1. Questions on Big-O Analysis. (25 marks)

1. Consider a polynomial function of order 𝑘 given by 𝑓(𝑛) = 𝑎k𝑛k + 𝑎k-1𝑛k-1 + ∙∙∙ + 𝑎0. Formally demonstrate that 𝑓(𝑛) ∈ Ο(𝑛k). Full marks for using basic definitions and concepts.

[12 marks]

**Big-O Rules: Drop smaller terms**

If f(n) = (1 + h(n)) with h(n) -> 0 as n -> ∞. Then, f(n) is O (1).

**Big-O Multiplication Rules**

Suppose two equations 𝑓1(𝑛) is Ο(𝑔1(𝑛)) and 𝑓2(𝑛) is Ο(𝑔2(𝑛)). From the definition, there exists positive constants c1, c2, n1 and n2 such that 𝑓1(𝑛) ≤ c1𝑔1(𝑛) for all n ≥ n1 and 𝑓2(𝑛) ≤ c2𝑔2(𝑛) for all n ≥ n2. Let n0 = max {n1, n2}. Multiplying 𝑓1(𝑛) and 𝑓2(𝑛) gives 𝑓1(𝑛) 𝑓2(𝑛) ≤ c1c2𝑔1(𝑛)𝑔2(𝑛) for all n ≥ n0. So, 𝑓1(𝑛)𝑓2(𝑛) is O(𝑔1(𝑛)𝑔2(𝑛)).

**Big-O Conventions**

Use the smallest (slowest growing) ‘reasonable’ possible class of functions. As an example, 2n is O(n) instead of O(n2)

Use the simplest expression of the class. As an example, 3n + 5 is O(n) instead of O(3n)

𝑓(𝑛) = 𝑎k𝑛k + 𝑎k-1𝑛k-1 + ∙∙∙ + 𝑎0

𝑓(𝑛) = nk (𝑎k+ 𝑎k-1/n+ ∙∙∙ + 𝑎0/nk)

As n -> ∞, (𝑎k+ 𝑎k-1/n+ ∙∙∙ + 𝑎0/nk) -> (𝑎k+ 0 + ∙∙∙ + 0) -> ak and 𝑓(𝑛) -> nkak by Drop smaller terms.

O(nk) is trivially O(nk) and O(ak) is O(1).

With Big-O Multiplication Rules, O(𝑓(𝑛)) = O(nkak) = O(nk) O(ak) = O(nk)O(1) = O(nk\*1) = O(nk)

So, 𝑓(𝑛) is O(nk) or 𝑓(𝑛) ∈ Ο(𝑛k).

1. Refer to the definition of Big-O in the lecture materials. In particular, the condition for which one can state that 𝑓(𝑛) is Ο(𝑔(𝑛)) is defined. Briefly explain why the notation 𝑓(𝑛) ∈ Ο(𝑔(𝑛)) is preferred compared to 𝑓(𝑛) = Ο(𝑔(𝑛)). Full marks for using basic definitions and concepts and mathematical formulation.

[13 marks]

𝑓(𝑛) is Ο(𝑔(𝑛)) if and only if there exists positive constants c and n0 such that 𝑓(𝑛) ≤ c𝑔(𝑛) for all n ≥ n0. In mathematical notation, ∃c > 0, n0 such that ∀n ≥ n0, 𝑓(𝑛) ≤ c𝑔(𝑛).

f: N+ -> R+ and g: N+ -> R+ where N+ = {1,2,3, …} and R+ = {x ∈ R| x ≥ 0} assuming that f(n) ≥ 0, ∀n ≥ 1. For convenience, the function is sometimes relaxed to f(n) ≥ 0, ∀n ≥ N for some constant N.

O(n) can be thought of as the set of all functions whose growth is no worse than linear for sufficiently large n. Hence, it can be thought of as the infinite set {1, 2, …, log n, 2 log n, …, n, 2n, 3n, …, n+1, n+2, …}. So, 3n+5 is O(n) is just the statement that 3n+5 is in this set or 3n+5 ∈ O(n).

We should avoid writing big-O notations in the form 𝑓(𝑛) = Ο(𝑔(𝑛)). As an example, if 𝑓(𝑛) is Ο(n), then f(n) is also Ο(n2) because if f(n) grows no worse than linear for sufficiently large n, then f(n) must also grow no worse than quadratic for sufficiently large n. If we write 𝑓(𝑛) = Ο(n) and f(n) = Ο(n2). Then this implies that Ο(n) = Ο(n2) which is incorrect as Ο(n) ⊂ Ο(n2). Instead, we should write 𝑓(𝑛) ∈ Ο(n) and f(n) ∈ Ο(n2).

2. Questions on Binary Search Tree, Heap, Balanced Binary Search Tree, Basic Data Structures (stack/queue)

(a)

startOfFile

TreeNode class implementation:

int val

TreeNode left

TreeNode right

In the TreeNode constructor (int x)

set val equals x

End constructor

In main method

arrayToBST arrayToBST = new arrayToBST

TreeNode root

TreeNode key

int array = *{//sample array*};

set root equals *to* arrayToBST method (array)

Print out "Preorder traversal of constructed BST "

call Preorder method (root)

create Scanner object

Print out "Choose a number to search it in the tree "

int num = number entered by user

set key equals *to* search method (root , num)

If key equals null

Print out "Number not found in the tree "

Else

Print out "Number is found " + the number

End main method

END TreeNode implementation

arrayToBST class implementation:

In method -type-TreeNode arrayToBST(int array)

If array length equals 0

return Null

Else

return treeFromArray(array, 0, array.length - 1)

END arrayToBST method

In method -type-TreeNode treeFromArray(int array, firstIndex, lastIndex)

If firstIndex > lastIndex

return Null

Else

int midpoint = firstIndex + (lastIndex-firstIndex) / 2

-type-TreeNode node = new TreeNode(array[midpoint])

node.left equals recursive call treeFromArray(arr,firstIndex,midpoint-1)

node.right equals recursive call treeFromArray(arr,midpoint+1, lastIndex)

return node

END treeFromArray method

In Preorder method (-type-TreeNode node )

If node equals Null

return

print out node.val + " "

recursive call Preorder(node.left)

recursive call Preorder(node.right)

END Preorder method

In method -type-TreeNode search (-type-TreeNode root, int number )

If root equals Null OR root.val equals number

return root

If root.val is LESS THAN number

return recursive method search(root.right , number)

return recursive method search (root.left , number)

END search method

END arrayToBST implementation endOfFile

By successfully constructing a Binary Search Tree from a sorted array, We successfully get O(log n) as time complexity when searching up any element in the tree. That is because every time we search in the tree, we are halving the number of inputs in each search iteration (recursion).

(b)

The conditions given can be satisfies by using an AVL Search Tree structure for the software. When doing insertion and Deletion, the process time for an AVL Search Tree will always be O(log n), the procedure for Insertion such as follows:

1. Search through the tree for empty node (Search right node if current node alphabetical order is smaller than insert value alphabetical order, else search left node)

2. When empty node is found, insert the new value

3. Check the balance factor of every single node

4. Case 1: If Balance factor is >=-1 and <=1, the insert operation ends

5. Case 2: If Balance factor doesn’t satisfy the condition in Case 1, rebalance the tree using rotation

The procedure for deletion is similar to Insertion, which is as follows:

1. Search through the tree for the value to be deleted

2. When the value to be deleted is found, set the node to null and replace with suitable node

3. Check the balance factor of every single node

4. Case 1: If Balance factor is >=-1 and <=1, the insert operation ends

5. Case 2: If Balance factor doesn’t satisfy the condition in Case 1, rebalance the tree using rotation

Since the AVL tree for employee names is constructed in alphabetical order, the software can finish the request in O(n) time.

Lastly, the promises can be declared as a global variable which connect to each node so the software is able to process it in O(1) time and everyone will be getting the same value.

**Pseudocode:**

Insertion (NewEmployee, EmployeeList)

SET CurrentNode = EmployeeList.RootNode

WHILE (CurrentNode != null)

IF (CurrentNode.AlphabeticalOrder < NewEmployee. AlphabeticalOrder)

SET CurrentNode = CurrentNode.right

ELSE SET CurrentNode = CurrentNode.left

END WHILE

Insert(NewEmployee)

Balance(EmployeeList)

ListEmployee(EmployeeList)

SET CurrentNode = EmployeeList.RootNode

InOrder (CurrentNode)

IF CurrentNode == null

RETURN null

InOrder (CurrentNode. left)

RETURN CurrentNode.EmployeeName

InOrder (CurrentNode. right)

Deletion(ExisitingEmployee, EmployeeList)

SET CurrentNode = EmployeeListRootNode

WHILE (CurrentNode != ExisitingEmployee)

IF (CurrentNode.AlphabeticalOrder < ExisitingEmployee. AlphabeticalOrder)

SET CurrentNode = CurrentNode.right

ELSE SET CurrentNode = CurrentNode.left

END WHILE

Remove(CurrentNode)

IF (CurrentNode.left != null && CurrentNode.right != null)

SET Successor = findMinNode(CurrentNode.right)

CurrentNode = Successor

CALL Deletion(Successor, EmployeeList)

ELSE IF (CurrentNode.left != null || CurrentNode.right != null)

IF (CurrentNode.right != null)

SET CurrentNode = CurrentNode.right

CALL Deletion(CurrentNode.right, EmployeeList)

ELSE

SET CurrentNode = CurrentNode.left

CALL Deletion(CurrentNode.left, EmployeeList)

Balance(EmployeeList)

findMinNode(Node)

SET CurrentNode = EmployeeListRootNode

WHILE (CurrentNode.left != null)

CurrentNode = CurrentNode.left

RETURN CurrentNode

(c)

For upgrade requests and cancellation to be in O(log n) time and determine the k highest-priority flyers on the waiting list in O(k log n) time, where n is the number of frequent flyers on the waiting list, the system must traverse nodes to a depth of at most O(log n). For this to be possible, we require the use of a self-balancing binary tree which has a depth of O(log n). Two of the most popular self-balancing binary trees are AVL tree and Red-Black tree. In this instance, we decided to use a Red-Black tree. This is because of program will experience frequent insertion and deletion. Although AVL trees are more balanced, they cause more rotations during insertion and deletion. So, a Red-Black tree is preferred in this case.

**Rules That Every Red-Black Tree Follows:**

1. Every node has a color either red or black.
2. The root of the tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendants’ NULL nodes has the same number of black nodes.
5. All leaf nodes are black nodes.

**Interesting points about Red-Black Tree:**

1. Black height of the red-black tree is the number of black nodes on a path from the root node to a leaf node. Leaf nodes are also counted as black nodes. So, a red-black tree of height h has black height >= h/2.
2. Height of a red-black tree with n nodes is h<= 2 log2(n + 1).
3. All leaves (NIL) are black.
4. Text

   Description automatically generatedDiagram

   Description automatically generatedThe black depth of a node is defined as the number of black nodes from the root to that node.

Text

Description automatically generatedText

Description automatically generated

This is the pseudocode for the default search element and add element methods of a red-black tree.

Text

Description automatically generatedText

Description automatically generated

This is the pseudocode for the default delete element method of a red-black tree.

For a program using a red-black tree, upgrade and cancellations thus will take O(log n) time. This is because the program will have to traverse the tree to find the correct position and insert or delete a node. This process takes O(log n) time because the red-black tree has a depth of O(log n). To determine the *k* highest-priority flyers on the waiting list, the process takes O(k log n) time. This is because the red-black tree has a depth of O(log n) and the program has to traverse the red-black tree k time. So, O(k \* log n) = O(k log n). A Java program has been created below using TreeSet which is implemented using a Red-Black Tree. The comparator of the program is designed to move the highest priority passengers to the left of the tree. To get the *k* highest-priority flyers on the waiting list, we get the leftmost passenger in the tree first.

Text

Description automatically generated

This is the public class for our program named FrequentFlyerProgram. The constructor initializes a new TreeSet using the Passenger comparator. TreeSet uses Red-Black Tree in its implementation. The referenceID is also set to 0.

Text

Description automatically generated

This is the Passenger class and it holds information about the passenger.

Passenger() – Used to initialize the comparator in the TreeSet

The constructor with the 6 parameters creates a new Passenger object with the given parameters which is saved inside the TreeSet.

Text

Description automatically generated

This is the comparator for the Passenger class. It is used to compare 2 passenger objects inside the TreeSet. It is used in the search, insert and delete functions of the TreeSet. First, we compare rank, higher rank goes left. If ranks are equal, we compare time, higher time goes left. If rank and time are equal, we compare ref number, lower ref number goes left.

Text

Description automatically generated

This is the getPassengerList method which returns k passengers with the highest priority.

This method returns the current referenceID and increments it by 1 so every passenger has a unique ID. This is to allow passengers with the same rank and waiting time to exist together in the Red-Black Tree.

Text

Description automatically generated

This is the addPassenger method which creates new Passenger object and inserts it into the TreeSet. A unique comfirmation code is also generated for each passenger. As an example, S20110 means the passenger has Silver rank, referenceID of 01 and waitingTime of 10. The new passenger is only inserted if no duplicates exist in the TreeSet.

Text

Description automatically generated

This is the removePassenger method which removes a passenger object from the TreeSet. It converts the confirmationCode of the passenger into rank, referenceID and waitingTime which is used to create a temporary passenger object. The passenger is deleted from the TreeSet if a duplicate is found.

Text

Description automatically generatedText

Description automatically generatedText

Description automatically generated

These are the pseudocode for the upgrade, cancellation and get k highest priority flyer methods of the program.