

SCAPEGOATTREE

**Introduction to Scapegoat Tree:**

A scapegoat tree is a self-balancing binary tree. Arne Andersson invented it in 1989. Scapegoat is based on the common saying, “when something goes wrong, find a scapegoat.” We can say that the scenario for scapegoat trees is the same. Scapegoat trees choose a scapegoat and completely rebuild the subtree rooted at the scapegoat into a complete binary tree.

**Time Complexity and Memory Efficiency:**

The scapegoat tree’s worst-case lookup time is O(log N), where n is defined as the number of entries. It provides O(log N) insertion and deletion time. Unlike all other binary trees, it gives O(log N) lookup time. The scapegoat tree does not use any additional per-node memory overhead used in other binary search trees. Thus, the scapegoat tree is much easier to implement and can reduce one-third of memory overhead. As scapegoat trees expensively choose a scapegoat and completely rebuild the subtree; thus, they have a worst-case update performance of O(n).

**Theorem:**

In a binary search tree, if half of the nodes are on the left of the root and the other half on the right, then it is said to be a weight-balanced binary search tree. An α weight balanced node is defined as meeting a relaxed weight balanced criterion, which is:

Size(left)<= α\* size(node)

Size (right)<= α\* size(node)

A binary search tree that is α- weight balanced must also be α- height-balanced. A not α- a height-balanced tree is also not α- weight balanced. Scapegoat trees are not guaranteed to keep α- weight balance but are always loosely α- height-balanced. This height balance condition can be detected at insertion time and implies that a weight balance condition violation must exist.

**Difference from other Binary Search Trees and AVL Trees:**

* Scapegoat trees are slightly similar to “Red-black trees,” where they both have restrictions on their heights. But the difference between them comes when “Red-black trees” determine where the rotation should occur.
* “Red-black trees” store additional color information inside each node to select the location. In contrast, the scapegoat tree finds a scapegoat that is not α-weight balanced to perform the rebalance operation.
* In the AVL trees, the actual rotation depends on the ‘balances’ of nodes, and it checks the balance value in every insertion or deletion. Still, in the case of the scapegoat tree, it can calculate the balance value only when needed, and this is where a scapegoat needs to be found.
* Unlike most other self-balancing binary trees, scapegoat trees are entirely flexible in their balancing. They support any α, such as 0.5<α<1. A high α value results in fewer balances, making insertion quicker but lookups and deletion slower and vice versa for a low α.

**Operations:**

**Lookup:**

Lookup is not modified from a standard binary search tree in scapegoat trees and has a worst-case time of O(log N). Compared to other self-balancing binary search trees, the reduced node memory overhead can improve the locality of reference and cache.

**Insertion:**

When finding the insertion point, the depth of the new node must also be recorded. This is implemented via a single counter that gets incremented during each iteration of the lookup, effectively counting the number of edges between the root and the inserted node. If this node violates the α height balance property, a rebalance is required.

When rebalance occurs, an entire subtree rooted at a scapegoat undergoes a balancing operation. The scapegoat is defined as an ancestor of the inserted node that isn’t an α-weight balanced. There will always be one such ancestor. Rebalancing any of them will restore the αheight-balanced property.

**Deletion:**

For scapegoat trees, deletion is easier than insertion. To delete an item, scapegoat trees need to store an additional value with the tree data structure. This property which we will call maximum Node count represents the highest achieved node count. It is set to node count when the entire tree is rebalanced. To perform a deletion, we remove the node as we would do for a simple binary search tree, but if

NodeCount <= α\* MaxNodeCount

, then we rebalance the entire tree about the root, remembering to set MaxNodeCount to NodeCount.

In conclusion, we can say that a scapegoat tree is a very effective method for binary tree data structure and is very efficient in terms of storage and time complexity.

References:

1. Morin Pat. “Chapter 8- Scapegoat Trees”- Open Data Structures.
2. Geeks For Geeks, Wikipedia