Introduction to Balanced Search Trees

- Want to guarantee worst case time $O(\log n)$ for dictionary operations on ordered data, irrespective of what sequence the operations come in (Vanilla BST is embarrassing when data is in increasing order)

- Possible balanced or approximately balanced choices include
  - B-Trees: Great for large amounts of data. Internal nodes are not fully “occupied,” however.
  - Bottom Up Red-Black Tree: Somewhat mysterious update operations. Difficult to handle the several cases. Two pass algorithms
  - Top-Down Red-Black Trees: One pass algorithm, difficult to handle delete
  - BB-trees, AA-trees: Two pass algorithms.
  - AVL trees: Simplest next step after BST. One function `restructure(x)` takes care of all cases. Two pass algorithm.
AVL Tree Is Simple To Describe

- An AVL tree $T$ is a binary search tree (BST) such that for every internal node $v$ the heights of the children of $v$ differ at most by 1

- The height of an empty tree (possible only in improper binary trees) is -1. Heights often explicitly stored in each node

- $v$ is said to be balanced, even though it is only approximately balanced

- Main trick: Keep nodes balanced as dynamic insert() and remove() operations are called. Difficult to ensure if we insist on exact balance
AVL Tree Height is $O(\log n)$

- The balance restriction is good enough
- Let $n(h)$ be the minimum number of internal nodes of an AVL tree of height $h$
  - $n(1) = 1$ and $n(2) = 2$
  - For $n > 2$, an AVL tree of height $h$ contains the root node, and a child of height $h - 1$.
  - The other child cannot have arbitrary height. Due to the AVL restriction, the height can be only $h - 1$ or $h - 2$.
  - $n(h)$ measures minimum and $n(h - 2) < n(h - 1)$
  - Both children are roots of AVL trees
  - Therefore $n(h) = 1 + n(h - 1) + n(h - 2) > 2n(h - 2)$
- Solving $n(h) \geq 2^{\frac{h}{2} - 1}$ or $h < 2 \log n(h) - 2$
Insertion

• General idea: Insert a node at the leaf level which changes the height of at most \(O(\log n)\) nodes

• Nodes on the path from the newly created inserted key to the root may have increased heights
  
  – Let \(x\) be the node (with parent \(y\)) such that its grandparent \(z\) is the first unbalanced node
  
  – The \(O(1)\) local procedure \(\text{restructure}(x)\) involving \(x, y,\) and \(z\) fixes the height of \(z\)

• In fact, for insertion, this procedure will fix the height of every unbalanced node
restructure(x) Is Easy

Five simple steps to understand the rearrangement

1. Identify \( x, y \) and \( z \). Let \( abc \) be the order in which \( x, y \) and \( z \) show up in an inorder listing. (Thus \( a \) is one of \( x \) or \( y \) or \( z \)).

2. Identify the four subtrees of the children of \( x, y \) and \( z \). Let \( T0, T1, T2, T3 \) be the order in which these trees show up in an inorder listing.

3. Replace the data at the subtree rooted at \( z \) by \( b \).

4. \( b\.left := a; a\.left := T0; a\.right := T1; \)

5. \( b\.right := c; c\.left := T2; c\.right := T3; \)
restructure(x) Is Easy

The zig-zag case

Inorder Listing: yxz

The zig-zig case
Remove Also Uses restructure(x)

- General idea: Follow the vanilla BST procedure for removal. Finally remove a node at the leaf level which potentially changes the height of at most $O(\log n)$ nodes
- Nodes on the path from the position of the deleted key to the root may have decreased heights
- But in fact at most one node $z$ can become unbalanced
  - Let $y$ be the taller child of $z$.
  - Let $x$ be the taller child of $y$. (If both children are equally tall, pick $x$ so that $x$, $y$ and $z$ are in a “straight line” (zig-zig case).
  - $\text{restructure}(x)$ fixes the height of $z$ but may cause parent of $z$ to become unbalanced
- Repeat the above procedure at most $O(\log n)$ times
ZigZag and ZigZig Removal

The zig-zag case

The zig-zig case
Example: Cascaded Removal
Pseudocode for Update Operations

```c
void insertAVL ( Key k ) {
    node pos = insertBST ( k , root );
    replace ( pos , new AVLItem ( k , 1 ));
    // AVL trees have height , BST don’t
    rebalance ( pos );
}
```

```c
Object removeAVL ( Key k ) {
    Object t = removeBST ( k , root );
    if ( t != NO_SUCH_KEY ) {
        rebalance (( node ) t );
    }
    return t ;
}
```
Pseudocode for Update Operations

```c
void rebalance (node z) {
    // traverse path from z to root
    // to start of, z is the point of insert or remove
    // and is never unbalanced
    while ( ! isRoot(z)) {
        z = z.parent();
        setHeight(z);
        // insert or remove may have caused the height to change
        if ( ! isBalanced(z)) {
            node x = z.tallerChild().tallerChild();
            z = restructure(x); // see earlier description
            setHeight(z.leftchild()); setHeight(z.rightchild());
            setHeight(z);
        }
        // height of parent of z may need to be changed
    }
}
```
**Pseudocode for Update Operations**

```plaintext
tallerChild(node p) {
    if (p.leftChild.height() > p.rightChild.height())
        return p.leftChild;
    if (p.leftChild.height() < p.rightChild.height())
        return p.rightChild;
    if (p.parent.leftChild == p)
        return p.leftChild;
    return p.rightChild;
}

setHeight(node p) {
    p.height = 1 + max(p.leftChild().height,
                       p.rightChild().height);
}

isBalanced(node p) {
    int diff = p.leftChild().height - p.rightChild().height;
    return ((-1 <= diff) && (diff <= 1))
}
```