Chapter 8

Scapegoat Trees

In this chapter, we study a binary search tree data structure, the ScapegoatTree. This structure is based on the common wisdom that, when something goes wrong, the first thing people tend to do is find someone to blame (the *scapegoat*). Once blame is firmly established, we can leave the scapegoat to fix the problem.

A ScapegoatTree keeps itself balanced by *partial rebuilding operations*. During a partial rebuilding operation, an entire subtree is deconstructed and rebuilt into a perfectly balanced subtree. There are many ways of rebuilding a subtree rooted at node u into a perfectly balanced tree. One of the simplest is to traverse u's subtree, gathering all its nodes into an array, a, and then to recursively build a balanced subtree using a. If we let m = a.length/2, then the element a[m] becomes the root of the new subtree, a[0],...,a[m−1] get stored recursively in the left subtree and a[m+1],...,a[a.length−1] get stored recursively in the right subtree.

ScapegoatTree

```
void rebuild(Node<T> u) {
  int ns = size(u);
  Node<T> p = u.parent;
  Node<T>[] a =  Array.newInstance(Node.class, ns);
  packIntoArray(u, a, 0);
  if (p == nil) {
    r = buildBalanced(a, 0, ns);
    r.parent = nil;
  } else if (p.right == u) {
    p.right = buildBalanced(a, 0, ns);
```

```
    p.right.parent = p;
  } else {
    p.left = buildBalanced(a, 0, ns);
    p.left.parent = p;
  }
}
int packIntoArray(Node<T> u, Node<T>[] a, int i) {
  if (u == nil) {
    return i;
  }
  i = packIntoArray(u.left, a, i);
  a[i++] = u;
  return packIntoArray(u.right, a, i);
}
Node<T> buildBalanced(Node<T>[] a, int i, int ns) {
  if (ns == 0)
    return nil;
  int m = ns / 2;
  a[i + m].left = buildBalanced(a, i, m);
  if (a[i + m].left != nil)
    a[i + m].left.parent = a[i + m];
  a[i + m].right = buildBalanced(a, i + m + 1, ns - m - 1);
  if (a[i + m].right != nil)
    a[i + m].right.parent = a[i + m];
  return a[i + m];
}
```

A call to `rebuild(u)` takes $O(\texttt{size}(u))$ time. The resulting subtree has minimum height; there is no tree of smaller height that has `size(u)` nodes.

8.1 ScapegoatTree: A Binary Search Tree with Partial Rebuilding

A `ScapegoatTree` is a `BinarySearchTree` that, in addition to keeping track of the number, `n`, of nodes in the tree also keeps a counter, `q`, that maintains an upper-bound on the number of nodes.

ScapegoatTree

```
int q;
```

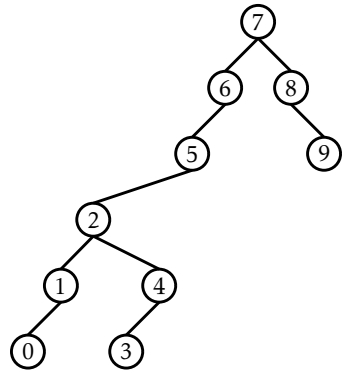

Figure 8.1: A ScapegoatTree with 10 nodes and height 5.

At all times, n and q obey the following inequalities:

$$q/2 \le n \le q \ .$$

In addition, a ScapegoatTree has logarithmic height; at all times, the height of the scapegoat tree does not exceed:

$$\log_{3/2} q \le \log_{3/2} 2n < \log_{3/2} n + 2 \ . \tag{8.1}$$

Even with this constraint, a ScapegoatTree can look surprisingly unbalanced. The tree in Figure 8.1 has $q = n = 10$ and height $5 < \log_{3/2} 10 \approx 5.679$.

Implementing the `find(x)` operation in a ScapegoatTree is done using the standard algorithm for searching in a BinarySearchTree (see Section 6.2). This takes time proportional to the height of the tree which, by (8.1) is $O(\log n)$.

To implement the `add(x)` operation, we first increment n and q and then use the usual algorithm for adding x to a binary search tree; we search for x and then add a new leaf u with u.x = x. At this point, we may get lucky and the depth of u might not exceed $\log_{3/2} q$. If so, then we leave well enough alone and don't do anything else.

Unfortunately, it will sometimes happen that `depth(u)` $> \log_{3/2} q$. In this case, we need to reduce the height. This isn't a big job; there is only

one node, namely u, whose depth exceeds $\log_{3/2} q$. To fix u, we walk from u back up to the root looking for a *scapegoat*, w. The scapegoat, w, is a very unbalanced node. It has the property that

$$\frac{\texttt{size(w.child)}}{\texttt{size(w)}} > \frac{2}{3} \ , \tag{8.2}$$

where w.child is the child of w on the path from the root to u. We'll very shortly prove that a scapegoat exists. For now, we can take it for granted. Once we've found the scapegoat w, we completely destroy the subtree rooted at w and rebuild it into a perfectly balanced binary search tree. We know, from (8.2), that, even before the addition of u, w's subtree was not a complete binary tree. Therefore, when we rebuild w, the height decreases by at least 1 so that height of the ScapegoatTree is once again at most $\log_{3/2} q$.

ScapegoatTree

```
boolean add(T x) {
  // first do basic insertion keeping track of depth
  Node<T> u = newNode(x);
  int d = addWithDepth(u);
  if (d > log32(q)) {
    // depth exceeded, find scapegoat
    Node<T> w = u.parent;
    while (3*size(w) <= 2*size(w.parent))
      w = w.parent;
    rebuild(w.parent);
  }
  return d >= 0;
}
```

If we ignore the cost of finding the scapegoat w and rebuilding the subtree rooted at w, then the running time of add(x) is dominated by the initial search, which takes $O(\log q) = O(\log n)$ time. We will account for the cost of finding the scapegoat and rebuilding using amortized analysis in the next section.

The implementation of remove(x) in a ScapegoatTree is very simple. We search for x and remove it using the usual algorithm for removing a node from a BinarySearchTree. (Note that this can never increase the

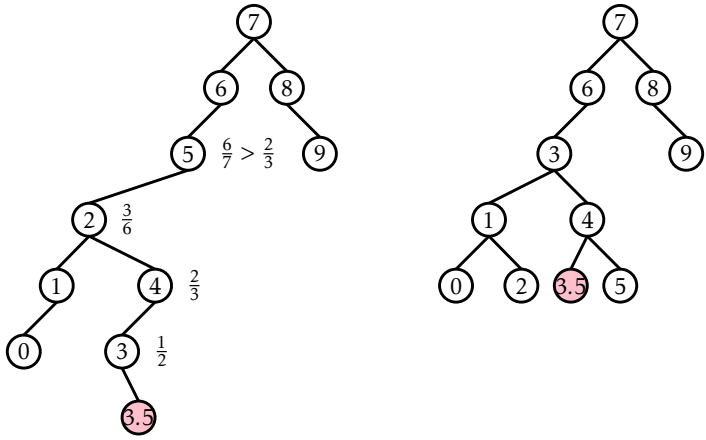

Figure 8.2: Inserting 3.5 into a ScapegoatTree increases its height to 6, which violates (8.1) since $6 > \log_{3/2} 11 \approx 5.914$. A scapegoat is found at the node containing 5.

height of the tree.) Next, we decrement n, but leave q unchanged. Finally, we check if $q > 2n$ and, if so, then we *rebuild the entire tree* into a perfectly balanced binary search tree and set $q = n$.

ScapegoatTree
```
boolean remove(T x) {
  if (super.remove(x)) {
    if (2*n < q) {
      rebuild(r);
      q = n;
    }
    return true;
  }
  return false;
}
```

Again, if we ignore the cost of rebuilding, the running time of the remove(x) operation is proportional to the height of the tree, and is therefore $O(\log n)$.

8.1.1 Analysis of Correctness and Running-Time

In this section, we analyze the correctness and amortized running time of operations on a ScapegoatTree. We first prove the correctness by showing that, when the add(x) operation results in a node that violates Condition (8.1), then we can always find a scapegoat:

Lemma 8.1. *Let u be a node of depth $h > \log_{3/2} q$ in a ScapegoatTree. Then there exists a node w on the path from u to the root such that*

$$\frac{\texttt{size}(w)}{\texttt{size}(\texttt{parent}(w))} > 2/3 \ .$$

Proof. Suppose, for the sake of contradiction, that this is not the case, and

$$\frac{\texttt{size}(w)}{\texttt{size}(\texttt{parent}(w))} \leq 2/3 \ .$$

for all nodes w on the path from u to the root. Denote the path from the root to u as $r = u_0, \ldots, u_h = u$. Then, we have $\texttt{size}(u_0) = n$, $\texttt{size}(u_1) \leq \frac{2}{3}n$, $\texttt{size}(u_2) \leq \frac{4}{9}n$ and, more generally,

$$\texttt{size}(u_i) \leq \left(\frac{2}{3}\right)^i n \ .$$

But this gives a contradiction, since $\texttt{size}(u) \geq 1$, hence

$$1 \leq \texttt{size}(u) \leq \left(\frac{2}{3}\right)^h n < \left(\frac{2}{3}\right)^{\log_{3/2} q} n \leq \left(\frac{2}{3}\right)^{\log_{3/2} n} n = \left(\frac{1}{n}\right) n = 1 \ . \qquad \square$$

Next, we analyze the parts of the running time that are not yet accounted for. There are two parts: The cost of calls to size(u) when searching for scapegoat nodes, and the cost of calls to rebuild(w) when we find a scapegoat w. The cost of calls to size(u) can be related to the cost of calls to rebuild(w), as follows:

Lemma 8.2. *During a call to* add(x) *in a ScapegoatTree, the cost of finding the scapegoat w and rebuilding the subtree rooted at w is $O(\texttt{size}(w))$.*

Proof. The cost of rebuilding the scapegoat node w, once we find it, is $O(\texttt{size}(w))$. When searching for the scapegoat node, we call size(u) on a

sequence of nodes $u_0, \ldots, u_k$ until we find the scapegoat $u_k = w$. However, since u_k is the first node in this sequence that is a scapegoat, we know that

$$\texttt{size}(u_i) < \frac{2}{3}\texttt{size}(u_{i+1})$$

for all $i \in \{0, \ldots, k-2\}$. Therefore, the cost of all calls to $\texttt{size}(u)$ is

$$\begin{aligned} O\left(\sum_{i=0}^{k} \texttt{size}(u_{k-i})\right) &= O\left(\texttt{size}(u_k) + \sum_{i=0}^{k-1} \texttt{size}(u_{k-i-1})\right) \\ &= O\left(\texttt{size}(u_k) + \sum_{i=0}^{k-1} \left(\frac{2}{3}\right)^i \texttt{size}(u_k)\right) \\ &= O\left(\texttt{size}(u_k)\left(1 + \sum_{i=0}^{k-1} \left(\frac{2}{3}\right)^i\right)\right) \\ &= O(\texttt{size}(u_k)) = O(\texttt{size}(w)) \ , \end{aligned}$$

where the last line follows from the fact that the sum is a geometrically decreasing series. □

All that remains is to prove an upper-bound on the cost of all calls to $\texttt{rebuild}(u)$ during a sequence of m operations:

Lemma 8.3. *Starting with an empty* ScapegoatTree *any sequence of* m $\texttt{add}(x)$ *and* $\texttt{remove}(x)$ *operations causes at most* $O(m \log m)$ *time to be used by* $\texttt{rebuild}(u)$ *operations.*

Proof. To prove this, we will use a *credit scheme*. We imagine that each node stores a number of credits. Each credit can pay for some constant, c, units of time spent rebuilding. The scheme gives out a total of $O(m \log m)$ credits and every call to $\texttt{rebuild}(u)$ is paid for with credits stored at u.

During an insertion or deletion, we give one credit to each node on the path to the inserted node, or deleted node, u. In this way we hand out at most $\log_{3/2} q \le \log_{3/2} m$ credits per operation. During a deletion we also store an additional credit "on the side." Thus, in total we give out at most $O(m \log m)$ credits. All that remains is to show that these credits are sufficient to pay for all calls to $\texttt{rebuild}(u)$.

If we call `rebuild(u)` during an insertion, it is because `u` is a scapegoat. Suppose, without loss of generality, that

$$\frac{\texttt{size(u.left)}}{\texttt{size(u)}} > \frac{2}{3} \ .$$

Using the fact that

$$\texttt{size(u)} = 1 + \texttt{size(u.left)} + \texttt{size(u.right)}$$

we deduce that

$$\frac{1}{2}\texttt{size(u.left)} > \texttt{size(u.right)}$$

and therefore

$$\texttt{size(u.left)} - \texttt{size(u.right)} > \frac{1}{2}\texttt{size(u.left)} > \frac{1}{3}\texttt{size(u)} \ .$$

Now, the last time a subtree containing `u` was rebuilt (or when `u` was inserted, if a subtree containing `u` was never rebuilt), we had

$$\texttt{size(u.left)} - \texttt{size(u.right)} \leq 1 \ .$$

Therefore, the number of `add(x)` or `remove(x)` operations that have affected `u.left` or `u.right` since then is at least

$$\frac{1}{3}\texttt{size(u)} - 1 \ .$$

and there are therefore at least this many credits stored at `u` that are available to pay for the $O(\texttt{size(u)})$ time it takes to call `rebuild(u)`.

If we call `rebuild(u)` during a deletion, it is because `q` > 2`n`. In this case, we have `q` − `n` > `n` credits stored "on the side," and we use these to pay for the $O(\texttt{n})$ time it takes to rebuild the root. This completes the proof. □

8.1.2 Summary

The following theorem summarizes the performance of the `ScapegoatTree` data structure:

Theorem 8.1. *A* `ScapegoatTree` *implements the* `SSet` *interface. Ignoring the cost of* `rebuild(u)` *operations, a* `ScapegoatTree` *supports the operations* `add(x)`, `remove(x)`, *and* `find(x)` *in* $O(\log n)$ *time per operation.*

Furthermore, beginning with an empty `ScapegoatTree`, *any sequence of* m `add(x)` *and* `remove(x)` *operations results in a total of* $O(m \log m)$ *time spent during all calls to* `rebuild(u)`.

8.2 Discussion and Exercises

The term *scapegoat tree* is due to Galperin and Rivest [33], who define and analyze these trees. However, the same structure was discovered earlier by Andersson [5, 7], who called them *general balanced trees* since they can have any shape as long as their height is small.

Experimenting with the `ScapegoatTree` implementation will reveal that it is often considerably slower than the other `SSet` implementations in this book. This may be somewhat surprising, since height bound of

$$\log_{3/2} q \approx 1.709 \log n + O(1)$$

is better than the expected length of a search path in a `Skiplist` and not too far from that of a `Treap`. The implementation could be optimized by storing the sizes of subtrees explicitly at each node or by reusing already computed subtree sizes (Exercises 8.5 and 8.6). Even with these optimizations, there will always be sequences of `add(x)` and `delete(x)` operation for which a `ScapegoatTree` takes longer than other `SSet` implementations.

This gap in performance is due to the fact that, unlike the other `SSet` implementations discussed in this book, a `ScapegoatTree` can spend a lot of time restructuring itself. Exercise 8.3 asks you to prove that there are sequences of n operations in which a `ScapegoatTree` will spend on the order of $n \log n$ time in calls to `rebuild(u)`. This is in contrast to other `SSet` implementations discussed in this book, which only make $O(n)$ structural changes during a sequence of n operations. This is, unfortunately, a necessary consequence of the fact that a `ScapegoatTree` does all its restructuring by calls to `rebuild(u)` [20].

Despite their lack of performance, there are applications in which a

ScapegoatTree could be the right choice. This would occur any time there is additional data associated with nodes that cannot be updated in constant time when a rotation is performed, but that can be updated during a rebuild(u) operation. In such cases, the ScapegoatTree and related structures based on partial rebuilding may work. An example of such an application is outlined in Exercise 8.11.

Exercise 8.1. Illustrate the addition of the values 1.5 and then 1.6 on the ScapegoatTree in Figure 8.1.

Exercise 8.2. Illustrate what happens when the sequence $1, 5, 2, 4, 3$ is added to an empty ScapegoatTree, and show where the credits described in the proof of Lemma 8.3 go, and how they are used during this sequence of additions.

Exercise 8.3. Show that, if we start with an empty ScapegoatTree and call add(x) for $x = 1, 2, 3, \ldots, n$, then the total time spent during calls to rebuild(u) is at least $cn \log n$ for some constant $c > 0$.

Exercise 8.4. The ScapegoatTree, as described in this chapter, guarantees that the length of the search path does not exceed $\log_{3/2} q$.

1. Design, analyze, and implement a modified version of ScapegoatTree where the length of the search path does not exceed $\log_b q$, where b is a parameter with $1 < b < 2$.

2. What does your analysis and/or your experiments say about the amortized cost of find(x), add(x) and remove(x) as a function of n and b?

Exercise 8.5. Modify the add(x) method of the ScapegoatTree so that it does not waste any time recomputing the sizes of subtrees that have already been computed. This is possible because, by the time the method wants to compute size(w), it has already computed one of size(w.left) or size(w.right). Compare the performance of your modified implementation with the implementation given here.

Exercise 8.6. Implement a second version of the ScapegoatTree data structure that explicitly stores and maintains the sizes of the subtree

rooted at each node. Compare the performance of the resulting implementation with that of the original ScapegoatTree implementation as well as the implementation from Exercise 8.5.

Exercise 8.7. Reimplement the rebuild(u) method discussed at the beginning of this chapter so that it does not require the use of an array to store the nodes of the subtree being rebuilt. Instead, it should use recursion to first connect the nodes into a linked list and then convert this linked list into a perfectly balanced binary tree. (There are very elegant recursive implementations of both steps.)

Exercise 8.8. Analyze and implement a WeightBalancedTree. This is a tree in which each node u, except the root, maintains the *balance invariant* that size(u) ≤ (2/3)size(u.parent). The add(x) and remove(x) operations are identical to the standard BinarySearchTree operations, except that any time the balance invariant is violated at a node u, the subtree rooted at u.parent is rebuilt. Your analysis should show that operations on a WeightBalancedTree run in $O(\log n)$ amortized time.

Exercise 8.9. Analyze and implement a CountdownTree. In a CountdownTree each node u keeps a *timer* u.t. The add(x) and remove(x) operations are exactly the same as in a standard BinarySearchTree except that, whenever one of these operations affects u's subtree, u.t is decremented. When u.t = 0 the entire subtree rooted at u is rebuilt into a perfectly balanced binary search tree. When a node u is involved in a rebuilding operation (either because u is rebuilt or one of u's ancestors is rebuilt) u.t is reset to size(u)/3.

Your analysis should show that operations on a CountdownTree run in $O(\log n)$ amortized time. (Hint: First show that each node u satisfies some version of a balance invariant.)

Exercise 8.10. Analyze and implement a DynamiteTree. In a DynamiteTree each node u keeps tracks of the size of the subtree rooted at u in a variable u.size. The add(x) and remove(x) operations are exactly the same as in a standard BinarySearchTree except that, whenever one of these operations affects a node u's subtree, u *explodes* with probability 1/u.size. When u explodes, its entire subtree is rebuilt into a perfectly balanced binary search tree.

Your analysis should show that operations on a `DynamiteTree` run in $O(\log n)$ expected time.

Exercise 8.11. Design and implement a Sequence data structure that maintains a sequence (list) of elements. It supports these operations:

- `addAfter(e)`: Add a new element after the element `e` in the sequence. Return the newly added element. (If `e` is null, the new element is added at the beginning of the sequence.)
- `remove(e)`: Remove `e` from the sequence.
- `testBefore(e1,e2)`: return `true` if and only if `e1` comes before `e2` in the sequence.

The first two operations should run in $O(\log n)$ amortized time. The third operation should run in constant time.

The Sequence data structure can be implemented by storing the elements in something like a `ScapegoatTree`, in the same order that they occur in the sequence. To implement `testBefore(e1,e2)` in constant time, each element `e` is labelled with an integer that encodes the path from the root to `e`. In this way, `testBefore(e1,e2)` can be implemented by comparing the labels of `e1` and `e2`.