Istanbul Technical University
Faculty of Computer and Informatics
Computer Engineering Department

Report of Assignment 3

Ahmet Secaettin Alıcı - 150190097

December 30nd, 2022

Contents

1 Description of Code **1**

2 Complexity Analysis **2**

3 Food For Thought **3**

1 Description of Code

Class Node: Because CFS use red black trees, I had to use red black trees too. I got help from our textbook Introduction to Algorithms. I used two classes to realize red black tree, class Node and class RBT. Class Node has attributes name(string) arrival time(int) and burst time(int), key(int). As you can see class Node represents the processes that CFS is scheduling, attribute key is represents the vruntime of process.

Class RBT: As the name suggests tihs class represents the red black tree itself. As attributes it has, Node* root and Node* nil. Pointer nil is the all leaves of the tree and parent of the root. As methods it has minimum, left rotate, right rotate, insert, insert fixup, transplant, delete, delete fixup, tree search, inorder tree walk. Some of this methods are interesting and very different from binary search tree and unique to red black trees itself, I will write especially their pseudocodes here.

```
RB-INSERT-FIXUP(T,z):
while z.p.color==RED
        if z.p==z.p.p.left
                y=z.p.p.right
                if y.color==RED
                        z.p.color=BLACK
                        y.color=BLACK
                        z.p.p.color=RED
                        z=z.p.p
                else
                        if z==z.p.right
                                z=z.p
                                LEFT-ROTATE(T,z)
                        z.p.color=BLACK
                        z.p.p.color=RED
                        RIGHT-ROTATE(T,z.p.p)
        else
                y=z.p.p.left
                if y.color==RED
                        z.p.color=BLACK
                        y.color=BLACK
                        z.p.p.color=RED
                        z=z.p.p
                else
                        if z==z.p.left
                                z=z.p
                                RIGHT-ROTATE(T,z)
                        z.p.color=BLACK
                        z.p.p.color=RED
                        LEFT-ROTATE(T,z.p.p)
T.root.color=BLACK
```

```
RB-DELETE-FIXUP(T,x):
while x!=T.root and x.color==BLACK
        if x==x.p.left
                w=x.p.right
                if w.color==RED
                        w.color=BLACK
                        x.p.color=RED
                        LEFT-ROTATE(T,x.p)
                        w=x.p.right
                if w.left.color==BLACK and w.right.color==BLACK
                        w.color=RED
                        x=x.p
                else
                        if w.right.color==BLACK
                                w.left.color=BLACK
                                w.color=RED
                                RIGHT-ROTATE(T,w)
                                w=x.p.right
                        w.color=x.p.color
                        x.p.color=BLACK
                        w.right.color=BLACK
                        LEFT-ROTATE(T,x.p)
                        x=T.root
        else
This half is the same but left and right are interchanged with each other.
x.color=BLACK
```

Especially this helper methods what makes red black tree a red black tree and different from a a simple binary search tree.

2 Complexity Analysis

First height of a red black tree h = O(logn). Let's think about methods of red black trees one by one. Insert and insert fixup are O(h) that is O(logn). Similarly delete and delete fixup are O(logn) too. Delete and insert are O(logn) because from the root we traverse to the node we are interested in, this is clearly O(logn). Their fixup'es are O(logn) too because these methods either jump to parent nodes and enter while loop again or they do couple or rotations and finish. Inorder travel of red black tree is again clearly O(n) because it does traverse every node.

3 Food For Thought

- As we know red black tree is a kind of binary search tree but different from a normal binary search tree it is self balanced. Because of its 5 properties and its algorithms(especially the fixup algorithms that I have shown above) which protecting these properties, it is a self balanced tree. So if we would have to use binary search tree we would want it to be balanced to benefit from trees' features, and one way to ensure that is using red black trees. Because of this red black trees are important.
- CFS is currently used in linux kernel to manage CPU resource allocation between processes and as the name suggests it tries to do it fairly.
- Maximum height of the RBTree with N processes is 2log(N+1) so T(N) = O(logN).

Proof: This is proof of **Lemma 13.1** from our textbook Introduction To Algorithms aka clrs.

We call the number of black nodes on any simple path from, but not including, a node x down to a leaf the black-height of the node, denoted bh(x).

We start by showing that the subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes. We prove this claim by induction on the height of x. If the height of x is 0, then x must be a leaf (T.nil), and the subtree rooted at x indeed contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes. For the inductive step, consider a node x that has positive height and is an internal node with two children. Each child has a black-height of either bh(x) or bh(x) - 1, depending on whether its color is red or black, respectively. Since the height of a child of x is less than the height of x itself, we can apply the inductive hypothesis to conclude that each child has at least $2^{bh(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains at least $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} + 1) = (2^{b(x)} - 1)$ internal nodes, which proves the claim. To complete the proof of the lemma, let h be the height of the tree. According to property 4, at least half the nodes on any simple path from the root to a leaf, not including the root, must be black. Consequently, the black-height of the root must be at least h/2; thus,

$$n \geq 2^{h/2} - 1$$

. Moving the 1 to the left-hand side and taking logarithms on both sides yields

$$lg(n+1) \geq h/2, \qquad \text{or} \qquad h \leq 2lg(n+1).$$

This proof was taken from our textbook Introduction to Algorithms.