AVL Trees 🌴

- Binary search tree has simple search, insert, and remove
  - O(H) - H is the height of tree
  - Binary search does not guarantee small H, in worst case O(n)
- Ideal to maintain a binary search tree who's height H is O(log(N))
  - Search, insert, min, max all O(log(N))
  - Balanced Tree

Adelson- Velski and Landis

- A balanced binary search tree
- Every node in tree, height of left and right subtree only differ by 1 (at most)
- Guarantees O(Log(N))

Height of AVL Tree

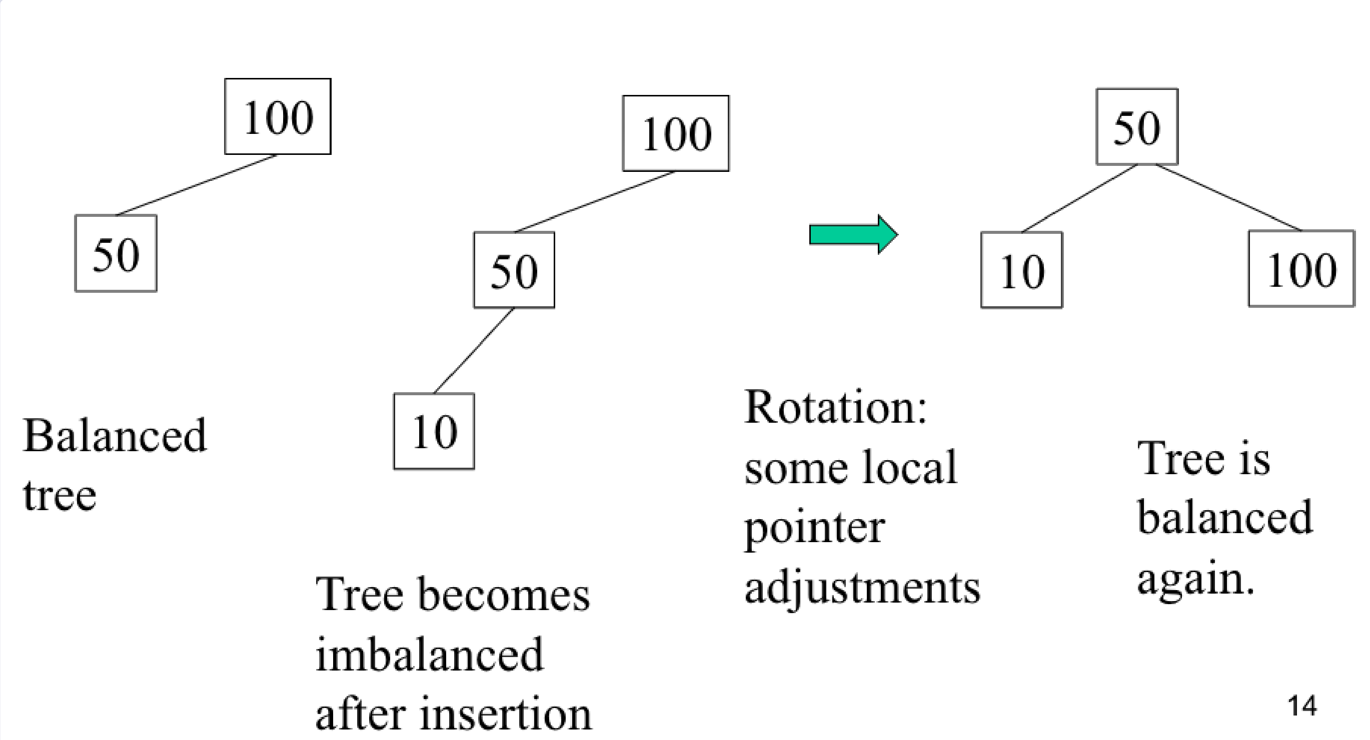
- For **every node** in tree, height of left and right subtree can differ by at most 1
- Let the number of nodes in the smallest AVL tree with height of H be N_H
  - The height of left and rightmost subtrees must have at least N_{H-2}
- Tree size at least doubles when H increases by 2, so the tree only needs to be twice the height as the full complete binary tree to have the same number of tree nodes
- **What is a balance condition**
  - The absolute difference of heights of left and right subtrees at any node is less than 1
- If we can maintain the balance condition in the insert and remove all operations to the AVL tree (with O(H) for each insert and remove we much have a data structure that archives O(log(N))) for search, insert and remove

Overhead

- Extra space needed for maintaining height information at each node, which is used to maintain balance of tree

Advantage

- Insert, Remove, and Search are all O(Log(N))



> **Important**
> Must Maintain Balance!
> This can be done through shifting the formation of the tree as shown above

Summary

- Find the deepest node whose AVL property is violated
  - Consider only the nodes from the root to the new node
- Preform the rotation

Delete

- Single rotation or double rotation, both are O(1)
- Depends on the shape of tree for single or double
- Delete can be done by deleting as in BST delete, and then fix all nodes along the path from the root to the deleted node, this would take at most O(Log(N))

Implementation

```c++
struct AvlNode
{
        Comparable      element;
        AvlNode         *left;
        AvlNode         *right;
        int                     height;
```

```
        AvlNode( const Comparable & ele, AvlNode *lt, AvlNode *rt, int
h = 0 )
          : element{ ele }, left{ lt }, right{ rt }, height{ h } { }

        AvlNode( Comparable && ele, AvlNode *lt, AvlNode *rt, int h = 0
)
          : element{ std::move( ele ) }, left{ lt }, right{ rt },
height{ h } { }
};

 /**
  * Return the height of node t or -1 if nullptr.
  */
 int height( AvlNode *t ) const
 {
        return t == nullptr ? -1 : t->height;
 }
```

Insertion

```C++
 /* Internal method to insert into a subtree.
     * x is the item to insert.
     * t is the node that roots the subtree.
     * Set the new root of the subtree.
*/
    void insert( const Comparable & x, AvlNode * & t )
    {
        if( t == nullptr )
            t = new AvlNode{ x, nullptr, nullptr };
        else if( x < t->element )
            insert( x, t->left );
        else if( t->element < x )
            insert( x, t->right );

        balance( t );
    }
```

```C++
// Assume t is balanced or within one of being balanced
    void balance( AvlNode * & t )
    {
        if( t == nullptr )
            return;

        if( height( t->left ) - height( t->right ) > ALLOWED_IMBALANCE
)
            if( height( t->left->left ) >= height( t->left->right ) )
```

```
                rotateWithLeftChild( t );
            else
                doubleWithLeftChild( t );
        else
        if( height( t->right ) - height( t->left ) > ALLOWED_IMBALANCE
)
            if( height( t->right->right ) >= height( t->right->left ) )
                rotateWithRightChild( t );
            else
                doubleWithRightChild( t );

        t->height = max( height( t->left ), height( t->right ) ) + 1;
    }
```

Single Rotation

```C++
void rotateWithLeftChild(AVLNode * &k2){
        AVLNode * k1 = k2->left;
        k2->left = k1->right;
        k1->right = k2;
        k2->height = max( height(k2->left), height( k2->right)) +1;
        k1->height = max( k1->left ), k2->height) + 1;
        k2 = k1;
}
```

Double Rotation

```C++
void doubleWithLeftChild(AVLNode * & k3){
        rotateWithRightChild(k3->left);
        rotateWithLeftChild(k3);
}
```

Remove

- Just do the binary search tree remove, and then call balance (t) at the end.

Example

```Markdown
T1, T2, T3 and T4 are subtrees.

         z                                y
        / \                             /   \
       y   T4     Right Rotate         x      z
      / \         - - - - - >         / \    / \
     x   T3                          T1  T2 T3  T4
```

```
    /  \
  T1    T2
```