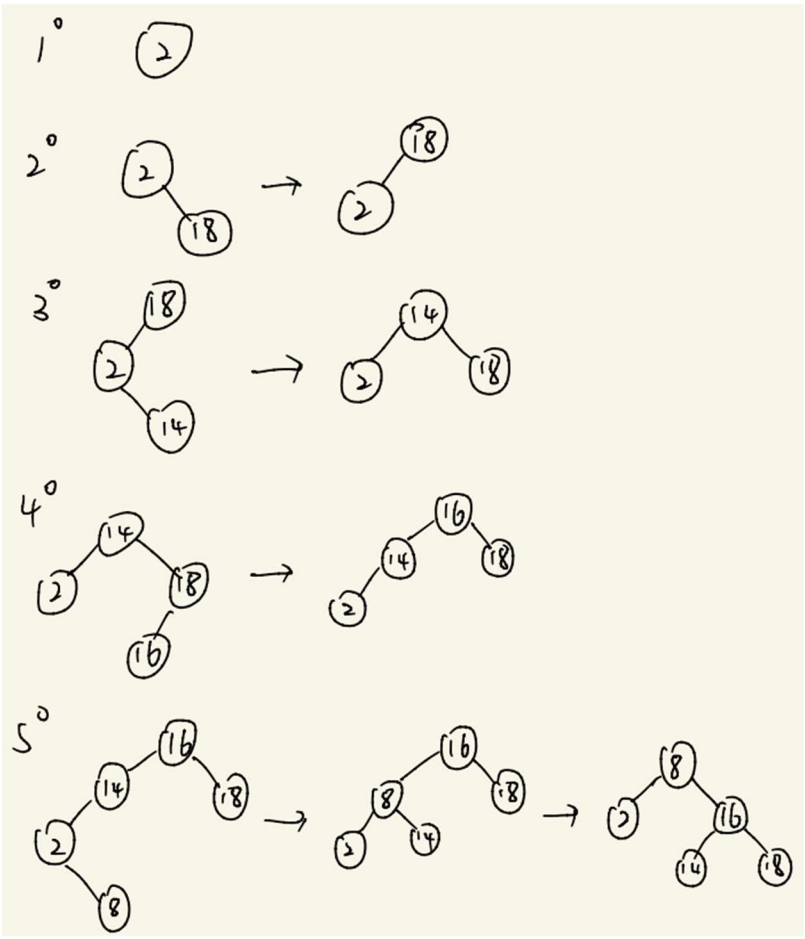
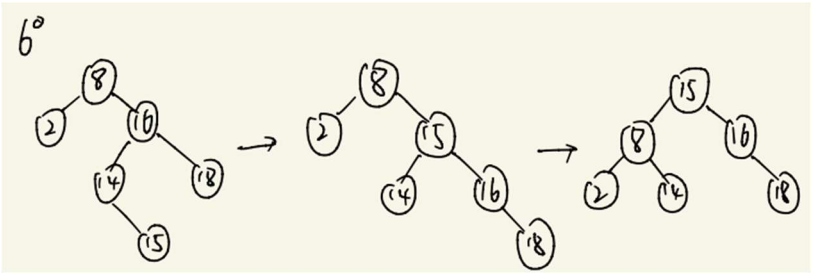
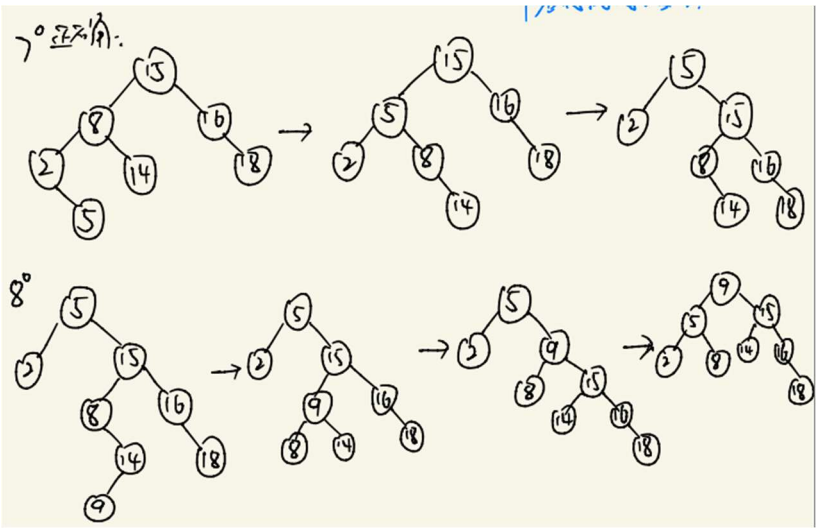
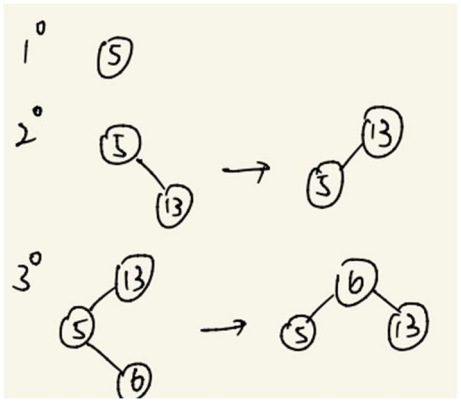
1.插入序列为 [2, 18, 14, 16, 8, 15, 5, 9]

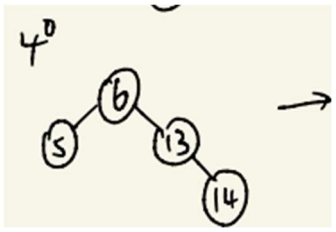


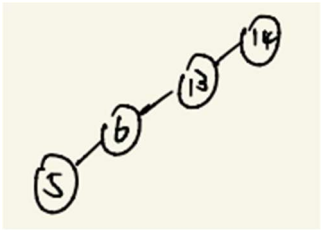


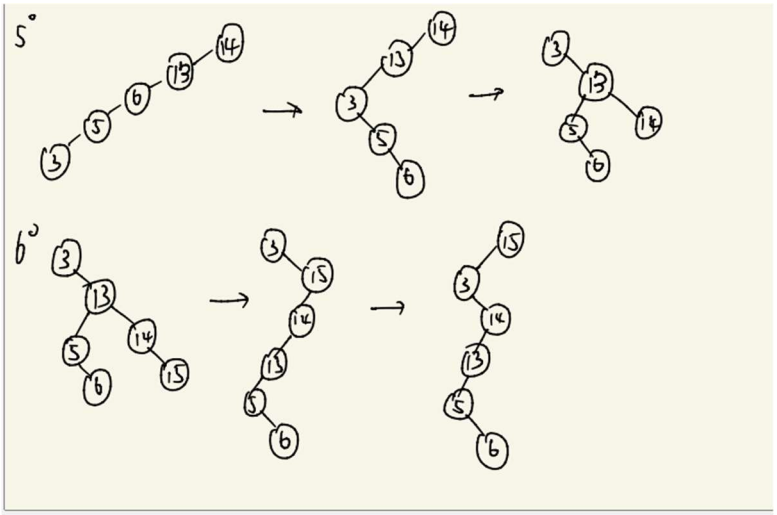


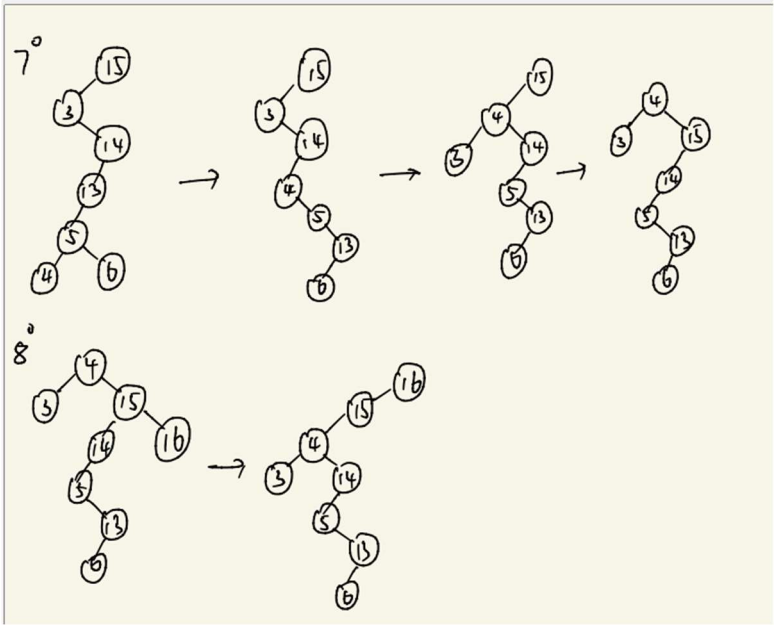
2.插入序列为 [5, 13, 6, 14, 3, 15, 4, 16]









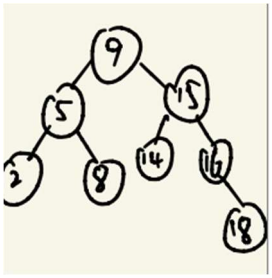


1.你怎么理解 splay 的均摊时间复杂度?

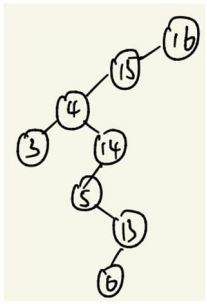
根据书上所写：“就二叉搜索树而言，数据局部性具体表现为：1、刚刚访问过的节点、极有可能在不久之后再次被访问到；2、将被访问的下一节点、极有可能就处于不久之前被访问过的某个节点的附近”。因此根据这种二叉搜索树的数据局部性，每一次查询、插入、删除操作都将目标节点或者是附近节点提升到了根节点处、因此查询、插入、删除的时间消耗被均摊到了每一次的节点提升动作所花费的复杂度中，提升节点之后、每次查询、插入、删除所消耗的时间便根据数据局部性原理，理论上减少了。因此我理解这就是 splay 的均摊时间复杂度的一个方面。另一方面，通过查找对应的知乎文章，我也认识到也可以利用对二叉树的势能分析的方法对其中每次操作和每次提升作出严谨分析。

2.对于实践环节的两个插入序列，它们最后产生的树的形态有什么区别？从插入序列来看，原因是什么?

第一个插入序列为 [2, 18, 14, 16, 8, 15, 5, 9]



第二个插入序列为 [5, 13, 6, 14, 3, 15, 4, 16]



第一个插入序列最后产生的树的形态是趋于平衡且高度接近 log2(n)的；第二个插入序列最后产生的树的形态是不平衡且高度接近 n 的。从插入序列看，我认为原因是第二个插入序列中多次插入了极大值和极小值。在第一个插入序列中，最大值 18 和最小值 2 已经在前两次插入中完成插入，后续数字均在这个范围之间，导致不会出现根节点左子树或者右子树为空的情况，能让树的高度更接近 log2(n)且让树更平衡。然而，在序列二中，插入 13，14，15，16 时，这些数字均是当前已经插入的数字的极大值，这就会导致这几次操作完成之后，根节点的右子树为空，而插入的 5，6，3，4 又是当前已经插入的数字中的极小值，这就会导致这几次操作完成之后，根节点的左子树中元素个数远小于右子树的元素个数。因此，随着类似的操作数增加，树的高度更接近 n，树更不平衡。