8.1 Introduction

Today we’re going to talk even more about binary search trees. B-trees, red-black trees, AVL trees, etc., go to a lot of effort to keep the tree balanced (or approximately balanced), making inserts pretty complicated. Today, we’re going to talk about an advanced (and amazing) binary search tree known as a splay tree, invented by Sleator and Tarjan. These are sometimes called “self-adjusting” binary search trees, since they do two things very differently from other balanced search trees: they don’t do anything to explicitly enforce balance, and they change the tree on lookups as well as on inserts. As we’ll see, the worst-case performance of splay trees might not be very good, but they have amazing amortized properties: any sequence of operations is actually very cheap.

8.2 Splay Trees

Splay trees take a different approach, and provide what is (in some sense) a weaker bound. We are only going to get $O(\log n)$-amortized time bounds. Some lookup queries might actually take a long time (even $\Omega(n)$) to complete. In return, we will get a much simpler algorithm with much less to keep track of, as well as a number of nice properties which we won’t really have time to talk about. Informally, though, it turns out that for splay trees you can prove the Static Optimality Theorem: if you compare the cost of doing $m$ queries on any fixed tree (such as a red-black tree or AVL tree after we have finished all inserts), then the splay tree is (essentially) optimal. This is true even if we know the sequence of lookups ahead of time, and can tailor our search tree exactly to the sequence of lookups! Splay trees manage this despite not knowing the queries in advance.

8.2.1 Tree rotations

Tree rotations are a fundamental building block in most binary search trees, including splay trees. It’s an operation which (in constant time) allows us to move a node one level higher in the tree, while still ensuring the search tree property by rearranging the tree structure appropriately. By repeatedly rotating the same node, we can eventually move it up to the root. This turns out to be a useful ability.

A basic rotation works as in the following figure. This is also called rotating on $h$, or Rotate($h$), since it moves $h$ up one level.
Clearly after we do a rotate we still have a search tree, and now $h$ is one level closer to the root (the parent of $q$ pre-rotation is the parent of $h$ post-rotation). I haven’t drawn it, but clearly there’s an equivalent rotation operation when $w$ starts out as the right child of $h$.

### 8.2.2 Splay Tree operations

Splay trees are usually described in terms of three basic operations that extend simple rotations: the zig operation, the zig-zag operation, and the zig-zig operation. Note that a basic rotation of a node only considers the parent and children. For a splay tree, we also need to consider the **grandparent**. The two major operations (zig-zig and zig-zag) move a node up *two* levels at a time, and then a zig is a one-level operation that is necessary in case the height is odd.

**Zig:** The zig operation is the simplest: it is just a simple rotation. We only use it when there is no grandparent, i.e. the parent of the node that we’re at is the root.
**Zig-Zag:** The zig-zag operation is also pretty simple: it’s just two rotations (Rotate($q$) followed by Rotate($q$) again). We only do it when the direction of the edge from the grandparent to the parent is different from the direction of the edge from the parent to the node. So there are two settings when we can use a zig-zag: if the node is the right child of the parent and the parent is the left child of the grandparent, or if the node is the left child of the parent and the parent is the right child of the grandparent. In this case, we can essentially do two rotations so that the node takes the place of the grandparent.

**Zig-Zig:** This is the operation which makes splay trees different from just simple rotations. There are two cases which are not covered by zig-zag: if the node is the left child of the parent and the parent is the left child of the grandparent, or if the node is the right child of the parent and the parent is the right child of the grandparent. In these cases, instead of doing two rotations on $h$, we rotate on $q$ and then rotate on $h$. This is called a zig-zig operation, and changes the tree as in the following figure.
8.2.3 Splay Tree algorithm

With these three operations in hand, it’s easy to define the splay tree algorithm. The combination of these three operations is called a splay. That is, we say that we “splay” a node $u$ if we first check which of the three situations it is in, and then apply the appropriate operation (zig, zig-zag, or zig-zig). So instead of rotating a node up to the root, in a splay tree we splay it up to the root.

On a Lookup query, we first walk down the tree as in every binary search tree to find the key. But once we have it, instead of returning it, we first splay it to the root, and then return it. Thus unlike trees you might be used to, in a splay tree a Lookup operation actually changes the structure of the tree.

An Insert operation is done the same way: we walk down the tree to figure out where to insert it, then insert it as a new leaf, and then splay it to the root.
8.3 Splay Tree Analysis

Note that a single operation might take a long time: the tree could get extremely unbalanced if we have a particularly bad sequence of queries, in which case a single operation could take \( \Omega(n) \) time (see the homework!). The amazing thing is that this cannot happen very often: the amortized complexity of a Lookup or an Insert is only \( O(\log n) \).

We will think of a single splay operation as having cost 1. This means that in the end, we'll get amortized bounds on the number of splay operations. Since each splay operation takes a constant amount of time, this will give us bounds on the running time (we're ignoring the cost of walking down the tree on a find or insert, since whenever we walk down we splay up the same amount, and so the adding in the walk down would (at most) double the running time).

We first need a few definitions. In the following, \( T \) is a (splay) tree, \( u \) is an arbitrary node in \( T \), and \( p \) is the parent of \( u \) and \( g \) is the grandparent of \( u \).

- Let \( s(u) \) (called the size of \( u \)) be the number of nodes in the subtree rooted at \( u \) (including \( u \) itself).
- Let \( r(u) = \lfloor \log(s(u)) \rfloor \) (called the rank of \( u \) – this is different from the rank we used in the homework).
- Let \( \Phi(T) = \sum_{u \in T} r(u) \). This is the potential function that we will use.

Let’s start by noticing some easy properties of ranks.

1. Doing a rotate on \( u \) affects the ranks of only \( u \) and \( p \) (the parent of \( u \)), and the rank of \( u \) after the rotation is equal to the rank of \( p \) before the rotation.

2. If two siblings both have rank \( i \), then the parent has rank \( i + 1 \). To see this, let \( u \) and \( v \) be siblings of rank \( r \) with parent \( p \). Then by the definition of rank, \( 2^i \leq s(u) < 2^{i+1} \) and \( 2^i \leq s(v) < 2^{i+1} \), and hence (when we include \( p \) in \( s(p) \)) we get that \( 2^{i+1} + 1 \leq s(p) < 2^{i+2} \). This implies that \( r(p) = i + 1 \).

3. Suppose that a node \( u \) and its parent \( p \) both have rank \( i \). Then \( r \), the other child of \( p \), has rank less than \( i \). Again, this is just by figuring out the sizes: if \( v \) also had rank \( i \), then we would have that \( s(p) \geq s(u) + s(v) \geq 2^i + 2^i = 2^{i+1} \) and so \( p \) would have rank \( i + 1 \).

Now let’s analyze the change in potential cause by each of the three splay operations. Let \( r \) be the rank function before the operation, and let \( r' \) be the rank function after.

**Lemma 8.3.1** In a zig operation, \( \Delta \Phi \leq r'(u) - r(u) \leq 3(r'(u) - r(u)) \).

**Proof:** Only \( p \) and \( u \) change rank, so by definition \( \Delta \Phi = r'(p) - r(p) + r'(u) - r(u) \). By our first property of ranks, we know that \( r'(u) = r(p) \), so we get that \( \Delta \Phi = r'(p) - r(u) \leq r'(u) - r(u) \).

**Lemma 8.3.2** In a zig-zag or zig-zig operation, \( \Delta \Phi \leq 3(r'(u) - r(u)) - 1 \)

**Proof:** Note that only \( g \), \( p \), and \( u \) change ranks, so \( \Delta \Phi = r'(g) - r(u) + r'(p) - r(p) + r'(u) - r(u) \). We analyze the two operations separately.

5
• Consider a zig-zag operation. We split into two cases, depending on the initial ranks.

First, suppose that \( r(g) = r(u) = r \) (and thus also \( r(p) = r \)). By our first property of ranks, we know that \( r'(u) = r \). So by the last two properties of ranks, either \( r'(p) \) or \( r'(g) \) is strictly less than \( r \). Thus \( \Delta \Phi \leq -1 = 3(r'(u) - r(u)) - 1 \).

In the second case, suppose that \( r(g) > r(u) \). Since \( r(g) = r'(u) \), we know that \( \Delta \Phi = r'(g) + r'(p) - r(p) - r(u) \). Now by the last two properties of ranks, we know that \( r'(g) + r'(p) \leq 2r'(u) - 1 \), and we also know that \( r(p) \geq r(u) \). Hence \( \Delta \Phi \leq 2r'(u) - 1 - 2r(u) = 2(r'(u) - r(u)) - 1 \leq 3(r'(u) - r(u)) - 1 \).

• Now consider a zig-zig operation. We again break into the same two cases.

In the first case, suppose that \( r(g) = r'(u) = r \) (and thus \( r(p) = r \) also). Recall that when we do a zig-zig, we first rotate \( p \) and then rotate \( u \). After rotating \( p \), the rank of \( p \) and the rank of \( u \) are still both \( r \), but \( g \) is now a child of \( p \) and thus its rank must be strictly less than \( r \). Now when we rotate \( u \) the rank of \( u \) becomes \( r \), the rank of \( p \) becomes at most \( r \), and the rank of \( g \) is still strictly less than \( r \). Hence \( \Delta \Phi \leq -1 = 3(r'(u) - r(u)) - 1 \).

In the second case, suppose that \( r(g) > r'(u) \). As always, \( r'(u) \) and \( r(g) \) cancel out and so \( \Delta \Phi = r'(g) + r'(p) - r(p) - r(u) \). Now we know that \( r'(g) + r'(p) \leq 2r'(u) \), and also that \( r(p) \geq r(u) \), so we get that \( \Delta \Phi \leq 2r'(u) - 2r(u) = 2(r'(u) - r(u)) \). Since \( r'(u) - r(u) \geq 1 \), we can conclude that \( \Delta \Phi \leq 3(r'(u) - r(u)) - 1 \).

So in every case, \( \Delta \Phi \leq 3(r'(u) - r(u)) - 1 \).

We can now prove the main lemma.

**Lemma 8.3.3** The amortized cost of splaying a node to the root is \( O(\log n) \)

**Proof:** We just need to bound the amortized cost of splaying an arbitrary node \( u \) up to the root of the tree. This will consist of a series of zig-zig or zig-zag operations, and then possibly one zig operation. Let \( g_1 \) be the grandparent of \( u \), let \( g_2 \) be the grandparent of \( g_1 \), etc., until we get a node \( g_k \) which is either the root or a child of the root. We will assume that \( g_k \) is a child of the root, since that is the more difficult case. So in total we will do \( k \) ZZ operations and one zig operation.

Let \( r_i \) be the rank function after we have done \( i \) splay operations, and let \( \Phi_i = \sum_{v \in T} r_i(v) \) be the potential after \( i \) splay operations. Then the total amortized cost is

\[
\sum_{i=1}^{k+1} (1 + \Phi_i - \Phi_{i-1}) \leq \sum_{i=1}^{k} (1 + 3(r_i(u) - r_{i-1}(u)) - 1) + (1 + 3(r_{k+1}(u) - r_k(u)))
\]

\[
\leq \sum_{i=1}^{k+1} 3(r_i(u) - r_{i-1}(u)) + 1
\]

\[
= 3(r_{k+1}(u) - r_0(u)) + 1
\]

\[
\leq 3 \log n + 1
\]
Now we’re essentially done! On a Find or an Insert, the time is essentially (up to a constant factor for the walk down) equal to the cost of splaying a node up to the root, and hence is at most $O(\log n)$. Slightly more formally, we get the following corollary.

**Theorem 8.3.4** The running time of doing $m$ operations on a splay tree with at most $n$ nodes is $O(m \log n + n \log n)$.

**Proof:** As we saw before when doing amortized analysis, the actual running time of doing a sequence of operations is equal to their amortized running time plus the initial potential minus the final potential. Since the final potential is at least 0 and the initial potential is at most $n \log n$, this means that the actual running time is $O(m \log n + n \log n)$.

## 8.3.1 More results

It turns out that splay trees have other, very appealing properties. I’m only going to discuss these informally, but there’s a lot more on the internet. If you’re interested, do some googling!

**Static Optimality:** Suppose that we want a binary search tree for a specific access sequence. Then we clearly will use our knowledge of this sequence to make a better (fixed) tree – we can put the most accessed elements towards the top of the tree, for example. Informally, as long as we do at least $n$ Finds, it turns out that splay trees are self-optimizing: they perform at least as well (up to a constant factor) as the best fixed tree.

**Working Set:** Suppose that we want to access item $x$, and let $k(x)$ be the number of distinct items that we have accessed since the last time we accessed $x$. Then the amortized time to access $x$ is only $O(1 + \log(k(x)))$. So if we have a small “working set” (number of items that we access regularly), then the cost of each access is actually less than $O(\log n)$.

**Dynamic Optimality Conjecture:** This is only a conjecture at this point, not a theorem. The conjecture is that splay trees are, up to a constant factor, as good as any other dynamic tree on every single access sequence. Here by a dynamic tree we mean a tree that is allowed to change through rotations. So for any access sequence, not only are splay trees as good as the best fixed tree, the conjecture is that they are as good as the best dynamic tree.