Algorithms & Data Structures

Lecture 08
Binary Search Trees

Giovanni Bacci
giovbacci@cs.aau.dk

Outline

- Binary Search Trees
- Operations on Binary Search Trees
- Red-Back Trees
- Operations on Red-Back Trees

Intended Learning Goals

KNOWLEDGE

- Mathematical reasoning on concepts such as recursion, induction, concrete and abstract computational complexity
- Data structures, algorithm principles e.g., search trees, hash tables, dynamic programming, divide-and-conquer
- Graphs and graph algorithms e.g., graph exploration, shortest path, strongly connected components.

SKILLS

- Determine abstract complexity for specific algorithms
- Perform complexity and correctness analysis for simple algorithms
- Select and apply appropriate algorithms for standard tasks

COMPETENCES

- Ability to face a non-standard programming assignment
- Develop algorithms and data structures for solving specific tasks
- Analyse developed algorithms

Why Binary Search Trees

- Binary search trees support many dynamic-set operations, including Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete

- We can use search trees both as dictionary and as priority queue.

- Many basic operations on binary search trees take time proportional to the height of the tree. If the tree is balanced this corresponds to $\Theta(\lg n)$

Binary Search Trees

Binary Search Trees

- BSTs can be represented by means of linked data structures where each node has
  - A key (and optional satellite data)
  - Links to its parent node, left child, and right child
  - The root node is the only node in the tree whose parent is Nil
  - Keys in the tree satisfy the binary-search-tree property

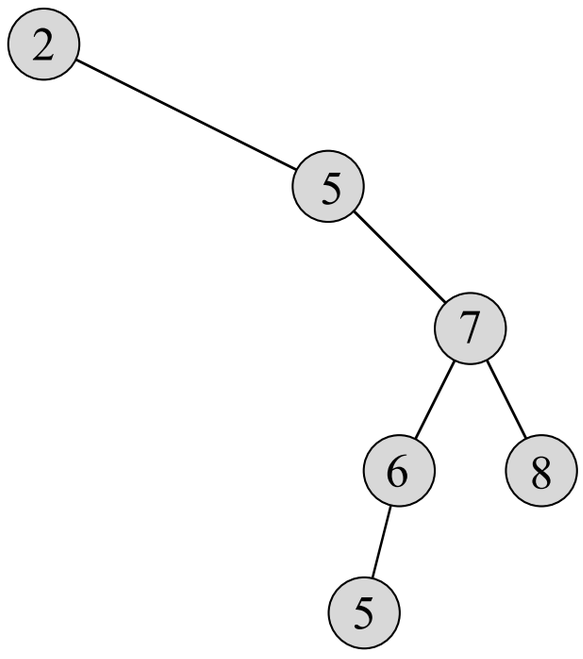
Binary-search-tree property:

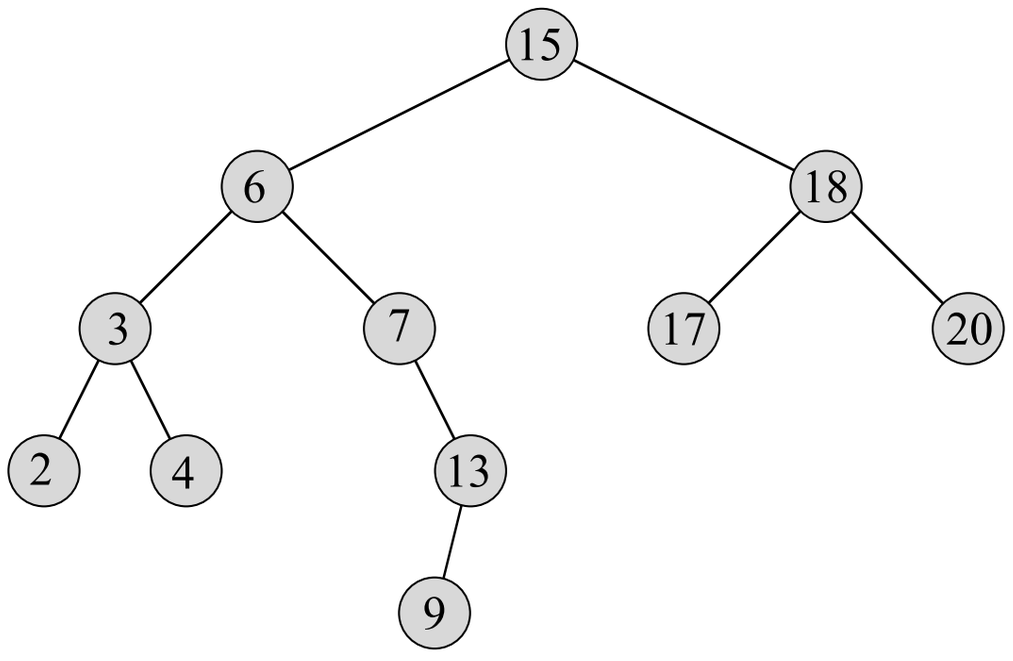
For any node node x in the tree, the following hold

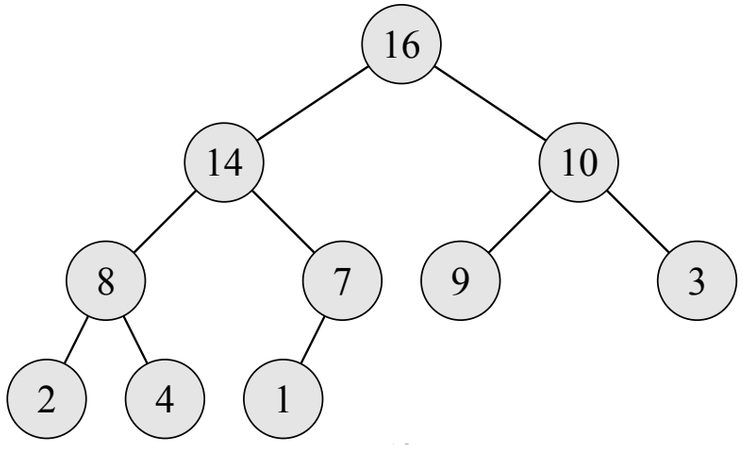
- If y is a node in the left subtree of x then $y\,.\,key \leq x\,.\,key$
- If z is a node in the right subtree of x then $x\,.\,key \leq z\,.\,key$

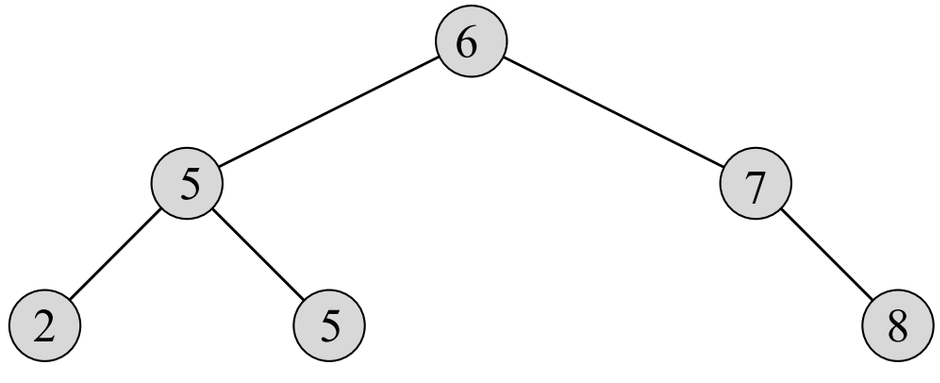
Quiz

Which of the following is a BST?









Inorder Tree Walk

- The binary-search-tree property allows one to print out all the keys in a BST in sorted order by a simple inorder visit of the tree, called **inorder tree walk**

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

First recursively visit the left child, then print the node x, and recursively visit the right child

- Recall from Exercise Session 05 that this way of visiting the tree takes $\Theta(n)$ time for an n-node binary tree (see also Theorem 12.1 CLRS)

Other Visiting Orders

Similarly to the **inorder tree walk**, we can visit the tree using

- **Preorder tree walk:** prints the root before making the recursive calls to the left and right children

- **Postorder tree walk:** prints the root after returning from the recursive calls to the left and right children

- Both procedures take $\Theta(n)$.

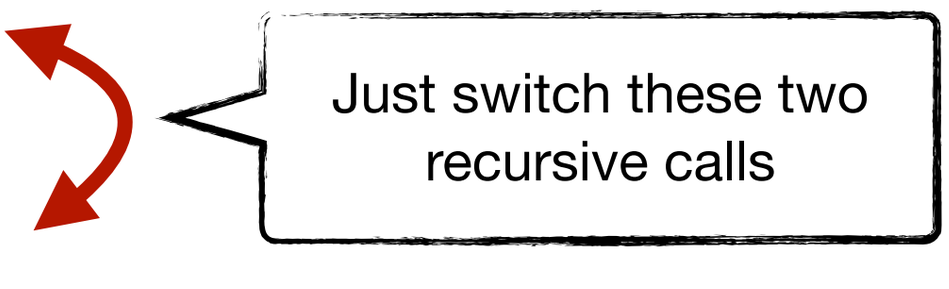
Quiz

Can you think of a procedure that prints all the elements of a binary search tree in reversed-sorted order?

Answer

Can you think of a procedure that prints all the elements of a binary search tree in reversed-sorted order?

```
INORDER-TREE-WALK(x)
1   if x ≠ NIL
2       INORDER-TREE-WALK(x.left)
3       print x.key
4       INORDER-TREE-WALK(x.right)
```



Searching in a BST

- Given a pointer to the root of the tree and a key k, TREE-SEARCH returns a pointer to a node with key k if it exists; otherwise it returns NIL
- **Exploit the BST property** to choose the appropriate subtree
- **Worst-case running time** $O(h)$ where h is the height of the tree rooted at x

```
TREE-SEARCH(x, k)
1   if x == NIL or k == x.key
2       return x
3   if k < x.key
4       return TREE-SEARCH(x.left, k)
5   else return TREE-SEARCH(x.right, k)
```

```
ITERATIVE-TREE-SEARCH(x, k)
1   while x ≠ NIL and k ≠ x.key
2       if k < x.key
3           x = x.left
4       else x = x.right
5   return x
```

Minimum and Maximum

- Given a pointer x to the root of the tree and a key k, MINIMUM and MAXIMUM resp. return a pointer to a node with minimal and maximal key k
- Exploit the **BST property to avoid comparisons**
- **Worst-case running time** $O(h)$ where h is the height of the tree rooted at x

TREE-MINIMUM(x)

```
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

TREE-MAXIMUM(x)

```
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

(The same procedures can be equivalently implemented using recursion)

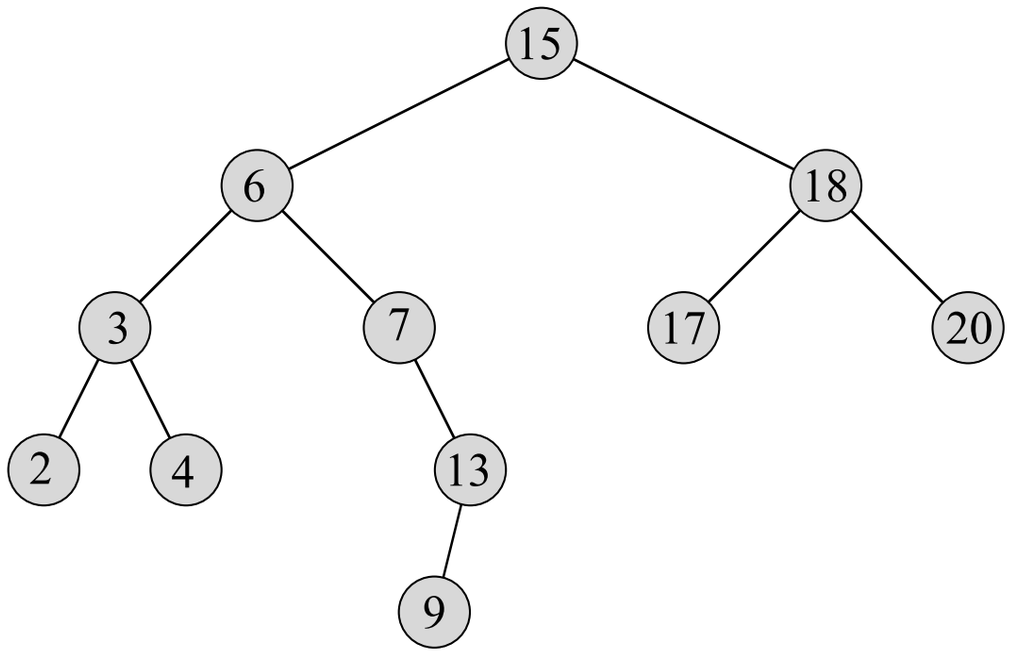
Successor

- We assume that all nodes in the tree are distinct.
- Given a node x in a BST, the procedure Tree-Successor(x) returns the node having the smallest key greater than $x\,.\,key$ if it exists; otherwise it returns Nil

Quiz

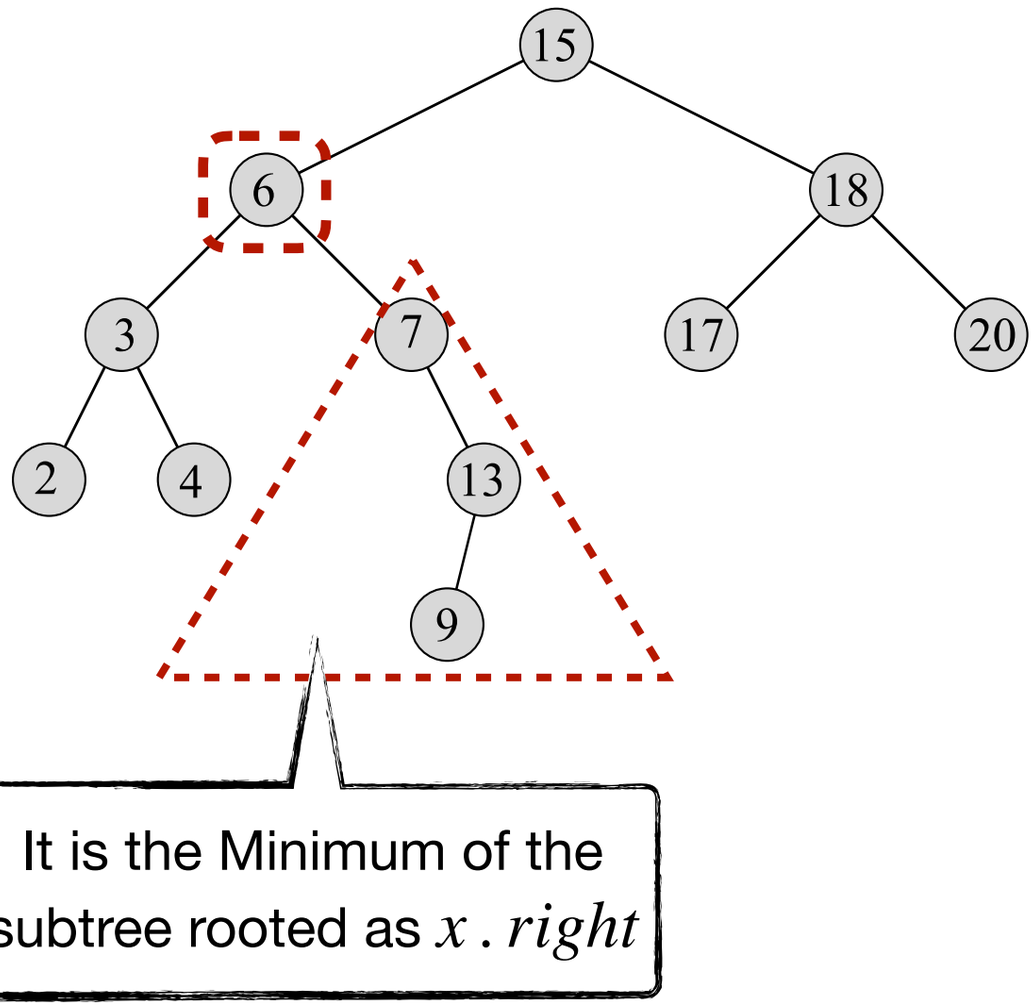
Consider the BST to the right

- Who is the successor of 6?
- And the successor of 13?
- Can we use the BST property to avoid comparisons?



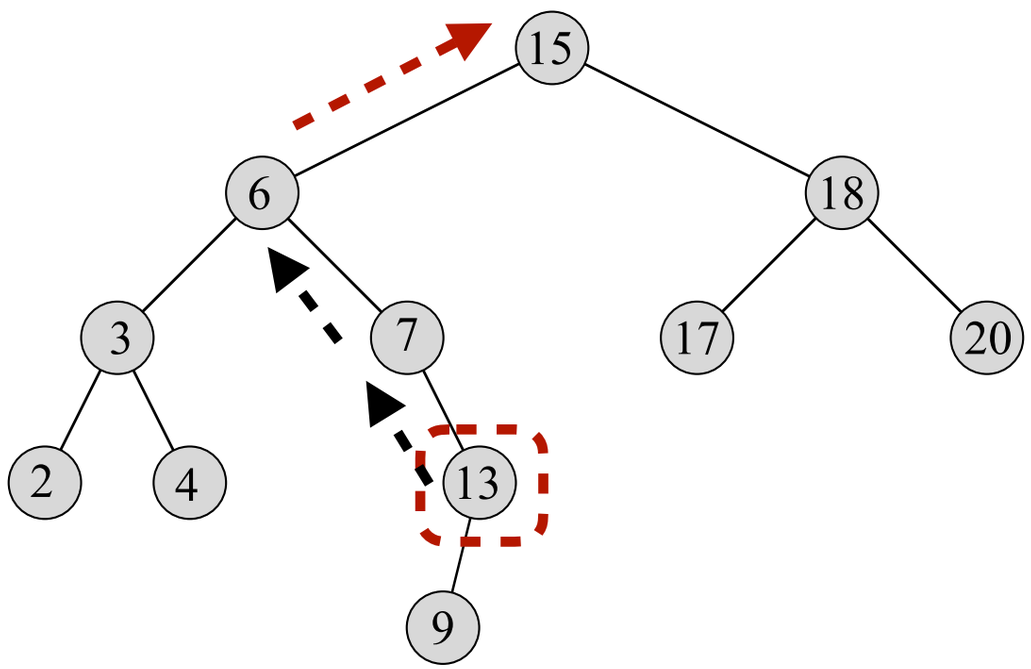
Answer

We can exploit the BST property for both cases!



It is the Minimum of the subtree rooted as $x.right$

It is the first ancestor who has 13 in its left subtree



Successor

- We assume that all nodes in the tree are distinct.
- Given a node x in a BST, the procedure Tree-Successor(x) returns the node having the smallest key greater than $x\,.\,key$ if it exists; otherwise it returns Nil

```
TREE-SUCCESSOR(x)
1   if x.right ≠ NIL
2       return TREE-MINIMUM(x.right)
3   y = x.p
4   while y ≠ NIL and x == y.right
5       x = y
6       y = y.p
7   return y
```

Case 1: minimum element in the right subtree

Case 2: move to the first ancestor who has x in the left subtree, by climbing up the parent pointers

Tree-Successor: run-time

- The running time of successor and predecessor operations is $\Theta(h)$ where h is the height of the whole BST

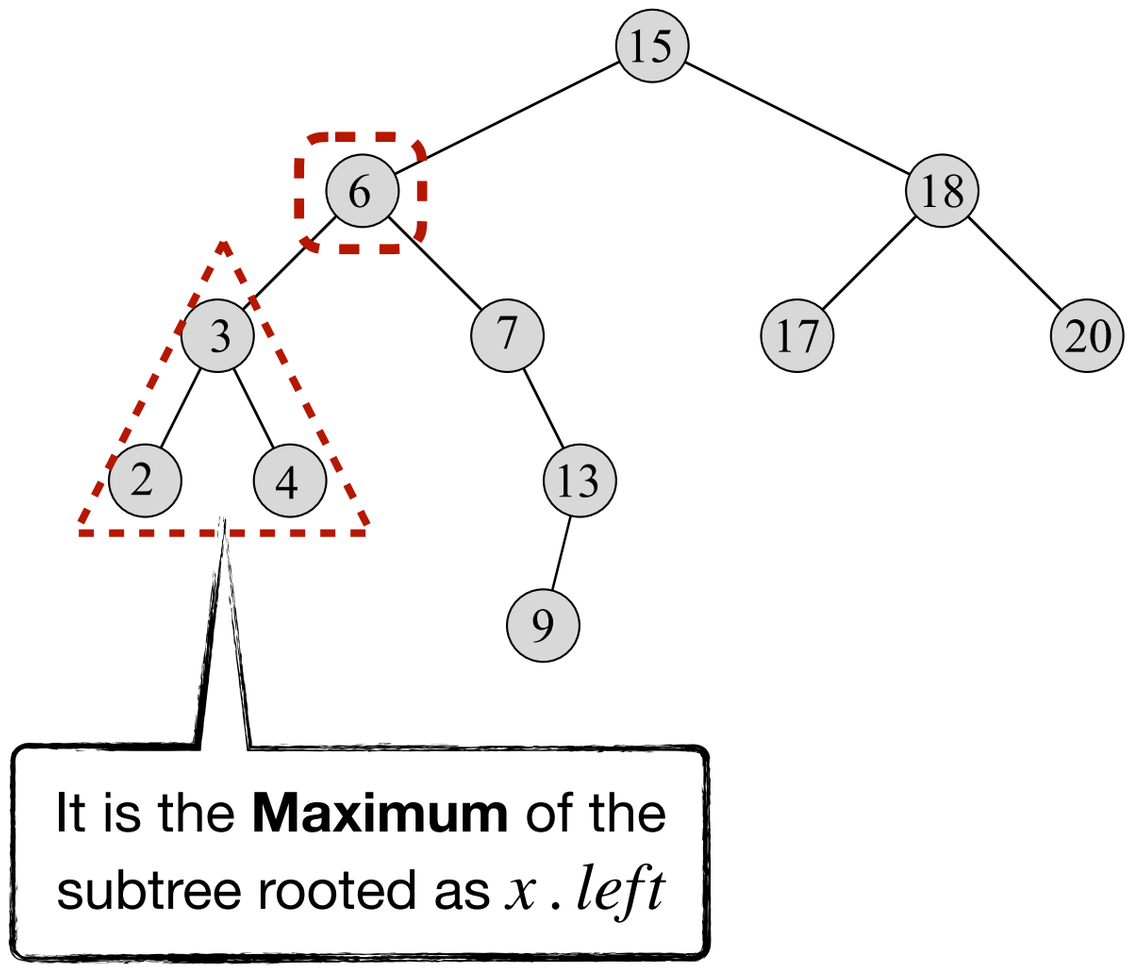
```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5      x = y
6      y = y.p
7  return y
```

Case 1: we have seen before $\Theta(h)$

Case 2: it visits a simple path which stops in the worst-case at the root.
The length of the path is $\Theta(h)$

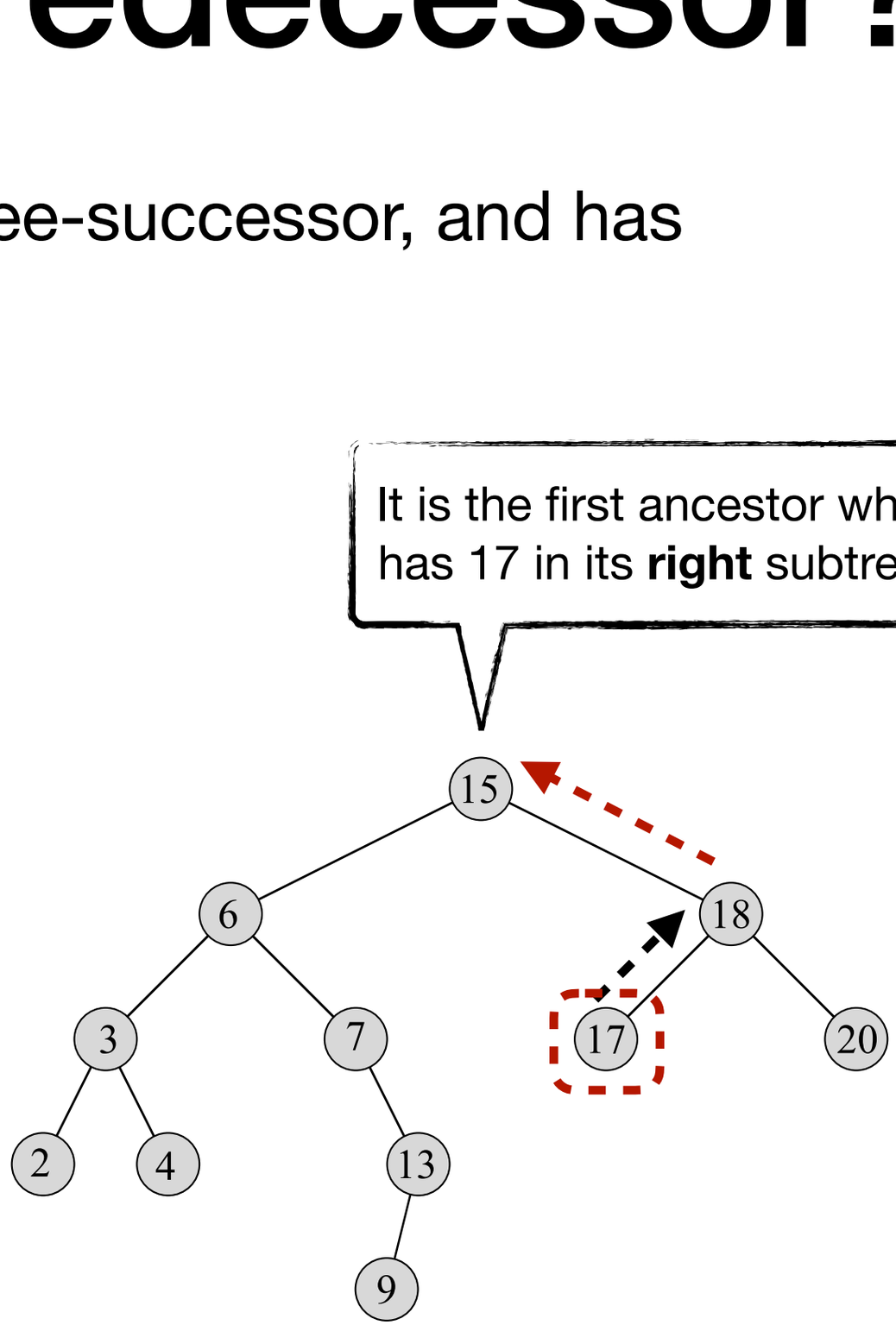
What about Predecessor?

The procedure is analogous to tree-successor, and has same running-time $\Theta(h)$



It is the **Maximum** of the subtree rooted as $x\,.\,left$

It is the first ancestor who has 17 in its **right** subtree

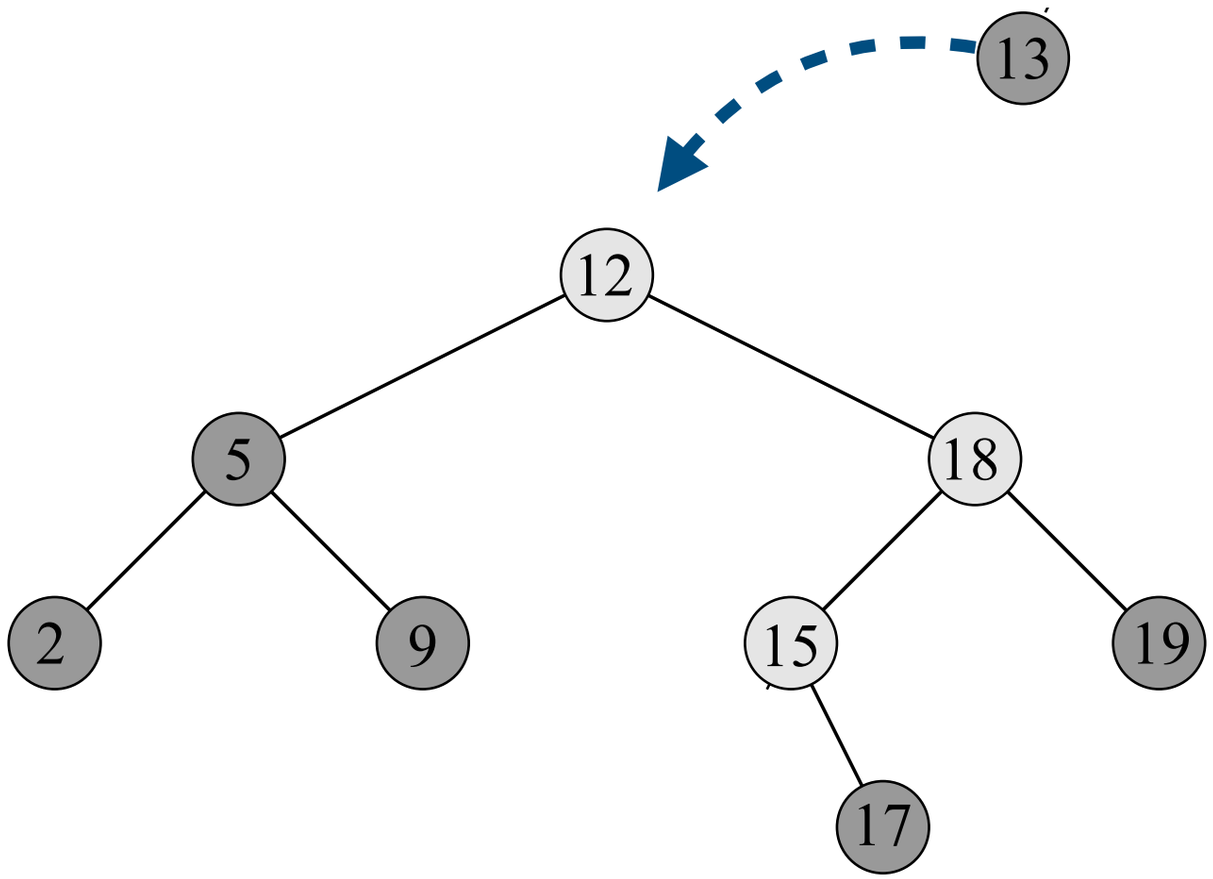


Insertion

The TREE-INSERT procedure takes a BST T and a node z and update the links so that z becomes a node of T while preserving the BST property

Basic Idea:

- Place the node as a new leaf of the tree
- Ensure the BST property holds by choosing an appropriate path starting from the root

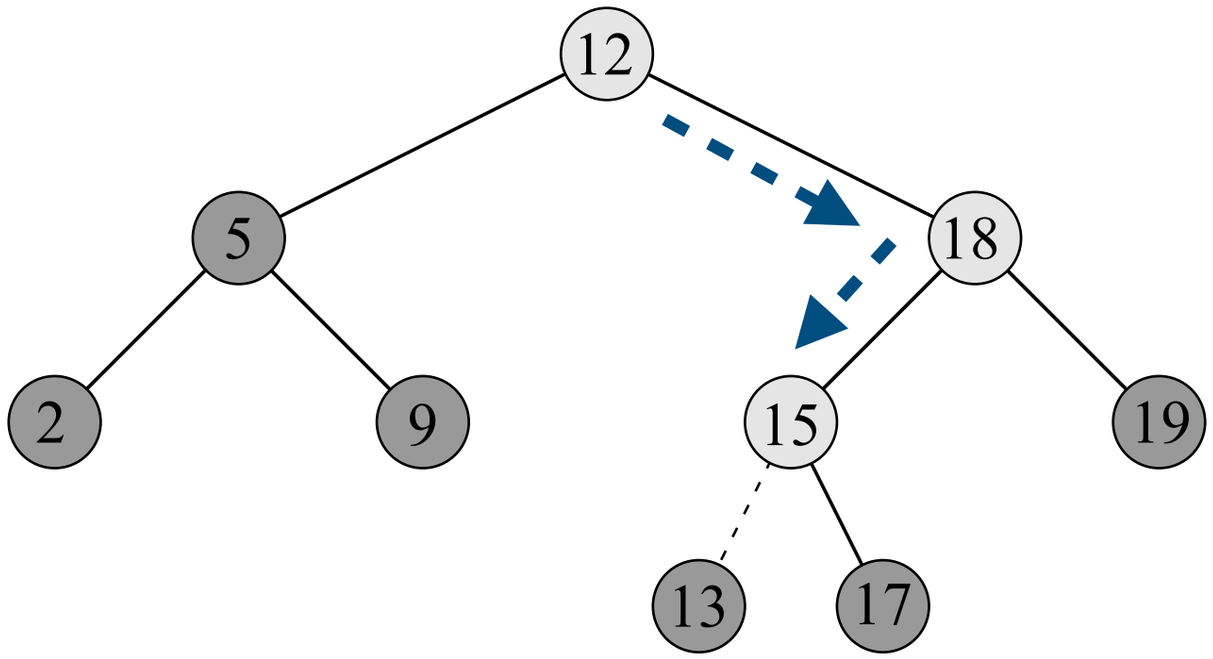


Insertion

The TREE-INSERT procedure takes a BST T and a node z and updates the pointers so that z becomes a node of T while preserving the BST property

Basic Idea:

- Place the node as a new leaf of the tree
- Ensure the BST property holds by choosing an appropriate path starting from the root



Tree-Insert: pseudocode

The TREE-INSERT procedure takes a BST T and a node z and update the links so that z becomes a node of T while preserving the BST property

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4      y = x
 5      if z.key < x.key
 6          x = x.left
 7      else x = x.right
 8  z.p = y
 9  if y == NIL
10      T.root = z
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
```

x is the current node and y its parent

Move x and y down in the tree comparing x.key with z.key

At this point x is Nil and z can replace it

Case 1: T was empty. Then, z becomes the root

Case 2: place z in the appropriate subtree by comparing z.key with y.key

Tree-Insert: run-time

The TREE-INSERT procedure takes a BST T and a node z and update the links so that z becomes a node of T while preserving the BST property

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.p = y
 9  if y == NIL
10        T.root = z
11  elseif z.key < y.key
12        y.left = z
13  else y.right = z
```

$\Theta(1)$

It visits a simple path from the root to a leaf node. Hence $\Theta(h)$

$\Theta(1)$

Deletion

The procedure for deleting takes a BST T and a node z in T, and updates the pointers so that z does no longer belong to T while preserving the BST property.

Basic Idea:

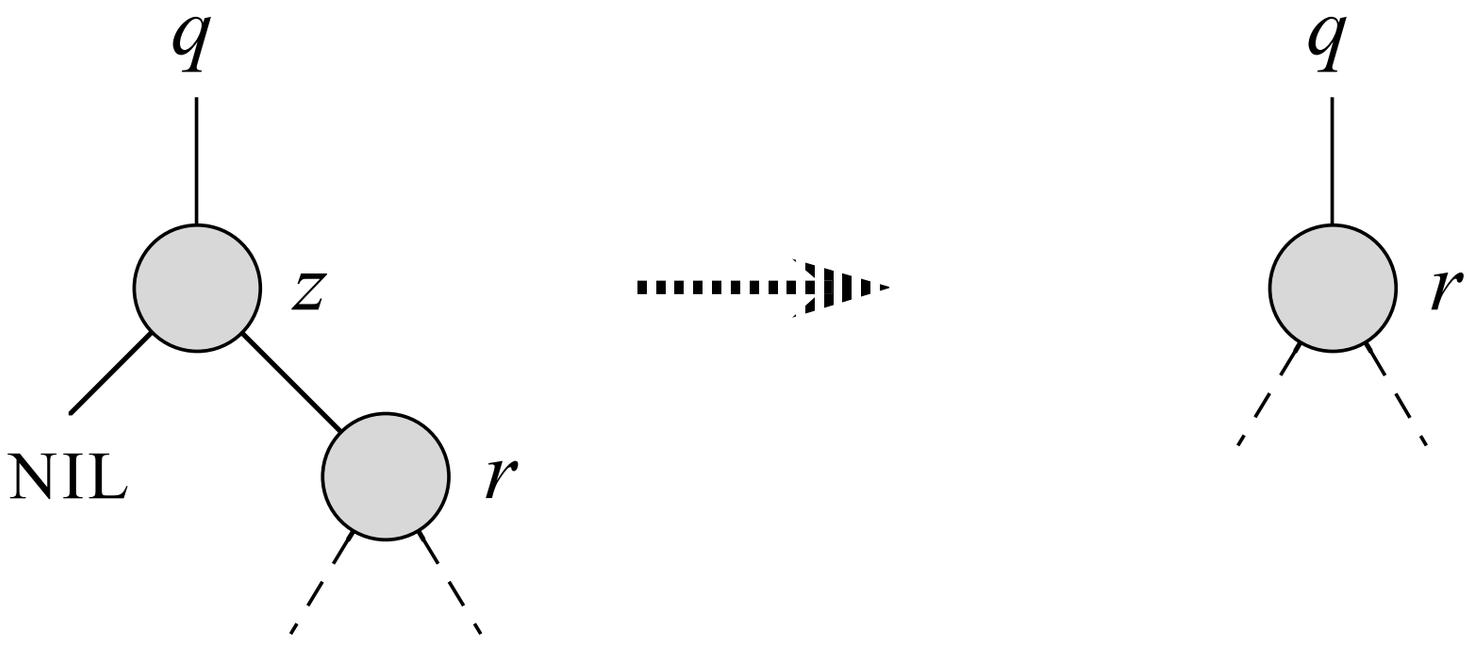
We consider three cases:

1. If z has no children we simply remove it
2. If z has just one child, then we elevate that child to take z's position in the tree.
3. If z has both children, then we find z's successor —which is in z's right subtree and has no left subtree— and have it take z's position in the tree

Tree-Delete

For the actual procedure, it will be more convenient to consider four cases instead:

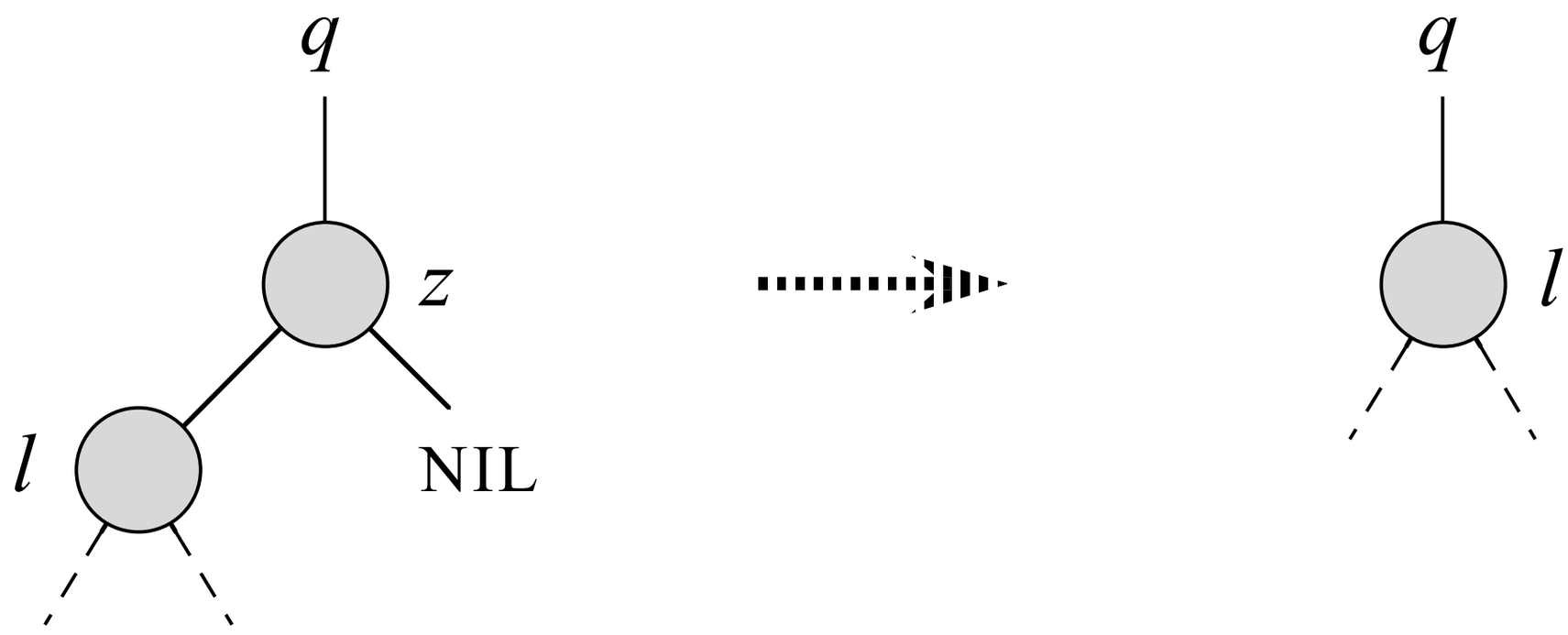
Case A: z has no left child. Then we replace z by its right child (which may be Nil)



Tree-Delete

For the actual procedure, it will be more convenient to consider four cases instead:

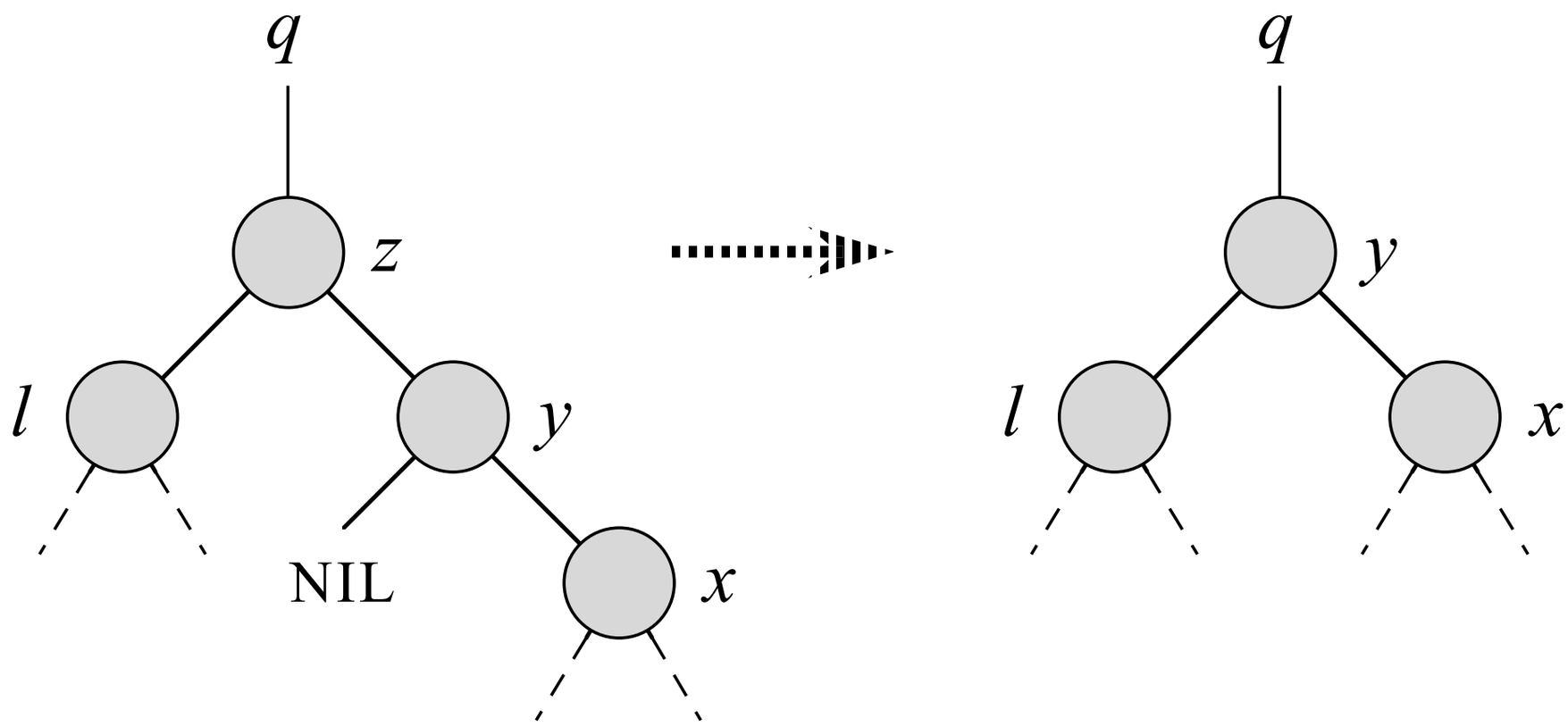
Case B: z has left child but no right child. Then we replace z by its left child



Tree-Delete

For the actual procedure, it will be more convenient to consider four cases instead:

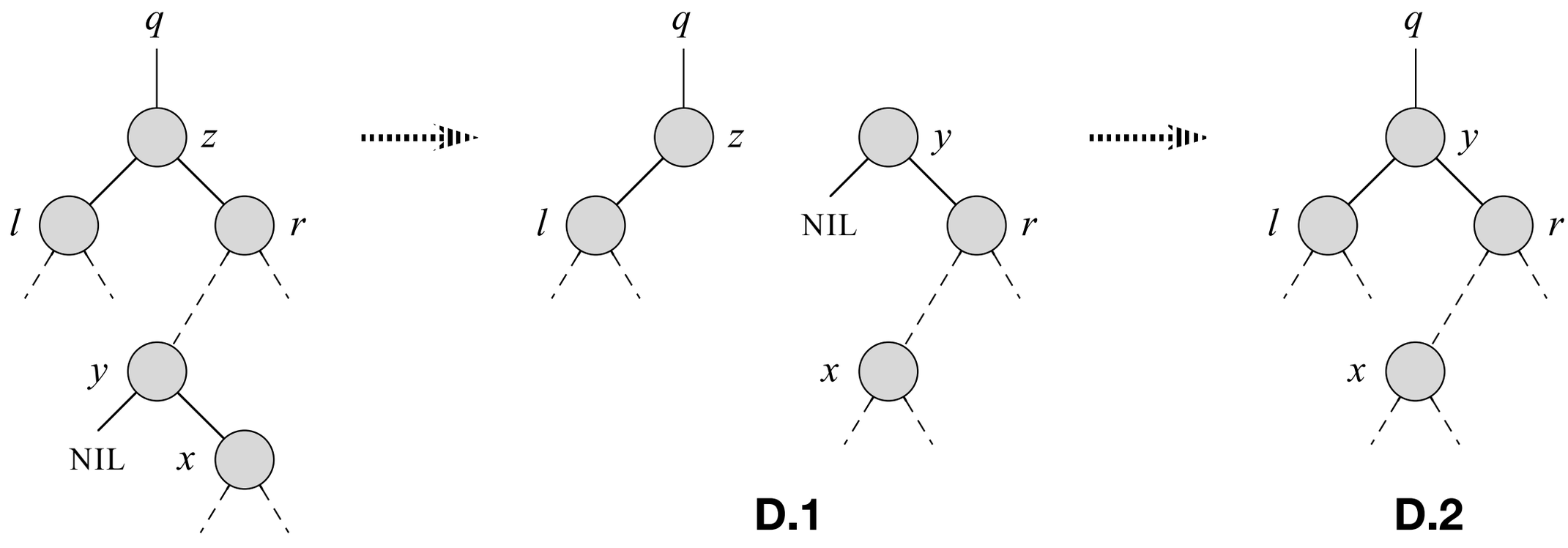
Case C: z has both children and z's successor is its right child. Then "push" y up replacing z's position



Tree-Delete

For the actual procedure, it will be more convenient to consider four cases instead:

Case D: z has both children and z's successor is not its right child. Then, we first remove y from T, after we replace y with z's position



D.1 **D.2**

The Transplant subroutine

- The pseudocode for TREE-DELETE will use a subroutine, called TRANSPLANT, that **replaces one subtree rooted at node u with the subtree rooted at node v.**
- Its worst-case **running-time** is $\Theta(1)$

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

Node u's parent ends up having v as its appropriate child

Node u's parent becomes node v's parent

Tree-Delete: pseudocode

```
TREE-DELETE(T, z)
1   if z.left == NIL
2       TRANSPLANT(T, z, z.right)
3   elseif z.right == NIL
4       TRANSPLANT(T, z, z.left)
5   else y = TREE-MINIMUM(z.right)
6       if y.p ≠ z
7           TRANSPLANT(T, y, y.right)
8           y.right = z.right
9           y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

Case A

Case B

Case D.1

Case C
+
Case D.2

Tree-Delete: run-time

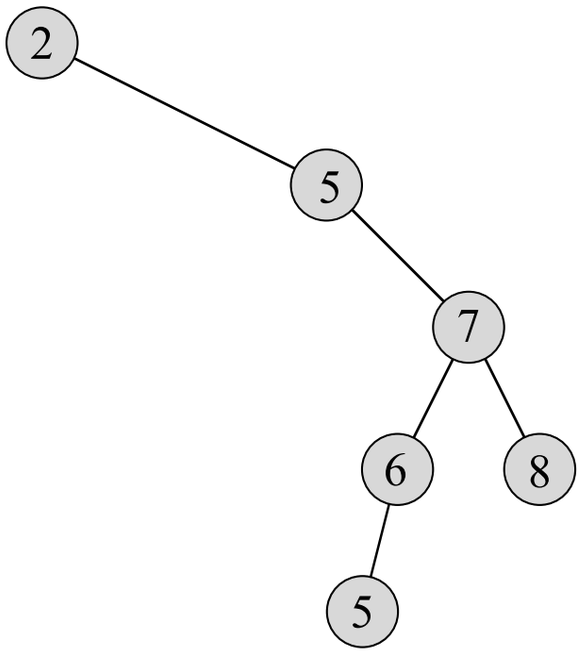
- **Worst-case** occurs when we are in Case C and Case D.
- Case A and Case B only take $\Theta(1)$

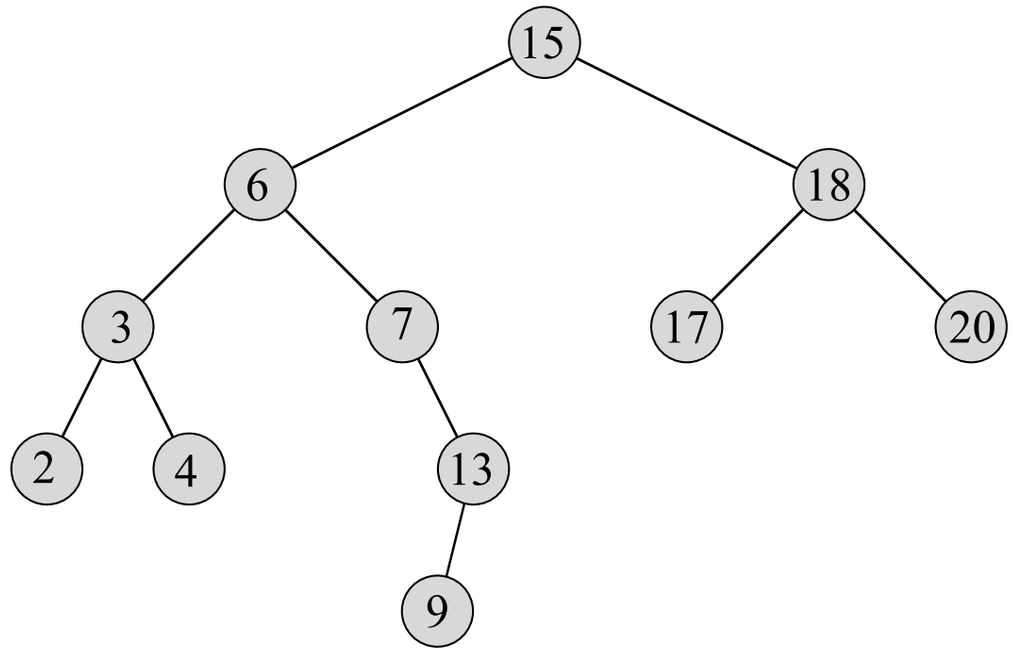
```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2        TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4        TRANSPLANT(T, z, z.left)
 5  else y = TREE-MINIMUM(z.right)
 6        if y.p ≠ z
 7              TRANSPLANT(T, y, y.right)
 8              y.right = z.right
 9              y.right.p = y
10        TRANSPLANT(T, z, y)
11        y.left = z.left
12        y.left.p = y
```

$\Theta(h)$

BST: Summary & Reflections

- BSTs are an effective data structure for dynamic sets
- **All the operations** we have seen **run in at most $\Theta(h)$ time**, where h is the height of the BST
  - **Good** when the tree is **balanced**: $h = \Theta(\lg n)$
  - **Bad** when the tree is **unbalanced**: $h = \Theta(n)$

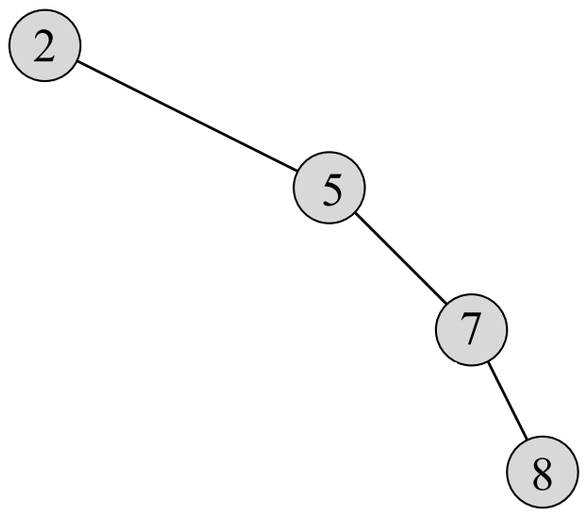




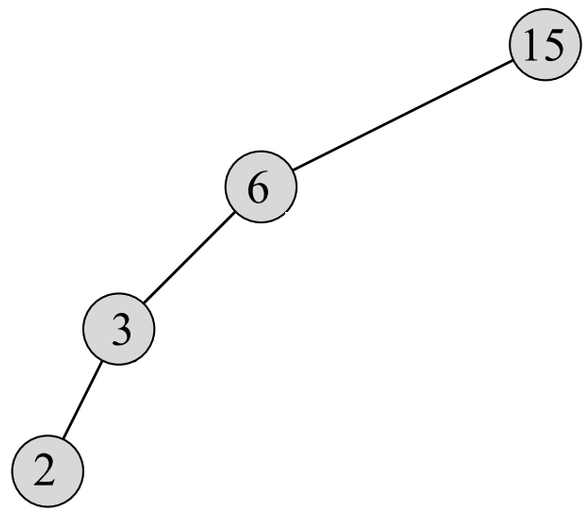
Balanced BSTs

- How easy is it to create an unbalanced tree by inserting a sequence of nodes in the tree?
  - If the sequence has increasing keys (reps. decreasing keys) the resulting tree boils down to a chain

Result of Insertion 2; 5; 7; 8



Result of Insertion 15; 6; 3; 2



Balanced BSTs

- Little is known about average height of a binary search tree when both insertion and deletion are used to create it.
- We can say something when the tree is created by insertion alone

Randomly-built BST:
n-keys BST obtained by *inserting the keys in random order* into an initially empty tree, where *each of the $n!$ permutations of the input keys are equally likely*

Theorem 12.4
The expected height of a randomly built binary search tree on n distinct keys is $O(\lg n)$.

Red-Black Trees

Red-Black Trees

- Red-back Trees **are BSTs** where each node has an additional colour attribute which can be either Red or Black

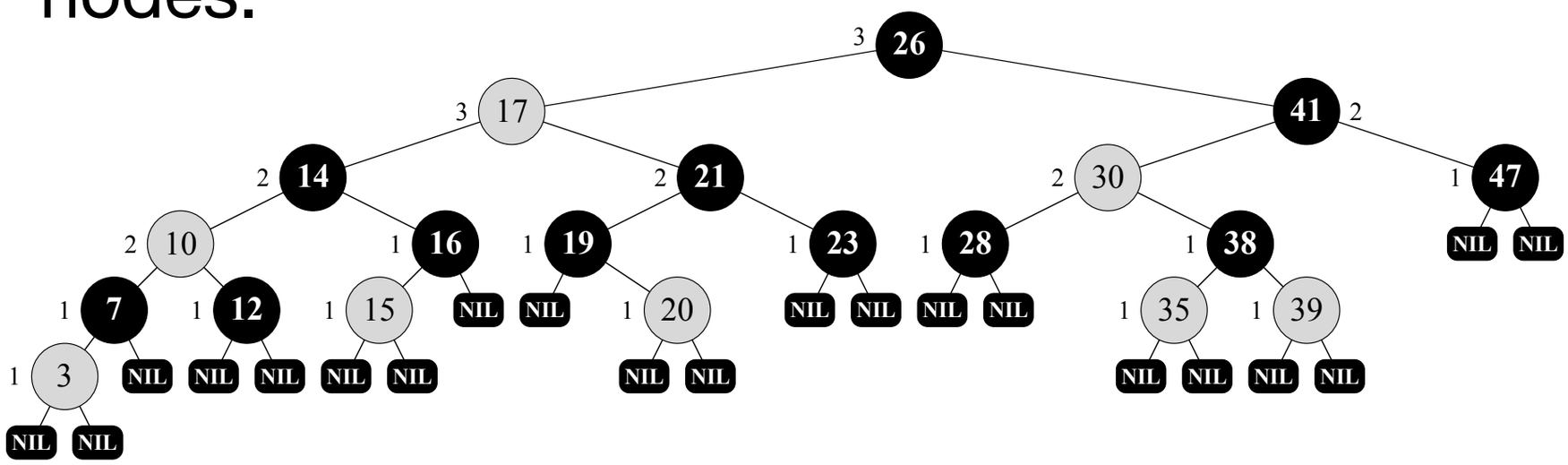
- By constraining node colours on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other

- Red-black trees are approximately balanced, that is $h = O(\lg n)$

Red-Black Trees

A red-black tree is a BST that additionally satisfies the following **red-black properties**:

1. Every node is either red or black
2. The root is black
3. Every leaf (NIL) is black
4. If a node is red, then both its children are black
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.



Red-Black Trees

Lemma 13.1
A red-black tree with n internal nodes has height at most $2\lg(n + 1)$.

Optional: read the proof of Lemma 13.1 in CLRS

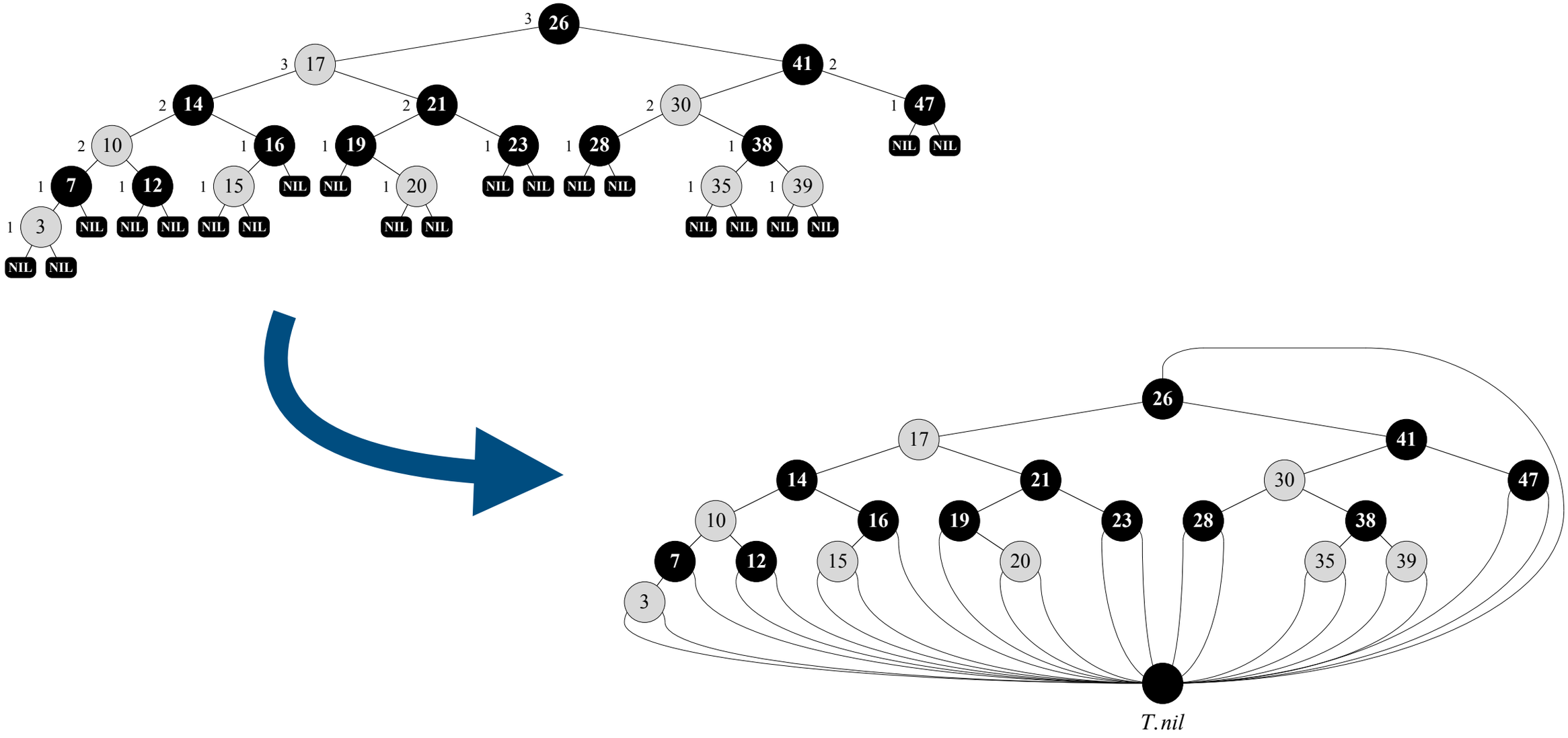
- Remember that red-black trees are special BSTs.

- Thus Lemma 13.1 implies that we can implement the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in $O(\lg n)$ time, where n is the number of elements in the tree

- Insertion and deletion require more work to preserve the red-black-tree property

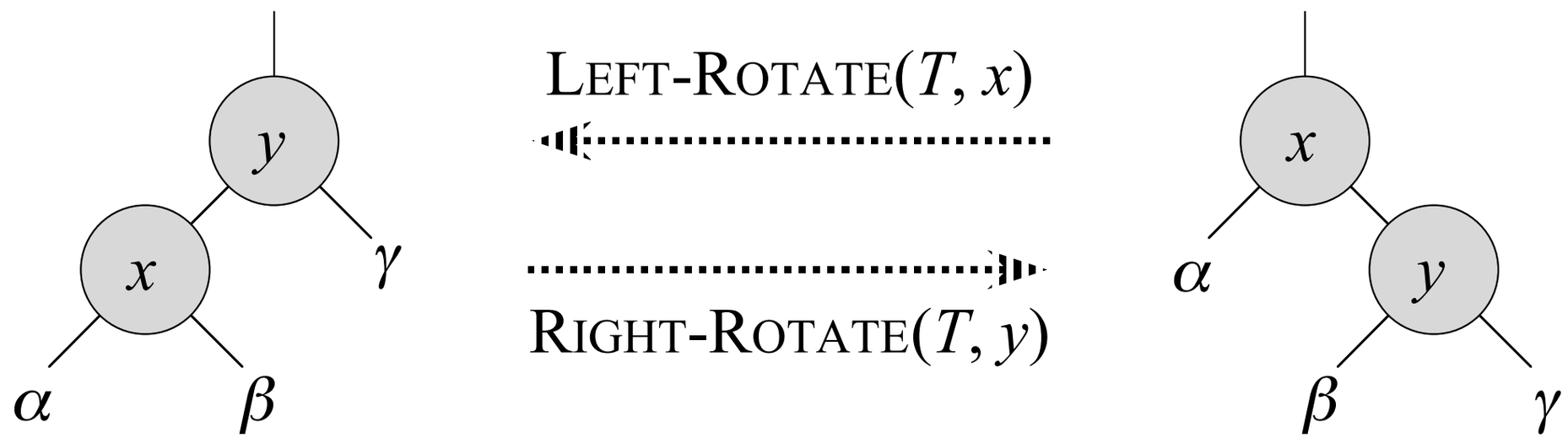
Red-Black Trees

- It will be convenient to replace Nil pointers with nil nodes
- In fact, we only need one such node which can be shared

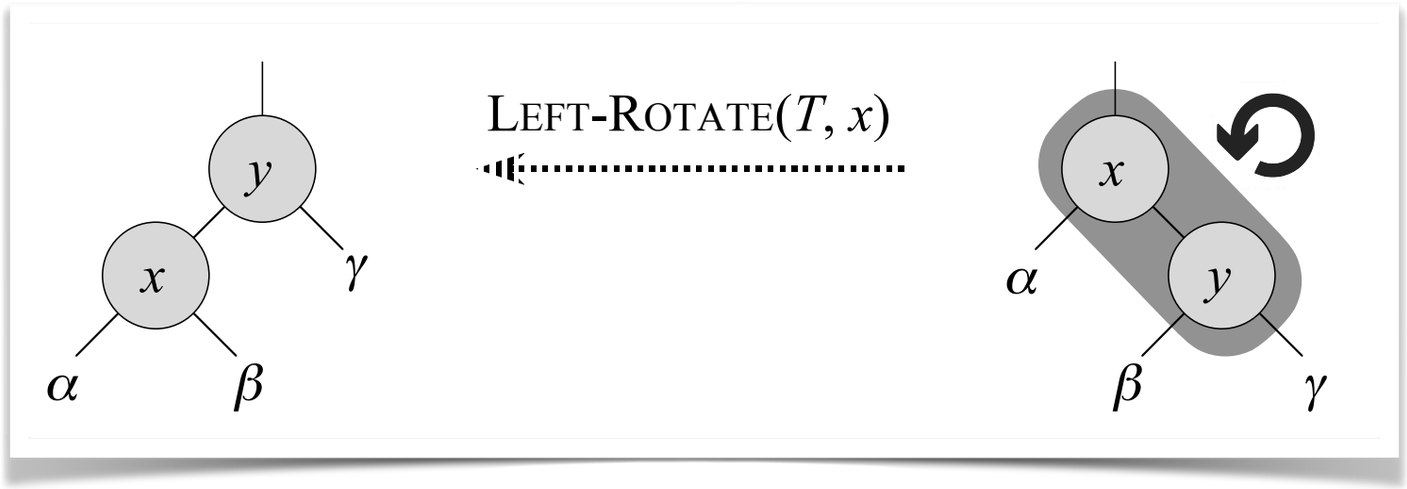


Rotations

- Insertion and deletion in red-black trees have to preserve the red-black tree property
- This is done by
  - Changing the colours of some nodes in the tree, and
  - Changing the pointer structure through rotations
- Rotation is a local operation that **preserves the BST property** (may not preserve the red-black tree property)



Left-Rotate: pseudocode




```
LEFT-ROTATE(T, x)
 1  y = x.right
 2  x.right = y.left
 3  if y.left ≠ T.nil
 4      y.left.p = x
 5  y.p = x.p
 6  if x.p == T.nil
 7      T.root = y
 8  elseif x == x.p.left
 9      x.p.left = y
10  else x.p.right = y
11  y.left = x
12  x.p = y
```

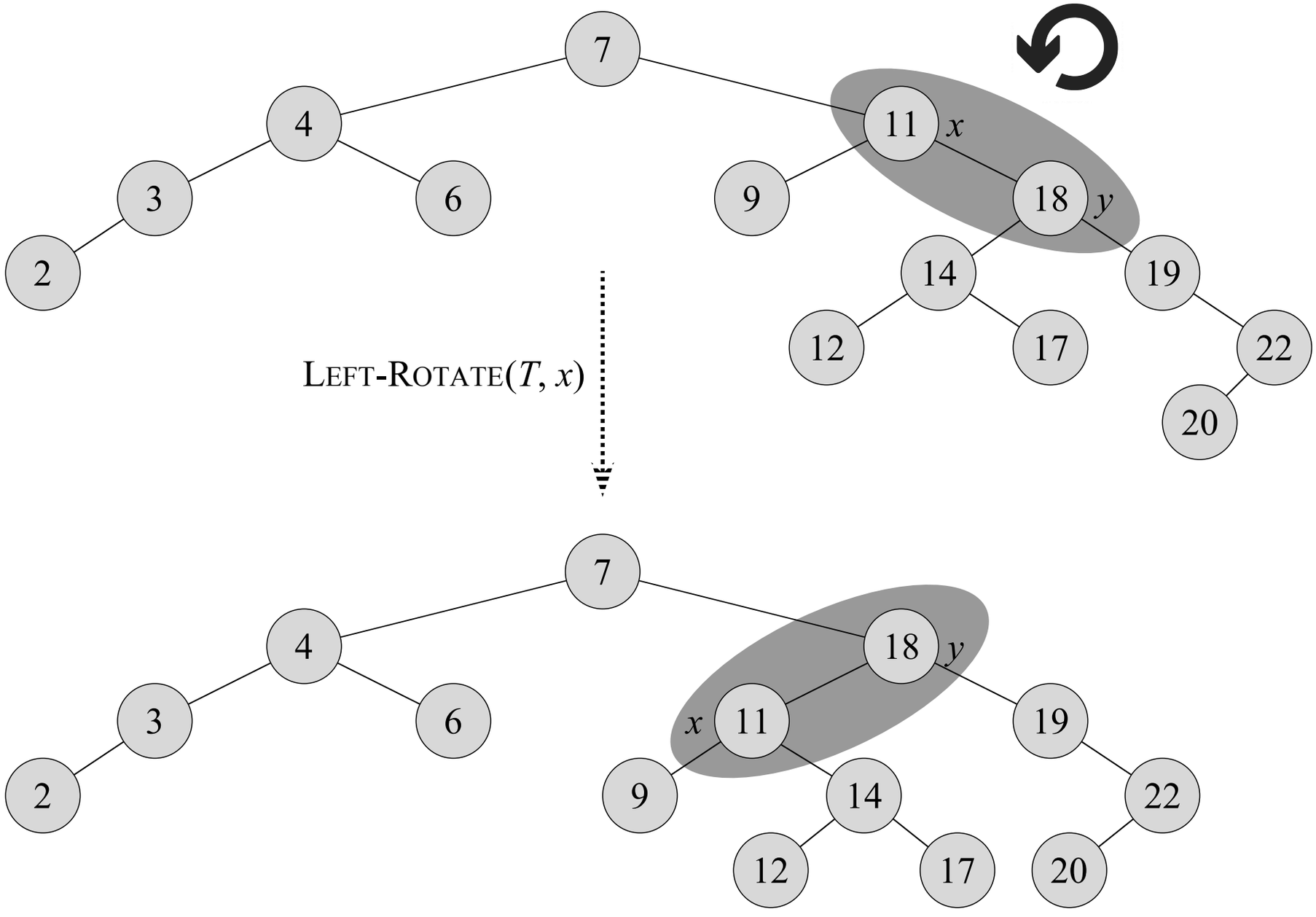
set y

Turn y's left subtree (i.e., β) into $x's$ right subtree

Link x's parent to y

Put x on $y's$ left

Left-Rotate: example



RB-Insert

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

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.p = y
 9  if y == NIL
10        T.root = z
11  elseif z.key < y.key
12        y.left = z
13  else y.right = z
```

Set z as **red** and set its children

Restore the RB property!

How to restore RB property?

To understand how RB-Insert-Fixup works we shall first determine what violations of the RB property are introduced in RB-Insert when z is inserted an coloured red

Red-black properties:

1. Every node is either red or black ✓
2. The root is black ✗
3. Every leaf (NIL) is black ✓
4. If a node is red, then both its children are black ✗
5. For each node, all simple paths from the node to descendant leaves contains the same number of back nodes. ✓

How to restore RB property?

Basic Idea

We start from the inserted element z (which is initially at the bottom of the tree) and we move up the violations of the RB property up to the root

Loop Invariant:

1. Node z is **red**
2. If $z's$ parent (i.e., $z.p$) is the root, then $z.p$ is black
3. If the tree violates any of the RB properties, then
   - Either it violates Prop. 2 (because z is the root), or
   - It violates Prop. 4 (because also z's parent is red)

RB-Insert-Fixup

RB-INSERT-FIXUP(T, z)

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

The while loop moves the violation up in the tree preserving the invariant

RB-Insert-Fixup

RB-INSERT-FIXUP(T, z)

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

The while loop moves the violation up in the tree preserving the invariant

There are two symmetric procedures respectively if z's parent is a left child or a right child

RB-Insert-Fixup

RB-INSERT-FIXUP(T, z)

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

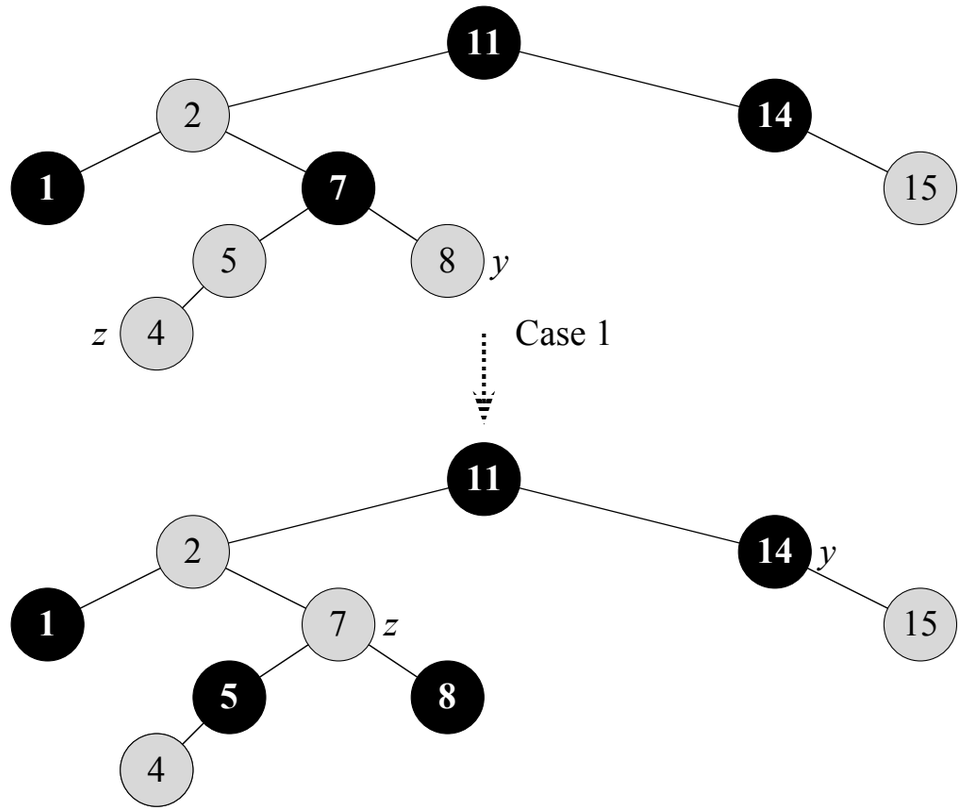
The while loop moves the violation up in the tree preserving the invariant

There are two symmetric procedures respectively if z's parent is a left child or a right child

Each are subdivided in 3 cases (where in particular Case 2 redirects to Case 3)

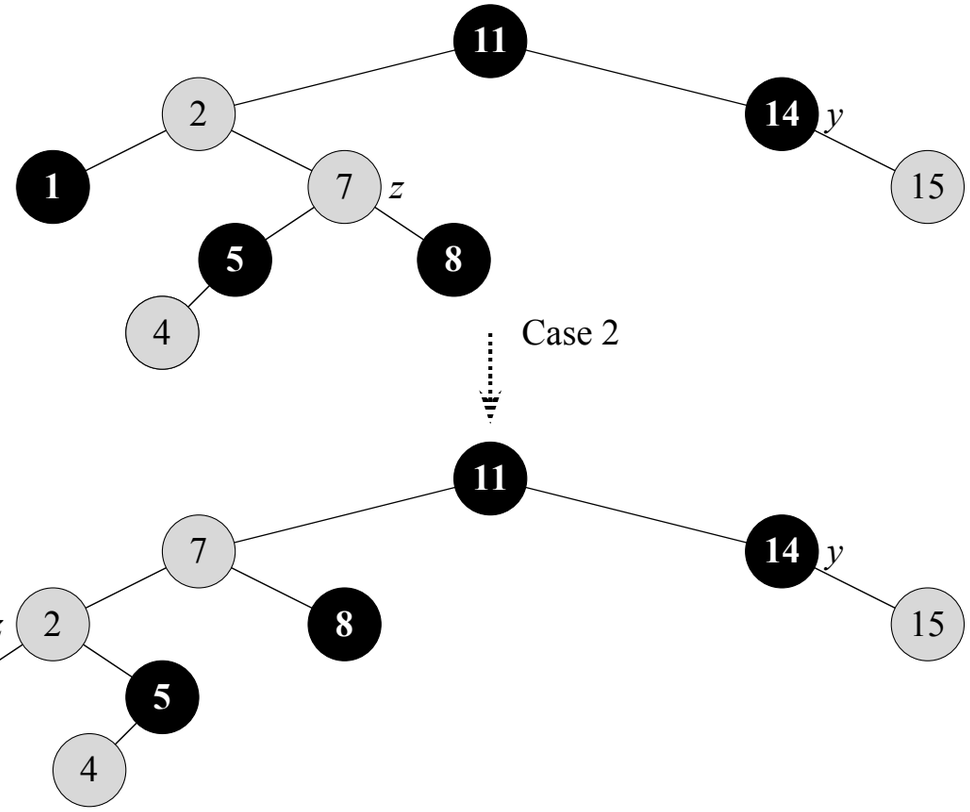
RB-Insert-Fixup: Case 1

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



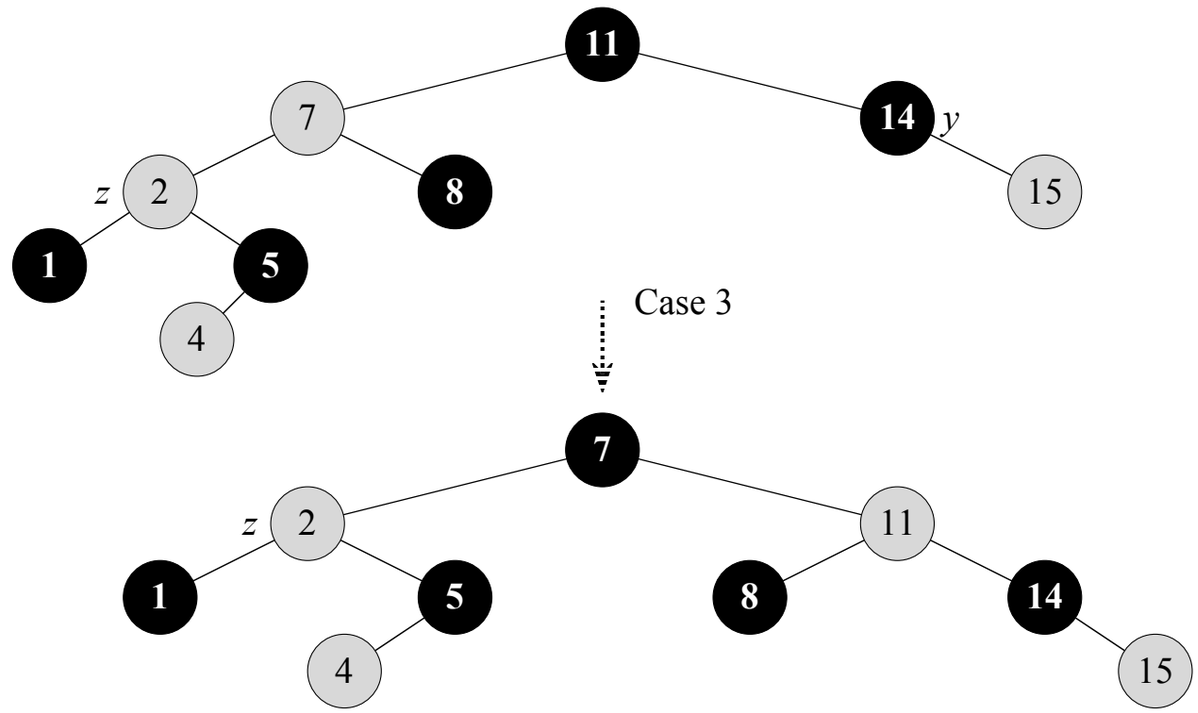
RB-Insert-Fixup: Case 2

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



RB-Insert-Fixup: Case 3

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



RB-Insert: run-time

Worst-case running time $O(\lg n)$ because $h = O(\lg n)$

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

$\Theta(h)$ as insertion in BST

$\Theta(1)$

$\Theta(h)$. The number of loops is bounded by h

RB-Delete

```
RB-DELETE(T, z)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4       x = z.right
 5       RB-TRANSPLANT(T, z, z.right)
 6  elseif z.right == T.nil
 7       x = z.left
 8       RB-TRANSPLANT(T, z, z.left)
 9  else y = TREE-MINIMUM(z.right)
10       y-original-color = y.color
11       x = y.right
12       if y.p == z
13            x.p = y
14       else RB-TRANSPLANT(T, y, y.right)
15            y.right = z.right
16            y.right.p = y
17       RB-TRANSPLANT(T, z, y)
18       y.left = z.left
19       y.left.p = y
20       y.color = z.color
21  if y-original-color == BLACK
22       RB-DELETE-FIXUP(T, x)
```

```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2       TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4       TRANSPLANT(T, z, z.left)
 5  else y = TREE-MINIMUM(z.right)
 6       if y.p ≠ z
 7            TRANSPLANT(T, y, y.right)
 8            y.right = z.right
 9            y.right.p = y
10       TRANSPLANT(T, z, y)
11       y.left = z.left
12       y.left.p = y
```

+

Some additional code which keeps track of two nodes y and x

Restore the RB property!

RB-Delete

```
RB-DELETE(T, z)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4      x = z.right
 5      RB-TRANSPLANT(T, z, z.right)
 6  elseif z.right == T.nil
 7      x = z.left
 8      RB-TRANSPLANT(T, z, z.left)
 9  else y = TREE-MINIMUM(z.right)
10      y-original-color = y.color
11      x = y.right
12      if y.p == z
13          x.p = y
14      else RB-TRANSPLANT(T, y, y.right)
15          y.right = z.right
16          y.right.p = y
17      RB-TRANSPLANT(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if y-original-color == BLACK
22      RB-DELETE-FIXUP(T, x)
```

If z's left (or right) child is Nil

- $y = z$ is the node removed, and
- x is the node that takes y's old position in T

If z has both children Nil

- y is the node that takes z's position, and
- x is the node that takes y's old position in T

- Here y has been removed or has taken z's place and colour
- If y's original colour was black then some RB property may be violated

How to restore RB property?

To understand how RB-Delete-Fixup works we shall first determine what violations of the RB property are introduced in RB-Delete when y is deleted (i.e., $y = z$) or moved in $z's$ original position.

If y's original colour was **red**, then no RB property is violated

Red-black properties:

1. Every node is either red or black ✓
2. The root is black ✓
3. Every leaf (NIL) is black ✓
4. If a node is red, then both its children are black ✓
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes. ✓

y was not the root

- y takes z's place in T, along with z's colour, we cannot have two adjacent red nodes at y's new position in the tree.
- In addition, if y was not z's right child, then y's original right child x replaces y in T. Since y was red, then x must be black.

No black heights have changed

How to restore RB property?

To understand how RB-Delete-Fixup works we shall first determine what violations of the RB property are introduced in RB-Delete when y is deleted (i.e., $y = z$) or moved in $z's$ original position.

If y's original colour was **Black**

Red-black properties:

1. Every node is either red or black ✓
2. The root is black ✗
3. Every leaf (NIL) is black ✓
4. If a node is red, then both its children are black ✗
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes. ✗

If y was the root, x may be red and take its position

If x and x's parent are both red after x has replaced y's original position

Moving y makes any simple path previously contain y to have one fewer black node

How to restore RB property?

To understand how RB-Delete-Fixup works we shall first determine what violations of the RB property are introduced in RB-Delete when y is deleted (i.e., $y = z$) or moved in $z's$ original position.

If y's original colour was **Black**

Red-black properties:

1. Every node is either red or black ✗
2. The root is black ✗
3. Every leaf (NIL) is black ✓
4. If a node is red, then both its children are black ✗
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes. ✓

We can fictionally 'fix' Prop. 5 by counting one more black token at x (who replaced y's position). Hence, x can be **'doubly black'** or **'red-and-black'**.

RB-Delete-Fixup

Basic Idea

We start from x and we **move up the extra black token** up the tree until:

- x points to a red-and-black node, in which case we colour x just back
- x points to the root, in which case we colour x just black; or
- Having performed suitable rotations and recolouring, the loop exits

RB-Delete-Fixup

RB-DELETE-FIXUP(T, x)

```
1  while x ≠ T.root and x.color == BLACK
2      if x == x.p.left
3          w = x.p.right
4          if w.color == RED
5              w.color = BLACK
6              x.p.color = RED
7              LEFT-ROTATE(T, x.p)
8              w = x.p.right
9          if w.left.color == BLACK and w.right.color == BLACK
10             w.color = RED
11             x = x.p
12         else if w.right.color == BLACK
13                 w.left.color = BLACK
14                 w.color = RED
15                 RIGHT-ROTATE(T, w)
16                 w = x.p.right
17             w.color = x.p.color
18             x.p.color = BLACK
19             w.right.color = BLACK
20             LEFT-ROTATE(T, x.p)
21             x = T.root
22     else (same as then clause with "right" and "left" exchanged)
23 x.color = BLACK
```

The while loop moves the extra black token (pointed by x) up in the tree

RB-Delete-Fixup

RB-DELETE-FIXUP(T, x)

```
1  while x ≠ T.root and x.color == BLACK
2      if x == x.p.left
3          w = x.p.right
4          if w.color == RED
5              w.color = BLACK
6              x.p.color = RED
7              LEFT-ROTATE(T, x.p)
8              w = x.p.right
9          if w.left.color == BLACK and w.right.color == BLACK
10             w.color = RED
11             x = x.p
12         else if w.right.color == BLACK
13                 w.left.color = BLACK
14                 w.color = RED
15                 RIGHT-ROTATE(T, w)
16                 w = x.p.right
17             w.color = x.p.color
18             x.p.color = BLACK
19             w.right.color = BLACK
20             LEFT-ROTATE(T, x.p)
21             x = T.root
22     else (same as then clause with "right" and "left" exchanged)
23 x.color = BLACK
```

The while loop moves the extra black token (pointed by x) up in the tree

There are two symmetric procedures respectively if x is a left child or a right child

RB-Delete-Fixup

RB-DELETE-FIXUP(T, x)

```
1  while x ≠ T.root and x.color == BLACK
2      if x == x.p.left
3          w = x.p.right
4          if w.color == RED
5              w.color = BLACK                          ┐
6              x.p.color = RED                          │
7              LEFT-ROTATE(T, x.p)                      │ Case 1
8              w = x.p.right                            ┘
9          if w.left.color == BLACK and w.right.color == BLACK
10             w.color = RED                            ┐
11             x = x.p                                  │ Case 2
12         else if w.right.color == BLACK               ┘
13             w.left.color = BLACK                     ┐
14             w.color = RED                            │
15             RIGHT-ROTATE(T, w)                       │ Case 3
16             w = x.p.right                            ┘
17         w.color = x.p.color                          ┐
18         x.p.color = BLACK                            │
19         w.right.color = BLACK                        │ Case 4
20         LEFT-ROTATE(T, x.p)                          │
21         x = T.root                                   ┘
22     else (same as then clause with "right" and "left" exchanged)
23 x.color = BLACK
```

The while loop moves the extra black token (pointed by x) up in the tree

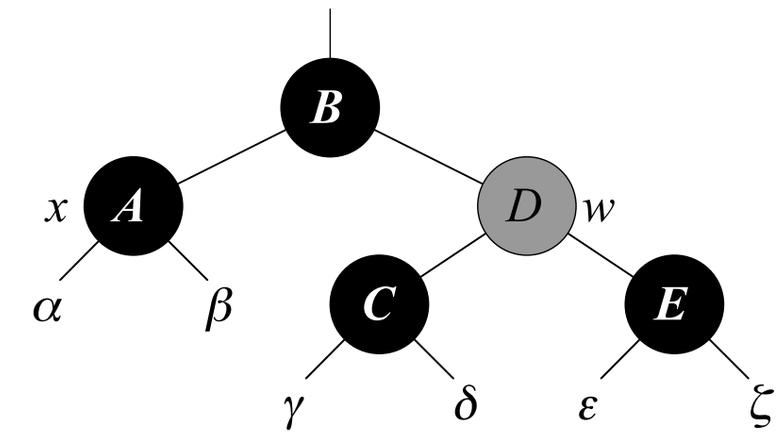
There are two symmetric procedures respectively if x is a left child or a right child

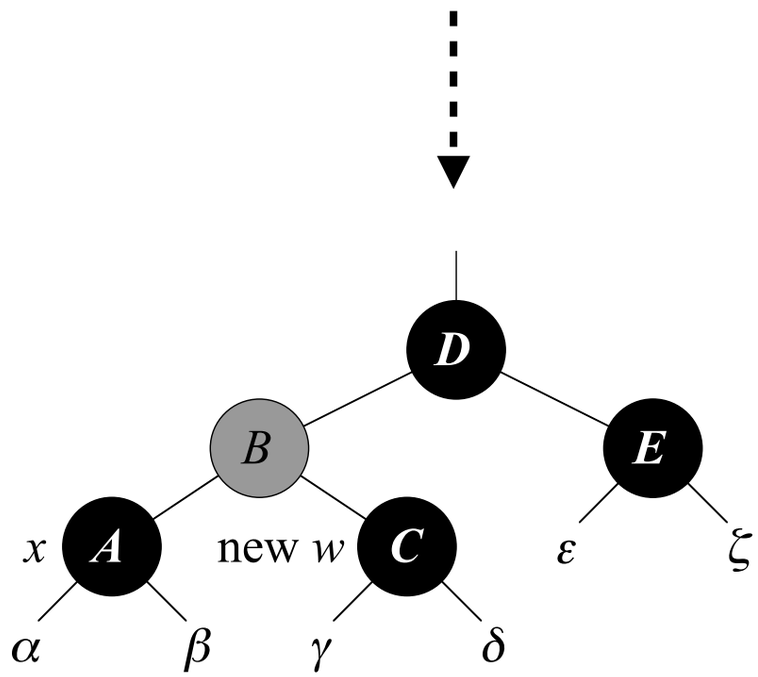
Each divided in 4 cases

RB-Delete-Fixup: Case 1

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left
 3          w = x.p.right
 4          if w.color == RED
 5              w.color = BLACK
 6              x.p.color = RED
 7              LEFT-ROTATE(T, x.p)
 8              w = x.p.right
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED
11              x = x.p
12          else if w.right.color == BLACK
13                  w.left.color = BLACK
14                  w.color = RED
15                  RIGHT-ROTATE(T, w)
16                  w = x.p.right
17              w.color = x.p.color
18              x.p.color = BLACK
19              w.right.color = BLACK
20              LEFT-ROTATE(T, x.p)
21              x = T.root
22      else (same as then clause with "right" and "left" exchanged)
23  x.color = BLACK
```

x's sibling w is red

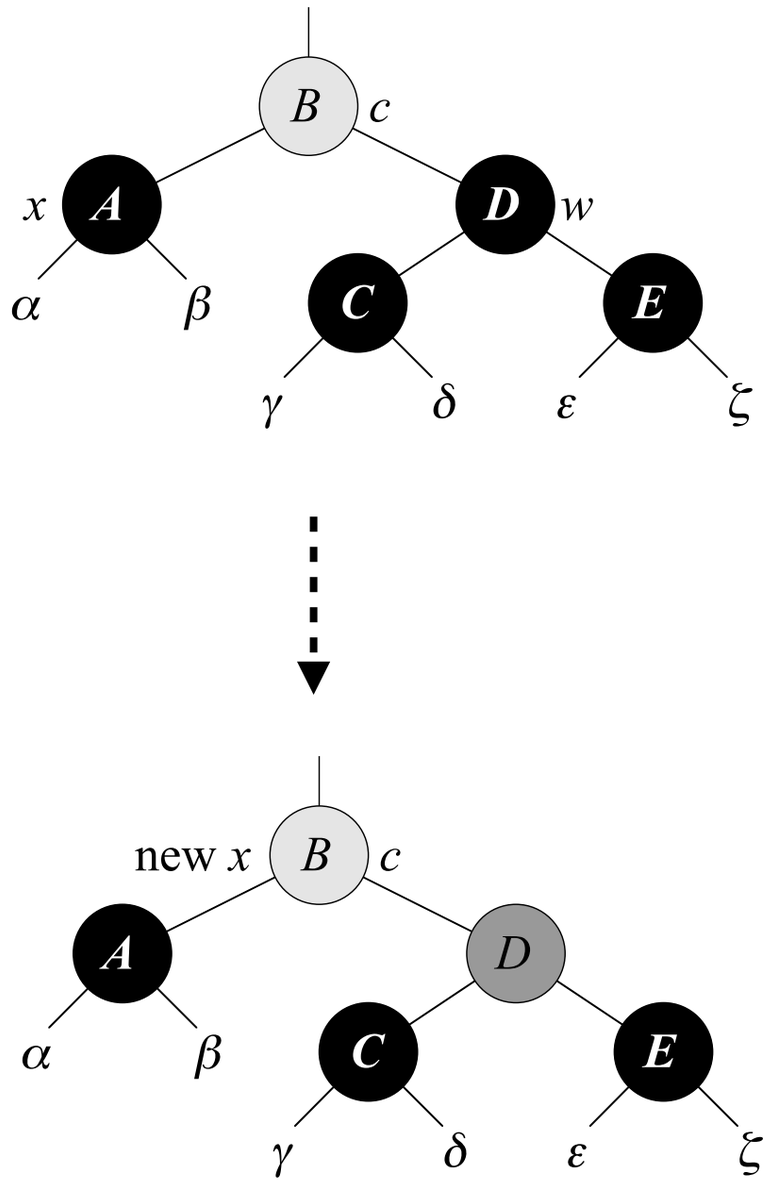




RB-Delete-Fixup: Case 2

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left
 3          w = x.p.right
 4          if w.color == RED
 5              w.color = BLACK
 6              x.p.color = RED
 7              LEFT-ROTATE(T, x.p)
 8              w = x.p.right
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED
11              x = x.p
12          else if w.right.color == BLACK
13                  w.left.color = BLACK
14                  w.color = RED
15                  RIGHT-ROTATE(T, w)
16                  w = x.p.right
17              w.color = x.p.color
18              x.p.color = BLACK
19              w.right.color = BLACK
20              LEFT-ROTATE(T, x.p)
21              x = T.root
22      else (same as then clause with "right" and "left" exchanged)
23  x.color = BLACK
```

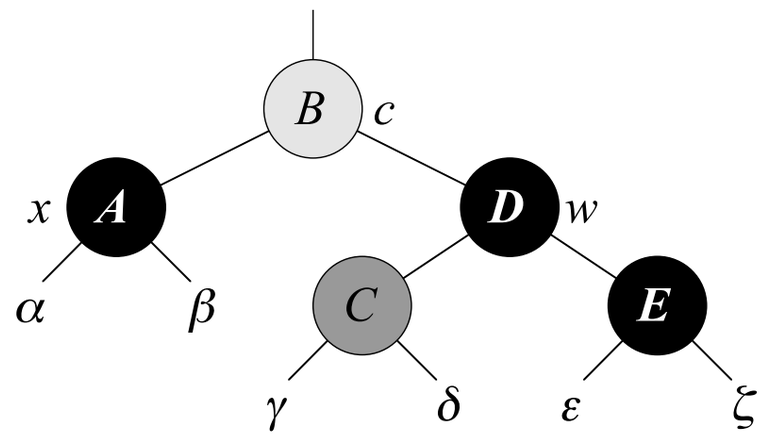
x's sibling w is black, and both of w's children are black

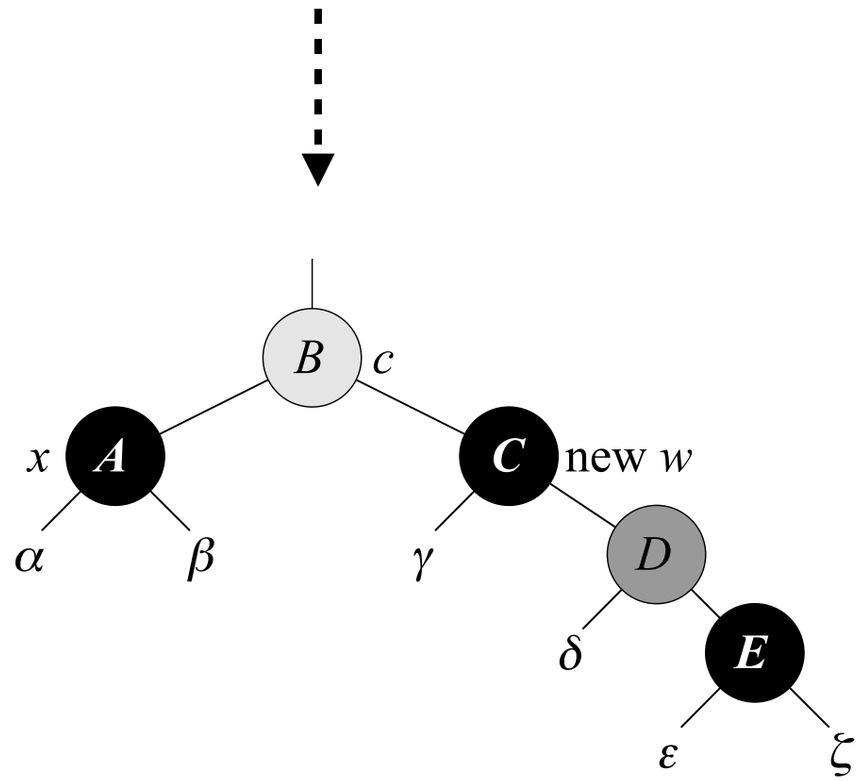


RB-Delete-Fixup: Case 3

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left
 3          w = x.p.right
 4          if w.color == RED
 5              w.color = BLACK
 6              x.p.color = RED
 7              LEFT-ROTATE(T, x.p)
 8              w = x.p.right
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED
11              x = x.p
12          else if w.right.color == BLACK
13              w.left.color = BLACK
14              w.color = RED
15              RIGHT-ROTATE(T, w)
16              w = x.p.right
17          w.color = x.p.color
18          x.p.color = BLACK
19          w.right.color = BLACK
20          LEFT-ROTATE(T, x.p)
21          x = T.root
22      else (same as then clause with "right" and "left" exchanged)
23  x.color = BLACK
```

x's sibling w is black, w's left child is red, and w's right child is black

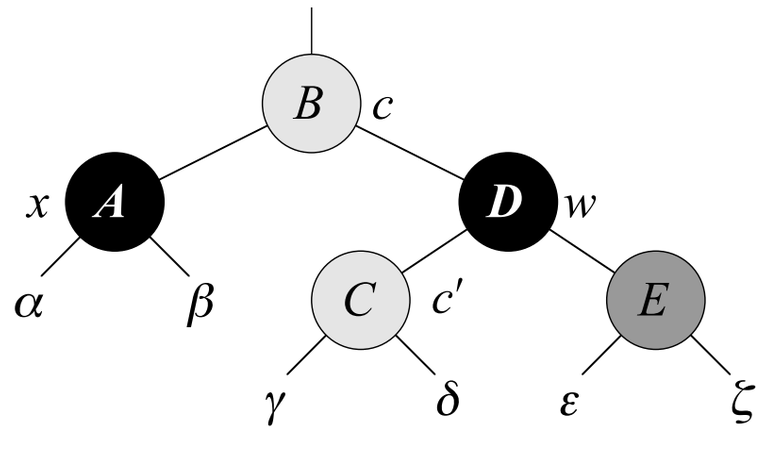


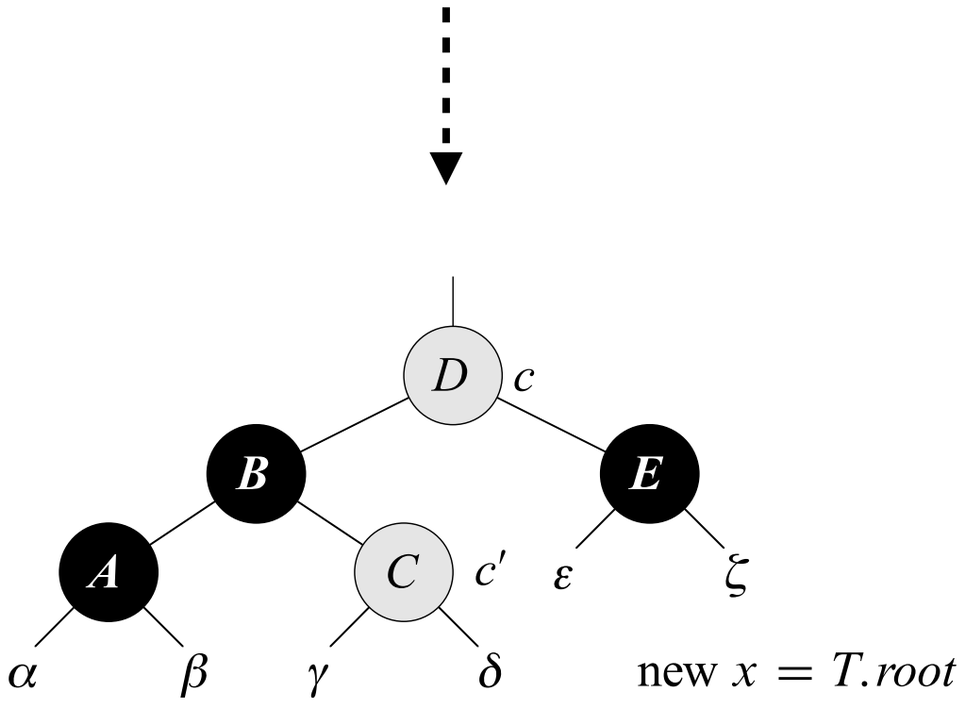


RB-Delete-Fixup: Case 4

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left
 3          w = x.p.right
 4          if w.color == RED
 5              w.color = BLACK
 6              x.p.color = RED
 7              LEFT-ROTATE(T, x.p)
 8              w = x.p.right
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED
11              x = x.p
12          else if w.right.color == BLACK
13                  w.left.color = BLACK
14                  w.color = RED
15                  RIGHT-ROTATE(T, w)
16                  w = x.p.right
17              w.color = x.p.color
18              x.p.color = BLACK
19              w.right.color = BLACK
20              LEFT-ROTATE(T, x.p)
21              x = T.root
22      else (same as then clause with "right" and "left" exchanged)
23  x.color = BLACK
```

x's sibling w is black, and
w's right child is red





RB-Delete: run-time

RB-DELETE(T, z)

```
1   y = z
2   y-original-color = y.color
3   if z.left == T.nil
4       x = z.right
5       RB-TRANSPLANT(T, z, z.right)
6   elseif z.right == T.nil
7       x = z.left
8       RB-TRANSPLANT(T, z, z.left)
9   else y = TREE-MINIMUM(z.right)
10      y-original-color = y.color
11      x = y.right
12      if y.p == z
13          x.p = y
14      else RB-TRANSPLANT(T, y, y.right)
15          y.right = z.right
16          y.right.p = y
17      RB-TRANSPLANT(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if y-original-color == BLACK
22      RB-DELETE-FIXUP(T, x)
```

$\Theta(h)$ as deletion in BST
+
$\Theta(1)$

$\Theta(h)$. The number of loops is bounded by h

RB Tres: Summary & Reflections

- As BSTs, Red-black tree are an effective data structure for dynamic sets
- They have in particular $h = O(\lg n)$ where h is the height of the BST
- **All the operations** we have seen **run in at most** $O(\lg n)$ **time**, where n is the number of elements in the tree
- Insertion and Deletion require 2 traverse of the tree:
  - One as for BST insertion and Deletion
  - One for fixing the RB property