ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 335E

Analysis of Algorithms I
HOMEWORK REPORT

HOMEWORK NO : 3

HOMEWORK DATE : 30.12.2022

150200915 : ADIL MAHMUDLU

FALL 2022

Contents

1 DESCRIPTION OF CODE AND COMPLEXITY ANALYSIS

Data structures used in the implementation are Nodes and RBTree (Red-Black Tree). They are used to create balanced Binary Search Trees. Node class has no method. RBTree has multiple methods such as setter and getters, as well as insertion and deletion fixups, rotations, insertion, deletion, and inorder traversal functions.

1.1 Rotate Left/Right

```
rotateLeft(Node x):
    xr <- x.r
    x.r <- xr.l
    if xr.l != leaf
        xr.l.p <- x
    xr.p <- x.p
    if xr.p == NULL
        root <- xr
    else if xr.p.l <- x
        xr.p.l <- xr
    else
        xr.p.r <- xr
    xr.l <- x
    x.p <- xr
```

The above pseudocode is of rotateLeft function. The pseudocode of rotateRight function is the same with left and right keywords exchanged. These methods are used to fix the problems that does not obey the rules of red-black tree. The code has time complexity $T(n) = O(1)$. It runs in constant time.

1.2 MinNode/MaxNode

```
minNode(Node x):
    if x == NULL or x == leaf
        return x
    while x.l != leaf
        x <-x.l
    return x
```

The function of maxNode is very similar, using $x.r$ instead of $x.l$. The function runs in $T(n) = O(logn)$ time complexity, as it executes for the height of the tree, which is maximum of $2logn$.

1.3 Insert Fixup

```
insertFixup(Node x):
    while x != root and x.p.colour == RED
        if x.p.p.l == x.p
            xu <- x.p.p.r
            if xu.colour == RED
                xu.p.colour <- BLACK
                x.p.colour <- BLACK
                x.p.p.colour <- RED
                x <- x.p.p
            else
                if x == x.p.r
                    x <- x.p
                    rotateLeft(x)
                x.p.colour <- BLACK
                x.p.p.colour <- RED
                rotateRight(x.p.p)
        else
            same as if with left and right exchanged
    root.colour <- BLACK
```

The function works with $T(n) = O(logn)$ time complexity, as it only works for the height of the tree, and height of the tree is at most $2logn$. It makes sure the insertion does not break the rules of red-black tree. There are handful of possible cases, and the function determines which case is present and fixes until there are no problems left.

1.4 Delete Fixup

```
deleteFixup(Node x):
    while x.colour == BLACK
        if x == x.p.l
            xs <- x.p.r
            if xs.colour == RED
                xs.colour <- BLACK
```

```
            x.p.colour <- RED;
            rotateLeft(x.p);
            xs <- x.p.r;
        if xs.l.colour == BLACK and xs.r.colour == BLACK
            xs.colour <- RED
            x <- x.p
            continue
        if xs.r.colour == BLACK
            xs.l.colour <- BLACK
            xs.colour <- RED
            rotateRight(xs)
            xs <- x.p.r
        xs.colour <- x.p.colour
        x.p.colour <- BLACK
        xs.r.colour <- BLACK
        rotateLeft(x.p)
        x <- root
    else
        same as if with left and right exchanged
x.colour <- BLACK
```

The code above has time complexity of $T(n) = O(logn)$ because of the reasons mentioned in previous subsections. The method is called after deletion to make sure the deletion does not break any rule, and if it does, the according fixes should be made.

1.5 Transplant

```
transplant(Node a, Node b):
    if a.p == NULL
        root <- b
    else if a.p.l == a
        a.p.l <- b
    else
        a.p.r <- b
    if b != NULL
        b.p <- a.p
```

This method is not necessary to implement, but makes it easier to write other functions. It works on constant time, i.e. $T(n) = O(1)$. The method puts one subtree in the place of another and makes necessary connections.

1.6 Insert

```
insert(Node x):
    x.l <- leaf
    x.r <- leaf
    p <- this.root
    if p == leaf
        root <- x
        root.colour <- BLACK
        return
    while 1
        if x.vRunTime < p.vRunTime
            if p.l == leaf
                p.l <- x
                x.p <- p
                break
            p <- p.l
            continue
        else
            if p.r == leaf
                p.r <- x
                x.p <- p
                break
            p <- p.r
            continue
    insertFixup(x)
```

One of the main functions of the Tree data structures, RBTree being one of them, is insert function. The function runs in $T(n) = O(logn)$ time and calls insertFixup function which runs in the same complexity, so overall complexity of this function is $T(n) = O(logn)$.

1.7 Delete

```
delete(Node x):
    originalx <- x
    originalColour <- x.colour
    if x.l == leaf
        replacement <- x.r
        transplant(x, x.r)
    else if x.r == leaf
```

```
        replacement <- x.l
        transplant(x, x.l)
    else
        successor <- minNode(x.r)
        originalColour <- successor.colour
        replacement <- successor.r
        if successor.p == x
            replacement.p <- successor
        else
            transplant(successor, successor.r)
            successor.r <- x.r
            successor.r.p <- successor
        transplant(x, successor)
        successor.l <- x.l
        successor.l.p <- successor
        successor.colour <- x.colour
    delete originalx
    if originalColour == BLACK
        deleteFixup(replacement)
```

Another important function of Trees are delete function. The function calls fixup that has time complexity $T(n) = O(logn)$ and therefore itself has the same time complexity.

1.8 Inorder Traversal

```
inorderTraversal(Node x):
    traverse <- ""
    if x != leaf
        traverse <- traverse + inorderTraversal(x.l)
        traverse <- traverse+ x.processID + ":" + x.vRunTime + "-"
        if x.colour
            traverse <- traverse + "Black"
        else
            traverse <- traverse + "Red"
        if maxNode(root) != x
            traverse <- traverse + ";"
        traverse <- traverse + inorderTraversal(x.r)
    return traverse
```

This function is used to return the string of the format required by the homework. It

passes through every node once, and therefore has the time complexity $T(n) = O(n)$, highest among all the rest of the functions.

1.9 Overall Complexity

Overall Complexity of the program depends the number of iterations, and thus the Simulator Run Time input. The most time-consuming part of the program is writing to the output file, where program makes use of inorderTraversal function with time complexity of $T(n) = O(n)$, as mentioned, highest among all the rest of the functions program utilizes. This function is called for each iteration of program, namely, Simulator Run Time times. Thus, the overall complexity can be calculated using $T(n) = O(SRT * n)$ where n is the number of nodes and SRT is Simulator Run Time.

2 FOOD FOR THOUGHT

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

Red-Black Trees are a type of Self-Balancing Binary Search Tree that has multiple advantages, most prominent being performance and simplicity *Performance:* The RBTrees are balanced and therefore the height of the tree does not get as big as other BSTs. The Tree's height is at most 2*logn, and this provides enough performance boost that it works on $O(logn)$ time complexity on both insertion, deletion, and search. *Simplicity:* Red-Black Trees are comparably easier to implement, understand, and maintain than other types of BSTs, such as AVL trees. (Adelson, Velski Landis).

2.2 Is the CFS used anywhere in the real world?

CFS is a process scheduler that manages the allocation of cpu time and has been used in Linux kernel for quite some time, being the default scheduler since its introduction. It is also used in pletora of servers, mobile devices, embedded systems and more that utilize Linux kernel.

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

As mentioned above, the maximum height the tree can reach is at most 2logn. The reason for that is because in a tree with the lowest height of a leaf being k, all of which

are black nodes, the biggest possible height of a leaf can be 2k, because there will be a red node for each of the black nodes, and because the number of black nodes must be the same starting from root to the leafs, and red nodes cannot be each other's parent/child, there will be k black and at most k red nodes in this particular branch, totaling 2k height, and the rest of the branches being of height in between k and 2k, the overall average of the tree is log(n) with n given nodes.