Splay Trees

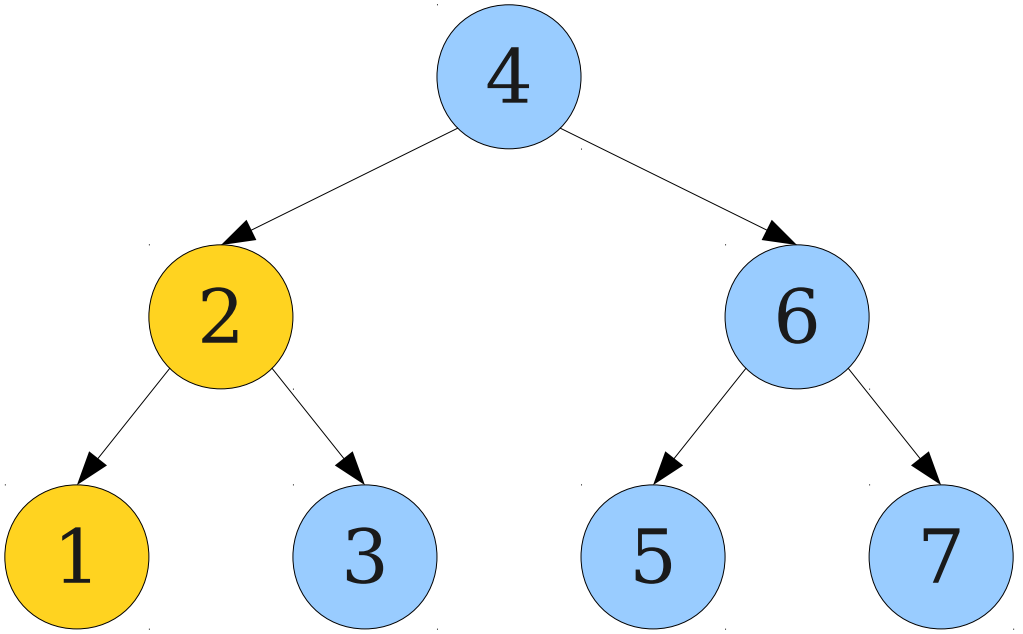
Outline for Today

- **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

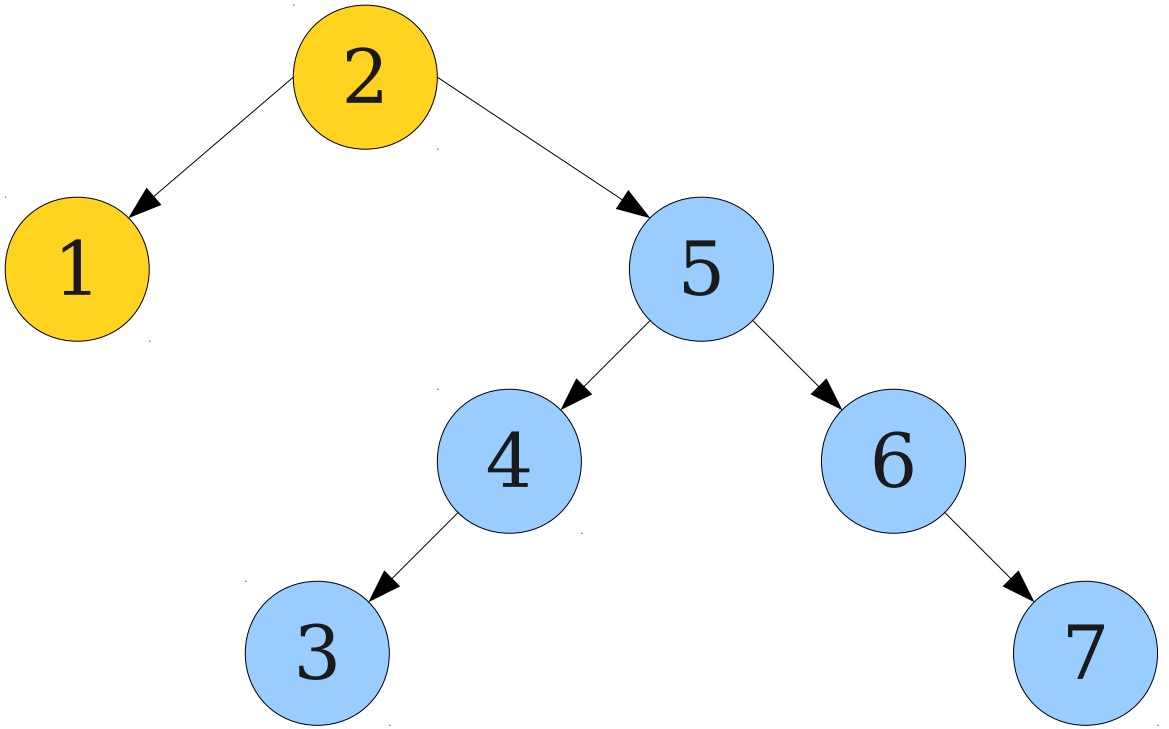
Balanced BSTs

- We've explored balanced BSTs this quarter because they guarantee worst-case O(log n) operations.
- **Claim:** Depending on the access sequence, balanced BSTs may not be optimal BSTs.



Balanced BSTs

- We've explored balanced BSTs this quarter because they guarantee worst-case O(log n) operations.
- **Claim:** Depending on the access sequence, balanced BSTs may not be optimal BSTs.



Static Optimality

- Let $S = \{ x_1, x_2, \ldots, x_n \}$ be a set with access probabilities $p_1, p_2, \ldots, p_n$.
- **Goal:** Construct a binary search tree T^* that minimizes the total expected access time.
- T^* is called a ***statically optimal binary search tree***.

Static Optimality

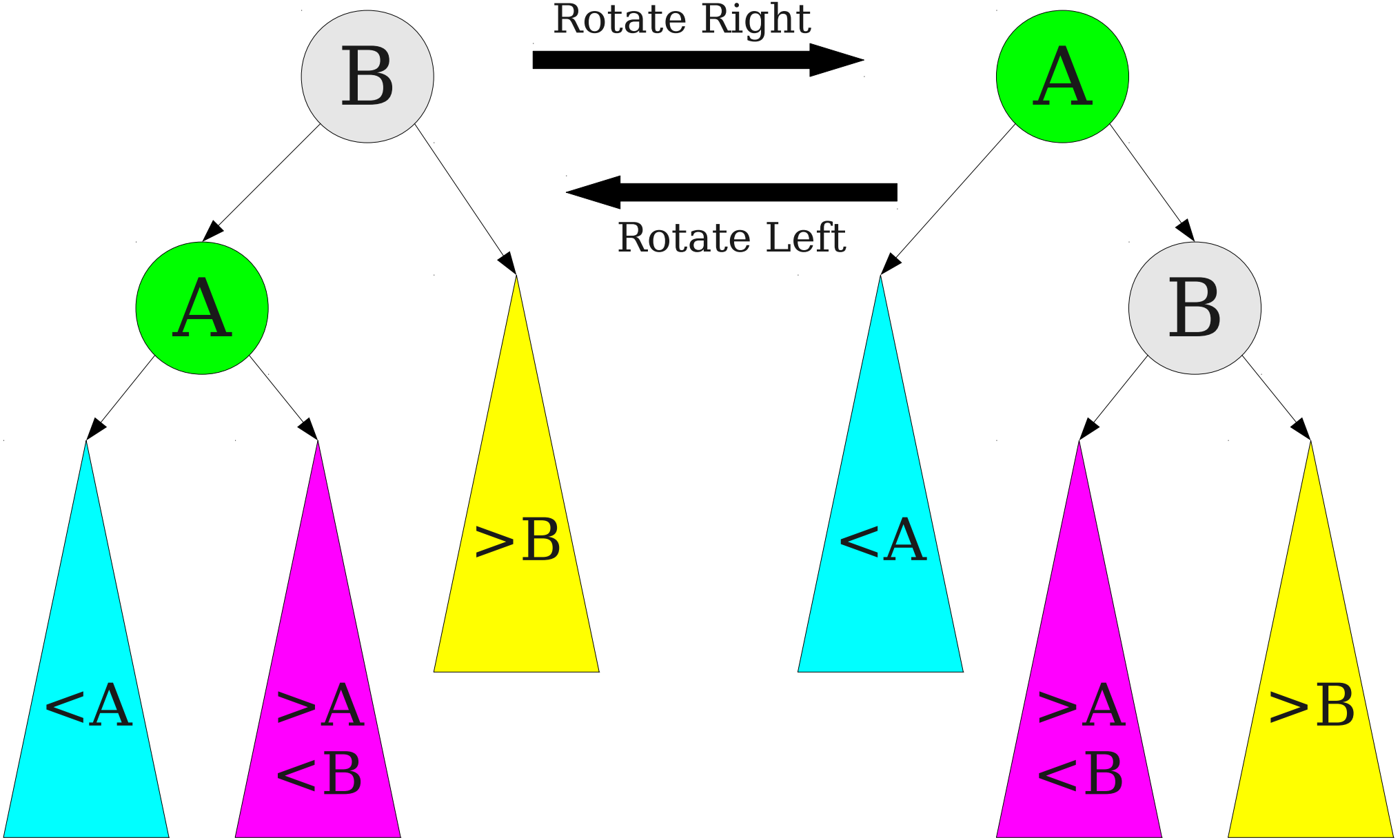
- There is an $O(n^2)$-time dynamic programming algorithm for constructing statically optimal binary search trees.
  - Knuth, 1971
- There is an $O(n \log n)$-time greedy algorithm for constructing binary search trees whose cost is within 1.5 of optimal.
  - Mehlhorn, 1975
- These algorithms assume that the access probabilities are known in advance.

Challenge: Can we construct an optimal BST without knowing the access probabilities in advance?

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.

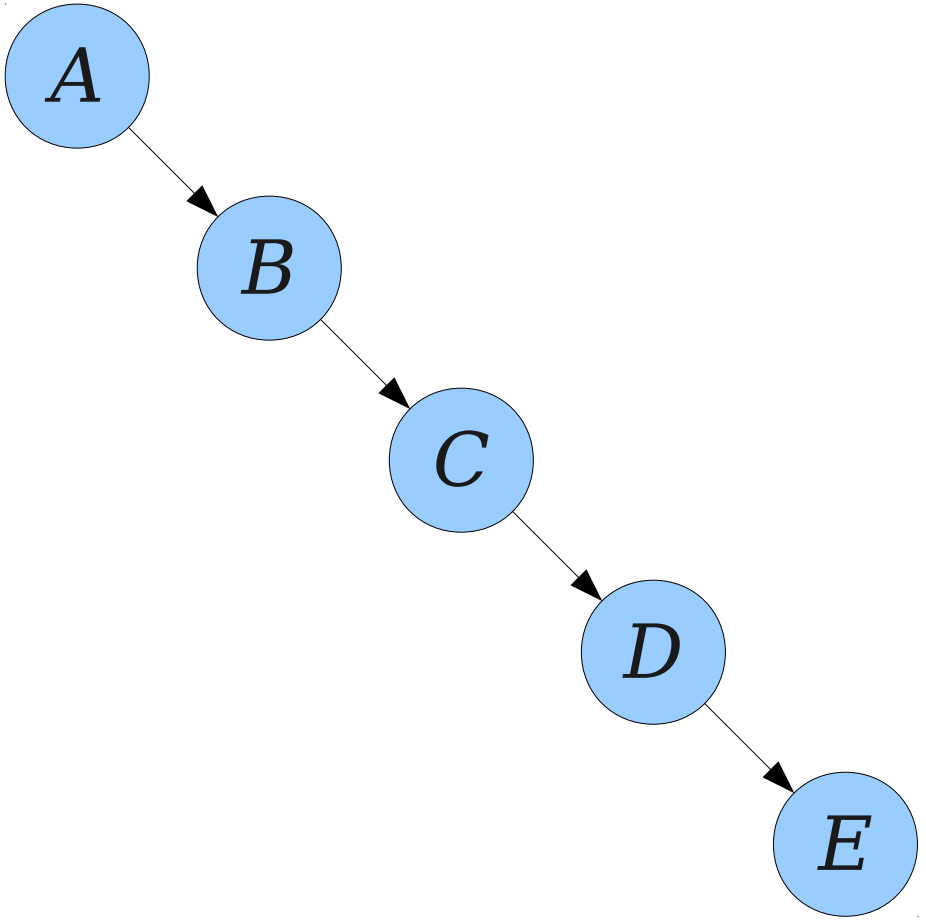
Refresher: Tree Rotations



An Initial 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



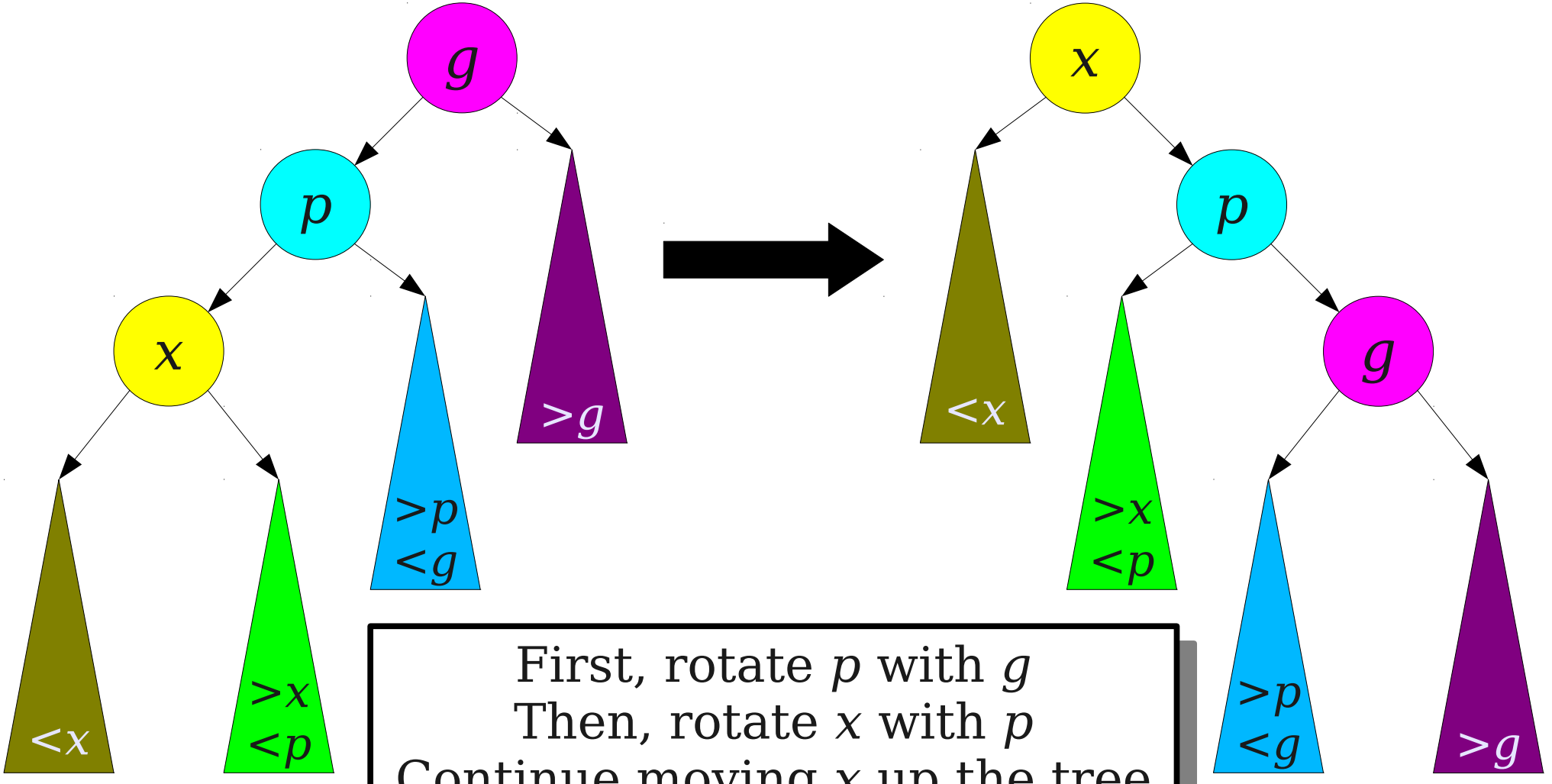
The Problem

- The “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

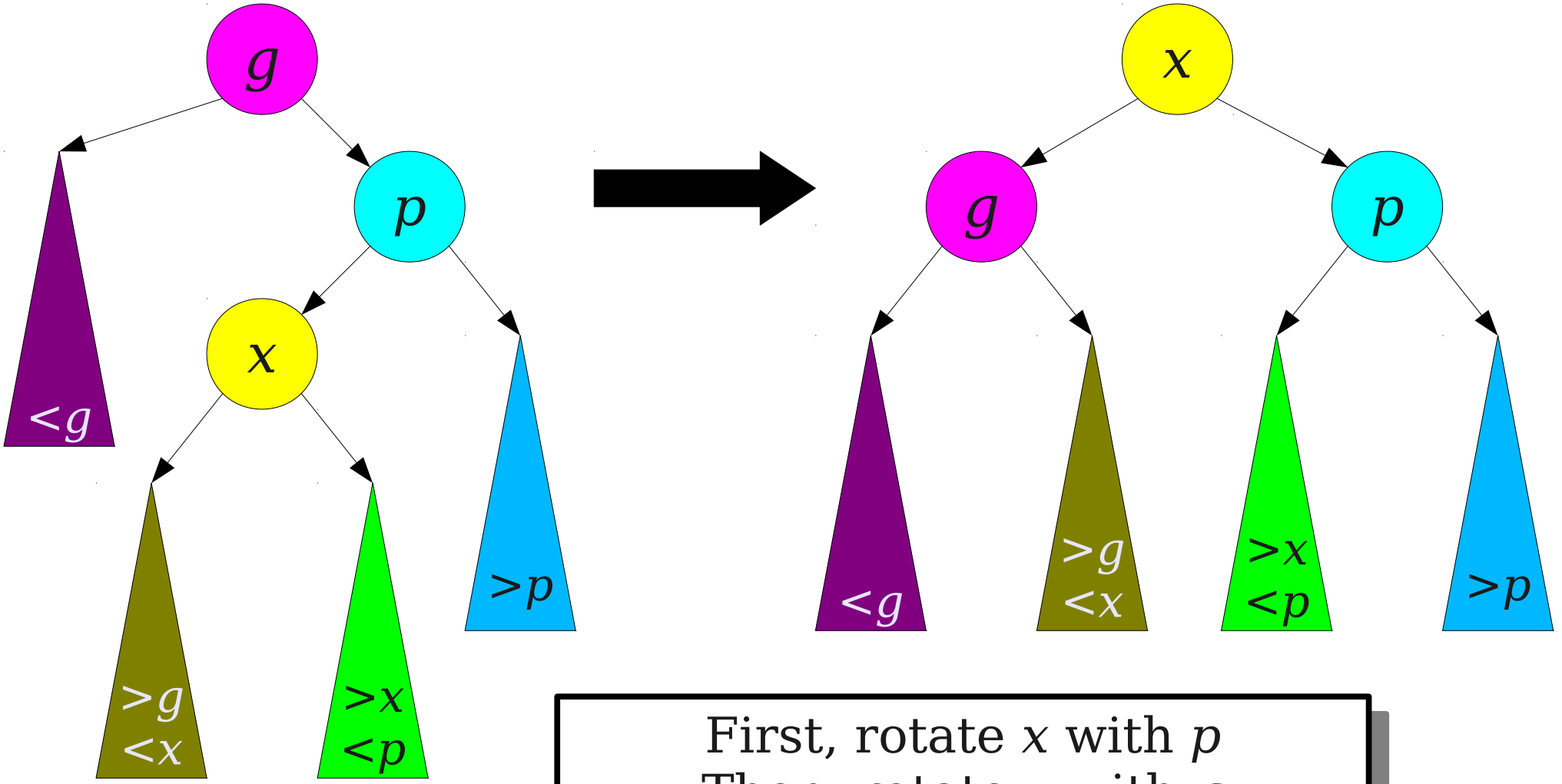
- In 1983, Daniel Sleator and Robert Tarjan invented an operation called ***splaying***.
- Rotates an element to the root of the tree, but does so in a way that's more “fair” to other nodes in the tree.
- There are three cases for splaying.

Case 1: Zig-Zig



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

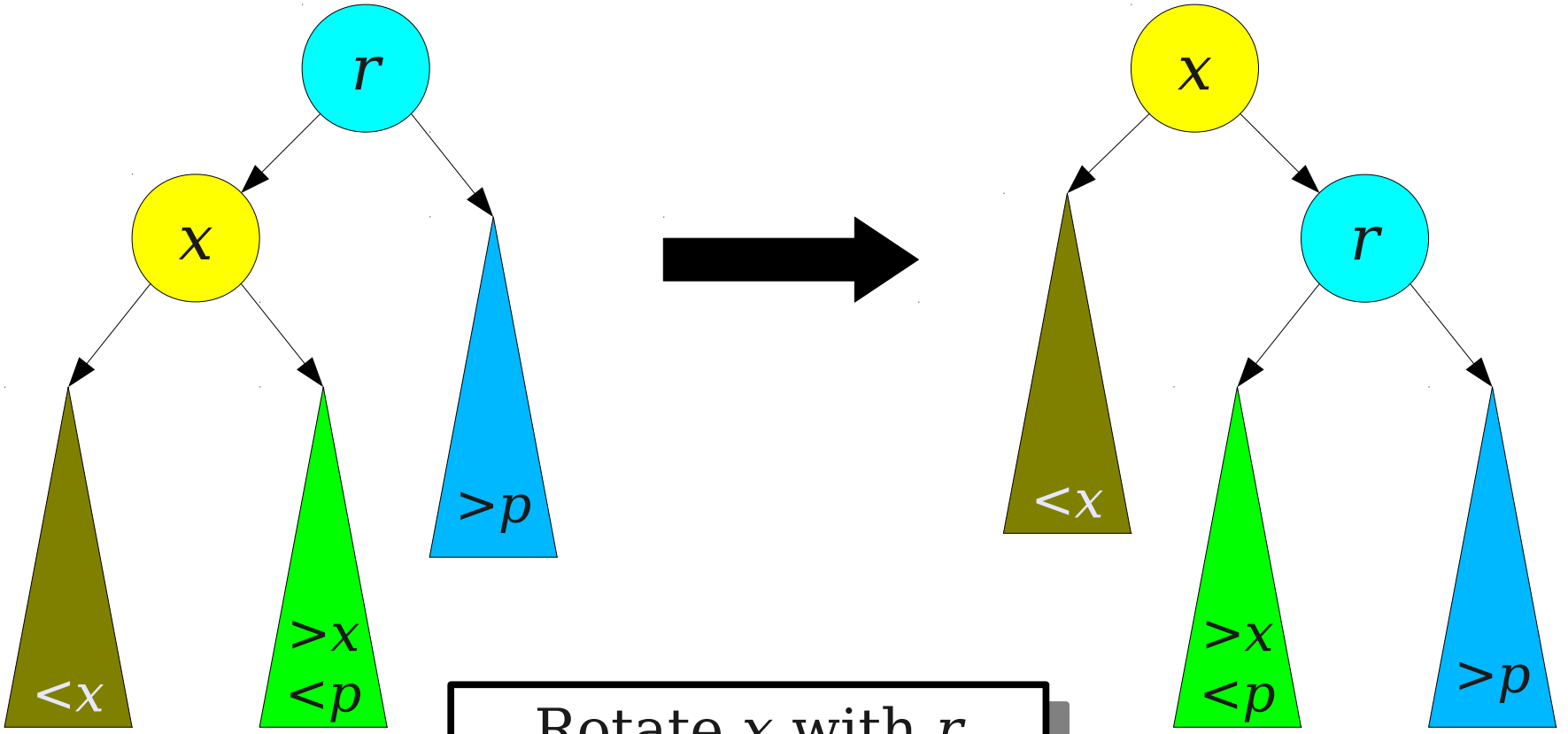
Case 2: Zig-Zag



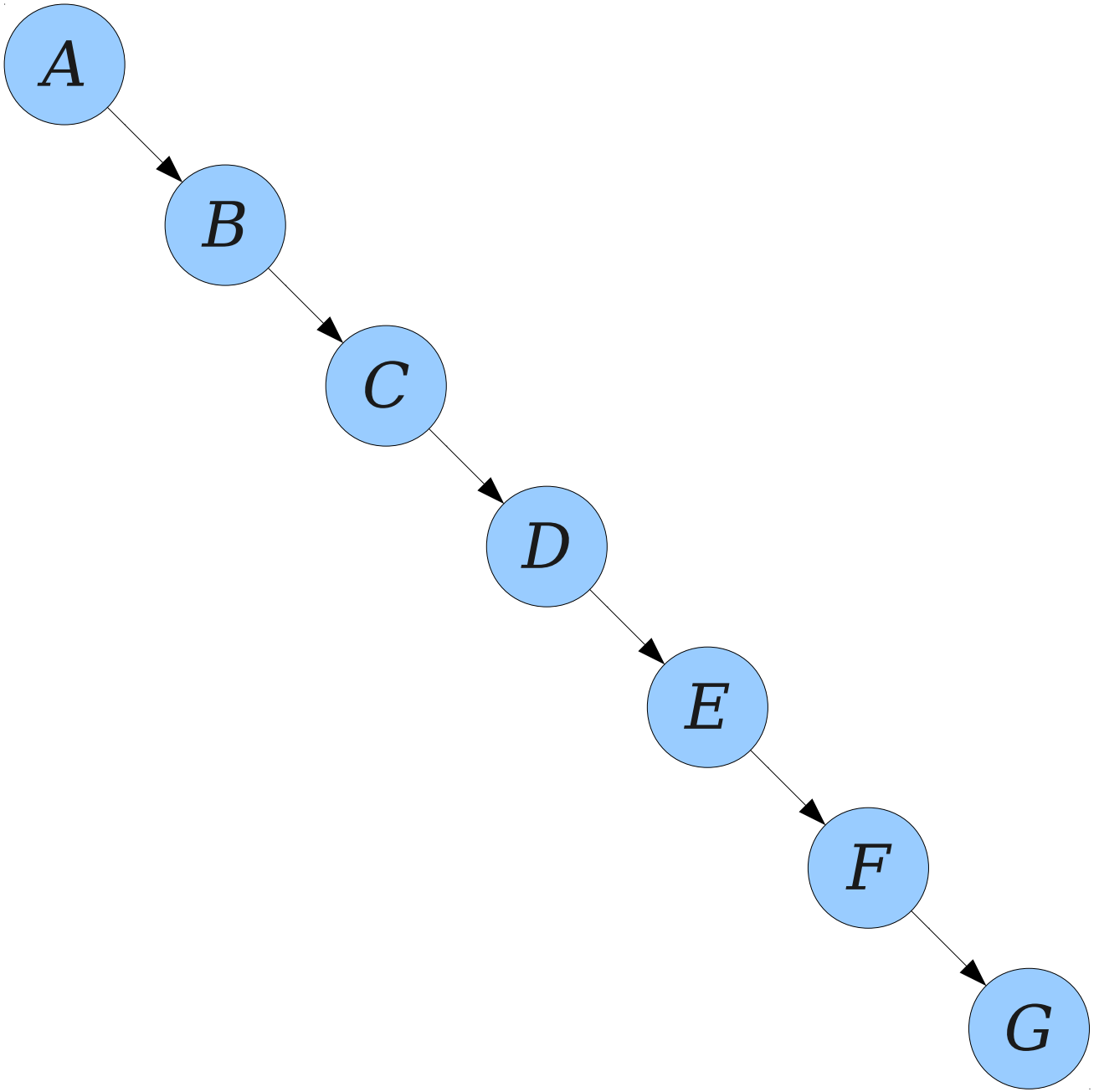
First, rotate x with p
Then, rotate x with g
Continue moving x up the tree

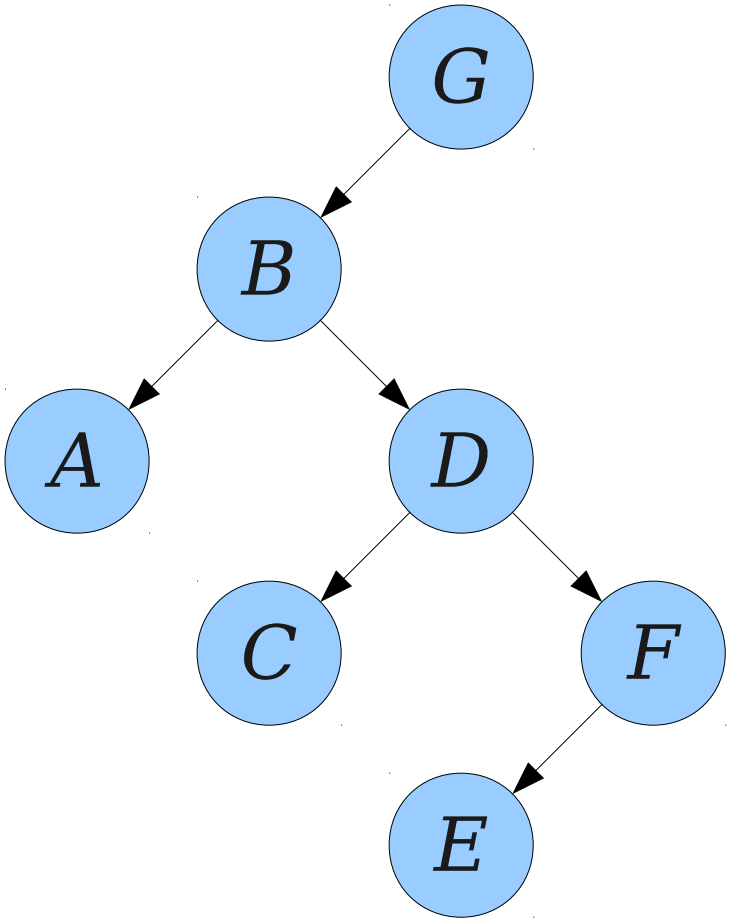
Case 3: Zig

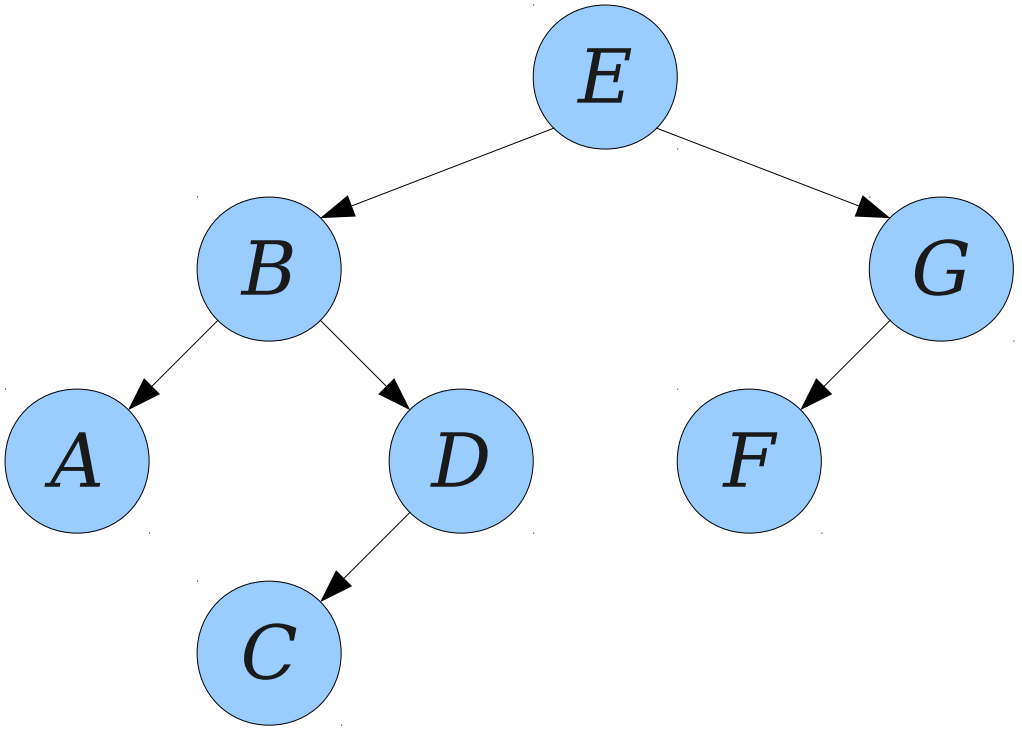
(Assume r is the tree root)



Rotate x with r
x is now the root.


A
B
C
D
E
F
G


G
B
A
D
C
F
E

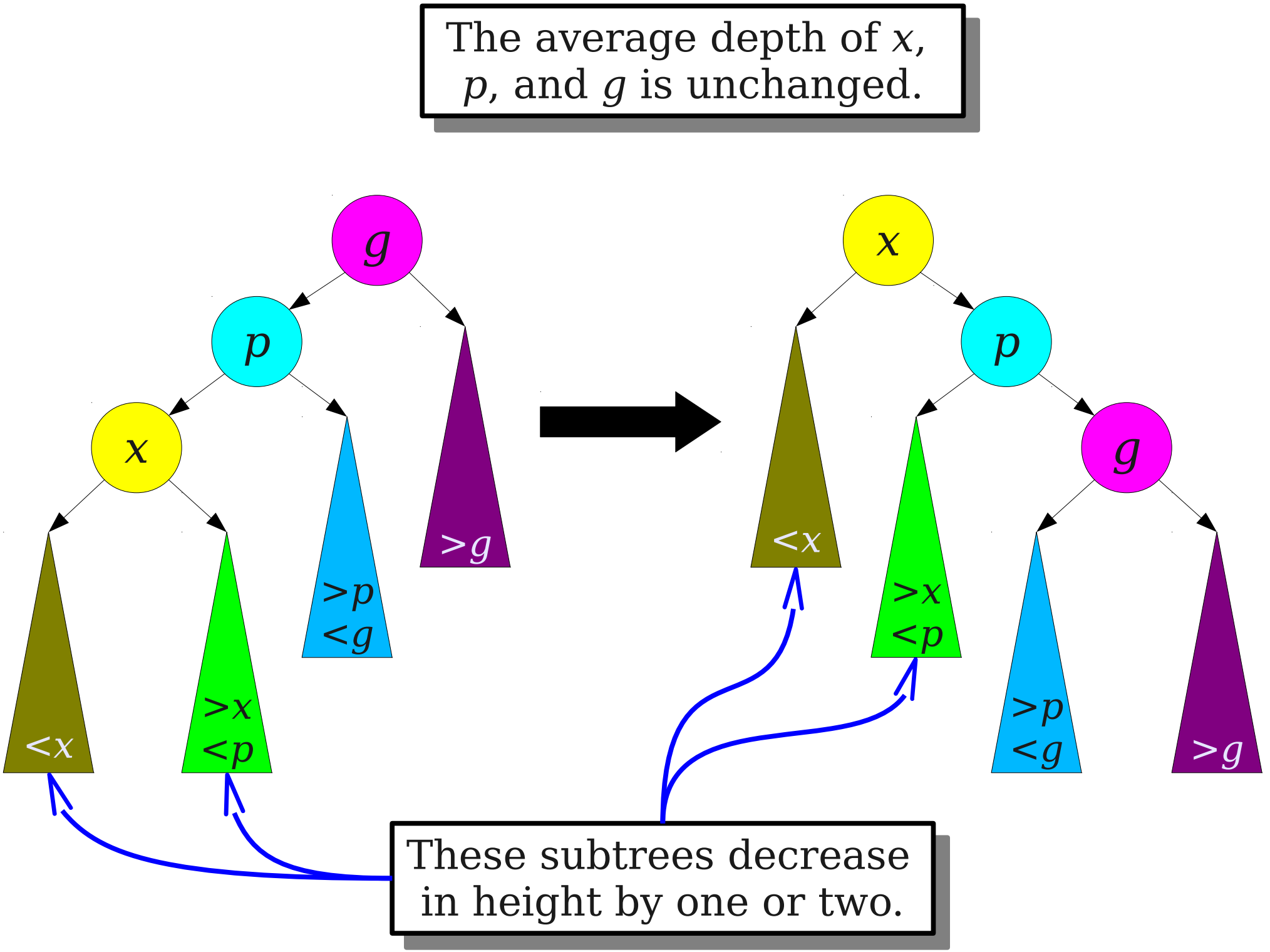

E
B
G
A
D
F
C

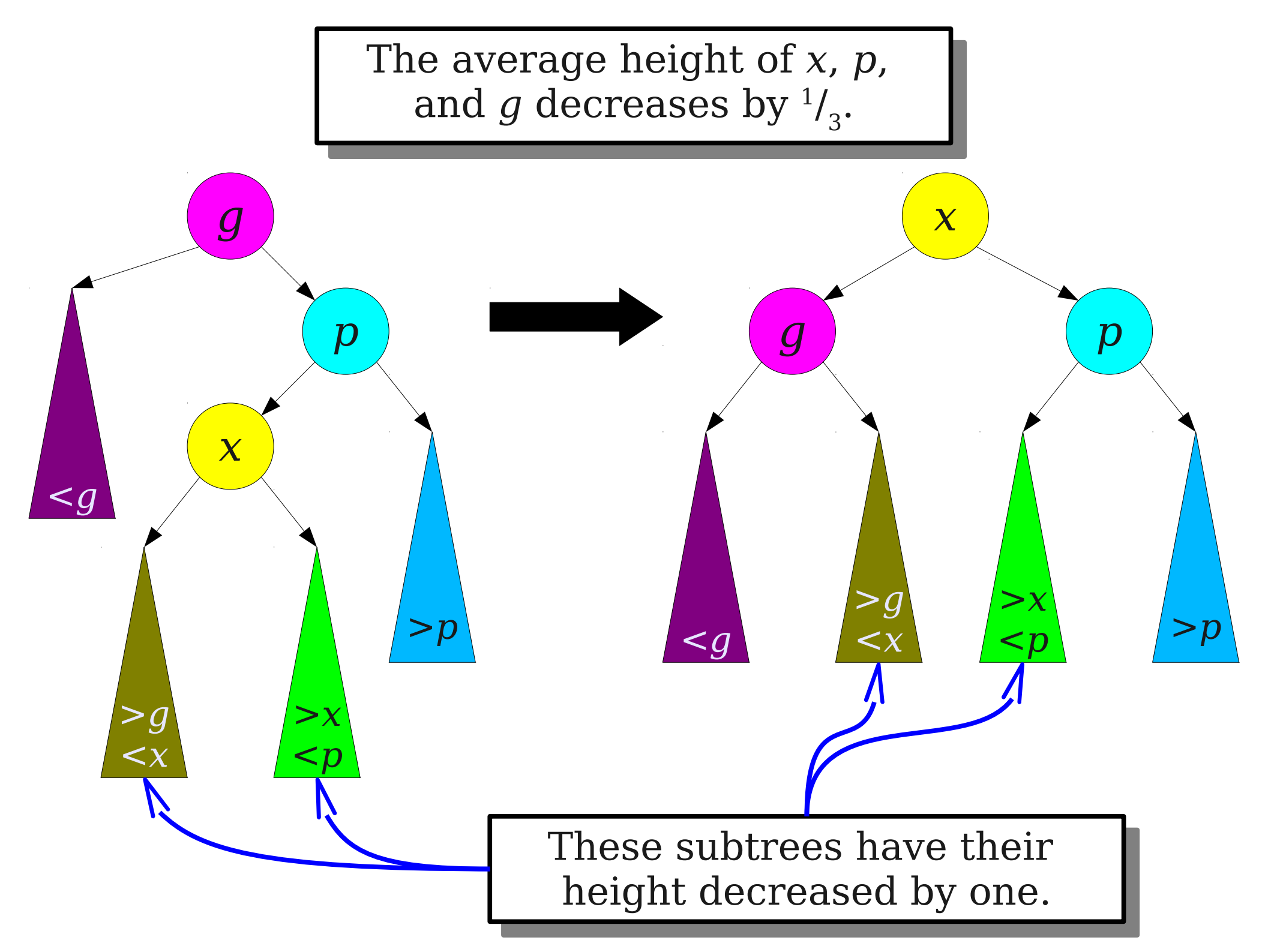
Splaying, Empirically

- 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?

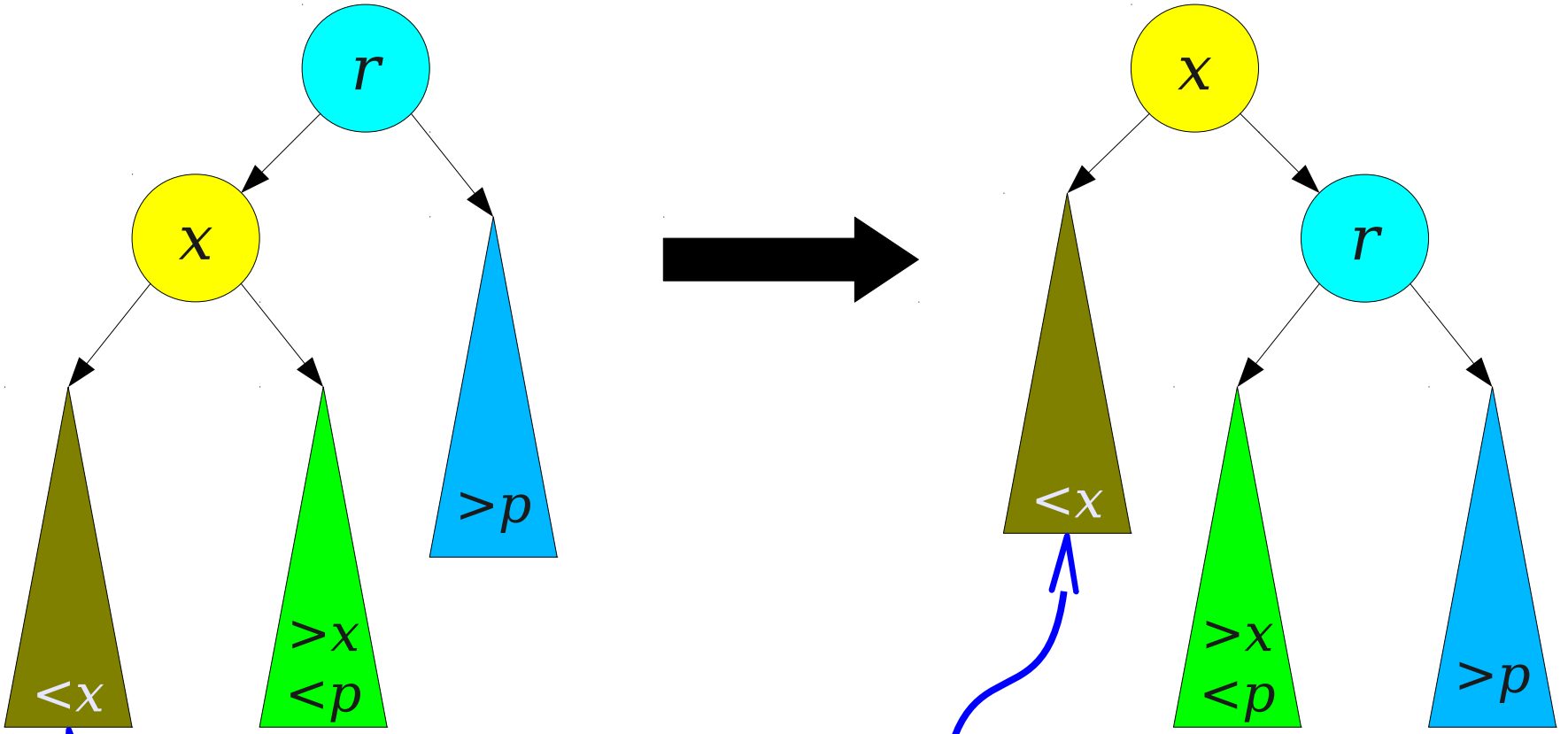
Why Splaying Works

- **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.



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.

Some Claims

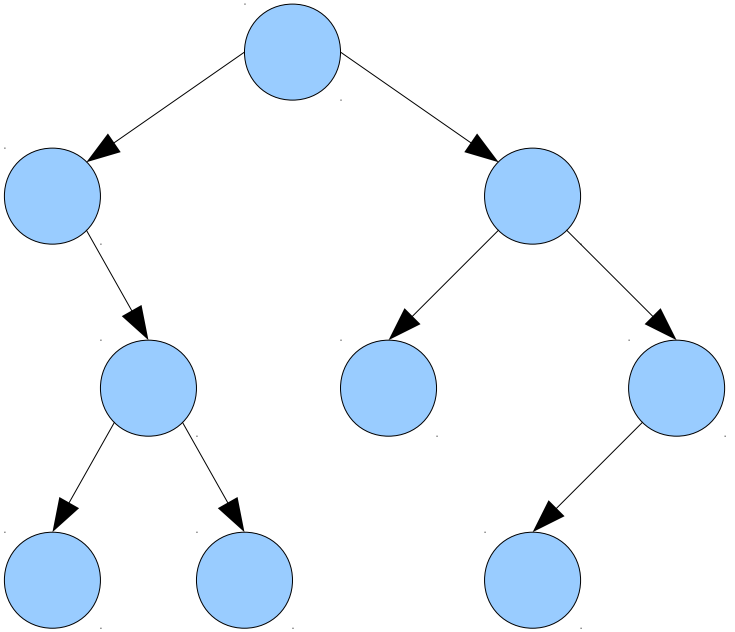
- **Claim 1:** The amortized cost of splaying a node up to the root is O(log n).
- **Claim 2:** The amortized cost of splaying a node up to the root can be o(log n) if the access pattern is non-uniform.
- We'll prove these results later today.

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.

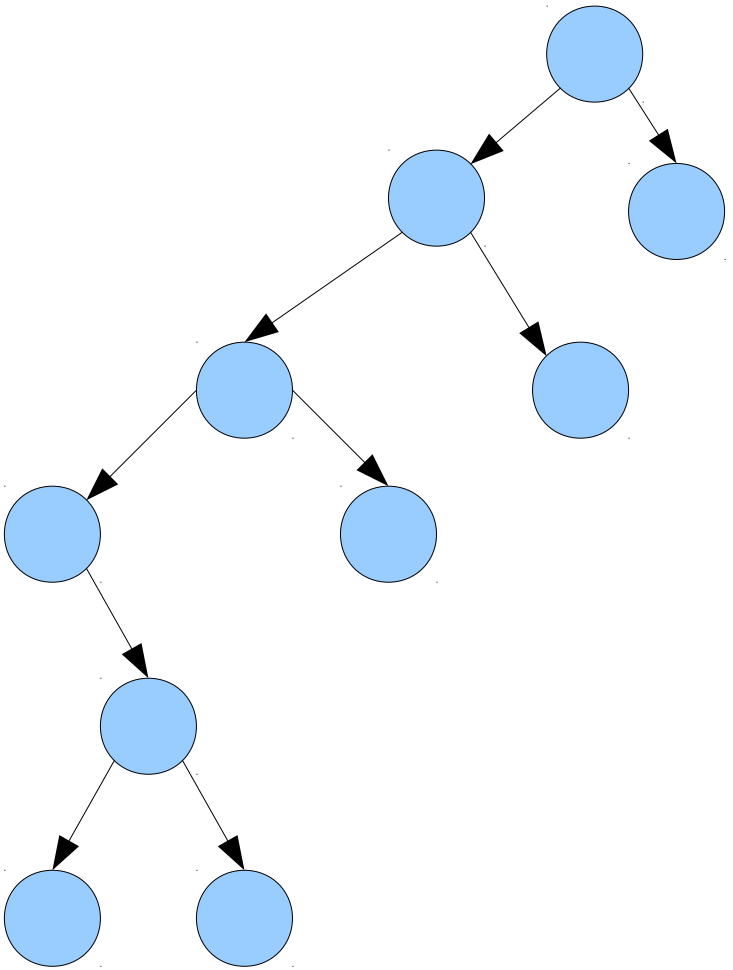
Lookups

- 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.



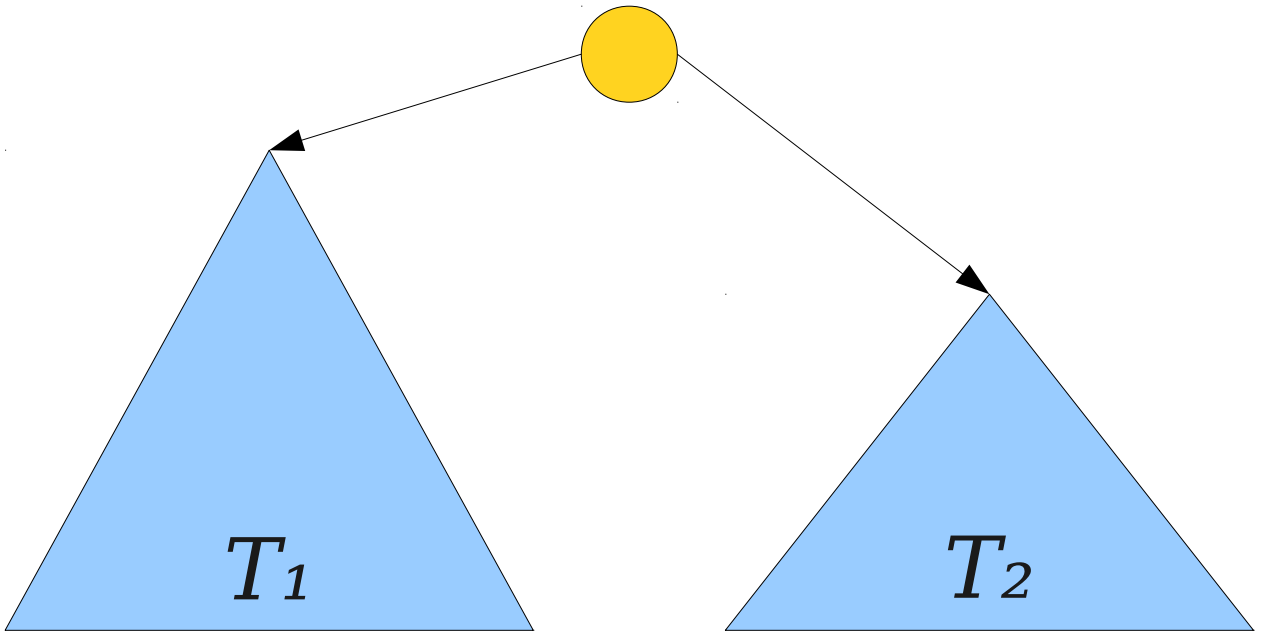
Insertions

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



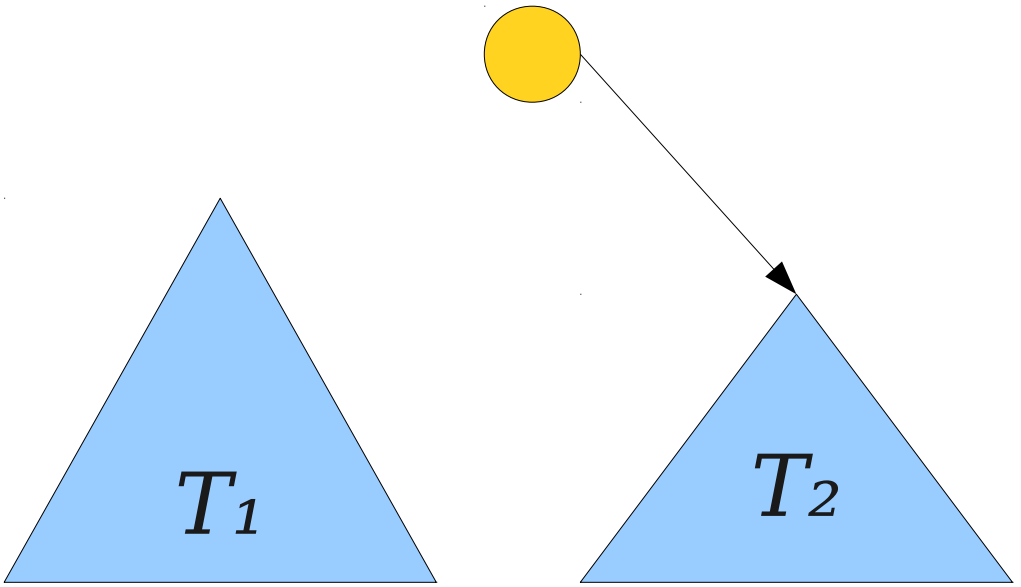
Join

- 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.



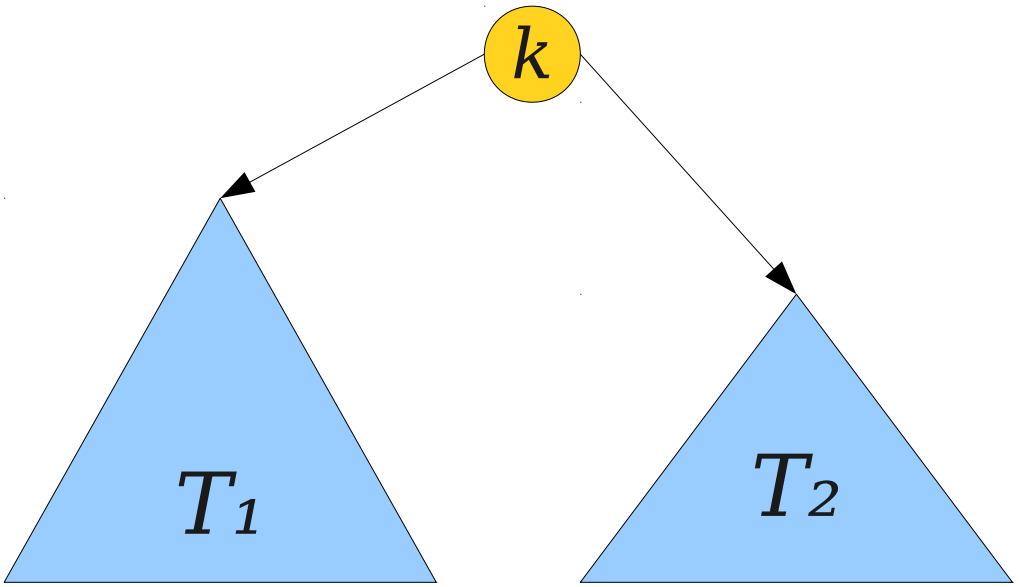
Split

- 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.



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.

The Tricky Part: Formalizing This

The Cost of a Splay

- We need to prove that splaying is amortized efficient.
- Historically, it has proven hard to analyze splay trees for several reasons:
  - Each lookup can significantly reshape the tree.
  - Deliberate lack of structure makes it hard to find invariants useful in the analysis.
- 30 years after splay trees were invented, we don't know the answer to the following questions:
  - What is the cost of performing n splits on a splay tree with no intervening operations?
  - What is the cost of performing n deque operations on a splay tree?

Time-Out for Your Questions!

“If computers don't work with binary numbers, but instead with decimal (base 10) numbers, would we be able to solve new questions and/or solve old questions more efficiently? Or would it be a big headache?”

You can actually use *any* base with only constant slowdown! See the paper

“Changing Base without Losing Space”

by Dodis et al for more details.

“Would you consider holding office hours on Friday, Saturday, or Sunday? By Monday, many of us will want to have already made substantial progress on the pset, but by Thursday many of us won't have had a chance to really think the problems over.”

I'll ask the TAs about this.

A note: we received a total of 12 emails last week with questions. *Please feel free to email us if you have questions!*

Back to CS166!

The Cost of a Splay

- Outline for the rest of today:
  - Provide an intuition for the potential function.
  - Perform an amortized analysis to justify the runtime of splay trees.
  - Use that analysis to derive powerful results about splay tree costs.
- Will not do the full analysis here; check the original paper for more details.

Weighted Path Lengths

- For any node x_i in a BST, define the ***path length*** of x_i, which we'll denote l_i, to be the length of the path from the root of the tree to x_i.
- Associate positive weight w_i with each node in the BST.
- The ***weighted path length*** of the BST is then

$$\sum_{i=1}^{n} w_i l_i$$

- If the weights are access probabilities, the expected cost of a lookup in the BST is

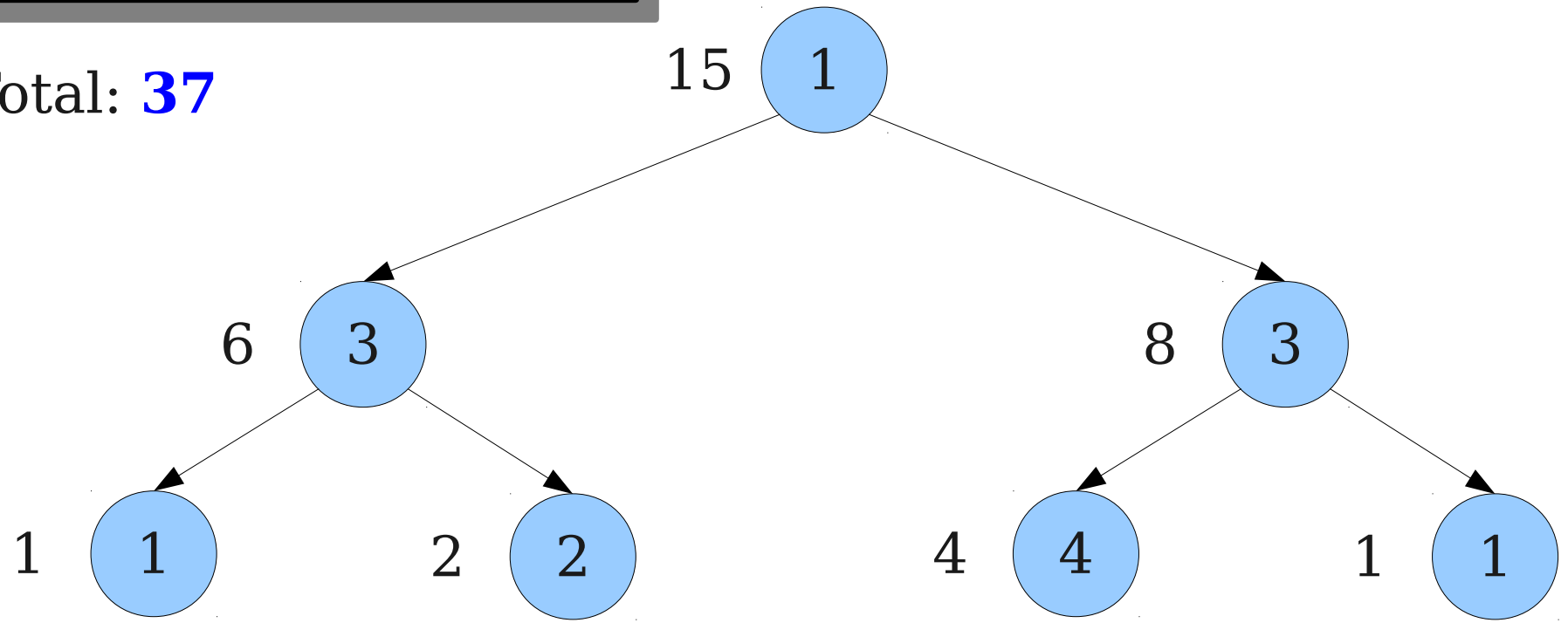
$$O\left(1 + \sum_{i=1}^{n} w_i l_i\right)$$

Intuiting Weighted Path Lengths

$$\sum_{i=1}^{n} w_i l_i$$

This is the weighted path length plus the sum of the weights.

Total: **37**



Node Sizes

- For each node x, define the ***size*** of x, denoted $s(x)$, to be the sum of the weights of x and all its descendants.
- The sum of the sizes of all the nodes gives the weighted path length of the tree plus the sum of the weights
- By trying to keep the sum of the node sizes low, we can try to keep the access costs in the tree low.

Node Ranks

- Although the sum of the node sizes gives a good sense for access costs in the tree, the actual analysis uses a different quantity.
- Define the ***rank*** of a node x, denoted $r(x)$, to be $\lg s(x)$.
- We will use the following potential function when analyzing splay trees:

$$\Phi = \sum_{i=1}^{n} r(x_i)$$

Some Observations

- Let $S = \{ x_1, \ldots, x_n \}$ be a set of keys in a BST and let $w_1, \ldots, w_n$ be the weights on those keys.
- **Claim 1:** The value of $s(t)$, where t is the tree root, is constant regardless of the tree shape.
  - It's the sum of all the weights in the trees.
- **Claim 2:** The value of $r(t)$, where t is the tree root, is constant regardless of the tree shape.
  - It's lg $s(t)$, which is the same regardless of the tree shape.

A Useful Mathematical Fact

Theorem: If $x > 0$, $y > 0$, and $x + y \leq 1$, then $\lg x + \lg y \leq -2$.

Proof: We will show that under these constraints, $\lg x + \lg y$ achieves a maximum value of -2.

Because $\lg x + \lg y = \lg xy$ and 2^k is increasing, we will equivalently maximize xy.

When x and y are nonnegative, increasing either x or y will increase xy. Thus the maximum value of xy, subject to our constraints, occurs when $x + y = 1$.

If $x + y = 1$, then $xy = x(1 - x)$. This is maximized when $x = ½$ and $y = ½$, at which point we have $\lg x + \lg y = \lg ½ + \lg ½ = -2$. Therefore, if $x > 0$, $y > 0$, and $x + y \leq 1$, then $\lg x + \lg y \leq -2$. ■

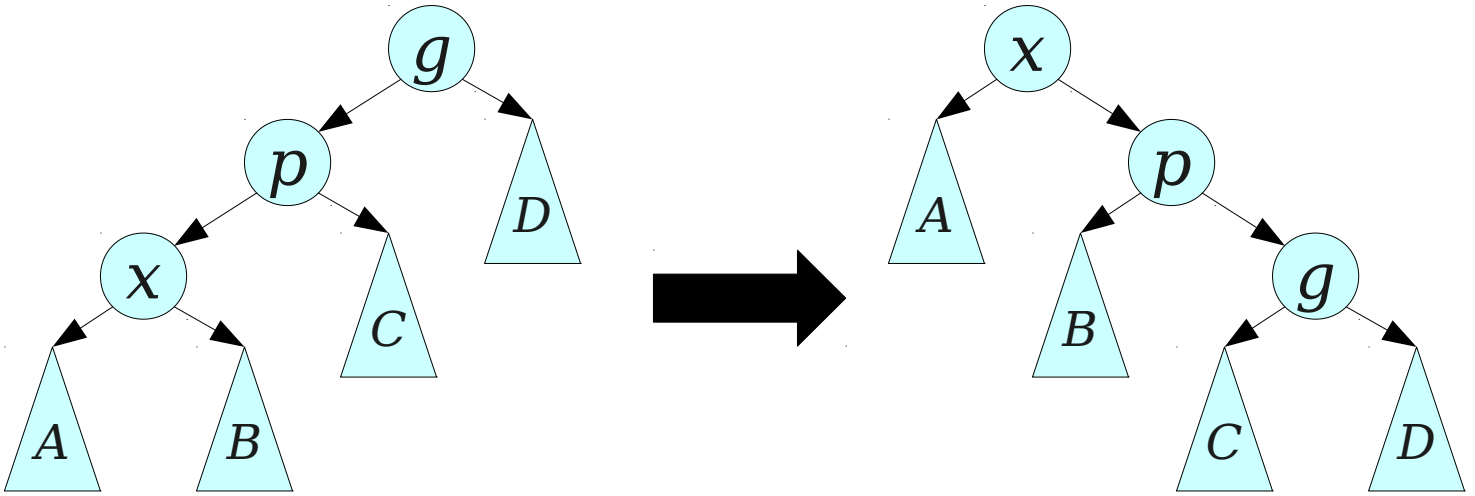
The Theorem

- ***Theorem:*** The amortized cost of splaying a node x in a tree with root t is at most

$$\mathbf{3(r(t) - r(x)) + 1}$$

- ***Proof outline:*** Show the following:
  - The amortized cost of a zig-zig or zig-zag is at most $3(r'(x) - r(x))$, where $r'(x)$ is the rank of x after the step and $r(x)$ is the rank of x before the step.
  - The amortized cost of a zig is at most $3(r'(x) - r(x)) + 1$.
  - Summing across all steps, the sum telescopes out to at most $3(r'(x) - r(x)) + 1$.
  - At the very end, x is the tree root. Therefore, we know that $r'(x) = r(t)$, so the total amortized cost is at most $3(r(t) - r(x)) + 1$.

The Zig-Zig Case



$$2 + \Delta\Phi = 2 + r'(x) + r'(p) + r'(g) - r(g) - r(p) - r(x)$$
$$= 2 + r'(p) + r'(g) - r(p) - r(x)$$

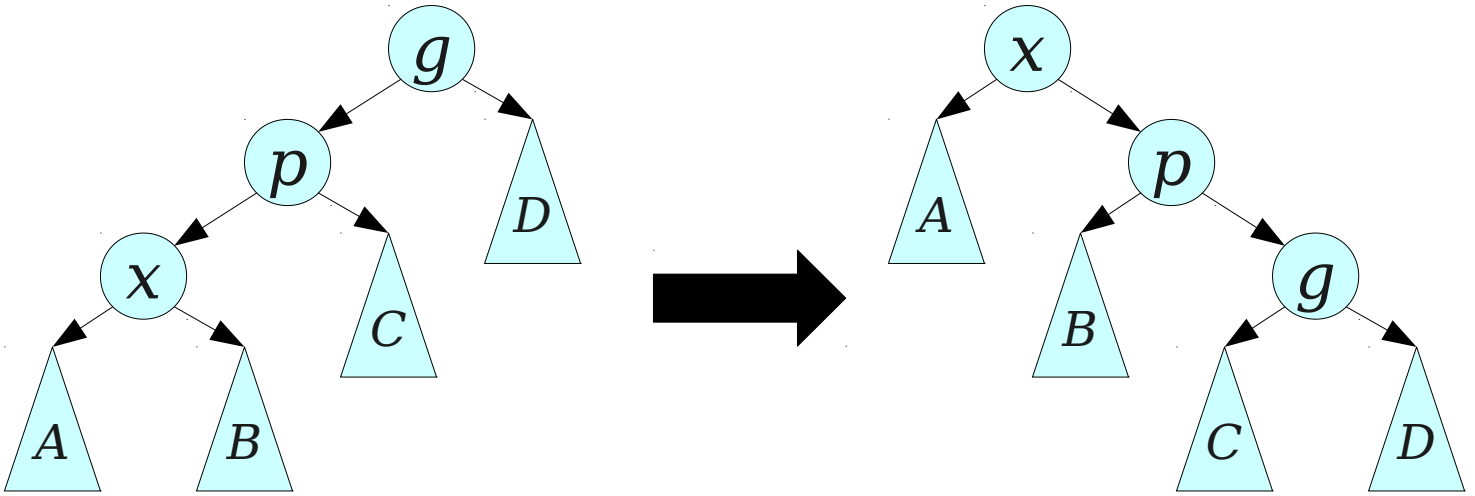
Goal: Bound the above expression by $3(r'(x) - r(x))$. We will therefore prove that

$$2 + r'(p) + r'(g) - r(p) - r(x) \leq 3(r'(x) - r(x))$$

Equivalently:

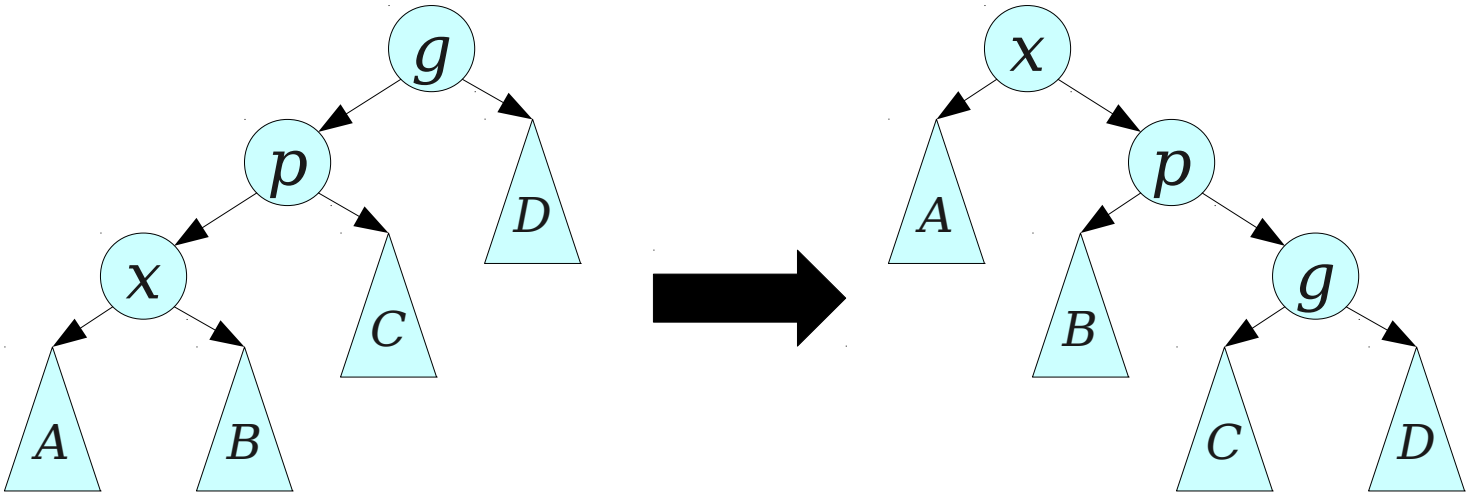
$$r'(p) + r'(g) - r(p) - r(x) - 3r'(x) + 3r(x) \leq -2$$

The Zig-Zig Case



$$
\begin{aligned}
& r'(p) + r'(g) - r(p) - r(x) - 3r'(x) + 3r(x) \\
= \; & r'(p) + r'(g) - r(p) - 3r'(x) + 2r(x) \\
\leq \; & r'(x) + r'(g) - r(p) - 3r'(x) + 2r(x) \\
= \; & r'(g) - r(p) - 2r'(x) + 2r(x) \\
\leq \; & r'(g) - r(x) - 2r'(x) + 2r(x) \\
= \; & r'(g) + r(x) - 2r'(x) \\
= \; & r(x) - r'(x) + r'(g) - r'(x)
\end{aligned}
$$

The Zig-Zig Case



$$r(x) - r'(x) + r'(g) - r'(x)$$
$$= \lg s(x) - \lg s'(x) + \lg s'(g) - \lg s'(x)$$
$$= \lg (s(x) / s'(x)) + \lg (s'(g) / s'(x))$$

Claim: $s(x) + s'(g) \leq s'(x)$

Therefore, $s(x) / s'(x) + s'(g) / s'(x) \leq 1$

So $\lg (s'(g) / s'(x)) + \lg (s(x) / s'(x)) \leq -2$

The Zig-Zag and Zig Cases

What We Just Did

- ***Theorem:*** The amortized cost of performing a splay at x is given by

$$\mathbf{3(}r(t) - r(x)\mathbf{) + 1}$$

- This theorem holds for any choice of weights we want to assign to the nodes.
- 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***.

Bounding the Potential Drop

- Let W be the sum of all the weights assigned to the nodes.
- Then $w_i \leq s(x_i) \leq W$, so $\lg w_i \leq r(x_i) \leq \lg W$.
- The maximum drop in potential purely for node x_i is therefore $\lg W - \lg w_i$.
- Therefore, the maximum drop in potential is given by

$$n \lg W - \sum_{i=1}^{n} \lg w_i$$

Properties of Splay Trees

- To use the theorem from before, we need to do three things:
  - Assign the weights to the nodes.
  - Determine the amortized cost of each operation on the splay tree.
  - Determine the maximum possible drop in potential from the operations.
- From this overall result, we can determine the total cost of performing a series of operations on a splay tree.

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 begin by showing that the amortized cost of a splay is $O(\log n)$.

Assign each node a weight of $1 / n$. Then $s(t) = 1$, so $r(t) = 0$. For any node x, we see $s(x) \geq 1 / n$, so $r(x) \geq -\lg n$. The amortized cost of a splay is at most

$$\begin{aligned} & 3(r(t) - r(x)) + 1 \\ \leq\ & 3(0 - (-\lg n)) + 1 \\ \leq\ & 3 \lg n + 1 \\ =\ & O(\log n) \end{aligned}$$

Additionally, the maximum drop in potential is given by

$$n \lg W - \sum_{i=1}^{n} \lg w_i = -\sum_{i=1}^{n} \lg \frac{1}{n} = \sum_{i=1}^{n} \lg n = n \lg n$$

So the total cost of performing these m operations is $O(m \log n + n \log n)$. ■

Interpreting the Balance Theorem

- The Balance Theorem says that m operations on an n-node splay tree takes time $O(m \log n + n \log n)$.
- As long as $m \geq n$ (reasonable in most circumstances), the overall runtime will then be $O(m \log n)$.
- Via the aggregate method, all operations on splay trees have amortized cost $O(\log n)$.
- However, if some existing tree is given as input and we only do a few splays, the amortized cost might be much higher.

A Stronger Result

- **Recall:** Motivation behind splay trees was a search for trees that perform better than O(log n) on skewed distributions.
- **Claim:** Splay trees have a runtime that's extremely close to optimal.

Static Optimality

- ***Theorem:*** (Entropy bound) In an optimal BST where element x_i is accessed with probability p_i, the expected cost of a successful search in the tree is

$$\Omega\left(\sum_{i=1}^{n} -p_i \lg p_i\right)$$

- This is the ***Shannon entropy*** of the distribution; it's $\lg n$ if accesses are perfectly uniform and 0 if accesses are totally skewed.
- Entropy is typically denoted H, so we can express the entropy bound as follows: the expected cost of a lookup in an optimal BST is $\boldsymbol{\Omega(H)}$.
- You'll prove this lower bound on the problem set.

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.

Interpreting Static Optimality

- If you perform lookups on a splay tree and look up each element once, the amortized cost of each lookup is $O(1 + H)$
- Given the lower bound of $\Omega(H)$ for any static BST, this is close to perfect.
- **Amazing fact:** Splay trees are never more than a constant factor slower than *any* fixed BST for the given set of elements, assuming that each element is always looked up at least once.

Theorem (Static Optimality Theorem): Let $S = \{ x_1, \ldots, x_n \}$ be a set of keys stored in a splay tree. Suppose a series of m lookups is performed where all lookups are successful and every node is looked up at least once. Then the total access time is $O(m + mH)$, where H is the entropy of the access probabilities.

Proof: Assign each x_i weight p_i, where p_i is the fraction of the lookups that access x_i. Then $s(t) = 1$, so $r(t) = 0$. For any node x_i, we know $s(x_i) \geq p_i$, so $r(x_i) \geq \lg p_i$. The amortized cost of a lookup of x_i is therefore at most

$$3(r(t) - r(x_i)) + 1 \leq -3 \lg p_i + 1$$

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

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

Additionally, the total drop in potential is given by

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

Therefore, the total cost is $O(m + mH)$, as required. ■

Beating Static Optimality

- On many classes of access sequences, splay trees can outperform statically optimal BSTs
- Example: 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.

- Another example: 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 $O(1)$.

An Open Problem: **Dynamic Optimality**

The BST Model

- Consider a BST with a pointer called the ***finger***, which points to some element.
- 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

- A BST that fits into this model is called ***valid*** if it correctly answers every lookup query performed on it.
- The ***cost*** of a valid BST on a series of queries (denoted ***c(T, Q)***) is the number of elementary operations performed while answering queries Q.
- We'll say that the cost of a sequence of queries Q, denoted ***c(Q)***, is $\min\{c(T, Q)\}$ over all valid dynamic BSTs T.

Competitive Ratios

- A BST T is said to be ***f(n)-competitive*** if the following holds for any series of queries Q:

$$c(T, Q) \leq f(n) \cdot c(Q)$$

- In other words, the cost T incurs when answering any series of queries Q is at most $f(n)$ times the cost of the optimal BST for Q.

What We Know

- **Known:** Statically-optimal BSTs are O(log n)-competitive.
- **Known:** There are data structures (like the *Tango tree*) that are O(log log n)-competitive.
- **Unknown:** Are there O(1)-competitive BST data structures?
- **Known:** A recently-developed data structure is O(1)-competitive iff there exists an O(1)-competitive BST data structure.
- **Conjectured:** Splay trees are O(1)-competitive.

Next Time

- **Tries**
  - A fundamental data structure for storing strings.
- **Aho-Corasick String Matching**
  - A fast data structure for finding occurrences of strings in a large text