Data Structures

Gabriel Istrate

April 23, 2024

Today

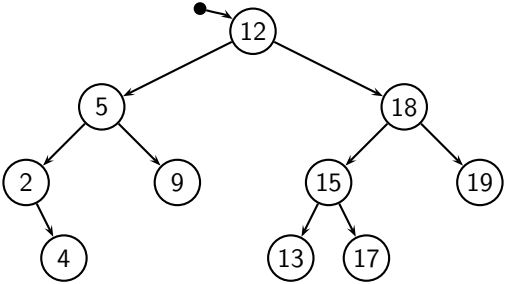
- Red-black trees

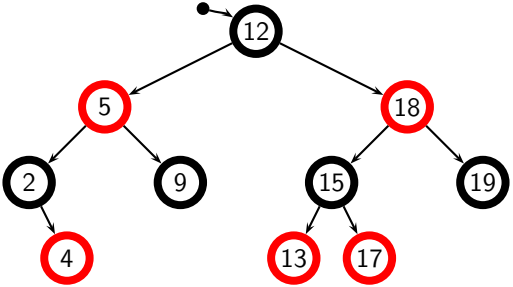
Where we are

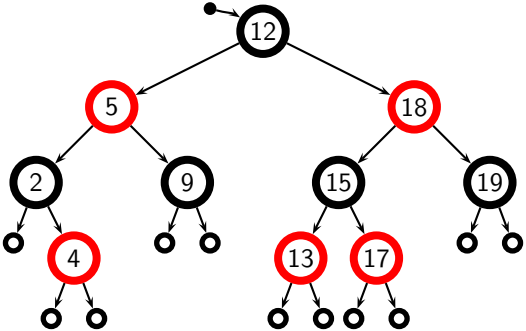
- Want: data structure to support INSERT, DELETE, SEARCH in $O(\log n)$ time.
- Binary search trees: insert, delete, search.
- But complexity bound not met unless trees balanced.
- Last time: AVL trees (meets $O(\log n)$ goal), splay trees (simpler, only meets it in an amortized sense).

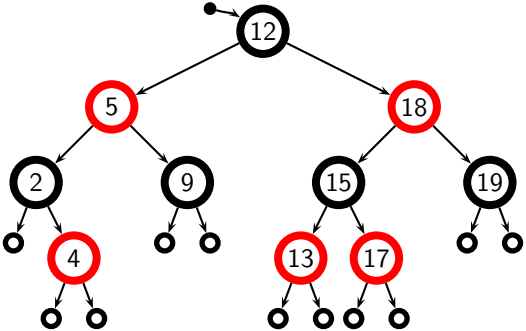
Today: red-black-trees.

Red-Black Trees

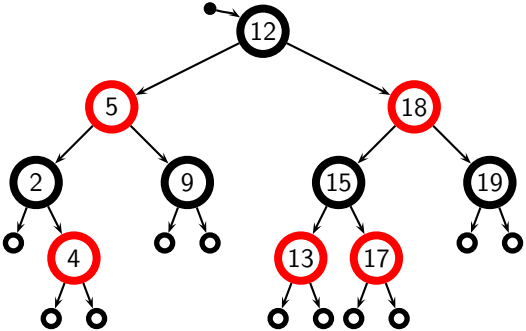



12
5
18
2
9
15
19
4
13
17

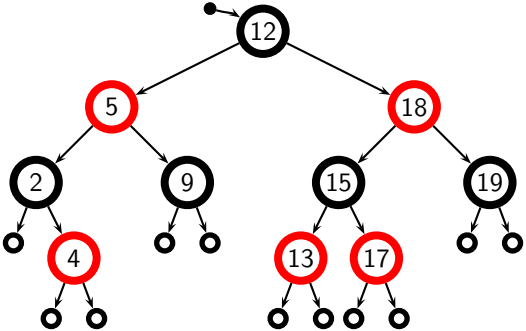

12
5
18
2
9
15
19
4
13
17



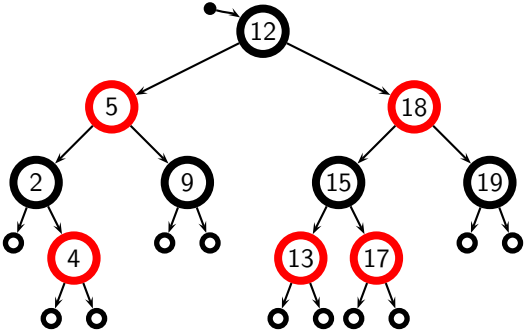
- *Red-black-tree property*



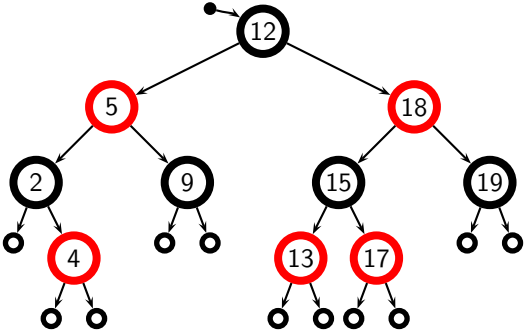
- *Red-black-tree property*
  1. every node is either **red** or **black**



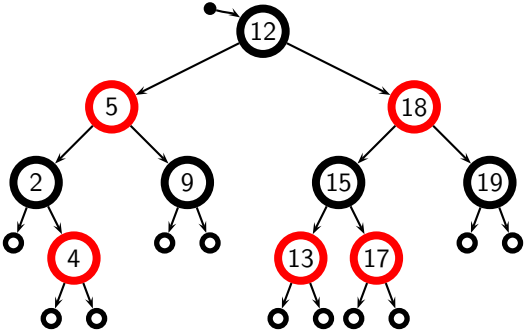
- *Red-black-tree property*
  1. every node is either **red** or **black**
  2. the root is **black**



- *Red-black-tree property*
  1. every node is either **red** or **black**
  2. the root is **black**
  3. every (NIL) leaf is **black**



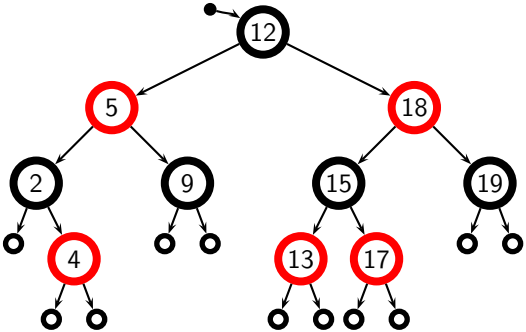
- *Red-black-tree property*
  1. every node is either **red** or **black**
  2. the root is **black**
  3. every (NIL) leaf is **black**
  4. if a node is **red**, then both its children are **black**



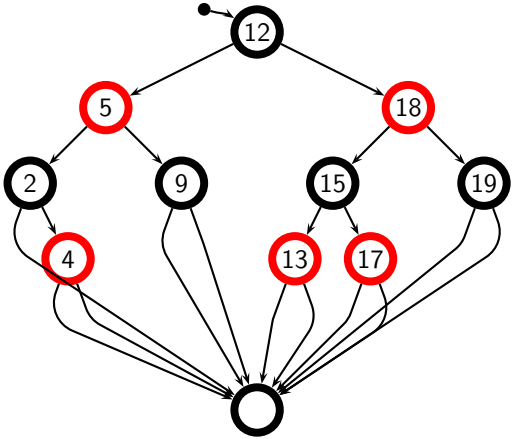
- *Red-black-tree property*
  1. every node is either **red** or **black**
  2. the root is **black**
  3. every (NIL) leaf is **black**
  4. if a node is **red**, then both its children are **black**
  5. for every node x, each path from x to its descendant leaves has the same number of **black** nodes $bh(x)$ (the *black-height* of x)

Red-Black Trees (2)

- *Implementation*

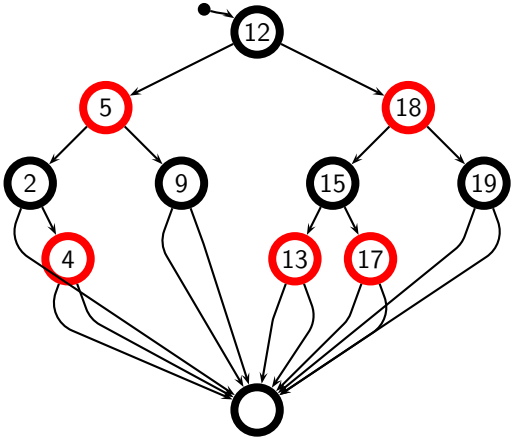


- *Implementation*



  - we use a common "sentinel" node to represent leaf nodes

- *Implementation*



  - we use a common "sentinel" node to represent leaf nodes
  - the sentinel is also the parent of the root node

Red-Black Trees (3)

- *Implementation*
  - T represents the tree, which consists of a set of *nodes*

Red-Black Trees (3)

- *Implementation*
  - T represents the tree, which consists of a set of *nodes*
  - $T.root$ is the root node of tree T

Red-Black Trees (3)

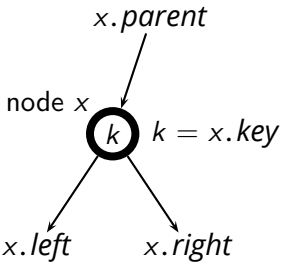
- *Implementation*
  - T represents the tree, which consists of a set of *nodes*
  - $T.root$ is the root node of tree T
  - $T.nil$ is the "sentinel" node of tree T

Red-Black Trees (3)

- *Implementation*
  - T represents the tree, which consists of a set of *nodes*
  - $T.root$ is the root node of tree T
  - $T.nil$ is the "sentinel" node of tree T

Nodes

- $x.parent$ is the parent of node x
- $x.key$ is the key stored in node x
- $x.left$ is the left child of node x
- $x.right$ is the right child of node x

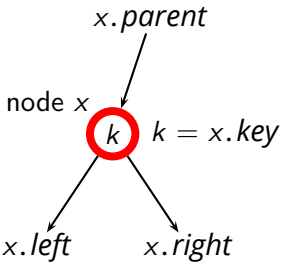


Red-Black Trees (3)

- *Implementation*
  - T represents the tree, which consists of a set of *nodes*
  - $T.root$ is the root node of tree T
  - $T.nil$ is the "sentinel" node of tree T

Nodes

- $x.parent$ is the parent of node x
- $x.key$ is the key stored in node x
- $x.left$ is the left child of node x
- $x.right$ is the right child of node x
- $x.color \in \{\text{RED}, \text{BLACK}\}$ is the color of node x



Height of a Red-Black Tree

Height of a Red-Black Tree

Lemma: the height $h(x)$ of a red-black tree with $n = \mathit{size}(x)$ internal nodes is at most $2\log(n+1)$.

Height of a Red-Black Tree

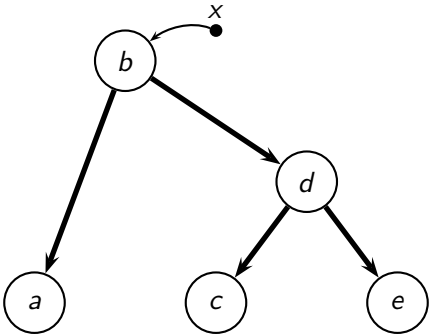
Lemma: the height $h(x)$ of a red-black tree with $n = \mathit{size}(x)$ internal nodes is at most $2\log(n+1)$.

- A red-black tree works as a binary search tree for search, etc.
- So, the complexity of those operations is $T(n) = O(h)$, that is

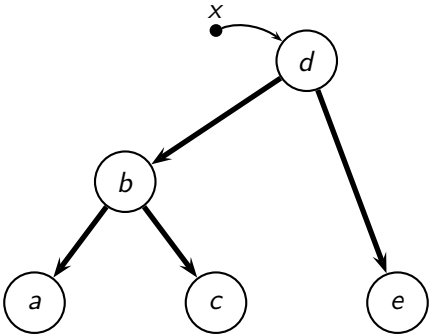
$$\boxed{T(n) = O(\log n)}$$

  - which is also the *worst-case* complexity

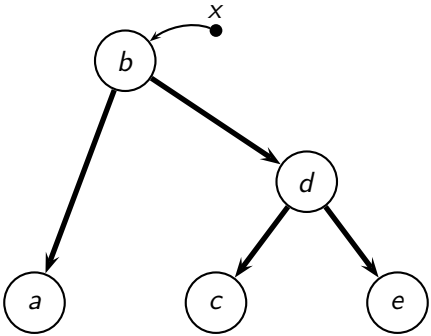
Rotation



Rotation

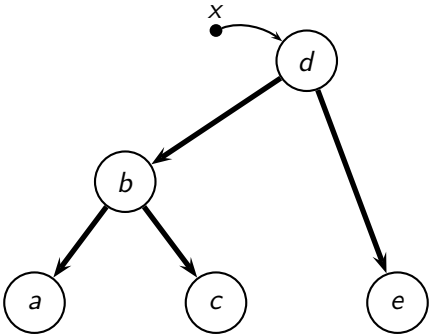


Rotation



- x = **Right-Rotate**(x)

Rotation



- x = **Right-Rotate**(x)
- x = **Left-Rotate**(x)

Red-Black Insertion

- **RB-Insert**(T, z) works as in a binary search tree

Red-Black Insertion

- **RB-Insert**(T, z) works as in a binary search tree
- Except that it must preserve the *red-black-tree property*

Red-Black Insertion

- **RB-Insert**(T, z) works as in a binary search tree
- Except that it must preserve the *red-black-tree property*
  1. every node is either **red** or **black**
  2. the root is **black**
  3. every (NIL) leaf is **black**
  4. if a node is **red**, then both its children are **black**
  5. for every node x, each path from x to its descendant leaves has the same number of **black** nodes $bh(x)$ (the *black-height* of x)

Red-Black Insertion

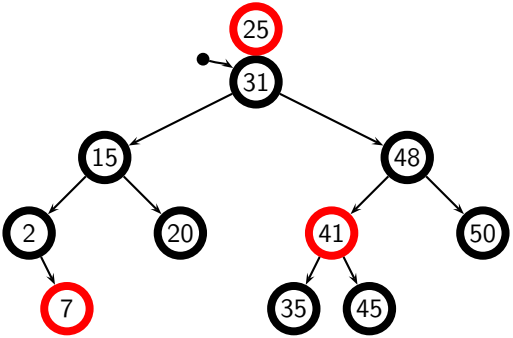
- **RB-Insert**(T, z) works as in a binary search tree
- Except that it must preserve the *red-black-tree property*
  1. every node is either **red** or **black**
  2. the root is **black**
  3. every (NIL) leaf is **black**
  4. if a node is **red**, then both its children are **black**
  5. for every node x, each path from x to its descendant leaves has the same number of **black** nodes $bh(x)$ (the *black-height* of x)
- *General strategy*

- **RB-Insert**(T, z) works as in a binary search tree
- Except that it must preserve the *red-black-tree property*
  1. every node is either **red** or **black**
  2. the root is **black**
  3. every (NIL) leaf is **black**
  4. if a node is **red**, then both its children are **black**
  5. for every node x, each path from x to its descendant leaves has the same number of **black** nodes $bh(x)$ (the *black-height* of x)
- *General strategy*
  1. insert z as in a binary search tree
  2. color z **red** so as to preserve property 5
  3. *fix the tree* to correct possible violations of property 4

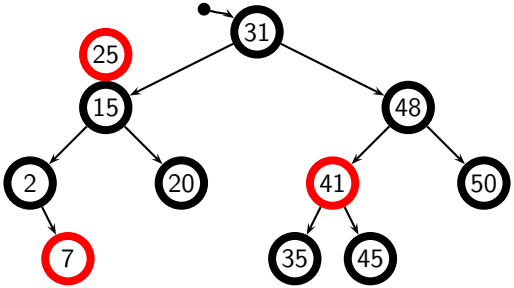
RB-Insert

```
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.parent = y
if y == T.nil
    T.root = z
else if z.key < y.key
        y.left = z
    else y.right = z
z.left = z.right = T.nil
z.color = RED
RB-Insert-Fixup(T, z)
```

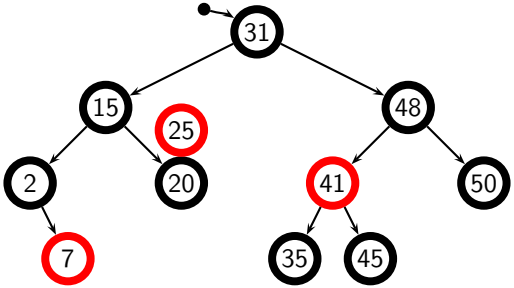
Red-Black Insertion (2)



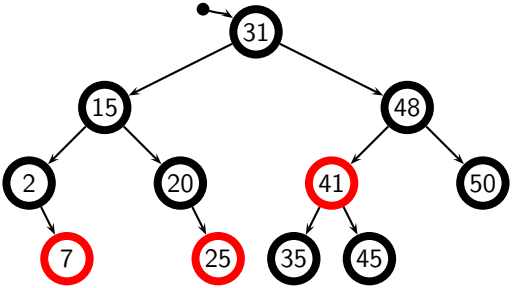
Red-Black Insertion (2)



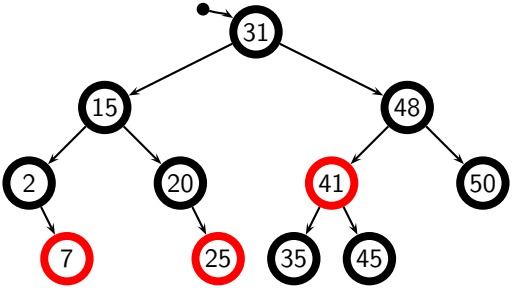
Red-Black Insertion (2)



Red-Black Insertion (2)



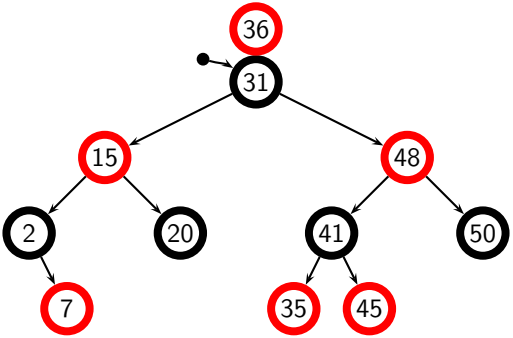
Red-Black Insertion (2)



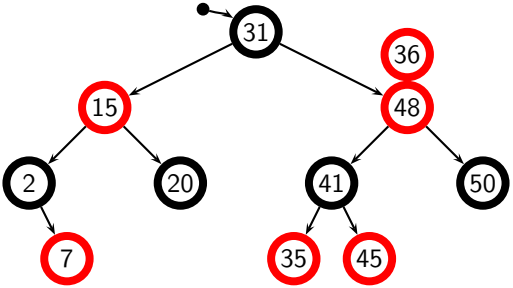
- z's father is **black**, so no fixup needed

Red-Black Insertion (3)

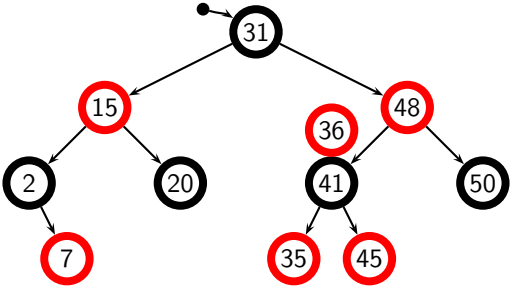
Red-Black Insertion (3)



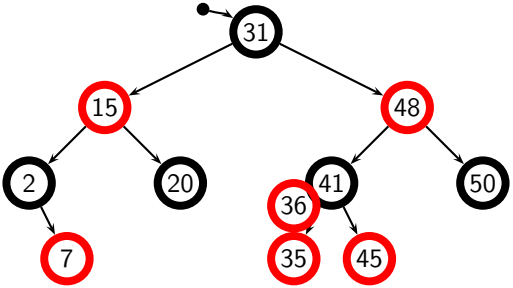
Red-Black Insertion (3)



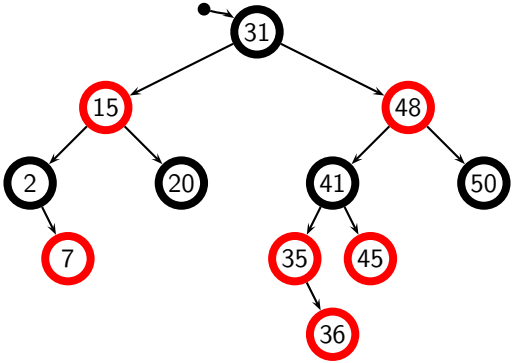
Red-Black Insertion (3)



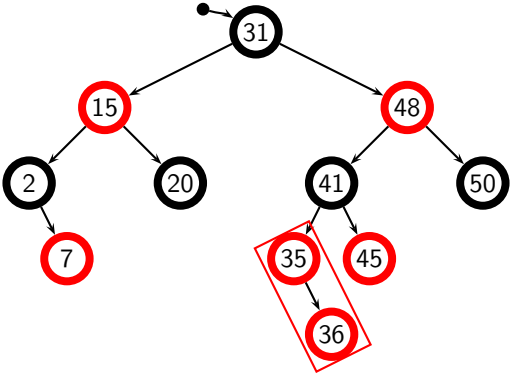
Red-Black Insertion (3)



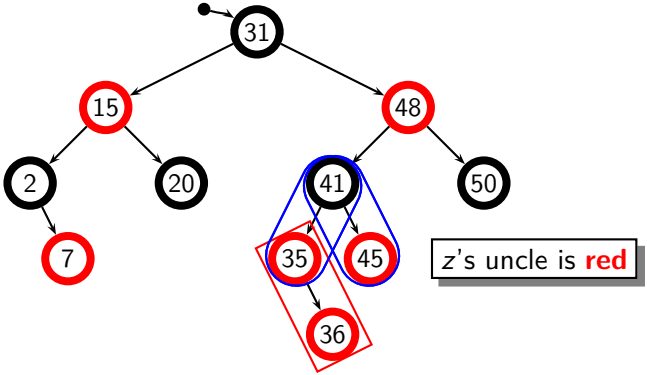
Red-Black Insertion (3)



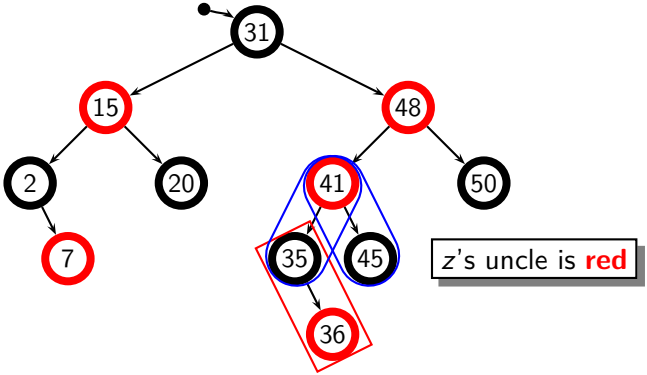
Red-Black Insertion (3)



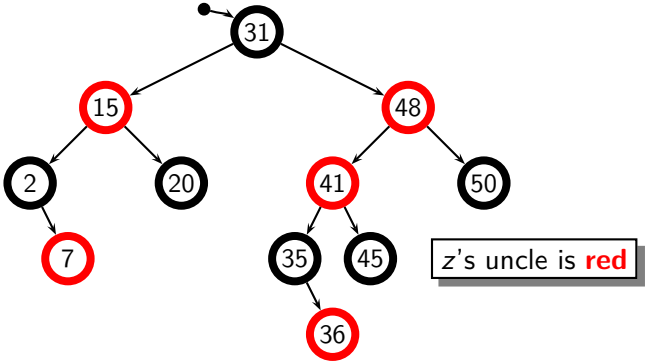
Red-Black Insertion (3)



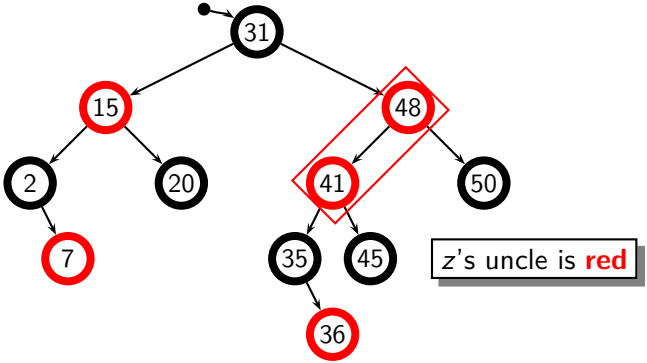
Red-Black Insertion (3)



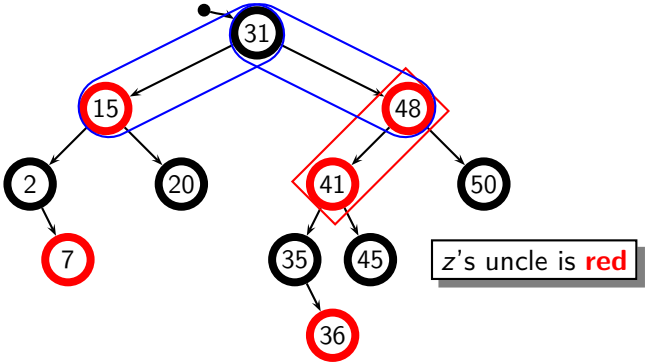
Red-Black Insertion (3)



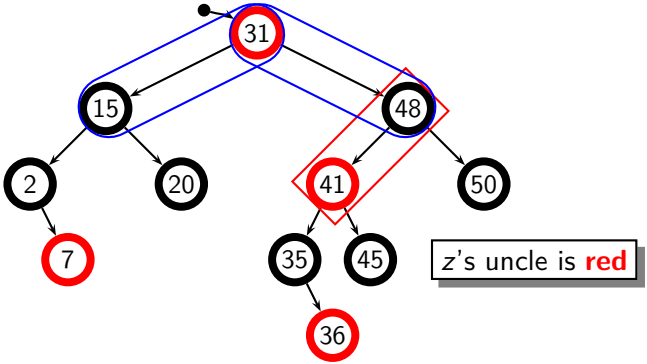
Red-Black Insertion (3)



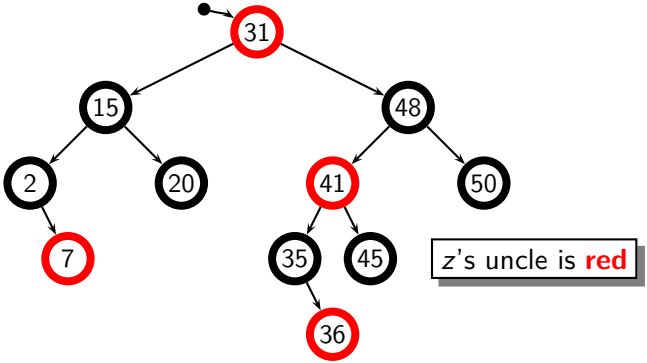
Red-Black Insertion (3)



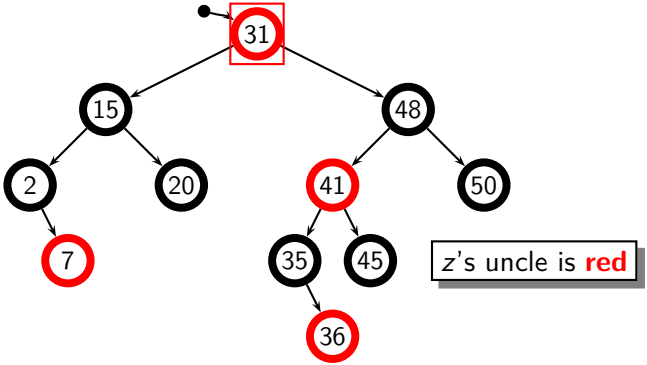
Red-Black Insertion (3)



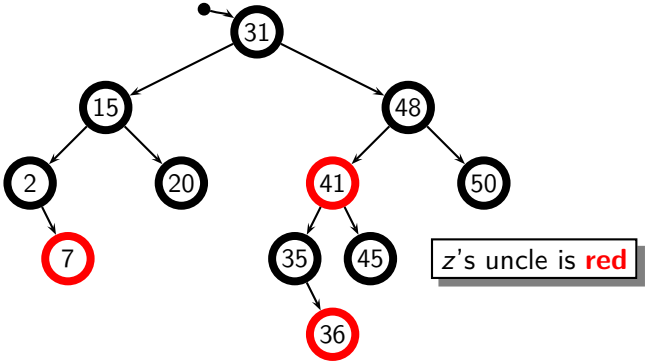
Red-Black Insertion (3)



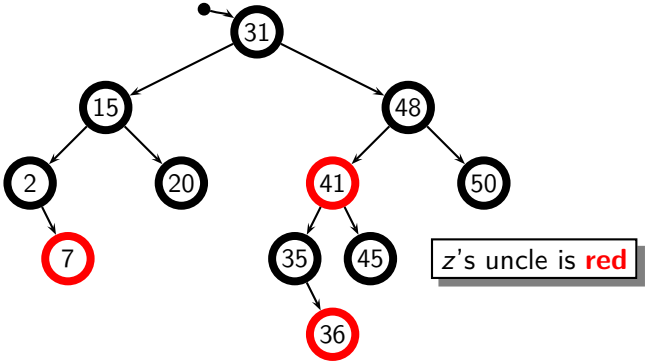
Red-Black Insertion (3)



Red-Black Insertion (3)

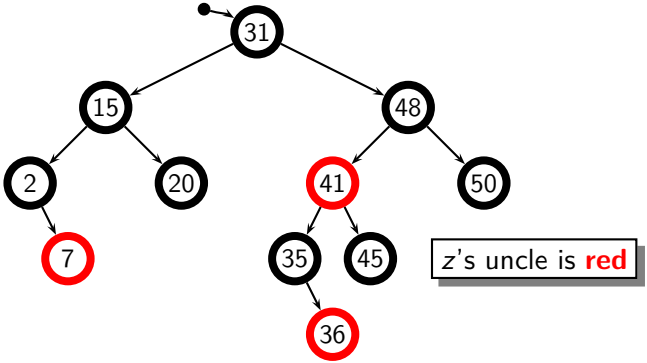


Red-Black Insertion (3)



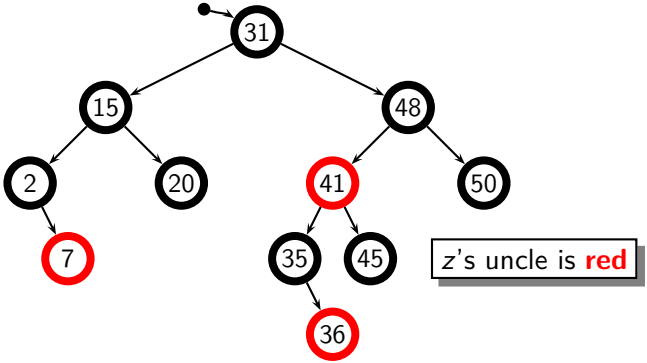
- A **black** node can become **red** and transfer its **black** color to its two children

Red-Black Insertion (3)



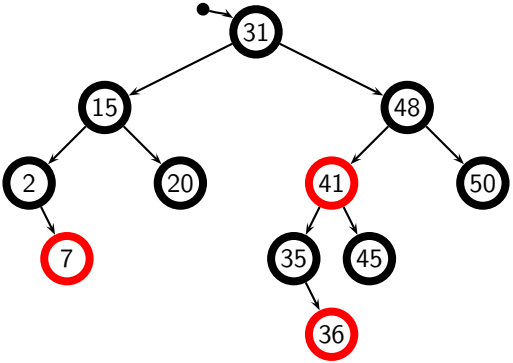
- A **black** node can become **red** and transfer its **black** color to its two children
- This may cause other **red**–**red** conflicts, so we iterate...

Red-Black Insertion (3)

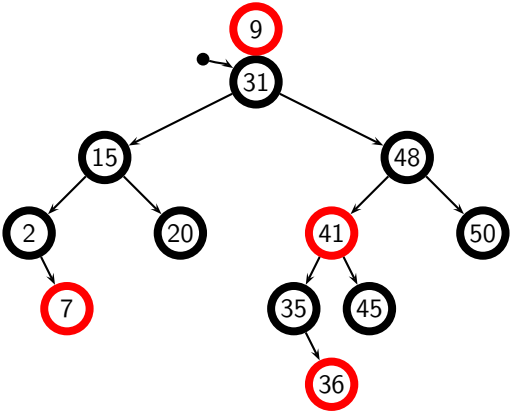


- A **black** node can become **red** and transfer its **black** color to its two children
- This may cause other **red**–**red** conflicts, so we iterate...
- The root can change to **black** without causing conflicts

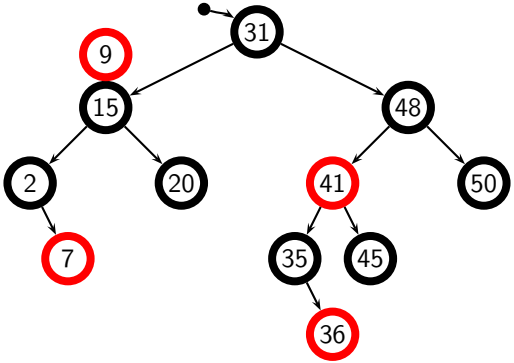
Red-Black Insertion (4)



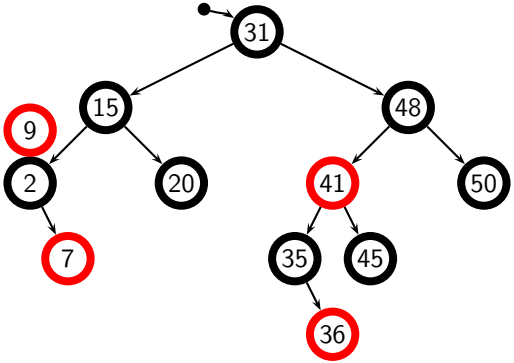
Red-Black Insertion (4)



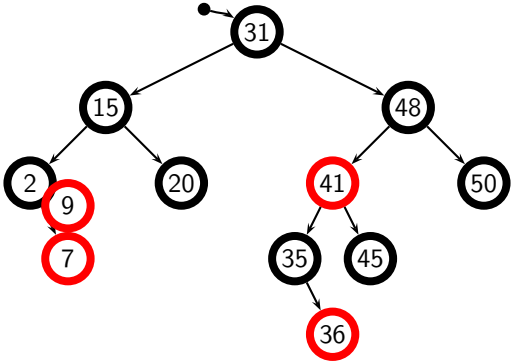
Red-Black Insertion (4)



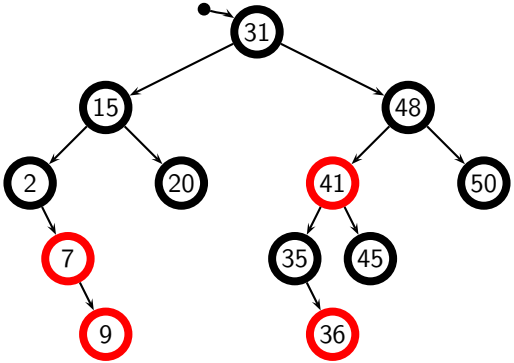
Red-Black Insertion (4)



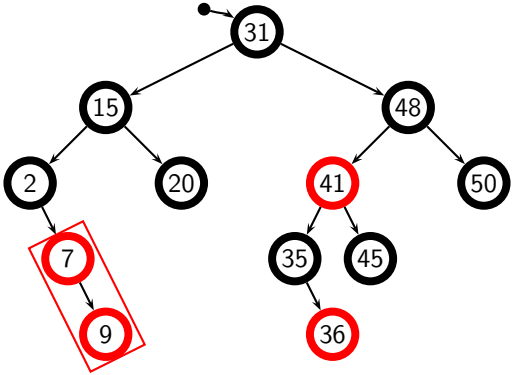
Red-Black Insertion (4)



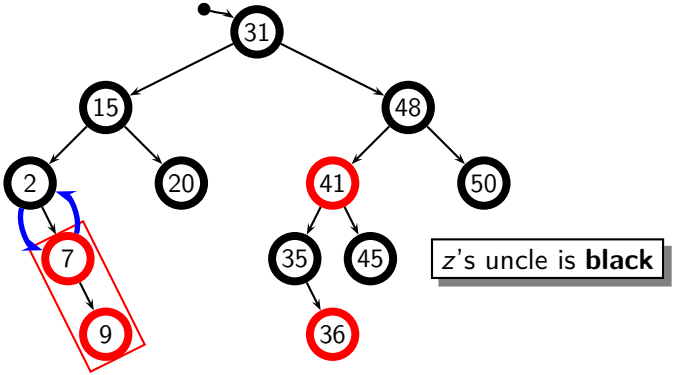
Red-Black Insertion (4)



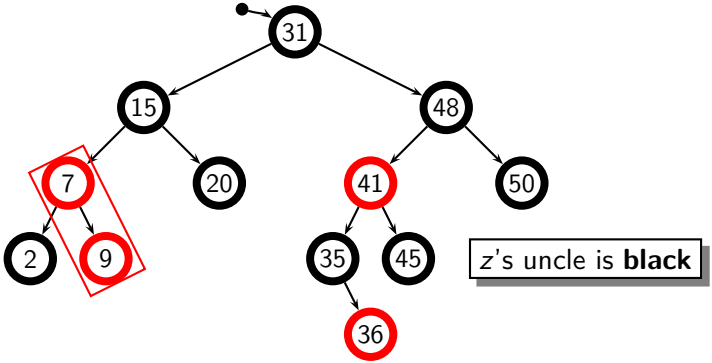
Red-Black Insertion (4)



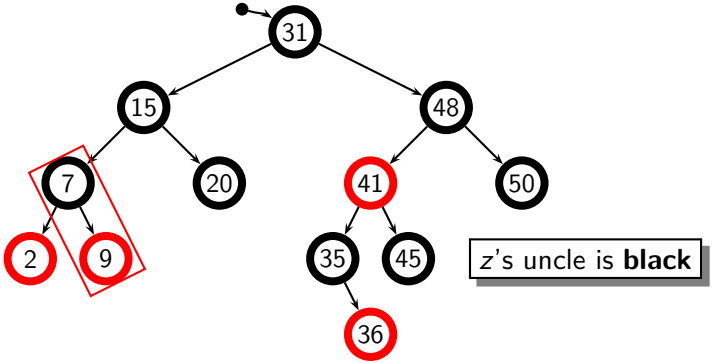
Red-Black Insertion (4)



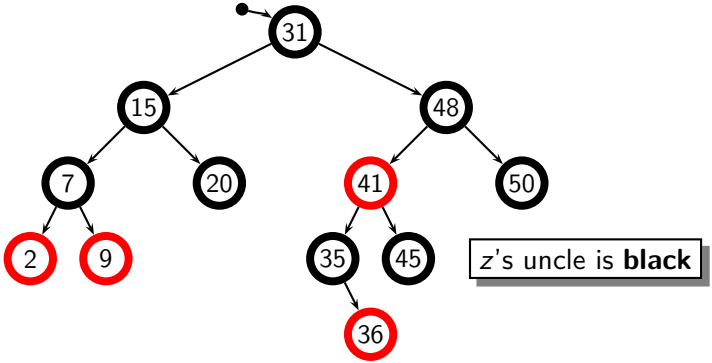
Red-Black Insertion (4)



Red-Black Insertion (4)

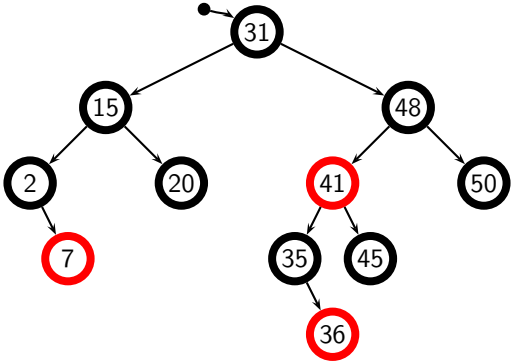


Red-Black Insertion (4)

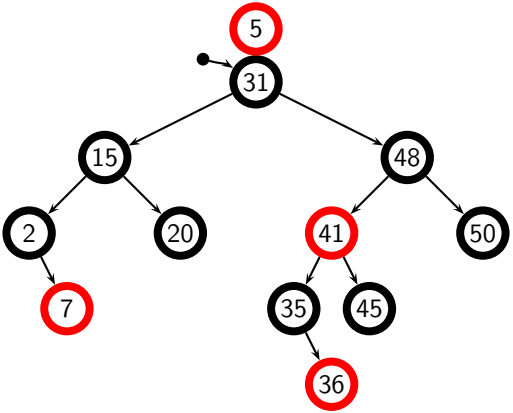


- An *in-line* ***red–red*** conflicts can be resolved with a rotation plus a color switch

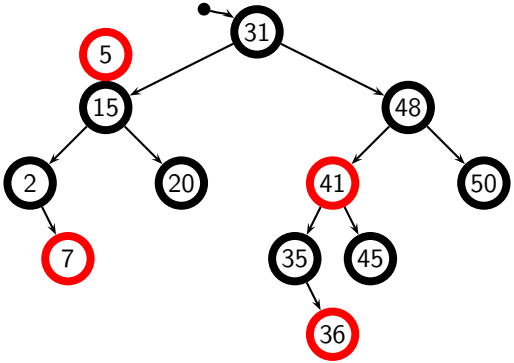
Red-Black Insertion (5)



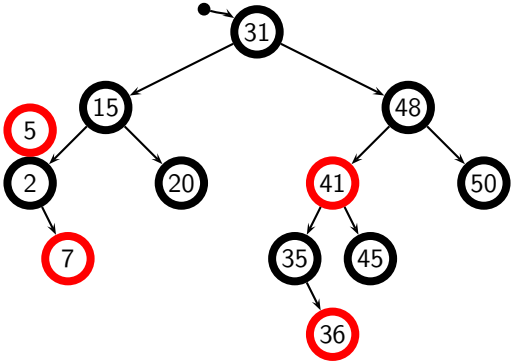
Red-Black Insertion (5)



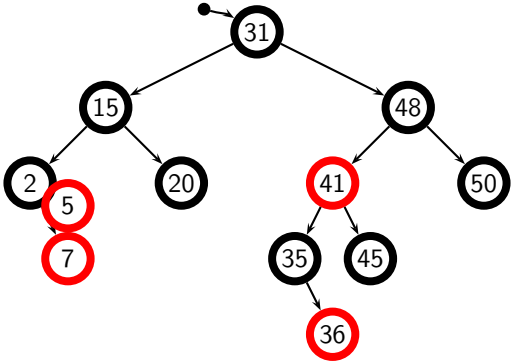
Red-Black Insertion (5)



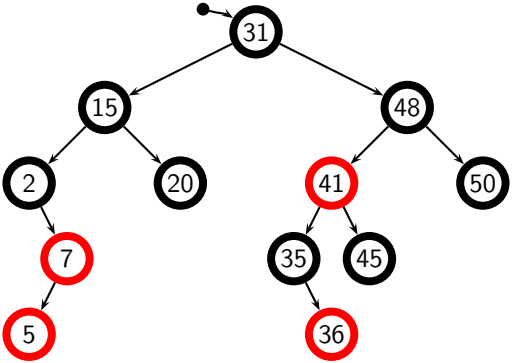
Red-Black Insertion (5)



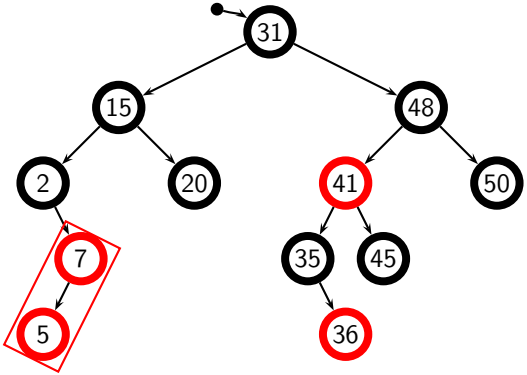
Red-Black Insertion (5)



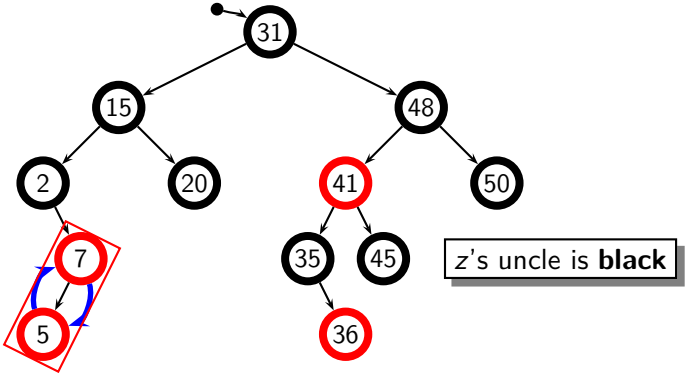
Red-Black Insertion (5)



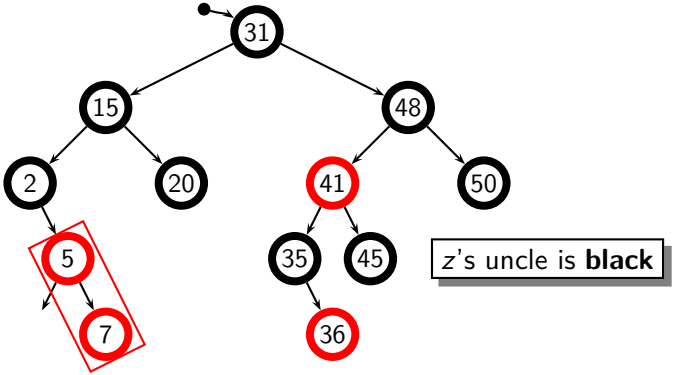
Red-Black Insertion (5)



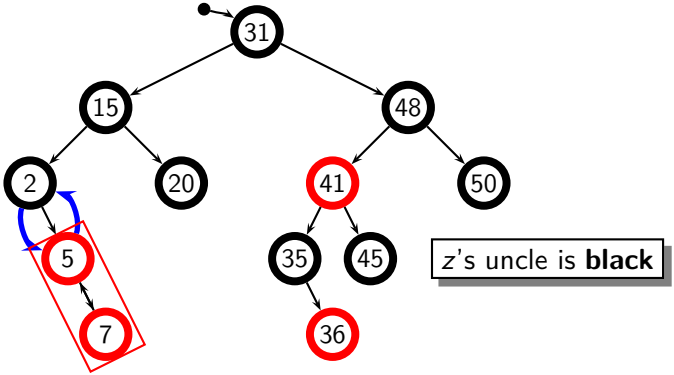
Red-Black Insertion (5)



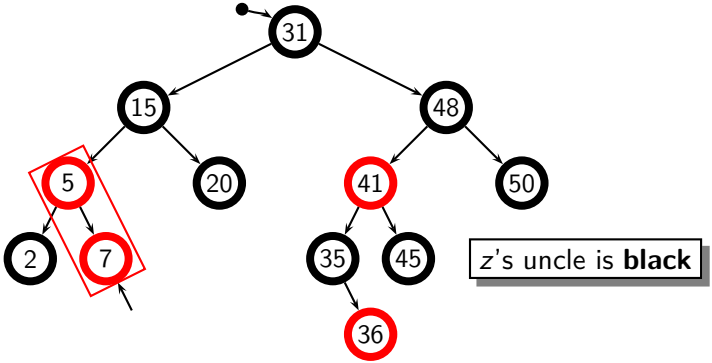
Red-Black Insertion (5)



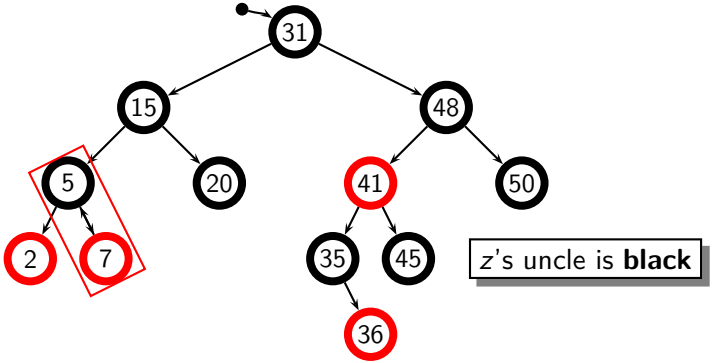
Red-Black Insertion (5)



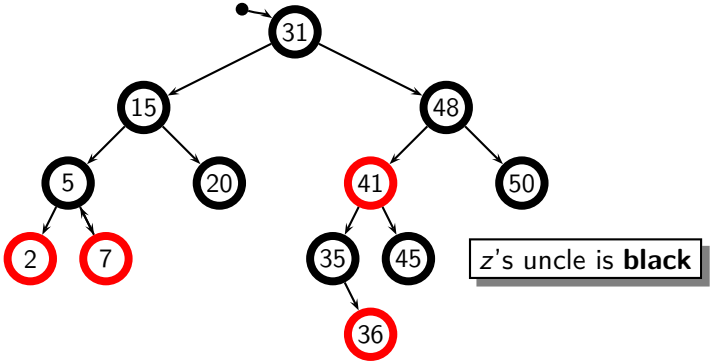
Red-Black Insertion (5)



Red-Black Insertion (5)



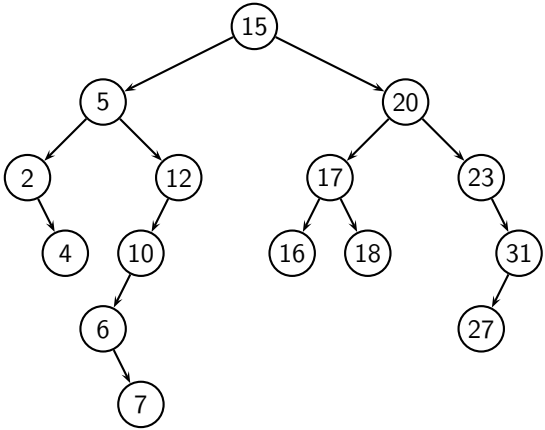
Red-Black Insertion (5)



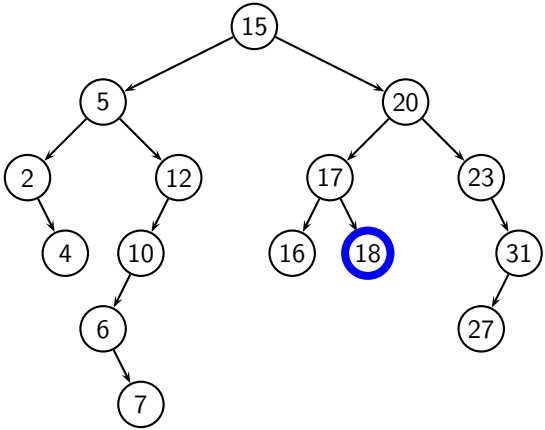
- A *zig-zag* **red–red** conflicts can be resolved with a rotation to turn it into an *in-line* conflict, and then a rotation plus a color switch

Recap on Deletion in Binary Trees

Recap on Deletion in Binary Trees

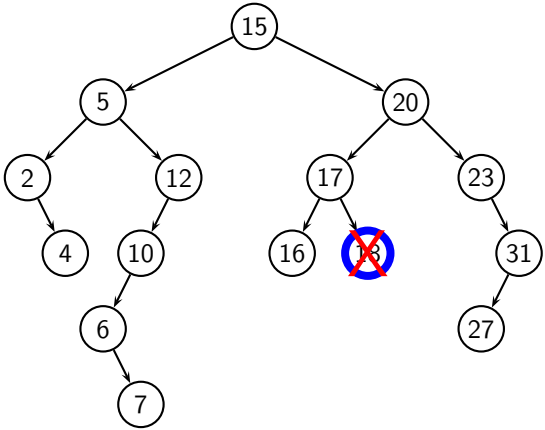


Recap on Deletion in Binary Trees



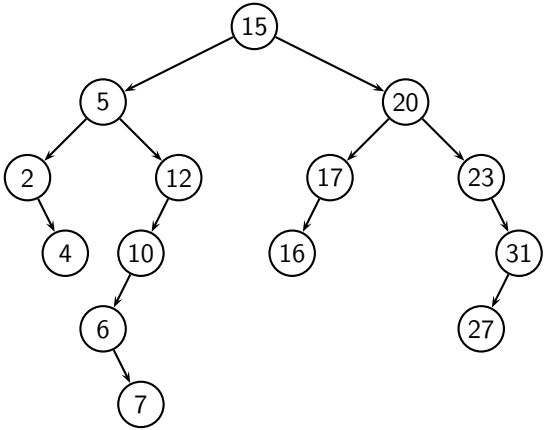
1. z has no children

Recap on Deletion in Binary Trees



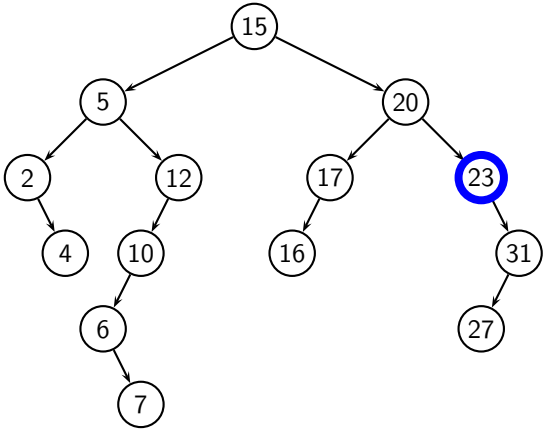
1. z has no children
   - simply remove z

Recap on Deletion in Binary Trees



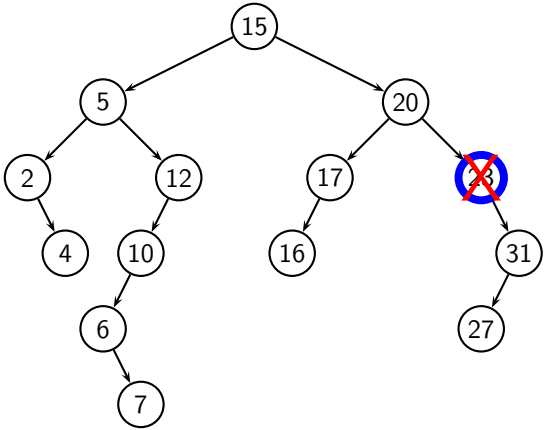
1. z has no children
   - simply remove z

Recap on Deletion in Binary Trees



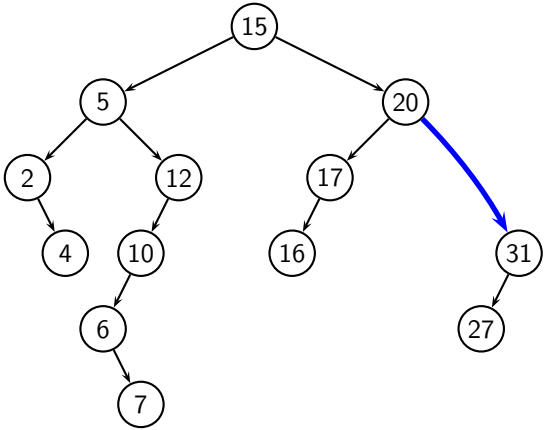
1. z has no children
   - simply remove z
2. z has one child

Recap on Deletion in Binary Trees



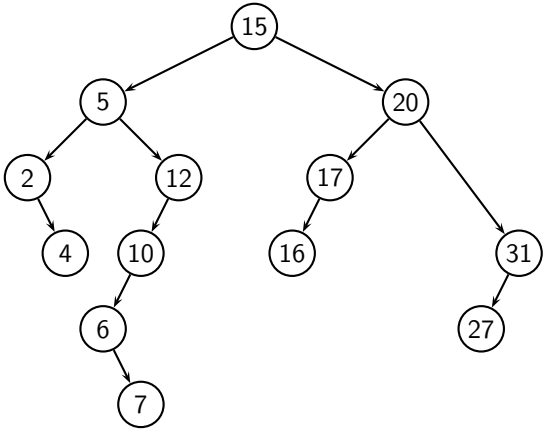
1. z has no children
   - simply remove z
2. z has one child
   - remove z

Recap on Deletion in Binary Trees



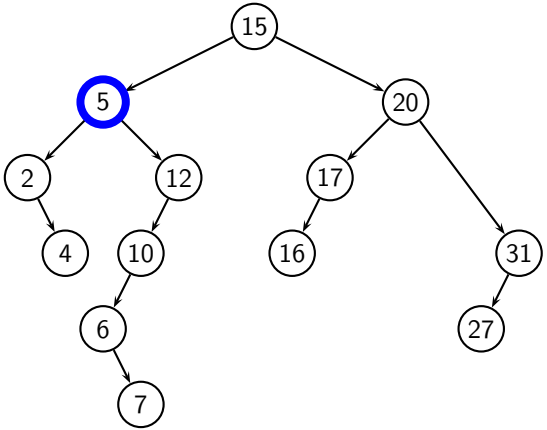
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect $z.parent$ to $z.right$

Recap on Deletion in Binary Trees



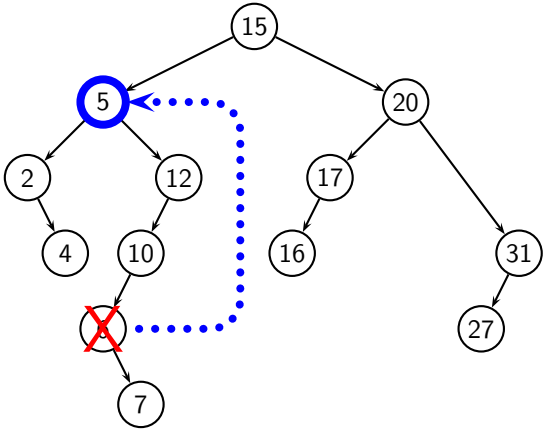
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect $z.parent$ to $z.right$

Recap on Deletion in Binary Trees



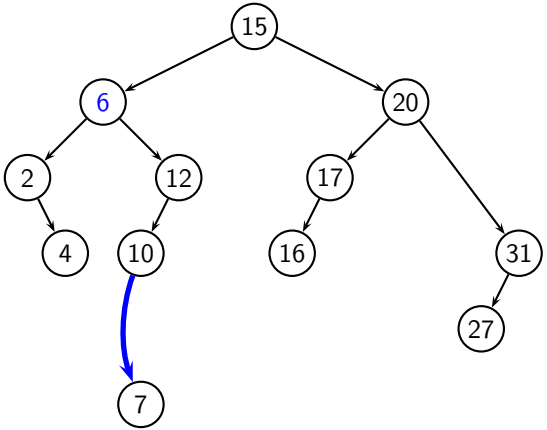
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect $z.parent$ to $z.right$
3. z has two children

Recap on Deletion in Binary Trees



1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect $z.parent$ to $z.right$
3. z has two children
   - replace z with $y = \textbf{Tree-Successor}(z)$
   - remove y (1 child!)

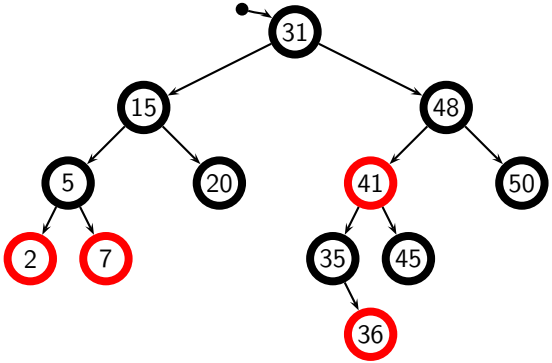
Recap on Deletion in Binary Trees



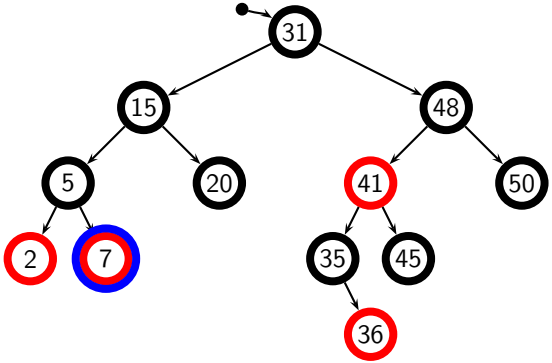
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect $z.parent$ to $z.right$
3. z has two children
   - replace z with $y = \textbf{Tree-Successor}(z)$
   - remove y (1 child!)
   - connect $y.parent$ to $y.right$

Red-Black Deletion

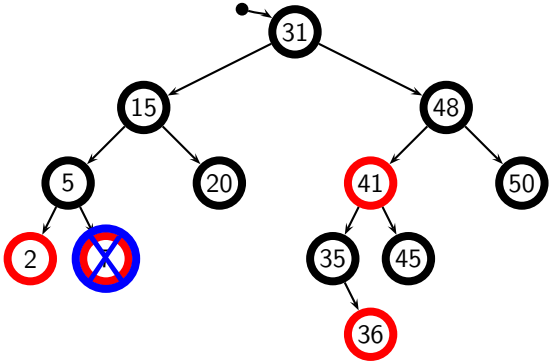
Red-Black Deletion



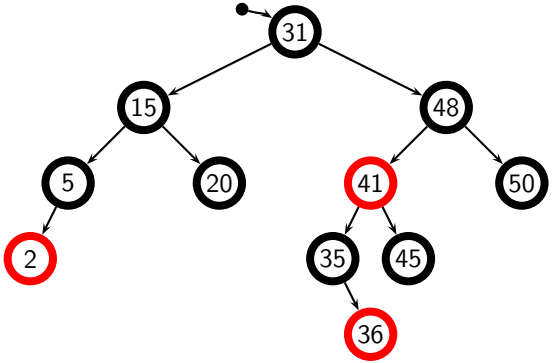
Red-Black Deletion

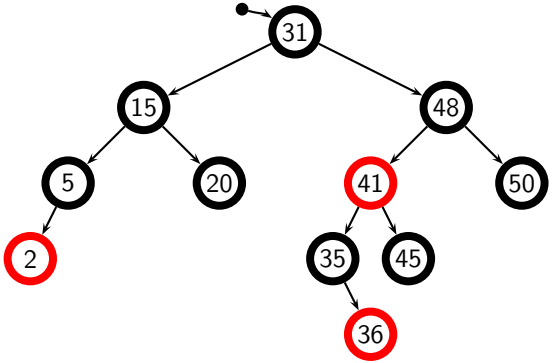


Red-Black Deletion

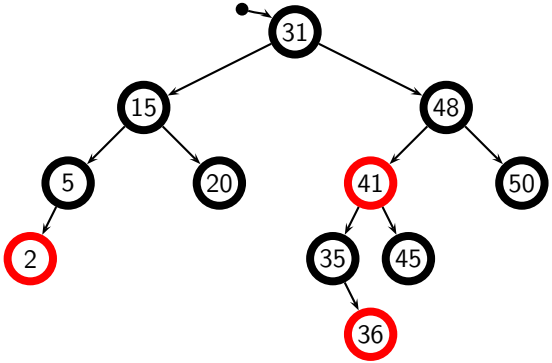


Red-Black Deletion

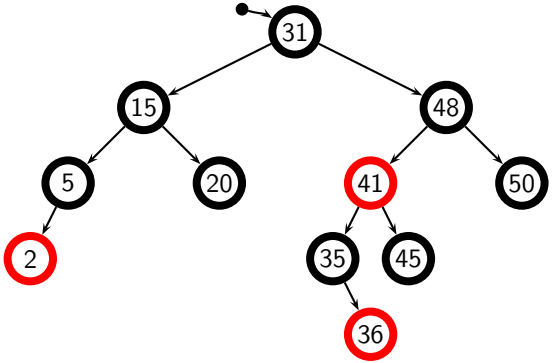




- A deleting a **red** *leaf* does not require any adjustment

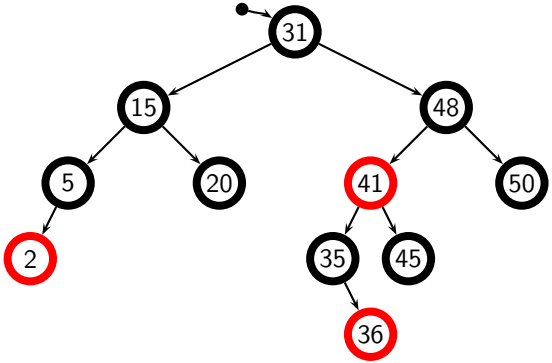


- A deleting a **red** *leaf* does not require any adjustment
  - the deletion does not affect the black height (property 5)

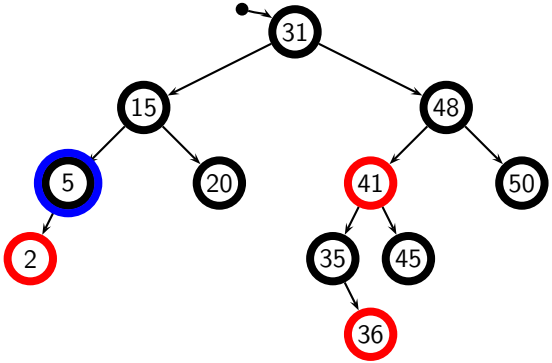


- A deleting a **red** *leaf* does not require any adjustment
  - the deletion does not affect the black height (property 5)
  - no two red nodes become adjacent (property 4)

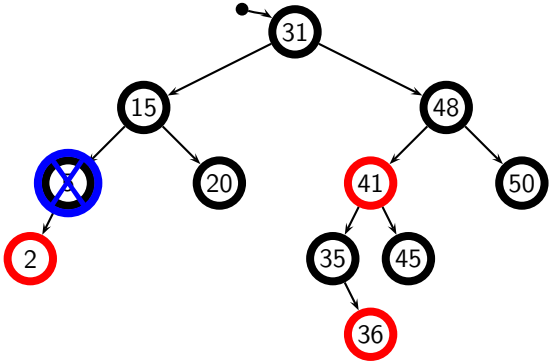
Red-Black Deletion (2)



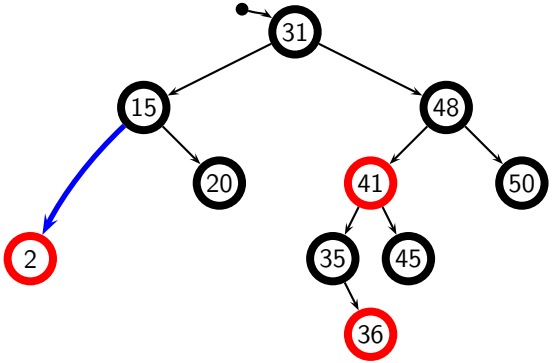
Red-Black Deletion (2)



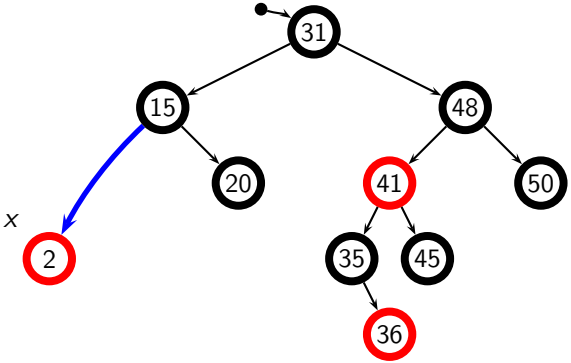
Red-Black Deletion (2)



Red-Black Deletion (2)

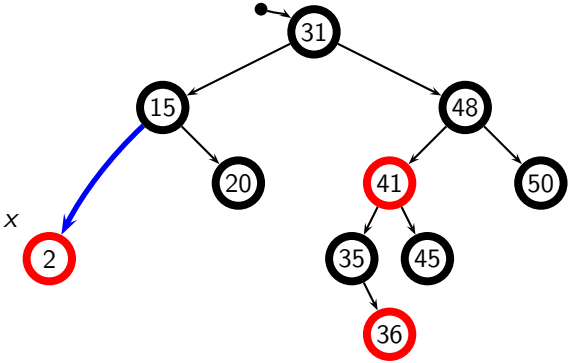


Red-Black Deletion (2)



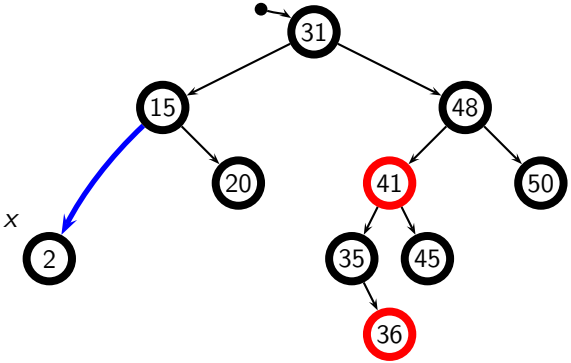
- Deleting a ***black*** *node* changes the balance of black-height in a subtree x

Red-Black Deletion (2)



- Deleting a ***black*** *node* changes the balance of black-height in a subtree x
  - **RB-Delete-Fixup**(T, x) fixes the tree after a deletion

Red-Black Deletion (2)



- Deleting a ***black*** *node* changes the balance of black-height in a subtree x
  - **RB-Delete-Fixup**(T, x) fixes the tree after a deletion
  - in this simple case: $x.color = \text{BLACK}$

Fixup Conditions

Fixup Conditions

- y is the spliced node ($y = z$ if z has zero or one child)
  - if y is **red**, then no fixup is necessary
  - so, here we assume that y is **black**

Fixup Conditions

- y is the spliced node ($y = z$ if z has zero or one child)
  - if y is **red**, then no fixup is necessary
  - so, here we assume that y is **black**
- x is either y's only child or $T.nil$
  - y was spliced out, so y can not have two children
  - $x = T.nil$ iff y has no (key-bearing) children

Fixup Conditions

- y is the spliced node ($y = z$ if z has zero or one child)
  - if y is **red**, then no fixup is necessary
  - so, here we assume that y is **black**
- x is either y's only child or $T.nil$
  - y was spliced out, so y can not have two children
  - $x = T.nil$ iff y has no (key-bearing) children
- ***Problem 1:*** $y = T.root$ and x is **red**
  - violates red-black property **??** (*root* must be **black**)

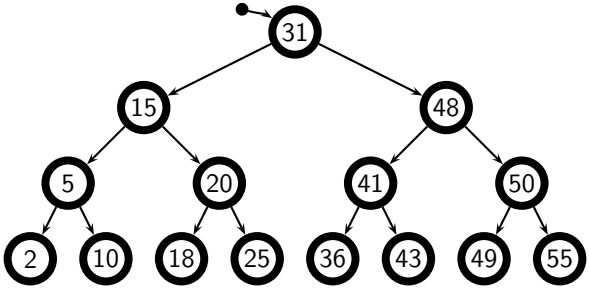
Fixup Conditions

- y is the spliced node ($y = z$ if z has zero or one child)
  - if y is **red**, then no fixup is necessary
  - so, here we assume that y is **black**
- x is either y's only child or $T.nil$
  - y was spliced out, so y can not have two children
  - $x = T.nil$ iff y has no (key-bearing) children
- ***Problem 1:*** $y = T.root$ and x is **red**
  - violates red-black property **??** (*root* must be **black**)
- ***Problem 2:*** both x and $y.parent$ are **red**
  - violates red-black property 4 (no adjacent red nodes)

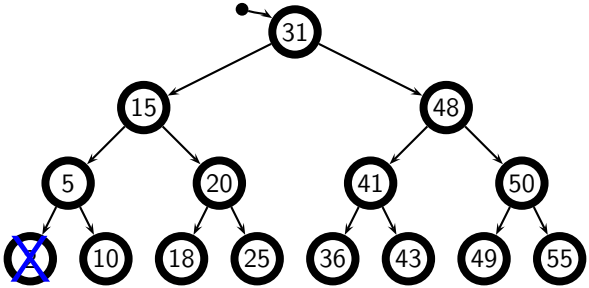
Fixup Conditions

- y is the spliced node ($y = z$ if z has zero or one child)
  - if y is **red**, then no fixup is necessary
  - so, here we assume that y is **black**
- x is either y's only child or $T.nil$
  - y was spliced out, so y can not have two children
  - $x = T.nil$ iff y has no (key-bearing) children
- ***Problem 1:*** $y = T.root$ and x is **red**
  - violates red-black property **??** (*root* must be **black**)
- ***Problem 2:*** both x and $y.parent$ are **red**
  - violates red-black property 4 (no adjacent red nodes)
- ***Problem 3:*** we are removing y, which is black
  - violates red-black property 5 (same *black height* for all paths)

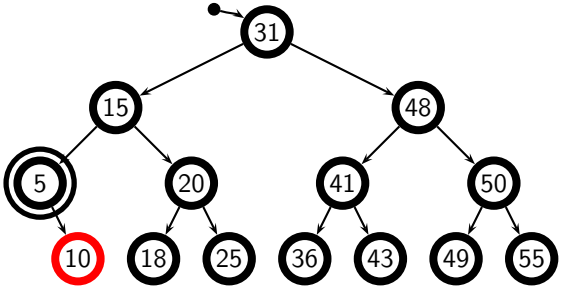
Red-Black Deletion (3)



Red-Black Deletion (3)

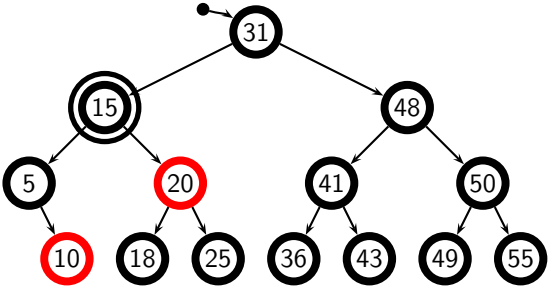


Red-Black Deletion (3)



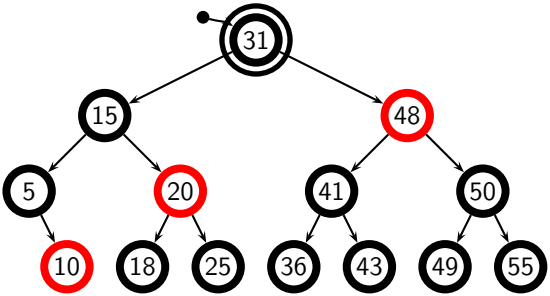
- x carries an *additional* ***black*** *weight*
  - the fixup algorithm pushes it up towards to root

Red-Black Deletion (3)



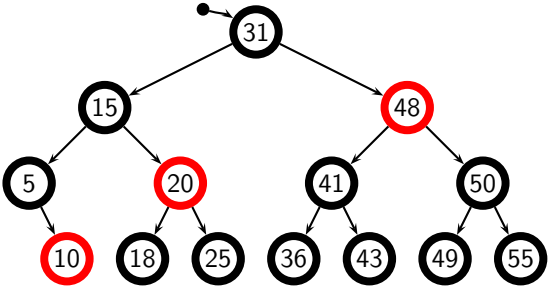
- x carries an *additional* ***black*** *weight*
  - the fixup algorithm pushes it up towards to root

Red-Black Deletion (3)



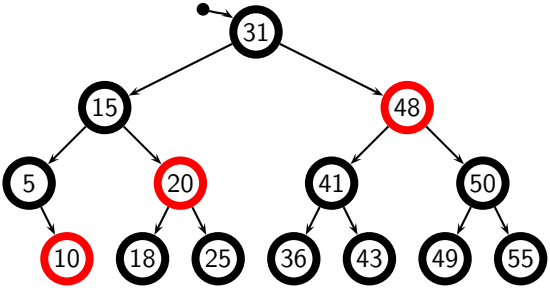
- x carries an *additional* ***black*** *weight*
  - the fixup algorithm pushes it up towards to root

Red-Black Deletion (3)



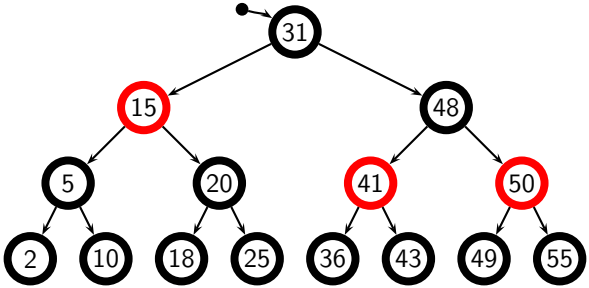
- x carries an *additional* ***black*** *weight*
  - the fixup algorithm pushes it up towards to root

Red-Black Deletion (3)

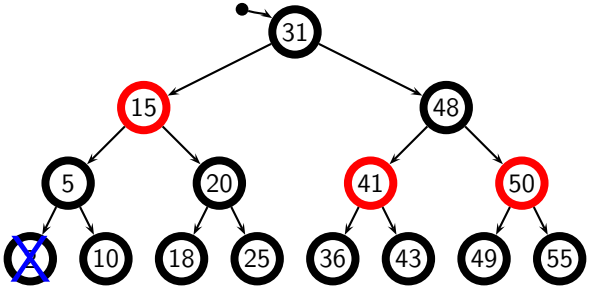


- x carries an *additional* ***black*** *weight*
  - the fixup algorithm pushes it up towards to root
- The *additional* ***black*** *weight* can be discarded if it reaches the *root*, otherwise...

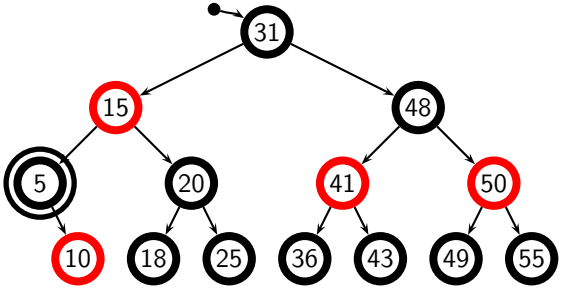
Red-Black Deletion (4)



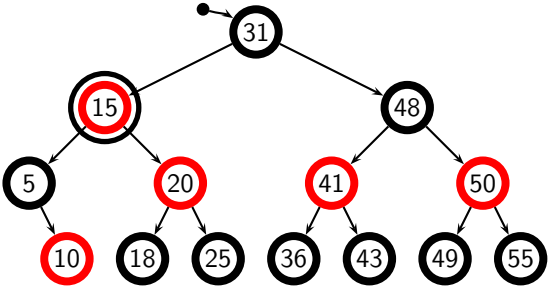
Red-Black Deletion (4)



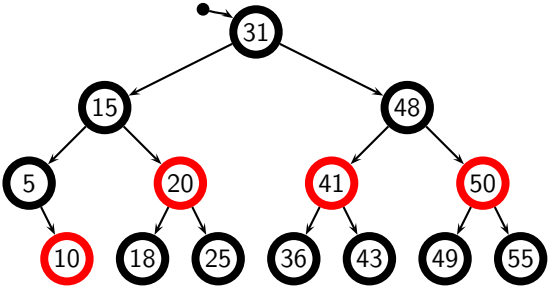
Red-Black Deletion (4)



Red-Black Deletion (4)

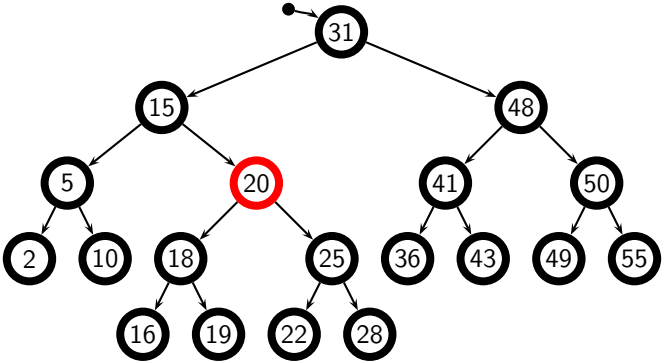


Red-Black Deletion (4)

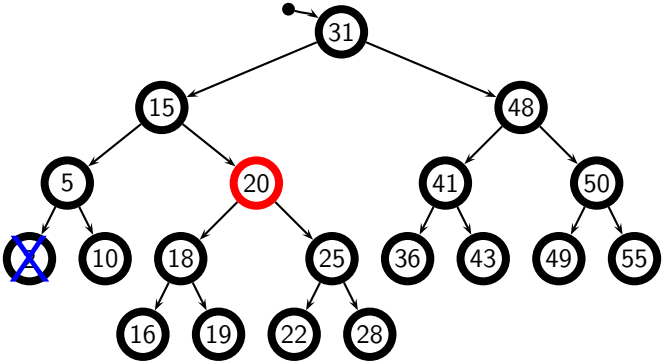


- The *additional* ***black*** *weight* can also stop as soon as it reaches a **red** node, which will absorb the extra **black** color

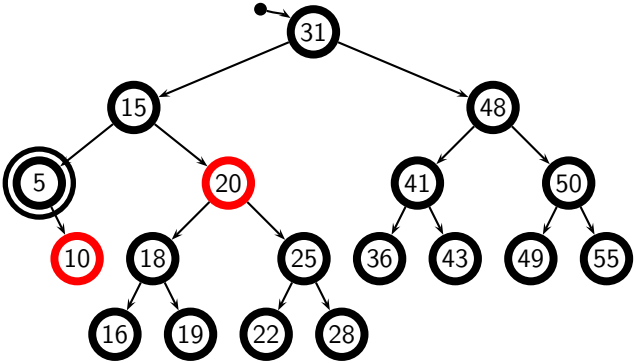
Red-Black Deletion (5)



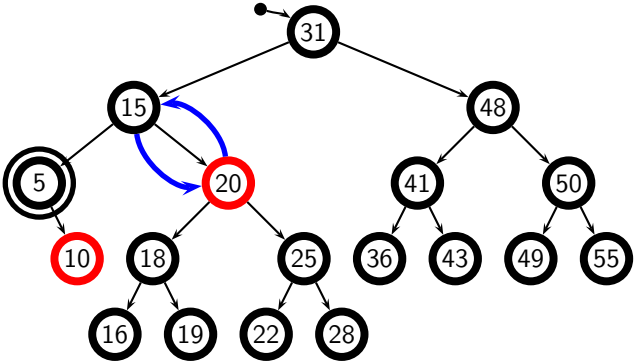
Red-Black Deletion (5)



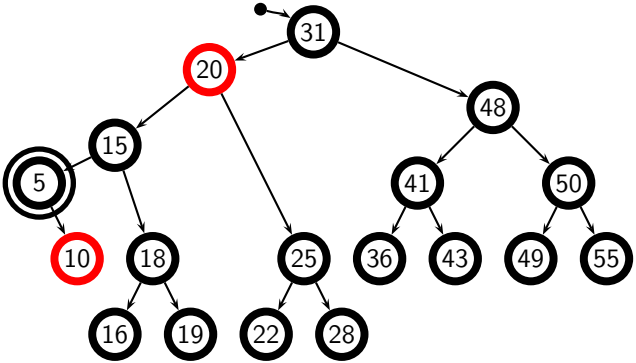
Red-Black Deletion (5)



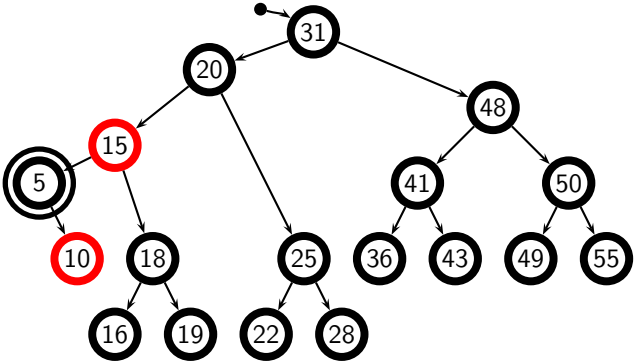
Red-Black Deletion (5)



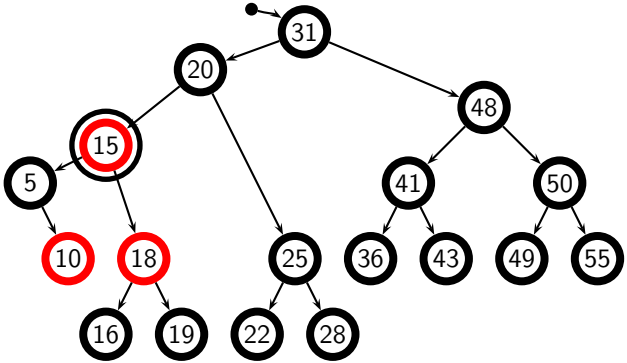
Red-Black Deletion (5)



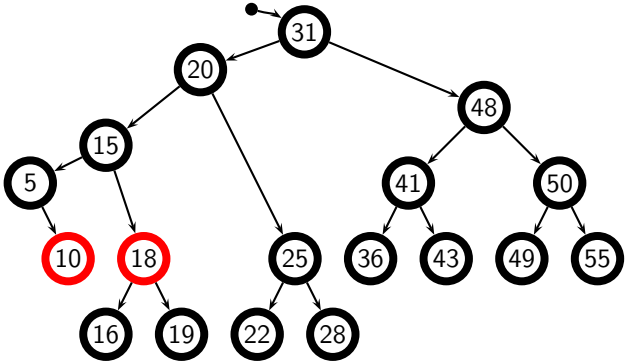
Red-Black Deletion (5)



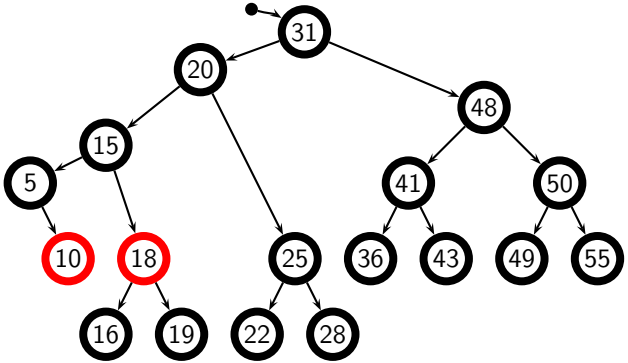
Red-Black Deletion (5)



Red-Black Deletion (5)



Red-Black Deletion (5)



- In other cases where we can not push the additional black color up, we can apply appropriate rotations and color transfers that preserve all other red-black properties

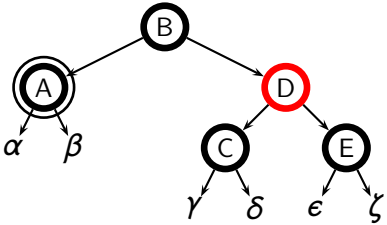
Basic Fixup Iteration (1)

Basic Fixup Iteration (1)

Case 1

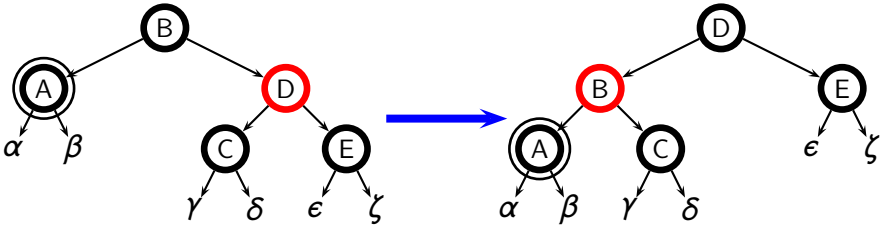
Basic Fixup Iteration (1)

Case 1



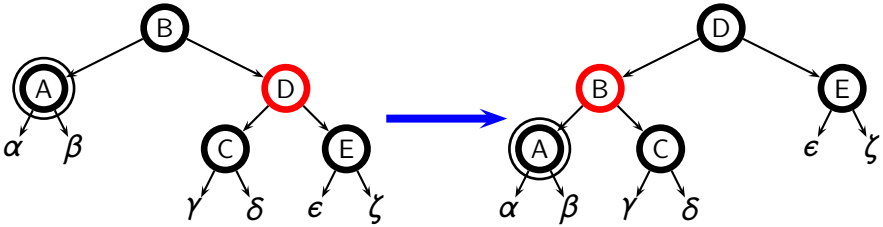
Basic Fixup Iteration (1)

Case 1



Basic Fixup Iteration (1)

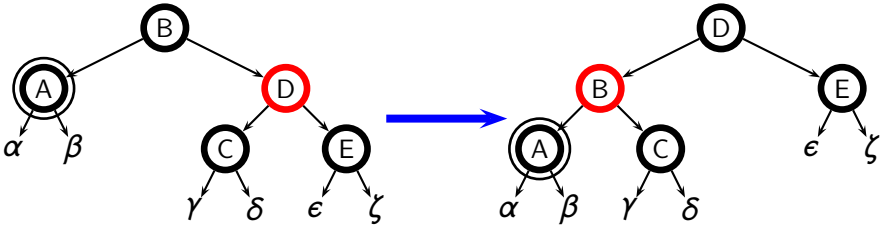
Case 1



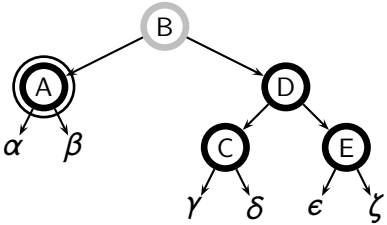
Case 2

Basic Fixup Iteration (1)

Case 1

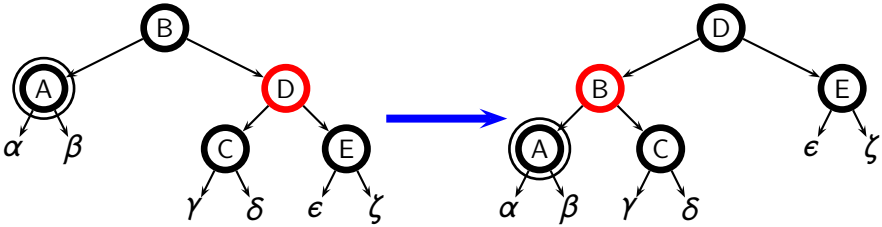


Case 2

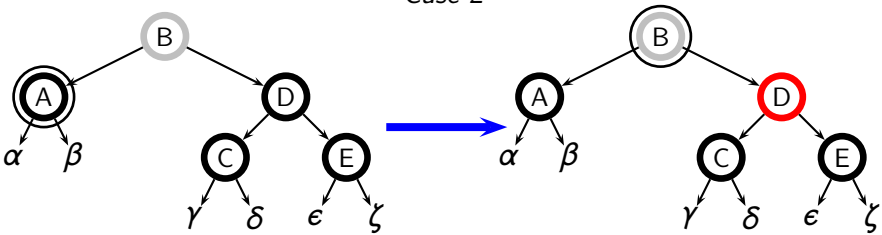


Basic Fixup Iteration (1)

Case 1



Case 2

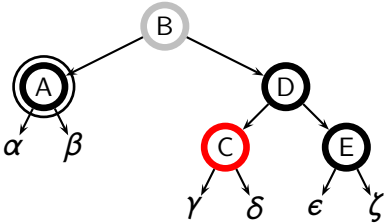


Basic Fixup Iteration (2)

Case 3

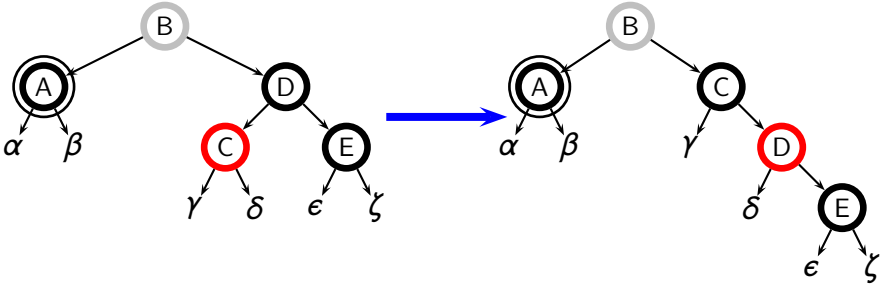
Basic Fixup Iteration (2)

Case 3



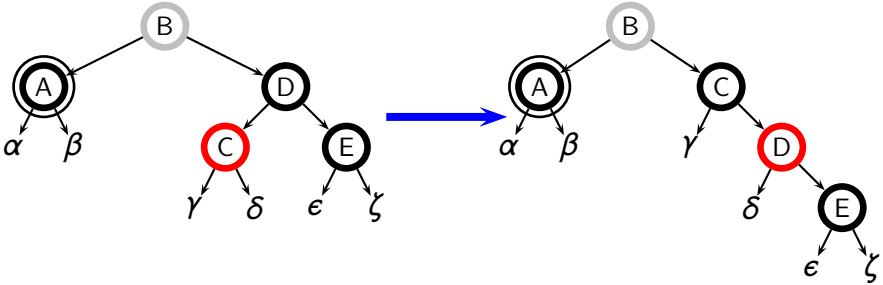
Basic Fixup Iteration (2)

Case 3

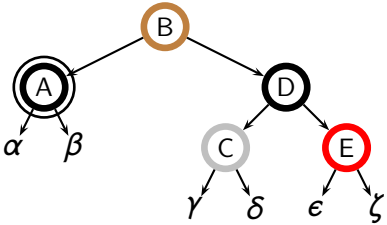


Basic Fixup Iteration (2)

Case 3

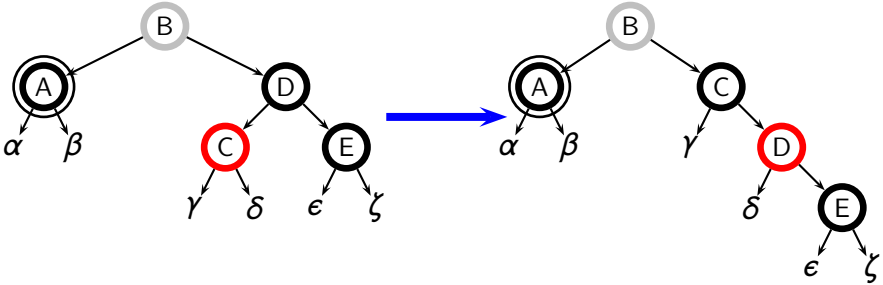


Case 4

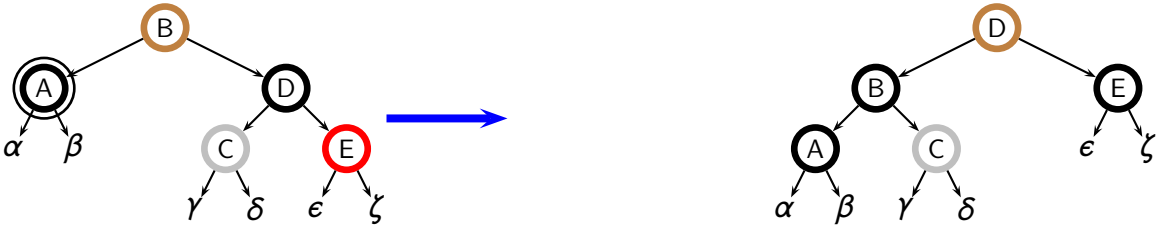


Basic Fixup Iteration (2)

Case 3



Case 4



Red-Black Delete Fixup

```
RB-Delete-Fixup(T, x)
while x ≠ T.root ∧ x.color = BLACK
    if x == x.parent.left
        w = x.parent.right
        if w.color == RED
            case 1...
        if w.left.color == BLACK ∧ w.right.color = BLACK
            w.color = RED                    // case 2
            x = x.parent
        else if w.right.color == BLACK
                case 3...
            case 4...
    else same as above, exchanging right and left
x.color = BLACK
```

Where we are (II)

Why did we want dynamic sets to start with ?

- To improve algorithms.
- Today: brief respite from data structures.
- Computational geometry
- We'll see some algorithms that use stacks, red-black trees.

Computational geometry

- Studies algorithms for geometric problems.
- Applications: computer graphics, robotics, VLSI, CAD.
- More applications: protein folding, molecular modeling, GIS.
- Huge area ! Only a sampler.
- Scientific conference: SOCG
- Software: CGAL.

Caution

- The biggest "enemy" to algorithms in computational geometry: degeneracy.
- Three points are collinear, three lines intersect at the same point, etc.
- Algorithms need patching to deal with degenerate situations.
- In the interest of teaching: Ignore it.

Want to know more ?

- De Berg, Cheong Kreveld, Overmars. "Computational Geometry: Algorithms and Applications", Third Edition, Springer.
- O'Rourke "Computational Geometry in C".

Computational geometry

- Input: set of points $\{p_i\}$, $p_i = (x_i, y_i)$. Example: polygon $P = (p_0, p_1, \ldots, p_n)$.
- Given $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, convex combination: any point $p_3 = (x_3, y_3)$ such that $x_3 = \lambda x_1 + (1 - \lambda)x_2$, $\lambda \in [0, 1]$, similarly $y_3 = \lambda y_1 + (1 - \lambda)y_2$.

1. Given two directed segments $\overrightarrow{p_0p_1}$ and $\overrightarrow{p_0p_2}$, is $\overrightarrow{p_0p_1}$ clockwise from $\overrightarrow{p_0p_2}$ with respect to their common endpoint p_0?
2. Given two line segments $\overline{p_1p_2}$ and $\overline{p_2p_3}$, if we traverse $\overline{p_1p_2}$ and then $\overline{p_2p_3}$, do we make a left turn at point p_2?
3. Do line segments $\overline{p_1p_2}$ and $\overline{p_3p_4}$ intersect?

Cross products

$$
\begin{aligned}
p_1 \times p_2 &= \det\begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\
&= x_1 y_2 - x_2 y_1 \\
&= -p_2 \times p_1 \ .
\end{aligned}
$$

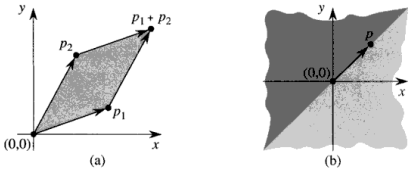


Figure 33.1 **(a)** The cross product of vectors p_1 and p_2 is the signed area of the parallelogram. **(b)** The lightly shaded region contains vectors that are clockwise from p. The darkly shaded region contains vectors that are counterclockwise from p.

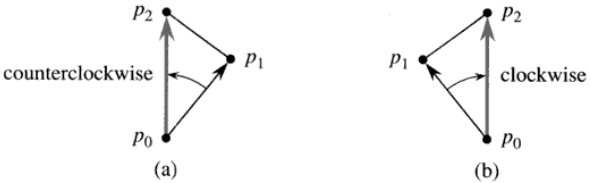


Figure 33.2 Using the cross product to determine how consecutive line segments $\overline{p_0 p_1}$ and $\overline{p_1 p_2}$ turn at point p_1. We check whether the directed segment $\overrightarrow{p_0 p_2}$ is clockwise or counterclockwise relative to the directed segment $\overrightarrow{p_0 p_1}$. **(a)** If counterclockwise, the points make a left turn. **(b)** If clockwise, they make a right turn.

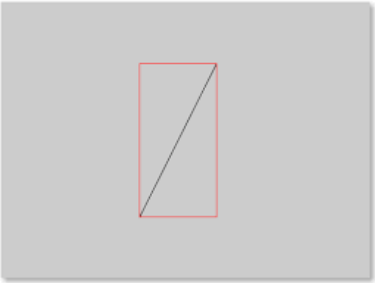
Procedures DIRECTION and ON-SEGMENT

```
DIRECTION(p_i, p_j, p_k)
return (p_k − p_i) × (p_j − p_i)
```

```
ON-SEGMENT(p_i, p_j, p_k)
if min(x_i, x_j) ≤ x_k ≤ max(x_i, x_j) and min(y_i, y_j) ≤ y_k ≤ max(y_i, y_j)
    then return TRUE
    else return FALSE
```

Testing whether two segments intersect

- QUICK REJECT: two segments cannot intersect if their BOUNDING BOXES don't.
- Smallest rectangle containing the segment with sides parallel to the xy axes.
- Bounding box of $\overline{p_1 p_2}$, $p_i = (x_i, y_i)$ is rectangle with corners $(min(x_1, x_2), min(y_1, y_2)$, $(min(x_1, x_2), max(y_1, y_2)$ $(max(x_1, x_2), max(y_1, y_2)$ and $(max(x_1, x_2), min(y_1, y_2)$.



Straddling

- Second stage: check whether each segment "straddles" the other.
- A segment $\overline{p_1 p_2}$ straddles a line if point p_1 lies on one side of the line and point p_2 lies on the other side. If p_1 or p_2 lies on the line, then we say that the segment straddles the line. Two line segments intersect if and only if they pass the quick rejection test and each segment straddles the line containing the other.

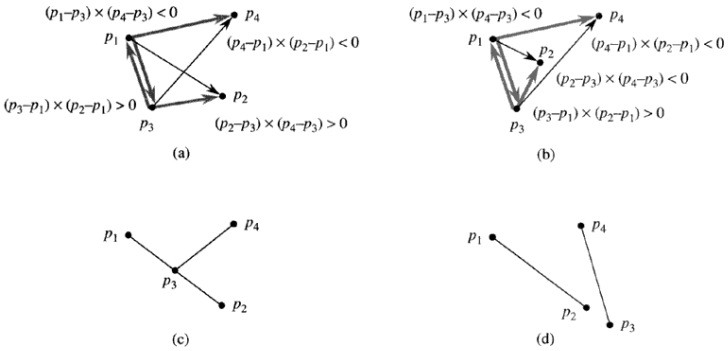


Figure 33.3 Cases in the procedure SEGMENTS-INTERSECT. **(a)** The segments $\overline{p_1 p_2}$ and $\overline{p_3 p_4}$ straddle each other's lines. Because $\overline{p_3 p_4}$ straddles the line containing $\overline{p_1 p_2}$, the signs of the cross products $(p_3 - p_1) \times (p_2 - p_1)$ and $(p_4 - p_1) \times (p_2 - p_1)$ differ. Because $\overline{p_1 p_2}$ straddles the line containing $\overline{p_3 p_4}$, the signs of the cross products $(p_1 - p_3) \times (p_4 - p_3)$ and $(p_2 - p_3) \times (p_4 - p_3)$ differ. **(b)** Segment $\overline{p_3 p_4}$ straddles the line containing $\overline{p_1 p_2}$, but $\overline{p_1 p_2}$ does not straddle the line containing $\overline{p_3 p_4}$. The signs of the cross products $(p_1 - p_3) \times (p_4 - p_3)$ and $(p_2 - p_3) \times (p_4 - p_3)$ are the same. **(c)** Point p_3 is collinear with $\overline{p_1 p_2}$ and is between p_1 and p_2. **(d)** Point p_3 is collinear with $\overline{p_1 p_2}$, but it is not between p_1 and p_2. The segments do not intersect.

Testing whether two segments intersect

```
SEGMENTS-INTERSECT(p1, p2, p3, p4)
d1 ← DIRECTION(p3, p4, p1)
d2 ← DIRECTION(p3, p4, p2)
d3 ← DIRECTION(p1, p2, p3)
d4 ← DIRECTION(p1, p2, p4)
if ((d1 > 0 and d2 < 0) or (d1 < 0 and d2 > 0)) and
        ((d3 > 0 and d4 < 0) or (d3 < 0 and d4 > 0))
    then return TRUE
elseif d1 = 0 and ON-SEGMENT(p3, p4, p1)
    then return TRUE
elseif d2 = 0 and ON-SEGMENT(p3, p4, p2)
    then return TRUE
elseif d3 = 0 and ON-SEGMENT(p1, p2, p3)
    then return TRUE
elseif d4 = 0 and ON-SEGMENT(p1, p2, p4)
    then return TRUE
else return FALSE
```

Testing whether any two segments intersect

- Given: n segments $v_1, \ldots v_n$.
- To test: do any two segments intersect ?
- Uses technique called sweeping.
- Running time: $O(n \log n)$. Naive algorithm $O(n^2)$.
- SWEEPING: an imaginary vertical sweep line passes through the given set of geometric objects, usually from left to right. The spatial dimension that the sweep line moves across, in this case the x-dimension, is treated as a dimension of time.
- Provides method for ordering geometric objects, usually by placing them into a dynamic data structure, and for taking advantage of relationships among them.
- line-segment-intersection algorithm: considers all line-segment endpoints in left-to-right order and checks for an intersection each time it encounters an endpoint.

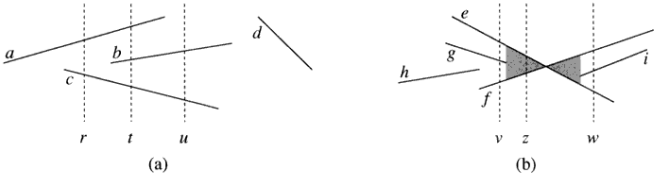


Figure 33.4 The ordering among line segments at various vertical sweep lines. **(a)** We have $a >_r c$, $a >_t b$, $b >_t c$, $a >_t c$, and $b >_u c$. Segment d is comparable with no other segment shown. **(b)** When segments e and f intersect, their orders are reversed: we have $e >_v f$ but $f >_w e$. Any sweep line (such as z) that passes through the shaded region has e and f consecutive in its total order.

Maintaining sweep line

- Sweeping algorithms: maintain two sets of data.
- sweep-line status: gives the relationships among objects intersected by the sweep line.
- event-point schedule: sequence of x-coordinates, ordered from left to right, that defines the halting positions of the sweep line.
- Call each such halting position an event point. Changes to the sweep-line status occur only at event points.
- Sweep-line status: total order T.
- $INSERT(T, s)$, $DELETE(T, s)$.
- $ABOVE(T, s)$: return segment above s in T.
- $BELOW(T, s)$: return segment below s in T.
- We can perform each of the above operations in $O(\log n)$ time using red-black trees.

Algorithm

```
ANY-SEGMENTS-INTERSECT(S)
T ← ∅
sort the endpoints of the segments in S from left to right,
        breaking ties by putting left endpoints before right endpoints
        and breaking further ties by putting points with lower
        y-coordinates first
for each point p in the sorted list of endpoints
    do if p is the left endpoint of a segment s
        then INSERT(T, s)
            if (ABOVE(T, s) exists and intersects s)
                or (BELOW(T, s) exists and intersects s)
                then return TRUE
        if p is the right endpoint of a segment s
        then if both ABOVE(T, s) and BELOW(T, s) exist
                and ABOVE(T, s) intersects BELOW(T, s)
                then return TRUE
            DELETE(T, s)
return FALSE
```

Algorithm: example

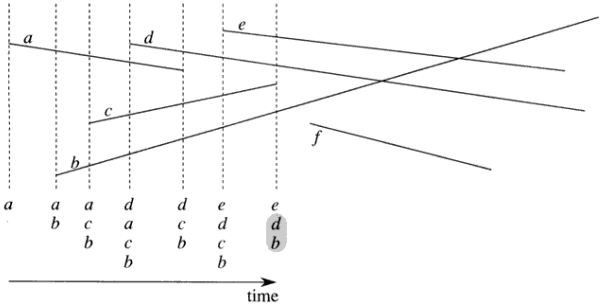


Figure 33.5 The execution of ANY-SEGMENTS-INTERSECT. Each dashed line is the sweep line at an event point, and the ordering of segment names below each sweep line is the total order T at the end of the **for** loop in which the corresponding event point is processed. The intersection of segments d and b is found when segment c is deleted.

Algorithm: correctness/performance

- Can only fail by not reporting intersecting segments.
- p = leftmost intersection point, breaking ties by choosing the one with the lowest y-coordinate. a and b = the segments that intersect at p.
- No intersections occur to the left of $p \Longrightarrow$ the order given by T is correct at all points to the left of p.
- no three segments intersect at the same point $\Longrightarrow$ there exists a sweep line z at which a and b become consecutive in the total order.
- z is to the left of p or goes through p.
- There exists segment endpoint q on z that is the event point at which a and b become consecutive.
- If p is on z, then $q = p$. If p is not on z, then q is to the left of p. In either case, the order given by T is correct just before q is processed.

Algorithm: correctness/performance

- Either a or b is inserted into T, and the other segment is above or below it in the total order. Lines 4-7 detect this case.
- Segments a and b are already in T, and a segment between them in the total order is deleted, making a and b become consecutive. Lines 8-11.
- In either case, the intersection p is found.
- $2n$ insert/delete/tests. Taking $O(\log n)$ time.

Convex hull

- Convex hull of a set of points: smallest convex polygon that contains the set of points.
- place elastic rubber band around set of points and let it shrink.
- Two algorithms: Graham's Scan $O(n \log n)$.
- Jarvis's March $O(n \cdot h)$, h the number of points on the convex hull.
- Other algorithms:
- Incremental: points sorted from left to right forming sequence $p_1, \ldots, p_n$. At stage i add p_i to convex hull $CH(p_1, \ldots, p_{i-1})$, forming $CH(p_1, \ldots, p_i)$.
- Divide-and-conquer: divide into leftmost n/2 points and rightmost n/2 points. Compute convex hulls and combine them.
- Prune-and-search method.

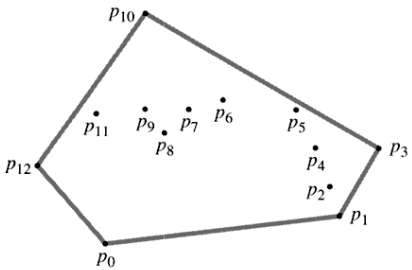


Figure 33.6 A set of points $Q = \{p_0, p_1, \ldots, p_{12}\}$ with its convex hull CH(Q) in gray.

Graham's scan

- Maintains a stack S of candidate points.
- Each point of Q is pushed onto the stack.
- Points not in $CH(Q)$ eventually popped from the stack.
- $TOP(S)$, $NEXT - TO - TOP(S)$: stack functions, do not change its contents.
- Stack returned by the algorithm: points of $CH(Q)$ in counterclockwise order.

Convex hull algorithm

```
GRAHAM-SCAN(Q)
let p_0 be the point in Q with the minimum y-coordinate,
        or the leftmost such point in case of a tie
let ⟨p_1, p_2, ..., p_m⟩ be the remaining points in Q,
        sorted by polar angle in counterclockwise order around p_0
        (if more than one point has the same angle, remove all but
        the one that is farthest from p_0)
PUSH(p_0, S)
PUSH(p_1, S)
PUSH(p_2, S)
for i ← 3 to m
    do while the angle formed by points NEXT-TO-TOP(S), TOP(S),
                and p_i makes a nonleft turn
            do POP(S)
        PUSH(p_i, S)
```

Graham's Scan:Example

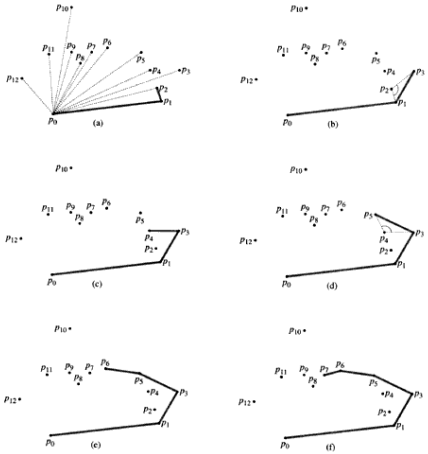
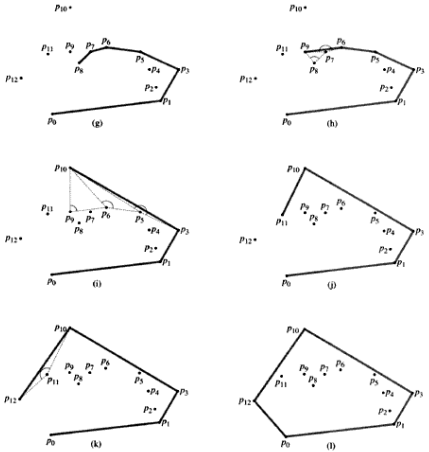


Figure 33.7 The execution of GRAHAM-SCAN on the set Q of Figure 33.6. The current convex hull contained in stack S is shown in gray at each step. **(a)** The sequence $\langle p_1, p_2, \ldots, p_{12} \rangle$ of points numbered in order of increasing polar angle relative to p_0, and the initial stack S containing p_0, p_1, and p_2. **(b)–(k)** Stack S after each iteration of the **for** loop of lines 6–9. Dashed lines show nonleft turns, which cause points to be popped from the stack. In part (h), for example, the right turn at angle $\angle p_3\, p_8\, p_9$ causes p_8 to be popped, and then the right turn at angle $\angle p_6\, p_7\, p_9$ causes p_7 to be popped. **(l)** The convex hull returned by the procedure, which matches that of Figure 33.6.

Graham's Scan:Example



Graham's Scan: Correctness and Performance

- Invariant: at the beginning of each iteration of the for loop stack S contains (from bottom to top) exactly the vertices of $CH(Q_{i-1})$ in counterclockwise order.
- Line 1: $\theta(n)$ time.
- Sorting $\theta(n \log n)$ time.
- Testing for left/right turn: vector product $\theta(1)$ time.
- The rest of the algorithm $O(n)$ time.

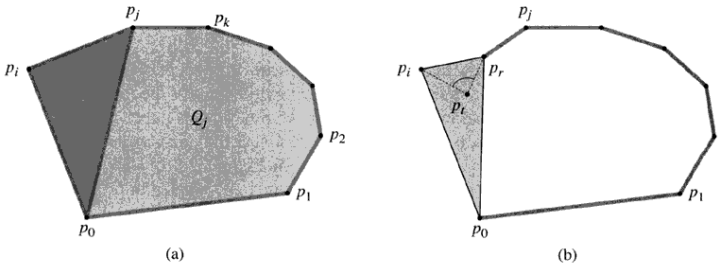


Figure 33.8 The proof of correctness of GRAHAM-SCAN. **(a)** Because p_i's polar angle relative to p_0 is greater than p_j's polar angle, and because the angle $\angle p_k p_j p_i$ makes a left turn, adding p_i to $\mathrm{CH}(Q_j)$ gives exactly the vertices of $\mathrm{CH}(Q_j \cup \{p_i\})$. **(b)** If the angle $\angle p_r p_t p_i$ makes a nonleft turn, then p_t is either in the interior of the triangle formed by p_0, p_r, and p_i or on a side of the triangle, and it cannot be a vertex of $\mathrm{CH}(Q_i)$.

Jarvis's March

- uses a technique known as gift wrapping.
- Simulates wrapping a piece of paper around set Q.
- Start at the same point p_0 as in Graham's scan.
- Pull the paper to the right, then higher until it touches a point. This point is a vertex in the convex hull. Continue this way until we come back to p_0.
- Formally: start at p_0. Choose p_1 as the point with the smallest polar angle from p_0. Choose p_2 as the point with the smallest polar angle from $p_1 \ldots$
- $\ldots$ until we reached the highest point p_k.
- We have constructed the right chain.
- Construct the left chain by starting from p_k and measuring polar angles with respect to the negative x-axis.

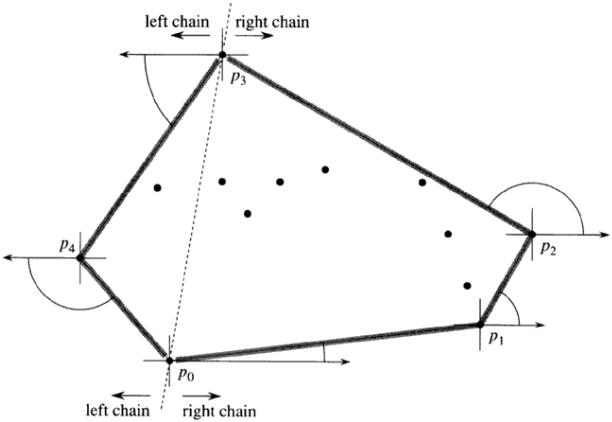


Figure 33.9 The operation of Jarvis's march. The first vertex chosen is the lowest point p_0. The next vertex, p_1, has the smallest polar angle of any point with respect to p_0. Then, p_2 has the smallest polar angle with respect to p_1. The right chain goes as high as the highest point p_3. Then, the left chain is constructed by finding smallest polar angles with respect to the negative x-axis.