搜索树 (续)
Search Trees Cont'd

钮鑫涛
Nanjing University
2023 Fall

The slides are mainly adapted from the original ones shared by Chaodong Zheng and Kevin Wayne. Thanks for their supports!

Efficient implementation of Ordered Dictionary

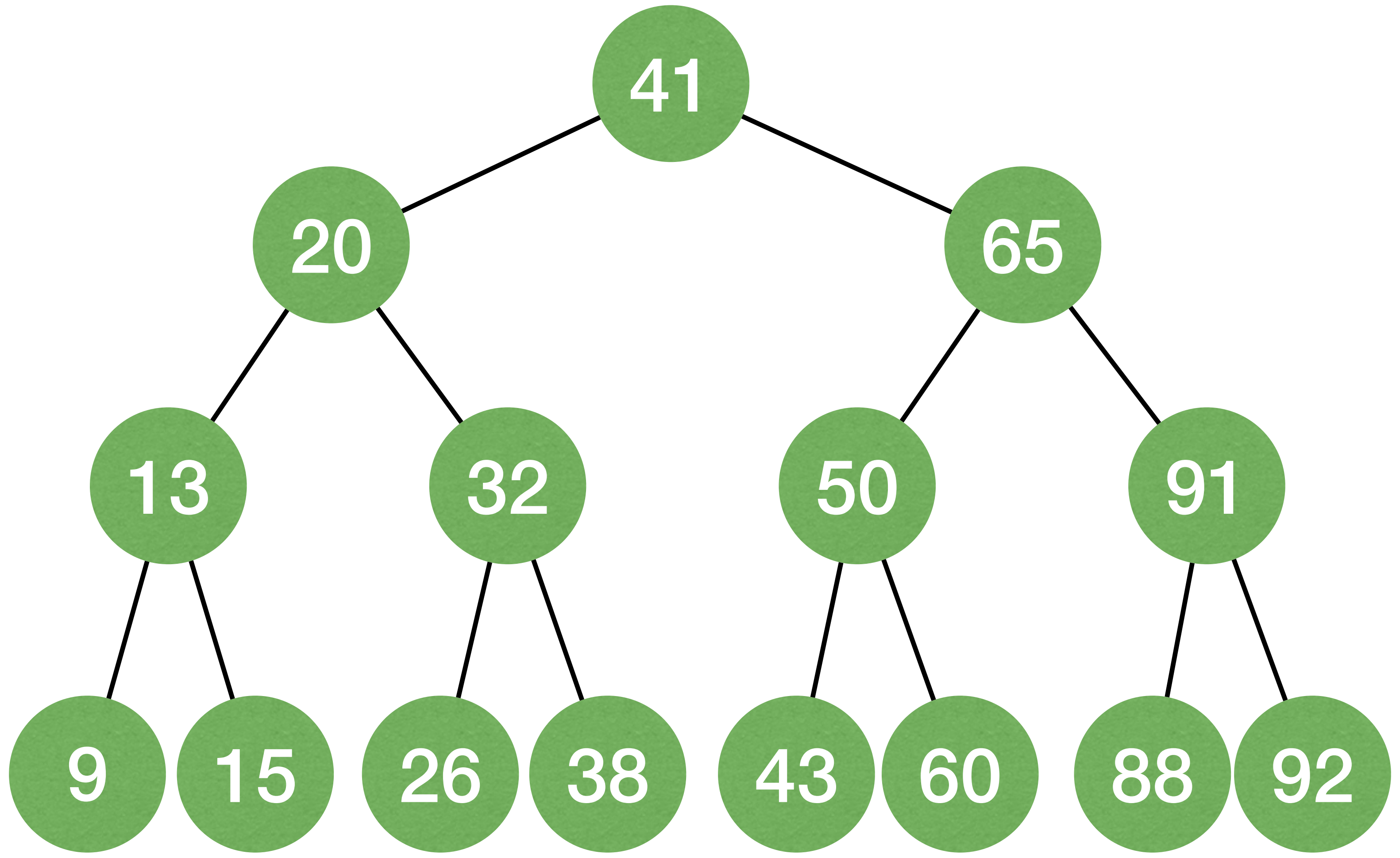
	`Search(S,k)`	`Insert(S,x)`	`Remove(S,x)`
`BinarySearchTree`	$O(h)$ in worst case	$O(h)$ in worst case	$O(h)$ in worst case
`Treap`	$O(\log n)$ in expectation	$O(\log n)$ in expectation	$O(\log n)$ in expectation

Can we have a data structure supporting ordered dictionary operations in $O(\log n)$ time, even in **worst-case**?

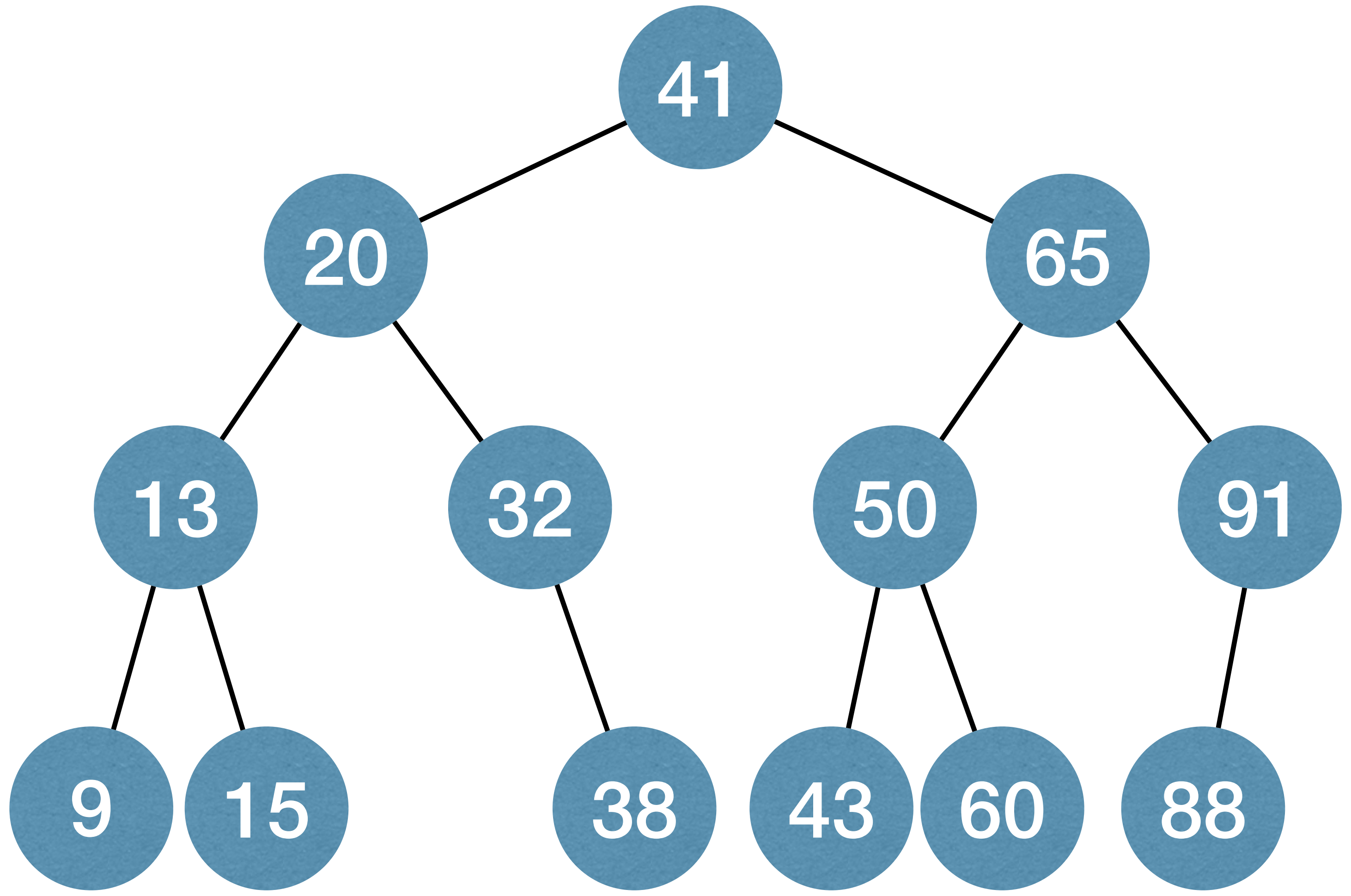
“Balanced” BST

- What does it mean to be “**balanced**”?

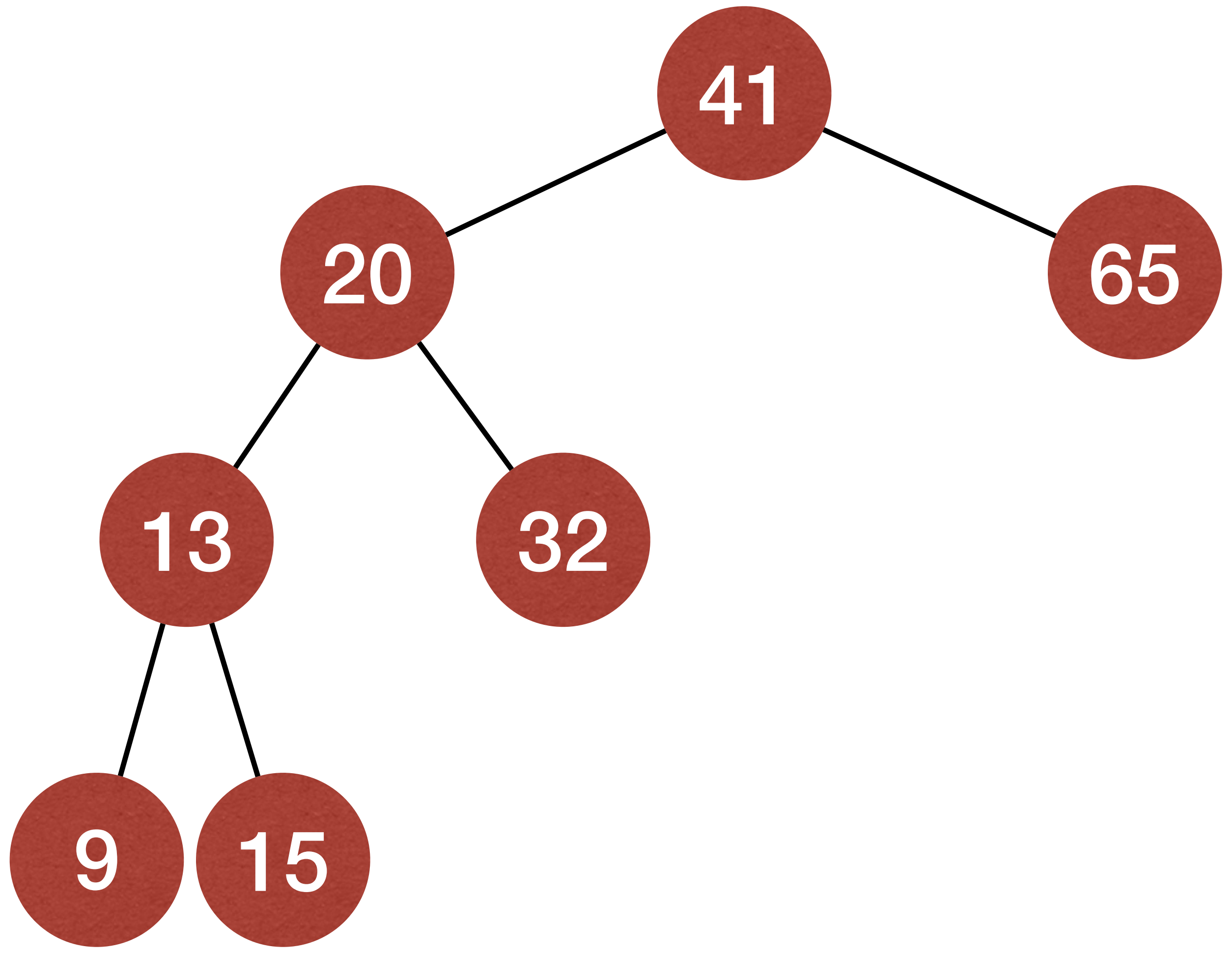
Perfectly Balanced



Almost Perfectly Balanced



Not Perfectly Balanced



An n-node BST is “balanced” if it has height $O(\log n)$.

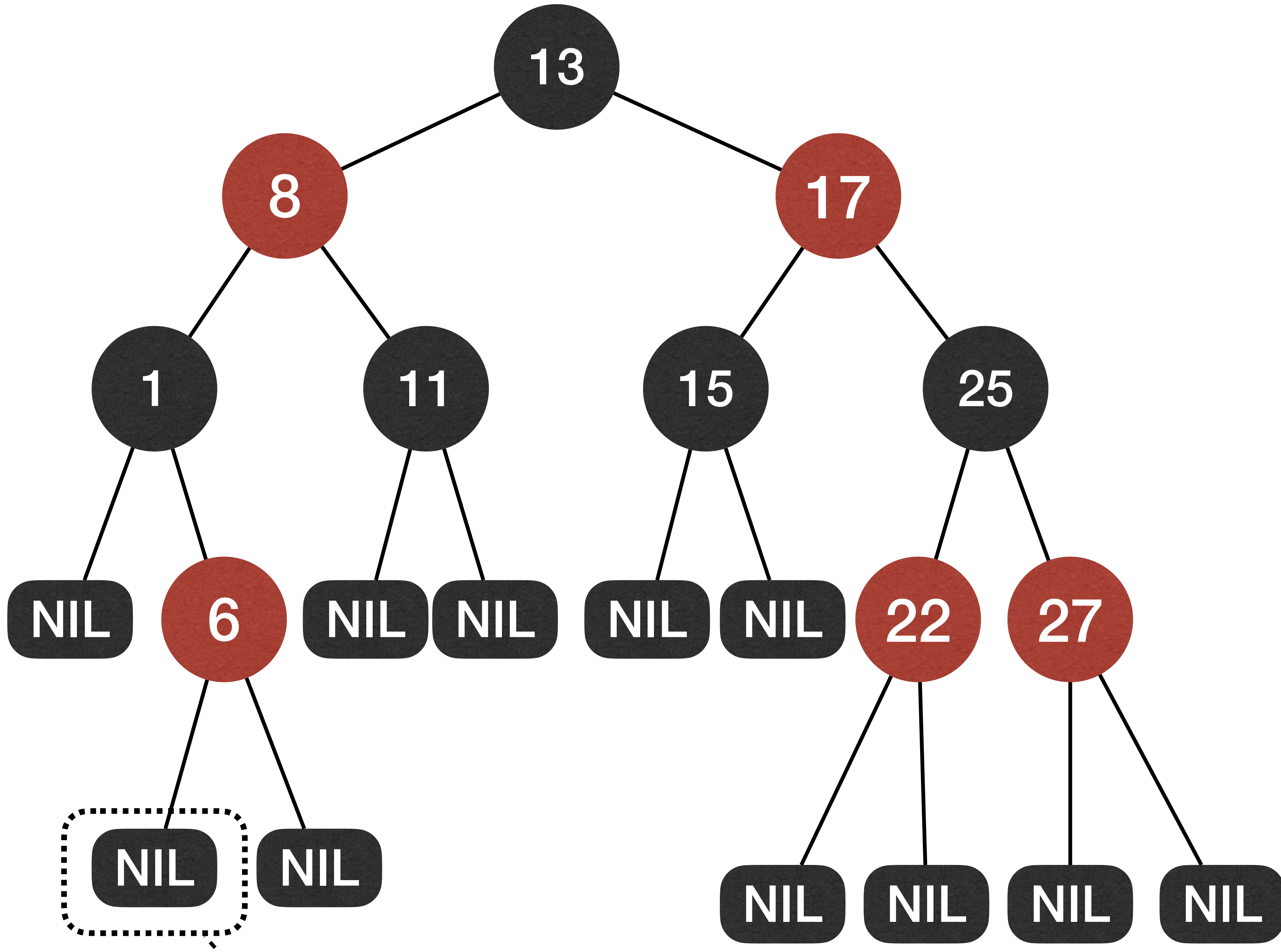
“Balanced” Binary Search Trees

- AVL tree (`Adelson-Velsii & Landis, 1962`)
- B-tree (`Bayer & McCreight, 1970`) — Not binary!
- **Red-black tree** (`Bayer, 1972`)
- Splay tree (`Sleator & Tarjan, 1985`)
- **Skip list** (`Pugh, 1989`)
- **Treap** (`Seidel & Aragon, 1996`)
- and so on …

Red-Black Tree

Red-Black Tree (RB-Tree)

- A **Red**-**Black** Tree (RB-Tree) is a BST in which each node has a color, and satisfies the following properties:
  - Every node is either **red** or **black**.
  - The root is **black**.
  - Every leaf (NIL) is **black**.
  - **[no-red-edge]** If a node is red, then both its children are black.
  - **[black-height]** For every node, all paths from the node to its descendant leaves contain same number of black nodes.

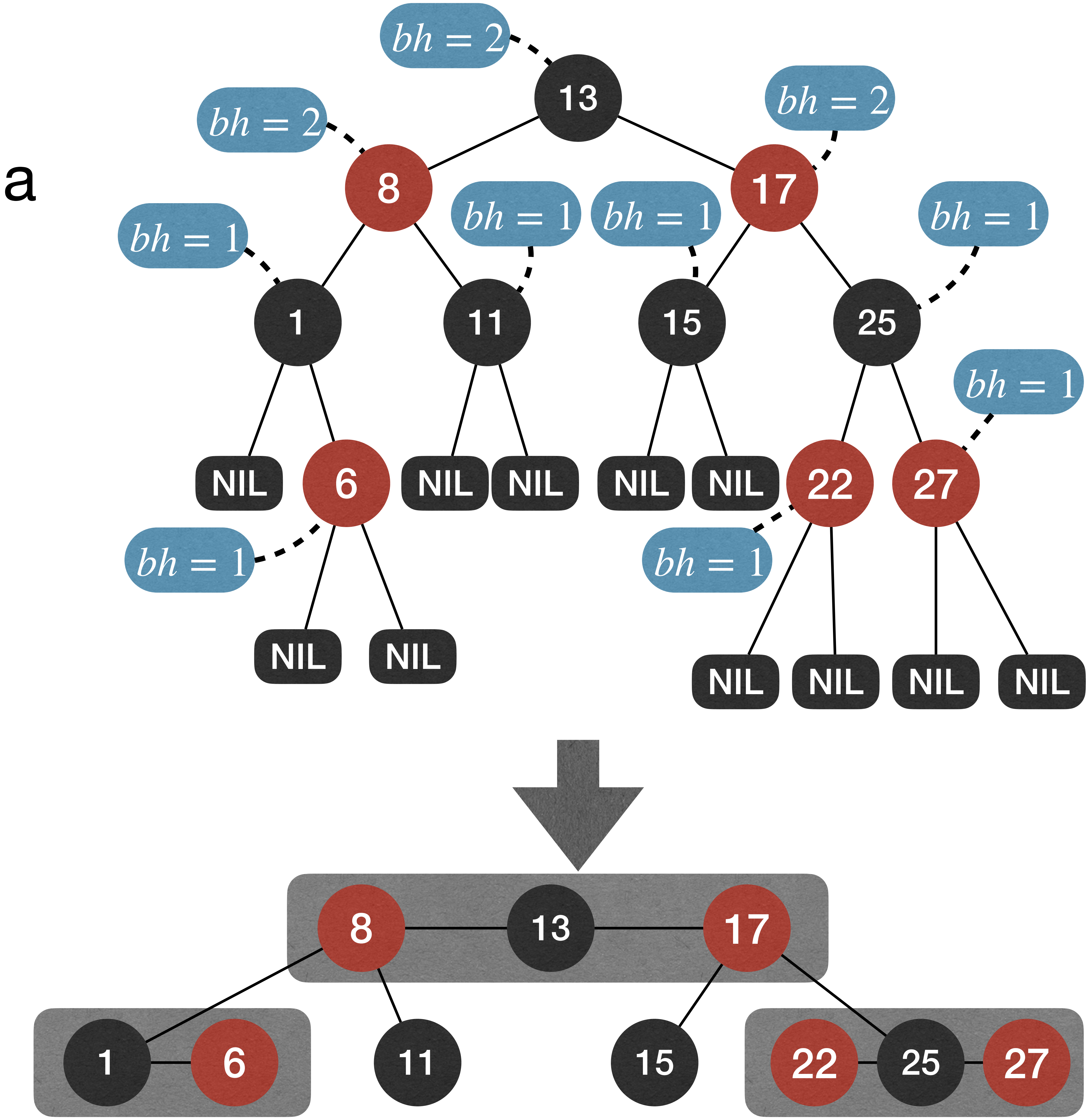


Leaf is sentinel node to represent boundary conditions (can link to the root), and will be omitted later for simplicity.

Black Height

- Call the number of black nodes on any simple path from, but not including, a node x down to a leaf the **black-height** of the node, denoted by $\boldsymbol{bh(x)}$.
  - Due to **black-height property**, from the black-height perspective, RB-Trees are "**perfectly balanced**".
  - Due to **no-red-edge property**, actual height of a RB-Tree does not deviate a lot from its black-height.

RB-Trees are well balanced!



Height of RB-Trees

- **Claim**: In a RB-Tree, the subtree rooted at x contains **at least** $2^{bh(x)} - 1$ internal nodes.
- **Proof** (via induction on height of x)
  - **[Basis]** If x is a leaf, $bh(x) = 0$ and the claim holds.
  - **[Hypothesis]** The claim holds for all nodes with height at most $h - 1$.

Height of RB-Trees

- **Claim**: In a RB-Tree, the subtree rooted at x contains at least $2^{bh(x)} - 1$ internal nodes.
  - **[Inductive Step]** Consider a node x with height $h \geq 1$. It must have two children. So the number of internal nodes rooted at x is:

WHY?

$$\geq 1 + (2^{bh(x.left)} - 1) + (2^{bh(x.right)} - 1)$$

$$\geq 1 + (2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1)$$

$$= 2^{bh(x)} - 1$$

Height of RB-Trees

- **Claim**: In a RB-Tree, the subtree rooted at x contains at least $2^{bh(x)} - 1$ internal nodes. ✓

- Due to **no-red-edge**: $h = height(\text{root}) \le 2 \cdot bh(\text{root})$
  - $n \ge 2^{bh(\text{root})} - 1 \ge 2^{\frac{h}{2}} - 1$, implying that $h \le 2 \cdot \lg(n+1)$.

Theorem The height of an n-node RB-Tree is $O(\log n)$

Therefore, RB-Trees support `Search`, `Min`, `Max`, `Predecessor`, `Successor` operations in worst-case $O(\log n)$ time! But, what about `Insert` and `Remove`?

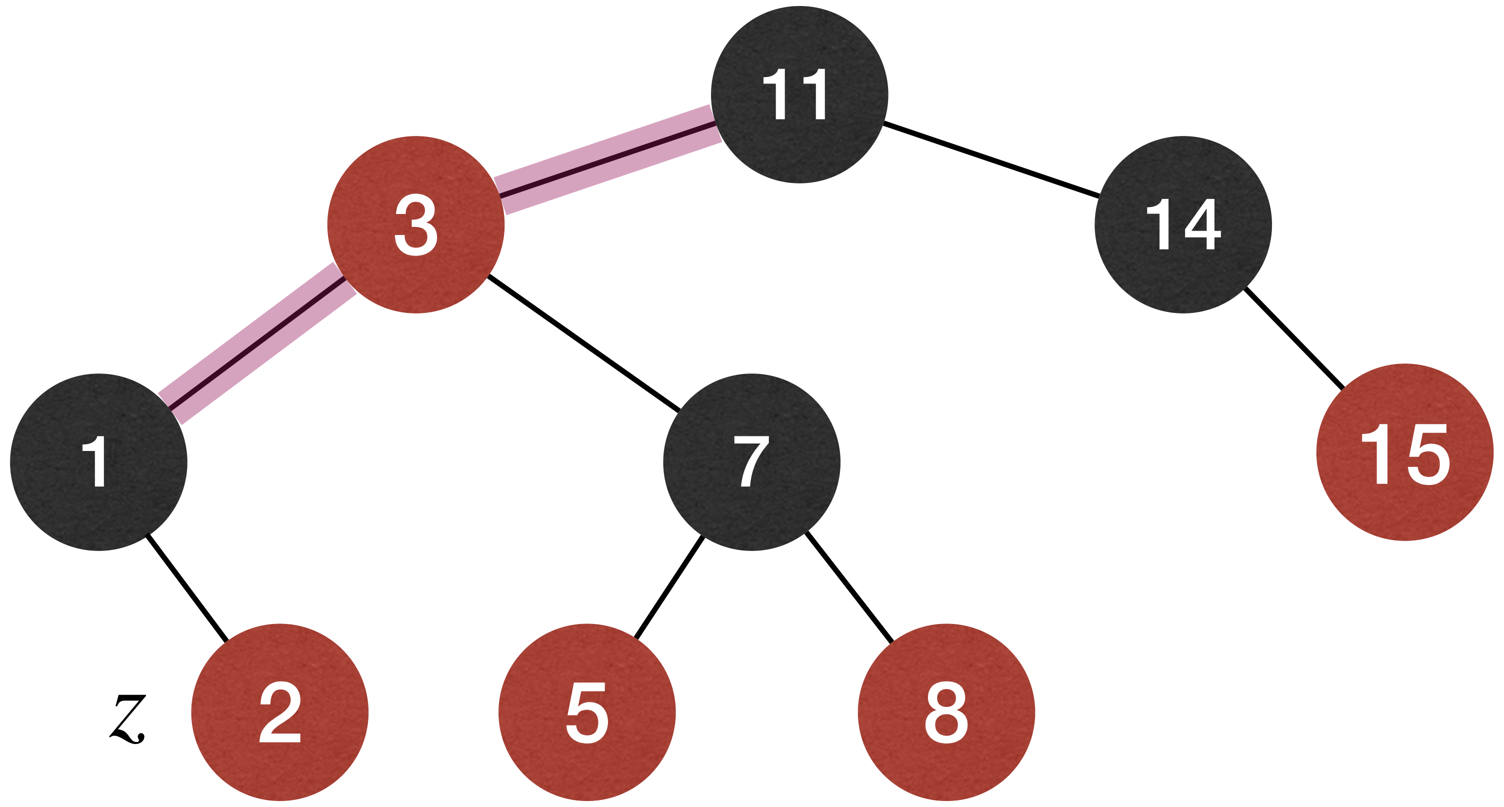
Insert node into an RB-Tree

Maintain black-height, fix no-red-edge if necessary.

- Step 1: Color z as red and insert as if the RB-tree were a BST.
- Step 2: Fix any violated properties.
  - No fix is needed if z has a **black** parent after insertion.

Few red nodes

Example: Insert element with key 2

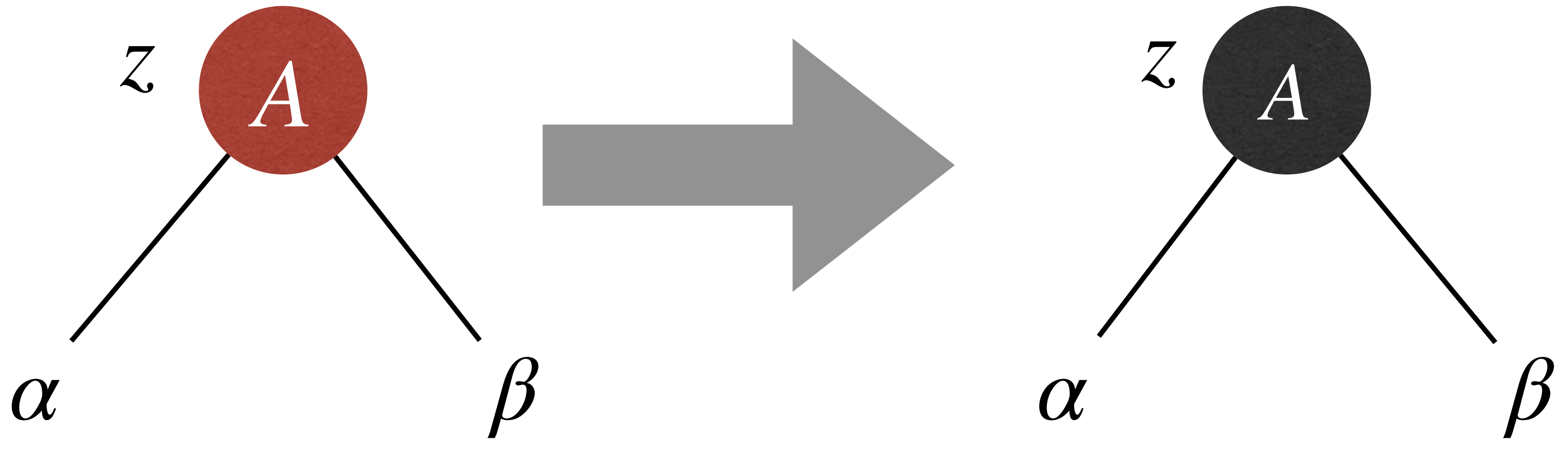


RB-Tree Properties

- ✓ Each node is red or black
- ✓ Root is black
- ✓ Leaves are black
- ✓ No-red-edge property
- ✓ Black-height property

Insert node into an RB-Tree

- Step 2: Fix any violated properties.
  - **Case 0**: z becomes the root of the RB-Tree.
  - **Fix**: simply recolor z to be black.



Note: with the execution of algorithm, we change our **focus** of the node: At the beginning, it is the node to be inserted. Later, it is the node that needs to be changed to fix some property ! We refer to the currently focused node as z

RB-Tree Properties

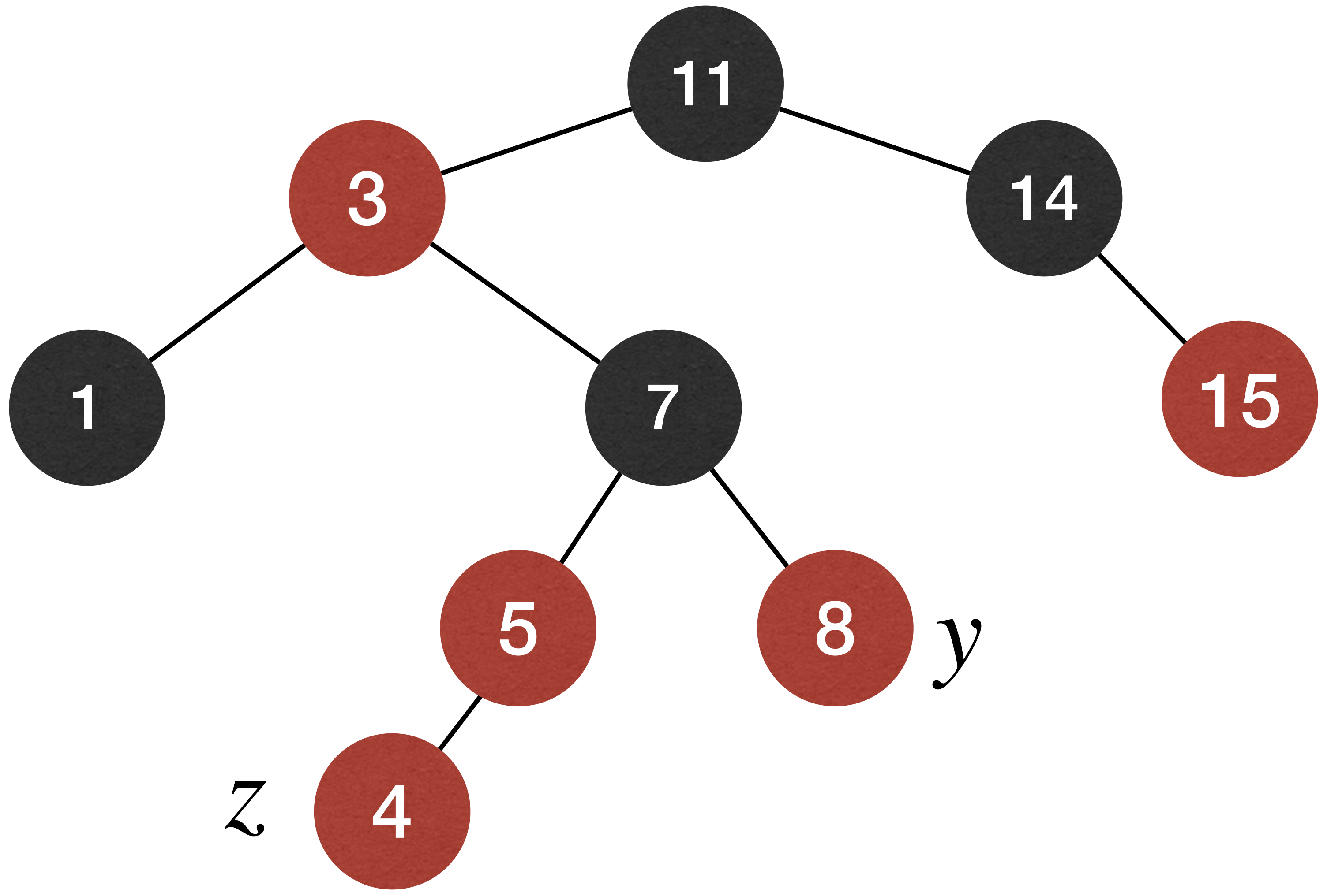
- ✓ Each node is red or black
- ✓ Root is black (easy fix)
- ✓ Leaves are black
- ✓ No-red-edge property
- ✓ Black-height property

Insert node into an RB-Tree

- Step 2: Fix any violated properties.
  - **Case 1**: z's parent is **red** (so z has **black** grandparent), and has **red** uncle y.

Too many red nodes!

Example: Insert element with key 4



RB-Tree Properties

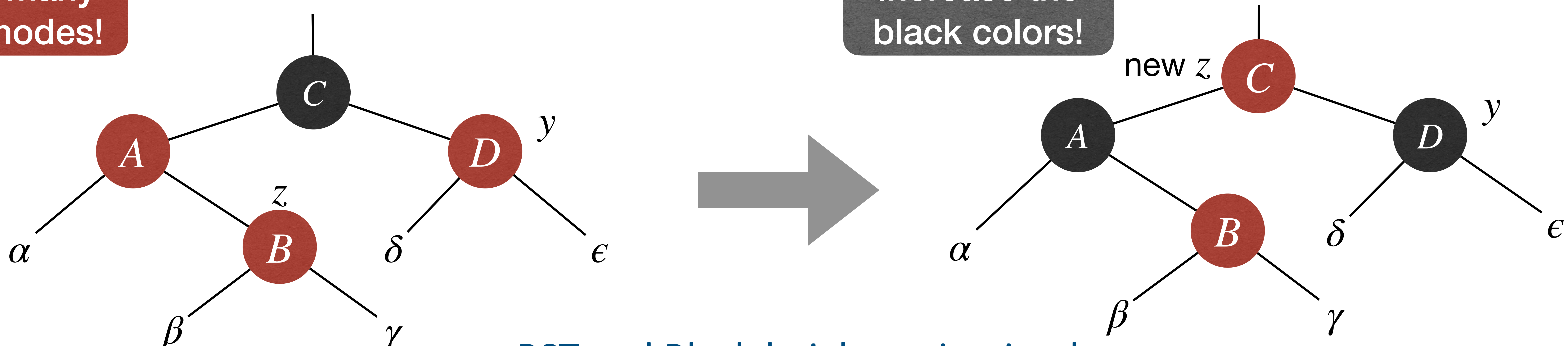
- ✓ Each node is red or black
- ✓ Root is black
- ✓ Leaves are black
- ✓ No-red-edge property (push up!)
- ✓ Black-height property (Maintain)

Insert node into an RB-Tree

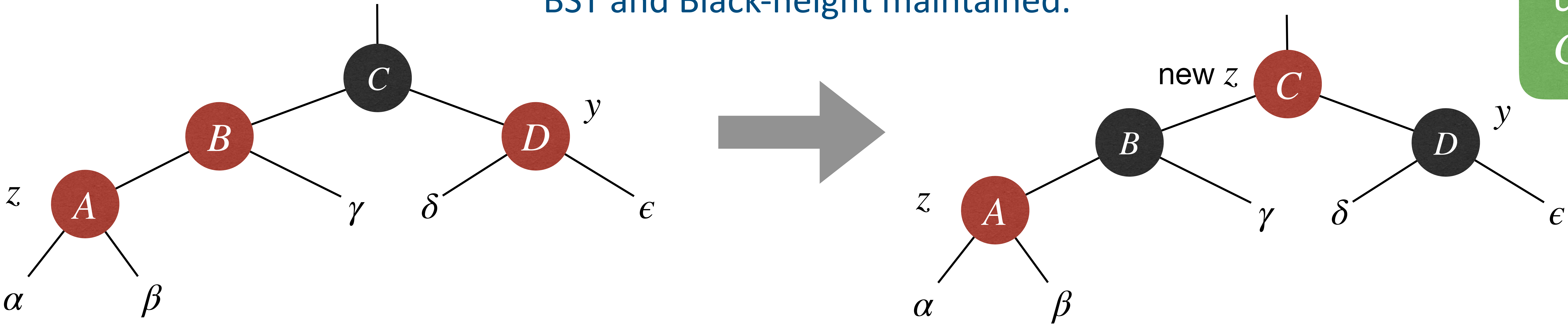
- **Case 1**: z's parent is **red** (so z has **black** grandparent), and has **red** uncle y.
  - Fix: recolor z's parent and uncle to **black**, recolor z's grandparent to **red**

Too many red nodes!

Increase the black colors!



BST and Black-height maintained.



- Black-height property satisfied for all nodes in subtree rooted at C.
- Black-height unchanged for $C.parent$ (if it exists).

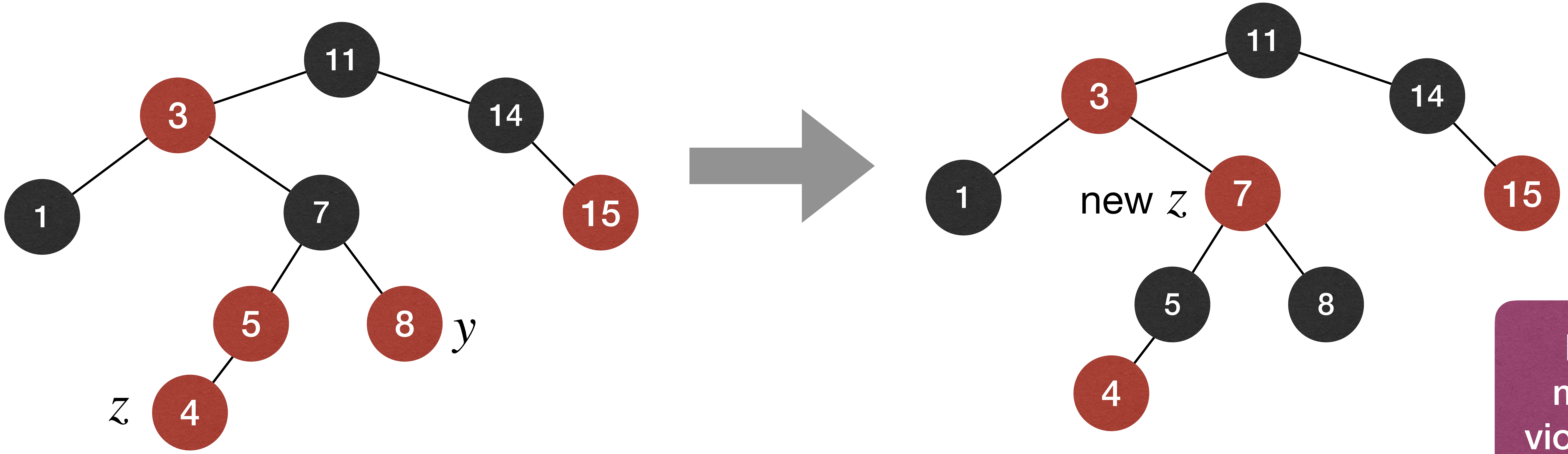
Insert node into an RB-Tree

- Step 2: Fix any violated properties.
  - **Case 1**: z's parent is **red** (so z has **black** grandparent), and has **red** uncle y.

Too many red nodes!

Example: Insert element with key 4

Increase the black colors!



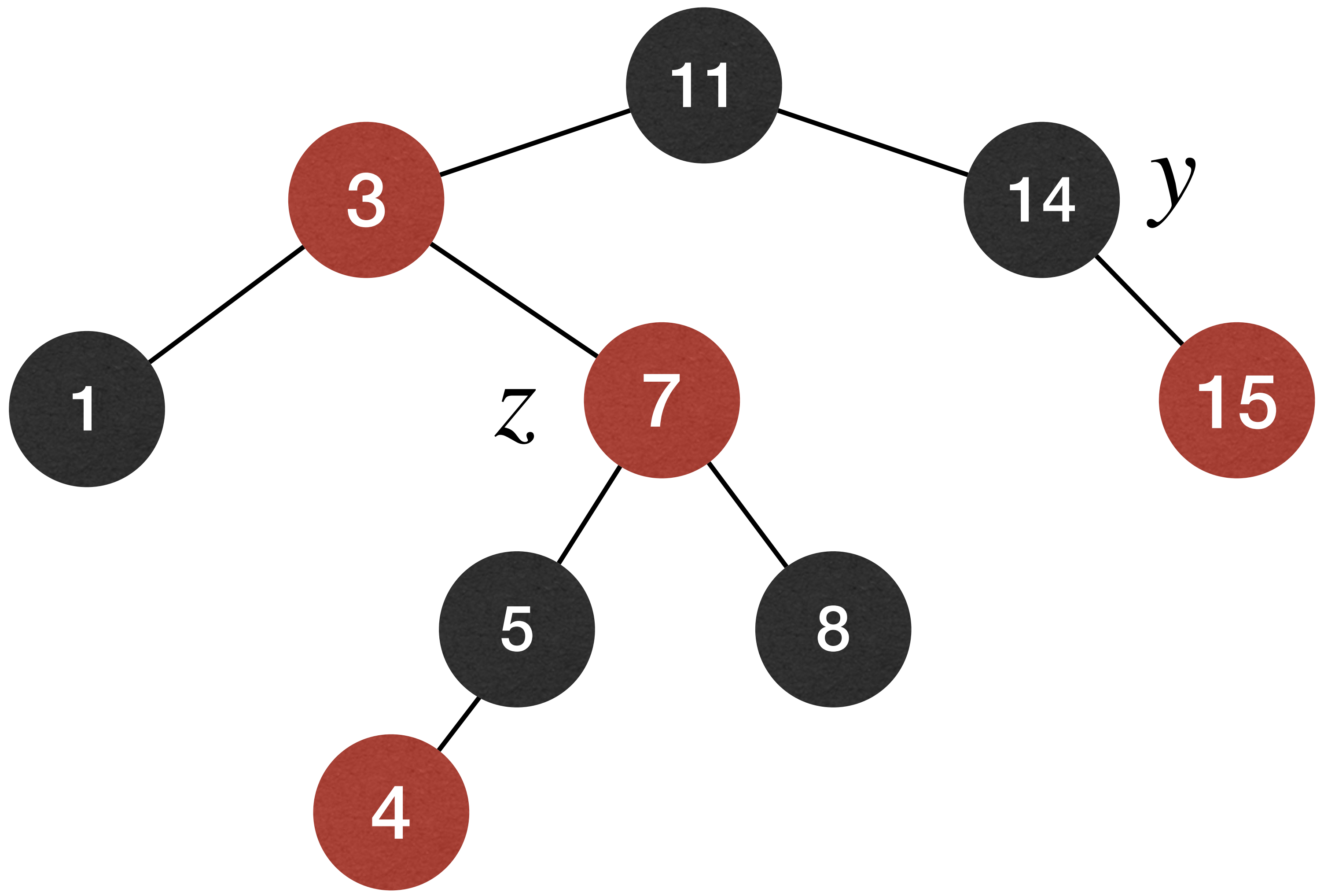
Effect: black-height property maintained, and we "**push-up**" violation of **no-red-edge** property.

Insert node into an RB-Tree

- Step 2: Fix any violated properties.
  - **Case 2**: z's parent is red, has **black** uncle y
    - (a): z is right child of its parent.

a modest number of red nodes

Example: Insert element with key 4



RB-Tree Properties

- ✓ Each node is red or black
- ✓ Root is black
- ✓ Leaves are black
- ✓ No-red-edge property (fix)
- ✓ Black-height property (Maintain)

Insert node into an RB-Tree

- **Case 2(a)**: z's parent is **red**, has **black** uncle y, and z is right child of its parent.
  - Fix: "left-rotate" at z's parent, and then turn to the **case 2 (b)** case!

a modest number of red nodes

rotate to proper location!

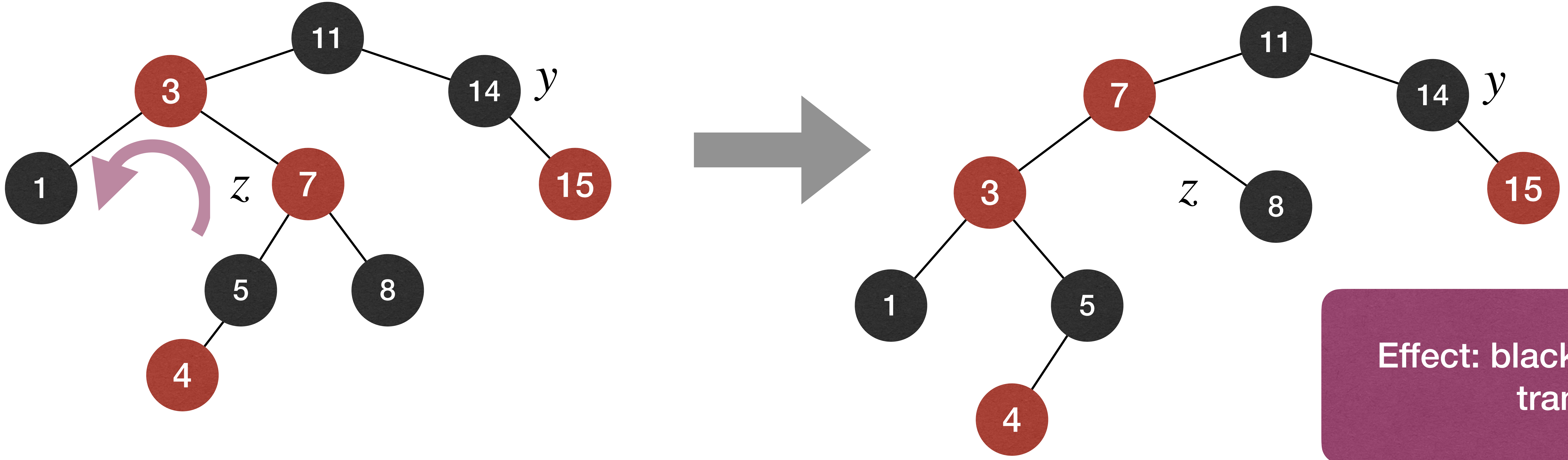
Insert node into an RB-Tree

- Step 2: Fix any violated properties.
  - **Case 2**: z's parent is **red**, has **black** uncle y
    - (a): z is right child of its parent.

a modest number of red nodes

Example: Insert element with key 4

left rotate



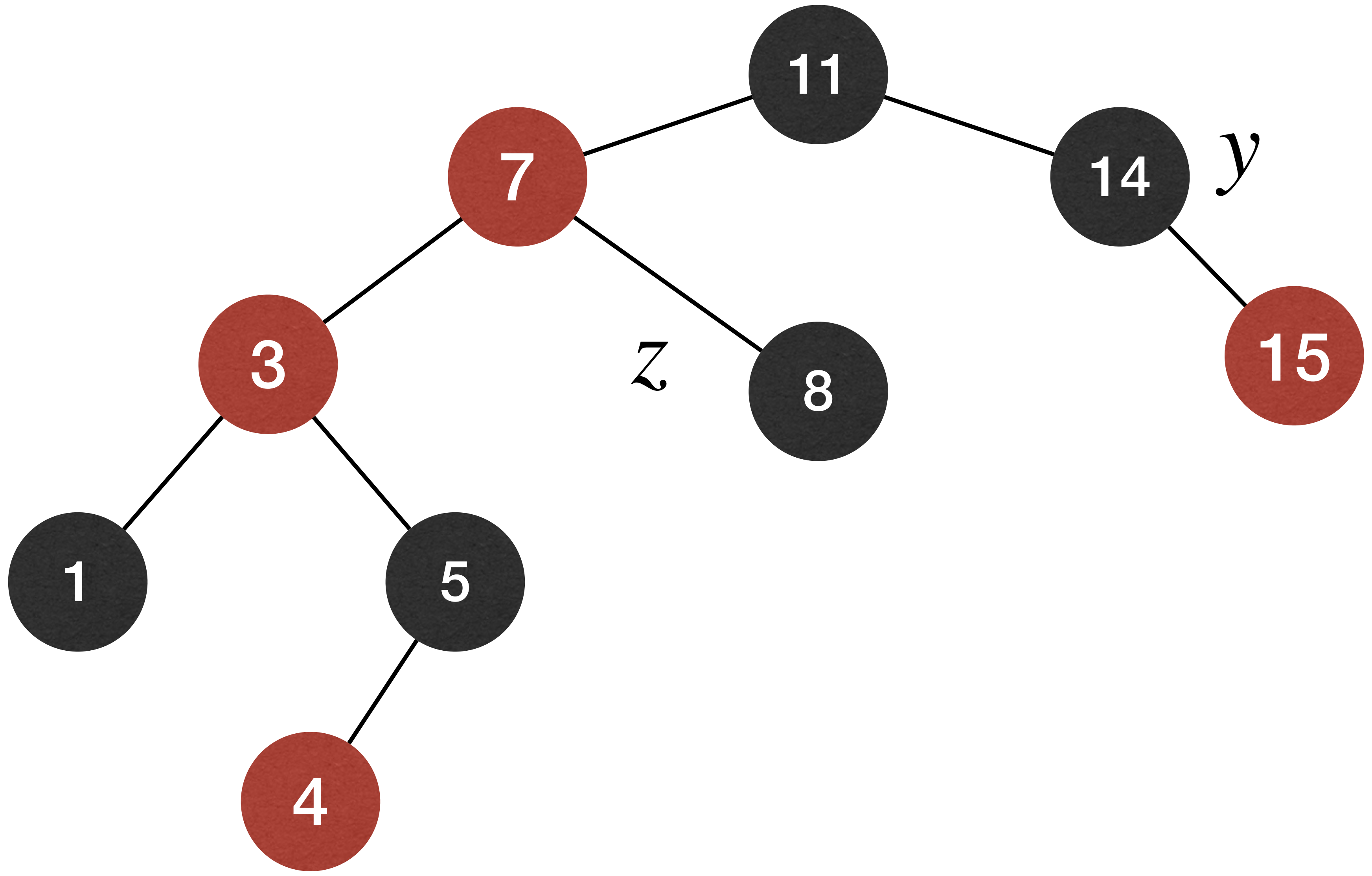
Effect: black-height property maintained, transform to Case 2(b).

Insert node into an RB-Tree

- Step 2: Fix any violated properties.
  - **Case 2**: z's parent is red, has **black** uncle y
    - (b): z is left child of its parent.

a modest number of red nodes

Example: Insert element with key 4

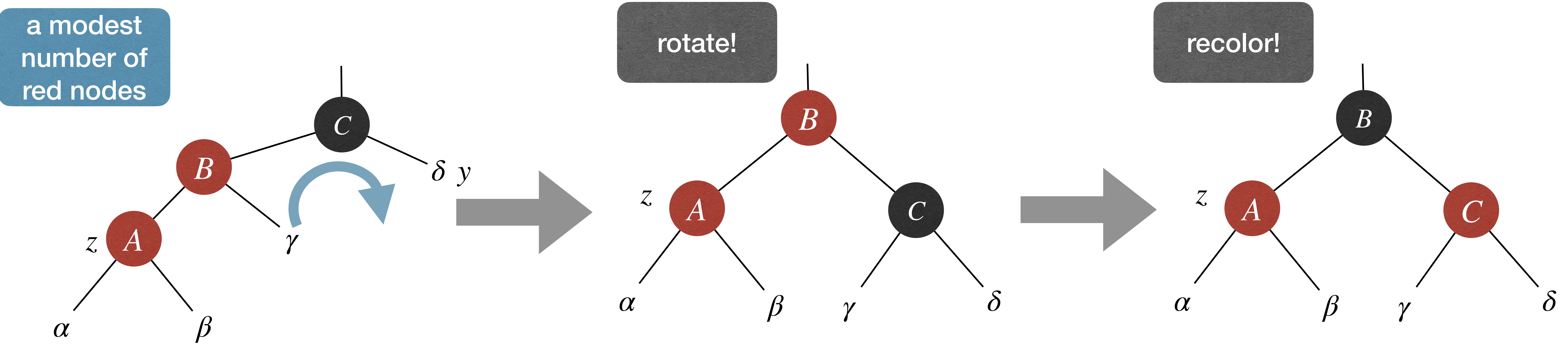


RB-Tree Properties

- ✓ Each node is red or black
- ✓ Root is black
- ✓ Leaves are black
- ✓ No-red-edge property (fix)
- ✓ Black-height property (Maintain)

Insert node into an RB-Tree

- **Case 2(b)**: z's parent is **red**, has **black** uncle y, and z is left child of its parent.
  - Fix: "right-rotate" at z's grandparent, recolor z's parent and grandparent.


a modest number of red nodes

rotate!

recolor!

Insert node into an RB-Tree

- Step 2: Fix any violated properties.
  - **Case 2**: z's parent is red, has **black** uncle y
    - (b): z is left child of its parent.

a modest number of red nodes

Example: Insert element with key 4

right rotate and recolor

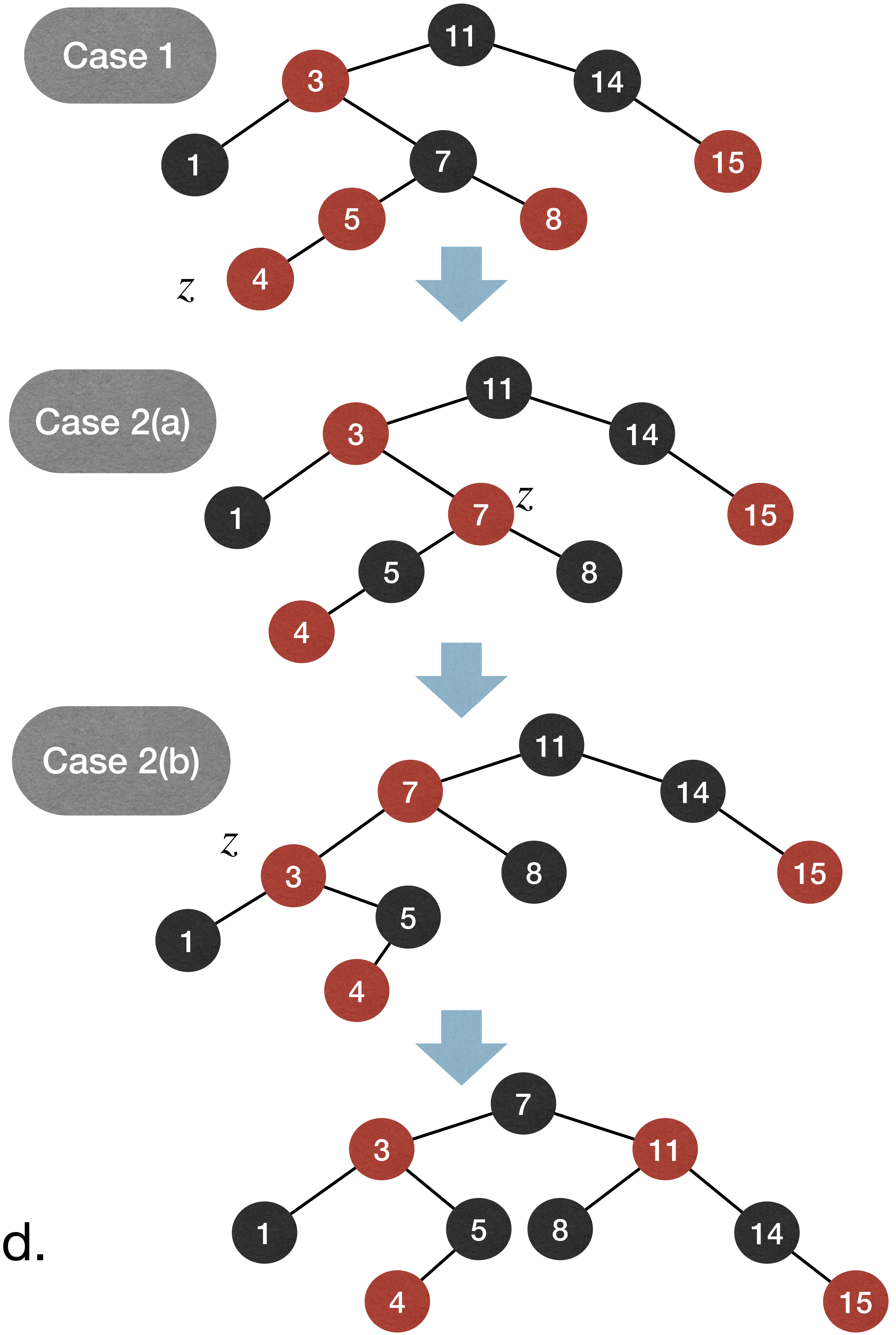
We are done

Insert node into an RB-Tree

Step 1: Color z as red and insert as if the RB-tree were a BST.

Step 2: Fix any violated properties.

- **No-Fix-Needed:** z has a black parent.
- **Case 0**: z becomes the root. - **Fix:** recolor z to be black.
- **Case 1**: z's parent is red, has red uncle.
  - **Fix:** recoloring to push-up "no-red-edge" violation.
- **Case 2**: z's parent is red, has black uncle.
  - **(a)** z is right child of its parent.
    - **Fix:** left-rotate z's parent to transform to Case 2(b).
  - **(b)** z is left child of its parent.
    - **Fix:** right-rotate z's grandparent and recolor nodes, all properties satisfied.



Insert node into an RB-Tree

Step 1: Color z as red and insert as if the RB-tree were a BST.

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

Step 2: Fix any violated properties.

- **No-Fix-Needed:** z has a black parent.

$O(1)$

- **Case 0:** z becomes the root. - **Fix:** recolor z to be black.

$O(1)$

- **Case 1:** z's parent is red, has red uncle.
  - **Fix:** recoloring to push-up "no-red-edge" violation.

appears at most $O(h)$ **times** → $O(\log n)$

- **Case 2:** z's parent is red, has black uncle.
  - **(a)** z is right child of its parent.
    - **Fix:** left-rotate z's parent to transform to Case 2(b).
  - **(b)** z is left child of its parent.
    - **Fix:** right-rotate z's grandparent and recolor nodes, all properties satisfied.

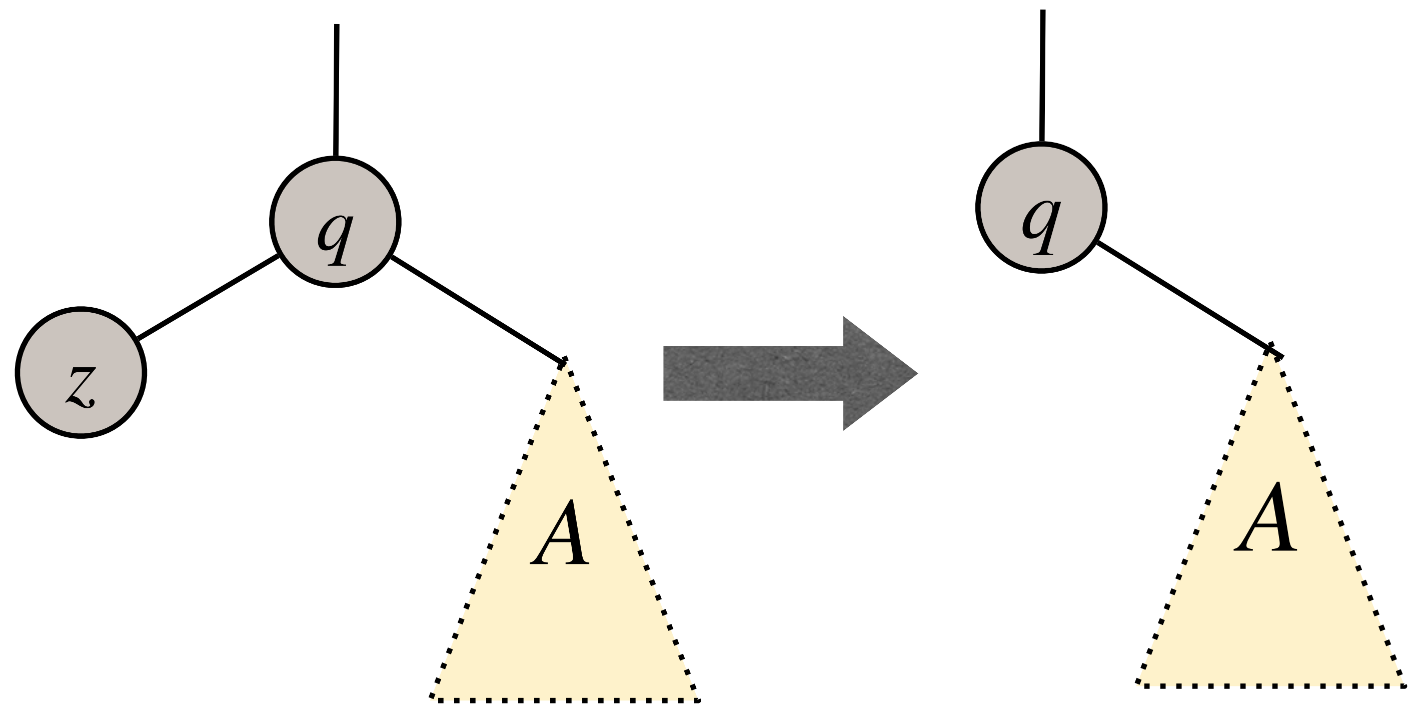
Rotation are limited and do not change the tree shape a lot!

$O(1)$

Time Complexity of Insert operation : $O(\log n)$

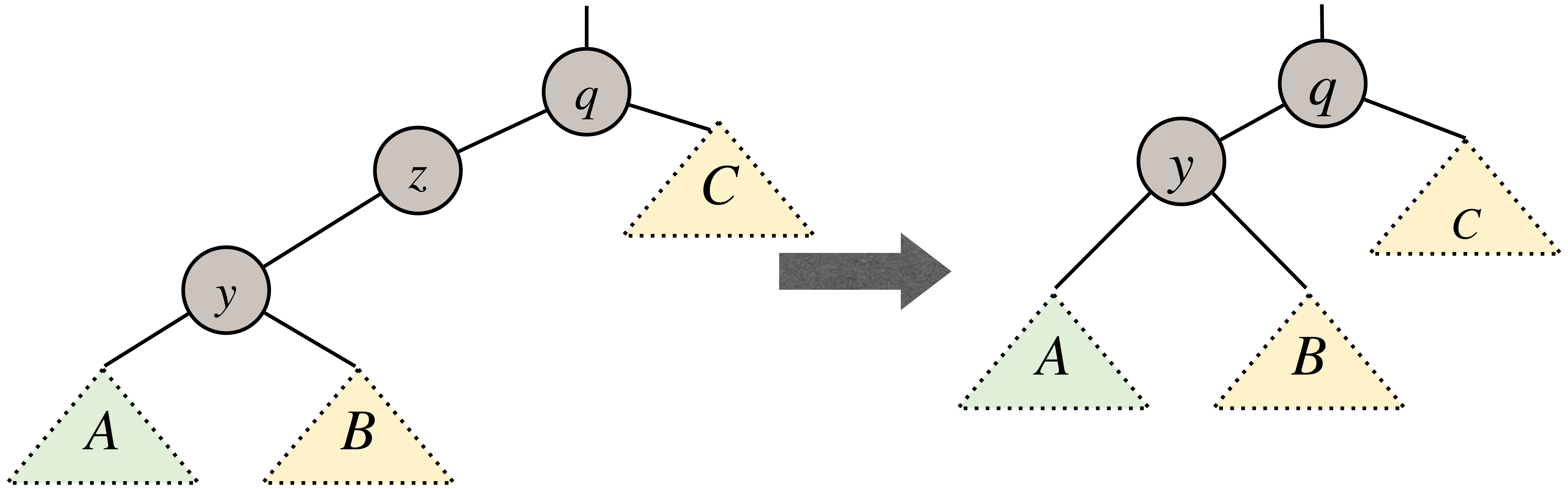
*Remove node from an RB-Tree

- First execute the normal remove operation for BST



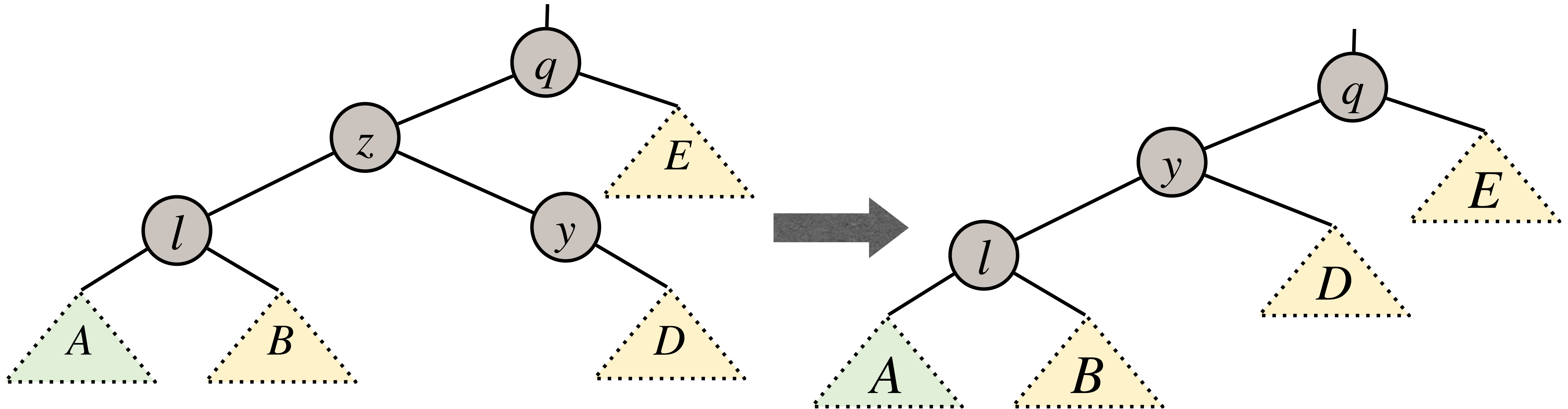
…, z, q, A, …

…, q, A, …



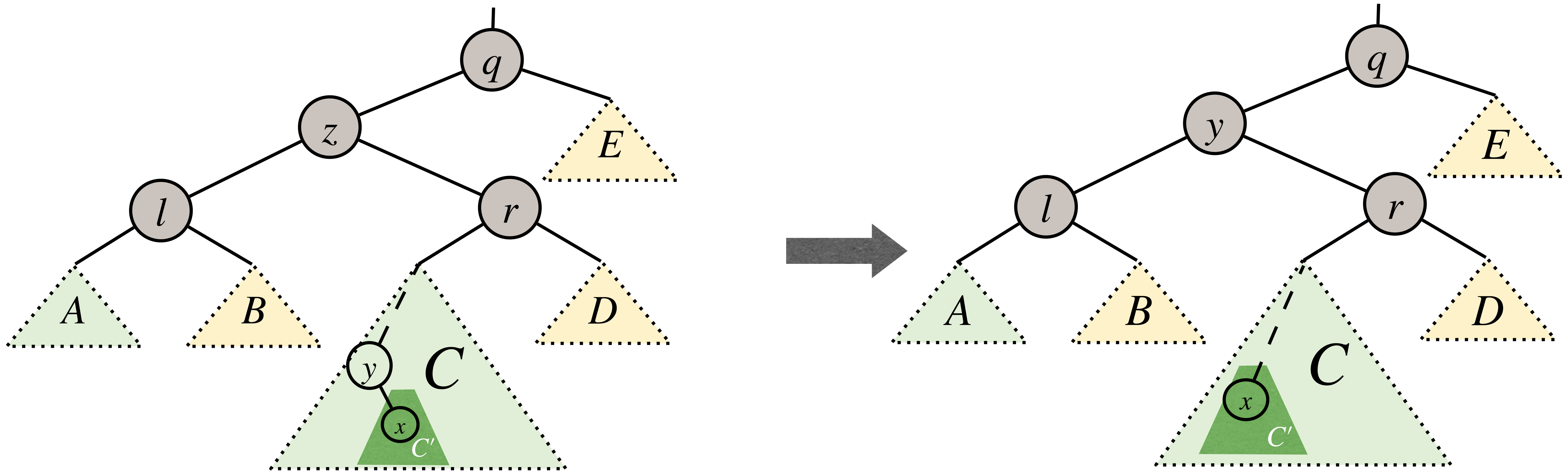
…, z, A, y, B, q, C, …

…, A, y, B, q, C, …



…, A, l, B, z, y, D, q, E …

…, A, l, B, y, D, q, E …

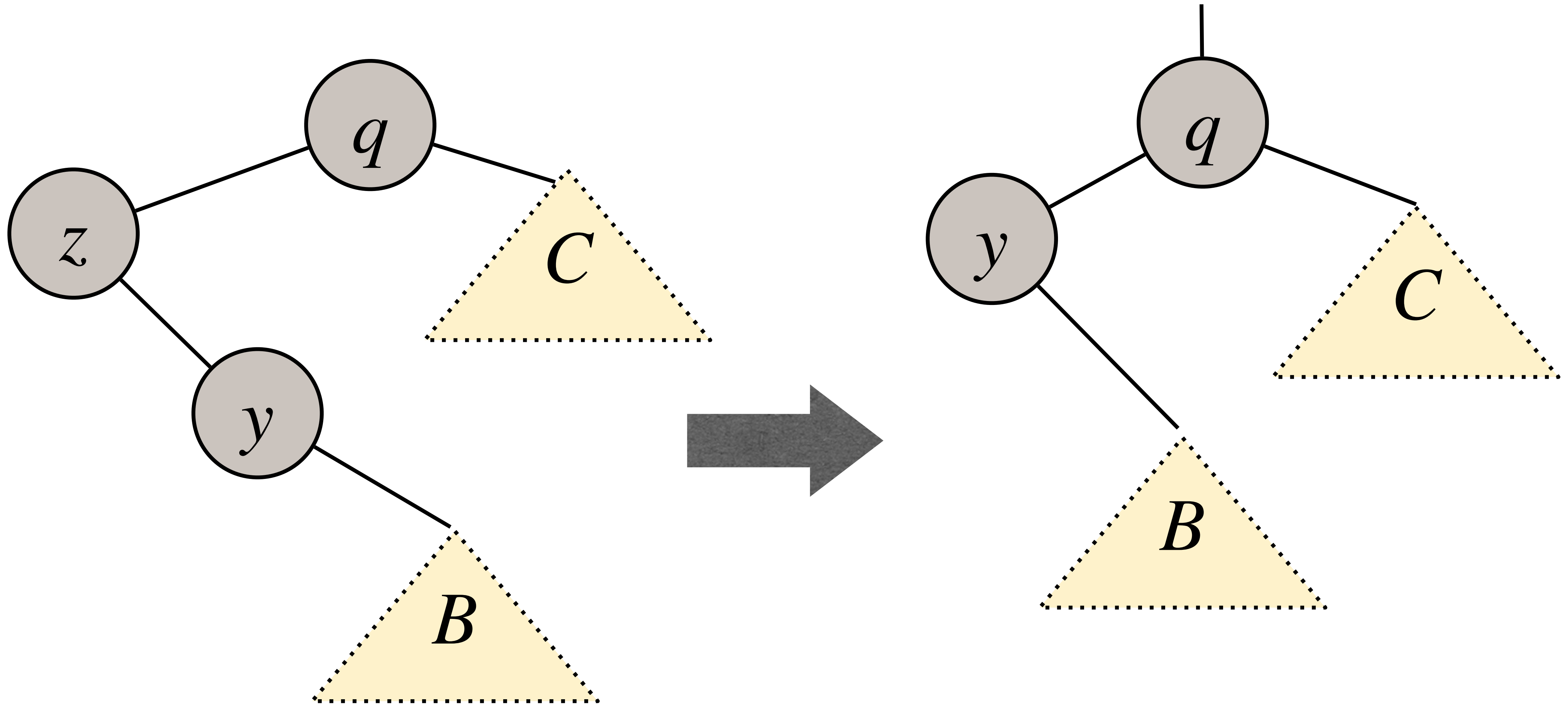


…, A, l, B, z, y, C’, C\C’, r, D, q, E …

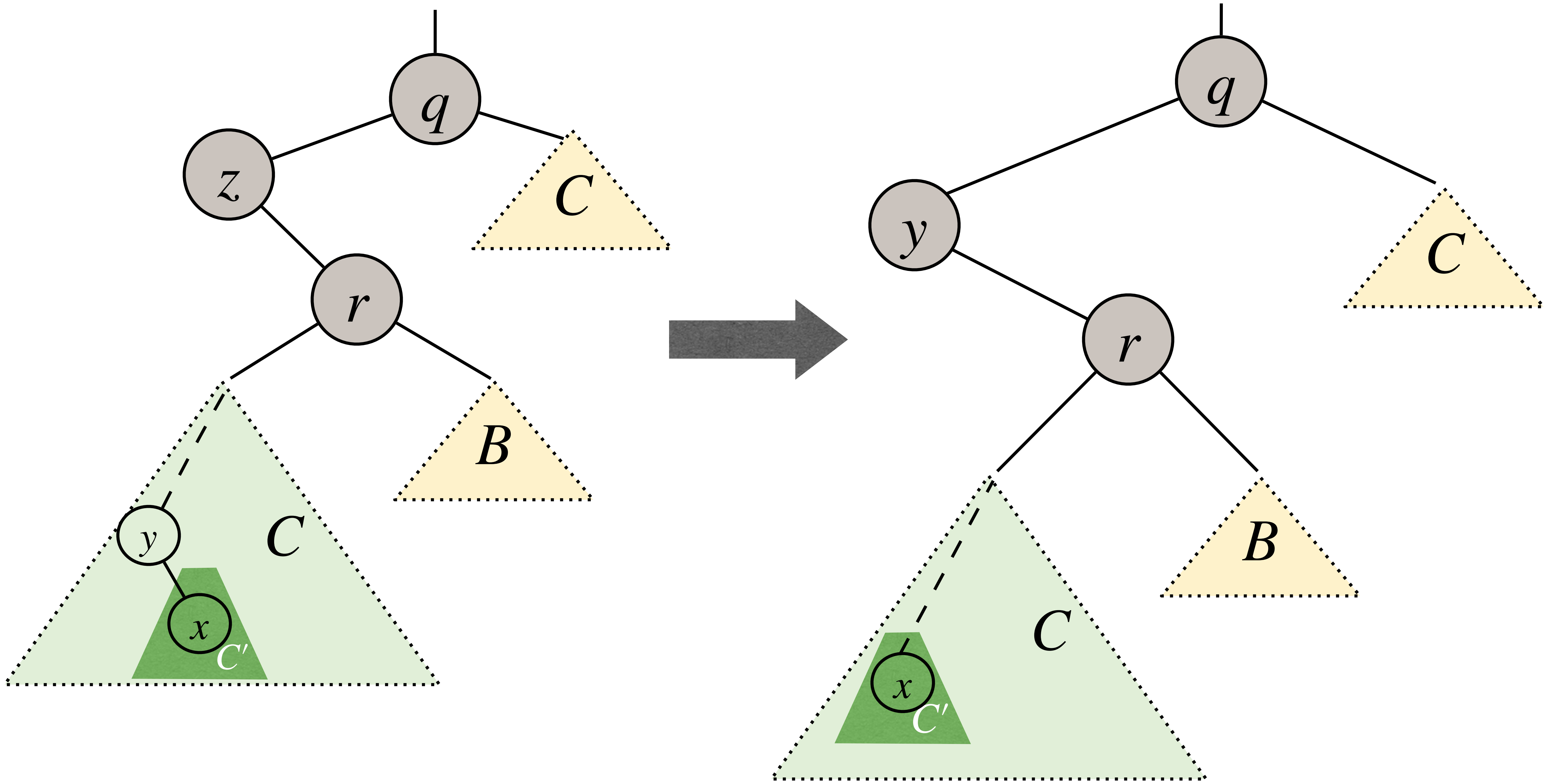
…, A, l, B, y, C’, C\C’, r, D, q, E …

Remove node from an RB-Tree

- To be convenient



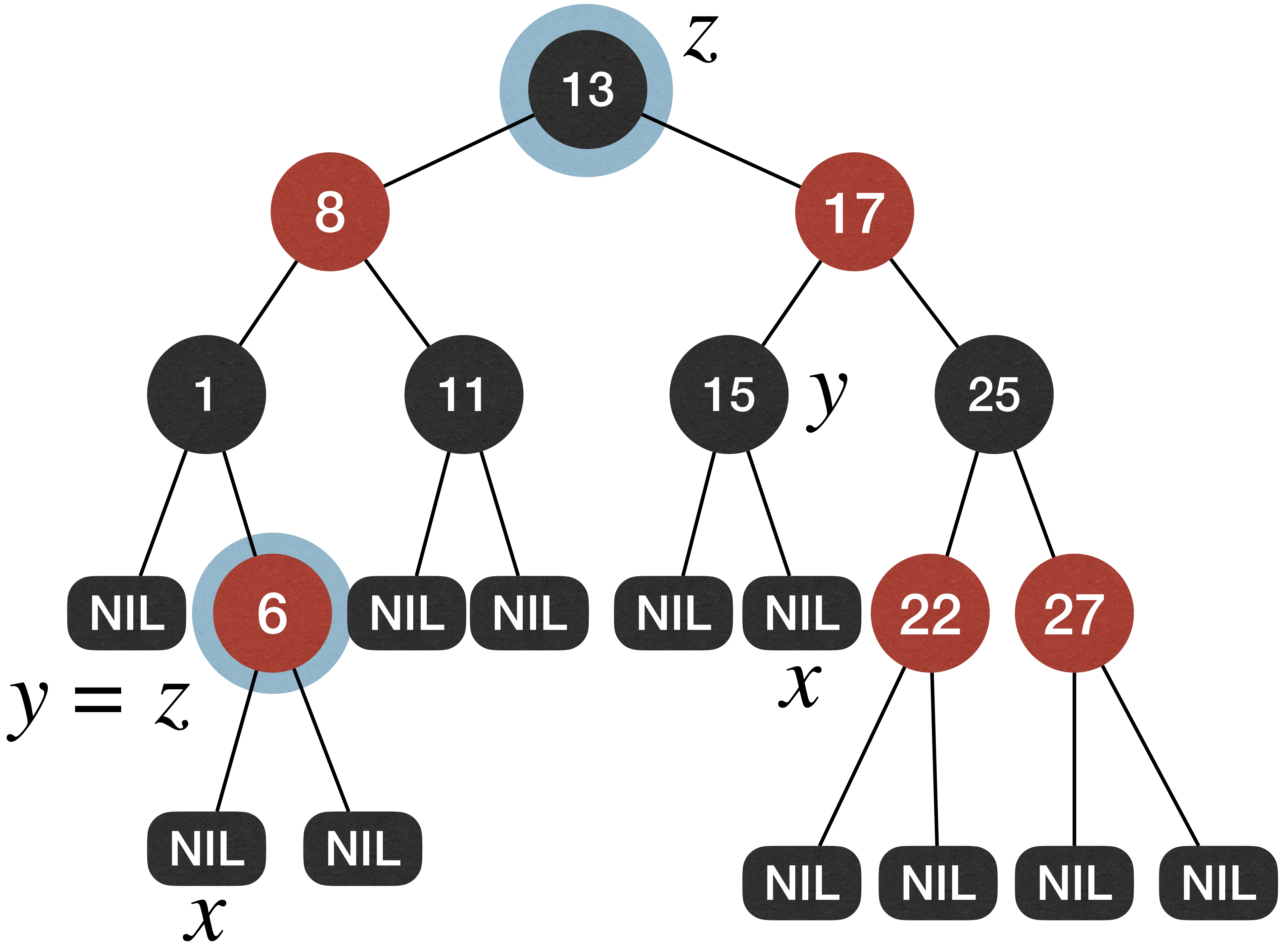
$\ldots, z, y, B, q, C, \ldots$ $\qquad$ $\ldots, y, B, q, C, \ldots$



$\ldots, z, y, C', C \setminus C', r, B, q, C, \ldots$ $\qquad$ $\ldots, y, C', C \setminus C', r, B, q, C, \ldots$

Remove node from an RB-Tree

- If z's right child is an external node (leaf) , then z is the node to be deleted **structurally**: subtree rooted at $z.left$ will replace z.
- If z's right child is an internal node, then let y be the min node in subtree rooted at $z.right$. Overwrite z's info with y's info, and y is the node to be deleted **structurally**: subtree rooted at $y.right$ will replace y.
- Either way, only **one** structural deletion needed!
- Apply the structural deletion, and repair violated properties.



Call the node to be deleted structurally y, and let x be the node that will replace y.

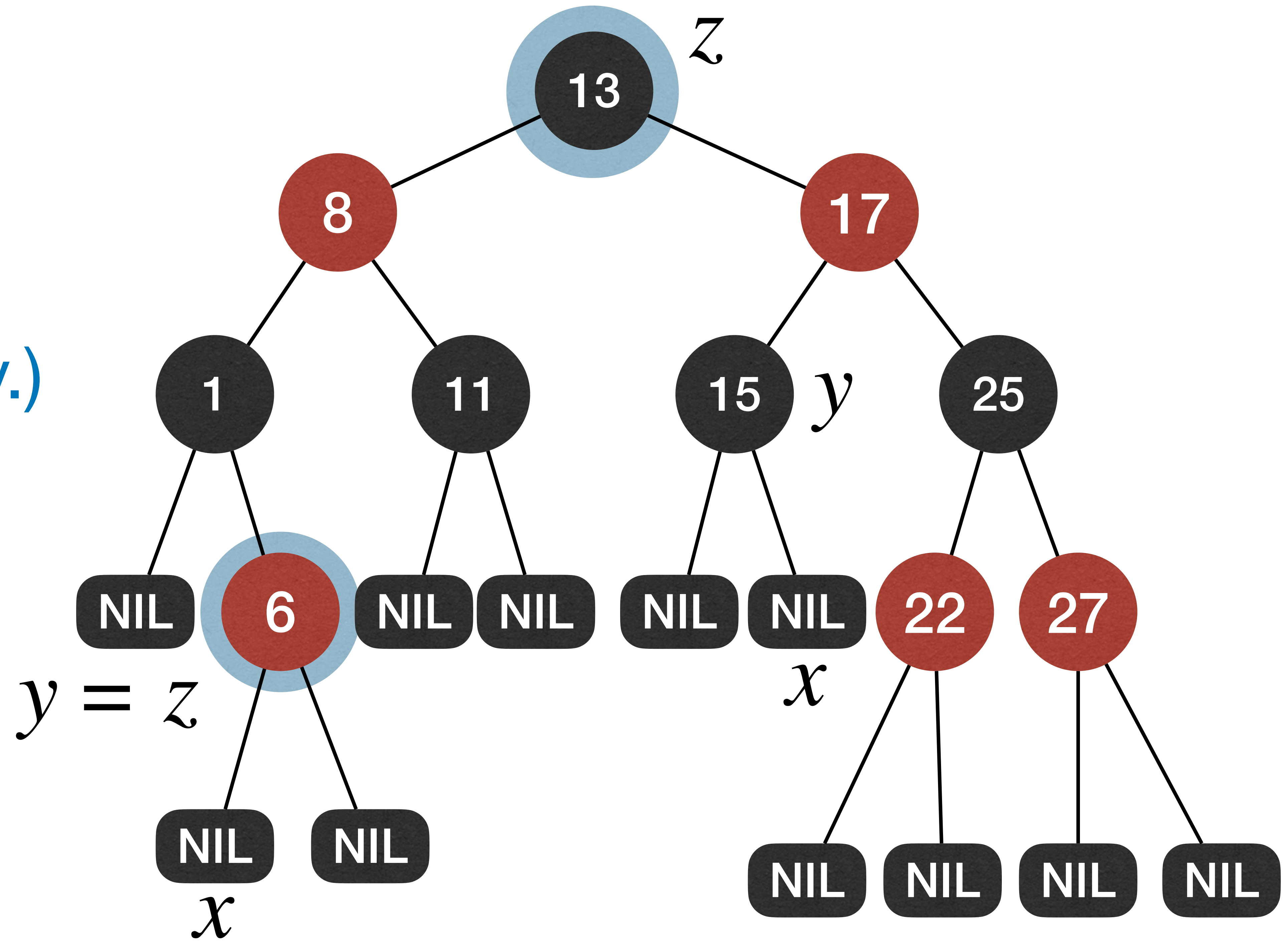
Remove node from an RB-Tree

Step 1: Identify the structural deletion.

Step 2: Apply the structural deletion. (Maintain BST property.)

Step 3: Repair violated RB-tree properties. (Maintain BST property.)

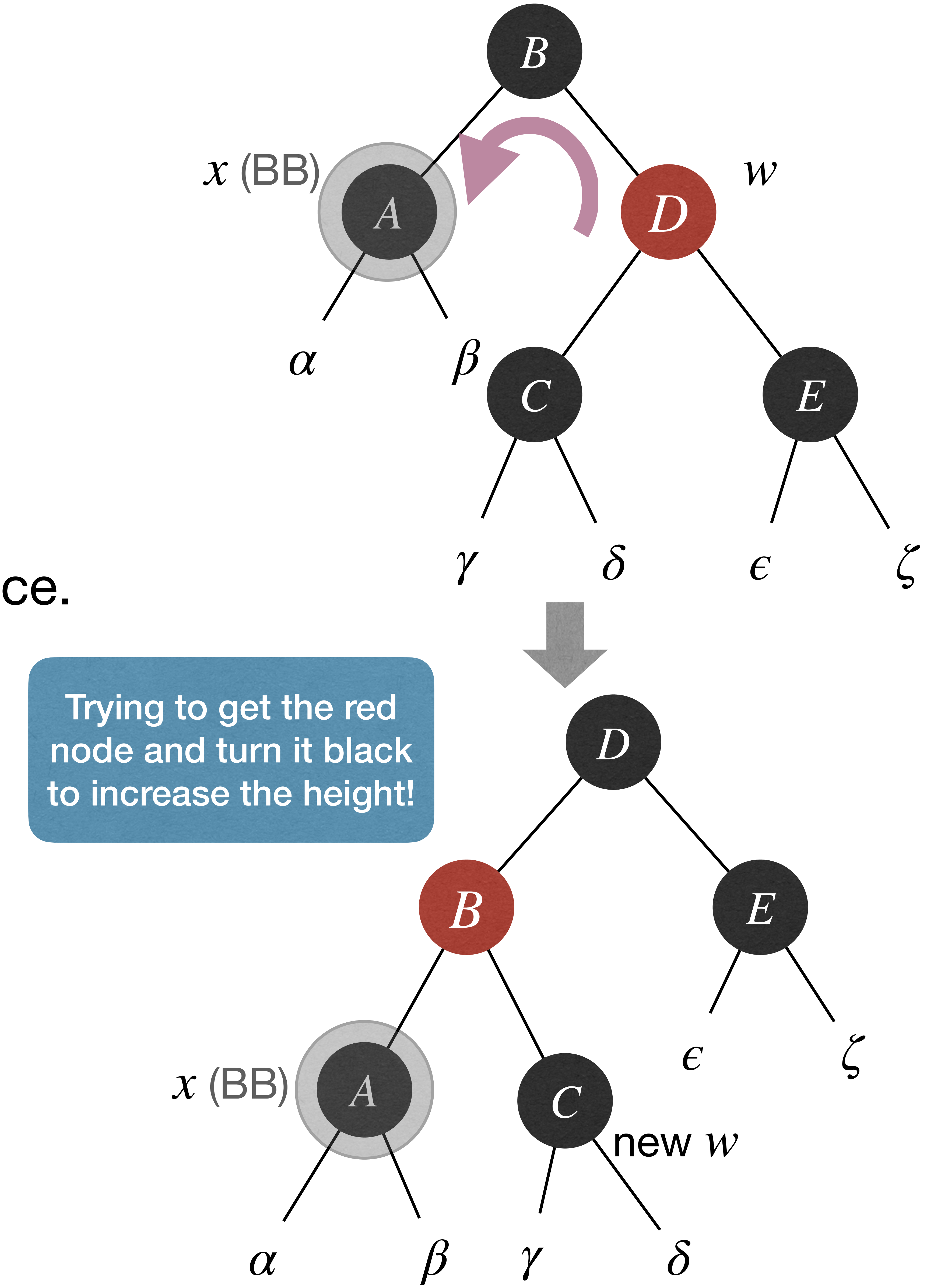
- If y is a red node: no violations.
- If y is a **black** node and x is a red node: recolor x to **black** and done.
- If y is a **black** node and x is a **black** node:
  - y's contribution to **black-height** removed, therefore, it violates **black-height property** → Need to fix!



✓ Each node is red or black
✓ Root is black
✓ Leaves are black
✓ No-red-edge property
✓ Black-height property

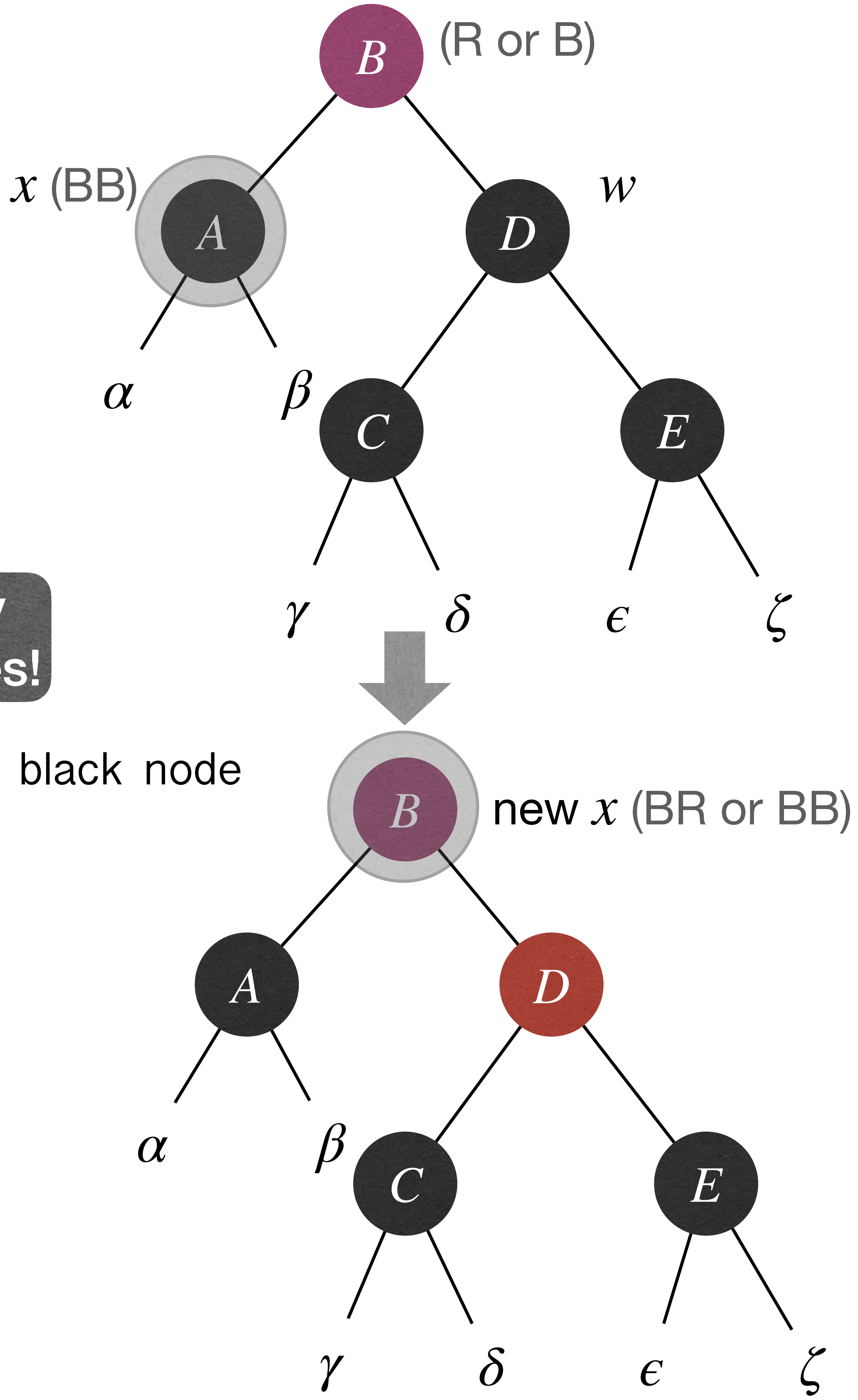
Remove node from an RB-Tree

- Step 1&2: Find & apply structural deletion. (Maintain BST property.)
  - Let y be the structurally removed node, and x takes its place.
- Step 3: Repair violated RB-tree properties. (Maintain BST property.)
  - Assume **black** x is left child of its parent **after** taking **black** y's place.
  - Focus on fixing black-height property.
- Case 1: x's sibling w is **red**.
  - Fix: rotate and recolor.
  - Effect: change x's sibling's color to black (i.e., transform to other cases).



Remove node from an RB-Tree

- Step 1&2: Find & apply structural deletion. (Maintain BST property.)
    - Let y be the structurally removed node, and x takes its place.
- Step 3: Repair violated RB-tree properties. (Maintain BST property.)
    - Assume x is left child of its parent.
    - Focus on fixing black-height property.
- Case 2: x's sibling w is **black,** and both w's children are **black**.
    - Fix: recolor and **push-up** extra blackness in x.
    - Effect: either we are done, or we have **push-up** x.



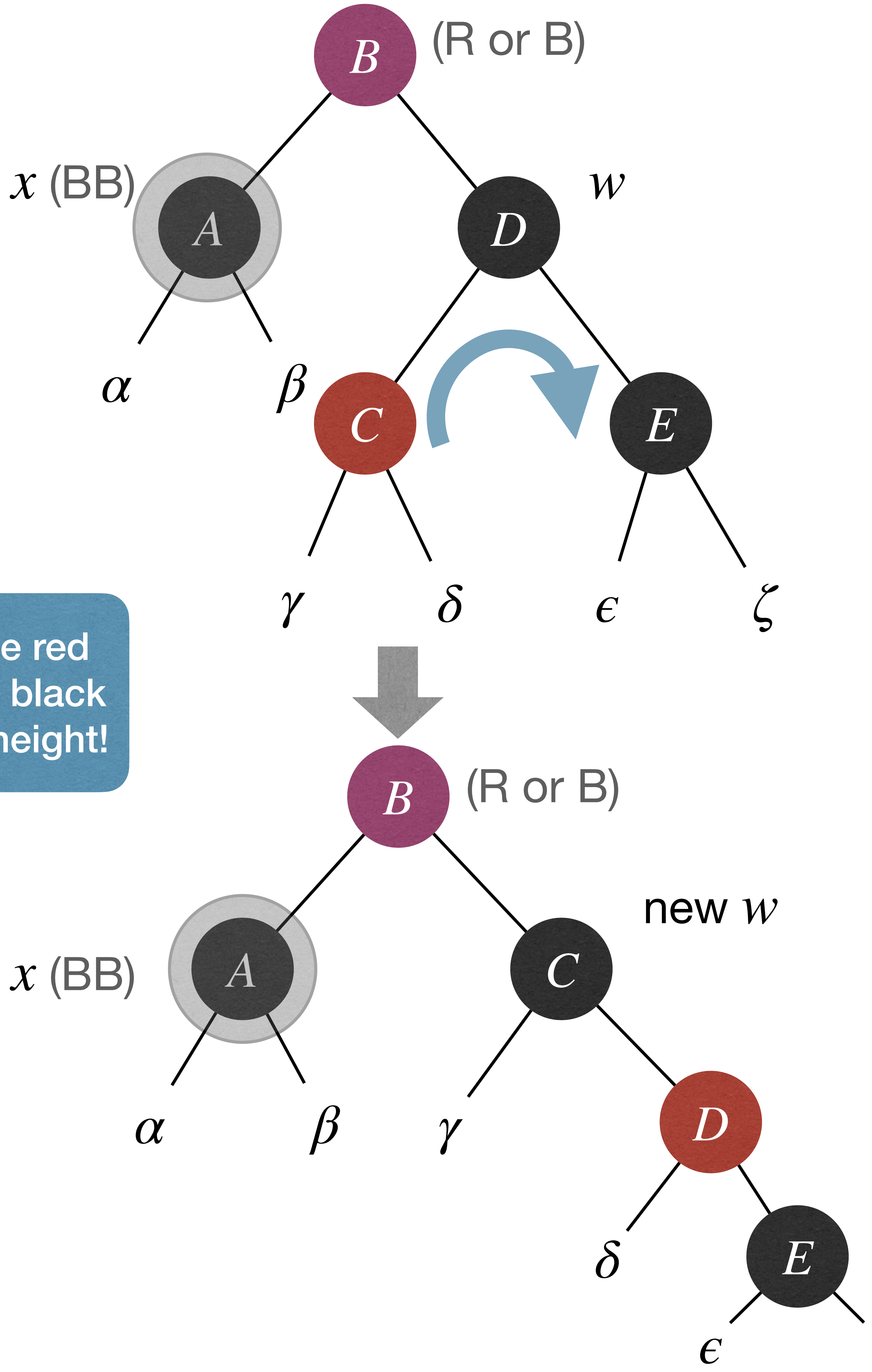
Too many black nodes!

Decrease the node of black node

Remove node from an RB-Tree

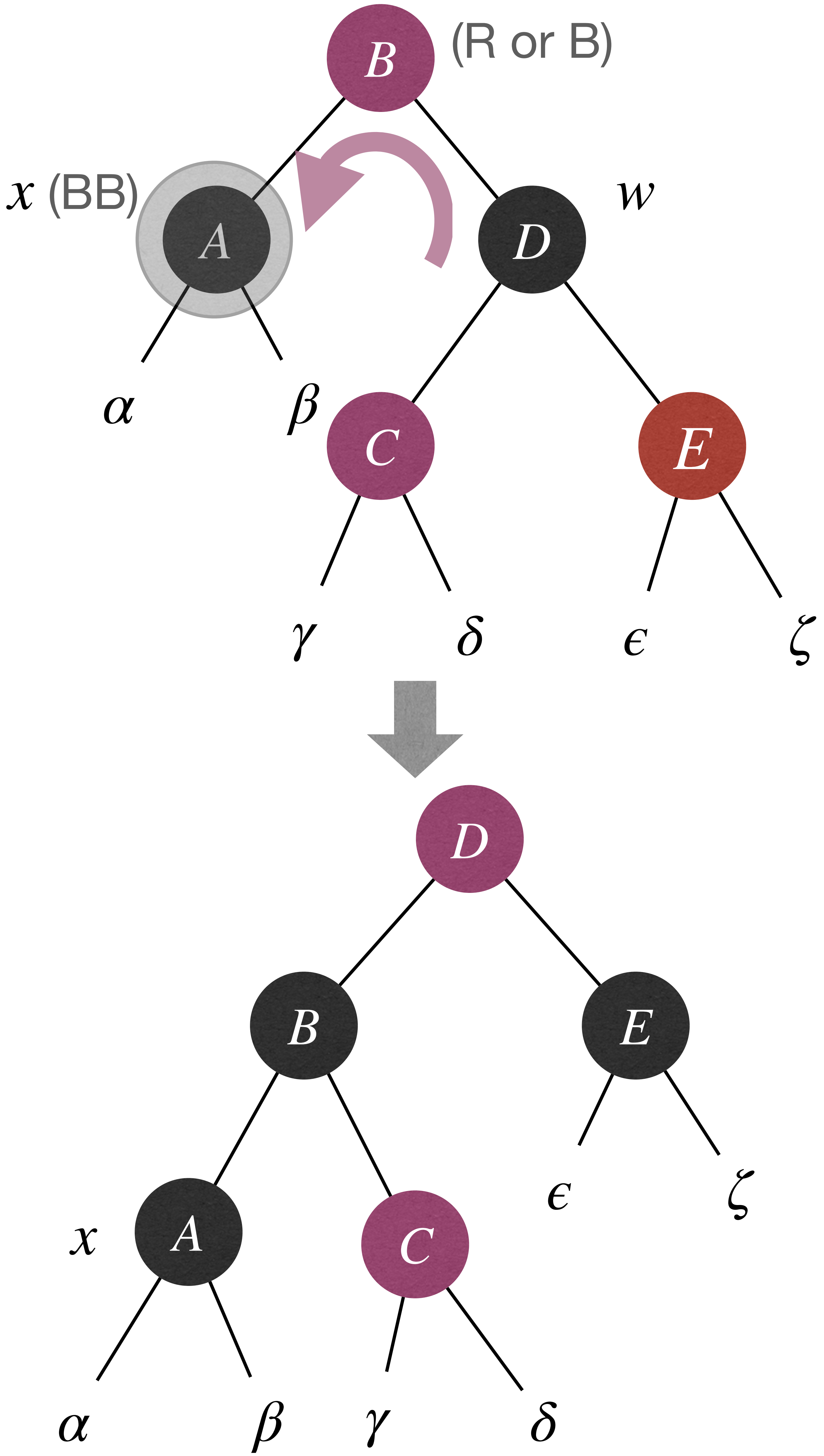
- Step 1&2: Find & apply structural deletion. (Maintain BST property.)
  - Let y be the structurally removed node, and x takes its place.
- Step 3: Repair violated RB-tree properties. (Maintain BST property.)
  - Assume x is left child of its parent.
  - Focus on fixing black-height property.
- Case 3: x's sibling w is **black,** w's left is **red** and w's right is **black**.
  - Fix: rotate and recolor.
  - Effect: $w.right$ becomes red (i.e., transform to last case).

Trying to get the red node and turn it black to increase the height!



Remove node from an RB-Tree

- Step 1&2: Find & apply structural deletion. (Maintain BST property.)
  - Let y be the structurally removed node, and x takes its place.
- Step 3: Repair violated RB-tree properties. (Maintain BST property.)
  - Assume x is left child of its parent.
  - Focus on fixing black-height property.
- Case 4: x's sibling w is **black**, $w.right$ is **red**.
  - Fix: rotate and recolor.
  - Effect: We are done!



Remove node from an RB-Tree

- Step 1&2: Find & apply structural deletion. (Maintain BST property.)
    - Let y be the structurally removed node, and x takes its place.

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

- Step 3: Repair violated RB-tree properties. (Maintain BST property.)
    - Assume **black** x is left child of its parent **after** taking **black** y's place.
    - Focus on fixing black-height property.
        - Case 1: rotate and recolor; transform to other cases.
        - Case 2: recolor; done or push-up violations.
        - Case 3: rotate and recolor; transform to Case 4.
        - Case 4: rotate and recolor; then done.

$O(1)$

appears at most $O(h)$ times
$O(h) = O(\log n)$

$O(1)$

Time Complexity of Remove operation : $O(\log n)$

Efficient implementation of Ordered Dictionary

	`Search(S,k)`	`Insert(S,x)`	`Remove(S,x)`
`BinarySearchTree`	$O(h)$ in worst case	$O(h)$ in worst case	$O(h)$ in worst case
`Treap`	$O(\log n)$ in expectation	$O(\log n)$ in expectation	$O(\log n)$ in expectation
`RB-Tree`	$O(\log n)$ in worst case	$O(\log n)$ in worst case	$O(\log n)$ in worst case

Efficiency versus Simplicity

Skip List

Skip List

	`Search(S,k)`	`Insert(S,x)`	`Remove(S,x)`
`SortedLinkedList`	$O(n)$	$O(n)$	$O(1)$

- Why **sorted** linked-list is slow?
  - To reach an element, you have to move from current position to destination **one element at a time.**

Can we get faster?

Skip List

from: 苏州新区   to: 南京大学

local

苏州新区 鸿福路 兴贤桥北 虎曙路 建通桥 华通花园南 通安站 树山站 白龙桥 恩顾山 漓江路 航船浜 龙塘港路 南京大学

express

苏州新区 兴贤桥北 建通桥 通安站 白龙桥 漓江路 龙塘港路

- Having express stops, we can quickly jump from one express stop to the next express stop.
- Then (if necessary) select the proper express stop to change to the normal stop, and finally jump to the destination.

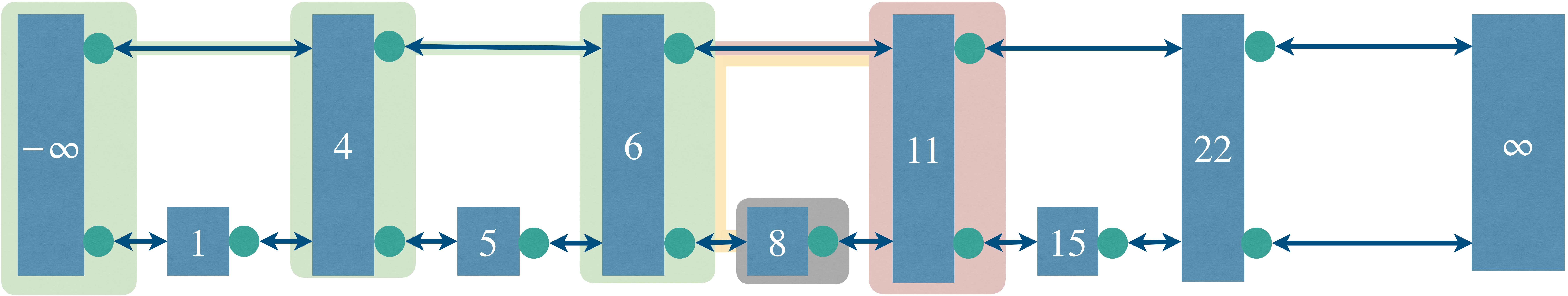
Skip List

- Getting back to the ordered linked list, we can represented it as two linked lists — one for express stops and one for all stops.

🟩 Forward and the next is smaller than target
🟥 Forward and the next is larger than target
🟨 backward and jump to the next level

Example: search for 8.

Search cost is reduced by half!



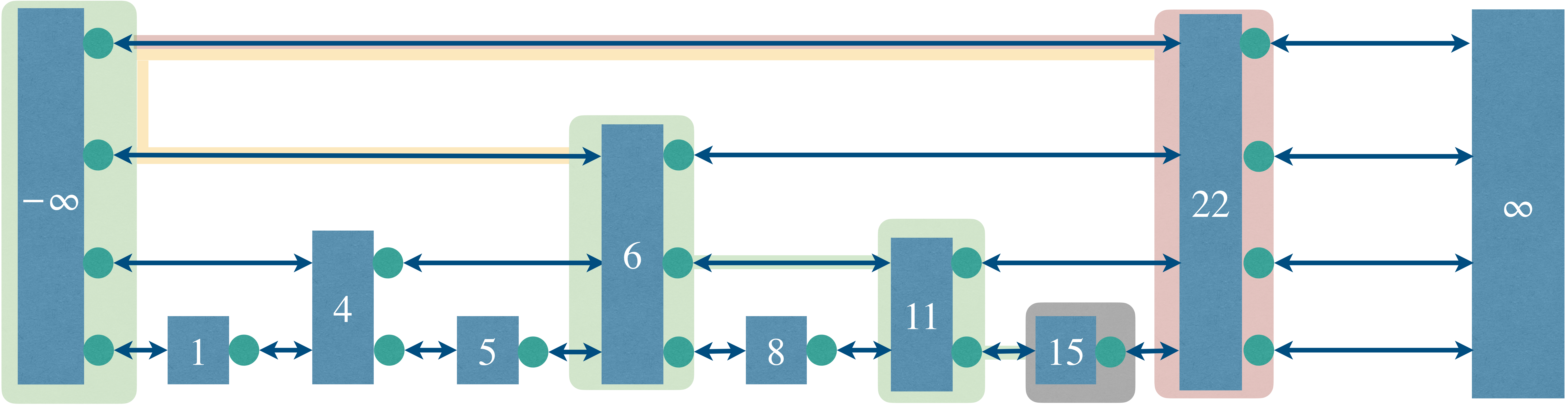
What about multiple layers of “expressway”?

Skip List

- Build multiple “expressways”:Reduce number of elements by half at each level.
  - This is just binary search: reduce search range by half at each level.
- This is very efficient: spend $O(1)$ time at each level, and $O(\log n)$ levels in total.

Example: search for 15.

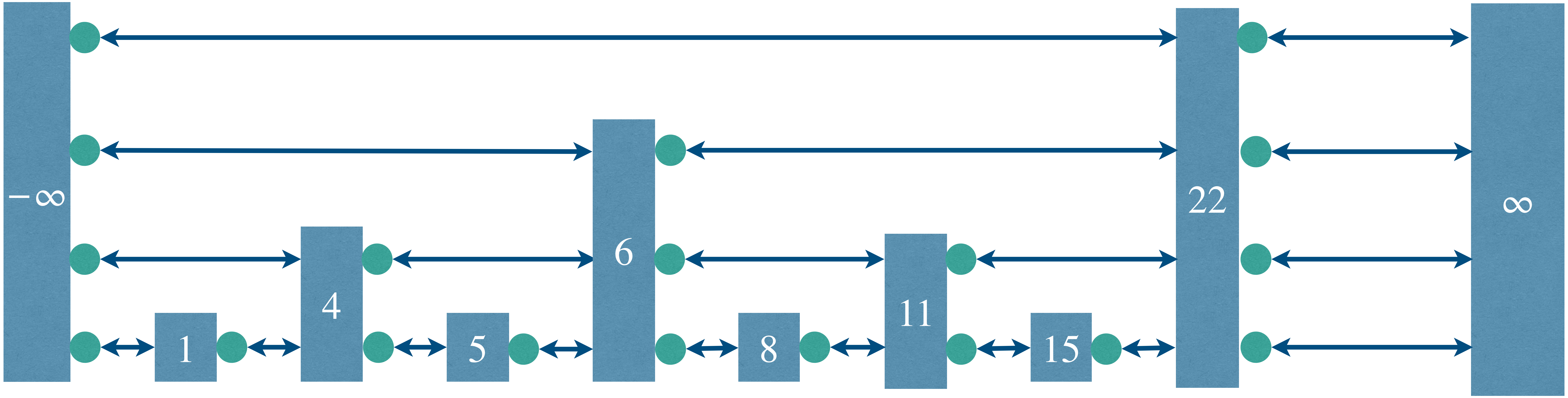
Search can be done in $O(\log n)$ time.
Space Complexity $\approx 2n$



Skip List

	`Search(S,k)`	**`Insert(S,x)`**	**`Remove(S,x)`**
`SortedLinkedList`	$O(n)$	$O(n)$	$O(1)$
`Static-SkipList`	$O(\log n)$	?	?

- Efficient Search with limited space overhead. But how to implement Insert and Remove?

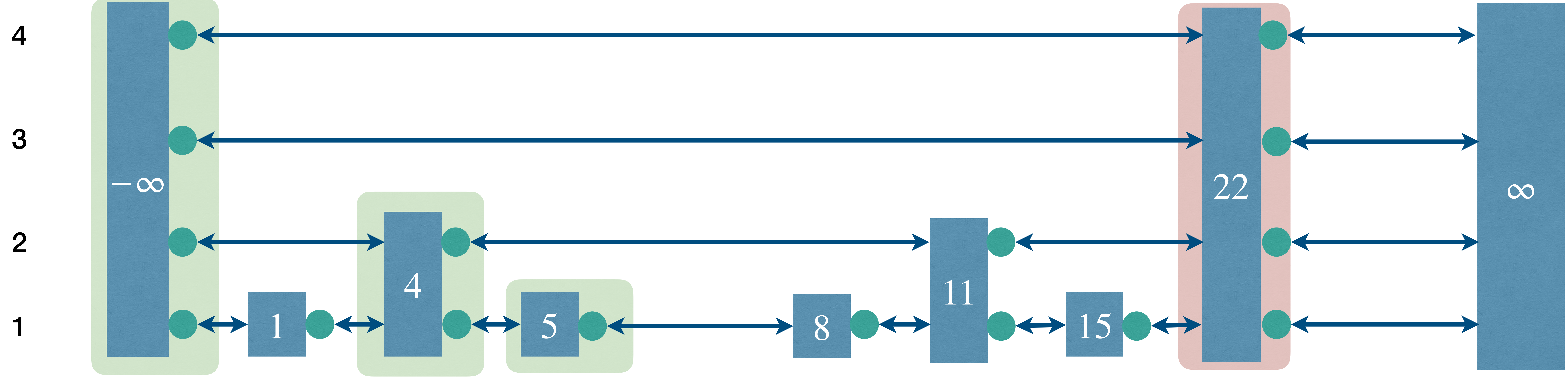


The real Skip List

```
Insert(L,x):
$level := 1, done := false$
while $!done$
    $x := y$
    Insert $x$ into level $k$ list.
    Flip a fair coin:
      With probability $1/2 \rightarrow$ $done := true$
      With probability $1/2 \rightarrow$ $k := k + 1$
```

Lvl:

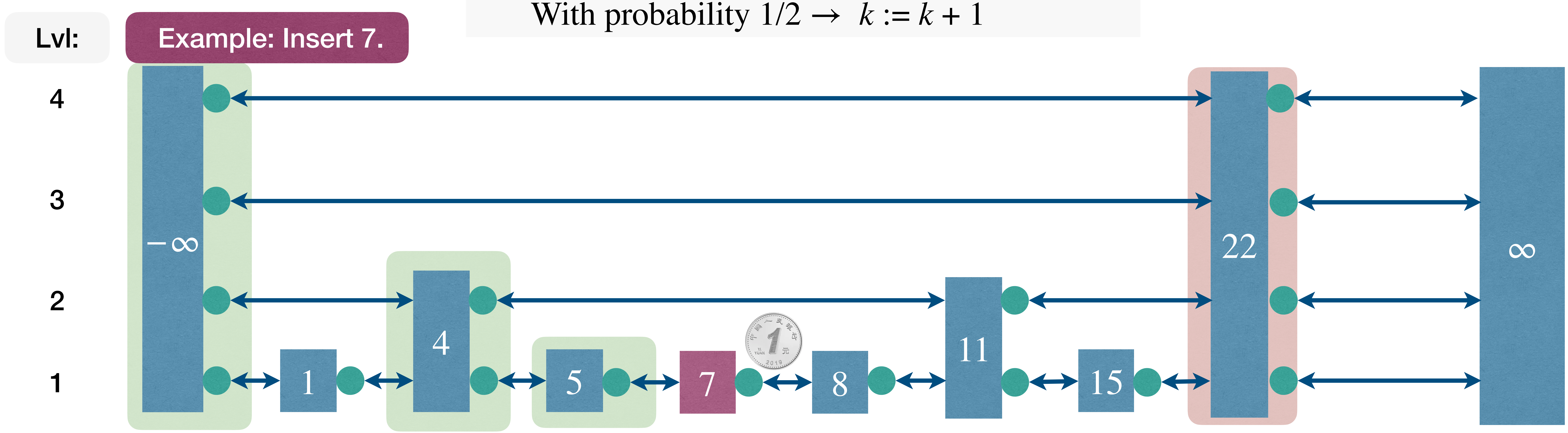
Example: Insert 7.



The real Skip List

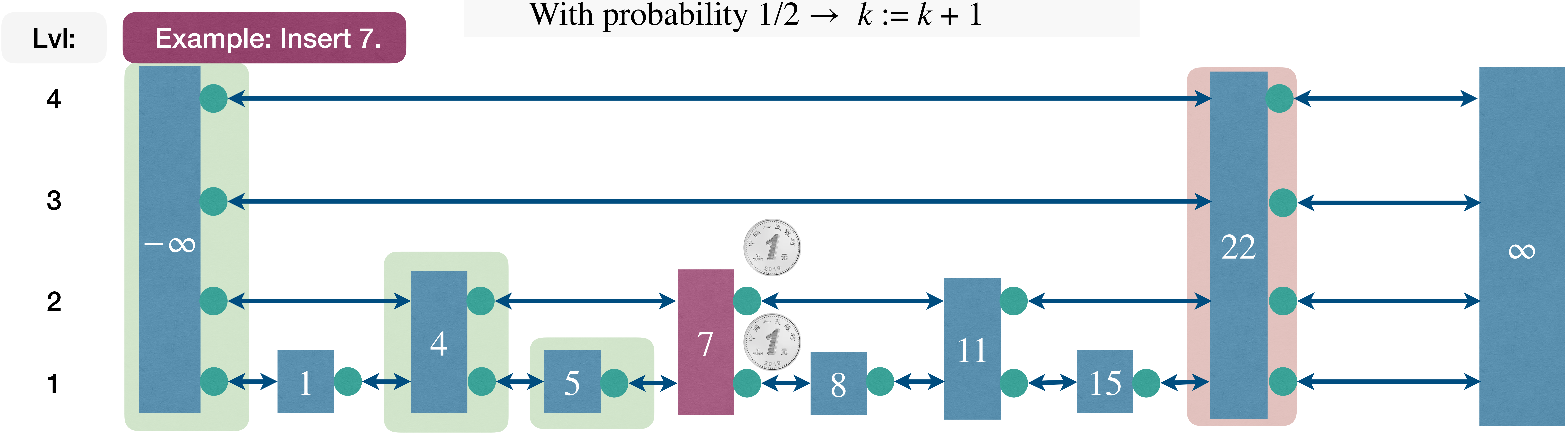
```
Insert(L,x):
$level := 1, done := false$
while $!done$
    $x := y$
    Insert $x$ into level $k$ list.
    Flip a fair coin:
      With probability 1/2 → $done := true$
      With probability 1/2 → $k := k + 1$
```

Example: Insert 7.



The real Skip List

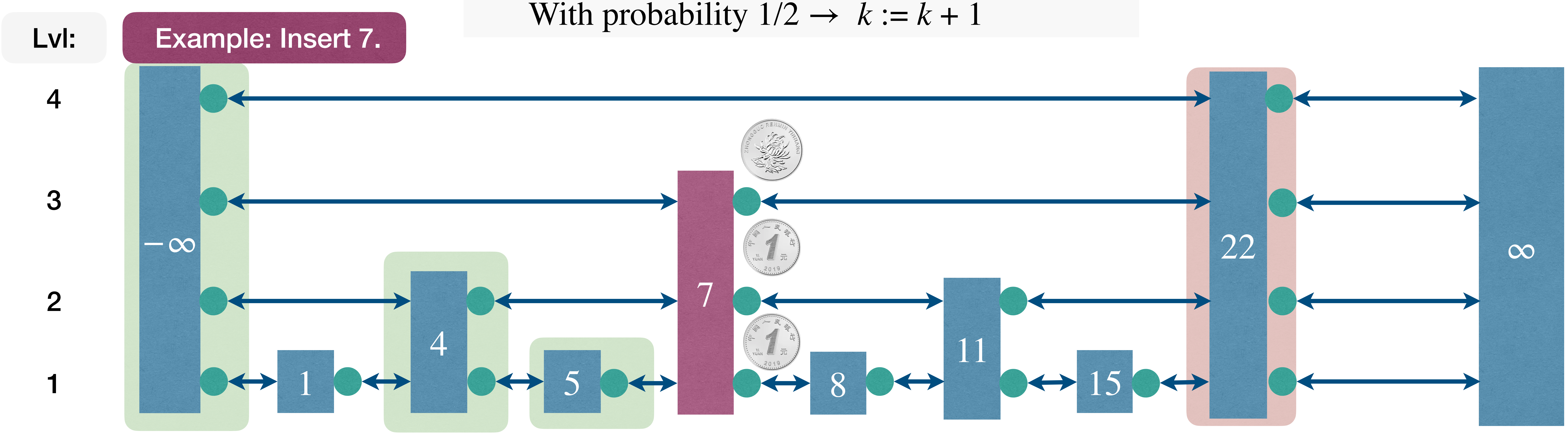
```
Insert(L,x):
$level := 1, done := false$
while $!done$
    $x := y$
    Insert $x$ into level $k$ list.
    Flip a fair coin:
      With probability $1/2 \rightarrow$ $done := true$
      With probability $1/2 \rightarrow$ $k := k + 1$
```


Example: Insert 7.

The real Skip List

Insert(L,x):

```
$level := 1, done := false$
while $!done$
    $x := y$
    Insert $x$ into level $k$ list.
    Flip a fair coin:
      With probability $1/2 \rightarrow$ $done := true$
      With probability $1/2 \rightarrow$ $k := k + 1$
```


Example: Insert 7.

The real Skip List

But search time is affected with such Insert!

~~("Every level reduce search range by half")~~

```
Insert(L,x):
$level := 1, done := false$
while $!done$
    $x := y$
    Insert $x$ into level $k$ list.
    Flip a fair coin:
     With probability $1/2 \rightarrow$ $done := true$
     With probability $1/2 \rightarrow$ $k := k + 1$
```

Time complexity of Insert

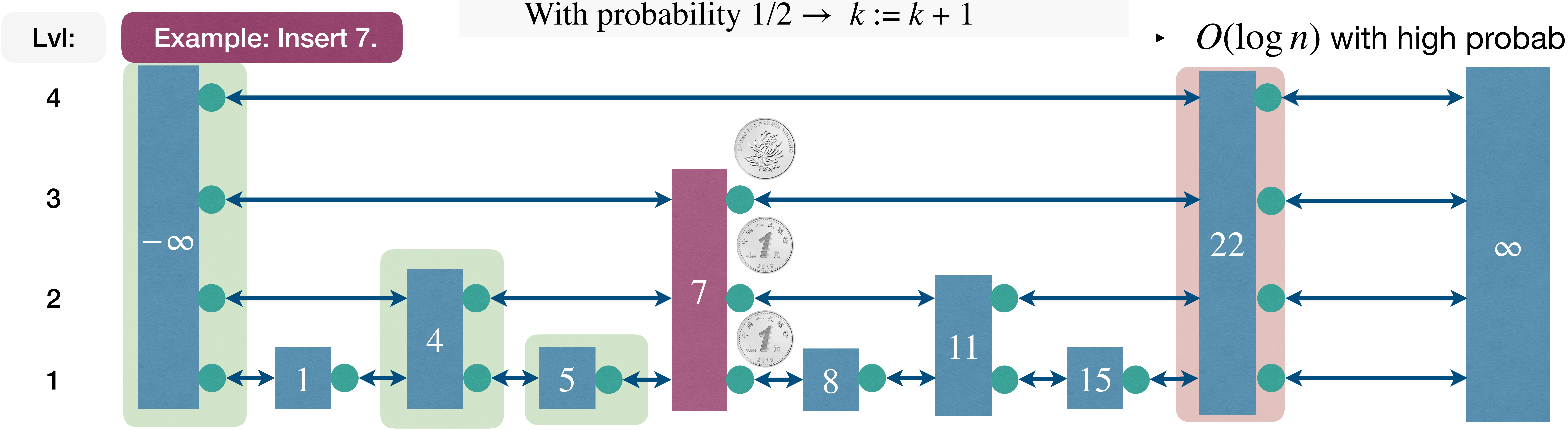
- $O(1)$ in expectation.
- $O(\log n)$ with high probability.

i.e., with prob $\geq 1 - \frac{1}{n^{\Theta(1)}}$

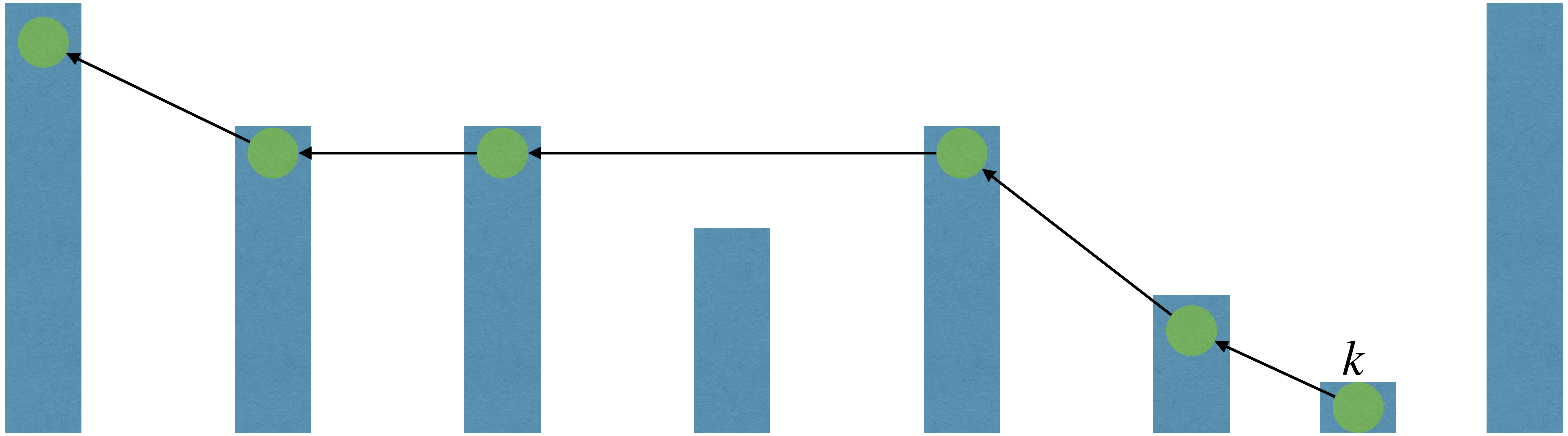
Max level of n-element SkipList

- $O(\log n)$ with high probability.

Example: Insert 7.



The real Skip List



- Consider the **reverse** of the path you took to find k.
- Note that you always move up if you can. (because you always enter a node from its topmost level when doing a find)
- What’s the probability that you can move up at a give step of the reverse walk?

0.5

The real Skip List

- Steps to go up j levels $C(j)$ =
- Make one step, then make either
  - $C(j-1)$ steps if this step went up $[\Pr = 0.5]$
  - $C(j)$ steps if this step went left $[\Pr = 0.5]$
- Expected number of steps to walk up j levels is:
  - $C(j) = 1 + 0.5 \cdot C(j-1) + 0.5 \cdot C(j)$

The real Skip List

- Then we have
    - $C(j) = 2 + C(j-1)$
- Expanding $C(j)$ above getting $C(j) = 2j$
- Since there are $O(\lg n)$ levels expected, we have $O(\lg n)$ steps expected.

Efficient implementation of Ordered Dictionary

	`Search(S,k)`	`Insert(S,x)`	`Remove(S,x)`
`BinarySearchTree`	$O(h)$ in worst case	$O(h)$ in worst case	$O(h)$ in worst case
`Treap`	$O(\log n)$ in expectation	$O(\log n)$ in expectation	$O(\log n)$ in expectation
`RB-Tree`	$O(\log n)$ in worst case	$O(\log n)$ in worst case	$O(\log n)$ in worst case
`SkipList`	$O(\log n)$ in expectation	$O(\log n)$ in expectation	$O(\log n)$ in expectation

Efficiency versus Simplicity

Further reading

- [CLRS] Ch.13
- [Morin] Ch.4