CSC263: Assignment 3

Jingqi Zhang, Yulin WANG, Tianjing Ruan

Due Date: *07-06-2019*

Question	**Written By**	**Checked By**
1	Tianjing Ruan	Jingqi Zhang
2	Yulin WANG	Jingqi Zhang

Question 1

a)

To make it possible to run minimum(x), maximum(x), successor(x) and predecessor(x) in O(1) time, we should store 4 augmented fields at each node of the AVL tree compared with original one (we divide 4 fields into two types for better explanation later): 1) x.min and x.max , which are the pointers pointed to smallest and the biggest node of the subtree rooted at the node x, respectively; 2) x.successor and x.predecessor, which are the pointers pointed to the successor(next biggest key node to x) and predecessor(next smallest key node to x), respectively.

-Explain minimum(x) and maximum(x).

For each node x in the AVL Tree, x.min points to the node of either x.left.min (the node which has minimum value in its left subtree if its left child exists) or x itself(its left child is NIL), and x.max points to the node of either x.right.max (the node which has maximum value in its right subtree if its right child exists) or x itself(its right child is NIL). So when calling minimum(x) and maximum(x), they just return the node, which cost O(1) time as required.

Now we should consider how to keep x.max and x.min always correct when we update our tree, that is only insertion and deletion may change x.min and x.max.

For insertion, each newly inserted node's max and min pointers are initially pointed to itself (since both left and right child are NIL now).Then we will show that the min and max information can be maintained at the same time as the insertion is carried out, without affecting the classic insertion run time. When we do classic insertion, with or without rotation, we have actually traversed all the nodes which min and max pointers should be changed. So with each key comparison and pointer rotation of newly inserted node to go to correct position, we update these nodes' min and max pointers using minimum() and maximum(), which just change the pointers' pointing and takes constant time. Since in an AVL tree the classic insertion costs O(log n) time, then the parallel maintenance takes O(log n) time, without affecting the running time of insert.

It follows a similar method in the DELETE process. When deleting a node x using the classic deletion, with or without rotation, we also have actually traversed all the nodes which min and max pointers should be changed. So with each key comparison and pointer rotation to keep AVL tree as a correct form, we update these nodes' min and max pointers using minimum() and maximum(), which just change the pointers' pointing and takes constant time. . Since in an AVL tree the classic deletion costs O(log n) time, then the parallel maintenance takes O(logn) time, without affecting the running time of delete.

-Explain successor(x) and predecessor(x).

First notice that since successor and predecessor are considered in the whole tree, not the subtree rooted at a node, and obviously a node's successor is its successor's predecessor, vise-versa. So rebalance doesn't affect this 2 fields, we only need to consider the new node exact insertion and the required deletion.

When inserting a new node x, we should consider in order traversal, the node goes to the left child of a node with value greater than it , and the node goes to right child of a node with value smaller than it. Therefore, the last "left" node which is greater than x is its successor and the last "right" node which is smaller than x is its predecessor. If x has a minimum value of entire AVL tree, it does not have predecessor. Similarly, if x has a maximum value of the tree, it does not have successor. In addition, x.predecessor.successor should point to x instead of x.successor, and x.successor.predecessor should point to x instead of x.predecessor. This process takes constant amount of additional time in classic insertion operation. Since in an AVL tree the classic insertion costs O(log n) time, then the parallel maintenance takes O(log n) time, without affecting the running time of insert.

When deleting a node x, the update is little different than insertion. we should update x.predecessor.successor and x.successor.predecessor since these point to x previously. Firstly, we make x.predecessor.successor points to x.successor and x.successor.predecessor points to x.predecessor. Then we delete x and do necessary rotations, which takes constant amount of additional time in classic deletion operation. Since in an AVL tree the classic deletion costs O(log n) time, then the parallel maintenance takes O(log n) time, without affecting the running time of delete.

Below is the pseudo-code.

```
minimum(x)
IF x.left = NIL
    return x
return x.left.min

maximum(x)
IF x.right = NIL
    return x
return x.right.max

successor(x)
return x.successor

predecessor(x)
return x.predecessor
```

We have to show you the insertion and deletion with some changing part. Since as we explain, in order to put parallel maintenance of x.max, x.min, x.successor, x.prdecessor, we need to add this maintenance UpdateMinMax(), RotateMinMax(), InsertSuccPre() and DeleteSuccPre() in classic function: InitNode(), Insert(), Delete(), RightRotate() and LeftRotate(), all classic codes based on the materal in class and GitHub course note.

```
InitNode(Int k)
x := new Node
x.left ← NIL, x.right ← NIL, x.height ← 0, x.key ← x
x.max ← x, x.min ← x                          # Augmented field initial values
x.successor ← NIL, x.predecessor ← NIL      # Augmented field initial values
RETURN x

Insert(Tree A, y)
IF A = NIL
    A ←InitNode(y)
ELSE
    IF y < A.key
        A.key ← y.predecessor
        Insert(A.left, y)
    ELSE
        A.key ← y.successor
        Insert(A.right, y)
    UpdateMinMax(A)
    InsertSuccPre(A)
    Rebalence(A) # Compute balence factor and keep roatation part, no change
                   except specific rotation partial function, show as below

Delete(Tree A, y)
IF A = NIL
    RETURN ERROR
IF y = A.key
    ptNode rm ← A
    IF A.left = NIL && A.right = NIL
        A ← NIL
    ELSE IF A.left != NIL && A.right = NIL
        A = A.left
        UpdateMinMax(A)
        DeleteSuccPre(A)
    ELSE IF A.left = NIL && A.right != NIL
        A ← A.right
        UpdateMinMax(A)
        DeleteSuccPre(A)
    ELSE
```

```
        Node SUB ← detachMin(A.right)
        SUB.left ← A.left
        SUB.right ← A.right
        SUB ←A
        UpdateMinMax(A)
        DeleteSuccPre(A)
    rm ← NIL
ELSE IF y < A.key
    Delete(A.left, y)
    UpdateMinMax(A)
    DeleteSuccPre(A)
ELSE
    Delete(A.right, y)
    UpdateMinMax(A)
    DeleteSuccPre(A)
    Rebalence(A) # Compute balence factor and keep roatation part, no change
                   except specific rotation partial function, show as below

LeftRotate(ptNode ptX)
ptNode ptY ← X.right
ptNode ptZ ← Y.left
Y.left ← ptX
X.right ← ptZ
ptX ← ptY
RotateMinMax(Z, X, Y)

RightRotate(ptNode ptX)
ptNode ptY ← X.left
ptNode ptZ ← Y.right
Y.right ← ptX
X.left ← ptZ
ptX ← ptY
RotateMinMax(Z, X, Y)

RotateMinMax(Node O, Node P, Node Q)
UpdateMinMax(O)
UpdateMinMax(P)
UpdateMinMax(Q)
```

Below are the maintenance functions we use.

```
UpdateMinMax(Node A)
A.max ← maximum(A)
A.min ← minimum(A)

DeleteSuccPre(Node A)
```

```
    A.successor ← A.predecessor.successor
    A.predecessor ← A.successor.predecessor

InsertSuccPre(Node A)
    A ← A.predecessor.successor
    A ← A.successor.predecessor
```

Question 2

a)

1) We can use hash table as the data structure in this algorithm.
Suppose we have a hash table H under SUHA. There is a constant load factor α and the size of hash table H is m, where $\alpha = w/m$

2) Psuedo code.

```
array Word_Puzzle(List Words, 2D array)
    data := List($w$ words)
    H ← load_Hash(data)
    A := new_array
    table := 2D array
    for every entry in table:
            word = create_word(entry)          # different lengths of new words
            if H.search(word) $\neq$ NIL:
                    A.append(tuple(word, coordinates))
    return A
```

3) Describe and explain the algorithm.
Firstly, we put all w words in the list to hash table H. Then we use the 2D array of characters to create new words by successively connect the characters through eight different directions(up-down, down-up, left-right, right-left, and diagonally in all four directions) for each entry and different lengths. And for each new word, we try to search(word) in the hash table H, if found, then add the search result as a tuple containing the word and corresponding coordinates of beginning and ending characters for the word into an array called A. If there are more than one pair of coordinates found, the tuple should include all possible coordinates for the word. Finally, return the array A.

4) Explain the Worst-Case time complexity.
Since the hash table H has constant load factor α and under SUHA, so both the insert and search operations on hash table H take constant time $O(1)$.
Firstly, load_Hash(List words) will run in $O(w)$ since there are w elements(words) to insert. Then, when creating new words from 2D array, it takes at most $O(8rc)$ time to go over every entry and go through 8 different directions, since the size of the 2D array is row $\times$ column (r$\times$c) and the length of each word is a small constant. And for each new word, it will take at most $O(1)$ time to search(word) in hash table H and append(tuple) into array A. Thus, it takes $O(rc)$ for creating words and finally getting array A. Since all other operations take at most $O(1)$, so in total, the algorithm Word_Puzzle takes at most $O(rc + w)$.