Splay Trees (9)

1 SPLAY TREES

1.1 Review

We want to support fast insertions, deletions, and finds into a binary search tree.
Standard binary search trees don't cut it because operations take too long: as much as $\Theta(n)$ for stringy trees.
Balanced binary search trees remedy this, but are complex to implement, and rebalancing has a fair amount of overhead.
Splay trees are easy to implement, but tricky to analyze. They give bushy trees in an *amortized* sense, that we will discuss.

1.2 Basic Idea

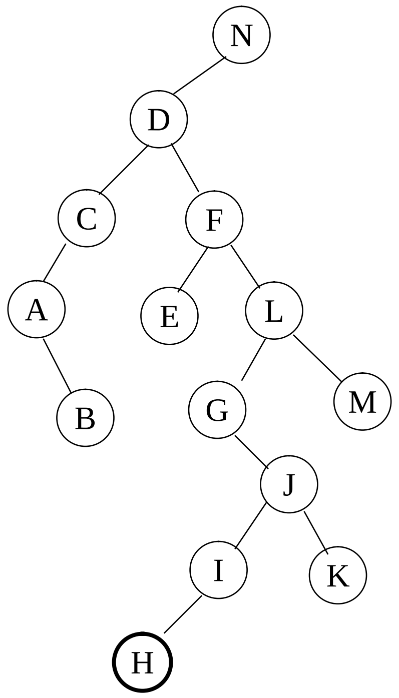
We allow the tree to become unbalanced.
The downside is that we might need to spend a lot of time finding a node that is deeper than it ought to be.
To make up for this, whenever we find a node, we move it up to the top!
Demonstrate rotation.

1.3 Splaying a Node

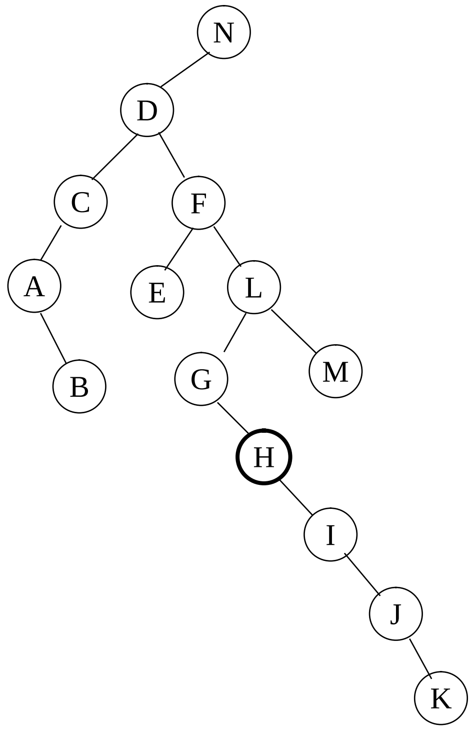
Brings node x to the top, two levels at a time.
Three rotation-pair cases:

- Zig: Child of root w... rotate node.
- Zig-zag: Left child of a right child z of w or right child of a left child... rotate node twice.
- Zig-zig: Left child of a left child y of z or right child of a right child... rotate parent, then rotate node.

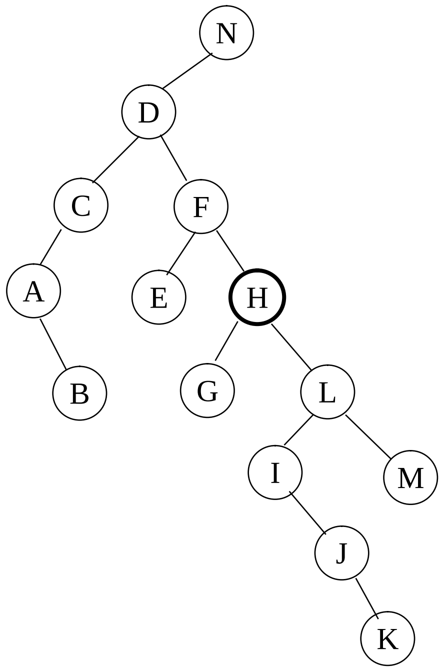
1.4 Splay Example Step 1



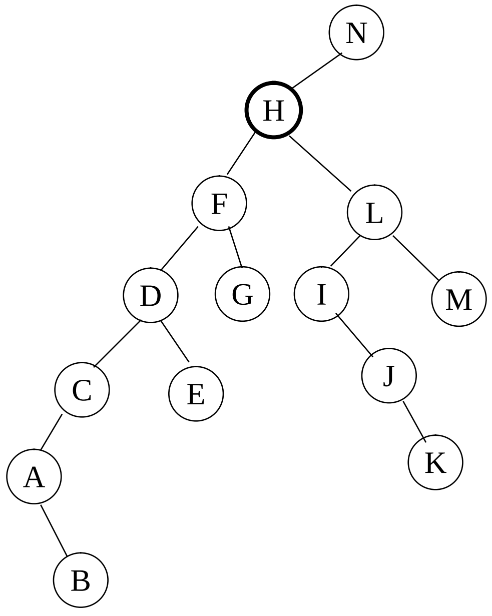
1.5 Splay Example Step 2



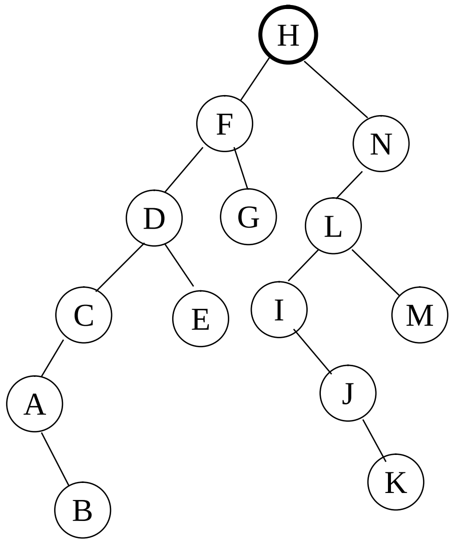
1.6 Splay Example Step 3



1.7 Splay Example Step 4



1.8 Splay Example Step 5



1.9 Splay Properties

We can observe what happens to the depth of various nodes in the tree when we splay x.

- The splayed node x?
- Children of x?
- Other nodes?
- Nodes along the path to the splayed node?

1.10 Operations Using Splays

We can perform all the standard binary-search-tree operations in terms of splays.

- Find: Locate the node, splay it to the top. If node is not found, splay last found node to the top.
- Min: Find $-\infty$.
- Max: Find ∞.
- Successor: Find node. Compute min of right subtree. (Much easier!)
- Alternate insert: Find node. Find min in right subtree (or max in left subtree, depending on what was found). It will have no left (or right, respectively) child, and therefore a place for the new element.
- Delete: Find node. Find max of left subtree. It will have no right child, so make the right subtree the new right child.

1.11 Splay Sort

Do a sequence of inserts, then traverse the resulting tree. How?

> Find min, then a series of successors.

Running time: Clearly $O(n^2)$. Why?

> A splay just goes to the bottom and rotates back up. That's $O(n)$. So, a sequence of n is $O(n^2)$.

We will see, in fact, that any sequence of k splays on a tree with n nodes runs in $O(k \log n)$.
Amazing!
We will use potential functions and *amortized analysis* to show this.
Therefore, what is splay sort's running time?

> $O(n \log n)$.

2 AMORTIZED ANALYSIS

2.1 Idea of Amortized Analysis

Closely related to the motivation for potential functions.
Sometimes the worst-case running time of an operation does not accurately capture the worst-case running time of a *sequence* of operations.
The idea of amortization is that we will charge induhvidual operations with the *average* cost for all operations in the sequence.
So, if a sequence of 100 operations requires 1000 units of time, we'll say each operation has an amortized cost of 10 (even if one of them takes 901 units and the rest take unit time).

2.2 Classic Example

Incrementing an n-bit binary number.

- 0 1 1 0 0 1 1 1 1 0 0 1 1 1 + 1 =
- 0 1 1 0 0 1 1 1 1 0 1 0 0 0
- (4 flips)

How many bit flips in the worst case? Best case?

> Can be all n, can be just 1.

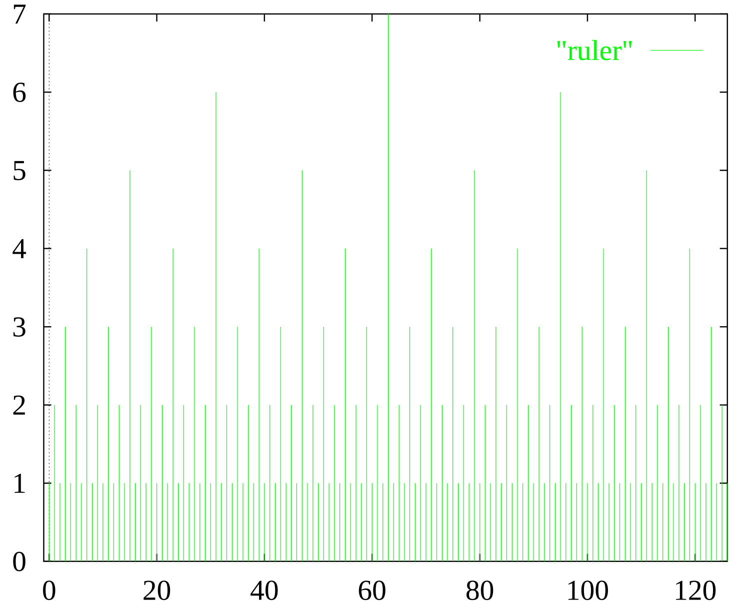
2.3 Increment Code

```
bit *
increment(bit *x)
{
   int i = n-1;
   while (i >= 0  &&  bit[i] == 1) {
     bit[i] = 0;
     i--;
   }
   if (i >= 0)
     bit[i] = 1;

   return bit;
}
```

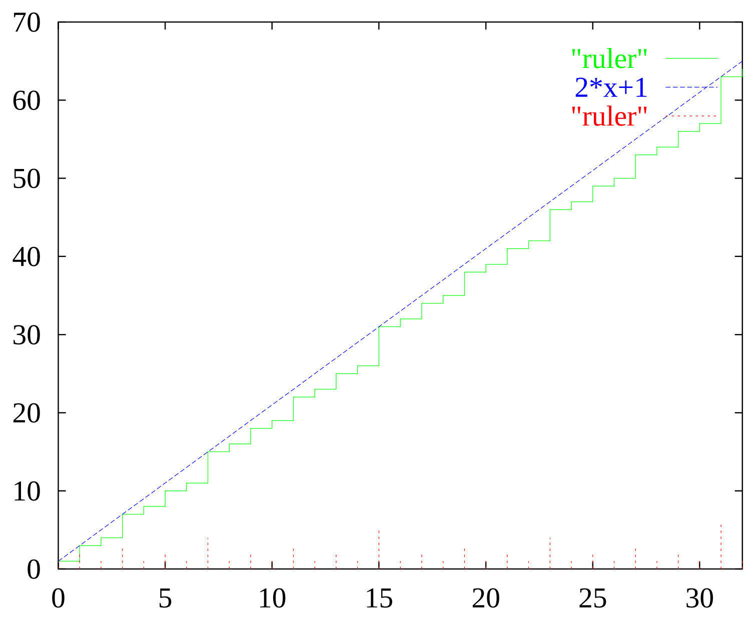
2.4 Counting Flips

Number of flips for incrementing all 7-bit numbers. (Number is on x axis, number of flips on y.) Can see the worst case, but it's rare.



2.5 Sequence of Flips

Total flips for incrementing, in sequence, all 5-bit numbers.



Note: Total number of flips for incrementing 0 to x never goes above $2x + 1$.
When does it equal $2x + 1$?

> When x is a power of 2. You get that big kick at the end as a bunch of ones turn to zeros.

2.6 Potential Function

We've been analyzing individual while loops with potential functions. This time, we'll analyze the entire "increment" operation.
Let x be the current value of the counter. Let $\Phi(x)$ be a potential function that returns the number of "on" bits in x.
Let's say each bit flip costs one unit.
The worst-case cost of increment number t is $c_t \leq n$. We'll define the amortized cost to be $\hat{c}_t = 2$. That is, we'll *pretend* that each increment costs 2, even when it's really n (or even if it's 1).

2.7 It Works!

We need to show that the fundamental inequality of potential functions holds:

$$\hat{c}_t + \Phi(x_{t-1}) - \Phi(x_t) \geq c_t.$$

$$\begin{aligned}
&\hat{c}_t + \Phi(x_{t-1}) - \Phi(x_t) \\
&\quad = \quad 2 + \Phi(x_{t-1}) - \Phi(x_t) \\
&\quad \geq \quad 2 + \text{number of trailing ones in } x_{t-1} \text{ minus } 1 \\
&\quad = \quad 1 + \text{number of trailing ones in } x_{t-1} \\
&\quad = \quad c_t
\end{aligned}$$

Note that the potential function measures the amount of work that went into building the data structure.

3 ANALYSIS OF SPLAY TREES

3.1 Splay Operations

Recall that inserts, deletes, and finds in a splay tree all consist of some number of splay operations (moving a node to the root) and a constant amount of pointer diddling.
We will show that splay operations have $O(\log n)$ amortized time and it will follow that a sequence of n inserts and deletes on an initially empty tree takes $O(\log n)$ per operation.
Like a balanced tree, but not!

3.2 Analysis Setup

Here's the setup for the analysis. First, define the *size* of a node x to be the number of nodes in the subtree rooted at x.

- A unit of work is a rotation.
- A splay at some node x up to the root r is the operation, and its amortized cost is $3(\log s(r) - \log s(x)) + 1$.
- The potential function for a tree T is $\Phi(T) = \sum_{x \in T} \log(s(x))$.

We need to show that the operation of splaying a node x to the root satisfies the potential function inequality. We'll do this one case at a time.

3.3 Case Setup

Let s_{t-1} and s_t be the size functions just before and after the step. Let y be the parent of x and z be the parent of y (if it exists).
We'll define the amortized cost to be $3(\log s_t(x) - \log s_{t-1}(x)) + 1$ for Zig and $3(\log s_t(x) - \log s_{t-1}(x))$ for the other cases. The amortized cost for a sequence of h steps splaying x up the tree, therefore, is

$$\sum_{t=1}^{h-1} 3(\log s_t(x) - \log s_{t-1}(x)) + 3(\log s_h(x) - \log s_{h-1}(x)) + 1 = 3(\log s_h(x) - \log s_0(x)) + 1.$$

This is $O(\log n)$. Why?

$$s_0(x) \geq 0, \; s_h(x) = n.$$

This means that our amortized cost for each complete splay is $O(\log n)$, even if we start deep in the tree. The intuition is that, the only way we could make the tree that deep in the first place is to have performed operations that increase the potential substantially. So, the average still works out.

3.4 Zig Case

We need to show that $\hat{c}_t + \Phi(D_{t-1}) - \Phi(D_t) \geq c_t$.
Here, we'll take $\hat{c}_t = 3(\log s_t(x) - \log s_{t-1}(x)) + 1$. The actual cost $c_t = 1$. Why?

Zig is one rotation.

$$\begin{aligned}
&\hat{c}_t + \Phi(D_{t-1}) - \Phi(D_t) \\
&= 3(\log s_t(x) - \log s_{t-1}(x)) + 1 + \sum_{w \in T} \log(s_{t-1}(w)) - \sum_{w \in T} \log(s_t(w)) \\
&= 3\log s_t(x) - 3\log s_{t-1}(x) + 1 + \log(s_{t-1}(x)) + \log(s_{t-1}(y)) - \log(s_t(x)) - \log(s_t(y)) \\
&= 2\log s_t(x) - 2\log s_{t-1}(x) + 1 + \log(s_{t-1}(y)) - \log(s_t(y)) \\
&\geq 2(\log s_t(x) - \log s_{t-1}(x)) + 1 \\
&\geq 1 = c_t.
\end{aligned}$$

First step: Only x and y change sizes. Next step: $s_t(y) \leq s_{t-1}(y)$. Finally, $s_t(x) \geq s_{t-1}(x)$.

3.5 Zig-Zig Case

We're splaying x, with y and z as the immediate parents.
We need to show that $\hat{c}_t + \Phi(D_{t-1}) - \Phi(D_t) \geq c_t$.
Here, we'll take $\hat{c}_t = 3(\log s_t(x) - \log s_{t-1}(x))$. The actual cost $c_t = 2$. Why?

Zig-zig is two rotations.

$$\begin{aligned}
&\hat{c}_t + \Phi(D_{t-1}) - \Phi(D_t) \\
&= 3(\log s_t(x) - \log s_{t-1}(x)) + \sum_{w \in T} \log(s_{t-1}(w)) - \sum_{w \in T} \log(s_t(w)) \\
&= 3\log s_t(x) - 3\log s_{t-1}(x) + \log(s_{t-1}(x)) + \log(s_{t-1}(y)) + \log(s_{t-1}(z)) - \log(s_t(x)) - \log(s_t(y)) - \log(\\
&\geq 2\log s_t(x) - \log s_{t-1}(x) - \log(s_t(z)) \\
&\geq 2.
\end{aligned}$$

First step: Only x, y, and z change sizes. Next step: $s_t(x) = s_{t-1}(z)$ and $s_t(x) \geq s_t(y)$ and $s_{t-1}(y) \geq s_{t-1}(x)$.

The last step here is sort of tricky.

- Claim: $2\log s_t(x) - \log s_{t-1}(x) - \log s_t(z) \geq 2$.
- Let $n = s_t(x)$.

- Note that if $a+b=n$, then $-\log a - \log b \geq -2\log n + 2$, and this is attained at $n/2$ (this isn't hard to show with calculus).
- Note that $s_t(z) + s_{t-1}(x) = n$. So,

$$\begin{aligned}
&2\log s_t(x) - \log s_{t-1}(x) - \log s_t(z) \\
&\quad\geq\ 2\log n - 2\log n + 2 \\
&\quad=\ 2.
\end{aligned}$$

Zig-zag case a little different, but same result.

3.6 Entire Splay

Repeating this analysis for each pair of rotations as we go up, plus the final single rotation if needed, we get a total amortized cost of $3(\log n - \log s_0(x)) + 1 = O(\log n)$, where the potential function inequality is satisfied (string them all together).
Therefore, a splay operation in a tree of size n has an amortized cost of $O(\log n)$. So, a sequence of k splays will cost $O(k \log n)$.