**Rotations and balanced trees**

- **Height-balanced trees**
  - For every node, left and right subtree heights differ by at most 1
  - After insertion/deletion need to rebalance
  - Every operation leaves tree in a balanced state: *invariant property of tree*

- **Find deepest node that’s unbalanced then make sure:**
  - On path from root to inserted/deleted node
  - Rebalance at this unbalanced point only

**Are these trees height-balanced?**

---

**What is complexity?**

- Assume trees are “balanced” in analyzing complexity
  - Roughly half the nodes in each subtree
  - Leads to easier analysis

- **How to develop recurrence relation?**
  - What is $T(n)$?
  - What other work is done?

- **How to solve recurrence relation**
  - Plug, expand, plug, expand, find pattern
  - A real proof requires induction to verify correctness

---

**Balanced trees we won’t study**

- **B-trees are used when data is both in memory and on disk**
  - File systems, really large data sets
  - Rebalancing guarantees good performance both asymptotically and in practice. Differences between cache, memory, disk are important

- **Splay trees rebalance during insertion and during search, nodes accessed often more closer to root**
  - Other nodes can move further from root, consequences?
    - Performance for some nodes gets better, for others ...
  - No guarantee running time for a single operation, but guaranteed good performance for a sequence of operations, this is good *amortized* cost (vector push_back)

---

**Balanced trees we will study**

- **Both kinds have worst-case $O(\log n)$ time for tree operations**
- **AVL (Adel’son-Velskii and Landis), 1962**
  - Nodes are “height-balanced”, subtree heights differ by 1
  - Rebalancing requires per-node bookkeeping of height
  - [http://www.seanet.com/users/arsen/avltree.html](http://www.seanet.com/users/arsen/avltree.html)

- **Red-black tree uses same rotations, but can rebalance in one pass, contrast to AVL tree**
  - In AVL case, insert, calculate balance factors, rebalance
  - In Red-black tree can rebalance on the way down, code is more complex, but doable
  - Standard java.util.TreeMap/TreeSet use red-black
Rotation to rebalance

- When a node N (root) is unbalanced height differs by 2 (must be more than one)
  - Change N.left.left
  - doLeft
- Change N.left.right
  - doLeftRight
- Change N.right.left
  - doRightLeft
- First/last cases are symmetric
- Middle cases require two rotations
  - First of the two puts tree into doLeft or doRight

Node doLeft(Node root)
{
  Node newRoot = root.left;
  root.left = newRoot.right;
  newRoot.right = root;
  return newRoot;
}

Rotation up close (doLeft)

- Why is this called doLeft?
  - N will no longer be root, new value in left.left subtree
  - Left child becomes new root
- Rotation isn’t “to the left”, but rather “brings left child up”
  - doLeftChildRotate?

Node doLeft(Node root)
{
  Node newRoot = root.left;
  root.left = newRoot.right;
  newRoot.right = root;
  return newRoot;
}

Rotation to rebalance

- Suppose we add a new node in right subtree of left child of root
  - Single rotation can’t fix
  - Need to rotate twice
  - First stage is shown at bottom
    - Rotate blue node right
      - (its right child takes its place)
    - This is left child of unbalanced

Node doRight(Node root)
{
  Node newRoot = root.right;
  root.right = newRoot.left;
  newRoot.left = root;
  return newRoot;
}

Double rotation complete

- Calculate where to rotate and what case, do the rotations
  - Node doRight(Node root)
  {
    Node newRoot = root.right;
    root.right = newRoot.left;
    newRoot.left = root;
    return newRoot;
  }
  - Node doLeft(Node root)
  {
    Node newRoot = root.left;
    root.left = newRoot.right;
    newRoot.right = root;
    return newRoot;
  }