Scapegoat tree

From Wikipedia, the free encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Čeština
Српски / srpski
ไทย
中文

Edit links

In computer science, a **scapegoat tree** is a self-balancing binary search tree, invented by Arne Andersson[1] and again by Igal Galperin and Ronald L. Rivest.[2] It provides worst-case $O(\log n)$ lookup time, and $O(\log n)$ amortized insertion and deletion time.

Unlike most other self-balancing binary search trees that provide worst case $O(\log n)$ lookup time, scapegoat trees have no additional per-node memory overhead compared to a regular binary search tree: a node stores only a key and two pointers to the child nodes. This makes scapegoat trees easier to implement and, due to data structure alignment, can reduce node overhead by up to one-third.

Scapegoat tree

Type	Tree
Invented	1962
Invented by	Arne Andersson, Igal Galperin, Ronald L. Rivest

Time complexity in big O notation

	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

Contents

Theory [edit]

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

```
size(left) <= α*size(node)
size(right) <= α*size(node)
```

Where size can be defined recursively as:

```
function size(node)
 if node = nil
  return 0
 else
  return size(node->left) + size(node->right) + 1
end
```

An α of 1 therefore would describe a linked list as balanced, whereas an α of 0.5 would only match almost complete binary trees.

A binary search tree that is α-weight-balanced must also be **α-height-balanced**, that is

$$\text{height}(\text{tree}) <= \log_{1/\alpha}(\text{NodeCount}) + 1$$

Scapegoat trees are not guaranteed to keep α-weight-balance at all times, but are always loosely α-height-balanced in

that

```
height(scapegoat tree) <= log_{1/α}(NodeCount) + 1
```

This makes scapegoat trees similar to red-black trees in that they both have restrictions on their height. They differ greatly though in their implementations of determining where the rotations (or in the case of scapegoat trees, rebalances) take place. Whereas red-black trees store additional 'color' information in each node to determine the location, scapegoat trees find a **scapegoat** which isn't α-weight-balanced to perform the rebalance operation on. This is loosely similar to AVL trees, in that the actual rotations depend on 'balances' of nodes, but the means of determining the balance differs greatly. Since AVL trees check the balance value on every insertion/deletion, it is typically stored in each node; scapegoat trees are able to calculate it only as needed, which is only when a scapegoat needs to be found.

Unlike most other self-balancing search trees, scapegoat trees are entirely flexible as to their balancing. They support any α such that $0.5 < \alpha < 1$. A high α value results in fewer balances, making insertion quicker but lookups and deletions slower, and vice versa for a low α. Therefore in practical applications, an α can be chosen depending on how frequently these actions should be performed.

Operations [edit]

Insertion [edit]

Insertion is implemented with the same basic ideas as an unbalanced binary search tree, however with a few significant changes.

When finding the insertion point, the depth of the new node must also be recorded. This is implemented via a simple 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 (defined above), a rebalance is required.

To rebalance, an entire subtree rooted at a **scapegoat** undergoes a balancing operation. The scapegoat is defined as being an ancestor of the inserted node which isn't α-weight-balanced. There will always be at least one such ancestor. Rebalancing any of them will restore the α-height-balanced property.

One way of finding a scapegoat, is to climb from the new node back up to the root and select the first node that isn't α-weight-balanced.

Climbing back up to the root requires O(log *n*) storage space, usually allocated on the stack, or parent pointers. This can actually be avoided by pointing each child at its parent as you go down, and repairing on the walk back up.

To determine whether a potential node is a viable scapegoat, we need to check its α-weight-balanced property. To do this we can go back to the definition:

```
size(left) <= α*size(node)
size(right) <= α*size(node)
```

However a large optimisation can be made by realising that we already know two of the three sizes, leaving only the third having to be calculated.

Consider the following example to demonstrate this. Assuming that we're climbing back up to the root:

```
size(parent) = size(node) + size(sibling) + 1
```

But as:

```
size(inserted node) = 1.
```

The case is trivialized down to:

```
size[x+1] = size[x] + size(sibling) + 1
```

Where x = this node, x + 1 = parent and size(sibling) is the only function call actually required.

Once the scapegoat is found, the subtree rooted at the scapegoat is completely rebuilt to be perfectly balanced.[2] This can be done in O(*n*) time by traversing the nodes of the subtree to find their values in sorted order and recursively choosing the median as the root of the subtree.

As rebalance operations take O(*n*) time (dependent on the number of nodes of the subtree), insertion has a worst-case performance of O(*n*) time. However, because these worst-case scenarios are spread out, insertion takes O(log *n*) amortized time.

Sketch of proof for cost of insertion [edit]

Define the Imbalance of a node *v* to be the absolute value of the difference in size between its left node and right node minus 1, or 0, whichever is greater. In other words:

$$I(v) = \max(|\text{left}(v) - \text{right}(v)| - 1, 0)$$

Immediately after rebuilding a subtree rooted at *v*, I(*v*) = 0.

Lemma: Immediately before rebuilding the subtree rooted at *v*,

(Ω is Big O Notation.)

Proof of lemma:

Let v_0 be the root of a subtree immediately after rebuilding. $h(v_0) = \log(|v_0| + 1)$. If there are $\Omega(|v_0|)$ degenerate insertions (that is, where each inserted node increases the height by 1), then
$I(v) = \Omega(|v_0|)$,
$h(v) = h(v_0) + \Omega(|v_0|)$ and
.

Since before rebuilding, there were insertions into the subtree rooted at that did not result in rebuilding. Each of these insertions can be performed in time. The final insertion that causes rebuilding costs . Using aggregate analysis it becomes clear that the amortized cost of an insertion is :

Deletion [edit]

Scapegoat trees are unusual in that deletion is easier than insertion. To enable deletion, scapegoat trees need to store an additional value with the tree data structure. This property, which we will call MaxNodeCount simply represents the highest achieved NodeCount. It is set to NodeCount whenever the entire tree is rebalanced, and after insertion is set to max(MaxNodeCount, NodeCount).

To perform a deletion, we simply remove the node as you would in 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.

This gives deletion its worst-case performance of O(n) time; however, it is amortized to O(log *n*) average time.

Sketch of proof for cost of deletion [edit]

Warning: the following paragraph is nonsense! "At most n/2 - 1" should be changed into "at least O(n)", and the time complexity should be considered across each level of the tree.

Suppose the scapegoat tree has elements and has just been rebuilt (in other words, it is a complete binary tree). At most deletions can be performed before the tree must be rebuilt. Each of these deletions take time (the amount of time to search for the element and flag it as deleted). The deletion causes the tree to be rebuilt and takes (or just) time. Using aggregate analysis it becomes clear that the amortized cost of a deletion is :

Lookup [edit]

Lookup is not modified from a standard binary search tree, and has a worst-case time of O(log *n*). This is in contrast to splay trees which have a worst-case time of O(*n*). The reduced node memory overhead compared to other self-balancing binary search trees can further improve locality of reference and caching.

See also [edit]

- Splay tree
- Trees
- Tree rotation
- AVL tree
- B-tree
- T-tree
- List of data structures

References [edit]

1. ^ Andersson, Arne (1989). *Improving partial rebuilding by using simple balance criteria*. Proc. Workshop on Algorithms and Data Structures. *Journal of Algorithms* (Springer-Verlag): 393–402. doi:10.1007/3-540-51542-9_33.
2. ^ [a] [b] Galperin, Igal; Rivest, Ronald L. (1993). "Scapegoat trees". *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*: 165–174.

External links [edit]

- Scapegoat Tree Applet by Kubo Kovac
- Scapegoat Trees: Galperin and Rivest's paper describing scapegoat trees
- On Consulting a Set of Experts and Searching (full version paper)
- Open Data Structures - Chapter 8 - Scapegoat Trees

v · t · e	**Tree data structures**
Search trees (dynamic sets/associative arrays)	2–3 · 2–3–4 · AA · (a,b) · AVL · B · B+ · B* · B^X · (Optimal) Binary search · Dancing · HTree · Interval · Order statistic · (Left-leaning) Red-black · **Scapegoat** · Splay · T · Treap · UB · Weight-balanced
Heaps	Binary · Binomial · Fibonacci · Leftist · Pairing · Skew · Van Emde Boas
Tries	Hash · Radix · Suffix · Ternary search · X-fast · Y-fast
Spatial data partitioning trees	BK · BSP · Cartesian · Hilbert R · *k*-d (implicit *k*-d) · M · Metric · MVP · Octree · Priority R · Quad · R · R+ · R* · Segment · VP · X
Other trees	Cover · Exponential · Fenwick · Finger · Fusion · Hash calendar · iDistance · K-ary · Left-child right-sibling · Link/cut · Log-structured merge · Merkle · PQ · Range · SPQR · Top

Categories: Binary trees | Search trees