Binary Search Trees

• Implementing Balancing Operations
  – AVL Trees
  – Red/Black Trees

• Reading: 11.5-11.6
Implementing Balancing Operations

• Knowing rotations, we must know how to detect that the tree needs to be rebalanced and which way
• There are 2 ways for tree to become unbalanced
  – By insertion of a node
  – By deletion of a node
• There are two mechanisms for detecting if a rotation is needed and which rotation to perform:
  – AVL Trees
  – Red/Black Trees
• It is best for both to have a parent reference in each child node to backtrack up the tree easily
Implementing Balancing Operations

- AVL trees (after Adel’son-Vel’ski and Landis) keep a balance factor attribute in each node that equals the height of the right sub-tree minus the height of the left sub-tree.
- Each time an insertion or deletion occurs:
  - The balance factors must be updated.
  - The balance of the tree must be checked from the point of change up to and including the root.
- If a node’s balance factor is > +1 or < -1, the sub-tree with that node as root must be rebalanced.
Implementing Balancing Operations

• If the balance factor of a node is -2
  – If the balance factor of its left child is -1
    • Perform a right rotation
  – If the balance factor of its left child is +1
    • Perform a leftright rotation

• If the balance factor of a node is +2
  – If the balance factor of its right child is +1
    • Perform a left rotation
  – If the balance factor of its right child is -1
    • Perform a rightleft rotation
AVL Tree Right Rotation

Initial Tree

- 7 (-1)
  - 5 (0)
    - 3 (0)
  - 9 (0)

After Add 1

- 7 (-2)
  - 5 (-1)
    - 3 (-1)
      - 1 (0)
  - 9 (0)

Node is -2
Left Child is -1
AVL Tree Right Rotation

After Right Rotation

```
       5 (0)
     /     \
    3 (-1)  7 (0)
  /    \    /    \
1 (0)   6 (0) 9 (0)
```
AVL Tree Rightleft Rotation

Initial Tree

- 10 (1)
- 5 (-1)
- 15 (-1)
- 3 (0)
- 11 (0)

After Remove 3

- 10 (2)
- 5 (0)
- 15 (-1)
- 11 (0)
- 17 (0)

Node is +2
Right Child is -1
AVL Tree Rightleft Rotation

After Right Rotation

After Left Rotation
Red/Black Trees

• Red/Black Trees (developed by Bayer and extended by Guibas and Sedgewick) keep a “color” red or black for each node in the tree
  – The root is black
  – All children of a red node are black
  – Every path from the root to a leaf contains the same number of black nodes

• The maximum height of a Red/Black tree is roughly $2 \times \log n$ (not as well controlled as an AVL tree), but the longest path is still $O(\log n)$
Red/Black Tree Insertion

- When inserting a new node, it is put in the normal place by its value and its color is always set to red.
- At the end of insertion, the root is always set to black.
- While current node’s parent is not the root and is red:
  - If the color of the parent’s sibling is red, no rotation is needed, but re-coloring is needed from current node:
    - Parent’s color is set to black
    - Parent’s sibling’s color is set to black
    - Grandparent’s color is set to red
    - Current node is set to point to the grandparent for loop
Red/Black Tree Insertion

- Insertion – No Rebalancing

Initial Tree
- After Insertion of an 8
  - After no rebalancing - just re-coloring the tree
Red/Black Tree Insertion

• Insertion – No Re-balancing

Initial Tree:

- 15
- 7
- 20
- 4
- 10

After Insertion of a 5:

- 15
- 7
- 20
- 4
- 10

Current:

- 5

After no rebalancing - just re-coloring the tree:

- 15
- 7
- 20
- 4
- 10

Note that AVL tree logic would have rebalanced!
Red/Black Tree Insertion

• (Note that we only continue if parent is red)
• If the color of the parent’s sibling is black (or null), the process gets more complex
• It is symmetric based on current’s parent’s sibling being the grandparent’s left or right child and current being its parent’s left or right child
Red/Black Tree Insertion

• If sibling is left child
  – If current is also left child, set current to parent and rotate current right around parent
  – Recolor and rotate left around grandparent

• If sibling is right child
  – If current is also right child, set current to parent and rotate current right around parent
  – Recolor and rotate right around grandparent
Red/Black Tree Insertion

• Insertion with Rebalancing (Example 1)

Initial Tree

After insertion of a 6 and re-coloring sub-tree

After rebalancing (Left rotation around 4)

Current

Parent of Current is black so that terminates the loop
Red/Black Tree Insertion

• Insertion with Rebalancing (Example 2)

Initial Tree

After insertion of a 5

After right rotation and re-coloring sub-tree

Current
Red/Black Tree Insertion

• Insertion with Rebalancing (Example 2)

After left rotation around 4
Move current to grandparent

Parent of Current is black
so that terminates the loop
Red/Black Tree Removal

• Start loop with current equal to found node
• Terminate loop when current is equal to root or color of current is red
  – If current’s sibling is red, rotate the sibling right around current’s parent
  – If both of sibling’s children are black, recolor
  – Else if sibling’s left child is black, rotate right child left around sibling
  – Rotate sibling right around current’s parent
Red/Black Tree Removal

• Removal with Rotations

Initial Tree

Sibling is Red

Element to be removed

Intermediate Step 1
Recolor for right rotation

Corresponds to upper right of Figure 11.19
Red/Black Tree Removal

• Removal with Rotations

Intermediate Step 2
After right rotation

Intermediate Step 3
Sibling’s left child is null/Black
but both children were not black

Corresponds to lower left of Figure 11.19
Red/Black Tree Removal

• Removal with Rotations

Intermediate Step 4
After left rotation around sibling

Intermediate Step 5
Recolor for right rotation
Red/Black Tree Removal

- Removal with Rotations

Intermediate Step 6
After right rotation
Set current to root to end loop

Remove 20 to get to Final Tree