Jay Kalyanbhai Savani
20009207

Assignment 2

Chapter – 2

EXERCISE | 2.5.13

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

Feedback?

Ans: Check if Queue Q1 is empty before using push() on the two Queues, Q1 and Q2. Start enqueuing elements if the queue is empty. In this case, we'll add elements 0, 1, and 2.

Now when we must pop() an element, rule 2 states that it should pop out first. To do this, we'll use Queue Q2 and begin enqueueing elements from Queue Q1. We'll keep doing this until Queue Q1 has just 1 element left, or 2.

We dequeue it now that there are just 2 items in Queue Q1.

We now wish to add element 3. So., we add it to Queue 2.

Push() and pop() operations are carried out in a stack in this manner. This means that two queues can be used to implement a stack.

Enqueue and dequeue operations have an O(1) running time. Push() and pop() operations will therefore likewise operate in O(1) time.

EXERCISE | 2.5.20

(a) 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.

Feedback?

Ans: We can create a recursive technique to print each node's depth. The depth is 0 when the current node is the tree's root. The depth of its children, which is 1, can then be printed. The depth of their offspring, which is 2, can then be printed. After that, we can print the depth of all the nodes using the recursion method.

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
```

Run time=O(n)

n=number of nodes in a tree.

EXERCISE 2.5.32

(a) 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?

Feedback?

Ans: Consider the root node and begin traversing it to get the lowest common ancestor between nodes x and y in a tree T. The root of a node is the lowest common ancestor if the specified values for x and y match. Call the lowest common ancestor algorithm iteratively for the left subtree and right subtree if x and y don't match. The parent node is lca if x and y are present as the left child and right child, respectively. If x and y are present in the left subtree, lca is from the left subtree, and vice versa for the 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
```

Run time=O(n)
n=height of the tree.

Chapter – 3

EXERCISE | 3.6.15

(a) 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).

Feedback?

Ans: In a union of keys from S and T, where S and T are both ordered arrays of n items each, one must locate the kth smallest key.
Look at the k/2 member in the array list S first.
Now perform a binary search on the element in T that is the greatest and less than k/2. Add these two items' indices now:

• Take no more than two items if the sum of them, or k, is equal.
• Binary search is carried out to the right of S if sum > k.
• A binary search is conducted to the left of S if sum k.

The identical operations are now carried out on T based on the largest element in S that is less than the element being used right now.

Performing a binary search for two arrays S and T will take O(log n) and O(log n) respectively. That is O($\log^2$ n).
Solving this will give O(log n) running time complexity.

EXERCISE 3.6.19

(a) 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.

Feedback?

Ans: Perform post order traversal on the tree, then recursively call the left and right subtrees and free the nodes in order to remove every node from a binary search tree.

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
```

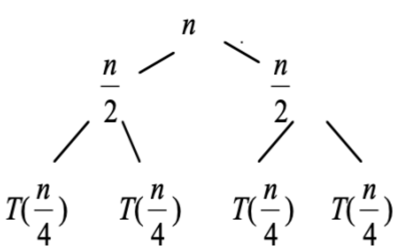
We spend O(h) time to find the node that equals to the target.
And then we spend s to remove all the duplicate. So, the total time complexity is O(h + s)

EXERCISE 3.6.26

(a) 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.

Feedback?

Ans: Using the binary search (division and conquer strategy) to store medicine in the xi millimeter-tiniest bottle in the inventory. The smallest bottle will be found on the left and the largest on the right, as ordered in an array T by capacity.



The above approach will give the recurrence relation:
T(n) = T(n/2) + c

Solving this recurrence relation using the 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 the 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).

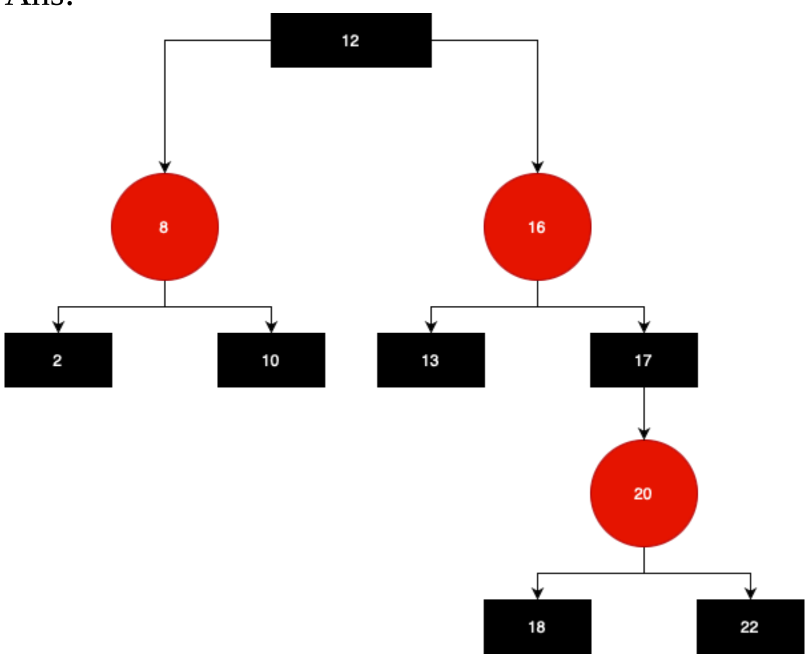
Chapter – 4

EXERCISE | 4.7.15

(a) 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.

Feedback?

Ans:



EXERCISE | 4.7.22

(a) 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 $k \geq 3$.

Feedback?

Given: $F_0 = 0$ and $F_1 = 1$, and $Fk = Fk-1 + Fk-2$, 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 $Fk \geq \varphi k-2$

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

EXERCISE | 4.7.47

(a) 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.

Feedback?

Ans: Bin packing problem: A finite number of bins or containers, each of volume V, must be filled with objects of varied volumes in a method that uses the fewest possible bins. Initial fit heuristic The objects are processed by the algorithm in any sequence. It makes an effort to put each thing in the first bin that has room for it. If no bin is present, a new bin is opened and the object is placed inside of it. We can reduce the running time complexity of saving the photos onto the hard drives from O(mn) to O with the aid of balancing search trees (AVL tree) (m log n). n m is the number of USB drives, and m is the number of photos. As opposed to checking putting images in an order to check all the m drives to see whether there is any space remaining in the previous bins (according to First fit), which takes O(log n) for n items, executing insertion operation in AVL tree takes m running time. Therefore, the initial fit's total running time complexity is O. (m log n).