AVL Trees
Outline

1 Introducing AVL Trees
2 Tree Representation in Code
3 How Does an AVL Tree Work?
4 Why Does an AVL Tree Work?
5 AVL Tree Examples
Left and right subtrees **recursively** have **heights** differing by at most one.

**Definition (balance)**

\[
\text{balance}(n) = \text{abs} (\text{height}(n.\text{left}) - \text{height}(n.\text{right}))
\]

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

AVL Trees rule out unbalanced BSTs.
This Definition Leads to Redundant Code

```java
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)
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.
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:
This “fix” is called a rotation. We’re “rotating” the child node “up”:

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.
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;
}
```
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

```c
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;
}
```
We Want...

Cases We’ve Handled

Cases To Handle
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**

```
1 void doubleRotation(Node current) {
2     rotation(current.right, RIGHT);
3     rotation(current, LEFT);
4 }
```
Food For Thought

Expectations and Tips

- For any one bug, debug for **30 minutes**, then stop.
- For any one exercise, you should be spending ≈ 30 minutes.
- Exercises are (almost always) a **direct application** of lecture. They are not 311 problems.
- Partners: There is no “formal” way of saying “my partner isn’t doing enough work”, but we DO factor that information in if you let us know.
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?
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?

For $h = 0$

For $h = 1$

This is not an AVL tree!
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
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 \( \frac{f(h+1)}{f(h)} > X \), then

\[
f(h+1) > X f(h) > X(X(f(h-1))) > \cdots > X^n
\]

So, we evaluate these ratios and see the following:

```
>> 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
>> 1.61861257606491
>> 1.618421052631579
>> ...
```
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 $f(h) > \text{some closed form}$, 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 $0 \leq h \leq k$ for some $k \geq 1$.
- **Induction Step:**
  \[
  f(k + 1) \geq f(k) + f(k - 1) + 1 \\
  > (\phi^k - 1) + (\phi^{k-1} - 1) + 1 \quad \text{[By IH]} \\
  = \phi^{k-1}(\phi + 1) + 1 - 2 \\
  = \phi^{k+1} - 1 \quad \text{[By $\phi$]}
  \]

In the step labeled “by $\phi$”, we use the property $\phi^2 = \phi + 1$. 
So, efficiency?

So, since \( n \geq f(h) > \phi^h - 1 \), 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:
- Worst-case complexity of insert:
  - 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:
- Worst-case complexity of delete: (requires more rotations)
- Worst-case complexity of lazyDelete:
So, efficiency?

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

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

So, \( h \in O(\lg n) \).

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

- \textbf{insert}(a) \rightarrow a
- \textbf{insert}(b) \rightarrow a \rightarrow b
- \textbf{insert}(e) \rightarrow b \rightarrow a \rightarrow e
- \textbf{rotate}(a) \rightarrow a \rightarrow b \rightarrow e
- \textbf{insert}(c) \rightarrow b \rightarrow a \rightarrow e \rightarrow c
- \textbf{insert}(d) \rightarrow b \rightarrow a \rightarrow e \rightarrow c \rightarrow d
- \textbf{rotate}(c) \rightarrow b \rightarrow a \rightarrow e \rightarrow d
- \textbf{rotate}(e) \rightarrow b \rightarrow a \rightarrow d \rightarrow e
- \textbf{rotate}(e) \rightarrow a \rightarrow d \rightarrow e

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)

```
insert(3) →

insert(33) →
```
Example (Insert 3, 33, 18, 32)

1. Insert 33:
   - Insertion point: 15
   - New tree:
     - 10
     - 5
       - 2
       - 9
       - 12
       - 20
     - 15
       - 3
       - 17
       - 30
     - 33

2. Rotate 15:
   - Rotation point: 10
   - New tree:
     - 10
       - 5
         - 2
         - 9
       - 15
         - 12
         - 20
     - 30
     - 33

3. Insert 18:
   - Insertion point: 15
   - New tree:
     - 10
       - 5
         - 2
         - 9
       - 15
         - 12
         - 17
         - 30
     - 33
     - 18

Diagram:
- Tree structure showing the operations of inserting 33, rotating 15, and inserting 18.
Some Examples

Example (Insert 3, 33, 18, 32)

1. Insert 32:

```
10
  5
  2
   3
   9
  15
  20
  30
  18

insert(32) →
```

2. Rotate 33:

```
10
  5
  2
   3
   9
  15
  20
  32
  33

rotate(33) →
```
Some Examples

Example (Insert 3, 33, 18, 32)

1. Initial tree:
   - 10
     - 5
       - 2
         - 3
     - 15
       - 9
     - 20
       - 12
         - 3
     - 30
       - 17
         - 33

2. Insert 32:
   - 10
     - 5
       - 2
         - 3
     - 15
       - 9
     - 20
       - 12
         - 33
       - 17
         - 32
     - 30
       - 18

3. Rotate 30:
   - 10
     - 5
       - 2
         - 3
     - 15
       - 9
     - 20
       - 12
         - 30
         - 17
         - 33
       - 18