Splay Trees

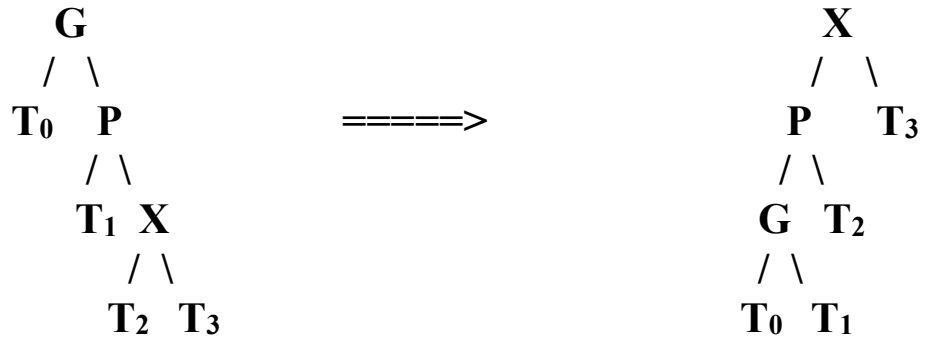
A splay tree is a normal binary search tree, structurally. In fact, the only difference between a normal binary search tree and a splay tree is that in a splay tree, after you perform a insertion, deletion, or search, you must "splay" the tree. A splay consists of restructuring the tree to bring the accessed node to the root of the tree. After every insertion, the inserted node gets splayed. After every deletion, the parent of the deleted node gets splayed. After every search, the searched node gets splayed. If a value not in the tree is searched, then the leaf node where the searched value would be inserted is splayed.

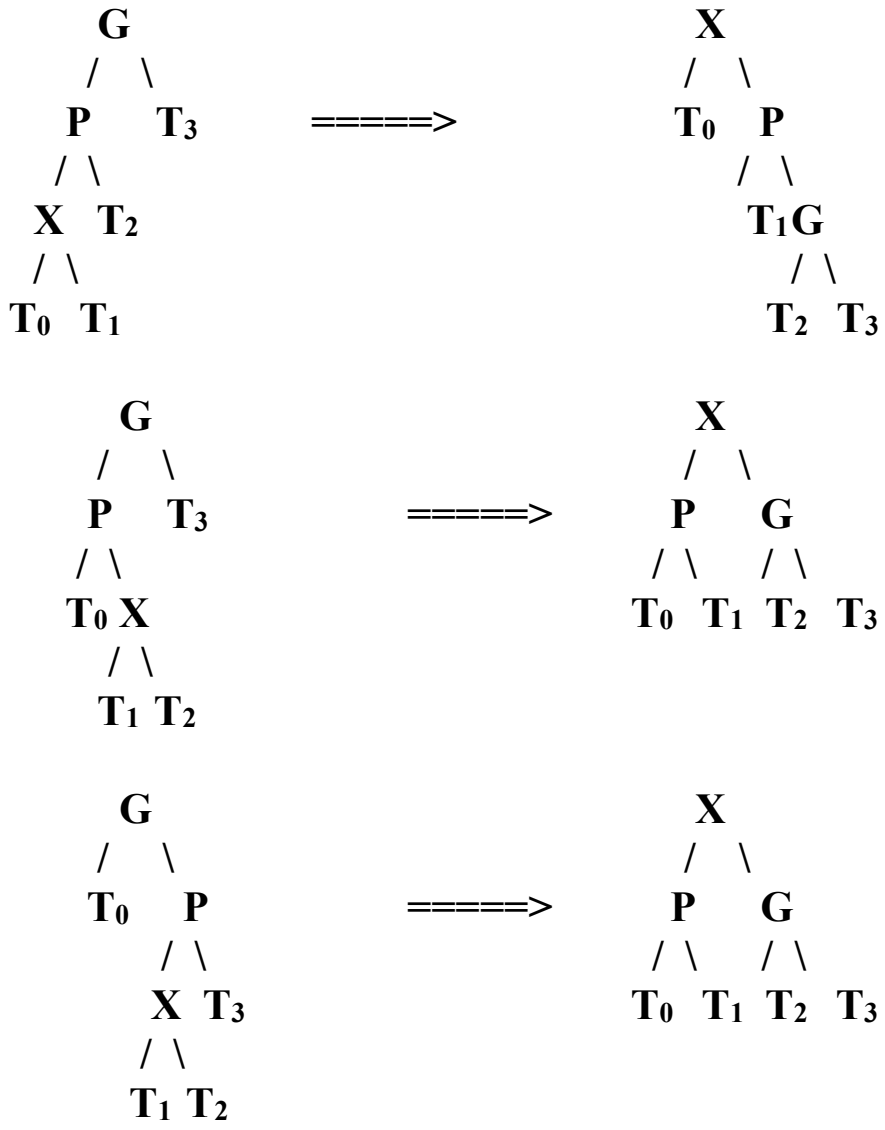
How to Splay a Node

To splay a node in a splay tree, a series of suboperations must be performed. These can be broken down into two categories:

1) When the splayed node has a grandparent in the tree.
2) When the splayed node's parent is the root.

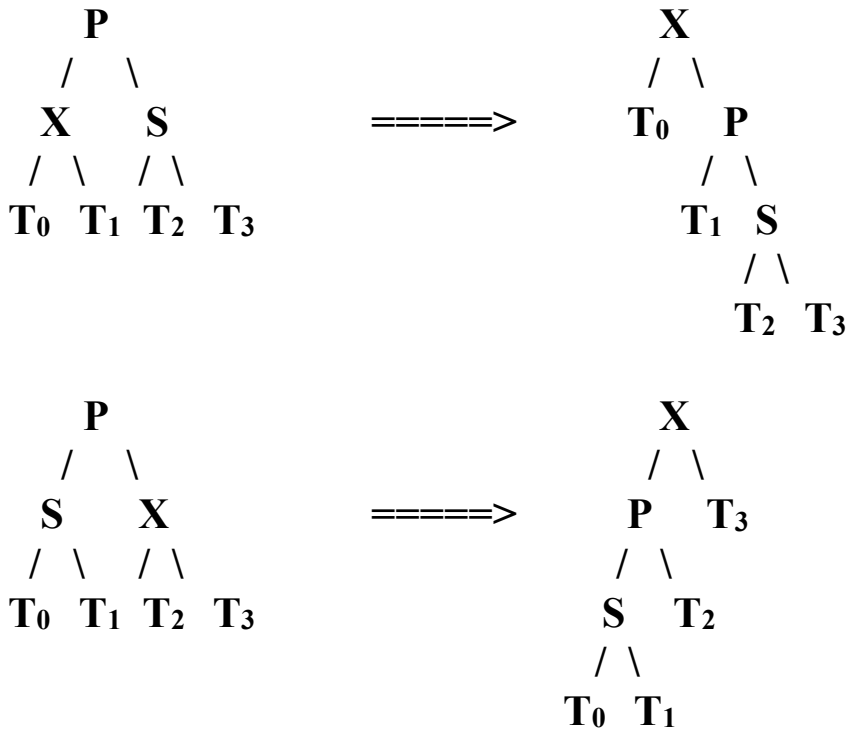
In the first situation, identify both the parent and grandparent of the node to be splayed. Label these nodes as P and G, respectively. Label the node to be splayed as X. In each of these situations, restructure the subtree to be rooted at X as follows:





In each of these cases, by swapping a few references, we move the node to be splayed a distance of two closer to the root.

But, these transformations won't work when the node to be splayed is directly underneath the root of the tree. But, we can just handle these in a separate case (S is the sibling of X):

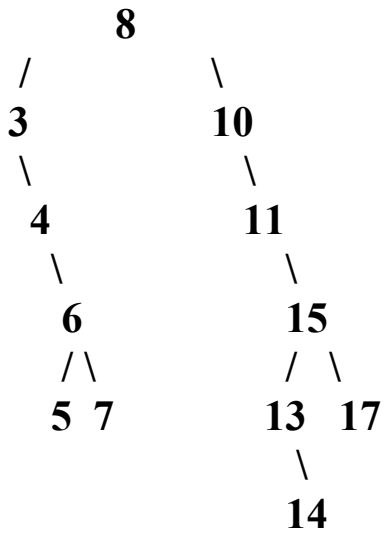


Even though it looks like a pain to remember these six transformations, it's easier once you realize these guidelines:

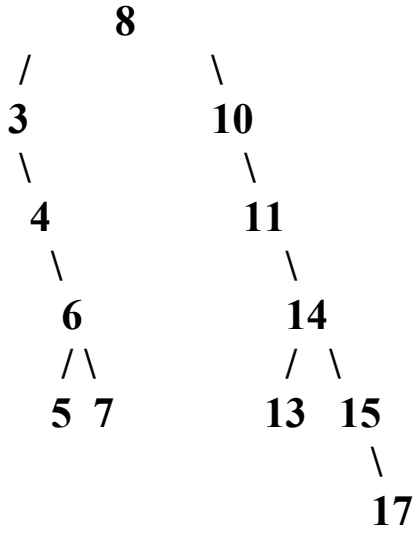
Bring X up to the root of the subtree being rearranged, then let the parent and grandparent fall into place in a "straight" line below X. Once you do this, the subtrees all must occupy a specific location within the adjusted structure. If the node being splayed is a child of the root, place X at the root, and then place the parent and sibling nodes in a "straight" line below them.

These transformations are only suboperations of a splay. To fully splay a node, continue calling splay operations on the node until it reaches the root of the tree.

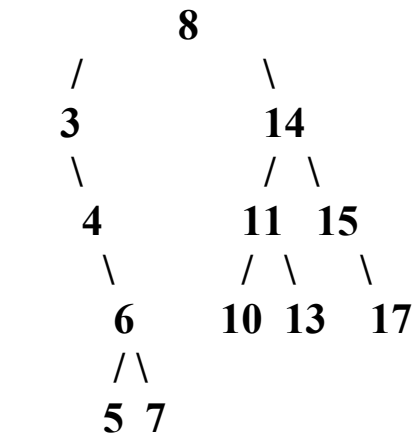
Consider splaying the 14 in the following example:



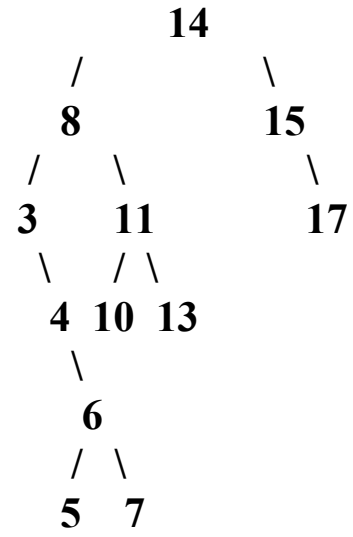
First we identify the parent(13) and grandparent(14) and restructure as follows:



Now, the parent is 11 and the grandparent 10, so restructure as follows:



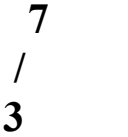
Finally, we have one more splay left: to put 14 at the root:

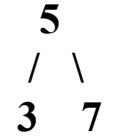


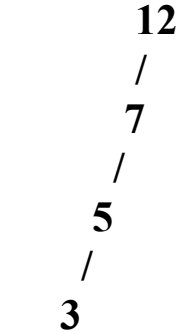
Let's trace through several operations on a splay tree. I will first list the operations, and then list each resulting tree, after each splaying.

1) Insert 3
2) Insert 7
3) Insert 5
4) Insert 12
5) Search 4
6) Delete 3
7) Insert 6
8) Insert 15
9) Insert 2
10) Delete 12
11) Search 5

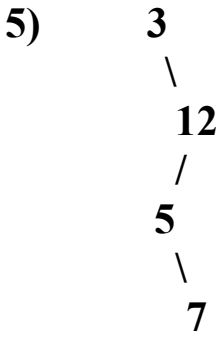
1) 3

2)


3)


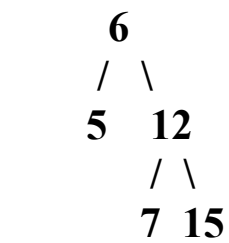
4)


5)

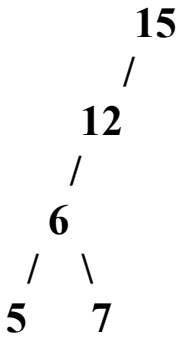


6)

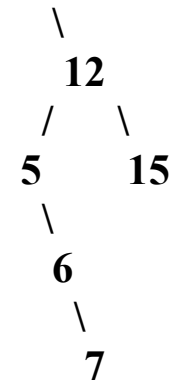
7)



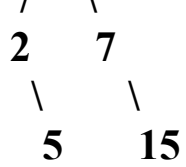
8)



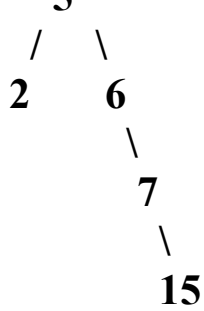
9)



10)



11)



Amortized Analysis of Splay Trees

One model of amortized analysis is having a set cost for each operation such that the set cost may be less than the actual cost on a single instance, but over a long number of steps, the sum of the set cost is at least as much as the actual cost.

For our splay tree analysis, imagine that each node must maintain a certain amount of money. (We will pay a certain amount of money for each node.) Also, we will pay d dollars for the splaying of each node in the tree, where d is the depth of the node on its initial insertion. Notice that we are only paying for the splaying of a node. That is because the splaying cost is proportional to the insertion, searching, and deleting cost.

It will be shown that if a payment of O(log n) dollars is made for each splaying in an empty tree with n operations, then that will be enough to cover the cost of each splaying AND the maintenance of each node.

First define n(v) to be the size of a node v. This size is the number of nodes in the subtree rooted at v, including external nodes. Next, define the rank of a node v $r(v) = \log_2 n(v)$. Finally, define r(T) to be the sum of all the ranks of all the nodes. Clearly, r(T) must increase from when T is empty until T contains several values as a result of n splay tree operations.

Next, we will define δ as the change in r(T) determined by a single splaying suboperation. Each suboperation moves a node either 1 or 2 steps closer to the root of the tree. Let r'(v) denote the rank of a node v right AFTER a splaying suboperation. Then δ is simply the sum of r'(v)-r(v) over all nodes v in the splay tree.

We will now show that for any zig-zig or zig-zag suboperation, $\delta \leq 3(r'(x)-r(x))-2$,

and for any zig operation
$\delta \leq 3(r'(x)-r(x))$.

<u>Case 1: Any zig-zig operation</u>
Note that the only nodes whos ranks can change in this type of an operation are the three nodes x, y, and z, that are directly involved. The rest of the nodes maintain the exact same number of successors. Let y be the parent of x and z be the parent of y. Then, we have $r'(x) = r(z)$, because x replaces z as the root node of the subtree in question, $r'(y) \leq r'(x)$, since y is a child of x after the suboperation, and $r(y) \geq r(x)$, since y used to be x's parent.

$\delta = r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$
$\leq r'(y) + r'(z) - r(x) - r(y)$, substituting for $r(z)$
$\leq r'(y) + r'(z) - 2r(x)$, since this value subtracts a smaller value than $r(x)+r(y)$ from the expression.
$\leq r'(x) + r'(z) - 2r(x)$, since $r'(x) > r'(y)$.

$n(x) + n'(z) \leq n'(x)$ because $n(x)$ is the sum of nodes that are distinct from $n'(z)$, and all of these nodes are rooted at x after the suboperation.

Now, we will use a mathematical fact that says, given that $c \geq a + b$, then $\log a + \log b \leq 2\log c - 2$, for all $a, b > 0$.

(If you want to derive this on your own, you must establish that $\log a + \log b \leq 2\log ((a+b/2)$, which follows from the fact that $\log x$ is always concave down, for all $x>0$.)

Since $n(x) + n'(z) \leq n'(x)$, we have that

$$\log n(x) + \log n'(z) \leq 2\log n'(x) - 2$$
$$r(x) + r'(z) \leq 2r'(x) - 2$$

Now, use this fact to find an inequality that δ satisfies:

$$\begin{aligned}
\delta &\leq r'(x) + r'(z) - 2r(x), \\
&= r'(x) + r'(z) + (r(x) - 3r(x)) \\
&= r'(x) + (r'(z) + r(x)) - 3r(x) \\
&\leq r'(x) + (2r'(x) - 2) - 3r(x), \text{ substituting from above} \\
&= 3r'(x) - 3r(x) - 2 \\
&= 3(r'(x) - r(x)) - 2
\end{aligned}$$

<u>Case 2: Any zig-zag operation</u>

In the text they present the work to show that $\delta \leq 3(r'(x) - r(x) - 2$ when we are doing a zig-zag operation. The only thing that changes in this proof is that the children under y' and z' are distinct (instead of x and z' as was in the previous case). Thus, we can claim that $n'(y) + n'(z) \leq n'(x)$. This implies that $r'(y) + r'(z) \leq 2r'(x) - 2$. Then, in our equation for δ we have:

$$\begin{aligned}
\delta &\leq r'(y) + r'(z) - 2r(x) \\
&\leq 2r'(x) - 2 - 2r(x) \\
&= 2(r'(x) - r(x)) - 2 \\
&\leq 3(r'(x) - r(x)) - 2
\end{aligned}$$

Case 3: Any zig operation

In the case of a single zig operation, the only changes in rank deal with the nodes x and y.

We have that $r'(y) \leq r(y)$ and $r(x) \leq r'(x)$:

$$\begin{aligned}\delta &= r'(x) + r'(y) - r(x) - r(y) \\ &= r'(x) - r(x) + (r'(y) - r(y)) \\ &\leq r'(x) - r(x), \text{ because } r'(y) - r(y) \text{ is a negative quantity.} \\ &\leq 3(r'(x) - r(x))\end{aligned}$$

Total variation r(T)

Next, we will show that Δ, the total variation in r(T) over a single splay operation on a node of depth d is bounded as follows:

$$\Delta \leq 3(r(t) - r(x)) - d + 2$$

Let the splaying consist of p suboperations. Then, based on d, we have that p = ceil(d/2). (This accounts for each operation being either a zig-zig or zig-zag except for the last.) Let δ_i stand for the change in r(T) during the ith splaying suboperation. Finally, let $r_i(x)$ stand for the rank of x after the ith suboperation.

Then we have:

$$\begin{aligned}\Delta &= \delta_1 + \delta_2 + \ldots + \delta_p \\ &\leq (3(r_i(x) - r_{i-1}(x)) - 2) + 2,\end{aligned}$$

Notice that each suboperation works on a node with the rank of a node from the previous operation. Thus, most of the $r_i(x)$ terms cancel. The +2 is for the possible last operation.

$= (3(r_p(x) - r_0(x)) - 2p + 2$

This accounts for the only terms in the sum that don't cancel. Note that $r_0(x)$ is nothing but r(x), the original rank of the node x before the splaying. Also, $r_p(x)$ is simply the rank of the root node, since that is where x will be after splaying.

$\leq 3(r(t)-r(x)) - d + 2$

So what does all of this prove???

Now, imagine that we pay 3r(t) +2 = O(log n) "dollars" for every operation. Using this money, we can pay for the cost of moving the splayed node from a depth of d in the tree to the root. This cost is d dollars, since each "step-up" is a constant time operation. In addition to this money, we can also pay the 3(r(t)-r(x)) – d + 2 dollars necessary to maintain the proper balance of money at each node. The sum of this money is 3r(t) - 3r(x) + 2, which must be less then or equal to 3r(t)+2 dollars. Finally, $3r(t)+2 \leq 3\log_2 n + 2 = O(\log n)$.

The key is to note that sometimes, this value, 3(r(t)-r(x)) - d + 2 will be negative. In these situations, we have extra money stored in the nodes, and this helps pay for a costly splay operation that moves a node from far down the tree to the root of the tree.

If this value is positive, then we have leftover money to replenish the cash at each node. Since each operation costs us at MOST O(log n) dollars on average, we can say that the amortized running time of any n operations on an empty splay tree is O(log n) for each operation.