# AVL Trees

The disadvantage of binary trees is that they can become very *unablanced*. Consider, as an example, inserting `n` elements that are in strictly increasing order into a binary tree. The complexity of this will be `O(n)`, and our resulting tree will have height `n`. If we could somehow keep the tree balanced, because the resulting tree will have height `O(log n)`, we will instead have complexity `O(log n)` for these `n` insertions. This can be achieved dynamically rebalancing on insertions and searches. When doing so, however, we must ensure that the *ordering invariant* of the search tree is not violated. One way to do this is by using *AVL Trees*.

**Height Invariant**

In order to describe AVL trees, we need a formalized definition of *tree height*. We define it as follows: the maximal length of a path from the root to a leaf. Thus, an empty tree has height 0, a tree with 1 leaf has height 1, a balanced tree with 3 nodes has height 2, etc. Alternatively, we can express this definition recursively by saying that an empty tree has height 0 and that the height of a tree is one more than the maximal height of its two children. We invoke this definition when expressing the *height invariant* that all AVL trees must conform to:

```
DEF: Height Invariant: At each node in an AVL tree, the height of left and right
subtrees must differ by at most 1.
```

Note that AVL trees, being balanced binary search trees, must also conform to the *ordering invariant*:

```
DEF: At any node with key k, all elements in the left subtree are strictly less than
k, while all elements in the right subtree are strictly greater.
```

If we insert into AVL trees like we do into binary trees, we're obviously going to violate this invariant often. We use *rotations* to restore it. Specifically, we can "rotate" a given node left or right. First, let's consider the code AVL-tree representation:

```
struct tree_node{
    elem data;
    struct tree_node* left;
    struct tree_node* right;
}
typedef struct tree_node tree;
```

Note that a generic `isOrdered` function implemented for binary trees works on AVL trees. We implement left rotations as follows:

```
tree* rotate_left(tree* T){
    tree* R = T->right;
    T->right = T->right->left;
    R->left = T;
    return R;
}
```

The transformation in code is made clearer by considering the actual transformation on a tree. In the simplest case, we have:

```
a                    b
  \                /   \
    b       =>   a       c
      \
        c
```

`b` becomes the new parent of `a` and `c`. If `b` has a left subtree, the result is only slightly more complicated:

```
    a                    b
  /   \                /   \
e       b       =>   a       c
      /   \        /   \
    d       c    e       d
```

Basically, we want a subtree with `a.right`, or `b` as our new root. We aren't finished, though, because we don't have `a` or anything from the left subtree of `a` in our tree now. We need to correct the left subtree of our tree. We also can't just get rid of the left subtree of `b`. In order to resolve this, we make the left subtree of `b` the new right subtree of `a`. `a` retains its original left subtree, and is inserted into our constructed subtree as the left subtree.

This is exactly what we do in code, given a tree `T`: . Initialize a tree `R` with root `T→right`. . Set the right subtree of `T`, which we'll insert as the left subtree of our constructed tree, to the left subtree of what is `b` in the above diagram. . Insert this left subtree . Return the tree we've constructed

The implementation of `rotate_right` is just the inverse:

```
tree* rotate_right(tree* T){
    tree* temp = T->left;
    T->left = T->left->right;
    temp->right = T;
    return temp;
}
```

The idea is similar to `rotate_left`. The new root is the previous left child. It's left subtree remains unchanged, but we need to preserve its right subtree along with our previous root. To do this, we

take the previous right subtree of the left subtree and make it the left subtree of the new right subtree. The right subtree of the previous root remains unchanged:

```
        c                   a
      /   \               /   \
    a       e  =>       b       c
  /   \                       /   \
b       d                   d       e
```

**Searching for a Key**

Identical to searching for a tree in any binary search tree.