Balanced Trees

Part Two

Problem Set One due at the start of class. Written assignments can be submitted up front.

Outline for This Week

- **B-Trees**
  - A simple type of balanced tree developed for block storage.
- **Red/Black Trees**
  - The canonical balanced binary search tree.
- **Augmented Search Trees**
  - Adding extra information to balanced trees to supercharge the data structure.
- **Two Advanced Operations**
  - The split and join operations.

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.

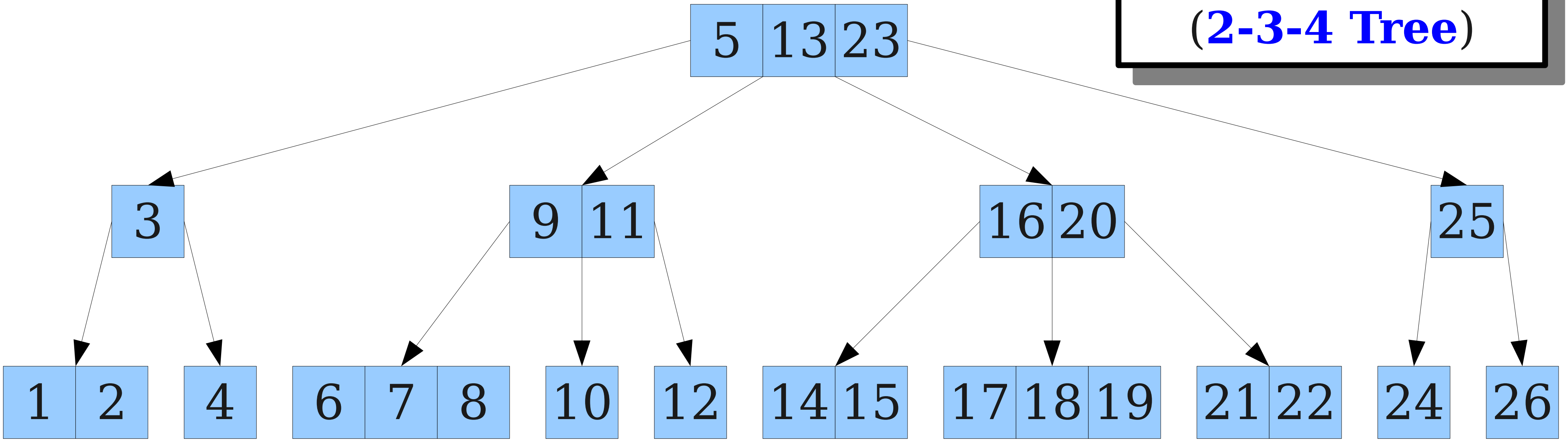
Review from Last Time

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.

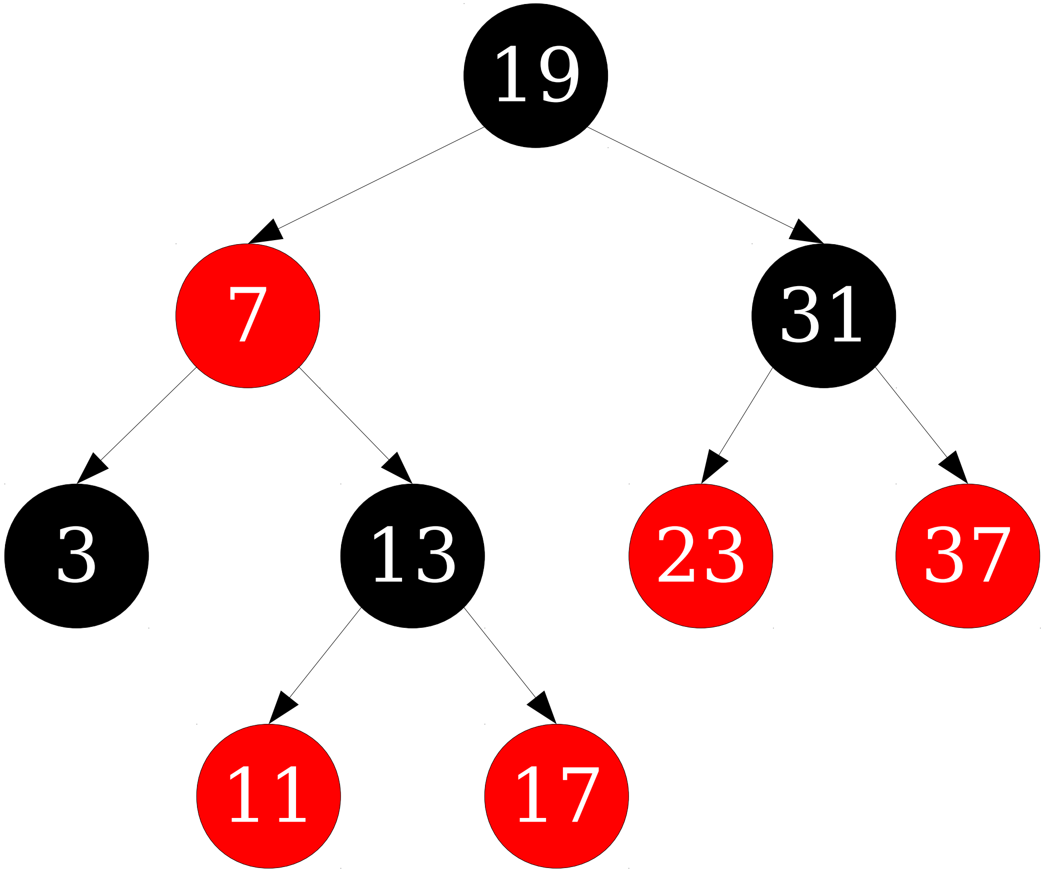
B-tree of order 2

(**2-3-4 Tree**)



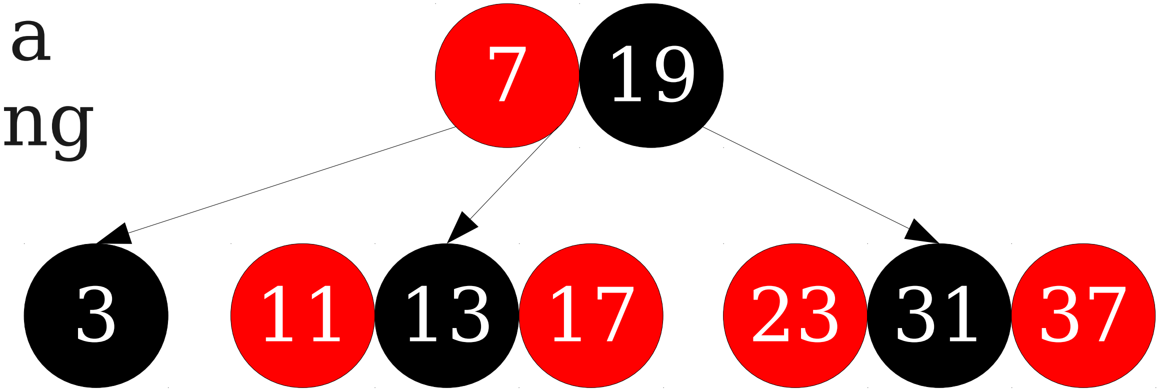
Red/Black Trees

- 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

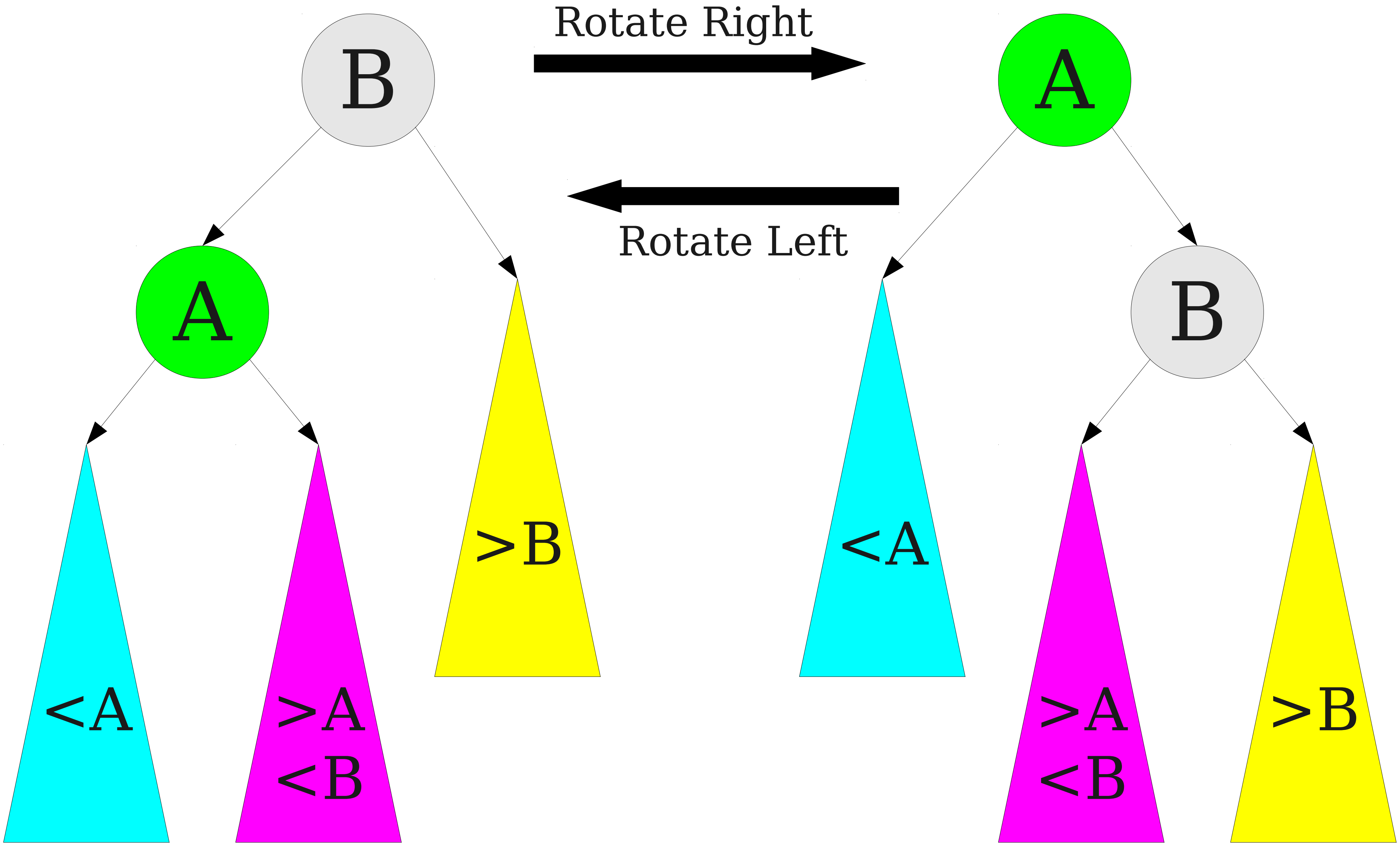
- A **red/black tree** is a BST with the following properties:
  - Every node is either red or black.
  - The root is black.
  - No red node has a red child.
  - Every root-null path in the tree passes through the same number of black nodes.



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

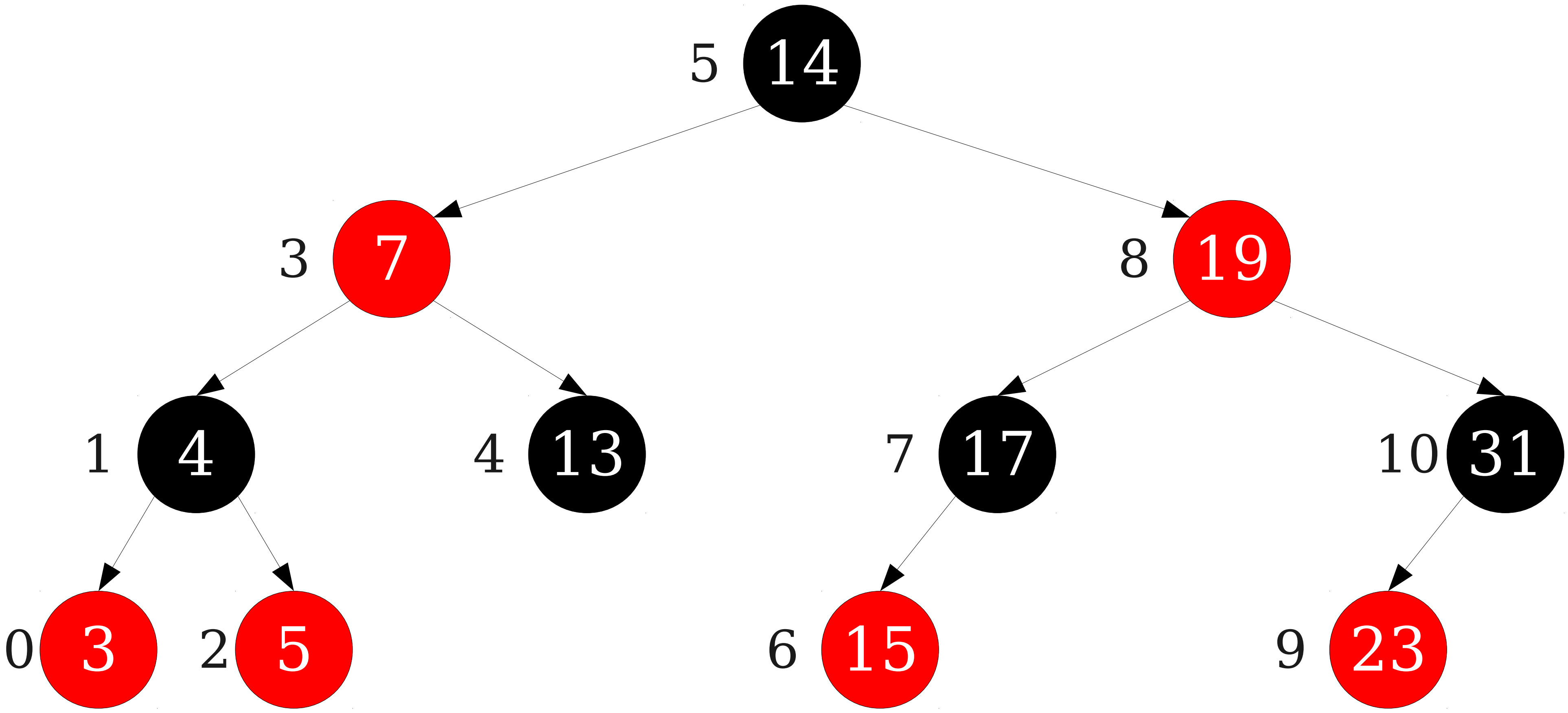


Dynamic Order Statistics

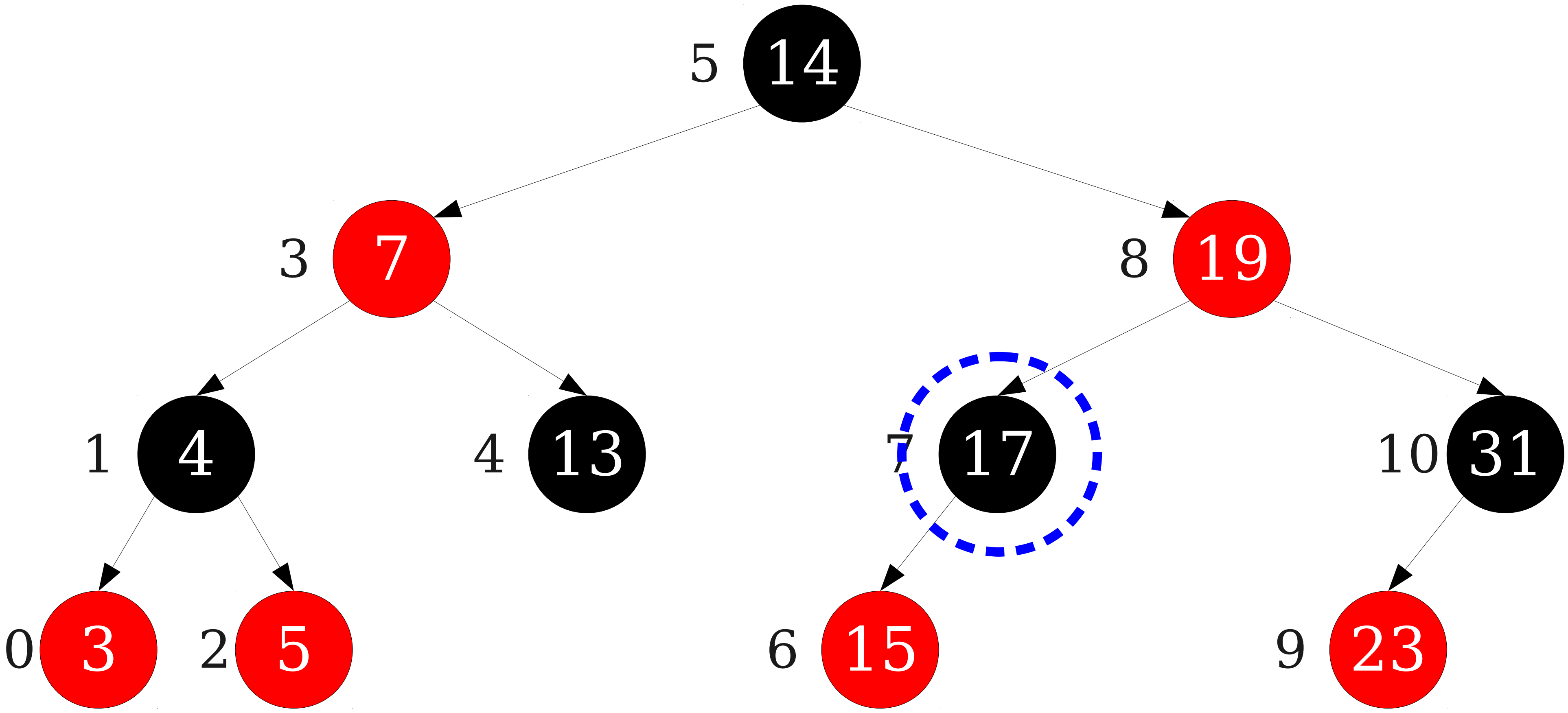
Order Statistics

- In a set S of totally ordered values, the ***k*th 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 (i.e. elements are added or removed).

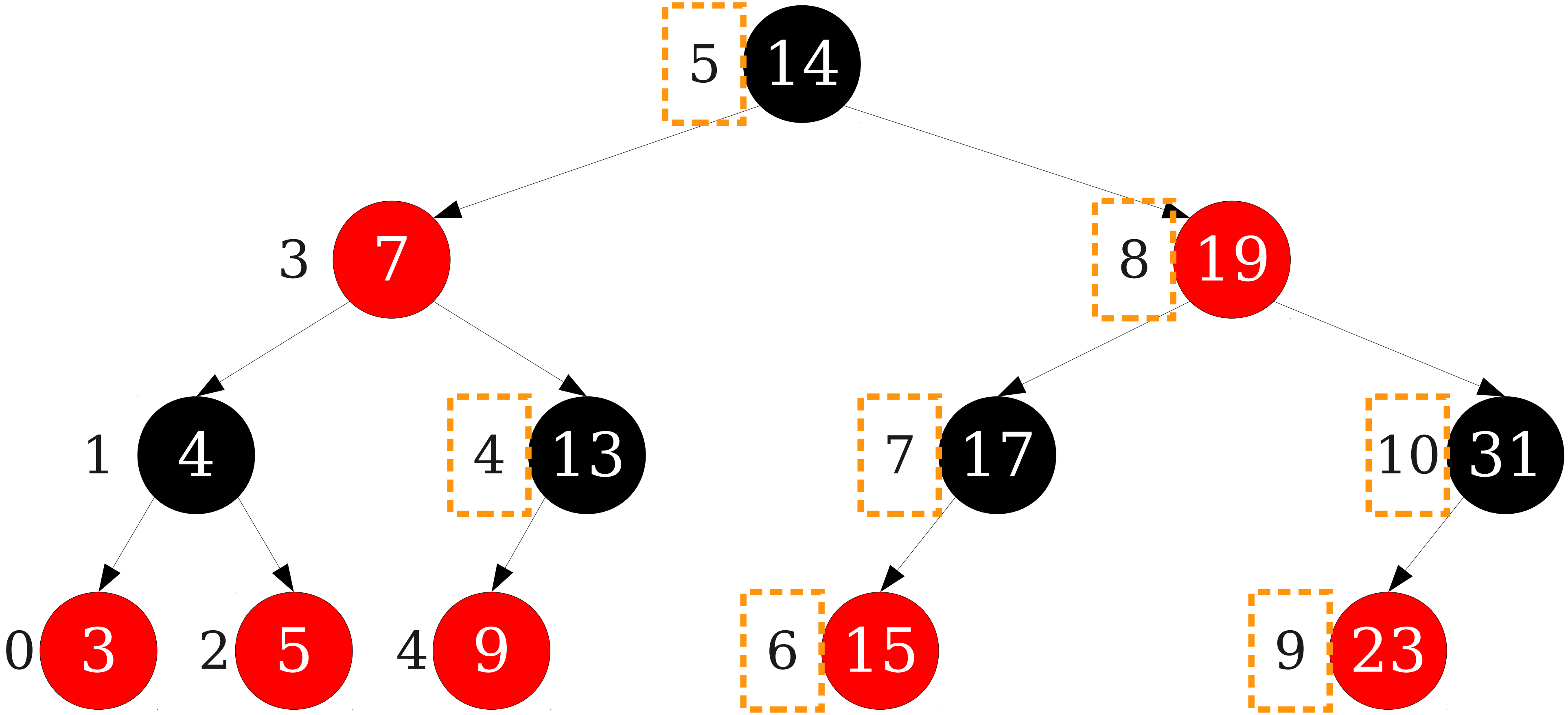
Finding Order Statistics



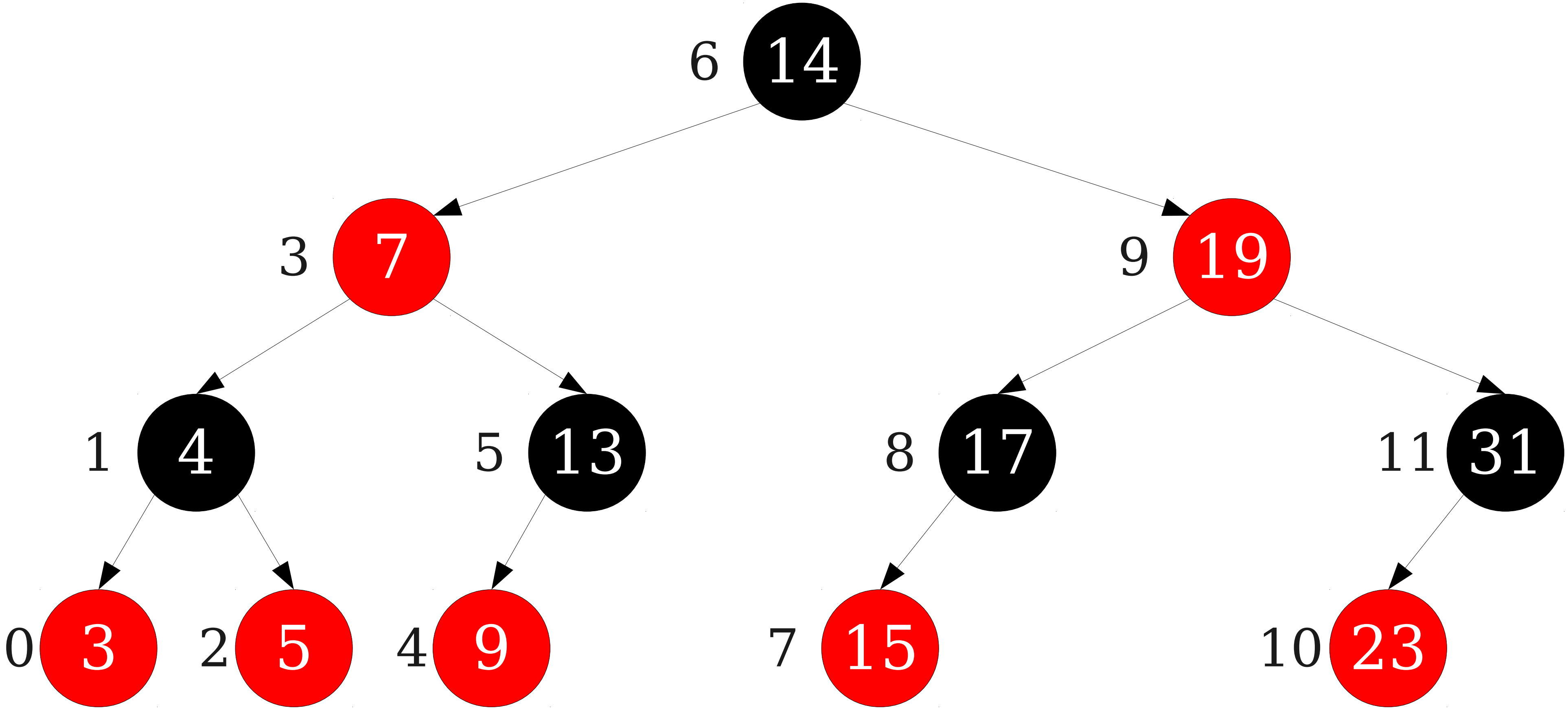
Finding Order Statistics



Finding Order Statistics



Finding Order Statistics

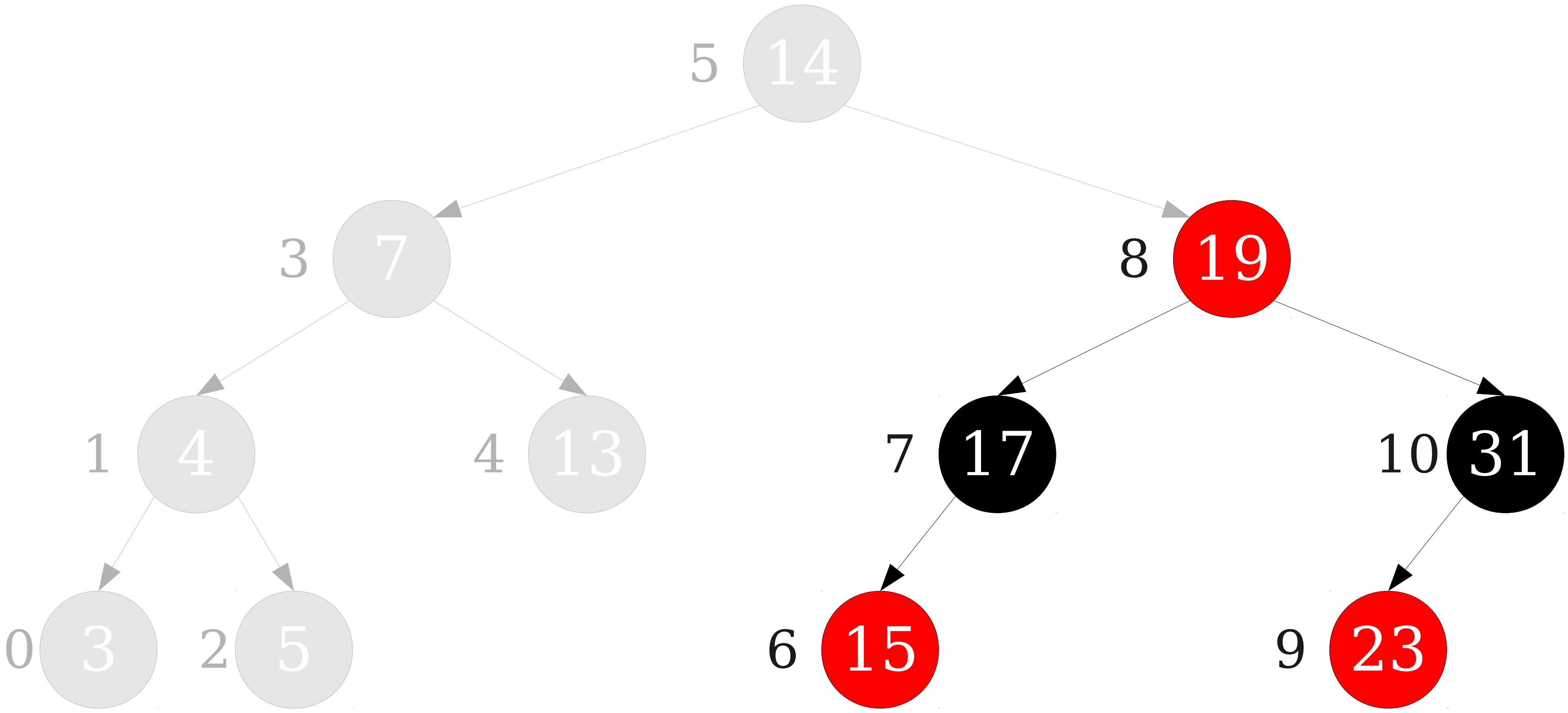


Problem: After inserting a new value, we may have to update $\Theta(n)$ values.

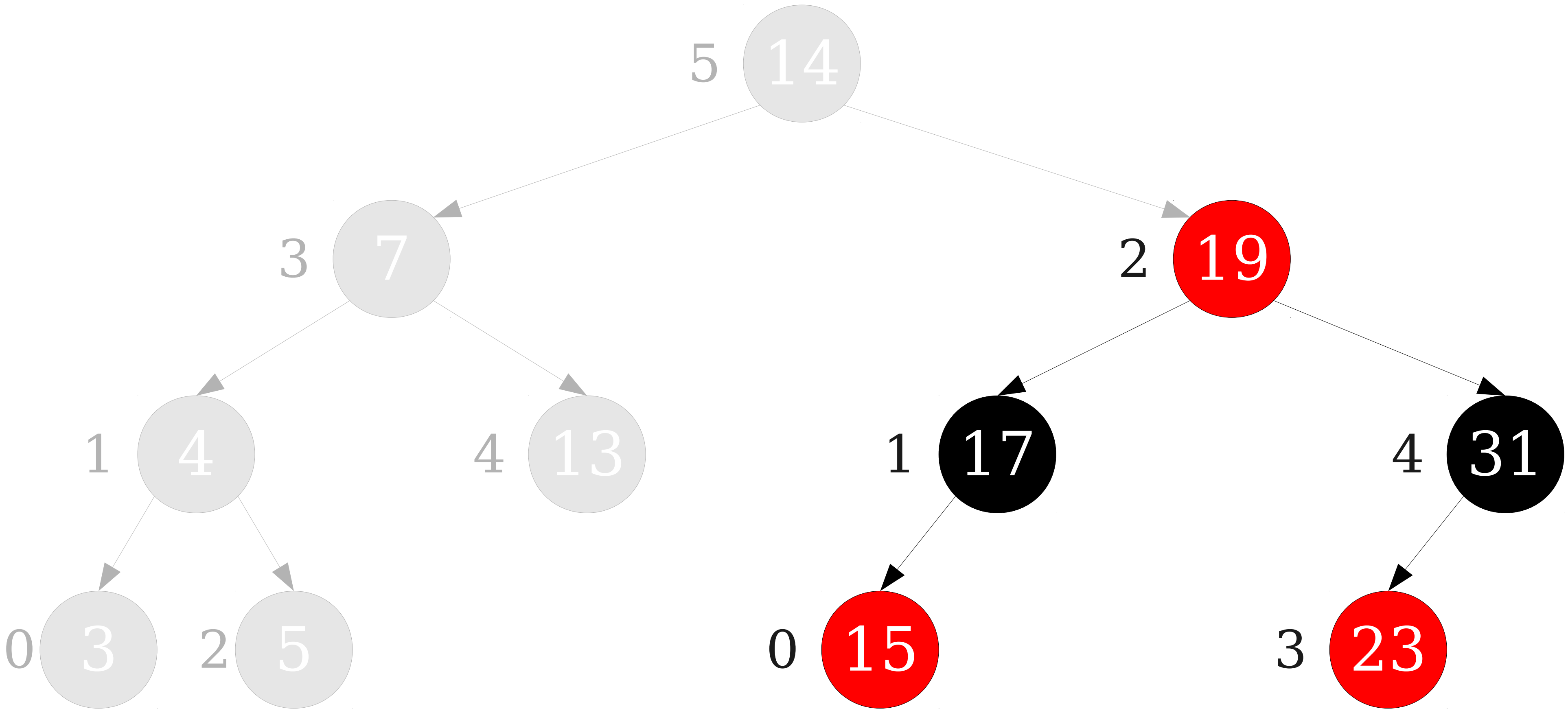
An Observation

- 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?
  - Depends purely on nearby nodes.

Finding Order Statistics

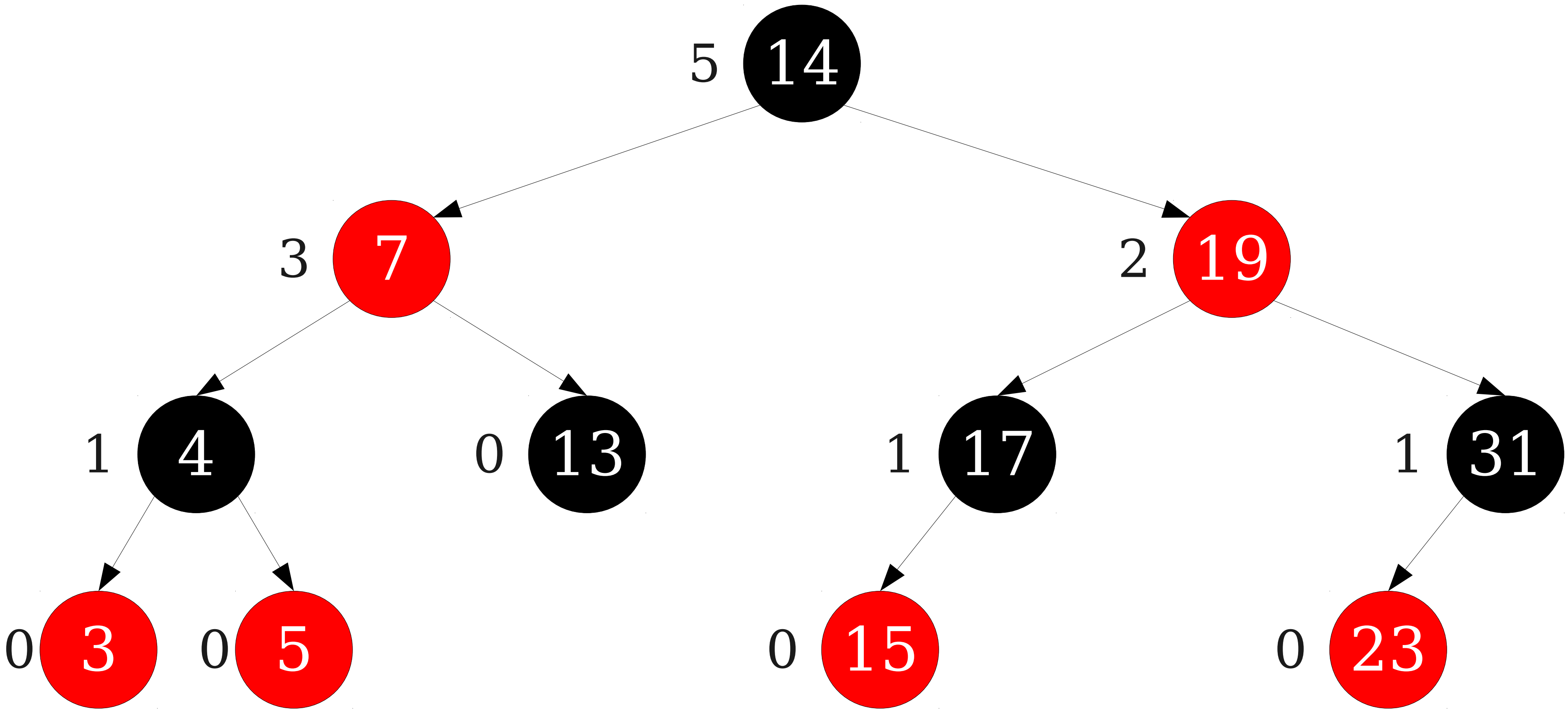


Finding Order Statistics



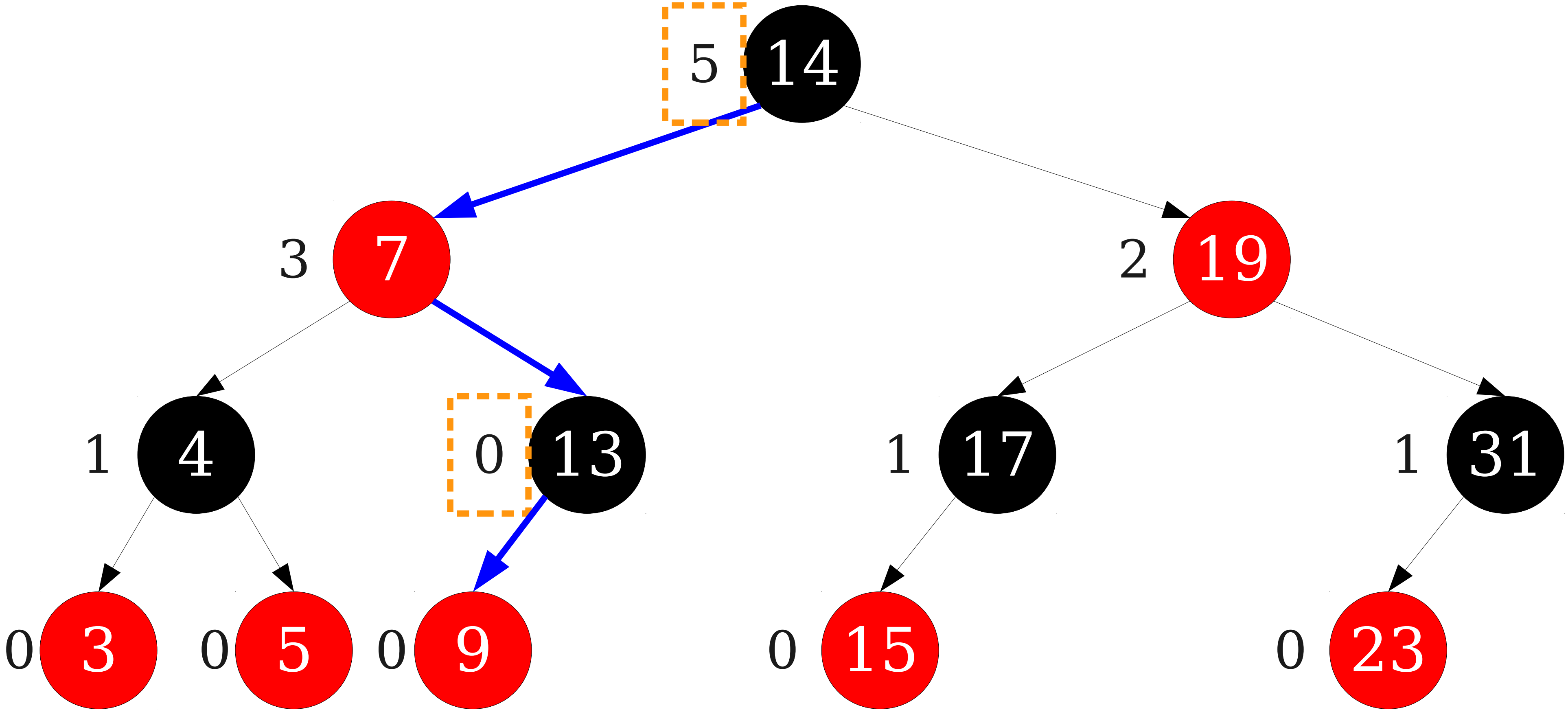
If new nodes are added to the left subtree, these numbers don't need to be updated.

Finding Order Statistics



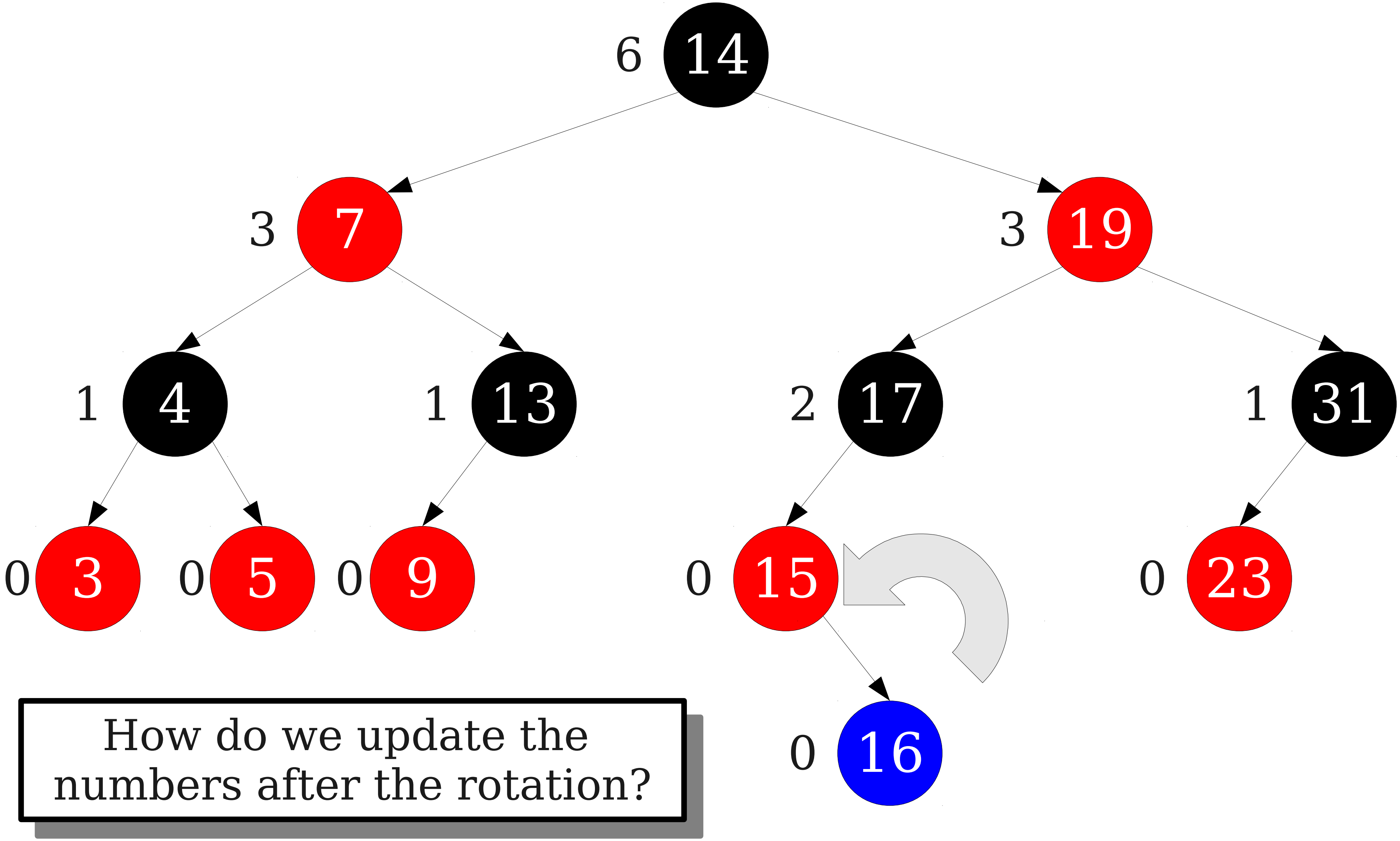
Each node is annotated with the number of children in its left subtree.

Finding Order Statistics



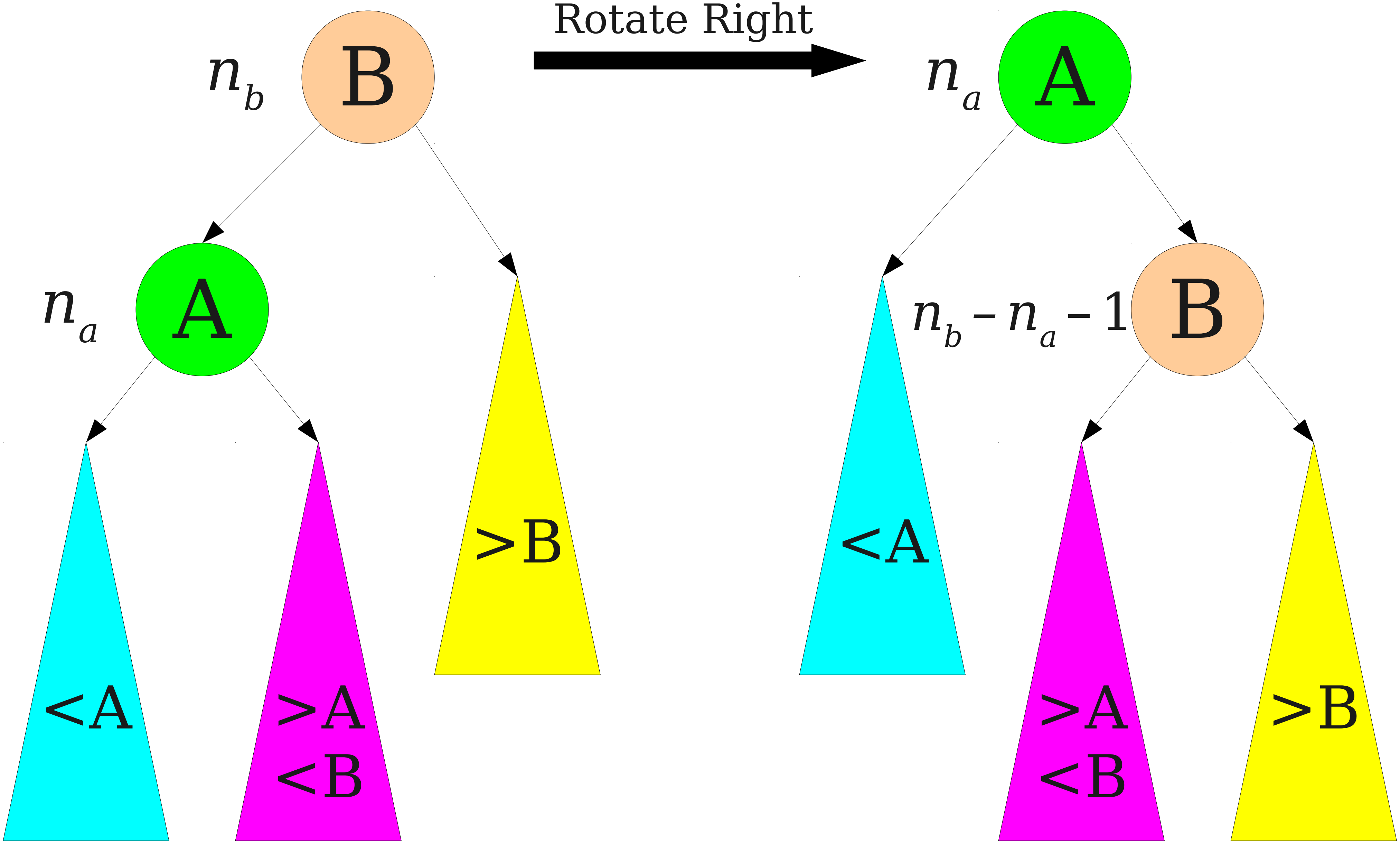
Since the number just holds the number of nodes in its left subtree, we only need to increment the value for nodes that have the new node in its left subtree.

Finding Order Statistics

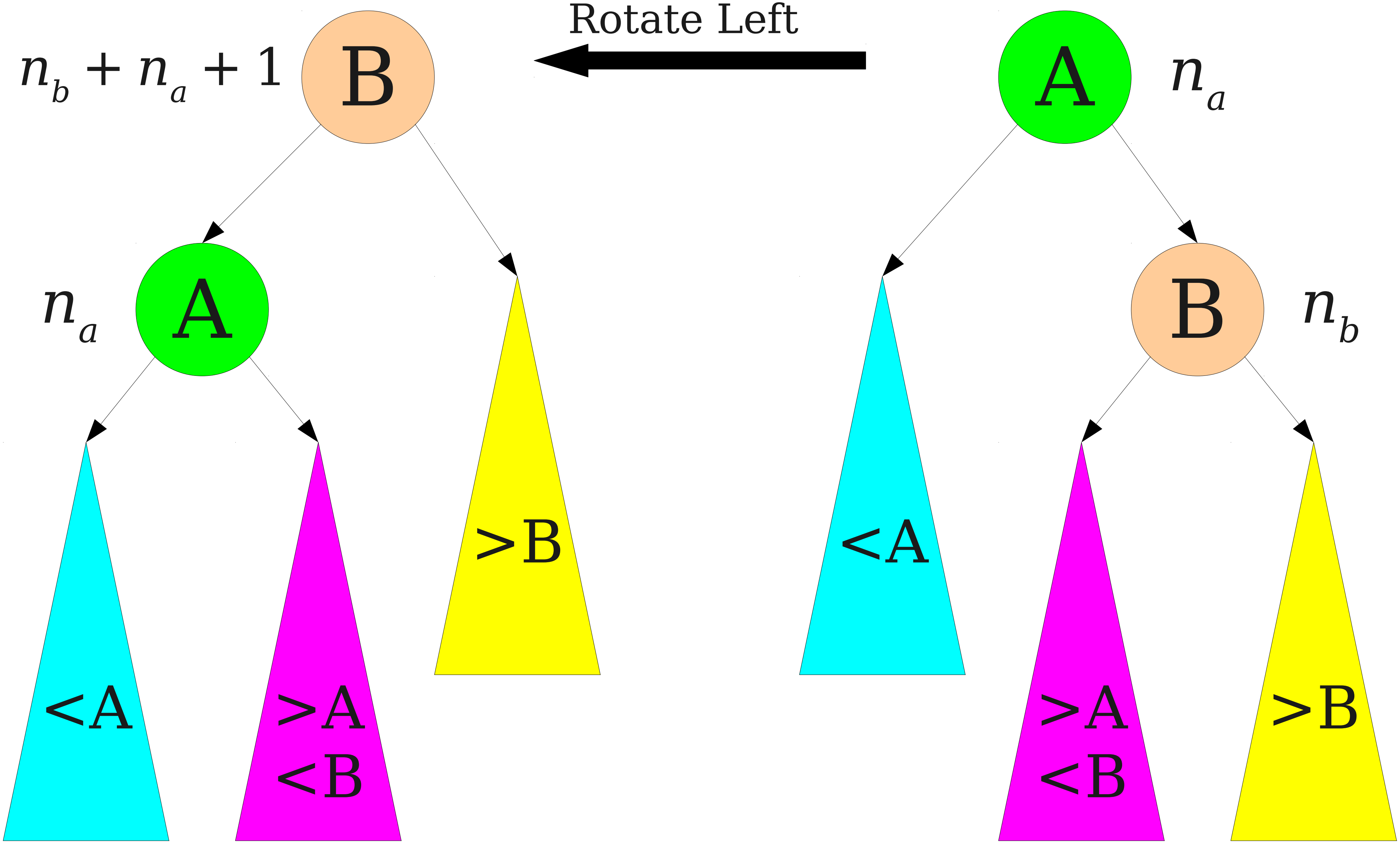


How do we update the numbers after the rotation?

Rotations and Order Statistics



Rotations and Order Statistics



Order Statistic Trees

- The tree we just saw is called an **order statistic tree**.
- Include in 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).

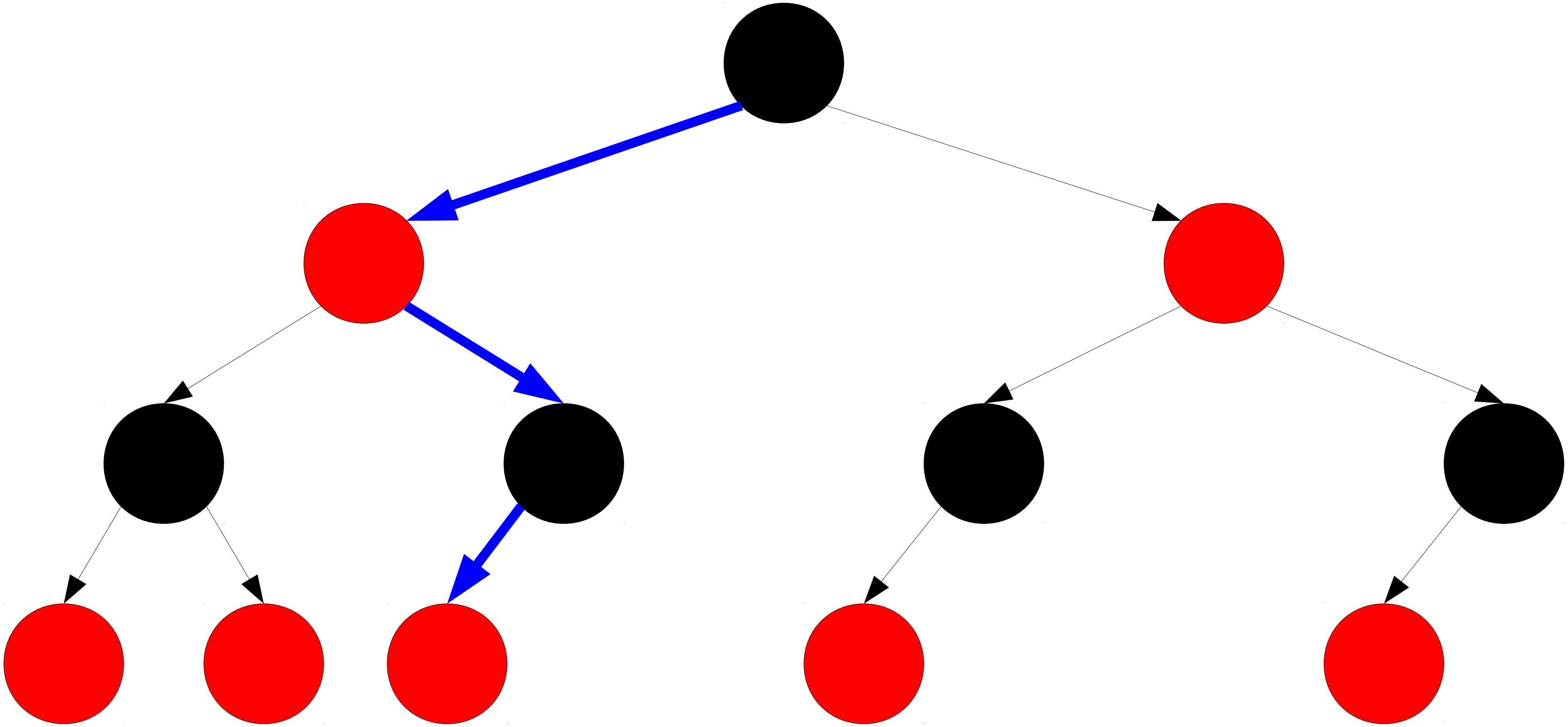
The General Pattern

- This data structure works in the appropriate time bounds because values only change in two cases:
  - Along the root-leaf access path.
  - 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.

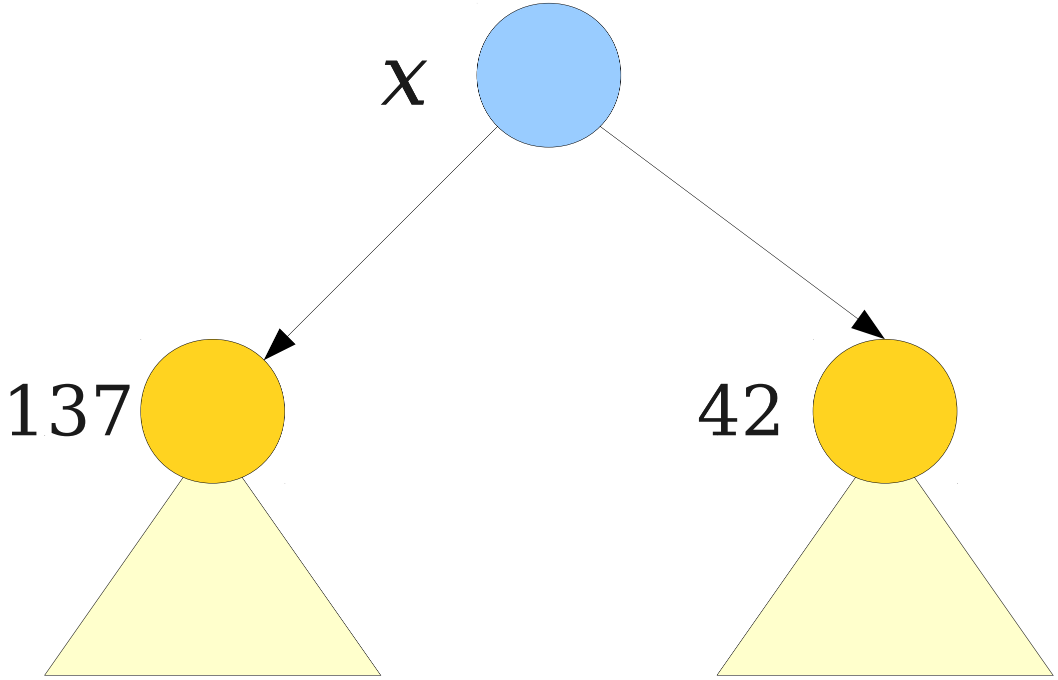
Augmented Red/Black Trees



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*?



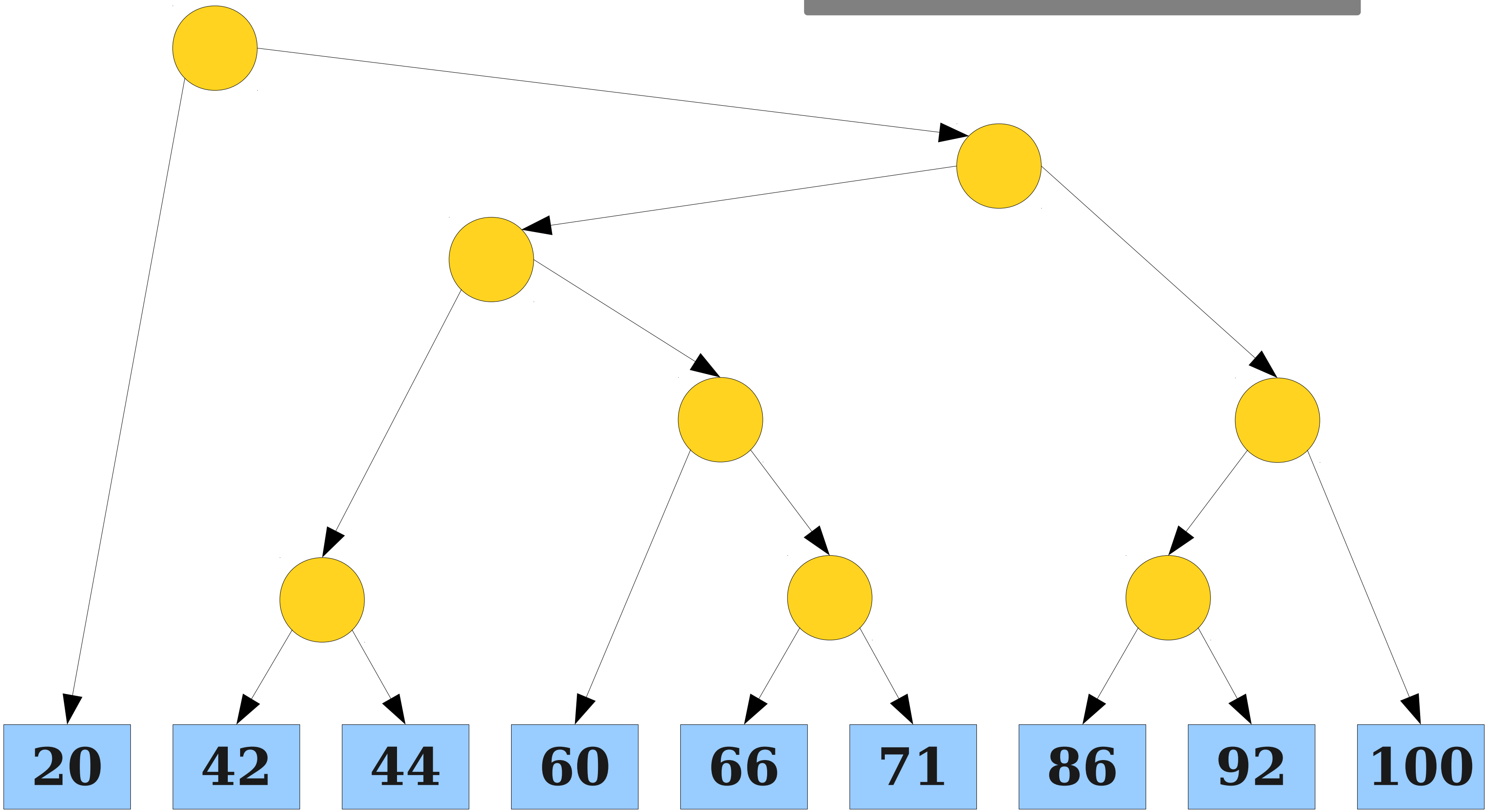
Order Statistics via Augmentation

- 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

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?

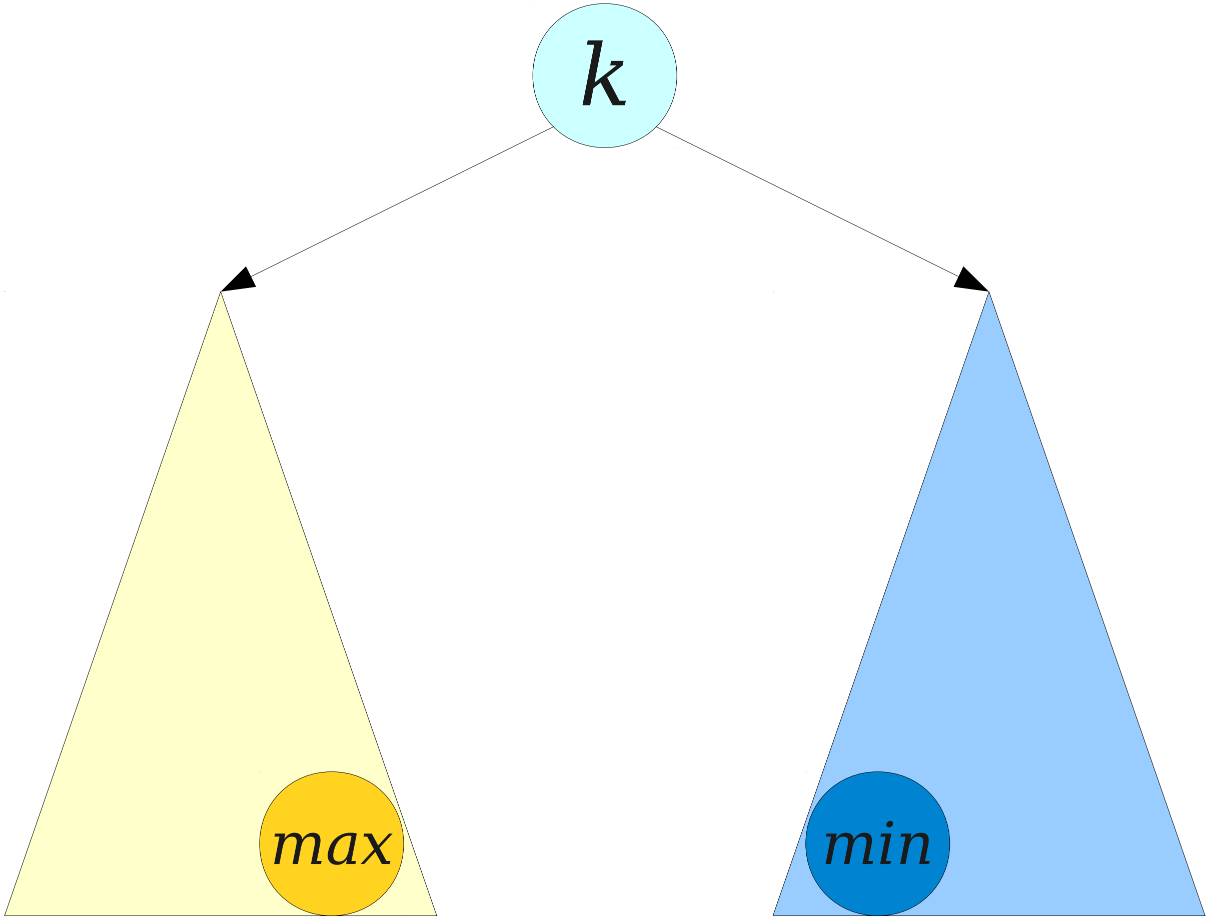
Dynamic 1D Closest Points

- 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?

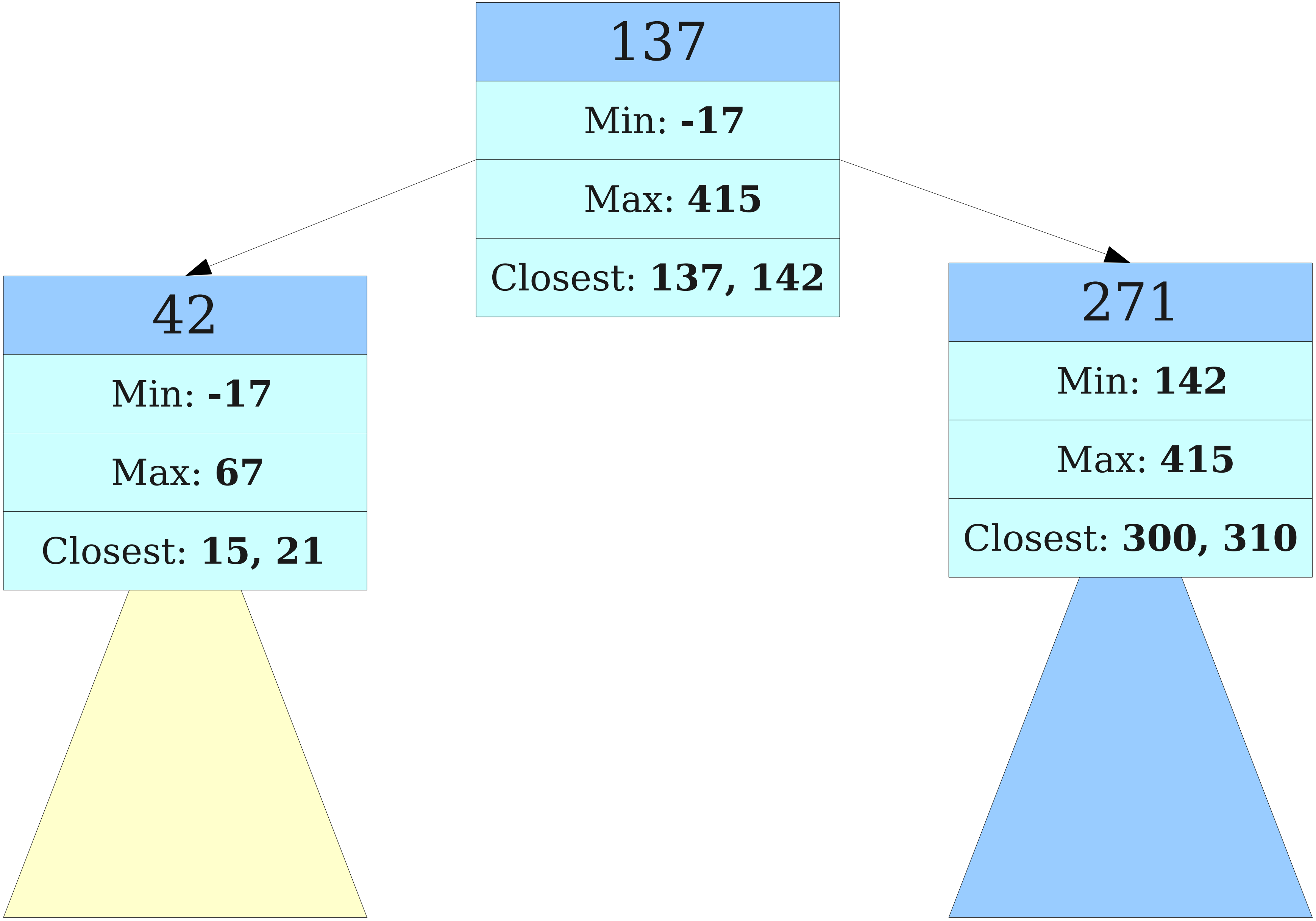
Dynamic 1D Closest Points



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).

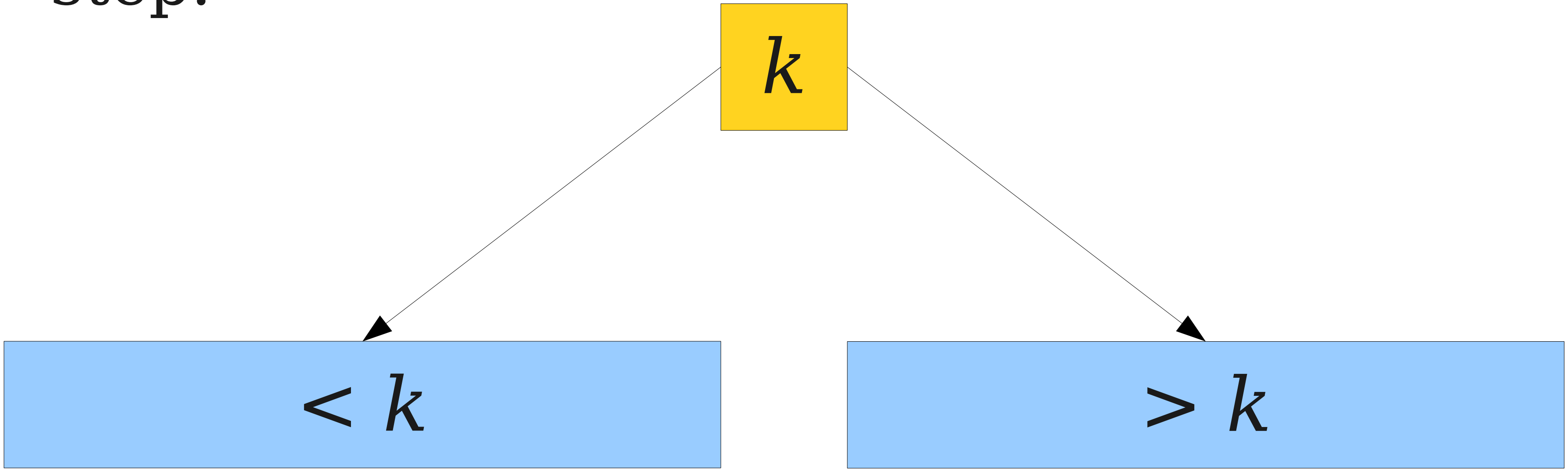
Dynamic 1D Closest Points



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 Two

- Problem Set Two goes out today. It's due next Wednesday, April 16, at 2:15PM.
- Play around with red/black trees, tree augmentations, splitting, and joining!

Your Questions

“Can you please post your extra lecture slides that you end up cutting (or, alternatively, make a course reader with the additional information that doesn't make it into the course for time's sake)?”

I'll try my best! Some of them need some polish, but I can release them “as-is.”

I don't think I have the time to put together a proper course reader this quarter (sorry!)

“Is there a particular reason that there are no late days for this class?”

A few reasons:

Logistics: Complex to make this fair with partner assignments, and difficult on our end to grade.

Pacing: Everything in this class builds off of itself and I want to ensure that everyone is caught up.

Scale: We can handle extensions for emergencies on a case-by-case basis.

“Is there a way to manage balanced BST so that they can handle interval queries, such as RMQ?”

It depends on the query, but usually **yes**! You can use the augmented BST framework for this.

“How long was the first homework supposed to take?”

I was aiming for 8 – 10 hours, assuming that you are working in pairs.

4 units × (3 hours / unit wk) = 12 hours / wk

12 hours / wk – 2.5 hours / wk = **9.5 hours / wk**

Let me know if this wasn't the case!

Join and Split

Joining and Splitting Trees

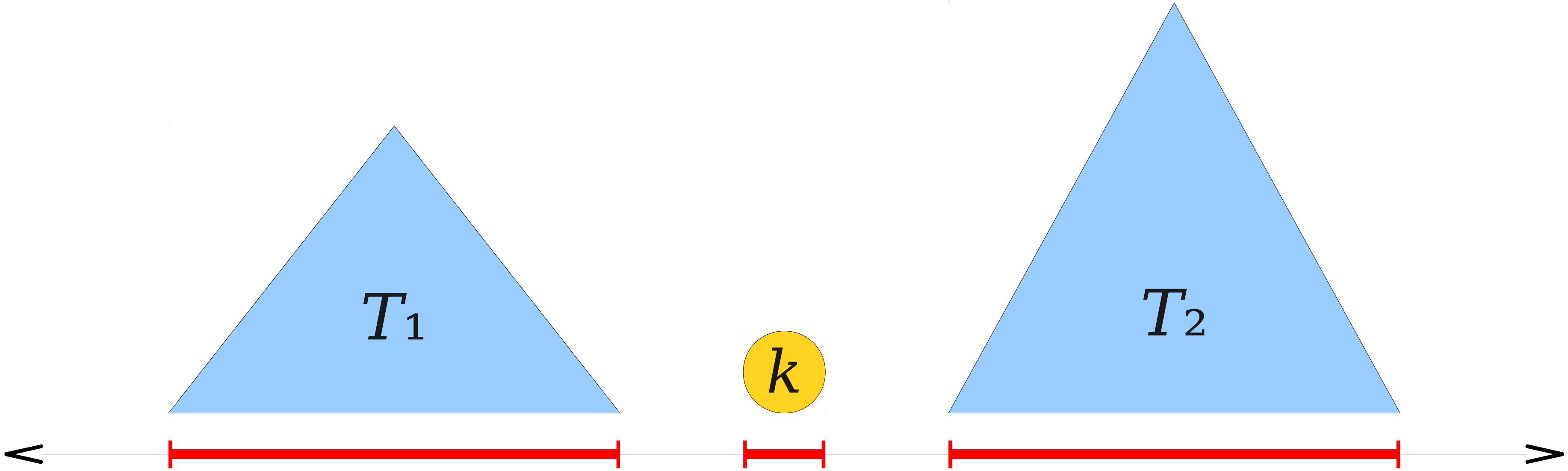
- The **join** and **split** operations are powerful primitives on balanced BSTs.
- You'll use them in the problem set and we'll see them on Monday.

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.

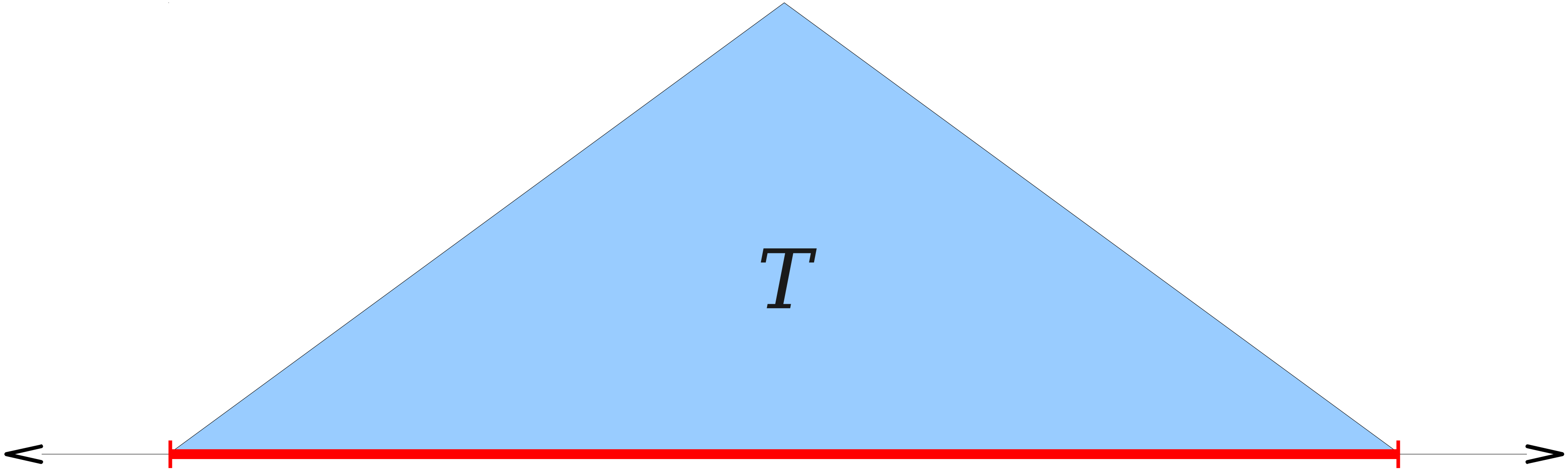
Joining Trees

- **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.



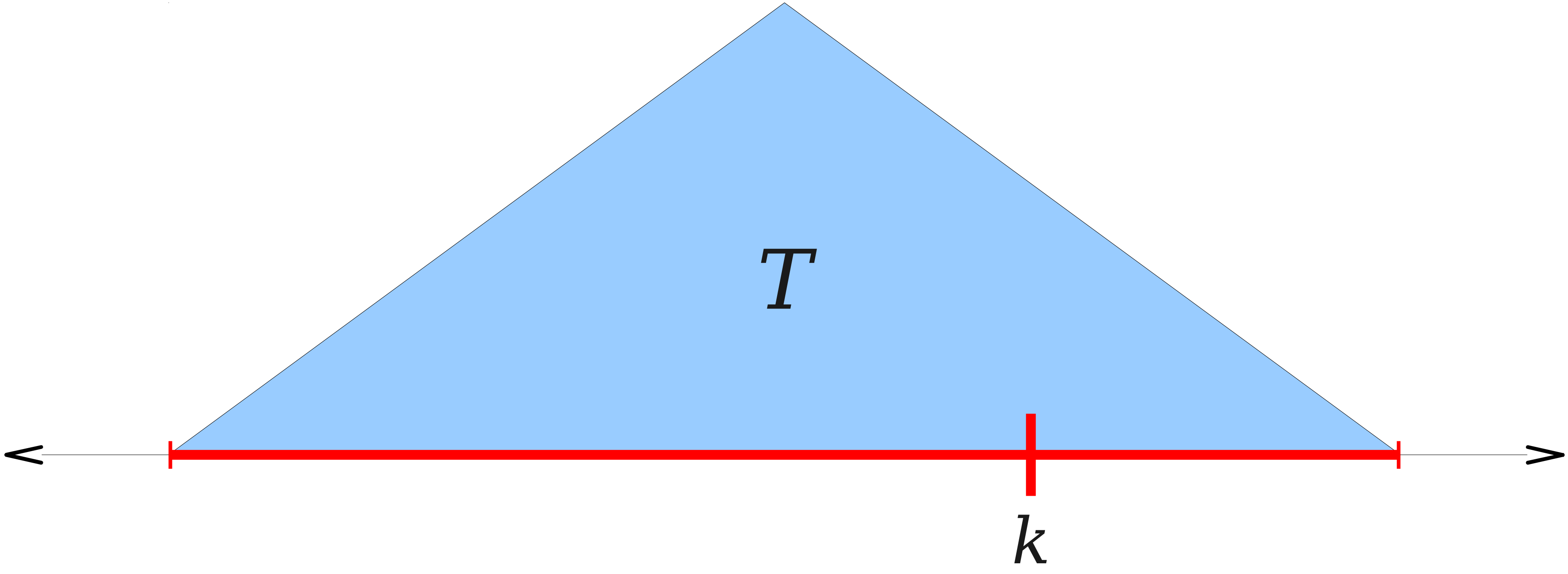
Joining Trees

- **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.



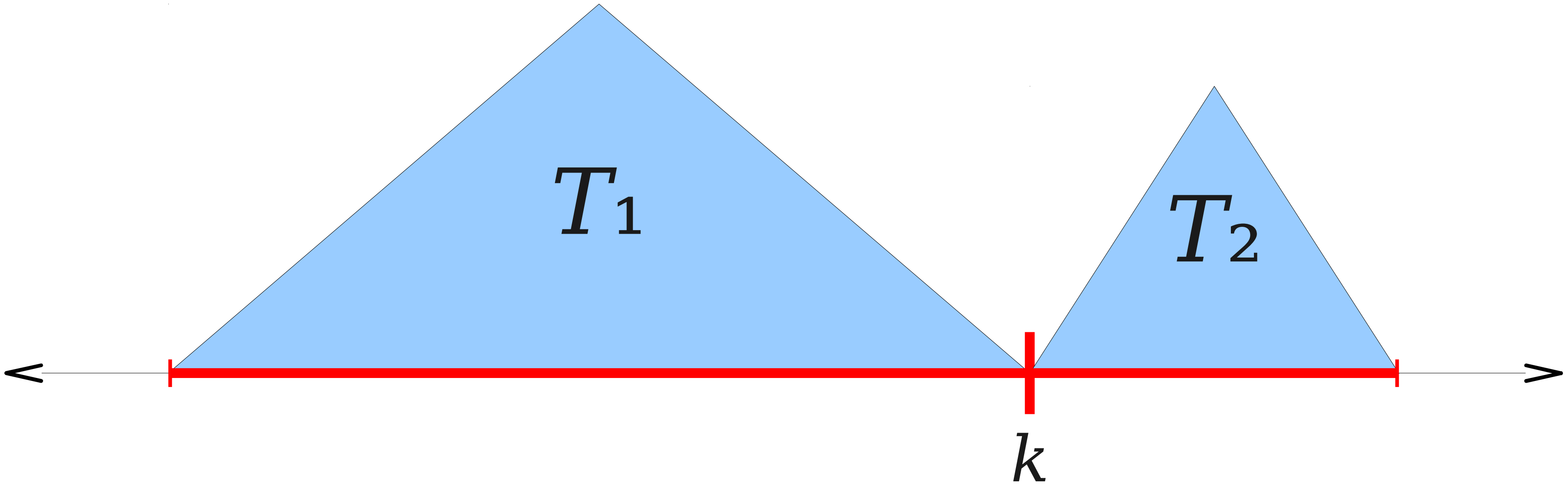
Splitting Trees

- **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.



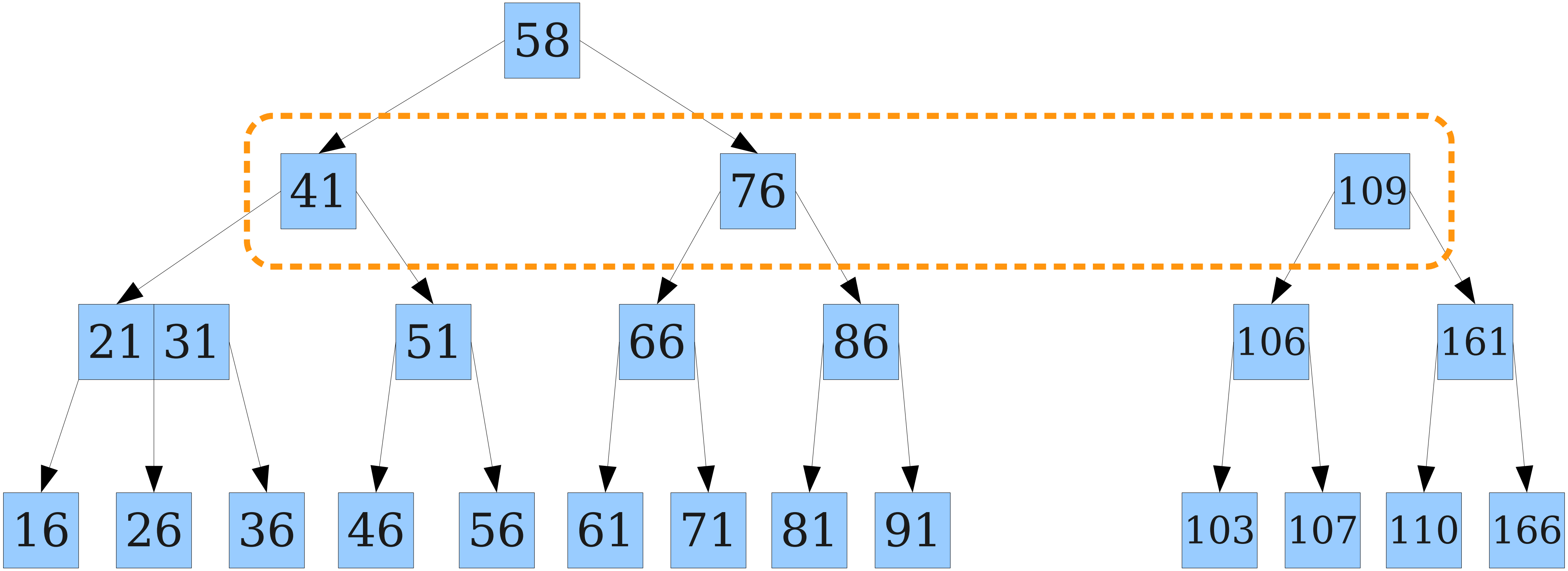
The Runtimes

- Both of these operations can be implemented in time O(n) by completely rebuilding the trees from scratch.
  - You'll design an algorithm for this in the problem set.
- Amazingly:
  - **join**(T_1, k, T_2) can be made to run in time $\Theta(1 + |h_1 - h_2|)$.
  - **split**(T, k) can be made to run in time O(log n).
- How is this possible?

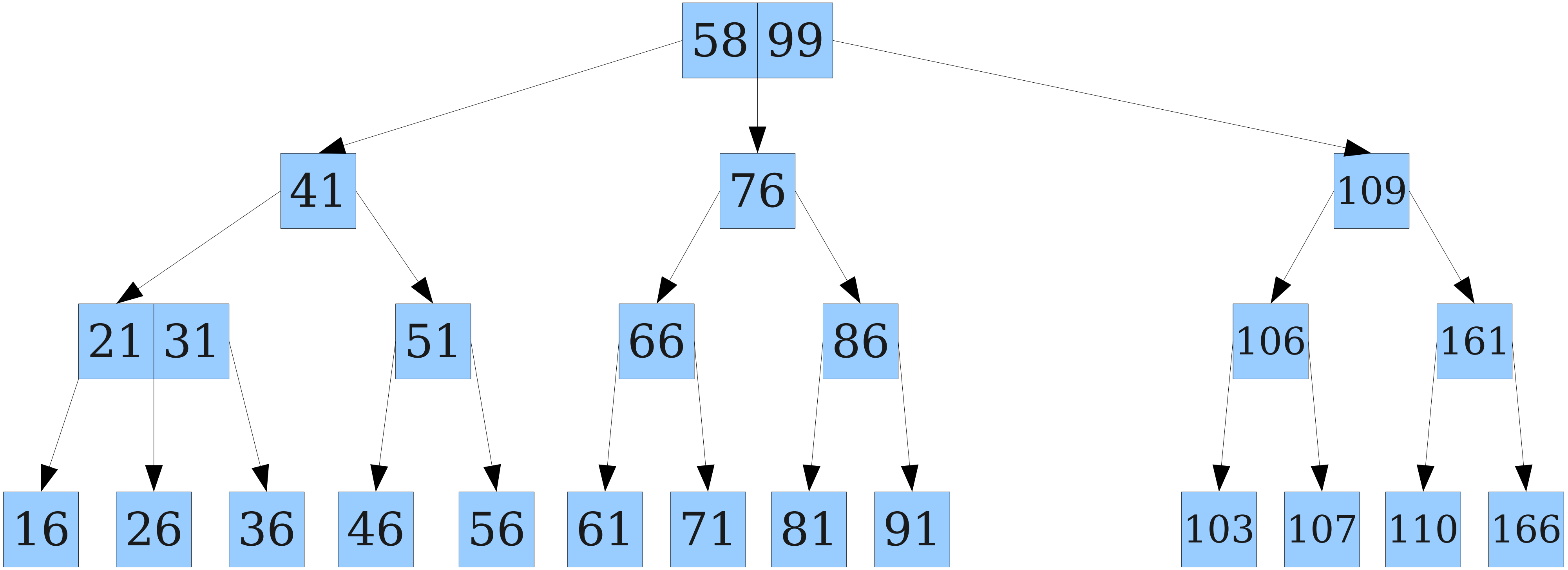
Joining 2-3-4 Trees

- 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.

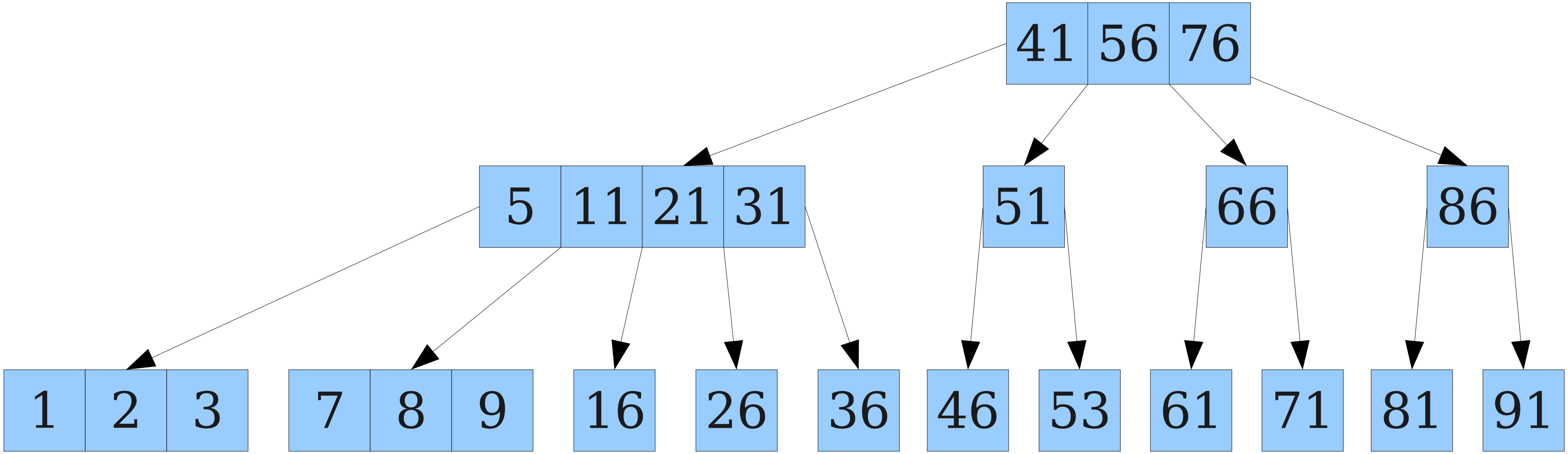
Joining 2-3-4 Trees



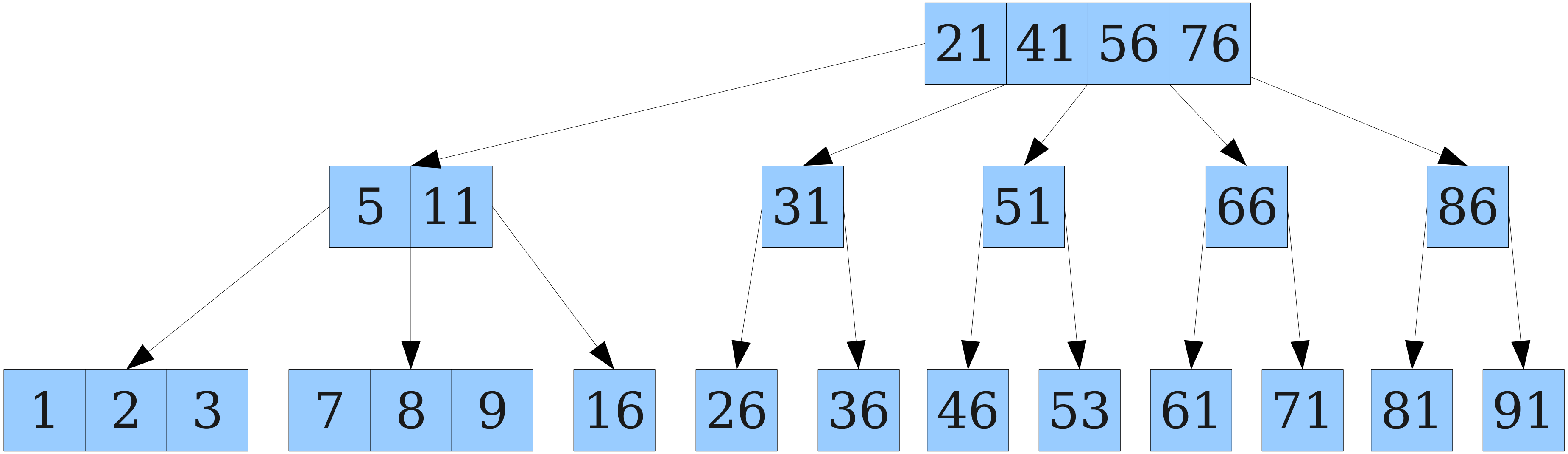
Joining 2-3-4 Trees



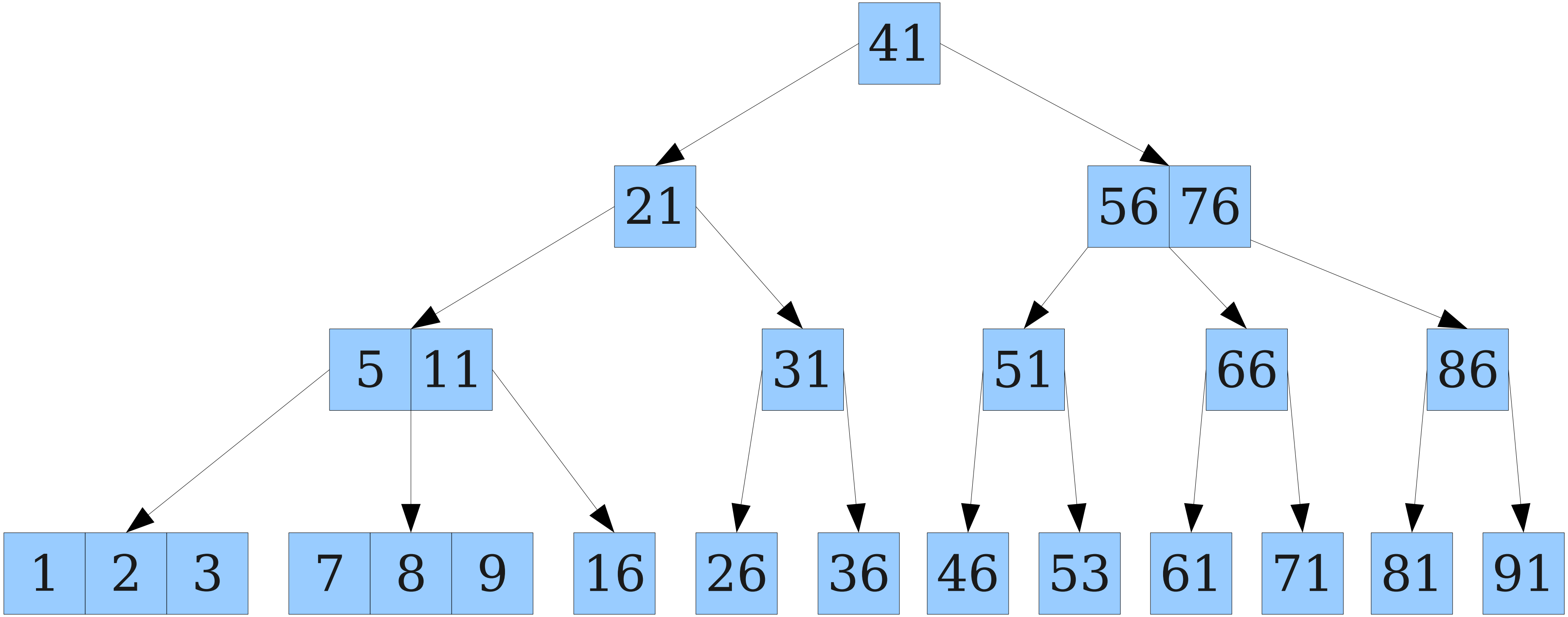
Joining 2-3-4 Trees



Joining 2-3-4 Trees



Joining 2-3-4 Trees

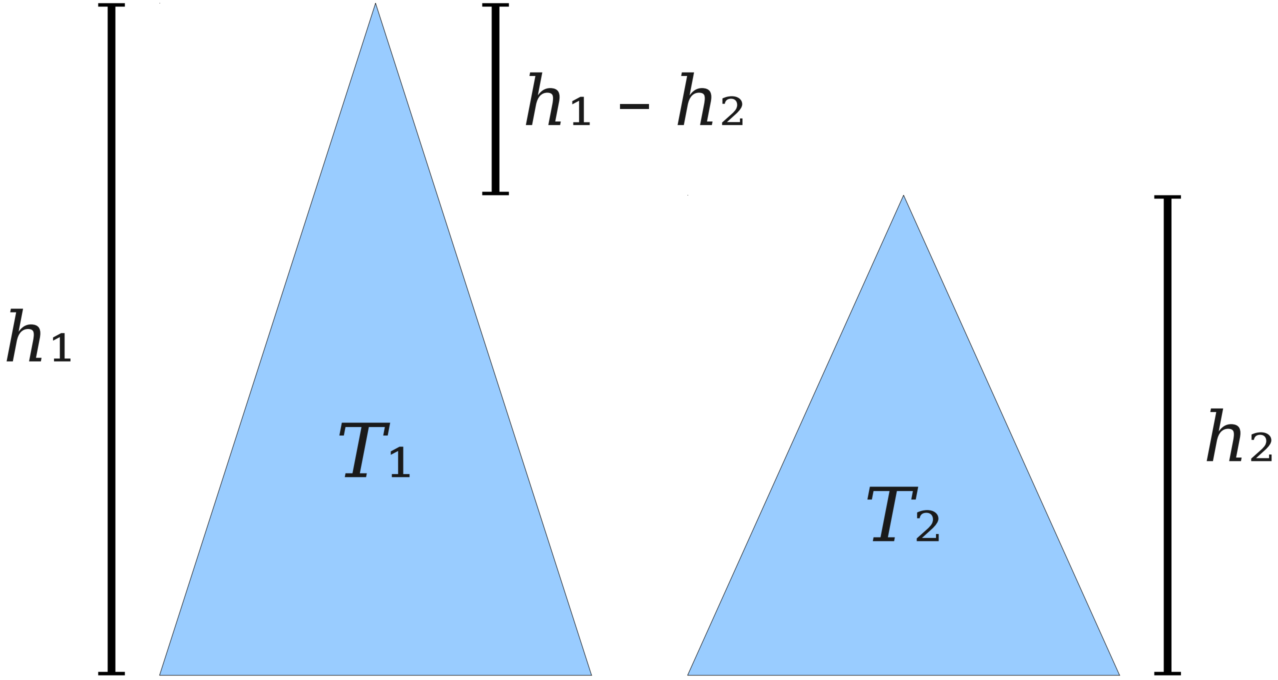


Joining 2-3-4 Trees

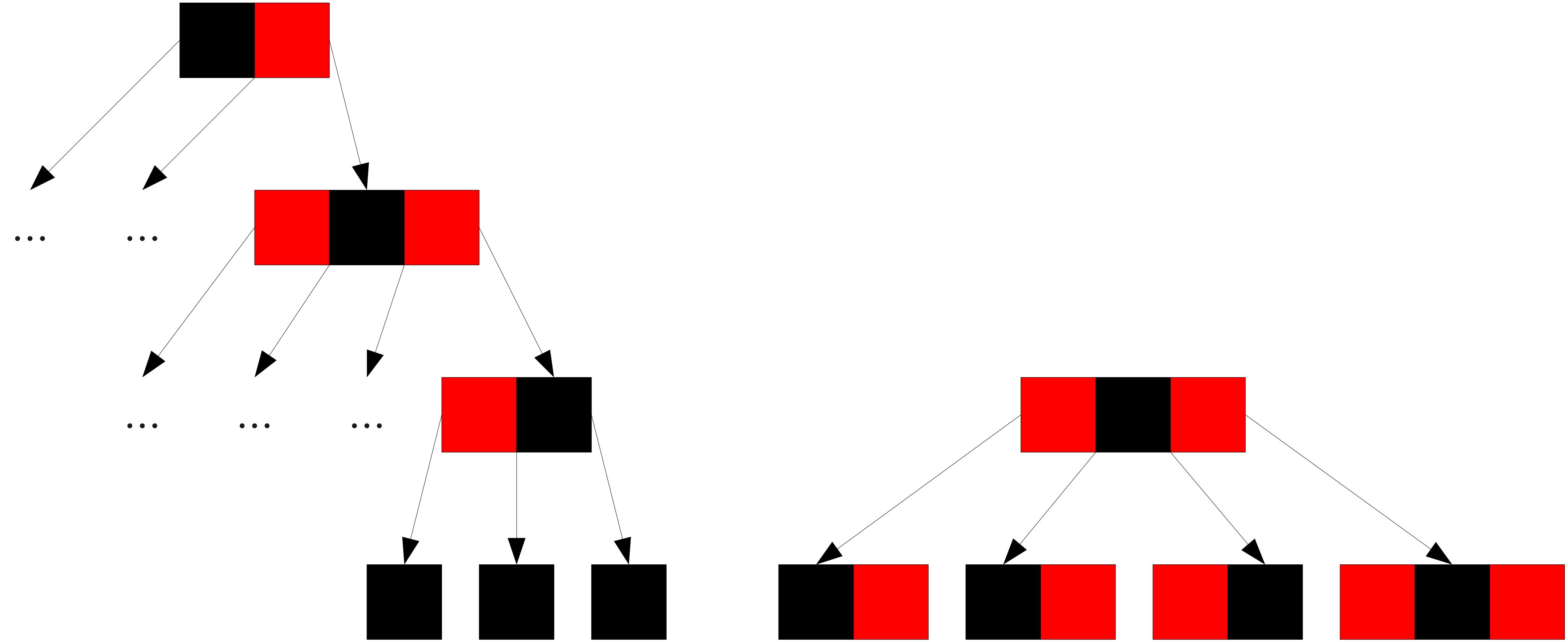
- To **join**(T_1, k, T_2):
  - Assume that T_1 is the larger of the two trees; if not, do the following, but mirrored.
  - Walk down the right spine of T_1 until a node v is found whose height is the height of T_2.
  - Add k as a final key of v's parent with T_2 as a right child.
  - Continue as if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

Analyzing the Runtime

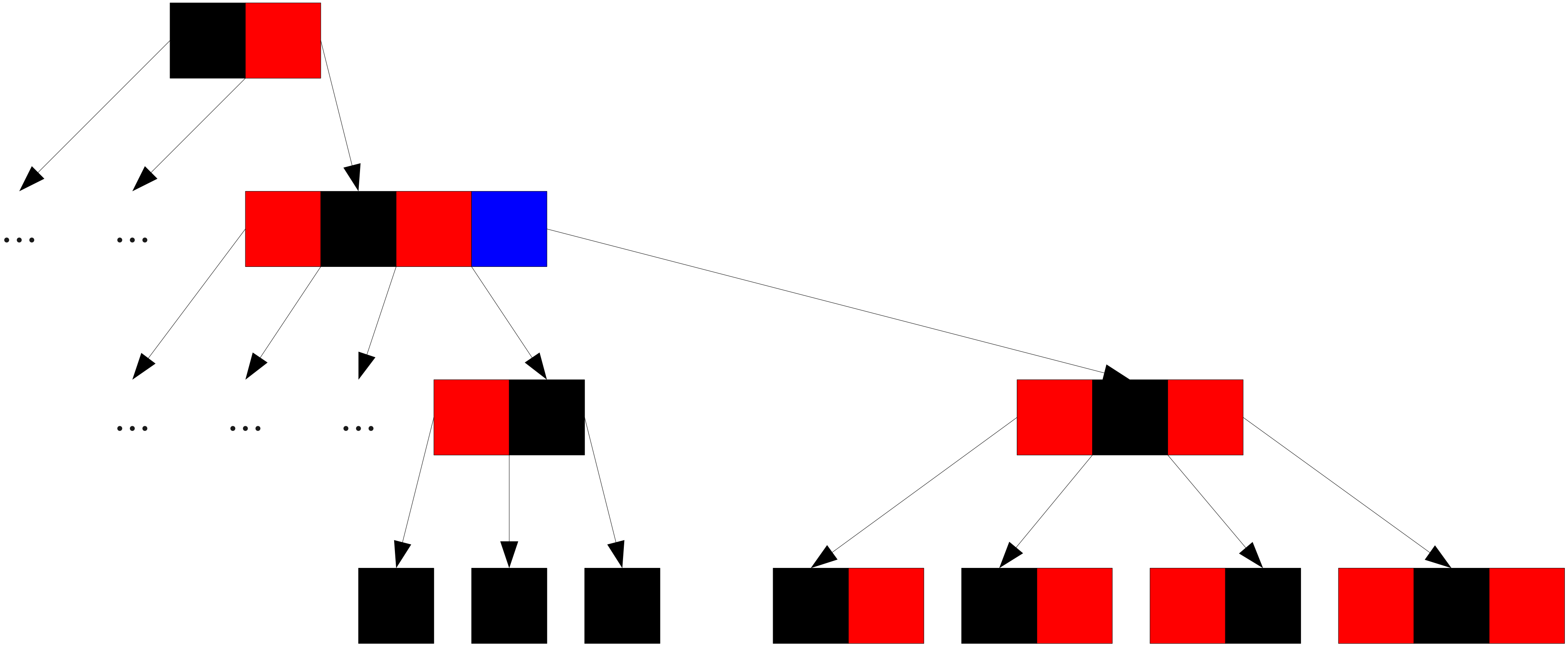
- Assume all 2-3-4 tree nodes are annotated with their heights.
- What is the runtime of **join**(T_1, k, T_2)?
- Runtime is $\boldsymbol{\Theta(1 + |h_1 - h_2|)}$.



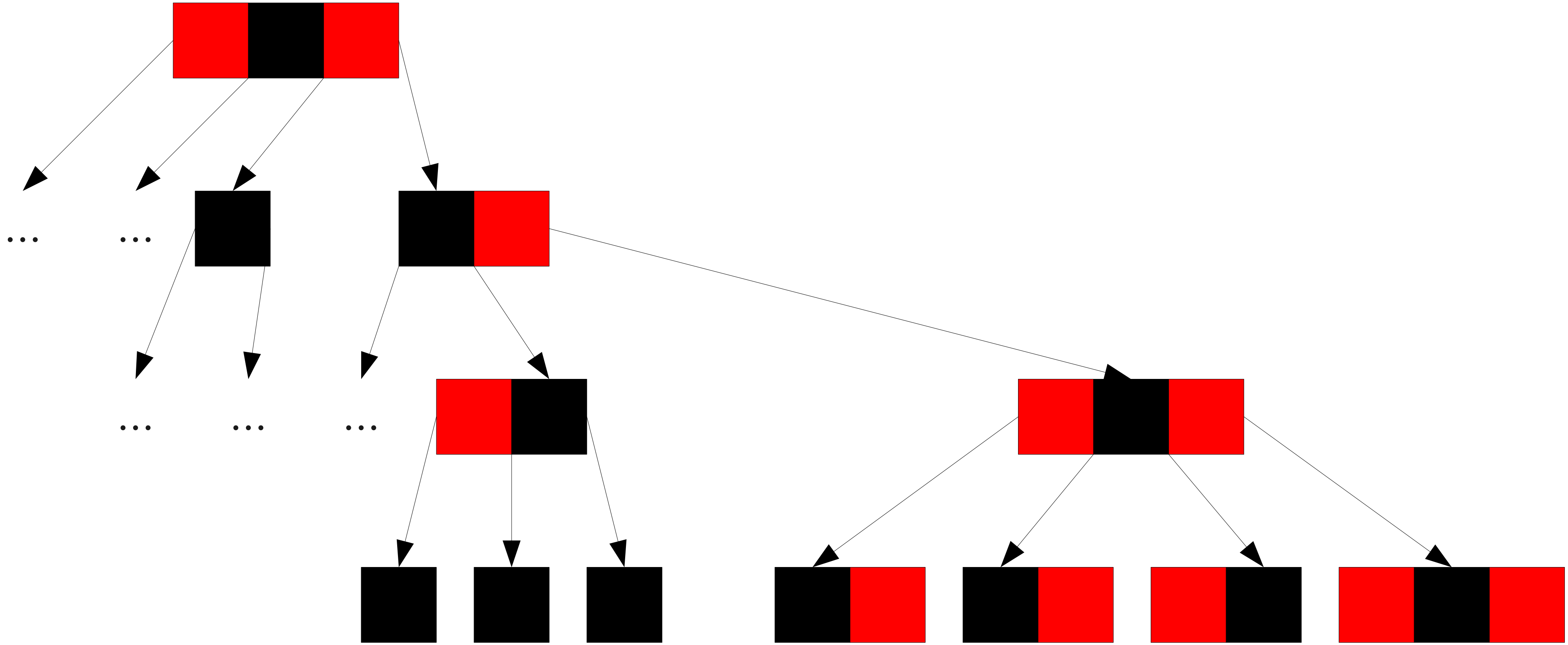
Joining Red/Black Trees



Joining Red/Black Trees

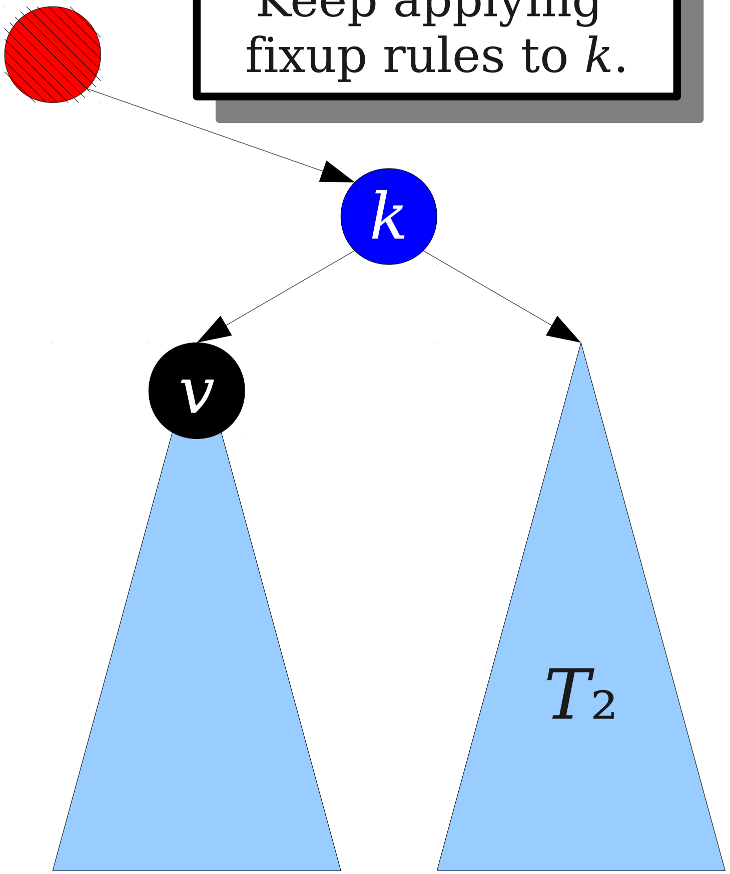


Joining Red/Black Trees



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.



Keep applying fixup rules to 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 $\boldsymbol{O(1 + |h_1 - h_2|)}$.
- This is $O(\log n_1 + \log n_2)$ in the worst-case.

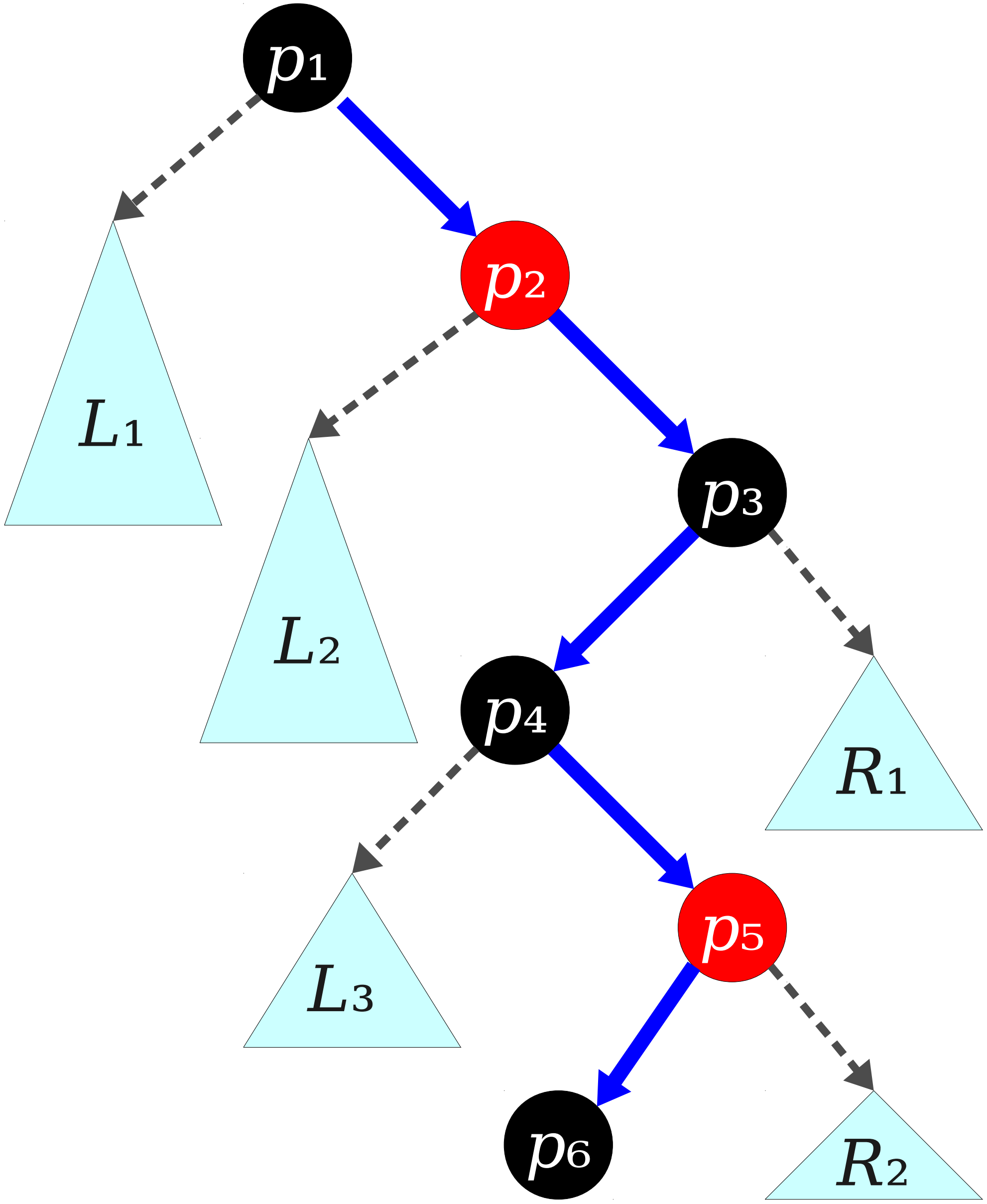
Joining Two Trees

- 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.

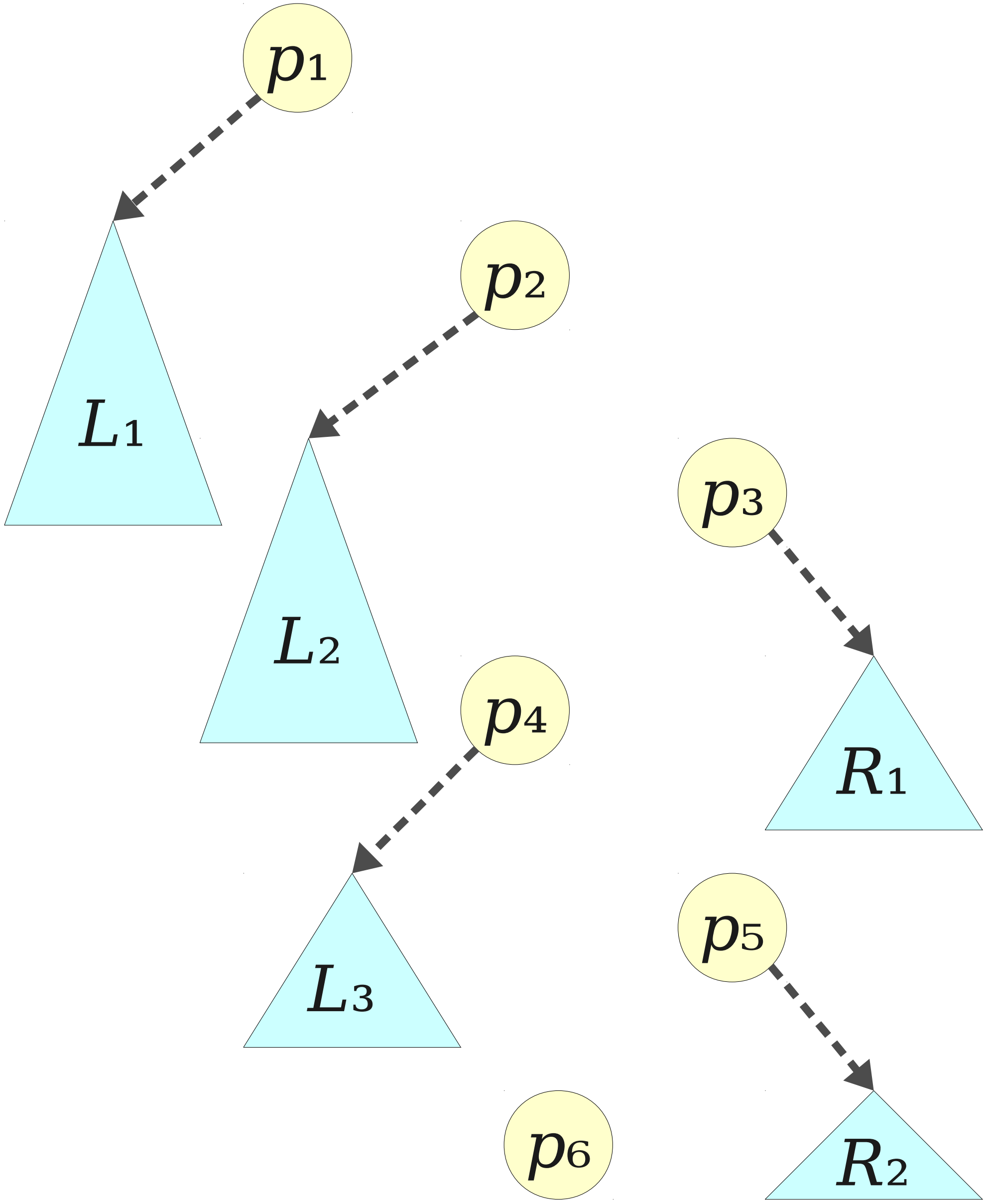
Implementing **split** Efficiently

The Intuition

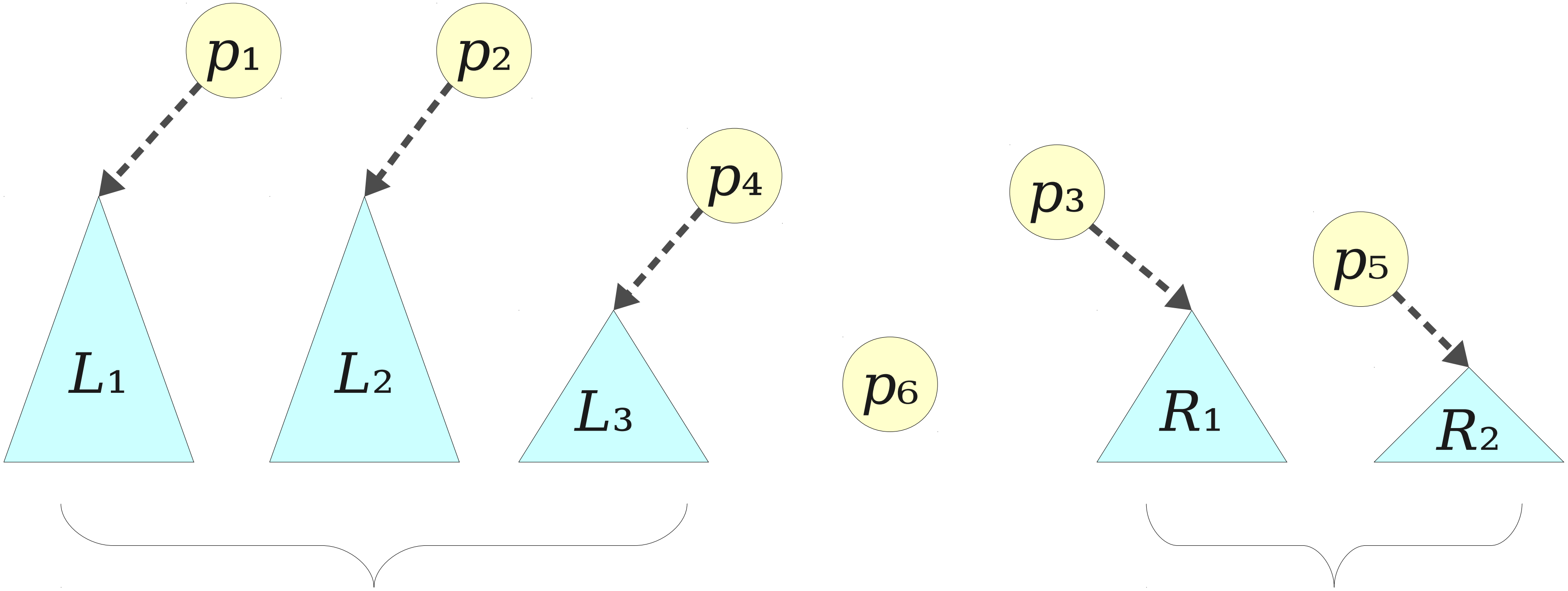
- Do a search for the inorder successor of k.
  - (search for k; if found, move right and search for k.)
- Splits the tree into the access path and a set of trees L and R hanging off of the access path.
- **Claim 1:** The keys in p_1, p_2, p_3, … are sandwiched between the keys in the trees in L and R.
- **Claim 2:** There are at most two trees of any given black height.



The Intuition



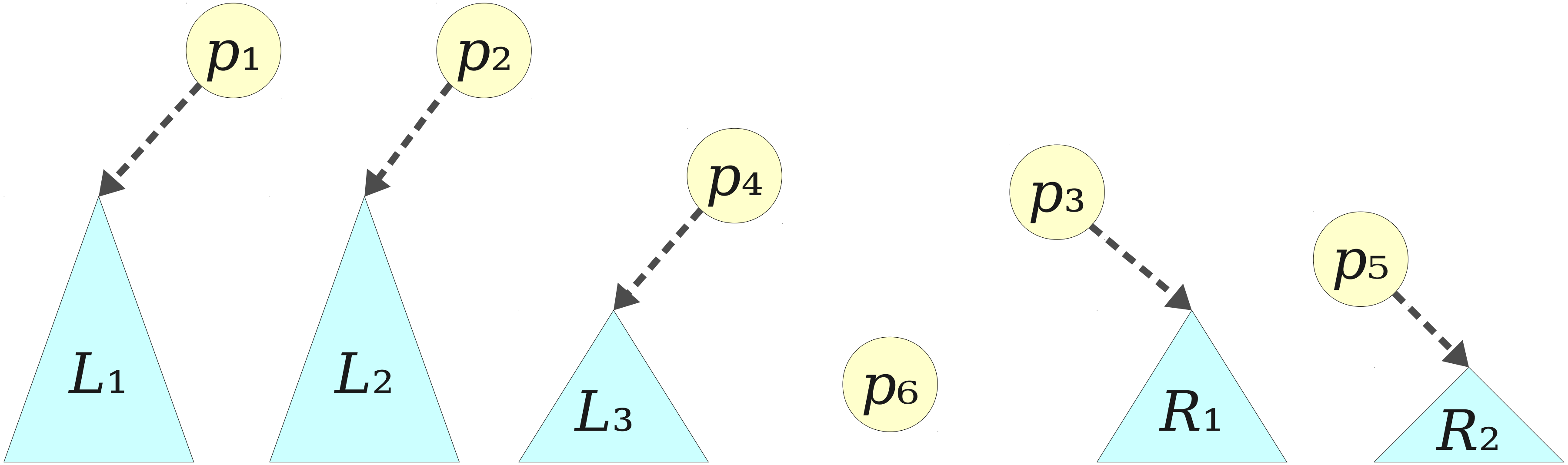
The Intuition



All keys here are less than or equal to k.

All keys here are greater than k.

The Intuition



Key idea: Join all the *L* trees back together and all the *R* trees back together. Because the height differences are low, the runtime works out to O(log *n*).

A Simplified Argument

- 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*)**.

A Simplified Argument

- 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 the L trees, the R trees, and the access path.
- In time O(log n), process the trees in L and R to ensure there's at most one tree of height h for each possible height h.
    - Details left as an exercise.
- In time O(log n), concatenate all trees in L and all trees in R using the previous approach.
- There will be O(1) leftover nodes from the access path. Insert them in time O(log n) into the proper trees.
- Net runtime: **O(log *n*)**.

Next Time

- **Dynamic Connectivity in Trees**
  - Maintaining connectivity under changing conditions.
- **Euler Tour Trees**
  - An elegant data structure for tree connectivity.
- **Bottleneck Edge Queries**
  - Putting everything together!