

## Lecture 7: Binary Trees II

### Last Time

- Learned to navigate the in-order traversal of a binary tree
- Learned to change tree structure by adding and removing leaves, and performing rotations

### Today

- Keep tree balanced after leaf insertions and deletions, i.e.,  $h = O(\log n)$
- Implement efficient Set and Sequence Interfaces using a Binary Tree

### 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** (1962)
- 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, to skew of magnitude at most 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

## Rebalancing

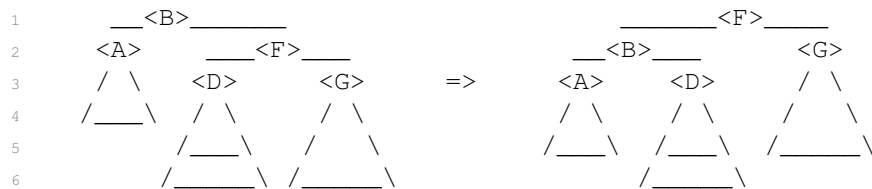
- **Local Rebalance:** Given binary tree node  $\langle B \rangle$ :

- whose skew is 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  $\langle B \rangle$



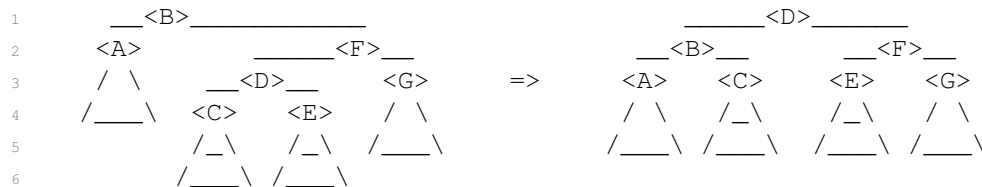
\* Let  $h = H(A)$ , then  $H(G) = h + 1$  and  $H(D)$  is  $h + 1$  in Case 1,  $h$  in Case 2

\* 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  $h = H(A)$ , then  $H(G) = h$  while  $H(C)$  and  $H(E)$  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
  - 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

- **Idea! Sequence Tree:** Traversal order is sequence order
  - To find an index, could just iterate through traversal order, but that's bad,  $O(n)$
  - However, if we could compute a subtree's size in  $O(1)$ , can index in  $O(h)$  time
    - How? Check the size  $n_L$  of the left subtree and compare to  $i$
    - If  $i < n_L$ , recurse on the left subtree
    - If  $i > n_L$ , recurse on the right subtree with  $i' = i - n_L - 1$
    - Otherwise,  $i = n_L$ , and you've reached the desired node!
  - Maintain the size of each node's subtree at the node via **augmentation!**
    - Can compute size from sizes of children by summing them and adding 1
  - Sequence operations follow directly from a fast `subtree_node_at(i)` operation
  - Naively, `build(A)` takes  $O(n \log n)$  time, but can be done in  $O(n)$  time
  - You will go over their implementations in recitation
- 

## Application: Set

- **Idea! Binary Search Tree:** Traversal order is sorted order increasing by key
- Then can find the node with key  $k$  in node  $<X>$ 's subtree in  $O(h)$  time:
  - If  $k$  is smaller than the key at  $<X>$ , recurse in left subtree (or return `None`)
  - If  $k$  is larger than the key at  $<X>$ , recurse in right subtree (or return `None`)
  - Otherwise, return the item stored at  $<X>$
- Other Set operations follow a similar pattern
- You will go over their implementations in recitation