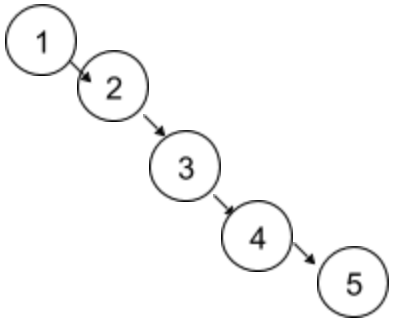
Worksheet 31: AVL Trees

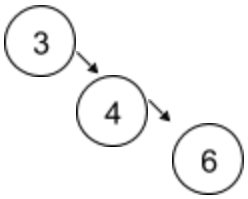
Group Members:
Erin Alltop
Nathan Fraser
Joshua Kevin Groves
David Luke Hartman
Matthew Toro

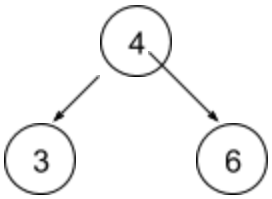
In Preparation: Read Chapters 8 and 10 on Bags and Trees, respectively. In you have not done so ready, do Worksheets 29 and 30 on Binary Search Trees.



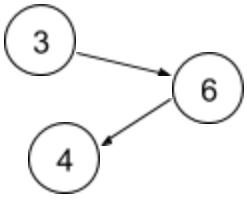
The time required to perform operations on a binary search tree is proportional to the length of the path from root to leaf. This isn't bad in a well-balanced tree. But nothing prevents a tree from becoming unbalanced. In the extreme, such as the tree shown on the right, the tree reduces to nothing more than a linked list, and the path from root to leaf is O(n).

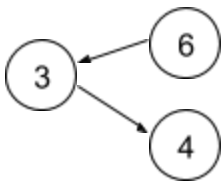
To preserve fast performance we need to ensure that the tree remains well balanced. One way to do this is to notice that the search tree property has some flexibility. Three nodes that are unbalanced can be restored by a *rotation*, making the right child into the new root, with the previous root as the new left child. Any existing left child of the old right child becomes the new right child of the new left child. The resulting tree is still a binary search tree, and has better balance.



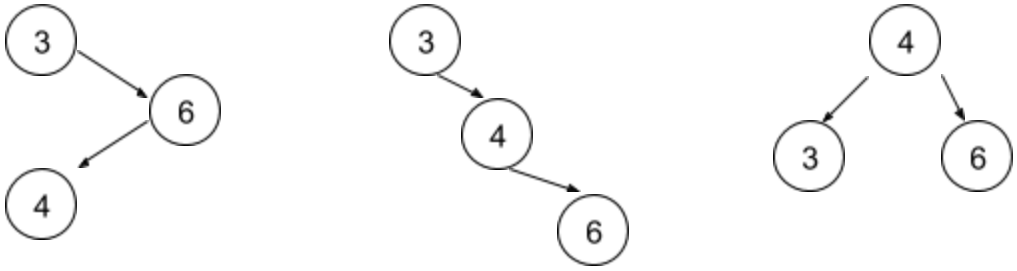


This is known as a *left rotation*. There is also a corresponding right rotation. There is one case where a simple rotation is not sufficient. Consider an unbalanced tree with a right child that itself has a left child. If we perform a rotation, the result is still unbalanced.





The solution is to first perform a rotation on the child, and then rotate the parent. This is termed a *double rotation*.



The data structure termed the AVL tree was designed using these ideas. The name honors the inventors of the data structure, the Russian mathematicians G. M. Adelson-Velskii and E.M. Landis.

In order to know when to perform a rotation, it is necessary to know the height of a node. We could calculate this amount, but that would slow the algorithm. Instead, we modify the Node so that each node keeps a record of its own height.

```
struct AVLnode {
   TYPE value;
   struct AVLnode *left;
   struct AVLnode *right;
   int height;
};
```

A function h(Node) will be useful to determine the height of a child node. Since leaf nodes have height zero, a value of -1 is returned for a null value. Using this, a function setHeight can be defined that will set the height value of a node, assuming that the height of the child nodes is known:

```
int _h(struct AVLnode * current)
      {if (current == 0) return -1;  return current->height;}

void _setHeight (struct AVLnode * current) {
     int lch = h(current->left);
     int rch = h(current->right);
     if (lch < rch) current->height = 1 + rch;
     else current->height = 1 + lch;
  }
```

Armed with the height information, the AVL tree algorithms are now easy to describe. The addition and removal algorithms for the binary search tree are modified so that their very last step is to invoke a method **balance**:

```
struct AVLnode * _AVLnodeAdd (struct AVLnode* current, TYPE newValue) {
   struct AVLnode * newnode;
   if (current == 0) {
      newnode = (struct AVLnode *) malloc(sizeof(struct AVLnode));
      assert(newnode != 0);
      newnode->value = newValue;
      newnode->left = newnode->right = 0;
      newNode->height = 0; //Error in the worksheet
```

```
      return newnode;  //why don't we balance here ??
   } else if (LT(newValue, current->value))
      current->left = AVLnodeAdd(current->left, newValue);
   else current->right = AVLnodeAdd(current->right, newValue);
   return balance(current);    /* <-   NEW  the call on balance */
}
```

The function balance performs the rotations necessary to restore the balance in the tree. Let the *balance factor* be the difference in height between the right and left child trees. This is easily computed using a function. If the balance factor is more than 2, that is, if one subtree is more than two levels different in height from the other, then a rebalancing is performed. A check must be performed for double rotations, but again this is easy to determine using the balance factor function. Once the tree has been rebalanced the height is set by calling setHeight:

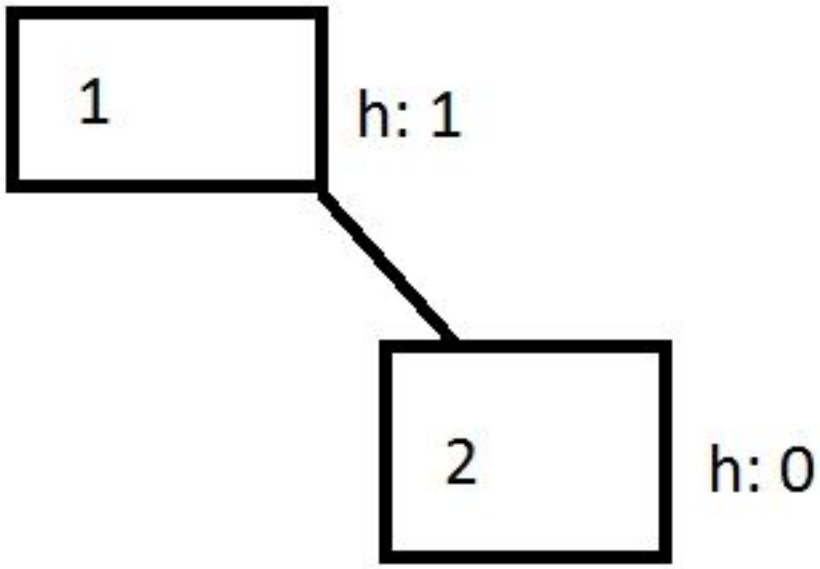
```
int _bf (struct AVLnode * current)
        { return h(current->right) - h(current->left); }

struct AVLnode * _balance (struct AVLnode * current) {
     int cbf = bf(current);
     if (cbf < -1) {
        if (bf(current->left) > 0) // double rotation
           current->left = rotateLeft(current->left);
        return rotateRight(current); // single rotation
     } else if (cbf > 1) {
        if (bf(current->right) < 0)
           current->right = rotateRight(current->right);
        return rotateLeft(current);
     }
     setHeight(current);
     return current;
  }
```

Since the balance function looks only at a node and its two children, the time necessary to perform rebalancing is proportional to the length of the path from root to leaf.

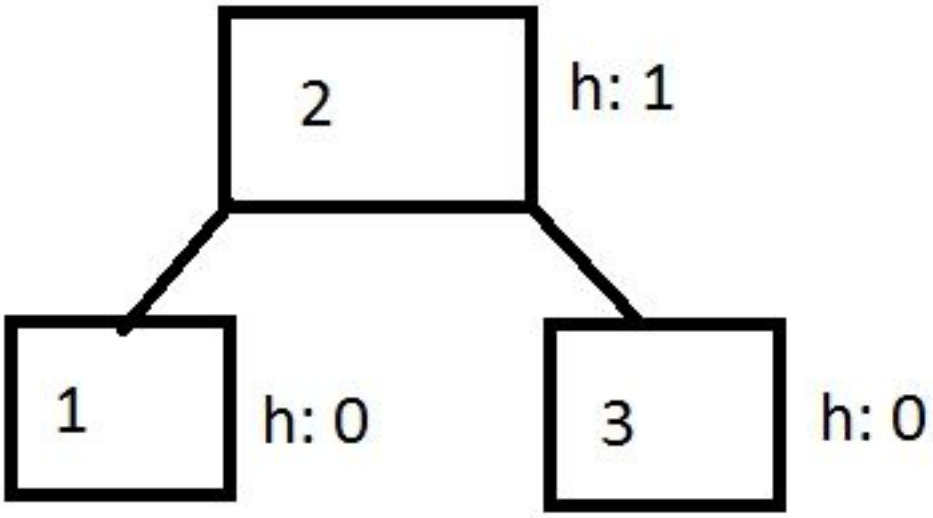
Insert the values 1 to 7 into an empty AVL tree and show the resulting tree after each step. Remember that rebalancing is performed bottom up after a new value has been inserted, and only if the difference in heights of the child trees are more than one.

Step 1

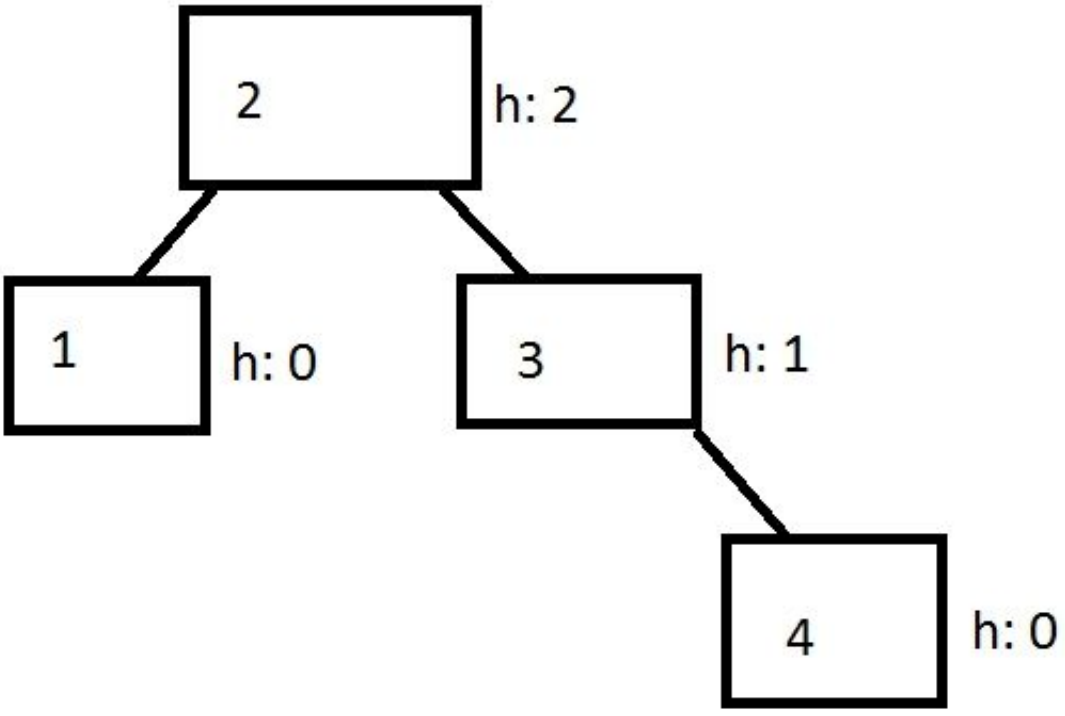
Step 2

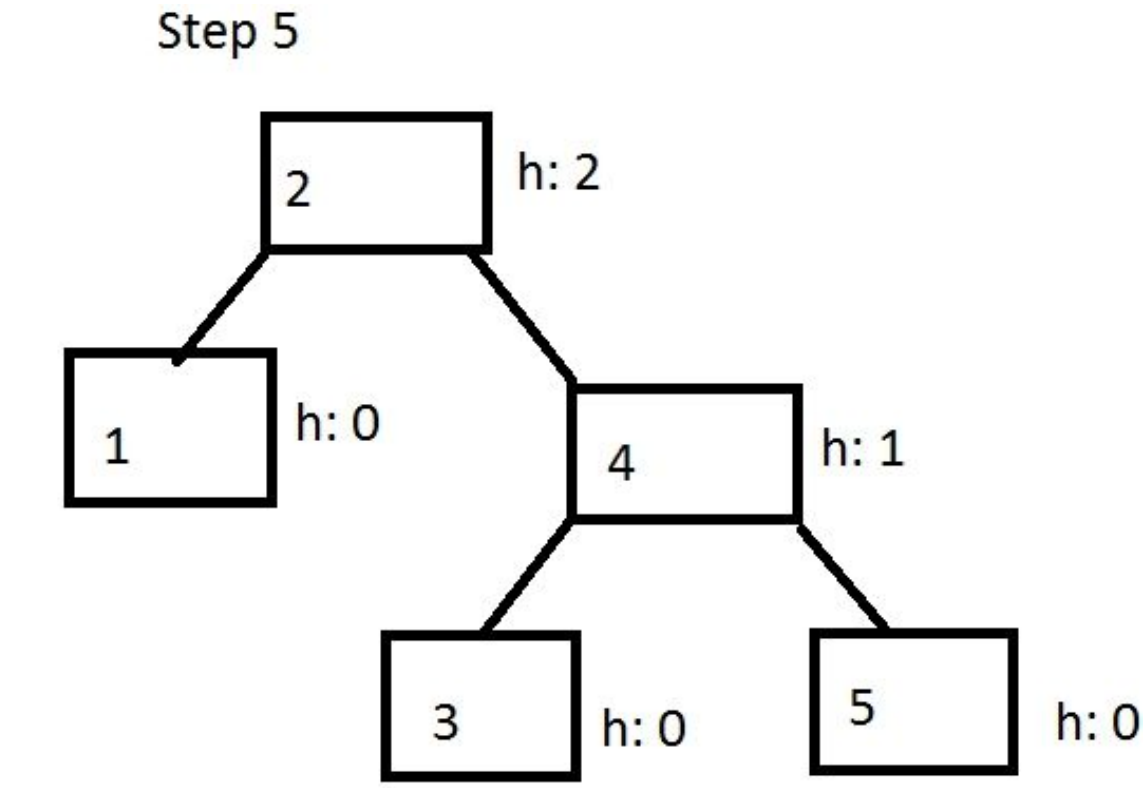


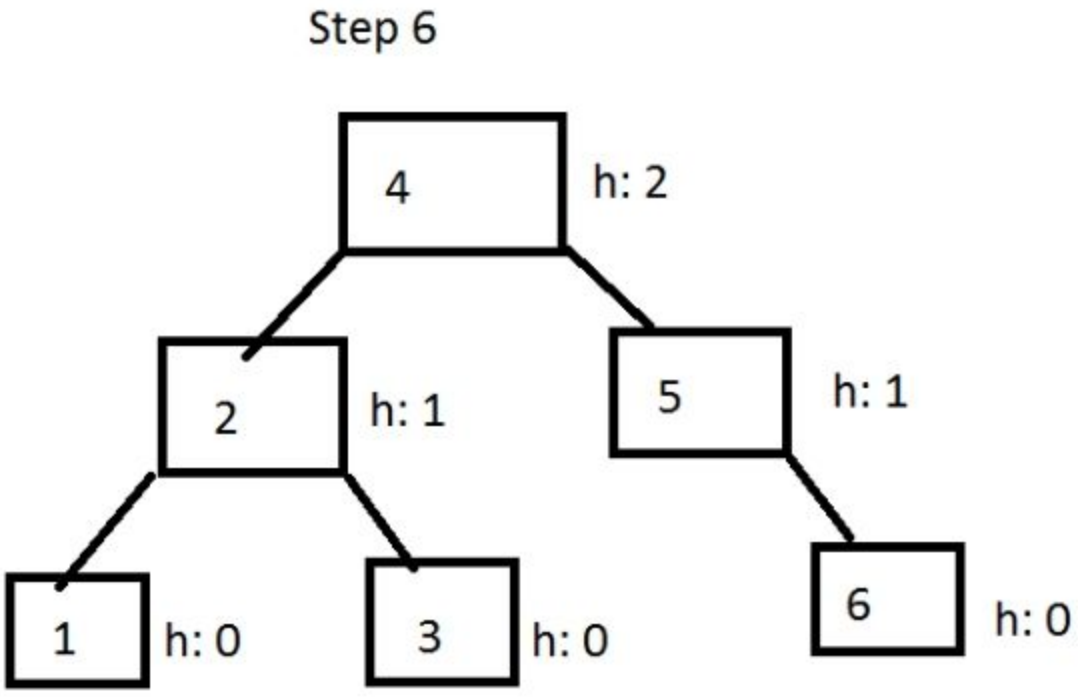
Step 3

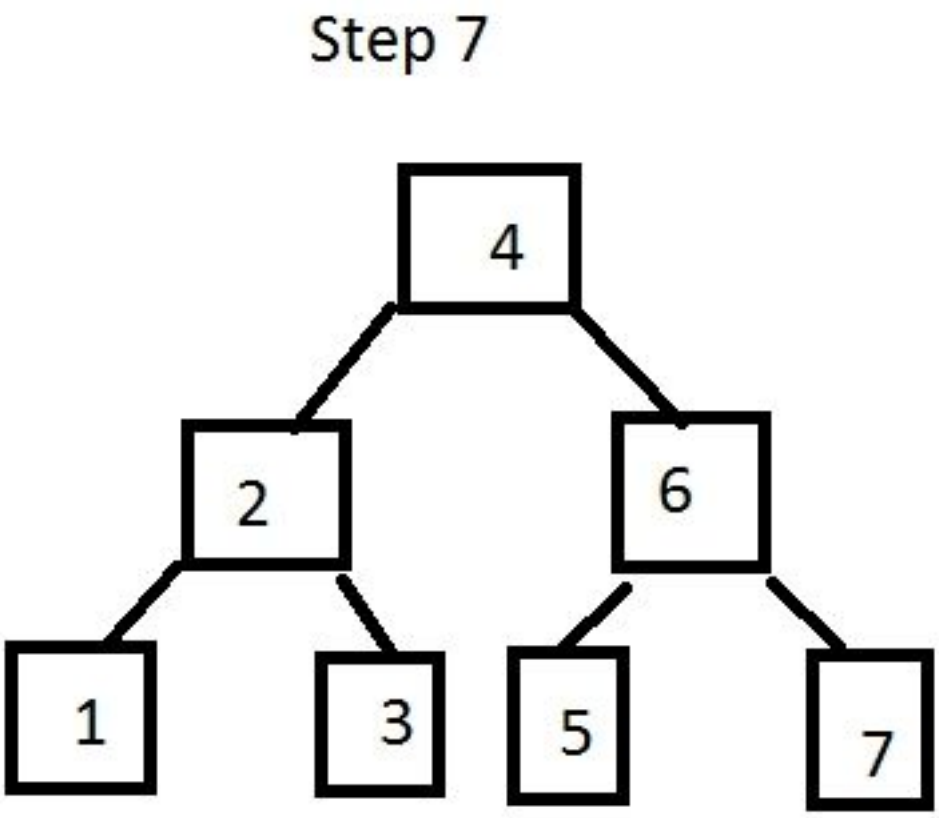


Step 4




Step 5
2
h: 2
1
h: 0
4
h: 1
3
h: 0
5
h: 0


Step 6
4
h: 2
2
h: 1
5
h: 1
1
h: 0
3
h: 0
6
h: 0



Complete the implementation of the AVLtree abstraction by writing the methods to perform a left and right rotation. Both these methods should call setHeight on both the new interior node that has been changed and the new root. Other methods that are similar to those of the Binary Search Tree have been omitted:

```
struct AVLnode * _rotateLeft (struct AVLnode * current) {
      struct AVLNode *newTop;

      newTop = cur->right;
      cur->right = newTop->left;
      newTop->left = cur;
      /* reset the heights for all nodes that have changed heights*/
      /* Note that subtrees under it are all ok...because we've worked
      from the bottom up */
      _setHeight(cur);
      _setHeight(newTop);
      return newTop;
}
```

```
struct AVLnode * _rotateRight (struct AVLnode * current) {
      struct AVLnode * newTop = current->left;
      current->left = newTop->right;
      newTop->right = current;
      _setHeight(current);
      _setHeight(newTop);
      return newTop;
}
```

Finally, let's write the remove function for the AVL Tree. It is very similar to the **remove()** for a BST, however, you must be sure to balance when appropriate. We've provide the remove function, you must finish the implementation of the removeHelper. Assume you have access to removeLeftMost and LeftMost which we have already written for the BST.

```
void removeAVLTree(struct AVLTree *tree, TYPE val) {
   if (containsAVLTree(tree, val)) {
      tree->root = _removeNode(tree->root, val);
      tree->cnt--;
   }
}

TYPE _leftMost(struct AVLNode *cur) {
   while(cur->left != 0) {
      cur = cur->left;
   }
    return cur->val;
}


struct AVLNode *_removeLeftmost(struct AVLNode *cur) {
struct AVLNode *temp;

   if(cur->left != 0)
   {
      cur->left = _removeLeftmost(cur->left);
      return _balance(cur);
   }

   temp = cur->rght;
   free(cur);
   return temp;
}
```

```
struct AVLNode *_removeNode(struct AVLNode *cur, TYPE val) {

        struct AVLNode * temp;
        if(EQ(val, cur->val)){
                if(cur->right != 0){
                        cur->val = _leftMost(cur->right);
                        cur->right = _removeLeftMost(cur->right);
                        return balance(cur);
        }

        else if (LT(val, cur->val))
                cur->left = _removeNode(cur->left, val);
        else
                cur->right = _removeNode(cur->right, val);

        return balance(cur);
        }
}
```