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 $\leq \log_{3/2} n \approx 1.7 \log n$

Example:

```
\begin{center}
 p: \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \\
 a \quad b \quad c \quad d \quad e \quad f \\
\end{center}
```

```
\begin{center}
 j = \lfloor \frac{4}{5} \rfloor = 3 \\
\end{center}
```

```
\begin{center}
 d \quad e \quad f \\
 a \quad b \quad c \\
\end{center}
```

```
\begin{center}
 L \quad R \\
 d \quad e \quad f \\
 a \quad b \quad c \\
\end{center}
```

```
\begin{center}
 Time = O(k) \\
\end{center}
```

Scapegoat Trees

```
\text{Node-scapegoat} \\
\\n\text{ rebuil} \\
\text{ this subtree}
```

```
\text{Time} = O(k) \\
```

How to rebuild? rebuild (p):
- Inorder traverse p's subtree $\rightarrow$ array $A[\cdot]$
- buildSubtree ($A$)
  - buildSubtree ($A[0..k-1]$):
    - if $k = 0$ return null
    - $j = \lfloor \frac{k}{2} \rfloor$; $x = A[j]$ median
    - $L \leftarrow$ buildSubtree ($A[0..j-1]$)
    - $R \leftarrow$ buildSubtree ($A[j+1..k-1]$)
    - return Node ($x$, $L$, $R$)
Details of Operations:

**Init:**
\[ n \leftarrow m \rightarrow 0; \text{ root} \leftarrow \text{null} \]

**Delete:**
- Same as std BST
  - \( n \leftarrow \)
  - if \( m > 2n \), rebuild(\text{root})

**Time:**
\( O(n) \)
\[ m = n \]

Example:

![Binary Tree Example](image)

**Insert:**
- \( n \leftarrow m \leftarrow m + 1 \)
  - Same as std BST but keep track of inserted node's depth \( d \)
- if \( d > \log_{3/2} m \) \( \Rightarrow \) rebuild event

\[ \approx 1.7 \log m \] - trace path back to root

- for each node \( p \) visited, \( \text{size}(p) = \text{no. of nodes in } p\text{'s subtree} \)
  - if \( \text{size}(p\text{'s child}) > \frac{2}{3} \text{size}(p) \)
    - rebuild(\( p \))
    - break

**Scapegoat Trees**

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

**Proof:** By contradiction
- Suppose \( p\)'s depth \( > \log_{3/2} n \) but \( \forall \) ancestors

**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

There is at least one node \( \exists ! \leftarrow d \geq \log_{3/2} n \)
**Theorem:** Starting with an empty tree, any seq. of $k$ inserts + deletes takes total $T(k)$ of $O(k \log k)$ time.

**Corollary:** Amortized time is $O(\log k) = T(k)/k$

**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 rebuilding

**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)

**Scapegoat Trees**

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

**Proof:**
If we call $\text{buildSubtree}(u)$, we know $u$ is a scapegoat. Assume w.l.o.g. that:
- $\frac{1}{2} \cdot \frac{\text{size}(u)}{\text{size}(u)} > \frac{2}{3}$
- By def: $\text{size}(u) = \text{size}(u) + \text{size}(u)$

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

This implies:
- $\frac{1}{2} \cdot \text{size}(u) > \text{size}(u)$
- $\Rightarrow \text{size}(u) - \text{size}(u) > \frac{1}{2} \cdot \text{size}(u)$
- $\Rightarrow \frac{1}{3} \cdot \text{size}(u)$
Other/Better Criteria?
- Expected case: Some keys more popular than others
- Self-adjusting: Tree adapts as popularity changes

How to design/analyze?

Recap: Lots of search trees
- Unbalanced BSTs
- AVL Trees
- 2-3, Red-black, AA Trees
- Treaps + Skip lists

Focus: Worst-case or randomized expected case

Lesson: Different combinations of rotations can:
- bring given node to root
- significantly change (improve) tree structure

Splay Tree: A self-adjusting binary search tree
- No rules! (yay anarchy!)
  - No balance factors
  - No limits on tree height
  - No colors/levels/priorities
- Amortized efficiency:
  - Any single op - slow
  - Long series - efficient on avg.

Intuition: Let T be an unbalanced BST. Suppose we access its deepest key.

Idea I: Rotate “a” to top (Future accesses to “a” fast)

Idea II: Rotate 2 at a time - upper + lower

Splay Trees I

Tree’s height has reduced by ~ half!
**ZigZig(p):** [LL case]

- **Splay (Key x):**
  - Node p ← find x by standard BST search
  - while (p ≠ root) {
    - if (p is child of root) zig(p)
    - else /* p has grand parent */
      - if (p is LL or RR grand child) zigZig(p)
      - else /* p is LR or RL grand child */ zigZag(p)
  }

**Subtrees A, C move up**

- **ZigZag(p):** [LR case]

**Subtrees C, E of p move up**

- **Zig(p):** [L case]

**Subtree A moves up**

- **C unchanged**

**Example:**

- splay(3)
  - RL zigZag
  - ZigZig

**Insert (x):**

- **Node p ← splay(x)**
  - if (p.key == x) Error!!
  - q ← new Node(x)
  - if (p.key < x)
    - q.left ← p
    - p.right ← p.right
    - p.right ← null
  - else ... (symmetrical)...
  - root ← q

**Find (x):**

- root ← splay(x)
  - if (root.key == x) return root.value
  - else return null
Announcements 4/4

- Prog assignment 2
  - Deadline extended to Fri, 11:59 pm

- Homework 3
  - Due Tue, Apr 11 in class

- Midterm 2
  - Thu, Apr 13
    - Closed book / closed notes
    - 2 cheat sheets
    - Coverage through this week - kd-trees, splay, skip-lists