CMSC 420: Lecture 12
Scapegoat Trees

Overview: In this lecture, we will discuss an ordered (tree-based) dictionary data structure that is efficient, $O(\log n)$ time, but in an amortized sense. Recall that this means that, over a series of operations, the average cost per operation is $O(\log n)$, even though the cost of any individual operation can be much higher. The worst case for find will be $O(\log n)$, but the worst case for insert and delete can both be as high as $O(n)$.

This data structure has the unusual name of *scapegoat tree*. The idea underlying this data structure was due initially to Arne Anderson (of AA trees) in 1989. This idea was rediscovered by Igal Galperin and Ron Rivest in 1993, who made some refinements and gave it the name "scapegoat tree," which we will explain below. (By the way, Rivest is quite famous in cryptography. The "R" in the RSA public crypto system comes from his name.)

Wreck it Ralph: While amortized data structures often interesting in their own right, there is a particular motivation for studying the scapegoat tree. So far, all of the binary search trees that we have studied achieve balance "incrementally" through the use of rotation operation. Scapegoat trees are unique in that they do not rely on rotation. This is significant because there exist binary trees that cannot be balanced through the use of rotations. (A prime example is the kd-tree data structure for storing geometric objects.) The scapegoat tree achieves good balance in a very different way—when a subtree is imbalanced, it is tossed out and rebuilt from scratch (or "wrecked and fixed").

Below, we will discuss the details of how the scapegoat tree works. Here is a high-level overview. A scapegoat tree is a binary search tree, which does not need to store any additional information in the nodes, other than the key, value, and left and right child pointers. Nonetheless, the height of a tree with n nodes will always be $O(\log n)$. (Note that this is not the case for splay trees, whose height can grow to as high as $O(n)$.)

In an ideally balanced tree, each child has almost nearly exactly half as many nodes as its parent. Define the *size* of a node to be the total number of nodes in its subtree. A node is said to be *weight balanced* if the sizes of its two subtrees are within a constant fraction of either other (e.g., split no worse than $\frac{1}{3} \cdots \frac{2}{3}$). The scapegoat tree attempts to maintain this property. Insertion and deletions work roughly as follows.

Insertion:

- The key is first inserted just as in a standard (unbalanced) binary tree
- We monitor the depth of the inserted node after each insertion, and if it is too high, we can infer there must be at least one node on the search path that has poor weight balance (that is, its left and right children have very different sizes).
- In such a case, we find such a node, called the *scapegoat*,[1] and we completely rebuild the subtree rooted at this node so that it is perfectly balanced.

Deletion:

- The key is first deleted just as in a standard (unbalanced) binary tree
- Once the number of deletions performed is sufficiently large relative to the entire tree size, rebuild the entire tree so it is perfectly balanced.

[1] The colorful term "scapegoat" refers to an individual who is assigned the blame when something goes wrong. In this case, the unbalanced node takes the blame for the tree's height being too great.

Why the Asymmetry? You might wonder why there is a notable asymmetry between the rebuilding rules for insertion and deletion. The existence of a single very deep node is proof that a tree is out of balance. Thus, for insertion, we can use the fact that the inserted node is too deep to trigger rebuilding. However, observe that the converse does not work for deletion.

The natural counterpart would be "if the depth of the leaf node containing the deleted key is too small, then trigger a rebuilding operation." But the fact that a single node has a low depth, does not imply that the rest of the tree is out of balance. (It may just be that a single search path has low depth, but the rest of the tree is perfectly balanced.)

Could we instead apply the deletion rebuilding trigger to work for insertion? Again, this will not work. The natural counterpart would be, "given a newly rebuild tree with n keys, we will rebuild it after inserting roughly $n/2$ new keys." However, if we are very unlucky, all these keys may fall along a single search path, and the tree's height would be as bad as $O((\log n) + n/2) = O(n)$, and this is unacceptably high.

How to Rebuild a Subtree: Before getting to the details of how the scapegoat tree works, let's consider the basic operation that is needed to maintain balance, namely rebuilding subtrees into balanced form. We shall see that if the subtree contains n keys, this operation can be performed in $O(n)$ time (see Fig. 1). Letting `p` denote the root node of the subtree to rebuild, call this function `rebuild(p)`:

- Perform an inorder traversal of `p`'s subtree, copying the keys to an array `A[0..k-1]`, where `k` denotes the number of nodes in this subtree. Note that the elements of A are sorted.
- Invoke the following recursive subtree-building function: `buildSubtree(A)`
  - Let `k = A.length`.
  - If `k == 0`, return an empty tree, that is, `null`.
  - Otherwise, let `x` be the median key, that is, `A[j]`, where `j` $= \lfloor$`k/2`$\rfloor$. Recursively invoke `L = buildSubtree(A[0..j-1])` and `R = buildSubtree(A[j+1..k-1])`. Finally, create an internal node containing x with left subtree L and right subtree R. Return a pointer to x.

Note that if `A` is implemented as a Java `ArrayList`, there is a handy function called `subList` for performing the above splits, without the need to explicitly copy elements into new arrays. The `buildSubtree` function is given in the code block below.

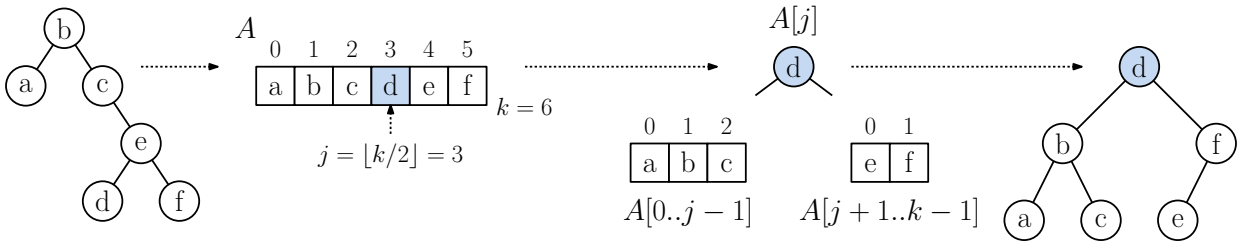

Fig. 1: Recursively building a balanced tree from a sorted array of keys.

Ignoring the recursive calls, we spend $O(1)$ time in each recursive call, so the overall time is proportional to the size of the tree, which is k, so the total time is $O(k)$.

Building a Balanced Tree from an Array

```
Node buildSubtree(Key[] A) {                    // A is a sorted array of keys
    k = A.length
    if (k == 0) return null                     // empty array
    else {
        j = floor(k/2)                          // median of the array
        Node p = new Node(A[j])                 // ...this is the root
        p.left = buildSubtree(A[0..j-1])        // build left subtree recursively
        p.right = buildSubtree(A[j+1..k-1])     // build right subtree recursively
        return p                                // return root of the subtree
    }
}
```

Scapegoat Tree Operations: In addition to the nodes themselves, the scapegoat tree maintains two integer values. The first, denoted by n, is just the actual number of keys in the tree. The second, denoted by m, is a special parameter, which is used to trigger the event of rebuilding the entire tree.

In particular, whenever we insert a key, we increment m, but whenever we delete a key we do not decrement m. Thus, $m \geq n$. The difference $m - n$ intuitively represents the number of deletions. When we reach a point where $m > 2n$ (or equivalently $m - n > n$) we can infer that the number of deletions exceeds the number of keys remaining in the tree. In this case, we rebuild the entire tree in balanced form.

We are now in a position to describe how to perform the dictionary operations for a scapegoat tree.

Initialization: $n \leftarrow m \leftarrow 0$ and the root is initialized to `null`.

`find(Key x):` The find operation is performed exactly as in a standard (unbalanced) binary search tree. We will show that the height of the tree never exceeds $\log_{3/2} n \approx 1.7 \lg n$, so this is guaranteed to run in $O(\log n)$ time.

`delete(Key x):` This operates exactly the same as deletion in a standard binary search tree. After deleting the node, decrement n (but do not change m). If $m > 2n$, rebuild the entire tree by invoking `rebuild(root)`, and set $m \leftarrow n$.

`insert(Key x, Value v):` First, increment both n and m. We start by applying the insertion process a standard (unbalanced) binary search tree. As we are tracing the search path to the insertion point, keep track of our depth in the tree. (Recall that depth is the number of edges to root.) If the depth of the inserted node exceeds $\log_{3/2} m$ then we trigger a *rebuilding event*. This involves the following:

- Walk back up along the insertion search path towards the root. Let `p` be the current node that is visited, and let `p.child` be the child of `p` that lies on the search path.
- Let `size(p)` denote the *size* of the subtree rooted at `p`, that is, the number of nodes in this subtree.
- If
$$\frac{\texttt{size(p.child)}}{\texttt{size(p)}} > \frac{2}{3},$$
then rebuild the subtree rooted at `p` by invoking `rebuild(p)`. The node `p` is the *scapegoat*. After the rebuild is done, we terminate the insertion process. Even if `p`

has an ancestor that satisfies the scapegout condition, we do not do a second rebuild as part of this insertion.

An example of insertion is shown in Fig. 2. After inserting 5, the tree has $n = 11$ nodes. The newly inserted node is at depth 6, and since $6 > \log_{3/2} 11$ (which is approximately 5.9), we trigger the rebuilding event. We walk back up the search path. We find node 9 whose size is 7, but the child on the search path has size 6, and $6/7 > 2/3$, so we invoke rebuild on the node containing 9.

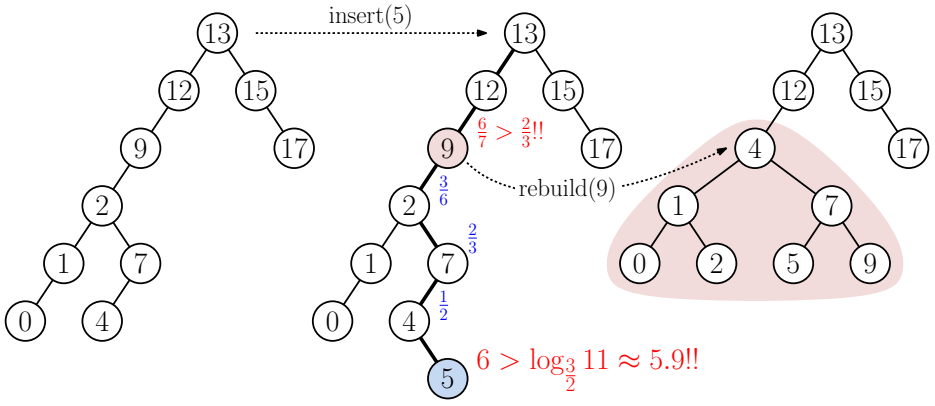

Fig. 2: Inserting a key into a scapegoat tree, which triggers a rebuilding event. The node containing 9 is the first scapegoat candidate encountered while backtracking up the search path and is rebuilt.

Must there be a scapegoat? The fact that a child has over 2/3 of the nodes of the entire subtree intuitively means that this subtree has (roughly) more than twice as many nodes as its sibling. We call such a node on the search path a *scapegoat candidate*. A short way of summarize the above process is "rebuild the scapegoat candidate that is closest to the insertion point.'

You might wonder whether we will necessarily encounter an scapegoat candidate when we trace back along the search path. The following lemma shows that this is always the case.

Lemma: Given a binary search tree of n nodes, if there exists a node p such that $\text{depth}(p) > \log_{3/2} n$, then p has an ancestor (possibly p itself) that is a scapegoat candidate.

Proof: The proof is by contradiction. Suppose to the contrary that no node from p to the root is a scapegoat candidate. This means that for every ancestor node u from p to the root, we have $\text{size}(u.\text{child}) \leq \frac{2}{3} \cdot \text{size}(u)$.

We know that the root has a size of n. Therefore, its child on the search path has size at most $(2/3)n$, its grandchild has size at most $(2/3)((2/3)n) = (4/9)n$, and generally the node at depth i along the search path as size at most $(2/3)^i n$.

Let d denote the depth of p. We know what its subtree rooted at p must have at least one node (namely p itself), and therefore

$$1 \;\leq\; \text{size}(p) \;\leq\; \left(\frac{2}{3}\right)^d n.$$

Solving for d, we have

$$\left(\frac{3}{2}\right)^d \;\leq\; n \quad\Longrightarrow\quad d \;\leq\; \log_{3/2} n.$$

However, this violates our hypothesis that p's depth exceeds $\log_{3/2} n$, yielding the desired contradiction.

Recall that $m \geq n$, and so if a rebuilding event is triggered, the insertion depth is at least $\log_{3/2} m$, which means that it is at depth at least $\log_{3/2} n$. Therefore, by the above lemma, there must be a scapegoat candidate along the search path.

The Sizeless Size? No, this is not the answer to some Zen koan. We mentioned earlier that the scapegoat tree does not store any information in the nodes other than the key, value, and left and right child pointers. So how can we compute `size(u)` for a node `u` during the insertion process, without this information? There is very clever trick for doing this on the fly.

Since we are doing this as we back up the search path, we may assume that we already know the value of $s' =$ `size(u.child)`, where this is the child that lies along the insertion search path. So, to compute `size(u)`, it suffices to compute the size of `u`'s other child. To do this, we perform a traversal of this child's subtree to determine its size s''. Given this, we have $\texttt{size}(u) = 1 + s' + s''$, where the $+1$ counts the node `u` itself.

You might wonder, how can we possibly expect to achieve $O(\log n)$ amortized time for insertion if we are using brute force (which may take as much as $O(n)$ time) to compute the sizes of the subtrees? The reason is to first recall that we do not need to compute subtree sizes unless a rebuild event has been triggered. Every node that we are visiting in the counting process will need to be visited again in the rebuilding process. Thus, the cost of this counting process can be "charged for" in the cost of the rebuilding process, and hence it essentially comes for free!

The Lazy Way: By the way, there is an easier way for computing subtree sizes, but this requires that we use additional space to store the size value of each node explicitly within each node. The size of a node is easy to update whenever there are changes in the tree's structure, since we have:

```
size(u) = (u == null ? 0 : 1 + size(u.left) + size(u.right))
```

Amortized Analysis: Because we rebuild the tree whenever its height exceeds $O(\log n)$, `find` operations run in $O(\log n)$ time, even in the worst case. The insert/delete operations can take longer, however, because they may trigger rebuilding. The next theorem shows that for a sequence of k operations, the total cost for insertion and deletion is $T(k) = O(k \log k)$. It follows immediately that the amortized cost for the operations is $T(k)/k = O(\log k)$. The proof is based on a presentation from the Open Data Structures web site.

Theorem: Starting with an empty Scapegoat Tree any sequence of k `insert` and `delete` operations (including rebuilding) can be performed in $O(k \log k)$ time.

Proof: As usual, we will apply our usual token-based argument. We imagine that each node stores a number of tokens. Each token can pay for some constant, c, units of time spent rebuilding. Our scheme will give out a total of $O(k \log k)$ tokens, and we will show that every call to `buildSubtree(u)` will be paid for with tokens stored at node `u`.

During an insertion or deletion, we give one token to each node along the path to the inserted or deleted node. Recalling that n denotes the number of elements in the tree at any time, we have $n \leq k$. As shown earlier, the height of the tree is never greater than $\log_{3/2} n$ (for otherwise we rebuild). Therefore, we generate at most $\log_{3/2} m \leq \log_{3/2} k$ new tokens per operation. During a deletion we also store an additional token on the

side. Thus, in total we generate at most $O(k \log k)$ tokens. All that remains is to show that these tokens are sufficient to pay for all calls to `buildSubtree(u)`.

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

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

By definition of size, $\texttt{size(u)} = 1 + \texttt{size(u.left)} + \texttt{size(u.right)}$. It follows 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), the tree was perfectly balanced, and thus

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

Therefore, the number of `insert` or `delete` operations that have affected `u.left` or `u.right` since then is at least

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

It follows that at least this many tokens stored at `u` that are available to pay for the $O(\texttt{size(u)})$ time it takes to call `buildSubtree(u)`.

If we call `buildSubtree(u)` during a deletion, it is because $m > 2n$. In this case, we have $m - n > n$ tokens stored on the side, and we use these to pay for the $O(n)$ time it takes to rebuild the root. This completes the proof.