Splay tree

A **splay tree** is a binary search tree with the additional property that recently accessed elements are quick to access again. Like AVL and Red-Black Trees, Splay tree is also self-balancing BST. 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.

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.

All splay tree operations run in O(log n) time on average, where n is the number of entries in the tree. Any single operation can take Theta(n) time in the worst case.

Operations

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 three factors:

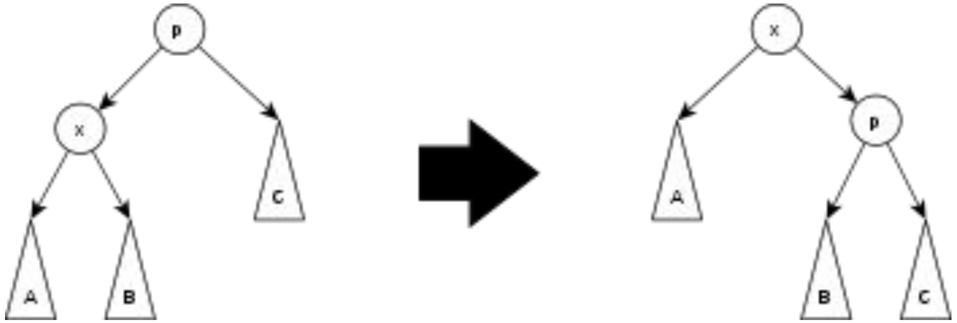
- Whether *x* is the left or right child of its parent node, *p*,
- whether *p* is the root or not, and if not
- whether *p* is the left or right child of *its* parent, *g* (the *grandparent* of x).

It is important to remember to set *gg* (the *great-grandparent* of x) to now point to x after any splay operation. If *gg* is null, then x obviously is now the root and must be updated as such.

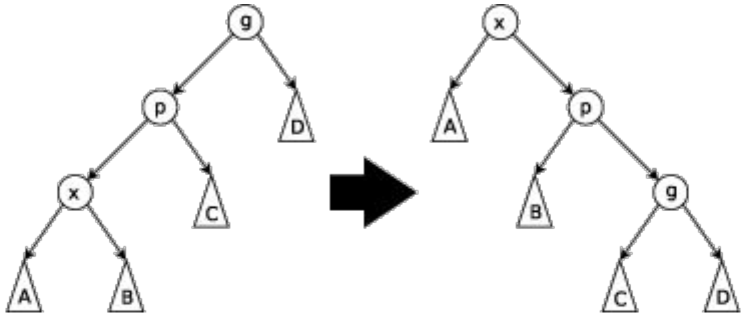
There are three types of splay steps, each of which has two symmetric variants: left- and right-handed. For the sake of brevity, only one of these two is shown for each type. (In the following diagrams, circles indicate nodes of interest and triangles indicate sub-trees of arbitrary size.)

The three types of splay steps are:

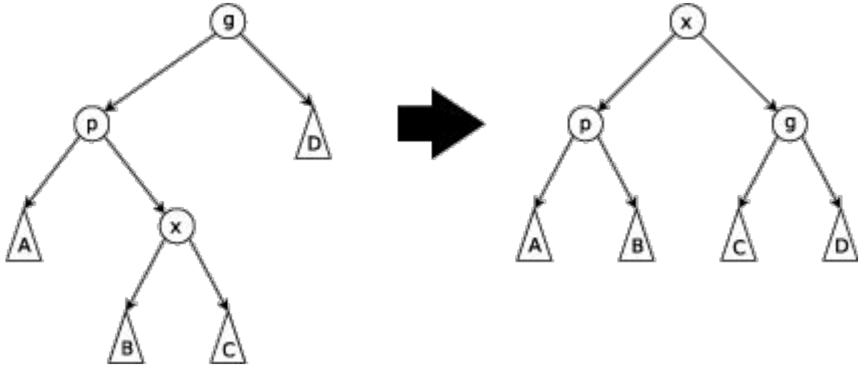
Zig step: this step is done when *p* is the root. The tree is rotated on the edge between *x* and *p*. Zig steps exist to deal with the parity issue, will be done only as the last step in a splay operation, and only when *x* has odd depth at the beginning of the operation.



Zig-zig step: this step is done when *p* is not the root and *x* and *p* are either both right children or are both left children. The picture below shows the case where *x* and *p* are both left children. The tree is rotated on the edge joining *p* with *its* parent *g*, then rotated on the edge joining *x* with *p*. Note that zig-zig steps are the only thing that differentiate splay trees from the *rotate to root* method introduced by Allen and Munro prior to the introduction of splay trees.



Zig-zag step: this step is done when *p* is not the root and *x* is a right child and *p* is a left child or vice versa. The tree is rotated on the edge between *p* and x, and then rotated on the resulting edge between *x* and g.



Join

Given two trees S and T such that all elements of S are smaller than the elements of T, the following steps can be used to join them to a single tree:

- Splay the largest item in S. Now this item is in the root of S and has a null right child.
- Set the right child of the new root to T.

Split

Given a tree and an element *x*, return two new trees: one containing all elements less than or equal to *x* and the other containing all elements greater than *x*. This can be done in the following way:

- Splay *x*. Now it is in the root so the tree to its left contains all elements smaller than *x* and the tree to its right contains all element larger than *x*.
- Split the right subtree from the rest of the tree.

Insertion

To insert a value *x* into a splay tree:

- Insert *x* as with a normal binary search tree.
- when an item is inserted, a splay is performed.
- As a result, the newly inserted node *x* becomes the root of the tree.

Alternatively:

- Use the split operation to split the tree at the value of *x* to two sub-trees: S and T.
- Create a new tree in which *x* is the root, S is its left sub-tree and T its right sub-tree.

Deletion

To delete a node *x*, use the same method as with a binary search tree:

- If *x* has two children:
    - Swap its value with that of either the rightmost node of its left sub tree (its in-order predecessor) or the leftmost node of its right subtree (its in-order successor).
    - Remove that node instead.

In this way, deletion is reduced to the problem of removing a node with 0 or 1 children. Unlike a binary search tree, in a splay tree after deletion, we splay the parent of the removed node to the top of the tree.

Alternatively:

- The node to be deleted is first splayed, i.e. brought to the root of the tree and then deleted. leaves the tree with two sub trees.
- The two sub-trees are then joined using a "join" operation.