Heuristic Analysis of The Splay Tree

Brad Michael Lindsay

Mathematics and Computer Science Department

University of Lethbridge

Table of Contents

[Abstract 3](#_Toc479097708)

[Introduction 4](#_Toc479097709)

[Operations 4](#_Toc479097710)

[Splay 4](#_Toc479097711)

[Rotation 5](#_Toc479097712)

[Experiments 5](#_Toc479097713)

[Linear Insertion 5](#_Toc479097714)

[Tabular Data 5](#_Toc479097715)

[Graph 6](#_Toc479097716)

[Odd Even Insertion 6](#_Toc479097717)

[Tabular Data 7](#_Toc479097718)

[Graph 7](#_Toc479097719)

[Linear Delete 7](#_Toc479097720)

[Tabular Data 8](#_Toc479097721)

[Graph 8](#_Toc479097722)

[Non-Linear Delete 9](#_Toc479097723)

[Tabular Data 9](#_Toc479097724)

[Graph 9](#_Toc479097725)

# Abstract

This report describes the Splay Tree and all its operations. We will attempt to establish the correctness of the operations, and explore the worst case amortized running time. A series of experiments on this data structure will be explained, and the conclusions from those experiments explored. We will then compare theoretical bounds on running time against the experimental bounds.

# Introduction

A Splay Tree is an expansion on the Binary Search Tree (BST). This tree maintains a set of properties which allow it to self-balance. Every Splay Tree is capable of the following BST operations: Search, Insert, and Delete. Splay Trees also have added operations outside of the standard BST interface. Once facet of the Splay Tree is that for every operation called on the tree, a balancing operation called a “splay” is performed. A splay operation effectively moves a specific node to the root of the tree through a series of transformations moves it to the top of the tree. These transformations are commonly referred to as rotations and can be executed in sequence to achieve single or double rotations around a given element. These rotations are like rotations seen in the Adelson-Velsky Landis (AVL) Tree.

# Operations

## Splay

The Splay Tree maintains all the operations provided by the BST interface with the addition of the splay operation. The splay operation is not available publicly and is performed during every BST action. This operation is the backbone of the Splay Tree. For arguments, the splay operation accepts both a node and a key. Since the splay operation is recursive in nature the initial call to splay is usually run from the root node of the tree. The key we provide as the second argument is the data to find and the node containing that data or key will be brought to the top of the tree through a manner of several recursive calls to splay. When we finally reach a node that matches our input data we return it to our calling function. The splay function uses a series of different rotations to bring the node we find to the surface before returning it.

## Rotation

The Splay Tree uses two different rotations on nodes. The rotations are very similar in implementation, so they will both be included in a single section for brevity. The rotation can be performed in two directions: Left, and Right. A left rotation effectively sets the right subtree of the left child and attaches it to the parent node. The right subtree of the child then becomes the left parent node and all its remaining subtrees. The reverse happens with the right rotation. When combined these rotations help perform an action called a double rotation, which is effective in smoothing out non-linear structures for balancing.

# Experiments

## Linear Insertion

The first test that was run against the Splay Tree was a Linear Insertion Test. This test started with 1,000,000 insertions and every run doubled the amount of insertions. This made the result easy to graph for visualization.

### Tabular Data

|  |  |  |
| --- | --- | --- |
| **RUN** | **INSERTIONS** | **RUNTIME** |
| **1** | 1000000 | 8.74E-02 |
| 2 | 2000000 | 1.75E-01 |
| 3 | 4000000 | 3.69E-01 |
| 4 | 8000000 | 6.91E-01 |
| 5 | 16000000 | 1.36E+00 |
| 6 | 32000000 | 2.71E+00 |
| 7 | 64000000 | 5.87E+00 |
| 8 | 128000000 | 1.23E+01 |
| 9 | 256000000 | 2.45E+01 |
| 10 | 512000000 | 5.05E+01 |

### Graph

As we can see here, the growth rate of a linear insertion is almost identical to that of a growth rate of amortized O(logn); where in a sequence of M operations on an N-node splay tree takes O(MlogN) time.

## Odd Even Insertion

In this test, the data values were inserted in an odd-even fashion. The first test was with 1000 keys, having all the odd keys inserted first. Once all the odd keys were inserted, the test timer was started, and even keys were inserted. This caused the tree to enter worse case splaying. The number of keys was then doubled and the tests run again. This was done for a total of ten different test runs.

### Tabular Data

|  |  |  |
| --- | --- | --- |
| **RUN** | **RUNTIME (Seconds)** | **INSERTIONS** |
| **1** | 0.000089 | 1000 |
| 2 | 0.000194 | 2000 |
| 3 | 0.000387 | 4000 |
| 4 | 0.000845 | 8000 |
| 5 | 0.001583 | 16000 |
| 6 | 0.004037 | 32000 |
| 7 | 0.006934 | 64000 |
| 8 | 0.014122 | 128000 |
| 9 | 0.029918 | 256000 |
| 10 | 0.057529 | 512000 |

### Graph

As with the linear insertion, it seems that the odd-even insertion obeys the theoretical bounds of the worst-case time complexity for splay insertion. The few data collected do seem to trend towards amortized O(logn) time complexity.

## Linear Delete

This test looks at the delete efficiency of the Splay Tree. As with the Linear insertion test, the values of the tree are inserted in and doubled every run, starting with 1,000,000 records, for a round of 10 runs. After the values are inserted, a timer is started and the delete operation occurs starting with the last element inserted to the first element inserted. This ensures that the tree structure maintains its linear shape during the trial.

### Tabular Data

|  |  |  |
| --- | --- | --- |
| **RUN** | **RUNTIME (Seconds)** | **INSERTIONS** |
| **1** | 0.014445 | 1000000 |
| 2 | 0.029814 | 2000000 |
| 3 | 0.064136 | 4000000 |
| 4 | 0.117785 | 8000000 |
| 5 | 0.226887 | 16000000 |
| 6 | 0.775938 | 32000000 |
| 7 | 2.231210 | 64000000 |
| 8 | 8.756970 | 128000000 |
| 9 | 12.068500 | 256000000 |
| 10 | 29.618600 | 512000000 |

### Graph

I found it interesting that during this run, there was a large upset in the standard curve realized at around the 128,000,000 records spot. This looks like it smooths itself out over the next two runs. Again, as with all the other tests run, the deletion seems to occur in amortized O(logn) time complexity.

## Non-Linear Delete

This test uses a pre-established tree from the odd-even insert test. This tree was selected because it contained a high level of reorganization in comparison to the linear tree we previously tested on. For this test, we started at 1000 elements and doubled each run for a total of 10 runs.

### Tabular Data

|  |  |  |
| --- | --- | --- |
| **RUN** | **RUNTIME (Seconds)** | **INSERTIONS** |
| **1** | 0.000015 | 1000 |
| 2 | 0.000039 | 2000 |
| 3 | 0.000084 | 4000 |
| 4 | 0.000098 | 8000 |
| 5 | 0.000203 | 16000 |
| 6 | 0.000443 | 32000 |
| 7 | 0.000866 | 64000 |
| 8 | 0.002443 | 128000 |
| 9 | 0.003496 | 256000 |
| 10 | 0.009556 | 512000 |

### Graph

For this test, the same hiccup seemed to occur near the second last test. The graph doesn’t follow the expected curve for O(nlogn). Following in fashion of our other tests, it does seem that the algorithm runs in amortized O(logn) time.

# Structure Size

When measuring the size of the data structure. We take a measure of each node in the tree. The size of each node is then multiplied by the size of our tree to attain the overall size of our structure. Since the node only really contains 3 pieces of data: a key, and two pointers, our structure remains small. Our key is stored as a long data type so that resolves to 32 bits of data. Each pointer is 32 bits of data long as well. So we get (32 \* 3)\*n where n is the total of the nodes for our structure size calculation.

# Conclusion

When implemented by following the theoretical outline of the Splay Tree data structure, all actions tested seem to perform within the theoretical bounds laid out in many different descriptions. The use of C++ and pointers aided in keeping the speed of the structure high, and allowed for very simple traversal, and transformation. The recursive nature of the splay function operated within the theoretical bounds and overall I was pleased with the results of the project.