Red-Black Trees

Wednesday, 27 September 2023 5:42 PM

For Data Structures, so far we've seen:

(i) Min heaps, through Fibonacci Heaps
    supporting Extract-Min $O(\log n)$ & Decrease-Key $O(1)$

(ii) Union-Find
    supporting Union $O(1)$ & Find $O(\log^* n)$

Will look at a basic DS used in maintaining a database, where each entry has a unique <u>key</u>.

Typical operations we need to support are:

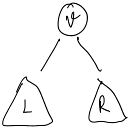
Find (x), Insert (x), Delete (x)

We want to do these in $O(\log n)$ worst-case time.

Perfect binary tree: each node has either 0 children (i.e., is a leaf) or has 2 children (i.e., is an internal node)

Claim: Every perfect binary tree w/ k leaves has $2k-1$ nodes (i.e., $k-1$ internal nodes)

A binary search Tree is a binary tree w/ the property that : for any keys l & r in the left & right subtrees of v respectively, $l.key < v.key < r.key$



(assume that all keys are unique)

This allows us to do Find(x) in time O(ht. of tree).
Further, we can list all keys in the tree in order, in time $O(n)$:

```
in-order (x) {
    if (x.left ≠ NIL) in-order (x.left)
    print x.key
    if (x.right ≠ NIL) in-order (x.right)
}
```

by calling in-order (r), where r is the root of the tree. this is called an in-order traversal of the binary tree.

Note : for any node x, the immediate successor is the left-most node of right subtree

Red-Black Trees :

In addition to usual info (key, ptrs. to parent, left-child, right-child) each node has a "colour", red or black. Each leaf is a NIL node, each internal node is a key node

A red-black tree has the following properties:

① (a) Root is black

(b) all leaves (NIL nodes) are black

② All children of a red node are black

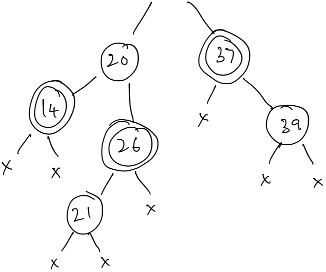
③ Any path from root to a leaf has the same number of black nodes

NOTE: ① it follows from ③ that if a node v has a single non-leaf child, this child must be red, and by ②, v cannot have any black non-leaf descendants (and hence, the red child is it's ONLY descendant).

② if the only violation is in 1(a), i.e., the root is coloured red, we can recolor the root black immediately.

Example: (throughout, double circles denote black nodes)

(x's denote leaf NIL nodes)


20
37
14
26
39
21
x
x
x
x
x
x
x
x

Lemma: A R-B tree with n key nodes has height $O(\log_2 n)$

Proof: Let l be the # of black nodes on any root-leaf path. Note that ht. of tree $\leq 2l$

Also, the binary tree is complete until ht. l

$$\Rightarrow \text{\# nodes (incl. NIL leaves)} = 2n+1 \overset{\text{by claim}}{\geq} 2^{l}$$

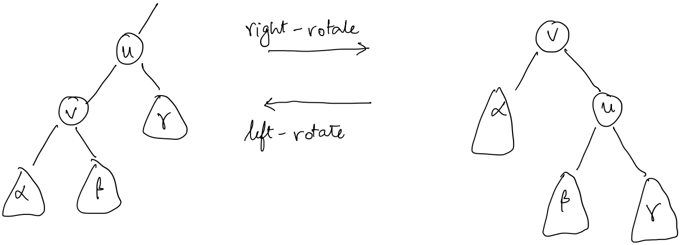
$$\Rightarrow l \leq \log_2(2n+1)$$

thus ht. of tree $\leq 2\log_2(2n+1)$ ▨

Find (x) operation is immediate.

We now consider Insert (x) & Delete (x)

A basic operation in R-B trees is rotation:

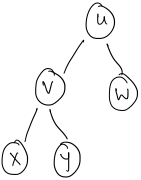

u
v
γ
α
β
right-rotate
left-rotate
v
α
u
β
γ

Note that ① Binary search tree property is maintained:

$\alpha.key < v.key < \beta.key < u.key < \gamma.key$

(ii) A rotation operation can be done in time $O(1)$

Notation :



w is v's sibling (v.s)
x is w's left nephew (w.ln)
y is w's right nephew (w.rn)
u is v's parent (u.p)
w is x's uncle (x.u)

INSERT (x) :

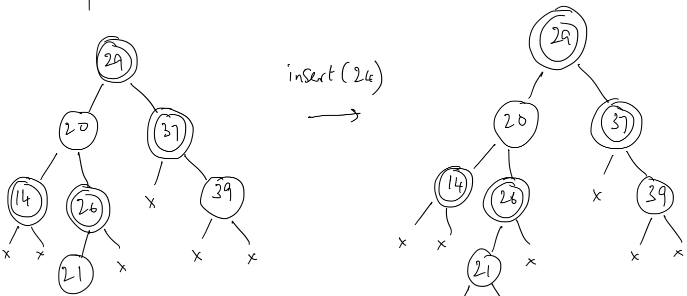
Assume Find(x) returns a pointer to leaf node where x should be.

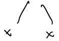
Replace this leaf node w/ a new node with key = x.

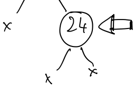
Colour it red.

This preserves all properties EXCEPT property 2, if parent is red.

Example:


29
20
37
14
26
21
39
x
insert(24)
29
20
37
14
26
39
21
x





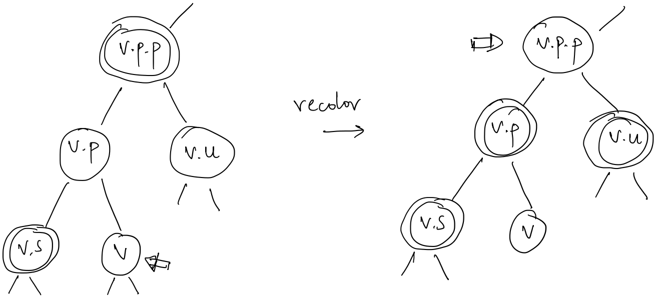
Idea: Maintain ptr to node v s.t.:

(i) v.color = red, v.p.color = red & this is ONLY violation,

(ii) either resolve violation (w/o creating any new ones), OR push v up the tree towards the root.

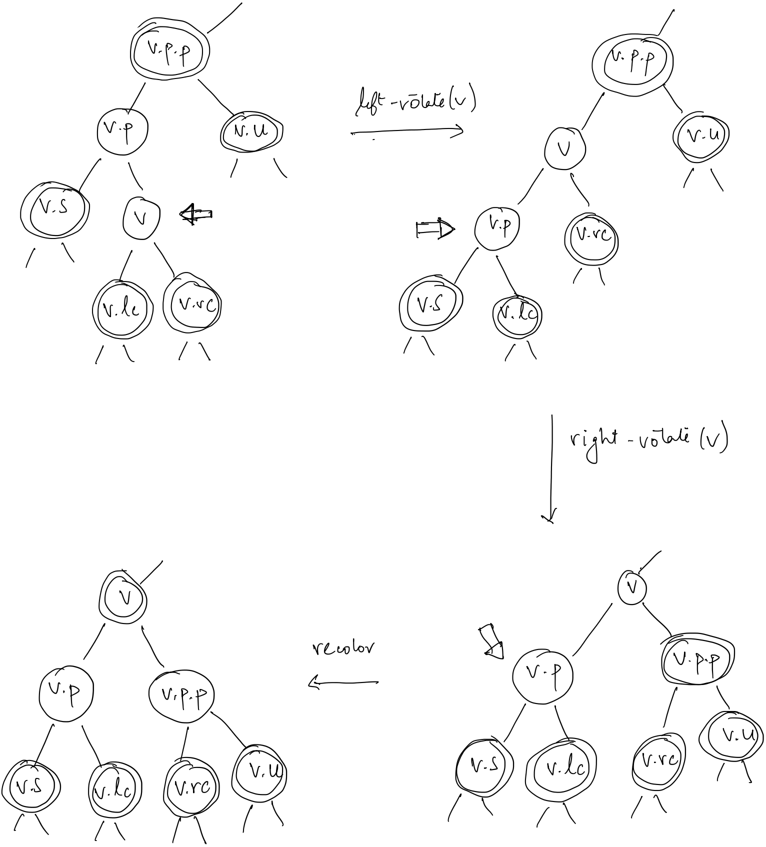
Assume now that v ≠ root, v.color = red, v.p.color = red
hence v.s.color = black, v.p.p.color = black
(and parent of parent must exist)

CASE I : v.uncle.color = red : recolor


v.p.p
v.p
v.u
v.s
v
recolor
v.p.p
v.p
v.u
v.s
v

This moves ptr. to violation up the tree

CASE II : v.uncle.color = black : 1 or 2 rotations & recolor


v.p.p
v.p
v.u
v.s
v
v.lc
v.rc
left-rotate(v)
right-rotate(v)
recolor

Note that here we are done, there are no more violations.
Also if v is left-child of parent, first rotate is not required.

This shows that we can insert a key in time $O(\log n)$.

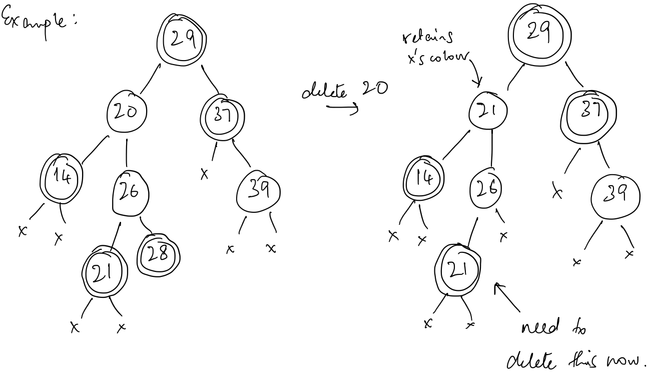
DELETE (x) :

To delete a node, we will replace it by it's in-order successor (say y), & then delete y.
Note that y is the left most node in right subtree.
If x has no right child, we replace x by its inorder predecessor (say y), & then delete y.
In this case y is the rightmost node in left subtree.
In both these cases, x's original colour is retained

If x has no children, let y = x.

Now note that y is either a non-NIL leaf, or has exactly one child.

Example:


29
20
37
14
26
39
21
28
x
delete 20
retains x's colour
29
21
37
14
26
39
21
need to delete this now.

If y has a child, then the child must be coloured red (else the tree has two paths of differing black-lengths) & hence y must be coloured black.
Here, we replace y by its child, colour it black. This satisfies all the constraints.

If y is a leaf (not a NIL node) & is coloured red, we can simply delete it.

If y is a leaf (not a NIL node) & is coloured black, the situation is complicated. Removing y would create a black-length imbalance.
NOTE: if y is coloured black, it must have a sibling.
In this case, we delete y (replacing it with a NIL node), & colour it "double-black". If this is counted as 2 black nodes, then there is no violation of the R-B tree properties.

In the remainder, we will try & resolve this, either by rotation / recoloring, or pushing up the double-black node.

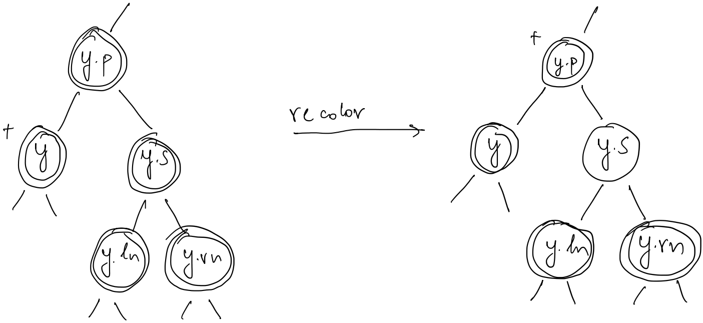
(note that in the following, the leaf NIL node, that is originally double-black, is never pushed upwards).

We use ◎$^{+}$ to denote a double-black node.

For the:

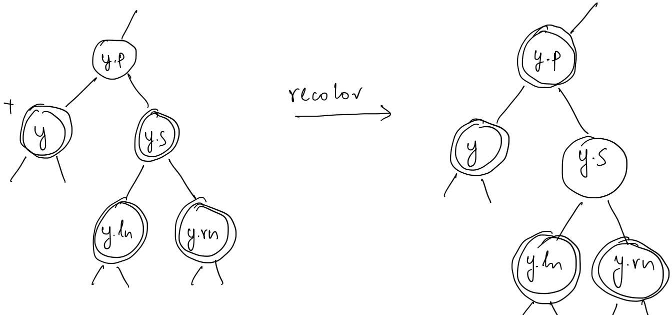
y.s → y's sibling
y.p → y's parent
y.u → y's uncle (parent's sibling)
y.lc, y.rc → y's left & right child
y.ln, y.rn → y's left & right nephew (sibling's children)

CASE I: y.s black, y.p, y.ln, y.rn all black


y.p
+
y
y.s
y.ln
y.rn
recolor
+
y.p
y
y.s
y.ln
y.rn

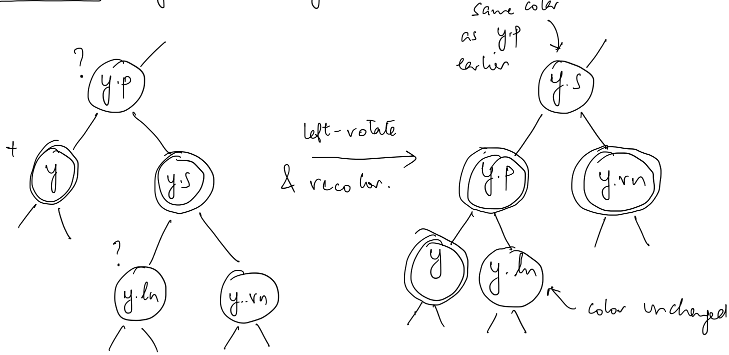
in this case, double black node is pushed up.

CASE II: y.s black, y.ln, y.rn black, y.p red


y.p
+
y
y.s
y.ln
y.rn
recolor
y.p
y
y.s
y.ln
y.rn

In this case, we stop

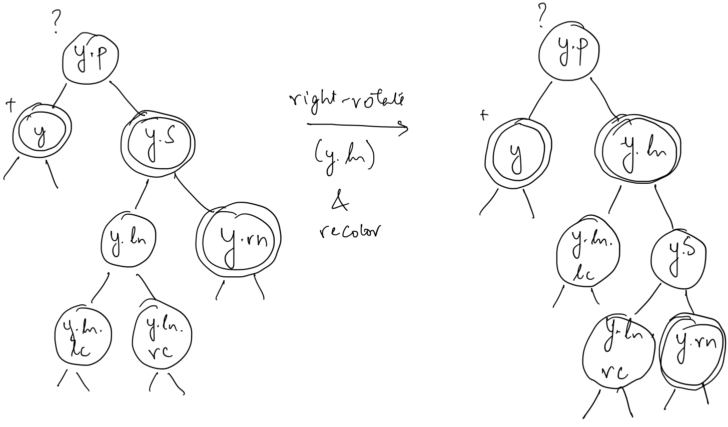
CASE III : y.s black, y.rn red



?
y.p
+
y
y.s
?
y.ln
y..rn
left-rotate
& recolor.
same color as y.p earlier
y.s
y.p
y.rn
y
y.ln
color unchanged

In this case, we stop.

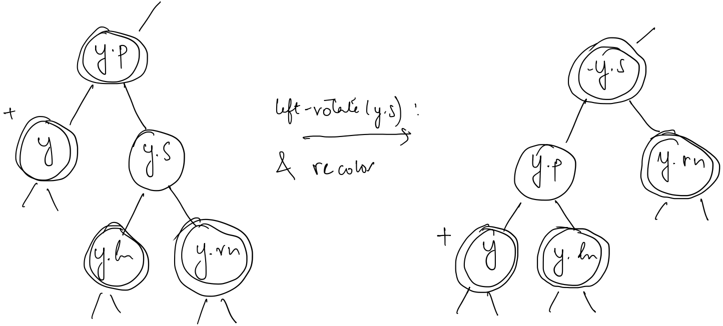
CASE IV : y.s black, y.rn black, y.ln red



?
y.p
+
y
y.s
y.ln
y.rn
y.ln.lc
y.ln.rc
right-rotate
(y.ln)
&
recolor
?
y.p
+
y
y.ln
y.ln.lc
y.s
y.ln.rc
y.rn

Now we are in Case III : y.s black, y.rn red

CASE V : y.s red (in which case, y.ln & y.rn are black)


y.p
+
y
y.s
y.ln
y.rn
left-rotate(y.s) :
& recolor
y.s
y.p
y.rn
+
y
y.ln

now y.s is black, and we are in one of the previous cases,

This completes the description of the delete opn.