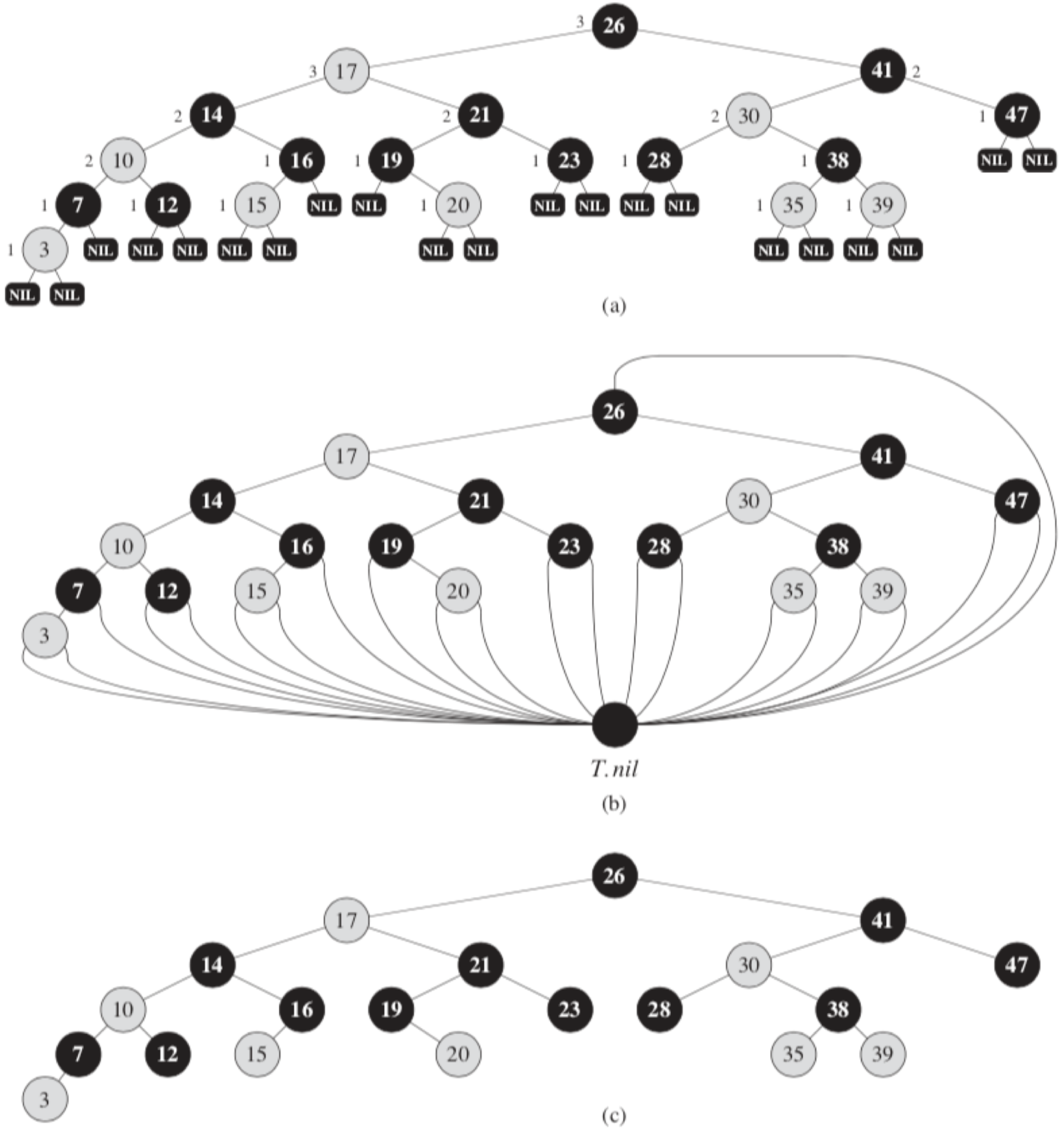
CIS 313:
Intermediate Data Structure

seventh slide

red-black trees

1. every node is either red or black
2. the root is black
3. every leaf (null) is black
4. if a node is red, both of its children are black
5. for each node, all simple paths from the node to descendant leaves contain the same number of black nodes

red-black trees



(a)
(b)
(c)

red-black tree height

- (too) simple analysis:
- the black-height is at most $log_2 n$
- the actual height is at most twice the black height
- so total at most $2log_2 n$
- OK, text says at most $2log_2(n+1)$

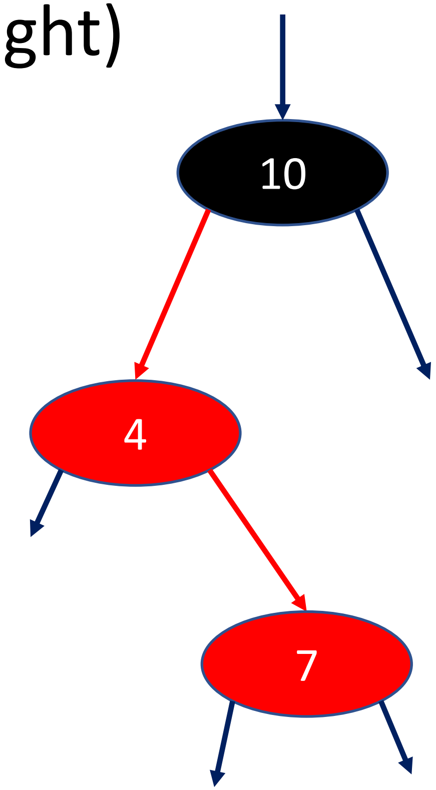
turn a binary search tree into a red-black tree

```
if n is root,
    n.color = black
    n.black-quota = height n / 2, rounded up.

else if n.parent is red,
    n.color = black
    n.black-quota = n.parent.black-quota.

else (n.parent is black)
    if n.min-height < n.parent.black-quota, then
        error "shortest path was too short"
    else if n.min-height = n.parent.black-quota then
        n.color = black
    else (n.min-height > n.parent.black-quota)
        n.color = red
    either way,
        n.black-quota = n.parent.black-quota - 1
```

code taken online

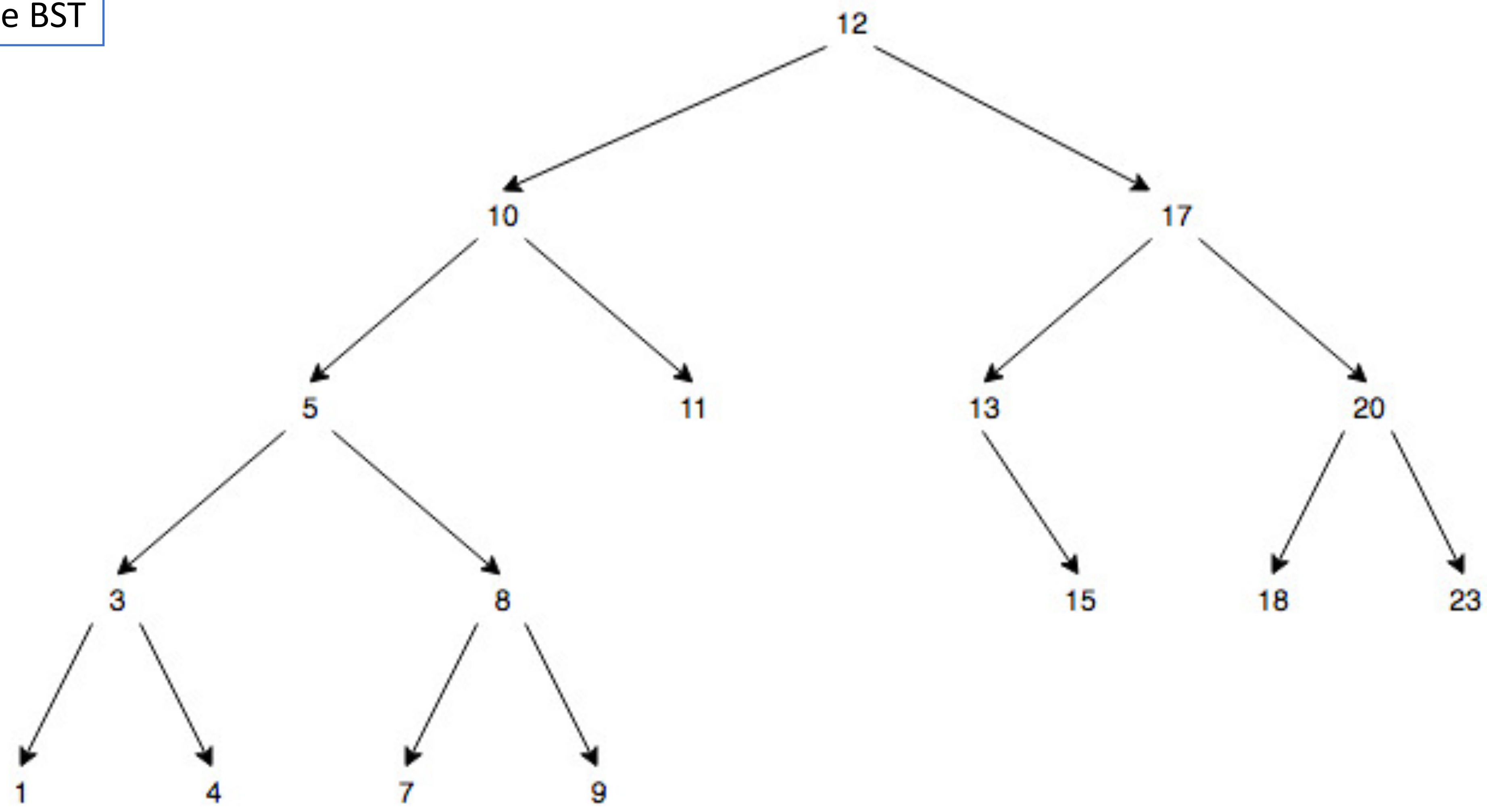
red-black tree insertion

- to insert new key x
- as always, search to the bottom of the tree for where x would go
- put x there and color it red (to maintain black-height)
- this might cause a problem: two reds in a row
- if no such problem, then done
- if double-red problem, then fix using
  - color shifts or
  - rotation



example: insert 7

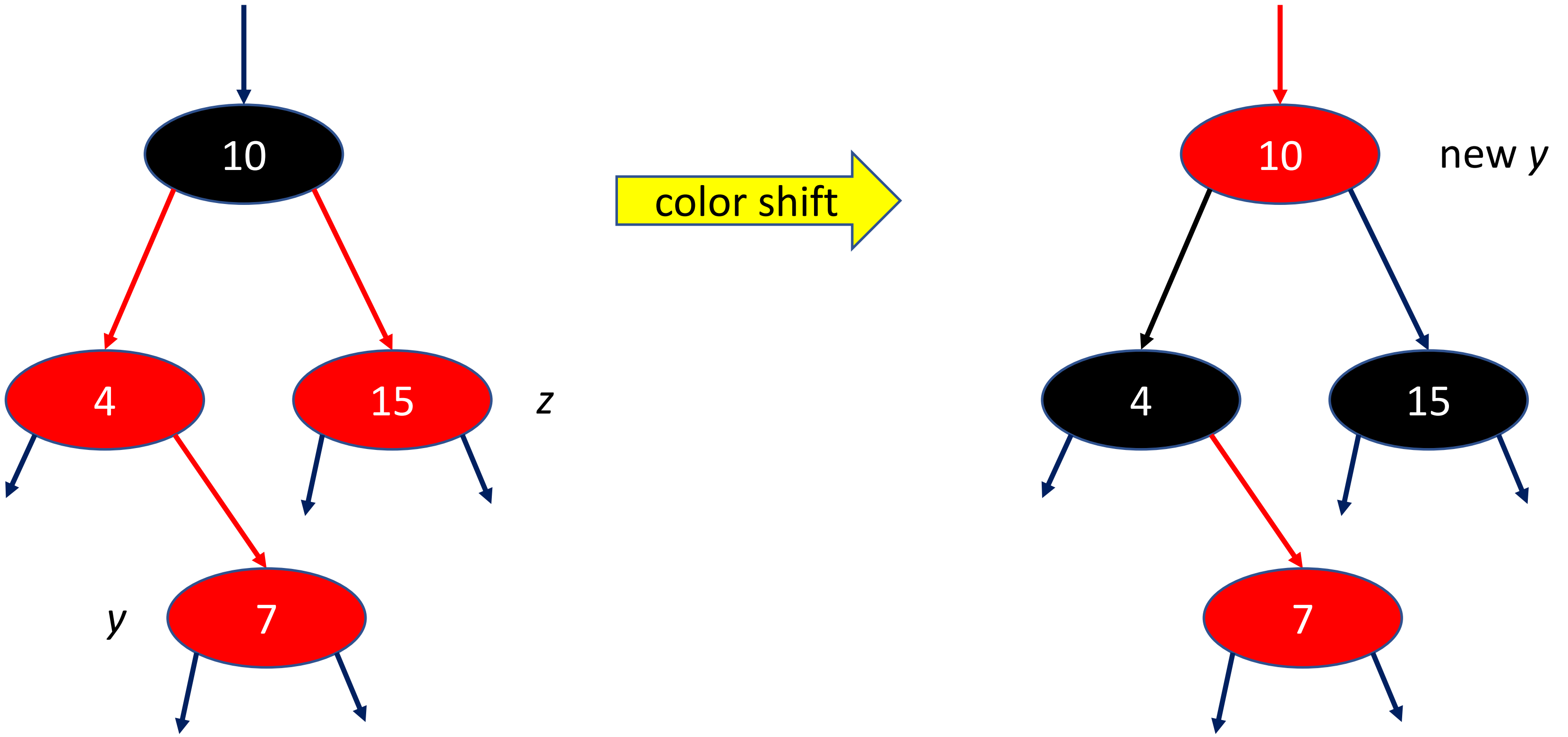
sample BST



RB-INSERT-FIXUP

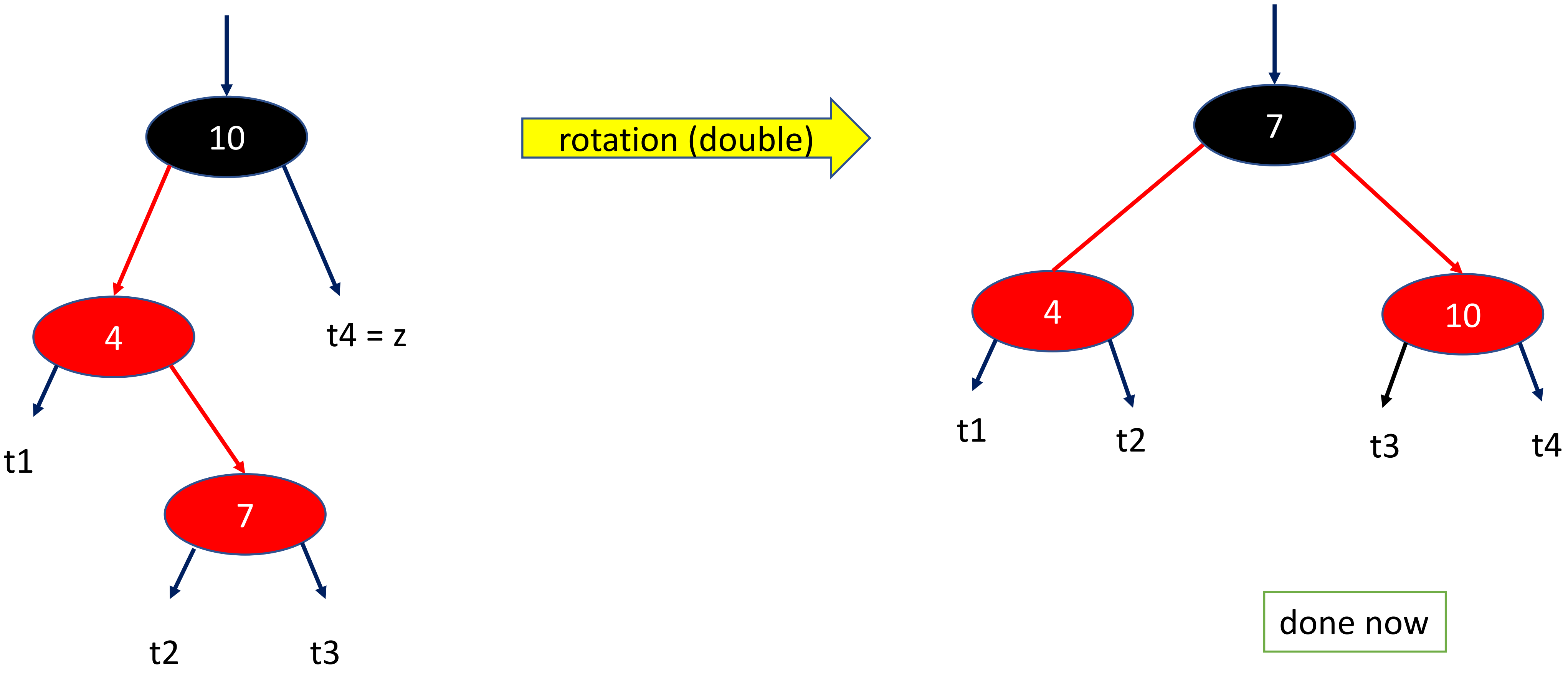
- section 13.3 of text
- this deals with the double red case after an insertion
- let y be the current node, both it and its parent are red
- let z be the "uncle" of y: the sibling of y's parent's parent
- two cases:
- z is red
  - color shift
  - then check again for double red, possibly continue
- z is black
  - rotate
  - done

z is red



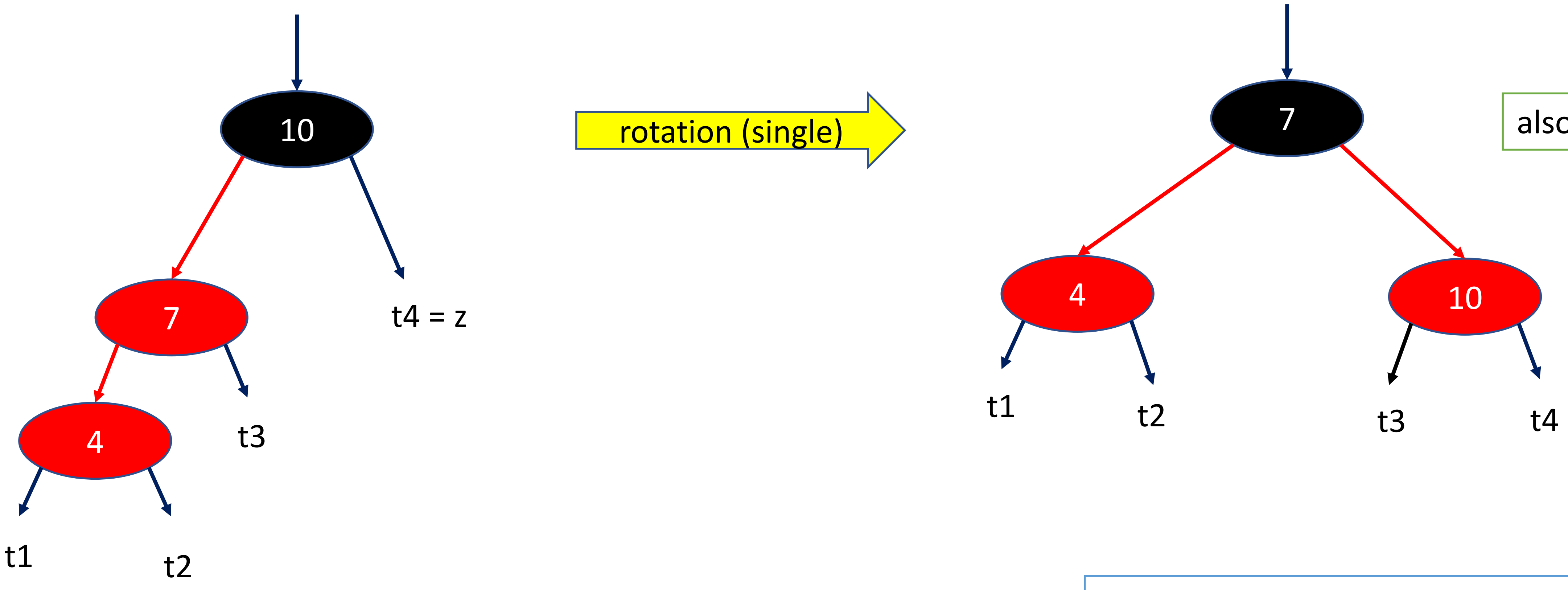
this swaps a black and red level, preserving black height along these paths, but may create another double-red at the new y

z is black



done now

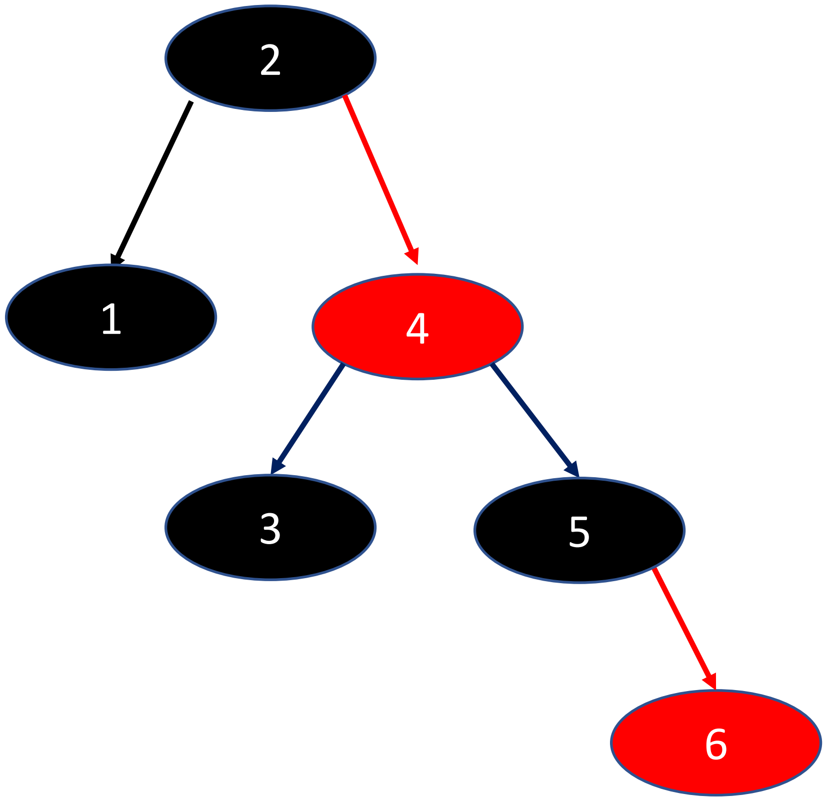
z is black (again, different case)



also done

there are two other cases similar to these needing single and double left rotations

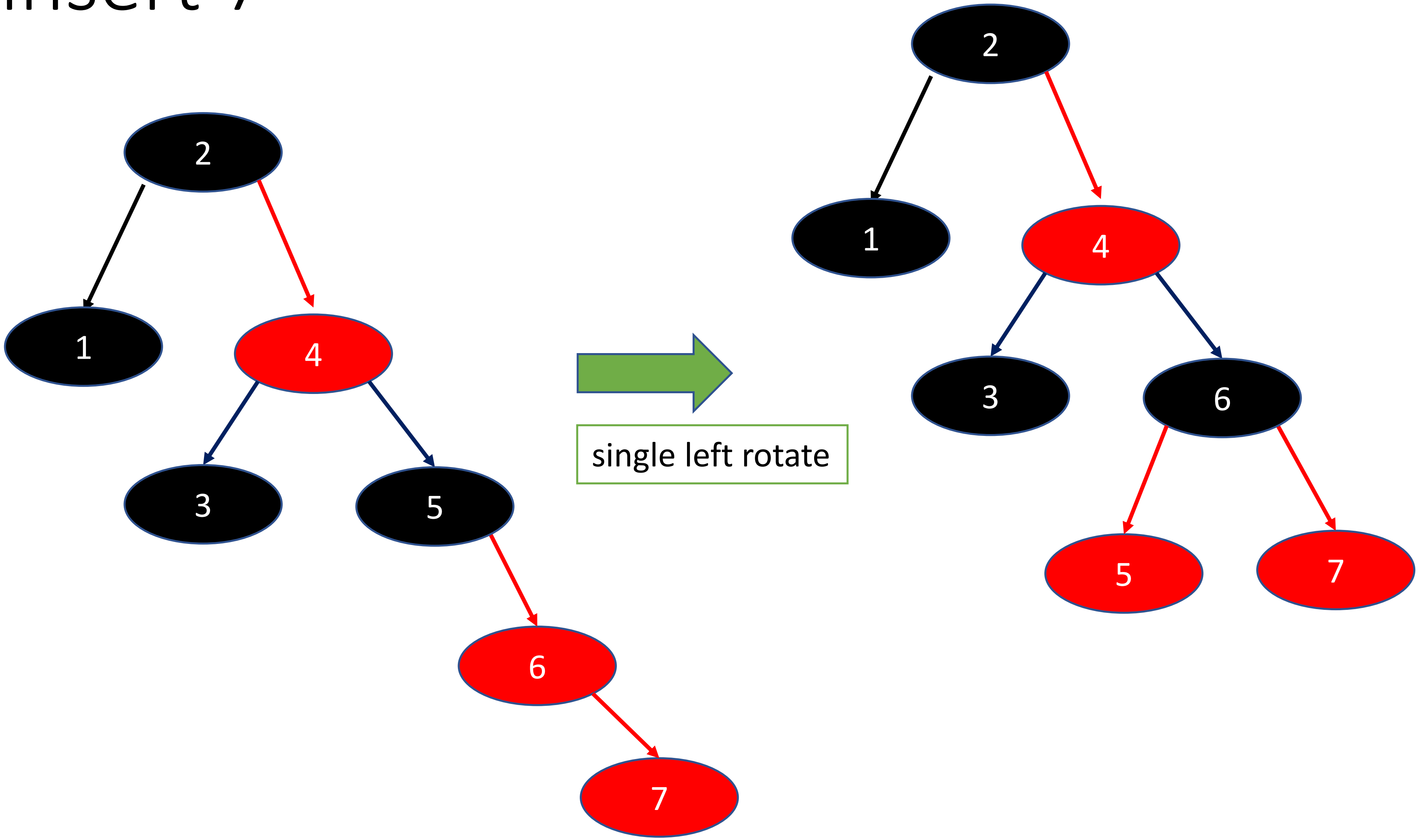
example insertions



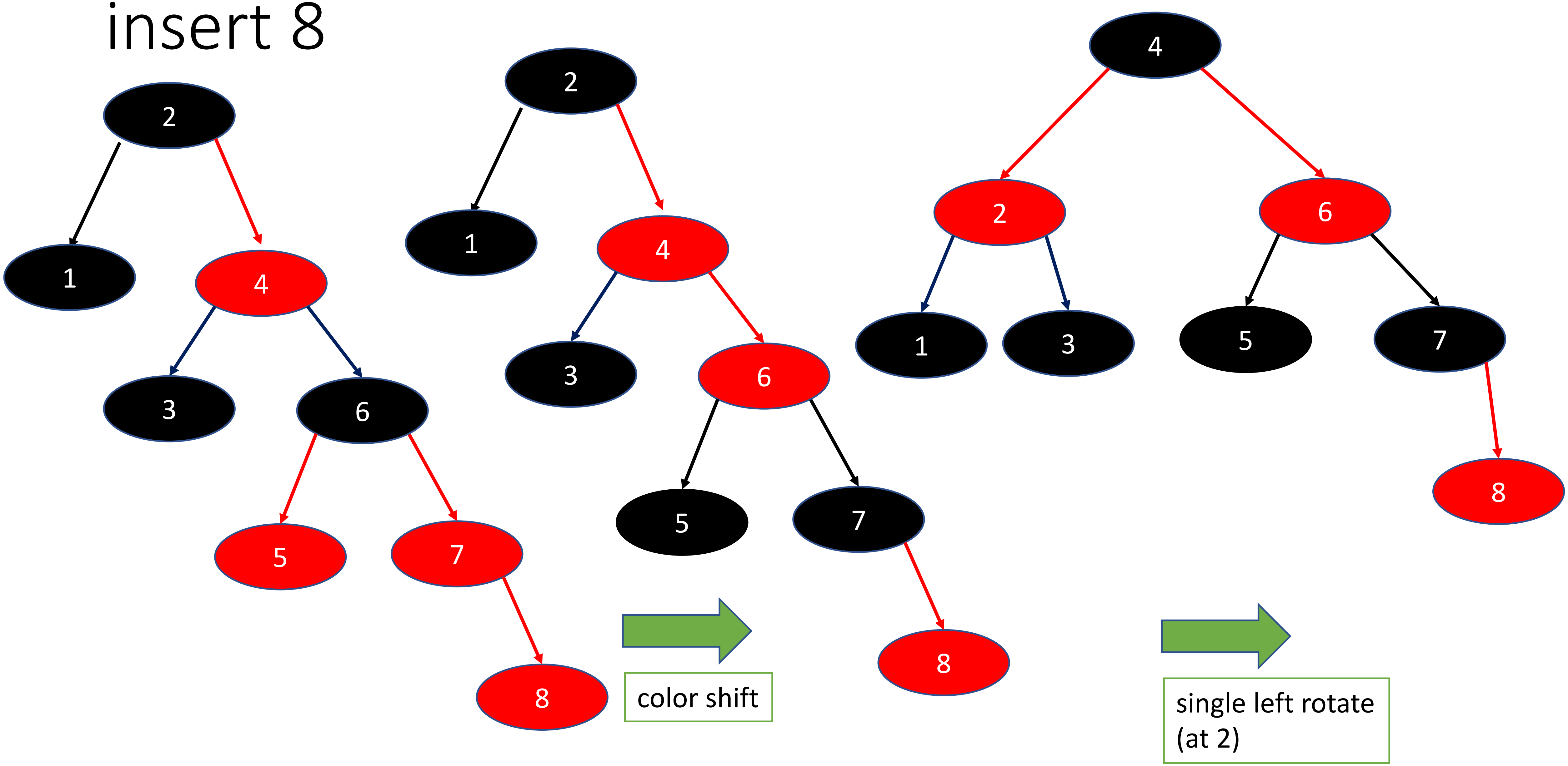
after insertion of 1,2,3,4,5,6 into empty RB tree

let's continue with 7, 8, …

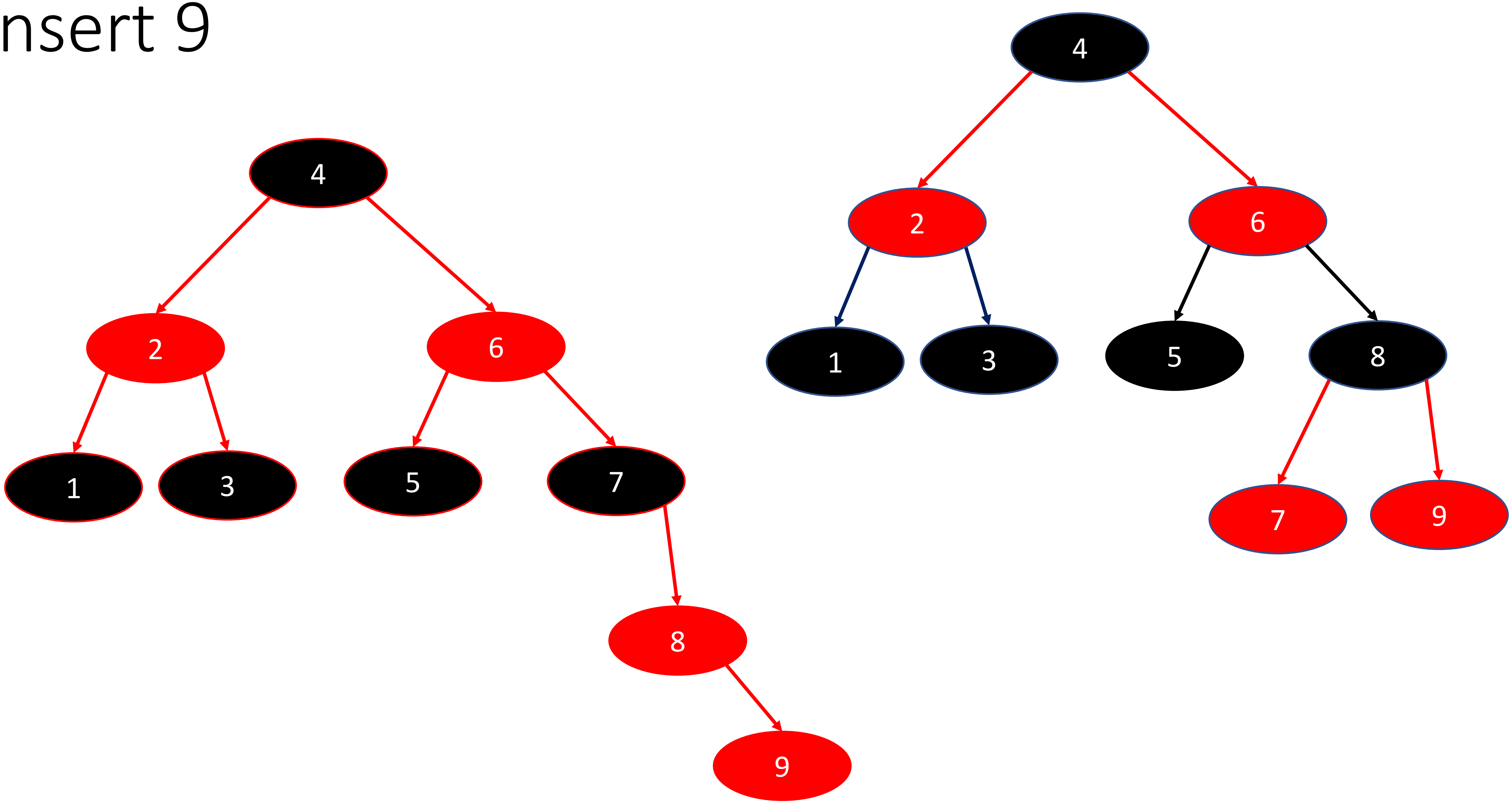
insert 7



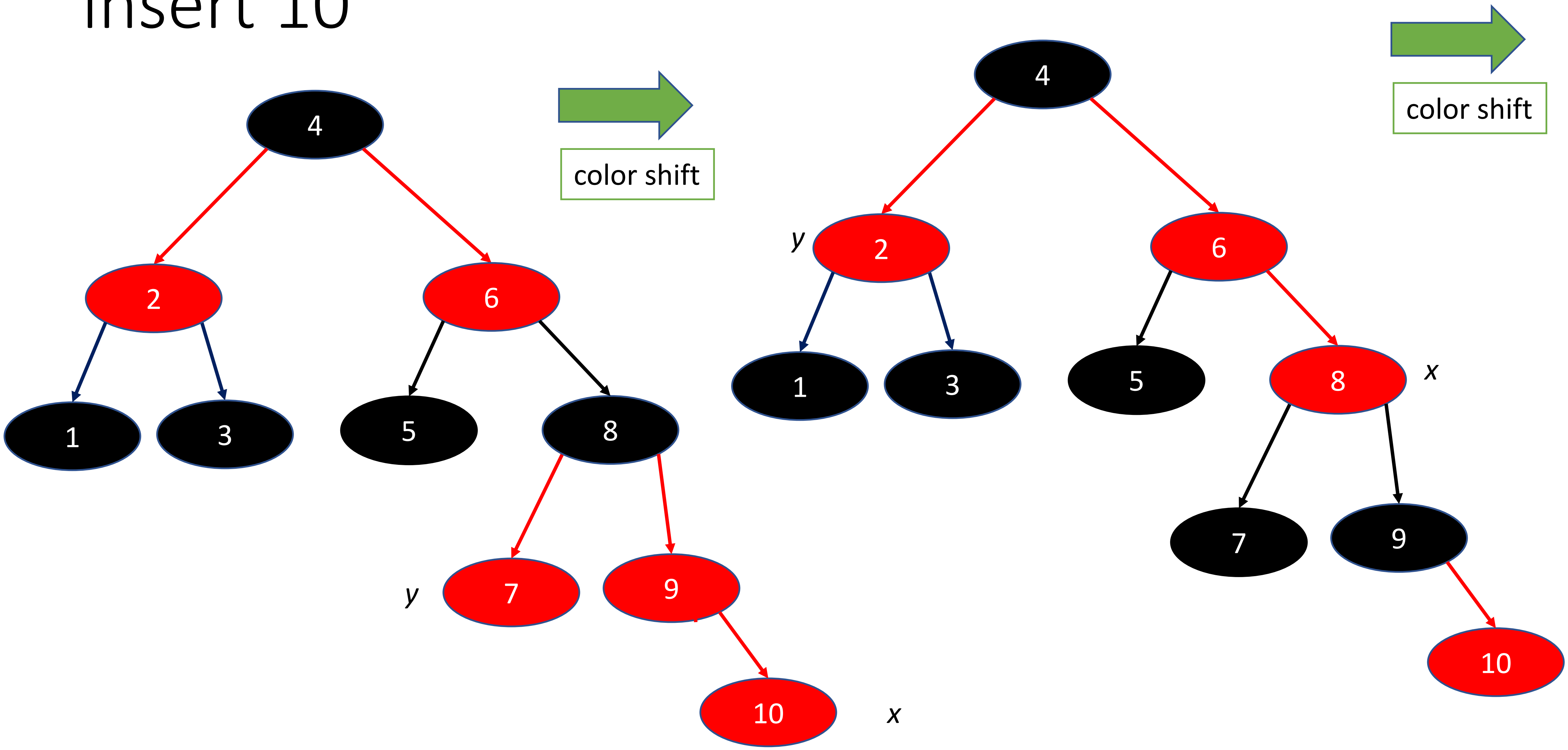
insert 8



insert 9



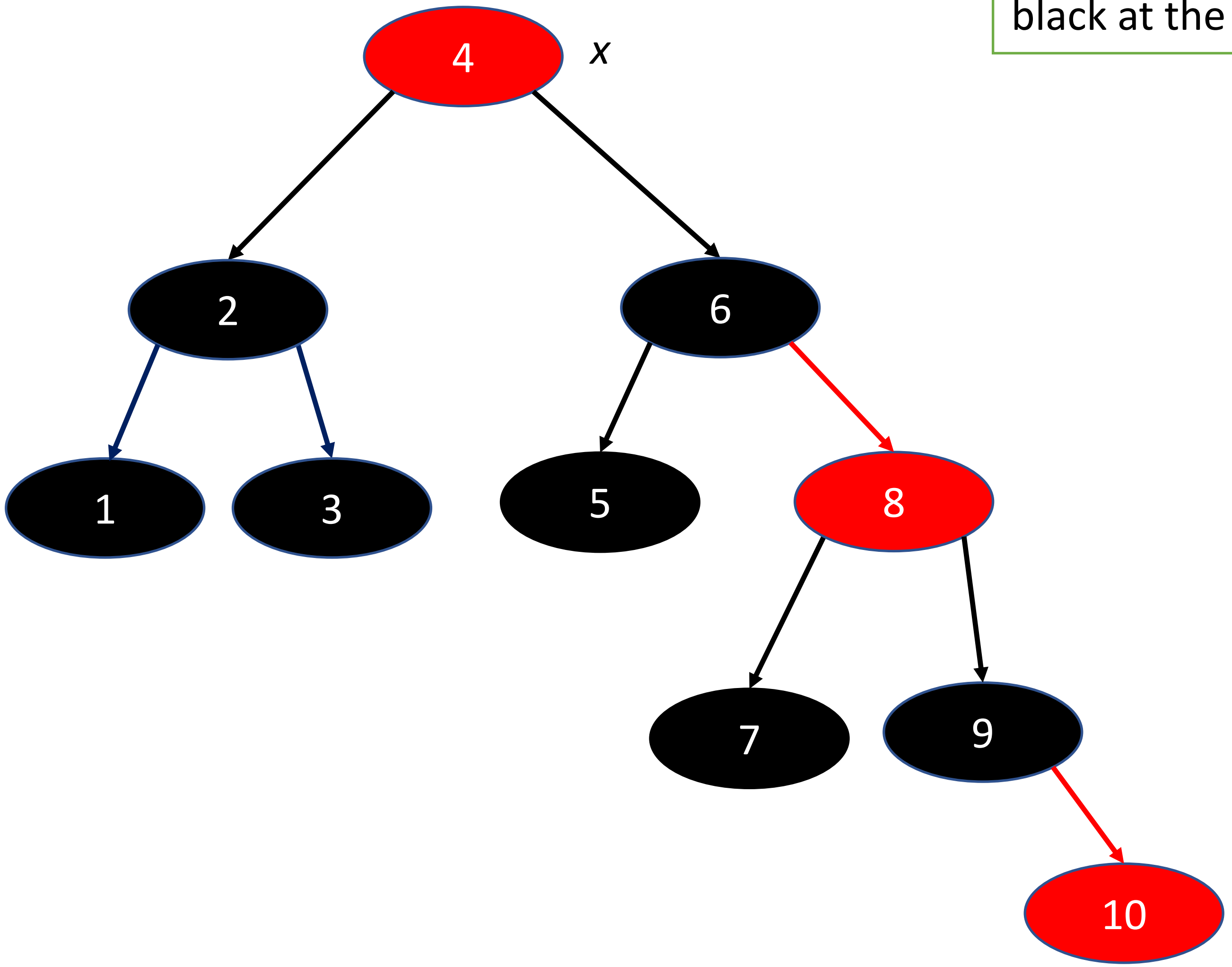
insert 10



insert 10 (cont’d)

color shift

note: 4 as root gets colored black at the end



RB Deletion

- BST Deletion Revisited: delete z
  - If z has no children, then just remove it
  - If z has only one child, then splice out z
  - If z has two children, then:
    - Find its successor y
    - Splice out y
    - Replace z's value with y's value

-> so the physical node deleted is z in the first two cases and y in the third case

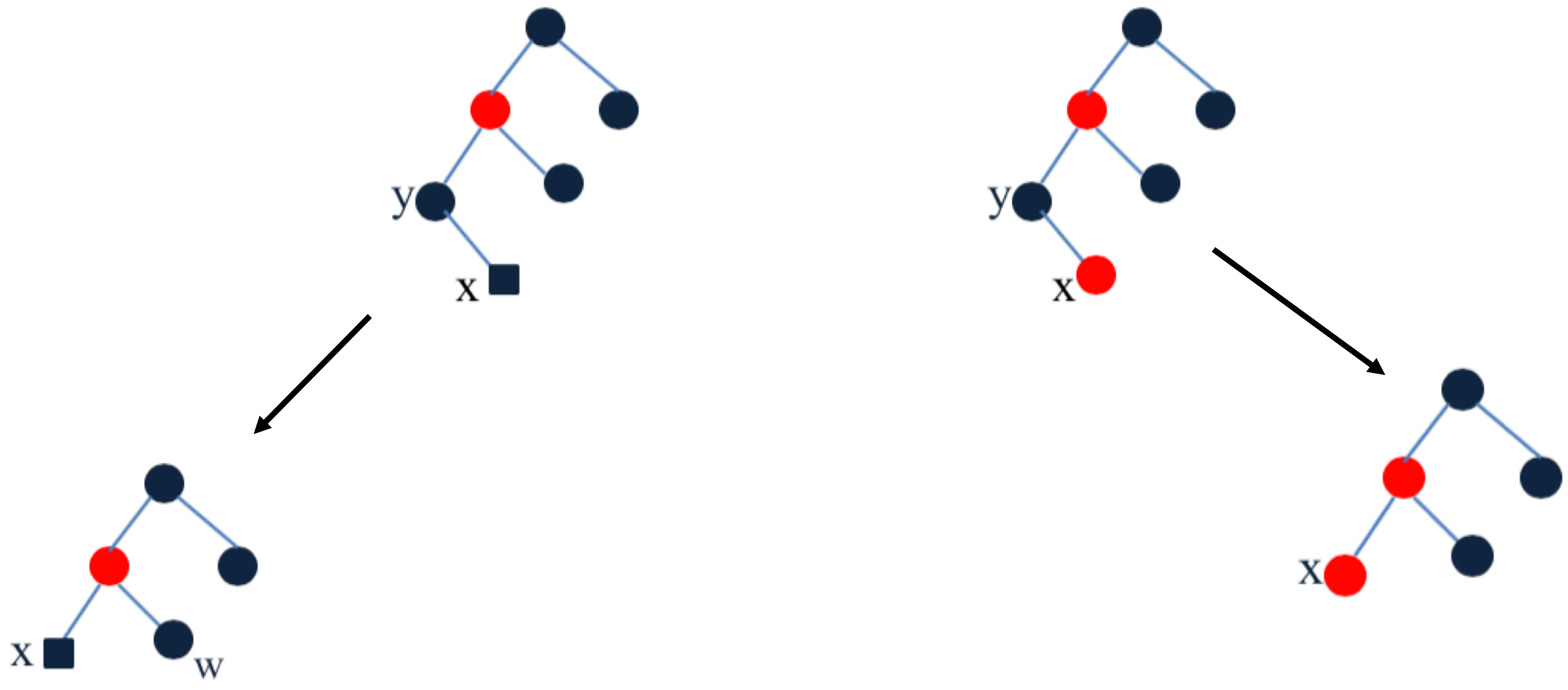
RB Deletion

- Delete z as in BST
- If z has two children, when replace z's value with the successor's value, keep z's color (don't change z's color)
- Let y be the node being removed or spliced out in this procedure

  (y would be either z or successor of z, thus y has at most one child)
- If y is red, no violation of the red-black properties, done
- If y is black, some violations might arise and we need to restore the red-black properties

RB Deletion: y is black

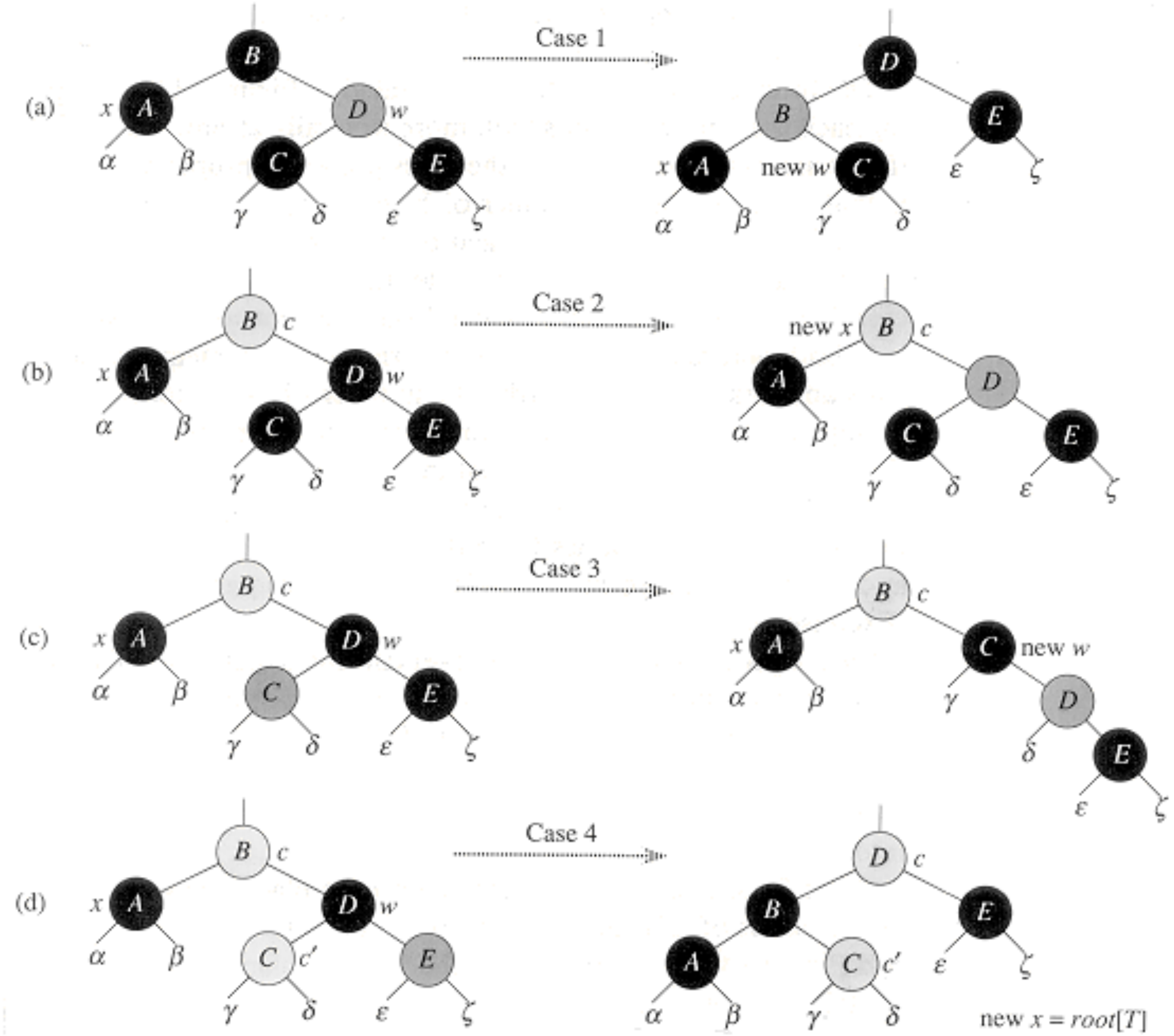
- Let x be the child of y before it was spliced out
  - So x is either nil (a leaf) or the only non-nil child of y



Restoring RB Properties

- The RB-DELETE-FIXUP routine in the text, applied to x
- If x is red, so easy, just change its color to black and done
- If x is black:
  - Transform the tree and move x up, until:
    - x points to a red node, or
    - x is the root
  - At each step:
    - need to consider 8 cases; four when x is a left child and four when x is a right child.
    - due to the symmetry, just consider the 4 cases when x is a left child here
  - REMEMBER: set the color of x to black in the end

Restoring RB Properties: x is black and is a left child



x’s sibling w is red:
Left rotate D, switch colors of B and D

x’s sibling w is black; both w’s children are black:
Move x up, change w’s color to red

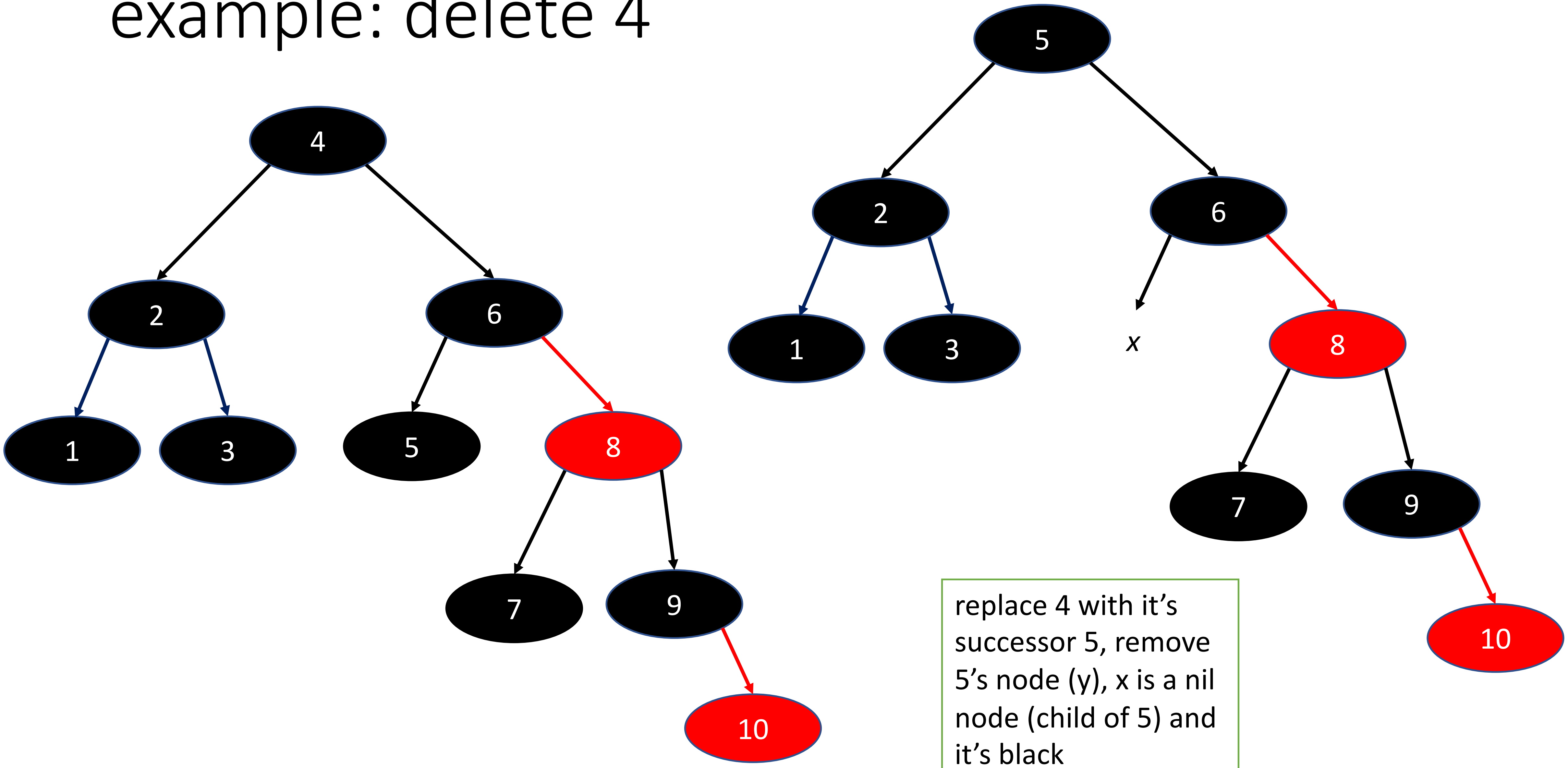
x’s sibling w is black; w’s right child is black, left child is red:
Right rotate on C, switch colors of C and D

x’s sibling w is black; w’s right child is red:
Left rotate on D, switch colors of B and D, change E’s color to black

The nodes with c or c’
can be either red or black

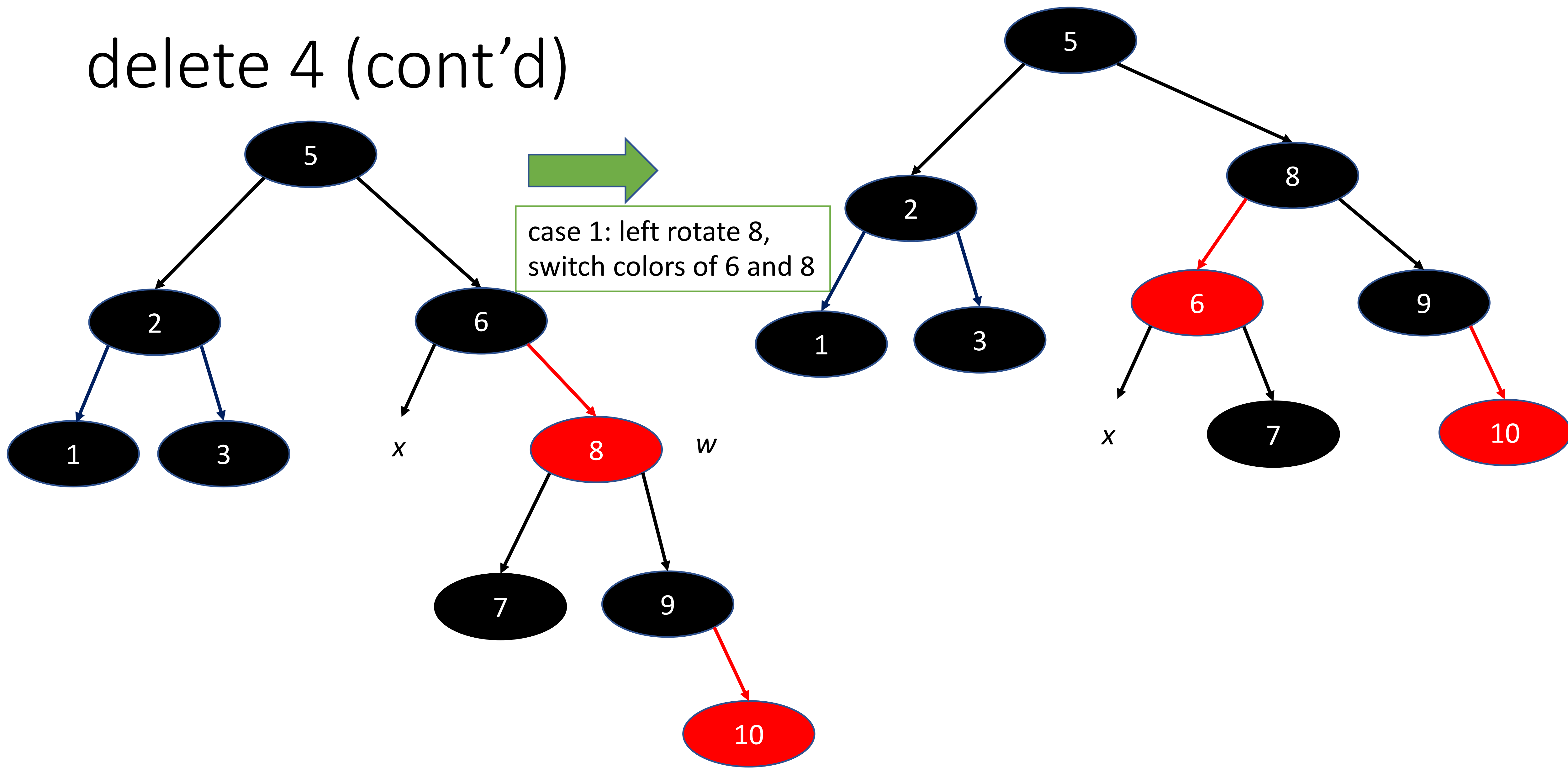
figure 13.7 from text

example: delete 4

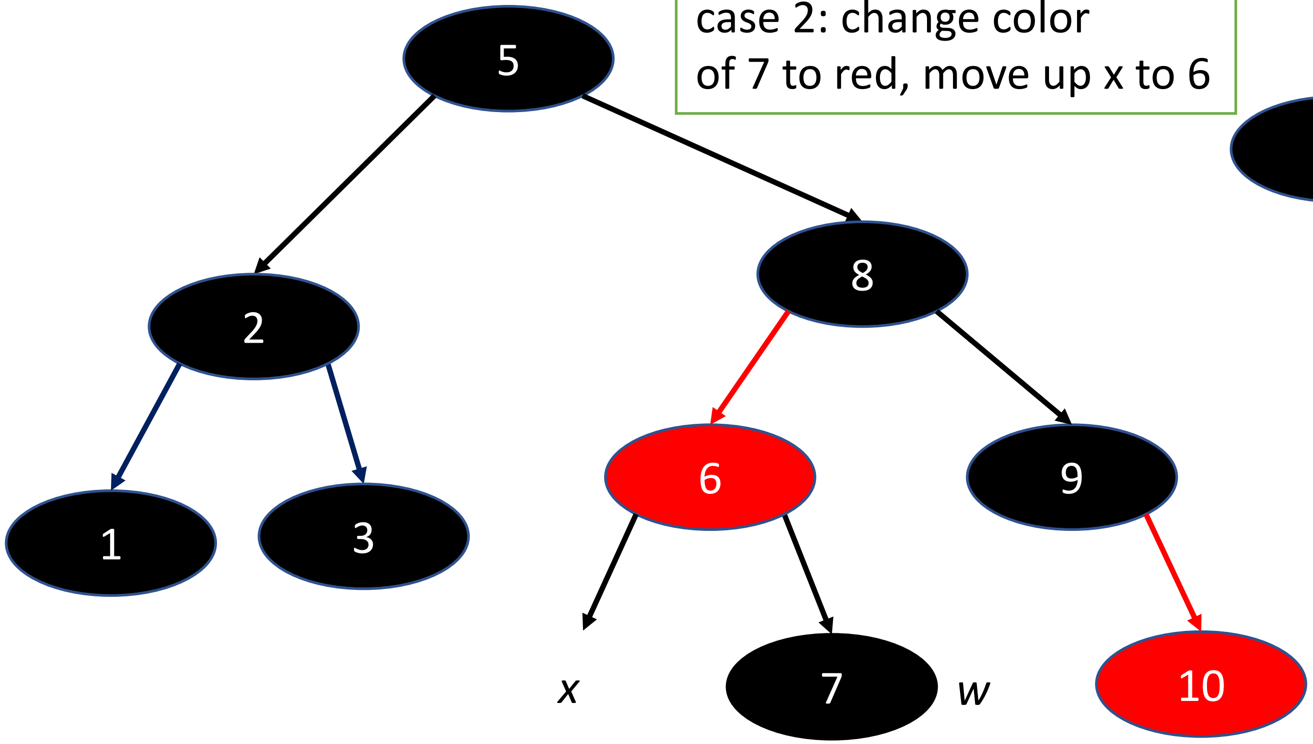


replace 4 with it's successor 5, remove 5's node (y), x is a nil node (child of 5) and it's black

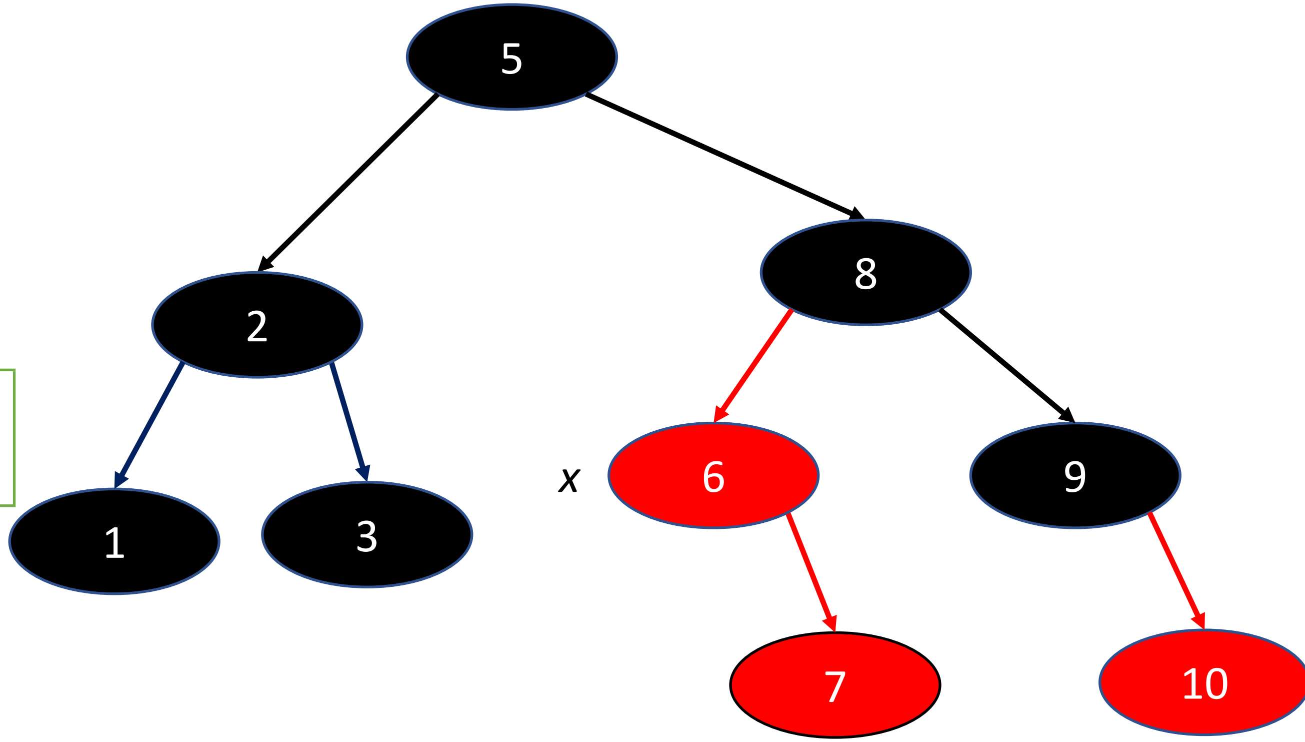
delete 4 (cont'd)



delete 4 (cont'd)

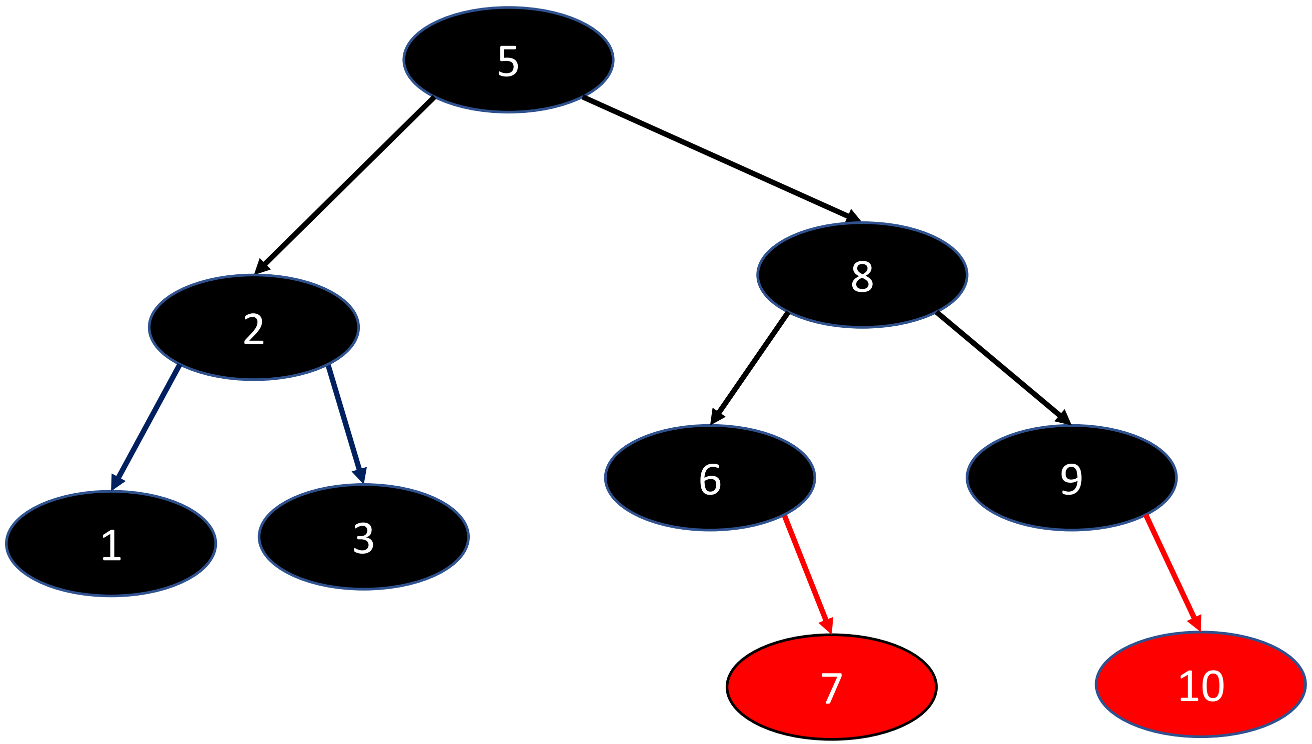


case 2: change color of 7 to red, move up x to 6



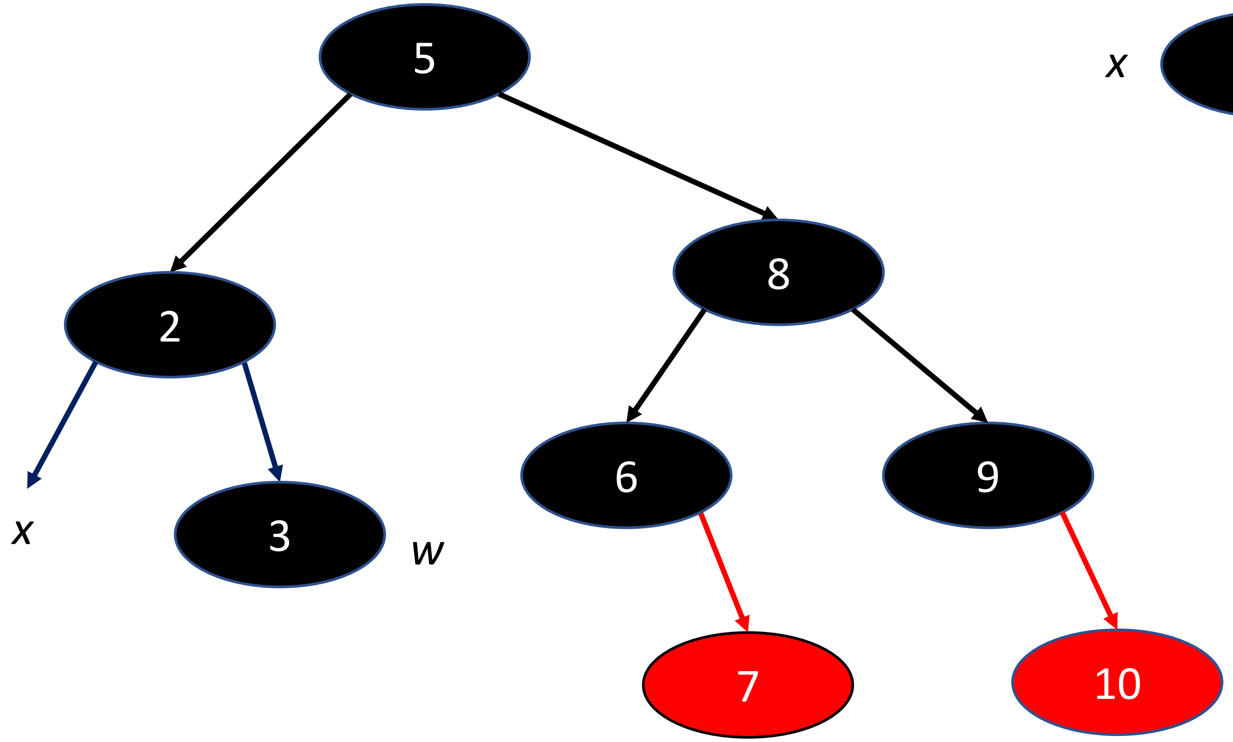
x is now red and we are done with the moving up of x
FINAL STEP: change color of x to black

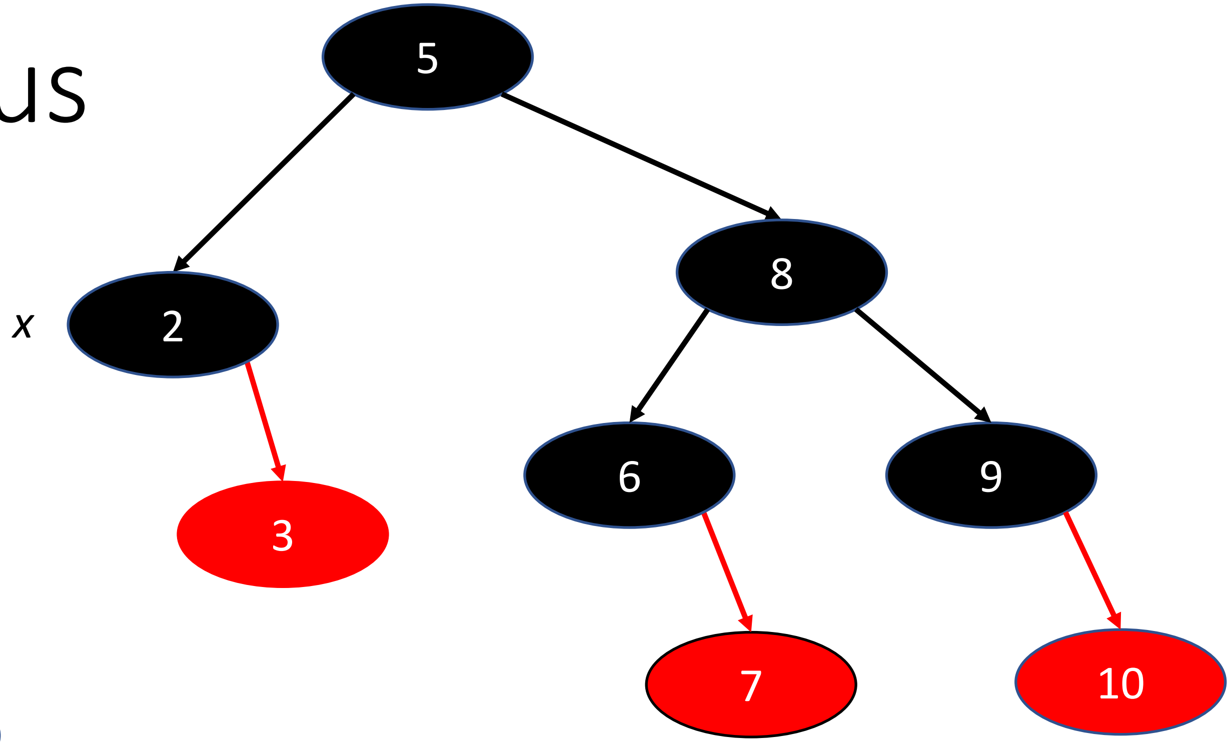
delete 4 (done)



NEXT: delete 1 from this
y is 1 and x is the nil child of 1 (black)

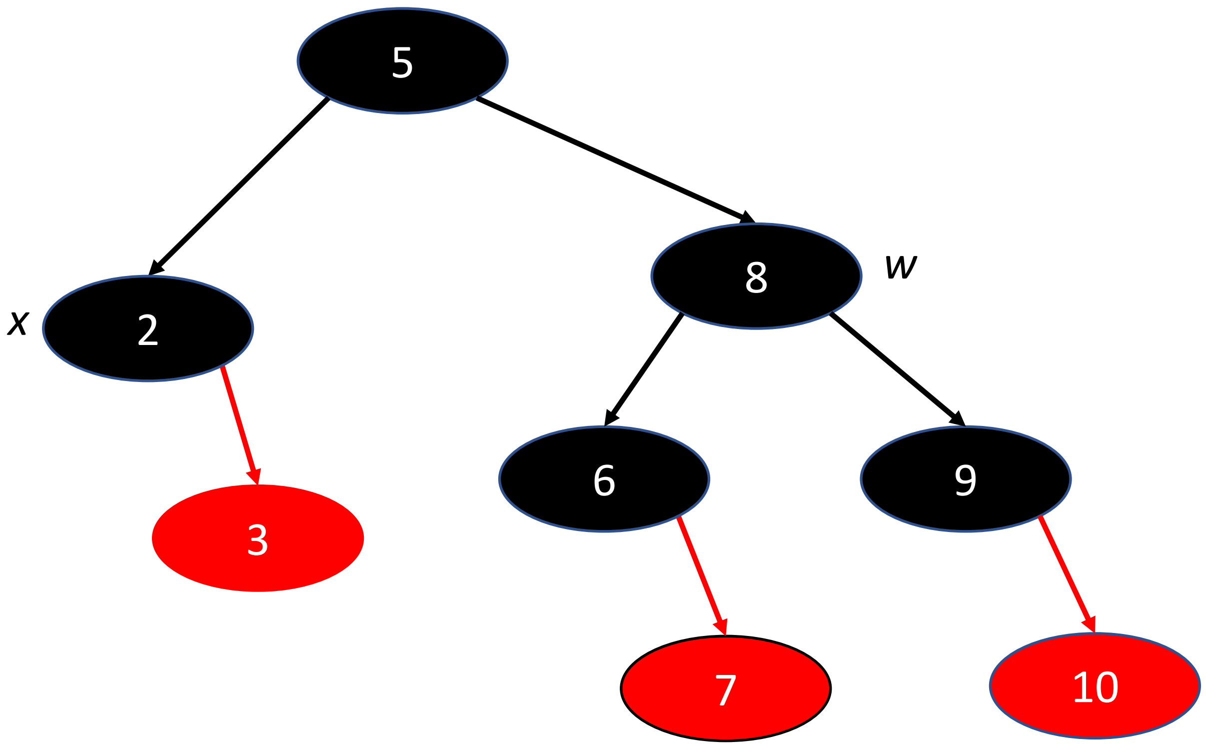
deleting 1 from previous

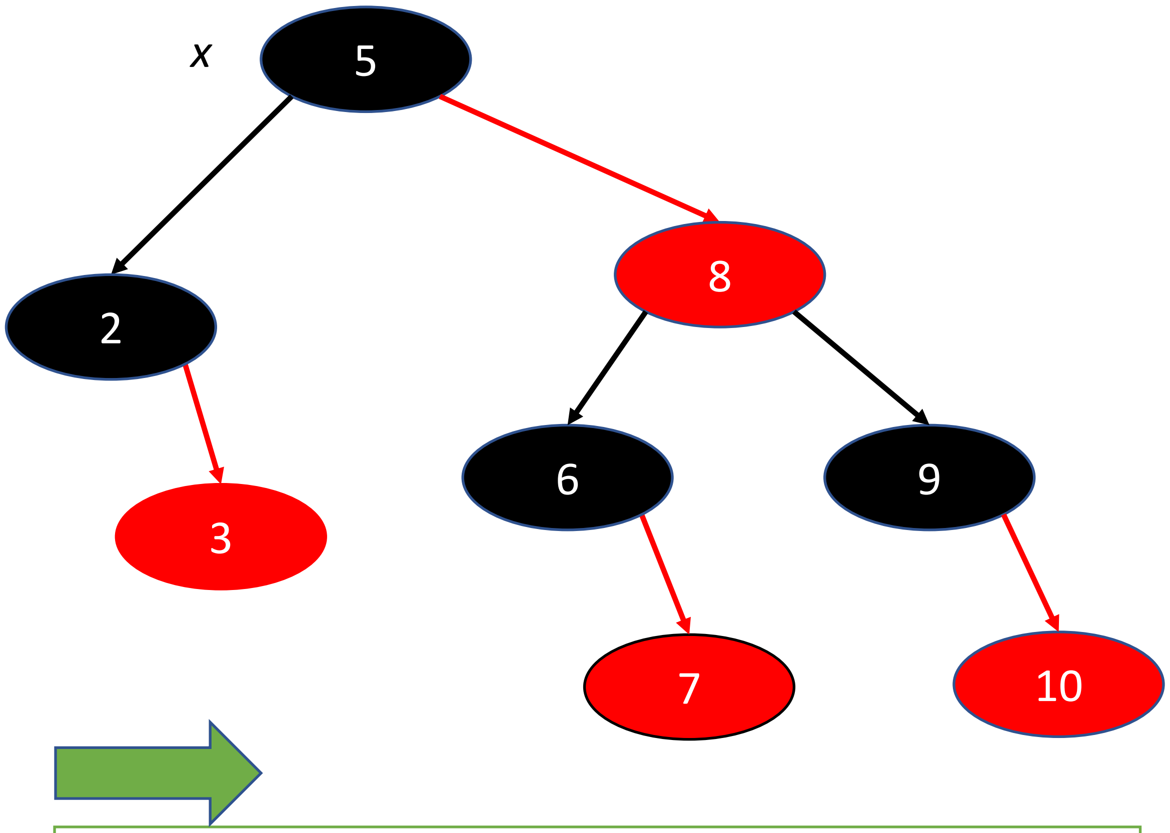




case 2: change color of 3 to red and move up x to 2

delete 1 (cont'd)

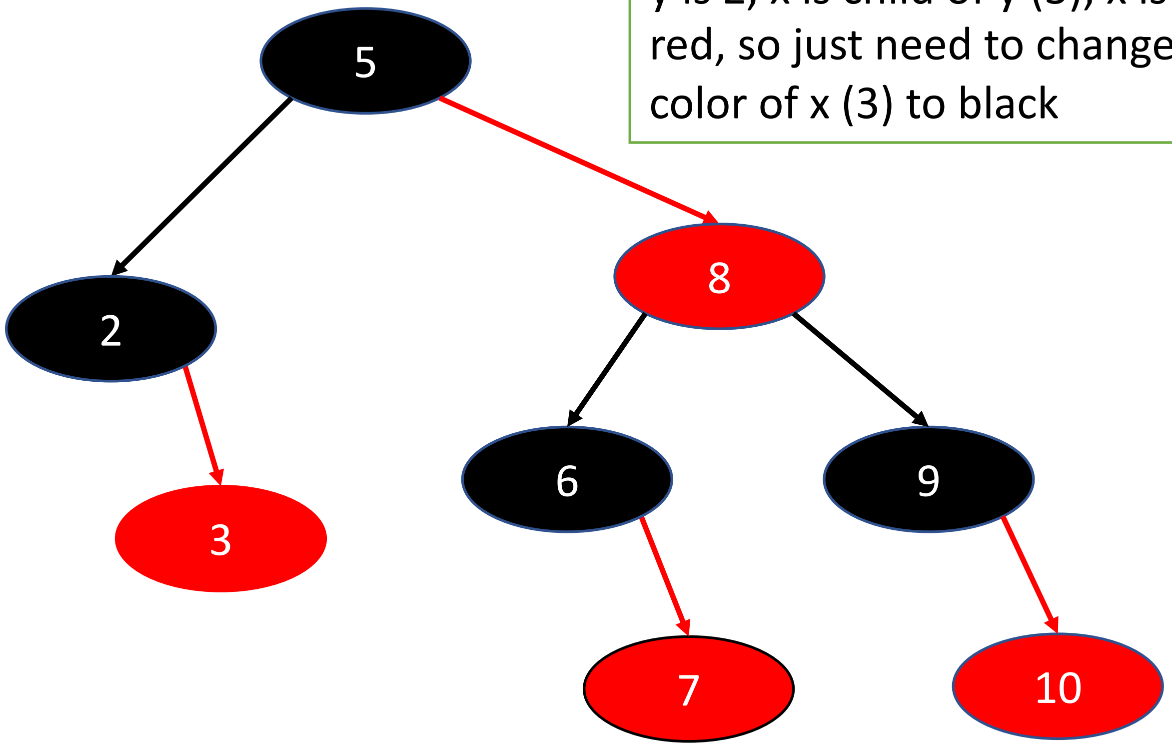




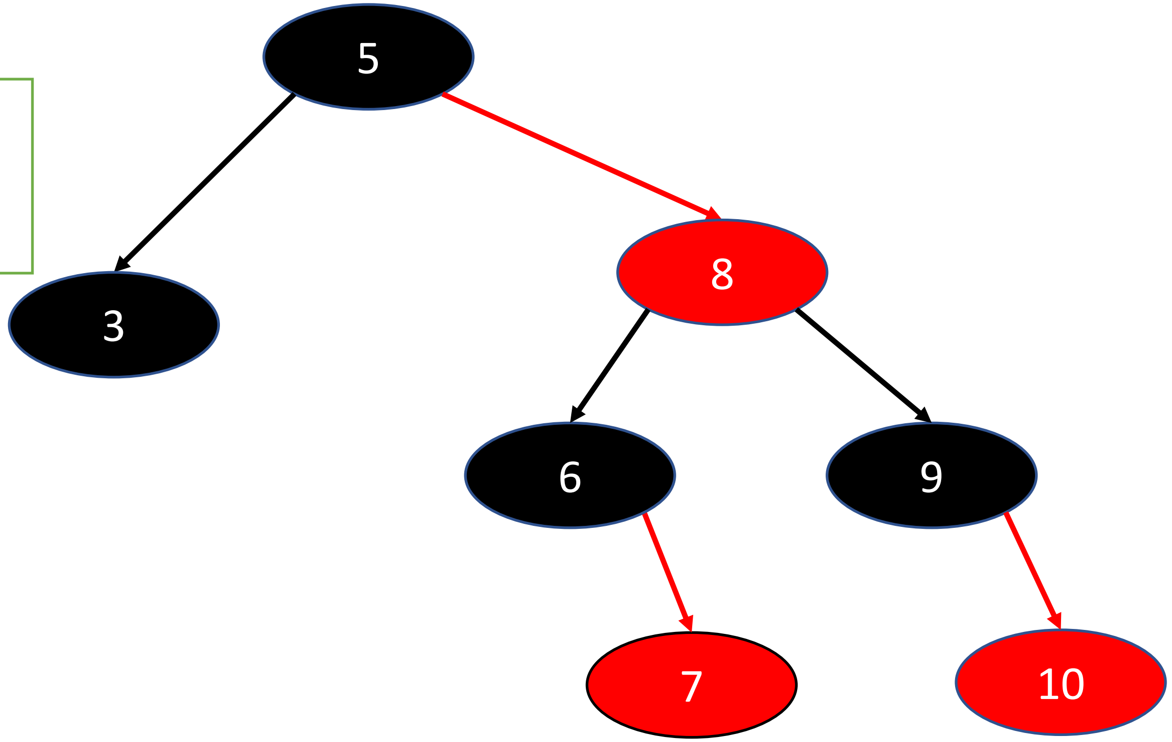
case 2 again: change color of 8 to red, move x up to 5

done since x is at root (we've reduced the black height of the tree)

remove 2

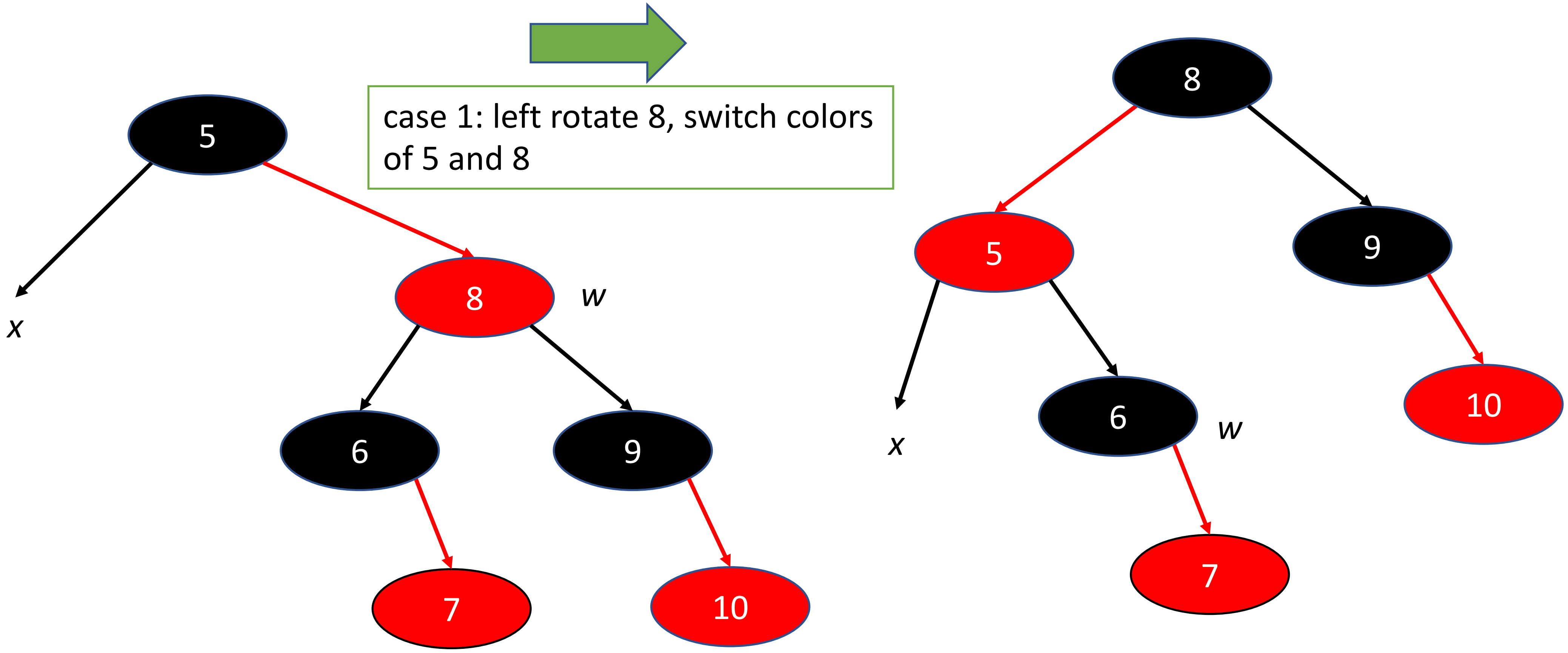


y is 2, x is child of y (3), x is red, so just need to change color of x (3) to black

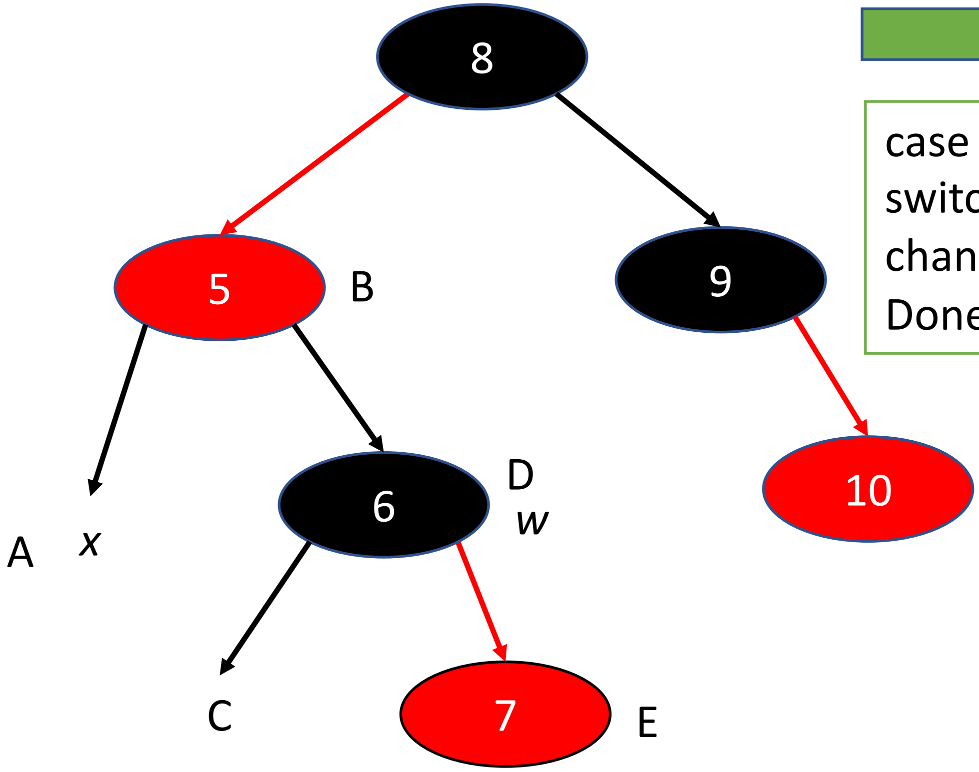


NEXT: delete 3 from this
y is 3, x is the nil node (child of y)
x is black

removing 3 from previous

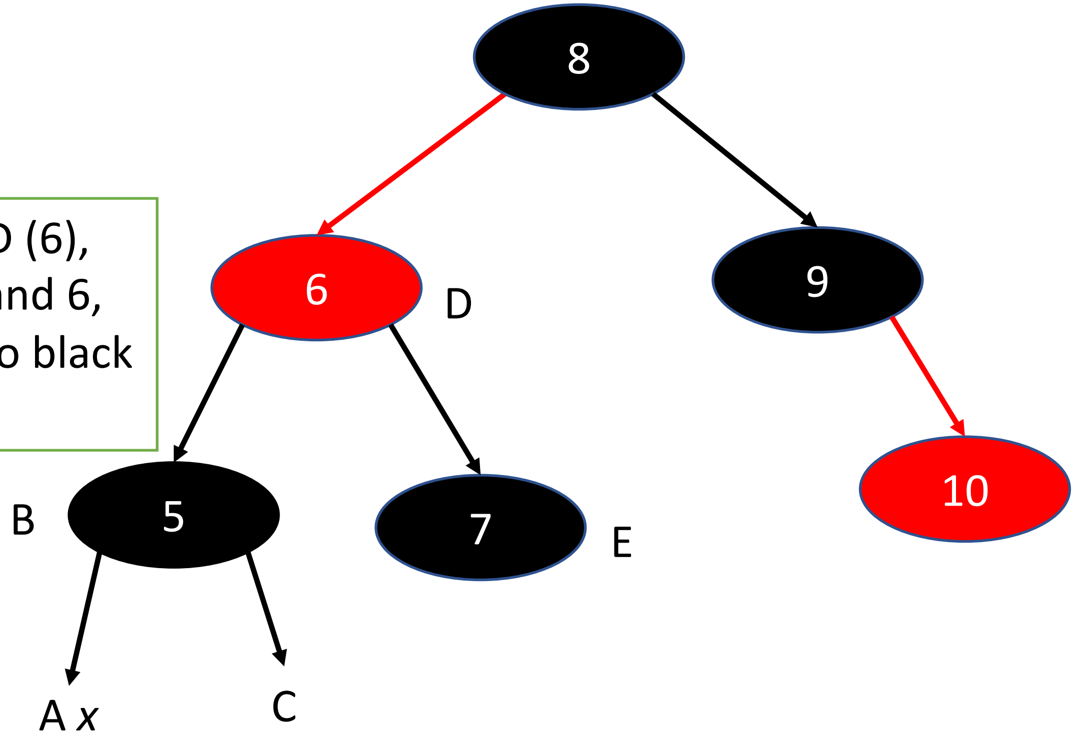


remove 3 (cont'd)



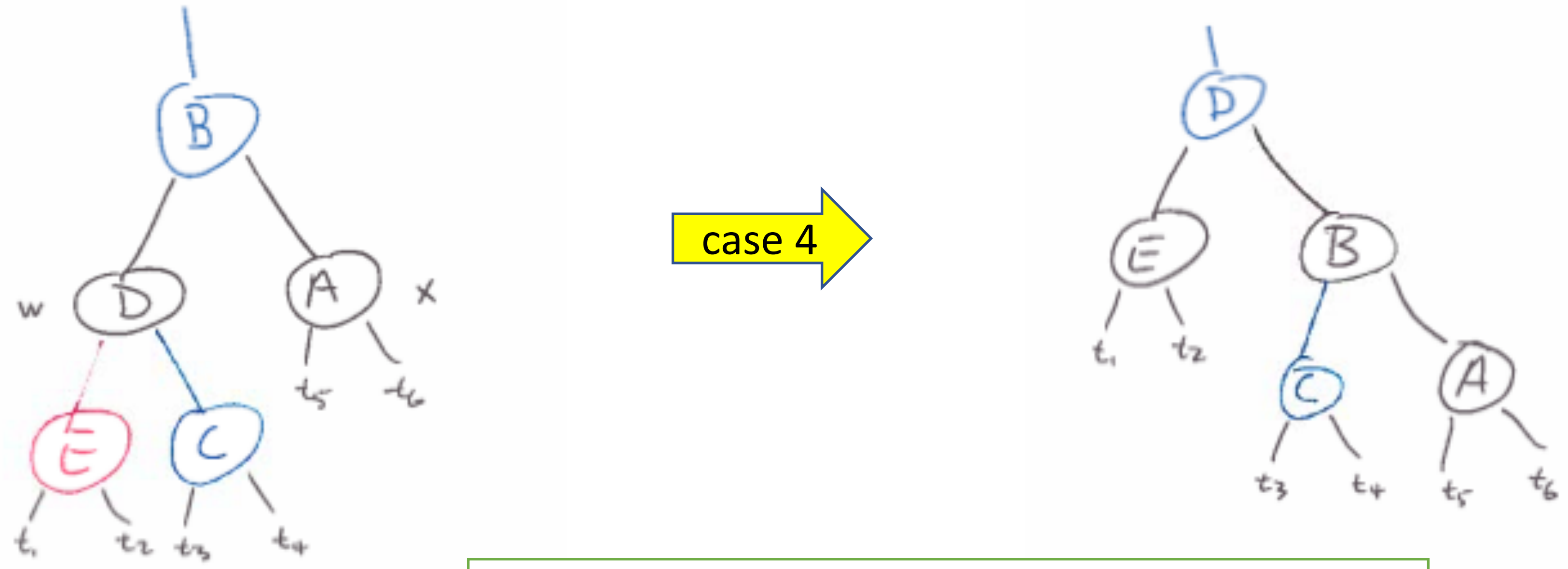
set up as case 4:

case 4: left rotate D (6),
switch colors of 5 and 6,
change color of 7 to black
Done!



motivated by a "transfer" in 2-3-4 tree

remember: all cases come with mirror image



- here x is right child of parent
- the left child of w is red
- fix-up can be completed with a right rotation and color changes
- note that the blue nodes (B and C) can be either red or black

B-trees

- very important data structure in computer science
- database indexing, hard disk referencing, MongoDB, …
- balanced, multi-way search tree
- many slight variations, we will use definition in CLRS text
- idea is that nodes are large and fit into a disk block (minimum amount of data that's pulled off a hard drive)
- node size parameters (here called t) depend on disk speeds, block sizes, etc.

B-tree specifications

- fixed parameter t, called *minimum degree*
- nodes have between t-1 and 2t-1 keys
- so therefore they have between t and 2t children
- root is exception: it may have as few as 1 key (2 children)
- all null pointers have the same depth (distance from root)
- a 2-3-4 tree is a B-tree with minimum degree t=2

different texts: things to look for

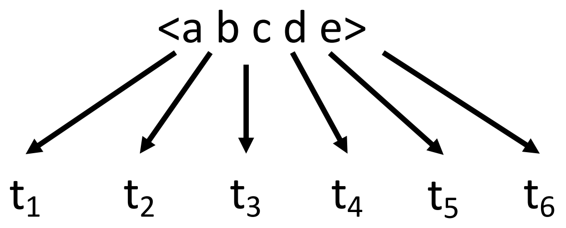
- top-down versus bottom-up insertion
  - CLRS does top-down, split full nodes during search
  - unlike how it does RB trees
  - bottom-up more common in practice, less wasted space
- ties to left or right
  - no duplicates here
  - need to know for B+ trees, which have all keys at an additional “leaf level”
- left/right bias: if middle key not well defined (when splitting a node with even number of keys)

B-tree node format

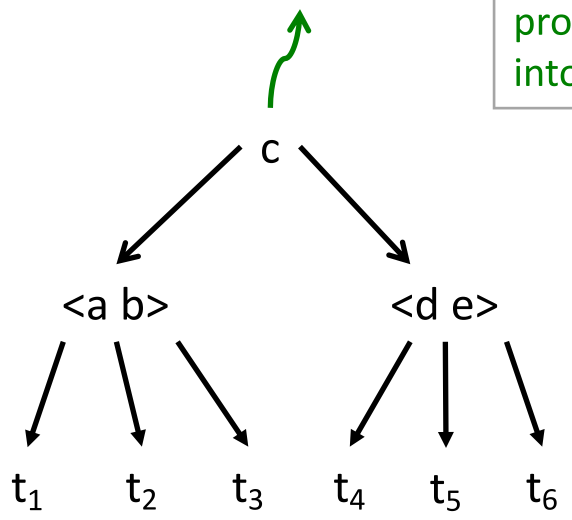
- each node can have between t and 2t children
- a node might look like
  - $<P_0, K_1, P_1, K_2, P_2, \ldots, P_{q-2}, K_{q-1}, P_{q-1}>$
  - for t <= q <= 2t (except for root 2<=q<=2t)
- during a search, we split full nodes
- a node is full when it has 2t-1 keys

node split (shown for t=3)

split a full node



becomes



promote c by inserting into parent

B-tree height

- theorem 18.1: if $n \geq 1$, then for any B-tree containing *n* keys of height *h* and minimum degree $t \geq 2$, $h \leq \log_t \frac{n+1}{2}$.
- example: t=50 and n=100,000,000
  - $h \leq \log_{50} 50{,}000{,}000.5 \cong 4.53 \leq 5$
- suppose 20 records fit on a page
- without the index to find an item we'd need to search about half the 100,000,000/20=5,000,000
- with the index we need at most 5+1=6 disk accesses (5 for the tree nodes and one for the page containing that key's record)

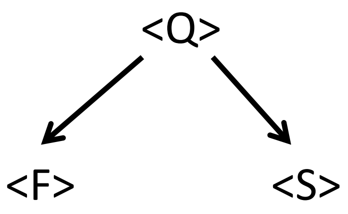
exercise 18.2-1

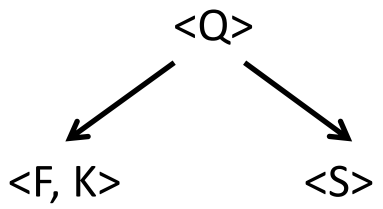
insert into initially empty B-tree of min degree t=2 the key values
F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E

after the first 3 values:

<F, Q, S>

a search to place K causes a split:

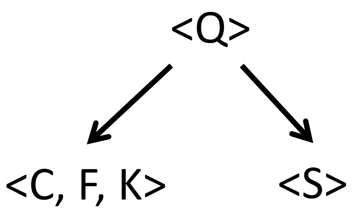




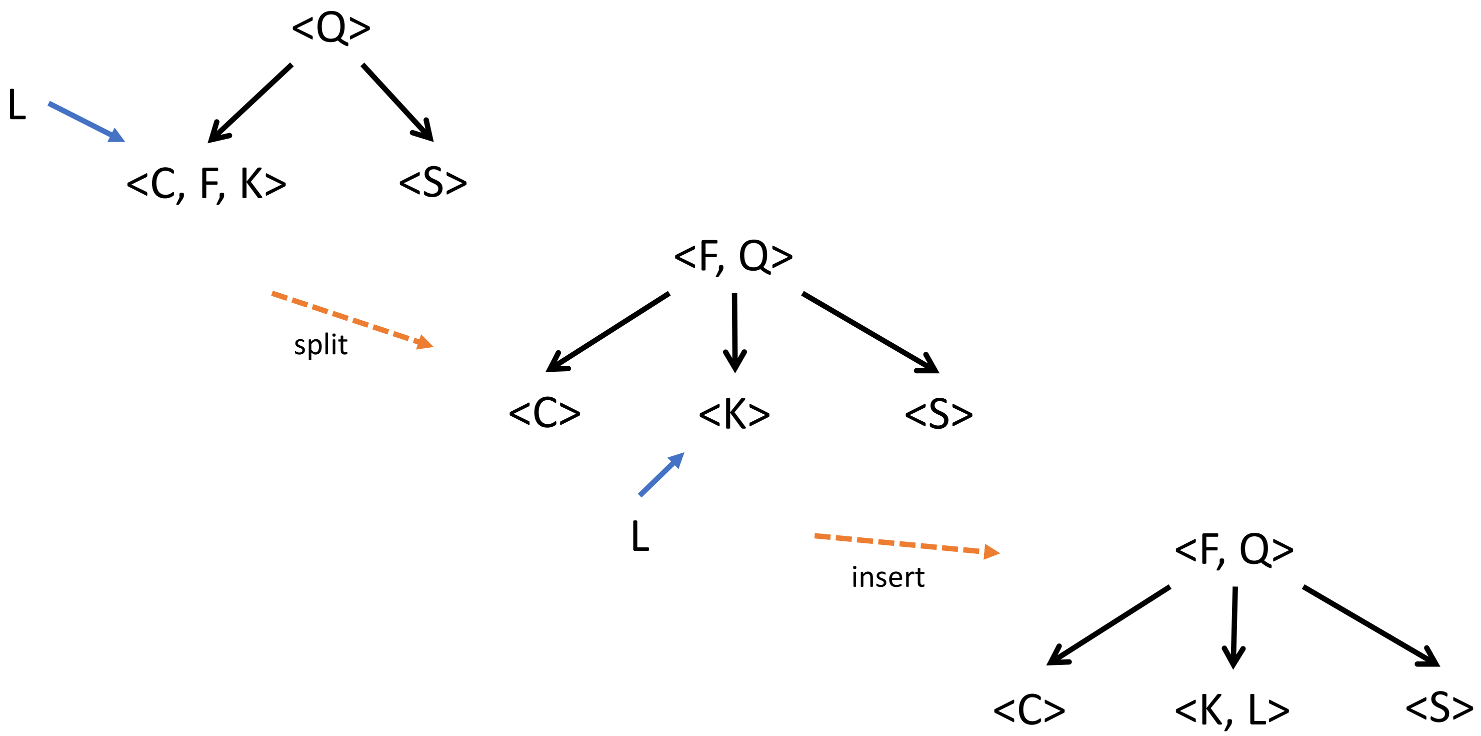
after which K is placed:

insert into initially empty B-tree of min degree t=2 the key values
F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E

place C

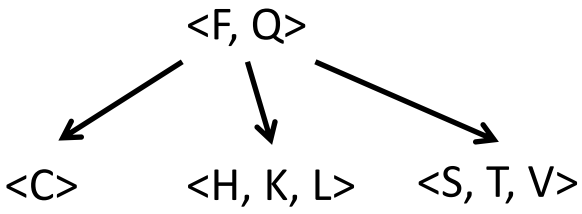


search for L splits full node:

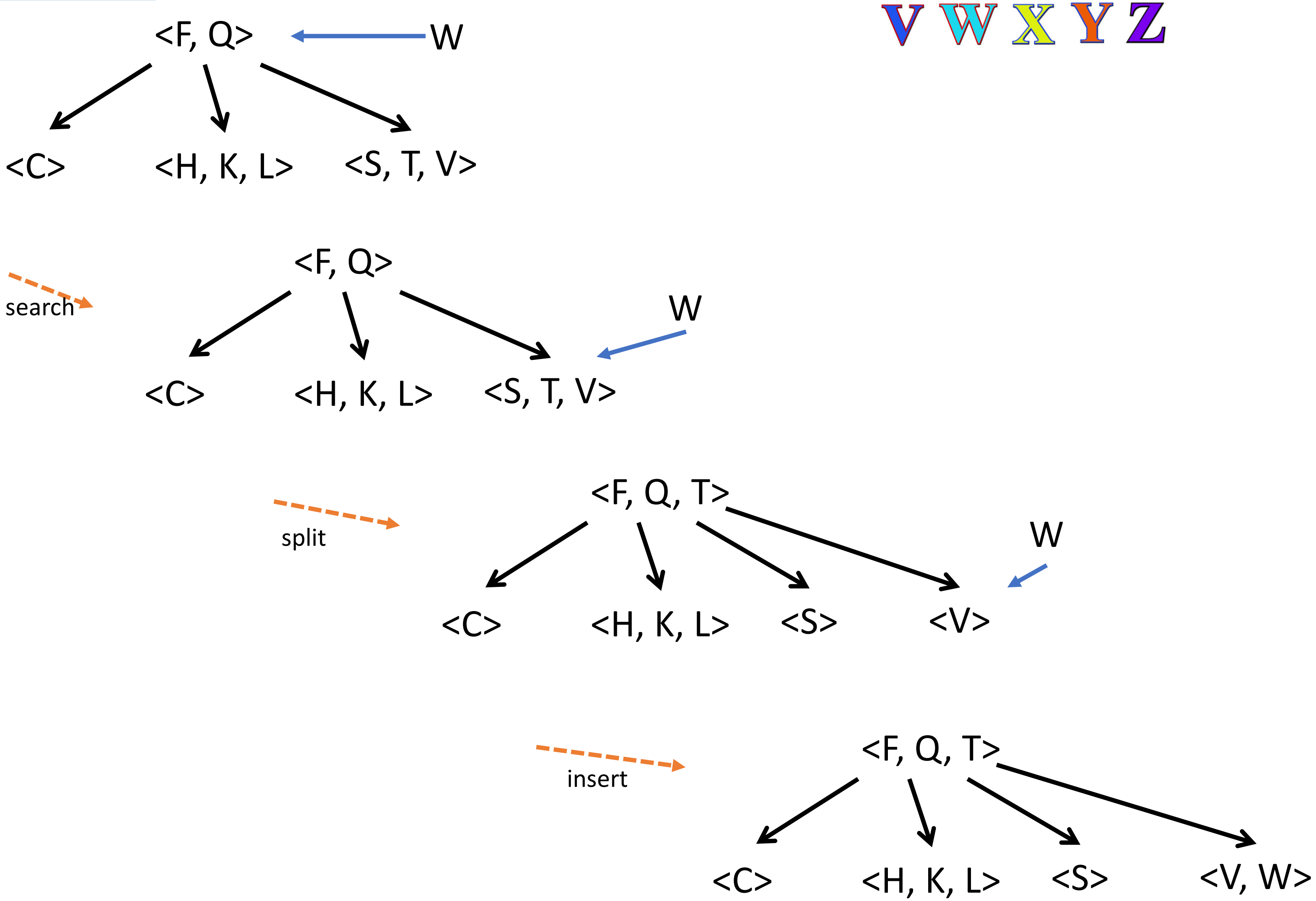


insert into initially empty B-tree of min degree t=2 the key values
F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E

place H, T, V

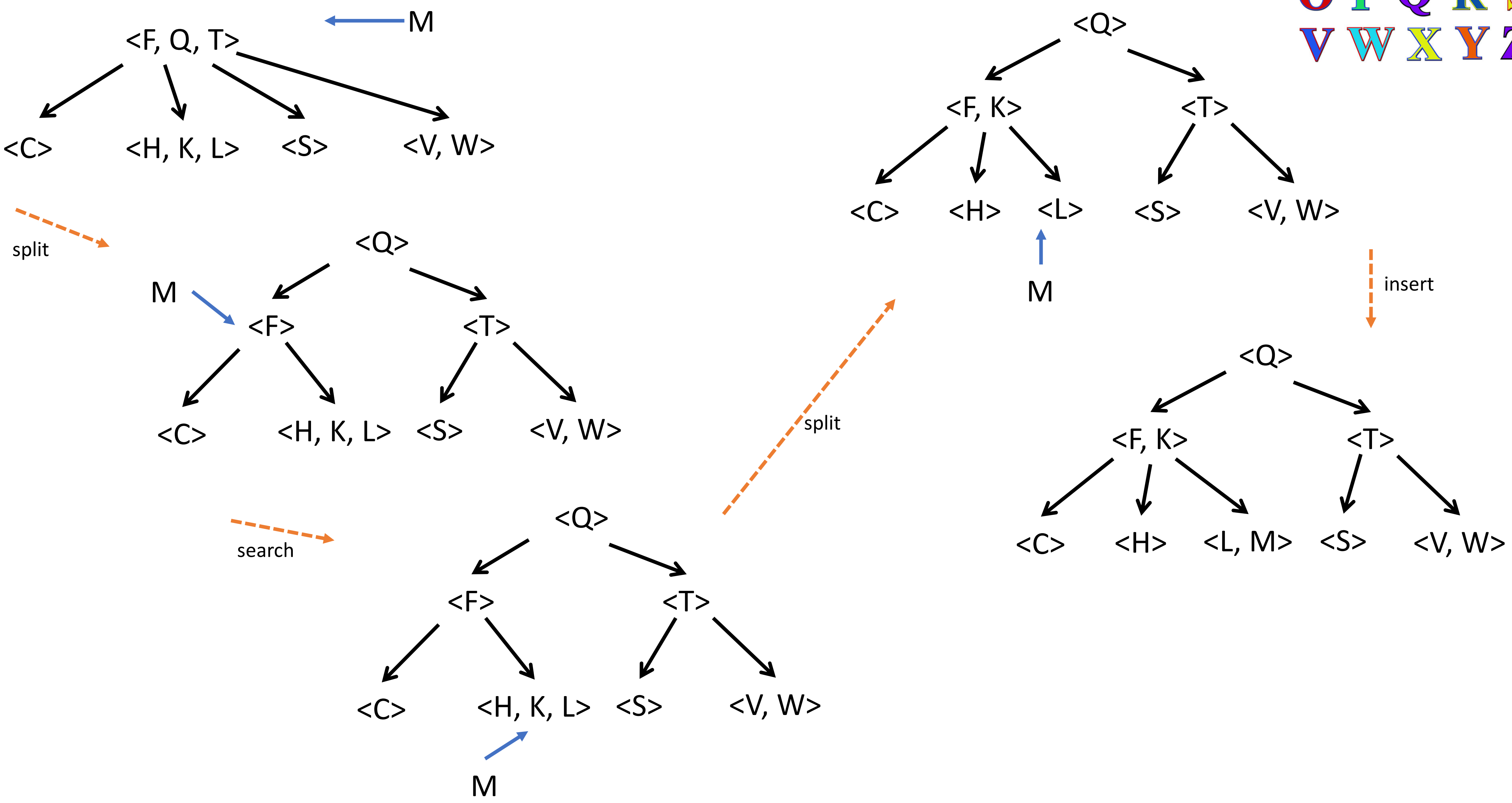


insert W



insert M

insert into initially empty B-tree of min degree t=2 the key values
F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E



ETC….

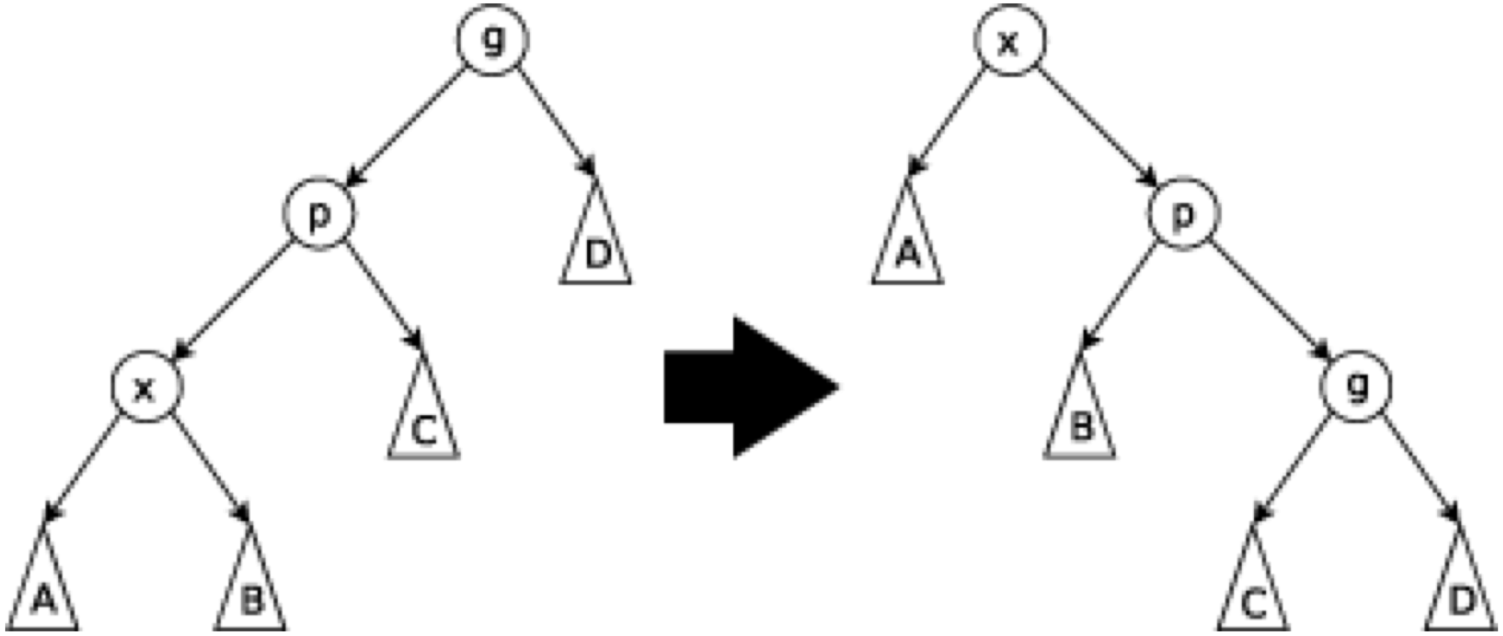
aside: splay trees

- maintain balance kind of
- insertion, deletion, union take O(*lg* n) amortized time
  - a series of m of these operations on n keys takes total time worst case O(m**lg*n)
  - possible that one operation takes O(n) time but cannot happen often
- NO balance information needs to be stored at node (balance factor, color)
- used in DNS servers sometimes

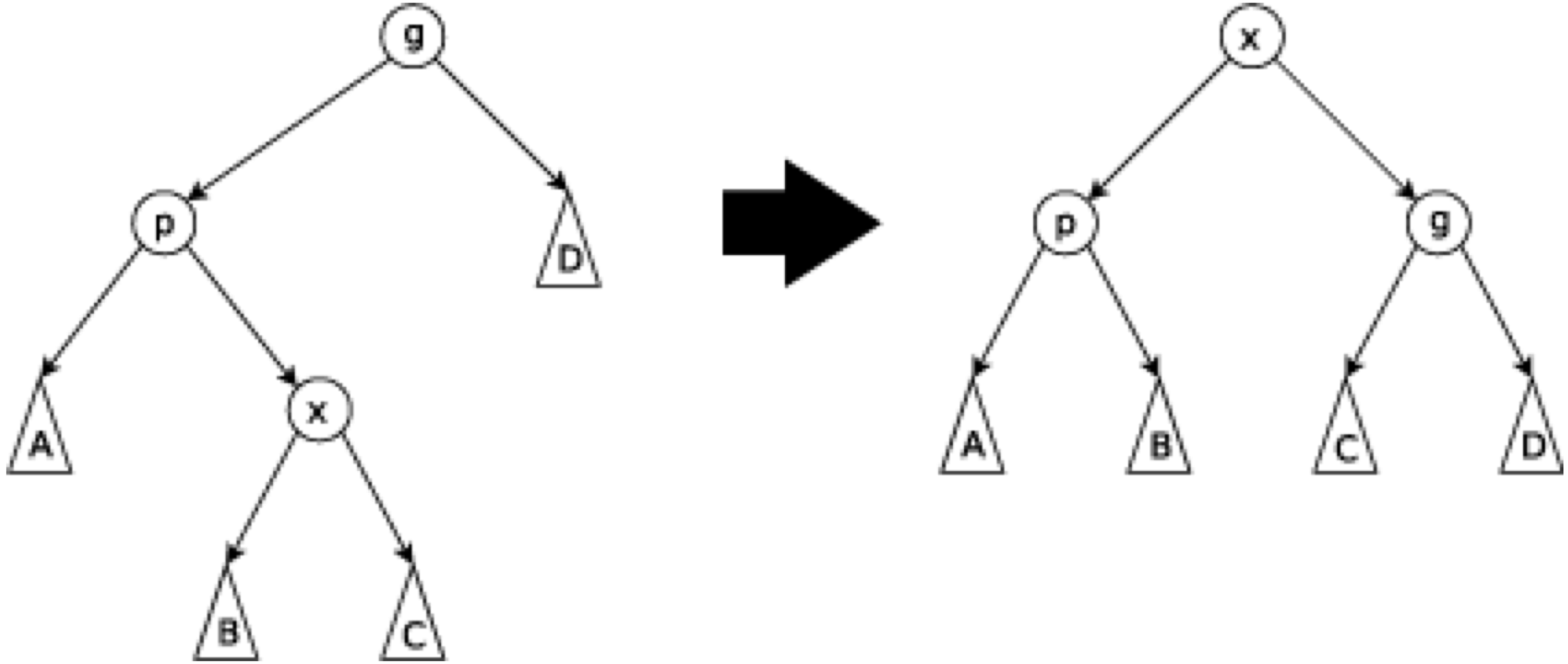
splay trees use rotations

- idea is that whenever a node is accessed, it is moved to the root by a series of rotations
    - LOTS OF ROTATIONS!
    - and slightly different ones
- if x is child of root, it is moved upwards with a single rotation
    - called a ZIG rotation
- if x has a (RL or LR) grandparent, it is moved up with a double rotation
    - called a ZIG-ZAG rotation
- if x has a (LL or RR) rotation, then moved up with a special rotation
    - ZIG-ZIG
- idea: zig-zigs and zig-zags tend towards rebalancing

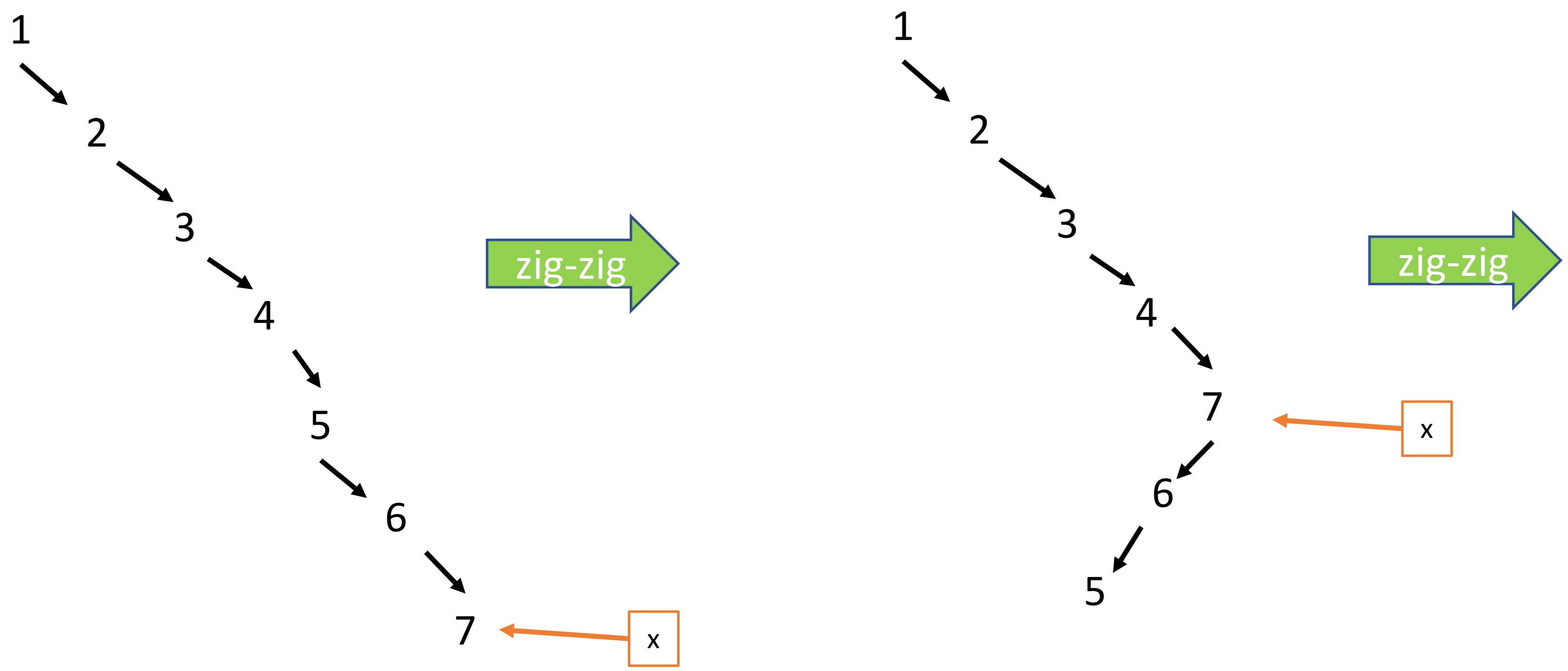
zig-zig

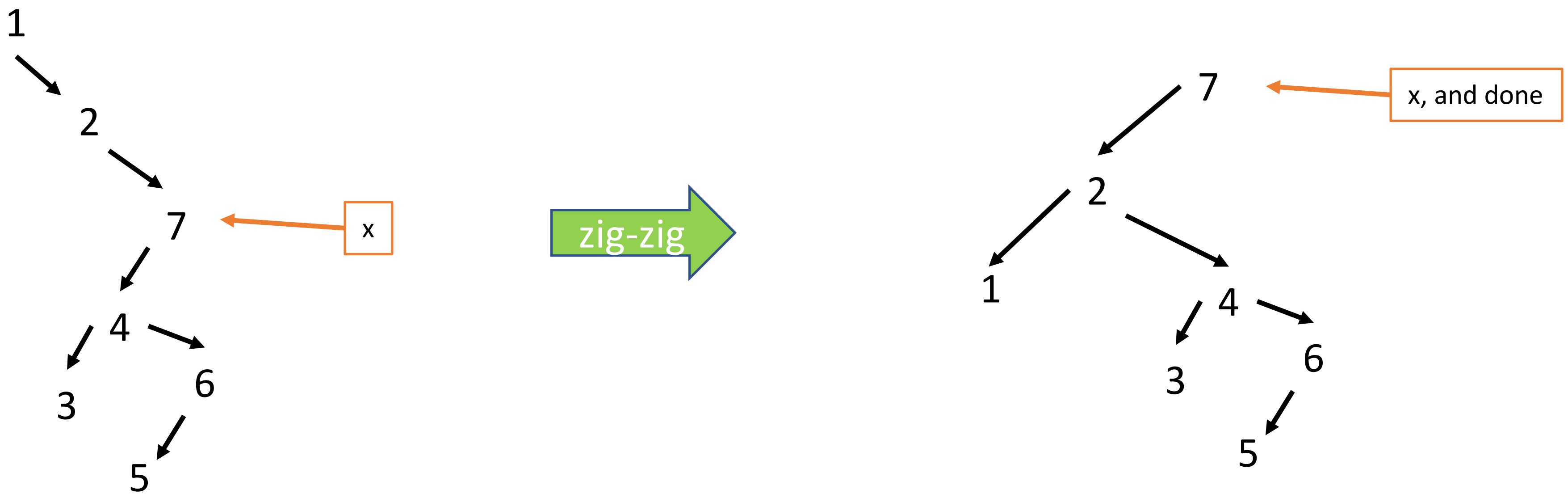


zig-zag



example: find 7




1
2
7
x
4
3
6
5
zig-zig
7
x, and done
2
1
4
3
6
5