SOLO

Assignment-4
Theory (CS-202)

Anukool Dwivedi
B19071
Grp-13

① The procedure to create red black tree from n elements (sorted) is :-

ⓐ Find the middle element and make it as root.

ⓑ Then place [left, mid) elements on left and [mid+1, right] elements on right recursively.

ⓒ Mark all the nodes as Black except the last level. Mark last level nodes as red.

In this way, the red black tree can be generated.

② Part ①; Tree 1 → $\sqrt{n}$ elements
Tree 2 → n elements

Insert nodes (or elements) from Tree 1 to Tree 2.
This would take $O(\sqrt{n} \log n)$ time which is $o(n)$.

As $\lim_{n \to \infty} \frac{\sqrt{n} \log n}{n} = \lim_{n \to \infty} \frac{\log n}{\sqrt{n}} \to 0$

So, our algorithm would take $o(n)$ time.

Part ② Given; Tree 1 → $n/2$ elements
Tree 2 → ~~$n/2$~~ n elements

Both the trees are Red Black.

(i) Take inorder traversal of both the trees and that will be sorted.
(ii) Merge both the inorders using 2-pointer merging as done in Merge function of Merge Sort.
(iii) Now, we have a sorted array of $3n/2$ elements.
(iv)
- Make middle as root.
- Recur on left side [[left, mid)] and on right side [[mid+1, right]].
- This way, we generate a red-black tree.

~~No of operations are $(3n/2 + n/2)$~~

Time complexity: $O(3n/2) + O(3n/2) + O(3n/2)$

$\Rightarrow O(9n/2)$

Sol, time complexity :-
let $f(n) = 9n/2$ and $g(n) = n$

So; $f(n) \leq 100n$
$\Rightarrow f(n) < C\,g(n)$; for some const C and $n > 1$.

Hence, it is $o(n)$.

③ Inorder of BST gives sorted order elements.

So, the procedure is :-

(a) ~~Sor~~ Find inorders of both the trees.

(b) Compare the inorders element wise.

(c) If all same, then same set of elements.
Else, different set of elements.

④ This is definitely ineffective.
Let us consider a RBTree of n nodes.
So; $ht = \log n$.
Let us ~~dd~~ insert 2^n nodes more and delete 2^n nodes.
So; Total height after all is $\log(2^n) \approx n$.
Now, for further operations, it would take $O(n)$ time per operation.
But if it were an original RBTree; it would take $O(\log n)$ per operation.

Also; this tweak also need extra storage.

Now, for effectiveness :-
⇒ As we are not deleting the node, so it has the property of the colors and hence it maintains the old black tree after deletion as there is no need to rearrange the nodes.
⇒ We can also insert the nodes as before only considering the colors of the new node.
⇒ Also, we can do search operation as before as all the elements are distinct.

(5) For insertion :-

Rotations :- O(1) [2 rotations]
(when the uncle is black in color)
There are 4 rotations :-
LL, LR, RR, RL

Recoloring :- O(logn)
(when the uncle is red in color)
We recolor parent, uncle and grand parent.
Then recur on grand parent.
So, it is proportional to height of Tree.

For deletion :-

Rotations :- O(1) [2 rotations]
Recoloring : O(logn)
If sibling is black & its both children are black, then we perform recoloring and recur for the parent. So, it could be O(logn), the height of tree.
~~In all other cases~~
But we require at most 2 rotations in all cases.

(6) The procedure is as follows :-

(a) If right subtree of node is not NULL, then successor lies in right subtree. So, we go to right subtree and return the node with minimum key value in the right subtree.

(b) If right subtree of node is NULL, then successor is one of the ancestors, So, we travel up using the parent pointer until we see a node which is left child of its parent. The parent of such a node is successor.