Splay Tree Implementation

Ashwin R Bharadwaj
Computer Science Engineering
PES University
Bengaluru, India.
bharathiashwin769@gmail.com

***Abstract*—Splay Trees are a form of binary search tree with the unique property that the most recently accessed element is located at the root of the tree to save access time.**

Keywords: Splay Tree, BST

INTRODUCTION

Splay trees have time complexity of O(logN) in most cases but can have O(N) in some cases. Unlike AVL/Red-Black trees Splay Trees do not have a strictly self-balancing feature. Balancing is achieved using rotations. Splay trees are faster as they are not strictly balanced. They act as a cache. Splay trees are used extensively as they are easy to implement and are more efficient than AVL/Red--Black trees. All operations are bounded by Log(n).

I. SPLAY

When an element is accessed either for deletion/insertion the element/ its closest neighbor is made as the root of the tree using rotations. Once a tree is splayed with respect to a node, that particular node is brought to the root and the inorder traversal of the tee is maintained. There are four cases: left-left, right-right, right-left, left-right. This action does not strictly balance the tree.

II. INSERTION

When we need to add a new element into the tree we check if the element is already present, if it is, we splay the tree to make the node as the root, else we add the element to the tree considering it as a normal BST and then splay the node to make it the root.

III. DELETION

If the element to be deleted is present in the tree the element is deleted considering the tree to be a BST and then splay the parent of the deleted node if it exists. If the element is not present the closest node to the node to be deleted is splayed. Helper functions such as get_parent() are used. Separate functions for root deletion is used.

IV. SEARCH

To check if an element is present in the tree we splay the tree with respect to the element that is in question. Once the tree is splayed if the root has the element that is needed it implies that the element is present else the element is missing.

V. ANALYSIS

When an element is inserted into the tree, the operation is overcharge, the extra credits are stored in the element added. This is done so the extra credits can pay for operation which have worse complexity then the average.

Assume S(x) is the number of nodes in the subtree rooted at x (inclusive).

let $\mu(x) = floor(log\ |S(x)|)$

Each operation can be performed using a constant number of splay operations plus a constant number of comparisons and pointer manipulations

The amount need to withdraw from x's account is at most $\mu(r)$ = log(floor(n)) The extra credits are used to pay for operations that take more than Log(n) time.

Let μ and μ' be the values of μ before and after the splay operation. The only two nodes that change sizes are x (the child) and y (the parent). So the cost of the operation to maintain the credits>0 is

$$\mu'(x) + \mu'(y) - \mu(x) - \mu(y)$$

Since y decreases in size, $\mu'(y) <= \mu'(x)$ and $\mu'(x) = \mu(y)$. Thus the cost of the operation is at most $\mu'(x) - \mu(x)$. Finally, since x increases in size, $\mu'(x) - \mu(x)$ is positive, and the cost is bounded by $3(\mu'(x) - \mu(x))$.

Since we can afford to spend $3(\mu'(x) - \mu(x))+1$, we are left with at least one credit left over to pay for the pointer manipulation and comparisons.

VI. Conclusion

Splay trees are better implementation of self balancing trees as they are easier to program, and give a faster access time to recently used elements We go with amortized time complexity when we feel that not all operations are worse and some can be efficiently done. in splay trees not all splay operations will lead to O(long) worst case complexity.