Splay trees are a powerful form of (lexicographic) balanced binary trees devised by Sleator and Tarjan [1]. Splay trees are self-adjusting, so that frequently-accessed items drift toward the root. Every time we access the tree, we reorganize it through a sequence of rotations—this organization will be expensive on a one-time basis, but analysis shows that the basic operations require $O(\log n)$ amortized time: we show that $m$ consecutive operations on a splay tree with $n$ nodes require amortized time $O((m+n) \log n + m)$. Moreover, splay trees do not require any extra information in each node, as is needed for red-black trees.

1 Splaying

The main operation in splay trees is the splay operation step. Splaying on a node $x$ moves $x$ to the root of the tree by a sequence of rotations that move it up two levels up at a time. If $x$ is an even number of levels from the root, then we use these two-level rotations to bring $x$ directly to the root. If $x$ is an odd number of levels from the root, then we use these rotations to bring it up to one level below the root, at which point we apply either a right rotation (called a ZIG) or a left-rotation (called a ZAG) to bring $x$ up to the node:

Thus, if our node $x$ is a left child we ZIG it to the root. Similarly, if it is a right child, we ZAG it from the right to the root.

To move the node $x$ two levels up the tree, consider its position relative to its grandparent. We apply either a ZIG-ZIG (that is, two ZIG’s in a row) to the appropriate subtree if $x$ is the leftmost grandchild, or a ZAG-ZAG if it is the rightmost grandchild.
Similarly, if \( x \) is the right child of a left parent, we apply a ZIG-ZAG:

That is, we first rotate left at \( y \) (the ZAG) and then rotate right at \( z \) (the ZIG).

If \( x \) is the left child of a right parent, we apply a ZAG-ZIG operation:

That is, we first rotate right at \( y \) (the ZIG) and then rotate left at \( z \) (the ZAG).

Using these operations, splaying on a node of depth \( d \) requires \( d \) rotations. We make a rotation our basic unit of time, as they can be implemented in constant time. Thus, we can say that we need time \( d \) to splay on a node at depth \( d \).

Consider the following example where we splay on the node 5. Note that the subtree rooted at 5 is constantly moving up the tree.

We see that the tree goes from being long and thin list to being shorter and bushier. This is the general behavior under splaying: paths in the tree tend shorten considerably. Play around with the following URL
to get an idea of the power of the splay operation to keep a tree balanced as it undergoes searches, insertions, and deletions:

http://www.ibr.cs.tu-bs.de/courses/ss98/audii/applets/BST/SplayTree-Example.html

2 Amortized Cost of Splaying

Since splaying is the primary operation, we now analyze its amortized cost. To do so, we define a potential function. Suppose we have a splay tree $T$. Then,

1. Assign each node $x$ in $T$ a positive weight $w(x)$.
2. Let the size of node $x$, denoted $S(x)$, be the sum of the weights of all the nodes in the subtree rooted at $x$.
3. Define the rank $r(x) = \log S(x)$.

Define the potential function of the tree $T$ to be

$$\Phi(T) = \sum_{\text{nodes } x \in T} r(x).$$

For example, let the weight function be $w(x) = 1$ for each (internal) node $x$ (weights of leaves—which are really null pointers—are 0), and consider the tree

```
      5
     / \   \\
   2   1
  /     \\
 4   3   1
  / \
3   4
```

The size of node 4, for example, is $w(4) + w(3) = 1 + 1 = 2$; thus, $r(4) = \log 2 = 1$. We can similarly compute the rank of each internal node to get the potential of the tree:

$$\Phi(T) = r(1) + r(2) + r(3) + r(4) + r(5)$$
$$= \log 1 + \log 3 + \log 1 + \log 2 + \log 5$$
$$= \log 30$$
$$\approx 4.91$$

We now use the tools developed during our study of amortized analysis. Recall that:

$$\text{AMORTIZED COST } = \text{ ACTUAL COST } + \text{ change in potential}$$

so that

$$\hat{c}_i = c_i + \Phi(T_i) - \Phi(T_{i-1})$$
and hence
\[ \sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(T_n) - \Phi(T_0). \]

Where \( T_i \) is the tree after the \( i \)th operation, \( \hat{c}_i \) is the amortized cost of the \( i \)th operation (that is, what you would charge a customer of our data-structure business) and \( c_i \) is the actual cost of that operation (that is, what you have to pay the graduate student to do the work). In other words, over a sequence of operations,

\[
\text{AMORTIZED COST (sequence)} = \text{ACTUAL COST (sequence)} + \Phi_{\text{end}} - \Phi_{\text{start}}
\]

or,

\[
\text{ACTUAL COST (sequence)} = \text{AMORTIZED COST (sequence)} - \Phi_{\text{end}} + \Phi_{\text{start}}. \tag{1}
\]

Because rotations take \( O(1) \) time, we define “time” to mean the number of rotations needed in an operation on a tree.

**Access Lemma**  The amortized time to splay at a node \( x \) in a tree with root \( t \) is at most

\[
3[r(t) - r(x)] + 1 = O \left( 1 + \log \frac{S(t)}{S(x)} \right).
\]

The proof of this lemma is by analyzing the various possible steps involved in splaying. Let \( r'(\cdot) \) be the rank of a node after the splay, and \( r(\cdot) \) be the rank of a node before the splay. We show that the amortized cost of any step of the splay operation is at most \( 3[r'(x) - r(x)] \), with exception of the one extra rotation needed at the root if \( x \) is initially at odd depth. Thus, when we perform several parts of a splay operation together, the \( 3[r'(x) - r(x)] \) terms telescope, giving an amortized cost of

\[
\sum \text{amortized costs} = 1 + 3[r'_\text{final}(x) - r'_\text{initial}(x)]
= 1 + 3[\lg S(\text{root}) - \lg S(x)]
= O \left( 1 + \log \frac{S(t)}{S(x)} \right),
\]

as claimed in the lemma. We now consider the various cases.

**Case 1: One rotation**  This case occurs as the last step when the splayed node is at an odd depth from the root.

The actual cost of one rotation is just one time unit (rotation). Thus, we are left to analyze the change potential. For the \text{ZIG} rotation

\[
\text{ZIG}
\]

the potentials of subtrees \( A, B, \) and \( C \) are unaffected by a \text{ZIG} because their internal node structure does not change. Thus we need be concerned only with the change in the ranks of \( x \) and \( y \). Denote by \( r(x) \) and
\(r(y)\) the ranks of \(x\) and \(y\), respectively, before the **ZIG**; denote by \(r'(x)\) and \(r'(y)\) the ranks of \(x\) and \(y\) after the **ZIG** operation.

The change in the potential function caused by a **ZIG** is

\[
\Delta \Phi = r'(x) + r'(y) - r(x) - r(y).
\]

Clearly, \(r'(y) \leq r(y)\) since \(y\) starts the **ZIG** overlooking subtrees \(A\), \(B\), and \(C\) and node \(x\), but ends the **ZIG** overlooking only subtrees \(B\) and \(C\). Thus, we can bound the potential difference by

\[
\Delta \Phi = r'(x) - r(x) + r'(y) - r(y) \leq r'(x) - r(x).
\]

Now we can bound the amortized cost, as per equation (2):

\[
\text{AMORTIZED COST} = \text{ACTUAL COST} + \Delta \Phi \\
= 1 + \Delta \Phi \\
\leq 1 + r'(x) - r(x) \\
\leq 1 + 3[r'(x) - r(x)].
\]

The last inequality is clearly a weak statement, since the 3 is unnecessary; nevertheless, this is the inequality needed for the telescoping mentioned above.

The amortized cost of a **ZAG** operation can be computed in the same way, with the appropriate relabelings.

**Case 2: ZIG-ZIG**  The actual cost of a **ZIG-ZIG** operation is two time units, the two rotations. We now need to compute the change in the potential function. Recall the definition of a **ZIG-ZIG**:

As in the previous case, the potentials of the subtrees \(A\), \(B\), \(C\), and \(D\) are unaffected by the operation. Thus, we have,

\[
\Delta \Phi = r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z).
\]

Now, to bring this \(\Delta \Phi\) to the desired form, we notice a few relationships. First, \(r'(x) = r'(z)\) because the rotated \(x\) is precisely in the same position as the old \(z\). Moreover, \(r(y) \geq r(x)\) because \(y\) overlooks \(x\) in the original tree. Similarly, \(r'(x) \geq r'(y)\).

Thus, (2) becomes

\[
\Delta \Phi \leq r'(x) + r'(z) - 2r(x),
\]

giving an amortized cost of

\[
\text{AMORTIZED COST} \leq 2 + r'(x) + r'(z) - 2r(x)
\]
We want to show that

\[
\text{AMORTIZED COST} \leq 3[r'(x) - r(x)],
\]

which would follow from

\[
2 + r'(x) + r'(z) - 2r(x) \leq 3[r'(x) - r(x)],
\]

or

\[
-2r'(x) + r(x) + r'(z) + 2 \leq 0;
\]

that is, proving that

\[
-2r'(x) + r(x) + r'(z) \leq -2
\]

will give the claimed bound. Let us analyze the lefthand side of this last inequality

\[
-2r'(x) + r(x) + r'(z) = -r'(x) + r(x) - r'(x) + r'(z)
\]

\[
= -\log \frac{S'(x)}{S(x)} - \log \frac{S'(x)}{S'(z)}
\]

\[
= \log \frac{S(x)}{S'(x)} + \log \frac{S'(z)}{S'(x)},
\]

(3)

where \(S'(\cdot)\) is the size of a node after the \textsc{zig-zig} operation, and \(S(\cdot)\) is the size of a node before the \textsc{zig-zig} operation.

Define

\[
a = \frac{S(x)}{S'(x)} \quad \text{and} \quad b = \frac{S'(z)}{S'(x)},
\]

so that (3) becomes

\[
\lg a + \lg b.
\]

Clearly \(a > 0\) and \(b > 0\). Moreover, \(S(x) + S'(z) \leq S'(x)\) because before the \textsc{zig-zig} \(x\) overlooks subtrees \(A\) and \(B\), and \(z'\) overlooks subtrees \(C\) and \(D\), but after the \textsc{zig-zig} \(x'\) overlooks all subtrees \(A\), \(B\), \(C\), and \(D\), in addition to \(y'\) and \(z'\). Thus,

\[
\frac{S(x)}{S'(x)} + \frac{S'(z)}{S'(x)} \leq 1
\]

(“less than” is possible because the weight of \(y\) is not included), and hence

\[
a + b \leq 1.
\]

Thus, we have \(a > 0\), \(b > 0\), and \(a + b \leq 1\). Using the convexity of the logarithm, elementary calculus tells us that in the region of interest \(\lg a + \lg b\) reaches a maximal value of \(-2\) for \(a = b = 1/2\). Thus,

\[
\lg a + \lg b \leq -2.
\]

Substituting back we get

\[
-2r'(x) + r(x) + r'(z) \leq -2,
\]

which is what we needed to show, so

\[
\text{AMORTIZED COST} \leq 3[r'(x) - r(x)]
\]

for the \textsc{zig-zig} operation, as we claimed.

By appropriate relabeling, we see that the \textsc{zag-zag} operation has the same amortized time.
Case 3: ZIG-ZAG This case is similar in fashion to the case of ZIG-ZIG that we just analyzed: First, recall the definition of a ZIG-ZAG:

\[
\begin{align*}
\text{ZIG-ZAG} & \\
\end{align*}
\]

As in the previous case, the actual cost of a ZIG-ZAG is 2 rotations, and we have to compute the change in the potential function,

\[
\Delta \Phi = r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z).
\]

Again we note that \(r'(x) = r(z)\). Furthermore, \(r(x) \leq r(y)\) because \(z\) is above \(x\) in the original tree. Thus we have

\[
\text{AMORTIZED COST} = 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\
\leq 2 + r'(y) + r'(z) - 2r(x).
\]

As in the previous case, we wish to show that

\[
\text{AMORTIZED COST} \leq 3[r'(x) - r(x)];
\]

so we will show that

\[
2 + r'(y) + r'(z) - 2r(x) \leq 2[r'(x) - r(x)],
\]

which is less than or equal to our desired bound of \(3[r'(x) - r(x)]\). The last inequality can be rearranged to

\[
2r'(x) - r'(y) - r'(z) \geq 2.
\]

As in the previous case, we know that \(S'(y) + S'(z) \leq S'(x)\), so that

\[
\frac{S'(y)}{S'(x)} + \frac{S'(z)}{S'(x)} \leq 1.
\]

Define

\[
a = \frac{S'(y)}{S'(x)} \quad \text{and} \quad b = \frac{S'(z)}{S'(x)}
\]

and again we find \(\log a + \log b \leq -2\) so that

\[
2r'(x) - r'(y) - r'(z) \geq 2,
\]

and the stated bound on the amortized cost of the ZIG-ZAG step follows.

The ZAG-ZIG case follows in exactly the same fashion, proving the Access Lemma. Note that the double-rotation steps are necessary in this calculation, because they do not carry the +1 term that the single-rotation amortized costs carry. This +1 term would destroy the telescoping.
3 Balance Theorem

We can now determine the actual time needed for multiple accesses to the tree.

**Balance Theorem** The total access time for \(m\) accesses of a tree with \(n\) items is:

\[
O((m + n) \log n + m).
\]

The proof of this theorem follows from our potential function and the Access Lemma, using (1).

Define the weight of a node \(x\) to be \(w(x) = 1/n\). From the Access Lemma, we know that the amortized cost for the \(m\) accesses is

\[
m \times O \left( 1 + \log \frac{S(t)}{S(x)} \right) = m \times O \left( 1 + \log \frac{1}{1/n} \right) = m \times O(1 + \log n).
\]

The first equality follows from the fact that \(t\) overlooks all the other nodes, so its weight is \(n \times 1/n = 1\).

The greatest possible starting potential is

\[
\Phi_{\text{start}} \leq \sum_{i=1}^{n} \log 1 = 0,
\]

because no vertex can have size greater than 1 (the total size of the whole tree is 1). On the other hand, the smallest ending potential is

\[
\Phi_{\text{end}} \geq \sum_{i=1}^{n} \log \frac{1}{n} \geq -n \log n
\]

because, at worst, every vertex has its own weight of \(1/n\); the second inequality comes from Stirling’s approximation.

Putting all of the pieces together,

\[
\text{ACTUAL COST} \leq m \times O(1 + \log n) + 0 - (-n \log n) \leq O((m + n) \log n + m),
\]

as the theorem states.

4 Static Optimality Theorem

We can refine the Balance Theorem if we know something about the access frequencies of the tree nodes.

**Static Optimality Theorem** If each item is accessed at least once, the total time for \(m\) accesses in a tree with \(n\) nodes is

\[
O \left( m + \sum_{i=1}^{n} q_i \log \frac{m}{q_i} \right),
\]

where \(q_i\) is the number of times item \(i\) is accessed so that \(\sum_{i=1}^{n} q_i = m\).
The proof follows as in the Balance Theorem, but using the weight function

\[ w(i) = \frac{q_i}{m} \]

(so the sum of all weights is 1) and observing that the amortized cost of an access of item \( i \) is \( O(1 + \log \frac{S(i)}{S(T)}) = O(1 + \log \frac{m}{q}) \).

The Static Optimality Theorem is amazing because it is within a constant multiple of the entropy—the information-theoretic lower bound on access time for a binary tree! (This is also true for the Balance Theorem.) Thus splay trees are within a constant multiple of the lower bound on the problem. Moreover, they are as good as finger trees (trees that keep “fingers” pointing to the most frequently-occurring items).

## 5 Operations on Splay Trees

With the splay operation and its analysis, it is not too difficult to implement and analyze operations such as search, insert, and delete on splay trees. First, however, we introduce three new tree operations.

**Access**  Access takes an item \( i \) as input, runs a search on the tree to find the node containing \( i \), then splays on that node, moving it to the top of the tree. If no such item is found, we splay on the last non-null node that we examined in the binary search; that is, we splay at either \( i^- \) or \( i^+ \), the predecessor or successor of \( i \), respectively.

**Join**  The join operation takes two trees, \( T_1 \) and \( T_2 \), for which every item in \( T_2 \) is greater than every item in \( T_1 \), and returns a single tree \( T \) containing the items of both trees. Implementing join on splay trees requires accessing the largest item, which we denote by \( i \), in \( T_1 \) by following non-null right pointers from the root, followed by a splay at the last node found, which is \( i \), the largest item in the tree. The splay puts \( i \) at the root of \( T_1 \) and, as the largest element, must have a null right subtree, which we replace with \( T_2 \):

\[
\begin{array}{c}
T_1 \\
\downarrow \\
T_2
\end{array} \quad \xrightarrow{\text{join}} \quad 
\begin{array}{c}
\text{i} \\
\downarrow \\
T_1 - i \\
\downarrow \\
T_2
\end{array}
\]

**Split**  This is the reverse of join. It takes a tree \( T \) and a node \( i \) as input, and creates two trees \( T_1 \) and \( T_2 \) such that all items in \( T_1 \) are smaller than (or equal to) \( i \) and all items in \( T_2 \) are greater than (or equal to) \( i \). Implementing split on splay trees involves accessing \( i \), and then breaking one of the root’s branches, depending on whether the root is greater or less than \( i \) (arbitrarily selecting one if \( i \) is the root).


If \( i \) is not in the tree, the root after the splay at \( i \) is either \( i^- \) or \( i^+ \), the lexicographic predecessor or successor of \( i \), respectively.

Now it is easy to implement the familiar operations of insert and delete:

**Insert** Takes a tree \( T \) and an item \( i \) (presumed not in the tree) as an input and inserts a node containing \( i \) into \( T \). To perform an insert, simply \( \text{split}(T, i) \) and then make a new tree whose left and right branches are the trees \( T_1 \) and \( T_2 \) returned from \( \text{split} \) and whose node contains the item \( i \).

**Delete** Takes a tree \( T \) and an item \( i \) in the tree and deletes \( i \) from \( T \). To perform delete, we again do a \( \text{split}(T, i) \). Then remove \( i \) and \( \text{join} \) the resulting subtrees.

All these operations have a logarithmic amortized time bound. Specifically, the following table gives the amortized times for the splay tree operations as a function of \( W \), the total weight of the items in the tree(s). The variables \( i^- \) and \( i^+ \) denote, respectively, the successor and predecessor of \( i \) in the tree. If \( i^- \) or \( i^+ \) is undefined, then \( w(i^-) = \infty \) and \( w(i^+) = \infty \), respectively.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Amortized Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>access(i, T)</td>
<td>$3 \lg \left( \frac{W}{w(i)} \right) + 1$ if i is in T</td>
</tr>
<tr>
<td>access(i, T)</td>
<td>$3 \lg \left( \frac{W}{\min{w(i-1), w(i+1)}} \right) + 1$ if i is not in T</td>
</tr>
<tr>
<td>join(T_1, T_2)</td>
<td>$3 \lg \left( \frac{W}{w(i)} \right) + O(1)$ where i is the last item in T_1</td>
</tr>
<tr>
<td>split(i, T)</td>
<td>$3 \lg \left( \frac{W}{w(i)} \right) + O(1)$ if i is in T</td>
</tr>
<tr>
<td></td>
<td>$3 \lg \left( \frac{W}{\min{w(i-1), w(i+1)}} \right) + O(1)$ if i is not in T</td>
</tr>
<tr>
<td>insert(i, T)</td>
<td>$3 \lg \left( \frac{W - w(i)}{\min{w(i-1), w(i+1)}} \right) + \lg \left( \frac{W}{w(i)} \right) + O(1)$</td>
</tr>
<tr>
<td>delete(i, T)</td>
<td>$3 \lg \left( \frac{W - w(i)}{w(i)} \right) + 3 \lg \left( \frac{W}{w(i)} \right) + O(1)$</td>
</tr>
</tbody>
</table>

The bounds for `access` and `split` follow directly from the Access Lemma. The other bounds follow by analyzing the change in potential. For example, the bound on `join` is found as follows: first we do an `access` on the largest value in T_1; this costs at most $3 \lg(S(T_1)/w(i)) + 1$ amortized time. The link requires only O(1) additional work, but linking trees $T_1$ and $T_2$ increases the potential so we also have to examine that. The only node whose weight changes is the root of $T_1$, which becomes the root of the entire tree. Thus, because the total weight is $W = S(T_1) + S(T_2)$, the change in potential of the new root $i$ is at most

$$\lg W - \lg S(T_1) = \lg \left( \frac{W}{S(T_1)} \right).$$

Combining these terms gives the desired bound:

$$3 \lg \left( \frac{S(T_1)}{w(i)} \right) + O(1) + \lg \left( \frac{W}{S(T_1)} \right) = 2 \lg \left( \frac{S(T_1)}{w(i)} \right) + \lg \left( \frac{S(T_1)}{w(i)} \cdot \frac{W}{S(T_1)} \right) + O(1)$$

$$= 2 \lg \left( \frac{S(T_1)}{w(i)} \right) + \lg \left( \frac{W}{w(i)} \right) + O(1)$$

$$\leq 3 \lg \left( \frac{W}{w(i)} \right) + O(1)$$

The bounds for `insert` and `delete` are proven in a similar manner.

Be warned: The constants hidden in the $O$ notation are large for this data structure, so it may not be practical in real life.

Reference