**International Institute of Information Technology**


# APS PROJECT REPORT

for the course

**Advance Problem Solving - CSE603**

## Splay Tree Implementation and Comparison with BST

Report Prepared By:
**Manojit Chakraborty** (2018201032)
**Shubham Das** (2018202002)
**Group Name :** The Binary

# Contents

Chapter 1

# Introduction

## 1.1 Title

Splay Tree Implementation and Performance Comparison with BST

## 1.2 Team Members

- **Manojit Chakraborty** - M.Tech CSIS - 2018201032
- **Shubham Das** - M.Tech CSIS - 2018202004

## 1.3 Deliverables

- Study the analysis of Splay Tree
- Implement Splay Tree
- Implement Binary Search Tree
- Compare performance of the Splay Tree with that of BST for different test case scenarios.
- Create an application where Splay Tree can be used for better effect.
- Make a detailed report of all the benchmarking and case studies.

Chapter 2

# Theory

## 2.1 Definition

A **splay tree** is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in **O(log n) amortized time**. For many sequences of non-random operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown. The splay tree was invented by **Daniel Sleator** and **Robert Tarjan** in 1985.

## 2.2 Operations on Splay Tree

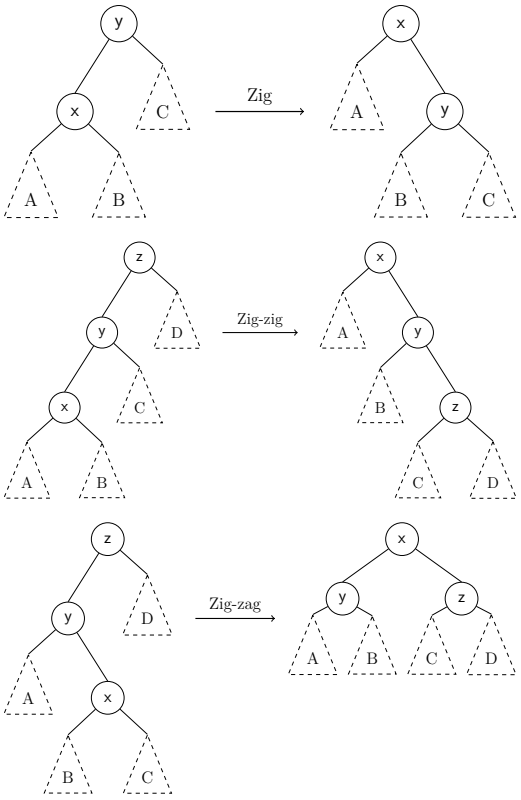
All normal operations on a binary search tree are combined with one basic operation, called splaying. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this with the basic search operation is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top. Alternatively, a top-down algorithm can combine the search and the tree reorganization into a single phase.

## 2.3 Splaying

When a node x is accessed, a splay operation is performed on x to move it to the root. To perform a splay operation we carry out a sequence of splay steps, each of which moves x closer to the root. By performing a splay operation on the node of interest after every access, the recently accessed nodes are kept near the root and the tree remains roughly balanced, so that we achieve the desired amortized time bounds. Each particular step depends on :

- Whether x is the left or right child of its parent node, y,
- whether y is the root or not, and if not
- whether y is the left or right child of its parent, z (grandparent of x).

## Splay Tree Rotations




The figure does not depict symmetric cases.

# When to splay

- `search(element)`: If the element is found in a node $u$, we splay $u$. Otherwise, we splay the node where the search terminates unsuccessfully.
- `insert(element)`: We splay the newly created node that contains the element.
- `remove(element)`: We splay the parent of the node that contains the element. If the node is the root, we splay its left child or right child. If the element is not in the tree, we splay the node where the search terminates unsuccessfully.

# Chapter 3
# Implementation

## 3.1 Project Delivery

- Study about how BST works and how insert, delete, search operations can be implemented. [ October 20-21 ]
- Implement a BST Class using Python [ October 21 ]
- Study thoroughly about Splay Tree and all the cases for its insert, delete and search operations from the online resources and books. [ October 22-25 ]
- Implement a Splay Tree class by extending the BST class methods of insert, search and delete. [ October 25-26 ]
- Compare the performance of the Splay Tree with that of BST for these different scenarios (Initial Thought) -
  - Insert in increasing order, Search in increasing order, Delete in increasing order.
  - Insert in increasing order, Search in decreasing order, Delete in decreasing order
  - Insert in random order, Search in same random order, Delete in same random order.
  - Insert in random order, search in different random order.
  - Insert in random order, Search same 4 elements randomly.
  - Insert in random order, search same 10 elements randomly.
  - Insert in random order, Delete 1000 elements randomly.

  Here, the tree size will vary from 5000 to 1000000 approximately.
- Calculate all these performance measures for a number of times, generate the average values for each type of measures for both the trees, with varying tree size. [ October 27-30 ]

- Create plots for the performance measure values in each test case for Splay Tree and BST using Plotly in Jupyter Notebook and generate an overall Performance Comparison Report. [ October 31-November 3 ]
- Find an application where Splay Tree is used extensively and test the application with our own implementation and collect performance measures and results. [ November 3-7 ]
- Create a Music Search Application using the Million Song Dataset (https://labrosa.ee.columbia.edu/millionsong/), where an user can
  - insert new titles from singers into the library.
  - can search for a particular singer and the search result will be all the songs from that corresponding singer.
  - delete a singer from the library.

  where each of these operations will work in amortized O(logn) time.

## 3.2 Technologies Used

- Python
- Jupyter Notebook
- Visualization tools like Plotly, Matplotlib

## 3.3 Online Resources

- https://en.wikipedia.org/wiki/Splay_tree
- https://www.geeksforgeeks.org/splay-tree-set-1-insert/
- https://www.geeksforgeeks.org/splay-tree-set-2-insert-delete/
- https://www.cs.usfca.edu/ galles/visualization/SplayTree.html

## 3.4 Performance Testing Implementation

We will test the performance of the Splay Tree with that of BST for these different scenarios (Initial Thought) -

- Insert in increasing order, Search in increasing order, Delete in increasing order.
- Insert in increasing order, Search in decreasing order, Delete in decreasing order
- Insert in random order, Search in same random order, Delete in same random order.
- Insert in random order, search in different random order.
- Insert in random order, Search same 4 elements randomly.
- Insert in random order, search same 10 elements randomly.
- Insert in random order, Delete 1000 elements randomly.

Here, the tree size will vary from 5000 to 1000000 approximately. We will take records of Time taken, Number of Comparisons, Space taken for each of these test cases by running them a number of times and then calculate average of each value for each of the test cases.

## 3.5 Application Implementation

The application that we implemented using Splay tree is - creating a Music Search Library Operation, where the library contains Name of Singers and their corresponding Song(s). The database we used to create our initial music library is - **Million Song Dataset**. The Million Song Dataset is a freely-available collection of audio features and metadata for a million contemporary popular music tracks. It started as a collaborative project between The Echo Nest and LabROSA. It was supported in part by the NSF.

We downloaded the dataset from this website - https://labrosa.ee.columbia.edu/millionsong/ . We took only the Name of the Singer and Title Columns of the dataset, preprocess it by removing Null Value rows. Then we inserted each singer name as a string into a node and creating a list on that node for all the titles done by the singer, inside our splay tree. We created a Menu Driven Command Line Application, where

the user can search for a Singer Name and get all the titles of that singer from the existent library, insert a new singer and his/her corresponding titles as a new entry into the library, delete a singer and all his/her songs from the library. User can also see the whole library by using an Inorder Traversal on the whole Splay Tree.

## 3.6 Repository of work being committed

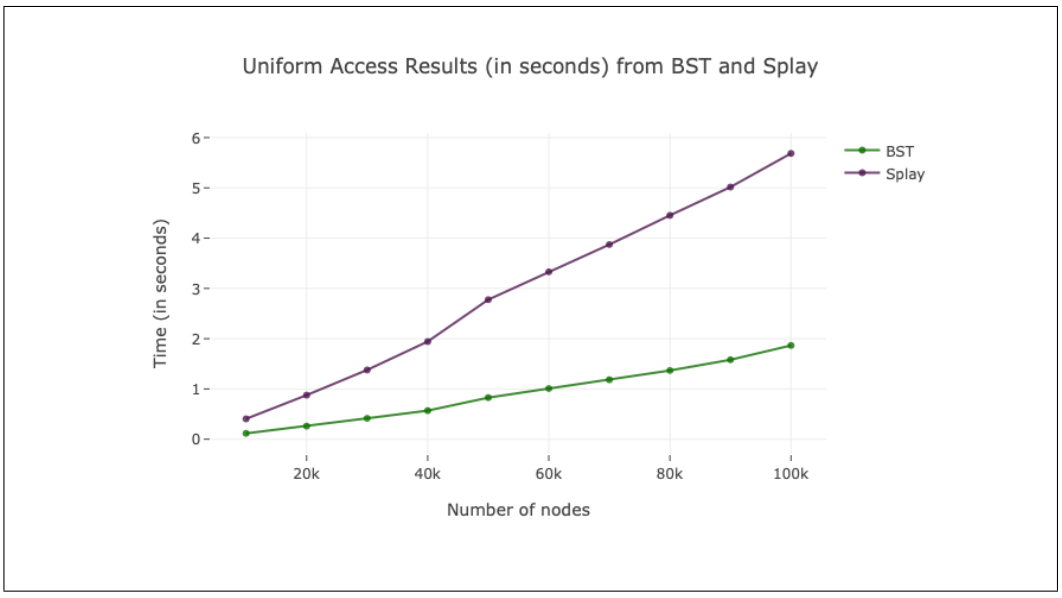
**Github Repository :** https://github.com/manojit32/APS_Project_Splay_Tree

Chapter 4

# Results of Performance Testing

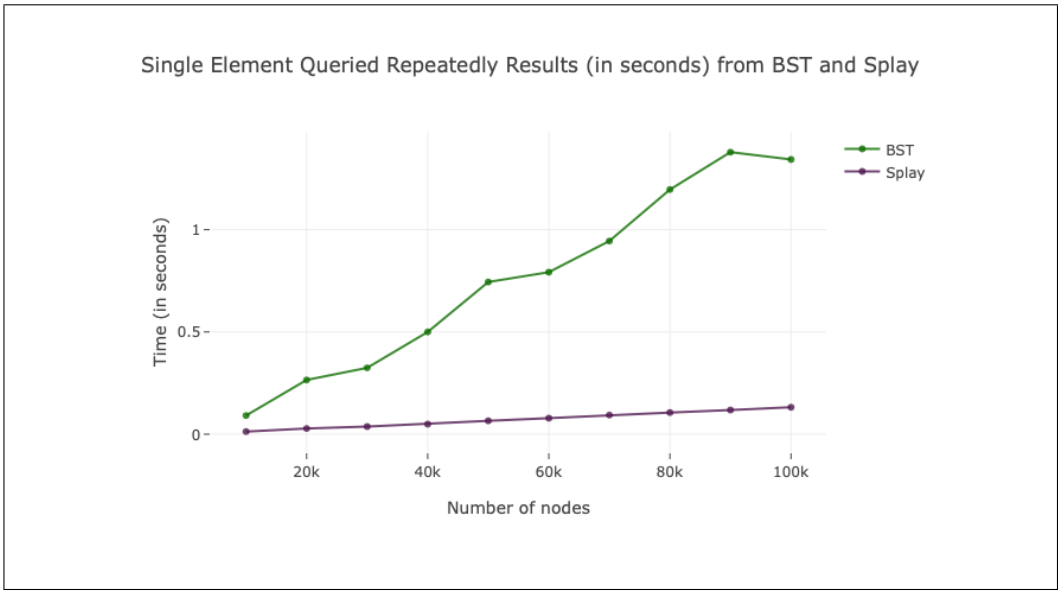
## 4.1 Uniform Access

The most unsurprising results are for the uniform access test cases. Splay trees performed far worse than standard BSTs with an average time of 0.318 seconds, with standard BSTs clocking in at 0.0602 seconds. This is exactly what I expected to happen: with random patterns of access, splay trees will be constantly changing shape randomly, and since about half of the splay operations will be logarithmic, this has a lot of overhead. We suppose for the small (10000 to be exact) number of elements, the relative times are small, but over longer sequences of operations with more elements, this difference potentially became sizable.



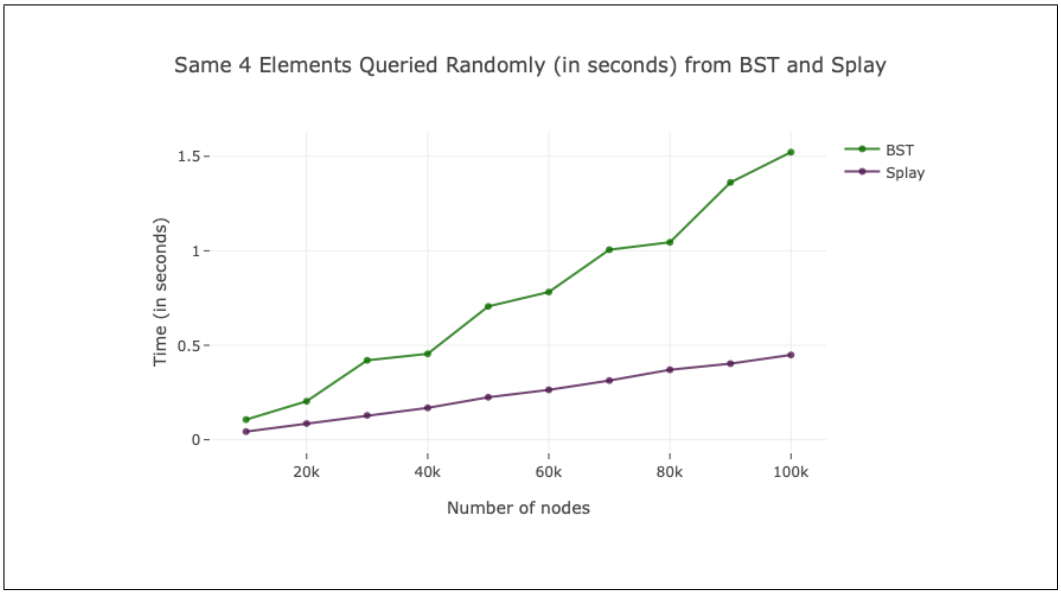
## 4.2 Single Element Queried Repetedly

The next test case we ran was to query a single element repeatedly. we decided to randomly choose an element in the range from 0 to the max element in the tree, and then query the two trees using this element. This showed the expected result: the splay tree beat out the Binary Search Tree, because after splaying, the element moves to the root of the tree, resulting in much lesser time to search the element from the splay tree than BST.



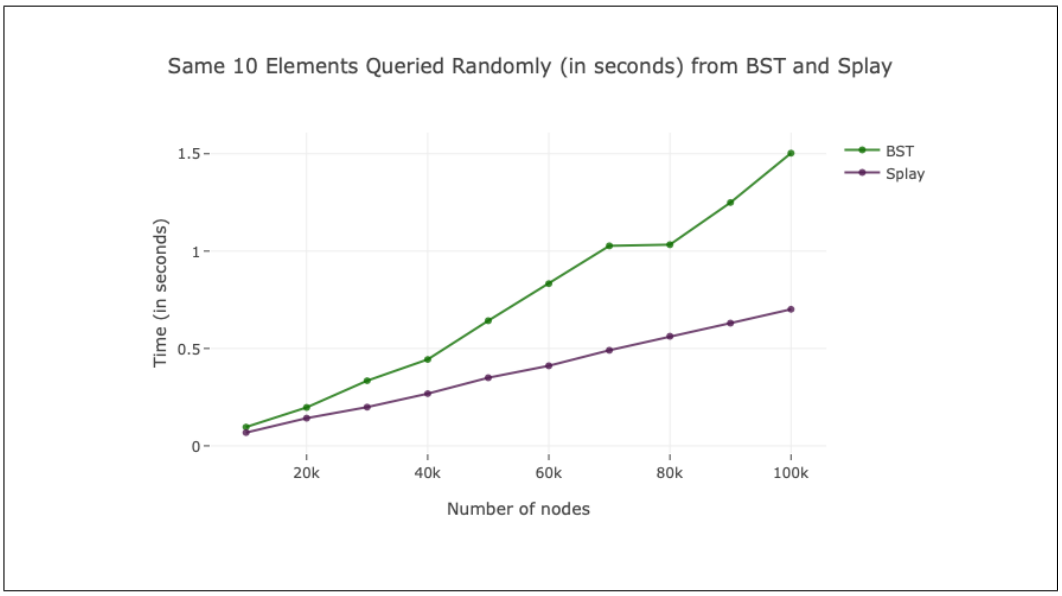
## 4.3 Same 4 elements repeatedly

The next test case we ran was to query the same four elements repeatedly. At first, we always queried the first four elements in the array that were loaded into the trees; however, in this case, the standard BST actually consistently outperformed Splay trees. My best guess as to why this was happening is that there is something about insertion rebalancing in Splay trees that place the elements that are inserted first further away from the root, though I cannot see exactly why. After noticing this, I randomly chose four adjacent elements in the data array (as opposed to just the first four), and repeatedly queried the trees using these in the same order. This also produced an interesting result; splay trees did not perform better than the other tree. The reason for this is that since a different number was being queried every time, splaying was occurring with every access, which increased the overhead of splay insertion. Finally, I decided to randomly choose four elements in the range from 0 to the max element in the tree, produce a long, random sequence of these four elements, and then query the three trees using this sequence. This showed the expected result: the splay tree beat out the Binary Search Tree.



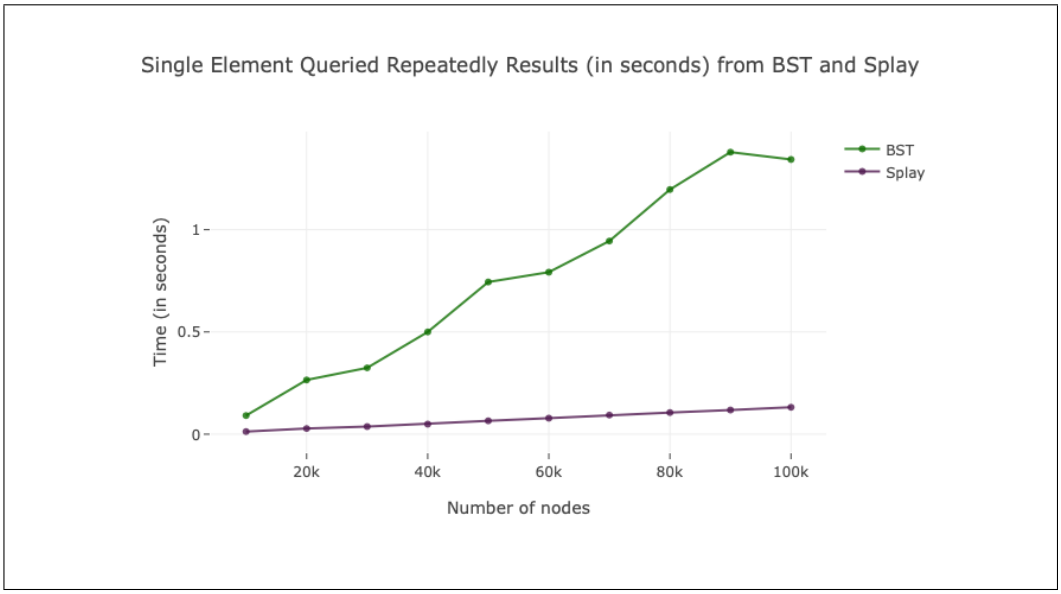
## 4.4 Same 10 elements repatedly

The next test case I ran was exactly the same as the previous one, but instead of querying four elements repeatedly, I queried ten. This produced same expected result: the Splay tree outperformed Binary Search trees with a time of 0.0603 seconds, with the standard BST at 0.0737 seconds.



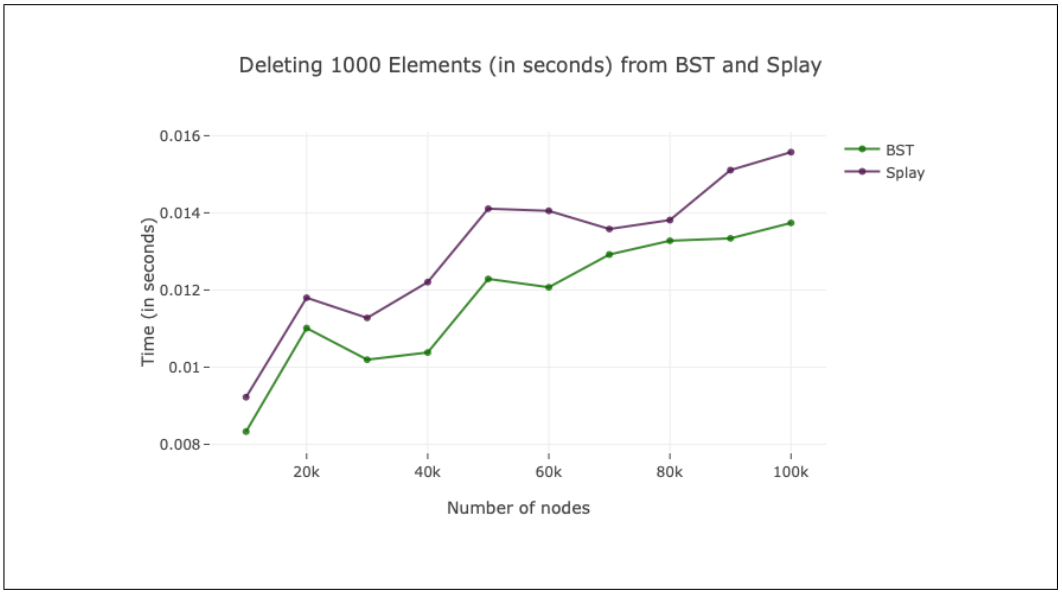
## 4.5 Single Element Queried Repetedly

The next test case we ran was to query a single element repeatedly. we decided to randomly choose an element in the range from 0 to the max element in the tree, and then query the two trees using this element. This showed the expected result: the splay tree beat out the Binary Search Tree, because after splaying, the element moves to the root of the tree, resulting in much lesser time to search the element from the splay tree than BST.

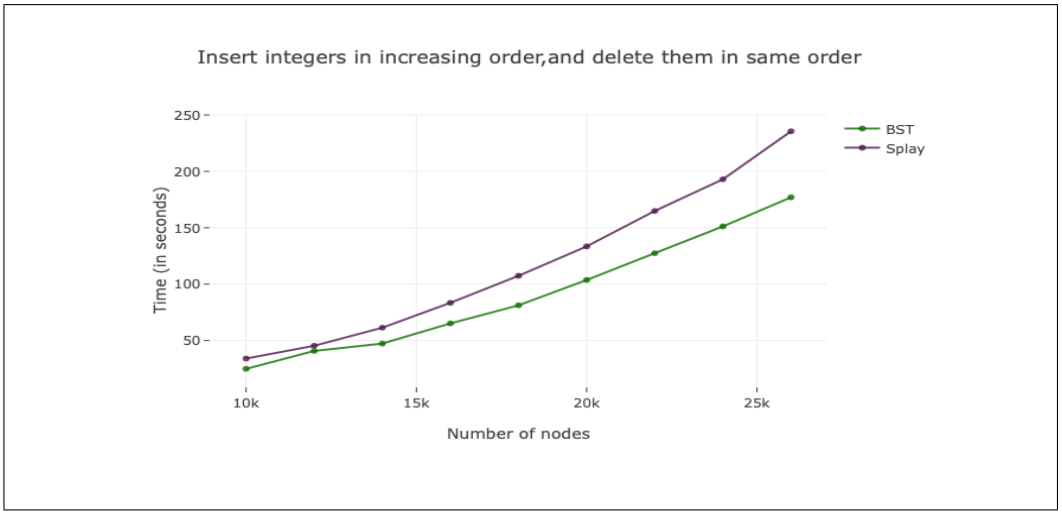


## 4.6 Deleting 1000 random elements

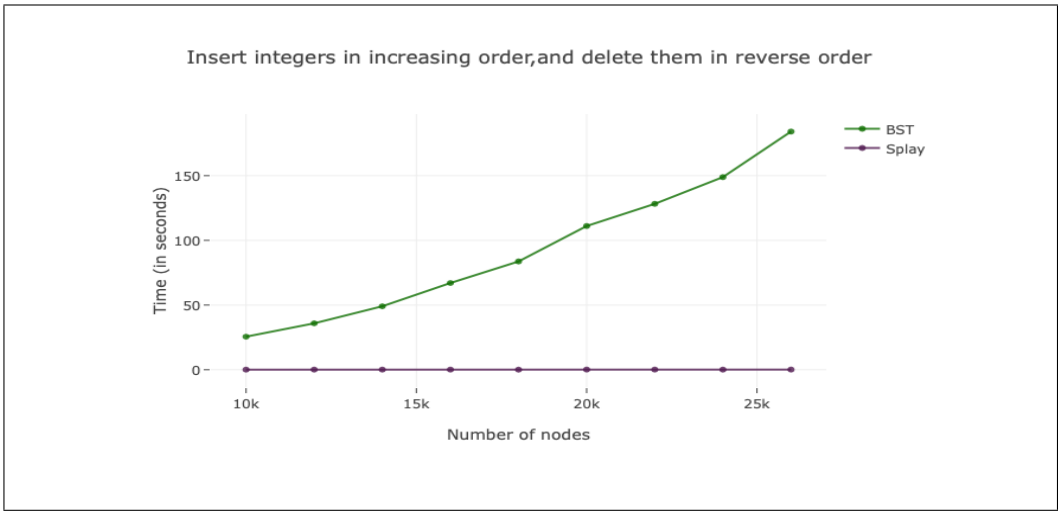
The final test case we ran was to perform the same sequence of 1000 deletions on each of the trees. This test also surprised me: the splay tree at 0.0131 seconds and the standard BST at 0.01306 seconds are pretty neck to neck. I expected the splay tree to perform much worse than the standard BST, since splaying and deletion rebalancing are both logarithmic. I suppose the reason for the results as they are is that Splay trees are more balanced, so the height of the tree is significantly less than that of the other two trees, giving a smaller hidden constant in the O(log n) upper bound on deletion.



## 4.7 Insertion and Deletion in increasing order

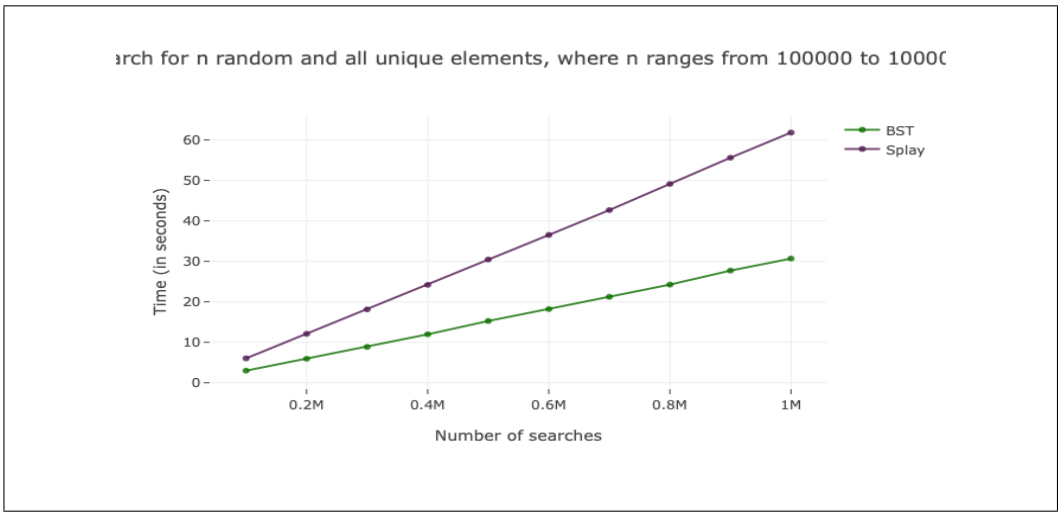



## 4.8 Insertion in incraesing, deletion in reverse order

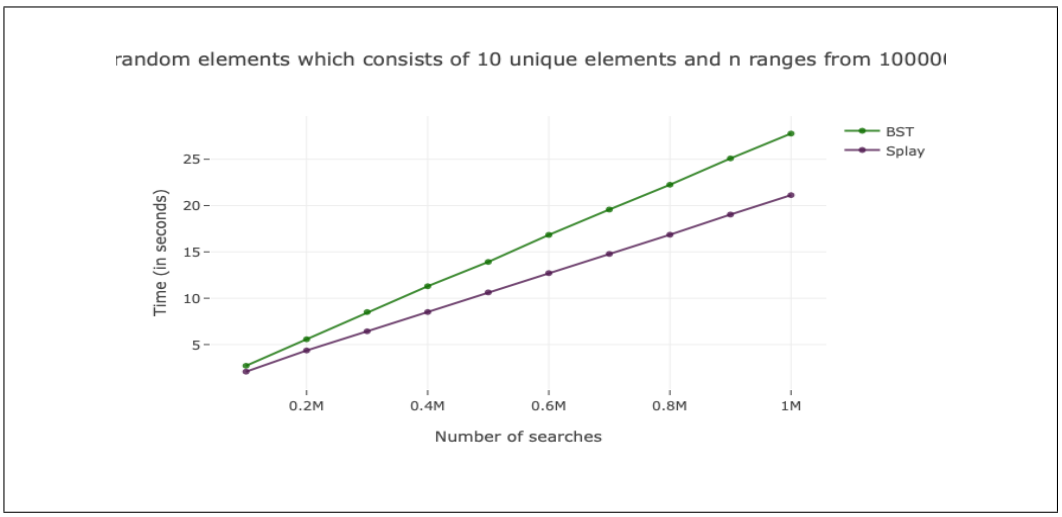


## 4.9 Music Library Application Testing

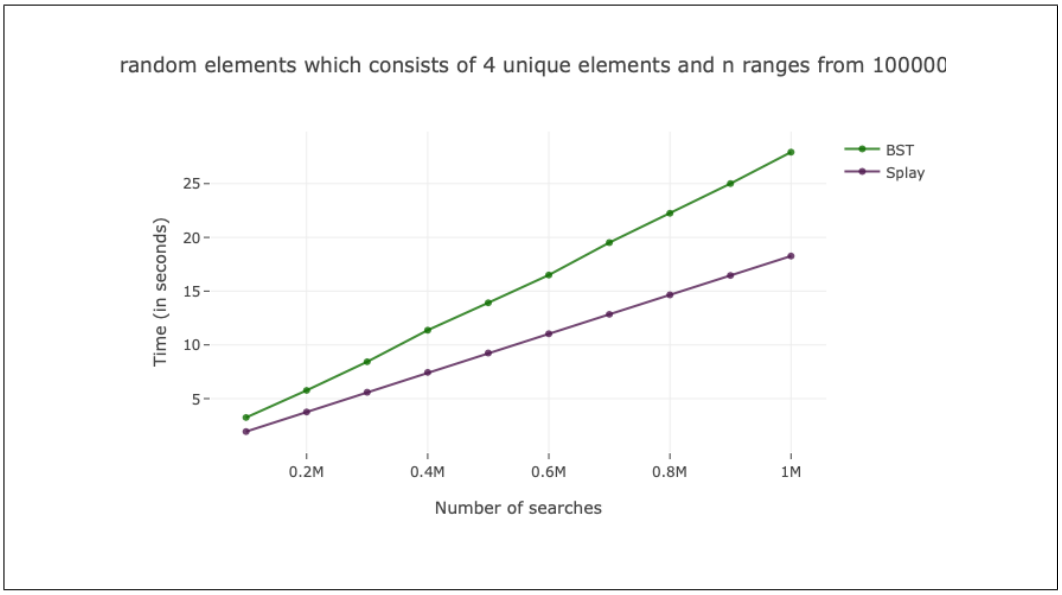
### 4.9.1 Search for n random and all unique elements, where n ranges from 100000 to 1000000




### 4.9.2 Search for n random elements which consists of 10 unique elements and n ranges from 100000 to 1000000



### 4.9.3 Search for n random elements which consists of 4 unique elements and n ranges from 100000 to 1000000




- For the first case, as all the elements are random and unique, Binary Search Tree becomes much faster than Splay tree as the set of search elements grow larger, because splaying operation takes O(logn) extra time for each iteration in case of Splay Tree.
- For the second and third case, number of unique elements becomes limited (4 and 10) , so Splay Tree works much faster than BST as expected, because previously searched element comes to the root of the Splay tree, whereas in BST, no such case happens and the search goes to inner depths of the search tree every time.

Chapter 5

# End user Document

## 5.1 Splay Tree

- Create a Splay Tree by using these following commands in Python Shell-

  *import splay*
  *splaytree = splay.Splay()*

- For insertion into the Splay Tree, write -

  *splaytree.insert(value)*

- To search a value within the Splay Tree, write -

  *splaytree.search(value)*

  which will return True if value exists, False otherwise.

- For deletion of a value from the Splay Tree, write -

  *splaytree.remove(value)*

  which will successfully delete the value if it exists in the tree. If the value is not present in the tree, it will raise an error.

## 5.2 Binary Search Tree

- Create a Binary Search Tree by using these following commands in Python Shell-

  *import bst*
  *bstree = bst.BST()*

- For insertion into the Binary Search Tree, write -

  *bstree.insert(value)*

- To search a value within the Binary Search Tree, write -

  *bstree.search(value)*

  which will return True if value exists, False otherwise.

- For deletion of a value from the Binary Search Tree, write -

  *bstree.remove(value)*

  which will successfully delete the value if it exists in the tree. If the value is not present in the tree, it will raise an error.

## 5.3 Performance Testing

To run an overall performance testing of Splay Tree against BST, run the following command in the terminal -

*import test_suite*

*test_suite.avg_uni1()*
*test_suite.avg_pretty_uneven1()*
*test_suite.avg_pretty_uneven41()*
*test_suite.avg_pretty_uneven101()*
*test_suite.avg_del1()*

*import test_suite2*

*test_suite2.insert_del_same_plot()*
*test_suite2.insert_de_opp_plot()*

## 5.4 Application

To run the Music Search Library Application built using Splay Tree, User has to do the following -

# Application

November 11, 2018

```
In [8]: import demo1
```

```
1
```

```
In [11]: demo1.face()
```

```
Enter 1 to enter new record, 2 to search by artist name, 3 to delete, 4 to exit:
2
Enter artist name: Enrique Iglesias
Contigo
Bailamos
Maybe
Roamer
Can You Hear Me
Quizás
Dímelo
Love To See You Cry
Love To See You Cry
Miente
No Puedo Mas Sin Ti (I'm Your Man)
Don't Turn Off the Lights
Can You Hear Me
Experiencia Religiosa
Roamer
Can You Hear Me
Para Qué La Vida
California Callin'
Escape
Ring My Bells
Can You Hear Me
Escape
California Callin'
Para Qué La Vida
Cosas Del Amor
Bailamos
Can You Hear Me
Dímelo
```

```
Enter 1 to enter new record, 2 to search by artist name, 3 to delete, 4 to exit:
1
Enter artist name: Enrique Iglesias
Enter song name: Hero
Successfully inserted.
Enter 1 to enter new record, 2 to search by artist name, 3 to delete, 4 to exit:
2
Enter artist name: Enrique Iglesias
Contigo
Bailamos
Maybe
Roamer
Can You Hear Me
Quizás
Dímelo
Love To See You Cry
Love To See You Cry
Miente
No Puedo Mas Sin Ti (I'm Your Man)
Don't Turn Off the Lights
Can You Hear Me
Experiencia Religiosa
Roamer
Can You Hear Me
Para Qué La Vida
California Callin'
Escape
Ring My Bells
Can You Hear Me
Escape
California Callin'
Para Qué La Vida
Cosas Del Amor
Bailamos
Can You Hear Me
Dímelo
Hero
Enter 1 to enter new record, 2 to search by artist name, 3 to delete, 4 to exit:
4
Exiting application.
```

```
Out[11]: 0
```