CSE 326 Data Structures

CSE 326 : Dave Bacon

Priority Queues : AVL Trees
Logistics

- Project 2a due tonight...
- Homework 3 due Moday

- Midterm will be Feb 2 in class

- Reading: Finishing Chapter 4
Balanced BST

Observation
- BST: the shallower the better!
- For a BST with $n$ nodes
  - Average height is $O(\log n)$
  - Worst case height is $O(n)$
- Simple cases such as insert(1, 2, 3, ..., $n$) lead to the worst case scenario

Solution: Require a Balance Condition that
1. ensures depth is $O(\log n)$ — strong enough!
2. is easy to maintain — not too strong!
Potential Balance Conditions

1. Left and right subtrees of the root have equal number of nodes

2. Left and right subtrees of the root have equal height
Potential Balance Conditions

3. Left and right subtrees of every node have equal number of nodes

4. Left and right subtrees of every node have equal height
The AVL Balance Condition

Left and right subtrees of every node have equal heights differing by at most 1

Define: \( \text{balance}(x) = \text{height}(x.\text{left}) - \text{height}(x.\text{right}) \)

AVL property: \(-1 \leq \text{balance}(x) \leq 1\), for every node \( x \)

- Ensures small depth
  - Will prove this by showing that an AVL tree of height \( h \) must have a lot of (i.e. \( O(2^h) \)) nodes
- Easy to maintain
  - Using single and double rotations
The AVL Tree Data Structure

Structural properties
1. Binary tree property
2. Balance property: balance of every node is between -1 and 1

Result:
Worst case depth is $O(\log n)$

Ordering property
- Same as for BST
Proving Shallowness Bound

Let $S(h)$ be the min # of nodes in an AVL tree of height $h$

Claim: $S(h) = S(h-1) + S(h-2) + 1$

Solution of recurrence: $S(h) = O(2^h)$ (like Fibonacci numbers)

AVL tree of height $h=4$ with the min # of nodes
Testing the Balance Property

We need to be able to:

1.

2.

3.

NULLs have height $-1$
An AVL Tree
AVL trees: find, insert

• **AVL find:**
  - same as BST find.

• **AVL insert:**
  - same as BST insert, except may need to “fix” the AVL tree after inserting new value.
AVL tree insert

Let \( x \) be the node where an imbalance occurs.

Four cases to consider. The insertion is in the

1. left subtree of the left child of \( x \).
2. right subtree of the left child of \( x \).
3. left subtree of the right child of \( x \).
4. right subtree of the right child of \( x \).

Idea: Cases 1 & 4 are solved by a single rotation.
Cases 2 & 3 are solved by a double rotation.
Bad Case #1

Insert(6)
Insert(3)
Insert(1)
Fix: Apply Single Rotation

AVL Property violated at this node (x)

Single Rotation:
1. Rotate between x and child
Single rotation in general

\[ X < b < Y < a < Z \]

Height of tree before? Height of tree after? Effect on Ancestors?
Single rotation example
Bad Case #2

Insert(1)
Insert(6)
Insert(3)
Fix: Apply Double Rotation

AVL Property violated at this node (x)

Double Rotation
1. Rotate between x’s child and grandchild
2. Rotate between x and x’s new child
Double rotation in general

$h \geq 0$

$W < b < X < c < Y < a < Z$

Height of tree before? Height of tree after? Effect on Ancestors?
Double rotation, step 1
Double rotation, step 2
Imbalance at node X

Single Rotation
1. Rotate between x and child

Double Rotation
1. Rotate between x’s child and grandchild
2. Rotate between x and x’s new child
Single and Double Rotations:

Inserting what integer values would cause the tree to need a:

1. single rotation?

2. double rotation?

3. no rotation?
Insertion into AVL tree

1. Find spot for new key
2. Hang new node there with this key
3. Search back up the path for imbalance
4. If there is an imbalance:
   - case #1: Perform single rotation and exit
   - case #2: Perform double rotation and exit

Both rotations keep the subtree height unchanged. Hence only one (single or double) rotation is sufficient!
Easy Insert

Insert(3)

Unbalanced?
Hard Insert (Bad Case #1)

Insert(33)

Unbalanced?
How to fix?
Single Rotation
Hard Insert (Bad Case #2)

Insert(18)

Unbalanced?

How to fix?
Single Rotation (oops!)
Double Rotation (Step #2)
Insert into an AVL tree: a b e c d
AVL Trees Revisited

- **Balance condition:**
  - For every node \( x \), \(-1 \leq \text{balance}(x) \leq 1\)
  - Strong enough: Worst case depth is \( O(\log n) \)
  - Easy to maintain: *one* single or double rotation

- Guaranteed \( O(\log n) \) running time for
  - Find ?
  - Insert ?
  - Delete ?
  - buildTree ?
AVL Trees Revisited

• What extra info did we maintain in each node?

• Where were rotations performed?

• How did we locate this node?
Other Possibilities?

- Could use different balance conditions, different ways to maintain balance, different guarantees on running time, ...

- Why aren’t AVL trees perfect?

- Many other balanced BST data structures
  - Red-Black trees
  - AA trees
  - Splay Trees
  - 2-3 Trees
  - B-Trees
  - ...
Splay Trees

- Blind adjusting version of AVL trees
  - Why worry about balances? Just rotate anyway!
- *Amortized* time per operations is $O(\log n)$
- Worst case time per operation is $O(n)$
  - But guaranteed to happen rarely

**Insert/Find always rotate node to the root!**

*SAT/GRE Analogy question:*
AVL is to Splay trees as ___________ is to ___________
Recall: Amortized Complexity

If a sequence of $M$ operations takes $O(M f(n))$ time, we say the amortized runtime is $O(f(n))$.

- Worst case time per operation can still be large, say $O(n)$
- Worst case time for any sequence of $M$ operations is $O(M f(n))$

Average time per operation for any sequence is $O(f(n))$

Amortized complexity is worst-case guarantee over sequences of operations.
Recall: Amortized Complexity

• Is amortized guarantee any weaker than worstcase?

• Is amortized guarantee any stronger than average case?

• Is average case guarantee good enough in practice?

• Is amortized guarantee good enough in practice?
The Splay Tree Idea

If you’re forced to make a really deep access:

Since you’re down there anyway, fix up a lot of deep nodes!