Ali Abdullah Ahmad
20031246
Assignment 2

CS 600
Prof. Reza Peyrovian

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.

We will be using two queues to implement a stack.
Steps:

1. Initialize two queues let's say q1 & q2.
2. A push operation is executed. Check if queue q1 is full. If yes then print a message stating stack is full, cannot add more elements. If not, then add the element in the queue q1.
3. A pop operation is executed. Check if the queue q1 is empty. If yes, then print a message stating stack is empty, element cannot be popped. If not, remove all the elements except the last element from queue q1 and add it to queue q2. Now remove the last element from q1 and return it. Once again remove all the elements from q2 and add it to q1.

The time complexity of push operation is O(1) and the time complexity of pop operation is O(2n) which is equivalent to O(n).
The space complexity for both the operations is O(2n).

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 Answer:
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, which is continued.
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)

Ans.

Inputs:

- A tree T represented as a rooted tree.
- Two nodes x and y.

Assumptions:

- Each node has a pointer to its parent or a way to traverse the tree.
- If x or y are the same, x (or y) is the LCA.

Steps:

- Traverse upward from x to the root and store all its ancestors in a set (or hash table).
- Start traversing upward from y. For each ancestor of y, check if it exists in the ancestor set of x.
- The first common ancestor found during this traversal is the LCA.

Output:

- Return the node that is the LCA of x and y.

PSUEDOCODE

```
function findLCA(Tree T, Node x, Node y):
   Set ancestorsX = empty set

   // Add all ancestors of x to the set
   while x is not null:
      ancestorsX.add(x)
      x = x.parent

   // Traverse ancestors of y and check for the first common ancestor
   while y is not null:
      if ancestorsX.contains(y):
         return y
      y = y.parent

   return null // If no common ancestor is found
```

Time Complexity

- **Step 1 (Storing ancestors of x)**: O(h), where h is the height of the tree.
- **Step 2 (Traversing ancestors of y)**: O(h)
- **Overall**: O(h) where h is the height of the tree.
  In a balanced tree, h=O(logn). In a worst-case unbalanced tree, h=O(n).

Space Complexity

- O(h) due to storing ancestors of x.

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.

Algorithm: kthSmallest(S, T)

Input: Ordered Arrays S & T storing the values

Ouput: kthSmallest element in the union of the keys from S & T.

```
if(Arraysize of S > Arraysize of T)
        kthSmallest(T, S)
low← Maximum(0, k - Arraysize of T)
high←Minimum(k, Arraysize of S)
while(low <= high)
        cut1← low + (high - low) / 2
        cut2← k - cut1
        l1← 0
        l2 ←0
        r1 ←0
        r2 ←0
        if(cut1 == 0) then
                l1← Integer.MIN_VALUE
        else
                l1 ←S[cut1 - 1]
        if(cut2 == 0) then
                l2← Integer.MIN_VALUE
        else
                l2← T[cut2 - 1]
        if(cut1 == ArraySize of S) then
                r1 ←Integer.MAX_VALUE
        else
                r1← S[cut1]
        if(cut2 == ArraySize of T) then
                l2← Integer.MAX_VALUE
        else
                l2 ←T[cut2 ]
        if(l1 <= r2 and l2 <= r1) then
                return min(S[l1], T[l2])
        else if(l1 >r2)
                high = cut1 - 1
        else
                low = cut1 + 1
return 0
```

The running time of the above algorithm is O(logn) and the space complexity is O(1)

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 Tree and s is the number of items returned

Ans.

Steps of the Algorithm:

1. **Start at the Root Node:**
   Begin traversal from the root of the BST.
2. **Base Case – Node is null:**
   If the current node is null, simply return as there is nothing to remove.
3. **Node's Key Matches k:**
   If the key of the current node is equal to k, there are three scenarios to consider:

a. Current Node is a Leaf (External Node):

- Directly remove the node by setting its parent pointer to null.

b. Current Node has One Child (Left or Right):

- Replace the node with its non-null child (either left or right).
- Adjust the parent's pointer to bypass the current node.

c. Current Node has Two Children:

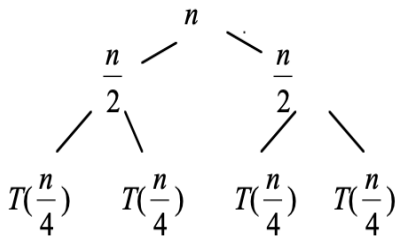
- Find the **in-order successor** (smallest node in the right subtree).
- Replace the key and value of the current node with the key and value of its in-order successor.
- Recursively delete the in-order successor node.

4. **Key k is Smaller or Larger than the Current Node's Key:**
   - If k is smaller than the current node's key, move to the left child.
   - If k is larger than the current node's key, move to the right child.
5. **Repeat Steps Recursively:**
   Continue the process until all nodes with key k are removed.
6. **Stop the Process:**
   The process ends once all nodes with key k are removed, and the tree is restructured accordingly.

Time Complexity Analysis:

1. **Height of the Tree (h):**
   Traversing the tree to locate nodes with key k may take up to O(h), where h is the height of the tree.
2. **Number of Deletions (s):**
   For every node with key k, a deletion operation is performed. The cost of each deletion depends on the structure of the tree but generally takes O(logn) time in a balanced tree. The total cost for s nodes is proportional to s.
3. **Combined Complexity:**
   Combining the traversal and deletion operations, the total complexity is O(h+s), where h is the height of the tree, and s is the number of nodes deleted.

Q6. No. 3.6.26
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).

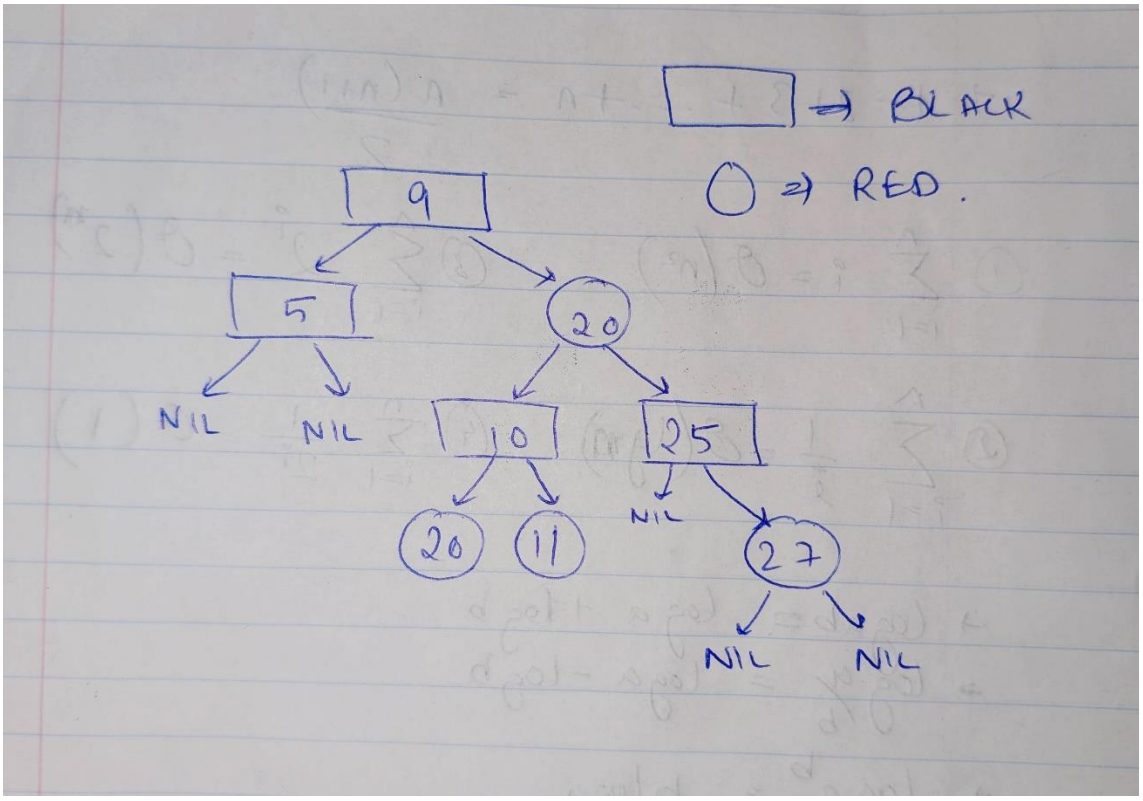
Since the order is already sorted for x_i it will take less time to search for k requests. After every x_i which changes the complexity to O(k*log(n/k)).

Algorithm to create balancedBST
Algorithm: balancedBST(Array T, start, end)
Input: An ordered array T, start and end variables
Output: A Balanced BST

```
    if(start > end) then
        return null
    mid = start + (end - start) / 2
    Node node = new Node(T[mid])
    node.left = balancedBST(T, start, mid - 1)
    node.right = balancedBST(T, mid + 1, end)
return node
```

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.



Nodes 20 and 11 have no children (NIL)

Q8. No. 4.7.22
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
Ans.

The best fit algorithm runs in $O(n^2)$ if we do not use the AVL tree. We can boil down the complexity of an algorithm to O(nlogn) with the help of AVL tree.
AVL trees uses logn operation for insert. Since we are implementing this process n times, thus the complexity would become O(nlogn).
Each node in the AVL tree is going to store the USB drives capacity.
The AVL tree searches for the smallest USB possible as the heuristics call for placing a picture into the smallest remaining storage USB first. If a sufficient capacity is found, then the AVL tree is updated.
If sufficient capacity is not found in an AVL tree then a new USB drive is made for the image and the leftover capacity is added to the AVL tree.
This operation takes O(logn) if we found or not found the capacity. Since we are doing this operation for m images therefore the complexity would become m * logn.
Consequently, we know that n < m where m is the number of images and n as the number of hard disk. So the time complexity would become O(m * logn) and the space complexity is O(n).