Adam Blank | Lecture 8 | Autumn 2015

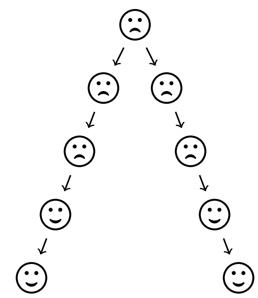
# CSE 332

## Data Abstractions

---

*CSE 332: Data Abstractions*

# AVL Trees



---

## Outline



---

## AVL Balance Condition!



Left and right subtrees **recursively** have heights differing by at most one.

**Definition (balance)**

$$\text{balance}(n) = \text{abs}(\text{height}(\texttt{n.left}) - \text{height}(\texttt{n.right}))$$

**Definition (AVL Balance Property)**

An AVL tree is balanced when:

For every node $n$, $\text{balance}(n) \leq 1$

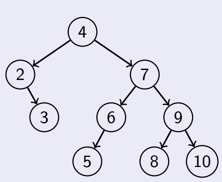
- This ensures a small depth
- It's relatively easy to maintain

---

## AVL Trees



**AVL Tree**




**Structure Property**:
0, 1, or 2 children

**BST Property**:
Keys in Left Subtree are smaller
Keys in Right Subtree are larger

**AVL Balance Property**:
Left and Right subtrees have heights that differ by at most one.

That is, **all AVL Trees are BSTs**, but the reverse is not true.

AVL Trees rule out **unbalanced BSTs**.

---

## Tree Representation in Code



**Node Class?**

```
class Node {
    Data data;
    Node left;
    Node right;
}
```

**This Definition Leads to Redundant Code**

```
boolean find(Node current, int data) {
    if (current == null) {
        return false;
    }
    else if (current.data == data) {
        return true;
    }

    if (current.data < data) {
        return find(current.left, data);
    }
    else {
        return find(current.right, data);
    }
}
```

But that's what we've been writing! Why is it ugly?

- It's redundant
- The left and right cases are **the same**, why write them twice?
- It's not ideomatic (e.g., the right abstraction would allow us to write the two cases found vs. not found)

## Tree Representation in Code


### A Bad Fix

```java
boolean find(Node current, int data) {
   if (current == null) {
      return false;
   }

   if (current.data == data) {
      return true;
   }
   else {
      Node next = null;
      if (current.data < data) { next = current.left; }
      else                     { next = curent.right; }
      return find(next, data);
   }
}
```

### Node Class?

```java
class Node {
   Data data;
   Node left;
   Node right;
}
```

### How is This Code?

```java
int a0 = 0;
int a1 = 0;
int a2 = 0;

for (int i = 0; i < 3; i++) {
   if (i == 0) { a0 = i; }
   else if (i == 1) { a1 = i; }
   else { a2 = i; }
}
```

This course is about **making the right data abstractions**. This is a perfect example of where we could improve.

**Keep an array of children!**

## Another Try!


### Node Class?

```java
class Node {
   Data data;
   Node[] children;
}
```

### Is This Really Any Better?

```java
boolean find(Node current, int data) {
   if (current == null) {
      return false;
   }
   else if (current.data == data) {
      return true;
   }

   int next = current.data < data ? 0 : 1;
   return current.children[next];
}
```

Actually, yes! How do I get "the other child" in each of these versions?

```java
Node getOtherChild(Node me, Node child1) {
   if (me.left == child1) { return me.right; }
   else                   { return me.left; }
}
```

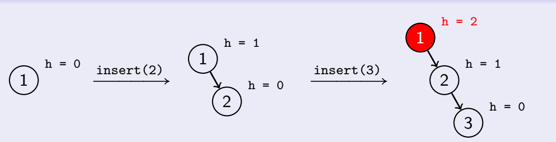
vs.

```java
Node getOtherChild(Node me, int child1) {
   return me.children[1 - child1];
}
```

Since operations on binary trees are **almost always symmetric**, this is a big deal for complicated operations. Keep this in mind.

## The BST Worst Case

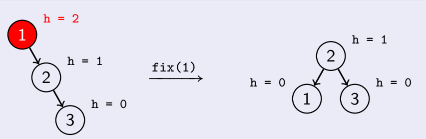

### Worst Case




When we insert 3, we violate the AVL Balance condition. What to do?

There's only one tree with the BST Property and the Balance Property:

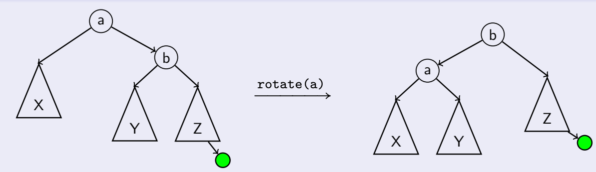
### FIXING The Worst Case




## AVL Rotation


This "fix" is called a rotation. We're "rotating" the child node "up":

### Rotation

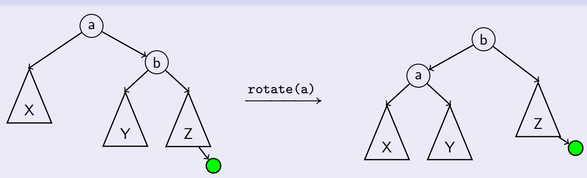



**This is the only fundamental of AVL Trees!**

You can either look at this as "the only way to correctly rearrange the subtrees" or it's helpful to think of it as gravity.

## AVL Rotation


### Rotation



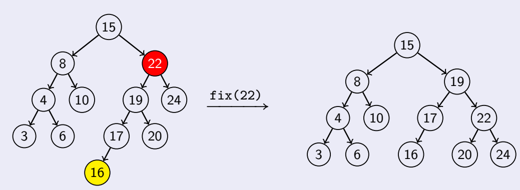

### The Code

```java
void rotate(Node current) {
   Node child = current.right;
   current.right = child.left;
   child.left = current;

   child.height = child.updateHeight();
   current.height = current.updateHeight();

   current = child;
}
```

## More Complicated Now...


### Inserting 16

Is the result an AVL tree? If not, how do we fix it?

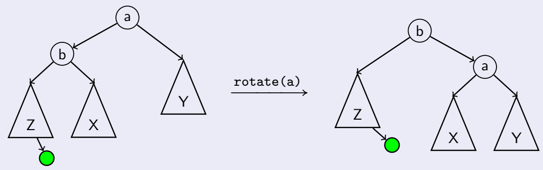



This is just the same rotation in the other direction!

## AVL Rotation: The Other Way



### Rotation



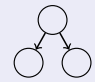

### The Code

```
void rotate(Node current) {
   Node child = current.left;
   current.left = child.right;
   child.right = current;

   child.height = child.updateHeight();
   current.height = current.updateHeight();

   current = child;
}
```

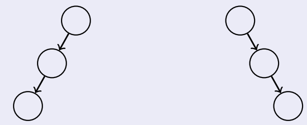
## AVL Rotations. . . Are We Done?



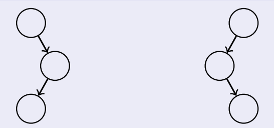
### We Want. . .



### Cases We've Handled



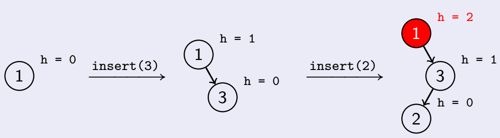
### Cases To Handle



## Another Case



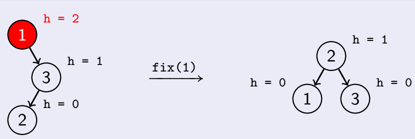
### Second Case




When we insert 2, we violate the AVL Balance condition. What to do?

There's only one tree with the BST Property and the Balance Property:

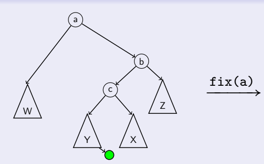
### FIXING The Second Case

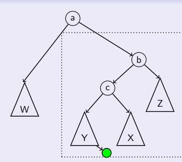



## It Doesn't Look Like a Single Rotation Will Do. . .

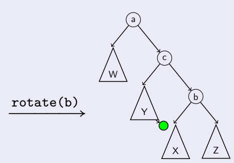


### **Double** Rotation

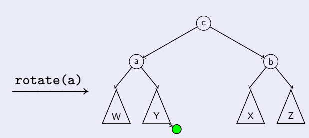



First, we rotate b.




Now, we're back to the **line** case.

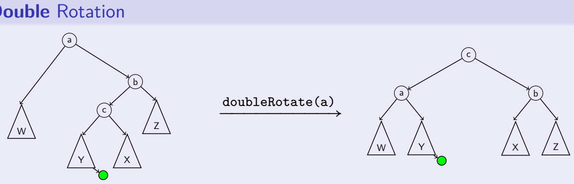



And now it's balanced!

## And The Code. . .



### **Double** Rotation




### Double Rotation Code

```
void doubleRotation(Node current) {
   rotation(current.right, RIGHT);
   rotation(current, LEFT);
}
```

## Putting Together the AVL Operations



### AVL Operations

- `find(x)` is identical to BST find
- `insert(x)` by (1) doing a BST insert, and (2) fixing the tree with either a rotation or a double rotation
- `delete(x)` by either a similar method to `insert`–or doing lazy delete

### AVL Fields

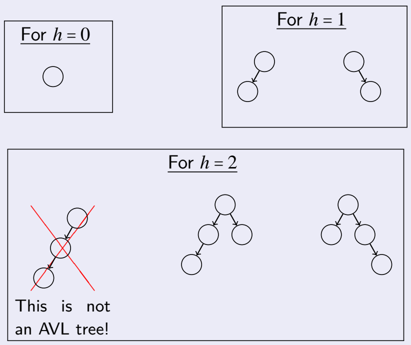
- We've seen that the code is very redundant if we use `left` and `right` fields; so, we should use a `children` array
- We've seen quick access to `height` is very important; so, it should be a field

Okay, so does it work?

## Does an AVL Tree Work? 16

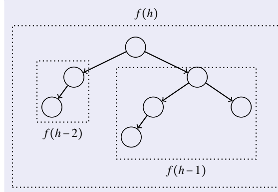
We must **guarantee** that the AVL property gives us a small enough tree.
Our approach: Find a big **lower bound** on the number of nodes necessary to make a tree with height $h$.

**What is the smallest number of nodes to get a height $h$ AVL Tree?**




## Does an AVL Tree Work? 17

**What is the smallest number of nodes to get a height $h$ AVL Tree?**




The general number of nodes to get a height of $h$ is:

$$f(h) = f(h-2) + f(h-1) + 1$$

We break down where each term comes from. We want a tree that has the **smallest** number of nodes where each branch has the AVL Balance condition.

- $f(h-1)$: To force the height to be $h$, we take the smallest tree of height $h-1$ as one of the children
- $f(h-2)$: We are allowed to have the branches differ by one; so, we can get a smaller number of nodes by using $f(h-2)$
- $+1$ comes from the root node to join together the two branches

## Does an AVL Tree Work? 18

So, now we solve our recurrence. How?

**Ratio Between Terms**

A good way of solving a recurrence that we expect to be of the form $X^n$ is to look at the ratio between terms. If $\dfrac{f(h+1)}{f(h)} > X$, then

$$f(h+1) > Xf(h) > X(X(f(h-1)) > \cdots > X^n$$

So, we evaluate these ratios and see the following:

OUTPUT

```
>> 2.0
>> 2.0
>> 1.75
>> 1.7142857142857142
>> 1.6666666666666667
>> 1.65
>> 1.6363636363636365
>> 1.6296296296296295
>> 1.625
>> 1.6223776223776223
>> 1.6206896551724137
>> 1.6196808510638299
>> 1.619047619047619
>> 1.618661257606491
>> 1.618421052631579
>> ...
```

## Proving the Closed Form 19

In this case, we see that $f(h)$ pretty quickly converges to $\phi(1.618\ldots)$.
Before trying to prove this closed form, we should look at a few examples:

- $f(0) = 1$ vs. $(\phi)^0 = 1$
- $f(1) = 2$ vs. $(\phi)^1 = \phi$

We want to show that $f(h) >$ some closed form, but looking at the first base case, $1 \not> 1$. So, we'll prove $f(h) > \phi^h - 1$ instead.

**Induction Proof**

- Base Cases: Note that $f(0) = 1 > 1 - 1 = 0$ and $f(1) = 2 > \phi - 1 \approx 0.618$
- Induction Hypothesis: Suppose that $f(h) > \phi^h - 1$ for all $0 \le h \le k$ for some $k \ge 1$.
- Induction Step:

$$\begin{aligned} f(n+1) &\ge f(n) + f(n-1) + 1 \\ &> (\phi^n - 1) + (\phi^{n-1} - 1) + 1 && \text{[By IH]} \\ &= \phi^{n-1}(\phi + 1) + 1 - 2 \\ &= \phi^{n+1} - 1 && \text{[By } \phi\text{]} \end{aligned}$$

In the step labeled "by $\phi$", we use the property $\phi^2 = \phi + 1$.

## So, efficiency? 20

So, since $n \ge f(h) > \phi^h - 1$, taking lg of both sides gives us:

$$\lg(n) > \lg(\phi^h - 1) \approx \lg(\phi^h) = h\lg(\phi)$$

So, $h \in \mathcal{O}(\lg n)$.

- Worst-case complexity of `find`: $\mathcal{O}(\lg n)$
- Worst-case complexity of `insert`: $\mathcal{O}(\lg n)$
  - Tree starts balanced
  - A rotation is $\mathcal{O}(1)$ and there's an $\mathcal{O}(\lg n)$ path to root
  - (Same complexity even without one-rotation-is-enough fact)
  - Tree ends balanced
- Worst-case complexity of `buildTree`: $\mathcal{O}(n \lg n)$
- Worst-case complexity of `delete`: (requires more rotations) $\mathcal{O}(\lg n)$
- Worst-case complexity of `lazyDelete`: $\mathcal{O}(1)$

## Pros and Cons of AVL Trees 21

**Pros of AVL trees**

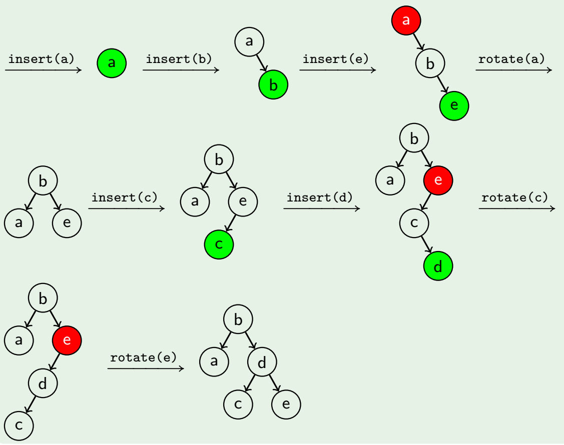
- All operations logarithmic worst-case because trees are always balanced
- Height balancing adds no more than a constant factor to the speed of insert and delete

**Cons of AVL trees**

- Difficult to program & debug
- More space for height field
- Asymptotically faster but rebalancing takes a little time
- Most large searches are done in database-like systems on disk and use other structures (e.g., B-trees, our next data structure)

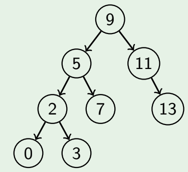
## Some Examples 

Example (Insert $a$,$b$,$e$,$c$,$d$ into an AVL Tree)

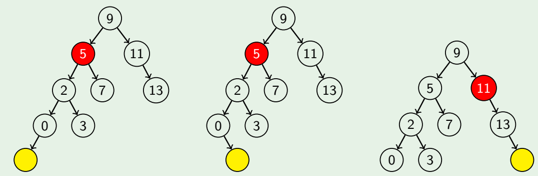



## Some Examples 

Example (Which Rotation?)

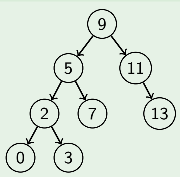


- Which insertions would cause a **single rotation**?

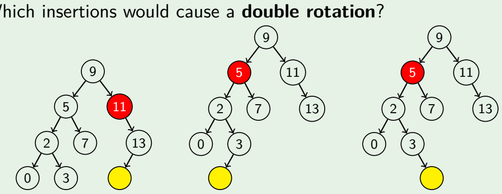


## Some Examples 

Example (Which Rotation?)

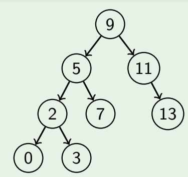


- Which insertions would cause a **double rotation**?

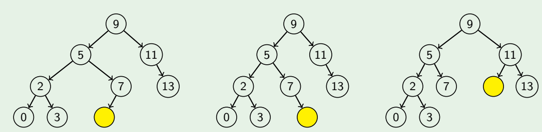


## Some Examples 

Example (Which Rotation?)

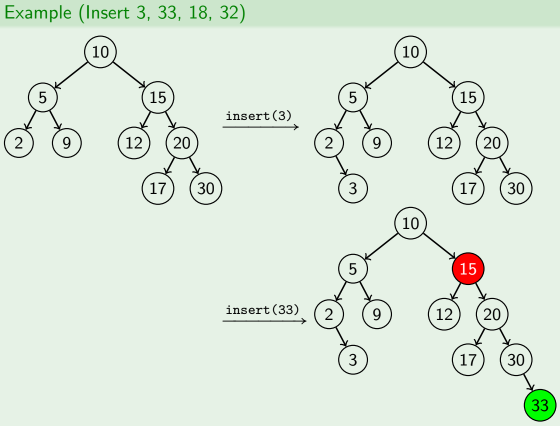


- Which insertions would cause no rotation?



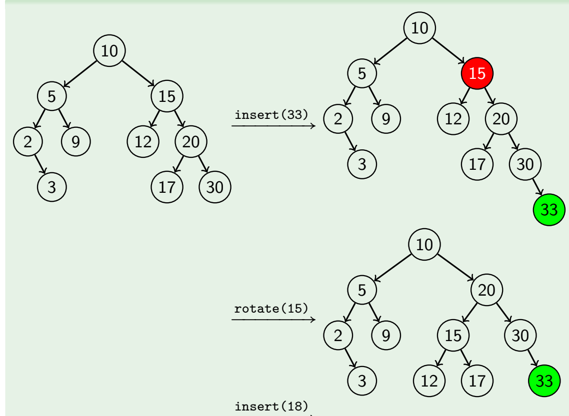
## Some Examples 

Example (Insert 3, 33, 18, 32)




## Some Examples 

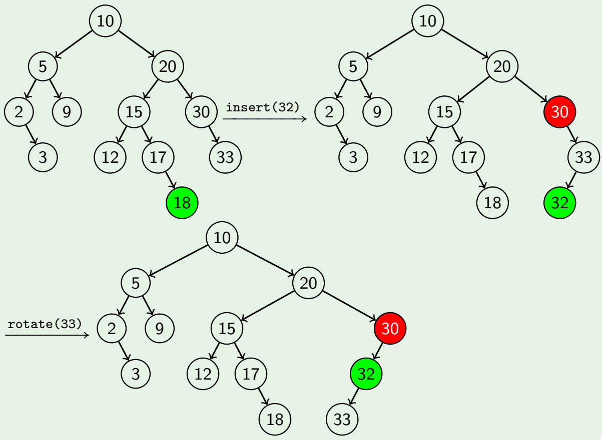
Example (Insert 3, 33, 18, 32)



## Some Examples



Example (Insert 3, 33, 18, 32)




## Some Examples



Example (Insert 3, 33, 18, 32)