Balanced Search Trees   CLRS Ch 13, Chap 18

[redacted] (mistake on [illegible])

Next: we discuss data structures that build approximately balanced binary trees

- Red black trees ⟵ binary
- 2-3-4 Trees, B-trees, $\overset{1970}{23}$ ⟵ not binary
- AVL trees (1962, 1st one)
- Skip Lists, Treaps ← 1996 most recent

RED-BLACK trees - all operations in O(logn) time

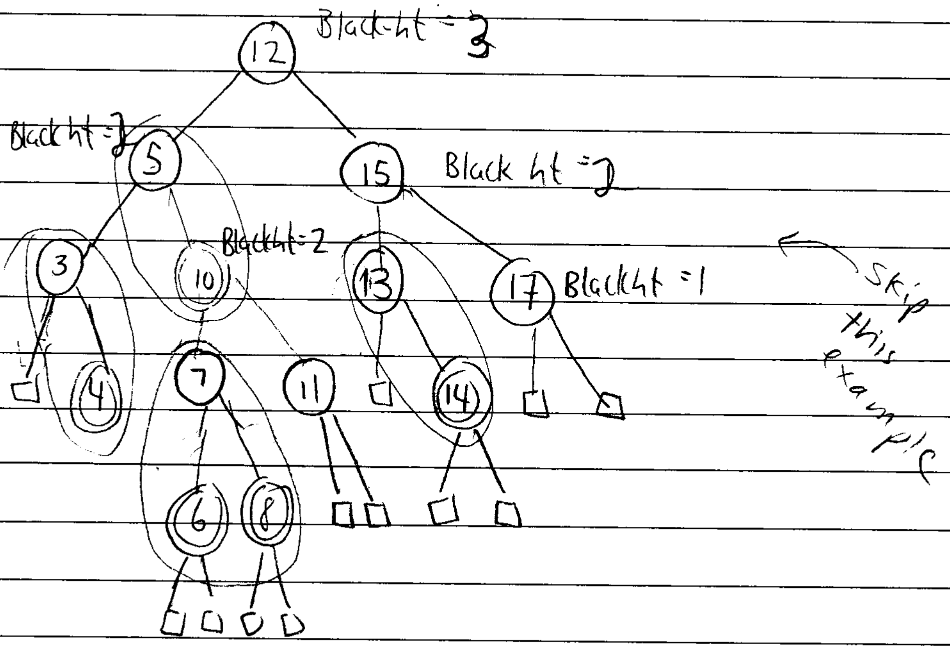
A BST is a red-black tree if it satisfies the following properties

1. Each node is red or black
2. Root is black
3. Leafs (NIL) are black
4. Parent of every red node is black (so no 2 consec. reds)
5. ∀ nodes, all paths from node to leaf have same # of black nodes

Black-ht(x) = # black nodes on any path from but not including x to a leaf

All internal nodes have exactly 2 children, leaves no children

Example

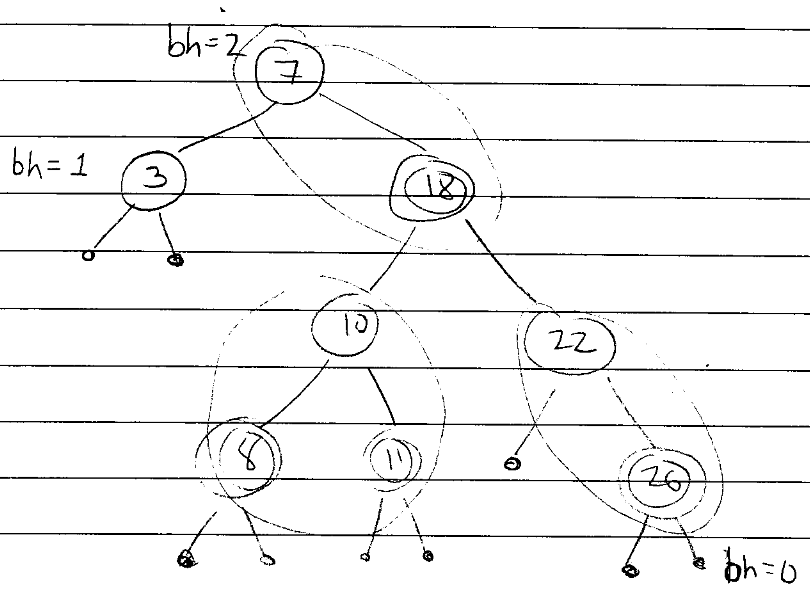


Skip this example

☆ the properties force the tree to have log ht, and make properties easy to maintain

Tricky to maintain properties as we do updates.

Ex.

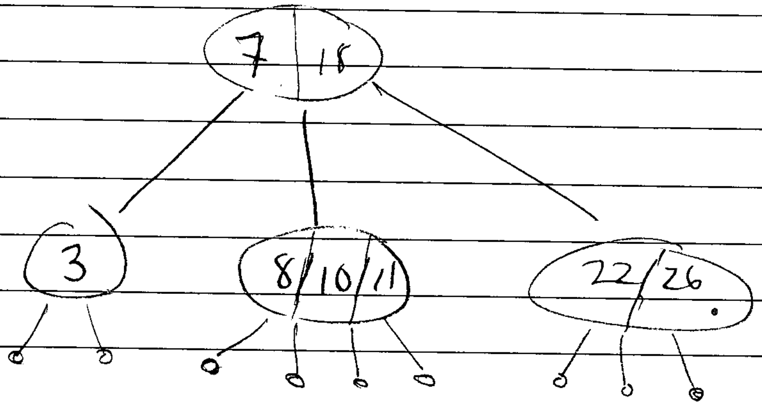


Valid binary search tree
inorder traversal is sorted ◎ is red

Theorem any red-black tree with n nodes has ht $\leq 2\ln(n+1) = O(\log n)$

Pf in book by induction
Pf here more intuition.

Take red nodes & merge with their parents
in our example we get:



This is a 2-3-4 tree. Each parent has 2, 3 or 4 kids
(Foreshadowing of what is to come)

- In a 2-3-4 tree all leaves have same depth (= bh of root)
- Every internal node has between 2 & 4 children

Number of leaves in a <u>binary</u> tree with
n internal nodes is $n+1$

Also $n+1$ leaves in the 2-3-4 tree. (because we didn't change the # of leaves)

A 2-3-4 tree of height h' has between $2^{h'}$ and $4^{h'}$ leaves.

so $2^{h'} \le n+1 \le 4^{h'}$

$h' \le \log(n+1) \le 2h'$

Now relate h (ht of Red-black tree) to h'.
Using property 4, we know $h \le 2h'$ $\quad h' \le h \le 2h'$

so $h \le 2h' \le 2\log(n+1)$.

Later we'll see how to manipulate 2-3-4 trees directly

Today we'll do red-black trees.

<u>Recap</u>: we know red-black trees always balanced

If we maintain red-black tree properties all operations will be efficient: $O(ht) = O(\log n)$.

Ordinary insert/delete on BSTs will not satisfy red-black properties. Updates must therefore modify tree to preserve red-black properties.

<u>Types of operations we'll do</u>

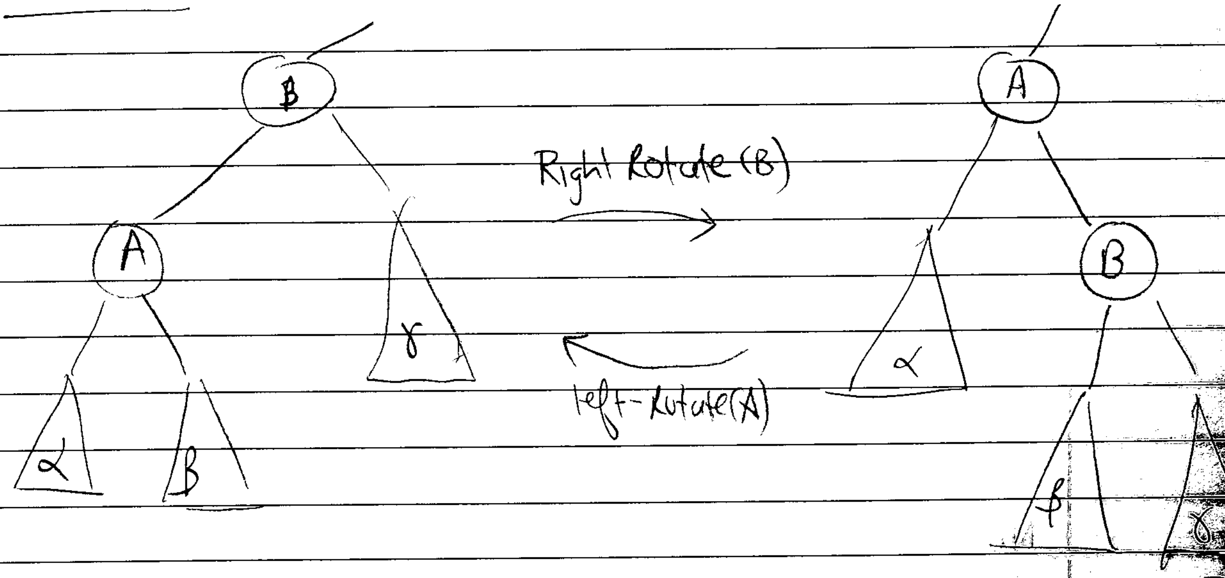
BST operation

Color changes

Change links via <u>rotations</u> (this is the only way we'll change links)

Search will BST search.

<u>Rotations</u>



check: it preserves BST property.

ie ∀ nodes v, nodes in left subtree all ≤ v

$a \in \alpha$ $b \in \beta$ $c \in \gamma$

in left tree $a \le A \le b \le B \le c$

same in rt tree.

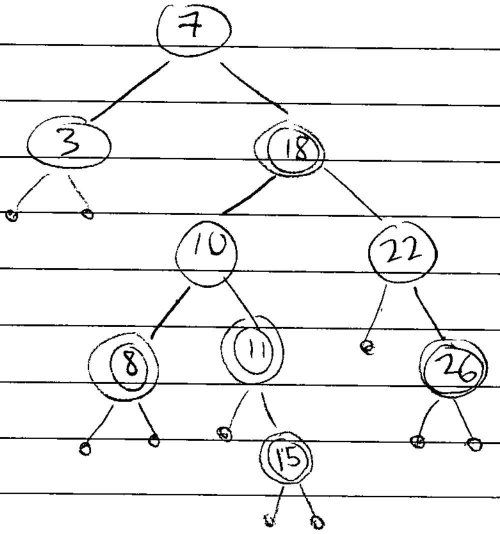
- Reversible operation.
- So LeftRotate(A) is opposite

- constant time to perform a rotation

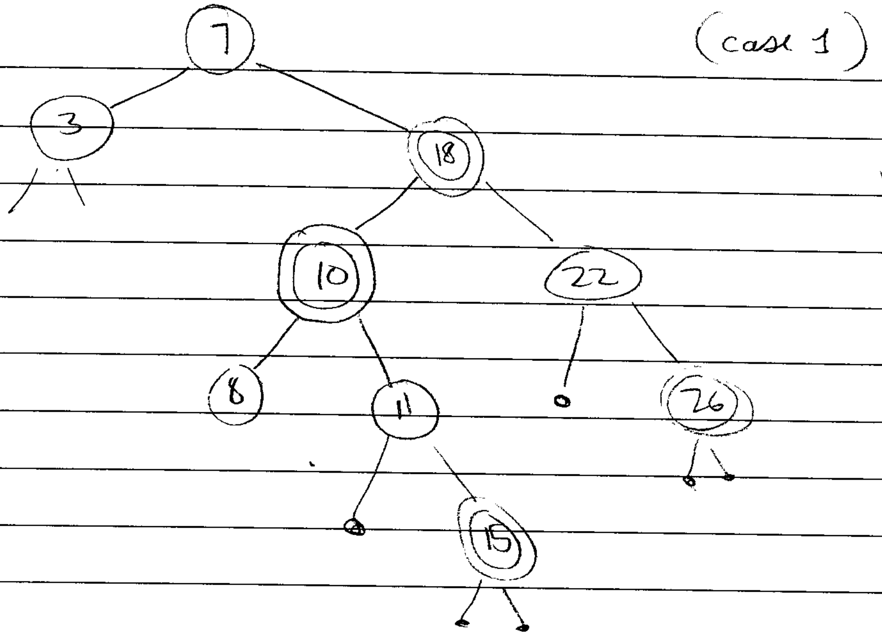
INSERTION (RB-insert)

Idea:
- Treeinsert(x)
- Make x red
- Problem if x's parent is red (Prop. 4 may be violated)
- But bh property still holds
- Move violation of prop 4 up the tree via recoloring until we can fix violation using a rotation (plus recoloring)

Example insert 15



violate prop 4.
Can't make it black since it screws up bh property.
Try to recolor
Look at grandparent of 15 –
it is black with 2 red children
turn it into red with 2 black children.

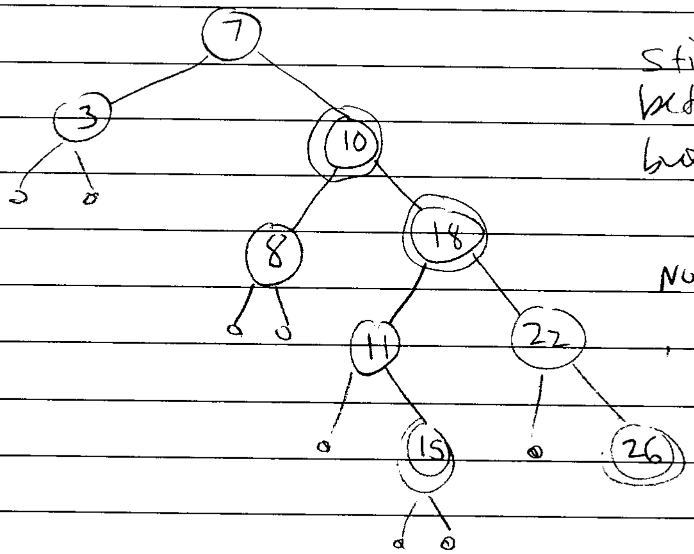


(case 1)

violation moved up the tree — between 10 + 18

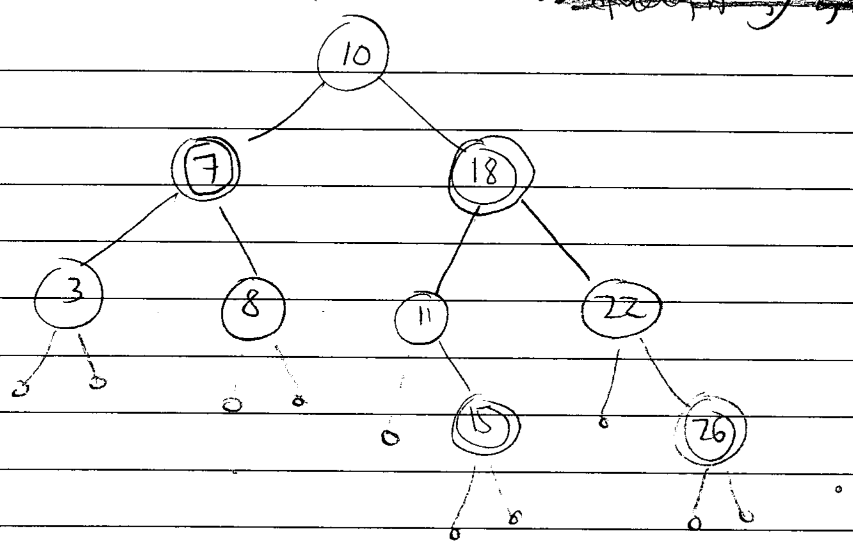
Look at grandparent of 10.
Cant do same trick since grandparent has one red, one black child
Instead do a rotation —
rotate 18 to the right

Rotate - Rt (18)



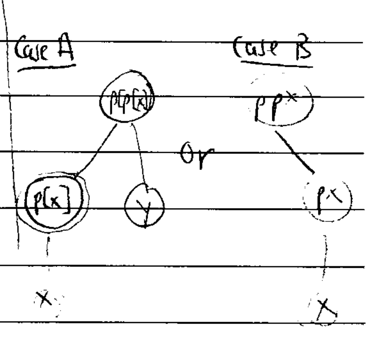
still a violation between 10 + 18
but now it is straight
(7-10-18)
Not zig-zagged

Now RotateLeft (7) + recoloring



<u>General Algorithm</u> (not terribly intuitive, but can figure it out if you restrict to rotations/recoloring)

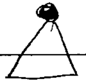
Try to recolor - pushing up tree.
If we get stuck, rotations work (at most 2)



p[x] parent
pp[x] grandparent

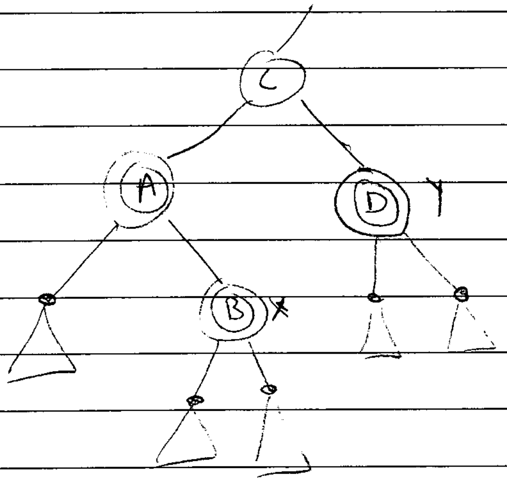
```
RB-insert (T, x)
    Tree-Insert (T, x)
    Color (x) ← Red
    while  x ≠ root[T] and color[x] = red
        If p(x) = left[p[p[x]]]   (Case A)

            y ← right(p[p[x]])        y is uncle/aunt
            If color(y) = red
                then CASE A1
            Else if x = right(p[x])
                then case A2
                case A3

        Else p[x] = right(p[p[x]])   (case B)
            reverse left & rt above

    Color [root[T]] ← BLACK
```

3 cases


means this subtree has a black root

all △ have same black ht

Case A.1 (Case A.1)



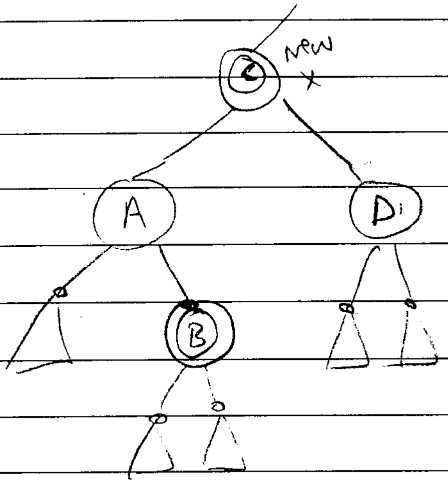
key D, node y

since A, B, D all red,

△'s all have same black ht

in 2-3-4 tree this has 5 children which is bad.

make A, D black C red



~~New x = C~~

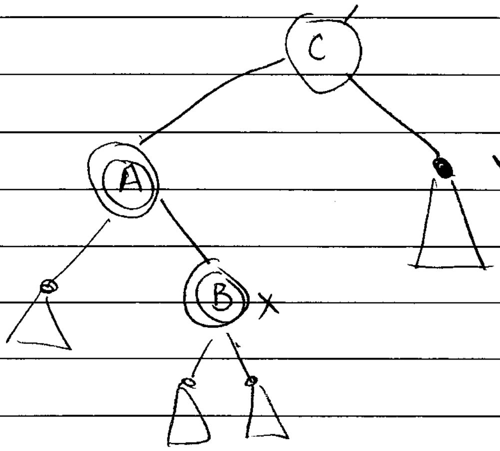
Now C may violate prop. 4

C is higher up.

Continue with x = C.

Note x can be on left or rt of A – doesn't matter.

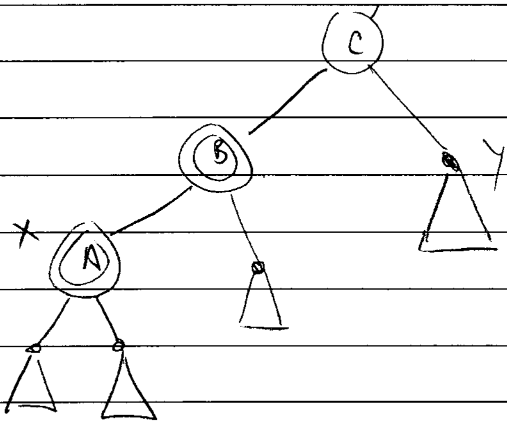
Case A.2



x is right child of A

2-3-4 .
only 4 children

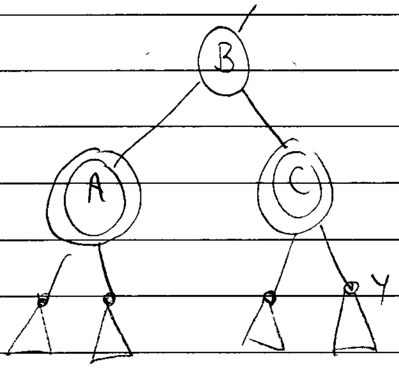
Do left-rotate (A)



b4
Zigzag between x
and grandparent
NOW zigzig

Case A3

↑ picture is as above
Do Right-Rotate (c), then recolor



No violations!
(check for hw)

Recap case 1 - recolor case 2 - ZigZag case 3 - ZigZig Final rotation/

Runtime

Cases 2 + 3 are terminal
(Case 3 - done)
(Case 2 → immediately case 3)
Case 1 repeats only logn times

So total time is O(logn)

Also - only O(1) rotations. Most changes are recolorings. So very fast in practice.

Also can do other stuff (search queries) during the recolorings
Rotation more expensive since nodes have to be locked during rotations

DELETION - more complicated.
see book

END lecture 2

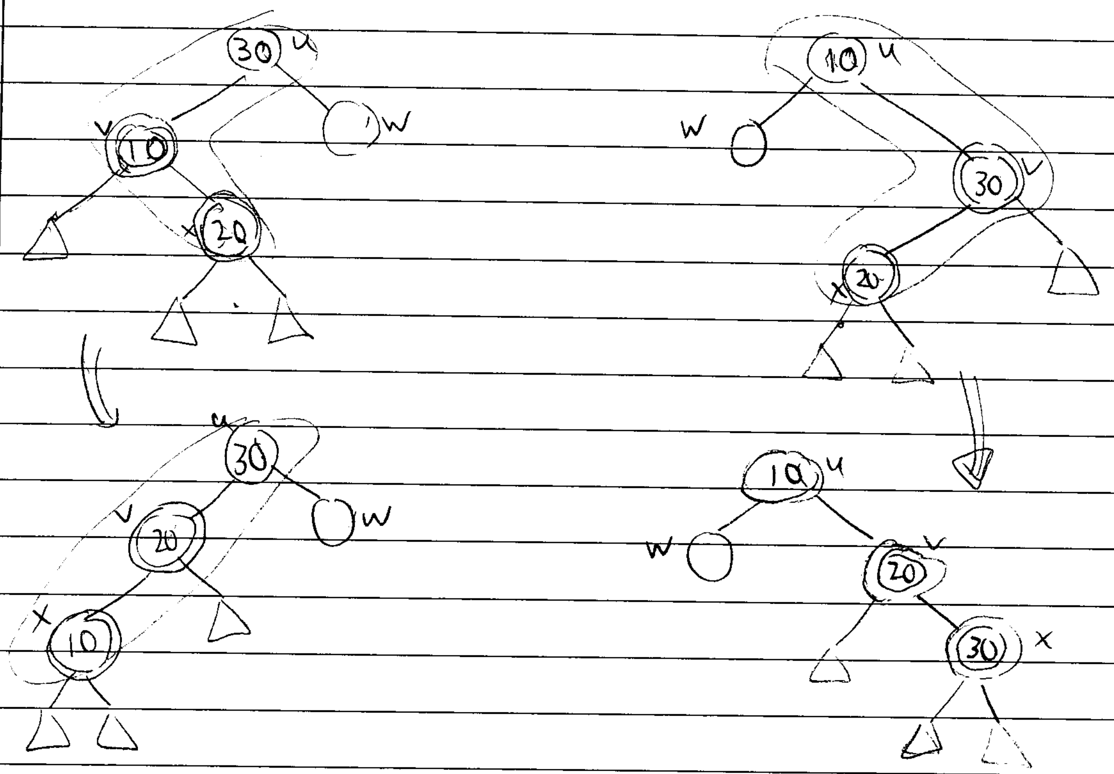
Tutorial 2: Deletion in Red/Black Trees

Midterm Q: 2-3 trees ↔
modified Red/Black
only Rt red. So no recolorings?

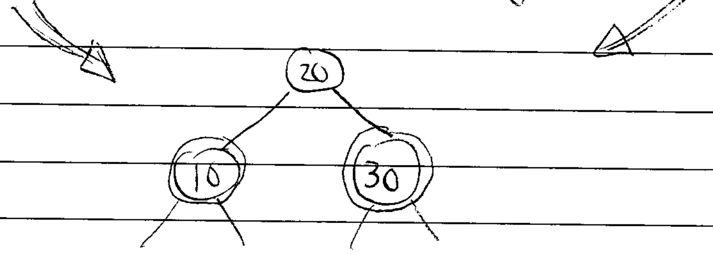
Another view

1. grandparent of x has a black child, w. 4 cases

Eg Z.

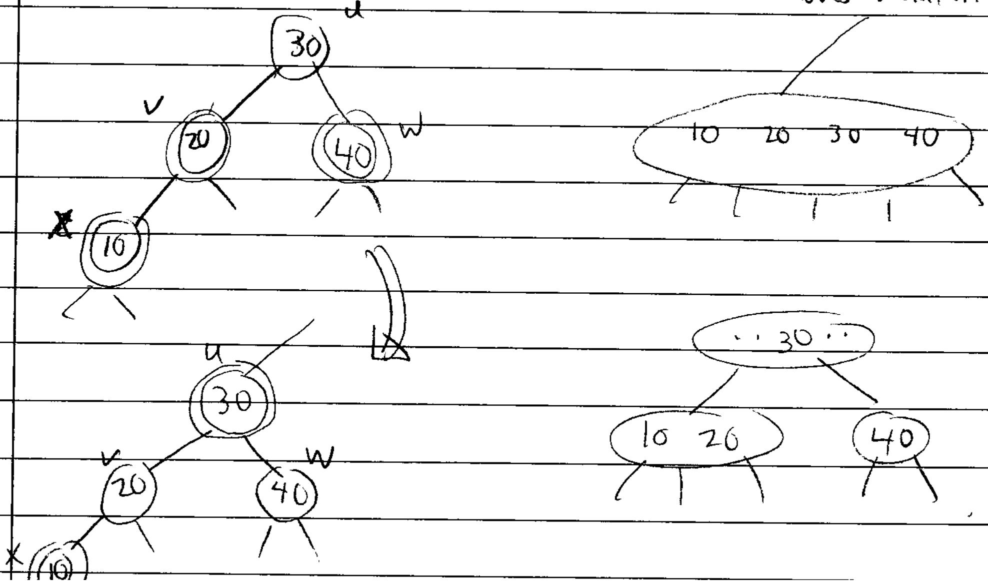


Restructure (1 or 2 rotations) to get

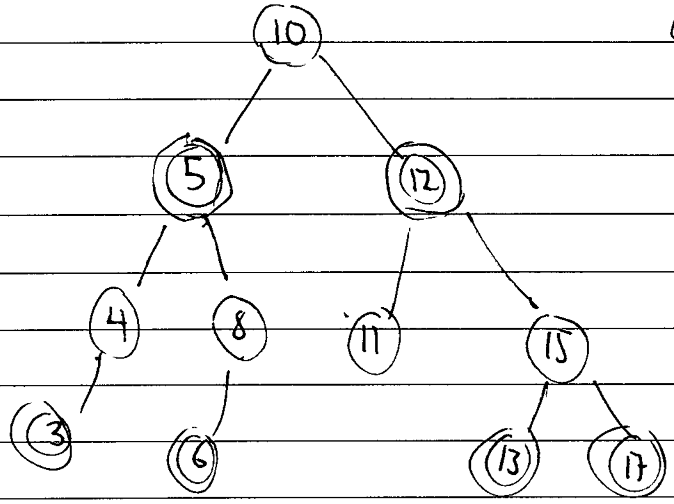


2. grandparent of x has 2 red children. Recolor.

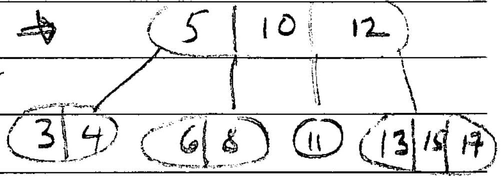
Eliminates violation or moves violation up tree ← equivalent to a split op

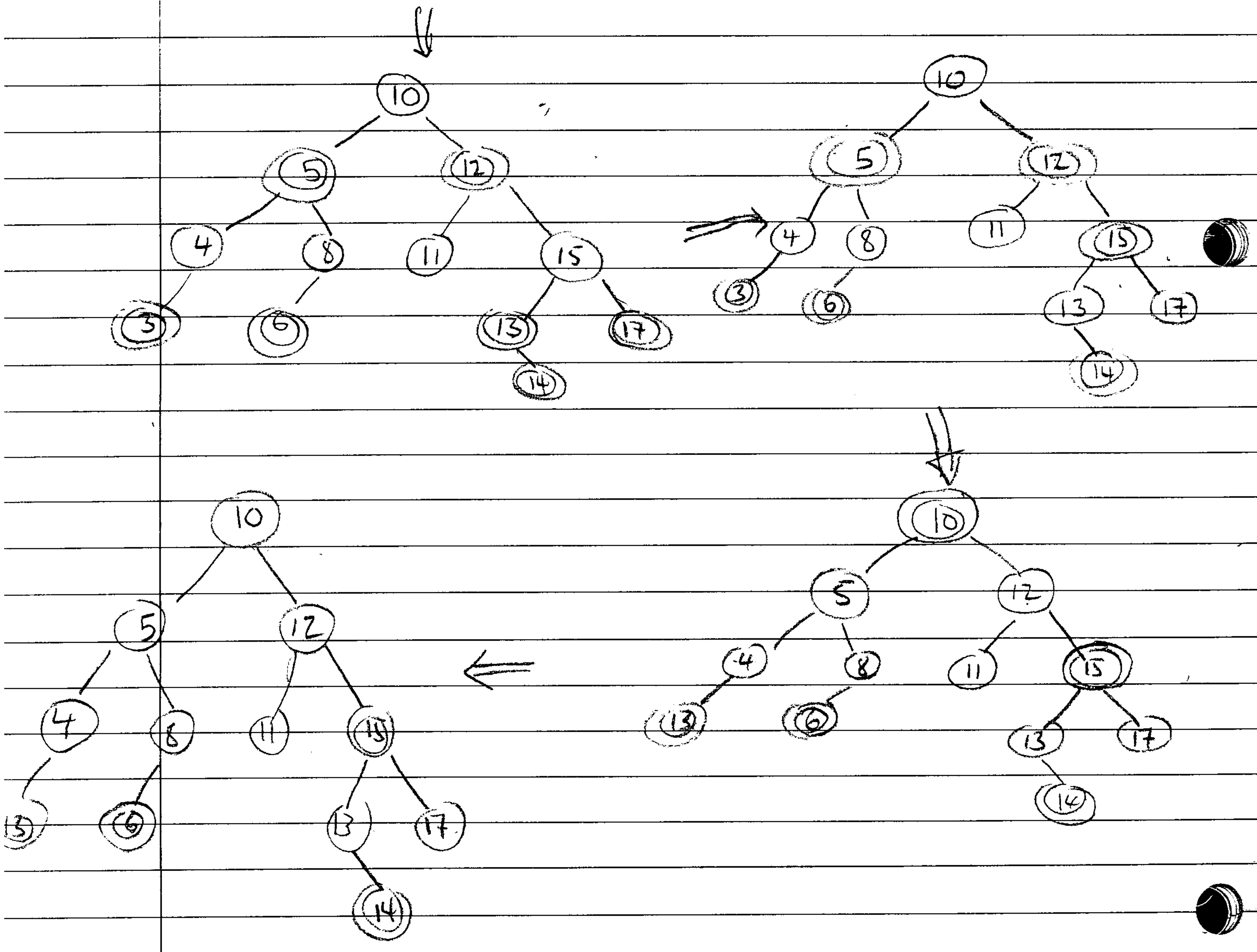


Example Red Black Insert (14)



corresponds to this 234 tree →

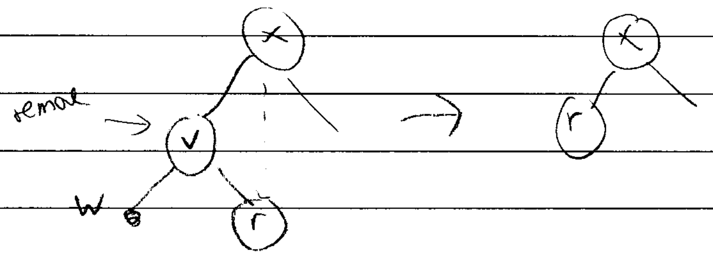




Deletion in Red-Black Trees

Do Binary search to find item $\overset{u}{}$ to be removed

If u does not have an external child, find immed successor v, swap u & v, + then remove u (which now has an external child) ← Same as BST delete



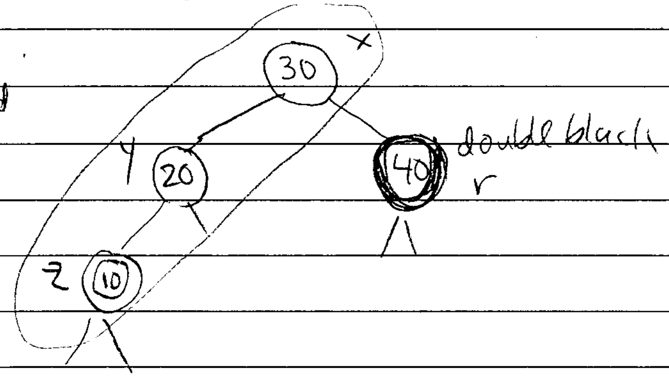
- If v was red (then r black) - done
- If v black and r red, make r black - done
- OW v black, r black. Make r double black (creates underflow in corresp 2-3-4 tree)

To remedy double-black, there are 3 cases

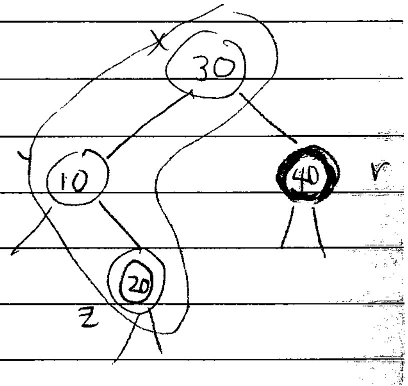
$\boxed{\text{Case 1}}$ Sibling y of r is Black + has red child z. Eliminates violation

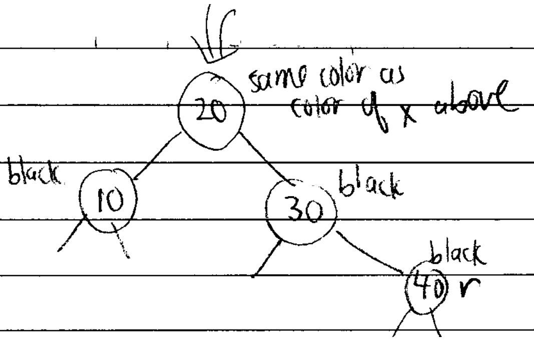
x can be black or red
y black
z red
r dbl black

rotation recoloring



or

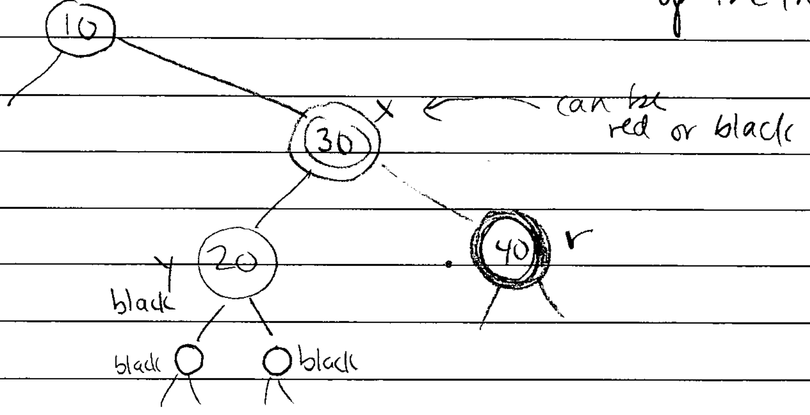




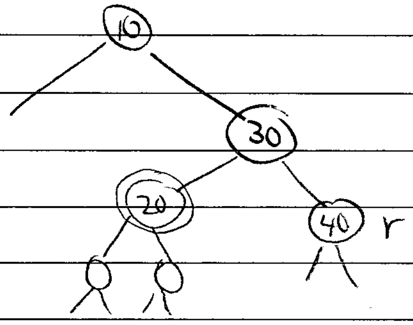
Case 2 Sibling y of r is black, both children of y are black

eliminates dbl black or propagates it up the tree

recoloring

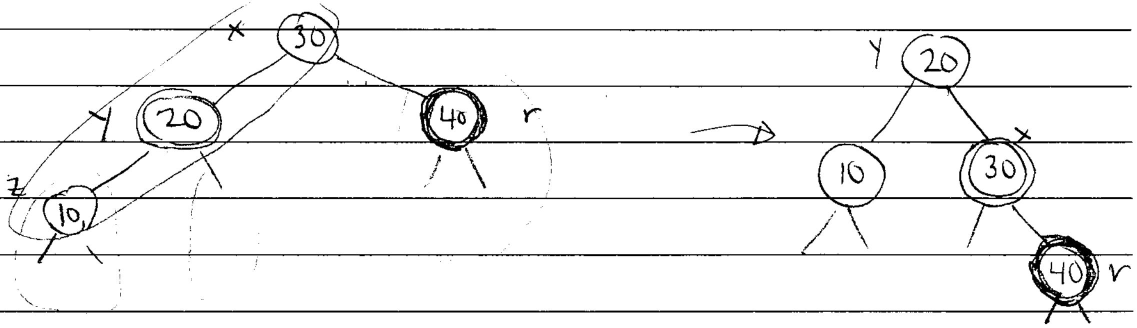


color r black
color y red
color x black if it was red, or dbl black if it was black



Case 3 Sibling y of r is red. Turns it into case 1 or case 2.

rotate
recolor



Now we're in case 1 or case 2

OR