Assignment-4, ~~coding~~ theory B19003

1) ~~a~~ Algorithm for generating a red-black tree in O(n) time -

1) first make a balanced binary tree.

a) place the middle element as the root.

b) place the elements from beginning to ~~end~~ middle-1 to the left subtree.

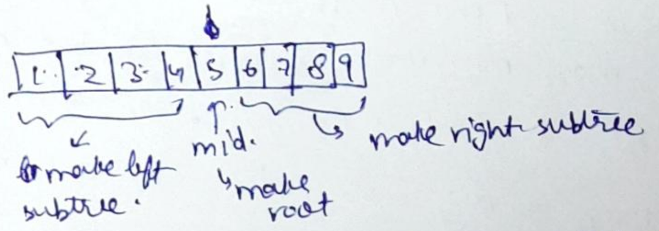
c) place the elements from middle+1 to end to the right subtree.

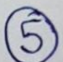
d) do steps b and c recursively. ~~like~~ This will make ~~the~~ your left and right subtrees.

e) Thus, you are traversing the whole sorted array once, hence O(n).

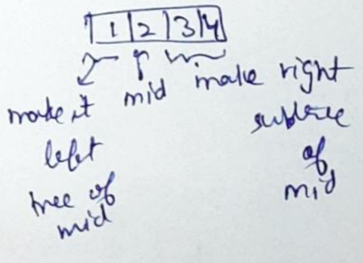
2) now, mark all the nodes as black and ~~leaves~~ the deepest nodes h. as red.

eg -

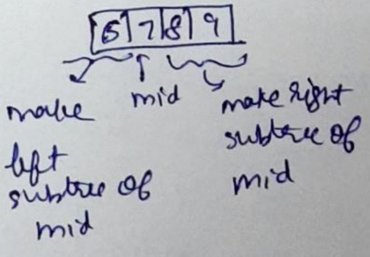


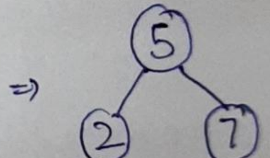


For leftsubtree;

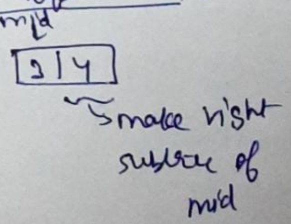


For right subtree

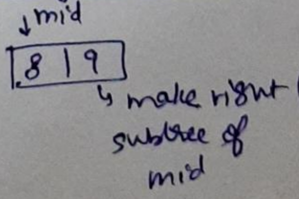


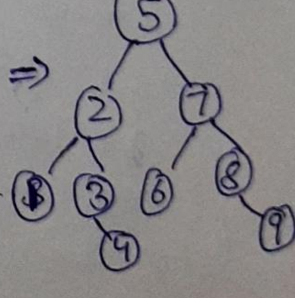


For left subtree,

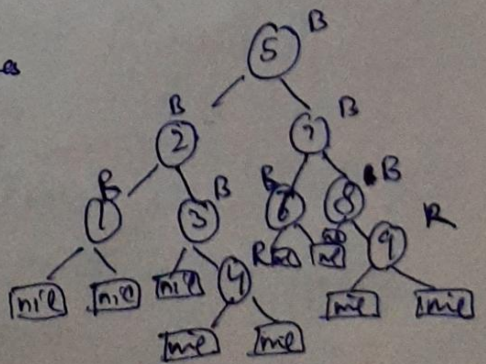


For right subtree





colour nodes as black and ~~leaves~~ deepest nodes as red.



2) a) Given : 2 RBTs of size $\sqrt{n}$ and n.

Algorithm:

a) Traverse the RB tree with $\sqrt{n}$ elements and store the values in an array. This is $O(\sqrt{n})$.

b) Now, add the elements from the above array to the RBT having n elements. We know that the time complexity of insertion of an element to RBT has TC of $O(\lg n)$. Thus, for $\sqrt{n}$ elements, it will take $O(\sqrt{n} \lg n)$.

c) Thus, the overall time complexity is $O(\sqrt{n} \lg n)$. Let's check if it's $o(n)$ or not. So,

$$\lim_{n\to\infty} \frac{\sqrt{n}\lg n}{n} = \lim_{n\to\infty} \frac{\lg n}{\sqrt{n}}$$

(use L'hospital).

$$= \lim_{n\to\infty} \frac{1/n}{\frac{1}{2} n^{-1/2}}$$

$$= 2 \lim_{n\to\infty} \frac{1}{\sqrt{n}}$$

$$= 0$$

Thus, there is a total order between $\sqrt{n}\lg n$ and n, hence it's a $o(n)$ algorithm.

FIITJEE 2) b) In this question, I am assuming that the elements in ~~both~~ both red-black trees are distinct. Consider two trees t_1 and t_2 such that t_1 has more nodes than t_2. Now we have ~~two~~ three cases.

- ① - root node of t_1 is more than that of t_2.
- ② - root node of t_2 is more than that of t_1.
- ③ - root node of t_1 is equal to that of t_2.

Case ① - Algorithm -

a) find ~~the~~ an element in t_1 which is smaller than the key of root node of t_2. If you find that element, detach the whole subtree under that node from t_1 and attach it in t_2 after finding a suitable position in left side.

~~b) Now, the attach~~

b) Now, attach the left-over subtree of t_1 ~~and~~ to the right of t_2.

c). Fix-up the created ~~to~~ red-black tree.

Case ② - Algorithm -

a) It's similar to the earlier algorithm. Just replace t_1 and t_2.

Case ③ - Algorithm -

a). Attach the left subtree of t_2 to the left of t_1 and the right subtree of t_1 to the right of t_2.

b) Fix-up the created red-black tree.

Fix-up algorithm - If the subtree to be attached has its node black; then it's fine. ~~But~~ No color changes and rotations are required. unless the black height is same. If there is imbalance of black heights, then we can colour the ~~right~~ ~~leaves~~ nodes/leaves of right subtree, to black, in order to maintain a balance.

FIITJEE

Similarily, if the subtree to be attached have its node red and it's getting attached to a red leaf, then there will be red-red conflict and should be resolved using ~~[illegible]~~ rotations. Then do color changes for maintaining the black height.

Correctness of algorithm -

Case I and II -

The subtree with values more than root of other tree ~~[illegible]~~ is being placed at the right subtree of the other one. Similarily, the lesser values are placed at the left subtree of the other one. This basically takes $O(\lg m + \lg n)$ where m is black height of one tree and n is black height of the other. This is because, we are only traversing along the heights.

Case III -

Since the roots are same, we are just placing the left subtree of t_1 to that of t_2 and right subtree of t_1 to that of t_2. This the properties are not violated. If they are, then we can fix it with colour changes and rotations.

For part 2, its $O(\lg n + \lg\sqrt{n})$ which is $o(n)$. so

$$\lim_{n\to\infty} \frac{\lg n + \lg\sqrt{n}}{n} = \lim_{n\to\infty} \frac{3}{2}\frac{\lg n}{n} = \lim_{n\to\infty} \frac{1/n}{1} = \underline{\underline{0}}$$

3) According the property of BST, elements of the left subtree must be smaller and elements of right subtree must be greater than root.

Two arrays represent the same BST if, for ~~any~~ every element x, the elements in left and right subtrees of x appears after it in both arrays. And this is true for roots of both left and right subtrees.

What we have to do is to check if next smaller and greater elements are same in both arrays. Same properties are checked for left and right subtrees recursively. By following this algorithm, we will actually check all the conditions for all elements.

5) Insertion and deletion of elements may violate the properties of red black tree. Thus, restoring the red-black properties requires a small number O(logn) ~~~~ of color changes and not more than three rotations ~~for rem~~ (in case of removal) and two rotations (in case of insertions). This is due to the red-black properties of red-black tree. This can be shown for different cases of insertion and deletion.

10

For insertion - (as given in lecture slides).

Case I : no colour changes / rotations.

Case IIa): In this only one rotation is required i.e. right ~~case IIa~~ rotation. But still the tree is not fixed.

Case II b): In this, only a left rotation is required. Then we make colour changes and at worst case, it will logn colour changes.

For deletion - (as given in lecture slides)

deletion of red node {
Case I : no rotations/ colour changes.
Case II : " " "
Case III : depends on further heights of tree. But in this operation, no colour changes / rotations

deletion of black node.

Case I: we only perform a left rotation, which creates further subproblems, depending on the color of the children of the sibling of double black node v.

Case II: 1) a) Only one colour change. Proceeds to 1(b).

b) 2 colour changes, Fixing ends here

2) Perform right rotation and 2 color changes.

3) Perform left rotation and make 2 color changes.

Thus for deletion, we see at most 2 rotations and at most lgn colour changes. This is due to the properties with which the red-black tree is formed.

6) Successor of x in a red-black tree can be found in the similar way as found in balanced binary search tree. The pseudocode is given below -

```
Tree-Successor (x).
  if x.right ≠ NIL,
        return TREE-MINIMUM (x.right)

  y = x.p
  while y ≠ NIL and x = y.right.
        x = y
        y = y.p
  return y
```

If the right subtree of node x is non-empty, then the successor of x is just the leftmost node in x's right subtree, which find by TREE-MINIMUM (x.right). If the right subtree is empty and x has a successor, then y is the lowest ancestor of x whose left child is also an ancestor of x. To find y, we simply go up the tree from x until we encounter a node that is the left child of its parent.

Time complexity = $O(\lg n)$

4) Effectiveness -

a) will lead to lesser computations as we have to just find the node to be deleted and change its ~~pom~~ boolean pointer.

Ineffectiveness -

a) Property S violation - If we make a red-black with a good number of insertions and deletions, it will be difficult to maintain the ~~balance~~ property S as the elements are not actually getting deleted.

b) ~~Propert~~ Insertion - More elements have to traversed in case of insertion in ~~a~~ this special RBT. This ~~might~~ increases the no. of computations in insertion for the above one. For the normal RBT, we have to traverse less no. of elements, hence less computations. It is also difficult to maintain RBT property

c) ~~stac~~ searching - More computations in special RBT than in normal RBT.

d) Deletion - Involves searching which increases computations. Hence, it is bad ~~for~~ with working with huge amount of data.

Improvements -

a) ~~Stem~~ Instead of using boolean value ~~of~~ for deletion, we use indexes of arrays from where it is made. This will facilitate faster searching and other augmentations involved with RBT like insertion, deletion etc.

b) We can remove the additional parameters and use it as a normal RBT.

~~c)~~