Test #1 is graded. I will hand them back individually during lab this week; I also want to (briefly) talk to each of you about how the course is going.

Prelab 6 is optional. Can hand it in now or by class on Wednesday.

Course Evaluation Comments: thank you, comments are helpful!

- lab difficulty: avg 3
- lab learning: avg 4.5
- prelabs: avg 3.5
- lecture learning: avg 3.9
- lab interest: avg 3.4
- hours per week: avg 9.5
- lecture pace: avg 3.2
- other: ppt pseudocode too java (totally, it’s mostly Java actually.) Trouble asking for help, lab helpers busy. More cookies, higher salary.
Red-Black Trees

Many BST operations are $O(\text{height of tree})=O(n)$. Want height=$\log n$

An **AVL tree** is a binary search tree such that for every node $r$,
- the heights of the left and right subtrees differ by at most 1,
- the left child is an AVL tree, and
- the right child is an AVL tree.

A red-black tree is another balanced BST, faster than AVL’s in practice.

A **red-black tree** is a binary search tree such that
- **color property**: every node is either red or black
- **root property**: the root node is black
- **internal property**: the children of a red node are black
- **depth property**: for each node $v$, all $v \rightarrow$ null pointer paths contain the same number of black nodes

Operations same as for AVL trees, same big-Oh runtime.
Less restrictive than AVL’s, so rebalancing is faster but height is slightly worse.
A red-black tree is a binary search tree such that
- **color property**: every node is either red or black
- **root property**: the root node is black
- **internal property**: the children of a red node are black
- **depth property**: for each node $v$, all $v \rightarrow$ null pointer paths contain the same number of black nodes

Is it still a red-black tree if we:
- change the 22 to black? 22 and 27?
- color all the nodes black?
We didn’t cover the following slides in class, but I’m including them in case you want a bit more information about red-black tree insertion and deletion.
As with binary search trees and AVL trees, always insert as a leaf node.
- if this new leaf is the root, color it black (to maintain the root property)
- otherwise, color the new leaf node red (to maintain the depth property)
- if this node’s parent P is black, we’re done.
- otherwise, this node’s parent is red and we violated the internal property
  - grandparent G must be black (because had RB tree before insertion)
  - two cases, based on sibling S of parent P:

**case 1: sibling S is red**

We’ll avoid this case completely: When searching for insertion point, at each node, check children’s colors:
- if both red, swap to black, make root red
- all paths still have same number of black nodes
- (if root is root of whole tree, keep it black)
- (this effects all paths by 1, no prob.)
- but G’s parent may also be red; if so, we know G’s parent’s sibling (!!) is not red because we’ve specifically been swapping to avoid this situation.
As with binary search trees and AVL trees, always insert as a leaf node.

- if this new leaf is the root, color it black (to maintain the root property)
- otherwise, color the new leaf node red (to maintain the depth property)
- if this node’s parent P is black, we’re done.
- otherwise, this node’s parent is red and we violated the internal property
  - grandparent G must be black (because had RB tree before insertion)
  - two cases, based on sibling S of parent P:

  case 2a: sibling S is black

Note: we may have to rotate multiple times along the path from the insertion point up to the root, so t0 and t1 are not necessarily empty.

(If X is a leaf then S is empty (depth prop) and this still works.)
As with binary search trees and AVL trees, always insert as a leaf node.
• if this new leaf is the root, color it black (to maintain the root property)
• otherwise, color the new leaf node red (to maintain the depth property)
• if this node’s parent P is black, we’re done.
• otherwise, this node’s parent is red and we violated the internal property
  • grandparent G must be black (because had RB tree before insertion)
  • two cases, based on sibling S of parent P:

  case 2b: sibling S is black

  The depth property still holds, and no two red in a row!
Red-Black Tree Deletion

For deletion, use normal BST algorithm until we remove a leaf or single child:
• if node is red, delete it and we’re done (maintains depth, internal prop)
• if node is black, removing it could violate depth property:
  • when we remove it, make sure the node is red...
  • do this by re-coloring the tree as we move down it
  • want to maintain that node to be removed is red.
  • X is the node we possibly want to remove, P its parent, S its sibling
  • P is already red because we previously thought we may remove it
  • so X and S are black (internal property not violated)
  • if X has 2 black children, case-work and rotations...
  • if one of X’s children are red... more case-work. Tedium ensues.