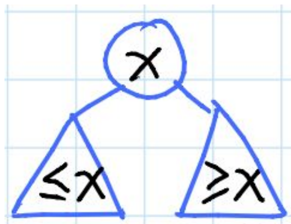
Lec6_AVL trees, AVL sort

Binary Search Trees (BSTs)

- Rooted binary tree
- Each node has (key / left pointer / right pointer / parent pointer)
- BST property



- Height of node = length (# edges) of the longest downward path to a leaf
- * For convenience height of Null ptr from leaves : -1

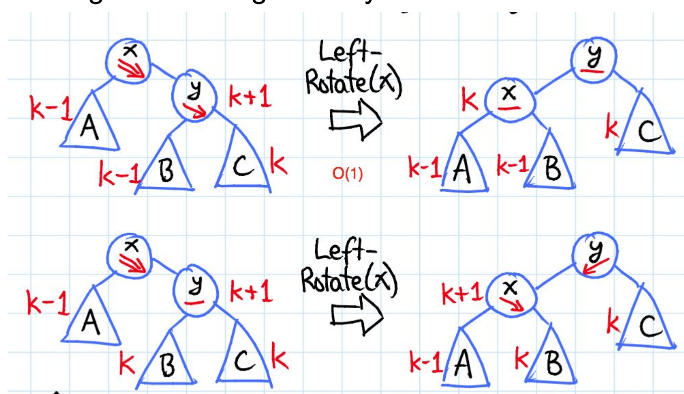
The importance of being balanced

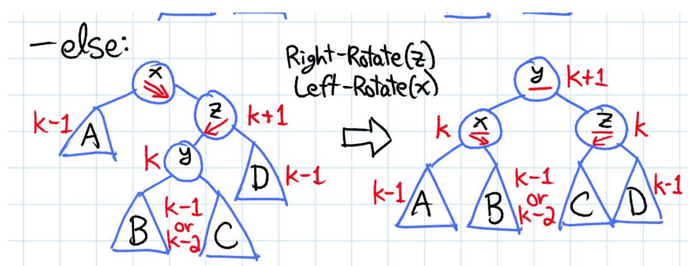
- Functions (insert, delete, min, max, next-larger, next-smaller, etc) from BST in O(h)
  h : height of tree
- balanced BST maintains h = O(lg(n))

AVL trees

- Properties
  - For every node, require heights of left & right children to differ by at most ±1
  - Treat NIL tree as height -1
  - Each node stores its **height (Data Structure Augmentation)**
- Balance
  - Worst when every node differs by 1

Let Nh = (min) # nodes in height-h AVL tree Approach 1) ⇒ $N(h) = N(h-1) + N(h-2) +1$ ⇒ $N(h) > 2N(h-2)$ ⇒ $N(h) > 2^{(h/2)}$ ⇒ $h < 2 \lg(N(h))$ Approach 2) ⇒ $N(h) > F(h)$ ⇒ $h < 1.440 \lg(n)$

- Insert
    - Steps
        - Insert as in simple BST
        - Work your way up tree, restoring AVL property (and updating heights as you go)
    - Balancing
        - Condition
          1. Suppose x is lowest node violating AVL
          2. Assume x is right heavy (left case symmetric)
          3. If x’s right child is right-heavy or balanced
        - First case
          If x’s right child is right heavy or balanced :
          

        - Second Case
          Else :
          

        - Then continue up to x’s grand parent, great grand parent …

AVL Sort

- Insert each item into AVL tree : Θ(nlg(n))
- In-order transversal : Θ(n)

Balanced Search Trees

1. AVL trees

2. B-trees / 2-3-4- trees
3. BB[a] trees
4. Red-black trees
5. Splay trees [A]
6. Skip lists [R]
7. Scapegoat trees [A]
8. Treaps [R]

R = use random numbers to make decisions fast with high probability
A = “amortized” : adding up costs for several operations => fast on average