**SPLAY TREE**

**ABSTRACT**

**Splay trees** are a self adjusting form of binary search tree that supports access operations in Ο(log n) amortized time. Splay trees also have several amazing distribution sensitive properties, the strongest two of which are the working set theorem and the dynamic finger theorem. However, these two theorems are shown to poorly bound the performance of splay trees on some simple access sequences. The unified conjecture is presented, which subsumes the working set theorem and dynamic finger theorem, and accurately bounds the performance of splay trees over some classes of sequences where the existing theorems' bounds are not tight. While the unified conjecture for splay trees is unproven, a new data structure, The unified structure, is presented where the unified conjecture does hold. This structure also has a worst case of Ο(log n) per operation, in contrast to the Ο(n) worst case runtime of splay trees. A second data structure, the working set structure, is introduced. The working set structure has the same performance attributed to splay trees through the working set theorem, except the runtime is worst case per operation rather than amortized.

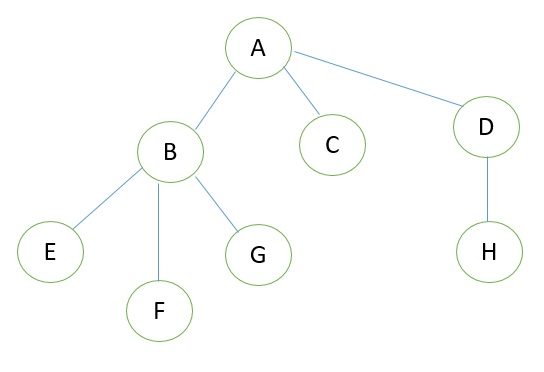
**INTRODUCTION**

## **Tree**

**In a data structure ,a tree is a collection of elements called nodes. Each node contains some value or element**. We will use the term node, rather than vertex with binary tree. Node is the main component of any tree structure. It stores the actual data along with links to other nodes.

**TREE TREMINOLOGIES**

**Consider the following tree**



**1. Root**

This is the unique node in the tree in which further subtrees were attached. **A** root node of a tree has its child. Left child and right child are the two childes a root node can have in a tree. In the given figure **A** is the root node.

**2. Degree of Node**

The total number of subtree attached to that node is called the degree of the node. **For node A the degree is 3**, **for node E the degree is 0**.

**3. Leaf Nodes**

These are the terminals nodes of the tree. The nodes which have degree 0 are called as leaf nodes of the tree. These nodes are always present at the end of the tree. Here in our example **E**, **F**, **C**, **G**, **H** are the leaf nodes of the tree.

**4. Internal Nodes**

The nodes in the tree which are other than leaf nodes and the root node are called as internal nodes. These nodes are present in between root node and leaf nodes in the tree that’s why these nodes are called as internal node. Here, **B and D are internal nodes**.

**5. Parent Node**

The node which is having further sub branches is called the parent node of those sub branches. In the above figure node **B** is the parent node of **E**, **F**, and **G** node and **E**, **F**, and **G** are called children of **B**.

**6. Predecessor**

While displaying the tree, if some particular nodes previous to some other nodes than that node is called the predecessor of the other node. In our example, **the node E is predecessor of node B**.

**7. Successor**

The node which occurs next to some other node is called successor of that node. In our example, **B is successor of F and G**.

**8. Level of tree**

The root node is always considering at level zero, and then its adjacent children are supposed to be at level 1 and so on. To understand the concept of level, see the above figure, the **node A** is at level 0, the **node B**, **C**, **D** are at level 1, the **nodes E**, **F**, **G**, **H** are at level 2.

**9. Height of the tree**

The maximum level is the height of the tree. Here **the height of the tree is 3**. The other terminology used for the height of the tree is depth of the tree.

**10. Forest**

A tree may be defined as a forest in which only a single node (root) has no predecessor Any forest is consist of collection of trees.

**11. Degree of tree**

The maximum degree of the node in the tree is called the degree of the tree. Here, the **degree of tree is 3**.

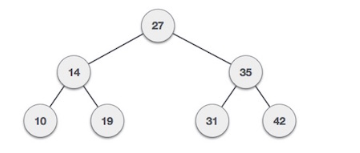
**DIFFERENT TYPES OF TREES IN DATA STRUCTURE**

* **General** **Tree**. If no constraint is placed on the hierarchy of the tree, a tree is called a general tree.
* **Binary** **Tree**. The binary tree is the kind of tree in which most two children can be found for each parent.
* **Binary Search** **Tree**.
* **Splay Tree** ,It is a example of **binary search tree**.
* **AVL Tree**. ...
* **Red-Black** **Tree**. ...
* **N-ary** **Tree**.

**BINARY SEARCH TREE**

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST −

****

**A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −**

* The left sub-tree of a node has a key less than or equal to its parent node's key.
* The right sub-tree of a node has a key greater than to its parent node's key.

**SPLAY TREE**

A splay tree is a self-balancing binary search tree with the additional property that recently accessed elements are quick to access again***.***

***Splaying*** is the basic operation for the splay trees which rearranges the tree so that element is placed at the root of the tree. The performance of the splay trees depends on the self balancing and self optimizing. The nodes of the tree are moved closer to the root so that they can be accessed quickly. In real time splay trees are used for implementing caches. The worst case with this splay tree algorithm is that this will sequentially access all the elements of the tree which makes the tree unbalanced.

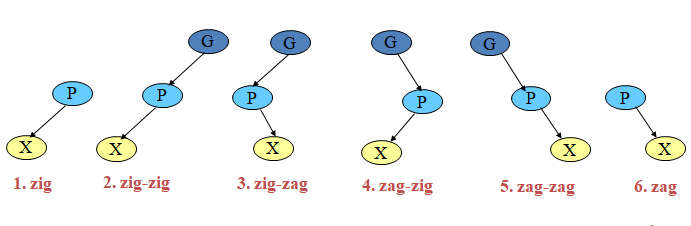
**HISTORY**

Splay trees are invented by Daniel Dominic Sleator and Robert Endre Tarjan in 1985. Splay trees are self adjusting binary search tree , these trees perform better than other search trees. It performs basic operations such as insertion, search and deletion in amortized time.

**SPLAY TREE TERMINOLOGY**

* Let X be a non-root node, i.e., has at least 1 ancestor.
* Let P be its parent node.
* Let G be its grandparent node (if it exists)
* Consider a path from G to X:
  + Each time we go left, we say that we “zig”
  + Each time we go right, we say that we “zag”

There are 6 possible cases:



**SPLAY TREE OPERATION**

In a splay tree, every operation is performed at the root of the tree. All the operations in splay tree are involved with a common operation called **"Splaying"**.

In a splay tree, splaying an element rearranges all the elements in the tree so that splayed element is placed at the root of the tree.  
  
By splaying elements we bring more frequently used elements closer to the root of the tree so that any operation on those elements is performed quickly. That means the splaying operation automatically brings more frequently used elements closer to the root of the tree.  
  
Every operation on splay tree performs the splaying operation. For example, the insertion operation first inserts the new element using the binary search tree insertion process, then the newly inserted element is splayed so that it is placed at the root of the tree. The search operation in a splay tree is nothing but searching the element using binary search process and then splaying that searched element so that it is placed at the root of the tree.

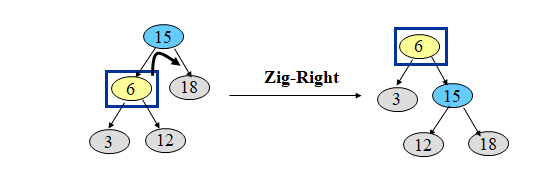
**DIFFERENT TYPES OF OPERATIONS OF SPLAY TREE**

Consider above binary search tree,

* + Single Rotations (X has a P but no G)
    - zig
    - zag
  + Double Rotations (X has both a P and a G)
    - zig-zig
    - zig-zag
    - zag-zig
    - zag-zag

**ZIG OPERATION**

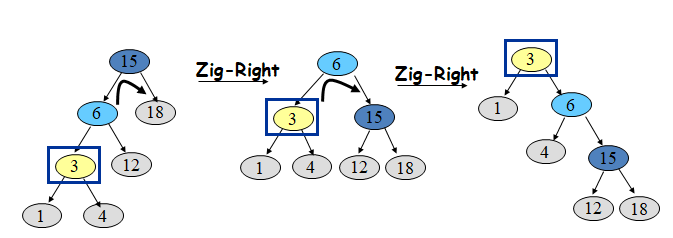
* “Zig” is just a single rotation, as in an AVL tree
* Suppose 6 was the node that was accessed (e.g. using Search)

****

* “Zig-Right” moves 6 to the root.
* Can access 6 faster next time: O(1)
* Notice that this is simply a right rotation in AVL tree terminology.

**ZIG- ZIG OPERATION**

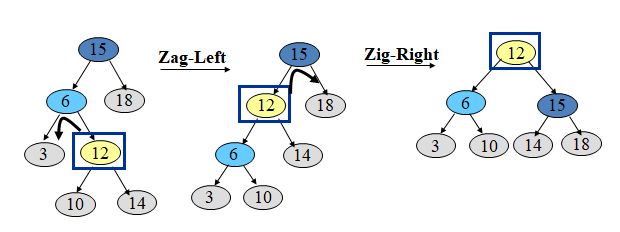
* “Zig-Zig” consists of two single rotations of the same type
* Suppose 3 was the node that was accessed (e.g., using Search)



* Due to “zig-zig” splaying, 3 has bubbled to the top!
* *Note: Parent-Grandparent is rotated first.*

**ZIG-ZAG OPERATION**

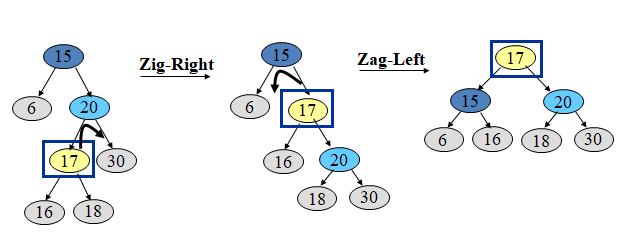
* “Zig-Zag” consists of two rotations of the opposite type
* Suppose 12 was the node that was accessed (e.g., using Search)



* Due to “zig-zag” splaying, 12 has bubbled to the top!
* Notice that this is simply an LR imbalance correction in AVL tree terminology (first a left rotation, then a right rotation)

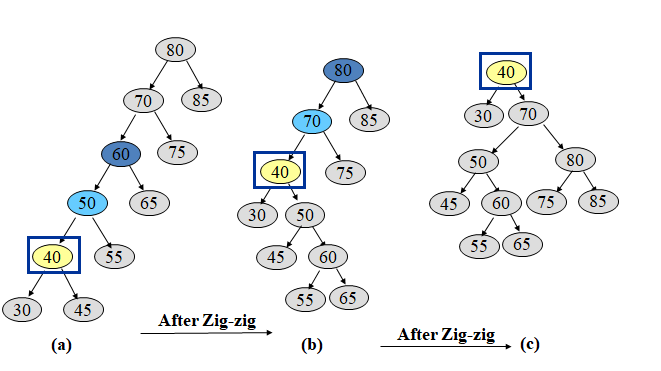
**ZAG-ZIG OPERATION**

* “Zag-Zig” consists of two rotations of the opposite type
* Suppose 17 was the node that was accessed (e.g., using Search)

****

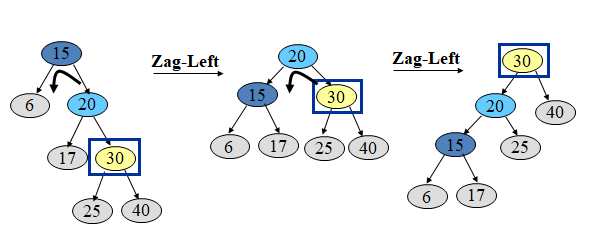
* Due to “zag-zig” splaying, 17 has bubbled to the top!
* Notice that this is simply an RL imbalance correction in AVL tree terminology (first a right rotation, then a left rotation)

**Splay Trees: Example – 40 is accessed**

****

**ZAG-ZAG OPERATION**

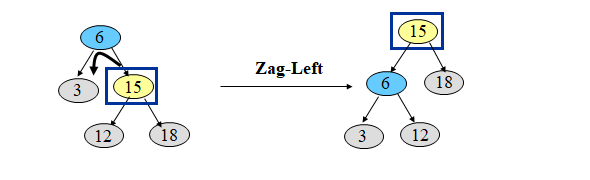
* “Zag-Zag” consists of two single rotations of the same type
* Suppose 30 was the node that was accessed (e.g., using Search)



* Due to “zag-zag” splaying, 30 has bubbled to the top!
* Note: Parent-Grandparent is rotated first.

**ZAG OPERATION**

* “Zag” is just a single rotation, as in an AVL tree
* Suppose 15 was the node that was accessed (e.g., using Search)



* “Zag-Left”moves 15 to the root.
* Can access 15 faster next time: O(1)
* Notice that this is simply a left rotation in AVL tree terminology

**BALANCING FACTOR**

**What is balancing factor?**

The *balance factor* of a node is defined to be Height(RightSubtree) Height(LeftSubtree).  Thus, in an AVL tree, the balance factor of each node will be in {-1, 0, +1}.

Insertion into an AVL tree may change change the balance factors of some nodes on the path from the inserted node to the root.  Any balance factor will change by at most 1 and the resulting value might become illegal (i.e., -2 or +2).  After determining the deepest node that is "illegal", one single or one double rotation suffices to fix up the entire tree.

**What is the balancing factor of Splay tree?**

We can show below three characteristics of binary search tree.

y rotate (xy) R x

/ \ ====> / \

x 3 1 y //R-Denotes to RIGHT

/ \ rotate (xy) L / \ //L-Denotes to LEFT

1 2 <==== 2 3

DOUBLE ROTATE (xyz) L

x 1. rotate (yz) R

\ 2. rotate (xz) L z

y =======> / \

/ x y

z DOUBLE ROTATE (xzy) R

1. rotate (xz) R

2. rotate (yz) L

<=======

A splay tree is a binary search tree with no explicit balance condition, in which a special operation called a *splay* is done after each search operation.  Splaying at node *x* causes node *x* to become the root of the binary search tree through a specific series of *rotations* as follows.

Three cases:

1. *x* has no grandparent (*zig*)
   * If *x* is left child of root *y*, then rotate (*xy*)R.
   * Else if *x* is right child of root *y*, then rotate (*yx*)L.
2. *x* is LL or RR grandchild (*zig-zig*)
   * If *x* is left child of *y*, and *y* is left child of *z*,  
     then rotate at grandfather (*yz*)R and then rotate at father (*xy*)R.
   * Else if *x* is right child of *y*, and *y* is right child of *z*,  
     then rotate at grandfather (*yz*)L and then rotate at father (*xy*)L.

If *x* has not become the root, then continue splaying at *x*.

1. *x* is LR or RL grandchild (*zig-zag*)
   * If *x* is right child of *y*, and *y* is left child of *z*,  
     then rotate at father (*yx*)L and then rotate at grandfather (*xz*)R.
   * Else if *x* is left child of *y*, and *y* is right child of *z*,  
     then rotate at father (*yx*)R and then rotate at grandfather (*xz*)L.

If *x* has not become the root, then continue splaying at *x*.

**APPLICATIONS**

The real time applications of the splay trees are

**•** It is used to implement caches. Cache keeps track of the contents of memory locations that were recently requested by processor. It can be made to deliver requests much faster than main memory. Simiarly, splay trees uses this concept of accessing the elements quickly that which were recently accessed. Splay trees are used to implement Cache algorithms.

**•** it has the ability of not to store any data, which results in minimization of memory requirements.

**•** It can also be used for data compression, e.g. dynamic huffman coding.

The main idealogy behind selecting the splay tree is to reduce the consumption of time while reaccessing of the elements and to improve the performance and also to reduce the storage space.

**ADVANTAGES OF SPLAY TREE**

Good performance for a splay tree depends on the fact that it is self -balancing, and indeed self-optimizing, in that frequently accessed nodes will move nearer to the root where they can be accessed more quickly. This is an advantage for nearly all practical applications, and is particularly useful for implementing caches and garbage collection algorithms.

Advantages include:

* Simple implementation—simpler than other self-balancing binary search trees, such as red-black trees or AVL trees.
* Comparable performance -- average-case performance is as efficient as other trees.
* Small memory footprint—splay trees do not need to store any bookkeeping data.
* Possibility of creating a persistent data structure version of splay trees —which allows access to both the previous and new versions after an update. This can be useful in functional programming, and requires amortized O(log *n*) space per update.
* Working well with nodes containing identical keys—contrary to other types of self-balancing trees. Even with identical keys, performance remains amortized O(log *n*). All tree operations preserve the order of the identical nodes within the tree, which is a property similar to stable sorting algorithms. A carefully designed find operation can return the left most or right most node of a given key.

**DISADVANTAGES OF SPLAY TREE**

* Poor performance on uniform access (with workaround) -- a splay tree's performance will be considerably (although not asymptotically) worse than a somewhat balanced simple binary search tree for uniform access.

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 an O(log *n*) operation). Reaccessing the first item triggers an operation that takes O(*n*) Operations to rebalance the tree before returning the first item. This is a significant delay for that final operation, although the amortized performance over the entire sequence is actually O(log *n*). However, recent research shows that randomly rebalancing the tree can avoid this unbalancing effect and give similar performance to the other self-balancing algorithms.

**CONCLUSION**

* A balanced binary search tree.
* Doesn’t need any extra information to be stored in the node, ie color, level, etc.
* Balanced in an amortized sense.
* Running time is O(mlog n) for m operations
* Can be adapted to the ways in which items are being accessed in a dictionary to achieve faster running times for the frequently accessed items. (O(1), AVL is about O(log n), etc.)

**REFERENCES**

* Albers, Susanne; Karpinski, Marek (28 February 2002). [*"Randomized Splay Trees: Theoretical and Experimental Results"*](http://www14.in.tum.de/personen/albers/papers/ipl02.pdf) *(PDF)*. [*Information Processing Letters*](https://en.wikipedia.org/wiki/Information_Processing_Letters). **81** (4): 213–221. [*doi*](https://en.wikipedia.org/wiki/Digital_object_identifier):[*10.1016/s0020-0190(01)00230-7*](https://doi.org/10.1016%2Fs0020-0190%2801%2900230-7).
* Allen, Brian; Munro, Ian (October 1978). "Self-organizing search trees". [*Journal of the ACM*](https://en.wikipedia.org/wiki/Journal_of_the_ACM). **25** (4): 526–535. [*doi*](https://en.wikipedia.org/wiki/Digital_object_identifier):[*10.1145/322092.322094*](https://doi.org/10.1145%2F322092.322094).
* Brinkmann, Gunnar; Degraer, Jan; De Loof, Karel (January 2009). [*"Rehabilitation of an unloved child: semi-splaying"*](http://caagt.ugent.be/preprints/splay_spe.pdf) *(PDF)*. Software—Practice and Experience. **39** (1): 33–45. [*CiteSeerX*](https://en.wikipedia.org/wiki/CiteSeerX) [*10.1.1.84.790*](https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.84.790). [*doi*](https://en.wikipedia.org/wiki/Digital_object_identifier):[*10.1002/spe.v39:1*](https://doi.org/10.1002%2Fspe.v39%3A1). The results show that semi-splaying, which was introduced in the same paper as splaying, performs better than splaying under almost all possible conditions. This makes semi-splaying a good alternative for all applications where normally splaying would be applied. The reason why splaying became so prominent while semi-splaying is relatively unknown and much less studied is hard to understand.
* Cole, Richard; Mishra, Bud; Schmidt, Jeanette; Siegel, Alan (January 2000). "On the Dynamic Finger Conjecture for Splay Trees. Part I: Splay Sorting log n-Block Sequences". [*SIAM Journal on Computing*](https://en.wikipedia.org/wiki/SIAM_Journal_on_Computing). **30** (1): 1–43. [*CiteSeerX*](https://en.wikipedia.org/wiki/CiteSeerX) [*10.1.1.36.4558*](https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.4558). [*doi*](https://en.wikipedia.org/wiki/Digital_object_identifier):[*10.1137/s0097539797326988*](https://doi.org/10.1137%2Fs0097539797326988).
* Cole, Richard (January 2000). "On the Dynamic Finger Conjecture for Splay Trees. Part II: The Proof". [*SIAM Journal on Computing*](https://en.wikipedia.org/wiki/SIAM_Journal_on_Computing). **30** (1): 44–85. [*CiteSeerX*](https://en.wikipedia.org/wiki/CiteSeerX) [*10.1.1.36.2713*](https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.2713).