

Lecture 7: Binary Trees II: AVL

Last Time and Today's Goal

| Sequence Data Structure | Operations $O(\cdot)$ | | | | |
|----------------------------|-----------------------|---------------------------|-----------------------------------|---------------------------------|---------------------------------|
| | Container | Static | Dynamic | | |
| | build(x) | get_at(i) set_at(i, x) | insert_first(x) delete_first() | insert_last(x) delete_last() | insert_at(i, x) delete_at(i) |
| Binary Tree | n | h | h | h | h |
| AVL Tree | n | $\log n$ | $\log n$ | $\log n$ | $\log n$ |

| Set Data Structure | Operations $O(\cdot)$ | | | | |
|-----------------------|-----------------------|----------|------------------------|--------------------------|------------------------------|
| | Container | Static | Dynamic | Order | |
| | build(x) | find(k) | insert(x) delete(k) | find_min() find_max() | find_prev(k) find_next(k) |
| Binary Tree | $n \log n$ | h | h | h | h |
| AVL Tree | $n \log n$ | $\log n$ | $\log n$ | $\log n$ | $\log n$ |

Height Balance

- How to maintain height $h = O(\log n)$ where n is number of nodes in tree?
 - A binary tree that maintains $O(\log n)$ height under dynamic operations is called **balanced**
 - There are many balancing schemes (Red-Black Trees, Splay Trees, 2-3 Trees, ...)
 - First proposed balancing scheme was the **AVL Tree** (Adelson-Velsky and Landis, 1962)

Rotations

- Need to reduce height of tree without changing its traversal order, so that we represent the same sequence of items
 - How to change the structure of a tree, while preserving traversal order? **Rotations!**

```

1      _____<D>_____
2      __<B>__       <E>
3      <A>   <C>   / \
4      / \   / \   /___\   rotate_right (<D>)    __<B>_____
5      /___\ /___\           =>          <A>   __<D>__
                                / \   <C>   <E>
                                /___\ / \   / \
                                /___\ /___\   rotate_left (<B>)    /___\ /___\

```

- A rotation relinks $O(1)$ pointers to modify tree structure and maintains traversal order

Rotations Suffice

- **Claim:** $O(n)$ rotations can transform a binary tree to any other with same traversal order.
 - **Proof:** Repeatedly perform last possible right rotation in traversal order; resulting tree is a canonical chain. Each rotation increases depth of the last node by 1. Depth of last node in final chain is $n - 1$, so at most $n - 1$ rotations are performed. Reverse canonical rotations to reach target tree. \square
 - Can maintain height-balance by using $O(n)$ rotations to fully balance the tree, but slow :(
 - We will keep the tree balanced in $O(\log n)$ time per operation!
-

AVL Trees: Height Balance

- AVL trees maintain **height-balance** (also called the **AVL Property**)
 - A node is **height-balanced** if heights of its left and right subtrees differ by at most 1
 - Let **skew** of a node be the height of its right subtree minus that of its left subtree
 - Then a node is height-balanced if its skew is $-1, 0$, or 1
- **Claim:** A binary tree with height-balanced nodes has height $h = O(\log n)$ (i.e., $n = 2^{\Omega(h)}$)
- **Proof:** Suffices to show fewest nodes $F(h)$ in any height h tree is $F(h) = 2^{\Omega(h)}$

$$F(0) = 1, F(1) = 2, F(h) = 1 + F(h-1) + F(h-2) \geq 2F(h-2) \implies F(h) \geq 2^{h/2} \quad \square$$

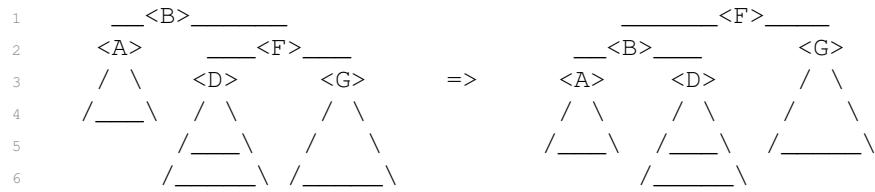
- Suppose adding or removing leaf from a height-balanced tree results in imbalance
 - Only subtrees of the leaf's ancestors have changed in height or skew
 - Heights changed by only ± 1 , so skews still have magnitude ≤ 2
 - **Idea:** Fix height-balance of ancestors starting from leaf up to the root
 - Repeatedly rebalance lowest ancestor that is not height-balanced, wlog assume skew 2

- **Local Rebalance:** Given binary tree node $\langle B \rangle$:

- whose skew 2 and
- every other node in $\langle B \rangle$'s subtree is height-balanced,
- then $\langle B \rangle$'s subtree can be made height-balanced via one or two rotations
- (after which $\langle B \rangle$'s height is the same or one less than before)

- **Proof:**

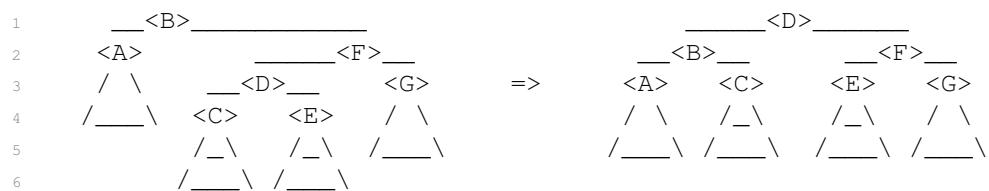
- Since skew of $\langle B \rangle$ is 2, $\langle B \rangle$'s right child $\langle F \rangle$ exists
- **Case 1:** skew of $\langle F \rangle$ is 0 or **Case 2:** skew of $\langle F \rangle$ is 1
 - * Perform a left rotation on $\langle B \rangle$



- * Let $h = \text{height}(\langle A \rangle)$. Then $\text{height}(\langle G \rangle) = h + 1$ and $\text{height}(\langle D \rangle)$ is $h + 1$ in Case 1, h in Case 2
- * After rotation:
 - the skew of $\langle B \rangle$ is either 1 in Case 1 or 0 in Case 2, so $\langle B \rangle$ is height balanced
 - the skew of $\langle F \rangle$ is -1 , so $\langle F \rangle$ is height balanced
 - the height of $\langle B \rangle$ before is $h + 3$, then after is $h + 3$ in Case 1, $h + 2$ in Case 2

-
- **Case 3:** skew of $\langle F \rangle$ is -1 , so the left child $\langle D \rangle$ of $\langle F \rangle$ exists

- * Perform a right rotation on $\langle F \rangle$, then a left rotation on $\langle B \rangle$



- * Let $h = \text{height}(\langle A \rangle)$. Then $\text{height}(\langle G \rangle) = h$ while $\text{height}(\langle C \rangle)$ and $\text{height}(\langle E \rangle)$ are each either h or $h - 1$
- * After rotation:
 - the skew of $\langle B \rangle$ is either 0 or -1 , so $\langle B \rangle$ is height balanced
 - the skew of $\langle F \rangle$ is either 0 or 1, so $\langle F \rangle$ is height balanced
 - the skew of $\langle D \rangle$ is 0, so D is height balanced
 - the height of $\langle B \rangle$ is $h + 3$ before, then after is $h + 2$

- **Global Rebalance:** Add or remove a leaf from height-balanced tree T to produce tree T' . Then T' can be transformed into a height-balanced tree T'' using at most $O(\log n)$ rotations.
 - **Proof:**
 - Only ancestors of the affected leaf have different height in T' than in T
 - Affected leaf has at most $h = O(\log n)$ ancestors whose subtrees may have changed
 - Let $\langle x \rangle$ be lowest ancestor that is not height-balanced (with skew magnitude 2)
 - If a leaf was added into T :
 - * Insertion increases height of $\langle x \rangle$, so in Case 2 or 3 of Local Rebalancing
 - * Rotation decreases subtree height: balanced after one rotation
 - If a leaf was removed from T :
 - * Deletion decreased height of one child of $\langle x \rangle$, not $\langle x \rangle$, so only imbalance
 - * Could decrease height of $\langle x \rangle$ by 1; parent of $\langle x \rangle$ may now be imbalanced
 - * So may have to rebalance every ancestor of $\langle x \rangle$, but at most $h = O(\log n)$ of them
 - So can maintain height-balance using only $O(\log n)$ rotations after insertion/deletion!
 - But requires us to evaluate whether possibly $O(\log n)$ nodes were height-balanced
-

Computing Height

- How to tell whether node $\langle x \rangle$ is height-balanced? Compute heights of subtrees!
- How to compute the height of node $\langle x \rangle$? Naive algorithm:
 - Recursively compute height of the left and right subtrees of $\langle x \rangle$
 - Add 1 to the max of the two heights
 - Runs in $\Omega(n)$ time, since we recurse on every node :(
- **Idea:** Augment each node with the height of its subtree! (Save for later!)
- Height of $\langle x \rangle$ can be computed in $O(1)$ time from the heights of its children:
 - Look up the stored heights of left and right subtrees in $O(1)$ time
 - Add 1 to the max of the two heights
- During dynamic operations, we must **Maintain** our augmentation as the tree changes shape
- Recompute subtree augmentations at every node whose subtree changes:
 - Update relinked nodes in a rotation operation in $O(1)$ time (ancestors don't change)
 - Update all ancestors of an inserted or deleted node in $O(h)$ time by walking up the tree

Steps to Augment a Binary Tree

- In general, to augment a binary tree with a **subtree property** P , you must:
 - State the subtree property $P(<X>)$ you want to store at each node $<X>$
 - Show how to compute $P(<X>)$ from the augmentations of $<X>$'s children in $O(1)$ time
 - Then stored property $P(<X>)$ can be maintained without changing dynamic operation costs
-

Application: Sequence

- For sequence binary tree, we needed to know subtree **sizes**
 - For just inserting/deleting a leaf, this was easy, but now need to handle rotations
 - Subtree size is a subtree property, so can maintain via augmentation
 - Can compute size from sizes of children by summing them and adding 1
-

Conclusion

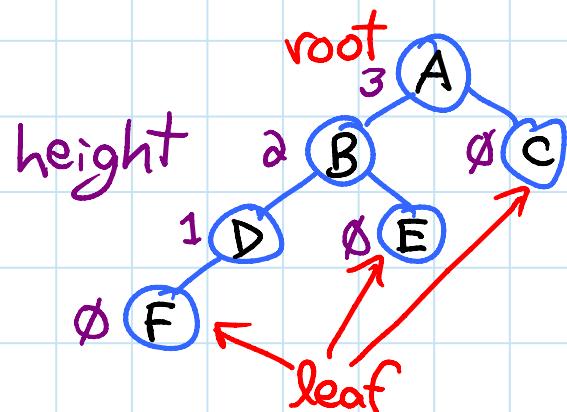
- Set AVL trees achieve $O(\lg n)$ time for all set operations, except $O(n \log n)$ time for build and $O(n)$ time for iter
 - Sequence AVL trees achieve $O(\lg n)$ time for all sequence operations, except $O(n)$ time for build and iter
-

Application: Sorting

- Any Set data structure defines a sorting algorithm: build (or repeatedly insert) then iter
- For example, Direct Access Array Sort from Lecture 5
- AVL Sort is a new $O(n \lg n)$ -time sorting algorithm

TODAY: Binary Trees II: AVL

- subtree augmentation
- sequence binary tree via subtree sizes
- height balance (AVL)
 - rotations - rebalancing
 - height augmentation

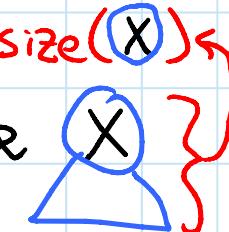
Recall: binary tree

- height(X) = max. depth of node within subtree(X) = length of longest downward path
 - height $\emptyset \Leftrightarrow$ leaf
- h = height(root) = max. depth of node (leaf)
- traversal order = traverse left subtree, root, traverse right subtree
- subtree insert, delete, first/last, pred./successor in $O(h)$ time/op.

Application: set binary tree (Binary Search Tree)

- traversal order = increasing key order
 \Rightarrow insert, delete, find/find-prev/next in $O(h)$ /op.

Application: Sequence binary tree

- Suppose we knew size of every subtree 

$\hookrightarrow \# \text{ nodes}$

Subtree-at(node, i): *i*th node in traversal order of subtree(node)

- $n_L = \text{size}(\text{node.left})$

- if $i < n_L$: subtree-at(node.left, i)
- if $i = n_L$: return node
- if $i > n_L$: subtree-at(node.right, $i - n_L - 1$)

get-at(i): subtree-at(root, i).item

for node

set-at(i, x): subtree-at(root, i).item = x

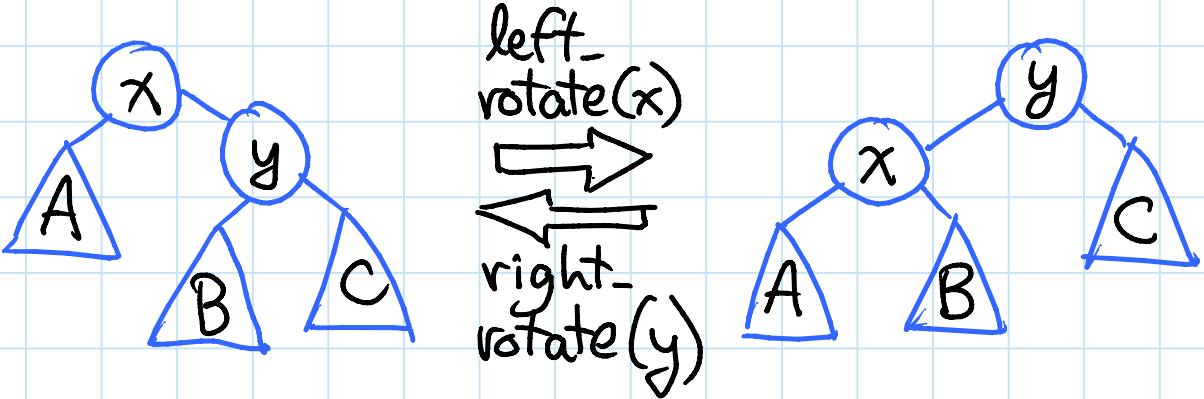
Subtree augmentation:

- maintain extra properties of nodes (subtrees)
- Subtree property of a node can be computed in $O(1)$ time given property values of node's children (& node)
 - e.g. sum/product/min/max of keys or other variable of items in subtree(node)
 - $\text{node.size} = 1 + \text{node.left.size} + \text{node.right.size}$
where $\text{None.size} = \emptyset$
 - NOT e.g. node's seq. index, or node's depth
- whenever insert/delete a leaf (e.g.), trigger augmentation update for all $O(h)$ ancestors in order up the tree
 - = subtrees changed by that update
 - \Rightarrow get/set/insert/delete-at in $O(h)$

Balanced binary tree maintains $h = O(\lg n)$
⇒ all set & sequence ops. run in $O(\lg n)$
except build & iter

- need to re-arrange tree while preserving traversal order
(to represent the same set/sequence)
- how?

Rotations:



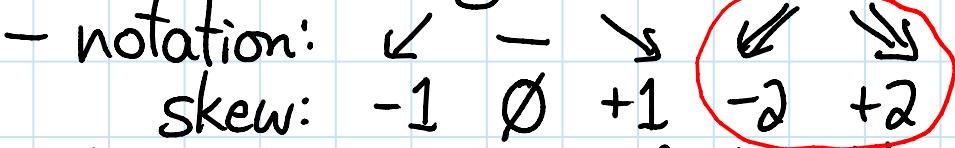
- preserves traversal order:

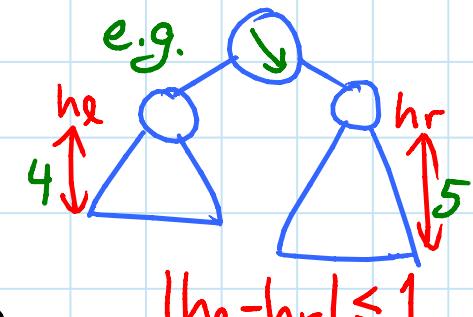


- $O(1)$ pointer changes ⇒ $O(1)$ time
- need to update subtree augmentations of x & y (& their ancestors) too

First (& still one of simplest) balanced binary tree:

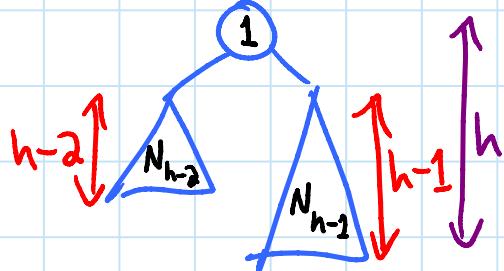
AVL trees: [Adel'son-Vel'skiy & Landis 1962]

- require every node to be height-balanced: heights of children differ by at most ± 1
- define $\text{height}(\text{None}) = -1$
- $\text{skew}(\text{node}) = \text{height}(\text{node.right}) - \text{height}(\text{node.left}) \in \{-1, 0, 1\}$
- notation: 
- each node stores `node.height` via subtree augmentation: $\text{node.height} = 1 + \max\{\text{node.left.height}, \text{node.right.height}\}$
(alternatively, can just store skews)



Height-balanced \Rightarrow Balanced

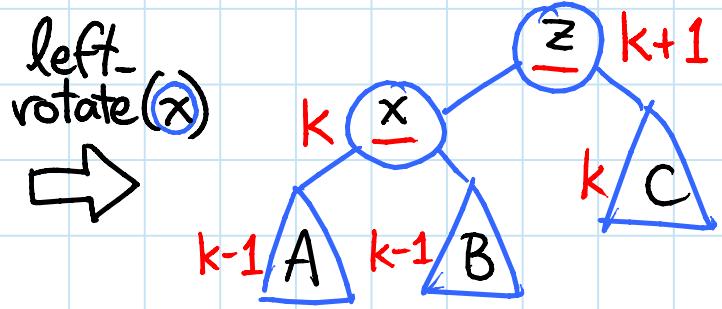
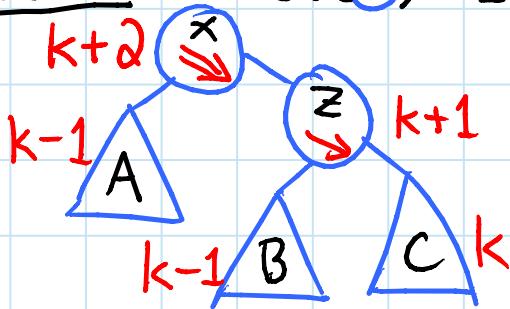
- worst when every node has skew ± 1
- let $N_h = \min.$ # nodes in height- h AVL tree
- $\Rightarrow N_h = N_{h-1} + N_{h-2} + 1$
- $\Rightarrow N_h > 2N_{h-2}$
- $\Rightarrow N_h > 2^{h/2}$
- $\Rightarrow h < 2 \lg N_h$



AVL rebalancing:

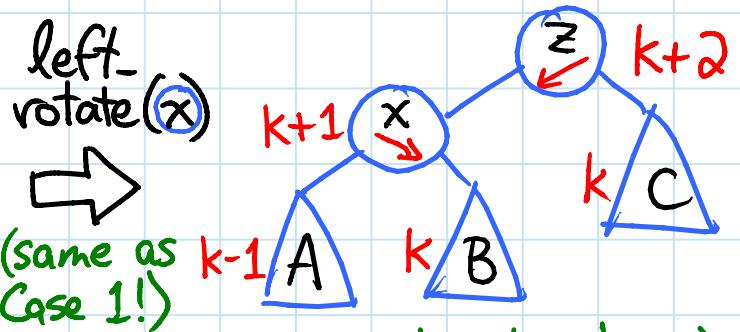
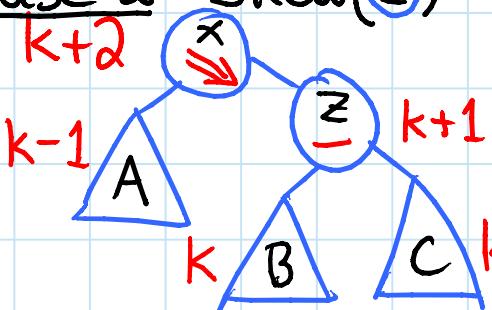
- binary tree insert/delete adds/removes a leaf, affecting ancestors' subtrees/heights by ± 1
 $\Rightarrow \text{Iskew} \leq 2$ still
 - for each unbalanced ancestor x in order up tree:
 - assume $\text{skew}(x) = 2$ (if -2 , flip left/right)
 - $\Rightarrow x$ has right child z , balanced by induction

Case 1: $\text{skew}(\bar{z}) = 1$



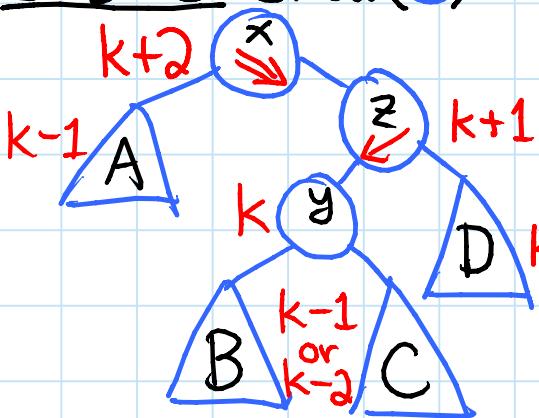
Case 2: $\text{skew}(z) = \emptyset$

can only happen during Delete



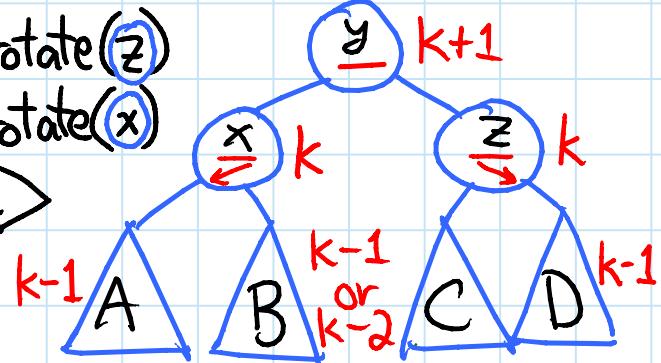
(next check z)

Case 3: $\text{skew}(z) = -1$



right-rotate(z)

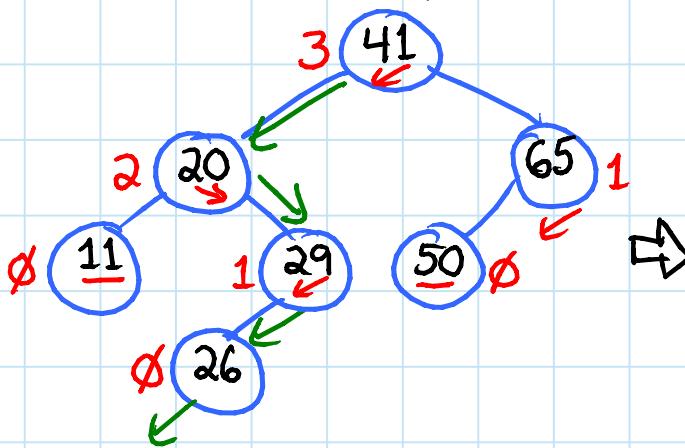
left-rotate(x)



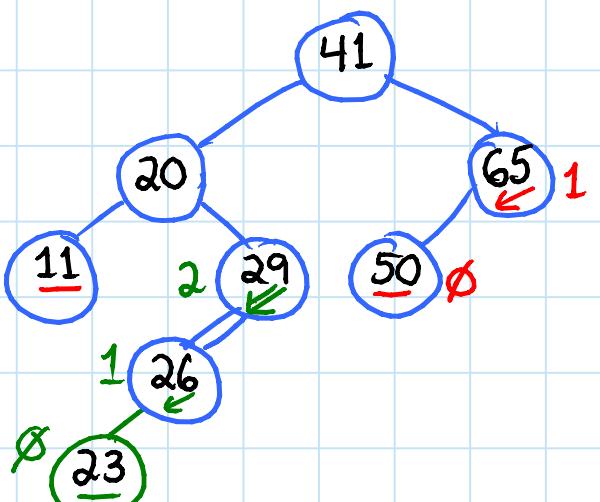
(next check y)

Example: set AVL

insert(23)

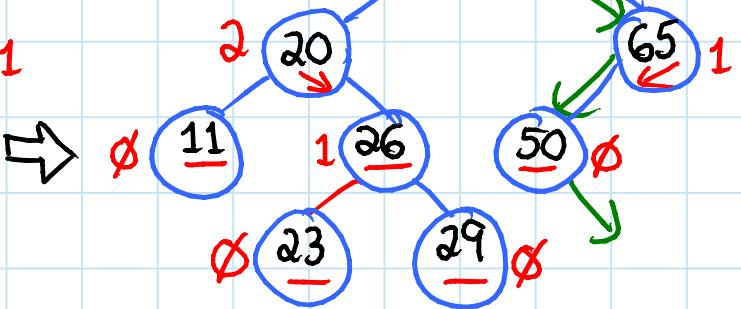
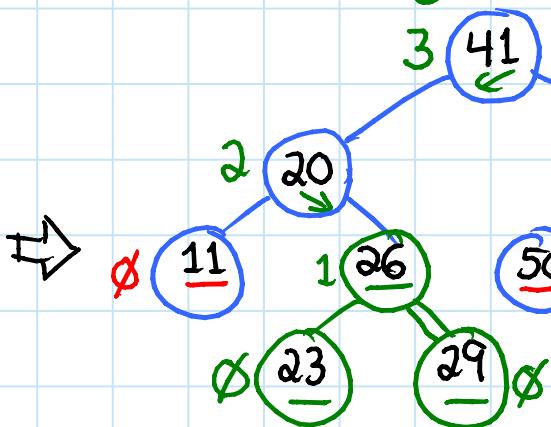


$x=29$: left-left case



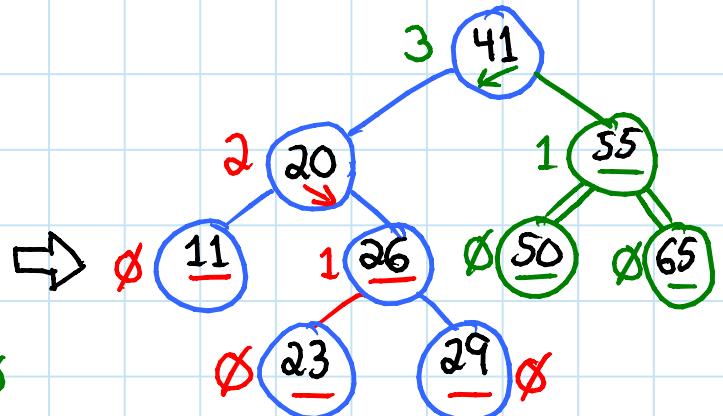
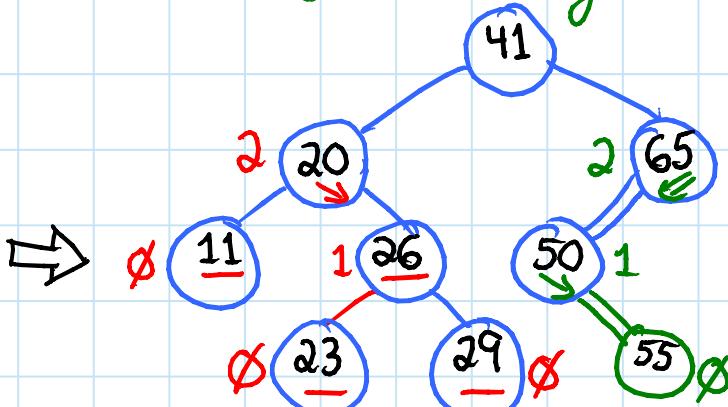
done.

insert(55)



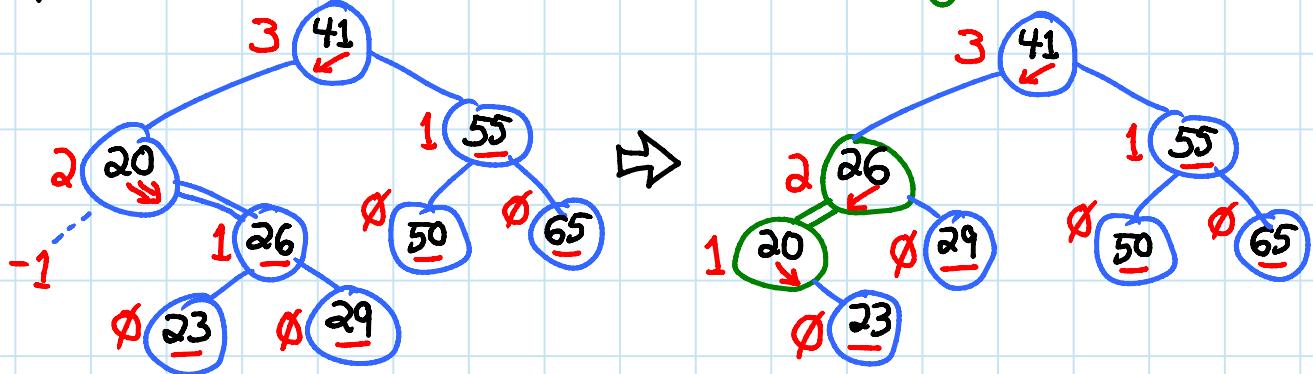
$x=65$: left-right case

done.



- in fact never need > 1 rotation pair for an Insert:
height of subtree after rebalancing =
height of subtree before insert

Example: $\text{delete}(11)$



Wow! $O(\lg n)$ for everything!

Set data structure \rightarrow Sorting algorithm

① $\text{build}(A)$ e.g. for x in A :

$\text{insert}(x)$

② $\text{iter_ord}()$ \rightarrow key order

- e.g. Direct Access Array Sort [L5]

AVL sort is $\Theta(n \lg n)$

- $\Theta(n \lg n)$ for ①

- $\Theta(n)$ for ②

↳ effectively maintaining the sorted order
of a dynamic set!

Balanced trees:

- AVL trees
- B-trees / 2-3-4 trees
- BB $[\alpha]$ trees
- red-black trees
- (A) - splay trees
- (R) - skip lists
- (A) - scapegoat trees
- (R) - treaps

there are many!

[Adel'son-Velskii & Landis 1962]

[Bayer & McCreight 1972] [CLRS 18]

[Nievergelt & Reingold 1973]

[CLRS ch. 13]

[Sleator & Tarjan 1985]

[Pugh 1989]

[Galperin & Rivest 1993]

[Seidel & Aragon 1996]

- (R) = use random numbers to make decisions
fast with high probability
- (A) = "amortized": adding up costs for
several operations \Rightarrow fast on average

e.g. Splay trees are a current research topic

- see 6.854 (Advanced Algorithms)
& 6.851 (Advanced Data Structures)