Splay Trees from a Functional Programming Perspective

bekauz

June 2019

Abstract

A self-adjusting version of binary search tree is observed from a functional programming perspective. This work mainly builds upon work done by Sleator and Tarjan on their original publishing about Splay Trees in 1987.

Contents

1 Introduction

In this paper the reader is presented with a Haskell implementation of the splay tree data structure and all of its main operations. The claim of $O(\log n)$ amortized time complexity for node access is verified using the potential method. Due to time restrictions, the initial plan of comparing the original splay operation with a slightly modified version of it was moved to further research section.

This work will begin with discussing a brief history of splay tree data structure, starting with its pre-decessing data structure - the standard binary search tree. Shortcomings of the latter are briefly discussed and then connected to the idea of splay trees. For the ease of the reader, information preliminary for further sections is given: reason behind using trees in the first place, basic tree rotations, their depth properties, rotation combinations. Then, the algorithm is presented and analyzed in terms of its Haskell implementation. Considering the original implementation of the algorithm was done in C, interesting differences between C and Haskell are pointed out. The algorithm is then analyzed using the potential method to verify the claim of $\log n$ access time.[1] Finally, we discuss potential modifications of the algorithm to make the node splaying process less aggressive and more customizable for specific cases.

1.1 Binary Search

The actual binary search was first mentioned in 1946 by John Mauchly, in an implementation which only accepted inputs of size $N = 2^n - 1$ [5]. Even with such constraints, the algorithm was still very significant for programmers: it is entirely possible that it was the first mention of non-numerical programming. In 1960, D.H. Lehmer published a modification of binary search which worked for all N [5]. In the same year, a tree-structured implementation of binary search was described by P.F. Windley, A.D. Booth, A.J.T. Colin, and T.N. Hibbard. It was a data structure capable of storing elements as nodes in memory and providing fast insertion, removal and lookup of its contents. Alongside the key used to uniquely identify the node, each node also held two sub-trees [2] commonly denoted as *left* and *right.* This way, the binary search property stating that the left sub-tree of a node must only contain nodes with keys lesser than that of the parent node, and that the right sub-tree of a node must only contain nodes with keys greater than that of the parent node, was satisfied.

Elements looked up in the tree are accessed via comparisons downwards from the root, which gives rise for four cases [4]:

1. The binary tree has no elements: element being looked up is not in the tree; search fails.
2. The lookup value matches the root value: search terminates successfully.
3. The lookup value is less than the root value: search continues by focusing on the left sub-tree of the root in recursive fashion.
4. The lookup value is more than the root value: search continues by focusing on the right sub-tree of the root in recursive fashion.

1.2 Knuth's Optimum Binary Search Tree Observation

Given that every element in the tree has a probability of $\frac{1}{n}$ to be accessed at any point in time where n is the number of elements, the best possible layout of the tree is going to be that of minimum path length to any node. This is more widely known as a complete binary search tree: a tree in which every level (with the possible exception of the last one) is completely filled and all of its nodes at depth $\log n$ are as far left as possible. That being said, in 1970 D.Knuth has observed

[1]We use binary logarithms throughout this paper.

[2]A sub-tree refers to a binary search tree with a root node and all of its descendants that is also a part of a bigger tree.

that "when some names are known to be much more likely to occur than others, the best possible binary tree will not necessarily be balanced". He described such variances of trees as the "optimal binary trees", when occurring frequencies were given [4]. Knuth's implementation was able to find the optimum binary search tree in order n^3 steps by using Cocke-Younger-Kasami context free grammar algorithm, which could be further refined to n^2 with some modifications if the context was known.

1.3 Optimum Binary Search Tree Shortcomings

This came with an obvious restriction, however: it was necessary to know the frequency of occurrence of each name in the tree, as well as the frequencies of occurrence of names not in the tree. This by no means dismisses Knuth's original idea: years later it turns out that the main takeaway from it was Knuth's observation about element access patterns and the fact that in real world, elements are not likely to have the same possibility of being accessed.

1.4 Sleator and Tarjan Splay Tree Background

The assumption of fixed probabilities of optimal search trees was one of the motivations for the development of splay trees by Sleator and Tarjan. Along that, shortcomings of various balanced binary search tree algorithms were noted as motivation as well, such as the fact that it was necessary to use extra space for storage of tree balancing information given that the access pattern was nonuniform. Authors observed that most balanced binary search trees were designed to reduce the worst-case time per operation. It was then noted that in typical real world applications of search trees, a sequence of operations was way more likely to be performed than a single operation. This meant that focusing on the time spent for an entire sequence of operations was the reasonable approach with real-world efficiency in mind. In 1985, R.E. Tarjan described such approach as the amortized analysis and formally defined it as a technique of analyzing the costs of a data structure by averaging out the worst-case operations over time and looking at a sequence of operations, rather than an individual operation [8].

1.5 Splay Tree Approach

The first way to achieve the amortized efficiency was to use a "self-adjusting" data structure: starting at an arbitrary state, a restructuring operation of constant time was to be applied. This way, the efficiency of future operations improved. Consequently, the space complexity of the algorithm was drastically reduced since no information related to re-balancing of the tree needed to be stored. Authors observed that if a sequence of node access operations was to be carried out on a binary search tree, minimizing the entire sequence access time was crucial. This could be achieved by placing the frequently accessed items near the root of the tree throughout the sequence of access pattern. The way this was achieved was by restructuring the binary search tree after every access operation by moving the item closer to the root, plausibly assuming that the item is probably going to be accessed sometime soon [6]. To rephrase, a high degree of locality of reference (tendency of a processor to access the same set of memory locations over a short period of time) was exploited and achieved with $O(1)$-time restructuring primitive called rotation, which preserves the symmetrical order of the tree.

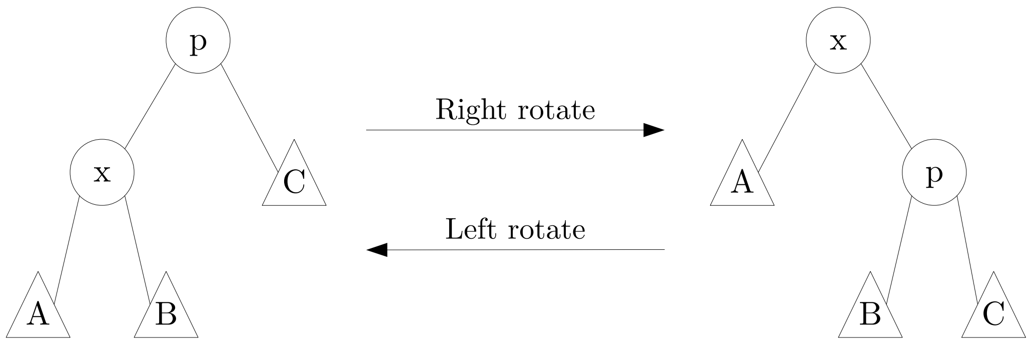
2 Preliminaries

Before getting into the actual analysis of the algorithm, it will be useful to go over some prerequisites that might help comprehend the following sections.

For simplicity, nodes are going to be identified by their labels, where x is the node in focus; p is the parent node of x, and gp is the parent node of p.

2.1 Single Rotations

Rotations in binary trees are used to restructure the tree while keeping the nodes in order, meaning that elements on the right subtree of a node are bigger than the node, and elements on the left subtree are smaller.



2.1.1 Right-Rotate

A right-rotation over the root node p of a tree (or subtree) in focus results in:

- the right child of p staying the right child of p,
- the left child of p becoming the parent of p and keeping its left subtree as its left subtree, and
- the right child of the left child of p becoming the left child of p.

This way, the properties of binary search tree are preserved:

- all right children of the new root node are bigger, and
- all left children of the new root node are smaller.

This, recursively, applies to all children (and parent) nodes of the tree.

2.1.2 Left-Rotate

Left rotations are just a mirror image (inverse) of right rotations. All properties are preserved in the same way.

2.2 Depth Properties of Rotations

A single rotation of a node makes either of its children go up 1 level and the other child to go down 1 level. With a bit of thought, if we perform a rotation like this, then switch to the parent of the new root of the tree and perform a rotation of a similar fashion - we can move any given node all the way up to the root, at which point no more rotations are possible and/or necessary.

2.3 Double Rotations

Double rotations over node x will perform two consecutive rotations in order to move x two levels up in the tree. Because of binary search tree mechanics, this requires to distinguish multiple combinations of rotations over node x or x's parent to achieve the depth reduction.

2.4 Tree Restructuring Heuristics and Splay Operation

The most elementary way of restructuring a tree is move-to-root: after accessing node x, the edge from x to its parent is rotated until x becomes the root of the tree. Although this sounds simple and making sense, it is still possible that access time will remain $O(n)$ because it works only with respect to the node we are moving itself. Even though x will become the root, it can result in other nodes drifting further away from root which is not an ideal solution. On the other hand, such operation requires very little knowledge of the topology, hence the lack of maintenance of it.

Sleator and Tarjan realized the potential of move-to-root and addressed its flaws by building on top of it. Their way of restructuring which was named splaying is similar to move-to-root in a way that the rotations it performs top-to-bottom along the access path and moves the accessed item all the way to the root [6]. The key part to it, and what makes it different, is that the rotations are performed in pairs of nodes, where the structure of the access path is the deciding factor of the order of rotations performed. This will be further explained in section §3.3.2.

Considering that splay operation performed on a node x of depth d happens on the same path node was located in the first place, it is easy to see that it takes $\Theta(d)$ time, which is proportional to the time to access node x in the first place. This will be further analyzed in section §4, where such proportional balance will be expressed in terms of the potential method [6].

3 Implementation

The original implementation of Splay Trees by Sleator and Tarjan was done in C. This allowed to make clever use of pointers which resulted in a very elegant and straightforward implementation of the splay operation [6]:

```
procedure splay(x);
  {p(null) = left(null) = right(null) = null}
  do x = left(p(x)) ->
    if g(x) = null -> rotate right (p(x))
     | p(x) = left(g(x)) -> rotate right(g(x)); rotate right(p(x))
     | p(x) = right(g(x)) -> rotate right(p(x)); rotate left(p(x))
    fi
   | x = right(p(x)) ->
    if g(x) = null -> rotate left(p(x))
     | p(x) = right(g(x)) -> rotate left(g(x)); rotate left(p(x))
     | p(x) = left(g(x)) -> rotate left(p(x)); rotate right(p(x))
    fi
  od {p(x) = null}
end splay;
```

The grandparent function g is defined as follows [6]:

```
function g(x);
  g:= p(p(x))
end g;
```

Functional programming implementation of such algorithm is quite different because of the lack of pointers.

3.1 Binary Tree Data Structure

The main data structure that the algorithm builds upon is a holder for the tree nodes. It can take on two forms. First - a leaf, which indicates that the binary tree instance holds no value and no children. This can mean that the entire tree is empty, or that it is an actual leaf node with a valid parent node. Other form it can take is an actual tree, holding two binary trees represented as children nodes and a value holder which represents the actual value held by the node. This value is how we will identify nodes further on.

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
    deriving (Show)
```

3.2 Path Memory (Zipper)

Considering Haskell can not access values based on pointers, an alternative way to recreate the path from root to node in focus was needed. Zipper turned out to be a perfect data structure for that. First described in 1997 by Gérard Huet, it is perfect for "representing a tree together with a subtree that is the focus of attention, where that focus may move left, right, up or down the tree" [3]. It can be further adapted to use with lists and various recursively defined data structures. In this case, it will indicate whether the node in focus is a left child (L) or a right child (R); what his parent node is; and the other child of the parent node. While the first two are rather self-explanatory, maintaining the other child of the parent node turned out to be crucial for rotations in order to maintain the order of nodes of the binary search tree.

```
data Path a = L a (Tree a) | R a (Tree a)
    deriving (Show)
```

An array of `Path` elements is built up step by step with each recursive call in a similar fashion as breadcrumbs dropped to mark the the ancestor nodes at each level. This way we are able to recreate the path bottom-up from node in focus all the way up to the root, which is crucial for our splay operations. The name, Zipper, comes from the fact that "going up and down in the structure is analogous to closing and opening a zipper in a piece of clothing" [3].

3.3 Splay

```
splay :: Tree a -> [Path a] -> Tree a
```

Once any regular binary search tree operation like delete, insert, or find is called, a splay call will happen as well, which requires a path memory to work. For this reason, a splay operation on a tree with no path will just mean that no nodes have been visited yet:

```
splay tree [] = tree
```

This leaves us with cases where path holds at least a single element. Based on the layout resembled by a node in focus and its ancestor nodes, such cases are often called Zig, Zig-Zig and Zig-Zag. For example, if a node is a left child of a left child - it is not hard to see the monotonic zig-zig route from it to its grandparent node. Depth of a node allows us to further distinguish the splay cases:

3.3.1 Splay of Depth = 1

If node in focus is of depth $d = 1$, it means that a single rotation over root is going to be needed. The type of rotation we are going to perform will depend on whether the node in focus is a left child or a right child: given that the node is a left child, we return a tree where the node in focus is the root; the left child of the node in focus remains the left child; the right child of the node in focus is going to be a new subtree which we structure based on the usual binary search tree rules.

```
splay (Node left x right) [L p p_rightChild] =
    Node left x (Node right p p_rightChild)
```

If the node is a right child, a rotation of similar fashion is going to be performed, where every operation is a mirror image of its left counterpart: the node in focus becomes the root; right child of that node in focus remains the right child; the left subtree is structured based on the usual binary search tree rules.

```
splay (Node left x right) [R p p_leftChild] =
    Node (Node p_leftChild p left) x right
```

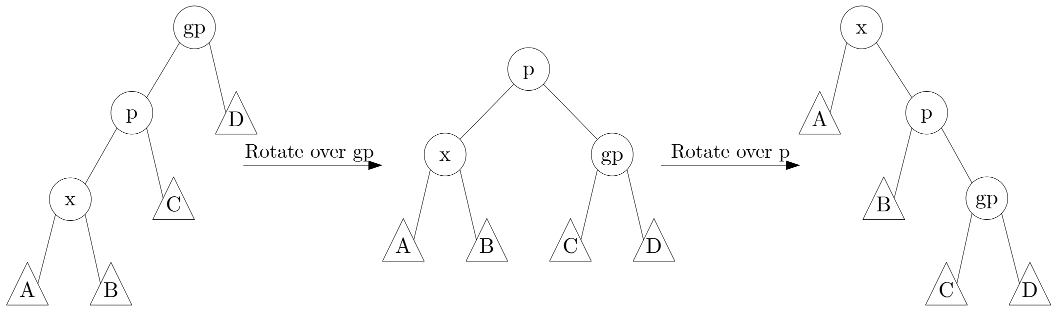
After either of the operations from depth $d = 1$, splaying operation terminates as the node in focus has been successfully rotated all the way to the top.

3.3.2 Splay of Depth ≥ 2

Splaying a node of depth $d \geq 2$ requires to take a few things into account: the first two elements of `[Path]`, which represent the parent and the grandparent of our node in focus. With this information available, the rotation, unlike in C, becomes a "one-liner" thanks to the functional implementation of Haskell. After every step like this, the first two elements of the `[Path]` are dropped, because we have already went back the two steps and we will need to consider other nodes in the future. Unlike splaying a node of depth $d = 1$, splaying of depth $d = 2$ requires to take two nodes above it into consideration (parent and grandparent). This is required to determine the right combination of rotations and distinguishes four different cases:

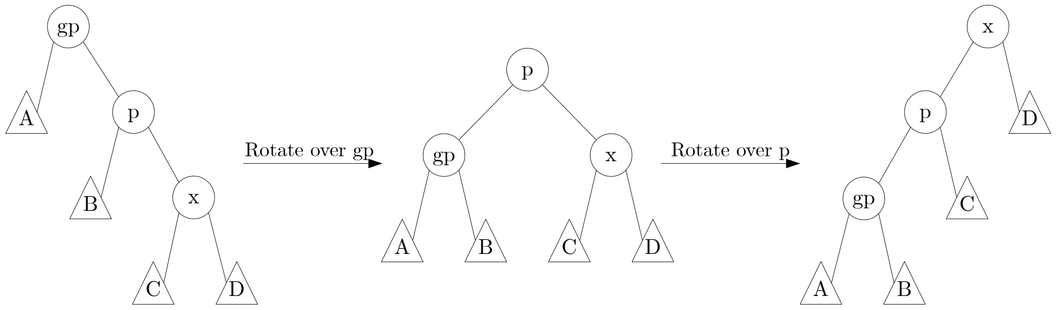
Case 1: Node in focus is a left child of a left child [Zig-Zig]

```
splay (Node left x right) (L p p_rightChild : L gp gp_rightChild : path) =
splay (Node left x (Node right p (Node p_rightChild gp gp_rightChild))) path
```



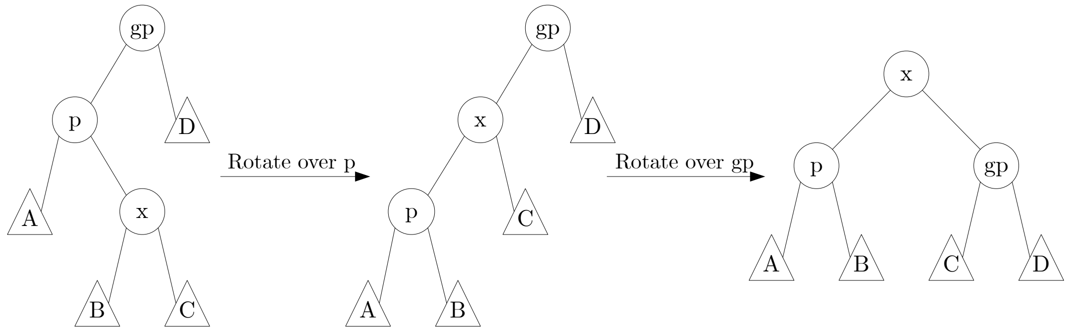
Case 2: Node in focus is a right child of a right child [Zig-Zig]

```
splay (Node left x right) (R p p_leftChild : R gp gp_leftChild : path) =
splay (Node (Node (Node gp_leftChild gp p_leftChild) p left) x right) path
```



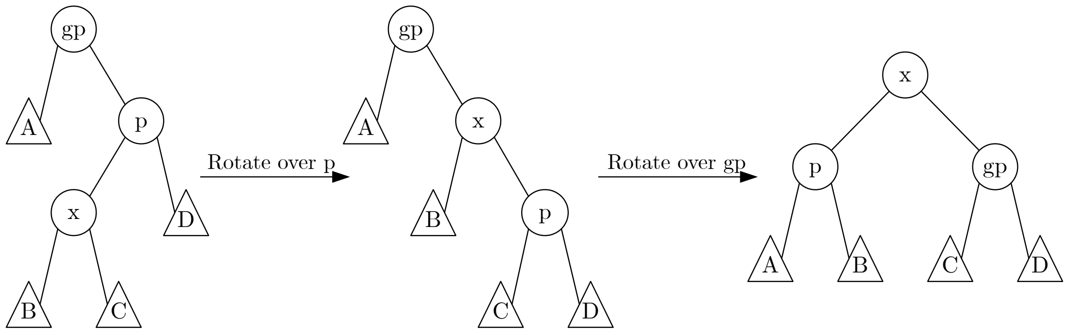
Case 3: Node in focus is a right child of a left child [Zig-Zag]

```
splay (Node left x right) (R p p_leftChild : L gp gp_rightchild : path) =
splay (Node (Node p_leftChild p left) x (Node right gp gp_rightchild)) path
```



Case 4: Node in focus is a left child of a right child [Zig-Zag]

```
splay (Node left x right) (L p p_rightChild : R gp gp_leftChild : path) =
splay (Node (Node gp_leftChild gp left) x (Node right p p_rightChild)) path
```



3.4 Insertion

Splay tree insertion is basically an extension of the regular binary search tree insert operation: starting at the root node, the tree is traversed until appropriate leaf is found. A node is then initialized to have the value we are inserting, and extended with two empty leaves of its own. This is where the normal binary search tree insertion would terminate successfully; in the case of splay trees, the node is then splayed all the way up to the root. This is explained by the assumption that the node is likely to be accessed again sometime soon. In practice this would mean populating the newly initiated node with $O(1)$ access, for instance a database entry of a newly created user which is about to fill in his details right after creating his account.

```
-- Insert value 'a' into tree and return updated tree
insert :: (Ord a) => a -> Tree a -> Tree a
insert a tree = extendPath a [] tree

-- Insert element and extend path
extendPath :: (Ord a) => a -> [Path a] -> Tree a -> Tree a
-- If tree is empty or we reached a leaf node
extendPath a path Leaf = splay (Node Leaf a Leaf) path
-- If tree isn't empty: start traversing downwards
extendPath a path (Node left x right)
    | x < a     = extendPath a ((R x left) : path)  right
    | x > a     = extendPath a ((L x right) : path) left
    | otherwise = error "Value already exists in the tree."
```

In this implementation case, the `insert` method is there as a helper – to simplify the input. With it, we can now call `'insert 5 treeName'`, which will in turn call the `extendPath` method with an empty `[Path]` to indicate a new call. A design decision was taken that the tree will not contain double values, thus an error is returned in case the value being inserted already exists in the tree.

3.5 Find

Find operation will traverse the tree starting at the root until one of two cases occur:

1. A node is found successfully, at which point we splay the node all the way to the top.
2. Node is not found, at which point we splay the last node visited in the search.

Reasoning behind splaying the last visited node in case of failed search is similar to that of insertion: even if we failed to find the node we were looking for in the first place, there is a good chance that we will be looking for nodes with similar values in the near future. This way, such nodes will be located closer to the root than before and result in faster further operations.

```
find :: (Ord a) => a -> Tree a -> Tree a
find _ Leaf = Leaf
-- Call helper function with empty path
find a tree = findPath a [] tree

findPath :: (Ord a) => a -> [Path a] -> Tree a -> Tree a
-- If x has no children, splay x regardless
findPath a path (Node Leaf x Leaf) = splay (Node Leaf x Leaf) path
-- If x has no bigger children
findPath a path (Node left x Leaf)
    | a > x     = splay (Node left x Leaf) path
    | a < x     = findPath a ((L x Leaf) : path) left
    | otherwise = splay (Node left x Leaf) path
-- If x has no smaller children
findPath a path (Node Leaf x right)
    | a < x     = splay (Node Leaf x right) path
    | a > x     = findPath a ((R x Leaf) : path) right
    | otherwise = splay (Node Leaf x right) path
-- If x has both children
findPath a path (Node left x right)
    | a < x     = findPath a ((L x right) : path) left
    | a > x     = findPath a ((R x left) : path) right
    | otherwise = splay (Node left x right) path
```

Once again, a helper method is used in order to be able to simply call `find value treeName`, which then in turn calls the "actual" method with an empty path which will perform the operations described above.

3.6 Delete

Removing a node starts by traversing the tree downwards, starting at the root as with ordinary binary search tree. In case the node has been found and successfully removed, we splay the parent of the removed node. If a node failed to be found, we will splay the node where the search for it ended. This creates some different cases in the implementation which starts with a helper method which takes just the value we wish to remove and the tree from which we wish to remove it from:

```
deleteNode :: (Ord a) => a -> Tree a -> Tree a
deleteNode _ Leaf = Leaf
deleteNode a (Node left x right) = deleteWithPath a [] (Node left x right)
```

First case is if we are trying to delete something from and empty tree, which simply returns an empty tree. If the tree is not empty, a further method is called with an empty path to keep track of traversed nodes:

```
deleteWithPath :: (Ord a) => a -> [Path a] -> Tree a -> Tree a
```

Four distinct cases arise from it:

Case 1: If node we are trying to remove has no children, we simply splay its parent with a leaf instead of the value we are trying to remove

```
deleteWithPath a path (Node Leaf x Leaf)
    | a == x    = splayParent path Leaf
    | otherwise = splay (Node Leaf x Leaf) path
```

Case 2: If node we are trying to remove has only smaller children, we will set the node we are deleting to be its left child and splay it

```
deleteWithPath a path (Node left x Leaf)
    | a < x     = deleteWithPath a (L x Leaf : path) left
    | a == x    = splayParent path left
    | otherwise = splay (Node left x Leaf) path
```

Case 3: If node we are trying to remove has only bigger children, we will set the node we are deleting to be its right child and splay it

```
deleteWithPath a path (Node Leaf x right)
    | a > x     = deleteWithPath a (R x Leaf : path) right
    | a == x    = splayParent path right
    | otherwise = splay (Node Leaf x right) path
```

Case 4: If node we are trying to remove has both smaller and bigger children, we will search its left subtree for the biggest leaf x, replace the node with that we are deleting with x, delete the original x, and splay the replaced node

```
deleteWithPath a path (Node left x right)
    | a < x     = deleteWithPath a (L x right : path) left
    | a > x     = deleteWithPath a (R x left : path) right
    | otherwise = splayParent path (Node (deleteMax left) (findMax left) right)
```

4 Amortized Analysis

Although in the worst-case scenario splay trees can take up to $\Theta(n)$ access time, it is highly unlikely. Such scenarios can occur for a single operation from time to time, but the true potential of the algorithm is seen when looking at a broader picture, which in this case is a sequence of n operations.

Potential method was chosen to be used for the amortized cost analysis. With it, we pay for the work required to perform the operations in advance. The prepaid work is represented as potential which can be released to pay for future operations. The potential is associated with the entire binary search tree T in focus. We then perform a sequence of n operations on the initial binary search tree T_0 and end up with a new binary search tree T'. For each of the $i = 1, 2...n$ operations, we let c_i denote the actual cost of the i'th operation, and T_i denote the binary search tree after applying that operation. The potential function ϕ applied to the binary search tree at any point will return the potential of the given tree. With this, we define the amortized cost $\widehat{c_i}$ of the i'th operation as the sum of the actual cost of the i'th operation and the change in potential that operation results in [1].

4.1 Notation

Every node in the tree is assigned a positive weight w_i such that

$$w : node \to \mathbb{R}^+ \tag{1}$$

Even though weighs for the proof can be arbitrary (as long as they are positive), for the following proofs it will suffice to assume that $w_i = 1$.
Let size $s(x)$ of a node x be defined as the sum of weights of all the nodes within the tree rooted at x as follows:

$$s(x) = \sum_{i \in subtree(x)} w_i \tag{2}$$

Having the size allows us to define rank $r(x)$ of a node x as:

$$r(x) = \log s(x) \tag{3}$$

We further define the potential function ϕ of a tree as the sum of ranks of all its nodes including the root. The potential of a well-balanced tree will tend to be low, and that of a poorly-balanced tree will tend to be high. By observing the potential of a tree, we will be able to see how performing some operation on a node affects the rank of a tree $r(T)$.

$$\phi = \sum_{i \in subtree(x)} r(i) \tag{4}$$

Finally, let s and s' denote the size of a tree before and after the splay operation, respectively, and let r and r' denote the rank of a tree before and after the splay operation, respectively. This will allow us to assign an upper bound for the change of potential of a given node.

4.2 Objectives

We will try to prove that any node access operation of splay trees takes $O(\log n)$ amortized time. Before that, we need to consider some properties of the rotations. After either of the double-rotations, the depth of the node being splayed reduces by 2, while single-rotation reduces it by 1. Thus, splaying a node x of depth d will consist of some sequence of $\lfloor \frac{d}{2} \rfloor$ double-rotations and an additional single-rotation in case d is odd. Considering that any of the rotations will only affect the ranks of a constant number of nodes, we are able to claim that [2]:

Splaying any node x in a binary search tree T will take $O(d)$ time where d is the depth of x in T.

With this in mind, we deduct that proving the splay operation complexity alone will be sufficient to prove the complexity of all splay tree operations.

In order to apply the potential method, we first have to calculate the change in potential $\Delta\phi$ caused by a single splaying step. We perform this with a proof of every distinct splaying step:

4.2.1 Single Rotation: Zig

In the simplest case of rotations only x and p will change ranks. We attempt to bound the change in potential:

$$\begin{aligned}\Delta\phi &= r'(x) + r'(p) - r(x) - r(p)\\ &\le r'(x) - r(x)\\ &\le 3(r'(x) - r(x))\end{aligned}$$

We are able to drop $r(p)$ and $r'(p)$ from the second line because $r'(p) \le r(p)$ and we are trying to achieve an upper bound.

4.2.2 Double Rotation: Zig-Zig

Considering the only ranks that will change during a Zig-Zig rotation are those of x itself, its parent and grandparent, we try to bound the change in potential:

$$\begin{aligned}\Delta\phi &= r'(x) + r'(p) + r'(gp) - r(x) - r(p) - r(gp)\\ &= r'(p) + r'(gp) - r(x) - r(p)\\ &\le r'(x) + r'(gp) - 2r(x)\end{aligned}$$

Now using the property that

$$\text{if } a, b > 0, \text{ and } c > a + b, \text{ then } \log a + \log b < 2\log c - 2 \tag{5}$$

We deduct:

$$\begin{aligned}r(x) + r'(gp) &< 2r'(x) - 2\\ r'(gp) &< 2r'(x) - r(x) - 2\end{aligned}$$

Which allows us to further bound $\Delta\phi$:

$$\begin{aligned}\Delta\phi &\le r'(x) + r'(gp) - 2r(x)\\ &\le r'(x) + 2r'(x) - r(x) - 2 - 2r(x)\\ &\le 3r'(x) - 3r(x) - 2\\ &\le 3(r'(x) - r(x)) - 2\end{aligned}$$

4.2.3 Double Rotation: Zig-Zag

As in previous double-rotation, x, x's parent and grandparent will change ranks. While keeping in mind that $r(x) < r(p) < r(gp) = r'(x)$, we attempt to bound the change in potential:

$$\begin{aligned}\Delta\phi &= r'(x) + r'(p) + r'(gp) - r(x) - r(p) - r(gp)\\ &= r'(p) + r'(gp) - r(x) - r(p)\\ &\leq r'(p) + r'(gp) - 2r(x)\end{aligned}$$

Using (5) we further bound $\Delta\phi$ as $r'(p) + r'(gp) < 2r'(x) - 2$:

$$\begin{aligned}\Delta\phi &\leq 2r'(x) - 2 - 2r(x)\\ &\leq 2(r'(x) - r(x)) - 2\\ &\leq 3(r'(x) - r(x)) - 2\end{aligned}$$

4.2.4 Change of Potential Summary

From the previous observations we conclude that the change of potential of the tree is affected by:

$$\Delta\phi \leq \begin{cases} 3(r'(x) - r(x)) & \text{if the rotation was single,}\\ 3(r'(x) - r(x)) - 2 & \text{if the rotation was double.}\end{cases}$$

4.2.5 Amortized Cost of Splay Operation

The amortized cost of any splay operation step is equal to the sum of change in potential $\Delta\phi$ and the actual cost c of rotations performed. Given that the actual cost c of any rotation is constant, the amortized cost $\widehat{c}$ of a single splaying step is:

$$\widehat{c} \leq 3(r'(x) - r(x)) + 1$$

Summing the amortized costs of all splaying steps involved in the entire splaying operation results in:

$$\begin{aligned}\text{amortized cost} &= \sum_i \widehat{c_i}\\ &\leq \sum_i 3(r^{i+1}(x) - r^i(x)) + 1\\ &= 3(r'(x) - r(x)) + 1\\ &= 3(r(T) - r(x)) + 1\end{aligned}$$

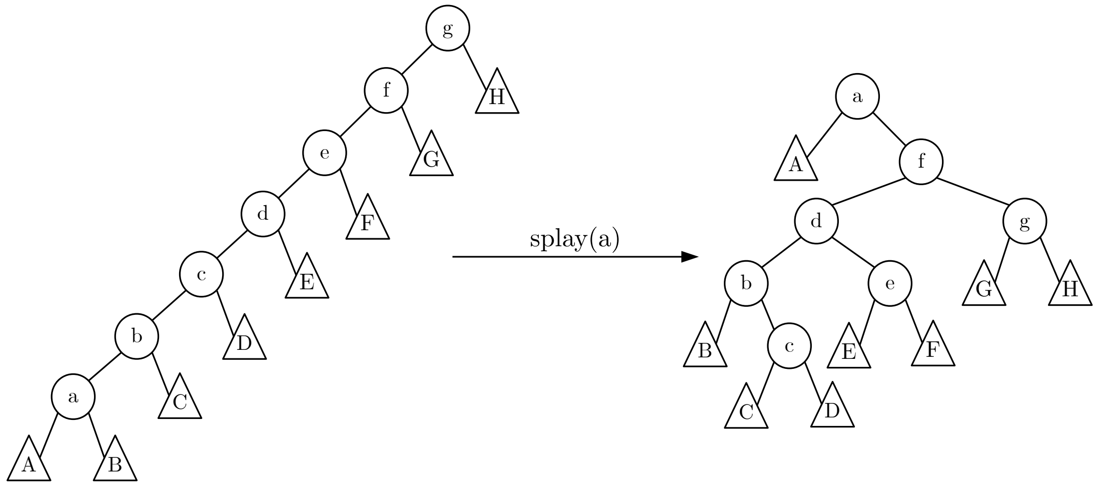
The addition of 1 comes in case the depth d of the node being splayed was odd and required the additional Zig rotation [6]. Considering our previous assumption that the weights $w(x) = 1$ for all nodes in the tree, the rank of the tree $r(T) = \log n$, we have:

$$\text{amortized cost} \leq 3\log n + 1$$

From which we conclude that the amortized cost of splaying a node x is $O(\log n)$.

4.3 Amortized Cost Summary

It is important to mention that a node that lies at initial depth d on the access path from the root downwards to any node x being accessed will move to a final depth $d \leq 3 + \frac{d}{2}$ [6]. This approximately translates to halving the depth of nodes that were far away from the root before the rotations. Looking at the following situation might help to visualize the way splaying performs in $O(\log n)$:



This way, even if we were to reach a situation where the depth d of the tree is equal to the number of nodes n, a single access operation is capable of re-balancing the tree to the point where such a situation is unlikely to happen again anytime soon. With this in mind, it is easier to see that any $O(n)$ access operation is going to be compensated for in terms of its cost by other operations of $O(1)$ or $O(\log n)$ access time over a sequence of operations.

5 Further Research

Upon learning about splay tree mechanics both me and my supervisors were surprised to learn that the splay operation was done so aggressively and all the way up to the root. Considering the space complexity of splay trees is essentially equal to that of a regular binary search tree, I believe that adding a little overhead for the sake of a more predictable performance would not be unreasonable. Certain observations about possible variations of the algorithm were made by Sleator and Tarjan themselves. They have noted that one of the disadvantages that splay trees posses is the huge amount of tree restructuring it performs. One of their suggestions was to only rotate a fraction of the edges along the access path thus moving the node only partway toward the root [6]. Building upon this idea, maybe it would be possible to perform splaying in levels. For this, a simple counter variable could be attached to each node. Given the small space complexity of the algorithm prior to this addition, it should be reasonable to assume little performance loss due to this. With this counter set to 1 at start, every time a node would be splayed, its counter variable would be doubled. Inspiration for this is essentially drawn from the common practice of doubling the array sizes in languages where their extension is not possible without copying each element of the array to a new one. With these additions, every time a node would be splayed, its height would decrease by x where x is the counter variable. This way, if a node is going to be accessed often, its counter will grow to at least $\log n$ (where n = number of nodes in tree) exponentially fast anyways. What this would also give is the security from an outlier node which would be only accessed once from restructuring the tree too drastically. It seems reasonable to assume that such modifications to the algorithm would allow to predict the run-time of the algorithm more easily. The amortized cost would remain the same, as the only thing changes is the number of levels splayed. At the same time, if node was going to be accessed often, it would still be splayed to the top for $O(1)$ access.

6 Conclusion

To finish up, it was interesting to notice how things as staple as a binary search tree continue to evolve and improve, even though there is a distinct definition of its limitations and rules. While splay trees were defined in 1987, various implementations of them keep on emerging even today. In 2017, one of the most used text editors - Atom, has announced that they switched from using a simple list to splay trees for its buffer implementation [7]. As text editor input is highly variable, splay trees are able to be used to their full potential in various auto-completion and other string-based applications. On the other hand, they do come with their share of disadvantages. One of such is the way splay trees behave in a read-dominated operation environment. This does not only fail to take advantage of the splay operation after insertion, but also complicates matters in a multi-threaded environment. Due to the extra management necessary in order to synchronize parallel threads, their applications in functional programming also become limited because of how well it scales with additional threads due to no shared state necessary.

References

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[2] Michael T Goodrich, Roberto Tamassia, and Michael H Goldwasser. *Data structures and algorithms in Java*. John Wiley & Sons, 2014.

[3] Gérard Huet. The zipper. *Journal of functional programming*, 7(5):549–554, 1997.

[4] Donald E Knuth. Optimum binary search trees. *Acta informatica*, 1(1):14–25, 1971.

[5] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., 1998.

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

[7] Nathan Sobo. Atom's new concurrency-friendly buffer implementation. *Atom's Developer Notes*, 2017.

[8] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

A Haskell Implementation

```
import Data.List hiding (insert,find)

-- Holds either nothing (leaf) or a node with its left and right subtrees
data Tree a = Leaf | Node (Tree a) a (Tree a)
    deriving (Show)

-- Holds data about the p of node in focus (direction taken & p itself)
data Path a = L a (Tree a) | R a (Tree a)
    deriving (Show)

-- Initialize empty tree
singleton :: a -> Tree a
singleton x = Node (Leaf) x (Leaf)

-- Initialize splay tree
splay :: Tree a -> [Path a] -> Tree a
splay tree [] = tree -- tree with no path to it is a new tree

{------------ Depth = 1 (single rotation over root) cases: ----------------}

-- Node splayed is a LEFT child: ZIG
splay (Node left x right) [L p p_rightChild] =
    Node left x (Node right p p_rightChild)

-- Node splayed is a RIGHT child: ZAG
splay (Node left x right) [R p p_leftChild] =
    Node (Node p_leftChild p left) x right

{---------  Depth >= 2 (double rotation over p/gp) ZIGZIG cases: ----------}

{-- Node splayed is a LEFT child of a LEFT child: --}
splay (Node left x right) (L p p_rightChild : L gp gp_rightChild : path) =
splay (Node left x (Node right p (Node p_rightChild gp gp_rightChild))) path

{-- Node splayed is a RIGHT child of a RIGHT child: --}
splay (Node left x right) (R p p_leftChild : R gp gp_leftChild : path) =
splay (Node (Node (Node gp_leftChild gp p_leftChild) p left) x right) path

{---- Depth >= 2 (double rotation over p/gp) ZIGZAG cases: --------}

{-- Node splayed is a RIGHT child of a LEFT child: --}
splay (Node left x right) (R p p_leftChild : L gp gp_rightchild : path) =
splay (Node (Node p_leftChild p left) x (Node right gp gp_rightchild)) path

{-- Node splayed is a LEFT child of a RIGHT child: --}
splay (Node left x right) (L p p_rightChild : R gp gp_leftChild : path) =
splay (Node (Node gp_leftChild gp left) x (Node right p p_rightChild)) path

{------------------------  Insertion into tree --------------------------}

-- Insert value 'a' into tree and return updated tree
insert :: (Ord a) => a -> Tree a -> Tree a
insert a tree = extendPath a [] tree

-- Insert element and extend path
```

```
extendPath :: (Ord a) => a -> [Path a] -> Tree a -> Tree a
-- If tree is empty or we reached a leaf node
extendPath a path Leaf = splay (Node Leaf a Leaf) path
-- If tree isn't empty: start traversing downwards
extendPath a path (Node left x right)
    | x < a     = extendPath a ((R x left) : path)  right
    | x > a     = extendPath a ((L x right) : path) left
    | otherwise = error "Value already exists in the tree."

{-------------------------- Splay (find) element --------------------------}

find :: (Ord a) => a -> Tree a -> Tree a
find _ Leaf = Leaf
-- Call helper function with empty path
find a tree = findPath a [] tree

findPath :: (Ord a) => a -> [Path a] -> Tree a -> Tree a
-- If x has no children, splay x regardless
findPath a path (Node Leaf x Leaf) = splay (Node Leaf x Leaf) path
-- If x has no bigger children
findPath a path (Node left x Leaf)
    | a > x     = splay (Node left x Leaf) path
    | a < x     = findPath a ((L x Leaf) : path) left
    | otherwise = splay (Node left x Leaf) path
-- If x has no smaller children
findPath a path (Node Leaf x right)
    | a < x     = splay (Node Leaf x right) path
    | a > x     = findPath a ((R x Leaf) : path) right
    | otherwise = splay (Node Leaf x right) path
-- If x has both children
findPath a path (Node left x right)
    | a < x     = findPath a ((L x right) : path) left
    | a > x     = findPath a ((R x left) : path) right
    | otherwise = splay (Node left x right) path

{--------------------------- Deletion from tree ---------------------------}

-- helper function
deleteNode :: (Ord a) => a -> Tree a -> Tree a
deleteNode _ Leaf = Leaf
deleteNode a (Node left x right) = deleteWithPath a [] (Node left x right)

deleteWithPath :: (Ord a) => a -> [Path a] -> Tree a -> Tree a
-- node in focus has no children
deleteWithPath a path (Node Leaf x Leaf)
    | a == x    = splayParent path Leaf
    | otherwise = splay (Node Leaf x Leaf) path
-- node in focus has only smaller children
deleteWithPath a path (Node left x Leaf)
    | a < x     = deleteWithPath a (L x Leaf : path) left
    | a == x    = splayParent path left
    | otherwise = splay (Node left x Leaf) path
-- node in focus has only bigger children
deleteWithPath a path (Node Leaf x right)
    | a > x     = deleteWithPath a (R x Leaf : path) right
    | a == x    = splayParent path right
    | otherwise = splay (Node Leaf x right) path
-- node in focus has both smaller and bigger children
deleteWithPath a path (Node left x right)
    | a < x     = deleteWithPath a (L x right : path) left
    | a > x     = deleteWithPath a (R x left : path) right
    | otherwise =
        splayParent path (Node (deleteMax left) (findMax left) right)

splayParent :: (Ord a) => [Path a] -> Tree a -> Tree a
splayParent ((L parent p_oC) : path) child =
    splay (Node child parent p_oC) path
splayParent ((R parent p_oC) : path) child =
    splay (Node child parent p_oC) path

-- returns max value from a tree
findMax :: (Ord a) => Tree a -> a
findMax (Node left x Leaf)  = x
findMax (Node _ x right)    = findMax right

-- deletes the biggest element from a tree
deleteMax :: (Ord a) => Tree a -> Tree a
deleteMax (Node left x Leaf)  = left
deleteMax (Node left x right) = Node left x (deleteMax right)
```