13.3-4

Professor Teach need not worry since the loop will never be entered if z is the root since p[root] = nil[T] = black. Assuming z is not the root, we can still prove that nil[T] will not be colored red. There are only two places in the function where any nodes are colored red. The first occurs in case 1, when z's uncle is red. The second occurs in case 3, when z's uncle is black and z is a left child. In both cases, the node that is colored red is p[p[z]]. Given this fact, in order for nil[T] to be colored red it would need to be the grandparent of z. This is the case only when the parent of z is root, which is black. Since this means that p[z] = black, the loop won't be entered. Therefore, there is no risk of nil[T] being set to red.

13.3-5

Since we can assume that the tree the node is being inserted into does not contain any red nodes, there are two possible cases:

1) The tree is empty
2) The tree is non-empty and contains all black nodes
3) The tree is non-empty and it has some red nodes.

In case 1, the tree becomes a non-empty tree with all black nodes as soon as we insert the first node. Since at this point n > 1, all that remains to be proven is that when additional nodes are inserted into a non-empty tree, there will be at least one red node. The next node will be inserted as a red node and its color will only be changed if a RB tree property has been violated. The only two possible violations are

1) z is the root
2) parent of z is red

Since the tree is non-empty, z will not be the root. If all previously existing nodes in the tree are black, the parent of z cannot be red. If there are existing red nodes and the parent of z is red, the fix function will always end up assigning at least one node to be red. Therefore, the properties of the RB tree have not been violated and z will be left in its place as a red node.

13.4-4

Lines: 1, 2, 3, 4, 9, 23

4.1 Recursive Algorithm

```c
int distance (A, B, i, j){
    if(length[A] == 2){
        // A is empty, return the num of chars in B
        return length[B] - 2;
    }
```
```plaintext
if(i == 0 || j == 0){
    // we've hit to beginning of one of the lists
    return abs(j - i);  // return diff
    // these chars need to be inserted / deleted
}
if(A[i] == B[j]){    // lists are teh same from i & j on
    return distance(A, B, i - 1, j - 1);
}
else{            // these correspond to insert, delete, & change respectively
    return(1 + min(distance(i - 1, j), distance(i, j - 1), distance(i - 1, j - 1));
}
}

4.2 Worst case run time

Worst case: A & B are the same length and all corresponding characters are different (A * B aren't empty). This means that 3 distance recursive calls are made per character in the shortest string.

T(n) = 3T(n - 1) + 1 = O(3^n)

4.3 DP pseudo code

String-edit-order(A, B){
    m[0, 0] = 0    // assign left column
    for(i = 1 to length[A])
        m[i, 0] =  i
    // assign bottom row
    for(j = 1 to length[A])
        m[0, j] =  j
    // assign upper left triangle (diagonals)
    // assign remaining values

    for( i = 1 to length[A])
        for(j = 1 to length[B])
            // chars are the same
            if(A[i] == B[j])
                // do nothing -- assign value to be same as left, lower
                m[i,j] = m[i-1, j-1]
                s[i, j] = n
```

else if(insertion needed)
    m[i, j] = 1 + m[i - 1, j]
    s[i, j] = i
else if(deletion needed)
    m[i, j] = 1 + m[i, j - 1]
    s[i, j] = d
else if(change needed)
    m[i, j] = 1 + m[i - 1, j - 1]
    s[i, j] = c

4.4 Worst case run time

The runtime is the time to fill the matrix. This is I * J where I = length[A] and J = length[B].

4.5 Pseudo code

Print-optimal-edits(s, i, j){
    // print the s elements corresponding to the minimum path in matrix m

    if i == j
        print s[i, j]
        Print-optimal-edits(s, i, s[i, j])
        Print-optimal-edits(s, s[i, j], j)