Non-mutating methods

min, max, search, pred, successor, in-order traversal

Just ignore the color!

It's a valid binary search tree

Cost $O(h) = O(\log n)$ ⟵

In a binary search tree, time complexity to return all elements in range $[b, e]$ is $O(h + k)$ where h is the height of tree & k is # elements in given range.

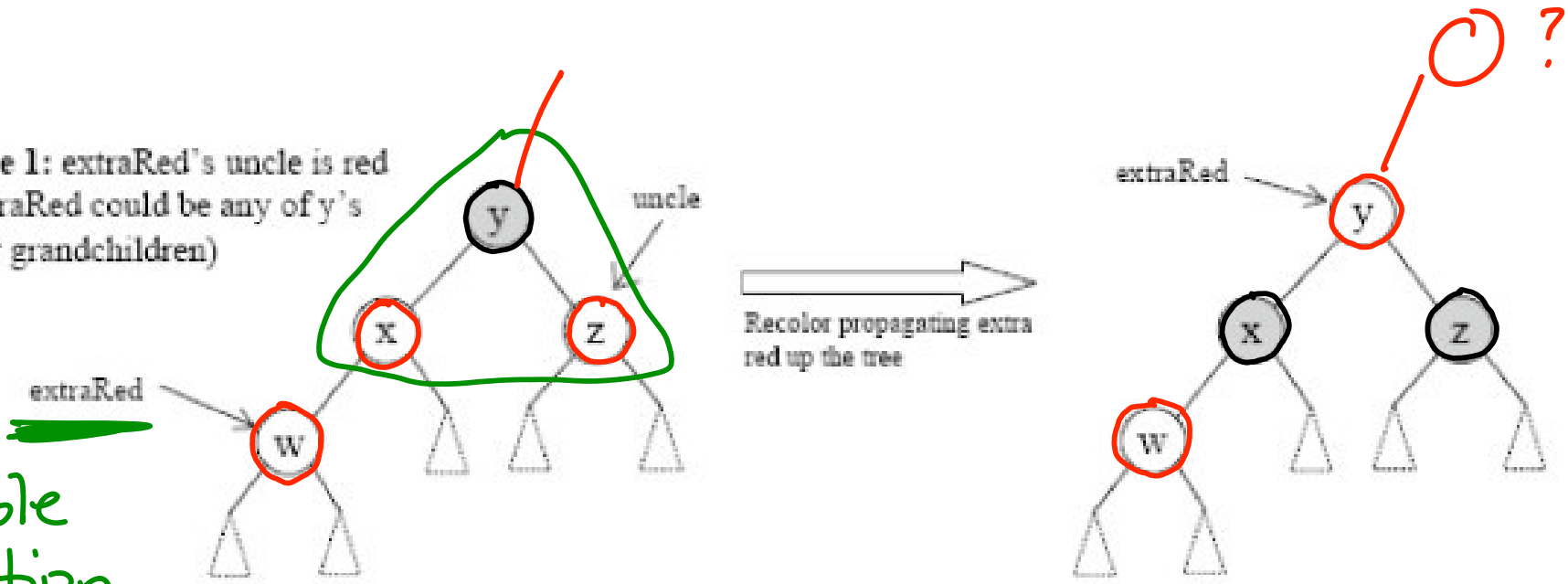
Insertion

Insert the new element in standard way as a red node.

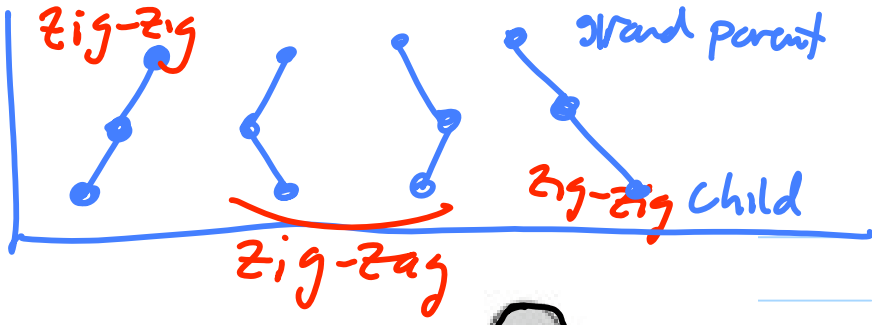
Black Balanced, NoDoubleReds } key properties to worry about

too expensive to fix if violated. Keep it preserved throughout

Take possible NoDoubleReds violation & use recoloring & rotations to move violation up towards root until fixed or reaches root.

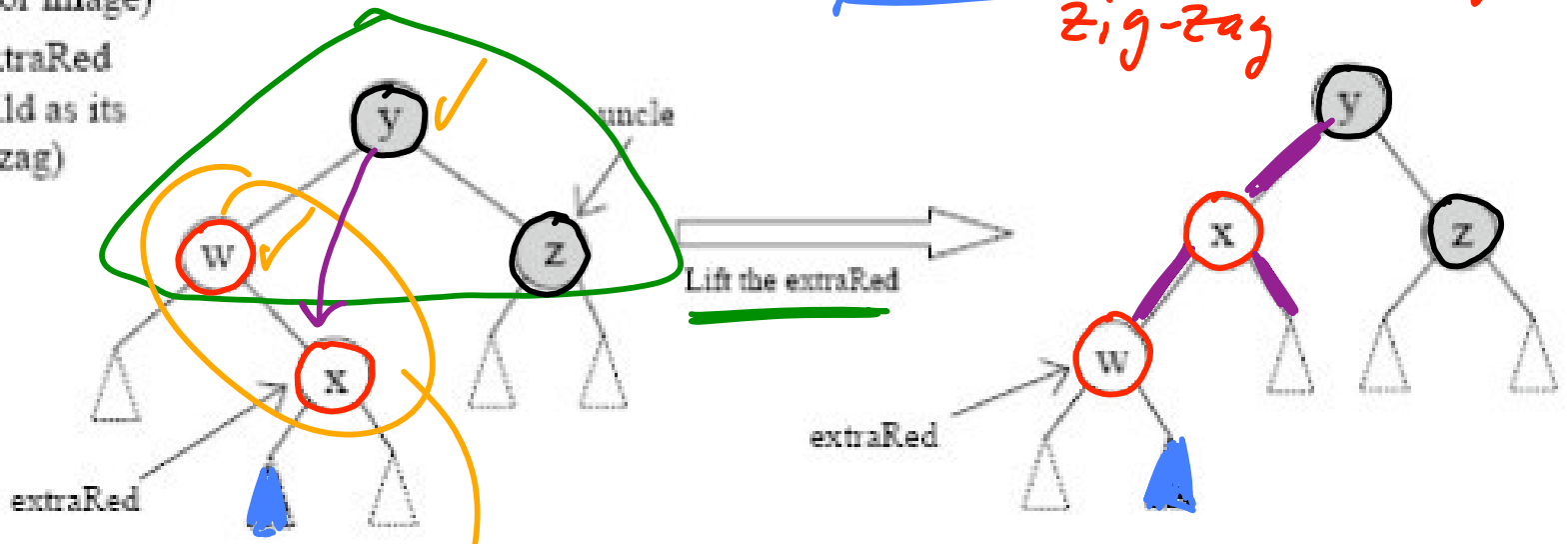


possible violation with its parent (no other violation of noDoubleRed+ BlackBalanced is preserved)



Case 2: extraRed's uncle is black (could be mirror image)

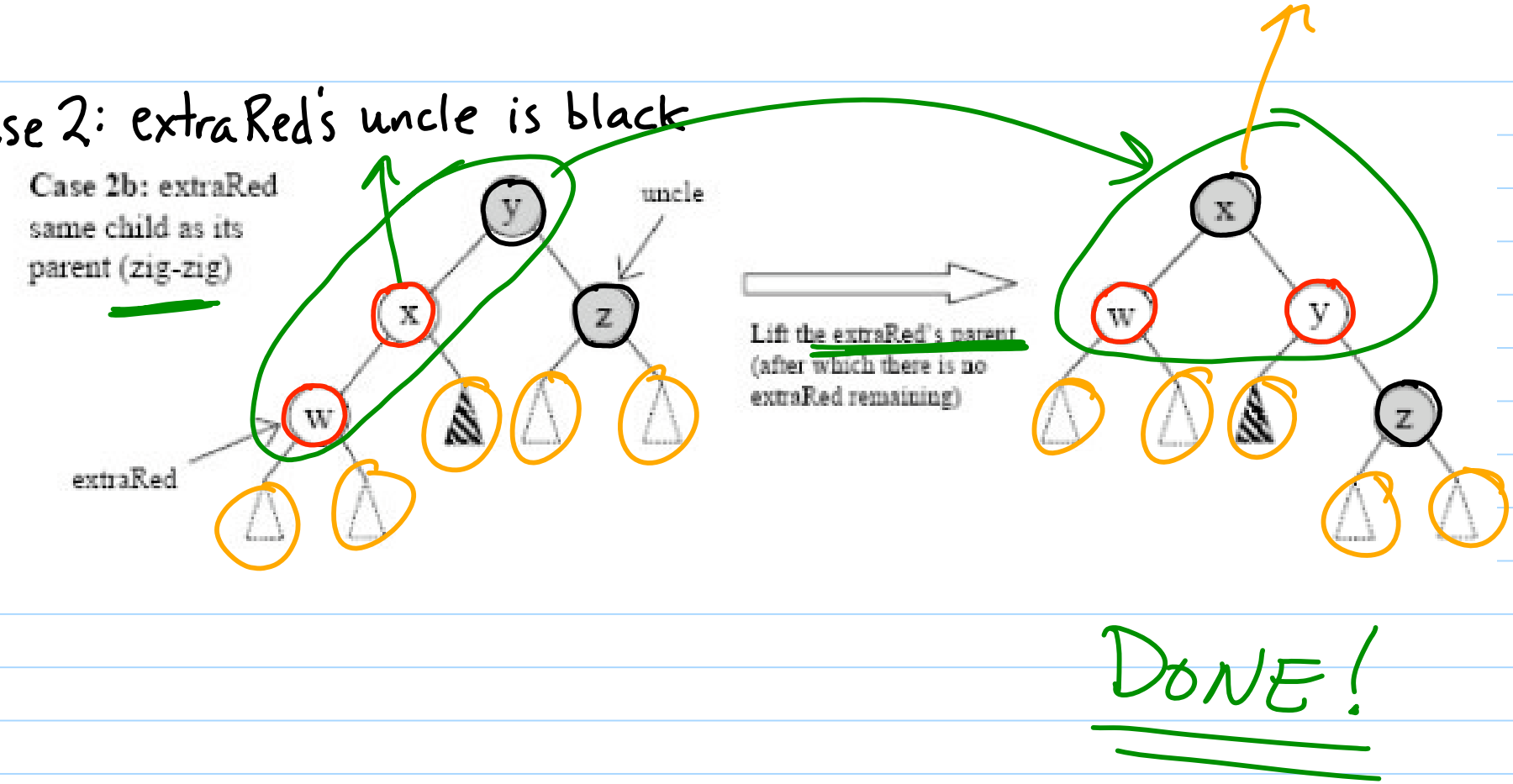
Case 2a: extraRed opposite child as its parent (zig-zag)



only violation of NoDoubleRed

still have violation (at same level of tree) but now zig-zig

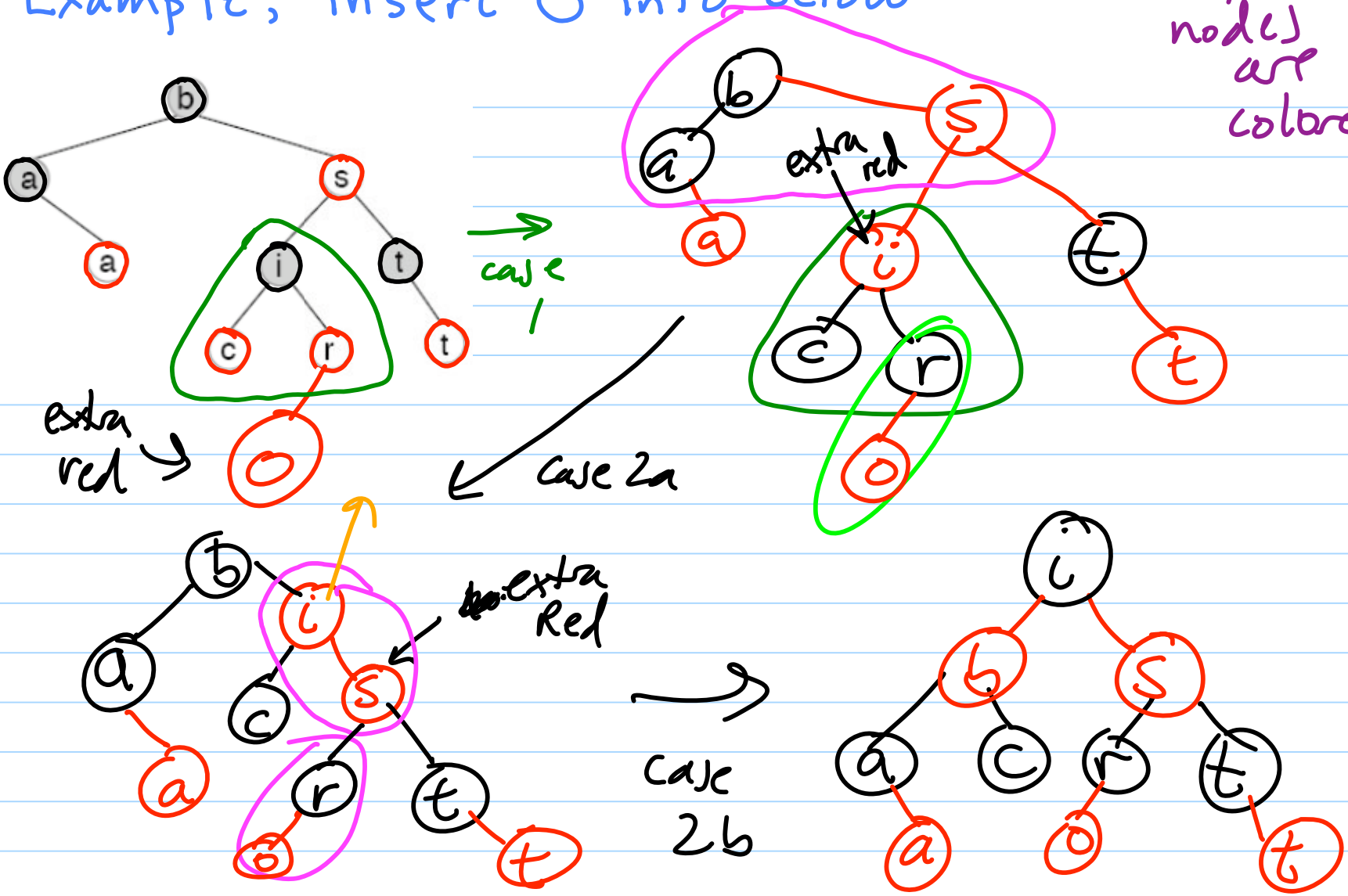
Case 2: extraRed's uncle is black



DONE!

Example, insert o into below

only nodes are colored



Time Complexity for insert

binary search tree insertion

$$O(h) = O(\log n)$$

insert fix up

≤ h/2 cases (constant time each
≤ once for case 2a + 2b

$O(h)$
$= O(\log n)$

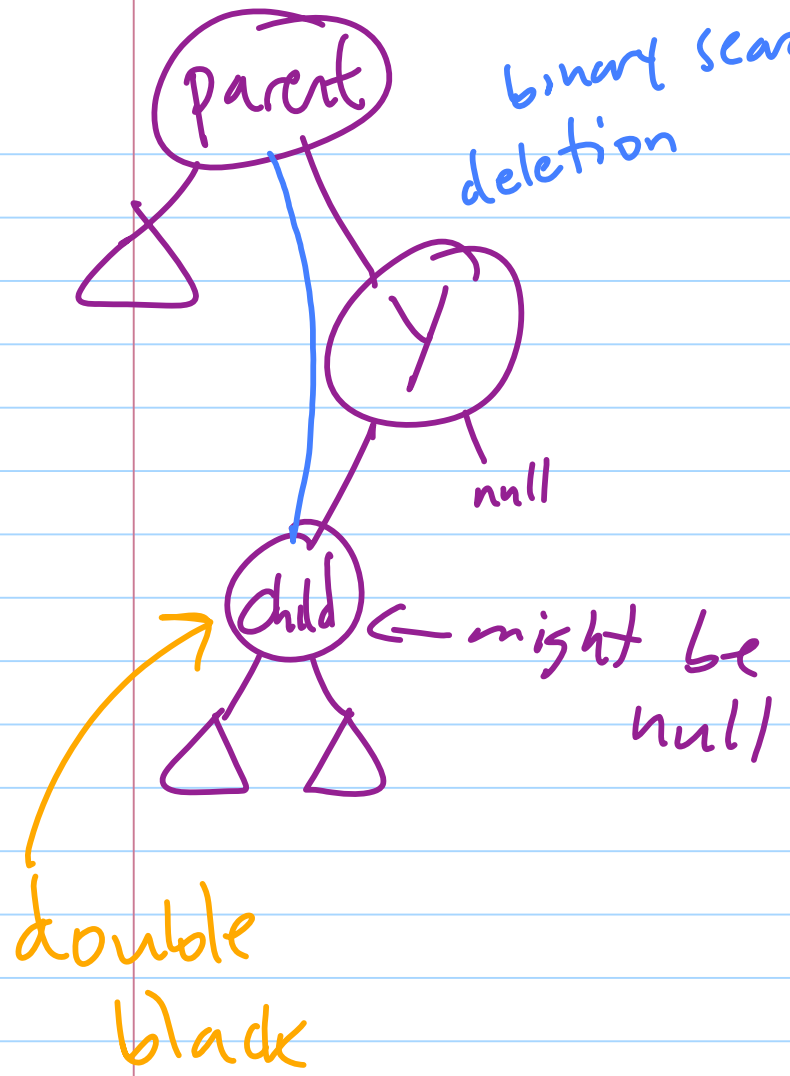
Deletion

(details on pages 523-528 GBG
288-293 CLRS)

If node x to delete has 2 children,
replace x by its successor y
(could use predecessor) where y is
given color of x

Then remove y.) How to do this!

In other cases (x has 0 or 1 child) let
y be the node to delete



binary search tree deletion

if y was red we're done.

If y is black we make its child "double black"

treat it as 2 in black height

B-Tree Data Structure

Note Title 10/24/2007

First finish overview of deletion in red-black tree

Recall that y is node being deleted (successor if original node to delete had 2 children).

doubleBlack is initialized to reference child of y (or frontier/null if y is leaf)

```
void deleteFixUp(RBNode doubleBlack){        //called on a node that has an "double" black
  while (doubleBlack != root && doubleBlack.isBlack()) {      //stop if at root or red node
    Use rotations/recoloring to move doubleBlack up towards root
  }
  doubleBlack.setBlack();          //used when loop terminates with a red node as doubleBlack
}
```

red (or root) when loop exits