COMS 3137 Class Notes

1 AVL Trees

- When we do an insert into a Binary Search Tree (BST), we can never be sure how balanced the tree will be, since the order of insertions will determine this. A solution is to create balanced BST's as we do the insertions. An AVL tree is such a tree.
- The balance condition is this: In an AVL tree, the height of the left and right subtrees of the root differ by at most 1, and in which the left and right subtrees are AVL trees also. This balance condiiton insures that searches, inseretions and deletetion will be close to $O(Log_2 N)$, as in a fully balanced BST.
- AVL trees keep an additional piece of information at each node: the height of each nodes left and right subtrees. Since these heights can only differ by 1, any insertion into the tree will be analyzed to see if it violates the balance condition. If it does, then we need to reorganize the tree by a series of *rotations*. Rotations are simply pointer changes that rearrange the structure of the tree to keep the AVL balance condition. The Weiss textbook has some good examples on rotations that explain how they work.

2 Height of AVL Trees

Proposition: The height of an AVL tree T storing n keys is O(log n).

Justification: The easiest way to approach this problem is to try to find the minimum number of internal nodes of an AVL tree of height h: $N(h)$. It is obvious that that $N(1) = 1$ and $N(2) = 2$. For $h \geq 3$, an AVL tree of height h with $N(h)$ minimal internal nodes contains the root node, one AVL subtree of height *h-1* and the other AVL subtree of height $h - 2$.

$$
\begin{aligned}
N(h) &= 1 + N(h-1) + N(h-2) && (1)\\
Since: \quad N(h-1) &> N(h-2) && (2)\\
N(h) &> 2N(h-2) \ \ (substitute\ in\ (1)\ \ above,\ drop\ the\ +1\ term) && (3)\\
N(h) &> 4N(h-4) && (4)\\
&\ldots && (5)\\
N(h) &> 2^i\, N(h-2i) && (6)
\end{aligned}
$$

now choose $i = \frac{h}{2} - 1$, and substituting in (6):

$$
\begin{aligned}
N(h) &> 2^{\frac{h}{2}-1}\, N(h - 2(\frac{h}{2} - 1)) && (7)\\
N(h) &> 2^{\frac{h}{2}-1}\, N(2) && (8)\\
N(h) &> 2^{\frac{h}{2}-1} \cdot 2 && (9)\\
N(h) &> 2^{\frac{h}{2}} && (10)\\
logN(h) &> \frac{h}{2} && (11)\\
2logN(h) &> h && (12)
\end{aligned}
$$

Thus the height of an AVL tree is O(log n)

3 AVL Rotation Templates

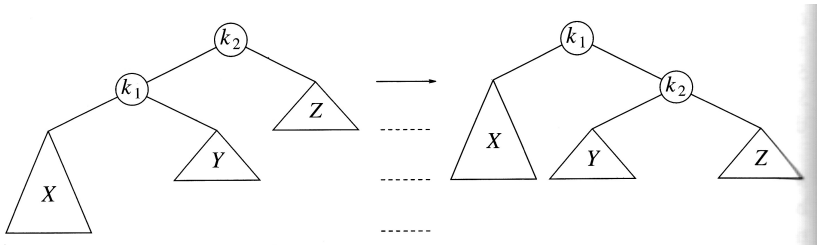


Figure 4.31 Single rotation to fix case 1

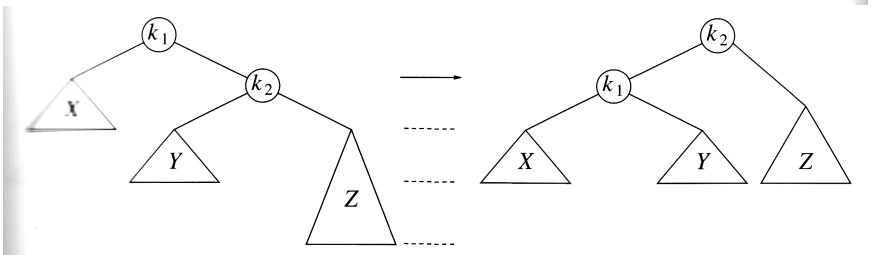


Figure 4.33 Single rotation fixes case 4

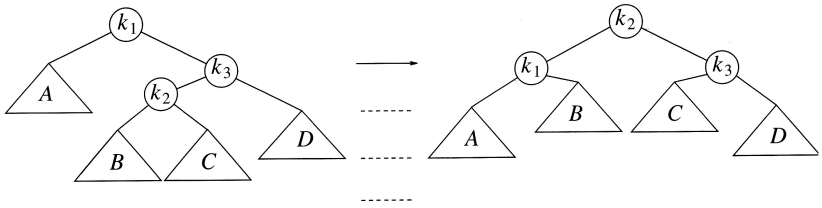


Figure 4.36 Right–left double rotation to fix case 3

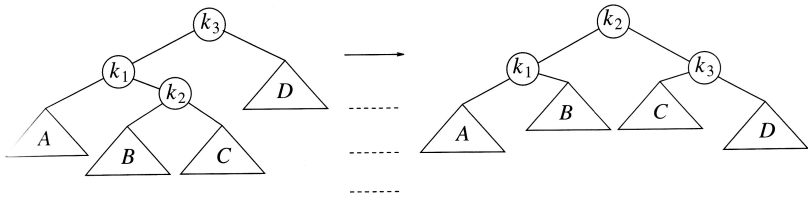


Figure 4.35 Left–right double rotation to fix case 2

Figure 1: Templates for LL, RR, LR, RL AVL rotations from Weiss textbook

Here is the Java code from the Weiss textbook for AVL insertion

```
/**
     * Internal method to insert into a subtree.
     * @param x the item to insert.
     * @param t the node that roots the subtree.
     * @return the new root of the subtree.
     */
    private AvlNode<AnyType> insert( AnyType x, AvlNode<AnyType> t )
    {
        if( t == null )
            return new AvlNode<>( x, null, null );

        int compareResult = x.compareTo( t.element );

        if( compareResult < 0 )
            t.left = insert( x, t.left );
        else if( compareResult > 0 )
            t.right = insert( x, t.right );
        else
            ;  // Duplicate; do nothing
        return balance( t );
    }

 private static final int ALLOWED_IMBALANCE = 1;

    // Assume t is either balanced or within one of being balanced
    private AvlNode<AnyType> balance( AvlNode<AnyType> t )
    {
        if( t == null )
            return t;

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

        t.height = Math.max( height( t.left ), height( t.right ) ) + 1;
        return t;
    }
```

```
1       /**
2        * Rotate binary tree node with left child.
3        * For AVL trees, this is a single rotation for case 1.
4        * Update heights, then return new root.
5        */
6       private AvlNode<AnyType> rotateWithLeftChild( AvlNode<AnyType> k2 )
7       {
8           AvlNode<AnyType> k1 = k2.left;
9           k2.left = k1.right;
10          k1.right = k2;
11          k2.height = Math.max( height( k2.left ), height( k2.right ) ) + 1;
12          k1.height = Math.max( height( k1.left ), k2.height ) + 1;
13          return k1;
14      }
```

Figure 4.41 Routine to perform single rotation

```
1       /**
2        * Double rotate binary tree node: first left child
3        * with its right child; then node k3 with new left child.
4        * For AVL trees, this is a double rotation for case 2.
5        * Update heights, then return new root.
6        */
7       private AvlNode<AnyType> doubleWithLeftChild( AvlNode<AnyType> k3 )
8       {
9           k3.left = rotateWithRightChild( k3.left );
10          return rotateWithLeftChild( k3 );
11      }
```

Figure 4.43 Routine to perform double rotation

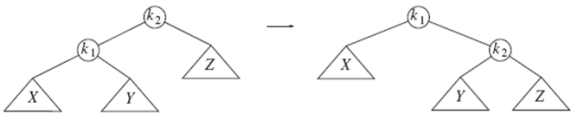


Figure 4.40 Single rotation

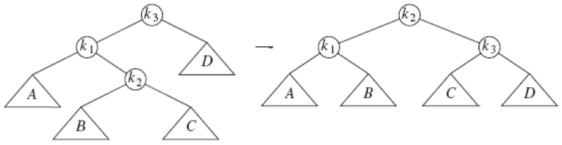


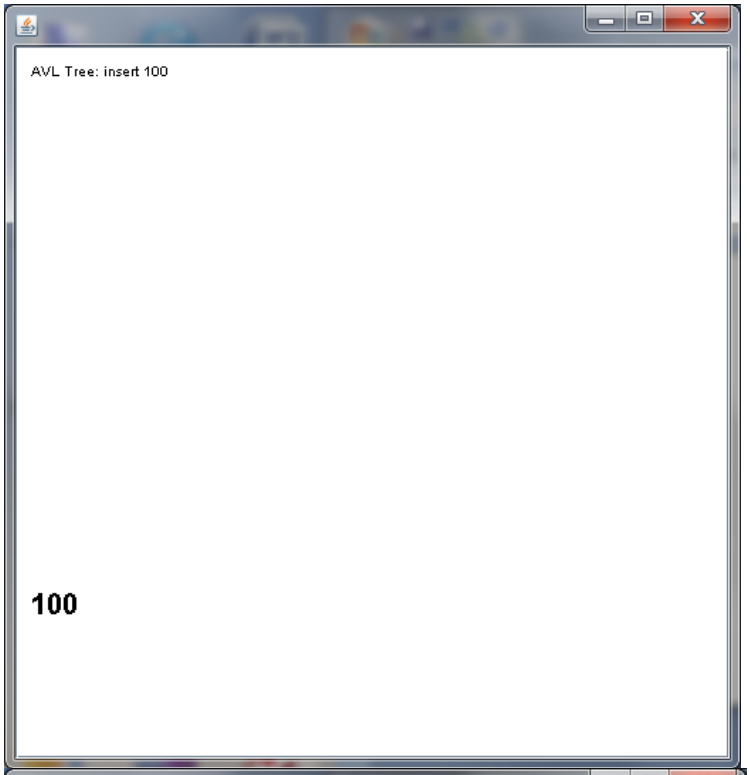
Figure 4.42 Double rotation

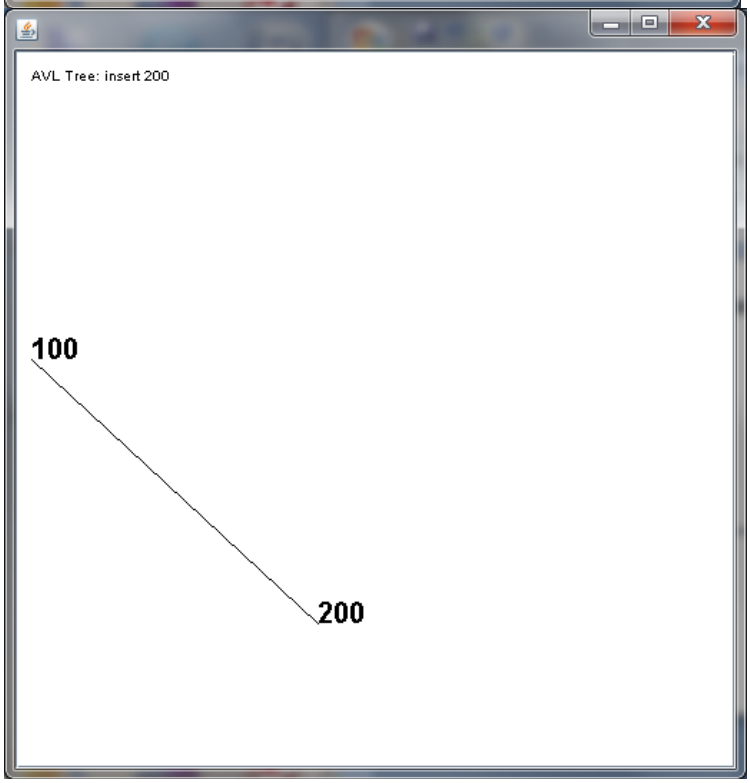
```
/**     * Rotate binary tree node with left child.
     * For AVL trees, this is a single rotation for case 1.
     * Update heights, then return new root. */     */
    private AvlNode<AnyType> rotateWithLeftChild( AvlNode<AnyType> k2 )
    {
        AvlNode<AnyType> k1 = k2.left;
        k2.left = k1.right;
        k1.right = k2;
        k2.height = Math.max( height( k2.left ), height( k2.right ) ) + 1;
        k1.height = Math.max( height( k1.left ), k2.height ) + 1;
        return k1;
    }

    /**     * Rotate binary tree node with right child.
     * For AVL trees, this is a single rotation for case 4.
     * Update heights, then return new root.     */
    private AvlNode<AnyType> rotateWithRightChild( AvlNode<AnyType> k1 )
    {
        AvlNode<AnyType> k2 = k1.right;
        k1.right = k2.left;
        k2.left = k1;
        k1.height = Math.max( height( k1.left ), height( k1.right ) ) + 1;
        k2.height = Math.max( height( k2.right ), k1.height ) + 1;
        return k2;
    }

    /**     * Double rotate binary tree node: first left child
     * with its right child; then node k3 with new left child.
     * For AVL trees, this is a double rotation for case 2.
     * Update heights, then return new root.     */
    private AvlNode<AnyType> doubleWithLeftChild( AvlNode<AnyType> k3 )
    {
        k3.left = rotateWithRightChild( k3.left );
        return rotateWithLeftChild( k3 );
    }

    /**     * Double rotate binary tree node: first right child
     * with its left child; then node k1 with new right child.
     * For AVL trees, this is a double rotation for case 3.
     * Update heights, then return new root.     */
    private AvlNode<AnyType> doubleWithRightChild( AvlNode<AnyType> k1 )
    {
        k1.right = rotateWithLeftChild( k1.right );
        return rotateWithRightChild( k1 );
    }
```

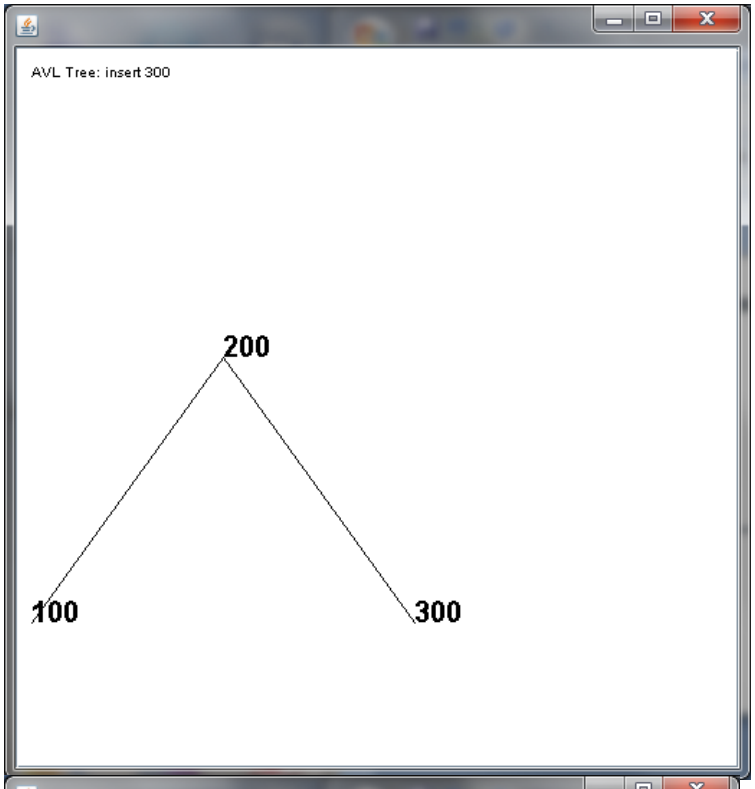
Example: Below is the output of insertions into an AVL tree. The AVL code and all code in the textbook is at: `http://users.cis.fiu.edu/~weiss/dsaajava3/code/`

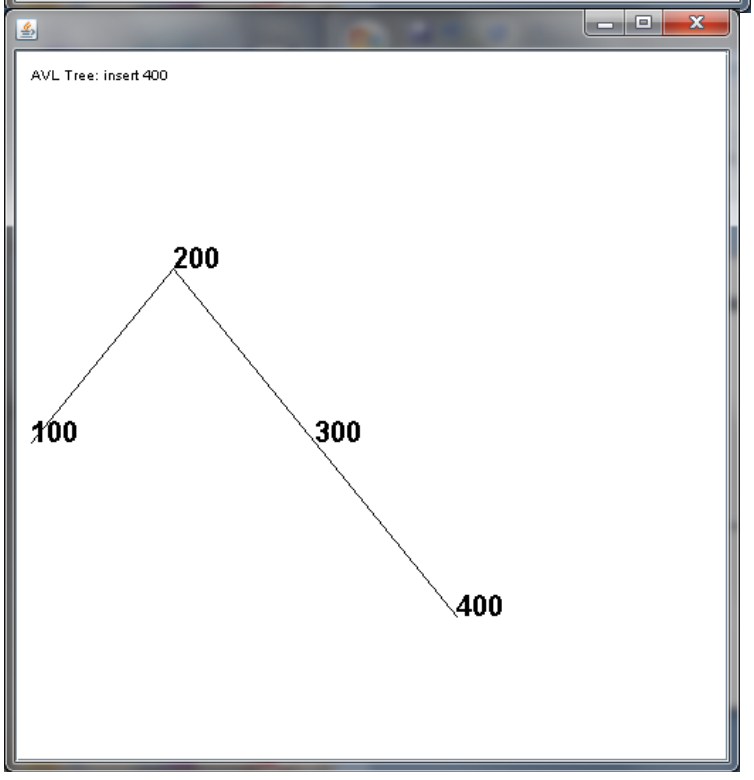
AVL Tree Example:

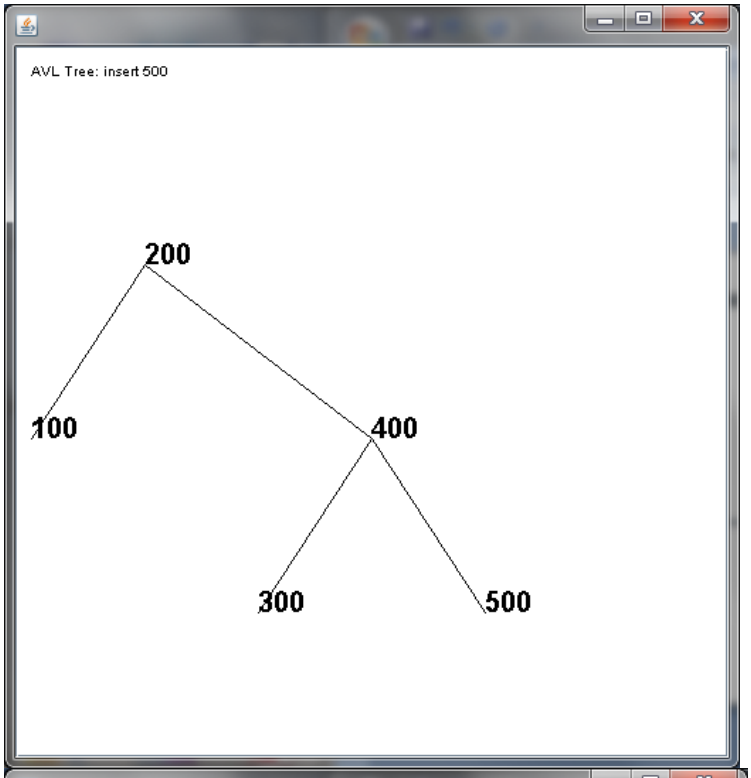
Insert 100, 200, 300, 400, 500, 600, 700, 450, 475, 550, 800, 750, 725, 10, 5, 150, 350

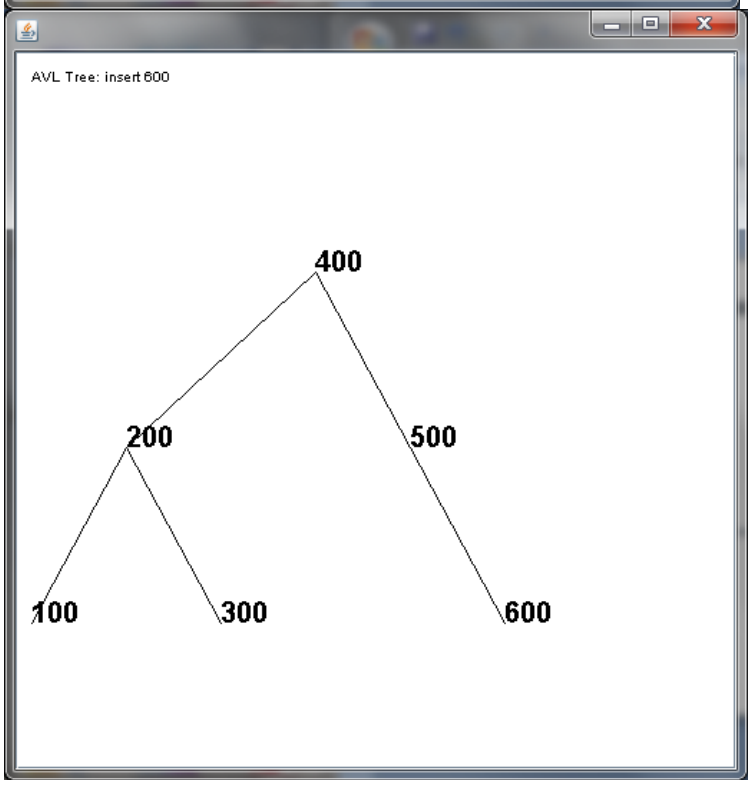


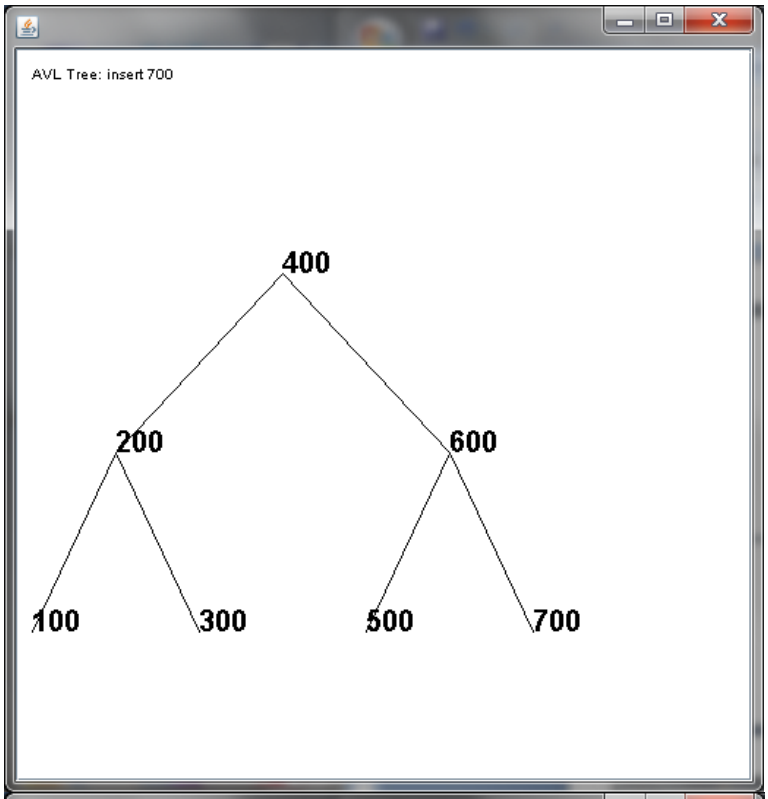


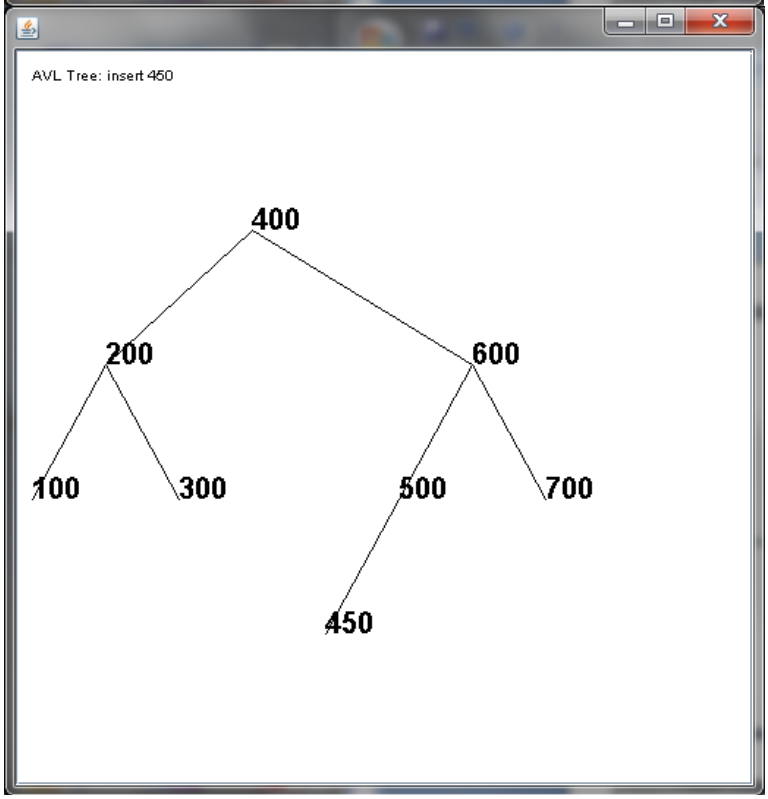

AVL Tree: insert 300
200
100
300

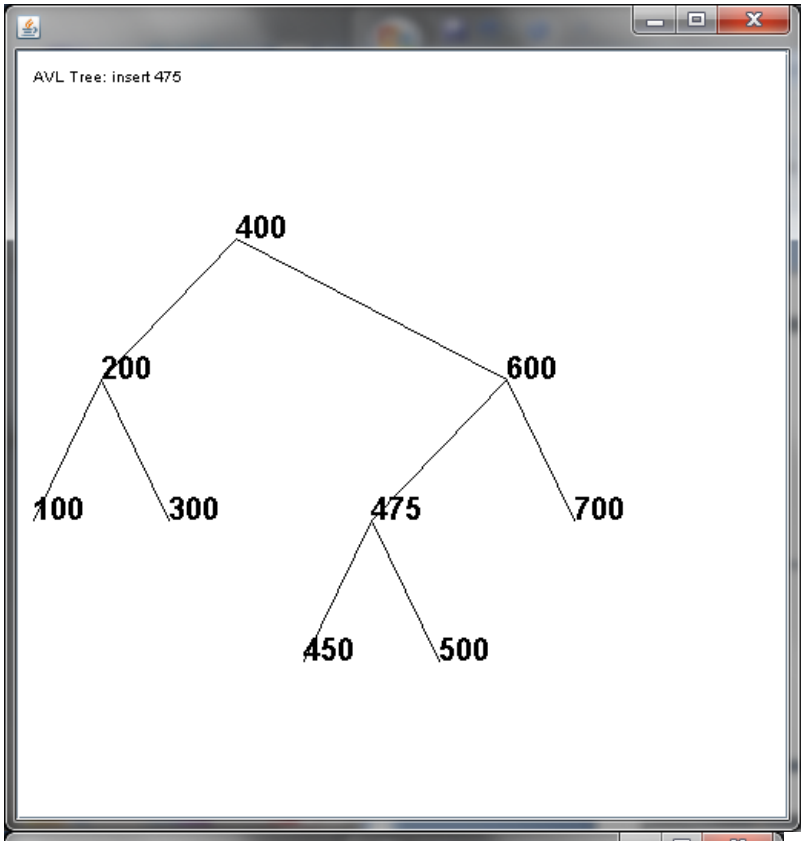

AVL Tree: insert 400
200
100
300
400

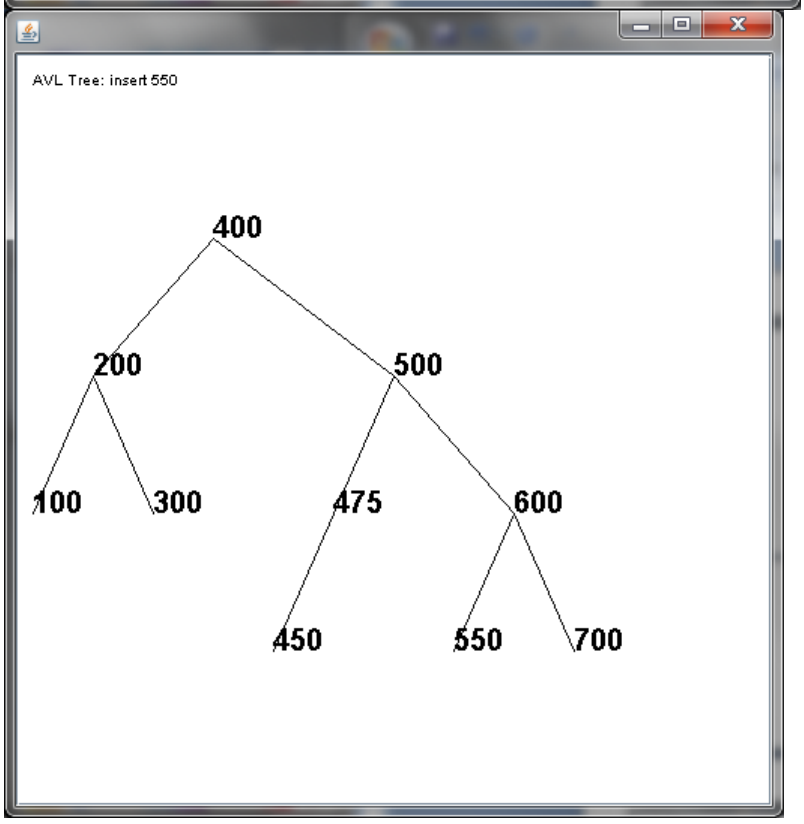

AVL Tree: insert 500
200
100
400
300
500

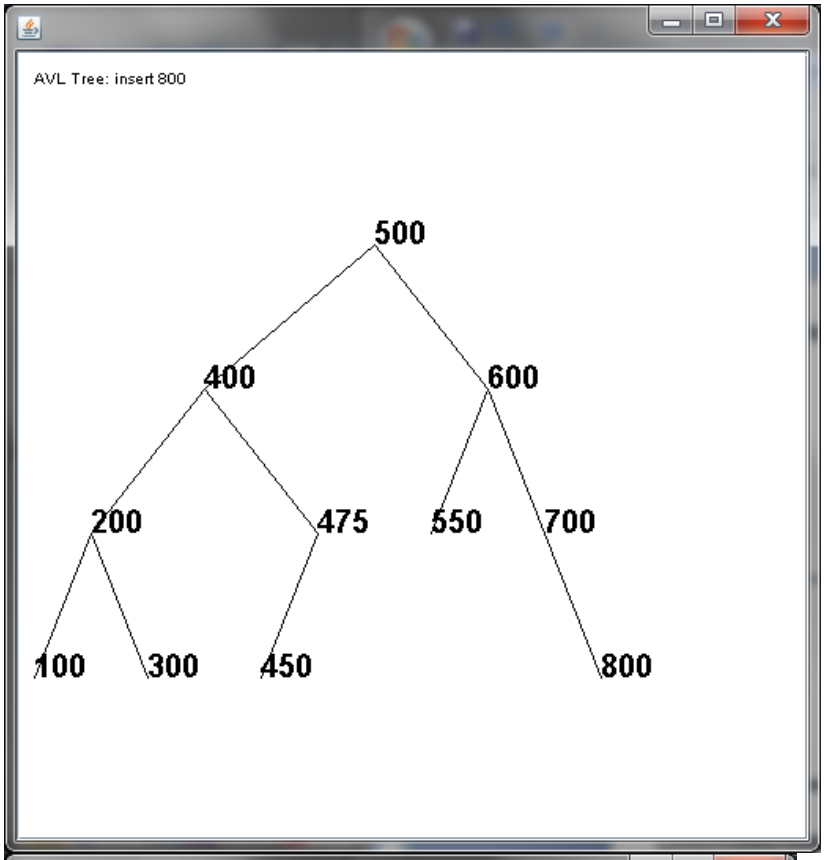

AVL Tree: insert 600
400
200
500
100
300
600

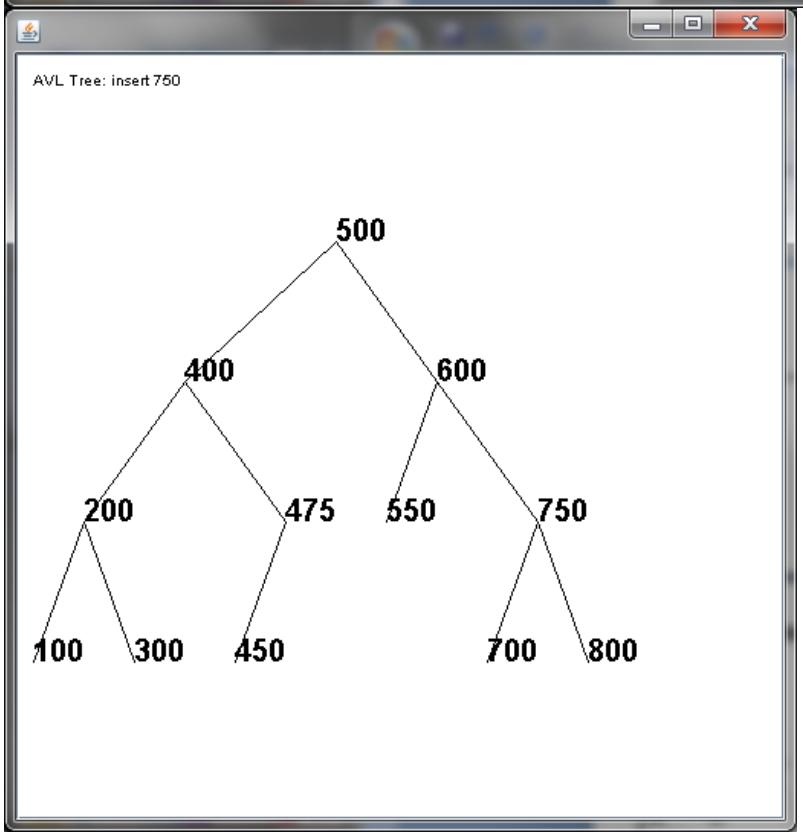

AVL Tree: insert 700
400
200
600
100
300
500
700

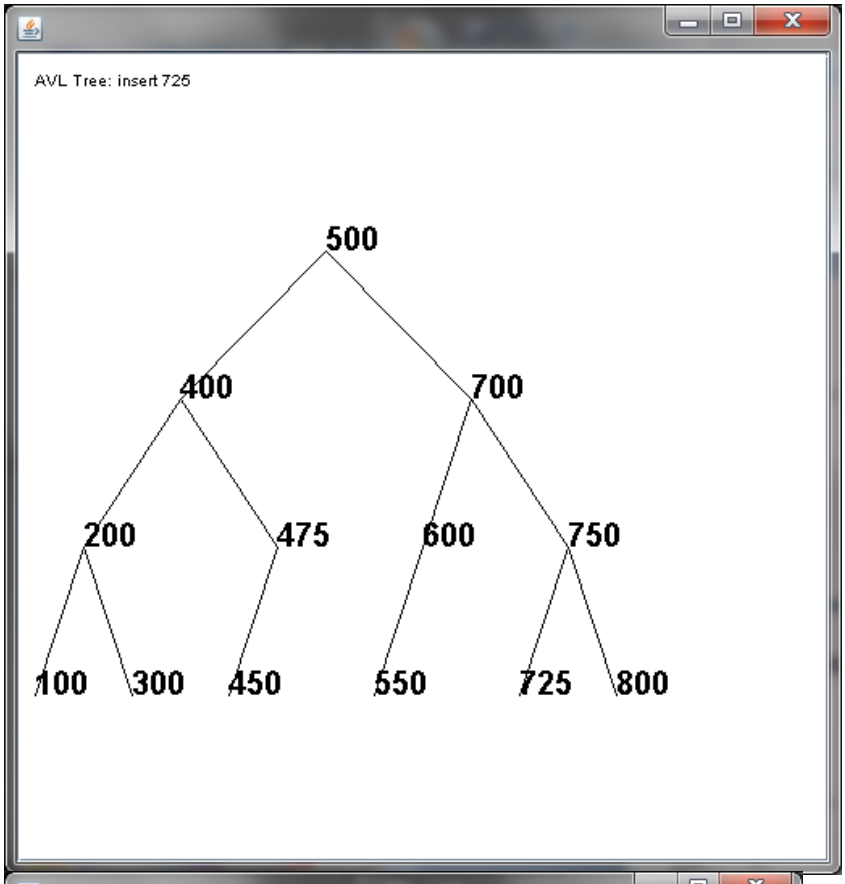

AVL Tree: insert 450
400
200
600
100
300
500
700
450

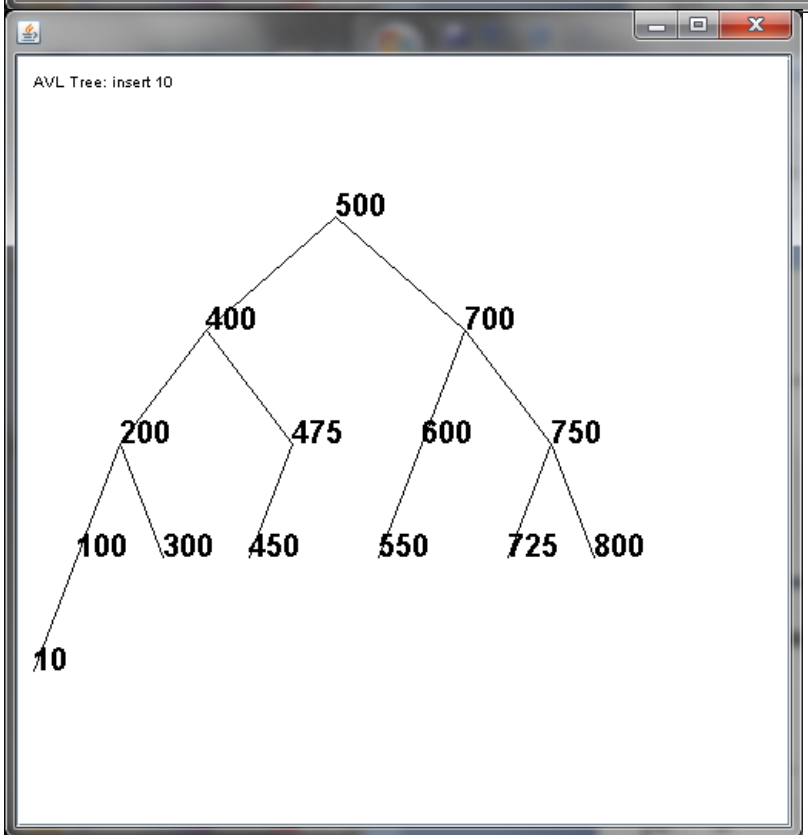

AVL Tree: insert 475
400
200
600
100
300
475
700
450
500

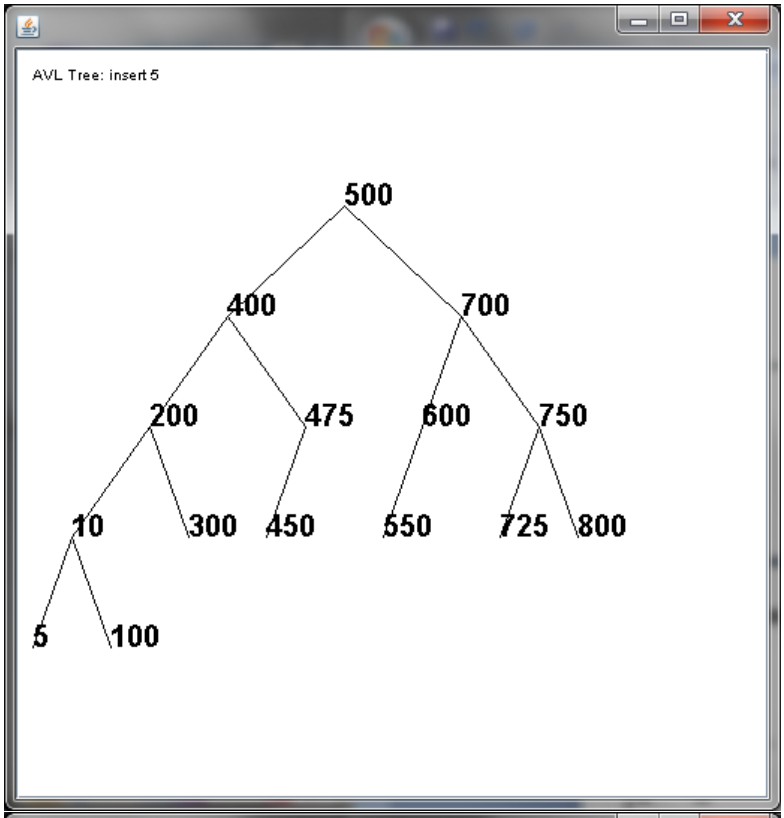

AVL Tree: insert 550
400
200
500
100
300
475
600
450
550
700

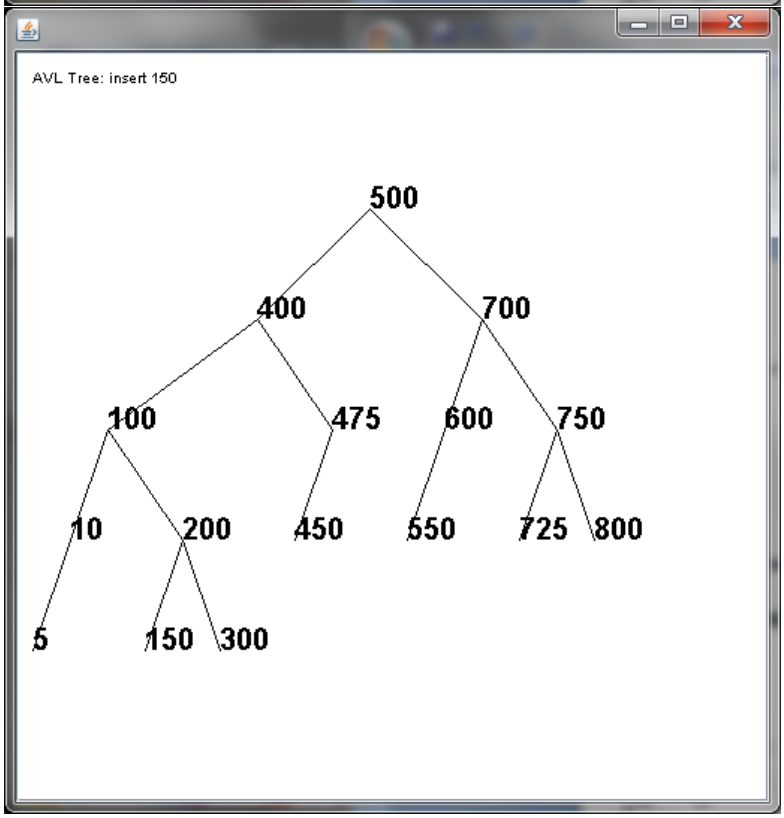

AVL Tree: insert 800
500
400
600
200
475
550
700
100
300
450
800

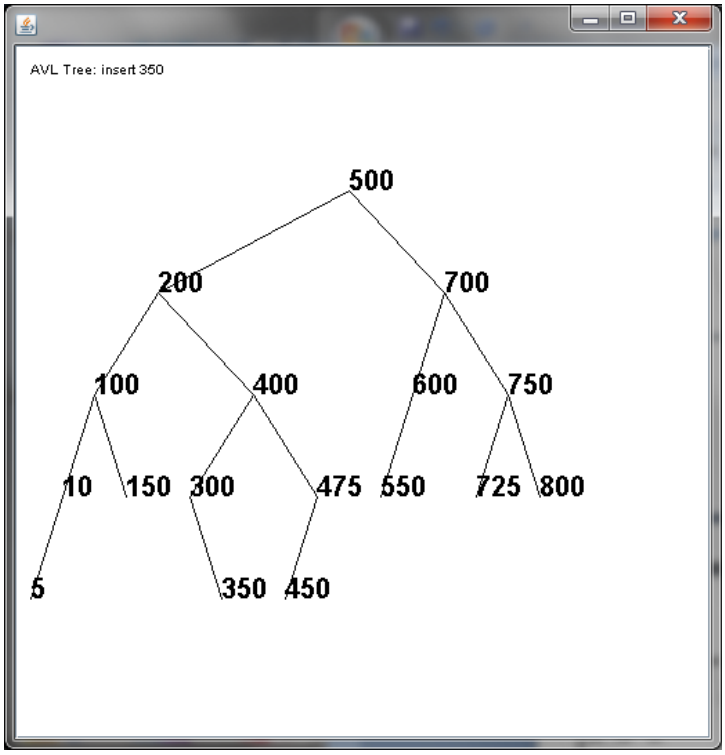

AVL Tree: insert 750
500
400
600
200
475
550
750
100
300
450
700
800


AVL Tree: insert 725
500
400
700
200
475
600
750
100
300
450
550
725
800


AVL Tree: insert 10
500
400
700
200
475
600
750
100
300
450
550
725
800
10


AVL Tree: insert 5
500
400
700
200
475
600
750
10
300
450
550
725
800
5
100


AVL Tree: insert 150
500
400
700
100
475
600
750
10
200
450
550
725
800
5
150
300


AVL Tree: insert 350
500
200
700
100
400
600
750
10
150
300
475
550
725
800
5
350
450