Balanced Trees

Part One

Balanced Trees

- Balanced search trees are among the most useful and versatile data structures.
- Many programming languages ship with a balanced tree library.
  - C++: `std::map` / `std::set`
  - Java: `TreeMap` / `TreeSet`
  - Python: `OrderedDict`
- Many advanced data structures are layered on top of balanced trees.
  - We'll see them used to build *y*-Fast Tries later in the quarter. (They’re really cool, trust me!)

Where We're Going

- ***B-Trees (Today)***
  - A simple type of balanced tree developed for block storage.
- ***Red/Black Trees (Today)***
  - The canonical balanced binary search tree.
- ***Augmented Search Trees (Tuesday)***
  - Adding extra information to balanced trees to supercharge the data structure.
- ***Two Advanced Operations (Tuesday)***
  - Splitting and joining BSTs.

Outline for Today

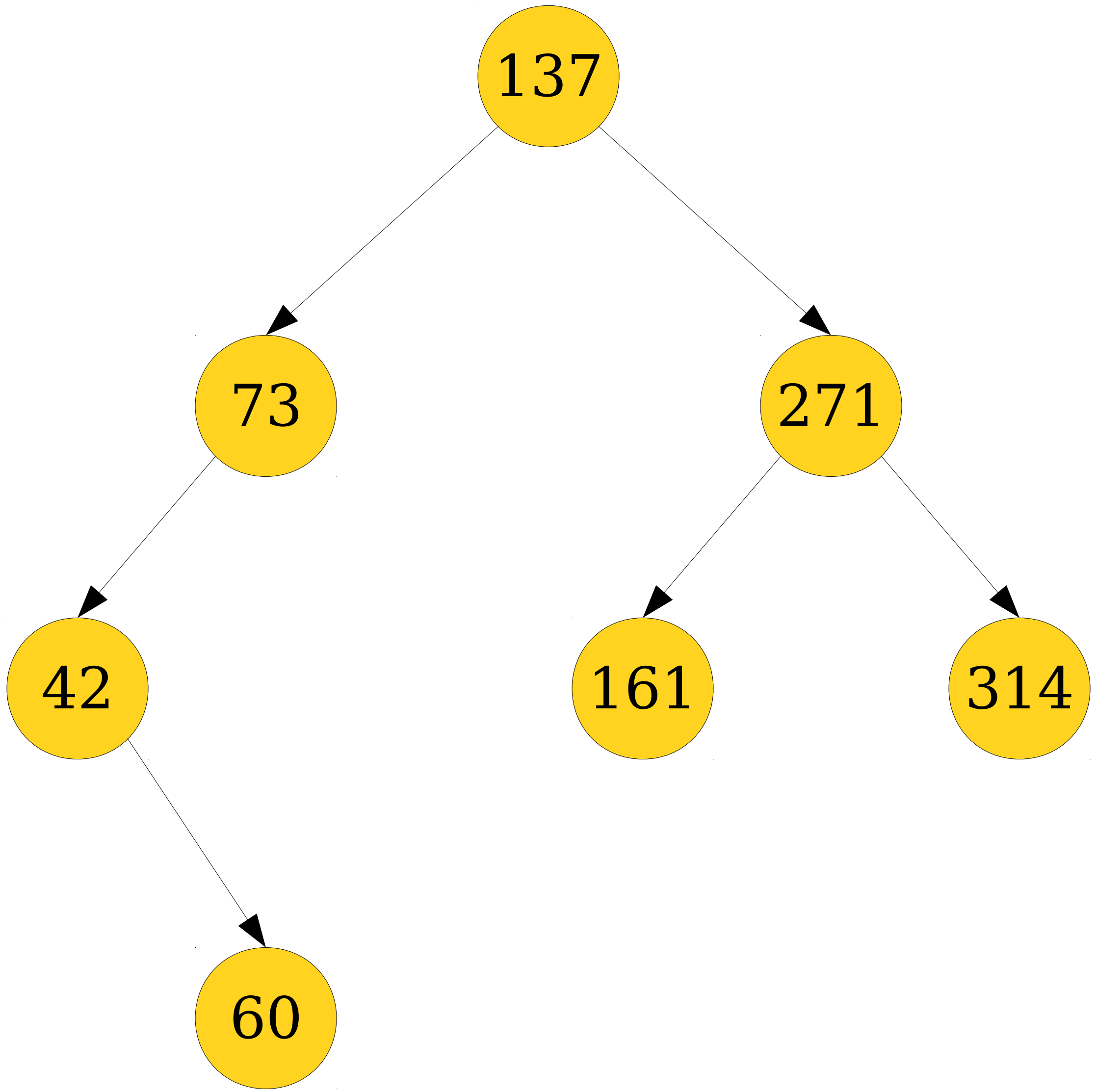
- ***BST Review***
  - Refresher on basic BST concepts and runtimes.
- ***Overview of Red/Black Trees***
  - What we're building toward.
- ***B-Trees and 2-3-4 Trees***
  - A simple balanced tree in depth.
- ***Intuiting Red/Black Trees***
  - A much better feel for red/black trees.

A Quick BST Review

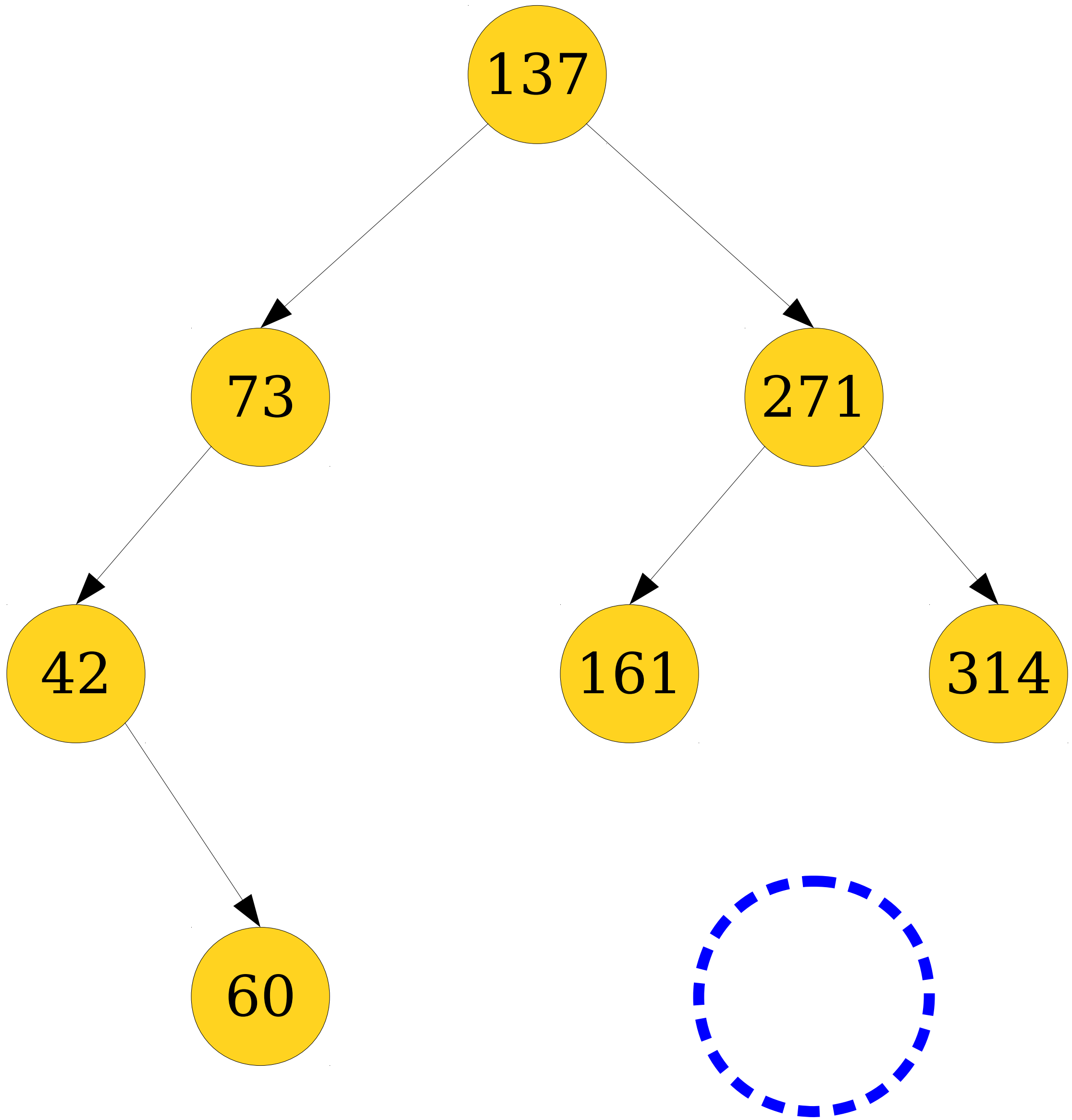
Binary Search Trees

- A ***binary search tree*** is a binary tree with the following properties:
  - Each node in the BST stores a ***key***, and optionally, some auxiliary information.
  - The key of every node in a BST is strictly greater than all keys to its left and strictly smaller than all keys to its right.
- The ***height*** of a binary search tree is the length of the longest path from the root to a leaf, measured in the number of *edges*.
  - A tree with one node has height 0.
  - A tree with no nodes has height -1, by convention.

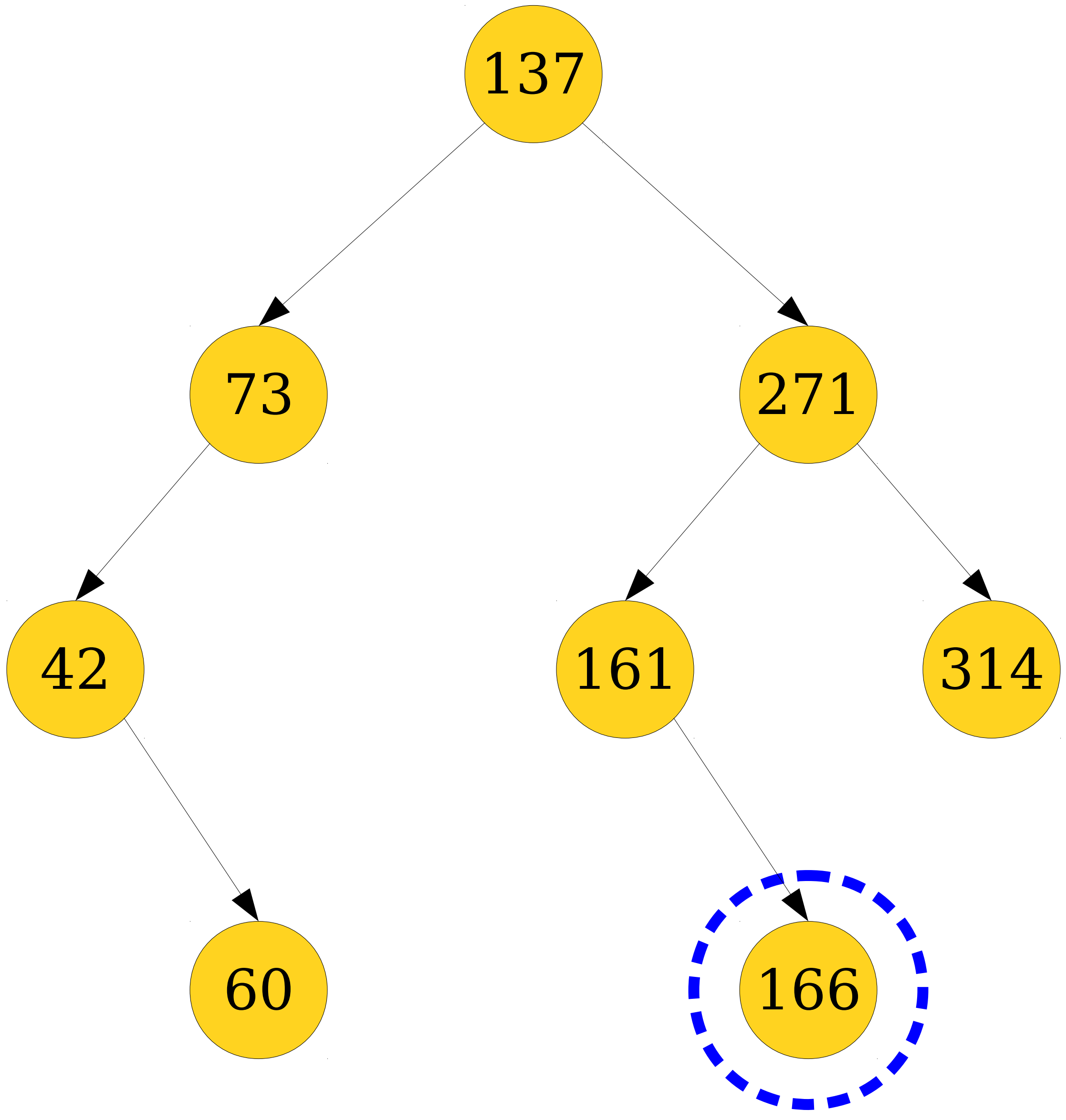
Searching a BST



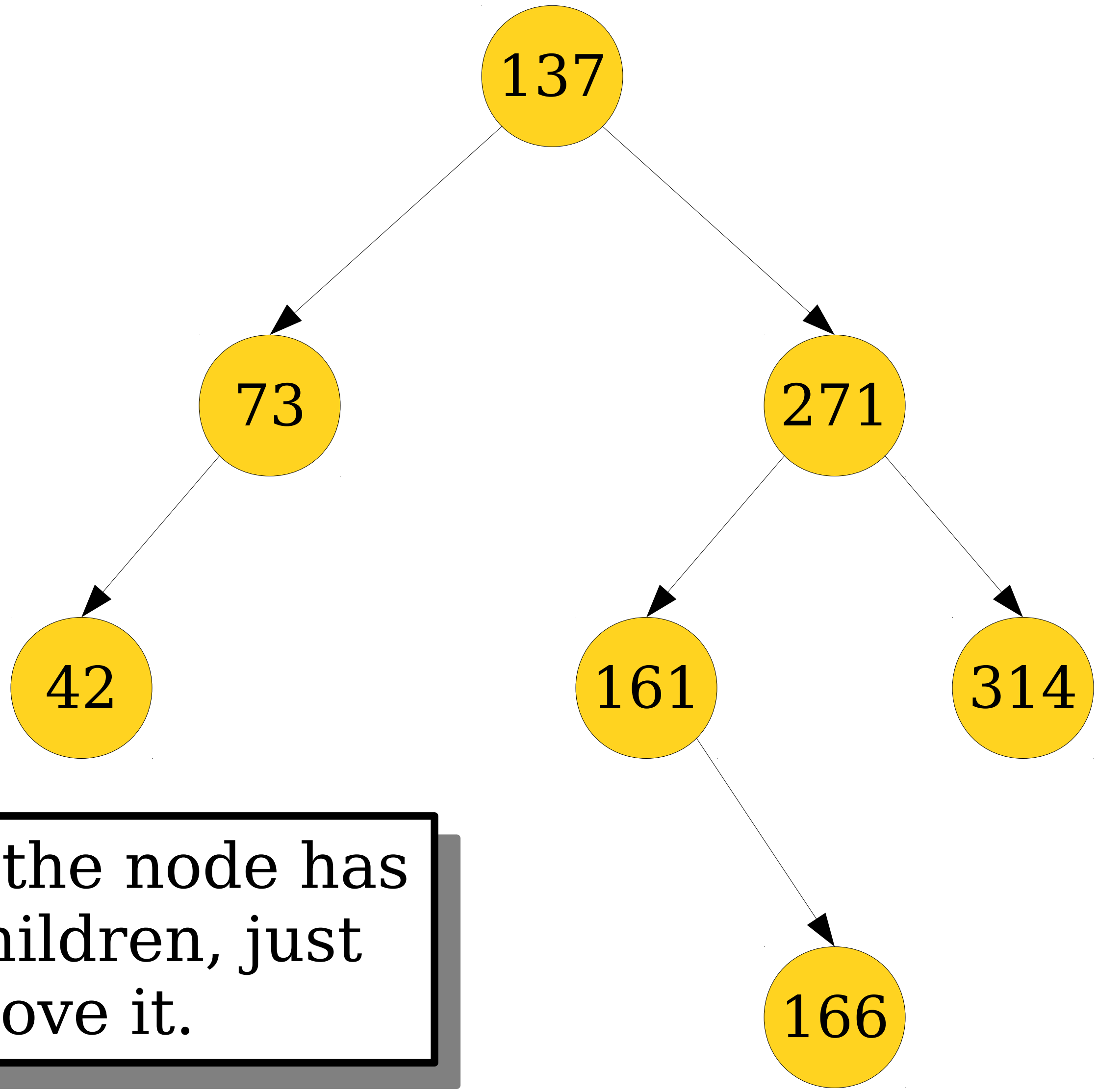
Inserting into a BST



Inserting into a BST

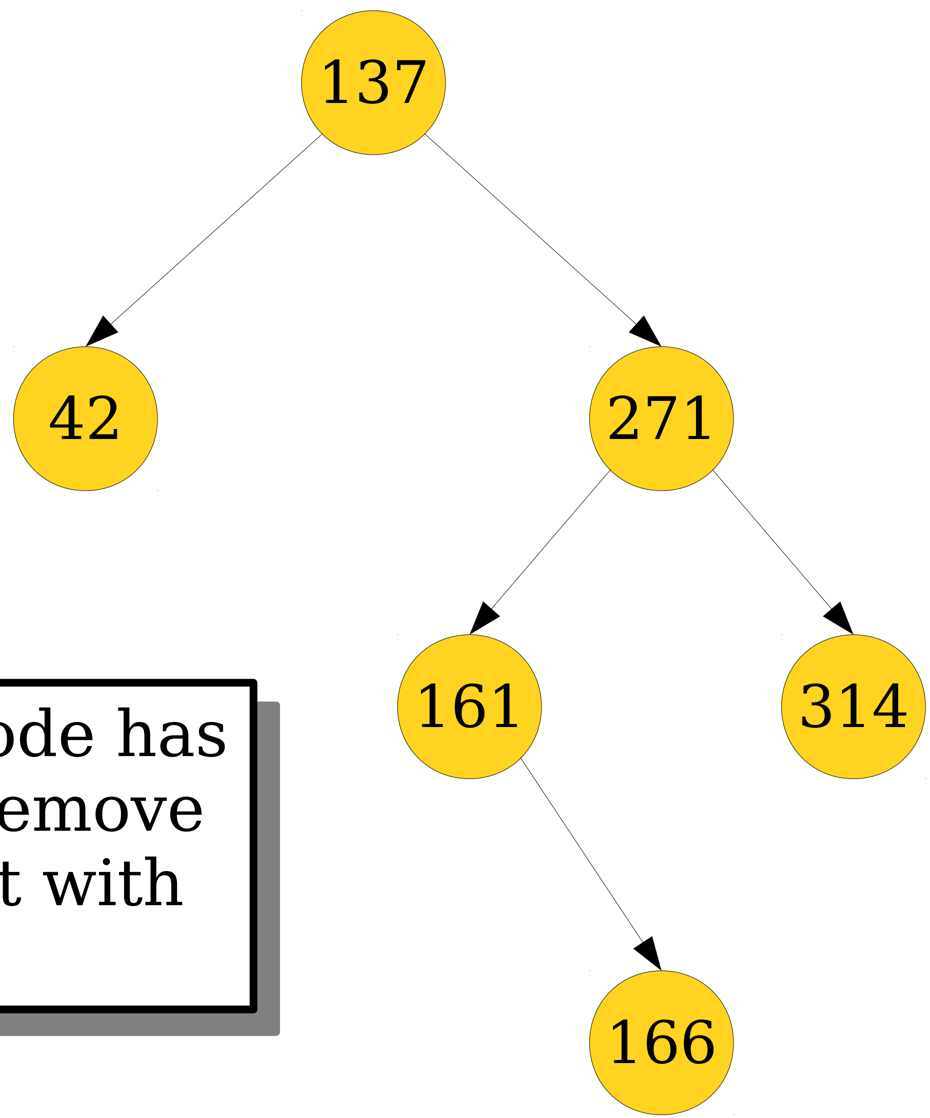


Deleting from a BST



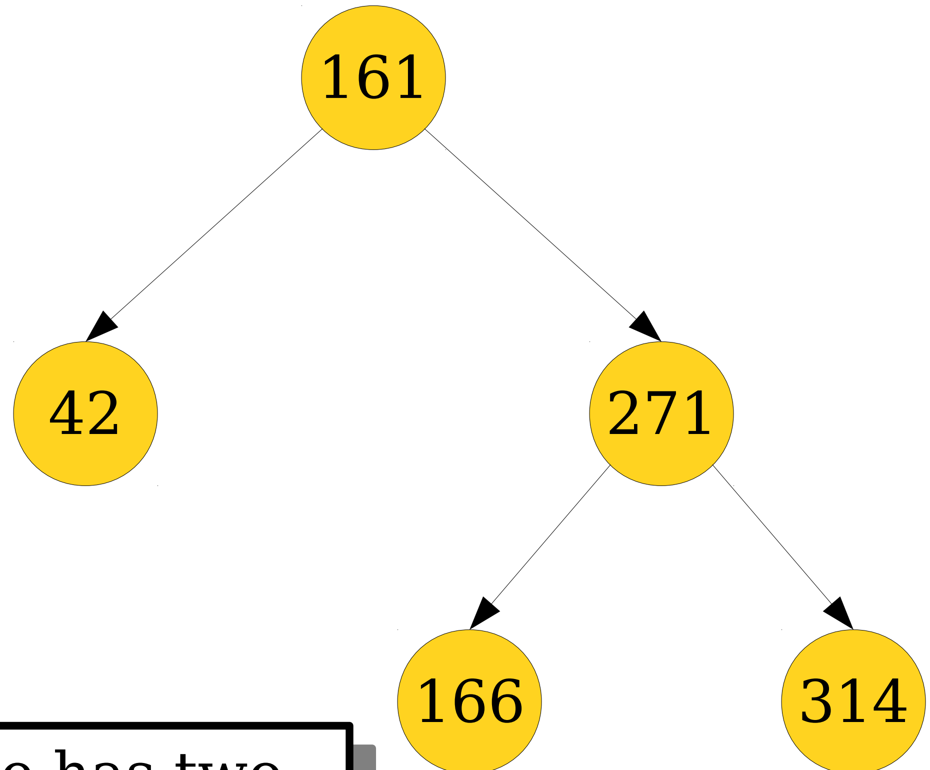
Case 1: If the node has just no children, just remove it.

Deleting from a BST



Case 2: If the node has just one child, remove it and replace it with its child.

Deleting from a BST



Case 3: If the node has two children, find its inorder successor (which has zero or one child), replace the node's key with its successor's key, then delete its successor.

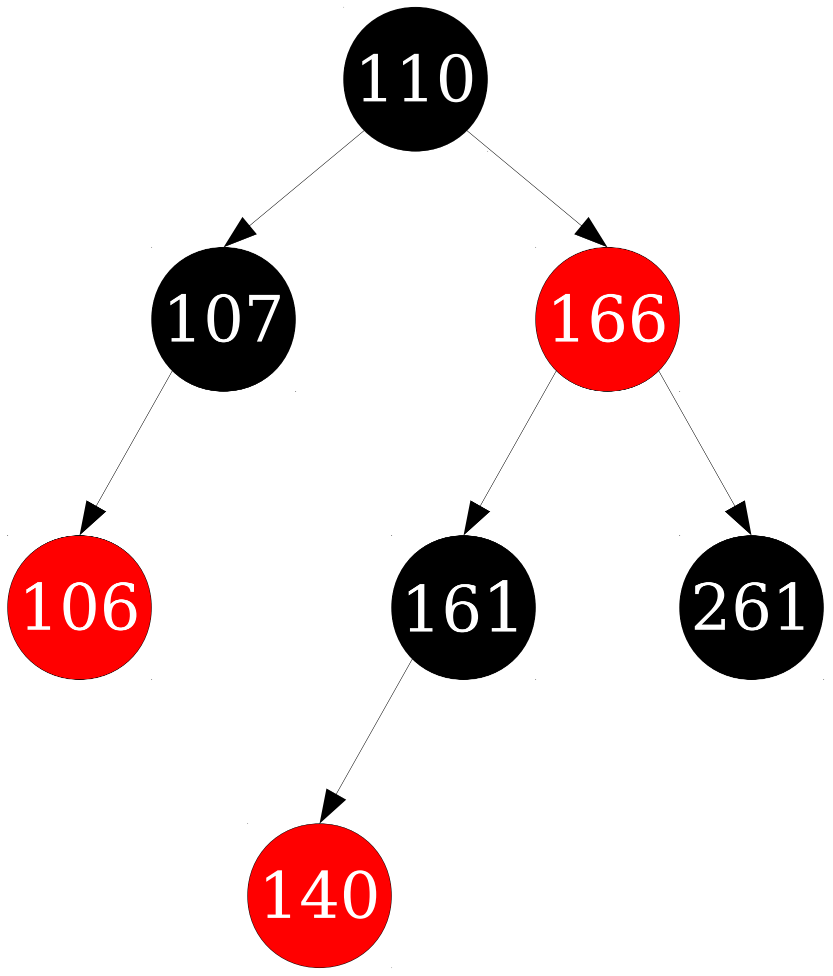
Runtime Analysis

- The time complexity of all these operations is $O(h)$, where h is the height of the tree.
  - That's the longest path we can take.
- In the best case, $h = O(\log n)$ and all operations take time $O(\log n)$.
- In the worst case, $h = \Theta(n)$ and some operations will take time $\Theta(n)$.
- ***Challenge:*** How do you efficiently keep the height of a tree low?

A Glimpse of Red/Black Trees

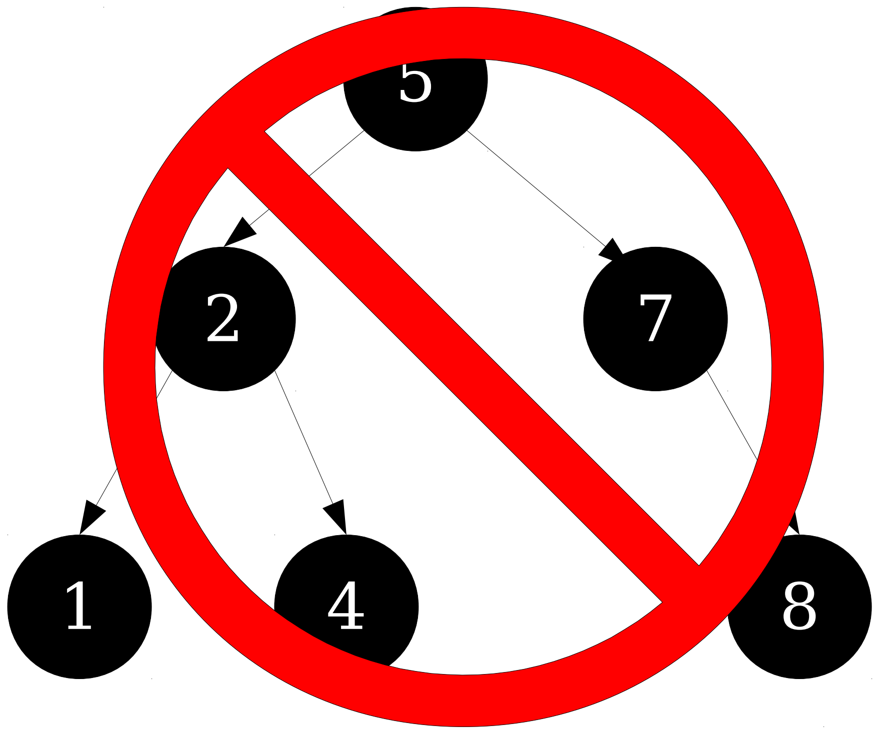
Red/Black Trees

- 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

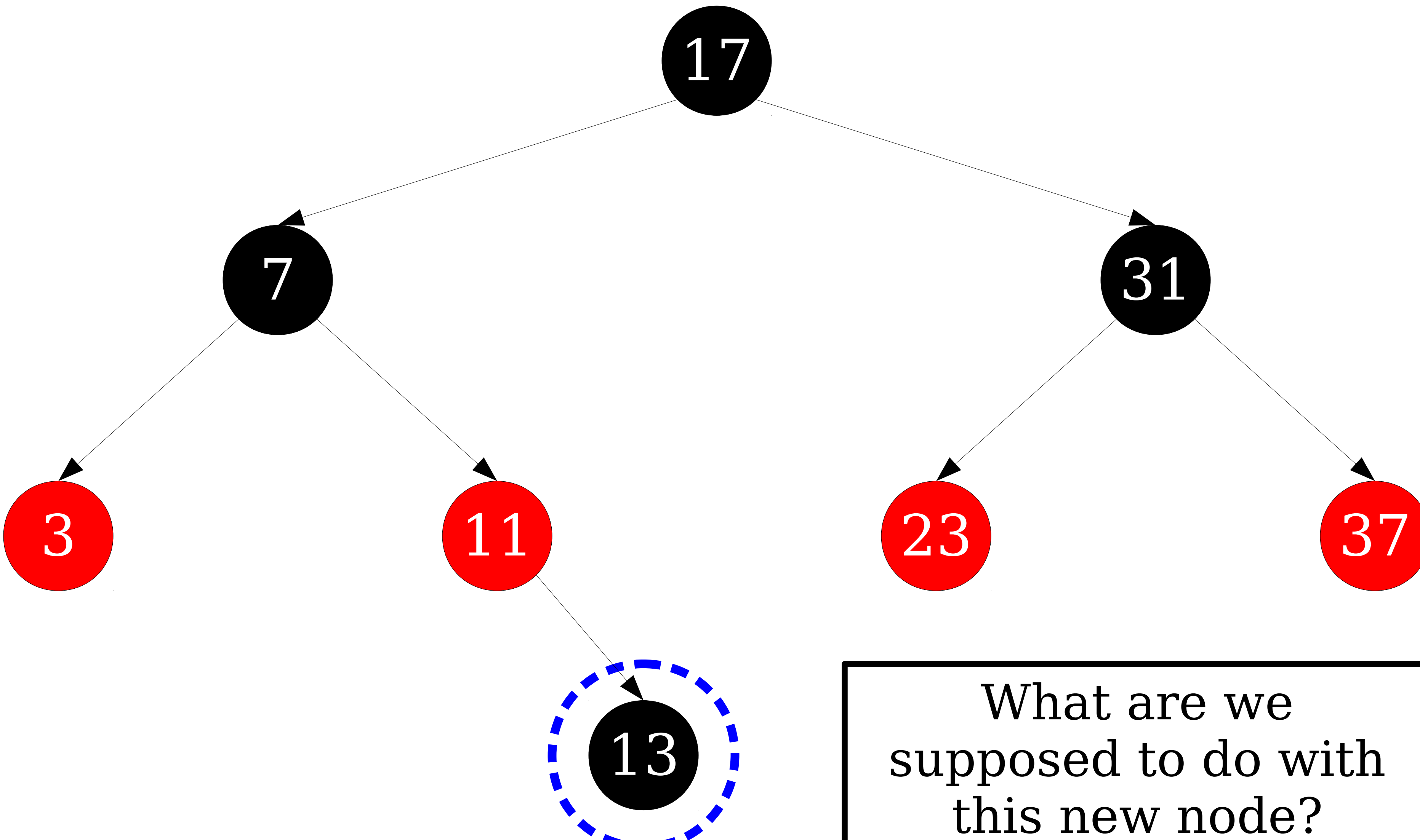
- A ***red/black tree*** is a BST with the following properties:
  - Every node is either red or black.
  - The root is black.
  - No red node has a red child.
  - Every root-null path in the tree passes through the same number of black nodes.



Red/Black Trees

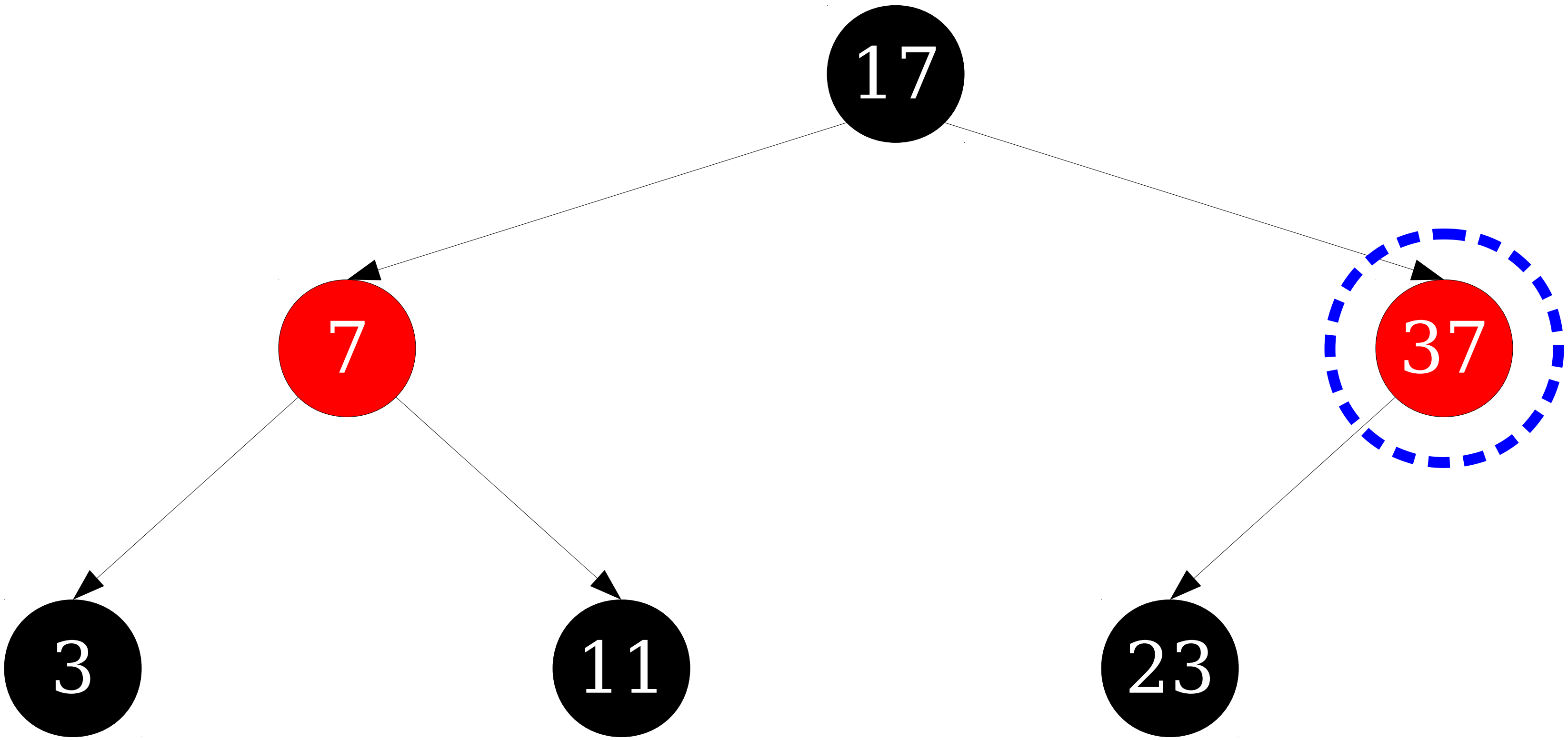
- ***Theorem:*** Any red/black tree with n nodes has height $O(\log n)$.
  - We could prove this now, but there's a *much* simpler proof of this we'll see later on.
- Given a fixed red/black tree, lookups can be done in time $O(\log n)$.

Mutating Red/Black Trees



What are we supposed to do with this new node?

Mutating Red/Black Trees



How do we fix up the black-height property?

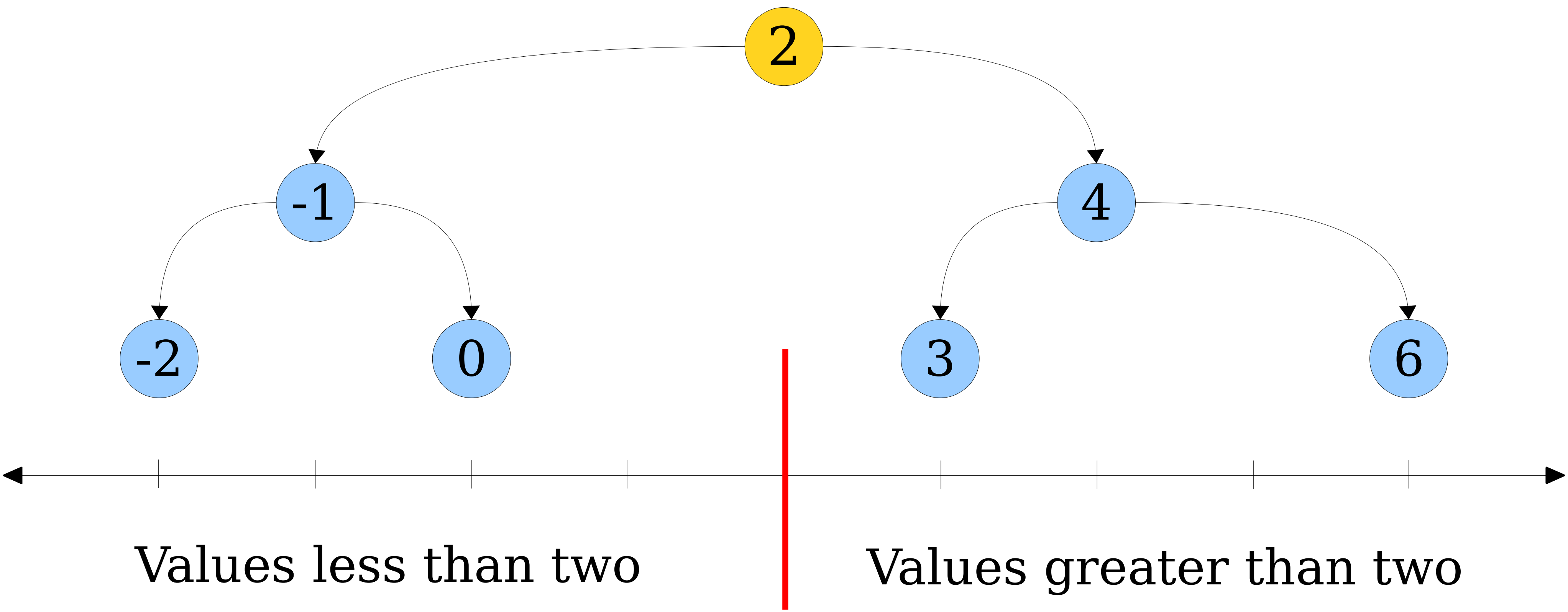
Fixing Up Red/Black Trees

- ***The Good News:*** After doing an insertion or deletion, can locally modify a red/black tree in time O(log *n*) to fix up the red/black properties.
- ***The Bad News:*** There are a *lot* of cases to consider and they're not trivial.
- Some questions:
  - How do you memorize / remember all the different types of rotations?
  - How on earth did anyone come up with red/black trees in the first place?

B-Trees

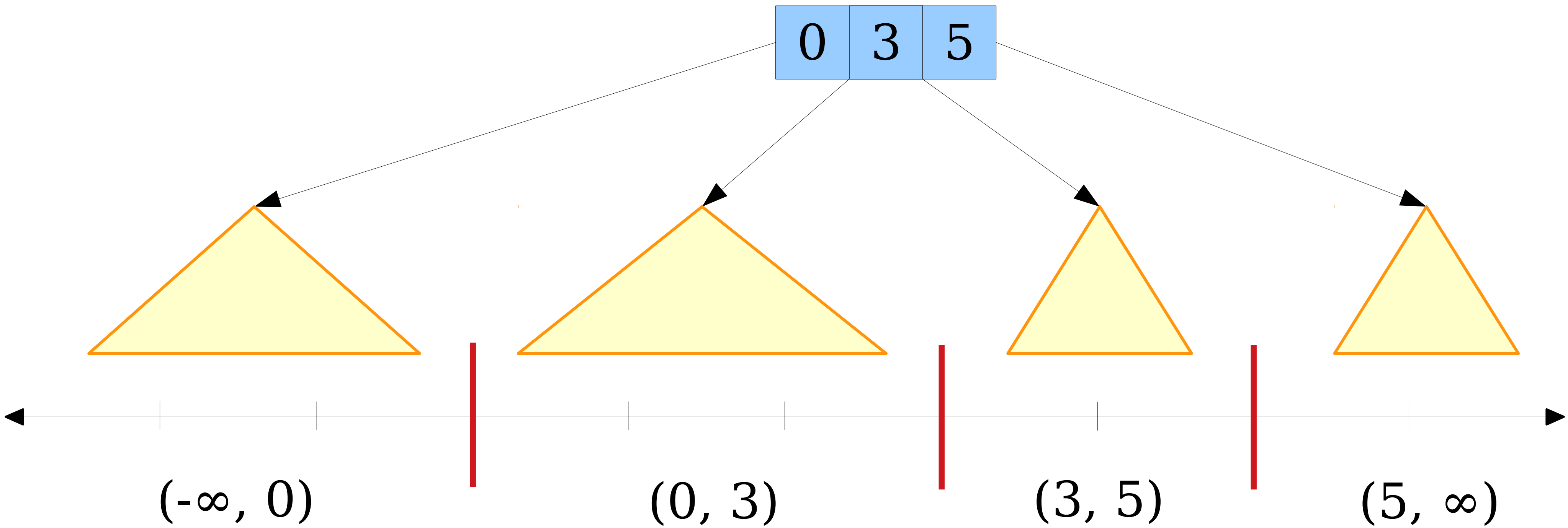
Generalizing BSTs

- In a binary search tree, each node stores a single key.
- That key splits the "key space" into two pieces, and each subtree stores the keys in those halves.



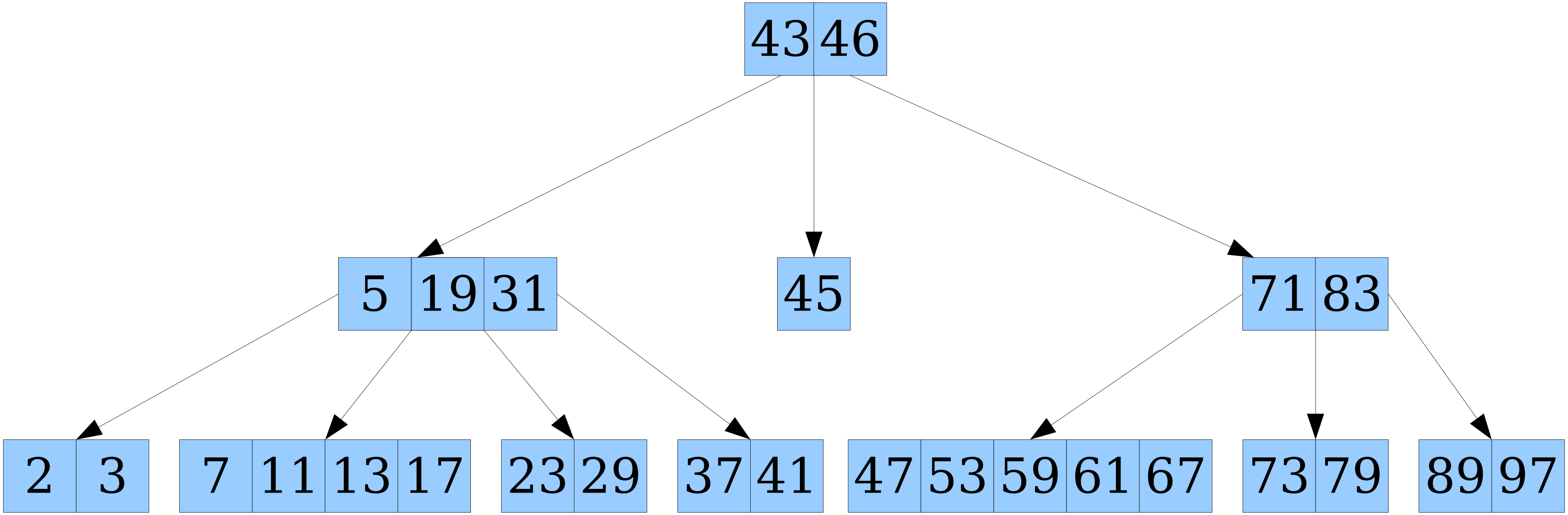
Generalizing BSTs

- In a ***multiway search tree***, each node stores an arbitrary number of keys in sorted order.
- A node with k keys splits the key space into $k+1$ regions, with subtrees for keys in each region.



Generalizing BSTs

- In a ***multiway search tree***, each node stores an arbitrary number of keys in sorted order.



- Surprisingly, it's a bit easier to build a balanced multiway tree than it is to build a balanced BST. Let's see how.

Balanced Multiway Trees

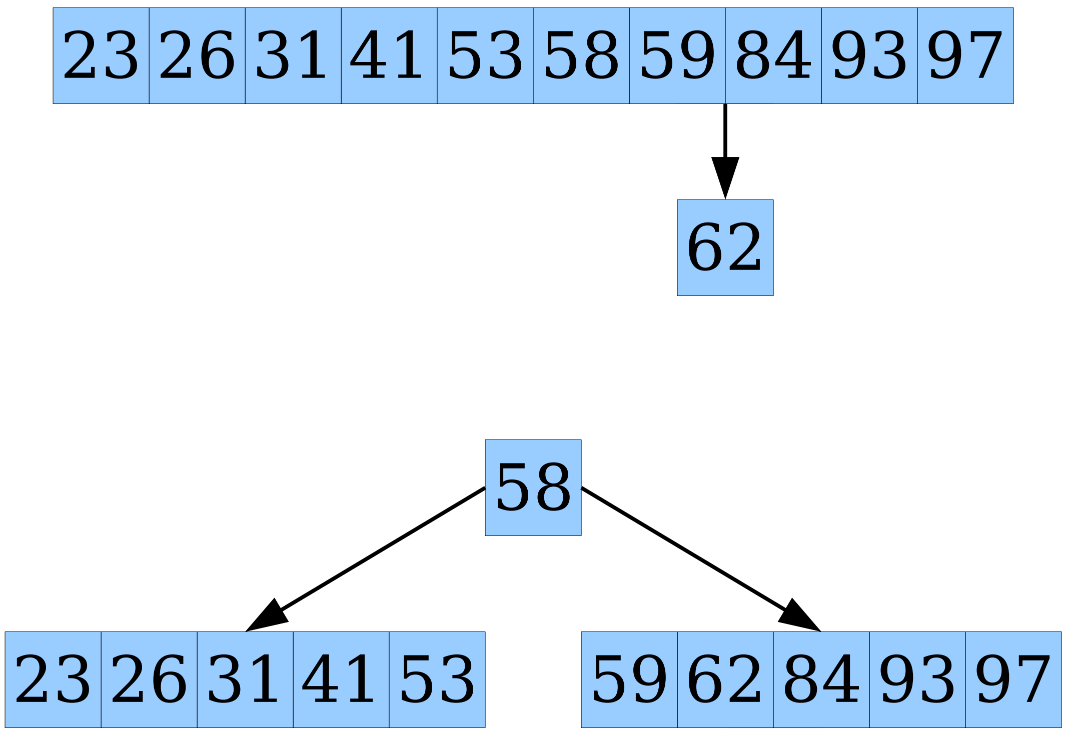
- In some sense, building a balanced multiway tree isn't all that hard.
- We can always just cram more keys into a single node!

23	26	31	41	53	58	59	62	84	93	97

- At a certain point, this stops being a good idea – it's basically just a sorted array.

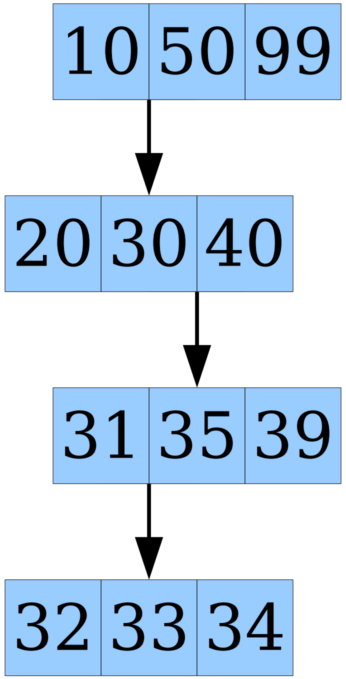
Balanced Multiway Trees

- What could we do if our nodes get too big?
- ***Option 1:*** Push keys down into new nodes.
- ***Option 2:*** Split big nodes, kicking keys higher up.
- What are some advantages of each approach?
- What are some disadvantages?



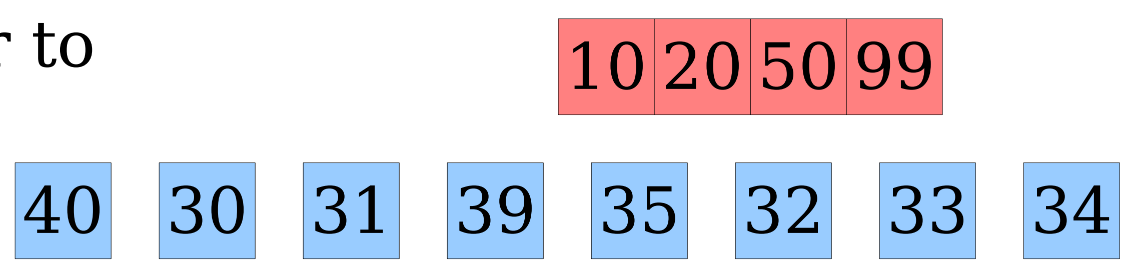
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.



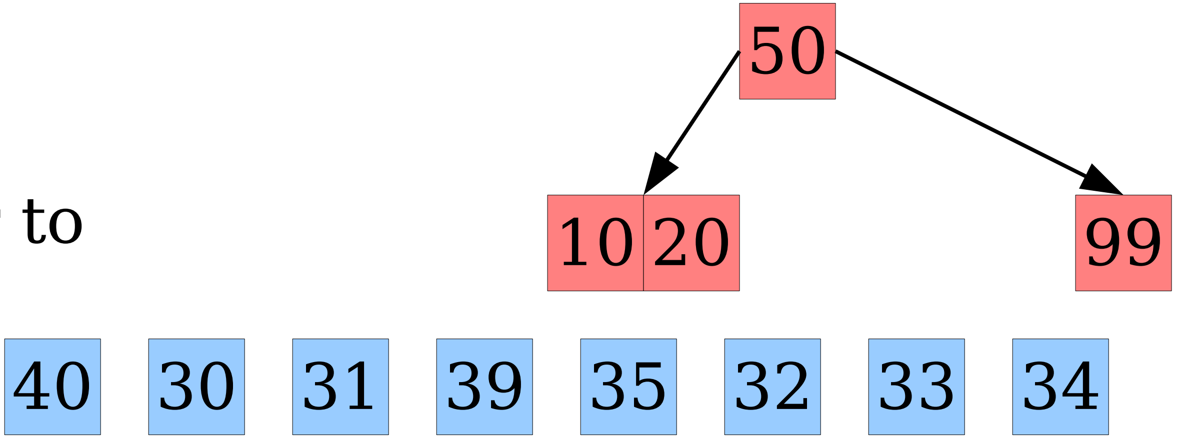
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



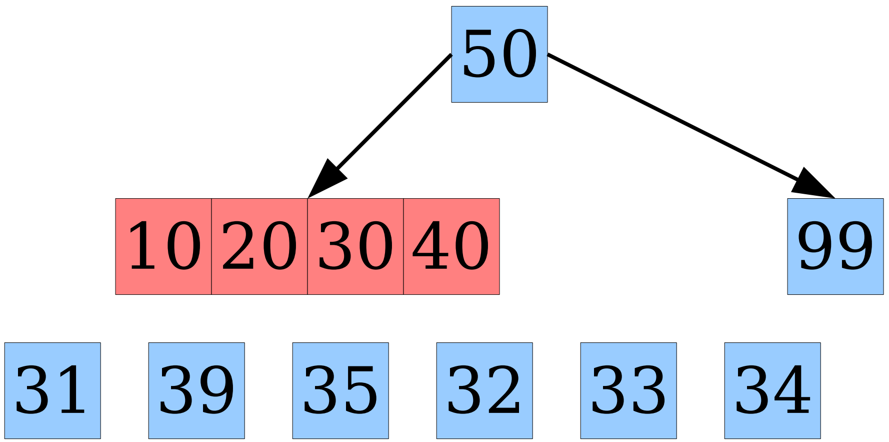
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



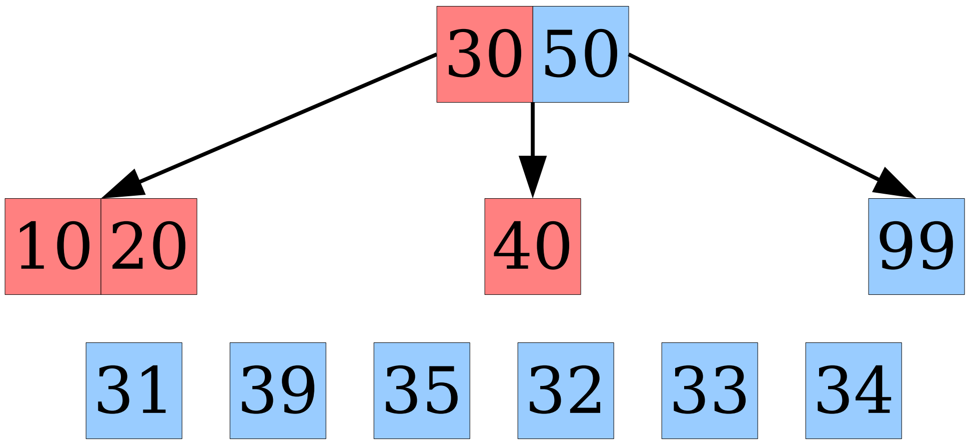
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



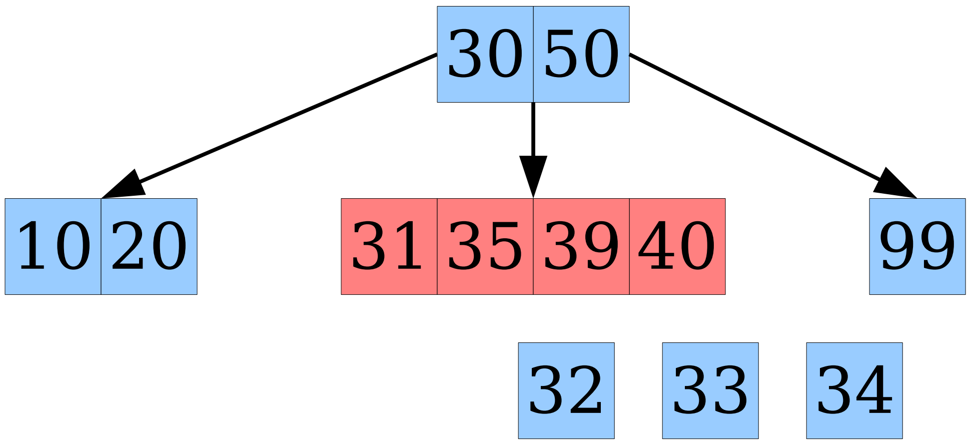
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



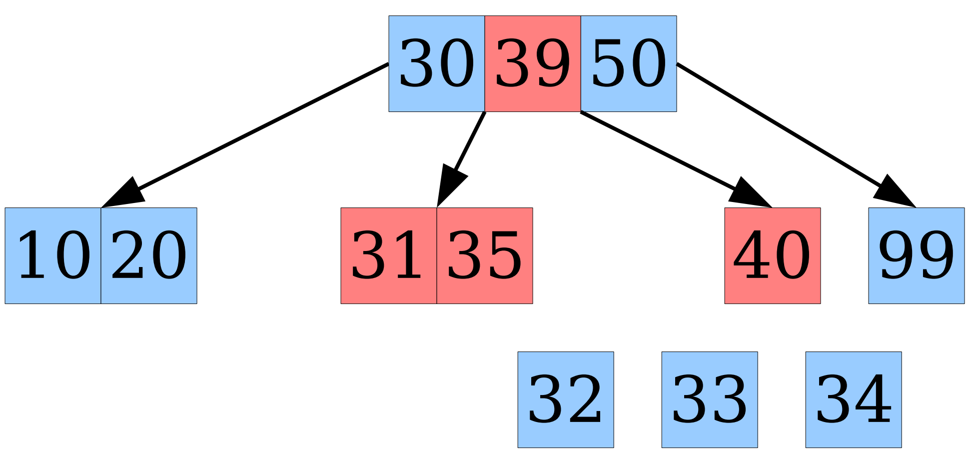
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



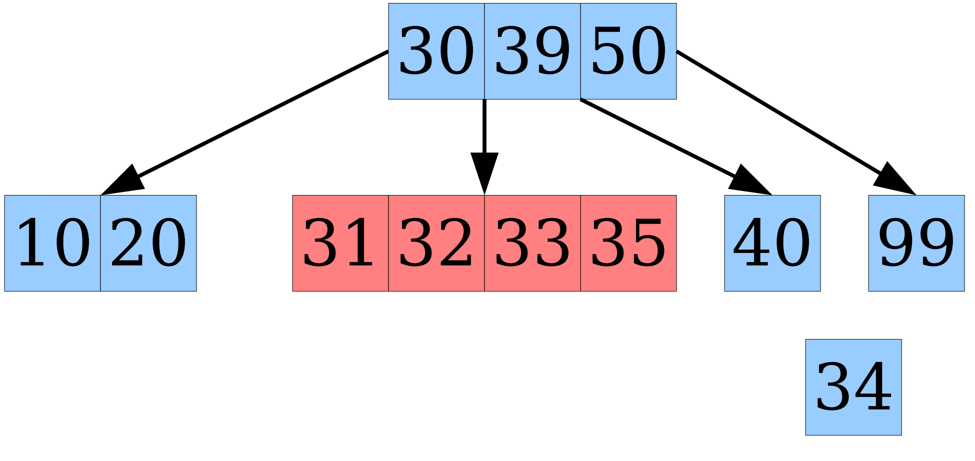
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



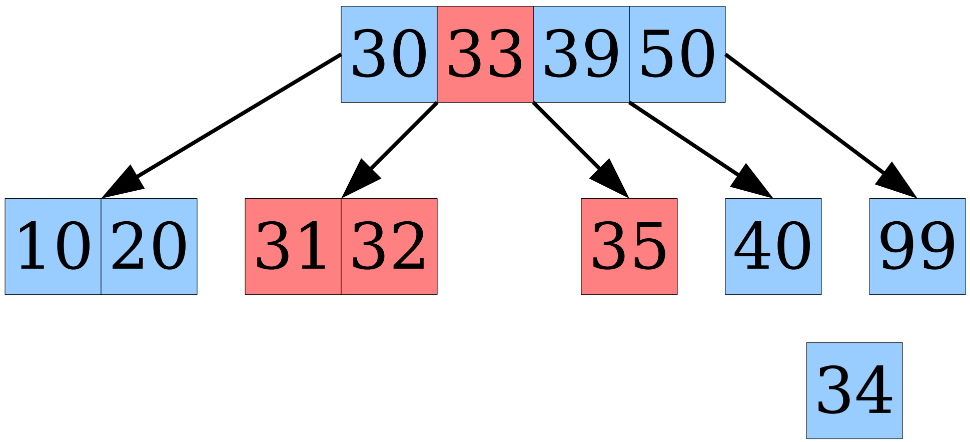
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



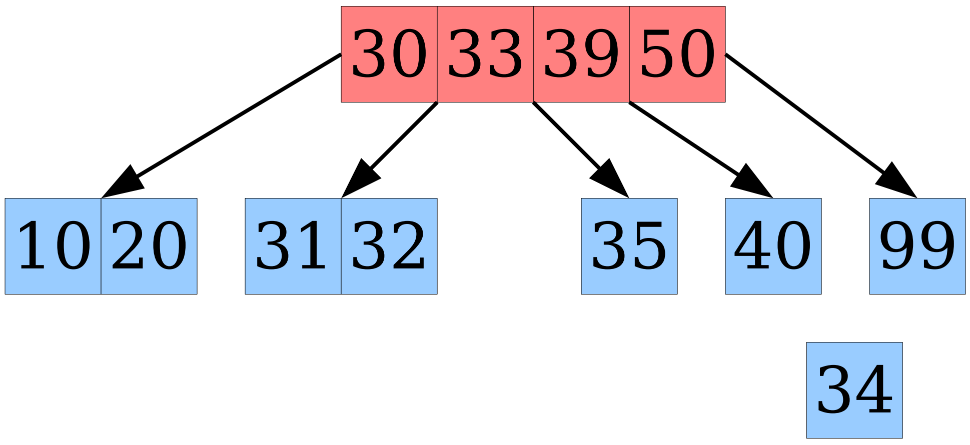
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



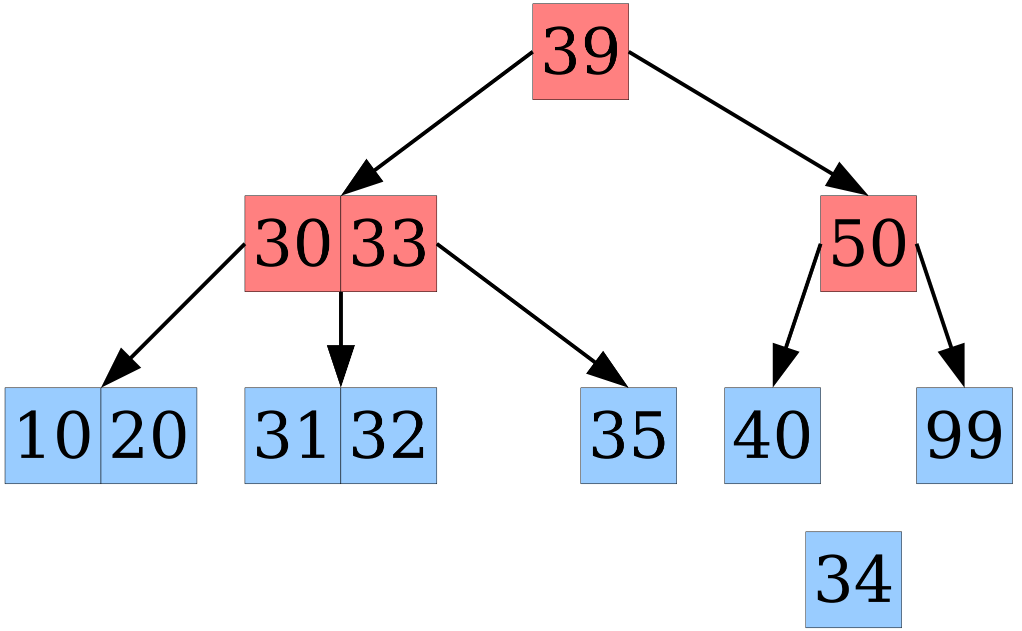
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



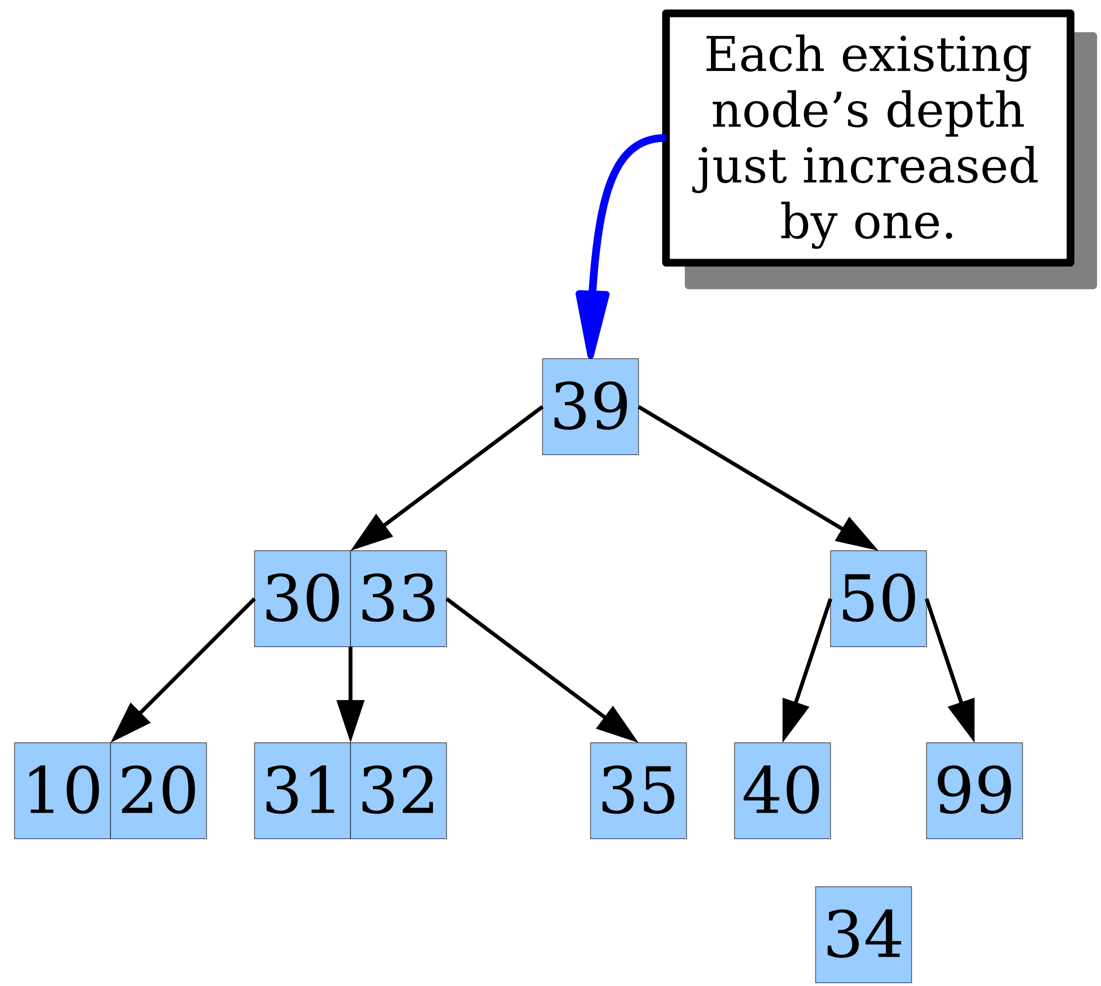
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



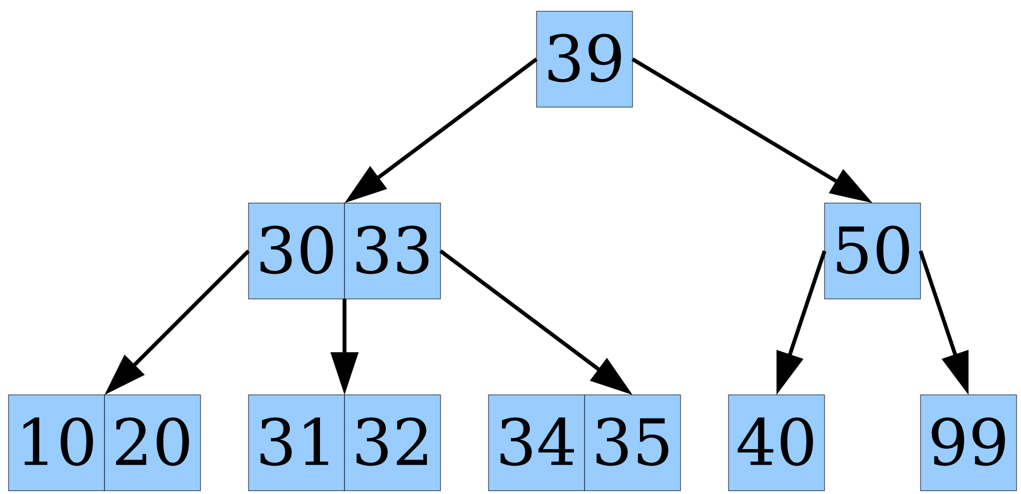
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



Balanced Multiway Trees

- ***General idea:*** Keep nodes holding roughly between b and $2b$ keys, for some parameter b.
  - (Exception: the root node can have fewer keys.)
- If a node gets too big, split it and kick a key higher up.



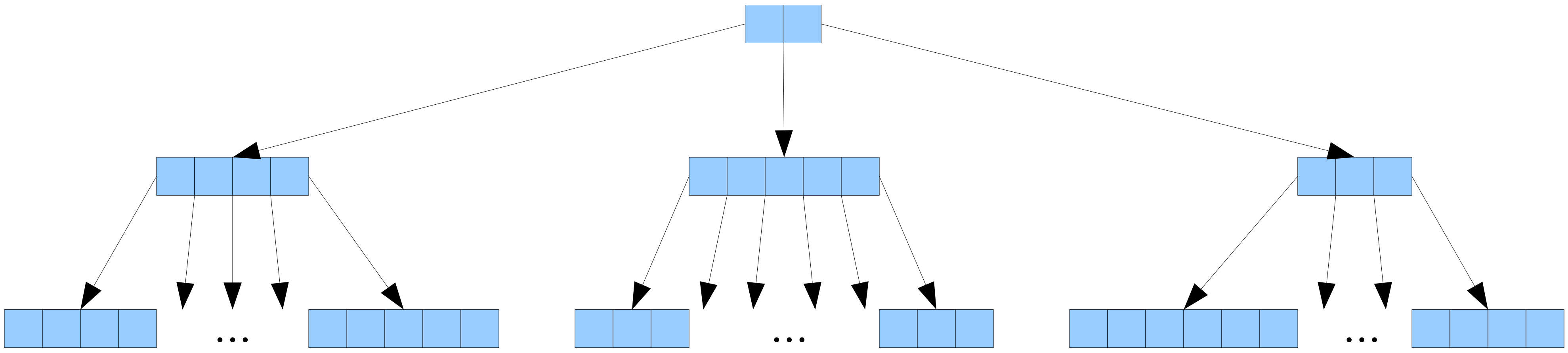
- ***Advantage 1:*** The tree is always balanced.
- ***Advantage 2:*** Insertions and lookups are pretty fast.

Balanced Multiway Trees

- We currently have a ***mechanical definition*** of how these balanced multiway trees work:
  - Nodes should have between roughly b and $2b$ keys in them.
  - Split nodes when they get too big and propagate the splits upward.
- We currently don't have an ***operational definition*** of how these balanced multiway trees work.
  - e.g. "A Cartesian tree for an array is a binary tree that's a min-heap and whose inorder traversal gives back the original array."
  - e.g. "A suffix tree is a Patricia trie with one node for each suffix and branching word of T."

B-Trees

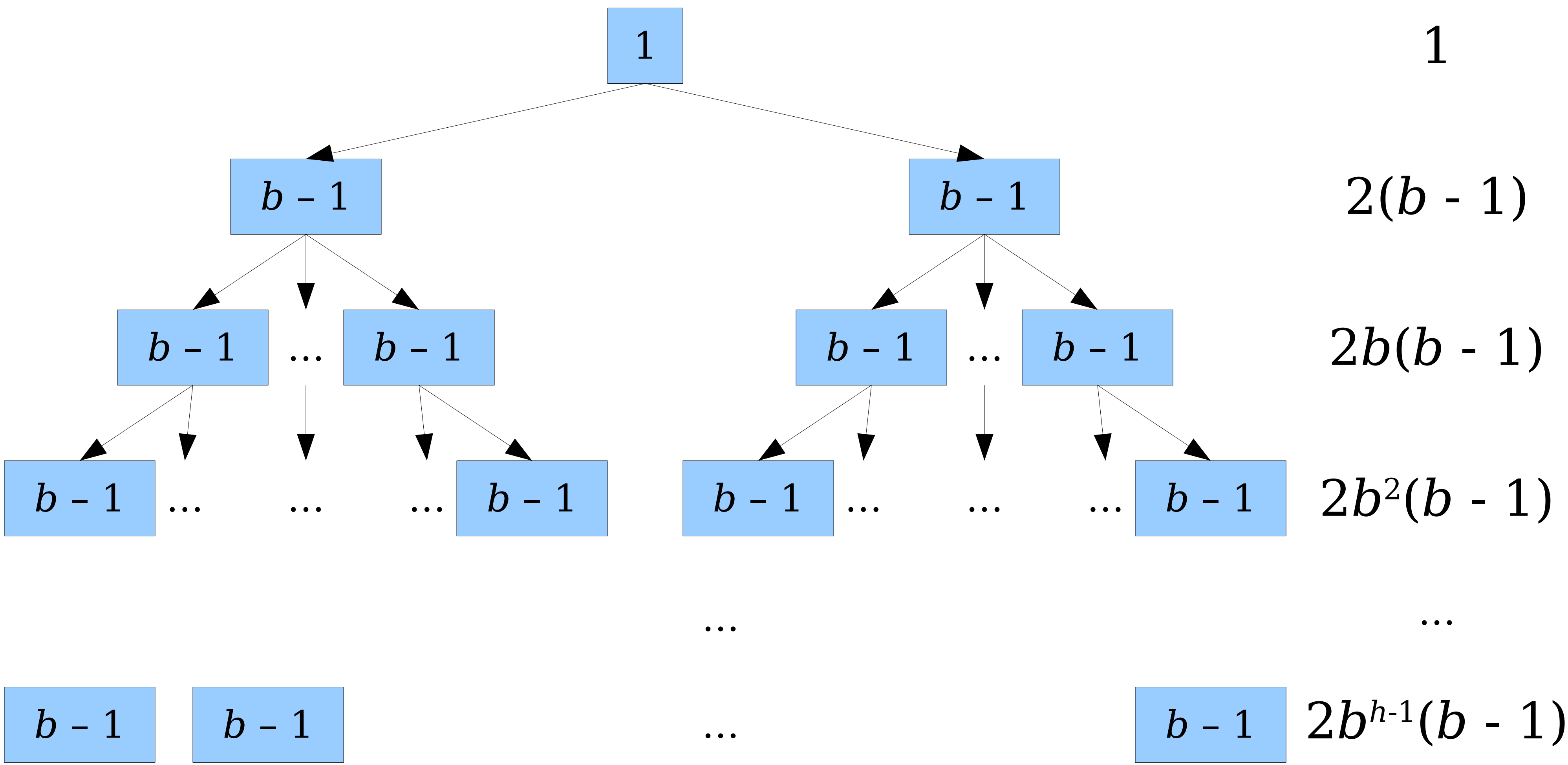
- A ***B-tree of order b*** is a multiway search tree where
  - each node has (roughly) between b and $2b$ keys, except the root, which may have beween 1 and b keys;
  - each node is either a leaf or has one more child than key; and
  - all leaves are at the same depth.
- Different authors give different bounds on how many keys can be in each node. The ranges are often [b–1, 2b–1] or [b, 2b]. For the purposes of today's lecture, we'll use the range [b-1, 2b-1] for the key limits, just for simplicity.



Analyzing Multiway Trees

The Height of a B-Tree

- What is the maximum possible height of a B-tree of order *b*?



The Height of a B-Tree

- ***Theorem:*** The maximum height of a B-tree of order b containing n keys is $\log_b ((n + 1) / 2)$.
- ***Proof:*** Number of keys n in a B-tree of height h is guaranteed to be at least

$$1 + 2(b - 1) + 2b(b - 1) + 2b^2(b - 1) + \ldots + 2b^{h-1}(b - 1)$$

$$= 1 + 2(b - 1)(1 + b + b^2 + \ldots + b^{h-1})$$

$$= 1 + 2(b - 1)((b^h - 1) / (b - 1))$$

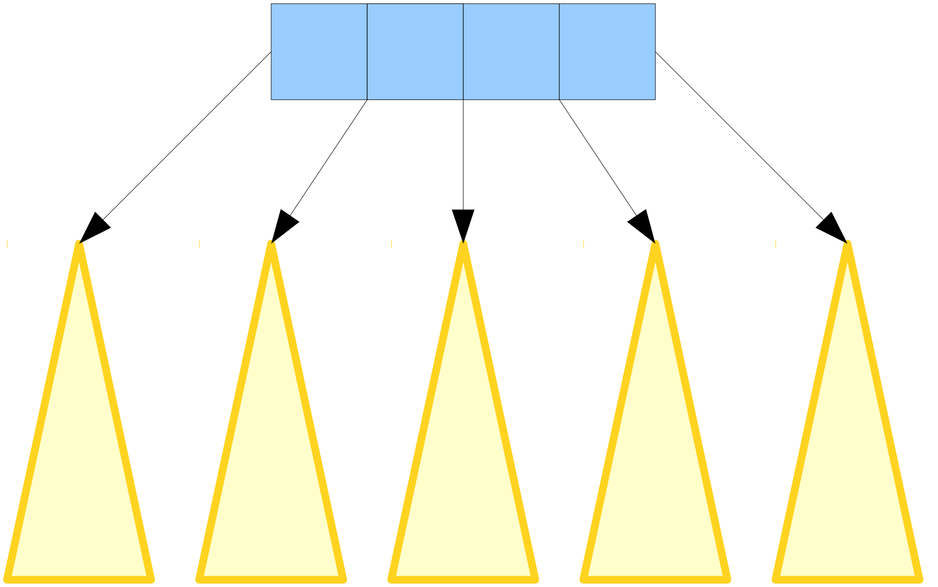
$$= 1 + 2(b^h - 1) = 2b^h - 1.$$

  Solving $n = 2b^h - 1$ yields $h = \log_b ((n + 1) / 2)$. ■

- ***Corollary:*** B-trees of order b have height $\Theta(\log_b n)$.

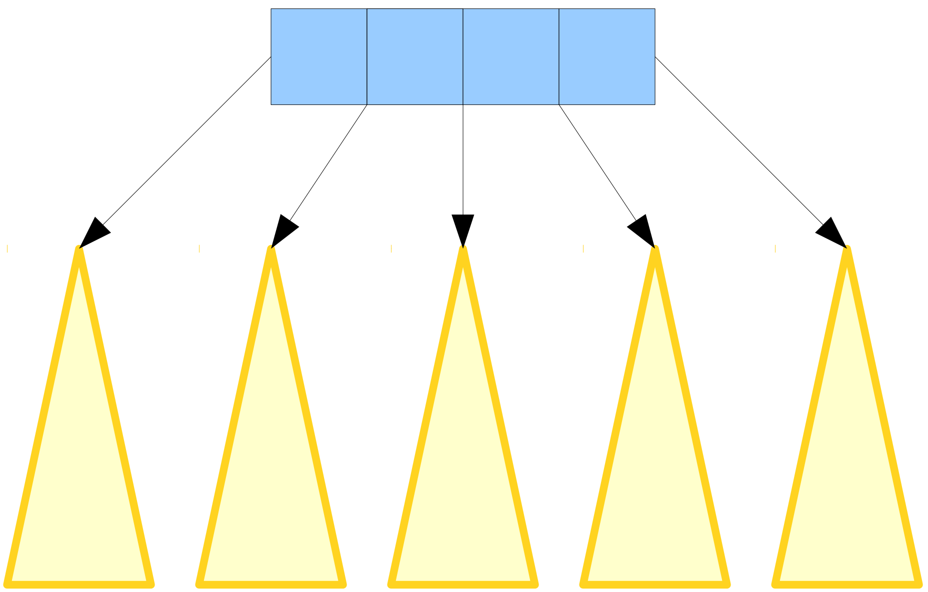
Analyzing Efficiency

- Suppose we have a B-tree of order *b*.
- What is the worst-case runtime of looking up a key in the B-tree?
- ***Answer:*** It depends on how we do the search!



Analyzing Efficiency

- To do a lookup in a B-tree, we need to determine which child tree to descend into.
- This means we need to compare our query key against the keys in the node.
- ***Question:*** How should we do this?

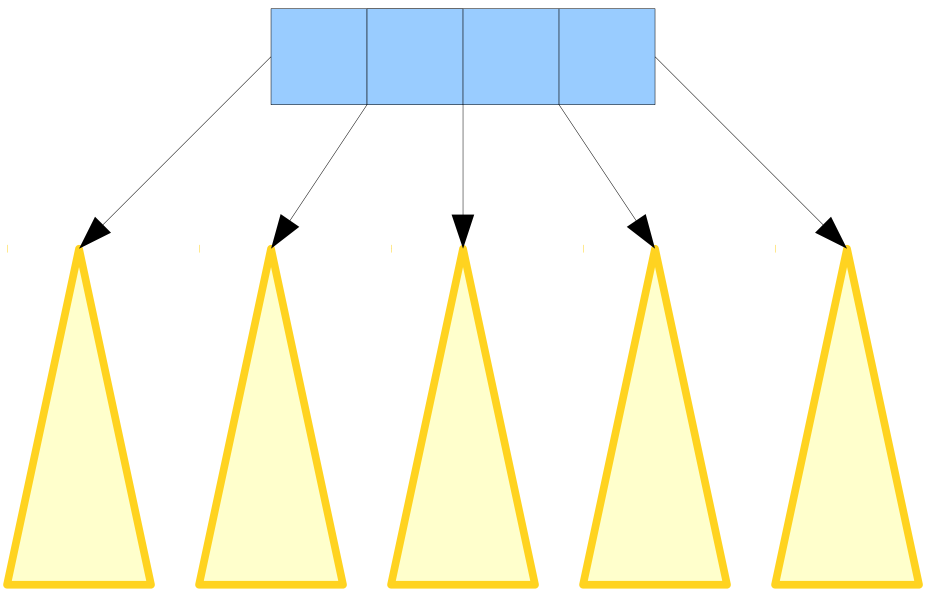


Analyzing Efficiency

- ***Option 1:*** Use a linear search!
- Cost per node: O(b).
- Nodes visited: O($\log_b n$).
- Total cost:

$$O(b) \cdot O(\log_b n)$$

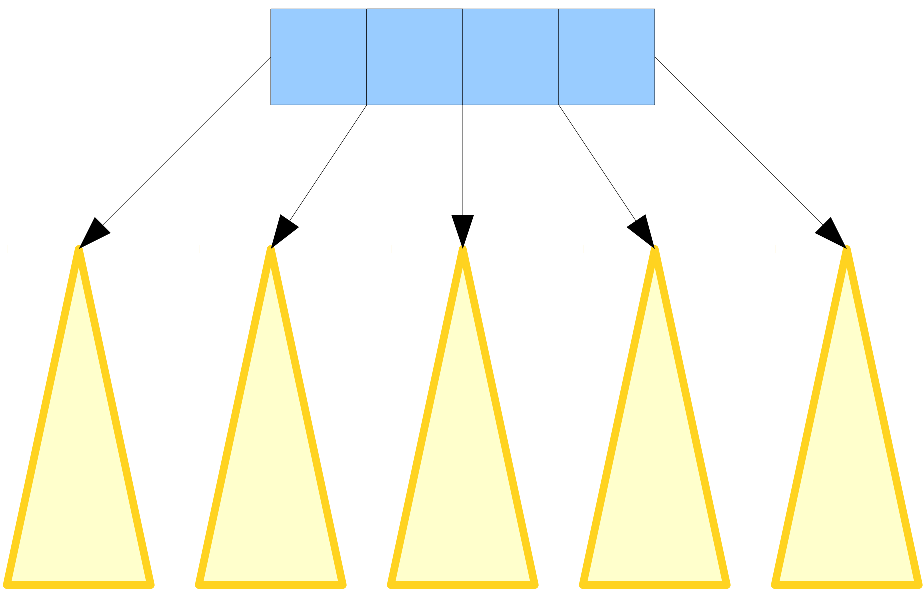
$$= \mathbf{O(b \log_b n)}$$



Analyzing Efficiency

- ***Option 2:*** Use a binary search!
- Cost per node: O(log b).
- Nodes visited: O(log$_b$ n).
- Total cost:

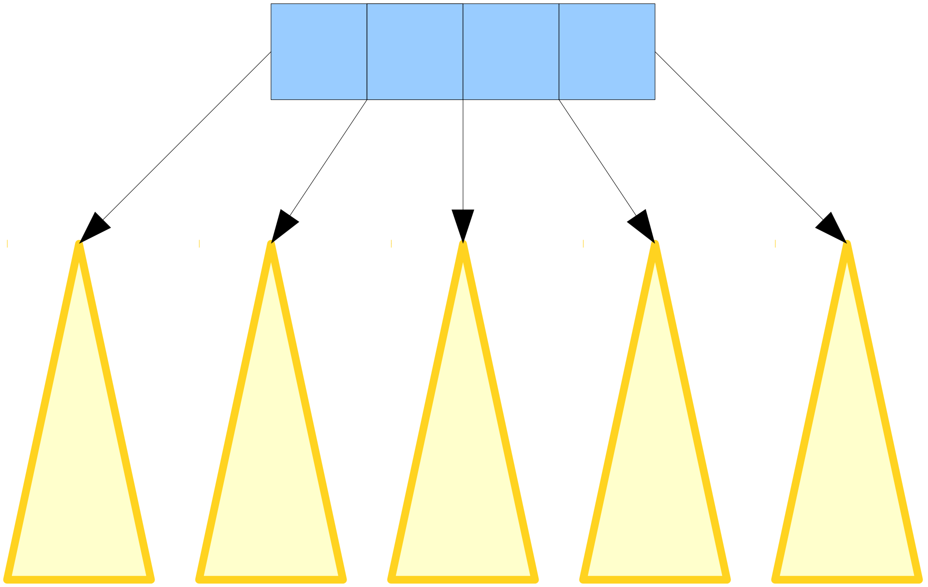
$$\begin{aligned} & O(\log b) \cdot O(\log_b n) \\ = \; & O(\log b \cdot \log_b n) \\ = \; & O(\log b \cdot (\log n) / (\log b)) \\ = \; & \mathbf{O(\log n)}. \end{aligned}$$



Intuition: We can't do better than O(log n) for arbitrary data, because it's the information-theoretic minimum number of comparisons needed to find something in a sorted collection!

Analyzing Efficiency

- Suppose we have a B-tree of order b.
- What is the worst-case runtime of inserting a key into the B-tree?
- Each insertion visits $O(\log_b n)$ nodes, and in the worst case we have to split every node we see.
- ***Answer:*** $O(b \log_b n)$.



Analyzing Efficiency

- The cost of an insertion in a B-tree of order b is $O(b \log_b n)$.
- What's the best choice of b to use here?
- Note that

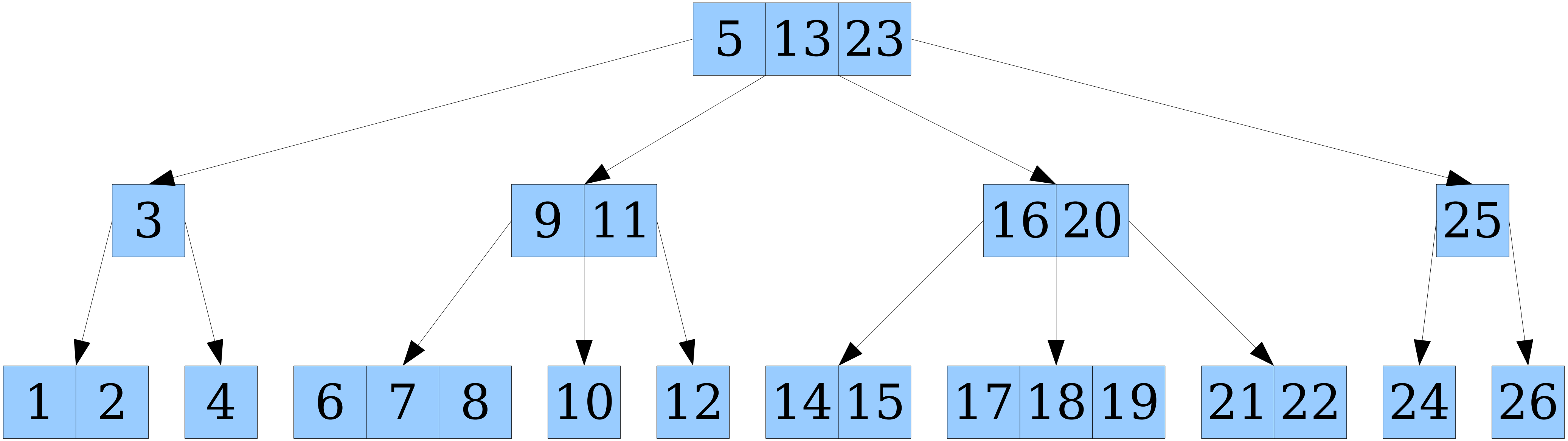
$$\begin{aligned} & b \log_{\mathrm{b}} n \\ = & \ b (\log n / \log b) \\ = & \ (b / \log b) \log n. \end{aligned}$$

Fun fact: This is the same time bound you'd get if you used a b-ary heap instead of a binary heap for a priority queue.

- What choice of b minimizes $b / \log b$?
- ***Answer:*** Pick $b = e$.

2-3-4 Trees

- A ***2-3-4 tree*** is a B-tree of order 2. Specifically:
  - each node has between 1 and 3 keys;
  - each node is either a leaf or has one more child than key; and
  - all leaves are at the same depth.
- You actually saw this B-tree earlier! It’s the type of tree from our insertion example.



The Story So Far

- A B-tree supports
  - lookups in time O(log n), and
  - insertions in time O($b \log_b n$).
- Picking b to be around 2 or 3 makes this optimal in Theoryland.
  - The 2-3-4 tree is great for that reason.
- ***Plot Twist:*** In practice, you most often see choices of b like 1,024 or 4,096.
- ***Question:*** Why would anyone do that?


Theoryland
IRL

The Memory Hierarchy

Memory Tradeoffs

- There is an enormous tradeoff between *speed* and *size* in memory.
- SRAM (the stuff registers are made of) is fast but very expensive:
  - Can keep up with processor speeds in the GHz.
  - As of 2010, cost is $5/MB. *(Anyone know a good source for a more recent price?)*
  - Good luck buying 1TB of the stuff!
- Hard disks are cheap but very slow:
  - As of 2019, you can buy a 4TB hard drive for about $70.
  - As of 2019, good disk seek times for magnetic drives are measured in ms (about two to four million times slower than a processor cycle!)

The Memory Hierarchy

- ***Idea:*** Try to get the best of all worlds by using multiple types of memory.

Registers	256B - 8KB	0.25 – 1ns
L1 Cache	16KB – 64KB	1ns – 5ns
L2 Cache	1MB - 4MB	5ns – 25ns
Main Memory	4GB – 256GB	25ns – 100ns
Hard Disk	1TB+	3 – 10ms
Network (The Cloud)	*Lots*	10 – 2000ms

External Data Structures

- Suppose you have a data set that's *way* too big to fit in RAM.
- The data structure is on disk and read into RAM as needed.
- Data from disk doesn't come back one *byte* at a time, but rather one *page* at a time.
- ***Goal:*** Minimize the number of disk reads and writes, not the number of instructions executed.

"Please give me 4KB starting at location *addr1*"

110111001011101111 0001…

External Data Structures

- Because B-trees have a huge branching factor, they're great for on-disk storage.
  - Disk block reads/writes are glacially slow.
  - The high branching factor minimizes the number of blocks to read during a lookup.
  - Extra work scanning inside a block offset by these savings.
- Major use cases for B-trees and their variants (B^+-trees, H-trees, etc.) include
  - databases (huge amount of data stored on disk);
  - file systems (ext4, NTFS, ReFS); and, recently,
  - in-memory data structures (due to cache effects).

Analyzing B-Trees

- Suppose we tune *b* so that each node in the B-tree fits inside a single disk page.
- We *only* care about the number of disk pages read or written.
  - It's so much slower than RAM that it'll dominate the runtime.
- ***Question:*** What is the cost of a lookup in a B-tree in this model?
  - Answer: The height of the tree, $O(\log_b n)$.
- ***Question:*** What is the cost of inserting into a B-tree in this model?
  - Answer: The height of the tree, $O(\log_b n)$.

Analyzing B-Trees

- The cost model we use will change our overall analysis.
- Cost is number of operations:

$$O(\log n) \text{ / lookup, } O(b \log_{\mathrm{b}} n) \text{ / insertion.}$$

- Cost is number of blocks accessed:

$$O(\log_b n) \text{ / lookup, } O(\log_b n) \text{ / insertion.}$$

- Going forward, we'll use operation counts as our cost model, though looking at caching effects of data structures would make for an awesome final project!

The Story So Far

- We’ve just built a simple, elegant, balanced multiway tree structure.
- We can use them as balanced trees in main memory (2-3-4 trees).
- We can use them to store huge quantities of information on disk (B-trees).
- We’ve seen that different cost models are appropriate in different situations.

Time-Out for Announcements!

CS Townhall

- John Mitchell (CS Department Chair) and Mehran Sahami (CS Associate Chair for Education) are holding a CS townhall event next month.
- What are we doing well? What can we improve on? This is your chance to provide input!
- Held in Gates 104 from 4:30PM – 6:00PM on Tuesday, May 14th.

Problem Sets

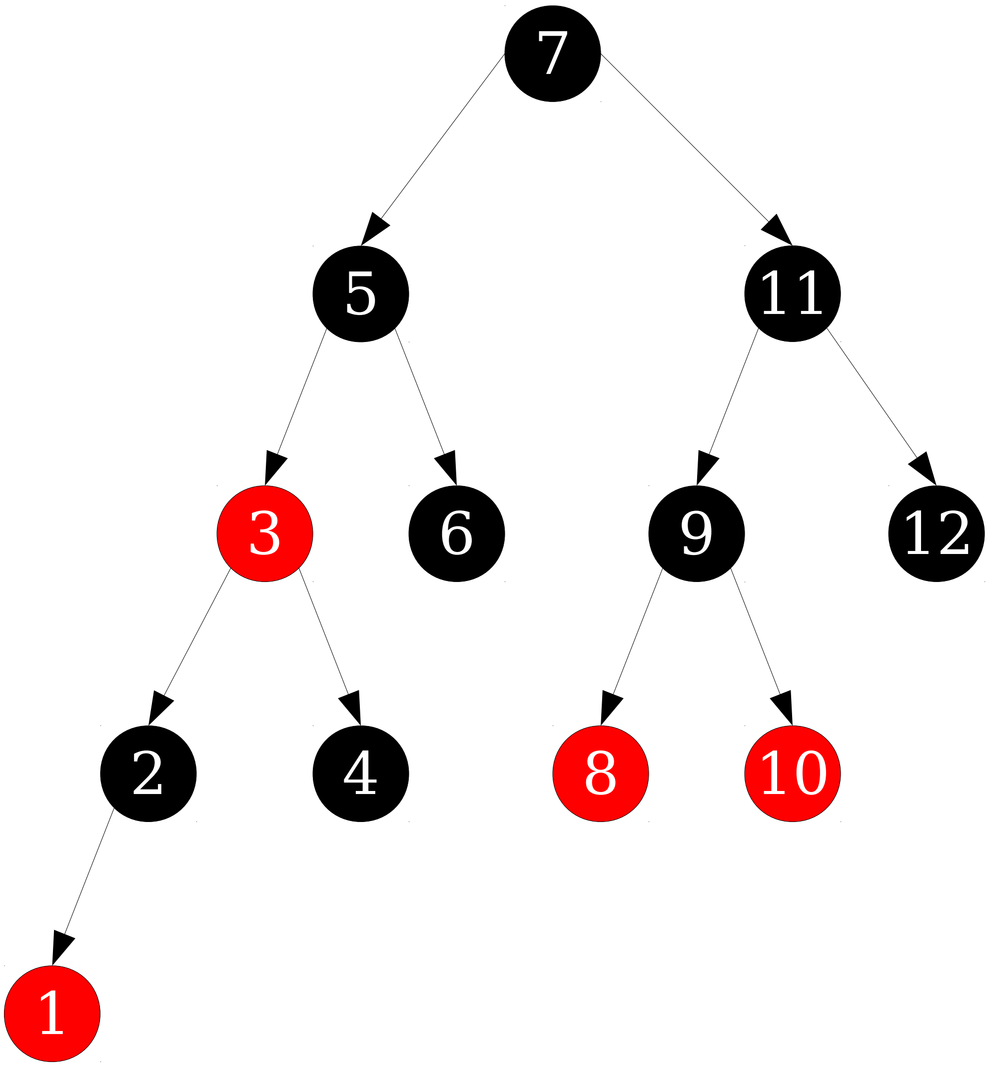
- Problem Set One solutions are now available up on the course website.
  - We’re working on getting them graded – stay tuned!
- Problem Set Two is due next Thursday.
  - Have questions? Ask them on Piazza or stop by our office hours!

Back to CS166!

So... red/black trees?

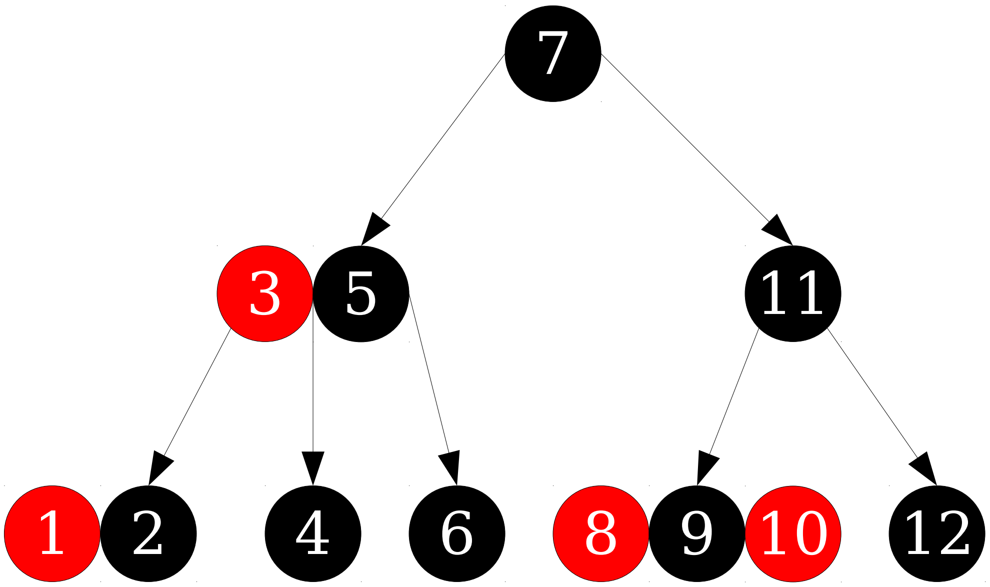
Red/Black Trees

- 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.
- After we hoist red nodes into their parents:
  - Each “meta node” has 1, 2, or 3 keys in it. (No red node has a red child.)
  - Each “meta node” is either a leaf or has one more key than node. (Root-null path property.)
  - Each “meta leaf” is at the same depth. (Root-null path property.)



This is a 2-3-4 tree!

Data Structure Isometries

- Red/black trees are an ***isometry*** of 2-3-4 trees; they represent the structure of 2-3-4 trees in a different way.
- Many data structures can be designed and analyzed in the same way.
- ***Huge advantage:*** Rather than memorizing a complex list of red/black tree rules, just think about what the equivalent operation on the corresponding 2-3-4 tree would be and simulate it with BST operations.

The Height of a Red/Black Tree

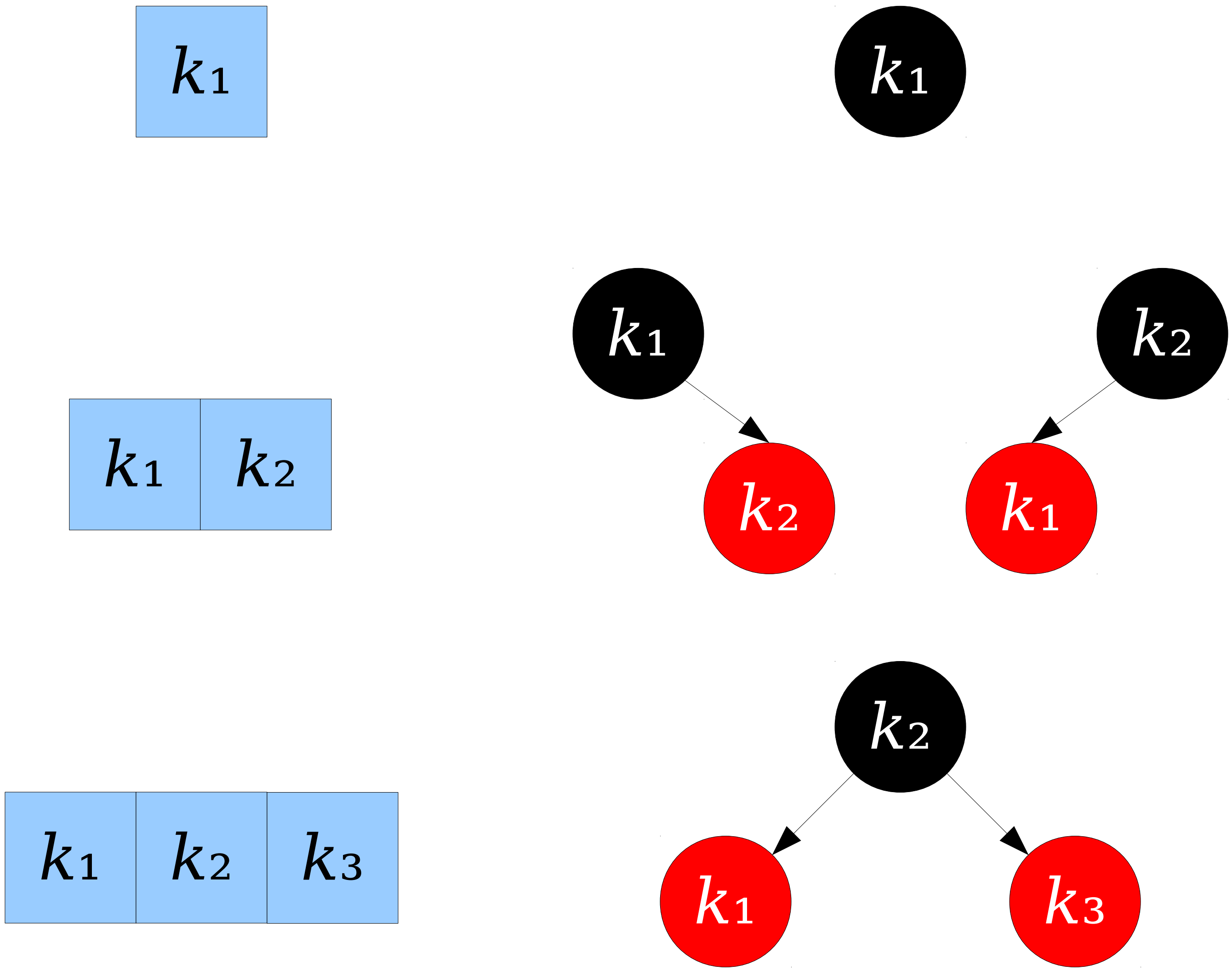
Theorem: Any red/black tree with n nodes has height $O(\log n)$.

Proof: Contract all red nodes into their parent nodes to convert the red/black tree into a 2-3-4 tree. This decreases the height of the tree by at most a factor of two. The resulting 2-3-4 tree has height $O(\log n)$, so the original red/black tree has height $2 \cdot O(\log n) = O(\log n)$. ■

Exploring the Isometry

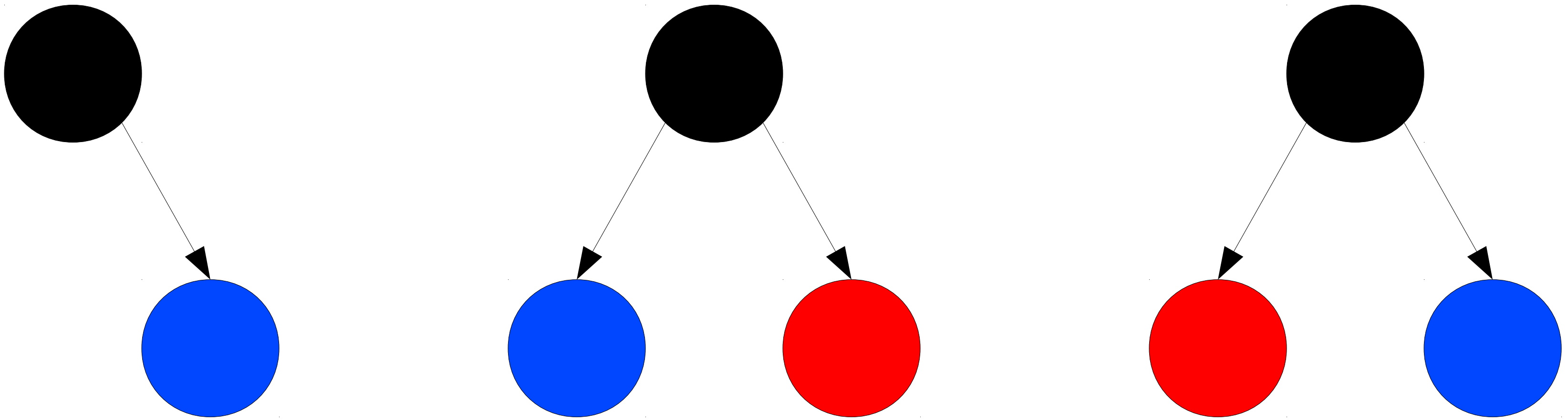
- Nodes in a 2-3-4 tree are classified into types based on the number of children they can have.
  - ***2-nodes*** have one key (two children).
  - ***3-nodes*** have two keys (three children).
  - ***4-nodes*** have three keys (four children).
- How might these nodes be represented?

Exploring the Isometry

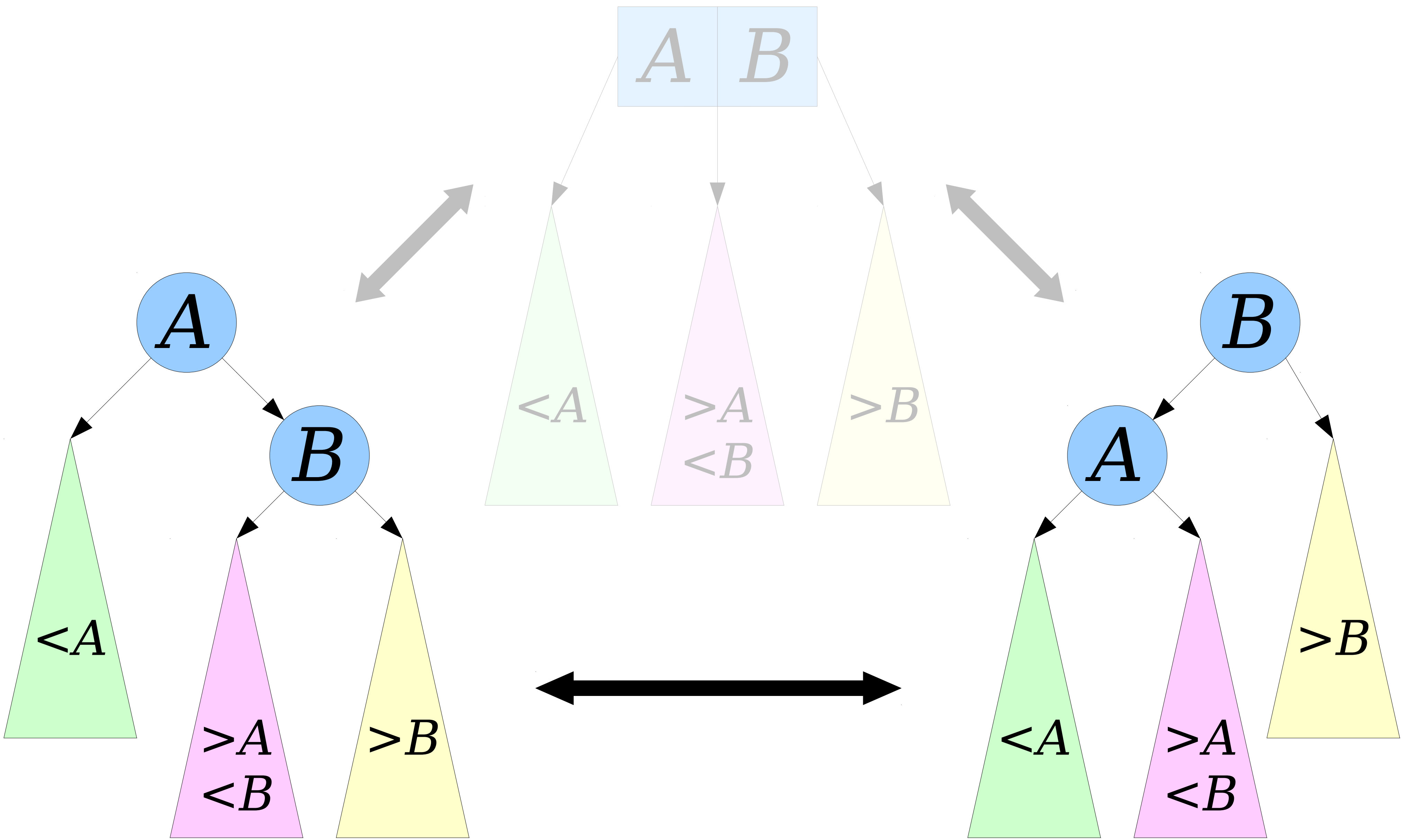


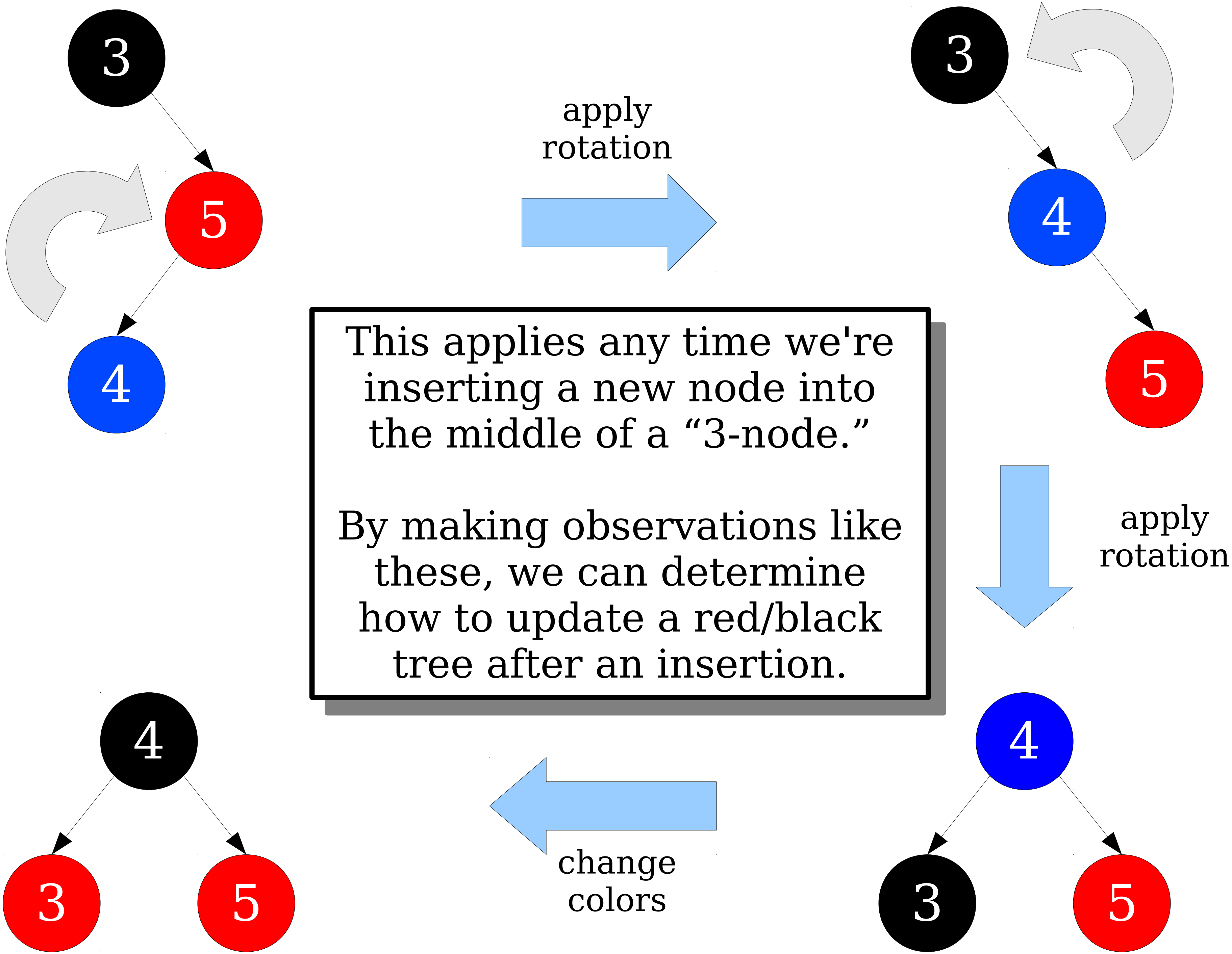
Red/Black Tree Insertion

- ***Rule #1:*** When inserting a node, if its parent is black, make the node red and stop.
- ***Justification:*** This simulates inserting a key into an existing 2-node or 3-node.



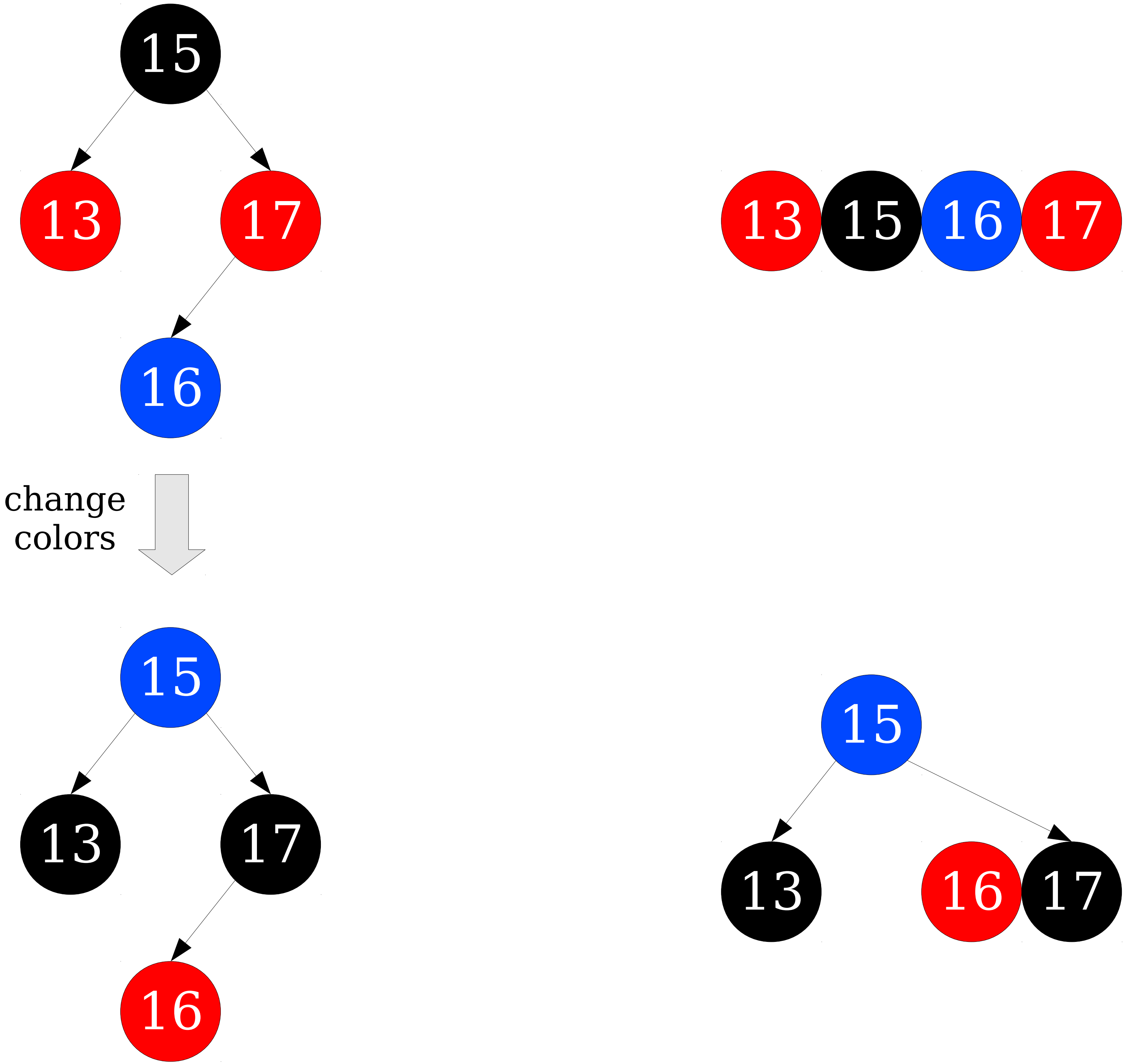
Tree Rotations





This applies any time we're inserting a new node into the middle of a "3-node."

By making observations like these, we can determine how to update a red/black tree after an insertion.


15
13
17
16
change
colors
15
13
17
16
13
15
16
17
15
13
16
17

Building Up Rules

- All of the crazy insertion rules on red/black trees make perfect sense if you connect it back to 2-3-4 trees.
- There are lots of cases to consider because there are many different ways you can insert into a red/black tree.
- ***Main point:*** Simulating the insertion of a key into a node takes time O(1) in all cases. Therefore, since 2-3-4 trees support O(log n) insertions, red/black trees support O(log n) insertions.
- The same is true of deletions.

My Advice

- ***Do*** know how to do B-tree insertions and searches.
  - You can derive these easily if you remember to split nodes.
- ***Do*** remember the rules for red/black trees and B-trees.
  - These are useful for proving bounds and deriving results.
- ***Do*** remember the isometry between red/black trees and 2-3-4 trees.
  - Gives immediate intuition for all the red/black tree operations.
- ***Don't*** memorize the red/black rotations and color flips.
  - This is rarely useful. If you're coding up a red/black tree, just flip open CLRS and translate the pseudocode. ☺

More to Explore

- The ***2-3 tree*** is another simple B-tree that's often used in place of 2-3-4 trees.
  - It gives rise to the ***AA-tree***, another balanced tree data structure.
- The ***left-leaning red/black tree*** is a simplification of red/black trees that forces 3-nodes to be encoded in only one way.
- ***Cache-oblivious data structures*** get the benefit of good cache performance, but without having advance knowledge of the cache size.
- B-tree variants are used in many contexts:
  - The ***B⁺-Tree*** are used in databases.
  - The ***R-tree*** is used for spatial indexing.
- These might be interesting topics to look into for a final project!

Next Time

- ***Augmented Trees***
  - Building data structures on top of balanced BSTs.
- ***Splitting and Joining Trees***
  - Two powerful operations on balanced trees.