- The splay tree is a balanced binary search tree. In a splay tree, mainly, for any node v that is encountered last on a search path, nodes on the root to v path P are reorganized so that v becomes the root and the induced subtree comprising nodes on P is roughly balanced. The latter operation of balancing a root to node simple path is called a splay operation whereas the key searching together with splay is called an access operation. The access operation leads to having frequently accessed items to go near the root. And, the splay operation compresses search paths by reducing the average depth of nodes along such paths. Apart from being a self-balanced binary search tree, since the balancing does not require storing any augmented information at nodes, the splay tree is self-adjusting. However, the splay tree provides only amortized guarantees.

- Almost all the functions in the ADT of splay tree are implemented in terms of splay operation. Hence, as part of the splay operation, tree balancing is done. For any node x at an even depth, splaying x involves issuing a series of two-level rotations. And, for any node x at an odd depth, splaying x involves issuing a series of two-level rotations followed by one one-level rotation. As detailed below, a two-level rotation decreases the depth of a node by two, and it comprises of two specific left/right rotations; also, a one-level rotation moves a node at level one to root, and it comprises of a specific single left/right rotation. [1] Since rotations preserve the BST property, the tree after applying these rotations continues to be a BST. Applying a series of two-level rotations helps some of the terms' telescope in the amortized analysis.

- Next, we define a potential function for amortized analysis. For every node i in the splay tree, let p_i be the access probability of node i, and let l_i be the length of the path from the root to i. Then, the expected cost of a search in the splay tree is $\sum_{i=1}^{n}(1+l_i)p_i$.[2] For every node i, let $A(i)$ be the sum of probabilities of accessing nodes in the subtree rooted at i. Then, $\sum_i A(i) = \sum_{i=1}^{n}(1+l_i)p_i$. In defining potential of a splay tree, for the convenience of algebra, we use $\lg(A(i))$, instead of $A(i)$.

  Specifically, we use the following potential function: $\Phi(T) = \begin{cases} 0 & \text{if } T \text{ is empty, and} \\ \sum_{x \in T} r(x) & \text{otherwise.} \end{cases}$
  Here, the rank $r(x)$ of x is defined as $\lg(size(x))$, where $size(x)$ is the number of nodes in the subtree rooted at x. Assuming the initial splay tree is empty, the initial potential is 0. Since the number of nodes in a tree can never be negative, it is immediate that after every dictionary operation, the potential of the splay tree is guaranteed to be greater than or equal to the initial potential.

- The following is the description of cases, based on which one-level and two-level rotations are defined. We let $s(z)$ and $s'(z)$, $r(z)$ and $r'(z)$ denote the size and rank of node z just before and just after the step in context, respectively.

  (i) $x.parent$ is the tree root, and

  (a) x is the left child of the root:

  This subcase is handled by a rightrotate at $x.parent$.

  The amortized cost of this one-level rotation
  $=$ actual cost $+(\Phi_{final} - \Phi_{initial})$

[1] The paper denotes a leftrotate with a zag and a rightrotate with a zig.
[2] we implicitly assumed to associate uniform access probability $\frac{1}{n}$ to each node

rightrotate(y)

[Courtesy: All figures in this note are from the paper cited at the end.]

$= 1 + (r'(x) + r'(y) - r(x) - r(y))$
$= 1 + (r'(y) - r(x))$ (as $r(y) = r'(x)$)
$\leq 1 + (r'(x) - r(x))$ (as $r'(y) < r'(x)$)
$\leq 1 + 3(r'(x) - r(x))$.

(b) x is the right child:

This subcase is handled by a leftrotate at $x.parent$.

The amortized analysis of this subcase is symmetric to analysis in (i)(a).

(ii) $x.parent$ is not the root, and

(a) if x and $x.parent$ are both left children:

This subcase is handled by a rightrotate at $x.parent.parent$ followed by a rightrotate at $x.parent$.

rightrotate(z) + rightrotate(y)

The amortized cost of this two-level rotation
$=$ actual cost $+(\Phi_{final} - \Phi_{initial})$
$= 2 + (r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z))$
$= 2 + (r'(y) + r'(z) - r(x) - r(y))$ (as $r'(x) = r(z)$)
$\leq 2 + (r'(x) + r'(z) - 2r(x))$. (as $r(x) \leq r(y)$ and $r'(y) \leq r'(x)$)

However, $2 + r'(x) + r'(z) - 2r(x) \leq 3(r'(x) - r(x))$.
$\Leftrightarrow -2r'(x) + r(x) + r'(z) \leq -2$.
$\Leftrightarrow \lg \frac{s(x)}{s'(x)} + \lg \frac{s'(z)}{s'(x)} \leq -2$.
And, for $a > 0, b > 0, a + b \leq 1$, maxima of $\lg a + \lg b$ is -2.

Therefore, the amortized cost is indeed upper bounded by $3(r'(x) - r(x))$.

(b) x and $x.parent$ are both right children:

This subcase is handled by a leftrotate at $x.parent.parent$ followed by a leftrotate at $x.parent$.

The amortized analysis of this subcase is symmetric to analysis in (ii)(a).

(iii) $x.parent$ is not the root, and

(a) if x is a left child and $x.parent$ is a right child:

This subcase is handled by a leftrotate at $x.parent$ followed by a rightrotate at $x.parent$.

leftrotate(y) + rightrotate(z)

The amortized cost of this two-level rotation
$= \text{actual cost} + (\Phi_{final} - \Phi_{initial})$
$\leq 2 + (r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z))$
$\leq 2 + (r'(y) + r'(z) - r(x) - r(y))$ (as $r'(x) = r(z)$)
$\leq 2 + (r'(y) + r'(z) - 2r(x))$. (as $r(x) \leq r(y)$)

However, analogous to analysis in (ii)(a), it can be shown that $2 + (r'(y) + r'(z) - 2r(x)) \leq 2(r'(x) - r(x))$.

Therefore, the amortized cost is upper bounded by $3(r'(x) - r(x))$.

(b) if x is a right child and $x.parent$ is a left child:

This subcase is handled by a rightrotate at $x.parent$ followed by a leftrotate at $x.parent$.

The amortized analysis of this subcase is symmetric to analysis in (iii)(a).

- From the amortized analysis of the above cases, the amortized cost of a splay is

$\leq$ sum of amortized subcosts of all two-level rotations plus the amortized cost of at most one one-level rotation
$= 1 + \sum_i 3(\text{rank of } x \text{ in the tree just after the } i^{th} \text{ two-/one-level rotation} - \text{rank of } x \text{ in the tree just before the } i^{th} \text{ two-/one-level rotation})$
$= 1 + 3(\text{rank of } x \text{ in the tree after the splay} - \text{rank of } x \text{ in the tree before the splay})$ (due to telescoping)
$= 1 + 3 \lg (\frac{\text{size of the subtree rooted at } x \text{ after the splay}}{\text{size of the subtree rooted at } x \text{ before the splay}})$
$\leq 1 + 3 \lg(n)$.

Hence, the amorized cost of splay is $O(\lg n)$.

- This dictionary ADT is supported with the following algorithms.

- access(k)[3]: a BST search for k followed by splaying the last accessed node[4]

[3] not called a search since it involves modifying the tree
[4] in another implementation, to avoid bottom-up pass, rotations are done while going downwards along the search path

(note that if it is an unsuccessful search, this algorithm splays the last accessed node)

- minimum(T): walks along the left spine, splays the last node on the left spine, and returns the minimum key stored at the root
- maximum(T) is defined analogous to minimum
- join(T_1, T_2)[5]: search for the largest key k in T_1; let r be the root of the resulting tree; hang T_2 as the right subtree of r
- split(T, i): access i in T; let r be the root of the resulting tree; delete the edge between r and the left/right child of r
- insert(i): split(T, i), attach two trees as children of root with key i[6]
- delete(T, i): assuming there exists a node with key i, split(T, i); join the left and right subtrees of root; delete the root node

As can be noted, almost all the operations rely on the access operation. The split and join algorithms are quite simple; again, thanks to access. It is known that these two operations are quite efficient in practice as well. Hence, several other operations are implemented in terms of split and join.

- Each of these operations takes $O(\lg n)$ amortized time.

  The amortized cost of join

  $=$ amortized cost of accessing the largest key x in T_1 $+$ amortized cost of linking $+$ change in potential

  $\leq 1 + 3\lg\frac{s(T_1)}{s(x)} + O(1) + (\lg(s(T_1) + s(T_2)) - \lg(s(T_1))$

  (only node whose size changes is x (that is, after accessing and making x as the root); for every other node y, potential due to y before joining is same as the potential due to y after joining - hence those potentials cancel out)

  $= 2\lg\frac{s(T_1)}{s(x)} + \lg(\frac{s(T_1)}{s(x)}\frac{s(T_1)+s(T_2)}{s(T_1)}) + O(1)$

  $= 2\lg\frac{s(T_1)}{s(x)} + \lg(\frac{s(T_1)+s(T_2)}{s(x)}) + O(1)$

  $\leq 3\lg(\frac{s(T_1)+s(T_2)}{s(x)}) + O(1).$

  The amortized costs of other operations can be upper bounded similarly.

- *Balance theorem*: The actual cost of m accesses with n items is $O((m+n)\lg n)$.

  Proof: By assigning weight of each item to $\frac{1}{n}$, the weight of root is 1, and the amortized cost of m accesses with n items $= O(m * (1 + \lg\frac{s(root)}{s(x)})) = O(m(1 + \lg\frac{1}{\frac{1}{n}})) = O(m\lg n)$.

  The actual cost of m accesses with n items $=$ amortized cost of m accesses with n items $-$ smallest ending potential $+$ maximum starting potential $\leq O(m\lg n) - \sum_{i=1}^{n}\lg\frac{1}{n} + \sum_{i=1}^{n}\lg 1$ [7] $= O((m+n)\lg n)$.

[5] every key in T_1 is less than every key in T_2

[6] the other way is to insert i as usual and splay on i

[7] due to Stirling's approximation, $\sum_{i=1}^{n}\lg\frac{1}{n} \geq -n\lg n$

References:

Self-Adjusting Binary Search Trees. D. E. Sleator and R. E. Tarjan. Journal of the ACM, 32(3): 652-686, 1985. (This note only covers pages 3-8 and 9-13 of this paper. Since this note is written for second-year undergrads, optimality aspects of splay trees are omitted from presenting.)