Istanbul Technical University
Faculty of Computer and Informatics
Computer Engineering Department

BLG 335E
Homework 3 Report

Erce Gülmez - 150210728

December 23nd, 2022

Contents

1 Description of Code

In this project, I implemented the Red-Black Tree data structure. There are 2 different classes for this implementation:

- Node class : represents the processes and the nodes in the RBTree.
- RBTree : represents Red-Black Tree structure, includes the methods for maintaining the properties of Red-Black Trees.

1.1 Node class

Node class represents the nodes of our binary search tree. There is nothing special about the methods of this class, it has some basic getters and setters for every field of the class. The fields(attributes) of an object of Node are as follows:

- int burstTime : Burst time of the process
- Node* left : Left Child
- Node* right : Right Child
- Node* parent : Parent
- string color : Color of the Node
- string name : ID of the Node
- bool isCompleted : Field for completion of the process
- int arrivalTime : Arrival time of the process
- int vtime : Virtual Running Time of the process

1.2 RBTree class

This class has the implementation of the Red-Black Trees. This structure satisfies the Binary Search Tree properties and augments extra details to it for self-balancing the tree. Here is the attributes for this class:

- Node* root : represents the root of the RBTree. It is NIL when the tree is empty.
- Node* NIL : represents the children of the leaves in the RBTree. It has the Black color, "NIL" name, nullptr children and 0 value for the other attributes of a Node.

The class has some methods that all BSTs have such as getRoot(), inorder_traverse(), isEmpty() and setRoot(). Since they are very straightforward, there is no need to explain them. You can check them from the source code.

Other than those simple methods, this class has its own methods for manipulating its data and maintaining its structure. Let's see them one-by-one:

1.2.1 right_rotate

This function applies a right-rotation to a Node which is given as the parameter. The process is the simple rotation procedure used in balanced trees in general. Here is the psuedocode for this function:

Algorithm 1 The right rotation algorithm

```
function RIGHT_ROTATE(n)
    p ← LEFT(n)
    if PARENT(n) not NULL then
        if n = n.parent.right then
            p.parent ← n.parent
            n.parent.right ← p
        else
            p.parent ← n.parent
            n.parent.left ← p
        end if
        n.left ← p.right
        p.right ← n
        n.parent ← p
        n.left.parent ← n
    else
        n.left ← p.right
        p.right ← n
        n.parent ← p
        n.left.parent ← n
        set the root as p
        p.parent ← NULL
    end if
end function
```

1.2.2 left_rotate

This function applies left rotation to the given node. The rotation is the basic rotation that is applied in every simple balancing tree. The psuedocode for this operation is given below:

Algorithm 2 The left rotation algorithm

```
function LEFT_ROTATE(n)
    p ← n.right
    n.right ← p.left
    p.left.parent ← n
    p.left ← n
    if n is an internal node then
        p.parent ← n.parent
        if p.parent.left = n then
            p.parent.left ← p
        else
            p.parent.right ← p
        end if
    else
        set the root as p
        p.parent ← NULL
    end if
    n.parent ← p
end function
```

1.2.3 insert_node

This function is used for inserting a node to the tree. The function first find the place to insert this node satisfying the BST properties. Then it basically inserts that node by making the connections between parent nodes etc. At last, it sets the color of the newly inserted node as Red and calls insert_fixup function which maintains the RBTree properties after insertion operation. Here is the psuedocode for the function:

Algorithm 3 The insertion algorithm

```
function INSERT_NODE(z)
    y ← NIL
    x ← root
    while x not NIL do
        y ← x
        if z.data < x.data then
            x ← x.left
        else
            x ← x.right
        end if
    end while
    z.parent ← y
    if y == NIL then
        root ← z
    else if z.data < y.data then
        y.left ← z
    else
        y.right ← z
    end if
    z.left ← NIL
    z.right ← NIL
    z.color ← RED
    INSERT_FIXUP(z)
end function
```

1.2.4 insert_fixup

This function maintains the RBTree properties after the insertion of a node. The function gets the newly inserted node as its parameter and does the operations according to these cases:

- Case 1 : Uncle of the inserted node is Red
- Case 2 : Uncle of the inserted node is Black and the inserted node is the right child of its parent
- Case 3 : Uncle of the inserted node is Black and the inserted node is the left child of its parent
- Case 4 : Inserted node is the root

Here is the psuedocode for the function which covers all these 4 cases:

Algorithm 4 The fixup algorithm for insertion

```
function INSERT_FIXUP(z)
    while z.parent.color == RED do
        if z.parent == z.parent.parent.left then
            y ← z.parent.parent.right
            if y.color == RED then
                z.parent.color ← BLACK
                y.colort ← BLACK
                z.parent.parent.color ← RED
                z ← z.parent.parent
            else
                if z == z.parent.right then
                    z ← z.parent
                    LEFT_ROTATE(z)
                end if
                z.parent.color ← BLACK
                z.parent.parent.color ← RED
                RIGHT_ROTATE(z.parent.parent)
            end if
        else
            (same procedure for the right)
        end if
    end while
    root.color ← BLACK
end function
```

1.2.5 transplant

This function is the BST's transplant operation developed for RBTree. It is used during the deletion operation. Here is the psuedocode of the transplant operation:

Algorithm 5 The transplant algorithm

```
function TRANSPLANT(n, p)
    if n.parent == NULL then
        root ← p
    else if n.parent.left == n then
        n.parent.left ← p
    else
        n.parent.right ← p
    end if
    p.parent ← n.parent
end function
```

1.2.6 delete_node

This is the augmented form of BST's delete operation. The function applies transplant operation if one of its children is NULL. Otherwise, it applies the classic deletion approach for

the BST: finding the successor of the node to be deleted and swap them. In the end, it call the delete_fixup function to maintain the RBTree properties after deletion. Note that the function just removes the element from the tree, not deletes it from the program since the removed process can be inserted back into the tree in CFS Schedular procedure. Here is the psuedocode for this purpose:

Algorithm 6 The deletion algorithm

```
function DELETE_NODE(n)
    p ← NULL
    colorOfDeleted ← n.color
    if n.left == NIL then
        p ← n.right
        TRANSPLANT(n, p)
    else if n.right == NIL then
        p ← n.left
        TRANSPLANT(n, p)
    else
        temp ← n.right
        while temp.left not NIL do
            temp ← temp.left
        end while
        colorOfDeleted ← temp.color
        p ← temp.right
        if temp.parent == n then
            n.parent ← temp
        else
            TRANSPLANT(temp, temp.right)
        end if
        TRANSPLANT(n, temp)
        temp.color ← colorOfDeleted
    end if
    if colorOfDeleted == BLACK then
        DELETE_FIXUP(p)
    end if
end function
```

1.2.7 delete_fixup

This function is used for maintaining the RBTree properties after the deletion operation. There are 4 cases in the fixup routine:

- Case 1 : The sibling of the node to be deleted is red
- Case 2 : The sibling of the node to be deleted and its children are Black
- Case 3 : The sibling of the node to be deleted and its right child is Black, left child is Red
- Case 4 : The sibling of the node to be deleted is Black and its children are Red

Here is the psuedocode that covers all these 4 cases when the node to be deleted is the left or the right child of its parent seperately:

Algorithm 7 The fixup algorithm of deletion

```
function DELETE_FIXUP(n)
    p ← NULL
    while n != root and n.color == BLACK do
        if n.parent.left == n then
            p ← n.parent.right
            if p.color == RED then
                p.color ← BLACK
                n.parent.color ← RED
                LEFT_ROTATE(n.parent)
                p ← n.parent.right
            end if
            if p.left.color == BLACKandp.right.color == BLACK then
                p.color ← RED
                n ← n.parent
            else if p.right.color == BLACK then
                p.left.color ← BLACK
                p.color ← RED
                RIGHT_ROTATE(p)
                p ← n.parent.right
            else
                p.color ← n.parent.color
                n.parent.parent.color ← BLACK
                p.right.color ← BLACK
                LEFT_ROTATE(n.parent)
            end if
        else
            (same procedure for the right)
        end if
    end while
end function
```

2 Complexity Analysis

2.1 Piece-wise Complexity Analysis

In this part, the complexity analysis of the implemented functions will be covered. The functions in the Node class are just getters and setters as explained in the previous section. These functions have the complexity of O(1) since they return values of attributes or assign values to attributes. The functions of RBTree are analysed one-by-one below:

2.1.1 findMinvruntime

The function gets the smallest key that exists in the tree, so it looks at the leftmost element. The path taken to reach the leftmost element is equal to log(n) since the height of the tree is log(n) in a balanced tree with n elements. So the complexity is O(log(n)).

2.1.2 inorder_traverse

The traversal takes O(n), linear time because every node is visited and printed one-by-one.

2.1.3 right_rotate and left_rotate

The rotation procedure is just updating the connections between the nodes according to some cases. For this reason the algorithm takes constant time. The complexity is O(1).

2.1.4 insert_node and insert_fixup

The insert operation is as simple as the BST's insertion operation. The node to be inserted will be a leaf in the tree and finding the correct spot for that node is the main thing here. Since the RBTree is a BST, binary search is applicable. So, finding the spot for the node takes O(log(n)). After this operation the function calls insert_fixup function for the inserted node.

The fixup function does some recolorings and rotations. The procedure starts from the bottom of the tree and in the worst case, goes until it reaches the root. Since the height of the tree is 2log(n+1) maximum, this procedure also takes O(log(n)) time.

In total, inserting the node and fixing the structure takes O(log(n)) since both inserting and fixing the structure takes about log(n) time seperately.

2.1.5 delete_node, transplant and delete_fixup

First of all, we can clearly see that transplant operation takes constant time since it just does some reconnection between some nodes according to the case.

The deletion part finds the node and swaps it with its successor. Finding the successor takes O(log(n)) time in the worst case which is the root being the node to be deleted. After the swap and some transplant operations according to the case, this function calls the fixup method. So, without the fixup method call, it takes log(n) time to delete a node.

The fixup method also takes log(n) time just like the fixup of insertion since it does some constant time operations, such as reconnection, starting from one end of the tree to the other end in the worst case.

So in total, deletion also takes O(log(n)) time since deletion and fixing the structure both take O(log(n)) time.

2.2 Overall Complexity Analysis

Since the CFS scheduler depends on insertion, deletion and searching(finding the task with the smallest vruntime) in a RBTree, the overall complexity of the whole procedure takes O(log(n)) time.

3 Food For Thought

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

Since the RB Tree is a self-balancing tree, the tree will remain balanced during the operations made on the tree. This means that the height of the tree will remain maximum 2log(n+1) during the program. That's why the operations such as the insertion, deletion, search will take O(log(n)) time no matter what.

3.2 Is the CFS used anywhere in the real world?

In the real world, this technique is not commonly encountered but can be used in some factory tasks that are done using electronic devices. For example in the production of a good, this procedure can be used to organize the working scheme for the automated factory robots.

3.3 What is the maximum height of the RBTree with N processes?

- For a BST, let x be the minimum number of nodes from root to NIL. The relationship between x and n will be: $n >= 2^x - 1$.
- Therefore, $x <= log(n + 1)$.
- For the edge case, if we build the path using one black and one red node each time, the height will be: $height <= 2log(n + 1)$.

4 Compiling Procedure

- The Makefile is provided in the project folder.
- By typing "./homework3 [input_file]", you can compile the program.
- For different test cases, change the args[] part in the launch.json file.
- The program will automatically produce output.txt file for the output information right after running the program.