Romil V. Shah

20008692

Assignment 2  
CS/CPE 600

Prof. Reza Peyrovian

Submission Date: 09/25/2022

Q1. No. 2.5.13



Ans.

Let’s take 2 Queues, Q1 and Q2 for push() operation, check that the Queue Q1 is empty.

If the queue is empty, start enqueueing elements, for this instance we insert, 0,1 and 2.

Now, we have to the pop() an element and as per the rule 2 should pop out first.

For this, we will take the Queue Q2 and start the enqueueing of the elements from Q1, we continue the operation till Queue Q1 has just 1 element i.e., 2.

Now that, only 2 is left in Queue Q1, we dequeue it.

Now, we want to insert element 3. So. We enqueue it in Queue 2.

This is how we perform push() and pop() operations in a stack. By this, we can say that a stack can be implemented using 2 queues.

The running time for enqueue and dequeue operation takes O(1) time.

So, the running time for push() and pop() operations will also be O(1) time.

Q.2 No. 2.5.20



Ans.

We can develop a recursion method to print the depth of each node.

When the current node is the root of the tree, the depth is 0.

Then, we can print its children’s depth which is 1.

Then we can print their children’s depth which is 2.

And so on, ……

With the recursion method, we can print the depth of the whole nodes.

**Algorithm:**

depthOfTree\_T(T,v):

**if** T.isRoot(v) **then**

return 0

**else**

**d = 0**

**for** each w which is a child of v **do**

d = 1 + depth(T, T.parent(v))

return d

The run time of this algorithm will be O(n) where n is the given number of nodes in a tree.

Q.3 (No. 2.5.32)

Text

Description automatically generated

Ans.

To find the lowest common ancestor between two nodes *x, y* of a tree *T*, first, consider root node and start traversing it. If given values *x, y* of a node matches then root is the lowest common ancestor. If *x, y* doesn’t match, call recursively for lowest common ancestor algorithm for the left subtree and right subtree. If *x, y* present as the left child and right child then the parent node is lca, if *x, y* present in the left subtree then lca is from left subtree vice versa with right subtree.

**Algorithm**:

LowestCommonAncestor(BinaryTree x, BinaryTree y, BinaryTree root):

Input: A BinaryTree node root T, a node x and a node y

Output: The lowest common ancestor of x and y

If root is null or x is root or y is root:

Return root

BinaryTree leftNode <- LowestCommonAncestor(x, y, root.left)

BinaryTree rightNode <- LowestCommonAncestor(x, y, root.right)

If leftNode = null:

Return rightNode

Else if rightNode = null:

Return leftNode

Return root

The running time of the algorithm is O(n) where n is the height of the tree.

Q4. No. 3.6.15



Ans.

To find kth smallest key in the union of keys from S and T where S and T are ordered arrays, each with n items.

Firstly, examine the k/2 element in the array list S.

Now analyze largest element in the T which is less than k/2 by binary search.

Now adding the indices of these 2 elements:

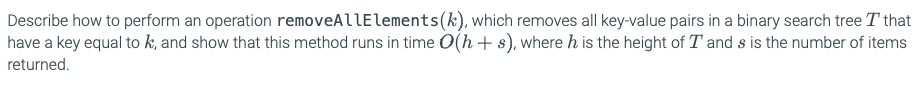
* if sum of them is equal i.e., k then take maximum of two elements.
* If sum > k, binary search is performed to the right of S.
* If sum < k, binary search is performed to the left of S.

Now same operations are performed on T based on the largest element but less than the current element in S.

Now calculating total time complexity for the process, performing binary search for two arrays S and T will take O(log n) and O(log n) respectively. That is O(log2 n).

After solving this will give O(log n) running time complexity for the whole process.

Q5. No. 3.6.19



Ans.

To remove all thenodes from a binary search tree, perform post order traversal on the tree and recursively call left subtree and right subtree and free the nodes respectively.

Algorithm:

removeAllElements(target, root):

**Input:** A search key k for node of a binary search tree T.

**Output:** Empty binary search tree

**if** T(k, T.root()) is null

**return** null

**else**

removeAllElements( binaryPostorder(T, T.leftChild(k)))

removeAllElements(binaryPostorder(T, T.rightChild(k)))

perform the “free” action for key (k) node

// (binaryPostorder algorithm is defined in Chapter 2 - Figure(2.4.12))

We spend O(h) time to find the node that equals to the target.

And then we spend s to remove all the duplicate. So, the total time complexity is O(h + s)

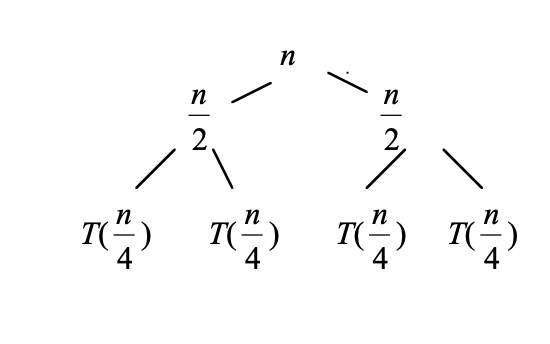
Q6. No. 3.6.26

Text

Description automatically generated

Ans.

Appling divide and conquer method (Binary search) to store a drug in the smallest bottle in the inventory hold xi millimeters. As sorted in an array T by capacities the smallest bottle will be find in the left whereas the largest in the right.



The above approach will give the recurrence relation:

T(n) = T(n/2) + c

Solving this recurrence relation using iteration method

T(n) = (c + c + T(n/4))

After solving will give

T(n) = k\*c + T(n/2k)

T(n) = c log n

For n requests the time complexity of the above method is c log n but we must process drug order of k requests will change the time complexity to O(k log n).

Since the order is already sorted for xi it will take less time to search for k requests.

After every xi  which changes the complexity to O(k log n/k).

Q7. No. 4.7.15



Ans.

Chart, bubble chart

Description automatically generated

Q8. No. 4.7.22

Graphical user interface, text, application, email

Description automatically generated

Ans.

**Given**: F0 =0 and F1 = 1, and Fk= Fk−1 + Fk−2, for k ≥ 2

**Proof:** Fk ≥ φk−2,

Base Case: for k=2

F2 = F1 + F0 = 1

Fk ≥ φk−2 => φ0 = 1

Therefore, true for k = 2

Base Case: for k=2

F3 = F2 + F1 = 2

Fk ≥ φk−2 => φ

Fk > φ

Therefore, true for k = 3

Now check for k-1

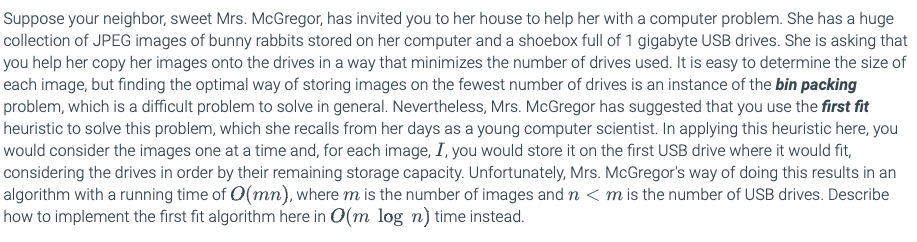
Fk−1 = Fk-2 + Fk−3 > φk−4 + φk−5 = = = (Given: φ2 =φ+1 )

Fk−1 ≥ φk−3 (Assumption true for k-1)

So, it will be true for Fk ≥ φk−2

Hence, we can say that, for k >= 3, Fk >= φk−2

Q9. No. 4.7.47



Ans.

Bin Packing Problem: Objects of different volumes must be packed into a finite number of bins or containers each of volume V in a way that minimizes the number of bins used.

First fit heuristic: The algorithm processes the items in arbitrary order. For each item, it attempts to place the item in the first bin that can accommodate the item. If no bin is found, it opens a new bin and puts the item within the new bin.

With the help of balancing search trees (AVL tree) we can minimize the running time complexity of storing the images into the hard drives from O(mn) to O(m log n).

Where m is the number of images and n < m is the number of USB drives.

While performing insertion operation in AVL tree takes O(log n) for n items and checking inserting images in an order to check all the m drives if any space left in the previous bins (according to First fit) takes m running time. So total running time complexity for first fit is O(m log n).