Istanbul Technical University
Faculty of Computer and Informatics
Computer Engineering Department

BLG 335E
Homework 3 Report

Yunus Emre Cakiroglu - 040190019

December 4nd, 2022

Contents

1 Section 1: Description of Code

In this homework, we are expected to create a C++ app that simulates CFS (Completely Fair Scheduler)

1.1 How2run?

It is pretty easy to run this program. You can compile with "`make all`" command then run with "`./homework3 input*.txt`" command. This program **requires** file path to supplied as its argument. Also it is **required** to have "input" in file name like given in example. Program determines output file based on input argument. It replaces input keyword with output.
For ex. "`./homework3 ./input_1.txt`" outputs to
"`./output_1.txt`"

1.2 Red-Black Tree Implementation

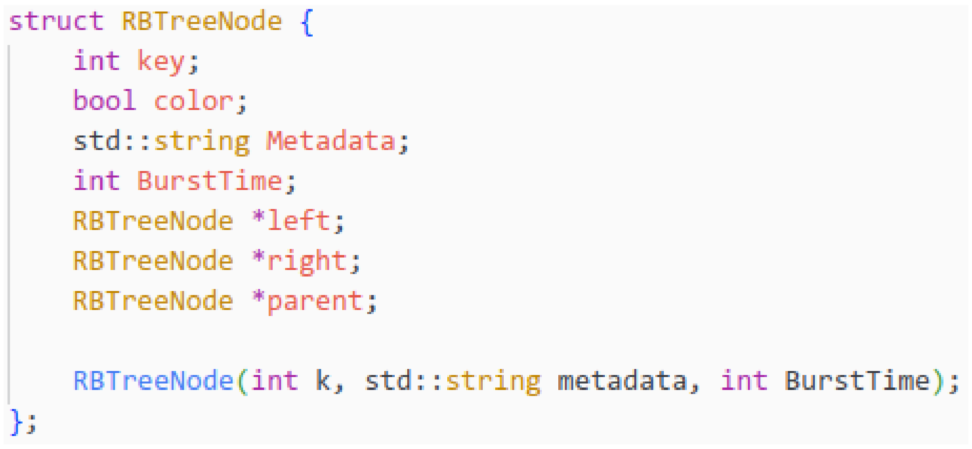
In this homework I decided to implement Red-Black Tree in a different file. I choose this way to seperate both CFS and Red-Black Tree logic. There consist two files. CFS.cpp and RBTree.cpp. CFS is where main logic runs. consist of a CFS class and a main function. RBTree consist of RBTreeNode struct and RBTree class.

Psuedocodes for algorithm taken from most famous algorithms book, CLRS (Introduction to Algorithms).

1.2.1 RBTreeNode

This is node structure of our RBTree. It consist extra datas to fit our needs in CFS implementation. Metadata to keep process label, color to keep track of color (true for red false for black) and BurstTime to get rid of process.

Figure 1: RBTreeNode struct

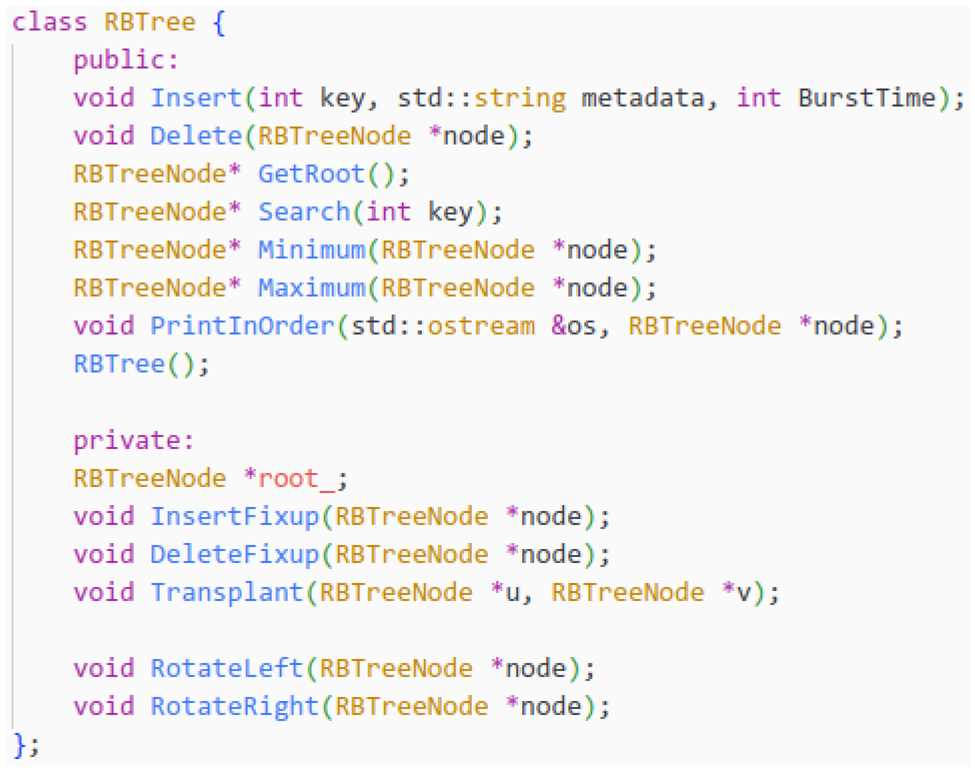


```
struct RBTreeNode {
    int key;
    bool color;
    std::string Metadata;
    int BurstTime;
    RBTreeNode *left;
    RBTreeNode *right;
    RBTreeNode *parent;

    RBTreeNode(int k, std::string metadata, int BurstTime);
};
```

1.2.2 RBTree

This is where real stuff happens. here is class structure of our RBTree data structure implemented in header file.

Figure 2: RBTree class

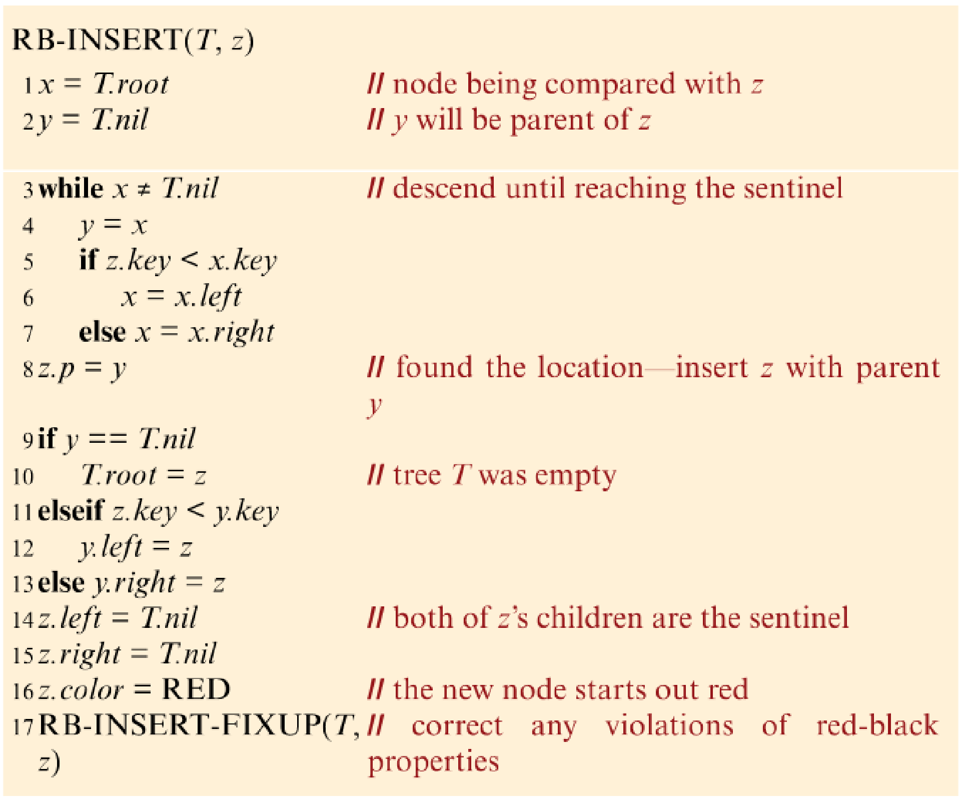


```
class RBTree {
    public:
    void Insert(int key, std::string metadata, int BurstTime);
    void Delete(RBTreeNode *node);
    RBTreeNode* GetRoot();
    RBTreeNode* Search(int key);
    RBTreeNode* Minimum(RBTreeNode *node);
    RBTreeNode* Maximum(RBTreeNode *node);
    void PrintInOrder(std::ostream &os, RBTreeNode *node);
    RBTree();

    private:
    RBTreeNode *root_;
    void InsertFixup(RBTreeNode *node);
    void DeleteFixup(RBTreeNode *node);
    void Transplant(RBTreeNode *u, RBTreeNode *v);

    void RotateLeft(RBTreeNode *node);
    void RotateRight(RBTreeNode *node);
};
```

1.2.2.1 Insert InsertFixup

Insert function inserts a new node with the given key and value into the red-black tree. The new node is inserted into the tree using a standard binary search algorithm, similar to the way nodes are inserted into a binary search tree.

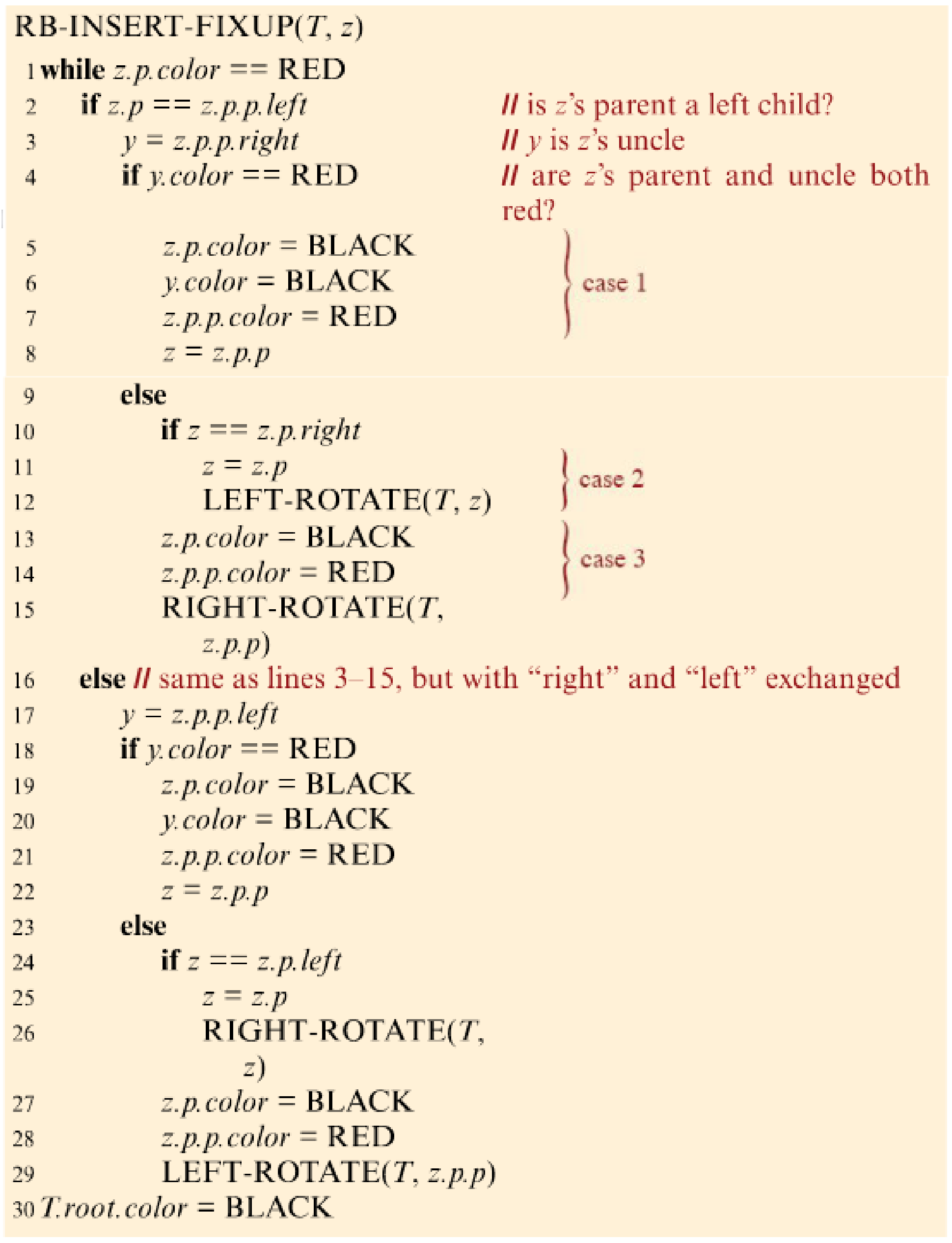
Figure 3: Insert function psuedocode



```
RB-INSERT($T$, $z$)
$x = T.root$                    // node being compared with $z$
$y = T.nil$                     // $y$ will be parent of $z$

while $x \neq T.nil$            // descend until reaching the sentinel
    $y = x$
    if $z.key < x.key$
        $x = x.left$
    else $x = x.right$
$z.p = y$                       // found the location—insert $z$ with parent $y$
if $y == T.nil$
    $T.root = z$                // tree $T$ was empty
elseif $z.key < y.key$
    $y.left = z$
else $y.right = z$
$z.left = T.nil$                // both of $z$'s children are the sentinel
$z.right = T.nil$
$z.color$ = RED                 // the new node starts out red
RB-INSERT-FIXUP($T$, $z$)       // correct any violations of red-black properties
```

After the new node is inserted, the function calls InsertFixup to restore the red-black tree properties and maintain the balance of the tree. The InsertFixup function performs a series of rotations and color changes on the nodes of the tree until the red-black tree properties are satisfied again.

Figure 4: Insertfixup function psuedocode



```
RB-INSERT-FIXUP(T, z)
 1 while z.p.color == RED
 2     if z.p == z.p.p.left                // is z's parent a left child?
 3         y = z.p.p.right                 // y is z's uncle
 4         if y.color == RED               // are z's parent and uncle both
                                           red?
 5             z.p.color = BLACK            ⎫
 6             y.color = BLACK              ⎬ case 1
 7             z.p.p.color = RED            ⎭
 8             z = z.p.p
 9         else
10             if z == z.p.right
11                 z = z.p                  ⎫ case 2
12                 LEFT-ROTATE(T, z)        ⎭
13             z.p.color = BLACK            ⎫
14             z.p.p.color = RED            ⎬ case 3
15             RIGHT-ROTATE(T,              ⎭
                   z.p.p)
16     else // same as lines 3–15, but with "right" and "left" exchanged
17         y = z.p.p.left
18         if y.color == RED
19             z.p.color = BLACK
20             y.color = BLACK
21             z.p.p.color = RED
22             z = z.p.p
23         else
24             if z == z.p.left
25                 z = z.p
26                 RIGHT-ROTATE(T,
                       z)
27             z.p.color = BLACK
28             z.p.p.color = RED
29             LEFT-ROTATE(T, z.p.p)
30 T.root.color = BLACK
```

The InsertFixup function is used to restore the red-black tree properties and maintain the balance of the tree after a new node has been inserted. It does this by performing a series of rotations and color changes on the nodes of the tree.

Here is how the InsertFixup function works:

1. If the new node is the root of the tree, it is colored black. This satisfies the first red-black tree property (every node is either red or black).

2. If the new node's parent is black, no further action is needed. This satisfies the second red-black tree property (the root is black).

3. If the new node's parent is red and its uncle (the sibling of its parent) is also red, both the parent and the uncle are colored black and the grandparent (the parent of the parent)

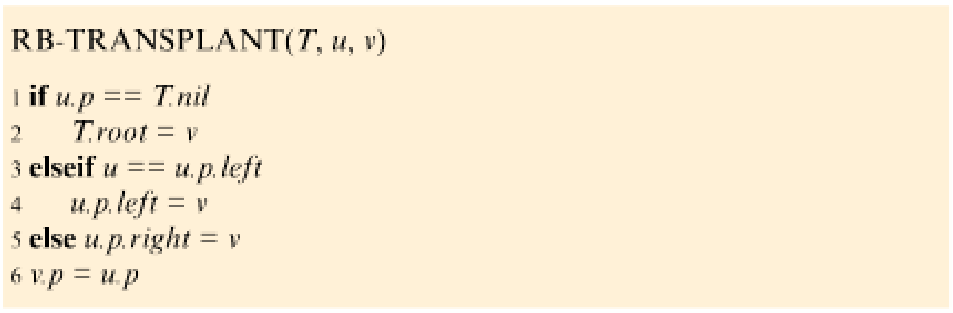
is colored red. This operation is repeated on the grandparent until the red-black tree properties are satisfied.

4. If the new node's parent is red and its uncle is black, the InsertFixup function performs one of four possible rotations on the nodes of the tree, depending on the relative positions of the new node and its parent. These rotations are used to maintain the balance of the tree.

1.2.2.2 Transplant, Delete, DeleteFixup

The Transplant function is used in the Delete function of the red-black tree to replace a node with its in-order successor or with one of its children (if it has no successor).

Figure 5: Transplant function psuedocode

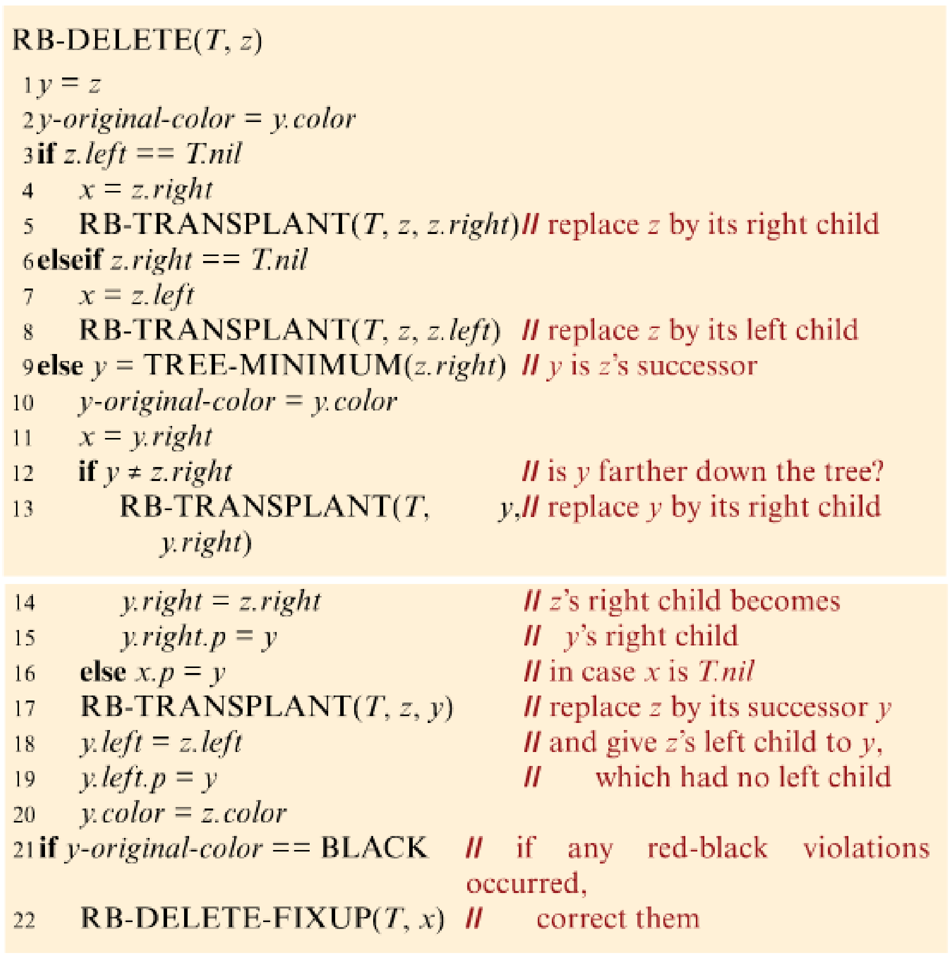


```
RB-TRANSPLANT(T, u, v)
if u.p == T.nil
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
v.p = u.p
```

Delete function handles three cases for deleting a node:

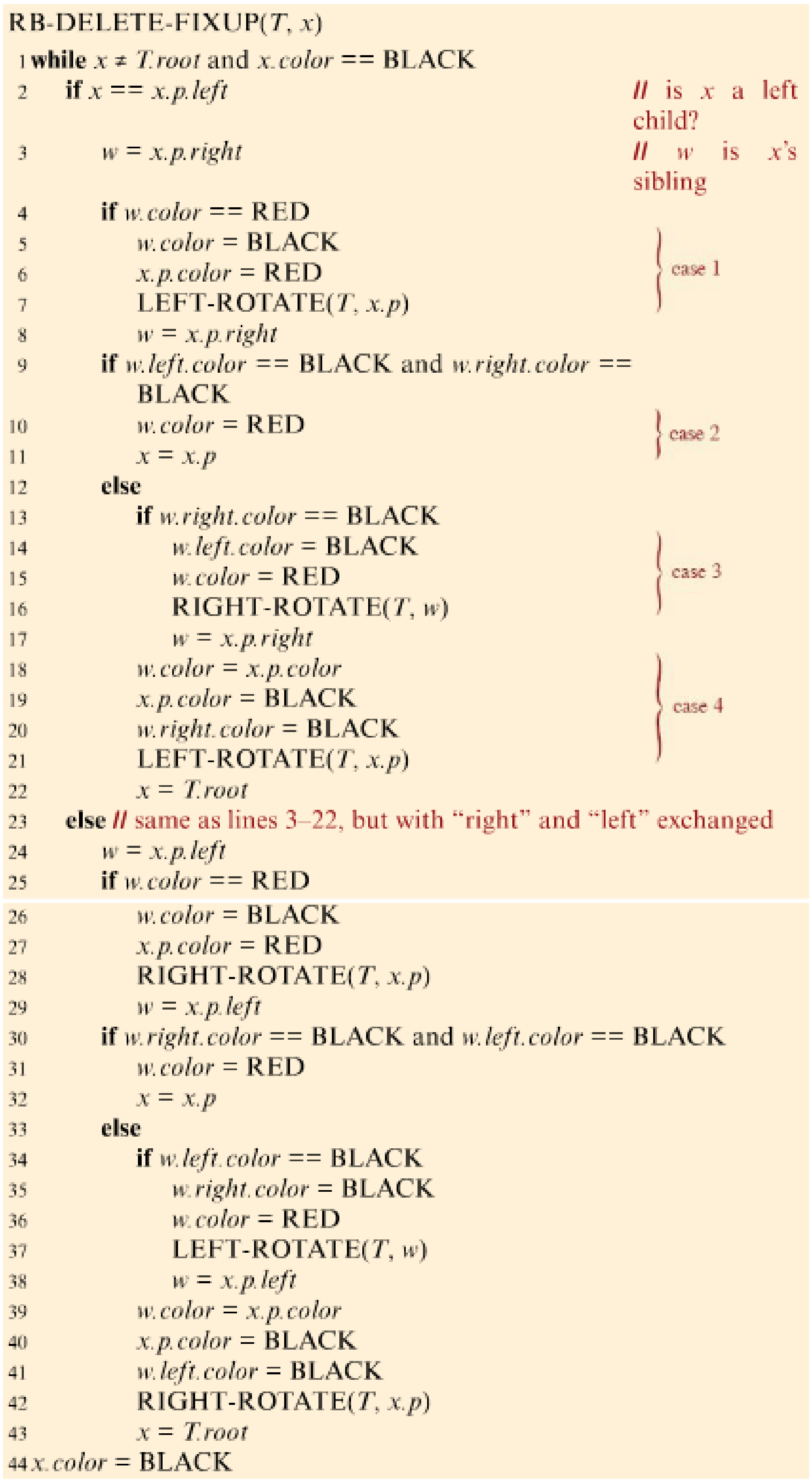
1. If the node to be deleted has no children (i.e., it is a leaf node), the function simply replaces the node with nullptr using the Transplant function.
2. If the node to be deleted has one child, the function replaces the node with its child using the Transplant function.
3. If the node to be deleted has two children, the function replaces the node with its in-order successor (the next larger node in the tree). The in-order successor is guaranteed to have at most one child, so the delete operation can be completed in the next step.

Figure 6: Delete function psuedocode



```
RB-DELETE($T$, $z$)
$y = z$
$y$-original-color $= y.color$
if $z.left == T.nil$
    $x = z.right$
    RB-TRANSPLANT($T$, $z$, $z.right$)   // replace $z$ by its right child
elseif $z.right == T.nil$
    $x = z.left$
    RB-TRANSPLANT($T$, $z$, $z.left$)    // replace $z$ by its left child
else $y$ = TREE-MINIMUM($z.right$)       // $y$ is $z$'s successor
    $y$-original-color $= y.color$
    $x = y.right$
    if $y \neq z.right$                  // is $y$ farther down the tree?
        RB-TRANSPLANT($T$, $y$, $y.right$)   // replace $y$ by its right child
        $y.right = z.right$              // $z$'s right child becomes
        $y.right.p = y$                  //   $y$'s right child
    else $x.p = y$                       // in case $x$ is $T.nil$
    RB-TRANSPLANT($T$, $z$, $y$)         // replace $z$ by its successor $y$
    $y.left = z.left$                    // and give $z$'s left child to $y$,
    $y.left.p = y$                       //     which had no left child
    $y.color = z.color$
if $y$-original-color == BLACK           // if any red-black violations occurred,
    RB-DELETE-FIXUP($T$, $x$)            //     correct them
```

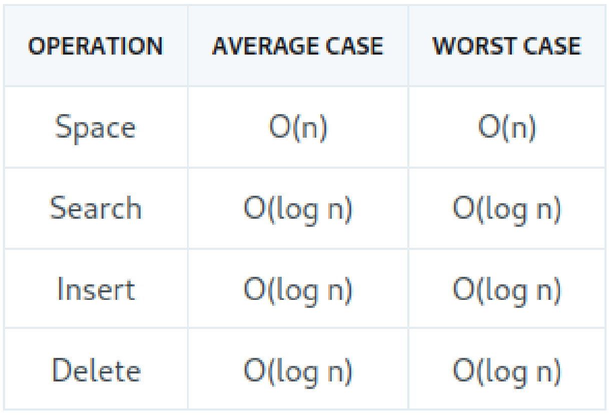
Figure 7: Deletefixup function psuedocode



```
RB-DELETE-FIXUP(T, x)
 1 while x ≠ T.root and x.color == BLACK
 2     if x == x.p.left                              // is x a left child?
 3         w = x.p.right                             // w is x's sibling
 4         if w.color == RED
 5             w.color = BLACK                       } case 1
 6             x.p.color = RED                       }
 7             LEFT-ROTATE(T, x.p)                   }
 8             w = x.p.right
 9         if w.left.color == BLACK and w.right.color ==
               BLACK
10             w.color = RED                         } case 2
11             x = x.p
12         else
13             if w.right.color == BLACK
14                 w.left.color = BLACK              } case 3
15                 w.color = RED                     }
16                 RIGHT-ROTATE(T, w)                }
17                 w = x.p.right
18             w.color = x.p.color                   } case 4
19             x.p.color = BLACK                     }
20             w.right.color = BLACK                 }
21             LEFT-ROTATE(T, x.p)                   }
22             x = T.root
23     else // same as lines 3–22, but with "right" and "left" exchanged
24         w = x.p.left
25         if w.color == RED
26             w.color = BLACK
27             x.p.color = RED
28             RIGHT-ROTATE(T, x.p)
29             w = x.p.left
30         if w.right.color == BLACK and w.left.color == BLACK
31             w.color = RED
32             x = x.p
33         else
34             if w.left.color == BLACK
35                 w.right.color = BLACK
36                 w.color = RED
37                 LEFT-ROTATE(T, w)
38                 w = x.p.left
39             w.color = x.p.color
40             x.p.color = BLACK
41             w.left.color = BLACK
42             RIGHT-ROTATE(T, x.p)
43             x = T.root
44 x.color = BLACK
```

2 Section 2: Complexity Analysis

Figure 8: Time Complexity Table



OPERATION	AVERAGE CASE	WORST CASE
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

2.1 Insert

The time complexity of the Insert function of a red-black tree is O(log n), where n is the number of nodes in the tree. This is because the function performs a standard binary search in the tree to find the correct position for the new node, and the time complexity of a binary search is O(log n).

After the new node is inserted, the function also performs a series of rotations and color changes to restore the red-black tree properties and maintain the balance of the tree. These operations take constant time (i.e., O(1)), so they do not affect the overall time complexity of the Insert function.

It's worth noting that the time complexity of the Insert function is dependent on the balance of the tree. If the tree is balanced, the time complexity of the Insert function will be O(log n), as described above. However, if the tree becomes unbalanced (e.g., due to repeated insertions and deletions), the time complexity may degrade to O(n).

2.2 Delete

The time complexity of the Delete function of a red-black tree is O(log n), where n is the number of nodes in the tree. This is because the function performs a standard binary search in the tree to find the node to be deleted, and the time complexity of a binary search is O(log n).

After the node is deleted, the function also calls the DeleteFixup function to restore the red-black tree properties and maintain the balance of the tree. The DeleteFixup function performs a series of rotations and color changes on the nodes of the tree until the red-black tree properties are satisfied again. These operations take constant time (i.e., O(1)), so they do not affect the overall time complexity of the Delete function.

It's worth noting that the time complexity of the Delete function is dependent on the balance of the tree. If the tree is balanced, the time complexity of the Delete function will be O(log n), as described above. However, if the tree becomes unbalanced (e.g., due to repeated insertions and deletions), the time complexity may degrade to O(n).

3 Section 3: Food For Thought

3.1 Can you think of any advantages of using the RB Tree as the underlying data structure?

There are several advantages to using a red-black tree as the underlying data structure for a Completely Fair Scheduler (CFS):

Red-black trees are self-balancing binary search trees, which means that the height of the tree is always O(log n), where n is the number of nodes in the tree. This allows the CFS to perform insert, delete, and search operations with a time complexity of O(log n).

Red-black trees maintain a balance between the left and right subtrees, which helps to distribute the load evenly across the entire tree. This can improve the overall performance of the CFS, as it reduces the chances of hot spots (areas of the tree that are accessed more frequently than others).

Red-black trees are relatively simple to implement and maintain, which can make them an attractive choice for a CFS.

Red-black trees support a wide range of operations, including insert, delete, search, minimum, maximum, and in-order traversal. This makes them a flexible choice for a CFS that needs to perform a variety of operations on the data.

3.2 Is the CFS used anywhere in the real world?

Yes, the Completely Fair Scheduler (CFS) is a scheduling algorithm that is used in the Linux kernel to schedule the execution of processes on a computer. It was introduced in the Linux kernel version 2.6.23, which was released in 2007, and has been the default scheduler in the Linux kernel since then.

The CFS is designed to provide good performance for a wide range of workloads, including real-time, batch, and interactive applications. It uses a red-black tree as the underlying data structure to store and organize the processes that are waiting to be scheduled.

The CFS is used on a wide range of devices, including desktop computers, servers, and embedded systems, and is widely considered to be a reliable and effective scheduling algorithm.

3.3 What is the maximum height of the RBTree with N processes? Upper bound T(N)=? Prove it.

The maximum height of a red-black tree with N nodes is O(log N). This is because a red-black tree is a self-balancing binary search tree, and the height of a binary search tree is always O(log N), where N is the number of nodes in the tree.

To prove this, we can use the fact that the height of a red-black tree with N nodes is at most 2 * log(N + 1). This is because the number of black nodes on any path from the root to a leaf is at least log(N + 1). Since each node in a red-black tree is either red or black, and there must be at least one black node on each path, the height of the tree is at most twice the number of black nodes on the longest path.

Therefore, we can say that the upper bound on the time complexity of the red-black tree (T(N)) is O(log N).