Top-Down/Bottom-Up Splay Duality and Generalized Top-Down Splay

Alex Ozdemir

21 April 2017

Written under the guidance of

Ran Libeskind-Hadas

in fulfillment of the

Capstone Requirement

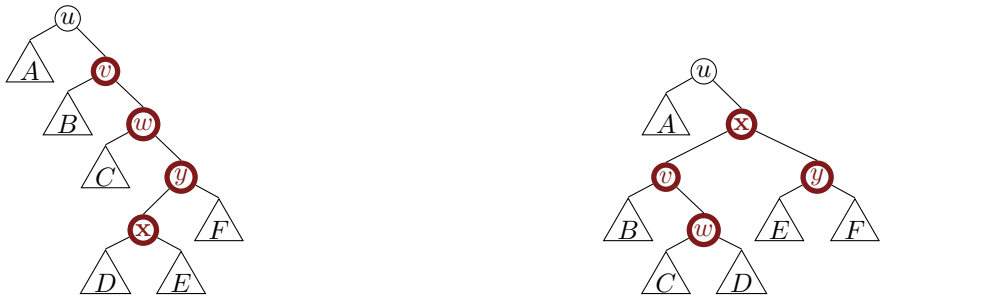
of Harvey Mudd College

Contents

1 Introduction

In 1985 Sleator and Tarjan [1] showed that a particular data structure, the "splay tree", is capable of restructuring itself to bring any item to the root in amortized logarithmic time. They provided two variants of the splay procedure: the bottom-up (BU) variant and the top-down (TD) variant. While they proved the the BU splay procedure takes logarithmic amortized time, they did not prove the TD splay procedure satisfies the same bound. However, programmers quickly found that the TD procedure is faster in practice. One year later Mäkinen [2] published a proof that the TD procedure indeed runs in amortized logarithmic time.

In the same year Subramanian [3] reformulated Sleator and Tarjan's BU splay procedure in a more general form. He described it as an algorithm that references a set of rules, consisting of pairs of *templates* (tree substructures that the area around the target node can match) and *results* (how to rearrange the matched node). The general BU splay procedure proceeds in steps: matching the tree to a template and applying the corresponding transformation (Figure 1), until the target node is brought to the top of the tree. He demonstrated that if the rule set meets certain conditions, the general splay procedure provides all the guarantees the original splay procedure does, most notably amortized logarithmic access time. Given that the conditions are fairly general, this proves the existence of a large set of BU splay procedures. Unfortunately, the existence of a generalized TD splay procedure with amortized logarithmic access remained unknown [3] [**?**].



(a) The template (the red nodes) have been identified above the target node x.

(b) The active nodes been rearranged into the result, with x at their root.

Figure 1: A tree undergoing a transformation described by some template-result pair. Such a pair is called a *rule*.

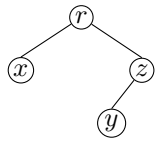
In this document we pursue a goal analogous to Subramanian's for TD splay procedures rather than BU ones. We outline a TD splay procedure that is generic over rule sets. We prove that if a rule set satisfies a few (fairly general) conditions, then the corresponding TD splay procedure runs in amortized logarithmic time[1]. This proves the existence of a large set of TD splay procedures that run in amortized logarithmic time. This result is particularly interesting given that programmers have found the original TD splay procedure performs better in practice than the original BU splay procedure.

2 Notation & Vocabulary

Henceforth we use common terms such as *trees*, *nodes*, *subtrees*, and *children*. We also use *left hole* and *right hole* to refer to the locations in the tree where a least or greatest item would be inserted (Figure 2). We also refer to *paths*: sequences of edges in a tree that connect a node to some descendant of it. A can be defined by it's starting node and a sequence of left/right steps. We can

[1] and per Section 5 has other desirable properties

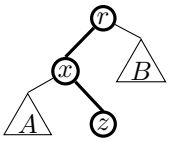
illustrate paths with or without showing the intermediate nodes and sibling subtrees, as shown in Figure 3.



(a) This tree's right hole is where z's right child would be placed.

(b) How we would denote the tree to indicate it's right hole.

Figure 2: Holes



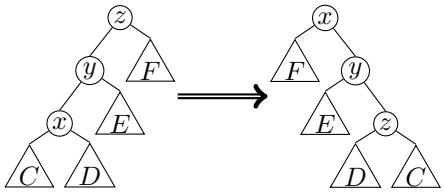
(a) A path in a tree from r to z is bolded. Subtrees hanging off the path are shown.

(b) A path in a tree from r to z. Subtrees hanging off the path are not shown.

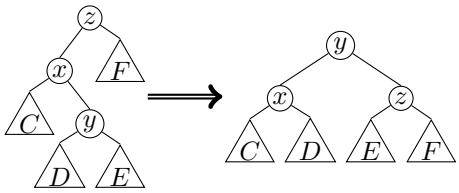
Figure 3: Subtrees and Paths

2.1 Rule Sets for the Original TD and BU Procedures

The prior notation can be used to express both the BU and TD splay procedures proposed by Sleator and Tarjan. These are shown in Figures 4 and 5 respectively. We express each procedure as a set of *rules*, which are each *template-result* pairs, where the template matches a substructure of the original tree and the result describes how the matched nodes should be restructured.



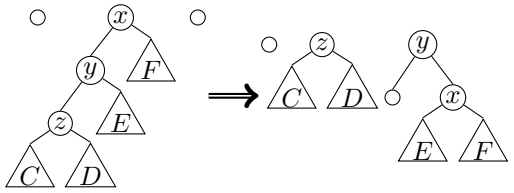
(a) Zig-zig



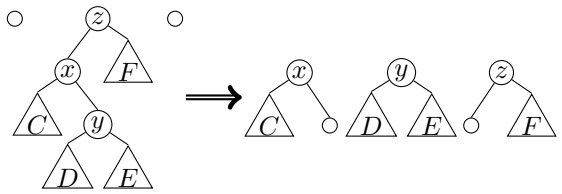
(b) Zig-zag

Figure 4: The BU splay zig-zig and zig-zag rules. The zag-zag and zag-zig rules are mirror images.

The TD splay procedure proposed by Sleator and Tarjan maintains 3 trees: a left, right, and center tree (which is initialized to be the original tree). During each step it takes some items from the center tree and inserts them into the left or right tree. It tracks the location of a *hole* in each of the left and right trees which is where future nodes will be inserted. These holes are denoted by small circles in Figure 5.



(a) Zig-Zig. This rule moved part of the center tree into the right tree, and nothing into the left tree.



(b) Zig-Zag. This rule moved part of the center tree into the right tree, and another part into the left tree.

Figure 5: The TD splay zig-zig and zig-zag rules. The zag-zag and zag-zig rules are mirror images. Notice that top-down rules are given in terms of three trees (the left, center, and right trees) and allow for moving nodes out of the center tree into the others.

Having written out the original BU and TD splay procedures in terms of their rule sets, an obvious question arises:

How are the original BU and TD splay procedures related?

We leave this question unanswered for now, but will return to it shortly.

3 Generalized Top-Down Splay

3.1 Bottom-Up / Top-Down Duality

Previous work has generalized the concept of a BU splay tree over the set of rules used by the splay procedure [3]. This implies the existence of a large set of BU splay procedures which run in amortized logarithmic time. However, the question of whether there exists a similar generalization of the TD splay procedure was left unanswered.

In this section we describe a dual relationship between BU and TD splay rules. This duality makes precise the relationship between the original BU and TD splay trees, and can be used to prove the existence of (and implement) a large set of TD splay procedures that run in amortized logarithmic time.

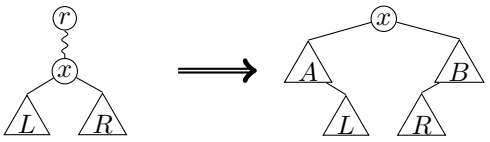
While the past work of Subramanian imposes relatively few restrictions on BU rules in order to cover concepts such as semi-splaying[1], we'll focus on those that satisfy slightly tighter restrictions by considering only rules where:

1. The lowest node in the template is the *target node*: the node being searched for.
2. The transformation triggered by the rule moves the target node up to the root of the structure matched by the template.

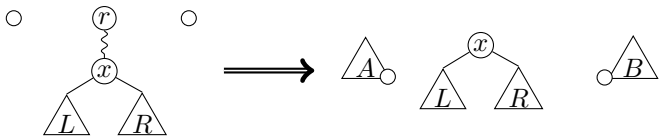
Rules that meet these restrictions have the general form presented in Figure 6a. Their template is a path from the root node r down to the target node x, which has (potentially empty) child subtrees L and R. Their result is rooted at x.

Notice that because the items in the R subtree are greater than x but less than any items in the path that are greater than x, the R subtree must be the leftmost part of x's right subtree, B, after the transformation. Similarly, the L subtree must be the rightmost part of x's left subtree, A, after the transformation.

This formulation of a general BU rule allows us to define each BU rule's *dual* TD rule, as shown in Figure 6b.



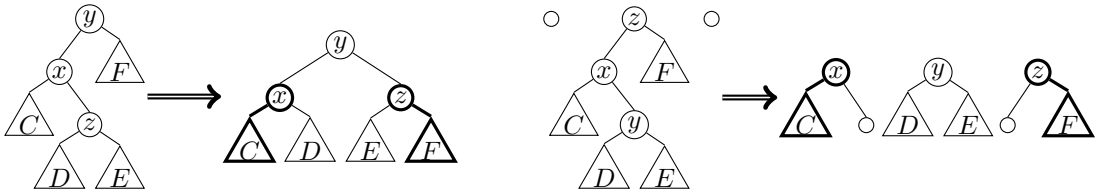
(a) A generic BU rule. Moves x to the root, taking nodes along the path (and left/right of the path), reforming them into subtrees A and B.



(b) The dual TD rule

Figure 6: A generic rule for a BU splay procedure and its dual for a TD splay procedure.

This general dual relationship between TD rules and BU rules describes the specific relationship between the original TD and BU rules. Figure 7 illustrates that the zig-zag BU and TD splay rules are an example of this dual relationship.



(a) The BU zig-zag rule.

(b) The dual TD rule.

Figure 7: The BU and dual TD zig-zag rules. The A and B subtrees from Figure 6 are bolded.

3.2 Generalized Top-Down Splay

We now describe a generalized TD splay procedure as a procedure that similarly references a rule set to determine which transformation to perform. It shares much with the original TD splay procedure: during the splay the procedure incrementally walks the path from the root to the target node by applying its transformation rules to that path from the top down. The rule which is used is the rule with the deepest template (since template are paths, the depth of a template is the number of nodes in that path) that matches the path towards the target node. During this process the procedure maintains three trees:

- The *center tree*, that is initialized as the entire original tree and always contains the target node. The center tree is incrementally pruned during each step until the target node is reached.
- The *left tree*, that starts empty and accumulates the nodes on the path less than the target node.
- The *right tree*, that starts empty and accumulates the nodes on the path greater than the target node.

However, the generalized TD splay procedure differs from the original in one important way. While the original procedure applies a specific rule set (shown in Figure 5), the general procedure is generic over its rule set. That is, it can use any TD rule set so long as

- The rule set contains the *reassembly rule* (shown in Figure 8) which reassembles the left, center, and right trees into a single final tree.
- Every other rule in the set is a TD rule that is dual to some BU rule.

(a) The template (b) The result

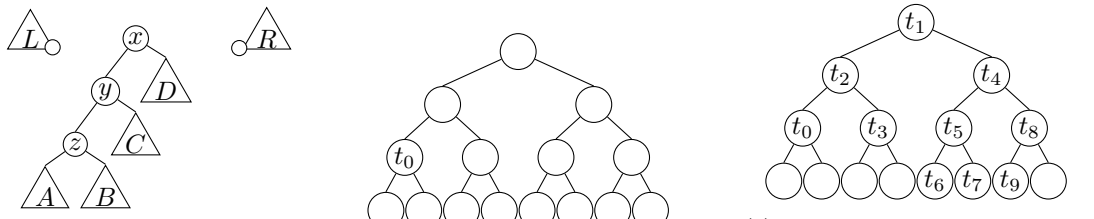
Figure 8: The reassembly rule.

If a TD rule set S satisfies these properties, we'll denote the set of dual BU rules $\hat{S}$ and say that S has *dual* $\hat{S}$. We'll focus our attention on TD rule sets that can be constructed as duals to some BU rule set.

3.3 Constraints

Our algorithms will use a TD rule set, constructed as the dual to a BU rule set. We'll further restrict our analysis to a subset of the possible rule sets. To define this subset we will first develop a way of visualizing the set of templates that an algorithm is using.

Recall that each template matches a path and, as such, one can refer to the unique bottom node matched by a template. This allows us to associate each template with a specific node in the complete binary tree. We'll call this specific node the *action point* of the template. For example, Figure 9 shows a template and its action point (along with the action points of other templates).



(a) The template t_0 (b) The action point of t_0. (c) The action point of t_0 and other templates.

Figure 9: The template t_0 and its action point in the complete tree (along with the action point of templates $t_1, \ldots, t_9$)

We'll define the *boundary action points* to be the action points with a child that is not an action point (In Figure 9c the boundary action points are $\{t_0, t_3, t_6, t_7, t_9, t_8\}$). We refer to rules corresponding to these action points as the *boundary rules*.

We'll require that the boundary rules are *depth reducing*[2]. We adapt Subramanian's definition of *depth-reducing* by defining a TD rule as depth reducing when its BU dual is depth-reducing.

[2]One might be wondering whether multiple rules with the same template (and same action point) can be in the same rule set. Our analysis permits this, with arbitrary choice between applicable rules, so long as *all* boundary rules are depth reducing.

Subramanian calls a BU rule depth reducing if it reduces the depth of each child of the target node [3].

As an example of a depth reducing TD rule, consider the TD zig-zag rule shown in Figure 7. In its BU dual the children of the target node, D and E, start at depth 3 and end at depth 2, so the BU rule is depth-reducing. Therefore the dual TD rule is as well.

Since templates are paths, we'll define the depth of a template to be the number of nodes in its path, and we'll require that the rule set has templates with heights bounded above by some k.

Consider Figure 9c. Notice that all templates have height less than or equal to 3. For the rule set corresponding to these action points to satisfy our conditions, an additional condition must be met: the boundary rules (those corresponding to action points $\{t_0, t_3, t_6, t_7, t_8, t_9\}$) must be depth reducing.

To recap, we require that our rule set is such that:

1. The boundary rules are all depth reducing.
2. The set of templates have heights bounded above by some constant k.

By imposing these constraints on the rule set used by an algorithm we will be able to show that the algorithm retrieves nodes in logarithmic amortized time and has other desirable properties.

4 Amortized Logarithmic Access Time

4.1 Factorability

The proof that Sleator's bottom-up splay trees provide logarithmic amortized access time leverages the fact that any one step of the bottom-up splay procedure has amortized cost bounded above by $3(\text{rank}(x_{i+1}) - \text{rank}(x_i))$, where rank is a function that maps a node in the tree to the (binary) logarithm of the number of nodes in its subtree, x_i is the target node before the step, and x_{i+1} is the target node after the step. This allows the total amortized cost of the algorithm to be written as the sum of the cost of each individual transformation:

$$\sum_{i=1}^{n} 3(\text{rank}(x_{i+1}) - \text{rank}(x_i))$$

where n transformations are done. This turns out to be a telescoping series, so it is equal to:

$$3(\text{rank}(x_{n+1}) - \text{rank}(x_0)) \leq 3\,\text{rank}(x_{n+1}) = 3\log N$$

where N is the number of nodes in the tree.

In essence, the BU splay tree proof depends on the fact that:

1. The algorithm performs a sequence of tree transformations.
2. Each transformation manipulates the target node.
3. The costs of each transformation form an alternating series.

Our TD splay procedure takes the tree apart during each step, and only reassembles it at the end of the procedure. The target node is only involved in the last step. Thus we cannot hope to directly apply the same proof technique to TD algorithms.

Fortunately, we can prove that TD procedures have a property called *factorability*, a property that will allow us bound their runtime above by an equivalent procedure which is more amenable to proof using a technique inspired by Subramanian [3].

Figure 10: Before the step

Before we define factorability, let's develop one more piece of notation. Let S denote a TD rule set and call the corresponding splay procedure P_S. Let $S_x(r)$ denote the process of applying P_S, searching for a node x, within the subtree rooted at r. This process begins at r.

Suppose that some binary search tree rooted at r is structured as shown in Figure 10, where the path from the root r to the target node x goes through some node z, such that the path $r \to z$ matches some template from S. P_S is *factorable* if executing $S_x(z)$ and then $S_x(r)$ (we denote this sequence as $S_x(z); S_x(r)$) produces the same tree as executing just $S_x(r)$. We will show that for any S that satisfies the rules outlined in section 3.3, P_S is factorable.

Let us describe the action of S on different sections of the tree rooted at r. Since S is a top-down generalized splay procedure, it adds to its left and right trees as it descends to node x in the center tree. Let the subtrees added to the left and the right during the step that descends from r to z be A and B, as shown in Figure 11

(a) Before (b) After

Figure 11: S's action on the $r \to z$ path. This occurs in one rule application.

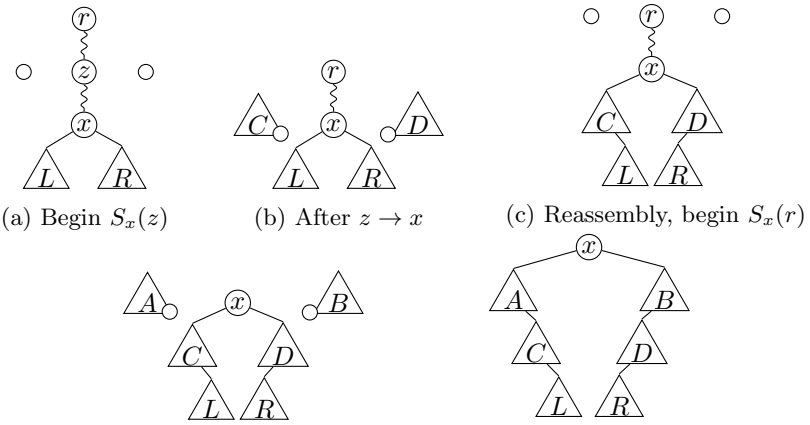
Let the left and right trees accumulated in the descent from z to x be C and D, as shown in Figure 12. These need not be produced in a single step of S.

(a) Before (b) After

Figure 12: S's action on the $z \to x$ path. This occurs over an unspecified number of rule applications.

Now that we have described the action of S on the $r \to z$ and $z \to x$ paths, we can analyze the effect of $S_x(z); S_x(x)$ and $S_x(x)$.

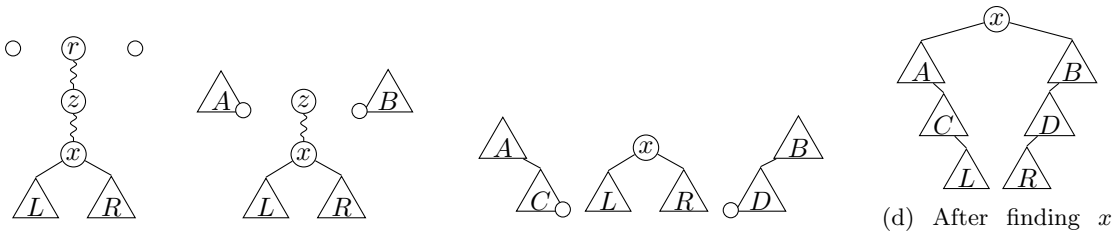
Figures 13 and 14 show that $S_x(z); S_x(r)$ and $S_x(r)$ produce equivalent trees. Also, since $S_x(z); S_x(r)$ does exactly one more step than $S_x(r)$ – one extra application of the reassembly rule – the figures show that the runtime of the latter is bounded above by the runtime of the former.



(a) Begin $S_x(z)$ (b) After $z \to x$ (c) Reassembly, begin $S_x(r)$

(d) After $x \to z$ (e) After reassembly.

Figure 13: The action of $S_x(z); S_x(r)$.



(a) Initial (b) After $r \to z$ (c) After $z \to x$ (d) After finding x triggers reassembly.

Figure 14: The action of $S_x(r)$.

4.2 The Potential Function and Consequences of Depth Reduction

We will use the same potential function used by Sleator and Tarjan. It is the function $\Phi(T)$ which is a sum over the nodes of the tree:

$$\Phi(T) = \sum_{n \in T} \text{rank}(n)$$

where the rank of n is defined in terms of *some* function mapping each node of the tree to a positive real. We call a function of this kind the *weight* function, and we define rank by

$$\text{rank}(n) = \log \text{sum}(n) \qquad \text{sum}(n) = \sum_{n' \in Sub(n)} \text{weight}(n')$$

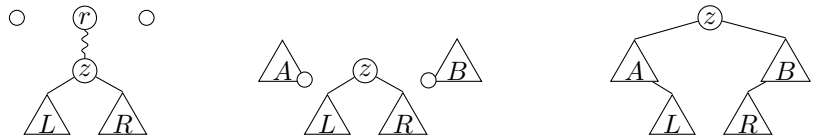
where $Sub(n)$ is the subtree rooted at n. An example of a weight function is the uniform weight, which assigns 1 to each node and turns the rank of a node into the logarithm of the size of its subtree.

We will show that if S is a general TD splay procedure, if a single depth-reducing rule is applied and then the left, center, and right subtrees are reassembled immediately, then the amortized cost of the application and reassembly is bounded above by

$$(d+1)(\text{rank}(r) - \text{rank}(z))$$

where r is the root of the subtree the rule's template, and z is action point of the template, and d is the depth of the template (so $d \leq k$).

Suppose that the tree initially looks like Figure 15a, then looks like Figure 15b after applying a template, and then like 15c after reassembly.



(a) Initial (b) After rule application (c) After reassembly

Figure 15: A single rule application followed by reassembly

Importantly, applying a single TD rule and then reassembling is equivalent to applying the dual BU rule (this is apparent from comparing Figures 15c and 6). Thus our task is really to show that applying a single depth-reducing BU rule has amortized cost bounded above by

$$(d+1)(\text{rank}(r) - \text{rank}(z))$$

which is exactly a result of Subramanian's [3]. **We repeat the argument here, but he should be given credit for the ideas of this section.**

We will use plain node names to indicate their location before the transformation, and primed names for their location after the transformation.

We begin with a few observations. The set of d nodes on the path from r to z (call them V) are the only nodes whose subtrees change as a part of this transformation. Thus any change in potential can be attributed to the change in rank of these nodes.

The rank of each node in V is bounded below by $\text{rank}(z)$, since z is in the subtree of all nodes along that path. Since $d = |V|$,

$$\sum_{v \in V} \text{rank}(v) \geq \sum_{v \in V} \text{rank}(z) = d\,\text{rank}(z) \tag{1}$$

After assembly, these nodes have moved and we, so we now call them V'. Notice that each of their ranks is bounded above by the rank of z', the root of the tree after the transformation, since each is in the subtree of z'. Furthermore, $\text{rank}(z') = \text{rank}(r)$, so we *could* argue that

$$\sum_{v' \in V'} \text{rank}(v') \leq \sum_{v' \in V'} \text{rank}(z') = \sum_{v' \in V'} \text{rank}(r) = d\,\text{rank}(r) \tag{2}$$

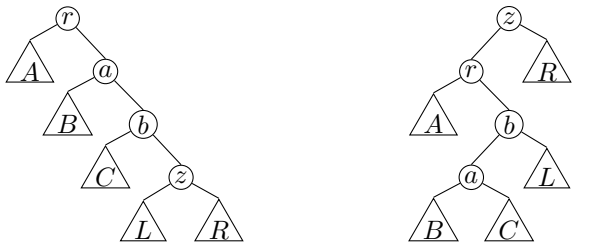
However, this would lead to us concluding that the amortized cost of the operation was bounded above by:

$$\underbrace{(d+1)(\text{rank}(r) - \text{rank}(z))}_{\Delta\Phi} + \underbrace{2}_{\text{Rule application \& reassembly}}$$

However, we can also provide a tighter bound through some case-by-case analysis. Let's say that applying the depth-reducing rule and reassembly rule causes the nodes of V' to end up in a stick.

4.2.1 Case 1: Stick

In this case the nodes on the path from r to z are rearranged into a stick (or path) in the tree. One example of this is the rule in Figure 16b.



(a) Before (b) After

Figure 16: A rule which produces a stick. In this rule, $V = \{r, a, b, z\}$

Since the result is a stick it has a unique bottom node, call it b'. Since the rule is depth reducing, z's child trees, L and R, must rise in the result. Since b' has the same depth in the result that z has in the template, L and R cannot be children of b'. Thus z and b' have disjoint subtrees. Thus

$$\text{sum}(z) + \text{sum}(b') \leq \text{sum}(r)$$

So

$$(\text{sum}(z) + \text{sum}(b'))^2 \leq (\text{sum}(r))^2$$

and since it is known that for real numbers a and b, $4ab \leq (a+b)^2$,

$$4\,\text{sum}(z)\,\text{sum}(b') \leq (\text{sum}(z) + \text{sum}(b'))^2 \leq (\text{sum}(r))^2$$

Taking logarithms yields:

$$2 + \text{rank}(z) + \text{rank}(b') \leq 2\,\text{rank}(r)$$

or

$$\text{rank}(b') \leq 2\,\text{rank}(r) - \text{rank}(z) - 2$$

We can now provide a tighter bound on the sum of ranks of V', by bounding all nodes in $V' \setminus \{b'\}$ in the prior manner (Eq. 2), and b' in the manner just described.

$$\sum_{v' \in V'} \text{rank}(v') \leq (2\,\text{rank}(z') - \text{rank}(z) - 2) + \sum_{v' \in V' \setminus \{b'\}} \text{rank}(r) = (d+1)\,\text{rank}(r) - \text{rank}(z) - 2$$

When we combine this with our earlier bounds on the ranks of the nodes before the transformation, we see:

$$\Delta\Phi = \sum_{v' \in V'} \text{rank}(v') - \sum_{v \in V} \text{rank}(v) \leq [(d+1)\,\text{rank}(r) - \text{rank}(z) - 2] - d\,\text{rank}(z)$$

$$\Delta\Phi \leq (d+1)(\text{rank}(r) - \text{rank}(z)) - 2 \tag{3}$$

Thus the amortized cost of a depth-reducing transformation that produces a stick is bounded like so:

$$c_a \leq (d+1)(\text{rank}(r) - \text{rank}(z)) \tag{4}$$

4.2.2 Case 2: Not Stick

In the case that the transformation does not arrange the nodes from V' into a stick, then there are two nodes $v', u' \in V'$ that are children of the same node in V'. Thus, the subtrees of v' and u' are disjoint. Since they're also both contained in the subtree of z',

$$\text{sum}(v') + \text{sum}(u') \leq \text{sum}(z')$$

The same manipulation from last section leads us to conclude that

$$\operatorname{rank}(v') + \operatorname{rank}(u') \leq 2\operatorname{rank}(z') - 2 \tag{5}$$

We can use this inequality to more tightly bound the change in potential during such a transformation.

$$\begin{aligned}
\Delta\Phi &= \sum_{v' \in V'} \operatorname{rank}(v') - \sum_{v \in V} \operatorname{rank}(v) \\
&\leq (2\operatorname{rank}(z') - 2) + \sum_{v' \in V \setminus \{u', v'\}} \operatorname{rank}(v') - \sum_{v \in V} \operatorname{rank}(v) \\
&\leq (2\operatorname{rank}(z') - 2) + (d-2)\operatorname{rank}(z') - d\operatorname{rank}(z) \\
&= d(\operatorname{rank}(z') - \operatorname{rank}(z)) - 2 \\
\Delta\Phi &\leq d(\operatorname{rank}(r) - \operatorname{rank}(z)) - 2
\end{aligned} \tag{6}$$

Which, since 2 steps were taken, gives rise to an amortized cost with the following bound:

$$c_a \leq d(\operatorname{rank}(r) - \operatorname{rank}(z)) \leq (d+1)(\operatorname{rank}(r) - \operatorname{rank}(z)) \tag{7}$$

4.2.3 Conclusion

Regardless of whether the depth-reducing rule produces a stick or not, the amortized cost of applying a single rule and then doing reassembly is bounded above by

$$c_a \leq (d+1)(\operatorname{rank}(r) - \operatorname{rank}(z)) \leq (k+1)(\operatorname{rank}(r) - \operatorname{rank}(z)) \tag{8}$$

where k is the upper bound on the depth of templates in the rule set.

4.3 Proving the Access Lemma

We will show that if S is a top-down splay procedure with a rule set that satisfies the constraints outlined in section 3.3, then $S_x(r)$ has amortized cost bounded above by $(k+1)\log N + e$, where k is the maximum depth of the rules in S's rule set, N is the number of items in the tree, and e is a constant.

We claim that the amortized cost is more precisely bounded above by

$$(k+1)(\operatorname{rank}(r) - \operatorname{rank}(x)) + e$$

and we will prove this bound by inducting on the number of rule applications needed to splay x.

Say that just one application of a rule is required. Then, since the template for each rule has bounded height, we can choose e to be large enough to bound the amortized cost of this rule application. Since the splay procedure brings x to the root of the tree, $(k+1)(\operatorname{rank}(r) - \operatorname{rank}(x))$ is positive, so

$$(k+1)(\operatorname{rank}(r) - \operatorname{rank}(x)) + e$$

upper bounds the amortized cost of $S_x(r)$.

Now say that the cost of $S_x(r)$ is bounded above by the desired expression when s or fewer rule applications are need. Now consider when $S_x(r)$ requires $s+1$ rule applications to compete. If this is the case then the first rule will have an action point corresponding to some z on the path from r to x. Since multiple steps are used, this action point must be a boundary action point, so the rule must be depth-reducing (by constraint 1 in section 3.3). Also recall that if z is the action point for the first rule during the search for x, then (a) $S_x(r)$ and $S_x(z); S_x(r)$ produce the same tree and (b) the cost of $S_x(r)$ is bounded above by $S_x(z); S_x(r)$. Thus the amortized cost of the former is

bounded above by the amortized cost of the latter. We now prove an upper bound on the amortized cost of $S_x(z); S_x(r)$.

By the inductive hypothesis $S_x(z)$ has amortized cost bounded above by $(k+1)(\text{rank}(z) - \text{rank}(x)) + e$. By our work in section 4.2, the subsequent $S_x(r)$ has amortized cost bounded above by $(k+1)(\text{rank}(r) - \text{rank}(z))$. Thus the total amortized cost of $S_x(z); S_x(r)$ is bounded above by

$$\begin{aligned} c_a(S_x(z); S_x(r)) &\leq [(k+1)(\cancel{\text{rank}(z)} - \text{rank}(x)) + e] + [(k+1)(\text{rank}(r) - \cancel{\text{rank}(z)})] \\ &= (k+1)(\text{rank}(r) - \text{rank}(x)) + e \end{aligned}$$

since $c_a(S_x(z)) \leq c_a(S_x(z); S_x(r))$, this means that

$$c_a(S_x(z)) \leq (k+1)(\text{rank}(r) - \text{rank}(x)) + e$$

which completes the inductive step.

Thus, for any x in a tree rooted at r, the amortized cost of splaying x to the root of the tree using some satisfactory TD splay procedure S is bounded above by

$$(k+1)(\text{rank}(r) - \text{rank}(x)) + e$$

which is in turn bounded above by

$$(k+1)(\text{rank}(r)) + e$$

Since k and e are constants associated with the algorithm, and r's subtree contains all N items in the tree, the amortized cost of the splay procedure is in

$$O(\log N)$$

5 Further Consequences of TD Splay's Amortized Cost

In the prior section we demonstrated that a given TD splay procedure from node r looking for node x has an amortized cost bounded above by

$$K(\text{rank}(r) - \text{rank}(x)) + e \in O(\text{rank}(r) - \text{rank}(x))$$

where $\text{rank}(n) = \sum_{n' \in Sub(n)} \text{weight}(n')$ and K, e are constants determined by the algorithm. This result holds for any positive-valued weight function.

In their seminal paper Sleator and Tarjan showed that the flexibility of this bound with respect to weight functions leads to a number of fantastic properties. **We repeat their arguments here (and the rest of this section should be credited to them)**.

We begin by framing our our analysis: consider a sequence of m splays on an n node tree with items $x_1, x_2, \ldots, x_n$. We define the sum of weights $W = \sum_{i=1}^{n} \text{weight}(x_i)$ and ask how much the potential of the tree might decrease during the sequence of operations. Since each node contributes at most $\log(W)$ to the potential and at least $\log(\text{weight}(x_i))$, the potential drops by at most

$$\sum_{i=0}^{n} \log(W) - \log(\text{weight}(x_i)) = \sum_{i=0}^{n} \log\left(\frac{W}{\text{weight}(x_i)}\right)$$

Since the total cost C of a sequence of operations is equal to the amortized cost, less the change in the potential, being able to bound the change in the potential from below (together with our prior work bounding the amortized cost from above) will provide upper bounds on the actual cost of a sequence of operations.

5.1 Five Hard Theorems [3]

We present five challenging but significant theorems which provide different perspectives of the cost of performing m accesses on an n item tree.

THEOREM 1 (BALANCE THEOREM). *The total access time is in*

$$O((m+n)\log(n+m))$$

PROOF. Define the weight function by $\text{weight}(x_i) = \frac{1}{n}$. Then $W = 1$ and the potential may decrease by at most $\sum_{i=1}^{n} \log\left(\frac{1}{1/n}\right) = n \log n$. The amortized cost of the operations is bounded above $mK\log(n) + me$, thus the total access time is bounded above by

$$mK\log(n) + me + n\log n$$

which is in $O((m+n)\log(m+n))$.

THEOREM 2 (STATIC OPTIMALITY THEOREM). *If every item x_i is accessed $q_i > 0$ times, the total access time is in*

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

PROOF. Define the weight function by $\text{weight}(x_i) = \frac{q_i}{m}$. Then $W = 1$ and the potential may decrease by at most $\sum_{i=1}^{n} \log(\frac{m}{q_i})$. Also, the amortized cost of accessing x_i is bounded above by $K(\log n - \log q_i) + e$ which is in turn bounded above by $K\log(m/q_i) + e$ since q_i being greater than 0 implies $m > n$. Thus the total cost is:

$$\sum_{i=1}^{n} \log(m/q_i) + \sum_{i=1}^{n} q_i[K\log(m/q_i) + e] = me + \sum_{i=1}^{n} q_i(K+1)\log(m/q_i)$$

which is in the desired set.

Call the item accessed in step j, i_j.

THEOREM 3 (STATIC FINGER THEOREM). *For any fixed item x, the total access time is in*

$$O\left(n\log n + m + \sum_{j=1}^{m} \log(|i_j - x| + 1)\right)$$

PROOF. Define the weight function by $\text{weight}(i) = (|i_j - f| + 1)^{-2}$. Then $W \leq 2\sum_{i=1}^{\infty} \frac{1}{i} = \pi^2/3$. Then the potential decreases by at most $n\log n$, and the access time for any single i_j is bounded above by $K(\log(\pi^2/3) + 2\log(|i_j - f| + 1))$. Thus the total access time is

$$n\log n + \sum_{i=1}^{m}[K(\log(\pi^2/3) + 2\log(|i_j - f| + 1))]$$

which is in the desired set.

Now we define a function t with respect to the access sequence. Let $t(j)$ be the number of distinct items accessed between access j and the last access of item i_j. If i_j has not been previous accessed, then define $t(j)$ as the number of distinct items accessed prior to access j. With this definition of t and a dynamically adjusted weight function we will prove our fourth result.

THEOREM 4 (WORKING SET THEOREM). *The total access time is in*

$$O(n\log n + m + \sum_{j=1}^{m} \log(t(j) + 1))$$

[3] The title of this section is inspired by the seventh chapter of Michael Spivak's *Calculus*: "Three Hard Theorems"[4]. While the theorems presented here may not be quite as hard, the name is too good to pass up on.

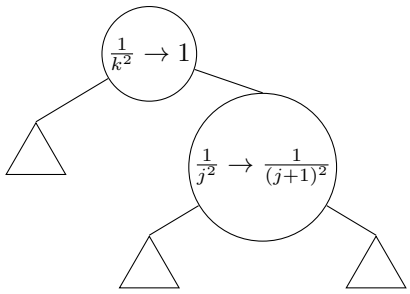


Figure 17: A visualization of the weight function being changed. The root's weight increases, no other weight does. The non-root changing node is intended to just be an example.

Define the weight function as follows. Initially the function is valued with respect to the order of first accesses for each item in the sequence. The first accessed item gets a 1, $\frac{1}{2^2}$ for the next, $\frac{1}{3^2}$ for the next, and so on down to $\frac{1}{n^2}$. If some items are not accessed, assign them the lowest weights in arbitrary order.

After an item with weight $1/k^2$ is accessed, change its weight to 1, and shift the weight of each item with weight $1/j^2$ where $1 < j < k$ to $1/(j+1)^2$. Notice that the weight-change just permutes the weights among the nodes. Also notice that this implies that whenever an item is accessed, say during access j, its weight is $\frac{1}{(t(j)+1)^2}$ where t maps the access j of item i_j to the number of distinct items that have been accessed since the last time i_j was accessed.

Our first concern is that this pattern of changing weights invalidates our analysis from earlier sections. Recall that we defined the amortized cost as the actual cost plus the change in potential. If transitioning the weight function causes potential to spontaneously increase, it might invalidate our analysis. However, it does not increase. When the weight function transitions, it increases the weight of the new root (from some $\frac{1}{k^2}$ to 1, and does not increase the weight of any other node, since their weights go from some $\frac{1}{k^2} \to \frac{1}{(k+1)^2}$ if they change at all). This situation is illustrated in Figure 17. Thus the net increase in the potential of the structure is bounded above by the amount the rank of the root might change during this transition. However, since the transition merely permutes the weights, the sum of weights in the tree is invariant, and so the rank of the root is invariant. Thus the potential of the structure cannot increase during the transition, and our early analysis demonstrating that the amortized cost of a splay operation is bounded above by $K(\text{rank}(r) - \text{rank}(x)) + e$ is correct.

Given this, we observe that the sum of all weights, $W = \sum_{i=1}^{n} 1/i^2$ is bounded above by $\sum_{i=1}^{\infty} 1/i^2 = \pi^2/6$[4]. This means that the potential of the structure decreases by at most $\sum_{i=1}^{n} \log(W/\text{ weight}(i)) \leq \sum_{i=1}^{n} \log(\pi^2 n^2/6) \leq 2n \log(n\pi^2/6)$. Furthermore, the net amortized access time is bounded above by

$$\begin{aligned}
\sum_{j=1}^{m} K(\text{rank}(r) - \text{rank}(i_j)) + e &= \sum_{j=1}^{m} -K \log(\frac{1}{(t(j)+1)^2}) + (K \log(W) + e) \\
&= m(K \log(W) + e) + 2K \sum_{j=1}^{m} \log(t(j)+1) \\
&\leq mK \log(\pi^2/6) + me + 2K \sum_{j=1}^{m} \log(t(j)+1)
\end{aligned}$$

[4] $\sum_{i=1}^{\infty} \frac{1}{i^2}$ is known to converge absolutely to $\frac{\pi^2}{6}$.

So the net actual access time is bounded above by

$$2n\log(n\pi^2/6) + mK\log(\pi^2/6) + me + 2K\sum_{j=1}^{m}\log(t(j)+1) \in O\left(n\log n + m + \sum_{j=1}^{m}\log(t(j)+1)\right)$$

THEOREM 5 (UNIFIED THEOREM). *The total time of m accesses on an n node splay tree is in*

$$O\left(n\log n + m + \sum_{j=1}^{m}\min\left\{\frac{m}{q(i_j)}, |i_j - x| + 1, t(j) + 1\right\}\right)$$

where i_j, x, and t are defined and quantified as they were previously. This follows from the previous 3 theorems.

5.2 Interpretation

To remind the reader of the significance of these theorems, we quote the apt words of Sleator and Tarjan:

> Let us interpret the meaning of these theorems. The balance theorem states that on a sufficiently long sequence of accesses a splay tree is as efficient as any form of uniformly balanced tree. The static optimality theorem implies that a splay tree is as efficient as any fixed search tree, including the optimum tree for the given access sequence, since by a standard theorem of information theory [..] the total access time for any fixed tree is $\Omega(m + \sum_{i=1}^{n} q(i)\log(m/q(i)))$. The static finger theorem states that splay trees support accesses in the vicinity of a fixed finger with the same efficiency as finger search trees. The working set theorem states that the time to access an item can be estimated as the logarithm of one plus the number of different items accessed since the given item was last accessed. That is, the most recently accessed items, which can be thought of as forming the "working set," are the easiest to access. All these results are to within a constant factor.
>
> Splay trees have all these behaviors automatically; the restructuring heuristic is blind to the properties of the access sequence and to the global structure of the tree. Indeed, splay trees have all these behaviors simultaneously; at the cost of a constant factor we can combine all [first] four theorems into [the fifth]. [1]

6 Implementation

Both the top-down general splay procedure described in this document and the bottom-up general splay procedure described in Subramanian's work [3] were implemented in C++. The implementation may be found in the repository at `github.com/alex-ozdemir/splay`. The implementation provides TD and BU splay operations which take

1. The tree to search in
2. An item to search for
3. A representation of the rules to use

and use these to perform the corresponding splay search on the input tree. Save this side-effect, the generalized splay procedures have no other output.

A benchmark script was written that generated the *complete* rule set for each natural number k. The complete rule set for k matches paths of length up to k and turns them into a result which (a) puts the target node at the root and (b) balances the left and right trees optimally.

The benchmarks were run on initially complete binary trees of height 14[5]. For each rule set, $2^{22} = 4.2 \times 10^6$ lookups were done for items drawn from a uniform random sequence that was seeded identically for each benchmark.

The results in Figure 18 indicate that for both the TD and BU variants the performance increases with k until k is comparable to the initial depth of the tree, after which performance degrades. Interestingly, the original TD and BU rules perform similarly to the complete rules of height 2, and worse than the complete rules of greater height. This should not be taken *too* seriously – it is possible that when implemented in a non-generic fashion the original rules optimize better than rules of greater height. Further, empirical, investigation is required here, but the potential for rule sets that perform better than the original splay rule set is quite exciting.

The results also lead to another natural question: Does the following heuristic provide amortized logarithmic lookup time?

1. Traverse the path to the target node.
2. Rearrange the nodes on the path into a tree with the target at the root and optimally balanced left and right subtrees.[6]

The TD variant also appear faster than their TD duals across the board, but that may just be an artifact of an implementation which unintentionally favors them.

7 Conclusion

We have shown that if a set of TD rules satisfy the following properties:

1. The boundary rules are all depth reducing.
2. The set of templates have heights bounded above by some constant k.

then the TD splay procedure that uses these rules has amortized runtime in $O(\log N)$, and has actual runtime that satisfies a number of other fantastic properties listed in Section 5.

The generality of our algorithm proves the existence of a large (in fact infinite) set of TD splay procedures which produce this behavior.

8 Open Problems

8.1 Top Splay Trees

In this paper we've explored top-down splay trees. The splay procedures of this class of data structures tear apart the tree during a traversal to the target node, and then re-assemble the tree afterwards.

However, one might imagine another type of splay procedure which begins at a trees root: what we'll call a *top splay procedure*. Such a procedure iteratively rearranges the nodes near the root in such a way as to eventually pull the target node to the root. One such procedure is described and illustrated in Figure 19.

There are two questions of interest here:

[5]The careful reader might wonder whether rules matching paths of length greater that 14 would ever be used, given the tree starts with height 14. Such rules could be used once the tree starts being manipulated, potentially leaving paths of length greater than 14.

[6]Using something like the Day-Stout-Warren algorithm.

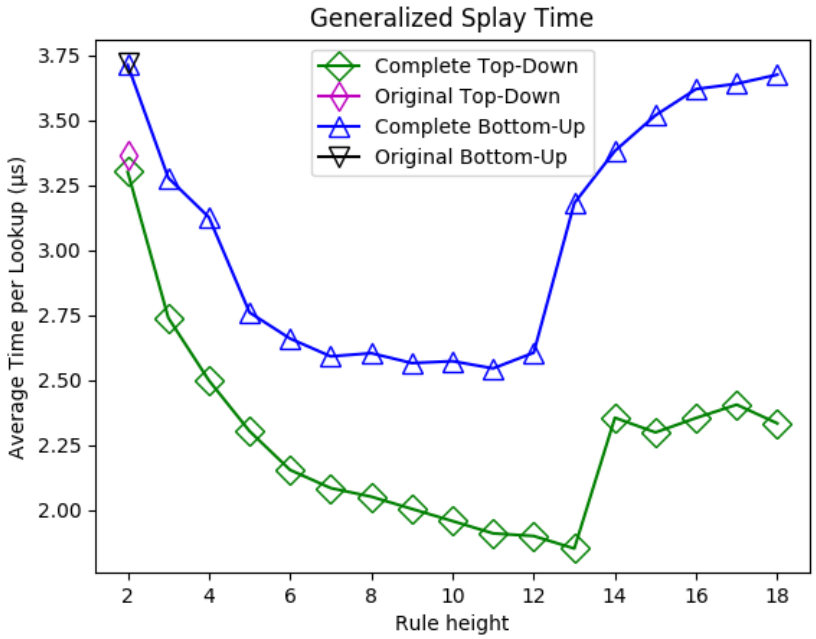
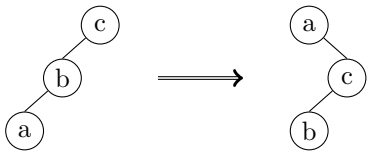


Figure 18: Performance of TD and BU complete rules. The Original TD and BU rulesets are also shown for comparison.

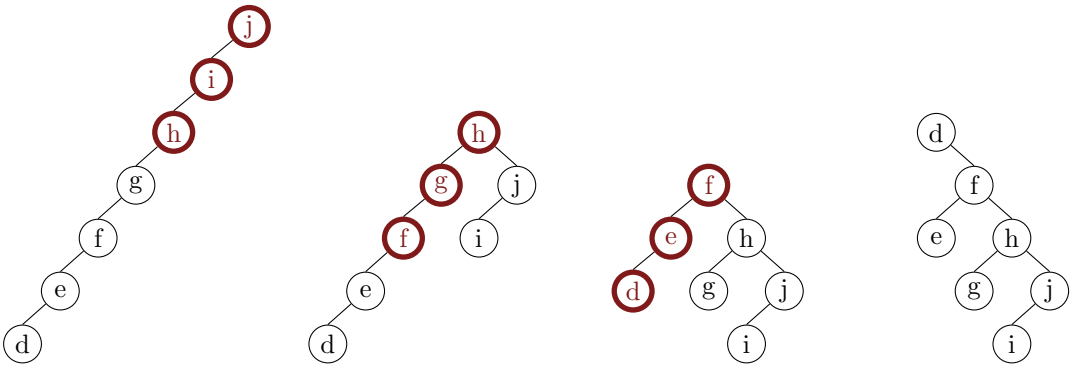
1. Does the specific procedure described in Figure 19 have runtime in $O(\operatorname{rank} r - \operatorname{rank} x)$? Does it run in amortized logarithmic time?

2. For what kinds of rulesets does a generalized top splay tree run in amortized logarithmic time?

Both the original Bottom-Up and this proposed Top splay were implemented in a fashion that allowed for (a) efficient tracking of the deepest element to facilitate worst-case testing and (b) reporting of the number of rotations done in splay operations. These implementations were used to compare how the worst-case number of rotations per splay scaled with the since of the tree. The results are shown in Figure 20. The implementation can be found at `github.com/alex-ozdemir/splay`.



(a) A rule used by this algorithm (the only relevant rule in this example)



(b) Search for d begins (c) $h \rightarrow$ root (d) $f \rightarrow$ root (e) $d \rightarrow$ root

Figure 19: A Top Splay Tree which searches 2-levels down the tree for the target node (in this case d). If it doesn't find the target or a null terminator, it rotates whatever it did find to root using bottom-up rotations, and continues to the next iteration.

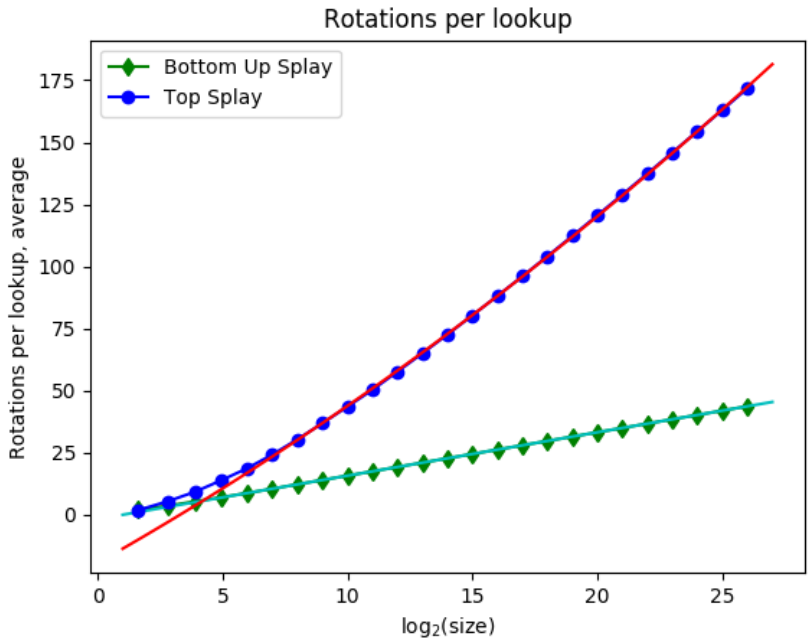


Figure 20: How the original splay procedure and the top-splay procedure scale, empirically.

These results suggest that Top-Splay is competitive with the original Bottom-Up-Splay to within a constant factor, or a function that grows slowly. Alas, top-splay remains open as a theoretical question.

8.2 The Path Balance Heuristic

Another rebalance heuristic one might imagine does not focus on bringing the target node to the root, but instead focuses on balancing as perfectly as possible. The procedure traverses the tree down to the target node (or a null terminator) and then optimally balances the nodes on the path it traversed. This is illustrated in Figure 21.

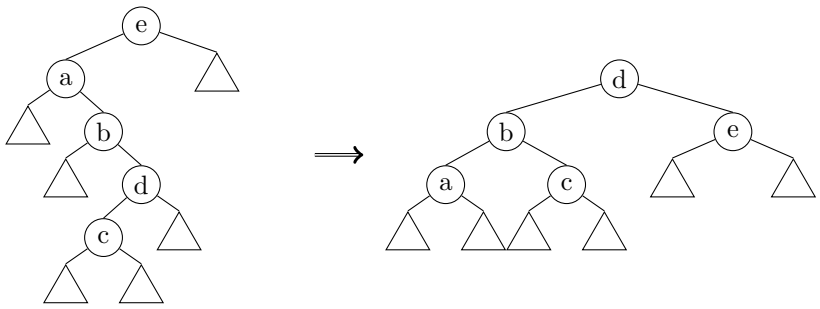


Figure 21: Node c is searched for and found. After locating c, the path to c is rearranged into an optimally balance tree.

Balasubramanian [5] showed that heuristic, the *Path Balance Heuristic* had amortized runtime in $O\left(\frac{\log n \log\log n}{\log\log\log n}\right)$. However, it is still is an open question whether this heuristic has amortized runtime in $O(\log n)$, and whether is satisfies the other theorems from Section 5.

8.3 The Path Rebalance Heuristic with Move To Root

Another natural heuristic is the one proposed at the end of Section 6. That is a heuristic which:

1. Traverses the path the the target node.
2. Rearranges that entire path into a tree such that
   (a) The target node is at the root.
   (b) The items less than the target are left of it and form an optimally balanced tree.
   (c) The items greater than the target are right of it and form an optimally balanced tree.

Does this heuristic provide a logarithmic amortized bound on access? From the experimentation in Section 6, it seems to perform well in practice relative to other template-based approaches.

9 Acknowledgments

Many people contributed to the development of these ideas in different ways.

To begin with I'd like to thank Calvin Leung who, in the wee hours of one sophomore morning, incorrectly implemented a splay tree with me. That misstep, noticed by Adam Dunlap, lead to questions and discussions which would ultimately motivate this thesis.

This work also benefited greatly from the revisions proposed by Chris Stone, who has been a *fantastically* good reviewer.

Finally, I'm eternally in the debt of Ran Libeskind-Hadas who teaches and advises with incredible enthusiasm, inspiring a love of algorithms, proof, and research. Without him I never would have gotten into algorithms to begin with, and it has only been through his guidance that this thesis has come to completion.

References

[1] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

[2] Erkki Mäkinen. On top-down splaying. *BIT Numerical Mathematics*, 27(3):330–339, 1987.

[3] Ashok Subramanian. An explanation of splaying. *Journal of Algorithms*, 20(3):512–525, 1996.

[4] M. Spivak. *Calculus*. Addison-Wesley World Student Series. W. A. Benjamin, 1973.

[5] R. Balasubramanian and Venkatesh Raman. *Path balance heuristic for self-adjusting binary search trees*, pages 338–348. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.