Recall from Last Time: AVL Trees

✦ AVL trees are **height-balanced** binary search trees

✦ **Balance factor** of a node = 
  height(left subtree) - height(right subtree)

✦ An AVL tree can only have balance factors **+1, 0, or −1**
  at **every** node
  ➔ Height of an **empty subtree** is defined to be **−1**

✦ **Implementation:** Store current heights in each node and calculate balance factors when needed from subtrees’ root nodes.
Is this tree AVL?

Heights

1 4
1 0

7 1
8 0

No! Ain’t ALV.
Why AVL?

✦ Can prove: Height of an AVL tree of N nodes is always $O(\log N)$ (see previous lecture and textbook)

✦ Run time for accessing any node is therefore $O(\log N)$

✦ One problem: Insert/Remove may upset AVL balance

Insert Example

No longer an AVL tree

Problem: Insert may cause balance factor to become 2 or −2 for some node on the path from insertion point to root node
Restoring Balance

**Idea:** After Inserting the new node,
1. Back up to root updating heights along the access path
2. If Balance Factor = 2 or –2, adjust tree by rotation around deepest such node.

Rotating to restore Balance: A Simple Example

AVL | Not AVL | AVL
--- | --- | ---
4 5 1 | 4 5 0 | 4 5 0
0 6 0 | 0 6 0 | 0 6 0
1 7 -1 | 1 7 -1 | 1 7 -1
0 9 0 | 0 9 0 | 0 9 0

Insert 3 | Rotate

R. Rao, CSE 326
Various Cases of Insertion

Tree before insertion
(BF = Balance Factor)

“Outside” Case

Tree after insertion
Insertions in AVL Trees

Let the node that needs rebalancing be $\alpha$.

There are 4 cases:

Outside Cases (require single rotation):
1. Insertion into left subtree of left child of $\alpha$.
2. Insertion into right subtree of right child of $\alpha$.

Inside Cases (require double rotation):
3. Insertion into right subtree of left child of $\alpha$.
4. Insertion into left subtree of right child of $\alpha$.

Rebalancing is performed through four separate rotation algorithms.
Consider a valid AVL subtree

Inserting into X destroys the AVL property
Insertions in AVL Trees: Outside Case

Do a “right rotation”:
1. Make $j$ point to $Y$
Insertions in AVL Trees: Outside Case

Do a “right rotation”:
2. Make $k$ point to $j$

AVL property has been restored!
(“Left rotation” is mirror symmetric)

Insertions in AVL Trees: Inside Case

Consider a valid AVL subtree

R. Rao, CSE 326
Insertions in AVL Trees: Inside Case

Inserting into $Y$ destroys the AVL property

$k$  $j$  $Z$

$X$  $Y$  

Does “right rotation” restore balance?

---

Insertions in AVL Trees: Inside Case

“Right rotation” don’t do nothin’ fa dis tree…

---

R. Rao, CSE 326

R. Rao, CSE 326
Insertions: Inside Case Take 2

Consider the structure of subtree Y...

Insertions in AVL Trees: Inside Case

Y = node i and subtrees V and W
Insertions in AVL Trees: Inside Case

Let’s try a left-right “double rotation” . . .

Steps for Left-Right Double Rotation

1. **Left Rotation**: Adjust relevant pointers…
Insertions in AVL Trees: Inside Case

Steps for Left-Right Double Rotation

2. **Right Rotation:** Adjust relevant pointers
   3. Make \( i \) the root

Balance has been restored!

(Right-left case is mirror-symmetric)

---

AVL Tree On Board Exercise

* Insert 8, 1, 0, 2 in that order into following AVL tree:
Pros and Cons of AVL Trees

Arguments for AVL trees:
1. Search is $O(\log N)$ since AVL trees are always balanced.
2. The height balancing adds no more than a constant factor to the speed of insertion. (Why?)

Arguments against using AVL trees:
1. Difficult to program & debug; more space for height info.
2. Asymptotically faster but can be slow in practice.
3. Most large searches are done in database systems on disk and use other structures (e.g. B-trees).
4. May be OK to have $O(N)$ for a single operation if total run time for many consecutive operations is fast…

Did someone say spay??

(enter Splay Trees)
Splay Trees

Splay trees are tree structures that:
1. Are not perfectly balanced all the time
2. Allow actual Find operations to balance the tree so that future operations may run faster

Based on the heuristic:
If X is accessed once, it is likely to be accessed again.

- After node X is accessed, perform “splaying” operations to bring X up to the root of the tree.
- Do this in a way that leaves the tree more balanced as a whole.

Splaying: A Motivating Example

Initial tree

After splaying with R

After Find(R)
Splay Idea: Get R up to the root using rotations
Splay Tree Terminology

- Let X be a non-root node with $\geq 2$ ancestors.
- Let P be its parent node.
- Let G be its grandparent node.

Splay Tree Terminology

<table>
<thead>
<tr>
<th>element</th>
<th>left</th>
<th>right</th>
<th>parent</th>
</tr>
</thead>
</table>

Splay Tree Operations

1. Nodes must contain a parent pointer.

2. When X is accessed, apply one of six rotation operations:
   - Single Rotations (X has a P but no G)
     - zig-left, zig-right
   - Double Rotations (X has both a P and a G)
     - zig-zig-left, zig-zig-right
     - zig-zag-left, zig-zag-right
Splay Trees: Zig operation

✦ “Zig” is just a single rotation, as in an AVL tree
✦ Suppose R was the node that was accessed (e.g. using Find)

```
    Q
   / \
  R   C
 / \
A   B
```

Zig-right moves R to the top can access R faster next time

Splay Trees: Zig operation

✦ Suppose Q is accessed (e.g. using Find)

```
    Q
   / \
  R   C
 / \
A   B
```

Zig-left moves Q to the top
Splay Trees: Zig-Zig operation

✦ “Zig-Zig” consists of two single rotations of the same type (assume R is the node that was accessed):

✦ Again, due to “zig-zig” splaying, R has bubbled to the top!
✦ Note: Parent-Grandparent rotated first.

Splay Trees: Zig-Zag operation

✦ “Zig-Zag” consists of two rotations of the opposite type (assume R is the node that was accessed):

✦ “Zig-Zag” splaying also causes R to move to the top.
Splay Trees: Example

Restructuring a tree with splaying after accessing $T$ (a–c) and then $R$ (c–d).

Splay Trees: Do-It-Yourself Exercise

- Insert the keys 1, 2, …, 7 in that order into an empty splay tree.
- What happens when you access “7”?
Examples suggest that splaying causes tree to get balanced. The actual analysis is rather advanced and is in Chapter 11.

**Result of Analysis:** Any sequence of $M$ operations on a splay tree of size $N$ takes $O(M \log N)$ time.

So, the amortized running time for one operation is $O(\log N)$.

This guarantees that even if the depths of some nodes get very large, you cannot get a long sequence of $O(N)$ searches because each search operation causes a rebalance. Without splaying, total time could be $O(MN)$.

Next Class:
Beyond Binary Trees: B-trees

**To Do:**
Finish Chapter 4 and Start Chapter 6
Homework # 2

Have a great weekend!