Splay Trees (with implementations in C++, Java, and Python)

Introduction

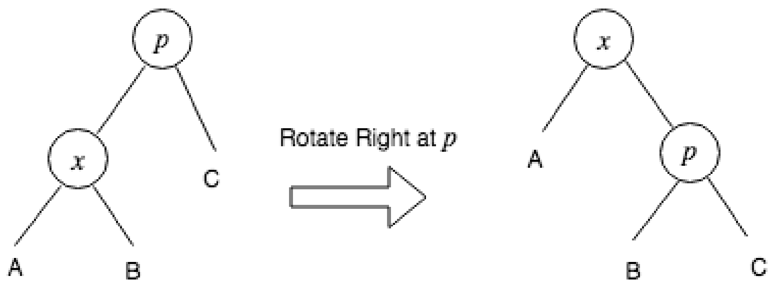
A Splay tree is a self-adjusting binary search tree invented by Sleator and Tarjan. Unlike an AVL tree (or a Red-Black tree), the structure of the splay tree changes even after the search operation. Every time we search an item x or insert x, it moves x to the root of the tree so that the next access of x is quick. The goal of the splay tree is not to make every operation fast rather make the sequence of operations fast. The individual operation in the splay tree can, sometimes, take $O(n)$ time making the worst case running time linear. The sequence of operations, however, take $O(\log n)$ amortized (https://algorithmtutor.com/Analysis-of-Algorithm/Amortized-Analysis-of-Algorithms/) time per operation. In other words, the sequence of M operations takes $O(M \log n)$ time. Since the splay tree adjusts itself according to usage, it performs much more efficiently than other binary search trees if the usage pattern is skewed.

Unlike an AVL or a Red-Black tree where the tree must maintain their invariants all the time, the structure of the splay tree can be in any arbitrary state (although it should maintain the binary search tree invariants all the time) but during every operation, it restructures the tree to improve the efficiency of future (incoming) operations.

The splay tree moves a node x to the root of the tree by performing series of single and double tree rotations (https://algorithmtutor.com/Data-Structures/Tree/Self-balancing-Binary-Search-Trees/#Tree-rotation). Each double rotations moves x to its grandparent's place and every single rotation moves x to its parent's place. We perform these rotations until x reaches to the root of the tree. This process is called *splaying*. Besides moving x to the root, *splaying* also shortens the height of the tree which makes the tree more balanced. There are two types of single rotations and four types of double rotations. Each of them is explained in detail below.

Zig Rotation

Zig is a single rotation. We do zig rotation on node x if x is a left child and x does not have a grandparent (i.e. x's parent is a root node). To make the zig rotation, we rotate x's parent to the right. This is illustrated in figure 1.

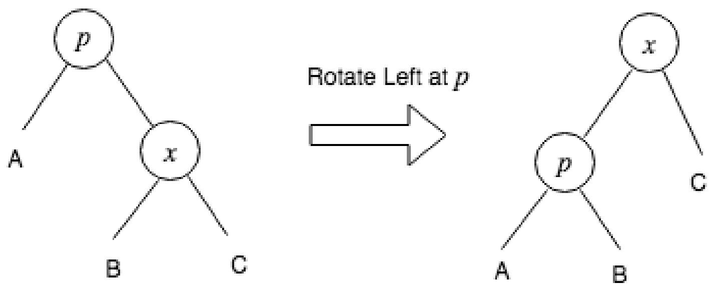


(data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAASwAAAEsCAMA

Figure 1: A zig rotation

Zag Rotation

Zag rotation is a mirror of zig rotation. We do zag rotation on node x if x is a right child and x does not have a grandparent. To make the zag rotation, we perform a left rotation at x's parent node. This is illustrated in figure 2.

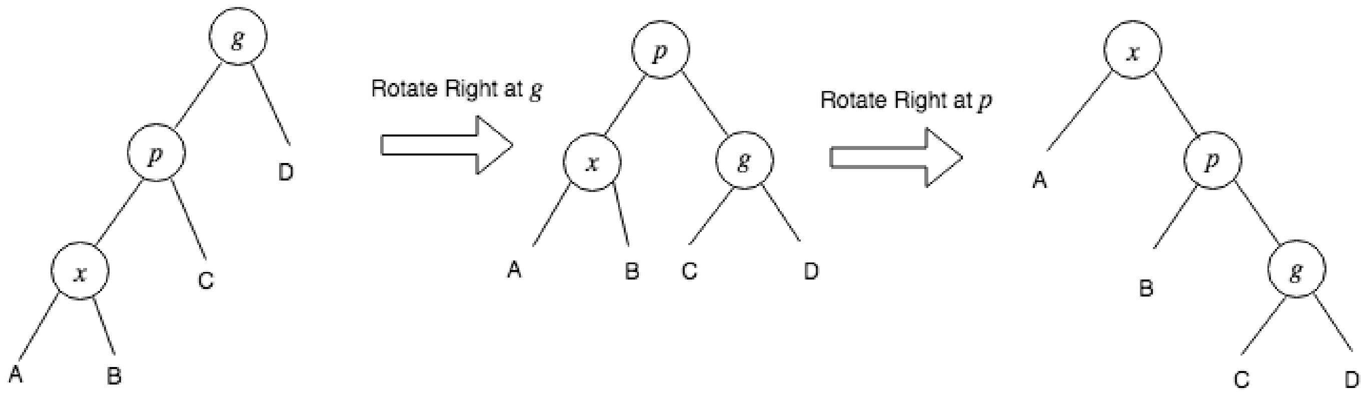


(data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAASwAAAEsCAMA

Figure 2: A zag rotation

Zig-Zig Rotation

Zig-Zig is a double rotation. We do a zig-zig rotation on x when x is a left child and x's parent node is also a left child. The zig-zig rotation is done by rotating x's grandparent node to the right followed by right rotation at x's parent node.

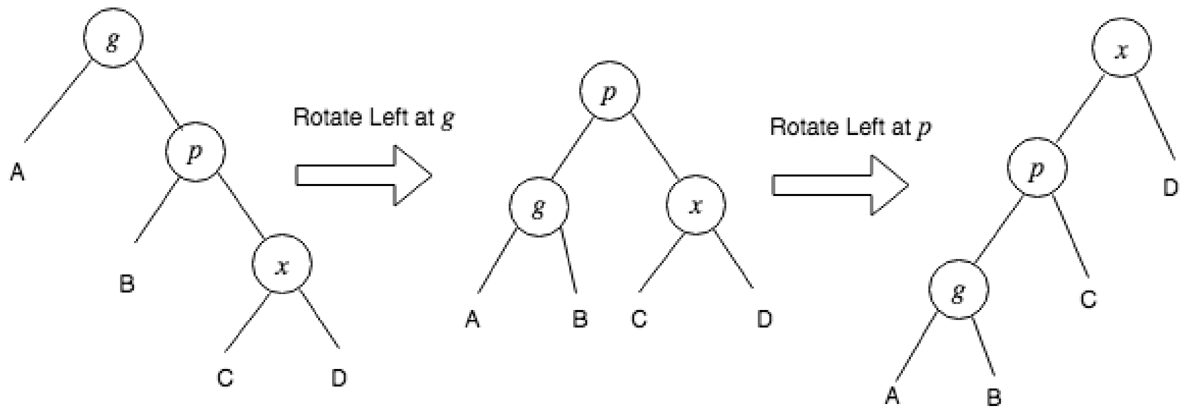


(data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAASwAAAEsCAMA

Figure 3: A zig-zig rotation

Zag-Zag Rotation

Zag-Zag rotation is a mirror of zig-zig rotation. We do zag-zag rotation on x if x is a right child and x's parent is also a right child. To make the zag-zag rotation, we first do a left rotation at x's grandparent and then do the left rotation at x's parent node. Figure 4 illustrates this.

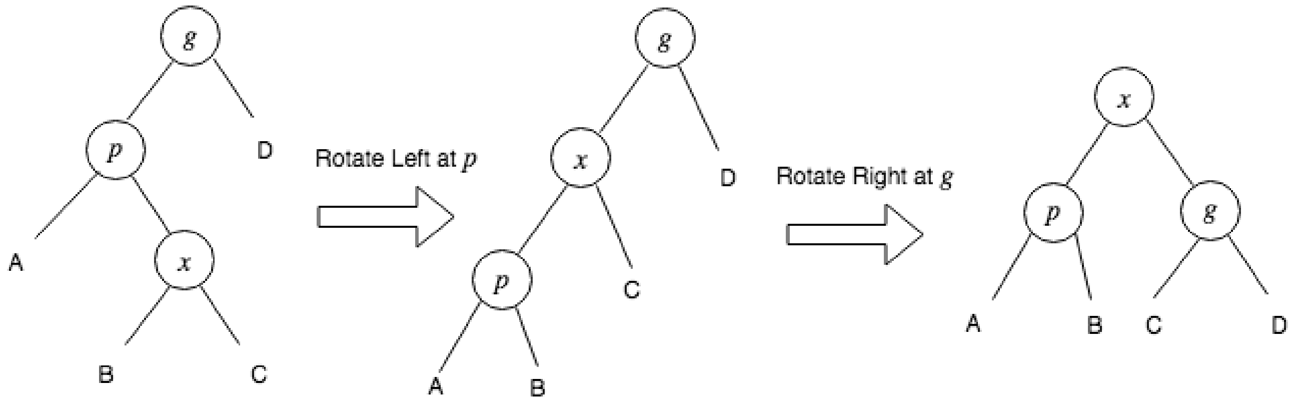


(data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAASwAAAEsCAMA

Figure 4: A zag-zag rotation

Zig-Zag Rotation

Zig-zag rotation is also a double rotation. We perform zig-zag rotation on x when x is a right child and x's parent is a left child. Zig-zag rotation is done by doing a left rotation at x's parent node followed by right rotating x grandparent (new parent) node. This is illustrated in figure 5.

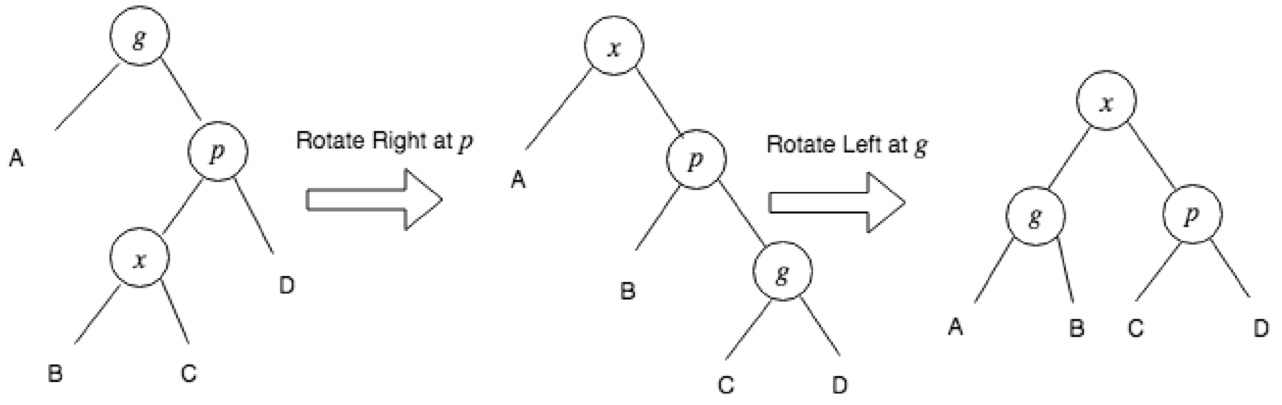


(data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAASwAAAEsCAMA

Figure 5: A zig-zag rotation

Zag-Zig Rotation

The last rotation is the zag-zig rotation. It is a mirror of zig-zag rotation. To do zag-zig rotation on node x, we do the right rotation at x's parent node and left rotation at x grandparent (new parent) node. Figure 6 illustrates this.



(data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAASwAAAEsCAMA

Figure 6: A zag-zig rotation

Operations

Now we know how to make several rotations on a node. We are ready to explore the operations on the splay tree.

Splaying

The splaying operation moves a node x to the root of the tree using the series of rotations. The pseudo code for splaying a node is given below.

```
SPLAY(x)
  while x.parent ≠ NIL
    if x.parent.parent == NIL
      if x == x.parent.left
        // zig rotation
        RIGHT-ROTATE(x.parent)
      else
        // zag rotation
        LEFT-ROTATE(x.parent)
    else if x==x.parent.left and x.parent == x.parent.parent.left
        // zig-zig rotation
        RIGHT-ROTATE(x.parent.parent)
        RIGHT-ROTATE(x.parent)
    else if x==x.parent.right and x.parent == x.parent.parent.right
        // zag-zag rotation
        LEFT-ROTATE(x.parent.parent)
        LEFT-ROTATE(x.parent)
    else if x==x.parent.right and x.parent == x.parent.parent.left
        // zig-zag rotation
        LEFT-ROTATE(x.parent)
        RIGHT-ROTATE(x.parent)
    else
        // zag-zig rotation
        RIGHT-ROTATE(x.parent)
        LEFT-ROTATE(x.parent)
```

Join

Join operations joins two trees S and T such that all the items in S are smaller than the items in T. This is a two-step process.

1. Splay the largest node in S. This moves the largest node to the root node having a NIL right child.
2. Set the right child of the new root of S to T.

The pseudo code for join operation is given below.

```
JOIN(S, T)
    if S == NIL
        return T
    if T == NIL
        return S
    x = TREE-MAXIMUM(S)
    SPLAY(x)
    x.right = t
    t.parent = x
    return x
```

Split

The split operation at node x divide the tree into two trees T and S such that all the elements in T are smaller than or equal to x and all the elements in T are greater than x. This is also a two-step process.

1. Splay x. This moves the x to the root of the tree.
2. Split the right subtree T from the rest of the tree T.

The pseudo code for split is as follows.

```
SPLIT(x)
    SPLAY(x)
    if x.right $\neq$ NIL
        T = x.right
        T.parent = NIL
    else
        T = NIL
    S = x
    S.right = NIL
    x = NIL
    return S, T
```

Search

The search operation in a splay tree is similar to the search operation in ordinary search operations. The difference is we splay the node after the search operation. To search a key, we do the following.

1. Perform the ordinary binary search tree search. Let us suppose the key is found in a node x.
2. Splay the node x.

The pseudo code for the search operation is as follows.

```
SPLAY-SEARCH(key)
    x = TREE-SEARCH(key)
    if x $\neq$ NIL
        SPLAY(x)
```

Insert

To insert a node x, we first insert the node using binary search insertion method. After that, we splay x that moves it to the root of the tree.

```
SPLAY-INSERT(x)
    TREE-INSERT(x)
    SPLAY(x)
```

Delete

To delete a node x, we do the following.

1. Split the tree that breaks it into S and T.
2. Remove x from the root of S.
3. Join T and S.

The pseudo-code is given below.

```
SPLAY-DELETE(x)
    T, S = SPLIT(x)
    if S.left $\neq$ NIL
        S.left.parent = NIL
    JOIN(S.left, T)
```

Amortized Analysis

We show that the amortized cost of an operation in a splay tree is $O(\log n)$ using the potential method (https://algorithmtutor.com/Analysis-of-Algorithm/Amortized-Analysis-of-Algorithms/#The-potential-method). The amortized cost (a) of an operation that changes the state of the tree from T to T' is the real cost (t) of the operation plus the change in potential as follows.

$$a = t + \Phi(T') - \Phi(T)$$

The total amortized cost of sequence of $i = 1, 2, 3, \ldots, m$ operations is therefore,

$$\begin{aligned}\sum_m a &= \sum_m t + \Phi(T_m) - \Phi(T_{m-1}) + \Phi(T_{m-1}) - \Phi(T_{m-2}) + \ldots + \Phi(T_2 \\ &= \sum_m t + \Phi(T_m) - \Phi(T_1)\end{aligned}$$

If we choose the potential in such a way that the difference in potential is always positive, then amortized time is an upper bound on the actual time. Given a node x, the size of the node $size(x)$ is the number of nodes in the tree rooted at x. Let $rank(x)$ is the log base 2 of $size(x)$. Clearly, the rank of the root of the tree is $\log_2 n$. The potential of a tree T is the sum of ranks of all the nodes in T as follows.

$$\Phi(T) = \sum_{x \in T} rank(x)$$

The maximum potential of the tree is $O(n \log n)$ as there are n nodes in the tree and each node has a maximum rank $O(\log n)$.

Now we calculate the amortized cost of the *splaying* operation. Remember splaying is the sequence rotations (zig, zag, zig-zig, zag-zag, zig-zag, zag-zig). In order to calculate the amortized cost of splaying, we first calculate the amortized cost of these individual rotations.

1. **Zig or Zag rotation**: When we do the zig (or zag) rotation, it changes the potential of nodes x and p. The potential of nodes A, B and C remains the same (why??). The change in potential is given by (after simplification),

$$\Delta\Phi = rank'(p) - rank(p) + rank'(x) - rank(x)$$

Where $rank'$ is the rank before the operation and $rank$ is the rank after the operation. As you can see in figure 1, the rank of node p before zig operation is same as the rank of x after the operation. Therefore,

$$\Delta\Phi = rank'(p) - rank(x)$$

Using the fact that $rank'(p) \le rank'(x)$,

$$\begin{aligned} \Delta\Phi &\le rank'(x) - rank(x) \\ &\le 3(rank'(p) - rank(x)) \end{aligned}$$

The amortized cost is, therefore, bounded by,

$$a_{zig} \le 1 + 3(rank'(p) - rank(x))$$

We add 1 because the actual cost of the zig is simply 1 as we do a single rotation.

2. **zig-zig or zag-zag rotation**: This changes the potential of x, p and g. The actual cost of zig-zig (or zag-zag) is 2 as we perform the double rotation. The amortized cost is therefore,

$$\begin{aligned} a_{zig-zig} &= 2 + rank'(g) - rank(g) + rank'(p) - rank(p) + rank'(x) - \\ &= 2 + rank'(g) + rank'(p) - rank(p) - rank(x) \end{aligned}$$

Since $rank(x) < rank(p)$ and $rank'(x) > rank'(p)$, we can write

$$a_{zig-zig} = 2 + rank'(g) + rank'(x) - 2rank(x)$$

Now we simplify this equation by showing the value of t = $2rank'(x) - rank(x) - rank'(g)$ is at least 2. We have,

$$\begin{aligned} t &= 2rank'(x) - rank(x) - rank'(g) \\ &= \log_2\left(\frac{size'(x)}{size(x)}\right) + \log_2\left(\frac{size'(x)}{size'(g)}\right) \end{aligned}$$

We also have $size'(x) \geq size(x) + size'(g)$. In order to make the term t minimum, we chose $size'(x) = 2size(x) = 2size'(g)$. Therefore,

$$t \geq \log_2\left(\frac{2size(x)}{size(x)}\right) + \log_2\left(\frac{2size'(g)}{size'(g)}\right)$$

This shows that t is at least 2. Now the amortized cost becomes

$$\begin{aligned} a_{zig-zig} &\le 2rank'(x) - rank(x) - rank'(g) + rank'(g) + rank'(x) - \\ &= 3(rank'(x) - rank(x)) \end{aligned}$$

3. **zig-zag or zag-zig rotation**: This also changes the potentials of x, p and g. Using the similar analysis as in zig-zig case, we can show that.

$$a_{zig-zag} \le 3(rank'(x) - rank(x))$$

For every zig-zag and zig-zig rotation the amortized cost is bounded by $3(rank'(x) - rank(x))$ and for every zig rotation the amortized cost is bounded by $3(rank'(x) - rank(x))$. We can ignore the +1 in case of single rotation since we do this only one time. The splaying is sequence of these rotations, therefore the amortized cost of splaying is the sum of amortized cost of M operations (rotations). Operation 1 moves x from position $T1$ to $T2$. Similarly, operation 2 moves x from position $T2$ to $T3$ and so on. The last operation moves x to the root. Therefore, the total amortized cost is some of the cost of all these operations.

$$\begin{aligned} a_{total} &= a_1 + a_2 + \ldots + a_M \\ &= 3(rank(T2) - rank(T1)) + 3(rank(T3) - rank(T2)) + \ldots + \\ &= 3(rank(T_M) - rank(T1)) \end{aligned}$$

The potential changes maximum when a node x is a leaf node and we splay it to the root of the tree. In this case, the amortized time is $3 \log n$. This shows that the amortized time for splay operation is $O(\log n)$.

Advantages and disadvantages of Splay Trees

Following are the advantages of splay trees over other binary search trees.

1. In an amortized sense, ignoring constant factors, they are never much worse than constrained structures, and since they adjust according to usage, they can be much more efficient if the usage pattern is skewed.
2. They need less space since no balance or other constraint information is stored.
3. Their access and update algorithms are conceptually simple and easy to implement.

Following are the disadvantages of splay trees.

1. They require more local adjustments, especially during accesses (look-up operations). (Explicitly constrained structures need adjusting only during updates, not during accesses.)
2. Individual operations within a sequence can be expensive, which may be a drawback in real-time applications.

Implementation

I have implemented the splay tree in C++, Java, and Python. The extreme cases are not tested. Please do not use these implementations for the production use, they are for educational use only. You can download the implementations from GitHub.

C++

Click Here (https://github.com/Bibeknam/algorithmtutorprograms/blob/master/data-structures/splay-trees/SplayTree.cpp)

balanced tree (/tags/balanced-tree/) binary tree (/tags/binary-tree/) splay tree (/tags/splay-tree/) tree (/tags/tree/)

Red Black Trees (/Data-Structures/Tree/Red-Black-Trees/)

B-Trees (/Data-Structures/Tree/B-Trees/)