Scapegoat Trees:
- Arne Anderson (1989)
- Galperin & Rivest (1993) rediscovered/extended
- Amortized analysis
  - $O(\log n)$ for dictionary ops amortized (guaranteed for find)
- Just let things happen
- If subtree unbalanced
  - rebuild it

Recap:
- Seen many search trees
- Restructure via rotation
- Today: Restructure via rebuilding
- Sometimes rotation not possible
- Better mem. usage

Overview:
Insert:
- Same as standard BST
  - if depth too high
    - trace search path back
    - find unbalanced node—scapegoat
    - rebuild this subtree

Find:
Same as std BST
- Tree height $\leq \log_{3/2} n \approx 1.71 \log n$

Delete:
- Same as std. BST
- If num. of deletes is large rel. to $n$—rebuild entire tree!

How to rebuild?
rebuild(p):
- inorder traverse p's subtree → array $A[]$
  - buildSubtree(A)
  - $\text{buildSubtree}(A[0..k-1])$
    - if $k = 0$ return null
    - $j \leftarrow \lfloor k/2 \rfloor$; $x \leftarrow A[j]$ median
    - $L \leftarrow \text{buildSubtree}(A[0..j-1])$
    - $R \leftarrow \text{buildSubtree}(A[j+1..k-1])$
    - return Node($x$, $L$, $R$)

Example:
\[\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
p: & b & \rightarrow & a & c & e & d & f \\
L & d & \rightarrow & a & b & c & e & f \\
\end{array}\]

Time $= O(k)$

Node(x, L, R)
Insert:
- \( n++ \); \( m++ \)
- Same as std BST but keep track of inserted node’s depth \( d \)
- if \( d > \log_{3/2} m \) \{ 
  \* rebuild event \*
- trace path back to root
- for each node \( p \) visited, size \( p \) = no. of nodes in \( p \)’s subtree
- if \( \frac{\text{size}(p, \text{child})}{\text{size}(p)} > \frac{2}{3} \)
  \( p \leftarrow \text{rebuild}(p) \)
- break

How to compute size \( p \)?
- Can compute it on the fly
- While backing out, traverse “other sibling”
- Too slow? No!
  \( \Rightarrow \) Charge to rebuild.

Details of Operations:
- Init:
  \( n \leftarrow m \); \( root \leftarrow \text{null} \)
- Delete:
  - Same as std BST
  - \( n-- \)
  - if \( m > 2n \), \( \text{rebuild}(\text{root}) \)

Example:

Scapegoat Trees

Must there be a scapegoat? Yes!

Proof: By contradiction
- Suppose \( p \)’s depth \( > \log_{3/2} n \) but \( \forall \) ancestors of \( p \) that satisfies scapegoat condition

Lemma: Given a binary tree with \( n \) nodes, if \( \exists \) node \( p \) of depth \( > \log_{3/2} n \), then \( \exists \) ancestor of \( p \) that satisfies scapegoat condition
Theorem: Starting with an empty tree, any seq. of \( k \) inserts + deletes takes total of \( O(k \log k) \) time.

Corollary: Amortized time is \( O(\log k) \)

Proof: Token-based argument

Overview:
- We will assign tokens to nodes of tree
- Add some tokens "on the side"
- Will show:
  - Total tokens = \( O(k \log k) \)
  - Enough tokens to pay for all rebuildings

Token assignment:
- Whenever we insert/delete, add a token to each node visited in the search
- During each deletion - add 1 token "on the side"
- By height bound - \( O(k \log k) \) token total.

Amortized Analysis:
- Tree height is \( O(\log n) \)
  - [since we rebuild whenever higher]
  \( \Rightarrow \) find is \( O(\log n) \) even in worst case
- But insert + delete can take up to \( O(n) \) time (if entire tree is rebuilt)

Therefore - Node \( u \) has collected at least \( \frac{1}{2} \) \( \text{size}(u) - 1 \) tokens

Since it takes \( O(\text{size}(u)) \) time to rebuild \( u \), it follows that (up to adjusting constants) we have enough tokens to pay for rebuild.

The last time a subtree containing \( u \) was rebuilt, it was perfect balanced
\( \Rightarrow \) \( \text{size}(u \text{left}) - \text{size}(u \text{right}) \leq 1 \)
(at that time)

This implies that since last rebuild, we had at least \( \frac{1}{3} \) \( \text{size}(u) - 1 \) inserts/deletes involving \( u \).

This implies:
\[
\frac{1}{2} \text{size}(u \text{left}) > \text{size}(u \text{right})
\]
\[
\Rightarrow \text{size}(u \text{left}) - \text{size}(u \text{right}) > \frac{1}{2} \text{size}(u \text{left})
\]
\[
> \frac{1}{3} \text{size}(u)
\]

Claim: There are always enough tokens to pay for rebuilding.

Proof:
If we call buildSubtree \( (u) \), we know \( u \) is a scapegoat. Assume w.l.o.g. that:
\[
\frac{\text{size}(u \text{left})}{\text{size}(u)} > \frac{2}{3}
\]
By def: \( \text{size}(u) = \text{size}(u \text{left}) + \text{size}(u \text{right}) \)

Scapegoat Trees

III

II

I

\( \implies \)