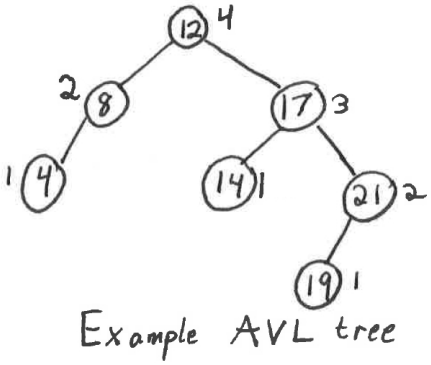
An **AVL tree** is a Binary Search Tree BST with the additional AVL property that the left and right subtrees heights differ by at most one. Every node in an AVL tree maintainss an integer height attribute in addition to the other attributes that we ordinarily see in BSTs.

The following is an example of an AVL tree, and also we see the structure of an AVL node.



Example AVL tree

```
struct AvlNode
{
    AvlNode *Left
    AvlNode *right
    int height
    int val
};
```

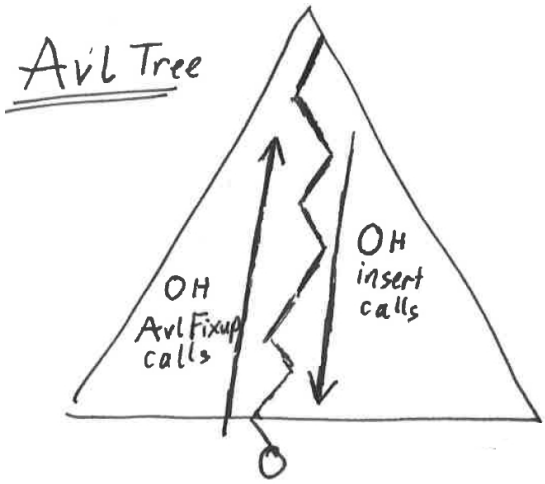
AvlNode Structure

The AVL property guarantees that the height of an AVL tree is at most O lg N. In the induction lecture, we proved that the height of an AVL tree has the height property:

$$H <= 2 \lg N$$

for any AVL tree of height 2 or more. Due to this logarythmic height property. The operations Find, Insert, and Remove all take logarythmic time. O lg N. This is because the running time of this operation is O H, and H <= 2 lg N. Therefore the runtime is O lg N.

The pseudocode for insert and remove for an AVL tree is very similar to the psudocode in a regular BST. The only difference is that the AVL versions of insert and remove also call a function called AvlFixup recursively prior to returning. These functions recursively travel down a path using the "binary search" algorithm in order to find the location of the node to insert or delete (as in an ordinary BST). However, the actual act of adding or removing a node may cause the heights of the subtrees to get out of balance.



AvlFixup uses "rotation" to ensure that the subtrees are in balance according to the AVL property. However, the act of rotating the subtrees to ensure the AVL property, may cause the parent's children to get out of balance. So upon returning from a recursive function, another AvlFixup is called by the parent further up the tree to ensure the parents subtrees obey the AVL property. Upon fixing up the

parent's children, the parent insert returns and the grandparent calls AvlFixup to ensure the grandparent's subtree obey the AVL property. And so on and so forth until we reach the root of the tree.

The pseudocode for insert is shown below. It is similar to the BST insert except for the call to AvlFixup at the bottom of the function. Avl Fixup ensures that the tree does not get too far out of balance (i.e. that it always maintains the AVL property).

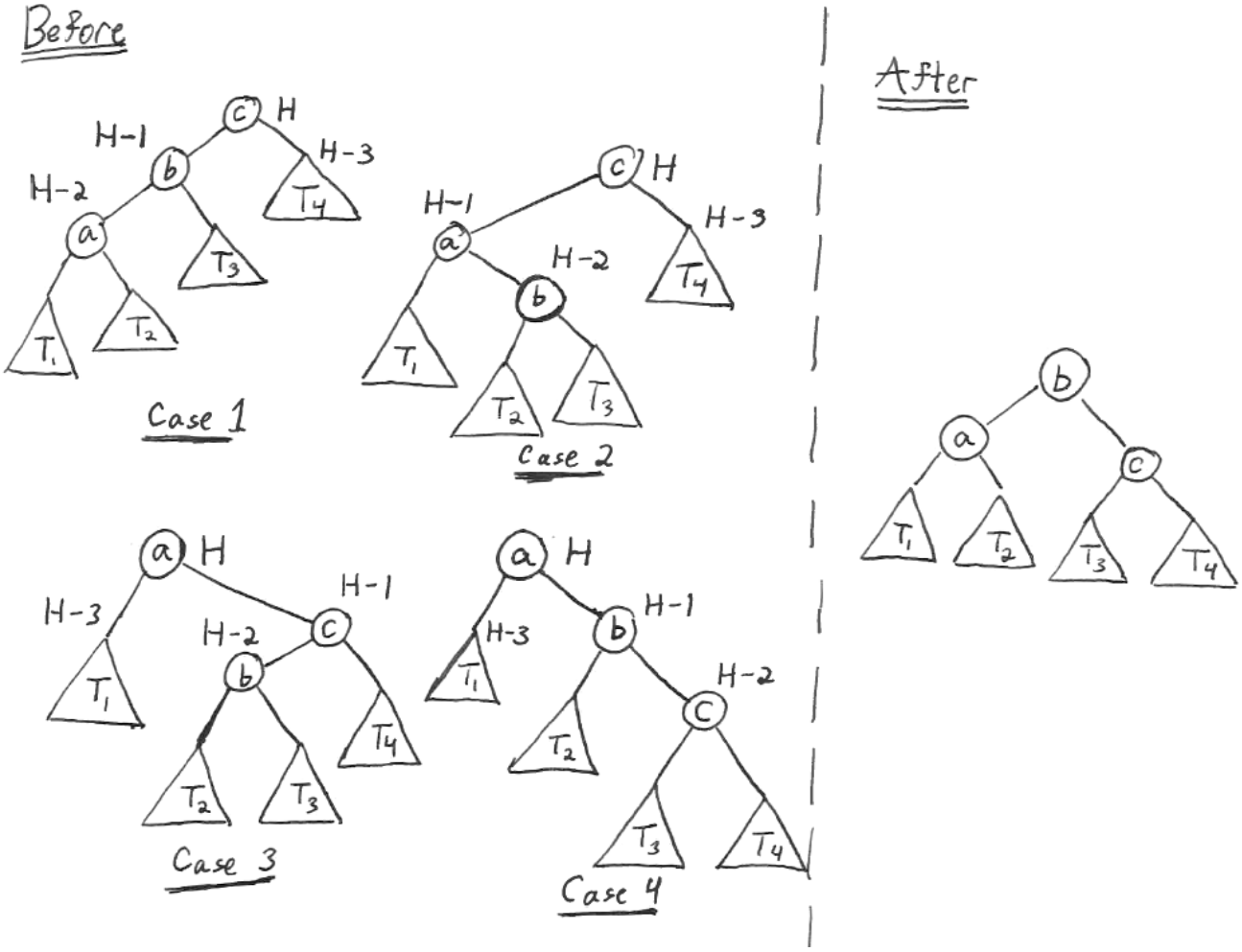
```
insert (x, val)
{
   if (x == Null)
      root = new AvlNode (val)
   else if (val < x→val)
   {
      if (x→L == Null)
         x→L = new AvlNode (val)
      else
         insert (x→L, val)
   }
   else if (x→val < val)
   {
      if (x→R == Null)
         x→R = new AvlNode (val)
      else
         insert (x→R, val)
   }
   else
      x→val = val
   AvlFixup (x, val)      // this one line of code
}                         // ensures the AVL tree
                          // does not get out of
                          // balance, and maintains
                          // the AVL property
```

The pseudocode for Remove is similar, although slightly more complicated. This is because Remove for an ordinary BST has a case in which we must find successor in the event that the BST has two children. It is important that the recursive implementation of find successor would call AvlFixup also as it traverses back up the tree.

The pseudocode for Remove is not included in this handout. However it is very similar to the following "insert" pseudocode. He key point is that at every level in the recursion we need to call AvlFixup to ensure the AVL propert is not violated by any level of reursion in the tree.

AvlFixup must ensure the AVL property, this means that the left and right subtrees must differ in height by at most one. This means that AVL fixup must be able to modify and fix the case in which the heights of the left and right subtrees differ by two.

There are four cases in which the left and right subtrees are out of balance. In all four cases, we see trees of slightly different starting shapes, but we change the pointers of the trees such that the ending shape is the same and exhibit the AVL property. The following are the four cases.



In all four cases, we see the following properties

there are four subtrees: **T1 T2 T3 T4**
there are three internal nodes: **a, b, c**

And the shape of the tree maintains the following ordering of nodes

T1 < a < T2 < b < T3 < c < T4

So we see that the cases improve the shape of the tree but they do not change the orderirng of the nodes

The four cases have the following properties

Case 1 & 2 left: H-1 right: H-3 left subtree is too high

Case 2 & 3 left: H-3 right: H-1 right subtree is too high

And more specifically, we observe the following

Case 1	left: H-1	left-left: H-2	right: H-3
Case 2	left: H-1	left-right: H-2	right: H-3
Case 3	left: H-3	right: H-1	right-left: H-2
Case 4	left: H-3	right: H-1	right-right: H-2

So AvlFixup has the following pseudocode

(Note: for simplicity this pseudocode assumes that a null node will return a height of zero. a real implementation would need to check for this case).

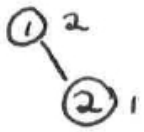
```
AvlFixup (x)
{
    x->H = 1 + max (x->L->H, x->R->H)
    if (x->L->H and x->R->H differ by at most one)
        return
    else if (x->R->H == H-3)                    // case 1 & 2
    {
        if (x->L->L->H == H-2)                  // case 1
        {
            a = x->L->L
            b = x->L
            c = x
            T1 = a->L
            T2 = a->R
            T3 = b->R
            T4 = c->R
        }
        else                                    // case 2
        {
            a = x->L
            b = x->L->R
            c = x
            T1 = a->L
            T2 = b->L
            T3 = b->R
            T4 = c->R
        }
    }
    else                                        // case 3 & 4
    {
```

```
if (x→R→L→H = H-2) //case 3
{
    a = x
    b = x→R→L
    c = x→R
    T1 = a→L
    T2 = b→L
    T3 = b→R
    T4 = c→R
}
else                                    // case 4
{
    a = x
    b = x→R
    c = x→R→R
    T1 = a→L
    T2 = b→L
    T3 = c→L
    T4 = c→R
}
// now we have our pointers for
// a T1 < a < T2 < b < T3 < c < T4
// so exchange pointers to the desired shape
x = b
b→L = a
b→R = c
a→L = T1
a→R = T2
c→L = T3
c→R = T4
a→H = 1 + max(T1→H, T2→H)      //adjust heights
c→H = 1 + max(T3→H, T4→H)
b→H = 1 + max(a→H, c→H)
}
```

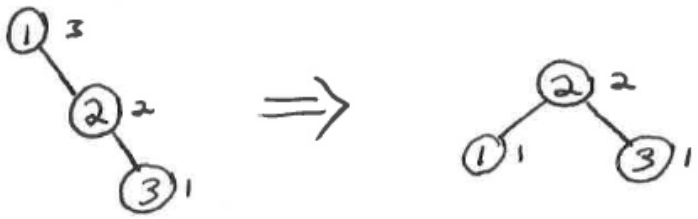
AvL Tree T
for i=1...8
  T.insert(i)

insert 1

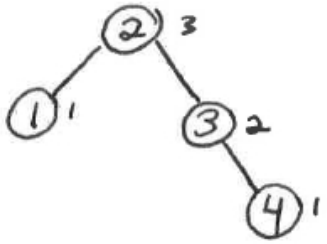
insert 2



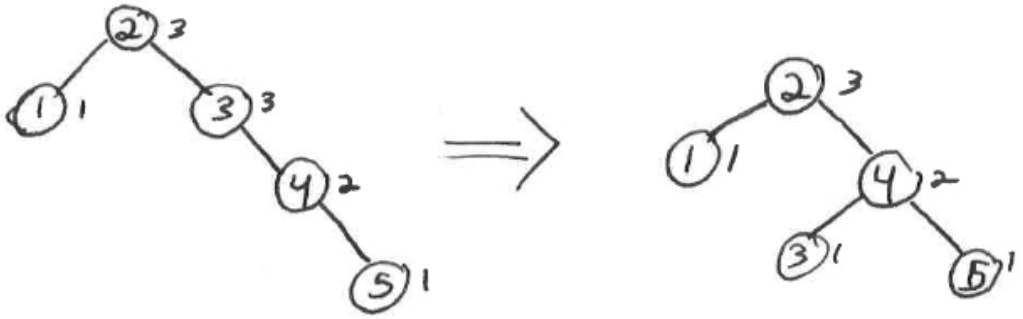
insert 3



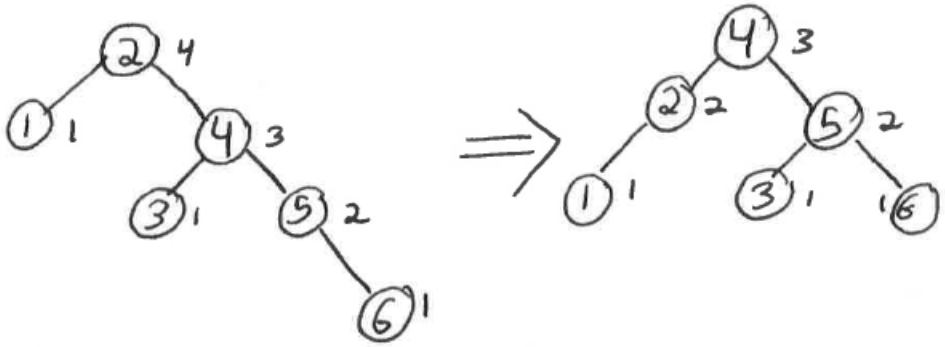
insert 4



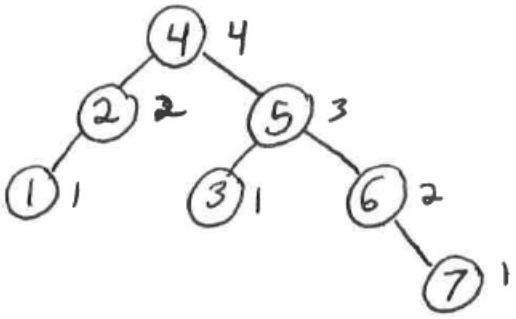
insert 5



insert 6



insert 7



insert 8