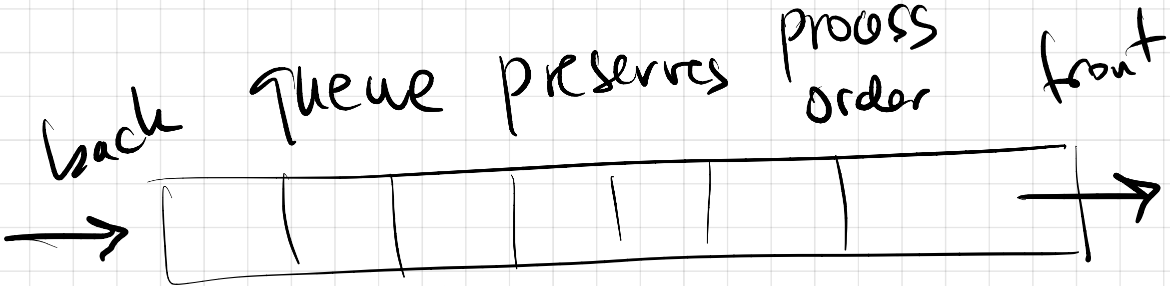
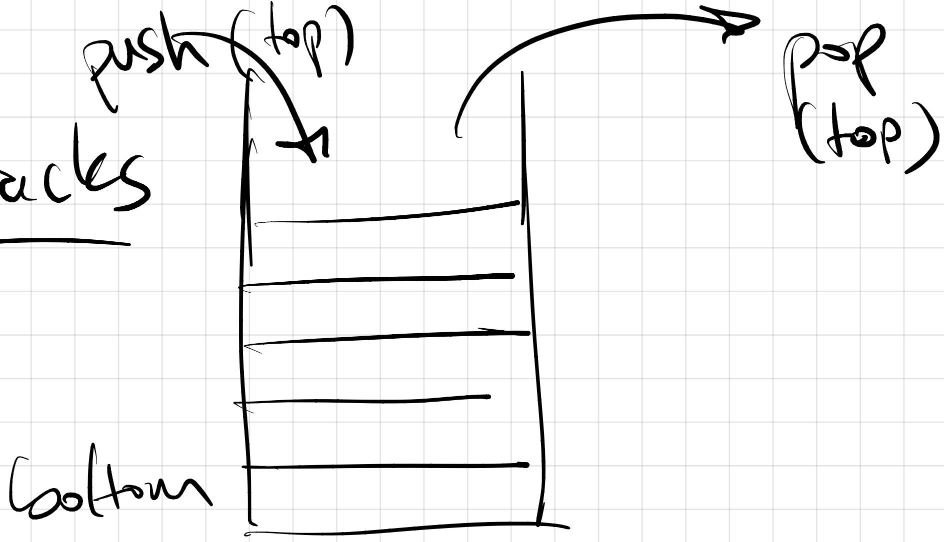
Data structures :

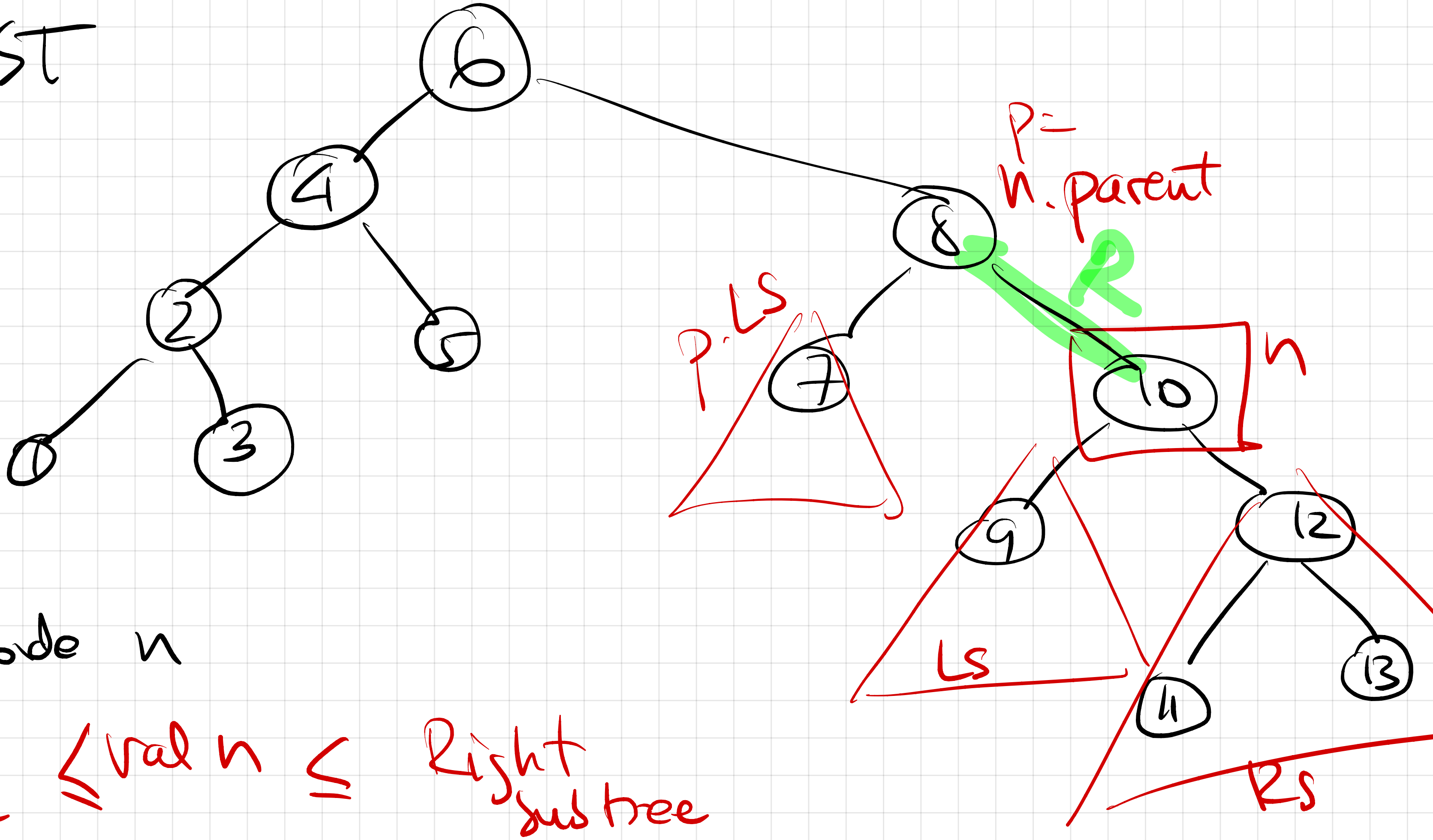
lists	arrays	trees	queues	stacks
- single linked	1D	binary t	FIFO	LIFO
- double linked	2D	BST	- enque	- push
			- deque	- pop



Stacks



BST



∀node n

Left subtree $<$ val n $\leq$ Right subtree

n =? n.parent.Left? NO

? n.parent.Right? Yes →

p.LS $<$ p $<$ n.subtree

p $<$ n.LS

p.LS $\leq$ n

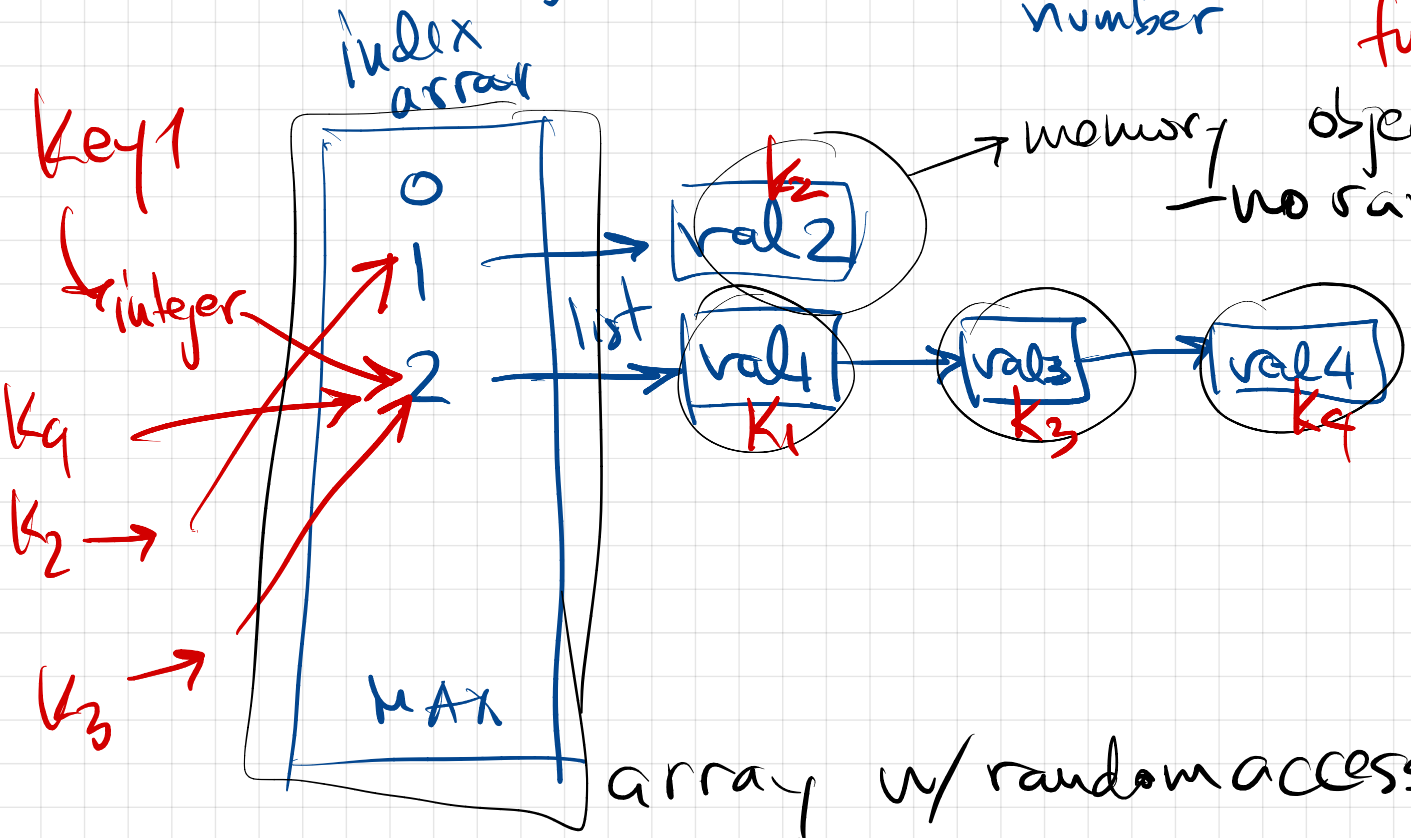
brother, cousins

Adv Datastructures: Hash Tables, RB trees, Skiplists, Fib Heap

Hashes (hash tables)

data = (key, $\boxed{\text{val}}$) pairs $\boxed{\text{val}}$ = val in Hashes

Key = memory object $\longrightarrow$ (large?) integer / number $\xrightarrow[\text{hash function}]{}$ index ∈ Range [0 : MAX]



→ memory objects, pointers (list)
– no random access

array w/ random access

array[index] = memory address of the list of values

- want hash function to be fast
- price = some collisions

n keys 1000,000 $\quad m =$ MAX Range size 100

want collision list avg $\simeq \boxed{\frac{n}{m}} \simeq \frac{1000,000}{100} = 10,000$

$\Rightarrow \alpha = E[\text{list size}]$

SIMPLE UNIFORM HASHING: any key k equally likely to be hashed into any of 0 : MAX

$$\text{prob}[\text{hash}(k) = i] = \frac{1}{\text{MAX}} = \frac{1}{m}$$

- unsuccessful search (key not found) $\Theta(1+\alpha) \simeq \Theta(\alpha)$ usuall $\alpha > 1$
- success search (key found) $\Theta(1+\alpha) \simeq \Theta(\alpha)$

proof (idea)

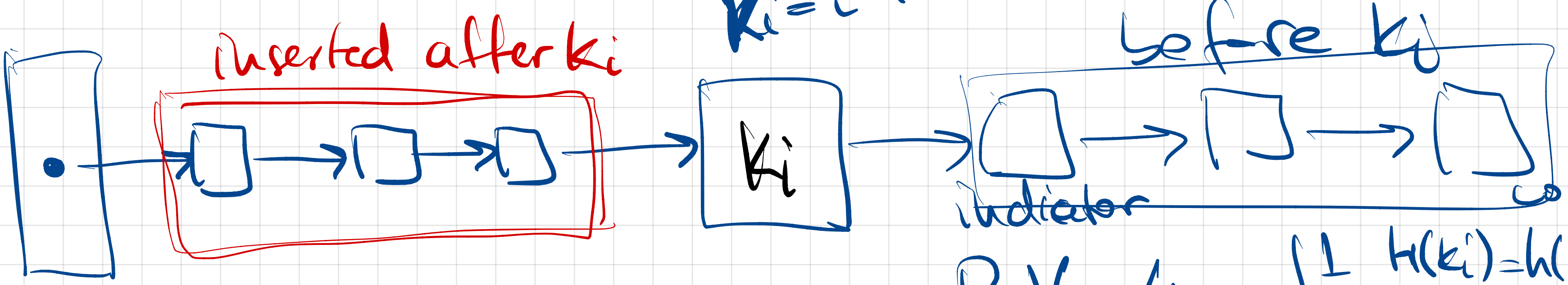
• unsuccess (key not found)

$$E[\text{time}] = E[\text{search in } \overset{i}{\text{list}} \ \text{hash}(k)=i]$$

$$= E[\text{size\_of\_list\_i}] = \alpha = \frac{n}{m}$$

• success $K \in$ Hash

k_i = i th inserted key $\quad$ k_i, k_j keys



indicator $\quad$ collision

$$\text{R.V. } X_{ij} = \begin{cases} 1 & h(k_i)=h(k_j) \\ 0 & h(k_i)\neq h(k_j) \end{cases}$$

$E[X_{ij}] = \text{prob}(X_{ij}=1) = \text{prob}$ later key gets hashed in list(first key) $= \frac{1}{m}$

$$\text{search time} = E\left[\frac{1}{n}\sum_{i=1}^{n}\underbrace{\left(1+\boxed{\sum_{j=i+1}^{n} X_{ij}}\right)}_{\text{search time for } k_i}\right]$$

Keys inserted after k_i colliding with k_i

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} E[X_{ij}]\right) = \frac{1}{n}\sum_{i=1}^{n}\left(1+\underbrace{\sum_{j=i+1}^{n}\frac{1}{m}}_{n-i \text{ terms}}\right)$$

$$= 1 + \frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$0 + 1 + 2 \ldots + n-1$

$$= 1 + \frac{1}{\cancel{n}m}\frac{(n-1)\cancel{n}}{2} = \Theta\left(1+\frac{n}{m}\right) = \Theta(1+\alpha)$$

Hash functions.

$h(\text{key}) = \text{indexe}[0 : MAX]$

$\text{key} \xrightarrow{?} \text{integer}$

heuristic

HW8

$\text{key} = \text{word} \longrightarrow \text{integer } x$ (very large) $\xrightarrow{h} \text{index}$

ex 701

$MAX = \text{far from } 2^p$

basic $h(x) = x \underbrace{\bmod}_{\text{remainder}} MAX \in [0 : MAX-1]$

$$h(x) = \left\lfloor m \cdot \text{fractional}(x \cdot A) \right\rfloor$$

MAX

power of 2

$$\text{fractional}(z) = z - \text{integer}(z)$$

$$A = \text{fractional of } \frac{S}{2^w}$$

w = bits required by MAX

Hash-functions - universal-set

- adversary: chooses many keys $h(k) = \text{same}$
  ⇒ long list of a
  because $h = \text{fixed}$.

Solution: $S = \{\text{hash functions}\}$ <u>universal</u>

• every hash created ⇒ $h \in S$ picked at random.

$|S|$ = size of the set of functions

UNIVERSAL

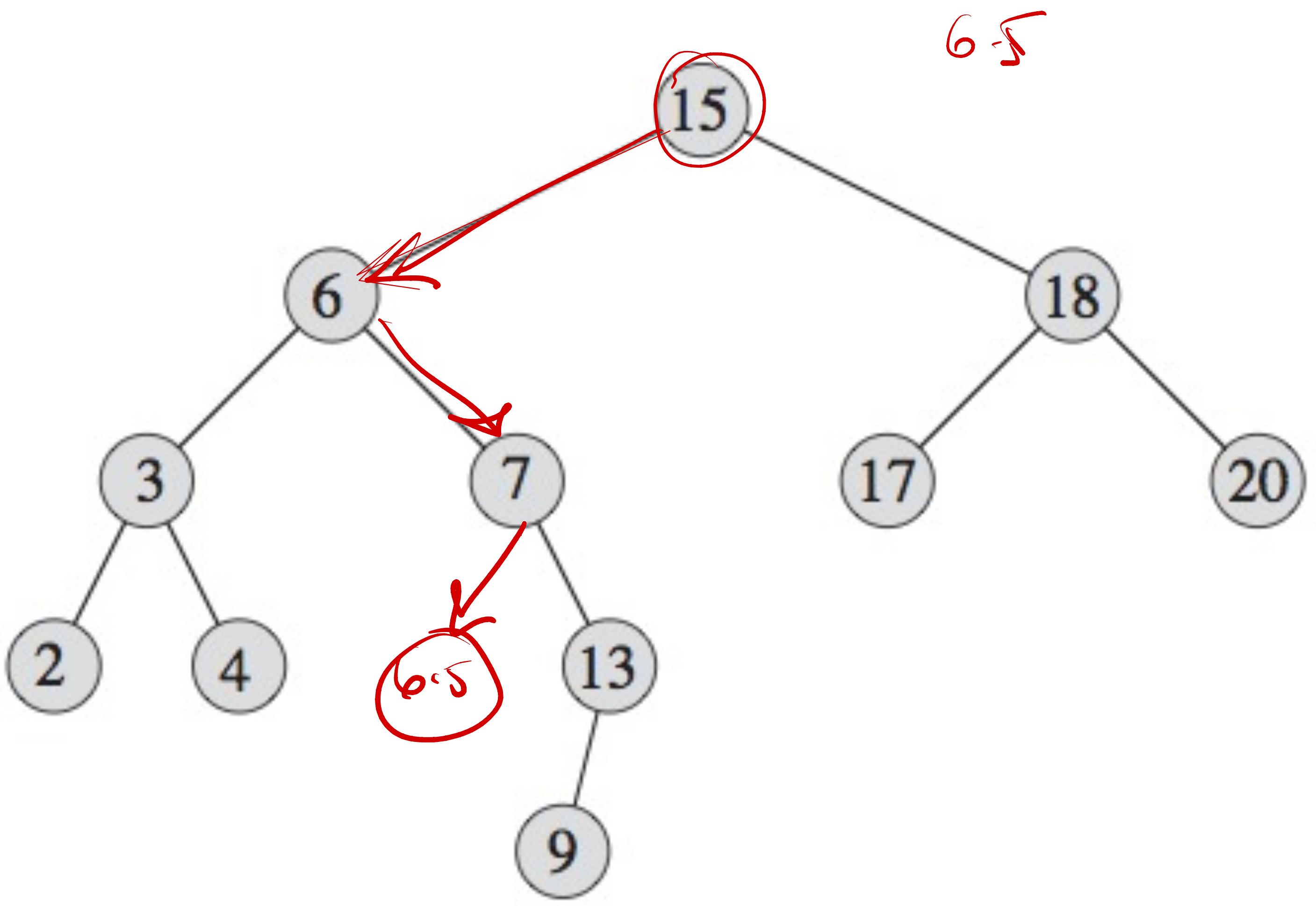
every 2 keys k, e

$$\{h \in S \mid h(k) = h(e)\} \leq \frac{|S|}{MAX}$$

(Th) Adversary cannot create long list of collisions

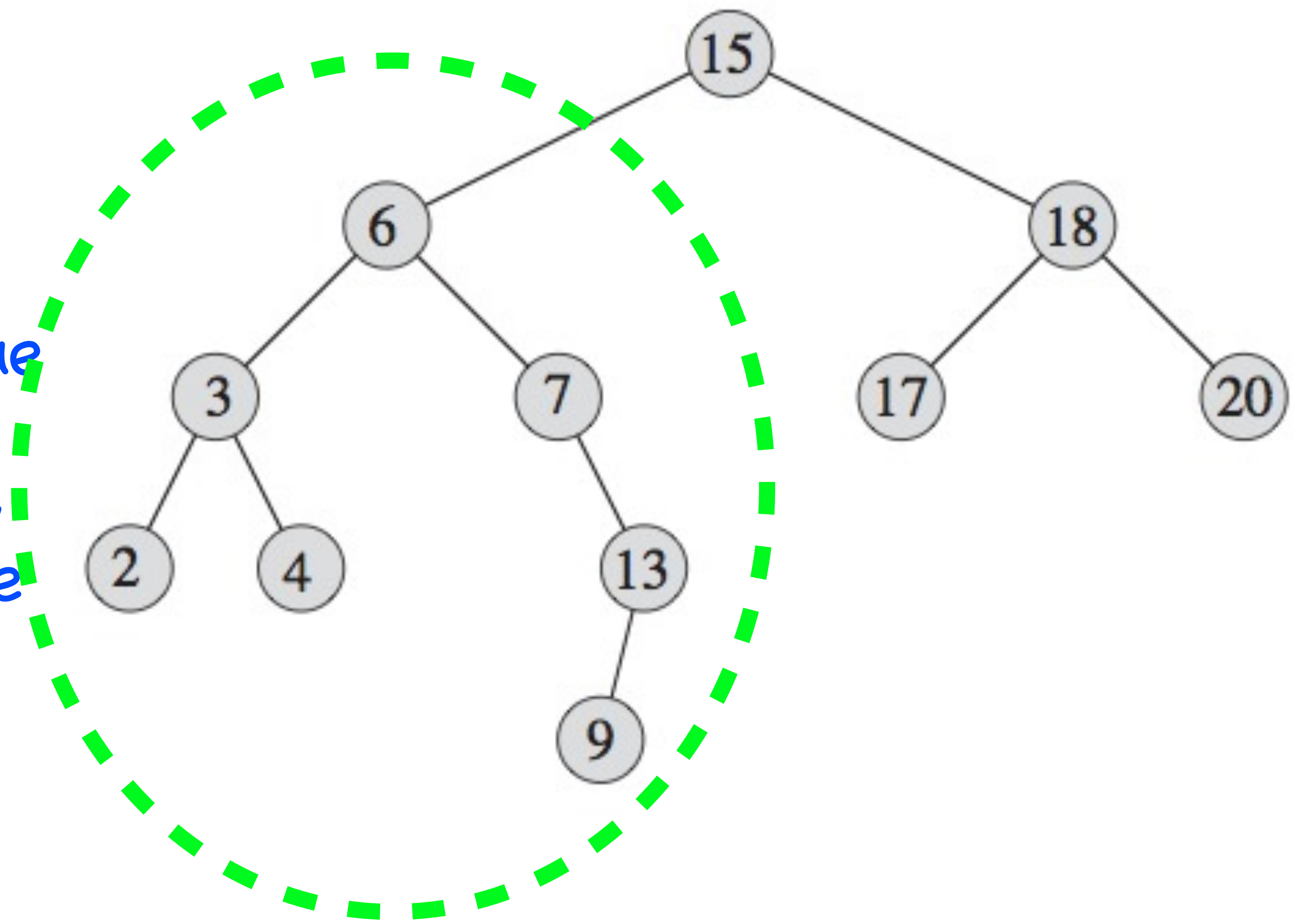
Binary Search Trees - Recap

- each node has at most two children
- any node value is
  - not smaller than any value in the left subtree
  - not larger than than any value in the right subtree
  - h = height of tree
- Operations:
  - search, min, max, successor, predecessor, insert, delete
  - runtime O(h)



Binary Search Trees - Recap

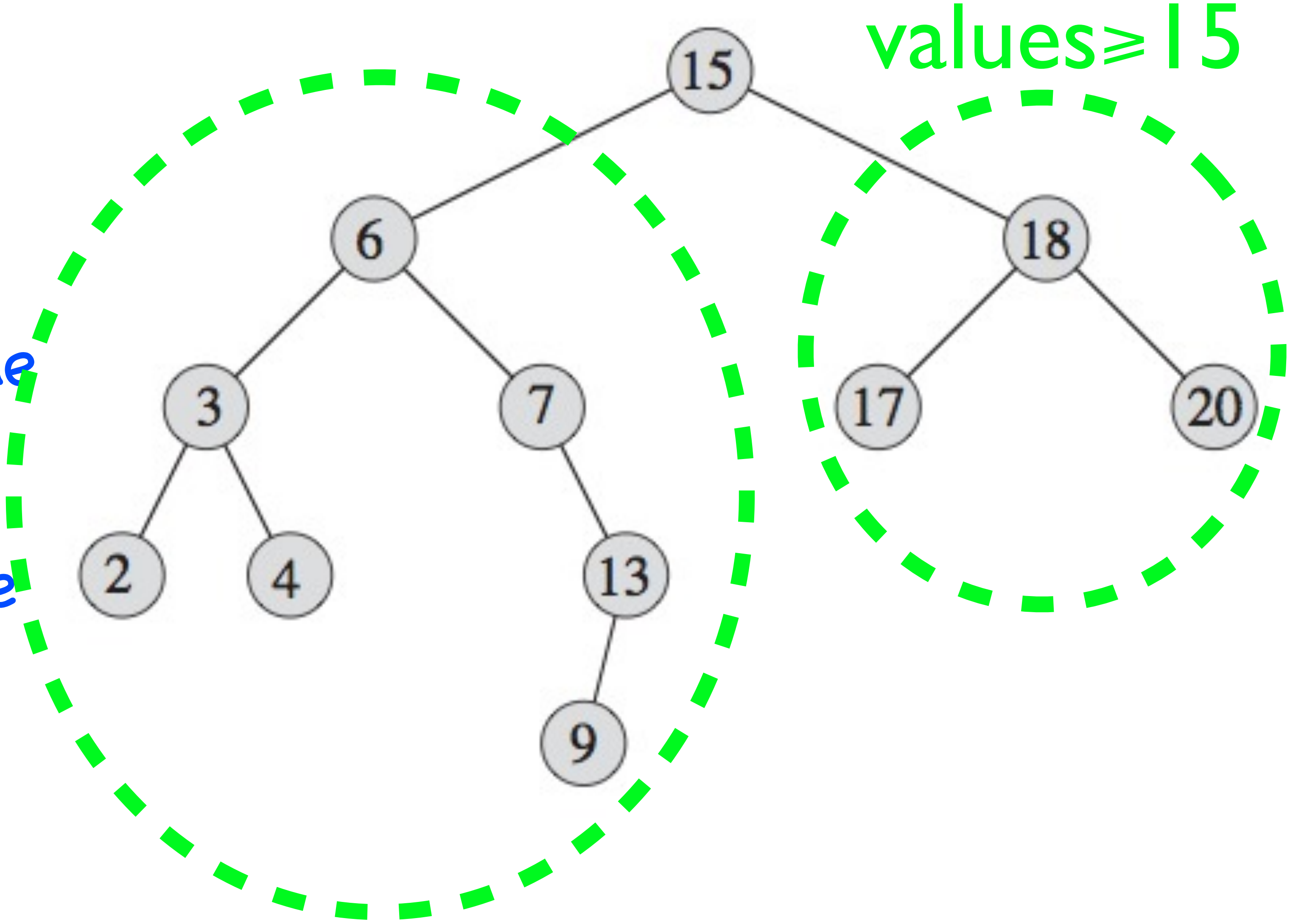
- each node has at most two children
- any node value is
  - not smaller than any value in the left subtree
  - not larger than than any value in the right subtree
  - h = height of tree
- Operations:
  - search, min, max, successor, predecessor, insert, delete
  - runtime O(h)



left subtree values≤15

Binary Search Trees – Recap

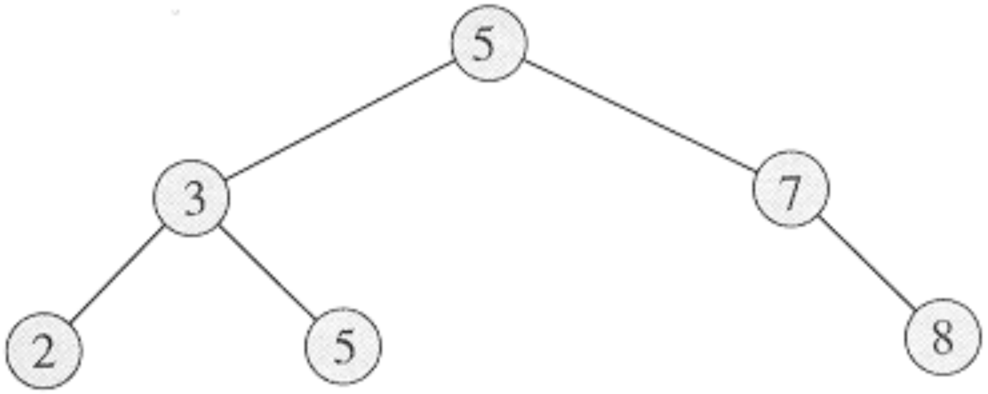
- each node has at most two children
- any node value is
  - not smaller than any value in the left subtree
  - not larger than than any value in the right subtree
  - h = height of tree
- Operations:
  - search, min, max, successor, predecessor, insert, delete
  - runtime O(h)



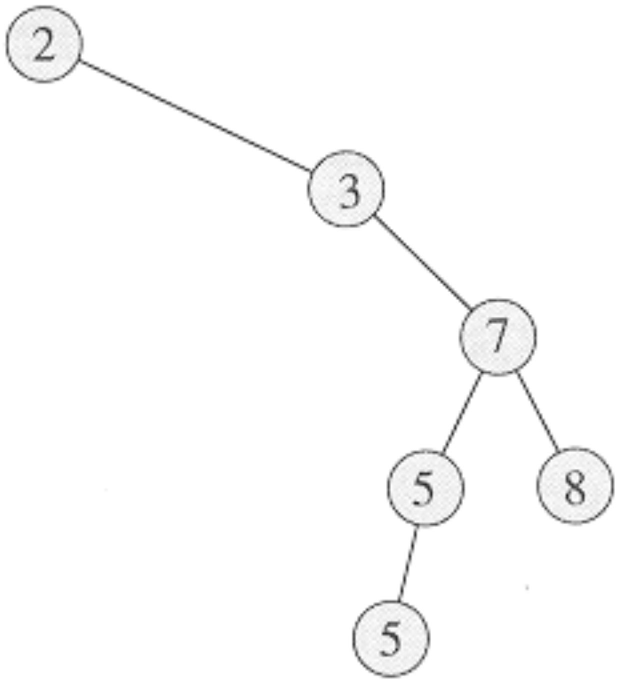
right subtree values≥15

left subtree values≤15

Balanced Trees



(a)

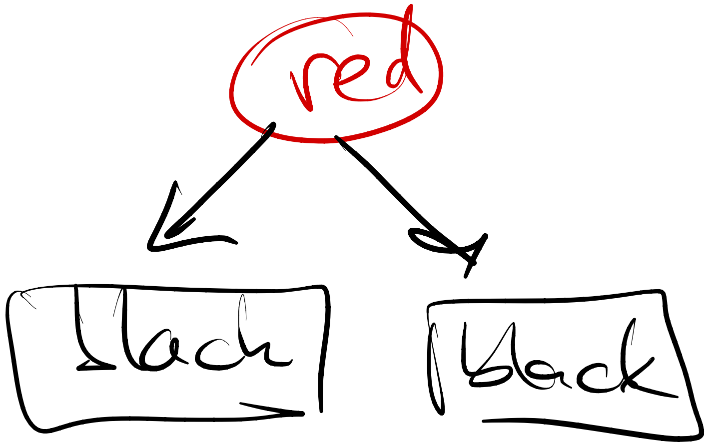


(b)

- a) balanced tree: depth is about log(n) - logarithmic
- b) unbalanced tree : depth is about n - linear

Red-Black Trees

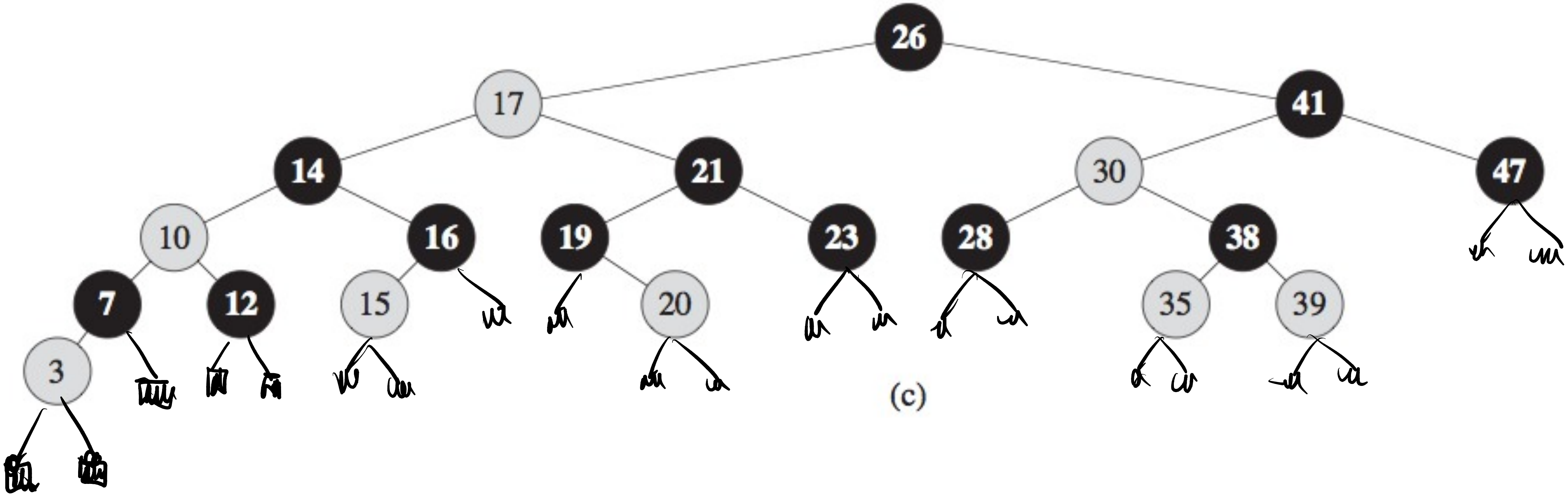
- binary search tree
- want to enforce **balancing** of the tree
  - height logarithmic in n=number of nodes in the tree
  - height = longest path root->leaf
- extra: each node stores a color
  - color can be either red or black
  - color can change during operations



- **red-black properties**
  - root is black
  - leafs (terminals) are black
  - if a node is red, then both children are black
  - for any given node, all paths to leaves (node->leaf) have the same number of black nodes

=> balanced on black nodes.

Red-Black Trees



(c)

- Theorem: a red-black tree with n nodes has height at most 2*log(n+1)
  - or logarithmic height
  - thus enforcing the balancing of the tree
  - and so the all operations can be implemented in O(log n) time.

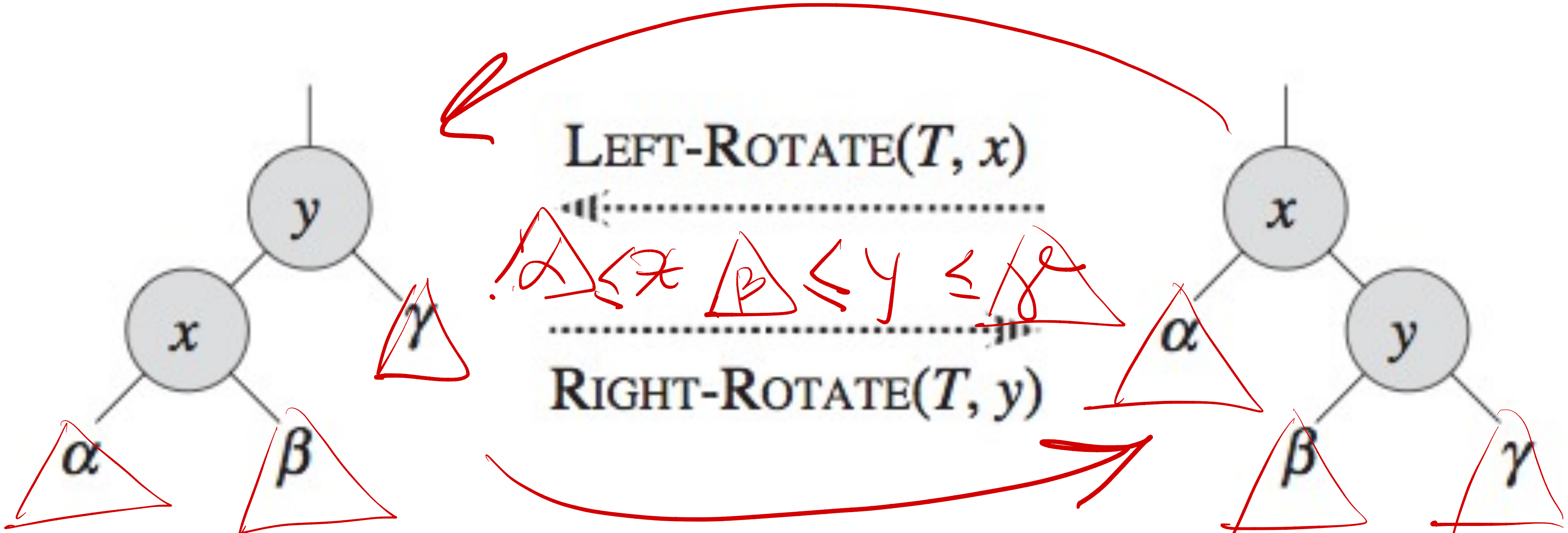
Tree operations

- insert, delete - need to account for colors
  - rest of the lecture: insert and delete in red-black trees
- search, min, max, successor, predecessor - same as for regular binary search trees

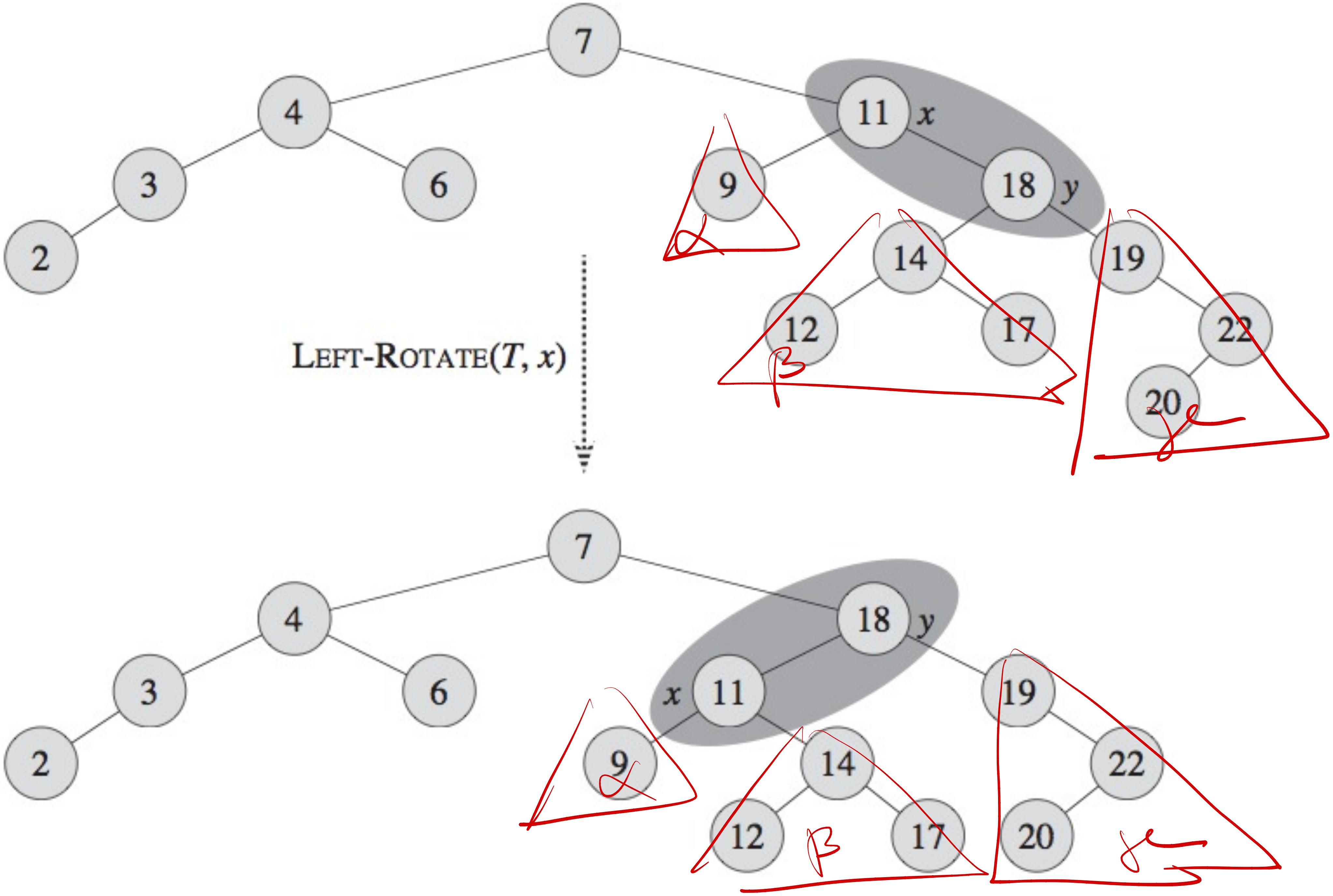
Red-Black Trees - Rotation

- **Rotation** is a utility operation that facilitates maintenance of red-black properties
  - during insert and delete, the tree might temporarily violate the red-black properties
  - using rotation we can fix the tree so it satisfies red-black.

- Rotate-left at node x
  - x is replaced by its right child y
  - β = left subtree of y becomes right subtree of x
  - x becomes the left child of y
- Rotate-right at y symmetric



Red-Black Trees – Rotation



- Example

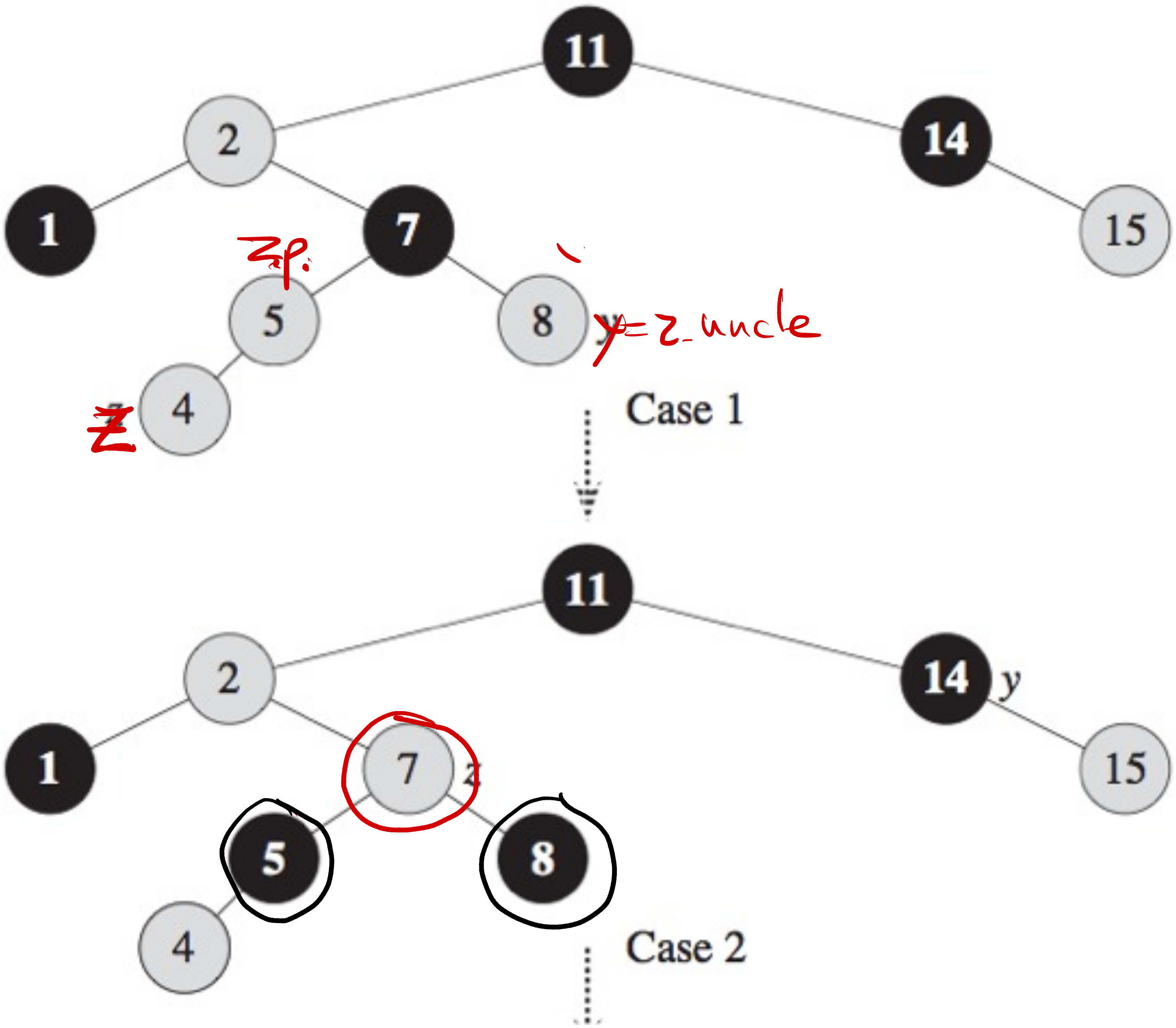
Red-Black Trees – Insertion

- add node "z" as a leaf
  - like usual in a binary search tree
- color z red, add terminal "NIL" nodes
- check red-black conditions
  - most conditions are still satisfied or easy to fix
  - the real problem might be the condition that requires children of red nodes to be black.
  - start fixing at the new node z, and as we proceed more fixes might be necessary
  - three "fixing cases"
  - overall still O(log n) time.
- RB-INSERT-FIXUP procedure in the textbook

Fixing insertion case 1

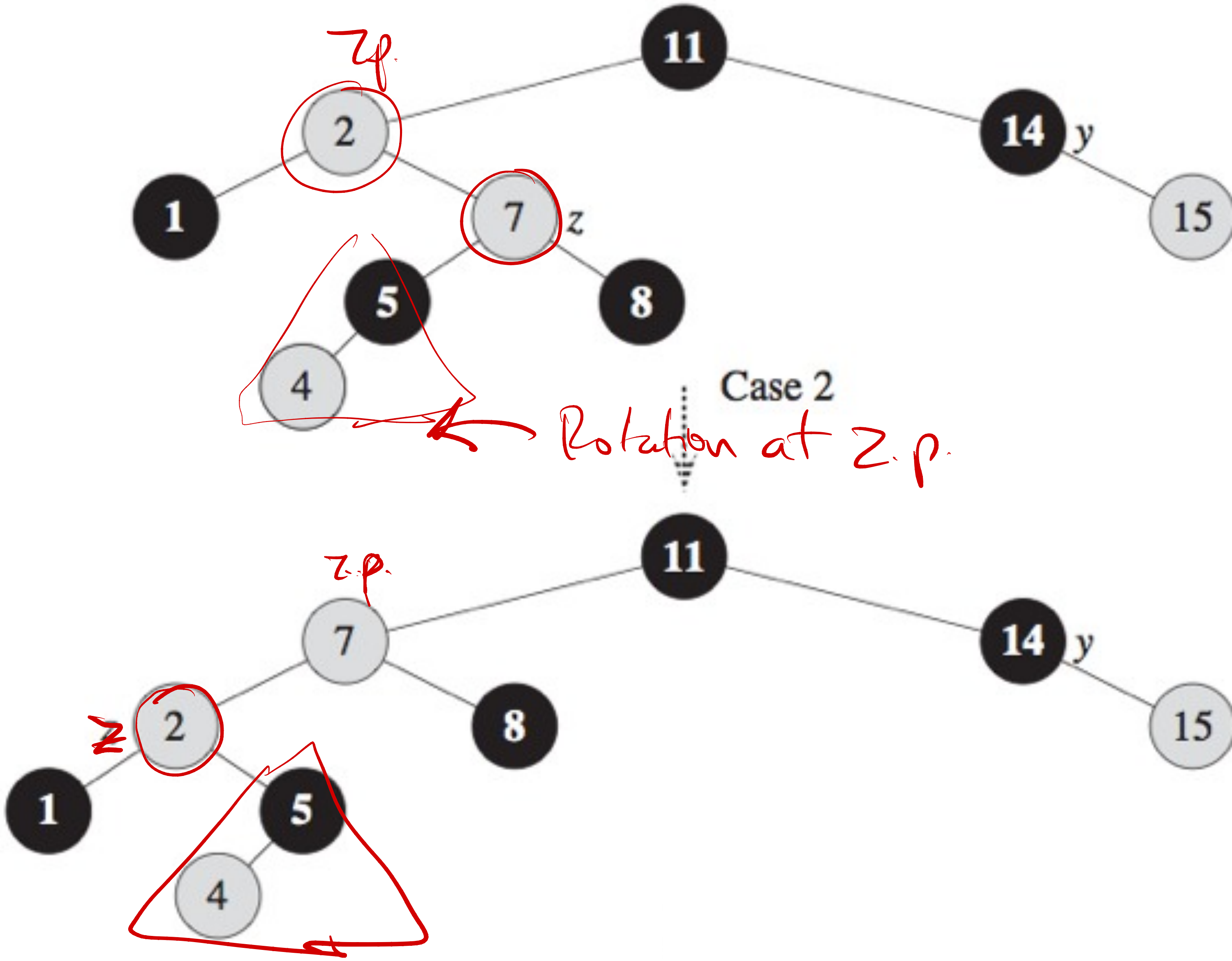
- z.p = z.parent and y=z.uncle are red
- fix:
  - make z.p and y black
  - make z.p.p red
  - advance z to z.p.p

"fixed locally, move up"



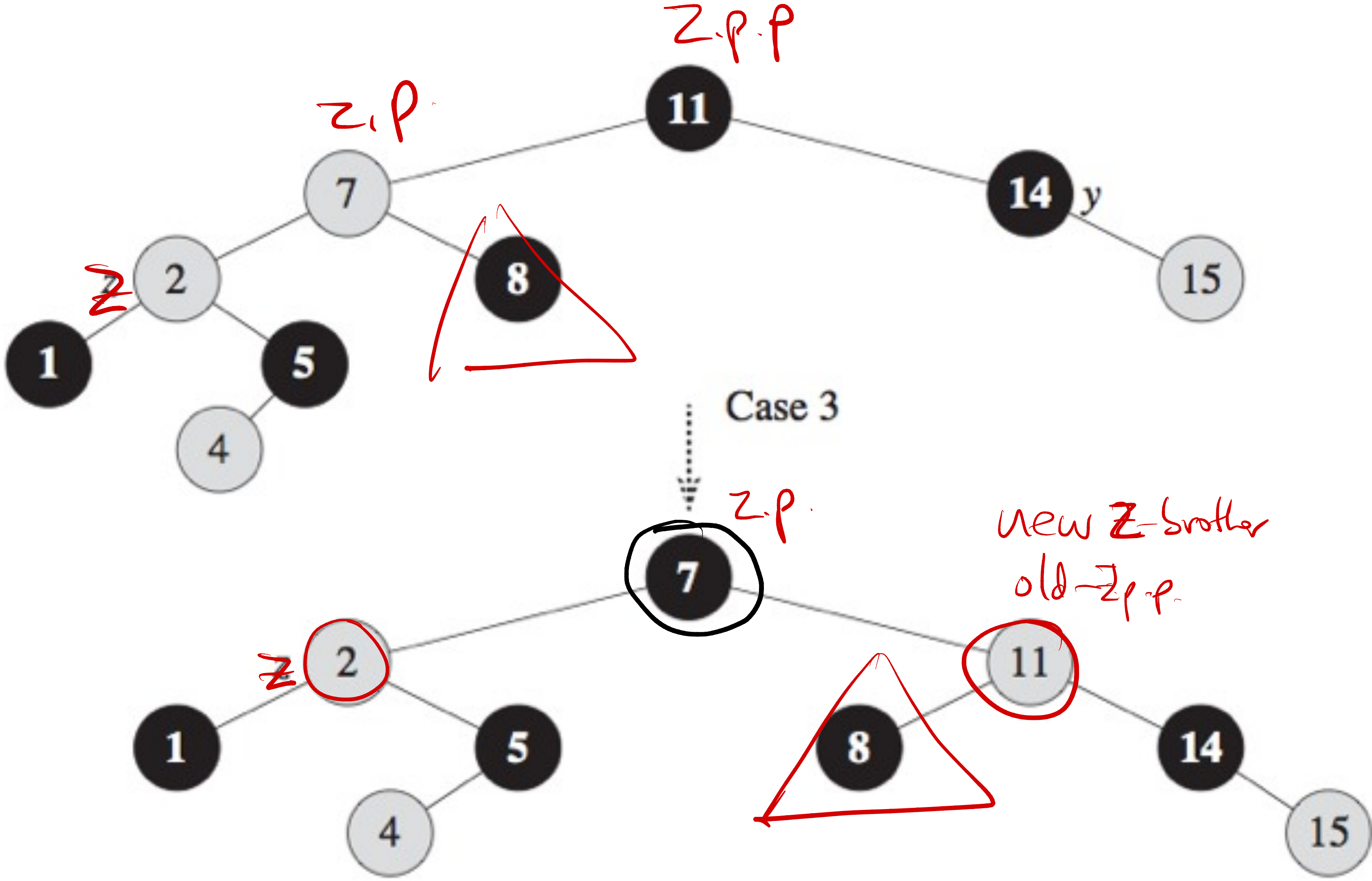
Fixing insertion case 2

- z.p is red, y is black, z is the right child
- fix:
  - rotate left at z.p
  - z advances to its old parent (now his left child)



Fixing insertion case 3

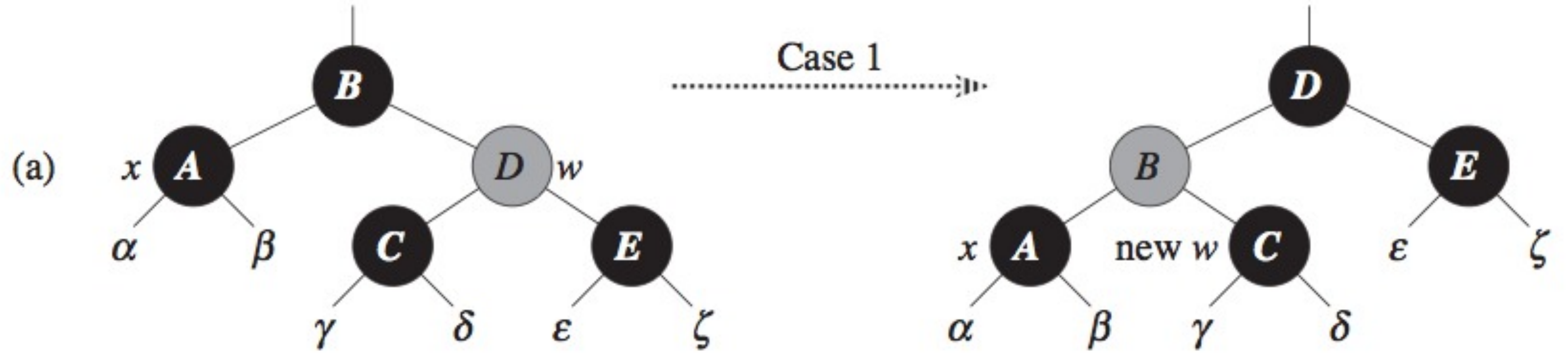
- z.p red, y black, z is left child
- fix:
  - rotate right at z.p.p
  - color z.p black
  - color old z.p.p (now z brother) red



Red-Black Trees – Deletion

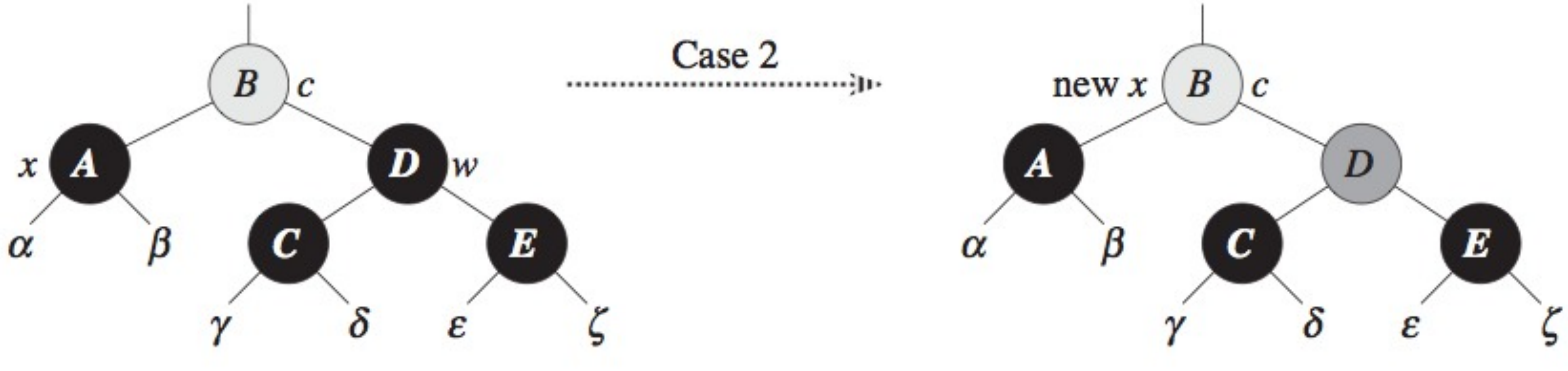
- delete “z” as we usually delete from a binary search tree
  - maintain search property: left values≤ node value ≤ right values
- additionally keep track of
  - y= the node to replace z
  - y original color (its color might change in the process)
- Fix-up the tree red-black properties, if they are violated
  - a procedure with 4 cases
  - RB-DELETE-FIXUP procedure in the textbook

Fixing deletion case 1



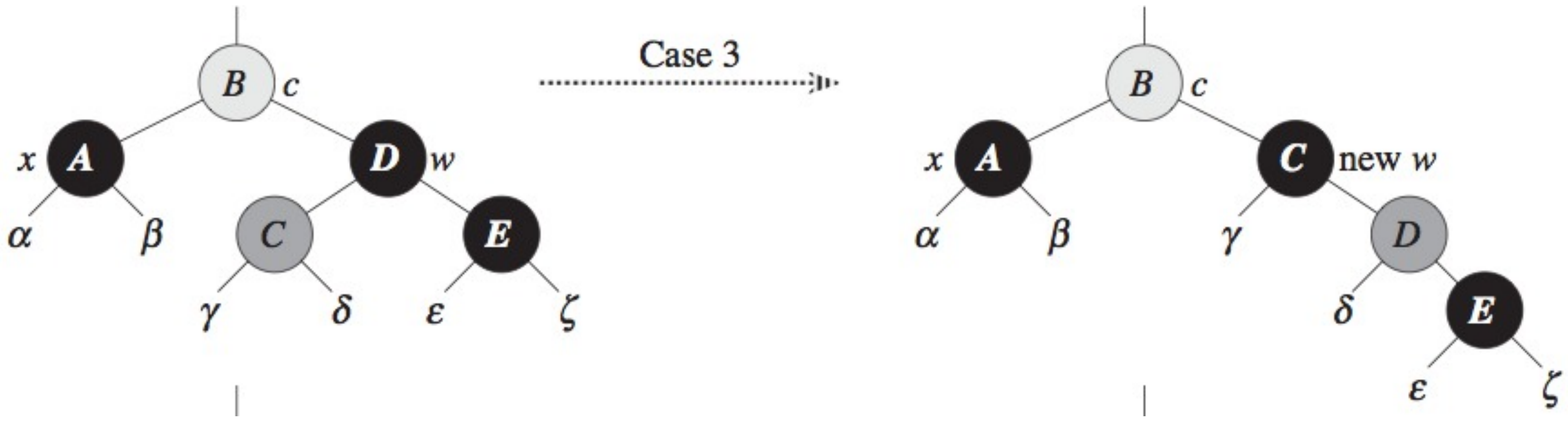
- case 1: x is black, brother w red
- fix :
  - rotate left at x.p;
  - color x.p red;
  - color w (now x.p.p) black

Fixing deletion case 2



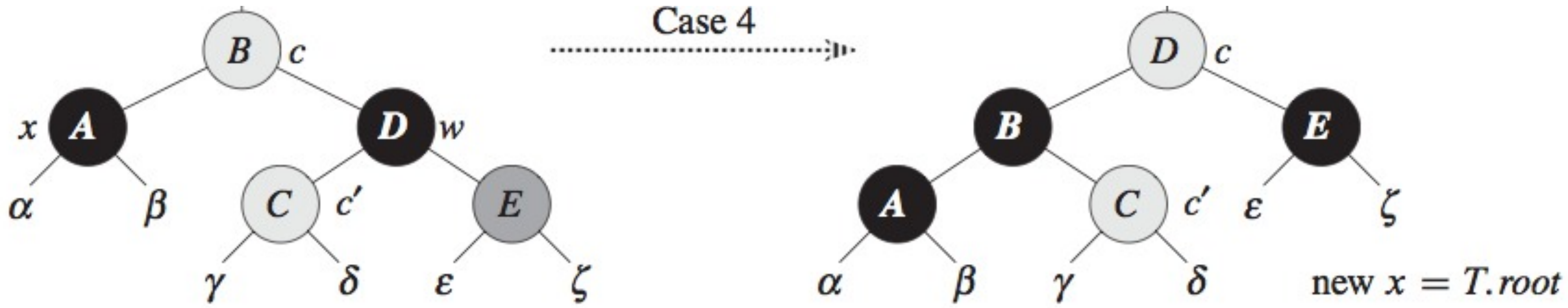
- case2: brother w is black, and w children also black
- fix:
  - color w red
  - advance x to its parent

Fixing deletion case 3



- case3: brother w is black; w's left child is red; w's right child is black
- fix:
  - rotate right at w
  - color the new brother from red to black
  - color the old brother from black to red

Fixing deletion case 4



- case4: brother w is black, w's right child is red
- fix:
  - rotate left at x.p
  - color old w's right child from red to black
  - color x.p from red to black
  - color old w from black to red

Running time

- most BST operations same running time as BST trees
  - search, min, max, successor, predecessor
  - these dont affect RB colors
- Insertion including fixup O(log n)
- Deletion including fixup O(log n)