Yash Avinash Patole

Course : CS600

Homework 1

CWID: 10460520

Chapter 2

Exercise 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?

Answer:

Use two queues QI and Q2, where QI stores elements and Q2 is used for auxiliary

```
boolean isEmpty() { QL.isEmpty()}:
int size() { Ql.size()}:
void push(Object o) { Q1.enqueue(o) }:
Object pop() |
        While (!QL.isEmpty())
                Q2.enqueue( Ql.dequeue()) ;
        Object 0 = Q2.dequeue():
        While (!Q2.isEmpty())
                Ql.enqueue( Q2.dequeue() );
        Return o;
)
```

Object top(): similar to pop()

isEmpty, size, push: O(1)

pop, top: O(n)

Exercise 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:

An approach for doing this is a pre-order traversal. During a traversal "visit," only record the depth of the node's parent, increased by 1. Each node will now have its depth.

Assuming that each node has a variable called depth, we may calculate each node's depth as follows:

computingDepths(v, d)

{

v.depth=d:

if (v has no child)

return;

else

for each child v' do

computingDepths(v', d+1);

return;

3

To compute the depth of each node, simply call computeDepth(root, 0). This algorithm visits each node only once and each visit takes constant time. Therefore, it's running time is O(n).

Exercise 2.5.32

Answer:

We will represent the Tree(T) with the help of the nodes. Each node consists of left child, right child and the data.
Algorithm: lowest LCA (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 ← lowest LCA (x, y, z.left) right_lca ← lowest LCA (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).

<u>Chapter 3</u>

Exercise 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).

Answer:
ALGORITHM kthsmallestKey(S, T, sFirst, sLast, tFirst, tLast, k):

Input: S - ordered array from lowest to highest of size n

T - ordered array from lowest to highest of size n

sFirst - first position (0) in the array S

sLast — number of elements (n) to be considered in the array S

tFirst — first position in the array T

tLast — number of elements (n) to be considered in the array S

k - location (starting from 1) of the key which is to be returned

Output: returns the kth smallest key in the union of the keys from S and T

SPos <« sLast - sFirst;

tPos « tlast - tFirst;

```
IF sPos <= 0 THEN
        RETURN T[tFirst + k - 1]
ENDIF
IF tPos <= 0 THEN
        RETURN S([sFirst + k - 1]
ENDIF
IF k == 1 THEN
        IF S[sFirst] < T[tFirst] THEN
                RETURN S[sFirst]
        ELSE
                RETURN T[tFirst]
        ENDIF
ENDIF
sMid <« (sFirst + sLast) / 2
tMid <« (tFirst + tlast) / 2
IF s[sMid] <= T[tMid] THEN
        IF (sPos / 2 + tPos / 2 + 1) >= k THEN
                RETURN kthSmallestKey(S, T, sFirst, slast, tFirst, tMid, k)
        ELSE
                RETURN kthSmallestKey(S, T, sMid + 1, sLast, tFirst, tlast, k - sPos
        / 2 - 1)
        END IF
ELSE
        IF (sPos / 2 + tPos / 2 + 1) >= k THEN
        RETURN kthSmallestKey(S, T, sFirst, sMid, tFirst, tLast, k)
        ELSE
```

RETURN kthsmallestKey(S, T, sFirst, sLast, tMid + 1, tlLast, k - tPos / 2 - 1)

END IF

END IF

END

Complexity:

Time complexity for the above algorithm is: O(logn) where n is the number of elements in each of the arrays.

Exercise 3.6.19

Describe how to perform an operation removeA11E1ements(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 the height of T fnd 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 contains the keys which are less than or equal to the node's key.

b. The right sub tree Of the contains keys whihc is greater than the 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 nade contains only one child(left child or right child).

    c. Current node c»ntains both left and right child.

4. Simply remove the current node and come back if it is an external one. Remove the node with key equal to k and replace it with either a left or right child that is not null if the current only has one child, then proceed on to step 2. If the children of the present node discover the correct sub tree's in-order successor. Step 2 is reached by removing the node with a key of k and replacing it with its successor.

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.

Exercise 3.6.26

Answer:

All bottle sizes are predetermined to be placed in an array T that is arranged according to each bottle's milliliter capacity. Additionally, it is assumed that a limitless supply of distinct-sized empty medicine bottles is available.
Building a balanced binary search tree is the first thing that comes to mind.
A tree is considered to be balanced if there is a height difference of exactly 1 between the left and right subtrees.
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 xi one by one to find the smallest element which can find xi

Steps:

1.Start with the root node and prev variable equal to -1

2. Check if the node value is equal to xi if yes then return xi.Else if check if the node value is greater than xi. 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 xi is greater than all the value present in arrayT.

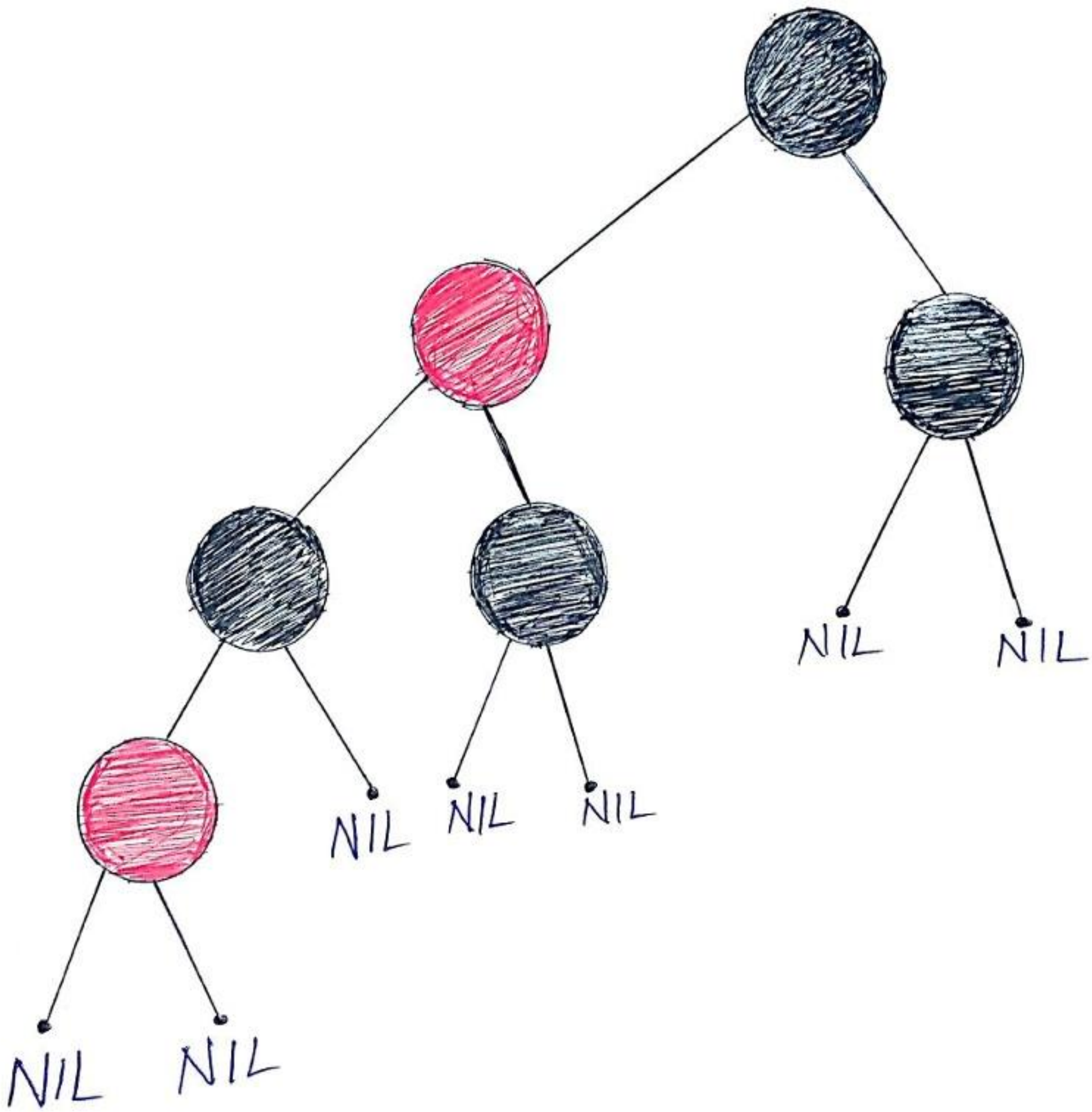
The time complexity of the above Algorithm is O(n) + O(k * log(n / k) which is equivalent to O(k * log(n / k)) and the space complexity is O(n).

P.T.O

Chapter 4

Exercise 4.5.15

Answer:



Exercise 4.5.22

Answer:

Answer :

Consider given statement as P(E), i.e,
$F_k \geq \Phi^{k-2}$ for $k \geq 3$

Now,
$k = 3$, $F_3 = 2 > \frac{1+\sqrt{5}}{2}$

∴ P(E) is true

∵ Let, P(E) is true for $k \in \{3, \ldots, m\}$
where $n \geq 3$

So, $\Phi^2 = \Phi + 1$

∵ $\Rightarrow 1 = \Phi^{-1} + \Phi^{-2}$

i.e $\Phi^k = \Phi^{k-1} + \Phi^{k-2}$ for $k \geq 3$

Therefore, for $k = m+1$,

$F_{m+1} = F_m + F_{m-1} \geq \Phi^{m-2} + \Phi^{m-3}$
$= \Phi^{n-1}$

According to the solution above P(E) is true for $k = m+1$

∴ The given statement is true for all $k \geq 3$ by induction hypothesis.

Exercise 4.7.47

Answer:

Red - Black Tree:

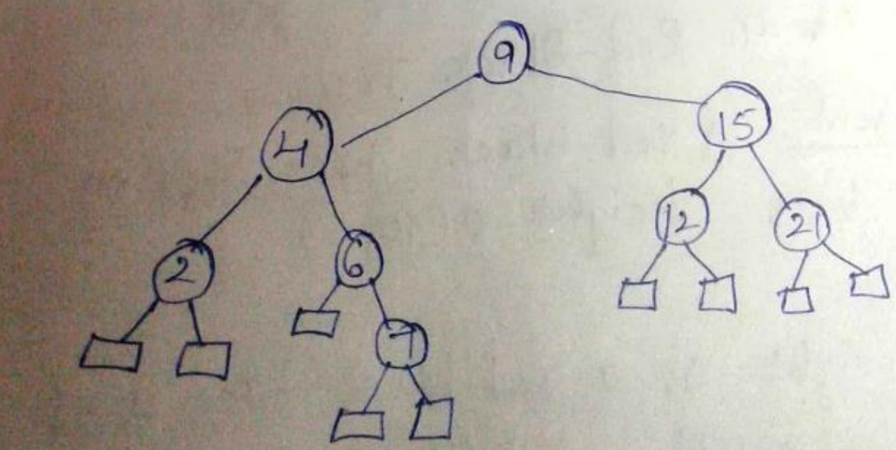
Red - Black tree can also be defined as a binary search tree that satisfies the following properties:

→ Root property: Root of tree is always black.

→ External property: Every leaf is black

→ Internal property: The children of a red node are black.

→ Depth property: all the leaves have same black depth.



Red- Black Tree

→ By the above theorem, Searching in a red-black tree takes o(logn) time.

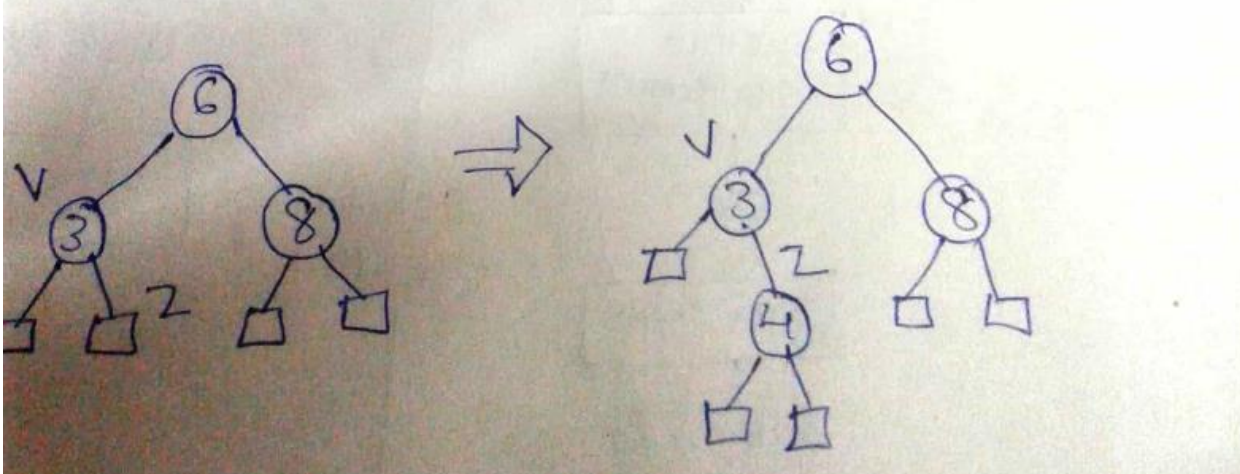
Insertion:- To perform operation put(k,o) we execute the insertion algorithm for binary search trees and color red the newly inserted node z unless it is the root

· we preserve the root, external, and depth properties.

→ If the parent v of z is black we also preserve the internal propety and we are done

→ Else (v is red) we have a double red (i.e a violation of the internal property) which requires a re organization of the tree

Example where the insertion of u causes a double red:

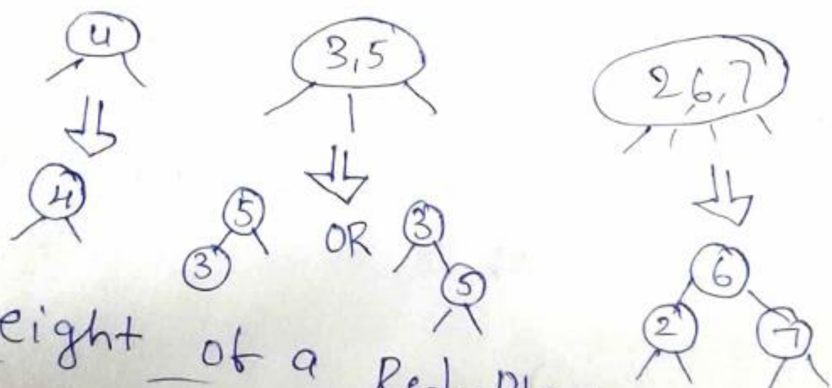


Form (2,4) to Red - Black Tree:

→ It representation of a (2,4) tree by means of a binary tree whose node are colored red or block

→ In comparsion with its associated (2,4) tree, a red - black tree has

- Same, logarithmic time performance
- Simpler Implementation with a single node type.



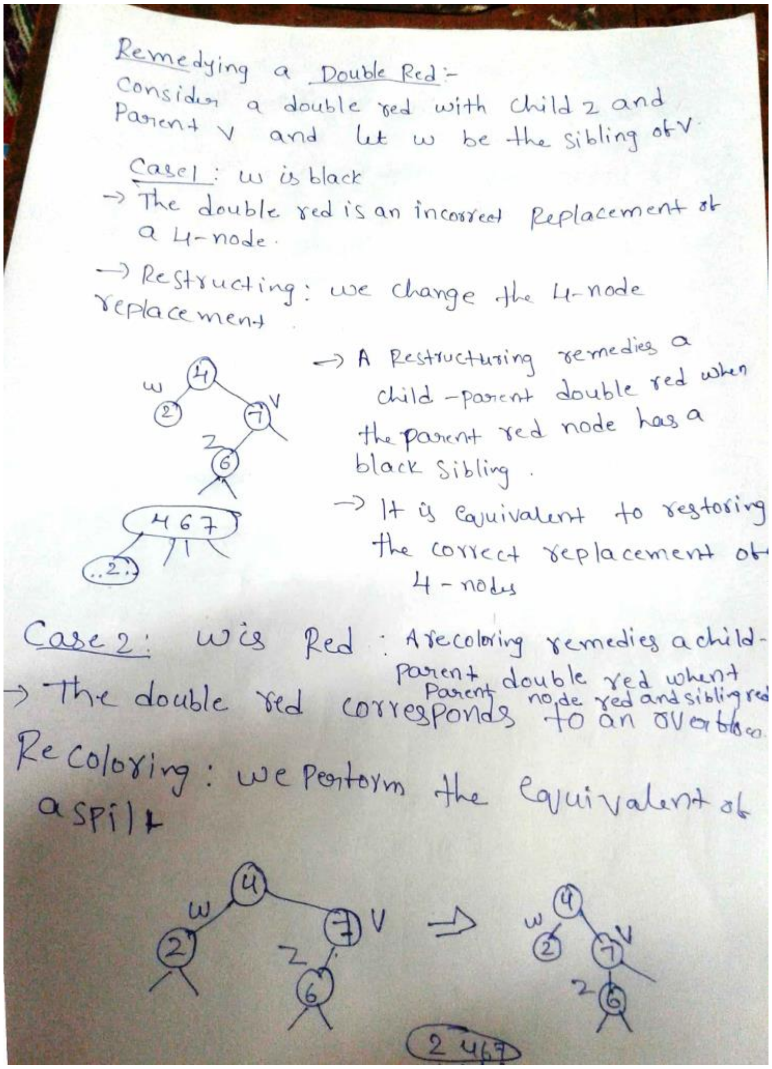
Height of a Red-Black Tree:

Theorem: A red-black tree storing n entries has height $O(\log n)$

Proof:

1. The height of a red-black tree is at most twice the height of its associated (2,4) tree, which is $O(\log n)$

The Search algorithm for a binary search tree is the same as that for binary search tree.



Remedying a Double Red:-
Consider a double red with child z and Parent v and let w be the sibling of v.

Case 1: w is black
→ The double red is an incorrect Replacement of a 4-node.
→ Restructing: we change the 4-node replacement.

→ A Restructuring remedies a child-parent double red when the parent red node has a black sibling.
→ It is equivalent to restoring the correct replacement of 4-nodes

Case 2: w is Red : A recoloring remedies a child-parent double red when Parent node red and sibling red
→ The double red corresponds to an overflow
Re coloring: we perform the equivalent of a split

The algorithm recommends inserting a photo into the smallest available storage USB first, thus we must proceed through our tree starting at the root node and working our way down to the smallest USB. This will take O(log n) time, and since it needs to be done m times for every image, it will also take O(m log n) time.

Finally, we delete the photo from our tree, fix the amount of storage that is still accessible, and add this new value to our tree each time we discover a USB key to store the picture in. It takes log (n) time to remove a value from the tree and add a new value, and this process is done for each image. So this is takes time O(2m log n).

Since, n < m thus n log(n) < m log(n), all of these procedures together take O(n log n + 3m log n) = O(m log n).