Balanced Trees

Part Two

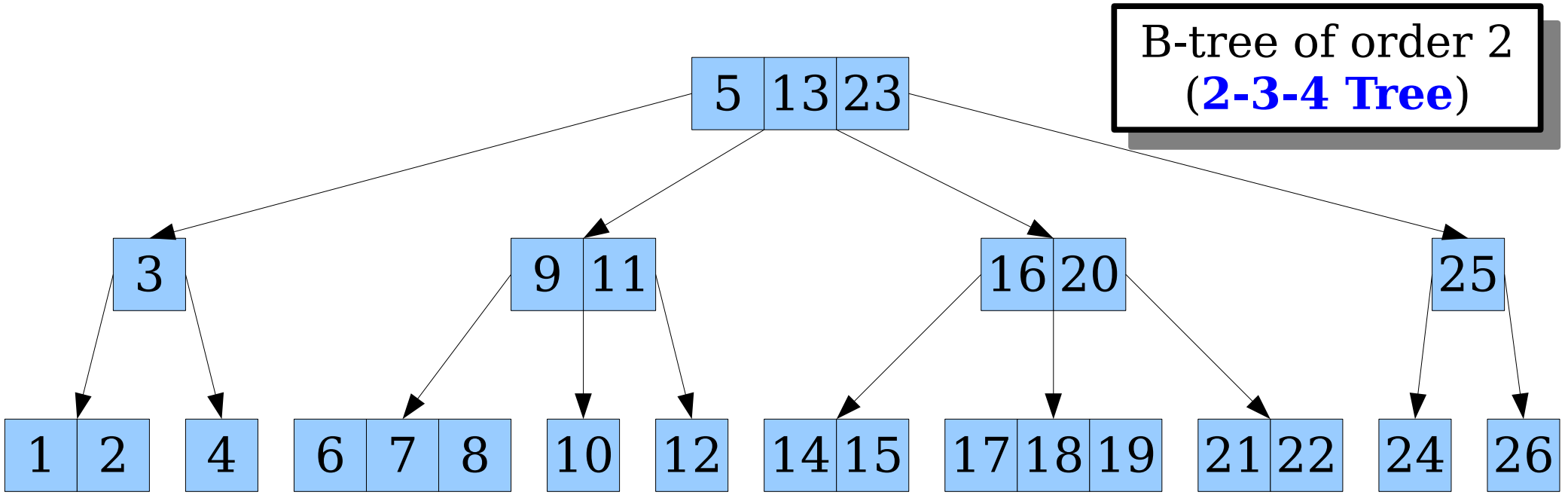
Outline for Today

- ***Recap from Last Time***
  - Review of B-trees, 2-3-4 trees, and red/black trees.
- ***Order Statistic Trees***
  - BSTs with indexing.
- ***Augmented Binary Search Trees***
  - Building new data structures out of old ones.
- ***Dynamic 1D Closest Points***
  - Applications to hierarchical clustering.
- ***Join and Split Operations***
  - Two powerful BST primitives.

Review from Last Time

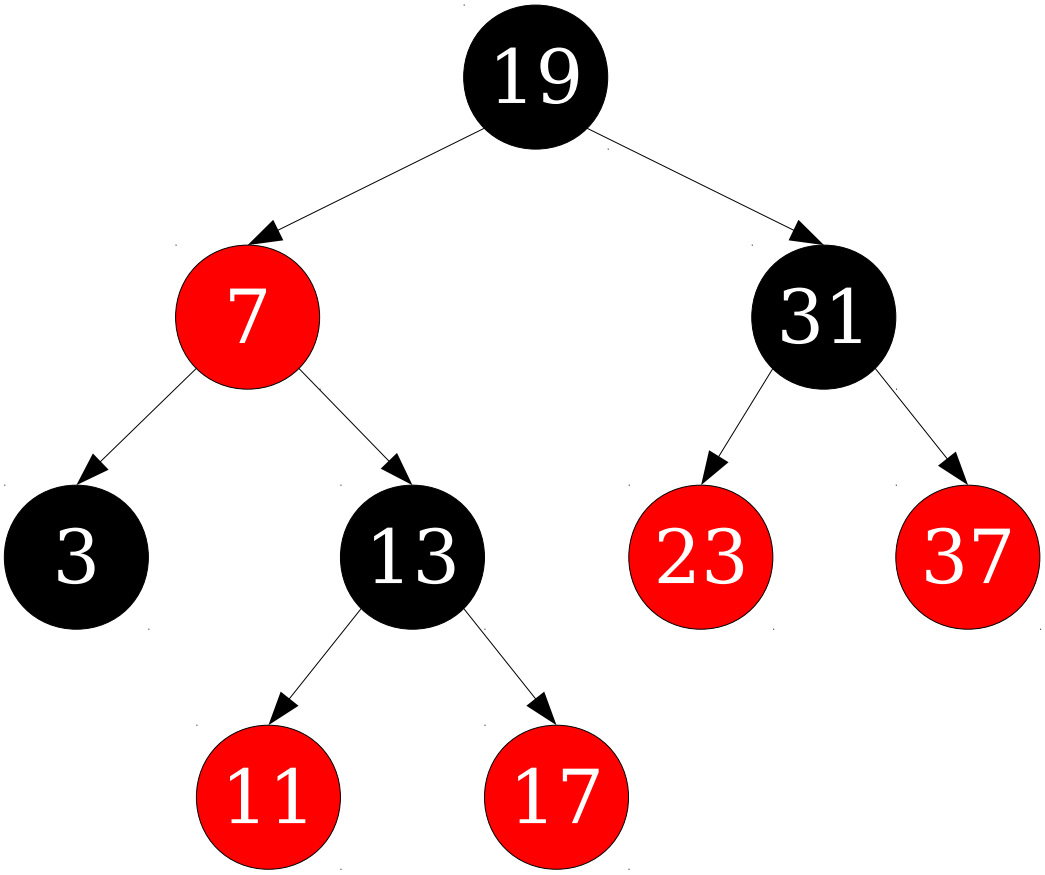
B-Trees

- A ***B-tree of order b*** is a multiway search tree with the following properties:
  - All leaf nodes are stored at the same depth.
  - All non-root nodes have between $b - 1$ and $2b - 1$ keys.
  - The root has at most $2b - 1$ keys.
  - All root-null paths pass through the same number of nodes.



Red/Black Trees

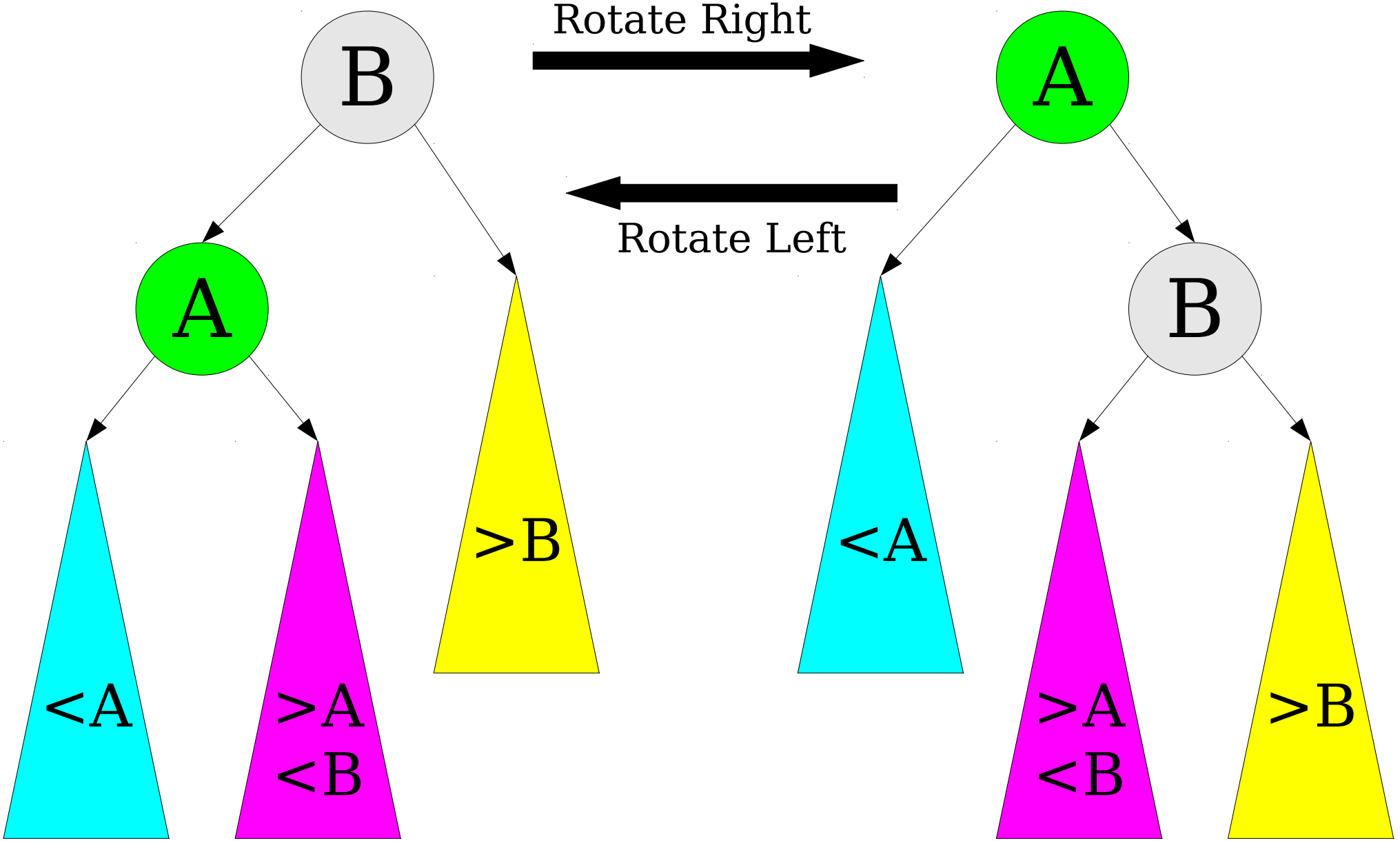
- A ***red/black tree*** is a BST with the following properties:
  - Every node is either red or black.
  - The root is black.
  - No red node has a red child.
  - Every root-null path in the tree passes through the same number of black nodes.



Red/Black Trees ≡ 2-3-4 Trees

- Red/black trees are an ***isometry*** of 2-3-4 trees; they represent the structure of 2-3-4 trees in a different way.
- Accordingly, red/black trees have height O(log n).
- After inserting or deleting an element from a red/black tree, the tree invariants can be fixed up in time O(log n) by applying rotations and color flips that simulate a 2-3-4 tree.

Tree Rotations

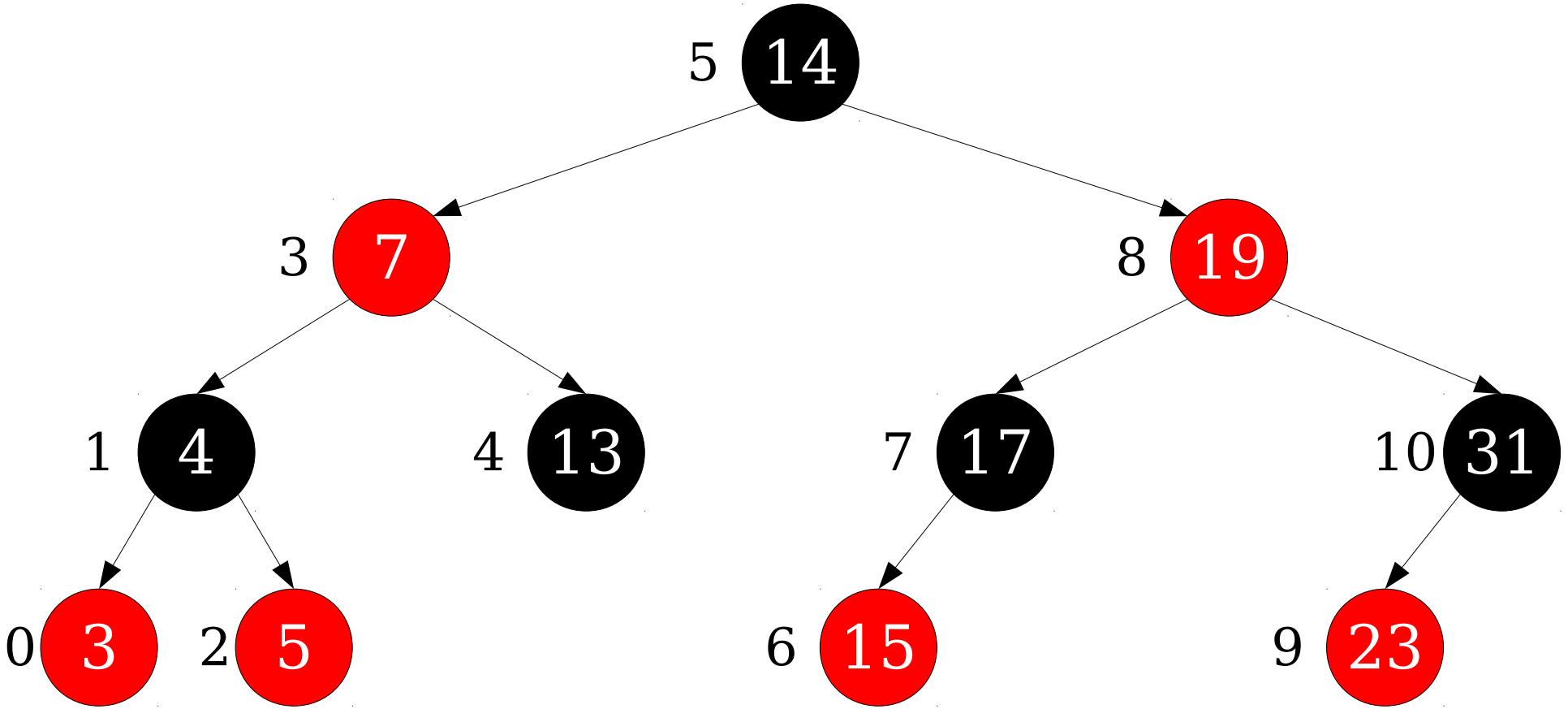


Dynamic Order Statistics

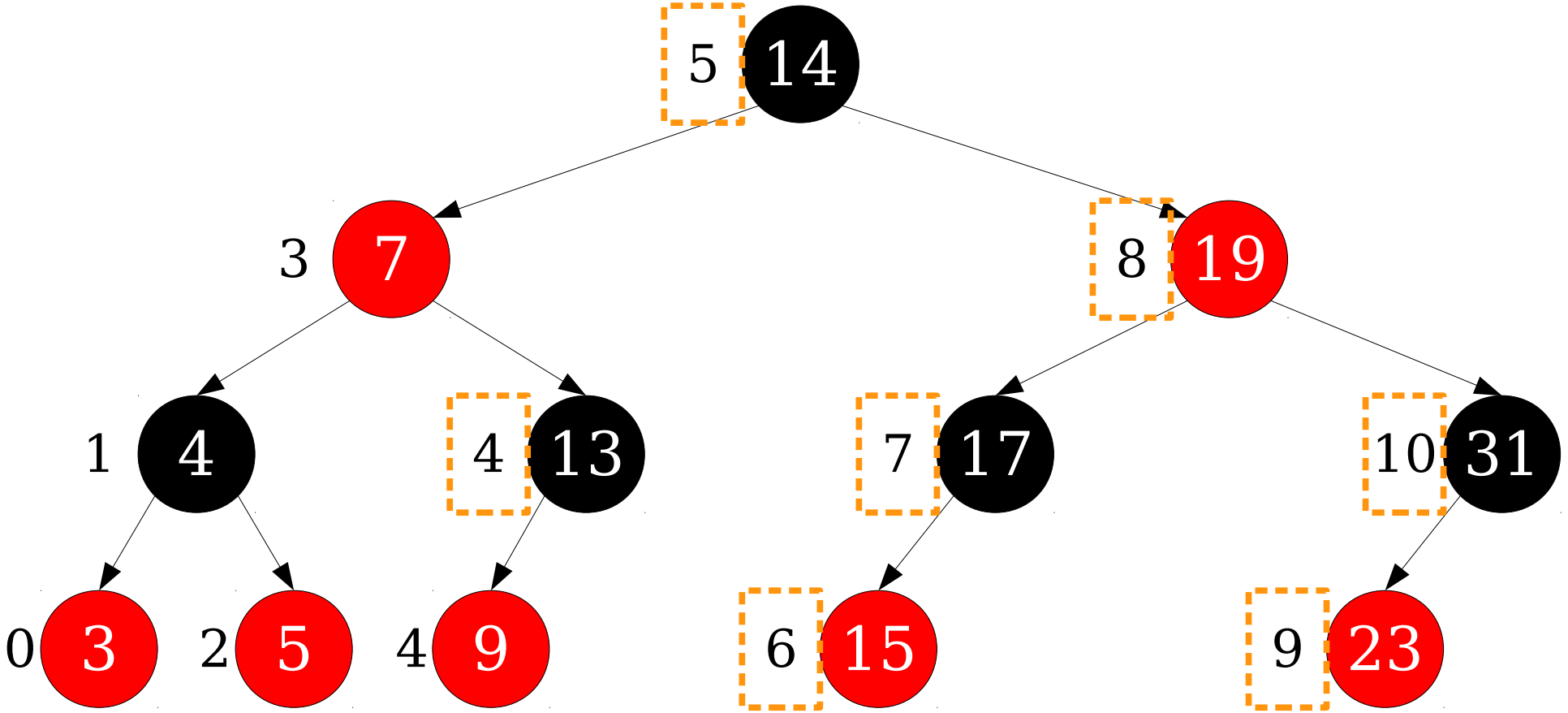
Order Statistics

- In a set S of totally ordered values, the ***kth order statistic*** is the kth smallest value in the set.
  - The 0^{th} order statistic is the minimum value.
  - The 1^{st} order statistic is the second-smallest value.
  - The $(n - 1)^{st}$ order statistic is the maximum value.
- In CS161, you (probably) saw quickselect or the median-of-medians algorithm for computing order statistics of a fixed array.
- ***Goal:*** Solve this problem efficiently when the data set is changing – that is, the underlying set of elements can have insertions and deletions intermixed with queries.

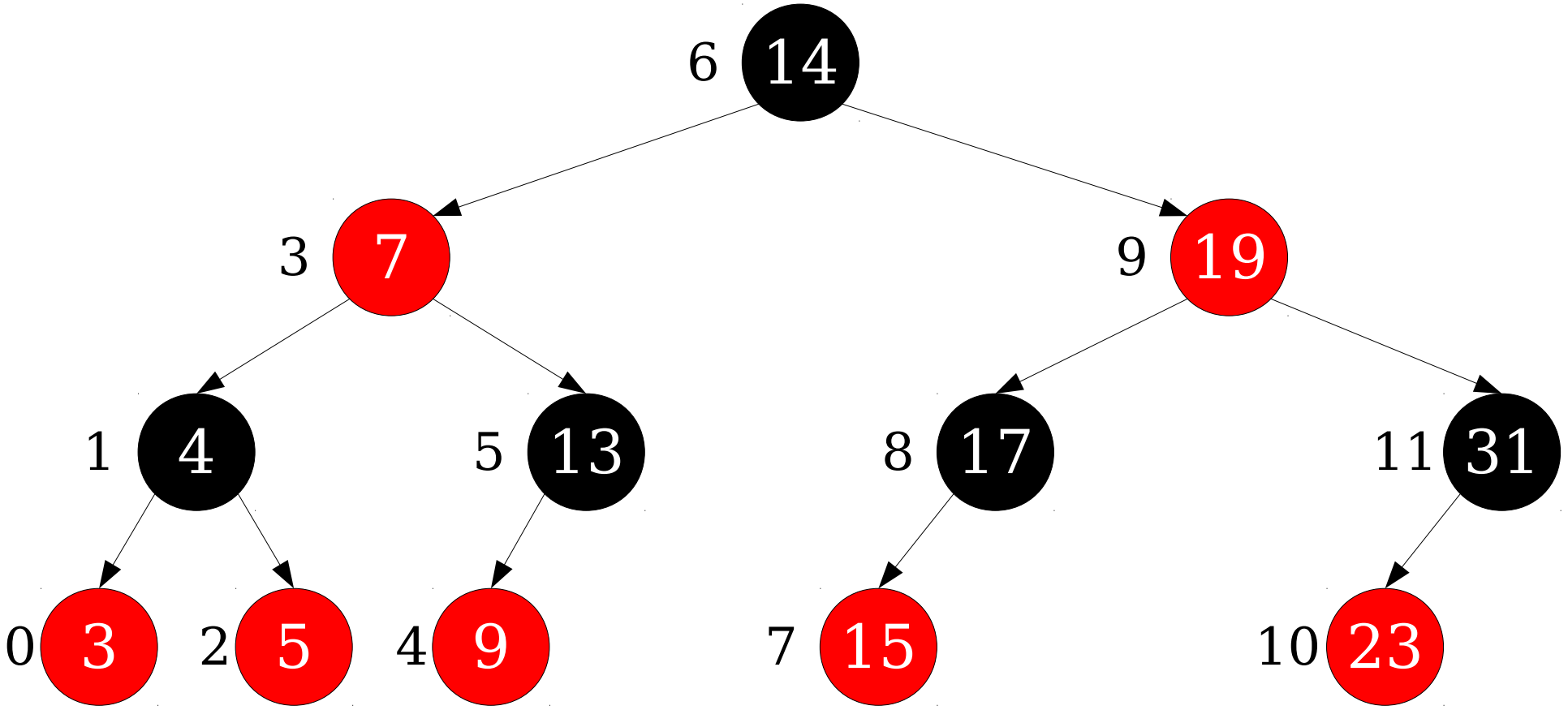
Finding Order Statistics



Finding Order Statistics



Finding Order Statistics

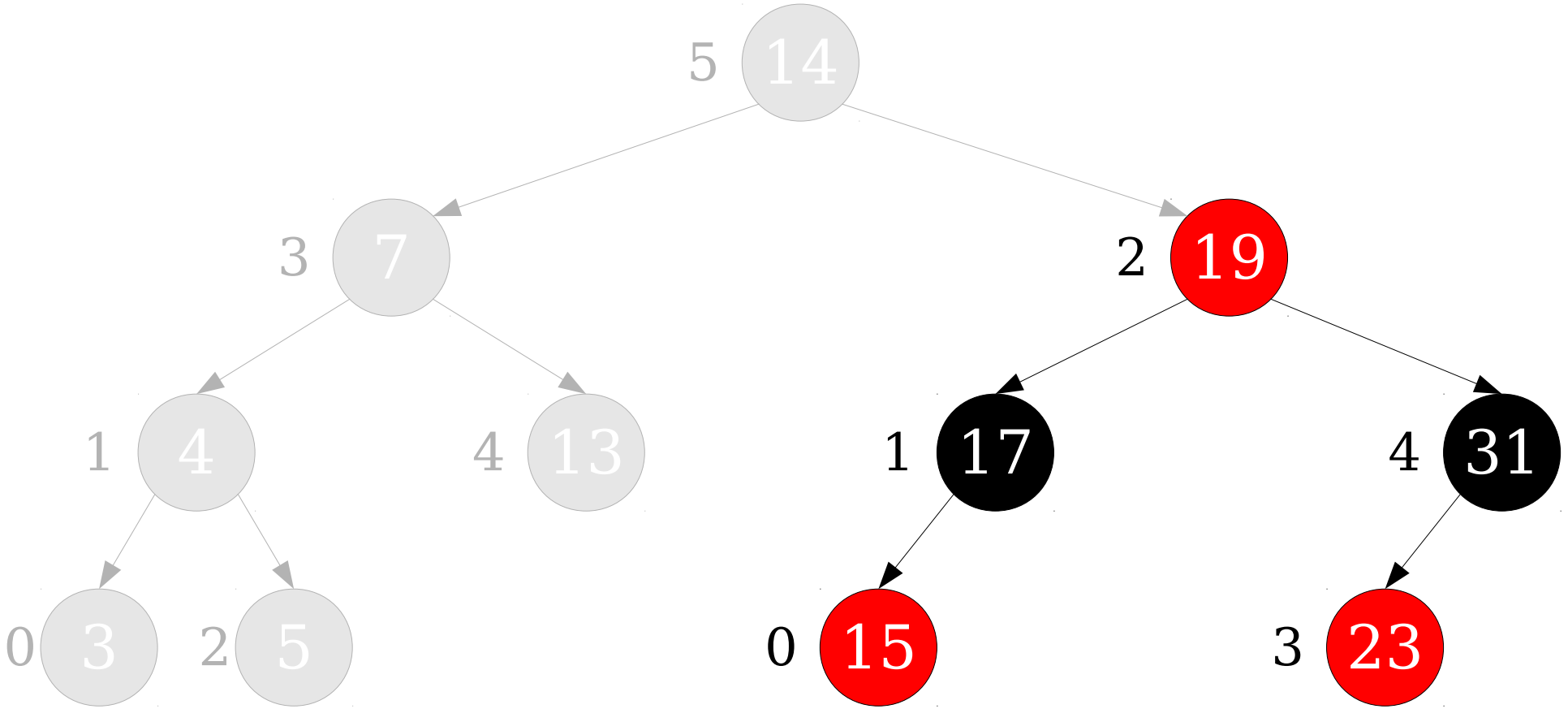


Problem: After inserting a new value, we may have to update $\Theta(n)$ values.

An Observation

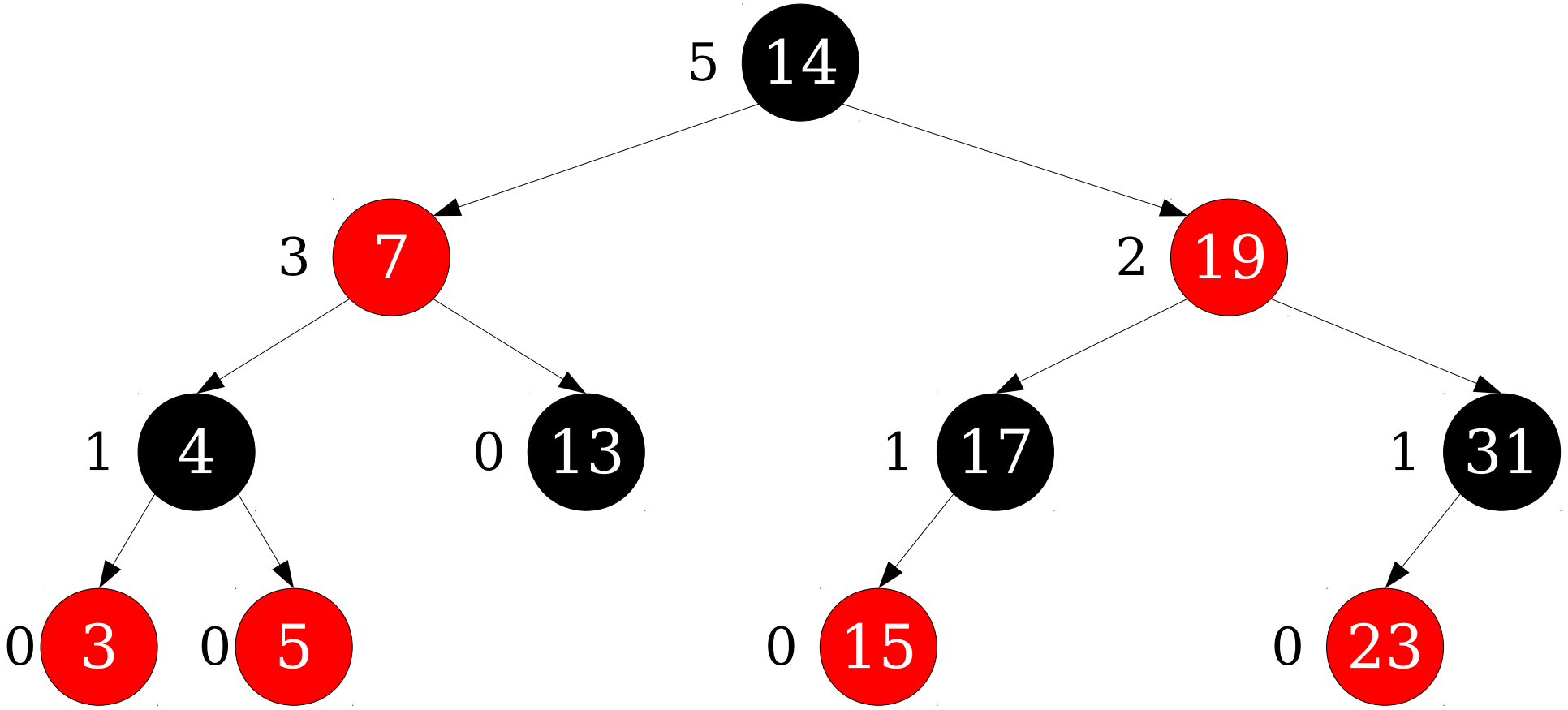
- The exact index of each number is a *global property* of the tree.
  - Depends on all other nodes and their positions.
- Could we find a *local property* that lets us find order statistics?
  - That is, something that depends purely on nearby nodes.

Finding Order Statistics



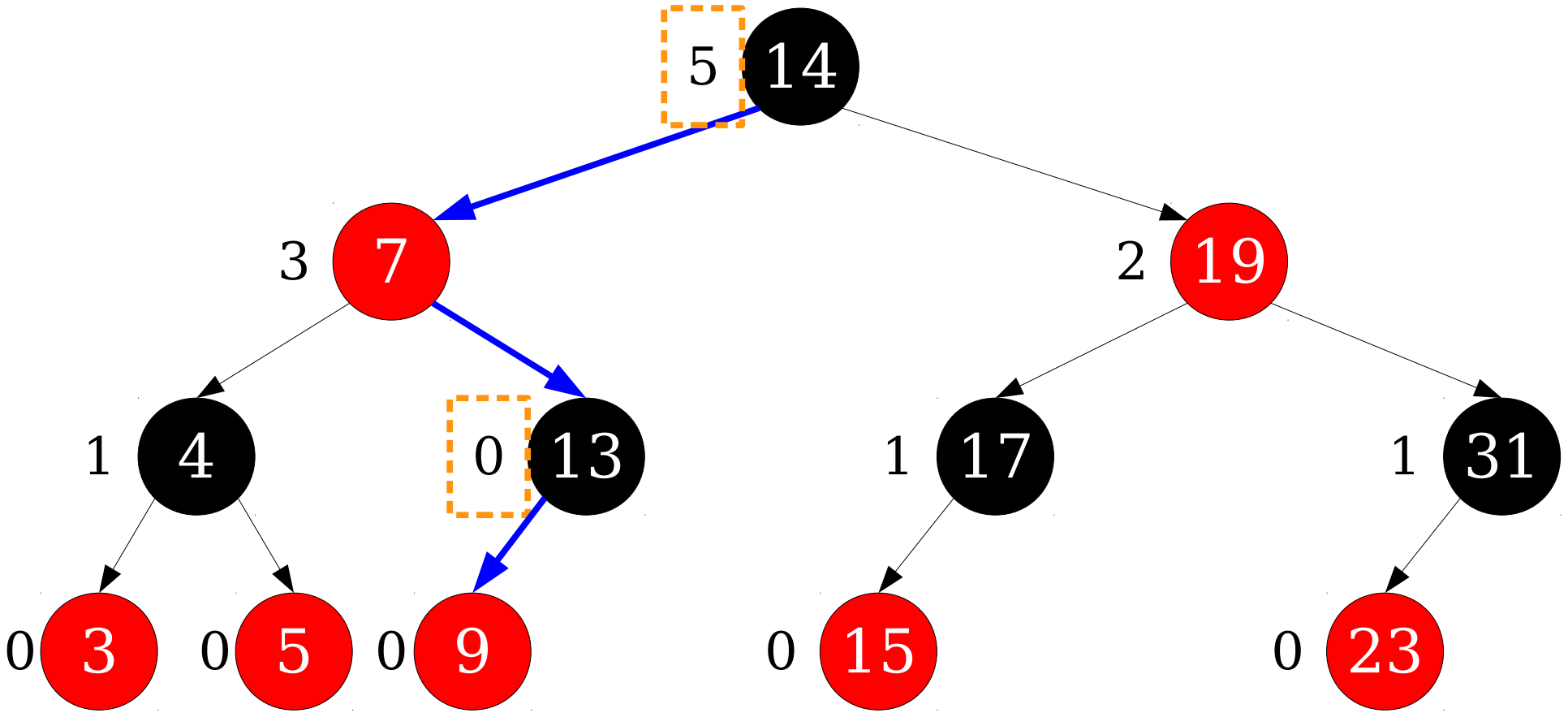
If new nodes are added to the left subtree, these numbers don't need to be updated.

Finding Order Statistics



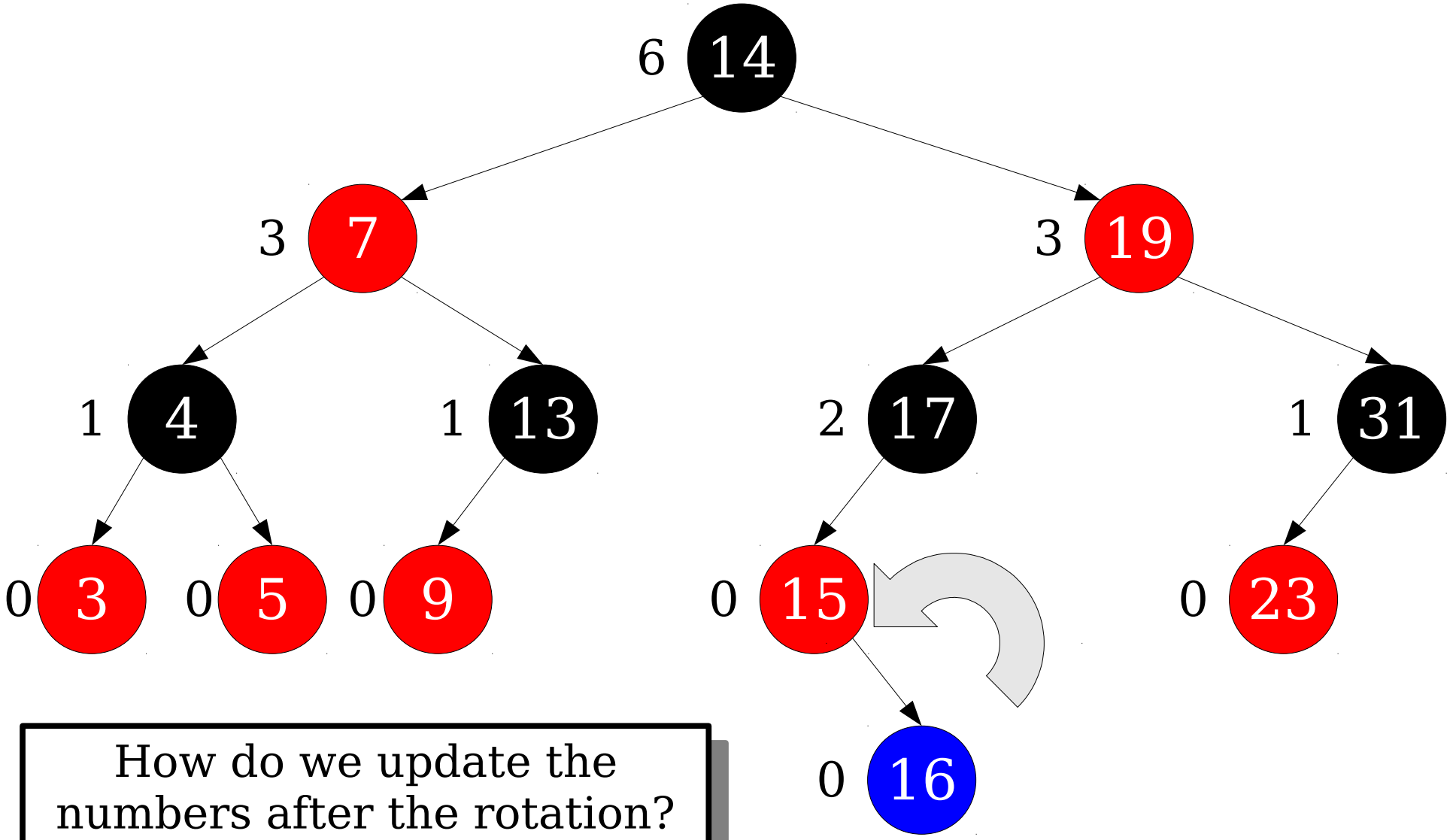
Each node is annotated with the number of children in its left subtree.

Finding Order Statistics



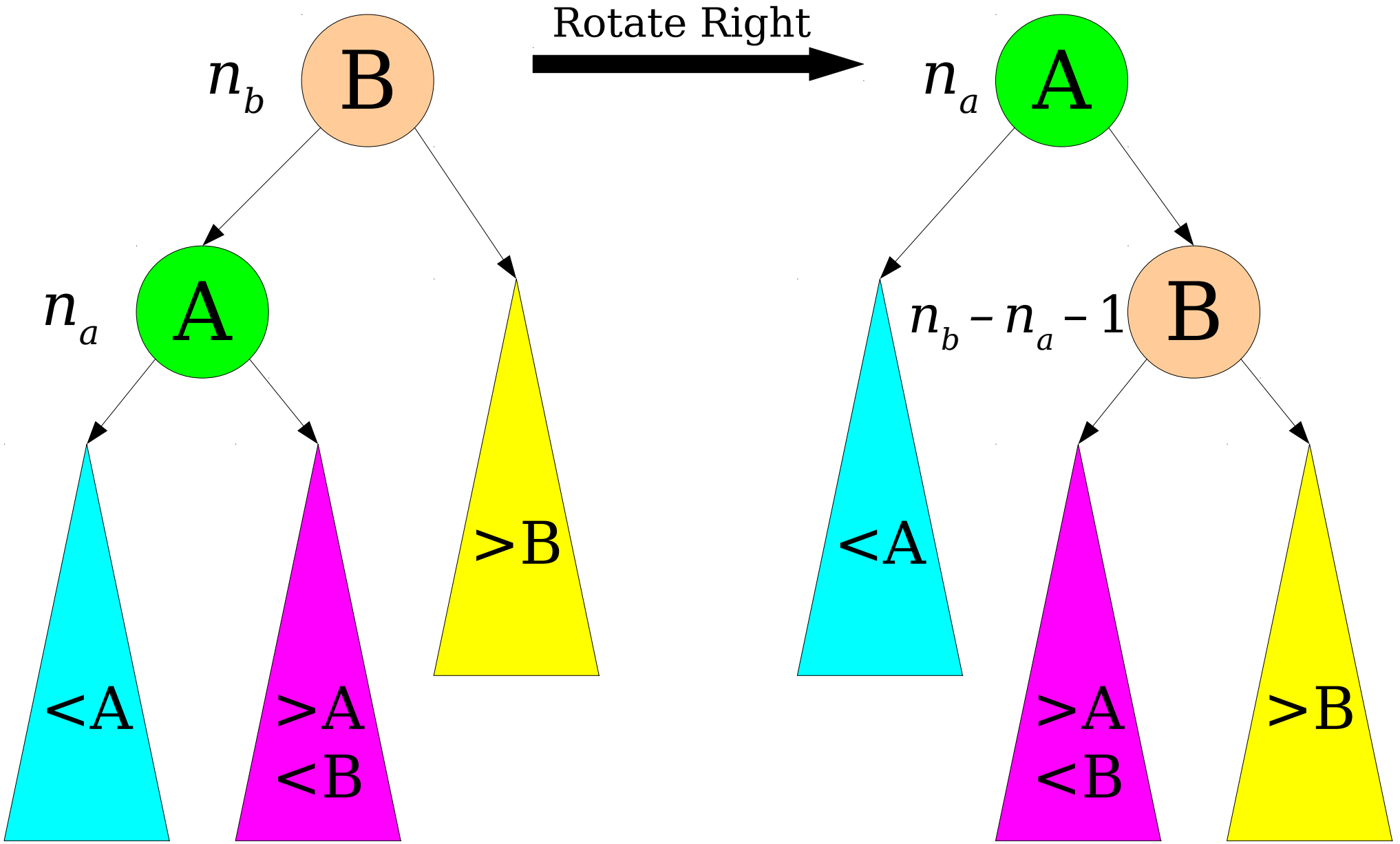
Since the number just holds the number of nodes in its left subtree, we only need to increment the value for nodes that have the new node in its left subtree.

Finding Order Statistics

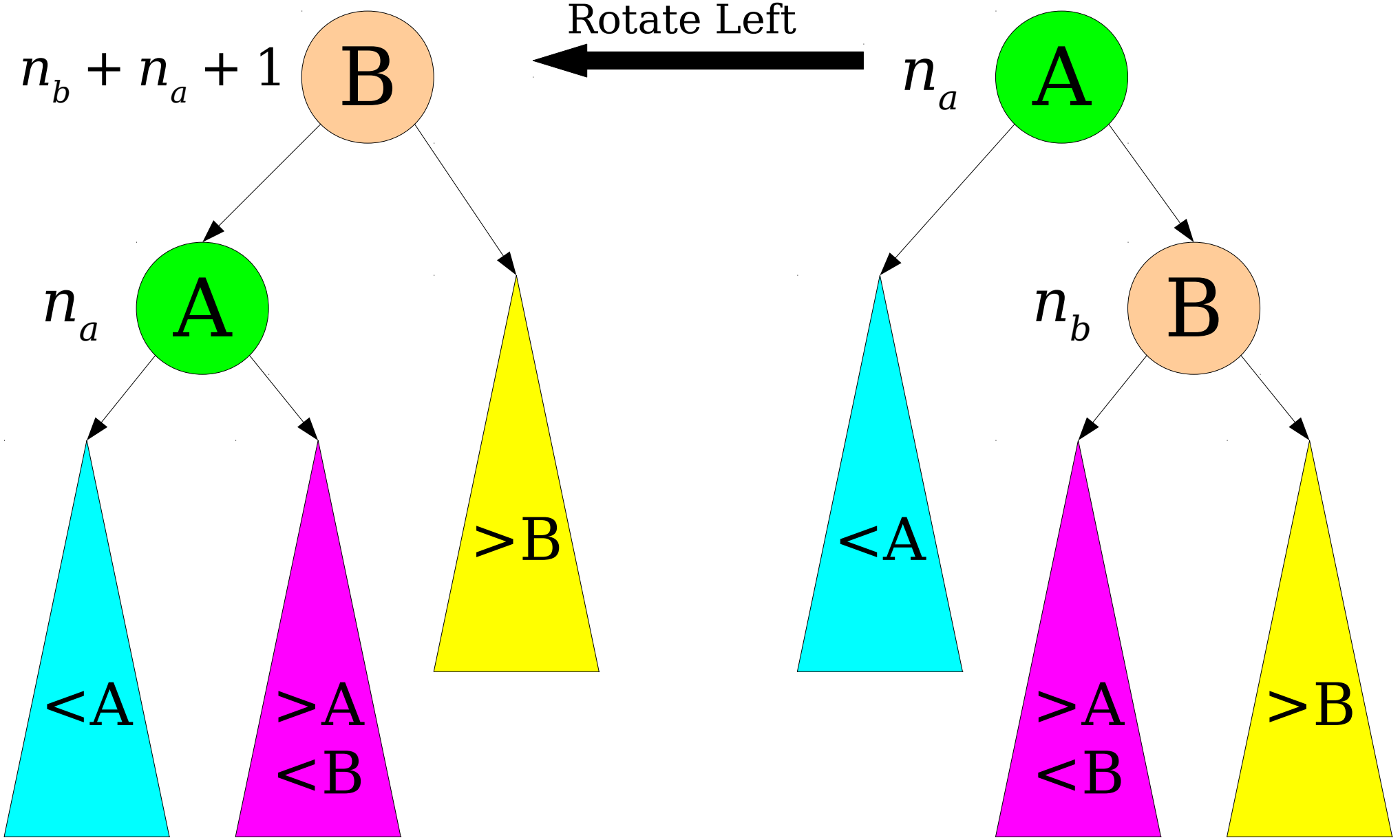


How do we update the numbers after the rotation?

Rotations and Order Statistics



Rotations and Order Statistics



Order Statistic Trees

- The tree we just saw is called an ***order statistic tree***.
- Structurally, it's a red/black tree where each node a count of the nodes in the left subtree.
- Only O(log n) values must be updated on an insertion or deletion and each can be updated in time O(1).
- Supports all BST operations plus ***select*** (find kth order statistic) and ***rank*** (tell index of value) in time O(log n).

Generalizing our Idea

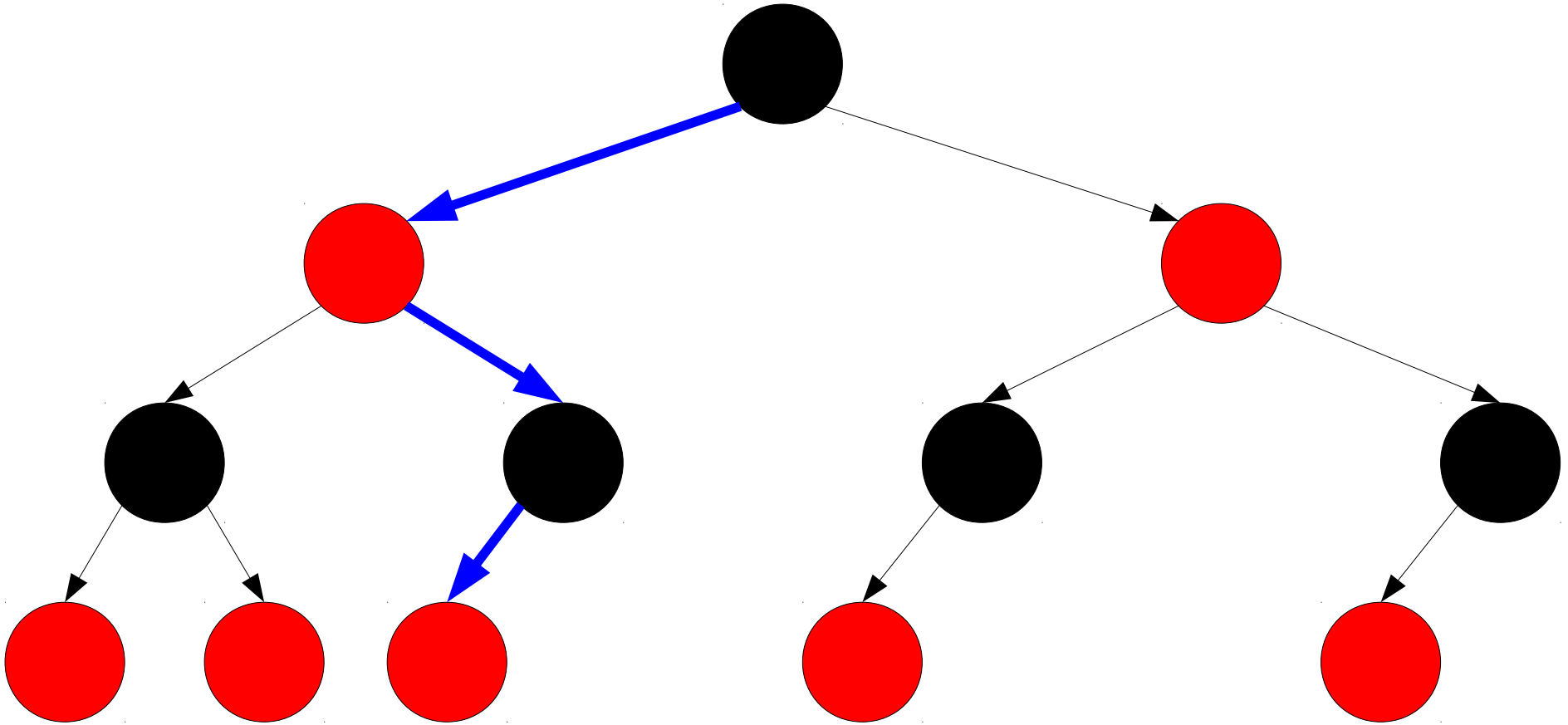
The General Pattern

- This data structure works in the appropriate time bounds because values only change on an insertion or deletion
  - along the root-leaf access path, and
  - during rotations.
- Red/black trees have height O(log n) and require only O(log n) rotations per insertion or deletion.
- We can augment red/black trees with any attributes we'd like as long as they obey these properties.

Augmented Red/Black Trees

- Let $f(node)$ be a function with the following properties:
  - f can be computed in time O(1).
  - f can be computed at a node based purely on that node's key and the values of f computed at *node*'s children.
- ***Theorem:*** The values of f can be cached in the nodes of a red/black tree without changing the asymptotic runtime of insertions or deletions.
- ***Proof sketch:*** After inserting or deleting a node, the only values that need to change are along the root-leaf access path, plus values at nodes that were rotated. There are only O(log n) of these.

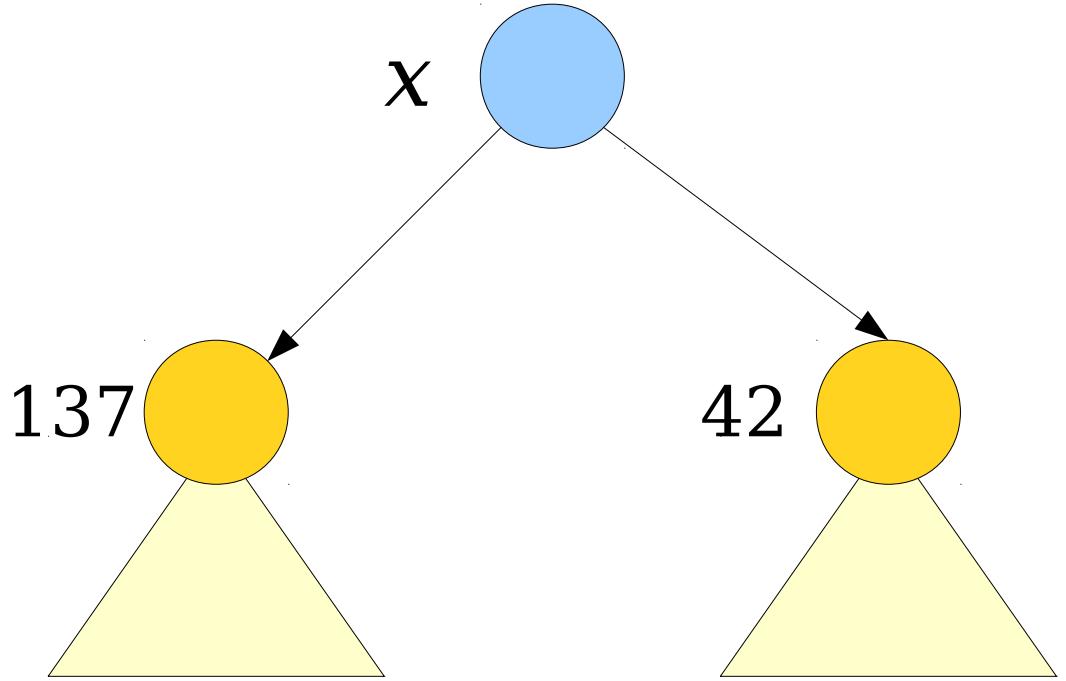
Augmented Red/Black Trees



f can be computed at a node based purely on the key in that node and the values of *f* in it that node's children.

Order Statistics

- ***Note:*** The approach we took for building order statistic trees does not fall into this framework.
- ***Example:*** The values below denote the number of nodes in the indicated nodes' left subtrees. What is the correct value of x?



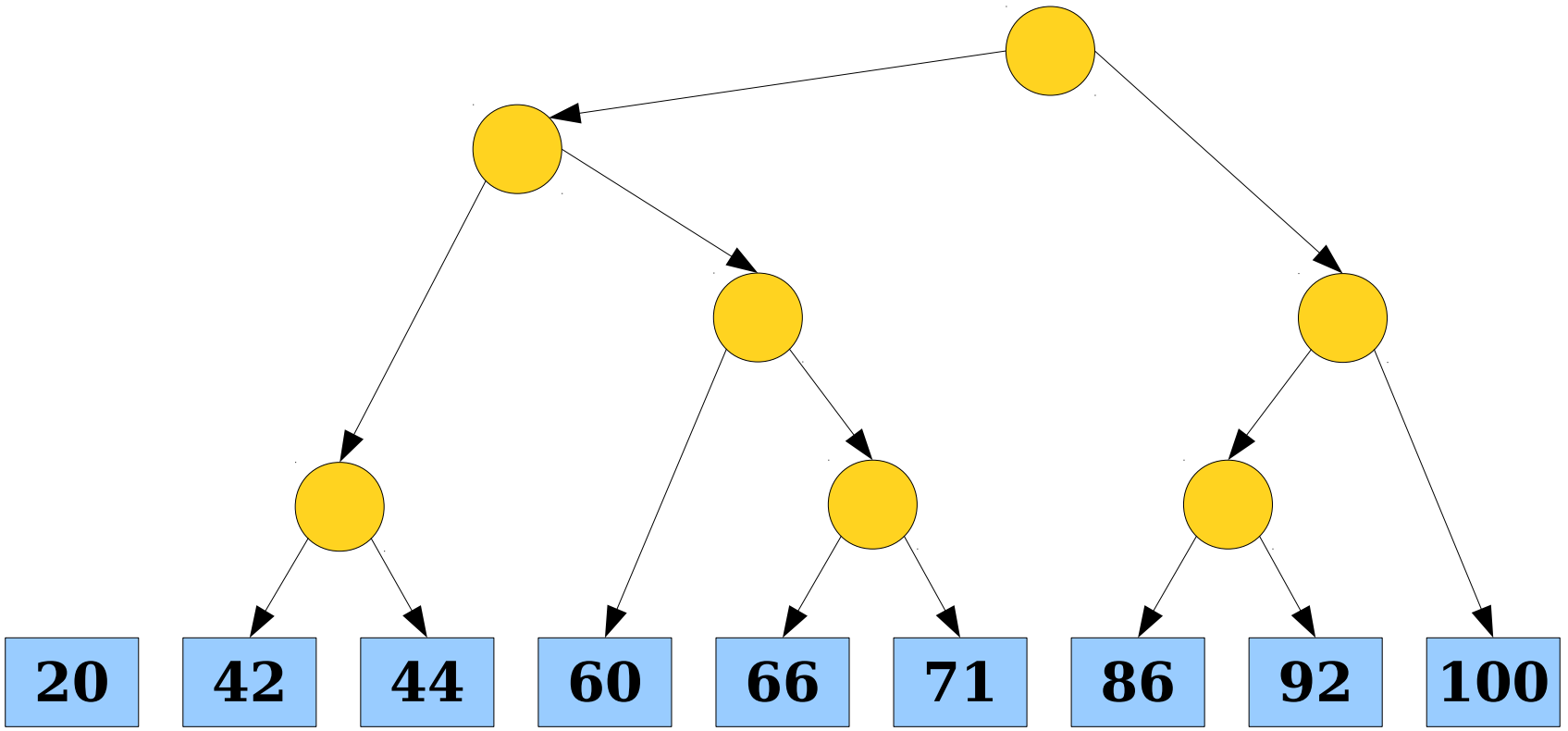
Order Statistics via Augmentation

- Have each node store three quantities:
  - *numLeft,* the number of nodes in the left subtree.
  - *numRight,* the number of nodes in the right subtree.
  - *numTotal,* the total number of nodes in the subtree.
- Can compute this information at a node in time O(1) based on subtree values:
  - *node.numLeft* = *node.left.numTotal*
  - *node.numRight* = *node.right.numTotal*
  - *node.numTotal* = 1 + *node.numLeft* + *node.numRight*
- Therefore, using the augmented BST framework, can compute subtree sizes.
- No need to reason about tree rotations!

Example: ***Dynamic 1D Closest Points***

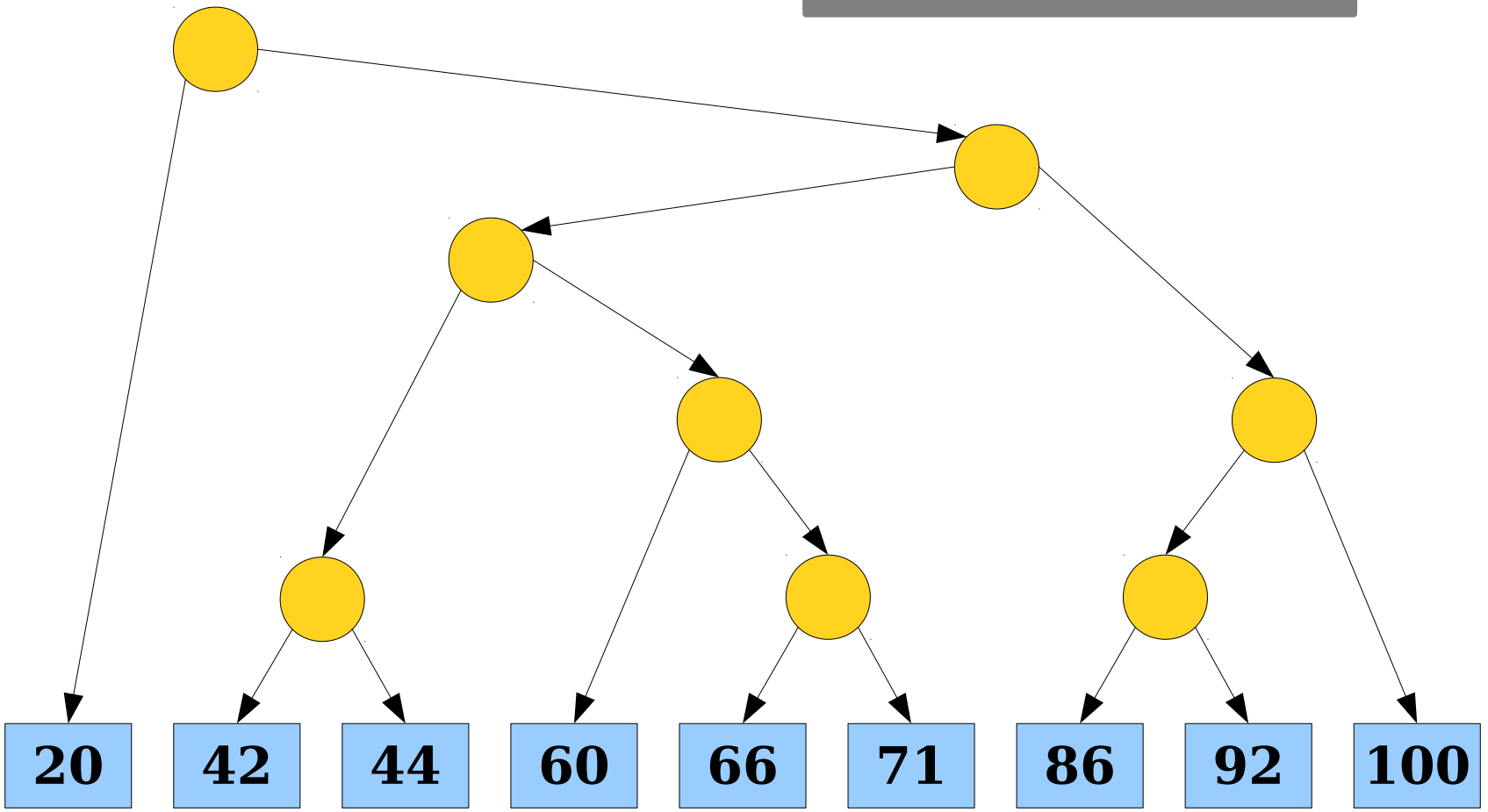
1D Hierarchical Clustering

64.56								
20	42	44	60	66	71	86	92	100



1D Hierarchical Clustering

This tree is called a ***dendrogram***.



Analyzing the Runtime

- How efficient is this algorithm?
  - Number of rounds: $\Theta(n)$.
  - Work to find closest pair: $O(n)$.
  - Total runtime: $\boldsymbol{\Theta(n^2)}$.
- Can we do better?

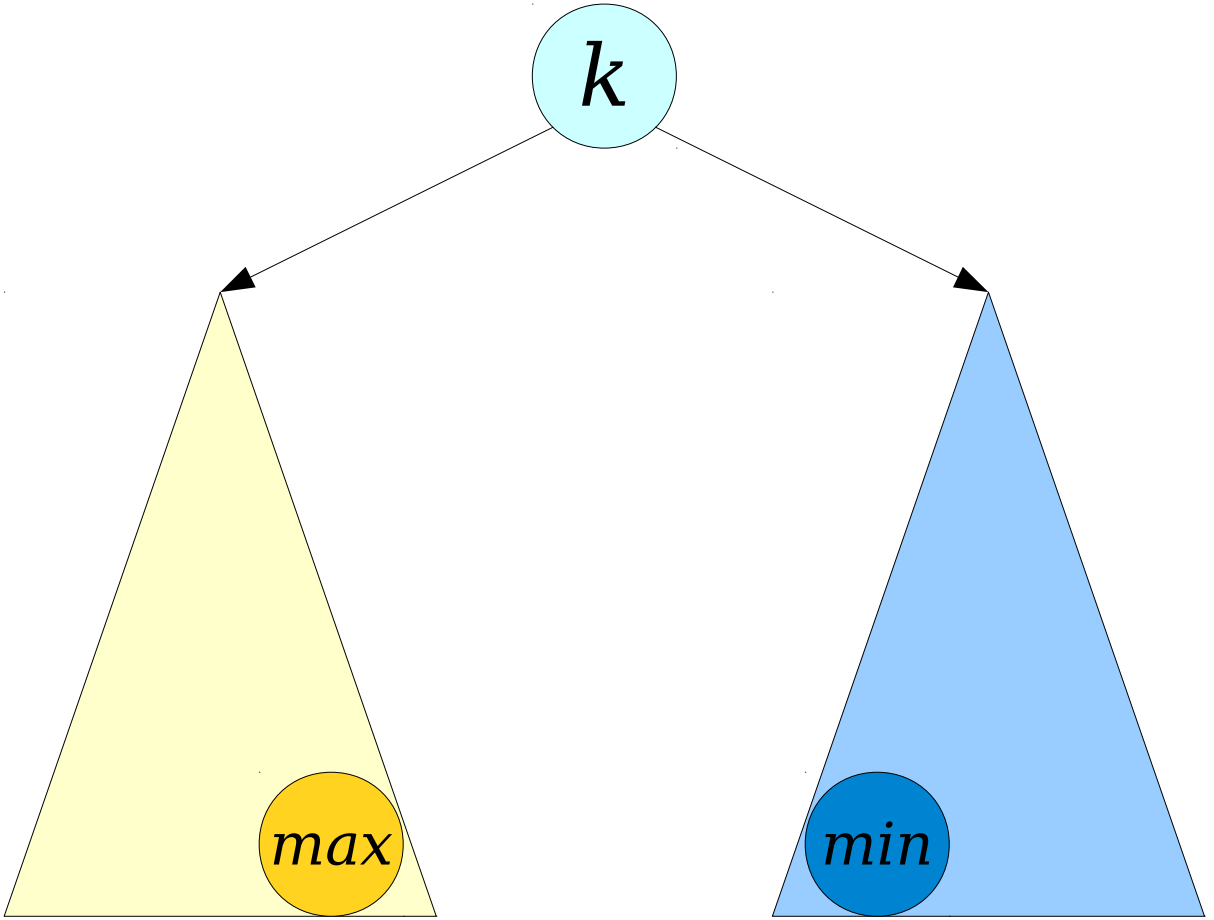
Dynamic 1D Closest Points

- The ***dynamic 1D closest points problem*** is the following:

  Maintain a set of elements undergoing insertion and deletion while efficiently supporting queries of the form "what is the closest pair of points?"

- Can we build a better data structure for this?

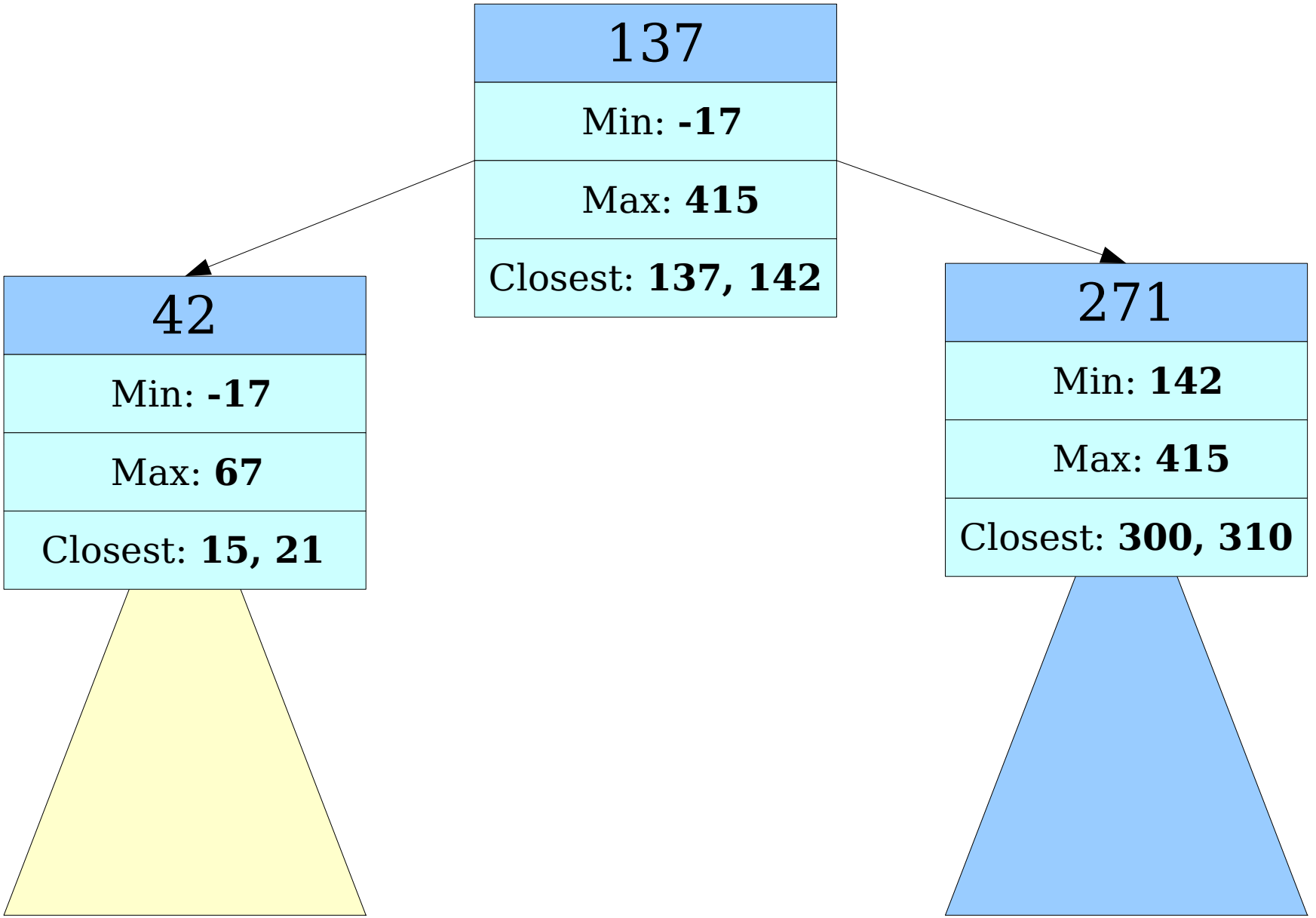
Dynamic 1D Closest Points



A Tree Augmentation

- Augment each node to store the following:
  - The maximum value in the tree.
  - The minimum value in the tree.
  - The closest pair of points in the tree.
- ***Claim:*** Each of these properties can be computed in time O(1) from the left and right subtrees.
- These properties can be augmented into a red/black tree so that insertions and deletions take time O(log n) and “what is the closest pair of points?” can be answered in time O(1).

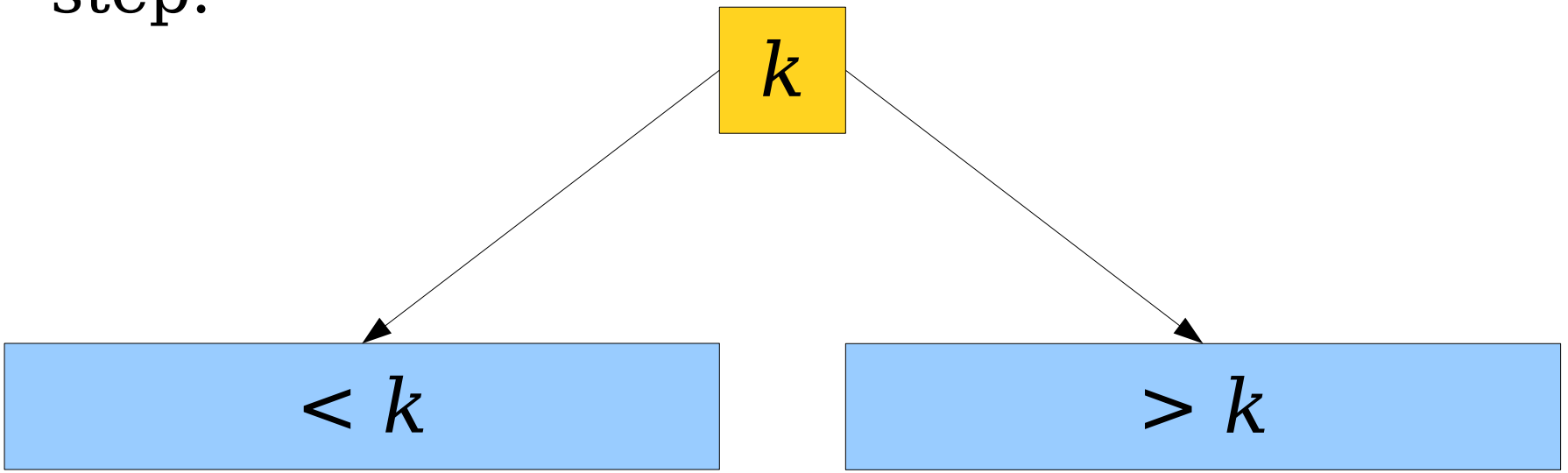
Dynamic 1D Closest Points



A Helpful Intuition

Divide-and-Conquer

- Initially, it can be tricky to come up with the right tree augmentations.
- ***Useful intuition:*** Imagine you're writing a divide-and-conquer algorithm over the elements and have O(1) time per “conquer” step.



Time-Out for Announcements!

Problem Set Three

- Problem Set Two was due today at 3:00PM.
- Problem Set Three goes out now. It's due next Thursday, April 28th, at the start of lecture.
  - Explore advanced tree operations, augmented search trees, and data structure isometries!
  - As always, feel free to ask questions on Piazza or to stop by office hours.
- Start this one early; Q3 has a couple of tricky parts and Q4 will require you to do a bit of independent reading.

Apply to Section Lead!

- Section leading applications are now open for Autumn quarter.
- Apply online at

  https://cs198.stanford.edu/cs106/Apply.aspx

  by **Thursday, April 28** at **11:59PM**.
- Highly recommended – this is one of the best programs the CS department offers.

BASES presents:

Founders leading the charge for **women in entrepreneurship** invite you to sit at the table.

1 Hear them speak

Danae Ringelmann
Co-Founder & CDO,
Indiegogo

Mike Cassidy
VP GoogleX,
Project Loon

Elizabeth Douglas
COO,
WikiHow

Lisa Sugar
Founder,
Popsugar

Selina Tobaccowala
President & CTO,
Surveymonkey

Eurie Kim
Partner,
Forerunner Ventures

Lauren Imparato
Founder,
I.AM.YOU.

Shannon Wu
Director of Marketing,
founder.org

Rebecca Lynn
Partner,
Canvas

2 **Engage in real-world, hands-on problem solving workshops** with a group of empowering peers and successful entrepreneurs. **All genders welcome!**

bWIE

BASES WIE Summit
April 30. 1-4pm. Huang.
Apply before April 19
at bit.ly/BASESWomenSummit

STANFORD WOMEN IN BUSINESS

{w}

Richard Tapia Conference

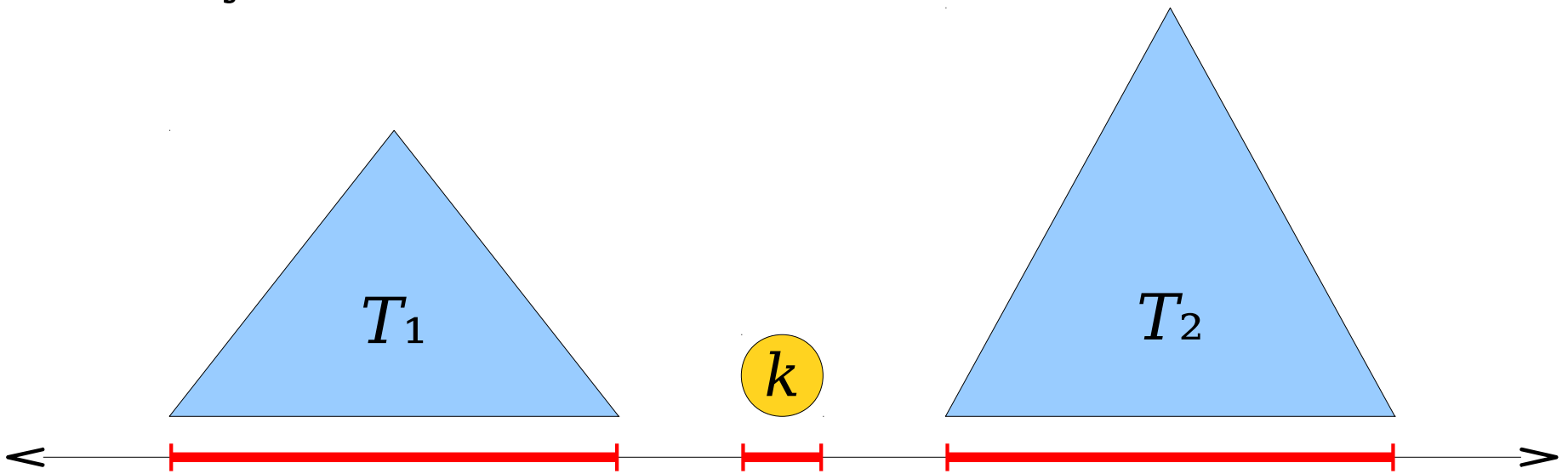
- Stanford will be sponsoring a number of students to attend the ACM Richard Tapia Celebration of Diversity in Computing.
- The conference is September 14 – 17 in Austin, Texas.
- Interested? Apply for sponsorship using ***this link***.

Back to CS166!

Join and Split

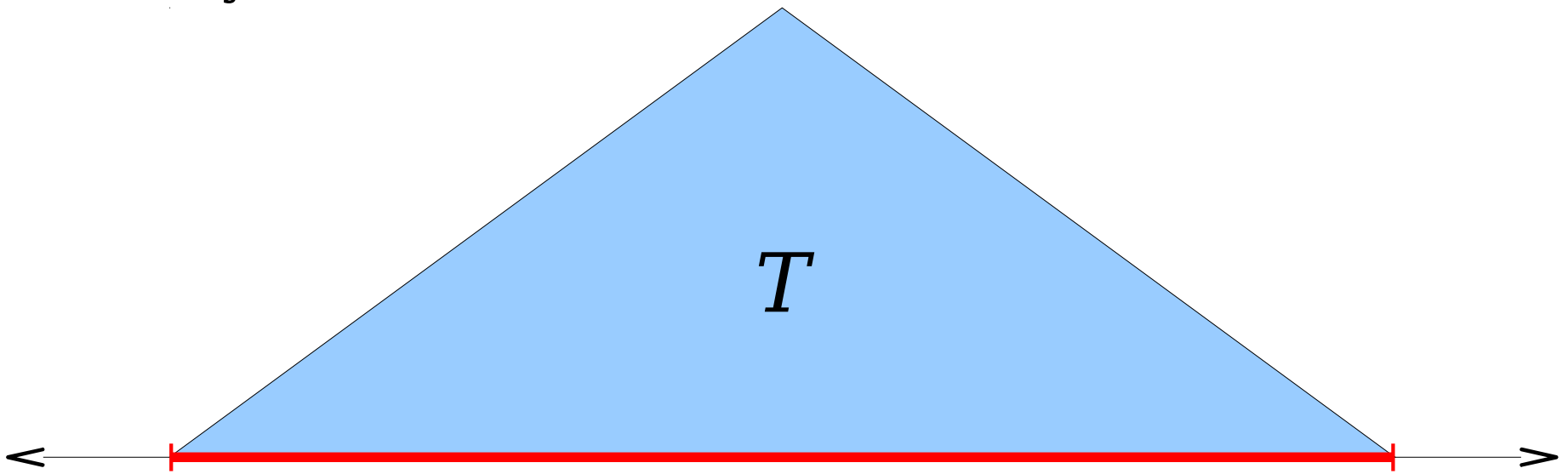
Joining Trees

- ***join***(T_1, k, T_2) takes in two BSTs T_1 and T_2 and a key k. The assumption is that all keys in T_1 are less than k and all keys in T_2 are greater than k.
- ***join***(T_1, k, T_2) destructively modifies T_1 and T_2 to produce a new BST containing all keys in T_1 and T_2 and the key k.



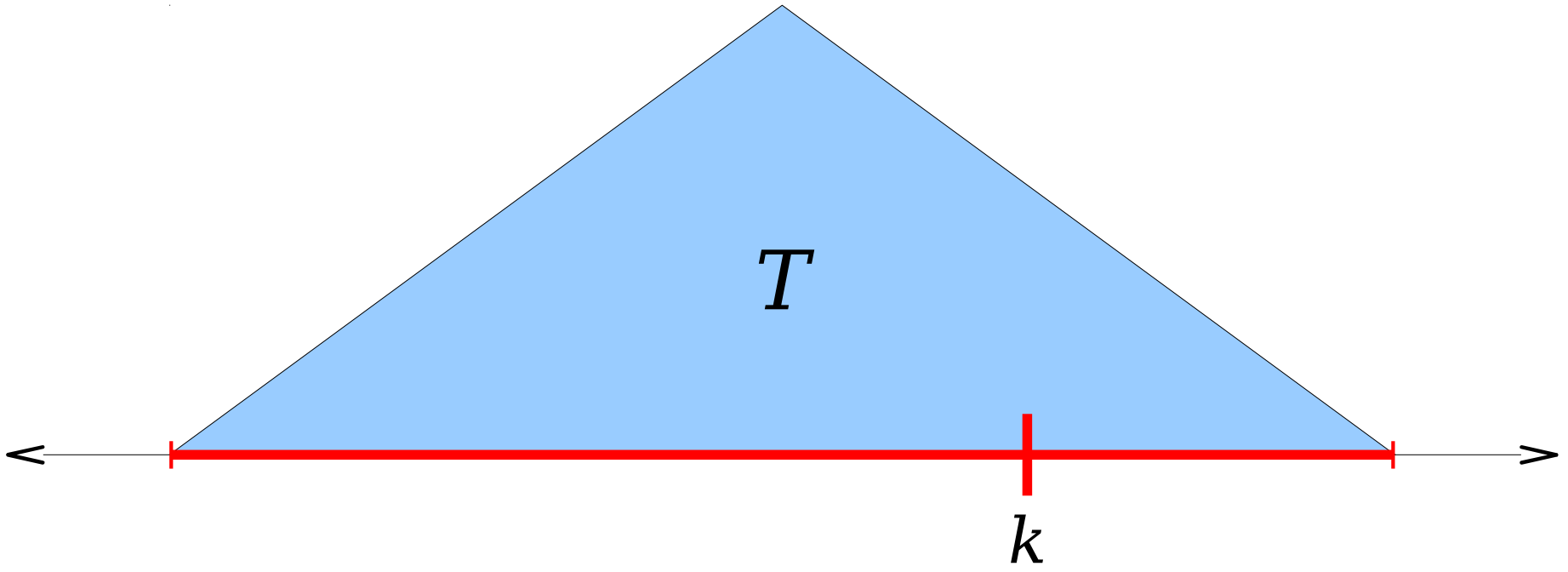
Joining Trees

- ***join***(T_1, k, T_2) takes in two BSTs T_1 and T_2 and a key k. The assumption is that all keys in T_1 are less than k and all keys in T_2 are greater than k.
- ***join***(T_1, k, T_2) destructively modifies T_1 and T_2 to produce a new BST containing all keys in T_1 and T_2 and the key k.



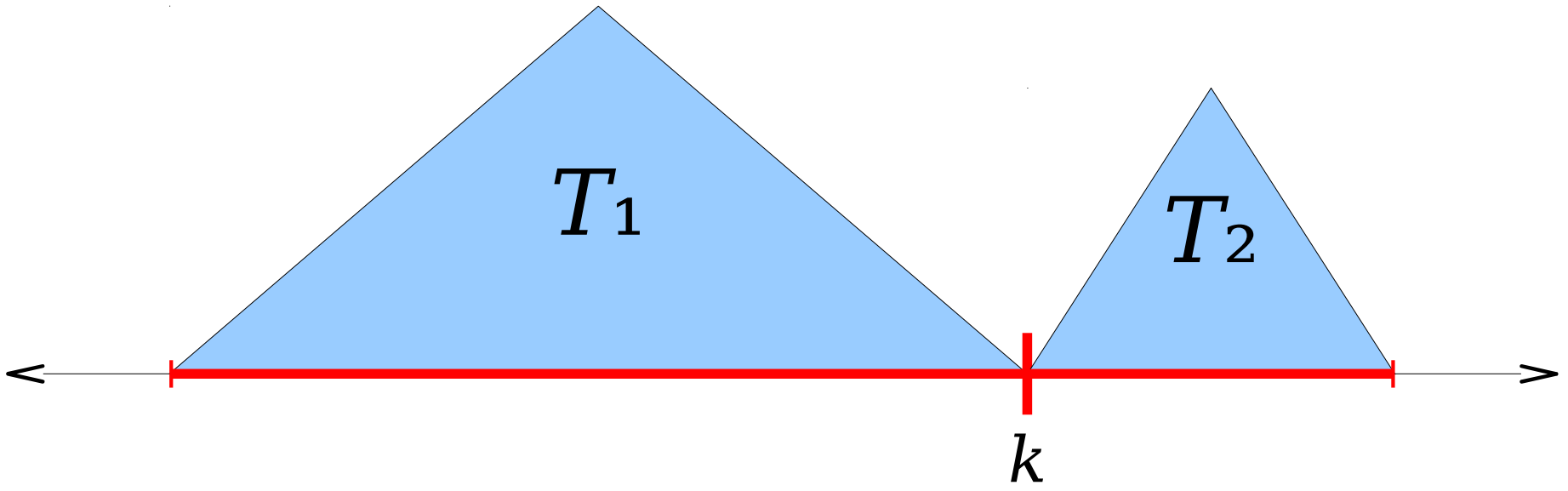
Splitting Trees

- ***split***(T, k) destructively modifies BST T by producing two new BSTs T_1 and T_2 such that all keys in T_1 are less than or equal to k and all keys in T_2 are greater than k.



Splitting Trees

- ***split***(T, k) destructively modifies BST T by producing two new BSTs T_1 and T_2 such that all keys in T_1 are less than or equal to k and all keys in T_2 are greater than k.



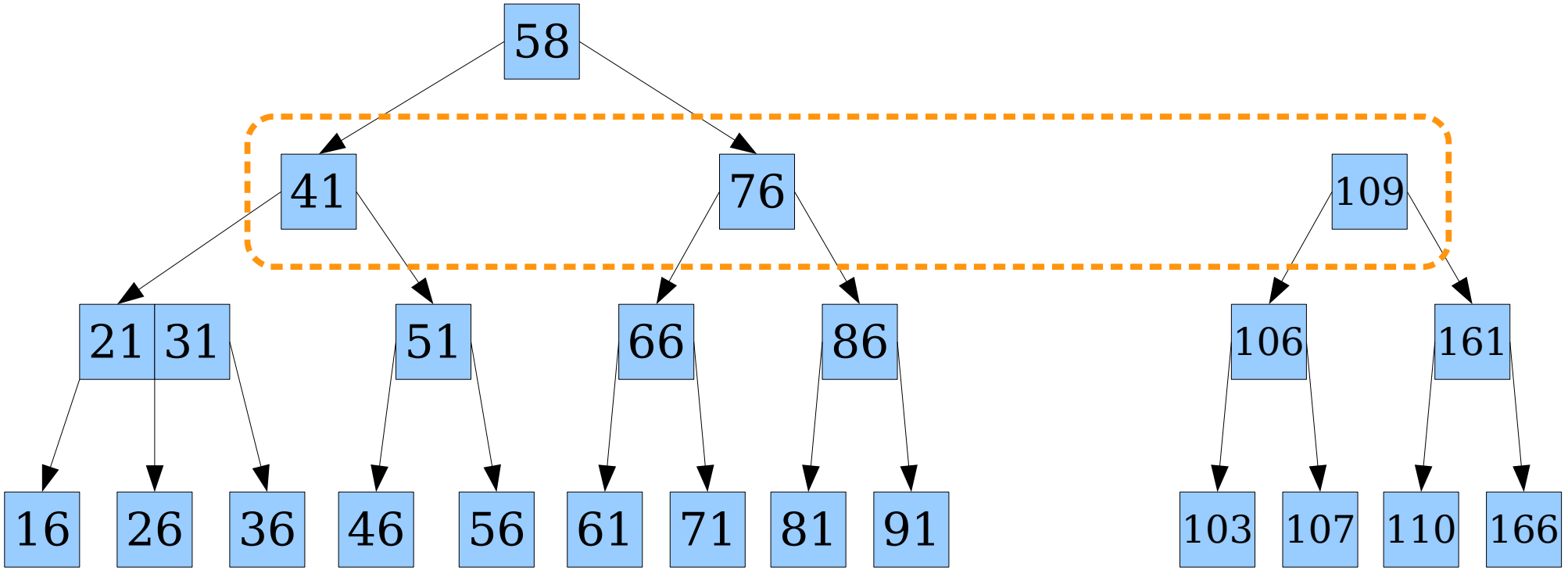
The Runtimes

- Both of these operations can be implemented in time O(n) by completely rebuilding the trees from scratch.
  - Good exercise: determine how to do this.
- Amazingly, with the right augmentations:
  - ***join***(T_1, k, T_2) can be made to run in time $\Theta(1 + |h_1 - h_2|)$, where h_1 and h_2 are the heights of T_1 and T_2, respectively.
  - **split**(T, k) can be made to run in time O(log n).
- How is this possible?

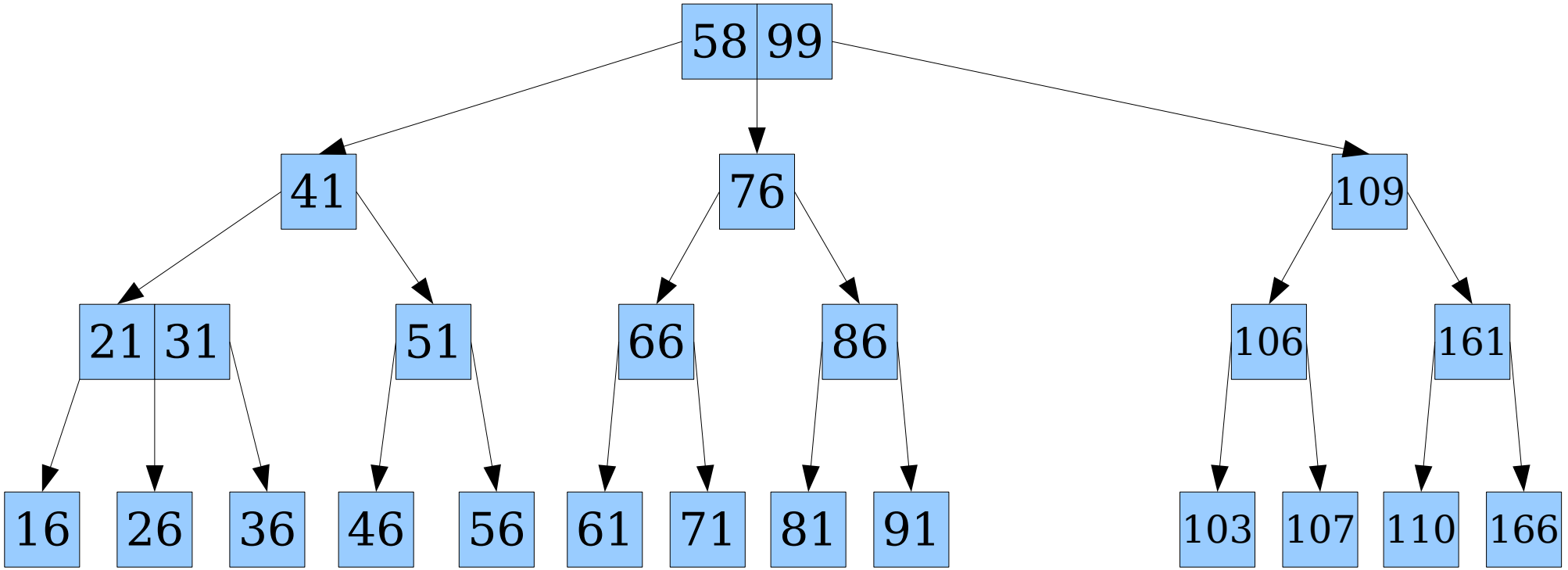
Joining 2-3-4 Trees

- The isometry between 2-3-4 trees and red/black trees is very useful here.
- Let's see how to ***join*** two 2-3-4 trees and a key together.
- Based on what we find, we'll develop an efficient algorithm for joining red/black trees.

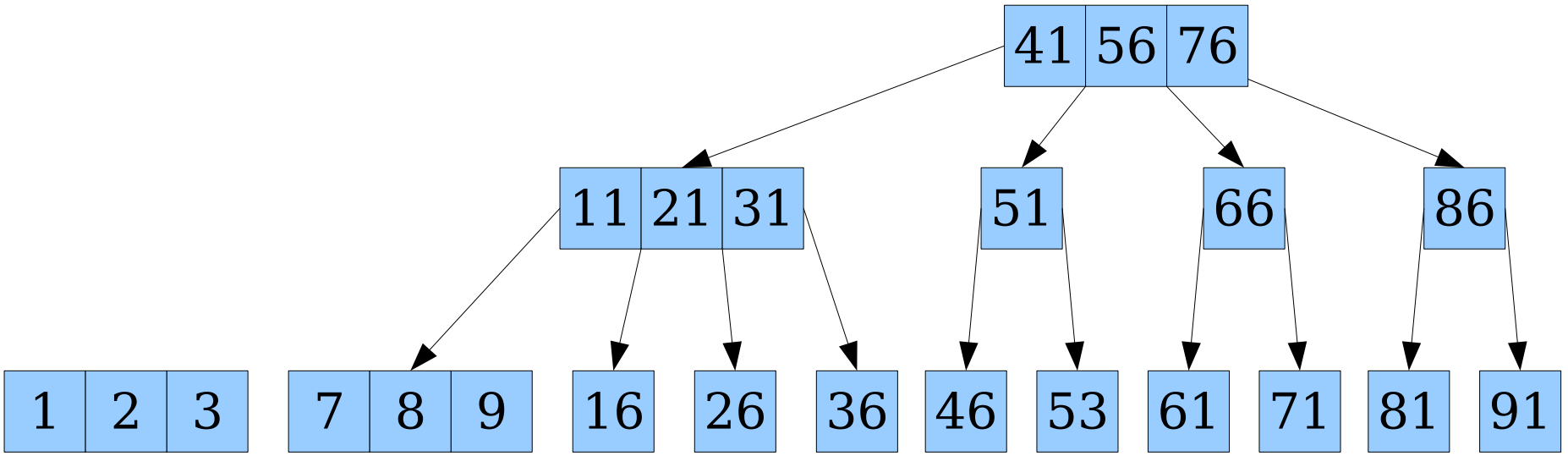
Joining 2-3-4 Trees



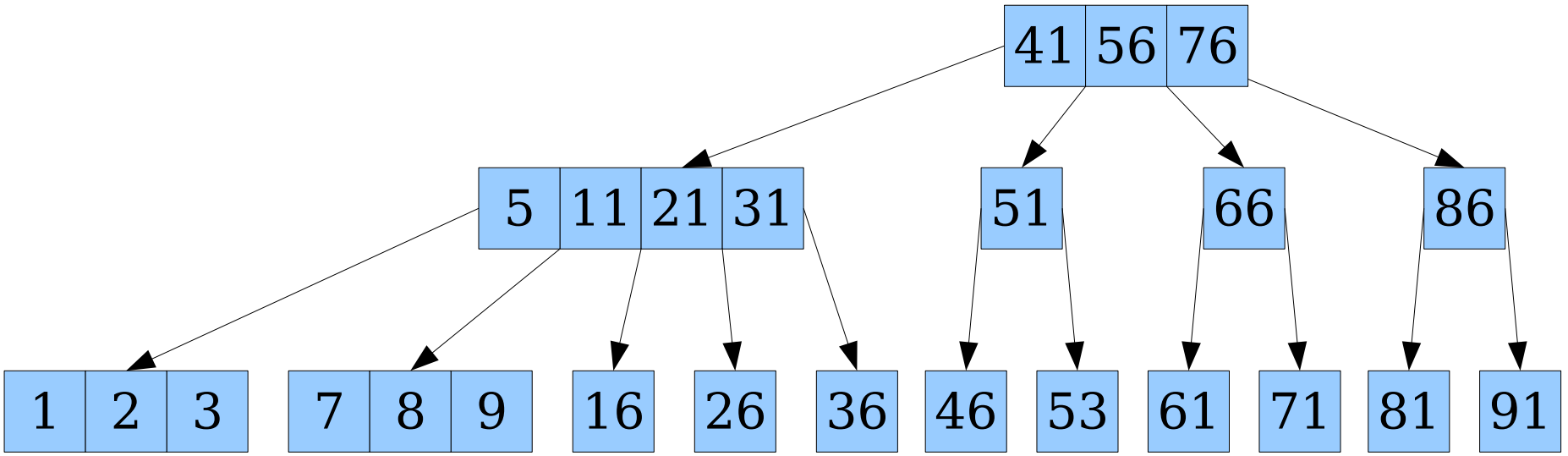
Joining 2-3-4 Trees



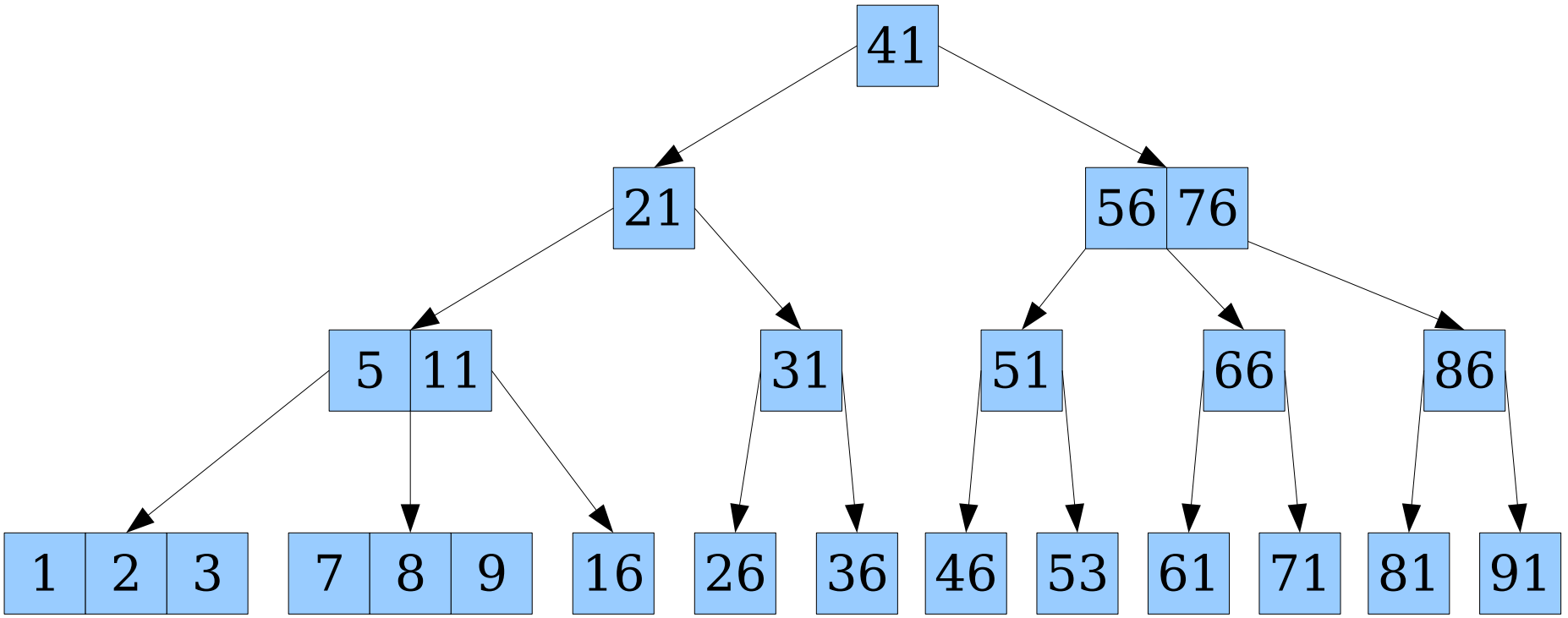
Joining 2-3-4 Trees



Joining 2-3-4 Trees



Joining 2-3-4 Trees

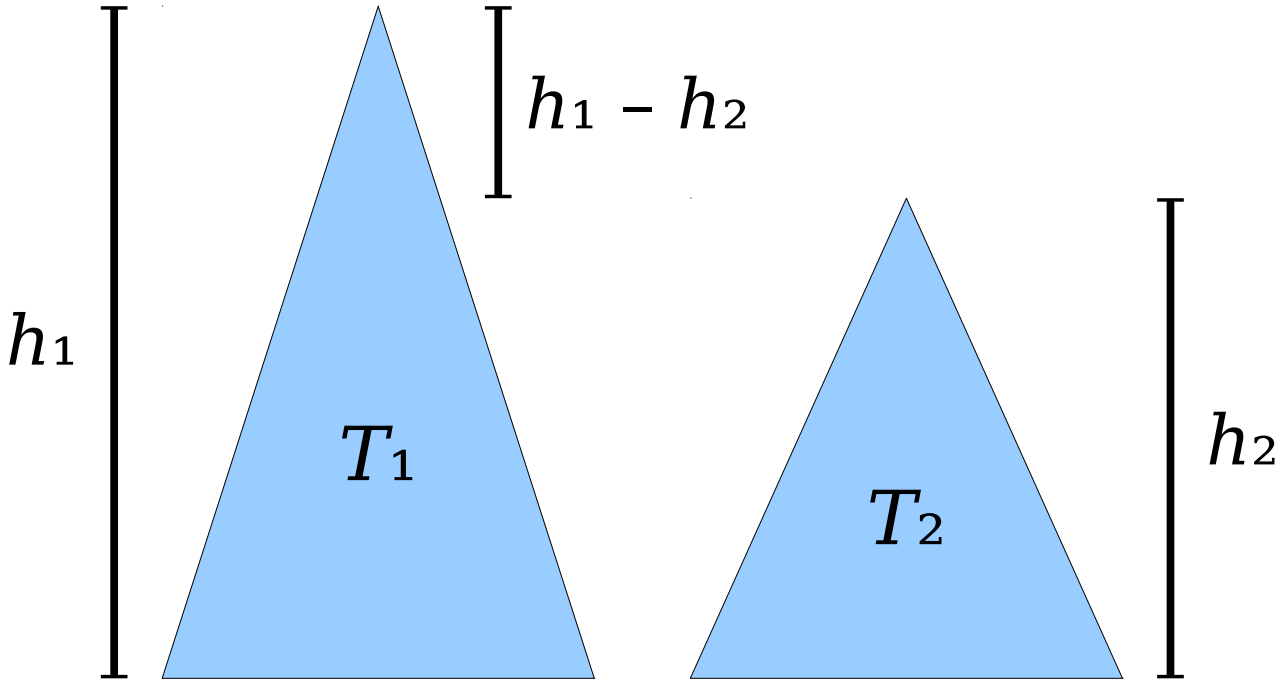


Joining 2-3-4 Trees

- To ***join***(T_1, k, T_2):
  - Assume that T_1 is the taller of the two trees; if not, do the following, but mirrored.
  - Walk down the right spine of T_1 until a node v is found whose height is the height of T_2.
  - Add k as a final key of v's parent with T_2 as a right child.
  - Continue as if you were inserting k into v's parent – possibly split the node and propagate upward, etc.

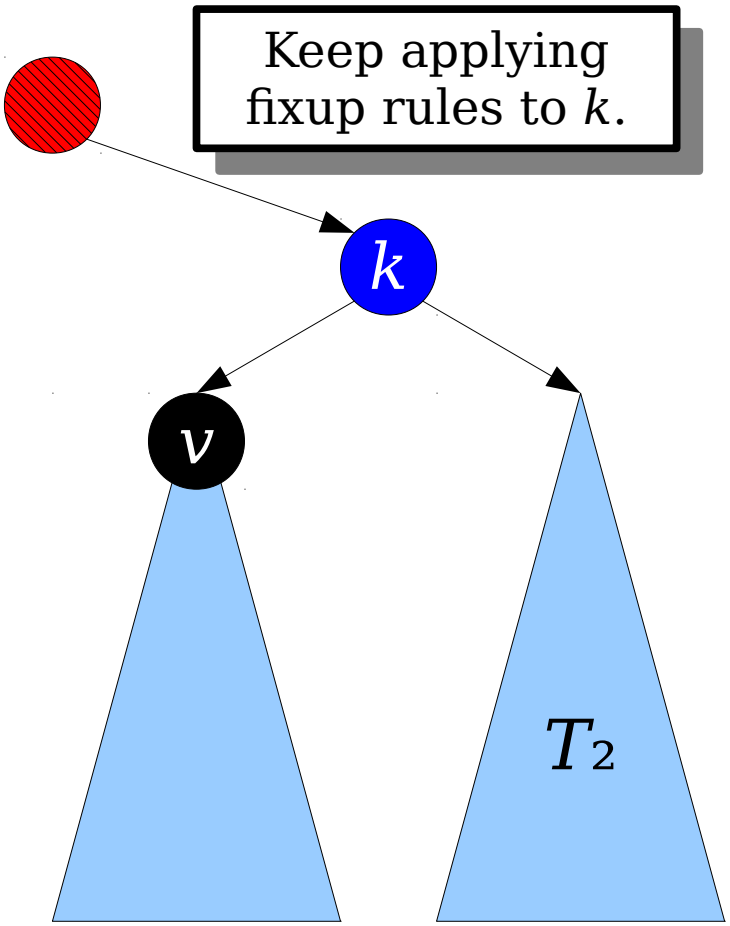
Analyzing the Runtime

- Assume all 2-3-4 tree nodes are annotated with their heights.
- What is the runtime of ***join***(T_1, k, T_2)?
- Runtime is $\boldsymbol{\Theta(1 + |h_1 - h_2|)}$.



Joining 2-3-4 Trees

- Define the ***black height*** of a node to be the number of black nodes on any root-null path starting at that node.
- To ***join***(T_1, k, T_2):
  - Assume that T_1 is the tree with larger black height; if not, do the following, but mirrored.
  - Walk down the right spine of T_1 until a black node v is found whose black height is the black height of T_2.
  - Insert a new node with key k, left child v, and right child T_2
  - Make this new node the right child of v's old parent.
  - Continue as if you had just inserted k.



Runtime Analysis

- Need to augment the red/black tree to store the black height of each node.
  - This fits into our augmentation framework – can be computed from the black heights of the left and right children and from the node's own color.
- Via the isometry with 2-3-4 trees, the runtime is $O(1 + |bh_1 - bh_2|)$.
- Since the black heights of the trees are at most twice the heights of the trees, this runtime is equivalently **$O(1 + |h_1 - h_2|)$**.
- This is $O(\log n_1 + \log n_2)$ in the worst-case.

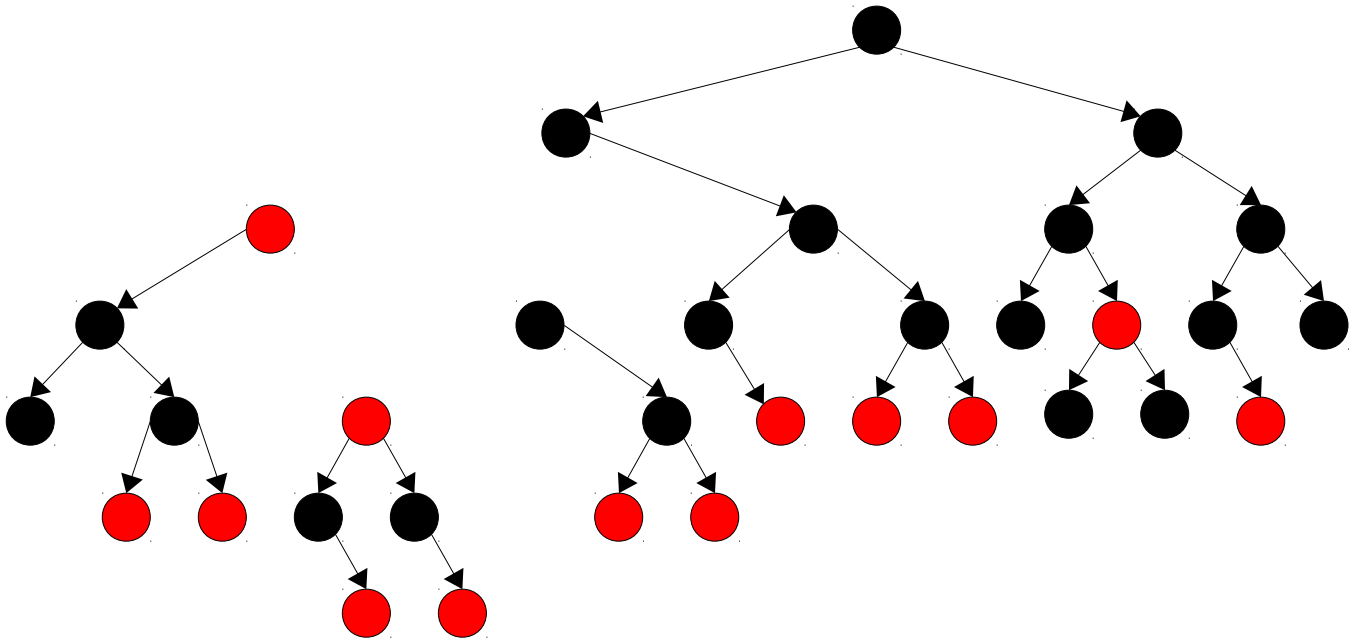
Joining Two Trees

- What if you want to join two red/black trees but don't have a key to join them with?
- Delete the minimum value from the second tree in time O(log n), then use that to join the two trees.

Implementing ***split*** Efficiently

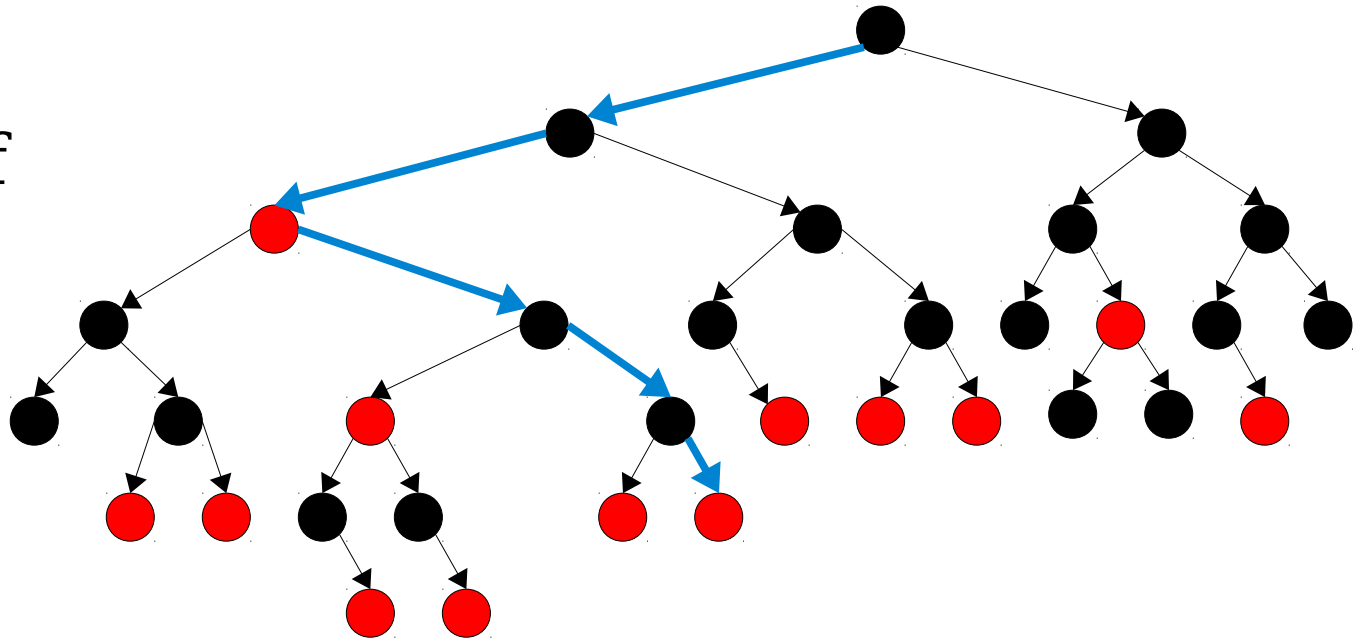
Splitting Trees is Hard

- ***Challenge 1:*** The split procedure might cut the existing tree into lots of smaller pieces.
- ***Challenge 2:*** Cutting a red/black tree into two pieces doesn't necessarily give you two red/black trees.



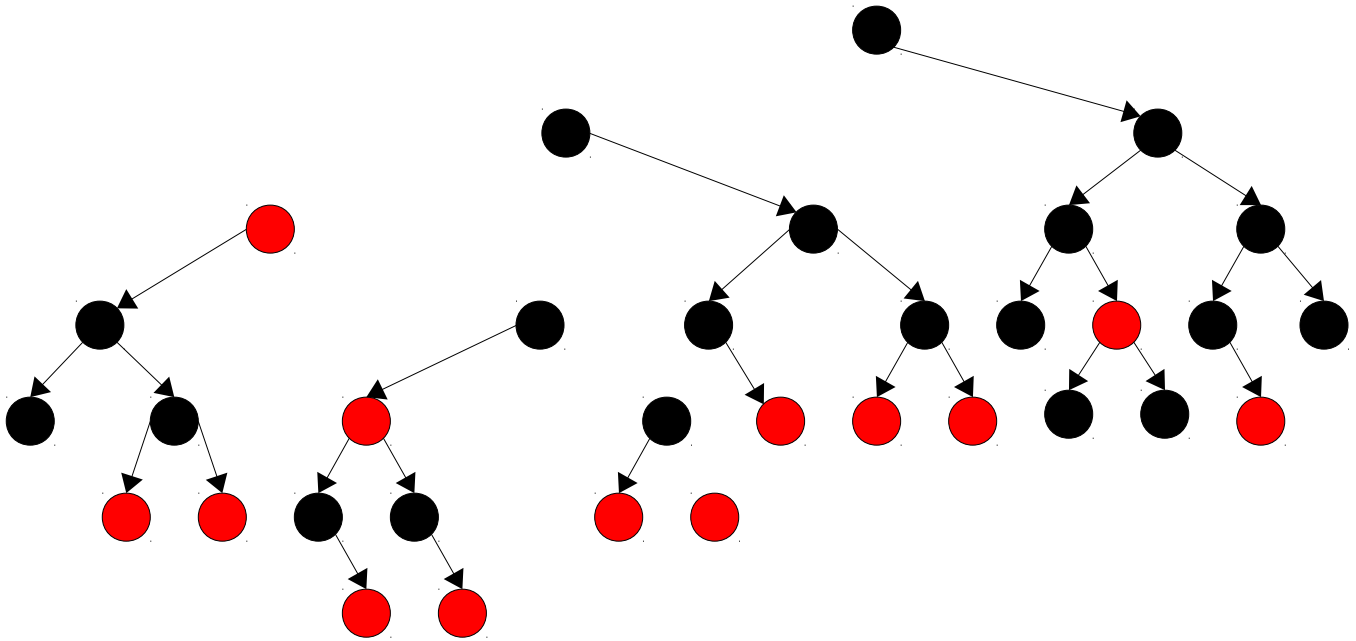
An Observation

- Suppose we want to perform a split on some key k.
- Begin by searching for k. If we find it, search for its inorder successor.
- Cut all links found along the way.



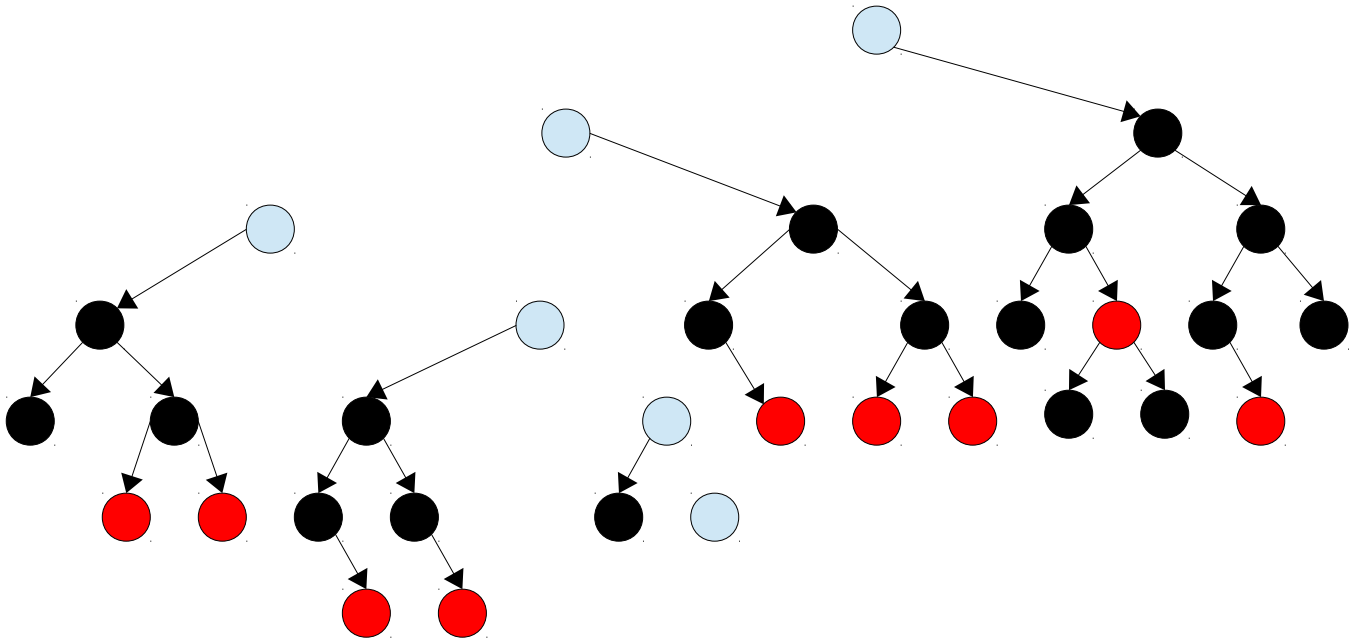
An Observation

- Notice that we're left with a collection of ***pennants***, trees whose roots have just one child.



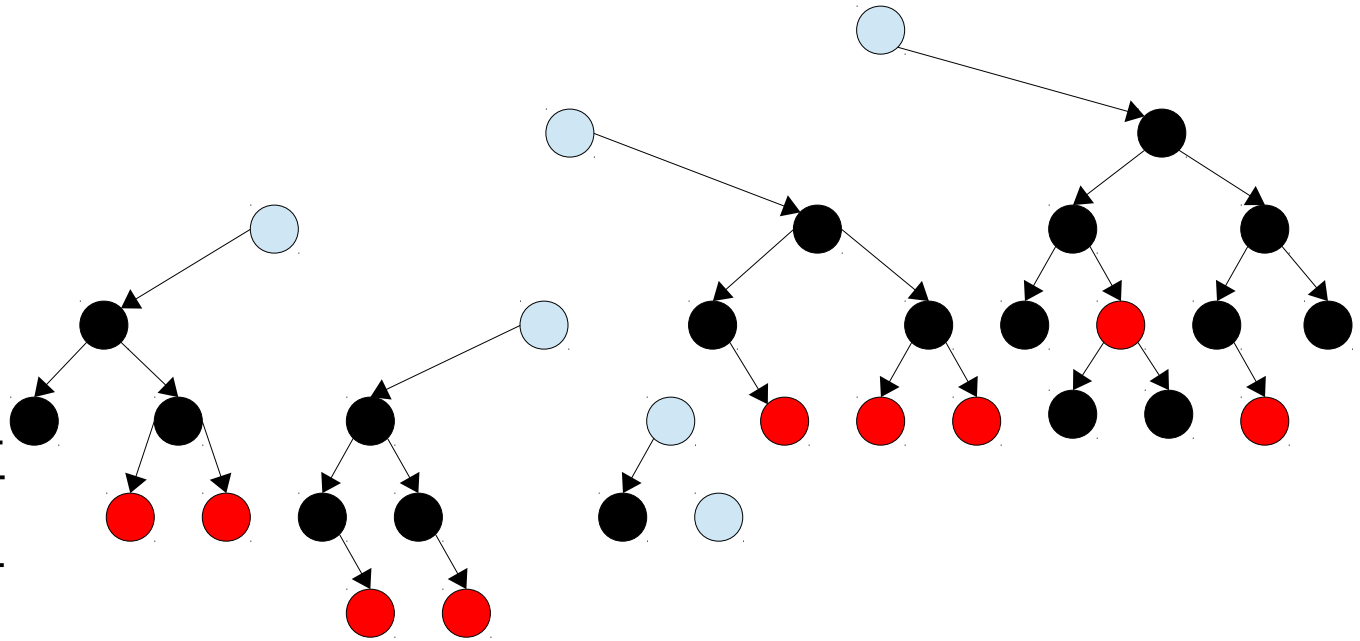
An Observation

- Let's imagine uncoloring all of these pennant roots.
- The trees below them are *almost* red/black trees, but their roots might be red.
- Let's recolor all the roots black.



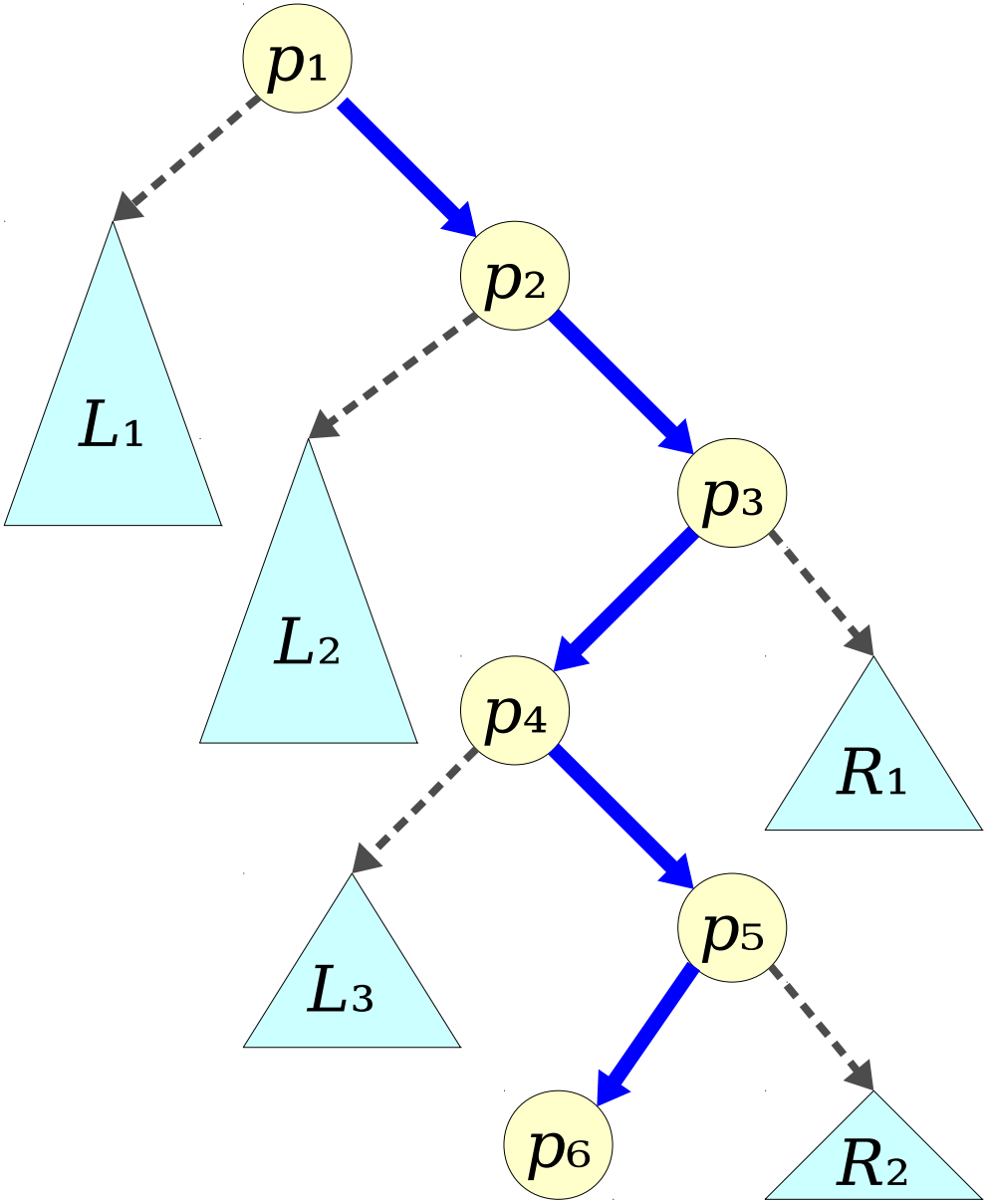
An Observation

- We now have a bunch of red/black trees hanging off of pennants.
- ***Key idea:*** Find a way to join these trees back together to form the two trees we want.



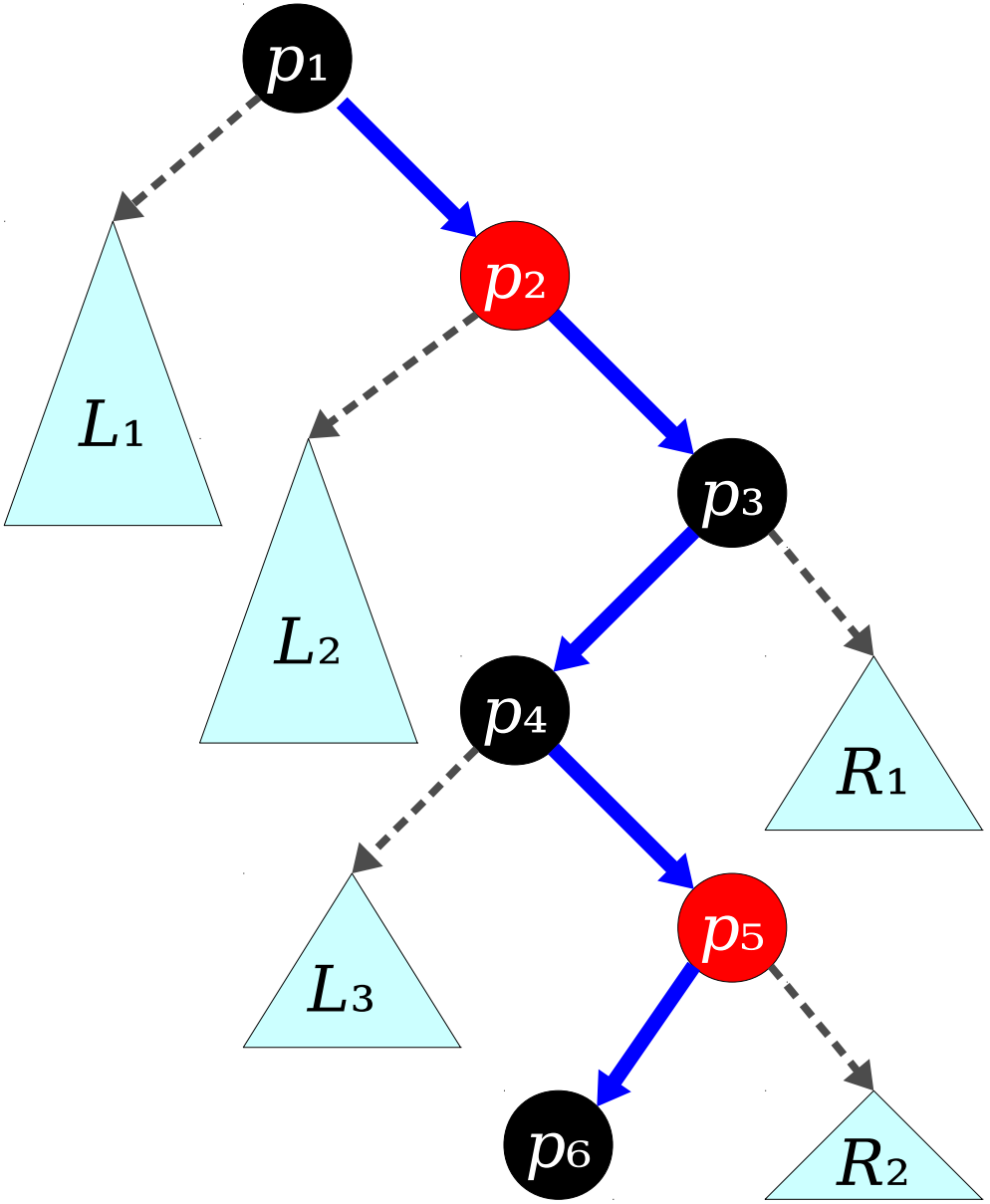
Fleshing Out the Algorithm

- Do a search for the inorder successor of k, cutting each link followed.
- For each pennant, color its child black. We now have a collection of red/black trees hanging off of random nodes.
- Categorize each hanging tree as of type L or type R depending on whether it's a left or right child of its pennant.

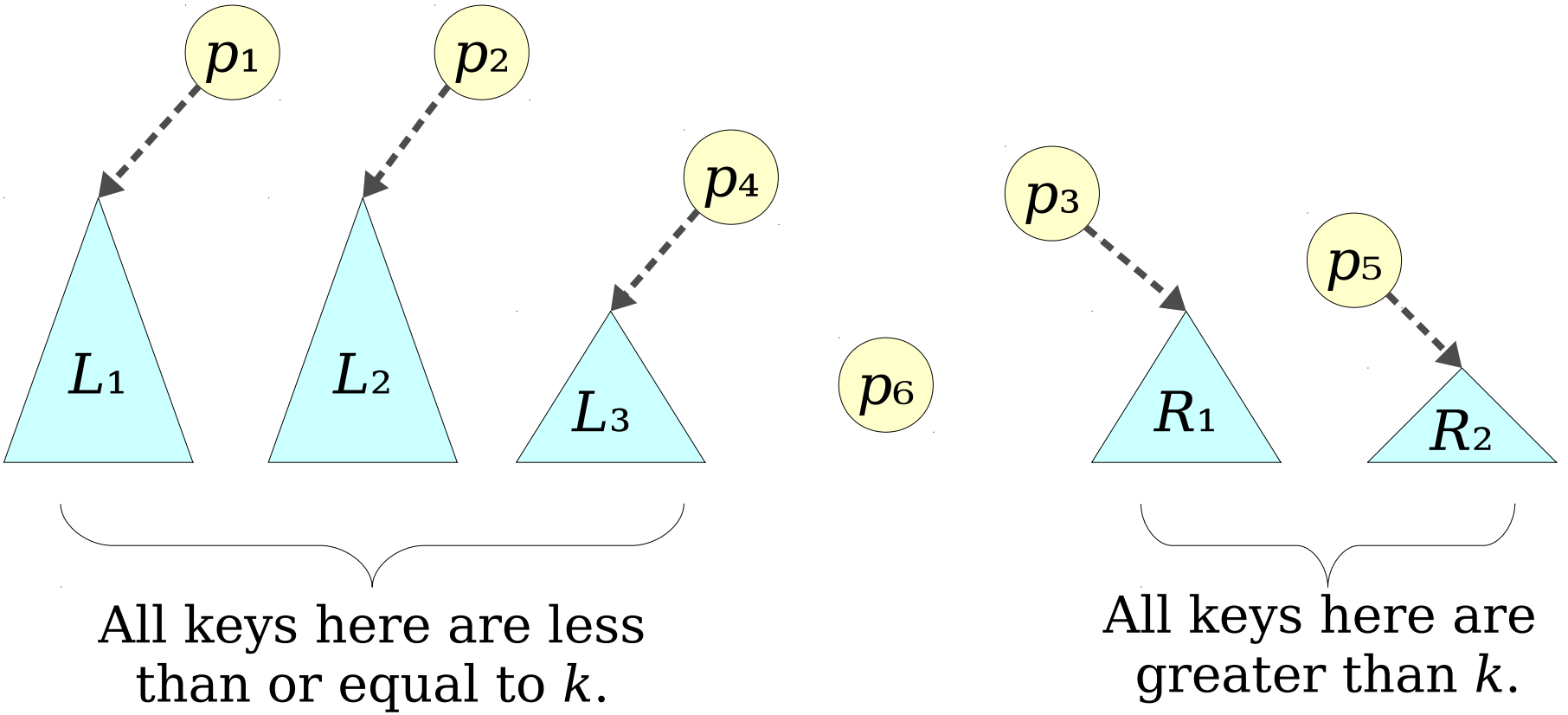


Fleshing Out the Algorithm

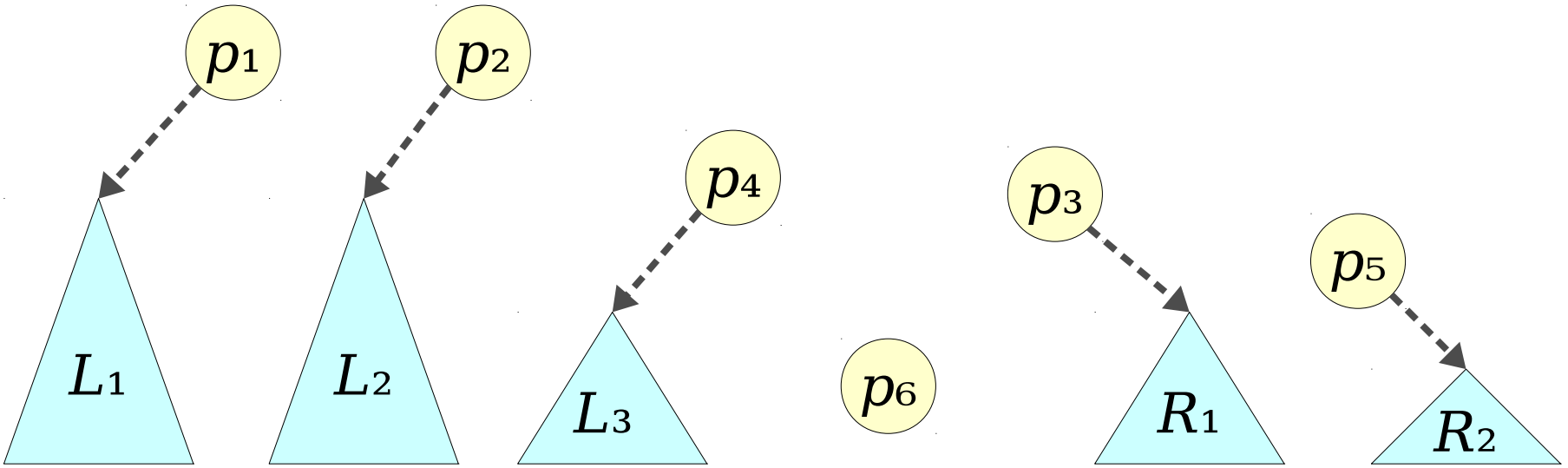
- ***Observation 1:*** Look at any two consecutive L trees or R trees and the root of the pennant of the first tree. Then the key in the pennant root is strictly between all the values of those two trees.
- ***Observation 2:*** There are at most two trees of each black height hanging off of the pennants.



Fleshing Out the Algorithm



Fleshing Out the Algorithm



Key idea: Join all the L trees back together and all the R trees back together, using the nodes at the root of the pennants as the joining key. Because the height differences are low, the runtime works out to O(log n).

Analyzing the Runtime

- Suppose there is one tree of each black height in L.
- What is the runtime of concatenating the trees in reverse order of heights?
- Each join takes time $O(1 + |bh_1 - bh_2|) = O(1)$.
- At most $O(\log n)$ joins (access path has length $O(\log n)$)
- Runtime is **$O(\log n)$**.

Analyzing the Runtime

- Suppose there are trees of some, but not all, heights.
- What is the runtime of concatenating the trees in reverse order of heights?
- Each join takes time $O(1 + bh_{s+1} - bh_s)$
- Summing across all joins:

$$\begin{aligned}\sum_{i=1}^{k-1} O(1+bh_{i+1}-bh_i) &= O\left(\sum_{i=1}^{k-1}(1+bh_{i+1}-bh_i)\right) \\ &= O\left(k+\sum_{i=1}^{k-1}(bh_{i+1}-bh_i)\right) \\ &= O(k+bh_k-bh_1)\end{aligned}$$

- The number of trees (k) is $O(\log n)$ and the maximum black height is $O(\log n)$. Runtime: **$O(\log n)$**.

The Split Algorithm

- Split the tree into L pennants and R pennants, as before.
- Iterate across the pennants in ascending order of heights, ***join***ing each of the corresponding trees together using the pennant node as the join key. This takes time O(log n).
- There will be O(1) leftover pennant nodes. Insert them in time O(log n) into the proper trees.
- Net runtime: **O(log *n*)**.

An Application: ***Flexible Sequences***

Sequence Data Structures

- The two major data structures you're probably used to seeing for sequences are dynamic arrays and linked lists.
- In a dynamic array:
  - Lookups take time O(1).
  - Insertions and deletions take time O(n).
  - Concatenations take time O(n).
- In a linked list:
  - Lookups take time O(n).
  - Insertions and deletions take time O(1) if you know where to insert and O(n) otherwise.
  - Concatenations take time O(1).

Flexible Sequences

- Imagine we store a sequence as a modified order statistic tree.
- We ignore the relative order of the elements and instead use the indices to guide BST lookups.
- Now, insertions, lookups, and deletions all take time O(log n).
- Armed with split and join, we can also concatenate and split sequences in time O(log n) each.
- After filling in the details, you can now manage a sequence of elements with O(log n) insertions, deletions, lookups, concatenations, and splits!

Next Time

- ***Amortized Analysis***
  - Lying about runtime costs in an honest manner.
- ***Frameworks for Amortization***
  - How can we think about assigning costs?
- ***Revisiting Earlier Structures***
  - Queues, Cartesian trees, and 2-3-4 trees.