Announcement

• Complete Quiz 8 by tomorrow discussion time.
• Assignment 5 due on Thursday at 4pm.
• CS Minor

http://www.cs.umass.edu/ugrad-education/cs-minor
Lecture 21: Self-Balancing Trees
AVL Tree

• **AVL** is the simplest self-balancing tree.
• It’s invented by G.M. Adelson-Velskii and E.M.Landis, in 1962.
• The idea is to keep track of the difference in height of each subtree. Example?
• This difference must be within a range of [-1,1].
• When inserting/removing nodes, if the balance is violated, it uses rotation to bring the tree back to balance. How?
AVL Tree

- We define the balance measure as $h_R - h_L$.
- The tree is ok if this measure is within $[-1,1]$, otherwise:
  - Balancing a Left-Left tree
  - Balancing a Left-Right tree
  - Balancing a Right-Right tree
  - Balancing a Right-Left tree
  - Demo
AVL Tree

• It can be shown that if an AVL tree contains $N$ nodes, the height of the tree is bounded by

$$1.44 \log_2(N)$$

• This means even in the worst case, the search/insert/delete costs are $O(\log N)$, so it never degenerates to the $O(N)$ case.
Red-Black Trees

• Nodes are colored as either red or black.
• Rules:
  1. Each node is either red or black.
  2. The root is always black.
  3. If a node is red, its children must be black (but the converse isn’t necessarily true).
  4. Every path from the root to a leaf, or a null child, must contain the same number of black nodes (this is called the black height).
Red-Black Trees

• Rule 4 is the same as saying: the black height must be the same for all paths from the root to a leaf.

• During insertion/deletion, when rules are violated, you can fix the violation by:
  – Change the colors of nodes;
  – Perform rotations.
Red-Black Trees

• Workshop applet demo
• Available operations:
  1. R/B
  2. RoL
  3. RoR
  4. Flip
Red-Black Trees

• The insertion process: assume we insert X
  X is the node of interest
  P is the parent of X
  G is the grant parent of X (P’s parent)
• The insertion follows the standard BST search to find the spot to insert. During the process, we perform color flips on the way down.
• This may cause a red-red conflict, which we will fix using rotation.
Red-Black Trees

• Color flips on the way down.
• Root remains black.
• This does not change black heights, because the flip adds one black node to each path.
  – Rule 4 is not violated.
  – However, rule 3 may be violated.
• Next, insert the new node X as a red node.
Red-Black Trees

• Inserting the new node $X$ as red.
• If $P$ is black, we are all set.
• If $P$ is red, we get a red-red conflict.
• There are two situations:
  – $X$ is an outside grandchild $\rightarrow$ One rotation to fix + two color switches
  – $X$ is an inside grandchild $\rightarrow$ Two rotations to fix + two color switches
Red-Black Trees

• Are there other possibilities?
  – What if X or P may have siblings?

• If X has a sibling:
  – P must be black, why?
  – Since P is black, there is no problem inserting X as a red node.

• If P has a sibling U (X’s uncle):
  – If P is black, we are all set.
  – If P is red, U must also be red, why? But this would have been flipped on the way down!
Red-Black Trees

• Are there other possibilities?
  – What if X or P may have siblings? → P must be black.

• Typo on the textbook:
  – Figure 9.15: 75 → 25
  – pp.453 middle: ‘unless P is red’ → ‘unless P is black’.
Red-Black Trees

• Rotations on the way down.

• So we’ve discussed inserting the new node. But the color flips all the way down can cause red-red violation in interior nodes.

• The situations are the same with the cases discussed previously:
  – If X is an outside grandchild.
  – If X is an inside grandchild.
• Insertion (putting everything together):
if (key < current.key) {
  if (current.leftChild == null) {
    current.leftChild = newNode;  // insert as red
  } else {
    if (current.leftChild and rightChild both red)
      color flip;                  // color flip on the way down
      recursively insert newNode to the left subtree;
    if (current.leftChild is red) {
      if (current.leftChild.leftChild is red)
        outside grandchild case;
      else if (current.leftChild.rightChild is red) {
        inside grandchild case;
      }
    }
  }
} else if (key > current.key) {  // insert to right subtree, symmetric
Red-Black Trees

In summary:
• Insert as in standard BST
• Flip if both children are red
• Insert the new node as red.
• Adjust if red-red conflict, by using rotation

Advantages:
• Good tree balance
• One pass algorithm.
Red-Black Trees

• Efficiency bounded by:

\[ 2 \log_2(N) \]