Leftist Tree（Leftist Heap）

左偏树（左倾堆）

描述：

左偏树是一种接近于堆的二叉树，它的根节点总是树中的最小值或最大值。两个小根堆（或大根堆）无法快速合并，而左偏树可以支持快速合并。本文只考虑根节点为最小值的左偏树。

左偏树的主要操作包括（1）合并两个左偏树；（2）在左偏树上插入新节点；（3）查找最小值（左偏树中根节点即为最小值）；（4）删除最小值（根节点）。其中（2）-（4）操作依赖于（1），因此合并操作是左偏树的核心操作。

定义左偏树中一个节点的距离d，其值为该节点递归的向右下一直到叶子节点的边的数量。左偏树具有以下性质：

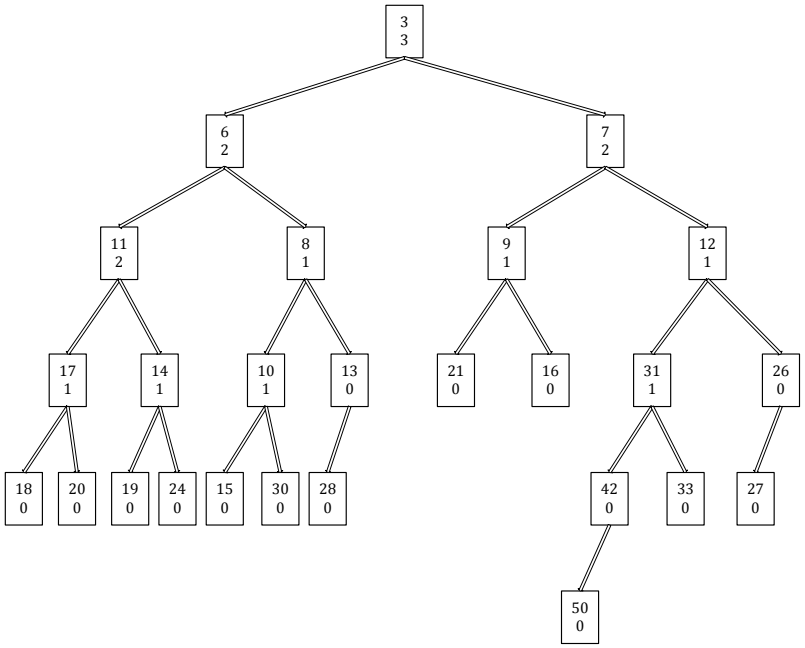
(1) 父节点键值小于等于其左右孩子节点的键值，$father \leq father.left\_child$ 且 $father \leq father.right\_child$；

(2) 父节点的左孩子节点的距离大于等于右孩子节点的距离，$father.left\_child.d \geq father.right\_child.d$；

(3) 父节点的距离等于其右孩子节点的距离加 1，$father.d = father.right\_child.d + 1$；

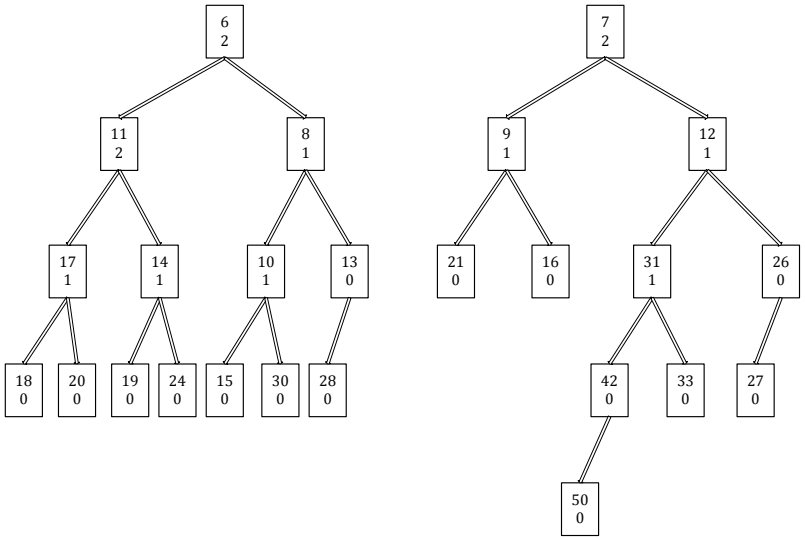
(4) 具有 N 个节点的左偏树的根节点的距离小于等于 $log_2(N+1)-1$，$root.d \leq log_2(N+1)-1$；

如下图：

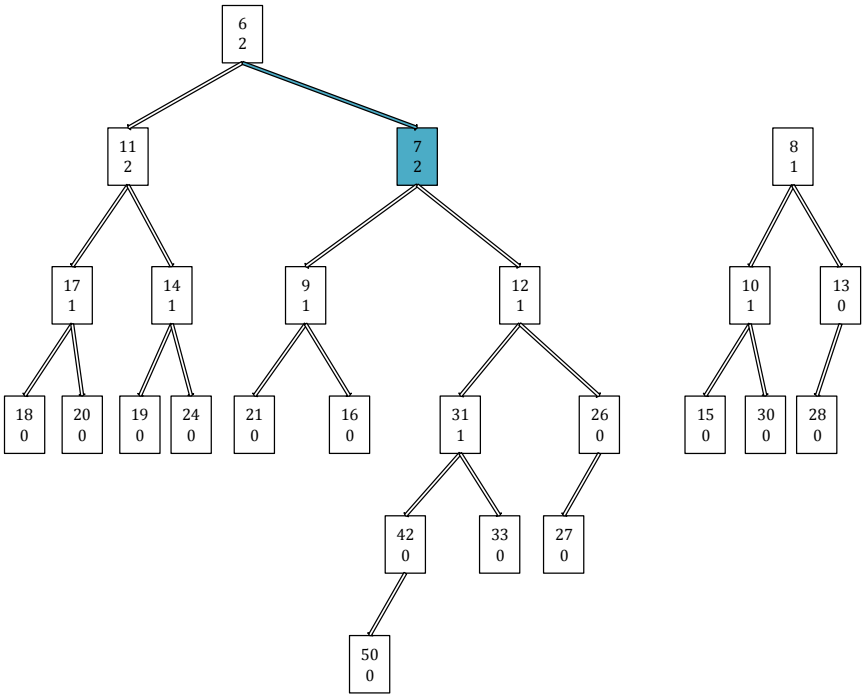


每个节点中上面的数字代表节点的下标号，下面的数字代表该节点的距离。

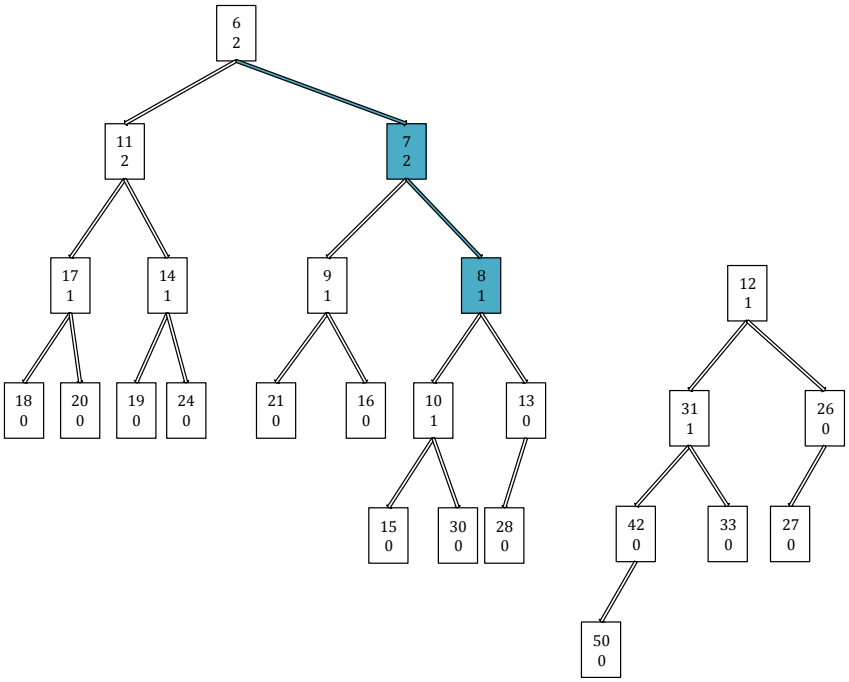
合并下图中的两个左偏树：



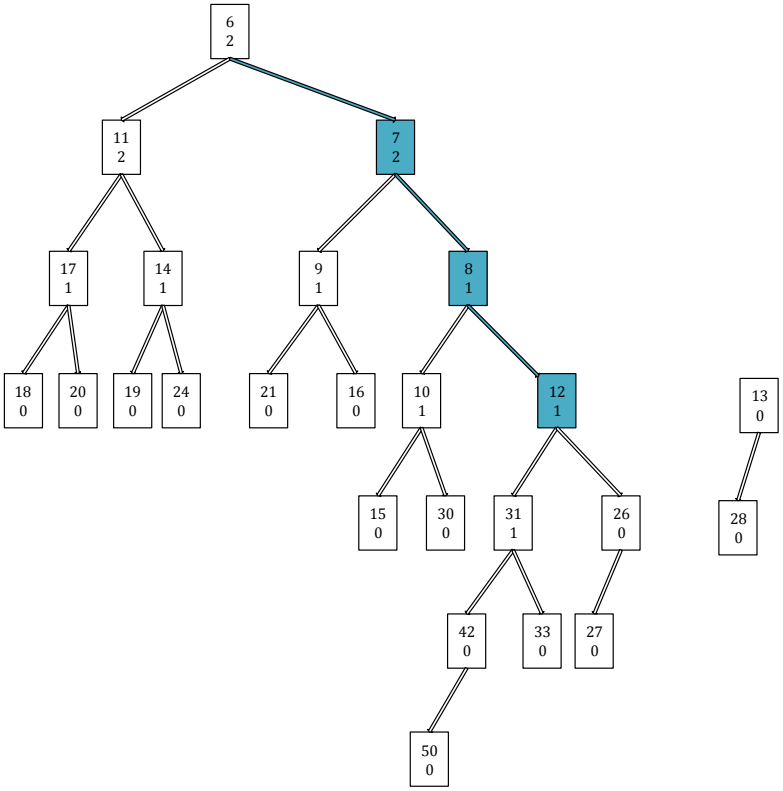
(1) 比较两树根节点的值$6 < 7$，节点 7 沿着节点 6 向右下寻找第 1 个满足$7 < x$的节点x，替换x作为节点 6 的新右孩子节点。该节点为节点 8（$7 < 8$），节点 6 的原右孩子节点 8 暂时脱离；



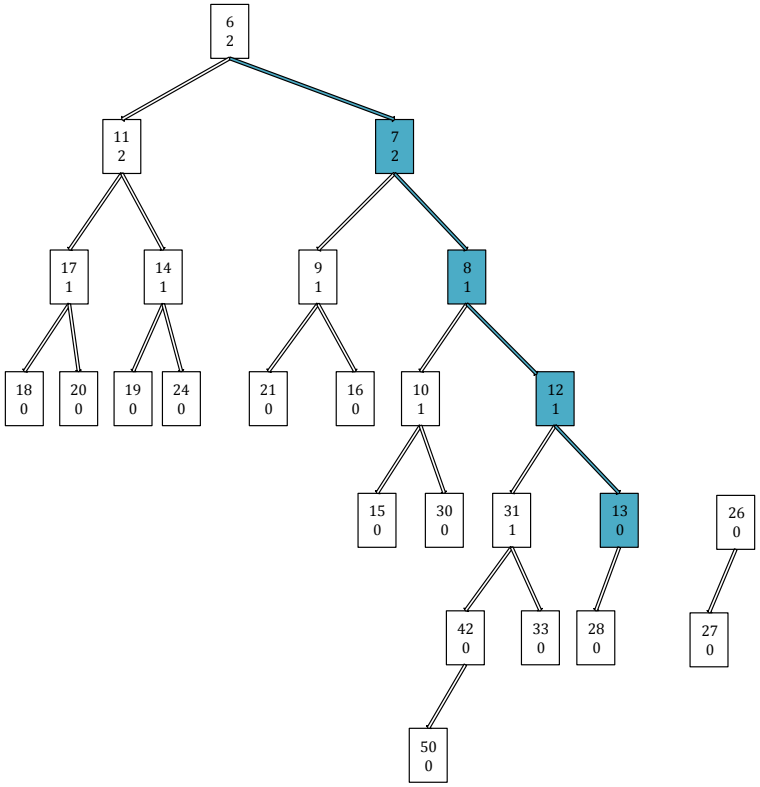
(2) 节点 8 沿着节点 7 向右下寻找第 1 个满足$8 < x$的节点x，替换x作为节点 7 的新右孩子节点。该节点为节点 12（$8 < 12$），节点 7 的原右孩子节点 12 暂时脱离；



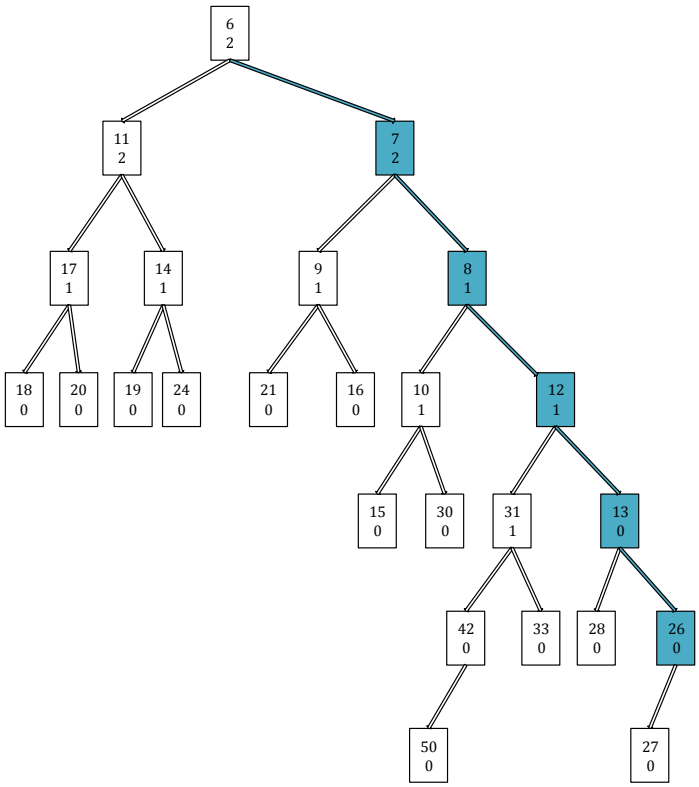
(3) 节点 12 沿着节点 8 向右下寻找第 1 个满足$12 < x$的节点x，替换x作为节点 8 的新右孩子节点。该节点为节点 13（$12 < 13$），节点 8 的原右孩子节点 13 暂时脱离；



(4) 节点 13 沿着节点 12 向右下寻找第 1 个满足$13 < x$的节点x，替换x作为节点 12 的新右孩子节点。该节点为节点 26（$13 < 26$），节点 12 的原右孩子节点 26 暂时脱离；



(5) 节点 26 沿着节点 13 向右下寻找第 1 个满足$26 < x$的节点x，节点 13 没有右孩子节点，因此节点 26 直接成为节点 13 的右孩子节点，不再需要替换，合并操作结束；

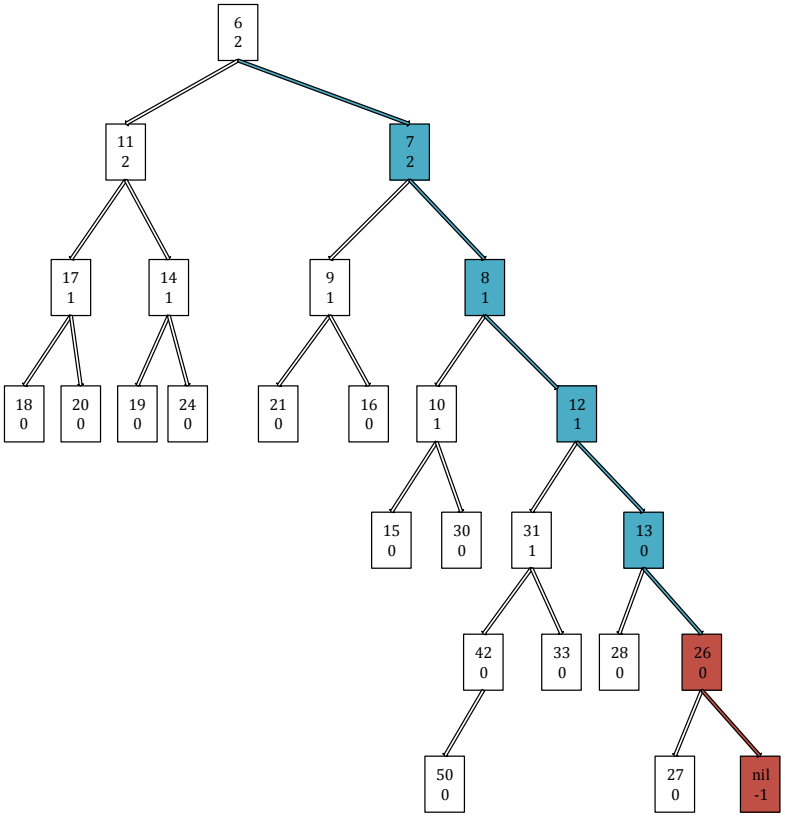


向右下插入节点的操作会影响到左偏树的平衡性，右子树变得越来越庞大。而且节点的距离也没有更新。实际上每一步合并操作后还需要检查左右子树的距离属性：（1）对于$left_child.d < right_child.d$的情况，交换左右子树；（2）对于$root.d \neq right_child.d + 1$的

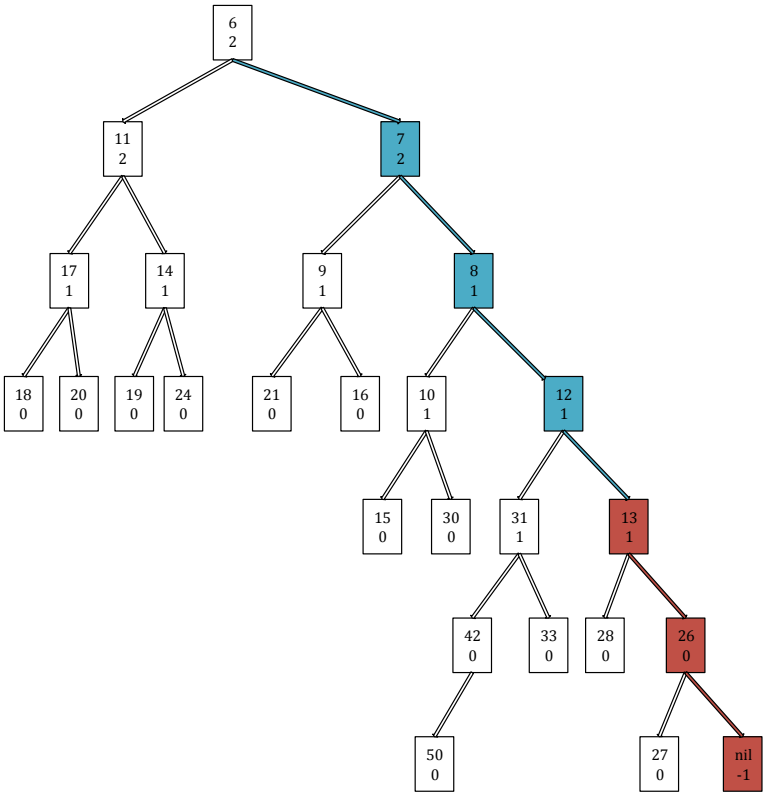
情况，更新$root.d$。

节点距离的更新必须从叶子节点开始向上进行，对于之前步骤中修改过的节点，重新遍历计算其距离（其中空节点的距离认为-1）：

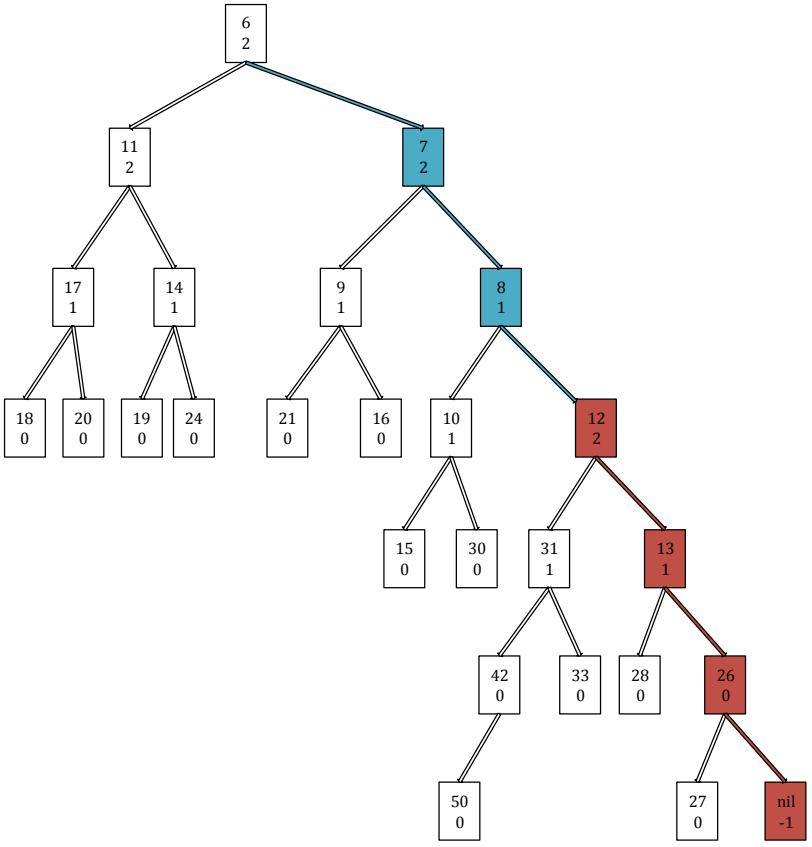
(1) 对于节点 26，$node_{26}.left\_child.d = node_{27}.d = 0 \geq node_{26}.right\_child.d = node_{nil}.d = -1$，不需要交换左右子树，$node_{26}.d = node_{26}.right\_child.d + 1 = node_{nil}.d + 1 = -1 + 1 = 0$；



(2) 对于节点 13，$node_{13}.left\_child.d = node_{28}.d = 0 \geq node_{13}.right\_child.d = node_{26}.d = 0$，不需要交换左右子树，$node_{13}.d = node_{13}.right\_child.d + 1 = node_{26}.d + 1 = 0 + 1 = 1$；

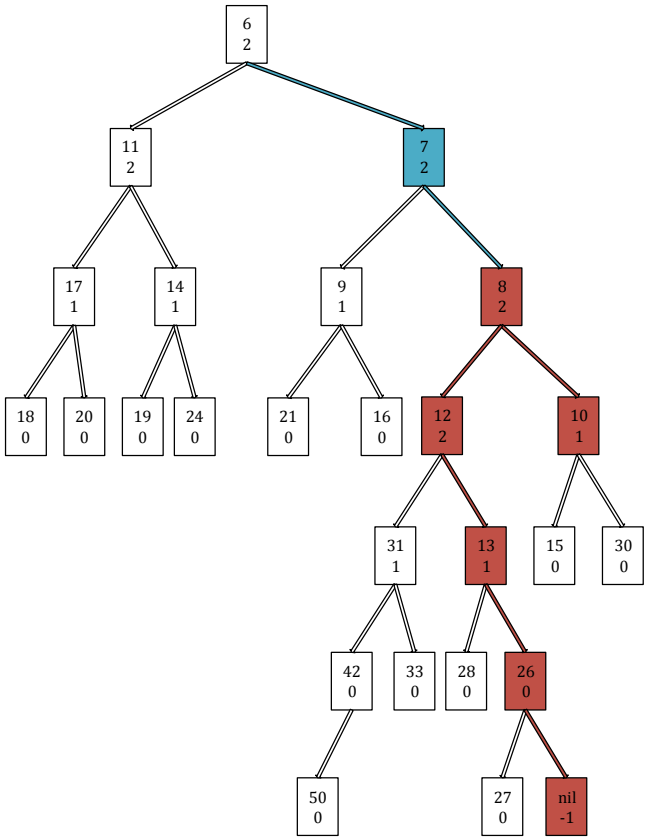


(3) 对于节点 12，$node_{12}.left\_child.d = node_{31}.d = 1 \geq node_{13}.right\_child.d = node_{26}.d = 1$，不需要交换左右子树，$node_{12}.d = node_{12}.right\_child.d + 1 = node_{13}.d + 1 = 1 + 1 = 2$；

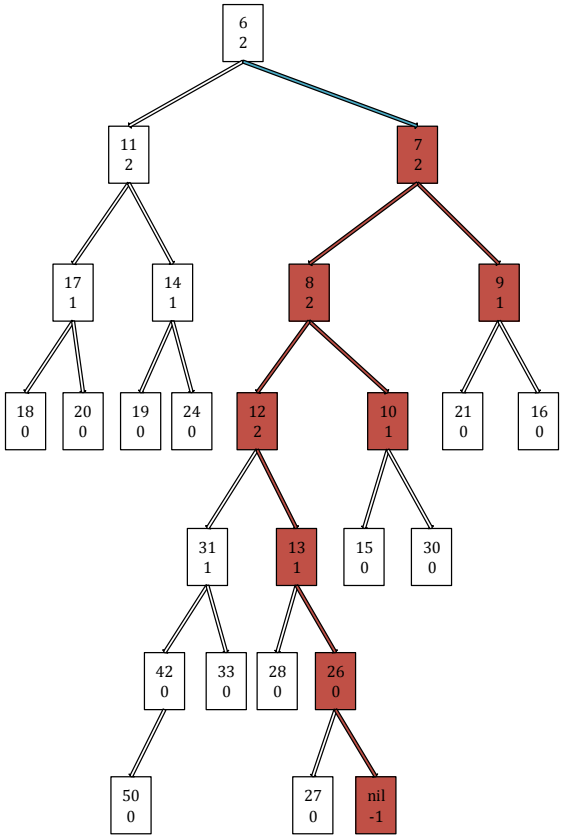


(4) 对于节点 8，$node_8.left\_child.d = node_{10}.d = 1 < node_8.right\_child.d =$

$node_{12}.d = 2$，需要交换子树 10 和子树 12，$node_8.d = node_8.right\_child.d + 1 = node_{10}.d + 1 = 1 + 1 = 2$；



(5) 对于节点 7，$node_7.left\_child.d = node_9.d = 1 < node_7.right\_child.d = node_8.d = 2$，需要交换子树 9 和子树 8，$node_7.d = node_7.right\_child.d + 1 = node_9.d + 1 = 1 + 1 = 2$；



(6) 对于节点 6，$node_6.left_{child}.d = node_{11}.d = 2 \geq node_6.right\_child.d = node_7.d = 2$，不需要交换左右子树，$node_6.d = node_6.right\_child.d + 1 = node_7.d + 1 = 2 + 1 = 3$；

实际编码时可以通过递归的方式将合并子树和更新距离两个操作放在同一个函数中，合并函数 Merge 返回合并后左偏树的根节点的距离d，在 Merge 中调用左右孩子的合并操作，获取左右孩子的距离，然后再决定是否交换左右子树，并更新父节点的距离。本文的将两个步骤分开是为了方便理解算法。

左偏树插入新节点的操作，可以看作左偏树与一个只有根节点的左偏树的合并操作；删除根节点的操作，可以看作删除根节点后，左右子树的合并操作。

左偏树的合并操作、插入节点操作、删除根节点操作的时间复杂度都为$O(log_2 N)$。