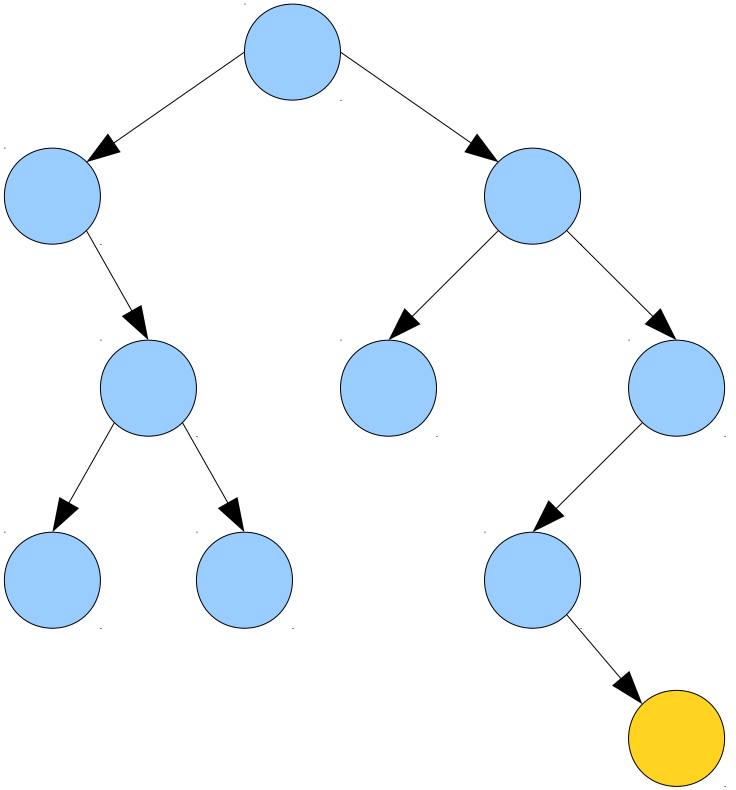
Insertions

- To insert a node into a splay tree:
  - Insert the node as usual.
  - Splay it up to the root.



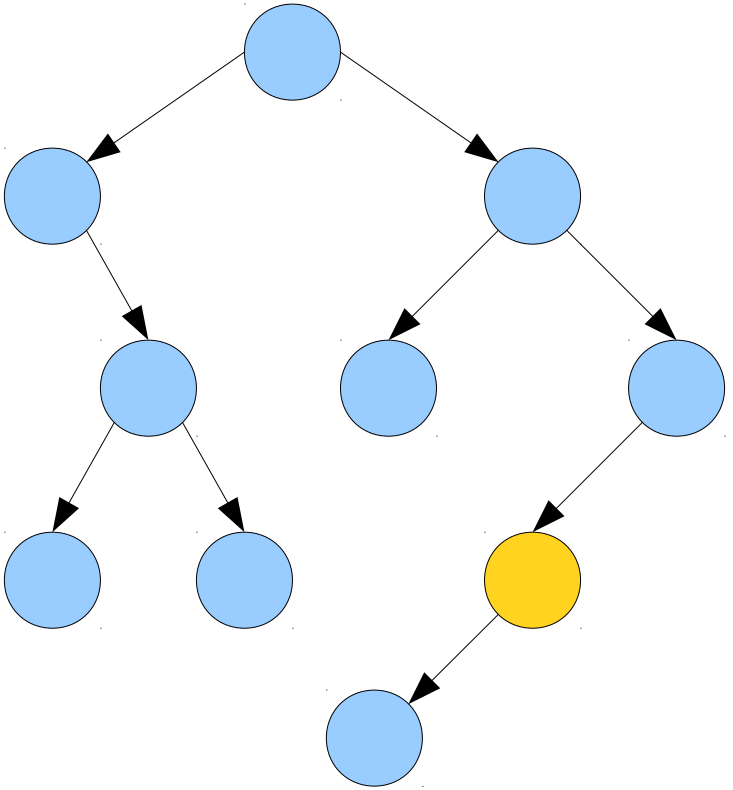
Insertions

- To insert a node into a splay tree:
  - Insert the node as usual.
  - Splay it up to the root.



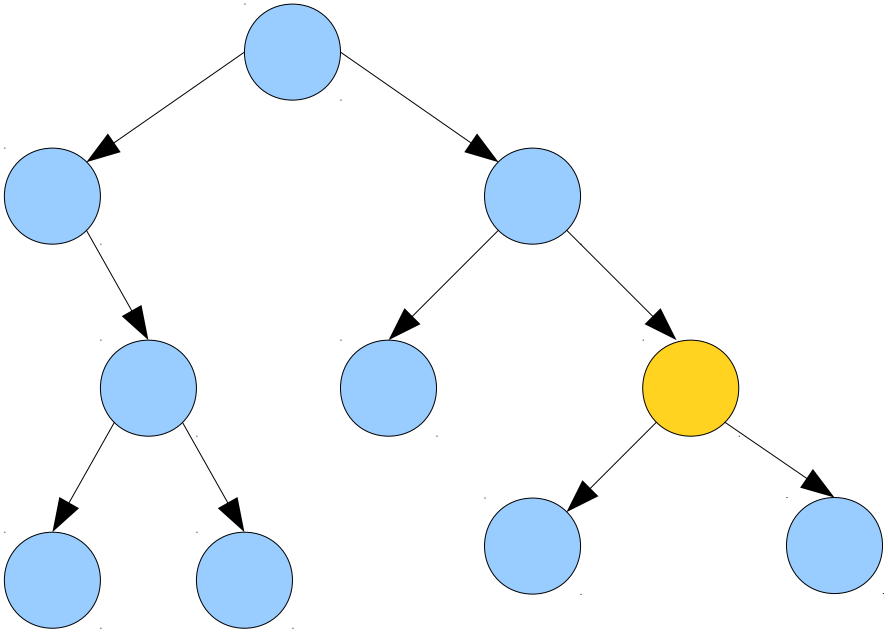
Insertions

- To insert a node into a splay tree:
  - Insert the node as usual.
  - Splay it up to the root.



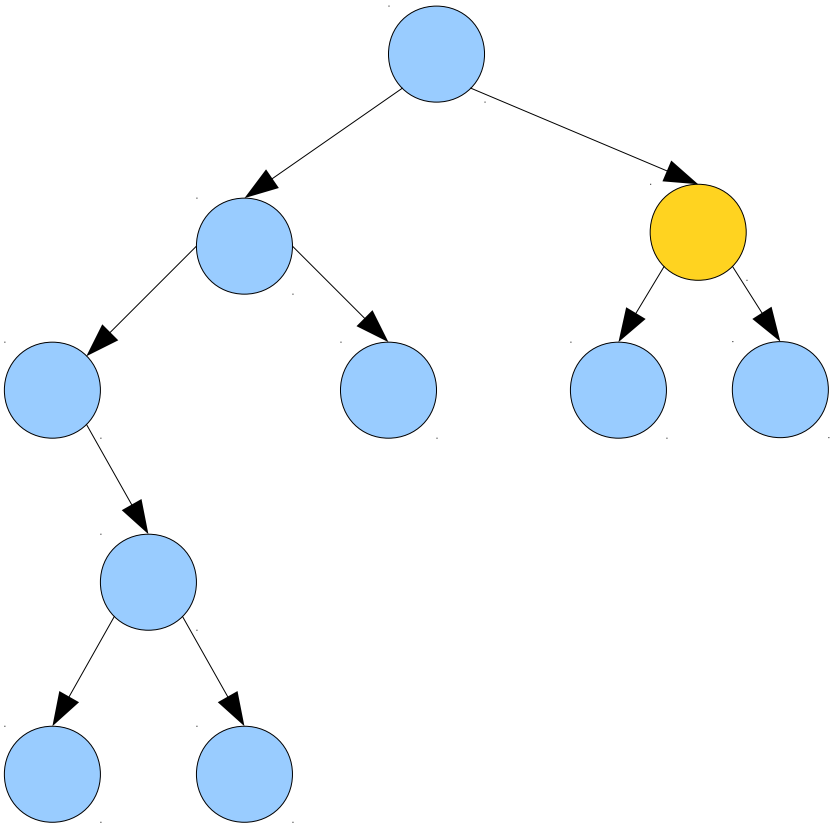
Insertions

- To insert a node into a splay tree:
  - Insert the node as usual.
  - Splay it up to the root.



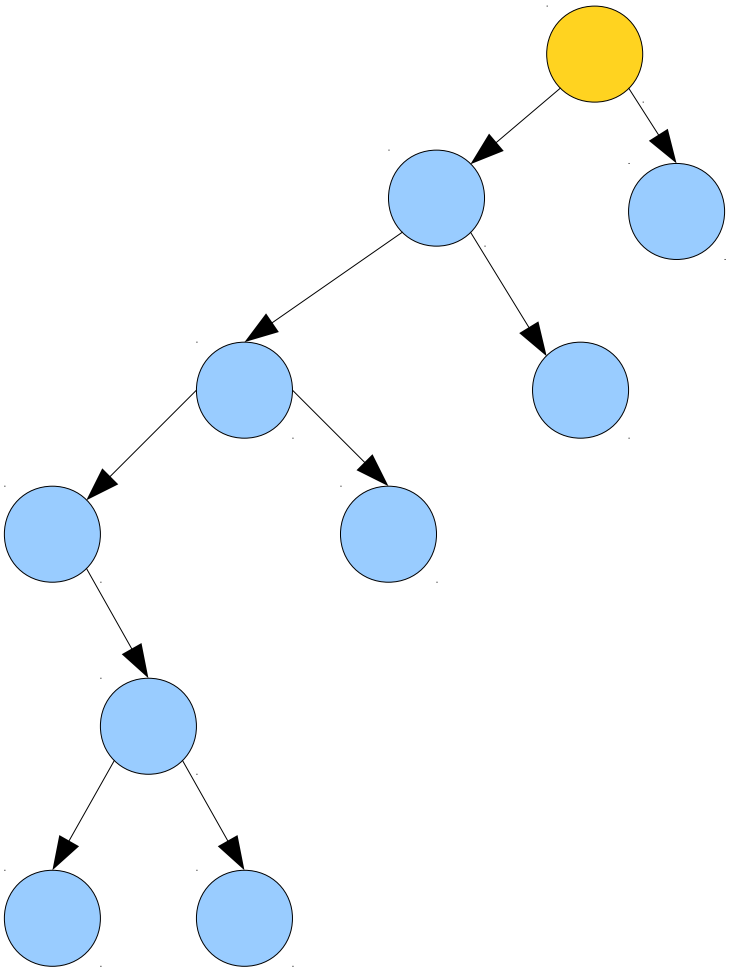
Insertions

- To insert a node into a splay tree:
  - Insert the node as usual.
  - Splay it up to the root.



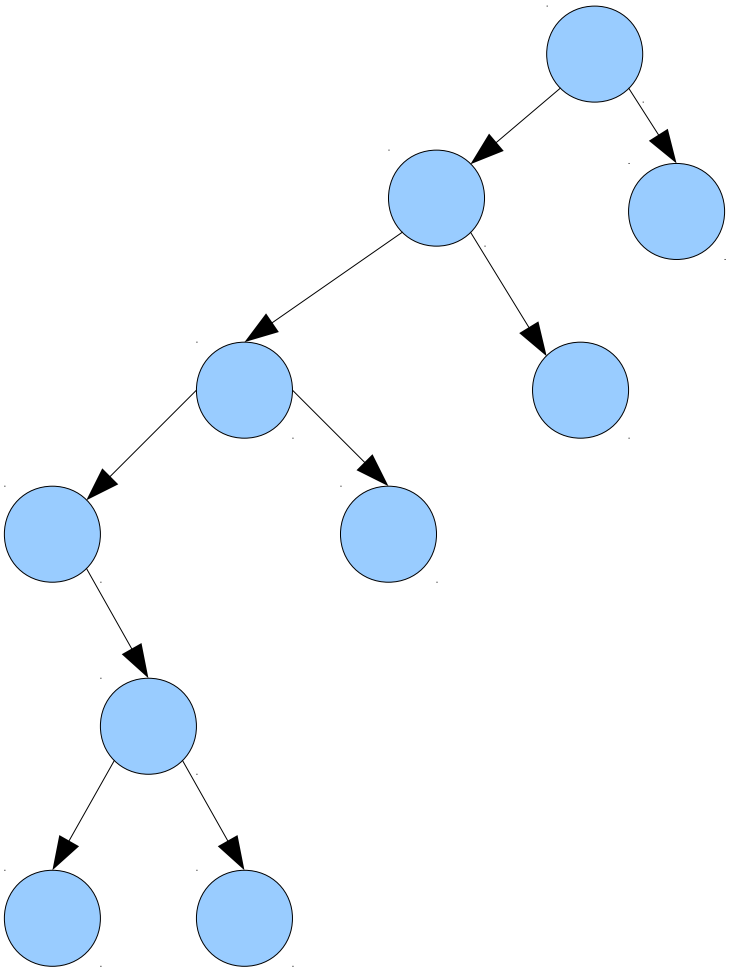
Insertions

- To insert a node into a splay tree:
  - Insert the node as usual.
  - Splay it up to the root.



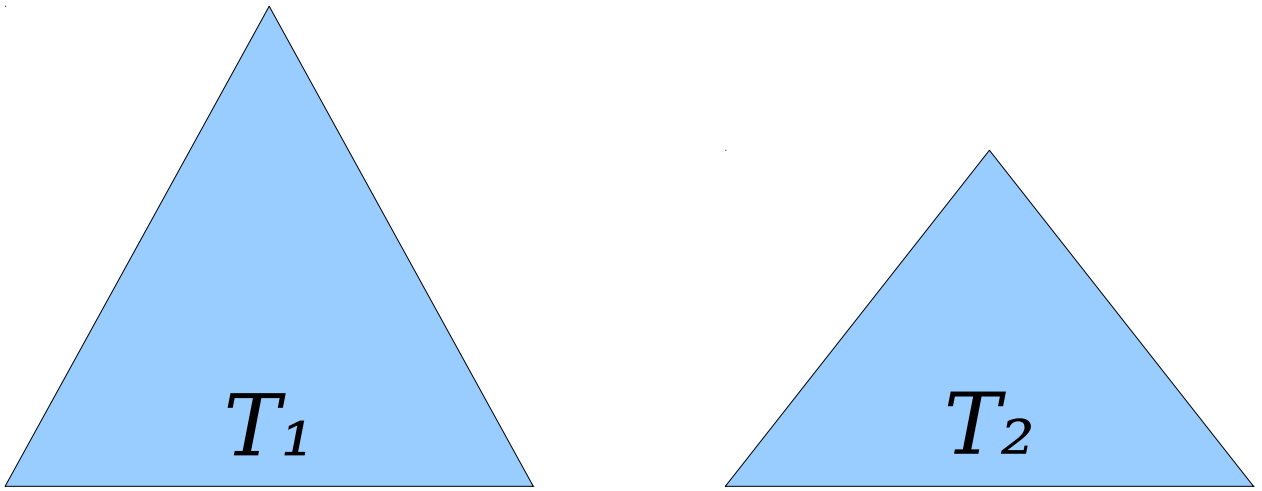
Insertions

- To insert a node into a splay tree:
  - Insert the node as usual.
  - Splay it up to the root.



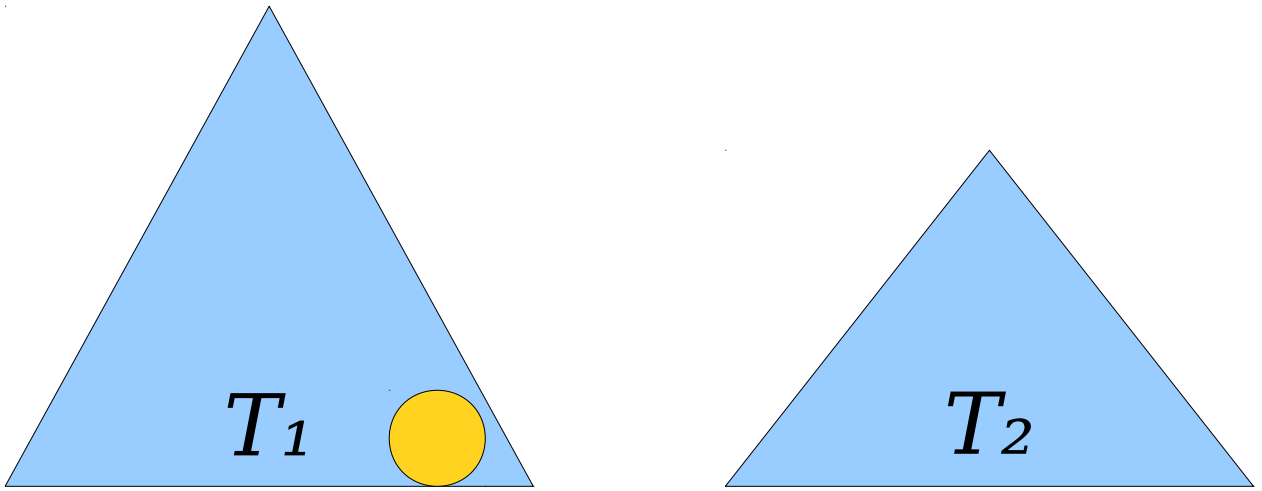
Join

- To join two trees T_1 and T_2, where all keys in T_1 are less than the keys in T_2:
  - Splay the max element of T_1 to the root.
  - Make T_2 a right child of T_1.



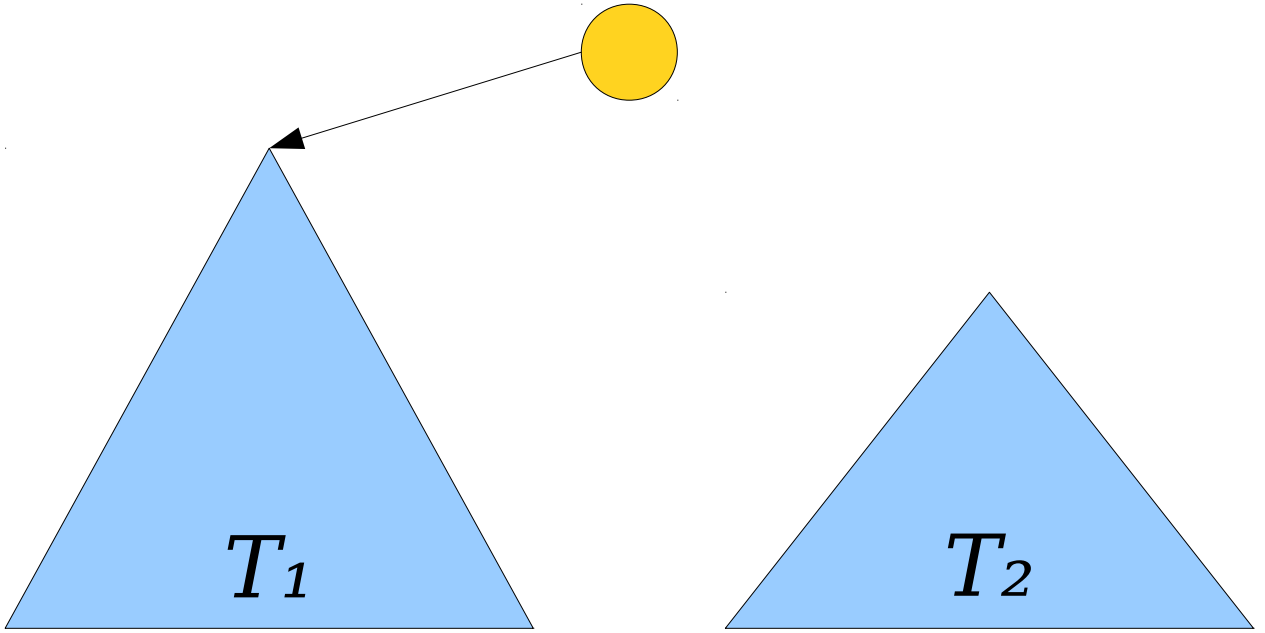
Join

- To join two trees T_1 and T_2, where all keys in T_1 are less than the keys in T_2:
  - Splay the max element of T_1 to the root.
  - Make T_2 a right child of T_1.



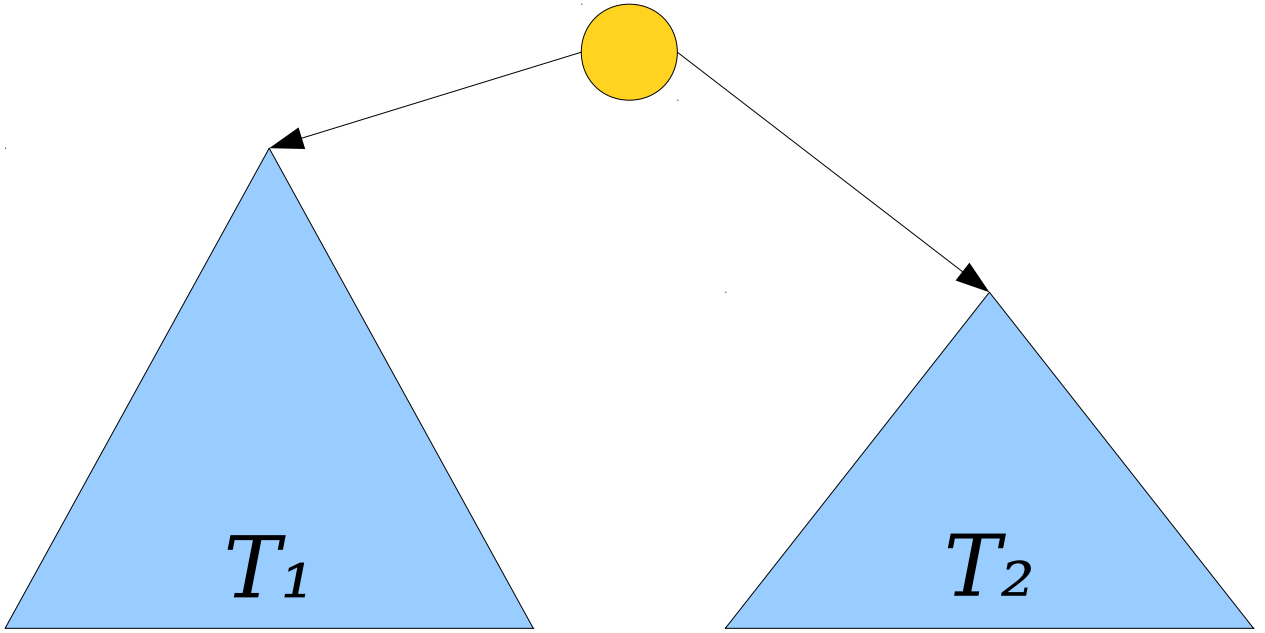
Join

- To join two trees T_1 and T_2, where all keys in T_1 are less than the keys in T_2:
  - Splay the max element of T_1 to the root.
  - Make T_2 a right child of T_1.



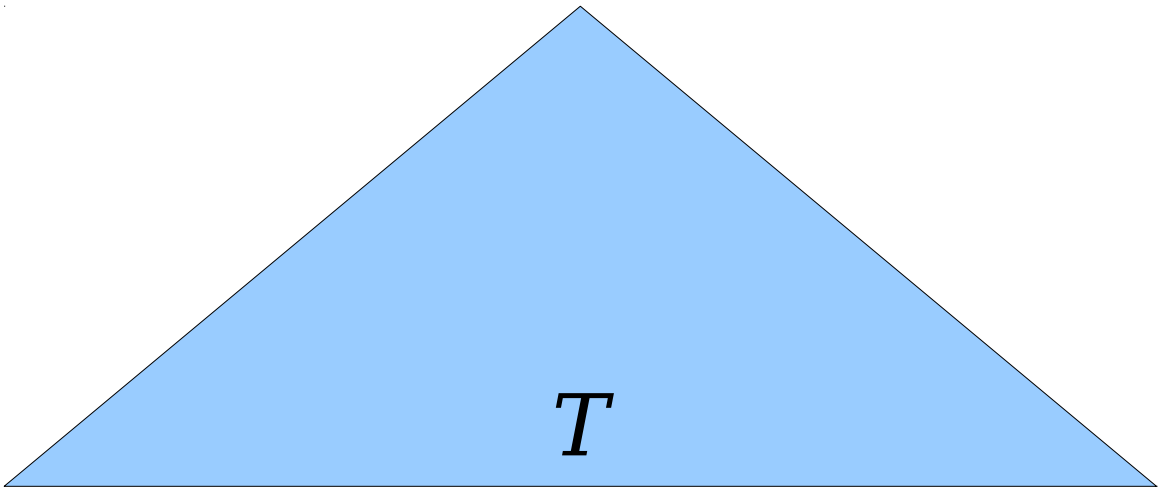
Join

- To join two trees T_1 and T_2, where all keys in T_1 are less than the keys in T_2:
  - Splay the max element of T_1 to the root.
  - Make T_2 a right child of T_1.



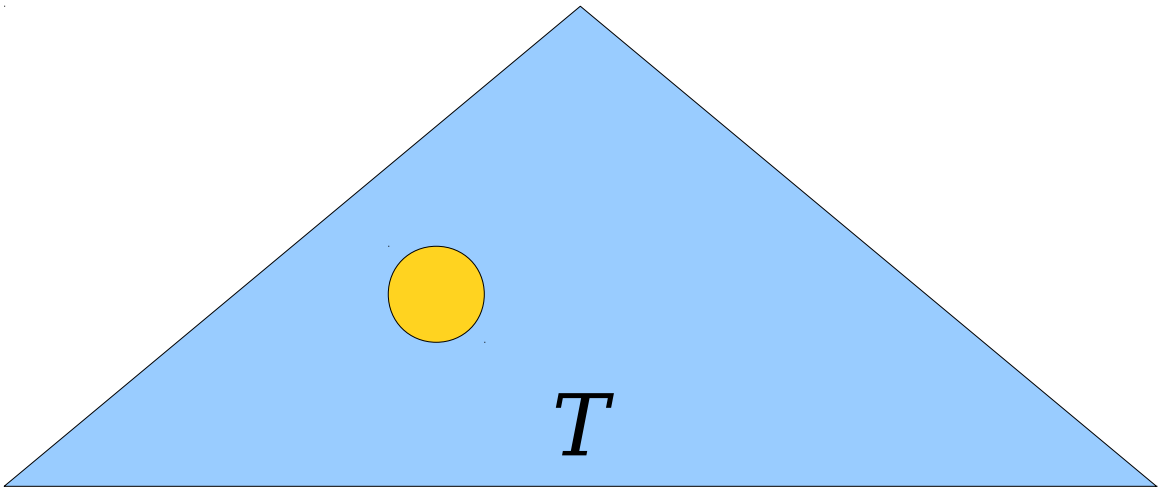
Split

- To split T at a key k:
  - Splay the successor of k up to the root.
  - Cut the link from the root to its left child.



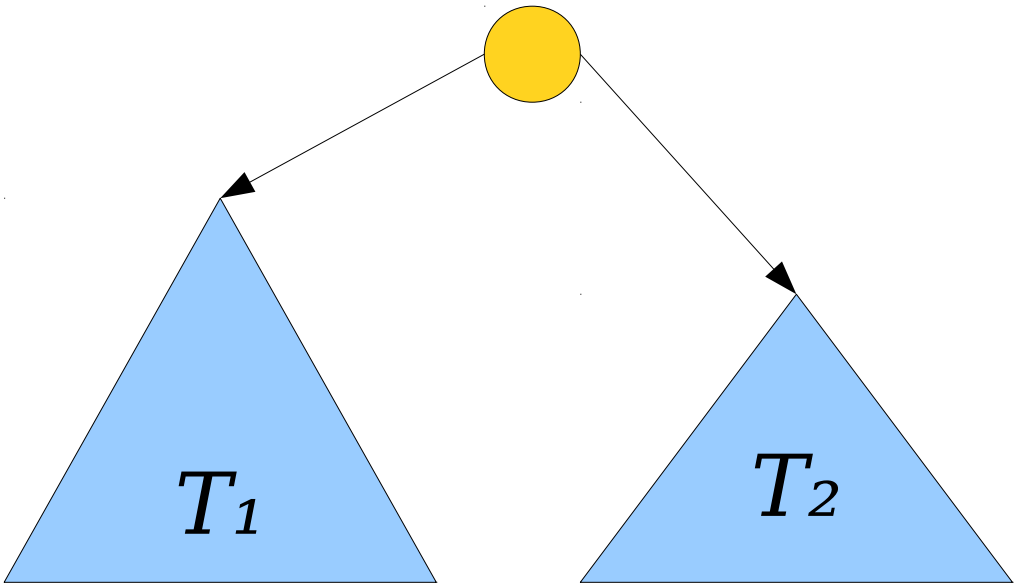
Split

- To split T at a key k:
  - Splay the successor of k up to the root.
  - Cut the link from the root to its left child.



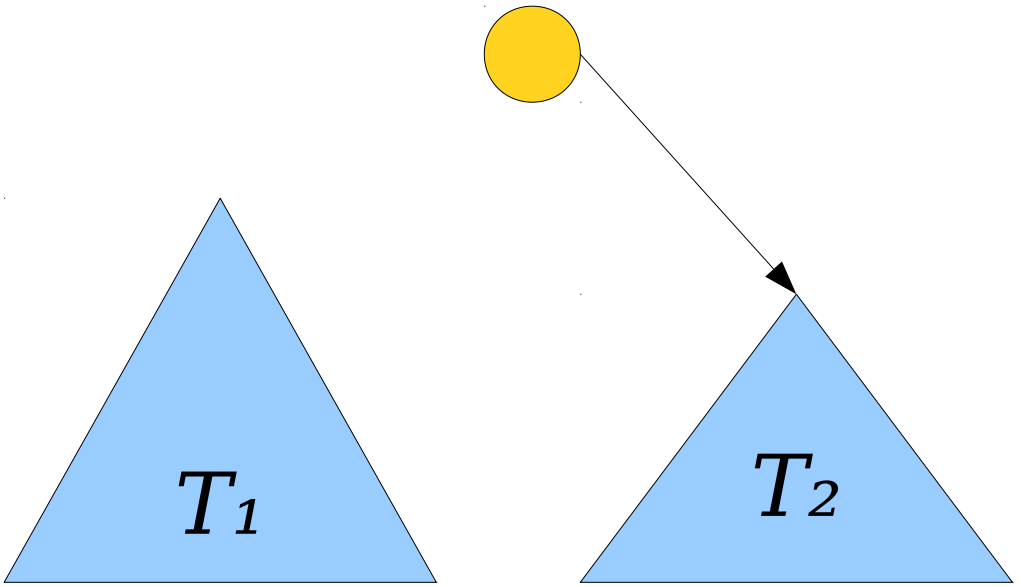
Split

- To split T at a key k:
  - Splay the successor of k up to the root.
  - Cut the link from the root to its left child.



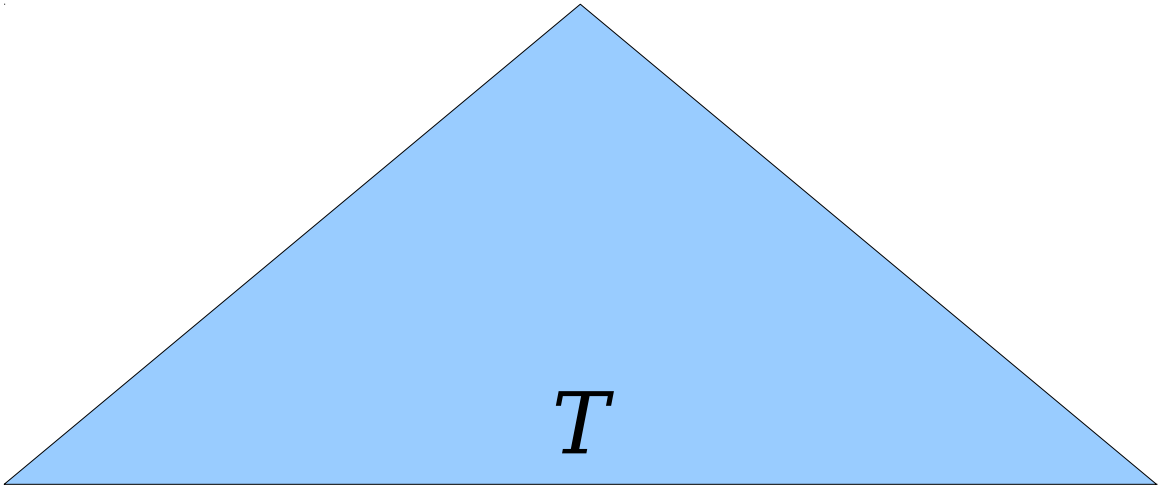
Split

- To split T at a key k:
  - Splay the successor of k up to the root.
  - Cut the link from the root to its left child.



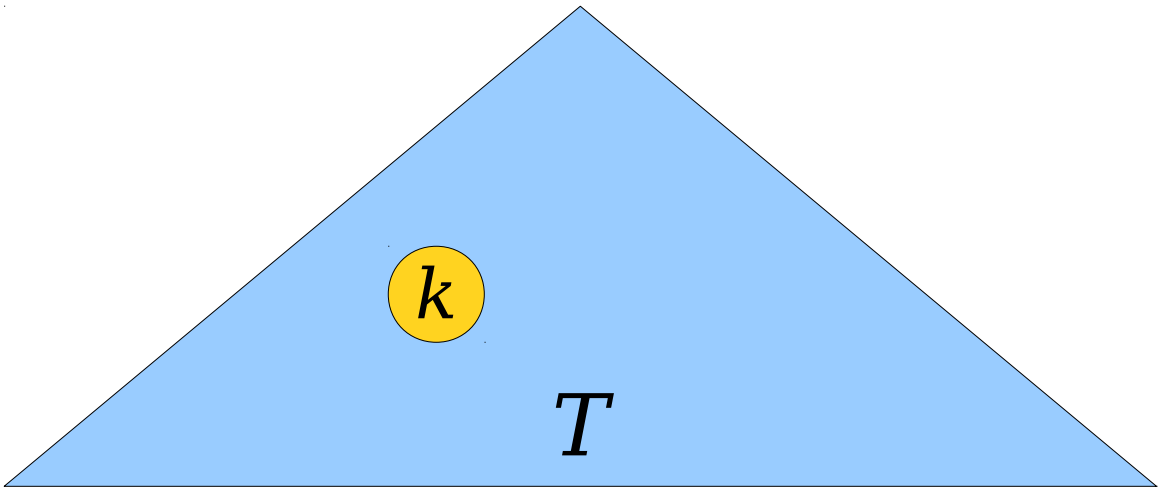
Delete

- To delete a key k from the tree:
  - Splay k to the root.
  - Delete k.
  - Join the two resulting subtrees.



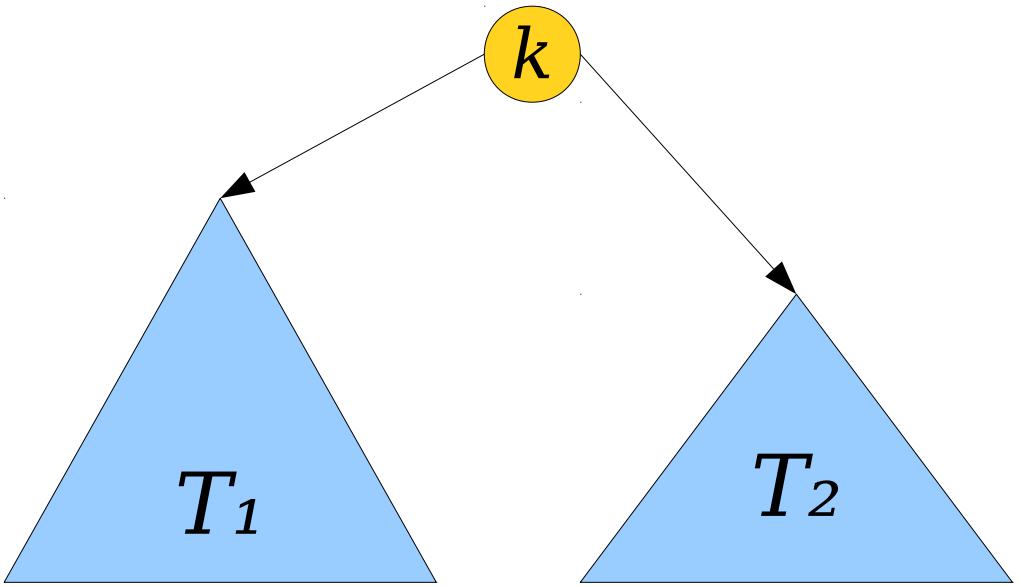
Delete

- To delete a key k from the tree:
  - Splay k to the root.
  - Delete k.
  - Join the two resulting subtrees.



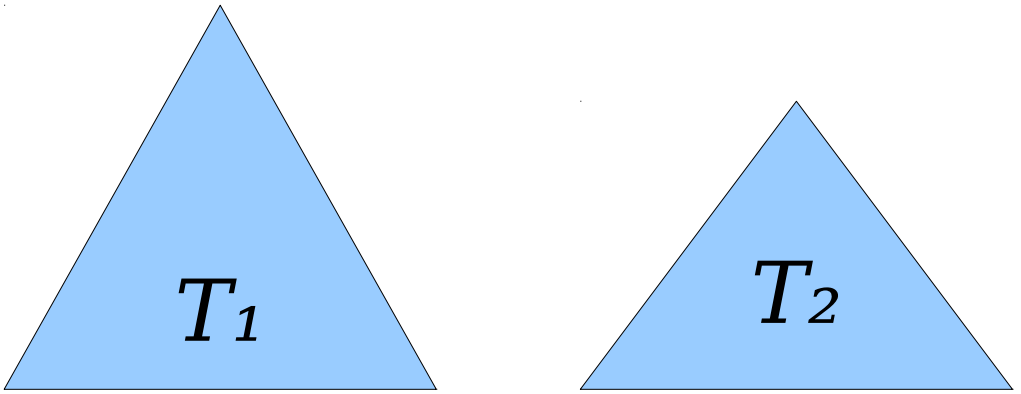
Delete

- To delete a key k from the tree:
  - Splay k to the root.
  - Delete k.
  - Join the two resulting subtrees.



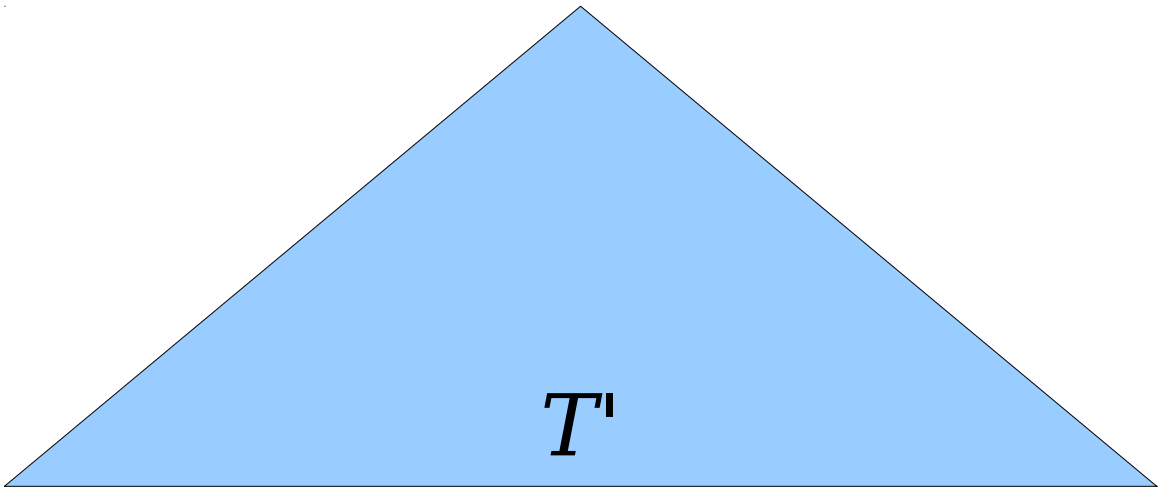
Delete

- To delete a key k from the tree:
  - Splay k to the root.
  - Delete k.
  - Join the two resulting subtrees.



Delete

- To delete a key k from the tree:
  - Splay k to the root.
  - Delete k.
  - Join the two resulting subtrees.



The Runtime

- ***Claim:*** All of these operations require amortized time O(log n).
- ***Rationale:*** Each has runtime bounded by the cost of O(1) splays, which takes total amortized time O(log n).
- Contrast this with red/black trees:
  - No need to store any kind of balance information.
  - Only three rules to memorize.

So… just how fast *are* splay trees?

Time-Out for Announcements!

Problem Set Four

- Problem Set Four goes out today. It's due next Thursday at 3:00PM.
  - Play around with amortized analyses, binomial heaps, Fibonacci heaps, and splay trees!
- Problem Set Three solutions are available in hardcopy. Grab them from Gates if you missed lecture today.
- Have questions? Feel free to stop by office hours or to ask on Piazza!

Final Project Logistics

- For the final project, you'll work in teams of two or three to research a data structure and then do something “interesting” with it.
- You will then
  - write a paper explaining the data structure, proving key results, and describing your “interesting” component; then
  - give a 10-15 minute presentation on your data structure and your “interesting” component.
- The “interesting” component is entirely up to you. Be creative! Have fun with this!

Final Project Logistics

- To ensure that we don't have too many teams presenting on the same topics, your first step is to submit a list of project topics you'd like to work on.
- By next Tuesday (May 10), form a team and submit to us a list of ***seven*** project topics you'd be interested in. For each topic, find a research paper on the subject and at least one supplementary source.
- We'll then run a matchmaking algorithm and get back to everyone with project topics by next Thursday.
- We've put together a list of recommended project topics on the course website, but you're not required to choose one of those topics. Be creative! Go exploring! If you find an interesting topic not on the list, you're encouraged to propose it!

WiCS Board

Interested in playing a larger role in the WiCS community? Passionate about encouraging and supporting women in CS?

Apply to be on WiCS Board 16-17!

From HackOverflow to the industry mentorship program, WiCS is constantly growing and improving the community for women in tech. As a board member, you'll be able to meet engineers and leaders from companies like Google and Pinterest, organize HackOverflow, meet women in STEM from all across the nation, and develop lasting relationships with your WiCS family. Learn more about the available teams and initiatives here.

Applications are due **Wednesday, May 4th, at 11:59 pm!** We look forward to reading your application, and please don't hesitate to reach out if you have questions.

Learn more about the events and programs we organize at wics.stanford.edu.

Latin@ Coder Summit

Next Saturday, May 14th, SOLE is hosting the Latin@ Coder Summit! We already have ~200 attendees signed up, but we need more volunteers to help run the event!

We have volunteer slots that only go until 1:00pm. We'd really appreciate your help, even if it's only for that time slot!

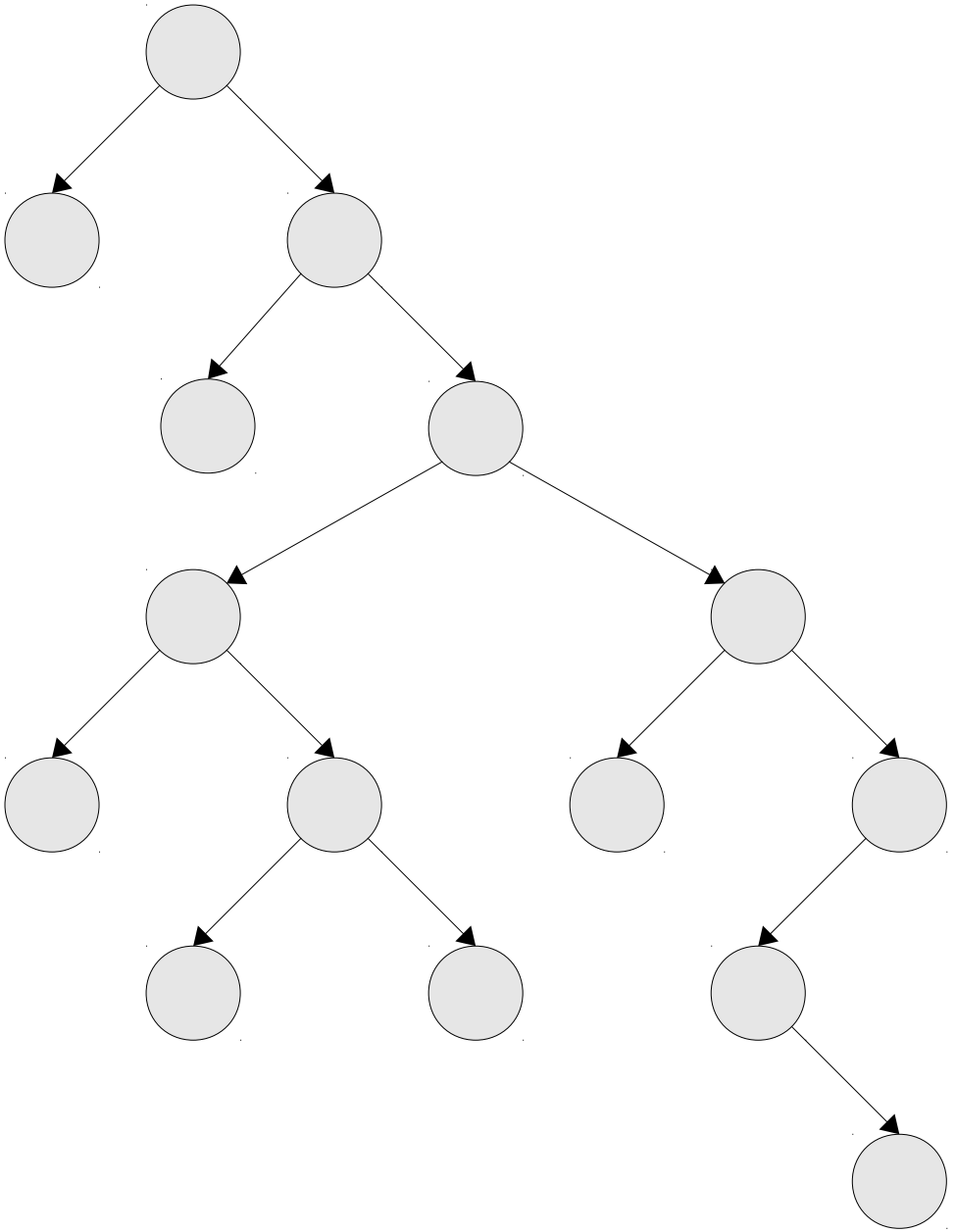
Please sign up ***here***!

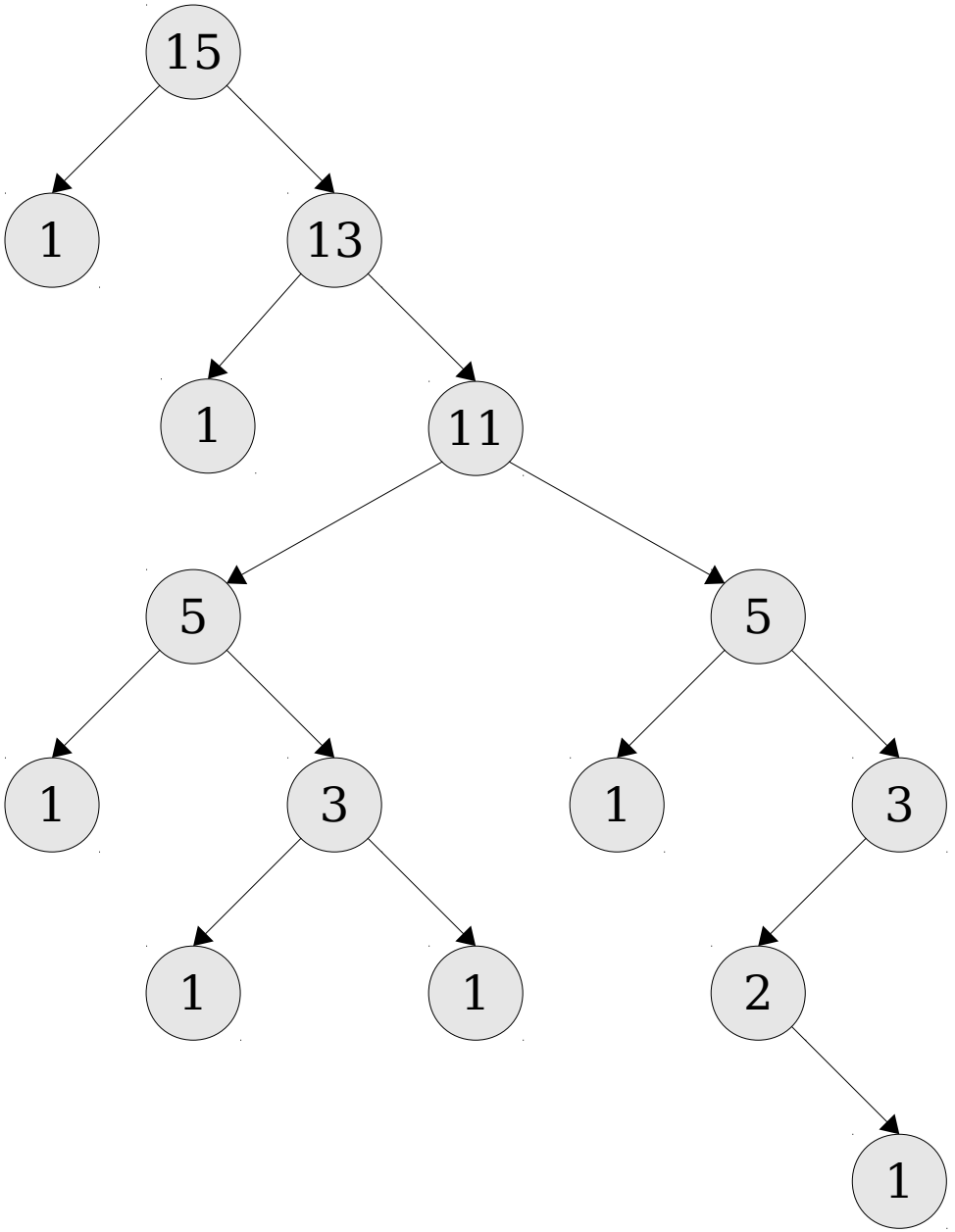
Back to CS166!

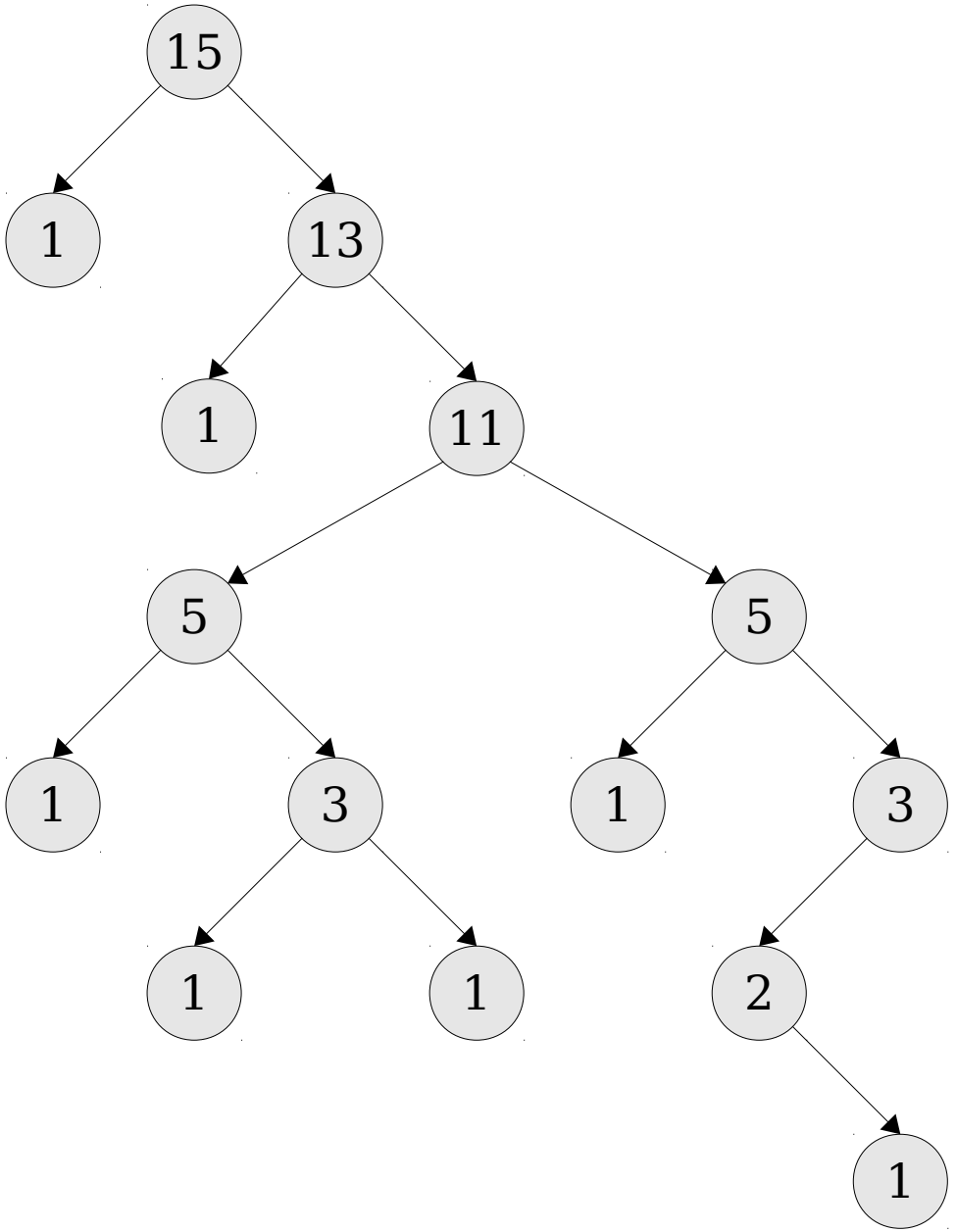
The Tricky Part: Formalizing This

Analyzing BSTs

- Let's assume that every element x_i in our BST has some associated weight w_i.
- We'll assume that access probabilities are proportional to weights, though we won't assume the weights sum to 1. (This is just for mathematical simplicity.)
- Let W denote the sum $w_1 + w_2 + \ldots + w_n$.
- Imagine we have some fixed BST T containing the keys $x_1, \ldots, x_n$. Let s_i denote the sum of all the weights of the keys in the subtree rooted at x_i.

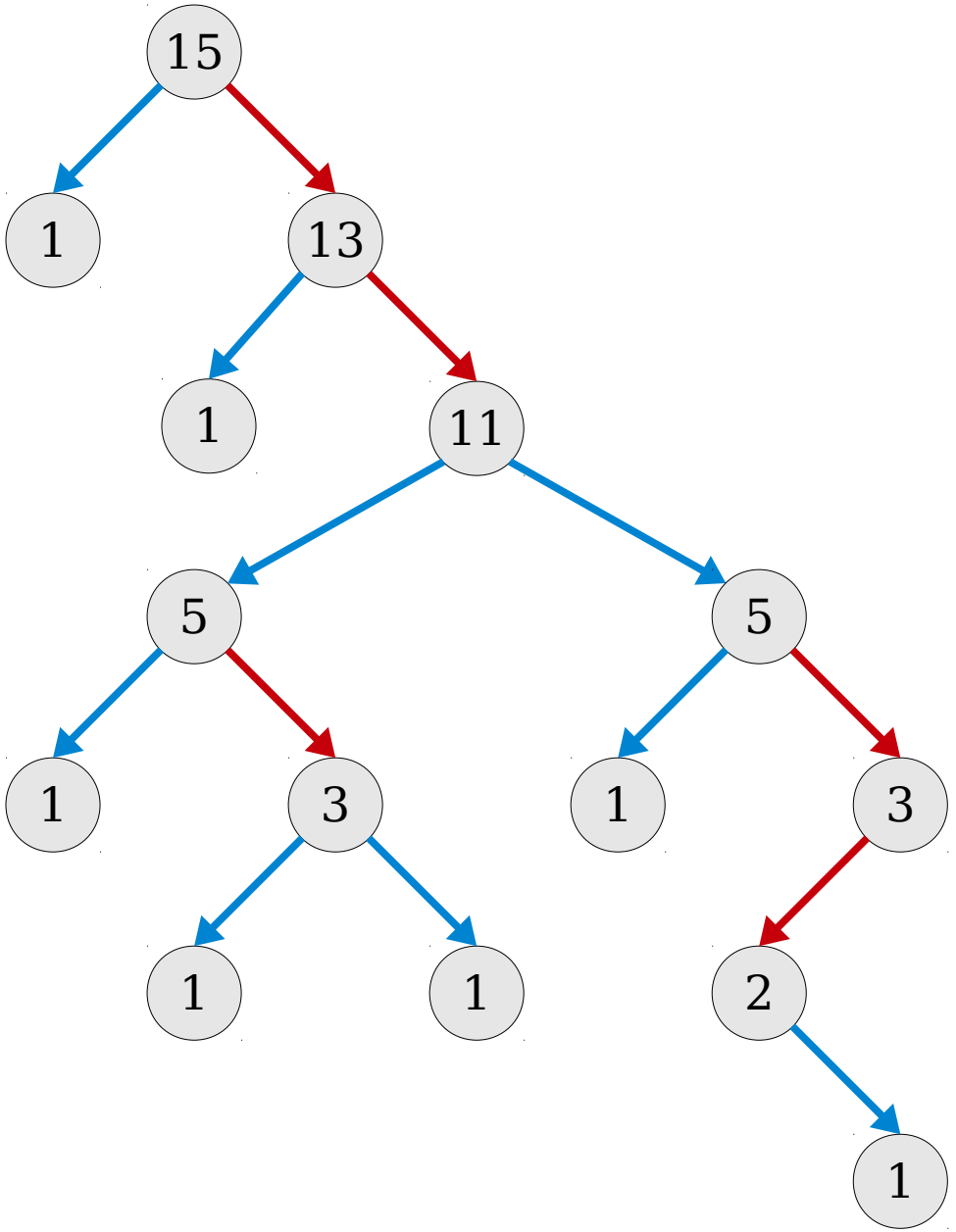



15
1
13
1
11
5
5
1
3
1
3
1
1
2
1



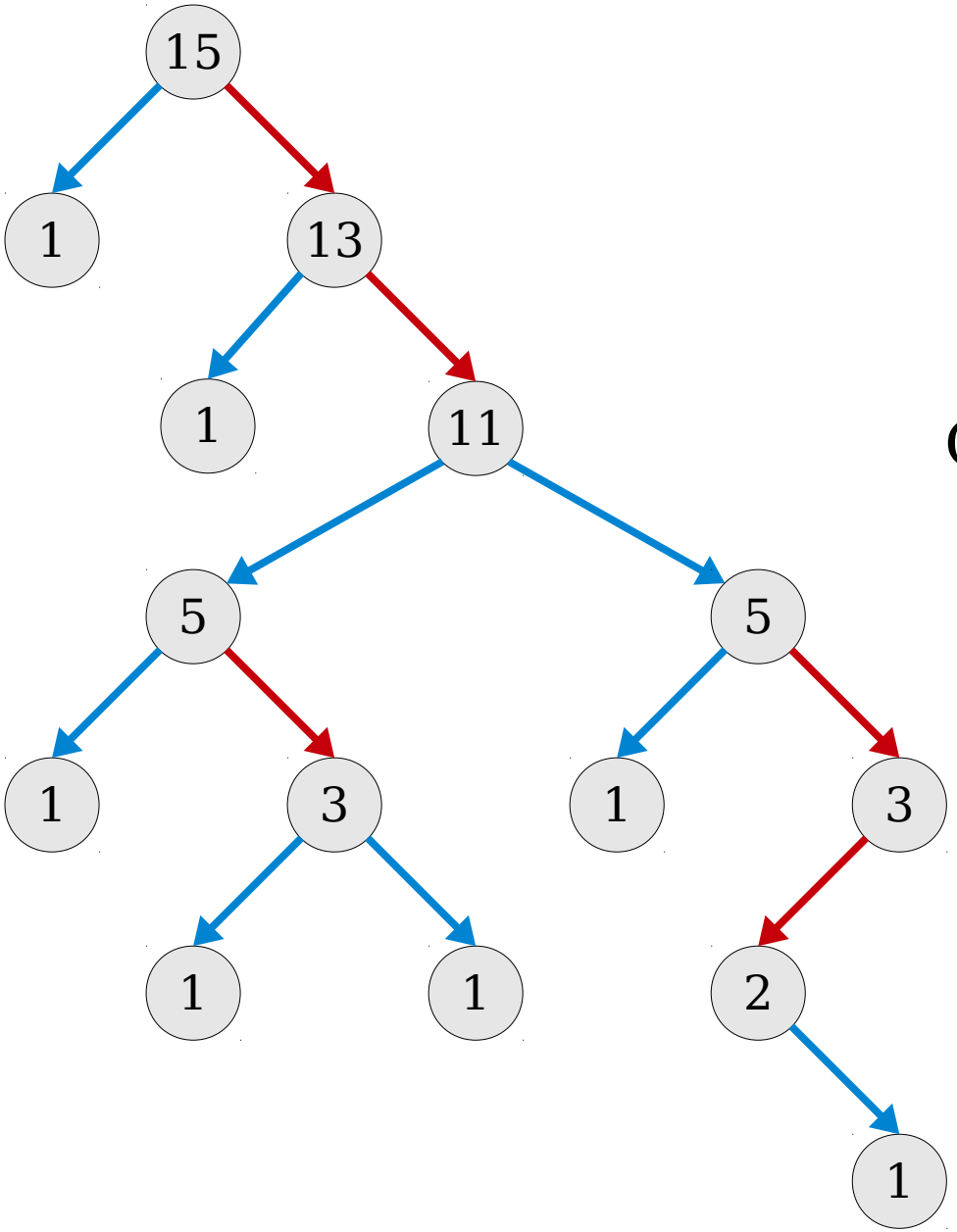
$s_{new} > \frac{1}{2} s_{old}$

$s_{new} \leq \frac{1}{2} s_{old}$



$s_{new} > \frac{1}{2} s_{old}$

$s_{new} \leq \frac{1}{2} s_{old}$

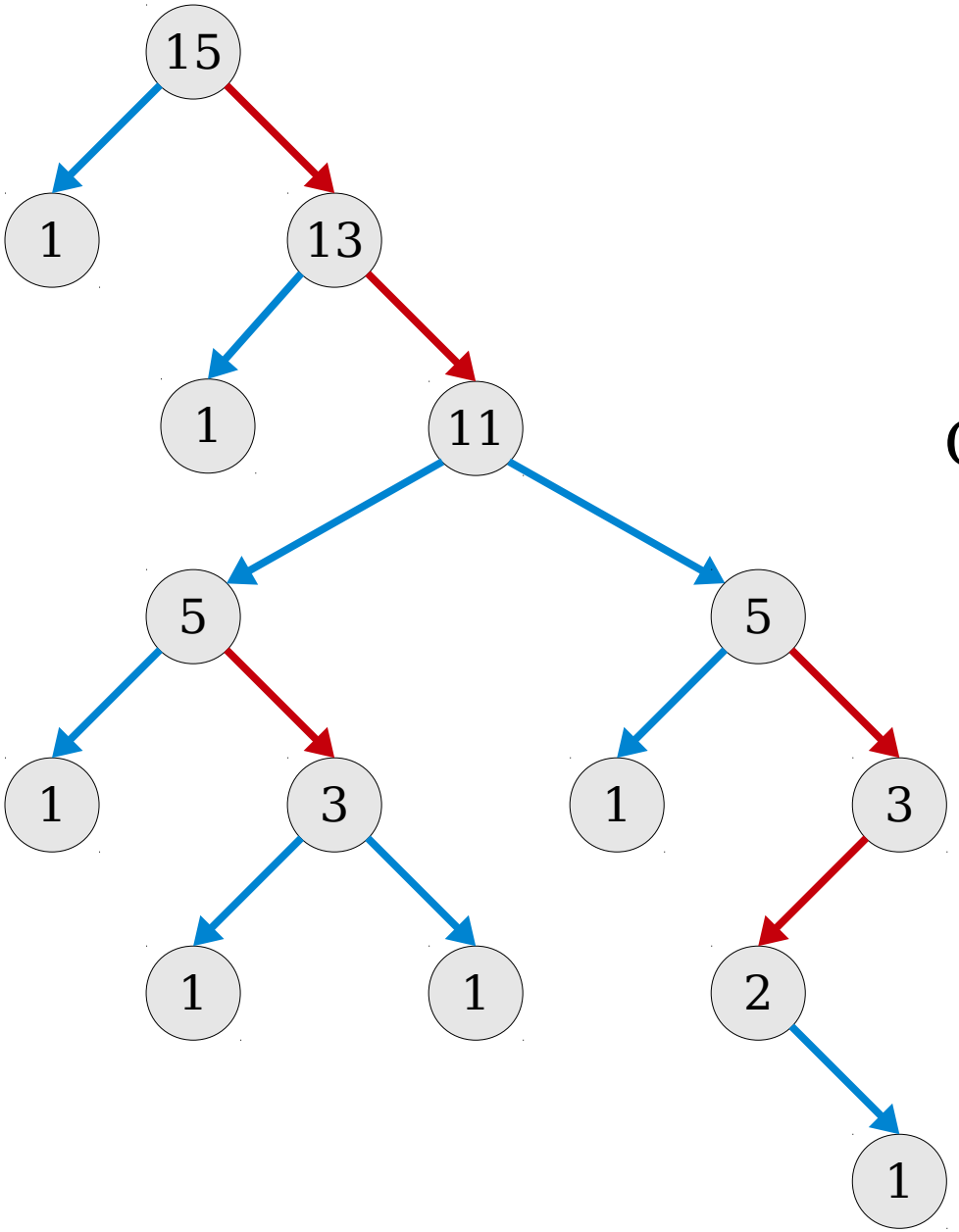


$s_{new} > \frac{1}{2} s_{old}$

$s_{new} \leq \frac{1}{2} s_{old}$

Cost of looking up x_i:

O(**#blue-used** + **#red-used**)

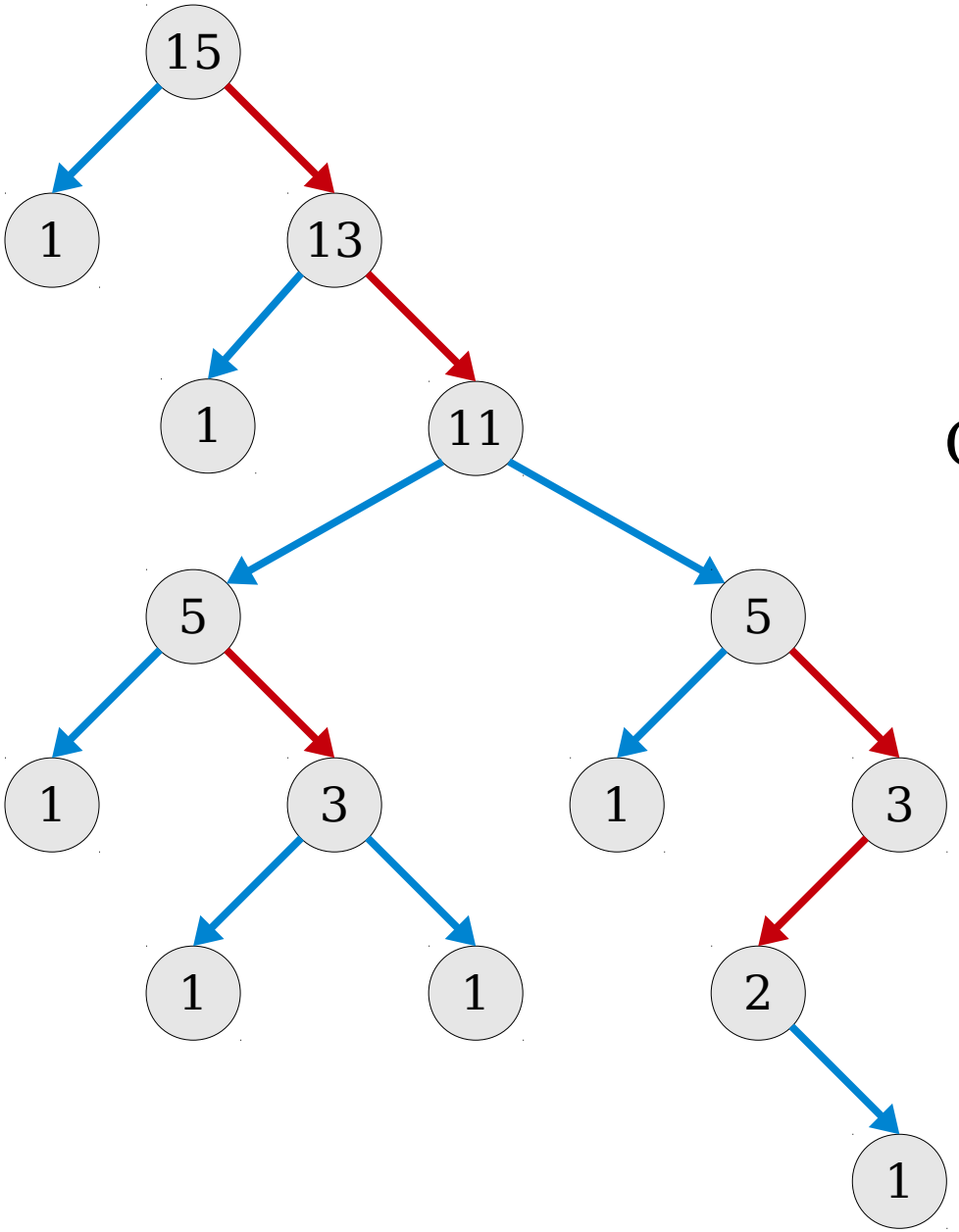


$s_{new} > \frac{1}{2} s_{old}$

$s_{new} \leq \frac{1}{2} s_{old}$

Cost of looking up x_i:

O(**log (W / w_i)** + **#red-used**)



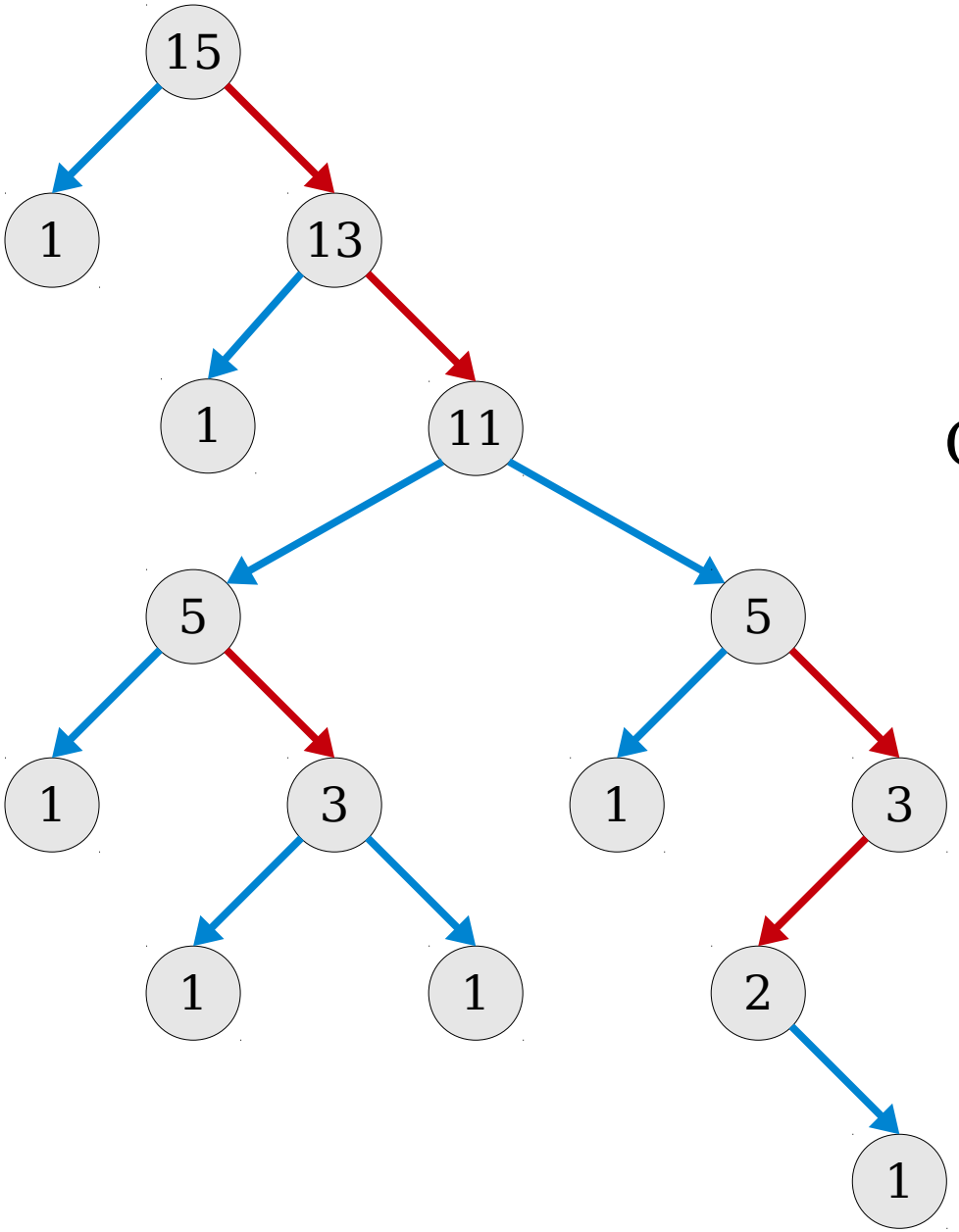
→ $s_{new} > \frac{1}{2} s_{old}$

→ $s_{new} \leq \frac{1}{2} s_{old}$

Cost of looking up x_i:

O(**log (W / w_i)** + **#red-used**)

Every blue edge throws away half of the total weight remaining.



→ $s_{new} > \frac{1}{2} s_{old}$

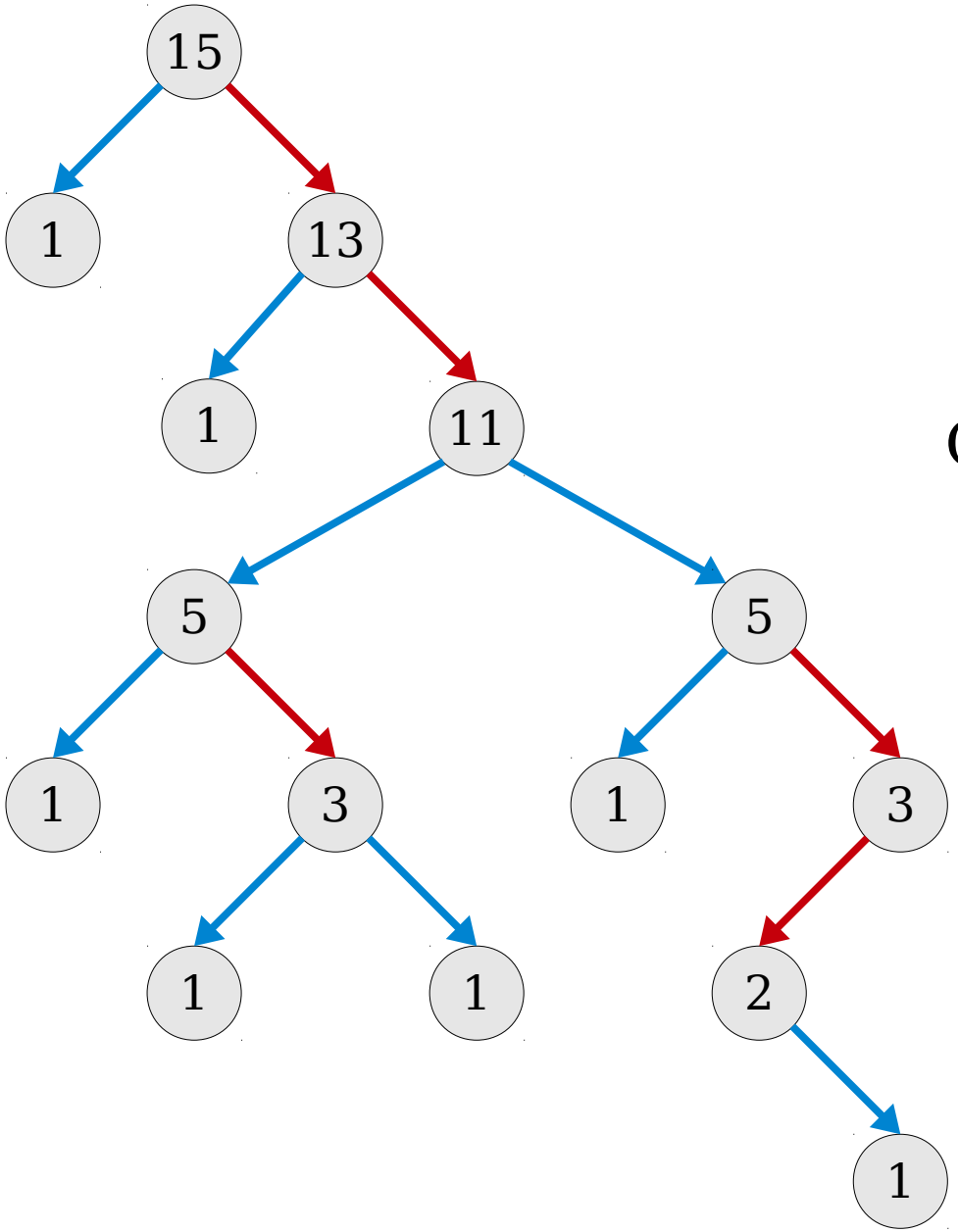
→ $s_{new} \leq \frac{1}{2} s_{old}$

Cost of looking up x_i:

O(**log (W / w_i)** + **#red-used**)

Every blue edge throws away half of the total weight remaining.

We begin with W total weight. If we're searching for x_i, the total weight at node x_i must be at least w_i.



→ $s_{new} > \frac{1}{2} s_{old}$

→ $s_{new} \leq \frac{1}{2} s_{old}$

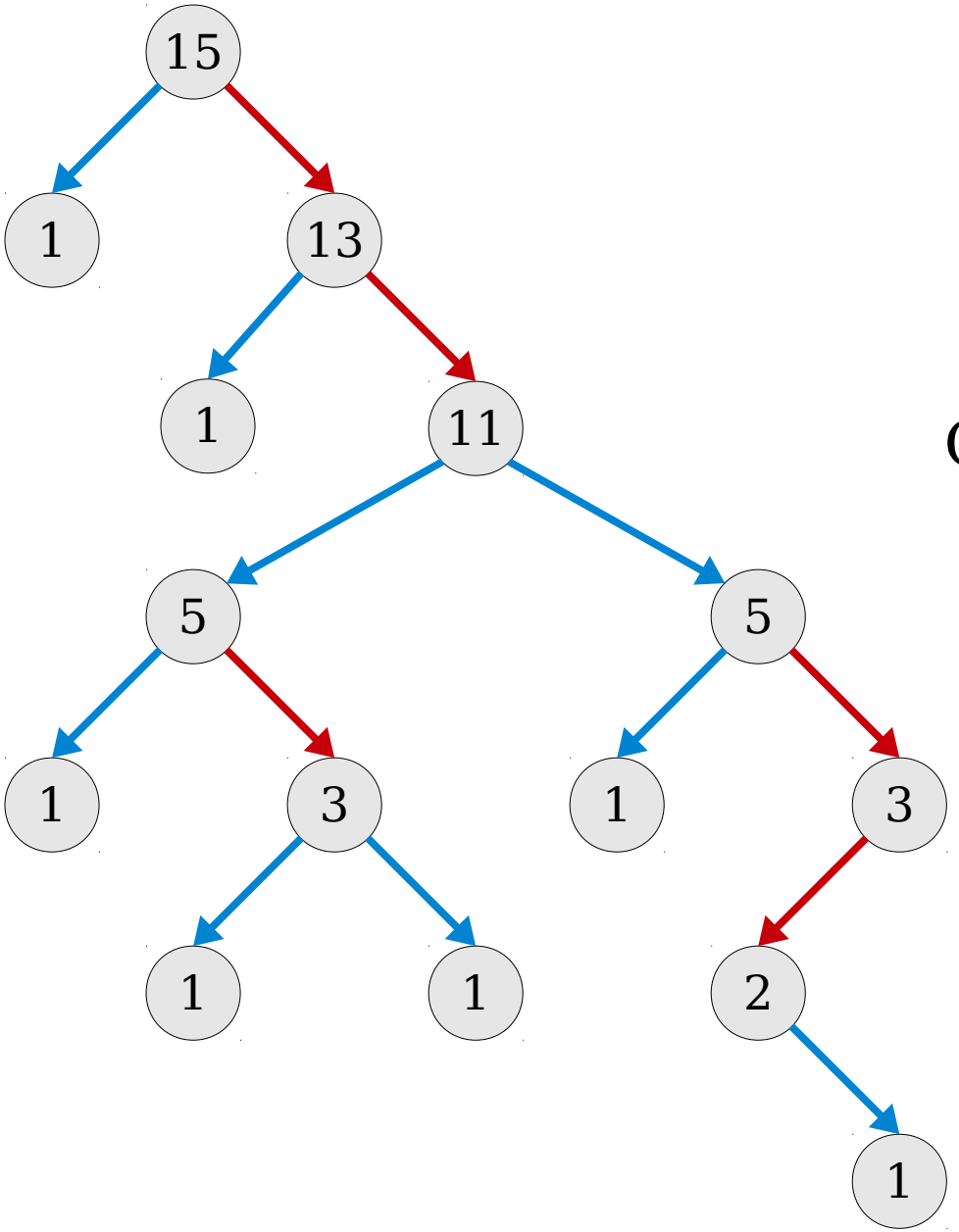
Cost of looking up x_i:

O(**log (W / w_i)** + **#red-used**)

Every blue edge throws away half of the total weight remaining.

We begin with W total weight. If we're searching for x_i, the total weight at node x_i must be at least w_i.

You can only throw away half the total weight log (W / w_i) times before the total weight drops to w_i.

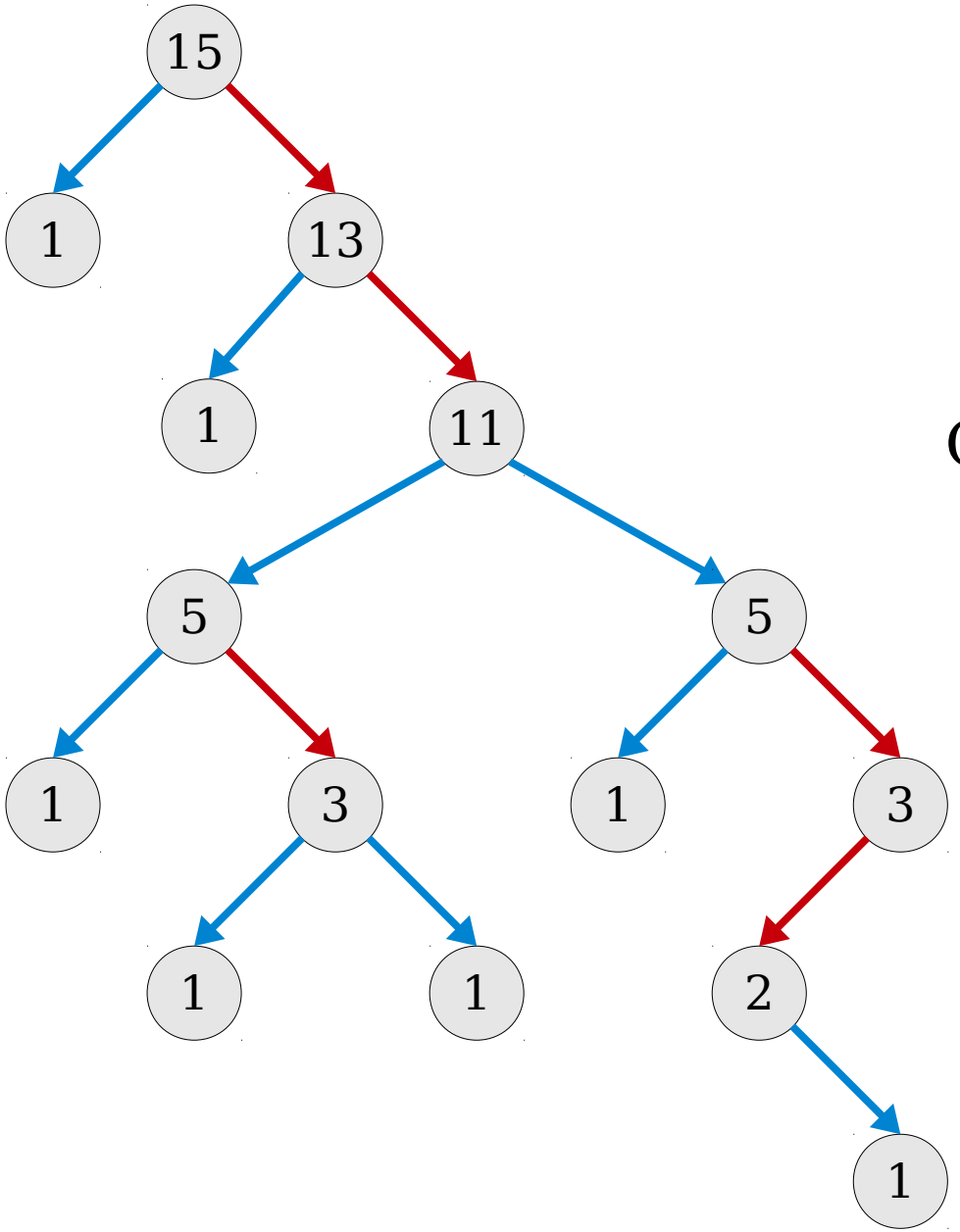


$s_{new} > \frac{1}{2} s_{old}$

$s_{new} \leq \frac{1}{2} s_{old}$

Cost of looking up x_i:

O(**log (W / w_i)** + **#red-used**)



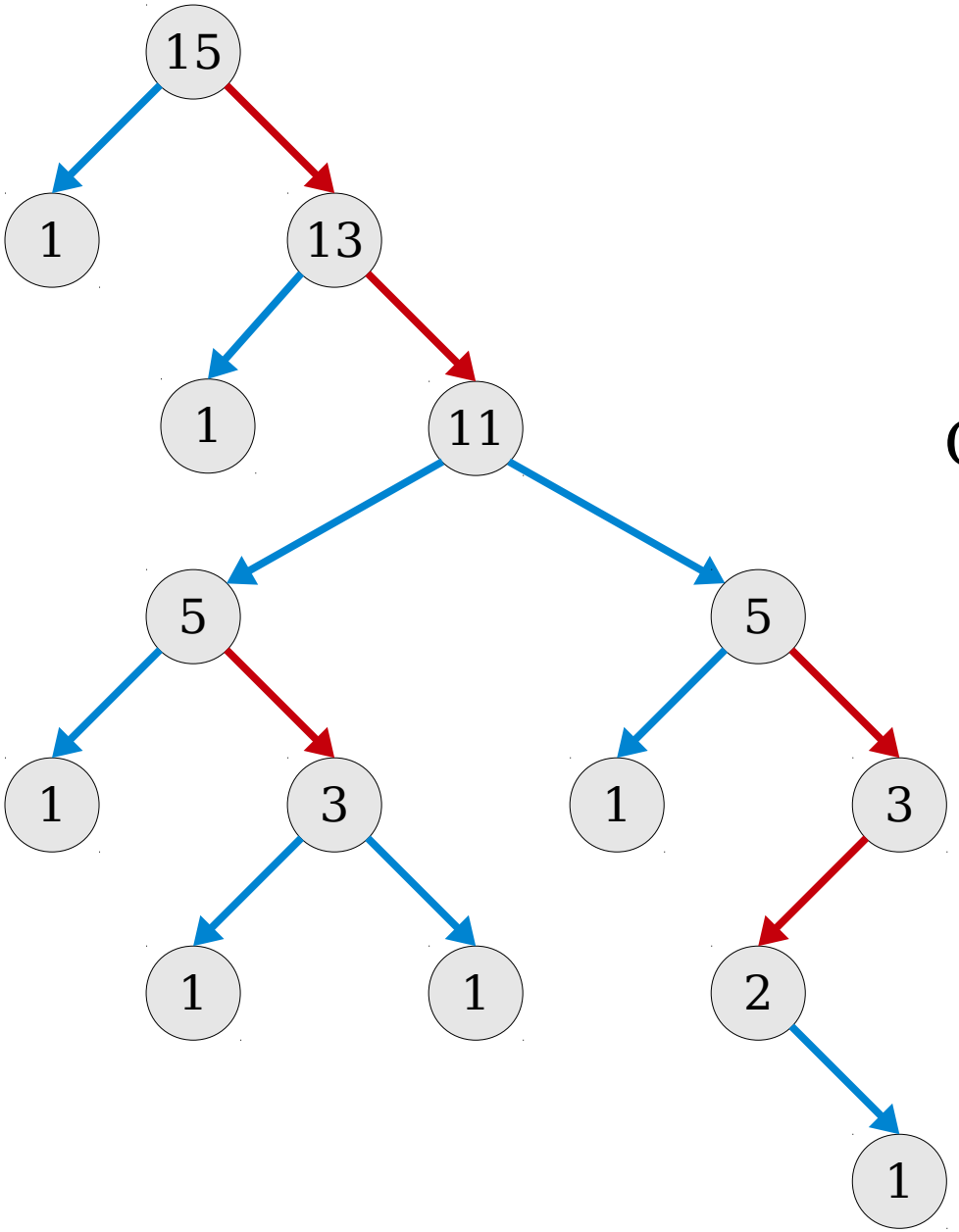
→ $s_{new} > \frac{1}{2} s_{old}$

→ $s_{new} \leq \frac{1}{2} s_{old}$

Cost of looking up x_i:

O(**log (W / w_i)** + **#red-used**)

This technique of splitting edges into "good" edges and "bad" edges is a technique called a ***heavy/light decomposition*** and is used extensively in the design and analysis of data structures.

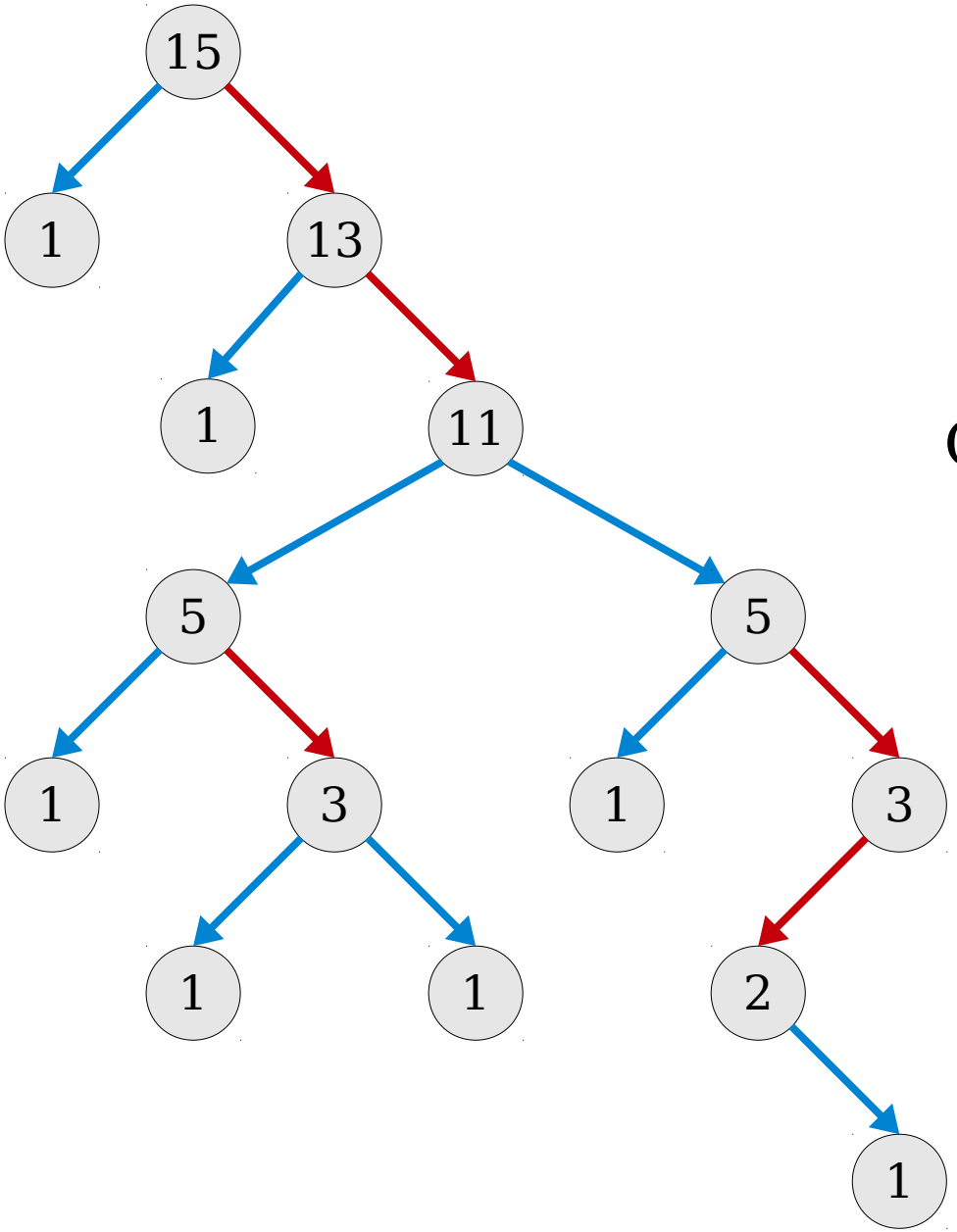


$s_{new} > \frac{1}{2} s_{old}$

$s_{new} \leq \frac{1}{2} s_{old}$

Cost of looking up x_i:

O(**log (W / w_i)** + **#red-used**)

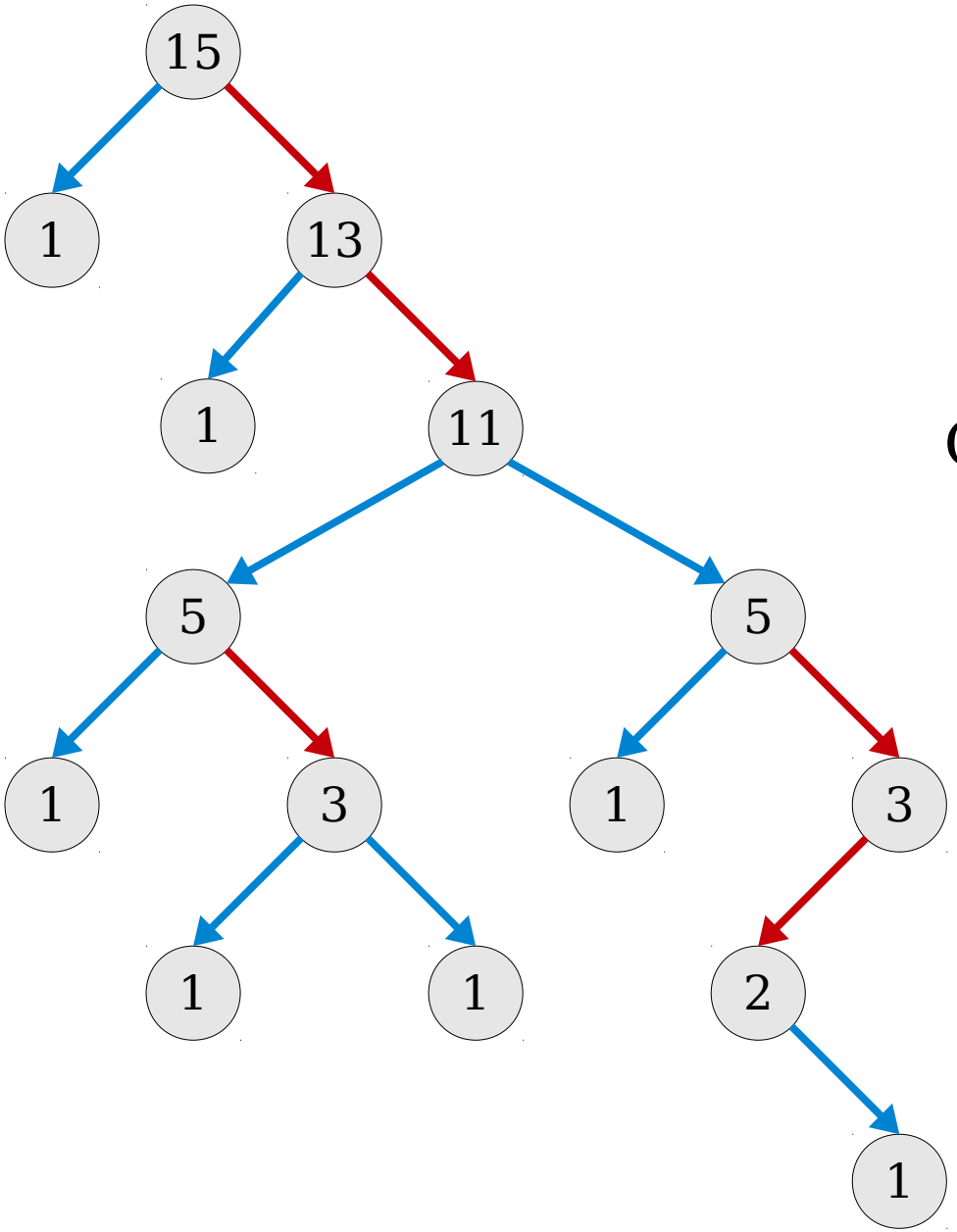


→ $\lg(s_{old} / s_{new}) < 1$

→ $\lg(s_{old} / s_{new}) \geq 1$

Cost of looking up x_i:

O(**log (W / w_i)** + **#red-used**)

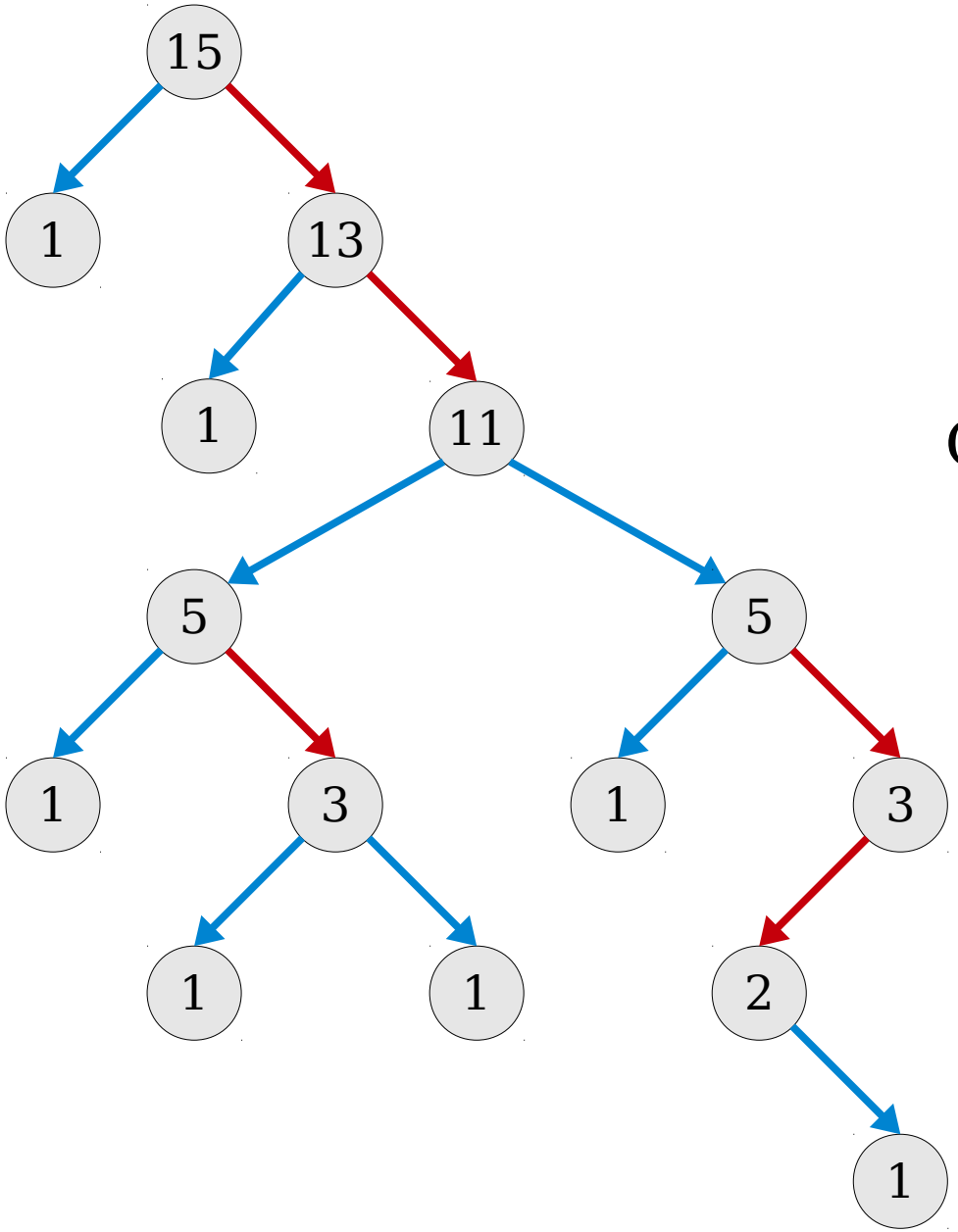


$\lg(s_{old}) - \lg(s_{new}) < 1$

$\lg(s_{old}) - \lg(s_{new}) \geq 1$

Cost of looking up x_i:

O(**log (W / w_i)** + **#red-used**)



→ $\lg(s_{old}) - \lg(s_{new}) < 1$

→ $\lg(s_{old}) - \lg(s_{new}) \geq 1$

Cost of looking up x_i:

O(**log (W / w_i)** + **#red-used**)

We need a potential function Φ that looks at $\lg s_i$ for each key x_i.

Reasonable guess: $\Phi = \sum_{i=1}^{n} \lg s_i$.

The Net Result

- ***Theorem:*** Using the potential function from before, the amortized cost of splaying key x_i is

  $$\mathbf{1 + 3 \lg (W / w_i)},$$

  where $W = w_1 + w_2 + \ldots + w_n$.
- The math behind this theorem is nontrivial and not at all interesting. It's just hard math gymnastics. Check Sleator and Tarjan's paper for details!
- This theorem holds for *any* choice of weights w_i assigned to the nodes, so it's useful for proving a number of nice results about splay trees.
- There's a subtle catch, though...

An Important Detail

- ***Recall:*** When using the potential method, the sum of the amortized costs relates to the sum of the real costs as follows:

$$\sum_{i=1}^{m} a(op_i) \;=\; \sum_{i=1}^{m} t(op_i) \;+\; \mathrm{O}(1)\cdot(\Phi_{m+1} - \Phi_1)$$

- Therefore:

$$\sum_{i=1}^{m} a(op_i) \;+\; \mathrm{O}(1)\cdot(\Phi_1 - \Phi_{m+1}) \;=\; \sum_{i=1}^{m} t(op_i)$$

- The actual cost is bounded by the sum of the amortized costs, ***plus the drop in potential***.

An Important Detail

- Previously, when we've analyzed amortized-efficient data structures, our potential function started at 0 and ended nonnegative.
- With our current choice of

$$\Phi = \sum_{i=1}^{n} \lg s_i$$

  if we're given a tree and then start splaying, our initial potential is nonzero, and our final potential might be lower than initial.
- This means that if we perform m operations on an n-element splay tree, we need to factor $\Phi_1 - \Phi_{m+1}$ into the total cost.

Analyzing Splay Trees

- To analyze the cost of splay tree operations, we'll proceed in three steps:
  - First, assign the weights to the nodes in a way that correlates weights and access patterns.
  - Second, use the amortized cost from before to determine the cost of each splay.
  - Finally, factor in the potential drop to account for the “startup cost.”
- The net result can be used to bound the cost of splaying.

Theorem (Balance Theorem): The cost of performing m operations on an n-node splay tree is $O(m \log n + n \log n)$.

Theorem (Balance Theorem): The cost of performing m operations on an n-node splay tree is $O(m \log n + n \log n)$.

Proof: The runtime of each operation is bounded by the cost of O(1) splays, so we'll bound the overall runtime by bounding the costs of the splays involved.

Theorem (Balance Theorem): The cost of performing m operations on an n-node splay tree is $O(m \log n + n \log n)$.

Proof: The runtime of each operation is bounded by the cost of O(1) splays, so we'll bound the overall runtime by bounding the costs of the splays involved.

Assign each key x_i weight $w_i = 1$.

Theorem (Balance Theorem): The cost of performing m operations on an n-node splay tree is $O(m \log n + n \log n)$.

Proof: The runtime of each operation is bounded by the cost of O(1) splays, so we'll bound the overall runtime by bounding the costs of the splays involved.

Assign each key x_i weight $w_i = 1$. The amortized cost of performing a splay at a key x_i is then

$$1 + 3 \lg (W / w_i)$$

Theorem (Balance Theorem): The cost of performing m operations on an n-node splay tree is $O(m \log n + n \log n)$.

Proof: The runtime of each operation is bounded by the cost of O(1) splays, so we'll bound the overall runtime by bounding the costs of the splays involved.

Assign each key x_i weight $w_i = 1$. The amortized cost of performing a splay at a key x_i is then

$$\begin{aligned} & 1 + 3 \lg (W / w_i) \\ = \; & 1 + 3 \lg (n / 1) \end{aligned}$$

Theorem (Balance Theorem): The cost of performing m operations on an n-node splay tree is $O(m \log n + n \log n)$.

Proof: The runtime of each operation is bounded by the cost of O(1) splays, so we'll bound the overall runtime by bounding the costs of the splays involved.

Assign each key x_i weight $w_i = 1$. The amortized cost of performing a splay at a key x_i is then

$$\begin{aligned} & 1 + 3 \lg (W / w_i) \\ = \; & 1 + 3 \lg (n / 1) \\ = \; & O(\log n). \end{aligned}$$

Theorem (Balance Theorem): The cost of performing m operations on an n-node splay tree is $O(m \log n + n \log n)$.

Proof: The runtime of each operation is bounded by the cost of O(1) splays, so we'll bound the overall runtime by bounding the costs of the splays involved.

Assign each key x_i weight $w_i = 1$. The amortized cost of performing a splay at a key x_i is then

$$\begin{aligned} & 1 + 3 \lg (W / w_i) \\ = \; & 1 + 3 \lg (n / 1) \\ = \; & O(\log n). \end{aligned}$$

To bound the total drop in potential, note that in the worst case any key x_i may initially be in a subtree of total weight n and end in a subtree of total weight 1.

Theorem (Balance Theorem): The cost of performing m operations on an n-node splay tree is $O(m \log n + n \log n)$.

Proof: The runtime of each operation is bounded by the cost of O(1) splays, so we'll bound the overall runtime by bounding the costs of the splays involved.

Assign each key x_i weight $w_i = 1$. The amortized cost of performing a splay at a key x_i is then

$$\begin{aligned} & 1 + 3 \lg (W / w_i) \\ = \; & 1 + 3 \lg (n / 1) \\ = \; & O(\log n). \end{aligned}$$

To bound the total drop in potential, note that in the worst case any key x_i may initially be in a subtree of total weight n and end in a subtree of total weight 1. Therefore, the maximum possible potential drop is

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

Theorem (Balance Theorem): The cost of performing m operations on an n-node splay tree is $O(m \log n + n \log n)$.

Proof: The runtime of each operation is bounded by the cost of O(1) splays, so we'll bound the overall runtime by bounding the costs of the splays involved.

Assign each key x_i weight $w_i = 1$. The amortized cost of performing a splay at a key x_i is then

$$\begin{aligned} & 1 + 3 \lg (W / w_i) \\ = \; & 1 + 3 \lg (n / 1) \\ = \; & O(\log n). \end{aligned}$$

To bound the total drop in potential, note that in the worst case any key x_i may initially be in a subtree of total weight n and end in a subtree of total weight 1. Therefore, the maximum possible potential drop is

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

So the total cost of performing m operations on an n-node splay tree is $O(m \log n + n \log n)$, as required. ■

A Stronger Result

- ***Recall:*** A statically optimal binary search tree has expected lookup cost $\Theta(1 + H)$, where H is the Shannon entropy of the access probability distribution.
- ***Claim:*** In a sense, splay trees achieve this statically optimal bound.

Static Optimality Theorem: Let $S = \{ x_1, \ldots, x_n \}$ be a set of keys stored in a splay tree. Suppose a series of lookups is performed where

- every node is accessed at least once, and
- all lookups are successful.

Then the amortized cost of each access is $O(1 + H)$, where H is the Shannon entropy of the access distribution.

Proof:

Proof: Assign each key x_i $w$eight p_i, where p_i is the fraction of the lookups that access x_i.

Proof: Assign each key x_i weight p_i, where p_i is the fraction of the lookups that access x_i. The amortized cost of a lookup of x_i is therefore at most

$$1 + 3 \lg (W / w_i) = 1 + 3 \lg (1 / p_i) = 1 - 3 \lg p_i.$$

Proof: Assign each key x_i weight p_i, where p_i is the fraction of the lookups that access x_i. The amortized cost of a lookup of x_i is therefore at most

$$1 + 3 \lg (W / w_i) = 1 + 3 \lg (1 / p_i) = 1 - 3 \lg p_i.$$

Since each element is accessed mp_i times, the sum of the amortized lookup times is given by

$$\sum_{i=1}^{n} \left(m p_i (1 - 3 \lg p_i) \right) = m \sum_{i=1}^{n} \left(p_i - 3 p_i \lg p_i \right) = m + 3mH.$$

Proof: Assign each key x_i weight p_i, where p_i is the fraction of the lookups that access x_i. The amortized cost of a lookup of x_i is therefore at most

$$1 + 3 \lg (W / w_i) = 1 + 3 \lg (1 / p_i) = 1 - 3 \lg p_i.$$

Since each element is accessed mp_i times, the sum of the amortized lookup times is given by

$$\sum_{i=1}^{n} \left(m p_i (1 - 3 \lg p_i) \right) = m \sum_{i=1}^{n} \left(p_i - 3 p_i \lg p_i \right) = m + 3mH.$$

To bound the total drop in potential, notice that each node contributes $\lg s_i$ to the potential, where s_i is the weight of the subtree rooted at s_i.

Proof: Assign each key x_i weight p_i, where p_i is the fraction of the lookups that access x_i. The amortized cost of a lookup of x_i is therefore at most

$$1 + 3 \lg (W / w_i) = 1 + 3 \lg (1 / p_i) = 1 - 3 \lg p_i.$$

Since each element is accessed mp_i times, the sum of the amortized lookup times is given by

$$\sum_{i=1}^{n} \left(m p_i (1 - 3 \lg p_i) \right) = m \sum_{i=1}^{n} \left(p_i - 3 p_i \lg p_i \right) = m + 3mH.$$

To bound the total drop in potential, notice that each node contributes $\lg s_i$ to the potential, where s_i is the weight of the subtree rooted at s_i. The maximum value of s_i is 1 (when all nodes are in x_i's tree) and the minimum value of s_i is p_i (when x_i is by itself), so the maximum possible potential drop from a single element is given by $-\lg p_i$.

Proof: Assign each key x_i weight p_i, where p_i is the fraction of the lookups that access x_i. The amortized cost of a lookup of x_i is therefore at most

$$1 + 3 \lg (W / w_i) = 1 + 3 \lg (1 / p_i) = 1 - 3 \lg p_i.$$

Since each element is accessed mp_i times, the sum of the amortized lookup times is given by

$$\sum_{i=1}^{n} \left(m p_i (1 - 3 \lg p_i) \right) = m \sum_{i=1}^{n} \left(p_i - 3 p_i \lg p_i \right) = m + 3mH.$$

To bound the total drop in potential, notice that each node contributes $\lg s_i$ to the potential, where s_i is the weight of the subtree rooted at s_i. The maximum value of s_i is 1 (when all nodes are in x_i's tree) and the minimum value of s_i is p_i (when x_i is by itself), so the maximum possible potential drop from a single element is given by $-\lg p_i$. Therefore, the maximum potential drop is

$$\sum_{i=1}^{n} -\lg p_i \leq \sum_{i=1}^{n} -m p_i \lg p_i = m \sum_{i=1}^{n} -p_i \lg p_i = mH$$

Proof: Assign each key x_i weight p_i, where p_i is the fraction of the lookups that access x_i. The amortized cost of a lookup of x_i is therefore at most

$$1 + 3 \lg (W / w_i) = 1 + 3 \lg (1 / p_i) = 1 - 3 \lg p_i.$$

Since each element is accessed mp_i times, the sum of the amortized lookup times is given by

$$\sum_{i=1}^{n} \left(m p_i (1 - 3 \lg p_i) \right) = m \sum_{i=1}^{n} \left(p_i - 3 p_i \lg p_i \right) = m + 3mH.$$

To bound the total drop in potential, notice that each node contributes $\lg s_i$ to the potential, where s_i is the weight of the subtree rooted at s_i. The maximum value of s_i is 1 (when all nodes are in x_i's tree) and the minimum value of s_i is p_i (when x_i is by itself), so the maximum possible potential drop from a single element is given by $-\lg p_i$. Therefore, the maximum potential drop is

$$\sum_{i=1}^{n} -\lg p_i \leq \sum_{i=1}^{n} -m p_i \lg p_i = m \sum_{i=1}^{n} -p_i \lg p_i = mH$$

So the cost of the m lookups is $O(m + mH)$, and since there are m lookups, the amortized cost of each is $O(1 + H)$. ■

Beating Static Optimality

- On many classes of access sequences, splay trees can outperform statically optimal BSTs.
- The ***sequential access theorem*** says that

  If you look up all n elements in a splay tree in ascending order, the amortized cost of each lookup is O(1).

- The ***working-set theorem*** says that

  If you perform $\Omega(n \log n)$ successful lookups, the amortized cost of each successful lookup is $O(1 + \log t)$, where t is the number of searches since we last looked up the element searched for.

- In the upcoming programming assignment, you'll compare the performance of splay trees to (nearly) optimal BSTs. See if you notice anything interesting in these cases!

An Open Problem: ***Dynamic Optimality***

The BST Model

- Consider a BST with a pointer called the ***finger***, which points to some element, initially the root.
- You are allowed to perform the following operations at any time:
  - Move the finger to a left or right child.
  - Move the finger to a parent.
  - Rotate the node pointed at by the finger with its parent.
  - Return the node pointed at by the finger.
- Note that the splay tree fits into this model; the finger starts at a root, descends to a node, then splays back up to the top.

Dynamic Optimality

- ***Observation:*** For any set of keys and initial BST, given the lookup sequence in advance, there is (nonconstructively) some optimal series of tree operations we can perform to minimize the lookup time.
- For a series of lookups S, we'll say that $OPT(S)$ is the number of operations required in this sequence.
- ***Open Problem:*** Is there a single BST data structure whose cost on any sequence S is $O(OPT(S))$?
- Such a binary search tree would be called a ***dynamically optimal binary search tree*** and would be within a constant factor of the fastest possible BST on any possible sequence of lookups.

Competitive Ratios

- A binary search tree data structure is called ***f(n)-competitive*** if the ratio of the cost of performing a sequence of operations S on that data structure is at most $f(n) \cdot OPT(S)$.
- A dynamically-optimal BST would be an O(1)-competitive BST, for example.
- We don't know of any BSTs with this property yet. But we do know a few things!

What We Know

- ***Known:*** Data structures exist that are provably O(log log *n*)-competitive (*Tango trees, multisplay trees*).
- ***Conjectured:*** Splay trees are O(1)-competitive.
- ***Suspected:*** A recently-developed data structure (the *online greedy BST*) may be the first tree structure that will be proven to be dynamically optimal.
- Tango trees, multisplay trees, online greedy BSTs, and the associated proofs would make for really, really interesting final project topics!

Next Time

- ***Randomized Data Structures***
  - How do we trade worst-case guarantees for probabilistic guarantees?
- ***Count[-Min] Sketches***
  - Counting in sublinear space.
- ***Concentration Inequalities***
  - How do we show that we're near the expected value most of the time?