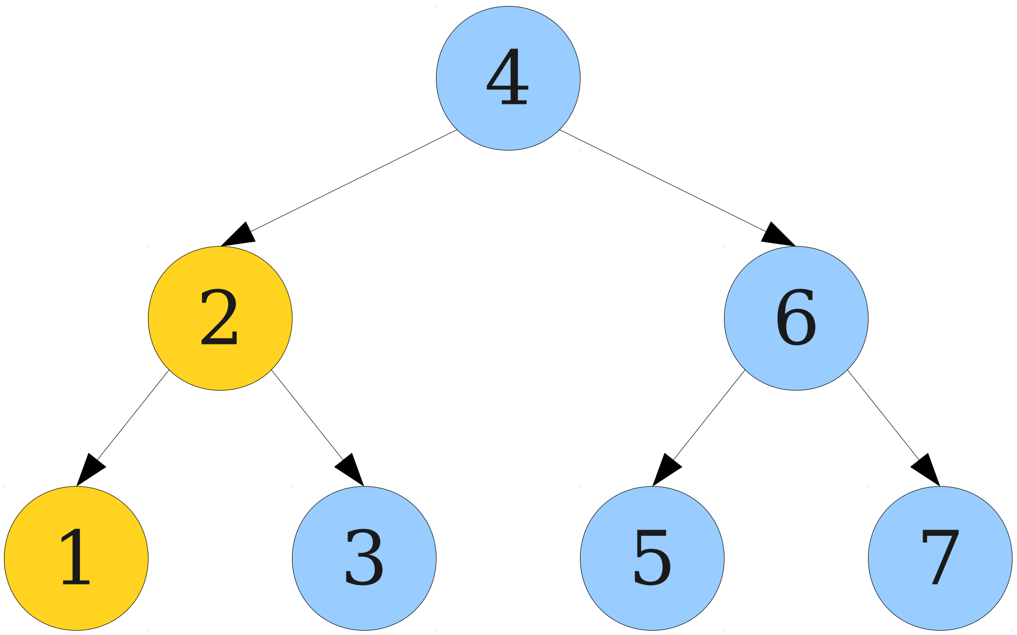
Splay Trees

Static Optimality

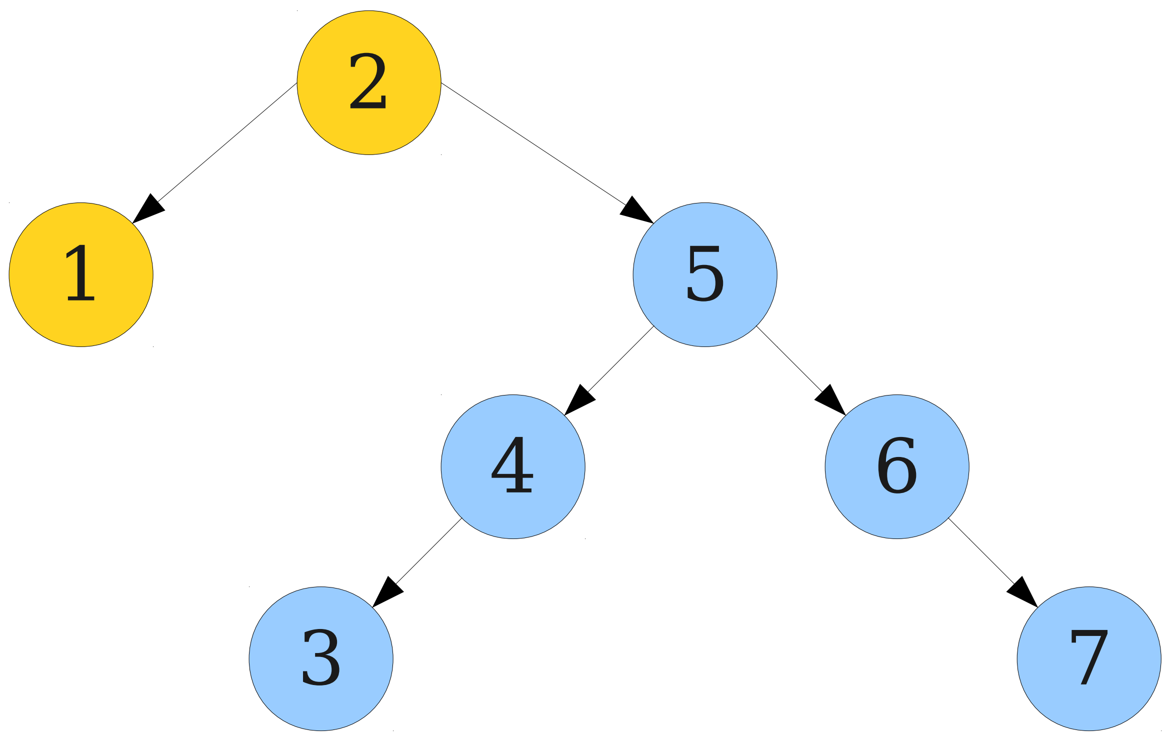
Balanced BSTs

- 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

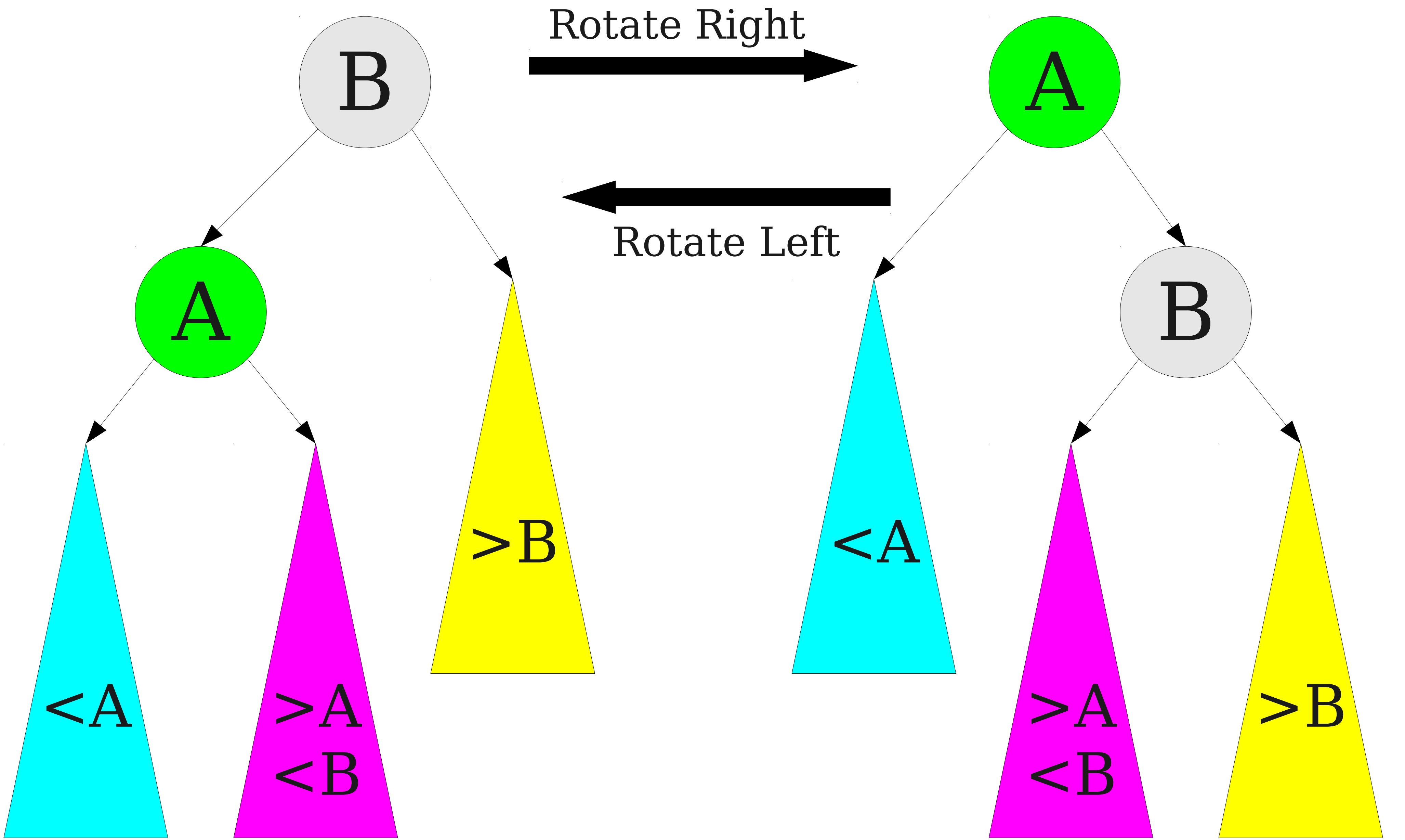
- There is an $O(n^2)$-time dynamic programming algorithm for constructing statically optimal binary search trees.
  - Knuth, 1971
- There is an $O(n \log n)$-time greedy algorithm for constructing binary search trees whose cost is within 1.5 of optimal.
  - Mehlhorn, 1975
- These algorithms assume that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

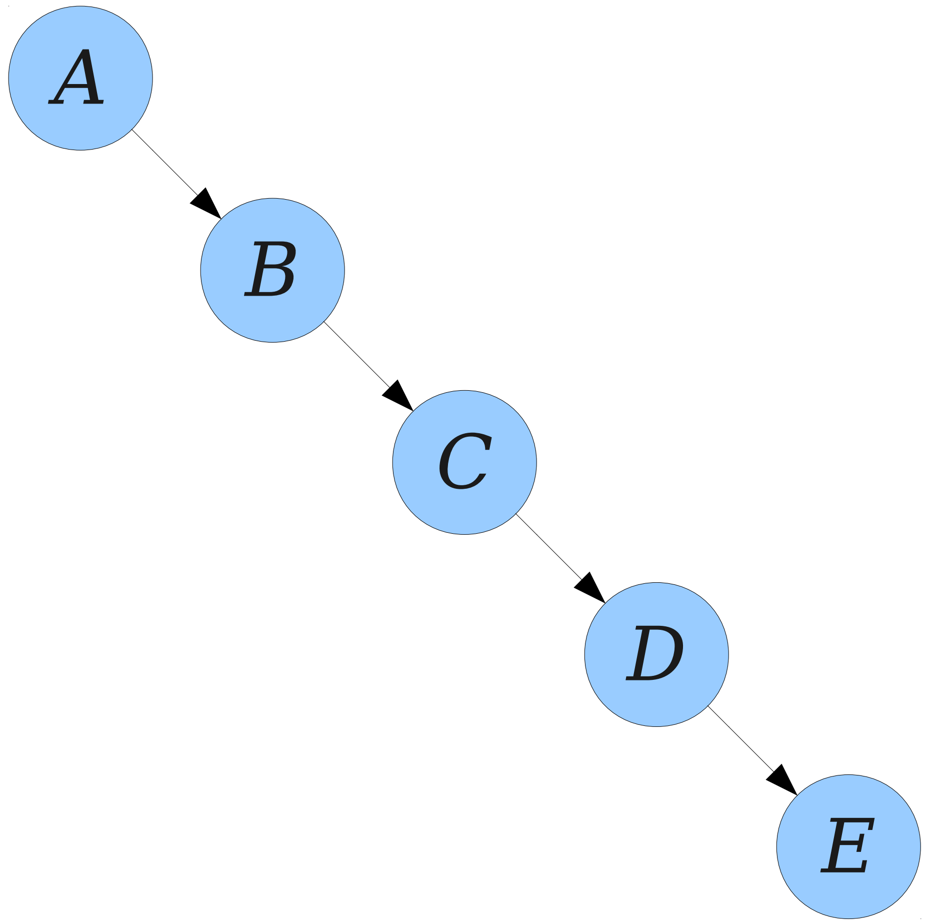
Refresher: Tree Rotations



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



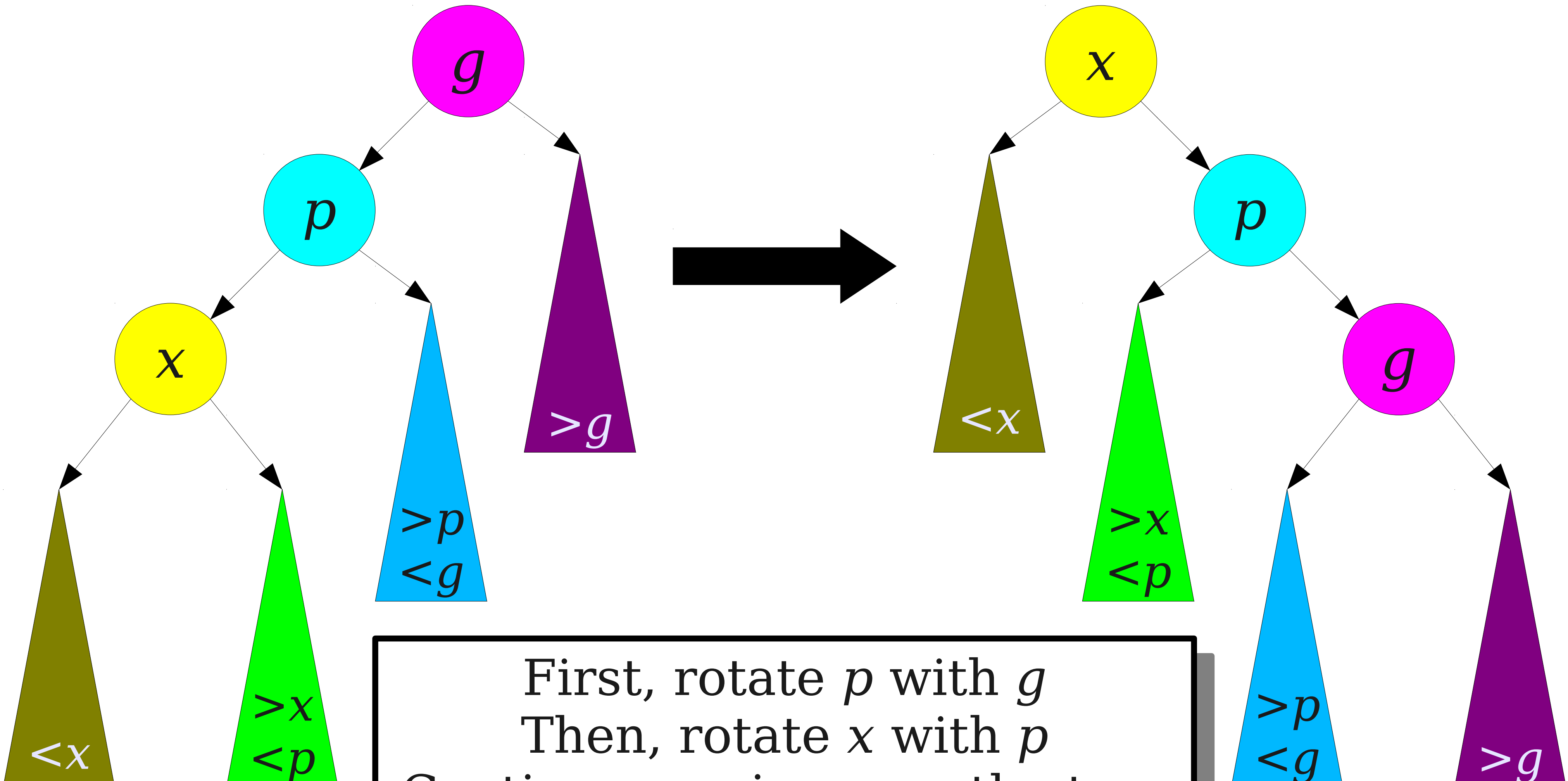
The Problem

- 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

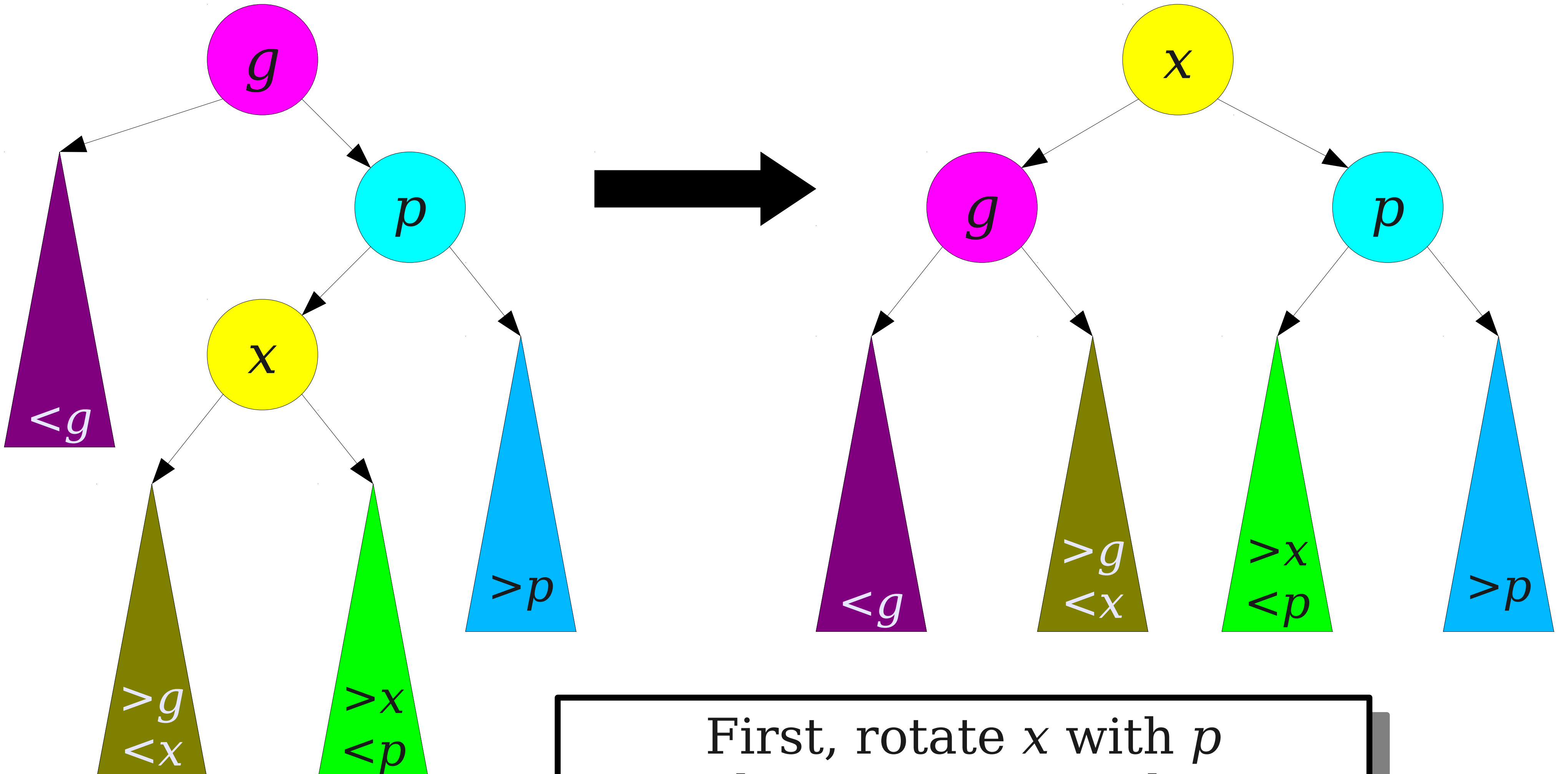
- In 1983, Daniel Sleator and Robert Tarjan invented an operation called ***splaying***.
- Rotates an element to the root of the tree, but does so in a way that's more “fair” to other nodes in the tree.
- There are three cases for splaying.

Case 1: Zig-Zig



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

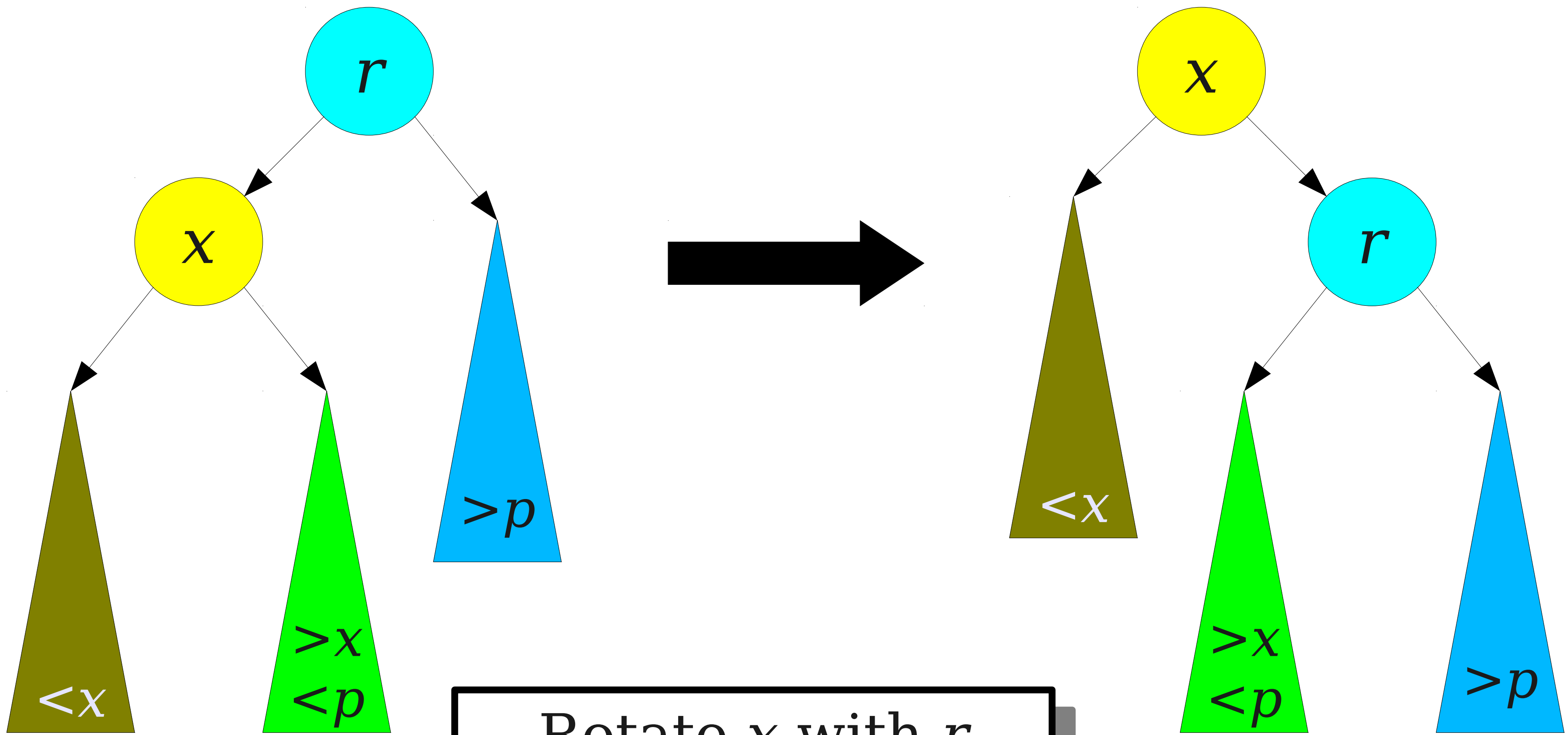
Case 2: Zig-Zag



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

Case 3: Zig

(Assume r is the tree root)



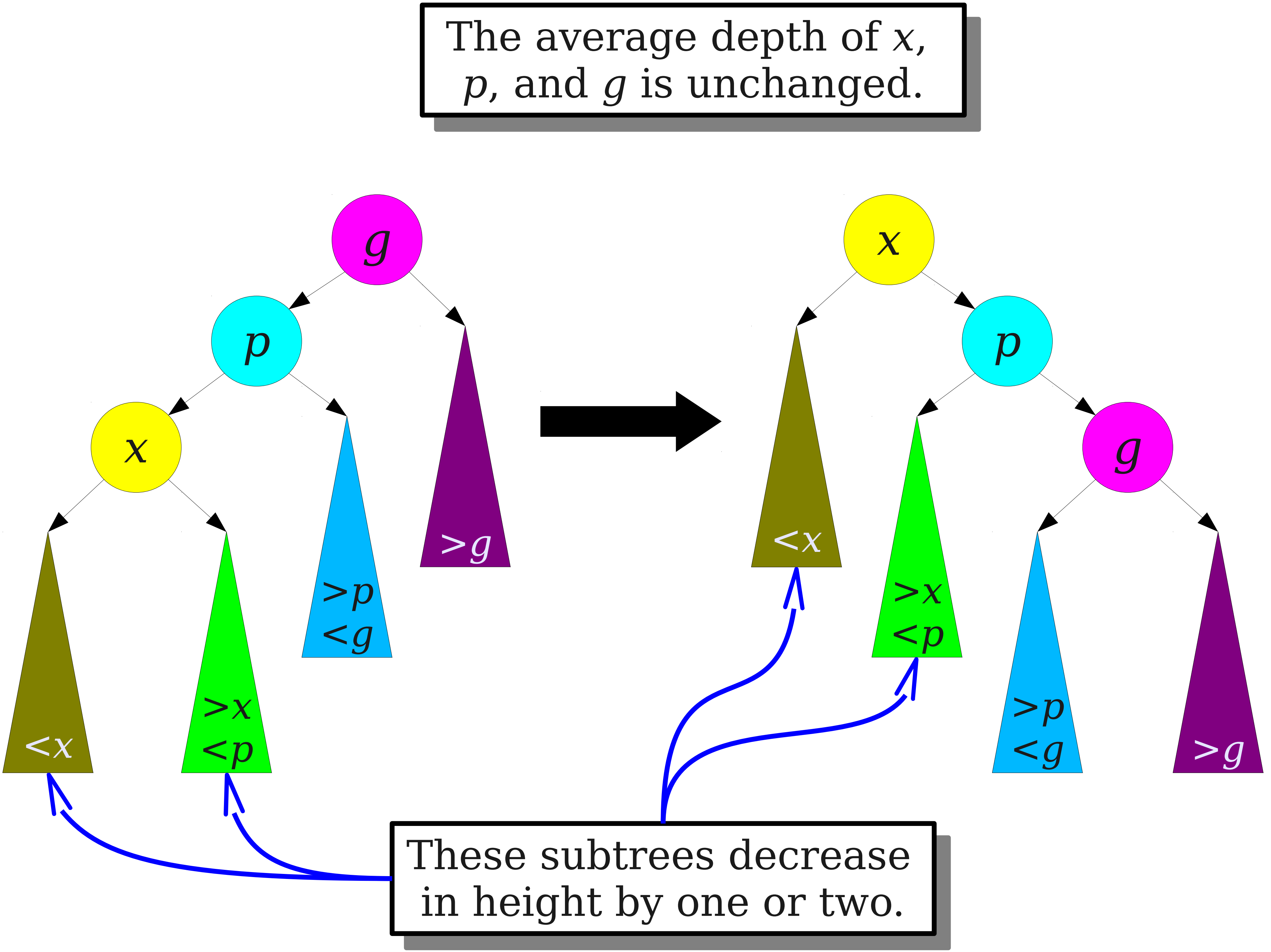
Rotate x with r
x is now the root.

Splaying, Empirically

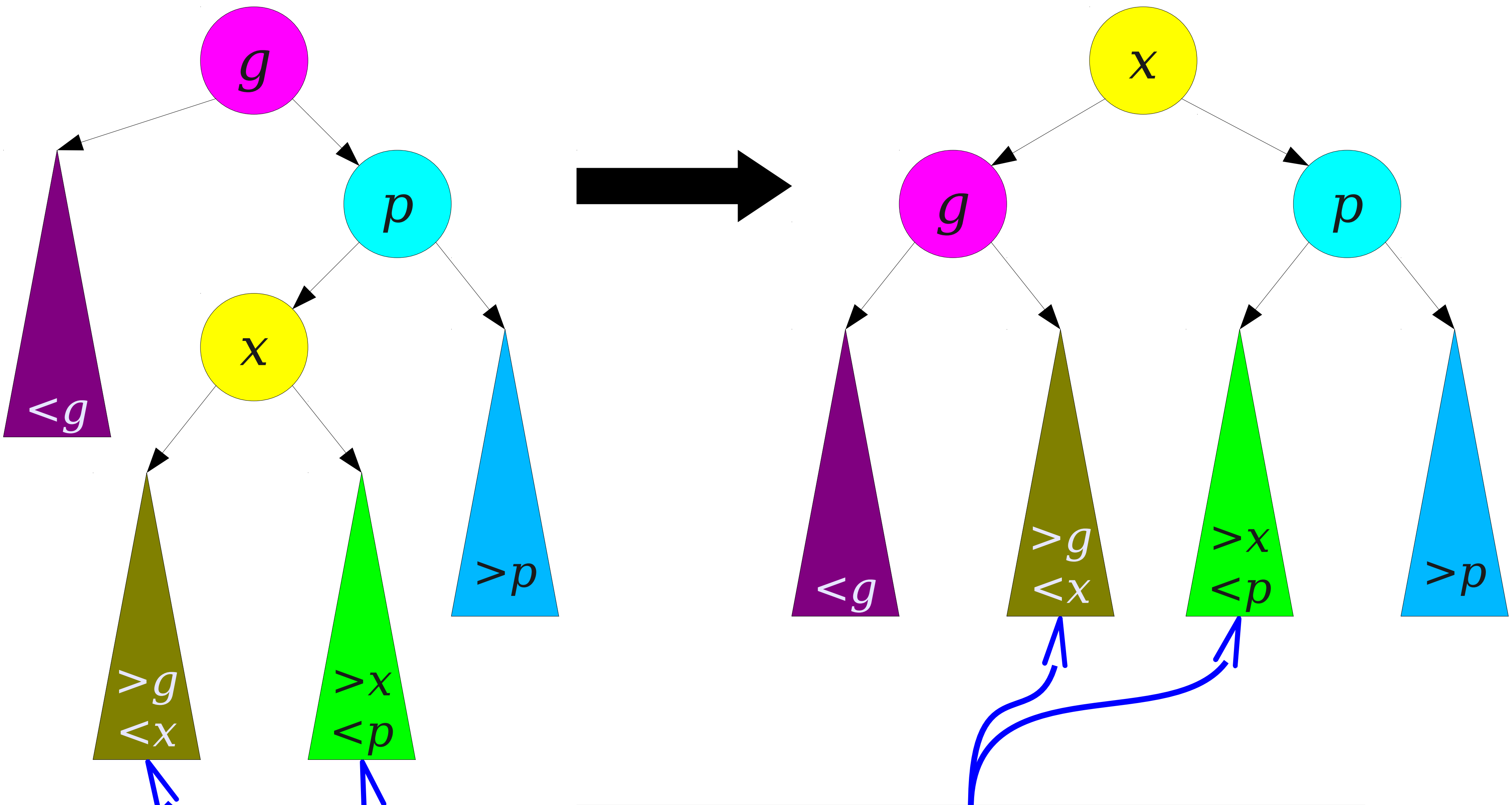
- Splaying nodes that are deep in the tree tends to correct the tree shape.
- Why is this?
- Is this a coincidence?

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.

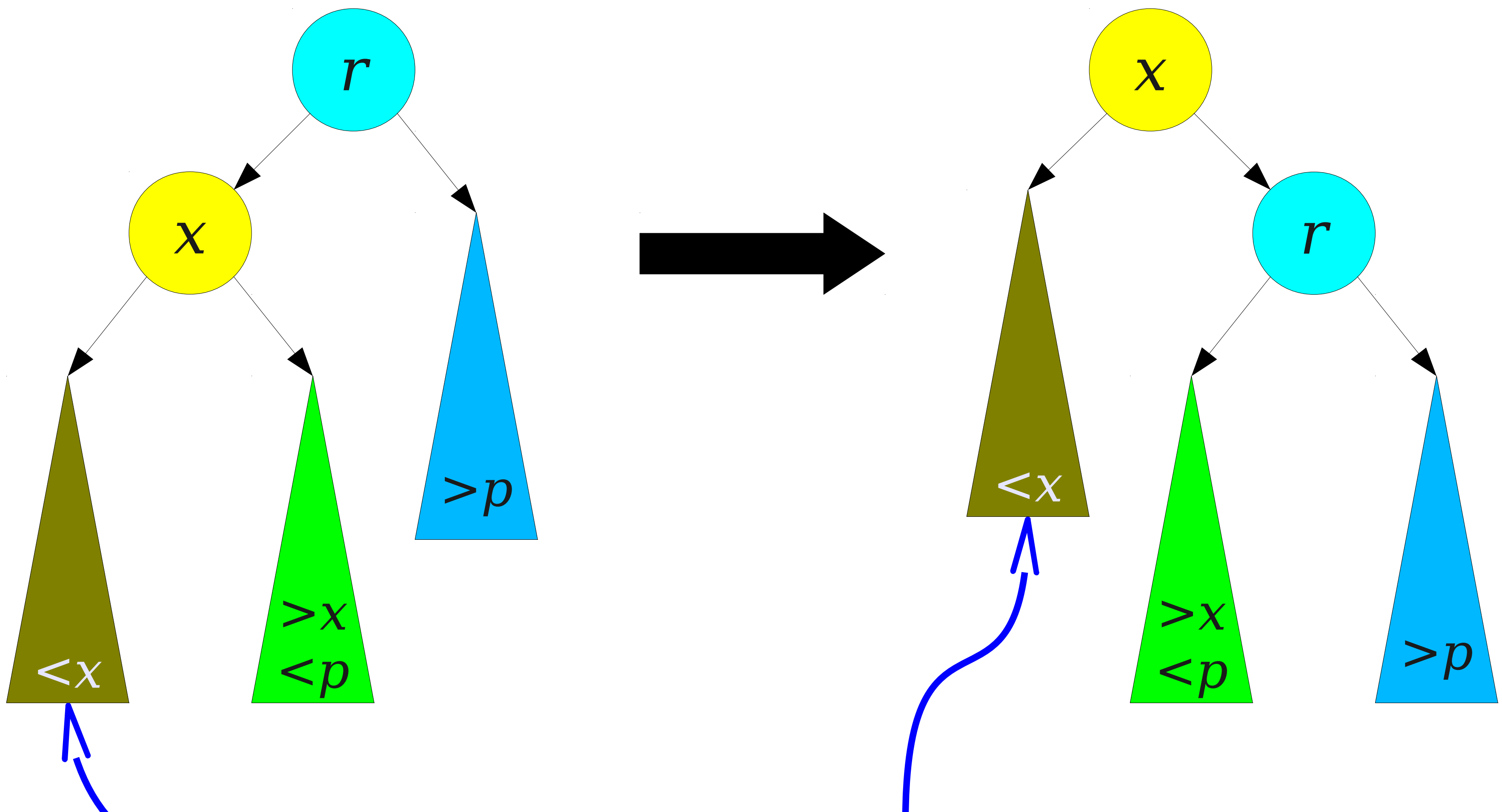

The average depth of x, p, and g is unchanged.
g
p
x
<x
>x <p
>p <g
>g
x
p
g
<x
>x <p
>p <g
>g
These subtrees decrease in height by one or two.

The average height of x, p, and g decreases by $^{1}/_{3}$.



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

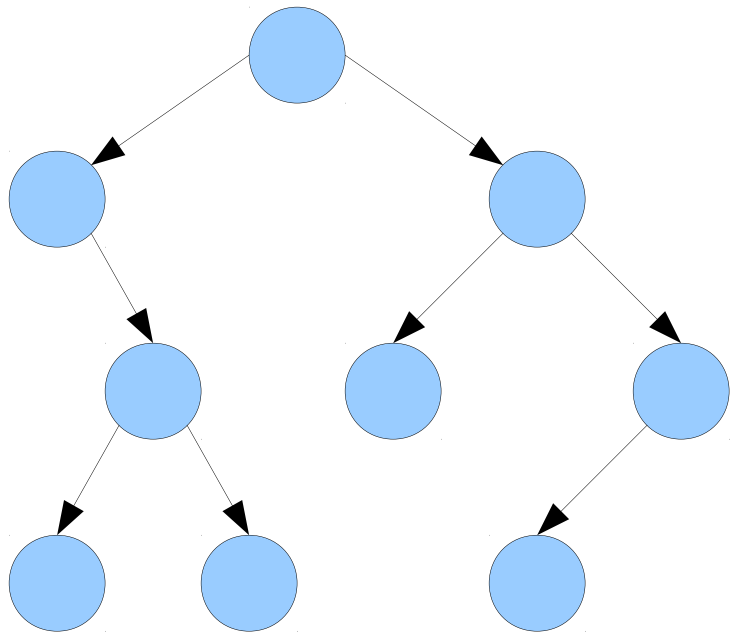
- Each rotation done only slightly penalizes each other part of the tree (say, adding +1 or +2 depth).
- Each splay rapidly cuts down the height of each node on the access path.
- Slow growth in height, combined with rapid drop in height, is a hallmark of amortized efficiency.

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.

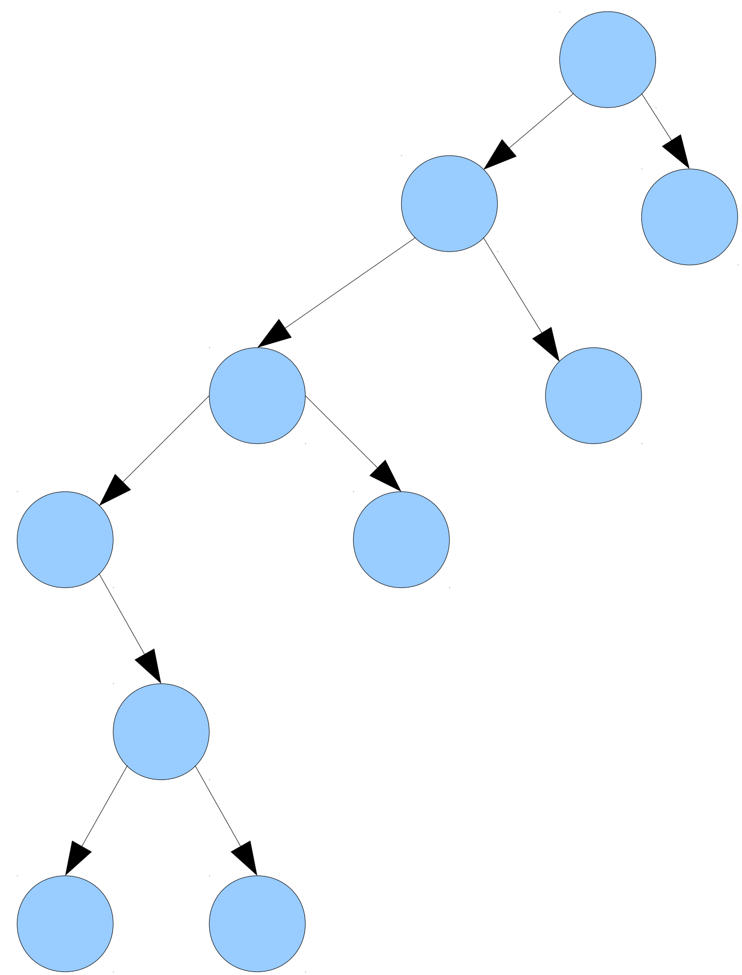
Lookups

- 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.



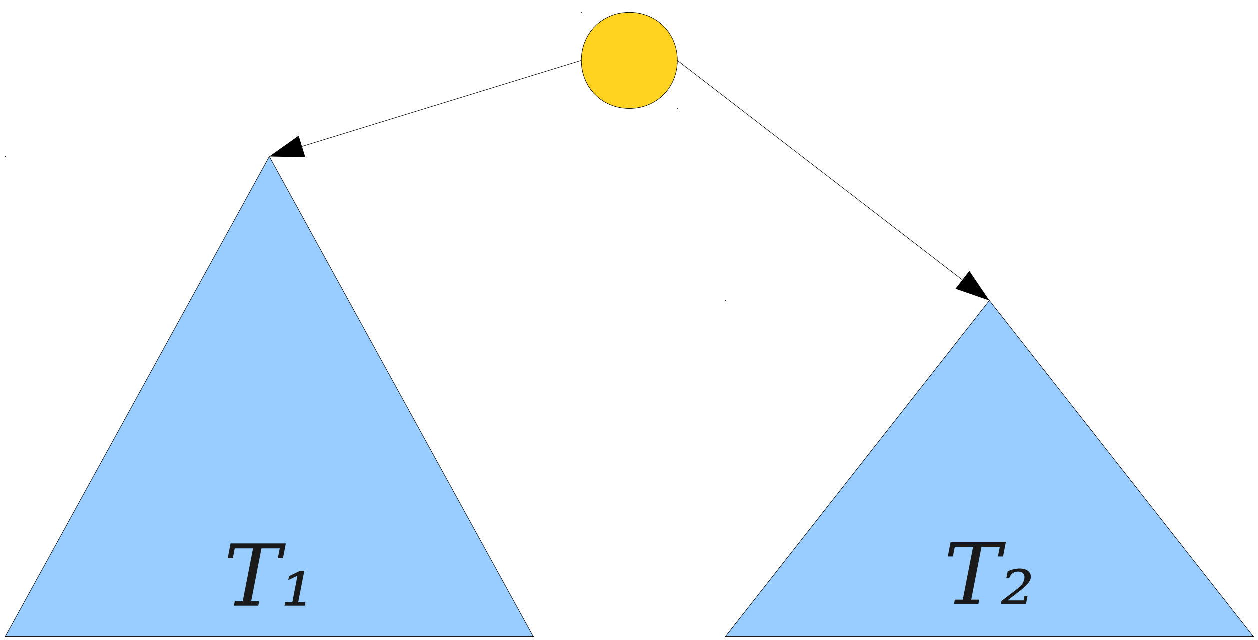
Insertions

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



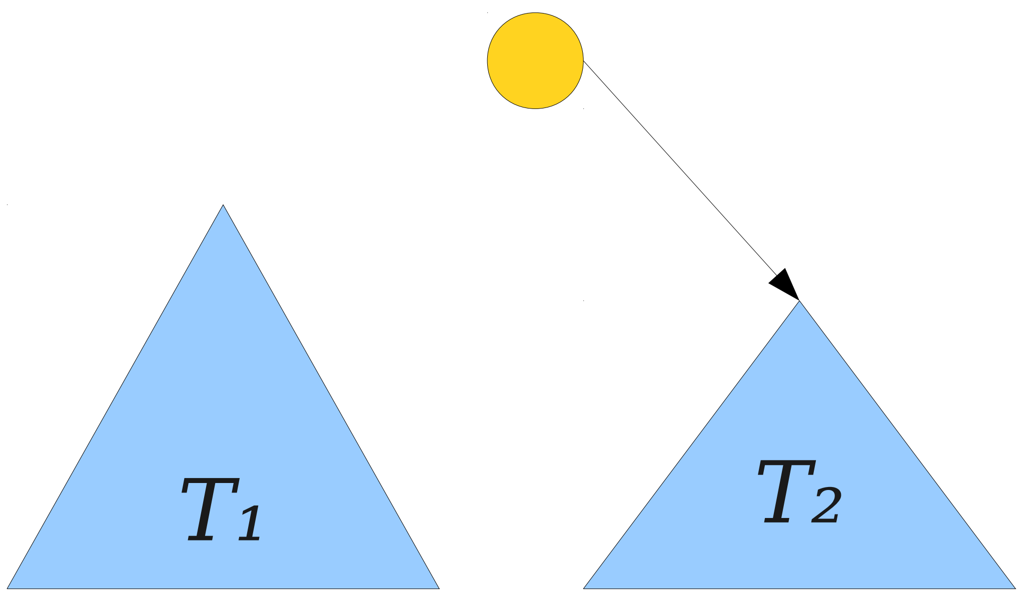
Join

- 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.



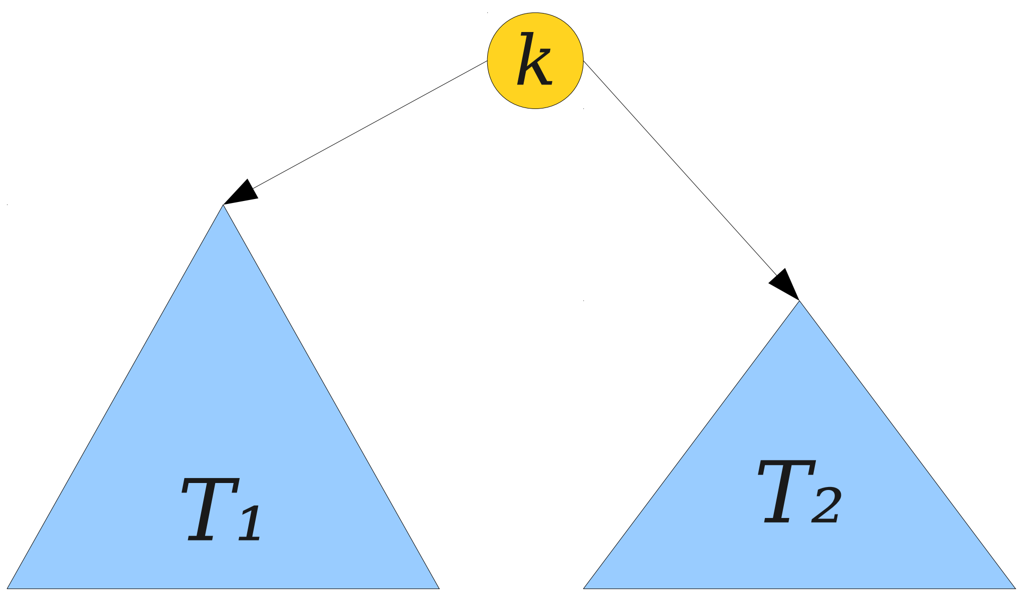
Split

- 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.