6. Complete the `Red_Black_Tree` class by coding the missing functions for removal. The functions `erase` and `find_largest_child` are adapted from the corresponding functions of the `Binary_Search_Tree` class. These adaptations are similar to those done for the AVL tree. A data field `fixup_required` performs the role analogous to the `decrease` data field in the AVL tree. It is set when a black node is removed. Upon return from a function that can remove a node, this variable is tested. If the removal is from the right, then a new function `fixup_right` is called. If the removal is from the left, then a new function `fixup_left` is called.

The function `fixup_right` is called with a reference to the local root of the subtree whose right sub-tree’s black height is one less than the left sub-tree. This local root is designated `P` in the figures that illustrate the various cases that must be considered. The right sub-tree is indicated by `X` in a dotted circle and with a dotted line. This node `X` represents a back leaf that has been deleted or it represents the root of the sub-tree whose black height has been reduced as shown in Figures 11.68, 11.69, 11.70, and 11.71.

If the node `X` is red, then the black height can be easily be restored by setting it black. Otherwise, the `fixup_right` function must consider four cases as, as follows:

- **Case 1:** The sibling of `X` (designated `S` in Figure 11.68(a)) is red. The parent (`P`) must be black, and if `S` has children, then they must be two black nodes (L, and R). We change the color of `P` to red, and `S` to black (Figure 11.68(b)) and then rotate right about `P` (Figure 11.68(c)). Now we have a case where `X` has a black sibling (R). Recall that null trees are considered black. This transforms the problem into one of the other cases where `R` is now the sibling of `X`.

**FIGURE 11.68**
Red-Black Removal Case 1

**FIGURE 11.69**
Red-Black Removal Case 2
- **Case 2**: The sibling of $X$ (designated $S$ in Figure 11.69(a)) is black, and it is either a leaf or it has two black children. Note that we do not care what color $P$ has, so we show it in a white circle. We change the color of $S$ to red (Figure 11.69(b)). This reduces the black height of the sub-tree whose root is $S$ so it is now equal to the black height of the tree whose root is $X$. The overall black height of $P$ has been reduced by one so we repeat the process at the next level ($P$’s parent). Note that we may now have a red parent ($P$) with a red child ($S$), but this will be fixed at the next level.

- **Case 3**: The sibling of $X$ (designated $S$ in Figure 11.70(a)) is black and it has a red right child ($R$). $S$ may also have a left child, but we do not care what its color is, so it is not shown. We change the color of $S$ to red and the color of $R$ to black. (Figure 11.70(b)). Then we rotate left about $S$ (Figure 11.70(c)). This transforms the problem into Case 4.

**FIGURE 11.70**
Red-Black Removal Case 3

Note that before making this transformation, $S$ was the root of a valid red-black tree. Therefore if $S$ had a black left child, then $R$ must have two back children. After performing the rotate, $R$ is still the root of a valid red-black sub-tree. On the other hand, if $S$ had a red left child, then after the rotate $R$’s left child ($S$) is red and has a red left child. This will be fixed when we consider case 4. However the black heights of $R$’s sub-trees remain balanced.

- **Case 4**: The sibling of $X$ (designated $S$ in Figure 11.71(a)) is black and it has a red left child ($L$). $S$ is the root of a red-black sub-tree whose black height is balanced and is one greater than the black height of $X$. We change the color of $L$ to black. (If we got here from case 3, the red-red problem is now fixed.) This increases the black height of $L$. We also change $S$ to be the same color as $P$, and then change the color of $P$ to black. (Figure 11.71(b)). By rotating right about $P$, we restore the black balance (Figure 11.77(c)).

**FIGURE 11.71**
Red-Black Removal Case 4