**Binary Search Tree Review**

* Data structure designed for **O(LogN) search**
* Can be viewed as a **recursive data structure (subtrees)**
* Have overall ordering: **values(L) < root < values(R)**
* Insert new nodes as leaves
  + We want to make it as balanced as possible via. rotation
* Delete from anywhere

**Rebalancing Trees**

An approach to balanced trees

* Insert into leaves as for simple BST, then periodically rebalance the tree

Questions: How frequently / when / how to rebalance?

* E.g. after X amount of insertions?

How to rebalance a BST?

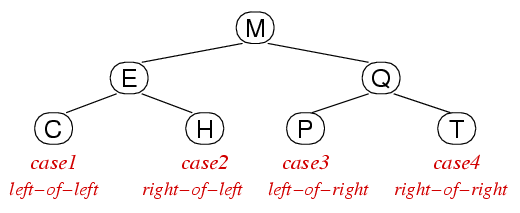
* **Move the median item to root (via. Rotations)**
  + Median / Midpoint item will always be the root – Roughly half of the nodes on the LEFT, half on the RIGHT
  + How do we find the median item?
    - We index into the tree and go to **N/2th item**
  + Move the Median Item to the root node via. rebalancing

**Rebalancing Trees – Analysis**

* Visit every node 🡪 O(n)
* Cost is not feasible to rebalance after each insertion
* When to rebalance?
  + After every k insertions
  + Whenever “imbalance” exceeds threshold

**Splay Trees**

* Another kind of “balanced tree”
* Splay tree insertion modifies **insertion-at-root** method:
  + Considers **Parent-Child-Grandchild** (three level analysis)
  + Performs double-rotations based on **P-C-G** orientations
* The idea: **appropriate double-rotations** **improve tree balance**
* Splay tree implementations also do **rotations-in-search**  
  (Modify tree when you’re doing a search – You move the item you just found to the top of the tree)  
  (Reasoning: studies shown that an item that you search for is likely to be searched for again)
  + Can provide **similar effects to periodic rebalance**
  + Improve balance, but **makes search more expensive**



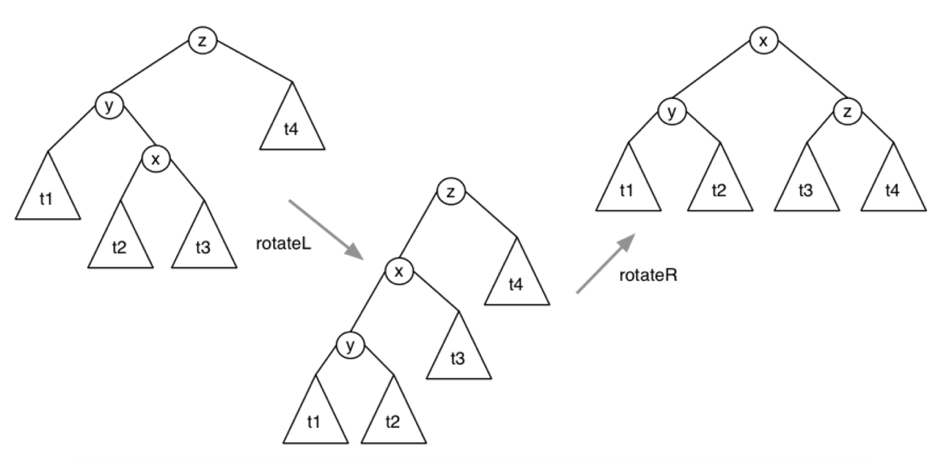
Case 1: Grandchild = left-child of left-child

Case 2: Grandchild = right-child of left-child

Case 3: Grandchild = left-child of right-child

Case 4: Grandchild = right-child of right-child

**Double-Rotation for Right-Child of Left-Child** (#1: Rotate X🡪LEFT then #2: Rotate X🡪RIGHT)



Gives good overall cost:

* Splay has higher insertion cost because of rotations
* Rotations potentially improve balance
* Potentially higher search costs (rotations)
* **Overall search cost is lower**(assuming recently searched items are often searched again)

Need empirical analysis to determine how much better.

**Worst case = O(n)**

**AVL Trees**

* Approach: AVL trees repairs balance as soon as imbalance is noticed.
  + Repairs are done locally (relative root node), not by the overall tree structure
* **Height / Depth of a tree is what makes searching expensive, not # of nodes**
  + Determining height / depth of tree is expensive
* Check if **LHS DEPTH > RHS DEPTH** or **RHS DEPTH > LHS DEPTH**
* Repaired by **Rotation**
  + LHS subtree too deep = rotateR
  + RHS subtree too deep = rotate