Chapter 19
Scapegoat Trees

Igal Galperin* Ronald L. Rivest*

Abstract

We present an algorithm for maintaining binary search trees. The amortized complexity per INSERT or DELETE is $O(\log n)$ while the *worst-case* cost of a SEARCH is $O(\log n)$.

Scapegoat trees, unlike most balanced-tree schemes, do not require keeping extra data (e.g. "colors" or "weights") in the tree nodes. Each node in the tree contains only a key value and pointers to its two children. Associated with the root of the whole tree are the only two extra values needed by the scapegoat scheme: the number of nodes in the whole tree, and the maximum number of nodes in the tree since the tree was last completely rebuilt.

In a scapegoat tree a typical rebalancing operation begins at a leaf, and successively examines higher ancestors until a node (the "scapegoat") is found that is so unbalanced that the entire subtree rooted at the scapegoat can be rebuilt at zero cost, in an amortized sense. Hence the name.

1 Introduction

There are a vast number of schemes available for implementing a "dictionary"—supporting the operations INSERT, DELETE, and SEARCH—using balanced binary search trees. Mehlhorn and Tsakalikis [9] survey the recent literature on such data structures. In this paper we propose a new method that achieves optimal amortized costs for update operations (INSERT and DELETE) and optimal *worst-case* cost for SEARCH, *without* requiring the extra information (e.g. colors or weights) normally required by many balanced-tree schemes. This is the first method ever proposed that achieves a worst-case search time of $O(\log n)$ without using such extra information, while maintaining optimal amortized update costs. In addition, the method is quite simple and practical. (Indeed, we wonder why it wasn't discovered much earlier!)

Many balanced-tree schemes are *height-balanced*; the extra information stored at each node helps to enforce a bound on the overall height of the tree. Red-black trees, invented by Bayer [2] and refined by by Guibas and Sedgewick [7], are an elegant example of the height-balanced approach. Red-black trees implement the basic dictionary operations with a *worst-case* cost of $O(\log n)$ per operation, at the cost of storing one extra bit (the "color" of the node) at each node. AVL trees [1] are another well-known example of height-balanced trees.

Other schemes are *weight-balanced* in that the extra information at each node records the size of the subtree rooted at that node. By ensuring that the weights of siblings are approximately equal, an overall bound on the height of the tree is enforced. Nievergelt and Reingold [10] introduce such trees and present algorithms for implementing the basic dictionary operations in $O(\log n)$ worst-case time. Overmars and van Leeuwen in [11] use such techniques too.

The scapegoat method is a modification of the weight-balanced method of Varghese [5, Problem 18-3], who presents an algorithm for maintaining weight-balanced trees with *amortized* cost $O(\log n)$ per operation. Our scheme combines the notions of height-balanced and weight-balanced to achieve an effective algorithm, without storing either height information or weight information at any node.

There have been previous binary tree schemes proposed that do not store any extra information at each node. Splay trees, due to Sleator and Tarjan [13], are perhaps the best-known example; they achieve $O(\log n)$ amortized complexity per operation. However, splay trees do not guarantee a logarithmic worst-case bound on the cost of a SEARCH, and require restructuring even during searches (unlike scapegoat trees, which do have a logarithmic worst-case cost of a SEARCH and do not restructure the tree during searches). Splay trees do have other desirable properties that make them of considerable practical and theoretical interest, however, such as their near-optimality when handling an arbitrary sequence of operations.

Section 2 introduces the basic scapegoat data structure, and some notation. Section 4 describes the algorithm for maintaining scapegoat trees and outlines the

*Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. Supported by NSF grant CCR-8914428, ARO grant N00014-89-J-1988, and the Siemens Corporation. E-mail addresses: galperin@theory.lcs.mit.edu and rivest@theory.lcs.mit.edu.

proof of their features. Section 5 proves the complexity claims. Section 6 describes an algorithm for rebuilding a binary search tree in linear time and logarithmic space. In Section 7 we show how our techniques can be used in $k-d$ trees, and state weak conditions that suffice to allow the application of our techniques to other tree-based data structures. Section 8 reports the results of experimental evaluation of scapegoat trees. We compare a few variations of the scapegoat algorithm and also compare it to other algorithms for maintenance of binary search trees. Finally, Section 9 concludes with some discussion and open problems.

2 Notations

In this section we describe the data structure of a scapegoat tree. Basically, a scapegoat tree consists of an ordinary binary search tree, with two extra values stored at the root.

Each node x of a scapegoat tree maintains the following attributes:

- $key[x]$ – The key stored at node x.
- $left[x]$ – The left child of x.
- $right[x]$ – The right child of x.

We'll also use the notations:

- $size(x)$ – the size of the sub-tree rooted at x (i.e., the number of keys stored in this sub-tree including the key stored at x).
- $brother(x)$ – the brother of node x; the other child of x's parent or NIL.
- $h(x)$ and $h(T)$ – height of a node and a tree respectively. The height of a node is the length of the longest path from that node to a leaf. The height of a tree is the height of its root.
- $d(x)$ – depth of node x. The depth of a node is the length (number of edges) of the path from the root to that node. (The root node is at depth 0.)

Note that values actually stored as fields in a node are used with brackets, whereas values that are computed as functions of the node use parentheses; each node only stores three values: *key*, *left*, and *right*. Computing $brother(x)$ requires knowledge of x's parent. Most importantly, $size(x)$ is *not* stored at x, but can be computed in time $O(size(x))$ as necessary.

The tree T as a whole has the following attributes:

- $root[T]$ – A pointer to the root node of the tree.
- $size[T]$ – The number of nodes in the tree. This is the same as $size(root[T])$. In our complexity analyses we also denote $size[T]$ by n.
- $max\_size[T]$ – The maximal value of $size[T]$ since the last time the tree was completely rebuilt. If DELETE operations are not performed, then the *max_size* attribute is not necessary.

3 Preliminary discussion

SEARCH, INSERT and DELETE operations on scapegoat trees are performed in the usual way for binary search trees, except that, occasionally, after an update operation (INSERT or DELETE) the tree is restructured to ensure that it contains no "deep" nodes.

A binary-tree node x is said to be **α-weight-balanced**, for some α, $1/2 \leq \alpha < 1$, if both

$$size(left[x]) \leq \alpha \cdot size(x), \text{ and} \tag{3.1}$$

$$size(right[x]) \leq \alpha \cdot size(x) . \tag{3.2}$$

We call a tree **α-weight-balanced** if, for a given value of α, $1/2 \leq \alpha < 1$, all the nodes in it are α-weight-balanced. Intuitively, a tree is α-weight-balanced if, for any subtree, the sizes of its left and right subtree are approximately equal.

We denote

$$h_\alpha(n) = \lfloor \log_{(1/\alpha)} n \rfloor,$$

and say that a tree T is **α-height-balanced** if it satisfies

$$h(T) \leq h_\alpha(n), \tag{3.3}$$

where $n = size(T)$. Intuitively, a tree is α-height-balanced if its height is not greater than that of the heighest α-weight-balanced tree of the same size. The following standard lemma justifies this interpretation.

LEMMA 3.1. *If T is an α-weight-balanced binary search tree, then T is α-height-balanced.*

Although scapegoat trees are not guaranteed to be α-weight-balanced at all times, they are **loosely** α-height-balanced, in that they satisfy the bound

$$h(T) \leq h_\alpha(T) + 1, \tag{3.4}$$

where $h_\alpha(T)$ is a shorthand for $h_\alpha(size[T])$.

We assume from now on that a fixed α, $1/2 < \alpha < 1$, has been chosen. For this given α, we call a node of depth greater than $h_\alpha(T)$ a **deep** node. In our scheme the detection of a deep node triggers a restructuring operation.

4 Operations on Scapegoat trees

4.1 Searching a scapegoat tree. In a scapegoat tree, SEARCH operations proceed as in an ordinary binary search tree. No restructuring is performed.

4.2 Inserting into a scapegoat tree. To insert a node into a scapegoat tree, we insert it as we would into an ordinary binary search tree, increment $size[T]$, and set $max\_size[T]$ to be the maximum of $size[T]$ and $max\_size[T]$. Then—if the newly inserted node is deep—we rebalance the tree as follows.

Let x_0 be the newly inserted deep node, and in general let x_{i+1} denote the parent of x_i. We climb the tree, examining x_0, x_1, x_2, and so on, until we find a node x_i that is not α-weight-balanced. Since x_0 is a leaf, $size(x_0) = 0$. We compute $size(x_{j+1})$ using the formula

$$(4.5)\quad size(x_{j+1}) = size(x_j) + size(brother(x_j)) + 1$$

for $j = 1, 2, \ldots, i$, using additional recursive searches.

We call x_i, the ancestor of x_0 that was found that is not α-weight-balanced, the **scapegoat** node. A scapegoat node must exist, by Lemma 5.1 below.

Once the scapegoat node x_i is found, we **rebuild** the subtree rooted at x_i. To rebuild a subtree is to replace it with a 1/2-weight-balanced subtree containing the same nodes. This can be done easily in time $O(size(x_i))$. Section 6 describes how this can be done in space $O(\log n)$ as well.

An alternative way to find a scapegoat node.

As can be seen in Figure 1, x_0 might have more than one weight-unbalanced ancestor. Any weight-unbalanced ancestor of x_0 may be chosen to be the scapegoat. Here we show that another way of finding a weight-unbalanced ancestor x_i of x_0 is to find the deepest ancestor of x_0 satisfying the condition

$$(4.6)\quad i > h_\alpha(size(x_i)).$$

Since this ancestor will often be higher in the tree than the first weight-unbalanced ancestor, it may tend to yield more balanced trees on the average. (In our experiments this heuristic performed better than choosing the first weight-unbalanced ancestor to be the scapegoat.) Inequality (4.6) is satisfied when $x_i = root[T]$, hence this scheme will always find a scapegoat node. The scapegoat node found is indeed weight-unbalanced by Lemma 5.2.

Note that applying condition (4.6) when searching for the scapegoat in the example in Figure 1 indeed results in node 6 being rebuilt, since it is the first ancestor of node 8 that satisfies the inequality.

4.3 Deleting from a scapegoat tree. Deletions are carried out by first deleting the node as we would from an ordinary binary search tree, and decrementing $size[T]$. Then, if

$$(4.7)\quad size[T] < \alpha \cdot max\_size[T]$$

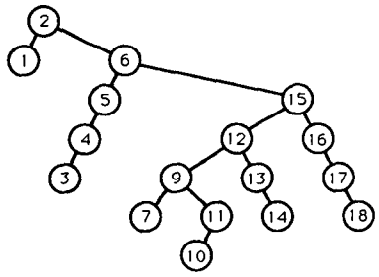

Figure 1: The initial tree, T. For $\alpha = 0.57$, $h_\alpha(17) = h_\alpha(18) = 5$, and T is loosely α-height-balanced (because node 10 is at depth 6). Nodes 2, 5, 6, 12, 15 and 16 are currently weight-unbalanced. Inserting 8 into this tree triggers a rebuild. We chose node 6 to be the scapegoat node.

we rebuild the whole tree, and reset $max\_size[T]$ to $size[T]$.

4.4 Remarks.

- Every time the whole tree is rebuilt $max\_size[T]$ is set to $size[T]$.
- Note that $h_\alpha(T)$ is easily computed from the information stored at the root. (Indeed, it could even be stored there as an extra attribute.)
- We do not need explicit *parent* fields in the nodes to find the scapegoat node, since we are just climbing back up the path we came down to insert the new node; the nodes x_i on this path can be remembered on the stack.

5 Correctness and Complexity

5.1 Correctness. The following two lemmas prove that the algorithm is indeed correct.

The first lemma guarantees that a deep node has an ancestor that in not α-weight-balanced.

LEMMA 5.1. *If x is a node at depth greater than $h_\alpha(T)$ then there is an α-weight-unbalanced ancestor of x.*

Proof. By negation according to equations (3.1) if x is a child of y, then $size(x) \le \alpha \cdot size(y)$. By induction on the path from x to the root, $size(x) \le \alpha^{d(x)} \cdot size[T]$. Therefore, the depth $d(x)$ of a node x is at most $\log_{(1/\alpha)} size[T]$, and the lemma follows.

The following lemma proves that a scapegoat node found using inequality (4.6) is weight-unbalanced.

LEMMA 5.2. *If a binary tree T contains a node x_0 at depth greater than $h_\alpha(n)$, then the deepest ancestor x_i of x_0 that is not α-height-balanced is not α-weight-balanced either.*

Proof. We chose x_i so that the following inequalities are satisfied.

$$i > h_\alpha(size(x_i)) ,$$

and

$$i - 1 \leq h_\alpha(size(x_{i-1})) .$$

Subtracting these two inequalities gives

$$\begin{aligned} 1 &> h_\alpha(size(x_i)) - h_\alpha(size(x_{i-1}) \\ &= \log_{1/\alpha}\left(\frac{size(x_i)}{size(x_{i-1})}\right) . \end{aligned}$$

Therefore,

$$size(x_{i-1}) > \alpha \cdot size(x_i).$$

5.2 Complexity of searching. Since a scapegoat tree is loosely α-height-balanced and α is fixed, a SEARCH operation takes *worst-case* time

$$O(h_\alpha(n)) = O(\log n) .$$

No restructuring or rebalancing operations are performed during a SEARCH. Therefore, not only do scapegoat trees yield an $O(\log n)$ worst-case SEARCH time, but they should also be efficient in practice for SEARCH-intensive applications since no balancing overhead is incurred for searches.

5.3 Complexity of inserting. The following lemma is key to the complexity analysis.

LEMMA 5.3. *The time to find the scapegoat node x_i is $O(size(x_i))$.*

Proof. The dominant part of the cost of finding the scapegoat node x_i is the cost of computing the values $size(x_0)$, $size(x_1)$, ..., $size(x_i)$. Observe that with the optimized *size* calculations described in equation (4.5), each node in the subtree rooted at the scapegoat node x_i is visited exactly once during these computations.

We now analyze the situation where no DELETE operations are done; only INSERT and SEARCH operations are performed. The following lemmas yield Theorem 5.1, which shows that a scapegoat tree is always α-height-balanced if no deletions are performed. The next lemma asserts that rebuilding a tree does not make it deeper.

LEMMA 5.4. *If T is a 1/2-weight-balanced binary search tree, then no tree of the same size has a smaller height.*

Proof. Straightforward.

LEMMA 5.5. *If the root of T is not α-weight-balanced then its heavy subtree contains at least 2 nodes more than its light subtree.*

Proof. Denote by s_h and s_l the sizes of the heavy and the light subtrees respectively. The root of the tree is not α-weight-balanced, hence:

$$s_h > \alpha \cdot (s_h + s_l + 1)$$

This yields:

$$s_h > \frac{\alpha}{1-\alpha} \cdot (s_l + 1)$$

Since $\alpha > 1/2$ and s_h and s_l are both whole numbers, we get:

$$s_h \geq s_l + 2 .$$

A tree T is **complete** of height h if a node cannot be added to T without making its height greater than h. A complete tree of height h has $2^{h+1} - 1$ nodes.

LEMMA 5.6. *If T is not α-weight-balanced and T contains only one node at depth $h(T)$ then rebuilding T decreases its height.*

Proof. Let x be the deepest node of T, and let T_l be the light subtree of T. Let T_l' be the tree we get by removing x from T_l if x is a node of T_l, or T_l itself if x is not a node of T_l. By Lemma 5.5, T_l' is not a complete tree of height $h(T)-1$. Therefore, Lemma 5.4 completes the proof.

THEOREM 5.1. *If a scapegoat tree T was created from a 1/2-weight-balanced tree by a sequence of* INSERT *operations, then T is α-height-balanced.*

Proof. By induction on the number of insert operations using Lemma 5.6.

Let us now consider a sequence of n INSERT operations, beginning with an empty tree. We wish to show that the amortized complexity per INSERT is $O(\log n)$.

For an overview of amortized analysis, see Cormen et al. [5]. We begin by defining a nonnegative *potential function* for the tree. Let

$$\Delta(x) = |size(left[x]) - size(right[x])|,$$

and define the potential of node x to be 0 if $\Delta(x) < 2$, and $\Delta(x)$ otherwise. The potential of a 1/2-weight-balanced node is thus 0, and the potential of a node x that is not α-weight-balanced is $\Theta(size(x))$. (Note that $\Delta(x)$ is not stored at x nor explicitly manipulated during any update operations; it is just an accounting fiction representing the amount of "prepaid work" available at node x.) The potential of the tree is the sum of the potentials of its nodes.

It is easy to see that by increasing their cost by only a constant factor, the insertion operations that build up a scapegoat tree can pay for the increases in potential at the nodes. That is, whenever we pass by a node x to insert a new node as a descendant of x, we can pay

for the increased potential in x that may be required by the resulting increase in $\Delta(x)$.

The potential of the scapegoat node, like that of any non-α-weight-balanced node, is $\Theta(size(x_i))$. Therefore, this potential is sufficient to pay for finding the scapegoat node and rebuilding its subtree. (Each of these two operations has complexity $\Theta(size(x_i))$.) Furthermore, the potential of the rebuilt subtree is 0, so the entire initial potential may be used up to pay for these operations. This completes the proof of the following theorem.

THEOREM 5.2. *A scapegoat tree can handle a sequence of n* INSERT *and m* SEARCH *operations, beginning with an empty tree, with $O(\log n)$ amortized cost per* INSERT *and $O(\log k)$ worst-case time per* SEARCH, *where k is the size of the tree the* SEARCH *is performed on.*

5.4 Complexity of deleting. The main lemma of this section, Lemma 5.10, states that scapegoat trees are loosely α-height-balanced (recall inequality (3.4)). Since we perform $\Omega(n)$ operations between two successive rebuilds due to delete operations we can "pay" for them in the amortized sense. Therefore, combining Lemma 5.10 with the preceding results completes the proof of the following theorem.

THEOREM 5.3. *A scapegoat tree can handle a sequence of n* INSERT *and m* SEARCH *or* DELETE *operations, beginning with an empty tree, with $O(\log n)$ amortized cost per* INSERT *or* DELETE *and $O(\log k)$ worst-case time per* SEARCH, *where k is the size of the tree the* SEARCH *is performed on.*

The first lemma generalizes Theorem 5.1.

LEMMA 5.7. *For any tree T let $T' =$* INSERT(T, x), *then*

$$h(T') \leq \max(h_\alpha(T'), h(T)) \ .$$

Proof. If the insertion of x did not trigger a rebuild, then the depth of x is at most $h_\alpha(T')$ and we are done.

Otherwise, suppose x was initially inserted at depth d in T, where $d > h_\alpha(T')$, thereby causing a rebuild. If T already contained other nodes of depth d we are done, since a rebuild does not make a tree deeper. Otherwise, the arguments in section 5.1 and Lemma 5.6 apply.

LEMMA 5.8. *If $h_\alpha(T)$ does not change during a sequence of* INSERT *and* DELETE *operations then $\max(h_\alpha(T), h(T))$ is not increased by that sequence.*

Proof. A DELETE operation can not increase $\max(h_\alpha(T), h(T))$. For an INSERT we have

$$h(T') \leq \max(h_\alpha(T'), h(T))$$

by Lemma 5.7. Hence

$$\max(h_\alpha(T'), h(T')) \leq \max(h_\alpha(T'), h(T)) = \max(h_\alpha(T), h(T)) \ .$$

The lemma follows by induction on the number of operations in the sequence.

LEMMA 5.9. *For $T' =$* INSERT(T, x), *if T is loosely α-height-balanced but is not α-height-balanced, and $h_\alpha(T') = h_\alpha(T) + 1$, then T' is α-height-balanced.*

Proof. We know that

$$h(T) = h_\alpha(T) + 1.$$

Hence

$$h(T) = h_\alpha(T').$$

Combining this with Lemma 5.7 gives

$$h(T') \leq h_\alpha(T') \ ,$$

i.e., T' is height balanced.

Now we have the tools to prove the main lemma of this section.

LEMMA 5.10. *A scapegoat tree built by* INSERT *and* DELETE *operations from an empty tree is always loosely α-height-balanced.*

Proof. Let $o_1, \ldots, o_n$ be a sequence of update operations that is applied to a 1/2-weight-balanced scapegoat tree, up until (but not including) the first operation, if any, that causes the entire tree to be rebuilt. To prove the lemma it suffices to show that during this sequence of operations the tree is always loosely α-height-balanced. During any sequence of update operations that do not change $h_\alpha(T)$, a loosely α-height-balanced tree remains loosely α-height-balanced, and an α-height-balanced tree remains α-height-balanced, by Lemma 5.8. Therefore, let $o_{i_1}, \ldots, o_{i_k}$ be the subsequence (not necessarily successive) of operations that change $h_\alpha(T)$. An INSERT operation in this subsequence leaves the tree α-height-balanced, by Lemma 5.9. The usage of *max_size*$[T]$ in DELETE implies that there are no two successive DELETE operations in this subsequence, since the entire tree would be rebuilt no later than the second such DELETE operation. Therefore a DELETE operation in this subsequence must operate on an α-height-balanced tree. Since the DELETE operation decreases $h_\alpha(T)$ by just one, the result is a loosely α-height-balanced tree. The lemma follows from applying the preceding lemmas in an induction on the number of operations.

This completes the proof of Theorem 5.3.

6 Rebuilding in place

A straightforward way of rebuilding a tree is to use a stack of logarithmic size to traverse the tree in-order in linear time and copy its nodes to an auxiliary array. Then build the new 1/2-weight-balanced tree using a "divide and conquer" method. This yields $O(n)$ time and space complexity. Our methods improve the space complexity to logarithmic.

6.1 A simple recursive method. The first algorithm links the elements together into a list, rather than copying them into an array.

The initial tree-walk is implemented by the following procedure, FLATTEN. A call of the form FLATTEN(x, NIL) returns a list of the nodes in the subtree rooted at x, sorted in nondecreasing order. In general, a call of the form FLATTEN(x, y) takes as input a pointer x to the root of a subtree and a pointer y to the first node in a list of nodes (linked using their *right* pointer fields). The set of nodes in the subtree rooted at x and the set of nodes in the list headed by y are assumed to be disjoint. The procedure returns the list resulting from turning the subtree rooted at x into a list of nodes, linked by their *right* pointers, and appending the list headed by y to the result.

FLATTEN(x, y)

```
1 if x = NIL
2    then return y
3 right[x] ← FLATTEN(right[x], y)
4 return FLATTEN(left[x], x)
```

The procedure runs in time proportional to the number of nodes in the subtree, and in space proportional to its height

The following procedure, BUILD-TREE, builds a 1/2-weight-balanced tree of n nodes from a list of nodes headed by node x. It is assumed that the list of nodes has length at least $n + 1$. The procedure returns the $n+1$st node in the list, s, modified so that *left*[s] points to the root r of the n-node tree created.

BUILD-TREE(n, x)

```
1 if n = 0
2    then left[x] ← NIL
3         return x
4 r ← BUILD-TREE(⌈(n − 1)/2⌉, x)
5 s ← BUILD-TREE(⌊(n − 1)/2⌋ , right[r])
6 right[r] ← left[s]
7 left[s] ← r
8 return s
```

A call to BUILD-TREE(n, *scapegoat*) runs in time $O(n)$ and uses $O(\log n)$ space.

The following procedure, REBUILD-TREE, takes as input a pointer *scapegoat* to the root of a subtree to be rebuilt, and the size n of that subtree. It returns the root of the rebuilt subtree. The rebuilt subtree is 1/2-weight-balanced. The procedure utilizes the procedures FLATTEN and BUILD-TREE defined above, and runs in time $O(n)$ and space proportional to the height of the input subtree.

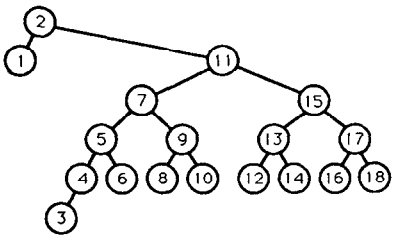

Figure 2: The tree INSERT(T, 8), where T is the tree of Figure 1.

REBUILD-TREE(n, *scapegoat*)

```
1 create a dummy node w
2 z ← FLATTEN(scapegoat, w)
3 BUILD-TREE(n, z)
4 return left[w]
```

Figures 1 and 2 illustrate this process.

6.2 A non-recursive method. This section suggests a non-recursive method for rebuilding a tree in logarithmic space, that proved to be faster in our experiments than the previous version. We only sketch the procedure here; details are given the full version of this paper.

We traverse the old tree in-order. Since the number of nodes in the tree is known, the new place of each node we encounter can be uniquely determined. Every node is "plugged into" the right place in the new tree upon being visited, thereby creating the new tree in place.

We need to keep track of the "cutting edge" of the two tree traversals as shown in Figure 3. Since the depth of both trees is logarithmic, two logarithmic size stacks suffice for this purpose.

7 More Applications of Scapegoat Techniques

The ideas underlying scapegoat trees are that of finding and rebuilding a subtree whose root is not weight-balanced when the tree gets too deep, and periodically rebuilding the root after enough DELETEs occurred. This technique can be applied to other tree-like data structures. To allow this, it should be possible to find the scapegoat node and to rebuild the subtree rooted at it. The time to find the scapegoat and the rebuilding

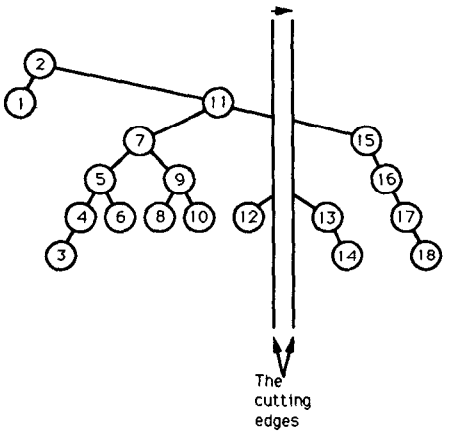

Figure 3: Non-recursive rebuilding in place. An intermediate state during the execution of a rebuilding in place of the tree INSERT$(T, 8)$. Node 11 is the new root of the subtree being rebuilt. (See T in Figure 1).

time does not have to be linear in the number of nodes in the subtree being rebuilt, as was the case with binary search trees (Theorem 5.3). It is also not necessary for the rebuilding algorithm to yield a perfectly balanced subtree. These generalizations of the main theorem, allow us to apply scapegoat techniques to an array of other tree-like data structures.

7.1 A stronger version of the main theorem. Suppose for a class of trees, some fixed $\alpha_{bal} \geq 1/2$ and a function F, $F(n) = \Omega(1)$, satisfying $F(Cn) = O(F(n))$ for any constant C, there exists an algorithm that when given n nodes can in $O(nF(n))$ steps build a tree containing those nodes that is α_{bal}-weight-balanced. We'll call such a rebuilding routine a **α_{bal}-relaxed** rebuilding routine. Also suppose there exists an algorithm that can find an ancestor of a given node that is not weight-balanced in $O(nF(n))$ time, where n is the size of the subtree rooted at the scapegoat node, provided such an ancestor exists. Then we can use scapegoat techniques to support dynamic updates to this class with amortized logarithmic complexity. When $F(n)$ is constant and $\alpha_{bal} = 1/2$, we have the previously handled situation of Theorem ??

For a fixed $\alpha_{trigger}$, $\alpha_{trigger} > \alpha_{bal}$, an insertion of a deep node with respect to $\alpha_{trigger}$ would trigger a rebuilding. Lemma 5.1 guarantees that such a node has an $\alpha_{trigger}$-weight-unbalanced ancestor. However, for any constants α, β, $1/2 < \alpha < \beta$ and for n large enough there exists a β-weight-unbalanced tree of size n that can be rebuilt into a deeper α-weight-balanced tree. Hence, we cannot choose any $\alpha_{trigger}$-weight unbalanced ancestor of the deep node to be the scapegoat. However, if we choose as a scapegoat an ancestor x of the deep node that satisfies condition (4.6):

$$h(x) > h_{\alpha_{trigger}}(size(x)), \tag{7.8}$$

we can prove the following theorem.

THEOREM 7.1. *A relaxed scapegoat tree can handle a sequence of n* INSERT *and m* SEARCH *or* DELETE *operations, beginning with an empty tree, with an amortized cost of $O(F(n)\log_{1/\alpha_{trigger}} n)$ per* INSERT *or* DELETE *and $O(\log_{1/\alpha_{trigger}} k)$ worst-case time per* SEARCH, *where k is the size of the tree the* SEARCH *is performed on.*

Proof. (sketch) The existence of an ancestor that satisfies equation (7.8) is guaranteed as explained in Section 5 (the root of the tree satisfies it). It follows from the way the scapegoat was chosen that rebuilding the subtree rooted at it decreases the depth of the rebuilt subtree, allowing us to prove a result similar to Lemma 5.7. The other lemmas leading to Theorem 5.3 can also be proven for relaxed rebuilding. Hence, we can indeed support a tree of depth at most $\log_{1/\alpha_{trigger}} k+1$, where k is the size of the tree, thereby establishing the bound on the worst-case search time.

To prove the amortized bound on the complexity of updates we will define a potential function Φ in an inductive manner. Let the potential of the nodes in a subtree that was just rebuilt and of newly inserted nodes be 0. Every time a node is traversed by an update operation, increase its potential by $F(N)$, where N is the size of the subtree rooted at that node. For any update operation, the node whose potential is increased the most is the root. Hence the total price of the update operation is bounded by $(F(N)+1)\log_{1/\alpha_{trigger}} N = O(F(N)\log_{1/\alpha_{trigger}} N)$ as $F(n) = \Omega(1)$.

If the root is $\alpha_{trigger}$-weight unbalanced, then CN different update operations traversed it since it was inserted or last rebuilt. Now $C \geq C_0$, where

$$C_0 = \frac{\alpha_{trigger} - \alpha_{bal}}{2\alpha_{trigger}\alpha_{bal}}.$$

At each one of the last C_0 passes the potential of the root was increased by at least $F((1-C_0)N)$. Hence, the total potential stored at the root is at least $C_0 NF((1-C_0)N) = O(NF(N))$, allowing it to pay for the rebuilding operation.

7.2 Scapegoat $k-d$ trees. Bentley introduced $k-d$ trees in [3]. He proved average-case bounds of $O(\lg n)$ for a tree of size n for both updates and searches. Bentley in [4] and Overmars and van Leeuwen in [11] propose a scheme for dynamic maintenance of $k-d$ trees that achieves a logarithmic worst-case bound

for searches with an average-case bound of $O((\lg n)^2)$ for updates. Both use an idea similar to ours of rebuilding weight-unbalanced subtrees. Overmars and van Leeuwen called their structure pseudo $k-d$ trees.

Scapegoat $k-d$ trees achieve logarithmic worst-case bounds for searches and a $\log^2 n$ amortized bound for updates. (The analysis of updates in [11] and [4] can be improved to yield amortized rather than average-case bounds.) However, scapegoat $k-d$ trees do not require maintaining extra data at the nodes. Also we believe they might prove to be faster in practice as they do not rebuild every weight-unbalanced node, thereby allowing for it to become balanced by future updates.

Applying Theorem 7.1 we get:

THEOREM 7.2. *A scapegoat $k-d$ tree can handle a sequence of n* INSERT *and m* SEARCH *or* DELETE *operations, beginning with an empty tree, with $O(\log^2 n)$ amortized cost per* INSERT *or* DELETE *and $O(\log k)$ worst-case time per* SEARCH, *where k is the size of the tree the* SEARCH *is performed on.*

Proof. To apply Theorem 7.1 we use the algorithm Bentley proposes in [3] for building a perfectly balanced $k-d$ tree of N nodes in $O(kN \lg N)$, by taking as a splitting point the median with respect to the splitting coordinate. Finding the scapegoat is done in a manner similar to that in binary search trees.

7.3 Scapegoat trees for orthogonal queries. For keys which are d dimensional vectors one may wish to specify a range for each component of the key and ask how many keys have all components in the desired range. Leuker in [8] proposed an algorithm that handles range queries in $O(log^d n)$ worst-case time where n is the size of the tree. Updates are handled in $O(nlog^d n)$ amortized time.

Leuker's paper proves that given a list of n keys a 1/3-balanced tree may be formed in $O(nlog^{\min(1,d-1)} n)$ time.

Using this in Theorem 7.1 proves

THEOREM 7.3. *A scapegoat orthogonal tree can handle a sequence of n* INSERT *and m* SEARCH *or* DELETE *operations, beginning with an empty tree, with $O(\log^{\min(2,d)} n)$ amortized cost per* INSERT *or* DELETE *and $O(\log^d k)$ worst-case time per range query, where k is the size of the tree the range query is performed on.*

Note that our algorithm improves Leuker's amortized bounds for updates, and does not require storage of balancing data at the nodes of the tree.

7.4 Scapegoat quad trees. Quad trees were introduced by Finkel and Bentley in [6]. They achieve a worst-case bound of $O(log_2 N)$ per search. (As in a d dimensional quad tree every node has 2^d children naively one could expect a $O(log_{2^d} N)$ worst-case search time.) They do not address deletion, and give only experimental results for insertion times. Samet in [12] proposed an algorithm for deletions. Overmars and van Leeuwen in [11] introduced pseudo-quad trees – a dynamic version of quad trees. They suggest an algorithm for achieving $O((\lg N)^2)$ average insertion and deletion times, where N is the number of insertions, while improving the worst-case search time to $\log_{d+1-\delta} n + O(1)$, where d is the dimension of the tree, n the size of the tree the search is performed on, and δ an arbitrary constant satisfying $1 < \delta < d$.

Comparing scapegoat quad trees can be compared to pseudo-quad trees we can point out that:

- Scapegoat trees offer worst-case search time of $C \log_{d+1} n$ for any constant C, or following the original notations of Overmars and van Leeuwen $\log_{d+1-\delta} n$ for any positive constant δ (note that we do not require $1 < \delta$).

- The bounds on updates are improved from average-case to amortized bounds. (Though careful analysis of the algorithm in [11] can yield amortized bounds too.)

- Scapegoat trees do not require maintenance of extra data at the nodes regarding the weight of the children of each node. This can be quite substantial in this case, as ecah node has 2^d children, where d is the dimension of the tree.

- Scapegoat trees might prove faster in practice, as they do not require the rebuilding of every weight-unbalanced node, thereby allowing some nodes to be balanced by future updates. Also more compact storage might result in greater speed.

We call a multi-way node, x, α-weight-balanced, if the every child y of x, satisfies $size(y) \leq \alpha size(x)$. Weight and height balanced trees are defined in a way similar to that used for binary trees.

Theorem 2.2.3 in [11] suggests how to build a $1/(d+1)$ weight balanced pseudo-quad tree in $O(n \log n)$ time. Finding a scapegoat in a multiway tree can be done by traversing a tree in a manner similar to that described for binary trees, starting at the deep node and going up. Plugging this into Theorem 7.1 proves:

THEOREM 7.4. *A scapegoat quad tree can handle a sequence of n* INSERT *and m* SEARCH *or* DELETE *operations, beginning with an empty tree, with $O(\log^2 n)$ amortized cost per* INSERT *or* DELETE *and $O(\log_{d+1-\delta} k)$ worst-case time per* SEARCH, *where k is the size of the tree the* SEARCH *is performed on.*

8 Experimental Results

We compared scapegoat trees to two other schemes for maintaining binary search trees – red-black trees and splay trees. We also compare the performance of scapegoat trees for different values of α. We compare the performance for each one of the three operations INSERT, DELETE, and SEARCH separately. We consider two types of workloads – uniformly distributed inputs and sorted inputs. The results are summarized in Tables 4 and 5. The tables list average time in seconds per $128K$ (131,072) operations.

To compare the performance for uniformly distributed inputs, we inserted the nodes into a tree in a random order, then searched for randomly chosen nodes in the tree, and finally deleted all of the nodes in random order. We tried trees of three sizes – $1K$, $8K$ and $64K$. The results appear in Table 4.

Table 5 summarizes the results of the comparison for sorted sequences. Here too we tried three tree sizes – $1K$, $8K$ and $64K$. First we inserted the nodes into a tree in increasing order of keys, then we searched for all of the keys that were inserted in increasing order, and finally we deleted all of the nodes in increasing order of keys.

For uniformly distributed sequences our experiments show that one can choose an α so that scapegoat trees outperform red-black trees and splay trees on all three operations. However, for the insertion of sorted sequences scapegoat trees are noticeably slower than the other two data structures. Hence, in practical applications, it would be advisable to use scapegoat trees when the inserted keys are expected to be roughly randomly distributed, or when the application is search intensive.

Unsurprisingly, as the value of α is increased the SEARCH and DELETE operations perform faster, while the INSERTs become slower. Therefore, in practical applications the value of α should be chosen according to the expected frequency in which these operations will be performed.

For the splay trees we used top-down splaying as suggested by Sleator and Tarjan in [13]. The implementation of red-black trees follows Chapter 14 in Cormen, Leiserson and Rivest [5].

The non-recursive method of rebuilding subtrees described in section 6.2 proved to work faster than the method described in section 6.1 by 25% – 30%. In section 4 we described two ways to choose the scapegoat. Our experiments suggest that checking for condition (4.6) yields a better overall performance.

In our experiments we used a variant of the non-recursive rebuilding algorithm described by the pseudo-code in section 6.2 which inserts all the nodes at the deepest level of the newly-built subtree at the leftmost possible positions, instead of spreading them evenly. This simplified the code somewhat and yielded a 6% – 9% percent speedup over the version described by the pseudo-code.

		1 K			8 K			64 K		
		search	insert	delete	search	insert	delete	search	insert	delete
splay trees		2.65	5.20	6.05	3.90	6.72	7.76	5.38	8.15	9.18
red-black trees		6.28	7.13	8.54	8.95	10.10	11.41	12.19	13.42	14.87
scapegoat trees	α=0.55	1.65	9.46	3.49	2.28	10.14	4.13	3.16	9.61	5.13
	α=0.6	1.66	5.88	3.87	2.37	7.05	4.42	3.45	7.89	5.24
	α=0.65	1.73	4.93	4.49	2.49	6.45	4.81	3.51	7.82	5.56
	α=0.7	1.87	4.22	5.62	2.62	5.72	5.91	3.53	6.93	6.52
	α=0.75	1.87	3.83	6.88	2.71	4.70	7.73	3.73	5.98	8.84

Figure 4: Results of comparative experiments for uniformly distributed inputs. Execution time in seconds per $128K$ (131,072) operations for splay trees, red-black trees and scapegoat trees with α varying between 0.55 – 0.75 for tree sizes of $1K$, $8K$ and $64K$.

9 Discussion and Conclusions

We leave as an open problem the average-case analysis of scapegoat trees (say, assuming that all permutations of the input keys are equally likely).

To summarize: scapegoat trees are the first "unencumbered" tree structure (i.e., having no extra storage per tree node) that achieves a worst-case SEARCH time of $O(\log n)$, with reasonable amortized update costs.

Acknowledgments

We are greatful to Jon Bentley for suggesting the applicability of our techniques to $k-d$ trees. We thank Charles Leiserson for some helpful discussions. We also thank David Williamson and John Leo for allowing us to use their software in our experiments.

References

[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[2] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[3] Jon L. Bentley. Multidimensional binary search trees used fro associative searching. *Communications of the ACM*, 19:509–517, 1975.

[4] Jon L. Bentley. Multidimensional binary search trees in database applications. *IEEE Transactions on Soft-*

		1 K			8 K			64 K		
		search	insert	delete	search	insert	delete	search	insert	delete
splay trees		2.83	7.75	6.17	3.59	9.49	7.13	4.96	11.19	8.49
red-black trees		2.40	3.38	3.50	2.65	3.21	3.81	2.68	3.46	3.74
scapegoat trees	α=0.55	1.37	21.25	2.90	1.88	29.84	3.45	2.50	36.91	3.72
	α=0.6	1.41	18.67	3.08	1.89	25.38	3.31	2.44	33.26	3.75
	α=0.65	1.38	16.17	3.04	1.88	23.39	3.19	2.58	29.47	3.93
	α=0.7	1.36	15.80	3.02	1.90	20.02	3.36	2.59	24.31	3.87
	α=0.75	1.54	13.94	3.83	1.88	19.25	3.49	2.67	24.79	4.15

Figure 5: Results of comparative experiments for monotone inputs. Execution time in seconds per $128K$ $(131,072)$ operations for splay trees, red-black trees and scapegoat trees with α varying between $0.55 - 0.75$ for tree sizes of $1K$, $8K$ and $64K$.

ware Engineering, 5(4):333–340, 1979.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[6] R. A. Finkel and J. L. Bentley. Quad-trees; a data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[7] Leo J. Guibas and Robert Sedgewick. A diochromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21. IEEE Computer Society, 1978.

[8] George S. Leuker. A data structure for orthogonal range queries. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 28–34. IEEE Computer Society, 1978.

[9] K. Mehlhorn and A. Tsakalidis. Data structures. In J. van Leeuwen, editor, *Algorithms and Complexity*, volume A, chapter 6, pages 301–341. Elsevier, 1990.

[10] I. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2:33–43, 1973.

[11] Mark H. Overmars and Jan van Leeuwen. Dynamic multi-dimentional data structures based on quad- and $k-d$ trees. *Acta Informatica*, 17:267–285, 1982.

[12] Hanan Samet. Deletion in two-dimentional quad trees. *Communications of the ACM*, 23(12):703–710, 1980.

[13] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.