A

Project Report

On

"Splay Tree and its comparison with AVL Tree"

(CSE603 – Advanced Problem Solving Project)

Prepared by :

Vatsal Soni (2018201005)

Darshan Kansagara (2018201033)

Under the Supervision of

Prof. Venkatesh Choppella

TA : Sunchit Sharma

GitHub Repository Link

https://github.com/darshank15/splaytree

Table of Contents

CHAPTER 1 INTRODUCTION

1.1 AVL Tree:

AVL tree is a self-balancing binary search tree. The heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. In a binary tree the balance factor of a node N is defined to be the height difference of its two child subtrees.

BalanceFactor(N) := Height(RightSubtree(N)) – Height(LeftSubtree(N))
BalanceFactor(N) ∈ **{–1,0,+1}** holds for every node N in the AVL tree.

1.2 Splay Tree:

A splay tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. 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. All normal operations on a binary search tree are combined with one basic operation, called splaying.

Splaying: It is an operation of the tree for a certain element rearranges the tree so that the element is placed at the root of the tree.

The main idea of splay tree is to bring the recently accessed item to root of the tree, this makes the recently searched item to be accessible in O(1) time if accessed again. The idea is to use locality of reference.

CHAPTER 2 IMPLEMENTATION DETAILS:

2.1 AVL Implementation :

AVL Operation:

1. **Insertion**

To insert node w in AVL tree:

a. Perform standard BST insertion.

b. Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.
Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
I) y is left child of z and x is left child of y (Left Left Case)
II) y is left child of z and x is right child of y (Left Right Case)
III) y is right child of z and x is right child of y (Right Right Case)
IV) y is right child of z and x is left child of y (Right Left Case)

I. Left Left Case

```
T1, T2, T3 and T4 are subtrees.
         z                                      y
        / \                                   /   \
       y   T4      Right Rotate (z)          x      z
      / \          - - - - - - - - ->      /  \    /  \
     x   T3                               T1  T2  T3  T4
    / \
  T1   T2
```

II. **Left Right Case**

```
     z                               z                           x
    / \                            /   \                        /  \
   y   T4  Left Rotate (y)        x    T4  Right Rotate(z)    y      z
  / \      - - - - - - - - ->    /  \      - - - - - - ->  / \    / \
T1   x                          y    T3                    T1  T2 T3  T4
    / \                        / \
  T2   T3                    T1   T2
```

III. **Right Right Case**

```
  z                                y
 /  \                            /   \
T1   y     Left Rotate(z)       z      x
    /  \   - - - - - - - ->    / \    / \
   T2   x                     T1  T2 T3  T4
       / \
     T3  T4
```

IV. **Right Left Case**

```
   z                            z                            x
  / \                          / \                          /  \
T1   y   Right Rotate (y)    T1   x      Left Rotate(z)   z      y
    / \  - - - - - - - - ->     /  \   - - - - - - -> / \    / \
   x   T4                      T2   y                  T1  T2  T3  T4
  / \                              /  \
T2   T3                          T3   T4
```

2. Deletion

To delete node w.

a. Preform standard BST deletion.

b. Then perform rebalancing to satisfied BF for each node.

3. Search

Searching for a specific key in an AVL tree can be done the same way as that of a normal unbalanced binary search tree.

2.2 Splay Implementation :

1. **Insertion :**

   To insert key k in AVL tree:

   a. Splay the given key k. If k is already present, then it becomes the new root. If not present, then last accessed leaf node becomes the new root.
   b. If new root's key is same as k, don't do anything as k is already present.
   c. Else allocate memory for new node and compare root's key with k.
      i. If k is smaller than root's key, make root as right child of new node, copy left child of root as left child of new node and make left child of root as NULL.
      ii. If k is greater than root's key, make root as left child of new node, copy right child of root as right child of new node and make right child of root as NULL.

a. Zig: The node has no grandparent. Node is a left child of root, we do a right rotation.

b. Zag: The node has no grandparent. Node is a right child of root, we do a left rotation.

c. When Node has Grandparent

I. When given node is left child of parent and parent is left child of grandparent **(Zig-Zig rotation).**

II. When given node is right child of parent and parent is right child of grandparent **(Zag-Zag rotation).**

III. When given node is left child of parent and parent is right child of grandparent **(Zag-Zig rotation).**

IV. When given node is right child of parent and parent is left child of grandparent **(Zig-Zag rotation).**

```
Zig-Zig (Left Left Case):

       G                        P                            X
      / \                     /   \                         / \
     P  T4   rightRotate(G)  X     G     rightRotate(P)  T1    P
    / \      ============>  / \   / \    ============>        / \
   X  T3                   T1  T2 T3 T4                      T2  G
  / \                                                            / \
 T1 T2                                                          T3  T4
```

```
  Zag-Zag (Right Right Case):

  G                           P                           X
 /  \                       /   \                        / \
T1   P     leftRotate(G)  G     X     leftRotate(P)     P   T4
    / \    ============> / \   / \    ============>    / \
   T2   X               T1 T2 T3 T4                   G   T3
       / \                                           / \
      T3 T4                                        T1  T2
```

```
Zag-Zig (Left Right Case):

       G                         G                            X
      / \                      /   \                        /   \
     P   T4  leftRotate(P)    X     T4    rightRotate(G)   P     G
   /  \      ============>  / \          ============>   / \   /  \
  T1   X                   P  T3                       T1  T2 T3  T4
      / \                 / \
    T2  T3              T1   T2
```

```
Zig-Zag (Right Left Case):

  G                          G                           X
 /  \                       /  \                       /   \
T1   P    rightRotate(P)  T1   X     leftRotate(P)    G     P
    / \   =============>      / \    ============>   / \   / \
   X  T4                    T2   P                 T1  T2 T3  T4
  / \                           / \
 T2  T3                       T3  T4
```

2. **Deletion :**

   To delete node having key k.

   a. Splay the given key k. If k is present, then it becomes the new root. If not present, then last accessed leaf node becomes the new root.
   b. If new root's key is not same as k, then node having key k is not present.
   c. Else the key k is present.
      i. Split the tree into two trees Tree1 which is root's left subtree and Tree2 which is root's right subtree and delete the root node.
      ii. Let the root's of Tree1 and Tree2 be Root1 and Root2 respectively.
      iii. If Root2 is NULL: Return Root1.
      iv. Else, Splay the minimum node (node having the minimum value) of Tree2.
      v. After the Splay procedure, make Root1 as the left child of Root2 and return Root2.

3. **Search :**

   a. The search operation in Splay tree does the standard BST search.
   b. It also splays (move a node to the root).
      i. If the search is successful, then the node that is found is splayed and becomes the new root.
      ii. Else the last node accessed prior to reaching the NULL is splayed and becomes the new root.
   c. If root's key is same as searching node's key then key found else key not found.

CHAPTER 3 COMPARISON BETWEEN AVL AND SPLAY

3.1 Single Operation analysis :

Worst Case		
	AVL	**Splay**
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)
Search	O(log n)	O(n)

Average case		
	AVL	**Splay**
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)
Search	O(log n)	O(n)

Best Case		
	AVL	**Splay**
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)
Search	O(1)	O(1)

- **Space Complexity : AVL – O(n) , Splay- O(n)**

3.2 Amortized analysis :

Amortized		
	AVL	**Splay**
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)
Search	O(log n)	O(log n)

3.3 Difference AVL vs Splay

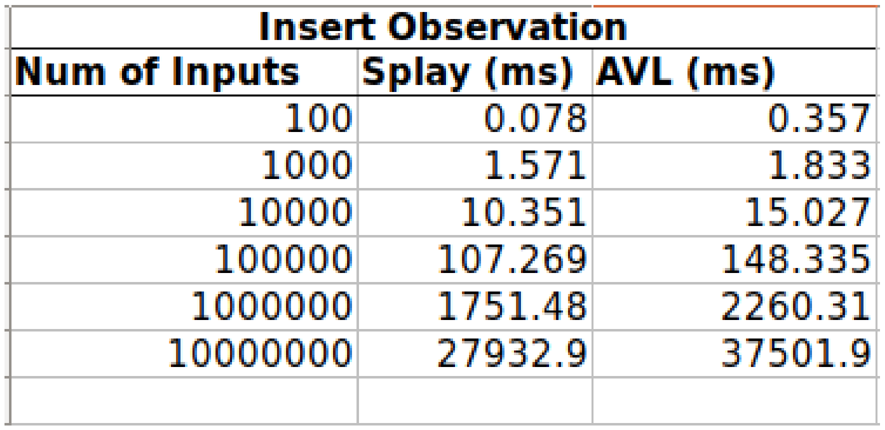
a. In an AVL tree, the shape of the tree is constrained at all times such that the tree shape is balanced, meaning that the height of the tree never exceeds O(log n). This shape is maintained on insertions and deletions, and does not change during lookups. Splay trees, on the other hand, maintain efficient by reshaping the tree in response to lookups on it. That way, frequently-accessed elements move up toward the top of the tree and have better lookup times. The shape of splay trees is not constrained, and varies based on what lookups are performed.

b. one key difference between the structures is that AVL trees guarantee fast lookup (O(log n)) on each operation, while splay trees can only guarantee that any sequence of n operations takes at most O(n log n) time. This means that if you need real-time lookups, the AVL tree is likely to be better. However, splay trees tend to be much faster on average, so if you want to minimize the total runtime of tree lookups, the splay tree is likely to be better. Additionally, splay trees support some operations such as splitting and merging very efficiently.

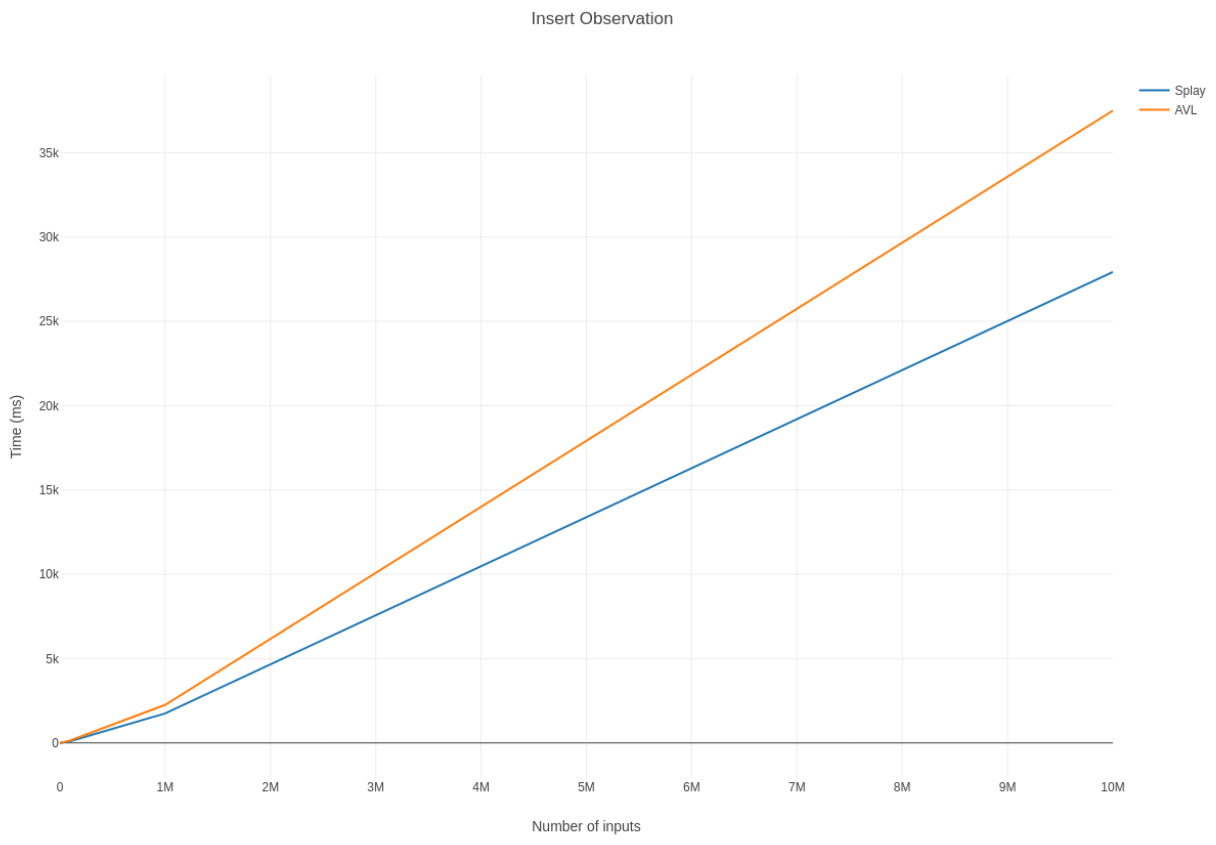
c. AVL tree insertion, deletion, and lookups take O(log n) time each. Splay trees have these same guarantees, but the guarantee is only in an amortized sense. Any long sequence of operations will take at most O(n log n) time, but individual operations might take as much as O(n) time.

3.4 Graphical Analysis of Splay and AVL Tree

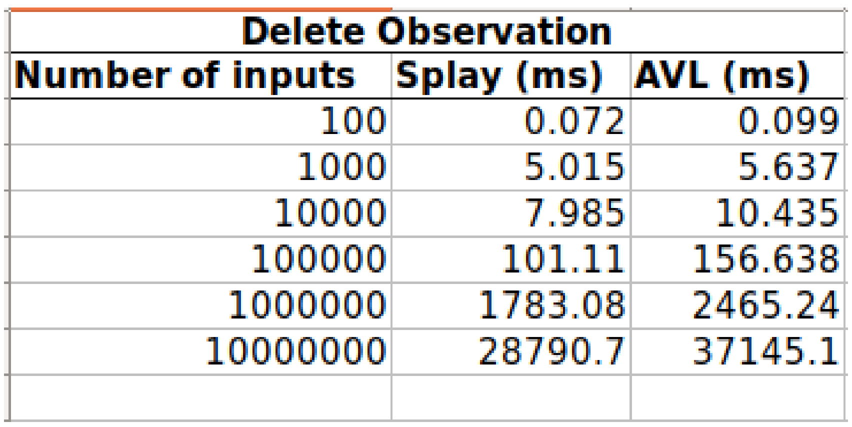
a. Insert operation



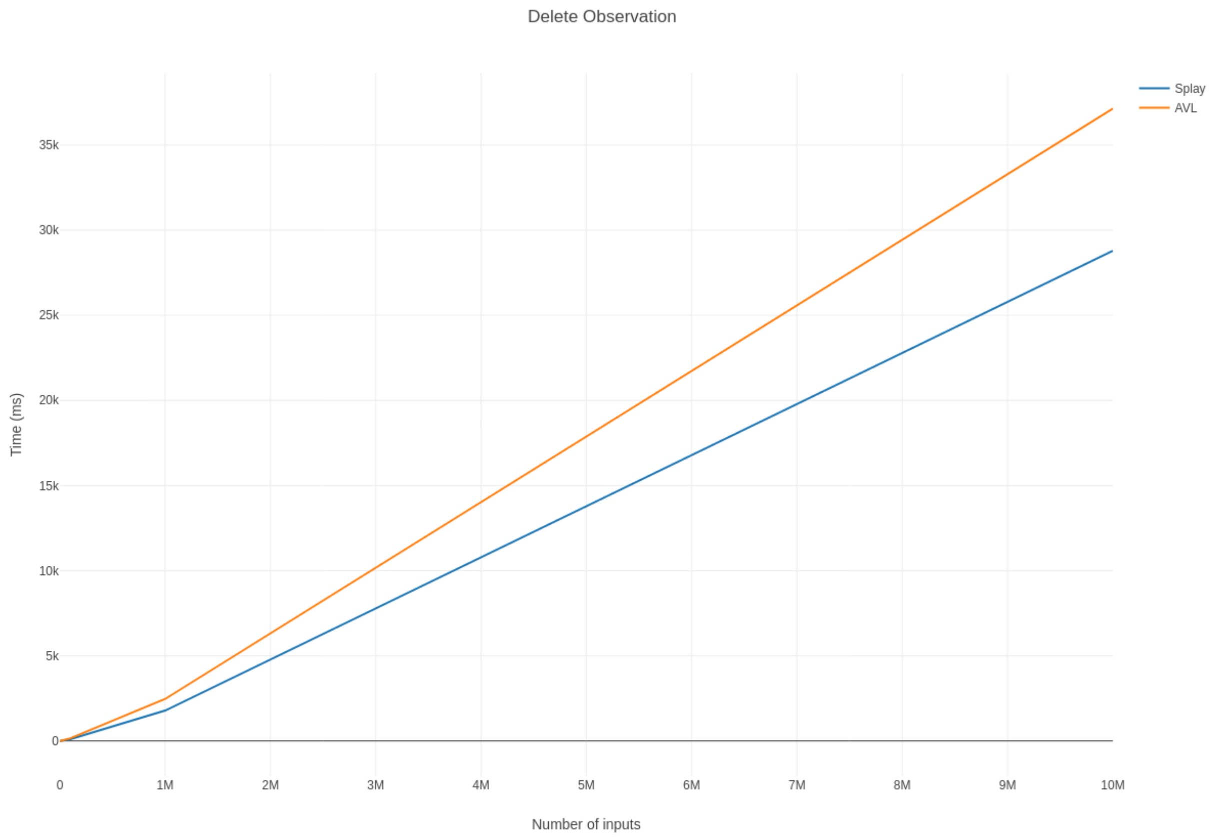
Insert Observation		
Num of Inputs	**Splay (ms)**	**AVL (ms)**
100	0.078	0.357
1000	1.571	1.833
10000	10.351	15.027
100000	107.269	148.335
1000000	1751.48	2260.31
10000000	27932.9	37501.9



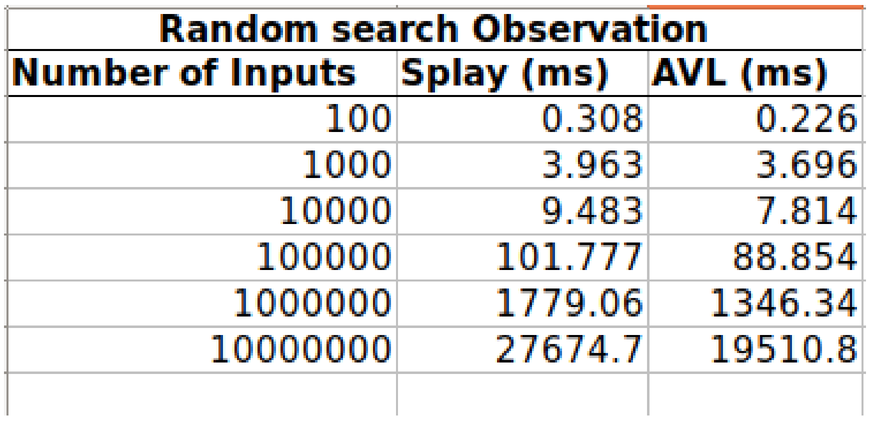
b. Delete Operation



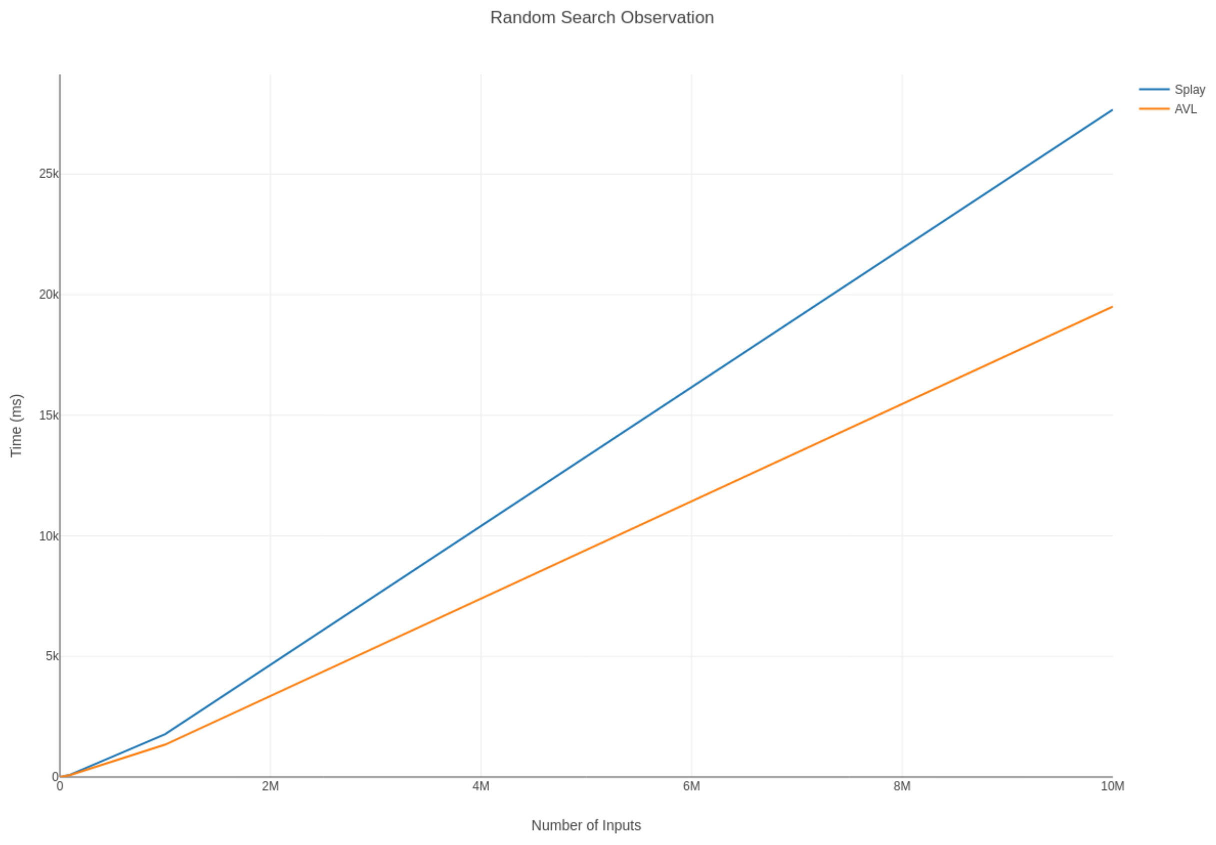
Delete Observation		
Number of inputs	Splay (ms)	AVL (ms)
100	0.072	0.099
1000	5.015	5.637
10000	7.985	10.435
100000	101.11	156.638
1000000	1783.08	2465.24
10000000	28790.7	37145.1



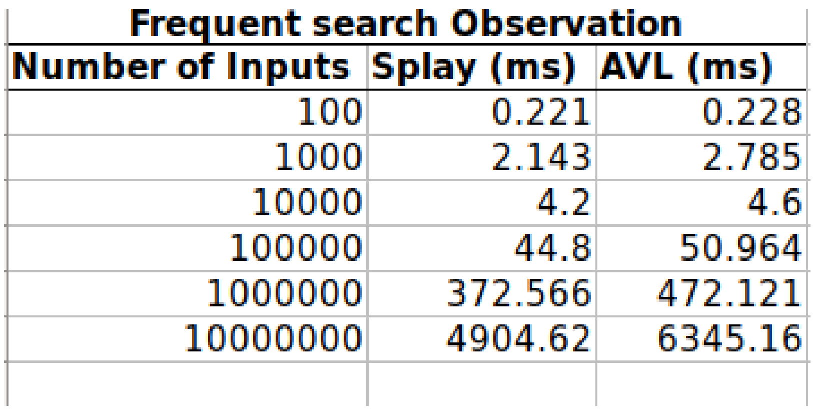
c. Random Search Operation



Random search Observation		
Number of Inputs	Splay (ms)	AVL (ms)
100	0.308	0.226
1000	3.963	3.696
10000	9.483	7.814
100000	101.777	88.854
1000000	1779.06	1346.34
10000000	27674.7	19510.8

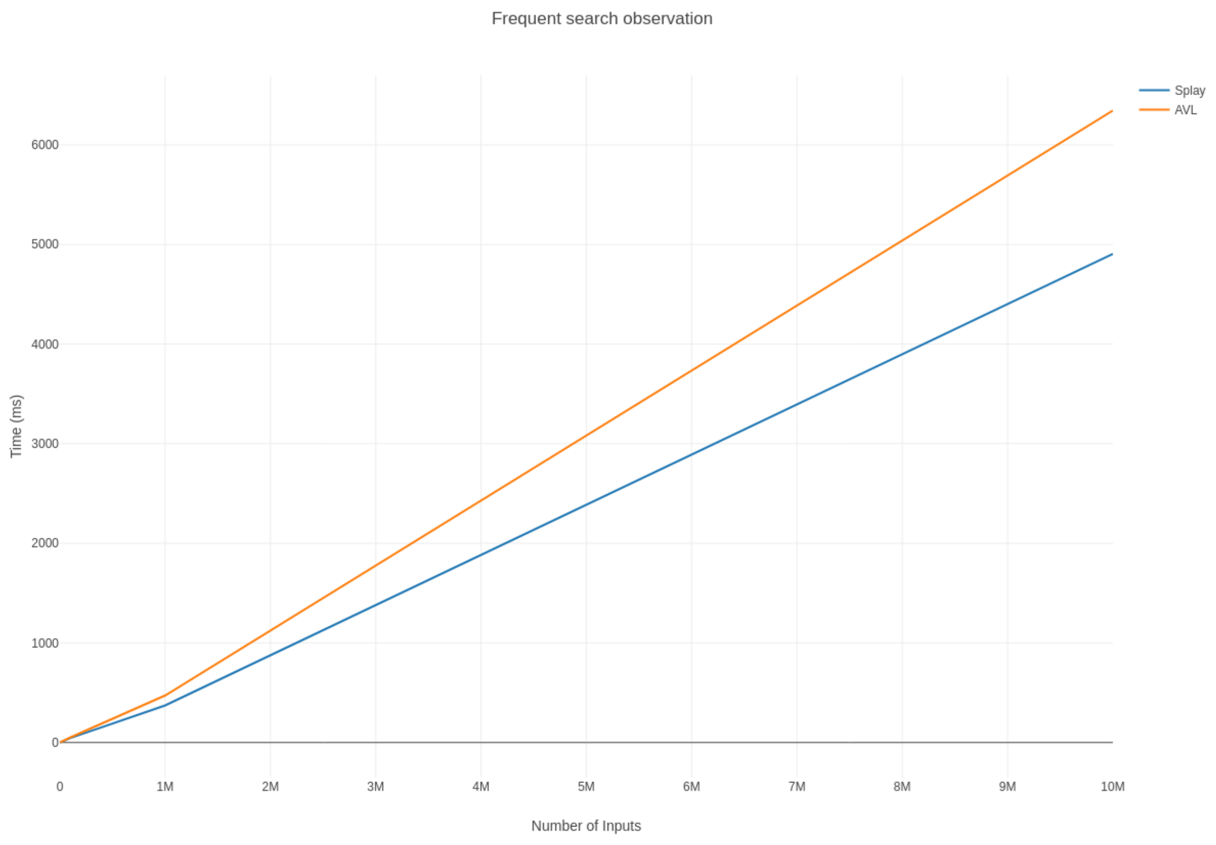


d. Frequent search operation



Frequent search Observation

Number of Inputs	Splay (ms)	AVL (ms)
100	0.221	0.228
1000	2.143	2.785
10000	4.2	4.6
100000	44.8	50.964
1000000	372.566	472.121
10000000	4904.62	6345.16



CHAPTER 4 APPLICATION OF SPLAY TREE

A **splay tree** is an efficient implementation of a balanced binary search tree that takes advantage of locality in the keys used in incoming lookup requests. For many applications, there is excellent key locality.

1. A good example is a network router. A network router receives network packets at a high rate from incoming connections and must quickly decide on which outgoing wire to send each packet, based on the IP address in the packet. The router needs a big table (a map) that can be used to look up an IP address and find out which outgoing connection to use. If an IP address has been used once, it is likely to be used again, perhaps many times. So we can build the above look up table using splay tree. if we have look up for a ip address once it will come on top of tree, so next time that ip address come again ,we can find its entry on top and don't have to go till bottom.

2. Splay trees are typically used in the implementation of caches, memory allocators, garbage collectors, data compression, ropes (replacement of string used for long text strings), in Windows NT (in the virtual memory, networking, and file system code) etc.

3. Splay tree are used in the gcc compiler and GNU C++ library, the sed string editor,the most popular implementation of Unix malloc, Linux loadable kernel modules, and in much other software.

4. Splay tree doesn't required to maintain balance information (such as Balance factor in AVL & color in RBT). So It can be used in memory constraint application.

CHAPTER 5 DRAWBACK OF SPLAY TREE

1. For uniform access, a splaytree's performance will be considerably (although not asymptotically) worse than a somewhat balanced simple binary search tree.

2. One worst case issue with the basic splay tree algorithm is that of sequentially accessing all the elements of the tree in the sorted order. This leaves the tree completely unbalanced (this takes n accesses - each a O(logn) operation).

3. We normally think of a search operation as nondestructive, that is, it doesn’t modify the tree, but in the case of splay trees, it does. The most obvious situation where this can become an issue is in a multithreading environment.

4. Splay trees have the disadvantage that individual operations can be expensive. For example, it is possible for a search to take time O(n) the first time it runs

CHAPTER 6 TESTING

6.1 Testing plan

1. After every operation make sure that inorder traversal of a tree must be sorted.
2. Comparing output of insertion and deletion with visualization.
3. Using random generator file, generate different sets of input and perform insertion and deletion on splay tree and AVL tree (try to cover all corner cases).
4. Test two type of search operation i.e. Random search and frequent element search.
    a. For random search, generate random numbers and insert it into the tree and store it in array/vector. Randomly choose some elements from array/vector and perform search operation.
    b. For frequent search choose only 10 elements randomly and then perform search operation many time (thousand time) on splay tree to search element from above sets (from 10 elements).

6.2 Test cases

Note: We have taken standard AVL implementation which always gives correct result for all operation. To check correctness of splay tree we are comparing output of every operation of splay tree with AVL tree.

Test Case ID	Function Name	Test Case	Expected Results	Actual Result	Pass/Fail
1	inorderTest()	This function performs inorder traversal on tree.	Inorder traversal must be in sorted order.	Got correct inorder traversal.	Pass
2	searchTest()	This function performs search operation with specific set of elements on splay tree and AVL tree and compare its output.	Search result for each element in splay tree and AVL tree must be same.	Got correct result for each search operation.	Pass
3	deleteTest()	This function performs delete operation with specific set of elements on splay tree and AVL tree and compare its output	Deletion result for each element in splay tree and AVL tree must be same.	Got correct result for each delete operation.	Pass

CHAPTER 7 USER MANUAL

7.1 Requirements

Platform: Linux

Software Requirement:

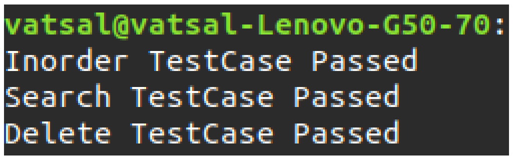
1. Python version 2.7
    a. To install python : sudo apt-get install python2.7 python-pip
2. G++ compiler
    a. To install G++ : sudo apt-get install g++
3. Matplotlib library for graph
    a. To install Matplot library : pip install matplotlib

7.2 How to run project

a. To compile splay tree and AVL tree
    i. g++ splay.cpp –o splay
    ii. g++ avl.cpp –o avl
b. To Run splay tree and AVL tree
    i. ./splay
    ii. ./avl

7.3 How to run test cases

a. Go to test folder
b. Run test file
    i. python test.py
c. If you want to test splay tree implementation, there are some test case for it. For that you have to run test.py . If implementation is correct then it shows following output.



```
vatsal@vatsal-Lenovo-G50-70:
Inorder TestCase Passed
Search TestCase Passed
Delete TestCase Passed
```

7.4 How to get graphical analysis

a. Go to graph folder
b. Run script file
    i. python script.py
c. If you want depth graphical analysis of splay tree and AVL tree, then run script.py .It will generate graph for each operation. Graph shows time taken for each operation.

7.5 Abstract definition of operations

a. **Insert :**
   Function Definition: *void insertnode(long long key)*
   Description: Insert function takes one integer argument and insert it into the tree. If key is already present then it will show appropriate message.

b. **Delete :**
   Function Definition: *bool deletenode(long long key)*
   Description: Delete function takes one integer argument and delete if from tree if it exists. Otherwise it shows error message.
   If element deleted successfully then it will return true, else return false.

c. **Search :**
   Function Definition: *bool search(long long key)*
   Description: Search function takes one integer argument and search that key in the tree and show appropriate message.
   If key found in the tree then it will return true, else return false.

d. **Tree Traversal :**
   Function Definition: void inorder(struct node *root)
   Description: Inorder function takes root of the tree as argument and print inorder traversal of the tree.

CHAPTER 8 REFERENCES

- https://en.wikipedia.org/wiki/Splay_tree
- https://www.cs.usfca.edu/~galles/visualization/SplayTree.html
- https://www.youtube.com/watch?v=IBY4NtxmGg8
- https://matplotlib.org/contents.html
- https://www.geeksforgeeks.org/splay-tree-set-1-insert/
- http://web.stanford.edu/class/archive/cs/cs166/cs166.1146/lectures/08/Small08.pdf