Reminder: test 2 in lab today, don't be late!

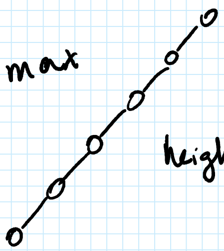
TODAY: how to rebalance a BST (at last!)

REBALANCING A BST

We previously observed that for a BST of height h and size n, the "get" operation took $O(h)$.

Q: What is the maximum height of a BST of size n?
What is the minimum height of a BST of size n?

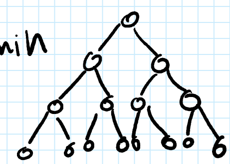
max



height n

worst case

min



height $O(\log_2(n))$

best case

A BST of size n has height h anywhere between $O(\log_2(n))$ and n.

We'd like to guarantee that the BST is balanced so that all operations which are $O(h)$ are $O(\log_2(n))$.

AVL tree (Adelson-Velskii and Landis)

Idea: add another invariant to the BST.

An AVL tree:
- is a BST (binary & BST property as invariant)
- for every node, the height of its left subtree and the height of its right subtree differ by at most 1.

So now we need a way to keep track of the heights of subtrees. Let's store it within each node!

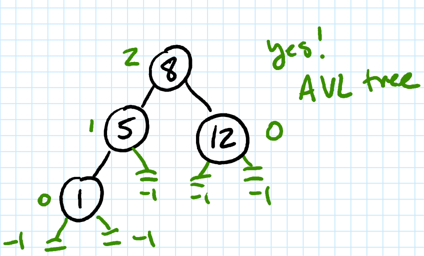
node now stores:

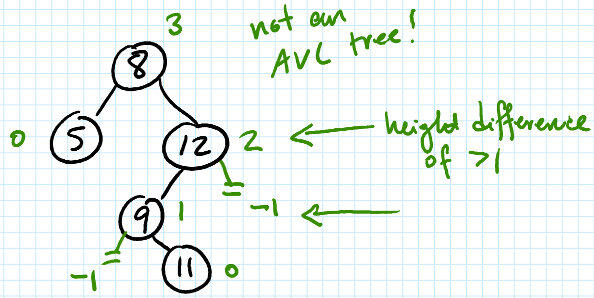
- value
- left

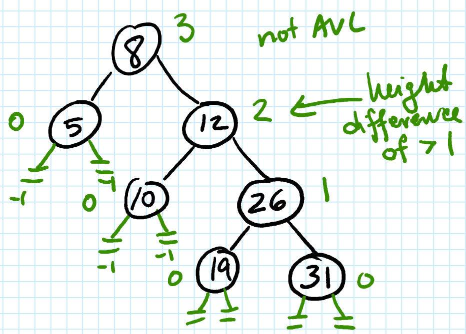
- right
- height

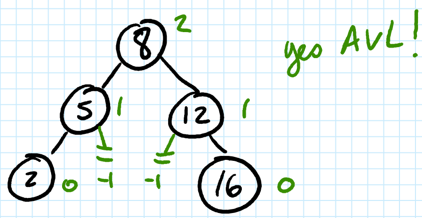
Reminder: The height is the depth of the deepest node.
Height of a single node is zero. Height of an empty tree is -1.

Are the following BSTs also AVL trees?

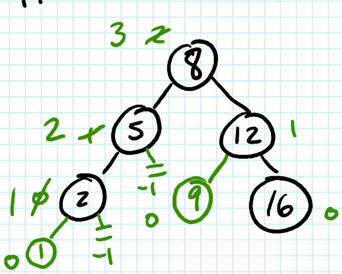








Suppose we start with an AVL tree:

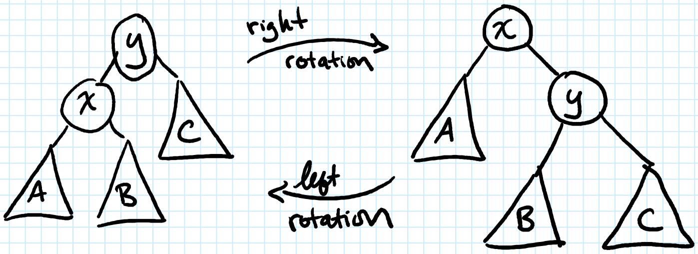


... and insert 9. Is it still an AVL tree? yes
Next we insert 1. Is it still an AVL tree? No, 5 node has imbalanced subtrees

Once we have an AVL tree, the AVL invariant can only be broken by the insert and remove operations. So we need a way to "rebalance" the AVL tree after an insert or remove.

REBALANCING TREES WITH ROTATIONS

Imagine putting your hands on a steering wheel at x and y, then turn:



See an animation of left/right rotation:
https://en.wikipedia.org/wiki/File:Binary_Tree_Rotation_(animated).gif

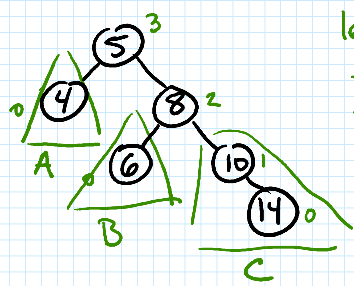
We are rebalancing the height while maintaining the BST property: $A < x < B < y < C$

As long as the rebalancing takes only $O(\log_2(n))$ work, all operations on the AVL tree will be $O(\log_2(n))$, which was our goal.

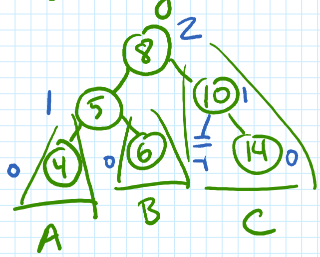
practice: rebalance the following BSTs to be AVLs:

rotation plan:

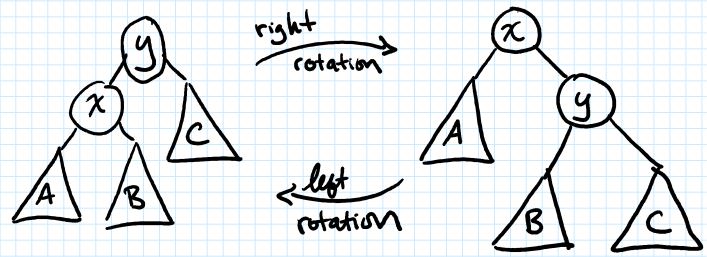
example 1



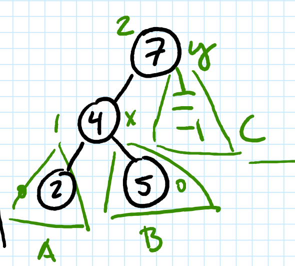
left rotation(5)
node 5 is x
node 8 is y



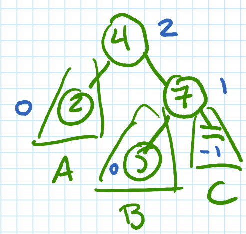
balanced AVL tree!



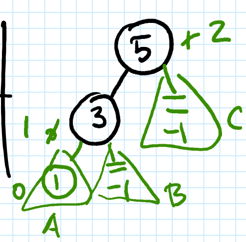
example 2



right rotation(7)



example 3



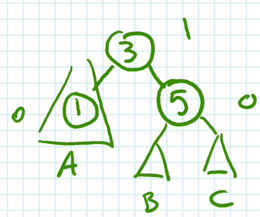
Is this an AVL tree? yes

insert (1)

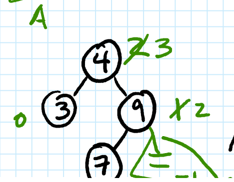
Now is it an AVL tree? no

if not, rebalance

right rotate(5)



example 4



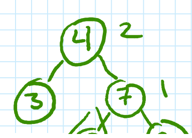
Is this an AVL tree? yes

insert (6)

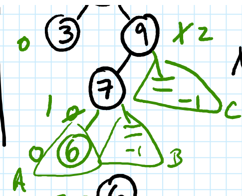
Now is it an AVL tree? no

if not, rebalance

right rotate(9)

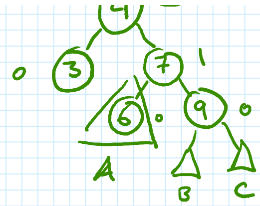


example 4

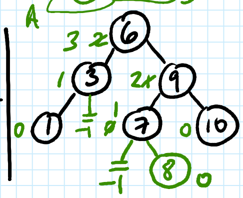


insert (6)
Now is it an AVL tree? no
if not, rebalance

right rotate (4)

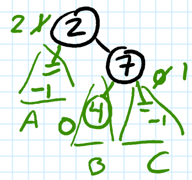


example 5



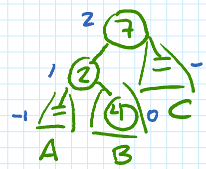
Is this an AVL tree? yes
insert (8)
Now is it an AVL tree? yes
if not, rebalance

example 6



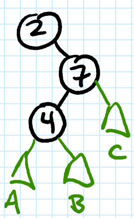
Is this an AVL tree? yes
insert (4)
Now is it an AVL tree? no
if not rebalance

leftrotate (2)



Some configurations require TWO rotations to rebalance.

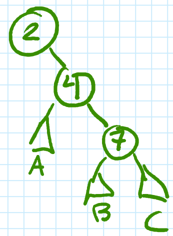
- first get a linear structure
- then do a rotation to fix the height imbalance

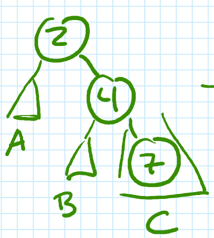


rightrotate(7) ⟶ gives

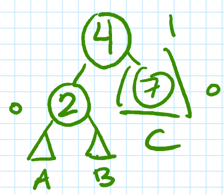
then

left rotate(2)





gives ⟶

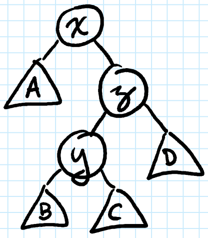


} this is an AVL tree! (balanced)

OVERALL REBALANCING SCHEME :

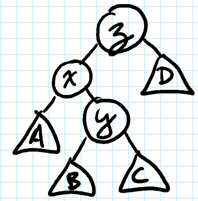
BST property: $(A < x < B < y < C < z < D)$

If the right subtree is too high:

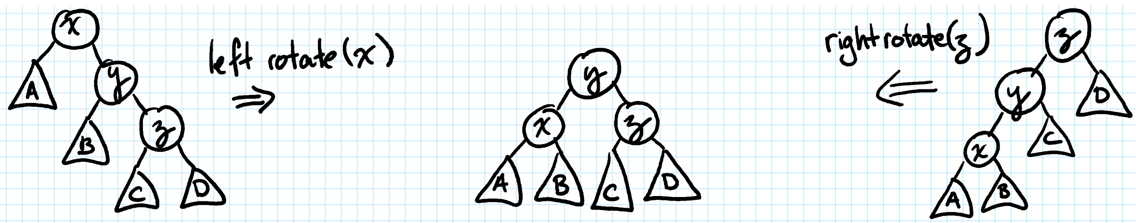


⇓ rightrotate(z)

If the left subtree is too high:



⇓ leftrotate(x)



BALANCED AT LAST!

pseudocode for rebalancing

Note: if subtree heights differ by at most 1, does nothing (no rebalancing necessary).

```
void rebalance(node)
    delta = (height of right subtree) − (height of left subtree)
    if delta > 1   // right subtree is too high
        if node's right's left height > node's right's right height
            set node's right to rightRotate(node's right)
        node = leftRotate(node)
    else if delta < -1   // left subtree is too high
        if node's left's left height < node's left's right height
            set node's left to leftRotate(node's left)
        node = rightRotate(node)
```

How much work does this require? left and right rotate take constant work

Overall, rebalance also takes constant work.

pseudocode for recalculating height of a node

```
void recalculateHeight(node)
    if node's left is null
        left_H = -1
    else
        left_H = node's left's height
    if node's right is null
        right_H = -1
    else
        right_H = node's right's height
    node's height = max(left_H, right_H) + 1
```

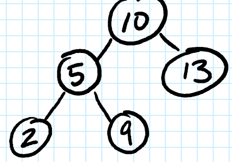
How much work does this require? constant

pseudocode for insert UPDATED FOR AVL trees

```
void insert(K key, V value) // public method
    // insert changes the structure of the tree, so we update root
    root = insertInSubtree(root, key, value)
    increment size

node * insertInSubtree(current, key, value) // private helper
    // base case
    if current is null
        create a new node to store key and value, with height 0
        return a pointer to that node
    if key is equal to current key
        throw error "no duplicate keys allowed!"
    // recursive cases
    if key < current's key
        set current's left to insertInSubtree(current's left, key, value)
        recalculateHeight(current)
        rebalance(current)
        return current
    if key > current's key
        set current's right to insertInSubtree(current's right, key, value)
        recalculateHeight(current)
        rebalance(current)
        return current
```

example



Start with an AVL tree
insert(7)

AVL tree efficiency

- rebalance and recalculate Height are $O(1)$ operations
- We may have to rebalance and recalculate height on every node along the path from the root
- There are $O(h)$ many nodes along that path
- $O(h) \times O(1) = O(h)$ total work to repair the tree after an insert or remove
- AVL invariant maintains that height is $O(\log_2(n))$
  ↳ thus insert and remove are $O(\log_2(n))$ operations!

We now have an efficient Dictionary implementation. Woohoo!