**Design Decision: Lazy Deletion**

*Lazy Deletion:* Rather than deleting a node from a tree, merely *mark* it as being deleted
- operate around it as usual
- (just don’t return it as the result of a `Find()` op)

```
Delete(T, 2);
Delete(T, 20);
Delete(T, 11);
```

---

**CSE 373: Self-Adjusting Trees**  
(“Cookbook Data Structures”)

Chapter 4
*(and Section 12.2)*
Motivation

All Binary Search Tree Operations are $O(d)$
- $d$ can range from $\log N$ to $N$
- generally, $d$ is $O(\log N)$
- statistically, $d$ is $O(\log N)$ on average
- **but**, for common(?) insertion orders, $d$ will be $O(N)$
  *i.e.*, inserting sorted lists in order

A Solution

**Self-Adjusting Binary Search Trees**: BST’s that automatically rearrange themselves to keep operations $O(\log N)$
- AVL Trees
- Splay Trees
- Red-Black Trees
AVL Trees

**The idea:** A balance condition is placed on the tree. Whenever an `insert()` breaks the condition, we rearrange the tree to fix it.

What should the balance condition be?

Rotations

Rotation: a simple way of rearranging a tree without breaking the binary search property
AVL Tree Strategy

Balance Condition: Every node’s left and right subtrees must have a height difference of no more than one

Two Cases of Bad Inserts: 

Given:

Case I

Case II
Fixing Case I

Trying to Fix Case II
Fixing Case II

AVL Tree Summary

- Keep every node’s subtrees “almost balanced”
- When insertions break the “almost balanced” condition, use rotations to fix things up
- Use lazy deletion to keep things simple
- All operations are O(log N)
- Implementation Cost:
  - must store depth of each node’s child subtrees
  - must implement 4 cases for bad insertion
Splay Trees

- Every time a node is accessed, rotate it to the top of the tree no matter what
- Over time, trees tend to get shallower because rotations don’t make the tree any deeper
- Result: Although any one operation may require $O(N)$ time, a series of $k$ operations is guaranteed to be $O(k \log N)$ – amortized analysis
- Benefits:
  - no need to store depths of nodes’ subtrees

Red-Black Trees

Red-Black Trees: Binary Search Trees with the following properties:
  - every node is colored either red or black
  - the root is always black
  - if a node is red, its children must be black
  - every path from a node to a NULL pointer must contain the same number of black nodes
Example Red-Black Tree

\[ \text{Intuitively...} \]
- every path from root to leaf has same number of black nodes
- though they may have different # red nodes, alternate at worst
- Thus, \( d = 2\log(N+1) = \Theta(\log N) \)

Inserting into Red-Black Trees

\[ \text{Insert} (T, 0); \]
\[ \text{Insert} (T, 3); \]
\[ \text{Insert} (T, 22); \]
\[ \text{Insert} (T, 23); \]
\[ \text{Insert} (T, 24); \]