# 7.2 Red Black Trees

**Proof of Lemma 15.**

**Induction on the height of $v$.**

**base case** ($\text{height}(v) = 0$)

- If $\text{height}(v)$ (maximum distance btw. $v$ and a node in the sub-tree rooted at $v$) is 0 then $v$ is a leaf.
- The black height of $v$ is 0.
- The sub-tree rooted at $v$ contains $0 = 2^{\text{bh}(v)} - 1$ inner vertices.

# 7.2 Red Black Trees

## Proof of Lemma 15.

**Induction on the height of $v$.**

**base case** ($\text{height}(v) = 0$)

- If $\text{height}(v)$ (maximum distance btw. $v$ and a node in the sub-tree rooted at $v$) is 0 then $v$ is a leaf.
- The black height of $v$ is 0.
- The sub-tree rooted at $v$ contains $0 = 2^{\text{bh}(v)} - 1$ inner vertices.

# 7.2 Red Black Trees

**Proof of Lemma 15.**

**Induction on the height of $v$.**

**base case** ($\mathrm{height}(v) = 0$)

- If $\mathrm{height}(v)$ (maximum distance btw. $v$ and a node in the sub-tree rooted at $v$) is $0$ then $v$ is a leaf.
- The black height of $v$ is $0$.
- The sub-tree rooted at $v$ contains $0 = 2^{\mathrm{bh}(v)} - 1$ inner vertices.

# 7.2 Red Black Trees

## Proof of Lemma 15.

**Induction on the height of $v$.**

**base case** ($\text{height}(v) = 0$)

- If $\text{height}(v)$ (maximum distance btw. $v$ and a node in the sub-tree rooted at $v$) is $0$ then $v$ is a leaf.
- The black height of $v$ is $0$.
- The sub-tree rooted at $v$ contains $0 = 2^{\text{bh}(v)} - 1$ inner vertices.

**Proof of Lemma 15.**

**Induction on the height of $v$.**

**base case** ($\text{height}(v) = 0$)

- If $\text{height}(v)$ (maximum distance btw. $v$ and a node in the sub-tree rooted at $v$) is $0$ then $v$ is a leaf.
- The black height of $v$ is $0$.
- The sub-tree rooted at $v$ contains $0 = 2^{\text{bh}(v)} - 1$ inner vertices.

# 7.2 Red Black Trees

## Proof (cont.)

**induction step**

$h \to h+1$

for every vertex $x$ with $\text{height}(x) \le h$

$\#\text{internal}(T_x) \ge 2^{\text{bh}(x)} - 1$

$h+1$

**Proof (cont.)**

**induction step**

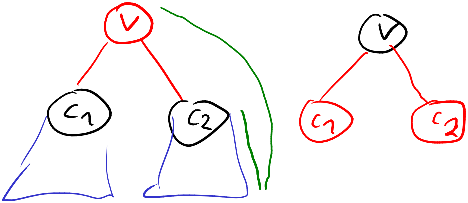
- Supose $v$ is a node with $\text{height}(v) > 0$.
- $v$ has two children with strictly smaller height.
- These children $(c_1, c_2)$ either have $\text{bh}(c_i) = \text{bh}(v)$ or $\text{bh}(c_i) = \text{bh}(v) - 1$.
- By induction hypothesis both sub-trees contain at least $2^{\text{bh}(v)-1} - 1$ internal vertices.
- Then $T_v$ contains at least $2(2^{\text{bh}(v)-1} - 1) + 1 \ge 2^{\text{bh}(v)} - 1$ vertices.

## Proof (cont.)

**induction step**

- Supose $v$ is a node with $\text{height}(v) > 0$.
- $v$ has two children with strictly smaller height.
- These children ($c_1$, $c_2$) either have $\text{bh}(c_i) = \text{bh}(v)$ or $\text{bh}(c_i) = \text{bh}(v) - 1$.
- By induction hypothesis both sub-trees contain at least $2^{\text{bh}(v)-1} - 1$ internal vertices.
- Then $T_v$ contains at least $2(2^{\text{bh}(v)-1} - 1) + 1 \geq 2^{\text{bh}(v)} - 1$ vertices.

# 7.2 Red Black Trees




### Proof (cont.)

**induction step**

- Supose $v$ is a node with $\text{height}(v) > 0$.
- $v$ has two children with strictly smaller height.
- These children ($c_1$, $c_2$) either have $\text{bh}(c_i) = \text{bh}(v)$ or $\text{bh}(c_i) = \text{bh}(v) - 1$.
- By induction hypothesis both sub-trees contain at least $2^{\text{bh}(v)-1} - 1$ internal vertices.
- Then $T_v$ contains at least $2(2^{\text{bh}(v)-1} - 1) + 1 \geq 2^{\text{bh}(v)} - 1$ vertices.

# 7.2 Red Black Trees

## Proof (cont.)

**induction step**

- Supose $v$ is a node with $\text{height}(v) > 0$.
- $v$ has two children with strictly smaller height.
- These children ($c_1$, $c_2$) either have $\text{bh}(c_i) = \text{bh}(v)$ or $\text{bh}(c_i) = \text{bh}(v) - 1$.
- By induction hypothesis both sub-trees contain at least $2^{\text{bh}(v)-1} - 1$ internal vertices.
- Then $T_v$ contains at least $2(2^{\text{bh}(v)-1} - 1) + 1 \geq 2^{\text{bh}(v)} - 1$ vertices.

# 7.2 Red Black Trees

## Proof (cont.)

**induction step**

- Supose $v$ is a node with $\text{height}(v) > 0$.
- $v$ has two children with strictly smaller height.
- These children ($c_1$, $c_2$) either have $\text{bh}(c_i) = \text{bh}(v)$ or $\text{bh}(c_i) = \text{bh}(v) - 1$.
- By induction hypothesis both sub-trees contain at least $2^{\text{bh}(v)-1} - 1$ internal vertices.
- Then $T_v$ contains at least $2(2^{\text{bh}(v)-1} - 1) + 1 \geq 2^{\text{bh}(v)} - 1$ vertices.

$\square$

# 7.2 Red Black Trees

**Proof of Lemma 13.**

Let $h$ denote the height of the red-black tree, and let $P$ denote a path from the root to the furthest leaf.

At least half of the node on $P$ must be black, since a red node must be followed by a black node.

Hence, the black height of the root is at least $h/2$.

The tree contains at least $2^{h/2} - 1$ internal vertices. Hence, $2^{h/2} - 1 \le n$.

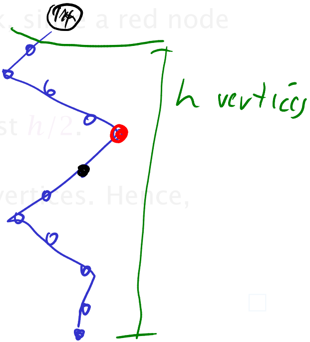
Hence, $h \le 2\log(n+1) = \mathcal{O}(\log n)$. □

# 7.2 Red Black Trees

## Proof of Lemma 13.

Let $h$ denote the height of the red-black tree, and let $P$ denote a path from the root to the furthest leaf.

At least half of the node on $P$ must be black, since a red node must be followed by a black node.




Hence, the black height of the root is at least $h/2$.

The tree contains at least $2^{h/2}-1$ internal vertices. Hence, $2^{h/2}-1 \le n$.

Hence, $h \le 2\log(n+1) = \mathcal{O}(\log n)$. □

# 7.2 Red Black Trees

## Proof of Lemma 13.

Let $h$ denote the height of the red-black tree, and let $P$ denote a path from the root to the furthest leaf.

At least half of the node on $P$ must be black, since a red node must be followed by a black node.

Hence, the black height of the root is at least $h/2$.

The tree contains at least $2^{h/2} - 1$ internal vertices. Hence, $2^{h/2} - 1 \leq n$.

Hence, $h \leq 2\log(n+1) = \mathcal{O}(\log n)$. □

# 7.2 Red Black Trees

## Proof of Lemma 13.

Let $h$ denote the height of the red-black tree, and let $P$ denote a path from the root to the furthest leaf.

At least half of the node on $P$ must be black, since a red node must be followed by a black node.

Hence, the black height of the root is at least $h/2$.

The tree contains at least $2^{h/2} - 1$ internal vertices. Hence, $2^{h/2} - 1 \le n$.

Hence, $h \le 2\log(n+1) = \mathcal{O}(\log n)$. □

# 7.2 Red Black Trees

**Proof of Lemma 13.**

Let $h$ denote the height of the red-black tree, and let $P$ denote a path from the root to the furthest leaf.

At least half of the node on $P$ must be black, since a red node must be followed by a black node.

Hence, the black height of the root is at least $h/2$.

The tree contains at least $2^{h/2} - 1$ internal vertices. Hence, $2^{h/2} - 1 \leq n$.

Hence, $h \leq 2\log(n+1) = \mathcal{O}(\log n)$. □

# 7.2 Red Black Trees

## Proof of Lemma 13.

Let $h$ denote the height of the red-black tree, and let $P$ denote a path from the root to the furthest leaf.

At least half of the node on $P$ must be black, since a red node must be followed by a black node.

Hence, the black height of the root is at least $h/2$.

The tree contains at least $2^{h/2} - 1$ internal vertices. Hence, $2^{h/2} - 1 \le n$.

Hence, $h \le 2\log(n+1) = \mathcal{O}(\log n)$. □

# 7.2 Red Black Trees

**Definition 1**

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a color, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

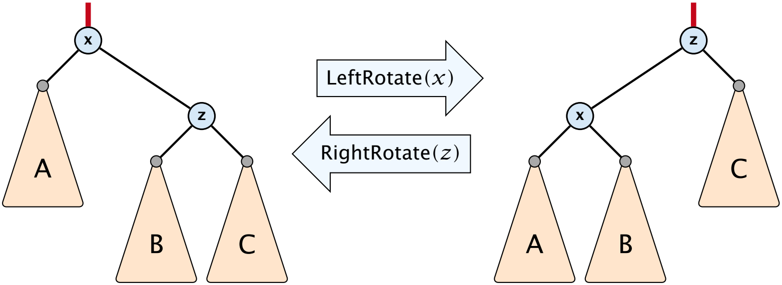
The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data.

# 7.2 Red Black Trees

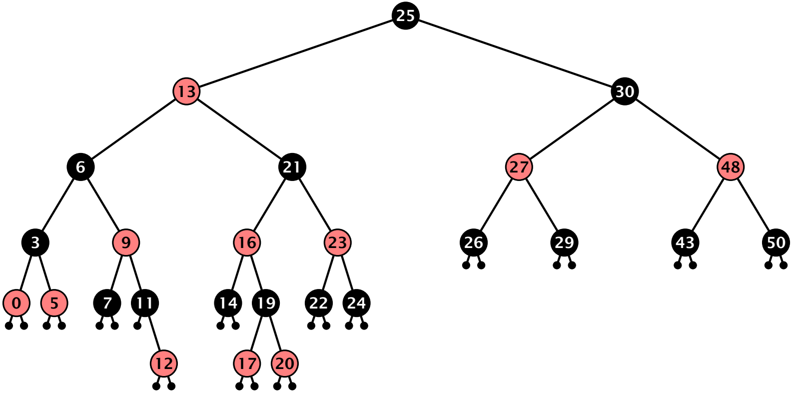
We need to adapt the insert and delete operations so that the red black properties are maintained.

# Rotations

The properties will be maintained through rotations:



# Red Black Trees: Insert

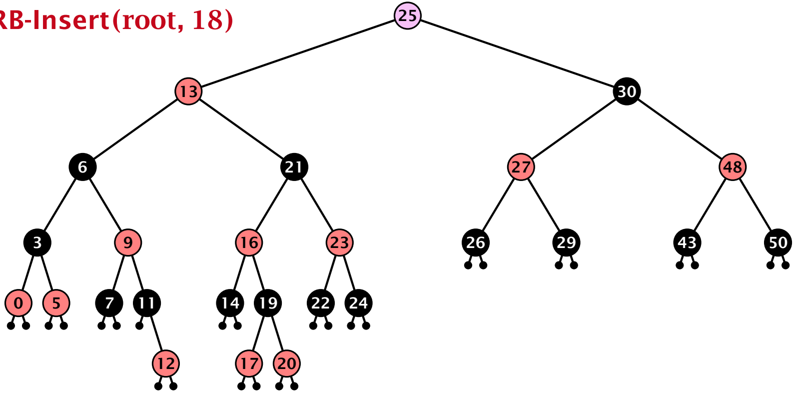



**Insert:**

- first make a normal insert into a binary search tree
- then fix red-black properties

# Red Black Trees: Insert

**RB-Insert(root, 18)**

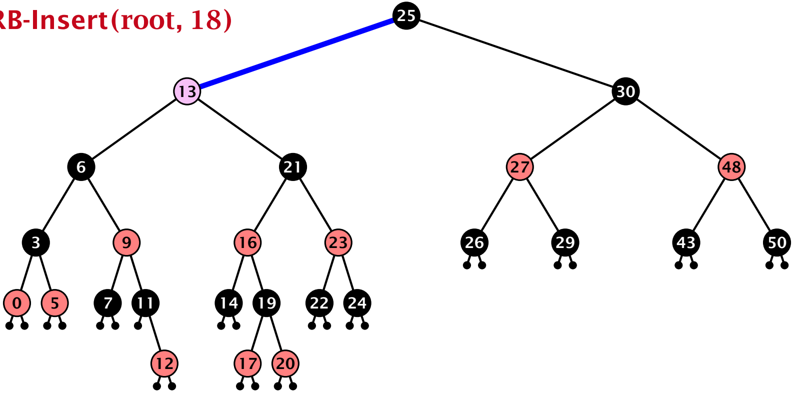



**Insert:**

- first make a normal insert into a binary search tree
- then fix red-black properties

# Red Black Trees: Insert

**RB-Insert(root, 18)**

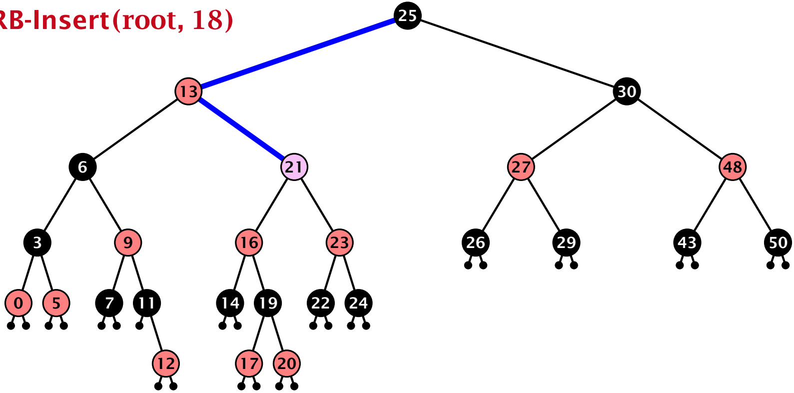



**Insert:**

- first make a normal insert into a binary search tree
- then fix red-black properties

# Red Black Trees: Insert

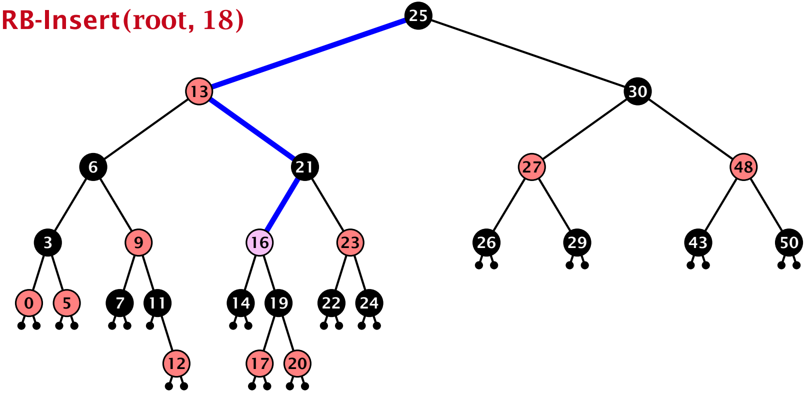
**RB-Insert(root, 18)**




**Insert:**

- first make a normal insert into a binary search tree
- then fix red-black properties

# Red Black Trees: Insert

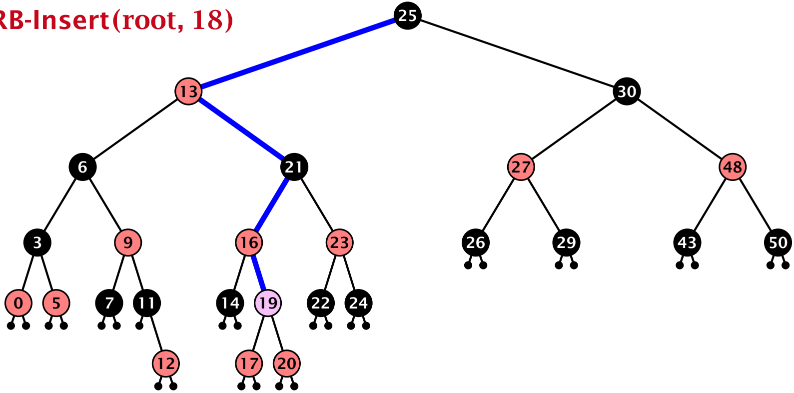



**Insert:**

- first make a normal insert into a binary search tree
- then fix red-black properties

# Red Black Trees: Insert

RB-Insert(root, 18)

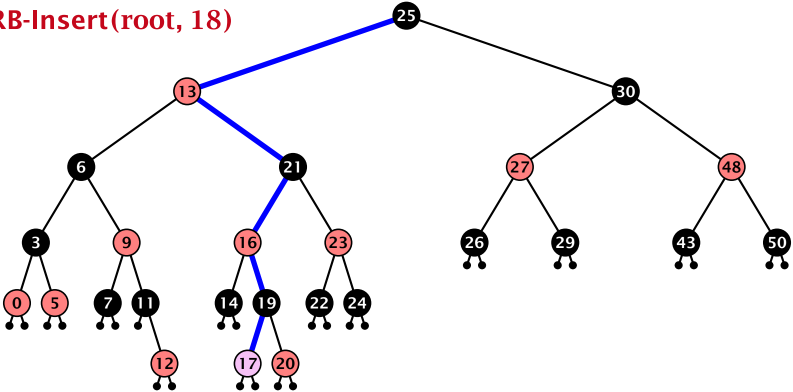



**Insert:**

- first make a normal insert into a binary search tree
- then fix red-black properties

# Red Black Trees: Insert

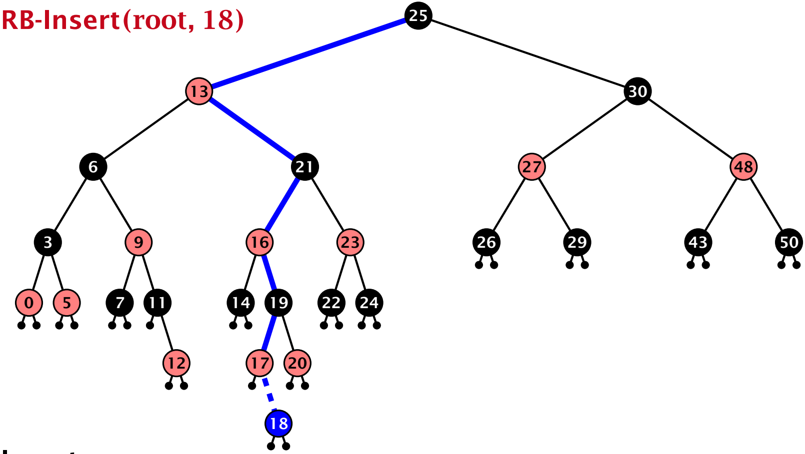
**RB-Insert(root, 18)**




**Insert:**

- first make a normal insert into a binary search tree
- then fix red-black properties

# Red Black Trees: Insert

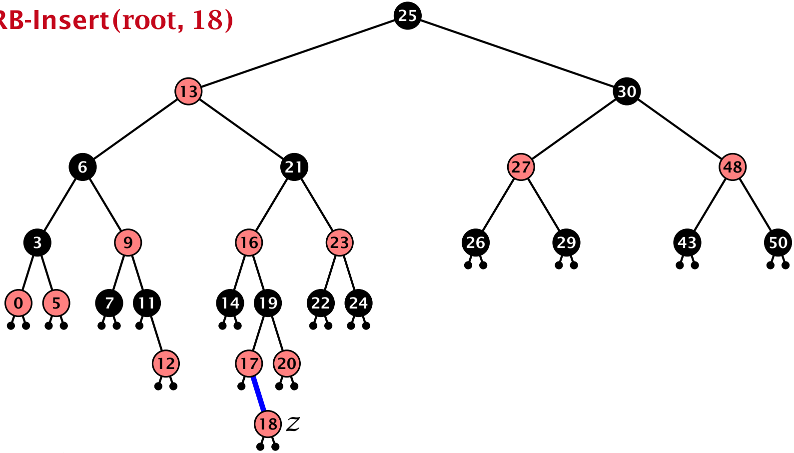



**Insert:**

- first make a normal insert into a binary search tree
- then fix red-black properties

# Red Black Trees: Insert

**RB-Insert(root, 18)**




**Insert:**

- first make a normal insert into a binary search tree
- then fix red-black properties

# Red Black Trees: Insert

**Invariant of the fix-up algorithm:**

- $z$ is a red node
- the black-height property is fulfilled at every node
- the only violation of red-black properties occurs at $z$ and $\text{parent}[z]$
  - either both of them are red (most important case)
  - or the parent does not exist (violation since root must be black)

If $z$ has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

# Red Black Trees: Insert

**Invariant of the fix-up algorithm:**

- $z$ is a red node
- the black-height property is fulfilled at every node
- the only violation of red-black properties occurs at $z$ and $\text{parent}[z]$
  - either both of them are red (most important case)
  - or the parent does not exist (violation since root must be black)

If $z$ has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

# Red Black Trees: Insert

**Invariant of the fix-up algorithm:**

- $z$ is a red node
- the black-height property is fulfilled at every node
- the only violation of red-black properties occurs at $z$ and $\text{parent}[z]$
  - either both of them are red (most important case)
  - or the parent does not exist (violation since root must be black)

If $z$ has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

# Red Black Trees: Insert

**Invariant of the fix-up algorithm:**

- $z$ is a red node
- the black-height property is fulfilled at every node
- the only violation of red-black properties occurs at $z$ and $\text{parent}[z]$
  - either both of them are red
    (most important case)
  - or the parent does not exist
    (violation since root must be black)

If $z$ has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

# Red Black Trees: Insert

**Invariant of the fix-up algorithm:**

- $z$ is a red node
- the black-height property is fulfilled at every node
- the only violation of red-black properties occurs at $z$ and $\text{parent}[z]$
  - either both of them are red
    (most important case)
  - or the parent does not exist
    (violation since root must be black)

If $z$ has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

# Red Black Trees: Insert

**Invariant of the fix-up algorithm:**

- $z$ is a red node
- the black-height property is fulfilled at every node
- the only violation of red-black properties occurs at $z$ and $\text{parent}[z]$
  - either both of them are red (most important case)
  - or the parent does not exist (violation since root must be black)

If $z$ has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

# Red Black Trees: Insert

**Algorithm 10** InsertFix($z$)

```
while parent[z] ≠ null and col[parent[z]] = red do
    if parent[z] = left[gp[z]] then
        uncle ← right[grandparent[z]]
        if col[uncle] = red then
            col[p[z]] ← black; col[u] ← black;
            col[gp[z]] ← red; z ← grandparent[z];
        else
            if z = right[parent[z]] then
                z ← p[z]; LeftRotate(z);
            col[p[z]] ← black; col[gp[z]] ← red;
            RightRotate(gp[z]);
    else same as then-clause but right and left exchanged
col(root[T]) ← black;
```

# Red Black Trees: Insert

**Algorithm 10** InsertFix($z$)

```
while parent[z] ≠ null and col[parent[z]] = red do
    if parent[z] = left[gp[z]] then    z in left subtree of grandparent
        uncle ← right[grandparent[z]]
        if col[uncle] = red then
            col[p[z]] ← black; col[u] ← black;
            col[gp[z]] ← red; z ← grandparent[z];
        else
            if z = right[parent[z]] then
                z ← p[z]; LeftRotate(z);
            col[p[z]] ← black; col[gp[z]] ← red;
            RightRotate(gp[z]);
    else same as then-clause but right and left exchanged
col(root[T]) ← black;
```

# Red Black Trees: Insert

**Algorithm 10** InsertFix($z$)

```
while parent[z] ≠ null and col[parent[z]] = red do
    if parent[z] = left[gp[z]] then
        uncle ← right[grandparent[z]]
        if col[uncle] = red then                              Case 1: uncle red
            col[p[z]] ← black; col[u] ← black;
            col[gp[z]] ← red; z ← grandparent[z];
        else
            if z = right[parent[z]] then
                z ← p[z]; LeftRotate(z);
            col[p[z]] ← black; col[gp[z]] ← red;
            RightRotate(gp[z]);
    else same as then-clause but right and left exchanged
col(root[T]) ← black;
```

# Red Black Trees: Insert

**Algorithm 10** InsertFix($z$)

```
while parent[z] ≠ null and col[parent[z]] = red do
    if parent[z] = left[gp[z]] then
        uncle ← right[grandparent[z]]
        if col[uncle] = red then
            col[p[z]] ← black; col[u] ← black;
            col[gp[z]] ← red; z ← grandparent[z];
        else                                    Case 2: uncle black
            if z = right[parent[z]] then
                z ← p[z]; LeftRotate(z);
            col[p[z]] ← black; col[gp[z]] ← red;
            RightRotate(gp[z]);
    else same as then-clause but right and left exchanged
col(root[T]) ← black;
```

# Red Black Trees: Insert

**Algorithm 10** InsertFix($z$)

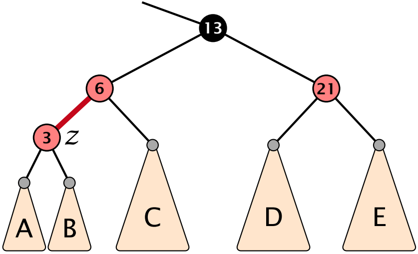
```
while parent[z] ≠ null and col[parent[z]] = red do
    if parent[z] = left[gp[z]] then
        uncle ← right[grandparent[z]]
        if col[uncle] = red then
            col[p[z]] ← black; col[u] ← black;
            col[gp[z]] ← red; z ← grandparent[z];
        else
            if z = right[parent[z]] then          2a: z right child
                z ← p[z]; LeftRotate(z);
            col[p[z]] ← black; col[gp[z]] ← red;
            RightRotate(gp[z]);
    else same as then-clause but right and left exchanged
col(root[T]) ← black;
```

# Red Black Trees: Insert

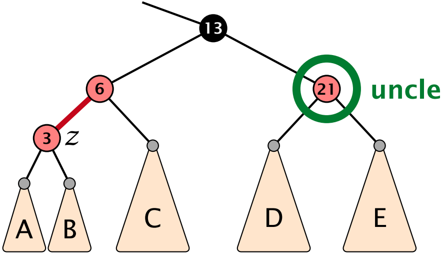
**Algorithm 10** InsertFix($z$)

```
while parent[z] ≠ null and col[parent[z]] = red do
    if parent[z] = left[gp[z]] then
        uncle ← right[grandparent[z]]
        if col[uncle] = red then
            col[p[z]] ← black; col[u] ← black;
            col[gp[z]] ← red; z ← grandparent[z];
        else
            if z = right[parent[z]] then
                z ← p[z]; LeftRotate(z);
            col[p[z]] ← black; col[gp[z]] ← red;   2b: z left child
            RightRotate(gp[z]);
    else same as then-clause but right and left exchanged
col(root[T]) ← black;
```

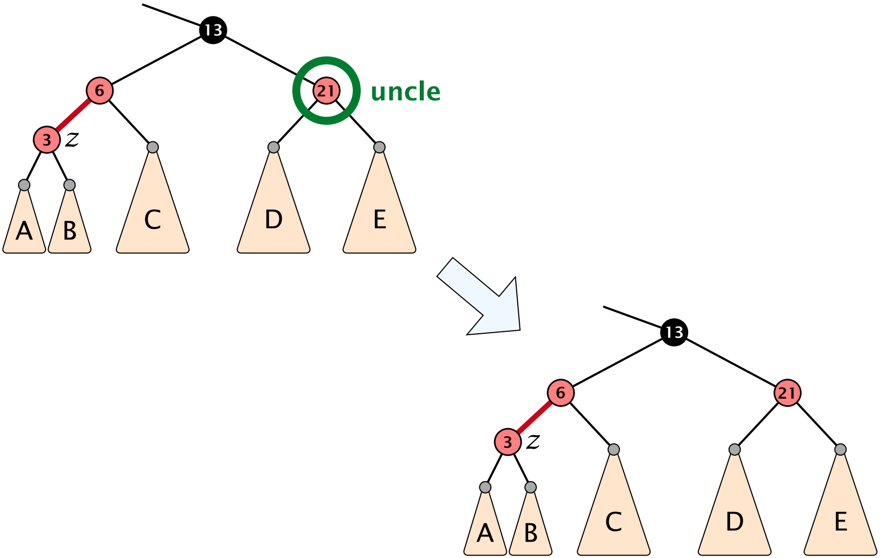
# Case 1: Red Uncle



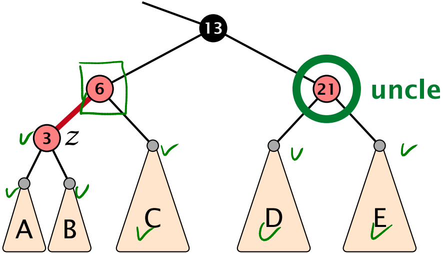
# Case 1: Red Uncle



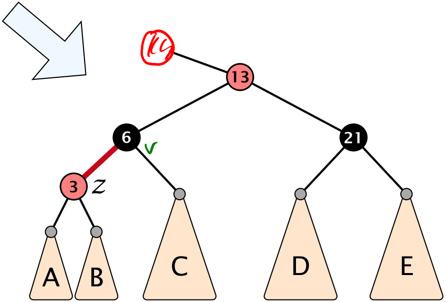
# Case 1: Red Uncle



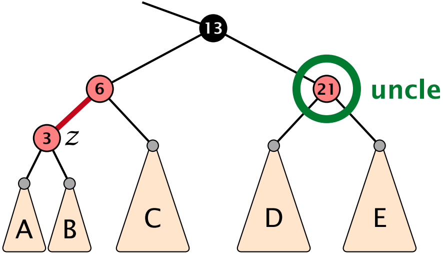
# Case 1: Red Uncle



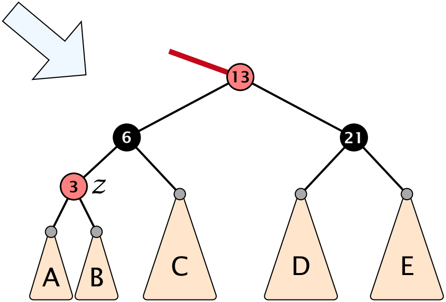

1. recolour



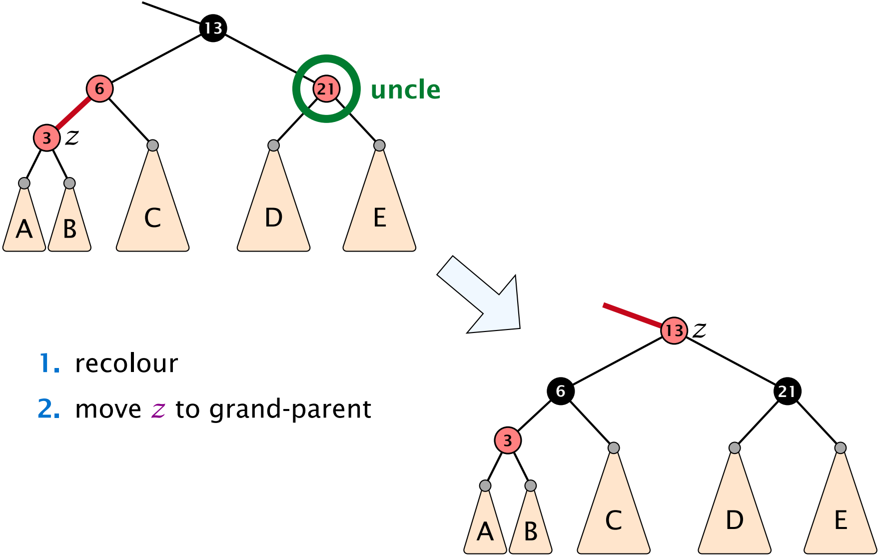
# Case 1: Red Uncle




1. recolour

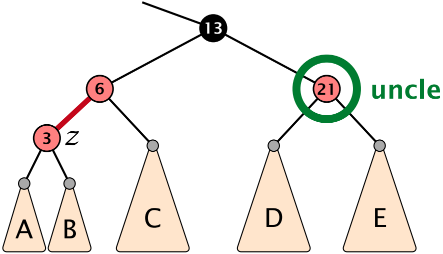


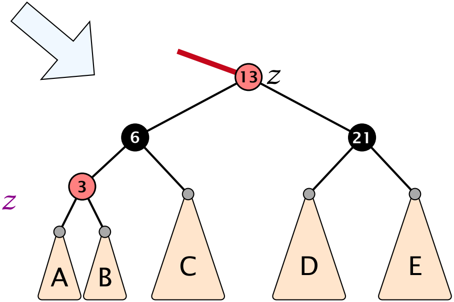
# Case 1: Red Uncle




1. recolour
2. move $z$ to grand-parent

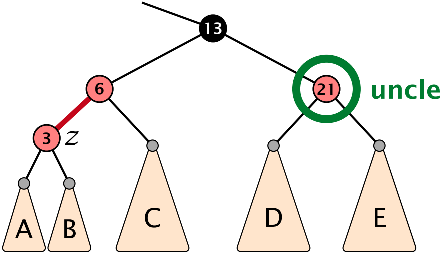
# Case 1: Red Uncle

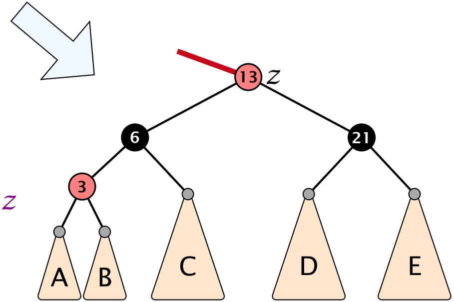



1. recolour
2. move $z$ to grand-parent
3. invariant is fulfilled for new $z$

# Case 1: Red Uncle



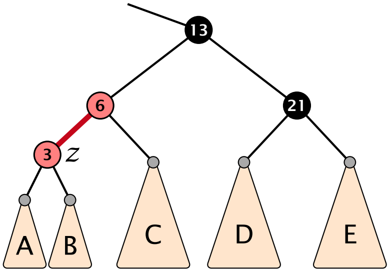




1. recolour
2. move $z$ to grand-parent
3. invariant is fulfilled for new $z$
4. you made progress

# Case 2b: Black uncle and $z$ is left child

1. rotate around grandparent
2. re-colour to ensure that black height property holds
3. you have a red black tree

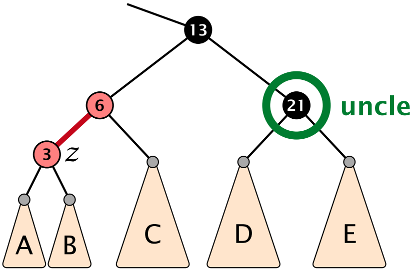


# Case 2b: Black uncle and $z$ is left child

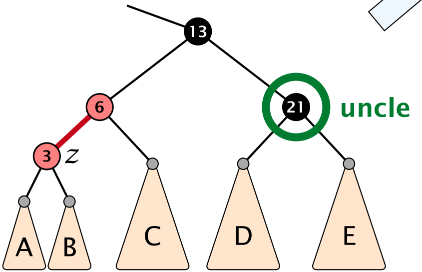
1. rotate around grandparent
2. re-colour to ensure that black height property holds
3. you have a red black tree

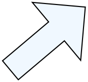


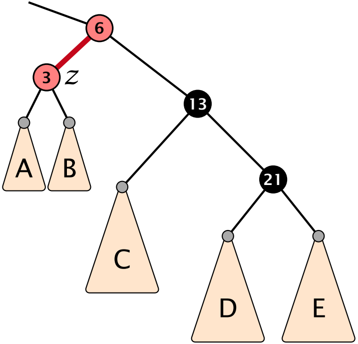



# Case 2b: Black uncle and $z$ is left child

1. rotate around grandparent
2. re-colour to ensure that black height property holds
3. you have a red black tree

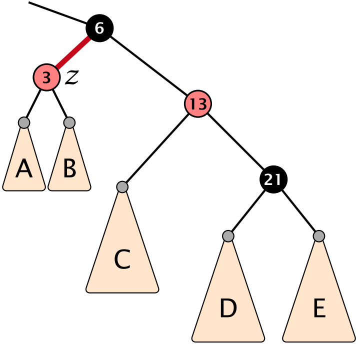


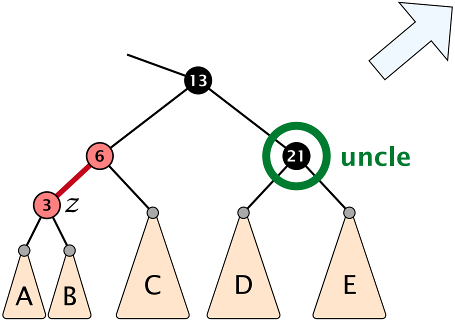





# Case 2b: Black uncle and $z$ is left child

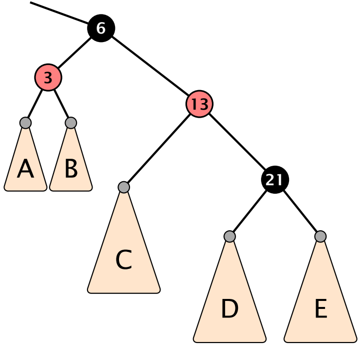
1. rotate around grandparent
2. re-colour to ensure that black height property holds
3. you have a red black tree

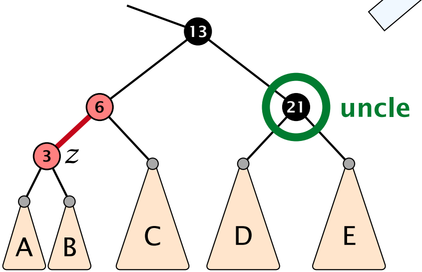


# Case 2b: Black uncle and $z$ is left child

1. rotate around grandparent
2. re-colour to ensure that black height property holds
3. you have a red black tree

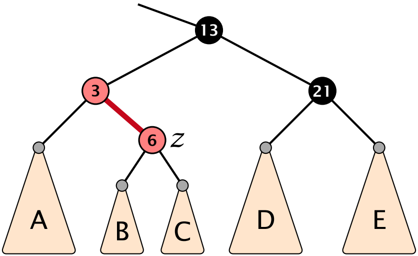


# Case 2a: Black uncle and $z$ is right child

1. rotate around parent
2. move $z$ downwards
3. you have Case 2b.

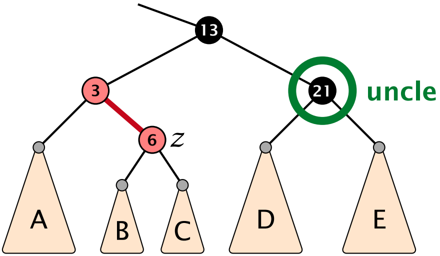


# Case 2a: Black uncle and $z$ is right child

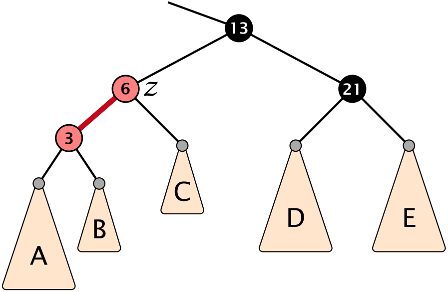
1. rotate around parent
2. move $z$ downwards
3. you have Case 2b.

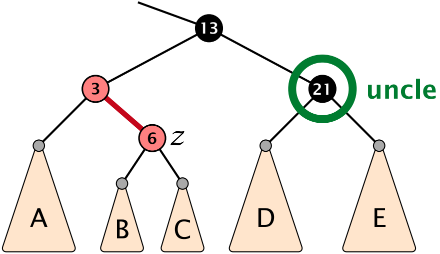


# Case 2a: Black uncle and $z$ is right child

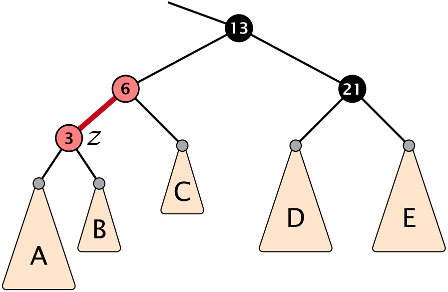
1. rotate around parent
2. move $z$ downwards
3. you have Case 2b.

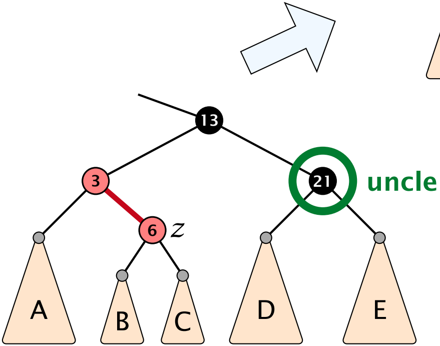


# Case 2a: Black uncle and $z$ is right child

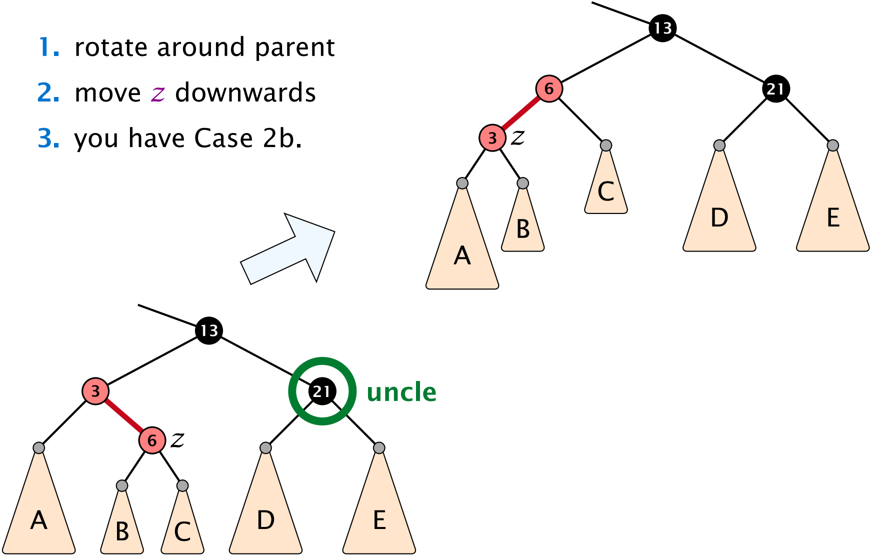
1. rotate around parent
2. move $z$ downwards
3. you have Case 2b.



# Case 2a: Black uncle and $z$ is right child

1. rotate around parent
2. move $z$ downwards
3. you have Case 2b.



# Red Black Trees: Insert

**Running time:**

- Only Case 1 may repeat; but only $h/2$ many steps, where $h$ is the height of the tree.
- Case 2a → Case 2b → red-black tree
- Case 2b → red-black tree

Performing Case 1 at most $\mathcal{O}(\log n)$ times and every other case at most once, we get a red-black tree. Hence $\mathcal{O}(\log n)$ re-colorings and at most 2 rotations.

# Red Black Trees: Insert

**Running time:**

- Only Case 1 may repeat; but only $h/2$ many steps, where $h$ is the height of the tree.
- Case 2a $\rightarrow$ Case 2b $\rightarrow$ red-black tree
- Case 2b $\rightarrow$ red-black tree

Performing Case 1 at most $\mathcal{O}(\log n)$ times and every other case at most once, we get a red-black tree. Hence $\mathcal{O}(\log n)$ re-colorings and at most 2 rotations.

# Red Black Trees: Insert

**Running time:**

- Only Case 1 may repeat; but only $h/2$ many steps, where $h$ is the height of the tree.
- Case 2a $\rightarrow$ Case 2b $\rightarrow$ red-black tree
- Case 2b $\rightarrow$ red-black tree

Performing Case 1 at most $\mathcal{O}(\log n)$ times and every other case at most once, we get a red-black tree. Hence $\mathcal{O}(\log n)$ re-colorings and at most 2 rotations.

# Red Black Trees: Insert

**Running time:**

- Only Case 1 may repeat; but only $h/2$ many steps, where $h$ is the height of the tree.
- Case 2a $\rightarrow$ Case 2b $\rightarrow$ red-black tree
- Case 2b $\rightarrow$ red-black tree

Performing Case 1 at most $\mathcal{O}(\log n)$ times and every other case at most once, we get a red-black tree. Hence $\mathcal{O}(\log n)$ re-colorings and at most 2 rotations.

# Red Black Trees: Delete

First do a standard delete.

If the spliced out node $x$ was red everything is fine.

If it was black there may be the following problems.

- Parent and child of $x$ were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of $x$ to a descendant leaf of $x$ changes the number of black nodes. Black height property might be violated.

# Red Black Trees: Delete

First do a standard delete.

If the spliced out node $x$ was red everything is fine.

If it was black there may be the following problems.

- Parent and child of $x$ were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of $x$ to a descendant leaf of $x$ changes the number of black nodes. Black height property might be violated.

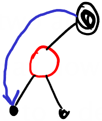
# Red Black Trees: Delete

First do a standard delete.

If the spliced out node $x$ was red everything is fine.

If it was black there may be the following problems.

- Parent and child of $x$ were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of $x$ to a descendant leaf of $x$ changes the number of black nodes. Black height property might be violated.



# Red Black Trees: Delete

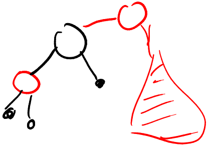
First do a standard delete.

If the spliced out node $x$ was red everything is fine.

If it was black there may be the following problems.

- Parent and child of $x$ were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of $x$ to a descendant leaf of $x$ changes the number of black nodes. Black height property might be violated.

# Red Black Trees: Delete



First do a standard delete.

If the spliced out node $x$ was red everything is fine.

If it was black there may be the following problems.

- Parent and child of $x$ were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of $x$ to a descendant leaf of $x$ changes the number of black nodes. Black height property might be violated.

# Red Black Trees: Delete

First do a standard delete.

If the spliced out node $x$ was red everything is fine.

If it was black there may be the following problems.

- Parent and child of $x$ were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of $x$ to a descendant leaf of $x$ changes the number of black nodes. Black height property might be violated.

# Red Black Trees: Delete

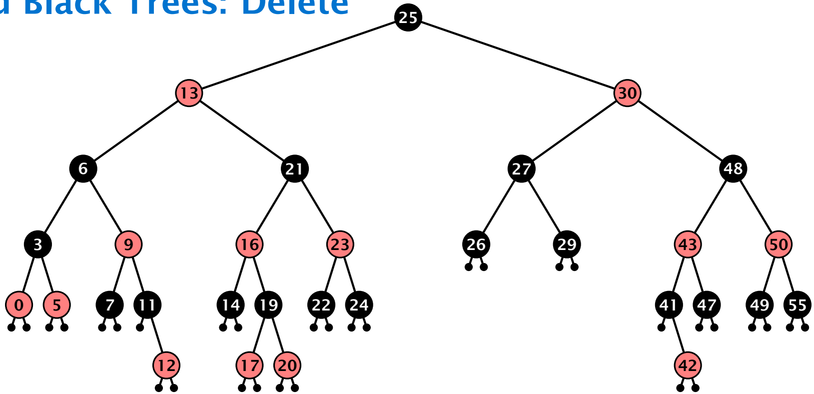
First do a standard delete.

If the spliced out node $x$ was red everything is fine.

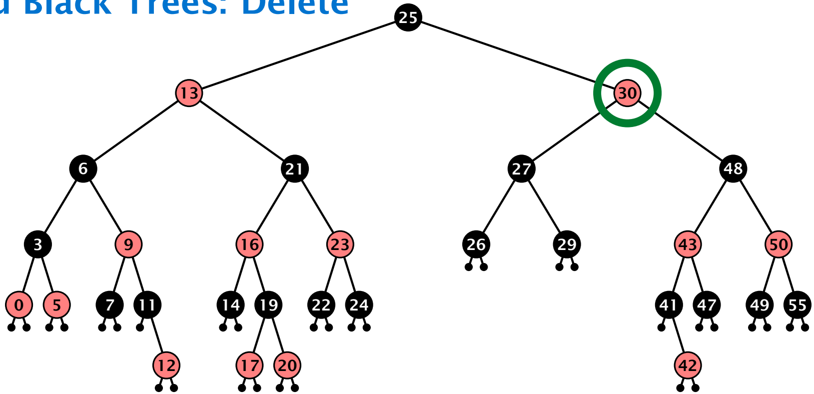
If it was black there may be the following problems.

- Parent and child of $x$ were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of $x$ to a descendant leaf of $x$ changes the number of black nodes. Black height property might be violated.

# Red Black Trees: Delete



# Red Black Trees: Delete

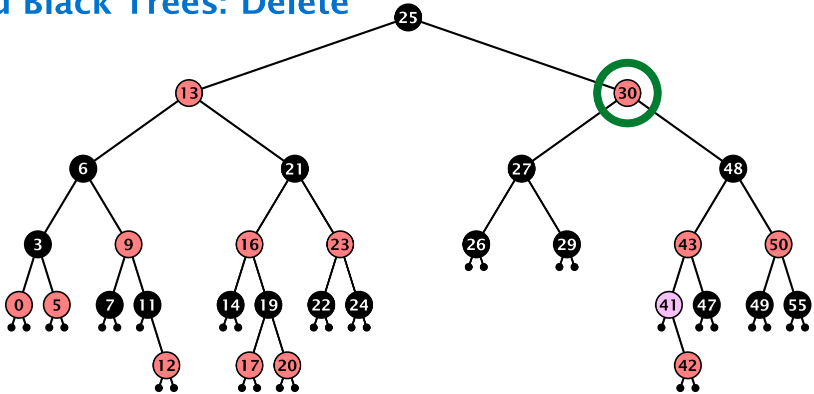



Case 3:

Element has two children

- do normal delete
- when replacing content by content of successor, don't change color of node

# Red Black Trees: Delete

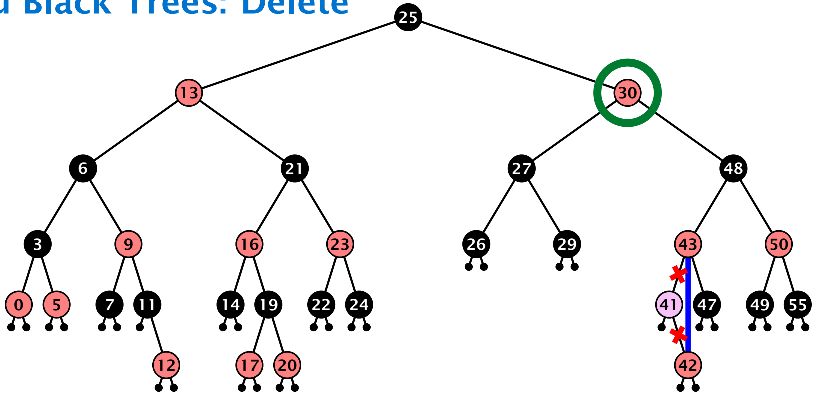



Case 3:

Element has two children

- do normal delete
- when replacing content by content of successor, don't change color of node

# Red Black Trees: Delete

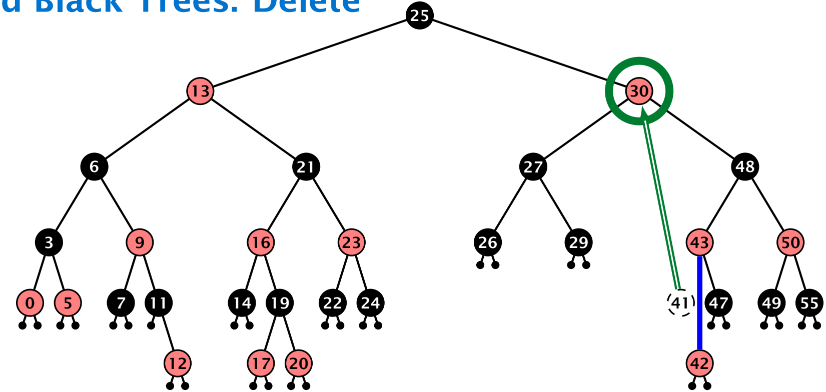



Case 3:

Element has two children

- do normal delete
- when replacing content by content of successor, don't change color of node

# Red Black Trees: Delete

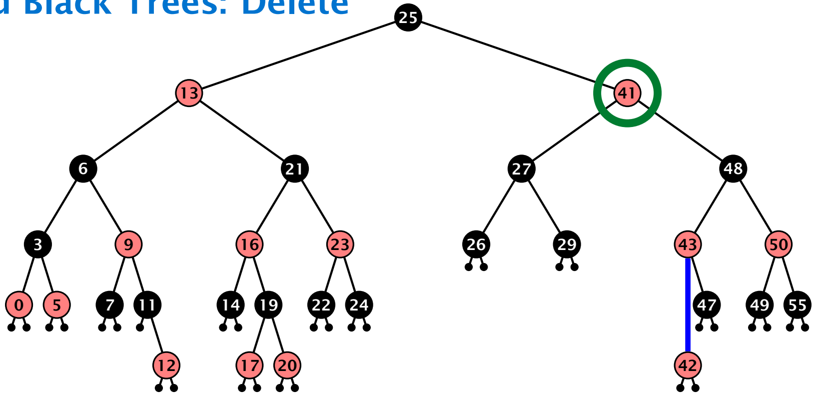



Case 3:

Element has two children

- do normal delete
- when replacing content by content of successor, don't change color of node

# Red Black Trees: Delete

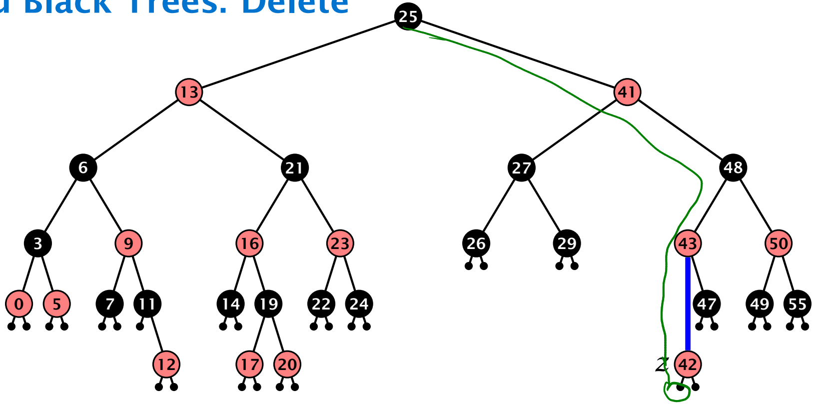



Case 3:

Element has two children

- do normal delete
- when replacing content by content of successor, don't change color of node

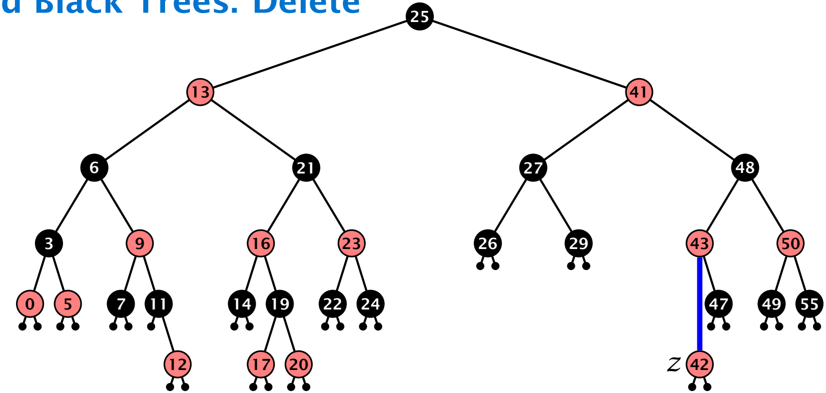
# Red Black Trees: Delete




Delete:

- deleting black node messes up black-height property
- if $z$ is red, we can simply color it black and everything is fine
- the problem is if $z$ is black (e.g. a dummy-leaf); we call a fix-up procedure to fix the problem.

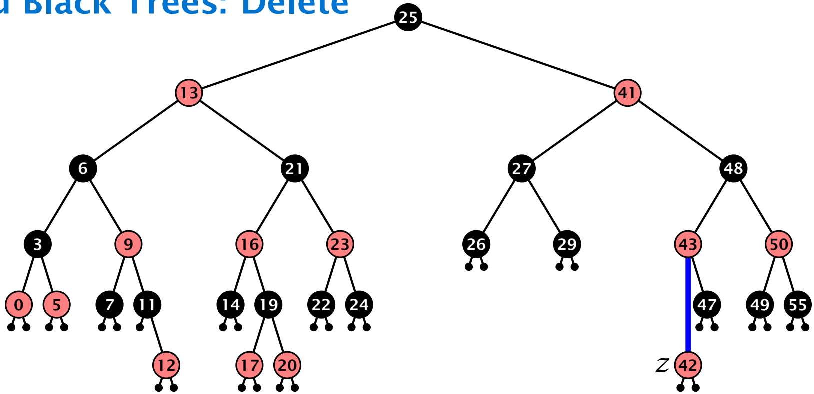
# Red Black Trees: Delete




Delete:

- deleting black node messes up black-height property
- if $z$ is red, we can simply color it black and everything is fine
- the problem is if $z$ is black (e.g. a dummy-leaf); we call a fix-up procedure to fix the problem.

# Red Black Trees: Delete




Delete:

- deleting black node messes up black-height property
- if $z$ is red, we can simply color it black and everything is fine
- the problem is if $z$ is black (e.g. a dummy-leaf); we call a fix-up procedure to fix the problem.

# Red Black Trees: Delete

**Invariant of the fix-up algorithm**

- the node $z$ is black
- if we “assign” a fake black unit to the edge from $z$ to its parent then the black-height property is fulfilled

**Goal:** make rotations in such a way that you at some point can remove the fake black unit from the edge.

# Red Black Trees: Delete

**Invariant of the fix-up algorithm**

- the node $z$ is black
- if we “assign” a fake black unit to the edge from $z$ to its parent then the black-height property is fulfilled

**Goal:** make rotations in such a way that you at some point can remove the fake black unit from the edge.

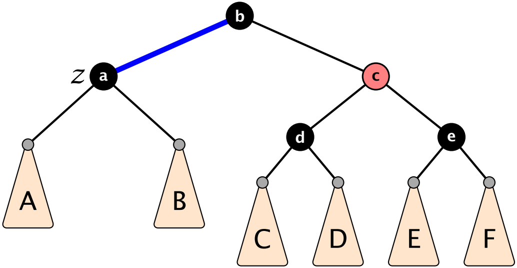
# Red Black Trees: Delete

**Invariant of the fix-up algorithm**

- the node $z$ is black
- if we “assign” a fake black unit to the edge from $z$ to its parent then the black-height property is fulfilled

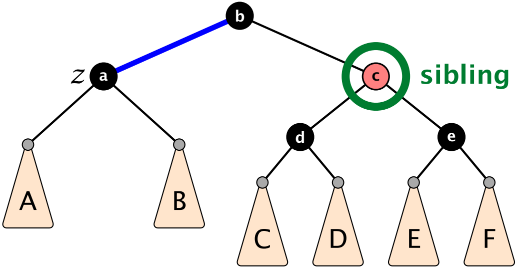
**Goal:** make rotations in such a way that you at some point can remove the fake black unit from the edge.

# Case 1: Sibling of $z$ is red



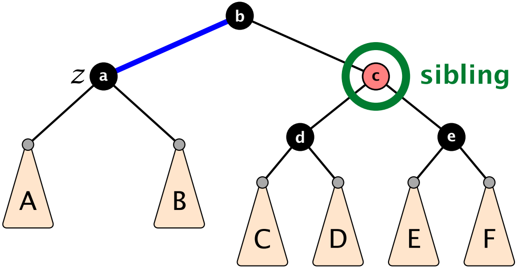

1. left-rotate around parent of $z$
2. recolor nodes $b$ and $c$
3. the new sibling is black
   (and parent of z is red)
4. Case 2 (special),
   or Case 3, or Case 4

# Case 1: Sibling of $z$ is red

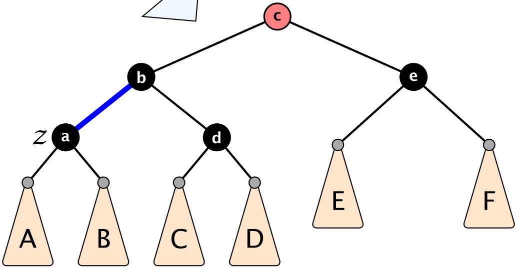



1. left-rotate around parent of $z$
2. recolor nodes $b$ and $c$
3. the new sibling is black
   (and parent of z is red)
4. Case 2 (special),
   or Case 3, or Case 4

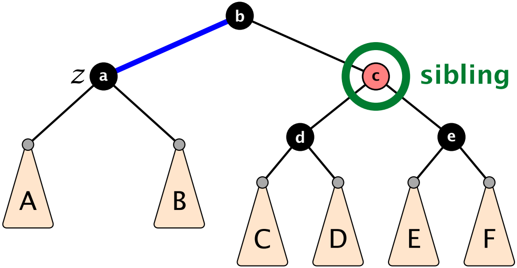
# Case 1: Sibling of $z$ is red



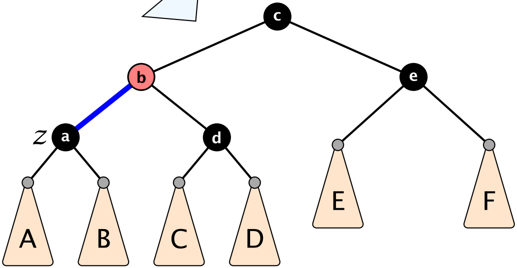

1. left-rotate around parent of $z$
2. recolor nodes $b$ and $c$
3. the new sibling is black (and parent of z is red)
4. Case 2 (special), or Case 3, or Case 4



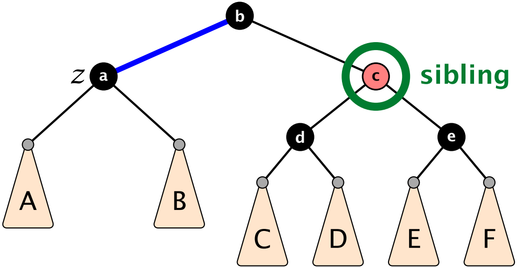
# Case 1: Sibling of $z$ is red



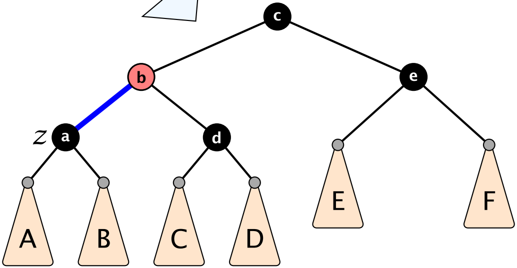

1. left-rotate around parent of $z$
2. recolor nodes $b$ and $c$
3. the new sibling is black
   (and parent of z is red)
4. Case 2 (special),
   or Case 3, or Case 4



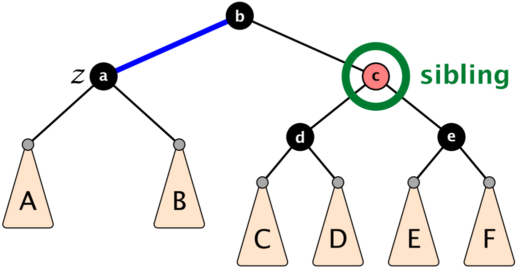
# Case 1: Sibling of $z$ is red



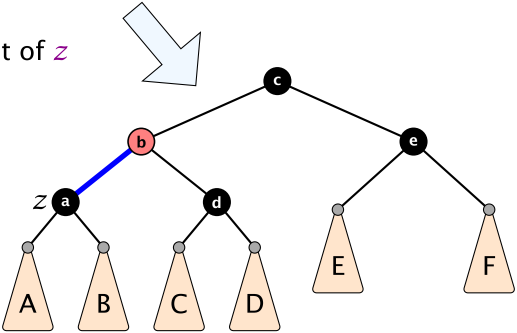

1. left-rotate around parent of $z$
2. recolor nodes $b$ and $c$
3. the new sibling is black (and parent of z is red)
4. Case 2 (special), or Case 3, or Case 4



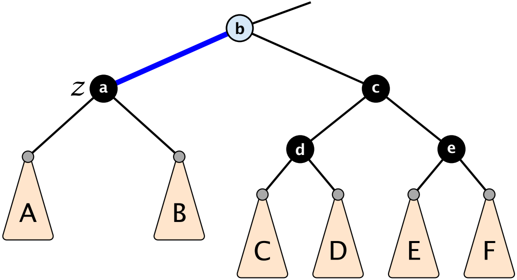
# Case 1: Sibling of $z$ is red




1. left-rotate around parent of $z$
2. recolor nodes $b$ and $c$
3. the new sibling is black
   (and parent of z is red)
4. Case 2 (special),
   or Case 3, or Case 4

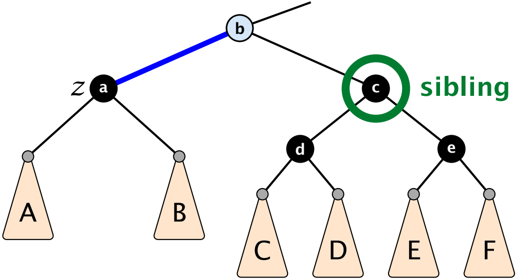


# Case 2: Sibling is black with two black children



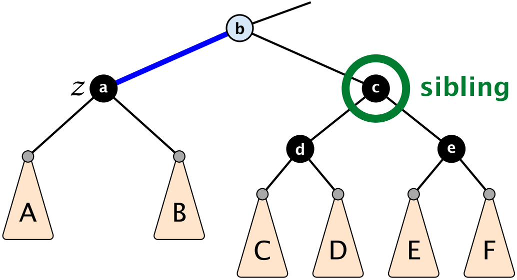

1. re-color node $c$
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if $b$ is red we color it black and are done

# Case 2: Sibling is black with two black children

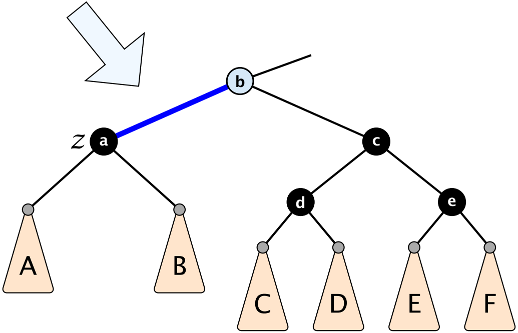



1. re-color node $c$
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if $b$ is red we color it black and are done

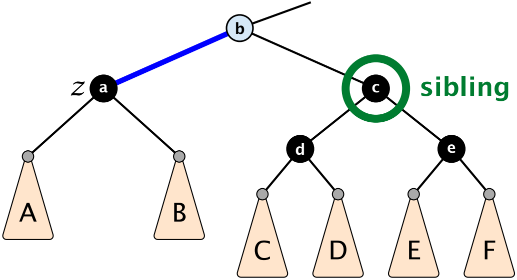
# Case 2: Sibling is black with two black children



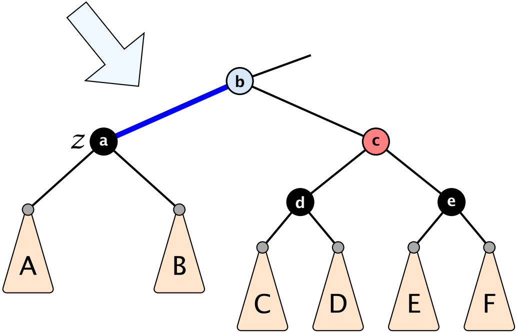

1. re-color node $c$
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if $b$ is red we color it black and are done



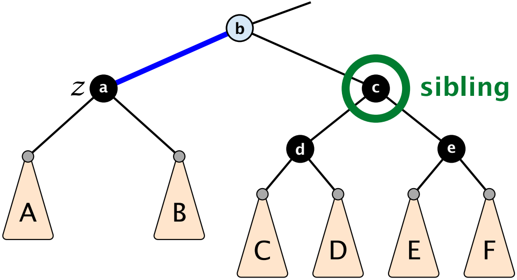
# Case 2: Sibling is black with two black children



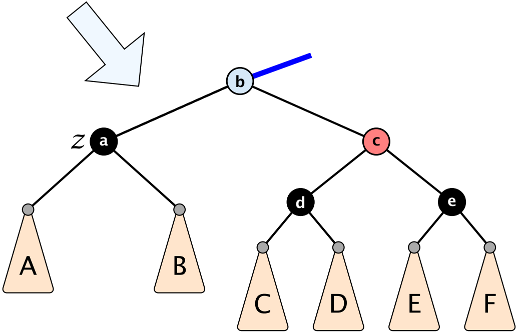

1. re-color node $c$
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if $b$ is red we color it black and are done



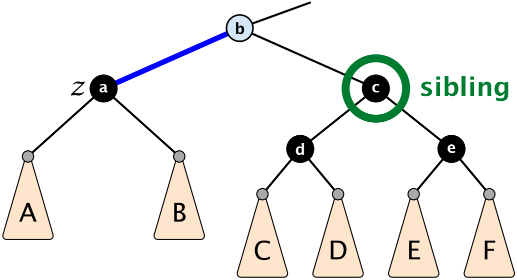
# Case 2: Sibling is black with two black children



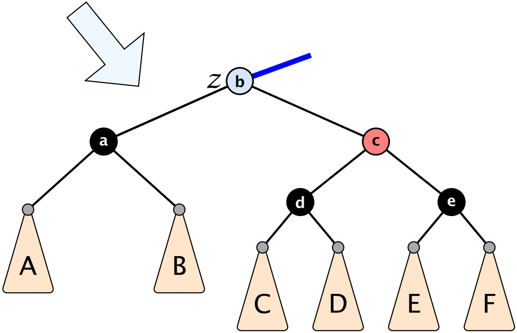

1. re-color node $c$
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if $b$ is red we color it black and are done



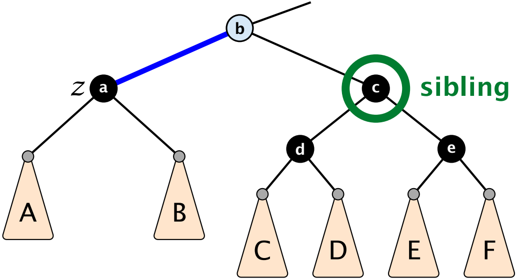
# Case 2: Sibling is black with two black children

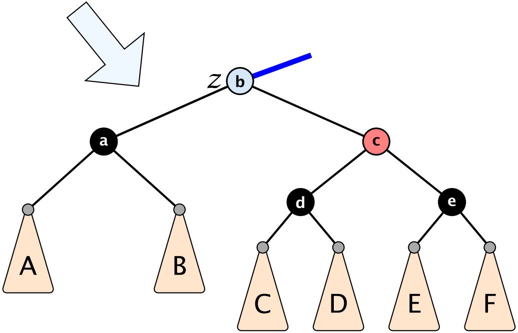



1. re-color node $c$
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if $b$ is red we color it black and are done



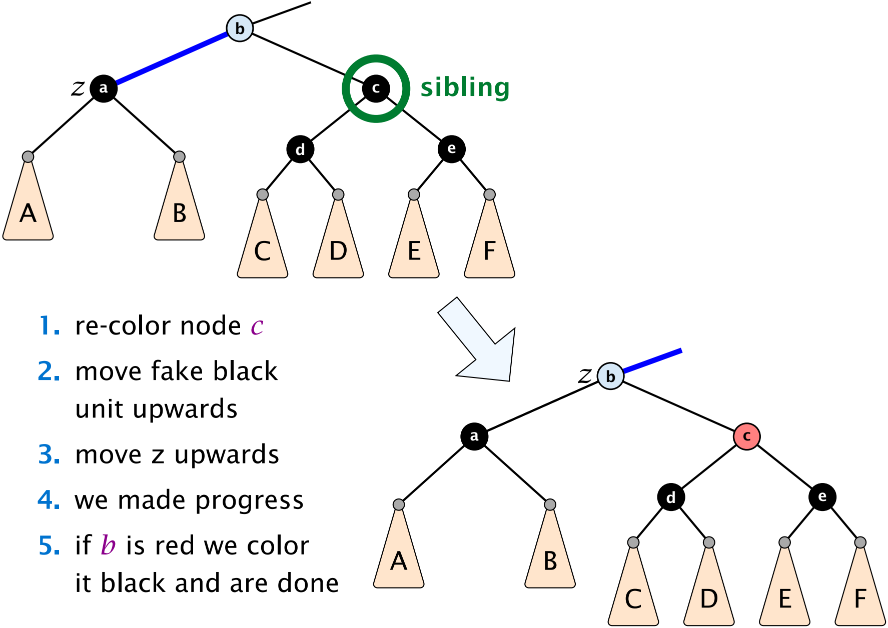
# Case 2: Sibling is black with two black children




1. re-color node $c$
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if $b$ is red we color it black and are done

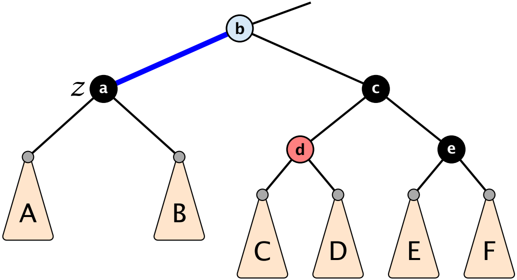
# Case 2: Sibling is black with two black children




1. re-color node $c$
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if $b$ is red we color it black and are done

# Case 3: Sibling black with one black child to the right

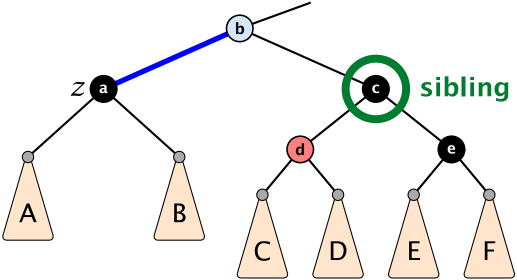
1. do a right-rotation at sibling
2. recolor $c$ and $d$
3. new sibling is black with red right child (Case 4)



# Case 3: Sibling black with one black child to the right

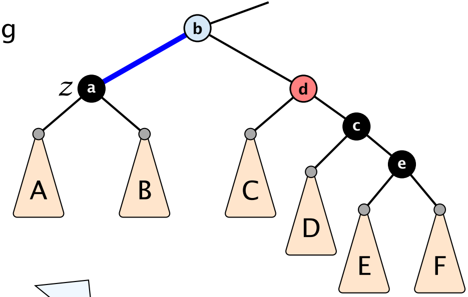
1. do a right-rotation at sibling
2. recolor $c$ and $d$
3. new sibling is black with red right child (Case 4)

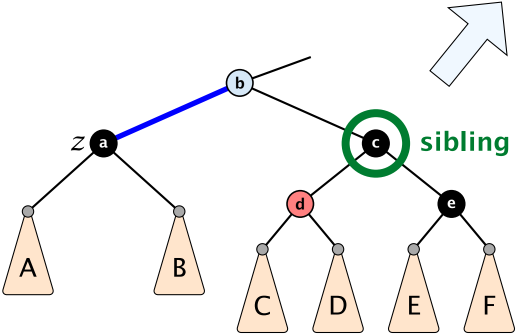


# Case 3: Sibling black with one black child to the right

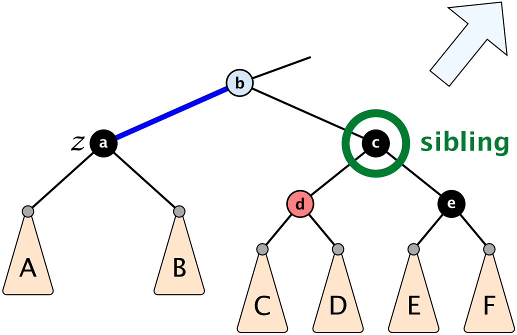
1. do a right-rotation at sibling
2. recolor $c$ and $d$
3. new sibling is black with red right child (Case 4)

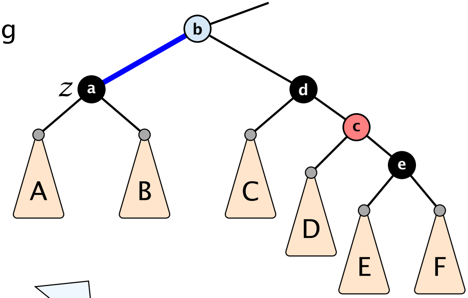


# Case 3: Sibling black with one black child to the right

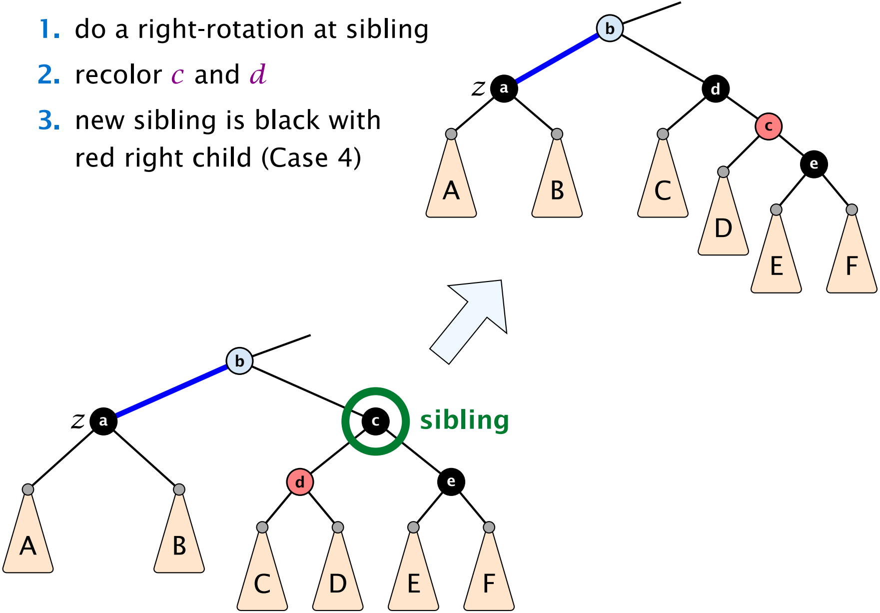
1. do a right-rotation at sibling
2. recolor $c$ and $d$
3. new sibling is black with red right child (Case 4)



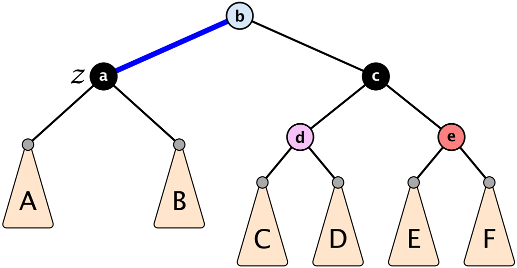



# Case 3: Sibling black with one black child to the right

1. do a right-rotation at sibling
2. recolor $c$ and $d$
3. new sibling is black with red right child (Case 4)

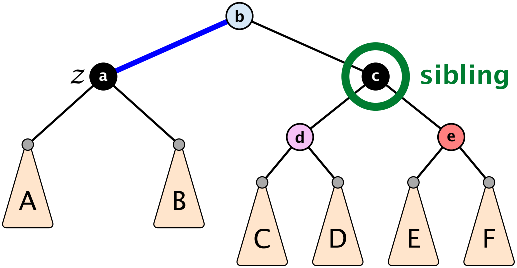


# Case 4: Sibling is black with red right child



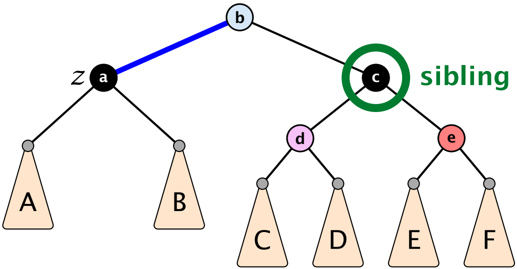

1. left-rotate around $b$
2. remove the fake black unit
3. recolor nodes $b$, $c$, and $e$
4. you have a valid
   red black tree

# Case 4: Sibling is black with red right child

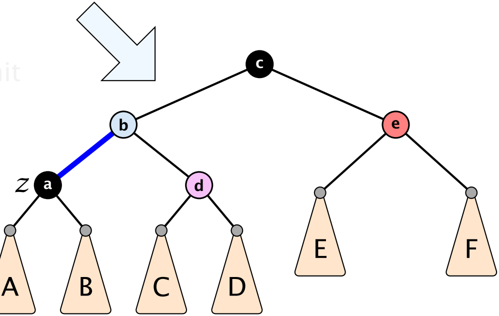



1. left-rotate around $b$
2. remove the fake black unit
3. recolor nodes $b$, $c$, and $e$
4. you have a valid red black tree

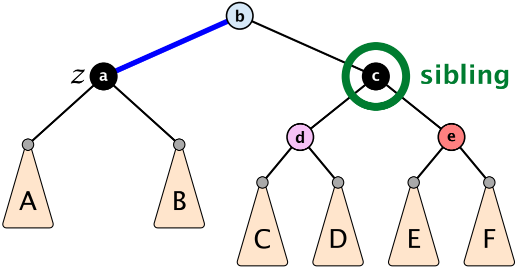
# Case 4: Sibling is black with red right child



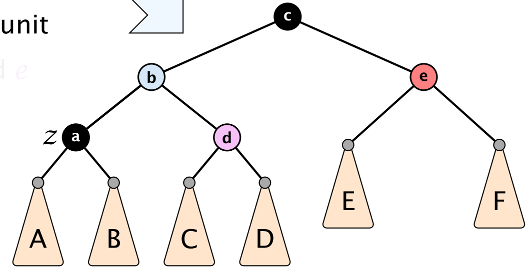

1. left-rotate around $b$
2. remove the fake black unit
3. recolor nodes $b$, $c$, and $e$
4. you have a valid red black tree



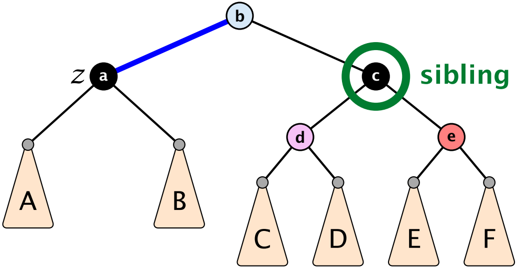
# Case 4: Sibling is black with red right child



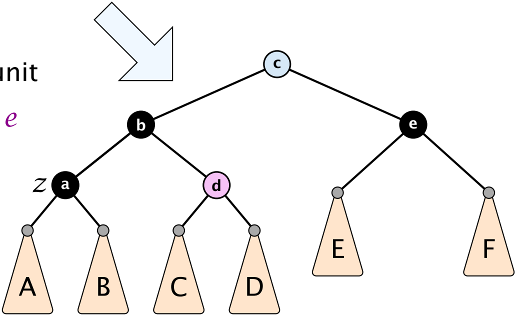

1. left-rotate around $b$
2. remove the fake black unit
3. recolor nodes $b$, $c$, and $e$
4. you have a valid red black tree



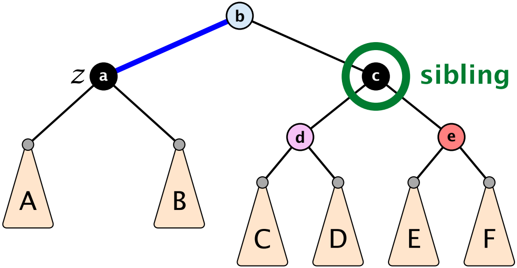
# Case 4: Sibling is black with red right child



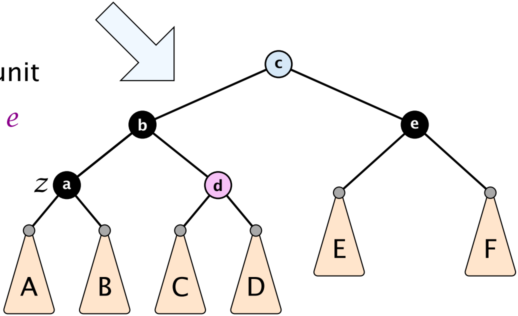

1. left-rotate around $b$
2. remove the fake black unit
3. recolor nodes $b$, $c$, and $e$
4. you have a valid red black tree



# Case 4: Sibling is black with red right child




1. left-rotate around $b$
2. remove the fake black unit
3. recolor nodes $b$, $c$, and $e$
4. you have a valid red black tree



**Running time:**

- only Case 2 can repeat; but only $h$ many steps, where $h$ is the height of the tree
- Case 1 → Case 2 (special) → red black tree
  Case 1 → Case 3 → Case 4 → red black tree
  Case 1 → Case 4 → red black tree
- Case 3 → Case 4 → red black tree
- Case 4 → red black tree

Performing Case 2 at most $\mathcal{O}(\log n)$ times and every other step at most once, we get a red black tree. Hence, $\mathcal{O}(\log n)$ re-colorings and at most $3$ rotations.

**Running time:**

- only Case 2 can repeat; but only $h$ many steps, where $h$ is the height of the tree
- Case 1 → Case 2 (special) → red black tree
  Case 1 → Case 3 → Case 4 → red black tree
  Case 1 → Case 4 → red black tree
- Case 3 → Case 4 → red black tree
- Case 4 → red black tree

Performing Case 2 at most $\mathcal{O}(\log n)$ times and every other step at most once, we get a red black tree. Hence, $\mathcal{O}(\log n)$ re-colorings and at most $3$ rotations.

**Running time:**

- only Case 2 can repeat; but only $h$ many steps, where $h$ is the height of the tree
- Case 1 → Case 2 (special) → red black tree
  Case 1 → Case 3 → Case 4 → red black tree
  Case 1 → Case 4 → red black tree
- Case 3 → Case 4 → red black tree
- Case 4 → red black tree

Performing Case 2 at most $\mathcal{O}(\log n)$ times and every other step at most once, we get a red black tree. Hence, $\mathcal{O}(\log n)$ re-colorings and at most $3$ rotations.

**Running time:**

- only Case 2 can repeat; but only $h$ many steps, where $h$ is the height of the tree
- Case 1 → Case 2 (special) → red black tree
  Case 1 → Case 3 → Case 4 → red black tree
  Case 1 → Case 4 → red black tree
- Case 3 → Case 4 → red black tree
- Case 4 → red black tree

Performing Case 2 at most $\mathcal{O}(\log n)$ times and every other step at most once, we get a red black tree. Hence, $\mathcal{O}(\log n)$ re-colorings and at most $3$ rotations.

**Running time:**

- only Case 2 can repeat; but only $h$ many steps, where $h$ is the height of the tree
- Case 1 → Case 2 (special) → red black tree
  Case 1 → Case 3 → Case 4 → red black tree
  Case 1 → Case 4 → red black tree
- Case 3 → Case 4 → red black tree
- Case 4 → red black tree

Performing Case 2 at most $\mathcal{O}(\log n)$ times and every other step at most once, we get a red black tree. Hence, $\mathcal{O}(\log n)$ re-colorings and at most $3$ rotations.