

Formally Verified Amortized Cost Analysis of Splay Trees in Agda

Lukas Kebuladze

August 2025

1 Introduction

Splay trees are a classic and well-studied example in amortized analysis. In this paper, we present a mechanization of amortized cost analysis for splay trees in a dependent type theory. Our goal for this analysis is to formally prove the following theorem:

THEOREM 1 The splay operation on a tree of n nodes has amortized cost $O(\log n)$

We will first provide background on the subject and similar work, detail the implementation of splay trees in Agda, explain the process for correctness and cost verification, and finally discuss future work.

1.1 Splay Trees

A splay tree is a self-adjusting binary search tree (BST). It achieves its logarithmic amortized cost through the splay operation which takes as input a tree T , key k , and returns a new tree T' such that k is the root of T' [8]. In general, for operations beyond just the splay operation, splay trees preserve two properties: (1) most recently used key is rotated to the root and (2) BST property is preserved (for every node x in the tree, all nodes to the left of x have keys less than x and vice versa for all nodes to the right of x).

An important part of the splay operation is the splay step which specifies how the tree should rotate to move the key k to the root based on the structure of the input tree. The splay step is divided into six different possible cases. Because of the symmetry in these cases, we only have to care about three of them, namely zig, zig-zig, zig-zag. Keep in mind that this is just one step, and that the splay operation is a repetition of these splay steps until the requested key is at the root of the tree. The figures below depict these three cases.

Let x, y, z be keys of a tree T , a, b, c, d be sub-trees, and x the key we are splaying to the root of the tree.

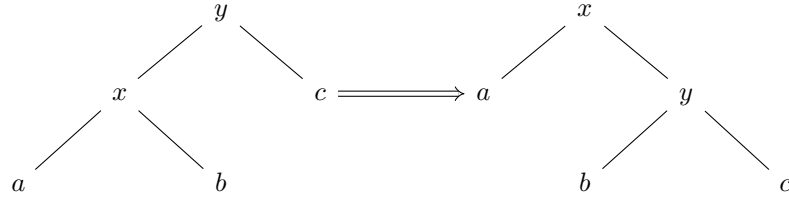


Figure 1.1.1 zig case

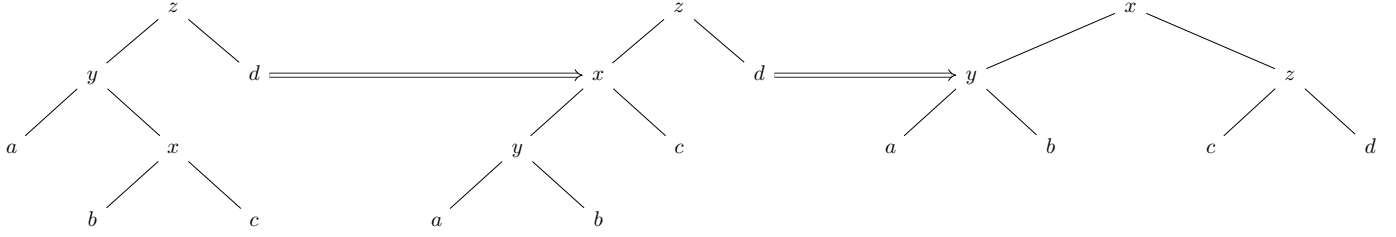


Figure 1.1.2 zig-zag case

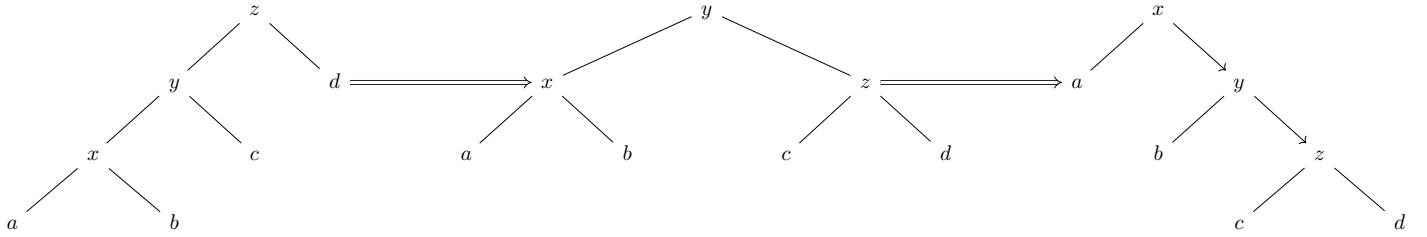


Figure 1.1.3 zig-zig case

Note that zig-zag and zig-zig (and the symmetric cases zag-zig and zag-zag) consist of two rotations instead of one like zig (and the symmetric case zag). However, in the context of counting splay steps, the double rotation cases still count as a single splay step.

1.2 Amortized Analysis

In computer science, the cost of data structures and algorithms is typically expressed in terms of Big-O notation considering worst-case cost. However, only considering the worst-case cost can be ineffective when considering operations on certain data structures. A classic and simple example of such a data structure is dynamic arrays [9]. In particular, consider the following specification where we maintain an array of some capacity $c \geq 1$ in which the first n of the slots are the actual list elements (which will be strings for simplicity), and the remaining slots are free space for future elements, so $c \geq n$

```
empty : unit -> array Create an array with capacity 1. ( $n = 0, c = 1$ )
insert : array -> string -> array If  $c = n$  then grow A where A is the input array. Write  $x$  at
position  $n$  and increment  $n$ 
nth : array -> int -> string Return the  $i$ th element of the array
grow : array -> array Double  $c$  and create a new array with capacity  $c$ , move every element from the
old into the new array
```

The operation of interest in this case is **insert**. If $c < n$, then we have an $O(1)$ operation since we are just writing a single element x . However, in the worst-case we must spend $O(n)$ time moving all the elements to a new array. In a classical analysis setting this would result in **insert** being considered an $O(n)$ operation. However, this does not realistically represent the cost of our algorithm, since this worst-case scenario happens very infrequently. More formally, after triggering the worst case, it is impossible to hit the worst case again until $c/2$ appends.

This is where amortized analysis comes in. Instead of considering the worst-case of any single operation, we consider the worst-case of a sequence of operations on a data structure.

There are many methods to approach this analysis, but the one of the most interest and relevance is known as the potential method.

DEFINITION 1.2.1 The Potential Method. Consider a sequence of m operations $\sigma_1, \sigma_2, \dots, \sigma_m$ on the data structure. Let the sequence of states through which the data structure passes be S_0, S_1, \dots, S_m . Notice that operation σ_i

changes the state from S_{i-1} to S_i . Let the actual cost of operation σ_i in the cost model be c_i . Given a potential function Φ , which is a function that maps a state S to some real number, we then define the amortized cost of c_i of operation σ_i by the following formula:

$$ac_i = c_i + \Phi(S_i) - \Phi(S_{i-1})$$

In short, (amortized cost) = (actual cost) + (change in potential). When we consider the total cost over m operations, we get the following by telescoping and some rearrangement:

$$\sum_i c_i = (\sum_i ac_i) + \Phi(S_0) - \Phi(S_m)$$

If $\Phi(S_0) \leq \Phi(S_m)$, we get

$$\sum_i c_i \leq \sum_i ac_i$$

This means that over a sequence of m operations, our actual cost is bounded by the amortized cost which is the desired result. Using the method applied to dynamic arrays with choosing the potential function as $\Phi(n, c) = 2(n - \frac{n}{2})$ allows us to show that the amortized cost of dynamic arrays is 3, which is far better than the original worst-case $O(n)$ bound.

1.3 Directed Effectful Cost-Aware Logical Framework (decalf)

The decalf, which is an extension of calf [7], language captures both extensional (behavioral) and intenstional (cost) properties of programs [3]. Intergrating cost through computational effects allows programs written using decalf, which is embedded in Agda, to have their cost internally tied to the implementation, rather than being an external property.

Another key part of decalf is call-by-push-value (CBPV) which is a programming language paradigm that is based on the slogan “a value is, a computation does” [5]. Furthermore, the types are separated into positive types (values) and negative types (computations).

In relation to this paper, decalf is especially important as it equipped to deal with inequalities, as well as equalities, with respect to comparing programs. But what does it mean to have two programs A, B with say $A \leq B$?

We formulate cost through $\mathbf{step}_X^c(e)$, an instruction and a computational effect, that is parameterized by a computation type X and an element of the cost type c . In plain English, this means that we charge c cost for e . To use \mathbf{step} to reason about comparison between programs, we must consider monotonicity. As a simple example, consider $a, b, c \in \mathbb{N}$ with $a \leq b$. We can then conclude that $a + c \leq b + c$ since addition is monotonic. In a similar way, using the fact that \mathbf{step} is monotonic, we can say if $c \leq c'$, then we have

$$\mathbf{step}_X^c(e) \leq \mathbf{step}_X^{c'}(e)$$

for any computation of type $e : X$. If we let e be some program, we now have a way to compare two programs that differ in cost. As will be seen in the discussion of the cost analysis, this is precisely what we will use as a basis for proving our desired cost theorem that was previously stated.

1.4 Related Work

1.4.1 Automatic Amortized Resource Analysis (AARA)

The purpose of AARA is to automatically infer symbolic bounds on the cost of a program by automating the potential method [4]. It works by trying to solve for the potential function, and then showing that the generated function satisfies the symbolic bounds it inferred. It has worked on a variety of examples for varying complexity levels, deriving cost bounds for data structures such as lists and batched queues. However, on splay trees it fails to come up with a bound, since the analysis for splay trees heavily depends on the rank of the input tree as will be seen in section 4.

1.4.2 Denotational Recurrence Extraction for Amortized Analysis

This work focuses on using recurrence solving in order to derive amortized analysis bounds [1]. Since many amortized data structures use induction or recursion in some way, this method is a general framework for amortized analysis. In this paper, splay trees were used as an example of this framework being successful. The drawback is that the analysis done was pen-and-paper, so there is a lack of formal verification.

1.4.3 Amortized Complexity Verified

In this paper, the amortized cost of splay trees is formally verified in Isabelle using Higher Order Logic [6]. Because this is outside of dependent type theory, the cost accounting is done externally from the program itself. Additionally, the use of Higher Order Logic makes the proof not easily composable, so proving downstream/client level theorems about splay trees would be not as easy as in a dependent type theory.

1.5 Splay Trees in decalf

From the related works section, we can see that while there is extensive work done in formal cost analysis for splay trees, a formal verification that is composable and has cost internal to the implementation in a dependent type theory proof assistant is not yet present. This work serves to fill that gap, through the use of decalf in Agda. Thus, this paper strongly demonstrates the validity and feasibility of complex amortized analysis in decalf.

2 Implementation

The implementation we chose for the splay operation was a bottom-up method inspired by Sam Westrick’s implementation of splay trees in SML¹ which is based on the original implementation [8]. It consists of first searching the tree to find the key to splay up to the root while maintaining a list of ancestors, then recursively splaying up the key to the root by induction on the list of ancestors.

2.1 Implementation Specific Terminology

In order to better communicate certain aspects specific to the implementation, here we will outline key definitions and functions essential to the Agda implementation.

2.1.1 Definitions

BST Definition

We equip the abstract BST structure with the carrier being a positive value type T and the operation as `splay`. Here, `splay` takes in a element of T , a natural number (specified by `nat`) k and returns a computation of a pair with the first element being the new root of the tree and the second element being the splayed tree.

```
record BST : Set where
  field
    T : tp+
    splay : cmp (II T (λ _ → II nat (λ _ → F (nat ×+ T))))
```

Tree Definition

The first tree definition is a standard definition of a functional tree. In a proper abstraction of trees, keys can be other than the natural numbers, but for the purposes of this analysis, natural numbers will suffice. The second is to convert the Agda type into a decalf positive type.

```
data Tree : Set where
  leaf : Tree
  node : Tree → val nat → Tree → Tree
```

¹<https://github.com/shwestrick/pure-splay/blob/main/BottomUpSplay.sml>

```
tree : tp+
tree = meta+ Tree
```

Context definition

This definition is used for the list of ancestors that is part of our implementation of splay trees.

```
data Context : Set where
  Left : (k : val nat) (t : Tree) → Context
  Right : (t : Tree) (k : val nat) → Context

context : tp+
context = meta+ (Context)
```

2.1.2 Helper Functions

`path` and `reconstruct`

Since `path` deconstructs the input tree by storing a list of ancestors, we need to have a way to invert that, which is precisely what `reconstruct` does. Additionally, because we use `path` within the implementation for the splay operation, we need some correctness criteria which is why we have `pathInordType`. This type specifies our correctness criteria for `path`. It is a tuple that contains a pair of the result tree and the list of ancestors, a proof that the reconstruction of the input tree and input ancestor list is same as the reconstruction of the output tree and the output ancestor list, and a proof that root of the output tree is the requested key `k`. As a note, the `root` function takes in two arguments (`t` and `k`) instead of just `t` in order to deal with the edge case where `t` is a leaf and therefore does not have any root. This case only happens when `k` is not in the tree so it's not interesting.

```
reconstruct : (t : Tree) (anc : List Context) → Tree
reconstruct t [] = t
reconstruct t (Left x r :: anc') = reconstruct (node t x r) anc'
reconstruct t (Right l x :: anc') = reconstruct (node l x t) anc'

pathInordType : val nat → Tree → List Context → tp+
pathInordType k t anc = Σ+ pathType (λ (t', anc') →
  (meta+ (reconstruct t anc ≡ reconstruct t' anc')) ×+ meta+ (root t' k ≡ k))

path : (k : val nat) (t : Tree) (anc : List Context) → cmp (F (pathInordType k t anc))
path k leaf anc = ret ((leaf, anc), refl, refl)
path k (node l x r) anc with j-cmp k x
... — tri< _ _ _ = path k l (Left x r :: anc)
... — tri≈ _ k≡x _ = ret ((node l x r, anc), refl, Eq.sym k≡x)
... — tri> _ _ _ = path k r (Right l x :: anc)
```

`inord`

The inorder traversal of a BST, or for short `inord`, is how we determine equality between trees. More formally, if trees T and T' satisfy `inord T ≡ inord T'`, we consider T and T' equivalent. This is especially important in the analysis of the correctness of splay trees abstractly as we want to ensure we are not altering the tree in any way from an outside perspective. Of course the internal shape of the tree may change but to a client using a splay tree abstractly that does not matter.

```
inord : Tree → val (list nat)
inord leaf = []
inord (node l z r) = inord l ++ z :: [] ++ inord r
```

`splay` ,

`SplayTree` `BST.splay` is the top level splay function that sequences the helper functions `path` and `splay`. Then, `splay` either immediately returns if `path` failed to find the key in the input tree, otherwise calls `splay'` which actually performs the splay operation.

Instead of taking in one tree as input, `splay'` takes in two, `a` and `b`. These represent the left and right subtrees respectively of the tree that currently has `k` as its root. With this setup, once we are done splaying (i.e. the list of ancestors is empty) we can just return the two trees `a`, `b` and join them with `k` in the middle in the top level functions for the splay operation.

Based on the length and contents of the ancestor list, `splay'` does a corresponding rotation(s). Then, it returns the rotated tree as well as a corresponding list associativity proof with respect to `++` (append) to show inorder traversal preservation. Here is an example of one such case.

```
splay' a b (Left p c :: []) k =
  step (F _) 1 (
    ret ((a , node b p c) , ++-assoc (inord a) (k :: inord b) (p :: inord c)))
```

Since our analysis only concerns the actual work done by the splay operation, we only need to consider the helper `splay'` for the rest of this paper.

```
splay'ResultType : val nat → Tree → Tree → List Context → tp+
splay'ResultType k a b anc = Σ+ (tree ×+ tree) (λ (a' , b') →
  meta+ (inord (reconstruct (node a k b) anc) ≡ inord (node a' k b'))))

splay' : (a : Tree) (b : Tree) (anc : List Context) (k : val nat) → cmp (F (splay'ResultType k a b anc))
```

3 Correctness

3.1 Correctness Criteria

Let t be any arbitrary tree, K be the set of keys in t , key $k \in \mathbb{N}$ such that $k \in K$, t' be the tree after performing `SplayTree` `.BST.splay` t k . For `splay` to be considered correct, we consider the following two criteria.

1. `inord t` \equiv `inord t'`
2. `root t'` $\equiv k$

The first criterion ensures maintenance of the BST property (since we assume that the input tree has the BST property), no nodes are added or deleted, and all keys are in the same order as in the input tree. The second criterion makes sure that our result tree actually contains the key as its root. As mentioned before, `root` takes in a key as its second argument in the case that t is a leaf and therefore has no root.

3.2 High-level Approach

The idea to prove these two criteria is to embed relevant proofs into the return types of each of the helper functions involved in `SplayTree` `.BST.splay`. This can be seen above in section 2.2 by examining the return types of each of the result type specifications for the helper functions. In these result types, the first element of the return tuple is the actual computed value, with the remaining elements proofs of certain criteria.

To prove the first criterion, we simply sequence the `inord` embedded correctness criteria from `path`, `splay`, and `splay'`. However, since `splay'` does not invoke any helper functions, it requires actual work to be done. For each case in `splay'`, we must show that inorder traversal is preserved in the tree across a single splay step. This is done by doing a short list append associativity proof for each case.

The proof for the second criterion is easier. The criteria from `path` tells us that the resulting output tree has k as its root if it exists in the tree. Then, we can use the last two tuple elements from the return of `splay` to show that the key at the root of the splayed tree is equivalent to the input key.

4 Cost

4.1 High-level Approach

As mentioned in section 3 of the Amortized Analysis via Coalgebra (AAVC) paper, when wanting to show amortized cost with a bound instead of a strict equality, a lax homomorphism is required as depicted below [2].

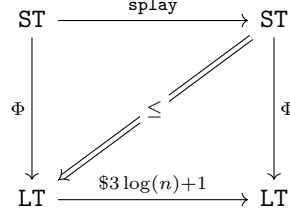


Figure 4.1.1 lax homomorphism diagram

Note: ST stands for splay tree, LT stands for list tree, and n is the number of nodes in the input tree

Essentially, this diagram is saying that the cost of a `splay` and Φ is bounded by the cost of Φ and the identity function with cost, which in this case is just charging a cost of $3 \log(n) + 1$.

Recall our earlier Agda definition for BST. It is designed as an algebraic structure, with the carrier being T and the set of operations being `splay`. Thus, any two implementations that ascribe to BST (such as ST and LT) can have morphisms between them, which is how we can frame this analysis as a homomorphism. The lax part just comes from the fact that we are showing a bound between the cost rather than a strict equality.

In the case of splay trees, the reference implementation is just a list with `splay` being implemented as the identity function annotated with cost. To simplify the analysis, we chose to omit the invocation of the `ListTree.BST.splay` and instead just charge the cost when formalizing our homomorphism in Agda. This can be seen in the central cost theorem (`splay' / amortized`) statement as follows:

```
splay' / amortized : (l : Tree) (r : Tree) (anc : List Context) (k : val nat) →
  bind (F _) (splay' l r anc k) (λ ((l', r'), _) → ϕ (node l' k r'))
≤ [ F _ ]
  step (F _) (1 + 3 * (rank (reconstruct (node l k r) anc) - rank (node l k r))) (ϕ (reconstruct (node l k r) anc))
```

This theorem is an encoding of Figure 4.1.1 above (except we replace the calls to `rank` with just $\log n$). Similar to the cost verification done for Batched Queues in Agda, following this theorem would come higher level theorems that bound the cost of a series of operations, but as will be discussed in more detail in the future work section, such theorems are relatively easy to show [2].

Thus, the cost verification process will aim towards completing the `splay' / amortized`.

4.2 Cost Model

In order to effectively discuss the proof of `splay' / amortized`, we must first outline the cost model and other helper functions pivotal to specifying cost.

Similar to the original analysis done by Sleator and Tarjan, our cost comes from charging unit cost at each invocation of a `splay` step [8]. In terms of the implementation, this means invoking `step` with a cost of one for each case of `splay'`. Here is an example from section 2.1.2 demonstrating this charging per unit cost per `splay` step

```
splay' a b (Left p c :: []) k =
  step (F _) 1 (
    ret ((a, node b p c), ++-assoc (inord a) (k :: inord b) (p :: inord c)))
```

To express the cost of the transition function Φ , we use the following helper functions:

`rank` and `sum-of-ranks`

Our goal is to have the cost charged by Φ to be high when the input tree is unbalanced and low when the tree is balanced. This goal motivates how we express **rank**. To demonstrate this, we will analyze the **sum-of-ranks** applied to a left/right spine and a perfectly balanced tree.

```
rank : (T : Tree) → val nat
rank t = ⌊log2 (tree-size t)⌋
```

```
sum-of-ranks : (T : Tree) → val nat
sum-of-ranks leaf = 0
sum-of-ranks (node l z r) = sum-of-ranks l + rank (node l z r) + sum-of-ranks r
```

Using these two functions, we can now specify the transition function.

```
ϕ : cmp (Π tree λ _ → F (list nat))
ϕ t = step (F _) (sum-of-ranks t) (ret (inord t))
```

In classical amortized analysis, the potential function simply outputs a real number representing the cost. The key difference here is that this transition function, which represents the homomorphism, charges the cost while also returning a value, in this case the inorder traversal of the input tree. With these functions, we can proceed with detailing the process for proving **splay' / amortized**.

4.3 The Rank Rule

An important lemma used in the cost analysis is the rank rule, which states that if two siblings in a tree have the same rank r , then the parent must have rank at least $r + 1$. While this theorem is quite simple and the pen-and-paper proof is short, the proof in Agda turned out to be much more difficult. This is due to the lack of many logarithm theorems in the Agda standard library. The point here is to demonstrate a common pattern in the cost verification where the actual logic will be very similar to a pen-and-paper proof, but the arithmetic is a major factor in increasing the size of the mechanization in Agda. Although it is important to note that this arithmetic mechanization was made significantly easier through the use of the Agda ring solver.

4.4 Proving **splay' / amortized**

To prove the theorem, we split into seven cases based on the seven cases in the implementation for **splay'**. One of the cases is trivial since there is no rotation done, and three of the six cases are exactly symmetrical. Thus, the work is really done in the zig, zig-zag, and zig-zig cases. Because the logic is very similar to what is shown in the proof done by Sleator and Tarjan, the following sub-sections will focus on the key differences between the pen-and-paper proof and the Agda mechanization [10].

4.4.1 Induction versus Splay Step Analysis

Before getting into each of the cases, it is important to note a key difference in proof approach between the original proof by Sleator and Tarjan and the cost proof done here. In the original proof the access lemma, which is very similar to **splay' / amortized**, is not proven directly, but rather a result of telescoping the result given by the splay step lemma. For reference, both of those lemmas will be shown here.

LEMMA 1: Access Lemma. Take any tree T with root t . Suppose you splay node x ; let T' be the new tree. Then

$$(\text{amortized number of splaying steps}) = (\text{actual number of splaying steps}) + \Phi(T') - \Phi(T) \leq 3(r(t) - r(x)) + 1$$

LEMMA: Cost of a splay step. The amortized cost of a zig-zag or zig-zig splay step is $3(r(z) - r(x))$. The cost of the zig splay step is $3(r(y) - r(x)) + 1$.

Note that r here refers to `rank`, y is the parent of x , and z is the grandparent. It is clear that LEMMA 1 is very similar to `splay'/amortized`, with the major change being that we move the $\Phi(T)$ to the other side of the inequality in `splay'/amortized`.

In contrast to the original proof, our analysis proves `splay'/amortized` directly, using induction which makes the telescope from the original proof explicit. However, in each case of the proof for `splay'/amortized`, once we apply the induction hypothesis, we proceed with what is essentially a proof of the splay step lemma. Thus, we are able to take advantage of the proof structure used in the original proof for the cost of a splay step lemma.

4.4.2 The Zig Case

We first start with the simplest of the three cases, since zig is just a single rotation which means that it is essentially a base case, so no application of the induction hypothesis is required.

We must first apply our correctness criteria to show that the `inord` of the input tree is the same as the output tree. This is because right hand side of our inequality for `splay'/amortized` applies Φ to the tree before the rotation, and the left hand side uses the tree after the rotation for Φ .

Once we have done that, we must extract the cost out of $\Phi(T')$ on the left hand side of the inequality which is as follows:

$$1 + \text{sum-of-ranks } a + \text{rank-}x' + (\text{sum-of-ranks } b + \text{rank-}y' + \text{sum-of-ranks } c)$$

Note that here x' is the position of node x after the rotation and similarly for y' . a, b, c are subtrees, and the 1 at the front is the actual cost accounting for the splay step. We want to show that this cost is bounded by the following:

$$1 + 3 * (\text{rank-}y - \text{rank-}x) + ((\text{sum-of-ranks } a + \text{rank-}x + \text{sum-of-ranks } b) + \text{rank-}y + \text{sum-of-ranks } c)$$

There are a lot of common terms between the two expressions, so it really boils down to showing the following

$$\text{rank-}x' + \text{rank-}y' \leq 3 * (\text{rank-}y - \text{rank-}x) + \text{rank-}x + \text{rank-}y$$

We can observe that since x' contains the same nodes as y' , $\text{rank-}x' = \text{rank-}y$. y' contains strictly less nodes than y , so $\text{rank-}y' \leq \text{rank-}y$. Using these two facts, we can easily show the equality above.

This follows a very similar structure to the pen-and-paper proof, but has a lot more arithmetic reasoning due to the mechanization.

Once we have shown this inequality, the proof is done for this case.

4.4.3 The Zig-Zag Case

As in the original proof, we case on whether or not the rank of x is different than the rank of x' . In the case where the rank of x does not change during the splay step, the original proof has a very short proof. Unfortunately, the same cannot be said for the mechanization.

The problem is after the splay step, the only claim we can make about y and z is that at least one of them has rank less than x' . Thus, we have to do another set of cases, based on which of y and z has rank less than x . This greatly lengthens the proof for this case of zig-zag and makes it by far the longest of all the cases.

It is also in these inductive cases that we must use the rank-rule, as it allows to make claims about the state of the final rotation.

4.4.4 The Zig-Zig Case

Even though zig-zig and zig-zag are not symmetric cases, there is a lot of similarity between the proofs. We start by again casing on the change between the ranks from the start position of the rotation to the final position. Fortunately, there is no need for nested casing, but the main change is that we need to consider the two rotations separately, as opposed to zig-zag where we only cared about the start and end of the splay step.

5 Future Work

As mentioned in the beginning of this paper, our goal was to prove that the amortized cost of the splay operation is $O(\log n)$ where n is the number of nodes in the tree. However, it is evident that `splay' / amortized` is not exactly that theorem. What we really want to prove is that over a sequence of m operations, the actual cost of splay is $O(m \log n + n \log n)$. This implies that the amortized cost over m operations is $O(m \log n)$ since the $n \log n$ comes from the cost bound for the potential function [10]. We can represent this theorem via the following homomorphism diagram which is an extension of Figure 4.1.1.

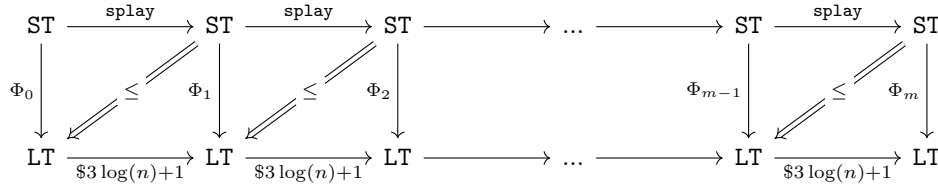


Figure 5.1

Note that Φ_i and $\Phi_j \forall i, j \in \{0, 1, \dots, m\}$ are equivalent. It is simply a notation that means that Φ_i represents calling the transition function after i calls to `splay`.

Furthermore, because of the composability that comes from working in a dependent type theory, much of the higher-level theorems from simpler examples like Batched Queues in Agda can be reused with some slight modification.

Thus, we feel that this result of `splay' / amortized` demonstrates a reasonably complete cost analysis for splay trees.

Using a homomorphism for this cost analysis yields great potential rewards. Defining the implementation of splay trees with good cost takes significant effort, but once the homomorphism is established, we can treat the splay trees as implemented via lists with good cost. Thus, any client wanting to reason with splay trees only needs to consider reasoning about lists while still reaping the benefits of the strong cost bound.

6 Acknowledgments

The author wishes to thank Harrison Grodin and Bob Harper for providing the opportunity for this work and especially Runming Li for providing continuous guidance and support throughout the process of producing this material. This material is based upon work supported by the United States Air Force Office of Scientific Research under grant number FA9550-21-0009 and FA9550-23-1-0434 (Tristan Nguyen, program manager). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR.

References

- [1] CUTLER, J.~W., LICATA, D.~R., AND DANNER, N. Denotational recurrence extraction for amortized analysis. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 1–29.
- [2] GRODIN, H., AND HARPER, R. Amortized analysis via coalgebra. *Electronic Notes in Theoretical Informatics and Computer Science Volume 4-Proceedings of...* (Dec. 2024).
- [3] GRODIN, H., NIU, Y., STERLING, J., AND HARPER, R. Decalf: A directed, effectful cost-aware logical framework. *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 273–301.
- [4] GROSEN, J., KAHN, D.~M., AND HOFFMANN, J. Automatic amortized resource analysis with regular recursive types, 2023.
- [5] LEVY, P.~B. Call-by-push-value. *ACM SIGLOG News* 9, 2 (May 2022), 7–29.
- [6] NIPKOW, T., AND BRINKOP, H. Amortized complexity verified. *J. Autom. Reason.* 62, 3 (Mar. 2019), 367–391.

- [7] NIU, Y., STERLING, J., GRODIN, H., AND HARPER, R. A cost-aware logical framework. *Proc. ACM Program. Lang.* 6, POPL (Jan. 2022).
- [8] SLEATOR, D.~D., AND TARJAN, R.~E. Self-adjusting binary search trees. *J. ACM* 32, 3 (July 1985), 652–686.
- [9] UNIVERSITY, C.~M. Amortized analysis. <https://www.cs.cmu.edu/~15451-f23/lectures/lecture05-amortized.pdf>, 2023.
- [10] UNIVERSITY, C.~M. Splay trees. <https://www.cs.cmu.edu/~15451-f23/lectures/lecture08-splay-trees.pdf>, 2023.