CMSC 420: Scapegoat Trees

Justin Wyss-Gallifent

February 29, 2024

1 Introduction 2
2 Essential Concepts 2
2.1 Balanced Nodes 2
2.2 Scapegoats 2
2.3 (Sub)Tree Rebuilding 3
2.4 Trigger Values 4
3 Scapegoat Trees: Process 4
3.1 Informal Process 4
3.2 Formal Process 5
3.3 Insertion 5
3.4 Deletion 6
4 Scapegoat Trees: Math 6
4.1 Tree Rebuilding 6
4.2 Scapegoat Existence 7
4.3 Height Bound 8
5 Keeping Track of Size 9
6 Time Complexity 9
6.1 Search Worst-Case 9
6.2 Insertion and Deletion Worst-Case 9
6.3 Amortized Analysis 9

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:

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

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 $\frac{1}{2} < \alpha < 1$:

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

2.2 Scapegoats

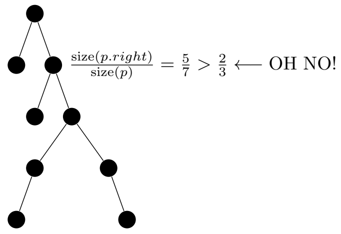
In order to proceed further we choose the ratio $\alpha = 2/3$ and so we will say that:

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

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

Here *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?


$\frac{\text{size}(p.right)}{\text{size}(p)} = \frac{5}{7} > \frac{2}{3} \longleftarrow$ OH NO!

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.

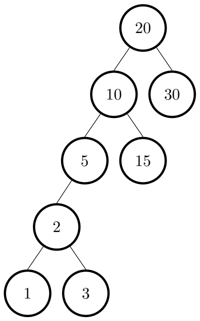
Note 2.2.1. It is possible to define the scapegoat condition to depend upon a constant other than $\alpha = 2/3$ and in fact original sources keep the α as a variable, but we present the specific case and encourage you to ponder the generalization.

2.3 (Sub)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 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:



The inorder traversal yields:

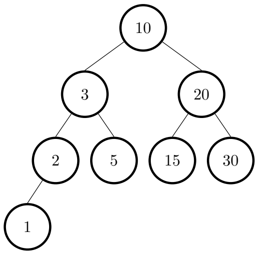
$$A = [1, 2, 3, 5, 10, 15, 20, 30]$$

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

On the left we have $AL = [1, 2, 3, 5]$ so we do the same recursively for that subtree with $AL[2] = 3$ for the root.

On the right we have $AR = [15, 20, 30]$ so we do the same recursively for that subtree. with $AR[1] = 20$ for the root.

The resulting rebuild tree is then:



> That's nice and balanced!

Note 2.3.1. It is tempting to think that a rebuilt tree will be complete, but this is false, as can be seen if we rebuild a tree with key list $[1, 2, 3, 4, 5]$.

> **Example 2.3.** In our example above we have $n = 8$ nodes and height $\lfloor \lg 8 \rfloor = \lfloor 3 \rfloor = 3$.

2.4 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 rebuilding and its relationship to m 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.

3 Scapegoat Trees: Process

3.1 Informal Process

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.

3.2 Formal Process

More formally, motivation behind how we respond to insertion and deletion is based upon the following pair of goals:

Goal 1: The height should stay at $\left\lfloor \log_{3/2} n \right\rfloor + 1$ or less. To achieve this, when inserting we will only panic and do something drastic if the inserted node is at a depth of more than $\log_{3/2} n$.

Goal 2: The tree should be "balanced" whenever possible. To achieve this, when deleting we will only panic if we have deleted "too many" which might suggest that the tree is unbalanced.

3.3 Insertion

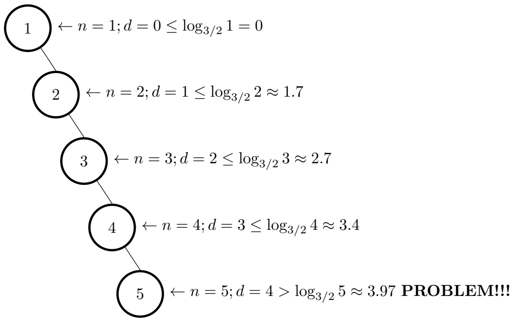
Insert is then initially as with a standard binary search tree. Then if the depth of the inserted node has $d > \log_{3/2} n$ we find a scapegoat and rebuild the subtree rooted at that scapegoat.

We find this scapegoat by following the path from the inserted node back to the root. Once we find a scapegoat (which it turns out must exist) we completely rebuild the subtree rooted at that scapegoat and we are done.

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

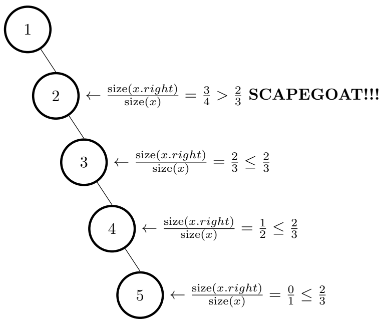
Example 3.1. Let's do something simple - we'll insert $1, 2, 3, 4, ...$ into an empty tree until we need to rebalance.

The first four inserts are fine but the fifth insert causes a problem, shown below.

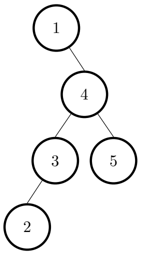

1
$\leftarrow n = 1; d = 0 \le \log_{3/2} 1 = 0$
2
$\leftarrow n = 2; d = 1 \le \log_{3/2} 2 \approx 1.7$
3
$\leftarrow n = 3; d = 2 \le \log_{3/2} 3 \approx 2.7$
4
$\leftarrow n = 4; d = 3 \le \log_{3/2} 4 \approx 3.4$
5
$\leftarrow n = 5; d = 4 > \log_{3/2} 5 \approx 3.97$ **PROBLEM!!!**

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:


1
2
$\leftarrow \frac{\text{size}(x.right)}{\text{size}(x)} = \frac{3}{4} > \frac{2}{3}$ SCAPEGOAT!!!
3
$\leftarrow \frac{\text{size}(x.right)}{\text{size}(x)} = \frac{2}{3} \leq \frac{2}{3}$
4
$\leftarrow \frac{\text{size}(x.right)}{\text{size}(x)} = \frac{1}{2} \leq \frac{2}{3}$
5
$\leftarrow \frac{\text{size}(x.right)}{\text{size}(x)} = \frac{0}{1} \leq \frac{2}{3}$

We rebuild at the scapegoat and get the result:



3.4 Deletion

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

$$n < \frac{2}{3}m$$

then we rebuild the entire tree and set $m = n$.

4 Scapegoat Trees: Math

There are several math results which we need to iron out to ensure that our process achieves our goals:

4.1 Tree Rebuilding

Theorem 4.1.1. Rebuilding 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 $\Theta(n)$. $\mathcal{QED}$

Theorem 4.1.2. A rebuilt tree with n nodes will have height equal to $\lfloor \lg n \rfloor$.

Proof. The proof is by strong induction on n and is left to the reader.

$\mathcal{QED}$

4.2 Scapegoat Existence

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 suppose p is a freshly inserted node satisfying:

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

Then 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 down to p we have:

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

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

$$\begin{aligned} n &= \text{size}(r) \\ &\geq \frac{3}{2}\text{size}(r.child) \\ &\geq \left(\frac{3}{2}\right)^2 \text{size}(r.child.child) \\ &\geq ... \\ &\geq \left(\frac{3}{2}\right)^{\text{depth}(p)} \text{size}(p) = \left(\frac{3}{2}\right)^{\text{depth}(p)} \qquad (1) \end{aligned}$$

From this we get the contradiction:

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

QED

4.3 Height Bound

Theorem 4.3.1. The height of a scapegoat tree with $n \geq 1$ satisfies:

$$h \leq \left\lfloor \log_{3/2} n \right\rfloor + 1$$

Proof. This can be found in the original paper by Galperin and Rivest. Their proof is not difficult but is rather convoluted and I am going to refrain trying to make it more friendly. The basic underlying idea, however, proceeds as follows:

Definition 4.3.1. We say a tree is *height-balanced* if $h \leq \lfloor \log_{3/2} n \rfloor$ and *loosely height-balanced* if $h \leq \lfloor \log_{3/2} n \rfloor + 1$. An empty tree is said to be height-balanced by default.

The goal is to then prove that a scapegoat tree is always loosely height-balanced. We do this via the following steps in which n and n' represent the number of nodes before and after an operation.

(a) Observe that an empty tree is height-balanced by definition.

(b) Prove that an insert or delete with $\lfloor \log_{3/2} n' \rfloor = \lfloor \log_{3/2} n \rfloor$ will preserve the balanced nature of the tree.

(c) Prove that an insert with $\lfloor \log_{3/2} n' \rfloor \neq \lfloor \log_{3/2} n \rfloor$ results in the tree being height-balanced even if it was loosely height-balanced before.

(d) Prove that a delete with $\lfloor \log_{3/2} n' \rfloor \neq \lfloor \log_{3/2} n \rfloor$ can only occur on a height-balanced tree and results in the tree being loosely height-balanced.

(e) Observe that we start with an empty tree (height balanced) and then:

- If we insert or delete with a height-balanced tree and if $\lfloor \log_{3/2} n' \rfloor = \lfloor \log_{3/2} n \rfloor$ then the result is height-balanced.
- If we insert or delete with a loosely height-balanced tree and if $\lfloor \log_{3/2} n' \rfloor = \lfloor \log_{3/2} n \rfloor$ then the result is loosely height-balanced.
- If we insert with a height-balanced tree and if $\lfloor \log_{3/2} n' \rfloor \neq \lfloor \log_{3/2} n \rfloor$then then the result is height-balanced.
- If we insert with a loosely height-balanced tree and if $\lfloor \log_{3/2} n' \rfloor \neq \lfloor \log_{3/2} n \rfloor$then then the result is height-balanced.
- If we delete and $\lfloor \log_{3/2} n' \rfloor \neq \lfloor \log_{3/2} n \rfloor$ then it must be with a height-balanced tree and the result is loosely height-balanced.

(f) We now know the tree is always loosely height-balanced (sometimes even height-balanced) and we are done.

QED

Corollary 4.3.1. The height of a scapegoat tree satisfies $h(n) = \mathcal{O}(\lg n)$.

Proof. This follows immediately from the previous theorem. $\mathcal{QED}$

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 Time Complexity

We can now comment on the time complexity of some operations.

6.1 Search Worst-Case

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

6.2 Insertion and Deletion Worst-Case

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

6.3 Amortized Analysis

However this is precisely where we need to stop and consider that the worst-case $\mathcal{O}(n)$ we got in the last subsection is not necessarily the best answer. The reason for this is that even though a single operation may take time $\mathcal{O}(n)$ such operations do not happen repeatedly. For example we will never do a compete tree rebuild followed immediately by another complete tree rebuild.

Consequently it's insightful to take an amortized analysis view and ask:

In the worst case if start with an empty tree and perform k operations (insertions or deletions) what can we say about the average per-operation worst-case time?

It turns out we have the following:

Theorem 6.3.1. Starting with an empty tree, any sequence of k insertions and deletions has an amortized cost of $\mathcal{O}(\log k)$ per operation which includes rebuilding.

Proof. We prove this by putting specific costs on each operation which accurately reflect the asymptotic time complexity of those operations. Using specific costs makes the argument easier to follow.

For simplicity we assume the following:

- It costs $d + 1$ to insert a node at depth d.
- It costs $d + 1$ to delete a node at depth d. If a replacement node is used then just consider deletion of the replacement node.
- It costs n to rebuild a (sub)tree with n nodes.

We use the token method with a tweak. Instead of having one account we have three, and one of those three is actually distributed as an account for each node in the tree.

Before proceeding note that if we perform a sequence of k insertions and deletions the maximum number of nodes in the tree at any instant is k.

(a) Each time we insert or delete a node we put $\left\lfloor \log_{3/2} k \right\rfloor + 3$ tokens into what we'll call the ID account, standing for "Insertion Deletion". This account will be used to cover the cost of insertion or deletion not counting any rebuilding. Since the height of a scapegoat tree satisfies $h \leq \left\lfloor \log_{3/2} k \right\rfloor + 1$ we know that before any rebuilding:

  - The depth of a deleted key will certainly be less or equal to than $\left\lfloor \log_{3/2} k \right\rfloor + 1$ so the cost will be less or equal to than $\left\lfloor \log_{3/2} k \right\rfloor + 2$.
  - The depth of an inserted key will certainly be less than or equal to $\left\lfloor \log_{3/2} k \right\rfloor + 2$ so the cost will be less or equal to than $\left\lfloor \log_{3/2} k \right\rfloor + 3$.

  It is then clear that the amount we put into the ID account for each insertion or deletion will cover the cost of that insertion or deletion, not counting any rebuilding.

  Observe that this cost is $\left\lfloor \log_{3/2} k \right\rfloor + 3$ tokens per operation.

(b) Each time we delete a node we give 3 tokens into what we'll call the DR account, standing for "Deletion Rebuild".

  We must show that this account is sufficient to cover the cost of an rebuilding due to deletion. Suppose now that we rebuild during a deletion and our tree has values n and m respectively. This will happen if $n < (2/3)m$ 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 $3((1/3)m) = m$

tokens which is certainly sufficient to rebuild our tree containing $n < m$ nodes.

Observe that this cost is 3 tokens per operation.

(c) Each time we insert or delete a node we put 3 tokens into each node on the path from the root to the inserted or deleted node. We'll call this the (distributed) IR account, standing for "Insertion Rebuild". to each node on the path from the root to the inserted or deleted node.

We must show that this account is sufficient to cover the cost of an rebuilding due to insertion. More specifically we will show that if we rebuild at a scapegoat u then the node u has enough tokens to cover its own subtree rebuilding.

Without loss of generality assume:

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

It follows that:

$$\begin{aligned}
\text{size}(u.left) &> \frac{2}{3}\text{size}(u) \\
\text{size}(u.left) &> \frac{2}{3}(1 + \text{size}(u.left) + \text{size}(u.right)) \\
\frac{1}{3}\,\text{size}(u.left) &> \frac{2}{3} + \frac{2}{3}\,\text{size}(u.right) \\
\frac{1}{2}\,\text{size}(u.left) &> 1 + \text{size}(u.right)
\end{aligned}$$

And therefore we know that currently we have:

$$\begin{aligned}
\text{size}(u.left) - \text{size}(u.right) &= \frac{1}{2}\,\text{size}(u.left) + \frac{1}{2}\,\text{size}(u.left) - \text{size}(u.right) \\
&> \frac{1}{2}\,\text{size}(u.left) + 1 + \text{size}(u.right) - \text{size}(u.right) \\
&> \frac{1}{2}\,\text{size}(u.left) + 1 \\
&> \frac{1}{3}\,\text{size}(u) + 1
\end{aligned}$$

Now let's consider u. We claim it has enough tokens to cover its own subtree rebuild.

If we look back at to the most recent step either when 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 betweeen 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 three token for each of those for a total of at least $3\left(\frac{1}{3}\text{size}(u)\right) = \text{size}(u)$ tokens and this is sufficient for rebuilding its subtree.

Observe that due to the height bound this cost is less than or equal to $3\left(\left\lfloor \log_{3/2} k \right\rfloor + 3\right)$ tokens per operation because this is the maximum number of nodes which can exist in a path from the root to a leaf after an insertion on a tree with the maximum height bound.

We see the per-operation cost is certainly $\mathcal{O}(\lg k)$ tokens per operation.

$\mathcal{QED}$

Example 6.1. Just to make sure we see how this works, here is a very simple example of how the accounts with no rebuilding. Let's insert the keys $1, 2, 3, 4$ into an empty scapegoat tree and then delete 4.

ID Account:

Since there are $k = 5$ operations so each insert and delete gives $\left\lfloor \log_{3/2} 5 \right\rfloor + 3 = 6$ tokens to the ID account and each insert and delete takes an amount equal to $d + 1$ where d is the depth of the node. Here is the ID account statement:

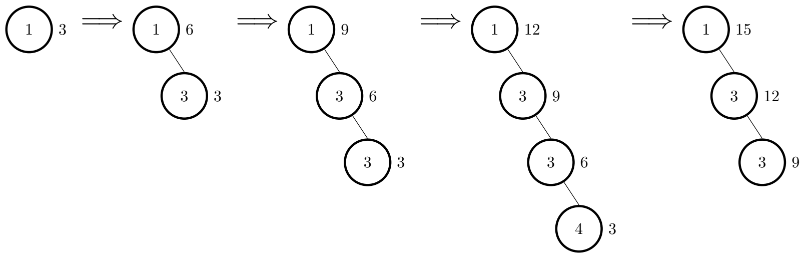
Operation	Deposit	Depth	Cost	Balance
Insert 1	6	0	1	5
Insert 2	6	1	2	9
Insert 3	6	2	3	12
Insert 4	6	3	4	14
Delete 4	6	3	4	16

DR Account:

Since we only deleted one node the DR account has $1(3) = 3$ tokens.

IR Account:

Here is the progression of the tree and next to each node the account total for that node at the end:


1 3
1 6
3 3
1 9
3 6
3 3
1 12
3 9
3 6
4 3
1 15
3 12
3 9