COMP 251

Algorithms & Data Structures (Winter 2021)

Red-Black Trees

School of Computer Science
McGill University

Based on (Cormen *et al.*, 2002) & slides of (Waldispuhl,2020), and (D. Plaisted).

Announcements

- A1 has been released.
  - Please start early.
  - Try by yourself + Use discussion board + OH.

Outline

- Introduction.
- Operations.

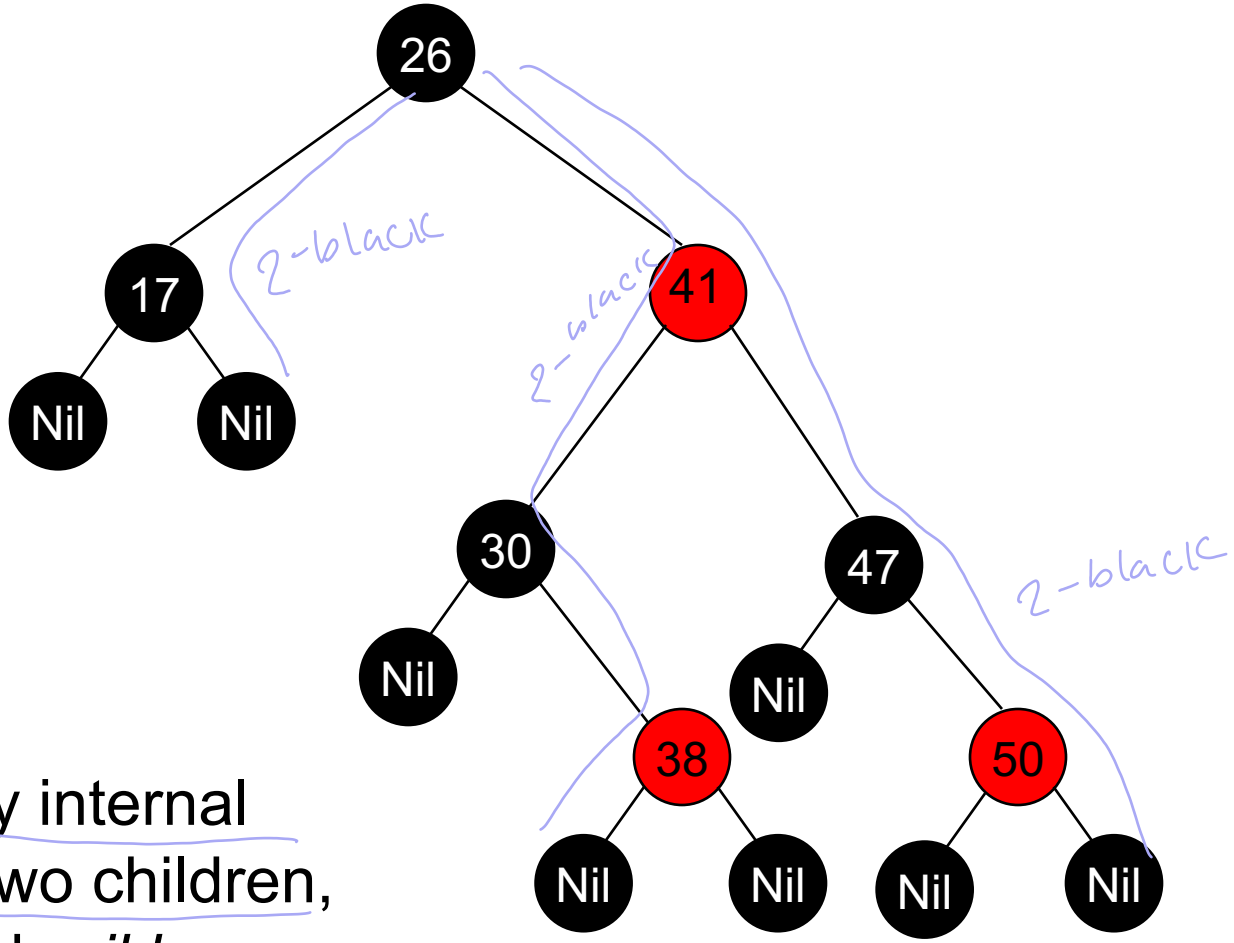
Introduction – Red-Black trees

- Red-black trees are a variation of binary search trees to ensure that the tree is 'approximately' **balanced**.
  - Height is $O(\lg n)$, where n is the number of nodes.
  - BST + 1 bit per node: the attribute color, which is either **red** or **black**.
  - All other attributes of BSTs are inherited (*key*, *left*, *right*, and *parent*).
  - no path (from the root to a leaf) is more than twice as long as any other path.
- Operations take $O(\lg n)$ time in the worst case.
- Invented by R. Bayer (1972).
- Modern definition by L.J. Guibas & R. Sedgewick (1978).

Red-Black trees - Properties

1. Every node is either red or black.
2. The root is black.
3. All leaves (*nil*) are black.
4. If a node is red, then its children are black (i.e. no 2 consecutive red nodes).
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes (i.e. same black height).

Red-Black trees - example



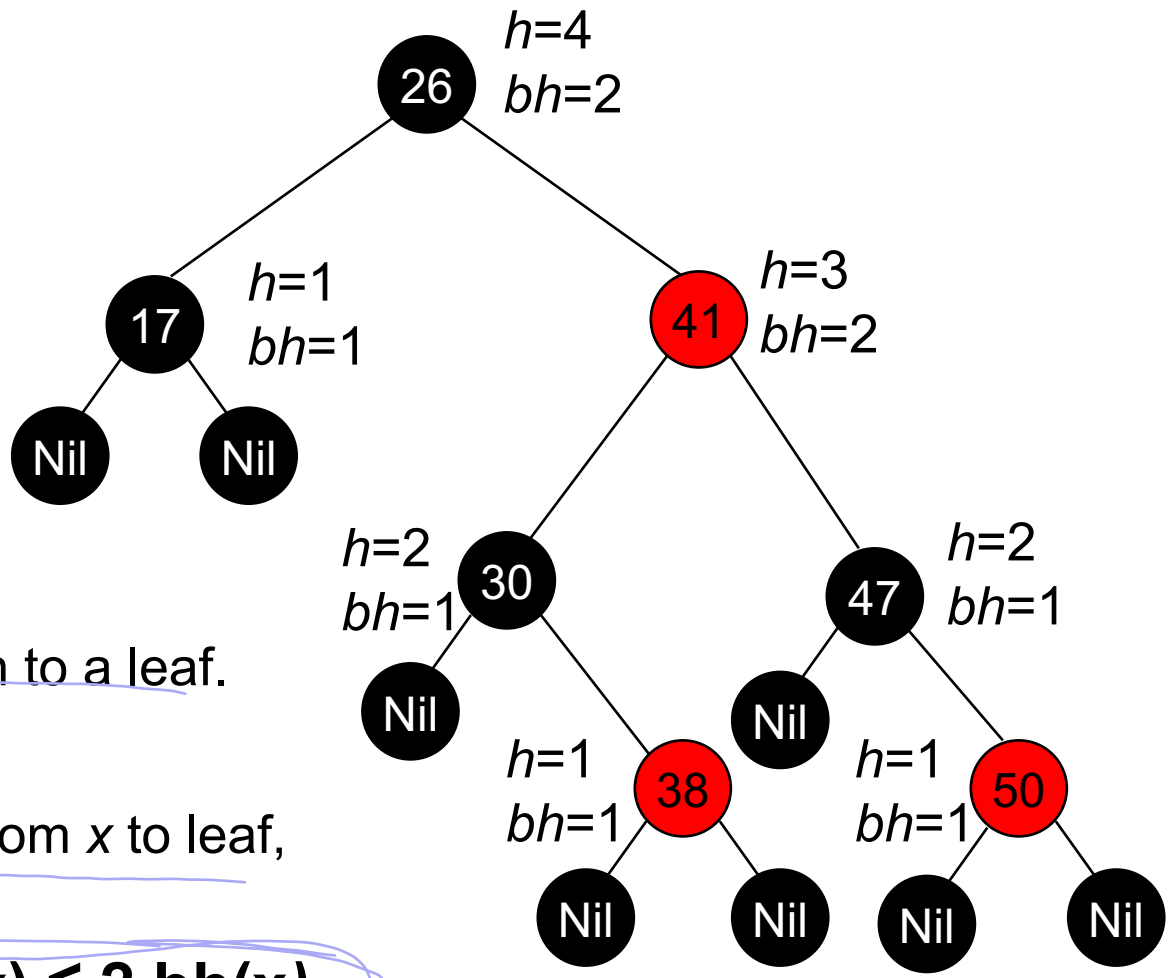
Note: every internal node has two children, even though *nil leaves are not usually shown*.

Red-Black trees - height

- Height of a node:
  - $h(x)$ = number of edges in the longest path to a leaf.
- Black-height of a node x, $bh(x)$:
  - $bh(x)$ = number of black nodes (including $nil[T]$) on the path from x to leaf, not counting x.
- Black-height of a red-black tree is the black-height of its root.
  - By RB Property 5, black height is well defined.

= all black-heights

Red-Black trees - height



- Height h(x):

  #edges in a longest path to a leaf.

- Black-height bh(x):

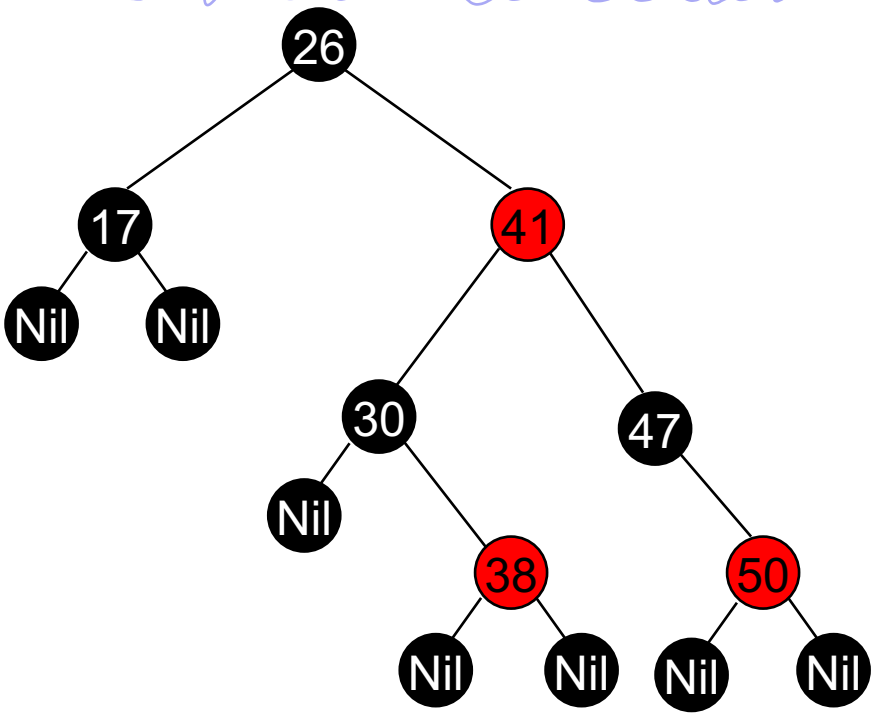
  # black nodes on path from *x* to leaf, *not counting x*.

- **Property: bh(x) ≤ h(x) ≤ 2 bh(x)** – allows for proof $h = O(\log n)$

Red-Black trees - height

Lemma 1: Any node x with height h(x) has a black-height bh(x) ≥ h(x)/2.

Proof: By RB property 4, ≤ h/2 nodes on the path from the node to a leaf are red. Hence ≥ h/2 are black. ■



Red-Black trees - height

Lemma 2: The subtree rooted at any node x contains $\geq 2^{bh(x)} - 1$ internal nodes.

Proof: By induction on height of x.

- **Base Case:** Height $h(x) = 0 \Rightarrow x$ is a leaf (Nil) $\Rightarrow$ $bh(x) = 0$. Subtree has $\geq 2^0 - 1 = 0$ nodes.
- **Induction Step:**
  - Each child of x has height $h(x) - 1$ and black-height either $bh(x)$ (child is red) or $bh(x) - 1$ (child is black).
  - By ind. hyp., each child has $\geq 2^{bh(x)-1} - 1$ internal nodes.
  - Subtree rooted at x has $\geq 2 \cdot (2^{bh(x)-1} - 1) + 1$ $= 2^{bh(x)} - 1$ internal nodes. ■

Red-Black trees - height

Lemma 1: Any node x with height h(x) has a black-height bh(x) ≥ h(x)/2.

Lemma 2: The subtree rooted at any node x has $\geq 2^{bh(x)}-1$ internal nodes.

Lemma 3: A red-black tree with n internal nodes has height at most $2 \lg(n+1)$.

Proof:

- By lemma 2, $n \geq 2^{bh} - 1$,
- By lemma 1, $bh \geq h/2$, thus $n \geq 2^{h/2} - 1$.
- $\Rightarrow \; h \leq 2 \lg(n + 1)$.

Outline

- Introduction.
- Operations.

RB trees - Insertion

- Insertion must preserve all red-black properties.
- Should an inserted node be colored Red? Black?
- Basic steps:
  - Use BST Tree-Insert to insert a node *x* into *T.*
    - Procedure **RB-Insert(*x*)**.
  - Color the node *x* red.
  - Fix the new tree by (1) re-coloring nodes, and (2) performing rotations to preserve RB tree property.
    - Procedure **RB-Insert-Fixup**.

AVL rotations (Lec 5)

RB trees - Insertion

RB-Insert(*T*, *z*)

```
1.   y ← nil[T]
2.   x ← root[T]
3.   while x ≠ nil[T]
4.       do y ← x
5.          if key[z] < key[x]
6.             then x ← left[x]
7.             else x ← right[x]
8.   p[z] ← y
9.   if y = nil[T]
10.      then root[T] ← z
11.  else if key[z] < key[y]
12.      then  left[y] ← z
13.  else right[y] ← z
```

RB-Insert(*T*, *z*) Contd.

```
14.  left[z] ← nil[T]
15.  right[z] ← nil[T]
16.  color[z] ← RED
17.  RB-Insert-Fixup (T, z)
```

Regular BST insert + color assignment + fixup.

RB trees – Insertion - Fixup

inserted red as default

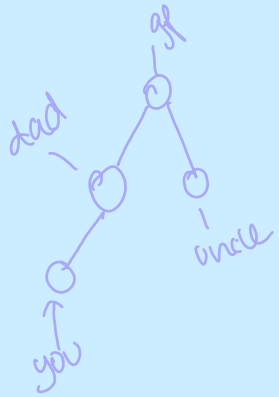
RB-Insert-Fixup (*T*, *z*)

```
1.  while color[p[z]] = RED
2.      do if p[z] = left[p[p[z]]]
3.          then y ← right[p[p[z]]]
4.               if color[y] = RED
5.                  then color[p[z]] ← BLACK   // Case 1
6.                       color[y] ← BLACK     // Case 1
7.                       color[p[p[z]]] ← RED // Case 1
8.                       z ← p[p[z]]          // Case 1
```

if parent is a left child of grandpa

identify uncle

color of uncle



uncle is red (exclusive)

RB trees – Insertion - Fixup

RB-Insert-Fixup(*T, z*) (Contd.)

```
9.          else if z = right[p[z]]  // color[y] ≠ RED
10.             then z ← p[z]                          // Case 2
11.                  LEFT-ROTATE(T, z)                 // Case 2
12.             color[p[z]] ← BLACK                    // Case 3
13.             color[p[p[z]]] ← RED                   // Case 3
14.             RIGHT-ROTATE(T, p[p[z]])               // Case 3
15.     else (if p[z] = right[p[p[z]]])(same as 10-14
16.             with "right" and "left" exchanged)
17. color[root[T ]] ← BLACK
```

if case 2 do this

RB trees – Insertion - Fixup

1. To determine what violations of the red-black properties are introduced in RB-Insert when node *z* is inserted and colored red.
2. To understand each of the three cases within the *while* loop's body
3. To understand the overall goal of the while loop in lines 1-15.

fixing up
to reach conditions
of RB tree

RB trees – Insertion - Fixup

1. To determine what violations of the red-black properties are introduced in RB-Insert when node *z* is inserted and colored red.

   1. Every node is either red or black.

   2. The root is black.

   - Violated if z is the root.

   when tree is empty before insertion easy to fix

   3. All leaves (*nil*) are black.

   insert new node then add its nil

   4. If a node is red, then its children are black (i.e. no 2 consecutive red nodes).

   - Violated if z's parent is red

   Hell to fix

   5. For each node, all paths from the node to descendant leaves contain the same number of black nodes (i.e. same black height).

   inserting a red does not change black height

   - node z replaces the (black) sentinel, and node z is red with sentinel children.

RB trees – Insertion - Fixup

1. To determine what violations of the red-black properties are introduced in RB-Insert when node *z* is inserted and colored red. (last slide)
2. To understand each of the three cases within the *while* loop's body
3. To understand the overall goal of the while loop in lines 1-15.

RB trees – Insertion - Fixup

2. To understand each of the three cases within the *while* loop's body.

- We actually need to consider six cases in the while loop, but three of them are symmetric to the other three.
  - Line2: Three cases if the parent of z is a left (right) child of the grandparent of z.
- The grand parent of z exists.
  - If the parent of z is the root (i.e., the grand parent will not exist), then the parent of z is black, but notice that we enter a loop iteration only if the parent of z is red, then we know that the parent of z cannot be the root. Hence, the grand parent of z exists.

RB-Insert-Fixup (*T, z*)

1. **while** *color*[*p*[*z*]] = RED
2. **do if** *p*[*z*] = *left*[*p*[*p*[*z*]]]
3. ……

RB trees – Insertion - Fixup

2. To understand each of the three cases within the *while* loop's body.

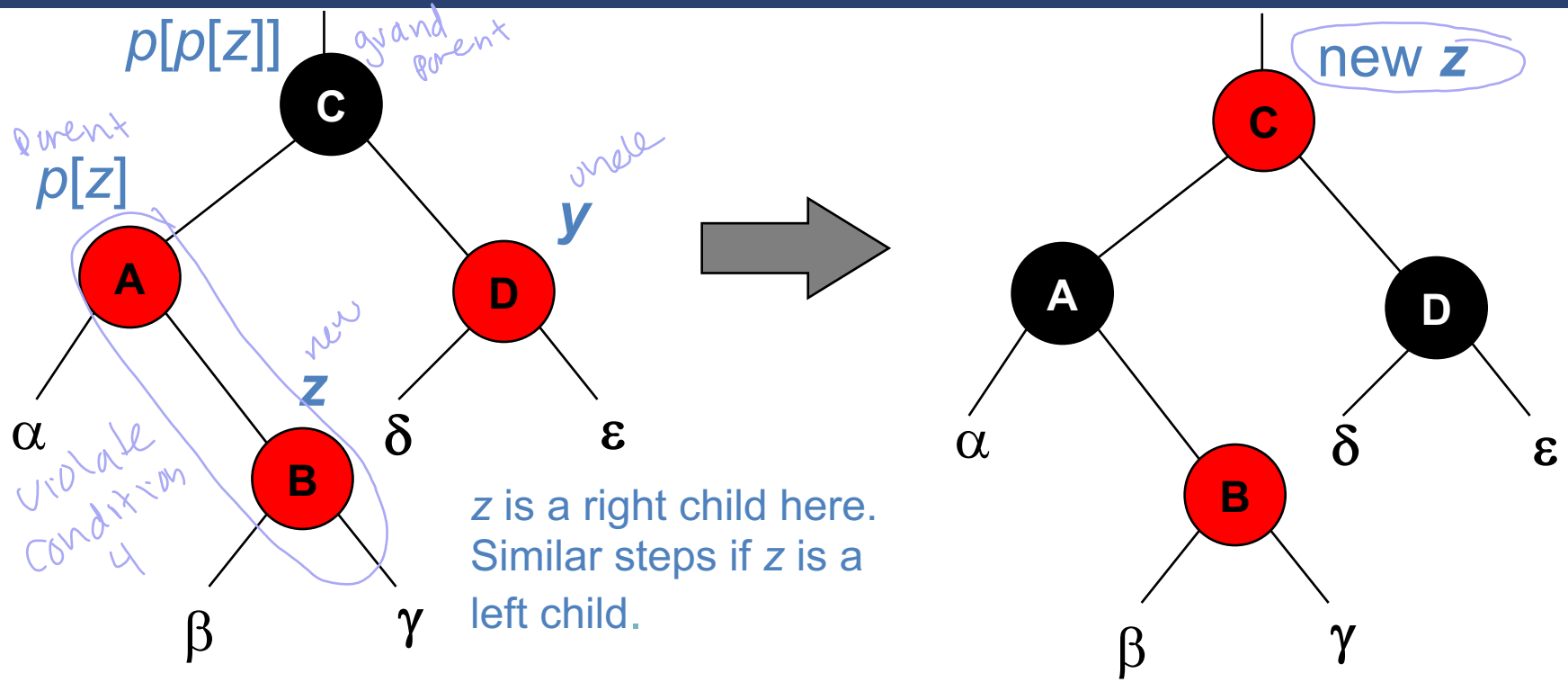
- In all three cases, the grand-parent of *z* is black. (because parent is red)
  - Since the parent of z is red (line 1), and property 4 is violated only between z and its parent (p[z]).
- We distinguish case 1 from cases 2 and 3 by the color of z's parent's sibling, or "uncle".
  - Line3: Makes *y* point to *z's* uncle.
  - Line4: Test *y's* color. If *y* is red, then we execute case 1. Otherwise, control passes to cases 2 and 3.

y black

RB-Insert-Fixup (*T, z*)

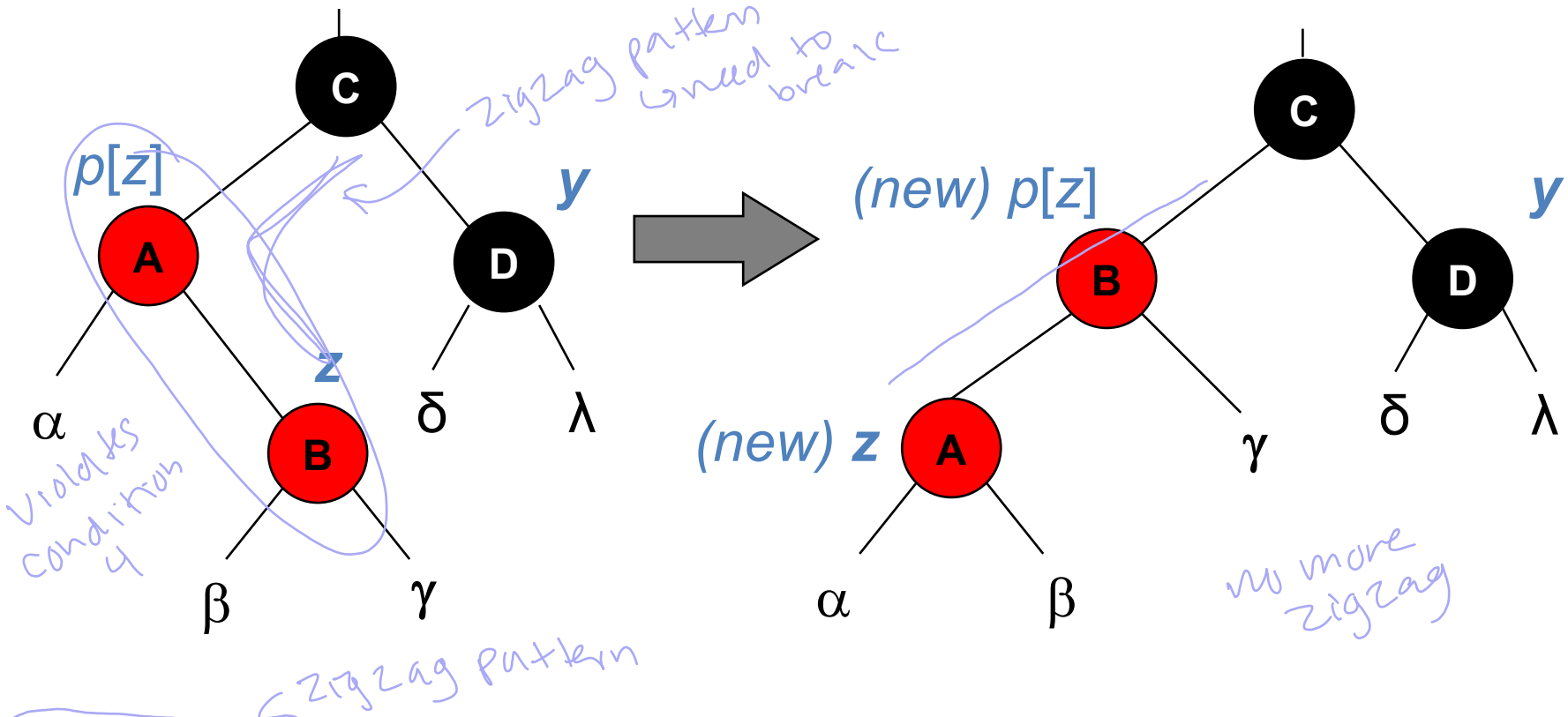
```
1.    while color[p[z]] = RED
2.        do if p[z] = left[p[p[z]]]
3.              then y ← right[p[p[z]]]
4.                   if color[y] = RED
```

Insertion – Case1 – Uncle y is red



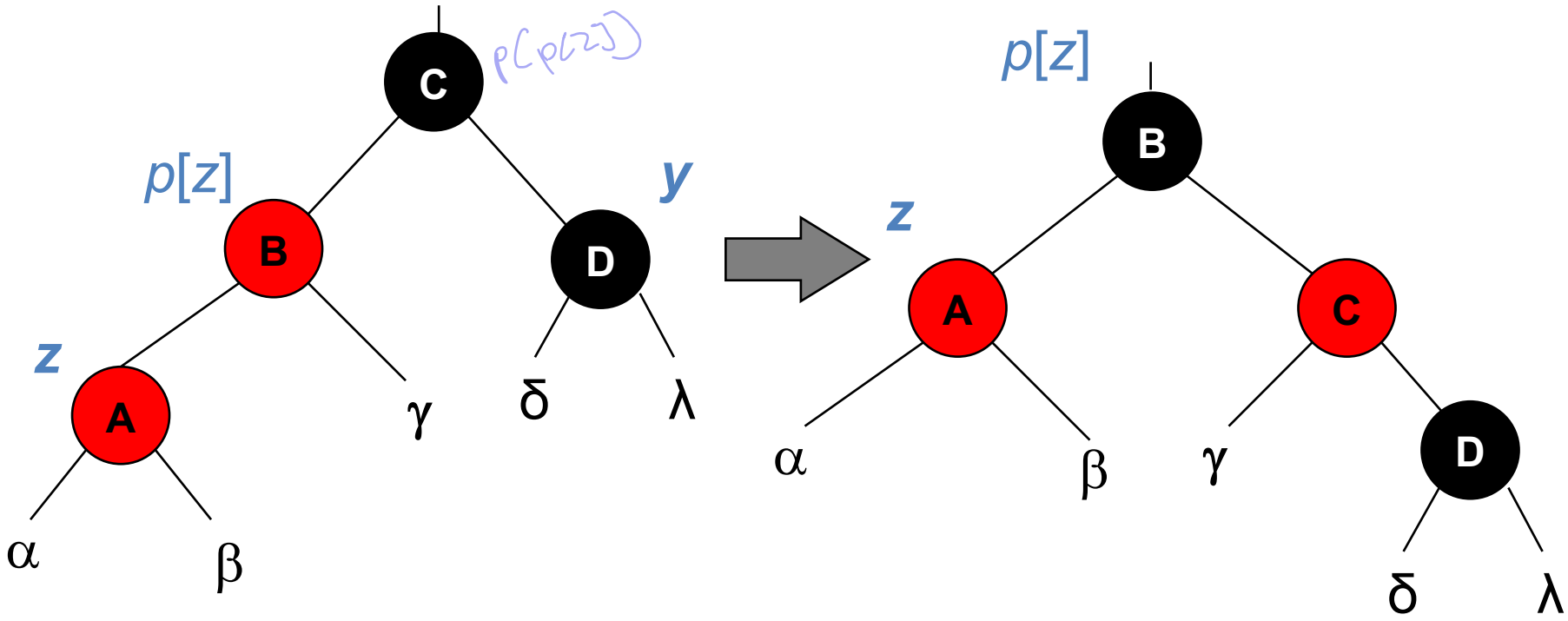
- $p[p[z]]$ (z's grandparent) must be black, since z and $p[z]$ are both red and there are no other violations of property 4.
- Make $p[z]$ and y black $\Rightarrow$ now z and $p[z]$ are not both red. But property 5 might now be violated.
- Make $p[p[z]]$ red $\Rightarrow$ restores property 5.
- The next iteration has $p[p[z]]$ as the new z (i.e., z moves up 2 levels).

Case2 – y is black, z is a right child



- Left rotate around $p[z]$, $p[z]$ and z switch roles ⇒ now z is a left child, and both z and $p[z]$ are red.
- Takes us immediately to case 3.

Case3 – y is black, z is a left child



- Make $p[z]$ black and $p[p[z]]$ red.
- Then right rotate on $p[p[z]]$ (in order to maintain property 4).
- No longer have 2 reds in a row.
- $p[z]$ is now black $\Rightarrow$ no more iterations.

RB trees – Insertion - Fixup

1. To determine what violations of the red-black properties are introduced in RB-Insert when node *z* is inserted and colored red.
2. To understand each of the three cases within the *while* loop's body
3. To understand the overall goal of the while loop in lines 1-15.

RB trees – Insertion - Fixup

3. To understand the overall goal of the while loop in lines 1-15.
   - The loop maintains the following invariant at the start of each iteration
   1. If the tree violates any of the red-black properties, then:
      - The violation is either property 2 or property 4.
      - It violates at most one of them.

loop invariant

violates only prop 2 or prop 4

Recall that we need to show that a loop invariant is true prior to the first iteration of the loop, that each iteration maintains the loop invariant, and that the loop invariant gives us a useful property at loop termination.

RB trees – Insertion - Fixup

3. To understand the overall goal of the while loop in lines 1-15.

- **Initialization:** Prior to the first iteration of the loop, we started with a red-black tree with no violations, and we added a red node z.
  1. If the tree violates any of the red-black properties, then the violation is at most one of property 2 or property 4:

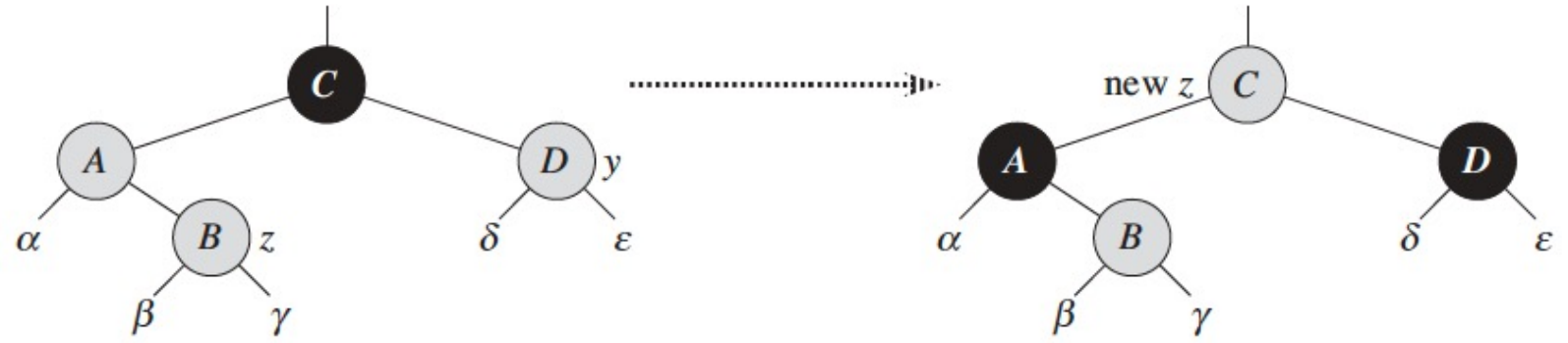
  - If property 2 is violated: z must be the red root. Both children of z are sentinels, which are black. Then, the tree does not also violate property 4.
  - If property 4 is violated: z and its parent must be red. Given that both children of z are sentinels, which are black and that the tree had no other violations prior to z being added, the parent of z can not be the root. Then, the tree does not also violate property 2.

RB trees – Insertion - Fixup

3. To understand the overall goal of the while loop in lines 1-15.

- **Maintenance:** Case 1: z's uncle y is red.

1. **If the tree violates any of the red-black properties, then the violation is at most one of property 2 or property 4:**
   - Case 1 maintains property 5, and it does not introduce a violation of properties 1 or 3.
   - If the new node z (i.e., the grand parent of the current z) is the root at the start of the next iteration, then case 1 corrected the lone violation of property 4. since the new node z is red and it is the root, property 2 becomes the only one that is violated.
   - If the new node z (i.e., the grand parent of the current z) is not the root at the start of the next iteration, then case 1 has not created a violation of property 2. The only possible violation in the next iteration will be to property 4.

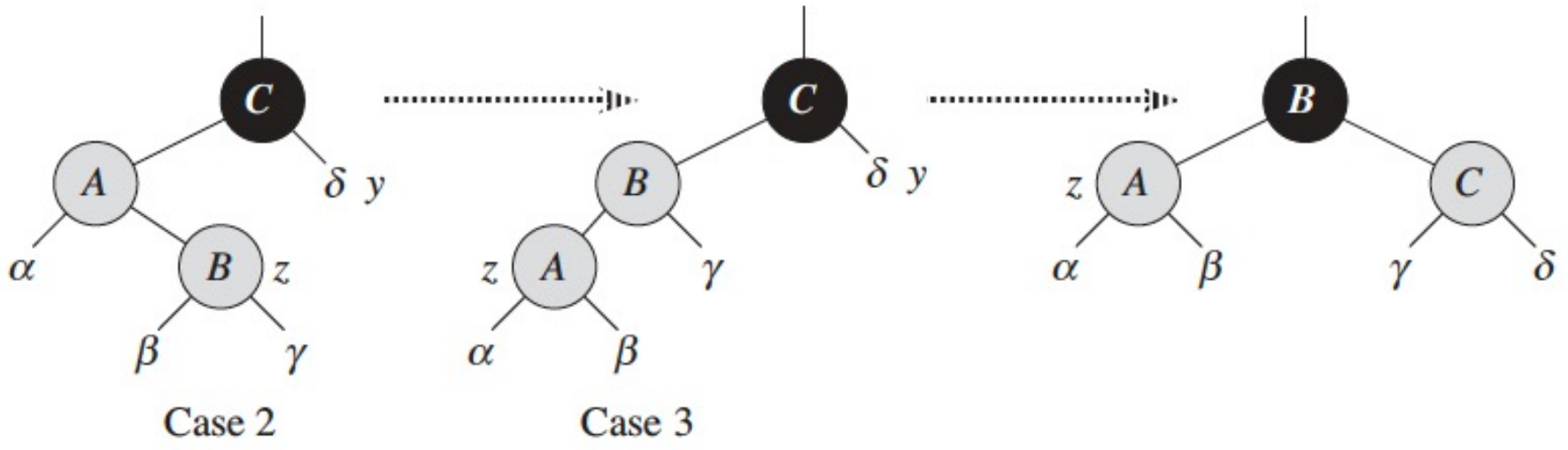


RB trees – Insertion - Fixup

3. To understand the overall goal of the while loop in lines 1-15.

- **Maintenance:** Case 2 and 3: z's uncle y is black.

1. **If the tree violates any of the red-black properties, then the violation is at most one of property 2 or property 4:**
   - Case 2 and 3 maintain properties 1, 3 and 5.
   - z is not the root in cases 2 and 3, then there is no violation of property 2.
   - Cases 2 and 3 correct the lone violation of property 4, and they do not introduce another violation.



RB trees – Insertion - Fixup

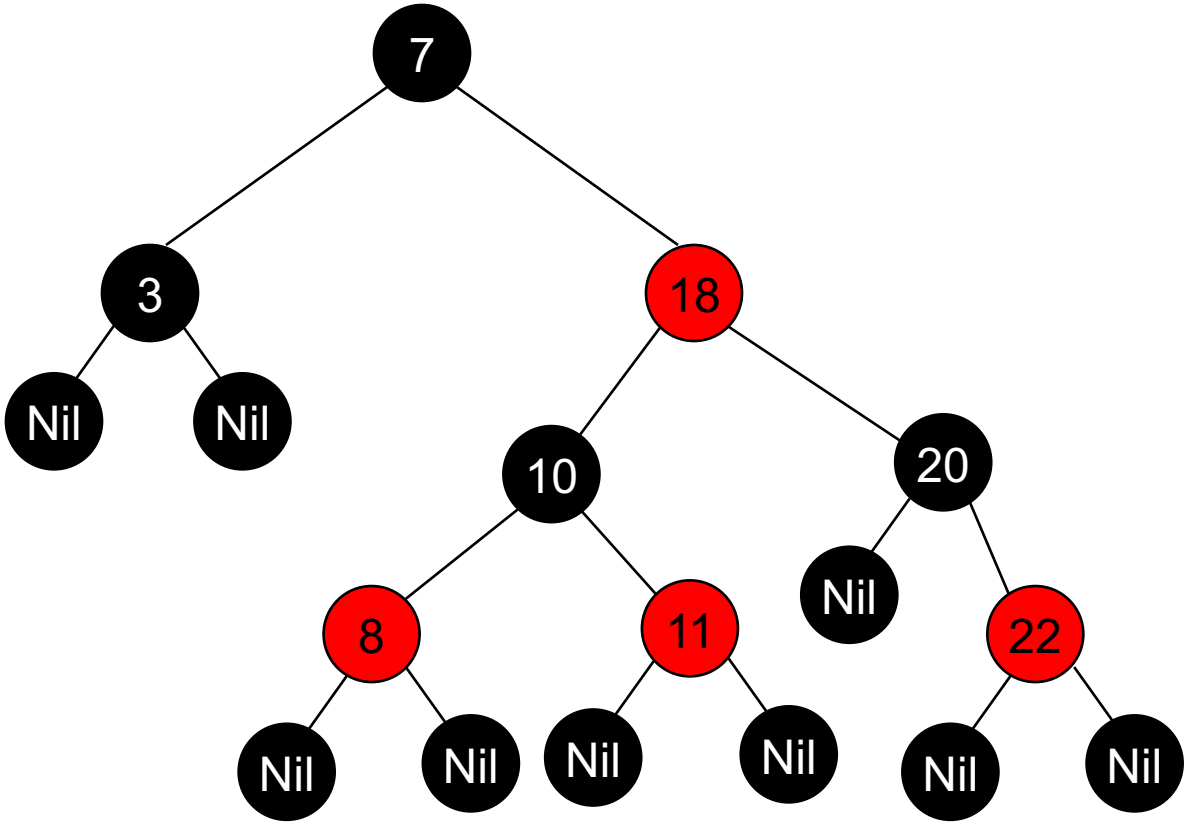
3. To understand the overall goal of the while loop in lines 1-15.

- **Termination:** When the loop terminates, it does so because p[z] is black. Thus, the tree does not violate property 4 at loop termination. By the loop invariant, the only property that might fail to hold is property 2. Line 17 restores this property.

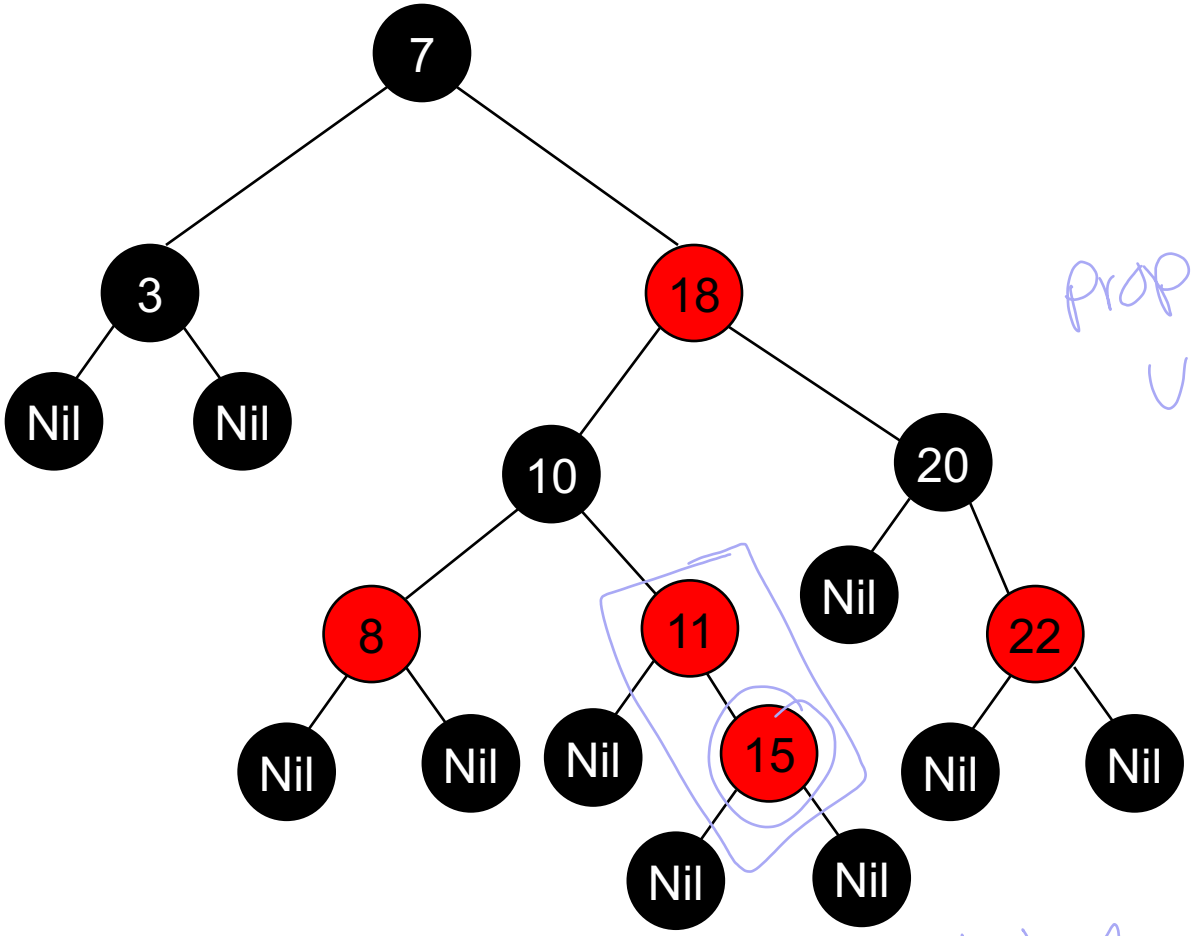
RB-Insert-Fixup(*T, z*) (Contd.)

```
1.    while color[p[z]] = RED
2.        do if p[z] = left[p[p[z]]]
3.    ……….
15.       else (if p[z] = right[p[p[z]]])(same as 10-14
16.                  with "right" and "left" exchanged)
17. color[root[T ]] ← BLACK
```

Insert RB tree - Example

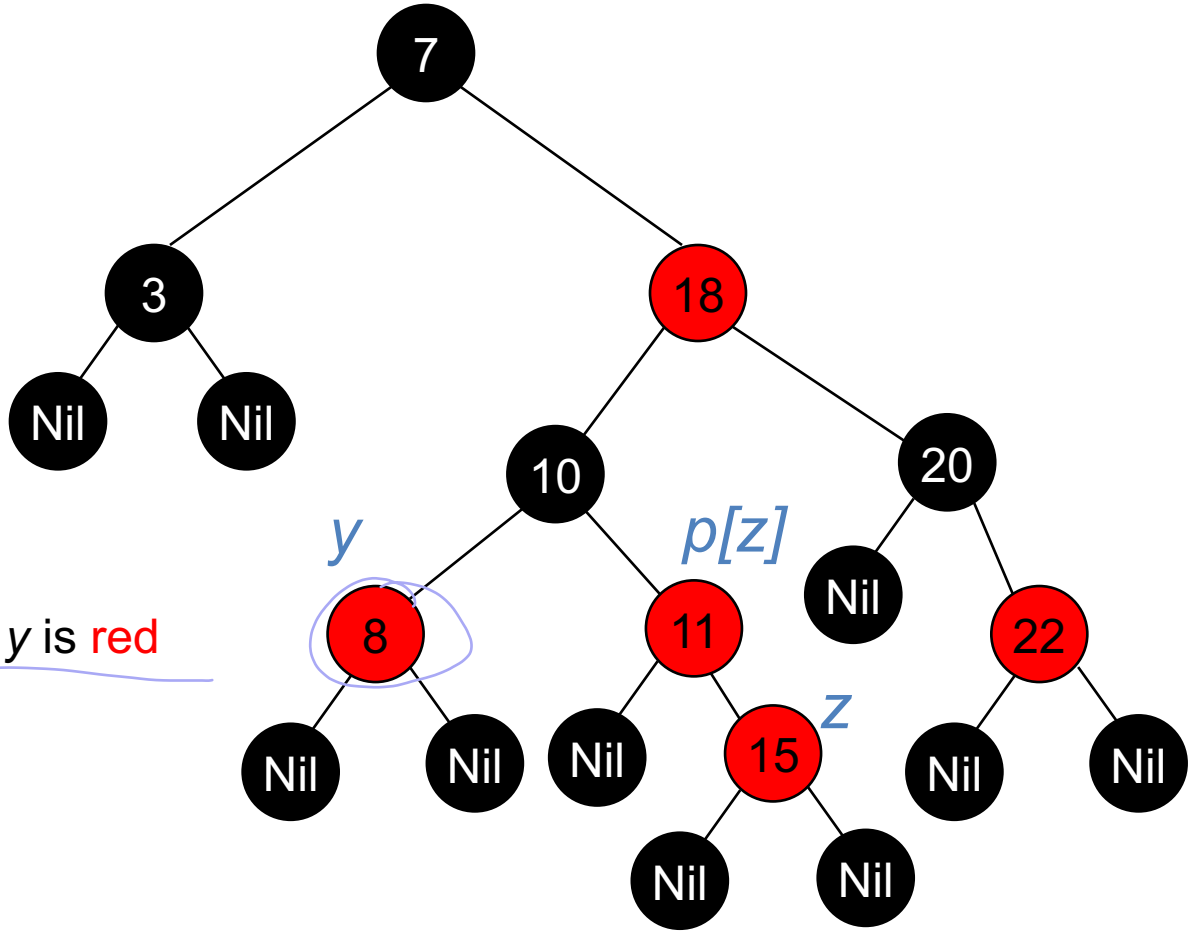


Insert RB tree - Example

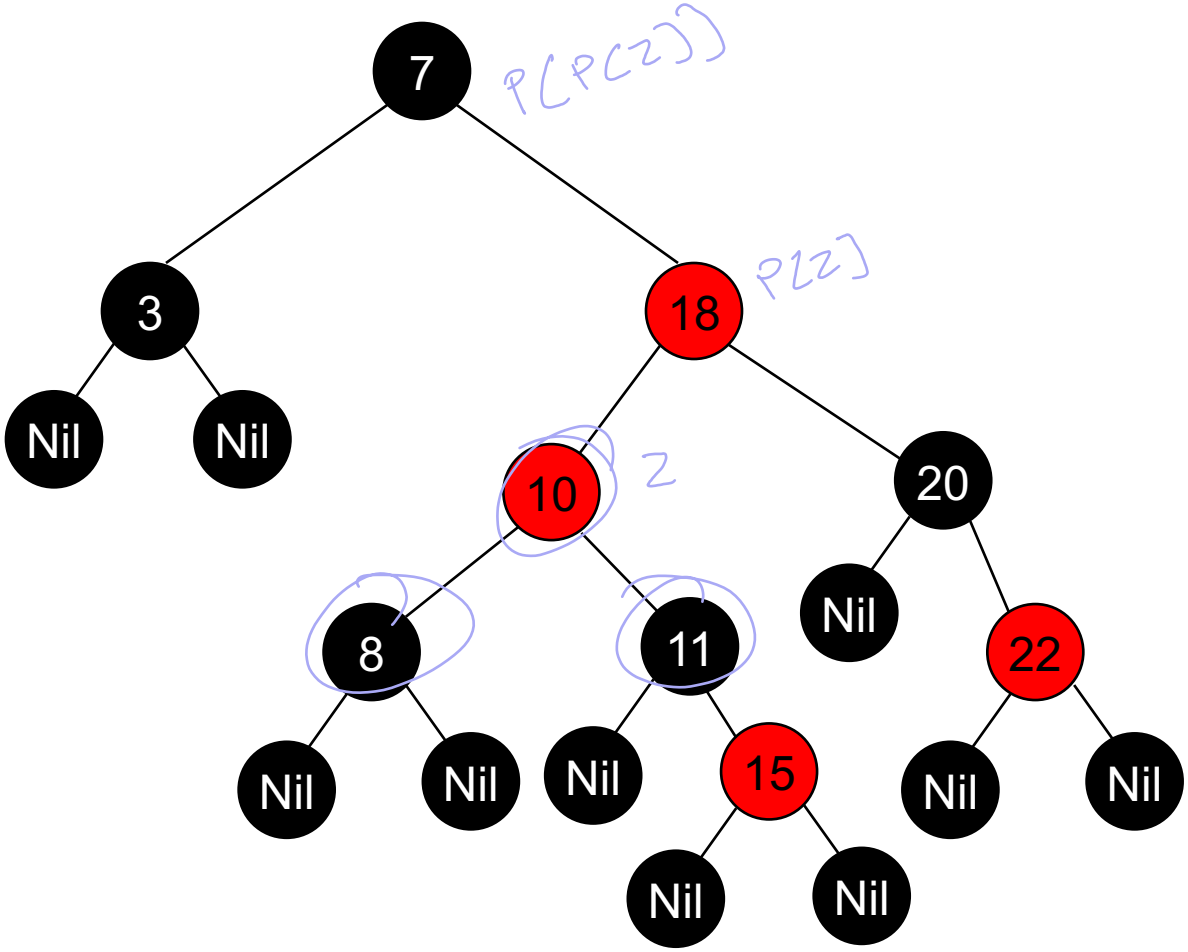


Insert RB tree - Example

Case 1 – uncle *y* is red

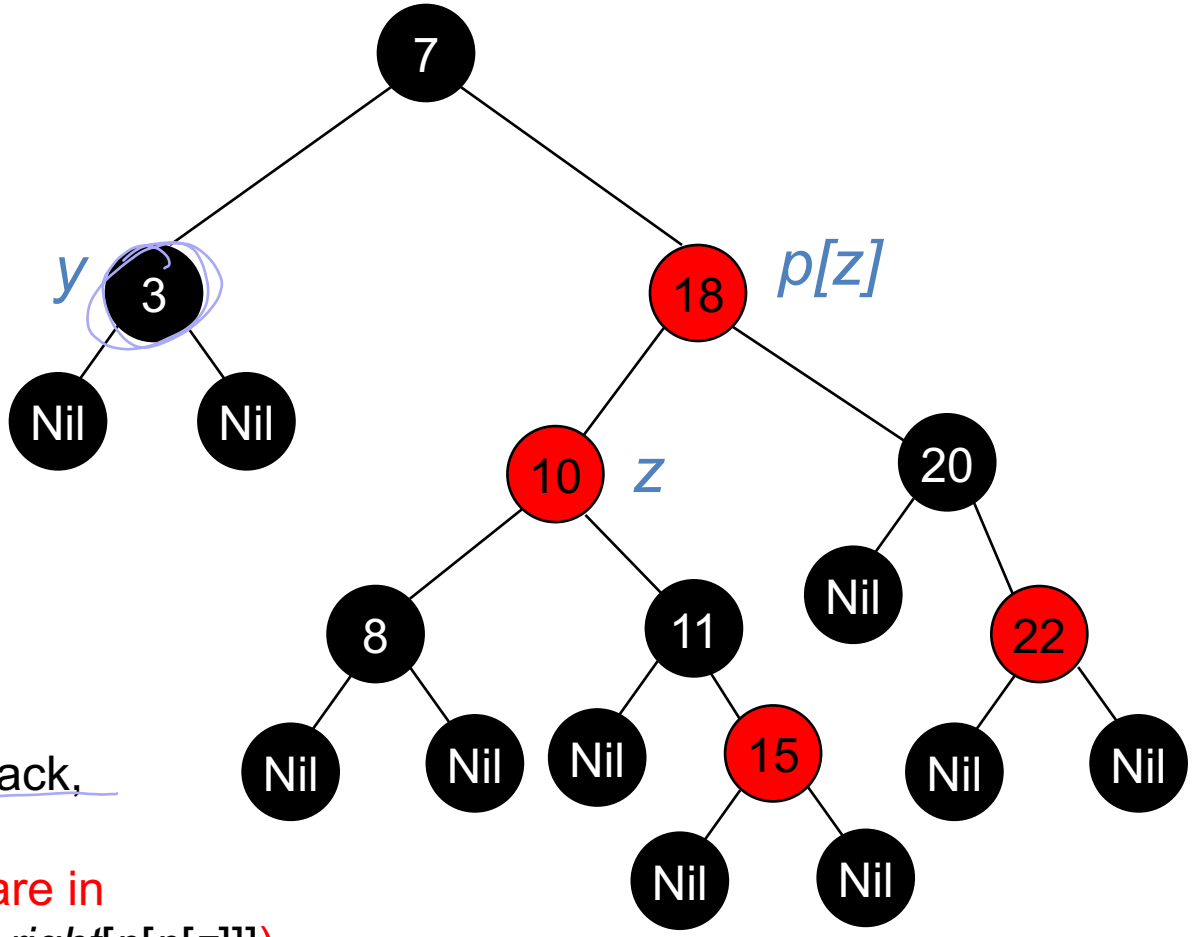


Insert RB tree - Example



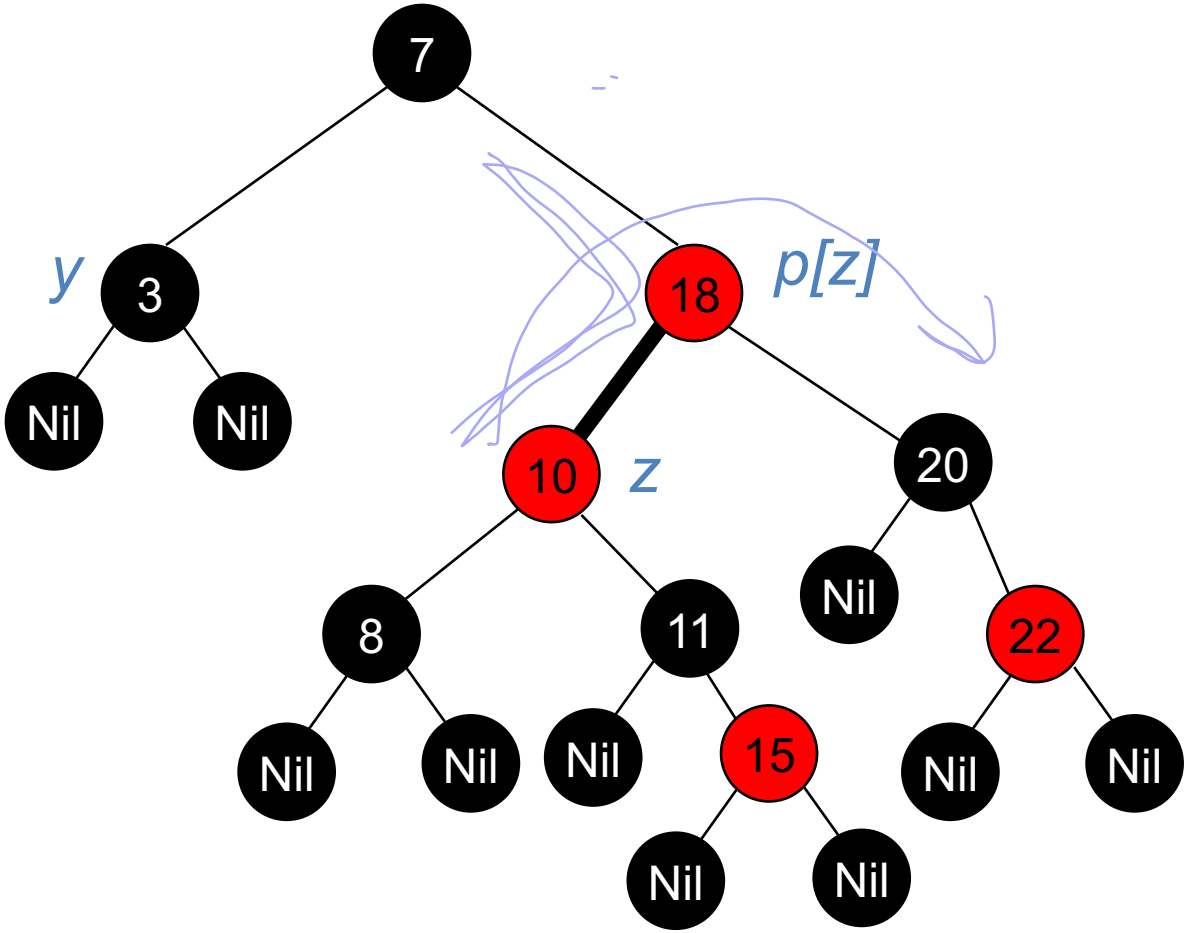
Recolor 10, 8 &11

Insert RB tree - Example



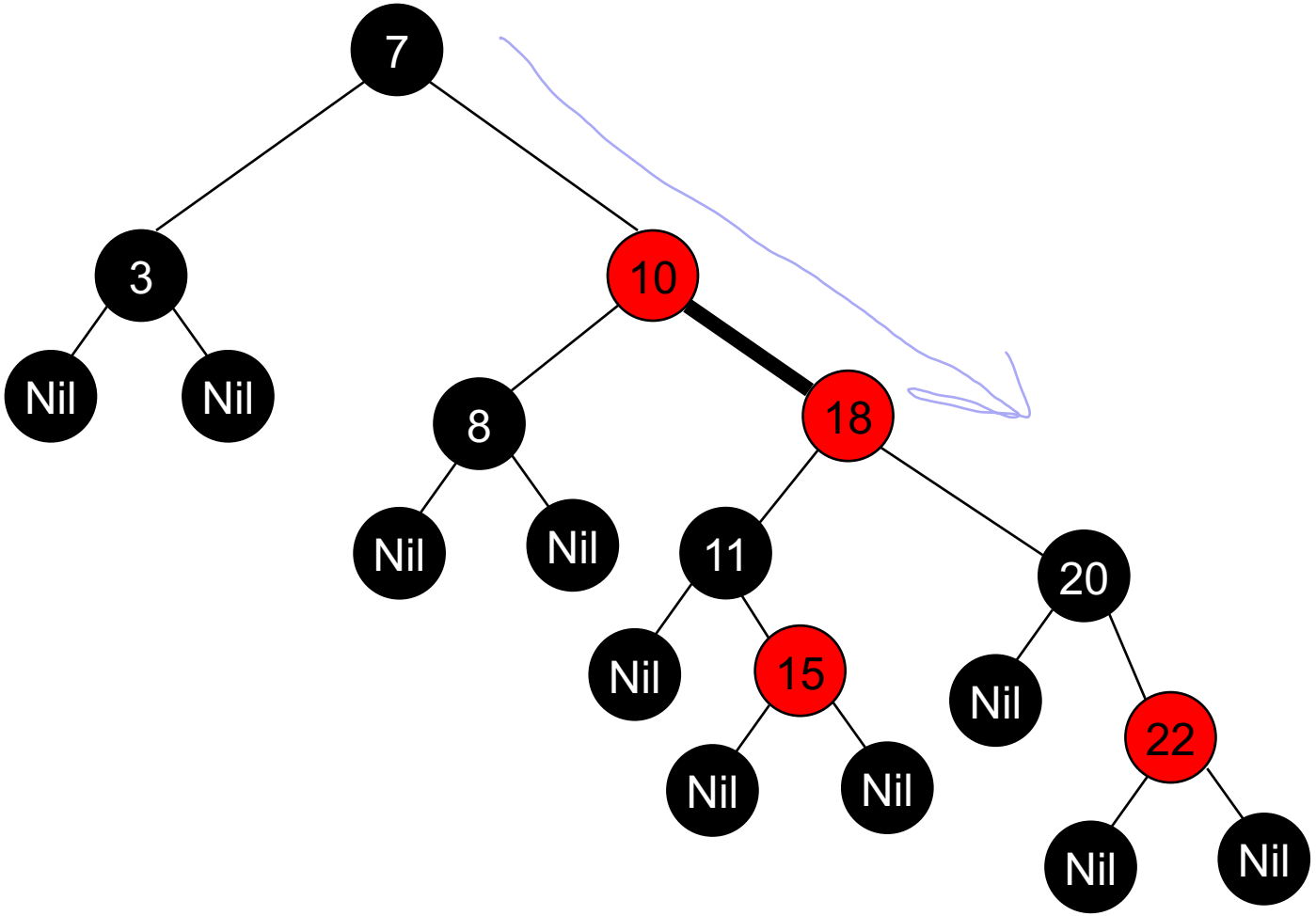
Case 2 – *y* is black,
z is a left child.
Notice that we are in
line 15 (if *p*[*z*] = *right*[*p*[*p*[*z*]]]).

Insert RB tree - Example



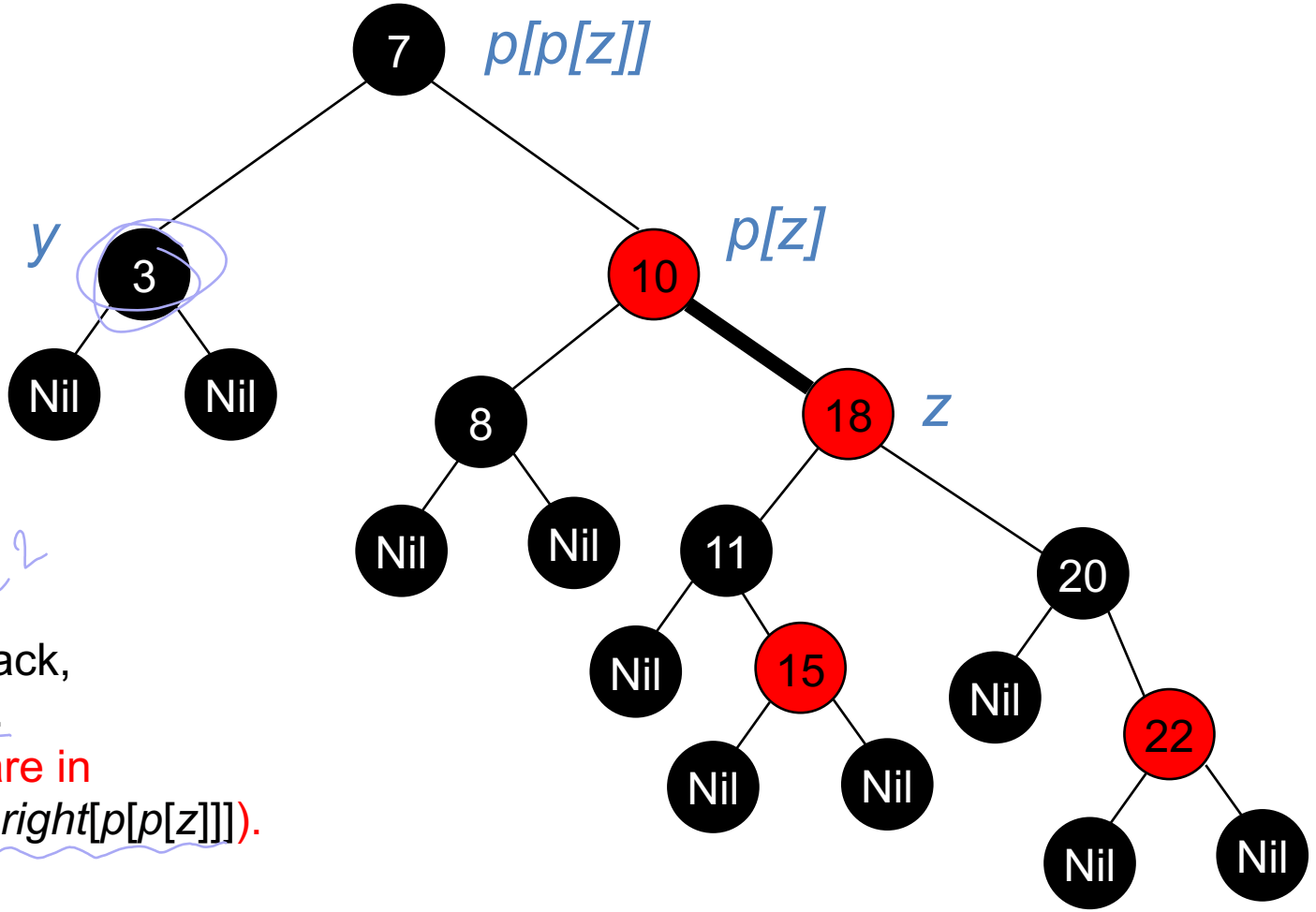
Right rotate at 18 p[z]

Insert RB tree - Example



Parent & child with conflict are now aligned with the root.

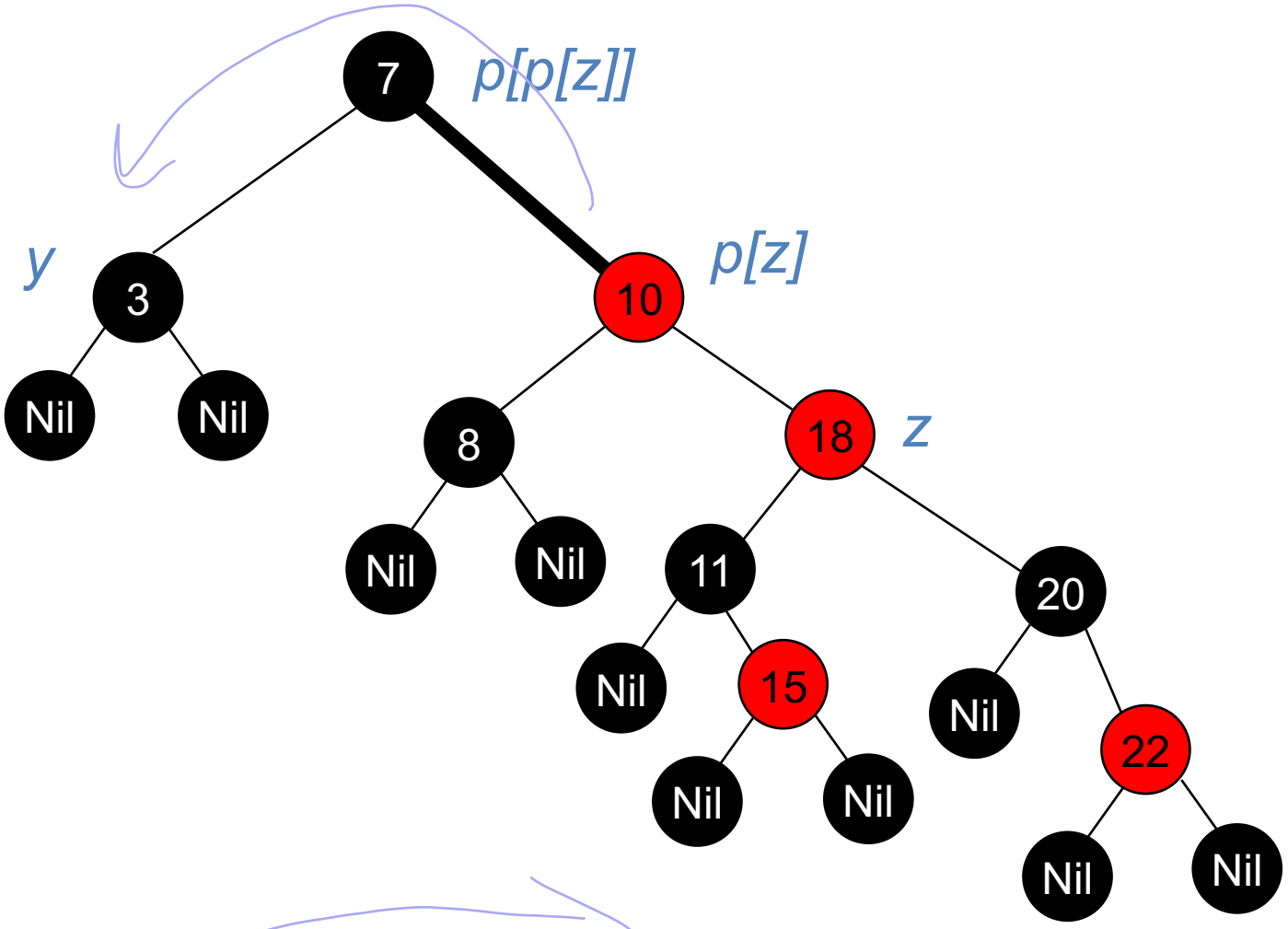
Insert RB tree - Example



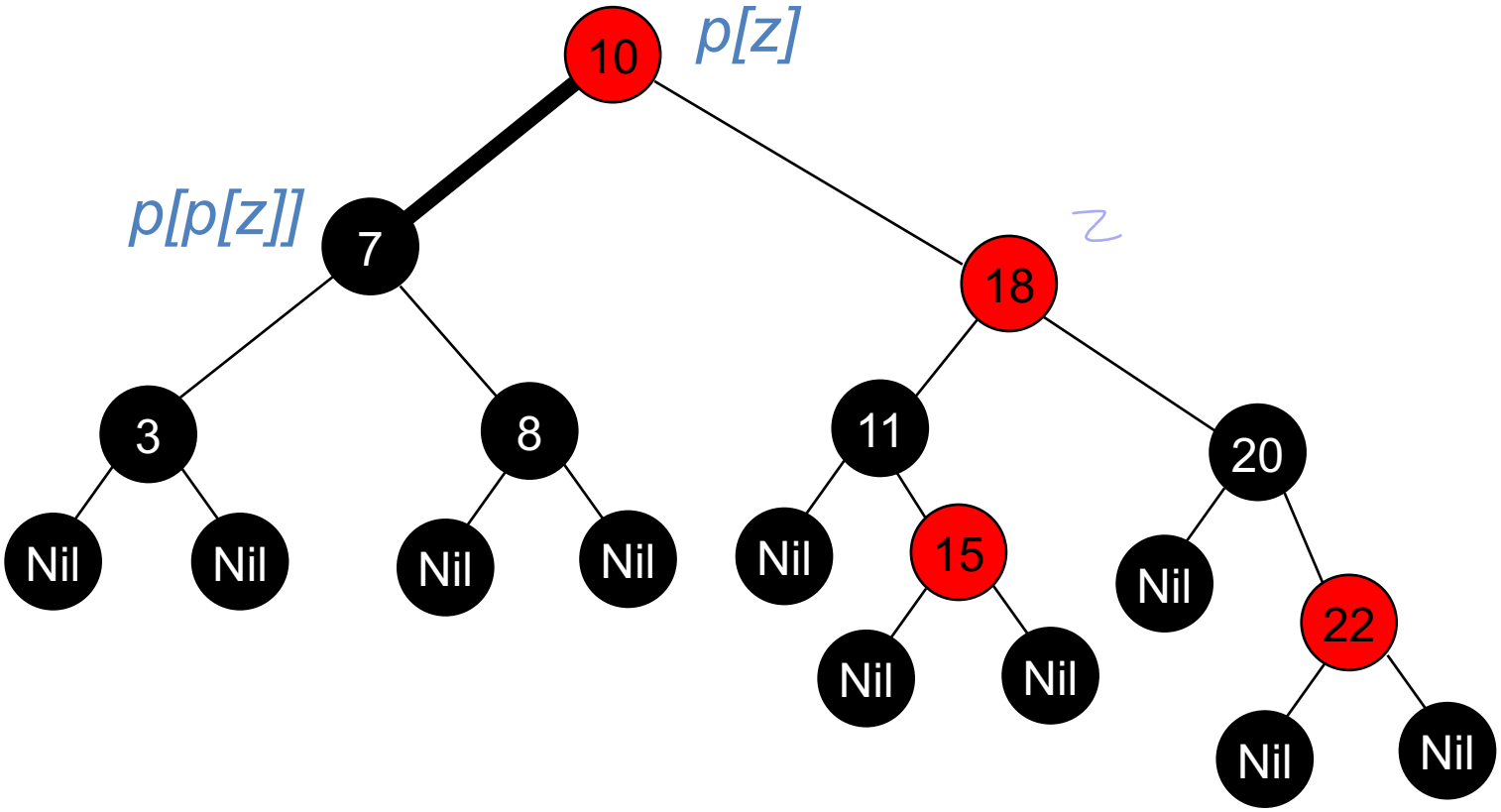
Case 3 – *y* is black,
z is a right child.
Notice that we are in
line 15 (if *p*[*z*] = *right*[*p*[*p*[*z*]]]).

Parent & child with conflict are now aligned with the root.

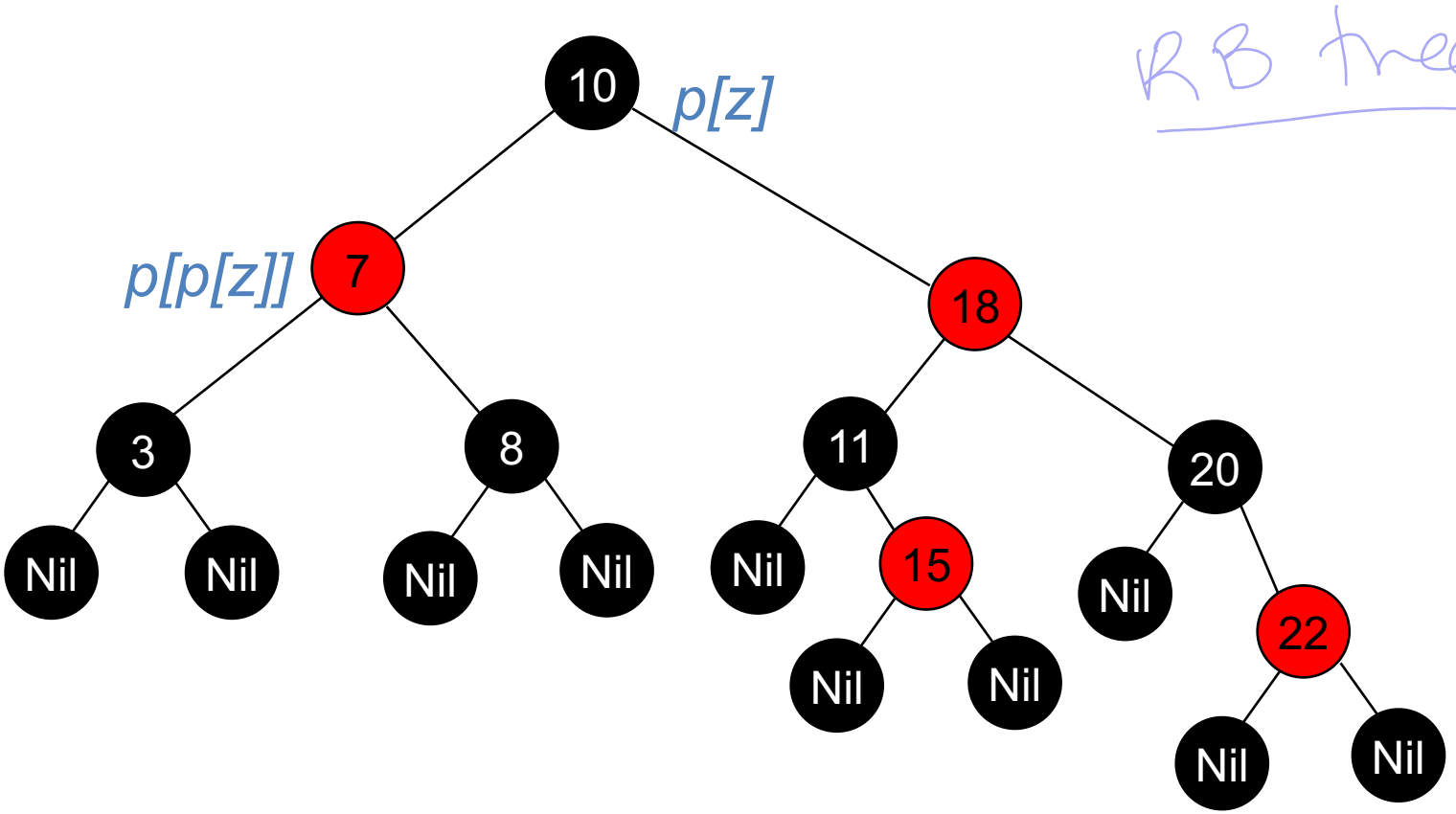
Insert RB tree - Example



Insert RB tree - Example



Insert RB tree - Example



Recolor 10 p[z] & 7 p[p[z]] (root must be black!)

Insert RB tree - Complexity

- $O(\lg n)$ time to get through RB-Insert up to the call of RB-Insert-Fixup.
- Within RB-Insert-Fixup:
  - Each iteration takes $O(1)$ time.
  - The while loop repeats only if case 1 occurs.
    - Each iteration but the last moves z up 2 levels.
    - $O(\lg n)$ levels $\Rightarrow$ $O(\lg n)$ time.
    - Thus, insertion in a red-black tree takes $O(\lg n)$ time.
  - Note: there are at most 2 rotations overall.
    - Since the while loop terminates if case 2 or case 3 is executed.

$O(\lg n)$

AVL vs Red-Black Trees

- AVL trees are more strictly balanced ⇒ faster search
- Red Black Trees have less constraints and insert/remove operations require less rotations ⇒ faster insertion and removal
- AVL trees store balance factors or heights with each node
- Red Black Tree requires only 1 bit of information per node

AVL

faster search

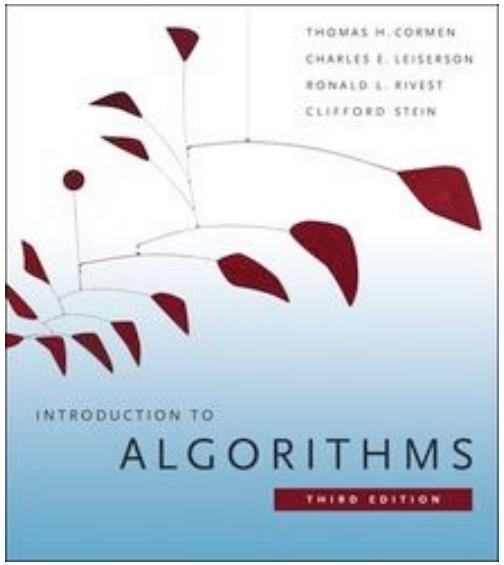
store balance factor or heights with each node

RB

faster insertion
faster removal
requires only 1 bit per node

Further readings

[CLRS2009] Cormen, Leiserson, Rivest, & Stein, *Introduction to Algorithms*. (available as E-book)



See Chapter 13 for the complete proofs & deletion

Outline

- Introduction.
- Operations.

Outline – Course so far

First 7 lectures

Next 8 lectures

Modified image initially taken from Programmer Humor

Algorithmic Paradigms

- General approaches to the construction of *efficient* solutions to problems.
- Such methods are of interest because:
  - They provide templates suited to solving a broad range of diverse problems.
  - They can be translated into common control and data structures provided by most high-level languages.
  - The temporal and spatial requirements of the algorithms which result can be precisely analyzed.
- Although more than one technique may be applicable to a specific problem, it is often the case that an algorithm constructed by one approach is clearly superior to equivalent solutions built using alternative techniques.

Algorithmic Paradigms

- Complete Search.
- Divide and Conquer.
- Dynamic Programming.
- Greedy
- Conclusions


time for questions