AVL Trees

AVL trees are one way to achieve $\Theta(\log(n))$ tree height.

An AVL tree:

- is a binary search tree
- at every node $v$: subtree heights differ by at most 1.

Imbalance Case 1/2

Insert 35.

```
        44
       / \
      32   78
     /     / \
    17    50   88
   /     /    / \
  35   48    62
```

Imbalance! What to do? What to do?

"Single rotate".

```
        z
       / \
      y   w
     /     \
    T0    T2
   /     /  \
  T1   T1   w
```

- $z$ is the unbalanced node closest to new node $w$
- path from $z$ to $w$ begins with "right, right"
- the result is an AVL tree again
- $y$'s new height = $z$'s old height!

Rebalanced!

Imbalance Case 2/2

Insert 20.

```
        44
       / \
      32   78
     /     / \
    17    50   88
   /     /    /  \
  20   48    62
```

Imbalance! What to do? What to do?

"Double rotate".

```
        z
       / \
      y   w
     /     \
    T0    T2
   /     /  \
  T1   T1   T2
```

Rebalanced!
Double Rotation, Generally

\[ z \]
\[ T_0 \]
\[ T_3 \]
\[ y \]
\[ T_1 \]
\[ T_2 \]
\[ x \]
\[ T_0 \]
\[ T_1 \]
\[ T_2 \]
\[ T_3 \]

- \( z \) is the unbalanced node closest to new node \( w \) (\( w \) not shown, but below \( x \) or is \( x \))
- path from \( z \) to \( w \) begins with “right, left”
- the result is an AVL tree again
- \( x \)'s new height = \( z \)'s old height!

Imbalance Cases 3/2, 4/2

Actually two more cases: if path from \( z \) to \( w \) begins with:

- “left, left”: mirror image of “right, right”, single rotation
- “left, right”: mirror image of “right, left”, double rotation

Practice Exercise 1

Practice Exercise 2

Height Update

To determine when to rotate, we need to remember heights of all nodes:

```java
class Node<K> {
    public K key;
    public Node<K> left, right, parent;
    public int height;
    ...
}
```

Assume that \( v \)'s child nodes already have their \textit{height} fields set correctly, then \( v \)'s height field can be set correctly in \( \Theta(1) \) time:

\[
v\.height = 1 +
\max((v\.left == null ? 0 : v\.left\.height),
    (v\.right == null ? 0 : v\.right\.height))
\]

Height Update

Suppose tree height is \( h \).

Insert a node (before rotation): \( \Theta(h) \) nodes need to change \textit{height} (path from that node to root).

Rotation: only 4 nodes need to change \textit{height} (\( x, y, z \), and one parent).
Summary of AVL Insert

Later we will see why AVL tree height is in $\Theta(\log(n))$.

With that in mind, AVL tree insert:

1. find which node to become parent of new node $[\Theta(\log(n))$ time]
2. put new node there $[\Theta(1)$ time]
3. walk towards root to update heights and find first node of imbalance $[\Theta(\log(n))$ time]
4. if imbalance found, rotate $[\Theta(1)$ time]

Both $[\Theta(\log(n))$ time] come from AVL tree height.

Remove: Easy Case

Remove 32, or 48, or 62, or 88.
If the node has no children, just unlink from parent.
(Then update heights of ancestors, rebalance...)

Remove: Slightly Harder Case

Remove 17. (Note that 32 is a good replacement.)
If the node has at most one child, just link parent to that child.
(Then update heights of ancestors, rebalance...)
This generalizes the easy case.

Remove: Slightly Harder Case, Generally

Prune $w$. $T_0$ may be empty. $p$ and ancestors need height updates and rebalance.

There are two more mirror images.

Remove: Hard Case

Remove 5. Call the node $w$. Both children non-empty.

Go right once. Go left all the way, call it $y$. Replace $w$.key by $y$.key.
$y$'s parent adopts $y$'s right child $T$.

Remove: Hard Case, Degenerate

Remove 5. Call the node $w$. Both children non-empty.

Go right. Go left all the way—can't! That's already $y$. $y$'s parent is $w$.
Still, replace $w$.key by $y$.key. $w$ also adopts $y$'s right child $T$. 
Rebalance after Remove

After a parent adopts a new child, need to update heights of parent and ancestors, and rebalance. Note that one rotation may be insufficient.

Remove 92. This triggers two rotations.

Summary of AVL Remove

Next we will see why AVL tree height is $\Theta(\log(n))$.

1. Find which node has the key, call it w \([\Theta(\log(n))\text{ time}]\)
2. If at most one child, w.parent adopts that child \([\Theta(1)\text{ time}]\)
3. Else:
   1. Go right, then go left all the way, call it y \([\Theta(\log(n))\text{ time}]\)
   2. w.key := y.key \([\Theta(1)\text{ time}]\)
   3. y.parent adopts y.right \([\Theta(1)\text{ time}]\)
4. Walk towards root, update heights and rebalance \([\Theta(\log(n))\text{ time}]\)

AVL Tree Height

If there are n nodes, what is the maximum possible height?

$\Leftrightarrow$

If the height is $h$, what is the minimum possible number of nodes?

- $\text{minsize}(0) = 0$
- $\text{minsize}(1) = 1$
- $\text{minsize}(h+2) = 1 + \text{minsize}(h+1) + \text{minsize}(h)$

Can prove by induction:

$\text{minsize}(h) = \phi^{h+2} - \phi^{h+2} - 1$

Golden ratio: $\phi = \left(\sqrt{5} + 1\right)/2 = 1.618 \ldots$

$\text{minsize}(h) = \frac{\phi^{h+2} - (1 - \phi)^{h+2}}{\sqrt{5}} - 1$

Dictionary

Stores a mapping (function) from finitely many keys to values.

Example: Store student names and marks. Find the mark of a student.

- d.insert(k, v): d now maps k to v
- d.lookup(k): return what k is mapped to (throw an exception if none)
- d.remove(k): forget what k is mapped to (some versions also return what k was mapped to)

Binary search trees are good for dictionaries too. Time costs are still $\Theta(\log(n))$. 

Can further get: $h < 1.4405 \cdot \log_2(n) + \text{constant}$

$1/\log_2(\phi) = 1.4405 \ldots$
Binary Search Trees for Dictionaries

- Tree nodes: Add a field for the value.

```java
class Node<K,V> {
    public K key;
    public V value;
    public Node<K,V> left, right, parent;
    ...
}
```

- `d.insert(k, v)`: Insert new node into tree (and rebalance).
  Just remember to store the value in the new node!
- `d.remove(k)`: Find and remove node from tree (and rebalance).
- `d.lookup(k)`: Search for node in tree. Return the value in the node found.
  (Throw exception if no node has key `k`)

Ranking

Rank of key `k` in set `s`: how many keys are smaller than `k`.

Example: if `s` stores these keys 42, 55, 61, then:

- `s.rank(42)` returns 0
- `s.rank(55)` returns 1
- `s.rank(61)` returns 2
- `s.rank(k)` throws exception if `k` not found

Similarly for dictionaries.

How to extend AVL trees for this? All operations Θ(log(n)) time.

In the remaining time, we do some correctness proofs.

B36-style.

Sketch: Loop Version of Find

Precondition: True.
Postcondition: `b` ↔ tree at root has key `k`

```java
b = false;
v = root;
while (!b && v != null) {
    int r = k.compareTo(v.key);
    if (r == 0) { b = true; }
    else if (r < 0) { v = v.left; }
    else { v = v.right; }
}
```

Use this loop invariant:

- `v_0` is a node in tree at root
- `(h_0, v subtree at v_0 has k) ↔ tree at root has k`

Sketch: Recursive Version of Find

Precondition: True.
Postcondition: `findaux` returns whether subtree at `v` has key `k`

```java
boolean findaux(Node<K> v) {
    if (v == null) { return false; }
    else { int r = k.compareTo(v.key);
        if (r == 0) { return true; }
        else if (r < 0) { return findaux(v.left); }
        else { return findaux(v.right); }
    }
}
```

Use this predicate:

`P(n)`: if height of subtree at `v` is `n`, then `findaux` returns whether subtree at `v` has key `k`