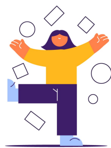
Lec 24: Balanced Binary Search Trees

April 22, 2026

Admin



- Apply to be a TA!!
  - Deadline April 24
- In-person lab today; be sure to come
  - Topic: iterators (we’ll do trees next week)
- Any questions?

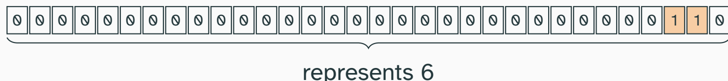
Bit Operations

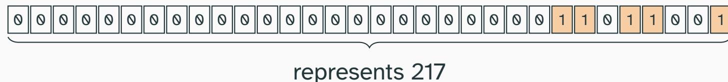
Bit Operations

- We’ll use on lab
- Good thing to have seen in general/useful tool
- Let’s discuss briefly, then we’ll get back to trees
- Basic idea: math tools that work well with binary numbers

Storing Numbers in Java

- An `int` is stored in binary, using 32 bits
- A `long` uses 64 bits
- The first bit helps us indicate if a number is negative or not. (It does *not* quite represent a "negative sign." We will only do bit operations on positive numbers in this class.)





Shift operations

- We can use `<<` to shift the digits of a number to the left, filling in `0`s.
- Example: `6 << 2` takes the binary representation of 6 and shifts it 2 digits to the left

9 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | $\xrightarrow{<<2}$ | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | $= 36$

Shift operations

- We can use `<<` to shift the digits of a number to the left, filling in `0`s.
- The `>>` operator shifts to the right. (Be careful with negative numbers.)
- Question: How can we rewrite `x << y` mathematically? (Hint: think of how it would work for decimal numbers.)
    - `x << y` multiplies x by 2, y times. So it is the same as $x \cdot 2^y$.
- Question: How can we rewrite `x >> y` mathematically? (Hint: think of how it would work for decimal numbers.)
    - `x >> y` divides x by 2, y times, rounding down each time. So it is the same as $x/2^y$ using integer division.
- Use parentheses with these!

Bitwise and, bitwise or

- The & operator (with only one &) works on two numbers. It goes *bit-by-bit* through both numbers; if both have a 1, the result has a 1; otherwise, the result has a 0

6	0	0	0	0	0	1	1	0
&12	0	0	0	0	1	1	0	0
= 4	0	0	0	0	0	1	0	0

- The | operator (with only one |) works on two numbers. It goes *bit-by-bit* through both numbers; if *either or both* have a 1, the result has a 1; if both have a 0, the result has a 0

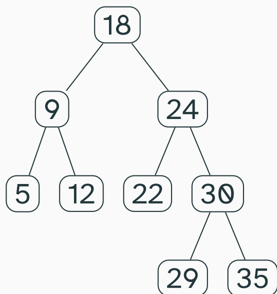
6	0	0	0	0	0	1	1	0
|12	0	0	0	0	1	1	0	0
= 14	0	0	0	0	1	1	1	0

Shift Practice

- In pairs: how can we create an `int` that has a 1 in the 10th and 20th slot (from the right), and 0s in all other slots?
  - Solution 1: `(1 << 10) | (1 << 20)`
  - Solution 2: `((1 << 10) + 1) << 10`
  - Solution 3: `(1 << 10) + (1 << 20)`
- In pairs: how can we test if there is a 1 in the 24th bit of `int x`?
  - Solution 1: `if(x & (1 << 24) != 0)`
  - Solution 2: `if((x >> 24) & 1 != 0)`
  - Solution 3: `if((x & (1 << 24) == (1 << 24))`

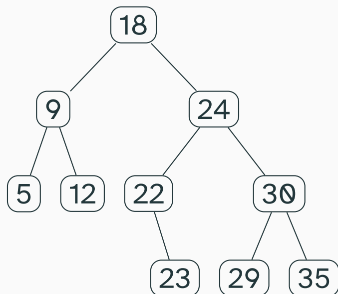
Implementing a Binary Search Tree

Finding an element in a binary search tree



- Idea: we can look at a node and know immediately if the element we're searching for is a descendant of the left child, or of the right child
- Recurse on the appropriate node
- If we find the element, or if we hit an empty node, we're done
- Practice: how can we find the *smallest* element in a Binary Search Tree?

Adding an element to a binary search tree



- Idea: we can look at a node and know immediately if the element we're adding should be a descendant of the left child, or of the right child
- Recurse on the appropriate node
- If we reach `null`, instead add a new node with the element we want as data
- Practice: Let's say we add n elements one-by-one using this algorithm. Which element will be the root?

Natural Comparator

- Let's say we have an item of type `E` that implements `Comparable<E>`
- That means we can already compare items of type `E`
- But, we want the flexibility to compare them other ways using a `Comparator<E>`
- The `NaturalComparator<E>` implements `Comparator<E>`, and compares items using their `compareTo()` method
- That way, we can write code assuming we always have a comparator; if we want we can replace it with a different comparator
- Let's look at the code

Tree Vocabulary

- *Descendant*: A node n' is a descendant of node n if there exists a sequence of nodes $n = n_1, n_2, \ldots, n_i = n'$ such that for all $1 \leq j < i$, n_{j+1} is a child of n_j. (*Ancestor* is the opposite)
- *Path*: the unique shortest sequence of edges between two nodes n_1 and n_2. Each successive edge in the path must share one of its nodes with the previous edge
- The *depth* of a node is the number of edges on the path to the root
- The *height* of a tree is the maximum depth of any node

Binary Search Tree Analysis

- How much time does a call to `contains()` take?
  - Worst case
  - Definitely not worse than $O(n)$ (we never look at a node multiple times)
  - Is there a tree where it's actually $O(n)$? Yes; let's try to create an example on the board
- Let's say we have a tree of height h. How long does a call to `contains()` take in terms of h?
  - Each time we call the method the depth of the node increases by one, so $O(h)$
  - If we have time: how can we prove this by induction?

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
  - h can be as bad as n sometimes

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

- 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 $O(\log n)$

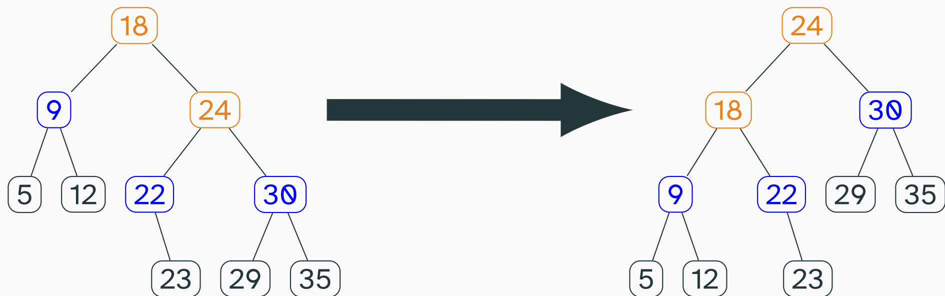
- Then: `add()` and `contains()` are always $O(\log n)$ time!

Tree Rotations

Updating Trees

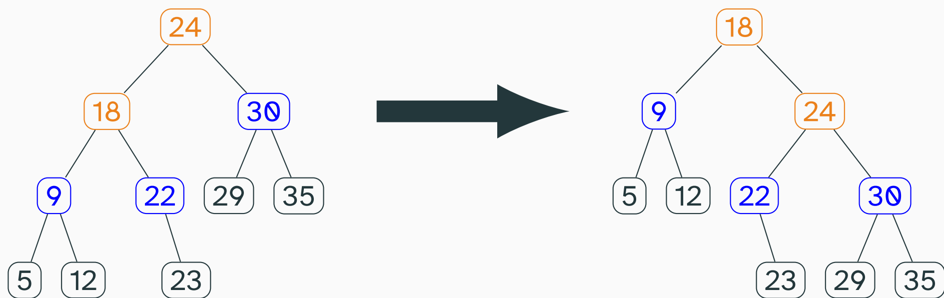
- If we're going to keep balance, need a way to move items around the tree
- Crucial: need to *maintain the Binary Search Tree invariant* while we're moving items
- Restructure tree while keeping BST ordering
- The building block of our rebalancing methods is a rotation

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