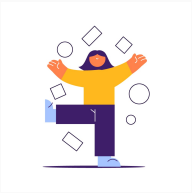
Lec 26: Tree Traversals

April 27, 2026

Admin



- Start early on the lab
- Midterm 2 discussion next slide
- Any questions?

Midterm 2

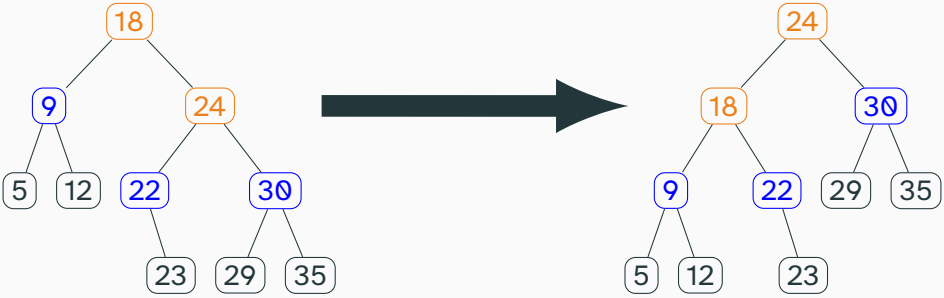
- Grades relatively high: median 87; average 83.5
  - Bear in mind that quiz and lab grades are very high
  - Your best midterm/final grade is worth 5% more and your lowest is worth 5% less
- TAs commented that some students felt surprised by the midterm content/that the practice midterm hadn't prepared them
- I think going over the midterm questions—and specifically, how they relate to exam questions—is a great office hours topic

Study Tips Looking Ahead

- If you get help on a topic/lab/question/project (especially if you get a lot of help), do it yourself later without help
  - The best way to learn is by doing something yourself, and making your own mistakes
- Other readings: Duane's textbook; Dan's 136 readings from last semester
  - Duane's textbook also has practice problems
  - Note that Dan's writeups use C# instead of Java. The languages are similar, but details are different
- In theory you spend $\approx$ 10 hours on studying/assignments for each class outside of class hours. If you're much less than that, and want to improve your grade, grinding out more hours can help
  - More familiarity with things you've seen does help with things you haven't seen
- As usual: office hours and TA hours are pretty empty; they're there to help you.

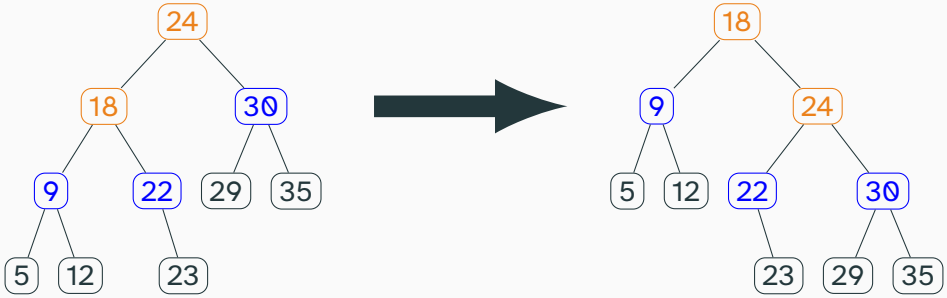
Quick AVL Tree Review

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).

AVL Trees

- Invented by Georgy Adelson-Velsky and Evgenii Landis in 1962
- 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)
- Today: how can we maintain this? If we do maintain this, what is the height of the tree?
- 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

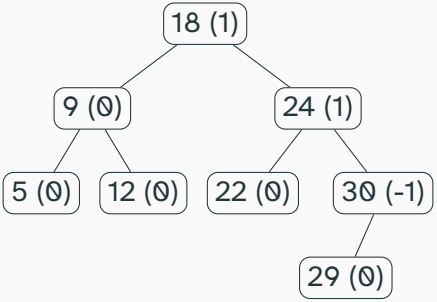
Maintaining AVL Invariant: Plan for Next Few Slides

- 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



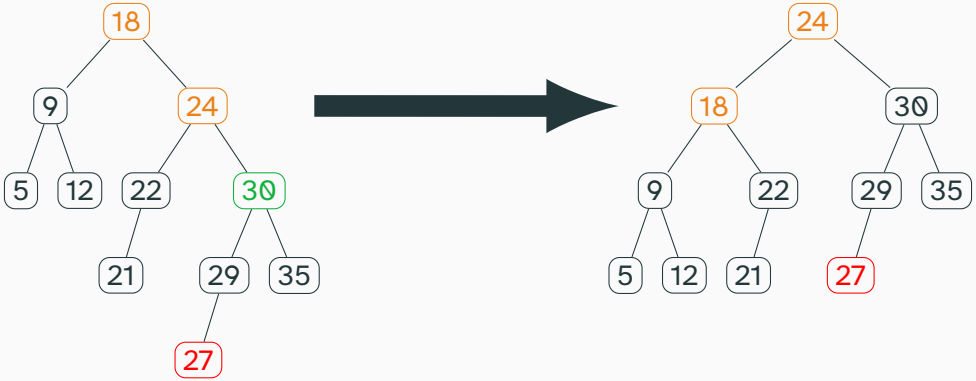
What happens if we insert 17? What about 24?

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

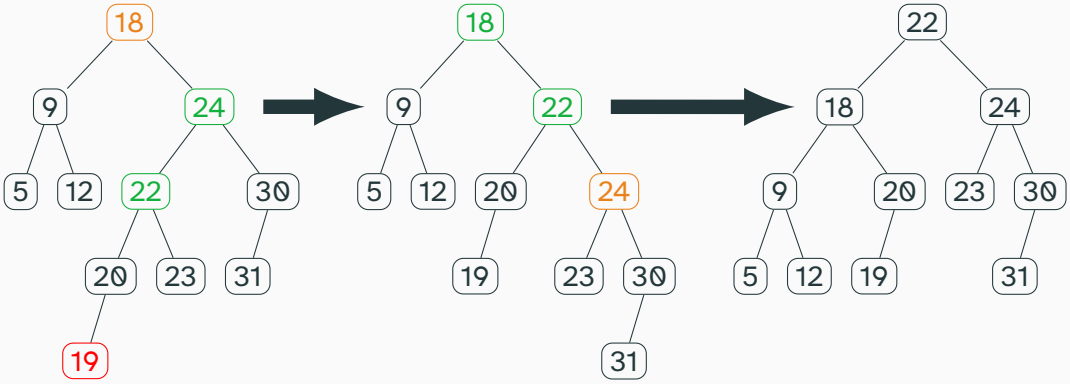
Rule 1 (For Intuition; Don’t Need to Memorize)



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? (emphHint: what do we know about an AVL tree?)
- 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!

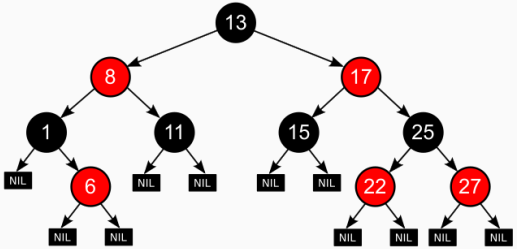
Putting it all together

- 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 $h + 2 \leq \log_{1.61} n$
- Note that $O(\log_{1.61} n) = O\left(\frac{\log_2 n}{\log_2 1.61}\right) = O(\log_2 n)$. 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



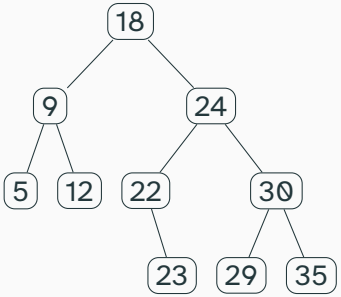
- Another way to implement a Balanced Binary Search Tree
- Some advantages; some disadvantages in practice compared to the AVL tree
- Also get $O(\log n)$-time operations
- The Java library uses Red-black trees; not AVL trees

Other BST Operations

Finding Predecessor and Successor Items

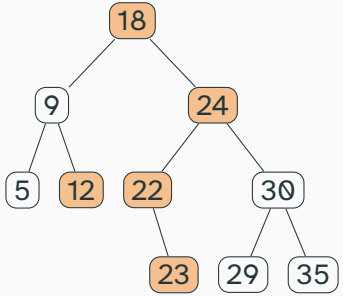
- Binary search trees are more powerful than just fast `contains()` queries
- What if we want to search for the largest item under some bound? (This is essentially what we did in the two towers lab.)
- `Predecessor query:` given a query q, what is the largest item in the data structure that is $\leq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

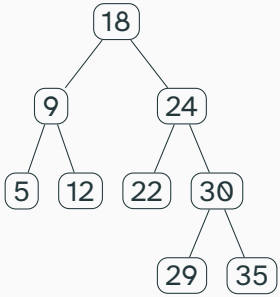
- Given a and b, can we find all items between a and b in a binary search tree?
- Yes; if there are k items takes $O(k + h)$ time!
- Basic idea: find a and b; do a careful traversal between them
- Extremely common: “get me all students with names in this range” or “find all dates in this range” and so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



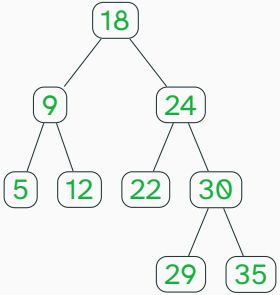
- Let's say I want to iterate through each of the items in my tree, one at a time
- In what order should I go through the nodes?
  - We say that we *traverse* the tree
- We'll see three different methods of traversing a tree
- Any ideas?

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

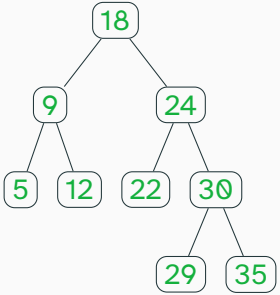


Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

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



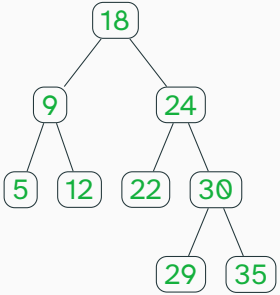
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Pre-order traversal

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

- Let’s see an example

Pre-order Traversal

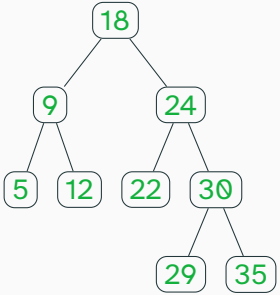


Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

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



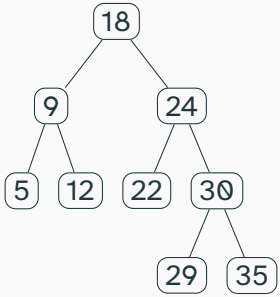
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Implementing Tree Iterators

Implementing Tree Iterators

- Goal: implement the traversals we mentioned as an iterator
- Can do `next()` and `hasNext()` on demand
- Problem: want to get values on demand (should be updated as the tree is updated)
  - Don't want to traverse the tree, store all tree values, and then dispense them one by one (you may do that in the lab however)
  - Instead: each call to `next()` should go to the next node in the tree we want to output
- Challenge: implementing a recursive traversal piece-by-piece
- To think about: what data structure helps with recursion?

Pre-order traversal

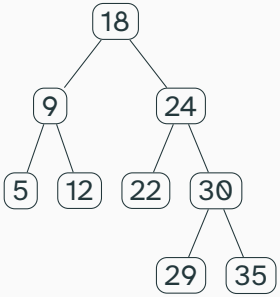


- Visits the node, then recursively traverses the left child, then the right child
- Keep track of the current node we're traversing
- What happens when we hit a leaf?
- Could backtrack by following pointers; might get confusing
- Instead: maintain nodes to visit on a stack!

Pre-order traversal

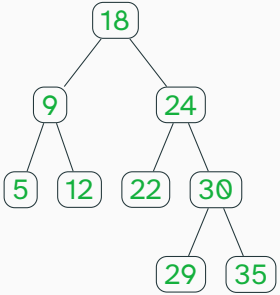
- Stack maintains the non-empty `BinaryTree<E>` objects that we *still need to traverse*
- So `next()`:
  - pops the top item off the stack
  - Stores its value to be returned
  - Pushes its right child onto the stack if nonempty
  - Pushes its left child onto the stack if nonempty
- `hasNext()`?
  - Just returns if the stack is empty

Post-order and In-order traversal



- We won’t go over in this class. In short: a stack works, but you need to use it a little differently.

Level-order Traversal



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

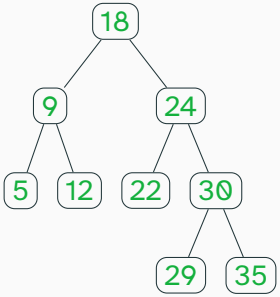
Level-order traversal

- Level-order traversal is not recursive!
- How do we keep track of what nodes to visit next?
- Key insight: the order we visit nodes at a given “level” is the same order we visited their parents
- So the *first* parents to be visited have the *first* children that are visited
- …Can we use a queue?

Level-order iterator

- To begin: push root onto the queue
- `next()`:
  - Dequeue node off the queue; store its value to be returned
  - Enqueue its non-empty children onto the queue
- `hasNext()`: return if queue is empty
- Let’s look at the code

Level-order Traversal



Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the *values* of the nodes, but in reality the objects stored are of type `BinaryTree`

Queue: 9 24 Queue: 24 5 12 Queue: 5 12 22 30 Queue: 12 22 30 Queue: 22 30 Queue: 30 Queue: 29 35 Queue: 35 Queue: Queue:

Priority Queues

Priority Queue: Goal

- Want to return the *minimum* item in the data structure efficiently
- Two operations:
  - `add()`: adds a new elemenet to the data structure
  - `deleteMin()`: find and return the smallest element in the data structure
- Oftentimes want a third operation:
  - `decreaseKey()`: make an item in the data structure smaller
- How can we do this?