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

Answer:

We will be using two queues to implement a stack.

Steps:
1. Initialize two queues lets say q1 & q2.
2. A push operation is executed. Check if the 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 poped. 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 operation is O(2n).

ii.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:

We will be using nodes to represent the tree data structures.

Each node will have one of the property named ‘depth’ which will be used to store the depth of the node.Intitially the depth of all nodes is zero. Then we will run the algorithm to compute the depth of each nodes.

Algorithm: depthAllNodes(Node node, d):
Input: A tree T with nodes n
Ouput: Computing the depth of all nodes

```
node.depth ← d
if(node does not contain any children) then
        return;

For each children of x in (node)
        depthAllNodes(x, d + 1)

return;
```

We will call the function depthAllNodes and pass the root node along with value of d as 0 to compute the depth.
For Example: **depthAllNodes(root, 0)**

The above algorithm runs in O(n) time
iii.

Answer:
We will represent the Tree(T) with the help of the nodes. Each nodes consists of left_child, right_child and the data.

Algorithm: lowestLCA(node x, node y, node z)
Input: A tree t with n nodes
Output: LCA of two nodes

```
if node z is null then
        return null

if(z.data is equal to  x.data or  z.data is equal toy.data)
        return z;

left_lca ← lowestLCA(x, y, z.left)
right_lca ←   lowestLCA(x, y, z.right)

//One key is present in the left subtree and other in right
if(left_lca is not equal to null and  right_lca is not equal to null) then
        return z

if(left_lca is null) then
        return right_lca

return left_lca
```

There can be three possibilties
a. Both the keys are present in the left subtree
b. Both the keys are present in the right subtree
c. One key is present in left subtree and other in right subtree

If both keys are present in the left subtree or right sub tree that means one node is the supervisor of the other node. We will return one of the node.

The above algorithm runs in O(logn) time and the space complexity is O(1).

iv.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 Sand T(assuming no duplicates)

Answer:

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).

v.

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

Answer:

A Binary Search Tree is a tree which consists of following properties:
a. The left sub tree of the node contains the keys which are less than or equal to the node's key.
b. The right sub tree of the node contains keys whihc is greater than the node's key.
c. The left and the right sub tree should also follow property a and b.

Here we are assuming that the binary search tree can store the duplicates.

Steps to remove the elements who key is equal to k
1. Start with the root node.

2. If the current node is null then return.

3. If the statement 2 is not executed then check if the key of the current node is equal to k. If yes then three can be three possiblitites.
a. The current node is an external node.
b. Current node contains only one child(left child or right child).
c. Current node contains both left and right child.

4. If the current node is an external node then just remove that node and return. If the current node contains only one child then remove the node with key equal to k and replace it with either left or right child which is not null and go to step 2. If the current node contains both children find the inorder successor of the right sub tree. Remove the current node with key equal to k and replace it with its successor and go to step 2.

5. If the statement 3 is not executed then check if the key k is less than the key of the current node. If yes then traverse left else traverse right.

6. Go back to step 2 and follow the same process.
7. Stop

The worst case of the above algorithm is O(logn) + O(s) which can also be expressed as O(h + s) where h is the height of the tree and s is the number of items it returned whose key is equal to k.

vi.

Answer:

It is given that all the bottle sizes are stored in an array T ordered by their capacities in milliliters .It is also given that there is unlimited suppy of n distinctly size empty drug bottles

First thing which is coming in our mind is to construct a Balanced binary search tree.

A Balanced BST is a tree where the absolute difference between the height of the left sub tree and right sub tree is 1.

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
```

This operation takes O(n) to construct a balanced BST.

Now comes the find operation.
We will process each x_i one by one to find the smallest element which can find x_i

Steps:
1.Start with the root node and prev variable equal to -1
2. Check if the node value is equal to x_i if yes then return x_i.Else if check if the node value is greater than x_i. If yes then store the value in a variable called prev and left recuse the tree. If not then recurse right tree.
3. If the node value is null then return prev value.
4. Follow step 1 until all the requests are fulfilled.
5. Stop

Prev will return -1 if the element of x_i is greater than all the value present in arrayT.

The time complexity of the above Algorithm is O(n) + O(k * log(n / k) which is equivalent to O(k * log(n / k)) and the space complexity is O(n).

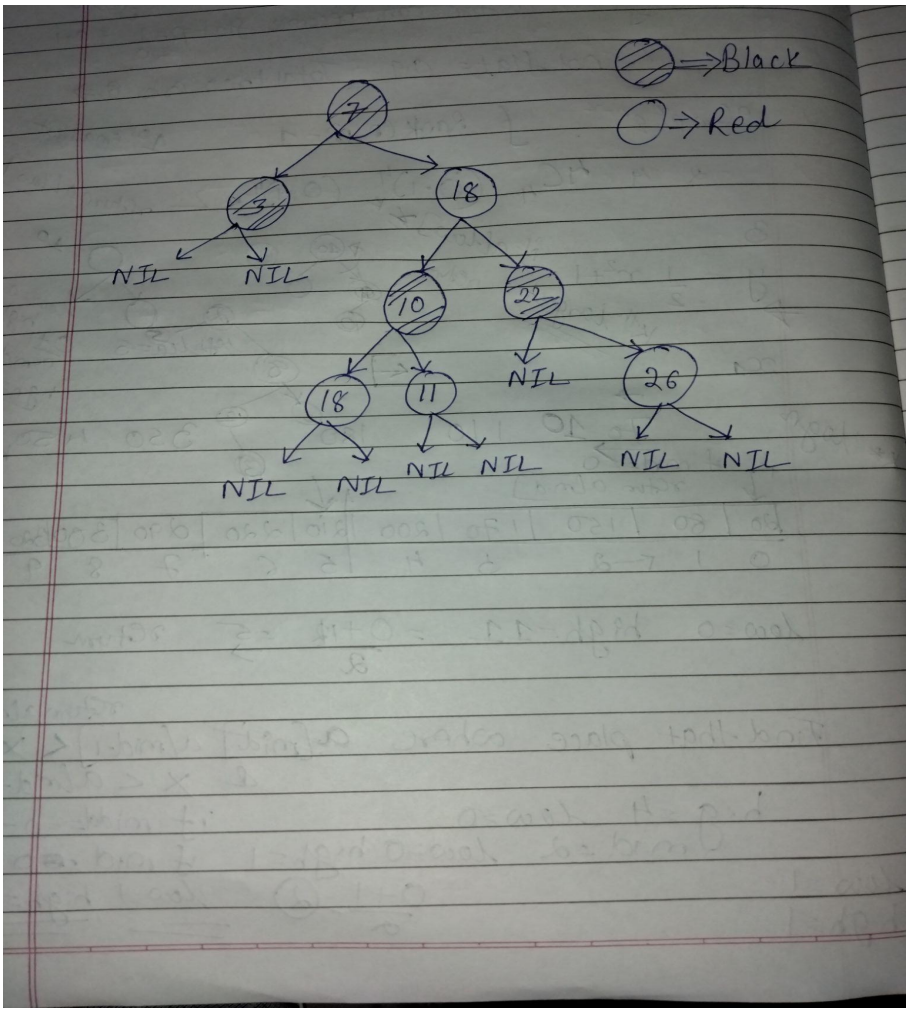
vii.
Draw an example red-black tree that is not an AVL tree. Your tree should have at least 6 nodes, but no more than 16.

Answer:
Property of Red Black Tree
1. Every node has a color either red or black

2. Every external node should be a black node.
3. Root is always a black node.
4. There are no two adjacent red nodes.
5. If you take any path from the current node to any of its descendant null nodes , there should be the same number of black nodes.

Property of AVL Tree

For every node the absolute difference between the height of the left sub tree and right sub tree should be not more than 1.



viii.

We are given that $F_0 = 0$ & $F_1 = 1$ so $F_2 = F_1 + F_0 = 0 + 1 = 1$ & $F_3 = F_2 + F_1 = 1 + 1 = 2$

Here we can see that $F_3 >= \phi$

Hence proved via subsitituting the value

Let us try to prove this statement via mathematical inducation

Let us assume that the statement is true for k

We have to prove that the statement is true for k + 1

$F_{k+1} = F_k + F_{k-1}$

It is also given that $\phi^k = \phi^{k-2} + \phi^{k-3}$

$F_k + F_{k-1} >= \phi^{k-2} + \phi^{k-3}$

Hence

$F_{k+1} >= \phi^{k-2} + \phi^{k-3}$

From the above statement $\phi^k = \phi^{k-2} + \phi^{k-3}$ we can say that

Hence

$F_{k+1} >= \phi^{k-1}$

Or

$F_{k+1} >= \phi^{(K+1)-2}$

Hence proved

ix.

We will be using AVL tree data structure to solve this problem.

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 sotrage USB first. If a sufficient capacity is found then the AVL tree is updated.

If a 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).