Romil V. Shah
20008692

Assignment 2
CS/CPE 600

Prof. Reza Peyrovian
Submission Date: 09/25/2022

Q1. No. 2.5.13

Describe how to implement a stack using two queues. What is the running time of the `push()` and `pop()` methods in this case?

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

Give an $O(n)$-time algorithm for computing the depth of all the nodes of a tree T, where n is the number of nodes of T.

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)

Suppose you work for a company, iPuritan.com, that has strict rules for when two employees, x and y, may date one another, requiring approval from their lowest-level common supervisor. The employees at iPuritan.com are organized in a tree, T, such that each node in T corresponds to an employee and each employee, z, is considered a supervisor for all of the employees in the subtree of T rooted at z (including z itself). The lowest-level common supervisor for x and y is the employee lowest in the organizational chart, T, that is a supervisor for both x and y. Thus, to find a lowest-level common supervisor for the two employees, x and y, you need to find the ***lowest common ancestor*** (LCA) between the two nodes for x and y, which is the lowest node in T that has both x and y as descendants (where we allow a node to be a descendant of itself). Given the nodes corresponding to the two employees x and y, describe an efficient algorithm for finding the supervisor who may approve whether x and y may date each other, that is, the LCA of x and y in T. What is the running time of your method?

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

Let S and T be two ordered arrays, each with n items. Describe an $O(\log\ n)$-time algorithm for finding the kth smallest key in the union of the keys from S and T (assuming no duplicates).

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 $O(\log^2 n)$.
After solving this will give O(log n) running time complexity for the whole process.

Q5. No. 3.6.19

Describe how to perform an operation `removeAllElements`(k), which removes all key-value pairs in a binary search tree T that have a key equal to k, and show that this method runs in time $O(h + s)$, where h is the height of T and s is the number of items returned.

Ans.

To remove all the nodes 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

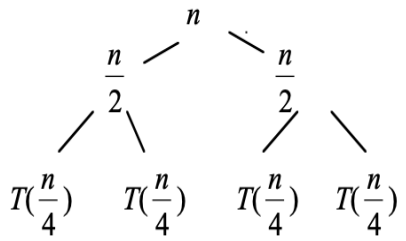
// (binaryPostorder algorithm is defined in Chapter 2 - Figure(2.4.12))

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

Suppose you are asked to automate the prescription fulfillment system for a pharmacy, MailDrugs. When an order comes in, it is given as a sequence of requests, "x_1 ml of drug y_1," "x_2 ml of drug y_2," "x_3 ml of drug y_3," and so on, where $x_1 < x_2 < x_3 < \cdots < x_k$. MailDrugs has a practically unlimited supply of n distinctly sized empty drug bottles, each specified by its capacity in milliliters (such 150 ml or 325 ml). To process a drug order, as specified above, you need to match each request, "x_i ml of drug y_i," with the size of the smallest bottle in the inventory than can hold x_i milliliters. Describe how to process such a drug order of k requests so that it can be fulfilled in $O(k \log(n/k))$ time, assuming the bottle sizes are stored in an array, T, ordered by their capacities in milliliters.

Ans.
Appling divide and conquer method (Binary search) to store a drug in the smallest bottle in the inventory hold x_i 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
$T(n) = k*c + T(n/2^k)$
$T(n) = c \log n$

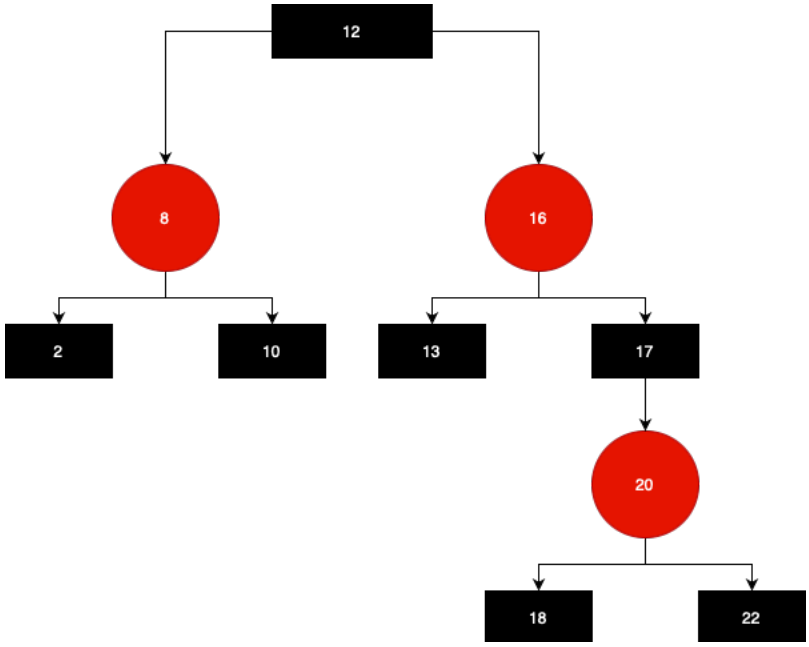
For n requests the time complexity of the above method is c log n but we must process drug order of k requests will change the time complexity to O(k log n).

Since the order is already sorted for x_i it will take less time to search for k requests. After every x_i which changes the complexity to O(k log n/k).

Q7. No. 4.7.15

Draw an example red-black tree that is not an AVL tree. Your tree should have at least 6 nodes, but no more than 16.

Ans.



Q8. No. 4.7.22

The ***Fibonacci sequence*** is the sequence of numbers,

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots,$

defined by the base cases, $F_0 = 0$ and $F_1 = 1$, and the general-case recursive definition, $F_k = F_{k-1} + F_{k-2}$, for $k \geq 2$. Show, by induction, that, for $k \geq 3$,

$F_k \geq \phi^{k-2}$,

where $\phi = (1 + \sqrt{5})/2 \approx 1.618$, which is the well-known ***golden ratio*** that traces its history to the ancient Greeks.

Hint: Note that $\phi^2 = \phi + 1$; hence, $\phi^k = \phi^{k-1} + \phi^{k-2}$, for $k \geq 3$.

Ans.

Given: $F_0 = 0$ and $F_1 = 1$, and $F_k = F_{k-1} + F_{k-2}$, for $k \geq 2$

Proof: $F_k \geq \varphi^{k-2}$,

Base Case: for k=2
$F_2 = F_1 + F_0 = 1$
$F_k \geq \varphi^{k-2} \Rightarrow \varphi^0 = 1$
Therefore, true for k = 2

Base Case: for k=2
$F_3 = F_2 + F_1 = 2$
$F_k \geq \varphi^{k-2} \Rightarrow \varphi$
$F_k > \varphi$
Therefore, true for k = 3

Now check for k-1

$F_{k-1} = F_{k-2} + F_{k-3} > \varphi^{k-4} + \varphi^{k-5} = \varphi^{k-4}\left(1 + \frac{1}{\varphi}\right) = \varphi^{k-4}\left(\frac{\varphi+1}{\varphi}\right) = \varphi^{k-3}$ (Given: $\varphi 2 = \varphi + 1$)

$F_{k-1} \geq \varphi^{k-3}$ (Assumption true for k-1)
So, it will be true for $F_k \geq \varphi k-2$

Hence, we can say that, for k >= 3, $F_k >= \varphi^{k-2}$

Q9. No. 4.7.47

Suppose your neighbor, sweet Mrs. McGregor, has invited you to her house to help her with a computer problem. She has a huge collection of JPEG images of bunny rabbits stored on her computer and a shoebox full of 1 gigabyte USB drives. She is asking that you help her copy her images onto the drives in a way that minimizes the number of drives used. It is easy to determine the size of each image, but finding the optimal way of storing images on the fewest number of drives is an instance of the ***bin packing*** problem, which is a difficult problem to solve in general. Nevertheless, Mrs. McGregor has suggested that you use the ***first fit*** heuristic to solve this problem, which she recalls from her days as a young computer scientist. In applying this heuristic here, you would consider the images one at a time and, for each image, I, you would store it on the first USB drive where it would fit, considering the drives in order by their remaining storage capacity. Unfortunately, Mrs. McGregor's way of doing this results in an algorithm with a running time of $O(mn)$, where m is the number of images and $n < m$ is the number of USB drives. Describe how to implement the first fit algorithm here in $O(m \log n)$ time instead.

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).