SFU CMPT-307 2008-2 Lecture: Week 10

Ján Maňuch

E-mail: jmanuch@sfu.ca

Lecture on July 15, 2008, 5.30pm-8.20pm

BST summary

BST property: for each node v with key k,

- nodes in left subtree have keys $\leq k$
- nodes in right subtree have keys $\geq k$

BST operations: Search, Minimum, Maximum, Predecessor, Successor, Insert, Delete

all take $O(h)$ time, where h is the height of BST

OK if BST is **balanced**, then $h = O(\log n)$

Bad but if BST is **degenerated**, then $h = \Omega(n)$

How to take care that BST does **not** degenerate when inserting or deleting?

Many approaches, we're going to see **red-black trees**

Red-black trees

Simple change: a red-black tree is an ordinary BST with **one extra bit** of storage per node: its **color**, red or black

Constraining how nodes can be colored makes the tree approximately balanced

Assumption: if either parent, left child, or right child does not exist, pointer to NULL

Regard NULLs as external nodes (leaves) of tree, thus all normal nodes are internal

Red-black property: A BST is a red-black tree if

(RB1) every node is either red or black

(RB2) the root is black

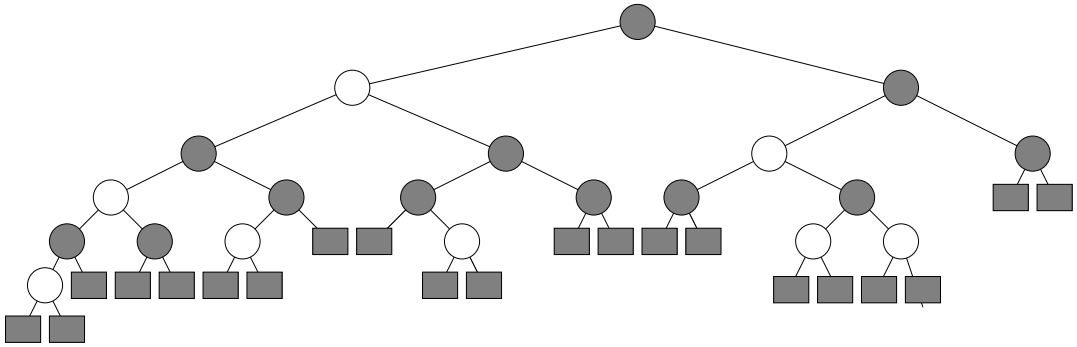
(RB3) every leaf (NULL) is black

(RB4) red nodes have black children

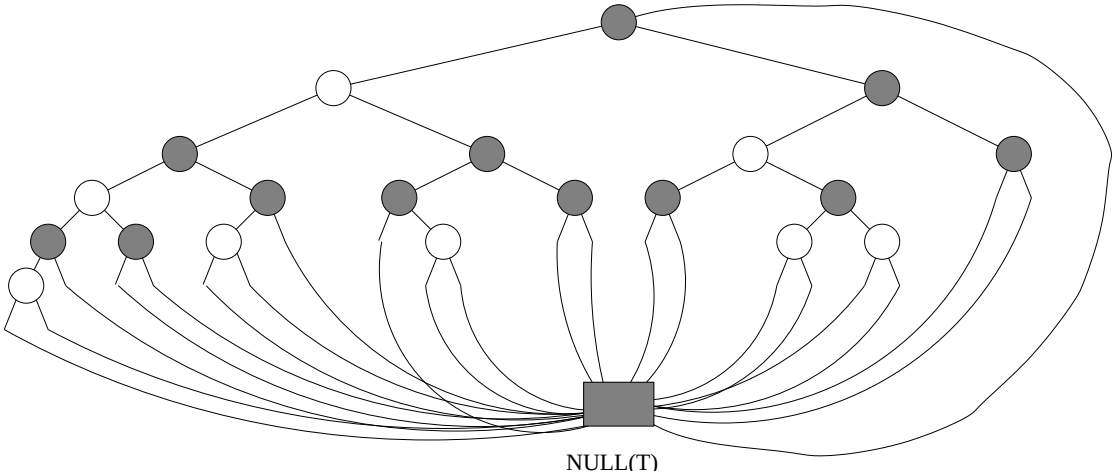
(RB5) for all nodes, all paths from node to descendant leaves contain the same number of black nodes

Representations

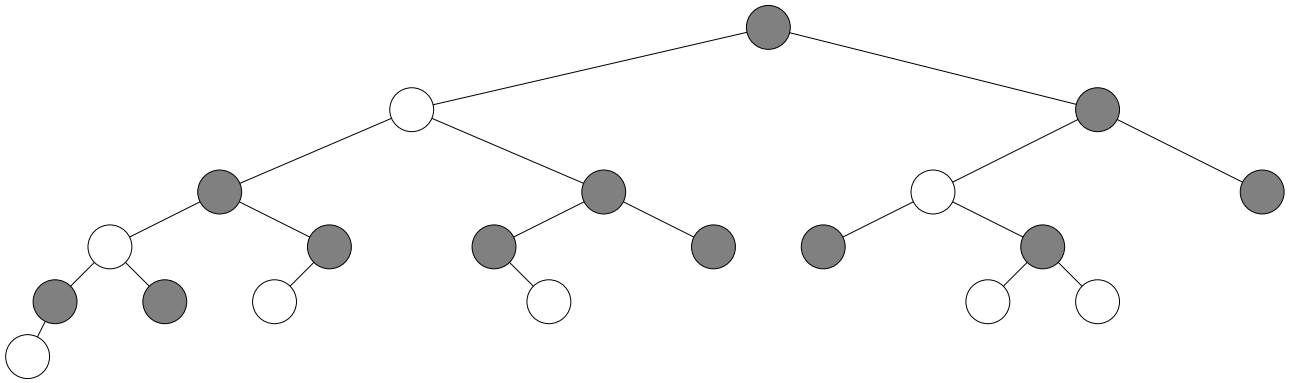
1. *red-black tree according to definition* (useful when dealing with boundary conditions)



2. *single sentinel* — replace NULLs with single "sentinel" NULL(T) (saves memory)



3. *normal drawing style* (no NULL leaves or sentinel are shown), but sentinel **is** still there (just not shown)



Definition: the **black-height** of node v, $\mathrm{bh}(v)$, is the number of black nodes on any (!) path from v to a leaf, **not** including v

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

Proof. First we show that a subtree rooted at any node x contains at least $2^{\text{bh}(x)} - 1$ internal nodes

By induction on height of x

Base case (height=0)

x is a leaf (NULL(T)), subtree rooted at x contains $2^{\text{bh}(x)} - 1 = 2^0 - 1 = 1 - 1 = 0$ internal nodes

Inductive step Suppose x has positive height and two children. Each child has black-height of either bh(x) (if red) or bh$(x) - 1$ (if black)

Height of children is less than height of x, thus can apply hypothesis: subtrees rooted in children contain at least $2^{\text{bh}(x)-1} - 1$ internal nodes each

Thus subtree rooted in x contains at least $2 \cdot (2^{\text{bh}(x)-1} - 1) + 1 = 2^{\text{bh}(x)} - 2 + 1 = 2^{\text{bh}(x)} - 1$ internal nodes

Now let h be height of tree

Property (RB4): at least half the nodes on any simple path from root to a leaf, not including root, must be black

Thus black-height of root is at least $h/2$, and

$$\begin{aligned} & n \geq 2^{\mathrm{bh}(root)-1} \geq 2^{h/2} - 1 \\ \Leftrightarrow \quad & n + 1 \geq 2^{h/2} \\ \Leftrightarrow \quad & \log(n+1) \geq h/2 \\ \Leftrightarrow \quad & h \leq 2\log(n+1) \end{aligned}$$

Very nice property, since now trivially Search, Minimum, Maximum, Successor, Predecessor work in $O(\log n)$ time!

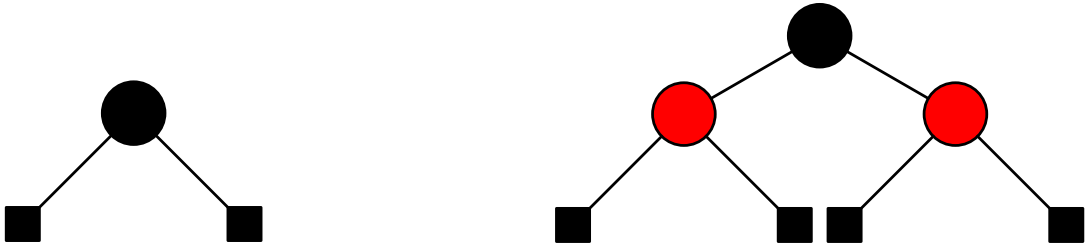
But what about Insert, Delete?

How to modify them so that they **maintain** the red-black properties?

Assignment Problem 10.1. (deadline: July 22, 5:30pm)
What is the largest possible number of internal nodes in a red-black tree with black-height k? What is the smallest possible number?

Example:

If $k = 1$, the smallest possible number of internal nodes is 1 and the largest possible number is 3:

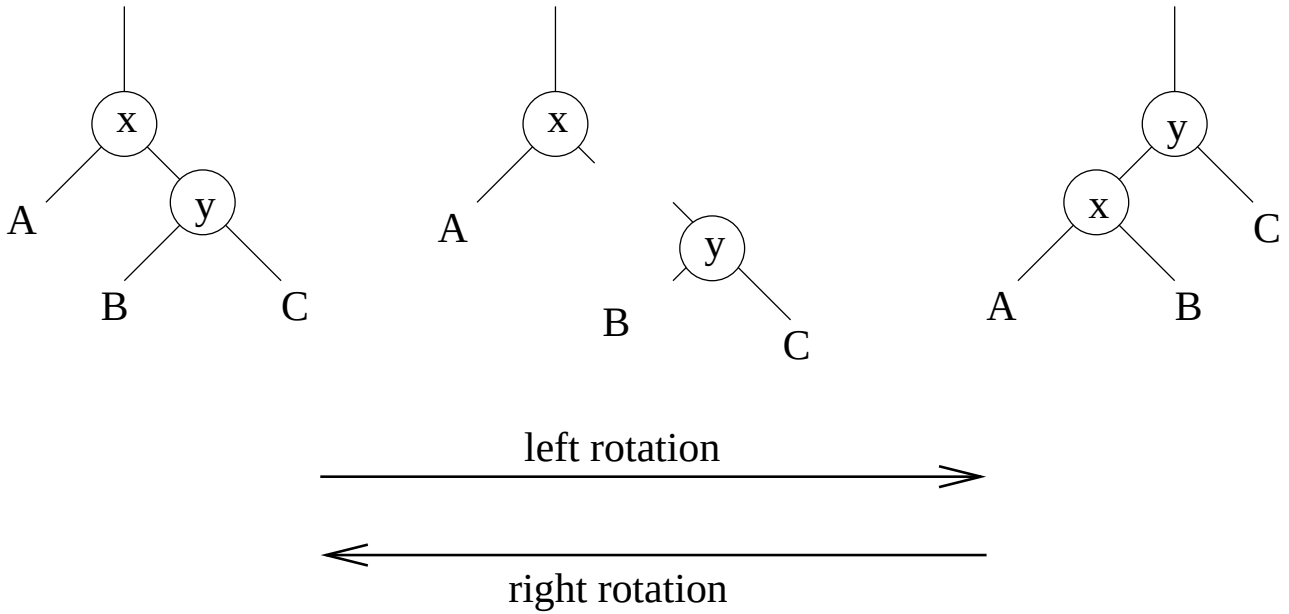


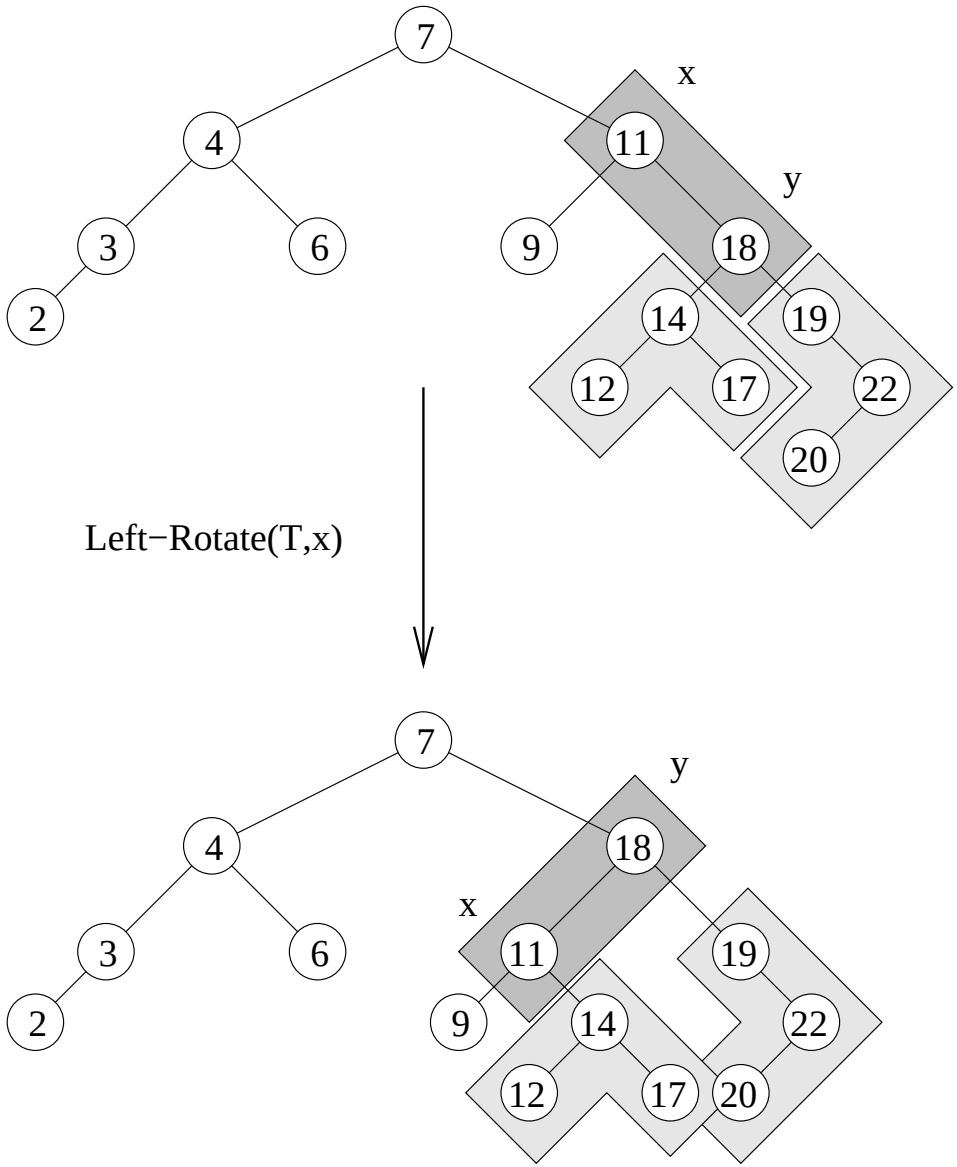
Rotations

– useful local operations: preserve the BST property

left and **right** rotations

Assumptions:
left-rotation on x: right child not NULL
right-rotation on y: left child not NULL





Exercise: Show that **Left-Rotate()** preserves the BST property.

Assignment Problem 10.2. (deadline: July 22, 5:30pm)
Prove that at most $n-1$ right rotations suffice to transform any n-node binary search tree into a right-going chain.

Argue that this implies that any n-node binary search tree can be transformed into any other n-node binary search tree using $\mathcal{O}(n)$ rotations (left and right).

Insertion

Can be done in $O(\log n)$ time

Suppose we insert new node z

First, **ordinary BST insertion** of z according to key value (walk down from the root as when searching for the element and then add new node instead of NULL-leaf)

```
Tree-Insert(T, z)
 1: y ← NULL
 2: x ← root[T]
 3: while x ≠ NULL do
 4:     y ← x
 5:     if key[z] < key[x] then
 6:         x ← left[x]
 7:     else
 8:         x ← right[x]
 9:     end if
10: end while
11: parent[z] ← y
12: if y = NULL then
13:     root[T] ← z      /* T was empty */
14: else if key[z] < key[y] then
15:     left[y] ← z
16: else
17:     right[y] ← z
18: end if
```

Then, color z **red**

May have **violated red-black property**

(e.g., z's parent is red)

Finally, call a **fixup** procedure (restores red-black property)

What can have gone wrong?

Reminder:

(RB1) every node is either red or black

(RB2) the root is black

(RB3) every leaf (NULL) is black

(RB4) red nodes have black children

(RB5) for all nodes, all paths from node to descendant leaves contain the same number of black nodes

(RB1) and **(RB3)** certainly still hold

(RB5) as well, since z replaces (black) sentinel, and z is red with sentinel as children

(RB2) and **(RB4)** may be violated (because z is red);
(RB2) if z is root, and (RB4) if z's parent is red

RB-Insert-Fixup(T, z)

```
1:  while color[parent[z]] = RED do
2:      if parent[z] = left[parent[parent[z]]]
        then
3:          y ← right[parent[parent[z]]]
4:          if color[y] = RED then
5:              color[parent[z]] ← BLACK
6:              color[y] ← BLACK
7:              color[parent[parent[z]]] ←
                RED
8:              z ← parent[parent[z]]
9:          else
10:             if z = right[parent[z]] then
11:                 z ← parent[z]
12:                 Left-Rotate(T, z)
13:             end if
14:             color[parent[z]] ← BLACK
15:             color[parent[parent[z]]] ←
                RED
16:             Right-Rotate(T, parent[parent[z]])
17:         end if
18:     else
19:         same as then with left and right
            exchanged
20:     end if
21: end while
22: color[root[T]] ← BLACK
```

Assignment Problem 10.3. (deadline: July 22, 5:30pm)
Consider a red-black tree formed by inserting n nodes with **RB-Insert** into an initially empty tree. Argue that if $n > 1$, the tree has at least one red node. What is the minimal number of red nodes of a red-black tree with 5 elements? What is the minimal number of red nodes of a red-black tree with 5 elements obtained by calling 5 times **RB-Insert** on an empty tree?

Loop invariant

at **start** of each iteration of **while** loop:

(1) node z is red,

(2) if parent$[z]$ is the root, then parent$[z]$ is black,

(3) if there is a violation of RB properties, there is at most one violation, and it's either property (RB2) or (RB4):

- if property (RB2), then z is the root and is red
- if property (RB4), then z and parent$[z]$ are both red

(1) and (2) for better understanding, (3) is important

Initialization

Prior to first iteration, we had OK red-black tree, then inserted red node z

1. when RB-Insert-Fixup is called, z is the red node that was added
2. if parent$[z]$ is the root, then parent$[z]$ was black and did not change prior to call of **RB-Insert-Fixup**
3. Properties (RB1), (RB3), and (RB5) hold in any case

   If (RB2) is violated ("root is black"), then root is newly added z, the only internal node in tree
   Also, parent and children of z are the (black) sentinel, thus **no** violation of (RB4) ("red nodes have black children"), and (RB2) is **only** violation

   If (RB4) violated, then **must** be that both z and parent$[z]$ are red (tree was OK, and red z has black children (sentinel))
   parent$[z]$ red implies that it's not the root, thus (RB2) is not violated

Termination

Nothing to show here, just see what we actually **get** from the invariant

Termination **only** because parent$[z]$ is black

Property (RB4) not violated [z red, parent$[z]$ black],
hence (RB2) is only possible problem.

Note: if z is root, then parent$[z]$ is sentinel NULL(T), which is black

Line 22 restores property (RB2), thus after termination, all red-black properties hold

Now, it's enough to show the maintenance.

Maintenance

Actually six cases, but three are symmetric to other three, depending on whether parent$[z]$ is **left** or **right** child of its parent (line 2)

Question: Can we assume that parent[parent$[z]$] exists?

Answer: part (2) of the invariant: if parent$[z]$ is the root, then parent$[z]$ is black. However, we enter **while** loop **only** if parent$[z]$ is red, thus parent$[z]$ **cannot** be the root, and parent[parent$[z]$] **must** exist.

Three cases, corresponding to lines 5–8, 11–12, and 14–16 (cases 2 and 3 are combined)

```
1:  while color[parent[z]] = RED do
2:      if parent[z] = left[parent[parent[z]]]
        then
3:          y ← right[parent[parent[z]]]
4:          if color[y] = RED then
5:              color[parent[z]] ← BLACK
6:              color[y] ← BLACK
7:              color[parent[parent[z]]] ←
                RED
8:              z ← parent[parent[z]]
9:          else
10:             if z = right[parent[z]] then
11:                 z ← parent[z]
12:                 Left-Rotate(T, z)
13:             end if
14:             color[parent[z]] ← BLACK
15:             color[parent[parent[z]]] ←
                RED
16:             Right-Rotate(T, parent[parent[z]])
17:         end if
18:     else
19:         same as then with left and right
            exchanged
20:     end if
21: end while
22: color[root[T]] ← BLACK
```

Case 1 clearly distinguished from 2 and 3 by color of z's parent's sibling (called z's "**uncle**")

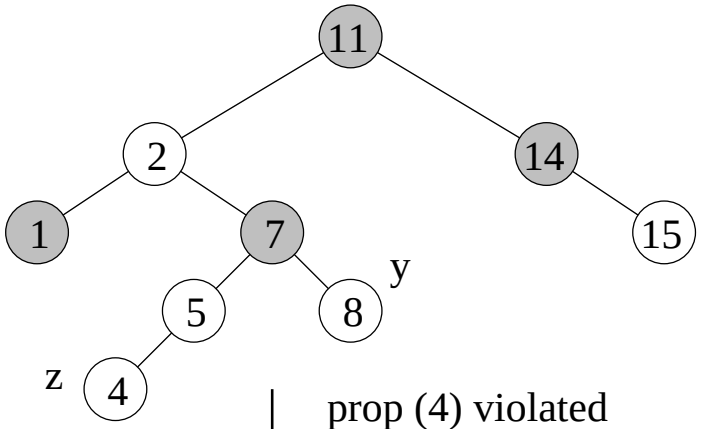
Line 3 points y to z's uncle right[parent[parent[z]]] (recall: parent[z] = left[parent[parent[z]]])

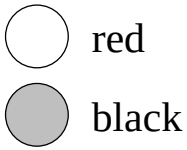
Test in line 4: if y, the uncle, is red, then case 1, otherwise control passes to cases 2 and 3.

In any case, z's grandparent parent[parent[z]] is black since parent[z] is red, and (RB4) can be violated **only** between z and parent[z]

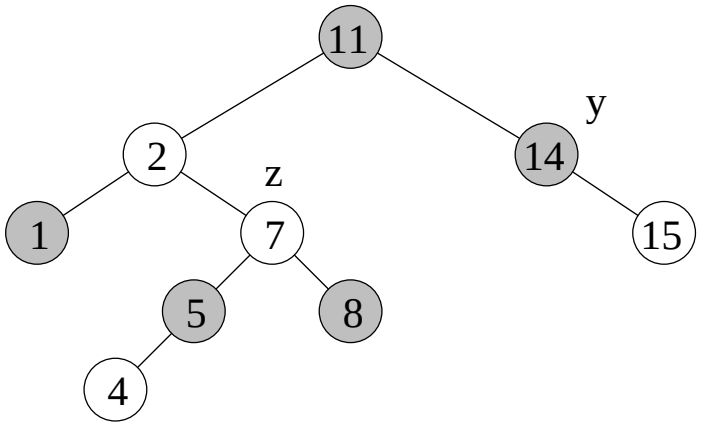
(if parent[parent[z]] were red, then parent[z] would have to be black, which it is not)

Case 1: z's uncle y is red





prop (4) violated
uncle red, thus case 1




```
3:  y ← right[parent[parent[z]]]
4:  if color[y] = RED then
5:      color[parent[z]] ← BLACK
6:      color[y] ← BLACK
7:      color[parent[parent[z]]] ← RED
8:      z ← parent[parent[z]]
9:  else
10:     ...
11: end if
```

Case 1 executed when parent$[z]$ and y are red

We know parent[parent$[z]$] is black, thus can recolor both parent$[z]$ and y black

This solves problem of z and parent$[z]$ both being red

Also, can color parent[parent$[z]$] red, maintaining (RB5) ["all paths the same number of black nodes"]

Repeat **while** loop with parent[parent$[z]$] as new node z (pointer z moves two levels up)

Main question: **Why does this maintain loop invariant at start of next iteration?**

Let z be the current z, and let z' be parent[parent[z]], new z in the next iteration

1. We color parent[parent[z]] red, thus z' is red at start of next iteration
2. Node parent[z'] is parent[parent[parent[z]]] in this current iteration, and we don't change its color. If it's the root, it was black prior to this iteration, and it remains black
3. We already know case 1 maintains property (RB5), and obviously it doesn't affect (RB1) or (RB3).
   If z' is root at start of next iteration, then case 1 corrected only violation of (RB4) in this iteration. z' is red and the root, thus (RB2) is the only one violated in the next iteration, and this is due to z'.
   Otherwise (z' not root), the case 1 has not violated (RB2), but fixed (RB4) violation that existed at start of iteration. It made z' red and left parent[z'] alone. If that one was black, no violation of (RB4). If parent[z'] was red, then coloring z' red created violation of (RB4) between z' and parent[z'].

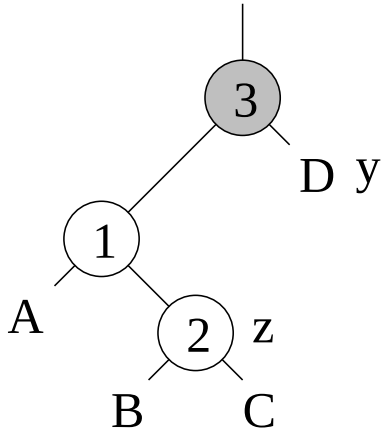
Case 1 is OK!

Case 2: z's uncle y is black and z is right child

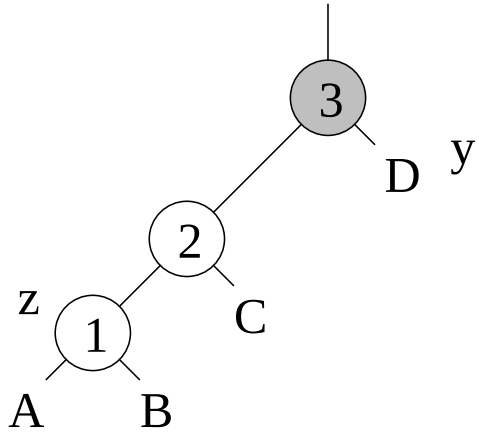
Case 3: z's uncle y is black and z is left child

Idea: transform case 2 into case 3 and then go from there

10: **if** $z = \text{right}[\text{parent}[z]]$ **then**
11: $z \leftarrow \text{parent}[z]$
12: Left-Rotate(T, z)
13: **end if**

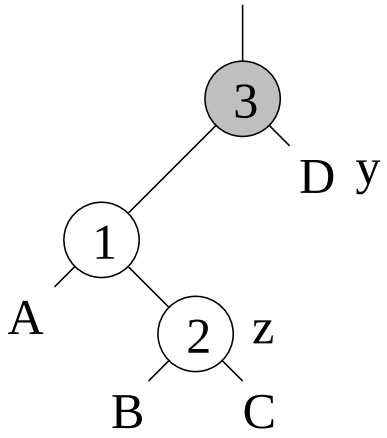


Case 2

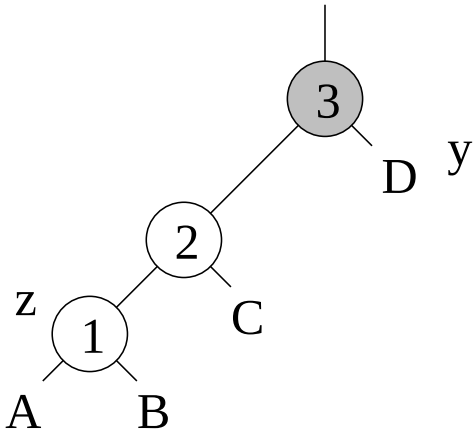


Case 3

In case 2, z is right child



Case 2



Case 3

Use left rotation to turn it into left child, and then apply case 3 (lines 14–16)

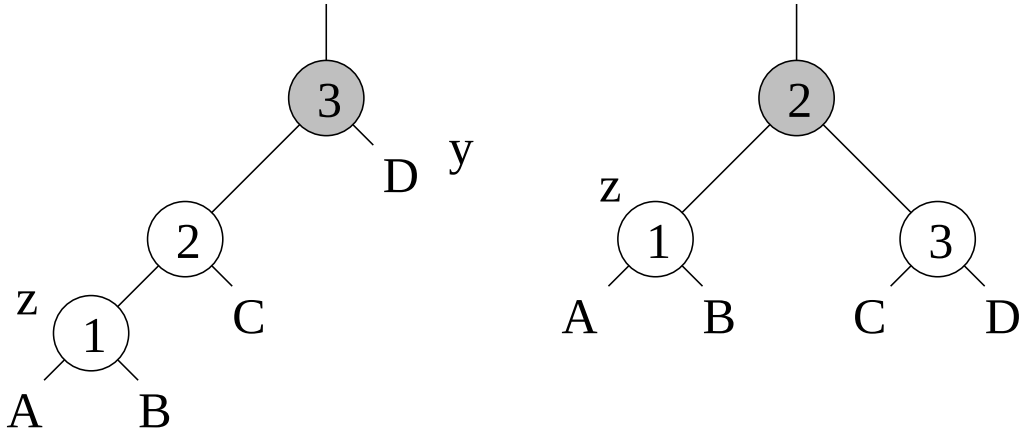
Both z and parent$[z]$ are red, so neither black-heights nor (RB5) are affected

at the end z's uncle y is still black (otherwise we'd be in case 1)

Also, parent[parent$[z]$] exists (has existed when lines 2–3 were executed), and moving z up in line 11 and down in line 12 doesn't change its position in the tree)

In case 3, we change colors and right-rotate; which preserves property (RB5)

14: color[parent[z]] $\leftarrow$ *BLACK*
15: color[parent[parent[z]]] $\leftarrow$ *RED*
16: Right-Rotate(T, parent[parent[z]])

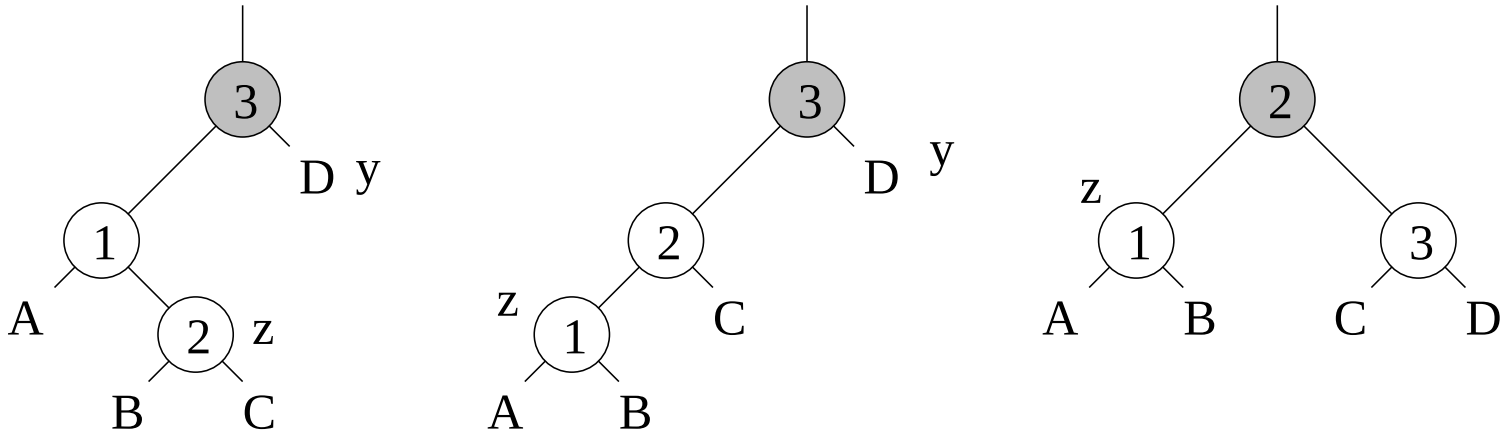


Case 3

Now, no longer two red adjacent nodes $\Rightarrow$ done; body of the **while** loop is not executed another time (since parent[z] is black)

Remark: there is no violation to (RB4) between node 3 and he uncle y, since y is *black* (otherwise it would be Case 1).

Assignment Problem 10.4. (deadline: July 22, 5:30pm)
Suppose that the black-height of the root of each of the subtrees A, B, C, D in the left most tree in the figure bellow is k. Label each node in each tree with its black-height to verify that property **(RB5)** is preserved by the transformations.



Case 2 Case 3

Again, why do these operations maintain loop invariant?

1. Case 2 makes z point to parent$[z]$, which is red
2. Case 3 turns parent$[z]$ black. If parent$[z]$ is root at start of next iteration, we're fine.
3. - Similar to case 1, properties (RB1), (RB3), and (RB5) hold
   - new z is not the root in cases 2 and 3, thus no violation of (RB2).
   - Cases 2 and 3 don't introduce any violation of (RB2), because only node made red becomes child of black node by rotation in case 3
   - Cases 2 and 3 correct the only violation of (RB4), and they don't introduce new violation

we're done!

Running time: Insert takes $O(\log n)$. Fixup loops only if case 1 is executed, and there z moved two levels up in each iterations. Thus, fixup also $O(\log n)$.

Deletion

Can be done in $O(\log n)$ time

Suppose we delete a node z from the tree.

First, call slightly modified **BST deletion** on z according to key value.
Differences:

- NULL replaced by $NULL[T]$
- line **11**: before we performed this operation only if x was not NULL; now, $NULL[T]$ is a regular node, so we can set its parent to parent of y — a node which is removed from the tree
  we will use this information later in fixup
- if removed node was *BLACK*, we call fixup procedure

Tree-Delete(T, z)

1: **if** left$[z]$ = NULL$[T]$ **or**
   right$[z]$ = NULL$[T]$ **then**
2: $y \leftarrow z$
3: **else**
4: $y \leftarrow$ Tree-Successor(z)
5: **end if**
6: **if** left$[y] \neq$ NULL$[T]$ **then**
7: $x \leftarrow$ left$[y]$
8: **else**
9: $x \leftarrow$ right$[y]$
10: **end if**
11: parent$[x] \leftarrow$ parent$[y]$
12: **if** parent$[y]$ = NULL$[T]$ **then**
13: root$[T] \leftarrow x$
14: **else if** $y =$ left[parent$[y]$] **then**
15: left[parent$[y]$] $\leftarrow x$
16: **else**
17: right[parent$[y]$] $\leftarrow x$
18: **end if**
19: **if** $y \neq z$ **then**
20: key$[z] \leftarrow$ key$[y]$
21: copy y's data into z
22: **end if**
23: **if** color$[y] =$ *BLACK* **then**
24: **RB-Delete-Fixup**(T, x)
25: **end if**
26: return y

Recall: Three cases.

1. z has *no children*: At parent$[z]$, just replace link to z with NULL$[T]$
2. z has *one child*: remove node z, make a new link between the z's parent and the z's child
3. z has *two children*: remove z's successor y (which has no left child, as seen from Homework 9.3), and replace z's key and data with the y's key and data

Recall:

- y is a node which is removed from the tree
- x is a child of y (remember y has at most one child); x can be NULL$[T]$
- whether x is NULL$[T]$ or not, parent$[x]$ points at the former parent of y (will be used in the **Fixup** procedure)

Delete Fixup

Removing node y may have **violated red-black property**.

Example: if y was *BLACK* and not the root, then a path going from a leaf to the root previously containing y will have 1 black node less than other paths: property **(RB5)** is violated.

What can have gone wrong?

(RB1) (red or black) and **(RB3)** (NULL is black) certainly still hold

2 cases:

Case RED: color$[y] = RED$ — no violation:

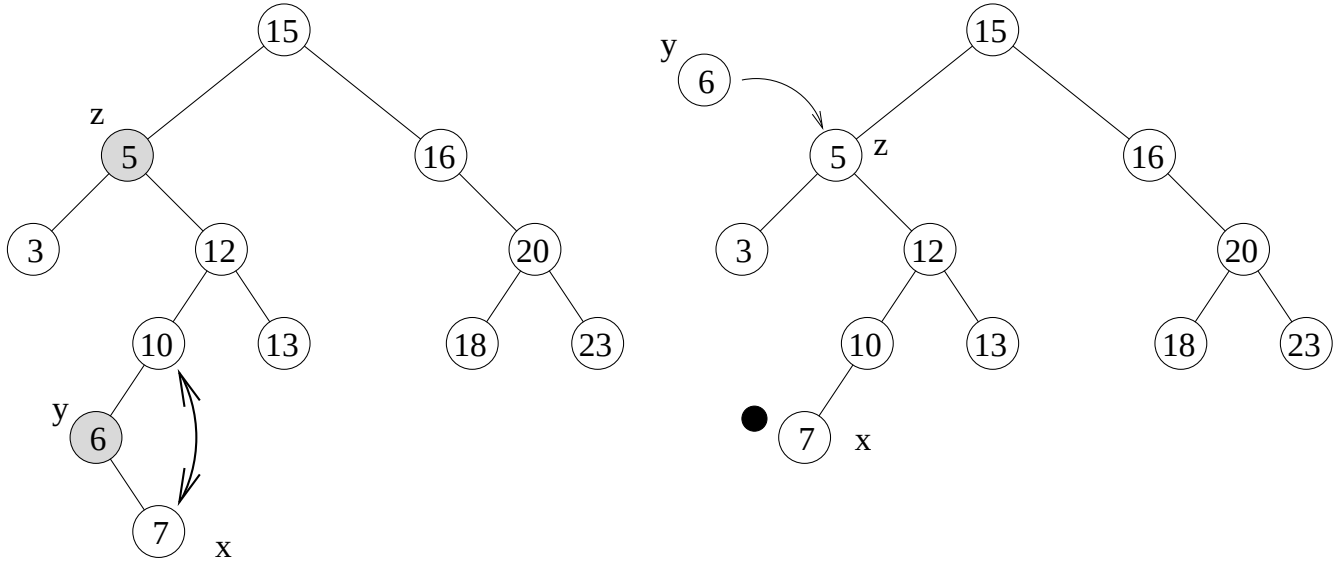
- no black-height in the tree has changes, **(RB5)** is ok!
- no red nodes have been made adjacent, **(RB4)** is ok!
- y couldn't be the root, so the root remains black, **(RB2)** is ok!

Case BLACK: color$[y] = BLACK$ — all three above can be violated:

- **(RB5)**: as above, violated for any ancestor of y
- **(RB4)**: if both parent$[y]$ = parent$[x]$ and x are *RED*
- **(RB2)**: if y was the root, then its child x has become a new root; if color$[x]$ is *RED*, we have a violation

*Fixing **(RB5)***:

mark x with an "extra black"



but now x is either "doubly black" or "red-and-black": we have a violation to **(RB1)**

*The idea of fixing **(RB1)***:

let x be the node with "extra black"; move x up the tree until

- x is "red-and-black", simply color it *BLACK*
- x is the root, we can forget about "extra black"
- suitable rotations and recolorings can be performed

*Fixing **(RB2)***:

x is the root and is *RED*

the procedure just colors x black and terminated which fixes the problem

Note: violation **(RB2)** implies there is no other violation

*Fixing **(RB4)***:

x and parent$[x]$ are both *RED*, so x is "red-and-black"

the procedure just colors x black and terminates which fixes **(RB4)** and also **(RB1)**

RB-Delete-Fixup(T, x)

```
 1: while x ≠ root[T] and color[x] = BLACK do
 2:     if x = left[parent[x]] then
 3:         w ← right[parent[x]] /* sibling */
 4:         /* Case 1: */
 5:         if color[w] = RED then
 6:             color[w] ← BLACK
 7:             color[parent[x]] ← RED
 8:             Left-Rotate(T, parent[x])
 9:             w ← right[parent[x]]
10:         end if
11:         /* Case 2: */
12:         if color[left[w]] = BLACK and color[right[w]] = BLACK then
13:             color[w] ← RED
14:             x ← parent[x]
15:         else
```

```
16:            /* Case 3: */
17:            if color[right[w]] = BLACK then
18:                color[left[w]] ← BLACK
19:                color[w] ← RED
20:                Right-Rotate(T, w)
21:                w ← right[parent[x]]
22:            end if
23:            /* Case 4: */
24:            color[w] ← color[parent[x]]
25:            color[parent[x]] ← BLACK
26:            color[right[w]] ← BLACK
27:            Left-Rotate(T, parent[x])
28:            x ← root[T]
29:        end if
30:    else
31:        same as then with "left" and "right" exchanged
32:    end if
33: end while
34: color[x] ← BLACK
```

*Fixing **(RB1)***:

Description:

x — the node with "extra black"

w — sibling of x

- $w \neq \text{NULL}[T]$ — otherwise: black-height of parent$[x]$ is 1, but a path going down to a leaf NULL$[T]$ through x will have at least 2 black nodes

4 cases:

1. lines 5–9: w is *RED*, converted to the other cases
   Note: parent$[x]$ must be *BLACK*
2. lines 12–14: w and both children of w are *BLACK*, move x one step up
   Note: when entering *Case 2.* from *Case 1.*, new x must be *RED*, so the procedure will terminate prior to the next step
3. lines 17–21: w and its right child are *BLACK*, the left child is *RED*, converted to *Case 4.*

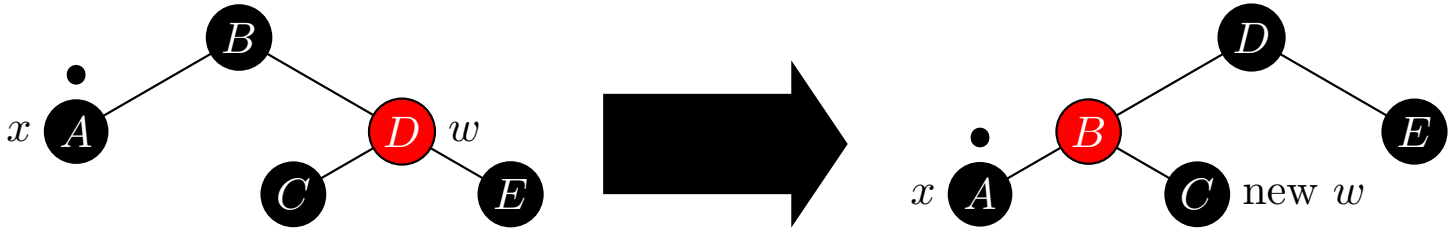
4. lines 24–28: w is *BLACK* and its right child is *RED*, we can remove "extra black" node, hence no violation; by setting x to root$[T]$ we will guarantee that the loop terminates prior to the next step

Note:

As we will see in the moment, execution of the loop continues only in *Case 2.* (assuming that "new x" is *BLACK* and not the root)

Case 1. lines 5–9: w is *RED*, converted to the other cases

Note: parent$[x]$ must be *BLACK*



6: color$[w] \leftarrow$ *BLACK*

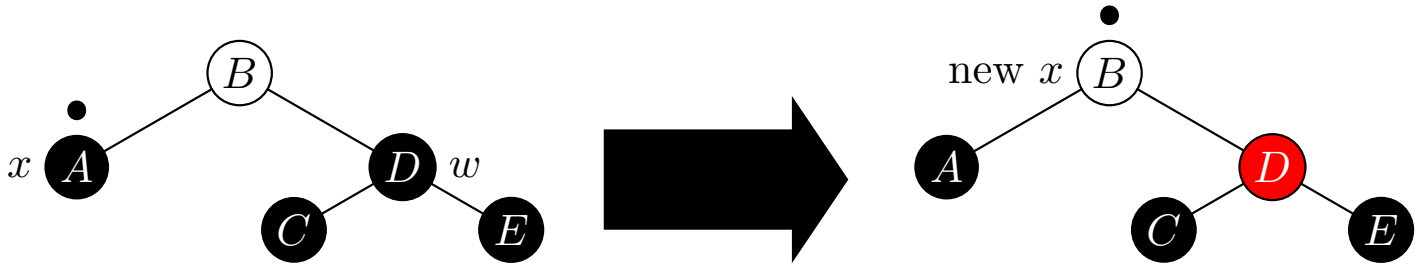
7: color[parent$[x]] \leftarrow$ *RED*

8: Left-Rotate$(T,$ parent$[x])$

9: $w \leftarrow$ right[parent$[x]]$

Case 2. lines 12–14: w and both children of w are *BLACK*, move x one step up

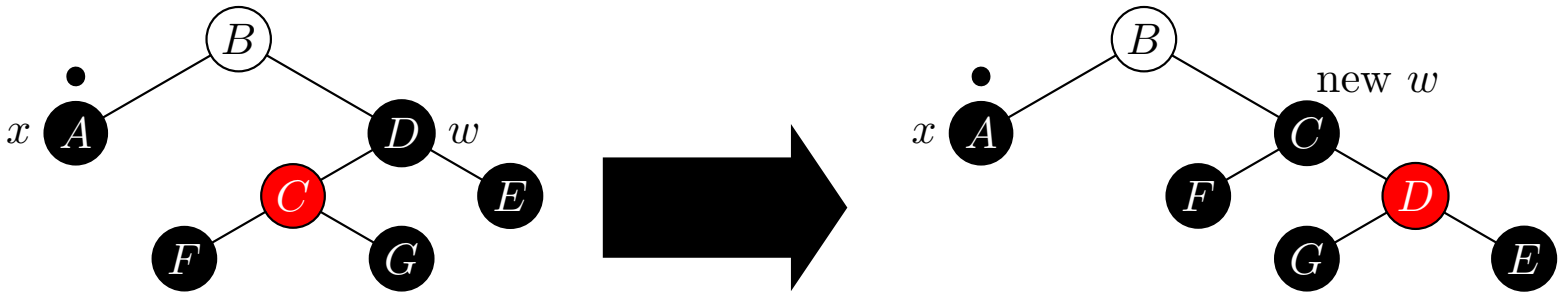
Note: when entering *Case 2.* from *Case 1.*, new x must be *RED*, so the procedure will terminate prior to the next step



13: color$[w] \leftarrow$ *RED*

14: $x \leftarrow$ parent$[x]$

Case 3. lines 17–21: w and its right child are *BLACK*, the left child is *RED*, converted to *Case 4.*



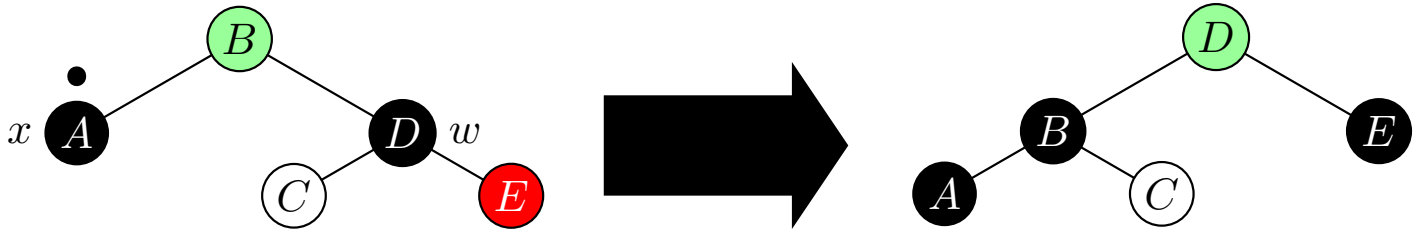
18: color[left[w]] $\leftarrow$ *BLACK*

19: color[w] $\leftarrow$ *RED*

20: Right-Rotate(T, w)

21: $w \leftarrow$ right[parent[x]]

Case 4. lines 24–28: w is *BLACK* and its right child is *RED*, we can remove "extra black" node, hence no violation; by setting x to root$[T]$ we will guarantee that the loop terminates prior to the next step



24: color$[w] \leftarrow$ color[parent$[x]]$
25: color[parent$[x]] \leftarrow$ *BLACK*
26: color[right$[w]] \leftarrow$ *BLACK*
27: Left-Rotate$(T,$ parent$[x])$
28: $x \leftarrow$ root$[T]$

Note: Black token was removed during transformation.

Assignment Problem 10.5. (deadline: July 22, 5:30pm)
Consider all 4 cases of the algorihtm **RB-Delete-Fixup** (see also Figure 13.7 in the textbook). Verify that all 4 transformation preserve property **(RB5)**.

Running time analysis.

BST-Delete takes $\mathcal{O}(h)$ time

RB-Delete-Fixup takes also $\mathcal{O}(h)$ time:

we continue loop only in *Case 2.*

in *Case 2.* the height of x increases by 1

loop terminates when x is the root or has *RED*

hence we iterate at most h times

since, $h = \mathcal{O}(\log n)$, procedure **RB-Delete** take time $\mathcal{O}(h) = \mathcal{O}(\log n)$