Splay Trees

What's a splay tree?

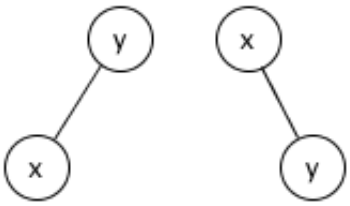
A splay tree is a binary search tree where we adjust the tree structure after every access in order to move the accessed element to the top. This is done by repeatedly performing a series of rotations to the element and its neighbors.

How do we splay?

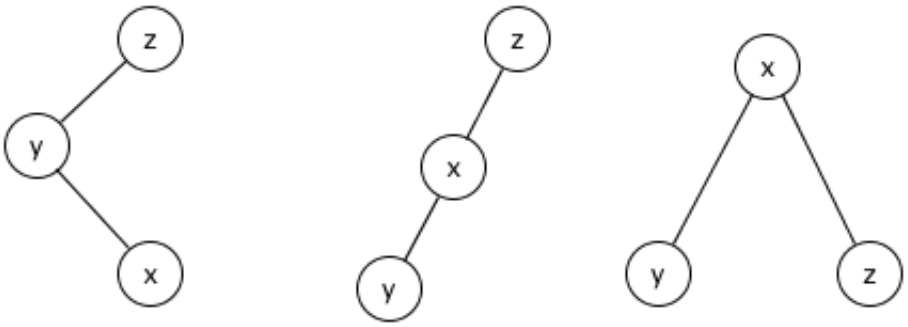
There are three cases for a splay, a zig, a zig-zag, and a zig-zig. Each can be done in at most two rotations, and moves the element up by one with each rotation.

The diagrams below show how we splay in various cases:

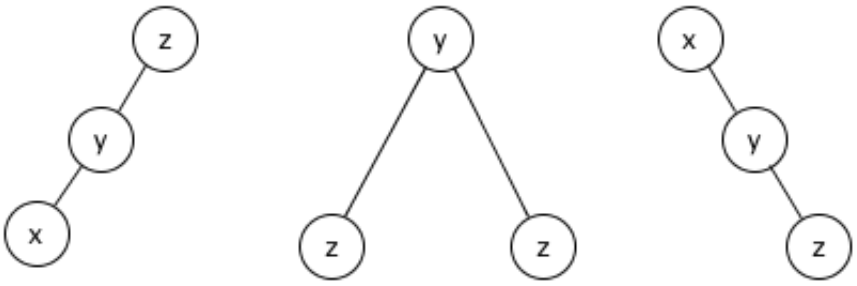
Zig



Zig Zag



Zig Zig



How fast is a splay tree?

To analyze the cost of accesses in a splay tree, we'll use the potential method. We'll give each of the nodes a weight of 1, and consider the *rank* and *score* of a node. The score of node n, $s(n)$ will be the sum of the weights of the subtree rooted at n. The rank of n, $r(n)$ will be

$$\lfloor \lg s(n) \rfloor$$

Initially, we'll start out with $r(x)$ on each node, x, or if you like potential functions, call the potential of the tree the sum of all the ranks in the tree. Here's an example of a splay tree with the ranks at each node:

There are a few useful properties of defining the rank in this way:

- If we rotate nodes x and y, only the ranks of x and y change.
- The potential of a perfectly balanced tree is $O(n)$, and the potential of a simple chain is $O(n \lg n)$

These definitions and facts bring us to our first proof, that of the **access lemma**.

The Access Lemma

The amortized number of splay operations we perform to access node x in a tree rooted at t is at most $3(r(t) - r(x)) + 1$

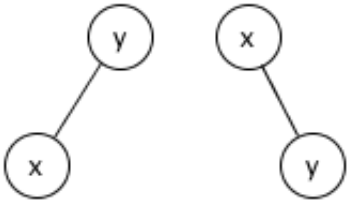
Recall that initially each node, y, has $r(y)$ tokens on it. We'll maintain this invariant by allocating $3(r(t) - r(x)) + 1$ tokens for the access.

First, we state the *rank rule*. If two siblings each have rank r, their parent must have rank at least $r + 1$, because the scores at the siblings must each be at least 2^r, so the score of the parent must be at least $2(2^r) = 2^{r+1}$. As a corollary, if a node has the same rank as its parent, its sibling must have strictly smaller rank.

We'll analyze the access splay-by-splay. On a given splay, let $r'(x)$ represent the rank of x after the splay, and let $r(x)$ be the rank before the splay. We'll claim that $3(r'(x) - r(x))$ will cover the cost of a zig-zag or zig-zig step. Note that because $r'(x)$ in one splay is $r(x)$ in the next splay, these costs will telescope to $3(r(t) - r(x))$, with the final $+1$ coming from the terminal zig.

The following diagrams show the rank differences for each case of a splay:

Zig

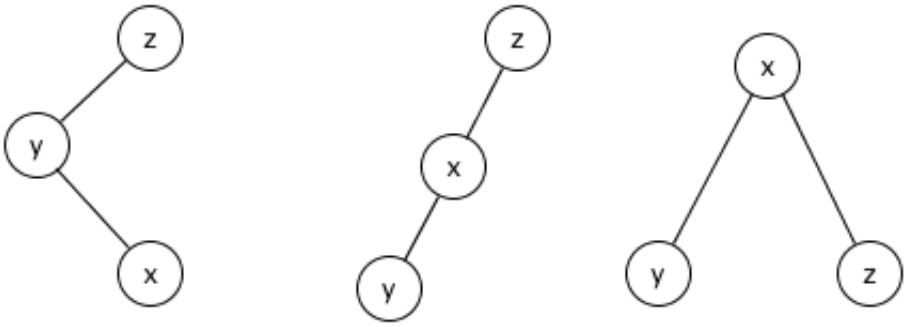


We spend 1 token in actual cost. We then take $r(y) - r(x)$ tokens and put them on y once we do the rotation. Our invariant about how many tokens each node has has not changed, and we've spent a total of $r(y) - r(x) + 1 \leq 1 + 3(r(y) - r(x))$ tokens

Zig Zag

We can split into two cases. Either the rank of the node we're splaying to the top doesn't change, or it does change. First, we'll analyze the case where it doesn't change.

No change in rank

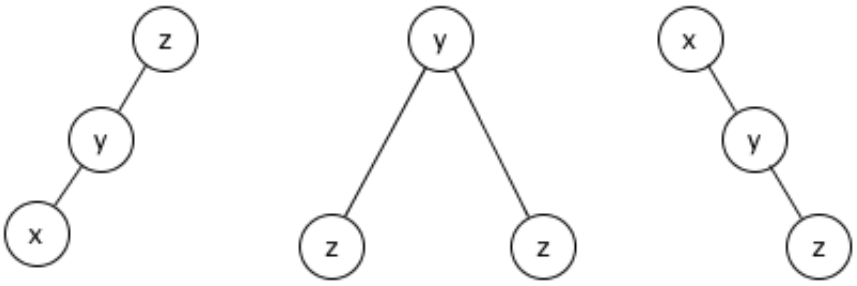


Here $r(x) = r'(x)$. By the rank rule, we know that either $r'(y) < r'(x)$ or $r'(z) < r'(x)$. In either case, at least one token is released from the tree. We'll use this token to pay for the splay step. So we paid a total of $3(r(x) - r(x)) = 0$ tokens to do this.

The rank of x increases

In this case, $r(z) - r(x) \geq 1$, so we can go ahead and use that to pay for the actual splay. Then we'll use another $r(z) - r(y)$ to pay for the extra tokens we need to keep on z after the rotation. In total we used $2(r(z) - r(x)) < 3(r(z) - r(x))$ tokens

Zig Zig



We break into cases again

No change in rank

Here, $r(x) = r(y) = r(z)$ initially. Note that after the first rotation, $r(z) < r(y)$. This case is analogous to the first case of the zig zag case.

The rank of x increases

In this case, we just pay $r(z) - r(x)$ for the actual cost of the splay, $r(z) - r(x)$ for rest needed on z, and use the same amount for the rest needed on z. We pay a total of $3(r(z) - r(x))$.

This concludes the proof of the access lemma!

The Balance Theorem

The Access Lemma is powerful, but it doesn't yet justify using splay trees. Here's where the Balance Theorem comes in. The Balance Theorem states:

A sequence of m splays in a tree of size n takes time $O((m + n) \lg n)$

To prove this, we first note that the amortized cost of a single access is, at most, $3 \lg n + 1$, which occurs when we access an element with rank 0. In order to account for the change in potential that may occur as well, we note that the most potential tree can have is $n \lg n$, and the least is 0. So in the worst case, we do

$$3m \lg n + 1 + \Phi_{initial} - \Phi_{final} = 3m \lg n + 1 + n \lg n \in O((m + n) \lg n)$$

work, which concludes the balance theorem.