ADSA Assignment 2

[redacted]

OS: Ubuntu 20.04 running inside WSL on Windows 10.
Compiler version: 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)

1. Problem Description

We have to solve the classic 2-Sum problem in a dynamic sense. Instead of a predefined array/list of numbers from which we have to find two numbers whose sum is equal to a target value, we have to use a dynamic data structure in which any element can be inserted or deleted at any time. In this data structure, we have to find the possible targets by adding any two numbers present in the data structure at any possible time.

2. Solution Description

2.1. Data Structure Intuition

We are using Red-Black trees (hereafter referred to as RB trees) as our solution's data structure. For choosing a proper data structure for this problem, We required a data structure that did not use more than ***O(n)*** space and have the lowest Search, Insert, and Delete time complexity. Though Hash Tables provides ***O(1)*** time complexity for Insert and Delete, in a dynamic structure, the time complexity for search will get worse as the number of data nodes increases.

We use RB trees as they are always balanced and won't have the drawback of adding nodes on only one side as in Binary Tree. Secondly, it is always guaranteed to have an ***O(log n)*** time complexity for Insert, Search, and Delete operations, even in the worst case.

2.2. Functions Description

In the main function, we have three possible cases: Insert, Delete, and Query.

2.2.1. Insert

This function is used to insert elements into the RB tree. This function first finds the appropriate location for inserting the given key. This is done using a binary search on the tree to find the proper position. Once the position is located for inserting the key, we insert the key and color it red. We now need to make sure our tree follows the RB property.

1. If ***N*** is the root node, make it black.
2. If ***N***'s parent ***P*** is black, then do nothing as no property is violated.
3. If ***N***'s parent ***P*** is red and ***N***'s uncle ***U*** is also red, then both the parent and uncle are repainted black, and grandparent ***G*** is painted red. Since the grandparent may now violate RB tree property, we rerun the fix until the violation reaches the root or is fixed.
4. If parent ***P*** is red and ***N***'s uncle ***U*** is black, we check if the node ***N*** is left or right child. If the node is inside the subtree (i.e., if ***N*** is the left child of the right child of ***G*** or the right child of the left child of ***G***), we perform a left rotation on ***P*** that switches the roles of node ***N*** and parent ***P***.
   If the node ***N*** is outside of the subtree under grandparent ***G***(left of the left child or right of the right child), we do a right rotation, putting ***P*** in place of ***G*** and making ***P*** the parent of ***N***.

The pseudocode for insert is given below:
```
Insert(T,z):
     y = T.null
     x = T.root
     while x!= T.null
          y = x
          if z.key < x.key
               x = x.left
          else x = x.right
     z.p = y
     if y == T.null
          T.root = z
     elseif z.key < y.key
          y.left = z
     else y.right = z
     z.left = T.null
     z.right  = T.null
     z.color = RED
     InsertFix(T,z)
```

Here InsertFix() fixes the RB tree after inserting the key according to the conditions given above.

2.2.2. Delete

This function is used to delete the node from the tree. Firstly, we find the node to delete as in BST. If the element does not exist in the tree, we do not make any changes to the tree. Otherwise, we delete the node following BST operations. Then a few cases arise:

1. If the deleted node was red, do nothing.
2. If the deleted node ***N*** was black then,
    a. If the child ***C*** of ***N*** was red, color it black.
    b. If ***C*** is black, we can't change it from red to black to preserve black depth. Instead, we'll color it double-black.
    c. What we do about the double black depends on the colors of the neighboring nodes (after *v* is removed).
    d. If double-black reaches the root, then it can be made a single black, removing one from the black depth of every single leaf node.
    e. If ***C*** is black and the sibling of ***C*** is red, rotate the tree to an equivalent position, but so that ***C***'s sibling is now black. Then we apply one of the previous rules.

The pseudocode for deletion is given below:

```
Delete(T,z):
     y = z
     y_original_color = y_color
     if z.left == T.null
          x = z.right
          Transplant(T,z,z.right)
     elseif z.right == T.null
          x = z.left
          Transplant(T,z,z.left)
     else y = Minimum (z.right)
          y-original_color = y.color
          x = y.right
          if y.p==z
               x.p = y
          else Transplant(T,y,y.right)
               y.right = z.right
               y.right.p = y
          Transplant(T,z,y)
          y.left = z.left
          y.left.p = y
          y.color = z.color
     if y_original_colo == BLACK
          DeleteFix(T,x)
```

Here DeleteFix() fixes the RB tree after deleting the node according to the above given conditions.

2.2.3. Query

This function is used to query the number of targets possible in a given range by adding any two distinct elements present in the RB tree.

For this, we firstly do a preorder traversal of the tree to find the elements present in the tree. We then iterate over the range of Query to check if that target is possible by adding any two numbers in the RB tree. In each iteration, we iterate through the list of elements present in the tree. We then find the value we need to search in the tree by subtracting the current element from the target value. If the element is present in the tree, we mark the target as possible and move to the next target. If the element is not present, we check for the next element in the tree. The pseudocode is given below:

```
Query(a,b)
     values = TreeTraversal()
     targets_possible = 0
     for target in a to b:
          for x in value:
               tofind = target - x
               if (SearchTree(tofind))
                    targets_possible++
                    break
     print targets_possible
```

To make sure that the elements selected are unique and same element is not counted twice, we create a countkeys which count the number of occurrences of a key if the target to be found is equal to the tofind value. The target_possible is incremented only if the count is more than 1.

2.3. Time Complexity

Let the tree contain n nodes. An RB tree with n nodes has ***O(log n)*** height.

2.3.1. Insert

1. To find a place to insert using binary search operation takes the time to iterate through the height of the tree, i.e., ***O(log n)***
2. To add the node, the time taken is ***O(1)***
3. To fix possible double reds, a rotation is ***O(1)***.
4. In the worst case, the double reds can cascade to the tree's height all the way up to the root. Since this is proportional to the height of the tree, the time taken is ***O(log n)***.

5. Hence, the cost for insertion is ***O(log n)***.

2.3.2. Delete

1. Finding the node to delete and the predecessor is proportional to the height of the tree, i.e., ***O(log n)***
2. Swapping and deletion operation takes ***O(1)***.
3. Each individual property fix takes ***O(1).***
4. In the worst case, the double black may be passed up to the root. Since each rotation takes constant time, this would be proportional to the height of the tree, i.e., ***O(log n)***.

5. Hence the worst-case cost of deletion is ***O(log n)***

2.3.3. Query

1. Finding the elements present in the tree requires traversing the tree. This takes ***O(n)*** time.
2. If the range of targets to be checked is m, and we check all elements for each possible target, the cost is ***O(mn)***.
3. Searching if the value exists in the tree is proportional to the tree's height, i.e., it takes ***O(log n).***

4. Hence the cost for deletion in the worst case is ***O(mn log n).***
   If the number of elements is large in comparison to the range, the cost becomes ***O(nlogn)***
   If the range to search is large in comparison to the number of elements, the cost becomes ***O(m)***.
   In the average case, the cost can be said to be ***O(max(m, nlogn))***.