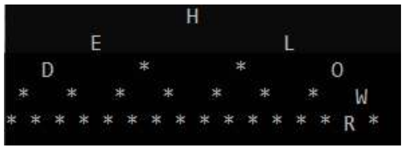
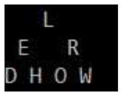
Lab7
Prof. Keith O'hara
Henry Chang

In this lab, I finished multiple (pre-, post-, in- and level-order) distinct traversals with similar recursion methods. This helps us see how binary trees(BT) are tied closely with its recursive definition. The recursive nature of BT, once understood, is easy to use for maintaining the data structure. This recursive feature also makes binary tree associated methods, such as getting the depth of a node, straightforward to implement with recursion.

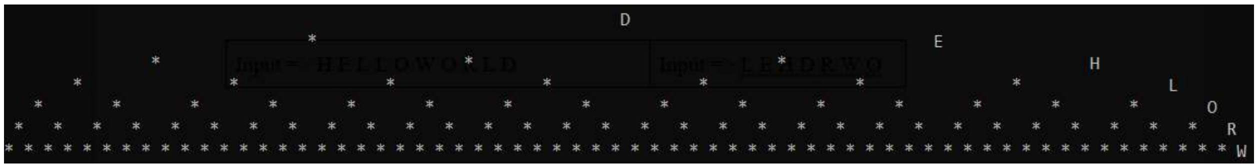
We also compare the binary trees implemented with different rules. Specifically, are objects are the prototypical Binary Search Tree(BST) and the more complex Left-Leaning Red-Black Binary Search Trees (LL-RB BST):

In my algorithm, the insertion method for BST is built with a recursion of finding the inserting positions by assigning the key/value to the root if it is null, inserting to the right subtree with the right child as the root if the input is bigger than current root's key, inserting to the left subtree with the left child as the root if smaller than or equal to the current root's key. My insert() algorithm blindly insert based on the input key and its comparable relation with the keys in the tree. It does not maintain the level structure of the BST. Thus, we see that the tree height is *highly affected* by the order of the input. If N distinct keys are inserted, the worst case could be a tree with N levels and the best case could be lgN levels. As the result shows:

Input => D E H L O R W	H E L L O W O R L D	L E H D R W O
See picture below		

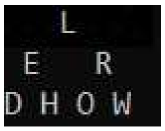
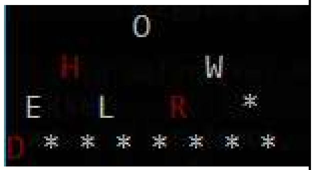
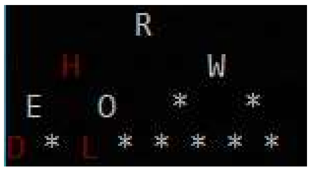
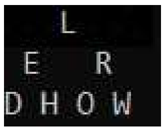
```
               H
       E               L
   D       *       *       O
 *   *   *   *   *   *   *   W
* * * * * * * * * * * * * * R *
```

```
   L
 E   R
D H O W
```



```
                                                               D
                               *                                                               E
               *                               *                               *                               H
       *               *               *               *               *               *               *               L
   *       *       *       *       *       *       *       *       *       *       *       *       *       *       *       O
 *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   R
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * W
```

On the other hand, the insert method for the LL-RB BST includes maintenance for the structure of the tree with rotations, so the tree is perfectly balanced in a 2-3 binary tree representation. Translated back to a prototypical binary tree, for N distinct keys inserted, that would be a tree with level between lgN or 2lgN. Sometimes it would be more than lgN because a three node in a 2-3 binary tree would stretch on more level in the ordinary binary tree representation. If a branch with height h is full of 3 nodes, its height in BST would be 2h. Regardless of the order of the input, the tree height will always be between lgN or 2lgN. Four examples are demonstrated:

Input =>D E H L O R W	H E L L O W O R L D	L E H D R W O	L E R W O H D
 L E R D H O W	 O H W E L R * D * * * * * * *	 R H W E O * * D * L * * * * *	 L E R D H O W

Comparing the two types of BSTs, we observe that LL-RB BST promises a smaller worst case height 2lgN than the ordinary BST N. The searching through LL-RB BST would be faster than BST, since the worst case searches would be proportional to the tree height. As the most time during the insert method is spent on searching the position in the tree to insert the key, we see that insert is also faster for LL-RB BST than BST as N grows large.