CMSC 420: Scapegoat Trees

Justin Wyss-Gallifent

November 2, 2022

1 Introduction .................................................. 2
2 Essential Concepts ........................................... 2
   2.1 Balanced Nodes ........................................... 2
   2.2 Scapegoats ............................................... 2
   2.3 Tree Rebuilding ......................................... 3
   2.4 Tree Heights ............................................. 4
3 Scapegoat Tree: Informal .................................... 4
4 Scapegoat Tree: Formal ....................................... 5
   4.1 Trigger Values ............................................ 5
   4.2 Insertion ................................................. 5
   4.3 Deletion .................................................. 7
5 Keeping Track of Size ....................................... 7
6 Height .......................................................... 7
7 Time Complexity ............................................ 8
   7.1 Search Worst-Case ....................................... 8
   7.2 Insertion and Deletion Worst-Case ..................... 8
   7.3 Average Case ........................................... 8
   7.4 Amortized Analysis ..................................... 8
1 Introduction

In simple terms a scapegoat tree is a binary search tree which is only modified if it gets “badly unbalanced”. It does this by rebuilding either a particular subtree or the entire tree.

2 Essential Concepts

2.1 Balanced Nodes

In an ideal binary search tree it would be reasonable to suggest that for any node $u$ each of its subtrees has size about half the size of the subtree rooted at $u$. Or, alternately, that for either $u.child$ we have:

$$\frac{\text{size}(p.child)}{\text{size}(p)} \approx \frac{1}{2}$$

We could relax this a bit and only panic, for example, if one child has too many nodes, meaning this ratio is too large beyond a certain predetermined threshold $\alpha$:

$$\frac{\text{size}(p.child)}{\text{size}(p)} > \alpha \leftarrow \text{OH NO!}$$

2.2 Scapegoats

In order to proceed further we choose the ratio $2/3$ and so we will say that:

Definition 2.2.1. A node $p$ is a scapegoat if it has a child $p.child$ such that:

$$\frac{\text{size}(p.child)}{\text{size}(p)} > \frac{2}{3}$$

Example 2.1. Here is a tree with a scapegoat pointed out. Are there any others?

Intuitively scapegoats are “bad” and we ought to correct them. In reality this is not exactly how it works, though, but we’ll use scapegoats as a way of keeping a tree under control. More on this later.

Note 2.2.1. It is possible to define the scapegoat condition to depend upon a constant rather than $2/3$ and in fact original sources do this, but we present the specific case and encourage you to ponder the generalization.
2.3 Tree Rebuilding

A binary search tree with \( n \) nodes can be rebuilt into an almost perfect binary tree as follows:

We start by doing an inorder traversal to construct a list containing all the keys in the tree. Then we take the node with index \( \lfloor n/2 \rfloor \) and use it as the new root. We then build the left and right subtrees using the lower and higher values respectively. This process continues recursively.

**Example 2.2.** Consider this tree:

\[
\begin{array}{c}
20 \\
10 30 \\
5 15 \\
2 \\
1 3
\end{array}
\]

The inorder traversal yields:

\[
A = [1, 2, 3, 5, 10, 15, 20, 30]
\]

Since \( n = 8 \) we have \( \lfloor 8/2 \rfloor = 4 \) and so \( A[4] = 10 \) will be our root.

On the left we have \([1, 2, 3, 5]\) so we do the same recursively for that subtree.

On the right we have \([15, 20, 30]\) so we do the same recursively for that subtree.

The resulting rebuild tree is then:

\[
\begin{array}{c}
20 \\
10 30 \\
3 20 \\
2 5 15 30 \\
1
\end{array}
\]

That’s nice and balanced!

**Theorem 2.3.1.** A restructured tree with \( n \) nodes will have height \( \lfloor \lg n \rfloor \).

**Proof.** The proof is by strong induction on \( n \) and is left to the reader. \( \square \)
**Example 2.3.** In our example above we have \( n = 8 \) nodes and height \( \lfloor \lg 8 \rfloor = 3 \).

**Theorem 2.3.2.** Restructuring a tree with \( n \) nodes takes time \( \Theta(n) \).

*Proof.* The inorder traversal via recursion can write to an array in \( \Theta(n) \). Rebuilding the tree is a recursive divide-and-conquer algorithm which is also recursive and handles each node once, hence is also \( \Theta(n) \). \( \Box \)

### 2.4 Tree Heights

**Theorem 2.4.1.** A tree with \( n \) nodes will have \( h \geq \lfloor \lg n \rfloor \).

*Proof.* A perfect binary tree (full on every level except the leaf level) with height \( h \) has \( 2^{h+1} - 1 \) nodes and so a general binary tree of height \( h \) has at most that many nodes. Thus we have:

\[
2^{h+1} - 1 \geq n \\
2^h \geq n + 1 \\
h + 1 \geq \lg(n + 1) \\
h \geq \lg(n + 1) - 1
\]

Since \( h \) is an integer it follows that \( h \geq \lfloor \lg n \rfloor \). \( \Box \)

### 3 Scapegoat Tree: Informal

It might be tempting to define a scapegoat tree as a binary search tree in which there are no scapegoats but this is not quite the definition.

Informally a scapegoat tree functions as follows:

1. Search is just as with a standard binary search tree.

2. Insertion is done by first inserting as with a standard binary search tree. Then we check the depth of the inserted node and if it is “too deep” we infer that the inserted node has an ancestor which is a scapegoat so we follow the path back to the root and when we find the scapegoat we rebuild the entire subtree rooted at that node.

3. Deletion is done by first deleting as with a standard binary search tree. If we have deleted too many without insertion then we conclude that we may have an unbalanced tree and we rebuild the entire tree.
4 Scapegoat Tree: Formal

4.1 Trigger Values

Before we get to a more formal description which can be turned into an algorithm we need to describe two value which we will store for the tree. These values will be used to trigger rebuilding operations.

When we initialize our tree to NULL we assign \( n = 0 \) and \( m = 0 \). For our operations then:

1. Insert: We increment both \( n \) and \( m \).
2. Delete: We decremenent \( n \) but leave \( m \) alone.

Observe that \( n \) is simply the number of nodes in the tree and \( m \geq n \) always.

4.2 Insertion

Insert is initially as with a standard binary search tree. If the depth of the inserted node is less than or equal to \( \log_{3/2} m \) then we leave the tree alone. Otherwise we find a scapegoat and rebuild the appropriate subtree.

We find this scapegoat by following the path from the inserted node back to the root. Once we find a scapegoat \( p \) we completely rebuild the subtree rooted at \( p \) and we are done.

Before looking at the related math let’s examine an example.

Example 4.1. Let’s do something simple - we’ll insert 1, 2, 3, 4, ... into an empty tree until we need to rebalance.

The first four inserts are fine (you should check this!) but the fifth insert causes a problem:

At this point the depth of the node with key 5 is \( d = 4 \) and since \( m = 5 \) and \( \log_{3/2} 5 \approx 3.969 \) we have \( d > \log_{3/2} m \) and we must rebuild.
We travel up the tree looking for a scapegoat. The first (you should check this!) we find is at the node \( x \) with key 2 since in that case:

\[
\frac{\text{size}(x.\text{right})}{\text{size}(x)} = \frac{3}{4} > \frac{2}{3}
\]

We rebuild at \( x \) and get the result:

\[
\begin{array}{c}
1 \\
4 \\
3 \quad 5 \\
2
\end{array}
\]

In order for this to work we must prove that if the depth of the inserted node is greater than \( \log_{3/2} m \) then it has an ancestor which is a scapegoat.

First observe:

**Theorem 4.2.1.** For a binary search tree with \( n \) nodes if there is a node \( p \) satisfying:

\[
\text{depth}(p) > \log_{3/2} n
\]

Then either \( p \) or an ancestor of \( p \) is a scapegoat.

**Proof.** Suppose that no node from \( p \) to the root is a scapegoat. This means that for every ancestor \( u \) from \( p \) to the root we have:

\[
\frac{\text{size}(u.\text{child})}{\text{size}(u)} \leq \frac{2}{3}
\]

The root has size \( n \) and so if we follow the search path the root’s child has size at most \((2/3)n\), that node’s child has size at most \((2/3)^2 n\), and so on, and so a node at depth \( i \) will have size at most \((2/3)^i n\).

If \( d \) is the depth of \( p \) then we know:

\[
1 \leq \text{size}(p) \leq (2/3)^d n
\]

Ignoring the middle and solving for \( d \) yields:

\[
d \leq \log_{3/2} n
\]

This is a contradiction. \( QED \)

Now then:

**Theorem 4.2.2.** We must prove that if the depth of the inserted node is greater than \( \log_{3/2} m \) then it has an ancestor which is a scapegoat.
Proof. Suppose our inserted node is \( p \) and depth\((p) > \log_{3/2} m \). Since \( m \geq n \) we have depth\((p) > \log_{3/2} m \geq \log_{3/2} n \) and so we are done by the previous theorem.

\[ \text{QED} \]

In addition we note:

**Theorem 4.2.3.** When we restructure a subtree at the scapegoat, the subtree is shallower.

*Proof.* This proof was incorrect and I have not fixed it yet. \[ \text{QED} \]

### 4.3 Deletion

Delete as with a standard binary search tree. If \( m > 2n \) this suggests that we have deleted too frequently without inserting which suggests an unbalanced tree. We rebuild the entire tree. We also set \( m = n \) once this is done.

### 5 Keeping Track of Size

During the search for a scapegoat we need to compute the size of various subtrees, specifically size\((u)\) and size\((u.child)\) as \( u \) follows the path from the inserted node back to the root.

We know at the start we have \( u = p \) where \( p \) is the inserted node and so we know that size\((u) = 1 \) and size\((u.child) = 0 \). Since we are following this path backwards when we go back to \( u \)'s parent we already know the size of one child (since that was the old \( u \)) so all we need to do is calculate the size of the other child.

In this entire process each node is counted at most once during all possible size calculations (keeping in mind the recursive calculation mentioned above) and so all size calculations needed during a single search for a scapegoat is \( \mathcal{O}(n) \).

### 6 Height

**Theorem 6.0.1.** The height of a scapegoat tree satisfies \( h \leq \log_{3/2} m \).

*Proof.* This is clearly true for an empty tree so we just need to show that if this property is true then it is true if we insert or delete a node. Thus suppose we have a tree with \( n \) nodes for which \( h \leq \log_{3/2} m \).

Suppose we delete a node and don’t restructure the tree. Then we don’t change \( m \) and we certainly don’t increase the height, consequently the property still holds.

Suppose we delete a node and restructure the tree. In such a case the restructuring results in \( h = \lfloor \log n \rfloor \) and since \( m = n \) and \( \lfloor \log n \rfloor < \log_{3/2} m \) the property still holds.
Suppose we insert a node and don’t restructure the tree. The fact that we are not restructuring the tree means that the depth of the inserted node is less than \( \log_{3/2} m \) and so the height of the tree does not change. We know \( m \) increases by 1 and so the property still holds.

Suppose we insert a node and restructure the tree. Since the tree satisfied the property before the insertion we know that it is the inserted node \( p \) with depth \( 1 + \log_{3/2} m \) which caused the issue. When we find the scapegoat node \( u \) and restructure the subtree that subtree is shallower, and since \( m \) increases by 1 the property holds again. \( \text{QED} \)

**Theorem 6.0.2.** The height of a scapegoat tree satisfies \( h(n) = O(\lg n) \).

**Proof.** This follows immediately from the fact that \( m \leq 2n \) and hence:

\[
\log_{3/2} m \leq \log_{3/2}(2n) = \log_{3/2} 2 + \log_{3/2} n
\]

**QED**

7 Time Complexity

7.1 Search Worst-Case

Since the height is \( O(\lg n) \) we know that search is worst-case \( O(\lg n) \).

7.2 Insertion and Deletion Worst-Case

Since the height is \( O(\lg n) \) it might take that long to insert and delete as with a BST and since either insertion and deletion could require a restructure, which is \( O(n) \), we can certainly say that insertion and deletion are worst-case \( O(n) \).

7.3 Average Case

Average case is as usual fairly challenging because of the requirement that we formally define what “average” means.

7.4 Amortized Analysis

Rather than trying to figure out what “average” means, let’s take an amortized approach.

**Theorem 7.4.1.** Starting with an empty tree any sequence of \( M \) insertions and deletions requires \( O(M \lg M) \) time for rebuilding operations.

**Proof.** We prove this using a credit-based approach where one credit is enough to pay for some fixed number of constant time operations.

Each time we insert or delete a node we give out one credit to each node along the path to the inserted or deleted node. In addition when we delete a node we give one credit to an account on the side.
During this sequence of $M$ insertions and deletions there will be at most $M$ nodes in the tree and so the length of a path to a leaf will be at most $\log_{3/2} 2 + \log_{3/2} M$ (we proved earlier that a tree with $n$ nodes has height satisfying $h \leq \log_{3/2} 2 + \log_{3/2} n$) and so we have given $O(M \lg M)$ credits to the nodes and at most $M$ on the side, which is still $O(M \lg M)$ in total.

Now we need to show that there are enough credits for any rebuilding operation that may happen.

Suppose we rebuild during an insertion. Without loss of generality suppose the scapegoat is $u$ and suppose that:

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

It follows that:

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

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

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

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

And therefore we know that currently we have:

$$\text{size}(u.left) - \text{size}(u.right) = \frac{1}{2} \text{size}(u.left) + \frac{1}{2} \text{size}(u.left) - \text{size}(u.right)$$

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

$$> \frac{1}{2} \text{size}(u.left) + 1$$

$$> \frac{1}{3} \text{size}(u) + 1$$

Now let’s consider $u$. If we look back at to the most recent step when either $u$ was inserted or a subtree containing $u$ was rebuilt we know that at that instant we had:

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

So between that instant and now we know that $\text{size}(u.left) - \text{size}(u.right)$ has increased by at least:

$$\frac{1}{3} \text{size}(u)$$
This means that exactly at least this many insert or delete operations must have taken place in \( u \)'s subtrees since that instant and so \( u \) will have earned 1 token for each of those for a total of: \( \frac{1}{4} \text{size}(u) \) tokens.

Since it takes \( \mathcal{O}(n) \) time to restructure a tree with \( n \) nodes this means that our node \( u \) has enough to pay for its own restructuring.

Suppose we rebuild during a deletion. This will happen if \( m > 2n \) and if this is the case it means there have been \( m - n \) deletions without a rebuild. Since \( m - n > 2n - n = n \) this means that our account on the side contains exactly the right amount to do the rebuild.

\[ \boxed{\text{QED}} \]

**Theorem 7.4.2.** The cost of any sequence of \( M \) operations (search, insert, or delete) is \( \mathcal{O}(M \lg M) \).

**Proof.** Since the tree has height \( \mathcal{O}(\lg n) \) it follows that a search is at worst \( \mathcal{O}(\lg M) \) in isolation, so combining them with insertions and deletions has no effect if those insertions and deletions are already \( \mathcal{O}(M \lg M) \).

For insertions and deletions we have proved that the restructuring is \( \mathcal{O}(M \lg M) \). The only tidying up that needs doing is to show that the searches (for insertion and deletion locations and for scapegoats) are no more time intensive.

We leave this to the reader.

\[ \boxed{\text{QED}} \]