Red-Black Trees

Need for red-black trees (and others)

- Binary search trees are efficient at insertion, deletion, and searching
- If unbalanced, the tree deteriorates
- Can become unbalanced if:
  - Items are added in sequence
  - Items are added in reverse sequence
- Red-black trees are binary search trees with some added features
- Code to implement is more complex

Red-black trees

- Searching is essentially the same
- Deletion and insertion are much more complex
  - Top-down insertion – changes to the tree structure are made as the routine searches for where to insert the item
  - Bottom-up insertion – changes to the tree structure are made after finding where to insert the item – i.e., the routine goes back up the tree and is less efficient
- Binary search trees can become either left or right unbalanced – right if in ascending order, left if in descending order
  - Totally unbalanced and the tree becomes a linked list and searching proceeds at O(N) rather than O(logN)

Red-black tree balancing act

- Balance is ensured when items are inserted
- Nodes are colored (red and black is arbitrary)
- Rules followed during insertion and deletion:
  - 1. Every node is either red or black
  - 2. The root is always black
  - 3. If a node is read, its children must be black
  - 4. Every path from the root to a leaf (null child) must contain the same number of black nodes ("black height" is the number of black nodes on a path from root to leaf)
- (see applet example)

Rotations

- Cannot just balance a red-black tree by changing colors, need to physically rearrange the nodes
- Rotations do the following:
  - Raise some nodes and lower others to help balance
  - Ensure that the characteristic of a binary search tree are not violated (left child is less, right child is greater than or equal)
- Right rotation: "top" node moves down and to the right, its left child will move up to take its place
- Left rotation: "top" child will move down and to the left, its right child will move up to take its place

Algorithm

- X is a particular node, P is the parent of X, G is the grandparent of X
- While looking for where to insert a node, do a color flip when you find a black node with two red children
  - If it causes a red-red conflict fix with single rotation if red child is outside grandchild, fix with double rotation if red child is inside grandchild
- After insertion if parent is black, attach red node
- If parent is red perform two color changes, then:
  - Inserted node is an outside grandchild, one rotation
  - Inserted node is an inside grandchild, two rotations

Color flips on the way down

- Every time the routing finds a black node with two red children it must change the children to black and the parent to red (unless the parent is the root)
  - Leaves the black height unchanged
  - If results in a parent and child being red, then do a rotation (will come back to this)

Rotations after insertion

- Three post insertion possibilities:
  - Parent is black
  - Parent is red and inserted node is an outside grandchild
  - Parent is red and inserted node is an inside grandchild
- If parent is black don't need to do anything (inserted is always red)
- If parent is red and inserted is outside grandchild
  - Switch the color of the grandparent
  - Switch the color of the parent
  - Rotate in the direction that raises the inserted node (right or left)

Rotations after insertion

- Parent is red and inserted node is an inside grandchild – two rotations and some color changes
  - Switch the color of inserted grandparent
  - Switch the color of the inserted
  - Rotate with inserted parent at the top in the direction that raises inserted
  - Rotate again with inserted grandparent at the top in the direction that raises inserted
- These are the only possible situations for post insertion, assuming we follow the rules

Rotations on the way down

- Insure the post insertion rules are simplified
- Parent and offending node are both red and offending is an outside grandchild:
  - Switch the color of offending node's grandparent
  - Switch the color of offending node's parent
  - Rotate with offending node's grandparent at the top in the direction that raises inserted
- Parent and offending node are both red and offending node is an inside grandchild:
  - Change the color of grandparent
  - Change the color of the offending node
  - Rotate with parent of offending node that raises offending node
  - Rotate with grandparent in the direction that raises offending node

Deletions

- Complicated for simple binary trees
- Much more complicated for red-black trees
- Often not done, but node marked for deletion

Efficiency of red-black trees

- Searching, insertion, and deletion in O(logN) time
- Storage is increases slightly to accommodate color of node
- Searching cannot require more than $2*\log_2 N$
- Insertion slightly slower because of need to do color shifts and rotations
- Major advantage is that insures that the tree is relatively balanced at all times

Implementation

- Need a red-black field for nodes
- On the way down check if current node is black and two children red, if so change the color of ll three (except if root)
- After the color flip check that there are no red-red parent child relationships, if so perform appropriate rotations: one for outside grandchild, two for inside grandchild