Lec 25: Balanced Binary Search Trees

April 24, 2026

Admin

- Apply to be a TA!!
    - Deadline today
- Lab out today (open-ended; start early!)
- Midterm back next day or so (really hoping for today)
- Any questions?

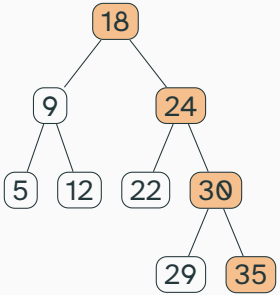
Next Classes

- Colloquium today with specifics
- Core classes:
  - CS 237 (Organization): programming; pointers; bit tricks; how a computer works
  - CS 256 (Algorithms): theory/proofs; recursion; important algorithms
  - CS 270 (AI): Probability; regression; search; basics of AI

Next Classes

- Keep an eye out for electives without core prereqs!
    - CS 334 (Programming Languages): Compare programming languages; learn how they work
    - CS 315 (Computational Biology) (Spring): Applying CS to biological applications

Review from Last Time



- `add()` and `contains()` are $O(h)$
- But h can be as bad as n!

Improving Binary Search Trees

- How *small* can the height of the tree be?
- How many nodes can there be in a tree of height h?
- There can be at most 1 node at depth 0, 2 at depth 1, 4 at depth 2; in general, 2^k nodes at depth k
- Max nodes for height h is (we'll briefly prove this equation on the board):

$$1 + 2 + 4 + 8 + \ldots + 2^h = 2^{h+1} - 1.$$

- At most $2^{h+1} - 1$ nodes in a tree of height h. So if we want the tree to be large enough to handle n nodes, need $2^{h+1} - 1 \geq n$, so $h \geq \log_2(n+1) - 1$.

Improving Binary Search Trees

- Conclusion: A binary search tree with n nodes can have height as much as n if it's terribly balanced, or as small as $\log_2(n+1) - 1$ if it's perfectly balanced

- Today: we'll talk about a way to maintain that the tree always has height $O(\log n)$

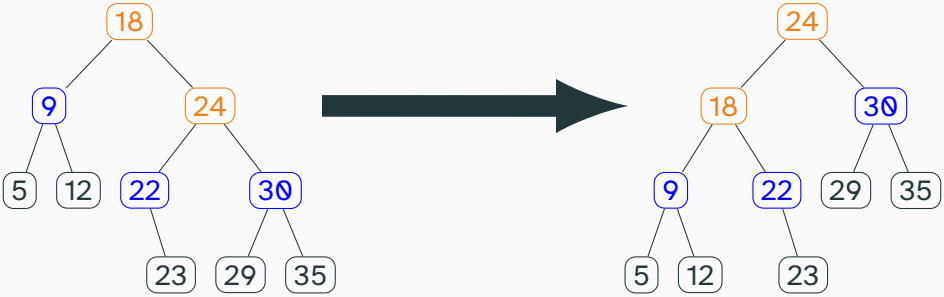
- Then: `add()` and `contains()` are $O(\log n)$ time!

Tree Rotations

Updating Trees

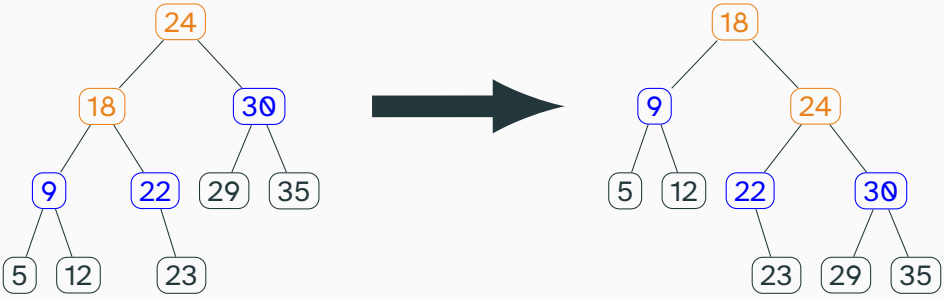
- If we're going to keep balance, need a way to move items around the tree
- Crucial: need to *maintain the Binary Search Tree invariant* while we're moving items
- Restructure tree while keeping BST ordering
- The building block of our rebalancing methods is a rotation
- A rotation is kind of like a "swap." Let's work out how this could work on the board.

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 Java library code that does tree rotations
- How long does a rotation 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"—has height $O(\log n)$

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

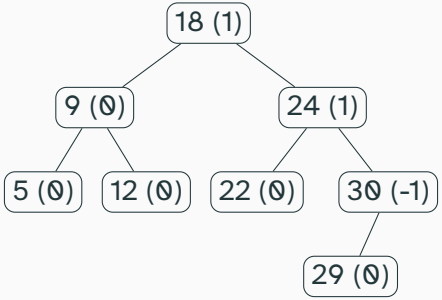
Maintaining AVL Invariant: Plan for Next Few Slides

- How can we insert a new item into an AVL tree?
- Insert proceeds like BST add
- Some nodes may have had their balance changed
  - Note that only nodes *along the path to the newly-inserted node* change their balance
  - Goal 1: how can we quickly calculate the balance of all nodes along the path? (No time to scan through the whole tree.)
  - Goal 2: Use rotations to ensure that these nodes have balance -1, 0, or 1

Recalculating node balances

- When does a node's *height* change? When does its *balance* change? We'll update both at the same time!
- Start at the bottom: The parent of the newly-inserted node's height increases by 1 if the new node does not have a sibling. The balance always changes.
- In general: let's say the height of a node increased after an insertion. Look at the parent. We can immediately update its balance!
- Now that we've updated the balance of the parent, how can we update the height of *its* parent?
  - 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 (or balance)

Recalculating Node Balances Example



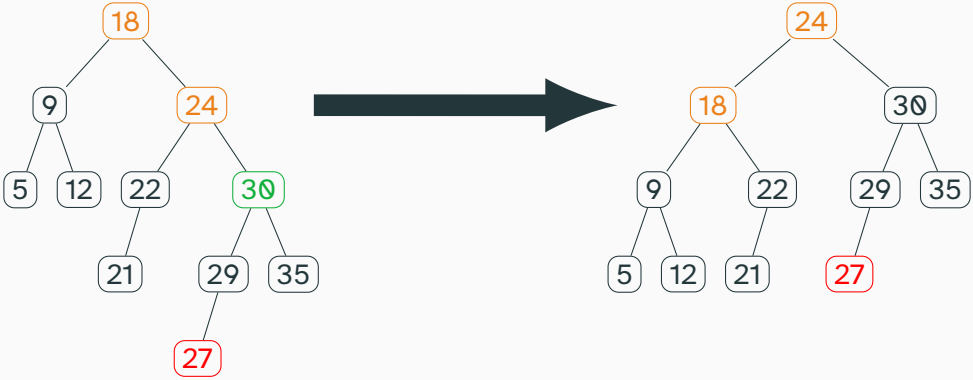
What happens if we insert 17? What about 24?

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

Maintaining AVL Invariant: Rotations

- Now, in $O(h)$ time have updated 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

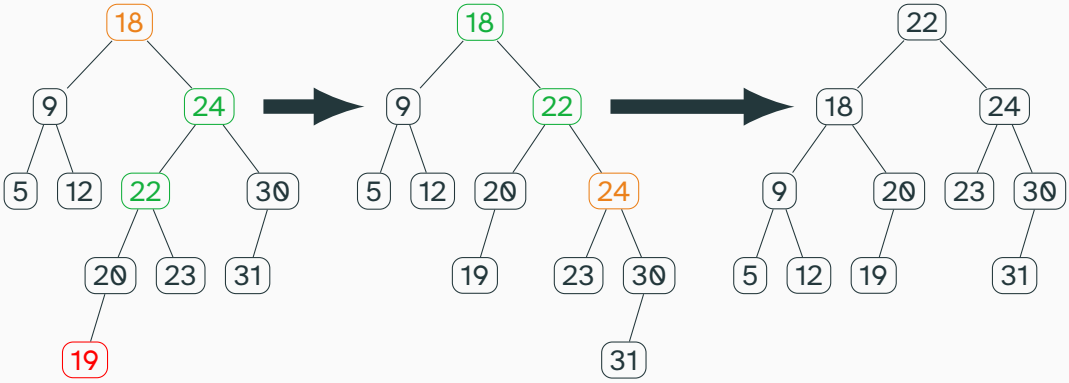
Rule 1 (For Intuition; Don’t Need to Memorize)



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 (For Intuition; Don’t Need to Memorize)



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 an AVL 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.).
  - Just 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? Let's work this out on the board
- 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 $w(0) = 1$, $w(1) = 2$...
- The fewest number of nodes possible in a tree of height h is the $h + 2$nd Fibonacci number!

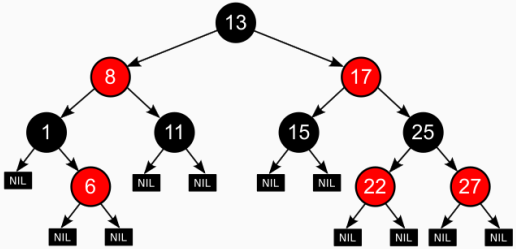
Putting it all together

- The lowest number of nodes in an AVL tree of height h is $w(h) = F_{h+2} \approx \phi^{h+2} \approx 1.61^{h+2}$
  - (Classic bound on nth Fibonacci number $F_n \approx 1.61^n$)
- Therefore, if an AVL tree has n nodes, then the height of the tree must satisfy $h + 2 \leq \log_\phi n$
- Note that $O(\log_\phi n) = O\left(\frac{\log_2 n}{\log_2 \phi}\right) = O(\log n)$. Therefore, $h = O(\log n)$! And we're done

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 $\approx$ 30 operations rather than $\approx$ 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



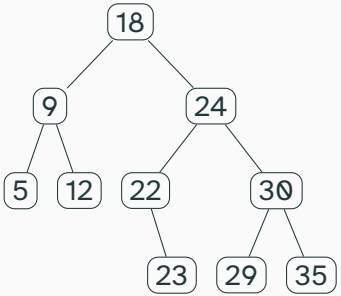
- Another way to implement a Balanced Binary Search Tree
- Some advantages; some disadvantages in practice compared to the AVL tree
- Also get $O(\log n)$-time operations
- The Java library uses Red-black trees; not AVL trees

Other BST Operations

Finding Predecessor and Successor Items

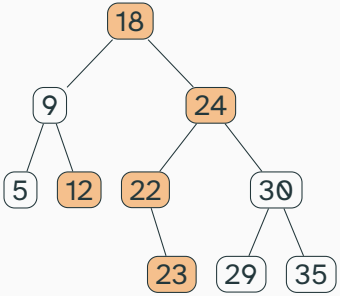
- Binary search trees are more powerful than just fast `contains()` queries
- What if we want to search for the largest item under some bound? (This is essentially what we did in the two towers lab.)
- `Predecessor query:` given a query q, what is the largest item in the data structure that is $\leq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

Returning Items in a Range: Example



Range query: $[12, 24]$ ■ = in range

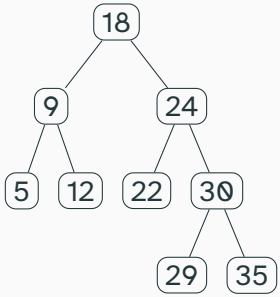
- Given a and b, can we find all items between a and b in a binary search tree?
- Yes; if there are k items takes $O(k + h)$ time!
- Basic idea: find a and b; do a careful traversal between them
- Extremely common: “get me all students with names in this range” or “find all dates in this range” and so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

Binary Tree Practice

- How can we calculate the size of a binary tree?
    - Hint: use recursion!
- How can we calculate the height of a binary tree?
    - Recursion again!

Iterating Over Trees

Goal



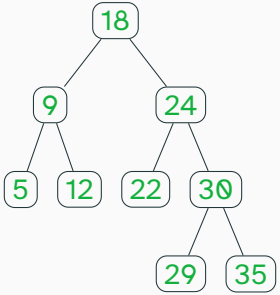
- Let's say I want to iterate through each of the items in my tree, one at a time
- In what order should I go through the nodes?
  - We say that we *traverse* the tree
- We'll see three different methods of traversing a tree
- Any ideas?

Post-order traversal

- *Post-order traversal*: First, we recursively traverse the left child of the root. Then, we recursively traverse its right child. Finally, we visit the root.

- Let’s see an example

Post-order Traversal

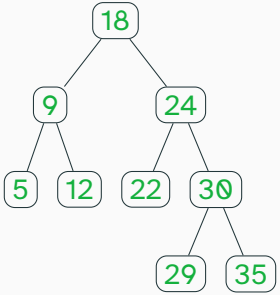


Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

In-order traversal

- *In-order traversal*: First, we recursively traverse the left child of the root. Then, we visit the root. Then, we recursively traverse its right child.
- Visually: in-order scans the tree from left to right.
    - This is just a mnemonic! The tree traversal depends on its edges, not the way it's drawn.
- Let's see an example

In-order Traversal

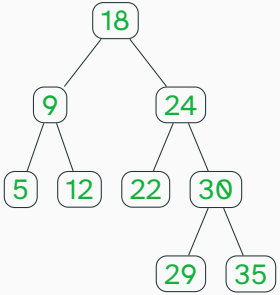


Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Level-order traversal

- *Level-order traversal*: We visit all nodes at the same depth from left to right
- Unlike the other traversals, doesn't recursively order the children vs the root
- Let's see an example

Level-order Traversal



Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.