Today …

• In-place mergesort
• Red-Black Trees
Comments on AVL Trees

• AVL vs other approaches
  – Rotations and traversals are hard to get right
  – But more importantly, the traversals create overhead

• A Red Black tree is an alternative approach
  – In practice, (slightly) faster insertion and deletion
  – But (slightly) slower retrieval time
  – Red-black trees are typically implemented over AVL trees in practice

Red Black Trees [Bayer, 1972]

• Basic idea
  – Further relax balancing constraints
  – Use a single top-down pass for insertion/deletion
  – Rebalance as part of this top-down pass
    • Unlike AVL where a bottom-up pass is needed to rebalance
  – Generally faster than an AVL tree (insertion and deletion)
  – Lookup/retrieve not as fast
    • Because height of tree can be longer … but still $O(\log n)$

So how does this work?
Balance Constraints

1. Every node is assigned a color (either Red or Black)
2. The root is always Black
3. The children of a Red node are always Black
4. Every path from the root to a NULL node has the same number of Black nodes

Properties of Red Black Trees

• If every path from the root to a NULL node contains \( B \) black nodes …
  – There are at least \( 2^B - 1 \) black nodes
  – Recall that a full tree with height \( h \) contains \( 2^h - 1 \) nodes
  – Thus \( 2^B - 1 \) is the size of the smallest, full Red Black tree!
Properties of Red Black Trees

• Since the root is black and there can not be 2 consecutive red nodes on a path …

  – The height of a Red Black tree is at most: \(2 \times \log_2(n+1)\)
  – Thus the height is \(O(\log n)\)!

How did we get this?

– Notice that a root-leaf path can have at most \(2B\) nodes (i.e., \(h/2 \leq B \leq h\))

\[
2^{h/2} - 1 \leq 2^B - 1 \leq n \\
2^{h/2} \leq n + 1 \\
h/2 \leq \log(n+1) \\
h \leq 2 \times \log(n+1)
\]

Properties of Red Black Trees

• Given a max height of \(2\log(n+1)\), is a Red Black tree always “Balanced”?

  – Recall the balance factor must be -1, 0, or 1 for each node in a balanced binary tree

• A Red Black tree technically is not balanced in this way … but still has \(O(\log n)\) height

  – And this is what we really care about!

\(B = 2\)

This is a valid Red Black tree
Inserting Nodes

- Always assign inserted nodes the color Red
  - If we assigned the node the color Black, we would create a path with more Black nodes than any other path
- If the parent of the inserted node is Black, then we are done!
  - Similar case if we are inserting the root
- If the parent is Red, we violate Property 3
  - We now have to adjust the tree
  - … using tree rotations and color changes

Inserting Nodes

There are two approaches:

“Bottom-up” insertion

- Similar to AVL by propagating changes up the tree

“Top-down” insertion

- Optimization to avoid traversing back up the tree

We’ll only discuss the bottom up approach
But look at a similar top-down approach later …
Inserting Nodes (Bottom Up)

If the inserted node’s parent is Red …
• We have to consider several cases
  – Let …
    \( X = \) inserted node
    \( P = \) parent
    \( S = \) sibling of \( P \)
    \( G = \) grandparent of \( X \)
  – Note that \( G \) must be Black!
• If \( S \) is Red
• If \( S \) is Black
  – If \( X \) is an outside grandchild
  – If \( X \) is an inside grandchild

Sibling is Red

• If \( S \) is Red
  – Change \( P \) and \( S \) to Black
  – Change \( G \) to Red
  – Repeat with \( G \) as the new \( X \)

NOTE: Because we are possibly propagating up, we consider the general case where \( X \) may have subtrees \( A \) and \( B \)

We have not changed the number of Black nodes on a path!