DS2030 DSA for DS
Quiz 1: 50 minutes

Name:

Roll Number:

Instructions

- Answer all the questions.
- Write your answers directly on the quiz paper in the spaces provided.
- **No additional sheets or main sheet will be provided**.
- The total marks for the quiz is **30 points**. Allocate your time wisely.
- Describe the algorithms using the pseudo-code notation used in the class or Python syntax.
- *You are **not** allowed to use Python list operations and other Python data structures/operations unless explicitly stated.*

1 True or False (6 points)

1. State True or False. There is no need for explanations. Incorrect answer receives -0.5 penalty.

   (a) Inserting n elements into a dynamic array that doubles in size when full, takes $O(n)$ time. True

   (b) Given the number of nodes n, any one of the standard tree traversals (such as preorder, inorder, or postorder) is sufficient to uniquely reconstruct a complete binary tree. True

   (c) If the top of the stack is the head of a singly linked list, the pop operation takes $O(1)$ time. True

   (d) The minimum height of a binary search tree storing n keys is $O(\log n)$. True

   (e) There exists a binary search tree storing n keys that is shorter than the corresponding AVL tree storing the same keys. True

   (f) The inorder traversal of a binary min-heap always yields the elements in sorted (non-decreasing) order. False

2 Short Answers (14 points)

1. Write the algorithm for UpHeap(A, i) for a heap that is implemented using an array; where A is the array and i is the index of the newly inserted element. (3 points)

Algorithm 1 UpHeap(A, i)

```
Require: Array $A[0 \dots n]$ representing a binary min-heap, index $i$ of the newly inserted element
1: while $i > 0$ & $A[\lfloor i/2 \rfloor] > A[i]$ do
2:     $p \leftarrow \lfloor i/2 \rfloor$
3:     $temp \leftarrow A[i]$
4:     $A[i] \leftarrow A[p]$
5:     $A[p] \leftarrow temp$
6:     $i \leftarrow p$
7: end while
```

Algorithm 2 MysteryAlgorithm(n)

```
$count \leftarrow 0$
$i \leftarrow 1$
while $i < n$ do
    $j \leftarrow 0$
    while $j < i$ do
        $count \leftarrow count + 1$
        $j \leftarrow j + 1$
    end while
    $i \leftarrow i \times 2$
end while
return $count$
```

2. Compute the running complexity of the following algorithm (3 points)

   $O(n)$

3. Give the pseudo-code for post-order traversal of a binary tree that is implemented using a Stack and **without** any recursion. Ensure that each node is visited exactly once. Use only standard operations on the stack (push, pop, top, isEmpty). You may use an auxiliary variable to track the last visited node, if necessary. (4 points)

Algorithm 3 postorder_traversal($root$)

```
Require: A binary tree with root node $root$
$stack \leftarrow$ empty stack
$lastVisited \leftarrow$ null
$current \leftarrow root$
while stack is not empty or current $\neq$ null do
    if current $\neq$ null then
        push current onto stack
        current $\leftarrow$ current.left
    else
        peek $\leftarrow$ top of stack
        if peek.right $\neq$ null and lastVisited $\neq$ peek.right then
            current $\leftarrow$ peek.right
        else
            visit(peek)
            lastVisited $\leftarrow$ peek
            pop from stack
        end if
    end if
end while
```

4. Given a Queue containing n elements and all the queue operations (enqueue, dequeue, isEmpty, front), write a pseudocode function that uses recursion to print the elements of the queue in the reverse order such that (1) the elements are printed in reverse of their original order; (2) the original order of the elements in the queue is restored after the function executes; and (3) you **cannot** use any explicit loops, or auxiliary data structure such as stacks or arrays. (4 points)

Algorithm 4 printReverse(Q)

```
Require: A queue $Q$ with $n$ elements and operations: enqueue, dequeue, isEmpty, front
function PRINTREVERSE(Q)
    if isEmpty(Q) then
        return
    end if
    $x \leftarrow$ front(Q)
    dequeue(Q)
    PRINTREVERSE(Q)
    print(x)
    enqueue(Q, x)
end function
```

3 Self-Balancing Trees(10 points)

We will complete the pseudo-code for insertion and deletion in an AVL Tree continuing from the psuedocode for left and right rotations that was discussed in the class. Assume that LeftRotate(node) and RightRotate(node) functions are correctly implemented and return the root of the subtree after the rotations. Each AVLTree node has the following attributes - parent, left, right, key, height, balance. Provide the pseudocode for the following fuctions.

1. UpdateHeight(node) - updates the height of a node. (1 point)

Algorithm 5 UpdateHeight(node)

```
function UpdateHeight(node)
    $h_{\text{left}} \leftarrow$ if node.left $\neq$ null then node.left.height else $-1$
    $h_{\text{right}} \leftarrow$ if node.right $\neq$ null then node.right.height else $-1$
    node.height $\leftarrow \max(h_{\text{left}}, h_{\text{right}}) + 1$
end function
```

2. UpdateBalance(node) - update the balance factor of a node. (1 point)

Algorithm 6 UpdateBalance(node)

```
function UpdateBalance(node)
    if node.left $\neq$ null then
        $h_{\text{left}} \leftarrow$ node.left.height
    else
        $h_{\text{left}} \leftarrow -1$
    end if
    if node.right $\neq$ null then
        $h_{\text{right}} \leftarrow$ node.right.height
    else
        $h_{\text{right}} \leftarrow -1$
    end if
    node.balance $\leftarrow h_{\text{left}} - h_{\text{right}}$
end function
```

3. AVLInsert(node, key) - Insert the key into the AVL tree and rebalance if necessary. The function returns the root of the subtree after the insertion/rebalancing (4 points)

Algorithm 7 AVLInsert(node, key)

```
function AVLINSERT(node, key)
    if node = null then
        return CREATENEWNODE(key)                ▷ new node has height 0
    else if key < node.key then
        node.left ← AVLINSERT(node.left, key)
        node.left.parent ← node
    else
        node.right ← AVLINSERT(node.right, key)
        node.right.parent ← node
    end if
    UPDATEHEIGHT(node)
    UPDATEBALANCE(node)
    if node.balance > 1 then
        if key < node.left.key then
            return RIGHTROTATE(node)
        else
            node.left ← LEFTROTATE(node.left)
            return RIGHTROTATE(node)
        end if
    else if node.balance < -1 then
        if key > node.right.key then
            return LEFTROTATE(node)
        else
            node.right ← RIGHTROTATE(node.right)
            return LEFTROTATE(node)
        end if
    end if
    return node
end function
```

4. AVLDelete (node, key) - Delete the key from the AVL tree and rebalance if necessary. The function returns the root of the subtree after the deletion/rebalancing(4 points)

Algorithm 8 AVLDelete(node, key)

```
function AVLDelete(node, key)
    if node = null then
        return null
    else if key < node.key then
        node.left ← AVLDelete(node.left, key)
    else if key > node.key then
        node.right ← AVLDelete(node.right, key)
    else
        if node.left = null  &  node.right = null then
            return null
        else if node.left = null then
            temp ← node.right
            temp.parent ← node.parent
            return temp
        else if node.right = null then
            temp ← node.left
            temp.parent ← node.parent
            return temp
        else
            successor ← MinValueNode(node.right)
            node.key ← successor.key
            node.right ← AVLDelete(node.right, successor.key)
        end if
    end if
    UpdateHeight(node)
    UpdateBalance(node)
    if node.balance > 1 then
        if node.left.balance ≥ 0 then
            return RightRotate(node)
        else
            node.left ← LeftRotate(node.left)
            return RightRotate(node)
        end if
    else if node.balance < -1 then
        if node.right.balance ≤ 0 then
            return LeftRotate(node)
        else
            node.right ← RightRotate(node.right)
            return LeftRotate(node)
        end if
    end if
    return node
end function
```

Use for rough work.