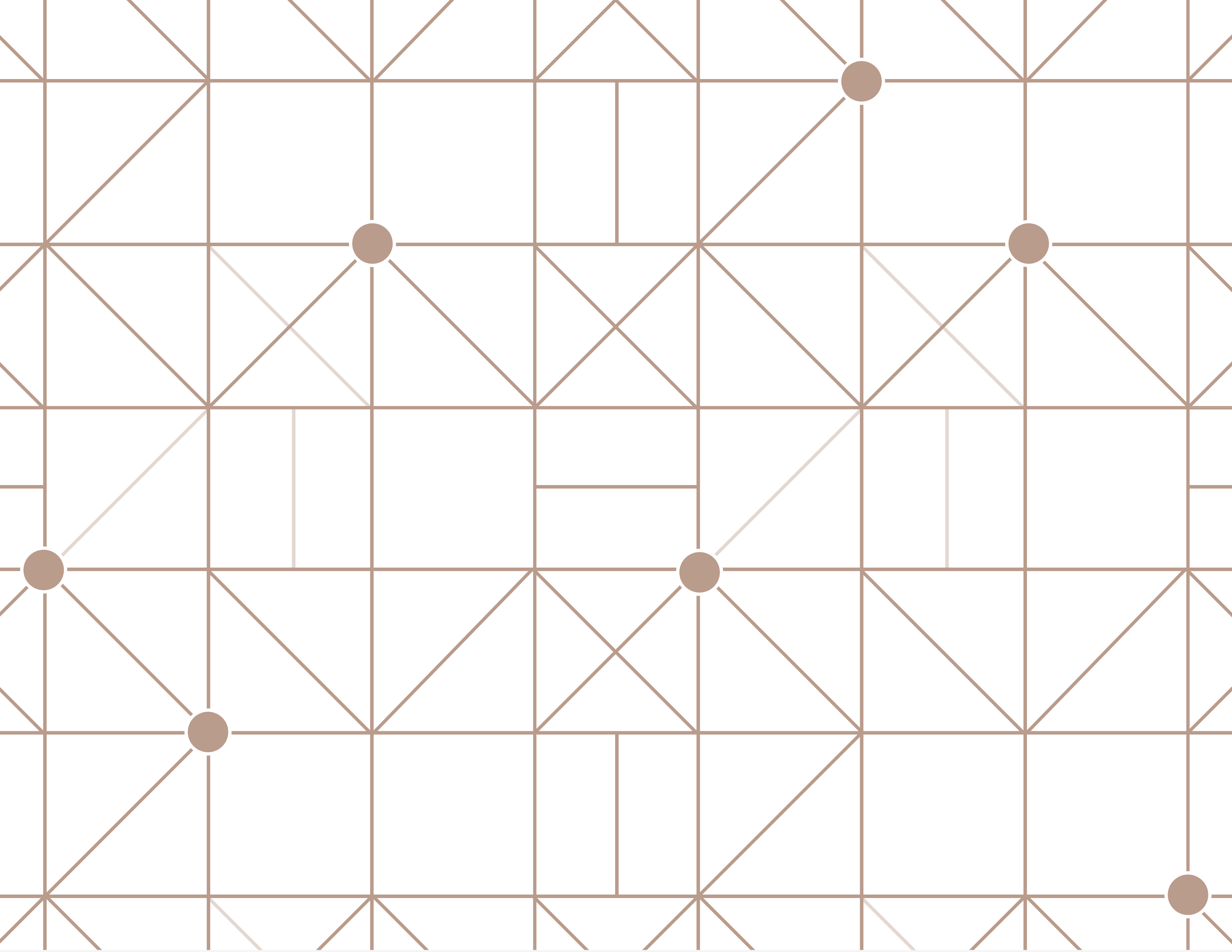


Wed 3/17

- midterm problems
- Graphs SP representation of DP (J. Aslam)

- Hash Tables
- Red Black Trees
- Skiplists

- Recap Datastructures (indiv. study)

midterm (d) $T(n) = T(n-1) + T(n-2) + 1$

assume $T(n) = \Theta(a^n)$ exp

$\cancel{c}a^n = \cancel{c}a^{n-1} + \cancel{c}a^{n-2} + \cancel{1}$?

$\boxed{a^2 = a + 1}$ Fibonacci

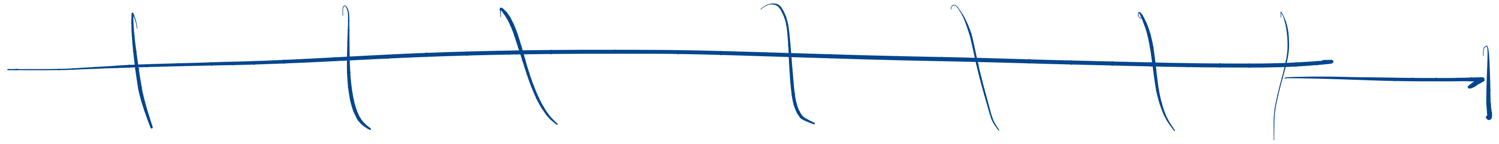
$$a^n = a^{n-1} + a^{n-2} + 1$$

$$a^2 = a + 1 + \frac{1}{a^{n-2}}$$???

Lower Bound $T(n) = \Omega(\text{Fibonacci})$

Upper bound $\boxed{T(n) \le c \cdot \varphi^n}$??

Rod Cuts



OBJ : max Π(lengths)

Greedy choice

cut "3" as much as possible

$$\boxed{\underbrace{3\cdot 3\cdot 3\cdots 3}\cdot 2}$$ ✓

or

$$3\cdot 3\cdot 3\cdots\underset{2\cdot 2}{\boxed{\cancel{3\cdot 1}}}$$ ✓

- any $\underline{k \geq 5} \Rightarrow 3 \cdot (k-3)$
- $4 \rightarrow 2 \cdot 2$

left $3 \cdot 3 \cdot 3 \cdot \cdots \cdot 3 \cdot \underbrace{2 \cdot 2 \cdot 2 \cdot \cdots \cdot 2}_{t}$

$t \geq 3 :$ $2 \cdot 2 \cdot 2 \rightarrow 3 \cdot 3$

$t \in \{1, 2\}$

Answer $3 \cdot 3 \cdot 3 \cdot \cdots \cdot 3 \cdot 2^{t}$

$t \in \{1, 2, 0\}$.

A = set of numbers $\{a_1, a_2, \ldots a_n\}$

Partition indices $B \cup C = \{1, 2, \ldots, n\}$

$B \cap C = \emptyset$

$$\left| \sum_{i \in B} a_i - \sum_{j \in C} a_j \right| = \min$$ (BALANCE)

Knapsack $W = \frac{\sum a_i}{2}$

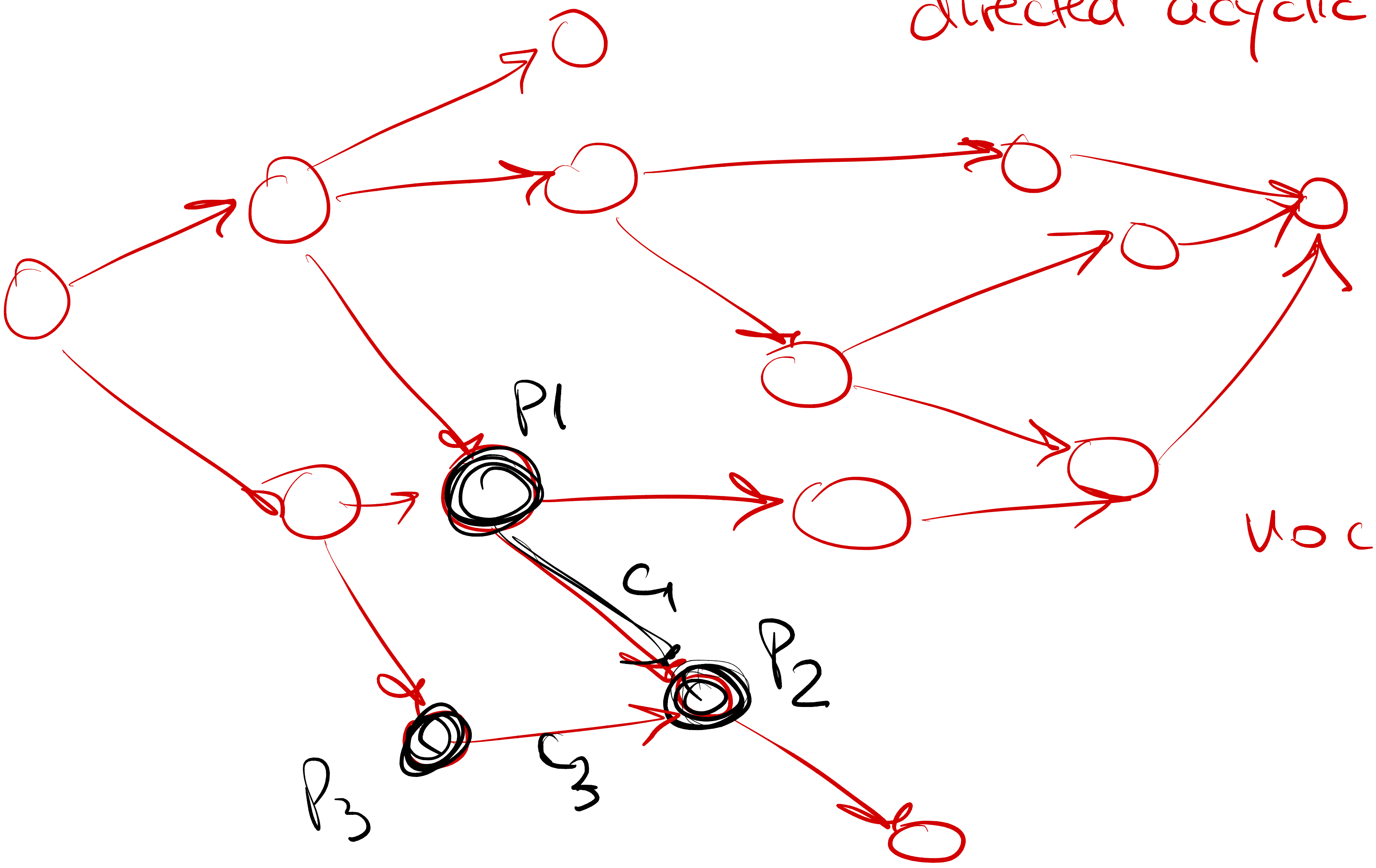
Values = a_i ???

weights = a_i

** Same pb $|B| = |C| = n/2$ n=even

DP = Shortest path in a DAG

directed acyclic graph

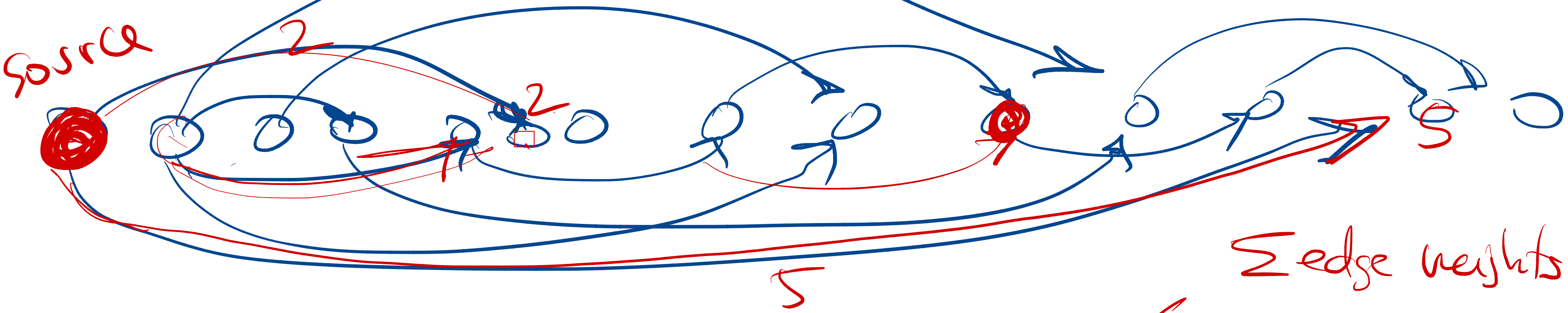


no cycles

DP table

c= edge weight = add cost of from P_1 to P_2

① DAG ⟹ Flattened on a line (topolgical sort)



Σ edge weights

② topological sorted DAG ⟹ SP (most left to all other nodes)

$\Theta(V+E)$

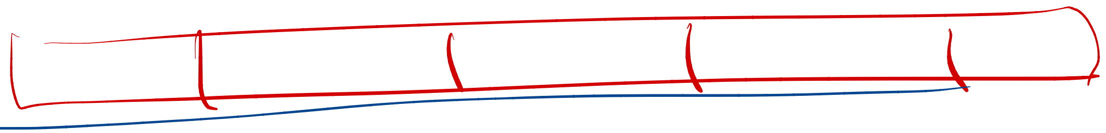
in practice $\Theta(E)$

Datastructures Recap

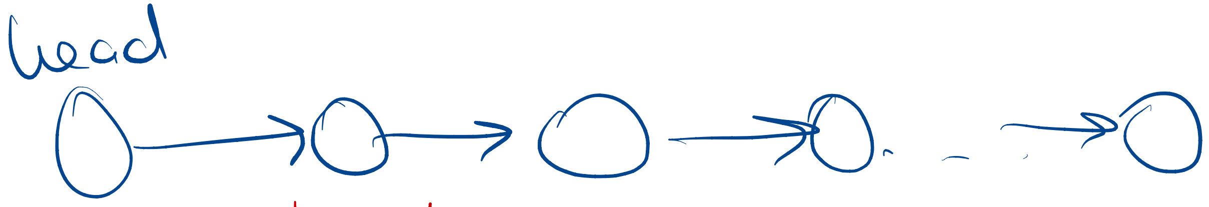
- Arrays

direct Access A[k]

continuous chunk in memory



- Linked Lists



- various locations in memory

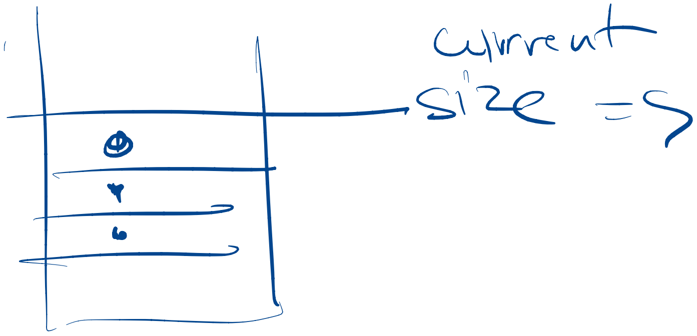
pointers in between

- requires traversal.

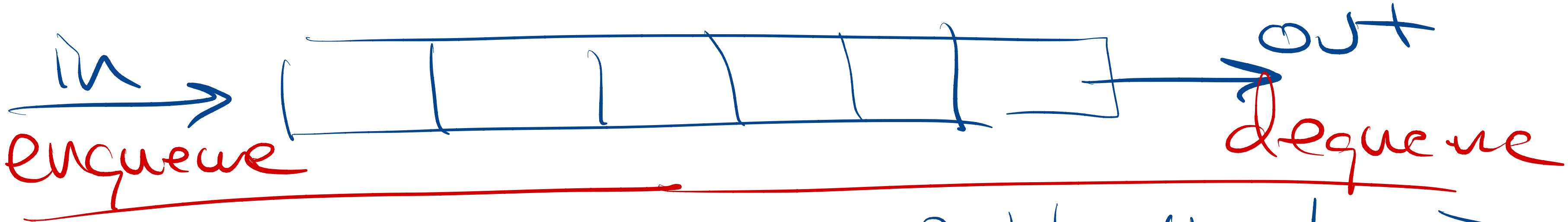
- Stacks FILO

Push(a): put a on stack

Pop(k): take out top k elements (or S if S≤k)



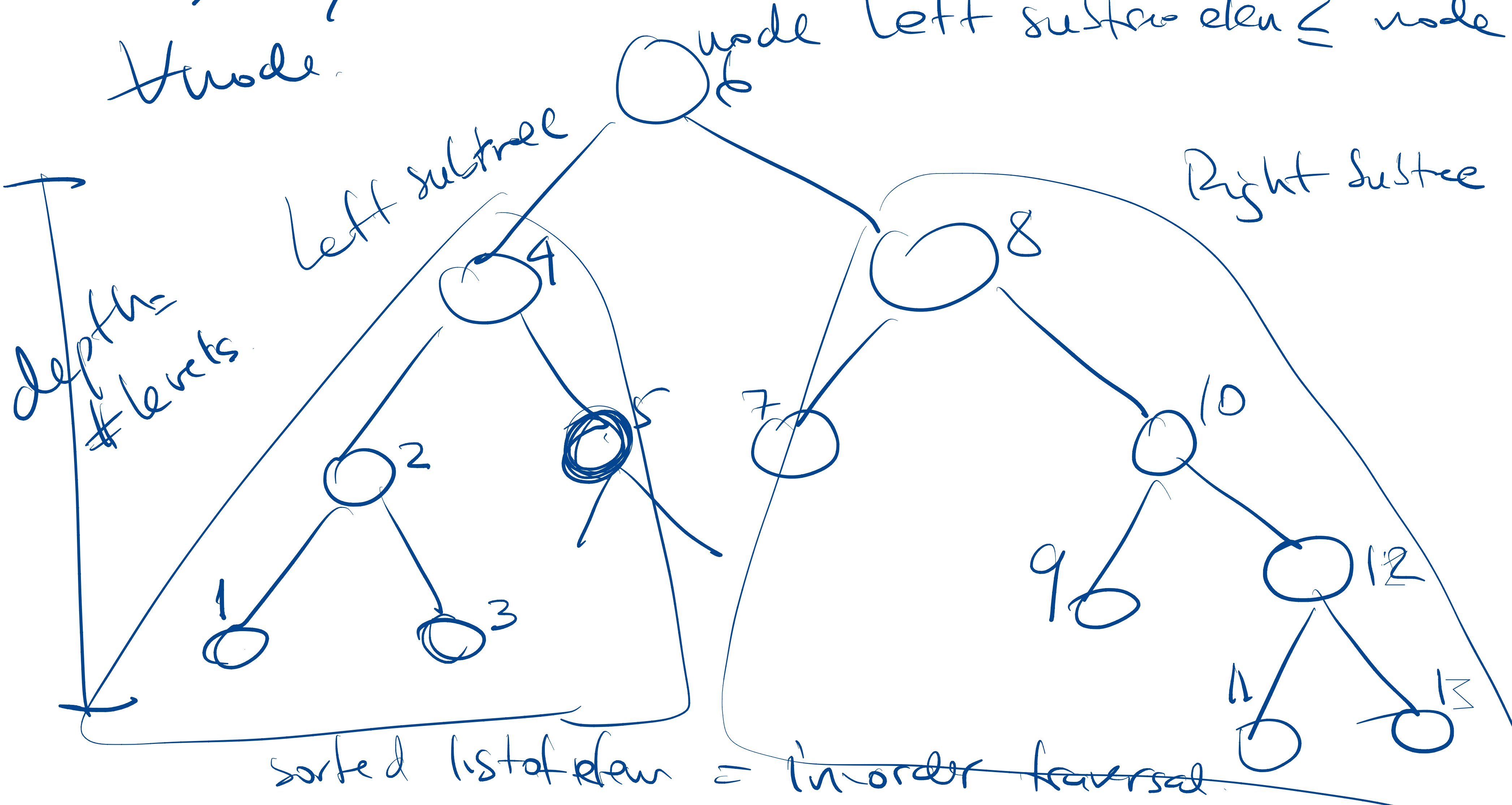
- **Queue.** FIFO



- Binary search trees.

Right subtree elem ≥ node
Left subtree elem ≤ node

∀node.



sorted list of elem = inorder traversal

(first module after the midterm)

Datastructures 1

Hash Tables

Red Black Trees

Week 8 Objectives

- Hash Tables, Hashing functions
- Red-Black Trees

Arrays VS Hash Tables

- typical computer storage is (key,value) pair
- arrays must have keys as integers
  - keys=indices=positions
  - due to how they work in computer's memory
  - have to be continuos
  - Example A[1]=2; A[2]=-1; A[3]=0
- Hash Table also stores (key,value) pairs
  - keys can be anything, like peoples names
  - H[Alice]=1; H[Bob]=-1; H[Charlie]=3
  - keys cannot be used as positions/indices

Basic hashing

Key | value

hash function / map

h(Key) = integer index

DataStruct[h(key)] = value

= array?

- arrays are very nice, but keys have to be integers
  - keys from 0 to N-1
- hashes very useful when keys are not integers
  - names, words, addresses, phone numbers etc
  - even if key=integer (like phone #) they are not the integers we want as indices
- text processing : natural keys are words/n-grams/ phrases
- databases: natural keys can be anything

Range of Indices : [0 — HASHMAX]

Hashing for integer keys

hashf ($\boxed{\text{key rep}}$) = calculation on representation

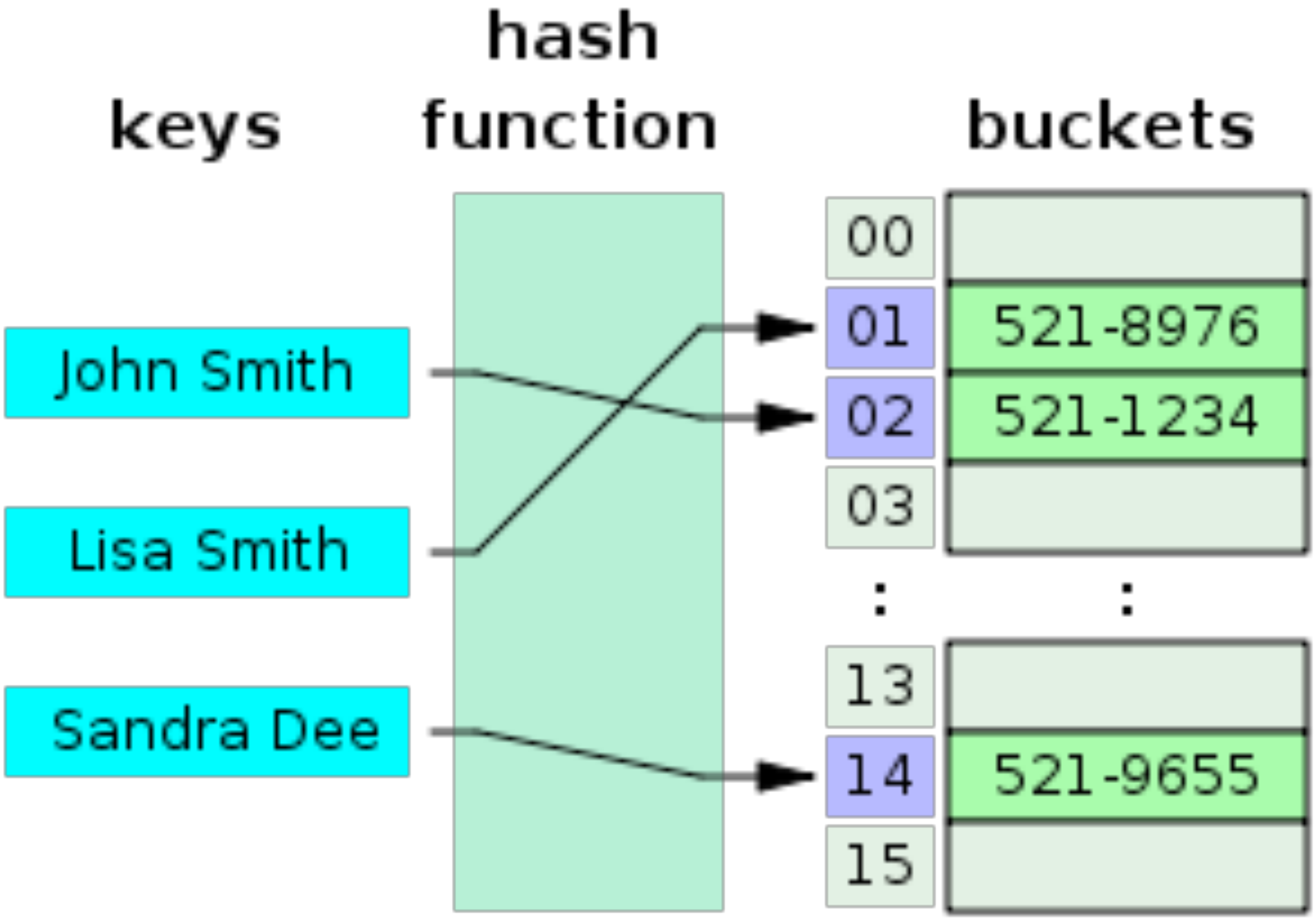
= - - -

= integer

- Even if the keys are integers, they might be inappropriate for storage indices.
- typically the case of few keys in a very large range.
- Example : phone numbers.
  - Might have to use about 10,000 phone numbers as keys
  - if each is used as a index, the resulting array must allocate 9Billion locations (U.S. phone numbers have 10 digits)

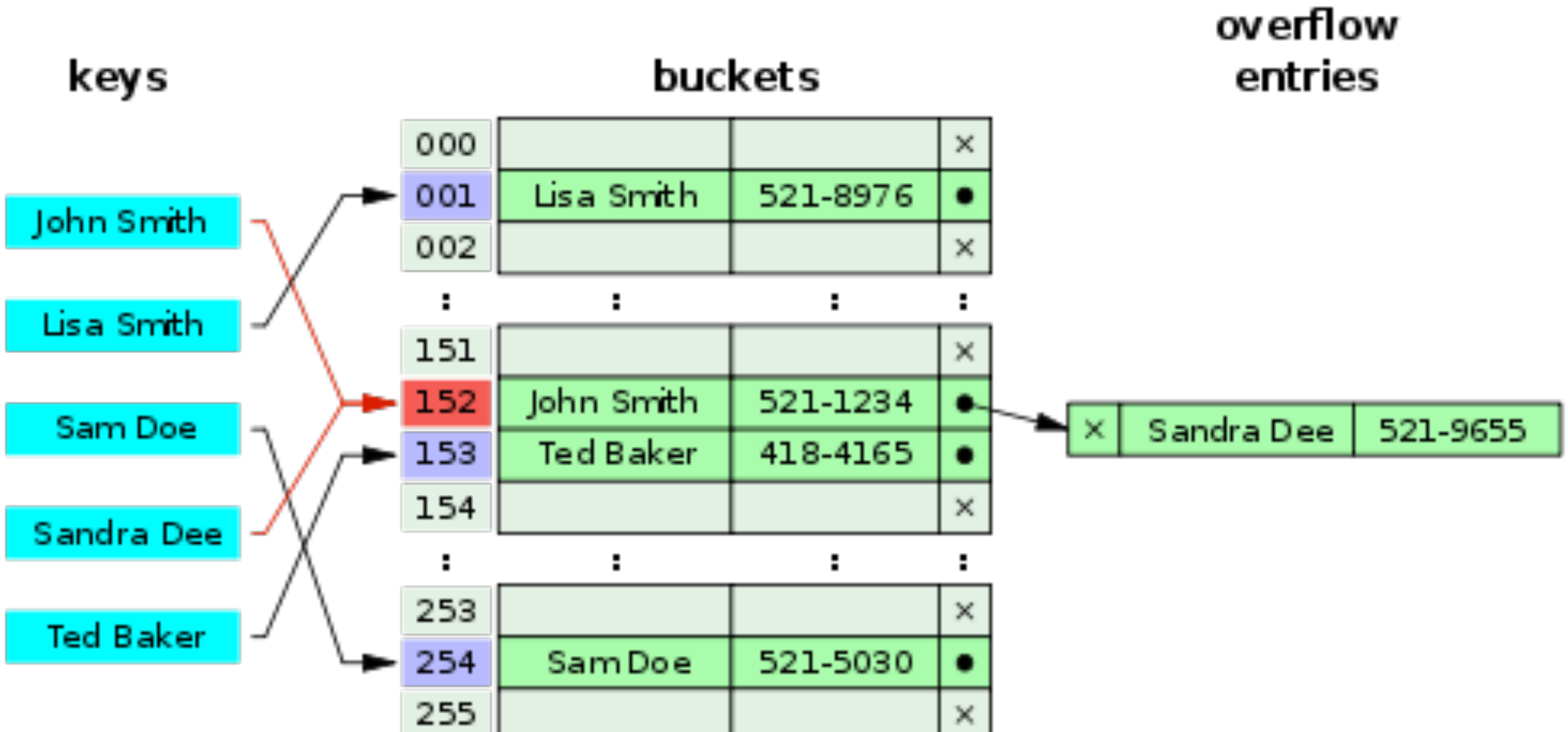
Hash Tables

- key -> index -> use array[index] = value



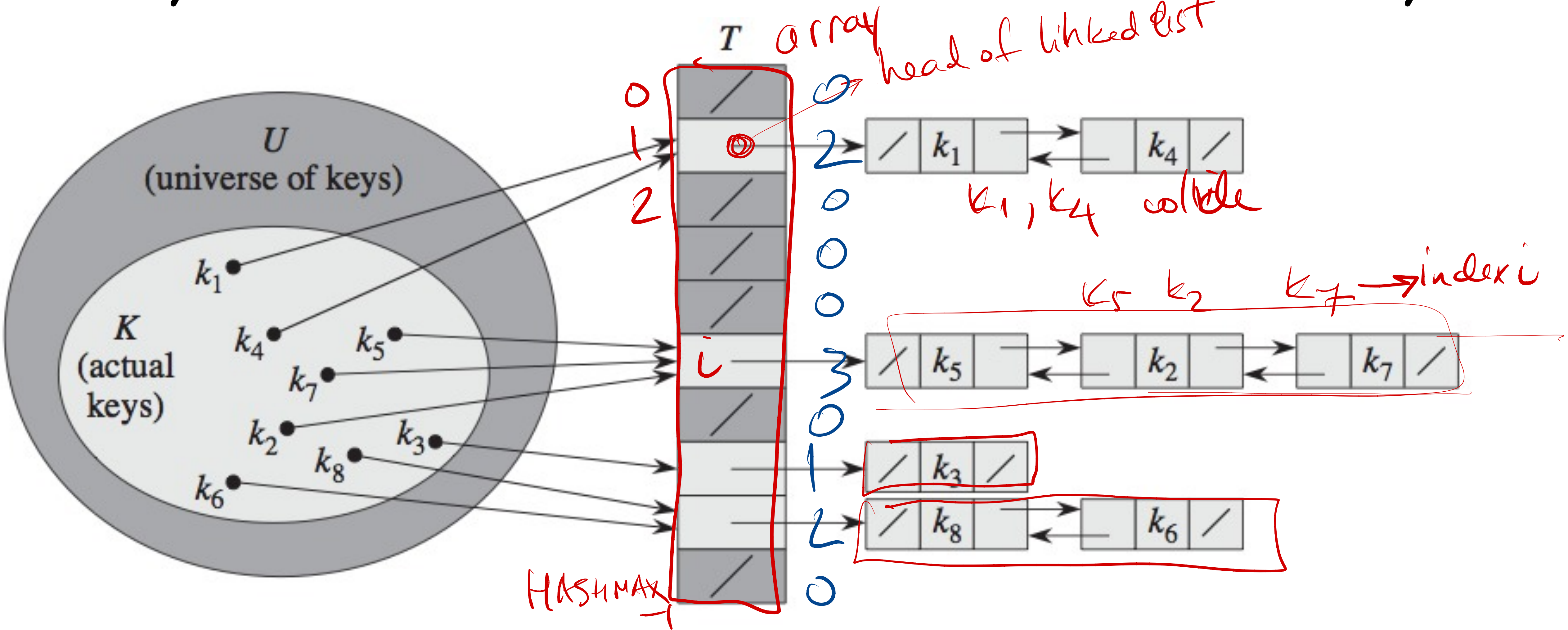
Hash Tables – Collisions

- when several keys (words) map to the same key (index)
- have to store the actual keys in a list
  - list head stored at the index
- key -> index -> list_head -> search for that key



Hash Tables- Collisions with chaining

- when several keys (words) map to the same key (index)
- have to store the actual keys in a list
  - list head stored at the index
- key -> index -> list_head -> search for that key



Hash Tables- Collisions with chaining

each index/pos of head array is linked list of keys collisions

=230000

= 200 fixed in advanced

- n=number of keys; m = MAXHASH; $\alpha = n/m$

balance of collisions

- **simple uniform hashing**: any key k equally likely to be mapped on any of the indices [0...m) hash function randomness
- If collisions are handled with chaining linked lists, assuming simple uniform hashing: $\Theta(\alpha)$
    - unsuccessful search for a key takes $\Theta(1+\alpha)$
    - successful search for a key also takes $\Theta(1+\alpha)$
    - proof in the book

Theorem

$\alpha =$ ratio $\frac{\text{keys}}{\text{spots indices}}$

$\Theta(1+\alpha) = \Theta(\alpha)$ if $\alpha \geq 1$ = Avg #collisions proof idea

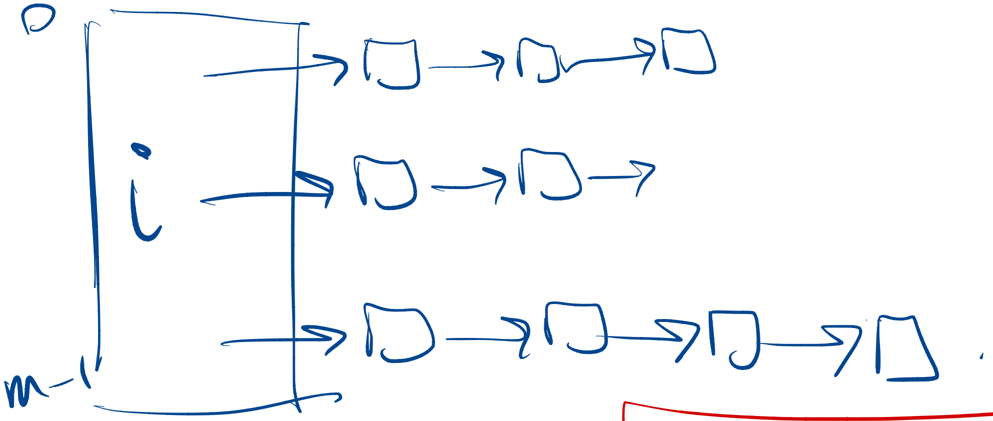
$m = \text{MAX HASH}$

un-successful search.

key K → Hash

⇓

any index is equally likely



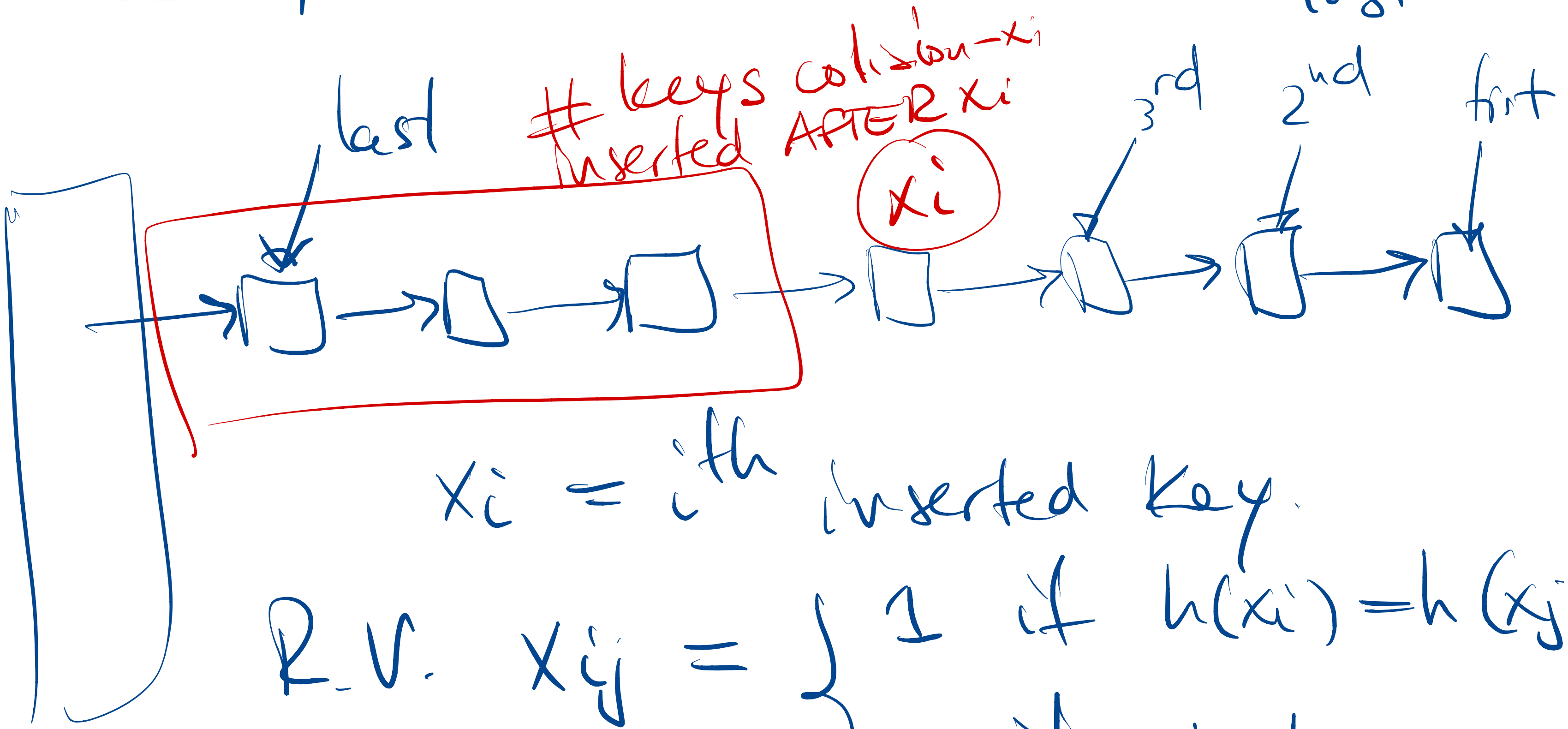
$$\text{prob}\left(h(k) = i\right) = \boxed{\frac{1}{m} = \text{uniform}}$$

$$E[\text{time}] = E[\text{search in } i \text{ list}] =$$

$$= E[\text{size of list}] = \alpha = \frac{n}{m}$$

successful search $k \in Hash$

- new keys (collisions) are added in front of list



x_i = i^{th} inserted key.

$$\text{R.V. } X_{ij} = \begin{cases} 1 & \text{if } h(x_i) = h(x_j) \\ 0 & \text{if not} \end{cases}$$

X_{ij} = 1/0 collision of x_i with x_j

$$\boxed{E[X_{ij}] = \frac{1}{m}}$$

search true

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{\boxed{j=i+1}}^{n} X_{ij}\right)\right]$$

Collisions with x_i

of keys "after" x_i

$$=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} E[X_{ij}]\right)$$

$$=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)$$

$$\alpha=\frac{n}{m}$$

$$1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) = 1+\frac{1}{\cancel{n}m}\frac{(n-1)\cdot\cancel{n}}{2}$$

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

$$=\Theta(1+\alpha)$$

Hash Function

- Easy for humans to use such a hash table
- but not easy for a computer
  - need integer memory locations
  - we have to map keys (names, colors etc) into integers
- hash function h: take input any key, returns an index (int) h(key)=index
- basic operations: INSERT, DELETE, SEARCH; all use the mapped value h(key)

Hash Function

- Usually two stages
  - convert key to a [large] integer (not necessary if keys are already large integers like phone numbers)
  - map the integer in interval [0, MAXHASH)

Simple hash function for words

- return a simple combination of characters, modulo MAXHASH
- int MAXHASH=100000;
- Example hashing word "Virgil" based on ASCII codes

V	i	r	g	i	l
$86*1^2$	$105*\ 2^2$	$114*\ 3^2$	$103*\ 4^2$	$105*\ 5^2$	$108*\ 6^2$

- `int hash_function(char[])` // returns integers between 0 and MAXHASH
  - `int sum=0,i=0;`
  - `while(char[i]>0) {sum+=char[i] * ++i*i;}`
  - `return sum % MAXHASH;`

Hash function: two qualities

- quality ONE: one-to-one (injection). Different inputs result in different outputs
  - collision: having many keys map to same index
- collisions eventually will happen, need to be solved
  - collisions should be balanced (uniformly distributed) per output indices; same as saying simple uniform hashing (approx) is desirable, even if not exact.
- quality TWO: the set of returned indices must be manageable
  - for example returns integers from 1 to 100000
  - or returns integers in range (0, MAXHASH)

Hash Function – division method

- map key to integer k (key=k if key is already integer)
- `h(k) = k mod m` (m=MAXHASH)
  - this equation guarantees that h(k) is one of {0,1,2,..., MAXHASH-1}
- bad choices for m : close to powers of 2
  - $m=2^p$
  - $m=2^p-1$
- good choice for m : prime numbers far away from powers of 2
  - example: m=701

Hash Function - multiplication method

- fractional(x)= fractional part of x, or $x - \lfloor x \rfloor$
  - example fractional(3.1472) = 0.1472
- `h(k)=`$\lfloor$`m* fractional(kA)`$\rfloor$
- typically m is a power of 2
- A is a fractional of form $s/2^w$ where $s<2^w$
  - for example A = 2654435769 / 2^{32}

Hash Function –Universal

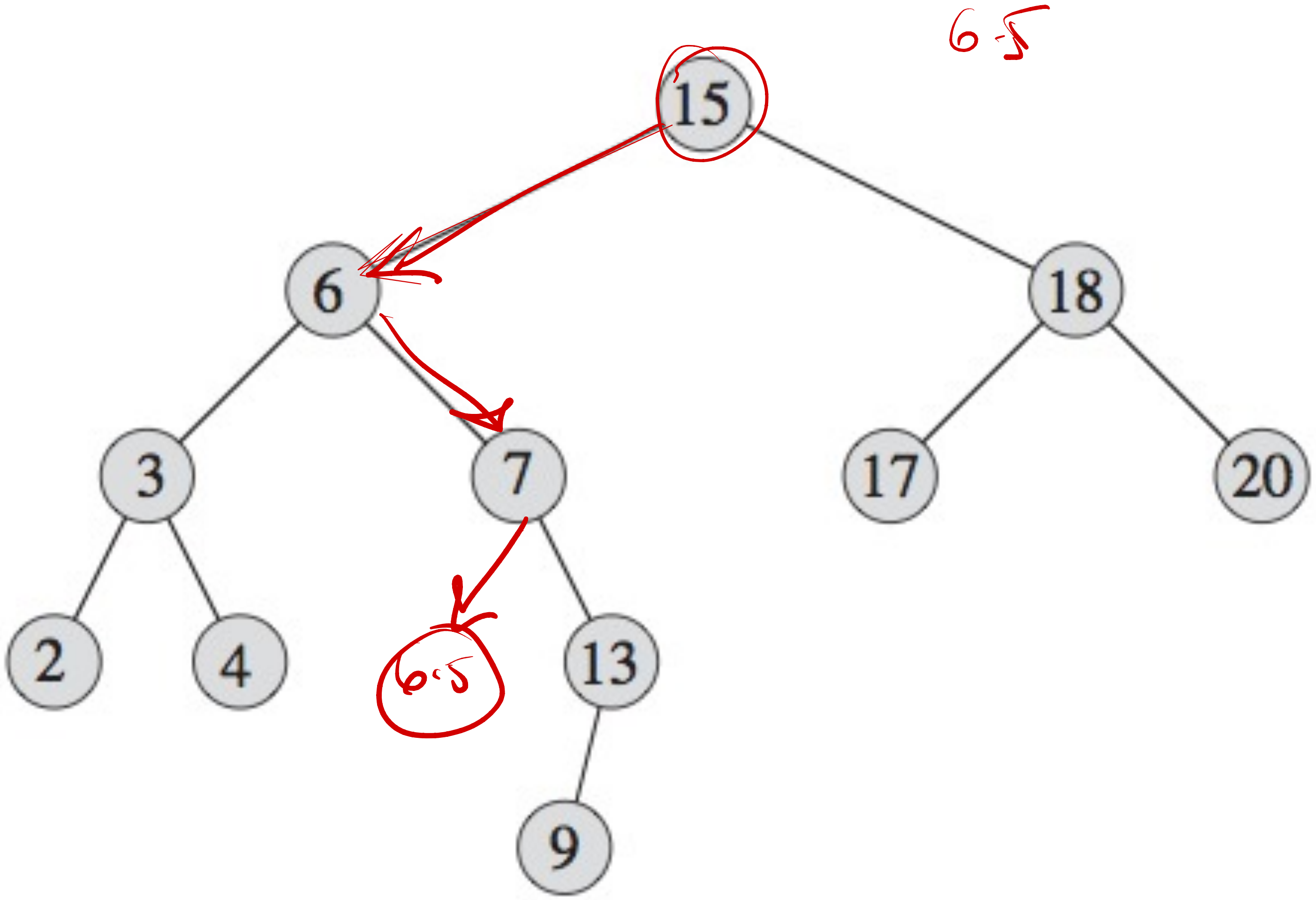
- if the hash function is known, an adversary can attack the hashing schema by using many keys that all collide to the same index
  - h(key1)=h(key2)=h(key3)...
- to prevent this, we can can use set H of hash functions
  - universal set H: for each pair of keys (k,l) the number of hash functions h∈H that collide k and l h(k)=h(l) is no more than |H|/m
  - each time we build a hash (run the code), a random hash function is selected from the set
- building a universal set H of hash functions relies on number theory - see book

Red-Black Trees

further reading necessary from textbook

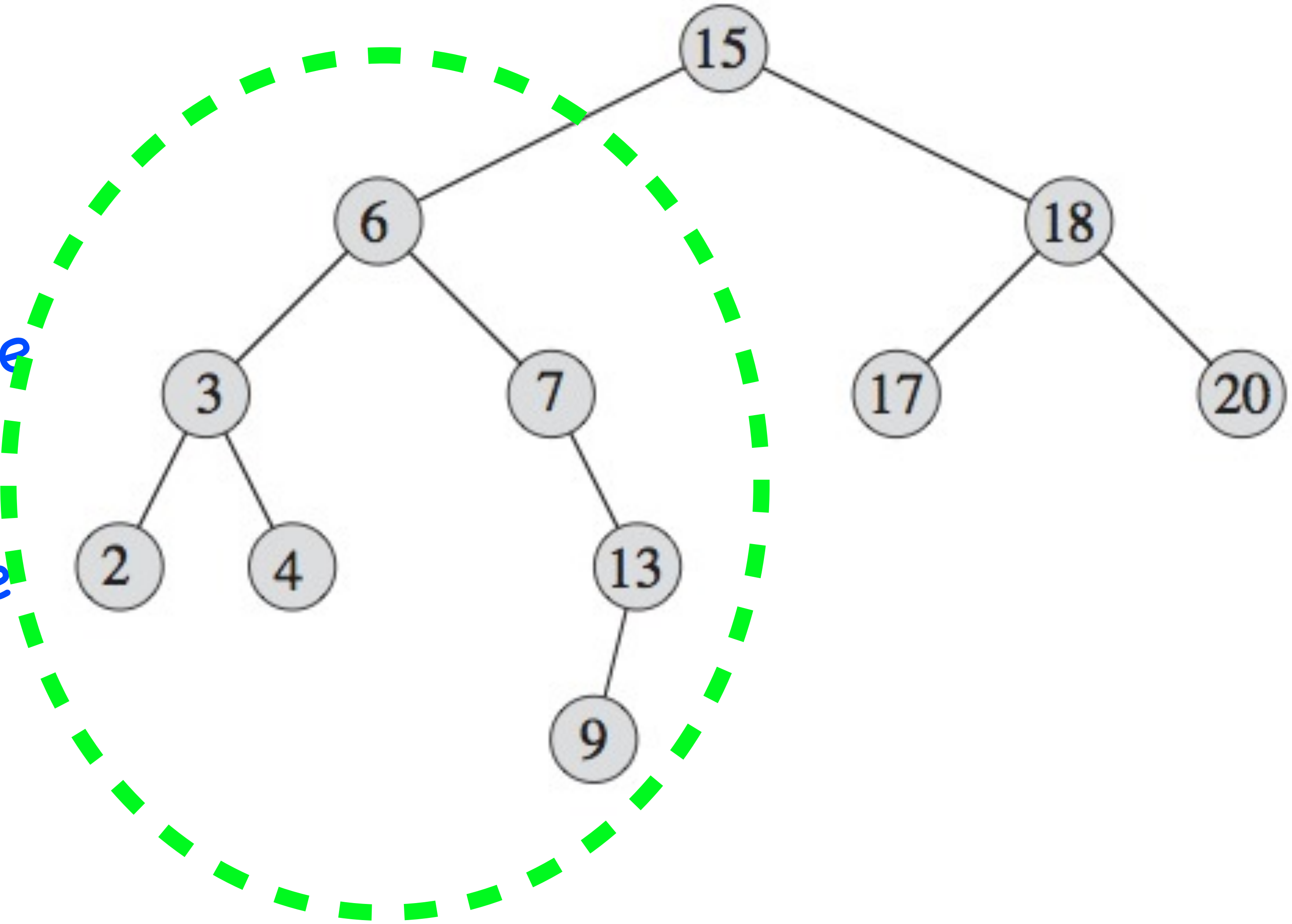
Binary Search Trees - Recap

- 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

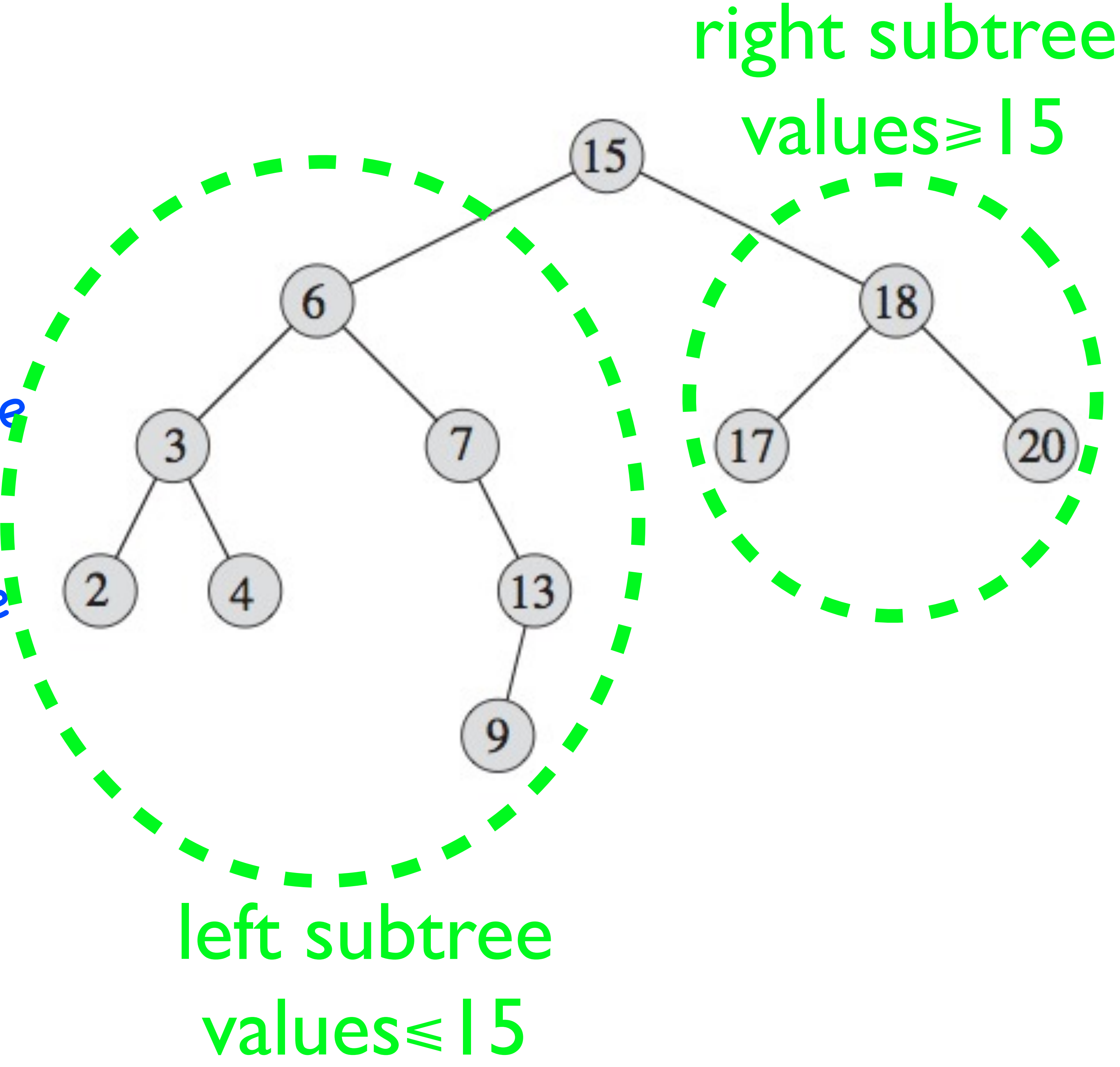
- each node has at most two children
- any node value is
  - not smaller than any value in the left subtree
  - not larger than than any value in the right subtree
  - h = height of tree
- Operations:
  - search, min, max, successor, predecessor, insert, delete
  - runtime O(h)



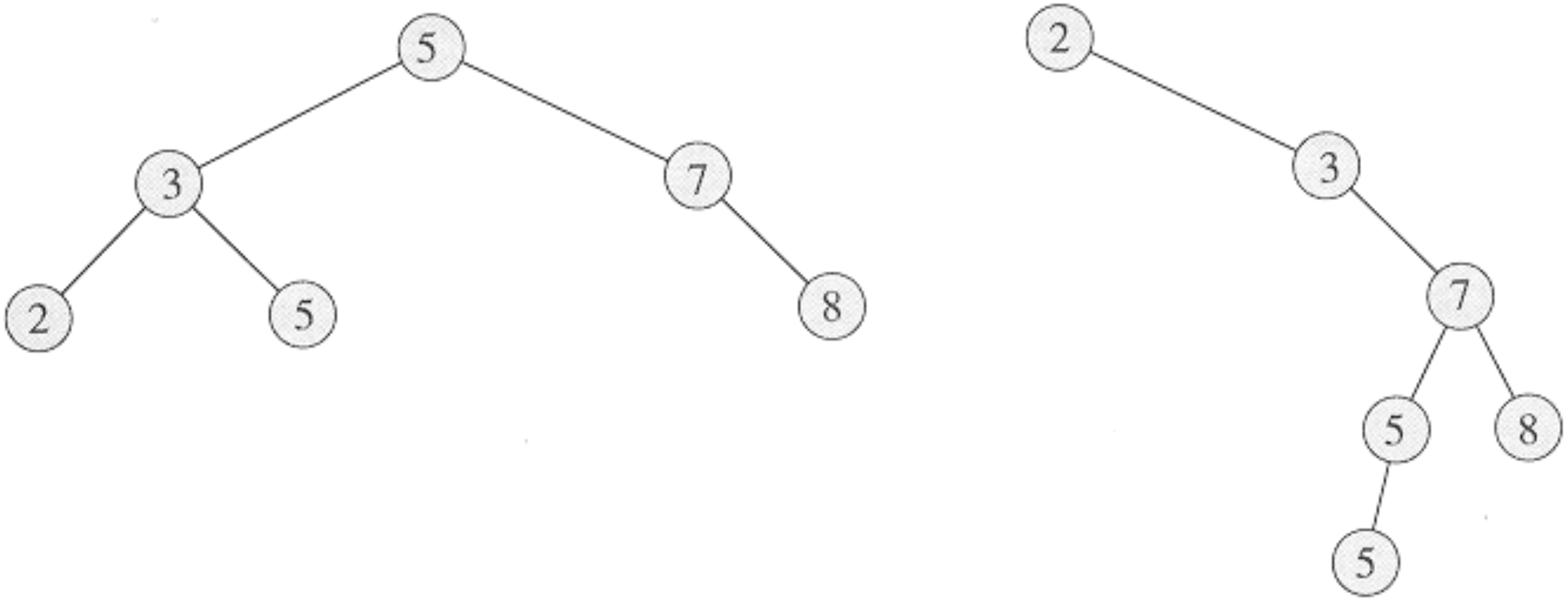
left subtree values≤15

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)



Balanced Trees



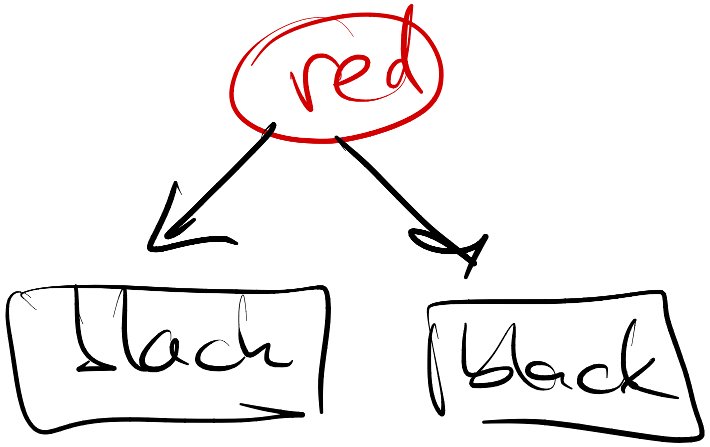
(a)

(b)

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

Red-Black Trees

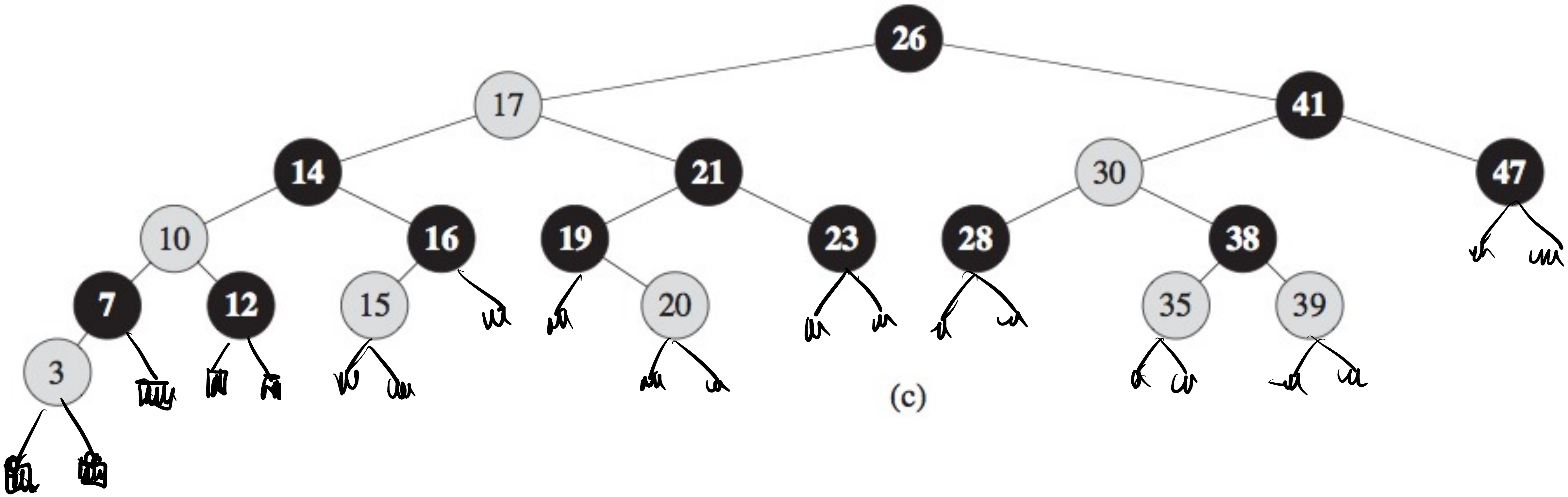
- binary search tree
- want to enforce **balancing** of the tree
  - height logarithmic in n=number of nodes in the tree
  - height = longest path root->leaf
- extra: each node stores a color
  - color can be either red or black
  - color can change during operations



- **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.

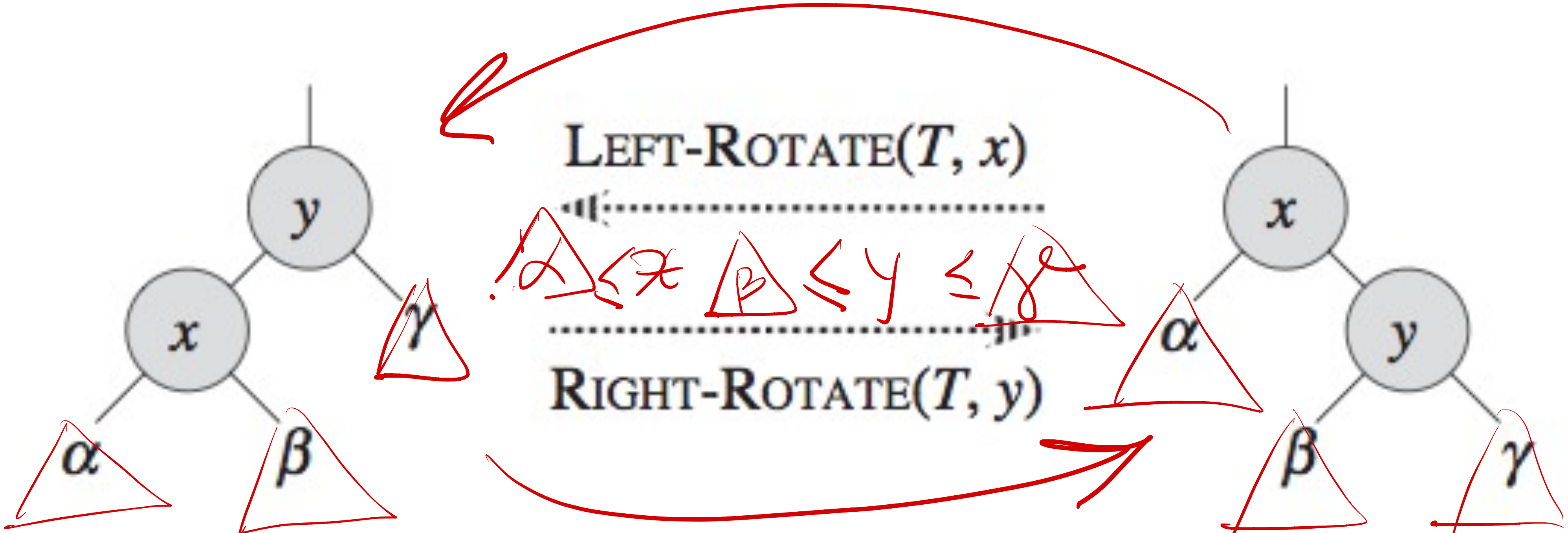
Tree operations

- 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

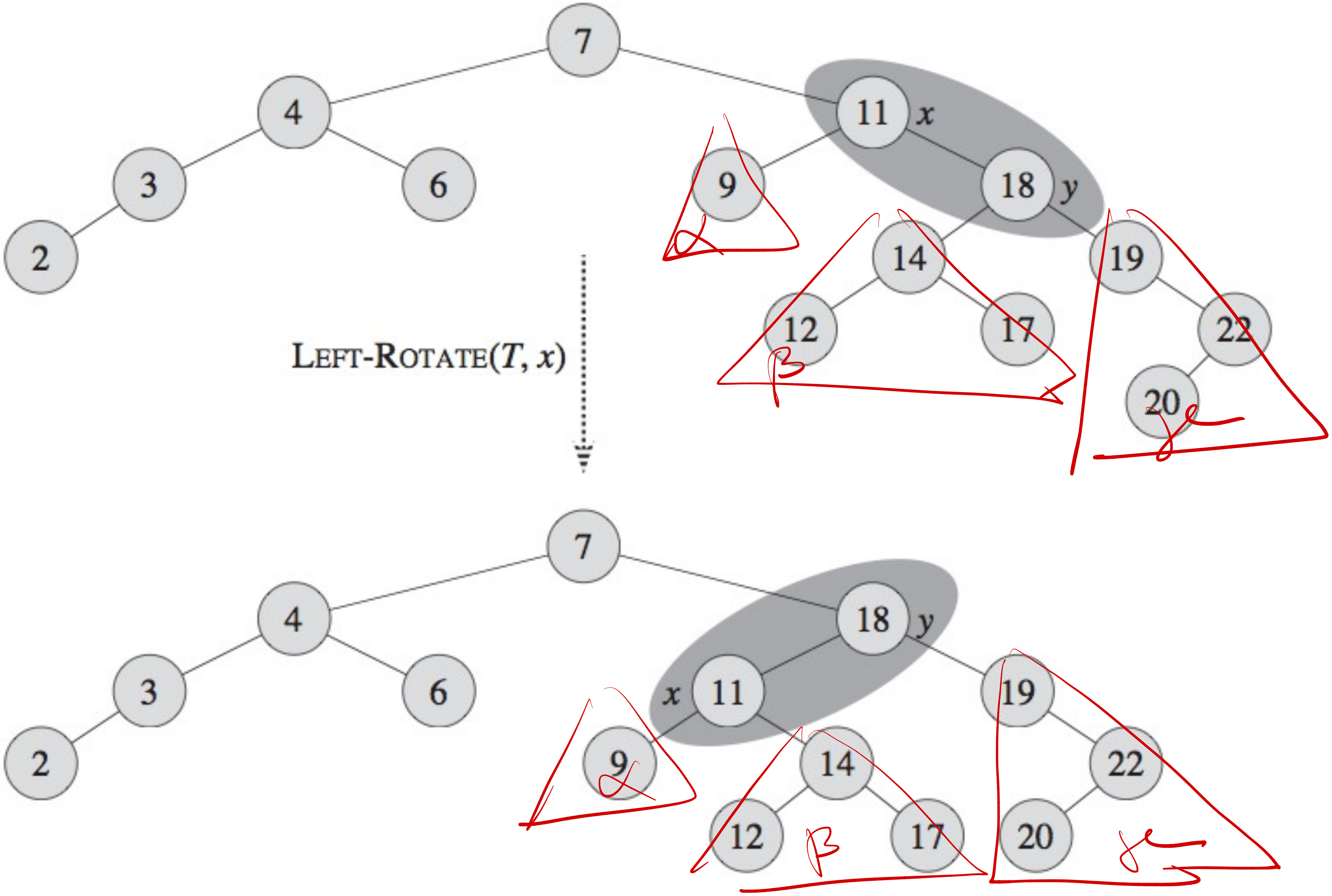
Red-Black Trees - Rotation

- **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

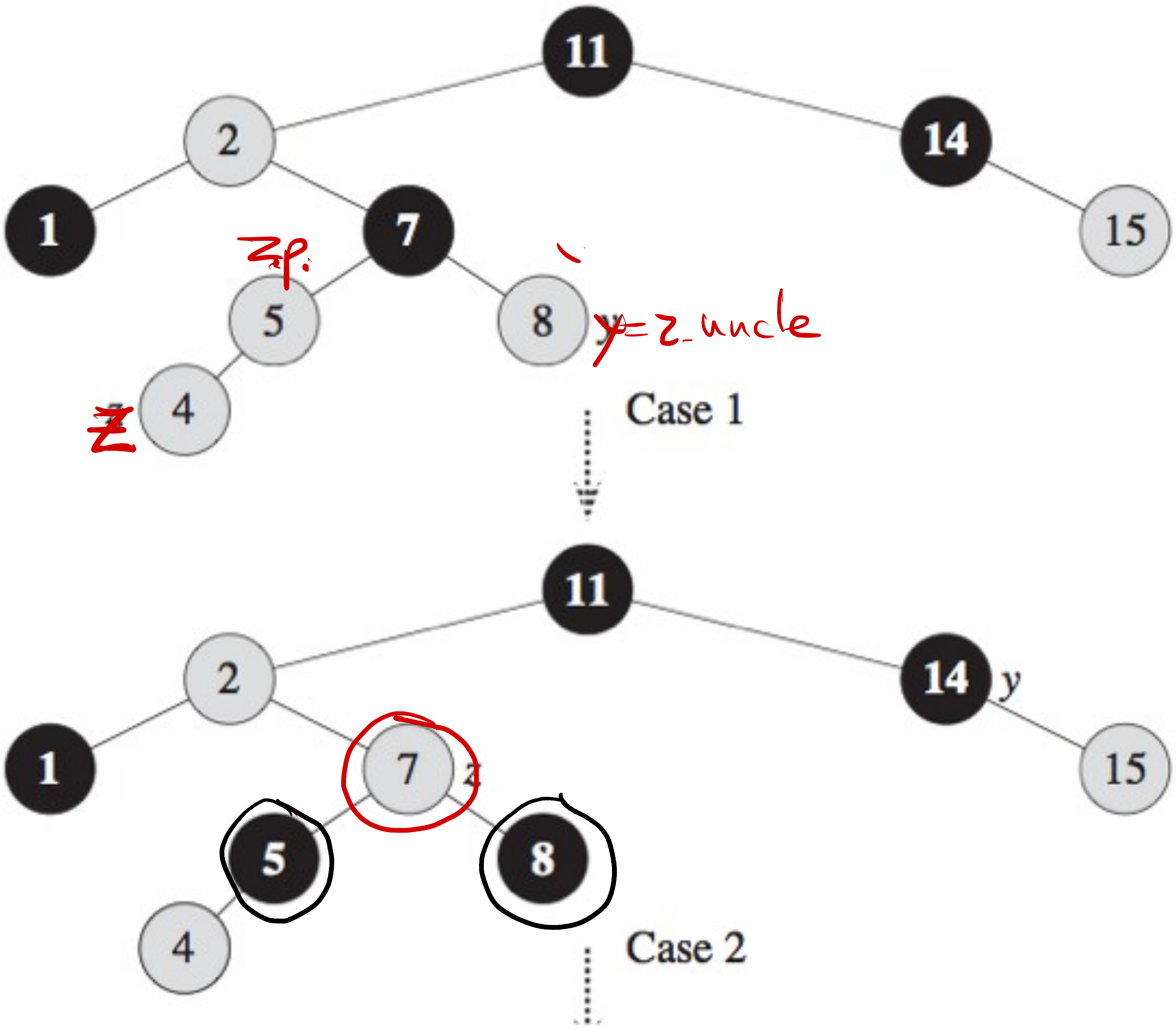
Red-Black Trees - Insertion

- 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

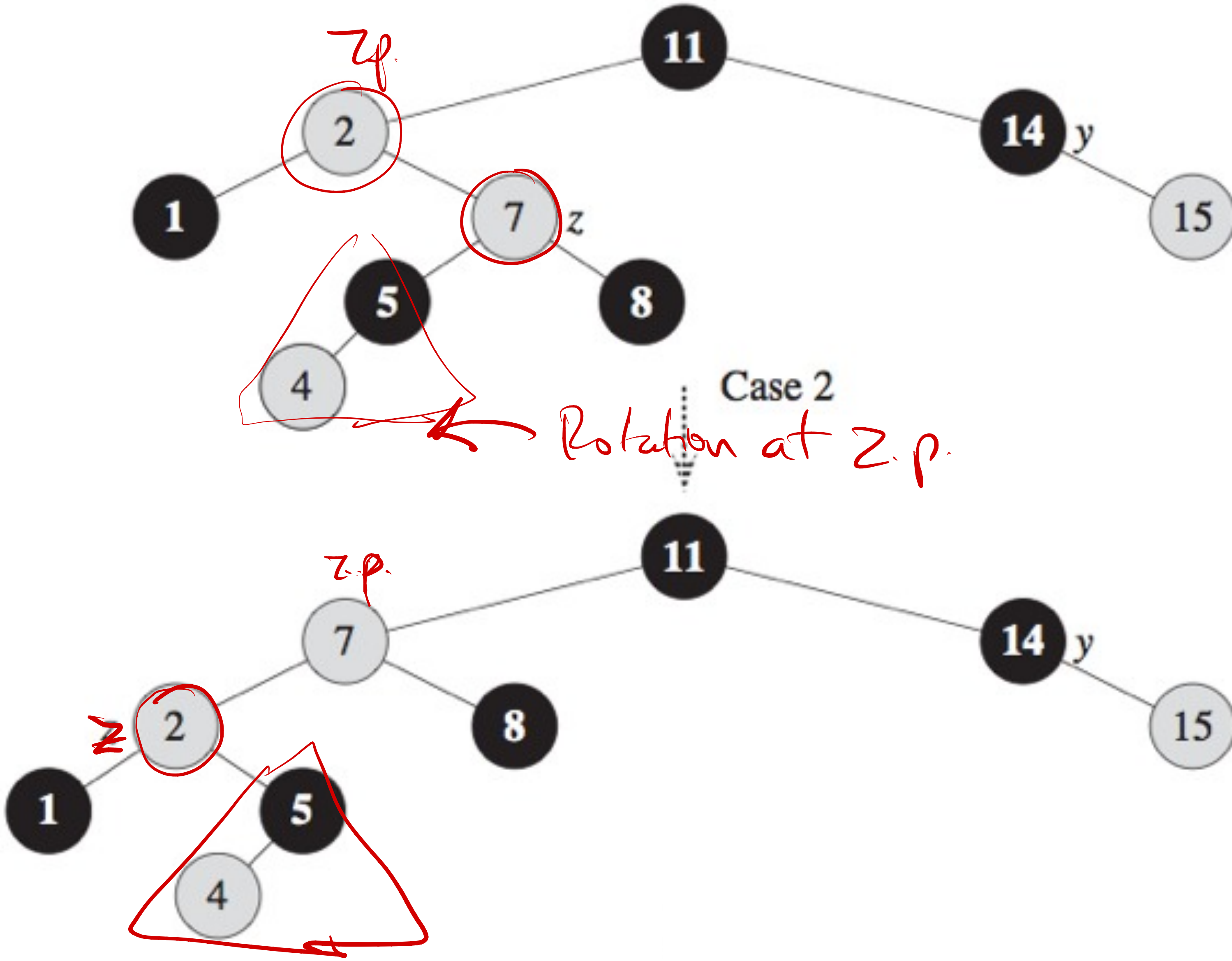
- z.p = z.parent and y=z.uncle are red
- fix:
  - make z.p and y black
  - make z.p.p red
  - advance z to z.p.p

"fixed locally, move up"



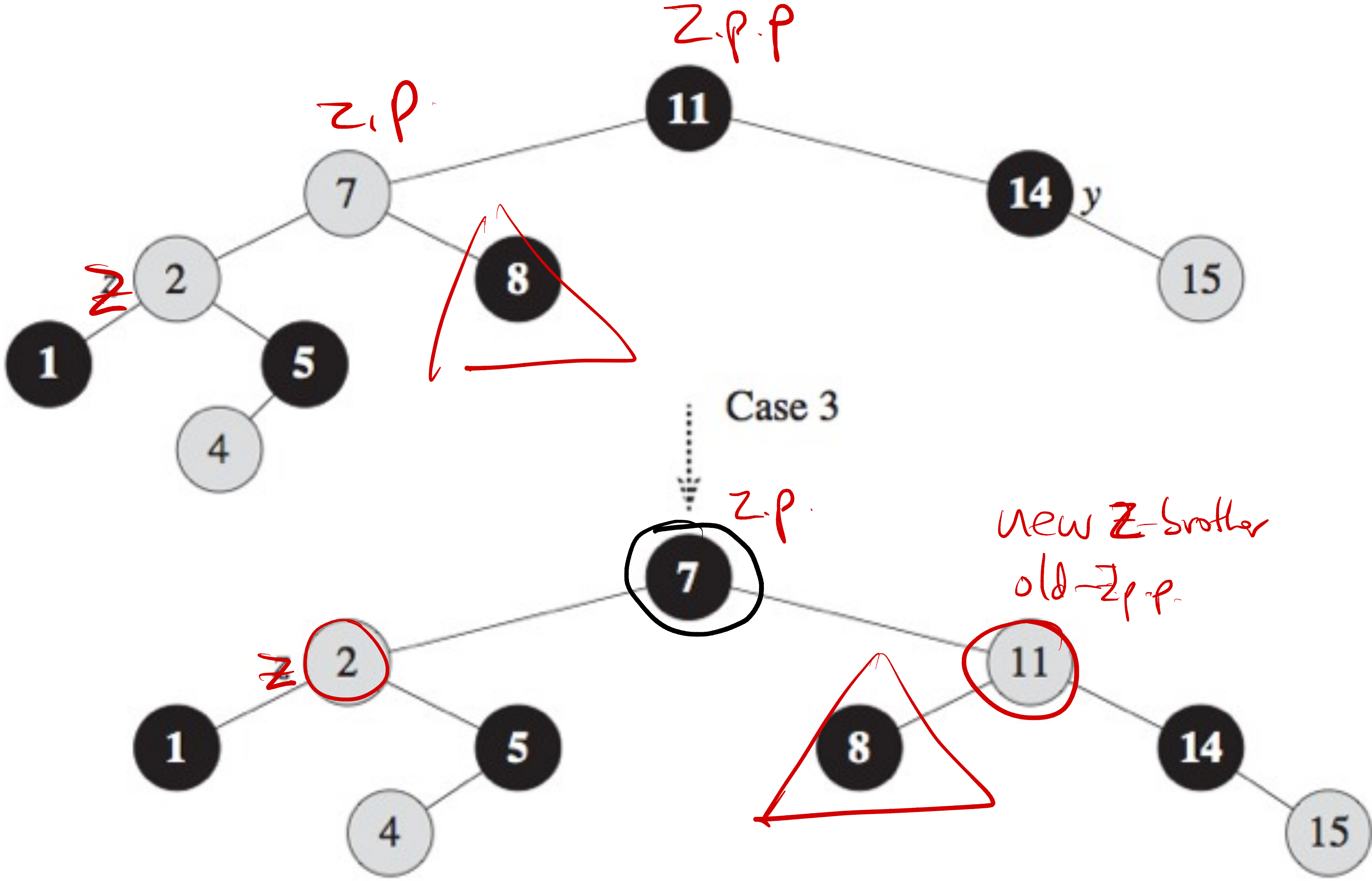
Fixing insertion case 2

- 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

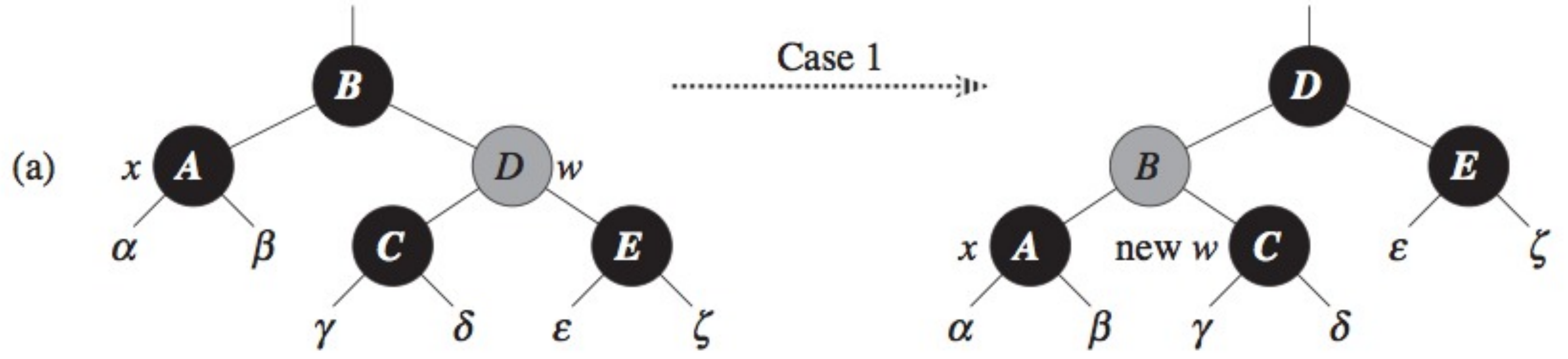
- z.p red, y black, z is left child
- fix:
  - rotate right at z.p.p
  - color z.p black
  - color old z.p.p (now z brother) red



Red-Black Trees - Deletion

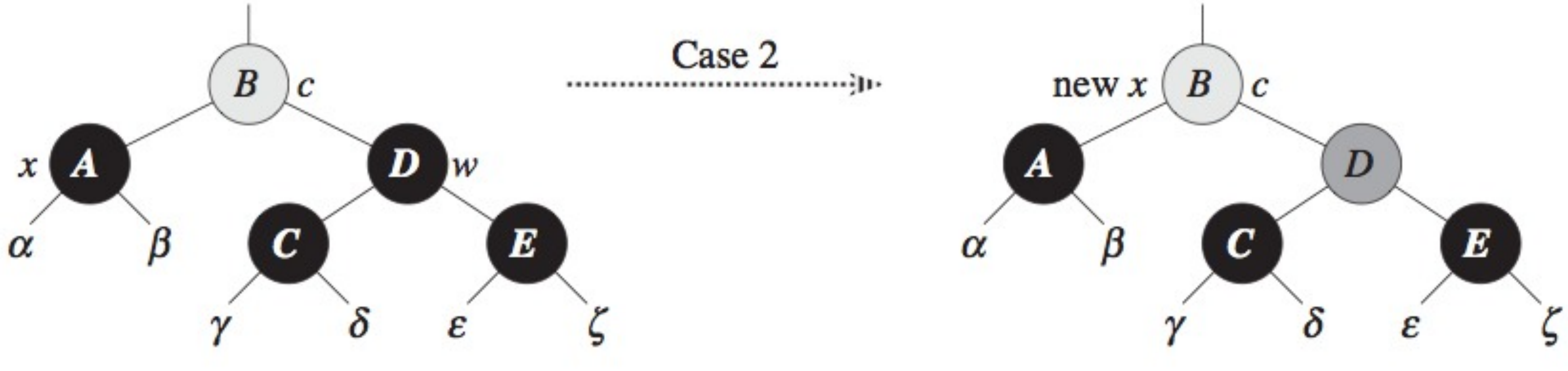
- delete "z" as we usually delete from a binary search tree
  - maintain search property: left values≤ node value ≤ right values
- additionally keep track of
  - y= the node to replace z
  - y original color (its color might change in the process)
- Fix-up the tree red-black properties, if they are violated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

Fixing deletion case 1



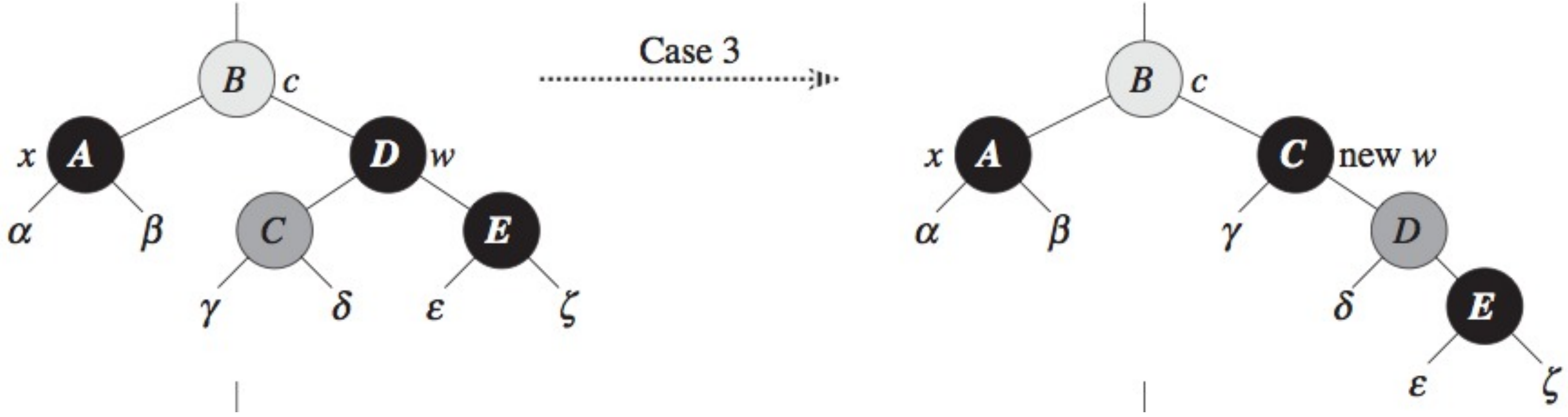
- case 1: x is black, brother w red
- fix :
  - rotate left at x.p;
  - color x.p red;
  - color w (now x.p.p) black

Fixing deletion case 2



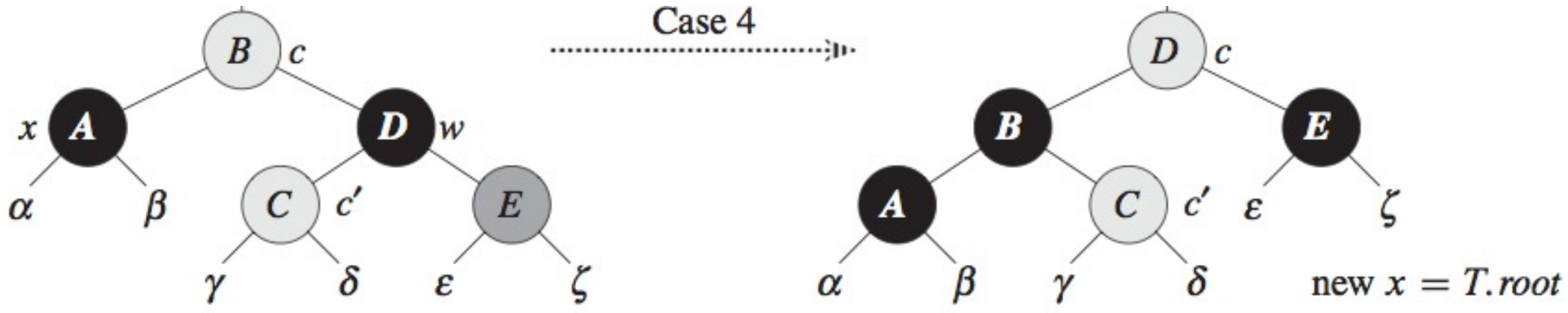
- case2: brother w is black, and w children also black
- fix:
  - color w red
  - advance x to its parent

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)