Balanced Binary Search Trees

Instructors: Sam McCauley and Dan Barowy

April 22, 2022

Admin

- Sign up to be a TA! Deadline today
- Colloquium on crowdsourcing today (inventor of FoldIt)
- Any questions?

Course Registration

- Be sure to preregister for CS courses. (Not first come first serve, so don’t need to rush otherwise.)
- Some electives in CS only require 136. (All the electives next semester require at least one more course though—instructor permission possible, though probably rare)
- Options for next course:
    - Algorithms (256): explore theoretical side further. Much more involved proofs, algorithmic methods, running time analysis. Not focused on coding.
    - Computer Organization (237): explore how a computer works more. Learn C, learn much more about bit operations, how the computer works (how do methods get called? How is the call stack stored? What are the performance implications of this)?
    - Programming Languages (334): learn more about programming languages in general. You’ll learn more programming languages (some *very* different from any you’ve likely used in the past). You’ll also learn how things like types, methods, data structures, etc., can be abstracted from specific programming language implementations

Balanced Binary Search Trees

Binary Search Tree Analysis

- How much time does a call to `add()` take?
  - $O(n)$ in a tree of size n
  - $O(h)$ in a tree of height h

Improving Tree Height

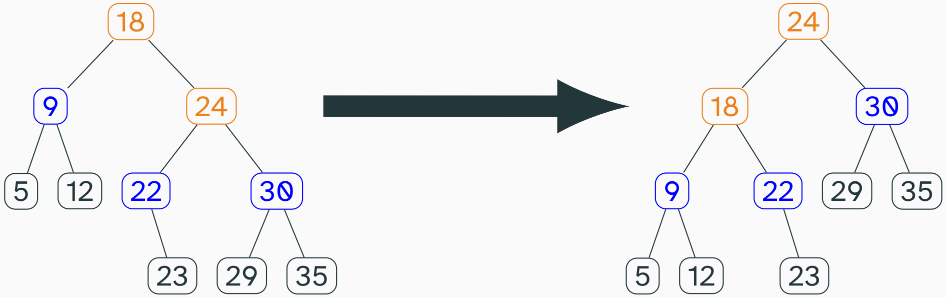
- What is the best height we can hope for in a binary tree?
- What kind of tree of height h has the most nodes?
  - A full/complete tree! (If the tree is not full there's room for more nodes.)
  - What is the height of a full/complete tree on n nodes?
  - (very close to) $\log n$
- Let's say we want to keep a complete tree. How long would `add()` take?
  - $O(n)$: would need to rebuild the whole tree every time
- Today: relax a bit to a tree of height $O(\log n)$; can implement `add()` in $O(\log n)$ as well
- Vocab for today: the *height* of a node is the longest path from the node to any descendant.

Tree Rotations

Updating Trees

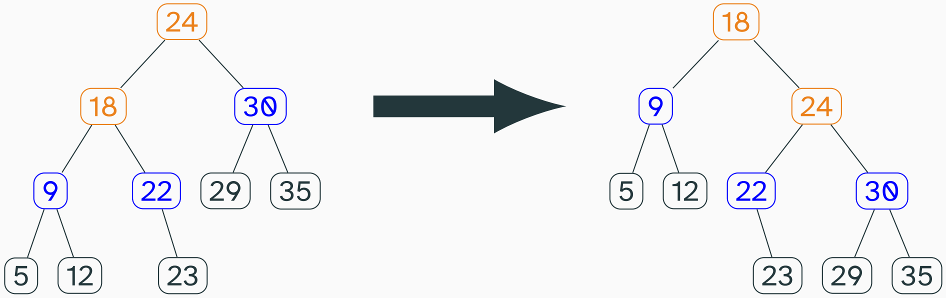
- If we’re going to keep balance, need a way to move items around the tree
- Crucial: need to *maintain the Binary Search Tree invariant* while we’re moving items
- Restructure tree while keeping BST ordering
- The building block of our rebalancing methods is a rotation

Tree Rotation: Rotate Left



This rotation is on the orange nodes (18 and 24); for a left rotation one must be a right child of the other. We rearrange the *children* of these nodes (in blue).

Tree Rotation: Rotate Right



This rotation is on the orange nodes (18 and 24); for a left rotation one must be a right child of the other. We rearrange the *children* of these nodes (in blue).

Implementing Tree Rotation

- Just change the child links
- Let's look at the code (`RedBlackTree.java` in `structure5`—you don't need to know this class outside of this method.)
- How long does this take?
  - $O(1)$ time!
- Goal: after we run `add()` (as we would in a BST), use tree rotations to ensure that the tree is balanced

AVL Trees

AVL Trees

- Invariant: for any node n in an AVL tree, the height of the left child of n must be at most 1 away from the height of the right child of n
- Store a number in each node representing its *balance* (height of right child $-$ height of left child) (so the invariant is that this is -1, 0, or 1)
- OK, so: how can we maintain this? If we do maintain this, what is the height of the tree? (Is that really enough for $O(\log n)$?)
- Note: you *do not need to memorize how all of this works*. You should know these rules exist, and be able to apply their logic in simpler scenarios

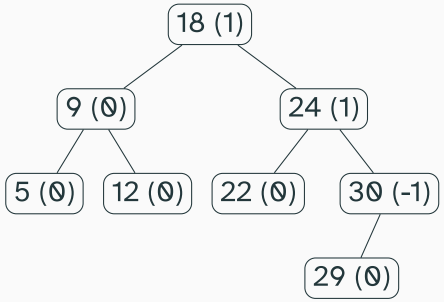
Maintaining AVL Invariant

- How can we insert a new item into an AVL tree?
- Insert proceeds like BST add
- Some nodes may now have balance $+2$ or -2
    - Note that only nodes *along the path to the newly-inserted node* may be out of balance
    - How can we quickly calculate the balance of all nodes along the path? (No time to scan through the whole tree.)
- Use rotations to fix these nodes

Recalculating node balances

- When does a node’s *height* change?
- The parent of the newly-inserted node’s height increases by 1 if the new node does not have a sibling
- In general: let’s say the height of a node increased after an insertion. Look at the parent. We can immediately update its balance. If its balance was previously 0, or if the current child was the larger-height child, its height also increased and we recurse. If the height doesn’t increase, no further node has a change of height
- Can update balance at the same time

Recalculating Node Balances Example



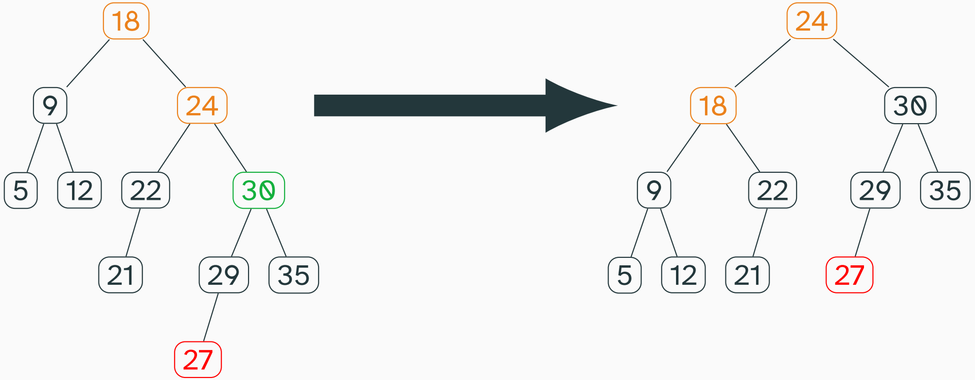
What happens if we insert 17? What about 24?

What is the running time of these updates? $O(h)$.

Maintaining AVL Invariant: Rotations

- Now, in $O(h)$ time can calculate balances of all nodes. Some are $+2$ or -2; we have to fix those
- Idea: two rotation rules can fix any node whose balance is $+2$ or -2
- We start at the new leaf; update nodes according to these rules as we walk towards the root

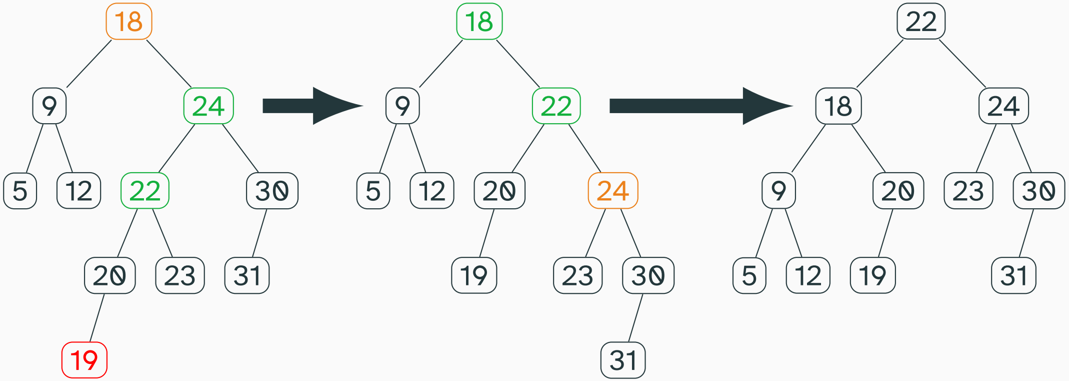
Rule 1



Rule 1: If the newly-added node is a right descendant of a right descendant of the unbalanced node, rotate the unbalanced node and its right child left

(Same idea: if left descendant of left descendant, rotate right.)

Rule 2 (Right-left case/Left-right case)



Rule 2: If new node is left descendant of right descendant of out-of-balance node, rotate bottom node right, then left

(Same idea if right descendant of left descendant)

Takeaway

- AVL trees: a few simple rules to maintain the invariant that each node has balance $-1, 0,$ or 1.
- Hard to memorize, but fairly easy to code!
- Now, the moment of truth: how does this affect performance?

AVL Tree Height

- All operations on an AVL tree (`add()` including rebalance, `contains()`, etc.) are $O(h)$ on a tree of height h
- What is the worst case for h on a tree of size n?
- Rephrasing: what is the fewest number of nodes that a tree of height h can have?
- To start: if $h = 0$, at least how many nodes must an AVL tree of height h have? What about if $h = 1$?
    - Just 1 if $h = 0$. (A tree of height 0 is just the root.).
    - 2 if $h = 1$. (Could have more, but has to have at least 2.)

AVL Tree Height

- Let $w(h)$ be the fewest nodes possible in an AVL tree of height h.
- How can we calculate this? We want to break this down into a smaller subproblem…
- Let's take a look at the root of the worst-case AVL tree of height h. At least one of its children must have height $h - 1$.
- What height must the other child have?
    - Height $h - 2$ by the AVL tree invariant
- So $w(h) = w(h - 1) + w(h - 2)$.
- And $w(0) = 1, w(1) = 2\ldots$
- The fewest number of nodes possible in a tree of height h is the $h + 2$nd Fibonacci number!

Bounding Fibonacci Numbers

- How big is $w(h)$? (This proof also works to bound the Fibonacci numbers)
- To show by strong induction: $w(h) \geq \left(\frac{3}{2}\right)^h$
- Base cases: for $h = 0$, $w(0) = 1 \geq (3/2)^0$; for $h = 1$, $w(h) = 2 \geq (3/2)^1$.
- Inductive hypothesis: assume that for some $n \geq 1$, for all $k = 0, \ldots, n$, $w(k) \geq (3/2)^k$.
- Inductive step: for any $n \geq 1$, $w(n+1) = w(n) + w(n-1)$. By the I.H., this is at least

$$\begin{aligned} w(n+1) &\geq (3/2)^n + (3/2)^{n-1} \\ &= (3/2)^n + (2/3)(3/2)^n \\ &\geq (3/2)^n + \frac{1}{2}(3/2)^n \\ &= (3/2)^{n+1} \end{aligned}$$

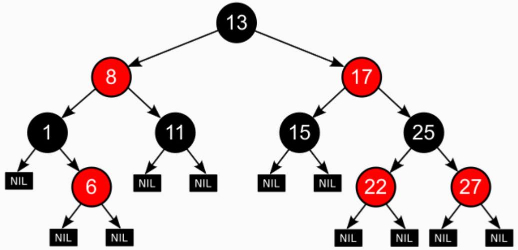
Putting it all together

- The lowest number of nodes in an AVL tree of height h is $w(h) \geq (3/2)^h$
- Therefore, if an AVL tree has n nodes, then the height of the tree must satisfy $n \geq (3/2)^h$, so $h \leq \log_{3/2} n$
- Therefore, $h = O(\log n)$! And we're done
- As you may have seen elsewhere, if $\phi = (1 + \sqrt{5})/2 \approx 1.62$, then $\log_\phi n$ is a much tighter bound.

Wrapping it Up

- AVL trees support `add()`, `contains()`, `remove()` (we didn't talk about remove; same idea but more complicated rules) all in $O(\log n)$ time
- Every other data structure we've seen requires at least $O(n)$ time!
- So on a data structure with a billion items, requires ≈ 30 operations rather than ≈ 1000000000.
- Incredible example of:
  - How more intricate data structures can improve performance (past what seemed possible)
  - How simple invariants can lead to performance improvements
  - How more-involved analysis can help us analyze complex data structures

Red-black trees



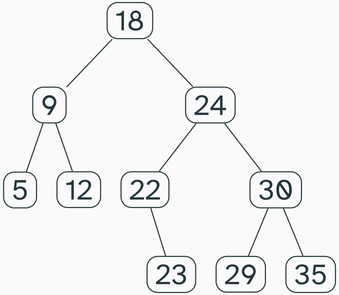
- Another way to implement a Balanced Binary Search Tree
- Some advantages; some disadvantages in practice compared to the AVL tree
- Also get $O(\log n)$-time operations

Other BST Operations

Finding Predecessor and Successor Items

- Binary search trees are more powerful than just fast `contains()` queries
- What if we want to search for the largest item under some bound? (This is essentially what we did in the two towers lab.)
- `Predecessor query:` given a query q, what is the largest item in the data structure that is $\leq q$?
- `Successor query:` given a query q, what is the largest item in the data structure that is $\geq q$?
- Generalizes `contains()`; very useful operations

Returning Items in a Range



- Given a and b, can we find all items between a and b in a binary search tree?
- Yes; if there are k items takes $O(k+h)$ time!
- Basic idea: find a and b; do a careful traversal between them
- Extremely common: “get me all students with names in this range” or “find all dates in this range” and so on