2 Splay trees

In this chapter, we will present self-adjusting binary search trees called the Splay trees. They were discovered in 1983 by Daniel Sleator and Robert Tarjan. They are based on a very simple idea: whenever we access a node, we bring it to the root by a sequence of rotations. Surprisingly, this is enough to guarantee amortized $\mathcal{O}(\log n)$ cost of all operations. In cases when the items are accessed with non-uniform probabilities, the Splay trees will turn out to be even superior to ordinary balanced trees.

2.1 Splaying

Let us consider an arbitrary binary tree. We define the operation $\textsc{Splay}(x)$, which brings the node x to the root using rotations. This can be obviously done by repeatedly rotating the edge above x until x becomes the root, but this does not lead to good amortized complexity (see exercise 1).

The trick is to prefer double rotations as shown in figure 2.1. If x is a left child of a left child (or symetrically a right child of a right child), we perform the *zig-zig* step. If x is a right child of a left child (or vice versa), we do the *zig-zag* step. When x finally becomes a child of the root, we perform a single rotation — the *zig* step.

We can see the process of splaying in figure 2.2: First we perform a zig-zig, then again a zig-zig, and finally a single zig. You can see that splaying tends to reform long paths to branched trees. This gives us hope that randomly appearing degenerate subtrees will not stay for long.

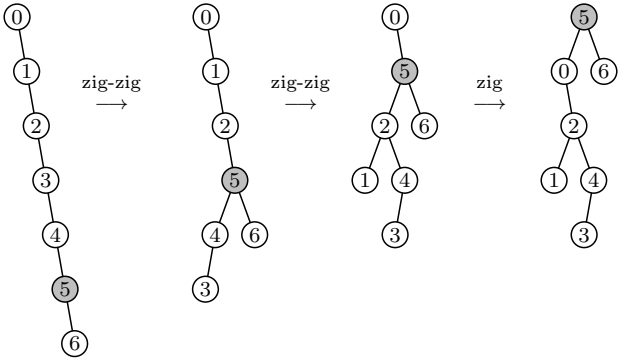

Figure 2.2: Steps of splaying the node 5

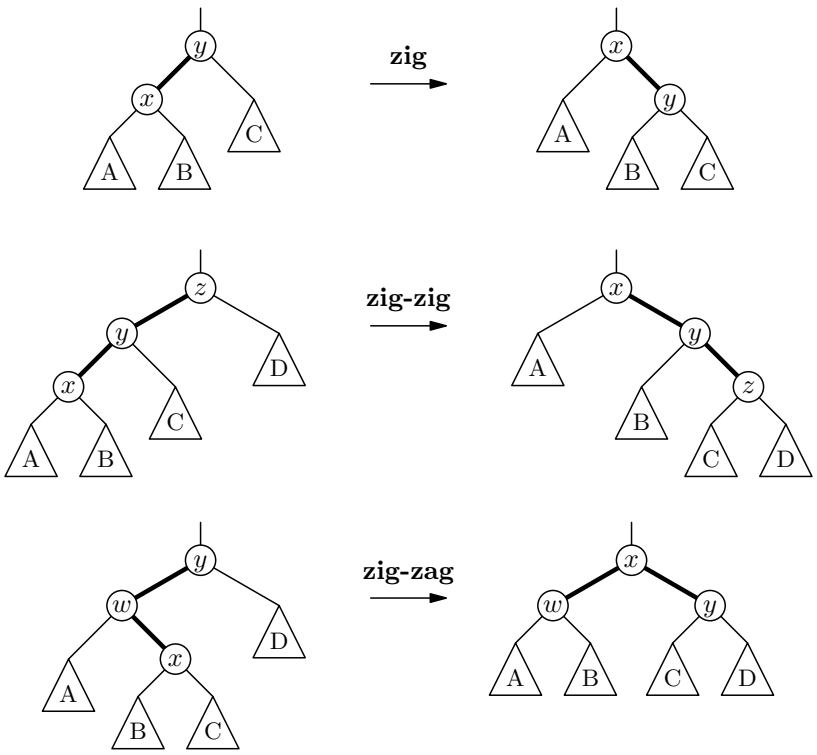

Figure 2.1: Three types of splay steps

Our amortized analysis will be based on a cleverly chosen potential. It might look magical, as if Sleator and Tarjan pulled it out of a magician's hat. But once we define the potential, the rest of the analysis becomes straightforward.

Notation:

- $T(v)$ denotes the subtree rooted at the node v.
- The *size* $s(v)$ is the cardinality of the subtree $T(v)$.
- The *rank* $r(v)$ is the binary logarithm of $s(v)$.
- The *potential* Φ of the splay tree is the sum of ranks of all its nodes.
- n is the total number of nodes in the tree.

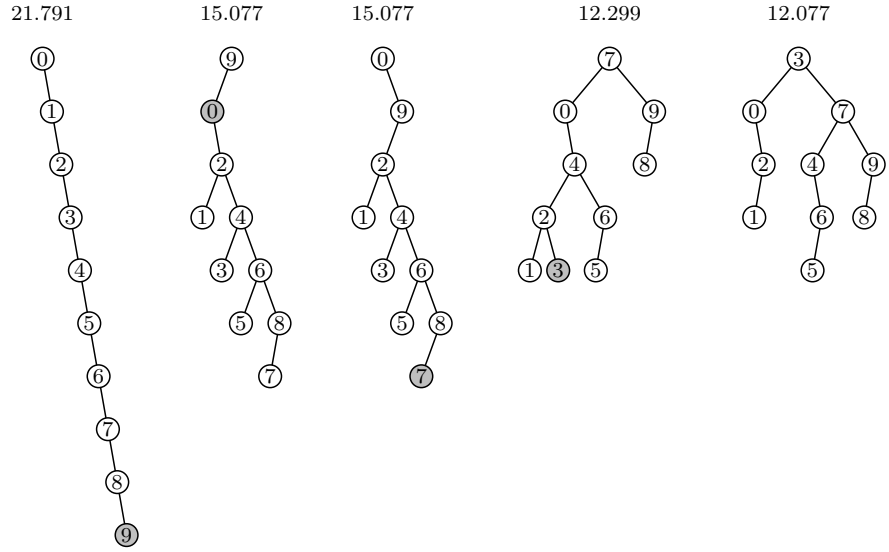

Figure 2.3: Evolution of potential during splays of nodes 9, 0, 7, 3

Observation: For any node v, we have $1 \leq s(v) \leq n$, $0 \leq r(v) \leq \log n$, $0 \leq \Phi \leq n \log n$.

Figure 2.3 suggests that higher potentials correspond to less balanced trees. By repeated splaying, the path gradually becomes a branching tree. The potential keeps decreasing; it decreases faster during expensive splays.

We are going to prove that this holds in general. We will quantify the cost of splaying by the number of rotations performed (so a zig-zig or zig-zag counts as two rotations). Real time complexity is obviously linear in this cost.

Theorem: The amortized cost of $\textsc{Splay}(x)$ is at most $3 \cdot (r'(x) - r(x)) + 1$, where $r(x)$ is the rank of the node x before the operation and $r'(x)$ the rank after it.

Proof: The amortized cost of the full Splay is a sum of amortized costs of the individual steps. Let $r_1(x), \ldots, r_t(x)$ denote the rank of x after each step and $r_0(x)$ the rank before the first step.

We will use the following claim, which will be proven in the rest of this section:

> **Claim:** The amortized cost of the i-th step is at most $3r_i(x) - 3r_{i-1}(x)$, plus 1 if it is a zig step.

As there is at most one zig step, the total amortized cost becomes:

$$A \leq \sum_{i=1}^{t} \big(3r_i(x) - 3r_{i-1}(x)\big) + 1.$$

This is a telescopic sum: each rank except r_0 and r_t participates once positively and once negatively. Therefore the right-hand side is equal to $3r_t(x) - 3r_0(x) + 1$ as claimed by the theorem. □

Corollary: As all ranks are logarithmic, the amortized cost of Splay is $\mathcal{O}(\log n)$.

When we perform a sequence of m splays, we can bound the real cost by a sum of the amortized costs. However, we must not forget to add the total drop of the potential over the whole sequence, which can be up to $\Theta(n \log n)$. We get the following theorem.

Theorem: A sequence of m Splays on an n-node binary tree runs in time $\mathcal{O}((n+m) \log n)$.

Bounding sums of logarithms

Analysis of individual steps will require bounding sums of logarithms, so we prepare a couple of tools first.

Lemma M (mean of logarithms): For any two positive real numbers α, β we have:

$$\log \frac{\alpha + \beta}{2} \geq \frac{\log \alpha + \log \beta}{2}.$$

Proof: The inequality holds not only for a logarithm, but for an arbitrary *concave* function f. These are the functions whose graphs lie above every line segment connecting two points on the graph. The natural logarithm is concave, because its second derivative is negative. The binary logarithm is a constant multiple of the natural logarithm, so it must be concave, too.

Let us consider a graph of a concave function f as in figure 2.4. We mark points $A = (\alpha, f(\alpha))$ and $B = (\beta, f(\beta))$. We find the midpoint S of the segment AB. Its coordinates are the means of coordinates of the endpoints A and B:

$$S = \left(\frac{\alpha + \beta}{2}, \frac{f(\alpha) + f(\beta)}{2} \right).$$

By concavity, the point S must lie below the graph, so in particular below the point

$$S' = \left(\frac{\alpha + \beta}{2}, f\left(\frac{\alpha + \beta}{2} \right) \right).$$

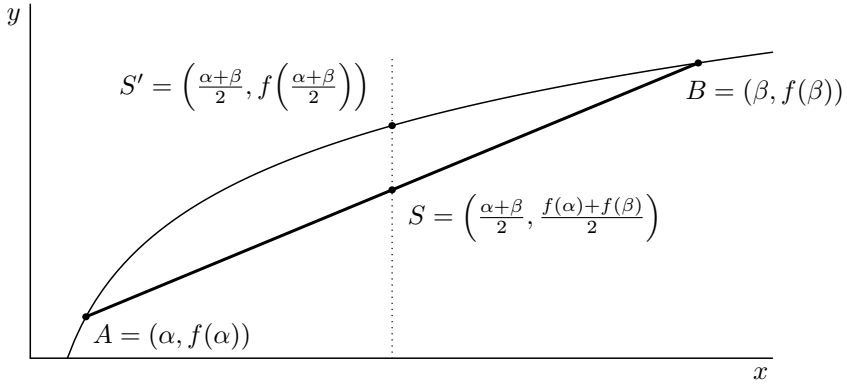

Figure 2.4: The mean inequality for a concave function f

Comparison of y-coordinates of the points S and S' yields the desired inequality. □

Note: The lemma also follows by taking logarithms of both sides of the Arithmetic Mean – Geometric Mean inequality $\sqrt{\alpha\beta} \leq (\alpha+\beta)/2$.

As $\log \frac{\alpha+\beta}{2} = \log(\alpha+\beta) - 1$, the lemma implies:

Corollary S (sum of logarithms): For positive α, β: $\log\alpha + \log\beta \leq 2\log(\alpha+\beta) - 2$.

Now we calculate amortized costs of all three types of splay steps. In each case, x is the node being splayed, $r(x)$ is its rank before the step, and $r'(x)$ its rank after the step. We will use the same convention for s/s' and T/T'.

Zig-zag step

Let us follow figure 2.1 and consider, how the potential changes during the step. The only nodes whose ranks can change are w, x, and y. Thus the potential increases by $(r'(w) - r(w)) + (r'(x) - r(x)) + (r'(y) - r(y))$. The real cost of the operation is 2, so the amortized cost becomes:

$$A = 2 + r'(w) + r'(x) + r'(y) - r(w) - r(x) - r(y).$$

We want to prove that $A \leq 3r'(x) - 3r(x)$. We therefore need to bound all other ranks using $r(x)$ and $r'(x)$.

We invoke Corollary **S** on the sum $r'(w) + r'(y)$:

$$\begin{aligned} r'(w) + r'(y) &= \log s'(w) + \log s'(y) \\ &\leq 2\log(s'(w) + s'(y)) - 2. \end{aligned}$$

The subtrees $T'(w)$ and $T'(y)$ are disjoint and they are contained in $T'(x)$, so we have $\log(s'(w) + s'(y)) \leq \log s'(x) = r'(x)$. Thus:

$$r'(w) + r'(y) \leq 2r'(x) - 2.$$

Substituting this to the inequality for A yields:

$$A \leq 3r'(x) - r(w) - r(x) - r(y).$$

The other ranks can be bounded trivially:

$$\begin{aligned} r(w) \geq r(x) \quad &\text{because } T(w) \supseteq T(x), \\ r(y) \geq r(x) \quad &\text{because } T(y) \supseteq T(x). \end{aligned}$$

Zig-zig step

We will follow the same idea as for the zig-zag step. Again, the real cost is 2. Ranks can change only at nodes x, y, and z, so the amortized cost becomes:

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

We want to get rid of all terms except $r(x)$ and $r'(x)$. To achieve this, we would like to invoke Corrolary **S** on a pair of subtrees, which are disjoint and their union contains almost all nodes. One such pair is $T(x)$ and $T'(z)$:

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

This is equivalent to the inequality $r'(z) \leq 2r'(x) - r(x) - 2$. Thus:

$$A \leq 3r'(x) + r'(y) - 2r(x) - r(y) - r(z).$$

All other terms can be bounded trivially:

$$\begin{aligned} r(z) &= r'(x) && \text{because } T(z) = T'(x), \\ r(y) &\geq r(x) && \text{because } T(y) \supseteq T(x), \\ r'(y) &\leq r'(x) && \text{because } T'(y) \subseteq T'(x). \end{aligned}$$

The claim $A \leq 3r'(x) - 3r(x)$ follows.

Zig step

The real cost is 1, ranks can change only at nodes x and y, so the amortized cost becomes:

$$A = 1 + r'(x) + r'(y) - r(x) - r(y).$$

By inclusion of subtrees, we have $r'(y) \leq r'(x)$ and $r(y) \geq r(x)$, hence:

$$A \leq 1 + 2r'(x) - 2r(x) \leq 1 + 3r'(x) - 3r(x).$$

The latter inequality holds, because by inclusion of subtrees, $r'(x) - r(x)$ is always non-negative.

Exercises

1. Prove that a naive splay strategy using only single rotations cannot have better amortized complexity than $\Omega(n)$. Consider what happens when splaying a path.

2. What is the potential of a path and of a perfectly balanced tree?

3.* *Top-down splaying:* In many operations with trees, we have to walk the path between the root and the current node twice: once down when we are looking up the node, once up when we are splaying it. Show how to combine splaying with the lookup, so that both can be performed during a single top-down pass.

4.* *Sequential traversal:* Prove that total cost of splaying all nodes in order of their keys is $\Theta(n)$.

5.* *Alternative analysis:* Alternatively, we can define the rank $r(v)$ as $\lfloor \log s(v) \rfloor$. Prove that this leads to a similar amortized bound. If $r(x) = r'(x)$ during a step, the difference pays for the step (if we assume that the real cost is scaled to 1). Otherwise, use that all ranks before the operation must have been equal.

2.2 Splaying in search trees

So far, we studied splaying in an otherwise static binary tree. Let us have a look on how to use the Splay tree as a binary search tree. It will turn out that we can use the ordinary FIND, INSERT, and DELETE as on unbalanced search trees, provided that we always *splay the lowest visited node.* This alone will lower amortized cost of all operations to $\mathcal{O}(\log n)$.

Find

Suppose that FIND found the given key at depth d. Going from the root to this node had real cost $\Theta(d)$ and it did not change the potential. The subsequent SPLAY will also have cost $\Theta(d)$ and it amortizes to $\mathcal{O}(\log n)$. We can therefore account the cost of the former part on the splay. This multiplies the real cost of splaying by a constant, so multiplying both the amortized cost and the potential by the same constant gives amortized complexity of the whole FIND in $\mathcal{O}(\log n)$.

An unsuccessful FIND also goes from the root down to some node, so if we splay the stopping node, the same argument applies. The MIN operation is essentially an unsuccessful FIND$(-\infty)$, so it is also $\mathcal{O}(\log n)$ amortized. The same goes with MAX.

This is a useful general principle: *Whenever we walk from the root to a node and we splay that node, together it will have amortized cost* $\mathcal{O}(\log n)$.

Insert

An INSERT tries to find the new key. If it succeeds, the key is already present in the set, so it stops, not forgetting to splay the discovered node. If it hits a null pointer, it connects a new leaf there with the new key. Again, we splay this node.

As usual, splaying the node will offset the cost of walking the tree, but there is one problem: while adding a leaf has constant real cost, it could increase the potential by a disastrous amount. We will prove that this is not the case.

Lemma: Adding a leaf to the tree increases potential by $\mathcal{O}(\log n)$.

Proof: Let us denote $v_1, \ldots, v_{t+1}$ the path from the root v_1 to the new leaf v_{t+1}. Again, we will use $r(v_i)$ for the rank before the operation and $r'(v_i)$ for the new rank. The rank $r'(v_{t+1})$ newly enters the potential; ranks $r(v_1), \ldots, r(v_t)$ increase. The potential difference is therefore:

$$\Delta\Phi = r'(v_{t+1}) + \sum_{i=1}^{t} (r'(v_i) - r(v_i)).$$

As leaves have size 1, the rank $r'(v_{t+1})$ is zero. As for the other ranks $r'(i)$: we know that $s'(v_i) = s(v_i) + 1$, but this is certainly at most $s(v_{i-1})$. So for $i > 1$, we have

$r'(v_i) \leq r(v_{i-1})$. The sum therefore telescopes and we can reduce it to $\Delta\Phi \leq r'(v_1) - r(v_t)$. This is $\mathcal{O}(\log n)$, because all ranks are at most $\log n$. □

The total amortized cost of Insert is therefore $\mathcal{O}(\log n)$.

Delete

The traditional implementation of Delete starts with finding the node. The actual deletion has three cases. Two of them are easy: we are removing either a leaf, or an internal node with one child. The case of a node with two children is solved by replacing the node (let's say) by the minimum of its right subtree, which reduces it to one of the easy cases.

Each deletion therefore consists of a walk from the root to a certain node (which is either the node with the given key, or its replacement, which is even deeper) and removal of this node. The cost of the walk can be offset by splaying the parent of the removed node; if there is no parent, the node had constant depth, so the cost is constant anyway.

Removing a node has constant real cost. The change in the potential is in favor of us: we are removing one node from the tree and decreasing the ranks of all its ancestors, so the potential difference is negative.

We can conclude that Delete runs in $\mathcal{O}(\log n)$ amortized time.

Splitting and joining

Alternatively, we can implement Insert and Delete using splits and joins of trees. It is easier to analyse and often also to implement. As we are considering multiple trees, we redefine the potential to sum ranks of all nodes of all trees together.

Insert starts with an unsuccessful Find. It stops in a node v, which is either a predecessor or a successor of the new key. We splay this node to the root. If it was a successor, we look for its left child ℓ. If there is none, we simply connect the new node as the left child. Otherwise we subdivide the edge $v\ell$ by the new node (we make the new node left child of v and we connect ℓ as the left child of the new node). If v was the predecessor, we do the same with the right child.

The cost of finding v is offset by the splay. Connecting the new node has constant real cost and it changes the potential by modifying at most three ranks. Regardless of how these ranks changed, the total difference is $\mathcal{O}(\log n)$.

Delete is similar:

1. We FIND the node and splay it to the root.
2. We remove the root, which splits the tree to a left subtree L and a right subtree R. If R is empty, we stop.
3. We find the minimum m of R and splay it. We note that m has no left child.
4. We connect the root of L as the left child of m.

Steps 1 and 3 consist of a walk followed by a splay, so they have $\mathcal{O}(\log n)$ amortized cost. Step 2 has constant real cost and it decreases the potential. Step 4 also has constant real cost and it increases the potential by increasing the rank of m, but all ranks are $\mathcal{O}(\log n)$.

This implementation of INSERT and DELETE therefore also runs in $\mathcal{O}(\log n)$ amortized time. We note that splits and joins are useful building blocks for other operations and they can be implemented easily and efficiently with splaying.

Conclusion

In the previous section, we proved a theorem bounding total time of a sequence of splays. We are going to prove a similar bound for a general sequence of set operations. Here we do not need the extra $\mathcal{O}(n \log n)$ term: if we start with an empty tree, the initial potential is zero and the final potential non-negative. The potential drop over the sequence is therefore non-positive.

Theorem: A sequence of m operations FIND, INSERT, and DELETE on an initially empty Splay tree takes $\mathcal{O}(m \log n)$ time, where n is the maximum number of nodes in the tree during the sequence.

Exercises

1. Show how to implement PRED and SUCC in $\mathcal{O}(\log n)$ amortized time.

2.3* Weighted analysis

Splay trees have surprisingly many interesting properties. Some of them can be proven quite easily by generalizing the analysis of *Splay* by putting different weights on different nodes.

We will assign a positive real *weight* $w(v)$ to each node v. The size of a node, which was the cardinality of its subtree, becomes the sum of weights. The definition of the rank and the potential will not change, but they will be based on the modified sizes. The original analysis therefore remains as a special case with all weights equal to 1.

Since weights are strictly positive, sizes will be also strictly positive. All ranks will therefore be well defined, but possibly negative. Our theory of amortized complexity does

not break on negative potentials, but we must take care to include the total potential drop in our time bounds.

Surprisingly, the amortized cost of splaying a node x stays bounded by $3 \cdot (r'(x) - r(x)) + 1$. The proof of this result relies on just two properties of sizes: First, all sizes must be positive, so all ranks are well defined and we can use the inequalities for logarithms. Second, we need the sizes to be monotonous: if $T(u)$ is a subset of $T(v)$, then $s(u) \leq s(v)$, so $r(u) \leq r(v)$. Both properties are satisfied in the weighted case, too.

However, the general $\mathcal{O}(\log n)$ bound on the amortized cost of splaying ceases to hold — ranks can be arbitrarily higher than $\log n$. We can replace it by $\mathcal{O}(r_{\max} - r_{\min})$, where $r_{\max} = \log(\sum_v w(v))$ is the maximum possible rank and $r_{\min} = \log \min_v w(v)$ the minimum possible rank.

We also have to be careful when analysing search tree operations:

- Find has the same complexity as Splay: $\mathcal{O}(r_{\max} - r_{\min})$.
- Insert based on adding a leaf and splaying it loses its beauty — when we add a leaf, the sum of rank differences no longer telescopes (and indeed, the amortized cost can be high).
- The second Insert performed by subdividing an edge below the root keeps working. Except for the splay, we are changing ranks of $\mathcal{O}(1)$ nodes, therefore the potential difference is $\mathcal{O}(r_{\max} - r_{\min})$ and so is the total amortized cost.
- The traditional version of Delete changes the structure by removing one leaf or one internal nodes with one child. Sizes of all other nodes do not increase, so their ranks do not increase either. Rank of the removed node could have been negative, so removing it can increase the potential by at most $-r_{\min}$. Total amortized cost of Delete is therefore also $\mathcal{O}(r_{\max} - r_{\min})$.
- Delete based on splits and joints is easier to analyse. Splitting a tree can only decrease sizes, so the rank does not increase. Joining roots of two trees changes only the rank of the new root, therefore the potential changes by $\mathcal{O}(r_{\max} - r_{\min})$. So the total amortized cost is again $\mathcal{O}(r_{\max} - r_{\min})$.

In the rest of this section, we will analyze Splay with different settings of weights. This will lead to stronger bounds for specific situations. In all cases, we will study sequences of accesses to a constant set of nodes — no insertions nor deletions will take place, only splays of the accessed nodes.

We will use W for the sum of weights of all nodes, so $r_{\max} = \log W$. This leads to the following bound on the cost of splaying:

Lemma W: The amortized cost of Splay(x) is $\mathcal{O}(\log(W/w(x)) + 1)$.

Proof: We are upper-bounding $3 \cdot (r'(x) - r(x)) + 1 = \mathcal{O}(\log(s'(x)/s(x)) + 1)$. As splaying makes x the root, we have $s'(x) = W$. As weights are non-negative, $s(x) \geq w(x)$. □

Warm-up: Uniform weights

Suppose that accesses to items are coming from a certain probability distribution. It is often useful to analyse the accesses by setting weights equal to access probabilities (assuming every item is accessed with a non-zero probability). This makes W always 1.

Let us experiment with the uniform distribution first. All n weights will be $1/n$. The size of each node therefore lies between $1/n$ and 1, ranks range from $-\log n$ to 0, so we have $-n \log n \leq \Phi \leq 0$.

By the above lemma, the amortized cost of a single Splay is $\mathcal{O}(\log(1/(1/n)) + 1) = \mathcal{O}(\log n)$. A sequence of m splays therefore takes $\mathcal{O}(m \log n)$ plus the total potential drop over the sequence, which is at most $n \log n$.

We got the same result as for unit weights. In fact, *scaling the weights by a constant factor never changes the result.* If we multiply all weights by a factor $c > 0$, all sizes are multiplied by the same factor, so all ranks are increased by $\log c$. Therefore, rank differences do not change. Potential is increased by $n \log c$, so potential differences do not change either.

Static optimality

Now consider that the accesses are coming from a general distribution, which can be very far from uniform. The actual cost of an access is a random variable, whose expected value can differ from $\Theta(\log n)$. If we know the distribution in advance, we can find an optimum static tree T, which minimizes the expected cost $\mathbb{E}_x[c_T(x)]$. Here $c_T(x)$ denotes the cost of accessing the item x in the tree T measured as the number of items on the path from the root to x.

The optimal static tree can be found in time $\mathcal{O}(n^2)$ by a simple algorithm based on dynamic programming. We will prove that even though the Splay trees possess no knowledge of the distribution, they perform at most constant-times worse than the optimal tree.

Let $p(x)$ be the probability of accessing the item x. This will be also the weight $w(x)$, so the total weight W is exactly 1. The amortized cost of the access to x will be $\mathcal{O}(\log(1/w(x)) + 1) = \mathcal{O}(\log(1/p(x)) + 1)$. Therefore the expected amortized cost is:

$$\mathbb{E}\left[\mathcal{O}\left(\log \frac{1}{p(x)} + 1\right)\right] = \mathcal{O}\left(1 + \sum_x p(x) \cdot \log \frac{1}{p(x)}\right).$$

The sum at the right-hand side is the Shannon entropy of the distribution. Since entropy is a lower bound on the expected code length of a prefix code, it is also a lower bound on

the expected node depth in a binary search tree. Therefore, the expected amortized cost of splaying is $\mathcal{O}$ of the expected access cost in the optimal static tree. This holds even for expected real cost of splaying if the access sequence is long enough to average out the total potential drop.

There is however a more direct argument, which does not need to refer to information theory. We will state it for a concrete access sequence with given *access frequencies.*

Theorem (Static optimality of Splay trees): Let $x_1, \ldots, x_m$ be an access sequence on a set X, where every item $x \in X$ is accessed $f(x) > 0$ times. Let T be an arbitrary static tree on X. Then the total cost of accesses in a Splay tree on X is $\mathcal{O}$ of the total cost of accesses in T.

Proof: The tree T will process all accesses in time $\mathcal{O}(\sum_x f(x) \cdot c_T(x))$, where $c_T(x)$ is the (non-zero) cost of accessing the item in the tree T. For analysis of the Splay tree, we set the weight of each item x based on this cost: $w(x) = 3^{-c_T(x)}$. The sum of all weights is $W < 1$, because in an infinite regular binary tree, it would be exactly 1. As for the ranks, we have $r(x) = \log s(x) \leq \log W < 0$, so all ranks are negative.

By Lemma **W**, the amortized cost of accessing x in the Splay tree is $\mathcal{O}(\log(W/w(x))+1) = \mathcal{O}(\log 3^{c_T(x)}+1) = \mathcal{O}(c_T(x)+1) = \mathcal{O}(c_T(x))$. The sum of amortized costs is therefore $\mathcal{O}$ of the cost in T, but we must not forget to add the total potential drop $\Phi_0 - \Phi_m \leq -\Phi_m$. However, the negative sum of all ranks is $\mathcal{O}(\sum_x c_T(x))$, which is again $\mathcal{O}$ of the access cost in T, because all frequencies are non-zero. □

It is still an open question whether Splay trees are *dynamically optimal* — that is, at most $\mathcal{O}(1)$ times slower than the best possible dynamic tree which knows the access sequence beforehand and adjusts itself optimally.

Static finger bound

There is also a number of results of the type "if accesses are *local* (in some sense), then they are faster". A simple result of this kind is the static finger bound. Without loss of generality we can renumber the items to $1, \ldots, n$. We select one item f as a *finger.* Locality of access to an item i can be measured by its distance from the finger $|i - f|$. Then we have the following bound:

Theorem (Static finger bound): Consider a Splay tree on the set of items $X = \{1, \ldots, n\}$ and a finger $f \in X$. Accesses to items $x_1, \ldots, x_m \in X$ are processed in time $\mathcal{O}(n \log n + m + \sum_i \log(|x_i - f| + 1))$.

Proof: We set items weights to $w(i) = 1/(|i - f| + 1)^2$. As the sum of all weights contains each rational number $1/k^2$ at most twice, the sum of all weights W is at most

$2 \cdot \sum_{k \geq 1} k^{-2} = \pi^2/3$, so it is $\mathcal{O}(1)$. By Lemma **W**, the amortized cost of an access to x_i is $\mathcal{O}(\log(W/w(x_i)) + 1) = \mathcal{O}(\log(1 + |x_i - f|)^2 + 1) = \mathcal{O}(\log(1 + |x_i - f|) + 1)$.

We sum this over all accesses, but we need to add the total potential drop over the sequence. Since all sizes are between $1/n^2$ and W, ranks are between $-2 \log n$ and a constant, so the potential lies between $-2n \log n$ and $\mathcal{O}(n)$. Therefore the potential drop $\Phi_0 - \Phi_m$ is $\mathcal{O}(n + n \log n) = \mathcal{O}(n \log n)$. □

Working set bound

A similar technique can be used to obtain a more interesting theorem: the cost of access to an item x can be bounded using the number of distinct items accessed since the previous access to x — the so-called *working set.* If there is no previous access to x, the working set contains all items accessed so far. The amortized cost of the access will be approximately the logarithm of the size of the working set.

We are going to compare the Splay tree with the *LRU (least-recently-used) list.* Initially, the list contains all items in order of their first access, starting at index 0. Items which are never accessed are placed in an arbitrary order at the end of the list. Whenever we access an item x_i, we move it from its position $p(x_i)$ to the start of the list. The items we skipped over are equal to the working set of x_i, so the size of the working set is $p(x_i)$.

Theorem (Working set bound): Consider a Splay tree on a set of n items. Accesses to items $x_1, \ldots, x_m$ are processed in time $\mathcal{O}(n \log n + m + \sum_i \log(1 + z_i))$, where z_i is the size of the working set for x_i.

Proof: We set item weights according to their positions in the current LRU list: $w(x) = 1/(p(x)+1)^2$. As each position occurs exactly once, the total weight W is again $\mathcal{O}(1)$. All sizes therefore range between $1/n^2$ and $\mathcal{O}(1)$, ranks range between $-2 \log n$ and a constant, and the potential is between $-2n \log n$ and $\mathcal{O}(n)$. However, we must not forget that any change of the LRU list causes re-weighting and thus a change of the potential.

Let us analyze a cost of the access to x_i. As usual, we invoke Lemma **W** to bound the amortized cost of Splay by $\mathcal{O}(\log(W/w(x_i)) + 1) = \mathcal{O}(\log(p(x_i)+1)^2 + 1) = \mathcal{O}(\log(p(x_i)+1) + 1) = \mathcal{O}(\log(1 + z_i) + 1)$.

Then we have to move x_i to the head of the LRU list (at least in our mind). How is the potential affected? The position of x_i decreases from $p(x_i)$ to 0, so the weight $w(x_i)$ increases from $1/(p(x_i) + 1)^2$ to 1. The positions of the skipped items increase by 1, so their weights decrease. The positions and weights of all other items stay the same. Therefore the size of x_i increases from at least $1/(p(x_i) + 1)^2$ to at most W, so its rank increases by $\mathcal{O}(\log W - \log 1/(p(x_i)+1)^2) = \mathcal{O}(1 + \log(z_i + 1))$. Sizes of all other items do not increase, because their subtrees do not contain x_i, which is currently the root. Total increase in the potential is therefore bounded by the amortized cost of the Splay itself.

We sum the amortized cost over all accesses, together with the total potential drop. As the absolute value of the potential is always $\mathcal{O}(n \log n)$, so is the potential drop. □

The working set property makes Splay tree a good candidate for various kinds of caches.