CMSC 420: Lecture 12
Splay Trees

Recap: We have discussed a number of different search structures for performing the basic ordered-dictionary operations (insert, delete, and find). Here is a brief review:

(Standard) Binary Search Trees: Very simple, but no effort is made to balance the tree. Height is $O(\log n)$ if keys are inserted in random order, but can be as bad as $\Omega(n)$.

AVL Trees: These are height-balanced trees. Height is guaranteed to be $O(\log n)$, and all dictionary operations run in $O(\log n)$ time. Each node stores height information (or the balance factor), and the tree is rebalanced by rotations.

2-3, Red-Black, and AA Trees: 2-3 trees were based on the notion of having a variable-width node, but the tree was perfectly balanced. Red-black and AA trees were binary tree encodings of 2-3 trees. For all three, the height is guaranteed to be $O(\log n)$, and all dictionary operations run in $O(\log n)$ time.

Treap and Skip Lists: These are randomized structures, which means that they rely on a random-number generator to determine their structures. All operations run in $O(\log n)$ time in expectation over the random choices. (The user's actions cannot impact the performance.) The skip list has expected space $O(n)$, but its worst case space is $O(n \log n)$.

Beyond Dictionary Operations: There are many other operations, beyond the standard ordered-dictionary operations (insert, delete, find) that a user might want the data structure to support. Here are a few examples:

Order-statistic queries: Given an integer k, where $1 \leq k \leq n$, find the kth smallest element in the set (see Fig. 1).

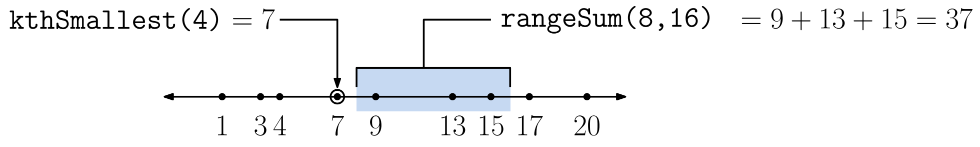

Fig. 1: Order-statistic and range queries.

Range queries: Given two keys x_0 and x_1, report (or generally compute some associative operation like sum or product over) all the values of the dictionary whose keys lie between x_0 and x_1 (see Fig. 1).

Split/Merge: Given a search tree T and a key x, *split* T into two search trees T_1 and T_2 such that all the keys of T_1 are $\leq x$ and all the keys of T_2 are $> x$ (see Fig. 2). Conversely, given two search trees T_1 and T_2 such that every key of T_1 is smaller than every key of T_2, *merge* them into a single search tree. The target is to solve both problems in $O(\log n)$ time.

Finger-Search Queries: We have just performed a find and located the node containing a key x. Now, we want to access a key y that is close to x in order. (E.g., you have a pointer to "Brenda" and now you want to find "Brandon.") Can this be done faster than restarting the search from the root?

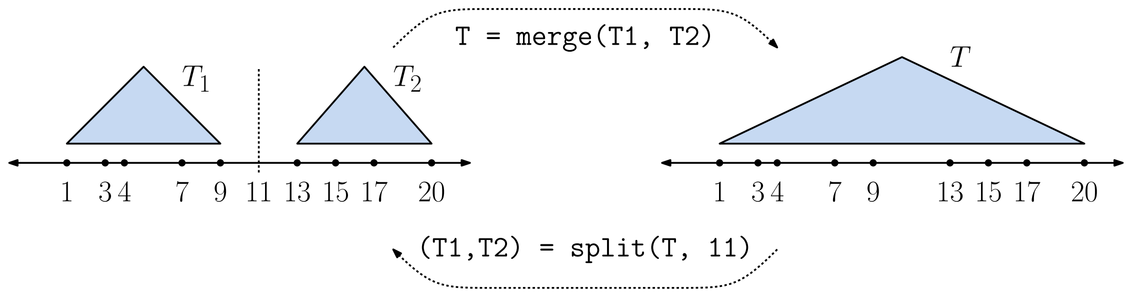

Fig. 2: Splitting and merging.

Beyond Worst-Case Complexity: In the above data structures, our analysis was based on worst-case performance. In practice, some queries are more likely than others. If so, *expected-case performance* may be more appropriate.

Expected-Case Optimal Search Tree: You are given n keys, $\{x_1, \ldots, x_n\}$. Let p_i denote the probability of accessing key x_i. Thus, $0 \leq p_i \leq 1$ and $\sum_{i=1}^{n} p_i = 1$. Suppose a binary search tree T stores x_i in a node at depth d_i from the root. The *expected search time* for this tree is $E(T) = \sum_{i=1}^{n} p_i d_i$.

Given the p_i's can we compute the binary search tree T that minimizes $E(T)$?

The structures we have seen so far do not address this question. There does exist an efficient algorithm for computing the optimum binary-search tree. (The answer is yes, and it involves an interesting exercise in dynamic programming.)

However, in order to compute this data structure, we assume that we *know* what the access probabilities. What if they are not known? What if they are known at some time, but they change over time? In such a dynamic setting, a better solution would be a *self-adjusting tree*. This is a tree that dynamically adjusts its structure according to a dynamically changing set of access probabilities. Intuitively, keys that are frequently accessed will filter up near the root, and keys that are rarely accessed will slowly fall to the deeper levels of the search tree.

Achieving this goal in a manner that can be made theoretically rigorous is a challenging problem. It was solved by Danniel Sleator and Robert Tarjan in 1985 by a data structure, called a *splay tree*. (This is the same Tarjan of Fibonacci heaps and depth-first search.) The splay tree itself is an amazing idea. While the tree is provably (theoretically) optimal with respect to a number of different criteria, the efficiency comes at a cost. Individual operations may take a long time, and efficiency is in the *amortized* sense.

Splay Trees and Amortization: All the balanced binary tree structures we have seen so far have two things in common: (1) they use rotations to maintain structure and (2) each node stores additional information to allow the tree to maintain balance. A *splay tree* is a binary search tree, and it uses rotations to maintain its structure, but unlike the others no additional storage is needed for balance information. (Thus, each node is just a node of a standard binary search tree. It stores a key, value, left child, and right child. That is all!)

Because a splay tree has no balance information, it is possible for the tree to become very unbalanced. Splay trees are remarkable in that they are *self-adjusting*. Having nodes that are great depth in a binary tree is not a problem *per se*, until such an element is accessed. Splay trees employ a clever trick so that whenever a very deep node is accessed, the tree will restructure itself so that the tree becomes significantly more balanced. *The splay tree maintains itself in balance (on average), but it has no idea whether it is balanced or not!*

This means that, as with standard binary search trees, it is possible that a single access operation could take as long as $\Omega(n)$ time (and not the $O(\log n)$ that we would like). However, splay trees are efficient in the amortized sense:

Splay Tree Performance Bound: Starting with an empty tree, the total time needed to perform any sequence of m insert/delete/find operations on a splay tree is $O(m \log n)$, where n is the maximum number of nodes in the tree.

Thus, although any individual operation may be quite costly (as high as $\Omega(n)$ time), the average cost of any operation is at most $O(\log n)$. As we have seen before, such an analysis (averaging over a sequence of operations) is called an *amortized analysis*. In the business world, amortization refers to the process of paying off a large payment over time in small installments. Here we are paying for the total running time of the data structure's algorithms over a sequence of operations in small installments. Even though each individual operation may be costly, the overall average is small.

As mentioned above, splay trees tend to bring frequently accessed keys up near the root. Indeed, it can be shown that if the access probabilities are stable, the cost of splay-tree operations asymptotically matches the performance of an optimal binary search tree.

No balance information, optimal expected search times, self-adjusting behavior? Splay trees sound amazing—and they are. However, they are not used that often in practice. In spite of their cool properties, they tend to be slower in practice than red-black trees, treaps, and skip lists. So, these other structures tend to be used more often than splay trees.

Splaying: The key to any self-adjusting data structure is the operation that incrementally modifies the organization of objects in the structure. In the case of a splay tree, this operation is called *splaying*. Given a key value x and a splay tree T the operation `T.splay(x)` searches for the key x within T, and reorganizes T while rotating the node with key x up to the root of the tree. If x is not in the tree, either the inorder predecessor or inorder successor of x will be brought to the root instead.

Here is how `T.splay(x)` works. We start with the normal binary search descent from the root of T to find the node p containing key x, or the last node visited before we fall out of the tree. (Observe that in the latter case, p is either the inorder successor or predecessor of x, depending on whether we fell out along a `null` left child link or a `null` right child link. Our objective is to bring the node p up to the root.

An idea that doesn't work: At this point you may see an obvious strategy to bring p to the root. We walk up the tree to p's ancestors, applying a rotation at each. While this will satisfy one of our requirements of moving p to the root, it will not do a good job of reorganizing the tree. To see why, suppose that we attempt to apply rotations to node a in Fig. 3. Observe that while a is brought up to the root, the tree is still very skewed and unbalanced.

A better idea: The poor performance of the single-rotation method suggests that we try something that "stirs things up" a bit more. Our next idea is to go two nodes at a time and apply rotations at each of these nodes. For example, in Fig. 4, we see that by applying two rotations at a time, first at the grandparent and then at the parent has

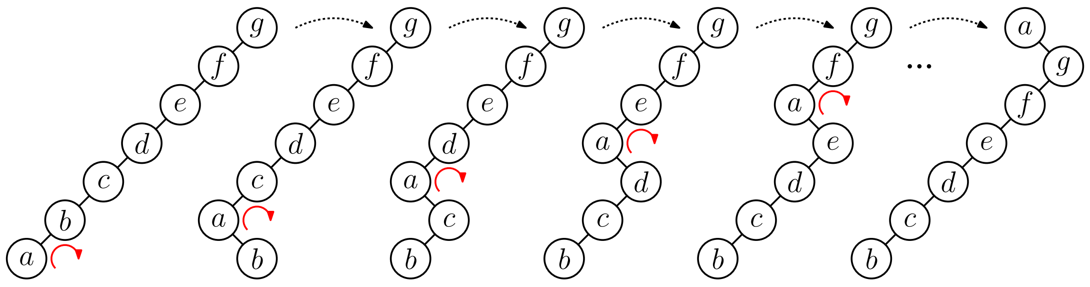

Fig. 3: Single rotations up to the root—the tree is still poorly balanced.

a dramatically better result on the tree height, cutting it roughly in half. (But will it work general?)

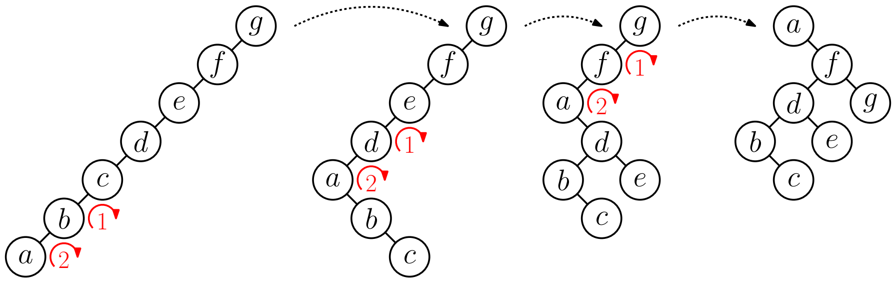

Fig. 4: Two-at-a-time rotations up to the root—the tree height is cut almost in half.

The above exercise suggests that we work two levels at a time. Here is a more formal description of the splay operation at a single node p of the tree:

- If p has both a parent and grandparent, q and r be the parent and grandparent, respectively. We consider two cases:
  - **Zig-zig:** If p and q are both right children or both left children, we apply a rotation at r followed by a rotation at q, to bring p to the top of this 3-node ensemble (see Fig. 5(a)), and continue up the tree.

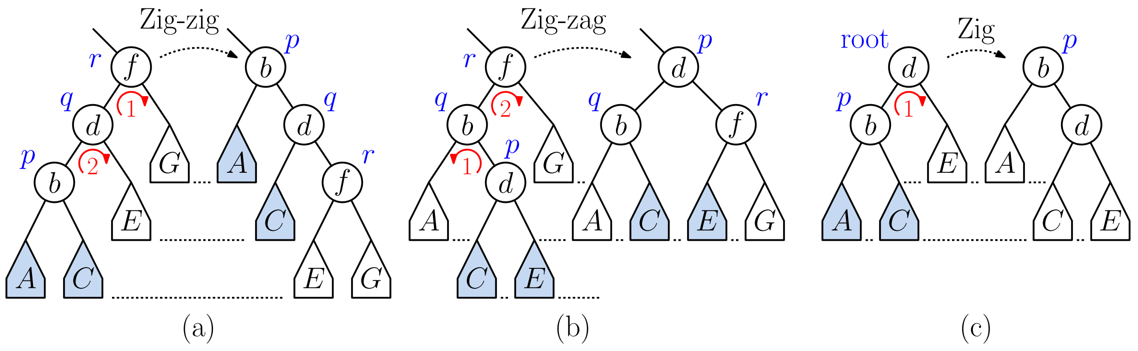

Fig. 5: Splaying cases: (a) Zig-Zig, (b) Zig-Zag and (c) Zig.

  - **Zig-zag:** If p and q are left-right or right-left children, we apply a rotation at q followed by a rotation at r, to bring p to the top of this 3-node ensemble (see Fig. 5(b)), and continue up the tree.

- **Zig:** If p is the child of the root, we do a single rotation at the root of T, making p the new root (see Fig. 5(a)), and are now done.
- If p is the root of T, we are done.

A full example is shown in Fig. 6. Note that the tree's inorder structure is preserved. Also observe that nodes lying on or near the search path to p (such as 1) tend to be lifted much closer to the root through this operation.

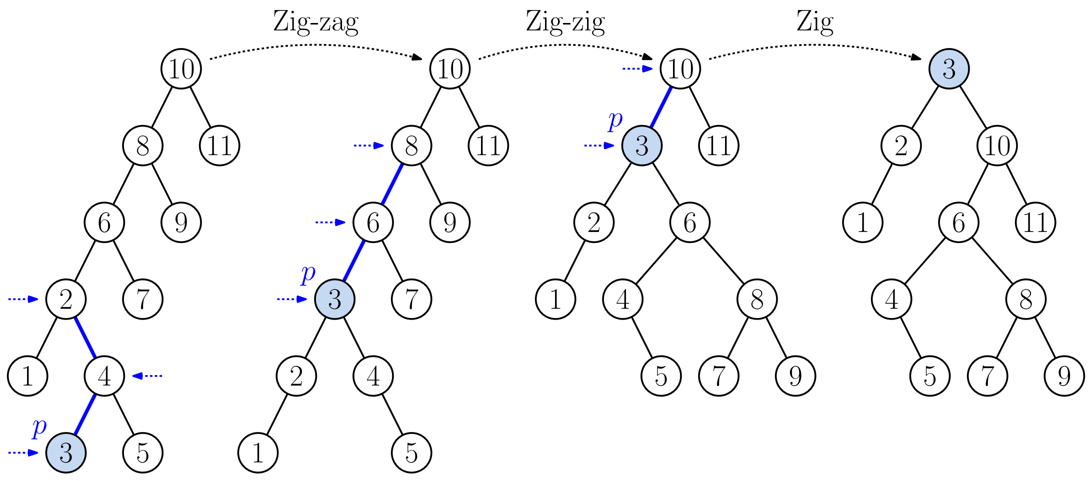

Fig. 6: The full splay operation on p.

Why these rotations? You might wonder why we performed these particular rotations in this particular order. The situation that we are most concerned about is where the tree is highly imbalanced and we repeatedly attempt to access elements that are unusually deep in the tree, say much deeper than $O(\log n)$ depth. Such an operation will be expensive. We want to be sure that we cannot repeat it many times.

Because we rotate as we are backing up the tree, we may assume that the key we sought was in one of p's two children (shaded in blue in the figure). Observe in Fig. 5(a) and (b) that both of these subtrees are lifted up at least one level in the tree following the zig-zig or zig-zag rotation. Thus, if this were a long search path, then after repeating this operation all the way to the root, the nodes along this long path would be lifted up to roughly half of their original level. (The reason we say "halved" is that for every two levels we perform an operation that lifts each subtree up by at least one level.) *Thus, splaying has the desirable effect that it tends to significantly reduce the length of long search paths, whenever we attempt to access something within one of these long paths.*

You may protest at this point. What about the negative impact splaying has on the levels of other subtrees (like G in Fig. 5(a))? This is true. Clearly, there must be winners and losers. But we the accessed node was in p's subtree, and our principal concern is repeat visits deep in the tree cannot be allowed to repeat excessively. A certain amount of damage to the rest of the tree's structure is the price that we pay for this. But to make this convincing, you should read the full amortized proof. We will not cover it since it is quite mathematically involved.

Implementation: As mentioned above. The splay tree requires no balance information, levels, or colors. We assume each node has left, right, and parent links. A high-level description is provided in the code block below. The function is invoked as `root = splay(x)`.

Splay Operation

```
Node splay(Key x) {
    Node p = find x using standard binary tree search
    // if x is not present, then p is the inorder successor or predecessor
    while (p != root) {
         if (p is a child of the rooot)
           zig(p)
        else if (p is left-left or right-right grandchild)
           zig-zig(p)
        else /* p is left-right or right-left grandchild */
           zig-zag(p)
    }
    return p                                // p is the new root
}
```

Rotation Utilities

```
void zig(Node p) {
    if (p == p.parent.left)                 // p is left child of root
        rotateRight(p.parent)               // rotate p to be new root
    else                                    // p is right child of root
        rotateLeft(p.parent)
}

void zig-zig(Node p) {
    if (p == p.parent.parent.left.left) {   // left-left grandchild
        rotateRight(p.parent.parent)        // rotate grandparent first
        rotateRight(p.parent)               // ... then parent
    } else {                                // right-right grandchild
        rotateLeft(p.parent.parent)         // symmetrical
        rotateLeft(p.parent)

    }
}

void zig-zag(Node p) {
    if (p == p.parent.parent.left.right) {  // left-right grandchild
        rotateLeft(p.parent)                // rotate parent first
        rotateRight(p.parent.parent)        // ... then grandparent
    } else {                                // right-left grandchild
        rotateRight(p.parent)               // symmetrical
        rotateLeft(p.parent.parent)
    }
}
```

Splay Tree Operations: Now that we know how to perform a single splay operation, how to we use this to perform the basic dictionary operations, insert, delete, and find? A key idea is to use the splay operation to do the "heavy lifting" after which we perform a few constant-time operations.

`find(x)`: To find key x in tree T, we simply invoke `T.splay(x)`. If x is in the tree it will be transported to the root. If after the operation, the root key is not x, we know that x is not in the dictionary.

This is a bit weird. In all our previous data structures, the `find` operation does not alter the tree structure. Why do it here? Recall that in the case of expected-case optimal trees, a small number of nodes may have very high access probabilities. The splaying operation has the tendency to keep these frequently accessed nodes nearest to the root.

Find in a Splay Tree

```
Value find(Key x) {
    root = splay(x)
    if (root.key == x) return x.value       // found it
    else return null                        // not found
}
```

`insert(x, v)`: To insert the key-value pair (x, v), we first invoke `T.splay(x)`. If x is already in the tree, it will be transported to the root, and we can take appropriate action (e.g., throw an exception). Otherwise, the root will consist of some key y that is either the key immediately before x or immediately after x in T. Let us consider the former case ($y < x$), since the other case is symmetrical. Let R denote the right subtree of the root. We know that all the keys in R are greater than x so we create a new root node containing x and v, and we make R its right subtree (see Fig. 7). The remaining nodes are hung off as the left subtree L of this node.

Insert in a Splay Tree

```
void insert(Key x, Value v) {
    Node p = splay(x)                       // pull x's pred/succ to root
    if (p.key == x) Error! "Duplicate key"  // Oops! x is already here
    q = new Node(x, v)                      // new node for x
    if (p.key < x) {                        // p is predecessor?
        q.left  = p                         // put p to our left
        q.right = p.right                   // ... and p's right to our right
        p.right = null
    } else /* p.key > x */ { ... }          // symmetrical
    root = q                                // new root holds x
}
```

`delete(x)`: To delete x, we first invoke `T.splay(x)` to bring the deleted node to the root. If the root's key is not x, then x is not in the tree, and we can take appropriate error action. Otherwise, let L and R be the left and right subtrees of the resulting tree (see Fig. 8). If L is empty, then x is the new smallest key in the tree. We remove x and R becomes the new tree. We can do the symmetrical thing if R is empty.

Otherwise, both subtrees are nonempty, and we next find an appropriate replacement node. To do this, we perform `R.splay(x)`. (`L.splay(x)` would work equally well). Huh?

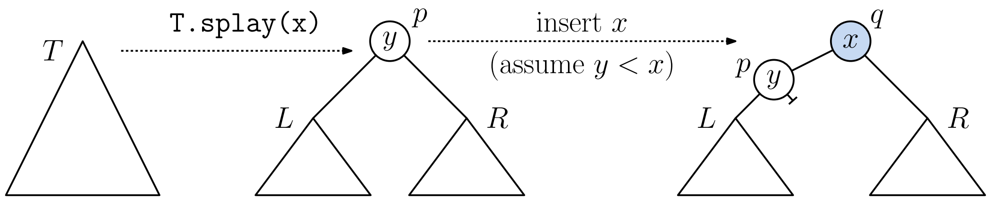

Fig. 7: Splay-tree insertion of x.

Delete in a Splay Tree

```
void delete(Key x) {
    Node p = splay(x)                          // pull x to root
    if (p.key != x) Error! "Non-existant"      // Oops! x is not here
    p.right.splay(x)                           // splay x in p's right subtree
    Node q = p.right                           // q is p's inorder successor
    q.left = p.left                            // transfer left child to q
    root = q                                   // make q the new root
}
```

We know x is not there. The reason is that we want to pull x's preorder successor, call it y, up and make it a child of x. Since y immediately follows x in the node ordering, y's left child must be null. To delete x, we make y the new root of the tree and make x's left child L to be y's left child. (Take time to convince yourself that this is a valid search tree!)

Caveat: The code given above intended to provide an intuitive understanding. We have shown how to update the left and right children, but we have neglected updating the parent links. We will leave this as an exercise.

Why are Splay Trees Great? Splay trees are amazing in the sense that they satisfy (at least theoretically) many optimality properties, which none of the other search trees that we have seen. Here is a partial list.

Balance Theorem: The cost of applying any sequence of m accesses (insert, delete, find) on a splay tree with n elements is $O(m \log n + n \log n)$

Static Optimality Theorem: Let q_x denote the number of times that an element x is accessed in a sequence of m accesses to a splay tree. (Think of $q_x/m = p_x$ as an empirical measure of the probability of accessing x.) Then the cost of performing these accesses is

$$O\left(m + \sum_x q_x \log \frac{m}{q_x}\right).$$

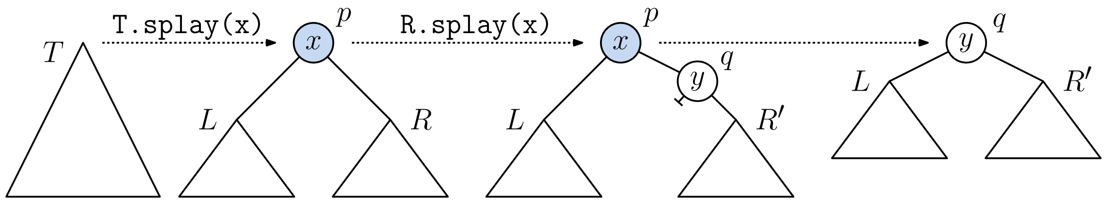

Fig. 8: Splay-tree deletion of x.

The expression in the summation is the *entropy* of the access probability distribution, and this is a theoretical lower bound on the performance of *any* decision-based data structure.

Static Finger Theorem: Assume that the items are numbered 1 through n in ascending order. Let f be any fixed element (the "finger"). Then the cost of performing any sequence of operations is

$$O\left(m + n\log n + \sum_{x} \log(|x - f| + 1)\right).$$

Intuitively, this says that the cost of any operation is the logarithm of the number of elements between the accessed element and the finger.

Dynamic Finger Theorem: Assume that finger for each step in accessing an element y is the location of element accessed in the previous step x. Then the cost of performing any sequence of operations is

$$O\left(m + n\log n + \sum_{x,y} \log(|y - x| + 1)\right).$$

Intuitively, this that if you start a search from some element y to another element x, the cost of the operation is the log of the number of elements between them.

Working-Set Theorem: Each time an element x is accessed, let $t(x)$ be the number of elements that were accessed since x's last access. Then the cost of performing the sequence is

$$O\left(m + n\log n + \sum_{x} \log(t(x) + 1)\right).$$

Intuitively, this says that if we accessed an element t steps ago, then the time to access it now is rougly $\log t$.

Scanning Theorem: If each element of a splay tree is accessed in ascending (or descending) order, the total time for all these accesses is $O(n)$. (A naive bound would be $O(n \log n)$.)

Why Splay Trees Work: (Optional) Sleator and Tarjan proved that, if you start with an empty tree and perform any sequence of m splay tree operations (insert, delete, find), then the total running time will be $O(m \log n)$, where n is the maximum number of elements in the tree at any time. Thus the average time per operation is $O(\log n)$, as we would like. Their amortized analysis involves a *potential-based argument.* We will sketch the idea without getting into the details.

The intuition is to associate a *potential* with any tree. Intuitively, the potential is a value that informs you how badly unbalanced the tree is. In financial terms, we think of potential as money in a bank account. As it accrues, we can use it to pay for the cost of rebalancing the tree. The argument relies on showing that no matter what sequence of operations occurs, there is always a nonnegative potential ("money in the bank"). The *amortized cost* of any operation is defined to be the sum of the actual cost of the operation (e.g., the number of rotations performed) and the change in potential. The objective is to show that the amortized cost of every operation is $O(\log n)$. Some operations may be very costly (e.g., splaying along a path of length n), but the resulting decrease in the tree's potential will be large enough to compensate for this.

So, what is the potential function used for proving splay trees have good amortized performance? First, for each node p of the tree, define $\text{size}(p)$ to be the number of nodes in the subtree rooted at p. Define $\text{rank}(p) = \lg \text{size}(p)$. Intuitively, the rank of a node is the "ideal height" of this subtree in a perfectly balanced tree. The potential function of a tree is defined to be

$$\Phi(T) \;=\; \sum_{p \in T} \text{rank}(p) \;=\; \sum_{p \in T} \lg \text{size}(p).$$

The following is the key to the analysis. It bounds the amortized cost of each rotation operation.

Rotation Lemma: Given any node p, let $\text{rank}(p)$ and $\text{rank}'(p)$ denote its rank before and after applying a rotation step. The amortized cost of a zig rotation at p is at most $1 + 3(\text{rank}'(p) - \text{rank}(p))$, and the amortized cost of a zig-zig or zig-zag rotation at any node p is at most $3(\text{rank}'(p) - \text{rank}(p))$.

(Partial) Proof: We will only prove the case of the zig-zig rotation. The other cases follow by a similar sort of derivation, which we leave as an exercise.

To simplify notation, let's write $\text{rank}(x)$ as $r(x)$ and $\text{size}(x)$ and $s(x)$. Suppose that we perform a zig-zig rotation involving three nodes x, y, and z, as shown in Fig. 5(a) (where x, y, and z play the roles of p, q, and r, respectively). Let $r(x)$, $r(y)$, and $r(z)$ denote the ranks of these items before the rotation, and let $r'(x)$, $r'(y)$, and $r'(z)$ denote these ranks after the rotation. The actual cost for the zig-zig is 2 rotations, and since these are the only changes made in the tree, the change in potential is $\Delta\Phi = (r'(x) + r'(y) + r'(z)) - (r(x) + r(y) + r(z))$. Thus, the amortized cost of this operation is

$$A \;=\; 2 + \Delta\Phi \;=\; 2 + (r'(x) + r'(y) + r'(z)) - (r(x) + r(y) + r(z)).$$

Rearranging terms we have

$$A \;=\; 2 + (r'(x) - r(z)) + r'(y) + r'(z) - r(x) - r(y).$$

Observe that after the rotation x's subtree is the same as z's subtree before the rotation, so $r'(x) = r(z)$. Also, observe that before the rotation y's subtree contains x's subtree, so $r(y) \geq r(x)$ or equivalently $-r(y) \leq -r(x)$. After the rotation y's subtree is contained in x's subtree, so $r'(y) \leq r'(x)$. Thus, we have

$$A \;\leq\; 2 + 0 + r'(x) + r'(z) - r(x) - r(x) \;=\; 2 + r'(x) + r'(z) - 2r(x).$$

Next, we will employ an observation about the logarithm function. It is a concave function, which implies that for any a and b, $(\lg a + \lg b)/2 \leq \lg((a+b)/2)$. Also, observe that $s(x) + s'(z) \leq s'(x)$. Using these and the fact that $r(x) = \lg s(x)$, we have

$$\begin{aligned} \frac{r(x) + r'(z)}{2} \;&=\; \frac{\lg s(x) + \lg s'(z)}{2} \;\leq\; \lg \frac{s(x) + s'(z)}{2} \\ &\leq\; \lg \frac{s'(x)}{2} \;=\; (\lg s'(x)) - 1 \;=\; r'(x) - 1. \end{aligned}$$

This implies that $r'(z) \leq 2r'(x) - r(x) - 2$. Plugging this in to our expression for A, we have

$$A \;\leq\; 2 + r'(x) + (2r'(x) - r(x) - 2) - 2r(x) \;\leq\; 3r'(x) - 3r(x) \;\leq\; 3(r'(x) - r(x)),$$

which is what we set out to prove. Whew!

It is rather difficult to see what the benefit of this symbol manipulation is, but the key is that we have completely eliminated the $+2$ term in the zig-zig and zig-zag cases. This means that we can perform any number of these and the only terms that accumulate will be of the form $r'(x) - r(x)$, which is just the potential change. By applying this all the way up the rotation path, we obtain a telescoping series (and a final $+1$ for the last zig rotation). This implies the following:

Splay Lemma: The amortized cost of a `T.splay(p)` is at most $1 + 3(\text{rank(root)} - \text{rank}(p))$.

Since the rank of the root cannot exceed $\lg n$ and the rank of p is nonnegative, we immediately have:

Corollary: The amortized cost of `T.splay(p)` is $O(\log n)$.

Finally, since each dictionary operation involves a constant number of splaying operations, we obtain the final result.

Theorem: The amortized cost of each dictionary operation in a splay tree is $O(\log n)$.

Wow! That is one impressive bit of data structure analysis. You are not responsible for knowing it, but this is great example of how amortized analyses are performed.