CSC265 Assignment 2

Written by Kaiyang Wen, 1002123891, kai.wen@mail.utoronto.ca
Revised by Mohan Zhang, 1002748716, morgan.zhang@mail.utoronto.ca

1 Question 1.

Claim: For extend the AVL tree does not involve re-balance.
Proof: For any node in that AVL tree, suppose the height before extend is m, then after extend, the height of it is m+1. So, if a node's balance factor is in k before, then after extend, it is still k.

```
;
; Parse an AVL Tree to be a Red-Black Tree with external nodes.
;
AVLToRed-Black(T):
    ;
    ; Invoke the Colour(T) function to colour all AVL Tree nodes.
    ; After that, force the colour of root node to be BLACK.
    ;
    Colour(T)
    T.root.colour := BLACK
end procedure.

;
; Assume T is a valid AVL Tree, returns the number of black nodes in every path
; from T to an external leaf. This is a recursive function.
;
int Colour(T):
    ;
    ; Base case is when T is a leaf-node already. In this case, we need to set
    ; the children of T to be external nodes, and colour T to be a red.
    ;
    ; When BF(T)<1, T is either balanced or left-leaning. In either cases, we
    ; recursively colour the nodes on the left subtree first, since the left
    ; subtree is at least as high as the right subtree. Then, we colour the
    ; right subtree, along with some special operations accordingly.
    ;
    ; Otherwise, according to the definition of AVL Tree, BF(T)=1, so T is right
    ; leaning. We would do "symmetric" procedure(see the comments).
    ;
    if IsLeaf(T),
        T.colour := RED
        AddTwoExternalBlackNodes(T)
        return 1.
    else if BF(T) < 1,
        ; {MARK 1}
        leftblacks := Colour(T.left)
        ;
        ; {MARK 2}
        ; When the right subtree of T does not exist, we need to add an external
        ; black node at the right subtree.
        ; Otherwise, the right subtree of T exists, and we need to colour the
        ; nodes on the right subtree.
```

```
    ;
    var rightblacks
    if T.right is NIL,
       T.right := CreateExternalBlackNode()
       rightblacks := 1
    else,
       ; {MARK 3}
       rightblacks := Colour(T.right)
    endif.
    ;
    ; Now, we have coloured both the left and right subtrees.
    ; However, since the height of left subtree can be higher than that of
    ; the right subtree, then the number of black nodes in every path T.left
    ; to an external leaf may be more than that of the right subtree.
    ;
    ; We are going to show an invariant that explains the following if-block
    ; and so by Invariant 1, if leftblacks > rightblacks, then we can set
    ; T.right.colour to BLACK in order to balance the number of black nodes
    ; from T to each external leaf-node.
    ;
    if (leftblacks > rightblacks),
       T.right.colour := BLACK
    endif.
    ;
    ; At this point, the number of black nodes in every path from T.right to
    ; an external leaf equals leftblacks.
    ;
    ; After colouring both subtrees, we need to colour T itself. If both of
    ; T.left.colour and T.right.colour equal BLACK, then we can color T to
    ; be a RED node in order to keep the return value minimal. The function
    ; then returns leftblacks as the correct return value.
    ;
    ; Otherwise, it is implied that T.left.colour or T.right.colour may be
    ; RED. In order to maintain the restrictions of red nodes we need to
    ; colour T to be black. The function then returns leftblacks+1 as the
    ; correct return value.
    ;
    ; {MARK 4}
    if (T.left.colour = T.right.colour = black),
       T.colour := RED
       return leftblacks.
    else,
       T.colour := BLACK
       return leftblacks+1.
    endif.
else,
    ;
    ; "Symmetric" procedure as above. We do recursion on the right subtree
    ; first, then colour the left subtree, and maintain the number of black
    ; nodes from each subtree to each of its leaf node (according to
    ; invariant 1).
    ;
    ; After that, we colour T itself, and the function returns the correct
    ; return value.
    ;
    ; {MARK 1}
    rightblacks := Colour(T.right)
    var leftblacks
```

```
        if T.left is NIL,
            T.left := CreateExternalBlackNode()
            leftblacks := 1
        else,
            ; {MARK 3}
            leftblacks := Colour(T.left)
        endif.
        if (rightblacks > leftblacks),
            T.left.colour := BLACK
        endif.
        if (T.right.colour = T.left.colour = black),
            T.colour := RED
            return rightblacks.
        else,
            T.colour := BLACK
            return rightblacks+1.
    endif.
end function.
```

Conventions: BH(T) denotes the black height of T. Heihgt(T) denotes the height of T.

(Claim 1) For all subtrees u, v of T (AVLxRed-Black Tree). * IF Height(u) ¿ Heihgt(v) * THEN BH(u) = BH(v) + 1.

Proof: Assume Height(u) ¿ Heihgt(v). By the balance property of AVL Tree, Height(u) = Height(v) + 1. Assume BH(u) is not BH(v) + 1. (for contradiction) We are in two cases. Assume BH(u) ¿ BH(v) + 1 This implies that H(u) ¿ H(v) + 1, because the height of a Black-Height-1 tree is at most 1. Note that the height of a Black-Height-1 tree cannot be 0 in this case, since it implies the node is an external node, which cannot be added into v. However, this is a contradiction since if H(u) ¿ H(v) + 1, then u and v are not balanced, since the balance factor will become +2 or -2. Assume BH(u) ¡ BH(v) + 1 Then Height(v) + 1 ¡ Heihgt(u). Same reason, the balance factor will become either +2 or -2. Therefore, BH(u) = BH(v) + 1.

(Invariant 1) For all subtrees u, v of T, and assuming Colour(p) works correctly for all descendants p of T, * IF Colour(T.u) ¿ Colour(T.v), * THEN T.v.colour=RED, Height(u) = Height(v)+1, and Colour(T.u)=Colour(T.v)+1.

Proof: Just a convention, the return value of Colour(T) satisfies the definition of black-height for Red-Black Tree, so they are equivalent. Assume BH(T.u) ¿ BH(T.v) Claim 1 implies BH(u) = BH(v) + 1. By the balance property of AVL Trees, Height(u) = Height(v)+1. So, we need to argue that T.v.colour=RED. Assume T.v.color=BLACK. By assumption, BH(T.v.left) = BH(T.v.right) = BH(T.v) + 1. BH(T.v.subtree) + 2 = BH(T.u). Height(T.v.subtree) + 2 = Height(T.u). According to the code segment at MARK 4, one of the subtree.colour must be RED. Let such subtree be T.p. Note that since we have coloured the subtrees of T.v, then their balance factor are equal. Since T.v.p.colour is RED, then the nodes of T.v.p must be BLACK... let such node be T.v.p.q. BH(T.v.p.q) = BH(T.v) - 1 = BH(T.u) - 2. H(T.v.p.q) = H(T.v) + 1 = H(T.v.p) + 2 = H(T.v.p.q) + 3. However, there does not exist a black-height-2 AVL Tree that has height 3 *according to our procedure*, since the root node of a height 3 black-height-2 AVL tree must be RED, but by assumption it must be BLACK. (Imagine putting in on top of T.v.p.q.) This is a contradiction. Therefore, T.v.color=RED.

(Claim 2) AVLToRed-Black(T) Terminates, returns the correct result and runs in O(n), where n is the size of T.

Proof: This algorithm depends on Colour(T). When Colour(T) works correctly and terminates, the following procedure is to make the root node black, and thus if Colour works correctly and terminates, the requirement will be satisfied. For the recursive function Colour(T), the correctness of this algorithm is explained on the in-line comments, along with invariant 1. This function terminates because each recursive call at MARK 1 and MARK 3, assuming the currently visiting subtree is T, will go to the

subtree of T, and will eventually reach base case. Note that except the recursive calls, all other operations take constant time. Also note that each node is visited by recursive-calls exactly once. Therefore, the runtime complexity of this algorithm is O(n), where n is the number of nodes in T aka the size of T.

2 Question 2.

```
Convention:
       int colour = "Total number of color inside the whole tree"

    int x.color = "The color of x."
    int x.key = "The key of x."
    int *x.left = "The left subtree of x."
    int *x.right = "The right subtree of x."

Specifications:
    Each node x contains the following augmented data:
    int x.mind = "The minimum distance of two multicolor pair in the tree
    rooted at x. Otherwise, the value is null."
    int *x.a = "Pointer pointed to the tree node having the largest key in the
    tree rooted at x. This is pointed to x when x is a leaf node."
    int *x.b = "If x.a is not x, then this is pointed to the tree node having
    the largest key in the tree rooted at x satisfying x.b.color not
    equal x.a. Otherwise, the value is null."
    int *x.c = "Pointer to the tree node having the smallest key in the tree
    rooted at x. This is x when x is a leaf node."
    int *x.d = "If x.a is not x, then this is the tree node having the smallest
    key in the tree rooted at x satisfying x.d.color does not equal x.c.
    Otherwise, point to null."


;
; Returns the element at A[i], where A[i] is not null and A[i].color is not
; x.color. i is the minimum possible index.
; Returns null otherwise.
; Since we need to check every element of A, the time complexity of
; this function is O(n), where n is the size of A.
;
node CN(x, A):
    for i in A[1 ... i],
        if A[i] is not null and A[i].color is not x.color,
            return A[i]
        end if.
    end for.
    return null.
end function.

;
; Returns the distance from x to at A[i], where A[i] is not null and
; A[i].color is not x.color. i is the minimum possible index.
; Returns null otherwise.
; Since we need to check every element of A, the time complexity of
; this function is O(n), where n is the size of A.
;
int CV(x, A):
    for i in A[1 ... i],
        if A[i] is not null and A[i].color is not x.color,
```

```
            return A[i].key - x.key
        end if.
    end for.
    return null.
end function.

;
; Returns the minimum number in A.
; For the minimum index i, if A[i] is null, then A[i] is ignored.
; If all elements in A is null, then this function returns null.
; Since we need to check every element of A, the time complexity of
; this function is O(n), where n is the size of A.
;
int MIN(A);

;
; Recover the augmented data for x.
; Assume all subtrees of x have correct augmented data.
; Assume any illegal access is null.
;
Recovery(x):
    if x is leaf,
        ;
        ; If x is leaf, then we are doing the right thing.
        ;
        ; {MARK 1}
        x.a := x
        x.b := null
        x.c := x
        x.d := null
        x.mind := null
    else,
        ;
        ; First, we want to check the nullity of subtrees.
        ; If the each of the subtree is null, then the corresponding augmented
        ; data points to x itself.
        ;
        ; {MARK 2}
        if x.right is null,
            x.a := x
        else,
            x.a := x.right.a
        end if.
        if x.left is null,
            x.c := x
        else,
            x.c := x.left.a
        end if.
        ;
        ; For x.b and x.d, we don't want the color of them equals the color of
        ; x.a and x.c respectively.
        ; According to the specification of each augmented datum, we put the
        ; "most preferable" item at the beginning of the arrays, and the
        ; "least preferable" item near the end of the arrays.
        ; Since by assumption, the subtrees of x are corrected augmented, then
        ; essentially the two assignments below will cover all the possible
        ; choices of data.
        ;
```

```
        ; {MARK 3}
        x.b := CN(x.a,
            { x.right.b, x,
                x.left.a, x.left.b }
            )
        x.d := CN(x.c,
            { x.left.d, x,
                x.right.c, x.right.d }
            )
        ;
        ; We still need recover 1 more data.
        ; The shortest distance between the key of x is computed by
        ; min {x.key - p.key, q.key - x.key}, where p contains the minimum key
        ; on the right subtree, and q contains the maximum key on the right
        ; subtree.
        ; However, we need the multicolor distance. So, by specification, the
        ; color of b and d are different from the color of a and c respectively
        ; for each tree, then we just put more arguments in the array being
        ; passed to CN.
        ; Because by specification, there exists the case that the computed
        ; distance is smaller than one of the mind value inside subtrees of x,
        ; then we also put them in the array being passed to MIN.
        ;
        ; {MARK 4}
        x.mind := MIN(
            { CV(x, {x.right.c, x.right.d} ),
                CV(x, {x.left.a, x.left.b} ),
                x.left.mind,
                x.right.mind }
            )
    end if.
end procedure.


(Claim 1)
    The function Recovery(x) runs in O(1).
```

Proof: First, this is not a recursive function, since there's no calls to recursive functions. At MARK 1 and MARK 2, it is obvious that Revocery(x) runs in constant time. From MARK 3 to the end of Recovery(x) function, every function call depends on the input size of the input array. For furthermore description of CN and CV, see the function-header-documentation. Since the sizes of arrays are fixed for all function calls, we conclude that the the function Recovery(x) runs in O(1).

(Claim 2) The function Recovery(x) is correct.

Proof: Claim 1 implies Recovery(x) terminates. For partial correctness, see the in-line comment. Therefore, Recovery(x) is correct.

How do we maintain data? Note that we have to keep track of the total number of colors. For insert... We want to insert the node first, then recover data from bottom-up. Since the augmented data after insertion have no effect on the sibling subtrees, then all the sibling subtrees does not need to change at all. If the function call-back requires rotation, we do the rotation first, then recover data furthermore. See below. For delete... We want to delete the node first. If the node j being deleted happens to be a non-leaf node, then we want to take the left-most node i on the right subtree and put it at the removed position j. Then, we recover data from the parent of i, anaologously a bottom-up process. If rotation is needed, we do the rotation first, then recover data furthermore. See below. If there's a rotation... We want to use a Deque to store all nodes that the subtree(s) is/are changed to a different one. By the property of AVL Tree, the number of such tree nodes will not exceed 3. We want to push the eventually-higher tree

nodes at the beginning of the Deque, and offer the eventually-lower tree nodes at the end of the Deque. After rotation, we pop the nodes from the Deque from beginning to the end, and recover data in order.

(Claim 3) Insert, Delete runs in O(log n). Proof: Using the theorem of augmented data-structure, since all the augmented data can be recovered in constant time, Insert and Delete then runs in O(1).

Other operations: For find color... Regular binary tree search. For closest mc pair... See below.

```
;
; Pseudocode for ClosestMCPair(S).
;
{p, q} ClosestMCPair(S):
    if fewer than two elements or less than two colors:
        ERROR.
    else,
        ;
        ; See the documentation at {MARK 4}, inside function Recovery(x).
        ; These function calls are all in O(1), since the input sizes are fixed.
        ;
        x := S.root
        r1 := CV(x, {x.right.c} )
        r2 := CV(x, {x.right.d} )
        l1 := CV(x, {x.left.a} )
        l2 := CV(x, {x.left.b} )
        gl := x.left.mind
        gr := x.right.mind
        ;
        ; For the following switch statement, case r1 r2 l1 l2 are very
        ; obvious.
        ; For the case gl, since there exist *closer* nodes on the left subtree,
        ; then we shall do a recursive call to determine the node.
        ; Same reasoning for case gr.
        ;
        switch MIN({l1, l2, r1, r2, gl, gr}):
            case: r1
                return {x.right.c, x}
            end case.
            case: r2
                return {x.right.d, x}
            end case.
            case: l1
                return {x.left.a, x}
            end case.
            case: l2
                return {x.left.b, x}
            end case.
            case: gl
                return ClosestMCPair(x.left)
            end case.
            case: gr
                return ClosestMCPair(x.right)
            end case.
            default:
                ;
                ; There wouldn't be any default cases, since S is not a leaf and
                ; contains more than 1 color.
                ;
                ERROR.
        end switch.
```

```
    end if.
end function.
```

(Claim 4) ClosestMCPair(S) runs in O(log n), where n is the size of S. Proof: The runtime complexity really depends on the recursive calls, since besides those calls all other operations run in constant time. For the recursive call, since the path goes to either left or right subtree, then the runtime complexity depends on the height. Since S is an AVL tree, then the runtime complexity of this algorithm is O(log n).

(Claim 5) ClosestMCPair(S) is correct.

Proof: Claim 4 implies ClosestMCPair(S) terminates. For the partial correctness, see the in-line comment. Therefore, ClosestMCPair(S) is correct.

Note: this algorithm satisfies all requirements.