CS 2233 - Assignment 3 - Theory questions

(1) A relaxed red black tree defined as BST satisfying:

1. root can be either red or black
2. every node is either red or black
3. Red node has both black children only.
4. Every ~~leaf~~ NIL is black
5. any path from a node its descendent NIL have same amount of black nodes (black depth)

taking the usual red black tree conditions, as you used the word "Relaxed red black tree".

T = red black tree with red root (given)

T' = red black tree with root of T made black, and no other changes.

(a) checking :- every node is either red or black.

initially, color (node $\in T$) $\in$ {Red, black}

node $\in T$: any node in T tree

↓ after changing root

color (root) = black

color ($\nexists$ node $\in T' \setminus$ {root}) $\in$ {Red, black} (Taking union)

(as others do not change) $\therefore$ color (node $\in T'$) $\in$ {Red, black}

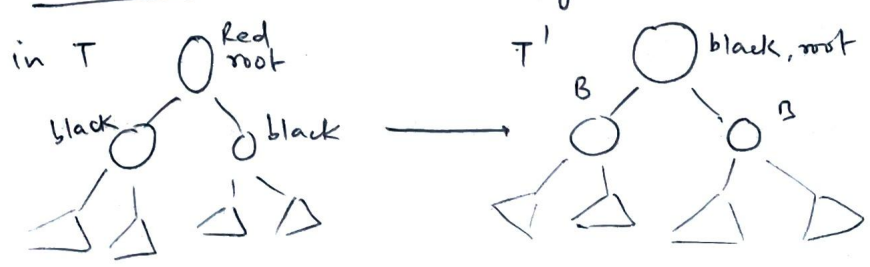
$\therefore$ every node in T' is of color either red or black, property retained

(b) checking :- every leaf (NIL) is black.

same logic as above, only the node gets modified, the NIL do not get modified, so they retain their color.

$\therefore$ every leaf (NIL) is black, property retained

(c) checking :- red nodes only have black children



here the only change is at the root

so, let us see at the root itself as everywhere else, nothing changes

at root :- black root has 2 (or less) black children

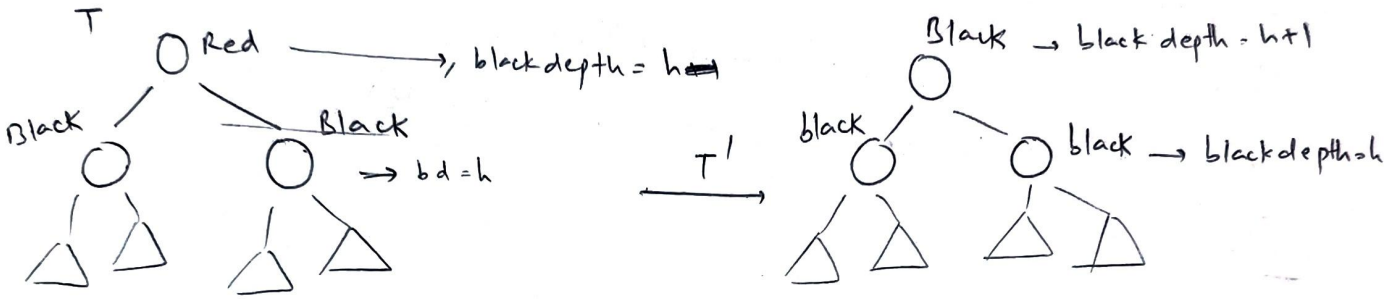
So no violation of property. (vacuously true).

at nodes other than root :- whenever there is a red node other than root in T, it has only 2 black children.

and in T', everything except root remains the same so, at every red node in T', it has only 2 black children.

∴ red nodes have black children, property retained.

(d) checking equal depth from every node



we will check the property for all the nodes.

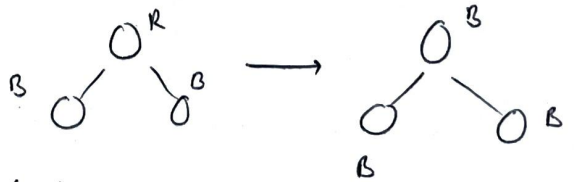
Cases → root node
→ not a root node

(a) if it is not a root node :- then everything will be same as before, so no change in the black depths of any non-root node

(b) root node

with same reasoning in (a), there is no change in any non-root node, we will check black depth of root node only

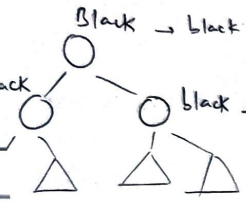
T → T' , root : red → black



as black depth of both children of root = d (from tree T),
so black depth of new root node = d+1 (as itself is black)

reason :- any path from a child of root → NIL = d black nodes.
so making new path by appending the root, we see,
any path from root → NIL = d+1 black nodes.
∴ property (d) is satisfied in T'

e is a red node other than
en.
nains the same so,
_ black children.
property retained.



_l the nodes.

thing will be same as
ths of any non-root node

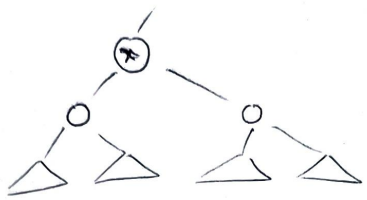
o change in any non-root
ot node only

B

(from tree T),
→ itself is black)
d black nodes.
we see,

∴ all conditions of red black tree are satisfied ∴ T' is a Red Black tree

2. Red black tree



Let Black depth of x = b.
then it means there are b black nodes in the path.

length of path :-
is minimum when the path has only black nodes.

$l_{min} = b$

$\therefore l \geq b$ (l - any general path length)

length of path is maximum when black and red are alternately placed in the path, $n = b + r$ (and $r \leq b$, as each red node has only black children but black can have any children)

$l_{max} = 2b$

$\therefore l \leq 2b$ (l - any general path length)

from a node x of black depth = b,

longest path length ≤ 2 (shortest path length)

← longest path length ≤ 2B
shortest path length ≥ B.
B ≤ shortest path length

Hence proved

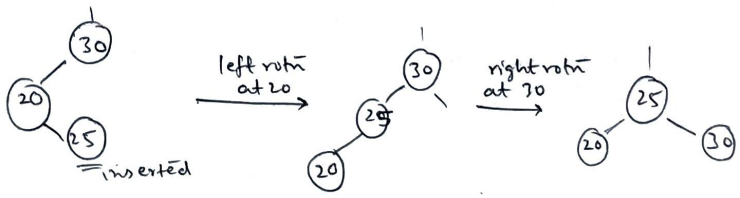
3. Double rotation is needed in insertion when insertion happens at :-

(i) insertion to the right of left subtree (LR)

ie, Balance factor (x) > 1 and balance factor (x → left) < 0,

where balancefa
= height (x → left
- height (x → ri
= BF(x)

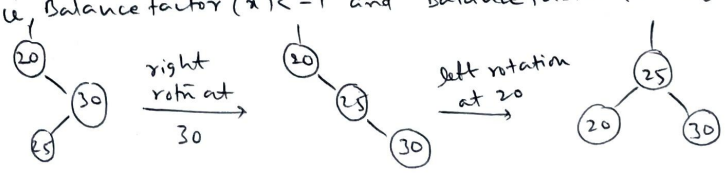
Ex



even on single rotation, it is unbalanced.

(ii) insertion to left of right subtree (RL)

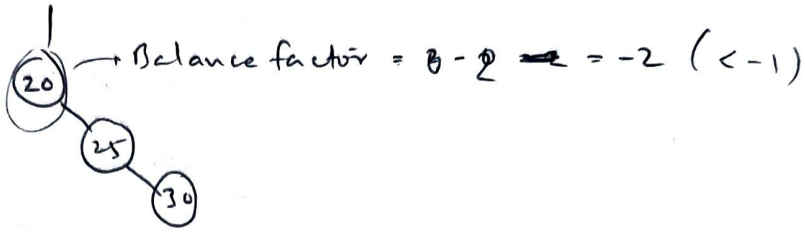
ie, Balance factor (x) < -1 and Balance factor (x → right) > 0



here also even aft
first rotation, it i
unbalanced.

why it is unbalanced even after single rotation?

lets see previous diagram, after single rotation



and in first case, it was

∴ it is unbalanced

∴ In these 2 cases, we need to do double rotation as single rotation doesn't suffice.

(5) Problem of augmeting Red - black trees :-

we need a psendocode for an algorithm for RB-ENUMERATE (x, a, b) and implements this algorithm in $\Theta(m + \log n)$, m = no of keys outputed n = size of BST (internal nodes), its task is to output all keys k such that $a \leq k \leq b$, in Red-black tree with root = x.

Psendocode

```
RB-ENUMERATE (x, a, b):
        v := TREE-LOWER-BOUND (x, a)        // first key w/ k ≥ a.
        while v != NIL and v.key ≤ b
                output v.key
                v := TREE-SUCCESSOR(v)

TREE-LOWER-BOUND (x, a)
        v := x
        cand := NIL
        while v != NIL
                if v.key ≥ a
                        cand := v
                        v := v.left
                else
                        v := v.right
        return cand.
```

Intuitive explanation : basically we traverse down and left till we get the node that is just greater than a and then, keep outputting the successor of that part (immediately greater one) till we reach b.

Correctness

Required to prove

- TREE-LOWER-BOUND (root, a) — returns smallest key k such that $k \geq a$ or NIL if it doesnt exist
- starting from v = TREE-LOWER-BOUND (x, a), taking successors outputs exactly all k such that $a \leq k \leq b$, in sorted order, and then stops.

Loop-invariant :- @ start of each while loop, cand = NIL or cand.key is the smallest key $\geq a$ so far., and subtree rooted at V. contains all remaining nodes other nodes which may have $k \geq a$ than our cand.key

Initialization :- V = root, cand = NIL, ∴ vacuously true.

Maintenance :- if V.key $\geq a$, → V. is a candidate to be least $k \ni k \geq a$.

so we set cand := V, and continue the search. as there might exist a smaller $k \ni v \geq k \geq a$, in V.left, if not, check in V.right

thus invariant holds

~~Ter~~ Termination :- if when V becomes NIL, there's no more remaining nodes to search, and cand. is therefore the smallest key in tree $\geq a$., so TREE-LOWER-BOUND. is correct.

Correctness of main loop :- let V0 = TREE-LOWER-BOUND (x, a)

if V_0 = NIL, loop terminates

if $V_0 \neq$ NIL, then BST successor prints a chain of successors starting from V_0, succ(V_0), succ (succ(V_0)) ... → is a strictly increasing order.

and these are outputed as long as $k \leq b$., so we are only outputs keys $k \ni a \leq k \leq b$.

* no key $u \ni a \leq u \leq b$ will be missed., because it will be outputed as the successor of the ~~dd~~ largest node smaller than u, and all will be covered, and each one gets printed only once

thus output = $\{k : a \leq k \leq b\}$

this proves correctness of algorithm

Proved

Time complexity analysis in next page.

Time complexity analysis

1. Time for TREE-LOWER-BOUND → each iteration goes one level down the tree. So → $O(h)$, ~~h=logn~~ and $h = O(\log n)$

$\therefore O(\log n)$

2. Time for printing successor sequence till b :-

each iteration = constant time printing + call to successor function

TREE-SUCCESSOR(v) :-

- if v.right != NIL, go v := v.right and go left
- if v.right == NIL, ascends up via parent pointers.

→ consider all edges (parent-child) traversed, from v0., for m outputs

* each edge can be traversed at most twice.

Reason:- edge is traversed down only if we descend down to node to get min of right subtree, but never after that.

edge is traversed up only if we backtrack where right subtree has been exhausted

$\therefore$ max = twice.

$= O(m+d)$ m = outputs in sequence

d = edges traversed

but we already have $O(h)$ = traversing down during initial search (TREE-SUCCESSOR) which already covers some of these traversals

$= O(m+h)$

$= O(m+\log n)$

lower bounded by $\Omega(m+\log n)$, minimum m for outputting and traversing down in first step

$\therefore$ algorithm is $\Theta(m+\log n)$