CMSC 420: Scapegoat Trees

Justin Wyss-Gallifent

October 1, 2023

1 Introduction ................................................. 2
2 Essential Concepts ........................................... 2
   2.1 Balanced Nodes ........................................... 2
   2.2 Scapegoats ............................................... 2
   2.3 Tree Restructuring ....................................... 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 Important Notes ............................................. 8
6 Keeping Track of Size ...................................... 8
7 Height .................................................... 8
8 Time Complexity ........................................... 10
   8.1 Search Worst-Case ..................................... 10
   8.2 Insertion and Deletion Worst-Case .................... 10
   8.3 Average Case .......................................... 10
   8.4 Amortized Analysis ................................... 10
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 (balanced) 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:

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

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

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

2.2 Scapegoats

In order to proceed further we choose the ratio \( \alpha = 2/3 \) and so we will say that:

**Definition 2.2.1.** A node \( p \) is a *scapegoat* if it has a child \( p.\text{child} \) such that:

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

Here \( \text{size} \) refers to the number of nodes in the subtrees rooted at the nodes in question.

**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 other than $\alpha = 2/3$ and in fact original sources keep the $\alpha$ as a variable, but we present the specific case and encourage you to ponder the generalization.

2.3 Tree Restructuring

A binary search tree with $n$ nodes can be restructured into an almost perfect binary tree as follows:

We start by doing an inorder traversal to construct a sorted 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 binary search tree:

```
  20
 /  \
10   30
 / \  /  \
 5  15 2
 / \  / \
 2 3 1 3
```

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:
That's nice and balanced!

**Note 2.3.1.** It is tempting to think that a restructured tree will be complete, but this is false, as can be seen if we restructure a tree with key list \([1, 2, 3, 4, 5]\).

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

*Proof.* The proof is by strong induction on \(n\) and is left to the reader. \(QED\)

**Example 2.3.** In our example above we have \(n = 8\) nodes and height \([\lg 8] = [3] = 3\).

**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 \(\Theta(n)\). \(QED\)

### 2.4 Tree Heights

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

*Proof.* A perfect binary tree (full on every 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:

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

Since \(h\) is an integer it follows that \(h \geq [\lg n]\). \(QED\)

### 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 towards the root and when we find a scapegoat we rebuild the entire subtree rooted at that scapegoat.

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 proceeding we note that we will keep track of two values, the first being \( n \), the number of nodes currently in the tree, and \( m \), the maximum number of nodes which have been in the tree since the last complete rebuilding. The nature of how \( m \) works will become more clear as we move forward.

Observe that we always have \( n \leq m \) but we will be able to say more soon.

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 \( \log_{\frac{3}{2}} n \) then we leave the tree alone. Otherwise we find a scapegoat and rebuild the subtree rooted at that scapegoat.

We find this scapegoat by following the path from the inserted node back to the root. Once we find a scapegoat we completely rebuild the subtree rooted at that scapegoat 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 but the fifth insert causes a problem, shown below.
At the fifth insert the depth of the node with key 5 is \( d = 4 \) and since \( n = 5 \) and \( \log_{3/2} 5 \approx 3.97 \) we have \( d > \log_{3/2} n \) and we must rebuild.

We travel up the tree looking for a scapegoat. Here are the values until we get one:

We rebuild at the scapegoat and get the result:

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

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

Following the path from the root \( r \) to the node \( p \) we then have:

\[
\begin{align*}
n &= \text{size}(r) \\
&\geq \frac{3}{2} \text{size}(r.\text{child}) \\
&\geq \left(\frac{3}{2}\right)^2 \text{size}(r.\text{child}.\text{child}) \\
&\geq \ldots \\
&\geq \left(\frac{3}{2}\right)^{\text{depth}(p)} \text{size}(p) \\
&\geq \left(\frac{3}{2}\right)^{\text{depth}(p)} \tag{1}
\end{align*}
\]

From this we get:

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

This is a contradiction. \( \Box \)

### 4.3 Deletion

Delete as with a standard binary search tree. However if we get find upon deletion that:

\[
n \leq \frac{2}{3} m
\]

then we rebuild the entire tree and set \( m = n \).
5 Important Notes

There are some important observations at this point:

(a) When we insert, the value of \( n \) increases and perhaps the value of \( m \) too. Either way the ratio \( n/m \) will not decrease.

(b) When we delete, the value of \( n \) decreases and as such the ratio \( n/m \) decreases. However this ratio will never get below \( 2/3 \) because a rebuild will set it back to 1.

(c) As a consequence of (a) and (b) we will always have:

\[
\frac{2}{3} < \frac{n}{m} \leq 1
\]

6 Keeping Track of Size

During the search for a scapegoat we need to compute the size of various subtrees, specifically \( \text{size}(u) \) and \( \text{size}(u.\text{child}) \) 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 \( \text{size}(u) = 1 \) and \( \text{size}(u.\text{child}) = 0 \). 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 \( O(n) \).

7 Height

**Theorem 7.0.1.** The height of a scapegoat tree satisfies \( h \leq \left\lfloor \log_{3/2} n \right\rfloor \).

**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 \( h \leq \left\lfloor \log_{3/2} n \right\rfloor \). We break the situation into four cases:

- Suppose we delete a node and do not need to restructure the entire tree. Recall this means that \( n \) decreases and since the height certainly does not increase the property still holds.

- Suppose we delete a node and do need to restructure the entire tree. In such a case the restructuring results in \( h = \lfloor \lg n \rfloor \) and since we have \( h = \lfloor \lg n \rfloor < \left\lfloor \log_{3/2} n \right\rfloor \) the property still holds.
• Suppose we insert a node and do not need to restructure the tree. The fact that we do not need to restructure the tree means that the depth of the inserted node is less than or equal to $\left\lfloor \frac{\log_3 n}{2} \right\rfloor$ and the original height of the tree was less than or equal to $\left\lfloor \frac{\log_3 n}{2} \right\rfloor$, thus the tree still satisfies the height requirement.

• Suppose we insert a node and do need to restructure at a scapegoat. The fact that we need to restructure at a scapegoat means that the inserted node is the unique node causing the height of the tree to be $\left\lfloor \frac{\log_3 n}{2} \right\rfloor + 1$.

We will show that restructuring the subtree rooted at the scapegoat results in the subtree rooted at the scapegoat being shallower than it was before insertion. This will ensure that after restructuring the tree has a height no more than $\left\lfloor \frac{\log_3 n}{2} \right\rfloor$.

Let $x$ be the scapegoat, let $T$ be the subtree rooted at $x$ before insertion, let $T'$ be the subtree rooted at $x$ after insertion, and let $T''$ be the subtree rooted at $x$ after restructuring. Note that $\text{size}(T'') = \text{size}(T')$.

Since $T$ is not a perfect binary tree (otherwise it would be perfectly balanced and $x$ would not be a scapegoat) we know that:

$$\text{size}(T) < 2^{\text{height}(T) + 1} - 1$$

It follows that since $T'$ just has one more node:

$$\text{size}(T') \leq 2^{\text{height}(T) + 1} - 1$$

When we restucture $T'$ we then have:

$$\text{height}(T'') \leq \left\lfloor \log (\text{size}(T'')) \right\rfloor = \left\lfloor \log (\text{size}(T')) \right\rfloor \leq \left\lfloor \log \left(2^{\text{height}(T) + 1} - 1\right) \right\rfloor < \left\lfloor \log \left(2^{\text{height}(T) + 1}\right) \right\rfloor = \text{height}(T) + 1$$

Since $\text{height}(T'')$ is an integer it follows that $\text{height}(T'') \leq \text{height}(T)$ and so the newly restructured subtree is no higher than the original subtree and thus the newly restructured tree is no higher than the original tree and so the property still holds.

$\text{QED}$

**Theorem 7.0.2.** The height of a scapegoat tree satisfies $h(n) = \mathcal{O}(\log n)$.

**Proof.** This follows immediately from the previous theorem. $\text{QED}$
8 Time Complexity

8.1 Search Worst-Case
Since the height is $O(\log n)$ we know that search is worst-case $O(\log n)$.

8.2 Insertion and Deletion Worst-Case
Since the height is $O(\log n)$ 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 $O(n)$.

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

8.4 Amortized Analysis
Rather than trying to figure out what “average” means, let’s take an amortized approach. This approach is a tweak of the token method we studied earlier.

**Theorem 8.4.1.** Starting with an empty tree, any sequence of $Q$ insertions and deletions requires $O(Q \log Q)$ time for rebuilding operations, thus averaging $O(\log Q)$ per rebuilding operation.

**Proof.** Each time we insert or delete a node we give out one token to each node along the path to the inserted or deleted node. In addition when we delete a node we give one token to a side-account.

Since the height of the tree satisfies $O(\log Q)$, by the time we have performed these $Q$ insertions and deletions we will have given $O(Q \log Q)$ tokens to the nodes and $O(Q)$ tokens to the side-account for a total of $O(Q \log Q)$.

Now we need to show that there are enough credits for any rebuilding operation that may happen. Rebuilds may happen during insertion or deletion:

- 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:
\[
size(u.left) > \frac{2}{3} \size(u) \\
size(u.left) > \frac{2}{3} \left(1 + \size(u.left) + \size(u.right)\right) \\
\frac{1}{3} \size(u.left) > \frac{2}{3} + \frac{2}{3} \size(u.right) \\
\frac{1}{2} \size(u.left) > 1 + \size(u.right)
\]

And therefore we know that currently we have:

\[
size(u.left) - size(u.right) = \frac{1}{2} \size(u.left) + \frac{1}{2} \size(u.left) - size(u.right) \\
> \frac{1}{2} \size(u.left) + 1 + size(u.right) - size(u.right) \\
> \frac{1}{2} \size(u.left) + 1 \\
> \frac{1}{3} \size(u) + 1
\]

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:

\[
size(u.left) - size(u.right) \leq 1
\]

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

\[
\frac{1}{3} \size(u)
\]

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 at least \(\frac{1}{3} \size(u)\) tokens.

Since it takes \(O(\size(u))\) time to restructure \(u\’s\) subtree it has enough tokens to pay for that restructuring.

• Suppose we rebuild during a deletion. This will happen if \(n \leq (2/3)m\) and if this is the case it means there have been at least \((1/3)m\) deletions without a rebuild and as such our side-account contains at least \((1/3)m = O(m)\) tokens which is certainly sufficient to rebuild our tree containing \(n < m\) nodes as rebuilding takes \(O(n)\) time.
Theorem 8.4.2. The cost of any sequence of $Q$ operations (search, insert, or delete) is $O(Q \lg Q)$.

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

For insertions and deletions we have proved that the restructuring is $O(Q \lg Q)$. 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. 

$\qed$