SPLAY TREES [Sleator, Tarjan 1985]

Splay trees are self-balancing search trees.
Search trees are widely used data structures that support:

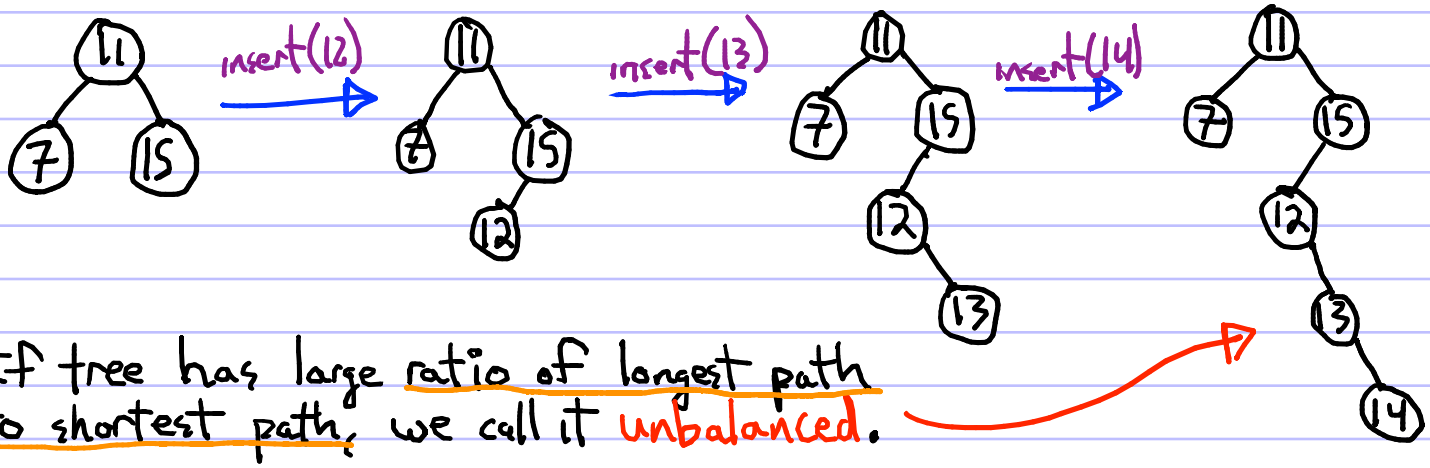
insert(object, int) object delete(int) object find(int)

Also commonly used as the basis of an augmented data structure for other types of queries,
e.g. range queries in geometry, predecessor/successor queries,
least common ancestor queries in string matching,
order statistics (median, min, max)
... a one-structure-fits-all data structure
for data that has a linear ordering and changes

dynamic = insert() + delete() quickly

what is a type of data without a linear ordering?

STATIC BINARY SEARCH TREES



If tree has large ratio of longest path to shortest path, we call it unbalanced.
Call this ratio the balancedness of the tree.

If balancedness is $O(1)$, find() takes $O(\log n)$ time.
For any tree, find() takes $\Omega(\log n)$ time. Why?
Goal: maintain $O(1)$ balancedness, regardless of insert(), delete(). (dynamicity)
Easy to build a $O(1)$ balanced BST without dynamicity.

Dynamic, for BSTs

SELF-BALANCING BINARY SEARCH TREES

AUGMENTATION
CACHE PERFORMANCE
PARALLELISM
SPACE USAGE

So many of these, some better in certain conditions.

SAME THING

- Red-Black Trees
- AVL Trees
- Scapegoat Trees
- 2-3 Trees
- ⋮
- Splay Trees

Why study splay trees? Simple, extra properties useful in practice, might be the theoretically best BST, small space, stable.

THEY ARE RAD

But like all radness, a cost: they don't actually stay O(1) balanced.

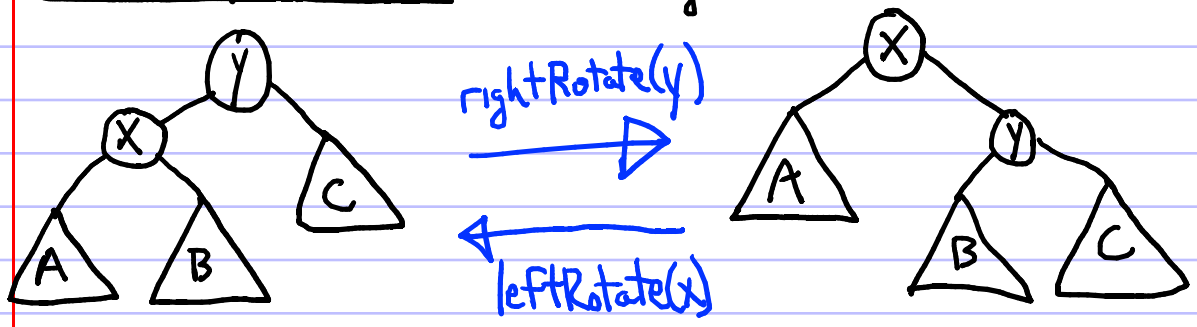
Worst case: $\Theta(n)$ balancedness. Ouch.

Worst case find(): $\Theta(n)$.

Amortized find(): $O(\log n)$.

THE TRICK: pay for trips down the tree by removing messiness. Longer trip = more messiness removed.

Tree Rotation a.k.a. Zig



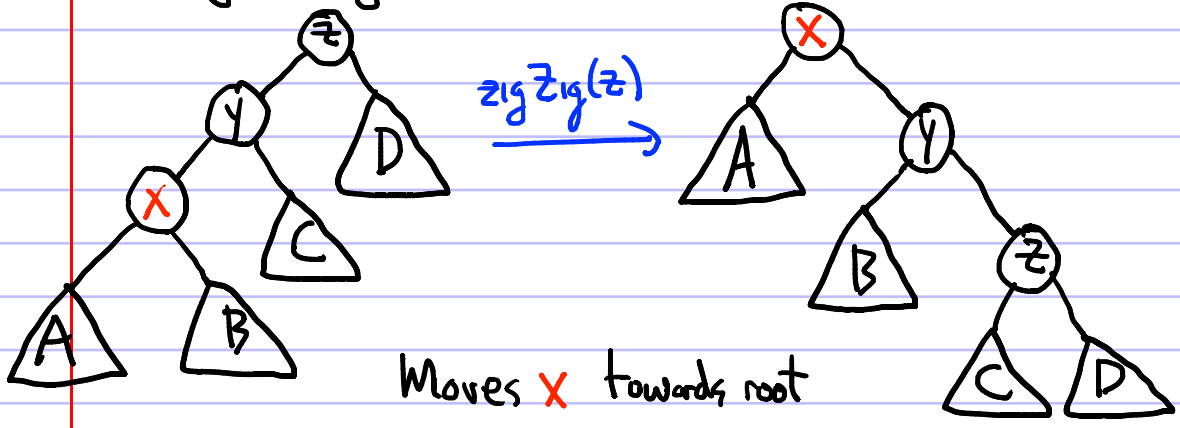
Everything in $A \le x \le$ Everything in $B \le y \le$ Everything in C

So rotations preserve binary-search-treeness.

Use rotations as a building block for insert(), delete(), find().

Complex rotations

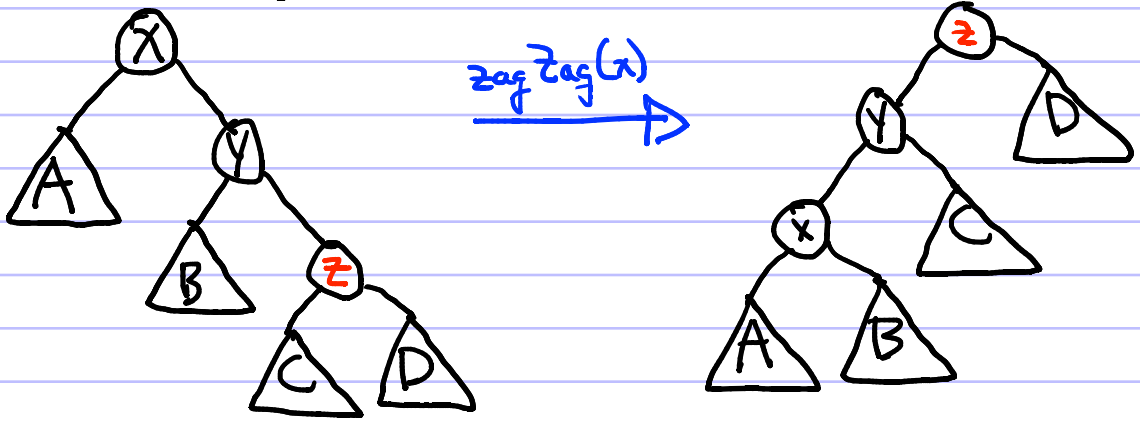
Zig-Zig



Moves x towards root

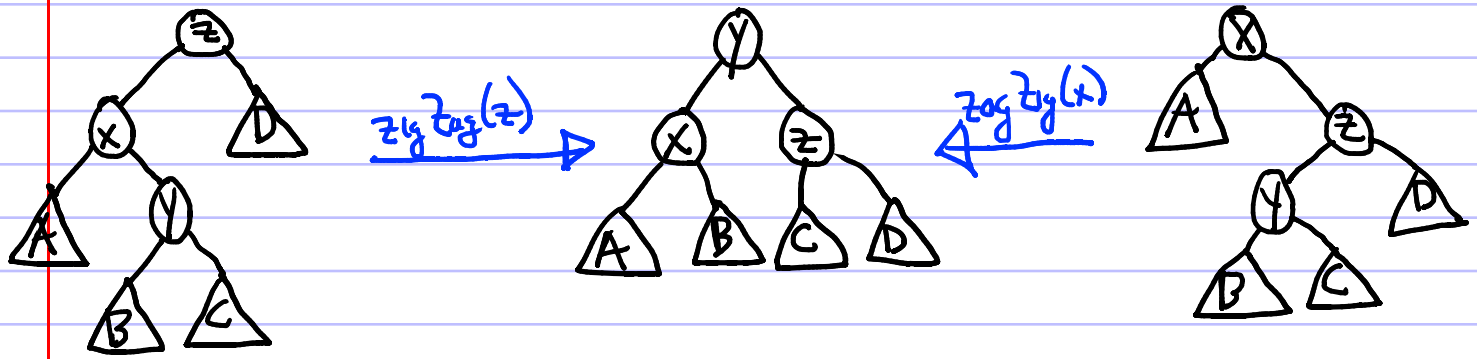
mirror operations

Zag-Zag



Moves z towards root

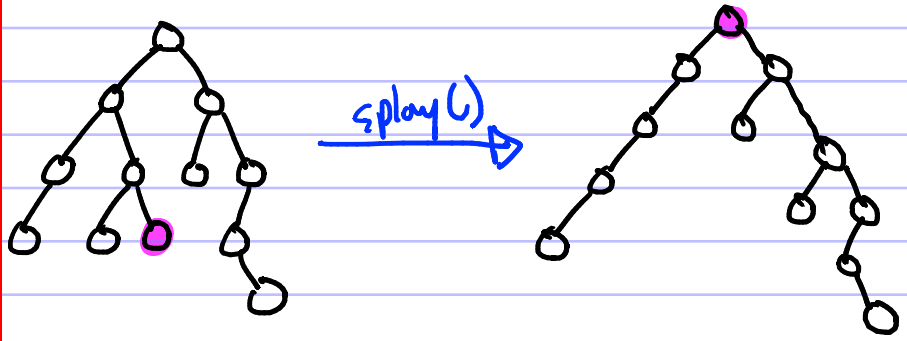
Zig-Zag and Zag-Zig



Reduces height of tree by 1.

splay()

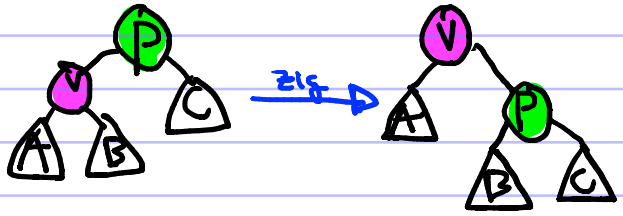
We compose zig(), zig zig(), zag zag(), zig zag(), zag zig() to create an operation called splay() that moves a node from any starting location to the root of the tree:



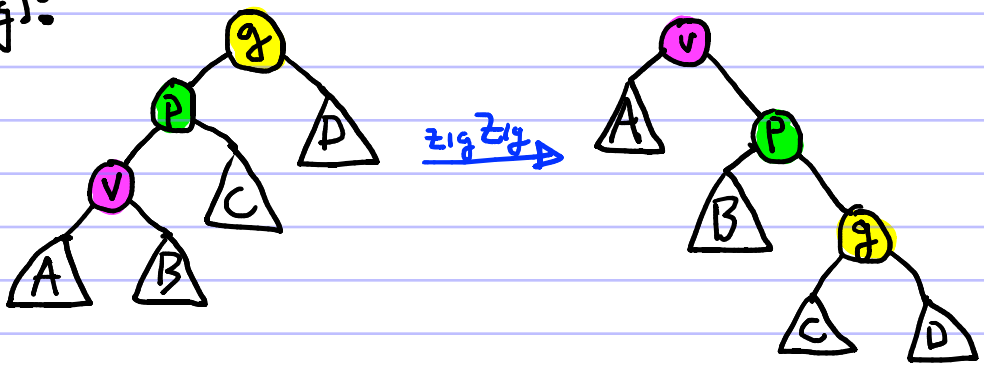
Just repeat the following step until v is the root:

Let p and g be parent and grandparent of v.

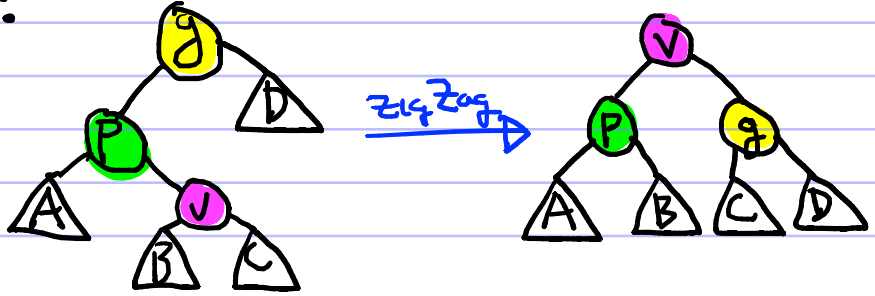
If p is root, zig:



If v is left (right) child of p and p is left (right) child of g, zig zig (zag zag):



If v is left (right) child of p and p is right (left) child of g, zag zig (zig zag):



find() (O(log n) amortized)

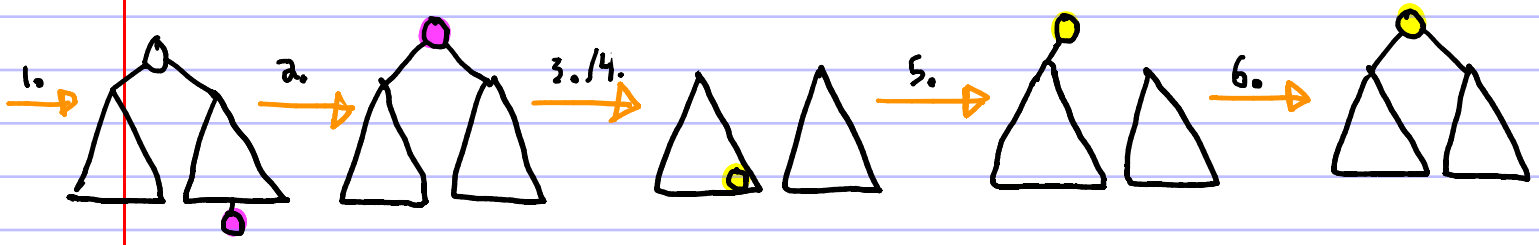
1. Search for node in usual BST way.
2. splay() node.
3. Return it.

insert() (O(log n) amortized)

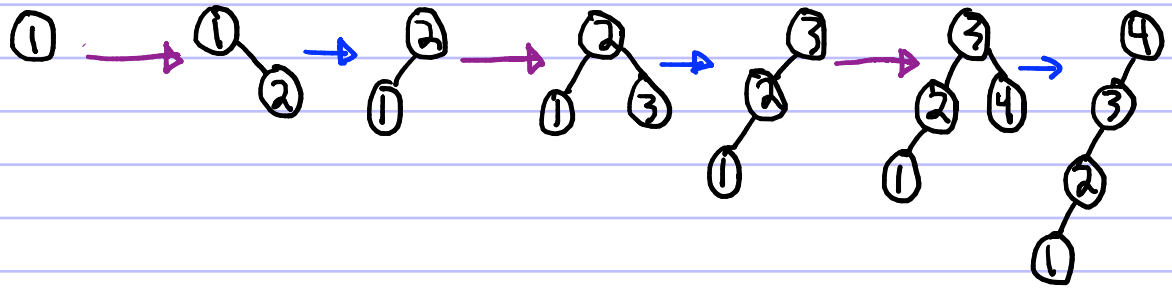
1. Search for node insertion site in usual BST way.
2. Insert node.
3. splay() node.

delete() (O(log n) amortized)

1. Search for node in usual BST way.
2. splay() node.
3. Remove node (creating two subtrees).
4. Find largest element in left subtree in usual BST way. Call it m_L.
5. splay() m_L to top of left subtree.] No right child of m_L now.
6. Set the right child of m_L to be the root of the right subtree.

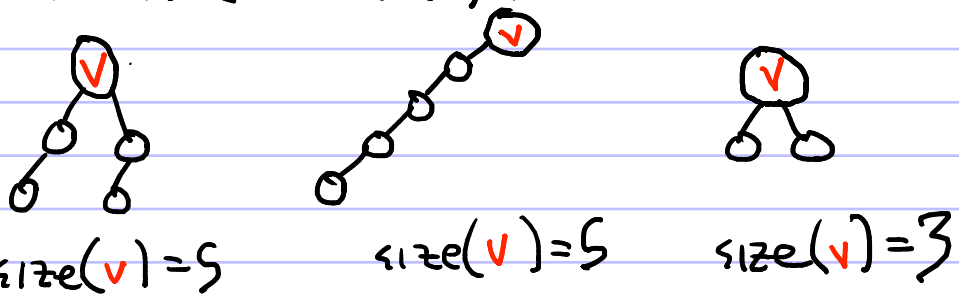


This is all fine and rad. But why are these operations O(log n) amortized since the tree can be very unbalanced?



AMORTIZED ANALYSIS OF SPLAY TREES

Messiness: Let size(v) be size of a subtree with root v.

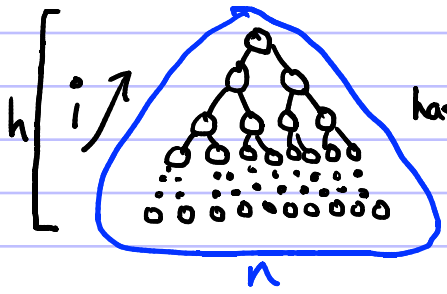


Let rank(v) = $\log_2(\text{size}(v))$, i.e. the log of the size of the subtree rooted at v.

Messiness = $\sum_{v \in \text{tree}} \text{rank}(v)$.

has messiness $\sum_{i=0}^{n} \log_2(i) = \log(n!)$
$= \Theta(n \log n)$

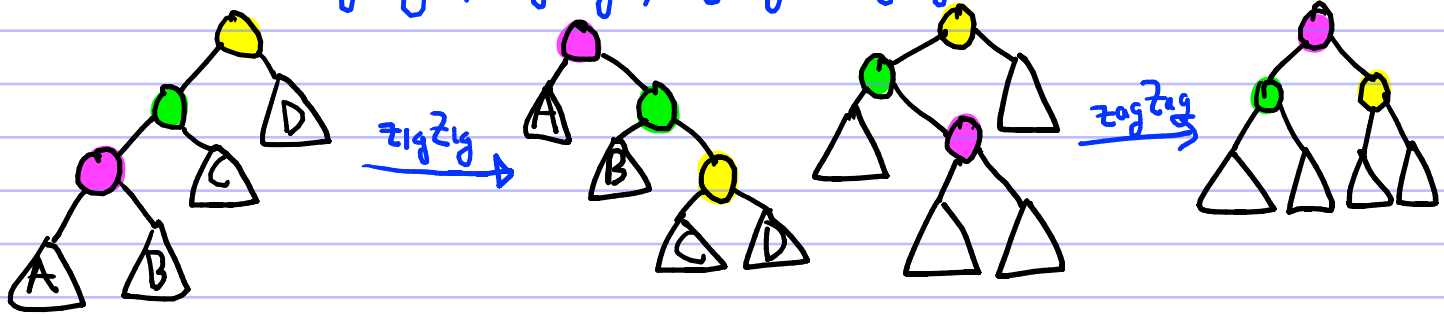
A more balanced tree has less messiness.



$h = \log_2 n$

has messiness $\sum_{i=0}^{h} 2^{h-i} \log(i) =$
$\Theta(n)$

Consider how zigZig(), zagZag(), zigZag(), zagZig() affect messiness:



rank(O) goes up, rank(O), rank(O) go down.
Let O_{i-1} and O_i be node at start and end of ith operation during splay.

Then $\text{rank}(O_{i-1}) = \text{rank}(O_i)$, $\text{rank}(O_{i-1}) \leq \text{rank}(O_{i-1})$, $\text{rank}(O_i) \leq \text{rank}(O_i)$.
So $\Delta\text{messiness} = \text{rank}(O_i) - \text{rank}(O_{i-1}) + \text{rank}(O_i) - \text{rank}(O_{i-1}) + \text{rank}(O_i) - \text{rank}(O_{i-1})$.
$= \text{rank}(O_i) - \text{rank}(O_{i-1}) + \text{rank}(O_i) + \text{rank}(O_i) - \text{rank}(O_{i-1}) - \text{rank}(O_{i-1})$.
$\leq \text{rank}(O_i) + \text{rank}(O_i) - \text{rank}(O_{i-1}) - \text{rank}(O_{i-1})$

Δ messiness $\leq$ rank(O_i) + rank(O_i) $-$ 2 rank(O_{i-1})

Thm: $\frac{\log a + \log b}{2} \leq \log\left(\frac{a+b}{2}\right)$

So $\text{rank}(O_{i-1}) + \text{rank}(O_i) \leq 2\log\left(\frac{\text{size}(O_{i-1}) + \text{size}(O_i)}{2}\right) \leq 2\log\left(\frac{\text{size}(O_i)}{2}\right)$

$$\text{rank}(O_i) \leq 2\,\text{rank}(O_i) - 2 - \text{rank}(O_{i-1})$$

So Δ messiness $\leq 3\,\text{rank}(O_i) - 3\,\text{rank}(O_{i-1}) - 2$ per zigzig(), zigzag()...

So total Δ messiness of all zigzag(), zigzig(),... in a splay() is:

$$\sum_{i=0}^{Z} 3\,\text{rank}(O_i) - 3\,\text{rank}(O_{i-1}) - 2 = 3\,\text{rank}(O_Z) - 3\,\text{rank}(O_0) - 2Z.$$

$$\leq 3 \cdot \underbrace{\log n}_{\text{rank(root of tree)}} - 2Z = 3\log n - 2Z$$

where Z is # of zigzig(), zigzag(),...

Total amortized cost of splay is then $\underbrace{O(3\log n - 2Z)}_{\Delta \text{ messiness}} + \underbrace{O(Z)}_{\text{real work}} = O(\log n)$

Splay in some sense has cost $O(\log n - Z)$, where Z is depth of splayed node.

Can achieve amortized $O(\log n)$ find() by charging cost of going down tree, which is $O(Z)$ to splayed node. ← If we don't find one, splay something nearby!

Similar trick works for insert(), delete(): pay for your searches by restoring cleanliness after you do them via a splay().

HOW GOOD ARE SPLAY TREES?

Amortized $O(\log n)$ find(), insert(), delete().] Optimal worst-case via sorting.

Consider a sequence of find() operations:

...find(7), find(9), find(1), find(3), find(22), find(17), ...

The splay tree does some amount of work across this sequence.
(work: searching down the tree and splaying)

That add to some total time t_{splay}.

Now consider the best BST for this sequence that does the minimum amount of work, including those that know the sequence ahead of time. Call the total running time for this BST t_{best}.

Conjecture: for every sequence of finds, $t_{splay} = O(t_{BEST})$

"Dynamic optimality conjecture"
[Sleator, Tarjan 1985]

Any always-balanced tree has time $t_{BALANCED} = O(\log(n) \cdot t_{BEST})$.
(always-balanced: not splay trees)

Homework: Design a sequence of finds() that turns an initially perfectly balanced splay tree into a tree consisting of a single chain. What operations increase messiness?