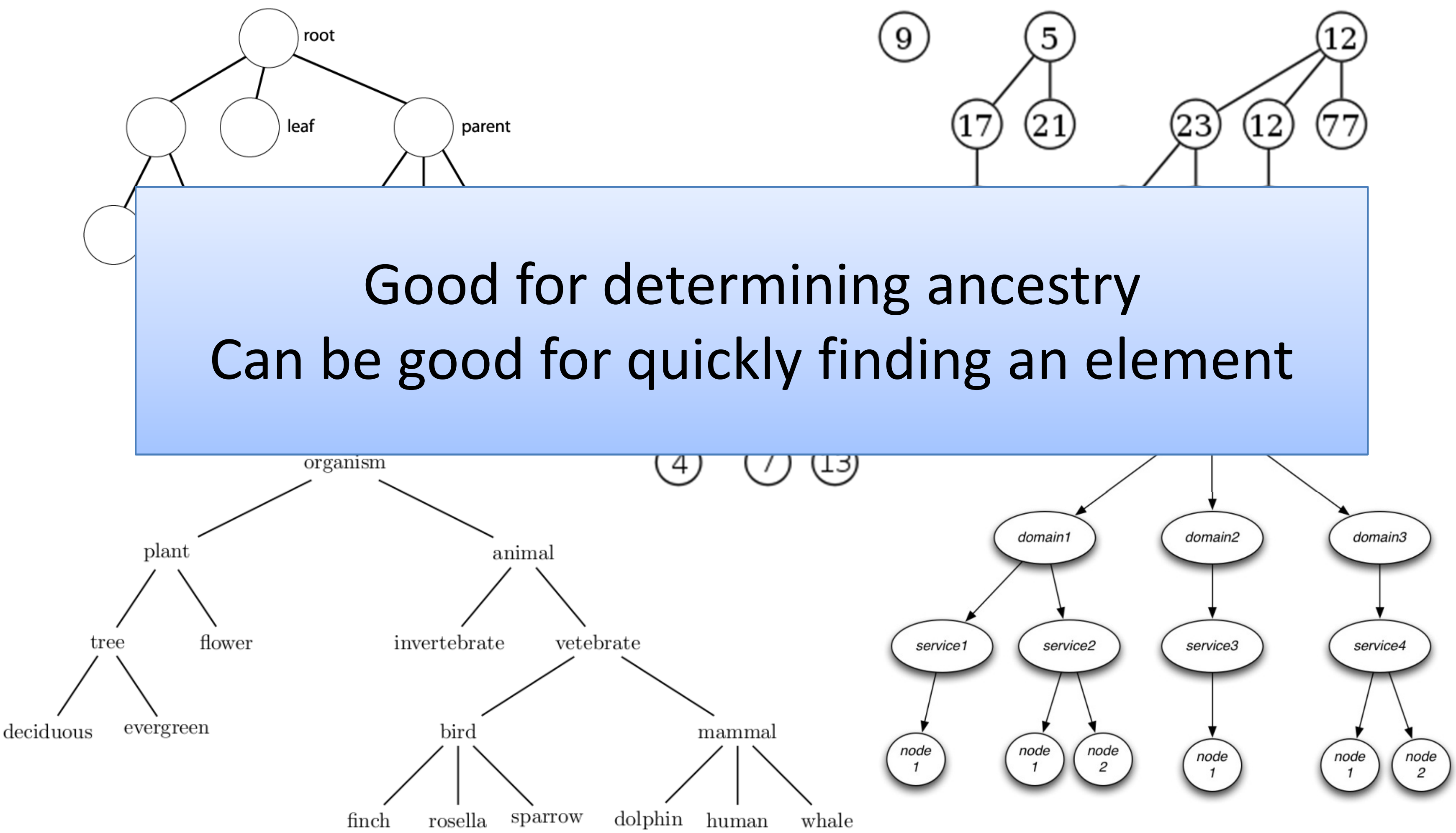
Augmenting AVL trees

How we've thought about trees so far



Good for determining ancestry

Can be good for quickly finding an element

Other kinds of uses?

- Any thoughts?
- Finding a minimum/maximum…
  - (heaps are probably just as good or better)

- Finding an average?
- **More** complicated things?!!!11one

Enter: idea of **augmenting** a tree

Augmenting

- Can quickly compute many global properties that seem to need knowledge of the whole tree!
- Examples:
  - size of any sub-tree
  - height of any sub-tree
  - averages of keys/values in a sub-tree
  - min+max of keys/values in any sub-tree, …
- Can quickly compute **any function f(u)** so long as you only need to know f(u.left) and f(u.right)!

Augmenting an AVL tree

- Can augment any kind of tree
- Only balanced trees are guaranteed to be fast
- **After augmenting an AVL tree to compute f(u), we can still do all operations in O(lg n)!**

Simple first example

- We are going to do one
- Then, you will help with

A regular AVL tree already does this

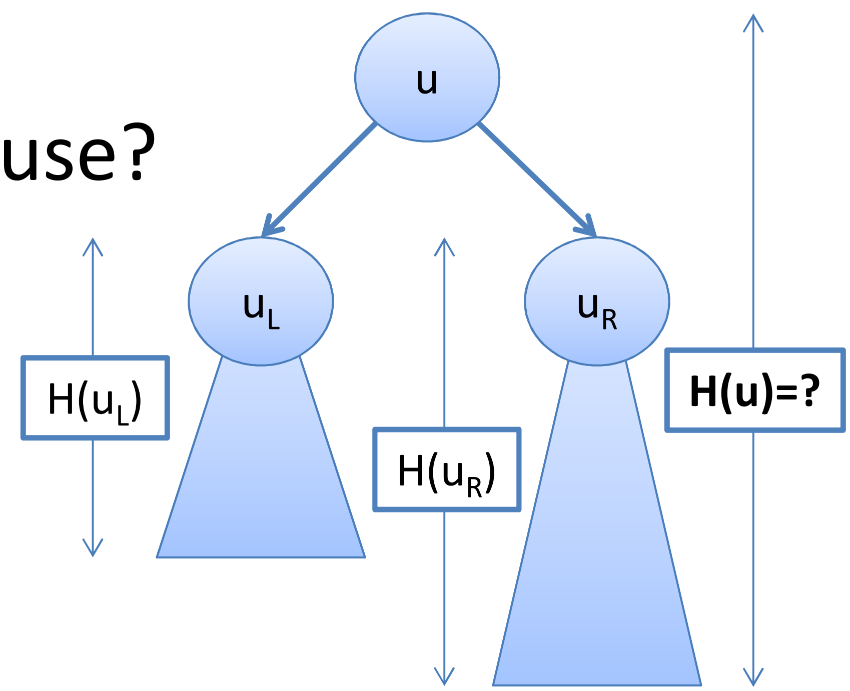
- Problem: augment an AVL tree so we can do:
  - **Insert(key)**: add key in O(lg n)
  - **Delete(key)**: remove key in O(lg n)
  - **Height(node)**: get height of sub-tree rooted at **node** in O(1)

How do we do this?

Store some extra data at each node… but what?

Can we compute this function quickly?

- Function we want to compute: Height(u) = H(u)
- If someone gives us $H(u_L)$ and $H(u_R)$, can we compute H(u)?
- What formula should we use?



- **If u is a leaf then**
  - H(u) = 0
- **Else**
  - $H(u) = \max\{H(u_L), H(u_R)\}+1$

Augmenting AVL tree to compute H(u)

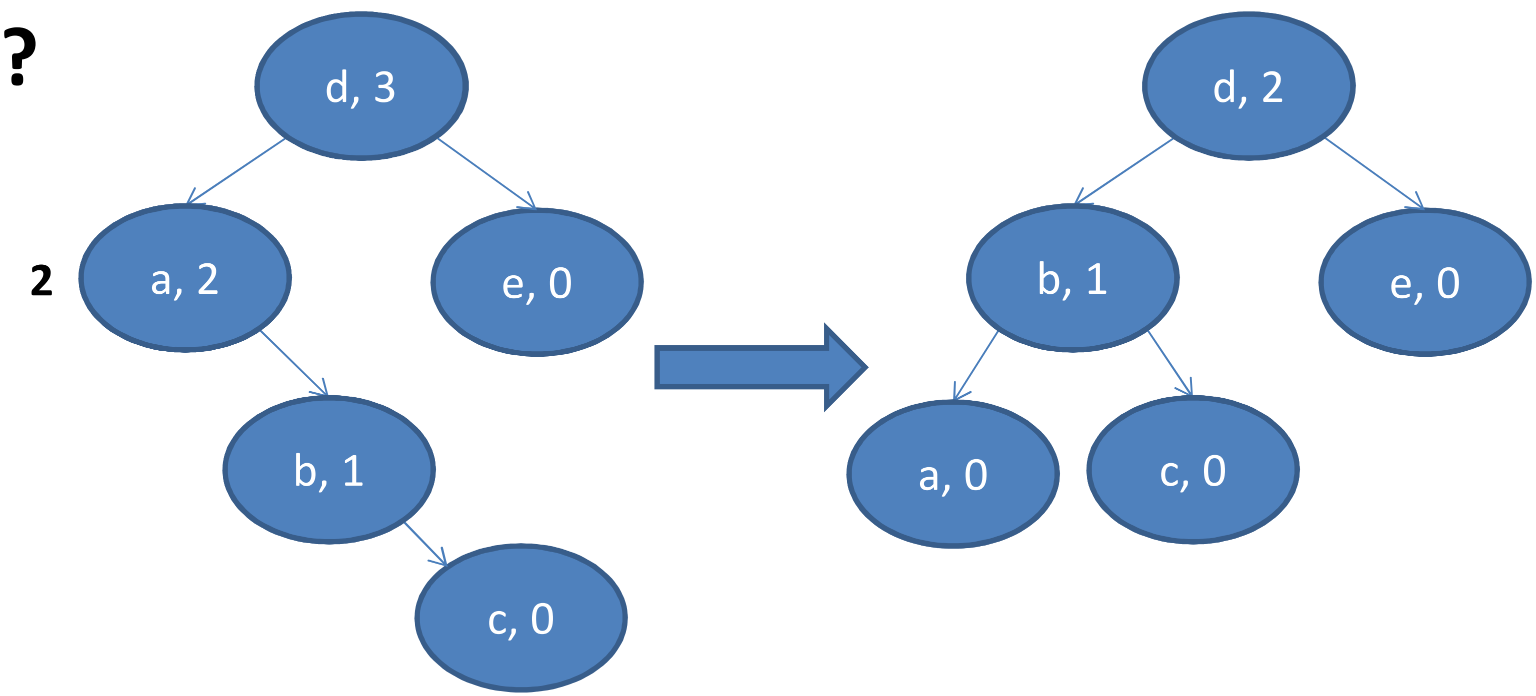
- Each node **u** contains
  - **key**: the key
  - **left**, **right**: child pointers
  - **h**: height of sub-tree rooted at **u**

The usual stuff…

Secret sauce!

- **How?**

Insert(d)
Insert(a)
Insert(e)
Insert(b)
Insert(c)



Algorithm idea:

- From the last slide, we develop an algorithm

Insert(key):

1 BST search for where to put **key**

2 Insert **key** into place like in a regular AVL tree

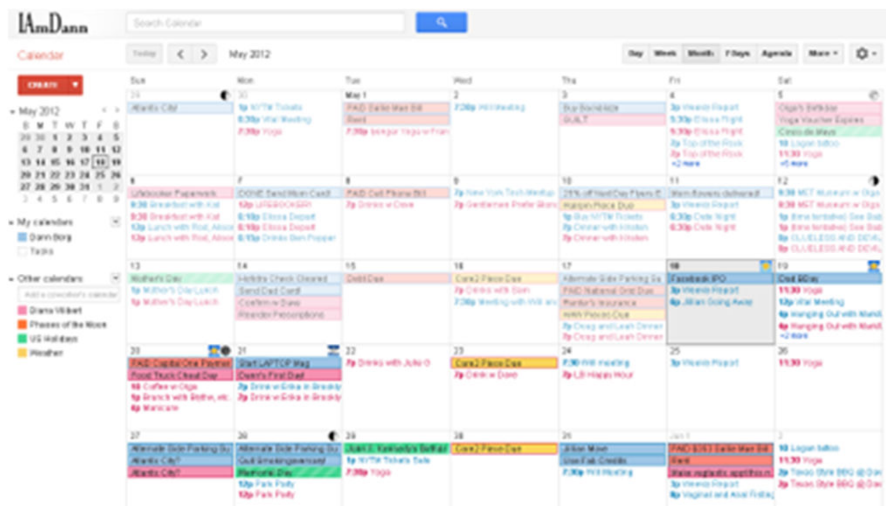
3 Fix balance factors and rotate as you would in AVL insert, but **fix heights at the same time**.

(Remember to fix heights all the way to the root. Don’t stop before reaching the root!)

- (When you rotate, remember to fix heights of all nodes involved, just like you fix balance factors!)

Harder problem: scheduling conflicts

- Your calendar contains a bunch of time **intervals [lo,hi]** where you are busy
- We want to be able to quickly tell whether a new booking conflicts with an earlier booking.



Breaking the problem down

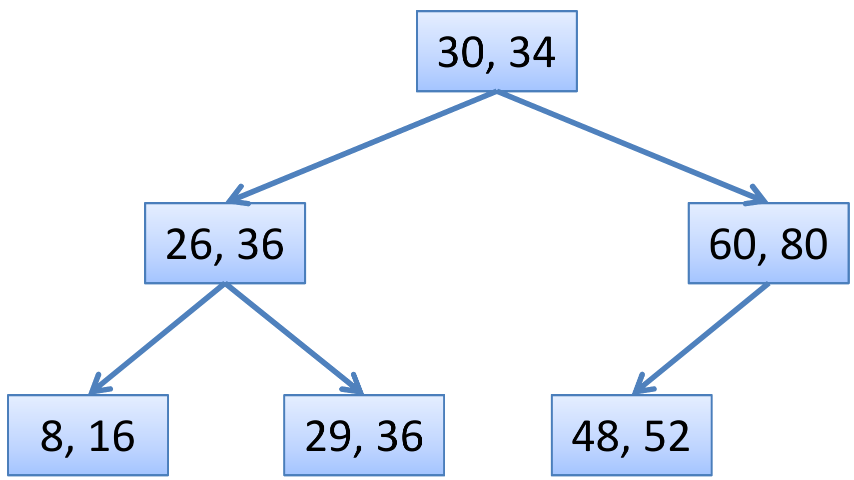
- You must **design a data structure** D to efficiently do:
  - **Insert(D; x):** Insert interval x into D.
  - **Delete(D; x):** Delete interval x from D.
  - **Search(D; x):** If D contains an interval that overlaps with x, return *any* such interval. Otherwise, return null.

The hard part

- **All functions must run in O(lg n)**

Figuring out the data structure - 1

- **Iterative process; <u>HARD</u> to get right the first time!**
- Need a way to insert intervals into the tree
  - Use low end-point of interval as the key
- Example tree:



Figuring out the data structure - 2

- **What function do we want to compute?**
  - Does an interval x intersect any interval in the tree?
- **What info should we store at each node u?**
  - ***Mhi(u) =*** *Maximum* ***high endpoint*** *of any node in the subtree.*
- **How can we use the info stored at each node to compute the desired function** (by looking at a <u>small</u> number of nodes)?
- Start by computing whether an interval x intersects any interval **in a subtree**.

Algorithm for Search within a subtree

Returns an interval in the subtree rooted at u that intersects [lo, hi]

Search(lo, hi, u):

if u is null then return null

Algorithm for Search within a subtree

Returns an interval in the subtree rooted at u that intersects [lo, hi]

Search(lo, hi, u):

if u is null then return null

if [lo, hi] intersects [lo(u), hi(u)] then return [lo(u), hi(u)]

Algorithm for Search within a subtree

Returns an interval in the subtree rooted at u that intersects [lo, hi]

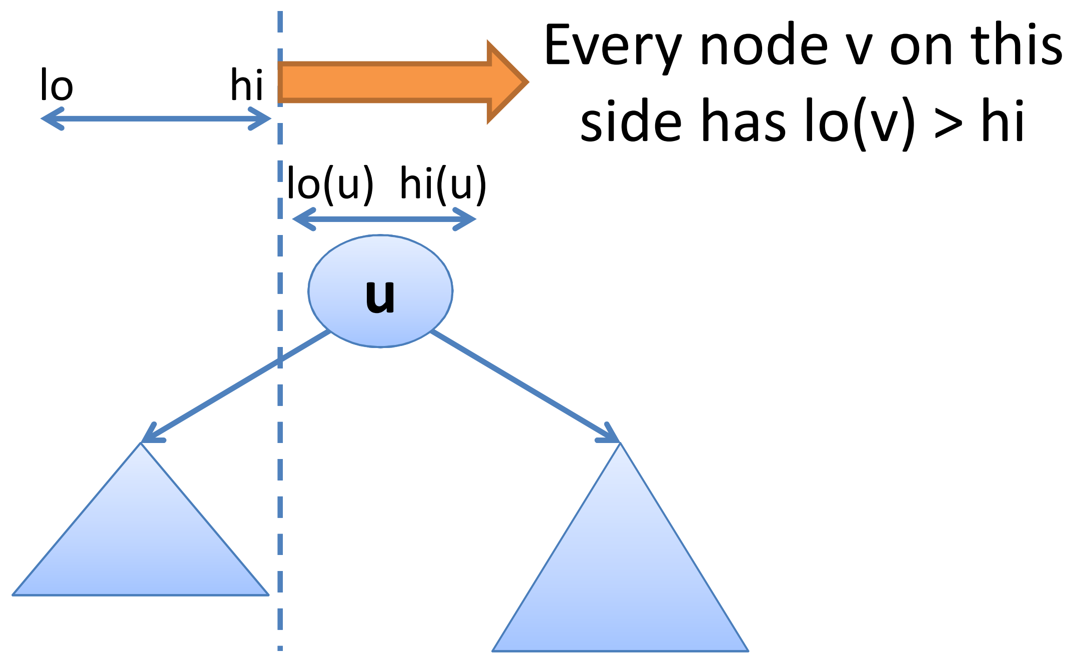
Search(lo, hi, u):

if u is null then return null

if [lo, hi] intersects [lo(u), hi(u)] then return [lo(u), hi(u)]

else (no intersection)

if lo < lo(u) return Search(lo, hi, left(u))



Algorithm for Search within a subtree

Returns an interval in the subtree rooted at u that intersects [lo, hi]

Search(lo, hi, u):

if u is null then return null

if [lo, hi] intersects [lo(u), hi(u)] then return [lo(u), hi(u)]

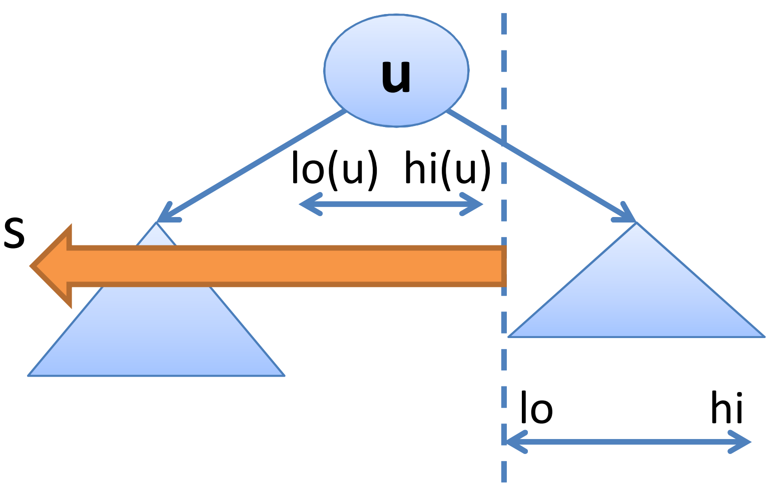
else (no intersection)

if lo < lo(u) return Search(lo, hi, left(u))

else (lo ≥ lo(u))

if lo > Mhi(left(u)) then return Search(lo, hi, right(u))

Every node v on this side has hi(v) < lo



Algorithm for Search within a subtree

Returns an interval in the subtree rooted at u that intersects [lo, hi]

Search(lo, hi, u):

if u is null then return null

if [lo, hi] intersects [lo(u), hi(u)

else (no intersection)

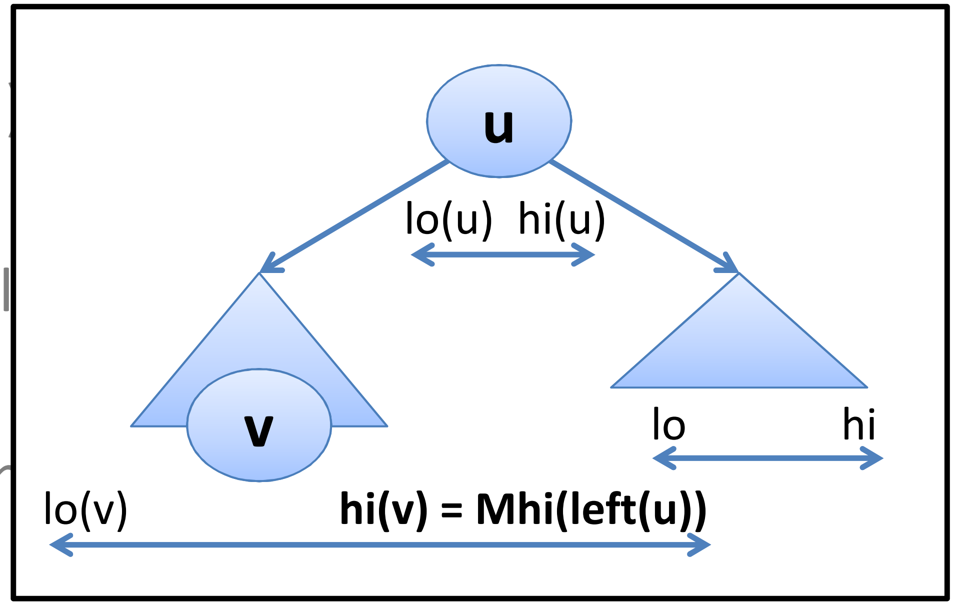
if lo < lo(u) return Search(l

else (lo ≥ lo(u))

if lo > Mhi(left(u)) then

else (lo ≤ Mhi(left(u))

return Search(lo, hi, left(u))



Final algorithm for Search

```
Search(lo, hi, u):
    if u is null then return null
    if [lo, hi] intersects [lo(u), hi(u)] then return [lo(u), hi(u)]
    else (no intersection)
        if lo < lo(u) return Search(lo, hi, left(u))
        else (lo ≥ lo(u))
            if lo > Mhi(left(u)) then return Search(lo, hi, right(u))
            else (lo ≤ Mhi(left(u))
                return Search(lo, hi, left(u))

Search(D, x=[lo, hi]):
    return Search(lo, hi, root(D))
```

Algorithms for Insert and Delete

- Insert(D, x=[lo, hi]):
  - Do regular AVL insertion of key **lo**, also storing **hi**.
  - Set Mhi of the new node to **hi**.
  - Fix balance factors and perform rotations as usual, but also update Mhi(u) whenever you update the balance factor of a node u.
  - Update Mhi(u) for all ancestors, and for every node involved in a rotation, using formula: Mhi(u) = max{hi(u), Mhi(left(u)), Mhi(right(u))}.
- Delete(D, x=[lo, hi]): similar to Insert

Why O(lg n) time?

- Insert/Delete: normal AVL operation = O(lg n)
  - PLUS: update Mhi(u) for each u on path to the root
    - Length of this path ≤ tree height, so O(lg n) in an AVL tree
  - PLUS: update Mhi(u) for each node involved in a rotation
    - At most O(lg n) rotations (one per node on the path from the root ≤ tree height)
    - Each rotation involves a constant number of nodes
    - Therefore, constant times O(lg n), which is O(lg n).
- Search
  - Constant work + recursive call on a child
  - Single recursive call means O(tree height) = O(lg n)