Scapegoat Trees I

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

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 $\leq \log_{3/2} n \approx 1.71 \lg n$

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

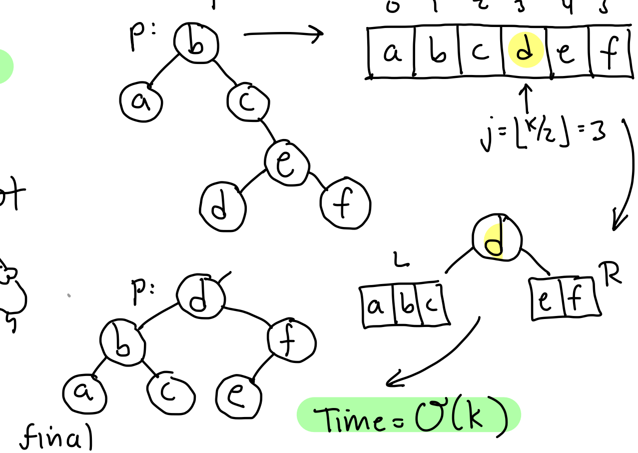
Delete:
- Same as std. BST
- If num. of deletes is large rel. to n – rebuild entire tree!

How? Maintain $n, m \leftarrow 0$

Insert: $n{+}{+}, m{+}{+}$

Delete: $n{-}{-}$ ⋯→ If $m > 2n$ rebuild

Example:



Time = $O(k)$

How to rebuild?

rebuild(p):
- inorder traverse p's subtree → array A[ ]
- buildSubtree(A)

buildSubtree(A[0..k-1]):
- if $k = 0$ return null
- $j \leftarrow \lfloor k/2 \rfloor$; $x \leftarrow A[j]$ median
- $L \leftarrow$ buildSubtree(A[0..j-1])
- $R \leftarrow$ buildSubtree(A[j+1..k-1])
- return Node(x, L, R)

Scapegoat Trees II

Details of Operations:

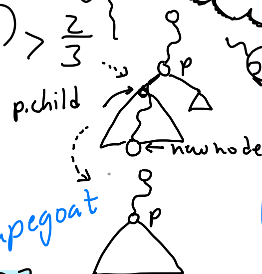
Init: $n \leftarrow m \leftarrow 0$ root $\leftarrow$ null

Delete:
- Same as std BST
- n--
- if $m > 2n$, rebuild(root)

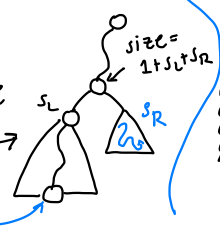
Time: $O(n)$

Insert:
- n++; m++
- same as std BST but keep track of inserted node's depth $\rightarrow d$
- 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 $\frac{\text{size(p.child)}}{\text{size(p)}} > \frac{2}{3}$

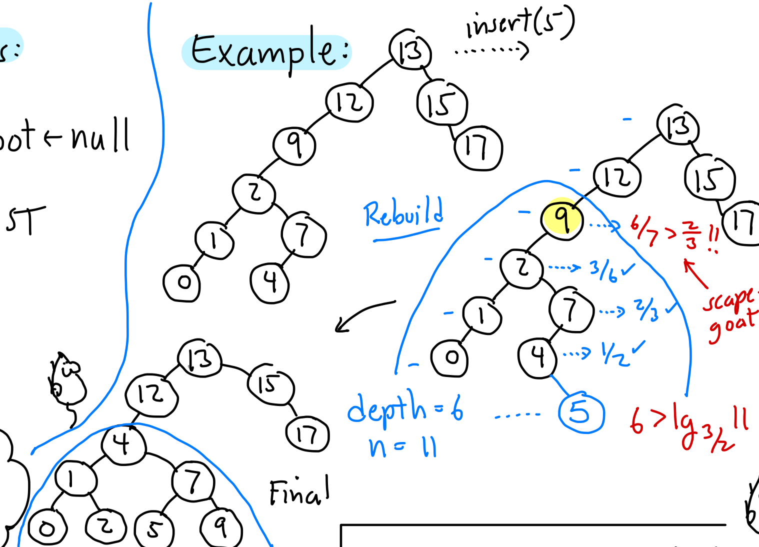
    p ← rebuild(p)
  - break
  }



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



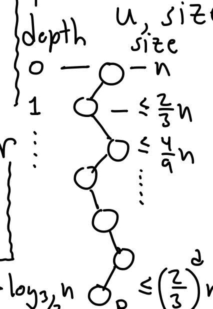
Example:



Must there be a scapegoat? yes!

Lemma: Given a binary tree with n nodes, if ∃ node p of depth $> \log_{3/2} n$, then ∃ ancestor of p that satisfies scapegoat condition

Proof: By contradiction
- Suppose p's depth $> \log_{3/2} n$ but ∀ ancestors u, size(u.child) ≤ 2/3 · size(u)



$\Rightarrow$ Since p has 1 node:

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

$\Rightarrow \left(3/2\right)^d \leq n$

$\Rightarrow d \leq \log_{3/2} n$ □

Scapegoat Trees III

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 - $O(k \log k)$ tokens 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)

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: $\text{size}(u) = 1 + \text{size}(u.\text{left}) + \text{size}(u.\text{right})$

This implies:

$$\tfrac{1}{2}\,\text{size}(u.\text{left}) > \text{size}(u.\text{right})$$

$$\Rightarrow \text{size}(u.\text{left}) - \text{size}(u.\text{right}) > \tfrac{1}{2}\,\text{size}(u.\text{left}) > \tfrac{1}{3}\,\text{size}(u)$$

The last time a subtree containing u was rebuilt, it was perfect balanced

$\Rightarrow$ (at that time) $\text{size}(u.\text{left}) - \text{size}(u.\text{right}) \le 1$

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

Therefore - Node u has collected at least $\tfrac{1}{3}\,\text{size}(u) - 1$ 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. $\square$