Algorithms (COMP3600/6466)
Data Structures: Red/Black Trees

Pascal Bercher

(working in the Intelligent Systems Cluster)

School of Computing
The Australian National University

Tuesday & Wednesday, 19. & 20.9.2023

Introduction

Motivation

As before, we want to do (at least) the following operations efficiently:

- access, i.e., search
- insertion
- deletion
- min or max, respectively (or both)

Which runtime did we have for binary search trees? All were $O(h)$.

What about AVL trees? All were $O(log(n))$.

What about heaps? Most are $O(log(n))$, max is $O(1)$, search is $O(n)$.

So, *can* we even do better?

- Not asymptotically. But in practice.
- (Deletion gets *much* cheaper via more efficient self-balancing.)

We want to improve, but how?

Similarities to AVL trees:

- Still a binary search tree!
- Still doing self-balancing to achieve height $h \in O(log(n))$ to achieve $O(log(n))$ runtime for most operations.

Differences to AVL trees:

- AVL trees enforce a strict maximal height difference of 1 between sub trees, so rotations can occur often after data updates.
- Red/Black trees might be deeper (but still with $h \in O(log(n))$) thus requiring fewer balancing operations.
  - The deepest leaf cannot be more than twice the depth of the shallowest leaf.
  - Checked by 'coloring' nodes into one of two colors: **red** and **black**.

Basics

Red/Black Trees

A **red**/**black** tree is a binary search tree with the following properties:

- Each node uses an additional bit representing its color: **red** and **black**.
- The root node it **black**.
- Every leaf is a **black** NIL.
- If a node is **red**, both its children are **black**. (Thus there can be no paths with two consecutive **red** nodes.)
- For each node, all paths from this node to each of its leaves contain the same number of **black** nodes. (Called "black height".)

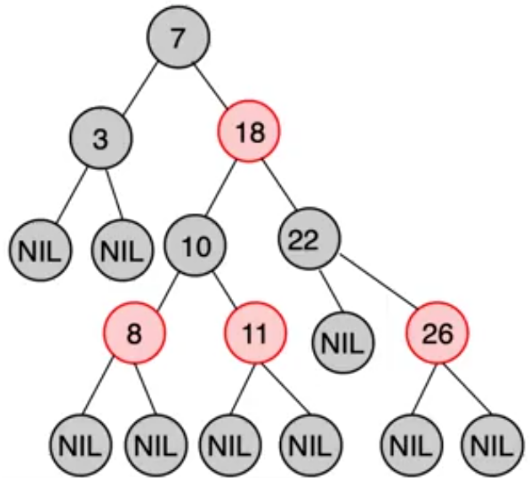
One advantage:
Deletion will only require a constant number of rotations!

Optimizations

- We require every leaf to be NIL, but there are exponentially many! So we just store a single one.
- We also assume that each inner node has exactly two children by letting one be NIL if required. (This simplifies some analyses.) Again, this is just one single (**black**) NIL node.
- Each node x has a "black height" $bh(x)$, which is the number of **black** nodes on any path from x to a leaf (***not*** including x itself).

Example

Is this a **red**/**black** tree?



Initial node **black**?
→ Check!

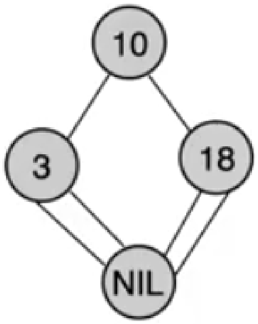
All nodes have two children?
→ Check!

All children of **reds** are **black**?
→ Check!

All paths from each node have the same number of **black** nodes?
→ Check!

Another Example

Is this a **red**/**black** tree?



Initial node **black**?
→ Check!

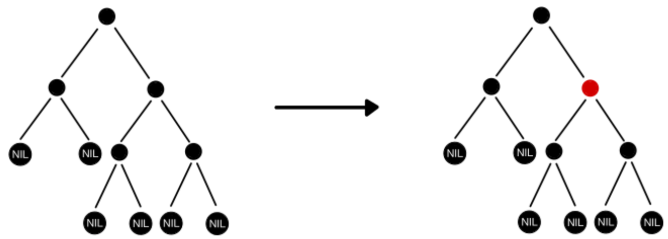
All nodes have two children?
→ Check!

All children of **reds** are **black**?
→ Check!

All paths from each node have the same number of **black** nodes?
→ Check!

Yet Another Example

So, why do we even have the **red** color, then?



- In the left tree, we didn't have the right black height for each node, e.g., the root had two **black** nodes on each path of its left, but three on its right.
- Introducing a **red** color turned it into a valid **red**/**black** tree.
- But this is still be a valid AVL tree anyway! Can we make an argument why this is still more flexible than AVL trees?
  Yes! Add more red nodes to increase height difference to 2.

On the Height of Red/Black Trees

The whole idea behind coloring is to obtain a height $h \in O(log(n))$.
But is that true? Does this follow from the red/black properties?

We will show $h \leq 2 \cdot log(n+1)$.

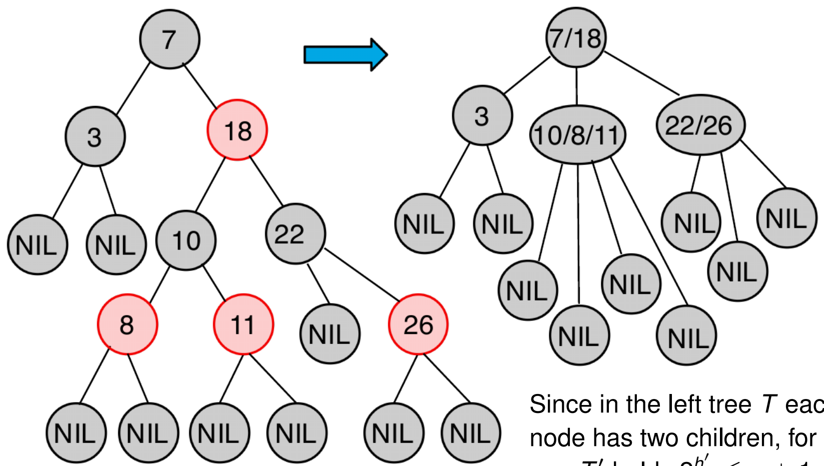
Note that here we refer with n to the *internal* nodes. This makes perfect sense since those are our keys! The "exponentially" many leafs are just NIL(s). (Remember: we just have one of them.)

How to show this? Exploit the property:

- If we remove all **red** nodes:
  All leaves are on the same level.
- Then relate the height of this 'new' tree to the original one.

On the Height of Red/Black Trees, Example

We merge all **red** nodes into their parents.



Since in the left tree T each inner node has two children, for the right one T' holds $2^{h'} \leq n+1$.

On the Height of Red/Black Trees, Proof

How many nodes does our **red**/**black** tree have?
leaves = # internal nodes + 1

Thus, # leaves of T: $n+1$ (with T being the **red**/**black** tree)
Thus, # leaves of T': $n+1$ (with T' being the new/'purely black' tree)

Let h be the height of T and h' that of T'.

We can conclude $2^{h'} \leq n+1$. ($2^{h'}$ can only *equal* $n+1$ if T didn't use red nodes. If it does, $2^{h'}$ will be strictly smaller.) Thus, $h' \leq log(n+1)$

Recall:

- If a node is **red**, both its children are **black**.
- Each node has a "unique" black height.

Now we can state $h \leq 2 \cdot h'$ and thus: $h \leq 2 \cdot log(n+1)$

How to maintain the tree's height?

With the $O(log(n))$ height guarantee the **red**/**black** tree guarantees $O(log(n))$ runtime for the following operations:

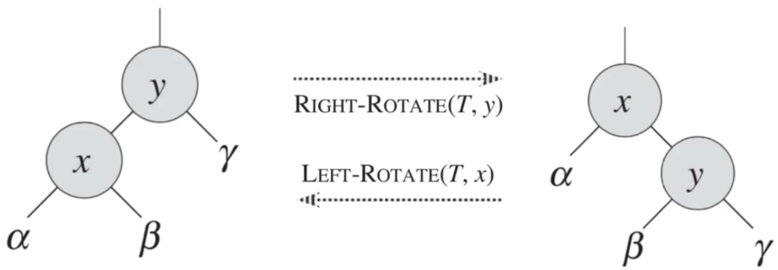
- Search
- Min, Max (both)
- Successor, Predecessor
- Insert, Delete

How to maintain the height for Insert and Delete?
→ Like for AVL trees: via re-balancing – here: also re-coloring!

Rotations

Rotations in Red/Black Trees

There is only a *single* rebalance operation, which is *rotation*:



Maybe a useful guide to remember and apply it correctly:

- Left-rotation: The left node is above and you push it down.
- Right-rotation: The right node is above and you push it down.

Insertion

Insertion

Procedure in a nutshell:

- Add a new node (like in binary search trees), color it **red**.
- Recolor where required.
- Rebalance via rotations. (Constantly many, also for deletion.)

Recolor and Rebalance

When the new node (denoted as z) is added as a child of a **black** node we are done.

Why is that the case?

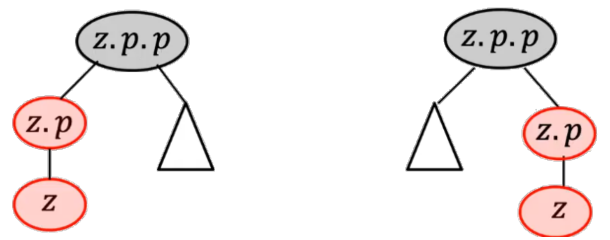
Because the number of **black** nodes from the root (or any parent node) to a leaf stays the same! (Recall: the new node is **red**.)

This is because we replace a **black** NIL by a **red** node – which again has only **black** NIL nodes. So the number of **black** nodes did not increase on this path.

Recolor and Rebalance

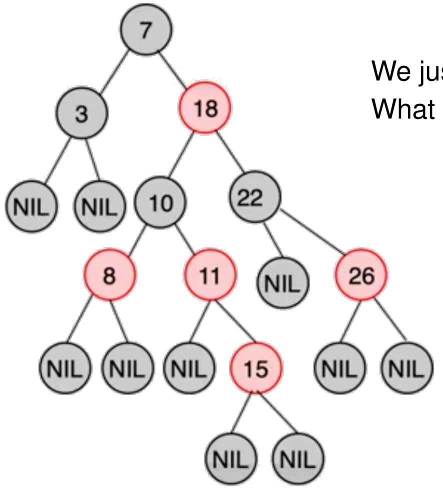
When the new node (denoted as z) is added as a child of a **black** node we are done. So we only have work to do if z is added to a **red** node.

- When z's parent is **red**, z's grandparent must exist (since the root can't be **red**) and must be **black** (otherwise we already had two **red** nodes in a row).
- We will then have six cases, three for each of two categories: z's parent is the left or the right child of z's grandparent.



Insertion, Example

Add 15 to the tree:



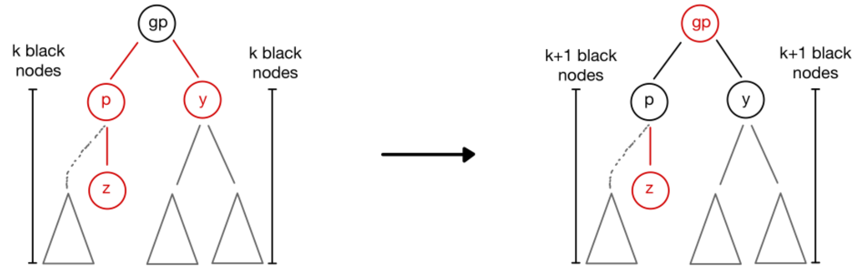
We just inserted node 15...
What now?

Recolor and Rebalance, category "on the left"

Category 1: *z*'s parent is the left child of *z*'s grandparent.

case 1 *z*'s uncle/aunt *y*[1] is **red**.

→ Recolor *z′s* parent and uncle/aunt to be **black** and *z*'s grandparent to be **red**. Then repeat checking **red**/**black** properties as if *z*'s grandparent is the new node. However, if the new *z* is the root, make it black and stop!



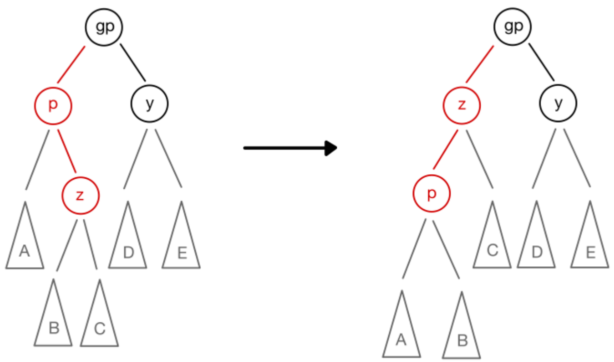
[1]The uncle/aunt of a node *x* is the other child of *x′s* grandparent, i.e., *x*'s parent's sibling.

Recolor and Rebalance, category "on the left"

Category 1: *z*'s parent is the left child of *z*'s grandparent.

case 2 *z*'s uncle/aunt *y* is **black** and *z* is a right child of its parent.

→ Left-rotate *z*'s parent and continue with case 3. Note that "p" in our final result will denote "z" in the next case 3.



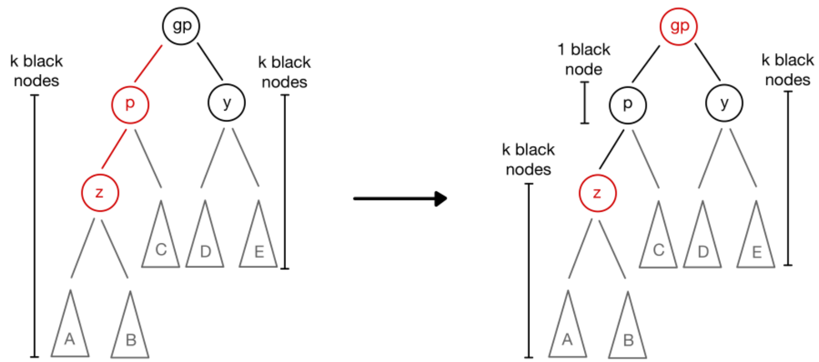
Note how the black heights remain unchanged.

Recolor and Rebalance, category "on the left"

Category 1: *z*'s parent is the left child of *z*'s grandparent.

case 3 *z*'s uncle/aunt *y* is **black** and *z* is a left child of its parent.

→ Recolor *z*'s parent to be **black** and *z*'s grandparent to be **red**. Then right-rotate *z*'s grandparent.



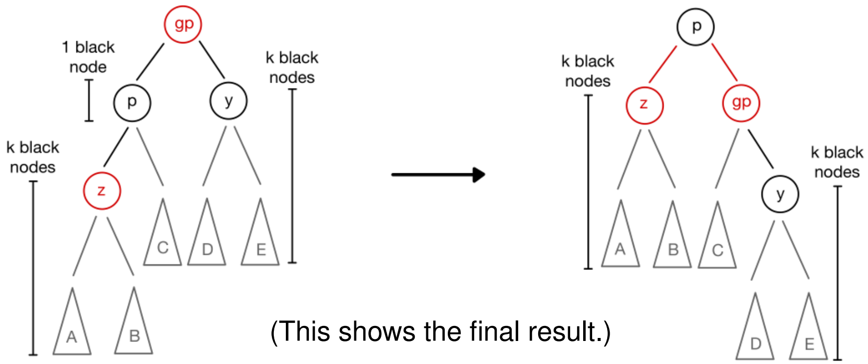
(Here we see just the recoloring step.)

Recolor and Rebalance, category "on the left"

Category 1: *z*'s parent is the left child of *z*'s grandparent.

case 3 *z*'s uncle/aunt *y* is **black** and *z* is a left child of its parent.

→ Recolor *z*'s parent to be **black** and *z*'s grandparent to be **red**. Then right-rotate *z*'s grandparent.



(This shows the final result.)

Recolor and Rebalance, category "on the left"

Category 1: *z*'s parent is the left child of *z*'s grandparent.

case 1 *z*'s uncle/aunt *y*[1] is **red**.

→ Recolor $z's$ parent and uncle/aunt to be **black** and *z*'s grandparent to be **red**. Then repeat checking **red**/**black** properties as if *z*'s grandparent is the new node. However, if the new *z* is the root, make it black and stop!

case 2 *z*'s uncle/aunt *y* is **black** and *z* is a right child of its parent.

→ Left-rotate *z*'s parent and continue with case 3. Note that "p" in our final result will denote "z" in the next case 3.

case 3 *z*'s uncle/aunt *y* is **black** and *z* is a left child of its parent.

→ Recolor *z*'s parent to be **black** and *z*'s grandparent to be **red**. Then right-rotate *z*'s grandparent.

(This is just a repetition, purely as overview.)

[1] The uncle/aunt of a node *x* is the other child of $x's$ grandparent, i.e., *x*'s parent's sibling.

Recolor and Rebalance, category "on the right"

Category 2: *z*'s parent is the right child of *z*'s grandparent.

case 1 *z*'s uncle/aunt *y* is **red**.

→ Recolor $z's$ parent and uncle/aunt to be **black** and *z*'s grandparent to be **red**. Then repeat checking **red**/**black**properties as if *z*'s grandparent is the new node.

case 2 *z*'s uncle/aunt *y* is **black** and *z* is a left child of its parent.

→ Right-rotate *z*'s parent and continue with case 3. Note that "p" in our final result will denote "z" in the next case 3.

case 3 *z*'s uncle/aunt *y* is **black** and *z* is a right child of its parent.

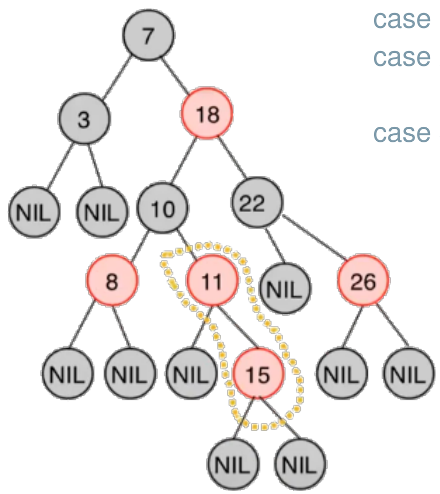
→ Recolor *z*'s parent to be **black** and *z*'s grandparent to be **red**. Then left-rotate *z*'s grandparent.

Note:

This is *identical* to category 1, just with "left" and "right" interchanged!

Insertion, Example

Add 15 to the tree:



Category 2:

case 1: *z*'s uncle/aunt is **red**.

case 2: *z*'s uncle/aunt is **black** and *z* is a left child of its parent.

case 3: *z*'s uncle/aunt is **black** and *z* is a right child of its parent.

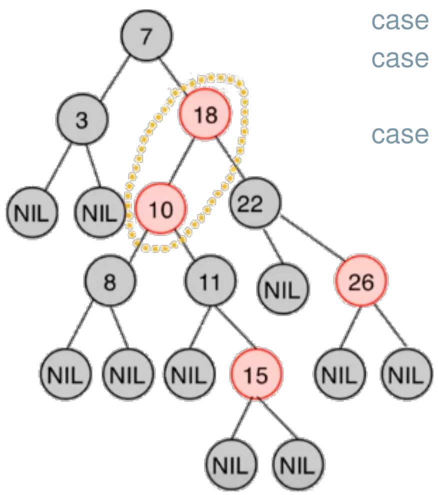
Now we have two **red** nodes in sequence with a **red** uncle/aunt, so we need to recolor. (Category 2, case 1)

I.e., recolor parent, uncle/aunt, and grandparent.

FYI: $z = 15$

Insertion, Example

Add 15 to the tree:



Category 2:
case 1: z's uncle/aunt is **red**.
case 2: z's uncle/aunt is **black** and z is a left child of its parent.
case 3: z's uncle/aunt is **black** and z is a right child of its parent.

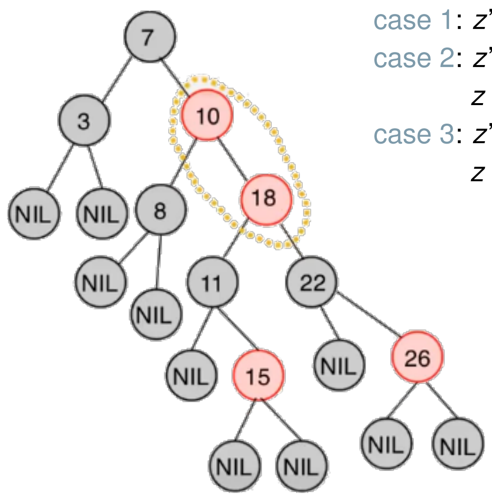
Now have again two **red** nodes in sequence but without **red** uncle/aunt, so we need to rotate. (Category 2, case 2)

I.e., right-rotate upper **red** node and continue with case 3.

FYI: $z = 10$

Insertion, Example

Add 15 to the tree:



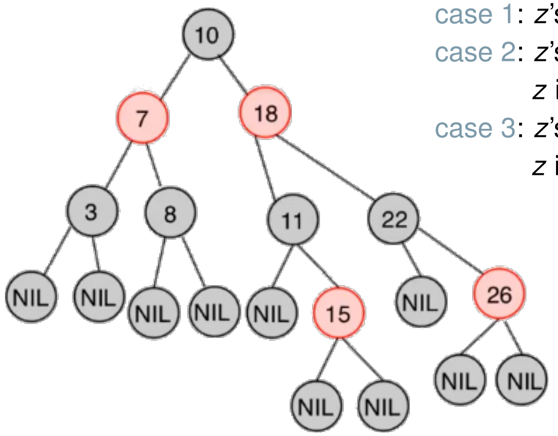
Category 2:
case 1: z's uncle/aunt is **red**.
case 2: z's uncle/aunt is **black** and z is a left child of its parent.
case 3: z's uncle/aunt is **black** and z is a right child of its parent.

For the third time we have two **red** nodes in sequence, but again without **red** uncle/aunt, so we left-rotate z's grandparent. (Category 2, case 3)

FYI: $z = 18$

Insertion, Example

Add 15 to the tree:



Category 2:
case 1: z's uncle/aunt is **red**.
case 2: z's uncle/aunt is **black** and z is a left child of its parent.
case 3: z's uncle/aunt is **black** and z is a right child of its parent.

We just rotated – and are done! :)

Not because we "reached the root", but because case 3 always terminates the process.

Insertion, Algorithm

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

The left code is for binary search trees.

Find the differences!

Insertion, Algorithm

```
RB-INSERT(T, z)
 1  y = T.nil
 2  x = T.root
 3  while x ≠ T.nil
 4      y = x
 5      if z.key < x.key
 6          x = x.left
 7      else x = x.right
 8  z.p = y
 9  if y == T.nil
10      T.root = z
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
14  z.left = T.nil
15  z.right = T.nil
16  z.color = RED
17  RB-INSERT-FIXUP(T, z)
```

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK                 // case 1
 6              y.color = BLACK                   // case 1
 7              z.p.p.color = RED                 // case 1
 8              z = z.p.p                         // case 1
 9          else if z == z.p.right
10                  z = z.p                       // case 2
11                  LEFT-ROTATE(T, z)             // case 2
12              z.p.color = BLACK                 // case 3
13              z.p.p.color = RED                 // case 3
14              RIGHT-ROTATE(T, z.p.p)            // case 3
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```


Insertion, Complexity

RB-Insert:

- Lines 1-16 take $O(log(n))$
- Line 17, which is RB-Insert-Fixup:
  - #rotations in an insertion: $O(1)$
    - For insertion, there are at most two rotations.
    - Rotation only happens in case 2 & case 3 of RB-Insert-Fixup.
    - Case 2, which contributes a rotation will always be followed by case 3, which also contributes a rotation.
    - Once case 3 is reached, we're done. Due to line 12 and line 13, the rotation will bring the mismatched color to an end.
  - Most changes are recoloring, which is quicker than rotation.
    - Recoloring takes $O(log(n))$, which happens in case 1.
    - After recoloring, we move two levels up to the node's grandparent, where the same process might be invoked again (and again . . .)

Deletion

Overview: Abstract Procedure

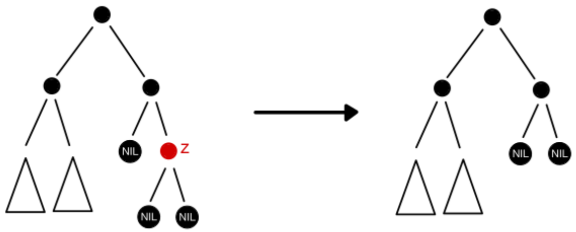
- First we delete the node *z* according to the standard deletion rules learned for binary search trees.
  - Recall that the node *z* gets replaced by either NIL (if it doesn't have children), by its child (if it has exactly one child), or by its successor (if it has two children).
  - **Important:** For a **red**/**black** tree, never change the color of any node during the pure deletion step! We do that in the second step when we repair the tree.
- In a second step we need to repair the **red**/**black** tree properties starting in a node *x* (defined later).

Deletion: Deletion, Algorithm

Deletion is similar to addition in that we also delete like in a binary search tree, and then repair the red/black properties as/if required.

Suppose the deleted node is *z*.

1. If *z* “represents” a leaf (i.e., *z* only has NIL children)
   - If *z* is **red**, remove *z*, set the edge that lead to *z* to now lead to a NIL node – and done!



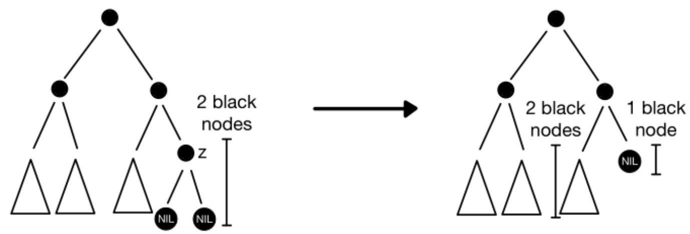
This works since the black height is not influenced.

Deletion: Deletion, Algorithm

Deletion is similar to addition in that we also delete like in a binary search tree, and then repair the red/black properties as/if required.

Suppose the deleted node is *z*.

1. If *z* “represents” a leaf (i.e., *z* only has NIL children)
   - If *z* is **black**, remove as before, but repair is now needed. (We consider later how we repair.)

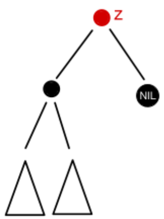


Deletion: Deletion, Algorithm

Deletion is similar to addition in that we also delete like in a binary search tree, and then repair the red/black properties as/if required.

Suppose the deleted node is *z*.

2. If *z* has 1 non-NIL child.
   - Note that in this case, *z* must be **black**, the non-NIL child must be **red**, and both its children are NIL. Why?



Why must *z* be **black**? Proof by contradiction:

- Otherwise we would have an imbalance! Since then on one side we had only one black node (NIL) and on the other at least two (on each path).
- The case where *z* has another **red** node child is not shown since this is obviously invalid.

Why must the **red** child have NIL children?

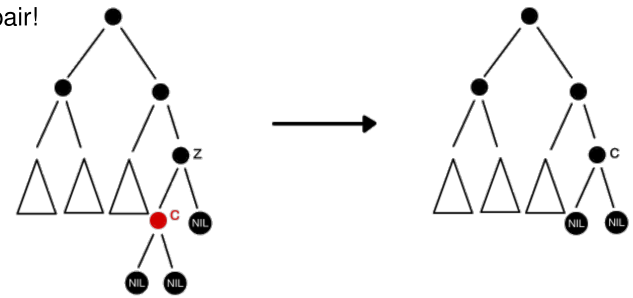
- So that *z* has a well-defined **black** height, as on its other path it has exactly one **black** node.

Deletion: Deletion, Algorithm

Deletion is similar to addition in that we also delete like in a binary search tree, and then repair the red/black properties as/if required.

Suppose the deleted node is *z*.

2. If *z* has 1 non-NIL child.
   - Note that in this case, *z* must be **black**, the non-NIL child must be **red**, and both its children are NIL.
   - Replace *z* with its only non-NIL child and color it in the color of *z*. No repair!

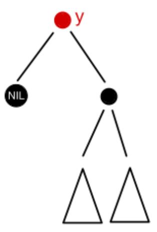


Deletion: Deletion, Algorithm

Deletion is similar to addition in that we also delete like in a binary search tree, and then repair the red/black properties as/if required.

Suppose the deleted node is *z*.

3. If *z* has 2 non-NIL children.
   - Let *y* be the node that replaces *z* (i.e., *z*'s successor).
   - If *y* is **red**, it can only have 2 NIL children. Why?



Proof by contradiction:

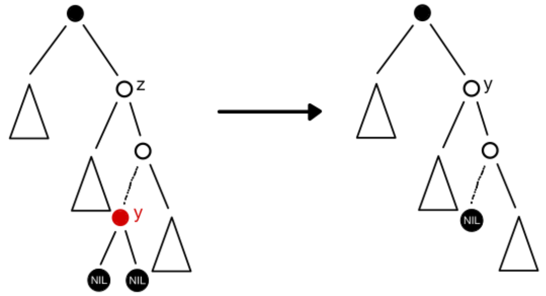
- The left node *must* be NIL since *y* is the successor!
- Its other child can be neither **red** (since then we had two in row) nor **black** as seen before because we'd get an imbalance (since the subtrees must contain **black** NIL nodes).

Deletion: Deletion, Algorithm

Deletion is similar to addition in that we also delete like in a binary search tree, and then repair the red/black properties as/if required.

Suppose the deleted node is *z*.

3. If *z* has 2 non-NIL children.
   - Let *y* be the node that replaces *z* (i.e., *z*'s successor).
   - If *y* is **red**, it can only have 2 NIL children.
   - Once *y* replaced *z* (keeping *z*'s original color), we are done!

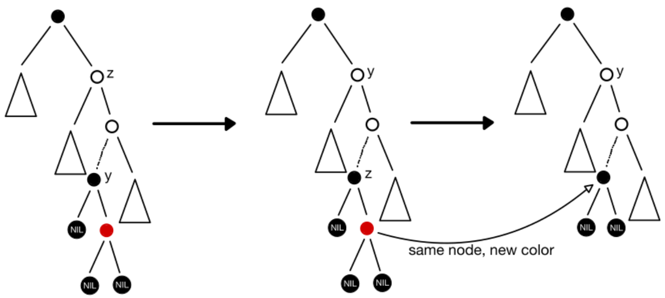


Deletion: Deletion, Algorithm

Deletion is similar to addition in that we also delete like in a binary search tree, and then repair the red/black properties as/if required.

Suppose the deleted node is *z*.

3. If *z* has 2 non-NIL children.
   - If *y* is **black** and has 1 non-NIL child (which is **red** and has NIL children), swap the key and data of *z* and *y*, apply the last rule (for 1 non-NIL child) to remove *z* (at the new position) – and done!

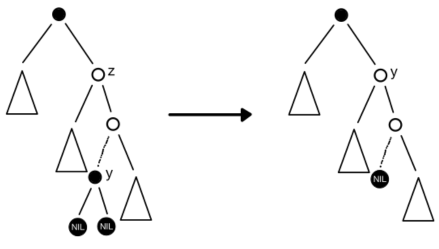


Deletion: Deletion, Algorithm

Deletion is similar to addition in that we also delete like in a binary search tree, and then repair the red/black properties as/if required.

Suppose the deleted node is *z*.

3. If *z* has 2 non-NIL children.
   - Otherwise (i.e., *y* is **black** and has two NIL children), swap *z*'s and *y*'s keys, but keep the original color of *z* (now *y*). Then delete the node that now contains key *z*. Repair is needed.



Deletion: Delete – Pseudocode

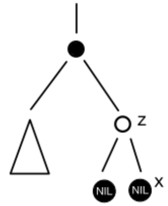
```
RB-DELETE(T, z)
  y = z
  y-original-color = y.color
  if z.left == T.nil
      x = z.right
      RB-TRANSPLANT(T, z, z.right)
  elseif z.right == T.nil
      x = z.left
      RB-TRANSPLANT(T, z, z.left)
  else y = TREE-MINIMUM(z.right)
      y-original-color = y.color
      x = y.right
      if y.p == z
          x.p = y
      else RB-TRANSPLANT(T, y, y.right)
          y.right = z.right
          y.right.p = y
      RB-TRANSPLANT(T, z, y)
      y.left = z.left
      y.left.p = y
      y.color = z.color
  if y-original-color == BLACK
      RB-DELETE-FIXUP(T, x)
```

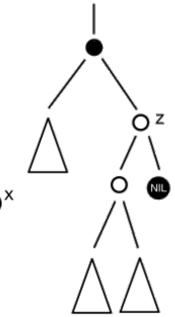
```
RB-TRANSPLANT(T, u, v)
  if u.p == T.nil
      T.root = v
  elseif u == u.p.left
      u.p.left = v
  else u.p.right = v
  v.p = u.p
```

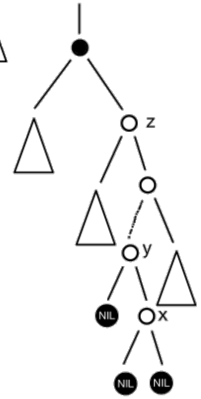
```
TREE-MINIMUM(x)
  while x.left ≠ NIL
      x = x.left
  return x
```


Repair: Overview: When to Repair?

1. *z* has zero children (both NIL)
   - *z* is **red**: No repair
   - *z* is **black**: *Repair*
2. *z* has one child (one NIL)
   - *z* is **red**: Can't be! (As seen before.)
   - *z* is **black**: No repair
3. *z* has two children (none NIL). Let *y* be its successor.
   - *y* is **red**: No repair (it takes *z*'s color)
   - *y* is **black**: *Repair* if two NIL children.
     (But pay attention to coloring in case *y* has 1 child)







After *z* got deleted, repair starts with node *x*. This will be a node that initially will get an "additional" black color.

Repair: Introduction

- To sum up, we first delete according to standard binary search tree deletion, and then repair if necessary. (See overview from last slide to see when repair is required.)
- **Repair starts from the node *x* that takes *z*'s (case 1 in the overview) resp. *y*'s (case 3 in the overview) position.**
  Always annotate the x (could be NIL) and check all heights!
- In all these repair cases, we deleted a **black** and are thus one **black** short! To compensate, we "add" an additional color **black** to *x*, making it **black**-**black**.
- We will re-distribute this color to other nodes, making them **black**-**black** or **red**-**black** (original, then added color).
- How to redistribute?
  - If *x* is **red**-**black**, make it **black**. (And we are done!)
  - If *x* is **black**-**black**, find "nearest" **red** and "distribute" one of *x*'s **black** colors to change that node color from **red**(-**black**) to **black**.

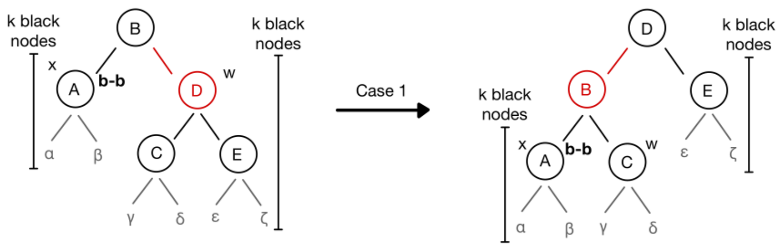
Repair: Categories and Cases

- There are 2 categories and 4 cases for each.
  1. Category 1: *x* is the left child of its parent.
  2. Category 2: *x* is the right child of its parent.
     (We don't cover this case explicitly since it is analogous)
- In the following, *x* is **black**-**black** and we denote *x*'s sibling (brother/sister) as *w*.

Repair: Category 1, Case 1

Case 1: *w* is **red**.

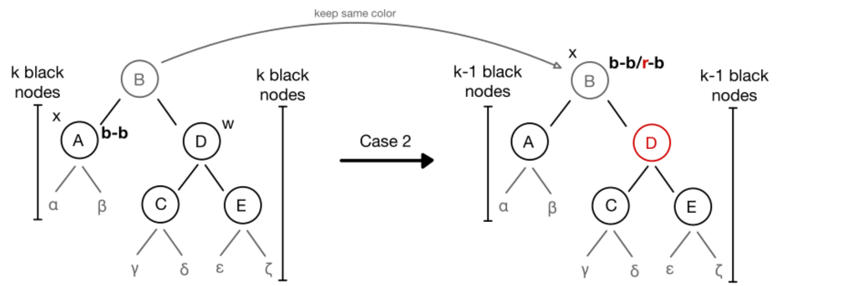
Swap the color between *w* and *x*'s parent, then rotate left on *x*'s parent. Then, continue to case 2/3/4 setting $w = x.p.right$.



Repair: Category 1, Case 2

Case 2: *w* and both of its children are **black**.

Take one **black** from *x* and *w* each (setting *w* to **red**), and move it to $x.p$. Since $x.p$ can initially be **red** or **black**, it becomes **red**-**black** or **black**-**black**. If we enter this case from case 1, $x.p$ will be **red**-**black**, and we can recolor it with **black** and are done. Otherwise, continue by setting $x = x.p$.

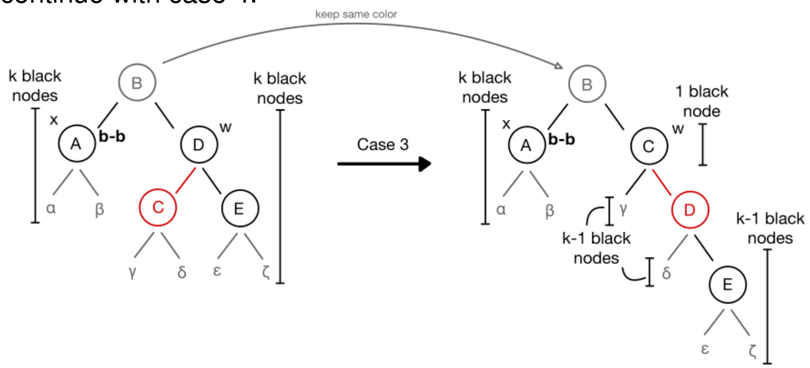


(Main idea: move one black from each side upwards.)

Repair: Category 1, Case 3

Case 3: *w* and its right child are **black**, but its left child is **red**.

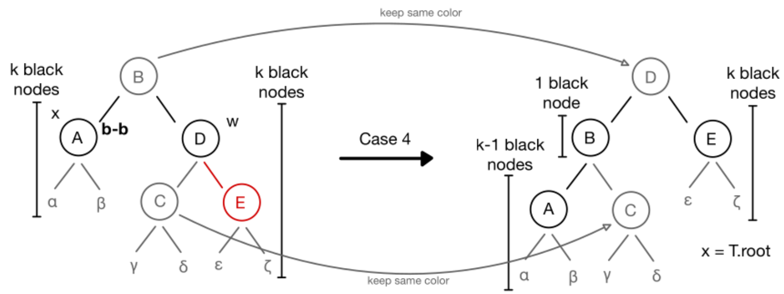
Swap color between *w* and its left child, then rotate right round *w*, and continue with case 4.



Repair: Category 1, Case 4

Case 4: *w* is **black**, *w*'s right child is **red**.

Set the color of *w* to be the color of $x.p$ and set the color of $x.p$ and $w.right$ to be **black**. Then, rotate left around $x.p$. The color change of $x.p$ and $w.right$ allows us to remove one of the **black** colors of *x* without violating the **red**/**black** tree requirements.



Repair: Repair – Pseudocode

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK                          // case 1
            x.p.color = RED                          // case 1
            LEFT-ROTATE(T, x.p)                      // case 1
            w = x.p.right                            // case 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                            // case 2
            x = x.p                                  // case 2
        else if w.right.color == BLACK
                w.left.color = BLACK                 // case 3
                w.color = RED                        // case 3
                RIGHT-ROTATE(T, w)                   // case 3
                w = x.p.right                        // case 3
            w.color = x.p.color                      // case 4
            x.p.color = BLACK                        // case 4
            w.right.color = BLACK                    // case 4
            LEFT-ROTATE(T, x.p)                      // case 4
            x = T.root                               // case 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```


Properties: Deletion, Time Complexity

- RB-Delete (without the repair) requires $O(log(n))$
- RB-Delete-Fixup (aka repair) requires $O(log(n))$
  - We need at most 3 rotations
  - Cases 1, 3, and 4: Constant number of color changes plus at most 3 rotations
  - Case 2: The pointer can move at most $O(log(n))$ times.

AVL vs. Red/Black Trees

Insertion and Deletion Compared

- What was the runtime of insert and delete?
  - AVL tree: $O(log(n))$
  - **red**/**black** tree: $O(log(n))$
- Part of the reason was traversing down the tree, which already takes $O(log(n))$.
- But traversals aren't the most expensive operation!

Rotations Compared

- How often do we have to rotate after insertion?
  - AVL tree: Only once! $O(1)$
  - **red**/**black** tree: At most 3 times. $O(1)$
- How often do we have to rotate after deletion?
  - AVL tree: Potentially in each node up to the root $O(log(n))$
  - **red**/**black** tree: At most 3 times! $O(1)$

Summary

- So, the **red**/**black** tree is more efficient for deletions.
- But the AVL tree is 'more balanced' (lower tree height), which leads to better look-up performance. (But has same performance in terms of asymptotic complexity.)
- Thus, if you do lots of deletions, the **red**/**black** tree is preferred. If in contrast the data is not changing (deleted) much and you do lots of look-ups, the AVL tree is preferred.

Summary

Summary

Today we covered ***red*/*black* *trees***.

They are a different way to achieve self-balancing.

Operations considered:

- Search: $O(log(n))$
- Insertion: $O(log(n))$
- Deletion: $O(log(n))$

We also considered when to use AVL trees and when to use **red**/**black** trees instead.