CSE 332
Data Structures and Parallelism

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

AVL Balance Condition!

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

AVL Trees

AVL Trees rule out unbalanced BSTs.

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

Tree Representation in Code

This Definition Leads to Redundant Code

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

Is This Really Any Better?

This course is about making the right data abstractions. This is a perfect example of where we could improve. Keep an array of children!

The BST Worst Case

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

Fixing the Worst Case

This is just the same rotation in the other direction!

AVL Rotation

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.

More Complicated Now...

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

X
Y
Z
rotate(a)

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;
}
```

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

h = 2
fix(1)

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

doubleRotate(a)

Left Double Rotation Code

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

Food For Thought

Expectations and Tips

- For any one bug, debug for 30 minutes, then stop.
- For any one exercise, you should be spending drastically different times.
- Most exercises are direct applications 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?

Does an AVL Tree Work?

What is the smallest number of nodes to get a height \( h \) AVL Tree?

\[
egin{align*}
\text{f}(h-2) & \quad \text{f}(h-1) \quad \text{f}(h) \\
\text{f}(h-2) & \quad \text{f}(h-1) \quad \text{f}(h)
\end{align*}
\]

The general number of nodes to get a height of \( h \) is:

\[
\text{f}(h) = \text{f}(h-2) + \text{f}(h-1) + 1
\]

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

Proving the Closed Form

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 \( f(h) \) grows as expected, but looking at the first base case, \( f(1) \) is 1. So, we’ll prove \( f(h) > \phi^h - 1 \) instead.

Induction Proof

- Base Cases: Note that \( f(0) = 1 > 1-0 = 0 \) and \( f(1) = 2 > \phi - 1 = 0.618 \)
- Induction Hypothesis: Suppose that \( f(h) > \phi^h - 1 \) for all \( 0 \leq h \leq k \)
- Induction Step:

\[
\begin{align*}
    f(k+1) & \geq f(k) + f(k-1) + 1 \\
    & > (\phi^k - 1) + (\phi^{k-1} - 1) + 1 \quad \text{[By IH]} \\
    & = \phi^{k-1}(\phi + 1) + 1 - 2 \\
    & = \phi^{k+1} - 1 \quad \text{[By } \phi]\end{align*}
\]

In the step labeled “by \( \phi \),” we use the property \( \phi^2 = \phi + 1 \).
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)

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

\[
\begin{align*}
\text{insert(a)} & \rightarrow a \\
\text{insert(b)} & \rightarrow a \rightarrow b \\
\text{insert(e)} & \rightarrow a \rightarrow b \rightarrow e \\
\text{rotate(a)} & \rightarrow b \rightarrow a \rightarrow e \\
\text{insert(c)} & \rightarrow b \rightarrow a \rightarrow c \rightarrow e \\
\text{insert(d)} & \rightarrow b \rightarrow a \rightarrow c \rightarrow d \rightarrow e \\
\text{rotate(c)} & \rightarrow b \rightarrow a \rightarrow d \rightarrow c \rightarrow e \\
\text{rotate(e)} & \rightarrow b \rightarrow a \rightarrow d \rightarrow c \rightarrow e \\
\end{align*}
\]

Example (Which Rotation?)

- Which insertions would cause a single rotation?
- Which insertions would cause a double 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)