*International Institute of Information Technology Hyderabad*

# Study and implement van Emde Boas and splay tree and compare wrt RB tree

Shubham Agrawal (2019201085)
Shrayans Jain (2019201086)

# 1. RED BLACK TREE

## *1.1 Introduction:*

A red-black tree is a self-balancing binary search tree with one extra bit of storage per node: its color, which can be either *RED* or *BLACK* **.**

## *1.2 Properties*

1. Every node is either red or black.
2. The root is black.
3. Every NULL leaf is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

**Black Height** of a Red-Black Tree **:**

*Black height is number of black nodes on a path from root to a leaf. Leaf nodes are also counted black nodes.*

*1. A* Red-Black Tree of height h has **black-height >= h/2 .**
2. Every Red Black Tree with n nodes has **height <= $2Log_2(n+1)$ .**

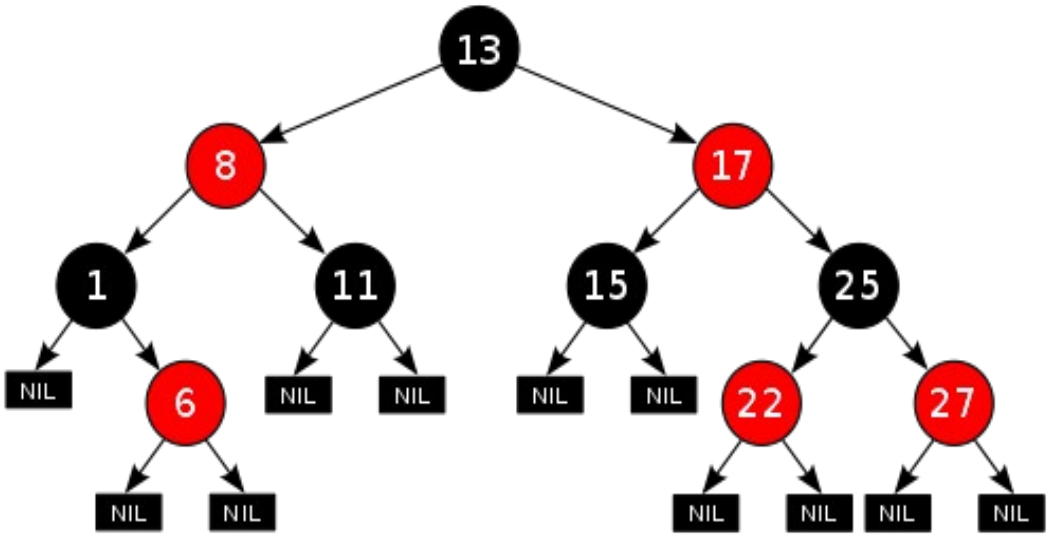



Figure 1 – Structure of Red Black tree

### *1.5 Applications*

- Java: java.util.TreeMap , java.util.TreeSet .
- C++ STL: map, multimap, multiset.
- Used in K-mean clustering algorithm for reducing time complexity.
- Linux kernel: completely fair scheduler, linux/rbtree.h

# 2. SPLAY TREE

## 2.1 Introduction:

Splay tree is also Self-Balancing like AVL and Red-Black Trees.

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 (In a typical application, 80% of the access are to 20% of the items). Imagine a situation where we have millions or billions of keys and only few of them are accessed frequently, which is very likely in many practical applications.

## *2.2 Set Operations*

The most important tree operation is **splay(x)**, which moves an element x to the root of the tree. In case x is not present in the tree, the last element on the search path for x is moved instead.

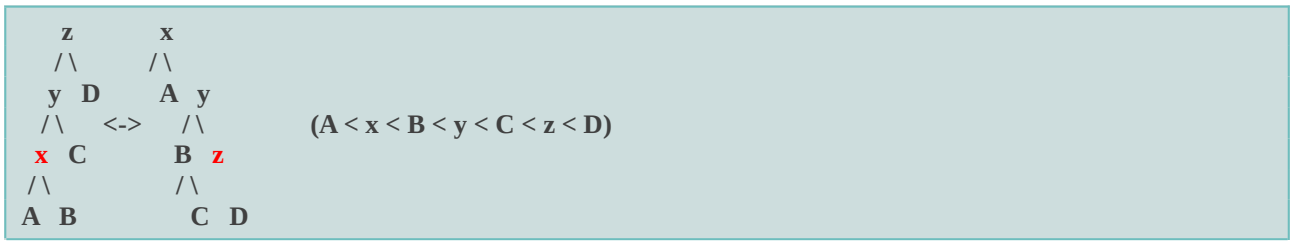
The run time for a splay(x) operation is proportional to the length of the search path for x: While searching for xwe traverse the search path top-down. Let y be the last node on that path. In a second step, we move y along that path by applying rotations as described below.

### Rotation 1: Simple rotation (Zig and Zag)

The simple tree rotation used in AVL tree is also applied at the root of the splay tree, moving the splayed node **x** up to become the new tree root. Here we have **A < x < B < y < C**, and the splayed node is either **x** or **y** depending on which direction the rotation is. It is highlighted in red.

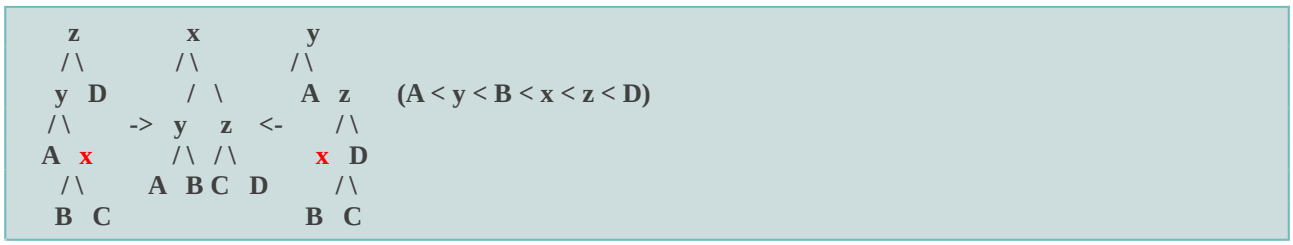

### Rotation 2: Zig-Zig and Zag-Zag

Lower down in the tree rotations are performed in pairs so that nodes on the path from the splayed node to the root move closer to the root on average. In the "zig-zig" case, the splayed node is the left child of a left child or the right child of a right child ("zag-zag").



**Rotation 3: Zig-Zag**

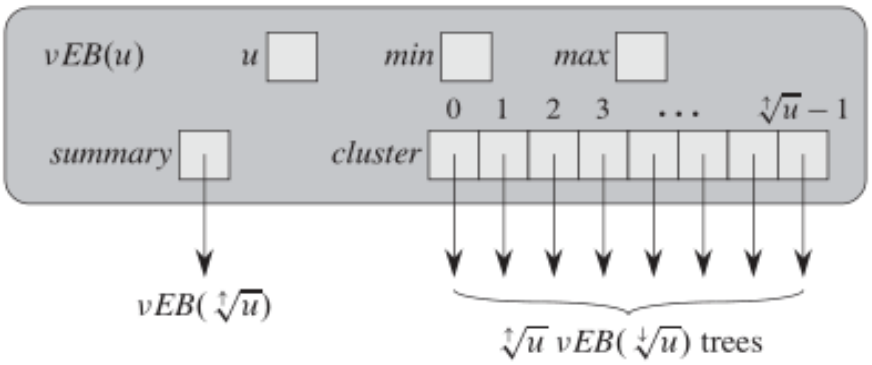
In the "zig-zag" case, the splayed node is the left child of a right child or vice-versa. The rotations produce a subtree whose height is less than that of the original tree. Thus, this rotation improves the balance of the tree. In each of the two cases shown,**y** is the splayed node:



# 3. VAN EMDE BOAS TREE

### *3.1 Introduction*

The van Emde Boas tree modifies proto-vEB structure. We denote a vEB tree with a universe size of u as vEB(u) and, unless u is equal the base size of 2, the attribute summary points to vEB(√u) tree and the array cluster[0,1....√u-1 ] points to √u vEB(√u). The following figure shows the basic node structure of vEB tree.




The attributes of node is as follows:

**clusters**: There are √ u clusters C0, . . . , C√ u−1 , where each cluster Cc is a van-Emde-Boas node and stores a set S 0 c ⊆ [ √ u] over the reduced universe [ √ u].

**summary**: The summary stores the clusters Cc with at least one element by the identifier c, i.e. c ∈ summary iff |Cc| ≥ 1. The summary itself is a van-Emde-Boas node over the reduced universe [ √ u].

**min/max**: min and max are respectively the minimum and maximum element of the set S. They will never be stored in any of the clusters below the node u and are the base case for the recursion. As long as |S| ≤ 2 the node will not initialize the clusters or the summary.

### *3.2 Opeartions*

**min(x) / max(x):** We store the minimum and maximum in the attributes min and max, two of operations are one-liners, taking constant time.

- *Time Complexity* : O(1)

**search(x):** The procedure search(x) has a recursive call. We check directly whether x equals the minimum or maximum element. Since a vEB tree doesn't store bits as a proto-vEB structure does, we design search(x) to return *true* or *false* rather than (1,0).

- *Time Complexity* : O(loglogu)

**successor(x) and predecessor(x):** With help of summary and max, min we have to make only one recursive call for searching successor or predecessor for any key. This reduces the time complexitiy from O(logu) to O(loglogu). If the element doesn't have successor or predecessor than it returns -1.

- *Time Complexity* : O(loglogu)

**insert(x):** When we insert an element, either the cluster that it goes into already has another element or it does not. If the cluster already has another element, then the cluster number is already in the summary, and so we do not need to make that recursive call. If the cluster does not already have another element, then the element being inserted becomes the only element in the cluster, and we do not need to recurse to insert an element into an empty vEB tree

- *Time Complexity* : O(loglogu)

**delete(x):** With help of min, max and summary we search the key to be deleted in only one recursive call. While recursion we handle different cases.

- *Time Complexity* : O(loglogu)

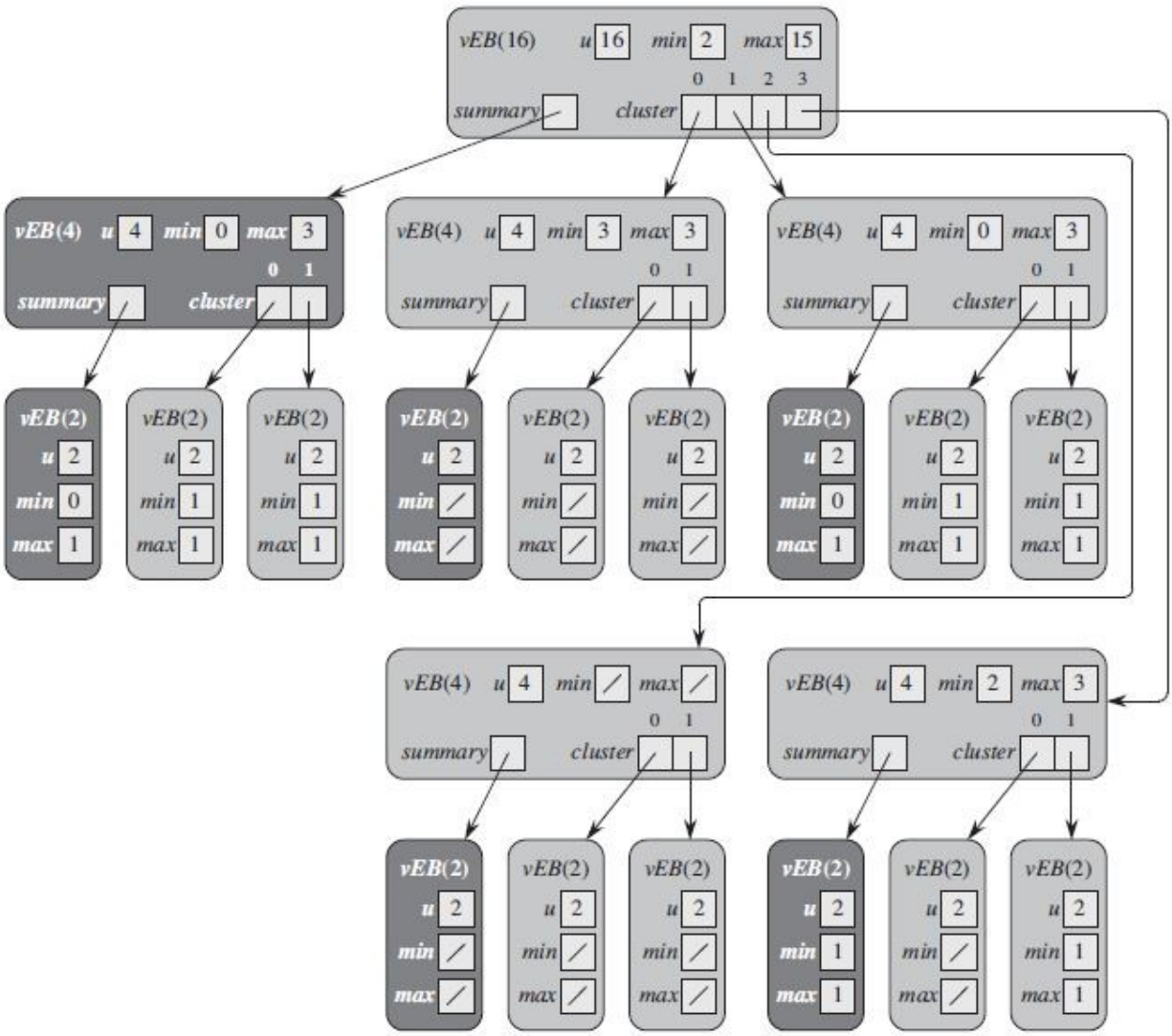



*Figure 3: shows vEB tree of size 16 containing elements [2,3,4,5,7,14,15]*

# *4. Comparison between Van Emde Boas Tree, Red-Black Tree and Splay Tree*

*4.1 Time Complexity Comparison table of Red-Black Tree, Splay Tree, Van Emde Boas Tree*

| **Trees** | ***Red*-Black Tree** | | ***Splay Tree*** | | ***Van Emde Boas Tree*** | |
|---|---|---|---|---|---|---|
| ***Operations*** | *Avg* | *Worst* | *Avg* | *Worst* | *Avg* | *Worst* |
| ***insert(v,x)*** | O(log n) | O(log n) | O(log n) | O(n) | O(log log u) | O(log log u) |
| ***search(v,x)*** | O(log n) | O(log n) | O(log n) | O(n) | O(log log u) | O(log log u) |
| ***maximum(v)*** | O(log n) | O(log n) | O(log n) | O(n) | O(1) | O(1) |
| ***minimum(v)*** | O(log n) | O(log n) | O(log n) | O(n) | O(1) | O(1) |
| ***successor(v,x)*** | O(log n) | O(log n) | O(log n) | O(n) | O(log log u) | O(log log u) |
| ***predecessor(v,x)*** | O(log n) | O(log n) | O(log n) | O(n) | O(log log u) | O(log log u) |
| ***delete(v,x)*** | O(log n) | O(log n) | O(log n) | O(n) | O(log log u) | O(log log u) |

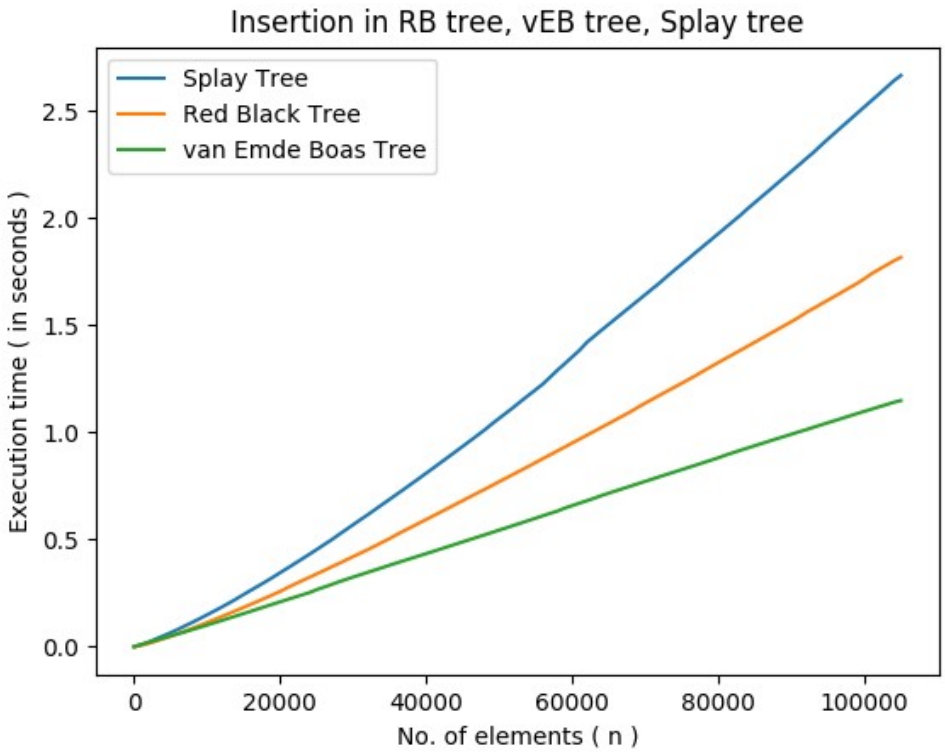



*Figure 4.1: Shows the graph of insertion in all three trees*

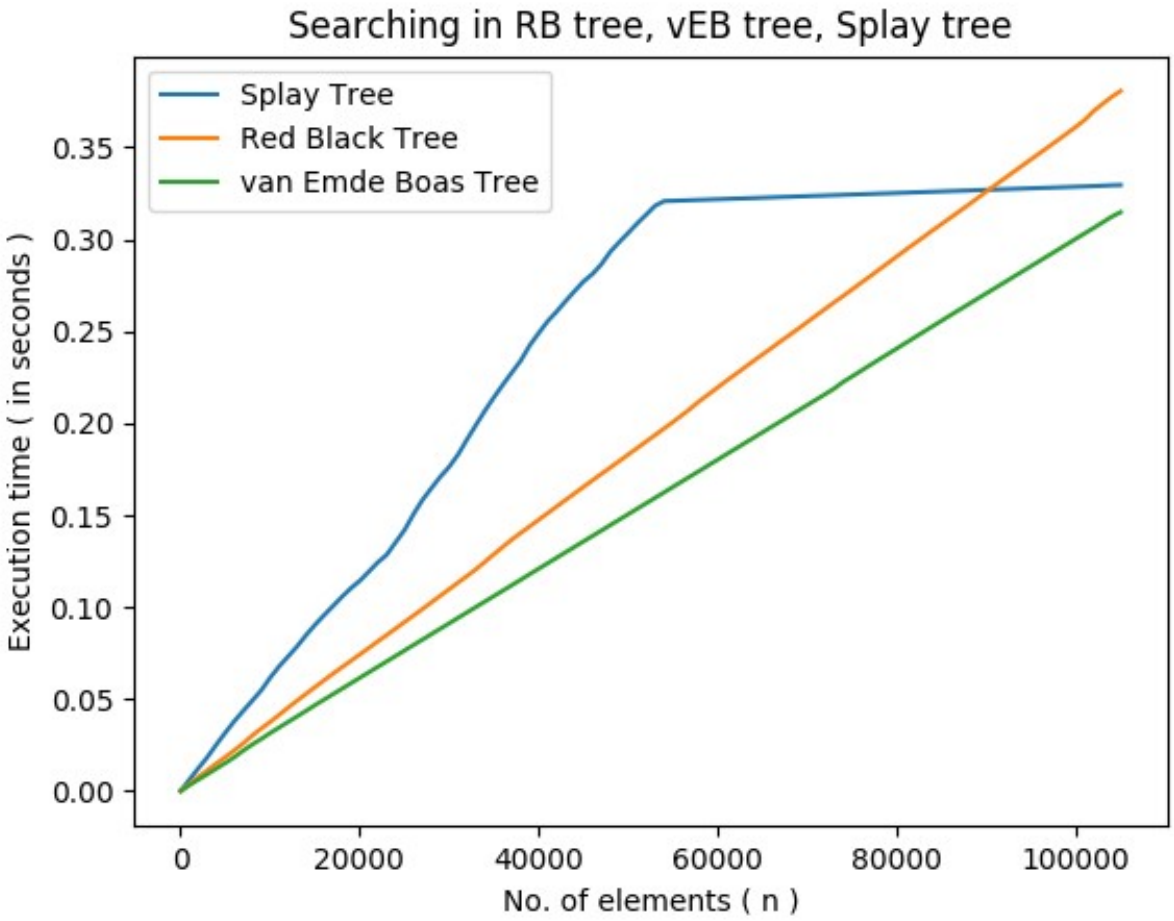



*Figure 4.2: Shows the graph of serching in all three trees*

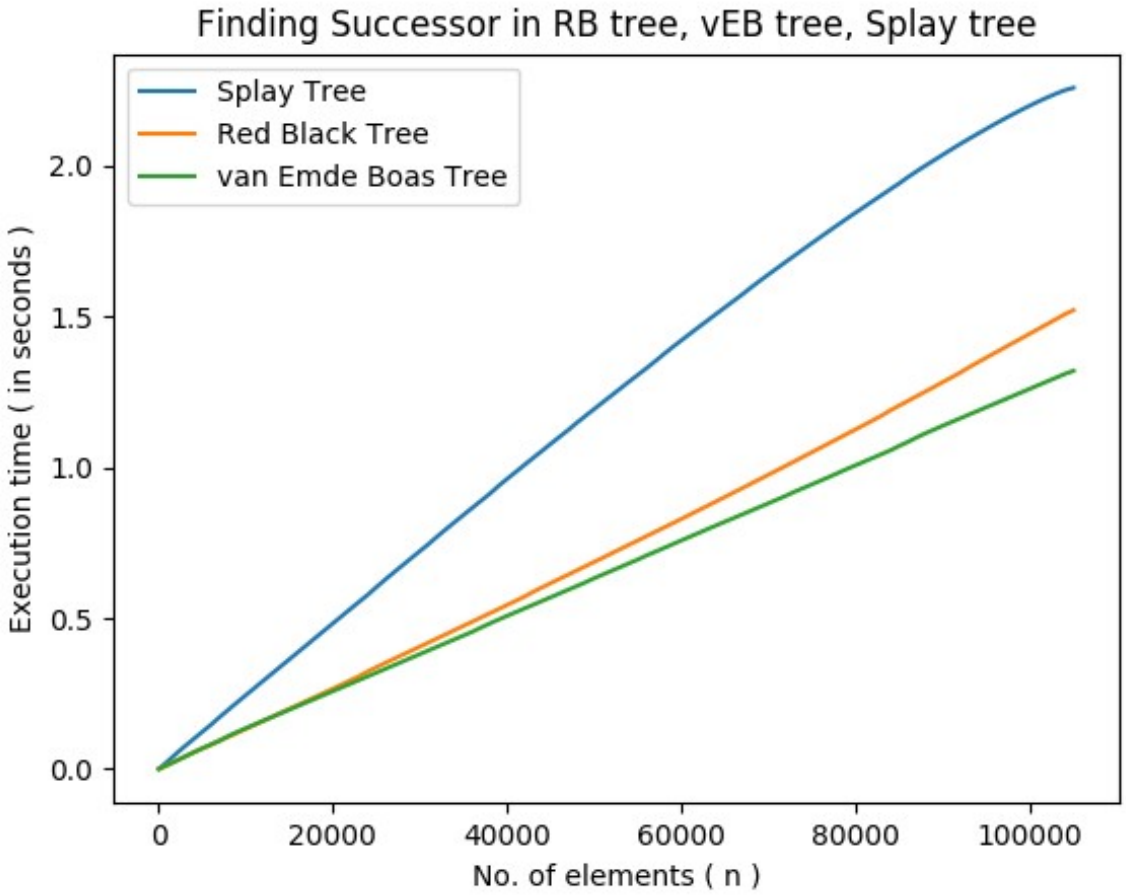



*Figure 4.3: Shows the graph of successor operation in all three trees*

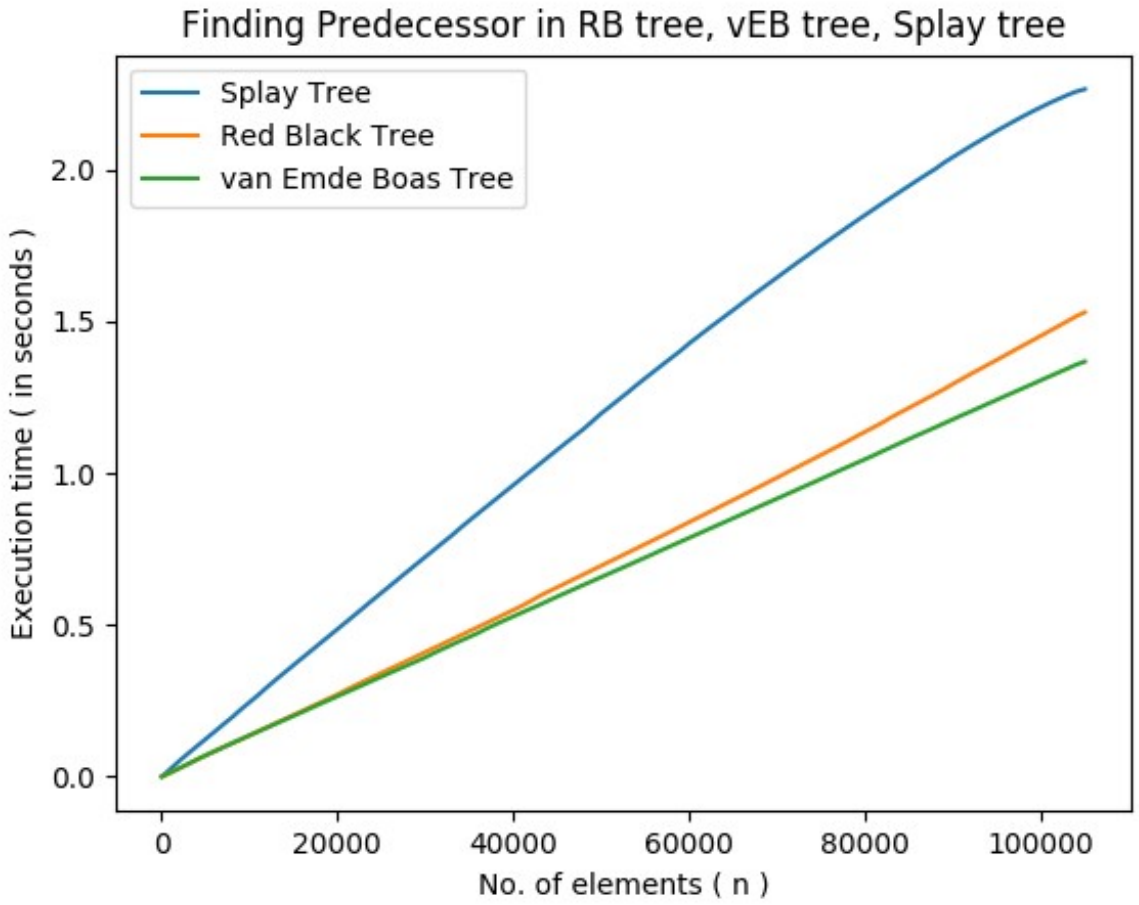



*Figure 4.4: Shows the graph of precedessor operation in all three trees*

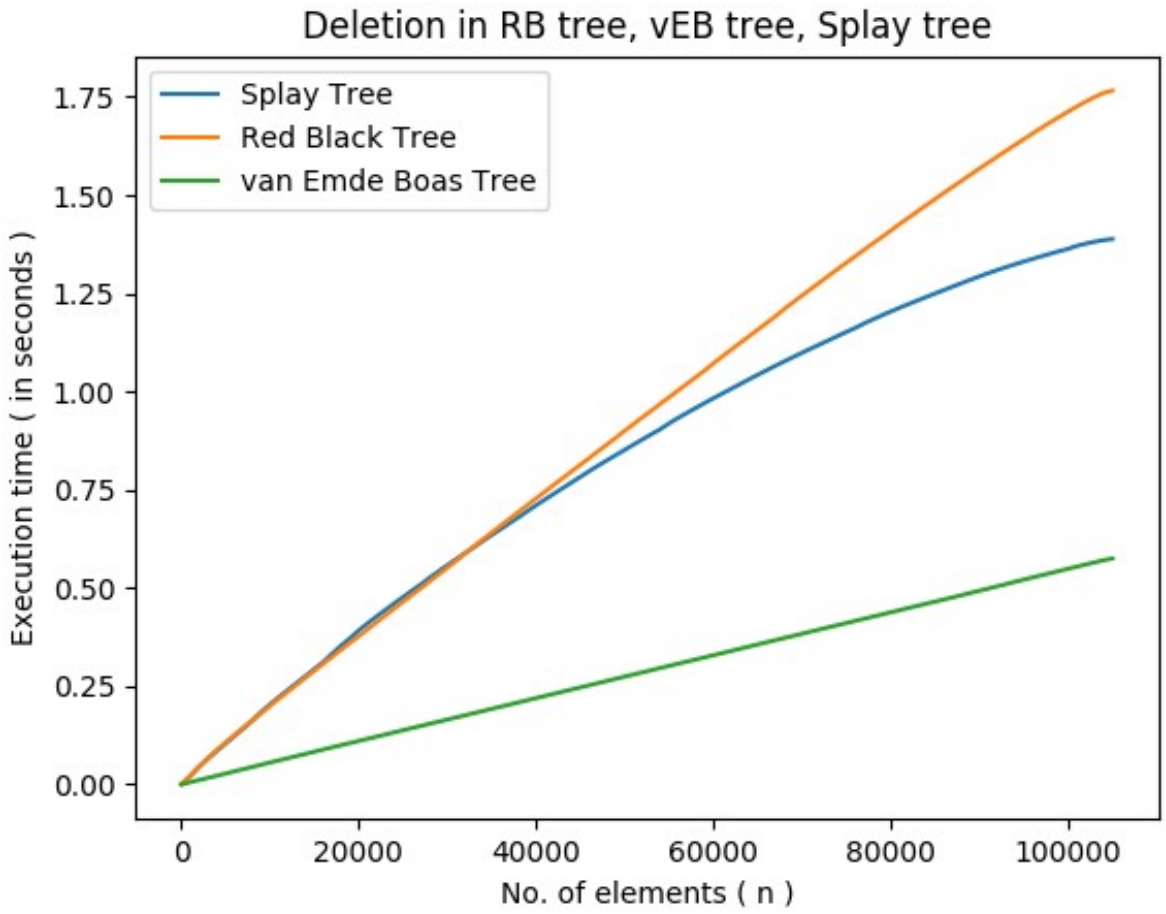



*Figure 4.5: Shows the graph of deletion operation in all three trees*

# *5. Conclusion*

- If we know the range of the data set in advance, then we can go for van emde boas tree.
- If we don’t know range in advance, then red black is the best choice.
- If we want to improve the recent search time, then we should go for splay tree.

# *6.* References


- https://en.wikipedia.org/wiki/Splay_tree
- https://www.geeksforgeeks.org/splay-tree-set-1-insert/
- http://www-di.inf.puc-rio.br/~laber/vanEmdeBoas.pdf
- http://web.stanford.edu/class/cs166/lectures/14/Small14.pdf
- https://en.wikipedia.org/wiki/Red%E2%80%93black_tree
- https://www.youtube.com/watch?v=UaLIHuR1t8Q