Problem 248: Seeing Red and Black

Difficulty: Hard

Author: Brett Reynolds, Bethesda, Maryland, United States

Originally Published: Code Quest 2024

Problem Background

Storing data is a critical task in any software application, but just as important is the ability to retrieve data. Data that cannot be accessed might as well not exist in the first place. Along the same lines, being able to retrieve data quickly is important for ensuring that an application is able to perform well and meet the user's needs. As a result, a lot of research and study is done into how to store and retrieve data efficiently.

Binary search trees are among some of the more efficient means by which to store large amounts of information, provided they're structured correctly. These trees consist of a series of "nodes" which each contain up to two child nodes: one left, and one right. Given any parent node, the left child and its descendants contain only values less than that of the parent; the right child and its descendants contain values greater than the parent. As a result, locating any node within the tree is a simple matter of repeatedly comparing your desired value against each node you encounter. So long as the tree remains "balanced," this guarantees an optimal search time for any value within the tree. In contrast, an "unbalanced" tree can be extremely inefficient, so many tree structures use algorithms to ensure they remain balanced even as data is added or removed.

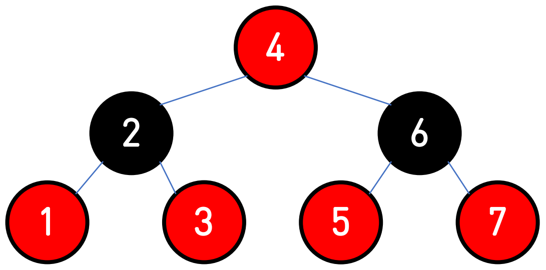
Problem Description

Your team is working at Lockheed Martin to develop and test a new database system which will be responsible for storing large amounts of data, each represented by a unique integer value. After a considerable amount of research, your team has decided to use a "red-black tree" to store this data.

Red-black trees are a form of self-balancing binary search tree. The "self-balancing" part comes from a set of rules that govern the structure of the tree. These rules are:

1. Each node in the tree is either red or black.
2. All NULL nodes (non-existent children) are considered black nodes.
3. A red node may not have red children.
4. Every path from a node to any descendent NULL node must pass through the same number of black nodes.

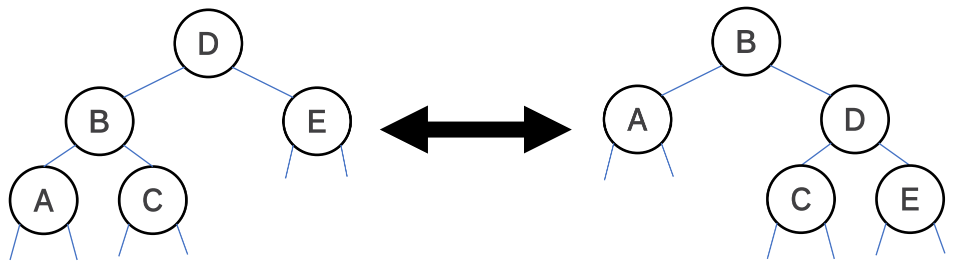
For example, let's look at this red-black tree, containing the numbers 1 through 7:



This coloring follows the red-black rules; any path from 4 to the bottom of the tree touches the same number of black nodes, and no red node has a red child (remember that non-existent nodes - here the "children" of 1, 3, 5, and 7 - are considered black). Now, to add new data to this tree, we'll need to follow a strict procedure to ensure that we're obeying all of the rules at all times. Whenever adding a new node with a value of N:

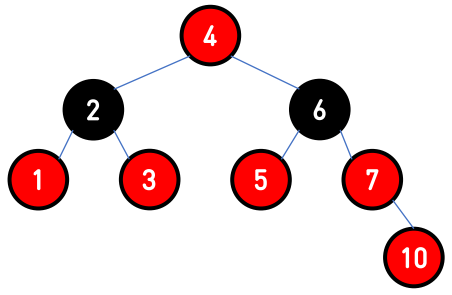
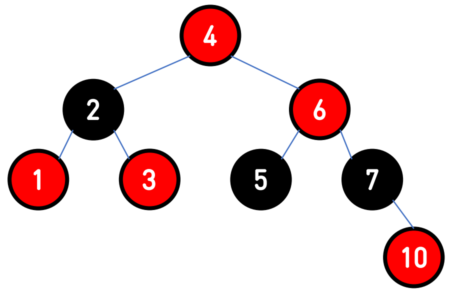
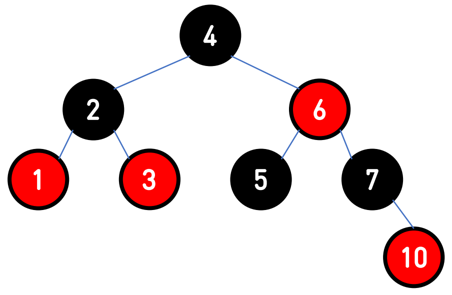
1. Add N to the tree in its expected position as a red node (if there are no other nodes yet, add N as the root of the tree and stop here).
2. Call N your "current node", C, for the following steps. The node being referenced by C may change as you continue.
3. For the following steps, find:
    a. C's parent, which we'll call P.
    b. P's parent (C's grandparent), which we'll call G.
    c. P's sibling (the other child of G), which we'll call U (for "uncle")
4. If P doesn't exist (C is the root) or P is black, stop here.
5. If G doesn't exist (P is the root), make P black and stop here.
6. If U exists and is red, make P and U black, and make G red. Update C to equal G's value, and go back to step 3.
7. Check the path used to get from G to C. If C is an "inner grandchild" - that is, the path is left-right or right-left - rotate P and C (we'll explain this in a minute). Update C to equal P's value, then update P and G accordingly.
8. At this point, C is now an outer grandchild of G (the path to reach C from G is left-left or right-right). Rotate P and G, then make P black and G red.

In most cases, adding a new node to the tree simply requires recoloring a few nodes to make the tree obey the rules. However, in complex cases - those that require steps 7 and 8 above - the structure of the tree needs to be changed as well. This requires "rotating" nodes of the tree. Rotation is a reversible process that "swaps" a child node with its parent. See the diagram below, which demonstrates how to rotate the nodes B and D in both directions:

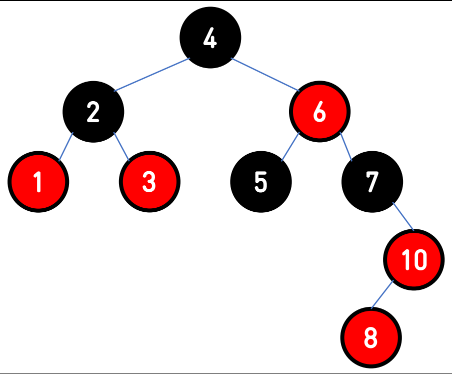
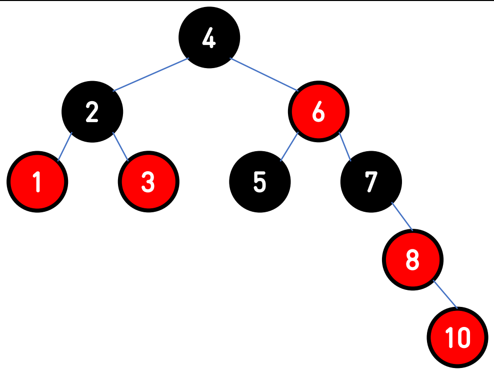
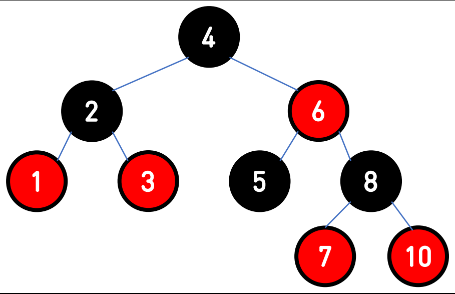


On the left side, D is the parent of B. To rotate them, C (B's child, and the inner grandchild of D) gets disconnected from B. D then takes C's place as a child of B, and C takes B's place as D's child. A and E remain in place, as do any children of A, C, and E.

We'll demonstrate this process by adding the nodes 10 and 8 to the tree from before. Adding 10 only requires recoloring a few nodes:

	Step 1: Add 10 as a red node. Step 2: C is 10. Step 3: P is 7. G is 6. U is 5. Step 4: P (7) is red, so continue. Step 5: G (6) exists, so continue.
	Step 6: P (7) and U (5) are red, so make them black, and make G (6) red. C is now 6; return to step 3. Step 3: P is 4. G does not exist. U is 2. Step 4: P (4) is red, so continue.
	Step 5: G does not exist, so make P (4) black and stop.

Adding 8 will be a little more complicated, as we have to perform a few rotations:

	Step 1: Add 8 as a red node. Step 2: C is 8. Step 3: P is 10. G is 7. U does not exist. Step 4: P (10) is red, so continue. Step 5: G (7) exists, so continue. Step 6: U does not exist, so continue.
	Step 7: C (8) is an inner child, so rotate C (8) and P (10). C is now 10, P is now 8, G remains 7.
	Step 8: Rotate P (8) and G (7) and swap their colors.

For this problem, your team will need to develop a red-black tree structure. Your program will be given instructions to add values to that tree using the procedure above. Your program must also occasionally search for values previously added to the tree and provide information about the state of that node. Your program will never be asked to perform any of the following actions:

- Add a duplicate value to the tree
- Remove a value from the tree
- Search for a value that has not already been added to the tree

Sample Input

The first line of your program's input, received from the standard input channel, will contain a positive integer representing the number of test cases. Each test case will include the following lines:

- A line containing a positive integer, X, representing the number of commands included in this test case

- X lines, each containing a command string as described below, and a positive integer less than 1000, separated by a space.

Command strings may include:

- ADD, indicating that the provided integer must be added to the tree
- FIND, indicating that the program must search the tree for the given integer and print output regarding the state of its node

```
1
15
ADD 4
ADD 2
ADD 6
ADD 1
ADD 3
ADD 5
ADD 7
FIND 1
FIND 2
FIND 4
FIND 7
ADD 10
FIND 10
ADD 8
FIND 8
```

Sample Output

For each test case, your program must print a single line of text for each FIND command, including the following values, separated by spaces:

- The target integer specified by the FIND command;
- Either "RED" or "BLACK", indicating the color of the node containing that integer; and
- An integer representing the depth of that node. The root node has a depth of 1; its children have a depth of 2, and so on.

```
1 RED 3
2 BLACK 2
4 RED 1
7 RED 3
10 RED 4
8 BLACK 3
```